第57章 WPF 3.0

本章内容：

- 不同类型的工作流：顺序工作流和状态机工作流
- 内置活动
- 如何创建自定义活动
- 工作流服务
- 与 WCF 的集成
- 驻留工作流
- 迁移到 Workflow Foundation 4 的提示

本章将概述 Windows Workflow Foundation 3.0(本章称之为 WF)，它提供一个模型，在该模型中，可以使用一组构建块(称为活动)定义和执行进程。WF 还提供了一个设计器，在默认情况下，该设计器驻留在 Visual Studio 中，允许将工具箱中的活动拖放到设计界面上，创建一个工作流模板。

创建一个 WorkflowInstance，运行该实例，就可以执行这个模板。执行工作流的代码称为 WorkflowRuntime，该对象也可以驻留许多服务，正在运行的工作流可以访问它们。在任意时刻，均有几个工作流实例在执行，运行库负责调度这些实例，保存和还原状态，它还可以记录每个工作流实例的执行情况。

工作流由许多活动构成，这些活动由运行库执行。活动可以是发送电子邮件、更新数据库中的一行，或在后端系统上执行一个事务。有许多内置活动，它们用于一般性的工作，也可以创建自己的自定义活动，根据需要将它们放在工作流中。

注意 Visual Studio 2010 有 Windows Workflow 的一个新版本，它不后向兼容。本章介绍工作流的老版本。对于新项目，建议使用 WF 4 版本，更多信息参见第 44 章。本章末尾还提供了一些升级到 WF 4 的建议。

本章从一个规范的例子 Hello World 开始，每个人在面对一种新技术时都要使用这个例子，并描述在开发计算机上使工作流运行所需要做的工作。

57.1 Hello World 示例

Visual Studio 2010 包含对创建工作流的内置支持。打开 New Project 对话框，会看到一个工作流

项目类型列表，如图 57-1 所示。

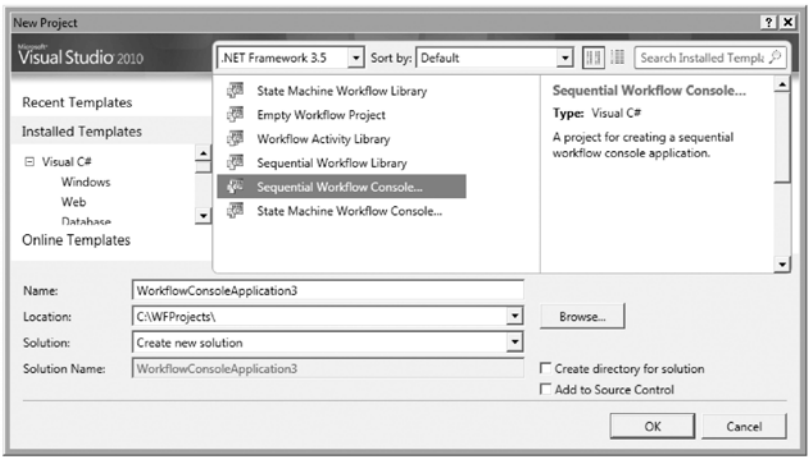

图 57-1

从可用的模板中选择 Sequential Workflow Console Application(它会创建一个驻留工作流运行库的控制台应用程序)和默认的工作流，以后要在该工作流上拖放活动。

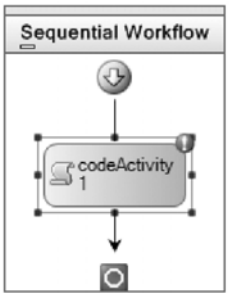

图 57-2

接着，把 Code 活动从工具箱拖放到设计界面上，这样就会得到如图 57-2 所示的工作流。

该活动右上角的感叹号标志符号表示，没有定义该活动的强制属性，这里它是 ExecuteCode 属性，它指定活动执行时调用的方法。57.3.1 节将学习如何把自己的属性标记为强制属性。如果双击 Code 活动，就在代码隐藏类中创建一个方法，该方法使用 Console.WriteLine 输出 Hello World 字符串，如下面的代码所示：

```
private void codeActivity1_ExecuteCode(object sender, EventArgs e)
{
   Console.WriteLine("Hello World");
}
```

如果构建并运行程序，就会在控制台上看到输出的文本。程序执行时，创建了一个 WorkflowRuntime 类型的实例，然后构建一个工作流实例，并执行它。在执行 Code 活动时，它调用已定义的方法，和将字符串输出到控制台上的方法。57.5 节将详细描述如何驻留运行库。上述示例的代码在 01 HelloWorkflowWorld 文件夹中。

57.2 活动

工作流中的内容是一个活动——该活动位于工作流中，其类型比较特殊，它通常允许在其中定义其他活动，这称为复合活动，本章的后面将介绍其他复合活动。活动是一个最终派生自 Activity 类的类。

Activity 类定义许多可重写的方法，其中最重要的是 Execute()方法，如下面的代码段所示：

```
protected override ActivityExecutionStatus Execute
  ( ActivityExecutionContext executionContext )
{
   return ActivityExecutionStatus.Closed;
}
```

在运行库调度活动以便执行时，最终会调用 Execute()方法，在该方法中，可以编写自定义代码，提供活动的行为。在上一节的简单示例中，当工作流运行库在 CodeActivity 上调用 Execute()方法时，这个方法的实现代码就会执行在代码隐藏类中定义的方法，在控制台上显示该消息。

Execute()方法需要一个 ActivityExecuteContext 类型的上下文参数，本章的后面将探讨这个参数。该方法的返回值是 ActivityExecutionStatus 类型，该返回值由运行库用于确定活动是已成功完成、仍在处理，还是处于其他几个状态中的一个状态，这几个状态可以向工作流运行库描述活动所处的状态。从这个方法中返回 ActivityExecutionStatus.Closed，表示活动已完成其工作，可以释放它。

WF 提供了许多标准活动，下面几节就介绍其中一些活动的例子，并说明使用这些活动的场合。活动的命名约定是在名称的后面追加 Activity。例如，图 57-2 中的代码活动就由 CodeActivity 类定义。

所有标准活动都在 System.Workflow.Activities 名称空间中定义，该名称空间位于 System.Workflow.Activities.dll 程序集中。还有另外两个程序集——System.Workflow.ComponentModel.dll 和 System. Workflow.Runtime.dll 也是 WF 的组成部分。

57.2.1　IfElseActivity

顾名思义，这个活动的操作类似于 C#中的 If-Else 语句。

当将一个 IfElseActivity 拖放到设计界面上时，会看到如图 57-3 所示的活动。IfElseActivity 是一个复合活动，它构建两个分支(这两个分支的类型也是活动，这里是 IfElseBranchActivity)。每个分支也都是派生自 SeqenceActivity 的复合活动，这个类自上而下执行每个活动。设计器添加了 Drop Activities Here 文本，指出可以把子活动添加到什么地方。

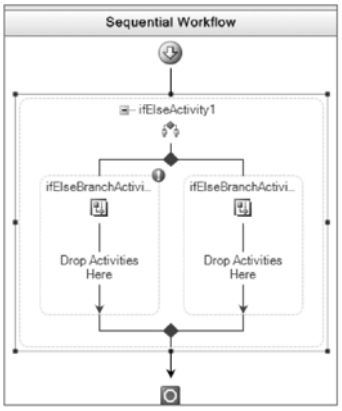

图　57-3

如图 57-3 所示，第一个分支包含一个符号，指定需要定义 Condition 属性。条件派生自 ActivityCondition，并用于确定是否执行该分支。

```
protected override ActivityExecutionStatus Execute
  ( ActivityExecutionContext executionContext )
{
   return ActivityExecutionStatus.Closed;
}
```

在执行 IfElseActivity 时，将判断第一个分支的条件，如果该条件等于 true，就执行该分支。如果条件等于 false，IfElseActivity 就尝试执行下一个分支，依此类推，直到活动中的最后一个分支为止。注意，IfElseActivity 可以有任意多个分支，每个分支都有自己的条件。最后一个分支不能有条件，因为它相当于 If-Else 语句中的 else 部分。要添加一个新分支，可以显示活动的上下文菜单，从

菜单中选择 Add Branch 命令——也可以从 Visual Studio 的 Workflow 菜单中选择它。在添加分支时，每个分支都有一个强制的条件，但最后一个分支例外。

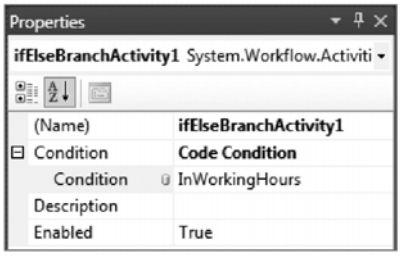

图 57-4

WF 定义两个标准的条件类型——CodeCondition 和 RuleConditionReference。CodeCondition 类在代码隐藏类上执行一个方法，它返回 true 或 false。要创建 CodeCondition，可以显示 IfElseActivity 的属性表，将条件设置为 Code Condition，再为要执行的代码输入一个名称，如图 57-4 所示。

将方法名输入属性网格时，设计器会在代码隐藏类上构建一个方法，如下面的代码段所示：

```
private void InWorkingHours(object sender, ConditionalEventArgs e)
{
   int hour = DateTime.Now.Hour;
   e.Result = ((hour >= 9) && (hour <= 17));
}
```

如果当前时间在 9 am 和 5 pm 之间，上面的代码就将所传递的 ConditionalEventArgs 的 Result 属性设置为 true。可以在代码中定义条件，如上面的代码所示，另一种方法是根据以类似方式判断的 Rule 定义条件。Workflow 设计器包含一个规则编辑器，它可以用于声明条件和语句(类似于上面的 If-Else 语句)。这些规则在运行时根据工作流的当前状态进行判断。

57.2.2 ParallelActivity

这个活动可以定义并行执行的一组活动，或者以伪并行的方式执行的一组活动。当工作流运行库调度一个活动时，它在一个线程中调度活动。这个线程先执行第一个活动，再执行第二个活动，直到完成所有活动为止(或者某个活动在等待某种形式的输入为止)。在执行 ParallelActivity 时，它会迭代每个分支，依次调度执行每个分支。工作流运行库为每个工作流实例维护一个已调度的活动队列，一般以 FIFO(先进先出)的方式执行它们。

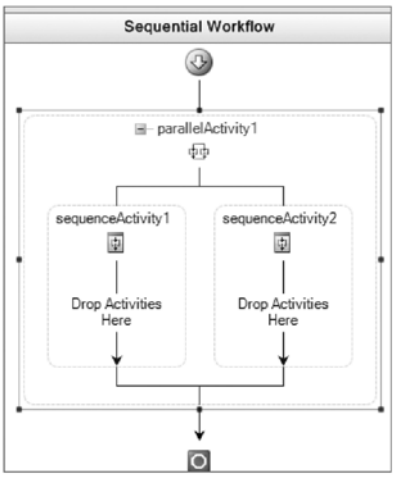

图 57-5

假定有如图 57-5 所示的一个 ParallelActivity，它调度执行 sequenceActivity1 和 sequenceActivity2。SequenceActivity 类型的工作方式是先用运行库调度执行第一个活动，这个活动执行完毕后，就调度第二个活动。这个调度/等待完成方法会遍历系列中的所有子活动，直到所有子活动都执行完毕后，序列活动才完成。

如果 SequenceActivity 一次调度执行一个活动，就表示 WorkflowRuntime 维护的队列会用可调度执行的活动连续不断地更新。假定有一个并行活动 P1，它包含两个系列(S1 和 S2)，每个系列都有两个代码活动(C1 和 C2)，这会在调度程序队列中生成如表 57-1 所示的项。

表　57-1

工作流队列	初始队列在队列中没有活动
P1	在工作流运行时并行执行
S1，S2	执行 P1 时，添加到队列中
S2，S1.C1	S1 执行，并将 S1.C1 添加到队列中
S1.C1，S2.C1	S2 执行，并将 S2.C1 添加到队列中
S2.C1，S1.C2	S1.C1 完成，所以 S1.C2 进入队列中
S1.C2，S2.C2	S2.C1 完成，所以 S2.C2 进入队列中
S2.C2	队列中的最后一项

这里，队列处理第一项(并行活动 P1)，这将序列活动 S1 和 S2 添加到工作流队列中。在执行序列活动 S1 时，它会将其第一个子活动(S1.C1)放在队列的尾部，调度并完成这个活动后，就把第二个子活动添加到队列中。

从上面的例子可以看出，ParallelActivity 的执行并非真正是并行的，它实际上在两个系列分支之间交叉执行。由此可以推断，最好的情况是活动的执行时间最短，因为每个工作流都只有一个线程为调度程序队列服务，所以运行时间较长的活动会妨碍队列中其他活动的执行。但是，因为活动的执行时间常常是任意的，所以必须采用某种方式将活动标记为“运行时间较长”，以便其他活动有机会执行。为此，可以从 Execute()方法中返回 ActivityExecutionStatus. Executing，在活动结束时让运行库知道，以后还要调用它。这种活动的一个例子是 DelayActivity。

57.2.3　CallExternalMethodActivity

工作流一般需要调用工作流外部的方法，这个活动可以定义一个接口和在该接口上调用的方法。WorkflowRuntime 维护一个服务列表(其键为一个 System.Type 值)，使用传递给 Execute()方法的 ActivityExecutionContext 参数可以访问该服务列表。

可以定义自己的服务，用于添加到这个集合中，再从自己的活动中访问这些服务。例如，可以构建一个数据访问层，作为一个服务接口提供，再为 SQL Server 和 Oracle 提供该服务的不同实现代码。因为活动只调用接口方法，所以从 SQL Server 到 Oracle 的切换对活动是不透明的。

将一个 CallExternalMethodActivity 添加到工作流中后，就定义两个强制属性(InterfaceType 和 MethodName)。接口类型定义在执行活动时运行库要使用的服务，方法名指定要调用该接口的哪个方法。

在执行这个活动时，它将查询该服务类型的执行上下文，查找有指定接口的服务，然后调用该接口上的相应方法。也可以从工作流中给方法传递参数，对应内容将在 57.4.5 节中讨论。

57.2.4　DelayActivity

业务流程常常需要等待一段时间才能完成——考虑使用工作流进行费用申请的过程。工作流给直接经理发送一封电子邮件，要求他批准某个费用申请。之后工作流进入等待状态，等待经理批准(或者不批准)，这里最好定义一个超时期限，这样如果在 1 天的时间内没有返回响应，费用申请就路由给命令链中的下一个经理。

DelayActivity 可以实现这个情形的一部分(另一部分是下面定义的 ListenActivity)，其任务是等待指定的时间，之后继续执行工作流。定义延迟的持续时间有两种方式：可以将延迟的 TimeoutDuration 属性设置为一个字符串，如“1.00:00:00”(1 天，没有指定小时、分钟和秒)；也可以提供一个方法，在执行活动时，调用该方法，从代码中将延迟的持续时间设置为一个值。为此，需要为延迟活动的 InitializeTimeoutDuration 属性定义一个值。这会在代码隐藏中创建一个方法，如下面的代码段所示：

```
private void DefineTimeout(object sender, EventArgs e)
{
   DelayActivity delay = sender as DelayActivity;

   if (null != delay)
   {
      delay.TimeoutDuration = new TimeSpan(1, 0, 0, 0);
   }
}
```

这里的 DefineTimeout()方法将发送者强制转换为一个 DelayActivity，然后在代码中把 TimeoutDuration 属性设置为 TimeSpan。尽管这里硬编码了这个值，但很可能从其他数据——或许是传递给工作流的一个参数或者从配置文件中读取的一个值——构建该属性值。工作流参数在 57.4 节中讨论。

57.2.5 ListenActivity

一个常见的编程结构是等待一组事件中某个可能的事件，例如 System.Threading.WaitHandle 类的 WaitAny()方法。ListenActivity 是在工作流中等待事件的一种方式，因为它可以定义任意多个分支，每个分支都把基于事件的活动作为该分支的第一个活动。

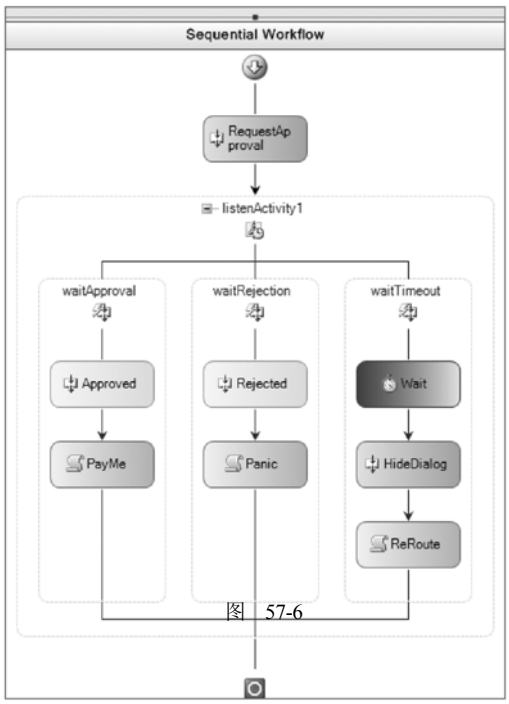

图 57-6

事件活动是实现在 System.Workflow.Activities 名称空间中定义的 IEventActivity 接口的活动。目前 WF 将 3 个这样的活动定义为标准活动：DelayActivity、HandleExternalEventActivity 和 WebServiceInputActivity。图 57-6 中的工作流正在等待外部输入或者某个延迟——这是前面讨论的费用申请工作流的一个示例。

在这个例子中，CallExternalMethodActivity 用作工作流中的第一个活动，它会调用在服务接口上定义的方法，服务接口提示经理批准或不批准——因为这是一个外部服务，所以该提示可以是电子邮件、IM 消息或用其他方式通知经理，需要处理一个费用申请。之后，工作流执行 ListenActivity，等待来自这个外部服务的输入(批准或不批准)，同时也在延迟。

在执行 ListenActivity 时，它实际上会将一个等待操作放在每个分支的第一个活动中，在触发一

个事件时，会取消所有其他等待事件，然后处理在其中触发事件的分支的其他活动。所以在该情况中，如果批准费用报告，就引发 Approved 事件，然后调度 PayMe 活动。但如果经理没有批准该费用申请，就引发 Rejected 事件，然后调度 Panic 活动。

最后，如果 Approved 事件和 Rejected 事件都没有引发，DelayActivity 最终就会在延迟到期后完成，费用报告可以路由给下一个经理——可能在 Active Directory 中查找这个人。在本示例中，因为在执行 RequestApproved 活动时会给用户显示一个对话框，所以如果执行 DelayActivity 时，那么还需要关闭这个对话框，而这就是图 57-6 中 HideDialog 活动的作用。

这个例子的代码在 02 Listen 目录下。该示例使用了前面没有介绍的一些概念，如工作流实例如何标识，如何引发事件返回工作流运行库，最终发送给正确的工作流实例。这些概念将在 57.4 节中探讨。

57.2.6　活动执行模型

到目前为止，本章仅讨论了运行库通过调用 Execute()方法执行活动。实际上，活动在执行过程中会经历许多不同的状态，如图 57-7 所示。

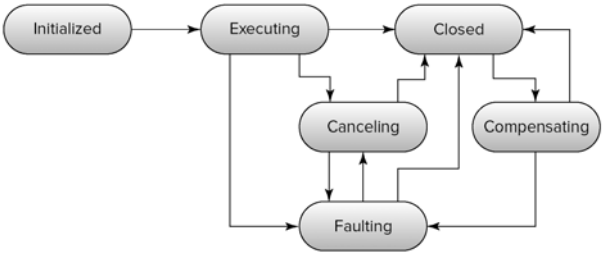

图　57-7

WorkflowRuntime 先调用活动的 Initialize()方法，初始化这个活动。给这个方法传递 IServiceProvider 实例，该实例映射到运行库中可用的服务。这些服务将在 57.6 节中讨论。大多数活动在这个方法中都是什么都不做，但这个方法会进行一些必要的设置。

接着，运行库调用 Execute()方法，活动就可以从 ActivityExecutionStatus 枚举中返回一个值。一般应从 Execute()方法中返回 Closed，表示活动处理完毕；但是如果返回其他状态值中的某个值，运行库就使用该值确定活动所处的状态。

从这个方法中可以返回 Executing，表示运行库还有额外的工作要做。一个典型的例子是当有一个复合活动并且它需要执行其子活动时。在这种情况下，活动可以调度其每个子活动以便执行，接着等待所有子活动执行完毕，之后通知运行库，活动已完成。

57.3　自定义活动

前面使用的都是 System.Workflow.Activities 名称空间中定义的活动。本节将学习如何创建自定义活动，扩展它们，从而在设计时和运行时提供更好的用户体验。

首先，创建 WriteLineActivity，它用于一行文本输出到控制台上。这是一个简单的例子，但后面将扩展它，介绍使用该示例的自定义活动的所有可用选项。在创建自定义活动时，可以仅在工作流项目中构建一个类，但最好在一个独立的程序集中构建自定义活动，因为 Visual Studio 的设计环

境(尤其是和工作流项目)会从程序集中加载活动，并能在更新程序集时锁定它。所以，应创建一个简单的类库项目，在其中构建自定义活动。

简单的活动(如 WriteLineActivity)直接派生自 Activity 基类。下面的代码显示一个构建的活动类，并定义 Message 属性，在调用 Execute()方法时会显示该属性：

```
using System;
using System.ComponentModel;
using System.Workflow.ComponentModel;

namespace SimpleActivity
{
   /// <summary>
   /// A simple activity that displays a message to the console when it executes
   /// </summary>
   public class WriteLineActivity: Activity
   {
      /// <summary>
      /// Execute the activity-display the message on screen
      /// </summary>
      /// <param name="executionContext"></param>
      /// <returns></returns>
      protected override ActivityExecutionStatus Execute
        (ActivityExecutionContext executionContext)
      {
        Console.WriteLine(Message);

        return ActivityExecutionStatus.Closed;
      }

      /// <summary>
      /// Get/Set the message displayed to the user
      /// </summary>
      [Description("The message to display")]
      [Category("Parameters")]
      public string Message
      {
        get { return _message; }
        set { _message = value; }
      }

      /// <summary>
      /// Store the message displayed to the user
      /// </summary>
      private string _message;
   }
  }
```

在 Execute()方法中，可以将消息写入控制台中，再返回 Closed 状态，通知运行库，活动已完成。

也可以在 Message 属性上定义特性，以便为这个属性定义其描述内容和类别。这将在 Visual Studio 的属性网格中使用，如图 57-8 所示。

本节用于创建活动的代码在 03 CustomActivities 解决方

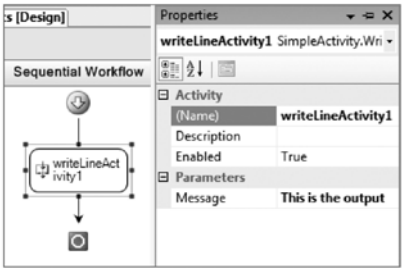

图 57-8

案中。如果编译该解决方案，就可以从工具箱的上下文菜单中选择 Choose Items 菜单项，导航到包含活动的程序集所驻留的文件夹，将自定义活动添加到 Visual Studio 的工具箱中。程序集中的所有活动都添加到工具箱中。

这个活动肯定是可以使用的，但是，有几个地方可以使这个活动更友好。与本章前面的 CodeActivity 一样，它有一些强制属性，如果没有定义，就会在设计界面上生成一个错误标志符号。为了从活动获得相同的行为，需要构建一个派生自 ActivityValidator 的类，将这个类与自定义活动关联起来。

57.3.1　活动的验证

当把活动放在设计界面上时，工作流设计器就会在该活动上查找一个属性，它定义对活动进行验证的类。为了验证活动，需要检查是否设置了 Message 属性。

给活动实例传递一个自定义验证器，以确定哪些强制属性没有定义，并给设计器使用的 ValidationErrorCollection 添加一个错误。之后工作流设计器读取这个集合，在该集合中发现的错误会将一个标志符号添加到活动中，并将每个错误链接到需要注意的属性上。

```
using System;
using System.Workflow.ComponentModel.Compiler;

namespace SimpleActivity
{
   public class WriteLineValidator: ActivityValidator
   {
      public override ValidationErrorCollection Validate
         (ValidationManager manager, object obj)
      {
         if (null == manager)
            throw new ArgumentNullException("manager");
         if (null == obj)
            throw new ArgumentNullException("obj");

         ValidationErrorCollection errors = base.Validate(manager, obj);

         // Coerce to a WriteLineActivity
         WriteLineActivity act = obj as WriteLineActivity;

         if (null != act)
         {
            if (null != act.Parent)
            {
               // Check the Message property
               if (string.IsNullOrEmpty(act.Message))
                errors.Add(ValidationError.GetNotSetValidationError("Message"));
            }
         }

         return errors;
      }
   }
}
```

更新活动的任一部分时，以及将活动拖放在设计界面上时，设计器都会调用 Validate()方法。设计器调用 Validate()方法时，会把活动传递为非类型化的 obj 参数。

在这个方法中，先验证传入的参数，再调用基类的 Validate() 方法得到一个 ValidationErrorCollection。尽管这不是严格必需的，但如果从有许多需要验证的属性的活动中派生，调用基类方法就能确保也检查这些属性。

代码然后将传递的 obj 参数强制转换为 WriteLineActivity 实例，检查活动是否有父活动。这个测试是必需的，因为 Validate()函数在编译活动的过程中调用(假定活动在一个工作流项目或活动库中)，此时并没有定义父活动。没有这个检查，实际上就不能构建包含活动的程序集和验证器。如果项目的类型是类库，就不需要这个额外的步骤。

最后一步是检查 Message 属性是否设置为一个值，而不是设置为空字符串——这使用 ValidationError 类的一个静态方法，它构造一个错误，说明没有定义属性。

为了给 WriteLineActivity 实例添加验证支持，最后一步是将 ActivityValidation 属性添加到活动中，如下面的代码段所示：

```
  [ActivityValidator(typeof(WriteLineValidator))]
public class WriteLineActivity: Activity
{
  .
  }
```

如果编译该应用程序，再将一个 WriteLineActivity 实例拖放在工作流上，就会看到如图 57-9 所示的验证错误；单击这个错误，就能在属性网格中查看这个属性。

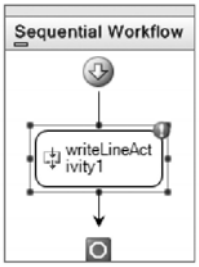

图 57-9

如果给 Message 属性输入一些文本，就会删除验证错误，之后就可以编译并运行应用程序。

既然完成了活动的验证后，接下来就要更改活动的呈现行为，给该活动添加填充色。为此，需要定义 ActivityDesigner 类和 ActivityDesignerTheme 类，如下一节所述。

57.3.2 主题和设计器

活动的屏幕呈现使用 ActivityDesigner 类执行，这也可以使用 ActivityDesignerTheme 类执行。主题类用于在工作流设计器中对活动的呈现行为进行简单的更改。

```
public class WriteLineTheme: ActivityDesignerTheme
{
  /// <summary>
  /// Construct the theme and set some defaults
  /// </summary>
  /// <param name="theme"></param>
  public WriteLineTheme(WorkflowTheme theme)
  : base(theme)
  {
    this.BackColorStart = Color.Yellow;
    this.BackColorEnd = Color.Orange;
    this.BackgroundStyle = LinearGradientMode.ForwardDiagonal;
  }
}
```

主题派生自 ActivityDesignerTheme，它有一个作为 WorkflowTheme 参数传递的构造函数。在该构造函数中，为活动设置起始颜色和结束颜色，再定义一个线性渐变画笔，用于绘制背景。

Designer 类用于重写活动的呈现行为——因为这里不需要重写，所以下面的代码就足够了：

```
[ActivityDesignerTheme(typeof(WriteLineTheme))]
public class WriteLineDesigner: ActivityDesigner
{
}
```

注意使用 ActivityDesignerTheme 属性将主题与设计器关联起来。

最后一步是用 Designer 属性修饰活动：

```
[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
public class WriteLineActivity: Activity
{
  .
}
```

于是，活动的呈现结果如图 57-10 所示。

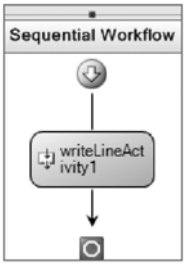

图 57-10

添加设计器和主题之后，活动现在看上去更专业。主题上还有许多其他属性，如用于呈现边框的钢笔、边框的颜色、边框的样式等。

重写 ActivityDesigner 类的 OnPaint()方法，可以完全控制活动的呈现。这里最好适量练习一下，因为可以更进一步，创建一个与工具箱中任何其他活动都不雷同的活动。

在 ActivityDesigner 类中，另一个有用的重写属性是 Verbs。它允许在活动的上下文菜单中添加菜单项。用 ParallelActivity 的设计器将 Add Branch 菜单项插入活动的上下文菜单和 Workflow 菜单中。还可以重写设计器的 PreFilterProperties()方法，修改为活动提供的属性列表，这是 CallExternalMethodActivity 的方法参数显示到属性网格中的方式。如果需要对设计器进行这类扩展，就应运行 Red Gate 的 Reflector(http://www.reflector.red-gate.com)，并在其中加载工作流程序集，查看 Microsoft 如何定义一些扩展属性。

这个活动快完成了，现在需要定义呈现活动时使用的图标，以及与活动关联的工具箱选项。

57.3.3　ActivityToolboxItem 和图标

为了完成自定义活动，需要添加一个图标。还可以创建一个派生自 ActivityToolboxItem 的类，在 Visual Studio 的工具箱中显示活动时，要用到这个类。

为了给活动定义图标，创建一幅 16×16 像素的图像。将它包含到项目中，之后将图像的构建操作设置为 Embedded Resource。这会将该图像包含到程序集的清单资源中。可以给项目添加一个 Resources 文件夹，如图 57-11 所示。

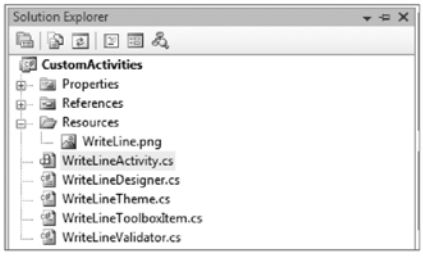

图　57-11

一旦添加了图像文件并将其构建操作设置为 Embedded Resource 后，就可以设置活动的属性，如下面的代码段所示：

```
  [ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
[ToolboxBitmap(typeof(WriteLineActivity),"Resources.WriteLine.png")]
public class WriteLineActivity: Activity
{
  .
}
```

ToolboxBitmap 属性定义许多构造函数，这里使用的构造函数的参数是活动程序集中定义的类型和资源名称。当将一个资源添加到文件夹中时，因为从程序集的名称空间和图像驻留的文件夹的名称构造其名称，所以该资源的完全限定名称是 CustomActivities.Resources.WriteLine.png。给 ToolboxBitmap 属性使用的构造函数将类型参数所驻留的名称空间追加到作为第二个参数传递的字符串后面，这样当 Visual Studio 加载它时，这个名称会解析为相应的资源。

最后一个需要创建的类派生自 ActivityToolboxItem。当把该活动加载到 Visual Studio 工具箱中时，会使用这个类。该类的一个典型应用是更改活动在工具箱上显示的名称——所有内置活动都会更改其名称，从类型中删除“Activity”。在这个类中，要将 DisplayName 属性设置为 WriteLine，以更改活动的名称。

```
  [Serializable]
public class WriteLineToolboxItem: ActivityToolboxItem
{
   /// <summary>
   /// Set the display name to WriteLine - i.e. trim off
   /// the 'Activity' string
   /// </summary>
   /// <param name="t"></param>
   public WriteLineToolboxItem(Type t)
     : base(t)
   {
      base.DisplayName = "WriteLine";
   }

   /// <summary>
   /// Necessary for the Visual Studio design time environment
   /// </summary>
   /// <param name="info"></param>
   /// <param name="context"></param>
   private WriteLineToolboxItem(SerializationInfo info,
                                StreamingContext context)
   {
      this.Deserialize(info, context);
   }
}
```

这个类派生自 ActivityToolboxItem，并重写构造函数，以更改显示名称；它还提供了一个序列化构造函数，当将选项加载到工具箱中时，工具箱会使用这个序列化构造函数。如果没有这个构造函数，在试图将活动添加到工具箱上时，就会接收到一个错误。注意这个类也标记为[Serializable]。

使用 ToolboxItem 属性把工具箱中的选项添加到活动中，如下面的代码所示：

```
  [ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
```

```
[ToolboxBitmap(typeof(WriteLineActivity),"Resources.WriteLine.png")]
[ToolboxItem(typeof(WriteLineToolboxItem))]
public class WriteLineActivity: Activity
{
  .
  }
```

所有这些更改都完成后，就可以编译程序集，然后创建一个新的工作流项目。要把活动添加到工具箱中，可以打开一个工作流，然后显示工具箱的上下文菜单，并单击 Choose Items 按钮。

图　57-12

接着可以浏览包含活动的程序集，把活动添加到工具箱中，结果如图 57-12 所示。图标看起来不太漂亮，但它说明我们的工作已经完成了。

下一节将在自定义复合活动中再次访问 ActivityToolboxItem，因为这个类有一些额外的功能，将复合活动添加到设计界面上时，需要这些功能。

再次重申，我们创建了一个自定义 WriteLineActivity，创建了 WriteLineValidator 以添加验证逻辑，使用 WriteLineDesigner 类和 WriteLineTheme 类创建设计器和主题，为活动创建一个位图，最后创建了 WriteLineToolboxItem 类，以修改活动的显示名称。

57.3.4　自定义复合活动

活动有两种主要类型，派生自 Activity 的活动可以看作能从工作流中调用的函数。派生自 CompositeActivity 的活动(如 ParallelActivity、IfElseActivity 和 ListenActivity)是其他活动的容器，它们在设计期间的行为完全不同于简单的活动，因为它们显示在设计器的一个区域中，可以在该区域中拖放子活动。

本节将创建一个活动，称为 DaysOfWeekActivity。这个活动可以根据当前的日期执行工作流的不同部分。例如，在工作流中，为周末到达的订单执行的路径需要与正常工作日到达的订单的执行路径不同。在这个例子中，读者将学习许多高级工作流主题，在本节末尾之前将很好地理解如何用自己的复合活动扩展系统。这个例子的代码也放在 03 CustomActivities 解决方案中。

首先，创建一个自定义活动，它的一个属性默认为当前日期/时间。可以将该属性设置为另一个值，该值来自于工作流中的另一个活动；或把该属性设置为一个执行工作流时传递给工作流的一个参数。这个复合活动包含用户定义的许多分支，每个分支都包含一个枚举常量，该常量定义将执行该分支的日期。下面的示例定义该活动及其两个分支：

```
DaysOfWeekActivity
  SequenceActivty: Monday, Tuesday, Wednesday, Thursday, Friday
    <other activites as appropriate>
  SequenceActivity: Saturday, Sunday
    <other activites as appropriate>
```

这个例子需要一个定义一周中各天的枚举，其中包含[Flags]属性(这样就不能使用 System 名称空间中定义的内置枚举 DayOfWeek，因为它不包含[Flags]属性)。

```
  [Flags]
[Editor(typeof(FlagsEnumEditor), typeof(UITypeEditor))]
public enum WeekdayEnum: byte
{
```

```
    None = 0x00,
    Sunday = 0x01,
    Monday = 0x02,
    Tuesday = 0x04,
    Wednesday = 0x08,
    Thursday = 0x10,
    Friday = 0x20,
    Saturday = 0x40
}
```

这个类型还包含一个自定义编辑器，它允许根据复选框修改枚举值。其代码可以下载。

定义了枚举类型后，就可以考虑活动的主干了。自定义复合活动一般派生自 CompositeActivity 类，此外，因为该基类定义 Activities 属性，它是所有后续活动的集合。

```
public class DaysOfWeekActivity: CompositeActivity
{
   /// <summary>
   /// Get/Set the day of week property
   /// </summary>
   [Browsable(true)]
   [Category("Behavior")]
   [Description("Bind to a DateTime property, set a specific date time,
                or leave blank for DateTime.Now")]
   [DefaultValue(typeof(DateTime),"")]
   public DateTime Date
   {
      get { return (DateTime)
               base.GetValue(DaysOfWeekActivity.DateProperty); }
      set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
   }

   /// <summary>
   /// Register the DayOfWeek property
   /// </summary>
   public static DependencyProperty DateProperty =
      DependencyProperty.Register("Date", typeof(DateTime),
         typeof(DaysOfWeekActivity));
}
```

Date 属性提供常规的 get 和 set 存取器，本例还添加了许多标准特性，以便 Date 属性能正确地显示在属性浏览器中。其代码看起来与正常的.NET 属性有所不同，因为 get 和 set 存取器没有使用标准字段存储其值，而是使用 DependencyProperty。

Activity 类(之所以是这个类，因为它最终派生自 Activity)派生自 DependencyObject 类，它定义了一个键为 DependencyProperty 的值字典。这种使用 get 和 set 存取器间接得到的属性值由 WF 用于支持绑定，即把一个活动的属性链接到另一个活动的属性上。例如，我们常常要在代码中传递参数，有时是按值传递，有时是按引用传递。因为 WF 使用绑定将属性值链接在一起，所以本例在工作流上定义了一个 DateTime 属性，这个活动需要在运行期间绑定到该属性值上。本章的后面将列举一个绑定的例子。

如果构建这个活动，那么并没有做太多的工作；实际上它甚至不允许在该活动中拖放子活动，因为还没有为该活动定义 Designer 类。

1. 添加设计器

与本章前面的 WriteLineActivity 一样，每个活动都有一个关联的 Designer 类，Designer 类用于更改该活动在设计期间的行为。虽然 WriteLineActivity 中的 Designer 类是空的，但对于复合活动，需要重写两个方法，用于添加某些特殊的分支处理。

```
public class DaysOfWeekDesigner: ParallelActivityDesigner
{
   public override bool CanInsertActivities
      (HitTestInfo insertLocation, ReadOnlyCollection<Activity> activities)
   {
      foreach (Activity act in activities)
      {
         if (!(act is SequenceActivity))
            return false;
      }

      return base.CanInsertActivities(insertLocation, activitiesToInsert);
   }

   protected override CompositeActivity OnCreateNewBranch()
   {
      return new SequenceActivity();
   }
}
```

这个 Designer 类派生自 ParallelActivityDesigner，它在添加子活动时提供很好的设计时行为。如果所拖放的任何活动不是 SequenceActivity，就需要将 CanIsertActivities 重写为返回 false。如果所有活动的类型都正确，就可以调用基类方法，进一步检查自定义活动所允许的活动类型。

还需要重写 OnCreateNewBranch()方法，它在用户选择 Add Branch 菜单项时调用。Designer 类使用[Designer]属性与该活动关联起来，如下面的代码所示：

```
 [Designer(typeof(DaysOfWeekDesigner))]
public class DaysOfWeekActivity: CompositeActivity
{
}
```

设计期间的操作就快完成了，然而，还需给活动添加一个派生自 ActivityToolboxItem 的类，指定将该活动的一个实例从工具箱中拖出时会发生什么。默认行为是仅构造一个新活动，然而，在本例中还希望创建两个默认分支。下面的代码完整地显示了这个工具箱选项类：

```
 [Serializable]
public class DaysOfWeekToolboxItem: ActivityToolboxItem
{
   public DaysOfWeekToolboxItem(Type t)
      : base(t)
   {
      this.DisplayName = "DaysOfWeek";
   }

   private DaysOfWeekToolboxItem(SerializationInfo info,
                                 StreamingContext context)
   {
```

```
            this.Deserialize(info, context);
        }

        protected override IComponent[] CreateComponentsCore(IDesignerHost host)
        {
            CompositeActivity parent = new DaysOfWeekActivity();
            parent.Activities.Add(new SequenceActivity());
            parent.Activities.Add(new SequenceActivity());
            return new IComponent[] { parent };
        }
    }
```

代码更改了活动的显示名称，实现了序列化构造函数，并重写了 CreateComponentsCore()方法。

这个方法在拖放操作的最后调用，在该方法中构造 DayOfWeekActivity 的一个实例。在代码中还构造两个子序列活动，为活动的用户提供了更好的设计时体验。几个内置活动与 DayOfWeekActivity 有相同的作用，当把一个 IfElseActivity 实例拖放到设计界面上时，它的工具箱选项类也会添加两个分支。当将 ParallelActivity 实例添加到工作流中时，也会添加两个分支。

序列化构造函数和[Serializable]特性是所有派生自 ActivityToolboxItem 的类都需要的。

最后，将这个工具箱选项类与该活动关联起来：

```
[Designer(typeof(DaysOfWeekDesigner))]
[ToolboxItem(typeof(DaysOfWeekToolboxItem))]
public class DaysOfWeekActivity: CompositeActivity
{
}
```

之后，活动的 UI 就接近完成了，如图 57-13 所示。

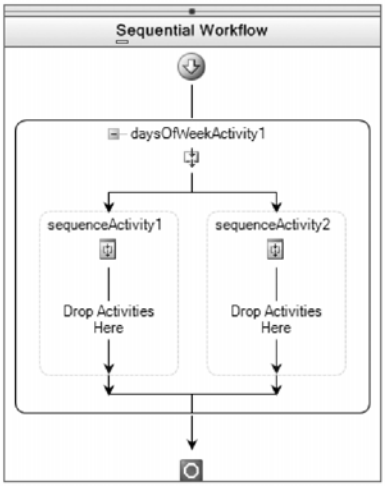

图 57-13

现在需要在图 57-13 的每个系列活动上定义一个属性，以便用户能指定分支执行的日期。这在 Windows 工作流中有两种方式：可以创建 SequenceActivity 的一个子类，在该子类中定义该属性；也可以使用依赖属性的另一个功能——附加属性。

这里采用第二种方式，因为它不需要创建子类，但可以有效地扩展序列活动，而无需活动的源代码。

2. 附加属性

在注册依赖属性时，可以调用 RegisterAttached()方法，创建一个附加属性。附加属性是在一个类中定义但在另一个类中显示的属性。所以这里在 DayOfWeekActivity 上定义一个属性，但这个属性实际上显示在 UI 上，作为系列活动的一个附加属性。

下面的代码段显示一个 WeekdayEnum 类型的 Weekday 属性，它会添加到驻留在复合活动中的序列活动中。

```
public static DependencyProperty WeekdayProperty =
    DependencyProperty.RegisterAttached("Weekday",
        typeof(WeekdayEnum), typeof(DaysOfWeekActivity),
        new PropertyMetadata(DependencyPropertyOptions.Metadata));
```

最后一行代码允许指定属性的额外信息，这个例子指定它是一个 Metadata 属性。

Metadata 属性与一般的属性不同，它只能在运行期间读取。Metadata 属性类似于 C#中的常量声明。在程序正在执行时不能更改常量，在工作流正在执行时也不能更改 Metadata 属性。

在这个例子中，因为要指定执行活动的日期，所以在设计器中将这个字段设置为“Saturday, Sunday”。在工作流发出的代码中，应有如下的声明(代码已重新设置了格式，以适应页面的大小)：

```
this.sequenceActivity1.SetValue
  (DaysOfWeekActivity.WeekdayProperty,
  ((WeekdayEnum)((WeekdayEnum.Sunday | WeekdayEnum.Saturday))));
```

除了定义依赖属性之外，还需要利用方法在任意活动上获取和设置这个值。这些方法一般定义为复合活动上的静态方法，如下面的代码所示：

```
public static void SetWeekday(Activity activity, object value)
{
   if (null == activity)
      throw new ArgumentNullException("activity");
   if (null == value)
      throw new ArgumentNullException("value");

   activity.SetValue(DaysOfWeekActivity.WeekdayProperty, value);
}

public static object GetWeekday(Activity activity)
{
   if (null == activity)
      throw new ArgumentNullException("activity");

   return activity.GetValue(DaysOfWeekActivity.WeekdayProperty);
}
```

还需要更改另外两个地方，才能使这个额外的属性显示为 SequenceActivity 的附加属性。第一处更改是创建一个扩展提供程序，它告诉 Visual Studio 在序列活动中包含额外的属性。第二处更改是注册这个提供程序，具体操作是重写活动设计器的 Initialize()方法，并添加如下代码：

```
protected override void Initialize(Activity activity)
{
   base.Initialize(activity);

   IExtenderListService iels = base.GetService(typeof(IExtenderListService))
      as IExtenderListService;

   if (null != iels)
   {
      bool extenderExists = false;

      foreach (IExtenderProvider provider in iels.GetExtenderProviders())
      {
         if (provider.GetType() == typeof(WeekdayExtenderProvider))
         {
            extenderExists = true;
            break;
         }
      }
      if (!extenderExists)
```

```
        {
            IExtenderProviderService ieps =
                base.GetService(typeof(IExtenderProviderService))
                    as IExtenderProviderService;
            if (null != ieps)
                ieps.AddExtenderProvider(new WeekdayExtenderProvider());
        }
    }
}
```

在上述代码中对 GetService()方法的调用允许自定义设计器查询宿主(这里是 Visual Studio)设置的服务。在 Visual Studio 中查询 IExtenderListService，它提供用于枚举所有可用的扩展提供程序的一种方式。如果没有找到 WeekdayExtenderProvider 服务的实例，就查询 IExtenderProviderService，并添加一个新的提供程序。

扩展提供程序的代码如下所示：

```
  [ProvideProperty("Weekday", typeof(SequenceActivity))]
public class WeekdayExtenderProvider: IExtenderProvider
{
    bool IExtenderProvider.CanExtend(object extendee)
    {
        bool canExtend = false;

        if ((this != extendee) && (extendee is SequenceActivity))
        {
            Activity parent = ((Activity)extendee).Parent;

            if (null != parent)
                canExtend = parent is DaysOfWeekActivity;
        }

        return canExtend;
    }

    public WeekdayEnum GetWeekday(Activity activity)
    {
        WeekdayEnum weekday = WeekdayEnum.None;

        Activity parent = activity.Parent;

        if ((null != parent) && (parent is DaysOfWeekActivity))
            weekday = (WeekdayEnum)DaysOfWeekActivity.GetWeekday(activity);

        return weekday;
    }

    public void SetWeekday(Activity activity, WeekdayEnum weekday)
    {
        Activity parent = activity.Parent;

        if ((null != parent) && (parent is DaysOfWeekActivity))
            DaysOfWeekActivity.SetWeekday(activity, weekday);
    }
}
```

扩展提供程序用它提供的属性标识，对于这些属性，它必须提供公共的 Get<Property>()和

Set<Property>()方法。这些方法的名称必须匹配属性的名称，并带有相应的 Get 或 Set 前缀。

对设计器进行了上述更改，并添加扩展提供程序之后，在设计器中添加一个序列活动(这里是 SequenceActivity1)时，就会在 Visual Studio 中看到该属性，如图 57-14 所示。

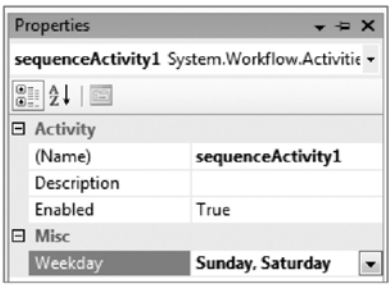

图　57-14

扩展提供程序用于.NET 中的其他功能，其中一个常见的功能是在 Windows 窗体项目中给控件添加工具提示。在窗体上添加工具提示控件时，会注册一个扩展器，为窗体上的每个控件添加一个 Tooltip 属性。

57.4　工作流

到目前为止，本章主要探讨了活动，但没有讨论工作流。工作流只是一个活动列表，实际上工作流本身是另一个类型的活动。使用这个模型简化运行库引擎，因为运行库引擎只需知道如何执行一种对象——派生自 Activity 类的对象。

每个工作流实例都用其 InstanceId 属性唯一地标识，InstanceId 属性是一个可以由运行库指定的 Guid，或者这个 Guid 可以由代码提供给运行库。Guid 的一个常见用途是将正在运行的工作流实例与在工作流外部维护的其他数据关联起来，如数据库中的一行。使用 WorkflowRuntime 类的 GetWorkflow(Guid)方法可以访问特定的工作流实例。

WF 中有两种工作流：顺序工作流和状态机工作流。

57.4.1　顺序工作流

顺序工作流中的根活动是 SequenceWorkflowActivity。这个类派生自前面介绍的 SequenceActivity，它定义两个根据需要可以附加处理程序的事件——Initialized 和 Completed。

顺序工作流首先执行其中的第一个子活动，之后执行第二个子活动，直到所有其他子活动都执行完毕为止。在两种情况下，工作流不会继续执行所有活动：一种是执行工作流的过程中引发了异常，另一种是工作流中存在 TerminateActivity。

工作流不会在所有情况下都执行。例如，当遇到 DelayActivity 时，工作流就进入等待状态，如果定义了工作流持续服务，就会从内存中删除工作流。工作流的持续性在 57.6.1 节中介绍。

57.4.2　状态机工作流

如果进程处于几种状态中的一种，只要给工作流传递数据，就可以使进程从一个状态转换到另一个状态，此时就可以使用状态机工作流。

一个例子是工作流用于对大厦的访问控制。此时，可以为一个 door 类建模，它可以关闭或打开，和一个 lock 类，它可以锁定或解锁。在最初启动系统(或大厦)时，就进入了一个已知状态——为了便于讨论，假定所有门都是关闭，且已上锁，那么某个门的状态是 closed locked。

当雇员从前门输入其访问代码时，会把一个事件发送给工作流，其中包含的细节有输入的代码和用户 ID。接着需要访问数据库，检索信息，例如，是否允许这个人在给定的这个时间打开选定的门，假定授权了该访问权限，工作流就会从其初始状态改为 closed unlocked 状态。

在这个状态下，有两种可能的结果——雇员打开该门(这是因为该门安装了打开/关闭传感器)；或者因为雇员发现把东西落在了汽车上，所以不打算进门了，于是，在过了延迟时间后要重新锁上门。因此该门返回 closed locked 状态，或进入 open unlocked 状态。

现在假定雇员进入大厦，并关上门——接着要从 open unlocked 状态进入 closed unlocked 状态，在过了延迟时间后要转换为 Closed locked 状态。如果该门处于 open unlocked 状态的时间很长，那么还要引发警报。

在 Windows Workflow 中为这种情形建模很简单：需要定义系统的状态，再定义可以使工作流从一个状态转换为另一个状态的事件。表 57-2 描述了系统的状态，并提供了每个已知状态可能的转换的细节，以及改变状态的输入(外部或内部输入)。

表 57-2

状　态	切　换
Closed Locked	这是系统的初始状态 为了响应用户的刷卡动作(且成功通过了访问检查)，状态会改为 Closed Unlocked，门锁会通过电子方式打开
Closed Unlocked	当门处于这种状态时，会发生如下两个事件中的一个： (1) 用户打开门——将状态转换为 Closed Locked 状态 (2) 定时器过期，门返回 Closed Locked 状态
Open Unlocked	在这个状态下，工作流只能转换为 Closed Unlocked 状态
Fire Alarm	这是工作流的最后一个状态，从其他 3 种状态都可以转换为这种状态

可以添加到系统中的另一个功能是响应火警的能力。当发出警报时，应打开所有门，让所有人离开大厦，消防员可以顺畅地进入大厦。可以把这个状态作为门工作流的最后一个状态建模，因为一旦取消火警，就可以从这个状态重置系统。

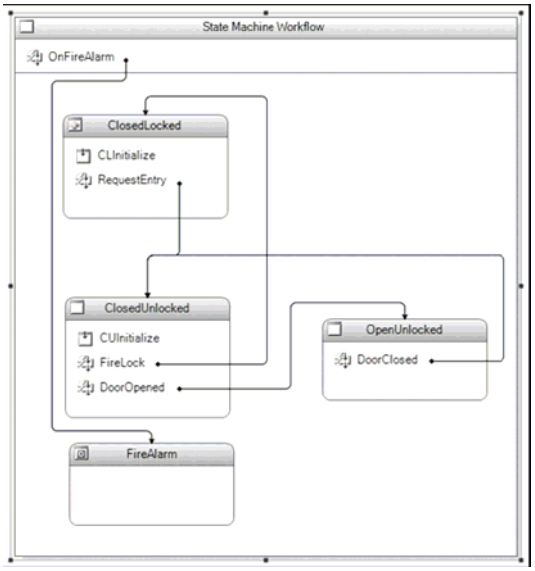

图 57-15

图 57-15 中的工作流定义这个状态机，并显示工作流可以处于的状态。线条说明系统中可以进行的状态转换。

工作流的初始状态用 ClosedLocked 活动建模。这包含一些初始化代码(锁上门)，然后是一个基于事件的活动，它等待一个外部事件，在这个例子中，雇员输入了大厦的访问代码。因为状态图中显示的每个活动都由序列工作流组成，所以为系统的初始化定义一个工作流(CLInitialize)，再定义一个 RequestEntry 工作流，该工作流会响应当雇员输入其 PIN 时引发的外部事件。如果查看 RequestEntry 工作流，它的定义就如图 57-16 所示。

每个状态都由许多子工作流组成，每个子工作流的开始都有一个事件驱动的活动，之后是任意多

个其他活动，它们构成了状态中的处理代码。在图 57-16 中，开头有一个 HandleExternalEventActivity，它在等待输入 PIN。之后工作流将检查 PIN，如果它是有效的，工作流就转换为 ClosedUnlocked 状态。

ClosedUnlocked 状态由两个工作流组成。一个工作流响应开门事件，将工作流转换为 OpenUnlocked 状态；另一个工作流包含一个延迟活动，该活动用于将状态改为 ClosedLocked。状态驱动的活动与本章前面的 ListenActivity 采用相同的方式工作——状态由许多事件驱动的工作流组成，当发生一个事件时，就执行其中一个工作流。

为了支持工作流，需要引发系统中的事件，实现状态的改变。为此，需要使用一个接口和该接口的实现代码，这对对象称为外部服务。本章后面将描述用于这个状态机的接口。

上述状态机例子的代码在 04 StateMachine 解决方案中，其中还包含一个用户界面，在该用户界面中可以输入 PIN，通过两扇门中的一扇门进入大厦。

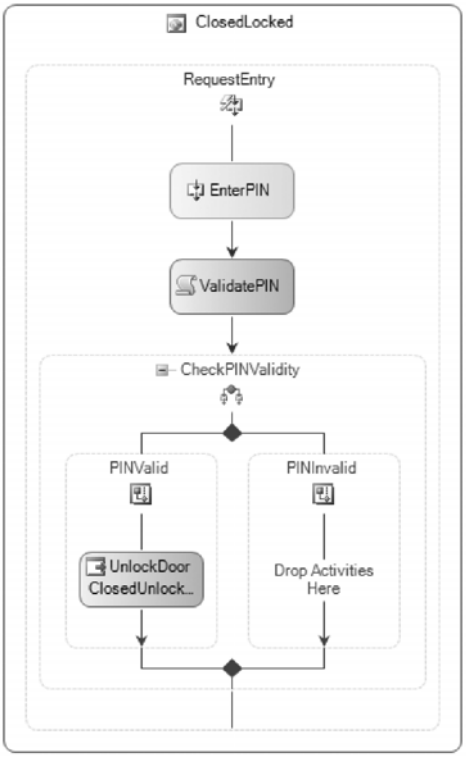

图　57-16

57.4.3　给工作流传递参数

工作流一般需要一些数据才能执行，例如，订单处理工作流需要一个订单 ID，付费处理工作流需要一个顾客账户 ID，或者其他必要的数据项。

工作流的参数传递机制与标准的.NET 类有所不同，在标准的.NET 类中，一般在方法调用中传递参数。而对于工作流，传递参数时，要把这些参数存储在一个名称/值对字典中，在构造工作流时，要经过这个字典。

当 WF 调度工作流以便执行时，它使用这些名称-值对设置关于工作流实例的公共属性。每个参数名都要用工作流的公共属性检查。如果找到了匹配的属性值，就调用属性设置器，把该参数的值

传递给设置器。如果将一个名称-值对添加到字典中，在该字典中，名称并不对应工作流上的一个属性，则在试图构造这个工作流时，会抛出一个异常。

例如，下面的工作流将 OrderID 属性定义为一个整数：

```
public class OrderProcessingWorkflow: SequentialWorkflowActivity
{
   public int OrderID
   {
      get { return _orderID; }
      set { _orderID = value; }
   }

   private int _orderID;
}
```

下面的代码说明了如何将订单 ID 参数传递给工作流的一个实例：

```
WorkflowRuntime runtime = new WorkflowRuntime ();

Dictionary<string,object> parms = new Dictionary<string,object>();
parms.Add("OrderID", 12345);

WorkflowInstance instance = runtime.CreateWorkflow(typeof(OrderProcessingWorkflow),
                                                   parms);
instance.Start();

. Other code
```

在上面的示例代码中，构造一个 Dictionary<string, object>，它包含要传递给工作流的参数，在构造工作流时要使用这个字典。上面的代码包含 WorkflowRuntime 类和 WorkflowInstance 类，这里没有介绍它们，57.8 节将介绍它们。

57.4.4 从工作流中返回结果

工作流的另一个常见要求是返回输出参数，它们可能用于将在数据库或其他永久存储器中记录数据。

因为工作流由工作流运行库执行，所以不能只使用标准的方法调用机制调用工作流，而需要创建一个工作流实例，启动这个实例，然后等待它的完成。工作流完成后，工作流运行库就会引发 WorkflowCompleted 事件。这是传递的关于工作流的上下文信息，该工作流已经完成，并包含从工作流中输出的数据。

所以，要获取从该工作流中返回的输出参数，需要将一个事件处理程序关联到 WorkflowCompleted 事件上，该处理程序可以从工作流中检索输出参数。下面的代码显示了如何实现上述操作的一个例子：

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
   AutoResetEvent waitHandle = new AutoResetEvent(false);
   workflowRuntime.WorkflowCompleted +=
      delegate(object sender, WorkflowCompletedEventArgs e)
      {
         waitHandle.Set();
         foreach (KeyValuePair<string, object> parm in e.OutputParameters)
         {
```

```
                Console.WriteLine("{0} = {1}", parm.Key, parm.Value);
            }
        };

        WorkflowInstance instance =
            workflowRuntime.CreateWorkflow(typeof(Workflow1));
        instance.Start();

        waitHandle.WaitOne();
    }
```

我们把一个委托关联到 WorkflowCompleted 事件上，在这个委托中，迭代传递给该委托的 WorkflowCompletedEventArgs 类中的 OutputParameters 集合，并在控制台上显示输出参数。这个集合包含工作流的所有公共属性。实际上工作流没有指定输出参数。

57.4.5　将参数绑定到活动上

既然理解了如何将参数传递给工作流，就还需要明白如何把这些参数链接到活动上。这通过绑定机制实现。在前面定义的 DaysOfWeekActivity 中，有一个 Date 属性，它可以硬编码，也可以绑定到工作流中的另一个值上。可绑定的属性显示在 Visual Studio 的属性网格中，如图 57-17 所示。Data 属性名右边的图标表示，这是一个可绑定的属性。

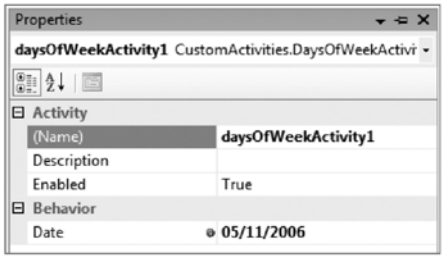

图　57-17

双击绑定图标会显示如图 57-18 所示的对话框。该对话框允许选择一个合适的属性，链接到 Date 属性。

在图 57-18 中，选择了工作流的 OrderDate 属性(把 OrderData 属性定义为工作流上一个普通的.NET 属性)。任何 Bindable 属性都可以绑定到定义活动的工作流的属性上，或者绑定到驻留在工作流中当前活动上方的任何活动的属性上。注意，被绑定的属性的数据类型必须匹配正在绑定的属性的数据类型，该对话框不允许绑定不匹配的类型。

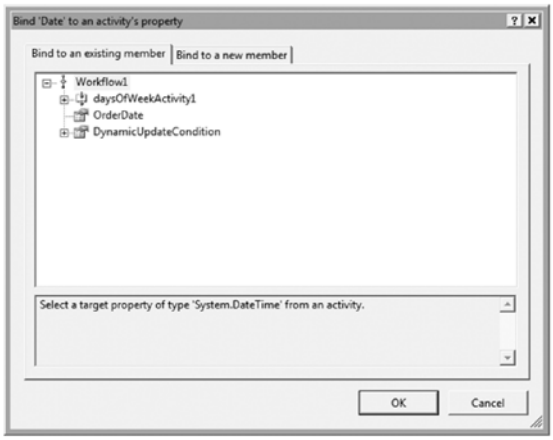

图　57-18

下面再次显示 Date 属性的代码，说明绑定的工作方式，并在后面几段中详细解释。

```
public DateTime Date
{
     get { return (DateTime)base.GetValue(DaysOfWeekActivity.DateProperty); }
     set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
}
```

当绑定工作流中的一个属性时，会在后台构造一个 ActivityBind 类型的对象，它就是存储在依赖属性中的值。所以，需要给属性设置器传递一个 ActivityBind 类型的对象，它存储在这个活动的属性字典中。这个 ActivityBind 对象包含的数据描述正在绑定的活动和要在运行时使用的活动的属性。

在读取属性值时，调用 DependencyObject 的 GetValue()方法，这个方法会检查基础属性值，确定它是否是 ActivityBind 对象。如果是，它就解析这个绑定链接的活动，然后从该活动中读取属性值。但是，如果绑定的值是另一种类型，它就仅从 GetValue()方法中返回该对象。

57.5 工作流运行库

为了启动工作流，需要创建 WorkflowRuntime 类的一个实例。这一般在应用程序中创建，这个对象通常定义为应用程序的一个静态成员，以便在应用程序的任意地方可以访问它。

在启动运行库时，它会从永久存储器中读取上次执行应用程序时执行的工作流实例，重新加载它们。这需要使用一个持久性服务，详见 57.6 节的内容。

运行库包含 6 个用于构造工作流实例的 CreateWorkflow()方法。运行库还包含几个方法，可以重载工作流实例，并枚举所有正在运行的实例。

运行库还有许多在执行工作流时引发的事件，例如，WorkflowCreated(在构造新的工作流实例时引发)、WorkflowIdled(在工作流等待输入时引发，如前面的费用处理例子)和 WorkflowCompleted(在工作流完成时引发)。

57.6 工作流服务

工作流不能独自存在，如前一节所述，工作流在 WorkflowRuntime 中执行，这个运行库提供了运行工作流的服务。

该服务可以是执行工作流时需要的任何类。运行库为工作流提供了一些标准服务，用户也可以选择构造自己的服务，由正在运行的工作流使用这些服务。

本节将描述运行库提供的两个标准服务，再说明如何构建自己的服务和何时需要一些实例。

在活动运行时，要通过 Execute()方法的 ActivityExecutionStatus 参数给活动传递一些上下文信息。

```
protected override ActivityExecutionStatus Execute
   (ActivityExecutionContext executionContext)
{
   .
}
```

在这个上下文参数上其中一个可用的方法是 GetService<T>()。在下面的代码中，它用于访问与工作流运行库关联的一个服务：

```
protected override ActivityExecutionStatus Execute
   (ActivityExecutionContext executionContext)
{
   ICustomService myService = executionContext.GetService<ICustomService>();
   . Do something with the service
}
```

在调用 StartRuntime()方法之前，运行库驻留的服务会添加到运行库中。如果试图将某个服务添加到已启动的运行库中，就会引发一个异常。

两个方法可用于将服务添加到运行库中：可以在代码中构造服务，再调用 AddService()方法将它们添加到运行库中；也可以在应用程序配置文件中定义服务，构造这些服务并把它们添加到运行库中。

下面的代码段说明了如何在代码中将服务添加到运行库中——所添加的服务将在本节的后面介绍。

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
   workflowRuntime.AddService(
      new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                        new TimeSpan(0,10,0)));
   workflowRuntime.AddService(new SqlTrackingService(conn));
   .
}
```

这段代码构造 SqlWorkflowPersistenceService 的实例，运行库使用这些实例存储工作流的状态。这段代码还构造 SqlTrackingService 的一个实例，它记录工作流运行时执行的事件。

要使用应用程序配置文件创建服务，需要为工作流运行库添加一个节处理程序，然后将服务添加到该节中，如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
      <section name="WF"
         type="System.Workflow.Runtime.Configuration.WorkflowRuntimeSection,
         System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
         PublicKeyToken=31bf3856ad364e35" />
   </configSections>

   <WF Name="Hosting">
     <CommonParameters/>
       <Services>
         <add type="System.Workflow.Runtime.Hosting.SqlWorkflowPersistenceService,
                    System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
                    PublicKeyToken=31bf3856ad364e35"
           connectionString="Initial Catalog=WF;Data Source=.;
                             Integrated Security=SSPI;"
           UnloadOnIdle="true"
           LoadIntervalSeconds="2"/>
         <add type="System.Workflow.Runtime.Tracking.SqlTrackingService,
                    System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
                    PublicKeyToken=31bf3856ad364e35"
           connectionString="Initial Catalog=WF;Data Source=.;
                             Integrated Security=SSPI;"
```

```
            UseDefaultProfile="true"/>
        </Services>
    </WF>
</configuration>
```

在配置文件中，添加了 WF 节处理程序(这个名称并不重要，但必须匹配其后配置节的名称)，之后为该节创建合适的数据项。<Services>元素可以包含一个任意数据项列表，其中包含一个.NET 类型和运行库构造服务时传递给服务的参数。

要从应用程序配置文件中读取配置设置，可以调用运行库上的另一个构造函数，如下所示：

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime("WF"))
{
    .
}
```

这个构造函数会实例化在配置文件中定义的每个服务，并把它们添加到运行库上的服务集合中。

下面几节将介绍 WF 提供的一些标准服务。

57.6.1 持久性服务

执行工作流时，它可能会进入等待状态——在执行延迟活动时，或者在 ListenActivity 中等待外部输入时，就会进入等待状态。此时，工作流处于空闲状态，这是持久性服务的一个候选对象。

假定开始在服务器上执行 1000 个工作流，之后每个工作流实例都进入空闲状态。此时，因为不需要在内存中维护这些实例的数据，所以最好能卸载工作流，释放它使用的资源。持久性服务就用于完成这个任务。

当某个工作流进入空闲状态时，工作流运行库会检查是否存在一个派生自 Workflow PersistenceService 类的服务。如果该服务存在，就给它传递工作流实例，接着服务就可以捕获工作流的当前状态，并将它存储在永久存储介质上。可以把工作流的状态存储在磁盘的一个文件中，或把这些数据存储在数据库(如 SQL Server)中。

工作流库包含持久性服务的实现代码，持久性服务可以把数据存储在 SQL Server 数据库中，即 SqlWorkflowPersistenceService。为了使用这个服务，需要对 SQL Server 实例运行两个脚本，其中一个脚本构造对应架构，另一个脚本创建持久性服务使用的存储过程。这些脚本默认位于 C:\Windows\Microsoft.NET\Framework\v3.5\Windows Workflow Foundation\SQL\EN 目录下。

在数据库上执行的脚本是 SqlPersistenceServiceProviderSchema.sql 和 SqlPersistenceProvider_ Logic. sql。这些脚本需要按顺序执行：先执行架构文件，再执行逻辑文件。SQL 持久性服务的架构包含两个表：InstanceState 和 CompletedScope。这些都是不透明的表，不在 SQL 持久性服务的外部使用。

当工作流空闲时，其状态用二进制序列化机制序列化，接着把这些数据插入 InstanceState 表中。重新激活工作流时，从这一行中读取状态，用于重新构建工作流实例。这一行用工作流实例 ID 作为键，一旦工作流完成就从数据库中删除该行。

SQL 持久性服务可以由多个运行库同时使用——它实现锁定机制，以便一次只能由工作流运行库的一个实例访问一个工作流。如果多个服务器都使用同一个永久存储器运行工作流，这个锁定行为就非常有用。

为了查看添加到永久存储器中的内容，需要构造一个新的工作流项目，并给运行库添加

SqlWorkflowPersistenceService 的一个实例。下面是使用声明性代码的一个例子：

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
   workflowRuntime.AddService(
      new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                        new TimeSpan(0,10,0)));
   // Execute a workflow here.
}
```

接着，如果构造一个包含 DelayActivity 的工作流，并将延迟时间设置为 10 秒，就可以查看在 InstanceState 表中存储的数据。05 WorkflowPersistence 示例包含上述代码，并在 20 秒的时间内执行一个延迟。

持久性服务的构造函数的参数如表 57-3 所示。

表　57-3

参　数	说　明	默 认 值
ConnectionString	持久性服务使用的数据库连接字符串	None
UnloadOnIdle	确定工作流在空闲时是否卸载它。它应总是设置为 true，否则就不会出现任何持久性	False
InstanceOwnershipDuration	定义已加载工作流的运行库拥有工作流实例的时间长度	None
LoadingInterval	在给数据库请求更新的永久性记录时使用的时间间隔	2 分钟

这些值也可以在配置文件中定义。

57.6.2　跟踪服务

当工作流执行时，它可能需要记录运行了哪些活动，对于 IfElseActivity 和 ListenActivity 等复合活动，执行其分支。这些数据可以用作工作流实例的一种审计线索，在以后的某个日期查看，证明执行了哪些活动，在工作流中使用了什么数据。跟踪服务可以用于这类记录操作，并可以配置，根据需要记录关于正在运行的工作流的尽可能少或尽可能多的信息。

在 WF 中，因为跟踪服务作为一个抽象类 TrackingService 实现，所以很容易用自己的跟踪服务替换标准的跟踪实现方式。在工作流程序集中，跟踪服务有一个具体的实现方式，即 SqlTrackingService。

要记录工作流的状态数据，就需要定义一个 TrackingProfile。它定义应记录什么事件，例如，可以只记录工作流的开头和结束，忽略正在运行的实例的所有其他数据。更一般的情况是，记录工作流的所有事件和其中的每个活动，提供工作流的执行配置文件的完整描述。

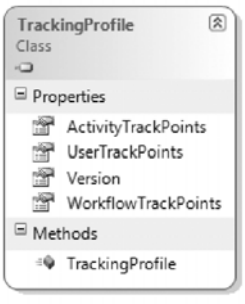

图　57-19

运行库引擎调度工作流时，引擎会检查工作流跟踪服务是否存在。如果找到了一个跟踪服务，它就会要求该服务为正在执行的工作流提供一个跟踪配置文件，接着使用它记录工作流和活动数据。还可以定义用户跟踪数据，把它们存储在跟踪数据存储器中，这些操作不需要改变该架构。

跟踪配置文件类如图 57-19 所示。这个类包含活动、用户和工作流跟踪点的集合属性。跟踪点是一个对象(如 WorkflowTrackPoint)，它一般定义

一个匹配位置和一些额外的数据，在单击这个跟踪点时，就会记录这些额外的数据。匹配位置指定这个跟踪点在什么地方有效。例如，可以定义一个 WorkflowTrackPoint，它记录了创建工作流时的一些数据，再定义一个 WorkflowTrackPoint，记录完成工作流时的一些数据。

一旦记录这些数据，就需要显示工作流的执行路径，如图 57-20 所示。该图显示了执行的工作流，运行的每个活动都包含一个说明它已执行的标志符号。这些数据从工作流实例的跟踪存储器中读取。

为了读取 SqlTrackingService 存储的数据，可以直接在 SQL 数据库上执行查询。Microsoft 还为此在 System.Workflow.Runtime.Tracking 名称空间中提供了 SqlTrackingQuery 类。下面的示例说明了如何检索在两个日期之间跟踪的所有工作流的一个列表：

```
public IList<SqlTrackingWorkflowInstance> GetWorkflows
   (DateTime startDate, DateTime endDate, string connectionString)
{
   SqlTrackingQuery query = new SqlTrackingQuery (connectionString);

   SqlTrackingQueryOptions queryOptions = new SqlTrackingQueryOptions();
   query.StatusMinDateTime = startDate;
   query.StatusMaxDateTime = endDate;

   return (query.GetWorkflows (queryOptions));
}
```

这段代码使用 SqlTrackingQueryOptions 类，该类定义对应的查询参数。可以定义这个类的其他属性，进一步约束要检索的工作流。

在图 57-20 中，可以看到所有活动都执行了。但如果工作流仍在运行，或者在工作流中做了一些决策，以便在执行过程中采用不同的路径，情况就有所改变。跟踪数据包含执行了哪些活动等信息，这些数据与生成图 57-20 中的映像的活动相关。还可以在工作流执行时检索这些数据，用于为工作流的执行流生成审计线索。

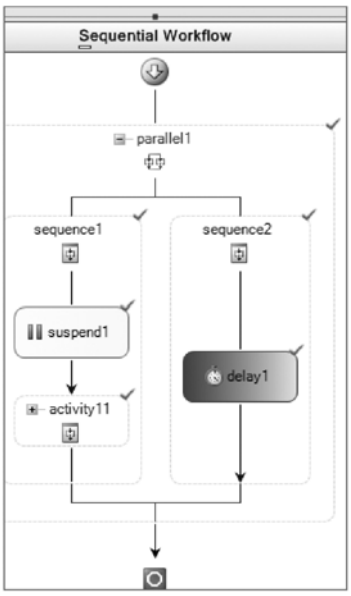

图 57-20

57.6.3　自定义服务

除了持久性服务和跟踪服务等内置服务之外，还可以在 WorkflowRuntime 维护的服务集合中添加自己的对象。这些服务一般用一个接口和一种实现方式定义，以便在不记录工作流的情况下替换该服务。

本章前面的状态机利用了下面的接口：

```
  [ExternalDataExchange]
public interface IDoorService
{
    void LockDoor();
    void UnlockDoor();

    event EventHandler<ExternalDataEventArgs> RequestEntry;
    event EventHandler<ExternalDataEventArgs> OpenDoor;
    event EventHandler<ExternalDataEventArgs> CloseDoor;
    event EventHandler<ExternalDataEventArgs> FireAlarm;

    void OnRequestEntry(Guid id);
    void OnOpenDoor(Guid id);
    void OnCloseDoor(Guid id);
    void OnFireAlarm();
}
```

工作流使用组成这个接口的方法调用该服务，该服务引发的事件由工作流使用。使用 ExternalDataExchange 属性是告诉工作流运行库，这个接口用于正在运行的工作流和服务实现之间的通信。

在状态机中，有许多 CallExternalMethodActivity 的实例，它们用于在这个外部接口上调用方法。例如，当门上锁或不上锁时，工作流需要执行对 UnlockDoor()或 LockDoor()方法的方法调用，服务的响应是给门锁发送一条命令，锁上门，或打开门。

当服务需要与工作流通信时，应使用一个事件完成该任务，因为工作流运行库也包含一个 ExternalDataExchangeService 服务，它作为这些事件的代理。这个代理在引发事件时使用，因为在传递事件时，工作流可能没有加载到内存中，所以该事件会先路由到外部数据交换服务，它检查工作流是否已加载，如果没有加载，就从永久存储器中解除冻结工作流，再把该事件传递给工作流。

下面的代码构造 ExternalDataExchangeService，还为该服务定义的事件构造多个代理：

```
WorkflowRuntime runtime = new WorkflowRuntime();
ExternalDataExchangeService edes = new ExternalDataExchangeService();

runtime.AddService(edes);
DoorService service = new DoorService();
edes.AddService(service);
```

这段代码构造外部数据交换服务的一个实例，把它添加到运行库中。接着它创建 DoorService 的一个实例(它本身实现 IDoorService)，并把它添加到外部数据交换服务中。

ExternalDataExchangeService.Add()方法为自定义服务定义的每个事件构造一个代理，以便在传递事件之前加载已持久化的工作流。如果没有把服务驻留在外部数据交换服务中，在引发事件时，就不会侦听这些事件，它们也不会传递给正确的工作流。

事件使用 ExternalDataEventArgs 类，因为它包含待传递事件的工作流实例 ID。如果需要将其他值从外部事件传递给工作流，就应从 ExternalDataEventArgs 中派生一个类，并把这些值作为属性添加到该类中。

57.7 与 WCF 集成

从.NET 3.5 开始可用两个新活动支持工作流和 WCF 之间的集成，这两个活动是 SendActivity 和 ReceiveActivity。SendActivity 称为 CallActivity 更恰当，因为它的工作是向 WCF 服务发出一个请求，并可以选择性地把结果显示为参数，这些参数可以绑定到主调工作流中。

但更有趣的是 ReceiveActivity。因为它允许工作流变成 WCF 服务的实现方式，所以现在工作流就是服务。下面例子提供一个使用一个工作流的服务，并且也使用一个新的服务测试宿主工具，测试服务，而无需编写独立的测试工具。

从 Visual Studio 2010 的 New Project 菜单中选择 WCF 节点，再选择 Sequential Workflow Service Library 项，如图 57-21 所示。

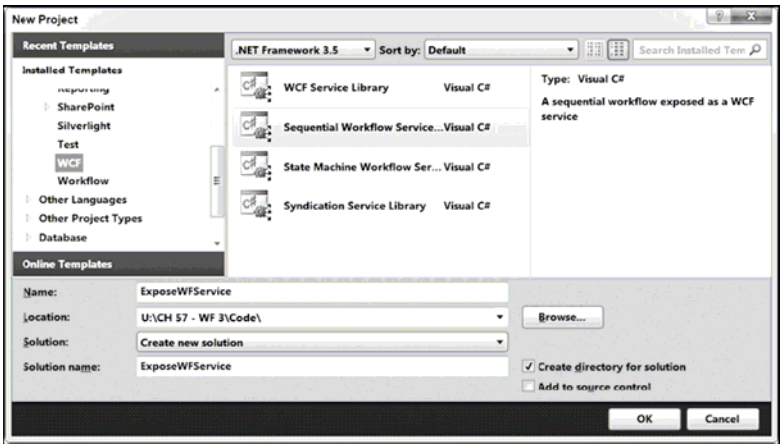

图 57-21

这会创建一个库，其中包含了一个工作流、一个应用程序配置文件和一个服务接口，如图 57-22 所示。这个示例的代码在 06 ExposeWFService 子目录中。

工作流提供协定的 Hello 操作，还定义要传递给这个操作的参数属性，以及该操作的返回值。接着只需添加代码，代码提供服务的执行行为，该服务就完成了。

为此，对于本示例，把一个 CodeActivity 拖放到 ReceiveActivity 上，如图 57-23 所示。再双击该活动，提供服务的实现代码。

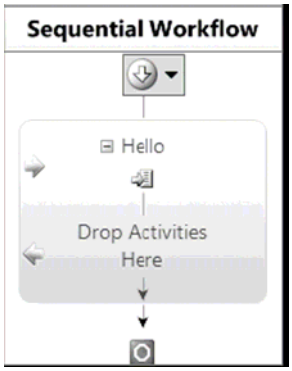

图 57-22

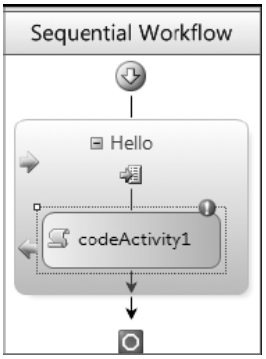

图 57-23

下面就是这个服务的实现代码。

```
public sealed partial class Workflow1: SequentialWorkflowActivity
{
    public Workflow1()
    {
        InitializeComponent();
    }

    public String returnValue = default(System.String);
    public String inputMessage = default(System.String);

    private void codeActivity1_ExecuteCode(object sender, EventArgs e)
    {
        this.returnValue = string.Format("You said {0}", inputMessage);
    }
}
```

因为 Hello 操作的服务协定包含 inputMessage 参数和一个返回值，所以它们都作为公共字段提供给工作流。在代码中，把 returnValue 设置为一个字符串值，这就是从 WCF 服务的调用中返回的内容。

如果编译这个服务，按 F5 键，就会注意到 Visual Studio 2010 的另一个新功能：WCF 测试客户端应用程序，如图 57-24 所示。

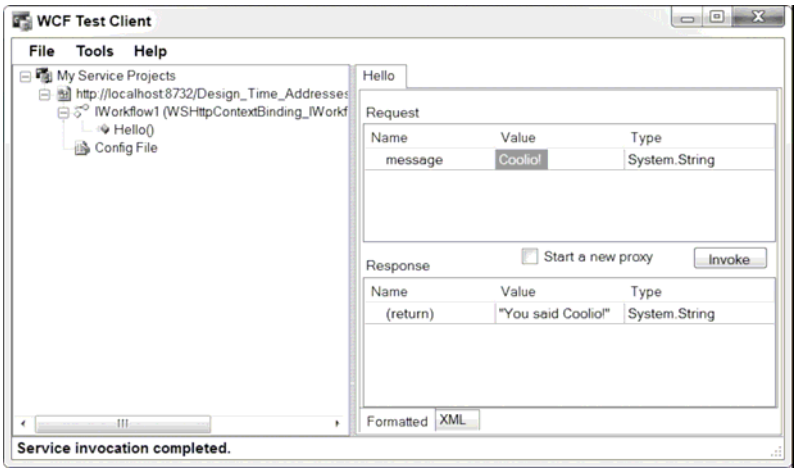

图　57-24

在这里可以浏览服务提供的操作，双击一个操作，会显示窗口的右半部分，其中列出了该服务使用的参数和提供的任意返回值。

要测试该服务，可以给 message 属性输入一个值，单击 Invoke 按钮。这会通过 WCF 给该服务发送一个请求，构造并执行工作流，调用代码活动，代码活动然后运行代码隐藏，最终给 WCF 测试客户端返回工作流的结果。

如果要手动作为服务驻留工作流，那么可以使用在 System.WorkflowServices 名称空间中定义的新的 WorkflowServiceHost 类。下面的代码段显示了一个非常小的宿主的实现方式：

```
using (WorkflowServiceHost host = new WorkflowServiceHost
```

```
                                        (typeof(YourWorkflow)))
{
   host.Open();
   Console.WriteLine ( "Press [Enter] to exit" );
   Console.ReadLine();
}
```

这里构造 WorkflowServiceHost 的一个实例，把它传递给要执行的工作流。这类似于驻留 WCF 服务时使用 ServiceHost 类的方式。它将读取配置文件，确定服务应侦听哪个端点，然后等待服务请求。

下一节介绍驻留工作流的某些其他选项。

57.8 驻留工作流

在进程中驻留 WorkflowRuntime 的代码随应用程序的不同而不同。

对于 Windows 窗体应用程序或 Windows 服务，一般在应用程序的开头构造运行库，把它存储在主应用程序类的一个属性中。

为了响应应用程序中的一些输入(如用户单击了用户界面上的一个按钮)，可能就需要构造工作流的一个实例，在本地执行这个实例。工作流可能还需要与用户通信，例如，可以定义一个外部服务，在将订单发送给后端服务器之前，提示用户确认。

在 ASP.NET 中驻留工作流时，一般不用消息框提示用户，而是导航到站点上请求确认的另一个页面上，再显示一个确认页面。在 ASP.NET 中驻留运行库时，一般要重写 Application_Start 事件，在那里构造工作流运行库的一个实例，以便在站点的所有其他部分访问它。运行库实例可以存储在一个静态属性中，但最好把它存储在应用程序状态中，再提供一个访问方法，从应用程序状态中检索工作流运行库，使之可用于应用程序的其他地方。

在 Windows 窗体或 ASP.NET 中，都要构造工作流运行库的一个实例，给它添加服务，如下所示：

```
WorkflowRuntime workflowRuntime = new WorkflowRuntime();

workflowRuntime.AddService(
   new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                     new TimeSpan(0,10,0)));
// Execute a workflow here.
```

要执行工作流，需要使用运行库的 CreateInstance()方法，创建该工作流的一个实例。这个方法有许多重写版本，重写版本可用于构造基于代码的工作流的实例或在 XML 中定义的工作流的实例。

本章前面都把工作流看作.NET 类，实际上这只是工作流的一种表示形式。还可以使用 XML 定义工作流，此时运行库会构造工作流在内存中的表示形式，在调用 WorkflowInstance 的 Start()方法时执行它。

在 Visual Studio 中，可以从 Add New Item 对话框中选择 Sequential Workflow(其代码是独立的)或 State Machine Workflow(其代码是独立的)。这将创建基于 XML 的工作流。这会创建一个扩展名为.xoml 的 XML 文件，并将它加载到设计器中。

把活动添加到设计器中时，会将这些活动持久化到 XML 中，元素的结构定义活动之间的父子关系。下面的 XML 是一个简单的序列工作流，其中包含一个 IfElseActivity 和两个代码活动，分别

用于 IfElseActivity 的每个分支。

```
<SequentialWorkflowActivity x:Class="DoorsWorkflow.Workflow1" x:Name="Workflow1"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
  <IfElseActivity x:Name="ifElseActivity1">
    <IfElseBranchActivity x:Name="ifElseBranchActivity1">
      <IfElseBranchActivity.Condition>
        <CodeCondition Condition="Test" />
      </IfElseBranchActivity.Condition>
      <CodeActivity x:Name="codeActivity1" ExecuteCode="DoSomething" />
    </IfElseBranchActivity>
    <IfElseBranchActivity x:Name="ifElseBranchActivity2">
      <CodeActivity x:Name="codeActivity2" ExecuteCode="DoSomethingElse" />
    </IfElseBranchActivity>
  </IfElseActivity>
</SequentialWorkflowActivity>
```

在活动上定义的特性作为属性持久化到 XML 中，每个活动都作为一个元素持久化。从 XML 可以看出，该结构定义了父活动(如 SequentialWorkflowActivity 和 IfElseActivity)和子活动之间的关系。

执行基于 XML 的工作流与执行基于代码的工作流没有区别，只需使用 CreateWorkflow()方法的一个重写版本，该版本接受一个 XMLReader 实例，接着调用 Start()方法启动该实例即可。

与基于代码的工作流相比，使用基于 XML 的工作流的一个优点是：很容易将工作流的定义存储在数据库中。之后可以在运行时加载这个 XML，执行工作流。修改该工作流定义时，也不需要重新编译代码。

无论工作流是在 XML 中定义，还是在代码中定义，都可以在运行时更改工作流。只需构造一个 WorkflowChanges 对象，它包含要添加到工作流中的所有新活动，接着调用在 WorkflowInstance 类上定义的 ApplyWorkflowChanges()方法，持久化这些更改。这非常有用，因为业务需求时常更改，例如，需要把这些更改应用于一个保险单工作流，在续保日期的一个月之前给客户发送电子邮件，告诉客户他们的保险单需要续保。因为这种更改要基于每个实例进行，所以如果系统中有 100 个保险单工作流，就需要对每个工作流进行这种更改。

57.9　工作流设计器

本章还有最后一个主题要讨论。用于设计工作流的工作流设计器不与 Visual Studio 关联，可以根据需要在自己的应用程序中重新驻留这个设计器。

这意味着，可以发布一个包含工作流的系统，允许最终用户在不需要 Visual Studio 的副本的情况下定制系统。但驻留设计器相当复杂，这个主题可能需要好几章的篇幅，但这超出了本章的范围。在 Web 上有许多重新驻留设计器的例子，建议读者在 http://msdn2.microsoft.com/enus/library/aa480213.aspx 上获得驻留设计器的更多信息。

允许用户定制系统的传统方式是定义一个接口，再让客户实现这个接口，根据需要扩展处理过程。

有了 Windows 工作流，该扩展在整体上变成更多图形，因为可以给用户提供一个空白的工作流作为模板，再提供一个工具箱，其中包含适合于应用程序的自定义活动。他们接着可以编写自己的工作流，添加他们自己编写的自定义活动。

57.10 从 WF 3.X 迁移到 WF 4

在.NET 4 和 Visual Studio .NET 2010 中有一个 WF 新版本，这个新版本与 3.*x* 不后向兼容。概念大都相同，但实现方式完全不同。其优点是可以继续运行 3.*x* 工作流，无需任何修改；但如果希望使用新功能，就必须迁移应用程序。本节介绍这个迁移过程的要点，并提供一些简化这个过程的建议。

WF4 的更多信息可参见第 44 章。

57.10.1 把活动代码提取到服务中

因为 WF 4 的类层次结构与 WF 3.*x* 大不相同，但活动类大都相同；所以为了简化升级过程，建议从活动中删除内联代码，并把它们放在一组服务中(如包含静态方法的简单.NET 类)。

因为数据绑定的本质在 WF 4 中有了很大的变化，所以在 WF 3.*x* 中可以使用依赖属性和/或标准的.NET 属性，而在 WF 4 中需要使用参数对象。迁移代码最简单的方法是移动活动内部的所有处理过程，并把它们移动到一个单独的类中。例如，考虑 WriteLine 活动的代码段：

```
public class WriteLineActivity : Activity
{
   public string Message { get; set; }

   public override ActivityExecutionStatus Execute
   ( ActivityExecutionContext context )
   {
      Console.WriteLine ( Message ) ;
      return ActivityExecutionStatus.Closed;
   }
}
```

如果要把这些代码转换为 WF 4 活动，就需要创建如下类：

```
public static class WriteLineService
{
   public static void WriteLine ( string message )
   {
      Console.WriteLine ( message ) ;
   }
}
```

接着就可以在当前活动中使用这个类(还可以进行独立的单元测试)，当升级到 WF 4 时，要编写的代码会比较少。尽管这是一个简单的例子，但它说明了如何简化升级过程。

57.10.2 删除代码活动

由于 Workflow 4 中活动的代码隐藏样式与 3.*x* 中已有的样式不同，因此，如果在代码隐藏文件中包含代码，就应使用前面建议的方法把这些代码移植到一个库中，以便在 WF 4 中使用它们。

57.10.3　同时运行 WF 3.x 和 4

只要可能，就应尽量同时运行 WF 3.*x* 和 WF 4 工作流，除非不再有 3.*x* 工作流。因为永久存储器和跟踪存储器在 WF 4 中也不同，所以一个合并的系统是最简单的使用方法。如果没有长时间运行的工作流，就没有什么问题，但如果有长时间运行的工作流，最简单的方法是不更改旧工作流，并确保在 WF 4 上创建新工作流。

当应用程序启动工作流时，能用启动 WF 4 工作流的代码替换这些代码吗？如果不能，建议在代码中添加一个能识别版本的执行策略，这样就不必更改调用工作流的代码，并可以调度新工作流实例。

57.10.4　考虑把状态机迁移到流程图上

Workflow 4 目前不支持状态机，也没有对状态机的内置支持。基础活动模型可以处理状态机(实际上可以处理任何工作流处理模型)；然而在最初的 WF 4 版本中，没有状态机活动或设计支持。

与 3.*x* 状态机最接近的是 WF 4 的流程图，它们不是精确匹配，但流程图允许工作流跳转回处理过程的一个早期阶段，状态机工作流一般会这么做。

57.11　小结

Windows 工作流为应用程序的构造方式带来了根本性的改变。现在可以将应用程序的复杂部分都显示为活动，允许用户将活动拖放到工作流中，更改系统的处理方式。

几乎没有应用程序不能应用工作流，从最简单的命令行工具，到包含上百个模块的最复杂的系统。现在 WCF 的通信功能和 WPF 新的 UI 功能使应用程序前进了一大步，Windows 工作流的出现使开发和配置应用程序的方式有了根本的改变。

在 Visual Studio 2010 中，Workflow 3.*x* 基本上被 WF 4 取代，本章提供了简化升级的一些建议。如果正在计划第一次使用工作流，建议从 WF 4 开始，完全绕过 Workflow 3.*x*。