ASP.NET MVC3 快速入门-第一节 概述

(2011-02-23 20:57:18)

转载

标签： 分类： ASP.NETMVC3

web 应用程序

1.1 本教程的学习内容

在本教程中，你将学会如下内容：

- 如何创建一个 ASP.NET MVC 的工程。
- 如何创建 ASP.NET MVC 的控制器（controller）与视图(view)。
- 如何使用 Entity Framework code-first 范例来创建一个新的数据库。
- 如何获取和显示数据。
- 如何编辑数据并且进行数据的有效性验证。

1.2 创建工程

如果要创建一个 ASP.NET MVC3 的工程时，首先运行 Visual Web Developer 2010 Express（本教程中简称“Visual Web Developer”）,并且在起始页(start page)中选择“新建项目”。Visual Web Developer 是一个集成开发环境，你可以使用它来进行各种应用程序的开发。在 Visual Web Developer 的菜单的下面有一个工具条，可以直接点击工具条中的各个工具按钮来进行各种操作，也可以直接点击菜单中的各个菜单项来进行各种操作，此处我们点击“文件”菜单中的“新建项目”菜单项。

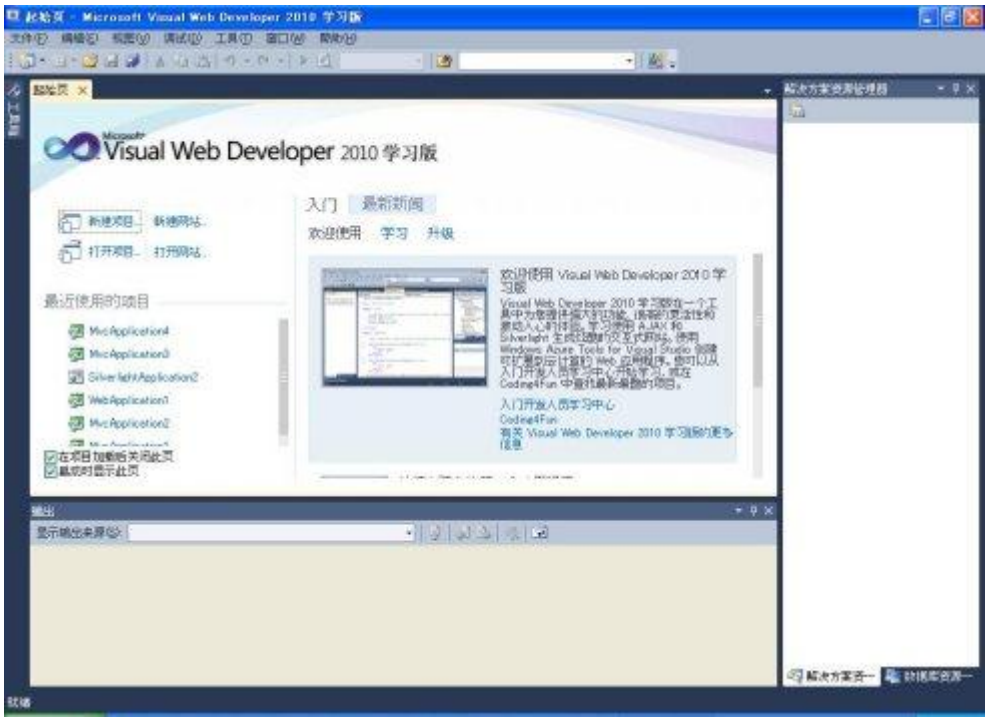


图 1-1 Visual Web Developer 2010 Express 中的起始页

1.3 创建你的第一个应用程序

你可以使用 Visual Basic 或 Visual C#作为开发语言来创建应用程序。在本教程中，选择 C#来作为开发语言。点击“新建项目”菜单项后，在打开的“新建项目”对话框中，双击左边的“Visual C#”使其成为展开状态，然后点击“Web”，点击右边的“ASP.NET MVC 3 Web 应用程序”，然后在下方的名称文本框中填入应用程序的名称，在本教程中命名为“MvcMovie”，然后点击确定按钮。

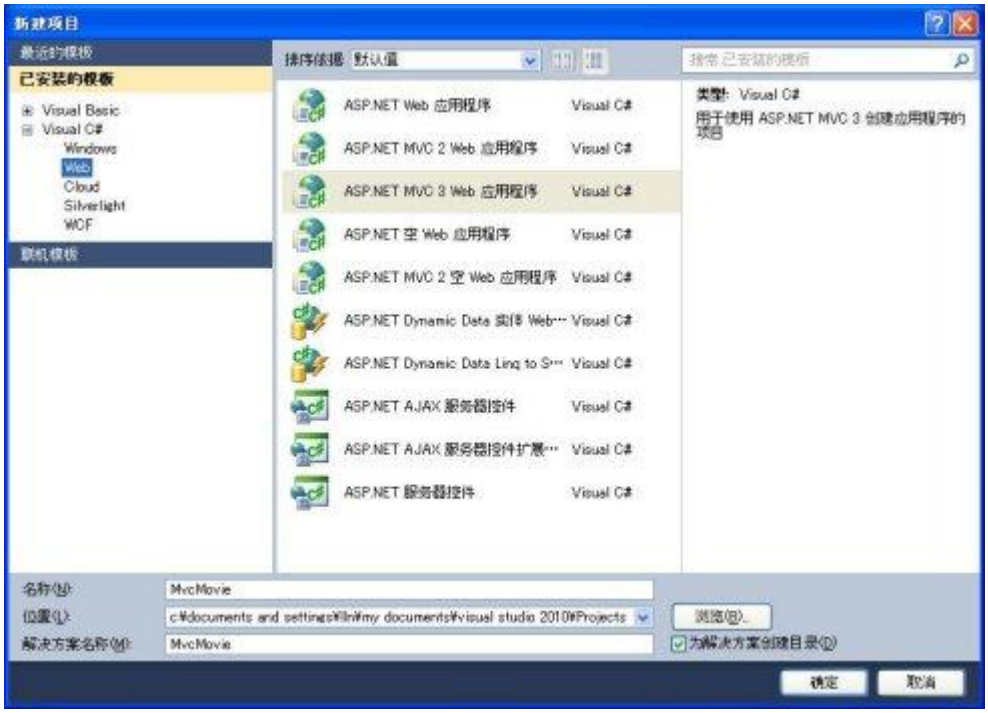


图 1-2 在新建项目对话框中选择 ASP.NET MVC3 应用程序并为应用程序命名

在接下来打开的“新 ASP.NET MVC 3 项目”对话框中，点击选中“Internet 应用程序”，在“视图引擎”下拉框中保持默认的“Razor”选项不作修改（Razor 视图是 ASP.NET MVC3 种新增的一种十分重要的视图类型，使用它可以使得 Web 应用程序的开发变得更加方便快捷，在后文中将对此进行详细介绍）。

图 1-3 选择项目模板与视图引擎

点击确定按钮，Visual Web Developer 会为你所创建的 ASP.NET MVC 项目提供一个默认模板，这样的话你就拥有了一个可以立刻运行的应用程序。默认的模板中提供的是一个很简单的显示“欢迎使用 ASP.NET MVC!”文字的应用程序，你可以以此作为你的开发起点。

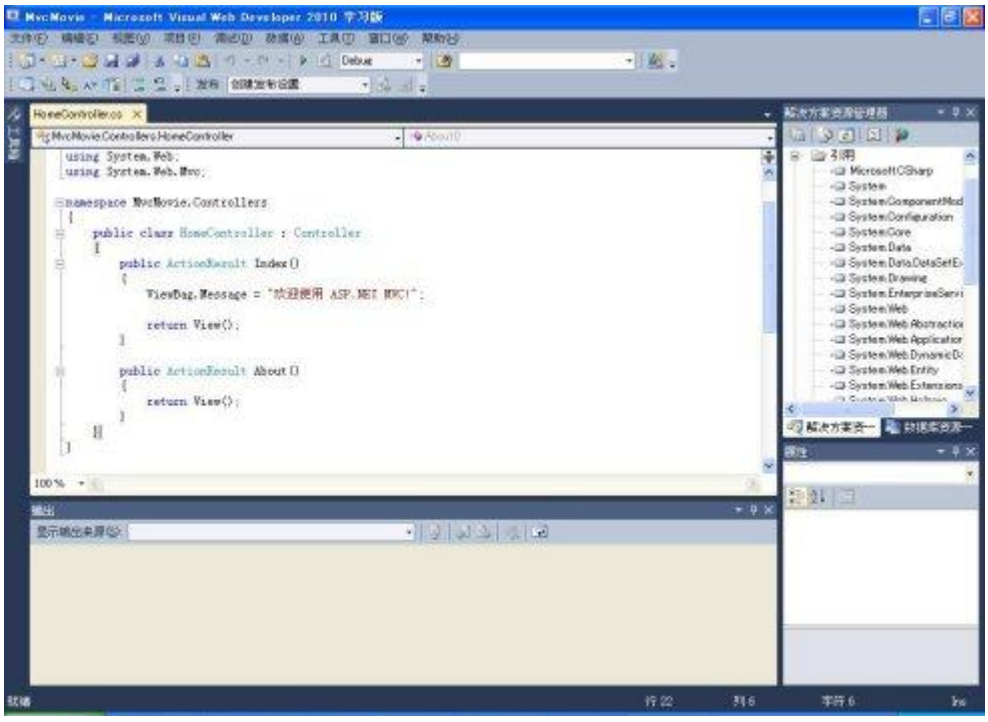


图 1-4 Visual Web Developer 提供了一个默认的应用程序模板

点击“调试”菜单中的“启动调试”菜单项（该菜单项的快捷键为 F5），Visual Web Developer 将启动一个内置的服务器，并且在该服务器中打开当前 Web 应用程序的主页，如图 1-5 所示。

图 1-5 ASP.NET MVC3 的默认应用程序模板的调试画面

请注意该页面在浏览器中的地址为“http://localhost:4423/”。其中“localhost”代表了本机上你刚刚创建的 Web 应用程序的临时网站地址，4423 代表了 Visual Web Developer 使用的一个随机端口，每次调试的时候，Visual Web Developer 都会使用这个端口来作为内置服务器的端口号。在各计算机上，该端口号都是不相同的，因为该端口号是 Visual Web Developer 随机选择的。

在这个模板应用程序的页面的右上角，提供了两个按钮与一个“登录”链接，点击“登录”链接，页面跳转到登录页面，点击“主页”按钮，页面返回到主页，点击“关于”按钮，页面跳转到“关于”页面。

接下来，让我们开始逐步将这个默认的应用程序修改为我们所要的应用程序，在这个过程中逐步了解 ASP.NET MVC 3 的有关知识。首先，让我们关闭浏览器并开始代码的修改工作。

ASP.NET MVC3 快速入门--第二节 添加一个控制器

(2011-02-24 19:39:57)

转载

标签：　　分类：　ASP.NETMVC3

控制器

杂谈

MVC 的全称为 model-view-controller(模型-视图-控制器)。MVC 是一种开发应用程序的模式，这个模式已经具有了很好的框架架构，并且十分容易维护。使用 MVC 开发出来的应用程序一般包括以下几块内容：

- 控制器(Controller)：控制器类处理客户端向 Web 应用程序发出的请求，获取数据，并指定返回给客户端，用来显示处理结果的视图。
- 模型（Model）：模型类代表了应用程序的数据，这些数据通常具有一个数据验证逻辑，用来使得这些数据必须符合业务逻辑。
- 视图（View）:视图类是 Web 应用程序中用来生成并显示 HTML 格式的服务器端对客户端请求的响应结果的模板文件。

在本教程中，将全面介绍这些概念，并且向你展示如何利用它们来搭建一个应用程序。

首先，让我们来创建一个控制器(controller)类。在解决方案资源管理器中，鼠标右击 Controllers 文件夹，并且点击添加-〉控制器，如图 2-1 所示。

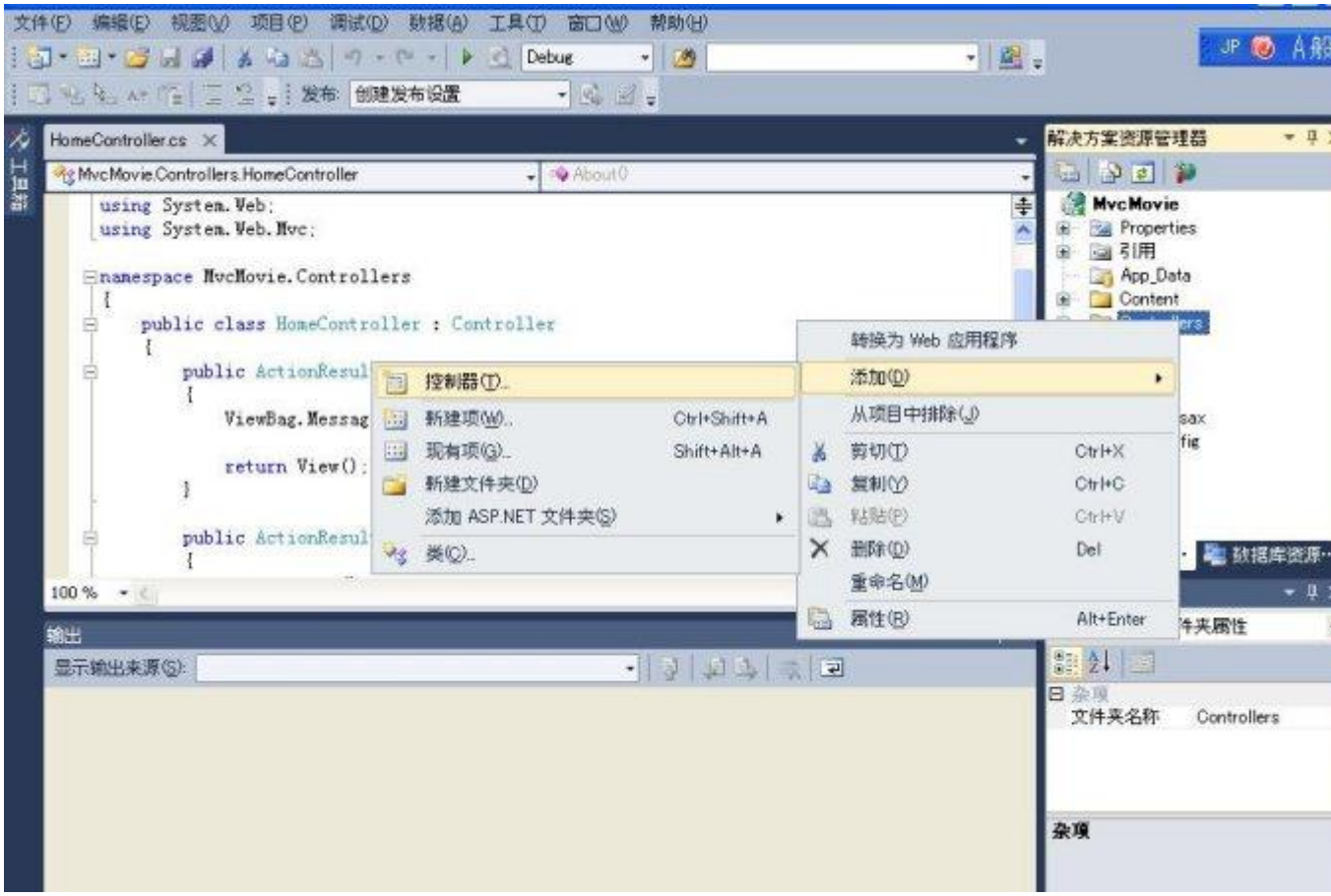
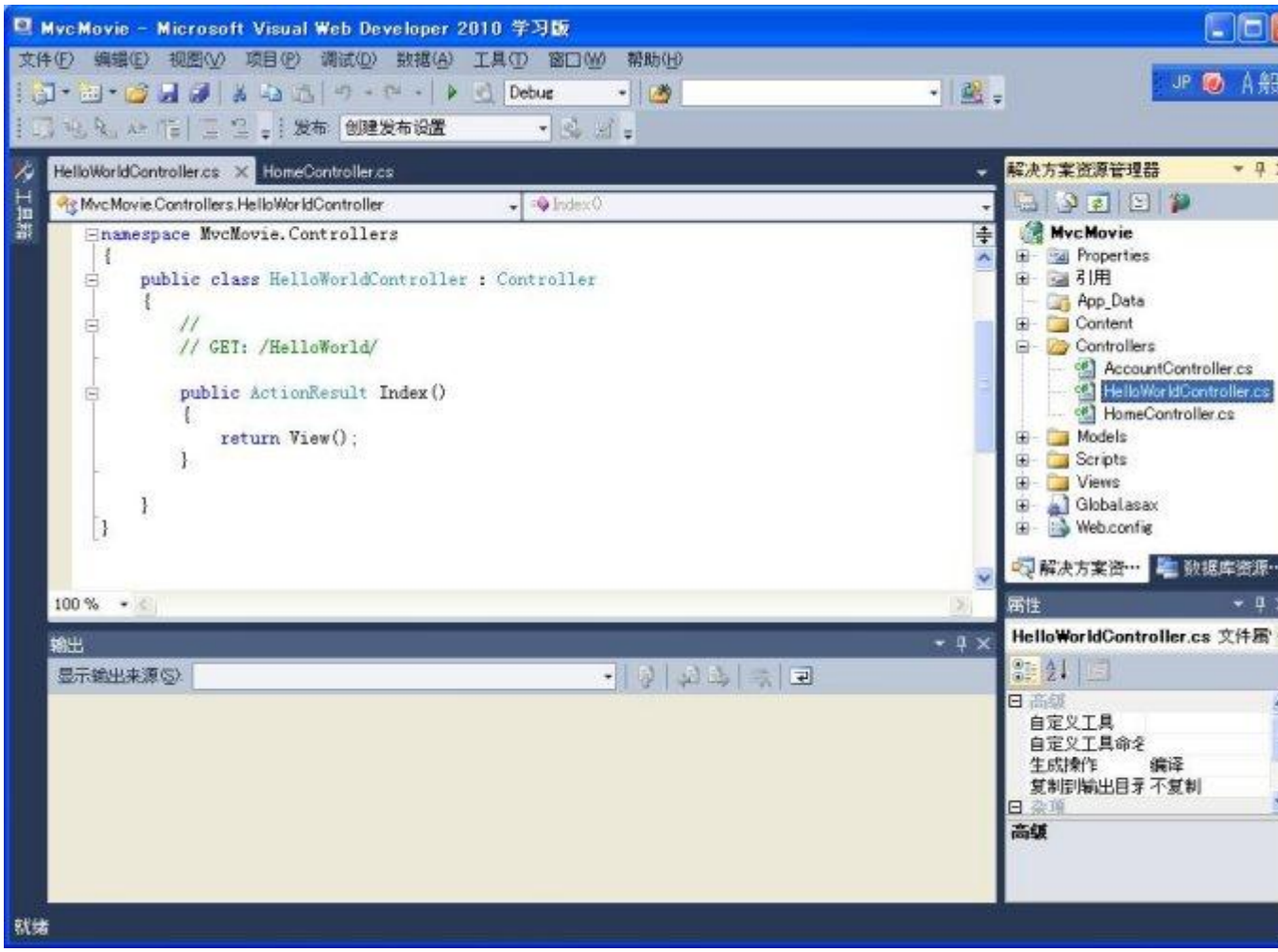


图 2-1 添加控制器

在弹出的“添加控制器”对话框中,将控制器命名为“HelloWorldController”,然后点击添加按钮，如图 2-2 所示。

图 2-2 命名控制器

观察解决方案资源管理器中新增加了一个文件，名字为 HelloWorldController.cs，并且该文件呈打开状态，如图 2-3 所示。



修改打开的 HelloWorldController.cs 文件，在 HelloWorldController 类中，创建如代码清单 2-1 中所示的两个方法，控制器将返回一个 HTML 格式的字符串。

代码清单 2-1 在控制器中创建方法

```
public class HelloWorldController : Controller
{
    //
```

```
        // GET: /HelloWorld/

        public string Index()
        {
            return "这是我的<b>默认</b>action...";
        }
        //
        // GET: /HelloWorld/Welcome/
        public string WelCome()
        {
            return "这是我的 Welcome 方法...";
        }
    }
```

在这个修改后的 HelloWorldController 控制器中，第一个方法名为 Index。现在让我们从浏览器中调用该方法。运行应用程序（按 F5 键或 Ctrl+F5 键）,在打开的浏览器中的地址栏后面，添加“HelloWorld”路径（譬如，在我的计算机上，浏览器中地址为 http://localhost:4423/HelloWorld），画面显示如图 2-4 所示。由于在 Index 方法中，直接返回了一个 HTML 格式的字符串，所以在浏览器中将该字符串显示出来。

图 2-4 HelloWorldController 控制器中 Index 方法的运行结果

在 ASP.NET MVC 中，可以根据浏览器中的输入地址来调用不同的控制器或控制七种不同的方法。ASP.NET MVC 的默认的映射逻辑使用如下所示的格式来决定应该调用什么控制器或控制器中的什么方法。

```
/[Controller]/[ActionName]/[Parameters]
```

URL 地址的第一部分决定调用哪个控制器类，所以“*/HelloWorld*”映射到 HelloWorldController 控制器类。第二部分决定调用控制器中的哪个方法。所以“/HelloWorld/Index”将会调用 HelloWorldController 控制器类的 Index 方法。由于 Index 方法是控制器类的默认方法(可以另外指定控制器类的默认方法),所以也可只输入“/HelloWorld”来调用该方法。

在浏览器的地址栏中，输入“http://localhost:xxxx/HelloWorld/Welcome”，将会调用 HelloWorldController 控制器类的 Welcome 方法，该方法返回“这是我的 Welcome 方法...”文字，所以浏览器中显示该文字，如图 2-5 所示。

图 2-5 HelloWorldController 控制器中 Welcome 方法的运行结果

接下来，让我们修改 Welcome 方法，以便在 URL 地址栏中可以传递一些参数给该方法（例如：/HelloWorld/Welcome?name=Scott&numtimes=4）。修改后的代码如下所示。注意这里我们使用了 C#的可选参数，当 URL 地址中没有使用 numtimes 参数时，该参数被默认设定为 1。

```
public string Welcome(string name,int numTimes=1)
{
   return HttpUtility.HtmlEncode("Hello " + name + ",NumTimes is:" + numTimes);
}
```

运行该应用程序，在浏览器中输入“http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4”，运行结果显示如图 2-6 所示。浏览器自动将 URL 地址栏中的参数映射成 Welcome 方法中的传入参数。

图 2-6　在 Welcome 方法中使用参数

到现在为止，我们展示了 MVC 中的“VC”（视图与控制器）部分的工作机制，控制器返回 HTML 字符串。很显然大多数情况下你不想让控制器直接返回 HTML 字符串，因为那样的话编码起来就太麻烦了。所以我们需要使用不同的视图模板文件来帮助生成 HTML 格式的页面文件，在下一节中让我们来看一下如何在 ASP.NET MVC3 中使用视图。

ASP.NET MVC3 快速入门-第三节 添加一个视图

(2011-02-26 18:58:25)

转载

标签： 分类： ASP.NETMVC3

视图

模板

应用程序

控制器

3.1 添加一个视图

在本节中我们修改 HelloWorldController 类，以便使用视图来向客户端展示 HTML 格式的响应结果。

我们使用 ASP.NET MVC3 中新增的 Razor 视图引擎来创建视图。Razor 视图模板文件的后缀名为.cshtml,它提供了一种简洁的方式来创建 HTML 输出流。Razor 视图大大减少了在书写视图模板文件时所需要输入的字符，提供了一个最快捷，最简便的编码方式。

这里，我们在 HelloWorldController 类的 Index 方法中添加使用一个视图。在修改前的 Index 方法中返回一个字符串，我们修改这个方法来使它返回一个视图，代码如下所示。

```
public ActionResult Index()
{
      return View();
}
```

这段代码表示 Index 方法使用一个视图模板来在浏览器中生成 HTML 格式的页面文件。接着，让我们来添加一个 Index 方法所使用的视图模板。在 Index 方法中点击鼠标右键，然后点击“添加视图”，将会弹出一个“添加视图”对话框。

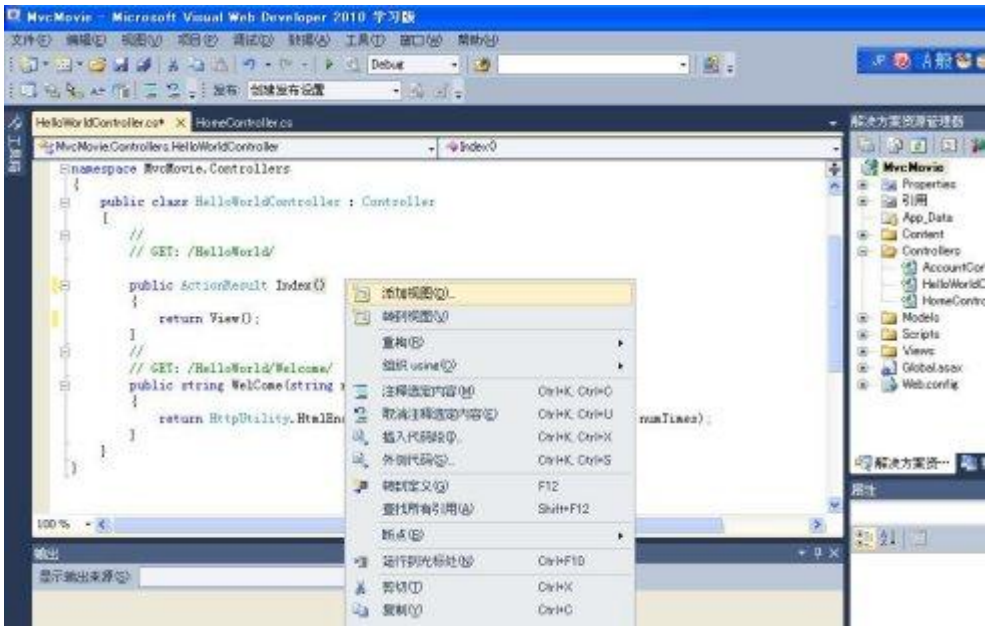


图 3-1 添加视图

图 3-2　添加视图对话框

在该对话框中，不做任何修改，直接点击添加按钮，观察解决方案资源管理器中，在 MvcMovie 项目下的 Views 文件夹下创建了一个 HelloWorld 文件夹，并且在该文件夹中创建了一个 Index.cshtml 文件，同时该文件呈打开状态，如图 3-3 所示。

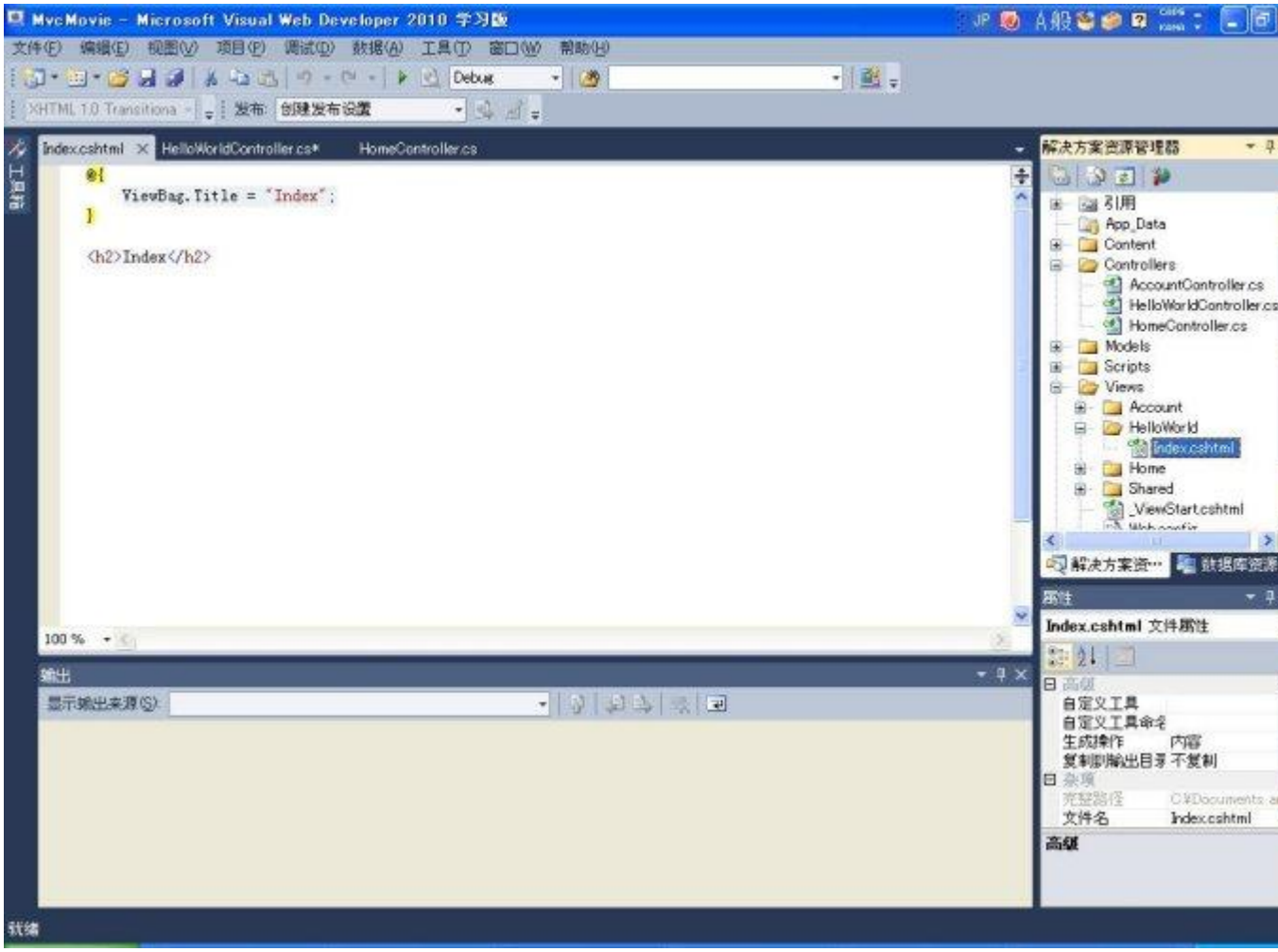


图 3-3　视图模板文件被创建并呈打开状态

让我们在该文件中追加一些文字，代码如代码清单 3-1 所示。

代码清单 3-1 Index.cshtml 视图模板文件

```
@{
    ViewBag.Title = "首页";
}
<h2>首页</h2>
<p>这是我的第一个视图模板</p>
```

运行应用程序，输入地址“http://localhost:xxxx/HelloWorld”。由于在 Index 方法中并没有做任何事情，只是简单地一行代码—“return View()”,该行代码表示我们使用一个视图模板文件来在浏览器中展示响应结果。因为我们并没有显式指定使用哪个视图模板文件，所以使用了默认的 Views 文件夹下的 HelloWorld 文件夹下的 Index.cshtml 视图模板文件。该视图模板文件中只有简单的两行文字，在浏览器中的显示结果如图 3-4 所示。

图 3-4　在浏览器中显示视图

看上去还不错，但是请注意，该网页的标题为“首页”,但是网页中的大标题文字却为“我的 MVC 应用程序”，需要修改一下。

3.2 修改视图，修改应用程序的页面布局

首先，让我们修改页面大标题中的“我的 MVC 应用程序”文字。这段文字是所有页面中的公共大标题，在这个应用程序中，虽然所有页面中都显示了这个共同的大标题，但只有一处地方对其进行了设置。打开解决方案资源管理器中 Views 文件夹下的 Shared 文件夹下的 _Layout.cshtml 文件。该文件被称为布局页面，位于公有文件夹 Shared 下，被所有其他网页所共用。

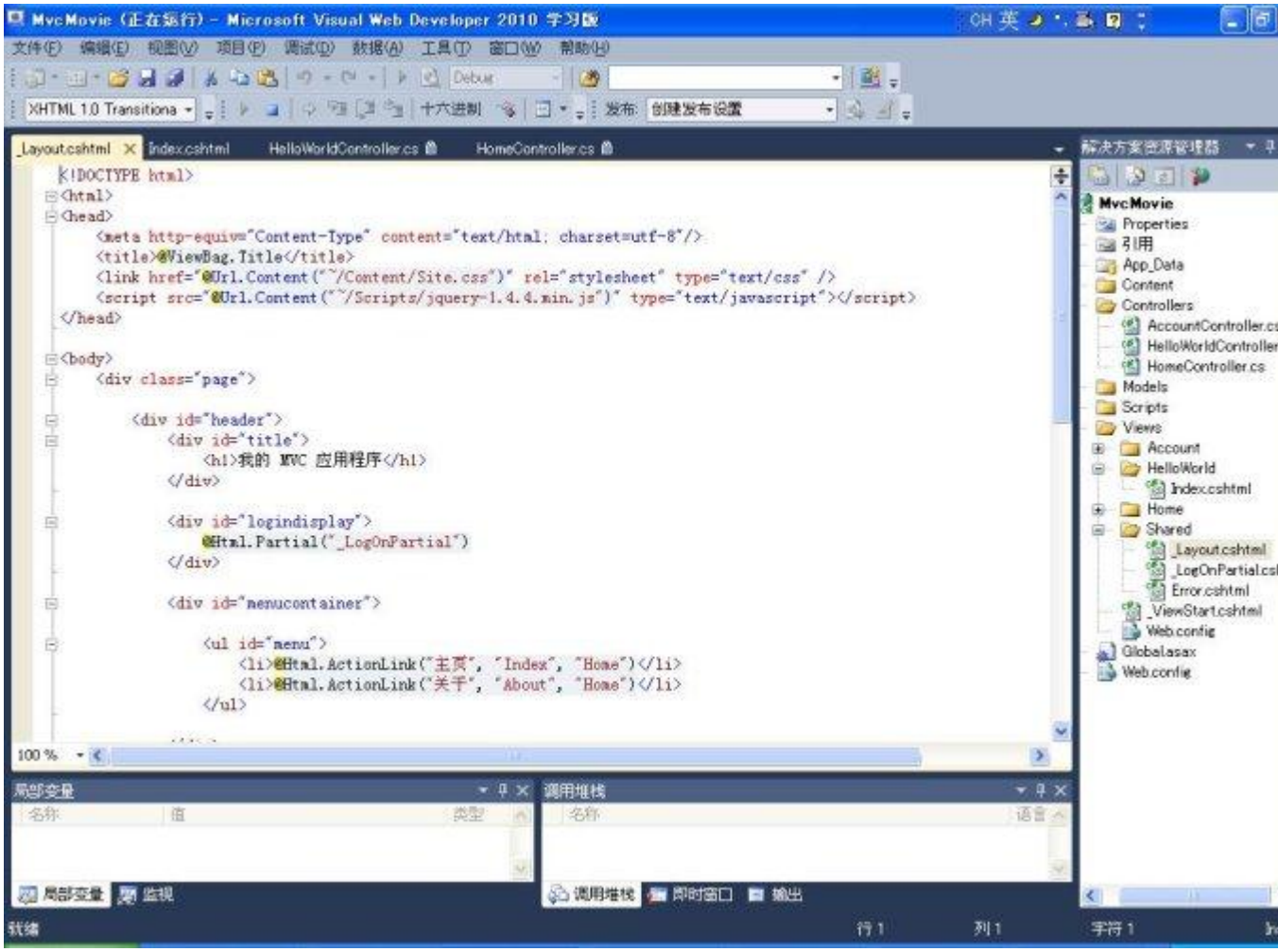


图 3-5　公有布局页面

布局模板页允许你统一在一个地方指定整个 Web 应用程序或 Web 网站的所有 HTML 页面的布局方法。请注意文件底部的“@RenderBody()”代码行。@RenderBody()是一个占位符，代表了所有你创建出来的实际应用的视图页面，在这里统一指定。将布局模板文件中的“我的 MVC 应用程序”修改为“我的 MVCMovie 应用程序”。代码如下所示。

```
<div id="title">
    <h1>我的 MVCMovie 应用程序</h1>
</div>
```

运行应用程序，注意网页中的大标题被修改为“我的 MVCMovie 应用程序”。点击“关于”链接，你可以看见“关于”页面中的大标题也被修改为“我的 MVCMovie 应用程序”。由此可以看出一旦修改了布局页面中的某处地方，该修改将会被应用到所有页面中。

图 3-6 在布局页面中修改了网页中显示的大标题

完整的_Layout.cshtml 文件中的代码如代码清单 3-2 所示。

代码清单 3-2 _Layout.cshtml 文件中的完整代码

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet"
   type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")"
   type="text/javascript"></script>
</head>
<body>
    <div class="page">
        <div id="header">
            <div id="title">
                <h1>我的 MVCMovie 应用程序</h1>
            </div>
            <div id="logindisplay">
                @Html.Partial("_LogOnPartial")
            </div>
            <div id="menucontainer">
                <ul id="menu">
                    <li>@Html.ActionLink("主页", "Index", "Home")</li>
                    <li>@Html.ActionLink("关于", "About", "Home")</li>
```

```
                </ul>
            </div>
        </div>
        <div id="main">
            @RenderBody()
            <div id="footer">
            </div>
        </div>
    </div>
</body>
</html>
```

现在，让我们修改 Index 视图页面的标题。

打开 Views 文件夹下的 HelloWorld 文件夹下的 Index.cshtml 文件。这里我们修改两处地方：首先，修改浏览器中的标题，然后修改<h2>标签中的小标题文字。修改后代码如代码清单 3-3 所示。

代码清单 3-3 修改后的 Index.cshtml 视图模板文件

```
@{
    ViewBag.Title = "电影清单";
}
<h2>我的电影清单</h2>
<p>这是我的第一个视图模板</p>
```

ViewBag 对象的 Title 属性代表了显示该页面时的浏览器中的标题文字。让我们回头看一下布局模板文件，在该文件的<head>区段中的<title>标签中使用了这个值来作为浏览器中的网页标题。同时，通过这种方法，你可以很容易地在你的视图模板文件与布局模板文件之间进行参数的传递。

运行应用程序，在地址栏中输入“http://localhost:xxxx/HelloWorld”，注意浏览器中的网页标题，页面中的小标题文字都变为修改后的标题文字（如果没有发生变化的话，则可能你的网页被缓存住了，可以按 Ctrl+F5 键来在重新刷新页面时取消缓存）。

同时也请注意_Layout.cshtml 文件中的占位符中的内容被替换成了 Index.cshtml 视图模板中的内容，所以浏览器中展示的是一个单一的 HTML 文件。浏览器中的运行结果如图 3-7 所示。

图 3-7 修改了标题后的 Index 视图模板文件

此处，我们的数据（“这是我的第一个视图模板”文字）是被直接书写在文件中的，也就是说我们使用到了 MVC 应用程序的“V”(视图 View)与“C”(控制器 Controller)。接下来，我们讲解一下如何创建一个数据库并从该数据库中获取模型数据。

3.3 将控制器中的数据传递给视图

在我们使用数据库并介绍模型之前，首先我们介绍一下如何将控制器中的信息传递给视图。浏览器接收到一个 URL 请求后，将会调用控制器类来进行响应。你可以在控制器类中进行对接收到的页面参数进行处理的代码，你可以在控制器类中书写从数据库中获取数据的代码，你也可以在控制器类中书写代码来决定返回给客户端什么格式的响应文件。控制器可以利用视图模板文件来生成 HTML 格式的响应文件并显示在浏览器中。

控制器类负责提供视图模板文件在生成 HTML 格式的响应文件时所需要的任何数据或对象。一个视图模板文件不应该执行任何业务逻辑，也不应该直接和数据库进行交互。它只能和控制器类进行交互，获取控制器类所提供给它的数据，这样可以使你的代码更加清晰，容易维护。

现在在我们的应用程序中，HelloWorldController 控制器类中的 Welcome 方法带有两个参数—name 与 numTimes,Welcome 方法直接向浏览器输出这两个参数的参数值。这里，我们修改该方法使其不再直接输出数据，而是使用一个视图模板。该视图模板将生成一个动态的响应流，这意味着我们需要将数据从控制器类传递给视图以便利用该数据来生成该响应流。我们在该控制器类中将视图模板所需要的数据送入一个 ViewBag 对象中，该对象可以被视图模板直接接收。

打开 HelloWorldController.cs 文件，修改 Welcome 方法，在该方法中为 ViewBag 对象添加一个 Message 属性与 NumTimes 属性，并且将属性值分别设定为经过处理后的 name 参数值与 numTimes 参数值。ViewBag 对象是一个动态对象，你可以为它添加任何属性并赋上属性值。在未赋值之前该属性是不生效的，直到你赋值为止。修改后的 HelloWorldController.cs 文件中的代码如代码清单 3-4 所示。

代码清单 3-4 修改后的 HelloWorldController.cs 文件

```
using System.Web;
using System.Web.Mvc;
namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/
        public ActionResult Index()
        {
            return View();
        }
        //
        // GET: /HelloWorld/Welcome/
        public ActionResult Welcome(string name, int numTimes = 1)
        {
            ViewBag.Message = "Hello " + name;
            ViewBag.NumTimes = numTimes;
            return View();
        }
    }
}
```

现在 ViewBag 对象中已经包含了数据，它将被自动传递给视图。

接下来，我们需要创建一个 Welcome 视图模板。在“调试”菜单中，点击“生成 MvcMovie”将应用程序进行编译，如图 3-8 所示。

图 3-8 编译应用程序

接下来，在 Welcome 方法中点击鼠标右键，然后点击“添加视图”，弹出对话框如图 3-9 所示。

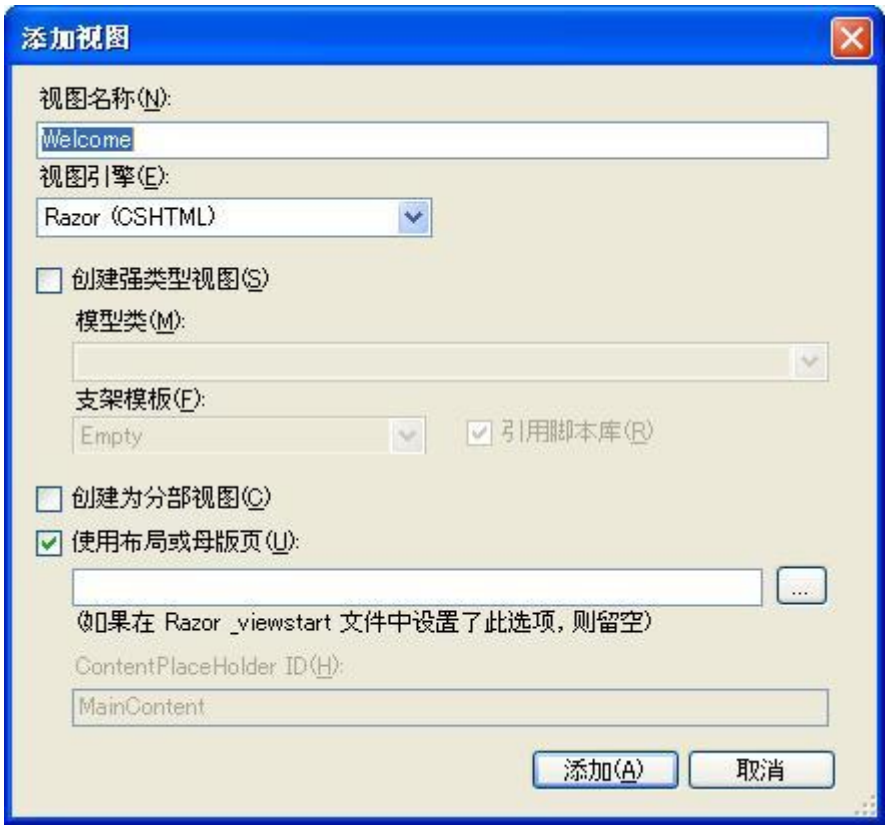


图 3-9 为 Welcome 方法添加视图

在该对话框中不做任何修改，直接点击添加按钮, View 文件夹下的 HelloWorld 文件假种自动被创建了一个 Welcome.cshtml 文件，打开该文件，在<h2>元素下添加代码，让浏览器显示 URL 地址中传入的 name 参数中设定的文字，显示次数等于 URL 地址中传入的 numTimes 参数中设定的次数。修改后的 Welcome.cshtml 文件中的代码如代码清单 3-5 所示。

代码清单 3-5 修改后的 Welcome.cshtml 文件

```
@{
    ViewBag.Title = "Welcome";
}
<h2>Welcome</h2>
<ul>
@for (int i = 0; i < ViewBag.NumTimes; i++)
{
    <li>@ViewBag.Message</li>
}
</ul>
```

运行应用程序，并且在地址栏中输入“http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4”，该地址栏中的页面参数将会自动传递给控制器。控制器将会把这些参数值放入 ViewBag 对象中并且传递给视图。视图再在浏览器中显示这些数据。

图 3-10 视图中显示从控制器类中传递过来的数据

这里，我们使用了模型“M”的一种方式，但是不是数据库的方式。在下一节中，我们将创建一个数据库，并且介绍如何对该数据库中的数据进行处理。

ASP.NET MVC3 快速入门-第四节 添加一个模型

(2011-02-27 20:04:47)

转载

标签： 分类： ASP.NETMVC3

数据库

对话框

杂谈

在本节中我们将追加一些类来管理数据库中的电影。这些类将成为我们的 MVC 应用程序中的“模型”部分。

我们将使用一个.NET Framework 的被称之为“Entity Framework”的数据访问技术来定义这些模型类，并使用这些类来进行操作。Entity Framework（通常被简称为“EF”）支持一个被称之为“code-first”的开发范例。Code-first 允许你通过书写一些简单的类来创建模型对象。你可以通过访问这些类的方式来访问数据库，这是一种非常方便快捷的开发模式。

4.1 利用 NuGet 来安装 EFCodeFirst

我们可以利用 NuGet 包管理器（安装 ASP.NET MVC3 时会自动安装）来把 EFCodeFirst 类库添加到我们的 MvcMovie 工程中。这个类库使得我们可以直接使用 code-first。点击“工具”菜单下的“Library Package Manager”子菜单下的“Add Library Package Reference”菜单选项，如图 4-1 所示。

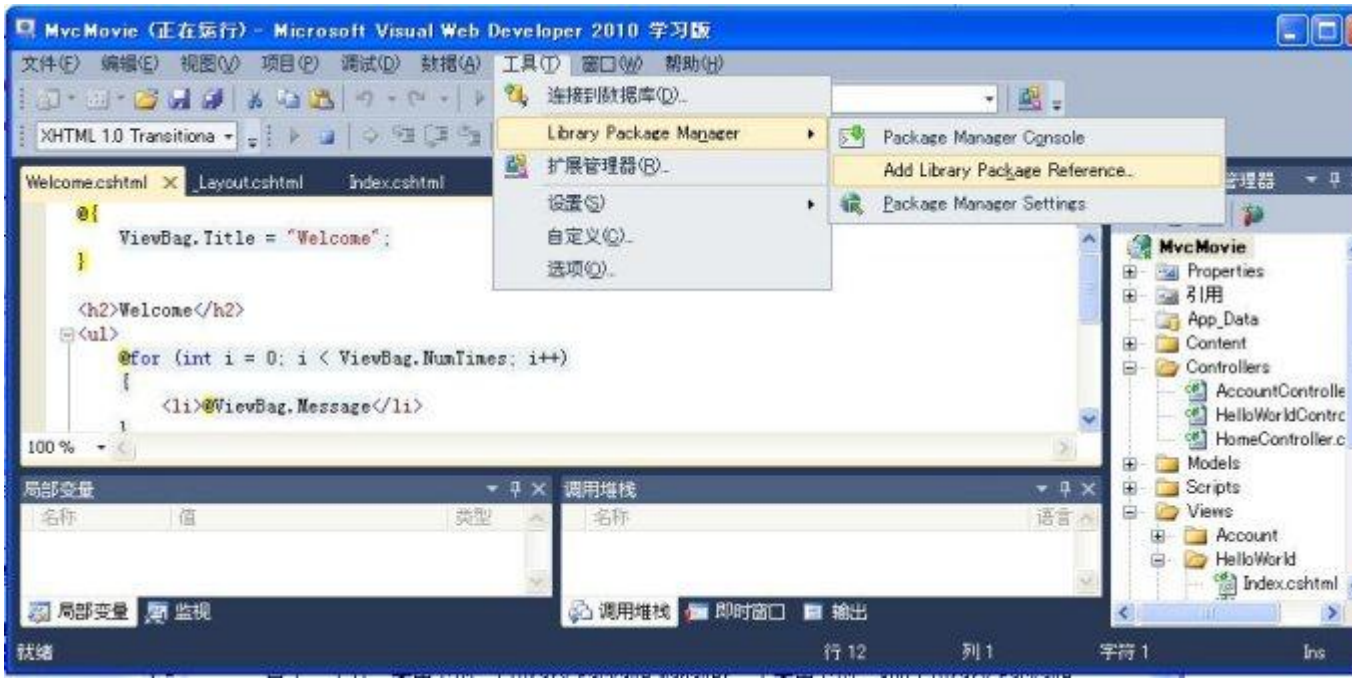


图 4-1 使用 NuGet 包管理器

点击“Add Library Package Reference”菜单选项后，将会弹出一个对话框，标题为“Add Library Package Reference”，如图 4-2 所示。

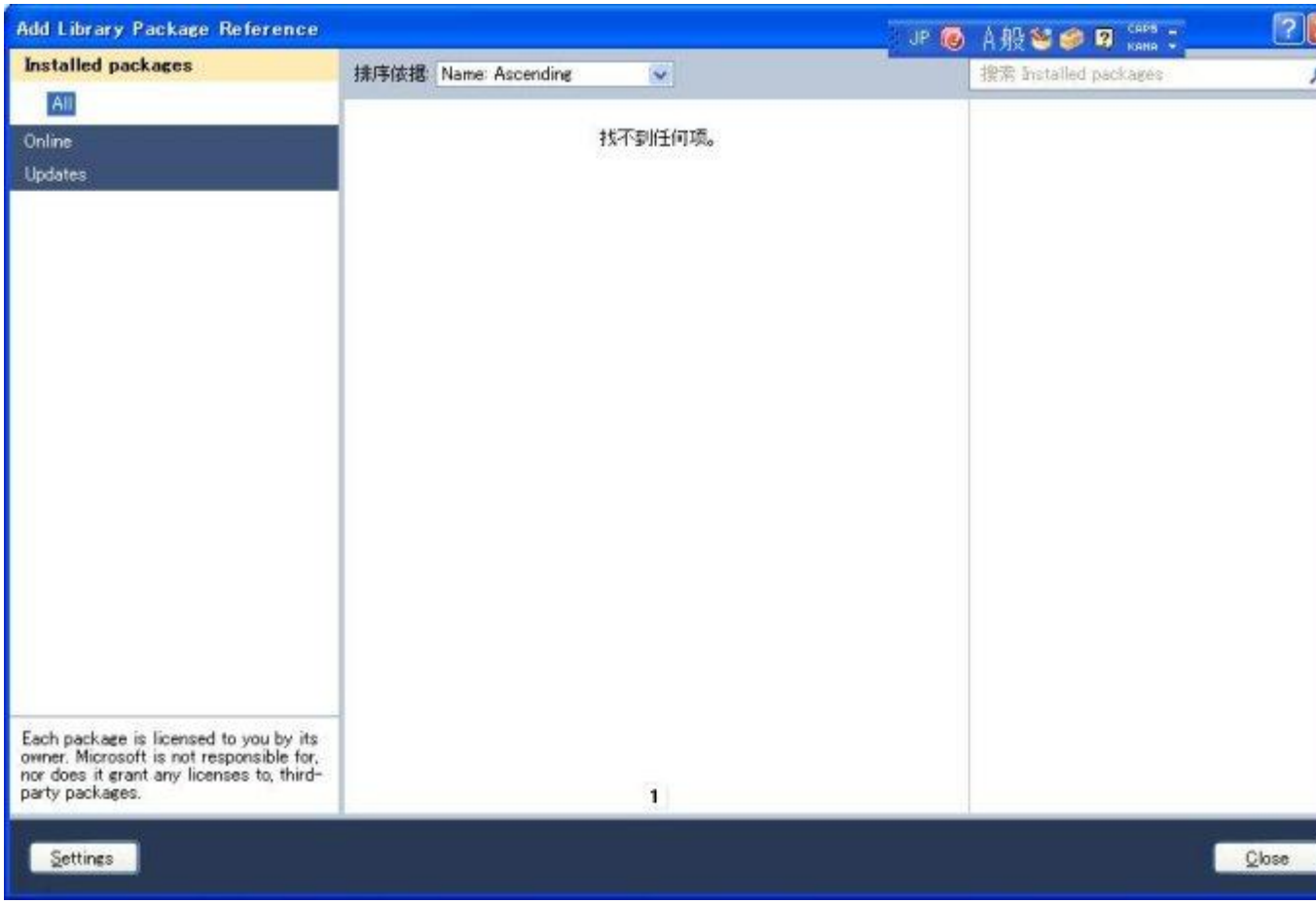


图 4-2 “Add Library Package Reference”对话框

默认状态下，左边的“All”选项处于选择状态。因为还没有安装任何包，所以右边面板中显示“找不到任何项”。点击左边面板中的“online”选项，NuGet 包管理器将会在服务器上检索所有当前能够获取的包，如图 4-3 所示。

图 4-3 NuGet 包管理器正在检索包信息

服务器上有几百个当前能够获取的包，现在我们只关注 EFCodeFirst 包。在右上角的搜索输入框中输入“EFCode”。在检索结果中，选择 EFCodeFirst 包，并且点击 Install 按钮安装包，如图 4-4 所示。

图 4-4 选择 EFCodeFirst 包并安装

点击了 install 按钮后，会弹出一个接受许可证窗口，如图 4-5 所示，在这个窗口中必须要点击“I Accept”按钮，接受许可证条款，安装才能继续进行。

图 4-5 接受许可证窗口

安装完毕后，点击 close 按钮。我们的 MvcMovie 工程中会自动加载 EntityFramework 程序集，其中包含了 EFCodeFirst 类库。

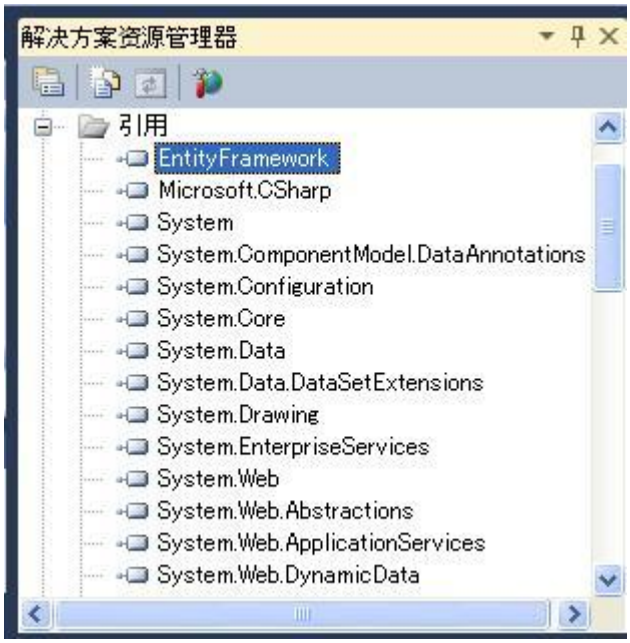


图 4-6　安装完毕后 EntityFramework 程序集被自动加载

4.2 添加模型类

在解决方案资源管理器中，鼠标右击 Models 文件夹，点击“添加”菜单下的“类”，如图 4-7 所示。

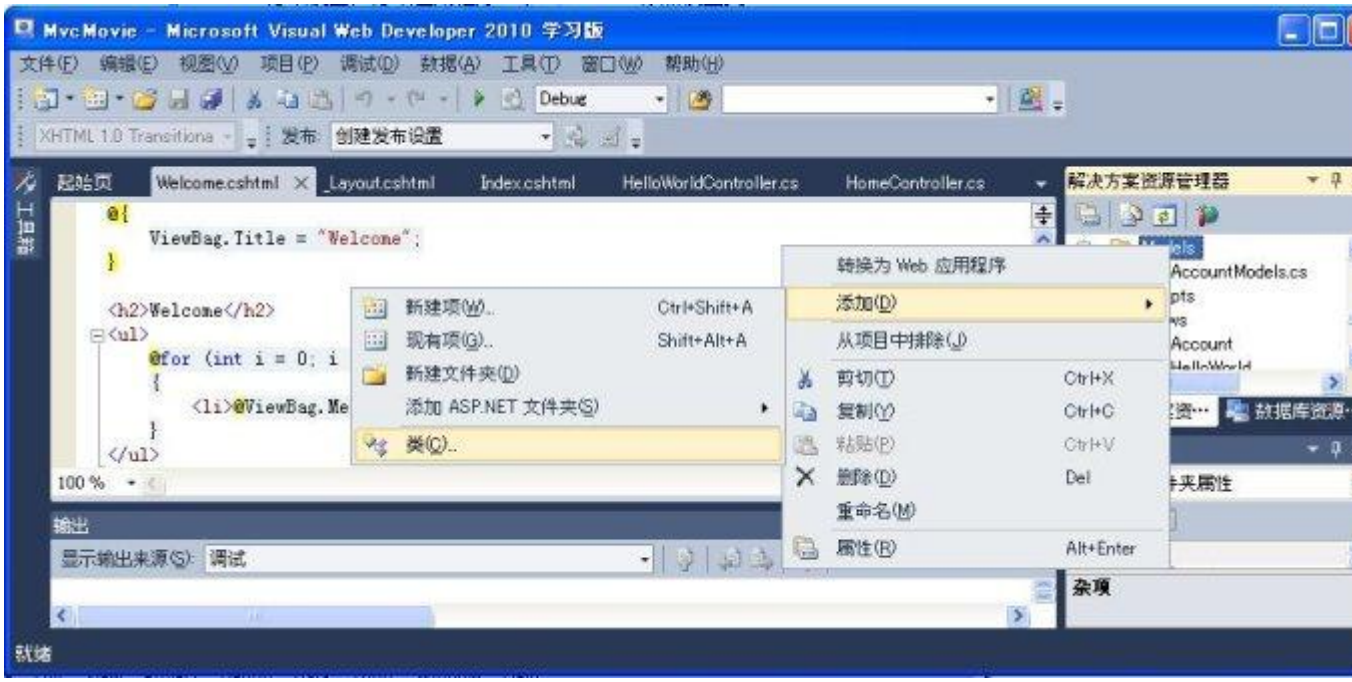


图 4-7 添加模型类

点击“类”菜单项后，会弹出“添加新项”对话框，在该对话框中将类名命名为“Movie”,如图 4-8 所示。

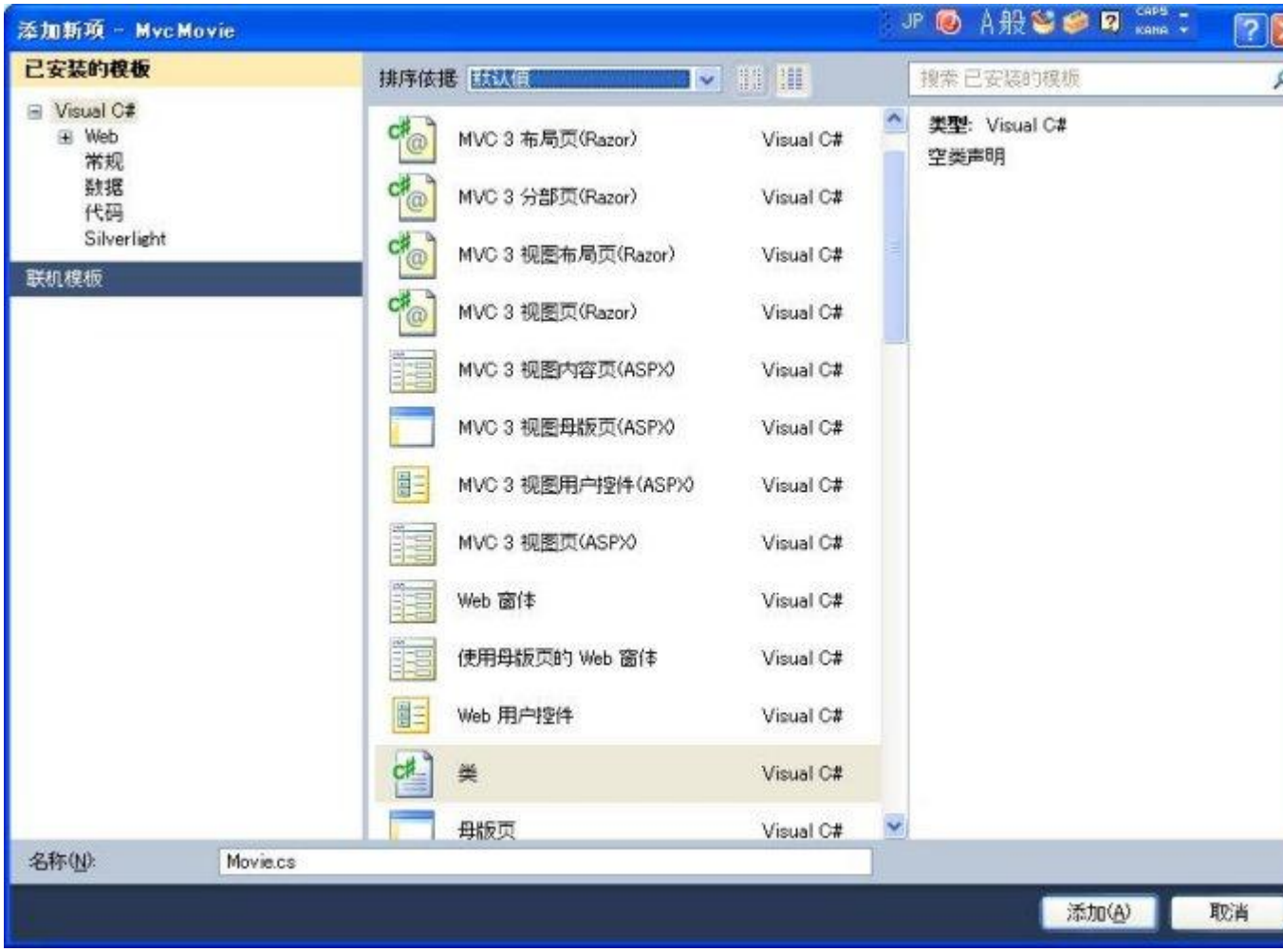


图 4-8 在“添加新项”对话框中为类命名

然后点击添加按钮，观察解决方案资源管理器中，Models 文件夹下添加了一个 Movie.cs 类定义文件，并且该文件呈打开状态，如图 4-9 所示。

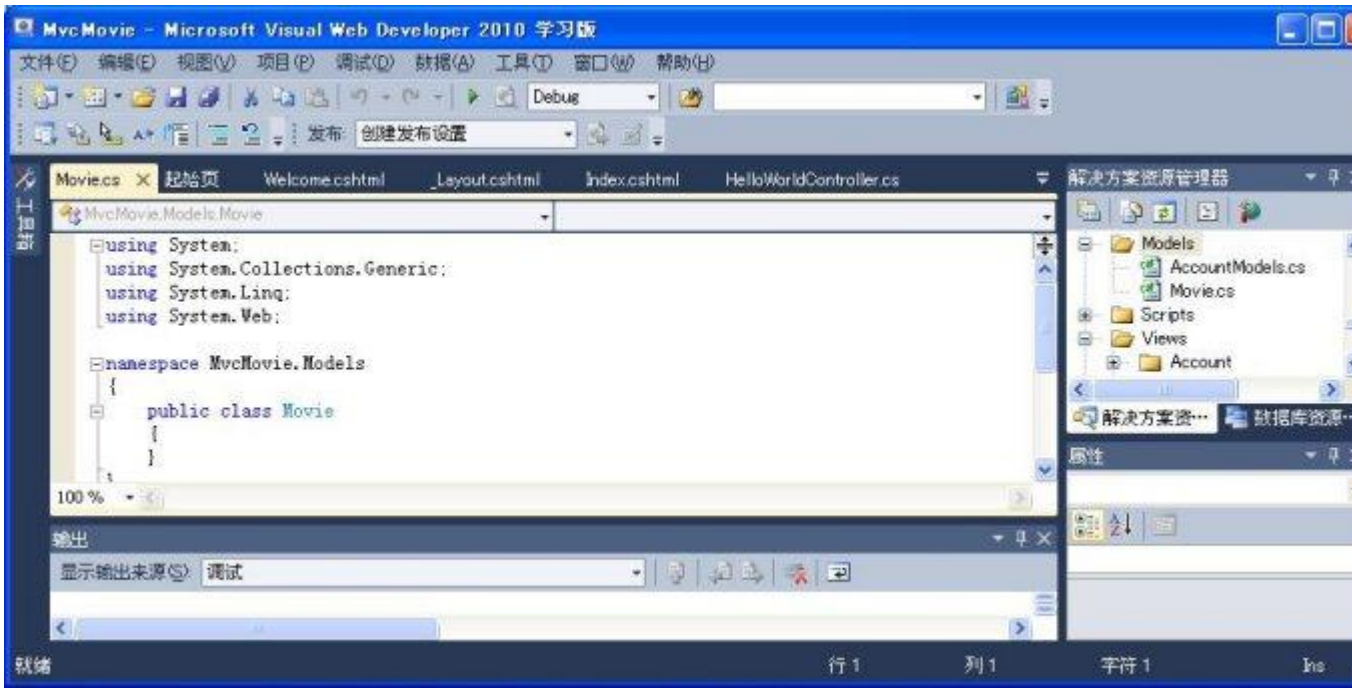


图 4-9 Movie.cs 类定义文件已被添加并呈打开状态

在 Movie.cs 文件中追加如下所示的五个属性。

```
public class Movie
{
        public int ID { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
}
```

我们将利用 Movie 类来代表数据库中的 movie(电影)。每一个 Movie 对象的实例对应于数据表中的一行，Movie 类中的每一个属性被映射到数据表的每一列。

在同一个 Movie.cs 文件中,追加如下所示的 MovieDBContext 类。

```
public class MovieDBContext : DbContext
{
        public DbSet<Movie> Movies { get; set; }
}
```

MovieDBContext 类代表了 Entity Framework 中的 movie 数据库的上下文对象，用来处理数据的存取与更新。MovieDBContext 对象继承了 Entity Framework 中的 DbContext 基础类。为了能够引用 DbContext 类，你需要在 Movie.cs 文件的头部追加如下所示的 using 语句。

```
using System.Data.Entity;
```

完整的 Movie.cs 文件中的代码如代码清单 4-1 所示。

代码清单 4-1 完整的 Movie.cs 文件

```
using System;
```

```
using System.Data.Entity;
namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

如果要从数据库中存取数据，类似以上所示的代码是必须要写的。在下一节中，我们将要创建一个新的 MoviesController 类，用来显示数据库中的数据，并且允许用户创建一个新的 movie(电影)的列表。

ASP.NET MVC3 快速入门-第五节 从控制器访问模型中的数据

(2011-03-05 17:07:04)

标签： 分类： ASP.NETMVC3

视图

控制器

模型

杂谈

5. 1 从控制器访问模型中的数据

在本节中，我们将要创建一个新的 MoviesController 类，并且书写代码来获取数据库中的数据，并通过视图模板来显示在浏览器中。

鼠标右击 Controllers 文件夹，点击“添加”菜单下的“控制器”菜单项，将会弹出一个“添加控制器”对话框，如图 5-1 所示。

图 5-1 添加控制器

在该对话框中将控制其命名为 MoviesController,然后点击添加按钮，该对话框被关闭。观察解决方案资源管理器中，Controllers 文件夹下增加了一个名为 MoviesController.cs 的文件，并且呈打开状态。让我们更新 MoviesController 类中的 Index 方法，以便获取 movie(电影)清单。

这里需要注意的是，为了引用我们前面创建的 MovieDBContext 类，你需要在文件头部追加如下所示的两个 using 语句。

```
using MvcMovie.Models;
using System.Linq;
```

修改 MoviesController 类中的代码为代码清单 5-1 中所示代码。

代码清单 5-1 MoviesController 类中的完整代码

```
using MvcMovie.Models;
using System.Linq;
using System;
using System.Web.Mvc;
namespace MvcMovie.Controllers
{
    public class MoviesController : Controller
    {
        MovieDBContext db = new MovieDBContext();
        public ActionResult Index()
        {
            var movies=from m in db.Movies
                       where m.ReleaseDate>new DateTime(1984,6,1)
                       select m;
            return View(movies.ToList());
        }
    }
}
```

这段代码实施了一个 LINQ 查询来获取 1984 年夏天之后发行的所有电影。我们还需要一个视图模板来显示这个电影清单，所以在 Index 方法内点击鼠标右键，然后点击“添加视图”来添加一个视图。

由于这里我们需要将一个 Movie 类传递给视图，所以在“添加视图”对话框中，与本教程中前几次在该对话框中之行的操作有所不同，前几次我们都是直接点击添加按钮来创建一个空白的视图模板，但是这一次我们想让 Visual Web Developer 为我们自动创建一个具有一些默认处理的强类型的视图，所以我们勾选“创建强类型视图”复选框，在模型类下拉框中选择“Movie(MvcMovie.Models)”（如果模型类中不存在这个类，请先点击调试菜单下的“生成 MvcMovie”生成该类),在支架模板下拉框中选择“List”,最后勾选“引用脚本”复选框，如图 5-2 所示。

图 5-2 添加强类型视图

点击添加按钮，Visual Web Developer 自动生成一个视图，并且自动在视图文件中添加显示电影清单所需要的代码。这里，我们首先用与前面修改 HelloWorld 控制器所用的视图中的标题同样的方法来修改这个 Movies 控制器所用视图中的标题。

代码清单 5-2 为修改后的这个视图中的完整代码。在这段代码中，我们将 releaseDate（发行日期）属性的格式化字符串从原来的“{0：g}”修改为“{0：d}”（长日期修改为短日期），将 Price(票价)属性的格式化字符串从原来的“{0：F}”修改为“{0：c}”（float 类型修改为货币类型）。

另外，将列表标题中的文字全部修改为中文名称。

代码清单 5-2 Movies 控制器所用视图中的完整代码

```
@model IEnumerable<MvcMovie.Models.Movie>
@{
    ViewBag.Title = "电影清单";
}
<h2>我的电影清单</h2>
<p>
    @Html.ActionLink("追加", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>
            电影名称
```

```
            </th>
            <th>
                发行日期
            </th>
            <th>
                种类
            </th>
            <th>
                票价
            </th>
        </tr>
    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.ActionLink("编辑", "Edit", new { id=item.ID }) |
                @Html.ActionLink("查看明细", "Details", new { id=item.ID }) |
                @Html.ActionLink("删除", "Delete", new { id=item.ID })
            </td>
            <td>
                @item.Title
            </td>
            <td>
                @String.Format("{0:d}", item.ReleaseDate)
            </td>
            <td>
                @item.Genre
            </td>
            <td>
                @String.Format("{0:c2}", item.Price)
            </td>
        </tr>
    }
    </table>
```

5.2 强类型模型与@model 关键字

在本教程的前文中，我们介绍了一个控制器可以使用 ViewBag 对象来将数据或对象传递到视图模板中。ViewBag 是一个动态对象，它提供了一种便利的，后期绑定的方法来将信息从控制器传递到视图中。

ASP.NET MVC 也提供了一种利用强类型的方法来将数据或对象传递到视图模板中。这种强类型的方法为你的编码过程提供了很丰富的编辑时的智能输入提示信息与非常好的编译时的检查。接下来我们将结合这种方法与我们的 Movies 控制器（MoviesController）与视图模板（Index.cshtml）一起使用。

请注意在我们的 MoviesController 控制器的 Index 方法中，我们在调用 View()方法时传入了一个参数，代码如下所示。

```
public class MoviesController : Controller
{
    MovieDBContext db = new MovieDBContext();
    public ActionResult Index()
    {
        var movies = from m in db.Movies
                     where m.ReleaseDate > new DateTime(1984, 6, 1)
                     select m;

        return View(movies.ToList());
    }
}
```

请注意如下这一行代码表示将一个 movies 列表从控制器传递到了视图中。

```
return View(movies.ToList());
```

通过在视图模板文件的头部使用@model 语句，视图模板可以识别传入的参数中的对象类型是否该视图模板所需要的对象类型。请记住当我们在创建这个 Movies 控制器所使用的模板时，我们在“添加视图”对话框中勾选了“创建强类型视图”复选框，在模型类下拉框中选择了“Movie(MvcMovie.Models)”，在支架模板下拉框中选择了“List”。所以 Visual Web Developer 自动在我们的视图模板文件的第一行中添加了如下所示的语句。

```
@model IEnumerable<MvcMovie.Models.Movie>
```

@model 关键字允许我们在视图模板中直接访问在控制器类中通过使用强类型的“模型”而传递过来的 Movie 类的列表。例如，在我们的 Index.cshtml 视图模板中，我们可以通过 foreach 语句来遍历这个强类型的模型，访问其中的每一个 Movie 对象。代码如下所示。

```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("编辑", "Edit", new { id=item.ID }) |
            @Html.ActionLink("查看明细", "Details", new { id=item.ID }) |
            @Html.ActionLink("删除", "Delete", new { id=item.ID })
        </td>
        <td>
            @item.Title
        </td>
        <td>
            @String.Format("{0:d}", item.ReleaseDate)
        </td>
        <td>
            @item.Genre
        </td>
        <td>
            @String.Format("{0:c2}", item.Price)
        </td>
    </tr>
```

```
}
```

因为这里的“模型”是强类型的（IEnumerable<Movie>),所以在循环遍历时“模型”中的每一个项目（“item”）也是一个强类型的 Movie 对象，可以直接访问该对象的每一个属性。同时这也意味着我们可以在编译时检查我们的代码，同时在书写代码时也可以使用代码编辑器提供的智能输入提示信息，如图 5-3 所示。

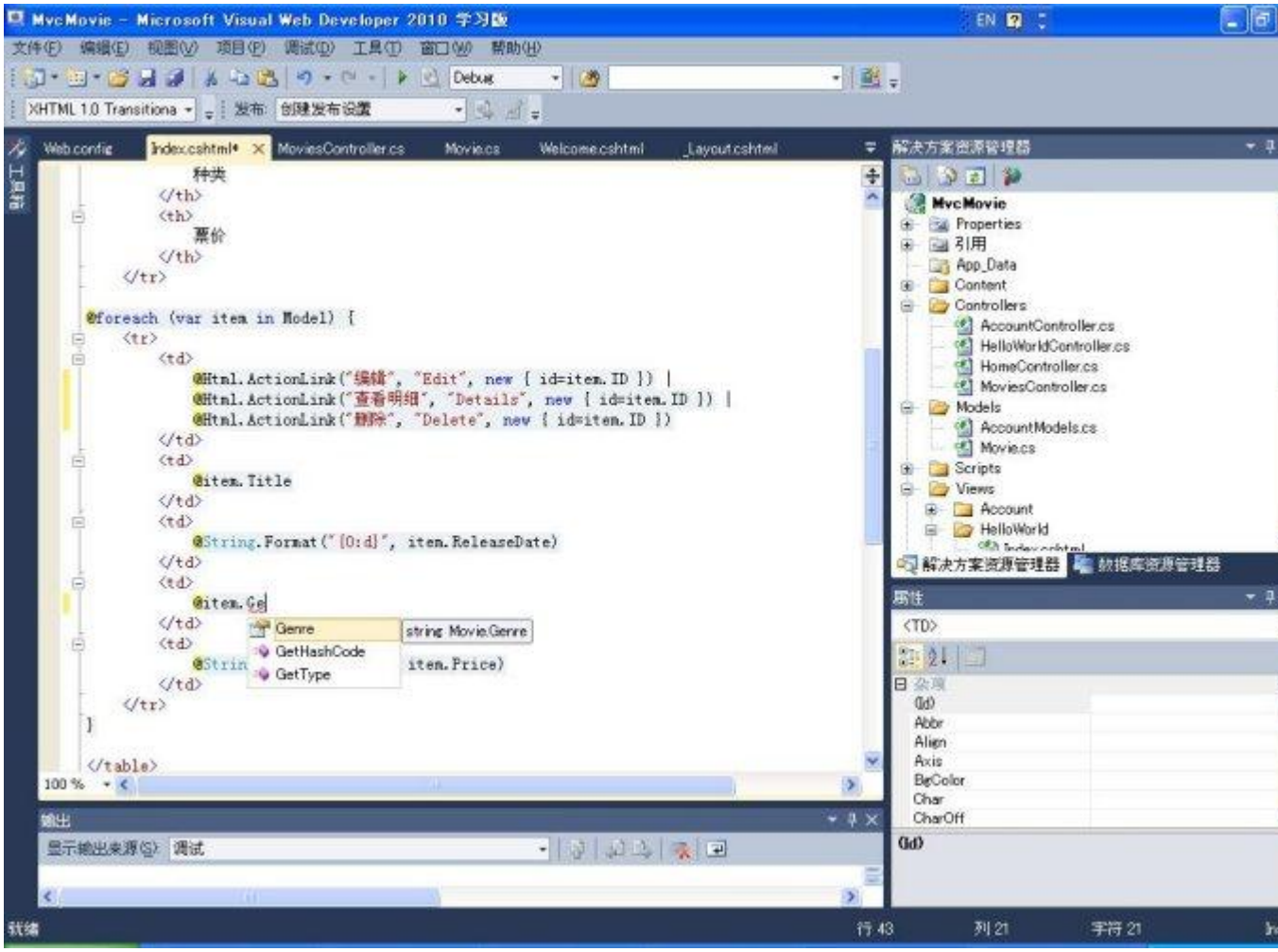


图 5-3 可以使用强类型“模型”所带来的智能输入提示信息

5.3 与 SQL Server Express 结合使用

我们在本节前面创建了一个 MovieDBContext 类，用来连接数据库，并将数据库中的记录映射到 Movie 对象。你也许会问一个问题，怎样定义数据库连接？接下来我们通过在 web.config 文件中增加一些连接信息来定义一个数据库的连接。

打开应用程序根目录下的 Web.config 文件（请注意不是 Views 文件夹下的 Web.config 文件），如图 5-4 所示。

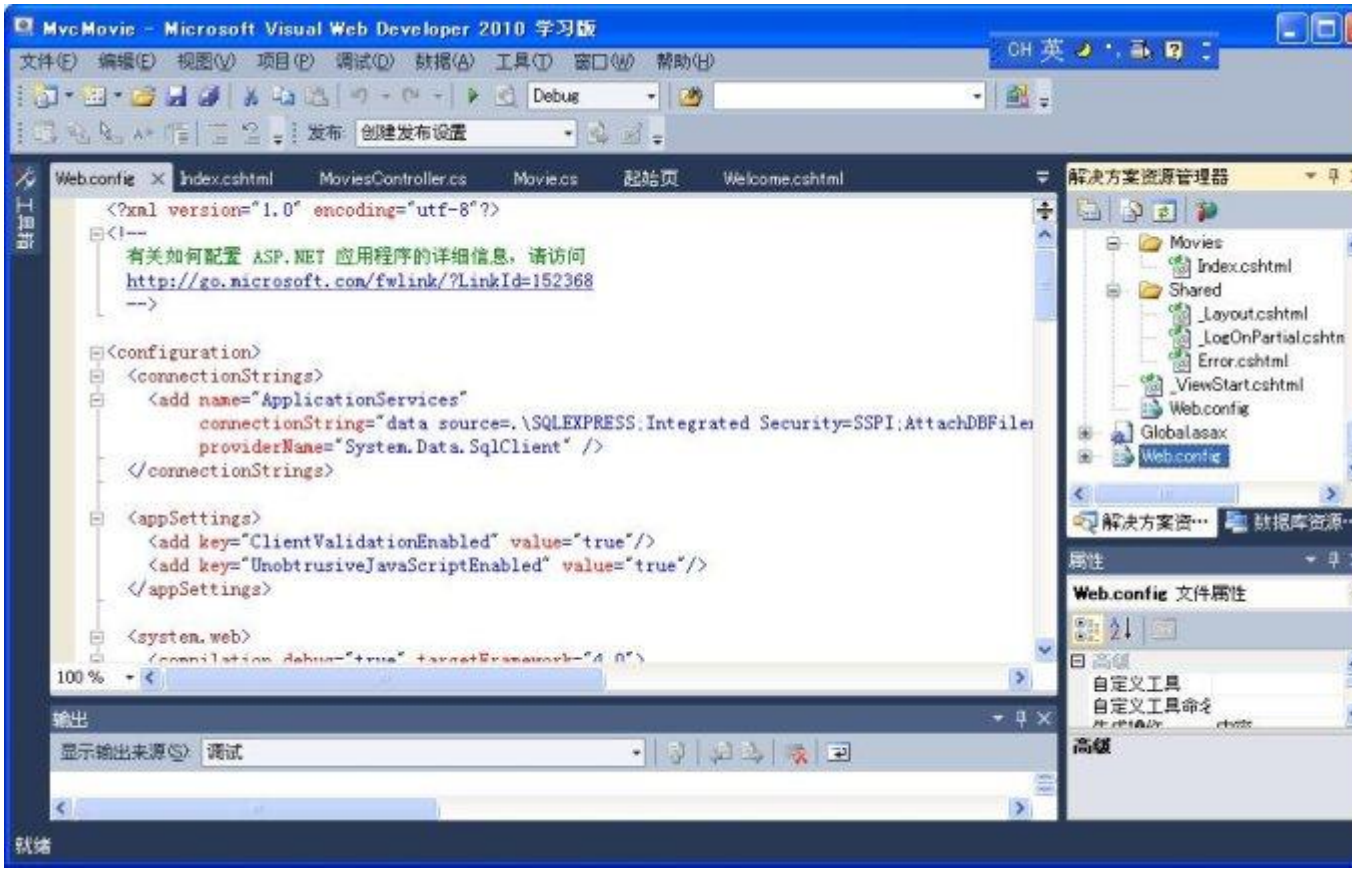


图 5-4 打开应用程序根目录下的 Web.config 文件

在 Web.config 文件的<connectionStrings>元素中追加类似如下所示的连接字符串。

```
<configuration>
  <connectionStrings>
    <add name="ApplicationServices"
         connectionString="data source=.\SQLEXPRESS;Integrated
         Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;
        User Instance=true"
         providerName="System.Data.SqlClient" />
    <add name="MovieDBContext"
         connectionString=" Data Source=.\SQLEXPRESS;
       Initial Catalog=Movies;Persist Security Info=True;
       User ID=aaa;Password=aaaaaaa "
         providerName="System.Data.SqlClient" />
  </connectionStrings>
```

connectionString 属性的值表示我们想要使用 SQL Server Express 的一个本地实例中的 Movies 数据库。当你安装 Visual Web Developer Express 的时候，安装过程中也会同时自动在你的计算机中安装 SQL Server Express,你可以利用它来进行有关数据库的管理工作。

运行应用程序，在浏览器中输入“http://localhost:xxxx/Movies”，浏览器中将会显示一张空的电影列表，如图 5-5 所示。

图 5-5 数据库中没有数据时将默认显示空的列表

EF code-first 如果发现使用我们提供的连接字符串而连接到的数据库中没有“Movies”数据库，它将自动为我们创建一个。你可以在类似“C:\Program Files\Microsoft SQL \MSSQL10.SQLEXPRESS\MSSQL\DATA”这样的 SQL Server 的安装目录下去查看是否该数据库已被创建。

另外请注意，在本教程的前面部分中，我们采用如下所示的代码创建了一个 Movie 模型。

```
using System;
using System.Data.Entity;
namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
```

```
    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

如您所见，当我们第一次使用 MoviesController 控制器类来访问 MovieDBContext 所指向的实例时，Entity Framework 可以自动为你创建一个新的 Movies 数据库，并且将 MovieDBContext 类的 Movies 属性映射到一个新的 Movies 表，并且自动将它创建。这个表中的每一行被映射到一个新的 Movie 类的实例，Movies 表的每一列被映射到 Movie 类的一个属性。

你可以使用 SQL Server Management Studio 来查看使用模型创建出来的数据库与数据表。

在 Windows 的开始菜单中打开 SQL Server Management Studio，并且连接到 Web.config 中所配置的数据库，如图 5-6 所示。

图 5-6　使用 SQL Server Management Studio 连接数据库

点击“连接”按钮进行连接，查看数据库，可以看见 Movies 数据库与数据表已被创建，如图 5-7 所示。

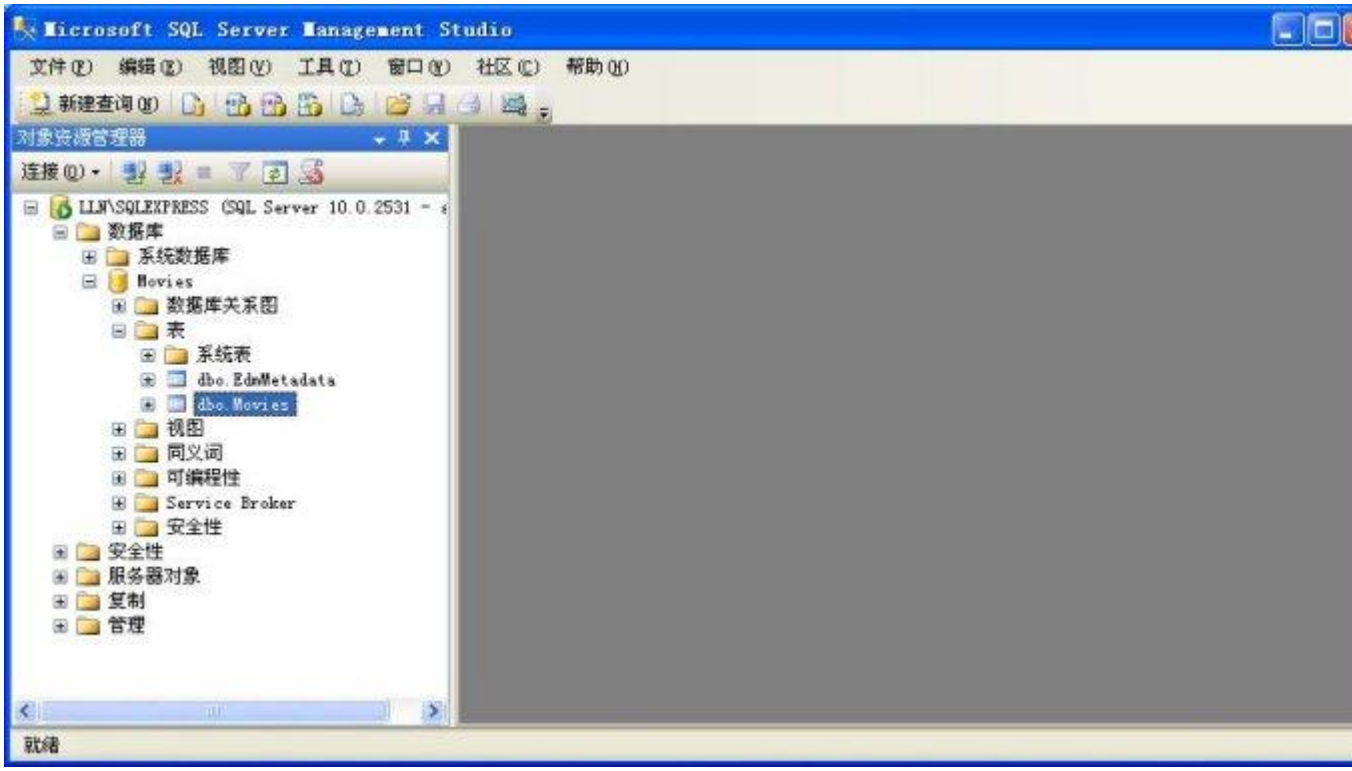


图 5-7 Movies 数据库与数据表已被创建

鼠标右击 Movies 数据表，并且点击“设计”，如图 5-8 所示。

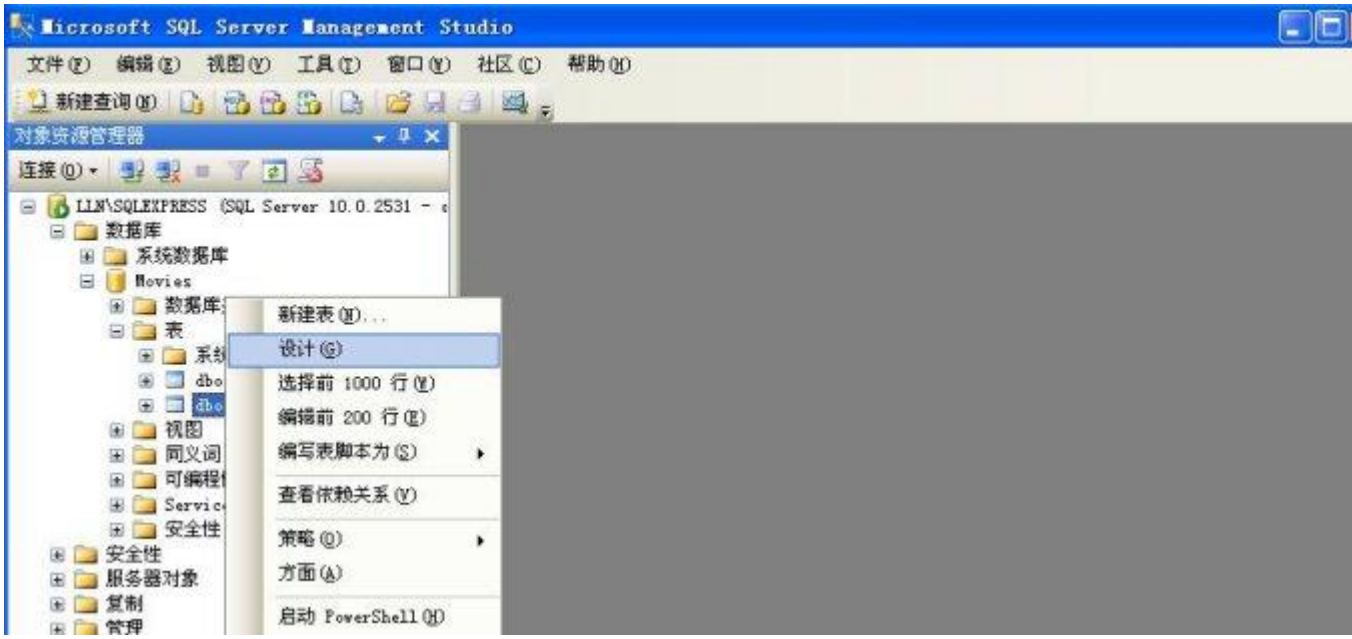


图 5-8 点击“设计”查看 Movies 表的属性

你可以看见 Movies 表中各字段的属性，其中 ID 字段被自动设定为自增长主键，如图 5-9 中所示。

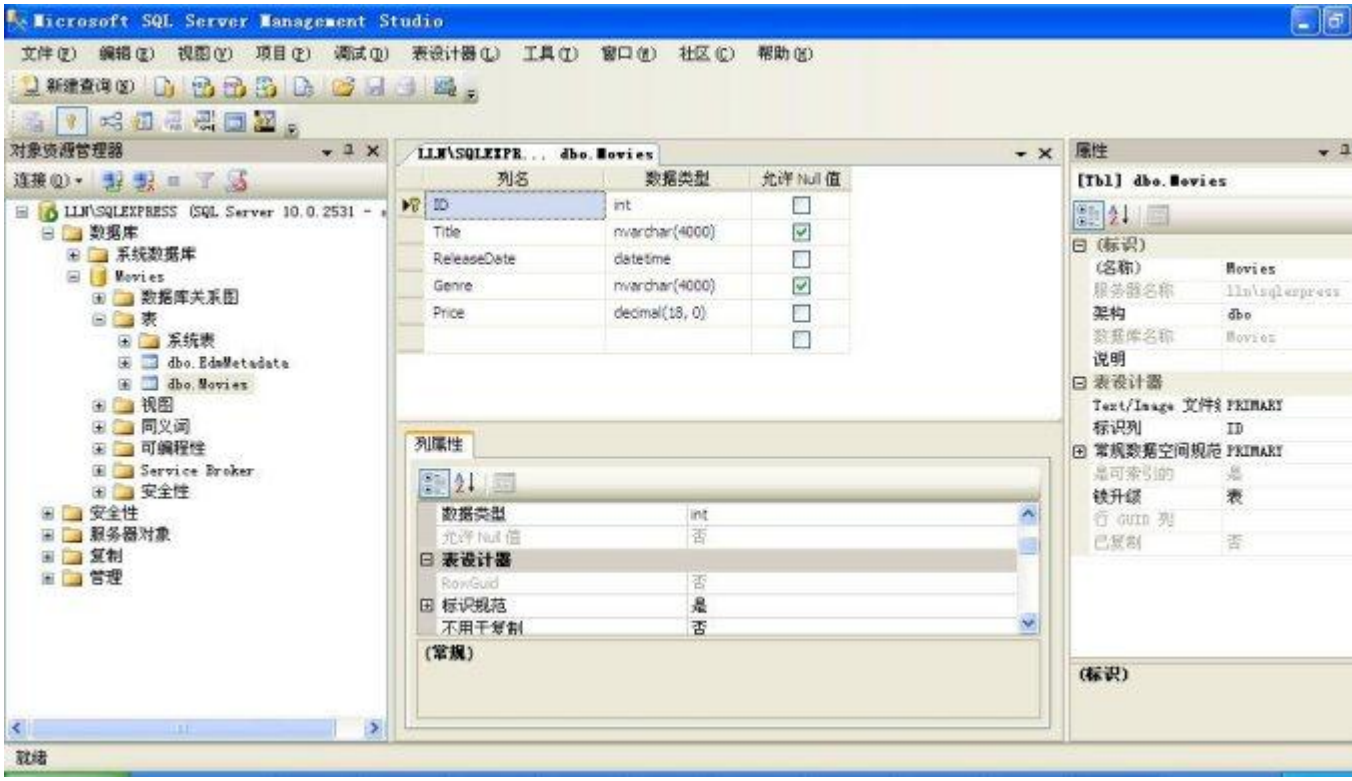


图 5-9 查看 Movies 表中各字段属性

这里请注意 Movies 表中各字段是如何映射到 Movie 类中各属性上的。Entity Framework code-fist 自动在你创建的 Movie 类的基础上创建了这张 Movies 数据表。

你现在已经可以访问数据库中的 Movies 数据表，并且有了一个简单的页面来显示这个表中的内容。在下一节，我们将增加一个追加数据的方法和一个追加数据的视图，并且向数据库中追加一些数据。

ASP.NET MVC3 快速入门-第六节 增加一个追加数据的方法和一个追加数据的视图

(2011-03-06 22:46:08)

标签： 分类： ASP.NETMVC3

视图

模板

表单

在本节中我们将要在数据库中追加并保存一些数据。我们将要创建一个表单以及一些表单输入控件，用来输入数据信息。当用户提交表单时将把这些用户输入的信息保存在数据库中。我们可以通过在浏览器中输入“http://localhost:xx/Movies/Create”这个 URL 地址来访问这个表单。

6.1 显示追加信息时所用表单

首先，我们需要在我们的 MoviesController 类中追加一个 Create 方法，该方法返回一个视图，该视图中包含了用户输入信息时所要用到的表单。

```
public ActionResult Create()
{
    return View();
}
```

现在让我们来实现这个 Create 方法中所要返回的视图，我们将在这个视图中向用户显示追加数据时所需要用到的表单。在 Create 方法中点击鼠标右键，并点击上下文菜单中的“添加视图”。

在“添加视图”对话框中选择“创建强类型视图”，将模型类指定为“Movie”，在支架模板中选择 Create，如图 6-1 所示。

图 6-1 添加追加数据时所用视图

点击添加按钮，Views 文件夹下的 Movies 文件夹中将会自动添加一个名为“*Create.cshtml*”的视图模板文件。因为你在支架模板中选择了“Create”,所以支架模板会在该视图模板文件中自动生成一些默认代码。打开该文件进行查看，在该文件中已经自动创建了一个 HTML 表单，以及一段用来显示校验时错误信息的文字。Visual Web Developer 检查 Movies 类，并自动创建与该类中每个属性相对应的<label>元素以及<input>元素。支架模板自动生成的创建数据所用视图中的代码如代码清单 6-1 所示。

代码清单 6-1 支架模板自动生成的创建数据所用视图中的代码

```
@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "Create";
}
```

```
<h2>Create</h2>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
type="text/javascript"></script>
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Movie</legend>
        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.ReleaseDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Genre)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Price)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>
        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Back to List", "Index")
```

```
</div>
```

这段代码中使用了几个 HTML 帮助器的方法来帮助简化 HTML 标签的书写方法。Html.LabelFor 帮助器用于显示字段名（”Title”,”ReleaseDate”,”Genre”或者”Price”）。Html.EditorFor 帮助器用于显示一个提供给用户输入信息的 HTML 的<input>元素。Html.ValidationMessageFor 帮助器用于显示一个针对属性的校验信息。请注意我们的视图模板的顶部有一个“@model MvcMovie.Models.Movie”的声明，该声明将我们的视图模板中的“模型”强类型化成一个 Movie 类。

运行应用程序，在浏览器中输入“http://localhost:xx/Movies/Create”,浏览器中显示如图 6-2 所示。

图 6-2 支架模板自动生成的视图

在书写中文网站或中文 Web 应用程序的时候，可以将代码清单 6-1 中的代码修改为代码清单 6-2 中所示代码。

代码清单 6-2 将支架模板自动生成的代码进行汉化

```
@model MvcMovie.Models.Movie
@{
   ViewBag.Title = "追加电影信息";
}
<h2>追加电影信息</h2>
```

```
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>电影</legend>
        <div class="editor-label">
            标题
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>
        <div class="editor-label">
            发行日期
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>
        <div class="editor-label">
          种类
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>
        <div class="editor-label">
            票价
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>
        <p>
            <input type="submit" value="追加" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("返回电影列表", "Index")
```

```
</div>
```

修改完毕后重新运行程序，然后访问“http://localhost:xx/Movies/Create”这个地址，浏览器中显示如图 6-3 所示。

图 6-3 中文化后的追加电影信息画面

鼠标右击浏览器中显示的该页面，点击“显示源代码”，显示出来的源代码如代码清单 6-3 所示。（为了突出本节内容，只显示与本节中相关部分）。

代码清单 6-3 追加电影信息画面在浏览器中的 HTML 代码

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>追加电影信息</title>
    <link href="/Content/Site.css" rel="stylesheet" type="text/css" />
    <script src="/Scripts/jquery- 1.4.4.min.js"
      type="text/javascript"></script>
</head>
<body>
<h2>追加电影信息</h2>
```

```
<script src="/Scripts/jquery.validate.min.js"
      type="text/javascript"></script>
<script src="/Scripts/jquery.validate.unobtrusive.min.js"
      type="text/javascript"></script>
<form action="/Movies/Create" method="post">
  <fieldset>
    <legend>电影</legend>
    <div class="editor-label">
      标题
    </div>
    <div class="editor-field">
      <input class="text-box single-line" id="Title" name="Title"
                type="text" value="" />
      <span class="field-validation-valid" data-valmsg-for="Title"
                data-valmsg-replace="true"></span>
    </div>
    <div class="editor-label">
      发行日期
    </div>
    <div class="editor-field">
      <input class="text-box single-line" data-val="true"
              data-val-required="The ReleaseDate field is required."
              id="ReleaseDate" name="ReleaseDate" type="text" value="" />
      <span class="field-validation-valid"
               data-valmsg-for="ReleaseDate"
               data-valmsg-replace="true">
             </span>
    </div>
    <div class="editor-label">
      种类
    </div>
    <div class="editor-field">
      <input class="text-box single-line" id="Genre" name="Genre"
              type="text" value="" />
      <span class="field-validation-valid" data-valmsg-for="Genre"
               data-valmsg-replace="true"/>
    </div>
    <div class="editor-label">
      票价
    </div>
    <div class="editor-field">
      <input class="text-box single-line" data-val="true"
              data-val-number="The field Price must be a number."
              data-val-required="The Price field is required." id="Price"
```

```
                    name="Price" type="text" value="" />
            <span class="field-validation-valid" data-valmsg-for="Price"
                    data-valmsg-replace="true"></span>
        </div>
        <p>
            <input type="submit" value="追加" />
        </p>
    </fieldset>
</form>
<div>
    <a href="/Movies">返回电影列表</a>
</div>
<div id="footer">
</div>
</body>
 </html>
```

从这段代码中可以看出，表单的 action 属性被设置为“/Movies/Create”，当点击追加按钮时会把表单中各文本框中的数据提交到服务器端进行保存。

6.2 处理 HTTP-POST

到此为止，我们已经实现了显示追加数据所用表单的所需代码。我们的下一步是编写代码来进行表单提交到服务器后的处理。我们将要获取用户在数据库中输入的信息并且将它们作为一个新的 Movie 保存到数据库中。

为了实现这一处理，我们需要在 MoviesController 类中追加第二个 Create 方法。这个 Create 方法具有一个[HttpPost]属性，它意味着我们将要用它来处理提交到“/Movies/Create”这个 URL 地址的请求。另外，所有提交到“/Movies/Create”这个 URL 地址的非 POST 的请求（即 GET 请求）将被第一个 Create 方法进行处理，即简单地返回一个空的表单。

代码清单 6-4 中所示代码为 MoviesController 类中的两个 Create 方法的全部代码。

代码清单 6-4 MoviesController 类中的两个 Create 方法的全部代码

```
public ActionResult Create()
{
    return View();
}
[HttpPost]
public ActionResult Create(Movie newMovie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(newMovie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    else
        return View(newMovie);
```

```
}
```

之前我们介绍了 ASP.NET MVC 可以自动地将一个 URL 地址中的查询字符串中的参数（例如：传递到“/HelloWorld/Welcome?name=Scott&numTimes=5”）作为一个方法的参数传递到方法中。同样地，除了传递查询字符串中的参数之外，ASP.NET MVC 也可以用这种方法来传递提交后的表单参数。

提交后的表单参数可以作为一个独立的参数传递到一个方法中。例如，ASP.NET MVC framework 可以将我们提交的表单中的控件值作为参数传递到具有 HttpPost 属性的 Create 方法中，如下所示。

```
[HttpPost]
 public ActionResult Create(string title, DateTime releaseDate, string genre,
decimal price)
{
```

提交的表单值也可以被映射到一个复合的，具有属性的对象（譬如我们的 Movie 类），并且作为一个单一的参数传递到一个方法中。在代码清单 6-4 中我们使用的就是这个方法。请注意 Create 方法是怎样作为一个参数来接收 Movie 对象的。

```
[HttpPost]
public ActionResult Create(Movie newMovie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(newMovie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    else
        return View(newMovie);
}
```

ModelState.IsValid 属性用来检查提交的表单中的数据是否能够被用来创建一个 Movie 对象。如果数据是有效的，我们的代码将把提交上来的这个 Movie 类追加在 MoviesDBContext 对象的实例中的 Movies 集合中。调用我们的 MoviesDBContext 对象实例的 SaveChanges 方法将把这个 Movie 对象保存在数据库中。保存数据完毕后，代码把画面重定向到 MoviesController 类的 Index 方法中，浏览器中将会打开电影清单显示画面，并且在电影清单中显示刚才追加的这条数据。

如果提交的值是无效的，将会返回到电影信息追加画面中，并且在表单的各输入控件中显示各自的提交的值。各输入控件的 Html.ValidationMessageFor 帮助器将会显示对应的出错信息。

6.3 追加一条电影信息

运行应用程序，在浏览器中输入“http://localhost:xx/Movies/Create”，在表单中输入一条电影信息，然后点击追加按钮，如图 6-4 所示。

图 6-4 追加电影信息

点击追加按钮进行提交，表单中输入的这条电影信息将会保存到数据库中，保存后浏览器中将打开电影清单画面，并且将这条追加的电影显示在清单中，如图 6-5 所示。

图 6-5 追加后电影将显示在电影清单中

你可能已经注意到了在这个电影清单画面中将刚才追加的电影票价显示成了 10 元，而不是用户输入的 9.99 元，这是因为当前该数据表中 Decimal 类型的默认精度只能识别与处理整数值，并且自动将小数部分四舍五入。关于如何解决这个问题，我们将在下一节中对模型进行一些调整的时候会同时进行介绍。

现在我们已经有了一个 Web 应用程序的雏形，我们可以在数据库中追加数据，显示数据。代码清单 6-5 是现在这个 MoviesController 类的完整代码。

代码清单 6-5 MoviesController 类的完整代码

```
using MvcMovie.Models;
using System.Linq;
using System;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class MoviesController : Controller
    {
```

```
        MovieDBContext db = new MovieDBContext();
        public ActionResult Index()
        {
            var movies=from m in db.Movies
                       where m.ReleaseDate>new DateTime(1984,6,1)
                       select m;
            return View(movies.ToList());
        }
        public ActionResult Create()
        {
            return View();
        }
        [HttpPost]
        public ActionResult Create(Movie newMovie)
        {
            if (ModelState.IsValid)
            {
                db.Movies.Add(newMovie);
                db.SaveChanges();
                return RedirectToAction("Index");
            }
            else
                return View(newMovie);
        }
    }
}
```

在下一节，我们将介绍如何为我们的模型添加附加的属性，如何在映射后的数据库中定制我们的票价列的精度。

ASP.NET MVC3 快速入门-第七节 在 Movie(电影)模型与数据表中添加一个字段

(2011-03-07 20:04:21)

标签：　　分类：　ASP.NETMVC3

应用程序

模型

杂谈

在本节中我们将要对我们的模型类进行修改，同时介绍如何在 ASP.NET MVC3 中根据这些修改来调整我们数据表的结构。

7.1 在我们的 Movie 模型中添加一个 Rating(电影等级)属性

首先，我们在现存的 Movie 类中添加一个附加的“Rating”属性。打开 Movie.cs 文件，在 Movie 类中添加一个 Rating 属性，如下所示。

```
public string Rating { get; set; }
```

现在完整的 Movie 类的代码如代码清单 7-1 所示。

代码清单 7-1 完整的 Movie 类的代码

```
public class Movie
{
    public int      ID          { get; set; }
    public string   Title       { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string   Genre       { get; set; }
    public decimal  Price       { get; set; }
    public string   Rating      { get; set; }
}
```

点击“调试”菜单下的“生成 MvcMovie”,重新编译应用程序。

现在我们已经将我们的模型进行了更新，让我们同样地修改我们的 Views 文件夹下的 Movies 文件夹下的 Index.cshtml 文件与 Create.cshtml 这两个视图模板文件，在视图中添加这个 Rating 属性。

首先打开 Index.cshtml 文件，在内容为“票价”（对应 Price 属性）的<th>元素后面追加“<th>电影等级</th>”列标题（对应 Rating 属性）。在显示 Price 属性内容的 td 元素后面追加一个<td>元素，用来显示 Rating 属性的内容。进行了这两个修改后的 Index.cshtml 文件中的主要内容如下所示。

```
<table>
    <tr>
        <th></th>
        <th>
            电影名称
        </th>
        <th>
            发行日期
        </th>
        <th>
            种类
        </th>
        <th>
            票价
        </th>
        <th>
            电影等级
        </th>
    </tr>
@foreach (var item in Model) {
```

```
    <tr>
        <td>
            @Html.ActionLink("编辑", "Edit", new { id=item.ID }) |
            @Html.ActionLink("查看明细", "Details", new { id=item.ID }) |
            @Html.ActionLink("删除", "Delete", new { id=item.ID })
        </td>
        <td>
            @item.Title
        </td>
        <td>
            @String.Format("{0:d}", item.ReleaseDate)
        </td>
        <td>
            @item.Genre
        </td>
        <td>
            @String.Format("{0:c2}", item.Price)
        </td>
        <td>
            @item.Rating
        </td>
    </tr>
}
</table>
```

接下来打开 Create.cshtml 文件，在表单底部追加如下所示的标签。它将显示为一个文本框，用来输入 Rating 属性的内容。

```
<div class="editor-label">
    电影等级
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Rating)
    @Html.ValidationMessageFor(model => model.Rating)
</div>
```

7.2 维护模型与数据库结构之间的差别

现在我们已经将应用程序修改完毕，在 Movie 模型中添加了一个 Rating 属性。

现在让我们重新运行应用程序，打开“http://localhost:xx/Movies”这个 URL 地址，这时，浏览器会显示一个应用程序出错画面。如图 7-1 所示。

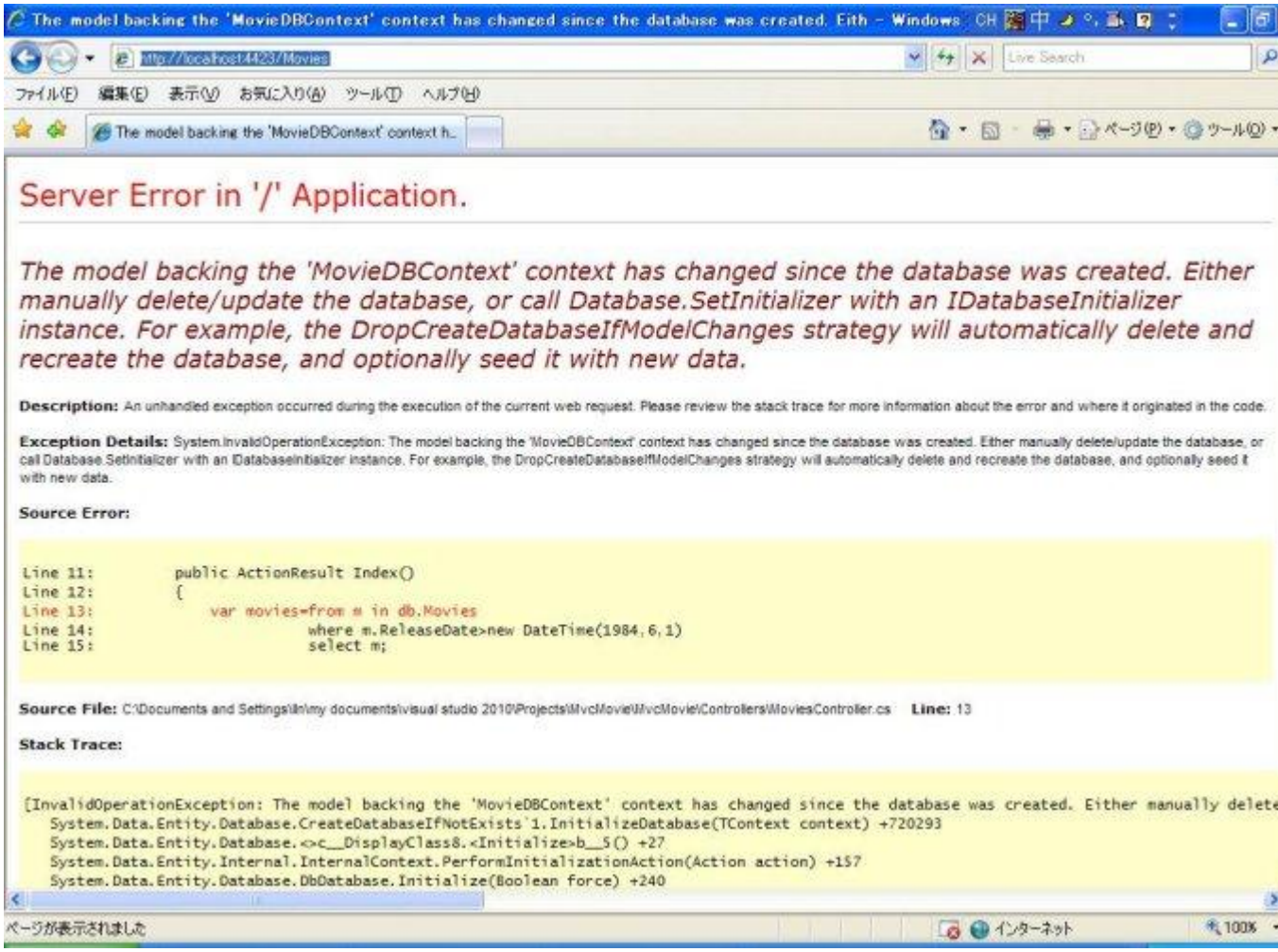


图 7-1 应用程序出错画面

导致这个问题发生的原因是因为在我们的应用程序中，更新后的 Movie 模型类与我们实际连接的数据库的结构并不统一（Movies 数据表中并没有 Rating 列）。

在默认情况下，当你使用 EF code-first 自动创建数据库时（就好像本教程中前面所介绍过的那样），EF code-first 会自动在数据库中追加数据表来使得数据库的结构与它自动生成的模型类保持同步。如果不同步，EF 将会抛出一个错误。这使得在开发程序时对于错误的跟踪会变得更加容易，否则你只能在运行时发现这个错误。同步检查特性正是引起以上显示的错误的原因。

有两种方法可以解决这个错误：

1. 让 Entity Framework 自动删除当前数据库，并在新的模型类的基础上重新创建该数据库。这种方法在使用一个测试数据库时对于开发来说是十分方便的，因为它允许你快速地同步修改模型与数据库。但缺点是你将丢失现存库中的数据（所以请不要将这个方法使用在实际使用中的数据库上）。
2. 修改数据库中的数据表的结构来使之与模型相匹配。这个方法的好处是可以让你保留表中的数据。你可以手工实现这一操作，或创建一个数据表修改脚本来实现这一操作。

在本教程中，我们使用第一种方法，在任何模型发生了改变的情况下让 EF code-first 自动重建数据库。

7.3 当模型改变时自动重建数据库

现在我们来修改我们的应用程序，使得我们的应用程序中如果任何模型发生了改变，都将自动删除与重建当前模型所使用的数据库。（请使用一个开发测试用的数据库来实践这一操作。用在实际使用的数据库中将造成库中数据的丢失）。

在解决方案资源管理器中，鼠标右击 Modes 文件夹，选择“添加”，然后点击“类”，如图 7-2 所示。

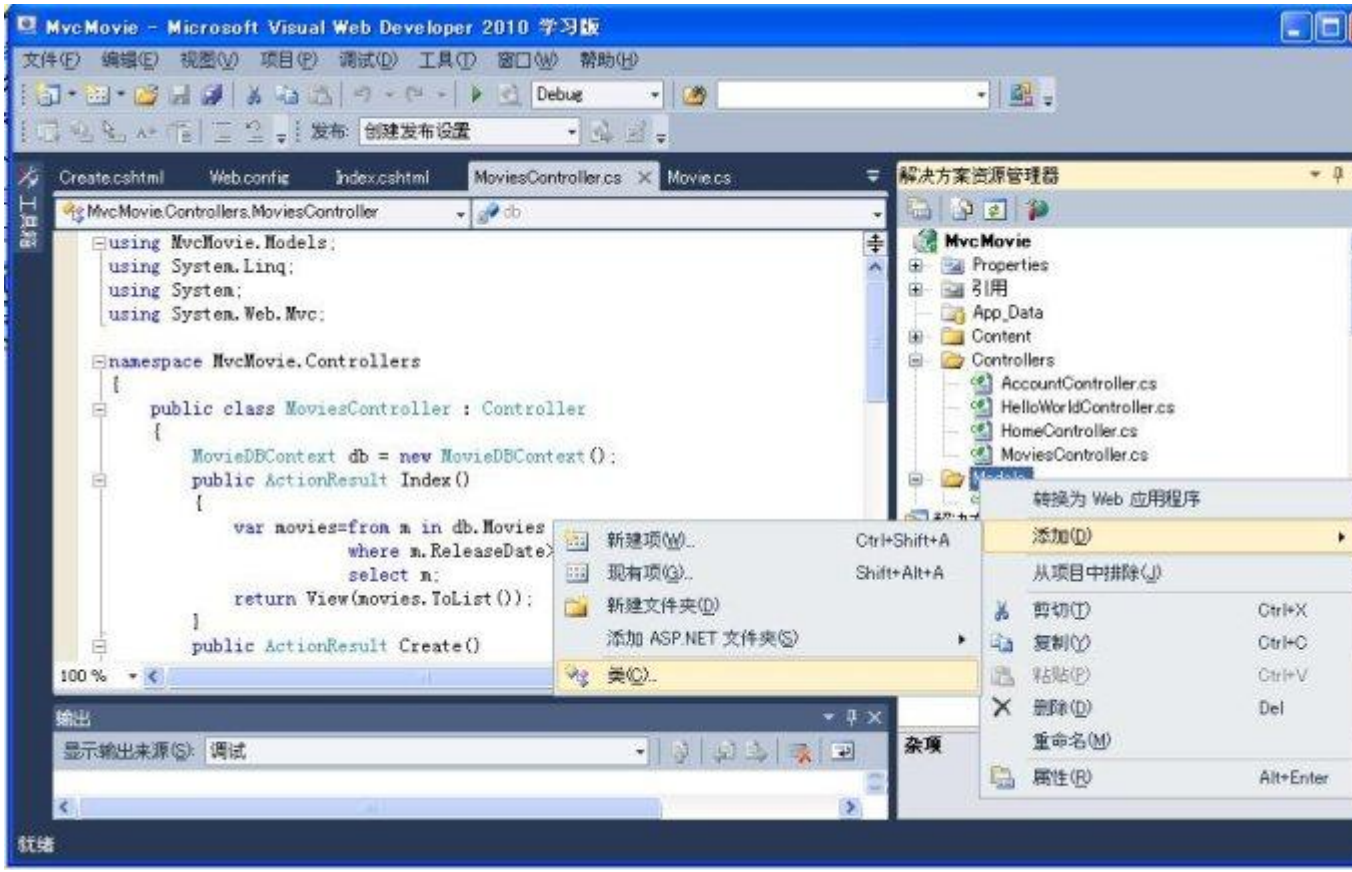


图 7-2 添加一个新类

在“添加新项”对话框中，将类名定义为“MovieIntializer”，然后点击添加按钮添加该类。书写该类的代码如代码清单 7-2 所示。

代码清单 7-2 MovieIntializer 类的完整代码

```
using System;
using System.Collections.Generic;
using System.Data.Entity.Database;
namespace MvcMovie.Models
{
   public class MovieIntializer :
DropCreateDatabaseIfModelChanges<MovieDBContext>
   {
      protected override void Seed(MovieDBContext context)
      {
         var movies = new List<Movie> {
```

```
            new Movie { Title = "非诚勿扰 2",
                    ReleaseDate=DateTime.Parse("2011-1-11"),
                    Genre="爱情",
                    Rating="R",
                    Price=7.00M},

            new Movie { Title = "赵氏孤儿",
                    ReleaseDate=DateTime.Parse("2011-2-23"),
                    Genre="历史",
                    Rating="R",
                    Price=9.00M},
          };

          movies.ForEach(d => context.Movies.Add(d));
        }

    }
}
```

使用这个 MovieInitializer 类之后，一旦我们的模型类发生改变后，我们的模型类所映射的数据库都会被自动重建。代码中使用了 Seed 方法来指定任何重建数据库的时候想要默认追加到某张数据表中的数据。这为将一些示例数据添加到数据表中的操作提供了一个有用的方法，而不需要重建了数据库之后再手工到数据表中添加示例数据。

现在我们已经定义好了我们的 MovieInitializer 类，接下来我们想在整个工程中使用这个类，这样每次在运行我们的应用程序的时候会自动检查当前我们的模型类结构是否与数据库结构不一致，如果不一致的时候就自动重建该数据库，并且追加 MovieInitializer 类中所指定的默认数据。

打开我们的 MvcMovies 工程的根目录下的 Global.asax 文件，如图 7-3 所示。

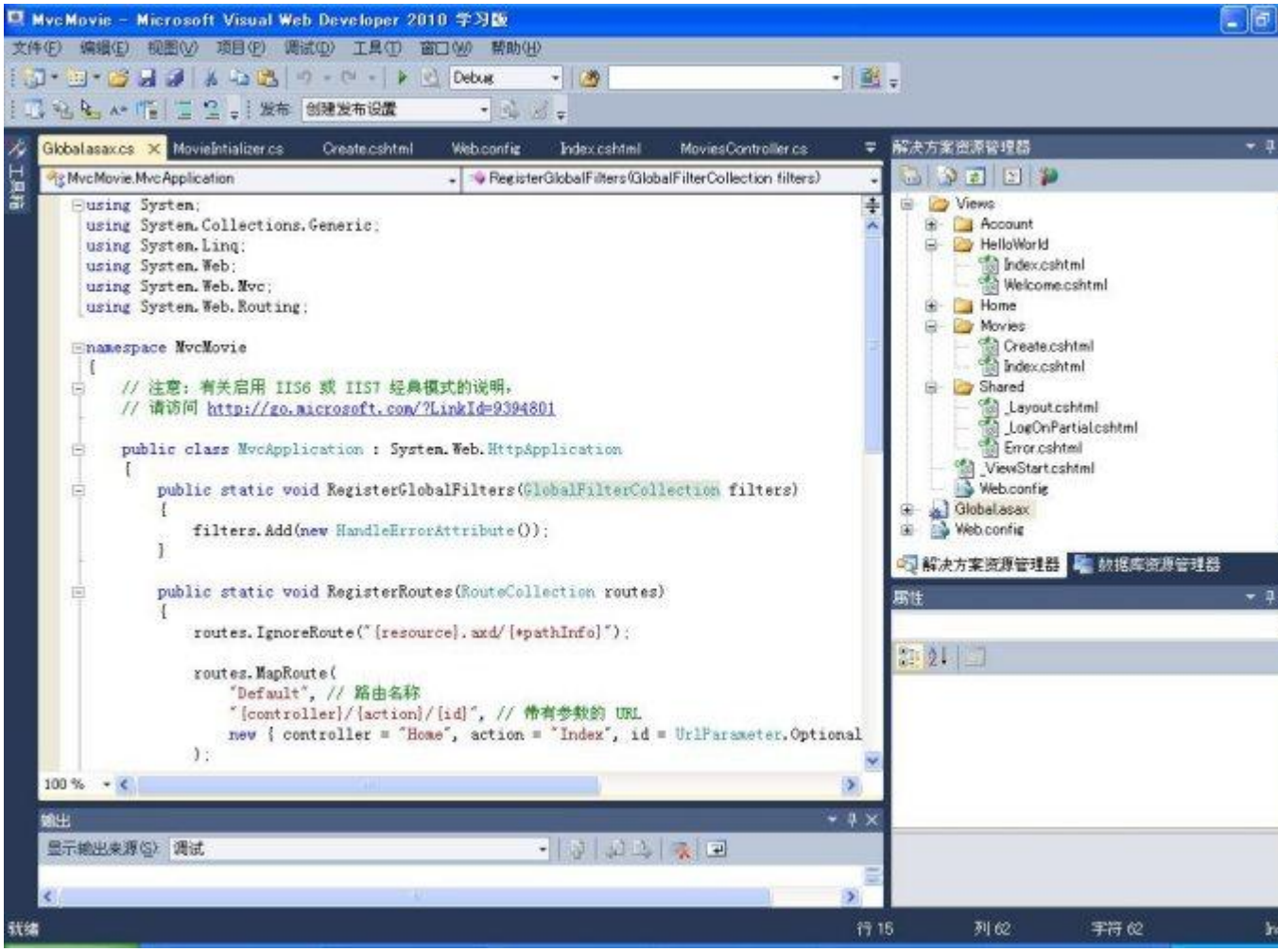


图 7-3 Global.asax 文件

Global.asax 文件中定义了当前工程所使用到的 Application(应用程序)主类，包含了一个 Application_Start()事件处理器，当第一次运行我们的应用程序时会触发这个事件。

让我们在文件头部追加两个有用的声明。第一个声明引用 Entity Framework 命名空间，第二个声明引用我们的 MovieInitializer 类所存在的命名空间。这两句声明的代码如下所示。

```
using System.Data.Entity.Database;  // DbDatabase.SetInitialize
using MvcMovie.Models;              // MovieInitializer
```

接下来寻找到 Application_Start 方法，在该方法的开头追加一个 DbDatabase.SetInitializer() 方法，代码如下所示。

```
protected void Application_Start()
{
    DbDatabase.SetInitializer<MovieDBContext>(new MovieInitializer());
    AreaRegistration.RegisterAllAreas();
    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

我们追加的 DbDatabase.SetInitializer 方法将会在实际使用的数据库中的结构与我们的 Movie 模型类所映射的数据库结构不匹配时自动重建该数据库，并且自动添加 MovieInitializer 类中所指定的默认数据。

关闭 Global.asax 文件。

重新运行我们的应用程序，并在浏览器中输入“http://localhost:xx/Movies”。当我们的应用程序启动的时候，会自动觉察出我们的模型类结构与数据库的结构不再匹配，于是重建该数据库使之相匹配，并且自动追加默认数据，浏览器中显示结果如图 7-4 所示。

图 7-4 修改了模型类结构并追加默认数据后浏览器中的显示结果

点击追加按钮进行数据的追加，如图 7-5 所示。

图 7-5 追加数据

点击追加按钮后，新追加的包括 Rating(电影等级)字段的数据能正常追加并在电影清单画面中正常显示，如图 7-6 所示。

图 7-6 包括 Rating(电影等级)字段的数据能被正常追加与显示

7.4 修正票价字段的精度

在第六节追加数据的时候我们遗留下来一个问题，就是我们在追加数据的时候，票价（Price）字段中输入的是 9.99 元，但是电影清单显示画面中该数据的票价字段显示为 10 元，这是为什么？

这个问题发生的原因是因为当 EF code-first 在创建数据表的时候，如果字段为 Decimal 类型，则使用默认的精度(18:0)，从而使得 9.99 元被四舍五入成为 10 元。现在我们想要将这个默认的精度修改为(18:2)，从而使得数据表中的票价字段能够存储小数点后的两位数字。可喜的是 EF code-first 允许你很容易地重载这个定义模型如何向数据库中存取数据的映射规则。你可以利用这个重载机制来重载 EF code-first 中默认的类型定义以及数据表的继承规则，然后在这个基础上进行数据的存取。

为了改变我们的票价（Price）字段在数据表中的精度，打开 Models 文件夹下的 Movie.cs 文件。追加一句引用 System.Data.Entity.ModelConfiguration 的语句，代码如下所示。

```
using System.Data.Entity.ModelConfiguration;
```

在我们现在的 MovieDBContext 类中重载 OnModelCreating 方法，代码如下所示。

```
public class MovieDBContext : DbContext
{
   public DbSet<Movie> Movies { get; set; }

   protected override void OnModelCreating(ModelBuilder modelBuilder)
   {
   modelBuilder.Entity<Movie>().Property(p => p.Price).HasPrecision(18, 2);
   }
}
```

OnModelCreating 方法可以被用来重载与定制规定我们的模型类如何与我们的数据表进行映射的映射规则。代码中使用了 EF 的 ModelBuilder API 来定义我们的 Movie 对象的票价(Price)字段的精度为准确到小数点后两位。

现在完整的 Movie.cs 中的代码如代码清单 7-3 所示。

代码清单 7-3 完整的 Movie.cs 中的代码

```
using System;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration;
namespace MvcMovie.Models
{
   public class Movie
   {
      public int ID { get; set; }
      public string Title { get; set; }
      public DateTime ReleaseDate { get; set; }
      public string Genre { get; set; }
      public decimal Price { get; set; }
      public string Rating { get; set; }
   }
   public class MovieDBContext : DbContext
   {
      public DbSet<Movie> Movies { get; set; }
      protected override void OnModelCreating(ModelBuilder modelBuilder)
      {
         modelBuilder.Entity<Movie>().Property(p =>
                  p.Price).HasPrecision(18, 2);
      }
    }
}
```

现在让我们重新运行我们的应用程序，并且在浏览器中输入“http://localhost:xx/Movies”。当应用程序启动的时候，EF code-first 将会再次察觉我们的模型类中的结构与数据表的结构不再匹配，然后自动重建数据表与新的模型类的结构进行匹配（具有新的票价（Price）字段的精度）。

重新创建一个新的电影（Movie）数据，在票价（Price）字段中输入 9.99。请注意数据表中保存后显示在电影清单画面中的该条数据的票价（Price）字段中现在也显示为 9.99 了，如图 7-7 中所示。

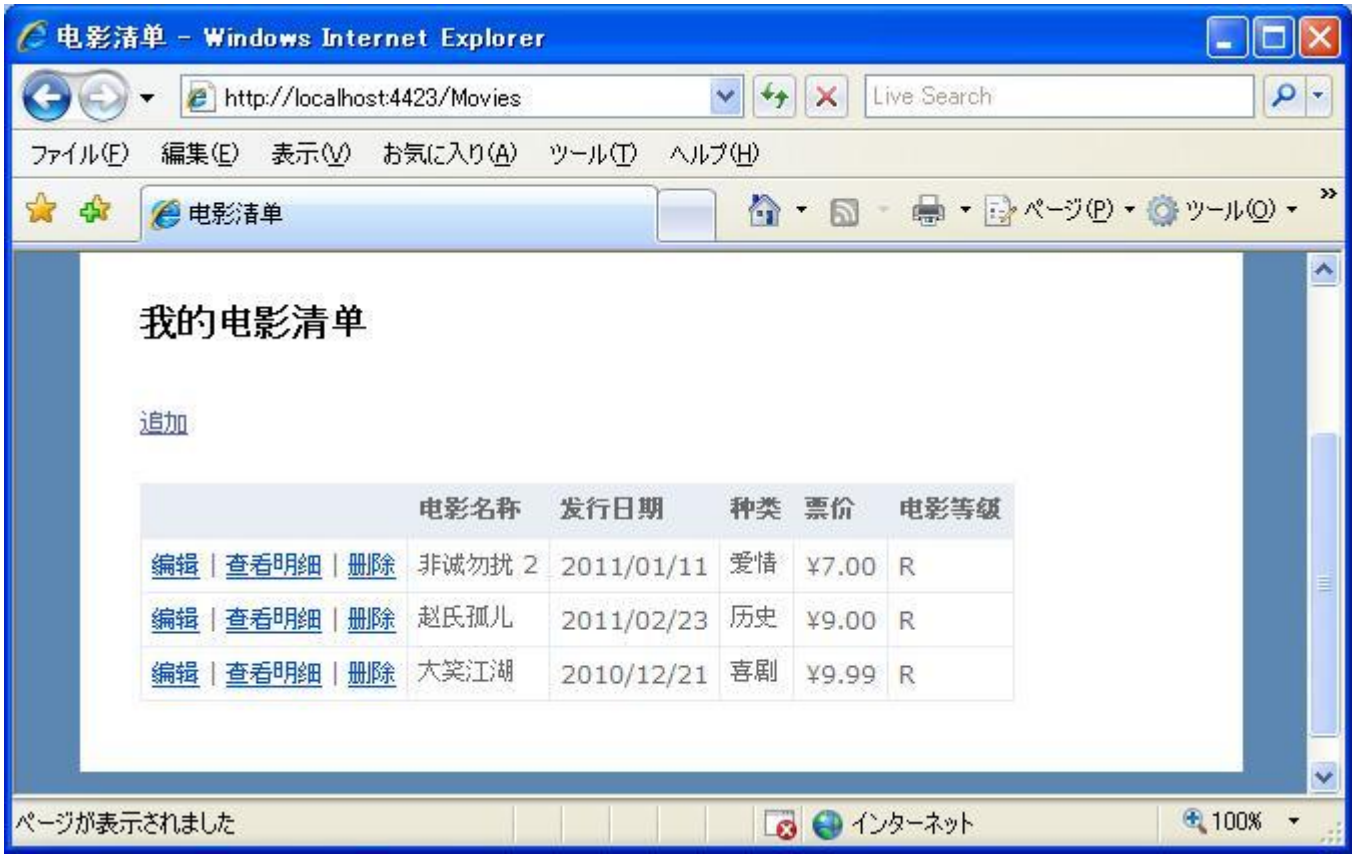


图 7-7 修改精度后电影清单画面中的显示结果

在本节中我们介绍了如何快速调整你的模型对象，同时在改变模型类的结构时自动将数据表中的结构保持同步。我们也介绍了如何预先在自动重建的数据表中追加默认数据，它使得你可以快速在数据表中追加一些测试数据。在下一节中，我们将介绍如何在我们的模型类中加入我们自定义的数据有效性校验规则，从而强制实现一些业务规则。

ASP.NET MVC3 快速入门-第八节 在模型中添加验证规则

(2011-03-09 20:53:37)

标签： 分类： ASP.NETMVC3

校验

本节介绍如何在我们的 Movie(电影)模型中添加一些验证规则，同时确认当用户使用我们的应用程序创建或编辑电影信息时将使用这些验证规则对用户输入的信息进行检查。

8.1 DRY 原则

在 ASP.NET MVC 中，有一条作为核心的原则，就是 DRY(“Don’t Repeat Yourself，中文意思为：不要让开发者重复做同样的事情)原则。ASP.NET MVC 提倡让开发者“一处定义、处处可用”。这样可以减少开发者的代码编写量，同时也更加便于代码的维护。

ASP.NET MVC 与 EF code-first 提供的默认验证规则就是一个实现 DRY 原则的很好的例子。你也可以在模型类中显式地追加一个验证规则，然后在整个应用程序中都使用这个验证规则。

现在让我们来看一下怎样在我们的应用程序中追加一些验证规则。

8.2 在 Movie 模型中追加验证规则

首先，让我们在 Movie 类中追加一些验证规则。

打开 Movie.cs 文件，在文件的头部追加一条引用 System.ComponentModel.DataAnnotations 命名空间的 using 语句，代码如下所示。

```
using System.ComponentModel.DataAnnotations;
```

这个 System.ComponentModel.DataAnnotations 命名空间是.NET Framework 中的一个命名空间。它提供了很多内建的验证规则，你可以对任何类或属性显式指定这些验证规则。

现在让我们来修改 Movie 类，增加一些内建的 Required(必须输入)，StringLength(输入字符长度)与 Range(输入范围)验证规则，代码如代码清单 8-1 所示。

代码清单 8-1 在 Movie 类中追加内建的验证规则

```
public class Movie
{
    public int ID { get; set; }
    [Required(ErrorMessage = "必须输入标题")]
    public string Title { get; set; }
    [Required(ErrorMessage = "必须输入发行日期")]
    public DateTime ReleaseDate { get; set; }
    [Required(ErrorMessage = "必须指定种类")]
    public string Genre { get; set; }
    [Required(ErrorMessage = "必须输入票价")]
    [Range(1, 100, ErrorMessage = "票价必须在 1 元到 100 元之间")]
    public decimal Price { get; set; }
    [StringLength(5,ErrorMessage = "最多允许输入五个字符")]
    public string Rating { get; set; }
}
```

上述这些验证属性指定了我们想要强加给模型中各属性的验证规则。Required 属性表示必须要指定一个属性值，在上例中，一个有效的电影信息必须含有标题，发行日期，种类与票价信息。Range 属性表示属性值必须在一段范围之间。StringLength 属性表示一个字符串属性的最大长度或最短长度。

EF code-first 在将一条数据保存到数据库中之前首先使用你对模型类指定的验证规则来对这条数据进行有效性检查。例如，在以下代码中，当程序调用 SaveChanges 方法时将抛出一个异常（也称例外），因为数据并不满足 Movie 属性的必须输入条件，同时票价属性的值为 0，不在指定的允许范围内（1-100）。

```
MovieDBContext db = new MovieDBContext();
Movie movie = new Movie();
movie.Title = "大笑江湖;
movie.Price = 0.0M;
db.Movies.Add(movie);
```

```
db.SaveChanges();      // 这里将抛出一个校验异常
```

通过 Entity Framework 来自动实现验证规则检查可以让我们的应用程序变得更强健。它也确保我们不会由于忘了实施数据验证而使得一些无效数据保存到数据库中。

代码清单 8-2 为现在 Movie.cs 文件中的完整代码。

代码清单 8-2 Movie.cs 文件中的完整代码

```
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration;
using System.ComponentModel.DataAnnotations;
namespace MvcMovie.Models
{
   public class Movie
   {
      public int ID { get; set; }
      [Required(ErrorMessage = "必须输入标题")]
      public string Title { get; set; }
      [Required(ErrorMessage = "必须输入发行日期")]
      public DateTime ReleaseDate { get; set; }
      [Required(ErrorMessage = "必须指定种类")]
      public string Genre { get; set; }
      [Required(ErrorMessage = "必须输入票价")]
      [Range(1, 100, ErrorMessage = "票价必须在 1 元到 100 元之间")]
      public decimal Price { get; set; }
      [StringLength(5,ErrorMessage = "最多允许输入五个字符")]
      public string Rating { get; set; }
   }
   public class MovieDBContext : DbContext
   {
      public DbSet<Movie> Movies { get; set; }
      protected override void OnModelCreating(ModelBuilder modelBuilder)
      {
        modelBuilder.Entity<Movie>().Property(p =>
                p.Price).HasPrecision(18, 2);
      }
    }
}
```

8.3 ASP.NET MVC 中的验证错误 UI(用户界面)

现在让我们运行我们的应用程序，并在地址栏中输入“http://localhost:xx/Movies”。

在电影清单画面中点击追加按钮打开追加电影画面。在该画面中的表单中填入一些无效的属性值，然后点击追加按钮。如图 8-1 所示。

图 8-1 ASP.NET MVC 中的验证错误 UI

请注意表单自动使用了一个背景颜色来高亮显示包含了无效数据的文本框，并且在每个文本框的旁边显示验证错误信息。使用的错误信息文字正是我们在前面代码中所指定的验证错误的错误信息文字。这个验证错误既可以由客户端引发（使用 JavaScript 脚本）,也可以由服务器端引发（当用户禁止使用 JavaScript 脚本时）。

这种处理方法是非常不错的，因为我们不再需要为了显示错误信息文字而在 MoviesController 类或 Create.cshtml 视图文件中书写不必要的代码。我们之前创建的控制器与视图将自动实施验证规则与显示验证错误信息文字。

8.4 在 Create 视图（追加电影视图）与 Create 方法内部是如何实现验证的

也许有的读者会问，既然我们没有追加任何显示错误信息提示的代码，那么我们的控制器或视图内部是如何生成这个显示错误信息提示的画面的。首先我们将 MovieController 类中的代码显示如下，在我们在 Movie 类中追加了验证规则后，我们并没有修改这个类中的任何代码。

```
//
// GET: /Movies/Create

public ActionResult Create()
{
    return View();
```

```
}

//
// POST: /Movies/Create

[HttpPost]
public ActionResult Create(Movie newMovie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(newMovie);
        db.SaveChanges();

        return RedirectToAction("Index");
    }
    else
    {
        return View(newMovie);
    }
}
```

第一个方法返回追加电影视图。在第二个方法中对追加电影视图中的表单的提交进行处理。该方法中的 ModelState.IsValid 属性用来判断是否提交的电影数据中包含有任何没有通过数据验证的无效数据。如果存在无效数据，Create 方法重新返回追加电影视图。如果数据全部有效，则将该条数据保存到数据库中。

我们之前创建的使用支架模板的 Create.cshtml 视图模板中的代码显示如下，在首次打开追加电影视图与数据没有通过验证时，Create 方法中返回的视图都是使用的这个视图模板。

```
@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "追加电影信息";
}
<h2>追加电影信息</h2>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>电影</legend>
        <div class="editor-label">
            标题
        </div>
```

```
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>
        <div class="editor-label">
            发行日期
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>
        <div class="editor-label">
          种类
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>
        <div class="editor-label">
            票价
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>
        <p>
            <input type="submit" value="追加" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("返回电影列表", "Index")
</div>
```

请注意在这段代码中使用了许多 Html.EditorFor 帮助器来为 Movie 类的每个属性输出一个输入文本框。在每个 Html.EditorFor 帮助器之后紧跟着一个 Html.ValidationMessageFor 帮助器。这两个帮助器将与从控制器传入的模型类的对象实例（在本示例中为 Movie 对象的一个实例）结合起来，自动寻找指定给模型的各个验证属性，然后显示对应的验证错误信息。

这种验证体制的好处是在于控制器和 Create 视图（追加电影视图）事先都即不知道实际指定的验证规则，也不知道将会显示什么验证错误信息。验证规则和错误信息只在 Movie 类中被指定。

如果我们之后想要改变验证规则，我们也只要在一处地方进行改变就可以了。我们不用担心整个应用程序中存在验证规则不统一的问题，所有的验证规则都可以集中在一处地方进行指定，然后在整个应用程序中使用这些验证规则。这将使我们的代码更加清晰明确，更加具

有可读性、可维护性与可移植性。这将意味着我们的代码是真正符合 DRY 原则（一处指定，到处可用）的。

在下一节中，作为结尾部分，我们将介绍如何修改与删除数据，同时介绍如何显示一条数据的细节信息。

ASP.NET MVC3 快速入门-第九节 实现编辑、删除与明细信息视图

(2011-03-12 19:42:32)

标签： 分类： ASP.NETMVC3

明细

视图

杂谈

9.1 实现数据的明细信息视图

首先，让我们来看一下如何实现一条数据的明细信息视图。为了更好地体会这一功能，首先我们在前文所述的电影清单视图（Views 文件夹下面的 Movies 文件夹下面的 Index.cshtml 文件）中删除电影清单中的种类、票价、电影等级字段，使其代码如代码清单 9-1 中所示。

代码清单 9-1 修改后的 Index.cshtml 文件

```
@model IEnumerable<MvcMovie.Models.Movie>
@{
    ViewBag.Title = "电影清单";
}
<h2>我的电影清单</h2>
<p>
    @Html.ActionLink("追加", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>
            电影名称
        </th>
        <th>
            发行日期
        </th>
    </tr>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("编辑", "Edit", new { id=item.ID }) |
            @Html.ActionLink("查看明细", "Details", new { id=item.ID }) |
            @Html.ActionLink("删除", "Delete", new { id=item.ID })
        </td>
```

```
        <td>
            @item.Title
        </td>
        <td>
            @String.Format("{0:d}", item.ReleaseDate)
        </td>
    </tr>
}
</table>
```

重新运行该应用程序，在浏览器中输入地址“http://localhost:xx/Movies”，浏览器中运行结果如图 9-1 所示。

图 9-1　修改后的电影清单画面

现在电影清单画面中就只显示每条数据的电影名称与发行日期了，如果像查看该条数据的详细信息，需要点击每条数据的“查看明细”链接，将画面导航到明细数据画面，在该画面中查看这条数据的明细信息。当一条数据的细节信息比较多，而我们只想在该数据的

列举清单中显示该数据的几个摘要信息，通过点击链接或按钮的操作来查看数据的细节信息时这种处理方法是比较有用的。

接下来让我们来追加这个明细数据视图。首先打开 Movie 控制器，追加一个返回明细数据视图的 Details 方法，代码如下所示。

```
//
// GET: /Movies/Details
public ActionResult Details(int id)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
        return RedirectToAction("Index");
    return View("Details", movie);
}
```

code-first 通过使用 Find 方法来让一条数据的寻找变得非常容易。这个方法的一个非常重要的安全特性就是我们可以确保我们寻找的是一条可以被映射为 Movie 对象的数据。为什么这种做法可以确保安全性呢？举个例子来说，一个黑客可以将“http://localhost:xxxx/Movies/Details/1”地址修改为“http://localhost:xxxx/Movies/Details/12345”，如果数据库中没有这条 id 为 12345 的数据，根据以上代码所示，作为寻找结果的 Movie 对象将被设定为 null,浏览器将重新返回显示电影清单画面。

在 Details 方法中点击鼠标右键，选择“添加视图”，依然勾选“创建强类型视图”，模型类选择 Movie,在支架模板中选择“Details”（明细数据），如图 9-2 所示。

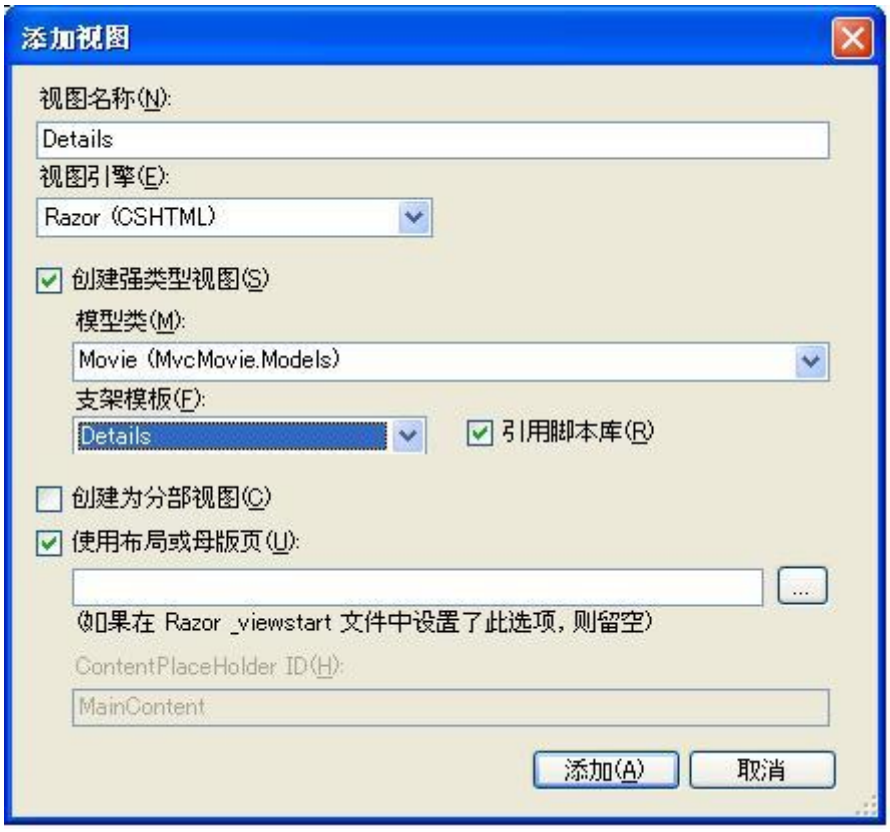


图 9-2 添加明细数据视图

如果要创建中文网站或应用程序，则将默认生成的 Details.cshtml 文件中有关英文文字修改为中文，修改完毕后该文件中的代码如代码清单 9-2 中所示。

代码清单 9-2 Details.cshtml 文件（明细数据视图）中的代码

```
@model MvcMovie.Models.Movie
@{
   ViewBag.Title = "电影详细信息";
}
<h2>电影详细信息</h2>
<fieldset>
   <legend>电影</legend>
   <div class="display-label">标题</div>
   <div class="display-field">@Model.Title</div>
   <div class="display-label">发行日期</div>
   <div class="display-field">@String.Format("{0:d}",
      Model.ReleaseDate)</div>
   <div class="display-label">种类</div>
   <div class="display-field">@Model.Genre</div>
   <div class="display-label">票价</div>
   <div class="display-field">@String.Format("{0:c2}", Model.Price)</div>
   <div class="display-label">等级</div>
   <div class="display-field">@Model.Rating</div>
</fieldset>
<p>
   @Html.ActionLink("编辑", "Edit", new { id=Model.ID }) |
   @Html.ActionLink("返回电影清单", "Index")
</p>
```

重新运行应用程序，在电影清单画面中点击某个电影的“查看明细”链接，浏览器显示画面如图 9-3 所示。

图 9-3 电影细节信息画面

9.2 实现数据的修改视图

接下来，让我们来看一下如何实现一个用来修改数据的视图。

首先打开 Movie 控制器，追加一个返回数据修改视图的 Edit 方法与一个对该视图中的表单提交进行处理的 Edit 方法，代码如下所示。

```
//
// GET: /Movies/Edit
public ActionResult Edit(int id)
```

```
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
        return RedirectToAction("Index");

    return View(movie);
}
//
// POST: /Movies/Edit
[HttpPost]
public ActionResult Edit(Movie model)
{
    try
    {
        var movie = db.Movies.Find(model.ID);

        UpdateModel(movie);
        db.SaveChanges();
        return RedirectToAction("Details", new { id = model.ID });
    }
    catch (Exception)
    {
        ModelState.AddModelError("", "修改失败，请查看详细错误信息。");
    }

    return View(model);
}
```

这两个 Edit 方法中，第一个方法将在用户点击外部画面的“编辑”链接时被调用,用来在浏览器中显示数据修改视图，并且在该视图中显示用户选择编辑的数据。第二个 Edit 方法前面带有一个[HttpPost]标记，负责将修改数据视图中提交的表单数据绑定到一个用模型创建出来的 Movie 对象实例之上（当用户在表单中完成数据修改并点击保存按钮的时候进行提交），UpdateModel(movie)方法将调用模型拷贝器，该模型拷贝器将修改后的数据(使用 model 参数,该参数指向一个各属性值为编辑后数据的 Movie 对象实例)拷贝到数据库中(即为数据的保存过程）。在保存数据的过程中如果发生任何错误而导致保存失败的话，则画面重新返回到数据修改视图。

接下来让我们来追加该数据修改视图、在 Edit 方法中点击鼠标右键，选择“添加视图”，依然勾选“创建强类型视图”，模型类选择 Movie,在支架模板中选择“Edit”（修改数据），如图 9-4 所示。

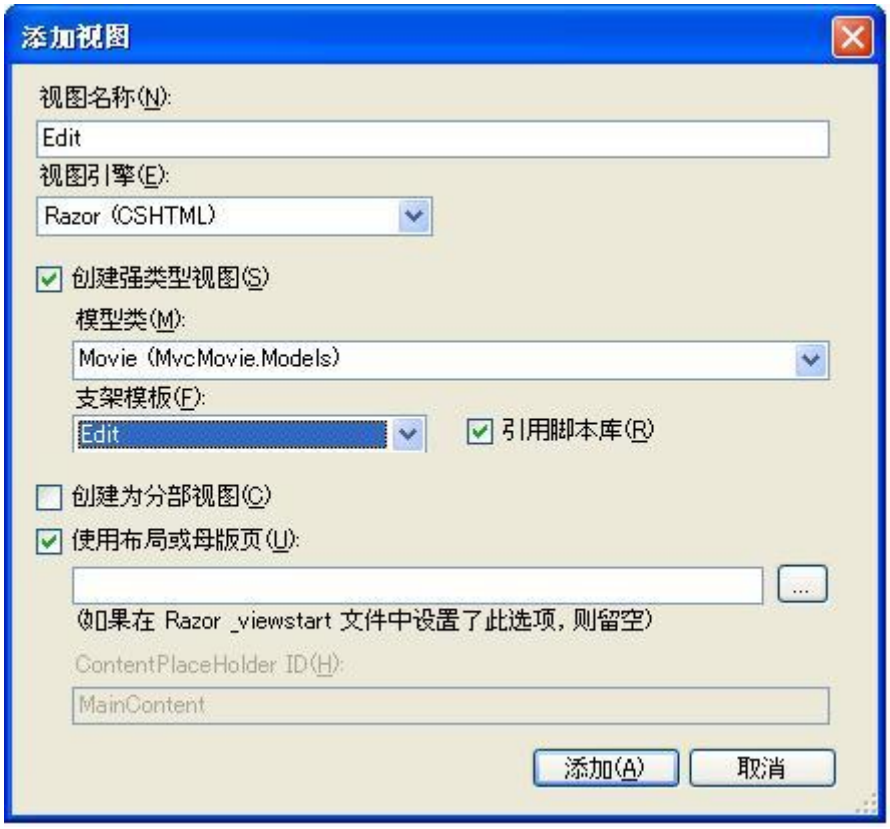


图 9-4 追加数据修改视图

如果要创建中文网站或应用程序，则将默认生成的 Edit.cshtml 文件中有关英文文字修改为中文，修改完毕后该文件中的代码如代码清单 9-3 中所示。

代码清单 9-3 Edit.cshtml 文件（修改数据视图）中的代码

```
@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "修改电影信息";
}
<h2>修改电影信息</h2>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
type="text/javascript"></script>
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>电影</legend>

        @Html.HiddenFor(model => model.ID)

        <div class="editor-label">
            标题
        </div>
```

```
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>

        <div class="editor-label">
            发行日期
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>

        <div class="editor-label">
            种类
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>

        <div class="editor-label">
            票价
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>

        <div class="editor-label">
            电影等级
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Rating)
            @Html.ValidationMessageFor(model => model.Rating)
        </div>

        <p>
            <input type="submit" value="保存" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("返回电影清单", "Index")
```

```
</div>
```

重新运行应用程序，在电影清单画面中点击某个电影的“编辑”链接，浏览器显示画面如图 9-5 所示。

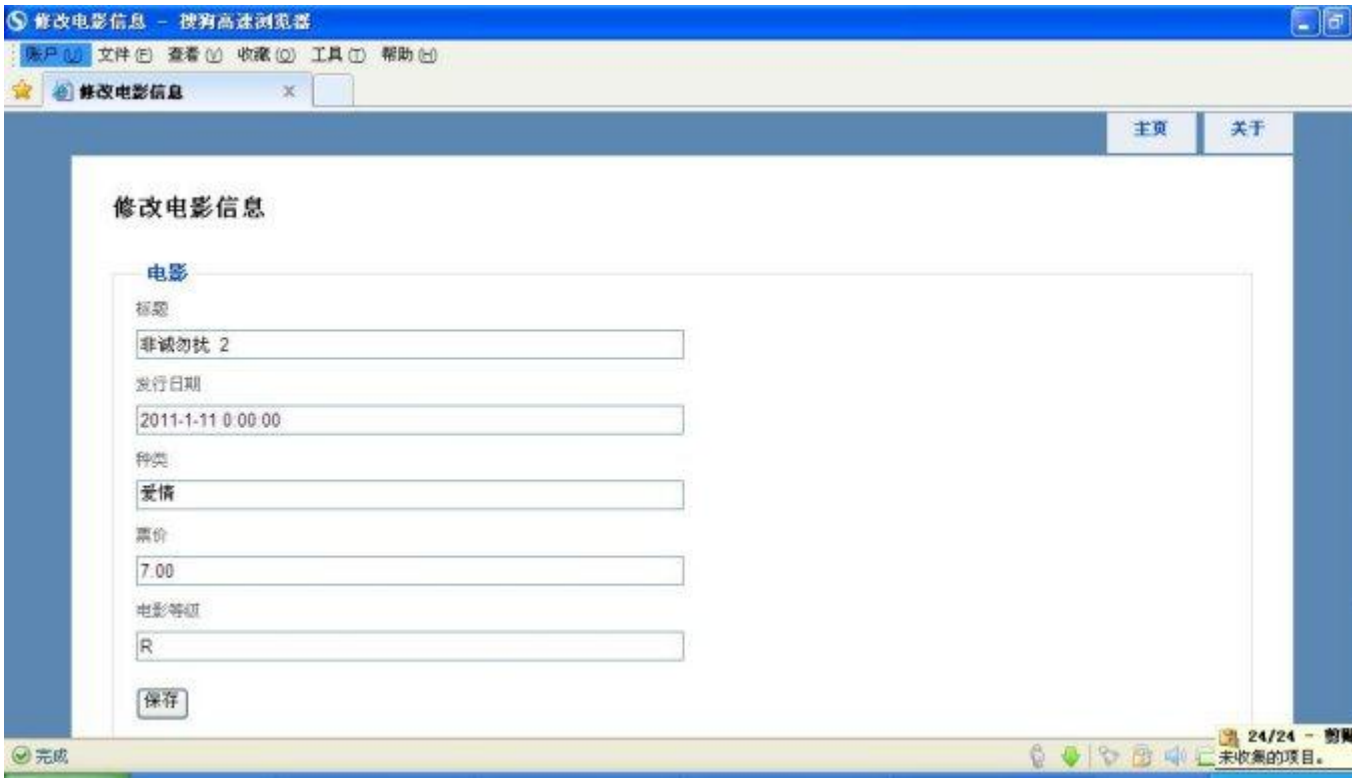


图 9-5 数据修改视图

在该视图中修改选中的数据，然后点击保存按钮，浏览器将修改后的数据显示在明细数据视图中，如图 9-6 所示。

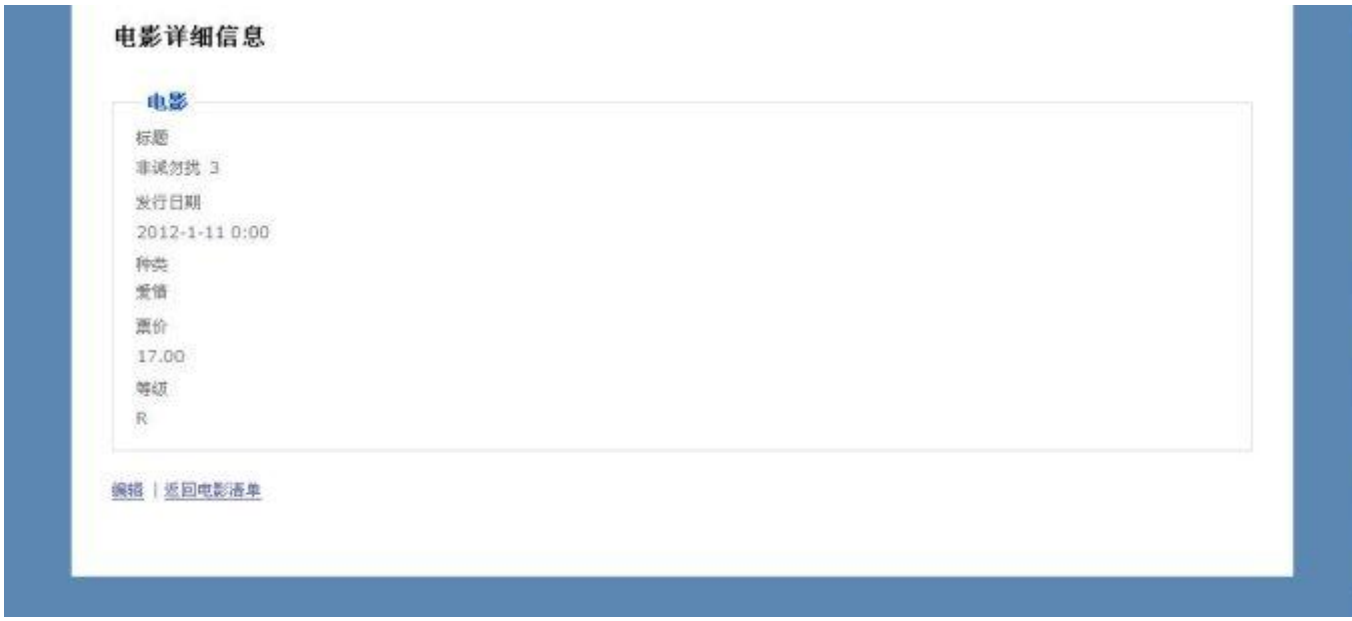


图 9-6 数据修改完成后被显示在明细数据视图中

9.3 实现数据的删除视图

接下来，让我们来看一下如何实现一个用来删除数据的视图。

首先打开 Movie 控制器，追加一个返回数据修改视图的 Edit 方法与一个对该视图中的表单提交进行处理的 Edit 方法，代码如下所示。

```
//
//GET: /Movies/Delete
public ActionResult Delete(int id)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
        return RedirectToAction("Index");

    return View(movie);
}
//
// POST: /Movies/Delete
[HttpPost]
public RedirectToRouteResult Delete(int id,FormCollection collection)
{
    var movie = db.Movies.Find(id);
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

这里请注意第一个没有[HttpPost]标记的 Delete 方法并不会将数据删除，因为如果通过 GET 请求而删除（或者追加、修改）删除数据的话都会打开一个安全漏洞。

接下来让我们来追加该数据删除视图、在 Delete 方法中点击鼠标右键，选择“添加视图”，依然勾选“创建强类型视图”，模型类选择 Movie,在支架模板中选择“Delete”（ 删除数据），如图 9-7 所示。

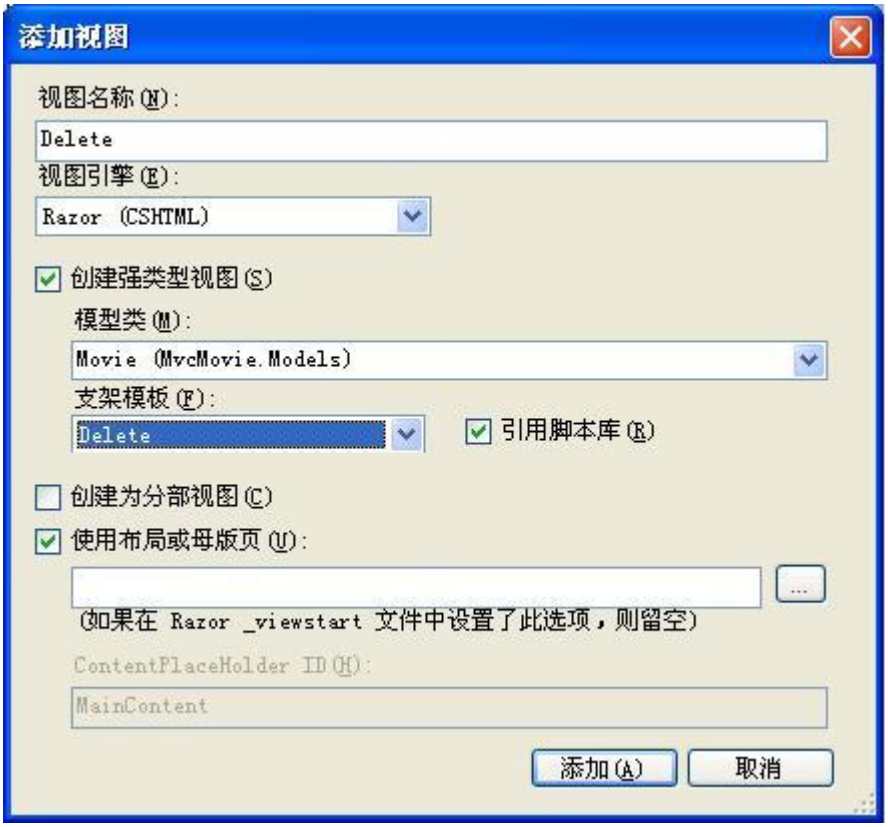


图 9-7　追加数据删除视图

如果要创建中文网站或应用程序，则将默认生成的 Delete.cshtml 文件中有关英文文字修改为中文，修改完毕后该文件中的代码如代码清单 9-3 中所示。

代码清单 9-3 Delete.cshtml 文件（删除数据视图）中的代码

```
@model MvcMovie.Models.Movie

@{
    ViewBag.Title = "删除电影数据";
}

<h2>删除电影数据</h2>

<h3>你确实想将这条电影数据删除吗?</h3>
<fieldset>
    <legend>电影</legend>

    <div class="display-label">标题</div>
    <div class="display-field">@Model.Title</div>

    <div class="display-label">发行日期</div>
    <div class="display-field">@String.Format("{0:d}",
        Model.ReleaseDate)</div>
    <div class="display-label">种类</div>
```

```
    <div class="display-field">@Model.Genre</div>
    <div class="display-label">票价</div>
    <div class="display-field">@String.Format("{0:F}", Model.Price)</div>
    <div class="display-label">电影等级</div>
    <div class="display-field">@Model.Rating</div>
</fieldset>
@using (Html.BeginForm()) {
    <p>
        <input type="submit" value="删除" /> |
        @Html.ActionLink("返回电影清单", "Index")
    </p>
}
```

在电影清单画面中点击一条数据的删除按钮，浏览器打开数据删除视图，如图 9-8 所示。

图 9-8 数据删除视图

点击删除按钮，该条数据将被删除，浏览器中返回显示电影清单画面。

最后，我们来回顾一下本教程中所讲述的内容。本教程中首先讲述了如何创建控制器、视图、如何将控制器中的数据传递给视图。然后我们设计并创建了一个数据模型。code-first 根据模型在指定的数据库服务器中创建了一个数据库。我们可以从这个数据库中获取数据并显示在一个 HTML 表格中。然后我们追加了一个添加数据所用的视图。接下来我们通过添加一个数据列（也称字段）的方式来改变数据表，同时修改了数据清单画面与数据追加视图来显示这个新追加的数据列。然后我们通过使用 DataAnnotations 命名空间，为数据模型标注属性的方式来追加了一些数据验证规则。这些数据验证即可以在客户端实现，也可以在服务器端实现。最后，我们添加代码与视图模板来创建了数据的修改视图，删除视图与明细数据视图。

接下来，我鼓励你继续看笔者即将发表的“MVC 音乐商店”这篇连载教程，来进一步了解一下 ASP.NET MVC 的实现方法。