第 8 章　Model 的验证（上篇）

ASP.NET MVC 采用 Model 绑定为目标 Action 生成相应的参数列表，但是在真正执行目标 Action 方法之前，还需要对绑定的参数实施验证以确保其有效性，我们将针对参数的验证称为 Model 验证。ASP.NET MVC 的 Model 验证不仅直接帮助我们实现了服务端验证，还为客户端验证生成相应的验证规则。

8.1 几种参数验证方式

ASP.NET MVC 的 Model 绑定系统能够从请求中提取相应的数据并为目标 Action 方法生成相应的参数对象，在目标 Action 方法执行之前针对参数的验证是一项十分必要的操作。除了采用预定义的规则对 Model 绑定生成的参数实施验证之外，ASP.NET MVC 下的 Model 验证还不得不考虑一个问题，那就是如何将验证结果保存下来并呈现在 View 中。

8.1.1 ModelError

通过前面对“Model 绑定”的介绍我们知道，ASP.NET MVC 在针对目标 Action 方法的某个参数实施绑定过程中会将由 ValueProvider 提供的数据（通过一个 ValueProviderResult 对象封装）保存在当前 Controller 的 ModelState 中。具体来说，Controller 的 ModelState 是其 ViewData 的一部分，它返回一个 ModelStateDictionary 对象，这是一个 Key 和 Value 类型分别为 String 和 ModelState 的字典对象。这里的 Key 实际上就是在 Model 绑定执行过程中传递给 ValueProvider 用于提取相应原始数据的那个 Key。

ModelState 对象维护的 Model 状态具有两种形式：一种是上面所说的通过 ValueProvider 提供的 ValueProviderResult 对象，对应着属性 Value；另一种是通过 ModelErrorCollection 类型表示的错误信息。ModelErrorCollection 是一个元素类型为 ModelError 的集合，而 ModelError 通过属性 ErrorMessage 和 Exception 表述错误的消息和抛出的异常。如下面的代码片段所示，所有的这些类型都是可序列化的。

```
[Serializable]
public class ModelState
{
    public ModelErrorCollection   Errors { get; }
    public ValueProviderResult    Value { get; set; }
}

[Serializable]
public class ModelErrorCollection : Collection<ModelError>
{
    public void Add(Exception exception);
    public void Add(string errorMessage);
}

[Serializable]
public class ModelError
{
```

```
    public ModelError(Exception exception);
    public ModelError(string errorMessage);
    public ModelError(Exception exception, string errorMessage);

    public string ErrorMessage { get; }
    public Exception Exception { get; }
}
```

如下面的代码片段所示，ModelStateDictionary 具有两个返回布尔值的方法。IsValidField 方法用于判断指定 Key 对应的 ModelState 是否有效，具体采用的有效性判断逻辑很简单：如果 ModelState 的 Errors 属性中包含一个或者多个 ModelError 则视为无效，如果该属性对应的集合为空则视为有效。至于 IsValid 方法，它帮助我们判断是否包含的所有 ModelState 均有效，我们经常会调用此方法判断请求提交的数据是否均通过验证。

```
public class ModelStateDictionary : IDictionary<string, ModelState>,
{
    //其他成员
    public bool IsValidField(string key);
    public bool IsValid { get; }}
}
```

通过 ModelError 对象封装的错误可以是对绑定的参数实施验证得到的错误信息，也可以是在执行 Action 方法过程中抛出的异常。如上面的代码片段所示，我们可以分别通过提供的错误消息和异常对象来创建一个 ModelError 对象，ModelErrorCollection 类型提供两个 Add 方法重载，它会根据提供的错误消息和异常对象来构建一个 ModelError 对象并添加到集合之中。

如下面的代码片段所示，ModelStateDictionary 为我们定义了两个 AddModelError 方法。当这两个方法被执行的时候，它会先根据提供的 Key 去提取对应的 ModelState 对象，并将创建的 ModelError 对象添加到它的 Errors 属性之中。如果这样的 ModelState 不存在，该方法会根据这个 Key 创建新的 ModelState 对象，创建的 ModelError 对象会被添加到它的 Errors 属性中。

```
public class ModelStateDictionary : IDictionary<string, ModelState>
    //其他成员
    public void AddModelError(string key, Exception exception);
    public void AddModelError(string key, string errorMessage);
}
```

8.1.2 验证消息的呈现

在很多情况下，我们以表单的方式向服务端提交数据，ASP.NET MVC 会提取表单元素的值并采用 Model 绑定生成目标 Action 方法对应的参数。如果针对参数的验证失败，相同的界面一般会再次呈现出来，用户输入的数据不仅会保留下来，相应的验证错误消息也会呈现出来。

那么当我们在进行 View 呈现的时候如何显示验证错误消息呢?

绑定的数据和验证错误均保存在当前 Controller 的 ModelState 中，后者是 ViewData 的一部分。所谓的 ViewData，实际上就是 Controller 传递给 View 的数据，所以我们在 View 中可以直接从当前 ModelState 中获取相应的验证错误。一旦得到了某个数据项对应的验证错误消息，我们就可以为它生成相应的 HTML 并最终将其呈现在 View 中。不过我们通常会调用帮助类 HtmlHelper 和 HtmlHelper<TModel>定义的扩展方法 ValidationMessage 和 ValidationMessageFor 来实现针对验证错误的呈现。

1．ValidationMessage/ValidationMessageFor

验证消息在 View 中的呈现可以借助 HtmlHelper/HtmlHelper<TModel>来实现。如下面的代码所示，HtmlHelper 和 HtmlHelper<TModel>分别提供了若干 ValidationMessage 和 ValidationMessageFor 扩展方法重载。

```
public static class ValidationExtensions
{
   public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
      string modelName);
   public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
      string modelName, IDictionary<string, object> htmlAttributes);
   //其他 ValidationMessage 方法重载

   public static MvcHtmlString ValidationMessageFor<TModel, TProperty>(
      this HtmlHelper<TModel> htmlHelper,
      Expression<Func<TModel, TProperty>> expression);
   public static MvcHtmlString ValidationMessageFor<TModel, TProperty>(
      this HtmlHelper<TModel> htmlHelper,
      Expression<Func<TModel, TProperty>> expression,
      string validationMessage);
   //其他 ValidationMessageFor 方法重载
}
```

HtmlHelper 的扩展方法 ValidationMessage 的参数 modelName 表示对应的 ModelState 在 ModelStateDictionary 中的 Key。如果针对这个 Key 找不到对应的 ModelState，或者对应的 ModelState 的 Errors 列表为空，意味着对应的数据成功通过验证，此时不会有任何 HTML 生成，否则该方法会生成一个<span>元素来显示验证消息。

如果通过 validationMessage 显式指定了验证消息，那么该消息将会直接作为该<span>元素的内部文本，否则 Errors 列表中第一个非空消息将会作为验证消息。除此之外，当我们调用扩展方法 ValidationMessage 时还可以通过参数 htmlAttributes 为这个<span>元素设置相应的 HTML 属性。ValidationMessageFor 与 ValidationMessage 的不同之处在于它会通过指定的表达式来提取

ValidationMessage 方法中的参数 modelName。

2．ValidationSummary

除了通过 ValidationMessageFor 与 ValidationMessage 这两个方法显示单条验证消息之外，我们还可以调用 HtmlHelper 的扩展方法 ValidationSummary 将所有的验证消息一并显示出来。如下面的代码片段所示，HtmlHelper 具有一系列 ValidationSummary 扩展方法重载，布尔类型的参数 excludePropertyErrors 表示是否需要排除基于属性的错误消息，而通过 message 参数可以为 ValidationSummary 指定一个作为标题的字符串。

```
public static class ValidationExtensions
{
    //其他成员
    public static MvcHtmlString ValidationSummary(
        this HtmlHelper htmlHelper);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        bool excludePropertyErrors);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        string message);
    //其他 ValidationSummary 方法重载
}
```

除了调用 ValidationMessage/ValidationMessageFor 和 ValidationSummary 显式控制错误消息在 View 中的呈现之外，当我们在一个强类型 View 中调用 HtmlHelper<TModel>的扩展方法 EditorForModel 将整个作为 Model 的数据对象以编辑模式呈现出来时，如果某个属性对应的 ModelState 具有相应的错误（通过 Errors 属性表示的 ModelError 集合不为空），错误消息也会一并呈现出来。当然，如果我们为 Model 类型定义了相应的模板就另当别论了。

8.1.3 手工验证绑定的参数

在定义具体 Action 方法的时候，对已经成功绑定的参数实施手工验证无疑是一种最为直接的编程方式，接下来我们通过一个简单的实例来演示如何将参数验证逻辑实现在对应的 Action 方法中，并在没有通过验证的情况下将错误信息响应给客户端。我们在一个 ASP.NET MVC 应用中定义了如下一个 Person 类作为被验证的数据类型，它的 Name、Gender 和 Age 属性分别表示一个人的姓名、性别和年龄。

```
public class Person
{
    [DisplayName("姓名")]
    public string Name { get; set; }
```

```
    [DisplayName("性别")]
    public string Gender { get; set; }

    [DisplayName("年龄")]
    public int? Age { get; set; }
}
```

接下来我们定义了如下一个 HomeController。在针对 GET 请求的 Action 方法 Index 中，我们创建了一个 Person 对象并将其作为 Model 呈现在对应的 View 中。另一个支持 POST 请求的 Index 方法具有一个 Person 类型的参数，我们在该 Action 方法中先调用 Validate 方法对这个输入参数实施验证。如果验证成功（ModelState.IsValid 属性返回 True），则会返回一个内容为“输入数据通过验证”的 ContentResult，否则会将此参数作为 Model 呈现在对应的 View 中。

```
public class HomeController : Controller
{
    [HttpGet]
    public ActionResult Index()
    {
        return View(new Person());
    }

    [HttpPost]
    public ActionResult Index(Person person)
    {
        Validate(person);

        if (!ModelState.IsValid)
        {
            return View(person);
        }
        else
        {
            return Content("输入数据通过验证");
        }
    }

    private void Validate(Person person)
    {
        if (string.IsNullOrEmpty(person.Name))
        {
            ModelState.AddModelError("Name", "'Name'是必需字段");
        }

        if (string.IsNullOrEmpty(person.Gender))
        {
            ModelState.AddModelError("Gender", "'Gender'是必需字段");
        }
        else if (!new string[] { "M", "F" , "m", "f" }.Any(
            g => string.Compare(person.Gender, g, true) == 0))
```

```
        {
            ModelState.AddModelError("Gender",
            "有效'Gender'必须是'M','F'之一");
        }

        if (null == person.Age)
        {
            ModelState.AddModelError("Age", "'Age'是必需字段");
        }
        else if (person.Age > 25 || person.Age < 18)
        {
            ModelState.AddModelError("Age", "有效'Age'必须在 18 到 25 周岁之间");
        }
    }
}
```

如上面的代码片段所示，我们在 Validate 方法中对作为参数的 Person 对象的 3 个属性进行逐条验证，如果提供的数据没有通过验证，我们会调用当前 ModelState 的 AddModelError 方法将指定的验证错误消息转换为 ModelError 保存起来。我们采用的具体验证规则如下。

- Person 对象的 Name、Gender 和 Age 属性均为必需字段，不能为 Null（或者空字符串）。
- 表示性别的 Gender 属性的值必须是“M”（Male）或者“F”（Female），其余的均为无效值。
- Age 属性表示的年龄必须在 18 到 25 周岁之间。

如下所示的是 Action 方法 Index 对应 View 的定义，这是一个 Model 类型为 Person 的强类型 View，它包含一个用于编辑人员信息的表单。我们调用 HtmlHelper<TModel>的扩展方法 ValidationSummary 对验证错误消息作统一显示。

```
@model Person
<html>
    <head>
        <title>编辑人员信息</title>
    </head>
    <body>
        @Html.ValidationSummary()
        @using (Html.BeginForm())
        {
            <div>@Html.LabelFor(m=>m.Name)</div>
            <div>@Html.EditorFor(m=>m.Name)</div>

            <div>@Html.LabelFor(m=>m.Gender)</div>
            <div>@Html.EditorFor(m => m.Gender)</div>

            <div>@Html.LabelFor(m=>m.Age)</div>
            <div>@Html.EditorFor(m => m.Age)</div>
```

```
            <input type="submit" value="保存"/>
        }
    </body>
</html>
```

直接运行该程序后，一个用于编辑人员基本信息的页面会被呈现出来，如果我们在输入不合法数据的情况下提交表单，相应的验证信息会以如图 8-1 所示的形式呈现出来。（S801）

图 8-1　以 ValidationSummary 形式呈现的验证错误信息

上面我们通过调用 HtmlHelper<TModel>的扩展方法 ValidationSummary 将所有的验证错误消息统一显示在当前界面的某个地方，其实在很多情况下我们会选择将消息紧贴被验证输入元素显示，为此我们对 View 作了如下的改动。

```
@using (Html.BeginForm())
{
    <div>@Html.LabelFor(m=>m.Name)</div>
    <div>
        @Html.EditorFor(m=>m.Name)
        @Html.ValidationMessage("Name")
    </div>

    <div>@Html.LabelFor(m=>m.Gender)</div>
    <div>
        @Html.EditorFor(m => m.Gender)
        @Html.ValidationMessage("Gender")
    </div>
```

```
    <div>@Html.LabelFor(m=>m.Age)</div>
    <div>
        @Html.EditorFor(m => m.Age)
        @Html.ValidationMessage("Age")
    </div>

    <input type="submit" value="保存"/>
}

或者
@using (Html.BeginForm())
{
    <div>@Html.LabelFor(m=>m.Name)</div>
    <div>
        @Html.EditorFor(m=>m.Name)
        @Html.ValidationMessageFor(m=>m.Name)
    </div>

    <div>@Html.LabelFor(m=>m.Gender)</div>
    <div>
        @Html.EditorFor(m => m.Gender)
        @Html.ValidationMessageFor(m => m.Gender)
    </div>

    <div>@Html.LabelFor(m=>m.Age)</div>
    <div>
        @Html.EditorFor(m => m.Age)
        @Html.ValidationMessageFor(m => m.Age)
    </div>

    <input type="submit" value="保存"/>
}
```

如上面的代码片段所示，我们分别调用 HtmlHelper 的扩展方法 ValidationMessage 和 HtmlHelper<TModel>的扩展方法 ValidationMessageFor 将验证错误消息显示在对应表单元素的右侧。如果运行该程序并在输入不合法数据的情况下提交表单，则会在浏览器中得到如图 8-2 所示的输出结果。（S802、S803）

图 8-2　以内联形式呈现的验证错误信息

我们还可以对 View 作相应的改动使之变得更加简单。如下面的代码片段所示，我们直接调用 HtmlHelper<TModel> 的扩展方法 EditorForModel 将作为 Model 的 Person 对象以编辑模式呈现在表单之中。运行此程序并在输入不合法数据的情况下提交表单，我们依然可以得到如图 8-2 所示的输出结果。（S804）

```
@using (Html.BeginForm())
{
    @Html.EditorForModel()
    <input type="submit" value="保存"/>
}
```

8.1.4　使用 ValidationAttribute 特性

将针对输入参数的验证逻辑和业务逻辑定义在 Action 方法中并不是一种值得推荐的编程方式。在大部分情况下，同一个数据类型在不同的应用场景中具有相同的验证规则，如果我们能将验证规则与数据类型关联在一起，让框架本身来实施数据验证，那么最终的开发者就可以将关注点更多地放在业务逻辑的实现上面。

实际上这也是 ASP.NET MVC 的 Model 验证系统默认支持的编程方式。当我们在定义数据类型的时候，可以在类型及其数据成员上面应用相应的 ValidationAttribute 特性来定义默认采用的验证规则。“System.ComponentModel.DataAnnotations”命名空间定义了一系列具体的 ValidationAttribute 特性类型，它们大都可以直接应用在自定义数据类型的某个属性上对目标数据成员实施验证。这些预定义验证特性不是本章论述的重点，我们会在“下篇”中对它们作一个概括性的介绍。

1．自定义 ValidationAttribute

常规验证可以通过预定义 ValidationAttribute 特性来完成，但是在很多情况下我们需要通过创建自定义的 ValidationAttribute 特性来解决一些特殊的验证。比如上面演示实例中针对 Person 对象的验证中，我们要求 Gender 属性指定的表示性别的值必须是“M/m”和“F/f”两者之一，这样的验证就不得不通过自定义的 ValidationAttribute 特性来实现。

针对“某个值必须在指定的范围内”这样的验证规则，我们定义了一个 DomainAttribute 特性。如下面的代码片段所示，DomainAttribute 为一个 IEnumerable<string>类型的只读属性 Values 提供了一个有效值列表，该列表在构造函数中被初始化。具体的验证实现在重写的 IsValid 方法中，如果被验证的值在这个列表中，则视为验证成功并返回 True。为了提供一个友好的错误消息，我们重写了方法 FormatErrorMessage。

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field,
   AllowMultiple = false)]
public class DomainAttribute : ValidationAttribute
{
   public IEnumerable<string> Values { get; private set; }

   public DomainAttribute(string value)
   {
      this.Values = new string[] { value };
   }

   public DomainAttribute(prams string[] values)
   {
      this.Values = values;
   }

   public override bool IsValid(object value)
   {
      if (null == value)
      {
         return true;
      }
      return this.Values.Any(item => value.ToString() == item);
   }

   public override string FormatErrorMessage(string name)
   {
      string[] values = this.Values.Select(value => string.Format("'{0}'",
         value)).ToArray();
      return string.Format(base.ErrorMessageString, name,string.Join(",",
         values));
   }
}
```

2．实例演示：针对 ValidationAttribute 特性的“自动化”验证（S805）

由于 ASP.NET MVC 在进行参数绑定的时候会自动提取应用在目标参数类型或者数据成员上的 ValidationAttribute 特性并利用它们对提供的数据实施验证，所以我们不再需要像上面演示的实例一样自行在 Action 方法中实施验证，而只需要在定义参数类型 Person 的时候应用相应的 ValidationAttribute 特性将采用的验证规则与对应的数据成员相关联。

我们在数据类型或者数据成员上应用 ValidationAttribute 特性的时候可以直接指定验证失败时描述验证结果的错误消息。如下面的代码片段所示，ValidationAttribute 特性提供了一个 ErrorMessage 属性供我们直接指定作为错误消息的字符串。为了避免相同的错误消息重复指定，同时也让我们可以对错误消息进行单独维护并提供多语言的支持，ValidationAttribute 特性支持将错误信息定义在资源文件中。

如果我们将错误消息定义在资源文件中，在应用 ValidationAttribute 特性的时候需要利用它的 ErrorMessageResourceType 和 ErrorMessageResourceName 属性来指定对定义资源文件自动生成的类型和对应资源项的名称。在我们演示的这个实例中，采用资源文件来定义验证的错误消息。

```
public abstract class ValidationAttribute : Attribute
{
    //其他成员
    public string     ErrorMessage { get;  set; }

    public Type       ErrorMessageResourceType { get;  set; }
    public string     ErrorMessageResourceName {  get;  set;  }
}
```

如下所示的是属性成员上应用了相关 ValidationAttribute 特性的 Person 类型的定义。我们在 3 个属性上均应用了 RequiredAttribute 特性将它们定义成必需的数据成员，Gender 和 Age 属性上则分别应用了 DomainAttribute 和 RangeAttribute 特性对有效属性值的范围作了相应限制。

```
public class Person
{
    [DisplayName("姓名")]
    [Required(ErrorMessageResourceName = "Required",
          ErrorMessageResourceType = typeof(Resources))]
    public string Name { get; set; }

    [DisplayName("性别")]
    [Required(ErrorMessageResourceName = "Required",
          ErrorMessageResourceType = typeof(Resources))]
    [Domain("M", "F", "m", "f", ErrorMessageResourceName = "Domain",
       ErrorMessageResourceType = typeof(Resources))]
    public string Gender { get; set; }
```

```
    [DisplayName("年龄")]
    [Required(ErrorMessageResourceName = "Required",
        ErrorMessageResourceType = typeof(Resources))]
    [Range(18, 25, ErrorMessageResourceName = "Range",
        ErrorMessageResourceType = typeof(Resources))]
    public int? Age { get; set; }
}
```

3 个 ValidationAttribute 特性采用的错误消息均定义在项目默认如图 8-3 所示的资源文件中（我们可以采用这样的步骤创建这个资源文件：右键选中项目文件，并在上下文菜单中选择“属性”选项打开“项目属性”对话框，然后在该对话框中选择“资源”Tab 页面，通过单击页面中的链接创建一个资源文件）。

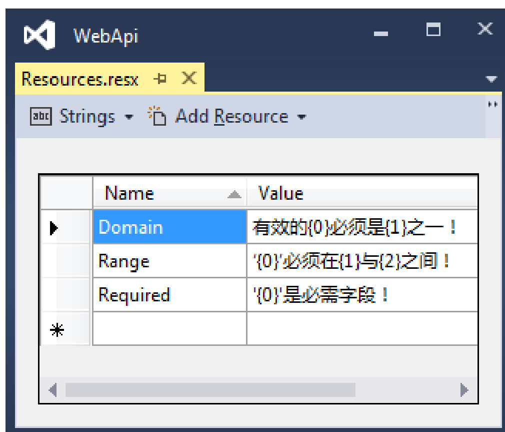

图 8-3 定义在资源文件中的错误消息

由于 ASP.NET MVC 会自动提取应用在绑定参数类型上的 ValidationAttribute 特性对绑定的参数实施自动化验证，所以我们根本不需要在具体的 Action 方法中来对参数作手工验证。如下面的代码片段所示，我们在 Action 方法 Index 中不再显式调用 Validate 方法，但是运行该程序并在输入不合法数据的情况下提交表单后依然会得到如图 8-2 所示的输出结果。

```
public class HomeController : Controller
{
    //其他成员
    [HttpPost]
    public ActionResult Index(Person person)
    {
        if (!ModelState.IsValid)
        {
            return View(person);
        }
```

```
        else
        {
            return Content("输入数据通过验证");
        }
    }
}
```

8.1.5 让数据类型实现 IValidatableObject 接口

除了将验证规则通过 ValidationAttribute 特性直接定义在数据类型上并让 ASP.NET MVC 在进行参数绑定过程中据此来验证参数之外，我们还可以将验证逻辑直接定义在数据类型中。既然我们将验证操作直接实现在了数据类型上，就意味着对应的数据对象具有"自我验证"的能力，我们姑且将这些数据类型称为"自我验证类型"。这些自我验证类型是实现了具有如下定义的接口 IValidatableObject，该接口定义在"System.ComponentModel.DataAnnotations"命名空间下。

```
public interface IValidatableObject
{
    IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext);
}
```

如上面的代码片段所示，IValidatableObject 接口具有唯一的方法 Validate，针对自身的验证就实现在该方法中。对于上面演示实例中定义的数据类型 Person，我们可以按照如下的形式将它定义成自我验证类型。

```
public class Person: IValidatableObject
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("性别")]
    public string Gender { get; set; }

    [DisplayName("年龄")]
    public int? Age { get; set; }

    public IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext)
    {
        Person person = validationContext.ObjectInstance as Person;
        if (null == person)
        {
            yield break;
        }
        if(string.IsNullOrEmpty(person.Name))
        {
```

```
            yield return new ValidationResult("'Name'是必需字段",
                new string[]{"Name"});
        }

        if (string.IsNullOrEmpty(person.Gender))
        {
            yield return new ValidationResult("'Gender'是必需字段",
                new string[] { "Gender" });
        }
        else if (!new string[]{"M","F"}.Any(
            g=>string.Compare(person.Gender,g, true) == 0))
        {
            yield return new ValidationResult("有效'Gender'必须是'M','F'之一",
                new string[] { "Gender" });
        }

        if (null == person.Age)
        {
            yield return new ValidationResult("'Age'是必需字段",
                new string[] { "Age" });
        }
        else if (person.Age > 25 || person.Age < 18)
        {
            yield return new ValidationResult("'Age'必须在 18 到 25 周岁之间",
                new string[] { "Age" });
        }
    }
}
```

如上面的代码片段所示，我们让 Person 类型实现了 IValidatableObject 接口。在实现的 Validate 方法中，我们从验证上下文中获取被验证的 Person 对象，并对其属性成员进行逐个验证。如果数据成员没有通过验证，我们通过一个 ValidationResult 对象封装错误消息和数据成员名称（属性名），该方法最终返回的是一个元素类型为 ValidationResult 的集合。在不对其他代码作任何改动的情况下，我们直接运行该程序并在输入不合法数据的情况下提交表单后依然会得到如图 8-2 所示的输出结果。（S806）

8.1.6 让数据类型实现 IDataErrorInfo 接口

上面我们让数据类型实现 IValidatableObject 接口并将具体的验证逻辑定义在实现的 Validate 方法中，这样的类型能够被 ASP.NET MVC 所识别，后者会自动调用该方法对绑定的数据对象实施验证。如果我们让数据类型实现 IDataErrorInfo 接口，也能实现类似的自动化验证效果。

IDataErrorInfo 接口定义在“System.ComponentModel”命名空间下，它提供了一种标准的错误信息定制方式。如下面的代码片段所示，IDataErrorInfo 具有两个成员，只读属性 Error 用于

获取基于自身的错误消息，而只读索引用于返回指定数据成员的错误消息。

```
public interface IDataErrorInfo
{
    string Error { get; }
    string this[string columnName] { get; }
}
```

同样是针对上面演示的实例，现在我们对需要被验证的数据类型 Person 进行了重新定义。如下面的代码片段所示，我们让 Person 实现了 IDataErrorInfo 接口。在实现的索引中，我们将索引参数 columnName 视为属性名称按照上面的规则对相应的属性成员实施验证，并在验证失败的情况下返回相应的错误消息。在不对其他代码作任何改动的情况下，我们直接运行该程序并在输入不合法数据的情况下提交表单后依然会得到如图 8-2 所示的输出结果。（S807）

```
public class Person : IDataErrorInfo
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("性别")]
    public string Gender { get; set; }

    [DisplayName("年龄")]
    public int? Age { get; set; }

    [ScaffoldColumn(false)]
    public string Error { get; private set; }

    public string this[string columnName]
    {
        get
        {
            switch (columnName)
            {
                case "Name":
                    {
                        if(string.IsNullOrEmpty(this.Name))
                        {
                            return "'姓名'是必需字段";
                        }
                        return null;
                    }
                case "Gender":
                    {
                        if (string.IsNullOrEmpty(this.Gender))
                        {
                            return "'性别'是必需字段";
                        }
                        else if (!new string[] { "M", "F"}.Any(
```

```
                        g => string.Compare(this.Gender, g, true) == 0))
                    {
                        return "'性别'必须是'M','F'之一";
                    }
                    return null;
                }
            case "Age":
                {
                    if (null == this.Age)
                    {
                        return "'年龄'是必需字段";
                    }
                    else if (this.Age > 25 || this.Age < 18)
                    {
                        return "'年龄'必须在18到25周岁之间";
                    }
                    return null;
                }
            default: return null;

        }
    }
  }
}
```

8.2 ModelValidator 及其提供策略

ASP.NET MVC 并不会对绑定的所有参数对象实施验证，真正能够被自动验证的仅限于复杂类型的参数。其实这也很好理解，不论是通过应用的 ValidationAttribute 特性来定义验证规则，还是将验证逻辑定义在实现了接口 IValidatableObject/IDataErrorInfo 的类型中，被验证的数据类型都是自定义的“复杂类型”。

8.2.1 ModelValidator 与 ModelValidatorProvider

虽然 Model 绑定的方式因被验证数据类型的差异而有所不同，但是 ASP.NET MVC 总是使用一个名为 ModelValidator 的对象来对绑定的数据对象实施验证。所有的 ModelValidator 类型均继承自具有如下定义的抽象类 ModelValidator。它的 GetClientValidationRules 方法返回一个元素类型为 ModelClientValidationRule 的集合，而 ModelClientValidationRule 是对客户端验证规则的封装，我们会在客户端验证部分对其进行详细介绍。

```
public abstract class ModelValidator
{
```

```
    //其他成员
    public virtual IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
    public abstract IEnumerable<ModelValidationResult> Validate(
        object container);

    public virtual bool IsRequired { get; }
}
```

针对目标数据的验证是通过调用 Validate 方法来完成的，该方法的输入参数 container 表示的正是被验证的对象。正是因为被验证的总是一个复杂类型的对象，后者又被称为一个具有若干数据成员的“容器”对象，所以对应的参数被命名为 container。Validate 方法表示验证结果的返回值并不是一个简单的布尔值，而是一个元素类型为具有如下定义的 ModelValidationResult 对象集合。

```
public class ModelValidationResult
{
    public string MemberName { get; set; }
    public string Message { get; set; }
}
```

ModelValidationResult 具有两个字符串类型属性 MemberName 和 Message，前者代表被验证数据成员的名称，后者表示错误消息。一般来说，如果 ModelValidationResult 对象来源于针对容器对象本身的验证，那么它的 MemberName 属性为空字符串。对于针对容器对象某个属性的验证来说，属性名称会作为返回的 ModelValidationResult 对象的 MemberName 属性。

ModelValidationResult 集合只有在验证失败的情况下才会返回。如果被验证的数据对象符合所有的验证规则，Validate 方法会直接返回 Null 或者一个空 ModelValidationResult 集合。值得一提的是，我们有时候会用 ValidationResult 的静态只读字段 Success 表示成功通过验证的结果，实际上该字段的值就是 Null。

```
public class ValidationResult
{
    //其他成员
    public static readonly ValidationResult Success;
}
```

ModelValidator 具有一个布尔类型的只读属性 IsRequired，表示该 ModelValidator 是否对目标数据进行“必需性”验证（即被验证的数据成员必须具有一个具体的值），该属性默认返回 False。我们可以通过应用 RequiredAttribute 特性将某个属性定义成“必需”的数据成员。

通过前面章节的介绍我们知道，ASP.NET MVC 大都采用 Provider 的模式来提供相应的组件，比如描述 Model 元数据的 ModelMetadata 通过对应的 ModelMetadataProvider 来提供，实现 Model 绑定的 ModelBinder 则可以通过对应的 ModelBinderProvider 来提供，用于实现 Model 验

证的 ModelValidator 也不例外，它对应的提供者为 ModelValidatorProvider，对应的类型继承自具有如下定义的抽象类 ModelValidatorProvider。

```
public abstract class ModelValidatorProvider
{
    public abstract IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context);
}
```

如上面的代码片段所示，GetValidators 方法具有两个参数，一个是用于描述被验证类型或者属性 Model 元数据的 ModelMetadata 对象，另一个是当前 ControllerContext。该方法返回的是一个元素类型为 ModelValidator 的集合。

ASP.NET MVC通过静态类型ModelValidatorProviders 对使用的ModelValidatorProvider 进行注册。如下面的代码片段所示，ModelValidatorProviders 具有一个静态只读属性 Providers，对应的类型为 ModelValidatorProviderCollection，它表示基于整个 Web 应用范围的全局 ModelValidatorProvider 集合。

```
public static class ModelValidatorProviders
{
    public static ModelValidatorProviderCollection Providers { get; }
}

public class ModelValidatorProviderCollection :
    Collection<ModelValidatorProvider>
{
    public ModelValidatorProviderCollection();
    public ModelValidatorProviderCollection(
        IList<ModelValidatorProvider> list);
    public IEnumerable<ModelValidator> GetValidators(ModelMetadata metadata,
        ControllerContext context);
}
```

值得一提的是，用于描述Model 元数据的ModelMetadata 类型具有如下一个 GetValidators 方法，它返回的 ModelValidator 列表正是利用注册到 ModelValidatorProviders 静态属性 Providers 上的 ModelValidatorProvider 创建的。

```
public class ModelMetadata
{
    //其他成员
    public virtual IEnumerable<ModelValidator> GetValidators(
        ControllerContext context);
}
```

如图 8-4 所示的 UML 列出了组成 Model 验证系统的 3 个核心类型。具体的 Model 验证工作总是通过某个具体的 ModelValidator 来完成，作为 ModelValidator 提供者的

ModelValidatorProvider 注册在静态类型 ModelValidatorProviders 之上。

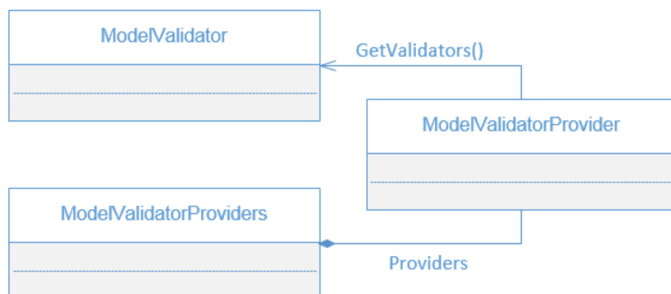

图 8-4　组成 Model 验证系统的 3 个核心类型

8.2.2　DataAnnotationsModelValidator

我们在上面一节中介绍了 3 种不同的“自动化验证”的编程方式，ASP.NET MVC 在内部会采用不同的 ModelValidator 来对绑定的参数实施验证。一个具体的 ModelValidator 通常由相应的 ModelValidatorProvider 来提供，在本节接下来的内容中我们将对 ASP.NET MVC 提供的原生的 ModelValidator 和对应的 ModelValidatorProvider 作详细的介绍。

对于上面提到的这 3 种验证编程方式，第一种（利用应用在数据类型或其数据成员上的 ValidationAttribute 特性来定义相应的验证规则）是最为常用的，这样的 Model 验证最终通过一个 DataAnnotationsModelValidator 对象来完成。一个 DataAnnotationsModelValidator 对象实际上是对一个 ValidationAttribute 特性的封装，这可以从如下所示的定义看出来。

```
public class DataAnnotationsModelValidator : ModelValidator
{
    public DataAnnotationsModelValidator(ModelMetadata metadata,
        ControllerContext context, ValidationAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();

    public override IEnumerable<ModelValidationResult> Validate(
        object container);

    protected internal ValidationAttribute Attribute { get; }
    protected internal string              ErrorMessage { get; }
    public override bool                   IsRequired { get; }
}
```

DataAnnotationsModelValidator 的提供者为 DataAnnotationsModelValidatorProvider。采用应用的 ValidationAttribute 特性将验证规则直接与数据类型关联是我们最为常用的编程方式，为了

让读者对此具有深刻的认识，我们会在第 9 章中对实现在 DataAnnotationsModelValidator 中的验证机制及 DataAnnotationsModelValidatorProvider 针对它的提供方式进行单独介绍。

8.2.3 ValidatableObjectAdapter

如果被验证的数据类型实现了 IValidatable 接口，ASP.NET MVC 会自动调用实现的 Validate 方法对其实施验证，此时创建的 ModelValidator 是一个 ValidatableObjectAdapter 对象。ValidatableObjectAdapter 定义如下，其 Validate 方法的实现逻辑很简单：它直接调用被验证对象的 Validate 方法，并将返回的 ValidationResult 对象转换成 ModelValidationResult 类型。

```
public class ValidatableObjectAdapter : ModelValidator
{
   public ValidatableObjectAdapter(ModelMetadata metadata,
      ControllerContext context);
   public override IEnumerable<ModelValidationResult> Validate(
      object container);
}
```

虽然 ValidatableObjectAdapter 继承自 ModelValidator，但是 ASP.NET MVC 貌似没有将其视为一个真正意义上的 ModelValidator，而是将其视为一个“适配器（Adapter）”。ASP.NET MVC 也没有为 ValidatableObjectAdapter 定义单独的 ModelValidatorProvider，它的提供者其实是上面提到过的 DataAnnotationsModelValidatorProvider，至于它对 ValidatableObjectAdapter 采用的提供策略，我们会在“下篇”中对其进行详细介绍。

8.2.4 DataErrorInfoModelValidator

如果我们让数据类型实现 IDataErrorInfo 接口，可以利用实现的 Error 属性和索引提供针对自身及所属数据成员的验证错误信息。针对这样的数据类型，ASP.NET MVC 最终会创建一个 DataErrorInfoModelValidator 对象来对其实施验证，DataErrorInfoClassModelValidator 和 DataErrorInfoPropertyModelValidator 是两个具体的 DataErrorInfoModelValidator。

DataErrorInfoClassModelValidator 和 DataErrorInfoPropertyModelValidator 是两个内部类型。前者针对容器对象自身实施验证，所以它只需要从实现的 Error 属性中提取错误消息并将其转换成返回的 ModelValidationResult 对象。后者则专门验证容器对象的某个属性，它在实现的 Validate 方法中会利用属性名从实现的索引中提取相应的错误消息并将其转换成返回的 ModelValidationResult 对象。

```
internal sealed class DataErrorInfoClassModelValidator : ModelValidator
{
   public DataErrorInfoClassModelValidator(ModelMetadata metadata,
      ControllerContext controllerContext);
   public override IEnumerable<ModelValidationResult> Validate(
      object container);
}

internal sealed class DataErrorInfoPropertyModelValidator : ModelValidator
{
   public DataErrorInfoPropertyModelValidator(ModelMetadata metadata,
      ControllerContext controllerContext);
   public override IEnumerable<ModelValidationResult> Validate(
      object container);
}
```

ASP.NET MVC 最终利用具有如下定义的 DataErrorInfoModelValidatorProvider 来提供这两种类型的 DataErrorInfoModelValidator。对于其实现的 GetValidators 方法来说，如果被验证对象的类型实现了 IDataErrorInfo 接口，它会创建一个 DataErrorInfoClassModelValidator 对象并添加到返回的 ModelValidator 列表中。如果被验证的是容器类型的某个属性值，并且容器类型实现了 IDataErrorInfo 接口，它会创建一个 DataErrorInfoPropertyModelValidator 对象并添加到返回的 ModelValidator 列表中。

```
public class DataErrorInfoModelValidatorProvider : ModelValidatorProvider
{
   public override IEnumerable<ModelValidator> GetValidators(
      ModelMetadata metadata, ControllerContext context);
}
```

8.2.5 ClientModelValidator

ASP.NET MVC 不仅仅可以利用 ModelValidator 在服务端对绑定的参数对象实施验证，而且还对客户端验证提供了支持。虽然客户端验证是由 JavaScript 来实现的，但是针对被验证的表单元素的验证规则必须出现在最终生成的 HTML 中。

顾名思义，ClientModelValidator 是一种专门针对客户端验证的 ModelValidator，当我们调用 HtmlHelper<TModel>的模板方法 EditorFor/EditorForModel 将某个数据对象或其属性成员以编辑模式呈现在某个 View 中时，ASP.NET MVC 会利用对应的 ClientModelValidator 来提取客户端验证规则并将其写入到最终生成的 HTML 中。

ClientModelValidator 是定义在程序集“System.Web.Mvc.dll”中的内部类型。如下面的代码片段所示，当我们通过调用构造函数创建一个 ClientModelValidator 的时候，不仅需要指定描述被验证对象类型的 ModelMetadata 和当前 ControllerContext，还需要以字符串的形式指定验证类

型和错误消息。

```
internal class ClientModelValidator : ModelValidator
{
   public ClientModelValidator(ModelMetadata metadata,
      ControllerContext controllerContext, string validationType,
      string errorMessage);

   public sealed override IEnumerable<ModelClientValidationRule>
      GetClientValidationRules();
   public sealed override IEnumerable<ModelValidationResult> Validate(
      object container);
}
```

由于 ClientModelValidator 仅限于客户端验证，用于实现服务端验证的 Validate 方法总是返回一个空的 ModelValidationResult 集合（表示验证成功）。它的 GetClientValidationRules 方法返回的是一个元素类型为 ModelClientValidationRule 的集合，表示需要出现在 HTML 中的客户端验证规则。

ClientModelValidator 具有两个继承者，分别是针对数值类型和日期类型验证的 NumericModelValidator 和 DateModelValidator。如下面的代码片段所示，这两个 ClientModelValidator 采用的验证类型的分别是“number”和“date”。表示错误消息的字符串是从内部维护的资源文件中获取的，这实际上带来了一个问题，就是我们无法对错误消息进行定制。

```
internal sealed class NumericModelValidator :ClientModelValidator
{
   public NumericModelValidator(ModelMetadata metadata,
      ControllerContext controllerContext)
      : base(metadata, controllerContext, "number",
      ClientDataTypeModelValidatorProvider.GetFieldMustBeNumericResource(
      controllerContext))
   {}
}

internal sealed class DateModelValidator :ClientModelValidator
{
   public DateModelValidator(ModelMetadata metadata,
      ControllerContext controllerContext)
      : base(metadata, controllerContext, "date",
      ClientDataTypeModelValidatorProvider.GetFieldMustBeDateResource(
      controllerContext))
   {}
}
```

NumericModelValidator 和 DateModelValidator 这两种 ClientModelValidator 最终是通过具有如下定义的 ClientDataTypeModelValidatorProvider 来提供的。在实现的 GetValidators 方法中，ClientDataTypeModelValidatorProvider 会根据指定的ModelMetadata 判断被验证类型是否属于数

字或者 DateTime 类型，如果是则直接返回一个包含单个 NumericModelValidator 或者 DateModelValidator 对象的 ModelValidator 集合。在这里被视为数字的类型包括 byte、sbyte、short、ushort、int、uint、long、ulong、float、double 和 decimal 等。

```
public class ClientDataTypeModelValidatorProvider : ModelValidatorProvider
{
   public ClientDataTypeModelValidatorProvider();
   public override IEnumerable<ModelValidator> GetValidators(
      ModelMetadata metadata, ControllerContext context);
}
```

我们已经介绍了 3 种类型的 ModelValidator 和它们对应的 ModelValidatorProvider（ValidatableObjectAdapter 的提供者为 DataAnnotationsModelValidatorProvider），ASP.NET MVC 在默认情况下会使用哪些呢？如下所示的是用于注册 ModelValidatorProvider 的静态类型 ModelValidatorProviders 的完整定义，我们可以看出 3 个 ModelValidatorProvider 对象会被默认注册，其类型正是上面介绍的这 3 个。

```
public static class ModelValidatorProviders
{
   private static readonly ModelValidatorProviderCollection _providers;

   static ModelValidatorProviders()
   {
      ModelValidatorProviderCollection providers =
         new ModelValidatorProviderCollection();
      providers.Add(new DataAnnotationsModelValidatorProvider());
      providers.Add(new DataErrorInfoModelValidatorProvider());
      providers.Add(new ClientDataTypeModelValidatorProvider());
      _providers = providers;
   }

   public static ModelValidatorProviderCollection Providers
   {
      get
      {
         return _providers;
      }
   }
}
```

8.2.6 CompositeModelValidator

虽然 CompositeModelValidator 仅仅是定义在程序集“System.Web.Mvc.dll”中的一个私有类型而已，但是它在 ASP.NET MVC 的 Model 验证系统中具有重要的地位，可以说真正用于 Model 验证的 ModelValidator 就是这么一个对象。

```
private class CompositeModelValidator : ModelValidator
{
    public CompositeModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext);
    public override IEnumerable<ModelValidationResult> Validate(
        object container);
}
```

从命名上可以看出，CompositeModelValidator 实际上并不是一个真正对数据对象实施验证的 ModelValidator，而是一系列 ModelValidator 的组合。它根据描述数据类型或其属性的 ModelMetadata 动态地获取相应的 ModelValidator（通过调用 ModelMetadata 的 GetValidators 方法）对目标数据实施验证。抽象类 ModelValidator 具有一个静态的 GetModelValidator 方法，它会根据指定的 ModelMetadata 和 ControllerContext 得到相应的 ModelValidator 对象。如下面的代码片段所示，它返回的正是一个 CompositeModelValidator 对象。

```
public abstract class ModelValidator
{
    //其他成员
    public static ModelValidator GetModelValidator(ModelMetadata metadata,
        ControllerContext context)
    {
        return new CompositeModelValidator(metadata, context);
    }
}
```

当 CompositeModelValidator 被用于验证某个容器对象的时候，它会先验证其属性成员。针对容器对象自身的验证只有在所有属性值都通过验证的情况下才会进行。具体的逻辑是这样的：它通过调用描述容器类型的 ModelMetadata 对象的 Properties 属性得到描述所有属性成员的 ModelMetadata 对象，然后调用它们的 GetValidators 方法得到一组 ModelValidator 对相应的属性值实施验证，验证得到的 ModelValidationResult 会被添加到最终返回的 ModelValidationResult 列表中。

如果在对所有属性实施验证之后该 ModelValidationResult 列表依然为空（所有的属性均成功通过验证），CompositeModelValidator 才会获取针对容器类型的 ModelMetadata 对象，并采用调用其 GetValidators 方法获取的 ModelValidator 列表对容器对象本身实施验证。表示验证结果的 ModelValidationResult 对象被添加到最终返回的列表中。

实例演示：CompositeModelValidator 采用的验证行为（S808, S809）

为了使读者对 CompositeModelValidator 的验证逻辑有一个深刻的理解，我们来演示一个具体的实例。我们在一个 ASP.NET MVC 应用中定义了如下一个名为 AlwaysFailsAttribute 的验证特性。如下面的代码片段所示，由于重写的 IsValid 方法总是返回 False，意味着针对数据的验证总

是会失败。我们还重写了只读属性 TypeId，让它真正能够唯一标识一个 AlwaysFailsAttribute 特性实例（具体原因我们会在本章后续部分予以介绍）。

```
[AttributeUsage( AttributeTargets.Class| AttributeTargets.Property)]
public class AlwaysFailsAttribute : ValidationAttribute
{
   private object typeId;
   public override bool IsValid(object value)
   {
      return false;
   }
   public override object TypeId
   {
      get { return typeId ?? (typeId = new object()); }
   }
}
```

我们将 AlwaysFailsAttribute 特性应用到表示联系人的 Contact 类型上。如下面的代码片段所示，我们在 Contact 和 Address 的类型和各自的属性上都应用了该特性，并且指定了相应的错误消息。

```
[AlwaysFails(ErrorMessage = "Contact")]
public class Contact
{
   [AlwaysFails(ErrorMessage = "Contact.Name")]
   public string Name { get; set; }

   [AlwaysFails(ErrorMessage = "Contact.PhoneNo")]
   public string PhoneNo { get; set; }

   [AlwaysFails(ErrorMessage = "Contact.EmailAddress")]
   public string EmailAddress { get; set; }

   [AlwaysFails(ErrorMessage = "Contact.Address")]
   public Address Address { get; set; }
}

[AlwaysFails(ErrorMessage = "Address")]
public class Address
{
   [AlwaysFails(ErrorMessage = "Address.Province")]
   public string Province { get; set; }

   [AlwaysFails(ErrorMessage = "Address.City")]
   public string City { get; set; }

   [AlwaysFails(ErrorMessage = "Address.District")]
   public string District { get; set; }
```

```
    [AlwaysFails(ErrorMessage = "Address.Street")]
    public string Street { get; set; }
}
```

我们创建了一个具有如下定义的 HomeController 类，并在 Action 方法 Index 中使用当前注册的 ModelMetadataProvider 创建了一个描述 Contact 类型的 ModelMetadata 对象，然后将它和当前 ControllerContext 作为参数调用抽象类型 ModelValidator 的静态方法 GetValidator 创建一个 CompositeModelValidator 对象。我们利用该 CompositeModelValidator 来验证创建的 Contact 对象，并将表示验证结果的 ModelValidationResult 列表作为 Model 呈现在默认的 View 中。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        Address address = new Address
        {
            Province   = "江苏",
            City   = "苏州",
            District   = "工业园区",
            Street     = "星湖街 328 号"
        };

        Contact contact = new Contact
        {
            Name            = "张三",
            PhoneNo         = "123456789",
            EmailAddress    = "zhangsan@gmail.com",
            Address         = address
        };

        ModelMetadata metadata = ModelMetadataProviders.Current
            .GetMetadataForType(() => contact, typeof(Contact));
        ModelValidator validator = ModelValidator.GetModelValidator(metadata,
            ControllerContext);
        return View(validator.Validate(contact));
    }
}
```

如下所示的是 Action 方法 Index 对应 View 的定义，这是一个 Model 类型为 IEnumerable <ModelValidationResult> 的强类型 View。我们在该 View 中将包含在集合中的每一个 ModelValidationResult 对象的成员名称和错误消息通过表格的形式呈现出来。

```
@model IEnumerable<ModelValidationResult>
<html>
  <head>
      <title>验证结果</title>
  </head>
  <body>
      <table >
          <tr><th>Member</th><th>Message</th></tr>
          @foreach (ModelValidationResult result in Model)
          {
              string propertyName = string.IsNullOrEmpty(result.MemberName) ?
                  "N/A" : result.MemberName;
              <tr><td>@propertyName</td><td>@result.Message</td></tr>
          }
      </table>
  </body>
</html>
```

该程序运行后会在浏览器中呈现出如图 8-5 所示的结果，可以看出 CompositeModelValidator 对 Contact 对象实施验证得到的 5 个 ModelValidationResult 都来源于针对 4 个属性的验证，应用在 Contact 类型上的 AlwaysFailsAttribute 特性并没有参与验证。（S808）

Member	Message
Name	Contact.Name
PhoneNo	Contact.PhoneNo
EmailAddress	Contact.EmailAddress
Address	Contact.Address
Address	Address

图 8-5　CompositeModelValidator 的验证规则（1）

按照前面介绍的“针对容器对象本身的验证只有在所有属性通过验证的情况下才会进行”的原理，为了让 Contact 的 4 个属性通过验证，我们将应用在 4 个属性和 Address 类型上的 AlwaysFailsAttribute 特性注释掉，只保留应用在 Contact 类型和 Address 4 个属性上的 AlwaysFailsAttribute 特性。再次运行我们的程序将会在浏览器中得到如图 8-6 所示的输出结果，不

难看出输出的 ModelValidationResult 来源于应用在 Contact 类型上的 AlwaysFailsAttribute 特性。（S809）

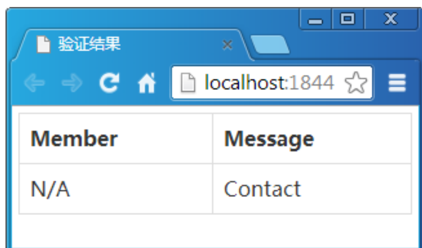

图 8-6 CompositeModelValidator 的验证规则（2）

8.3 Model 验证的实施

Model 绑定解决了针对目标 Action 方法参数的初始化问题，而 Model 验证的目的在于对绑定的参数对象实施验证以确保输入数据的有效性，Model 验证是伴随着 Model 绑定进行的。在上面一节中我们详细地介绍了真正用于 Model 验证的 ModelValidator 及相关的提供机制，接下来讨论在这个以 ModelValidator 为核心的 Model 验证系统中，针对通过 Model 绑定得到的数据对象的验证是如何实现的。

8.3.1 Model 绑定过程中的验证

在前面我们不止一次地提到，Model 验证可以看成是 Model 绑定的一个中间环节，默认情况下采用的 Model 绑定实现在 DefaultModelBinder 中。那么现在有这么一个问题：是 DefaultModelBinder 得到最终的参数对象后再递交给 ModelValidator 实施验证呢，还是它在实施 Model 绑定的过程中动态地调用 ModelValidator 对由 ValueProvider 提供的数据值实施验证？

实际上我们前面演示的两个实例已经回答了这个问题。通过前面演示的两个例子我们知道，CompositeModelValidator 这个默认 ModelValidator 在进行 Model 验证过程中并不是递归进行的（S808），但是从整个 Model 绑定过程来看，Model 验证却具有递归性，所以 Model 绑定和 Model 验证绝对不可能是一前一后的过程，唯一的可能是 DefaultModelBinder 在递归地进行 Model 绑定的过程中调用 ModelValidator 对提供的数据实施验证。

同样以针对 Contact 类型的 Model 绑定为例，当 DefaultModelBinder 通过得到一个被初始

化的空 Contact 对象之后，会将描述 Contact 类型的 ModelMetadata 对象作为参数调用 ModelValidator 的静态方法 GetModelValidator，得到的 CompositeModelValidator 会被用于对 Contact 对象实施验证。由于 CompositeModelValidator 的 Model 验证不具有递归性，所以只有应用在 Contact 4 个属性（Name、PhoneNo、Email 和 Address）及其自身类型上的验证规则在本轮验证中有效。

由于 Contact 的 Address 属性是一个复杂类型，所以 DefaultModelBinder 在针对 Contact 类型的 Model 绑定过程中会递归地创建一个空 Address 对象作为 Contact 对象的 Address 属性。在完成对 Address 对象的绑定之后，又会调用 ModelValidator 的静态方法 GetModelValidator 根据描述 Address 类型的 ModelMetadata 得到一个 CompositeModelValidator，初始化后的 Address 对象将交给它验证。

描述 Model 元数据的 ModelMetadata 具有一个树形层次化结构，我们的验证规则可以应用到每一个节点上。DefaultModelBinder 就是在递归地绑定复杂对象的过程中对绑定后的对象实施验证，从而使各个层次上的验证得以实现。不过 CompositeModelValidator 只有在所有属性值都验证通过的情况下，才会使用应用在类型上的验证规则对数据对象实施验证，所以验证的结果也不能完全反映所有的验证规则。

8.3.2 实例演示：模拟 Model 绑定中的验证（S810）

在第 7 章“Model 的绑定（下）”中，我们自定义了一个 MyDefaultModelBinder 实现了针对简单类型、复杂类型、集合和字典的绑定。在本例中，我们在这个自定义的 ModelBinder 中引入 Model 验证部分。通过前面的介绍我们知道，真正实施 Model 验证的是通过 ModelValidator 的静态方法 GetModelValidator 创建的 CompositeModelValidator 对象，我们按照其采用的 Model 验证逻辑自定义了如下一个 MyCompositeModelValidator 类型。

```
public class MyCompositeModelValidator: ModelValidator
{
   public MyCompositeModelValidator(ModelMetadata metadata,
      ControllerContext controllerContext)
      : base(metadata, controllerContext)
   {}

   public override IEnumerable<ModelValidationResult> Validate(
      object container)
   {
      bool isPropertiesValid = true;
      foreach (ModelMetadata propertyMetadata in Metadata.Properties)
```

```
        {
            foreach (ModelValidator validator in
                propertyMetadata.GetValidators(this.ControllerContext))
            {
                IEnumerable<ModelValidationResult> results =
                    validator.Validate(this.Metadata.Model);
                if (results.Any())
                {
                    isPropertiesValid = false;
                }
                foreach (ModelValidationResult result in results)
                {
                    string key = (propertyMetadata.PropertyName ?? "") + "." +
                        (result.MemberName ?? "");
                    yield return new ModelValidationResult
                    {
                        MemberName = key,
                        Message = result.Message
                    };
                }
            }
        }

        if (isPropertiesValid)
        {
            foreach (ModelValidator validator in
                Metadata.GetValidators(this.ControllerContext))
            {
                IEnumerable<ModelValidationResult> results =
                    validator.Validate(Metadata.Model);
                foreach (ModelValidationResult result in results)
                {
                    yield return result;
                }
            }
        }
    }
}
```

重写的 Validate 方法具体采用这样的验证策略：它先获取描述被验证数据类型或属性成员的 ModelMetada 对象，然后遍历所有描述其属性成员的 ModelMetadata 对象。对于每一个针对属性成员的 ModelMetadata 对象，我们调用其 GetModelValidators 方法得到应用在该属性上的 ModelValidator 列表。我们接下来调用列表中每一个 ModelValidator 的 Validate 方法对属性值实施验证，并根据返回值创建相应的 ModelValidationResult 对象，该对象被添加到最终返回的 ModelValidationResult 集合中。

只有在所有属性通过验证的情况下，我们才根据当前 ModelMetadata 对象获取相应的

ModelValidator 列表对容器对象自身实施验证，验证的结果直接添加到最终返回的 ModelValidationResult 集合中。

在自定义的 MyDefaultModelBinder 中，我们定义了单独的方法来完成针对简单类型和复杂类型的 Model 绑定，但是只需要在进行针对复杂类型的 Model 绑定过程中利用 CompositeModelValidator 进行 Model 验证。CompositeModelValidator 能够完成针对容器对象所有属性及其自身的验证，虽然它不递归地去验证复杂类型属性的数据成员，但是由于 Model 绑定本身是一个递归的过程，所以从整个流程上看 Model 验证也是递归进行的。

针对复杂数据类型的 Model 验证实现在 MyDefaultModelBinder 的 BindComplexModel 方法中。如下面的代码片段所示，我们在目标容器对象生成之后会利用创建的 CompositeModelValidator 对象对它进行验证。对于返回的每一个 ModelValidationResult，我们将其错误消息添加到相应的 ModelState 之中。

```
public class MyDefaultModelBinder : IModelBinder
{
    private object BindComplexModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        //其他操作
        Type modelType = bindingContext.ModelType;
        object model = this.CreateModel(controllerContext, bindingContext,
            modelType);
        bindingContext.ModelMetadata.Model = model;
        ICustomTypeDescriptor modelTypeDescriptor =
            new AssociatedMetadataTypeTypeDescriptionProvider(modelType)
            .GetTypeDescriptor(modelType);
        PropertyDescriptorCollection propertyDescriptors =
            modelTypeDescriptor.GetProperties();
        foreach (PropertyDescriptor propertyDescriptor in
            propertyDescriptors)
        {
            this.BindProperty(controllerContext, bindingContext,
                propertyDescriptor);
        }

        //Model 验证
        ModelMetadata metadata = ModelMetadataProviders.Current
           .GetMetadataForType(() => model, modelType);
        MyCompositeModelValidator validator =
            new MyCompositeModelValidator(metadata, controllerContext);
        foreach (ModelValidationResult result in validator.Validate(null))
        {
            string key = (bindingContext.ModelName ?? "") + "." +
                (result.MemberName ?? "");
```

```
            controllerContext.Controller.ViewData.ModelState
               .AddModelError(key.Trim('.'), result.Message);
         }

         return model;
      }
}
```

在第 7 章“Model 的绑定（下篇）”中已经验证过了自定义 MyDefaultModelBinder 的 Model 绑定功能，现在我们通过一个简单的实例来验证刚刚增加的 Model 验证功能。我们直接使用上面实例中定义的 Contact 类型，并且在 Contact 和 Address 类型和属性上应用自定义的 AlwaysFailsAttribute 特性。接下来我们定义了如下一个 HomeController，针对 GET 请求的 Action 方法 Index 直接将一个空 Contact 对象呈现在默认的 View 中，而针对 POST 请求的 Action 方法 Index 则将作为参数的 Contact 对象呈现在默认的 View 中。

```
public class HomeController : Controller
{
   public ActionResult Index()
   {
      return View(new Contact());
   }

   [HttpPost]
   public ActionResult Index(Contact contact)
   {
      return View(contact);
   }
}
```

如下所示的是 Action 方法 Index 对应 View 的定义，这是一个 Model 类型为 Contact 的强类型 View。我们在该 View 中将作为 Model 的 Contact 对象的所有属性（包括 Address 的所有属性）以编辑模式呈现在一个表单之中，该表单具有一个用于提交的“保存”按钮。

```
@model Contact
<html>
  <head>
      <title>Model 绑定中的验证</title>
   </head>
     <body>
      @using (Html.BeginForm())
      {
         @Html.EditorForModel()
         @Html.EditorFor(m=>m.Address)
         <input type="submit" value="保存" />
      }
```

```
    </body>
</html>
```

为了让ASP.NET MVC采用我们自定义的MyDefaultModelBinder来绑定Action方法Index中的参数，我们在Global.asax中按照如下方法将创建的MyDefaultModelBinder对象注册到Contact和Address类型上。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他成员
        ModelBinders.Binders.Add(typeof(Contact),
            new MyDefaultModelBinder());
        ModelBinders.Binders.Add(typeof(Address),
            new MyDefaultModelBinder());
    }
}
```

该程序运行之后会在浏览器中呈现一个“编辑联系人信息”的页面，直接单击“保存”按钮会呈现出如图8-7所示的输出结果，文本框右侧显示的文本正是应用在Contact和Address相应属性上的AlwaysFailsAttribute特性定义的错误消息。

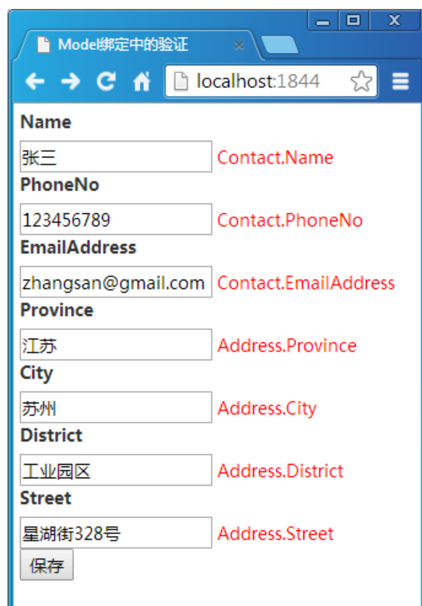

图8-7 实现在自定义MyDefaultModelBinder中的Model验证

8.3.3 针对“必需”数据成员的验证

通过上面的介绍我们知道，作为默认 ModelBinder 的 DefaultModelBinder 会在对复杂类型实施 Model 绑定之后会对绑定的数据对象实施验证。换句话说，在每次针对复杂类型的绑定迭代中，DefaultModelBinder 会先生成一个完整的数据对象，再利用创建的 CompositeModelValidator 对其实施验证。但是在这之前，DefaultModelBinder 会对必需数据成员实施验证。

我们知道 DefaultModelBinder 针对复杂类型的 Model 绑定会先从创建空对象开始，然后再针对其属性实施 Model 绑定并得到对应的属性值。在对属性进行赋值之前，DefaultModelBinder 会利用注册的 ModelProvider 根据描述属性的 ModelMetadata 创建相应的 ModelValidator 列表，并从中筛选出 IsRequired 属性为 True 的 ModelValidator 对属性数值实施验证。

对于自定义的 MyDefaultModelBinder 来说，我们将针对容器对象属性成员的 Model 绑定实现在 BindProperty 方法中，现在我们将针对必需数据成员的验证实现在该方法中。如下面的代码片段所示，我们将针对必需数据成员的验证定义在 ValidateRequiredPropertyValue 方法中，在对属性赋值之前调用其方法对绑定得到的属性值实施验证。

```
public class MyDefaultModelBinder : IModelBinder
{
    //其他成员
    private void BindProperty(ControllerContext controllerContext,
        ModelBindingContext bindingContext,
        PropertyDescriptor propertyDescriptor)
    {
        //将属性名附加到现有的前缀上
        string prefix = (bindingContext.ModelName ?? "") + "." +
            (propertyDescriptor.Name ?? "");
        prefix = prefix.Trim('.');

        //针对属性创建绑定上下文
        ModelMetadata metadata = bindingContext
            .PropertyMetadata[propertyDescriptor.Name];
        ModelBindingContext context = new ModelBindingContext
        {
            ModelName = prefix,
            ModelMetadata = metadata,
            ModelState = bindingContext.ModelState,
            ValueProvider = bindingContext.ValueProvider
        };

        //针对属性实施 Model 绑定并对属性赋值
        object propertyValue = ModelBinders.Binders
            .GetBinder(propertyDescriptor.PropertyType).BindModel(
             controllerContext, context);
        if (bindingContext.ModelMetadata.ConvertEmptyStringToNull &&
```

```
            object.Equals(propertyValue, string.Empty))
        {
            propertyValue = null;
        }
        context.ModelMetadata.Model = propertyValue;
        if (null == propertyValue)
        {
            this.ValidateRequiredPropertyValue(controllerContext,
                bindingContext, metadata, propertyValue);
        }
        propertyDescriptor.SetValue(bindingContext.Model, propertyValue);
    }

    private void ValidateRequiredPropertyValue(
        ControllerContext controllerContext,
        ModelBindingContext bindingContext,
        ModelMetadata propertyModelMetadata, object propertyValue)
    {
        string key = (bindingContext.ModelName ?? "") + "." +
            (propertyModelMetadata.PropertyName ?? "");
        key = key.Trim('.');
        ModelStateDictionary modelState = bindingContext.ModelState;

        ModelValidator validator = ModelValidatorProviders.Providers
           .GetValidators(propertyModelMetadata, controllerContext)
           .FirstOrDefault(v => v.IsRequired);
        if (null != validator)
        {
            foreach (ModelValidationResult result in
               validator.Validate(bindingContext.Model))
            {
                modelState.AddModelError(key, result.Message);
            }
        }
    }
}
```