# 第 9 章　Model 的验证（下篇）

在“上篇”中我们介绍了 3 种不同的自动化验证编程方式，ASP.NET MVC 会为之创建 3 种不同的 ModelValidator 对绑定的参数对象实施验证。对于这 3 种验证方式来说，最为常用的还是借助应用在数据类型或者属性成员上的 ValidationAttribute 特性来定义相应的验证规则。这种情况下对绑定参数实施验证的是一个 DataAnnotationsModelValidator 对象，它的提供者是 DataAnnotationsModelValidatorProvider，这两个对象是本章论述的核心。

# 9.1 ValidationAttribute 特性

应用在数据类型或者属性成员上的 ValidationAttribute 特性不仅仅用于定义相应的验证规则，其自身就具有验证的能力。一个 DataAnnotationsModelValidator 对象是对一个 ValidationAttribute 特性的封装，并直接利用它来对绑定的数据实施验证。所有的 ValidationAttribute 特性类型均继承自具有如下定义的抽象类型 ValidationAttribute，该类型定义在命名空间“System.ComponentModel.DataAnnotations”下。

```
public abstract class ValidationAttribute : Attribute
{
    public string     ErrorMessage { get; set; }
    public string     ErrorMessageResourceName { get; set; }
    public Type       ErrorMessageResourceType { get; set; }
    protected string  ErrorMessageString {get;}

    public virtual string FormatErrorMessage(string name);
    public virtual bool IsValid(object value);
    protected virtual ValidationResult IsValid(object value,
        ValidationContext validationContext)
    public void Validate(object value, string name);
    public ValidationResult GetValidationResult(object value,
        ValidationContext validationContext);
}
```

如上面的代码片段所示，ValidationAttribute 具有一个字符串类型的 ErrorMessage 属性用于指定错误消息。出于对本地化和对错误消息单独维护的需要，我们可以采用资源文件来存储错误消息，在这种情况下只需要通过 ErrorMessageResourceName 和 ErrorMessageResource-Type 这两个属性指定错误消息所在资源项的名称和类型即可。如果我们通过 ErrorMessage 属性指定一个字符串作为验证错误消息，同时通过 ErrorMessageResourceName 和 ErrorMessageResourceType 属性指定错误消息资源项对应的名称和类型，则后者具有更高的优先级。ValidationAttribute 具有一个受保护的只读属性 ErrorMessageString 用于返回最终的错误消息文本。

对于错误消息的定义，我们可以指定完整的消息内容，比如“年龄必须在 18 至 25 之间”。但是对于像资源文件这种对错误消息进行独立维护的情况，为了让定义的资源文本能够最大限度地被重用，我们倾向于定义一个包含占位符的文本模板，比如“{DisplayName}必须在{LowerBound}和{UpperBound}之间”，这样的消息适用于所有针对数值范围的验证。模板中的占位符可以在虚方法 FormatErrorMessage 中进行替换，该方法中的参数 name 实际上代表的是

被验证数据成员的显示名称，即 ModelMetadata 的 DisplayName 属性。

FormatErrorMessage 方法在 ValidationAttribute 中仅仅是按照如下方式调用 String 的静态方法 Format 并将参数 name 作为替换占位符的参数，所以在默认情况下定义错误消息模板只允许包含一个针对显示名称的占位符“{0}”。如果具有额外的占位符，或者需要采用非序号（“{0}”）占位符定义方式（比如采用类似于“{DisplayName}”这种基于文字的占位符更具可读性），我们需要重写 FormatErrorMessage 方法。

```
public abstract class ValidationAttribute : Attribute
{
    //其他成员
    public virtual string FormatErrorMessage(string name)
    {
        return string.Format(CultureInfo.CurrentCulture,
            ErrorMessageString, new object[] { name });
    }
}
```

## 9.1.1 数据是如何被验证的

当我们通过继承抽象类 ValidationAttribute 创建自己的验证特性的时候，可以将验证逻辑定义在任意一个重写的 IsValid 方法中。之所以能够通过重写任意一个 IsValid 方法来实现对目标数据的验证，原因在于定义在 ValidationAttribute 的这两个 IsValid 方法之间存在相互调用的关系。很显然，这种相互调用必然造成“死循环”，所以我们需要重写至少其中一个方法来避免“死循环”的发生。这里的“死循环”被加上引号，是因为 ValidationAttribute 在内部作了处理，当这种情况出现的时候会抛出一个 NotImplementedException 异常。

```
//调用公有 IsValid 方法
public class ValidatorAttribute : ValidationAttribute
{
    static void Main()
    {
        ValidatorAttribute validator = new ValidatorAttribute();
        validator.IsValid(new object());
    }
}

//调用受保护 IsValid 方法
public class ValidatorAttribute : ValidationAttribute
{
    static void Main()
    {
        ValidatorAttribute validator = new ValidatorAttribute();
```

```
        validator.IsValid(new object(),null);
    }
}
```

我们通过一个简单的实例来演示自定义 ValidationAttribute 对两个 IsValid 方法进行重写的必要性。我们在一个控制台应用中分别编写了如上两段程序，继承自 ValidationAttribute 自定义的 ValidatorAttribute 没有重写任何一个 IsValid 方法。当我们在 Debug 模式下分别运行这两段程序的时候，都会抛出如图 9-1 所示的 NotImplementedException 异常，提示“此类尚未实现 IsValid(object value)。首选入口点是 GetValidationResult()，并且类应重写 IsValid(object value, ValidationContext context)。”。

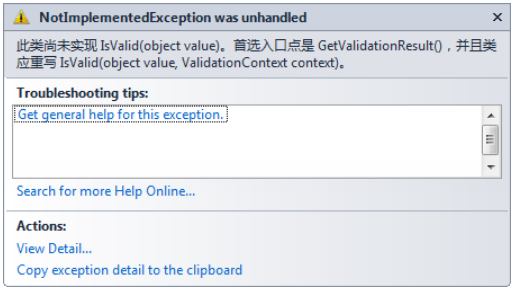



图 9-1　没有在自定义 ValidationAttribute 中重写 IsValid 方法导致的异常

受保护的 IsValid 方法中除了包含一个表示被验证对象的参数 value 之外，还有一个类型为 ValidationContext 的参数。顾名思义，ValidationContext 旨在为当前的验证维护相应的上下文。如下面的代码片段所示，一个 ValidationContext 对象携带的验证上下文信息包括通过 ObjectInstance 和 ObjectType 属性表示的验证对象及其类型，以及通过 MemberName 和 DisplayName 属性表示的成员名称（一般指属性名称）和显示名称。

```
public sealed class ValidationContext
{
    //其他成员
    public ValidationContext(object instance);
    public ValidationContext(object instance,
        IDictionary<object, object> items);

    public string     DisplayName { get; set; }
    public string     MemberName {  get;  set; }
    public object     ObjectInstance {  get; }
    public Type       ObjectType { get; }
}
```

受保护的 IsValid 方法返回值类型为具有如下定义的 ValidationResult。ValidationResult 与作为 ModelValidator 验证结果的 ModelValidationResult 类型具有类似的定义，它依然是错误消息

和成员名称的组合。不过 ModelValidationResult 对应某个单一的成员名称，而 ValidationResult 包含一组相关成员名称的列表。

```
public class ValidationResult
{
    //其他成员
    public ValidationResult(string errorMessage);
    public ValidationResult(string errorMessage,
        IEnumerable<string> memberNames);

    public string                 ErrorMessage { get; set; }
    public IEnumerable<string>    MemberNames { get; }
}
```

IsValid 方法在验证失败的情况下会返回一个具体的 ValidationResult 对象，如果指定的 ValidationContext 不为 Null，那么其 MemberName 属性表示的成员名称将会包含在这个返回的 ValidationResult 对象的 MemberNames 列表中。ValidationContext 对象的 DisplayName 属性将会作为调用 FormatErrorMessage 的参数，得到的格式化错误消息将会作为 ValidationResult 对象的 ErrorMessage 属性。如果成功通过验证，该方法会直接返回 Null。

我们可以通过调用 ValidationAttribute 的方法 GetValidationResult 对指定的数据对象实施验证，并得到以 ValidationResult 对象形式返回的验证结果，得到的 ValidationResult 对象实际上就是调用受保护 IsValid 方法的返回值。我们也可以调用 Validate 方法直接验证某个指定的对象，该方法在验证失败的情况下会直接抛出一个 ValidationException 异常，通过调用 FormatErrorMessage 方法（将参数 name 表示的字符串作为参数）格式化后的错误消息将会作为该异常的消息。

## 9.1.2 几个常用的 ValidationAttribute

在“System.ComponentModel.DataAnnotations”命名空间下定义了一系列具体的验证特性，它们大都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，所以在这里只是对它们作一个概括性的介绍。

- RequiredAttribute：用于验证必需数据成员。
- RangeAttribute：用于验证数据值是否在指定的范围之内。
- StringLengthAttribute：用于验证字符串数据的长度是否在指定的范围之内。
- MaxLengthAttribute/MinLengthAttribute：用于验证字符/数组数据长度是否小于/大于指定的

上/下限。

- RegularExpressionAttribute：用于验证字符串数据的格式是否与指定的正则表达式相匹配。
- CompareAttribute：用于验证数据值是否与另一个成员一致，在用户注册场景中可以用于确认两次输入密码的一致性。
- CustomValidationAttribute：指定一个用于验证目标成员的验证类型和验证方法。

除了上面这些常用的 ValidationAttribute 类型之外，还具有一个类型为 DataTypeAttribute 的验证特性帮助我们实现针对某种数据类型的验证，关于这个类型在本书第 4 章“Model 元数据的解析”中已经有过详细的介绍。DataTypeAttribute 还具有一系列子类，比如 UrlAttribute、CreditCardAttribute、EmailAddressAttribute、EnumDataTypeAttribute、File-Extensions-Attribute 和 PhoneAttribute 等，它们可以帮助我们实现针对某种具体数据类型（URL、信用卡号、电子邮箱地址、枚举成员、文件扩展名和电话号码）的验证。

在“System.Web.Security”命名空间下还具有一个类型为 MembershipPasswordAttribute 的验证特性。在进行用户注册时，我们可以利用它验证用户输入的密码是否符合预设的条件。我们可以利用 MinRequiredPasswordLength 和 MinRequiredNonAlphanumericCharacters 属性设置密码的最短长度和必须具有的非字母数字字符的个数，还可以通过设置属性 PasswordStrengthRegularExpression 指定一个正则表达式来控制密码的格式。它的额外 3 个属性（MinNonAlphanumericCharactersError、MinPasswordLengthError 和 PasswordStrengthError）分别代表相应的错误消息，如果我们通过其 ResourceType 指定了资源类型，那么这 3 个属性值将会作为相应资源项的名称。

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Field |
    AttributeTargets.Property, AllowMultiple=false)]
public class MembershipPasswordAttribute : ValidationAttribute
{
    public override string FormatErrorMessage(string name);
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext);

    public int    MinRequiredNonAlphanumericCharacters { get; set; }
    public int    MinRequiredPasswordLength { get; set; }
    public string PasswordStrengthRegularExpression { get; set; }

    public string MinNonAlphanumericCharactersError { get; set; }
    public string MinPasswordLengthError { get; set; }
    public string PasswordStrengthError { get; set; }

    public Type   ResourceType { get; set; }
}
```

如果我们没有对 MinRequiredPasswordLength、MinRequiredNonAlphanumericCharacters 和 PasswordStrengthRegularExpression 属性作显式设置，则它们的属性值来源于默认注册的 MembershipProvider。如下面的代码片段所示，MembershipProvider 类型具有对应的属性定义。

```
public abstract class MembershipProvider : ProviderBase
{
    //其他成员
    public abstract int MinRequiredNonAlphanumericCharacters { get; }
    public abstract int MinRequiredPasswordLength { get; }
    public abstract string PasswordStrengthRegularExpression { get; }
}
```

## 9.1.3 应用 ValidationAttribute 特性的唯一性

对于上面列出的这些预定义 ValidationAttribute，它们都具有一个相同的特征，那就是在同一个目标元素中只能应用一次，这可以通过应用在它们上面的 AttributeUsageAttribute 特性的定义看出来。以如下所示的 RequiredAttribute 特性为例，应用在该类型上的 AttributeUsageAttrribute 特性的 AllowMultiple 属性被设置为 False。

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Field |
    AttributeTargets.Property, AllowMultiple=false)]
public class RequiredAttribute : ValidationAttribute
{
    //省略成员
}
```

但这是否意味着如果我们在自定义 ValidationAttribute 的时候将 AttributeUsageAttribute 特性的 AllowMultiple 设置为 True，它们就可以被多次应用到同一个属性或者类型上了呢？我们不妨通过实例演示的方式来证实一下。

我们知道 RangeAttribute 特性可以帮助我们验证数据值的范围，但是有时候我们需要进行“条件性范围验证”。举个例子，我们现在对某个员工的薪水范围进行限定，但是不同级别员工的薪水范围是不同的，为此我们创建了一个名为 RangeIfAttribute 的验证特性帮助我们进行针对不同级别的薪水范围验证。如下面的代码片段所示，我们将 3 个 RangeIfAttribute 特性应用到了表示薪水的 Salary 属性上，分别针对 3 个级别（G7、G8 和 G9）的薪水范围作了相应的设定。

```
public class Employee
{
    public string Name { get; set; }
    public string Grade { get; set; }
```

```
    [RangeIf("Grade", "G7", 2000, 3000)]
    [RangeIf("Grade", "G8", 3000, 4000)]
    [RangeIf("Grade", "G9", 4000, 5000)]
    public decimal Salary { get; set; }
}
```

如下面的代码片段所示，RangeIfAttribute 直接继承自 RangeAttribute。RangeIfAttribute 根据被验证容器对象的另一个属性值来决定是否对当前属性实施验证，属性 Property 和 Value 就分别代表这个属性的名称和与之匹配的值。在重写的 IsValid 方法中，我们通过反射获取到了容器对象用于匹配的属性值，如果该值与 Value 属性值相匹配，则调用基类同名方法对指定对象进行验证，否则直接返回 ValidationResult.Success（Null）。应用在 RangeIfAttribute 上的 AttributeUsageAttribute 特性的 AllowMultiple 被设置为 True。

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = true)]
public class RangeIfAttribute: RangeAttribute
{
    public string Property { get; set; }
    public string Value { get; set; }

    public RangeIfAttribute(string property, string value, double minimum,
        double maximum) : base(minimum, maximum)
    {
        this.Property = property;
        this.Value = value??"";
    }

    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        PropertyInfo property =
            validationContext.ObjectType.GetProperty(this.Property);
        object propertyValue =
            property.GetValue(validationContext.ObjectInstance, null);
        propertyValue = propertyValue ?? "";
        if (propertyValue.ToString()!= this.Value)
        {
            return ValidationResult.Success;
        }
        return base.IsValid(value, validationContext);
    }
}
```

那么这样一个 RangeIfAttribute 特性真的能够按照我们期望的方式进行验证吗？为此我们在创建的空 ASP.NET MVC 应用中定义了如下一个 HomeController。我们在默认的 Action 方法 Index 中创建了用于描述 Employee 的 Salary 属性 ModelMetadata 对象，并通过调用其 GetValidators 方法得到用于验证该属性的 ModelValidator 列表，然后将这个 ModelValidator 列表

转化为数组作为 Model 呈现在对应的 View 中。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelMetadata employeeMetadata = ModelMetadataProviders.Current
            .GetMetadataForType(() => new Employee(), typeof(Employee));
        ModelMetadata salaryMetadata = employeeMetadata.Properties
            .FirstOrDefault(p => p.PropertyName == "Salary");
        IEnumerable<ModelValidator> validators = salaryMetadata
            .GetValidators(ControllerContext);
        return View(validators.ToArray());
    }
}
```

如下所示的是 Action 方法 Index 对应 View 的定义，这是一个 Model 类型为 ModelValidator 数组的强类型 View。我们在该 View 中将所有 ModelValidator 对象的类型名称通过表格的形式呈现出来。

```
@model ModelValidator[]
<html>
  <head>
     <title>ModelValidators</title>
  </head>
  <body>
     <table>
        @for(int i= 0; i<Model.Length; i++)
        {
           <tr><td>@(i+1)</td><td>@Model[i].GetType().Name</td></tr>
        }
     </table>
  </body>
</html>
```

该程序运行之后会在浏览器中呈现出如图 9-2 所示的输出结果。由于 Employee 的 Salary 属性类型为非空值类型，所以 ASP.NET MVC 会自动添加一个 RequiredAttributeAdapter 来进行必要性验证，另一个用于数值验证的 NumericModelValidator 也是源于 Salary 属性的类型。只有第一个 DataAnnotationsModelValidator 是针对应用在 Salary 属性上的 RangeIfAttribute 特性而创建的，换句话说，应用在同一属性上的 3 个 RangeIfAttribute 特性只有一个是有效的，具体原因何在呢？（S901）

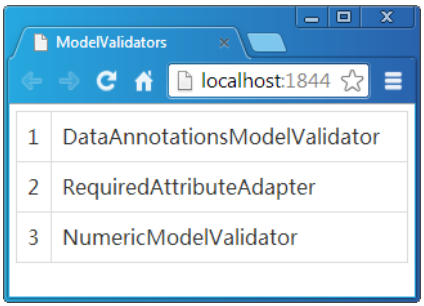



图 9-2　应用了多个同类验证特性的属性具有的 ModelValidator（默认）

我们知道 Attribute 具有一个 object 类型的 TypeId 属性，它默认返回代表自身类型的 Type 对象。ASP.NET MVC 在创建 DataAnnotationsModelValidator 对象的时候会根据该属性值对解析出来的所有特性进行分组，同一组只会选择一个特性。这就意味着对于多个应用到相同目标元素的同类 ValidationAttribute，有且只有一个是有效的。

那么如何来解决这个问题呢？其实很简单，既然 ASP.NET MVC 会根据 TypeId 属性对所有 ValidationAttribute 进行筛选，我们只需要通过重写 TypeId 属性使每个 ValidationAttribute 具有不同的属性值就可以了，为此我们按照如下方式在 RangeIfAttribute 中重写了 TypeId 属性。

```
[AttributeUsage( AttributeTargets.Field| AttributeTargets.Property,
    AllowMultiple = true)]
public class RangeIfAttribute: RangeAttribute
{
    //其他成员
    private object typeid;
    public override object TypeId
    {
        get{ return typeid?? (typeid= new object());}
    }
}
```

再次运行程序后将会在浏览器中得到如图 9-3 所示的输出结果，可以看到针对 3 个 RangeIfAttribute 特性的 3 个 DataAnnotationsModelValidator 被创建出来了。顺便说一下，通过重写 TypeId 属性使多个应用到相同元素上的同类 ValidationAttribute 生效的解决方案并不适合客户端验证，因为这会导致多组相同的验证规则被生成。（S902）

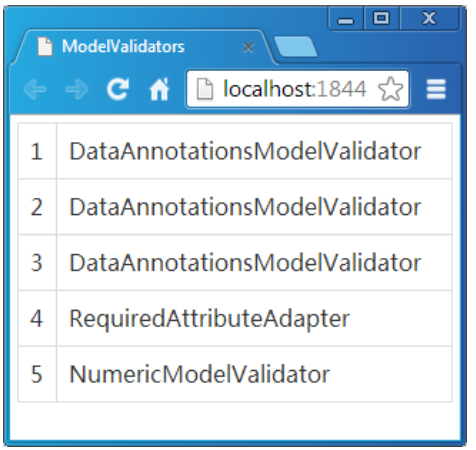



图 9-3 应用了多个同类验证特性的属性具有的 ModelValidator（重写 TypeId 属性）

## 9.2 DataAnnotationsModelValidator 及其提供策略

ModelValidator 是最终对绑定数据对象实施 Model 验证的对象，应用在数据类型或其属性成员上的 ValidationAttribute 特性最终会转换成相应的 ModelValidator 参与到针对目标数据的验证中。这个 ModelValidator 类型就是具有如下定义的 DataAnnotationsModelValidator，它的只读属性 Attribute 返回的就是对应的 ValidationAttribute 对象。

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

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

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

下面的代码片段给出了用于实施验证的核心方法 Validate 的完整定义。该方法首先针对被验证容器对象创建出作为验证上下文的 ValidationContext 对象，并采用 ModelMetadata 的 DisplayName 属性作为该 ValidationContext 的同名属性值。真正的验证工作通过调用 ValidationAttribute 的 GetValidationResult 方法来完成，如果该方法返回值不为 Null

（ValidationResult.Success），则将返回的 ValidationResult 转换成 ModelValidationResult 对象并添加到最终返回的 ModelValidationResult 集合中。

```
public class DataAnnotationsModelValidator : ModelValidator
{
    //其他成员
    public override IEnumerable<ModelValidationResult> Validate(
        object container)
    {
        ValidationContext validationContext = new ValidationContext(
            container ?? this.Metadata.Model, null, null)
        {
            DisplayName = this.Metadata.GetDisplayName()
        };
        ValidationResult validationResult =
            this.Attribute.GetValidationResult(
            this.Metadata.Model, validationContext);
        if (validationResult != ValidationResult.Success)
        {
            ModelValidationResult iteratorVariable2 =
                new ModelValidationResult
            {
                Message = validationResult.ErrorMessage
            };
            yield return iteratorVariable2;
        }
        else
        {
            yield break;
        }
    }
}
```

顺便再说说定义在 DataAnnotationsModelValidator 中的另外两个受保护只读属性的逻辑。用于返回错误消息的 ErrorMessage 属性来源于调用验证特性的 FormatErrorMessage 方法的返回值，指定的参数就是当前 ModelMetadata 对象的 DisplayName 属性值。由于只有 RequiredAttribute 特性才会对被验证数据实施必要性验证，所以只有被封装的 ValidationAttribute 为 RequiredAttribute 时，对应 DataAnnotationsModelValidator 的 IsRequired 属性才返回 True。

## 9.2.1 “适配”型 DataAnnotationsModelValidator

ValidationAttribute 特性本身具有数据验证功能，并且 DataAnnotationsModelValidator 直接利用它实现服务端验证。如果被封装的 ValidationAttribute 没有实现 IClientValidatable 接口（实际上定义在“System.ComponentModel.DataAnnotations”命名空间下的 ValidationAttribute 类型都没有实现这个接口），对应 DataAnnotationsModelValidator 就不能提供客户端验证的支持。

出于客户端验证的需要，ASP.NET MVC 为一些 ValidationAttribute 提供了“适配”型的 DataAnnotationsModelValidator。这些作为适配器的 DataAnnotationsModelValidator 具有一个共同的特征，那就是它们都通过重写的 GetClientValidationRules 方法返回相应的客户端验证规则。

## 1．DataTypeAttributeAdapter

顾名思义，DataTypeAttributeAdapter 是为了实现针对某种数据类型的客户端验证而为 DataTypeAttribute 定义的适配器。如下面的代码片段所示，DataTypeAttributeAdapter 直接继承自 DataAnnotationsModelValidator。它的 RuleName 属性表示客户端验证规则名称，可以在构造函数中通过参数 ruleName 指定，重写的 GetClientValidationRules 方法根据此名称和指定的验证错误消息提供相应的验证规则。

```
internal class DataTypeAttributeAdapter : DataAnnotationsModelValidator
{
    public DataTypeAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, DataTypeAttribute attribute,
        string ruleName);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();

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

## 2．DataAnnotationsModelValidator<TAttribute>

ASP.NET MVC 为针对其他 ValidationAttribute 的适配器定义了一个具有如下定义的泛型基类 DataAnnotationsModelValidator<TAttribute>。它是 DataAnnotationsModelValidator 的子类，泛型参数表示被封装的 ValidationAttribute 的类型。

```
public class DataAnnotationsModelValidator<TAttribute> :
    DataAnnotationsModelValidator where TAttribute: ValidationAttribute
{
    public DataAnnotationsModelValidator(ModelMetadata metadata,
        ModelBindingExecutionContext context, TAttribute attribute);
    protected TAttribute Attribute { get; }
}
```

ASP.NET MVC 为 4 个常用的验证特性（RequiredAttribute、RangeAttribute、RegularExpressionAttribute 和 StringLengthAttribute）定义了相应的适配器。如下面的代码片段所示，它们都是泛型 DataAnnotationsModelValidator<TAttribute>的子类，并且通过重写的 GetClientValidationRules 方法提供对应的客户端验证规则。

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

public class RangeAttributeAdapter :
    DataAnnotationsModelValidator<RangeAttribute>
{
    public RangeAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, RangeAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

public class RegularExpressionAttributeAdapter :
    DataAnnotationsModelValidator<RegularExpressionAttribute>
{
    public RegularExpressionAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, RegularExpressionAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

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

上面这 4 个具体的 DataAnnotationsModelValidator<TAttribute>类型均是公有类型，除此之外，ASP.NET MVC 还为其他 3 种 ValidationAttribute 类型（CompareAttribute、FileExtensionsAttribute 和 MembershipPasswordAttribute）定义了对应的适配器，它们的类型分别是 CompareAttributeAdapter、FileExtensionsAttributeAdapter 和 MembershipPassword-AttributeAdapter。如下面的代码片段所示，它们均是内部类型。

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

```
}

internal class FileExtensionsAttributeAdapter :
    DataAnnotationsModelValidator<FileExtensionsAttribute>
{
    public FileExtensionsAttributeAdapter(ModelMetadata metadata,
       ControllerContext context, FileExtensionsAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

internal class MembershipPasswordAttributeAdapter :
    DataAnnotationsModelValidator<MembershipPasswordAttribute>
{
    public MembershipPasswordAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, MembershipPasswordAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}
```

## 9.2.2 DataAnnotationsModelValidatorProvider

DataAnnotationsModelValidator 最终是通过对应的 DataAnnotationsModelValidatorProvider 创建的，后者是 AssociatedValidatorProvider 的子类。类型名称前缀“Associated”表示与数据类型“关联”的特性，所以 AssociatedValidatorProvider 实际上提供了一种根据应用在数据类型或者数据成员上的特性来创建 ModelValidator 的方式。

### 1．AssociatedValidatorProvider

如下面的代码片段所示，AssociatedValidatorProvider 是一个继承自 ModelValidatorProvider 的抽象类，它定义了一个抽象的受保护方法 GetValidators 根据提供的特性列表来获取验证目标数据对象或其属性成员的 ModelValidator 列表。除了作为参数的特性列表之外，该方法还具有额外两个参数，它们分别代表描述被验证数据类型或者属性成员的 ModelMetadata 和当前 ControllerContext。

```
public abstract class AssociatedValidatorProvider : ModelValidatorProvider
{
    //其他成员
    public sealed override IEnumerable<ModelValidator>
        GetValidators(ModelMetadata metadata, ControllerContext context);

    protected abstract IEnumerable<ModelValidator>
        GetValidators(ModelMetadata metadata, ControllerContext context,
```

```
        IEnumerable<Attribute> attributes);
}
```

AssociatedValidatorProvider 实现了定义在 ModelValidatorProvider 中的抽象方法 GetValidators（公有）。当该方法被执行的时候，如果提供的 ModelMetadata 对象是对一个根容器类型的描述，那么应用在数据类型上的特性会被提取出来。如果 ModelMetadata 对象是对容器类型某个属性成员的描述，那么应用在数据类型和属性上的特性均会被提取出来。这些通过反射方式提取出来的特性列表会作为参数调用受保护的 GetValidators 方法，方法最终返回的是一组 ModelValidator 列表。

## 2．DataAnnotationsModelValidatorProvider 的 ModelValidator 提供策略

作为 AssociatedValidatorProvider 的子类，DataAnnotationsModelValidatorProvider 只需要在实现的抽象方法 GetValidators 中从提供的特性列表中筛选出继承自 ValidationAttribute 的特性并创建对应的 DataAnnotationsModelValidator 就可以了。接下来我们根据其类型定义来讨论一下实现在 DataAnnotationsModelValidatorProvider 中具体的 ModelValidator 提供机制。

首先来认识一下定义在 DataAnnotationsModelValidatorProvider 之中的如下两组静态字段，第一组适用于根据 ValidationAttribute 特性来创建 ModelValidator，另一组则适用于根据实现了 IValidatableObject 接口的数据类型来创建 ModelValidator。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    internal static Dictionary<Type, DataAnnotationsModelValidationFactory>
        AttributeFactories;
    internal static DataAnnotationsModelValidationFactory
        DefaultAttributeFactory;

    internal static DataAnnotationsValidatableObjectAdapterFactory
        DefaultValidatableFactory;
    internal static
        Dictionary<Type, DataAnnotationsValidatableObjectAdapterFactory>
        ValidatableFactories;
}

public delegate ModelValidator DataAnnotationsModelValidationFactory(
    ModelMetadata metadata, ControllerContext context,
    ValidationAttribute attribute);
public delegate ModelValidator
    DataAnnotationsValidatableObjectAdapterFactory(ModelMetadata metadata,
    ControllerContext context);
```

具体来说，字典类型的字段 AttributeFactories 和 ValidatableFactories 分别维护着一组

ValidationAttribute 类型或者实现了 IValidatableObject 接口的数据类型与创建 ModelValidator 的委托对象之间的映射关系。如果给定的 ValidtionAttribute 类型或者数据类型没有在这个映射关系中，通过字段 DefaultAttributeFactory 和 DefaultValidatableFactory 表示的委托对象将用来创建对应的 ModelValidator。

当 DataAnnotationsModelValidatorProvider 类型被加载的时候，上述这 4 个静态字段会被初始化。如下的代码片段反映了采用的默认 ModelValidator 提供策略：如果被验证的数据类型或者属性成员上应用了 ValidationAttribute 特性，则 DataAnnotationsModelValidatorProvider 会根据此特性创建一个 DataAnnotationsModelValidator 对象。如果被验证的数据类型实现了 IValidatableObject 接口，则默认创建的 ModelValidator 是一个 ValidatableObjectAdapter 对象。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    static DataAnnotationsModelValidatorProvider()
    {
        //其他操作
        DefaultAttributeFactory = (metadata, context, attribute) =>
            new DataAnnotationsModelValidator(metadata, context, attribute);
        DefaultValidatableFactory = (metadata, context) =>
            new ValidatableObjectAdapter(metadata, context);
        AttributeFactories = BuildAttributeFactoriesDictionary();
        ValidatableFactories = new Dictionary<Type,
            DataAnnotationsValidatableObjectAdapterFactory>();
    }
}
```

DataAnnotationsModelValidatorProvider 的静态字段 AttributeFactories 体现了它针对某些特殊 ValidationAttribute 类型采用的 ModelValidator 提供策略。在静态构造函数被执行的时候，BuildAttributeFactoriesDictionary 方法会被调用来初始化该字段。具体来说，它会为如表 9-1 所示的 11 种类型的 ValidationAttribute 注册相应的创建 ModelValidator 的委托对象，该表还列出了委托对象最终创建的 ModelValidator 类型。

表 9-1　11 种 ValidationAttribute 和对应的适配型 ModelValidator

| ValidationAttribute | ModelValidator |
|---|---|
| RangeAttribute | RangeAttributeAdapter |
| RegularExpressionAttribute | RegularExpressionAttributeAdapter |
| RequiredAttribute | RequiredAttributeAdapter |
| StringLengthAttribute | StringLengthAttributeAdapter |
| MembershipPasswordAttribute | MembershipPasswordAttributeAdapter |
| CompareAttribute | CompareAttributeAdapter |
| FileExtensionsAttribute | FileExtensionsAttributeAdapter |
| CreditCardAttribute | DataTypeAttributeAdapter（RuleName="creditcard"） |
| EmailAddressAttribute | DataTypeAttributeAdapter（RuleName="email"） |
| PhoneAttribute | DataTypeAttributeAdapter（RuleName="phone"） |
| UrlAttribute | DataTypeAttributeAdapter（RuleName="url"） |

上面介绍的这 4 个静态字段体现了 DataAnnotationsModelValidatorProvider 采用的 ModelValidator 提供策略，即针对具体某种类型的 ValidationAttribute 或者实现了 IValidatableObject 接口的数据类型应该创建怎样的 ModelValidator。如果默认的提供策略不能满足我们的需求，还可以调用如下 8 个静态方法对其进行任意定制。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    public static void RegisterAdapter(Type attributeType, Type adapterType);
    public static void RegisterAdapterFactory(Type attributeType,
        DataAnnotationsModelValidationFactory factory);
    public static void RegisterDefaultAdapter(Type adapterType);
    public static void RegisterDefaultAdapterFactory(
        DataAnnotationsModelValidationFactory factory);

    public static void RegisterValidatableObjectAdapter(Type modelType,
        Type adapterType);
    public static void RegisterValidatableObjectAdapterFactory(Type modelType
        ,DataAnnotationsValidatableObjectAdapterFactory factory);
    public static void RegisterDefaultValidatableObjectAdapter(
        Type adapterType);
    public static void RegisterDefaultValidatableObjectAdapterFactory(
        DataAnnotationsValidatableObjectAdapterFactory factory);
}
```

接下来我们来具体地谈谈 DataAnnotationsModelValidatorProvider 在实现的 GetValidators 方法中是如何根据提供的 ModelMetadata 和特性列表来提供 ModelValidator 的。当该方法被执行的时候，所有 ValidationAttribute 特性会被从提供的特性列表中提取出来。针对每个具体的 ValidationAttribute 特性，如果能够在 AttributeFactories 字段中找到对应的委托对象，那么该委托对象会被用来创建相应的 ModelValidator，否则用于创建 ModelValidator 的是通过静态字段 DefaultAttributeFactory 表示的委托对象。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes);
}
```

如果被验证的数据类型实现了 IValidatableObject 接口，并且能够从静态字段 ValidatableFactories 找到对应的委托对象，那么该委托对象会被用来创建对应的 ModelValidator，否则对应的 ModelValidator 将会由 DefaultValidatableFactory 字段表示的委托来创建。

如果绑定的目标数据为值类型，在请求未曾提供相应数据的情况下，默认值会作为它们最终的绑定值。但是这样的绑定行为带来一个问题，即无法区分具有默认值的绑定数据是否真正由当前请求提供。在大部分情况下，不论值类型的属性上是否应用了 RequiredAttribute 特性，ASP.NET MVC 都将其视为一个“必需”的数据成员，并会为之创建一个 RequiredAttributeAdapter 对象作为对应的 ModelValidator。

如下面的代码片段所示，DataAnnotationsModelValidatorProvider 具有一个布尔类型的属性 AddImplicitRequiredAttributeForValueTypes，表示是否需要为值类型数据成员显式添加一个 RequiredAttribute 特性。该属性默认返回 True，意味着值类型的数据成员会默认视为必需的。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    public static bool AddImplicitRequiredAttributeForValueTypes { get; set; }
}
```

### 3．实例演示：解析针对不同属性成员创建的 ModelValidator（S903）

上面我们对实现在 DataAnnotationsModelValidatorProvider 中的 ModelValidator 的提供策略

进行了详细介绍，为了让读者对此有更加深刻的认识，我们来作一个简单的实例演示。我们在一个 ASP.NET MVC 应用中定义了如下一个数据类型 DemoModel。

```
public class DemoModel
{
    //第 1 组
    [Foobar]
    public object Foobar { get; set; }

    //第 2 组
    [Range(1,10)]
    public object Range { get; set; }

    [RegularExpression("...")]
    public object RegularExpression { get; set; }

    [Required]
    public object Required { get; set; }

    [StringLength(10)]
    public object StringLength { get; set; }

    [MembershipPassword]
    public object MembershipPassword { get; set; }

    [Compare("Foobar")]
    public object CompareAttribute { get; set; }

    [FileExtensions(Extensions = ".xml,.doc")]
    public object FileExtensions { get; set; }

    [CreditCard]
    public object CreditCard { get; set; }

    [EmailAddress]
    public object EmailAddress { get; set; }

    [Phone]
    public object Phone { get; set; }

    [Url]
    public object Url { get; set; }

    //第 3 组
    public ValidatableObject ValidatableObject { get; set; }

    //第 4 组
    public int ValueType { get; set; }
```

```
}

public class ValidatableObject : IValidatableObject
{
    public IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext)
    {
        throw new NotImplementedException();
    }
}

public class FoobarAttribute : ValidationAttribute{}
```

我们将定义在 DemoModel 中的若干属性成员划分为 4 组。包含在第一组的属性 Foobar 上应用了一个自定义的 FoobarAttribute 特性，第二组的属性上分别应用了表 9-1 中列出的 11 种类型的 ValidationAttribute，隶属于第三组的属性 ValidatableObject 的返回类型实现了 IValidatableObject 接口，划分为第四组的属性 ValueType 返回一个整数。

我们定义了如下一个 HomeController。在默认的 Action 方法 Index 中，我们解析出描述 DemoModel 所有属性的 ModelMetadata，并利用 DataAnnotationsModelValidatorProvider 根据它们的搭配验证对应属性成员采用的 ModelValidator 列表。我们将属性的名称和采用的 ModelValidator 列表组成一个 Dictionary<string, ModelValidator[]>对象，并作为 Model 呈现在对应的 View 中。

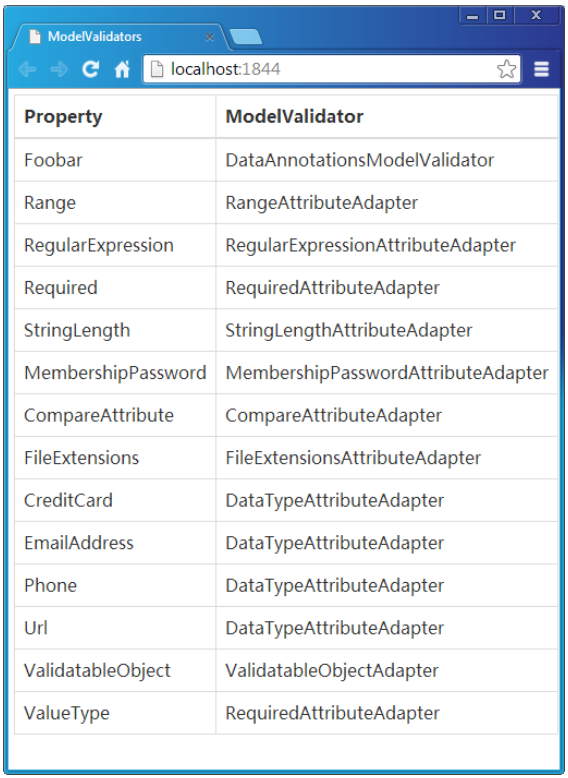
```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        Dictionary<string, ModelValidator[]> modelValidators =
            new Dictionary<string, ModelValidator[]>();
        DataAnnotationsModelValidatorProvider validatorProvider =
            new  DataAnnotationsModelValidatorProvider();
        foreach (ModelMetadata metadata in ModelMetadataProviders.Current
            .GetMetadataForProperties(new DemoModel(), typeof(DemoModel)))
        {
            modelValidators.Add(metadata.PropertyName,
                validatorProvider.GetValidators(metadata,
                this.ControllerContext).ToArray());
        }
        return View(modelValidators);
    }
}
```

如下所示的是 Action 方法 Index 对应 View 的定义，这是一个 Model 类型为 IDictionary<string, ModelValidator[]>的强类型 View。我们在该 View 中将属性的名称和采用的 ModelValidator 的类

型以表格的形式呈现出来。

```
@model IDictionary<string, ModelValidator[]>
<html>
    <head>
        <title>ModelValidators</title>
    </head>
    <body>
        <table>
            <thead>
                <tr><th>Property</th><th>ModelValidator</th></tr>
            </thead>
            <tbody>
                @foreach(var item in Model)
                {
                    <tr>
                        <td rowspan="@item.Value.Length">@item.Key</td>
                         @foreach (ModelValidator validator in item.Value)
                         {
                            <td>@validator.GetType().Name</td>
                         }
                    </tr>
                }
            </tbody>
        </table>
    </body>
</html>
```

运行该程序之后会在浏览器中得到如图 9-4 所示的输出结果，可以看到验证定义在 DemoModel 类型中各属性所采用的 ModelValidator 与 DataAnnotationsModelValidatorProvider 采用的 ModelValidator 提供策略是完全一致的。




| Property | ModelValidator |
|---|---|
| Foobar | DataAnnotationsModelValidator |
| Range | RangeAttributeAdapter |
| RegularExpression | RegularExpressionAttributeAdapter |
| Required | RequiredAttributeAdapter |
| StringLength | StringLengthAttributeAdapter |
| MembershipPassword | MembershipPasswordAttributeAdapter |
| CompareAttribute | CompareAttributeAdapter |
| FileExtensions | FileExtensionsAttributeAdapter |
| CreditCard | DataTypeAttributeAdapter |
| EmailAddress | DataTypeAttributeAdapter |
| Phone | DataTypeAttributeAdapter |
| Url | DataTypeAttributeAdapter |
| ValidatableObject | ValidatableObjectAdapter |
| ValueType | RequiredAttributeAdapter |

图 9-4 验证 DemoModel 各个属性所采用的 ModelValidator

## 9.2.3 将 ValidationAttribute 特性应用到参数上

如果足够细心应该会发现我们常用的 ValidationAttribute 特性都可以直接应用到方法的参数上。以如下定义的 RangeAttribute 为例，应用在该类型上的 AttributeUsageAttribute 表明该特性可以直接标注到方法的参数和数据类型的字段及属性成员上。

```
[AttributeUsage(
    AttributeTargets.Parameter |
    AttributeTargets.Field |
    AttributeTargets.Property,
    AllowMultiple=false)]
public class RangeAttribute : ValidationAttribute
{
   //省略成员
}
```

但是对于ASP.NET MVC的Model验证来说，应用在Action方法参数上的ValidationAttribute特性起不到任何作用，因为用于进行Model验证的ModelValidator对象是通过描述参数类型的ModelMetadata创建的，它根本不会去解析应用在参数本身上的ValidationAttribute特性。

但在笔者看来直接针对Action方法参数来定义验证规则具有很高的实用价值。一方面，目前的Model验证仅限于针对容器对象本身及其属性的验证，如果目标Action方法的参数为String、Int32和Double等简单类型，则针对它们的验证只能通过手工的方式来完成。另一方面，具有相同参数类型的多个Action方法中可能具有不同的验证规则。如果我们能够将ValidationAttribute特性应用在参数上进行针对性的验证规则定义，这两个问题都将迎刃而解。

到目前为止，我们对ASP.NET MVC的Model验证系统已经有了一个全面的了解，现在我们试着通过对应的扩展使直接应用到参数上的ValidationAttribute特性能够生效。我们需要自定义一个ModelValidatorProvider来解析应用到参数上的验证特性，并据此生成对应的ModelValidator。但在这之前需要解决的另一个问题是如何将应用于参数的特性提供给我们自定义的ModelValidatorProvider，在这里我们将当前的ControllerContext作为载体。

Action方法的执行是通过ActionInvoker来实现的，默认的ControllerActionInvoker和AsyncControllerActionInvoker都定义了一个受保护的虚方法GetParameterValue，它会根据描述参数的ParameterDescriptor对象和当前的ControllerContext实施Model绑定得到对应的参数值。我们可以通过继承ControllerActionInvoker/AsyncControllerActionInvoker以重写该方法的方式将ParameterDescriptor保存到当前的ControllerContext中。

为此我们自定义了如下两个ActionInvoker类型，其中ParameterValidationActionInvoker继承自ControllerActionInvoker，而ParameterValidationAsyncActionInvoker是AsyncControllerActionInvoker的子类。在重写的GetParameterValue方法中，我们在调用基类的同名方法之前将描述参数的ParameterDescriptor对象保存到当前ControllerContext中，具体来说是放到了表示当前路由数据的RouteDataDictionary对象的DataTokens集合中。在方法调用之后我们将它从ControllerContext中移除。

```
public class ParameterValidationActionInvoker : ControllerActionInvoker
{
    protected override object GetParameterValue(
        ControllerContext controllerContext,
        ParameterDescriptor parameterDescriptor)
    {
        try
        {
            controllerContext.RouteData.DataTokens.Add(
                typeof(ParameterDescriptor).FullName,parameterDescriptor);
            return base.GetParameterValue(controllerContext,
```

```
                parameterDescriptor);
        }
        finally
        {
            controllerContext.RouteData.DataTokens.Remove(
                  typeof(ParameterDescriptor).FullName);
        }
    }
}

public class ParameterValidationAsyncActionInvoker :
    AsyncControllerActionInvoker
{
    protected override object GetParameterValue(ControllerContext
        controllerContext, ParameterDescriptor parameterDescriptor)
    {
        try
        {
            controllerContext.RouteData.DataTokens.Add(
                typeof(ParameterDescriptor).FullName,parameterDescriptor);
            return base.GetParameterValue(controllerContext,
                parameterDescriptor);
        }
        finally
        {
            controllerContext.RouteData.DataTokens.Remove(
                  typeof(ParameterDescriptor).FullName);
        }
    }
}
```

被 ParameterValidationActionInvoker 和 ParameterValidationAsyncActionInvoker 存放到当前 ControllerContext 中的 ParameterDescriptor 被自定义的 ModelValidatorProvider 提取出来用于创建相应的 ModelValidator。如下面的代码所示，自定义的 ParameterValidationModelValidatorProvider 直接继承自 DataAnnotationsModelValidatorProvider，我们在重写的 GetValidators 方法中将 ParameterDescriptor 从 ControllerContext 中提取出来，并利用它得到应用在参数上的所有特性并与当前的特性列表进行合并，最后将合并的特性列表作为参数调用基类的 GetValidators 方法。

```
public class ParameterValidationModelValidatorProvider :
    DataAnnotationsModelValidatorProvider
{
    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes)
    {
        object descriptor;
        if (metadata.ContainerType == null && context.RouteData.DataTokens
            .TryGetValue(typeof(ParameterDescriptor).FullName,
            out descriptor))
        {
```

```
            ParameterDescriptor parameterDescriptor =
                (ParameterDescriptor)descriptor;
            DisplayAttribute displayAttribute = parameterDescriptor
                .GetCustomAttributes(true).OfType<DisplayAttribute>()
                .FirstOrDefault()
                ?? new DisplayAttribute
                { Name = parameterDescriptor.ParameterName };
            metadata.DisplayName = displayAttribute.Name;
            var addedAttributes =
                parameterDescriptor.GetCustomAttributes(true)
                .OfType<Attribute>();
            return base.GetValidators(metadata, context,
                attributes.Union(addedAttributes));
        }
        else
        {
            return base.GetValidators(metadata, context, attributes);
        }
    }
}
```

值得一提的是，应用在参数上的特性是针对最外层的容器类型，而不是针对容器类型的属性。比如我们在类型为 Contact 的参数上应用一个 ValidationAttribute 特性，该特性应该与应用在 Contact 类型上的特性具有相同的效果，但是与 Address 属性无关。所以 ParameterDescriptor 的提取及特性的合并仅仅在当前 ModelMetadata 的 ContainerType 属性为 Null 的情况下才会进行。除此之外，我们还利用标注在参数上的 DisplayAttribute 特性对 ModelMetadata 的 DisplayName 属性作了相应的设置。

在默认情况下只有在针对复杂类型的 Model 绑定过程中才会进行 Model 验证。虽然我们通过 ParameterValidationModelValidatorProvider 能够根据应用在 Action 方法参数上的验证特性生成相应的 ModelValidator，但是如果 ValidationAttribute 特性是应用在一个简单类型的参数上，对应的 ModelValidator 也是不会真正用于参数验证的。为了使 Model 验证发生在针对简单类型的 Model 绑定过程中，我们不得不创建一个自定义的 ModelBinder。

我们定义了一个具有如下定义的 ParameterValidationModelBinder，它直接继承自默认使用的 DefaultModelBinder，针对简单类型的 Model 验证定义在重写的 BindModel 方法中。

```
public class ParameterValidationModelBinder : DefaultModelBinder
{
    public override object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        object model = bindingContext.ModelMetadata.Model =
            base.BindModel(controllerContext, bindingContext);
        ModelMetadata metadata = bindingContext.ModelMetadata;
        if (metadata.IsComplexType || null == model)
        {
```

```
            return model;
        }

        //针对简单类型的 Model 验证
        Dictionary<string, bool> dictionary =
            new Dictionary<string, bool>(StringComparer.OrdinalIgnoreCase);
        foreach (ModelValidationResult result in ModelValidator
            .GetModelValidator(metadata, controllerContext).Validate(null))
        {
            string key = bindingContext.ModelName;
            if (!dictionary.ContainsKey(key))
            {
                dictionary[key] = bindingContext.ModelState.IsValidField(key);
            }
            if (dictionary[key])
            {
                bindingContext.ModelState.AddModelError(key, result.Message);
            }
        }
        return model;
    }
}
```

到此为止，为了能够将验证特性直接应用于 Action 方法的参数，我们创建了自定义的 ActionInvoker、ModelValidatorProvider 和 ModelBinder。为了验证它们是否能够最终实现期望的验证效果，我们将它们应用到一个简单的 ASP.NET MVC 应用中。我们在一个 ASP.NET MVC 应用中创建了一个具有如下定义的 HomeController，在重写的 CreateActionInvoker 方法中，如果调用基类同名方法返回的 ActionInvoker 类型为 ControllerActionInvoker，那么该方法将返回一个 ParameterValidationActionInvoker 对象，否则返回一个 ParameterValidation-AsyncActionInvoker 对象，这样做的目的是与默认的同步/异步 Action 执行方式保持一致。

```
public class HomeController : Controller
{
    protected override IActionInvoker CreateActionInvoker()
    {
        IActionInvoker actionInvoker = base.CreateActionInvoker();
        if (actionInvoker is ControllerActionInvoker)
        {
            return new ParameterValidationActionInvoker();
        }
        else
        {
            return new ParameterValidationAsyncActionInvoker();
        }
    }

    public ActionResult Add(
        [Display(Name = "第 1 个操作数")]
        [Range(10, 20, ErrorMessage = "{0}必须在{1}和{2}之间！")]
```

```
        [ModelBinder(typeof(ParameterValidationModelBinder))]
        double operand1,

        [Display(Name = "第 2 个操作数")]
        [Range(10, 20, ErrorMessage = "{0}必须在{1}和{2}之间！")]
        [ModelBinder(typeof(ParameterValidationModelBinder))]
        double operand2)
    {
        double result = 0.00;
        if (ModelState.IsValid)
        {
            result = operand1 + operand2;
        }
        return View(new OperationData
        {
            Operand1   = operand1,
            Operand2   = operand2,
            Operator   = "Add",
            Result     = result
        });
    }
}

public class OperationData
{
    [DisplayName("操作数 1")]
    public double Operand1 { get; set; }

    [DisplayName("操作数 2")]
    public double Operand2 { get; set; }

    [DisplayName("操作符")]
    public string Operator { get; set; }

    [DisplayName("运算结果")]
    public double Result { get; set; }
}
```

我们在 HomeController 中定义了一个进行加法运算的 Action 方法 Add，传入的参数代表两个操作数。我们在两个参数上应用了 3 个特性，其中 DisplayAttribute 特性用于设置显示名称，RangeAttribute 特性用于限制数值的范围（10～20），而 ModelBinderAttribute 特性则是为了让针对这两个参数的绑定采用我们自定义的 ParameterValidationModelBinder 来完成。我们在 Add 方法中会进行相应的运算，并将相关的信息封装到一个 OperationData 对象，此对象将作为 Model 呈现在对应的 View 中。

如下所示的是 Action 方法 Add 对应 View 的定义，这是一个 Model 类型为 OperationData 的强类型 View。我们在该 View 中直接调用 HtmlHelper<TModel>的扩展方法 EditorForModel 将作为 Model 的 OperationData 对象以编辑模式呈现出来。

```
@model OperationData
<html>
    <head>
        <title>将验证特性应用在参数上</title>
    </head>
    <body>
        @Html.EditorForModel()
    </body>
</html>
```

为了让访问 Action 方法 Add 的请求能够直接将操作数放到请求的 URL 中，我们在默认生成的 RouteConfig 类型中作了如下所示的路由注册。

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        //其他操作
        routes.MapRoute(
            name        : "Add",
            url         : "{action}/{operand1}/{operand2}",
            defaults    : new { controller = "Home"}
        );
    }
}
```

我们在 Global.asax 中通过下面的代码对自定义的 ParameterValidationModel-ValidatorProvider 进行了注册，在进行注册之前需要将现有的 ModelValidatorProvider 移除。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        DataAnnotationsModelValidatorProvider validatorProvider =
            ModelValidatorProviders.Providers
            .OfType<DataAnnotationsModelValidatorProvider>()
            .FirstOrDefault();
        if (null != validatorProvider)
        {
            ModelValidatorProviders.Providers.Remove(validatorProvider);
        }
        ModelValidatorProviders.Providers.Add(
            new ParameterValidationModelValidatorProvider());
    }
}
```

现在运行我们的程序，并在浏览器中指定相应的 URL 访问定义在 HomeController 中的 Action 方法 Add。如果提供不合法的操作数（比如“/Add/50/50”），验证消息将会以如图 9-5 所示的效果显示在相应的文本框旁边。（S904）

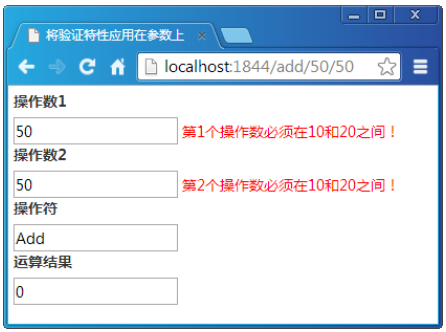



图 9-5　通过应用在 Action 方法参数的验证特性进行 Model 验证

## 9.2.4　一种 Model 类型，多种验证规则

理想的 Model 验证应该是场景驱动的，而不是类型驱动的，因为同一个数据类型在不同的使用场景中可能具有不同的验证规则。举个简单的例子，对于一个表示应聘者的数据对象来说，应聘的岗位不同，肯定对应聘者的年龄、性别、专业技能等方面具有不同的要求。

但是 ASP.NET MVC 的 Model 验证恰恰是类型驱动的，因为验证规则是通过应用在数据类型及其属性上的 ValidationAttribute 特性来定义的，这样的验证方式实际上限制了数据类型在基于不同验证规则的场景中被重用的可能性。上面的扩展将 ValidationAttribute 特性直接应用在参数上变成了可能，这在一定程度上解决了这个问题，但是也只能解决部分问题，因为应用到参数的 ValidationAttribute 特性只能用于针对参数类型级别的验证，而不能用于针对参数类型属性级别的验证。

现在我们利用对 ASP.NET MVC 的扩展来实现一种基于不同验证规则的 Model 验证。为了让读者对这种验证方式有一个直观的认识，我们先通过一个简单的实例来看看这个扩展最终实现了怎样的验证效果。我们在一个 ASP.NET MVC 应用中定义了如下一个 Person 类型。

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

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

    [DisplayName("年龄")]
    [RangeValidator(10, 20, RuleName="Rule1",
       ErrorMessage = "{0}必须在{1}和{2}之间！")]
```

```
    [RangeValidator(20, 30, RuleName = "Rule2",
       ErrorMessage = "{0}必须在{1}和{2}之间！")]
    [RangeValidator(30, 40, RuleName = "Rule3",
       ErrorMessage = "{0}必须在{1}和{2}之间！")]
    public int Age { get; set; }
}
```

我们在表示年龄的 Age 属性上应用了 3 个自定义的 RangeValidatorAttribute（不是 RangeAttribute）特性，它们对应着通过 RuleName 属性指定的 3 种不同的验证规则。3 种验证规则（Rule1、Rule2 和 Rule3）分别要求年龄在 10~20、20~30 和 30~40 岁之间。

然后我们定义了如下一个 HomeController，它具有 3 组 Action 方法（Index、Rule1 和 Rule2）。方法 Rule1、Rule2 和 HomeController 类上应用了一个自定义的 ValidationRuleAttribute 特性表明当前采用的验证规则。用于指定验证规则的 ValidationRuleAttribute 特性可以同时应用于 Controller 类型和 Action 方法上，后者具有更高的优先级。针对 HomeController 的定义，Action 方法 Index、Rule1 和 Rule2 分别采用的验证规则为 Rule3、Rule1 和 Rule2。

```
[ValidationRule("Rule3")]
public class HomeController : RuleBasedController
{
    public ActionResult Index()
    {
        return View("person", new Person());
    }
    [HttpPost]
    public ActionResult Index(Person person)
    {
        return View("person", person);
    }

    [ValidationRule("Rule1")]
    public ActionResult Rule1()
    {
        return View("person", new Person());
    }
    [HttpPost]
    [ValidationRule("Rule1")]
    public ActionResult Rule1(Person person)
    {
        return View("person", person);
    }

    [ValidationRule("Rule2")]
    public ActionResult Rule2()
    {
        return View("person", new Person());
    }
    [HttpPost]
    [ValidationRule("Rule2")]
```

```
    public ActionResult Rule2(Person person)
    {
        return View("person", person);
    }
}
```

定义在 HomeController 中的 6 个方法具有相同的操作，它们都将创建/接收的 Person 对象呈现到具有如下定义的 View 中。这是一个 Model 类型为 Person 的强类型 View，我们在该 View 中将作为 Model 的 Person 对象以编辑模式呈现在一个表单中，并在表单中提供一个提交按钮。

```
@model Person
<html>
    <head>
        <title>编辑个人信息</title>
        <style type="text/css">
            .field-validation-error{color: red;}
        </style>
    </head>
    <body>
        @using (Html.BeginForm())
        {
            @Html.EditorForModel()
            <input type="submit" value="保存" />
        }
    </body>
</html>
```

现在运行我们的程序并通过在浏览器中指定相应的地址分别访问定义在 HomeController 中的 3 个 Action（Index、Rule1 和 Rule2），一个用于编辑个人信息的表单会呈现出来。然后根据 3 个 Action 方法采用的验证规则输入不合法的年龄，单击“保存”按钮会看到输入的年龄按照对应的规则被验证，具体的验证效果如图 9-6 所示。（S905）

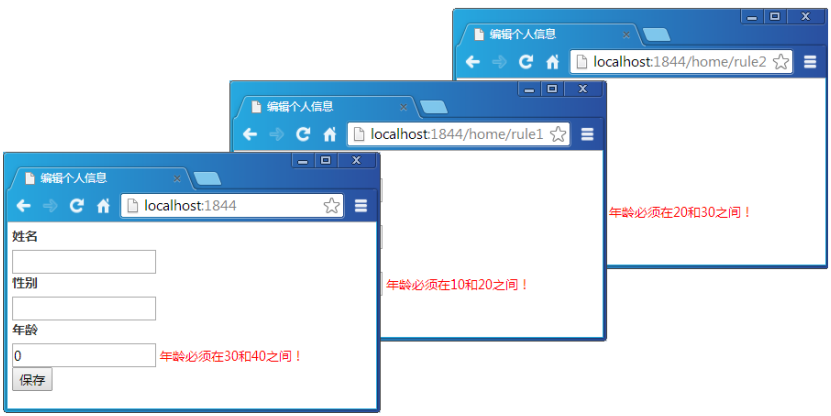



图 9-6 针对不同规则的 Model 验证

我们现在就来具体谈谈上面这个例子所展示的基于不同规则的 Model 验证是如何实现的。我们首先需要重建一套新的验证特性体系，因为需要指定具体的验证规则。我们定义了如下一个 ValidatorAttribute 类型，它是一个直接继承自 ValidationAttribute 的抽象类，其属性 RuleName 表示采用的验证规则名称。我们重写了 TypeId 属性，因为需要在相同的属性或者类型上应用多个同类的 ValidatorAttribute 特性。

```
[AttributeUsage( AttributeTargets.Class| AttributeTargets.Property,
    AllowMultiple = true)]
public abstract class ValidatorAttribute: ValidationAttribute
{
    private object typeId;
    public string RuleName { get; set; }
    public override object TypeId
    {
        get{return typeId ?? (typeId = new object());}
    }
}
```

上面演示实例中采用的 RangeValidatorAttribute 特性定义如下，可以看到它仅仅是对 RangeAttribute 的封装。RangeValidatorAttribute 具有与 RangeAttribute 一致的构造函数定义，并直接使用被封装的 RangeAttribute 实施验证。除了能够通过 RuleName 指定具体采用的验证规则之外，其他的使用方式与 RangeAttribute 完全一致。

```
[AttributeUsage( AttributeTargets.Property, AllowMultiple = true)]
public class RangeValidatorAttribute:ValidatorAttribute
{
    private RangeAttribute rangeAttribute;

    public RangeValidatorAttribute(int minimum, int maximum)
    {
        rangeAttribute = new RangeAttribute(minimum, maximum);
    }
    public RangeValidatorAttribute(double minimum, double maximum)
    {
        rangeAttribute = new RangeAttribute(minimum, maximum);
    }
    public RangeValidatorAttribute(Type type, string minimum, string maximum)
    {
        rangeAttribute = new RangeAttribute(type, minimum, maximum);
    }
    public override bool IsValid(object value)
    {
        return rangeAttribute.IsValid(value);
    }

    public override string FormatErrorMessage(string name)
    {
        return string.Format(CultureInfo.CurrentCulture,
            base.ErrorMessageString, new object[] {
```

```
            name, rangeAttribute.Minimum, rangeAttribute.Maximum });
    }
}
```

ValidatorAttribute 的 RuleName 属性仅仅指定了验证特性采用的验证规则名称，当前应该采用的验证规则通过应用在 Action 方法或者 Controller 类型上的 ValidationRuleAttribute 特性指定。如下所示的就是这个 ValidationRuleAttribute 的定义，验证规则名称通过 RuleName 属性表示。

```
[AttributeUsage( AttributeTargets.Class| AttributeTargets.Method)]
public class ValidationRuleAttribute: Attribute
{
    public string RuleName { get; private set; }
    public ValidationRuleAttribute(string ruleName)
    {
        this.RuleName = ruleName;
    }
}
```

对于这个用于实现针对不同验证规则的扩展来说，其核心是如何将通过 ValidationRuleAttribute 特性设置的验证规则应用到 ModelValidator 的提供机制中，从众多 ValidationAttribute 列表中筛选出与当前验证规则匹配的那一部分。在这里我们依然使用当前的 ControllerContext 来保存这个验证规则名称。细心的读者应该留意到了上面演示实例中创建的 HomeController 不是继承自 Controller，而是继承自 RuleBasedController，这个自定义的 Controller 基类定义如下。

```
public class RuleBasedController: Controller
{
    private static Dictionary<Type, ControllerDescriptor>
        controllerDescriptors = new Dictionary<Type, ControllerDescriptor>();

    public ControllerDescriptor ControllerDescriptor
    {
        get
        {
            ControllerDescriptor controllerDescriptor;
            if (controllerDescriptors.TryGetValue(this.GetType(),
                out controllerDescriptor))
            {
                return controllerDescriptor;
            }
            lock (controllerDescriptors)
            {
                if (!controllerDescriptors.TryGetValue(this.GetType(),
                    out controllerDescriptor))
                {
                    controllerDescriptor =
                        new ReflectedControllerDescriptor(this.GetType());
                    controllerDescriptors.Add(this.GetType(),
                        controllerDescriptor);
```

```
                }
                return controllerDescriptor;
            }
        }
    }

    protected override IAsyncResult BeginExecuteCore(AsyncCallback callback,
        object state)
    {
        SetValidationRule();
        return base.BeginExecuteCore(callback, state);
    }

    protected override void ExecuteCore()
    {
        SetValidationRule();
        base.ExecuteCore();
    }

    private void SetValidationRule()
    {
        string actionName = this.ControllerContext.RouteData
            .GetRequiredString("action");
        ActionDescriptor actionDescriptor = this.ControllerDescriptor
            .FindAction(this.ControllerContext, actionName);
        if (null != actionDescriptor)
        {
            ValidationRuleAttribute validationRuleAttribute =
                actionDescriptor.GetCustomAttributes(true)
                .OfType<ValidationRuleAttribute>().FirstOrDefault() ??
                this.ControllerDescriptor.GetCustomAttributes(true)
                    .OfType<ValidationRuleAttribute>().FirstOrDefault() ??
                new ValidationRuleAttribute(string.Empty);
             this.ControllerContext.RouteData.DataTokens.Add(
                 "ValidationRuleName", validationRuleAttribute.RuleName);
        }
    }
}
```

在继承自 Controller 的 RuleBasedController 中，ExecuteCore 和 BeginExecuteCore 方法被重写。在调用基类的同名方法之前，我们调用 SetValidationRule 方法提取应用在当前 Action 方法/Controller 类型上的 ValidationRuleAttribute 特性指定的验证规则名称，并将其保存到当前的 ControllerContext 中。由于针对 ValidationRuleAttribute 特性的解析需要用到用于描述 Controller 的 ControllerDescriptor 对象，出于性能考虑，我们对该对象进行了全局缓存。

对于应用在同一个属性或者类型上的多个基于不同验证规则的 ValidatorAttribute 特性，对应的验证规则名称并没有应用到具体的验证逻辑中。以上面定义的 RangeValidatorAttribute 为例，具体的验证逻辑通过被封装的 RangeAttribute 来实现，如果不作任何处理，基于不同规则的 RangeValidatorAttribute 都将参与到最终的 Model 验证过程中。我们必须要做的是：在根据

特性创建 ModelValidator 的时候，只选择那些与当前验证规则一致的 ValidatorAttribute 特性，这样的操作实现在具有如下定义的 RuleBasedValidatorProvider 中。

```
public class RuleBasedValidatorProvider :
    DataAnnotationsModelValidatorProvider
{
    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes)
    {
        object validationRuleName = string.Empty;
        context.RouteData.DataTokens.TryGetValue("ValidationRuleName",
            out validationRuleName);
        string ruleName = validationRuleName.ToString();
        attributes = this.FilterAttributes(attributes, ruleName);
        return base.GetValidators(metadata, context, attributes);
    }

    private IEnumerable<Attribute> FilterAttributes(
        IEnumerable<Attribute> attributes, string validationRule)
    {
            var validatorAttributes = attributes.OfType<ValidatorAttribute>();
            var nonValidatorAttributes =
                attributes.Except(validatorAttributes);
            List<ValidatorAttribute> validValidatorAttributes =
                new List<ValidatorAttribute>();

        if (string.IsNullOrEmpty(validationRule))
        {
            validValidatorAttributes.AddRange(validatorAttributes.Where(
                v => string.IsNullOrEmpty(v.RuleName)));
        }
        else
        {
            var groups = from validator in validatorAttributes
                         group validator by validator.GetType();
            foreach (var group in groups)
            {
                ValidatorAttribute validatorAttribute = group.Where(
                    v => string.Compare(v.RuleName, validationRule, true) ==
                    0).FirstOrDefault();
                if (null != validatorAttribute)
                {
                    validValidatorAttributes.Add(validatorAttribute);
                }
                else
                {
                    validatorAttribute = group.Where(
                        v => string.IsNullOrEmpty(v.RuleName))
                            .FirstOrDefault();
                    if (null != validatorAttribute)
                    {
```

```
                    validValidatorAttributes.Add(validatorAttribute);
                }
            }
        }
        return nonValidatorAttributes.Union(validValidatorAttributes);
    }
}
```

如上面的代码所示，RuleBasedValidatorProvider 继承自 DataAnnotationsModelValidatorProvider，基于当前验证规则（从当前的 ControllerContext 中提取）对 ValidatorAttribute 的筛选及最终对 ModelValidator 的创建实现在重写的 GetValidators 方法中，具体的筛选机制是：

- 如果当前的验证规则存在，则选择与之具有相同规则名称的第一个 ValidatorAttribute。
- 如果这样的 ValidatorAttribute 找不到，则选择第一个没有指定验证规则的 ValidatorAttribute。
- 如果当前的验证规则没有指定，那么同样选择第一个没有指定验证规则的 ValidatorAttribute。

我们需要在 Global.asax 中通过如下方式对自定义的 RuleBasedValidatorProvider 进行注册，然后我们的应用就能按照期望的方式根据指定的验证规则实施 Model 验证了。

```
public class MvcApplication : System.Web.HttpApplication
{
    //其他成员
    protected void Application_Start()
    {
        //其他操作
        DataAnnotationsModelValidatorProvider validator =
            ModelValidatorProviders.Providers
            .OfType<DataAnnotationsModelValidatorProvider>()
            .FirstOrDefault();
        if(null != validator)
        {
            ModelValidatorProviders.Providers.Remove(validator);
        }
        ModelValidatorProviders.Providers.Add(
            new RuleBasedValidatorProvider());
    }
}
```

## 9.3 客户端验证

之前我们一直讨论的 Model 验证仅限于服务端验证，即在 Web 服务器端根据相应的规则对请求提交的数据实施验证。如果我们能够在客户端（浏览器）对用户输入的数据先进行验证，

这样会减少针对服务器请求的频率，从而缓解 Web 服务器的访问压力。ASP.MVC 2.0 及其之前的版本采用 ASP.NET Ajax 进行客户端验证，在 ASP.NET MVC 3.0 中引入了 jQuery 验证框架。

## 9.3.1 jQuery 验证

Unobtrusive JavaScript 已经成为了 JavaScript 编程的一个指导方针，但是到目前为止貌似还没有一个针对它的确切定义，但是一些 Unobtrusive JavaScript 的基本原则却已经被广泛地接受。Unobtrusive JavaScript 体现了一种被称为“渐进式增强（Progressive Enhancement，PE）”的 Web 设计模式，它采用分层的方式实现了 Web 页面内容与功能的分离。用于实现某种功能的 JavaScript 不再内嵌于用于展现内容的 HTML 中，而是作为独立的层次建立在 HTML 之上。

我们就以验证为例，假设一个 Web 页面中具有如下一个表单，需要针对表单中 3 个文本框（foo、bar 和 baz）的输入进行验证。假设具体的验证操作实现在 validate 函数中，那么我们可以采用如下的 HTML 使相应的文本框在失去焦点的时候对输入的数据实施验证。

```
<form action="/">
    <input id="foo" type="text" onblur="validate()" />
    <input id="bar" type="text" onblur="validate()" />
    <input id="baz" type="text" onblur="validate()" />
    ...
</form>
```

但这不是一个好的设计，理想的方式是让 HTML 定义内容呈现的结构，让 CSS 控制内容呈现的样式，所有功能的实现定义在独立的 JavaScript 中，所以用于实现验证对 JavaScript 的调用不应该以内联的方式出现在 HTML 中。按照 Unobtrusive JavaScript 的编程方式，我们应该将以内联方式实现的事件注册（onblur="validate()"）替换成如下形式。

```
<form action="/">
  <input id="foo" type="text"/>
  <input id="bar" type="text"/>
  <input id="baz" type="text" />
</form>

<script type="text/javascript">
    window.onload = function () {
        document.getElementById("foo").onblur = validate;
        document.getElementById("bar").onblur = validate;
        document.getElementById("baz").onblur = validate;
    }
</script>
```

Unobtrusive JavaScript 是一个很宽泛的话题，我们不可能在本书中对它进行系统的介绍。

Unobtrusive JavaScript 在 jQuery 的验证中得到了很好的体现，接下来我们就简单地介绍一下基于 jQuery 验证的编程。

### 1．以内联的方式指定验证规则

jQuery 的验证实际上是对表单中的输入元素进行验证，它支持一种内联的编程方式，使我们可以直接将验证的规则定义在被验证表单元素的 class（表示 CSS 类型）属性中。考虑到有一些读者对 jQuery 的验证框架可能不太熟悉，为此我们来作一个简单的实例演示。我们在一个 ASP.NET MVC 应用中定义了如下一个 HomeController，在 Action 方法 Index 中将默认的 View 呈现出来。

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

我们将作为 Web 页面的整个 HTML 定义在 Action 方法 Index 对应的 View 中，如下所示的代码片段是该 View 的定义。

```
<html>
  <head>
      <script type="text/javascript"
              src="@Url.Content("~/Scripts/jquery-1.10.2.js")"></script>
      <script type="text/javascript"
              src="@Url.Content("~/Scripts/jquery.validate.js")"></script>
      <script type="text/javascript">
          $(document).ready(function () {
              $("form").validate();
          });
      </script>
      <title>编辑个人信息</title>
  </head>
  <body>
      <form action="/">
      <table>
          <tr>
              <td>姓名: </td>
              <td>
                  <input class="required"  id="name" name="name" type="text"/>
              </td>
          </tr>
          <tr>
              <td>出生日期: </td>
              <td>
```

```
                <input class="required date" id="birthDate"
                    name="birthDate" type="text"/>
            </td>
        </tr>
        <tr>
            <td>Blog 地址: </td>
            <td>
                <input class="required url" id="blogAddress"
                    name="blogAddress" type="text"/>
            </td>
        </tr>
        <tr>
            <td>Email 地址: </td>
            <td>
                <input class="required email"id="emailAddress"
                    name="emailAddress" type="text"/>
            </td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="保存"/>
            </td>
        </tr>
    </table>
    </form>
  </body>
</html>
```

我们需要将两个必要的.js 文件包含进来，一个是 jQuery 的核心文件 jquery-1.10.2.js，另一个是实现验证的 jquery.validate.js。整个 HTML 文件的主体部分是一个表单，我们可以通过其中的文本框输入一些个人信息（姓名、出生日期、Blog 地址和 E-mail 地址），最后单击“保存”按钮对输入的数据进行提交。

对于这 4 个文本框对应的<input>元素来说，其 class 属性在这里被用于进行验证规则的定义。其中 required 表示对应的数据是必需的，而 date、url 和 email 则对输入数据的格式进行验证以确保是一个合法的日期、URL 和 E-mail 地址。真正对输入实施验证体现在如下一段 JavaScript 调用中，在这里我们仅仅是调用<form>元素的 validate 方法而已。

```
<script type="text/javascript">
   $(document).ready(function () {
       $("form").validate();
   });
</script>
```

现在运行我们的程序，一个用于提交个人信息的页面会被呈现出来。当我们输入不合法的数据时（第一次验证发生在提交表单时，之后的验证会在被验证表单元素失去焦点时触发），对应的验证将会自动被触发，而预定义的错误消息将会显示在被验证表单元素的右侧。具体的显示效果如图 9-7 所示。（S906）

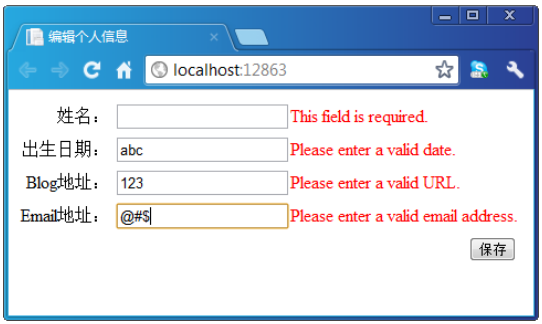



图 9-7 jQuery 验证及其默认错误消息的呈现

## 2．单独指定验证规则和错误消息

验证规则其实可以不用以内联的方式定义在被验证表单元素对应的 HTML 中，我们可以直接将它们定义在用于实施验证的 validate 方法中，该方法不仅可以指定表单被验证的输入元素对应的验证规则，还可以指定验证消息，以及控制其他验证行为。

现在我们将上面演示实例中 View 的 HTML 进行相应的修改，将包含在表单中的 4 个文本框通过 class 属性设置的验证规则移除，然后在调用表单的 validate 方法实施验证的时候按照如下方式手工地为被验证输入元素指定相应的验证规则和错误消息。验证规则和错误消息与验证元素之间是通过 name 属性（不是 id 属性）进行关联的。

```
<script type="text/javascript">
    $(document).ready(function () {
        $("form").validate({
            rules: {
                name         : { required: true },
                birthDate    : { required: true, date: true },
                blogAddress  : { required: true, url: true },
                emailAddress     : { required: true, email: true }
            },

            messages: {
                name         : { required: "请输入姓名" },
                birthDate    : { required: "请输入出生日期",
                                 date: "请输入一个合法的日期" },
                blogAddress  : { required : "请输入 Blog 地址",
                                 url: "请输入一个合法的 URL" },
                emailAddress: { required: "请输入 Email 地址",
                                 email: "请输入一个合法的 Email 地址"}
            }
        });
    });
```

```
</script>
```

再次运行程序后发现，定制的错误消息就会按照如图 9-8 所示的效果呈现出来。（S907）

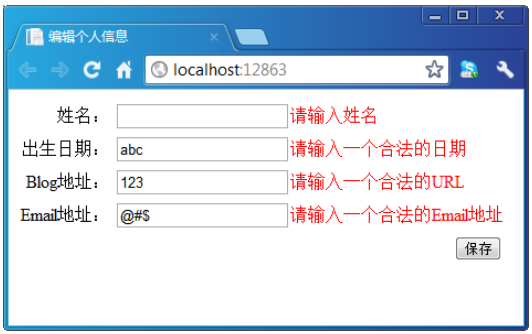



图 9-8　jQuery 验证及其定制错误消息的呈现

## 9.3.2　基于 jQuery 的 Model 验证

在简单了解了 Unobtrusive JavaScript 形式的验证在 jQuery 中的实现之后，我们来具体讨论 ASP.NET MVC 是如何利用它实现客户端验证的。服务端验证最终实现在相应的 ModelValidator 中，最终的验证规则定义在相应的 ValidationAttribute 中，而客户端验证规则通过 HtmlHelper<TModel>相应的模板方法（比如 TextBoxFor、EditorFor 和 EditorForModel 等）输出到生成的 HTML 中。服务端验证和客户端验证必须采用相同的验证规则，通过应用 ValidationAttribute 特性定义的验证规则也同样体现在基于客户端验证规则的 HTML 上。

### 1．ValidationAttribute 与 HTML

ASP.NET MVC 默认采用基于验证特性的声明式验证，服务端验证最终实现在两个重写的 IsValid 方法中。对于客户端验证，ASP.NET MVC 对 jQuery 的验证插件进行了扩展，它将验证规则以表单元素属性的方式输出到最终生成的 HTML 中。为了让客户端和服务端采用相同的验证规则，应用在 Model 类型某个属性上的 ValidationAttribute 特性最终会反映在被验证表单元素对应的 HTML 上。

```
public class Contact
{
    [DisplayName("姓名")]
    [Required(ErrorMessage ="请输入{0}!")]
    [StringLength(8, ErrorMessage="作为{0}字符串长度不能超过{1}!")]
```

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

    [DisplayName("电子邮箱地址")]
    [RegularExpression(@"^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$",
        ErrorMessage="请输入正确的电子邮箱地址！")]
    public string EmailAddress { get; set; }
}
```

假设我们具有如上一个数据类型 Contact，RequiredAttribute 和 StringLengthAttribute 特性应用到表示姓名的 Name 属性上用于确保用户必须输入一个不超过 8 个字符的字符串，而表示 E-mail 地址的 EmailAddress 属性应用了一个 RegularExpressionAttribute 用于确保用户输入一个合法的 E-mail 地址。在一个以此 Contact 为 Model 类型的 View 中，如果我们调用 HtmlHelper<TModel>的扩展方法 EditorForModel，最终会生成如下一段 HTML。

```
<div class="editor-label">
  <label for="Name">姓名</label>
</div>

<div class="editor-field">
  <input class="text-box single-line"
         data-val            ="true"
         data-val-length     ="作为姓名字符串长度不能超过 8!"
         data-val-length-max ="8"
         data-val-required   ="请输入姓名!"
         id="Name" name="Name" type="text" value="" />
  <span class="field-validation-valid" data-valmsg-for="Name"
        data-valmsg-replace="true"></span>
</div>

<div class="editor-label">
  <label for="EmailAddress">电子邮箱地址</label>
</div>

<div class="editor-field">
  <input class="text-box single-line"
         data-val                ="true"
         data-val-regex          ="请输入正确的电子邮箱地址！"
         data-val-regex-pattern  ="^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$"
         id="EmailAddress" name="EmailAddress" type="text" value="" />
  <span class="field-validation-valid" data-valmsg-for="EmailAddress"
         data-valmsg-replace="true"></span>
</div>
```

通过上面的这段 HTML 我们可以看到，Contact 的两个属性对应的<input>元素具有一个“data-val”属性和一系列以“data-val-”为前缀的属性，前者表示是否需要对用户输入的值进行验证，后者则代表相应的验证规则。具体来说，去除“data-val-”前缀后的属性名称就是 jQuery 验证规则名称。

一般来说，一个 ValidationAttribute 对应着一种验证类型和一系列可选的验证参数。比如 RequiredAttribute、StringLengthAttribute 和 RegularExpressionAttribute 对应的验证类型分别是“required”、“length”和“regex”，而 StringLengthAttribute 和 RegularExpressionAttribute 各自具有一个验证参数 length-max（表示允许的字符串最大长度）和 regex-pattern（正则表达式）。验证错误消息一般作为验证类型属性的值，而验证参数对应的属性值自然就是相应的参数值。

对于上面生成的 HTML 还有一点值得一提，对应着被验证属性的<input>元素会紧跟一个元素用于显示验证失败后的错误消息。该元素的 CSS 类型为“field-validation-valid”，当验证失败后被替换为“field-validation-error”，可以通过它来定制错误消息的显示样式。

## 2．客户端验证规则的生成

ASP.NET MVC 在利用 jQuery 进行客户端验证的时候，虽然验证规则并没有采用其原生的方式通过被验证元素的 class 属性来提供，但是却可以通过“data-val-{rulename}”的命名模式提取相应的验证规则属性值，并最终得到对应的验证规则，ASP.NET MVC 只需要对此作简单的“适配”即可。我们现在关心的是当调用定义在 HtmlHelper<TModel>中相应的模板方法将 Model 对象的某个属性以表单元素的形式进行呈现的时候，ASP.NET MVC 是如何生成这些以“data-val-”为前缀的验证属性的呢？

在这里我们需要涉及一个表示客户端验证规则的 ModelClientValidationRule 类型。如下面的代码所示，ModelClientValidationRule 具有 3 个属性，字符串属性 ErrorMessage 和 ValidationType 表示验证错误消息和验证的类型，类型为 IDictionary<string, object>的只读属性 ValidationParameters 表示辅助客户端验证的参数，其中 Key 和 Value 分别表示验证参数名和参数值。

```
public class ModelClientValidationRule
{
    public string ErrorMessage { get; set; }
    public string ValidationType { get; set; }
    public IDictionary<string, object> ValidationParameters { get; }
}

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

```
}
```

通过前面的介绍我们知道，抽象类 ModelValidator 中具有一个虚方法 GetClientValidationRules，用于创建一个 ModelClientValidationRule 对象的列表，所有支持客户端验证的 ModelValidator 必须重写该方法以生成相应的客户端验证规则。

以用于进行范围验证的 RangeAttribute 特性对应的 RangeAttributeAdapter 为例，如下面的代码片段所示，它重写了 GetClientValidationRules，返回的 ModelClientValidationRule 列表中包含一个 ModelClientValidationRangeRule 对象。该 ModelClientValidationRangeRule 对象的验证类型为“range”，采用 RangeAttributeAdapter 的 ErrorMessage 属性作为自身的错误消息。作为验证范围上、下限的两个属性（Maximum 和 Minimum）成为了该 ModelClientValidationRule 的两个验证参数，参数名分别为“max”和“min”。

```
public class RangeAttributeAdapter :
    DataAnnotationsModelValidator<RangeAttribute>
{
    //其他成员
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules()
    {
        string errorMessage = base.ErrorMessage;
        return new ModelClientValidationRangeRule[] {
            new ModelClientValidationRangeRule(errorMessage,
            base.Attribute.Minimum, base.Attribute.Maximum) };
    }
}

public class ModelClientValidationRangeRule : ModelClientValidationRule
{
    public ModelClientValidationRangeRule(string errorMessage,
        object minValue, object maxValue)
    {
        base.ErrorMessage = errorMessage;
        base.ValidationType = "range";
        base.ValidationParameters["min"] = minValue;
        base.ValidationParameters["max"] = maxValue;
    }
}
```

客户端验证在这里还涉及一个重要的接口 IClientValidatable，它具有唯一的方法 GetClientValidationRules，用来返回一个以 ModelClientValidationRule 对象表示的客户端验证规则列表。

```
public interface IClientValidatable
{
    IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context);
}
```

所有支持客户端验证的 ValidationAttribute 都需要实现 IClientValidatable 接口，并通过实现 GetClientValidationRules 方法提供对应的验证规则，而生成的验证规则需要与通过重写的 IsValid 方法实现的服务端验证逻辑一致。DataAnnotationsModelValidator 重写了 GetClientValidationRules 方法，如果对应的 ValidationAttribute 实现了 IClientValidatable 接口，那么它（ValidationAttribute）的 GetClientValidationRules 方法会被调用并将返回的 ModelClientValidationRule 列表作为该方法的返回值。

当我们在某个 View 中调用 HtmlHelper<TModel>的模板方法将 Model 对象的某个属性以表单元素呈现出来的时候，它会采用前面介绍的 ModelValidator 的提供机制根据目标属性对应的 ModelMetadata 创建相应的 ModelValidator，然后调用 GetClientValidationRules 方法得到一组表示客户端验证规则的 ModelClientValidationRule 列表。如果该列表不为空，它们将作为验证属性附加到目标属性对应的<input>元素中。

## 9.3.3 自定义验证

虽然在命名空间“System.ComponentModel.DataAnnotations”中具有一系列继承自 ValidationAttribute 的验证特性可以帮助我们完成基本的数据验证，但是在很多场景下的验证需要按照我们自定义的方式来进行，接下来以实例演示的方式来指导读者如何以自定义的方式实现服务端和客户端验证。

假设需要对表示出生日期的输入数据进行验证以确保其年龄在允许的范围内，为此我们创建了一个自定义的验证特性 AgeRangeAttribute。如下面的代码片段所示，AgeRangeAttribute 应用在表示出生日期的属性上，并且指定允许年龄范围（18~25）的上下限。

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

    [DisplayName("出生日期")]
    [AgeRange(18,25,ErrorMessage="年龄必须在{0}到{1}周岁之间！")]
    [DisplayFormat(ApplyFormatInEditMode = true,
        DataFormatString = "{0:yyyy-MM-dd}")]
    public DateTime? BirthDate { get; set; }
}
```

为了简单起见，我们直接让 AgeRangeAttribute 继承自 RangeAttribute。如下面的代码片段所示，我们在构造函数中以整数的形式指定表示年龄范围的下限和上限。服务端验证体现在重写的 IsValid 方法中，而为了让格式化的错误消息是针对年龄而不是针对出生日期，我们重写了

FormatErrorMessage 方法。

```
[AttributeUsage( AttributeTargets.Property)]
public class AgeRangeAttribute: RangeAttribute, IClientValidatable
{
    public AgeRangeAttribute(int minimum, int maximum)
        : base(minimum, maximum)
    { }

    public override bool IsValid(object value)
    {
        DateTime? birthDate = value as DateTime?;
        if (null == birthDate)
        {
            return true;
        }
        DateTime age = new DateTime(DateTime.Today.Ticks -
            birthDate.Value.Ticks);
        return (int)this.Minimum <= age.Year && age.Year <= (int)this.Maximum;
    }

    public override string FormatErrorMessage(string name)
    {
        return string.Format(CultureInfo.CurrentCulture,
            this.ErrorMessageString,
            this.Minimum, this.Maximum);
    }

    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context)
    {
        string errorMessage = FormatErrorMessage("");
        ModelClientValidationRule rule = new ModelClientValidationRule
            {ValidationType = "agerange", ErrorMessage = errorMessage };
        rule.ValidationParameters.Add("minage", this.Minimum);
        rule.ValidationParameters.Add("maxage", this.Maximum);
        yield return rule;
    }
}
```

AgeRangeAttribute 实现了 IClientValidatable 接口，在 GetClientValidationRules 方法中返回一个验证类型为“agerange”的 ModelClientValidationRule 对象。它具有两个验证参数 minage 和 maxage，分别表示年龄范围的下限和上限。通过调用 FormatErrorMessage 方法格式化后生成的消息成为该 ModelClientValidationRule 的错误消息。

ModelClientValidationRule 对象的 ValidationType 属性值最终表示 jQuery 验证插件的验证规则，而该规则通过一个对应的函数来实施验证。对于 AgeRangeAttribute 来说，其对应的客户端验证类型为“agerange”，为此我们在一个.js 文件中以此命名注册的验证函数，具体的定义如下所示。

```
jQuery.validator.addMethod("agerange",function (value, element, params) {
    value = value.replace(/(^\s*)(\s*$)/g, "");
    if (!value) {
        return true;
    }
    var minAge = params.minage;
    var maxAge = params.maxage;

    var birthDateArray = value.split("-");
    var birthDate = new Date(birthDateArray[0], birthDateArray[1],
        birthDateArray[2]);
    var currentDate = new Date();
    var age = currentDate.getFullYear() - birthDate.getFullYear();
    return age >= minAge && age <= maxAge;
});

jQuery.validator.unobtrusive.adapters.add("agerange", ["minage", "maxage"],
    function (options) {
    options.rules["agerange"] = {
        minage: options.params.minage,
        maxage: options.params.maxage
    };
    options.messages["agerange"] = options.message;
});
```

如上面的代码片段所示，我们调用全局对象 jQuery 的 validator 属性的 AddMethod 方法添加了一个用于进行年龄范围验证的函数，该函数对应的验证规则为“agerange”。验证函数具有 3 个参数，其中 value 和 element 分别代表被验证的数据值（表示出生日期的字符串）和 HTML 元素，而 params 参数则表示传入的验证参数（表示年龄范围的上、下限）。最后我们调用 jQuery.validator.unobtrusive.adapters 的 add 方法对验证规则 agerange 进行注册，并指定对应的验证参数名称列表。

我们在一个 ASP.NET MVC 应用中定义了以前面定义的 Person 类型为 Model 的 View。如下面的代码片段所示，我们调用 HtmlHelper<TModel>的扩展方法 EditorForModel 将作为 Model 的 Person 对象以编辑模式呈现在一个表单之中，通过<script>标签引用了 4 个.js 文件，前 3 个是 ASP.NET MVC 原生脚本，最后一个是我们自定义的脚本文件，上面定义的“agerange”相关的扩展就定义在这个文件中。

```
@model Person
<html>
    <head>
        <script type="text/javascript"
                src="@Url.Content("~/Scripts/jquery-1.10.2.js")">
        </script>
        <script type="text/javascript"
                src="@Url.Content("~/Scripts/jquery.validate.js")">
        </script>
        <script type="text/javascript"
```

```
            src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.js")">
        </script>
        <script type="text/javascript"
                src="@Url.Content("~/Scripts/jQuery.validator.extend.js")">
        </script>
        <title>编辑个人信息</title>
    </head>
    <body>
        @using(Html.BeginForm())
        {
            @Html.EditorForModel()
            <input type="submit" value="保存" />
        }
    </body>
</html>
```

运行程序并在浏览器中将该 View 呈现出来，在输入违反年龄限制的出生日期的时候，客户端验证将会生效并以图 9-9 所示的形式显示出错误消息。（S908）

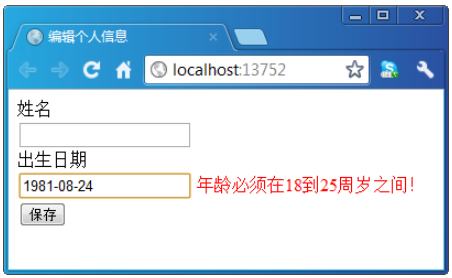



图 9-9 自定义验证对错误消息的呈现

如果我们查看最终生成的 HTML，会发现表示出生日期的文本框具有如下所示的定义，高亮显示的 3 个以“data-val-”为前缀的属性内容正是根据 AgeRangeAttribute 特性的 GetClientValidationRules 方法返回的 ModelClientValidationRule 对象生成的。另一个名为“data-val-date”的属性是默认添加的针对日期格式的验证规则，由于没有引用相应的本地化 JavaScript 文件，所以错误消息被默认设置为英文。

```
<input class="text-box single-line"
       data-val="true"
       data-val-agerange          ="年龄必须在 18 到 25 周岁之间！"
       data-val-agerange-maxage  ="25"
       data-val-agerange-minage  ="18"
       data-val-date="The field 出生日期 must be a date."
       id="BirthDate" name="BirthDate" type="text" value="" />
```