# 模型验证

**模型验证**（Model Validation）：是确保用户接收的数据适合于绑定的模型，并且在不合适时，给用户提供有用的信息，以帮助他们修正其问题的过程。

**模型验证过程一**：检查接收的数据——是保持域模型完整性的方式之一。

**模型验证过程二**：帮助用户修正问题。

## 示例项目介绍

项目模板：Basic

项目名称：ModelValidation

一个新的模型类文件：Appointment.cs

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Web;

namespace ModelValidation.Models

{

public class Appointment

{

public string ClinetName { get; set; }

[DataType(DataType.Date)]

public DateTime Date { get; set; }

public bool TermsAccepted { get; set; }

}

}

控制器：Home，用来提供处理Appointment模型类的动作方法

using ModelValidation.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ModelValidation.Controllers

{

public class HomeController : Controller

{

public ViewResult MakeBooking()

{

return View(new Appointment { Date = DateTime.Now });

}

[HttpPost]

public ViewResult MakeBooking(Appointment appt)

{

// 在实际项目中，此处是存储库中存储新的 Appointment 的语句

return View("Completed", appt);

}

}

}

本例不打算实现具体存储等方面的功能，主要是为了能够将注意力集中在模型绑定以及验证过程方面。要清楚验证模型的主要目的就为了防止在存储库中放置劣质或无意义的数据，并避免引起问题。

下面继续创建两个视图MakeBooking.cshtml、Completed.cshtml：

MakeBooking.cshtml，用来提供一个让用户创建新预约的表单：

@model ModelValidation.Models.Appointment

@{

ViewBag.Title = "Make A Booking";

}

<h4>Book an Appointment</h4>

@using (Html.BeginForm())

{

<p>Your name: @Html.EditorFor(m => m.ClinetName)</p>

<p>Appointment Date: @Html.EditorFor(m => m.Date)</p>

<p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms $ conditions</p>

<input type="submit" value="Make Booking" />

}

用于提交表单后给用户显示预约细节的内容的Completed.cshtml视图：

@model ModelValidation.Models.Appointment

@{

ViewBag.Title = "Completed";

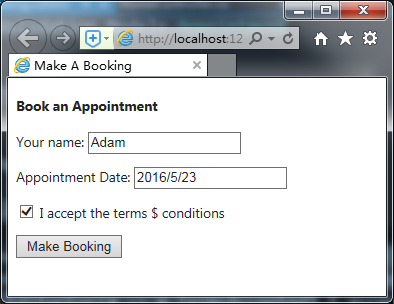
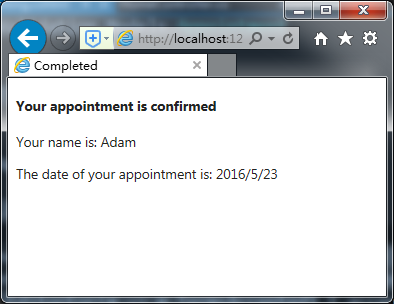
}

<h4>Your appointment is confirmed</h4>

<p>Your name is: @Html.DisplayFor(m => m.ClinetName)</p>

<p>The date of your appointment is: @Html.DisplayFor(m => m.Date)</p>

导航地址到/Home/MakeBooking可以看到效果：

按照现在的情况，此刻程序会接收用户递交的任何数据，但为了保持程序和域模型的完整性，在接收用户递交的Appointment之前，需要确保三件事为“true”。

* 用户必须提供一个姓名
* 用户必须提供一个未来的日期（以yyyy/mm/dd格式）
* 用户必须选择了复选框，已接受条款和条件

在这个示例中，模型验证就是实施这些需求的过程。后面将采不同的技术来实现，以检查用户提供的数据，并在输入无效数据时给予提示。

## 明确地验证模型

最直接的验证模型的方式是在动作方法中进行验证。下面是在MakeBooking动作方法的HttpPost版本中对Appointment类所定义的每一个属性添加明确的检查：

[HttpPost]

public ViewResult MakeBooking(Appointment appt)

{

**if (string.IsNullOrEmpty(appt.ClinetName))**

**{**

**ModelState.AddModelError("ClinetName", "Please enter your name");**

**}**

**if (ModelState.IsValidField("Date") && DateTime.Now > appt.Date)**

**{**

**ModelState.AddModelError("Date", "Please enter a date in the future");**

**}**

**if (!appt.TermsAccepted)**

**{**

**ModelState.AddModelError("TermsAccepted", "You must accept the terms");**

**}**

**if (ModelState.IsValid)**

**{**

// 在实际项目中，此处是存储库中存储新的 Appointment 的语句

return View("Completed", appt);

**}**

**else**

**{**

**return View();**

**}**

}

上面对Appointment对象的各个属性进行检测时，使用的ModelState.IsValidField方法检查的是绑定器是否能够对一个属性赋值。

注意：如果无法从请求数据中解析到一个值，执行额外检测或报告其他错误消息是没有意义的。

上面示例中在验证了模型对象的所有属性之后，读取了ModelState.IsValid属性，以考察是否有错误发送，如果在检查期间调用Model.State.AddModelError方法，或创建Appointment对象时模型绑定器遇到了问题，该方法返回false。

if (ModelState.IsValid)

{

// 在实际项目中，此处是存储库中存储新的 Appointment 的语句

return View("Completed", appt);

}

else

{

return View();

}

如果IsValid值为true，则会得到一个有效的Appointment对象，且会渲染Completed.cshtml视图；如果是false，则说明出现了问题，此时将掉牙View方法渲染默认视图。

**向用户显示验证错误**

通过调用View方法来处理验证错误看上去可能有点怪，但是在MakeBooking.cshtml视图中用来生成input元素的模板视图辅助器，会检查视图模型的验证错误。

如果一个属性报告了错误，那么辅助器对相应的input元素添加一个CSS的class标签属性，其值为input-validation-error**。**~/Content/site.css文件中包含了一个用于该样式的默认定义，如：

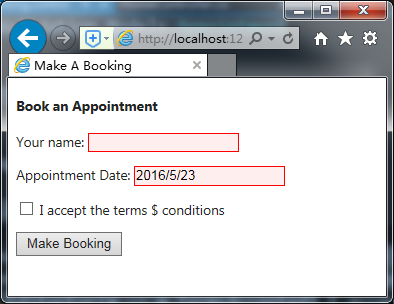
.input-validation-error {

border: 1px solid #f00;

background-color: #fee;

}

验证错误的效果如图：



设置复选框样式

有些浏览器，包括Chrome和Firefox，会忽略运用于复选框的样式，这会导致不协调的视觉反馈。解决办法是在~/Views/Shared/EditorTemplates/Boolean.cshtml中创建一个自定义模板，并将复选框封装在一个div元素中。以下是所使用的一个模板，但我们可以将其定制到自己的应用程序中：

@model bool?

@if (ViewData.ModelMetadata.IsNullableValueType)

{

@Html.DropDownListFor(m => m, new SelectList(new[] { "Not Set", "True", "False" }, Model))

}

else

{

ModelState state = ViewData.ModelState[ViewData.ModelMetadata.PropertyName];

bool value = Model ?? false;

if (state != null && state.Errors.Count > 0)

{

<div class="input-validation-error" style="float:left">

@Html.CheckBox("", value)

</div>

}

else

{

@Html.CheckBox("", value)

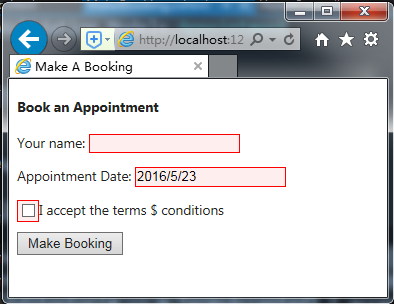
}

}

如果运用这个模板的属性出现错误，该模板会将一个复选框（CheckBox）封装在一个运用了input-validation-error样式的div元素中。

只有当提交的数据能被模型绑定器解析，且能被验证通过，才会显示完成视图，否则，将会提示验证失败的视图。

运用了上面复选框样式后的效果：



## 显示验证消息

前面已经实现了高亮显示错误字段（通过对input元素的CSS样式设置），但用户仍不能清除的知道问题何在。对于这一问题，可以使用Html.ValidationSummary辅助器方法显示已经注册到该页面的验证错误的摘要，这也是一种方便而简单的做法，如果没有错误，该辅助器便不会产生任何HTML，用法如下（粗体部分）：

@model ModelValidation.Models.Appointment

@{

ViewBag.Title = "Make A Booking";

}

<h4>Book an Appointment</h4>

@using (Html.BeginForm())

{

**@Html.ValidationSummary()**

<p>Your name: @Html.EditorFor(m => m.ClinetName)</p>

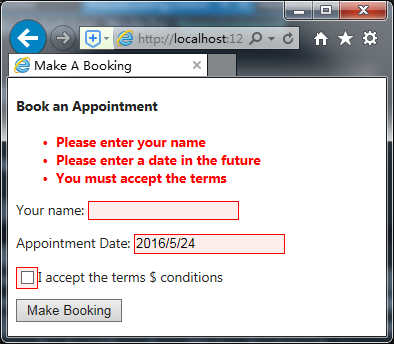
<p>Appointment Date: @Html.EditorFor(m => m.Date)</p>

<p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms $ conditions</p>

<input type="submit" value="Make Booking" />

}

效果如图：



对应的显示这段错误消息的HTML如下：

<div class="validation-summary-errors" data-valmsg-summary="true">

<ul>

<li>Please enter your name</li>

<li>Please enter a date in the future</li>

<li>You must accept the terms</li>

</ul>

</div>

ValidationSummary辅助器还有一些重载版本，有一些是允许开发时候设置显示模型级错误的。一些常用的如下所列：

* Html.ValidationSummary()：生成所有验证错误的摘要
* Html.ValidationSummary(bool)：如果bool参数为true，那么只显示模型级错误。如果是false则显示所有错误
* Html.ValidationSummary(string)：在所有验证错误摘要之前显示一条消息（消息内容在string参数中）
* Html.ValidationSummary(bool,string)：在验证错误消息前显示一条消息。如果bool参数为true，则只显示模型级错误

当存在由于两个或多个属性值之间的相互作用而引发的错误时，可以使用模型级错误。这里假设一个场景，作为我们的示例：假设姓名为“Joe”的客户不能在星期一预约。办法就是在MakeBooking动作方法中进行明确的验证检查，并报告模型级验证错误：

[HttpPost]

public ViewResult MakeBooking(Appointment appt)

{

if (string.IsNullOrEmpty(appt.ClinetName))

{

ModelState.AddModelError("ClinetName", "Please enter your name");

}

if (ModelState.IsValidField("Date") && DateTime.Now > appt.Date)

{

ModelState.AddModelError("Date", "Please enter a date in the future");

}

if (!appt.TermsAccepted)

{

ModelState.AddModelError("TermsAccepted", "You must accept the terms");

}

**if (ModelState.IsValidField("ClientName")**

**&& ModelState.IsValidField("Date")**

**&& appt.ClinetName == "Joe"**

**&& appt.Date.DayOfWeek == DayOfWeek.Monday)**

**{**

**ModelState.AddModelError("","Joe cannot book appointment on Mondays");**

**}**

if (ModelState.IsValid)

{

// 在实际项目中，此处是存储库中存储新的 Appointment 的语句

return View("Completed", appt);

}

else

{

return View();

}

}

示例的粗体部分的ModelState.AddModelError方法的第一个参数给出的是空字符串（""），这样就可以注册一条模型级错误。后面修改一下MakeBooking.cshtml视图，使其只显示模型机错误，这样方便我们看到最终的效果：

@model ModelValidation.Models.Appointment

@{

ViewBag.Title = "Make A Booking";

}

<h4>Book an Appointment</h4>

@using (Html.BeginForm())

{

**@Html.ValidationSummary(true)**

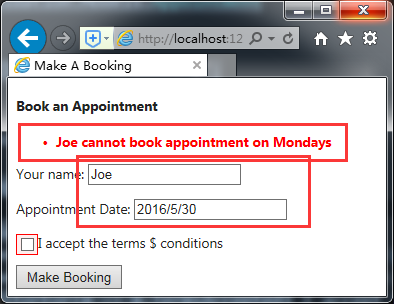
<p>Your name: @Html.EditorFor(m => m.ClinetName)</p>

<p>Appointment Date: @Html.EditorFor(m => m.Date)</p>

<p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms $ conditions</p>

<input type="submit" value="Make Booking" />

}



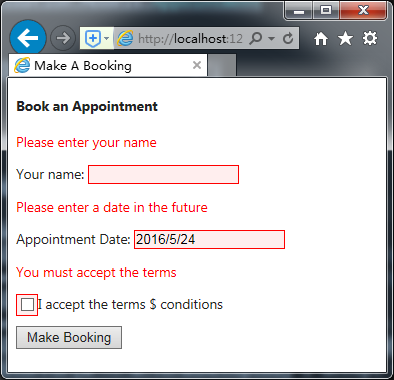
这样已修改，从上图的效果中可以看出出现了两个错误。一个是模型级的错误，就是红色错误消息部分；第二个是是没有勾选复选框导致的，但是现在不会再显示问题的描述消息了。

**显示属性级验证消息**

由于属性级验证消息是可以显示在相应字段的旁边的，所以，就很容易把模型级错误和属性级错误分开，将模型级错误作为整体的摘要显示了，这样也更合理一些。

要想单独在相应的字段旁边显示属性级验证消息，一般可以通过使用Html.ValidationMessageFor辅助器方法实现。

下面是再次对MakeBooking.cshtml视图的调整：

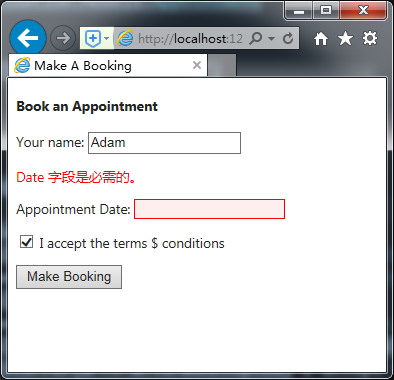


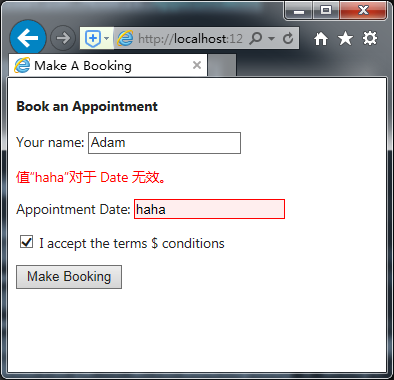
## 使用其他验证技术

### 在模型绑定器中执行验证

验证也是默认模型绑定器的绑定过程的一部分。模型绑定器会为模型对象中的每一个属性执行一些基本的验证。

下图展示了如果为示例项目中模型对象的Date属性递交空或错误内容的验证结果，这都是默认模型绑定器实现的：





默认的模型绑定器类DefaultModelBinder提供了一些可以重写的方法，以使得开发时可以方便的为一个绑定器添加验证。通过重写这些方法，便可以在创建自定义模型绑定器时，将自己的验证逻辑放在绑定过程之中。

建议：对于使用自定义验证逻辑时，建议在模型类运用元数据的方式处理验证，而不在模型绑定器中添加自定义验证逻辑。

DefaultModelBinder中对绑定过程添加验证的方法：

|  |  |  |
| --- | --- | --- |
| 方法 | 描述 | 默认实现 |
| OnModelUpdated | 在绑定器试图对模型对象中的所有属性进行赋值时调用 | 运用由模型元数据定义的验证规则，并用ModelState注册错误。 |
| SetProperty | 在绑定器对一个特定属性运用一个值时调用 | 如果该属性不能保存null值，并且没有可以运用的值，那么，将用ModelState注册一条“The<name> field is required (<name>字段是必需的)”的错误消息（如上图中显示的“Date 字段是必需的”）。如果有一个值，但不能进行解析，将注册“The value <> is not valid for <name>（值”<value>”对 <name> 无效）”的错误 |

### 用元数据指定验证规则

使用元数据指定规则的一个优点是，在整个项目中运用绑定过程的任何地方，都会强制执行验证规则，而不只存在于个别动作方法之中。

拿Appointment作为示例，看看是如何实现这种验证方式的：

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Web;

namespace ModelValidation.Models

{

public class Appointment

{

**[Required]**

public string ClinetName { get; set; }

[DataType(DataType.Date)]

**[Required(ErrorMessage = "Please enter a date")]**

public DateTime Date { get; set; }

**[Range(typeof(bool), "true", "true", ErrorMessage = "You must accept the terms")]**

public bool TermsAccepted { get; set; }

}

}

上面代码中运用了两个验证注解属性——Required和Range。下面给出了内建验证注解属性的一些介绍：

* Compare：

示例：[Compare("MyOtherProperty")]

描述：两个属性必须有同样的值。当要求用户对一个属性提供两次同样的值时，这是有用的。如：一个邮件地址或一个口令

* Range：

示例：[Range (10,20)]

描述：一个数字值（或实现IComparable的任何属性类型），必须不超出指定的最小值和最大值。为了指定只有一端的边界，可以用一个MinValue或MaxValue常数。如：[Range (int.MinValue,50)]

* RegularExpression：

示例：[RegularExpression ("pattern")]

描述：一个字符串值，必须匹配指定的正则表达式模式。注意，该模式必须匹配用户提供的所有值，而不只是其中的一个子串。默认它是大小写敏感的，但可以通过运用(?!)修饰符，使大小写不敏感。如：[RegularExpression ("(?!)mypattern")]

* Required：

示例：[Required]

描述：必须是一个非空值，或一个不是只含空格的字符串。如果希望空格作为可接受值，可以用[Required(AllowEmptyStrings = true)]

* StringLength：

示例：[StringLength (10)]

描述：一个字符串，必须不超过指定的最大长度。也可以指定一个最小长度：[StringLength (10,MinimumLength=2)]

所有上述这些验证注解属性都可以给ErrorMessage属性设置一个值，以指定一个自定义的错误消息，如：

[Required(ErrorMessage = "Please enter a date")]

内建的验证注解属性是很基本的，且只能做属性级验证。即使这样，仍需要一些技巧。可以考虑下面这个例子：

[Range(typeof(bool), "true", "true", ErrorMessage = "You must accept the terms")]

如果希望用户选中了复选框，以接受条款。此时不能使用Required注解属性，因为用于布尔值的模板辅助器会产生一个隐藏的HTML元素，以确保即使复选框未选中也会获得一个值。为了解决这一问题，可以利用Range注解属性的一个特性，它让我们提供一个类型，并以字符串值指定上下限。通过把上下限均设置为true，可以为使用复选框的布尔属性的编辑器视图创建一个等效的Required注解属性。

注意：DataType注解属性不能用于验证用户输入——只能对使用模板辅助器进行渲染的值提供提示。因此，不要指望用DataType(DataType.EmailAddress)这样的注解属性来强制一个特定的格式。

1. 创建自定义的属性验证注解属性

使用Range注解属性类固然能够变相的实现Required注解属性的功能，但这还是显得有些笨拙。好在，可以创建自定义的属性验证注解属性，而这只需要从ValidationAttribute类进行派生即可。

为了演示其工作机制，需要向项目中添加一个Infrastructure文件夹，并在其中创建一个MustBeTrueAttribute类：

using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Infrastructure

{

public class MustBeTrueAttribute : ValidationAttribute

{

public override bool IsValid(object value)

{

return value is bool && (bool)value;

}

}

}

在这个示例中，验证逻辑很简单。如果这是一个具有true值的bool型，那么这个值就是有效的。下面看看如何使用这个自定义的注解属性，以实现前面的Range注解属性的验证效果：

using ModelValidation.Infrastructure;

using System;

**using System.ComponentModel.DataAnnotations;**

namespace ModelValidation.Models

{

public class Appointment

{

[Required]

public string ClinetName { get; set; }

[DataType(DataType.Date)]

[Required(ErrorMessage = "Please enter a date")]

public DateTime Date { get; set; }

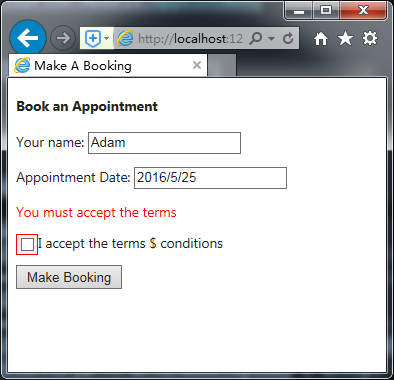
**[MustBeTrueAttribute(ErrorMessage = "You must accept the terms")]**

public bool TermsAccepted { get; set; }

}

}

这样看起来会更简便，下图是最终效果：



还有一种方式是，通过内建的验证注解属性来派生新类，这等于提供了扩展内建验证注解属性行为的能力。下面是Infrastructure中的示例类FutureDateAttribute：

using System;

using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Infrastructure

{

public class FutureDateAttribute : RequiredAttribute

{

public override bool IsValid(object value)

{

return base.IsValid(value) && ((DateTime)value) > DateTime.Now;

}

}

}

在代码中，由于使用的基类的IsValid方法，所以重写方法包含了所有基本验证步骤，下面是对其应用：

using ModelValidation.Infrastructure;

using System;

using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models

{

public class Appointment

{

[Required]

public string ClinetName { get; set; }

[DataType(DataType.Date)]

**[FutureDateAttribute(ErrorMessage = "Please enter a date in the future")]**

public DateTime Date { get; set; }

[MustBeTrueAttribute(ErrorMessage = "You must accept the terms")]

public bool TermsAccepted { get; set; }

}

}

1. 创建模型验证注解属性

现在来看看自定义的模型验证注解属性，为了演示，在Infrastructure文件夹中创建NoJoeOnMondaysAttribute.cs类：

using System;

using System.ComponentModel.DataAnnotations;

using ModelValidation.Models;

namespace ModelValidation.Infrastructure

{

public class NoJoeOnMondaysAttribute : ValidationAttribute

{

public NoJoeOnMondaysAttribute()

{

ErrorMessage = "Joe cannot book appointment on Mondays";

}

public override bool IsValid(object value)

{

Appointment app = value as Appointment;

if (app == null

|| string.IsNullOrEmpty(app.ClinetName)

|| app.Date == null)

{

// 还没有正确的类型的模型要验证，或者还没有所需要的 ClientName 和 Date 属性的值

return true;

}

else

{

return !(app.ClinetName == "Joe" && app.Date.DayOfWeek == DayOfWeek.Monday);

}

}

}

}

与单个属性的情况相比，在模型类上使用的验证注解属性时，模型绑定器传递给IsValid方法的object参数将是模型对象，比如Appointment。

对于上述代码，该验证注解属性会进行检查，以确保已经有了一个Appointment对象，且如果有，则也就有了可以使用的ClientName和Date属性的值。如果是这样，就可以确保Joe不能试图预约星期一。下面给出了在Appointment类上的使用情况：

using ModelValidation.Infrastructure;

using System;

using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models

{

**[NoJoeOnMondays]**

public class Appointment

{

[Required]

public string ClinetName { get; set; }

[DataType(DataType.Date)]

[FutureDateAttribute(ErrorMessage = "Please enter a date in the future")]

public DateTime Date { get; set; }

[MustBeTrueAttribute(ErrorMessage = "You must accept the terms")]

public bool TermsAccepted { get; set; }

}

}

此时，在动作方法中执行着两个同一种类的验证，而且都使用验证注解属性，这也意味着，对同一验证问题，用户会看到两个类似的错误消息。为了解决这一问题，已经从Home控制器的MakeBooking动作方法中删除了明确的验证检查，效果是让验证注解属性全权负责执行自定义的验证检查：

using ModelValidation.Models;

using System;

using System.Web.Mvc;

namespace ModelValidation.Controllers

{

public class HomeController : Controller

{

public ViewResult MakeBooking()

{

return View(new Appointment { Date = DateTime.Now });

}

**[HttpPost]**

**public ViewResult MakeBooking(Appointment appt)**

**{**

**if (ModelState.IsValid)**

**{**

**// 在实际项目中，此处是存储库中存储新的 Appointment 的语句**

**return View("Completed", appt);**

**}**

**else**

**{**

**return View();**

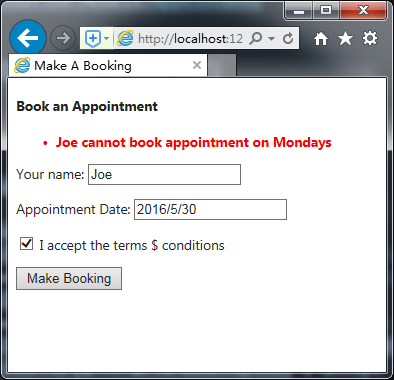
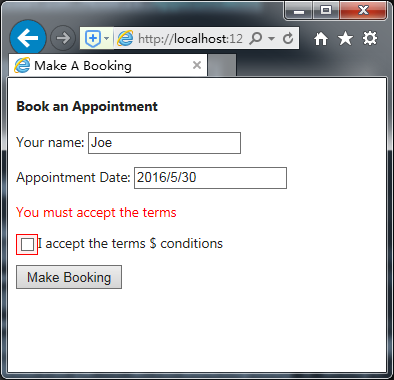
**}**

**}**

}

}

需要了解当检查到属性级问题时，模型级验证注解属性不会生效。此时，启动程序，并输入姓名Joe和日期2016/5/30，并取消复选框的选中状态，当递交表单时，会看到只有关于复选框的警告。然后选中复选框，然后再次递交，只有这时才会看到模型级错误：



### 定义自验证模型

通过继承IValidatableObject接口，便可以将验证逻辑在模型中实现，者便可得到自验证模型：

using ModelValidation.Infrastructure;

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models

{

public class Appointment : IValidatableObject

{

public string ClinetName { get; set; }

[DataType(DataType.Date)]

public DateTime Date { get; set; }

public bool TermsAccepted { get; set; }

**public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)**

**{**

**List<ValidationResult> errors = new List<ValidationResult>();**

**if (string.IsNullOrEmpty(ClinetName))**

**{**

**errors.Add(new ValidationResult("Please enter your name"));**

**}**

**if (DateTime.Now > Date)**

**{**

**errors.Add(new ValidationResult("Please enter a date in the future"));**

**}**

**if (errors.Count == 0 && ClinetName == "Joe")**

**{**

**errors.Add(new ValidationResult("Joe cannot book appointment on Mondays"));**

**}**

**if (!TermsAccepted)**

**{**

**errors.Add(new ValidationResult("You must accept the terms"));**

**}**

**return errors;**

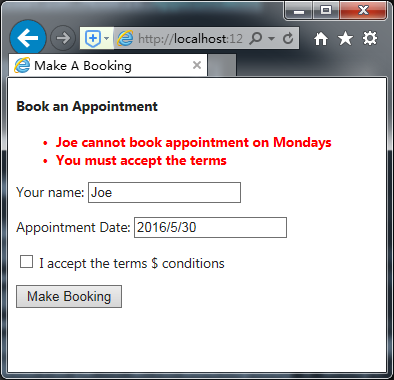
**}**

}

}

IValidatableObject接口定义的Validate方法以ValidationContext为参数，这个参数不是MVC专用的，且也不常用。Validate方法的返回结果是ValidationResult对象的一个枚举，其中每一个都表示一个验证错误。

如果模型类实现了IValidatableObject接口，那么在模型绑定器对每一个模型属性赋值之后，便会调用该Validate方法。这样做的好处是，它结合了将验证逻辑放入动作方法的灵活性，但又具有任何时候都会运用模型绑定过程创建模型类型实例的一致性。另一个好处是，可以在一个地方将模型级和属性级验证结合在一起，这意味着所有错误都显示在一起。很多人不习惯这么做，但这也是十分适合MVC的设计模式的，且这样拥有了灵活性和一致性。



## 执行客户端验证

之前讲的验证技术都是服务器端验证。很多情况下需要得到即时的反馈，这就需要客户端验证了。客户端验证通常是使用JavaScript实现的，由于这种验证是在客户端进行的，也就是说在数据传递到服务端之前，这样就给用户提供了即时反馈并修正错误的机会。

MVC框架支持**渐进式客户端验证（Unobtrusive Client-Side Validation）**。渐进式指的是在生成的HTML元素上添加验证标签属性来表示验证规则。这些标签属性由包含在MVC框架中的JavaScript库进行解释，框架又转而依靠对jQUery验证库所作的配置，并完成实际的验证工作。

提示：客户端验证致力于验证个别属性。并且如果使用MVC内建的支持，则很难建立模型级的客户端验证。

客户端验证机制可参见下图：

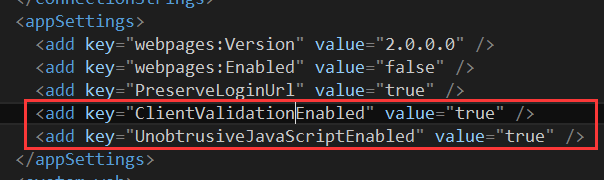
验证标签属性

JavaScript解析

依靠对jQuery验证库所作的配置与验证库完成实际的验证工作

### 启用和禁用客户端验证

客户端验证是由Web.config文件中的两个设置来控制的。如：



为了是客户端验证生效，红框中的这两个属性必须设置为true。默认创建MVC项目时这些会自动创建，并被设置为true。

提示：也可以配置基于个别视图的客户端验证，只需要在视图的一个razor代码块中设置HtmlHelper.ClientValidationEnabled和HtmlHelper.UnobtrusiveJabaScriptEnabled即可。

在上面配置正确后，还需要确保在发送给浏览器的HTML中引用了以下这三个JavaScript库才行：

* /Scripts/jquery-1.8.2.min.js：jQuery主库
* /Scripts/jquery.validate.min.js：jQuery验证
* /Scripts/jquery.validate.unobtrusive.min.js：jQuery渐进式验证

给一个视图添加这些JavaScript文件最简单的方式是使用新的脚本捆绑包特性（其工作原理在相关专项中再做介绍）。这是MVC4中新增特性。下面是一个示例，我们修改的是项目的布局视图：\_Layout.cshtml：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width" />

<title>@ViewBag.Title</title>

@Styles.Render("~/Content/css")

@Scripts.Render("~/bundles/modernizr")

</head>

<body>

@RenderBody()

@Scripts.Render("~/bundles/jquery")

**@Scripts.Render("~/bundles/jqueryval")**

@RenderSection("scripts", required: false)

</body>

</html>

### 使用客户端验证

下面删除了Appointment模型类中IValidatableObject接口的实现，因为它对客户端验证无效，另外，还做了其他修改，以使其支持客户端验证：

using ModelValidation.Infrastructure;

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models

{

public class Appointment

{

**[Required]**

**[StringLength(10, MinimumLength = 3)]**

public string ClinetName { get; set; }

[DataType(DataType.Date)]

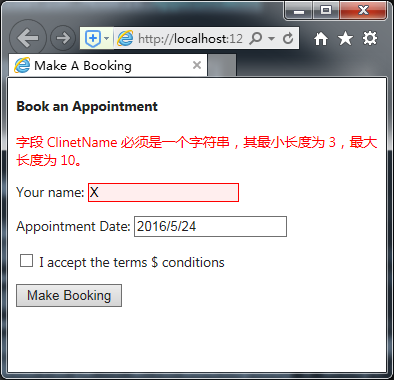
public DateTime Date { get; set; }

public bool TermsAccepted { get; set; }

}

}

下图是在”Your name”文本框中输入X后，使用Tab键或点击其他输入元素后即可显示出来的效果：



验证结果之所以即时生效，实际上是因为执行验证的JavaScript代码会阻止递交表单，直到不再出现为止（如果用户继续输入正确的内容，错误消息将会即时消失）。

### 理解客户端验证机制

使用MVC框架客户端验证特性的优点之一是不必编写JavaScript，而是用HTML标签属性来表示验证规则。下面对比一下启用和禁用客户端验证时所得到的HTML，下例是以ClientName属性，通过Html.EditorFor辅助器所渲染的结果：

禁用：

<input name="ClinetName" class="text-box single-line" id="ClinetName" type="text" value="">

启用：

<input name="ClinetName" class="text-box single-line" id="ClinetName" type="text" value="" data-val-required="ClinetName 字段是必需的。" data-val-length-min="3" data-val-length-max="10" data-val-length="字段 ClinetName 必须是一个字符串，其最小长度为 3，最大长度为 10。" data-val="true">

上面启用时被标黄色背景的为与客户端验证相关的项。

MVC的客户端验证支持不会生成任何JavaScript脚本或JSON数据来指导验证过程，这就像框架的其他部分，依靠的是约定：添加的标签属性以data-val为前缀，jQuery验证库通过检查这个标签属性，会识别出这是一个需要验证的字段。

各个验证规则是以一个data-val-<name>标签属性的形式指定的，其中name是所运用的规则。如：运用于模型类的Required注解属性，生成的是data-val-required标签属性。有些规则需要额外的标签属性，如长度规则：data-val-length-min和data-val-length-max标签属性，用例指定最小和最大字符串长度。

MVC客户端验证的一个很好的特点就是，不管是客户端还是服务器端验证，用以指定验证规则的属性是相同的。也就是说，从不支持JavaScript的浏览器而来的数据会得到通用的验证——将转由服务器端进行验证，而无需做额外的任何工作。**渐进式**也就是指的这一点：当jQuery验证库不能正常发挥作用时，这些标签属性在客户端将是一些正常的HTML标记（但不起作用），此时客户端验证失效，而验证则会在服务器端进行。

MVC客户端验证与jQuery验证

MVC客户端验证：建立在jQuery验证库之上，借助于验证标签属性来使用jQuery验证库。无需编写脚本。

jQuery验证：通过JavaScript直接使用jQuery验证库的方法叫做jQuery验证。需要通过脚本调用jQuery验证库。

## 执行远程验证

**远程验证（Remote Validation）**：这是一项客户端技术，但需要调用服务器上的一个动作方法来执行验证。

一个常见的例子：当要求用户名必须唯一时，检查这个用户名在应用程序中是否有效。用户递交数据，并执行客户端验证。作为这个过程的一部分，会形成一个发送到服务器的Ajax请求，以验证被请求的这个用户名。如果取到了这个用户名，便显示一个验证错误。

这看起来很像常规的服务器端验证，但这种方法有一些好处：

1. 只有某几个属性会被远程验证，客户端验证的好处仍然运用于用户已经输入的其他数据值。
2. 期间的请求相对较轻，且侧重于验证，而不是处理整个模型对象。这样最小化了请求所产生的性能冲突。
3. 远程验证是在后台执行的。用户不必点击提交按钮，然后等待渲染并返回一个新的视图。它形成更灵敏的用户体验，特别是当浏览器与服务器之间的网络较慢时。

这就是说，远程验证是一种折衷：它让用户在客户端与服务器端验证之间进行权衡，但它确实请求了应用服务器，而且它不如常规的客户端验证那么迅速。

从上面的描述可知，使用远程验证是需要有一个能够验证某个模型属性的动作方法。下面以Appointment模型的Date属性为例，实现确保所请求的预约是一个未来的日期。

在Home控制器中添加ValidateDate动作方法：

using ModelValidation.Models;

using System;

using System.Web.Mvc;

namespace ModelValidation.Controllers

{

public class HomeController : Controller

{

public ViewResult MakeBooking()

{

return View(new Appointment { Date = DateTime.Now });

}

[HttpPost]

public ViewResult MakeBooking(Appointment appt)

{

if (ModelState.IsValid)

{

// 在实际项目中，此处是存储库中存储新的 Appointment 的语句

return View("Completed", appt);

}

else

{

return View();

}

}

**public JsonResult ValidateDate(string Date)**

**{**

**DateTime parsedDate;**

**if (!DateTime.TryParse(Date, out parsedDate))**

**{**

**return Json("Please enter a valid date (yyyy/mm/dd)", JsonRequestBehavior.AllowGet);**

**}**

**else if (DateTime.Now > parsedDate)**

**{**

**return Json("Please enter a date in the future", JsonRequestBehavior.AllowGet);**

**}**

**else**

**{**

**return Json(true, JsonRequestBehavior.AllowGet);**

**}**

**}**

}

}

支持远程验证的动作方法的要求：

* 方法必须返回JsonResult类型的结果，以告知MVC框架，正在使用JSON数据。
* 相对结果而言，验证动作方法必须定义一个与被验证的数据字段同名的参数。——这也是为什么动作方法ValidateDate的参数为Date的原因。

提示：可以使用模型绑定，以使递交到该动作方法的参数是一个DateTime对象，但这样做将意味着如果用户输入一个无意义的值，如apple，验证方法将不会被调用。这是因为模型绑定器不能为apple创建DateTime对象，并在尝试创建时，会抛出一个异常。远程验证特性没有表示这种异常的方式，因此，它会被静静地终止。这将导致不会高亮该数据字段的不良效果，而造成用户认为输入数据合法的印象。作为一种既定的规则，远程验证最好的办法是在动作方法中接收一个string参数，并执行各种类型的转换、解析或明确的模型绑定。

用Json方法表示验证结果。该方法创建了一个JSON格式的结果，这种结果能够被客户端远程验证脚本解析和处理。如果所处理的值满足要求，便把true作为参数传递给Json方法。如果这个值不合适，便把用户应当看到的验证错误消息作为参数进行传递。

注意在使用Json方法返回结果时，必须把JsonRequestBehavior.AllowGet值作为参数进行传递。——因为MVC框架默认不允许产生JSON格式的GET请求，因而开发者不得不重写这个行为。没有这个附加参数，验证请求将静静地终止，并且客户端将不会显示验证错误。

在模型类中通过对希望远程验证的属性使用Remote注解属性就可以达到目的了，就像下面这样：

using System;

using System.ComponentModel.DataAnnotations;

using System.Web.Mvc;

namespace ModelValidation.Models

{

public class Appointment

{

[Required]

[StringLength(10, MinimumLength = 3)]

public string ClinetName { get; set; }

[DataType(DataType.Date)]

**[Remote("ValidateDate", "Home")]**

public DateTime Date { get; set; }

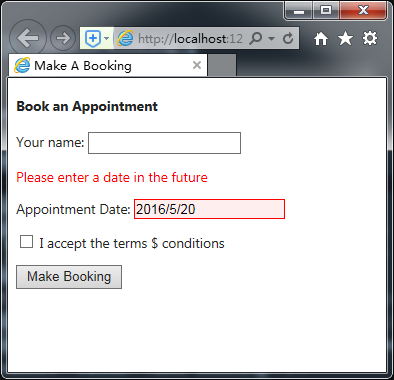
public bool TermsAccepted { get; set; }

}

}

该注解属性的参数是应该用来生成URL的动作和控制器的名称，JavaScript验证库将调用该URL来执行验证。

效果如下图：



注意：当用户第一次递交表单时，会调用验证动作方法，然后每次编辑数据时，这个方法都会被调用。其实，每输入一次内容都会导致对服务器的一次请求。对某些应用程序，这可能是相对数量的请求，当程序需要一定的服务器容量和带宽时，必须对此加以考虑。此外，也可以不对高开销的属性使用远程验证。