# 模型绑定

## 创建项目

项目名称：MvcModels

模板：Basic

下面是项目的基础文件及其内容：

模型类：Person.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace MvcModels.Models

{

public partial class Person

{

public int PersonId { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public DateTime BirthDate { get; set; }

public Address HomeAddress { get; set; }

public bool IsApproved { get; set; }

public Role Role { get; set; }

}

public class Address

{

public string Line1 { get; set; }

public string Line2 { get; set; }

public string City { get; set; }

public string PostalCode { get; set; }

public string Country { get; set; }

}

public enum Role

{

Admin,

User,

Guest

}

}

控制器：HomeController.cs

using MvcModels.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MvcModels.Controllers

{

public class HomeController : Controller

{

private Person[] personData =

{

new Person{PersonId = 1, FirstName = "Adam",LastName = "Freeman",Role = Role.Admin},

new Person{PersonId = 2 ,FirstName = "Steven",LastName = "Sanderson",Role = Role.Admin},

new Person{PersonId = 3 ,FirstName = "Jacqui",LastName = "Griffyth",Role = Role.User},

new Person{PersonId = 4 ,FirstName = "John",LastName = "Smith",Role = Role.User},

new Person{PersonId = 5 ,FirstName = "Anne",LastName = "Jones",Role = Role.Guest}

};

public ActionResult Index(int id)

{

Person dataItem = personData.Where(p => p.PersonId == id).First();

return View(dataItem);

}

}

}

视图：Index.cshtml（强类型）

@model MvcModels.Models.Person

@{

ViewBag.Title = "Index";

}

<h2>Person</h2>

<div><label>ID:</label>@Html.DisplayFor(m => m.PersonId)</div>

<div><label>First Name:</label>@Html.DisplayFor(m => m.FirstName)</div>

<div><label>Last Name:</label>@Html.DisplayFor(m => m.LastName)</div>

<div><label>Role:</label>@Html.DisplayFor(m => m.Role)</div>

CSS样式：Site.css

…

label {

display: inline-block;

width: 100px;

font-weight: bold;

margin: 5px;

}

form label {

float: left;

}

input.text-box {

float: left;

margin: 5px;

}

button[type=submit] {

margin-top: 5px;

float: left;

clear: left;

}

form div {

clear: both;

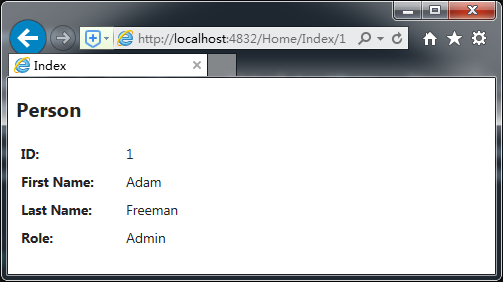
}

…

## 理解模型绑定

模型绑定在HTTP请求和C#（指的是MVC中的动作方法）直接起到了桥梁的作用。大多数MVC项目都在某种程度上依赖模型绑定。

看看刚才创建的项目是否能够正常启动，并查看模型绑定是否正常工作：



从上图看我们的模型绑定是没有问题的，这个页面对应的URL是Home/Index/1，它包含了查看Person对象的PersonId属性值，如：Home/Index/1。

MVC自动地对该URL进行了解析，并在调用Home控制器中的Index方法时获取1作为其参数，因此使得程序能够按照预期运行。这种将URL片段转换成int型方法参数的过程是模型绑定的一个例子。

当程序接收到请求并由路由引擎处理时，模型绑定过程便开始了。该示例中未对默认路由进行调整，也就不再这里展示了。

动作调用器会使用该路由信息推断出，为该请求进行服务所需要的是Index动作方法。但此时还不能调用，直到获取该方法的参数所需的有用的值。

默认的动作调用器，ControllerActionInvoker，要依靠模型绑定器来生成调用动作所需要的数据对象。模型绑定器由IModeBinder接口所定义：

namespace System.Web.Mvc

{

public interface IModelBinder

{

object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext);

}

}

在项目中，可以有多个模型绑定器，每一个绑定器可以绑定一个或多个模型类型。在动作调用器需要调用一个动作方法时，它会检查该方法所定义的参数，并查找各个参数类型所依赖的模型绑定器。

就这个示例而言，动作调用器会检查Index方法，并发现它具有一个int型的参数。然后便会查找负责int值绑定的绑定器，并调用它的BindModel方法。

模型绑定器负责提供能用于调用Index方法的int值，这通常意味着要对请求数据（如表单数据或查询字符串值）的某些元素进行转换，但是MVC框架对如何获取这些数据并无任何限制。

模型绑定的过程：

* 检测目标对象（要创建的对象——这种对象通常是动作方法的参数）的名称和类型；
* 通过对象名称查找数据源（请求），并找到可用数据（通常是字符串）；
* 根据对象类型将找到的数据值转换成目标类型；
* 通过对象名称、对象类型和这种经过处理的数据来构造目标对象；
* 将构造好的对象送给动作调用器，并由动作调用器将对象注入到目标动作方法中去。

## 使用默认的模型绑定器

当动作调用器找不到绑定某个类型的自定义绑定器时，将会使用默认的模型绑定器（DefaultModelBinder）。默认情况下，这个模型绑定器会按顺序依次搜索四个位置，具体如下表所示：

|  |  |  |
| --- | --- | --- |
| 序号 | 源 | 描述 |
| 1 | Request.Form | 由用户在HTML的form元素中提供的值 |
| 2 | RouteData.Values | 用应用程序路由获得的值 |
| 3 | Request.QueryString | 包含在请求URL中的查询字符串部分的数据 |
| 4 | Request.Files | 请求中上传的文件 |

就以上面的例子，DefaultModelBinder会这样搜索id参数的值：

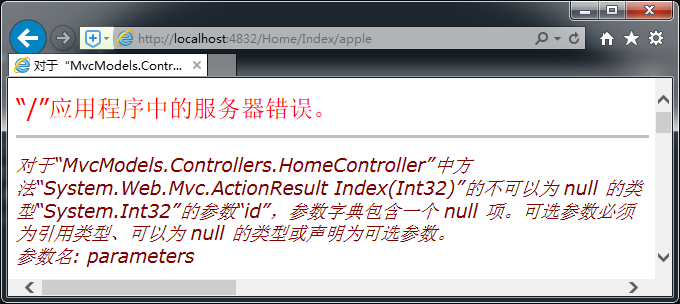
* Request.Form["id"]；
* RouteData.Values["id"]；
* Request.QueryString["id"]；
* Request.Files["id"]；

在上述的搜索过程中，直到找到适用的值，搜索才会停止。对于上述示例在搜索表单数据时将会失败，但会找到具有正确名称的路由变量（之所以能从路由中找到与参数对应的值，是因为动作方法的参数名称与路由变量名相对应，如果将动作方法参数改为类似personId这样的其他值，将不会得到匹配的数据值，最终请求将会失败。所以，在依靠默认模型绑定器的情况下，重要的是保证动作方法参数与寻找的数据属性相匹配）。也就是说在找到值后，将不会继续搜索查询字符串和上传文件的名称。

### 绑定简单类型

在处理简单类型时，DefaultModelBinder会尝试使用System.ComponentModel.TypeDescriptor类，将已经从请求数据获得的字符串值转换成参数类型。如无法成功转换，那么DefaultModelBinder便不能绑定该模型。

如下面启动程序后，导航到/Home/Index/apple地址，将会看到绑定失败的问题：



可以为模型绑定器做一点简化处理，即使其能够接受一个可空的（nullable）类型，这为绑定器提供一个可选方案。这样，可以让模型绑定器在调用动作时，可以选择将动作方法参数设置为null。具体调整以及效果如下：

…

public ActionResult Index(**int?** **id**)

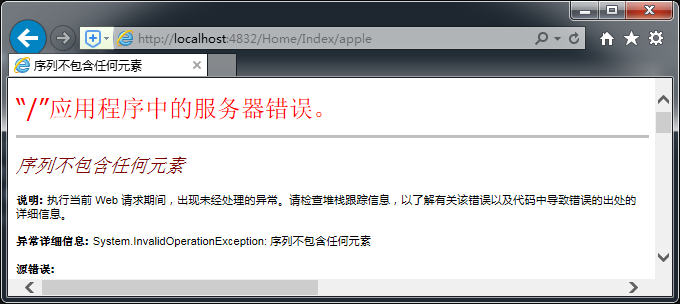
{

Person dataItem = personData.Where(p => p.PersonId == id).First();

return View(dataItem);

}

…



从上图可看出，问题已经发生变化了，模型绑定器可以使用null值作为Index方法的id参数值了，只是动作方法中的代码未对null值进行处理。

下面是通过使用为参数设置默认值的方式解决了这一问题：

…

public ActionResult Index(**int id = 1**)

{

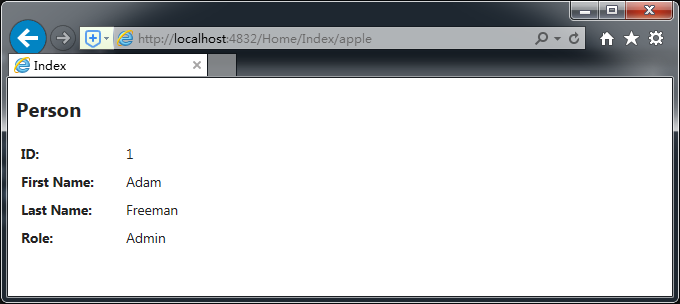
Person dataItem = personData.Where(p => p.PersonId == id).First();

return View(dataItem);

}

…

结果如下：



到这里，我们已经做了初步的问题处理，但是，还没有对各种情况进行检查，如路由中给出-1或500等这样在代码中不存在的id值，即超出了代码处理的范围的值，这是依然会存在一些问题。所以，在实际项目中要多注意动作方法可能接受到的参数值的范围并进行相应的测试，以保证程序的健康运行。

文化敏感解析

DefaultModelBinder类对来自不同地域的请求数据，会采用相应的设置来执行类型转换。从URL获得的值（路由及查询字符串数据）会采用非文化敏感解析进行转换。但是，从表单数据获取的值，则会考虑文化因素进行转换。

（注：这里的文化指的是计算机上设置不同区域或地域，如中国、美国、德国等。不同文化的差异主要表现为日期和货币的表示方式不同。）

这种情况引起的最普遍的问题与DateTime值有关。通常希望非文化敏感日期采取通用格式yyyy-mm-dd，而表单的日期值是服务器设定的格式。即，如果服务器的文化设置为UK（英国），则希望日期的格式是dd-mm-yyyy；而服务器如果设置为US（美国），则希望其格式为yyyy-mm-dd，尽管这两种情况下都可以接受yyyy-mm-dd格式。

如果日期值不是正确格式，则不会被转换。即，必须确保URL中的所有日期都被表示成通用格式。必须小心处理用户提供的日期值——默认绑定器假定，用户将用服务器的文化设置格式来表示日期，这是具有国际用户的MVC应用程序不愿意遇到的情况。

### 绑定复合类型

复合类型：不能使用TypeConverter类进行转换的类型，反之称之为基元类型或简单类型。

如果动作方法的参数是复合类型，DefaultModelBinder类将用反射来获取public属性集，然后依次逐个绑定。

下面是为演示这一工作机制，在Home控制器中添加的两个新动作方法：

…

public ActionResult CreatePerson()

{

return View(new Person());

}

[HttpPost]

public ActionResult CreatePerson(Person model)

{

return View("Index", model);

}

…

动作方法CreatePerson对应的视图如下：

@model MvcModels.Models.Person

@{

ViewBag.Title = "CreatePerson";

}

<h2>Create Person</h2>

@using (Html.BeginForm())

{

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

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

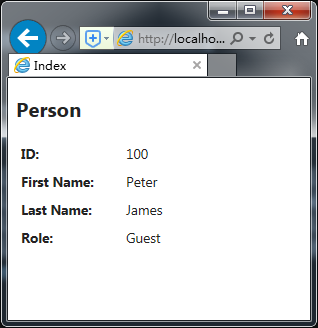
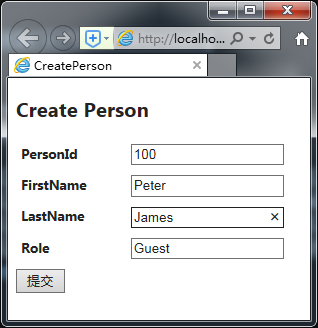
<div>@Html.LabelFor(m => m.LastName)@Html.EditorFor(m => m.LastName)</div>

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

<button type="submit">提交</button>

}

下面是这些修改带来的效果：



在提交表单到CreatePerson动作方法时，形成了一种不同的模型绑定情况。默认模型绑定器发现，动作方法需要一个Person对象，于是会依次处理每个属性。对于每个简单类型的属性，绑定器会试图从请求中找到一个值，如PersonId属性，绑定器会从请求的表单数据中查找到对应的PersonId的数据值。

如果，属性是一个复合类型，那么该过程会针对新类型重复执行——获取该类型的public属性集，而绑定器也会试图找出所有这些属性的值。不同的是这些属性名是嵌套的。如：Person类的HomeAddress属性是Address类型，在为Line1查找值时，模型绑定器查找的是HomeAddress.Line1的值。

**1、创建易于绑定的HTML**

下面是针对复合类型的属性对Create.cshtml试图进行的调整：

@model MvcModels.Models.Person

@{

ViewBag.Title = "CreatePerson";

}

<h2>Create Person</h2>

@using (Html.BeginForm())

{

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

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

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

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

**<div>@Html.LabelFor(m => m.HomeAddress.City)@Html.EditorFor(m => m.HomeAddress.City)</div>**

**<div>@Html.LabelFor(m => m.HomeAddress.Country)@Html.EditorFor(m =>** **m.HomeAddress.Country)</div>**

<button type="submit">提交</button>

}

通过这样的修改，辅助器会自动地设置input元素的name标签属性，以便与默认模型绑定器所使用的格式相匹配，如：

<input **name="HomeAddress.Country"** class="text-box single-line" id="HomeAddress\_Country" type="text" value=""></input>

这样一来，不需要采取任何特别的手段，就可以确保模型绑定器能够为HomeAddress属性创建Address对象。为此演示，我们对/Views/Home/Index.cshtml视图作如下修改：

@model MvcModels.Models.Person

@{

ViewBag.Title = "Index";

}

<h2>Person</h2>

<div><label>ID:</label>@Html.DisplayFor(m => m.PersonId)</div>

<div><label>First Name:</label>@Html.DisplayFor(m => m.FirstName)</div>

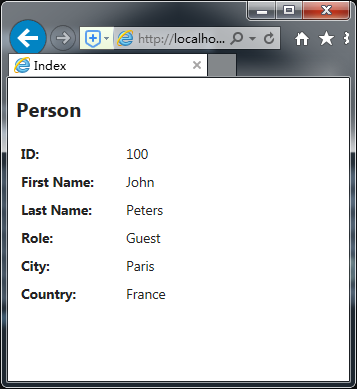
<div><label>Last Name:</label>@Html.DisplayFor(m => m.LastName)</div>

<div><label>Role:</label>@Html.DisplayFor(m => m.Role)</div>

**<div><label>City:</label>@Html.DisplayFor(m => m.HomeAddress.City)</div>**

**<div><label>Country:</label>@Html.DisplayFor(m => m.HomeAddress.Country)</div>**

效果如下：



**2、指定自定义前缀**

有时候也需要生成的HTML与一种类型的对象有关，但希望将其绑定到另一个对象。这意味着是包含的前缀与模型绑定器期望的结构不对应，于是对数据不能作适当的处理。

下面在Models文件夹中添加一个新类AddressSummary.cs予以演示这种情况：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace MvcModels.Models

{

public class AddressSummary

{

public string City { get; set; }

public string Country { get; set; }

}

}

然后，在Home控制器中添加一个新的动作方法，以使用这个新类：

public ActionResult DisplaySummary(AddressSummary summary)

{

return View(summary);

}

对应的视图DisplaySummary.cshtml：

@model MvcModels.Models.AddressSummary

@{

ViewBag.Title = "DisplaySummary";

}

<h2>Display Summary</h2>

<div><label>City:</label>@Html.DisplayFor(m => m.City)</div>

<div><label>Country:</label>@Html.DisplayFor(m => m.Country)</div>

为了演示绑定到不同模型类型时的前缀问题，修改/Views/Home/CreatePerson.cshtml文件中对BeginForm辅助器方法的调用，以便将表单回递给新的DisplaySummary动作方法：

@model MvcModels.Models.Person

@{

ViewBag.Title = "CreatePerson";

}

<h2>Create Person</h2>

**@using (Html.BeginForm("DisplaySummary", "Home"))**

{

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

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

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

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

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

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

<button type="submit">提交</button>

}

现在启动程序，并导航至/Home/CreatePerson就可以看到了。提交表单后，为City和Country属性输入的值并未显示在DisplaySummary视图生成的HTML中。原因是表单中的name属性具有HomeAddress前缀，这不是模型绑定器在试图绑定AddressSummary类型时要查找的前缀。要解决这个问题，只要对动作方法的参数运用Bind注解属性即可，目的是用它来告诉绑定器应该查找哪一个前缀，如：

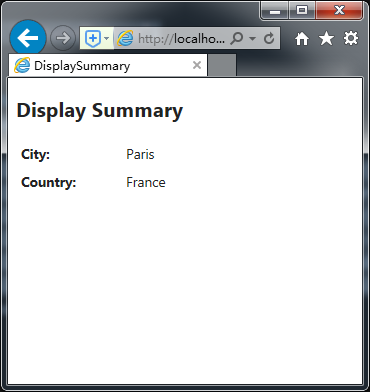
public ActionResult DisplaySummary(**[Bind(Prefix = "HomeAddress")]**AddressSummary summary)

{

return View(summary);

}

这样一来，在装配AddressSummary对象的属性时，模型绑定器会查找请求中的HomeAddress.City和HomeAddress. Country的数据值。在该例子中，显示了Person对象各属性的编辑器，但在提交表单数据时，用模型绑定器创建的却是AddressSummary类的实例，如下图所示。这看起来像是对一个简单的问题太长的设置，但这种绑定到不同类型对象的需求却出奇地普遍，且在实际项目中很可能会需要使用这种技术。



**3、有选择地绑定属性**

在实际项目中，模型类的一些属性可能是敏感的，这时需要采取一些措施隐藏这些属性。如：隐藏属性显示、阻止该属性出现在发送给浏览器的HTML中，或简单地不在视图中添加该属性的编辑器等。

然而，恶意的用户可以在递交表单数据时，简单地编辑发送给服务器的表单数据，然后（从返回数据中）挑出对他们有用的Country属性的值。此时真正要做的事是告诉模型绑定器不要绑定请求中的Country属性值，这可以通过在动作方法参数上使用Bind注解属性来实现。下面这段代码演示了如何在Home控制器的DisplaySummary动作方法中使用这一属性，以防止用户为Country属性提供值：

public ActionResult DisplaySummary([Bind(Prefix = "HomeAddress", **Exclude = "Country"**)]AddressSummary summary)

{

return View(summary);

}

Exclude属性可以将一些属性排除在模型绑定过程之外。导航至/Home/CreatePerson，输入一些数据，就可以看到效果了（将看到不会显示Country属性的值了）。还有一种做法，就是使用Include属性，用以指定只应该在模型绑定中绑定的属性，而忽略其他属性。

当Bind注解属性被用于一个动作方法参数时，它只会影响动作方法所绑定的类的实例；其他动作方法仍然会尝试绑定该参数类型所定义的所有属性。如果需要影响更加广泛，可以将Bind注解属性运用到模型类本身，如：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MvcModels.Models

{

**[Bind(Include = "City")]**

public class AddressSummary

{

public string City { get; set; }

public string Country { get; set; }

}

}

注意：

* 当Bind注解属性运用于模型类，同时也用于动作方法参数时，只有这两处注解属性都未排除的模型属性，才会被包含在绑定过程中。
* Bind注解属性同时用于模型类和动作方法参数时，模型类的绑定优先于动作方法参数的绑定。比如，在模型类上排除了Country，但在动作方法参数中包含了Country，该属性仍然是被排除的。

### 绑定到数组与集合

1. 绑定到数组

为了演示这一特性，需要在Home控制器中添加一个新的方法Names：

public ActionResult Names(string[] names)

{

// 必须检查是否为空，且参数的默认值只能是常数或文字值

names = names ?? new string[0];

return View(names);

}

创建用例显示数组绑定的视图文件，/Views/Home/Names.cshtml：

@model string[]

@{

ViewBag.Title = "Names";

}

<h2>Names</h2>

@if (Model.Length == 0)

{

using (Html.BeginForm())

{

for (int i = 0; i < 3; i++)

{

<div><label>@(i + 1):</label>@Html.TextBox("names")</div>

}

<button type="submit">提交</button>

}

}

else

{

foreach (string str in Model)

{

<p>@str</p>

}

@Html.ActionLink("返回", "Names")

}

该视图根据视图模型的数据项数显示不同的内容。如果没有数据项，则显示一个表单，其中有三个input元素：

<form action="/Home/Names" method="post">

<div><label>1:</label><input id="names" **name="names"** type="text" value="" /></div>

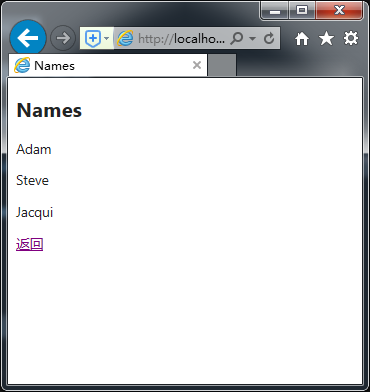
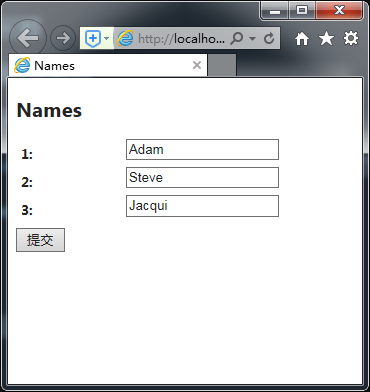
<div><label>2:</label><input id="names" **name="names"** type="text" value="" /></div>

<div><label>3:</label><input id="names" **name="names"** type="text" value="" /></div>

<button type="submit">提交</button>

</form>

递交该表单时，默认的模型绑定器将能够查找与动作方法中的字符串数组同名的数据项。就该示例，会将所有input元素的内容聚集到一起，如下图：



1. 绑定到集合

对于.NET的集合类一样可以绑定。如：

public ActionResult Names(IList<string> names)

{

// 必须检查是否为空，且参数的默认值只能是常数或文字值

names = names ?? **new List<string>()**;

return View(names);

}

注意，这里使用的是IList接口，也就是说，可以不指定其具体的实现类（当然也可以指定，这就看个人喜好了）。下面是对应的视图类的修改：

@model **IList<string>**

@{

ViewBag.Title = "Names";

}

<h2>Names</h2>

@if (Model.**Count** == 0)

{

using (Html.BeginForm())

{

for (int i = 0; i < 3; i++)

{

<div><label>@(i + 1):</label>@Html.TextBox("names")</div>

}

<button type="submit">提交</button>

}

}

else

{

foreach (string str in Model)

{

<p>@str</p>

}

@Html.ActionLink("返回", "Names")

}

Names动作方法的功能并没有发生变化，只是现在使用的是集合类而不是数组。

**3、绑定到自定义模型类型集合**

我们还可以将一些单个的数据属性绑定到一个自定义类型的数组，如上面的AddressSummary模型类。下面是为控制器添加的新的动作方法：

public ActionResult Address(IList<AddressSummary> address)

{

address = address ?? new List<AddressSummary>();

return View(address);

}

对应的视图文件是/Views/Home/Address.cshtml，内容如下：

@using MvcModels.Models

@model IList<AddressSummary>

@{

ViewBag.Title = "Address";

}

<h2>Address</h2>

@if (Model.Count() == 0)

{

using (Html.BeginForm())

{

for (int i = 0; i < 3; i++)

{

<fieldset>

<legend>Address @(i + 1)</legend>

<div><label>City:</label>@Html.Editor("[" + i + "].City")</div>

<div><label>Country:</label>@Html.Editor("[" + i + "].Country")</div>

</fieldset>

}

<button type="submit">提交</button>

}

}

else

{

foreach (AddressSummary address in Model)

{

<p>@address.City,@address.Country</p>

}

@Html.ActionLink("返回", "Address")

}

如果模型集合中无数据项，该视图渲染的结果如下：

<form action="/Home/Address" method="post">

<fieldset>

<legend>Address 1</legend>

<div>

<label>City:</label>

<input class="text-box single-line" **name="[0].City"** type="text" value="" /></div>

<div>

<label>Country:</label>

<input class="text-box single-line" **name="[0].Country"** type="text" value="" /></div>

</fieldset>

<fieldset>

<legend>Address 2</legend>

<div>

<label>City:</label>

<input class="text-box single-line" **name="[1].City"** type="text" value="" /></div>

<div>

<label>Country:</label>

<input class="text-box single-line" **name="[1].Country"** type="text" value="" /></div>

</fieldset>

<fieldset>

<legend>Address 3</legend>

<div>

<label>City:</label>

<input class="text-box single-line" **name="[2].City"** type="text" value="" /></div>

<div>

<label>Country:</label>

<input class="text-box single-line" **name="[2].Country"** type="text" value="" /></div>

</fieldset>

<button type="submit">提交</button>

</form>

在递交这个表单时，默认模型绑定器知道它需要创建一个AddressSummary对象集合，并用name标签属性中的数组索引前缀获取对象的属性值。以[0]为前缀的那些属性表示第一个AddressSummary对象，以[1]为前缀的那些属性表示第二个AddressSummary对象，依次类推。

Address视图为三个这样的索引对象定义了input元素，并在模型集合含有数据项时显示它们。在能正常演示之前，还需要从AddressSummary模型类中删除Bind注解属性，否则，模型绑定器会忽略Country属性：

// 该注解属性已被注掉

//[Bind(Include = "City")]

public class AddressSummary

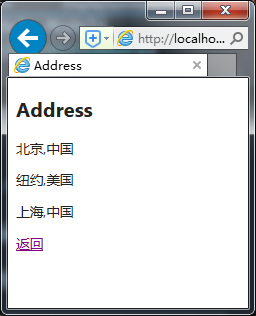
{

public string City { get; set; }

public string Country { get; set; }

}

启动程序后，输入一些数据，并提交后模型绑定器会找到并处理被索引的数据值，并用它们创建AddressSummary对象的集合，然后将该集合回递给视图，并显示结果，效果如图：



## 手工调用模型绑定

当动作方法定义了参数，模型绑定过程将是自动执行的。但这一过程也可以自行控制。下面就通过手动调用，来进一步明确地控制如何实例化模型对象、从何处获取数据，以及如何处理数据解析错误等。

首先，将Home控制中的Address动作方法修改为手工调用绑定过程：

public ActionResult Address()

{

IList<AddressSummary> address = new List<AddressSummary>();

UpdateModel(address);

return View(address);

}

上面代码中的UpdateModel方法以上一条语句定义的模型对象为参数，并试图用标准的绑定过程来获取其public属性的值。

当手工调用绑定过程时，可以将绑定过程限制到一个单一的数据源。默认情况下，绑定器会查看四个地方：表单数据、路由数据、查询字符串和上传文件。下面演示了如何将绑定器限制到表单数据：

public ActionResult Address()

{

IList<AddressSummary> address = new List<AddressSummary>();

UpdateModel(address, **new FormValueProvider(ControllerContext)**);

return View(address);

}

UpdateModel这一重载版本以IValueProvider接口的一个实现为参数，该实现也成为绑定过程的唯一数据。四个默认数据位置中的每一个都是由一个IValueProvider实现表示的，如下表：

|  |  |  |
| --- | --- | --- |
| 源 | IValueProvider实现 | 备注 |
| Request.Form | FormValueProvider | 都以ControllerContext为构造器参数，这是Controller类的一个属性 |
| RouteData.Values | RouteDataValueProvider |
| Request.QueryString | QueryStringValueProvider |
| Request.Files | HttpFileCollectionValueProvider |

限制数据源最常用的方式是只查找表单值。可以用一个更优美的绑定技巧，而不必创建FormValueProvider实例：

public ActionResult Address(**FormCollection formData**)

{

IList<AddressSummary> address = new List<AddressSummary>();

UpdateModel(address, **formData**);

return View(address);

}

FormCollection类也实现了IValueProvider接口，而且，如果把该动作方法定义成以这个类型为参数，那么绑定器将提供一个能够直接传递给UpdateModel方法的对象。

UpdateModel方法的一些其他重载版本允许指定一个搜索前缀，并指定绑定过程中应当包含哪些模型属性。

**处理绑定错误**

在实际项目中，难免会有些无法绑定的值，所以，当明确地调用模型绑定时，需要负责处理诸如此类的错误。模型绑定器会通过抛出InvalidOperationException异常来表示绑定错误，其中细节通过ModelState特性进行检查，所以，在使用UpdateModel方法时，必须做好捕捉该异常的准备，并用ModelState将错误消息展示给用户：

public ActionResult Address(FormCollection formData)

{

IList<AddressSummary> address = new List<AddressSummary>();

**try**

**{**

UpdateModel(address, formData);

**}**

**catch (InvalidOperationException ex)**

**{**

**// 给用户提供反馈**

**}**

return View(address);

}

另一个办法是使用TryUpdateModel方法，如果绑定成功将返回true，否则返回false，如：

public ActionResult Address(FormCollection formData)

{

IList<AddressSummary> address = new List<AddressSummary>();

**if (TryUpdateModel**(address, formData)**)**

**{**

**// 正常处理**

**}**

**else**

**{**

**// 给用户提供反馈**

**}**

return View(address);

}

使用TryUpdateModel方法的好处是不需要我们单独捕捉异常了。但这在模型绑定过程方面，与UpdateModel方法没有功能上的差异。

提示：当自动调用模型绑定时，绑定错误不会发出异常信号。因此，必须通过ModelState.IsValid属性来检查结果。

## 定制模型绑定系统

还有一种绑定方式，就是自定义模型绑定系统来实现绑定系统的定制。

### 创建自定义的值提供器

创建一个自定义的值提供器就可以将自己的数据源添加到模型绑定过程了。值提供器需要实现IValueProvider接口：

using System;

namespace System.Web.Mvc

{

public interface IValueProvider

{

bool ContainsPrefix(string prefix);

ValueProviderResult GetValue(string key);

}

}

ContainsPrefix方法由模型绑定器调用，以确定这个值提供器是否可以解析给定前缀的数据。

GetValue方法返回给定数据键的值，如果无法得到合适的数据时，则返回null。

下面是我们在Infrastructure文件夹中添加的一个自定义的值提供器CountryValueProvider：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Globalization;

namespace MvcModels.Infrastructure

{

public class CountryValueProvider : IValueProvider

{

public bool ContainsPrefix(string prefix)

{

return prefix.ToLower().IndexOf("country") > -1;

}

public ValueProviderResult GetValue(string key)

{

if (ContainsPrefix(key))

{

return new ValueProviderResult("USA", "USA", CultureInfo.InvariantCulture);

}

else

{

return null;

}

}

}

}

上面代码对请求Country属性的值进行响应，并总是返回USA。对所有其他的请求，返回null，来表示无法提供数据。

必须将数据值作为一个ValueProviderResult类来返回。这个类有三个构造参数：

* 第一个参数是与请求键关联的数据项；
* 第二个参数是该数据的安全显示形式；
* 最后一个参数是与该值相关的文化信息（如这里指定了InvariantCulture）。

在有了值提供器后，还需要对其进行注册，这里采用工厂类为其注册的方式，该工厂类（CustomValueProviderFactory.cs）位于Infrastructure文件夹，该类派生于抽象类ValueProviderFactory，具体内容如下：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MvcModels.Infrastructure

{

public class CustomValueProviderFactory : ValueProviderFactory

{

public override IValueProvider GetValueProvider(ControllerContext controllerContext)

{

return new CountryValueProvider();

}

}

}

当模型绑定器要为绑定过程获取值时，会调用这个GetValueProvider方法。上述示例代码中简单地创建并返回了CountryValueProvider类的一个实例，但可以使用controllerContext参数提供的数据创建不同的值提供器，以便对不同类型的请求进行响应。

剩下的就是在Global.asax的Application\_Start方法中注册这个工厂类了：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Optimization;

using System.Web.Routing;

**using MvcModels.Infrastructure;**

namespace MvcModels

{

// 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，

// 请访问 http://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

**ValueProviderFactories.Factories.Insert(0,new CustomValueProviderFactory());**

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

}

}

这里注册的时候将自定义值提供器注册工厂类放在了第一个位置，目的是为了能够让自定义值提供器被优先考查。如果是需要将自定义的值提供器作为备选方案，则可以是Add方法把工厂类追加到集合的末尾，如：

ValueProviderFactories.Factories.**Add**(0,new CustomValueProviderFactory())

现在可以测试我们定义的这个值提供器了，但是，在此之前仍需修改一处，那就是Address动作方法，以使模型绑定器为模型属性值不只考察表单数据，如下：

public ActionResult Address()

{

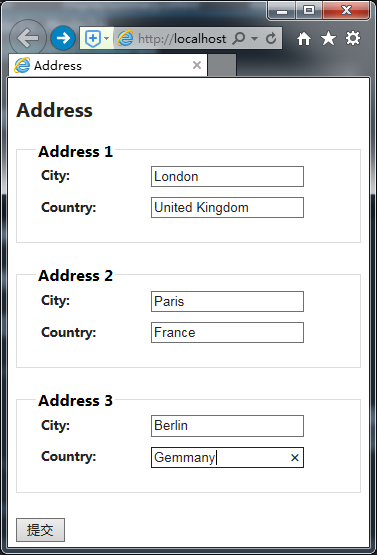
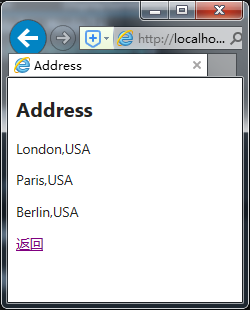
IList<AddressSummary> address = new List<AddressSummary>();

UpdateModel(address);

return View(address);

}

效果如下图：

### 创建自定义模型绑定器

通过创建一个自定义模型绑定器可以覆盖默认绑定器的行为。自定义模型绑定器需要实现IModelBinder接口。

在Infrastructure文件夹中添加自定义绑定器AddressSummaryBinder.cs：

using MvcModels.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MvcModels.Infrastructure

{

public class AddressSummaryBinder : IModelBinder

{

public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)

{

AddressSummary model = (AddressSummary)bindingContext.Model ?? new AddressSummary();

model.City = GetValue(bindingContext, "City");

model.Country = GetValue(bindingContext, "Country");

return model;

}

private string GetValue(ModelBindingContext context, string name)

{

name = (context.ModelName == "" ? "" : context.ModelName + ".") + name;

ValueProviderResult result = context.ValueProvider.GetValue(name);

if (result == null || result.AttemptedValue == "")

{

return "<Not Specified>";

}

else

{

return (string)result.AttemptedValue;

}

}

}

}

当MVC框架需要一个模型绑定器所支持的模型类型的实例时，将调用BindModel方法。建议：一个模型绑定器中可以创建多个类型的支持，但最好在一个绑定器中仅支持一个类型。

**BindModel方法简介：**

**参数：**

* **ControllerContext：**可以用于获取当前请求的细节；
* **ModelBindingContext：**用于提供当前寻找的模型对象的细节，并能访问MVC应用程序中其他模型绑定工具，其中一些最有用的属性如下：
  + Model：如果手工调用了绑定，可返回传递给UpdateModel方法的模型对象；
  + ModelName：返回绑定模型的名称；
  + ModelType：返回被创建模型的类型；
  + ValueProvider：返回能用于从请求中获取数据值的IValueProvider实现。

上述示例功能说明：

1. 当调用BindModel方法时，检测是否已设置了ModelBindingContext的Model属性，如果已设置，则使用当前模型为将要位置生成数据值的对象，否则仅创建一个新的AddressSummary实例；
2. 通过私有的GetValue方法分别获取City和Country的值，然后返回给AddressSummary类型对象相应的属性；
3. 在GetValue方法中使用ValueProviderResult. ValueProvider属性获取的IValueProvider实现，以获取模型对象属性的值。
4. ModelName属性可以指出正在寻找的属性名称是否需要追加一个前缀。比如我们的动作方法在试图创建AddressSummary对象的集合，这就是说各个input元素将具有附带了[0]和[1]等前缀的name属性值。这样，在请求中的值将是[0].City、[0].Country等。
5. 如果无法为某一属性找到值或该属性为空字符串时，便提供一个默认值：<Not Specified>。

### 注册自定义模型绑定器

如果要是该模型绑定器能够正常工作，需要对其进行注册。这可以在Global.asax的Application\_Start方法中完成：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Optimization;

using System.Web.Routing;

using MvcModels.Infrastructure;

using MvcModels.Models;

namespace MvcModels

{

// 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，

// 请访问 http://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

// 此句已被注释

//ValueProviderFactories.Factories.Insert(0, new CustomValueProviderFactory());

**ModelBinders.Binders.Add(typeof(AddressSummary), new** **AddressSummaryBinder());**

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

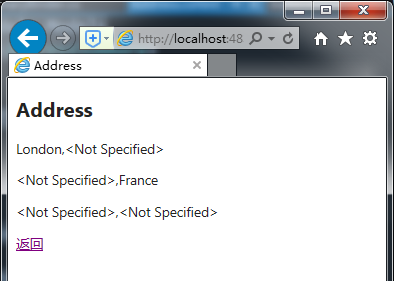
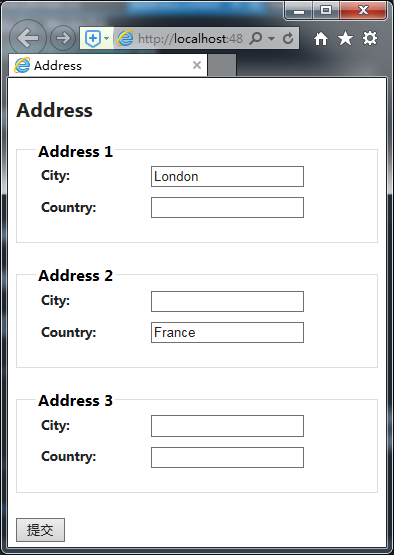
BundleConfig.RegisterBundles(BundleTable.Bundles);

}

}

}

下面导航至http://localhost:4832/Home/Address，并只填充部分内容，以便可以对这一自定义模型绑定器进行测试，表单被递交时，自定义模型绑定器会对其所有未输入值的属性使用<Not Specified>默认值：



提示：也可以通过在模型类上使用ModelBinder注解属性进行修饰，也可以指定自定义模型绑定器。