# **一、 ModelBinder**

用于进行Model绑定的ModelBinder对象实现了接口[IModelBinder](http://msdn2.microsoft.com/dd505004.aspx" \o "IModelBinder Interface)。如下面的代码片断所示，IModelBinder接口具有唯一的BindModel方法用于实现针对某个参数的绑定操作，该方法的返回值表示的就是最终作为参数值的对象。

1: public interface IModelBinder

2: {

3: object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext);

4: }

IModelBinder的BindModel方法接受两个参数，一个是表示当前的Controller上下文，另一个是表示针对当前Model绑定的上下文，通过类型[ModelBindingContext](http://msdn2.microsoft.com/hh195362.aspx" \o "ModelBindingContext Class)表示。在Controller初始化的时候，Controller上下文已经被创建出来，所以我们只要能够针对当前的Model绑定创建相应的ModelBindingContext，我们就能使用基于某个参数的ModelBinder得到对应的参数值。关于ModelBindingContext的创建我们会在后续部分进行的单独介绍，我们先来介绍一下ModelBinder的提供机制。

# **二、CustomModelBinderAttribute与ModelBinderAttribute**

如果针对某个参数的ParameterDescriptor具有相应的ModelBinder，那么它会被优先选择用于针对该参数的Model绑定，那么ParameterDescriptor的ModelBinder是如何来提供的呢？这是实际上设置一个具有如下定义的[CustomModelBinderAttribute](http://msdn2.microsoft.com/dd492121.aspx" \o "CustomModelBinderAttribute Class)特性。抽象类CustomModelBinderAttribute定义了唯一的抽象方法GetBinder用于获取相应的ModelBinder对象。

1: [AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Interface | AttributeTargets.Enum | AttributeTargets.Struct

2: | AttributeTargets.Class, AllowMultiple=false, Inherited=false)]

3: public abstract class CustomModelBinderAttribute : Attribute

4: {

5: public abstract IModelBinder GetBinder();

6: }

在ASP.NET MVC应用编程接口中，CustomModelBinderAttribute具有一个具有如下定义的唯一继承类型[ModelBinderAttribute](http://msdn2.microsoft.com/dd492740.aspx" \o "ModelBinderAttribute Class)。我们可以通过应用ModelBinderAttribute特性动态地选择用于Model绑定的ModelBinder类型。

1: [AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Interface |

2: AttributeTargets.Enum | AttributeTargets.Struct | AttributeTargets.Class,

3: AllowMultiple=false, Inherited=false)]

4: public sealed class ModelBinderAttribute : CustomModelBinderAttribute

5: {

6: public ModelBinderAttribute(Type binderType);

7: public override IModelBinder GetBinder();

8:

9: public Type BinderType { [CompilerGenerated] get; }

10: }

从应用在ModelBinderAttribute类型上的AttributeUsageAttribute定义可以看出该特性不仅仅可以应用在参数上，也可以应用类型（接口、枚举、结构和类）上，这意味我们既可以将它应用在Action方法的某个参数上，也可以将它应用在某个参数的类型上。但是ParameterDescriptor只会解析应用在参数上的特性，所以应用在参数对象类型上的ModelBinderAttribute对它是无效的。

为了演示ModelBinderAttribute特性对ParameterDescriptor的影响，我们来进行一个简单的实例演示。在一个通过Visual Studio的ASP.NET MVC项目模板创建的空Web应用中定义了如下几个类型，其中FooModelBinder和BarModelBinder是显现了IModelBinder的自定义ModelBinder类型，而Foo、Bar和Baz是三个将被作为Action方法参数的数据类型，其中Bar上应用了ModelBinderAttribute特性并将ModelBinder类型设置为BarModelBinder。

1: public class FooModelBinder : IModelBinder

2: {

3: public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)

4: {

5: throw new NotImplementedException();

6: }

7: }

8: public class BarModelBinder : IModelBinder

9: {

10: public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)

11: {

12: throw new NotImplementedException();

13: }

14: }

15:

16: public class Foo { }

17: [ModelBinder(typeof(BarModelBinder))]

18: public class Bar { }

19: public class Baz { }

然后再创建的默认HomeController中定义如下两个Action方法。DoSomething方法具有三个参数，类型分别是Foo、Bar和Baz，在第一个参数上应用了ModelBinderAttribute特性并将ModelBinder类型设置为FooModelBinder。

1: public class HomeController : Controller

2: {

3: public void Index()

4: {

5: ControllerDescriptor controllerDescriptor = new ReflectedControllerDescriptor(typeof(HomeController));

6: ActionDescriptor actionDescriptor = controllerDescriptor.FindAction(ControllerContext, "DoSomething");

7: IModelBinder foo = actionDescriptor.GetParameters().First(p => p.ParameterName == "foo").BindingInfo.Binder;

8: IModelBinder bar = actionDescriptor.GetParameters().First(p => p.ParameterName == "bar").BindingInfo.Binder;

9: IModelBinder baz = actionDescriptor.GetParameters().First(p => p.ParameterName == "baz").BindingInfo.Binder;

10:

11: Response.Write(string.Format("foo: {0}<br/>", null == foo? "N/A": foo.GetType().Name));

12: Response.Write(string.Format("bar: {0}<br/>", null == bar ? "N/A" : bar.GetType().Name));

13: Response.Write(string.Format("baz: {0}<br/>", null == baz ? "N/A" : baz.GetType().Name));

14: }

15:

16: public void DoSomething([ModelBinder(typeof(FooModelBinder))]Foo foo,Bar bar, Bar baz)

17: {}

18: }

在默认的Action方法Index中，我们针对HomeController类型的ReflectedControllerDescriptor对象并获取到用于描述Action方法DoSomething的ActionDescriptor对象。最后我们通过该ActionDescriptor对象得到用于描述其三个参数的ParameterDescriptor对象，并将其ModelBinder类西国内呈现出来。当我们运行该程序的时候，会在浏览器中产生如下的输出结果，可以看出对于分别应用在参数和参数类型上的ModelBinderAttribute特性，只有前者会对ParameterDescriptor的ModelBinder的选择造成影响。

1: foo: FooModelBinder

2: bar: N/A

3: baz: N/A

# **三、ModelBinders**

如下面的代码片断所示，ModelBinders具有一个静态只读属性Binders，表示当前注册ModelBinder列表，其类型为[ModelBinderDictionary](http://msdn2.microsoft.com/dd492249.aspx" \o "ModelBinderDictionary Class)。

1: public static class ModelBinders

2: {

3: public static ModelBinderDictionary Binders { get; }

4: }

5:

6: public class ModelBinderDictionary :

7: IDictionary<Type, IModelBinder>,

8: ICollection<KeyValuePair<Type, IModelBinder>>,

9: IEnumerable<KeyValuePair<Type, IModelBinder>>,

10: IEnumerable

11: {

12: //其他成员

13: public IModelBinder GetBinder(Type modelType);

14: public virtual IModelBinder GetBinder(Type modelType, bool fallbackToDefault);

15: }

ModelBinderDictionary是一个以数据类型（Model类型）为Key，ModelBinder对象为Value的字典，即它定义了针对某种数据类型的ModelBinder。ModelBinderDictionary具有两个GetBinder方法重载用于获取针对某个数据类型的ModelBinder，布尔类型的参数fallbackToDefault表示在数据类型不存在的时候是否采用默认的ModelBinder，基于默认ModelBinder的后备机制会在第一个GetBinder方法重载中采用。在这里默认ModelBinder类型为[DefaultModelBinder](http://msdn2.microsoft.com/dd470193.aspx" \o "DefaultModelBinder Class)。

在为某个参数获取相应的ModelBinder的时候，如果对应的ParameterDescriptor的ModelBinder不存在，则通过ModelBinders的静态属性Binders表示获取到当前注册的ModelBinder列表的ModelBinderDictionary对象，并将参数类型作为参数调用其GetBinder方法获取相应ModelBinder对象。

我们根据ModelBinder的提供机制对上面演示的实例进行相应的修改。我们在HomeConroller中添加了一个CheckModelBinder方法，三个参数分别表示用于描述相应Action方法的ActionDescriptor对象、参数名称和类型。在该方法中我们先获取到用于描述制定参数的ParameterDescriptor对象，如果它具有相应的ModelBinder，则将具体的类型名称输出，否则输出通过ModelBinders获取的针对参数类型的ModelBinder类型。

1: public class HomeController : Controller

2: {

3: //其他成员

4: public void Index()

5: {

6: ControllerDescriptor controllerDescriptor = new ReflectedControllerDescriptor(typeof(HomeController));

7: ActionDescriptor actionDescriptor = controllerDescriptor.FindAction(ControllerContext, "DoSomething");

8:

9: CheckModelBinder(actionDescriptor, "foo", typeof(Foo));

10: CheckModelBinder(actionDescriptor, "bar", typeof(Bar));

11: CheckModelBinder(actionDescriptor, "baz", typeof(Baz));

12: }

13:

14: private void CheckModelBinder(ActionDescriptor actionDescriptor, string parameterName, Type modelType)

15: {

16: ParameterDescriptor parameterDescriptor = actionDescriptor.GetParameters().First(p=>p.ParameterName == parameterName);

17: IModelBinder modelBinder = parameterDescriptor.BindingInfo.Binder ?? ModelBinders.Binders.GetBinder(modelType);

18: Response.Write(string.Format("{0}: {1}<br/>", parameterName, null == modelBinder ? "N/A" : modelBinder.GetType().Name));

19: }

20: }

在Index方法中，我们调用CheckModelBinder方法将Action方法DoSomething的三个参数对应的ModelBinder类型呈现出来。当我们运行该程序的时候，在浏览器上会得到如下的输出结果，应用在类型Bar上的BarModelBinder会用于针对参数bar的Model绑定，而参数baz则会使用默认的DefaultModelBinder。

1: foo: FooModelBinder

2: bar: BarModelBinder

3: baz: DefaultModelBinder

对于上面的这个例子，由于数据类型Baz没有关联ModelBinder注册到通过ModelBinders的静态属性Binders表示的全局ModelBinder列表中，所以才导致DoSomething的baz参数采用默认的DefaultModelBinder。如果我们实现针对数据类型Baz进行了相应的ModelBinder注册，那么被注册的ModelBinder将会自动用于该类型参数的Model绑定。同样是针对上面演示的这个实例，我们定义了如下一个实现了IModelBinder的BazModelBinder。

1: public class BazModelBinder : IModelBinder

2: {

3: public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)

4: {

5: throw new NotImplementedException();

6: }

7: }

现在我们希望使用这个BazModelBinder用于针对所有类型为Bar的参数的Model绑定，那么我们可以通过Global.asax在应用启动的时候进行如下的ModelBinder注册。

1: public class MvcApplication : System.Web.HttpApplication

2: {

3: //其他成员

4: protected void Application\_Start()

5: {

6: //其他操作

7: ModelBinders.Binders.Add(typeof(Baz), new BazModelBinder());

8: }

9: }

再次运行我们的程序，在浏览器中会得到如下的输出结果，从中可以清楚地看出我们注册的BazModelBinder并用于baz参数的Model绑定。

1: foo: FooModelBinder

2: bar: BarModelBinder

3: baz: BazModelBinder

# **四、ModelBinderProvider**

ASP.NET MVC的Model绑定系统还涉及到另一个重要的组件ModelBinderProvider。顾名思义，ModelBinderProvider专门用于提供相应的ModelBinder对象，它们均实现了[IModelBinderProvider](http://msdn2.microsoft.com/gg402095.aspx" \o "IModelBinderProvider Interface)面的代码片断所示，IModelBinderProvider接口定义了唯一的GetBinder方法用于根据数据类型获取相应的ModelBinder对象。不过在ASP.NET MVC现有的应用编程接口中并没有定义任何一个实现该接口的ModelBinderProvider类型。

1: public interface IModelBinderProvider

2: {

3: IModelBinder GetBinder(Type modelType);

4: }

我们可以利用[ModelBinderProviders](http://msdn2.microsoft.com/gg416533.aspx" \o "ModelBinderProviders Class)为应用注册一组ModelBinderProvider对象为某个数据类型提供相应的ModelBinder。如下面的代码片断所示，静态类型ModelBinderProviders具有一个静态只读属性BinderProviders，其类型[ModelBinderProviderCollection](http://msdn2.microsoft.com/gg402083.aspx" \o "ModelBinderProviderCollection Class)实际上是一个型ModelBinderProvider的集合，该集合表示针对当前应用的ModelBinderProvider列表。

1: public static class ModelBinderProviders

2: {

3: public static ModelBinderProviderCollection BinderProviders { get; }

4: }

5:

6: public sealed class ModelBinderProviderCollection : Collection<IModelBinderProvider>

7: {

8: //省略成员

9: }

通过ModelBinderProviders的静态属性BinderProviders表示的ModelBinderProvider列表最终被ModelBinderDictionary使用。如下面的代码片断所示，ModelBinderDictionary除了具有一个表示基于数据类型的ModelBinder字典（\_innerDictionary字段）和一个默认ModelBinder（\_defaultBinder）之外，还具有一个ModelBinderProvider列表（\_modelBinderProviders字段）。

1: public class ModelBinderDictionary

2: {

3: //其他成员

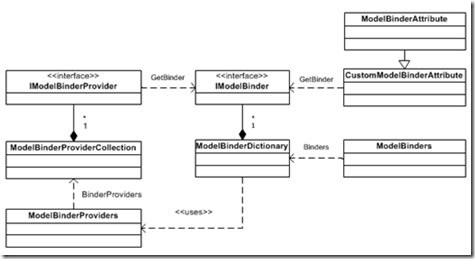
4: private IModelBinder \_defaultBinder;

5: private readonly Dictionary<Type, IModelBinder> \_innerDictionary;

6: private ModelBinderProviderCollection \_modelBinderProviders;

7: }

当ModelBinderDictionary被创建的时候，通过ModelBinderProviders的静态属性BinderProviders表示的ModelBinderProvider列表会用于初始化\_modelBinderProviders字段。围绕着ModelBinder的Model绑定系统中的核心组件之间的关系基本上可以通过下图所示的UML来表示。

[](http://images.cnblogs.com/cnblogs_com/artech/201205/201205210812043833.png)

当我们调用GetBinder或者指定数据类型对应的ModelBinder时，\_innerDictionary字段表示的ModelBinder字典会被优先选择。如果数据类型在该字典中找不到，则选择使用通过\_modelBinderProviders字段表示的ModelBinderProvider列表进行ModelBinder的提供。只有在两种ModelBinder提供方式均失败的情况下才会选择通过\_innerDictionary字段表示的默认ModelBinder。

前面三种方式的ModelBinder提供机制我们已经通过实例演示过了，现在我们来演示基于自定义ModelBinderProvider的ModelBinder提供机制。在前面的例子中我们为Foo、Bar和Baz这三种数据类型创建了相应的ModelBinder（FooModelBinder、BarModelBinder和BazModelBinder），现在我们创建如下一个自定义的ModelBinderProvider将两种（数据类型和ModelBinder对象）进行关联。

1: public class MyModelBinderProvider : IModelBinderProvider

2: {

3: public IModelBinder GetBinder(Type modelType)

4: {

5: if (modelType == typeof(Foo))

6: {

7: return new FooModelBinder();

8: }

9: if (modelType == typeof(Bar))

10: {

11: return new BazModelBinder();

12: }

13: if (modelType == typeof(Baz))

14: {

15: return new BazModelBinder();

16: }

17: return null;

18: }

19: }

现在我们需要通过利用Global.asax通过如下的方式在应用启动时将一个我们自定义的MyModelBinderProvider注册到通过ModelBinderProviders的静态属性BinderProviders表示的ModelBinderProvider列表中。

1: public class MvcApplication : System.Web.HttpApplication

2: {

3: //其他成员

4: protected void Application\_Start()

5: {

6: //其他操作

7: ModelBinderProviders.BinderProviders.Add(new MyModelBinderProvider());

8: }

9: }

由于MyModelBinderProvider实现了针对Foo、Bar和Baz三种数据类型的ModelBinder的提供，所以我们可以将应用在Action方法参数和数据类型上的ModelBinderAttribute特性删除。再次运行我们的程序，会在浏览器中得到如下的输出结果，从中可以看到DoSomething方法的三个参数此时采用了我们期望的ModelBinder类型。

1: foo: FooModelBinder

2: bar: BarModelBinder

3: baz: BazModelBinder