# Toolkit5.5数据读写

## 简介

在开发中，Xml，Json等数据格式越来越重要。而传统分析Xml和Json的方式要将相应的格式解析为对应.net的类数据，需要大量的编码。当然，也有很多第三方的类，对某种数据类型的读写做了相关的优化，比如newtonsoft的json解析就比较方便。不过，迄今为止，还没有一种类库可以支持多种数据格式的解析和转化。

Toolkit平台通过长期的开发，建立了一套有效的数据读取体系。通过声明一系列的Attribute在类型上进行标注，然后通过插件的方式，陆续建立了对象和Xml，对象和Json，对象和QueryString，对象和DataRow，对象和XElement等结构的相互转化。随着新的需求或者新的数据方式的提出，可以开发出新的插件，满足相应的需求。

同时对象可以作为一个桥梁，可以将任意两种数据格式进行相互转化。比如将Xml转化成Json，将QueryString转化成DataRow。

例如，我们声明以下的类：

public class SampleConfig

{

[SimpleAttribute]

public int Height { get; private set; }

[SimpleAttribute]

public bool Check { get; private set; }

[SimpleElement]

public string Name { get; private set; }

}

对应的，如果是Xml，应该是如下结构：

<tk:Toolkit Height="100" Check="true" xmlns:tk="http://www.qdocuments.net">

<Name>Hello</Name>

</tk:Toolkit>

那么，读取这段Xml的代码很简单，就如下两行：

SampleConfig config = new SampleConfig();

config.ReadXml(xml);

如果是Json，应该是如下结构：

{"Height": 100, "Check": true, "Name": "Hello"}

那么，读取这段Json的代码很简单，就如下两行：

SampleConfig config = new SampleConfig();

config.ReadJson(json);

如果是QueryString，应该是如下结构

Height=100&Check=true&Name=Hello

那么，读取这段QueryString的代码很简单，就如下两行：

SampleConfig config = new SampleConfig();

config.ReadQueryString(query);

相应的，还有WriteXml，WriteJson，WriteQueryString的扩展方法，可以将对象直接写成对应的格式。因此通过上述结构，亦可以完成两种数据格式的相互转换。

和原先Toolkit5的版本相比，最新的版本还添加了直接读取List和Dictionary两种类型的特性。

比如如下代码：

string json = "[1, 3, 5]";

List<int> list = new List<int>();

list.ReadJson(json);

string json = "{'Hello':100, 'World':'true'}";

Dictionary<string, string> dict = new Dictionary<string, string>();

dict.ReadJson(json);

## 初级篇

### 概述

在Toolkit中，为了满足对象和数据格式的相互转换，定义了一系列的Attribute，不同的Attribute有不同的场景。通过在类上的Property上进行标注，来达到自动和对应数据格式的数据关联。由于这些Attribute最早是来源于Xml的分析，所以讲述的重点以Xml格式为主。

以下是Toolkit中定义的Attribute：

* SimpleAttribute
* SimpleElement
* ObjectElement
* TextContent
* Dictionary
* ObjectDictionary
* TagElement
* ComplexContent
* SimpleComplexContent
* DynamicElement
* DynamicDictionary

其中后四种Attribute属于比较灵活，且比较难掌握的概念，将不在本章中阐述。我将在中级篇中详细阐述这四种Attribute的定义和使用方法。

在简介中，我们已经看到了SimpleAttribute和SimpleElement的使用方法。理想情况下，我们不需要给这些Attribute附加任何属性。但这只是理想状况，实际情况下，可能会出现这样那样的问题，因此每个Attribute都有一堆属性来应付这些情况的发生。下面先介绍一些通用的属性。

### 通用属性

Attribute的声明，采用了面向对象的继承结构。有两个基本的Attribute，一个是BaseObjectAttribute，一个是NamedAttribute。其中BaseObjectAttribute是所有Attribute的基类，NamedAttribute从BaseObjectAttribute继承，是所有带有名称的Attribute的基类。

BaseObjectAttribute有以下几个属性

* DefaultValue
* ObjectType
* Required

NamedAttribute有以下几个属性

* LocalName
* NamingRule
* NamespaceType
* NamespaceUri

下面来逐一介绍这些属性的含义

#### DefaultValue

DefaultValue可以设定属性的默认值，当设定默认值后，如果该属性没有出现在Xml中，那么Property将会自动设置为默认值所对应的值。

DefaultValue可以是int，bool，double，string，System.Type，枚举类型。如果不是上述类型，请使用string类型表达。系统会进行适当的类型转换。

#### ObjectType

在Xml，Json等格式中，具体每个标签或属性对应的值都是字符串。而对应的C#类型可能是int，boolean，double或者string等类型。如何将字符串转换成目标类型，也是Toolkit重点解决的一个问题。系统内置了int，double，boolean，long，string，DateTime，Guid，TimeSpan，Encoding，byte[]，string[]，以及任意枚举类型的转换器（如何自定义转换器，这将在高级篇中进行阐述），此外还可以通过转换，使用.net类库中的TypeConverter来进行字符串和类型的相互转换。ObjectType就是指明目标类型，好让Toolkit自动选择合适的类型转换器进行数据类型转换。不过通常情况下，是不需要指明ObjectType的，因为系统默认会把其附着的Property所声明的类型作为其目标类型。而且，即使是List或者Dictionary情况下，如果采用泛型声明，系统也会自动寻找对应的泛型类型，而无须做特别声明。但是如果声明的类型和目标类型不匹配，就需要进行声明。我们少许修改前面例子的代码，如下：

[SimpleAttribute(ObjectType = typeof(bool))]

public object Check { get; private set; }

#### Required

当其为true时，会在读取完毕后检测对应的值是否为null，如果为null，系统将抛错。对于必须要有的节点声明该属性为true，可以有效的在读取阶段查找到相应的配置错误。

#### NamingRule

NamingRule是枚举值，有Camel，Pascal，Upper，Lower，UnderLineLower五种类型，默认是Pascal。它的作用是将Property的名称转换为对应标签的名称。

Camel：Camel名称方式，即首字母小写

Pascal：Pascal名称方式，即首字母大写

Upper：全大写

Lower：全小写

UnderLineLower：将HelloWorld转换为hello\_world，GoodGoodStudy转换为good\_good\_study。

Toolkit中对名称采用严格的大小写敏感匹配方式，在简介中，Property的Height和属性Height关联，是因为默认的NamingRule是Pascal，其名称就会自动转换为Height，刚好和Xml/Json中属性Height相对应。如果NamingRule设置为其他值，那么Property的Height将无法和属性Height相对应，那么其Property的值将只能是0。例：

[SimpleElement(NamingRule = NamingRule.Lower)]

public string OpenId { get; private set; }

#### LocalName

NamingRule提供了Property的名称和最终名字的一种便捷转换方式。但是，这种转换方式，只能适配80%-90%的情况。在一些特殊情况下，如对应元素刚好是C#的关键字，或者使用了TagElement这种特殊Attribute，或者Xml/Json中的名称和对应C#代码的编程规范无法适配。这时，系统将无法将Property的名称和对应元素名称进行关联，这时需要显式指定LocalName的值。LocalName的值具有最高优先等级，一旦指定，NamingRule无论什么值都将无视。例：

[ObjectElement(NamespaceType.Toolkit, IsMultiple = true, LocalName = "GroupSection")]

[TagElement(NamespaceType.Toolkit, LocalName = "ControlGroup")]

public List<GroupSection> ControlGroupList { get; protected set; }

#### NamespaceType

NamspaceType和NamespaceUri仅仅针对Xml这种数据结构提出的，其他数据格式因为无此概念，可以无视它们的配置。

NamespaceType是枚举值，有None，Toolkit和Namespace三种值。默认为None，即Xml中不使用Namespace。如果是Toolkit，NamespaceUri将自动赋值为http://www.qdocuments.net，也就是说，无需指定NamespaceUri的内容。如果有Namespace，且Namespace与Toolkit无关，那么，NamespaceType需要设定为Namespace，并同时指定NamespaceUri为对应的值。

#### NamespaceUri

当NamespaceType设置为Namespace时，需要在此属性中设置对应的内容（通常都是Uri）。

### SimpleAttribute

SimpleAttributeAttribute从NamedAttribute继承，是针对Xml中的Attribute，在Json中任意的简单属性值也都可以看为SimpleAttribute。

如图，红框中的内容都是Attribute。

C:\Users\ADMINI~1\AppData\Local\Temp\1562229515(1).png

SimpleAttribute的属性为：

* UseSourceType
* AutoTrim

#### UseSourceType

该属性是针对Json数据格式的。默认值为false。通常情况下，所有数据在写出时，都是把数据变成字符串然后写出。如果是Json格式，那么所有的值都是加了双引号引用的，即使是int，bool也会加双引号。这可能会引起前端JavaScript对Json的处理产生问题。此时，如果设置该属性为true，那么系统不在把数据值变为字符串，而是尝试使用原始的值来写入Json。那么int，bool这些值将不再会有双引号了。

例如，简介中的例子，如果输出Json，那么格式将是如下：

{"Height":"100","Check":"true","Name":"Hello"}

如果我们将定义变为如下：

public class SampleConfig

{

[SimpleAttribute(UseSourceType = true)]

public int Height { get; private set; }

[SimpleAttribute(UseSourceType = true)]

public bool Check { get; private set; }

[SimpleElement]

public string Name { get; private set; }

}

那么，输出Json后，格式将是

{"Height":100,"Check":true,"Name":"Hello"}

#### AutoTrim

默认值为false。在读取数据时，如果AutoTrim为true，那么会先对读取的数据进行Trim操作，然后才做相应的数据类型转换。为了防止用户在输入数据中无意中在两端录入空格，可以是考虑把该属性设为true，那么将自动过滤这些空格。

### SimpleElement

SimpleElementAttribute从NamedAttribute继承，是针对Xml的Element，但是它约定该元素没有多余的属性，也没有嵌套子元素。在Json中，SimpleElement和SimpleAttribute都可以认为是简单属性值。

如图，红框中的内容是Element

C:\Users\ADMINI~1\AppData\Local\Temp\1562400820(1).png

SimpleElement的属性为：

* IsMultiple
* CollectionType
* Order
* UseCData
* UseSourceType
* AutoTrim

其中UseSourceType和AutoTrim和SimpleAttribute中的含义相同，不再敖述。

#### IsMultiple

IsMultiple默认为false，即xml数据中只有一个元素。在false情况下，SimpleElement的行为和SimpleAttribute没有太大区别。但是xml中的元素是可以重复出现的，一旦重复出现，IsMultiple就应当设置为true。如果IsMultiple设置为true，那么对Property的定义，以及分析行为将出现如下变化：

1. 为了能够包容重复的元素，必须将Property的类型定义为诸如List<T>这样的类型。Toolkit规定，一旦IsMultiple为true，其Property定义的类型必须实现了IList接口。通常情况下，我们都是用List<T>的。但是，只要你实现了IList接口，任何其他非List<T>类型都可以，比如Toolkit提供RegNameList<T>也实现了IList接口，同样可以作为Property的类型。
2. IsMultiple为true后，Property的定义类型将自动映射到CollectionType属性上，如果该类型的定义使用了泛型，即将List的元素类型定义在泛型声明中，那么这个类型声明会自动映射到ObjectType属性上。但是如果类型定义没有使用泛型，那么需要显示指定List的元素类型到ObjectType属性上，否则系统会报错。
3. CollectionType必须有默认的构造函数可以直接实例化，否则，在分析过程中将无法实例化对应类型导致程序出错（List<T>，RegNameList<T>保证满足条件，所以无需担心）
4. 通常情况下，使用IsMultiple=true后，Property的名称都是复数形式，而Xml中可能采用单数描述，这时，可能需要强行指定LocalName。

在Json格式中，如果数据是数组形态，那么应该将IsMultiple设置为true。

后续属性中，如果还存在有IsMultiple属性，其含义与这里描述一致，因此将不在重复。

例：

[SimpleElementAttribute(IsMultiple = true, LocalName = "Score")]

public List<int> Scores { get; private set; }

#### CollectionType

当IsMultiple为true时，可以显式设定集合属性的类型。如果不设定将默认采用Property的类型。如果声明，要求必须和Property声明的类型兼容（如Property声明的是接口类型，就需要在CollectionType中显式指定相应的实例类型）。

后续属性中，如果还存在有CollectionType属性，其含义与这里描述一致，因此将不在重复。

#### Order

Order是针对Xml数据格式的。它可以控制元素的输出顺序。在Xml中，属性没有前后顺序之说，但是元素却有。如果Xml文档有DTD或者Schema对其进行约束时，如果属性没有安装顺序排列，Xml编辑器通常会报错。设置Order属性后，Xml输出将按照Order从小到大依次输出。这个属性只对写有效，对读无效。而且即使不设置这个属性，使用Toolkit输出的Xml，一样可以被再次读取，分析程序无视元素出现的顺序。只有介意Element的输出顺序时，才需要设置此属性。

后续还有一些和Element相关的Attribute都将会出现Order属性，含义都是一样。

#### UseCData

UseCData是针对Xml数据格式的，默认为false。正常情况下，在写Xml时，都会把内容直接写出到Xml中，如果有特殊字符，也会自动转义。但是，如果需要输出的Xml内容放在<![CDATA[ ]]>中，那么将UseCData设为true，输出的内容将会放在CDATA区域。

例：

[SimpleElement(UseCData = true, Order = 10)]

public string ScanType { get; private set; }

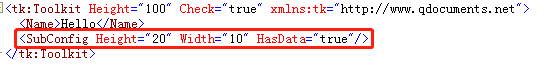
输出格式是

<ScanType><![CDATA[Test]]></ScanType>

### ObjectElement

ObjectElementAttribute从SimpleElementAttribute继承。SimpleElement只是最简单的一种元素表达方式。事实上，元素通常包含有若干个属性，若干个子元素，而子元素同样可以包含有若干个属性，若干个子元素。因此，针对这种复杂的元素，我们必须声明一个新的类来定义它的属性，它的子元素。而分析这个类型，我们必须使用ObjectElement这个Attribute。

例：



红框部分，使用SimpleElement就不合适了，这里我们需要定义一个新的C#类型：

internal class TestSubConfig

{

[SimpleAttribute]

public int Height { get; private set; }

[SimpleAttribute]

public int Width { get; private set; }

[SimpleAttribute]

public bool HasData { get; private set; }

}

同时，在SimpleConfig中，我们需要追加如下定义：

[ObjectElement]

public TestSubConfig SubConfig { get; private set; }

ObjectElement和SimpleElement的最大区别在于SimpleElement的Field通常都是简单类型，而ObjectElement肯定是一个包含有Toolkit定义的Attribute的C#类。ObjectElement可以不断的嵌套定义，这样可以钻取非常深的层次。SimpleAttribute，SimpleElement和ObjectElement是这套Attribute最基础的部分，可以解决90%以上的功能需求。

ObjectElement的属性有：

* UseConstructor

#### UseConstructor

UseConstructor默认为false。如果分析的类有默认构造函数，或者声明了public的无参数构造函数，那么使用该类型实例化类是没有任何问题的。但是如果该类型的构造函数不是public，是protected，甚至是private的时候，这时候，通过类型是没法进行实例化的。在这种情况下，将UseConstructor设置为true，那么系统将去查找对应无参数的构造函数，并进行对应的实例化，这种方式虽然效率差，但是可以解决类型因为封装，而无默认构造函数的问题。

### TextContent

TextContentAttribute从BaseObjectAttribute继承，说明TextContent本身是没有名称的。Xml中存在着一种元素，它类似于SimpleElement中定义的元素，但是它却含有属性，如下图。如果此时用SimpleElement标记，虽然可以取得其内容，但是却无法获取它的属性值。如果使用ObjectElement，那么其内容又不可能用SimpleElement或者ObjectElement标注，那么该如何得到其内容呢？TextContent就是为了解决这个问题。

C:\Users\ADMINI~1\AppData\Local\Temp\1562639724(1).png

在图中，要读取内容Sample，必须把一个Property设置为TextContent，因此，我们可以声明下面的类：

internal class SampleContent

{

[TextContent]

public string Content { get; private set; }

[SimpleAttribute]

public Color Color { get; private set; }

}

由于读取的内容，没有标签，所以这个Property的名字是无视的，即可以使用任意你想用的名字。

下一步，要读取Data标签，我们必须在父类中加上

[ObjectElement]

public SampleContent Data { get; private set; }

在Json以及其他很多种格式中，都没有TextContent这种形式。它们的默认方式是将其转换为名称为“Content”的属性和SimpleAttribute类似。

TextContent的属性有：

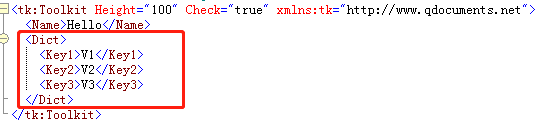
* UseSourceType
* AutoTrim

这些属性和前面描述的相同，这里不再解释。

### Dictionary

由于系统提供了一系列和Dictionary相关的Attribute，为了统一规范，所以定义了BaseDictionaryAttribute作为Dictionary的共同基类。BaseDictionaryAttribute从NamedAttribute继承，DictionaryAttribute从BaseDictionaryAttribute继承。

Dictionary是处理这种场景的。如果一个元素一下有一堆元素，且这堆元素都是和SimpleElement标注的元素一致，在这种情况下，完全可以用一个Dictionary<string, string>来存储这些元素。如下图：



在分析的类中，做如下声明：

[Dictionary]

public Dictionary<string, string> Dict { get; private set; }

对Property的类型，常规大家都是使用Dictionary<string, T>这种泛型结构。不过，Toolkit对此不作严格规定，只要求类型必须是实现IDictionary接口即可，这一点和SimpleElement要求类型实现IList接口类似。

同时，和SimpleElement的IsMultiple=true的情况类似，它自动把Property的类型映射到CollectionType属性上，同时，当Property类型是泛型定义，且有两个泛型时，自动把内容的类型（即第二个泛型参数）映射到ObjectType上。第一个泛型参数，一般要求是string类型。

根据上面的假设，Dictionary在处理Xml时，会无视子元素的Namespace，在输出时也一样不会输出Namespace。

BaseDictionary的属性如下：

* CollectionType
* Order

Dictionary的属性如下：

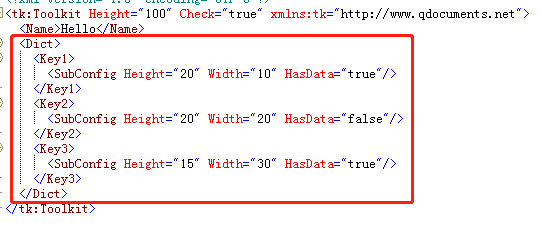
* AutoTrim

这些属性和前面描述的相同，这里不再解释。

### ObjectDictionary

ObjectDictionaryAttribute从BaseDictionaryAttribute继承。ObjectDictionary其实和Dictionary类似。它的场景和Dictionary类似，只不过它的内容不是简单的诸如string那样的类型。而是还嵌套了子元素，需要用ObjectElement标记的类才能表示。这时声明的类型需要改为Dictionary<string, SomeObject>。和Dictionary的行为类似，它自动把Property的类型映射到CollectionType属性上，同时，当Property类型是泛型定义，且有两个泛型时，自动把内容的类型（即第二个泛型参数）映射到ObjectType上。因此，除非是自己写的类型实现了IDictionary接口，才需要显示指定ObjectType，一般情况下，无需设定。

ObjectDictionary的示例如下图：



在分析类中，做如下声明：

[ObjectDictionary]

public Dictionary<string, TestSubConfig> Dict { get; private set; }

ObjectDictionary的属性有：

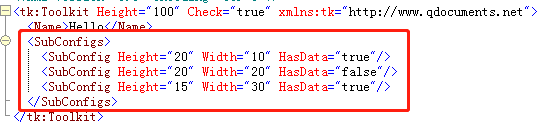
* UseConstructor

这些属性和前面描述的相同，这里不再解释。

### TagElement

TagElementAttribute从NamedAttribute继承。TagElement是对Xml定义的一种优化，有了它，可以减少我们声明类的个数。

我们看以下的Xml：



要分析SubConfigs中的内容，我们需要做如下声明：

internal class TestSubConfigList

{

[ObjectElement(IsMultiple = true, LocalName = "SubConfig")]

public List<TestSubConfig> SubConfigList { get; private set; }

}

然后在顶层类中加上声明，如下：

[ObjectElement]

public TestSubConfigList SubConfigs { get; private set; }

上述标准过程在读写Xml或者Json数据时，完全没有问题。只是如果可以直接在父类中声明：

[ObjectElement(IsMultiple = true, LocalName = "SubConfig")]

public List<TestSubConfig> SubConfigList { get; private set; }

这样使用起来会比较方面，也少了一个类。TagElement就是适用于这个场景。它的目标是存在这么一种元素，没有属性，而且子元素有且仅有一种类型（注意是类型，不是个数。当前情况下，子类型通常是SimpleElement或者是ObjectElement，引入DynamicElement后，它下面的所有配置插件也可以认为是一种类型）。这时，使用TagElement可以消除一个嵌套类，同时也可以简化程序访问代码。

那么前述代码，我们做如下改造，首先取消TestSubConfigList类。然后在父类中声明：

[ObjectElement(IsMultiple = true, LocalName = "SubConfig")]

[TagElement]

public List<TestSubConfig> SubConfigs { get; private set; }

这里，Property的名字SubConfigs是映射给TagElement的。没有名字会映射给附着的其他Attribute，所以，必须强行指定附着的ObjectElement或者SimpleElement的LocalName属性。

TagElement的属性有：

* Order

这些属性和前面描述的相同，这里不再解释。

### API调用

学习完Attribute后，如果读写Xml或者Json，需要调用ObjectExtension或者ObjectExtension2中包装好的扩展函数（只用使用using YJC.Toolkit.Sys;即可）。

#### 读写Json

ReadJson/WriteJson是读写Json的函数。

读写Json的代码如下：

SampleConfig config = new SampleConfig();

config.ReadJson(json);

string json2 = config.WriteJson();

#### 读写Xml

ReadXml和WriteXml是读写Xml的函数。

读写Xml要稍许麻烦一点。因为Xml会有根节点的存在，而Json是没有的。读写Xml必须首先指定根节点。这里默认的根节点是{http://www.qdocuments.net}Toolkit，即带有命名空间的Toolkit节点。如果不是这个根节点，需要使用QName.Get方法来获取。没有命名空间的节点可以使用QName.Get("Test")获取，有命名空间的节点，可以用QName.Get("youruri", "Test")获得。读写Xml的示例代码如下：

SampleConfig config = new SampleConfig();

QName root = QName.Get("Test");

config.ReadXml(xml, ReadSettings.Default, root);

string xml2 = config.WriteXml(WriteSettings.Default, root);

如果根节点是{http://www.qdocuments.net}Toolkit，读写代码可以简化为：

config.ReadXml(xml);

string xml2 = config.WriteXml();

此外，还有ReadXmlFromFile扩展函数，可以直接从Xml文件中读取数据到对象中。例如：

InternalTk5DataXml xml = new InternalTk5DataXml();

xml.ReadXmlFromFile(fileName);

#### 读写QueryString

ReadQueryString和WriteQueryString是读写QueryString的函数

读写QueryString的代码如下：

SampleConfig config = new SampleConfig();

config.ReadQueryString(query);

string query2 = config.WriteQueryString();

#### 读写DataRow

ReadFromDataRow和AddToDataRow是读写DataRow的函数

读写DataRow的代码如下：

PageInfo = new CountInfo(1, 1, 1);

PageInfo.ReadFromDataRow(countRow);

DataRow row = resolver.NewRow();

WeUser user = WeUser.GetUser(openId);

user.AddToDataRow(row);

除了对单条Row进行操作外，还在这个基础上包装了CreateListFromTable和CreateTable两个函数，让集合类型和DataTable进行相互转换。示例代码如下：

List<TestContent> list2 = table.CreateListFromTable<TestContent>();

DataTable table = listOpertors.CreateTable("ListOperator");

if (table != null)

DataSet.Tables.Add(table);

#### 写Dictionary

这个是当初Java因为引入FreeMarker后，移植过来的。感觉用处不大。将一个对象转换为Dictionary层次，如果觉得有用可以使用。

WriteDictionary是写入Dictionary的函数。代码如下：

Dictionary<string, object> dict = obj.WriteDictionary();

#### 写DataSet

读DataSet的代码由于种种原因，没用完成。但是将复合结构写入DataSet，这个功能存在。

AddToDataSet是写入DataSet的函数，代码如下：

fPageInfo.AddToDataSet(dataSet);

#### 写XElement

Toolkit4曾经大量的使用过微软提供的XDocument和XElement，因此Toolkit5也提供了转换函数。但是实际用处并不大。

CreateXDocument是写入XDocument的函数，读取因为有Xml，所以这里无意义。示例代码如下：

XDocument doc = obj.CreateXDocument();

### 小结

通过上面的章节，我们对Toolkit定义的Attribute已经有了大致的了解，可以对任意的Xml格式或者Json格式或者QueryString格式，写出对应分析的C#类，并用简单的API对此进行读写。初级部分只要能够掌握这些就足够了。下面我们开始中级篇的介绍。

## 中级篇

进入中级篇，我们会越来越多的要编写程序，而配置也变得更加高深。

先介绍两个非常简单的编程接口。

### 两个简单的变成接口

#### IReadObjectCallback

IReadObjectCallback的定义如下：

public interface IReadObjectCallBack

{

void OnReadObject();

}

当一个类实现了此接口后，在分析程序读完Xml/Json后，会调用此接口的OnReadObject方法，这样，可以在此方法中做一些需要。比如如果一个类的值为null，可以设置默认值。例如：

public virtual void OnReadObject()

{

if (OperatorPosition == null)

OperatorPosition = YJC.Toolkit.Razor.OperatorPosition.Left;

}

#### IParentObject

IParentObject的定义如下：

public interface IParentObject

{

void SetParent(object parent);

}

当一个类实现了这个接口后，如果在分析过程中，父类创建了此类，并在分析该类完成后，将调用此接口，传入父类的实例。在此方法中可以保存父类的实例对象，并做相关的逻辑操作。

如果一个类即实现了IReadObjectCallback接口，又实现了IParentObject接口。从时序上来说，先调用IReadObjectCallback接口，后调用IParentObject接口。

在实战中IReadObjectCallback的用处要远远高于IParentObject。

### 继承问题

在实际项目中，如果多个配置有共同的配置项，那么这些共同的配置项可以继承吗？答案是肯定的，可以继承，而且被继承Property无需在子类中重新定义。但是需要注意一个问题，那就是Property的可见度问题。

在基类中，定义Property的读写访问，必须是子类可以访问得到的可见度，**也就是说至少是protected，而不能是private**。推荐使用protected。如果把基类的Property的set定义为private，将导致变量写入失败。

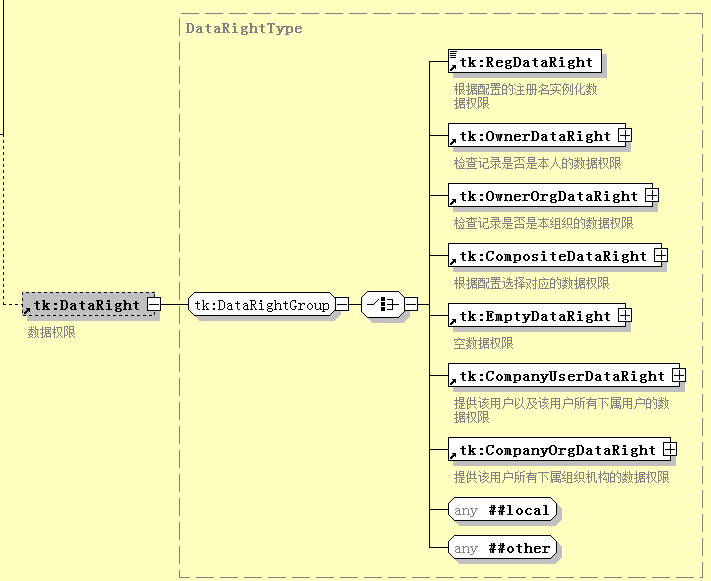
### 配置插件

到目前为止，我们使用Attribute读取的Xml内容都是静态的，预定义好的。理想状态下这样当然很好，可是实际情况并非完全如此。比如，对于Xml配置，有需求希望对于某一类型的节点，可以根据实际需要不断的扩充，而这种扩充具有不确定性，不可预见性。最早出现这种需求的时候，是我开发J2ME，使用Xml配置来画界面。当时将常用的控件封装成Xml配置，比如Label，Button，Radio等。但是随着应用的开发，会不时出现新的配置需求。而每次出现新的配置，就意味着要修改读取配置的代码，烦不胜烦。

在引入使用Attribute解决Xml读写配置的时候，针对上面的情况，我想到了使用配置插件。即定义一个插件工厂作为容器，每一种配置作为插件加入到这个容器中，以标签的元素名称作为注册名。在分析Xml时候，遇到一个节点后，将检索插件工厂是否有这样名字的插件存在，如果存在，则调用此插件进行分析。如果不存在，则忽略此节点。而每个插件分析后，都可以根据其配置来生成一个具有相同基类（或接口）的类实例，以供程序进一步使用。采用了配置插件机制后，无论增加什么插件，增加多少插件，都不需要修改最底层的核心代码。而增加的插件，既可以属于核心类库的类，也可以属于具体项目中的类。

在实际应用中，使用配置插件的地方也很多。比如对某个字段的数据进行校验，如果是数值范围，需要配置一个最大值，一个最小值。如果是对长度进行限制，则需要配置最大长度的值。如果是进行正则表达式的匹配，需要配置相应的正则表达式。不同的要求，对配置的参数，以及其类型要求都不同，只有配置插件可以满足其需求。

在Toolkit框架中，配置插件也得到了广泛的应用。比如数据权限，就采用配置插件设计，如下图：



不同的配置需要的参数不同，但它们都是平等的插件。而且根据以后的需求可以不断的扩展。需要注意，我在Xml Schema定义上使用了xs:any节点，这就意味着，除了上面的节点外，还可以增加新的节点（即新的配置插件），这样Xml编辑器发现新的节点也不会报错。在我定义的Xml Schema中，如果发现了xs:any节点，那就意味着，这块属于配置插件概念，与其平行的所有节点都是配置插件。

配置插件是一个比较深奥的概念，如果没接触过Xml，对此比较难于理解。但是它确实一个非常灵活，非常有用的概念。如果掌握的好，你的程序将变得非常灵活，可扩展。另外，配置插件这块大多数情况下都是存在于Xml中，在原先的版本中，只有部分情况才能在Json中使用，不过现在最新版本已经解决了这个问题，配置插件可以在Json中使用。

### DynamicElement

DynamicElementAttribute从BaseObjectAttribute继承。 DynamicElement就是配合配置插件而出现的。它的属性比较简单，有如下属性：

* FactoryName
* IsMultiple
* CollectionType
* Order
* UseJsonObject

这其中，IsMultiple，CollectionType和Order和前面的含义相同，不再解释，这里解释一下DynamicElement特有的两个属性。

#### FactoryName

FactoryName是配置插件工厂的注册名，这是必须指定的。每个插件工厂都会有一个唯一名字作为其注册名（如果两个工厂的注册名一样，那么只会有一个被系统接纳，至于是哪一个，那就看运气了）。使用DynamicElement，必须告诉它，将使用哪个配置插件工厂。

例如，上面的Schema配置图对应的代码就是：

[DynamicElement(DataRightConfigFactory.REG\_NAME)]

[TagElement(NamespaceType.Toolkit)]

public IConfigCreator<IDataRight> DataRight { get; protected set; }

DynamicElement没有从NamedAttribute继承，而是从BaseObjectAttribute继承，说明它不关注元素的名称，因为每个插件会声明它自己的元素名称。而且，由于各个插件的类型肯定不一样，所以ObjectType属性也多余了。

前面在TagElement中曾经说过，这里重复一遍。DynamicElement下虽然每个节点的元素名称不同，但系统视其为一种类型的节点。也就是说，可以使用TagElement减少配置的类。

#### UseJsonObject

在早期Toolkit的版本中，使用DynamicElement的配置时，只有当IsMultiple为false的时候，才能用Json的方式读出，而IsMultiple为true时，因为Json在这里采用数组的方式读取，而Json数组每一项都完全一样，无法展现出不同的节点不同的内容。UseJsonObject是为了解决这种情况，来定义的。很显然，这个属性只针对Json数据格式，其他格式就无视了。

当UseJsonObject为true时，且IsMultiple也为true时，那么将不再使用Json数组来表达，而是使用Json对象来进行表示，这就完全解决了Json数组的弱点。当然，这也意味着，分析程序将使用和原先不一样的分析方法来分析。

为了将新增这个属性的配置影响缩小，同时也为了让原有的代码更好的适配，这里做了优化，当IsMultiple设置为true时，自动将UseJsonObject也设置为true。所以正常情况下，可以无视这个属性。

### DynamicDictionary

DynamicDictionaryAttribute 从BaseDictionaryAttribute继承。DynamicDictionary是DynamicElement和ObjectDictionary的混合体。理论上，DynamicDictionary和DynamicEelement在IsMultiple设置为true时，具有相同的功能。不同的是，DynamicEelement将读出的对象存储到IList中，而DynamicDictionary将读出的对象存储到IDictionary中，以每个配置插件的注册名做为Dictionary的Key。

DynamicDictionaryAttribute有如下属性：

* FactoryName
* UseJsonObject

含义和DynamicElement相同。

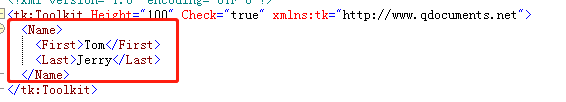
### SimpleComplexElement

SimpleComplexElementAttribute是从SimpleElementAttribute继承。SimpleElement主要是处理内容简单的Xml标签，对于内容复杂的标签，常常采用ObjectElement来处理。

但是，有时还有一个新的需求，即Xml的元素中有内嵌的子元素，但是希望它们直接读到string中。这样，已有的SimpleElement和ObjectElement都不合适。

这时，就引入了SimpleComplexElement。它的作用就是把复杂的内容读入到简单变量（通常都是string）中。这种需求在后面高级篇中介绍ICustomReader后，变得很有用。因为配置插件的缘故，很多复杂定义可以内嵌在Xml元素中，如果没有SimpleComplexElement，那么Xml内容要么需要转义，要么必须放在CDATA中，无法使用xml schema定义带来的编写便捷性和校验作用。而有了SimpleComplexElement，这些问题就解决了。而读出的内容可以用过系统提供的ReadXmlFromFactory和ReadJsonFromFactory扩展函数，变成对应的配置插件对象。

如图，红框中的内容是需要读取的



对应的，代码定义为：

[SimpleComplexElement]

public string Name { get; private set; }

SimpleComplexElement没有定义新的属性，完全都继承于SimpleElement的属性。

需要注意的问题是，在Xml中，子元素的内容是Xml，而Json中，子元素的内容是Json。如果用Xml读入，再用Json输出，那么输出的内容依旧是Xml。而用Json读入，Xml输出，那么输出的内容依旧是Json，这里系统不会做自动转换。

### ComplexContent

ComplexContentAttribute从从BaseObjectAttribute继承，说明ComplexContent和TextContent一样本身是没有名称的。

ComplexContent和SimpleComplexElement处理的问题类似，都是将ObjectElement处理的内容当做SimpleElement来处理。只不过，ComplexContent的作用和TextContent类似。这里就不再单独为ComplexContent举例了。

ComplexContent没有定义新的属性，完全都继承于BaseObjectAttribute的属性。

### NameModel

NameModel是一个被微信API整出来的概念。主要是微信API有Xml/Json两套体系，而两套体系才用了两套不同的命名规范。

看下面的例子：

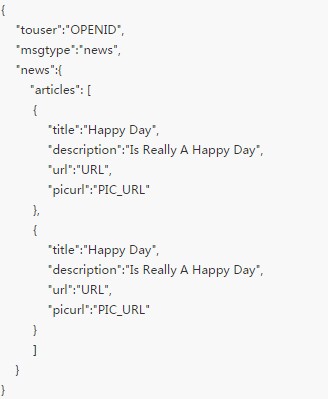
微信中有个消息叫图文消息，如图：



在微信的API中，如果是接受消息后被动回复图文消息，其格式如下：



而作为客服消息主动发送时，它的格式如下：



注意Articles中的内容，每个元素就是一个图文消息。在Xml中，它是Pascal方式，而在Json中，它是全小写方式。如果没有NameModel引入，就必须定义两个图文消息结构，而且两个结构还必须做相互的类型转换，这种折腾只有做的人才明白。

引入NameModel后，就是期望增加一种新的模式，在这种模式下，元素的命名方式将采用其他方式，而不是原有SimpleElement或者SimpleAttribute等规定的格式。NameModel可以重复定义，这样一个属性在不同的模式下，有不同的名称。

NameModelAttribute的属性有：

* ModelName
* NamingRule
* LocalName
* NamespaceType
* NamespaceUri

其中ModelName必须输入。这里的含义就是给它一种模式的名称。当读写Xml时，使用了该名称作为模式参数，那么系统将无视附着的诸如SimpleElement，SimpleAttribute等关于名字的配置，而采用才NameModel中定义的名字进行读写。如果一个Property没有配置NameModel，而只有比如SimpleElement，那么即使读写使用了NameModel定义的模式，它也将使用SimpleElement定义的名称进行读写。

下面是解决上面微信图文消息的定义：

public class Article

{

[SimpleElement(Order = 10, UseCData = true)]

[NameModel(WeConst.JSON\_MODE, NamingRule = NamingRule.Camel)]

public string Title { get; set; }

[SimpleElement(Order = 20, UseCData = true)]

[NameModel(WeConst.JSON\_MODE, NamingRule = NamingRule.Camel)]

public string Description { get; set; }

[SimpleElement(Order = 30, UseCData = true)]

[NameModel(WeConst.JSON\_MODE, NamingRule = NamingRule.Lower)]

public string PicUrl { get; set; }

[SimpleElement(Order = 40, UseCData = true)]

[NameModel(WeConst.JSON\_MODE, NamingRule = NamingRule.Camel)]

public string Url { get; set; }

}

### List模型和Dictionary模型

#### 背景

在原先的Toolkit版本中，并没有专门对List和Dictionary做出分析。由于Json往往存在单独输出Array的情况，所以在Json的插件中，单独对这个情况作了特别处理。但是并没有把它上升到一个模型来处理，也就是说，其他插件都不会支持单独对List或者Dictionary的处理。

在实践中，同样还发现，如果Json数据是[[1,2],[2,3]]这样的内嵌数组，那么现有的Attribute无论怎么声明，也无法读出上面的数据结构。

在Toolkit5.5的新版框架开发中，也存在着需要直接将List或者Dictionary类型和Json直接进行相互转换的需求。

基于上述特性，在Toolkit5的基础上，对原有的读写框架进行了扩展，尝试让所有的插件，只要可能，就能够与List和Dictionary进行相互转换。（具体参加最后的一览表）

#### 特点

由于List模型和Dictionary模型的提出和实现都是在相应的插件中。在实际使用中，和原先并无什么大不同。

这两个模型主要是针对Json这种数据格式的，所以，Json对这两种格式支持是所有插件中最好的，这一点，从最后的一览表可以看出。

其他大多数插件，都能比较稳妥的支持Dictionary模型，但是List模型需要额外的配置，由于这些模型是刚刚开发的，缺少特别的扩展函数，所以，很遗憾，除Json外，其他数据模型即使支持，暂时也没有办法直接和List进行相互转换。这点不足，将很快的补足。

#### 使用方法

以下是Json的例子：

public void TestSimpleList()

{

string json = "[1, 3, 5]";

List<int> list = new List<int>();

list.ReadJson(json);

string json2 = list.WriteJson();

json = "[[1, 2],[2, 3]]";

List<List<int>> list3 = new List<List<int>>();

list3.ReadJson(json);

json2 = list3.WriteJson();

int[][] intArr = new int[][] { new int[] { 1, 3 }, new int[] { 2, 4, 6 } };

json2 = intArr.WriteJson();

json = "[{\"Hello\":true},{\"Hello\":false},{\"Hello\":true}]";

List<TestSubObject> list2 = new List<TestSubObject>();

list2.ReadJson(json);

json2 = list2.WriteJson();

}

public void TestDictionary()

{

string json = "{'Hello':100, 'World':'true'}";

Dictionary<string, string> dict = new Dictionary<string, string>();

dict.ReadJson(json);

Assert.AreEqual(dict["Hello"], "100");

string json2 = dict.WriteJson();

json = "[{'Hello':100, 'World':'true'}, {'Hello':300, 'World':'false'}]";

List<Dictionary<string, string>> list = new List<Dictionary<string, string>>();

list.ReadJson(json);

Assert.AreEqual(list.Count, 2);

json2 = list.WriteJson();

json = "{'Hello':[100, 200], 'World':[200, 100]}";

Dictionary<string, List<int>> dict2 = new Dictionary<string, List<int>>();

dict2.ReadJson(json);

Assert.AreEqual(dict2["Hello"].Count, 2);

json2 = dict2.WriteJson();

json = "{'Hello':[{\"Hello\":true},{\"Hello\":false}], 'World':[{\"Hello\":true},{\"Hello\":false},{\"Hello\":true}]}";

Dictionary<string, List<TestSubObject>> dict3 = new Dictionary<string, List<TestSubObject>>();

dict3.ReadJson(json);

Assert.AreEqual(dict3["Hello"].Count, 2);

json2 = dict3.WriteJson();

}

以下是Xml的例子：

string xml = "<tk:Toolkit xmlns:tk='http://www.qdocuments.net'><Hello Hello=\"true\"/><World Hello=\"false\" /><Midas Hello=\"true\" /></tk:Toolkit>";

Dictionary<string, TestSubObject> dict = new Dictionary<string, TestSubObject>();

dict.ReadXml(xml, ReadSettings.Default, QName.Toolkit);

Assert.AreEqual(dict.Count, 3);

xml = "<tk:Toolkit xmlns:tk='http://www.qdocuments.net'><Hello>Alt</Hello><World>Shift Alt</World></tk:Toolkit>";

Dictionary<string, ConsoleModifiers> strDict = new Dictionary<string, ConsoleModifiers>();

strDict.ReadXml(xml, ReadSettings.Default, QName.Toolkit);

Assert.AreEqual(strDict["Hello"], ConsoleModifiers.Alt);

以下是QueryString的例子：

public void TestDict()

{

string querystring = "hello=123&world=234";

Dictionary<string, string> dict = new Dictionary<string, string>();

dict.ReadQueryString(querystring);

Assert.AreEqual(dict["hello"], "123");

string query2 = dict.WriteQueryString();

}

#### 读取复合Json数组

前面在背景中说了，Json数据是[[1,2],[2,3]]这样的内嵌数组，原有的Attribute模型无法声明。但是，现在有了List模型后，这种类型的数据可以解决了。根据上面的例子，我们可以这么声明：

[SimpleElement(IsMultiple = true)]

public List<List<int>> Value { get; private set; }

如此，便可以直接读取了。

### 如何编写配置插件

配置插件在读写Xml上和普通的类并无区别。一样使用SimpleAttribute，SimpleElement，ObjectElement等Attribute。但是，配置插件需要指明自己的元素名，此外，配置插件还需要根据配置，创建出相应的对象，这是配置插件和普通分析类所不同的地方。

下面将重点说明这两部分。

#### 配置插件的Attribute

一般在创建插件工厂的时候，和插件工厂配套会创建相应的Attribute来对应这个插件工厂。当一个类标注了相应的Attribute，系统就会自动认为该类属于这个Attribute的对应插件工厂的插件，并会自动将此类添加入插件工厂。

配置插件的Attribute都是从BaseObjectElementAttribute继承的，而BaseObjectElementAttribute是从BasePlugInAttribute继承的。

BaseObjectElementAttribute有以下属性：

* NamespaceType
* NamespaceUri
* UseConstructor
* Version

BasePlugInAttribute有以下属性：

* RegName
* Author
* Description
* CreateDate

下面介绍以下这些属性的含义

##### RegName

RegName顾名思义就是注册名。系统要求同一个插件工厂的注册名必须唯一。如果不唯一，重复的插件将不会被添加入插件工厂，至于哪个不会被添加，那就看运气了。

在Toolkit中，可以不用显式的为一个类添加注册名，它可以按照一定的规则来获取。对于配置插件，默认的规则如下：

1. 如果显式指定注册名，以显式指定为准。
2. 如果没有注册名，自动获取该类的类名，如果以Config结尾，将Config截除，取剩余部分作为注册名。
3. 如果类名没有以Config结尾，取整个类名作为注册名。

在配置插件中，注册名实际上就是对应于相应Xml Element的LocalName。两者吻合才会调用该类对Xml进行分析。

##### Description，Author，CreateDate

这三个属性的含义分别是描述，作者和创建日期。从开始定义插件的时候起，就一直有这么三个属性，标注这个插件的作用，创建人和创建日期。这三个属性对程序无影响，但是它可以最终插件的原始信息。而且如果开发了插件浏览的页面可以看到系统注册的插件以及附属的这些信息。这些信息可以不填，但是为了软件开发的规范性，原则上都应该填写的。

##### NamespaceType，NamespaceUri

和初级篇定义的规则一样。它们和注册名RegName一起组成一个Xml的元素名。如果元素没有Namespace，那么可以不设置。在Json分析时，是无视Namespace的。

##### UseConstructor

同初级篇定义一样，如果插件类不是没有默认构造函数，那么就应该设置为true，否则无法实例化插件类。

##### Version

如果插件工厂支持Version特性，那么，同一个标签，不同的Version，可以映射到不同的类。这为定义升级带来了便利。这里声明相应的配置，当Xml定义的配置和这里的配置相同时，将使用该类进行分析处理。

如果插件工厂不支持Version，那么该属性将被无视。

#### 配置插件生成对应的类

插件类定义了各种配置，在读取了相应的Xml后，就可以生成与该配置相关的类了。一般来说，配置插件都是一种中介类，它真正的目的是要生成最终的具有某种基类或实现某个接口的业务类型。当然，有时为了偷懒，确实可以让配置类也同样是相应的业务类，不过一般推荐小型的业务类可以这么做，大型的业务类最好把配置和业务分离，一个类做配置，一个类做业务。

在这里引入一个泛型接口IConfigCreator，它的定义如下：

public interface IConfigCreator<T>

{

T CreateObject(params object[] args);

}

它提供了创建业务类的方式，当然它不是必须的，只是推荐。通常配置类实现该接口，并通过CreateObject方法来创建业务类。正常情况下，创建业务类是不需要外部参数的，但也不能排除不需要，因此，函数采用了可变参数传入。如果确实有外部参数，也可以带入。此外ObjectUtil提供了QueryObject工具函数，可以根据参数的类型，从args中查到对应的参数，这样外部参数的传入就不需要按照严格的顺序了，同时也提高了程序的健壮性。例如：

IInputData input = ObjectUtil.QueryObject<IInputData>(args);

#### 具体示例

有了上面两个基础概念，下面是一个具体的例子。在Toolkit中，把系统提供的数据权限作为配置插件来处理。因此系统定义的插件工厂DataRightConfigFactory和对应的Attribute DataRightConfigAttribute（具体这些如何创建，参见高级篇对应章节）。那么下面是EmptyDataRight的配置类和实现类。

public class EmptyDataRight : IDataRight, IRightCustomMessage

{

public EmptyDataRight(bool allowAll)

{

AllowAll = allowAll;

}

#region IDataRight 成员

public IParamBuilder GetListSql(ListDataRightEventArgs e)

{

return AllowAll ? null : SqlParamBuilder.NoResult;

}

public void Check(DataRightEventArgs e)

{

if (!AllowAll)

throw new NoDataRightException(ErrorMessage);

}

#endregion

public bool AllowAll { get; private set; }

public string ErrorMessage { get; set; }

}

[DataRightConfig(NamespaceType = NamespaceType.Toolkit, Author = "YJC",

CreateDate = "2013-10-28", Description = "空数据权限")]

internal class EmptyDataRightConfig : IConfigCreator<IDataRight>

{

#region IConfigCreator<IDataRight> 成员

public IDataRight CreateObject(params object[] args)

{

return new EmptyDataRight(AllowAll).SetErrorText(ErrorMessage);

}

#endregion

[SimpleAttribute]

public bool AllowAll { get; private set; }

[ObjectElement(NamespaceType.Toolkit)]

public MultiLanguageText ErrorMessage { get; protected set; }

}

## 高级篇

### 高级编程接口ICustomReader

大多数情况下，已有的Attribute已经足够处理各种各样的情况了。但是实际情况，往往比想象的要复杂。比如，某个节点的名称不确定，只有在具体运行的时候才能确定，在这种情况下，就没有办法通过Attribute声明来解决。

因此，在设计的时候，设计了ICustomReader接口，来解决上面碰到的问题。

ICustomReader接口定义如下：

public interface ICustomReader

{

bool SupportVersion { get; }

CustomPropertyInfo CanRead(string localName, string version);

object GetValue(string localName, string version);

void SetValue(string localName, string version, object value);

}

该接口的工作流程如下：

1. 当分析程序查到一个节点，已有的所有的Element节点都没有匹配上时，将检测当前对象是否实现了ICustomReader接口。
2. 如果当前对象实现ICustomReader接口，将调用CanRead函数，并传入Element节点的LocalName。
3. 如果CanRead认为可以处理该节点，将返回对应的处理类型Type和对应的Attribute。否则返回null
4. 如果CanRead返回不是null，那么将根据返回的Type和Attribute进行对应的分析。如果检测出Attribute需要List对象或者Dictionary对象，那么将调用GetValue尝试获取。
5. 分析完毕后，将Attribute产生的对应的对象通过SetValue传入，由相应的代码进行处理。
6. 如果节点支持Version属性，那么SupportVersion为true，在相应的CanRead和GetValue函数中，都会传入节点配置的version。如果SupportVersion为false，那么对应的参数无效。

有了这个接口后，就可以通过这接口做一些有意思的事情。比如，有一个Xml的片段，是某个配置插件工厂的，如果直接转换成对应的对象，那么我们可以通过如下手段来实现：

1. 声明一个类，通过实现ICustomReader接口，让此类来分析相应的Xml

internal class ToolkitDynObject : ICustomReader

{

private string fLocalName;

private string fVersion;

private readonly IConfigFactoryData fConfigData;

public ToolkitDynObject(string factoryName)

{

BaseXmlConfigFactory factory = BaseGlobalVariable.Current.FactoryManager.GetConfigFactory(

factoryName);

fConfigData = factory.ConfigData;

}

#region ICustomReader 成员

public bool SupportVersion

{

get

{

return fConfigData.SupportVersion;

}

}

public CustomPropertyInfo CanRead(string localName, string version)

{

ObjectElementAttribute attr = fConfigData.GetObjectElementAttribute(localName, version);

if (attr != null)

{

fLocalName = localName;

fVersion = version;

return new CustomPropertyInfo(attr.ObjectType, attr);

}

return null;

}

public object GetValue(string localName, string version)

{

return null;

}

public void SetValue(string localName, string version, object value)

{

if (SupportVersion)

{

if (localName == fLocalName && version == fVersion)

Data = value;

}

else

{

if (localName == fLocalName)

Data = value;

}

}

#endregion ICustomReader 成员

public object Data { get; private set; }

}

1. 封装一个方法，给Xml片段加上外壳，并调用上面的方法，从而最终达到读取的目的

public const string TOOLKIT\_XML\_SKELETON = "<tk:Toolkit xmlns:tk='http://www.qdocuments.net'>{0}</tk:Toolkit>";

public static object ReadXmlFromFactory(this string xml, string factoryName, ReadSettings settings)

{

TkDebug.AssertArgumentNullOrEmpty(xml, "xml", null);

TkDebug.AssertArgumentNullOrEmpty(factoryName, "factoryName", null);

string newXml = string.Format(ObjectUtil.SysCulture, ToolkitConst.TOOLKIT\_XML\_SKELETON, xml);

ToolkitDynObject obj = new ToolkitDynObject(factoryName);

ReadXml(obj, newXml, settings, QName.Toolkit);

return obj.Data;

}

1. 由于配置工厂读取的配置往往都是IConfigCreator<T>类型，因此在外层再加一层封装

public static T CreateFromXmlFactory<T>(this string xml, string factoryName, params object[] args)

{

IConfigCreator<T> obj = ReadXmlFromFactory<IConfigCreator<T>>(xml, factoryName);

TkDebug.AssertNotNull(obj, string.Format(ObjectUtil.SysCulture,

"\"{0}\"在配置工厂{1}中无法读出，请确认相关参数", xml, factoryName), null);

return obj.CreateObject(args);

}

public static object ReadXmlFromFactory(this string xml, string factoryName)

{

return ReadXmlFromFactory(xml, factoryName, ObjectUtil.ReadSettings);

}

1. 有了上面的工作，在代码中我们可以直接这样做了

private const string PAGEMAKER = "<FreeRazorPageMaker RazorFile=\"razortemplate"

+ "/BootCss/Bin/TraceManager.cshtml\"/>";

return PAGEMAKER.CreateFromXmlFactory<IPageMaker>(

PageMakerConfigFactory.REG\_NAME, pageData);

类似的，系统还提供了ReadJsonFromFactory和CreateFromXmlFactoryUseJson，和上面介绍的方法相类似。

此外，系统还提供了

public static T ReadJson<T>(this string json, bool useConstructor)

public static T ReadJson<T>(this string json) where T : new()

public static T ReadXml<T>(this string xml, bool useConstructor)

public static T ReadXml<T>(this string xml) where T : new()

这几个扩展函数。它们的实现和上面的方式类似。不同于初级篇介绍的读取Xml和Json的方法，这里不需要实例化相关的对象实例，只要一个Xml片段或者Json片段，通过ICustomReader接口，就能够实例化出对应.net对象，并进行相应的读取操作。

### 自定义ITkTypeConverter

SimpleAttribute，SimpleElement这些Attribute可以把Xml和Json中的字符串转换成相应的.net类型。而这个转化功能靠的都是ITkTypeConverter接口。

ITkTypeConverter接口的思想实际上来自于WPF中的TypeConverter，它拥有上述功能。但是，Toolkit5开发此功能时，使用的是可移植类库，这种类型的库，只能使用很小一部分.net framework的库，因为这个原因，无法直接使用TypeConverter。因此，只能独立声明了ITkTypeConverter接口。

ITkTypeConverter接口定义如下：

public interface ITkTypeConverter

{

string DefaultValue { get; }

object ConvertFromString(string text, ReadSettings settings);

string ConvertToString(object value, WriteSettings settings);

}

它实际就是提供了字符串和.net类型进行相互转换。和TypeConverter类相比，它也要简单很多。系统内置了常用的TypeConverter类型，有：

* int
* string
* short
* long
* float
* double
* DateTime
* Guid
* bool
* byte[]（Base64格式）
* string[]（用,号隔开的字符串）
* TimeSpan
* Encoding
* CultureInfo
* 枚举类型
* Nullable类型

SimpleAttribute和SimpleElement实际上就是需要获取类型的ITkTypeConverter，通过ITkTypeConverter进行类型转换。所以，即使是.net类，如果有对应的ITkTypeConverter，一样可以用SimpleAttribute和SimpleElement读取。

下面举一个具体的例子，在Toolkit中，有一种控件叫做CheckBoxList，它由一组CheckBox组成，存储选中的值，存储格式是"Value1","Value2"...,"Valuen"

对应的，提供了QuoteStringList类来读取并输出这种格式的数据。如果要让SimpleAttribute和SimpleElement识别，必须开发对应的ITkTypeConverter，因此，系统中有如下代码：

class QuoteStringListTypeConverter : BaseTypeConverter<QuoteStringList>

{

protected override object InternalConvertFromString(string text, ReadSettings settings)

{

return QuoteStringList.FromString(text);

}

}

有了这个类型后，我们对QuoteStringList类做如下声明

[TkTypeConverter(typeof(QuoteStringListTypeConverter))]

public class QuoteStringList

{

这样，系统在分析QuoteStringList时，就能找到对应的TypeConverter进行分析处理了。

再举一例，微信API中，时间被定义为1970年1月1日到现在的秒数。因此，得到的往往是一个很大的整数，而不是具体的时间，这样非常不直观（钉钉也是这样规定的）。因此，我们开发了针对这个时间转换的TypeConverter。

private static readonly DateTime Start = TimeZone.CurrentTimeZone.ToLocalTime(

new DateTime(1970, 1, 1));

public static DateTime ToDateTime(int createTime)

{

long tick = createTime \* 10000000L;

return Start + new TimeSpan(tick);

}

public static int ToCreateTime(DateTime date)

{

return (int)(date - Start).TotalSeconds;

}

public sealed class IMDateTimeConverter : ITkTypeConverter

{

#region ITkTypeConverter 成员

public string DefaultValue

{

get

{

return IMUtil.ToCreateTime(DateTime.Now).ToString(ObjectUtil.SysCulture);

}

}

public object ConvertFromString(string text, ReadSettings settings)

{

try

{

int value = int.Parse(text, ObjectUtil.SysCulture);

return IMUtil.ToDateTime(value);

}

catch

{

return DateTime.Now;

}

}

public string ConvertToString(object value, WriteSettings settings)

{

try

{

DateTime date = (DateTime)value;

return IMUtil.ToCreateTime(date).ToString(ObjectUtil.SysCulture);

}

catch

{

return DefaultValue;

}

}

#endregion

}

有了这个TypeConverter后，我们就在微信返回的日期类型下，做如下标记：

[SimpleElement(Order = 30)]

[TkTypeConverter(typeof(IMDateTimeConverter))]

public DateTime CreateTime { get; protected set; }

这样，在程序中，我们可以使用DateTime的各种函数进行操作，而读取和输出都是符合微信API规范的整数。此外，在Property上标记TkTypeConverterAttribute将具有最高的优先级，可以覆盖任何已有的TypeConverter。

除了内置的TypeConverter，系统还提供了如下TypeConverter，在需要的时候可以选择：

* BoolIntConverter：bool和0/1相互转换
* DateStringTypeConverter：日期转换为yyyyMMdd格式
* DateTimeStringTypeConverter：日期转换为yyyyMMddHHmmss格式
* LowerCaseEnumConverter：枚举类型输出为全小写
* UpperCaseEnumConverter：枚举类型输出为全大写
* StringHashSetConverter：用,或者空格分隔的字符串，转换成HashSet<string>
* EnumFieldValueTypeConverter：在每个枚举值上标注EnumFieldValueAttribute，然后根据EnumFieldValueAttribute中的值，进行相互转换

额外提一下，Toolkit还提供了一个非常有意思的扩展函数Value<T>，将尝试把你提供的对象类型转换为T类型，它就是使用ITkTypeConverter进行的。

### 如何创建自己的配置工厂

配置插件的功能很强大，使用好配置插件可以使自己的业务系统变得功能很强大，而且可扩展性也会非常好。下面就介绍如何定义并注册配置插件。

在Toolkit中，由于已经做了大量的基础工作。所以定义一个配置插件并不是太难的事情。它需要两个类，一个是工厂类，一个是Attribute类。

下面以Toolkit的DataRight配置插件为例来说明。

首先定义一个工厂类，定义工厂类非常简单，但是你必须给工厂类一个唯一的注册名，保证在整个系统不重复。具体例子如下：

public sealed class DataRightConfigFactory : BaseXmlConfigFactory

{

public const string REG\_NAME = "\_tk\_xml\_DataRight";

internal const string DESCRIPTION = "数据权限的配置插件工厂";

public DataRightConfigFactory()

: base(REG\_NAME, DESCRIPTION)

{

}

}

这里，我们定义了一个类，必须从BaseXmlConfigFactory继承。然后定义了名字和描述的常量，在构造函数中，向基类传入这两个常量。通常建议把名字定义为常量，这样在DynamicElementAttribute中可以直接使用这个常量，防止由于书写错误而导致程序出错。

接下来需要创建Attribute类了，示例如下：

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]

public sealed class DataRightConfigAttribute : BaseObjectElementAttribute

{

public override string FactoryName

{

get

{

return DataRightConfigFactory.REG\_NAME;

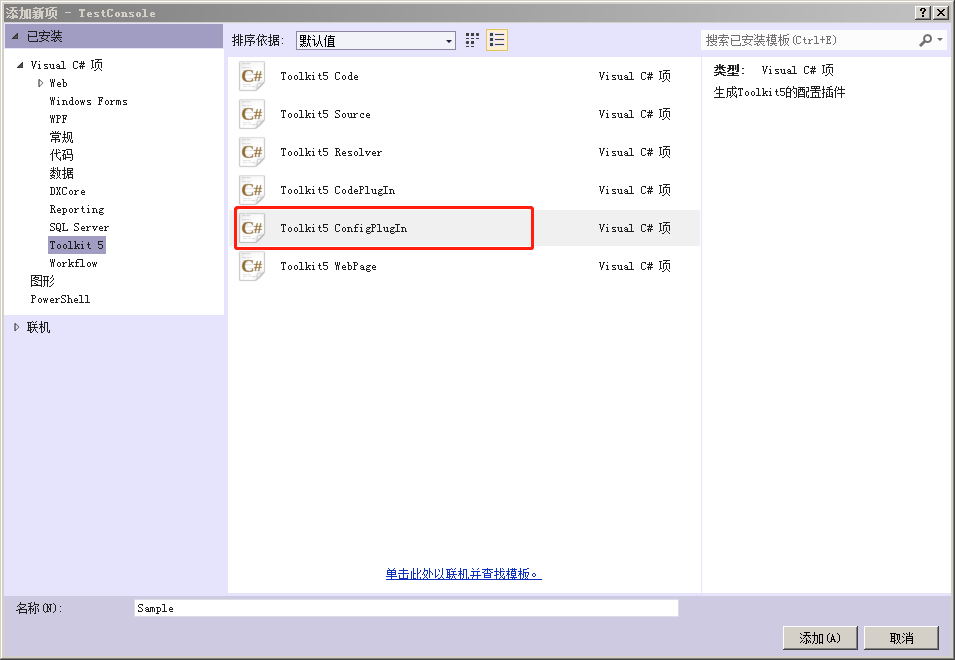
}

}

}

Attibute需要从BaseObjectElementAttribute继承，这里只需要实现FactoryName这个属性，这里返回对应工厂的注册名即可。由此，Attribute和工厂之间就建立了关系。

由于代码呈现出比较强的规律性，因此，Toolkit开发了对应的Visual Studio模板，用模板创建这两个类更加简单。在安装了Toolkit的Visual Studio模板的环境中，通过添加新项的对话框，我们就能看到下图



这时候，按添加，会自动生成Attribute和Factory类，稍许修改即可。

C:\Users\ADMINI~1\AppData\Local\Temp\1563863331(1).png

BaseXmlConfigFactory有两个属性：

* SupportVersion
* DefaultVersion

在原先的基础上，让配置工厂支持version特性。

原先最早的版本是没有version这个特性的。一个Element的名称只能对应一个类，正常情况下这样没什么大问题。但是随着软件的升级，可能出现了大的变更，这是对Xml会有大的升级，甚至改变整个逻辑。这样就带来一个问题，原有已经配置好的，如果要进行修改，势必会有很大的工作量，还有可能存在遗漏。这样，就提出了version的问题来解决升级的问题。比如version="1.0"，那么就是以前的版本，还是以前的类来分析处理。而version="2.0"，则是新的版本，采用新的类，新的方法来分析处理。同时考虑到原先的配置都没有支持version，所以设计DefaultVersion，如果设置为"1.0"，那么没有配置version属性的Xml即等同为version="1.0"。

需要注意的是，当前SupportVersion为true时，**只有Xml才支持这个特点**，Json暂时不支持。

### 如何注入创建好的配置工厂

配置插件开发好后，需要做一些配置，这样系统才能找到这个工厂进行注册。注册工厂的方式有两种，这里只介绍一种比较简单，也比较常用的方式。

在.net的项目中，通常都有一个AssemlyInfo.cs，在这个文件中，添加如下代码即可：

[assembly: AssemblyPlugInFactory(typeof(DataRightConfigFactory))]

另外，要求生成的DLL必须在WebSite的bin目录下。

### API调用的源头

前面的章节中，我们已经基本上说了读写Xml，Json等数据的方法了。现在说说系统的基本架构。如果只追求用法，可以无视此节。

在Toolkit中，定义了IObjectSerializer接口负责.net类型和特定数据格式之间的转换。这个接口定义异常复杂，有兴趣的话可以查看源码，这里就不说了。

根据这个接口，系统分别有XmlObjectSerializer，JsonObjectSerializer，QueryStringObjectSerializer等基于IObjectSerializer的实现。很容易理解，XmlObjectSerializer负责Xml和.net对象转换，JsonObjectSerializer负责Json和.net对象转换，以此类推。当然这些类的实现确实也是比较复杂的，尤其是功能越强的类，实现的代码就越复杂。

在ObjectExtension类中，定义了这么一个函数，如下：

private static void ReadFromReader(object receiver, ReadSettings settings, QName root,

IObjectSerializer serializer, object reader, string modelName)

{

using (reader as IDisposable)

{

if (serializer.ReadToRoot(reader, root))

{

serializer.Read(reader, receiver, modelName, settings, root, null);

}

}

}

它应该是所有读取API的源头。在此基础上分别有两个读取函数，一个从Stream读取，一个从String读取，它们的实现分别如下：

public static void ReadFromStream(this object receiver, string method, string modelName,

Stream data, ReadSettings settings, QName root)

{

TkDebug.AssertArgumentNull(receiver, "receiver", null);

TkDebug.AssertArgumentNullOrEmpty(method, "method", null);

TkDebug.AssertArgumentNull(data, "data", null);

TkDebug.AssertArgumentNull(settings, "settings", null);

IObjectSerializer serializer = CreateSerializer(method);

object reader = serializer.CreateReader(data, settings);

ReadFromReader(receiver, settings, root, serializer, reader, modelName);

}

public static void ReadFromString(this object receiver, string method, string modelName,

string data, ReadSettings settings, QName root)

{

TkDebug.AssertArgumentNull(receiver, "receiver", null);

TkDebug.AssertArgumentNullOrEmpty(method, "method", null);

TkDebug.AssertArgumentNullOrEmpty(data, "data", null);

TkDebug.AssertArgumentNull(settings, "settings", null);

IObjectSerializer serializer = CreateSerializer(method);

object reader = serializer.CreateReader(data, settings);

ReadFromReader(receiver, settings, root, serializer, reader, modelName);

}

剩下那些ReadXml，ReadJson，ReadQueryString基本上都是调用上面两个函数，不同的是，它们的method参数参入的都是各自IObjectSerializer的插件注册名，这样系统根据不同的注册名会实例化出相应的IObjectSerializer，进行相应的处理。这里给出以下几个例子：

public static void ReadXml(this object receiver, string xml)

{

ReadFromString(receiver, "Xml", null, xml, ObjectUtil.ReadSettings, QName.Toolkit);

}

public static void ReadJson(this object receiver, string json)

{

ReadFromString(receiver, "Json", null, json, ObjectUtil.ReadSettings, QName.Toolkit);

}

public static void ReadQueryString(this object receiver, string modelName, string queryString)

{

ReadFromString(receiver, "QueryString", modelName, queryString,

ReadSettings.Default, QName.Toolkit);

}

读是这样，写也差不多。只不过写稍微复杂一点，写有两个基础函数，一个是WriteToString，一个是WriteToStream。以下是两个函数的分别实现：

public static string WriteToString(this object receiver, string method, string modelName,

WriteSettings settings, QName root)

{

TkDebug.AssertArgumentNullOrEmpty(method, "method", null);

TkDebug.AssertArgumentNull(settings, "settings", null);

if (receiver == null)

return null;

IObjectSerializer serializer = CreateSerializer(method);

MemoryStream stream = new MemoryStream();

object writer = serializer.CreateWriter(stream, settings);

using (writer as IDisposable)

{

SerializerUtil.WriteSerializer(serializer, writer, receiver, modelName, settings, root, null);

byte[] data = stream.ToArray();

return settings.Encoding.GetString(data, 0, data.Length);

}

}

public static void WriteToStream(this object receiver, string method, string modelName,

Stream stream, WriteSettings settings, QName root)

{

TkDebug.AssertArgumentNullOrEmpty(method, "method", null);

TkDebug.AssertArgumentNull(stream, "stream", null);

TkDebug.AssertArgumentNull(settings, "settings", null);

if (receiver == null)

return;

IObjectSerializer serializer = CreateSerializer(method);

object writer = serializer.CreateWriter(stream, settings);

using (writer as IDisposable)

{

SerializerUtil.WriteSerializer(serializer, writer, receiver, modelName, settings, root, null);

}

}

类似的，前面说的WriteXml，WriteJson，WriteQueryString也都是调用这两个函数

public static string WriteXml(this object receiver, string modelName)

{

return WriteToString(receiver, "Xml", modelName, ObjectUtil.WriteSettings, QName.Toolkit);

}

public static string WriteJson(this object receiver)

{

return WriteToString(receiver, "Json", null, ObjectUtil.WriteSettings, QName.Toolkit);

}

综上，可以对读写数据的框架有个初步了解。

## 插件一览

|  |  |
| --- | --- |
| **注册名** | **描述** |
| Dictionary | Dictionary对象格式转换器 |
| QueryString | QueryString格式转换器 |
| Json | Json格式转换器 |
| Xml | Xml格式转换器 |
| DataSet | DataSet格式转换器 |
| DataRow | DataRow格式转换器 |
| XElement | XElement格式转换器 |

## Attribute一览

|  |  |
| --- | --- |
| Attribute名称 | 说明 |
| SimpleAttribute | 针对Xml中的Attribute |
| SimpleElement | 针对Xml的Element，无属性，内容中无嵌套的Element |
| ObjectElement | 针对Xml的Element，含属性，含内嵌Element |
| TagElement | 针对仅是嵌套的Element，可以减少类型的声明 |
| TextContent | 针对Xml的Element含有属性，但是内容无嵌套时的内容 |
| ComplexContent | 相对TextContent，ComplexContent可以将Element中嵌套的内容完整的读出做为字符串赋值给相应的变量 |
| SimpleComplexElement | 和ComplexContent类似，但是从SimpleElement演化出来 |
| DynamicElement | 支持配置插件工厂，只要存在于配置工厂的Xml都能被解析出来 |
| Dictionary | 使用Dictionary<string,string>将对应配对读出Xml/Json等键值配对 |
| ObjectDictionary | 和Dictionary类似，但是将值不是简单的对象，而是一个复杂对象。要求每个对象结构相同 |
| DynamicDictionary | 支持配置插件工厂，这里Dictionary的键将是插件的注册名，而值是插件注册类型对应读取的对象 |

## 插件支持一览

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **读取** | **Dictionary** | **QueryString** | **Json** | **Xml** | **DataSet** | **DataRow** | **XElement** |
| SimpleAttribute | × | √ | √ | √ | × | √ | √ |
| SimpleElement | × | √ | √ | √ | × | 注3 | √ |
| ObjectElement | × | × | √ | √ | × | 注4 | √ |
| TagElement | × | × | √ | √ | × | × | √ |
| TextContent | × | 注1 | 注1 | √ | × | 注1 | √ |
| ComplexContent | × | × | √ | √ | × | × | √ |
| SimpleComplexElement | × | × | √ | √ | × | √ | √ |
| DynamicElement | × | × | √ | √ | × | √ | √ |
| Dictionary | × | × | √ | √ | × | × | √ |
| ObjectDictionary | × | × | √ | √ | × | × | √ |
| DynamicDictionary | × | × | √ | √ | × | × | √ |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **写入** | **Dictionary** | **QueryString** | **Json** | **Xml** | **DataSet** | **DataRow** | **XElement** |
| SimpleAttribute | √ | √ | √ | √ | × | √ | √ |
| SimpleElement | √ | √ | √ | √ | × | 注3 | √ |
| ObjectElement | √ | × | √ | √ | √ | 注5 | √ |
| TagElement | √ | × | √ | √ | × | × | √ |
| TextContent | 注2 | 注2 | 注2 | √ | × | 注2 | √ |
| ComplexContent | × | × | √ | √ | × | × | √ |
| SimpleComplexElement | × | × | √ | √ | × | √ | √ |
| DynamicElement | √ | × | √ | √ | × | √ | √ |
| Dictionary | √ | × | √ | √ | √ | × | √ |
| ObjectDictionary | √ | × | √ | √ | √ | × | √ |
| DynamicDictionary | √ | × | √ | √ | × | × | √ |

注1 : 假定有名称为Content的节点，读取相关内容

注2 : 将内容写入名称为Content的节点

注3 : 不支持IsMultiple为true时的配置

注4 : 假定DataColumn中的内容为Xml，进行Xml反序列化读为对象

注5 : 将内容变为Xml写入DataColumn中

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **写入** | **Dictionary** | **QueryString** | **Json** | **Xml** | **DataSet** | **DataRow** | **XElement** |
| SimpleAttribute | √ | √ | √ | √ | × | √ | √ |
| SimpleElement | √ | √ | √ | √ | × | 注3 | √ |
| ObjectElement | √ | × | √ | √ | √ | 注5 | √ |
| TagElement | √ | × | √ | √ | × | × | √ |
| TextContent | 注2 | 注2 | 注2 | √ | × | 注2 | √ |
| ComplexContent | × | × | √ | √ | × | × | √ |
| SimpleComplexElement | × | × | √ | √ | × | √ | √ |
| DynamicElement | √ | × | √ | √ | × | √ | √ |
| Dictionary | √ | × | √ | √ | √ | × | √ |
| ObjectDictionary | √ | × | √ | √ | √ | × | √ |
| DynamicDictionary | √ | × | √ | √ | × | × | √ |

## List|Dictionary支持一览表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 插件 | Object | | List | | | Dictionary | |
| 读取 | 写入 | 是否支持 | 显式Attribute | 是否递归 | 是否支持 | 是否递归 |
| Dictionary | × | √ | × | × | × | × | × |
| QueryString | √ | √ | √ | √ | × | √ | × |
| Json | √ | √ | √ | × | √ | √ | √ |
| Xml | √ | √ | √ | √ | √ | √ | √ |
| DataSet | × | √ | × | × | × | × | × |
| DataRow | √ | √ | × | × | × | √ | × |
| XElement | √ | √ | √ | √ | √ | √ | √ |