CBF平台设计之组件对象

目录

[1. 组件对象 4](#_Toc501696627)

[1.1. 序列化引擎 4](#_Toc501696628)

[1.2. OXM引擎 9](#_Toc501696629)

[1.3. 组件对象模型 14](#_Toc501696630)

[1.4. 组件对象模型扩展 23](#_Toc501696631)

# 组件对象

组件对象（Component Object，简称CO）模块致力于构建在所有业务应用领域都具有普遍需要的引擎、模型、框架和工具——序列化、OXM、组件、组件工厂等等，这些设计为所有组件化的应用程序的编写模式和规范奠定了基础。

组件对象模块由：1、序列化引擎；2、OXM引擎；3、组件对象模型；4、组件对象模型扩展；共4个子模块构成：

序列化引擎子模块提供了一个帮助应用实现对象序列化方法的引擎。不同于一般的序列化方法，这是基于可设定格式的字符串字节流的序列化方法。

OXM引擎子模块提供了一个帮助应用实现对象-XML映射（Object Xml Mapping，简称OXM）方法的引擎。

组件对象模型子模块确立了组件类的标准构建方式。组件类具有自描述构造的特性，并由此自动具备OXM方法和序列化方法，以及可通过组件对象设计器查看、设计组件对象运行时状态的特性。在组件工厂的配合下，组件具备可热更新的特性，并能够为IoC设计模式提供支持。

组件对象模型扩展子模块提供常用的基础组件类、集合组件类和相关的辅助类，以方便设计具有复杂组件对象结构的应用。



## 序列化引擎

序列化引擎子模块提供了一个帮助应用实现对象序列化（支持从对象转换成字节流，以及从字节流转换成对象）方法的引擎。不同于一般的序列化方法，这是基于可设定格式（编码字符集，域长度，分隔符，等等）的字符串字节流的序列化方法。

#### 需求背景和目标

交易系统运行时刻，服务系统需要把客户端系统传来的字节流转换成业务数据对象，交由应用处理，还需要把应用处理后产生的业务数据对象转换成字节流发送回客户端系统，这个转换过程就称为序列化/反序列化。如果业务数据对象自身具备序列化/反序列化能力，则应用和应用框架的开发就会更加简洁优美。

但让所有业务数据对象自身都具备序列化/反序列化能力，不是一件简单的事情。如果要求所有业务数据类的开发者自行编写代码实现序列化/反序列化接口，开发工作量巨大，质量也难以保证。为此，我们提出序列化引擎设计，这是用于支持业务数据对象以极其简易的方式获得序列化/反序列化能力的一种框架设计。

序列化所产生或需处理的字节流可以有多种形式，在这个序列化设计中，我们约定的字节流形式是由字符串数据、数值数据等基本类型数据按约定编码方式产生的二进制编码，按约定位置和分隔符紧密连接形成的字节流。

例如，有如下数据类和应用代码：



执行后，我们希望by是如下状态（‘\_’表示空格字符对应的ASCII码，即0x20，其他数字、字母字符均表示相应字符对应的ASCII码，空白表示0x00）：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \_ | 3 | , | O | K | , | h | e | L | L | o | \_ | \_ | , | \_ | 1 | 0 | 0 |  |  |  |  |  |  |

#### 序列化接口和可序列化类

一个方法，只要能够实现业务数据对象与字节流之间的转换，似乎就可以被称为序列化接口，不必在意其表现形式——但是，我们不这么看，统一序列化接口的形态具有很大的意义，序列化框架不仅仅以帮助实现序列化接口为目标，其内部工作机制也与序列化接口的形态有关。我们提出如下接口，作为序列化接口的标准形态：



其中，nStartPosition、nEndPosition表示byValue中的有效数据的起始、终止位置；nLength表示对于当前对象而言，序列化所用数据的长度。外部应用在调用Serialize方法时，如果填写的bIsLoading为false（表示要序列化），则nLength的Value值只会是0，如果填写的bIsLoading为true（表示要反序列化），则nLength的Value值可能不会是0——这表示调用者知道当前对象序列化所用数据的长度。不管怎样，Serialize方法执行完后，都应当在返回前设置nLength的Value值为经过处理后的准确的占用数据的长度。

只有实现了该接口的类，我们才称之为可序列化类（相应的实例称为可序列化对象），该接口产生或需处理的字节流，我们称之为序列化数据。基于这个统一的序列化接口，再借助于序列化图和序列化器的支持，我们能够让拥有大量可序列化对象（在组件对象模块中，我们将会提供最常用的可序列化的基本数据类型：String、Integer、Long、Bolean、Float等等）作为其成员的、结构复杂的数据类很轻松地实现这个序列化接口，也成为可序列化类。

#### 序列化图和序列化器

假定我们已经有了A、B、C等3个可序列化类，现在，我们需要构建一个拥有A、B、C等3类对象作为其成员的可序列化类D：



我们只需在D类中实现一个接口来填写序列化图，标明其有哪些成员对象（必须都是可序列化的）会加入到D类对象的序列化过程中，随后，即可将此图交给序列化器，由序列化器完成D类对象的序列化操作。

所谓序列化图其实就是一个可序列化（成员）对象的登记表：



D类填写序列化图的接口的实现非常简单：



再借助序列化器，D类就可以轻松实现序列化接口：



D类对象的序列化数据，实际上就是其下a、b、c这3个成员对象的序列化数据的紧密连接（连接处是否有分隔符，取决于序列化控制）。注意，如果A、B、C这3个可序列化类中有些也是用上述方法构建的，那么，D类就是一个具有多层数据结构的可序列化类。

#### 序列化控制

上文中有一个多次提到但未深入阐述的概念：序列化控制（SerializeControl），这实际上就是一个包含了一组可以控制、影响序列化过程的参数的对象。参数包括：分隔符是哪个字符，字符串数据编码方式，是否所有域之间都必须插入分隔符，是否直接使用二进制布局存放数值型数据，等等。要注意的是：对于一个具体的可序列化类而言，序列化控制中的参数是否起作用，是由该类的序列化接口的实现决定的——这意味着相关参数不一定都能起作用，比如，序列化控制指出所有域之间都必须插入分隔符，但某个序列化接口实现得不是很好的对象仍然产生了不含分隔符的序列化数据。

序列化控制的存在意味着同一个可序列化对象产生的多份序列化数据，可以是不同的。

#### 动态序列化支持

某些可序列化对象运行时刻拥有数量不定的、动态变化的可序列化子对象，例如集合型的数据对象，在父对象序列化时，我们希望这些动态变化的子对象也能加入到这个过程中——相应的序列化数据成为父对象序列化数据的组成部分，这就是我们所讲的动态序列化。为支持动态序列化，我们对序列化器和序列化图进行了增强。

首先，增加IGetDynamicSerialize接口，用于获取所有需要加入序列化过程的动态（子）对象：



然后，在序列化图中增加了SetGetDynamicSerialize方法，用于在序列化图中登记IGetDynamicSerialize接口：



运行时刻，序列化器会调用序列化图的GetGetDynamicSerialize方法来获得其中登记的IGetDynamicSerialize接口，进而反复调用该接口的GetDynamicSerialize方法，获得所有需要加入序列化过程的动态对象，并调用这些对象的Serialize方法，对序列化数据中的动态区域（紧随存放非动态子对象序列化数据的静态区域之后）进行读、写，完成序列化过程。

nIndex表示在序列化器中是连续第几次调用GetDynamicSerialize方法。

集合型数据类应当实现IGetDynamicSerialize接口。如果是要做序列化，集合型数据类的IGetDynamicSerialize接口实现通常应该先判断nIndex是否已经大于自己包含的元素的个数，如果已经大于，则返回null，序列化器将不会再调用GetDynamicSerialize方法，如果不大于，则返回第nIndex个元素（这个元素必须是可序列化对象）；如果是要做反序列化，则IGetDynamicSerialize接口实现通常会依据byValue中的数据（从nStartPosition开始），创建新的、类型合适的元素加入集合，并返回新创建的元素。

由于在反序列化时，byValue中的部分信息可能是用于断定应该产生什么样的对象，而不是用于新对象的反序列化，因此，nStartPosition是一个引用型的整数，在该方法返回时，可以调整其Value值，指明对应新对象的序列化数据的最新起始位置。

反序列化时刻，序列化器如何能知道何时该停止调用GetDynamicSerialize方法呢？序列化器有3种方式来确定动态对象个数：1、使用确知的总长度减去序列化数据静态区域的长度，如果已处理的序列化数据动态区域长度达到这个值，就表明已经完成所有动态对象的创建和反序列化；2、使用确知的序列化数据动态区域长度，如果已处理的序列化数据动态区域长度达到这个长度，就表明已经完成所有动态对象的创建和反序列化；3、使用确知的动态对象的个数——所谓“确知”的信息，通常都是写在序列化数据中的。究竟采用哪种方式，则取决于序列化图的一个名为LengthRecordMode的可读/可写的属性。

#### 分隔符的运用

我们知道，在反序列化一个不定长的字符串对象时，如果序列化数据中也没有表述其长度，就只能依赖序列化数据中的分隔符来终止该对象的反序列化过程。实际上，不只是字符串对象，所有不定长的、又不在序列化数据中表述其长度的可序列化对象，都必须依赖序列化数据中的分隔符来终止其反序列化过程——这意味着即便序列化控制不要求所有域之间都必须插入分隔符，某些域后面也一定会跟着分隔符。

#### 占位支持

在实际应用场景中，有些子对象（包括静态成员对象和动态子对象）在父对象序列化过程中需要采用占位模式来完成其本身的序列化过程——这类占位子对象序列化时，都会先在父对象序列化数据某个位置放置一个占位标志，然后，在所有非占位子对象序列化数据后依次放置占位子对象的序列化数据。需要注意的是，占位标志应该能够区别表示在父对象序列化数据中有、无该子对象序列化数据两种情形。

序列化器支持这种占位需求。ISerialize接口中有一个名为OccupyingMode的可读属性，该属性值为0，则表示该子对象不采用占位模式进行序列化；为1则表示该子对象采用bit占位模式，即其占位标志是字节中的一个bit，1表示有，0表示无；为2则表示该子对象采用字节占位模式，即其占位标志是一个字节，0x31（即ASCII字符'1'）表示有，0x30（即ASCII字符'0'）表示无；为3则表示该对象采用长度占位模式，即其占位标志是一个4位长度的ASCII编码的数字字符串描述的（子对象序列化数据的）长度值。

#### 序列化引擎的特性和优势

与常见的开源序列化引擎Protocal Buffers，Thrift，Avro，Bond相比，本序列化引擎具有很鲜明的特性和很显著的优势——

首先，它没有一个用IDL或类似IDL的文件描述序列化数据结构的过程，自然也不需要通过工具依据IDL描述的信息来自动生成一个数据类，而是需要可序列化类实现一个设置序列化图的接口，这就意味着可序列化类的开发、使用都很方便（接口实现比编写IDL描述更加简洁，部署时无需部署配套的IDL脚本），运行速度也会快很多（运行时不会再访问IDL描述的信息，也不会通过反射来操作对象的属性，2K长度的序列化数据，处理时间消耗通常都能控制在1微秒以内，与常见开源引擎性能相差超过10倍），同时也意味着使用java语言编写的可序列化类如果需要转换为其他语言，需要重新编程——系统开发环境配备了统一的架构设计和管理工具，数据结构定义是这个工具管理的重要内容，在这个工具上，我们实际上已经完成了与使用IDL定义序列化数据结构非常类似的工作，通过这个工具就可以自动生成各种语言版本的可序列化类！

其次，它完全基于接口，而非基于类型工作，这就意味着应用更加灵活，能够很好地应对为现有数据类添加序列化能力、在序列化过程中同步进行数据交验、使用自管理的组件工厂来支持动态序列化等一系列问题，解决方案都非常合理、高效。

最后，它是独创的支持占位的序列化引擎，这意味着它能够很好地适应传统交易系统开发人员的报文设计习惯和遗留的报文规范。而对bit占位的支持，甚至让它可以适应ISO8583报文和很多类ISO8583的报文。

## OXM引擎

OXM引擎子模块提供了一个帮助应用实现OXM（支持从对象转换成XML，以及从XML转换成对象）方法的引擎。

#### 需求背景和目标

XML在企业应用系统中已经得到广泛深入的运用，传统的交易系统也希望能够支持XML格式的交易报文。运行时刻，服务系统需要把客户端系统传来的XML报文转换成业务数据对象，交由应用处理，还需要把应用处理后产生的业务数据对象转换成XML报文发送回客户端系统。从XML报文转换成业务数据对象的过程通常被分为两个阶段，一是通过XML解析程序，把XML报文转换为DOM对象，二是把DOM对象转换为业务数据对象，从业务数据对象转换成XML报文的过程通常也被分为两个阶段，一是把业务数据对象转换为DOM对象，二是基于DOM对象的信息生成XML报文。XML报文与DOM对象之间的转换技术手段很多，也很成熟，我们在工具包中提供的DocumentHelper就是一个非常高效的把XML报文转换为DOM对象的工具，而DOM对象自身又提供方法，能够很方便地生成XML报文。

DOM对象与业务数据对象之间的转换更为关键，所谓OXM，主要就是指这个过程。如果业务数据对象自身具备OXM能力，则应用和应用框架的开发就会更加简洁优美。

但让所有业务数据对象自身都具备OXM能力，不是一件简单的事情。如果要求所有业务数据类的开发者自行编写代码实现OXM接口，开发工作量巨大，质量也难以保证。为此，我们提出OXM引擎设计，这是用于支持业务数据对象以极其简易的方式获得OXM能力的一种框架设计。

与业务数据对象对应的DOM对象有两种：一是Attribute（属性）对象，二是Element（元素）对象，Attribute对象只有一个Value（值）属性，Element对象可以包含一组指定名称的Attribute对象和子Element对象。Attribute对象通常只能用来对应那些不可再细分的基本业务数据对象，例如账号，即在Attribute对象的Value属性中存放基本业务数据对象的值；Element对象则通常用来对应那些组合业务数据对象，例如，账户信息，即在其包含的Attribute对象的Value属性中存放作为组合业务数据对象成员的基本业务数据对象的值，在其包含的子Element对象中存放作为组合业务数据对象成员的组合业务数据对象。

例如，有如下数据类和应用代码：



执行后，我们希望strXml是如下状态（注意Attribute对象名称是另行指定的）：



由于Element对象本身也有一个Value（值）属性，所以，很多应用场景会使用Element对象代替Attribute对象，用来对应那些不可再细分的基本业务数据对象，如果采用这种策略，那么，在上面的例子中，执行后，我们希望strXml是如下状态：



#### OXM接口和OXM实现类

一个方法，只要能够实现业务数据对象与DOM对象之间的转换，似乎就可以被称为OXM接口，不必在意其表现形式——但是，我们不这么看，统一OXM接口的形态具有很大的意义，OXM框架不仅仅以帮助实现OXM接口为目标，其内部工作机制也与OXM接口的形态有关。我们提出如下接口，作为OXM接口的标准形态：



其中，nDetranslateMode表示从DOM的Element对象转换为业务数据对象（bIsLoading为true）时，对Element对象数据结构合法性的判定方式：为0则Element对象中不能有任何多余的Attribute对象和子Element对象；为1则Element对象中可以有多余的Attribute对象和子Element对象。请注意！如果Element对象中缺少某些与业务数据对象中信息对应的Attribute对象和子Element对象，这种情形不应被判定为数据结构非法，对于缺少部分，应该忽略相应转换，即保持业务数据对象中的相应信息不变。

对于基本业务数据对象而言（例如String类型的账号），它们也可能会基于DOM的Attribute对象进行OXM，因此，我们又提出如下接口，作为基于Attribute对象的OXM接口的标准形态：



在组件对象模块中，我们将会提供最常用的、同时实现了ITranslateAttribute接口和ITranslateElement接口的基本数据类型：String、Integer、Long、Bolean、Float等等。这些基本数据类型能够满足一般应用开发的需要，应用很少会另行开发基本数据类型。

只有实现了上述接口的类，我们才称之为OXM实现类（相应的实例称为OXM实现对象）。基于这些统一的OXM接口，再借助于转换图和转换器的支持，我们能够让拥有大量OXM实现对象作为其成员的、结构复杂的数据类很轻松地实现ITranslateElement接口，也成为OXM实现类。

#### 转换图和转换器

假定我们已经有了A、B、C、D等4个OXM实现类（其中，A、B是ITranslateAttribute接口实现类，例如String、Integer），现在，我们需要构建一个拥有A、B、C、D等4类对象作为其成员的OXM实现类E：



我们只需在E类中实现一个接口来填写转换图，标明其有哪些成员对象（必须都是OXM实现的）会加入到E类对象的OXM过程中，随后，即可将此图交给转换器，由转换器完成E类对象的OXM操作。

所谓转换图其实就是一个由两个HashMap构成的OXM实现（成员）对象的登记表（注意，这张表中既可以登记ITranslateElement接口实现对象，也可以登记实现ITranslateAttribute接口实现对象，并且登记时都要指定一个名称，即DOM的Element对象下对应的Element子对象和Attribute对象的名称）：



E类填写转换图的接口的实现非常简单：



再借助转换器，E类就可以轻松实现OXM接口：



E类对象的OXM过程产生/面对的Element对象，可以拥有两个名为A1、B2的，与a、b这两个成员对象对应的Attribute对象，以及两个名为C3、D4的，与c、d这两个成员对象对应的子Element对象。

#### 动态转换支持

某些OXM实现对象运行时刻拥有数量不定的、动态变化的OXM实现子对象，例如集合型的数据对象，在父对象进行OXM时，我们希望这些动态变化的子对象也能加入到这个过程中——相应的DOM对象成为父对象对应的DOM对象的组成部分，这就是我们所讲的动态转换。为支持动态转换，我们对转换器和转换图进行了增强。

首先，增加IGetUnnamedTranslateElement接口，用于获取所有需要加入OXM过程的动态（子）对象：



然后，在转换图中增加了SetGetUnnamedTranslateElement方法，用于在转换图中登记IGetUnnamedTranslateElement接口：



在从业务数据对象转换为DOM对象的过程中，转换器在完成所有转换图中的静态对象（即通过SetTranslateElement和SetTranslateAttribute方法登记的OXM实现对象）的OXM过程后，会继续调用转换图的GetGetUnnamedTranslateElements方法来获得其中登记的IGetUnnamedTranslateElement接口，进而反复调用该接口的GetUnnamedTranslateElement方法，获得所有需要加入OXM过程的动态对象，并调用这些动态对象的Translate方法，基于相应的子Element对象进行转换。

nIndex表示在转换器中是连续第几次调用GetUnnamedTranslateElement方法。

集合型数据类应当实现IGetUnnamedTranslateElement接口。在从业务数据对象转换为DOM对象的过程中，集合型数据类的IGetUnnamedTranslateElement接口实现通常应该先判断nIndex是否已经大于自己包含的元素的个数，如果已经大于，则返回null，转换器将不会再调用GetUnnamedTranslateElement方法，如果不大于，则返回第nIndex个元素（这个元素必须是OXM实现对象），同时，应当设置strName对象的Value值，告知转换器需要创建的子Element对象的名称。

在从DOM对象转换为业务数据对象的过程中，转换器一旦发现一个子Element对象的名称不能对应到转换图中的静态对象，就会调用转换图的GetGetUnnamedTranslateElements方法，获得IGetUnnamedTranslateElement接口并调用该接口的GetUnnamedTranslateElement方法，获得需要加入OXM过程的动态对象。集合型数据类的IGetUnnamedTranslateElement接口实现通常应依据strName对象的Value值创建新的、类型合适的元素加入集合，并返回新创建的元素。

#### OXM引擎的特性和优势

与本文中的序列化引擎相对常见的开源序列化引擎所具有的特性和优势相似，本OXM引擎也因为其不依赖IDL，而完全依赖设置转换图的接口的运作机制，使其与常见的开源OXM引擎相比，OXM实现类的开发、使用都更加方便，运行速度也快很多（2K长度的XML报文，处理时间消耗通常都能控制在3微秒以内），应用更加灵活，能够很好地应对为现有数据类添加OXM能力、在OXM过程中同步进行数据交验、使用自管理的组件工厂来支持动态转换等一系列问题，解决方案都非常合理、高效。

## 组件对象模型

组件对象模型子模块确立了组件类的标准构建方式（以Component类为最终基类，并运用常用元素类和其他组件类组装新的组件类）。组件类具有自描述构造的特性，并由此自动具备OXM方法和序列化方法（基于OXM引擎、序列化引擎），以及可通过组件对象设计器查看、设计组件对象运行时状态的特性。在组件工厂的配合下，组件具备可热更新的特性，并能够为IoC设计模式提供支持。

#### 需求背景和目标

我们知道，无论多么庞大的企业应用程序，运行时刻都可以被看作一组对象，这些对象的状态是至关重要的信息：一方面，我们需要了解对象的状态，用以判断系统当前的运行状况，诊断问题所在；另一方面，我们可以设置、调整对象的状态，用以控制系统的运行。

当企业的应用程序是由成千上万个组件共同构成的时候，每一个组件都提供其独特的查看和设置对象状态的手段，是不可想象的，我们必须采用某种规则来约束组件的开发，并在此基础上获得统一的查看和设置对象状态的手段。这个规则就是组件对象模型（Component Object Modal，简称COM），我们要求所有组件开发时都必须遵照COM，按照特定的要求编写代码。请注意这里所讲的“组件”通常都是有明确业务含义的数据和方法的封装，它不是泛指一切应用逻辑，我们并不要求应用程序中所有的对象类型都必须是组件。

#### COM基本规范

COM基本规范要求：

1）组件开发时都必须直接或间接继承自组件对象模块提供的Component类；

2）组件对象中可查看和设置状态的成员对象都应是各种Element派生类（也称元素类，注意这个Element类不是DOM的Element类）和Component派生类对象，即元素对象和组件对象；

3）根据Component类的设计，每个组件对象都会内含一个构造图，组件对象构造时刻必须调用SetElement方法和SetComponent方法，在其构造图中登记可查看和设置状态的成员对象；

4）组件开发完成后都应当在组件工厂配置文件中登记；

5）组件对象的创建，应当统一使用组件工厂提供的CreateComponent方法，而不是new——以解除代码的强依赖关系，并更好地支持组件的热更新机制。

#### 元素对象和赋值校验接口

元素对象用来存放组件对象中需要查看和设置的最小粒度、不可再细分的状态信息。

所有的元素类都是以Element类为基础派生的类型，Element类统一提供了只读的OwnerComponent属性，访问这个属性，就可以知道元素对象所归属的组件对象——即属主组件对象。元素对象创建时OwnerComponent属性值缺省为null，一旦我们调用属主组件对象的SetElement方法，相应的元素对象的OwnerComponent属性值就会随之发生变化——从null变成属主组件对象，或者从属主组件对象变回null。

Element类统一实现了ISerialize、ITranslateAttribute等接口，并统一提供了可读/写的StringValue属性，这些接口和属性的实现都依赖Element类中的抽象方法Translate（这个方法是ITranslateString接口所定义的）：



Element类还统一提供了Initialize方法，调用这个方法，就可以让元素对象恢复到最初的状态。Initialize方法是调用Element类定义的抽象方法OnInitialize来完成初始化过程的，这意味着所有元素类都应当提供OnInitialize方法的具体实现。

组件对象模型提供了组件开发中常用的元素类，包括：Binary、Boolean、Integer、String、Float、Long、Decimal、DateTime，它们都提供了OnInitialize方法和ITranslateString接口中的Translate方法的具体实现。

常用的元素类都包含一个基于java原生类型定义的、存放元素对象状态信息的成员变量（简称值成员），并基于值成员提供了可读/写的Value属性，应用主要通过常用元素对象的Value属性，而不是StringValue属性，来读/写其中的数据。实际上，常用的元素类都依赖值成员实现OnInitialize方法和ITranslateString接口中的Translate方法。不同常用元素类值成员通常也使用不同的java原生类型来定义，对应关系如下：

|  |  |
| --- | --- |
| 元素类 | 值成员的类型 |
| Binary | byte[] |
| Boolean | boolean |
| Integer | int |
| String | java.lang.String |
| Float | double |
| Long | long |
| Decimal | java.math.BigDecimal |
| DateTime | java.util.Calendar |

所有常用元素类都支持赋值校验——即在设置常用元素对象的Value属性值时检查参数是否合法。检查逻辑是由常用元素对象实例化时设置的赋值校验接口实现对象提供的。例如，在创建Integer对象时，可以用一个IValidateIntegerValue接口实现对象作为Integer对象构造函数的参数，其后，在设置Integer对象的Value属性值时刻，就会调用到该接口实现对象的ValidateIntegerValue方法，如果该方法返回的字符串非null，则表示校验失败（返回值就是校验失败原因的描述），对Value属性值的设置就会抛出包含了校验失败原因描述的异常。



在常用元素对象反序列化/OXM时刻（ISerialize和ITranslateAttribute接口方法调用时刻），这个赋值校验机制都会自动发挥作用。

#### 组件对象和组件对象的构造图

所有的组件类都是直接或间接以Component类为基础派生的类型，和Element类一样，Component类也统一提供了只读的OwnerComponent属性，访问这个属性，就可以知道当前组件对象所归属的组件对象——即属主组件对象。组件对象创建时OwnerComponent属性值缺省为null，一旦我们调用属主组件对象的SetComponent方法，相应的组件对象的OwnerComponent属性值就会随之发生变化——从null变成属主组件对象，或者从属主组件对象变回null。

基于OwnerComponent属性，Component类统一提供了IsOwnerComponent方法，用于判定两个组件对象之间是否有归属关系，并提供了只读的SupremacyComponent属性和GetLocalSupremacyComponent方法，用于获取最高层级的属主组件对象。

每个组件对象内部都隐含有一个构造图，这个构造图描述了当前组件对象下有哪些以其为属主组件对象的元素对象和组件对象。构造图主要就是两个HashMap，一个用来登记元素对象，一个用来登记组件对象。登记时每个元素/组件对象都要标定一个名称——即构造登记名称，在同一个构造图中，构造登记名称是不能重复的。Component类统一定义了构造图所需的HashMap成员变量，并统一提供了访问构造图，在其中登记或者获取已登记的元素/组件对象的属性和方法：ElementCount/GetElement/GetElementEntrySet/SetElement、ComponentCount/GetComponent/ GetComponentEntrySet/SetComponent、...、等等：



一个组件对象，一旦作为SetComponent方法的参数被登记到属主组件对象中，这个归属关系通常就不再改变了，因此，这类组件对象我们称之为静态子组件对象。相对的，某些属主组件对象（例如集合型的数据对象）运行时刻拥有数量不定的、动态变化的、无既定构造登记名称的子组件对象，这些子组件对象我们称之为动态子组件对象。为支持有动态子组件对象的属主组件对象的构造描述，我们定义了IGetUnnamedComponent接口：



Component类允许（在构造图中）设置一个IGetUnnamedComponent接口实现对象：



拿到属主组件对象（构造图中）的IGetUnnamedComponent接口实现对象，调用其接口中的GetUnnamedComponent方法，即可获取该属主组件对象下所有动态子组件对象以及对应的构造登记名称（bIsLoading为false），或者尝试使用某个构造登记名称创建合适的动态子组件对象，并将其纳入属主组件对象的管理（bIsLoading为true）。

拥有构造图的组件对象是一个自描述信息结构的对象，依据构造图，我们可以获得其管理的信息结构中任意位置的元素/组件对象，这比通过反射要高效得多，也更加安全、可控。

#### 组件对象的定位、查找方法

拥有构造图和OwnerComponent属性的组件对象，实际上就是一个即可以向上、也可以向下浏览、查找的，由组件对象和元素对象共同构成的一棵对象树，元素对象是这棵树的叶结点，组件对象则是这棵树的中间结点和根结点。

从对象树中的一个结点，到达与其有归属关系的另一个结点，所经过的路径都可以用一个字符串来描述，这个字符串由一组对象在其属主组件对象构造图中的构造登记名称连接（组件对象的构造登记名称之前用'/'隔开，元素对象的构造登记名称之前用'.'隔开）而成——我们称这个字符串为路径字符串。

Component类统一提供了GetPath方法，用以获取路径字符串，以及FindElement方法和FindComponent方法，用于基于给定的路径字符串，以当前组件对象为出发点，查找目标元素对象或目标组件对象：



这些定位、查找方法对于一般的业务应用而言用处不大（而且不推荐使用），但对于应用框架、分析工具等面向切面设计（AOP）的特殊应用而言意义重大。例如，交易系统的应用框架可以调用基于COM开发的交易请求对象的FindComponent方法，试图在标定的目标位置上获得一个账户对象，如果能够获得账户对象，即可以依据一定规则对其进行统一的处理（例如，判别是否为需要监控的黑名单账户，如果是，则立即把相关信息发往报警中心）。

#### 组件对象的OXM和序列化方法

Component类统一提供了OXM方法族和序列化方法族，所有组件类都自动成为OXM实现类和可序列化类：



Component类统一提供的OXM方法和序列化方法，内部都是依赖OXM引擎和序列化引擎的，也就是说，Component类统一实现了填写转换图和序列化图的接口。填写转换图和序列化图所需的信息均来自构造图——

构造图中的元素对象都是实现了ITranslateAttribute接口的OXM实现对象，组件对象都是实现了ITranslateElement接口的OXM实现对象，因此，Component类依据构造图就能自动填写转换图，既把构造图中的元素/组件对象作为参数，调用转换图的SetTranslateAttribute/SetTranslateElement方法，登记名称就使用元素/组件对象的构造登记名称，同时，实现IGetUnnamedTranslateElement接口，把构造图中的动态子组件对象转换为转换图中的动态对象，并设置转换图中的IGetUnnamedTranslateElement接口对象——这意味着Component类OXM方法生成和处理的DOM对象的结构基本是由构造图决定的。

例如，有如下组件类和应用的代码：



执行后， strXml通常是如下状态：



构造图中的元素对象和组件对象都是实现了ISerialize接口的可序列化对象，因此，Component类依据构造图就能自动填写序列化图，既把构造图中的元素/组件对象作为参数，调用序列化图的SetSerialize方法，调用顺序和填写构造图时以相应元素/组件对象作为参数调用SetElement/SetComponent方法的顺序保持一致，同时，实现IGetDynamicSerialize接口，把构造图中的动态子组件对象转换为序列化图中的动态对象，并设置序列化图中的IGetDynamicSerialize接口对象——这意味着Component类序列化方法生成和处理的序列化数据的格式也基本是由构造图决定的。

在前面的示例代码中，如果对s对象调用Serialize方法，通常会得到如下序列化数据：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 1 | | | B | 3 | \_ | 4 | U | \_ | 1 | 1 | 0 | 8 | | | z | h | a | n | g | \_ | \_ | \_ | s | a | n | \_ | \_ | \_ |
| \_ | \_ | 8 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | z | h | a | n | g | @ | c | c | . | c | o | m |  |  |

实际上，在绝大多数应用场景中，组件对象并非是在OXM/序列化方法调用时才填写转换图/序列化图，而是其自身就是一个转换图/序列化图——我们定义了转换图/序列化图接口TranslateMap/ISerializeMap，所有实现了相应接口的类的实例，都可被称为转换图/序列化图。转换器/序列化器实际上是基于转换图/序列化图接口实现对象工作，而不一定非得是一个TranslateMap/SerializeMap对象——Component类统一实现了ITranslateMap/ISerializeMap接口，因此，组件对象的OXM/序列化方法执行很高效。

基于COM开发的接口数据对象的OXM方法和序列化方法都是最常被调用的方法。当接口数据对象需要跨系统传输时，调用这些方法产生的XML文本（字符串）或序列化数据就是所谓请求、应答报文中的主要内容。

基于COM开发的应用对象的OXM方法则常被用于观测运行时应用和对运行时应用进行控制。由于OXM方法对应的XML文件非常具备可读性，所以：一方面，我们可以在应用系统运行出现异常或者其他需要观测、分析应用系统状态的时刻，调用OXM方法，把应用对象的状态输出到指定的XML文件（通常就是应用系统的某种日志文件）中，以便开发人员或者运维人员进行分析；另一方面，我们可以在应用系统启动或者其他需要变更、调整应用系统状态的时刻，调用OXM方法，把指定位置的XML文件中的信息转换为相关的应用对象的状态——这个XML文件通常就是应用系统的某种配置文件，可以由运行管理人员预先编写好。

#### 组件对象的初始化方法和差分方法

Component类统一提供了基于DOM对象或相应XML文件的组件对象的Initialize方法，实际上，这些Initialize方法只是OXM方法的进一步封装：



Component类统一提供了类似拷贝构造函数的组件对象的Initialize方法，即把一个组件对象的状态复制到另一个组件对象上，显然，这个方法内部是依据构造图工作的：



Component类统一提供了让组件对象状态恢复到最初构造时刻的Initialize方法，显然，这个方法内部也是依据构造图工作的，并且会利用到元素对象的Initialize方法：



基于OXM方法，Component类还统一提供了组件对象的差分方法，即对两个组件对象的状态进行比较，把差异部分或相同部分输出到DOM对象中——如果以表示差异部分的DOM对象为参数，再调用Initialize方法，就会消除两个组件对象的状态差异。组件对象的差分方法常被用于分析程序和设计管理程序，例如，我们可以使用设计管理程序创建一个应用对象，并通过可视化界面设置该对象的状态，然后，调用该对象的OXM方法就可以把输出的XML文件用作应用系统的某种配置文件，但这个XML文件由于包含了整个应用对象的所有状态信息，往往过于庞大，可读性很差。如果最后输出时刻以默认状态的应用对象为参数调用该对象的差分方法，就可以得到一个最简洁的XML文件，这个文件不但可读性更好，系统加载时消耗的资源也更少。



#### 组件工厂

所有按COM基本规范开发的组件类的实例化都应当通过（某个）组件工厂（ComponentFactory）对象的CreateComponent方法进行。组件工厂对象内部有一个HashMap，其中存放了所有已登记的组件类型，CreateComponent方法将通过名称查找到对应的组件类型，再通过反射机制完成组件对象的创建。



所有按COM基本规范开发的组件都应当在（某个）组件工厂对象中登记相应类型，即调用组件工厂对象的SetComponentType方法，在其用来存放组件类型的HashMap中增加相应数据（通过组件工厂对象的GetComponentType等方法和属性，我们就可以知道已经登记了哪些组件类型）：



实际上，组件类型的登记大多是在组件工厂对象的初始化过程（即调用组件工厂对象的LoadComponentFactory方法或Initialize方法）中自动完成的，而不是在应用中调用SetComponentType方法来进行的：



从上述方法的参数可以看到，组件工厂对象初始化时大多是依据一个组件工厂配置文件来完成组件类型登记的。组件工厂配置文件是一个XML格式的文件（所以，LoadComponentFactory/Initialize方法族也提供了以DOM对象为参数的方法原型），在这个XML文件中，除去根节点、名为“Catalogue”的节点和名为“Using”的节点以外，其他每个节点都用以登记一个组件类型，所提供的信息包括：组件类型名称（节点名），以及Java类的Package名称（Namespace属性值）——这些节点也被称为类型节点，Catalogue节点下可以放置类型节点和下一层Catalogue节点。



组件工厂配置文件通常由应该工具程序自动生成和修改，而不是手工编辑，但有个例外——组件工厂配置文件支持引用机制，即由多个配置文件共同构成一个配置文件体系，并在主配置文件或者上级配置文件中引用辅配置文件或者下级配置文件——添加Using节点，提供被引用的配置文件的名称（File属性值），和对应的JAR文件的名称（AssemblyFile属性值）。引用时指定被引用的配置文件对应的JAR文件，这既意味着同一个组件工厂配置文件中指定要登记的组件类型必须在同一个JAR文件中能够找到，也意味着不同组件工厂配置文件对应的JAR文件可以不同。但要注意，主配置文件或最顶级的配置文件对应的JAR文件不能指定，其中指定要登记的组件类型通常都应包含在JVM启动时刻加载的JAR文件中。

有一个广泛运用的设计理念：控制反转（Inversion Of Control，即IoC），这是一种设计模式——“IoC模式可被看做是工厂模式的升华，可以把IoC看作是一个大工厂，只不过这个大工厂里要生成的对象都是在XML文件中给出定义的，然后利用Java 的反射编程，根据XML中给出的类名生成相应的对象。从实现来看，IoC是把以前在工厂方法里写死的对象生成代码，改变为由XML文件来定义，也就是把工厂和对象生成这两者独立分隔开来，目的就是提高灵活性和可维护性......IoC模式主要的特点就是通过文本的配置文件进行应用程序组件间相互关系的配置，而不用重新修改并编译具体的代码”——实际上，在组件工厂的支持下进行的组件对象的创建和（基于XML文件的）初始化，就是典型的IoC设计模式的一种运用，这个机制能够有效解除应用代码的强依赖关系。

运行时，组件工厂对象既可以实时更新其中登记的某个组件类型，也可以实时按照临时指定的组件工厂配置文件进行多个组件类型（必须在同一个JAR文件中能够找到）的同步更新——这就是我们常说的热更新机制。显然，要想让热更新机制的运作达到完美的程度——即充分有效（所有底层组件的更新都会在依赖该组件的上层组件执行时生效），又不会因组件类型版本混杂而导致系统运行不可靠——组件工厂就必须得到慎重维护，即必须保证任何一个组件工厂对象中登记的所有组件类型版本是自洽的，同时，应用也必须严格按照开发规范的约定，调用“指定的”组件工厂对象的CreateComponent方法创建组件对象，不能直接new，更不能使用错误的组件工厂对象。

#### 组件对象寄存器

组件对象基于构造图提供了很多有用特性，但这些特性的获得是有代价的——运行时，每个组件对象都至少需要管理两个HashMap，并且构造时都要操作这些数据结构。如果在应用中大量使用组件对象（数量达到万级），系统CPU和内存资源的消耗都不可忽视。

为消除这个不利影响，我们设计了组件对象寄存器机制，使得我们可以重复利用那些已经确定暂时废弃无用的组件对象，从而减少系统CPU和内存资源的消耗。具体机制是，组件对象创建时同步建立一个组件对象寄存器对象，并在两者之间建立相互的引用，然后，把组件对象寄存器对象放入组件对象寄存器管理对象。组件对象寄存器管理对象中有两个链表（链表中的节点就是组件对象寄存器对象，组件对象寄存器对象自身有两个引用，分别指向前一个和后一个组件对象寄存器对象）：正在使用的组件对象寄存器对象链表（简称有效链表）和废弃无用的组件对象寄存器对象链表（简称废弃链表）。新产生的组件对象寄存器对象在加入组件对象寄存器管理对象时，都会插入有效链表。确定一个组件对象已经废弃无用时，可以调用组件对象的Dispose方法，此时，对应的组件对象寄存器对象将会被移入废弃链表。如果调用组件工厂对象的CreateComponent方法再创建一个组件对象，就会先在废弃链表中查找同类组件对象，如果能找到，就将相应的组件对象寄存器对象移回有效链表并返回。注意，组件对象寄存器对象的生命周期往往是和线程生命周期一致的——超过任务（例如一个交易处理过程）的生命周期，因此，下一个任务有可能重用上一个任务中废弃的组件对象。在任务结束时，即便具体处理应用没有及时废弃其使用的组件对象，任务的处理框架也应该把有效链表中的所有组件对象寄存器对象统统移入废弃链表。

组件对象寄存器机制本身也会产生内存消耗，因此，我们设计了一些参数，可以控制组件对象寄存器管理对象对组件对象寄存器对象的管理策略——例如，废弃链表总长度不能超过某个数值，等等。但这还不够，我们还允许应用自行决定是否启用组件对象寄存器机制——默认状态下，组件对象寄存器机制是关闭的，应用只有实现IGetComponentRegisterManager接口（组件对象模块中提供了该接口要返回的ComponentRegisterManager对象的类的缺省实现），并把Component类的公共静态变量GetComponentRegisterManager赋值为一个IGetComponentRegisterManager接口实现对象，才会启用组件对象寄存器机制。



## 组件对象模型扩展

组件对象模型扩展子模块提供常用的基础组件类、集合组件类和相关的辅助类，以方便设计具有复杂组件对象结构的应用。

#### 组件对象集合

为方便构建集合型的组件类，即其实例可以作为指定类型组件对象集合存在的组件类（一种运行时实例拥有动态子组件对象的组件类），组件对象模型扩展子模块提供了集合组件类：ComponentContainer和ComponentArrayList。ComponentContainer对象最多可以管理一个动态子组件对象， ComponentArrayList对象则可以管理任意数目的动态子组件对象：



ComponentContainer类和ComponentArrayList类都继承自Component基类，因此，我们只需在组件类中定义ComponentContainer或ComponentArrayList类型的成员，并在构造时刻调用SetComponent方法完成ComponentContainer或ComponentArrayList类型成员对象的登记，即可快速开发出新的集合型的组件类：



注意ComponentContainer和ComponentArrayList都采用了泛型设计，即在声明时即可指定集合中元素类型限制。如果声明时未指定类型，还可以在构造方法参数中指定类型——在上面的示例中，ConnectionList成员对象创建时构造方法参数指定了Connection类型，这就意味着此后调用ConnectionList成员对象的SetComponent方法，只能传入Connection类型的组件对象。

上面示例中的Staffer类，其OXM方法产生或能处理的DOM对象结构通常为：



显然，名称为ConnectionList的Element对象下的DOM对象结构是ComponentArrayList对象类提供的OXM方法所产生或能处理的。

在上面的DOM对象结构中，名称为ConnectionList的Element对象显得有些多余，在这样的应用中，我们往往更希望直接在根Element对象下直接包含一组名为Connection的子Element对象，以这样的DOM对象结构作为Staffer类的OXM方法产生或能处理的DOM对象结构。

为了支持这样的需求，组件对象模型扩展子模块提供了集合组件装饰器类：ComponentContainerDecorator和ComponentArrayListDecorator。装饰器类本身并非组件类，但在组件类中运用这些装饰器，也能够快速开发集合型的组件类。实际上，ComponentContainer和ComponentArrayList就是分别基于ComponentContainerDecorator和ComponentArrayListDecorator构建的：



运用ComponentArrayListDecorator重构上面的示例中的Staffer类：



其OXM方法产生或能处理的DOM对象结构通常为：



在上面的示例中，无论是使用ComponentArrayList构建的Staffer类，还是使用ComponentArrayListDecorator构建的Staffer类，其序列化方法产生或能处理的序列化数据通常均为（注意动态子组件对象序列化数据前均有类型名称——这个信息是必须存在的——反序列化时需要依据类型名称，通过组件工厂提供的方法创建动态子组件对象）：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| z | h | a | n | g | \_ | \_ | \_ | s | a | n | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | 3 | C | o | n | n | e | c | t | i | o | n |
| | | 8 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | B | 3 | \_ | 4 | U |  | 1 | 1 | 0 | 7 | | | z | h | a | n |
| g | s | a | n | @ | c | c | . | c | o | m | | | C | o | n | n | e | c | t | i | o | n | | | 8 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 1 | 0 | B | 3 | \_ | 4 | U | \_ | 1 | 1 | 0 | 8 | | | z | h | a | n | g | s | a | n | @ | d | d | . |
| c | o | m | | | C | o | n | n | e | c | t | i | o | n | | | 8 | 0 | 0 | 0 | 1 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 1 | 0 | B |
| 3 | \_ | 4 | U | \_ | 1 | 1 | 0 | 9 | | | z | h | a | n | g | s | a | n | @ | e | e | . | c | o | m |  |  |  |  |  |

#### 元素对象集合和元素工厂

#### 字符串枚举和可设定规则的字符串校验器

#### 可作为组件对象运用的XML文档对象