Spring.NET 框架参考文档

方玉中整理 2014.11.21

目录

第一章. 序言

即使有先进的工具和技术，软件开发也是一件相当令人头疼的工作。Spring.NET 为建立企业级应用提供了一套轻量级的解决方案。通过 Spring.NET，我们可以用统一且透明的方式来配置应用程序，并在应用中集成 AOP 的功能。Spring.NET 的重点是为中间层提供声明式事务管理，以及一个功能齐全的 ASP.NET 扩展框架。

Spring.NET 可以为很多领域的企业级应用开发提供“一站式服务”。虽然功能强大，Spring.NET 仍然是模块化的，允许单独使用其中的任一部分。在使用 IoC 容器来配置应用程序时，我们既可以用传统的 ADO.NET 来访问数据库，也可以使用 Spring.NET 的 Hibernate 集成代码或 ADO.NET 抽象层来访问数据库。
Spring.NET 是非侵入式的，代码对框架本身不会产生任何依赖（或者只需要极少的依赖，取决于应用的范畴）。

本文档是 Spring.NET 的参考指南。因为文档内容尚在更新过程中，所以如果读者有什么问题或建议，可以提交到 Spring.NET 的用户邮件列表或支持论坛（论坛地址 forum.springframework.net）。

在正文开始之前，要感谢 Chritian Bauer（Hibernate 小组的成员）为我们准备并修改了 DocBook-XSL 软件，Hibernate 小组用它来创建 Hibernate 的参考文档，我们也用它来创建这份文档。同时也要感谢 Russell Healy 为部分素材作了详细的检查。

第二章. 简介

2.1. 概述

Spring.NET 是一个应用程序框架，其目的是协助开发人员创建企业级的.NET 应用程序。它提供了很多方面的功能，比如依赖注入、面向方面编程（AOP）、数据访问抽象及 ASP.NET 扩展等等。Spring.NET 以 Java 版的 Spring 框架为基础，将 Spring.Java 的核心概念与思想移植到了.NET 平台上。

企业级应用一般由多个物理层组成，每个物理层也经常划分为若干功能层。不同层次之间需要相互协作，例如，业务服务层一般需要使用数据访问层的对象来实现某个用例。不管应用程序如何构建，最终都会表现为一系列相互协作的对象，这些对象一起组成了完整的应用程序。所以我们说，应用程序中的对象之间相互具有依赖性。

.NET 平台为构建应用程序提供了丰富的功能，从非常基础的基元类型和基础类库（以及定义新类的方法），到功能完善的应用程序服务器和 Web 框架，都有很好的支持。但.NET 平台本身并没有提供任何方式来管理基础的应用模块并将它

们组合为一个相互协作的整体，只能依靠架构师或开发人员去创建（一系列）应用程序。诚然，目前有很多设计模式可用于业务系统的设计，我们可以使用这些模式将各种类或对象组合成能够正常工作的完整应用。工厂、抽象工厂、Builder、装饰及服务定位器（Service Locator）等模式已被现今的软件开发行业广泛接受和采用(这也许正是这些模式最早被定型为模式的原因)。这些模式都非常好，但也不过是些已命名的最佳编程方法，在对这些模式的介绍中一般还会说明它们是作什么用的、最好应用到什么场合、可以解决什么问题等等。我们可以从许多书籍和 wiki 上找到这些模式，然后仔细研读，然后实现在我们自己的应用中。

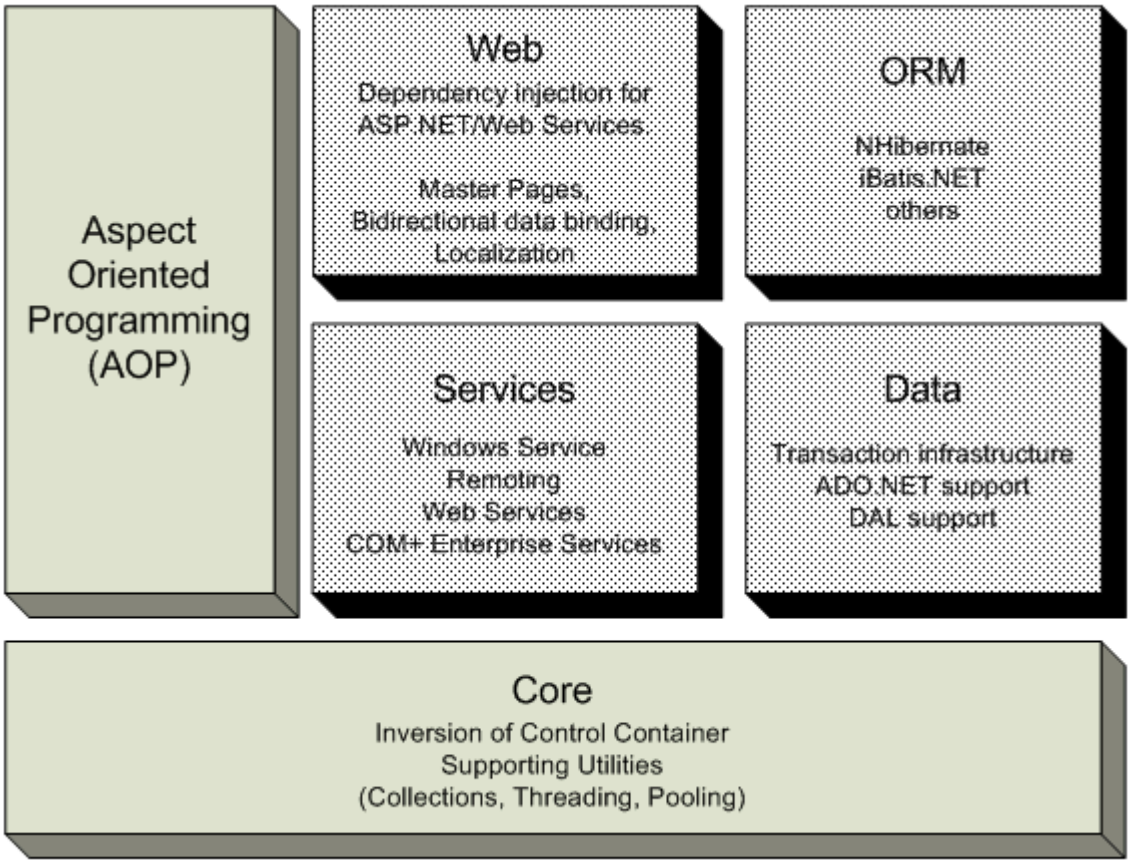
Spring.NET 的 IoC 容器所解决的，正是如何在企业应用中将类、对象和服务组合成应用程序的问题。IoC 容器通过很正统（按：formalized，言下之意是这些方式都是已经定型了的、经过了业界多年考验的）的方式将分散的组件组合成完整的应用程序。Spring.NET 框架所采用的，都是被业界无数应用程序考验多年的、已经被定型为设计模式的最佳编程方式，实际上，这些模式已经成为我们架构和开发时的法典，而通过 Spring.NET，我们可以直接将它们整合到自己的应用程序中。目前已有很多组织和机构用 Spring 框架开发出了强壮的、维护性好的应用程序，这确实是一件非常好的事情。

2.2. 背景

在 2004 年初，Martin Fowler 曾经问他网站的读者：当我们谈到控制反转时，“问题是，它们反转的是哪方面的控制？”。随后 Fowler 建议为控制反转重新命名（或者起码给它一个更具自我描述性的名字），所以依赖注入（Dependency Injection）这一术语才得以使用。Fowler 在论文中也讨论了控制反转和依赖注入原理背后的一些概念。如果您需要了解正宗的 IoC 和 DI 理论，可以参考这篇论文：http://martinfowler.com/articles/injection.html。

2.3. 模块

Spring.NET 框架包括很多功能，这些功能被很好的组织进一系列模块当中，如下图所示。Spring.NET 1.0 包括完整的 IoC 容器和 AOP 类库。1.1 版将加入 Web、ORM 和数据模块。Spring.NET 的下载包中并不包含与其它类库（如 NHibernate，TIBCO EMS，Anthem，和 IIOP.NET）集成的模块，如果需要您可以单独下载。下图为 Spring.NET 的各个核心模块。灰色阴影部分在 1.0 版中已经可用了，其它模块会在未来版本中陆续发布。目前可以从我们的网站上单独下载这些新的模块。



Spring.Core 作为整个框架的基础，实现了依赖注入的功能。Spring.NET 的大部分模块都要依赖或扩展该模块。Spring.Core 的基础是 IObjectFactory 接口，该接口用一个简单而优雅的方式实现了工厂模式，使我们可以无需自行编写 singleton 类型和众多的服务定位器，并允许将对象配置及其依赖关系与具体的程序逻辑解耦。该模块中的 IApplicationContext 接口是 IObjectFactory 的扩展，增加了诸多企业级功能，包括使用资源文件进行文本本地化、事件传播和资源装载等等。

Spring.AOP 为业务对象提供面向方面编程（AOP）的支持。AOP 完善了 IoC 容器的功能，为创建企业应用和使用声明式服务奠定了坚实的基础。

Spring.Web 对 ASP.NET 进行了一系列功能扩展，包括对 ASP.NET 页面进行依赖注入、双向数据绑定、在 ASP.NET 1.1 中使用 Master page、以及增强的本地化功能支持等。

Spring.Services 允许将任意的“普通”对象（意为没有继承任何指定基类型的对象）发布为企业服务（COM+）或远程对象。通过依赖注入和特性元数据覆盖等功能，该模块可使.NET 的 Web 服务获得极大的灵活性。同时也支持 Windows 后台服务。

Spring.Data 定义了一个抽象的数据访问层，可以跨越各种数据访问技术（从 ADO.NET 到各种 ORM）进行数据访问。该模块包含一个 ADO.NET 的抽象层，减少了使用传统 ADO.NET 进行编码和事务管理时的工作量。

Spring.ORM 为时下流行的 ORM 类库提供了一个整合层，其中包含声明式事务管理等诸多功能。

本文是对 Spring.NET 功能的参考性指南。因为本文尚未完成，所以，如果您有什么问题或需要，请在我们的用户论坛上发帖，网址为 forum.springframework.net。最新版的文档可以从这里下载。

2.4. 许可证信息

Spring.NET 使用 Apache 许可证 2.0 版的条款。该许可证的全部内容请参看 http://www.apache.org/licenses/LICENSE-2.0，或阅读 Spring.NET 根目录下的 license.txt 文档。

2.5. 支持

Interface21 提供相关的培训和支持，您可以从 Spring.NET 网站的邮件列表和支持论坛找到部分信息。

第三章. 背景

3.1. 控制反转(Inversion of Control)

在 2004 年初，Martin Fowler 曾经问他网站的读者：当我们谈到控制反转时，“问题是，它们反转的是哪方面的控制？”。随后 Fowler 建议为控制反转重新命名（或者起码给它一个更具自我描述性的名字），所以依赖注入（Dependency Injection）这一术语才得以使用。在 Fowler 的论文中，也讨论了控制反转和依赖注入原理背后的一些概念。

第四章. 对象、对象工厂和应用程序上下文

4.1. 简介

(Available in 1.0)

Spring.Core 程序集是 Spring.NET 控制反转（IoC，也叫做依赖注入）功能的基础(可参见 3.1 节,控制反转，其中提到了一些相关的参考资源)。Spring.Core

程序集中的 IObjectFactory 接口为 Spring.NET 提供了一种高级的配置机制，可用所有可能的存储介质保存任意对象的配置信息。同位于此程序内的 IApplicationContext 接口则扩展了 IObjectFactory，增加了面向方面编程（AOP）和消息资源处理（用于国际化）等功能。

简单的说，IObjectFactory 接口提供了配置框架和基本功能，IApplicationContext 接口又在其基础上扩展了许多企业级特性。可以说 IApplicationContext 是 IObjectFactory 的一个超集，具备 IObjectFactory 所有的功能与行为。

本章分为两部分，第一部分介绍 IObjectFactory 和 IApplicationContext 接口共通的基本原理，第二部分则专门讨论 IApplicationContext 接口所特有的功能。

Spring.NET 或 IoC 容器的新用户可以考虑从第二十七章, IoC 快速入门看起，其中有一些入门级的例子实际上是本章许多内容的示例。如果一开始理解不了，也不要着急——这些例子只是用来让读者了解 Spring.NET 是如何与实际开发相结合的，看完了这些例子，再回头阅读本章的内容就会容易的多了。

4.2. IObjectFactory,IApplicationContext 和 IObjectDefinition 接口介绍

4.2.1. IObjectFactory 和 IApplicationContext

IObjectFactory 是初始化、配置及管理对象的实际容器（按：它是所有容器的父接口）。对象间通常会相互协作，我们也可以说它们相互间具有依赖性。这些依赖性通过 IObjectFactory 的配置数据反映出来。（但某些依赖性从配置数据中是看不到的，比如运行时对象之间的方法调用。）

Spring.Objects.Factory.IObjectFactory 接口有多个实现类。最常用的是 Spring.Objects.Factory.Xml.XmlObjectFactory。关于如何在代码中与 IObjectFactory 交互，请参见 4.7, 与 IObjectFactory 交互。IApplicationContext 接口所定义的增强功能将在 4.11, IApplicationContext 简介中讨论。

前文（按：第一章）提到过，Spring.NET 框架的核心原则是非侵入性。简单的说，就是应用程序的代码不需要对 Spring.NET 的 API 有任何依赖。然而，如果要通过 IObjectFactory 或 Spring.Context.IApplicationContext 接口充分利用 IoC 容器的功能，有时候还**必须**要初始化这两个接口的某个实现类。为此，可以在代码中使用 new 操作符来显式创建容器（在 C#中，VB 开发人员则使用 New 操作符。按：后文中，举凡涉及到含义为“管理对象的容器”而非特指接口的名称时，将原文中的 IObjectFactory 或 IApplicationContext 称为“容器”或“IoC 容器”）；另一种更为简单的方式是在.NET 应用程序的标准配置文件中用一个

自定义节点来配置容器。一旦容器建立，应用程序代码就可能不再需要与之发生显式的交互了。

下面代码创建了 XmlObjectFactory 类的一个实例，XmlObjectFactory 是 IObjectFactory 的实现类之一。我们假定在 objects.xml 文件中定义了要装配（按：装配的概念见后文）和发布的服务对象。将该文件的信息传递给 XmlObjectFactory 的构造器，即可创建一个容器，参见如下代码：

```
[C#]
IResource input = new FileSystemResource ("objects.xml");
IObjectFactory factory = new XmlObjectFactory(input);
```

代码中使用了 Spring.NET 的 IResource 接口。IResource 能以简单统一的方式访问许多可用 System.IO.Stream 表示的 IO 资源。在第六章， IResource 接口中将对 IResource 接口展开讨论。这些 IO 资源一般是独立的文件或者 URL，但也可以是.NET 程序集的内嵌资源。通过 IResource 接口，可以用简单的 URI 格式来描述资源的位置，比如可用 file://object.xml 来表示一个文件。此外，IResource 也支持很多其它协议，如 http 等。

前文提到 IApplicationContext 是 IObjectFactory 的超集，我们一般都会用 IApplicationContext 来作为容器。在创建容器时可以像上例一样用 IResource 实例化 IApplicationContext 接口的任何一个实现类。另外，IApplicationContext 支持用多个配置文件创建容器（按：此处的配置文件是指包括了 Spring.NET 对象定义的 XML 文件，而非特指.config 文件，下同）：

```
IApplicationContext context = new XmlApplicationContext(
        "file://objects.xml",
        "assembly://MyAssembly/MyProject/objects-dal-layer.xml");

// of course, an IApplicationContext is also an IObjectFactory...
IObjectFactory factory = (IObjectFactory) context;
```

下面是引用.NET 程序集内嵌资源时的 URI 语法：
assembly://<AssemblyName>/<NameSpace>/<ResourceName>

> **注意**
>
> 若要在 VS 中创建一个内嵌的资源，必须在文件属性编辑器中将 xml 文件的 Build Action 设为 Embedded Resource。并且，如果自上次成功建立项目之后，该属性的变更是本次所做的唯一更改，则需要显式的重新生成项目。如果使用 NAnt 建立项目，需要在 csc 任务中添加一个<resources>节点。可参见 Spring.Core.Tests 项目中的 build 文件，该项目随 Spring.NET 一起发布。

更好的创建方式是在标准.NET 应用程序配置文件中(App.config 或 Web.config)添加自定义配置节点。以下的 XML 节点可以创建与前例相同的容器：

```
<spring>
  <context type="Spring.Context.Support.XmlApplicationContext,
Spring.Core">
    <resource uri="file://objects.xml"/>
    <resource
uri="assembly://MyAssembly/MyProject/objects-dal-layer.xml"/>
  </context>
</spring>
```

<context>节点的 type 属性是可选的，在 Windows 应用中，其默认值就是 Spring.Context.Support.XmlApplicationContext，所以下面的配置和上面完全相同：

```
<spring>
  <context>
    <resource uri="file://objects.xml"/>
    <resource
uri="assembly://MyAssembly/MyProject/objects-dal-layer.xml"/>
  </context>
</spring>
```

spring 和 context 节点的名称不是任意的，必须是"spring"和"context"，Spring.NET 本身将"spring/context"作为字符串常量定义在了 AbstractApplicationContext 类中以表示上下文的节点名称。若要引用由以上配置创建的容器，可使用下面的代码：

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

ContextRegistry 类既可用来初始化应用程序上下文，也可用来以服务定位器风格对容器中的对象进行访问（4.16 节,服务定位器访问会详细讨论有关服务定位器访问的内容）。注意，使这一切成为可能的是 Spring.Context.Support.ContextHandler 类，该类实现了 FCL 的 IConfigurationSectionHandler 接口。必须在.NET 配置文件的<configSections>节点中注册这个类，如下所示：

```
<configSections>
  <sectionGroup name="spring">
    <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core"/>
  </sectionGroup>
</configSections>
```

注册了这个节点处理器后，配置文件中的<spring>节点才能起作用。

在某些情况下，用户不需要以任何方式显式创建容器，Spring.NET 可以自行创建。例如，Spring.NET 中的 Spring.Web 模块可以将 IApplicationContext 作为 Web 应用程序正常启动进程的一部分进行自动装载。目前正在研究如何为 WinForms 应用程序提供类似的支持。（按：请参考 19.2.1 节，以了解 Spring.Web 是如何自动创建容器的。）

关于如何通过编程方式管理 IObjectFactory，稍后再讲。下面我们先讨论 IObjectFactory 接口所使用的对象配置格式。（按：凡适用于 IObjectFactory 的内容，必定也适用于 IApplicationContext）

基本上，IObjectFactory 的配置信息由一个或多个对象定义构成。在基于 XML 的工厂中，这些对象定义表现为一个或多个<object>子节点，它们的父节点必须是<objects>（按：objects 节点的 xmlns 元素是必需的，必须根据不同的应用添加不同的命名空间，请留意各个章节中的相关内容。

```
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net

        http://www.springframework.net/xsd/spring-objects.xsd">
  <object id="..." type="...">
  ...
  </object>
  <object id="...." type="...">
  ...
  </object>

  ...
</objects>
```

随 Spring.NET 一起发布的 schema 文档可以简化对 XML 对象定义的验证过程。该文档是完全开放的（参见附录 A, Spring.NET 的 spring-objects.xsd）。目前，除了验证 XML 文档外，该文档还有一个用途，就是在具备 XSD 感知能力的编辑器（比如 VS.NET）内进行代码提示。可参考第二十四章, 与 Visual Studio.NET 集成。在 Spring.NET 的网站上可以下载到 spring-objects.xsd 的最新版本。

XML 对象定义也可以放在.NET 的标准应用程序配置文件中。此时也需要为<objects>节点预先注册相应的节点处理器，类型为 Spring.Context.Support.DefaultSectionHandler。然后，就可以在.NET 的.config 文件中为一或多个容器配置对象定义了，如下所示：

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
```

```
      <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>

  </configSections>

  <spring>

    <context>
      <resource uri="config://spring/objects"/>
    </context>


    <objects xmlns="http://www.springframework.net">
        ...
    </objects>

  </spring>

</configuration>
```

4.13.1,上下文嵌套和 4.15,从其它文件中导入对象定义会讨论创建配置文件时可用的其它节点。

我们也可以在配置文件中为 IApplicationContext 注册自定义的资源处理器、schema 解析器、类型转换器和类型别名等等。在 4.12,IApplicationContext 配置一节中将讨论这些内容。

4.2.2.对象定义

容器所管理的对象由对象定义来配置，一个对象定义包含以下信息：

- 对象类型，即所定义对象的实际类型。
- 对象行为，用来规定对象在 IoC 容器中的行为（例如，是否布署为 singleton，自动装配的模式，依赖检查的模式，初始化和销毁方法等）。
- 对象创建后要设置的属性值。例如，一个线程池管理对象的可用线程数，或者用来创建线程池的类型信息，都可以通过属性或构造器参数进行设置。
- 对象所需要的其它对象，例如一个对象的协作对象（同样可通过属性或构造器设置）。这些对象也可以叫做依赖项。

上面提到了用属性和构造器参数来设置依赖项。Spring.NET 支持两种类型的 IoC：类型 2 和类型 3（分别是构造器参数注入和属性注入）。也就是说，当一个对象

被 IoC 容器创建时，既可以使用常规的属性设值方法为属性设值，也可以直接向构造器传递参数来为属性赋值。（按：对.NET 来说，“属性注入”似乎比“设值方法注入”更贴切）

上述概念直接对应对象定义中的一系列 xml 子节点，这些节点及相关的章节如下表所示：

表 4.1.

对象定义内容

内容	详细信息
对象类型	4.2.3,对象的创建
id 和 name	4.2.5,对象标识符（id 和 name）
singleton 或 prototype	4.2.6,Singleton 和 Prototype
对象属性	4.3.1,设置对象的属性和协作对象
构造器参数	4.3.1,设置对象的属性和协作对象
自动装配模式	4.3.8,自动装配协作对象
依赖检查模式	4.3.9,检查依赖项
初始化方法	4.5.1,生命周期接口
销毁(destruction)方法	4.5.1,生命周期接口

4.2.3.对象的创建

对象定义会包含对象的类型信息（也有例外，参见 4.2.3.3,通过实例工厂方法创建对象和 4.6,抽象及子对象定义）。多数情况下，容器会根据对象定义中的 type 属性值去直接调用相应类型的某个构造器。另外，容器也可以调用工厂方法来创建对象，这时 type 属性的值就应该是包含工厂方法的类型（按：而不是要创建的类型，但通过该对象定义的名称获取的则是由工厂方法所创建的对象）。工厂方法的产品对象可以是工厂方法所在的类型，也可以是其它类型（按：很多情况下工厂方法位于单独的类型中），这无关紧要。

4.2.3.1.通过构造器创建对象

使用构造器创建对象时，并不要求对象必须是某种特定的类型，也不需要了解它的实现方式（按：也就是说，类型不必去实现某个接口或扩展某个基类以求和 Spring.NET 兼容，任何对象都可以布署在容器中）。只要指明对象类型（以及它所在的程序集名称）就可以了。不过，根据不同 IoC 容器的要求，可能需要为

类型（显式的）定义默认构造器（即无参的构造器）。（按：由于.NET 只会为没有构造器的类型自动添加默认构造器，所以 Spring.NET 允许类型不定义任何构造器；但如果在定义了含参构造器后仍需使用无参构造器，则必须进行显式定义。）

XmlObjectFactory 类实现了 IObjectFactory 接口，它可以处理 XML 文件中的对象定义，例如：

```
<object id="exampleObject"
      type="Examples.ExampleObject, ExamplesLibrary"/>
```

这个节点定义了一个名为 exampleObject 的对象，它的类型是位于 ExamplesLibrary 程序集中的 Examples.ExampleObject 类。请特别留心一下 type 属性的格式：类型的全名，然后是一个逗号，最后是类型所在的程序集名称。在上面的例子中，ExampleObject 类定义在 Examples 命名空间，且位于 ExamplesLibrary 程序集中。

type 属性值必须包含类型所在的程序集名称。另外，如果需要确保 Spring.NET 能按照预期的类型创建对象，则推荐使用程序集的强命名。不过，一般只有在用到 GAC 中的程序集时，才需要使用强命名。（按，例如，type="System.Windows.Forms.Form, System.Windows.Forms, Version=2.0.0.0, Culture=neutral,PublicKeyToken=b77a5c561934e089"）

如果需要为嵌套类型创建对象，可以使用+号。例如，如果在类型 Examples.ExampleObject 嵌套定义了类型 Person，就可以用下面的方式创建对象定义：

```
<object id="exampleObject"
      type="Examples.ExampleObject+Person, ExamplesLibrary"/>
```

如果应用程序能够以标准的程序集探测机制访问程序集（例如 ASP.NET 中的 bin 文件夹），那么 type 属性的值只需包括类型全名即可。这样，当程序集改变后，不需要去修改每个对象定义的 type 属性（主要是改些版本号等等），Spring.NET 就会自动使用最新的程序集。

4.2.3.2.通过静态工厂方法创建对象

在使用静态工厂方法创建对象时，除了要将对象定义的 type 属性设为包含静态工厂方法的类型外，还要设置一个名为 factory-method 的属性来指定静态工厂方法的名称。Spring.NET 会调用这个方法（稍后会看到，静态工厂方法还可以包含一个可选的参数表）来创建对象，结果和通过构造器创建对象是一样的。在实际项目中，可以用这种方法去调用原有代码中的静态工厂。

下面的对象就是通过静态工厂方法创建的。注意：对象定义中的 type 并非是要创建的对象的类型，而是包含了工厂方法的类型；同时，CreateInstance 必须是静态方法。

```
<object id="exampleObject"
 type="Examples.ExampleObjectFactory, ExamplesLibrary"
 factory-method="CreateInstance"/>
```

稍后会简单的讨论如何为工厂方法配置（可选的）参数，以及如何设置产品对象的属性。（按：见 4.3.1,设置对象的属性和协作对象一节的最后一部分）

4.2.3.3.通过实例工厂方法创建对象

通过实例工厂方法创建对象与通过静态工厂方法创建对象的配置方式类似。此时，实例工厂方法所在的对象必须也要配置在同一容器（或父容器）中。

如果要通过实例工厂方法创建对象，对象定义就不能包含 type 属性，而要用 factory-object 属性引用工厂方法所在的对象；注意，该属性值必须是包含工厂方法的对象的名称，且该对象必须定义在当前容器或父容器中。工厂方法的方法名则通过 factory-method 属性指定。

请看下面的例子：

```
<!-- the factory object, which contains an instance method called
'CreateInstance' -->
<object id="exampleFactory" type="..."/>
<!-- the object that is to be created by the factory object -->
<object id="exampleObject"
      factory-method="CreateInstance"
      factory-object="exampleFactory"/>
```

虽然我们还没有讲过依赖注入机制，但请记住，为对象定义设置 factory-object 属性这一行为就是依赖注入。工厂对象本身也是由容器管理并配置的对象。

4.2.4.泛型类的对象创建

泛型对象的创建方法和普通对象是一样的。

4.2.4.1.通过构造器创建泛型对象

下面是一个泛型类的代码：

```
namespace GenericsPlay
{
    public class FilterableList<T>
    {
        private List<T> list;

        private String name;

        public List<T> Contents
        {
            get { return list; }
            set { list = value; }
        }

        public String Name
        {
            get { return name; }
            set { name = value; }
        }

        public List<T> ApplyFilter(string filterExpression)
        {
            /// should really apply filter to list ;)
            return new List<T>();
        }

    }
}
```

下面是该类的对象定义：

```
<object id="myFilteredIntList"
type="GenericsPlay.FilterableList&lt;int>, GenericsPlay">
  <property name="Name" value="My Integer List"/>
</object>
```

在为泛型类对象指定 type 属性的时候要注意：第一，左尖括号<要替换成字符串“<”，因为在 XML 中左尖括号会被认为是小于号。从可读性来讲，我们都知道这并不是理想的方式。第二，type 参数值中不能包含程序集的名称，因为程序集名称要求和类型全名用逗号隔开，而在这里逗号已经被用来分隔泛型类的类型参数了。将来可能会用其它字符代替这两个符号，但目前还没找到更具可读性的方案。若要提高可读性，建议使用类型别名，如下所示：

```
<typeAliases>
```

```
 <alias name="GenericDictionary" type="
System.Collections.Generic.Dictionary&lt;,>" />
 <alias name="myDictionary"
type="System.Collections.Generic.Dictionary&lt;int,string>" />
</typeAliases>
```

然后，下面的对象定义：

```
<object id="myGenericObject"

type="GenericsPlay.ExampleGenericObject&lt;System.Collections.Generic
.Dictionary&lt;int , string>>, GenericsPlay" />
```

就可以缩短为：

```
<object id="myOtherGenericObject"

type="GenericsPlay.ExampleGenericObject&lt;GenericDictionary&lt;int ,
string>>, GenericsPlay" />
```

或者更短：

```
<object id="myOtherOtherGenericObject"

type="GenericsPlay.ExampleGenericObject&lt;MyIntStringDictionary>,
GenericsPlay" />
```

关于类型别名，可以参考 4.12,配置 IApplicationContext 。

4.2.4.2.通过静态工厂方法创建泛型类

请看一个例子。下面是一个泛型类：

```
public class TestGenericObject<T, U>
{
    public TestGenericObject()
    {
    }

    private IList<T> someGenericList = new List<T>();

    private IDictionary<string, U> someStringKeyedDictionary =
        new Dictionary<string, U>();
```

```
    public IList<T> SomeGenericList
    {
        get { return someGenericList; }
        set { someGenericList = value; }
    }

    public IDictionary<string, U> SomeStringKeyedDictionary
    {
        get { return someStringKeyedDictionary; }
        set { someStringKeyedDictionary = value; }
    }

}
```

对应的工厂类为：

```
public class TestGenericObjectFactory
{
    public static TestGenericObject<V, W> StaticCreateInstance<V, W>()
    {
        return new TestGenericObject<V, W>();
    }

    public TestGenericObject<V, W> CreateInstance<V, W>()
    {
        return new TestGenericObject<V, W>();
    }
}
```

使用静态工厂方法创建 TestGenericObject 对象的对象定义如下，其中 V 是一个整型 List，W 是整型：

```
<object id="myTestGenericObject"
        type="GenericsPlay.TestGenericObjectFactory, GenericsPlay"

factory-method="StaticCreateInstance&lt;System.Collections.Generic.List&lt;int>,int>"/>
```

StaticCreateInstance 方法的职责就是创建名为“myTestGenericObject”的对象。

4.2.4.3.通过实例工厂方法创建泛型对象

以下是使用实例工厂方法创建泛型对象的对象定义：

```
<object id="exampleFactory"
type="GenericsPlay.TestGenericObject&lt;int,string>, GenericsPlay"/>

<object id="anotherTestGenericObject"
        factory-object="exampleFactory"

factory-method="CreateInstance&lt;System.Collections.Generic.List&lt;
int>,int>"/>
```

这样就可以创建一个类型为 TestGenericObject<List<int>,int>的对象。

4.2.5. 对象标识符（id 和 name）

每个对象都有一个或多个 id（就是所谓的标识符或名称，这些术语的含义是相同的）。id 在容器中应该是唯一的。一个对象通常只有一个标识符，如果指定了一个以上名称，其余的就会被认为是别名。

在 XML 对象定义中，用 id 或者 name 属性来定义对象的标识符。每个对象都需要用 id 或 name 属性定义至少一个标识符。id 属性允许为对象定义指定一个唯一的 id，因为在 Spring.NET 的 shcema 文档中，id 被标识为 XML 元素的 ID 属性，XML 解析器可以在其它元素引用它的时候进行验证，在配置对象标识符时，应该优先使用 id 属性。但是，id 属性值不能包含任何 XML ID 不允许使用的字符。如果一定要使用这些字符，应该使用 name 属性，在 name 属性中也可以通过逗号或分号为对象指定一个或多个别名。

4.2.6.Singleton 和 Prototype

对象可以通过两种模式布署：singleton 和非 singleton（或者叫做 prototype，只是用在这里不是很合适）。当一个对象被定义为 singleton 时，容器中就只会有一个共享的实例，任何时候通过 id 或别名请求该对象都会返回这个共享实例的引用（也就是说这个对象只会被创建一次）。

当使用非 singleton，或者说原型模式布署时，每次请求对象都会创建新的实例。在某些场合，如果需要为每个用户返回单独的用户对象或其它对象，非 singlton 布署模式就比较理想。

如果没有显式指定，对象的布署模式默认为 singleton。注意非 singleton（原型）模式会使 Spring.NET 在每次请求对象时都创建新的实例，这也许并非是我们预期的行为。所以，除非绝对需要，否则不要使用原型模式。

注意

当使用原型模式布署对象时，对象生命周期的改变（按：对 Spring.NET 来说）就没那么明显了。Spring.NET 无法通过对象定义来完全管理非

singleton（原型）对象的生命周期，因为这些对象一经创建就交由客户代码维护，容器不可能再继续跟踪它们。可以将 Spring.NET 的非 singleton（原型）创建方式看做是 new 操作符的替代物。通过这种方式创建的任何对象，其生命周期都只能由客户代码管理。关于容器内对象的生命周期，可参见 4.5.1,生命周期接口

下面的两个对象分别用 singleton 和 prototype 模式布署。对象 exampleObject 是一个原型对象，每次客户代码请求时，容器都会创建一个新的实例；而对象 yetAnotherExample 是一个 singleton 对象，只会创建一次，每次请求返回的都是同一个实例。

```
<object id="exampleObject" type="Examples.ExampleObject,
ExamplesLibrary"
  singleton="false"/>
<object name="yetAnotherExample" type="Examples.ExampleObjectTwo,
ExamplesLibrary"
  singleton="true"/>
```

可使用<object>节点的 lazy-init 属性控制 singleton 对象的创建时机。singleton 设计模式经常会涉及到对象的惰性创建，也就是说只在某对象第一次使用时去创建它。默认情况下，singleton 对象的 lazy-init 属性值为 false，IoC 容器在初始化的时候就会创建它们。将该属性设置为 true，就可将对象的创建推迟到第一次使用前，这里，“第一次使用”可能是客户代码的请求，也可能是容器在处理对象的依赖注入。

4.3.属性，协作对象，自动装配和依赖检查

4.3.1.设置对象的属性和协作对象

控制反转也称为依赖注入。其基本理论是对象应该只通过构造器参数、工厂方法参数或者属性来定义自己的依赖项（即和它一起工作的其它对象）。这样，在创建对象时，将这些依赖项注入到对象内部就是容器要做的工作了。这就从根本上反转了对象在（通过 new 操作符）创建时自行初始化或者（通过类似于服务定位器模式的方法）查找依赖项的行为（所以叫控制反转）。在此我们不打算花太多的篇幅去讲依赖注入的优点，因为这是很明显的。由于对象不需要自己查找依赖项，甚至不需要知道依赖项的位置或具体类型，只需要由容器将依赖项注入给它，这就很容易让代码变得更加简洁，并且做到高度解耦。

前文提到过，控制反转（依赖注入）主要有两种方式：

- 属性注入（按：即设值方法注入，*setter-based* dependency injection，但对于.NET 来说，恐怕称为属性注入更为合适）：在创建对象以后，通过（调用）属性（的设值方法）将依赖项注入。Spring.NET 建议使用属

性注入，因为构造器的参数如果太多的话，会使类的代码和对象定义变得很臃肿，特别是在某些属性为可选的时候（即不一定非要注入）。

- 构造器注入（*constructor-based* dependency injection）：调用含参构造器，在对象创建时将依赖项通过构造器参数注入。虽然 Spring.NET 建议尽量使用属性注入，但也完全支持构造器注入，因为有时候我们只能使用具有含参构造器且没有定义属性的旧类型。另外，对比较简单的对象来说，某些人可能更喜欢使用构造器注入，以确保对象在创建以后马上处于有效的状态。

IObjectFactory 接口支持这两种注入方式。在容器中，我们通过 XML 对象定义来配置依赖项，容器会在必要时使用类型转换器进行转型。

一般来说，在处理对象依赖项的时候，Spring.NET 会做以下几件事情：

1. 根据包含对象定义的配置信息来创建和初始化容器。大多数用户都会使用支持 XML 配置文件的 IObjectFactory 或 IApplicationContext 实现类作为容器。
2. 每个对象的依赖项都通过属性或构造器配置在对象定义中。当对象被创建时，容器会将依赖项注入给对象。
3. 属性或构造器参数既可以设置为实际的值，也可以设为同一容器中其它对象的引用。如果用 IApplicationContext 作为容器，也可以引用父容器中的对象。
4. 配置给属性或构造器参数的值必须能够转型为属性或参数的实际类型。默认情况下，Spring.NET 可以将字符串值转型为任意基元类型，如 int，long，bool 等。另外，基于 XML 的容器也可以使用 XML 节点配置 IList、IDictionary 和 Set 等集合类型的值，Spring.NET 会使用 TypeConverter 将字符串值转换为其它任意类型。可参考 5.3,类型转换以了解 TypeConverter 的详细信息，以及使用 Spring.NET 自动转换自定义类型的方法。（按：Set 是 Spring.NET 提供的集合类型）
5. 在容器本身被创建的时候，Spring.NET 会验证容器内每个对象的配置信息，并验证该对象的依赖项也是有效的（即：对象引用的对象也要定义在同一 IObjectFacotry 中，对于 IApplicationContext，也可以定义在父容器中）。但是属性的赋值仅在对象被创建时才会发生。对于一个以 singleton 模式布署、非惰性创建的对象（比如定义在 IApplicationContext 中的 singleton 对象）来说，对象的创建就发生在容器本身被创建的时候；否则（按：惰性创建，或非 singleton 时）就发生在对象被请求的时候。当一个对象被创建时，可能会导致其它一系列对象同时被创建，因为对象的依赖项，以及依赖项的依赖项此时都需要被创建并赋值。
6. 一般情况下 Spring.NET 可以把这些事情做的很好。在载入容器时，Spring.NET 会处理配置中出现的问题，比如引用了一个不存在的对象定义或发生循环依赖等等。而属性的设置和依赖项的解析（即在需要时创建所有依赖项的行为）则会推迟到对象实际被创建时才会执行。也就是说，如果某个对象或其依赖项无法正确创建，那么即便是容器可以正常创建，在请求这个对象时也会抛出异常。例如，如果遗漏了本该赋值的属性，或

者所赋予的属性值无效时，会抛出异常而导致该对象无法正确创建。这也是 IApplicationContext 使用非惰性 singleton 模式作为默认布署方式的原因。所有对象都在实际需要前被创建，这种时间和空间上的开销可以保证 IApplicationContext 在创建时及早发现问题。如果愿意，可随时覆盖这一默认行为，将任意对象设置为惰性创建。

下面是一些 XML 对象定义的例子...

首先看一个使用 IoC 容器进行属性注入的例子。下面是 XML 对象定义；随后是相关类型的代码。

```
<object id="exampleObject" type="Examples.ExampleObject,
ExamplesLibrary">
    <property name="objectOne" ref="anotherExampleObject"/>
    <property name="objectTwo" ref="yetAnotherObject"/>
    <property name="IntegerProperty" value="1"/>
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject,
ExamplesLibrary"/>

<object id="yetAnotherObject" type="Examples.YetAnotherObject,
ExamplesLibrary"/>
```

（按：原文的许多例子中，XML 对象定义所使用的属性名以小写字母开头，这是由于 IoC 容器这部分文档的很多内容都直接来自 Spring.Java。在译文中，按照 C#的命名习惯将属性名或方法名的第一个字母改成了大写。）

```
[C#]
public class ExampleObject
{
  private AnotherObject objectOne;
  private YetAnotherObject objectTwo;
  private int i;

  public AnotherObject ObjectOne
  {
    set { this.objectOne = value; }
  }

  public YetAnotherObject ObjectTwo
  {
    set { this.objectTwo = value; }
  }
```

```
  public int IntegerProperty
  {
    set { this.i = value; }
  }
}
```

下面的例子使用 IoC 容器进行第三类 IoC（即构造器参数注入）。首先是 XML 对象定义，随后是类型声明：

```
<object id="exampleObject" type="Examples.ExampleObject,
ExamplesLibrary">
    <constructor-arg name="objectOne" ref="anotherExampleObject"/>
    <constructor-arg name="objectTwo" ref="yetAnotherObject"/>
    <constructor-arg name="IntegerProperty" value="1"/>
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject,
ExamplesLibrary"/>

<object id="yetAnotherObject" type="Examples.YetAnotherObject,
ExamplesLibrary"/>
[Visual Basic.NET]
Public Class ExampleObject

    Private myObjectOne As AnotherObject
    Private myObjectTwo As YetAnotherObject
    Private i As Integer

    Public Sub New (
        anotherObject as AnotherObject,
        yetAnotherObject as YetAnotherObject,
        i as Integer)

        myObjectOne = anotherObject
        myObjectTwo = yetAnotherObject
        Me.i = i
    End Sub
End Class
```

在本例中，配置在 XML 对象定义中的构造器参数会在创建对象时传递给 ExampleObject 类的构造器。

注意属性注入（类型 2）和构造器注入（类型 3）并不排斥，完全可以在同一对象定义中同时使用，如下所示：

```
<object id="exampleObject" type="Examples.MixedIocObject,
ExamplesLibrary">
    <constructor-arg name="objectOne" ref="anotherExampleObject"/>
    <property name="objectTwo" ref="yetAnotherObject"/>
    <property name="IntegerProperty" value="1"/>
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject,
ExamplesLibrary"/>

<object id="yetAnotherObject" type="Examples.YetAnotherObject,
ExamplesLibrary"/>
[C#]
public class MixedIocObject
{
  private AnotherObject objectOne;
  private YetAnotherObject objectTwo;
  private int i;

  public MixedIocObject (AnotherObject obj)
  {
    this.objectOne = obj;
  }

  public YetAnotherObject ObjectTwo
  {
    set { this.objectTwo = value; }
  }

  public int IntegerProperty
  {
    set { this.i = value; }
  }
}
```

现在，考虑一种构造器的替代方案。下面例子中，Spring.NET 使用静态工厂方法来创建对象：

```
<object id="exampleObject" type="Examples.ExampleFactoryMethodObject,
ExamplesLibrary"
     factory-method="CreateInstance">
    <constructor-arg name="objectOne" ref="anotherExampleObject"/>
    <constructor-arg name="objectTwo" ref="yetAnotherObject"/>
    <constructor-arg name="intProp" value="1"/>
```

```
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject,
ExamplesLibrary"/>

<object id="yetAnotherObject" type="Examples.YetAnotherObject,
ExamplesLibrary"/>
[C#]
public class ExampleFactoryMethodObject
{
  private AnotherObject objectOne;
  private YetAnotherObject objectTwo;
  private int i;

  // a private constructor
  private ExampleFactoryMethodObject()
  {
  }

  public static ExampleFactoryMethodObject
CreateInstance(AnotherObject objectOne,
                               YetAnotherObject objectTwo,
                               int intProp)
  {
    ExampleFactoryMethodObject fmo = new ExampleFactoryMethodObject();
    fmo.AnotherObject = objectOne;
    fmo.YetAnotherObject = objectTwo;
    fmo.IntegerProperty = intProp;
    return fmo;
  }

  // Property definitions

}
```

注意在对象定义中，静态工厂方法所需要的参数也通过 constructor-arg 节点配置，和直接使用构造器是一样的。要注意工厂方法产品对象的类型不需要一定是包含工厂方法的类型。实例工厂方法的配置方法和静态工厂方法基本相同（除了要用 factory-object 属性代替 type 属性外），所以不再赘述。

4.3.2.构造器参数解析（按：构造子决议）

（按:构造器参数解析是指将对象定义中的 constructor-arg 节点与构造器的实际参数相关联）。构造器参数的解析可通过匹配参数类型来完成。当 constructor-arg 节点引用其它对象时，由于容器可以获知被引用对象的类型，所以能据此判断该节点对应构造器的哪个参数。但如果构造器参数为简单类型，就只能在 constructor-arg 节点中直接用文本来配置参数值，例如 <value>1</value>，此时 Spring.NET 无法确定文本值的类型，也就无法根据类型判断 constructor-arg 节点和具体参数的关联关系。请先看下面的类定义，后面两节中都会用到这个类：

```
using System;

namespace SimpleApp
{
  public class ExampleObject
  {
    private int years;                //No. of years to the calculate the
Ultimate Answer

    private string ultimateAnswer;  //The Answer to Life, the Universe,
and Everything

    public ExampleObject(int years, string ultimateAnswer)
    {
       this.years = years;
       this.ultimateAnswer = ultimateAnswer;
    }

    public string UltimateAnswer
    {
      get { return this.ultimateAnswer; }
    }

    public int Years
    {
      get { return this.years; }
    }
  }
}
```

4.3.2.1.根据参数类型匹配构造器参数

在配置上一节类型的对象时，可以在 constructor-arg 节点中用 type 属性显式的为参数指定类型。如下：

```
<object name="exampleObject" type="SimpleApp.ExampleObject,
SimpleApp">
  <constructor-arg type="int" value="7500000"/>
  <constructor-arg type="string" value="42"/>
</object>
```

在创建对象时，容器会根据 constructor-arg 节点的 type 属性值将 value 属性值传递给类型相同的参数，type 属性值可以是 System 中的类型名，例如 System.Int32，也可以使用别名，见下表..

表 4.2.

类型别名

类型	别名	数组别名
System.Char	char, Char	char[], Char()
System.Int16	short, Short	short[], Short()
System.Int32	int, Integer	int[], Integer()
System.Int64	long, Long	long[], Long()
System.UInt16	ushort	ushort[]
System.UInt32	uint	uint[]
System.UInt64	ulong	ulong[]
System.Float	float, Single	float[], Single()
System.Double	double, Double	double[], Double()
System.Date	date, Date	date[], Date()
System.Decimal	decimal, Decimal	decimal[], Decimal()
System.Bool	bool, Boolean	bool[], Boolean()
System.String	string, String	string[], String()

4.3.2.2.根据参数索引匹配构造器参数

使用索引匹配构造器参数要比仅使用类型进行匹配精确的多。可以显式的在 constructor-arg 节点中用 index 属性来指定参数的索引，如下：

```
<object name="exampleObject" type="SimpleApp.ExampleObject,
SimpleApp">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</object>
```

构造器参数索引匹配适用于构造器具有多个简单类型参数的情况，也可以解决构造器具有多个相同类型参数的问题。注意索引是从 0 开始的。

4.3.2.3.根据名称匹配构造器参数

最精确的方式是使用构造器参数的名称来匹配；可以在 constructor-arg 节点中使用 name 属性来指定参数的名称：

```
<object name="exampleObject" type="SimpleApp.ExampleObject,
SimpleApp">
  <constructor-arg name="years" value="7500000"/>
  <constructor-arg name="ultimateAnswer" value="42"/>
</object>
```

4.3.3.详细讨论对象属性和构造器参数

前面提到过，属性和构造器参数也可以引用其它对象（协作对象）。Spring.Objects.Factory.Xml 命名空间下的 XMLObjectFactory 类（按：及其它基于 XML 的容器类）可以通过 property 和 constructor-arg 节点的子元素来引用协作对象。

在对象定义中，用 property 和 constructor-arg 的 value 属性来为属性或构造器的参数设值。前文已经讨论过，Spring.NET 使用 TypeConverter 将字符串值转换为属性或者参数的实际类型。Spring.Objects.TypeConverters 命名空间增强了.NET 基础类库提供的标准 TypeConverter 类。

下面的例子用到了 System.Data.SqlClient 命名空间下的 SqlConnection 类。该类有一个公共属性用于设置连接字符串。同其它类型一样，很容易在 Spring.NET 对象工厂中配置它的对象：

```
<objects xmlns="http://www.springframework.net">
  <object id="myConnection"
type="System.Data.SqlClient.SqlConnection">
      <!-- results in a call to the setter of the ConnectionString
property -->
      <property
          name="ConnectionString"
```

```
          value="Integrated
Security=SSPI;database=northwind;server=mySQLServer"/>
  </object>
</objects>
```

4.3.3.1.设置空值

<null/>节点可用于设置空值。Spring.NET 将对象定义中空的 value 节点值作为空字符串处理。下面的对象定义可说明这一行为：

```
<object type="Examples.ExampleObject,
      ExamplesLibrary"> <property
      name="email"><value></value></property>
      <!-- equivalent, using value attribute as opposed to nested
      <value/> element... <property name="email"
      value=""/> </object>
```

其结果是将 Email 属性值设为""，和下面的代码是一样的：

```
exampleObject.Email = "";
```

如果需要赋空值，可以用<null/>节点，见下例：

```
<object type="Examples.ExampleObject,
      ExamplesLibrary"> <property
      name="email"><null/></property>
      </object>
```

结果是将 Email 属性设为 null，和下面的代码是一样的：

```
exampleObject.Email = null;
```

4.3.3.2.设置集合值

IList，NameValueCollection，Set，IDictionary 等类型的属性，可以用 list，name-values 和 set，dictionary 节点来设置。

```
<objects xmlns="http://www.springframework.net">
  <object id="moreComplexObject" type="Example.ComplexObject">
      <!--
      results in a call to the setter of the SomeList
(System.Collections.IList) property
      -->
```

```
      <property name="SomeList">
          <list>
              <value>a list element followed by a reference</value>

              <ref object="myConnection"/>
          </list>
      </property>
      <!--
      results in a call to the setter of the SomeDictionary
(System.Collections.IDictionary) property
      -->
      <property name="SomeDictionary">
          <dictionary>

              <entry key="a string => string entry" value="just some
string"/>
              <entry key-ref="myKeyObject" value-ref="myConnection"/>
          </dictionary>
      </property>
      <!--
      results in a call to the setter of the SomeNameValue
(System.Collections.NameValueCollection) property
      -->

      <property name="SomeNameValue">
          <name-values>
              <add key="HarryPotter" value="The magic property"/>
              <add key="JerrySeinfeld" value="The funny (to Americans)
property"/>
          </name-values>
      </property>

      <!--
      results in a call to the setter of the SomeSet
(Spring.Collections.ISet) property
      -->
      <property name="someSet">
          <set>
              <value>just some string</value>
              <ref object="myConnection"/>

          </set>
      </property>
  </object>
```

```
</objects>
```

.NET 基础类库中有很多类都定义了只读的集合类型属性（按：指没有 set 方法的属性），在“设置”这类属性时，Spring.NET 会先用 get 方法读出集合的引用，然后向集合中添加元素。也就是说，此时“设置属性”实际上是向此类属性后台的集合中添加元素（按：注意不是将另一个集合的引用注入给该属性，因为它没有 set 方法。）。

注意 Dictionary 和 Set 的值可以用下面的任一子节点进行配置：

(object | ref | idref | list | set | dictionary | name-values | value | null)

在设置集合属性时，使用 value 和 ref 节点的简短格式可以避免 XML 配置过于冗长。请参见 4.3.3.8, value 和 ref 节点的简短格式 。

请注意，Spring.NET 已经计划在未来版本中支持对 NameValueCollection 类型的属性进行设置。

4.3.3.3.设置泛型集合的值

Spring.NET 也可以为 IList<T>和 IDictionary<TKey,TValue>类型的属性设值。在对象定义中，用 element-type 属性指定 IList<T>的类型参数；用 key-type 和 value-type 分别指定 IDictionary<TKey,TValue>中键和值的类型参数。容器会自动将字符串值转换为相应类型。如果使用自定义类型作为泛型的类型参数，需要注册自定义的类型转换器，可以参考 4.4, 类型转换一节。在.NET 框架中，IList<T>和 IDictionary<TKey,TValue>的实现类分别是 System.Collections.Generic.List 和 System.Collections.Generic.Dictionary。

下面的彩票类用来示范如何设置泛型 IList 的值。

```
public class LotteryTicket
{
 List<int> list;
 DateTime date;
 public List<int> Numbers
 {
 set { list = value; }
 get { return list; }
 }

 public DateTime Date
 {
```

```
 get { return date; }
 set { date = value; }
 }
}
```

该类的对象定义如下：

```
<object
      id="MyLotteryTicket"
      type="GenericsPlay.Lottery.LotteryTicket, GenericsPlay">
      <property name="Numbers">
          <list element-type="int">
              <value>11</value>
              <value>21</value>
              <value>23</value>
              <value>34</value>
              <value>36</value>
              <value>38</value>
          </list>
      </property>

      <property name="Date" value="4/16/2006"/>
</object>
```

下面的类型稍微复杂一点，其中使用 Spring.Expressions.IExpression 接口作为 IList<T>和 IDictionary<TKey,TValue>值的类型参数。
Spring.Expressions.IExpression 有一个相应的类型转换器，
Spring.Objects.TypeConverters.ExpressionConverter，已经在 Spring.NET 中注册过了。

```
public class GenericExpressionHolder
{
 private System.Collections.Generic.IList<IExpression> expressionsList;
 private System.Collections.Generic.IDictionary<string, IExpression>
           expressionsDictionary;

 public System.Collections.Generic.IList<IExpression> ExpressionsList
 {
 set { this.expressionsList = value; }
 }

 public System.Collections.Generic.IDictionary<string,IExpression>
      ExpressionsDictionary
 {
 set { this.expressionsDictionary = value; }
```

```
 }

 public IExpression this[int index]
 {
 get { return this.expressionsList[index]; }
 }

 public IExpression this[string key]
 {
 get { return this.expressionsDictionary[key]; }
 }
}
```

该类的对象定义如下：

```
<object id="genericExpressionHolder"
 type="Spring.Objects.Factory.Xml.GenericExpressionHolder,
Spring.Core.Tests">
 <property name="ExpressionsList">
 <list element-type="Spring.Expressions.IExpression, Spring.Core">
 <value>1 + 1</value>
 <value>date('1856-7-9').Month</value>
 <value>'Nikola Tesla'.ToUpper()</value>
 <value>DateTime.Today > date('1856-7-9')</value>
 </list>
 </property>
 <property name="ExpressionsDictionary">
 <dictionary key-type="string"
  value-type="Spring.Expressions.IExpression, Spring.Core">
 <entry key="zero">
 <value>1 + 1</value>
 </entry>
 <entry key="one">
 <value>date('1856-7-9').Month</value>
 </entry>
 <entry key="two">
 <value>'Nikola Tesla'.ToUpper()</value>
 </entry>
 <entry key="three">
 <value>DateTime.Today > date('1856-7-9')</value>
 </entry>
 </dictionary>
 </property>
</object>
```

4.3.3.4.设置索引器属性

C#的索引器属性允许我们使用方括号[]来访问类型内部的集合字段。Spring.NET 可以在对象定义中为索引器属性设值，也支持重载索引器和多参数索引器。Spring.NET 使用属性表达式解析器（property expression parser）将对象定义中的文本值转换为与索引器匹配的实际类型，可参见第十章,表达式求值。下面是一个例子：

```
public class Person
{
 private IList favoriteNames = new ArrayList();
 private IDictionary properties = new Hashtable();

 public Person()
 {
 favoriteNames.Add("p1");
 favoriteNames.Add("p2");
 }

 public string this[int index]
 {
 get { return (string)favoriteNames[index]; }
 set { favoriteNames[index] = value; }
 }
 public string this[string keyName]
 {
 get { return (string) properties[keyName]; }
 set { properties.Add(keyName,value); }
 }

}
```

下面的对象定义为索引器属性赋值：

```
<object id="person" type="Test.Objects.Person, Test.Objects">
<property name="[0]" value="Master Shake"/>
<property name="['one']" value="uno"/>
</object>
```

注意

在 1.02 版中，属性表达式解析器的使用方式与旧版不同。请看下面的解释：

老的配置方式使用下面的语法：

```
<object id="objectWithIndexer" type="Spring.Objects.TestObject,
Spring.Core.Tests">
 <property name="Item[0]" value="my string value"/>
</object>
```

Item 是索引器的默认名称，也可以在索引器声明上应用 [IndexerName("MyItemName")]特性来改变索引器的标识名称，然后就可以用 MyItemName[0]配置索引器的第一个元素。

这种方法有一些限制。索引器必须是单参数且其参数类型必须可以从字符串成功转型。另外，也不支持多重索引器（按：即重载的索引器）。现在则可以用 IndexerName 特性来避开这一局限。

4.3.3.5.内联对象定义

在 property 节点内部，可以用内嵌的 object 节点为属性定义一个内联对象，这就相当于直接引用容器内的其它对象。内联对象不需要定义 id 或 name（即使是定义了，Spring.NET 也会将其忽略）。

```
<object id="outer" type="...">
 <!-- Instead of using a reference to target, just use an inner object
-->
 <property name="Target">
 <object type="ExampleApp.Person, ExampleApp">
 <property name="Name" value="Tony"/>
 <property name="Age" value="51"/>
 </object>
 </property>
</object>
```

4.3.3.6.idref 节点

如果需要将某个字符串属性设置为容器中其它对象的 id 或 name（按：注意是设置为对象 id 或 name 的字符串内容，而非引用该对象），可使用 idref 节点以避免错误。

```
<object id="theTargetObject" type="...">
</object>
<object id="theClientObject" type="...">
 <property name="TargetName">
 <idref object="theTargetObject"/>
 </property>
</object>
```

在运行时，这和下面的配置结果是一样的：

```
<object id="theTargetObject" type="...">
</object>
<object id="theClientObject" type="...">
 <property name="TargetName" value="theTargetObject"/>
</object>
```

前一种方法的好处在于，idref 节点可使 Spring.NET 在布署时验证以 theTargetObject 为名的对象定义是否存在。在后一种定义中，对象 theClientObject 必须自己来进行这一验证，并且只能在容器将其创建以后进行，而此时容器可能已经运行了相当长一段时间了。

另外，如果要引用同一 XML 文件中的对象定义名称，可以用 local 属性来代替 object 属性，这样 XML 解析器可以在解析时对目标对象进行验证，但此时引用的必须是对象的 id（按：即只能是对象定义的 id 属性值，而不能是 name 属性中定义的别名）。

```
<property name="TargetName">
 <idref local="theTargetObject"/>
</property>
```

4.3.3.7.引用协作对象

<ref/>是我们要讲的 property 节点的最后一个子节点。ref 节点用于引用容器内的其它对象（也就是所谓的协作对象）。在前面的例子中，我们曾经将 SqlConnection 的实例作为协作对象，用<ref object=""/>将其设置给了其它对象的属性。前文也提到过，被引用的对象叫做引用它的对象的依赖项，并且会在需要前被初始化(如果依赖项是 singleton 对象,则可能已经被容器初始化了)。在 XML 对象定义中，共有 3 种方法可以引用协作对象，不同的引用方法决定了目标对象的查找与验证方式。

通过 ref 节点的 object 属性来指定目标对象是最常用的方式，object 属性可以引用同一容器或父容器中的对象定义，而不管这些对象定义是否保存在同一个 XML 文件中。object 属性的值可以是目标对象的 id 属性，也可以是用 name 属性定义的别名。

```
<ref object="someObject"/>
```

使用 local 属性则可以充分利用 XML 解析器的功能来对同一文件中的 XML ID 进行引用验证。local 属性的值必须是目标对象的 id。如果在同一文件中找不到目标对象定义，XML 解析器就会报错。因此，如果目标对象定义在同一 XML 文件中，使用 local 属性进行引用是最好的选择，这样可以及早发现错误。

```
<ref local="someObject"/>
```

ref 节点的 parent 属性可以引用当前容器父容器中的对象，parent 属性的值可以是目标对象的 id，也可以是由 name 属性定义的别名，并且，目标对象必须位于当前容器的父容器中。比如，如果创建一个代理对象来包装父容器中的某个已知对象（代理对象可能和目标对象的名字相同），就需要从父容器中获得原始对象的引用。

```
<ref parent="someObject"/>
```

4.3.3.8.value 和 ref 节点的简化格式

我们可以使用 value 和 ref 节点的简化格式来缩短 XML 对象定义。property、constructor-arg 和 entry 节点都支持用 value 属性代替完整的 value 节点，如下：

```
<property name="MyProperty">
 <value>hello</value>
</property>

<constructor-arg>
 <value>hello</value>
</constructor-arg>

<entry key="myKey">
 <value>hello</value>
</entry>
```

这等价于：

```
<property name="MyProperty" value="hello"/>

<constructor-arg value="hello"/>

<entry key="myKey" value="hello"/>
```

一般来说，在手工键入对象定义时，这些简化格式可以减少键入工作。

类似的，property 和 constructor-arg 节点也支持用 ref 属性代替完整的 ref 节点，如下：

```
<property name="MyProperty">
 <ref object="anotherObject"/>
</property>
```

```
<constructor-arg index="0">
 <ref object="anotherObject"/>
</constructor-arg>
```

这等价于...

```
<property name="MyProperty" ref="anotherObject"/>

<constructor-arg index="0" ref="anotherObject"/>
```

注意

ref 简化格式只等价于<ref object="xxx">，<ref local="xxx">和<ref parent="xxx">并没有相应的简化格式，必须使用完整节点。

最后，entry 节点也允许用 key/key-ref 和 value/value-ref 简化词典键和值的定义。下面的定义：

```
<entry>
 <key><ref object="MyKeyObject"/></key>
 <ref object="MyValueObject"/>
</entry>
```

等同于：

```
<entry key-ref="MyKeyObject" value-ref="MyValueObject"/>
```

前面提到过，简化格式只等价于<ref object="xxx">，不能用来表示 local 或 parent 形式的对象引用。

4.3.3.9.复合属性名

在设置对象属性时，可以使用复合或嵌套的属性名，但要保证属性名路径中除最后一部分外的每一部分都不为空值，例如：

```
<object id="foo" type="Spring.Foo, Spring.Foo">
 <property name="Bar.Baz.Name" value="Bingo"/>
</object>
```

4.3.4.方法注入

多数用户都会将容器中的大部分对象布署为 singleton 模式。当一个 singleton 对象需要和另一个 singleton 对象协作，或者一个非 singleton 对象需要和另一个非 singleson 对象协作时，Spring.NET 都能很好的处理它们的依赖关系。但

是，如果对象的生存周期不同，就可能会产生问题。例如，假设一个 singleton 对象 A 要使用一个非 singleton（原型）对象 B，A 中的每个方法都会用到 B 的新实例。由于 A 是 singleton 对象，容器只有会创建它一次，也就是说只有一次给 A 的属性赋值的机会,所以不可能在每次 A 需要的时候都给它注入一个新的 B。

有一种解决的办法有点违背控制反转原则：类 A 可以通过实现 IObjectFactoryAware 接口来获取容器的引用，并调用 GetObject("B")在每次需要的时候从容器中请求一个（新的）对象 B。但这并不是一个很好的解决方案，因为客户代码此时必须要和 Spring.NET 发生紧耦合。

通过方法注入，我们可以用更优雅的方式解决类似的问题。

4.3.4.1.查询方法注入（Lookup Method Injection）

Spring.NET 可以动态覆盖对象的抽象方法或虚方法，并且可以在容器内查找已命名对象，查询方法注入就利用了这些功能。被查询对象一般应该是非 singleton 的（但也可以是 singleton）。Spring.NET 使用 System.Reflecttion.Emit 命名空间中的类型在运行时动态生成某个类的子类并覆盖其方法，以实现查询方法注入。

在需要进行方法注入的类中，被注入（即被覆盖）的方法必须按以下格式声明：（按：只需是一个抽象无参方法，可见性在 private 以上即可，方法名称没有限制。要注意的是，虽然包含抽象方法的类必须是抽象类，但是由于 Spring.NET 的动态子类化，使得抽象类看上去也可以被实例化，稍后请看一个例子。）

```
protected abstract SingleShotHelper CreateSingleShotHelper();
```

如果方法不是抽象的（按：但必须是虚方法），Spring.NET 也可以将其覆盖。在对象定义中，可以使用 object 节点的 lookup-method 子节点让 Spring.NET 在运行时覆盖抽象的 CreateSingleShotHelper 方法，使其返回容器中名为 singleShotHelper 的对象，如下：

```
<!-- a stateful object deployed as a prototype (non-singleton) -->
<object id="singleShotHelper" class="..." singleton="false"/>

<!-- myobject uses singleShotHelper -->
<object id="myObject" type="...">
  <lookup-method name="CreateSingleShotHelper"
object="singleShotHelper"/>
  <property>
    ...
  </property>

</object>
```

注入后，myObject 对象可随时调用自己的 CreateSingleShotHelpher 方法来获取一个新的 SingleShotHelper 实例。注意在定义 singleShotHelper 对象时必须要非常小心，如果确实每次都需要新的实例，一定要将其声明为非 singleton 模式。否则（不管使用默认设置还是显式指定 singleton 属性为 true），每次返回的仍将是同一个 SingleShotHelper 对象。

注意，查询方法注入可以和构造器注入（通过向构造器提供可选的参数值）及属性注入（通过设置属性值）一起使用。

（按:请看一个例子,首先,声明需要进行方法注入的类型和被查询对象的类型：

```
public abstract class MyClass //注意，可以直接在配置中定义这个类的对象
{
        public abstract SingleShotHelper CreateSingleShotHelper();
        //或者可以是一个虚方法
        //public virtual SingleShotHelper CreateSingleShotHelper()
        //{
        // return null;
        //}

        public void Process1()
        {
        //调用 this.CreateSingleShotHelper()
        }

        public void Process2()
        {
        //调用 this.CreateSingleShotHelper()
        }
}

public class SingleShotHelper
{
}
```

下面是相关的对象定义：

```
<object id="singleShotHelper" type="Sample1.SingleShotHelper, Sample1"
singleton="false"/>

<object id="myObject" type="Sample1.MyClass, Sample1">
        <lookup-method name="CreateSingleShotHelper"
object="singleShotHelper"/>
</object>
```

)

4.3.4.2.替换任意方法

这是一种较为少见的方法注入形式,可以用指定的方法替换容器内对象的任意方法。该功能略显复杂，用户可以跳过本节，等有实际需要时再来参考。

在 Xml 对象定义中，可以用 object 节点的 replaced-method 子节点将对象的任一方法替换为其它实现。下面的类定义了一个虚方法 ComputeValue()：

```
public class MyValueCalculator {

  public virtual string ComputeValue(string input) {
    // ... some real code
  }

  // ... some other methods
}
```

提供（要注入的）新方法的类必须实现
Spring.Objects.Factory.Support.IMethodReplacer 接口：

```
/// <summary>
/// Meant to be used to override the existing ComputeValue(string)
/// implementation in MyValueCalculator.
/// </summary>
public class ReplacementComputeValue : IMethodReplacer
{
        public object Implement(object target, MethodInfo method,
object[] arguments)
        {
                // get the input value, work with it, and return a computed
result...
                string value = (string) arguments[0];
                // compute...
                return result;
        }
}
```

对象定义如下：

```
<object id="myValueCalculator" type="Examples.MyValueCalculator,
ExampleAssembly">
  <!-- arbitrary method replacement -->
```

```
  <replaced-method name="ComputeValue"
replacer="replacementComputeValue">
    <arg-type match="String"/>
  </replaced-method>
</object>

<object id="replacementComputeValue"
type="Examples.ReplaceMentComputeValue, ExampleAssembly"/>
```

replaced-method 节点可以用一或多个 arg-type 子节点来标识被替换方法的签名，但注意只有当目标方法有重载时 arg-type 才是必需的。为方便起见，arg-type 的类型值可用类型全名的部分字符来表示，比如说，下面的字符都能表示 System.String 类型：

```
System.String
String
Str
```

因为通常仅靠参数的个数就足以区分不同的方法重载，所以用最简短的类型字符串来匹配方法参数可以节省不少键入工作。

4.3.5.引用其他对象或类型的成员

本节详细讨论如何使用其它对象或类型的成员来进行依赖注入。这种情况很普遍，特别是在使用某些不能(或不希望)按 Spring.NET 的约定来修改的现有类型时。比如说，某个类的构造器参数值需要从数据库的记录中计算出来，并且必须由另外一个类型处理后才能得出确切的值。MethodInvokingFactoryObject 类可以处理这些问题，它允许把任意方法的返回值注入给对象的构造器参数或属性。类似的，PropertyRetrievingFactoryObject 类和 FieldRetrievingFactoryObject 类分别允许用其它类型的属性或字段进行注入。这些类都实现了 IFactoryObject 接口，以表明自己的对象是一个工厂对象，通过工厂对象的 id 获取的不是工厂对象本身，而是其产品对象。在 4.5.3,IFactoryObject 接口中将讨论工厂对象。

4.3.5.1.使用对象或类的属性值进行注入

PropertyRetrievingFactoryObject 是 IFactoryObject 接口的实现类，用于将其它对象或类型的属性值注入给对象的属性或构造器参数。该类可以从对象或类型中（指静态属性）获取任何**公有**属性的值。

如果要用 PropertyRetrievingFactoryObject 获取实例属性的值，需要为其指定目标对象及属性名，目标对象可以是任意其它对象，甚至可以是内联对象。如果要获取静态属性，则需要指定目标类型的全名和属性名。

然后，通过 PropertyRetrievingFactoryObject 对象的 id 即可获取目标对象或类型的属性值，获取的结果可直接注入给对象的属性或构造器参数。注意，实例属性和静态属性都支持嵌套。

下面使用 PropertyRetrievingFactoryObject 获取目标对象的属性值。在此目标对象的属性值是个内联的对象定义，所以在 PropertyRetrievingFactoryObject 对象定义的 TargetProperty 属性中，使用了嵌套式的属性路径。

```
<object name="person" type="Spring.Objects.TestObject,
Spring.Core.Tests">
  <property name="age" value="20"/>
  <property name="spouse">
    <object type="Spring.Objects.TestObject, Spring.Core.Tests">
      <property name="age" value="21"/>
    </object>

  </property>
</object>

// will result in 21, which is the value of property 'spouse.age' of object
'person'
<object name="theAge"
type="Spring.Objects.Factory.Config.PropertyRetrievingFactoryObject,
Spring.Core">
  <property name="TargetObject" ref="person"/>
  <property name="TargetProperty" value="spouse.age"/>
</object>
```

下面是使用 PropertyRetrievingFactoryObject 获取静态属性的方法：

```
<object id="cultureAware"

type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject,
Spring.Core.Tests">
  <property name="culture" ref="cultureFactory"/>
</object>

<object id="cultureFactory"

type="Spring.Objects.Factory.Config.PropertyRetrievingFactoryObject,
Spring.Core">
  <property name="StaticProperty">
      <value>System.Globalization.CultureInfo.CurrentUICulture,
Mscorlib</value>
```

```
  </property>
</object>
```

类似的，获取实例属性的方法如下：

```
<object id="instancePropertyCultureAware"

type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject,
Spring.Core.Tests">
  <property name="Culture" ref="instancePropertyCultureFactory"/>
</object>

<object id="instancePropertyCultureFactory"

type="Spring.Objects.Factory.Config.PropertyRetrievingFactoryObject,
Spring.Core">
  <property name="TargetObject"
ref="instancePropertyCultureAwareSource"/>
  <property name="TargetProperty" value="MyDefaultCulture"/>

</object>

<object id="instancePropertyCultureAwareSource"

type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject,
Spring.Core.Tests"/>
```

4.3.5.2.使用字段值进行注入

FieldRetrievingFactoryObject 类的功能和 PropertyRetrievingFactoryObject 很相似。如其名称所示，FieldRetrievingFactoryObject 可以获取对象或类（指静态字段）的**公有**字段值。

下面的例子使用 FieldRetrievingFactoryObject 获取一个类的公有静态字段：

```
<object id="withTypesField"

type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject,
Spring.Core.Tests">
  <property name="Types" ref="emptyTypesFactory"/>
</object>
```

```
<object id="emptyTypesFactory"

type="Spring.Objects.Factory.Config.FieldRetrievingFactoryObject,
Spring.Core">
  <property name="TargetType" value="System.Type, Mscorlib"/>
    <property name="TargetField" value="EmPTytypeS"/>

</object>
```

获取公有实例字段的方法如下：

```
<object id="instanceCultureAware"

type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject,
Spring.Core.Tests">
  <property name="Culture" ref="instanceCultureFactory"/>
</object>

<object id="instanceCultureFactory"
  type="Spring.Objects.Factory.Config.FieldRetrievingFactoryObject,
Spring.Core">
  <property name="TargetObject" ref="instanceCultureAwareSource"/>
  <property name="TargetField" value="Default"/>

</object>

<object id="instanceCultureAwareSource"

type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject,
Spring.Core.Tests"/>
```

4.3.5.3.使用方法的返回值进行注入

本节要讲的第三个类是 MethodInvokingFactoryObject。
MethodInvokingFactoryObject 允许使用任意方法的返回值进行注入。

MethodInvokingFactoryObject 类可以处理实例方法和静态方法。此外，Spring.NET 中另有一种处理对象初始化的机制（参考 4.5.1.1, IInitializingObject 接口和 init-method 属性）也可以用来进行（初始化）方法的调用，但是用这种机制调用方法的目的只是进行初始化工作，并且不允许向方法中传递任何参数，调用的时机也被限制在对象被容器创建时。

MethodInvokingFactoryObject 类则允许在任意时刻调用任意对象的任意方法（或者任意类的静态方法）。

在下面的例子中，使用 MethodInvokingFactoryObject 类强制在 myService 对象创建之前调用一个静态方法。

```
<object id="force-init"
 type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject,
Spring.Core">
 <property name="TargetMethod" value="Initialize">
 <property name="TargetType"
value="ExampleNamespace.ExampleInitializerClass, ExampleAssembly">
</object>

<object id="myService" type="MyService" depends-on="force-init"/>
```

（按：原文中上面的例子似有错误，此处已做修正；具体可参考英文文档）

注意对象定义 myService 的 depends-on 属性引用了一个名为 force-init 的 MethodInvokingFactoryObject 对象，该引用会使对象 force-init 在 myService 之前初始化（并且调用 force-init 的目标类型上的目标方法。注意，若要使这个配置正常工作，myService 对象必须以 singleton 模式操作(按:否则 myService 对象只会在请求时创建，而此时 force-init 已经被容器预先创建了，也就失去了 depends-on 的意义）——这也是默认的对象布署模式，参见下一章）。

MethodInvokingFactoryObject 类也可用于访问工厂方法（按：当在代码中通过名称"force-init"从容器中获取对象时，如果其目标方法有返回值，那么获得的对象就是目标方法的返回值；如果目标方法返回 null，那么获得对象的类型则是 System.Reflecton.Missing），一般情况下，工厂方法都是以 singleton 模式操作的。如果 MethodInvokingFactoryObject 对象为 singleton 模式，那么在它所有属性被设置后（按：应该是设置了足够的属性后，即 TargetType 和 TargetMethod），目标方法会立即被调用并将返回值存入缓存以备用。随后，如果容器向此工厂请求对象，那么返回的将是缓存中的值。
MethodInvokingFactoryObject 对象的 singleton 属性可以设置为 false，此时每次请求对象都会去调用目标方法（返回值不会被缓存）。

在 MethodInvokingFactoryObject 的对象定义中，用 TargetMethod 和 TargetType 属性指定目标静态方法名和方法所在的类型全名；而用 TargetMethod 和 TargetObject 属性来指定目标实例方法名和方法所在对象的引用。

目标方法的参数可以通过两种方式来设置（这两种方式可混合使用）。第一种是通过 Arguments 属性来指定参数列表。注意列表中项的顺序是很重要的——必须和方法签名中的参数列表在次序上完全一致，且类型兼容，如下例：

```
<object id="myObject"
type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject,
Spring.Core">
  <property name="TargetType" value="Whatever.MyClassFactory,
MyAssembly"/>
  <property name="TargetMethod" value="GetInstance"/>

  <!-- the ordering of arguments is significant -->
  <property name="Arguments">
    <list>

      <value>1st</value>
      <value>2nd</value>
      <value>and 3rd arguments</value>
      <!-- automatic Type-conversion will be performed prior to invoking
the method -->

    </list>
  </property>
</object>
```

第二种方式是通过 NamedArguments 属性配置一系列键/值对，其中键名对应参数名（文本类型），值对应参数值（可以是任意对象）。参数名是大小写不敏感的，并且顺序（当然）是不重要的（因为词典类型本身就没有顺序）。见下面的例子：

```
<object id="myObject"
type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject,
Spring.Core">
  <property name="TargetObject">
    <object type="Whatever.MyClassFactory, MyAssembly"/>
  </property>
  <property name="TargetMethod" value="Execute"/>

  <!-- the ordering of named arguments is not significant -->

  <property name="NamedArguments">
    <dictionary>
      <entry key="argumentName"><value>1st</value></entry>
      <entry key="finalArgumentName"><value>and 3rd
arguments</value></entry>

      <entry key="anotherArgumentName"><value>2nd</value></entry>
    </dictionary>
  </property>
```

```
</object>
```

下面使用 MethodInvokingFactoryObject 来调用一个实例方法：

```
<object id="myMethodObject" type="Whatever.MyClassFactory, MyAssembly"
/>

<object id="myObject"
type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject,
Spring.Core">
  <property name="TargetObject" ref="myMethodObject"/>
  <property name="TargetMethod" value="Execute"/>
</object>
```

上例中的目标对象也可以是匿名的内联对象定义...如果对象的方法不会在工厂对象之外调用，那么最好通过内联对象定义将目标方法限制在工厂对象内部。

最后要注意，如果使用 MethodInvokingFactoryObject 来调用一个有变长参数列表的方法，必须使用 list 来按顺序定义要传递的参数值。请看下面的例子，其中 CreateObject 方法的参数 arguments 是用 C#关键字 params 来定义的；随后是相应的 XML 配置：

```
[C#]
public class MyClassFactory
{
    public object CreateObject(Type objectType, params string[]
arguments)
    {
        return ... // implementation elided for clarity...
    }
}
<object id="myMethodObject" type="Whatever.MyClassFactory, MyAssembly"
/>

<object id="paramsMethodObject"
type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject,
Spring.Core">
  <property name="TargetObject" ref="myMethodObject"/>
  <property name="TargetMethod" value="CreateObject"/>
  <property name="Arguments">
    <list>

      <value>System.String</value>
      <!-- here is the 'params string[] arguments' -->
      <list>
```

```
        <value>1st</value>
        <value>2nd</value>

      </list>
    </list>
</object>
```

4.3.6.IFactoryObject 接口的其它实现类

除了前一节讲过的 PropertyRetrievingFactoryObject，MethodInvokingFactoryObject 和 FieldRetrievingFactoryObject 三个类之外，Spring.NET 还提供了另外几个非常有用的 IFactoryObject 接口实现类，下面我们来逐一讨论它们。

4.3.6.1. Log4Net

Log4NetFactoryObject 类允许在多个对象中共享一个日志对象，而不需为每个对象都创建单独的日志对象。下面这个 Log4NetFactoryObject 对象的 LogName 属性值为“DAOLogger”，该对象由 SimpleAccoundDao 和 SimpleProductDao 对象共用。

```
<objects xmlns="http://www.springframework.net"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.springframework.net
         http://www.springframework.net/xsd/spring-objects.xsd" >

    <object name="daoLogger"
type="Spring.Objects.Factory.Config.Log4NetFactoryObject,
Spring.Core">
      <property name="logName" value="DAOLogger"/>
    </object>

    <object name="productDao" type="PropPlayApp.SimpleProductDao,
PropPlayApp ">
      <property name="maxResults" value="100"/>

      <property name="dbConnection" ref="myConnection"/>
      <property name="log" ref="daoLogger"/>
    </object>

    <object name="accountDao" type="PropPlayApp.SimpleAccountDao,
PropPlayApp ">
      <property name="maxResults" value="100"/>
```

```
      <property name="dbConnection" ref="myConnection"/>

      <property name="log" ref="daoLogger"/>
    </object>

    <object name="myConnection" type="System.Data.Odbc.OdbcConnection,
System.Data">
      <property name="connectionstring"
value="dsn=MyDSN;uid=sa;pwd=myPassword;"/>
    </object>

</objects>
```

4.3.7.使用 depends-on

Spring.NET 使用<ref/>节点来引用对象的依赖项。除非有特殊的初始化需求，一般不需要使用 depends-on 属性。但是，如果需要使用其它静态（类型或方法）或对象来做一些初始化的工作，就可以借助 depends-on 属性来确保在使用依赖对象之前将其初始化（按：与 ref 不同，depends-on 只影响创建顺序，一个对象可以和它 depends-on 的对象没有任何依赖关系；另外，如果 a 和 b 都未创建，那么 a ref b 时，创建次序是先 a 后 b，也就是先创建 a，在为 a 进行依赖注入时才考虑 b；而 a depends-on b 的创建次序是先 b 后 a，不论 a 是否要引用 b，都先创建 b）。例如：

```
<object id="objectOne" type="Examples.ExampleObject, ExamplesLibrary"
depends-on="manager">
  <property name="manager" ref="manager"/>
</object>

<object id="manager" type="ManagerObject"/>
```

4.3.8.自动装配协作对象

Spring.NET 具有自动装配的能力，也就是说，Spring.NET 可以通过对象定义自动分辨某个对象的协作对象。自动装配是针对单个对象（按：针对每个协作对象）进行的，所以可对某些对象启用而某些对象关闭（按：即自动装配某些协作对象，而不自动装配其它协作对象）。使用自动装配可以减少甚至完全消除属性或参数值的设置工作。[2]自动装配有五种模式：

表 4.3.　自动装配模式

模式	解释
no	不进行自动装配。这是默认的设置，一般不建议修改该设置，因为显式指定对象的协作关系可以让开发人员很清楚自己在干什么，并且有助于为系统结构建立文档。
byName	Spring.NET 会检查容器中的对象，查找和被装配属性完全同名的对象定义，并将其作为该属性的值。例如，如果将一个对象定义设置为按名称装配，且该对象有一个名为 Master 的属性，Spring.NET 就会查找名为 Master 的对象定义，并将其作为 Master 属性的值。
byType	Spring.NET 会根据指定的类型来查找协作对象。假设某对象定义有一个类型为 SqlConnection 的协作对象，Spring.NET 会在整个对象工厂中查找类型为 SqlConnection 的对象定义来为其装配。如果找不到该类型的对象，或者找到了不止一个此类型的对象，Spring.NET 会抛出异常，也就无法对此对象进行自动装配了。
constructor	和 byType 很相似，只是查找的对象要赋值给构造器的参数。如果对象工厂中没有或不只一个与参数类型相同的对象定义，会抛出致命异常。
autodetect	根据对象自身的配置信息来自动确定是使用 constructor 还是 byType 模式。如果发现对象具有默认构造器，则使用 byType 模式。

注意

显式指定的依赖项会覆盖自动装配的设置。自动装配可以和依赖检查结合使用，依赖检查会在所有自动装配完成之后进行。

4.3.9.检查依赖项

Spring.NET 可为容器中的对象检查依赖关系。依赖关系是由未定义实际值（按：就是没有用 value 子节点或属性指定值的情况，比如使用 ref 引用其它对象）的属性或自动装配来指定的。

依赖检查可确保对象的所有属性（或所有某种类型的属性）都能被正确注入。当然，多数类都会为自己的属性设置默认值，或者某些属性并非在所有情况下都是必需的，所以依赖检查的用处实际上有限。依赖检查同样可以象自动装配那样，针对单个对象启用或关闭。默认的设置是不进行依赖检查。依赖检查也有几种不同的处理模式，在 XML 对象定义中，可通过 object 节点的 dependency-check 属性来设置，如下表所示：

表 4.4. 依赖检查模式

模式	解释
none	不进行依赖检查。没有定义值的属性不会被设置。
simple	只对基元类型和集合属性进行检查（也就是除了协作对象引用外的其它所有情况）。
object	只对协作对象进行依赖检查。

模式	解释
all	对包括协作对象和基元类型、集合属性在内的所有依赖项进行检查。

4.4. 类型转换

类型转换器（TypeConverter）用于将对象从一种类型转换为另一种类型。比如在用 XML 文件配置 IoC 容器时，就需要将文本值转换成目标属性的实际类型。如果没有为某种类型注册类型转换器，Spring.NET 就会使用标准的.NET 类型转换方法进行转型。本节将简述如何注册自定义的类型转换器。在标准的.NET 代码中,可用 TypeConverter 特性将类型转换器与目标类型相关联[3],例如 FCL 中 Font 类型的定义：

```
[Serializable, TypeConverter(typeof(FontConverter)), ...]
public sealed class Font : MarshalByRefObject, ICloneable, ISerializable,
IDisposable
{
  // Methods

  ... etc ..
}
```

4.4.1.枚举类型的转换

枚举类型的默认转换器是 System.ComponentModel.EnumConverter 类。在对象定义中，要使用枚举的某个值只需使用对应的枚举名称即可。例如，假设 TestObject 类中有一个 FileMode 枚举类型的属性，该枚举的一个值为 Create。在下面的 XML 定义中可以看到枚举类型属性值的设置方法：

```
<object id="rod" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="name" value="Rod"/>
  <property name="FileMode" value="Create"/>
</object>
```

4.4.2.内置的类型转换器

Spring.NET 预注册了一部分自定义类型转换器（例如将类的文本名称转换为 System.Type 类型的对象）。这些转换器都定义在 Spring.Core 程序集的 Spring.Objects.TypeConverters 命名空间下，如下表所示：

表 4.5. 内置的类型转换器

类型	解释
RuntimeTypeConverter	将字符串表示的类型名转换为实际的 System.Type 对象
FileInfoConverter	将字符串转换为 System.IO.FileInfo 对象
StringArrayConverter	将以逗号分隔的字符串列表转换为字符串数组，或相反
UriConverter	将字符串表示的 URI 转换为实际的 URI 对象
StreamConverter	将 Spring 的 IResource URI 字符串表示形式转换为 InputStream 对象.
ResourceConverter	将 Spring 的 IResource URI 字符串表示形式转换为 IResource 对象
ResourceManagerConverter	将一个两段字符串（资源名，程序集名）转换为 System.Resource.ResourceManager 对象
RGBColorConverter	将以逗号分隔的红，绿，蓝值转换为 System.Drawing.Color 结构
ExpressionConverter	将字符串转换为 IExpression 接口实现类的对象

Spring.NET 使用标准的.NET 机制来解析各种类型，包括但不限于检查和应用程序相关的所有配置文件、GAC 和程序集探测机制。

4.4.3.自定义类型转换器

自定义的类型转换器可通过几种方法进行注册。在 Spring.NET 中，自定义类型转换器由 TypeConverterRegistry 类管理，在使用基于 XML 的容器时，最便捷的方法是在配置文件中添加自定义的节点处理器 TypeConverterSectionHandler，请参见 4.12 节, 配置 IApplicationContext。

另外一种方法与目前 Java 版的 Spring 相兼容，即使用对象工厂后处理器 Spring.Objects.Factory.Config.CustomConverterConfigurer。下一节我们会讨论这个类。

如果通过编程方式创建 IoC 容器，则需要使用 IConfigurableObjectFactory 接口的 RegisterCustomConverter(Type requiredType, TypeConverter converter) 方法注册类型转换器。

4.4.3.1.使用 CustomConverterConfigurer 类

本节详细讨论如何在不使用 TypeConverter 特性的情况下为类型定义转换器。类型转换器也是独立的类，继承自 TypeConverter。本节的方法利用了 Spring.NET 的工厂后处理机制。

请看下面的 ExoticType 类，以及包含一个 ExoticType 类型属性的 DependsOnExoticType 类：

```
public class ExoticType
{
    private string name;

    public ExoticType(string name)
    {
        this.name = name;
    }

    public string Name
    {
        get { return this.name; }
    }
}
```

及：

```
public class DependsOnExoticType
{
    public DependsOnExoticType() {}

    private ExoticType exoticType;

    public ExoticType ExoticType
    {
        get { return this.exoticType; }
        set { this.exoticType = value; }
    }

    public override string ToString()
    {
        return exoticType.Name;
    }
}
```

如果设置得当，我们希望能在配置文件中通过字符串来为该类的 ExoticType 属性设定值，在后台，Spring.NET 会使用一个 TypeConverter 将字符串值转换为 ExoticType 对象。

```
<object name="sample" type="SimpleApp.DependsOnExoticType, SimpleApp">
  <property name="exoticType" value="aNameForExoticType"/>
</object>
```

相应的类型转换器为：

```
public class ExoticTypeConverter : TypeConverter
{
        public ExoticTypeConverter()
        {
        }

        public override bool CanConvertFrom (
                ITypeDescriptorContext context,
                Type sourceType)
        {
                if (sourceType == typeof (string))
                {
                        return true;
                }
                return base.CanConvertFrom (context, sourceType);
        }

        public override object ConvertFrom (
                ITypeDescriptorContext context,
                CultureInfo culture, object value)
        {
                if (value is string)
                {
                        string s = value as string;
                        return new ExoticType(s.ToUpper());
                }
                return base.ConvertFrom (context, culture, value);
        }
}
```

最后，在容器的配置文件中，用 CustomConverterConfigurer 对象注册新的 TypeConverter，Spring.NET 就可以使用它了：

```
<object id="customConverterConfigurer"
       type="Spring.Objects.Factory.Config.CustomConverterConfigurer,
Spring.Core">
  <property name="CustomConverters">
    <dictionary>
      <entry key="SimpleApp.ExoticType">
        <object type="SimpleApp.ExoticTypeConverter"/>
      </entry>

    </dictionary>
  </property>
```

```
</object>
```

4.5. 自定义对象的行为

4.5.1.生命周期接口

Spring.NET 通过几个专门的接口来控制容器中对象的行为，这些接口包括 Spring.NET 定义的 IInitializingObject 接口和标准的 System.IDisposable 接口。容器会在实现了这两个接口的对象上调用 AfterPropertyiesSet()方法和 Dispose()方法，这样我们就有机会在对象初始化和销毁时做一些额外的工作。

在内部，Spring.NET 使用 IObjectPostProcessor 接口的实现类来处理这两个接口并调用适当的方法。如果需要某些特殊的功能或是 Spring.NET 没有提供的生命周期行为，可以自行创建 IObjectPostProcessor 的实现类。请参考 4.8 节，使用 IObjectPostProcessor 接口自定义对象。

下面讨论所有的生命周期接口。

4.5.1.1.IInitializingObject 接口和 init-method 属性

Spring.Objects.Factory.IInitializingObject 接口允许容器在完成对象的属性设置之后执行对象的初始化工作。该接口只有一个方法：

- void AfterPropertiesSet()：在对象的所有属性被设置之后由容器调用。在该方法中，我们可以检查必需的属性是否都已被正确设值，或者可以执行进一步的初始化工作。此方法可以抛出任何异常以标识配置错误、初始化失败等情况。

注意

一般来说，尽量不要使用 IInitializingObject 接口。因为在对象定义中可以通过 init-method 属性来指定初始化方法。

```
<object id="exampleInitObject" type="Examples.ExampleObject"
init-method="init"/>
[C#]
public class ExampleObject
{
    public void Init()
    {
        // do some initialization work
    }
}
```

和下面的方法是一样的...

```
<object id="exampleInitObject" type="Examples.AnotherExampleObject"/>
[C#]
public class AnotherExampleObject : IInitializingObject
{
    public void AfterPropertiesSet()
    {
        // do some initialization work
    }
}
```

但在使用 init-method 时，无需对 Spring.NET 产生依赖。

注意

在布署 prototype 模式的对象时，容器无法处理对象的生命周期事件。Spring.NET 不可能仅通过对象定义来完全控制非 singleton/原型对象的生存期。对象一旦创建就会交付给客户代码，容器不再保存其引用。对于 Spring.NET 来说，非 singleton 对象就如同使用 new 操作符一样，所有与对象生命周期有关的工作都应该由客户代码来处理。

4.5.1.2.IDisposable 接口和 destroy-method 属性

System.IDisposable 接口使我们有机会在容器被销毁时，通过对象的回调方法来处理容器中对象的销毁工作。该接口只有一个方法：

- void Dispose()：当容器被销毁时调用。在此可释放对象保留的任何资源（比如数据库连接）。可在此方法中抛出任何异常，但是此处抛出的任何异常都不会阻止容器的销毁，只会被记录到日志中。

注意

由于在对象定义中（XML 或数据库中）可以通过 destroy-method 来指定对象的清理方法，所以在使用 Spring.NET 时，可以不去实现 IDisposable 接口。

```
<object id="exampleInitObject" type="Examples.ExampleObject"
destroy-method="cleanup"/>
[C#]
public class ExampleObject
{
    public void cleanup()
    {
        // do some destruction work (such as closing any open connection
(s))
```

```
    }
}
```

和下面的方式完全相同：

```
<object id="exampleInitObject" type="Examples.AnotherExampleObject"/>
[C#]
public class AnotherExampleObject : IDisposable
{
    public void Dispose()
    {
        // do some destruction work
    }
}
```

4.5.2.让对象了解自己的容器

4.5.2.1.IObjectFactoryAware 接口

实现了 Spring.Objects.Factory.IObjectFactoryAware 接口的对象可以在被容器创建后获取它所在容器的引用。该接口只有一个（只写的）属性：

- IObjectFactory ObjectFactory：该属性会在初始化方法（IInitializingObject 的 AfterPropertiesSet 方法，或在对象定义中由 init-method 属性指定的方法）完成后被赋值。

IObjectFactoryAware 接口允许对象通过编程方式访问创建它的容器，一般主要用来在编程时获得容器内的其它对象。虽然这种方式在某些情况下（似乎）很有用，但我们应该尽量避免这么做，因为这会导致代码同 Spring.NET 发生紧耦合，并且违反了控制反转原则。

4.5.2.2.IObjectNameAware 接口

Spring.Objects.Factory.IObjectNameAware 接口可让对象知道自己在容器中的名称，如果对象需要知道自己在容器中的名称，可以实现该接口：

- string ObjectName：该属性使实现了 IObjectNameAware 接口的对象能够知道自己在容器中的名称。

4.5.3.IFactoryObject 接口

如果某个类型本身要作为工厂使用，可以实现 Spring.Objects.Factory.IFactoryObject 接口。该接口有一个方法和两个（只读）属性：

- object GetObject()：必须返回该工厂所创建的对象。返回的对象可能会被共享（取决于工厂创建对象的方式是 singleton 还是 prototype，见 IsSingleton 属性）
- bool IsSingleton：如果 IFactoryObject 返回的对象是 singleton，该属性必须为 true；否则为 false。
- Type ObjectType：要么返回 GetObject()方法创建的对象的类型，要么返回 null（当无法预知产品对象的具体类型时）。

在 4.3.5,引用其它对象或类型的成员中讲过，ProprteryRetrievingFactoryObject 和 FieldRetrievingFactoryObject 类就实现了 IFactoryObject 接口。

4.6. 抽象对象定义和子对象定义

对象定义可能会包含大量的信息，比如与容器相关的信息（即初始化方法、静态工厂方法名等）、构造器参数和属性值等。子对象定义是指从一个父对象定义中继承了配置数据的对象定义。子对象定义可以根据需要重写或添加某些配置的值。使用父对象和子对象的定义方式可能会节省大量的键入工作。实际上这是模版的一种形式。

在 Spring.NET，子对象定义由 ChildObjectDefintion 类处理。但用户一般不会直接接触到该类，仍可使用 XML 对象定义进行配置。在 XML 对象定义中，通过 parent 属性标识一个对象定义是子对象定义，该属性的值即为父对象定义的名称：

```
<object id="inheritedTestObject" type="Spring.Objects.TestObject,
Spring.Core.Tests">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</object>
<object id="inheritsWithDifferentClass"
type="Spring.Objects.DerivedTestObject, Spring.Core.Tests"
  parent="inheritedTestObject" init-method="Initialize">
  <property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->

</object>
```

没有在子对象定义中显式指定的属性会从父对象定义中继承值;同时子对象定义也可覆盖父对象定义的属性值。对前一种情况来说，子对象必须与父对象兼容，也就是说，子对象必须能够接受父对象的属性值。

子对象定义可继承或覆盖父对象定义的构造器参数、属性值及方法，也可以添加新值。如果子对象指定了初始化方法、销毁方法和/或静态工厂方法，就会覆盖父对象定义中的相应配置。

下面的设置对子对象定义来说始终有效：depends-on 属性的值、自动装配模式、依赖检查模式、singleton 和 lazy-init 属性的值。（按：即子对象定义不会从父对象定义继承这些属性，举例来说，若父对象（抽象）定义了 depends-on，而子对象未定义，那么子对象就没有依赖项）

如果父对象定义未指定类型...

```
<object id="inheritedTestObjectWithoutClass">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</object>
<object id="inheritsWithClass" type="Spring.Objects.DerivedTestObject,
Spring.Core.Tests"
  parent="inheritedTestObjectWithoutClass" init-method="Initialize">
  <property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->

</object>
```

那么父对象就无法被初始化，因为它的定义是不完整的。这种定义也被隐式的看作为抽象定义。对象定义也可通过 abstract 属性显式指定为抽象定义，该属性的有效值是 true 或者 false。一个抽象的定义仅仅是为用做模版，或者说是作为子对象定义的父对象定义而存在。将对象定义声明为抽象可防止容器创建该对象，并且任何试图获取该对象的操作都会抛出 ObjectIsAbstractException 异常（按：原文对此异常名称的描述有误，已更正）。容器内负责预创建对象的 PreInstantiateSingletons 方法会忽略抽象的对象定义。

注意

应用程序上下文（即 IApplicationContext 的实现类，而非简单的对象工厂）在默认情况下会预先创建所有的 singleton 对象。所以，对应用程序上下文来说，下面一点非常重要（至少对于 singleton 对象来说）：如果要定义一个仅用做模版的（父）对象定义，而且在该定义中指定了类型，那么就必须保证将其 abstract 属性设为 true，否则应用程序上下文会尝试预创建这个对象。

```
<object id="abstractObject" abstract="true"
  type="Spring.Objects.DerivedTestObject, Spring.Core.Tests">
```

```
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</object>
<object id="inheritsFromAbstract"
type="Spring.Objects.DerivedTestObject, Spring.Core.Tests"
  parent="inheritedTestObjectWithoutClass" init-method="Initialize">
  <property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->

</object>
```

4.7. 与 IObjectFactory 接口交互

IObjectFactory 本质上是一个注册了各种对象及其依赖项的高级工厂。通过 IObjectFactory 接口，可以读取容器内的对象定义并访问它们所代表的对象。如果只需要使用 IOjbectFactory 接口的功能，可以用代码创建其实现类的实例，然后读取 XML 对象定义中的内容，如下：

```
[C#]
IResource input = new FileSystemResource ("objects.xml");
XmlObjectFactory factory = new XmlObjectFactory(input);
```

对象工厂创建之后，就可以使用 GetObject(string)方法（或者使用索引器 factory[string]，更为简洁）获取需要的对象：

```
[C#]
object foo = factory.GetObject ("foo"); // gets the object defined as
'foo'
object bar = factory ["bar"];           // same thing, just using the
indexer
```

上面的代码执行后，如果对象以 singleton 方式（默认的方式）定义，就会获得该对象一个唯一的引用；如果对象定义中 singleton 属性为 false，那么就每次调用这段代码都会返回一个新的对象：

```
<object id="exampleObject" type="Examples.ExampleObject,
ExamplesLibrary"/>
<object id="anotherObject" type="Examples.ExampleObject,
ExamplesLibrary" singleton="false"/>
[C#]
object one = factory ["exampleObject"];   // gets the object defined as
'exampleObject'
object two = factory ["exampleObject"];
Console.WriteLine (one == two)             // prints 'true'
```

```
object three = factory ["anotherObject"]; // gets the object defined as
'anotherObject'
object four = factory ["anotherObject"];
Console.WriteLine (three == four);        // prints 'false'
```

对客户代码而言，IObjectFactory 的使用非常简单，只需要用到其中的 7 个方法（或索引器），如下：

- bool ContainsObject(string)：如果 IObjectFactory 包含以某个名称命名的对象，则返回 true。
- object GetObject(string)：返回以指定名称命名的对象。根据对象的配置方式，可能返回一个 singleton 对象（共享的），也可能返回一个新创建的对象。如果请求的对象没有找到，就会抛出 NoSuchObjectDefinitionException 异常，如果在初始化和准备对象时出错，则抛出 ObjectsException 异常。
- Object this[string]：IObjectFactory 接口定义的索引器。工作方式同 GetObject(string)方法一样。本文一般都使用 GetObject(string)方法，不过要注意只要能使用 GetObject(string)方法的地方就可以使用索引器。
- Object GetObject(string,Type)：返回以给定名称注册的对象，并转换为指定的类型。若无法进行转型，会抛出 ObjectNotOfRequiredTypeException 异常。此外，GetObject(string)遵守的规则也适用于该方法。
- bool IsSingleton(string)：判断以给定名称命名的对象是 singleton 还是 prototype。如果找不到所请求的对象，会抛出 NoSuchObjectDefinitionException 异常。
- string[] GetAliases(string)：返回以给定名称命名的对象的别名（如果该对象定义了别名的话）。
- void ConfigureObject(object target)：以容器中某个抽象对象定义为模板，为目标对象进行依赖注入。要注意的是，在用作模板的抽象对象定义中，类型不能省略，且必须和目标对象的类型全名完全一致（按：是否包含程序集名均可；因为指定了类型，所以一定要显式使用 abstract 属性；此外，该抽象对象定义不能以任何方式（id 或 name）配置标识符；原文对此描述不准确，读者可以自行实验）。如果对象的创建和初始化不受开发人员的控制，比如 ASP.NET 创建的 Web 控件、或者 WinForm 程序创建的用户控件，就可以用这种方式对其进行依赖注入。
- void ConfigureObject(object target, string name)：功能与上面的方法一样，不同是，这次是按照某个指定名称的对象定义为模板，为目标对象进行依赖注入。（按：注意，此时以 name 为名的对象定义不必是抽象的，并且在类型上可以和目标对象完全不一致，但其依赖项必须兼容）。

4.7.1.获得 IFactoryObject 对象本身，而非其产品

有时需要通过 IObjectFactory 接口请求一个 IFactoryObject 对象本身，而非其产品对象的引用。为此，可将字符'&'加在对象名称的前面作为 IObjectFactory.GetObject(string)方法的参数。举例来说，假如有一个 IFactoryObject 类型的对象，其 id 为 myObject，那么调用 GetObject("myObject")获得的是 IFactoryObject 创建的对象，而 GetObject("&myObject")获得的则是 IFactoryObject 对象本身。

4.8.使用 IObjectPostProcessor 接口自定义对象

对象后处理器（object post-processor）是指实现了 Spring.Objects.Factory.Config.IObjectPostProcessor 接口的类，该接口包括两个方法：

```
object PostProcessBeforeInitialization(object instance, string name);

object PostProcessAfterInitialization(object instance, string name);
```

在把实现了该接口的对象作为对象后处理器注册给容器后，容器就会在每个对象的初始化方法被调用之前调用对象后处理器的 PostProcessBeforeInitialization 方法，初始化方法可以是 IInitializingObject 接口的 AfterPropertiesSet 方法，也可以是通过对象定义（按：init-method 属性）指定的任意方法；在对象的初始化方法返回之后，容器就会调用对象后处理器的 PostProcessAfterInitialization 方法。（按：对象后处理器只对指定了初始化方法的对象有效）

对象后处理器可以对一个对象做任何操作，当然也可以不做任何事情而直接退出。后处理器通常用于检查标识接口，或将对象包装到某个代理对象中。Spring.NET 中的一些辅助类实现了 IObjectPostProcessor 接口。

IObjectPostProcessor 另有两个扩展接口：IInstantiationAwareObjectPostProcessor 和 IDestructionAwareObjectPostProcessor。

```
public interface IInstantiationAwareObjectPostProcessor :
IObjectPostProcessor
{
    object PostProcessBeforeInstantiation(Type objectType, string
objectName);
}
```

```
public interface IDestructionAwareObjectPostProcessor :
IObjectPostProcessor
{
     void PostProcessBeforeDestruction (object instance, string name);
}
```

PostProcessBeforeInstantiation 方法由容器在创建对象前调用（按：注意是创建前）。如果该方法的返回值不为空，那么容器的默认初始化行为就会被短路。返回的对象会代替原来的对象被注册到容器，且不会再向它应用任何 IObjectPostProcessor。当希望获取一个对象的代理而非对象本身时，该机制相当有用。另外，PostProcessBeforeDestruction 方法会在一个 singleton 对象被销毁前调用。（按：IObjectPostProcessor 接口的两个方法只对指定了初始化方法的对象有效，而 IInstantiationAwareObjectPostProcessor 接口和 IDestructionAwareObjectPostProcessor 接口的方法会在每个对象创建或销毁前调用。）

注意：IObjectFactory 的实现类和 IApplicationContext 的实现类在对象后处理器的处理方式上稍微不同。IApplicationContext 会自动检测布署到其中的对象，如果对象实现了 IObjectPostProcessor 接口，就将其注册为对象后处理器，并在适当的时候调用。对象后处理器的对象定义与普通对象完全一样。但对 IObjectFactory 的实现类来说，对象后处理器就必须显式的手工添加，如以下代码所示：

```
ConfigurableObjectFactory factory = new .....; // create an
IObjectFactory
... // now register some objects
// now register any needed IObjectPostProcessors
MyObjectPostProcessor pp = new MyObjectPostProcessor();
factory.AddObjectPostProcessor(pp);
// now start using the factory
...
```

由于手工注册的步骤并不方便，而 IApplicationContext 在功能上又是 IObjectFactory 的超集，所以，当需要使用对象后处理器的时候，推荐使用 IApplicationContext 作为容器。

4.9.使用 IObjectFactoryPostProcessor 定制对象工厂

实现了 Spring.Objects.Factory.Config.IObjectFactoryPostProcessor 接口的类称为工厂后处理器（ojbect factory post-processor）。工厂后处理器用于手工（对于 IObjectFactory 来说）或自动（对于 IApplicationContext 来说）的改变整个容器的行为。通过实现这些接口，就可以在对象定义被载入容器之后并且在被实例化之前执行某些行为。这样我们就有机会修改容器内对象的配置信息，或在容器开始创建其中的对象前执行某些初始化操作。该接口定义如下：

```
public interface IObjectFactoryPostProcessor
{
  void PostProcessObjectFactory (IConfigurableListableObjectFactory
factory);
}
```

一般来说，如果希望执行与容器内对象生命周期有关的操作，可以借助 Spring.Object.Factory.Config 命名空间提供的工厂后处理器类型，它们的功能可满足任何需要。例如，使用工厂后处理器可以动态替换容器中某些对象的属性值。开发人员可以通过实现 IObjectFactoryPostProcessor 接口来创建自定义的工厂后处理器。Spring.NET 内置了一部分工厂后处理器，例如 PropertyResourceConfigurer 和 PropertyPlaceHolderConfigurer，本节稍后对这两个类进行讨论。

对 IObjectFactory 来说，需要手工添加工厂后处理器，如下代码所示：

```
XmlObjectFactory factory = new XmlObjectFactory(new
FileSystemResource("objects.xml"));
// create placeholderconfigurer to bring in some property
// values from a Properties file
PropertyPlaceholderConfigurer cfg = new
PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("ado.properties"));
// now actually do the replacement
cfg.PostProcessObjectFactory(factory);
```

IApplicationContext 则会检测其中的对象，如果某对象实现了 IObjectFactoryPostProcessor 接口，IApplicationContext 就会自动将其作为工厂后处理器并在适当的时候使用。工厂后处理器对象的定义方法和普通对象一样。

由于手工注册的步骤并不方便，且 IApplicationContext 在功能上又是 IObjectFactory 的超集，所以，当需要使用工厂后处理器的时候，推荐使用 IApplicationContext 作为容器。

4.9.1.PropertyPlaceholderConfigurer 类

如果需要在运行时从配置文件中动态导出某些值来为对象进行注入，PropertyPlaceholderConfigurer 可提供一个极好的解决方案。比如说，应用程序的布署人员经常需要设置数据库的 URL，用户名及密码。当然可以让他们直接去修改所有相关对象定义的属性值，但这么做是有风险的，较好的方法是使用 PropertyPlaceHolderConfigurer，在运行时让容器用配置文件中的某个值来替换这些对象的属性值。（按：本例的意思是说，布署人员可以逐个修改容器中使用了诸如连接字符串等信息的对象定义，但这些信息往往是由应用程序的多个模

块共享的，这么做难免会有遗漏或导致某些使用同样信息的对象得到不一致的值，而使用 PropertyPlaceHolderConfigurer 可以将这些信息定义在一个集中的位置，在运行期使用这些信息对相关的属性值进行替换，这和我们在代码中将连接字符串定义为一个常量，在需要时使用该常量而非硬编码的做法是类似的。）

注意 IApplicationContext 可以自动识别并应用实现了 IObjectFactoryPostProcessor 接口的对象。所以在需要使用对象工厂后处理器的时候，最好用 IApplicaitonContext 作为容器。

在下面的例子中，需要为一个数据访问对象配置数据库连接和最大查询记录数。在配置文件中，将这两个属性值各定义为一个键值对，同时注册了一个可以处理键值对的节点处理器：System.Configuration.NameValueSectionHandler；在对象定义中，如果需要用到这两个值，不要用硬编码把值直接定义给相关的属性，而是使用一个 NAnt 风格的占位符。配置文件如下：

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core"/>
    </sectionGroup>
    <section name="DaoConfiguration"
type="System.Configuration.NameValueSectionHandler"/>

    <section name="DatabaseConfiguration"
type="System.Configuration.NameValueSectionHandler"/>
  </configSections>

  <DaoConfiguration>
    <add key="maxResults" value="1000"/>
  </DaoConfiguration>

  <DatabaseConfiguration>

    <add key="connection.string"
value="dsn=MyDSN;uid=sa;pwd=myPassword;"/>
  </DatabaseConfiguration>

  <spring>
    <context>
      <resource uri="assembly://DaoApp/DaoApp/objects.xml"/>
    </context>

  </spring>
```

```
</configuration>
```

注意配置文件中出现的两个键值对配置（NameValueSection），它们的值将被其它对象定义引用。在本例中，我们使用内嵌在程序集资源中的对象定义文件，以减少在布署时被任意篡改的可能。对象定义如下：

```
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net

        http://www.springframework.net/xsd/spring-objects.xsd" >

    <object name="productDao" type="DaoApp.SimpleProductDao, DaoApp ">
      <property name="maxResults" value="${maxResults}"/>
      <property name="dbConnection" ref="myConnection"/>
    </object>

    <object name="myConnection" type="System.Data.Odbc.OdbcConnection,
System.Data">
      <property name="connectionstring" value="${connection.string}"/>
    </object>

    <object name="appConfigPropertyHolder"

type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer,
Spring.Core">

        <property name="configSections">
          <value>DaoConfiguration,DatabaseConfiguration</value>
        </property>

    </object>
</objects>
```

其中，属性 MaxResults 和 ConnectionString 的值分别用占位符${maxResults}和${connection.string}表示，这和前面定义的两个键/值对的键值是匹配的。容器中定义了一个类型为 PropertyPlaceHolderConfigurer 的对象，它通过 ConfigSections 属性来指定占位符要引用的键/值对子节点的名称，其值是一个以逗号分隔的字符串，可以表示多个子节点名称。

PropertyPlaceholderConfigurer 也可以从其它 IResource 位置中读取键/值对。可通过 PropertyPlaceholderConfigurer 类的 Location 和 Locations 属性指定这些位置。

如果在不同的位置定义了相同名称的键，默认的行为是：后处理的值会覆盖先处理的值。该行为由 PropertyPlaceholderConfigurer 的 LastLocationOverrides 属性控制，值为 true 时允许覆盖；为 false 时，指定的值将添加到键值对列表中，同 NameValueCollection.Add 方法的行为一样。

注意

在 ASP.NET 环境下定义 NameValueFileSectionHandler 时，必须使用它的四段全名，如下...

```
<section name="hibernateConfiguration"
 type="System.Configuration.NameValueFileSectionHandler, System,
 Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"/>
```

4.9.1.1.使用环境变量进行替换

也可用 PropertyPlaceholderConfigurer 类将占位符替换为环境变量的值（按：占位符格式同上，为：${环境变量名}）。替换的行为由 PropertyPlaceholderConfigurer 类的 EnvironmentVariableMode 属性控制，该属性的类型为 EnvironmentVariableMode 枚举，有三个可用值：Never、Fallback 和 Override。其中 Fallback 是默认值，会尝试用环境变量替换未在自定义键值对中找到匹配项的占位符；Override 则不理会自定义键值对中是否有与占位符匹配的键，直接用环境变量值替换占位符；Nerver 的含义更明确，即不使用环境变量进行替换。见下例：

```
<object name="appConfigPropertyHolder"

type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer,
Spring.Core">
        <property name="configSections"
value="DaoConfiguration,DatabaseConfiguration"/>
        <property name="EnvironmentVariableMode" value="Override"/>
    </object>
</objects>
```

（按：按照上面例子配置好之后，如果一个对象定义的某个属性值为 ${JAVA_HOME}，如下：

```
<property name="Msg" value="${JAVA_HOME}"/>
```

那么，在运行时该属性的值就会被赋予本机环境变量 JAVA_HOME 的值。）

4.9.2. PropertyOverrideConfigurer 类

PropertyOverrideConfigurer 是 Spring.NET 中的另一个工厂后处理器，与 PropertyPlaceholderConfigurer 不同的是，（按：替换的依据不是占位符，而是属性名），要被替换的属性可以定义初始值甚至完全不定义值（按：这个说法不准确，因为“不定义值”的属性是非法的，也就是类似<property name="Msg"/>的定义会引起异常；原意是说，要被替换的属性可以用一个空的 value 元素赋值，或用<null/>赋值，即：<property name="Msg"><value></value></propert>或<property name="Msg"><null/></property>）。如果没有为某个属性找到用于替换的匹配项，容器就会使用对象定义中的初始值为属性赋值。（按：如果在用于替换的键/值对中定义了一个找不到目标属性的键，就会抛出异常；例如在稍后的例子中，如果 productDao 对象另有一个属性 MinResults，而节点 DaoConfigurationOverride 中并未定义键名为 productDao.MinResults 的项，那么容器在运行时就按普通的方式给 minResult 赋值；但是，如果节点 DaoConfigurationOverride 中定义了一个键名为 productDao.MaximunResults 的项，而对象 productDao 却没有这个属性，此时便会报错。这是显而易见的，在此仅做提醒。）

注意：对象定义并不会“知道”它的属性值被覆盖了，所以从配置文件中不会马上看到结果。如果在容器中为同一个对象属性配置了多个 PropertyOverrideConfigurer 对象，那么只有最后一个会起作用（因为这是一种覆盖的机制）。

该类的用法和 PropertyPlaceHolderConfigurer 大致相同，除了一点：定义键值对时，键名必须和要覆盖的属性名完全一致（按：全名，即"对象名.属性名"的格式）。参见下面的例子：

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core"/>
    </sectionGroup>
    <section name="DaoConfigurationOverride"
type="System.Configuration.NameValueSectionHandler"/>

  </configSections>

  <DaoConfigurationOverride>
    <add key="productDao.maxResults" value="1000"/>
  </DaoConfigurationOverride>

  <spring>
    <context>
```

```
      <resource uri="assembly://DaoApp/DaoApp/objects.xml"/>
    </context>
  </spring>

</configuration>
```

随后，Spring.NET 会根据上面定义的键/值对从容器中查找名为 productDao 的对象，并将其属性 MaxResults 的值替换为 1000。相关的对象定义如下：

```
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net
http://www.springframework.net/xsd/spring-objects.xsd" >

    <object name="productDao" type="PropPlayApp.SimpleProductDao,
PropPlayApp " >
      <property name="maxResults" value="2000"/>
      <property name="dbConnection" ref="myConnection"/>
      <property name="log" ref="daoLog"/>
    </object>


    <object name="daoLog"
type="Spring.Objects.Factory.Config.Log4NetFactoryObject,
Spring.Core">
      <property name="logName" value="DAOLogger"/>
    </object>

    <object name="myConnection" type="System.Data.Odbc.OdbcConnection,
System.Data">
      <property name="connectionstring">
        <value>dsn=MyDSN;uid=sa;pwd=myPassword;</value>

      </property>
    </object>

    <object name="appConfigPropertyOverride"
type="Spring.Objects.Factory.Config.PropertyOverrideConfigurer,
Spring.Core">
      <property name="configSections">
        <value>DaoConfigurationOverride</value>
      </property>
    </object>
```

```
</objects>
```

4.9.3. IVariableSource 接口

IVariableSource 接口是属性值替换的基础，它可以从各种变量源（Variable Source）中为属性占位符读取“名称－值”对的值。目前 Spring.NET 支持一系列变量源，允许用户从.NET 配置文件、Java 风格的属性文件、环境变量、命令行参数、注册表以及.NET2.0 新增的连接字符串配置节点中读取变量值。该接口的各个实现类如下所示，详细信息请参考 SDK 文档。

- ConfigSectionVariableSource
- PropertyFileVariableSource
- EnvironmentVariableSource
- CommandLineArgsVariableSource
- RegistryVariableSource
- SpecialFolderVariableSource
- ConnectionStringsVariableSource

4.10.使用 alias 节点为对象添加别名

在对象定义中，可以使用 id 和 name 属性为对象设置一个以上名称，相关内容在 4.2.5,对象标识符（id 和 name） 中提到过。但是，如果想从多个文件组合应用程序的配置信息，这种定义别名的方式就有一定的局限，详细内容可参见 4.15,从其它文件中导入对象定义，如果在应用程序中用一个配置文件来表示一个逻辑层或一个组件，会经常用到配置文件的导入功能。此时我们可能希望通过唯一的 name 属性来引用一个公共依赖项，如果公共依赖项定义在应用程序的主配置（按：也就是其它容器的父容器配置）中，我们就可以使用它的 name 属性作为别名。但是如果公共依赖项没有定义在主配置中，就不一定能通过 name 属性来引用它，因为从逻辑上说，该依赖项属于另外一个层次或组件。

在此情况下，可以在主应用程序配置中用 alias 元素显式的定义一个别名：

```
<alias name="fromName" alias="toName"/>
```

这样，就允许用 toName 跨文件访问名为 fromName 的对象了。

来看一个具体的例子，假如在配置文件"a.xml"（代表组件 A）定义了一个名为 componentA-connection 的连接对象。在另一配置文件"b.xml"中（表示组件 B）希望通过名称 componentB-connection 来引用该连接对象；并且，主应用程序 MyApp 也定义了自己的容器，希望以 myApp-connection 为名引用该连接对象。在运行期这三个配置文件会被组合进主配置文件。通过在主配置文件中添加下面的元素，可以很容易的处理这个问题：

```
<alias name="componentA-connection" alias="componentB-connection"/>
<alias name="componentA-connection" alias="myApp-connection"/>
```

现在，所有组件以及主程序都能通过自己希望的名称访问连接对象，且能避免其它对象产生命名冲突（特别是当使用了命名空间时），但它们引用的都是同一对象。

4.11. IApplicationContext 简介

Spring.Objects 命名空间提供了管理和操作对象的基本功能，但大多数需要以编程的方式来处理，Spring.Context 命名空间的 IApplicationContext 接口则扩展了 IObjectFactory 接口的功能。

上下文模型的基础就是 IApplicationContext 接口，该接口位于 Spring.Context 命名空间。Spring.Context 命名空间还提供了以下功能，用以支持面向框架的编程方式及上下文嵌套等。

- 载入多个（分层、嵌套的）上下文，允许将某些上下文的使用范围限制在特定的层次，比如应用程序的 web 层。
- 通过实现 IMessageSource 接口，在应用程序层访问本地化资源。
- 通过实现 IResourceLoader，用统一的方式访问包括 URL 和文件系统在内的各种资源。
- 松耦合的事件传播机制，事件的发布者和订阅者无需了解对方，而是通过应用程序上下文间接关联自己感兴趣的事件。

因为 IApplicationContext 通过扩展 IObjectFactory 接口而继承了其所有功能，所以在应用时推荐使用 IApplicationContext 作为容器。框架中只实现了 IObjectFactory 接口的类，例如 XmlObjectFactory，是最初在 java 环境下为支持特定资源环境而创建的类型。

因为 IApplicationContext 是 IObjectFactory 的子接口，所以也可以使用 XML 来配置对象（参考 4.2.1, IObjectFactory 和 IApplicationContext 以及附录 A, Spring.NET 的 spring-objects.xsd）并通过唯一标识符来访问对象。同样，IApplicationContext 也具有前面讨论过的所有功能，比如自动装配、属性和构造器参数依赖注入、依赖检查、生命周期接口等等。

下面讨论 IApplicationContext 接口特有的功能，包括：

- Singleton 模式的服务定位器访问方式。参见 4.16, 服务定位器访问。
- 自动注册实现了 IObjectPostProcessor 接口的类。参见 4.14. 定制 IApplicationContex 中对象的行为。
- 注册由 IApplicationContext 内部使用的类型，比如资源协议处理器、用来解析对象定义的 XML 解析器、以及类型别名等。参见 4.12 节, 配置应用程序上下文。

4.12.配置应用程序上下文

除了前面讲过的 context 和 objects 节点，还可以在.NET 应用程序配置文件中添加其它节点来注册资源处理器、自定义解析器、类型别名和自定义类型转换器等。下面的配置文件包含了所有可能的节点。每个节点都需要定义相应的节点处理器。注意下面定义的资源处理器和解析器都是虚构的。

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />


      <section name="parsers"
type="Spring.Context.Support.ConfigParsersSectionHandler,
Spring.Core"/>
      <section name="resources"
type="Spring.Context.Support.ResourcesSectionHandler, Spring.Core"/>
      <section name="typeAliases"
type="Spring.Context.Support.TypeAliasesSectionHandler,
Spring.Core"/>
      <section name="typeConverters"
type="Spring.Context.Support.TypeConvertersSectionHandler,
Spring.Core"/>

    </sectionGroup>
  </configSections>

  <spring>
    <parsers>
      <parser namespace="http://www.springframework.net/aop"
              type="Spring.Aop.Support.AopConfigParser, Spring.Aop"

schemaLocation="assembly://Spring.Aop/Spring.Aop.Support/spring-aop.x
sd"/>
      <parser namespace="http://www.springframework.net/web"
              type="Spring.Web.Support.WebConfigParser, Spring.Web"
```

```
schemaLocation="assembly://Spring.Web/Spring.Web.Support/spring-web.x
sd"/>
    </parsers>

    <resources>
      <resource protocol="db"
type="MyCompany.MyApp.Resources.MyDbResource"/>
    </resources>

    <context caseSensitive="false">

      <resource uri="config://spring/objects"/>
      <resource uri="db://user:pass@dbName/MyDefinitionsTable"/>
    </context>

    <typeAliases>
       <alias name="WebServiceExporter"
type="Spring.Web.Services.WebServiceExporter, Spring.Web"/>
       <alias name="DefaultPointcutAdvisor"
type="Spring.Aop.Support.DefaultPointcutAdvisor, Spring.Aop"/>

       <alias name="AttributePointcut"
type="Spring.Aop.Support.AttributeMatchMethodPointcut, Spring.Aop"/>
       <alias name="CacheAttribute"
type="Spring.Attributes.CacheAttribute, Spring.Core"/>
       <alias name="MyType"
type="MyCompany.MyProject.MyNamespace.MyType, MyAssembly"/>
     </typeAliases>

    <typeConverters>
      <converter for="Spring.Expressions.IExpression, Spring.Core"
type="Spring.Objects.TypeConverters.ExpressionConverter,
Spring.Core"/>

      <converter for="MyTypeAlias"
type="MyCompany.MyProject.Converters.MyTypeConverter, MyAssembly"/>
    </typeConverters>

    <objects xmlns="http://www.springframework.net">
        ...
    </objects>

  </spring>
```

```
</configuration>
```

后面几个小节会讨论配置中新出现的几个节点。context 节点中的 caseSensitive 属性用于控制容器是否对对象名称的大小写敏感。在 Web 应用程序中这个属性很重要，可以让 ASP.NET 按大小写无关的方式解析页面。该属性的默认值是 true。

4.12.1.注册自定义解析器

在定义对象属性和依赖关系时，除了使用默认的 XML schema，也可以针对不同的应用领域创建专门的 schema。好处是可能减少键入工作，并可利用 schema 进行 XML 节点的代码提示，缺点是需要自己去写将这种 XML 节点转换为 Spring 对象定义的代码。一般来说，要创建自定义解析器，可继承 DefaultXmlObjectDefinitionParser 类，并覆盖其中的 int ParseRootElement(XmlElement root,XmlResourceReader reader)方法和 intParseElement(XmlElement element, XmlResourceReader reader)方法。

4.12.2.创建自定义资源处理器

自定义资源处理器需实现 IResource 接口。我们可以从 AbstractResource 抽象基类继承。请参见 Spring.NET 源码中 FileSystemResource 和 AssemblyResource 等类型的实现方式。我们既可以在 App.config 中注册资源处理器，就象前面例子中注册 ResourcesSectionHandler 那样，也可以通过创建类型为 Spring.Objects.Factory.Config.ResourceHandlerConfigurer 的对象定义来注册资源处理器，如下：

```
<object id="myResourceHandlers"
type="Spring.Objects.Factory.Config.ResourceHandlerConfigurer,
Spring.Core">
      <property name="ResourceHandlers">
          <dictionary>
              <entry key="db"
value="MyCompany.MyApp.Resources.MyDbResource, MyAssembly"/>
          </dictionary>
      </property>

</object>
```

4.12.3.注册类型别名

作为类型全名的替代物，类型别名可以简化 Spring.NET 的配置文件。别名可以在 config 文件中注册，也可以通过编程方式注册，注册之后就可以在对象定义中任何需要类型全名的地方使用。也可以为泛型类定义类型别名。

若要配置类型别名，一种方式是在 Web/App.config 文件中添加 typeAliases 节点和相应的节点处理器。请参看本节一开始的例子，其中为 WebServiceExporter 类定义了一个类型别名。一旦定义了类型别名，就可以象全名一样使用它们：

```
<object id="MyWebService" type="WebServiceExporter">
    ...
</object>

<object id="cacheAspect" type="DefaultPointcutAdvisor">
    <property name="Pointcut">
        <object type="AttributePointcut">
            <property name="Attribute" value="CacheAttribute"/>
        </object>

    </property>
    <property name="Advice" ref="aspNetCacheAdvice"/>
</object>
```

关于如何为泛型类型定义类型别名，请参看 4.2.4,创建泛型类型的对象。

另外，也可以通过 Spring.Objects.Factory.Config.TypeAliasConfigurer 类来注册类型别名。这种定义方法更加模块化，如果没有 App/Web.config，也可以通过这种方式注册别名。请看下面的例子：

```
<object id="myTypeAlias"
type="Spring.Objects.Factory.Config.TypeAliasConfigurer,
Spring.Core">
      <property name="TypeAliases">
          <dictionary>
              <entry key="WebServiceExporter"
value="Spring.Web.Services.WebServiceExporter, Spring.Web"/>

              <entry key="DefaultPointcutAdvisor"
value="Spring.Aop.Support.DefaultPointcutAdvisor, Spring.Aop"/>
              <entry key="MyType"
value="MyCompany.MyProject.MyNamespace.MyType, MyAssembly"/>
          </dictionary>
      </property>
```

```
</object>
```

4.12.4.注册类型转换器

在.NET 中，注册类型转换器的标准方法是为类型应用 TypeConverter 特性来为其指定转换器。如果可以控制类的源代码，这么做是最合适的。而对于无法控制源码的类型，Spring.NET 则可以通过配置文件为任何类型添加转换器（如前面的例子），也可以覆盖.NET 基础类库为某类型定义的标准类型转换器。

我们既可以象前文例子一样，先在 App.config 中注册类型为 Spring.Context.Support.TypeConvertersSectionHandler 的节点处理器，然后再注册类型转换器，也可以通过 Spring.Objects.Factory.Config.CustomConverterConfigurer 来注册类型转换器。请看下面的例子：

```
<object id="myTypeConverters"
type="Spring.Objects.Factory.Config.CustomConverterConfigurer,
Spring.Core">
      <property name="CustomConverters">
          <dictionary>
              <entry key="System.Date"
value="MyCompany.MyProject.MyNamespace.MyCustomDateConverter,
MyAssembly"/>
          </dictionary>
      </property>

</object>
```

4.13.IApplicationContext 接口的扩展功能

前面提到过，IApplicationContext 有一些 IObjectFactory 不具备的功能。下面我们来逐一进行讨论：

4.13.1.上下文继承

可以用分层的（继承的）方式定义应用程序上下文，以便自然的反映应用程序的各个层次。比如说，可以将抽象的对象定义集中放在一个父上下文配置中，并将配置文件嵌入到程序集中以避免不必要的改动。

```
<spring>
  <context>
```

```
    <resource
uri="assembly://MyAssembly/MyProject/root-objects.xml"/>
    <context name="mySubContext">
      <resource uri="file://objects.xml"/>
    </context>

  </context>
</spring>
```

上例中，嵌套的 context 节点表明它与外层的 context 是父子的继承关系。嵌套的深度是没有限制的，但在实际应用中一般不会把程序层次划分得太深。resource 节点所指定的 xml 文件必须以<objects>作为根节点。另外，也可以直接将对象定义放在.config 配置文件中，如下：

```
<configSections>
  <sectionGroup name="spring">
    <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core"/>
    <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    <sectionGroup name="child">
      <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />

    </sectionGroup>
  </sectionGroup>
</configSections>

<spring>

  <context name="ParentContext">
    <resource uri="config://spring/objects"/>

    <context name="ChildContext">
      <resource uri="config://spring/child/objects"/>
    </context>
  </context>

  <objects xmlns="http://www.springframework.net">

    ...

  </objects>
```

```
  <child>
    <objects xmlns="http://www.springframework.net">

      ...

    </objects>
  </child>

</spring>
```

注意，在 Windows 应用中，context 的 type 属性是可选的，默认值是 Spring.Context.Support.XmlApplicationContext。context 的 name 属性值可以在服务定位器类 ContextRegistry 中使用，请参见 4.16 节,服务定位器访问。

4.13.2.使用 IMessageSource 接口

IApplicationContext 本身也是 IMessageSource 接口的子接口，提供了针对文本和其它诸如图像等数据的本地化（或国际化）服务。这些功能在应用层简化了.NET 的本地化功能，且能够进行资源缓存，所以在性能上也有所增强。IMessageSource 和用于解析嵌套消息的 IHierarchicalMessageSource 接口一起，构成了 Spring.NET 处理本地化时的基本接口。（按：原文中说的是 NestingMessageSource 类，但 Spring.NET 中并没有这个类，根据下文的描述，应该是 IHierarchicalMessageSource 接口，请参阅原文），我们先大致看一下 IMessageSource 接口定义的方法：

- string GetMessage(string name)： 以 CultureInfo 类的 CurrentUICulture 属性值为语言文化信息，从 IMessageSource 返回一个指定名称的消息。
- string GetMessage(string name, CultureInfo cultureInfo)：使用指定的名称和语言文化信息从 IMessageSource 中获取消息。
- string GetMessage(string name, params object[] args)：以 CultureInfo.CurrentUICulture 的值为语言文化信息，从 IMessageSource 中返回指定名称的消息，并用 args 参数的值依次替换消息中的占位符。
- string GetMessage(string name, CultureInfo cultureInfo, params object[] args)：根据指定的名称和语言文化信息从 IMessageSource 中获取消息，并用 args 参数的值依次替换其中的占位符。
- string GetMessage(IMessageSourceResolvable resolvable, CultureInfo culture)：将前面几个方法用到的所有信息包装在一个 IMessageSourceResolvable 中，并结合指定的语言文化信息返回一个消息。
- object GetResourceObject(string name)：根据指定名称返回一个资源对象，如图标、图像等，语言文化则使用 CultureInfo.CurrentUICulture 的值。

- object GetResourceObject(string name, CultureInfo cultureInfo)：根据指定的名称和语言文化信息返回一个资源对象，如图标、图像等。
- void ApplyResources(object value, string objectName, CultureInfo cultureInfo)：利用 ComponentResourceManager 类将资源中键值与指定名称相匹配的项赋值给指定对象的同名属性，资源中键的格式应是“对象名.属性名”。（按：原文中，包括 API 参考文档中对该方法的描述都很含糊，在这里：第一个参数 value 是要进行资源应用的目标对象，即所谓的“指定对象“，而第二个参数 objectName 则是一个定义在资源文件中的、逻辑上的“对象“名称，也是搜索键值的依据，即所谓的“指定名称“：以一个例子来说明，假如有一个对象 p，包含 Name 和 Age 属性；而在资源文件中，有两个键值分别为 person.obj.Age 和 person.obj.Name 的项，这两个项表示，在资源文件中定义了一个名为 person.obj 的“对象”，它也包含两个属性，Age 和 Name；当调用 ctx.ApplyResources(p,"person.obj",CultureInfo.CurrentUICulture) 时，就会在资源中查找名为“person.obj”的“对象”，并把它的“属性”值赋值给对象 p 的同名属性。）

IApplicationContext 会在加载时自动搜索其中定义的 IMessageSource 对象，该对象必须以 messageSource 为名。如果找到了这个对象，上面讲到的所有方法都会由该对象代理。如果找不到，容器会查看它的父容器是否定义了名为 messageSource 的 IMessageSource 对象，如果找得到就使用父容器中的 messageSource 对象，如果找不到，则会实例化一个空的 StaticMessageSource 来代理以上的方法调用。

注意

.NET 处理资源本地化的默认方法似乎有个 bug，在.NET 1.1 的 SP1 中已经被修正了。这会影响 GetMessage 方法重载中包含 CultureInfo 参数的版本，因为它们都分别调用了.NET 基础类库中的 ResourceManager.GetObject 方法的相应重载版本。

Spring.NET 为 IMessageSource 接口定义了两个实现类，分别是 ResourcesetMessageSource 和 StaticMessageSource。这两个类同时也实现了 IHierarchicalMessageSource 接口，以解析嵌套的消息。StaticMessageSource 类基本用不到，不过可以通过它用编程方式向消息源添加消息。ResourceSetMessageSource 的功能要丰富的多，在第二十七章,IoC 快速入门中有这个类的例子。ResourceSetMessageSource 类要用一系列 ResourceManager 来进行配置，当需要解析一个消息时，就会在 ResourceManager 列表中查找合适的项来解析，Spring.NET 会为每个 ResourceManager 返回一个 SesourceSet 对象来执行实际的解析工作。注意这里的查找行为并未替换标准的星型（hub-and-spoke）本地化资源查找方式，而是通过 ResourceManager 列表为标准的星型资源查找指定了多个“中心”。

```
<object name="messageSource"
type="Spring.Context.Support.ResourceSetMessageSource, Spring.Core">
  <property name="resourceManagers">
```

```
    <list>
       <value>Spring.Examples.AppContext.MyResource,
Spring.Examples.AppContext</value>
    </list>

  </property>
</object>
```

在配置 ResourceManagers 属性时，可以用包含资源 BaseName 和程序集名称的字符串设置其中的一项，如上面的代码所示。这个字符串会被 ResourceManagerConverter 转换成一个 ResourceManager 对象。任何包含 ResourceManager 类型属性的对象都可以用这种方式设置。另外，也可以单独配置一个 ResourceManager 对象，由 ResourceManagers 属性引用。在配置文件中可以使用 Spring.Object.Factory.Config.ResourceManagerFactoryObject 类，该类用于创建一个 ResourceManager 对象。

```
<object name="myResourceManager"
type="Spring.Objects.Factory.Config.ResourceManagerFactoryObject,
Spring.Core">
  <property name="baseName">
    <value>Spring.Examples.AppContext.MyResource</value>
  </property>
  <property name="assemblyName">

    <value>Spring.Examples.AppContext</value>
  </property>
</object>
```

在代码中，GetMessage 方法会根据给定的键值从资源中返回一个消息字符串。返回值中的占位符会按与 String.Format 方法相同的方式进行替换。返回的字符串，以及相关的 ResourceManager 和 ResourceSet 都会被缓存以提高性能。假如在资源文件中，一个键为 HelloMessage 的字符串的值为"Hello {0} {1}"，那么下面的代码就会返回 Hello Mr. Anderson，注意在缓存 ResourceSet 时，键值是由 ResourceManager 的名称和语言文化信息字符串组合而成的，可以保证对每种语言文化都是唯一的：

```
string msg = ctx.GetMessage("HelloMessage",
                            new object[] {"Mr.", "Anderson"},
                            CultureInfo.CurrentCulture );
```

注意消息的解析可以链式进行，只需要向 GetMessage 传递特定的参数，这些参数本身也代表资源中可以被解析的消息。这种链式的解析方式可以使消息的定义和解析更为灵活。为此，需要将实现了 IMessageResolvable 接口的对象作为 GetMessage 方法的参数。在代码中可以直接使用 DefaultMessageResolvable 类。比如，如果在资源文件中定义了键为 error.required，值为"{0 is required

{1}}"；以及键为 field.firstname，值为"First name"的项。那么下面的代码将返回”First name is required dude!“：

```
string[] codes = {"field.firstname"};
DefaultMessageResolvable dmr = new DefaultMessageResolvable(codes,
null);
ctx.GetMessage("error.required",
                new object[] { dmr, "dude!" },
                 CultureInfo.CurrentCulture ));
```

可参考随 Spring.NET 一起发布的示例项目 Spring.Examples.AppContext，以了解这些功能的用法。

4.13.3.在 Spring.NET 内部使用资源

很多应用程序都需要访问资源。这里说的资源可以是文件，也可以来自 internet 或是普通的网页。Spring.NET 用一种干净、透明、与协议无关的方式来访问资源。IApplicationContext 的 GetResourse(string)方法可以处理这些功能。请参考 6.1,简介以了解其参数的格式和 IResource 接口的用法。

4.13.4.松耦合事件模型

Spring.NET 允许开发人员用松耦合的事件装配机制来处理事件。通过解耦事件的发布者和订阅者，大部分事件装配工作都可由 IoC 容器处理。事件的发布者可以向事件注册中心发布它们的所有事件，或者以事件委托的类型、名称及返回值为过滤依据发布部分事件。事件的订阅者可以订阅任意数量的已发布事件。订阅者可以用同一个订阅者处理某个类型所有实例的所有事件，而不理会究竟为该类型创建了多少实例；也可以名称、委托类型等等为依据对事件进行选择性的订阅，并且能够以正则表达式作为事件订阅的过滤依据。

Spring.Objects.Events.IEventRegistry 接口封装了事件注册中心的功能，该接口定义的方法用于发布和订阅事件。

- void PublishEvents(object sourceObject)：发布 sourceObject 的所有事件，以供具备适当处理器的订阅者订阅
- void Subscribe(object subscriber)：在事件中心为 subscriber 订阅能处理的所有事件
- void Subscribe(object subscriber, Type targetSourceType)：subscriber 向事件中心订阅它能够处理的、由某个特定类型的源对象发布的所有事件

IApplicationContext 扩展了该接口，并用 Spring.Objects.Events.Support.EventRegistry 对象代理该接口的方法。可以

在应用程序中创建任意多的 EventRegistry，但通常只会用到一个，使用 IApplicationContext 接口可以很方便的访问 EventRegistry。（按：即 IApplicationContext 本身就是一个事件注册中心）

在 example/Spring/Spring.Examples.EventRegistry 目录下，有一个关于事件注册的示例项目。请注意其中的 EventRegistryApp.cs 文件。程序从配置中创建了应用程序上下文，并请求了一个 MyEventPublisher 对象和两个 MyEventSubscriber 对象。MyEventPublisher 对象将自己的事件发布给了 IApplicationContext：

```
// Create the Application context using configuration file
IApplicationContext ctx = ContextRegistry.GetContext();

// Gets the publisher from the application context
MyEventPublisher publisher =
(MyEventPublisher)ctx.GetObject("MyEventPublisher");

// Publishes events to the context.
ctx.PublishEvents( publisher );
```

我们以发布者的类型（即 MyEventPublisher 类）为过滤标准，为其中一个 MyEventSubscriber 对象订阅容器中所有满足条件的事件，如下：

```
// Gets first instance of subscriber
MyEventSubscriber subscriber =
(MyEventSubscriber)ctx.GetObject("MyEventSubscriber");

// Gets second instance of subscriber
MyEventSubscriber subscriber2 =
(MyEventSubscriber)ctx.GetObject("MyEventSubscriber");

// Subscribes the first instance to the any events published by the type
MyEventPublisher
ctx.Subscribe( subscriber, typeof(MyEventPublisher) );
```

这段代码将 suscriber1 和事件发布者关联起来。发布者在任何时候触发的事件（publisher.ClientMethodThatTriggersEvent1();）都会被 suscriber1 处理，但 suscriber2 则不会响应任何事件。这样，就可以有选择的订阅事件，而不管事件订阅者的布署模式是 Singleton 还是 Prototype 模式布署。（按：这句话的意思是说，是否响应事件只与订阅者对象的引用有关，而与订阅者对象的布署方式无关;以上例来说,假设MyEventSubscriber对象是以Singleton模式布署的，那么 subscriber2 所引用的对象就和 subscriber1 相同，但如果用 Subscribe 方法为 subscriber2 也订阅了事件,那么虽然这两个引用所指向的对象是一样的，MyEventPublisher 对象的事件还是会有两个不同的订阅者，当事件触发时，分别会被 subscriber1 和 subscriber2 响应两次。）

4.13.5.IApplicationContext 的事件通知

IApplicationContext 本身事件的处理是通过 IApplicationListener 接口完成的，该接口只有一个方法 void OnApplicationEvent(object source,ApplicationEventArgs applicationEventArgs)。实现了该接口的类会被自动注册为 IApplicationContext 事件的侦听器。事件的发布通过 IApplicationContext 的 PulishEvent(ApplicationEventArgs eventArgs)方法完成（按：注意不是前文的 PublishEvents 方法）。这种处理方式基于传统的观察者模式。

事件参数类 ApplicationEventArgs 保存了事件触发的时间。其子类 ContextEventArgs 可将应用程序上下文的生命周期事件通知给所有观察者。ContextEventArgs 类定义一个 ContextEvent 类型的 Event 属性，ContextEvent 是一个枚举类型，包含两个值：Refreshed 和 Closed，Refreshed 表示 IApplicationContext 被初始化或者被刷新了，此处的初始化是指所有的对象都被载入容器、以 singleton 模式布署的对象已经被初始化并且 IApplicationContext 已经可以使用。Closed 表示调用了 IConfigurableApplicationContext 接口的 Dispose 方法将 IApplicationContext 关闭了,在此,关闭是指所有的 singleton 对象都被销毁。

我们也可以实现自定义的事件。只需要调用 IApplicationContext 的 PulishEvent 方法将自定义事件参数类的实例作为参数传入即可：

我们来看一个例子，首先是配置文件：

```
<object id="emailer" type="Example.EmailObject">
    <property name="blackList">
        <list>
            <value>black@list.org</value>
            <value>white@list.org</value>

            <value>john@doe.org</value>
        </list>
    </property>
</object>

<object id="blackListListener" type="Example.BlackListNotifier">
    <property name="notificationAddress">

        <value>spam@list.org</value>
    </property>
</object>
```

然后是代码：

```
public class EmailObject : IApplicationContextAware {

    // the blacklist
    private IList blackList;

    public IList BlackList
    {
      set { this.blackList = value; }
    }

    public IApplicationContext ApplicationContext
    {
      set { this.ctx = value; }
    }

    public void SendEmail(string address, string text) {
        if (blackList.contains(address))
        {
            BlackListEvent evt = new BlackListEvent(address, text);
            ctx.publishEvent(evt);
            return;
        }
        // send email...
    }
}

public class BlackListNotifier : IApplicationListener
{

    // notification address
    private string notificationAddress;

    public string NotificationAddress
    {
      set { this.notificationAddress = value; }
    }

    public void OnApplicationEvent(ApplicationEvent evt)
    {
        if (evt instanceof BlackListEvent)
        {
            // notify appropriate person
        }
    }
```

```
}
```

4.14.定制 IApplicationContex 中对象的行为

IObjectFactory 接口已经提供了一系列机制来控制其中对象的生命周期（例如使用 IInitializingObject 和 System.IDisposable 接口，它们的作用和在对象定义中使用 init-method、destroy-method 属性一样），同时可以用对象后处理器自定义容器内的任意对象。在 IApplicationContext 中，这些机制仍然可用，除此以外还添加了新的功能来定制容器中对象的其它行为。

4.14.1.IApplicationContextAware 标识接口

原来用于 IObjectFactory 的所有标识接口现在都是可用的。同时 IApplicationContext 也添加了一个新的接口：IApplicationContextAware。该接口只有一个 ApplicationContext 属性，当容器创建其实现类的对象时，会将自身的引用赋值给此属性，以供实现类同容器交互。

4.14.2.IObjectPostProcessor 接口

对象后处理器是指实现了 Spring.Objects.Factory.Config.IObjectPostProcessor 接口的类，已经在前文讨论过。在此很有必要重申一下，对象后处理器在 IApplicationContext 中的使用方法要比在 IObjectFactory 中的用法简单的多。在 IApplicationContext 中，任何实现了 IObjectPostProcessor 接口的对象都会被容器自动注册为对象后处理器，并在适当的时机调用。

4.14.3.IObjectFactoryPostProcessor 接口

工厂后处理器是指实现了 Spring.Objects.Factory.Config.IObjectFactoryPostProcessor 接口的类，已经在前文讨论过。在此很有必要重申一下，在 IApplicationContext 中使用工厂后处理器的方式比在 IObjectFactory 中的用法要简单的多。在 IApplicationContext 中，任何实现了 IObjectFactoryPostProcessor 接口的对都会被容器自动注册为工厂后处理器，并在适当的时机调用。

4.14.4.PropertyPlaceholderConfigurer 类

PropertyPlaceholderConfigurer 类在 IObjectFactory 中的用法已经在前文讨论过。同样，在 IApplicationContext 中使用该类的方法要简单的多，只要象普通对象一样布署在容器中，容器会自动识别并应用它，无需手动执行。

4.15.从其它文件中导入对象定义

在配置容器的时候，经常需要将对象定义存放在多个 XML 文件中。相应的，也可以在代码中载入多个文件来创建应用程序上下文，方法是将多个资源位置依次作为构造器的参数传入。如果使用对象工厂，可以重复使用同一个对象定义读取器将对象定义依次读入（按：XmlObjectFactory 的构造器没有可以接受多个资源参数的重载）。在下面的例子中，应用程序上下文分别从一个文件和一个程序集中读取配置信息。

```
IApplicationContext context = new
XmlApplicationContext( "file://objects.xml",
"assembly://MyAssembly/MyProject/objects-dal-layer.xml");
```

在用多文件配置容器时，推荐使用上面的方法。因为不同的配置文件之间不需要相互了解。另外，也可以在配置文件中使用一或多个 import 节点从其它文件中导入对象定义。注意所有的 import 元素都必须放在<objects>节点中第一个<object>定义之前，见下面的例子：

```
<objects xmlns="http://www.springframework.net">
  <import resource="services.xml"/>
  <import resource="resources/messageSource.xml"/>
  <import resource="/resources/themeSource.xml"/>
  <object id="object1" type="..."/>
  <object id="object2" type="..."/>
```

该例从外部导入了 3 个对象定义文件：services.xml，messageSource.xml 和 themeSource.xml。导入时使用的文件路径是当前文件的相对路径，也就是说 services.xml 文件必须和当前文件在同一目录下，而 themeSource.xml 和 messageSource.xml 必须在当前文件所在目录的 resources 子目录下。可以看到路径中第一个反斜线是可有可无的，不过既然用的是相对路径，不用反斜线可能会更好看一点。

被导入的文件必须完全满足 XML 对象定义的格式，必须以<objects>作为根节点。

4.16.服务定位器访问

应用程序中大部分代码都最好是以依赖注入（控制反转）的风格写就，代码独立于 IObjectFactory 或 IApplicationContext 容器之外，在创建时由容器进行依赖项注入，并且对容器毫无所知。然而，在某些情况下代码却需要以 singleton（或类似 singleton）的模式访问容器。例如，第三方代码可能会不通过容器而直接创建对象；又比如 WinForm 应用程序中，嵌套的用户控件是在 VS 自动生成的 InitializeComponent 方法内创建的。如果这些对象或控件需要使用容器中的对象，就可以使用服务定位器方式在代码内部从容器中请求。

Spring.Context.Support.ContextRegistry 类允许通过静态的定位器方法获取 IApplicationContext 引用。当 IApplicatoinContext 被前面讲过的 ContextHandler 创建时，ContextRegistry 同时也被初始化。可以用 ContextRegistry 的静态方法 GetContext()获取上下文的引用。另外，如果 IApplicationContext 是通过其它途径创建的，也可以在程序开始时用 ContextRegistry 的 RegisterContext 方法将其注册到 ContextRegistry 中。通过 GetContext(string name)方法也可获得子上下文的引用，例如：

```
IApplicationContex ctx = ContextRegistry.GetContext("mySubContext");
```

这样就可以获取下面配置中的子上下文：

```
<spring>
  <context>
    <resource
uri="assembly://MyAssembly/MyProject/root-objects.xml"/>
        <context name="mySubContext">
            <resource uri="file://objects.xml"/>
        </context>

  </context>
</spring>
```

ContextRegistry.Clear()方法会清空所有的上下文。在.NET 2.0 中，该方法也会调用 ConfigurationManager 的 RefreshSection 方法，以便在再次访问容器时从磁盘文件中重新读取 Spring.NET 的配置。注意在 Web 应用程序中 RefreshSection 方法不起作用，需要重新读取整个 Web.Config 文件。

---

[1]关于程序集的名称，可以参考.NET SDK 文档（随.NET SDK 一起安装），或者在 MSDN 网站上搜索 *AssemblyNames*。

[2] 参考 4.3.1,设置对象的属性和协作对象

[3]关于如何创建自定义类型转换器，可以在 MSDN 网站上搜索 *Implementing a Type Converter*。

第五章. IObjectWrapper 接口和类型转换

5.1. 简介

(Available in 1.0)

IObjectWrapper 是 Spring.NET 核心类库的一个底层接口，一般情况下开发人员不会直接去使用它，但由于本文档是一份参考文档，我们认为还是应该对此核心接口稍作阐述。开发人员可以用这个接口来实现数据绑定，因为数据绑定恰好是 IObjectWrapper 要解决的问题。

5.2. 使用 IObjectWrapper 接口管理对象

IObjectWrapper 接口和 ObjectWrapper 类是 Spring.Objects 命名空间下的重要类型。IObjectWrapper 接口的功能包括（单个或成批的）设置和读取属性的值、获取属性的描述信息（即 System.Reflection.PropertyInfo 类的实例）以及查询属性的可读/写性。IObjectWrapper 支持嵌套属性，可以为任意深度的属性赋值。该接口一般不会在应用程序代码中直接使用，而是由 Spring.NET 框架内部的类使用，比如 IObjectFactory 的各个实现类就用到了这个接口。

IObjectWrapper 的名字基本可以说明它的工作方式：将一个对象包装起来，再对其进行操作，比如读/写被包装对象的属性值等。

注意：如果不打算直接使用 IObjectWrapper 接口，那么本节的内容可以跳过不看。

5.2.1. 读、写普通及嵌套的属性

IObjectWrapper 接口中设置和读取属性值的方法分别是 SetPropertyValue()和 GetPropertyValue()，这两个方法都有很多重载，详细内容可参考 API 文档。

SetPropertyValue()和 GetPropertyValue()方法可以用属性路径来查找属性。属性路径是一个表达式，实现了 IObjectWrapper 的类可以根据该路径查找被包装对象的属性，下面是一些例子...

表 5.1. 属性路径的例子

路径	解释
name	表示被包装对象中名为 name 的属性。
account.name	表示被包装对象中 account 属性的 name 属性。
account[2]	表示被包装对象中 account 属性索引器的第 3 个元素。索引属性一般用于访问集合字段，也可以访问任何支持索引的类型。

我们来看一个使用 IObjectWrapper 读写对象属性的例子，先看下面两个类：

```
[C#]
public class Company
 {
 private string name;
 private Employee managingDirector;

 public string Name
 {
 get { return this.name; }
 set { this.name = value; }
 }

 public Employee ManagingDirector
 {
 get { return this.managingDirector; }
 set { this.managingDirector = value; }
 }
}
[C#]
public class Employee
{
 private string name;
 private float salary;

 public string Name
 {
 get { return this.name; }
 set { this.name = value; }
 }

 public float Salary
 {
 get { return salary; }
 set { this.salary = value; }
 }
}
```

在下面的代码中，用 IObjectWrapper 通过属性名来读写被包装的 Company 和 Employee 对象的属性：

```
[C#]
Company c = new Company();
IObjectWrapper owComp = new ObjectWrapper(c);
// setting the company name...
owComp.SetPropertyValue("name", "Salina Inc.");
// can also be done like this...
PropertyValue v = new PropertyValue("name", "Salina Inc.");
owComp.SetPropertyValue(v);

// ok, let's create the director and bind it to the company...
Employee don = new Employee();
IObjectWrapper owDon = new ObjectWrapper(don);
owDon.SetPropertyValue("name", "Don Fabrizio");
owComp.SetPropertyValue("managingDirector", don);

// retrieving the salary of the ManagingDirector through the company
float salary =
(float)owComp.GetPropertyValue("managingDirector.salary");
```

注意 Spring.NET 是遵循 CLS（Common Language Specification，通用语言规范）的，任何要解析为属性、事件或类型的字符串都是大小写无关的。所以在上面的例子中，虽然 C#是大小写敏感的，我们也可以用小写的"name"来设置 Employee 对象的 Name 属性。下面的例子（仍使用上面的类型）更能说明这个问题...

```
[C#]
// ok, let's create the director and bind it to the company...
Employee don = new Employee();
IObjectWrapper owDon = new ObjectWrapper(don);
owDon.SetPropertyValue("naMe", "Don Fabrizio");
owDon.GetPropertyValue("nAmE"); // gets "Don Fabrizio"

IObjectWrapper owComp = new ObjectWrapper(new Company());
owComp.SetPropertyValue("ManaGINGdirecToR", don);
owComp.SetPropertyValue("mANaGiNgdirector.salARY", 80000);
Console.WriteLine(don.Salary); // puts 80000
```

Spring.NET 的这种大小写无关的处理方式是遵循 CLS 规范的，不是什么功能缺陷。如果您正好有这样一个类型，其中部分属性、事件或者方法仅能以大小写来区分，那么您就需要考虑重构代码了，因为一般说来都会认为这不是好的编码风格。

5.2.2. 其它功能

除了上一小节讲到的功能，IObjectWrapper 还有一些其它功能，虽然不值得单独花一小节来讲，不过您可能会比较感兴趣：

- *检测属性的可读性和可写性*。可以用 IsReadable()和 IsWritable()方法检测某个属性是否可读或可写。
- *获取 PropertyInfo 实例*。使用 GetPropertyInfo(string)和 GetPropertyInfos()方法，可以获取 System.Reflection.PropertyInfo 类的实例，如果需要访问被包装对象属性的元数据，可以用这两个方法。

5.3. 类型转换

如果已经用标准的.NET 机制将一个类型转换器和某个类型关联了起来（看下面的代码），Spring.NET 就会使用这个类型转换器进行类型转换。

```
[C#]
[TypeConverter (typeof (FooTypeConverter))]
public class Foo
{
}
```

Spring.Core 中的很多类都用到了 System.ComponentModel 命名空间下的 TypeConverter 类，这些类“...可以用统一的方式来进行类型转换和普通值及子属性的访问。”。[4]

例如，我们可以用可读性较好的格式表示一个日期（比如 1984 年 8 月 30 日），也可以将它转换为原始的日期格式（或者）System.DateTime 类型。在.NET 中，通过为某个类型应用 TypeConverterAttribute 特性，可以为它指定类型转换器。Spring.NET 则提供了另外几种关联类型转换器的方式。Spring.NET 可以进行某些无法用标准方式完成的转换，比如，Spring.Core 程序集中有一个类型转换器，可以将以逗号分隔的字符串转换为字符串数组。用 Spring.NET 注册了自定义类型转换器之后，IObjectWrapper 接口就知道如何将某个值转换为指定类型了。

我们在 IoC 容器中配置对象的属性时，Spring.NET 就在后台应用了 TypeConverter。当用字符串值定义某个对象的属性(比如通过 XML 定义来声明)，且这个属性的类型是 System.Type 时，Spring.NET 会尝试用 RuntimeTypeConverter 将字符串值转换为 Type 对象。下面这个例子就将字符串“Example.Xml.SAXParser”自动转换为了相应的 Type 对象：

```
<objects xmlns="http://www.springframework.net">
<object id="parserFactory" type="Example.XmlParserFactory,
ExamplesLibrary"
```

```
destroy-method="Close">
 <property name="ParserClass" value="Example.Xml.SAXParser,
ExamplesLibrary"/>
</object>
</objects>
[C#]
public class XmlParserFactory
{
        private Type parserClass;

        public Type ParserClass
        {
        get { return this.parserClass; }
        set { this.parserClass = value; }
        }

        public XmlParser GetParser ()
        {
        return Activator.CreateInstance (ParserClass);
        }
}
```

5.3.1. 转换枚举类型

枚举类型的默认转换器是 System.ComponentModel.EnumConverter，它会将枚举值的名字转换为对应的值。比如说，TestObject 类有个 FileMode 枚举类型的属性，该枚举有个名为 Create 的值，那么这个类的对象就可以这么配置：

```
<object id="rod" type="Spring.Objects.TestObject, Spring.Core.Tests">
 <property name="Name" value="Rod"/>
 <property name="FileMode" value="Create"/>
</object>
```

5.4. 内置类型转换器

Spring.NET 预先注册了一部分自定义类型转换器（例如将以字符串表示的类型名转换为 System.Type 对象的转换器）。这些转换器都定义在 Spring.Core 程序集的 Spring.Objects.TypeConverters 命名空间下，如下表所示：

表 5.2. 内置类型转换器

类型	解释

类型	解释
RuntimeTypeConverter	将字符串表示的类型名转换为实际的 System.Type 对象
FileInfoConverter	将字符串转换为 System.IO.FileInfo 对象
StringArrayConverter	将以逗号分隔的字符串列表转换为字符串数组，反之亦可
UriConverter	将字符串表示 URI 转换为实际的 URI 对象
StreamConverter	将 Spring 的 IResource URI 的字符串表示形式转换为 InputStream 对象
ResourceConverter	将用字符串表示的 Spring IResource 格式的 URI 转换为 IResource 对象
ResourceManagerConverter	将一个以逗号分隔的两段字符串（资源名，程序集名）转换为一个 System.Resource.ResourceManager 对象
RGBColorConverter	将以逗号分隔的红，绿，蓝值转换为 System.Drawing.Color 结构

Spring.NET 会使用标准的.NET 机制来解析对象的类型，这些机制包括但不限于：检查和应用程序相关的所有配置文件、检查 GAC 以及使用程序集探测机制等。

---

[4] 原文为“... provides a unified way of converting types of values to other types, as well as for accessing standard values and subproperties.”。有关自定义 TypeConverter 的详细内容可以参见 MSDN 网站，相关内容可以搜索 *Implementing a Type Converter*。

第六章. IResource 接口

6.1. 简介

IResource 接口位于 Spring.Core.IO 命名空间中，可以用统一的方式描述和访问不同类型的资源。通过该接口，可以用类似多态的、与协议无关的方式处理文件和 URL 资源——.NET 的 FCL 中没有具备这些功能的接口。IResource 接口继承自 IInputStream 接口，后者只有一个属性：Stream InputStream，IResource 又其基础上添加了一系列属性用来描述资源的信息。

6.2. IResource 接口

IResource 接口的定义如下所示：

```
public interface IResource : IInputStreamSource
{
 bool IsOpen { get; }

 Uri Uri { get; }

 FileInfo File { get; }

 string Description { get; }

 bool Exists { get; }

 IResource CreateRelative(string relativePath);
}
```

表 6.1. IResource 接口的属性

属性	解释
InputStream	继承自 IInputStream 接口。打开并返回一个 System.IO.Stream。每次读取都会返回新的 Stream，调用者必须负责 Stream 的关闭工作。
Exists	检查资源是否存在，如果不存在就返回 false。
IsOpen	返回一个布尔值，以表明该资源是否一个已经打开的流。如果返回 true，则 InputStream 属性就不能多次读取，只能是读取一次然后关闭，以防资源泄漏。IResource 接口的实现类，除了 InputStreamResource，其它都应该将该属性返回 false。
Description	返回对资源的描述，比如说完整的文件名或实际的 URL。
Uri	资源的 Uri 表示形式。
File	如果资源可以被解析为一个绝对的文件路径，就返回一个 System.IO.FileInfo。

IResource 接口只有一个方法：

表 6.2. IResource 接口的方法

方法	解释
IResource CreateRelative(string relativePath)	使用类似于相对路径的字符串(./ 和 ../)创建与当前资源相关的资源

实现类可以用该方法获取一个代表实际 URL 或 File 的 IResource 对象，但要保证能够处理这些类型的资源。

在 Spring.NET 的内部大量使用了 IResource 接口，一般都是作为参数传递给需要使用 IResource 的方法。还有一些方法（比如 IApplicationContext 实现类的构造器），可以接受字符串类型的参数，并用该参数的值在内部创建一个 IResource 的实例。

实际上，在开发中我们也可以把 IResource 接口作为一个非常有用的工具类型，应用程序代码可以只使用 IResource 来访问资源，不需要引用 Spring.NET 的其它部分。虽然这样作会使代码依赖 Spring.NET，但也仅仅是依赖几个小小的工具类而已，不防就把它当成是一个用来解决小问题的第三方类库。

6.3. 内置的 IResource 实现类

Spring.NET 内置的 IResource 实现类包括：

- AssemblyResource：用于访问内嵌在.NET 程序集中的资源。相应的 Uri 格式为：assembly://<AssemblyName>/<NameSpace>/<ResourceName>。
- ConfigSectionResource：用于处理.NET 应用程序配置文件（即 App.Config）中与 Spring.NET 有关的配置数据，Uri 格式为：config://<path to section>
- FileSystemResource：访问文件系统资源，Uri 格式为：file://<filename>
- InputStreamResource：用于包装一个原始的 System.IO.Stream 对象。不支持 Uri 格式。
- UriResource：通过诸如 http 等标准的协议访问资源。Uri 格式即对应的标准协议格式。

相关内容可参考 MSDN 文档 supported Uri scheme types。

6.4. IResourceLoader 接口

要从指定的 Uri 中载入资源，需要使用相应的 IResourceLoader 实现类。默认情况下 Spring.NET 使用的是 ConfigurableResourceLoader。通常不需要直接访问该类，因为 IAppicationContext 自己也扩展了 IResourceLoader 接口。IResourceLoader 接口只有一个方法：IResource GetResource(string loaction)。IApplicationContext 将该方法委托给 ConfigurableResourceLoader 类代理，可以支持上面列出的各种 Uri 格式。如果未在 Uri 中指定协议类型则使用文件协议。请看下面的例子：

```
IResource resource =
appContext.GetResource("http://www.springframework.net/license.html");
resource =
appContext.GetResource("assembly://Spring.Core.Tests/Spring/TestResou
rce.txt");
resource = appContext.GetResource("https://sourceforge.net/");
resource = appContext.GetResource("file:///C:/WINDOWS/ODBC.INI");

StreamReader reader = new StreamReader(resource.InputStream);
Console.WriteLine(reader.ReadToEnd());
```

通过创建新的 IResource 实现类可以注册新的协议，但要保证新类的构造器能够正确解析 Uri 字符串。在 Spring.Web 命名空间中，有一个使用 Server.MapPath 来解析资源文件名的例子，可以参考一下。

CreateRelative 方法允许使用相对路径载入资源。对于相对的程序集资源（按：relative assembly resources）来说，使用相对路径可以根据程序集的命名空间来载入资源，例如：

```
IResource res = new
AssemblyResource("assembly://Spring.Core.Tests/Spring/TestResource.tx
t");
IResource res2 = res.CreateRelative("./Core.IO/TestIOResource.txt");
```

这段代码首先载入资源文件 TestResource.txt，然后转到 Spring.Core.IO 命名空间下，载入其中的资源文件 TestIOResource.txt。

6.5. IResourceLoaderAware 接口

IResourceLoaderAware 是一个特殊的接口，实现该接口的对象可以获取一个 IResourceLoader 引用。

```
public interface IResourceLoaderAware
{
 IResourceLoader ResourceLoader
 {
 set;
 get;
 }
}
```

应用程序上下文会自动判断布署在其中的对象是否实现了 IResourceLoaderAware 接口。如果对象实现了该接口，应用程序上下文会将自

己的引用传递给该属性（别忘了应用程序上下文也实现了 IResourceLoader 接口）。

当然了，既然 IApplicationContext 扩展了 IResourceLoader 接口，我们就完全可以用 IApplicationcontextAware 接口先获取应用程序上下文，再去访问资源。不过一般来说，如果我们只需要 IResourceLoader 的功能，还是单独使用 IResourceLoaderAware 接口比较好。代码只需要与 IResourceLoader 接口耦合，无需理会整个应用程序上下文。

6.6. 应用程序上下文和 IResource 路径

某些 IApplicationContext 实现类的构造器可以用字符串或字符串数组类型的参数来表示 XML 文件路径。例如，下面的代码使用两个 XML 文件创建了 XMLApplicationContext：

```
IApplicationContext context = new XmlApplicationContext(
        "file://objects.xml",
"assembly://MyAssembly/MyProject/objects-dal-layer.xml");
```

第七章. 多线程和并发操作

7.1. 简介

Spring.Threading 命名空间的目的，是用一系列功能强大的抽象来增强 FCL 中的相关功能。由于 Doug Lea 在他的 java 版 EDU.oswefo.cs.dl.util.concurrent 类库中已经定义了很多用于并发操作的成熟类型，所以我们决定将其中的一部分导入到 Spring.NET 中来。到目前为止我们只导入了 3 个类，包括用来支持 AOP 池化方面（AOP based pooling aspect）、提供对象池基本功能的类和 Semaphore 类（.NET 1.0/1.1 没有相应的类型，2.0 中新增了 Semaphore 类）。

另外，在 Java 5 中，java.util.concurrent 包中的类也建立在 Doug Lea 的类库之上，并且已经通过了严格的审查。Java 5 类库及.NET 2.0 中新增的类型有可能导致 Spring 在这方面进行修改。

Spring.Threading 中还有一个很重要的工具类 LogicalThreadContext，用于进行线程的本地存储。

7.2. 线程本地存储

根据运行环境的不同，在线程本地存储中保存对象也需要应用不同的策略。在 Web 应用程序中，如果一个请求可能会被多个线程处理，线程局部对象就应该保存在 HttpContext.Current 中。其它情况则可使用 System.Runtime.Remoting.Messaging.CallContext。这两种策略的选择依据，以及与[ThreadStatic]特性的比较可以参考“Piers7”的 blog 和 Spring.NET 的论坛。而 LogicalThreadContext 则把这方面的细节隐藏了起来，所以可以使代码更具可移植性。

LogicalThreadContext 类的定义相当简单：

```
public sealed class LogicalThreadContext
{
 public static object GetData(string name)

 public static void SetData(string name, object value)

 public static void FreeNamedDataSlot(string name)

}
```

该类定义了一系列静态方法，可以用字符串作为键值设置或读取数据。如果想要释放存储区的空间，可以调用 FreeNamedDataSlot 方法。

7.3. 同步基础

用户可能会很奇怪，既然 System.Threading 已经提供了那么多用于同步的类，为什么 Spring.NET 还要定义这些同步类型呢？这是因为 System.Threading 中的相关类型虽多，却没有为我们提供一个优雅的抽象或接口，我们只能在一个很低的层面编写代码。凭经验来讲，要让代码工作的好，我们最终还是得拿出一个抽象层次来。Doug Lea 已经在这方面作了很多研究，我们可以充分的利用他的研究成果。

7.3.1. ISync

ISync 是所有用于控制多线程资源访问的类的核心接口。这个接口很简单，主要有两种基本的用途。第一种是用来阻塞当前线程，直到满足了某个条件：

```
void ConcurrentRun(ISync lock) {
 lock.Acquire(); // block until condition met
 try {
```

```
 // ... access shared resources
 }
 finally {
 lock.Release();
 }
}
```

另一种是用来指定在满足某个条件前将线程阻塞的最多次数：

```
void ImpatientConcurrentRun(ISync lock) {
 // block for at most 10 milliseconds for condition
 if ( lock.Attempt(10) ) {
 try {
 // ... access shared resources
 }
 finally {
 lock.Release();
 }
 } else {
 // complain of time out
 }
}
```

7.3.2. SyncHolder

SyncHolder 类实现了 System.IDisposable 接口，利用这个类我们可以在 c#的 using 语句中使用 ISync 接口：进入 using 语句后 ISync 的 Acquire 方法会自动调用，退出 using 语句后 ISync 的 Release 方法会被自动调用。

使用 using 语句可以简化编程模型，请看下面的例子：

```
ISync sync = ...
...
using (new SyncHolder(sync))
 {
 // ... code to be executed
 // holding the ISync lock
 }
```

当然这是同步版本（按：timed version），如果要自己处理超时，就稍微麻烦一点儿：

```
ISync sync = ...
long msecs = 100;
...
// try to acquire the ISync for msecs milliseconds
try
{
 using (new SyncHolder(sync, msecs))
 {
 // ... code to be executed
 // holding the ISync lock
 }
}
catch (TimeoutException)
{
 // deal with failed lock acquisition
}
```

7.3.3. Latch

Latch 类实现了 ISync 接口，它是一个*闭锁*。闭锁是一个布尔条件，最多只能设置一次。一旦闭锁开放，所有等待着的线程都会解除阻塞。这就如同一个初始状态设置为非终止的 ManualResetEvent 对象（即创建时传给构造器的参数值为 false，或者调用 Reset 将事件状态设置为非终止状态，从而导致其它线程阻塞）一样，只有在调用了 Set()方法将事件状态设为终止之后，才能允许其它等待着的线程继续执行。Latch 典型的用法是作为一组工作线程的开始信号。

```
class Boss {
 Latch _startPermit;

 void Worker() {
 // very slow worker initialization ...
 // ... attach to messaging system
 // ... connect to database
 _startPermit.Acquire();
 // ... use resources initialized in Mush
 // ... do real work
 }

 void Mush() {
 _startPermit = new Latch();
 for (int i=0; i<10; ++i) {
```

```
 new Thread(new ThreadStart(Worker)).Start();
 }
 // very slow main initialization ...
 // ... parse configuration
 // ... initialize other resources used by workers
 _startPermit.Release();
 }

}
```

7.3.4.信号量

Semaphore 类实现了 ISync 接口，实现了信号量的功能。从概念上说，信号量维护了一组许可证，用来控制允许同时进入同步代码的线程数。每次 Acquire 方法调用都需要等待有一个许可证变为可用，然后便占用此许可证。每次调用 Release 方法后都会释放一个许可证。其实 Semaphore 并没有使用实际的“许可证”对象，只是在内部维护了一个许可证的可用数量。最典型的用法是控制对共享对象池的访问。

```
class LimitedConcurrentUploader {
 // ensure we don't exceed maxUpload simultaneous uploads
 Semaphore _available;
 public LimitedConcurrentUploader(maxUploads) {
 _available = new Semaphore(maxUploads);
 }
 // no matter how many threads call this method no more
 // than maxUploads concurrent uploads will occur.
 public Upload(IDataTransfer upload) {
 _available.Acquire();
 try {
 upload.TransferData();
 }
 finally {
 _available.Release();
 }
 }
}
```

第八章. 对象池

8.1. 简介

(Available in 1.0)

Spring.Pool 命名空间中包含一组用于实现对象池的通用类型。对象池是一项非常著名的技术，可用来最小化大对象的创建工作。最常见的例子就是数据库连接池，数据库连接池允许为客户端的每次请求重用已有的连接对象。线程池也是一个常用的对象池，用于提高应用程序对多个并发请求的响应性能。

.NET 本身支持常用的对象池。在 ADO.NET 中，通过 Data Provider 的配置可以使用数据库连接池。System.ThreadPool 则用来实现线程池。存储其它对象的对象池可以通过用 System.EnterpriseServices 命名空间下的托管 API 操作 COM+ 来实现。

但是，在很多场合我们都需要使用其它的对象池机制。这有可能是因为.NET 本身的机制（比如 System.ThreadPool）不满足我们的要求（参考 Ami Bar：Smart Thread Pool 中关于 ThreadPool 高级用法的讨论）。也可能是因为我们想将没有继承 System.EnterpriseServices.ServicedComponent 的对象放入对象池。Spring.NET 使用 AOP 代理和通用的池化 API 实现对象池，与.NET 本身的对象池不同，它不对对象模型有任何的限制，可以支持任意对象。

注意，如果您只关心如何对一个现有对象应用对象池，那么也可以不看本章的内容，使用 Spring.Aop.Target.SimplePoolTargetSource 更为合适。甚至通过编程或配置使用 IoC 容器也能做到这一点。.NET 框架也会在未来版本中通过特性来应用对象池，就如同企业服务组件中的用法一样。

第二十五章,IoC 快速入门中有一个例子，专门讲解如何在不使用 AOP 的情况下单独使用池 API。

8.2. 接口和实现

Spring.Pool 命名空间有两个简单的接口，用来管理对象池。第一个是 IObjectPool，负责从池中取出、放回对象。第二个是 IPoolableObjectFactory，负责为 IObjectPool 管理的对象定义生命周期方法（按：参考 API 文档）。这些接口都是基于 Jakarta Commons Pool 组件的 API 创建的。
Spring.Pool.Support.SimplePool 是 IObjectObject 接口的默认实现类，Spring.Aop.Target.SimplePoolTargetSource 则是 IPoolableObjectFactory 接口的实现类，用于 AOP 框架中。目前 Spring.Pool 命名空间不是作为 Jakarta Commons Pool API 的替代品，而是为普通的 AOP 应用提供基础的对象池功能。因此，Spring.NET 没有实现 Jakarta 包里的其它接口和基类。

第九章. Spring.NET 杂记

9.1. 简介

本章内容不属于任何范畴，算是给读者的一些提示和建议。

9.2. PathMatcher

Spring.Util.PathMatcher 用于匹配 Ant/NAnt 风格的路径名：

PathMatcher 的静态方法 Match 可以用来进行匹配，如下：

```
public static bool Match(string pattern, string path)
```

如果不考虑大小写，需要用下面的重载：

```
public static bool Match(string pattern, string path, bool ignoreCase)
```

9.2.1.通用规则

可以使用*，?和**创建模式字符串：

- *：匹配任意数量的非斜线字符。
- ?：匹配一个非斜线、非小数点的任意字符。
- **：匹配任意子目录，不考虑子目录的深度。

9.2.2.匹配文件名

下面的字符串可用于匹配文件名：

```
foo?bar.*
```

该字符串可以匹配下面所有的值：

```
fooAbar.txt
foo1bar.txt
foo_bar.txt
foo-bar.txt
```

但不能匹配下面的值：

```
foo.bar.txt
foo/bar.txt
foo\bar.txt
```

下面是经典的通配符：

.

可以匹配以下值：

foo.db
.db
foo
foo.bar.db
foo.db.db
db.db.db

但不匹配以下值：:

c:/
c:/foo.db
c:/foo
c:/.db
c:/foo.foo.db
//server/foo

9.2.3.匹配子目录

下面的字符串可用于匹配任意深度的子目录：

/db/

这个字符串可以匹配下面所有的值：

/db
//server/db
c:/db
c:/spring/app/db/foo.db
//Program Files/App/spaced dir/db/foo.db
/home/spring/spaced dir/db/v1/foo.db

但不能匹配以下值：

c:/spring/app/db-v1/foo.db
/home/spring/spaced dir/db-v1/foo.db

可以用下面的组合来表示子目录的通配符：

/bin//tmp/**

可以匹配以下值：

c:/spring/foo/bin/bar/tmp/a
c:/spring/foo/bin/tmp/a/b.c

但不匹配以下值

c:/spring/foo/bin/bar/temp/a
c:/tmp/foo/bin/bar/a/b.c

也可以用更为复杂的匹配模式：

/.spring-assemblies*/

可以匹配：

c:/.spring-assemblies
c:/.spring-assembliesabcd73xs
c:/app/.spring-assembliesabcd73xs
c:/app/.spring-assembliesabcd73xs/foo.dll
//server/app/.spring-assembliesabcd73xs

但不能匹配：

c:/app/.spring-assemblie

9.2.4. 大小写需要考虑，斜线可以任意

.NET 是一个跨平台的开发环境。所以，PathMatcher 类在匹配时会考虑模式字符串和路径中的大小写，比如说：

/db//*.DB

可以匹配：

c:/spring/service/deploy/app/db/foo.DB

但不匹配：

c:/spring/service/deploy/app/DB/foo.DB
c:spring/service/deploy/app/spaced dir/DB/foo.DB
//server/share/service/deploy/app/DB/backup/foo.db

如果需要用大小写无关的方式匹配，应该显式的对 PathMatcher 进行相应的设置。（按：使用 PathMatcher 静态方法 Match 的一个重载，见 API 文档）

但是，斜线和反斜线的含义是一样的：

spring/foo.bar

可以匹配下面所有的值：

c:\spring\foo.bar
c:/spring\foo.bar
c:/spring/foo.bar
/spring/foo.bar
\spring\foo.bar

第十章. 表达式求值

10.1. 简介

Spring.Expressions 命名空间可以用一种强大的表达式语言在运行时操作对象。这种语言可以读写属性值、调用方法、访问数组/集合/索引器的元素、进行算术和逻辑运算，同时支持命名变量，并且能够通过名称从 IoC 容器获取对象。

在 Spring.NET 中，该命名空间是其它许多功能（比如 IoC 容器中的增强属性求值、数据验证框架及 ASP.NET 的数据绑定框架）的基础。如果需要以对象的运行时状态为依据进行求值，您还会发现一些更加有趣的特性。对于有 Java 背景的开发人员来说，Spring.Expressions 命名空间的功能和 Java 的 OGNL（Object Graph Navigation Language）很相似。

本章将用一个 Inventor 类及与其相关的类型为例来讲解表达式语言。这些类的代码和使用到的数据都列举在本章的最后一节 10.4,本章所用的示例类型中。这些类型是从 Spring.Expression 命名空间的 NUnit 测试项目中拿出来的，您可以参考该项目的源码来学习表达式的其它用法。

10.2. 表达式求值

Spring.Expressions 命名空间的核心类是 ExpressionEvaluator，其主要方法如下：

```
public static object GetValue(object root, string expression);

public static object GetValue(object root, string expression,
IDictionary variables)
```

```
public static void SetValue(object root, string expression, object
newValue)

public static void SetValue(object root, string expression, IDictionary
variables, object newValue)
```

其中,参数 root 就是我们要在其上进行求值的目标“根”对象,参数 expression 是用于求值的字符串表达式。其余参数用来指定在表达式中使用的变量,稍后再讨论它们。下面的代码从 Inventor 实例中读取属性值。Invertor 类的代码可以参见 10.4,本章所用的示范类型。

```
Inventor tesla = new Inventor("Nikola Tesla", new DateTime(1856, 7, 9),
"Serbian");

tesla.PlaceOfBirth.City = "Smiljan";

string evaluatedName = (string) ExpressionEvaluator.GetValue(tesla,
"Name");

string evaluatedCity = (string) ExpressionEvaluator.GetValue(tesla,
"PlaceOfBirth.City"));
```

上面的代码执行后，变量 evalutedName 的值为"Nikola Tesla"，evalutedCity 的值为"Smijan"。在表示嵌套属性时要用小数点分隔路径。设置属性值与读取属性值的用法类似，如果我们要改写历史，将 Tesla 的出生地改为其它城市，可以用下面的代码：

```
ExpressionEvaluator.SetValue(tesla, "PlaceOfBirth.City", "Novi Sad");
```

Spring.Expressions 命名空间中另有一个不太常用到的类：Expression。如果要用某个表达式针对同一对象进行多次求值，该类可以提高性能。
ExpressionEvaluator 类会在每次调用 GetValue 或 SetValue 时解析表达式；而 Expression 则会将解析和反射的结果缓存起来。该类的部分方法如下（按：方法很多，参见 API 文档）：

```
public static IExpression Parse(string expression)

public override object Get(object context, IDictionary variables)

public override void Set(object context, IDictionary variables, object
newValue)
```

可以用下面的代码读取前面例子的属性值：

```
IExpression exp = Expression.Parse("Name");
```

```
string evaluatedName = (string) exp.GetValue(tesla, null);
```

在使用 ExpressionEvaluator 的时候，会涉及到几个异常类型。如果引用的属性不存在，会抛出 InvalidPropertyException；如果在遍历嵌套属性的路径时遇到了 null 值，会抛出 NullValueInNextedPathException（按：对于嵌套的属性，除了最末一个点后的实际属性，路径当中的值都必须不能为空）；如果传值时发生其它错误，会抛出 ArgumentException 和 NotSupportedException 异常。

表达式语言有自己的语法，并且使用 ANTLR 来构建语法分析器和解析器，语法错误在解析时就会被捕获。如果想深入了解解析器的实现细节，可以参考源文件中的语法文件 Expression.g。另外，目前 Spring.NET 使用的 ANTLR.DLL 程序集是用 Spring.NET 的 key 标记的。将来 ANTLR 会在新版本中使用自己的强命名程序集。

10.3. 语言参考

10.3.1.文字表达式

文字表达式可以表示字符串、日期、数字值（int、real 和 hex）、布尔及空值。字符串以单引号包围。表达式内的单引号要用反斜线进行转义。下面是文字表达式的一些简单用法。注意文字表达式一般不会这样单独使用，多是作为复杂表达式的一部分，比如作为逻辑比较操作符的一个操作数。

```
string helloWorld = (string) ExpressionEvaluator.GetValue(null, "'Hello World'"); // evals to "Hello World"

string tonyPizza  = (string) ExpressionEvaluator.GetValue(null, "'Tony\\'s Pizza'"); // evals to "Tony's Pizza"

double avogadrosNumber = (double) ExpressionEvaluator.GetValue(null, "6.0221415E+23");

int maxValue = (int)  ExpressionEvaluator.GetValue(null, "0x7FFFFFFF"); // evals to 2147483647

DateTime birthday = (DateTime) ExpressionEvaluator.GetValue(null, "date('1974/08/24')");

DateTime exactBirthday =
  (DateTime) ExpressionEvaluator.GetValue(null, "
date('19740824T131030', 'yyyyMMddTHHmmss')");
```

```
bool trueValue = (bool) ExpressionEvaluator.GetValue(null, "true");

object nullValue = ExpressionEvaluator.GetValue(null, "null");
```

注意‘Tony\\’s Pizza’里多出来的那个反斜线是为了遵循 C#的语法（按：在字符串中\\转义为\）。数字中可以出现负号、指数幂和小数点。默认情况下，实数由 Double.Parse 来解析，除非指定了格式字符“M”或“F”。对于包含格式字符“M”的数字，将由 Decimal.Parse 解析；包含“F”的数字则由 Single.Parse 解析。如果在日期文本中指定了两个参数（如上面例子倒数第三句），就会用 DateTime.ParseExact 进行解析。注意在解析时所有类型的 Parse 方法都会在内部引用 CultureInfo.InvariantCulture 作为当前的语言文化信息。

10.3.2.属性，数组，列表，字典，索引器

前面 10.2,表达式求值一节中曾经讲到，用属性路径表示嵌套的属性是很简单的，只需要用小数点分隔嵌套的属性名即可。例子中 Invertor 的两个实例：pupin 和 tesla，已经用 10.4,本章用到的示例类型中定义的数据赋过值了。如果要继续“向下”访问 Tesla 生日中的年份和 Pupin 出生地中的城市信息，可以用下面的代码：

```
int year = (int) ExpressionEvaluator.GetValue(tesla, "DOB.Year"));  // 1856

string city = (string) ExpressionEvaluator.GetValue(pupin, "PlaCeOfBirTh.CiTy");  // "Idvor"
```

眼尖的人可能已经发现了，pupin 的出生地大小写交替，显得乱七八糟。这并非排版错误，而是有意为之，为的是证明表达式语言是忽略大小写的。

在表达式中，可以用方括号获取数组和列表的元素：

```
// Inventions Array
string invention = (string) ExpressionEvaluator.GetValue(tesla, "Inventions[3]"); // "Induction motor"

// Members List
string name = (string) ExpressionEvaluator.GetValue(ieee, "Members[0].Name"); // "Nikola Tesla"

// List and Array navigation
string invention = (string) ExpressionEvaluator.GetValue(ieee, "Members[0].Inventions[6]") // "Wireless communication"
```

在访问字典的元素时，键值要用单引号括起来。在本例中，因为字典 Officers 的键值是字符串类型，我们可以直接使用文本字符串：

```
// Officer's Dictionary
Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(ieee,
"Officers['president']";

string city = (string) ExpressionEvaluator.GetValue(ieee,
"Officers['president'].PlaceOfBirth.City"); // "Idvor"

ExpressionEvaluator.SetValue(ieee,
"Officers['advisors'][0].PlaceOfBirth.Country", "Croatia");
```

也可以在方括号中直接使用其它表达式，例如变量名或某个类型的静态属性/方法名。我们会在其它小节中讨论这些内容。

同样的，索引器也用方括号访问。下面是一个例子。也可以访问多维索引器。

```
public class Bar
{
    private int[] numbers = new int[] {1, 2, 3};

    public int this[int index]
    {
        get { return numbers[index];}
        set { numbers[index] = value; }
    }
}

Bar b = new Bar();

int val = (int) ExpressionEvaluator.GetValue(bar, "[1]") // evaluated to
2

ExpressionEvaluator.SetValue(bar, "[1]", 3);  // set value to 3
```

10.3.2.1. 定义内联的数组、列表和字典

除了能在表达式中引用容器中的数组、列表和词典外，Spring.NET 还允许在表达式中定义内联数组、列表和词典。内联的列表用花括号定义，在花括号内，用逗号来分隔列表的元素。

```
{1, 2, 3, 4, 5}
{'abc', 'xyz'}
```

如果需要初始化一个强类型的数组，而非弱类型的列表，可以使用数组的初始化操作符：

```
new int[] {1, 2, 3, 4, 5}
new string[] {'abc', 'xyz'}
```

内联字典的定义表达式有点复杂：首先需要在花括号前用#作为前缀，在花括号内，用逗号将键值对分隔开，键和值之间则用冒号隔开：

```
#{'key1' : 'Value 1', 'today' : DateTime.Today}
#{1 : 'January', 2 : 'February', 3 : 'March', ...}
```

用这种方式创建的数组、列表和字典可以在任意表达式中使用，稍后我们会在例子中看到。

请注意，虽然前面都是用简单的文字值来定义数组/列表的项和词典的键、值，但这只是一个简单的例子——实际上我们可以用任何有效的表达式来定义元素。

10.3.3.方法

在文字表达式中，可以用标准的 c#语法调用方法：

```
//string literal
char[] chars = (char[]) ExpressionEvaluator.GetValue(null,
"'test'.ToCharArray(1, 2)"))  // 't','e'

//date literal
int year = (int) ExpressionEvaluator.GetValue(null,
"date('1974/08/24').AddYears(31).Year") // 2005

// object usage, calculate age of tesla navigating from the IEEE society.

ExpressionEvaluator.GetValue(ieee,
"Members[0].GetAge(date('2005-01-01'))") // 149 (eww..a big anniversary
is coming up ;)
```

10.3.4.操作符

10.3.4.1.关系操作符

关系操作符：相等、不等、小于、小于等于、大于、大于等于在文字表达式里都用普通的符号表示。如果被比较的对象实现了 IComparable 接口，这些操作符就

能起作用。另外，文字表达式也支持枚举类型的关系操作，但如果要使用的枚举不是 mscorlib 中的类型，则需要进行注册，可参见 10.3.8,类型注册。

```
ExpressionEvaluator.GetValue(null, "2 == 2")  // true

ExpressionEvaluator.GetValue(null, "date('1974-08-24') !=
DateTime.Today")  // true

ExpressionEvaluator.GetValue(null, "2 < -5.0") // false

ExpressionEvaluator.GetValue(null, "DateTime.Today <=
date('1974-08-24')") // false

ExpressionEvaluator.GetValue(null, "'Test' >= 'test'") // true
```

下面是枚举类型的比较：

```
FooColor fColor = new FooColor();

ExpressionEvaluator.SetValue(fColor, "Color", KnownColor.Blue);

bool trueValue = (bool) ExpressionEvaluator.GetValue(fColor, "Color ==
KnownColor.Blue"); //true
```

其中 FooColor 的定义为：

```
public class FooColor
{
    private KnownColor knownColor;

    public KnownColor Color
    {
        get { return knownColor;}
        set { knownColor = value; }
    }
}
```

除了标准的关系操作符，Spring.NET 的表达式语言还支持一些功能很强大的特殊关系操作符，这些操作符是从 SQL 中“借”来的，比如 in、like 和 between，以及 is 和 matches 等等，使用这些操作符可以判断某个对象是否是特定的类型，或者判断某个值是否和一个正则表达式相匹配。

```
ExpressionEvaluator.GetValue(null, "3 in {1, 2, 3, 4, 5}")  // true

ExpressionEvaluator.GetValue(null, "'Abc' like '[A-Z]b*'")  // true
```

```
ExpressionEvaluator.GetValue(null, "'Abc' like '?'")  // false

ExpressionEvaluator.GetValue(null, "1 between {1, 5}")  // true

ExpressionEvaluator.GetValue(null, "'efg' between {'abc', 'xyz'}")  //
true

ExpressionEvaluator.GetValue(null, "'xyz' is int")  // false

ExpressionEvaluator.GetValue(null, "{1, 2, 3, 4, 5} is IList")  // true

ExpressionEvaluator.GetValue(null, "'5.0067' matches
'^-?\\d+(\\.\\d{2})?$'"))  // false

ExpressionEvaluator.GetValue(null, @"'5.00' matches
'^-?\d+(\.\d{2})?$'")  // true
```

注意 like 操作符的匹配字符串使用的是 VB 的语法，而非 SQL 语法。（*注意：在 1.1 的最终版中可能改变该行为，所以如果使用了 like 操作符，可能将来需要做必要的修改。*）

10.3.4.2.逻辑操作符

逻辑操作符包括 and，or 和 not，用法如下所示：

```
// AND
bool falseValue = (bool) ExpressionEvaluator.GetValue(null, "true and
false"); //false

string expression = @"IsMember('Nikola Tesla') and IsMember('Mihajlo
Pupin')";
bool trueValue = (bool) ExpressionEvaluator.GetValue(ieee, expression);
//true

// OR
bool trueValue = (bool) ExpressionEvaluator.GetValue(null, "true or
false");  //true

string expression = @"IsMember('Nikola Tesla') or IsMember('Albert
Einstien')";
bool trueValue = (bool) ExpressionEvaluator.GetValue(ieee, expression);
// true
```

```
// NOT
bool falseValue = (bool) ExpressionEvaluator.GetValue(null, "!true");

// AND and NOT
string expression = @"IsMember('Nikola Tesla') and !IsMember('Mihajlo
Pupin')";
bool falseValue = (bool) ExpressionEvaluator.GetValue(ieee,
expression);
```

10.3.4.3.算术运算符

算术操作符可用于数字、字符串和日期的操作。其中数字和日期可以做减法。乘法和除法则只适用于数字。同时，也可以支持其它类型的数学操作符，如求模(%)和指数幂(^)。见下例：

```
// Addition
int two = (int)ExpressionEvaluator.GetValue(null, "1 + 1"); // 2

String testString = (String)ExpressionEvaluator.GetValue(null, "'test'
+ ' ' + 'string'"); //'test string'

DateTime dt = (DateTime)ExpressionEvaluator.GetValue(null,
"date('1974-08-24') + 5"); // 8/29/1974

// Subtraction

int four = (int) ExpressionEvaluator.GetValue(null, "1 - -3"); //4

Decimal dec = (Decimal) ExpressionEvaluator.GetValue(null, "1000.00m -
1e4"); // 9000.00

TimeSpan ts = (TimeSpan) ExpressionEvaluator.GetValue(null,
"date('2004-08-14') - date('1974-08-24')"); //10948.00:00:00

// Multiplication

int six = (int) ExpressionEvaluator.GetValue(null, "-2 * -3"); // 6

int twentyFour = (int) ExpressionEvaluator.GetValue(null, "2.0 * 3e0 *
4"); // 24

// Division
```

```
int minusTwo = (int) ExpressionEvaluator.GetValue(null, "6 / -3"); // -2

int one = (int) ExpressionEvaluator.GetValue(null, "8.0 / 4e0 / 2"); //
1

// Modulus

int three = (int) ExpressionEvaluator.GetValue(null, "7 % 4"); // 3

int one = (int) ExpressionEvaluator.GetValue(null, "8.0 % 5e0 % 2"); //
1

// Exponent

int sixteen = (int) ExpressionEvaluator.GetValue(null, "-2 ^ 4"); // 16

// Operator precedence

int minusFortyFive = (int) ExpressionEvaluator.GetValue(null,
"1+2-3*8^2/2/2"); // -45
```

10.3.5.赋值

在表达式中可以用等号为属性赋值。因为 ExpressionEvaluator 类的 SetValue 方法可以直接用来赋值，所以用等号赋值一般是用在 GetValue 方法中。在用表达式列表组合多个操作数时，这种赋值方法就很有用，下一小节会讨论表达式列表，现在先看一个使用赋值表达式的例子：

```
Inventor inventor = new Inventor();
String aleks = (String) ExpressionEvaluator.GetValue(inventor, "Name =
'Aleksandar Seovic'");
DateTime dt = (DateTime) ExpressionEvaluator.GetValue(inventor, "DOB =
date('1974-08-24')");

//Set the vice president of the society
Inventor tesla = (Inventor) ExpressionEvaluator.GetValue(ieee,
"Officers['vp'] = Members[0]");
```

10.3.6.表达式列表

在花括号中用分号分隔多个表达式，可以使这些表达式同时（按：注意此处不是指并发，而是象编程语言中的语句块一样，“同时”执行）执行。表达式列表的返回值是列表中最后一个表达式的值。请看下面的例子：

```
//Perform property assignments and then return Name property.

String pupin = (String) ExpressionEvaluator.GetValue(ieee.Members,
  "( [1].PlaceOfBirth.City = 'Beograd'; [1].PlaceOfBirth.Country =
'Serbia'; [1].Name )"));

// pupin = "Mihajlo Pupin"
```

10.3.7.类型

很多情况下，都可以通过名称来引用一个类型：

```
ExpressionEvaluator.GetValue(null, "1 is int")

ExpressionEvaluator.GetValue(null, "DateTime.Today")

ExpressionEvaluator.GetValue(null, "new string[] {'abc', 'efg'}")
```

如果类型是定义在 mscorlib.dll 中的，或是通过 TypeRegistry 注册过的，都可以通过名称直接引用。我们会在下一小节讨论类型注册。

对于其它类型，则需要使用特殊的 T(typeName)表达式：

```
Type dateType = (Type) ExpressionEvaluator.GetValue(null,
"T(System.DateTime)")

Type evalType = (Type) ExpressionEvaluator.GetValue(null,
"T(Spring.Expressions.ExpressionEvaluator, Spring.Core)")

bool trueValue = (bool) ExpressionEvaluator.GetValue(tesla,
"T(System.DateTime) == DOB.GetType()")
```

注意

类型的解析是由 Spring.NET 的 ObjectUtils.ResolveType 方法完成的，也就是说，定义在表达式中的类型，其解析方式和定义在配置文件中的类型是完全一样的。

10.3.8.类型注册

如果要在表达式中使用自定义类型（按：指没有在 mscorlib 中定义的类型），需要先用 TypeRegistry 类进行注册。注册以后即可以用名称来引用该类型，比

如在 new 操作符（按：下一小节）或静态属性的引用表达式中就经常会用名称来引用类型。请看下面的例子：

```
TypeRegistry.RegisterType("Society", typeof(Society));

Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(ieee,
"Officers[Society.President]");
```

此外，我们也可以在配置文件中用 typeAliases 节点注册类型。

10.3.9.构造器

在表达式中，可以用 new 操作符调用类型的构造器。如果类型不是 mscorlib 中的标准类型，需要先进行注册。请看下面的例子：

```
// simple ctor
DateTime dt = (DateTime) ExpressionEvaluator.GetValue(null, "new
DateTime(1974, 8, 24)");

// Register Inventor type then create new inventor instance within Add
method inside an expression list.
// Then return the new count of the Members collection.

TypeRegistry.RegisterType(typeof(Inventor));
int three = (int) ExpressionEvaluator.GetValue(ieee.Members, "{ Add(new
Inventor('Aleksandar Seovic', date('1974-08-24'), 'Serbian'));
Count}"));
```

为方便起见，Spring.NET 允许在表达式中定义已命名构造器参数，其作用是在对象被初始化之后对其属性赋值，这与标准.NET 特性的创建方式类似。比如，下面的表达式创建了一个 Inventor 实例，并对其 Inventions 属性赋值。注意表达式中使用的 Inventor 构造器只有 3 个参数，最后一个等式就是已命名参数：

```
Inventor aleks = (Inventor) ExpressionEvaluator.GetValue(null, "new
Inventor('Aleksandar Seovic', date('1974-08-24'),'Serbian', Inventions
= {'SPELL'})");
```

这里唯一要遵守的规则是：已命名参数必须位于所有构造器参数之后，同.NET 特性的创建方式一样。

我们已经两次提到.NET 特性的创建方式，现在就来看看如何用 Spring.NET 的表达式语言为.NET 特性创建实例。在创建.NET 特性的实例时，我们可以用更为简短和熟悉的方式来代替标准的构造器调用，如下：

```
WebMethodAttribute webMethod = (WebMethodAttribute)
ExpressionEvaluator.GetValue(null, "@[WebMethod(true, CacheDuration =
60, Description = 'My Web Method')]");
```

可以看出，除了其中的前缀@，创建特性所用的表达式与在 C#中应用特性的语法在形式上完全一样。

然尔语法上的轻微差异并不是特性表达式与构造器调用表达式之间的唯一区别。在使用特性表达式时，后台所用的类型解析机制与调用构造器时也稍有不同，此时会同时尝试按指定的类型名称和按指定名称加上"Attribute"后缀两种方式来创建特性实例，这与 C#编译器的行为是一样的。

10.3.10.变量

在表达式中用#variableName 的格式来表示变量名。变量通过一个字典类型的参数传递给 ExpressionEvaluator 的 GetValue 或 SetValue 方法。（按：此处的变量是指表达式的变量，而非由 c#定义在代码中的变量；所有的变量都保存在一个字典中，变量名就是字典项的键值，变量值就是字典项的值。）

```
public static object GetValue(object root, string expression,
IDictionary variables)

public static void SetValue(object root, string expression, IDictionary
variables, object newValue)
```

变量名是字典的键值。请看下面的例子：

```
IDictionary vars = new Hashtable();
vars["newName"] = "Mike Tesla";
ExpressionEvaluator.GetValue(tesla, "Name = #newName", vars));
```

表达式内部求值的结果也可以用字典来保存。下面的例子将 Tesla 的名字改回原来的值，并保存在键值为 oldName 的项中：

```
ExpressionEvaluator.GetValue(tesla, "{ #oldName = Name; Name = 'Nikola
Tesla' }", vars);
String oldName = (String)vars["oldName"]; // Mike Tesla
```

在索引器中或 map 中，也可以使用变量名作为参数：

```
vars["prez"] = "president";
Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(ieee,
"Officers[#prez]", vars);
```

10.3.10.1. '#this'和'#root'变量

在表达式中，有两个特殊的变量：#this 和#root。

#this 变量可以显式的引用表达式中当前节点的上下文：

```
// sets the name of the president and returns its instance
ExpressionEvaluator.GetValue(ieee, "Officers['president'].( #this.Name
= 'Nikola Tesla'; #this )")
```

而#root 变量用来引用表达式的根上下文：（按，注意它们的含义：表达式的“根上下文”，就是要在其上求值的对象，也就是说，是 GetValue 方法的第一个参数；而“表达式的节点”，是指表达式中以小数点分开的各个部分，比如 node1.node2.node3，node3 所在的上下文即为 node2）

```
// removes president from the Officers dictionary and returns removed
instance
ExpressionEvaluator.GetValue(ieee,
"Officers['president'].( #root.Officers.Remove('president'); #this )")
```

10.3.11. 三元操作符（If-Then-Else）

可以在表达式中用下面的三元操作符来表示 if-then-else 条件逻辑：

```
String aTrueString  = (String) ExpressionEvaluator.GetValue(null,
"false ? 'trueExp' : 'falseExp'") // trueExp
```

上面的 false 值使得表达的返回值为“trueExp”。下面是一个更为真实的例子：

```
ExpressionEvaluator.SetValue(ieee, "Name", "IEEE");
IDictionary vars = new Hashtable();
vars["queryName"] = "Nikola Tesla";

string expression = @"IsMember(#queryName)
                      ? #queryName + ' is a member of the ' + Name + '
Society'
                      : #queryName + ' is not a member of the ' + Name
+ ' Society'";

String queryResultString = (String) ExpressionEvaluator.GetValue(ieee,
expression, vars));

// queryResultString = "Nikola Tesla is a member of the IEEE Society"
```

10.3.12. 列表的投影（Projection）和选择（Selection）

列表的投影和选择是表达式语言一项很强大的功能，可以从源列表中选出某些列或某些行来创建一个新列表。也就是说，投影可以看做是和 SQL 中 Select 语句功能相似的列选择器，而选择则相当于 where 子句。

例如，假设我们需要一个发明家的出生地列表，通过投影 PlaceOfBirth.City 属性，很容易获取此列表：

```
IList placesOfBirth = (IList) ExpressionEvaluator.GetValue(ieee,
"Members.!{PlaceOfBirth.City}")  // { 'Smiljan', 'Idvor' }
```

我们也可以获取社团中要员的名称列表：

```
IList officersNames = (IList) ExpressionEvaluator.GetValue(ieee,
"Officers.Values.!{Name}")  // { 'Nikola Tesla', 'Mihajlo Pupin' }
```

从这两个例子可以看出，投影语句使用的语法是*!{projectionExpression}*，它返回的新列表长度和原列表一样，但元素的类型一般不相同。

另一方面，选择语句的语法为*?{selectionExpression}*，它返回的新列表是从源列表中过滤出的一个子集。例如，通过选择语句我们很容易获取一个塞尔维亚籍发明家的列表：

```
IList serbianInventors = (IList) ExpressionEvaluator.GetValue(ieee,
"Members.?{Nationality == 'Serbian'}")  // { tesla, pupin }
```

或者过滤出声纳的发明者：

```
IList sonarInventors = (IList) ExpressionEvaluator.GetValue(ieee,
"Members.?{'Sonar' in Inventions}")  // { pupin }
```

或者，同时使用投影和选择来获取声纳发明者的全名：

```
IList sonarInventorsNames = (IList) ExpressionEvaluator.GetValue(ieee,
"Members.?{'Sonar' in Inventions}.!{Name}")  // { 'Mihajlo Pupin' }
```

为方便起见，Spring.NET 的表达式语言可以用专门的语法选择第一个和最后一个匹配的列表项。注意这与正则表达式有区别，在用正则表达式选择时，如果找不到匹配项就会返回一个空列表，而首尾项选择表达式则会在找到匹配项时返回该项的引用，找不到时返回 null。要返回第一个匹配项，要在选择表达式中用^{代替?{，选择最后一个匹配项时应该用$｛：

```
ExpressionEvaluator.GetValue(ieee, "Members.^{Nationality ==
'Serbian'}.Name")  // 'Nikola Tesla'
```

```
ExpressionEvaluator.GetValue(ieee, "Members.${Nationality ==
'Serbian'}.Name")  // 'Mihajlo Pupin'
```

请注意，在上面代码中我们直接使用选择结果访问了 Name 属性，因为通过首尾项选择表达式返回的不是列表，而是列表中的元素。

10.3.13. 集合处理器和聚合器（Aggregator）

除了列表的投影和选择，Spring.NET 表达式语言还支持几种集合处理器，比如 distinct、nonNull 和 sort；以及一系列通用的聚合器，如 max、min、count、sum 和 average。

处理器和聚合器的区别是：处理器会返回一个新的或者转换过的集合，而聚合器返回的则是一个单一值。除此之外它们非常相似——处理器和聚合器都使用标准的方法调用表达式来处理集合的节点，它们都很简单，也很容易组成处理器链。

10.3.13.1. Count 聚合器

通过 Count 聚合器，可以安全的获取集合元素的总数。Count 适用于所有集合类型，包括数组，我们不需要考虑应该使用 Count 属性还是 Length 属性来获取集合的大小，并且，即便集合为 null，也不会抛出 NullReferenceException 异常，而是返回 0，在复杂的表达式中，这样要比使用.NET 的标准属性更为安全。

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3}.count()")  // 3
ExpressionEvaluator.GetValue(null, "count()")  // 0
```

10.3.13.2. Sum 聚合器

如果列表元素是数字值，可以用 sum 聚合器计算所有元素的总和。如果列表中数字的类型或精度不一致，则会自动执行必要的转型，然后以其中最大精度的类型返回结果值。如果集合中的元素都不是数字，就会抛出 InvalidArgumentException 异常。

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3, 10}.sum()")  // 13 (int)
ExpressionEvaluator.GetValue(null, "{5, 5.8, 12.2, 1}.sum()")  // 24.0
(double)
```

10.3.13.3. Average 聚合器

average 聚合器用于返回数字集合所有元素的平均值。在类型上，average 使用和 sum 一样的规则以最大程度的保证求值的精度。如果集合中的元素都不是数字，也和 sum 一样，抛出 InvalidArgumentException。

```
ExpressionEvaluator.GetValue(null, "{1, 5, -4, 10}.average()")  // 3
ExpressionEvaluator.GetValue(null, "{1, 5, -2, 10}.average()")  // 3.5
```

10.3.13.4. Minimum 聚合器

minimun 聚合器返回列表元素中的最小值。为了确定“最小值”的真正含义，minimun 要求目标集合中所有元素的类型相同并且都实现了 IComparable 接口。否则，该聚合器会抛出 InvalidArgumentException 异常。

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3, 10}.min()")  // -3
ExpressionEvaluator.GetValue(null, "{'abc', 'efg', 'xyz'}.min()")  //
'abc'
```

10.3.13.5. Maximum 聚合器

maximum 聚合器返回列表元素中的最大值。为了确定“最大值”的真正意义，maximum 要求目标集合中所有元素的类型相同并且实现了 IComparable 接口。否则，该聚合器会抛出 InvalidArgumentException 异常。

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3, 10}.max()")  // 10
ExpressionEvaluator.GetValue(null, "{'abc', 'efg', 'xyz'}.max()")  //
'xyz'
```

10.3.13.6. nonNull 处理器

nonNull 处理器非常简单，作用是从集合中清除所有空值（null）。

```
ExpressionEvaluator.GetValue(null, "{'abc', 'xyz', null, 'abc', 'def',
null}.nonNull()")  // { 'abc', 'xyz', 'abc', 'def' }
ExpressionEvaluator.GetValue(null, "{'abc', 'xyz', null, 'abc', 'def',
null}.nonNull().distinct().sort()")  // { 'abc', 'def', 'xyz' }
```

10.3.13.7. distinct 处理器

distinct 处理器可以确保集合中不包含重复的元素。该处理器还可以接收一个可选的布尔参数，用来决定结果中是否应该包含空值。该参数默认值是 false，也就是说结果集合不包含空值。

```
ExpressionEvaluator.GetValue(null, "{'abc', 'xyz', 'abc', 'def', null, 'def'}.distinct(true).sort()")  // { null, 'abc', 'def', 'xyz' }
ExpressionEvaluator.GetValue(null, "{'abc', 'xyz', 'abc', 'def', null, 'def'}.distinct(false).sort()")  // { 'abc', 'def', 'xyz' }
```

10.3.13.8. sort 处理器

如果集合元素的类型统一且实现了 IComparable 接口，就可以用 sort 处理器进行排序。

```
ExpressionEvaluator.GetValue(null, "{1.2, 5.5, -3.3}.sort()")  // { -3.3, 1.2, 5.5 }
ExpressionEvaluator.GetValue(null, "{'abc', 'xyz', 'abc', 'def', null, 'def'}.sort()")  // { null, 'abc', 'abc', 'def', 'def', 'xyz' }
```

10.3.14.引用容器中的对象

表达式也可以引用定义在 IoC 容器中的对象，语法为@contextName:objectName（按：contextName 是指 XML 中<context>节点的名字，而非代码中的 IApplicaitonContext 变量的名称)。如果找不到以 contextName 命名的上下文，就使用根上下文的名字（Spring.RootContext）。可以参考第二十五章, IoC 快速入门中的例子，下面的代码可以返回由 Roberto Benigni 指导的电影。（按：Roberto Benigni，罗伯特·贝尼尼，意大利著名演员，代表作有《美丽人生》、《皮诺曹》等，有不少是他自己编剧或导演的）

```
public static void Main()
{
  . . .

// Retrieve context defined in the spring/context section of
// the standard .NET configuration file.
IApplicationContext ctx = ContextRegistry.GetContext();

int numMovies = (int) ExpressionEvaluator.GetValue(null,
                      "@(MyMovieLister).MoviesDirectedBy('Roberto Benigni').Length");
```

```
  . . .
}
```

最后，变量 numMovies 的值为 2。

10.3.15. Lambda 表达式

Lambda 表达式是 Spring.NET 表达式语言中一项稍显复杂但非常强大的功能。Lambda 表达式允许在表达式中定义内联函数，并且可以象真正的函数或方法一样在表达式中调用。

定义 Lambda 表达式的语法如下：

#*functionName* = {|*argList*| *functionBody* }

例如，可以在表达式中定义一个 max 函数并调用它：

```
ExpressionEvaluator.GetValue(null, "(#max = {|x,y| $x > $y ? $x : $y }; #max(5,25))", new Hashtable())  // 25
```

可以看到，在函数体内引用前面定义的参数时，要使用*本地变量*的语法，即$加变量名。在调用时，将参数值以逗号分隔，列在函数名后的小括号内。

Lambda 表达式支持递归，也就是说可以在函数体内部调用自身：

```
ExpressionEvaluator.GetValue(null, "(#fact = {|n| $n <= 1 ? 1 : $n * #fact($n-1) }; #fact(5))", new Hashtable())  // 120
```

请注意，在上面的两个例子中，我们必须为 GetValue 方法指定一个字典类型的参数，这里我们用哈希表。因为 Lambda 表达式实际上只是一些参数化变量，我们需要用一个字典来保存这些变量。如果不给 GetValue 方法指定一个有效的 IDictionary 实例，就会发生运行时错误。请参考 10.3.10.节的相关内容。

另外，上面两个例子都是在同一表达式内定义和调用函数的。但一般来说我们希望定义了函数后能够在多个表达式中使用，Spring.NET 通过一个静态方法 Expression.RegisterFunction 来预注册 Lambda 表达式，该方法接受函数名，Lambda 表达式和存放变量的字典作为参数，如下：

```
IDictionary vars = new Hashtable();
Expression.RegisterFunction("sqrt", "{|n| Math.Sqrt($n)}", vars);
Expression.RegisterFunction("fact", "{|n| $n <= 1 ? 1 : $n * #fact($n-1)}", vars);
```

一旦注册了函数，就可以在其它表达式中使用它们，注意要确保在每次求值时将字典变量 vars 传给了表达式求值引擎：

```
ExpressionEvaluator.GetValue(null, "#fact(5)", vars)  // 120
ExpressionEvaluator.GetValue(null, "#sqrt(9)", vars)  // 3
```

最后，请注意 Lambda 表达式本身也是按变量来处理的，它们可以被赋值给其它变量或作为参数传递给其他 Lamba 表达式。在下面的例子中，我们定义了一个委托函数，它有两个参数：第一个 f 要调用的函数，第二个 n 则是要传递给 f 的值。随后我们通过这个委托来调用前面注册过的两个函数，然后又调用了一个内联函数：

```
Expression.RegisterFunction("delegate", "{|f, n| $f($n) }", vars);
ExpressionEvaluator.GetValue(null, "#delegate(#sqrt, 4)", vars)  // 2
ExpressionEvaluator.GetValue(null, "#delegate(#fact, 5)", vars)  // 120
ExpressionEvaluator.GetValue(null, "#delegate({|n| $n ^ 2 }, 5)", vars)
// 25
```

虽然这个例子不算实用，但它确实证明了一点：Lambda 表达式其实是参数化的变量，我们要牢记这一点。

10.3.16.空目标（Null Context）

如果没有在 GetValue 方法中指定根对象而使用了 null（按：即第一个参数，前面已经有多处这样的用法）。那么表达式必须满足下面的某种情况：

- 字面值，如：ExpressionEvaluator.GetValue(null, "2 + 3.14");
- 引用某个类型的静态方法或字段，如：ExpressionEvaluator.GetValue(null,"DateTime.Today");
- 创建新的对象，如：ExpressionEvaluator.GetValue(null,"new DateTime(2004, 8, 14)");
- 或是引用其它对象，比如定义在变量词典或 IoC 容器中的对象。

后两种用法以后会专门讨论。

10.4. 本章使用的示例类型

下面的类是在本章例子中使用的类型：

```
public class Inventor
{
    public string Name;
    public string Nationality;
```

```
    public string[] Inventions;
    private DateTime dob;
    private Place pob;

    public Inventor() : this(null, DateTime.MinValue, null)
    {}

    public Inventor(string name, DateTime dateOfBirth, string
nationality)
    {
        this.Name = name;
        this.dob = dateOfBirth;
        this.Nationality = nationality;
        this.pob = new Place();
    }

    public DateTime DOB
    {
        get { return dob; }
        set { dob = value; }
    }

    public Place PlaceOfBirth
    {
        get { return pob; }
    }

    public int GetAge(DateTime on)
    {
        // not very accurate, but it will do the job ;-)
        return on.Year - dob.Year;
    }
}

public class Place
{
    public string City;
    public string Country;
}

public class Society
{
    public string Name;
    public static string Advisors = "advisors";
```

```
    public static string President = "president";

    private IList members = new ArrayList();
    private IDictionary officers = new Hashtable();

    public IList Members
    {
        get { return members; }
    }

    public IDictionary Officers
    {
        get { return officers; }
    }

    public bool IsMember(string name)
    {
        bool found = false;
        foreach (Inventor inventor in members)
        {
            if (inventor.Name == name)
            {
                found = true;
                break;
            }
        }
        return found;
    }
}
```

下面是例子中使用的数据：

```
Inventor tesla = new Inventor("Nikola Tesla", new DateTime(1856, 7, 9),
"Serbian");
tesla.Inventions = new string[]
    {
        "Telephone repeater", "Rotating magnetic field principle",
        "Polyphase alternating-current system", "Induction motor",
        "Alternating-current power transmission", "Tesla coil
transformer",
        "Wireless communication", "Radio", "Fluorescent lights"
    };
tesla.PlaceOfBirth.City = "Smiljan";
```

```
Inventor pupin = new Inventor("Mihajlo Pupin", new DateTime(1854, 10, 9),
"Serbian");
pupin.Inventions = new string[] {"Long distance telephony & telegraphy",
"Secondary X-Ray radiation", "Sonar"};
pupin.PlaceOfBirth.City = "Idvor";
pupin.PlaceOfBirth.Country = "Serbia";

Society ieee = new Society();
ieee.Members.Add(tesla);
ieee.Members.Add(pupin);
ieee.Officers["president"] = pupin;
ieee.Officers["advisors"] = new Inventor[] {tesla, pupin};
```

第十一章. 验证框架

11.1. 简介

数据验证是企业应用中一个非常重要的部分。ASP.NET 本身有一个验证框架，但是功能十分有限，无法执行复杂的验证。Perter Blum 的网站上有详细的文档对 ASP.NET 验证框架的已知问题进行阐述，我们在这儿就不专门列举了。另外，Peter 还开发了一套验证框架作为 ASP.NET 本身验证框架的替代品，如果您喜欢标准的 ASP.NET 验证方式，那么 Peter 的框架很值得参考一下。Peter 和 Spring.NET 的验证框架都可以进行非常复杂的验证，不过，出于稍后列出的几条原因，Spring.NET 的验证框架在设计上与 Peter 的框架有很大的差别。

对于 Windows 窗体应用程序来说，情况更加糟糕。按照 Lan Griffiths 在他论文中的说法，目前流行的验证机制基本没有一个能满足要求。主要原因是大部分验证框架（不管是开源的还是商业的）都把验证行为与特定的表示层技术绑定在一起。APS.NET 的验证框架使用 Web 控件来定义验证规则，所以这些规则对于 HTML 标记就不起作用。Peter Blum 的框架采用的也是相同的方式。我们认为，验证不应只适用于表示层，不应该把验证行为和任何表示层绑定在一起。因此，Spring.NET 验证框架的设计目标就是使应用程序的不同层次都能使用同一验证规则进行数据验证。

设计验证框架的目标如下：

1. 允许对任意对象进行验证，不管是 UI 框架还是领域对象。
2. 允许在 Windows 窗体、APS.NET 以及服务层（服务层需要验证传来的参数）中使用统一的验证框架。
3. 允许对验证规则进行组合以支持任意复杂的验证行为。
4. 允许根据特定条件启用/禁用验证。

在后面的小节中，我们会了解到 Spring.NET 是如何实现这些目标的，并学习如何在应用开发中使用 Spring.NET 的验证框架。

11.2. 用法示例

验证框架的主要目标是将验证行为与表示层解耦。基本上，我们希望能够定义一系列独立于表示层的验证规则，以便在不同的应用层中重用它们。这样以来，微软 ASP.NET 本身的验证方式就无法满足要求了，所以我们不打算通过自定义验证控件来实现验证框架。我们所采用的方式是：在应用程序上下文中，象配置普通对象一样来配置验证规则。不过，由于验证规则的复杂性，我们决定不用标准的 schema 来定义验证对象，而是用专门的、更加易用的自定义 schema。注意，配置框架并未和 IoC 容器绑定，可以通过编程方式单独使用。下面的例子是在 SpringAir 示例项目中为 Trip 对象定义的验证规则：

```
<objects xmlns="http://www.springframework.net"
xmlns:v="http://www.springframework.net/validation">

  <object type="TripForm.aspx" parent="standardPage">
     <property name="TripValidator" ref="tripValidator" />
  </object>

  <v:group id="tripValidator">
     <v:required id="departureAirportValidator"
test="StartingFrom.AirportCode">
        <v:message id="error.departureAirport.required"
providers="departureAirportErrors, validationSummary"/>
     </v:required>

     <v:group id="destinationAirportValidator">
        <v:required test="ReturningFrom.AirportCode">
           <v:message id="error.destinationAirport.required"
providers="destinationAirportErrors, validationSummary"/>
        </v:required>

        <v:condition test="ReturningFrom.AirportCode !=
StartingFrom.AirportCode" when="ReturningFrom.AirportCode != ''">
           <v:message id="error.destinationAirport.sameAsDeparture"
providers="destinationAirportErrors, validationSummary"/>
        </v:condition>
     </v:group>

     <v:group id="departureDateValidator">
        <v:required test="StartingFrom.Date">
```

```
            <v:message id="error.departureDate.required"
providers="departureDateErrors, validationSummary"/>
        </v:required>
        <v:condition test="StartingFrom.Date >= DateTime.Today"
when="StartingFrom.Date != DateTime.MinValue">
            <v:message id="error.departureDate.inThePast"
providers="departureDateErrors, validationSummary"/>
        </v:condition>
     </v:group>

     <v:group id="returnDateValidator" when="Mode == 'RoundTrip'">
        <v:required test="ReturningFrom.Date">
            <v:message id="error.returnDate.required"
providers="returnDateErrors, validationSummary"/>
        </v:required>
        <v:condition test="ReturningFrom.Date >= StartingFrom.Date"
when="ReturningFrom.Date != DateTime.MinValue">
            <v:message id="error.returnDate.beforeDeparture"
providers="returnDateErrors, validationSummary"/>
        </v:condition>
     </v:group>
  </v:group>

</objects>
```

在这个例子中，有几件事情需要注意一下：

- 在根结点<objects>中，需要添加 xmlns:v="http://www.springframework.net/validation"命名空间来引用验证 schema。
- 因为在根结点中引用了标准和验证两种 schema，所以可以在同一配置文件中定义普通对象和验证对象。
- 配置文件中定义的验证对象由其 id 属性惟一确定，可以用标准的 Spring 方式引用，例如，上面的配置将 tripValidator 注入到了前面定义的 TripForm.aspx 页面中。
- 验证框架使用 Spring 强大的表达式求值引擎来处理验证对象的验证规则和应用条件。所以，在验证对象定义的 test 和 when 属性中，可以使用任何有效的 Spring 表达式。

上面这个例子涉及到验证框架的很多功能，我们来逐个讨论它们。

11.3. 验证对象组

验证对象组的重要性从很多方面都能体现出来，最典型的用例是用多个验证规则来验证同一个值。在前一小节的例子中，我们为 Trip 对象的每个属性都定义了一组验证对象。最上层的分组还定义了针对 Trip 对象本身的验证对象。

验证对象组共有三种类型，它们的行为各不相同：

第一种类型“与”（AND)绝对是最常用的;另外两种则是“或“和“异或”（OR、XOR），也可以用很简单的方式处理某些特殊的验证需求。在设计验证规则时，要牢记这三种类型：

表 11.1. 验证对象组

类型	XML 标签	行为
AND	group	仅当其中的所有验证对象返回 true 才返回 true。这是最常用的验证对象组。
OR	any	只要其中有对象返回 true 就返回 true。
XOR	exclusive	仅当其中一个验证对象为 true 时返回 true。

要记住，验证对象组实际就是一个验证对象，可以应用在任何能使用普通验证对象的场合。验证对象组支持嵌套，可以用验证对象的专用引用语句（稍后讨论）来引用它们，这样能最大程度的满足重用验证规则的要求。

11.4. 验证对象（Validator）

首先，要为每一项需要验证的数据定义一或多个验证对象。Spring.NET 内置的验证对象可以满足大部分要求，它们已经能够处理相当复杂的验证。验证框架是可扩展的，所以，必要时可以创建自定义验证对象，使用的方法与内置验证对象一样。

11.4.1. 条件验证对象（Condition Validator）

条件验证对象可对任何有效的 Spring 表达式进行求值，定义的语法为：

```
<v:condition id="id" test="testCondition"
when="applicabilityCondition" parent="parentValidator">
   actions
</v:condition>
```

下面是一个例子（按：这个定义是包含在一个验证组内的，是针对 Trip 对象的验证对象，见前文的代码）：

```
<v:condition test="StartingFrom.Date >= DateTime.Today"
when="StartingFrom.Date != DateTime.MinValue">
   <v:message id="error.departureDate.inThePast"
providers="departureDateErrors, validationSummary"/>
</v:condition>
```

在本例中，验证对象会判断 Trip 对象的 StartingFrom 属性是否不小于当前日期，且仅当该属性被赋初值以后才进行验证（DateTime 若不赋初值，其值为 DateTime.MinValue）。

我们可以认为条件验证对象是“验证对象之母”。条件验证对象可以完成其它验证对象能做的任何验证，但在某些情况下，条件验证对象的 test 属性会相当复杂，所以只要可能我们还是应该使用更为专用的验证对象。不过，在需要检查某个值是否属于特定范围，或进行其它类似的验证时，条件验证对象仍然是最佳选择，因为这些条件一般都很简单。

> **注意**
>
> 要记住 Spring.NET 验证框架一般是和业务对象一起工作的。验证发生在控件的数据绑定之后，以确保要验证的对象已经是强类型的（按：绑定时会作相应转换）。所以，在验证对象定义的表达式中可以直接比较数字和日期，不用担心 Spring.NET 会按字符串来处理。

11.4.2. 必需性验证对象（Required Validator）

必需性验证对象可以确保所验证的值不为空，语法为：

```
<v:required id="id" test="requiredValue" when="applicabilityCondition"
parent="parentValidator">
   actions
</v:required>
```

下面是一个例子：

```
<v:required test="ReturningFrom.AirportCode">
   <v:message id="error.destinationAirport.required"
providers="destinationAirportErrors, validationSummary"/>
</v:required>
```

其中的 test 属性的值就是要验证的目标。

表 11.2. 检测必需值是否有效的规则

类型	判断
System.Type	对象是否存在
System.String	是否为 null 或空字符串

必需性验证对象也是最常用的验证对象之一，它比 ASP.NET 本身的必需性验证控件要强大的多，因为它的目标对象可以是各种强类型数据，而不只是字符串。例如，DateTime 类型(注意在对 DateTime 进行必需性验证时，MinValue 和 MaxValue 都会返回 false）、整型、小数、以及其它引用类型。对于引用类型，如果值非空（null）则返回 true，否则返回 false。

必需性验证对象定义中的 test 属性一般是某个领域对象的属性，但也可以是任何能够返回值的有效表达式，比如方法调用。

11.4.3.正则表达式验证对象

其语法为：

```
<v:regex id="id" test="valueToEvaluate" when="applicabilityCondition"
parent="parentValidator">
  <v:property name="Expression" value="regularExpressionToMatch"/>
  <v:property name="Options" value="regexOptions"/>
  actions
</v:regex>
```

请看下面的例子：

```
<v:regex test="ReturningFrom.AirportCode">
   <v:property name="Expression" value="[A-Z][A-Z][A-Z]"/>
   <v:message id="error.destinationAirport.threeCharacters"
providers="destinationAirportErrors, validationSummary"/>
</v:regex>
```

如果需要验证某个值是否满足预定义的格式，可使用正则表达式验证对象，比如验证某个值是否是电话号码、邮件地址或 URL 等。

正则表达式验证对象在定义上与其它验证对象最大的区别是需要设置一个名为 Expression 的属性，该属性是必需的，其值即为用于匹配的模式字符串。

11.4.4. 通用验证对象（Generic Validator）

语法为：

```
<v:validator id="id" test="requiredValue"
when="applicabilityCondition" type="validatorType"
parent="parentValidator">
  actions
</v:validator>
```

下面是一个例子：

```
<v:validator test="ReturningFrom.AirportCode"
type="MyNamespace.MyAirportCodeValidator, MyAssembly">
   <v:message id="error.destinationAirport.invalid"
providers="destinationAirportErrors, validationSummary"/>
</v:required>
```

在通用验证对象定义中，可以用类型名称来引用自定义验证对象，以执行实际的验证操作（按：见上例的 type 属性）。自定义验证对象很容易实现，只要继承抽象基类 BaseValidator，实现它的 bool Validate(object objectToValidate) 抽象方法即可，在该方法中，如果要检查的对象满足条件，就返回 true，否则就返回 false。

11.4.5. 条件型验证（Conditional Validator Execution）

从前面的例子中，我们看到每种验证对象（及验证对象组）都可以通过 when 属性定义一个应用条件。ASP.NET 验证框架最大的缺陷之一就是没有这项功能。这项功能非常有用，在很多情况下，都可能需要针对某些值启用或关闭特定的验证过程。

例如，在验证 Trip 对象时，仅当 Trip.Mode 属性值为 TripMode.RoundTrip 时才需要验证回程日期。为此，我们使用下面的验证对象定义：

```
<v:group id="returnDateValidator" when="Mode == 'RoundTrip'">
      // nested validators
</v:group>
```

该组内定义的所有验证对象都只在 Trip.Mode 属性值为 TripMode.RoundTrip 时有效。

> **注意**
>
> 您可能注意到了，returnDateValidator 验证对象组使用枚举值的名称字符串来比较枚举值。在这里也可以用枚举值的全名，例如：
>
> ```
> Mode == TripMode.RoundTrip
> ```
>
> 但是，必须先用 Spring.NET 的类型别名机制为 TripMode 枚举类型定义别

名。

11.5. 验证行为

我们可以在验证对象定义中定义验证行为。每次执行验证时验证行为都会被执行。在验证行为中，可以根据验证的结果做任何希望的工作。目前最常用的验证行为是向错误集合中添加验证失败信息，但理论上讲验证行为可以执行任何操作。由于向错误集合中添加错误信息是特别常见的情况，所以 Spring.NET 在验证 schema 中定义了一个专门的 XML 节点来配置这种验证行为。

11.5.1. 错误消息行为

语法为：

```
<v:message id="messageId" providers="errorProviderList"
when="messageApplicabilityCondition">
  <v:param value="paramExpression"/>
</v:message>
```

请看下面的例子：

```
<v:message id="error.departureDate.inThePast"
providers="departureDateErrors, validationSummary">
   <v:param value="StartingFrom.Date.ToString('D')"/>
   <v:param value="DateTime.Today.ToString('D')"/>
</v:message>
```

在处理错误消息时，需要注意几件事情：

- id 属性的值用来在某个消息源中查找错误信息。
- providers 属性用一个以逗号分隔的字符串来指定一个“错误桶”，这个“桶”可以用来保存特定的错误信息，并且用特定的显示机制在必要的时候将错误信息显示出来。
- 错误消息可以有 0 或多个参数。每个参数都是一个有效的 Spring 表达式，表达式的值将传递给 IMessageSource.GetMessage 方法以返回一个完整的消息。

11.5.2. 通用行为

通用行为的定义如下：

```
<v:action type="actionType" when="actionApplicabilityCondition">
```

```
  properties
</v:action>
```

下面是一个例子：

```
<v:action type="Spring.Validation.Actions.ExpressionAction,
Spring.Core" when="#page != null">
   <v:property name="Valid" value="#page.myPanel.Visible = true"/>
   <v:property name="Invalid" value="#page.myPanel.Visible = false"/>
</v:action>
```

通用行为可以执行任意操作。拿上面的例子来说，我们用 ExpressionAction 类和两条简单的表达式进行验证，并用验证的结果来调整控件的可见性。

在某些情况下，可能需要创建自定义的验证行为，方法很简单，只要实现 IValidationAction 接口即可：

```
public interface IValidationAction
{
    /// <summary>
    /// Executes the action.
    /// </summary>
    /// <param name="isValid">Whether associated validator is valid or
not.</param>
    /// <param name="validationContext">Validation context.</param>
    /// <param name="contextParams">Additional context
parameters.</param>
    /// <param name="errors">Validation errors container.</param>
    void Execute(bool isValid, object validationContext, IDictionary
contextParams, ValidationErrors errors);
}
```

11.6. 引用验证对象

有时候,不太可能(或者不希望)将所有的验证规则都放在顶层的验证对象组内。比如，如果有两个对象 ObectA 和 ObjectB，它们都会引用另一个对象 ObjectC，我们可能打算为 ObjectC 单独配置一个验证规则,在 ObjectA 和 ObjectB 的验证规则中去引用它，而不需要将 ObjectC 的验证规则再重复一遍。

引用验证规则的语法如下：

```
<v:ref name="referencedValidatorId"
context="validationContextForTheReferencedValidator"/>
```

下面看一个例子：

```
<v:group id="objectA.validator">
   <v:ref name="objectC.validator" context="MyObjectC"/>
   // other validators for ObjectA
</v:group>

<v:group id="objectB.validator">
   <v:ref name="objectC.validator" context="ObjectCProperty"/>
   // other validators for ObjectB
</v:group>

<v:group id="objectC.Validator">
   // validators for ObjectC
</v:group>
```

就这么简单：为 ObjectC 单独定义一个验证对象组，然后在其它验证对象组中引用它。请注意，如果我们需要将被引用的验证对象限制在某个适用范围（context）内，可以将 context 属性的值设为要验证的属性名。在上例中，ObjectA.MyObjectC 和 ObjectB.ObjectCProperty 的类型都是 ObjectC。在 ObjectA 和 ObjectB 的验证对象中，我们打算让 ObjectC 的验证规则只对 ObjectA 的 MyObjectC 属性和 ObjectB 的 ObjectCProperty 属性适用，所以在用<v:ref/>引用 object.Validator 时，用 context 属性将 objectC.validator 的验证范围限定在了这两个属性上。（按：如果 Object 另有一个类型为 ObjectC 的属性 PropC，那么验证对象不会对 PropC 执行验证。）

11.7. 在 ASP.NET 中的使用技巧

我们已经了解了验证规则的配置方法，现在来看看如何在 ASP.NET 应用程序中进行验证并显示错误信息。

首先，要将验证对象注入到 ASP.NET 页面中，如下所示：

```
<objects xmlns="http://www.springframework.net"
xmlns:v="http://www.springframework.net/validation">

  <object type="TripForm.aspx" parent="standardPage">
     <property name="TripValidator" ref="tripValidator" />
  </object>

  <v:group id="tripValidator">
     // our validation rules
  </v:group>
```

```
</objects>
```

然后，就要在各个页面事件的处理器中进行验证了，验证的代码一般如下所示：

```
public void SearchForFlights(object sender, EventArgs e)
{
    if (Validate(Controller.Trip, tripValidator))
    {
        Process.SetView(Controller.SearchForFlights());
    }
}
```

注意

要让上面的代码正常工作，页面需要继承自 Spring.Web.UI.Page 类。

最后，需要在 Web 窗体内用<spring:validationError>和<spring:validationSummary>控件来显示错误信息。

```
<%@ Page Language="c#" MasterPageFile="~/Web/StandardTemplate.master"
Inherits="TripForm" CodeFile="TripForm.aspx.cs" %>

<%@ Register TagPrefix="spring"
Namespace="Spring.Web.Anthem.UI.Controls"
Assembly="Spring.Web.Anthem" %>
<%@ Register TagPrefix="anthem" Namespace="Anthem" Assembly="Anthem" %>

<asp:Content ID="head" ContentPlaceHolderID="head" runat="server">

    <script language="javascript" type="text/javascript">
      <!--
      function showReturnCalendar(isVisible)
      {
          document.getElementById('<%=
returningOnDate.ClientID %>').style.visibility = isVisible? '':
'hidden';

document.getElementById('returningOnCalendar').style.visibility =
isVisible? '': 'hidden';
      }
      -->
    </script>

</asp:Content>

<asp:Content ID="body" ContentPlaceHolderID="body" runat="server">
```

```
    <div style="text-align: center">
        <h4><asp:Label ID="caption" runat="server"></asp:Label></h4>
        <spring:ValidationSummary ID="validationSummary"
runat="server" />

        <table>
            <tr class="formLabel">
                <td> </td>
                <td colspan="3">
                    <spring:RadioButtonGroup ID="tripMode"
runat="server">

                        <asp:RadioButton ID="OneWay"
onclick="showReturnCalendar(false);" runat="server" />
                        <asp:RadioButton ID="RoundTrip"
onclick="showReturnCalendar(true);" runat="server" />
                    </spring:RadioButtonGroup>
                </td>
            </tr>
            <tr>

                <td class="formLabel" align="right">
                    <asp:Label ID="leavingFrom" runat="server" /></td>
                <td nowrap="nowrap">
                    <anthem:DropDownList ID="leavingFromAirportCode"
AutoCallBack="true" runat="server" />
                    <spring:ValidationError
id="departureAirportErrors" runat="server" />
                </td>

                <td class="formLabel" align="right">
                    <asp:Label ID="goingTo" runat="server" /></td>
                <td nowrap="nowrap">
                    <anthem:DropDownList ID="goingToAirportCode"
AutoCallBack="true" runat="server" />
                    <spring:ValidationError
id="destinationAirportErrors" runat="server" />
                </td>

            </tr>
            <tr>
                <td class="formLabel" align="right">
                    <asp:Label ID="leavingOn" runat="server" /></td>
                <td nowrap="nowrap">
```

```
                    <spring:Calendar ID="leavingFromDate"
runat="server" Width="75px" AllowEditing="true" Skin="system" />

                    <spring:ValidationError id="departureDateErrors"
runat="server" />
                </td>
                <td class="formLabel" align="right">
                    <asp:Label ID="returningOn" runat="server" /></td>
                <td nowrap="nowrap">
                    <div id="returningOnCalendar">

                        <spring:Calendar ID="returningOnDate"
runat="server" Width="75px" AllowEditing="true" Skin="system" />
                        <spring:ValidationError id="returnDateErrors"
runat="server" />
                    </div>
                </td>
            </tr>
            <tr>

                <td class="buttonBar" colspan="4">
                    <br/>
                    <anthem:Button ID="findFlights"
runat="server"/></td>
            </tr>
        </table>
    </div>

    <script language="javascript" type="text/javascript">
          if (document.getElementById('<%=
tripMode.ClientID %>').value == 'OneWay')
              showReturnCalendar(false);
          else
              showReturnCalendar(true);
    </script>

</asp:Content>
```

11.7.1. 显示验证错误

Spring.NET 可以用几种不同的方式来显示验证错误信息，如果这些方式都不能满足需要，我们也可以创建自己的验证错误显示控件。错误显示控件都要实现

Spring.Web.Validation.IValidationErrorsRenderer 接口，该接口在 Spring.NET 中有以下实现类：

表 11.3. 验证显示控件

名称	类	描述
Block	Spring.Web.Validation.DivValidationErrorsRenderer	将错误信息以列表形式显示在 标签内。这是<spring:validationSummary>控件的默认显示控件。
Inline	Spring.Web.Validation.SpanValidationErrorsRenderer	将验证错误显示在标签内。这是<spring:vallidationError>控件的默认显示控件。
Icon	Spring.Web.Validation.IconValidationErrorsRenderer	将错误信息显示为图标，并在图标的 Tooltip 中显示错误信息。如果很重视屏幕的实时状态，该类是最佳选择。

这三个错误显示类可以满足大部分需求，但如果需要用其它方式显示错误，可以实现 Spring.Web.Validation.IValidationErrorsRenderer 接口：

```
namespace Spring.Web.Validation
{
    /// <summary>
    /// This interface should be implemented by all validation errors
renderers.
    /// </summary>
    /// <remarks>
    /// <para>
    /// Validation errors renderers are used to decouple rendering
behavior from the
    /// validation errors controls such as <see cref="ValidationError"/>
and
    /// <see cref="ValidationSummary"/>.
    /// </para>
    /// <para>
    /// This allows users to change how validation errors are rendered
by simply pluggin in
```

```
    /// appropriate renderer implementation into the validation errors
controls using
    /// Spring.NET dependency injection.
    /// </para>
    /// </remarks>
    public interface IValidationErrorsRenderer
    {
        /// <summary>
        /// Renders validation errors using specified <see
cref="HtmlTextWriter"/>.
        /// </summary>
        /// <param name="page">Web form instance.</param>

        /// <param name="writer">An HTML writer to use.</param>
        /// <param name="errors">The list of validation errors.</param>
        void RenderErrors(Page page, HtmlTextWriter writer, IList
errors);
    }
}
```

11.7.1.1. 配置错误显示类

错误显示机制最棒的功能就是允许在配置模板（按：就是一个抽象对象定义）中轻松的为<spring:validationSummary>和<spring:validationError>控件更换显示对象。

```
<!-- Validation errors renderer configuration -->

<object id="Spring.Web.Anthem.UI.Controls.ValidationError"
abstract="true">
  <property name="Renderer">
    <object type="Spring.Web.Validation.IconValidationErrorsRenderer,
Spring.Web">
      <property name="IconSrc" value="validation-error.gif"/>
    </object>
  </property>
</object>

<object id="Spring.Web.Anthem.UI.Controls.ValidationSummary"
abstract="true">
  <property name="Renderer">
    <object type="Spring.Web.Validation.DivValidationErrorsRenderer,
Spring.Web">
```

```
      <property name="CssClass" value="validationError"/>
    </object>
  </property>
</object>
```

就这么简单!

第十二章. 使用 Spring.NET 进行面向方面的编程

12.1. 简介

AOP（*Aspect-Oriented Programming*）是对 OOP 的一种补充，它从一个不同于 OOP 的角度来看待程序的结构：OOP 将应用程序分解为一系列表现为继承关系的对象；AOP 则把程序分解为一系列方面（*aspects*）或者关注点（*concerns*）。AOP 将诸如事务管理等本来横向分布在多个对象中的关注点进行了模块化处理（这些关注点也常称为横切（*crosscutting*）关注点）。

AOP 框架是 Spring.NET 的一个关键组件。Spring.NET 的 IoC 容器与 AOP 框架是相互独立的，两者完全可以不依赖对方而单独使用，但是 AOP 做为一个强大的中间件解决方案，完善了 IoC 容器的功能。

在 Spring.NET 中，AOP 可用于：

- 提供声明式（*declarative*）企业服务，特别是做为 COM+声明式服务的替代品。其中最重要的是建立在 Spring.NET 事务抽象基础上的声明式事务管理服务。该功能计划在未来版本中推出。
- 允许用户实现自定义方面，用 AOP 来增强 OOP。

所以，可以将 AOP 框架看作是一种允许在不使用 COM+的情况下提供事务管理的技术，也可以利用其强大的功能来实现自定义方面。

如果急于了解 Spring AOP 框架的用法，可以先阅读第二十六章, AOP 指南。

12.1.1. AOP 基本概念

首先我们来了解 AOP 中的一些基本概念。这些概念都是 AOP 的通用术语，并非 Spring.NET 所特有。很遗憾 AOP 的术语不是特别的直观。但如果让 Spring.NET 来定义自己的专用名词，可能会更加教人糊涂。

- 方面（*Aspect*）：对横向分布在多个对象中的关注点所做的模块化。在企业应用中，事务管理就是一个典型的横切关注点。Spring.NET 将方面实

现为 Advisor 或拦截器（*interceptor*）。（按：Advisor 是通知和切入点的组合，拦截器实际就是指通知，注意在本文档中，一般会把环绕通知称为拦截器，而将其它类型的通知称为通知，这是因为环绕通知实现的是 AopAlliance.Intercept.IMethodInterceptor 接口，而其它通知类型实现的都是 Spring.Aop 命名空间下的通知接口。）

- 连接点（*Joinpoint*）：程序执行过程中的一个点，例如对某个方法的调用或者某个特定异常的抛出都可以称为连接点。
- 通知（*Advice*）：AOP 框架在某个连接点所采取的行为。通知有多种类型，包括“环绕”通知，“前置”通知和“异常”通知等，后文将对通知类型进行讨论。包括 Spring.NET 在内的很多 AOP 框架都把通知建模为拦截器（*interceptor*），并且会维护一个"包围"在连接点周围的拦截器链。
- 切入点（*Pointcut*）：指通知的应用条件，用于确定某个通知要被应用到哪些连接点上。AOP 框架应允许让开发人员指定切入点，例如，可以使用正则表达式来指定一个切入点。
- 引入（*Introduction*）：向目标对象添加方法或字段的行为。Spring.NET 允许为任何目标对象引入新的接口。例如，可以利用引入让任何对象在运行期实现 IAuditable 接口，以简化对象状态变化的跟踪过程。（按：也称为 mixin，混入）
- 目标对象（*Target object*）：指包含连接点的对象。也称为被通知或被代理对象。（按：“被通知对象”实际是“被应用了通知的对象”，在译文中，将 advised object 或 proxied object 统称为目标对象，这样更为统一）
- AOP 代理（*AOP proxy*）：由 AOP 框架在将通知应用于目标对象后创建的对象。在 Spring.NET 中，AOP 代理是使用 IL 代码在运行时创建的动态代理。
- 织入（*Weaving*）：将方面进行组装，以创建一个目标对象。织入可以在编译期完成（例如使用 Gripper_Loom.NET 编译器），也可以在运行时完成。Spring.NET 在运行时执行织入。

各种通知类型包括：

- 环绕通知（Around Advise）：包围（按：即在连接点执行的前、后执行）某个连接点（如方法调用）的通知。这是功能最强大的一种通知。环绕通知允许在方法调用的前后执行自定义行为。它可以决定是让连接点继续执行，还是用自己的返回值或异常来将连接点“短路”。
- 前置通知（Before Advise）：在某个连接点执行之前执行，但是不具备阻止连接点继续执行的能力（除非它抛出异常）。
- 异常通知（Throws Advise）：当方法（连接点）抛出异常时执行。Spring.NET 的异常通知是强类型的（按：Spring.NET 用标识接口来定义异常通知，异常通知的处理方法仅需遵循一定的命名规则，可以用具体的异常类型声明其参数，参见 12.3.2.3 节），所以，可以在代码中直接捕捉某个类型的异常（及其子类异常），不必从 Exception 转型。

- 后置通知（After returning Advise）：在连接点正常执行完成后执行，例如，如果方法正常返回，没有抛出异常时，后置通知就会被执行。

Spring.NET 内置了以上所有类型的通知。在应用时，应尽量使用功能最少（只要对要实现的行为来说是足够的）的通知类型，这样可简化编程模型并减少出错的可能。例如，如果只需使用某个方法的返回值来更新缓存，那么用后置通知就比环绕通知合适。因为，尽管环绕通知可以完成同样的功能，但在后置通知中不用象环绕通知那样必须调用 IMethodInvocation 接口的 Proceed()方法来允许连接点继续执行，所以连接点总是能正常执行。（按：换句话说，因为环绕通知可以控制连接点的继续执行，所以如果没有调用 IMethodInvocation 接口的 Proceed()方法，连接点就会被“短路”；而使用后置通知就不存在这个问题）。

切入点是 AOP 的关键概念，使 AOP 从根本上区别于旧的拦截技术。切入点使通知可以独立于 OO 的继承层次之外。例如，一个声明式事务处理的环绕通知可以应用于不同对象的方法。切入点是 AOP 的结构性要素。

12.1.2. Spring.NET AOP 的功能

Spring.NET 的 AOP 框架完全用 C#实现。所有的织入工作都在运行时完成，不需要特殊的编译过程。由于在不必控制或修改程序集装载的方式，也不依赖于非托管 API，所以 Spring.NET 的 AOP 框架适用于任何 CLR 环境。

目前，Spring.NET 支持对方法调用的拦截。虽然也可以在不破坏核心 API 的前提下加入对字段拦截的支持，但 Spring.NET 没有实现这一功能。这源于一个有争议的话题：字段拦截破坏了 OO 的封装性。在应用开发中这并不是明智之举。

Spring.NET 为切入点和不同的通知类型都提供了相应的类。在 Spring.NET 的类库中，**方面由 Advisor 对象来表示，而 Advisor 又由通知和切入点组成**（切入点用于确定将通知应用在哪些连接点上）。

各种类型的通知分别是由 IMethodInterceptor 接口（由 AOP 联盟定义的拦截器 API，在 AopAlliance.Intercept 命名空间下）和 Spring.Aop 命名空间下的各个通知接口定义的。所有通知都必须（最终）实现 AopAlliance.Aop.IAdvice 接口（这是个标识接口，没有定义任何成员），Spring.Aop 命名空间下的 IMethodInterceptor、IThrowsAdvice、IBeforeAdvice 和 IAfterReturningAdvice 接口都扩展了 IAdvice 接口。我们将在后文中讨论它们。

Spring.NET 将 AOP 联盟定义的 java 接口移植到了.NET 平台上。环绕通知必须实现 AOP 联盟的 AopAlliance.Intercept.IMethodInterceptor 接口。在 Java 领域，AOP 联盟的接口已经得到了广泛的应用，但在.NET 领域，Spring.NET 是目前唯一实现了这些接口的 AOP 框架。就短期来说，这为同时使用.NET 和 Java 的开发人员提供了一个统一的编程模型；就长期而言，我们也希望看到更多的.NET 项目能采用 AOP 联盟的接口。

Spring.NET 的 AOP 框架并不是要象 AspectJ（Aspect#）一样全面支持 AOP。但 Spring.NET 的 AOP 框架可以为.NET 应用领域中许多应该用 AOP 来处理的问题提供很好的解决方案。

一般我们会将 Spring.NET 的 AOP 框架与 IoC 容器一起使用。在 IoC 容器中，可以象普通对象一样来布署 AOP 通知(也可以使用强大的自动代理(autoproxying)功能）；通知和切入点都可以由 Spring.NET 的 IoC 容器管理。

12.1.3. Spring.NET 的 AOP 代理

Spring.NET 利用 System.Reflection.Emit 命名空间下的类在运行时动态创建 IL 代码来生成 AOP 代理。这使得代理（的创建）非常高效，并且不受任何继承层次的限制。

在.NET 中，实现 AOP 代理的另一种方法是使用 ContextBoundObject 类和 Remoting 基础框架。但这个方法不是太好，因为它要求被代理的类必须直接或间接继承自 ContextBoundObject。这个不必要的限制会影响到对象模型的设计，也无法将 AOP 应用在设计者无法直接控制的“第三方”类型上。同时，由于上下文切换和 Remoting 基础框架的开销，所生成的代理也会比 IL 直接生成的代理慢得多。

Spring.NET 的 AOP 代理是非常“智能”的。在生成代理时，由于代理的配置是能确定的，所以生成的代理对象仅会在必要时才通过反射调用目标方法（即，当已有通知应用到目标方法时）。其他情况下，目标方法是直接调用的，避免了因反射调用而带来的性能开销。

最后，Spring.NET 的 AOP 代理永远不直接返回目标对象的原始引用。如果检测到目标对象的某个方法返回了自身的引用（比如用 return this; ），AOP 代理会将返回值替换为代理自身的引用。

目前，AOP 代理生成器使用对象的组合来转发从代理到目标对象的调用，这和传统 Decorator 模式的实现方式很相似。这表示被代理的类必须实现一或多个接口，让类实现接口并不是限制，而是在任何时候都应该遵守的最佳编程方式。同时，这样也会比让类继承 ContextBoundObject 更少侵入性，

未来的版本会使用继承来实现代理，这样就可以代理没有实现任何接口的类，同时也可以消除一些用组合方式无法解决的遗留问题。

12.2. Spring.NET 中的切入点

我们来看一下 Spring.NET 是如何处理切入点这一关键概念的。

12.2.1. 概念

在 Spring.NET 中，切入点与通知相互独立，切入点可以不受通知的限制而单独重用。一个切入点可以关联不同的通知。

Spring.Aop.IPointcut 是切入点的核心接口，用来表示通知的应用条件。该接口的声明为：

```
public interface IPointcut
{
    ITypeFilter TypeFilter { get; }

    IMethodMatcher MethodMatcher { get; }
}
```

IPointcut 内部其实由两个接口组成，这两个接口分别为切入点进行类型过滤和方法匹配。拆分成这两部分有助于类型过滤器和方法匹配器的重用，且方便对它们进行细粒度的组合操作(例如同另一个切入点的方法匹配器进行一个“并”操作）。

ITypeFilter 接口将切入点定位在一组指定的类型上。如果 Matches()方法始终返回 true，则所有的类型都可以被匹配：

```
public interface ITypeFilter
{
    bool Matches(Type type);
}
```

IMethodMatcher 接口更为重要。（按：用于确定要应用通知的目标方法）该接口声明为：

```
public interface IMethodMatcher
{
    bool IsRuntime { get; }

    bool Matches(MethodInfo method, Type targetType);

    bool Matches(MethodInfo method, Type targetType, object[] args);
}
```

（按：IMethodMatcher 的匹配方式可以是静态的，也可以是动态的，这取决于其 IsRuntime 属性的值。静态匹配只检查方法的签名及应用于此方法的特性是否满足条件。动态匹配则会在每次方法调用时检查实际传入的参数值。见参考文档。另外，请注意 IMethodMathcher 接口的 Matches 方法有两个重载。）

如果实现类的 IsRuntime 属性为 false，那么匹配就是静态执行的。静态匹配时，Matches(MethodInfo,Type)方法仅在 AOP 代理被 Spring.NET 创建时调用*一次*，只要某个方法的签名满足要求，就可以匹配，而不会去管它在调用时使用了什么参数。如果 IMethodMatcher 实现类是静态匹配的，AOP 框架不会去调用重载的 Matches(MethodInfo, Type, object[])。大多数 IMethodMatcher 的实现类都是静态匹配的。

如果实现类的 IsRuntime 属性值为 true，那么匹配就是动态的。动态匹配时，如果 Matches(MethodInfo,Type)对某个方法返回 true，那么，AOP 框架会在每次调用该方法前去调用重载的 Matches(MethodInfo, Type, object[])，以决定是否应该将通知应用到此次方法调用上。所以，动态匹配允许切入点在通知执行前检查传递给目标方法的参数值。

应该尽量将切入点实现为静态的（按：即是静态匹配的），因为在创建 AOP 代理时，AOP 框架会将静态切入点的匹配结果缓存起来。

12.2.2. 切入点的操作

Spring.NET 支持切入点的并（*union*）和交（*intersection*）操作。

并操作指只要一个切入点匹配就匹配。

交操作指仅当两个切入点都匹配时才匹配。

通常，并操作的应用更为广泛。

可以使用 Spring.Aop.Support.Pointcuts 类的 Union()静态方法，或 ComposablePointcut 类的 Union()实例方法对切入点进行组合。

12.2.3. Spring.NET 提供的切入点实现类

Spring.NET 提供了几个 IPointcut 的实现类。其中有些可以直接使用；有些需要在应用时子类化。

12.2.3.1. 静态切入点

静态切入点是面向目标类型和方法的，不会考虑方法调用时的实际参数值。静态切入点的效率很高，是多数情况下的最佳选择。Spring.NET 可以仅在方法第一次调用时对静态切入点作匹配，随后将匹配的结果缓存起来，下次该方法调用时就会直接参考第一次的匹配结果。

下面我们来讨论 Spring.NET 中实现的静态切入点。

12.2.3.1.1. 正则表达式切入点

正则表达式是指定切入点的常用手段，包括 Spring.NET 在内的很多 AOP 框架都支持用正则表达式指定静态切入点。
Spring.Aop.Support.SdkRegularExpressionMethodPointcut 类是通用的正则表达式切入点，该类在实现时借助了.NET 基础类库中的正则表达式类型。

使用该类时，可以定义一个模式字符串列表。列表中任一项匹配都认为切入点是匹配的（也就是说匹配结果是这些模式列表项的并集）。

该类的对象定义如下所示：

```
<object id="settersAndAbsquatulatePointcut"
    type="Spring.Aop.Support.SdkRegularExpressionMethodPointcut,
Spring.Aop">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</object>
```

为方便起见，Spring.NET 实现了一个 RegularExpressionMethodPointcutAdvisor 类（按：注意我们开始使用 Advisor 了），可以在引用 IAdvice 实例的同时定义正则表达式切入点的规则（注意 IAdvice 实例可以是拦截器、前置通知或异常通知等等）。这可以简化配置工作，因为 Advisor 的对象定义中已经包含了一个切入点的定义（按：Advisor 其实包含两组信息，即通知和切入点，其中切入点由 Advisor 本身定义，通知则可引用单独定义的 IAdvice 对象。所以说 Advisor 是通知和切入点的组合），如下：

```
<object id="settersAndAbsquatulateAdvisor"
    type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor,
Spring.Aop">
    <property name="advice">
        <ref local="objectNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</object>
```

RegularExpressionMethodPointcutAdvisor 类可以和任何通知类型一起使用。

如果只需要定义一个模式字符串，可以使用 Pattern 属性，不必用 Patterns 属性定义模式列表。

12.2.3.1.2. 特性切入点

AOP 框架可以通过匹配与某个方法相关联的特性来指定切入点。与切入点关联的通知可以自行读取该特性的元数据来配置自己。特性切入点由 AttributeMatchMethodPointcut 类定义。下面的切入点会匹配所有应用了 Spring.Attributes.CacheAttribute 特性的方法：

```
<object id="cachePointcut"
type="Spring.Aop.Support.AttributeMatchMethodPointcut, Spring.Aop">
    <property name="Attribute" value="Spring.Attributes.CacheAttribute,
Spring.Core"/>
</object>
```

或者使用 DefaultPointcutAdvisor 类，如下所示：

```
<object id="cacheAspect"
type="Spring.Aop.Support.DefaultPointcutAdvisor, Spring.Aop">
  <property name="Pointcut">
      <object type="Spring.Aop.Support.AttributeMatchMethodPointcut,
Spring.Aop">
          <property name="Attribute"
value="Spring.Attributes.CacheAttribute, Spring.Core"/>
      </object>
  </property>
  <property name="Advice" ref="aspNetCacheAdvice"/>
</object>
```

其中 aspNetCacheAdvice 是 IMethodInterceptor 接口的一个实现类（按：指 CacheAdvice 类）的实例，用于缓存方法的返回值。关于这个特殊的通知，可参考 SDK 中关于 Spring.Aop.Advice.CacheAdvice 类的内容。

为方便起见，可使用 AttributeMatchMethodPointcutAdvisor 类来定义基于特性的 Advisor，这比使用 DefaultPointcutAdvisor 类的对象定义要简短。如下所示：

```
<object id="AspNetCacheAdvice"
type="Spring.Aop.Support.AttributeMatchMethodPointcutAdvisor,
Spring.Aop">
   <property name="advice">
      <object type="Aspect.AspNetCacheAdvice, Aspect"/>
```

```
    </property>
    <property name="attribute" value="Framework.AspNetCacheAttribute,
Framework" />
</object>
```

12.2.3.2. 动态切入点

动态切入点在匹配时的开销要比静态切入点大。动态切入点除了会检查静态信息，还会检查方法的参数。也就是说，对于每次方法调用，都会去进行匹配，匹配的结果不进行缓存，因为每次调用的参数值是可变的，缓存没有意义。

最主要的动态切入点是控制流程切入点。

12.2.3.2.1. 控制流程切入点

Spring.NET 的控制流程切入点与 AspectJ 的 cflow 切入点在概念上很相似，只是功能较少（目前尚不能在一个切入点内部指定另一个切入点）。控制流程切入点是动态的，它会为每次方法调用检查当前的调用堆栈。假如有一个方法 ClassA.A()，其中调用了 ClassB.B()，那么 ClassB.B()的执行就发生在 ClassA.A()的控制流程之内。控制流程切入点就是要把某个通知应用到 ClassA.A()方法上，但只有当 ClassA.A()中的 ClassB.B()在单独的调用堆栈中执行时，该通知才有效，而 ClassA.A()中 ClassB.B()之外的代码运行时，该通知就不会被执行。控制流程切入点可用 Spring.Aop.Support.ControlFlowPointcut 类来定义。

> **注意**
>
> 控制流程切入点在运行时的开销要比其他类型的动态切入点大很多。

在使用控制流程切入点时，要注意 JIT 可能会将方法进行内联编译。JIT 这么做是为了提高性能，但是在调用堆栈中我们会找不到被内联的方法。因为内联编译会将内联方法的 IL 代码插入到调用方法的 IL 代码中，就如同内联方法不存在一般。由于 ControlFlowPointcut 是通过 System.Diagnostics.StackTrace 类跟踪堆栈信息来匹配方法调用的，所以无法匹配内联方法。

一般来说，如果一个方法的代码很少（IL 代码少于 32 字节），就可能被内联。若对此感兴趣可参阅 David Notario 的 blog(JIT Optimizations I 和 JIT Optimizations II)。另外，在使用 Release 选项进行编译时，程序集的元数据会通知 CLR 启用 JIT 优化。当使用 Debug 选择编译时 CLR 会关闭（某些？）优化选项。凭经验来说，Debug 编译会关闭方法内联。

为保证控制流程切入点不会因方法内联而无效，可以为方法应用 System.Runtime.CompilerServices.MethodImplAttribute 特性，并使用 MethodImplOptions.NoInlining 作为该特性的参数。如下面的简单例子所示，

如果使用 release 模式编译，这个方法就和方法“GetAge“的控制流程切入点不匹配。

```
public int GetAge(IPerson person)
{
    return person.GetAge();
}
```

应用以下特性后，即可防止方法被内联：

```
[MethodImpl(MethodImplOptions.NoInlining)]
public int GetAge(IPerson person)
{
    return person.GetAge();
}
```

12.2.4. 自定义切入点

在 Spring.NET 中，切入点是.NET 类型，而不像 AspectJ 一样是语言特性，所以开发人员可以创建自定义切入点。自定义切入点可以是静态的，也可以是动态的。目前 Spring.NET 虽不支持复杂的切入点语句（AspectJ 中可用代码做到这点），但自定义切入点本身可以像任意对象一样做得相当复杂。

Spring.NET 提供了一系列基类型来协助开发人员实现自定义切入点。

例如，静态切入点是最常用的切入点类型，开发人员可通过继承 StaticMethodMatcherPointcut 来实现自定义静态切入点。只需实现其中的一个抽象方法即可：

```
public class TestStaticPointcut : StaticMethodMatcherPointcut {
    public override bool Matches(MethodInfo method, Type targetType) {
        // return true if custom criteria match
    }
}
```

12.3. Spring.NET 的通知类型

下面，我们来看 Spring.NET 的 AOP 框架是如何处理通知的。

12.3.1. 通知的生命周期

Spring.NET 的通知既可由某个类的所有对象共享，也可由该类型的单个实例独占。共享的通知称为基于类型（per-class）的通知，而独占的通知称为基于实例（per-instance）的通知。

基于类型的通知最为常用。很多常用功能很适合用基于类型的通知实现，比如说事务。它们不依赖于目标对象的状态，也不会向目标对象添加新状态，仅仅对方法及其参数进行操作。

基于实例的通知比较适合做引入（introductions）。此时通知可以向目标对象添加状态。在 AOP 代理中，可以同时使用基于类型和基于实例的通知。

12.3.2. 通知类型

目前 Spring.NET 支持各种标准的通知类型，并能通过扩展添加新的通知类型。我们先来看一下通知的基本概念和标准的通知类型。

12.3.2.1. 拦截环绕通知

Spring.NET 中最基本的通知类型是拦截环绕通知（*interception around advice*），即方法拦截器。

Spring.NET 用 IMethodInterceptor 扩展了 AOP 联盟的拦截器接口。IMethodInterceptor 是环绕通知的父接口，定义如下：

```
public interface IMethodInterceptor : IInterceptor
{
    object Invoke(IMethodInvocation invocation);
}
```

在 Invoke 方法中，IMethodInvocation 类型的参数包含了目标方法及其参数等信息。Invoke 方法应该返回调用的结果，也就是连接点（目标方法）的返回值。

下面是 IMethodInterceptor 的一个简单实现类：

```
public class DebugInterceptor : IMethodInterceptor {
    public object Invoke(IMethodInvocation invocation) {
        Console.WriteLine("Before: invocation=[{0}]", invocation);
        object rval = invocation.Proceed();
        Console.WriteLine("Invocation returned");
        return rval;
```

```
    }
}
```

注意其中 IMethodInvocation.Proceed()方法的调用。该方法会依次调用拦截器链上的其它拦截器。大部分拦截器都需要调用这个方法并返回它的返回值。当然，也可以不调用 Proceed 方法，而返回一个其它值或抛出一个异常，但一般不太会这么做。

12.3.2.2. 前置通知

前置通知是一种相对简单的通知类型。前置通知只在进入方法前执行，不需要了解目标方法的具体信息，所以也不需要使用 IMethodInvocation 对象。

使用前置通知最大的好处是不用调用 Proceed()方法，也不存在因为忘记调用该方法而导致错误的可能。

IMethodBeforeAdvice 接口如下所示：

```
public interface IMethodBeforeAdvice : IBeforeAdvice
{
    void Before(MethodInfo method, object[] args, object target);
}
```

注意 Before 方法的返回值是 void。前置通知可以在连接点执行前插入自定义行为，但不会改变连接点的返回值。如果前置通知抛出异常，就会中止拦截器链的继续执行。异常会沿着拦截器链传递。

下面的例子是一个前置通知，用于统计正常返回的方法个数：

```
public class CountingBeforeAdvice : IMethodBeforeAdvice {

    private int count;

    public void Before(MethodInfo method, object[] args, object target)
{
        ++count;
    }

    public int Count  {
        get { return count; }
    }
}
```

前置通知可以用于任何切入点。

12.3.2.3. 异常通知

异常通知在连接点抛出异常时执行。Spring.Aop.IthrowsAdvice 接口没有定义任何方法：它是一个标识接口（按：之所以用标识接口，原因有二：1、在通知方法中，只有最后一个参数是必须的。如果声明为接口的方法，参数列表就被固定了。2、如果第一个原因可以用重载的接口方法解决，那么这个原因就是使用标识接口的充分原因了：实现此接口的类必须声明一或多个通知方法，接口方法做不到这一点），用以表明实现它的类声明了一或多个强类型的异常通知方法。这些异常通知方法必须是以下格式：

```
AfterThrowing([MethodInfo method, Object[] args, Object target],
Exception subclass)
```

异常通知方法必须以“AfterThrowing“为名。返回值会被 AOP 框架忽略，所以一般只要返回 void。至于方法的参数，要么是一个，要么是四个，取决于通知方法是否对目标对象、被调用方法及其参数感兴趣。注意，最后一个参数是必须的。

下面是一个通知方法的例子，这个通知会在抛出 RemotingException 异常（及其子类）的时候调用：

```
public class RemoteThrowsAdvice : IThrowsAdvice {
    public void AfterThrowing(RemotingException ex) {
        // Do something with remoting exception
    }
}
```

下面的通知在抛出 SqlException 异常时调用。和上面的例子不同，这个通知方法声明了四个参数，因为它要访问目标对象、被调用方法及其参数：

```
public class SqlExceptionThrowsAdviceWithArguments : IThrowsAdvice {
    public void AfterThrowing(MethodInfo method, object[] args, object
target, SqlException ex) {
        // Do something will all arguments
    }
}
```

下面这个异常通知声明了两个通知方法，分别用于处理 RemotingException 和 SqlException 异常。异常通知中可以声明任意多的异常通知方法：

```
public class CombinedThrowsAdvice : IThrowsAdvice {

    public void AfterThrowing(RemotingException ex)  {
        // Do something with remoting exception
```

```
    }

    public void AfterThrowing(MethodInfo method, object[] args, object
target, SqlException ex) {
        // Do something will all arguments
    }
}
```

最后，很有必要说一下，只有当被抛出的异常和异常通知能处理的类型完全一致时，异常通知才会执行。这是什么意思呢？也就是说，如果定义了一个用来处理 RemotingException 异常的异常通知，那么 AfterThrowing 方法就只在被抛出的异常类型确实是 RemotingException 时才调用。如果某个 RemotingExcetion 异常被抛出后随即被包装为另一个异常的内部异常，没能被异常通知拦截器截获，那么这个异常通知就不会被触发。举例来说，假设该异常通知的目标对象是一个业务对象，在调用它的方法时抛出了一个 RemotingException 异常，这个异常被立即包装进了一个与业务相关的 BadConnectionExcetion 异常中（参见下面的代码），那么异常通知就永远不会响应 RemotingException 异常，因为所有的异常通知都只能看到 BadConnectionException 异常。它们根本不理会 BadConnectionException 内部是不是还包装了一个 RemotingException 异常。

```
public void BusinessMethod()
    {
        try
        {
            // do some business operation...
        }
        catch (RemotingException ex)
        {
            throw new BadConnectionException("Couldn't connect.", ex);
        }
    }
```

注意异常通知可用于任意切入点。

12.3.2.4. 后置通知

在 Spring.NET 中，后置通知必须实现 Spring.Aop.IAfterReturningAdvice 接口，该接口如下所示：

```
public interface IAfterReturningAdvice : IAdvice
{
  void AfterReturning(object returnValue, MethodBase method, object[]
args, object target);
}
```

从接口的方法定义可以看出，后置通知会访问目标对象、目标方法及其参数与返回值（但不能修改返回值）。

下面的后置通知用于统计所有未抛出异常的（成功的）方法调用：

```
public class CountingAfterReturningAdvice : IAfterReturningAdvice {
    private int count;

    public void AfterReturning(object returnValue, MethodBase m, object[]
args, object target) {
        ++count;
    }

    public int Count  {
        get { return count; }
    }
}
```

后置通知对切入点的执行没有影响，如果通知抛出异常，就会沿拦截器链向上抛出，从而中断拦截器链的继续执行。

注意

后置通知可以应用于任何切入点。

12.3.2.5. 引入通知

Spring.NET 允许开发人员向目标类型添加新的方法和属性。一般来说，如果我们想要添加的功能是一个横切关注点，并且希望用此功能来改变对象从继承层次中获得的静态结构时，比较适合使用引入通知。例如，我们可能希望在运行时将对象转型为某个其它类型的接口。引入也可以作为一种模拟多继承的方法。

引入通知可以用一个扩展了标识接口 IAdvice 的普通接口来定义。

注意

将来可能不再需要扩展该标识接口。

下面看一个例子，接口 IAuditable 扩展了 IAdvice，可以用它来定义一个引入通知，用以记录对象状态的上次修改时间。

```
public interface IAuditable : IAdvice
{
    DateTime LastModifiedDate
    {
```

```
        get;
        set;
    }
}
```

标识接口 IAdvice 的定义如下：

```
public interface IAdvice
{
}
```

为了访问目标对象，可以让 IAuditable 扩展 ITargetAware 接口。

```
public interface ITargetAware
{
  IAopProxy TargetProxy
  {
    set;
  }
}
```

其中的 IAopProxy 接口提供了一个间接层，通过它可以间接地访问目标对象。

```
public interface IAopProxy
{
  object GetProxy();
}
```

IAuditable 的完整定义如下：

```
public interface IAuditable : IAdvice, ITargetAware
{
    DateTime LastModifiedDate
    {
        get;
        set;
    }
}
```

下面的 AuditableMixin 类则实现了 IAuditable 接口：

```
public class AuditableMixin : IAuditable
{
    private DateTime date;
    private IAopProxy targetProxy;
```

```
    public AuditableMixin()
    {
      date = new DateTime();
    }

    public DateTime LastModifiedDate
    {
       get { return date; }
       set { date = value; }
    }

    public IAopProxy TargetProxy
    {
       set { targetProxy = value; }
    }
}
```

由于引入通知是应用在类型级而非方法级的，所以不需要关联切入点。引入通知可以用专门的 IAdvisor 子接口来指定目标类型，这个接口就是 IIntroductionAdvisor：

```
public interface IIntroductionAdvisor : IAdvisor
{
    ITypeFilter TypeFilter { get; }

    Type[] Interfaces { get; }

    void ValidateInterfaces();
}
```

其中：TypeFilter 属性返回一个过滤器，以确定引入通知要应用的目标类型。

Interfaces 属性则返回该 Advisor 要引入的接口。

ValidateInterfaces()方法由内部使用，用于检查引入通知能否实现要引入的接口。

该接口在 Spring.NET 中有一个默认的实现类(DefaultIntroductionAdvisor 类)，应该能够满足大多数要求。如果需要自定义引入通知的 Advisor，最简单的方法是继承 DefaultIntroductionAdivsor 类，并创建一个目标类型的新实例传给基类构造器。这一点很重要，因为我们需要针对每个目标对象都返回一个应用了引入通知的实例。

```
public class AuditableAdvisor : DefaultIntroductionAdvisor
```

```
{
  public AuditableAdvisor() : base(new AuditableMixin())
  {
  }
}
```

可以在自定义 Advisor 中用其它构造器来显式的指定要引入的接口。请参见 SDK 文档。

在代码中,可以通过编程方式用 IAdvised 接口的 AddIntroduction 方法来使用引入通知的 Advisor，也可以在 IoC 容器中通过配置 ProxyFactoryObject 对象的 IntroductionNames 属性来使用，Spring.NET 推荐使用 IoC 容器，稍后会介绍 ProxyFactoryObject 类。

与 Java 版 Spring 不同，Spring.NET 没有将引入通知实现为特殊的拦截器。这样的好处是引入通知不会被保存进拦截器链，所以可以带来显著的性能提升。不管是由目标对象本身定义的方法，还是被引入通知添加的新方法，只要该方法没有应用拦截器，都会直接去调用它，而不需要通过反射。也就是说被引入的方法完全可以看做是目标对象本身的方法。所以我们能为细粒度的对象添加引入通知。引入通知的缺点是被引入的方法无法访问调用堆栈。将来会解决这个问题。

12.4. Spring.NET 中的 Advisor

在 Spring.NET 中,Advisor 是对方面进行模块化处理后抽象出来的概念。Advisor 一般是通知和切入点的组合。

除了比较特殊的引入通知，Advisor 可以和任意类型的通知一起工作。Spring.Aop.Support.DefaultPointcutAdvisor 类是最常用的一个 Advisor 实现类。例如，它可以同 IMethodInterceptor、IBeforeAdvice 或 IThrowsAdvice 通知以及任意切入点协作。

其它的 Advisor 实现类包括：在 12.2.3.1.2,特性切入点中讲过的包含特性切入点的 AttributeMatchMethodPointcutAdvisor，以及包含正则表达式切入点的 RegularExpressionMethodPointcutAdvisor。

在 AOP 代理中,既可以使用 Advisor 类,也可以直接使用各种类型的通知。例如,可以在代理对象的配置中引用拦截环绕通知、异常通知和前置通知：Spring.NET 会自动创建必要的拦截器链。

12.5. 使用 ProxyFactoryObject 创建 AOP 代理

如果我们用 Spring.NET 的 IoC 容器来管理业务对象,那么,可以使用 Spring.NET 的 ProxyFactoryObject 类，该类实现了 IFactoryObject 接口，专门用于创建

AOP 代理（还记得吗，IFactoryObject 如同一个间接层，可以创建不同类型的对象，参见 4.3.5,引用其他对象或类型的成员）。

在 Spring.NET 中，创建 AOP 代理最基本的方法就是使用 Spring.Aop.Framework.ProxyFactoryObject 类。该类可以控制切入点和通知的所有应用细节。不过，如果不关心应用细节，也有更加简单的方式。

12.5.1. 基本原理

与 Spring.NET 中其它 IFacotryObject 实现类一样，ProxyFactoryObject 引入了一个间接层。如果以“foo“为名定义了一个 ProxyFactoryObject，那么通过“foo“从容器中请求的对象 不是 ProxyFactoryObject 实例的本身，而是其 GetObject()方法返回的对象，而这个对象就是包装了目标对象的 AOP 代理对象。

使用 ProxyFactoryObject（或其它能与 IoC 容器协作的 AOP 代理创建类型）最大的好处，是可以将通知和切入点也交给 IoC 容器管理。这使得 Spring.NET 的 AOP 框架可以实现其它 AOP 框架不具备的功能。例如，通知对象可以自己引用一个应用对象（除目标对象之外的应用对象，通知引用目标对象是所有 AOP 框架都可以做到的），这就能充分利用依赖注入所提供的灵活性。

12.5.2. ProxyFactoryObject 的属性

在容器中，也可以设置 ProxyFactoryObject 对象定义的属性。它的属性可用于：

- 指定要代理的目标对象
- 指定应用于目标对象的通知

其中一部分重要属性继承自 Spring.Aop.Framework.ProxyConfig 类，该类是所有 AOP 代理工厂的基类。这些属性包括：

- ProxyTargetType：布尔类型，如果目标类是被直接代理的，该属性为 true；反之则只代理目标类所实现的接口（比如代理某个由接口定义的方法）。（按：目前默认的方式是代理接口）
- Optimize：是否使用强制优化（aggressive optimization）来创建代理。除非很清楚相关的 AOP 代理是如何处理优化的，否则不要将其设为 true。这个属性的确切含义会随代理的实现方式不同而不尽相同，并且，强制优化通常是对代理的创建时间和运行性能的折衷。优化可能被某些代理实现类所忽略，也可能因其它属性的某些值而被禁用（在后台被禁用，不会有任何形式的通知），如 ExposeProxy 属性等。
- IsFrozen：代理工厂配置完成后是否允许改变通知。默认值为 false。
- ExposeProxy：是否通过 AopContext 暴露当前代理，以便目标对象能够访问自己的代理对象。（若不使用 AopContext 则可通过 IMethodInvocation 接口获得当前代理的引用）如果目标对象需要引用代理，并且

ExposeProxy 属性的值为 true，那么目标对象就可通过 AopContext.CurrentProxy 属性获取当前代理的引用。

- AopProxyFactory：在生成代理时要使用的 IAopProxyFactory 实现类。该属性允许我们选择 AOP 代理对象的创建策略：是使用远程代理还是动态生成 IL，或是其它的方式，默认的实现类会使用动态生成 IL 的方式来创建基于对象组合的代理。

其它属性包括：

- ProxyInterfaces：一个字符串数组，用于保存要代理的接口名。（按：即目标类所实现的某个或某些接口。SDK 参考文档上该属性的类型是 void，是错的）
- InterceptorNames：一个字符串数组，保存要应用的 IAdvisor、拦截器或其它通知的名称。其中的顺序很重要，排在前面的会先处理（按：使用编程方式时，”排在前面“是指后使用 AddAdvice 方法加入的拦截器）。列表中第一个拦截器会第一个拦截目标调用（当然如果它是 MethodInterceptor 或 BeforeAdvice 时）。名称所指向的对象必须定义在当前容器或父容器中。这里不能直接引用对象定义，否则 IsSingleton 属性就没有意义了。
- IntroductionNames：引入通知对象定义的名称列表。如果某个名称指向的对象没有实现 IIntroductionAdvisor 接口，AOP 框架就会创建一个 DefaultIntroductionAdvisor 的实例并将该名称分配给这个实例，这样，引入通知对象的所有方法都会被添加到目标对象中去。如果使用实现了 IIntroductionAdvisor 的类，就可以精确的控制要引入的接口。
- IsSingleton：表示工厂是否要返回唯一的代理对象，而不管 GetObject 方法被调用多少次（部分 IFactoryObject 的实现类提供了这个方法）。默认值是 true。如果要应用基于实例的通知，就要将该属性设为 false，且 IsFrozen 属性也应该为 false。如果需要使用有状态的通知——例如一个有状态的、prototype 模式的引入通知——就可以将该属性设为 false。

12.5.3.代理接口（按：即代理目标对象实现的接口）

下面用一个例子来看看 ProxyFactoryObject 的用法，容器中配置了：

- 一个名为“personTarget“的对象定义，作为要代理的目标对象。
- 一个 IAdvisor 对象定义和一个 IInterceptor 对象定义，用于提供通知。
- 一个 AOP 代理对象定义，定义中指定了目标对象（“personTarget”）和要代理的接口（按：指目标类所实现的某个接口），以及要应用的通知：

```
<object id="personTarget" type="MyCompany.MyApp.Person, MyCompany">
    <property name="name" value="Tony"/>
    <property name="age" value="51"/>
</object>
```

```
<object id="myCustomInterceptor"
type="MyCompany.MyApp.MyCustomInterceptor, MyCompany">
    <property name="customProperty" value="configuration string"/>
</object>

<object id="debugInterceptor" type="Spring.Aop.Advice.DebugAdvice,
Spring.Aop">
</object>

<object id="person" type="Spring.Aop.Framework.ProxyFactoryObject,
Spring.Aop">
    <property name="proxyInterfaces" value="MyCompany.MyApp.IPerson"/>
    <property name="target" ref="personTarget"/>
    <property name="interceptorNames">
        <list>
            <value>debugInterceptor</value>
            <value>myCustomInterceptor</value>
        </list>
    </property>
</object>
```

注意 InterceptorNames 属性是一个字符串列表，即当前容器中的拦截器或 Advisor 对象定义的名称列表。这里可以使用 Advisor、拦截器、前置/后置/异常通知对象的名称。名称的顺序很重要。

您可能很奇怪为什么不直接引用通知或 Advisor 对象。原因是，如果 ProxyFactoryObject 的 IsSingleton 属性被设为 false，就需要在每次调用 GetObject 方法时返回新的代理对象。若所使用的 Advisor 中有某一个本身是 prototype 模式，ProxyFactoryObject 就可以通过它的名称每次从容器中获取新的 Advisor 对象，这样以来，直接引用就不大合适了。

在使用时，通过“person”获取的对象可以转型为 IPerson（因为它是工厂对象，见上文），如下所示：

```
IPerson person = (IPerson) factory.GetObject("person");
```

同一容器中的对象也可以对 person 对象进行强类型依赖，就如同使用普通的 IPerson 对象一样：

```
<object id="personUser" type="MyCompany.MyApp.PersonUser, MyCompany">
    <property name="person" ref="person"/>
</object>
```

PersonUser 类包含一个 IPerson 类型的属性。也就是说，必要时，IPerson 对象的 AOP 代理可以直接作为一个“真正的”IPerson 对象来使用。同时，代理对象仍然是一个代理类型，可以将其转型为 IAdvised 接口（下文将讨论）。

可以在容器中用匿名的内联对象（*inline object*）来隐藏目标对象，如下所示。（关于内联对象的详细内容可参见 4.3.3.5,内联对象定义）与前例相比，只有 ProxyFactoryObject 对象的定义有所不同：

```
<object id="myCustomInterceptor"
type="MyCompany.MyApp.MyCustomInterceptor, MyCompany">
    <property name="customProperty" value="configuration string"/>
</object>

<object id="debugInterceptor" type="Spring.Aop.Advice.DebugAdvice,
Spring.Aop">
</object>

<object id="person" type="Spring.Aop.Framework.ProxyFactoryObject,
Spring.Aop">
    <property name="proxyInterfaces" value="MyCompany.MyApp.IPerson"/>
    <property name="target">
      <!-- Instead of using a reference to target, just use an inline
object -->
      <object type="MyCompany.MyApp.Person, MyCompany">
        <property name="name" value="Tony"/>
        <property name="age" value="51"/>
      </object>
    </property>
    <property name="interceptorNames">
        <list>
            <value>debugInterceptor</value>
            <value>myCustomInterceptor</value>
        </list>
    </property>
</object>
```

使用内联对象的优点是只在容器定义了一个与 Person 类相关的对象：如果要防止容器的客户代码获取未经代理的对象，或者要避免使用 IoC 容器时的二义性（即对于同一业务实体需要定义两个对象），内联对象定义就非常有用。另外一个优点存有争议，ProxyFactoryObject 的定义是自包含的（self-contained）。然而，有时确实需要直接获取目标对象的引用，比如说在某些测试环境下。

12.5.3.1. 为每个代理对象应用通知

我们来看一下如何配置 ProxyFactoryObject 返回的代理对象：

```
<!-- create the object to reference -->
<object id="RealObjectTarget" type="MyRealObject" singleton="false"/>
<!-- create the proxied object for everyone to use-->
<object id="MyObject" type="Spring.Aop.Framework.ProxyFactoryObject,
Spring.Aop">
        <property name="proxyInterfaces" value="MyInterface" />
        <property name="isSingleton" value="false"/>
        <property name="targetName" value="RealObjectTarget" />
</object>
```

如果目标对象声明为 prototype，就必须将 ProxyFactoryObject 的 TargetName 属性设为目标对象的名字或 ID，而不能使用 Target 属性引用目标对象。这样，框架才会为每个新的对象创建新的代理。

考虑一下上面的对象定义。注意 ProxyFactoryObject 的 IsSingleton 属性设成了 false。这表示每个代理对象都是不相同的。于是，就可以用下面的代码来配置每个代理对象的通知：

```
MyInterface myProxyObject1 = (MyInterface)ctx.GetObject("MyObject"); //
Will return un-advised instance of proxy object

IAdvised advised = (IAdvised)myProxyObject1;
advised.AddAdvice( new DebugAdvice() );       // myProxyObject1
instance now has an advice attached to it.

MyInterface myProxyObject2 = (MyInterface)ctx.GetObject("MyObject"); //
Will return a new, un-advised instance of proxy object
```

12.5.4.代理类（按：即代理没有实现任何接口的类）

如果需要代理一个类，而不是一个或多个接口，那么又该怎么办?

假如在我们上面的例子中没有 IPerson 接口，我们需要代理的是一个没有实现任何业务接口的 Person 类。在这种情况下，如果没有显式指定接口，同时发现目标对象没有实现任何接口，ProxyFactoryObject 就会代理所有的公有虚方法。我们可以通过配置来强制 Spring.NET 进行类代理而不进行接口代理，方法是将 ProxyFactoryObject 的 ProxyTargeType 属性设为 true。

类代理的工作原理是在运行时创建目标类的子类。Spring.NET 把对目标对象的方法调用转发给生成的子类：生成子类的时应用了 Decorator 模式，并向其中织入了通知。

一般情况下类代理对开发人员来说是透明的。然而必须考虑到一个重要的问题：非虚方法是不能被代理的，因为它们不能被覆盖。这在使用旧代码时是一个限制，因为在默认情况下我们都不会把方法声明为虚方法。

12.6. 使用 ProxyFactory 类通过编程方式创建 AOP 代理

在 Spring.NET 中很容易通过编程方式创建 AOP 代理。编程方式允许在不依赖 IoC 的情况下单独使用 AOP 的功能。

下面例子中，用一个拦截器和一个 Advisor 为目标对象创建代理对象。目标对象的接口（按：指目标对象所实现的接口，因为没有显式定义切入点，这个代理对象会代理目标对象所实现的所有接口）会被自动代理：

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.AddAdvice(myMethodInterceptor);
factory.AddAdvisor(myAdvisor);
IBusinessInterface tb = (IBusinessInterface) factory.GetProxy();
```

第一步是创建 Spring.Aop.Framework.ProxyFactory 的实例。上例中，直接使用目标对象创建了该实例，也可以用重载的构造器指明要代理的接口。

可以为 ProxyFactory 实例添加拦截器或 Advisor，并在该实例的生命周期内管理它们。 *TODO:note on introductions.*

另外，ProxyFactory 也提供了几个快捷方法（继承自 AdvisedSupport），用于添加特定的通知类型，如前置和异常通知。AdviseSupport 类是 ProxyFactory 和 ProxyFactoryObject 的共同基类。

> **注意**
>
> 在多数应用中，使用 IoC 容器管理 AOP 代理是最好的方法，推荐使用容器将 AOP 从.NET 代码中分离出来。

12.7. 管理目标对象

不管用什么方式创建 AOP 代理，都可以用 Spring.Aop.Framework.IAdvised 接口管理它。AOP 代理都实现了这个接口，所以都可以转型为 IAdvised。该接口包括以下方法和属性：

```
public interface IAdvised
```

```
{
    IAdvisor[] Advisors {      get; }

    IIntroductionAdvisor[] Introductions { get; }

    void  AddInterceptor(IInterceptor interceptor);

    void  AddInterceptor(int pos, IInterceptor interceptor);

    void  AddAdvisor(IAdvisor advisor);

    void  AddAdvisor(int pos, IAdvisor advisor);

    void  AddIntroduction(IIntroductionAdvisor advisor);

    void  AddIntroduction(int pos, IIntroductionAdvisor advisor);

    int IndexOf(IAdvisor advisor);

    int IndexOf(IIntroductionAdvisor advisor);

    bool RemoveAdvisor(IAdvisor advisor);

    void RemoveAdvisor(int index);

    bool RemoveInterceptor(IInterceptor interceptor);

    bool RemoveIntroduction(IIntroductionAdvisor advisor);

    void RemoveIntroduction(int index);

    void ReplaceIntroduction(int index, IIntroductionAdvisor advisor);

    bool ReplaceAdvisor(IAdvisor a, IAdvisor b);
}
```

索引器 Advisors 会为每个添加到代理工厂中的 Advisor、拦截器或其它通知类型返回一个 IAdvisor。如果向工厂中添加的是 IAdvisor 类型，那么通过该索引器返回的值就是该对象本身，若添加的是拦截器或其它通知类型，Spring.NET 会将其连同一个总是返回 true 的切入点（按：即一个 TruePointcut 对象，参见 25.2.2 节）一起包装进一个 Advisor 中返回。这样的话，如果向工厂添加一个 IMethodInterceptor，那么通过该索引器返回的 Advisor 就是一个包装了该 IMethodInterceptor 和一个能匹配所有类型和方法的切入点的 DefaultPointcutAdvisor 对象。

AddAdvisor()方法用于添加 IAdvisor。一般添加的是通用的 DefaultPointcutAdvisor，它可以用来包装除引入通知之外所有类型的通知或切入点。

默认情况下，即便是代理已经创建完成，也可以继续添加或删除 Advisor 或拦截器。唯一的限制是不能添加或删除引入 Advisor，因为已创建的代理不会反映出接口的变化。（为解决这一问题，可以从工厂获取一个新代理）

*修改业务对象上的通知是否明智（advisable，这个单词在这儿没有双关意），这是个有争议的问题，尽管确实有过这样的做法。不过，在开发中，这么做可能会非常有用：例如，在测试时，有时会发现以拦截器或其它通知的形式向程序中添加测试代码很方便，这样可以进入一个要测试的方法调用的内部。（比如说，通知可以进入为该方法所创建的事务内部，以便在该事务回滚前用 SQL 来检查数据表是否已正确更新）（按：*不是指方法的内部，而是方法的执行环境内部，拦截器不可能进入一个方法）

根据代理的创建方式不同，可以将 IAdvised 的 IsFrozen 属性设为 true，此时任何修改通知的操作都会抛出 AopConfigException 异常。某些情况下，锁定目标对象的状态是很重要的，例如可以防止调用代码从代理中删除一个用于安全性的拦截器。

12.8.使用“自动代理”功能

到目前为止，我们已经讨论过使用 ProxyFactoryObject 或其它类似的工厂对象显式创建 AOP 代理的方法。如果应用程序需要创建很多 AOP 代理，比如当需要代理某个服务层的所有对象时，这种方法就会使配置文件变的相当庞大。为简化配置过程，Spring.NET 提供了“自动代理”的功能，可以根据条件自动创建代理对象，也就是说，可以将多个对象分组以作为要代理的候选对象。

该功能建立在 IoC 容器的“对象后处理”功能之上，对象后处理允许在容器装载对象时自定义对象。可参考 4.8,使用 IObjectPostProcessor 自定义对象。

为使用自动代理功能，需要在容器中配置一些特殊的对象定义。这样，就可以定义符合自动代理条件的目标对象，而无需使用 ProxyFactoryObject。

- 在容器中配置自动代理的创建对象（Creator），并为其指定位于同一容器中的目标对象。
- 有一种自动代理的创建比较特殊，应该专门考虑一下，即以源代码中的.NET 特性为驱动创建自动代理。

自动代理的好处是不允许调用者或依赖项获得未通知对象。当调用一个应用程序上下文的 GetObject(“MyBusinessObject1”)时，返回的是 AOP 代理，而不是目标对象本身。前面 12.5.3,代理接口中讲过的内联对象也可以提供类似的功能。

12.8.1.自动代理对象的定义

Spring.Aop.Framework.AutoProxy 命名空间定义了一个通用的自动代理框架，我们既可以使用 Spring.NET 提供的自动代理实现类，也可以创建自己的自动代理类。自动代理对象后处理的基础是 AbstractAutoProxyCreator 抽象基类。Spring.NET 为其创建了两个子类，ObjectNameAutoProxyCreator 和 DefaultAdvisorAutoProxyCreator。下面讨论这两个类。

12.8.1.1. ObjectNameAutoProxyCreator

ObjectNameAutoProxyCreator 可以用特定的文本值或通配符匹配目标对象的名称，并为满足条件的目标对象创建 AOP 代理。该类支持模式匹配字符串，如："*name"，"name*"，”*name*“和精确文本如"name"。我们可以通过下面这个简单的例子了解一下自动代理的功能。

```
public enum Language
{
    English = 1,
    Portuguese = 2,
    Italian = 3
}

public interface IHelloWorldSpeaker
{
   void SayHello();
}

public class HelloWorldSpeaker : IHelloWorldSpeaker
{
    private Language language;

    public Language Language
    {
        set { language = value; }
        get { return language; }
    }

    public void SayHello()
    {
        switch (language)
        {
            case Language.English:
                Console.WriteLine("Hello World!");
```

```
                break;
            case Language.Portuguese:
                Console.WriteLine("Oi Mundo!");
                break;
            case Language.Italian:
                Console.WriteLine("Ciao Mondo!");
                break;
        }
    }
}

public class DebugInterceptor : IMethodInterceptor
{
    public object Invoke(IMethodInvocation invocation)
    {
        Console.WriteLine("Before: " + invocation.Method.ToString());
        object rval = invocation.Proceed();
        Console.WriteLine("After:  " + invocation.Method.ToString());
        return rval;
    }
}
```

下面的对象定义用于自动创建 AOP 代理，并为名称匹配“English*“和“PortugueseSpeaker”的类应用 Debug 拦截器。

```
<object id="ProxyCreator"
type="Spring.Aop.Framework.AutoProxy.ObjectNameAutoProxyCreator,
Spring.Aop">
  <property name="ObjectNames">
      <list>
          <value>English*</value>
          <value>PortugeseSpeaker</value>
      </list>
  </property>
  <property name="InterceptorNames">
      <list>
          <value>debugInterceptor</value>
      </list>
  </property>
</object>

<object id="debugInterceptor" type="AopPlay.DebugInterceptor,
AopPlay"/>
```

```
<object id="EnglishSpeakerOne" type="AopPlay.HelloWorldSpeaker,
AopPlay">
  <property name="Language" value="English"/>
</object>

<object id="EnglishSpeakerTwo" type="AopPlay.HelloWorldSpeaker,
AopPlay">
  <property name="Language" value="English"/>
</object>

<object id="PortugeseSpeaker" type="AopPlay.HelloWorldSpeaker,
AopPlay">
  <property name="Language" value="Portuguese"/>
</object>

<object id="ItalianSpeakerOne" type="AopPlay.HelloWorldSpeaker,
AopPlay">
  <property name="Language" value="Italian"/>
</object>
```

ProxyFactoryObject 的 InterceptorNames 属性保存的是拦截器的名称而非引用，这可以纠正以原型模式部署的 Advisor 的行为。以这些名称命名的对象可以是 Advisor 或其它类型的通知。

一个通知可以应用于所有匹配的对象。注意，如果使用了 Advisor（而不是直接使用通知），其中的切入点可能会以不同的方式应用在其它对象上。（按：因为 Advisor 是通知（拦截器）和切入点的组合）

在下面的简单程序中，我们应用了 AOP 拦截器。

```
IApplicationContext ctx = ContextRegistry.GetContext();
IDictionary speakerDictionary =
ctx.GetObjectsOfType(typeof(IHelloWorldSpeaker));
foreach (DictionaryEntry entry in speakerDictionary)
{
    string name = (string)entry.Key;
    IHelloWorldSpeaker worldSpeaker = (IHelloWorldSpeaker)entry.Value;
    Console.Write(name + " says; ");
    worldSpeaker.SayHello();
}
```

该程序输出为：

```
ItalianSpeakerOne says; Ciao Mondo!
EnglishSpeakerTwo says; Before: Void SayHello()
```

```
Hello World!
After:  Void SayHello()
PortugeseSpeaker says; Before: Void SayHello()
Oi Mundo!
After:  Void SayHello()
EnglishSpeakerOne says; Before: Void SayHello()
Hello World!
After:  Void SayHello()
```

12.8.1.2. DefaultAdvisorAutoProxyCreator

另一个更为通用、功能也更强大的自动代理创建类是 DefaultAdvisorAutoProxyCreator。该类会在当前容器中自动应用满足条件的 Advisor，而不用在自动代理 Advisor 的对象定义中包含特定的对象名。它既可以保持配置文件的一致性，又可避免 ObjectNameAutoProxyCreator 引起的配置文件的臃肿。

使用此类时需要：

- 配置 DefaultAdvisorAutoProxyCreator 的对象定义
- 在相同或相关的上下文中定义一系列 Advisor。注意必须是 Advisor，不能直接使用拦截器或其它类型的通知。这是因为必须要有一个切入点来检查每个候选对象是否满足通知的条件。

DefaultAdvisorAutoProxyCreator 会自动检查每个 Advisor 中的切入点，以便为容器中的每个对象应用合适的通知（如果有通知的话）。

也就是说，容器可以自动的为每个业务对象应用任意多的 Advisor。如果所有 Advisor 的切入点都与业务对象的方法不匹配，该对象就不会被代理。

如果需要把同一通知应用在多个业务对象上，DefaultAdvisorAutoProxyCreator 的便利性就能体现出来了。只要自动代理配置得当，就可以任意添加新的业务对象，而不必为其配置专门的代理对象，自动代理可自动将通知应用到新对象上。还可以在对配置改动最少的情况下删除一些额外的（与需求无关的）方面，如跟踪和性能监视等。

我们来看看下面例子中 DefaultAdvisorAutoProxyCreator 的用法。在前文例子的基础上，添加了一个新类 SpeakerDao，用来查找和存储 IHelloWorldSpeaker 对象：

```
public interface ISpeakerDao
{
    IList FindAll();
```

```
    IHelloWorldSpeaker Save(IHelloWorldSpeaker speaker);
}

public class SpeakerDao : ISpeakerDao
{
    public System.Collections.IList FindAll()
    {
        Console.WriteLine("Finding speakers...");
        // just a demo...fake the retrieval.
        Thread.Sleep(10000);
        HelloWorldSpeaker speaker = new HelloWorldSpeaker();
        speaker.Language = Language.Portuguese;

        IList list = new ArrayList();
        list.Add(speaker);
        return list;
    }

    public IHelloWorldSpeaker Save(IHelloWorldSpeaker speaker)
    {
        Console.WriteLine("Saving speaker...");
        // just a demo...not really saving...
        return speaker;
    }
}
```

在 XML 配置中，定义了两个 Advisor，即通知（要添加的行为）和切入点（要添加行为的位置）的组合。RegularExpressionMethodPointcutAdvisor 可以用正则表达式切入点来匹配目标对象的方法名。当然也可以使用自定义的切入点，此时就应该用 DefaultPointcutAdvisor 来定义 Advisor 了。本例中，所有这些 Advisor、通知和 SpeakerDao 的对象定义如下：

```
<object id="SpeachAdvisor"
type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor,
Spring.Aop">
    <property name="advice" ref="debugInterceptor"/>
    <property name="patterns">
        <list>
            <value>.*Say.*</value>
        </list>
    </property>


</object>
```

```
<object id="AdoAdvisor"
type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor,
Spring.Aop">

    <property name="advice" ref="timingInterceptor"/>
    <property name="patterns">
        <list>
            <value>.*Find.*</value>

        </list>
    </property>

</object>

// Advice
<object id="debugInterceptor" type="AopPlay.DebugInterceptor,
AopPlay"/>

<object id="timingInterceptor" type="AopPlay.TimingInterceptor,
AopPlay"/>

// Speaker DAO Object - has 'FindAll' Method.
<object id="speakerDao" type="AopPlay.SpeakerDao, AopPlay"/>

// HelloWorldSpeaker objects as previously listed.
```

在配置文件中添加一个 DefaultAdvisorAutoProxyCreator：

```
<object id="ProxyCreator"
type="Spring.Aop.Framework.AutoProxy.DefaultAdvisorAutoProxyCreator,
Spring.Aop"/>
```

框架会为上下文中所有名称包含“Say“的方法应用 Debug 拦截器，为所有名称包含“Find“的方法应用 Timing 拦截器。运行以下代码，注意 SpeakerDao 的 Save 方法没有应用任何通知：

```
IApplicationContext ctx = ContextRegistry.GetContext();
IDictionary speakerDictionary =
ctx.GetObjectsOfType(typeof(IHelloWorldSpeaker));
foreach (DictionaryEntry entry in speakerDictionary)
{
    string name = (string)entry.Key;
    IHelloWorldSpeaker worldSpeaker = (IHelloWorldSpeaker)entry.Value;
    Console.Write(name + " says: ");
```

```
    worldSpeaker.SayHello();
}
ISpeakerDao dao = (ISpeakerDao)ctx.GetObject("speakerDao");
IList speakerList = dao.FindAll();
IHelloWorldSpeaker speaker = dao.Save(new HelloWorldSpeaker());
```

程序输出为：

```
ItalianSpeakerOne says; Before: Void SayHello()
Ciao Mondo!
After:  Void SayHello()
EnglishSpeakerTwo says; Before: Void SayHello()
Hello World!
After:  Void SayHello()
PortugeseSpeaker says; Before: Void SayHello()
Oi Mundo!
After:  Void SayHello()
EnglishSpeakerOne says; Before: Void SayHello()
Hello World!
After:  Void SayHello()
Finding speakers...
Elapsed time = 00:00:10.0154745
Saving speaker...
```

DefaultAdvisorAutoProxyCreator 支持过滤（即只根据命名规则应用特定的 Advisor，允许在同一容器中使用多个配置不相同的 AdvisorAutoProxyCreator）和排序。如果需要排序，Advisor 可以实现 Spring.Core.IOrdered 接口以确保排序的正确性。默认为不排序。

12.8.1.3. AbstractAutoProxyCreator

这是 DefaultAdvisorAutoProxyCreator 的基类。如果 DefaultAdvisorAutoProxyCreator 不满足某些特殊要求，可以考虑继承该类来创建自定义的自动代理创建类。

12.8.2.使用特性驱动的自动代理

有一类特别重要的自动代理，即特性驱动的自动代理。这类自动代理的编程模型与使用服务组件发布企业服务差不多。

在这种情况下，要把 DefaultAdvisorAutoProxyCreator 和能够处理特性的 Advisor 结合起来使用。这种 Advisor 的切入点由源码中的.NET 特性指定，并且

用特性的数据和/或方法进行配置。特性驱动的自动代理功能很强大，不论是在编程方式中还是在配置方式中，都可以代替传统的自动代理方法。

TODO: Example based on transaction attributes.

12.9. 使用 TargetSources

Spring.NET 定义了一个名为 TargetSource 的概念，由 Spring.Aop.ITargetSource 接口表示。该接口的功能是返回包含连接点的“目标对象”。每次 AOP 代理处理方法调用的时候，都会向 ITargetSource 的实现类请求目标对象。

开发人员一般不会直接用到 ITargetSource，但这个接口可以支持池化（pooling）、动态切换（hot swappable）等复杂的目标对象类型。例如一个池化的 TargetSource 可以为每次方法调用返回不同的目标实例，并用对象池来管理这些实例。

如果不显式指定 TargetSource 实现类，AOP 框架会用默认的 TargetSource 类来包装目标对象。对每次调用来说，返回的都是同一个目标对象（和我们的要求一致）。

我们先来看 Spring.NET 实现的几个标准 TargetSource 类以及它们的用法。

在使用自定义 TargetSource 的时候，目标对象一般应该定义为 prototype 模式。这样 Spring.NET 就可以在需要时创建新的目标对象。

12.9.1.动态切换 TargetSource

Spring.Aop.Target.HotSwappableTargetSource 可以在运行期切换 AOP 代理的目标对象，同时允许调用者保存目标对象的引用.

目标对象的改变会立即生效。HotSwappableTargetSource 是线程安全的。

我们可以调用 HotSwappableTargetSource 的 Swap()方法来改变目标对象，如下：

```
HotSwappableTargetSource swapper =
    (HotSwappableTargetSource) objectFactory.GetObject("swapper");
object oldTarget = swapper.swap(newTarget);
```

XML 定义如下：

```
<object id="initialTarget" type="MyCompany.OldTarget, MyCompany">
</object>
```

```
<object id="swapper"
    type="Spring.Aop.Target.HotSwappableTargetSource, Spring.Aop">
    <constructor-arg><ref local="initialTarget"/></constructor-arg>
</object>

<object id="swappable"
    type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop"
>
    <property name="targetSource">
        <ref local="swapper"/>
    </property>
</object>
```

在上面的代码中，Swap()方法改变了 AOP 代理 swappable 的目标对象。引用 swappable 的客户代码不会查觉到目标对象的改变，直接就可以操作新的目标对象。

虽然这个例子没有添加任何通知——实际上使用 TargetSource 并不要求添加通知——但 TargetSource 可以和任何通知类型一起使用。

12.9.2.池化 TargetSource

池化 TargetSource 允许开发人员用对象池维护一组目标实例，当有方法调用时，从池中获取目标对象。

Spring.NET 的对象池与.NET 企业服务的对象池之间最大的区别是：Spring.NET 的对象池适用于任意的 PONO（Plain old .NET object）。在 Spring.NET 中，对象池的应用同样也采用非侵入的方式。

目前，Spring.NET 对象池的实现是以 Jakarta Commons Pool 1.1 为基础的，Jakarta Commons 的 Pool 组件非常高效。为使用对象池，应用程序需要引用 Spring.Pool 程序集。同时，也可以扩展
Spring.Aop.Target.AbstractPoolingTargetSource 类来支持其它对象池 API。

下面是一个示例配置：

```
<object id="businessObjectTarget" type="MyCompany.MyBusinessObject,
MyCompany" singleton="false">
    ... properties omitted
</object>

<object id="poolTargetSource"
type="Spring.Aop.Target.SimplePoolTargetSource, Spring.Aop">
    <property name="targetObjectName" value="businessObjectTarget"/>
```

```
    <property name="maxSize" value="25"/>
</object>

<object id="businessObject"
type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
    <property name="targetSource" ref="poolTargetSource"/>
    <property name="interceptorNames" value="myInterceptor"/>
</object>
```

注意目标对象——在本例中即 businessObjectTarget——必须是 prototype 模式。这样 PoolingTargetSource 才能在必要时创建新的实例来增长对象池。请参考 SDK 文档中 AbstractPoolingTargetSource 及其子类的属性：最基本的是 MaxSize 属性，一般来说必须设置该属性。

在这个例子中，myInterceptor 是拦截器的名称，拦截器必须定义在同一个 IoC 容器中。但是，使用对象池不要求必须指定拦截器。如果只需要使用对象池而不需要应用其它通知，不设置 InterceptorNames 即可。

通过配置，可以将对象池内的任意对象转型为 Spring.Aop.Target.PoolingConfig 接口，该接口可以获取对象池的配置信息和当前大小，AbstractPoolingTargetSource 类就实现了该接口。请看下面这个 Advisor 的对象定义：

```
<object id="poolConfigAdvisor"
    type="Spring.Object.Factory.Config.MethodInvokingFactoryObject,
Spring.Aop">
    <property name="target" ref="poolTargetSource" />
    <property name="targetMethod" value="getPoolingConfigMixin" />
</object>
```

调用 AbstractPoolingTargetSource 的方法，再通过 MethodInvokingFactoryObject 类即可获得这个 Advisor。在用来创建池内对象的 ProxyFactoryObject 对象定义（按：即前文的 businessObject 对象）中，属性 InterceptorNames 必须包含此 Advisor 的名称(此处是 poolConfigAdvisor)。

转型的方法如下所示：

```
PoolingConfig conf = (PoolingConfig)
objectFactory.GetObject("businessObject");
Console.WriteLine("Max pool size is " + conf.getMaxSize());
```

一般情况下，无状态的服务对象不需要池化。因为大多数无状态对象都是线程安全的，如果缓存了某些资源，对象池反而可能出现问题。

通过自动代理可以实现功能较为简单的对象池。多个自动代理创建对象（autoproxy creator）可以共用同一个 TargetSource。

12.9.3. PrototypeTargetSource

Prototype 的 TargetSource 与池化 TargetSource 很相似，此时，AOP 框架会为每次方法调用创建新的目标实例。创建新对象的开销可能会很大，而装配新对象（即注入依赖项）的开销则可能更大。所以，除非绝对需要，否则不要使用这种方法。

要使用 Prototype 的 TargetSource，只要修改前面的 poolTargetSource 对象定义。（为清晰起见，对象定义的名称也改了。）

```
<object id="prototypeTargetSource"
        type="Spring.Aop.Target.PrototypeTargetSource, Spring.Aop">
    <property name="targetObjectName" value="businessObject" />
</object>
```

这里只设置了一个属性：目标对象的名字。在 TargetSource 的实现中，用继承方式来保证该属性值的一致性。如果使用池化 TargetSource，目标对象本身必须定义为 prototype。

12.10. 定义新的通知类型

Spring.NET 的 AOP 框架是可以扩展的，除了目前已实现的拦截环绕、前置、异常和后置通知，也可以自定义其它任意的通知类型。

Spring.Aop.Framework.Adapter 命名空间（这是一个 SPI——Service Provider Interface）允许在不改变核心框架的前提下，加入新的自定义通知类型。对自定义通知类型的唯一约束是，必须实现 AopAlliance.Aop.IAdvice 标识接口。

可以参考 Spring.Aop.Framework.Adapter 命名空间的文档以了解更多信息。

12.11. 参考资源

Spring.NET 团队推荐阅读 Ramnivas Laddad(Manning,2003)的精彩著作《AspectJ in Action》，书中介绍了 AOP 基础知识。

如果对 Spring.NET AOP 框架的高级性能感兴趣，可参考测试包，其中包含一些本章未提及的高级特性。

第十三章. 通用日志抽象层

13.1. 简介

Spring.NET 用一组简单的抽象类库在具体的日志 API（如 log4net，EntLib logging，NLog）和 Spring.NET 的日志调用之间形成了一个间接层。很多.NET 项目也都做了类似的工作，所以我们将相关的类库移出了 Spring.NET，创建了一个更为通用的开源项目。请随时留意这方面的信息。

Spring.NET 的日志抽象层名为“Common.Logging”，它来自 iBATIS 项目，算是 iBATIS 日志抽象的一个紧凑版本。在此十分感谢 iBATIS!该类库适用于.NET 1.0、1.1 和 2.0，支持调试功能和强命名程序集。

Common.Logging.dll 是随 Spring.NET 一起发布的，支持基于控制台和 Trace 的 Logger。该程序集位于 lib 文件夹下，且分别针对 log4net 1.2.9 和 log4net1.2.10 有两个版本的实现。这么做是因为 log4net 这两个版本的程序集都使用强命名，不能在运行时重定向。

注意 Common.Logging 的目的不是要取代现有的其它日志类库。它的 API 相当简陋，将来也不打算做的更复杂。该类库的惟一用途是解决同时使用多种日志 API 的需求。

13.1.1. Logging API

API 相当的简单：

```
public interface ILog
{
 void Debug( object message );
 void Debug( object message, Exception exception );
 void Error( object message );
 void Error( object message, Exception exception );
 void Fatal( object message );
 void Fatal( object message, Exception exception );
 void Info( object message );
 void Info( object message, Exception exception );
 void Warn( object message );
 void Warn( object message, Exception exception );

 bool IsDebugEnabled { get; }
 bool IsErrorEnabled { get; }
 bool IsFatalEnabled { get; }
 bool IsInfoEnabled { get; }
 bool IsWarnEnabled { get; }
}
```

可以用 LogManager 类获取 ILog 的引用。LogManager 定义如下：

```
public sealed class LogManager
{
 public static ILog GetLogger( Type type ) ...
 public static ILog GetLogger( string name ) ...

 public static ILoggerFactoryAdapter Adapter ...

}
```

其中的 Adapter 属性由框架本身使用。

调用代码基本上如下所示：

13.2. 实现与配置

目前的实现都很简单，包括一个基于 log4net 的实现。

13.2.1.控制台 Logger

基础类库 Common.Logging 包含一个控制台输出 Logger，可以用以下方式配置：

```
 <configSections>

 <sectionGroup name="common">
 <section name="logging"
type="Common.Logging.ConfigurationSectionHandler, Common.Logging" />
 </sectionGroup>

 </configSections>

 <common>
 <logging>
 <factoryAdapter
type="Common.Logging.Simple.ConsoleOutLoggerFactoryAdapter,
Common.Logging">
 <arg key="showLogName" value="true" />
 <arg key="showDataTime" value="true" />
 <arg key="level" value="DEBUG" />
 <arg key="dateTimeFormat" value="yyyy/MM/dd HH:mm:ss:fff" />
 </factoryAdapter>
```

```
</logging>
</common>
```

13.3. Log4Net

基于 log4net 的实现有两个版本，它们的配置很相似，惟一的区别是指定给工厂适配器的类型。基于 log4net 1.2.10 的实现是这样：

```
<common>
<logging>
<factoryAdapter
type="Common.Logging.Log4Net.Log4NetLoggerFactoryAdapter,
Common.Logging.Log4Net">
<!-- choices are INLINE, FILE, FILE-WATCH, EXTERNAL-->
<!-- otherwise BasicConfigurer.Configure is used -->
<!-- log4net configuration file is specified with key configFile-->
<arg key="configType" value="INLINE" />
</factoryAdapter>
</logging>
</common>
```

如果用 log4net 1.2.9，要把程序集名称改为 Common.Logging.Log4Net129。

第十四章. 事务管理

14.1. 简介

Spring.NET 用一个统一的抽象层来进行事务管理，其优点如下：

- 为不同的事务 API，包括 ADO.NET、企业服务、System.Transactoins 和 NHibernate，提供统一的编程模型。
- 能够在上述任一种数据访问技术中使用声明式事务管理。
- 为编程方式的事务管理提供了一套易用的 API。
- 与 Spring.NET 高层持久化 API（如 AdoTemplate）完美集成。

本章的几个小节分别阐述 Spring.NET 事务管理技术的几项功能。另外也会围绕事务管理这一主题讨论一些最佳的编程方式。

- 第二节，动机，阐述我们为什么要使用 Spring.NET 的事务抽象，而不直接使用 System.Transactions 或某个数据访问技术的事务 API。
- 第三节，核心接口，介绍 Spring.NET 事务管理的核心类型及其用法。
- 第五节，声明式事务管理，讨论如何用声明方式进行事务管理。

- 第六节，编程式事务管理，讨论如何用编程方式进行事务管理。

14.2. 动机

目前有很多种数据访问技术。在.NET FCL 中，有三类 API 可以执行事务管理，分别是 ADO.NET、企业服务和 System.Transactions。其它的数据访问技术，如对象关系映射（object relational mappers）和结果集映射（result-set mapping）等等的应用也很广泛，每种技术也都有自己的事务管理 API。事务管理的代码一般是直接和各种事务 API 绑定在一起的，所以在开发时必须根据所用的具体技术来决定采用哪种 API。但是，这种代码与事务 API 的紧耦合决定了很难通过简单的重构来解决更换数据访问技术的问题。而 Spring.NET 的事务框架允许在各种数据访问技术之上使用相同的 API。通过配置或者集中的编程方式，可以很容易的更换后台事务 API，而不需要对代码进行“大修”。

我们可以用业界公认的最佳方式来建立一种数据访问机制。Martin Fowler 的著作《Patterns of Enterprise Application Architecture》讲到了许多在实际应用中非常成功的数据访问方法。其一便是在应用程序架构中引入一个数据访问层。数据访问层不仅要考虑到与不同的数据库和数据访问技术的兼容性，而且职责要严格限制在数据访问功能上。数据访问层应该只包含数据访问对象（DAO）以及“创建/获取/更新/删除”（CRUD，Create/Retrieve/Update/Delete）的操作，不应该涉及任何业务逻辑。业务逻辑应该位于单独的业务服务层，并且需要与一或多个 DAO 协作来完成高层次的用户功能。

为了在事务中“要么全执行要么全不执行”这些用户功能，事务环境（transaction context）就应该由业务服务层（或某个“更高”的层次）控制。在实现上，一个很重要的细节是如何让 DAO 了解在其它层次中开始的“外部”事务。如果让 DAO 自己负责连接和事务的管理，就把问题看的过于简单化了，因为此时每个 DAO 都会执行自己的事务/资源管理，所以无法在同一事务中执行多个 DAO 操作。我们需要一种有效的手段，将连接/事务成对的从业务服务层传递给 DAO。方法有很多种，最不具侵入性的就是将连接/事务作为方法参数显式的传递给 DAO。另一种方法是将连接/事务放在线程本地存储内。不管使用哪种方法，只要在用 ADO.NET，就必须得自己创建一个基础框架来完成这个任务。

但是，等一下，企业服务不是能解决这个问题吗——还有 System.Transactions 命名空间呢？关于这个嘛，答案是对...也不对。企业服务确实能够让我们在事务环境中使用“原生”的 ADO.NET 在同一事务中执行多个 DAO 操作。但它的缺点是必须通过 MS-DTC（Microsoft Distributed Transaction Coordinator）使用分布式（全局的）事务。如果只为了使用全局事务就必须依赖 MS-DTC，那应用程序在性能上就会大打折扣了。

使用.NET2.0 新增 System.Transactions 命名空间下的 TransactonScope 类时，也有相同的问题。TransactonScope 类的目的实际上是——用 using 语句使一段代码成为事务性代码。只要访问事务性的资源，using 语句中的普通 ADO.NET 代码就会在一个 ADO.NET 本地事务中运行。但是，System.Transactions（和数据

库）的“神奇“之处在于，如果需要访问第二个事务性资源，本地事务就会升级为分布式事务。这个过程叫做 PSPE（Promotable Single Phase Enlistment）。此外，还需提醒读者：在同一个数据库上使用同一连接字符串打开第二个连接，就会使本地事务升级为分布式事务。所以，如果每个 DAO 都执行自己的连接管理，那就完了：本地事务会突然升级为分布式事务！如果应用程序只使用一个数据库，要想避免这个问题，就必须将唯一的连接对象传递给系统中的所有 DAO。另外要注意的是某些数据库不支持 PSPE，就算用单个数据库连接也会使用分布式事务（比如 Oracle）。

Spring.NET 的声明式事务管理功能非常强大。在数据库事务领域，讨论声明式事务管理的话题并不是很多，因为现在开发人员已经可以不再直接用凌乱繁复的事务 API 来进行事务管理了，而是可以通过在类和方法上应用某些特性来进行。但是在 FCL 中，只有企业服务提供了这一功能。Spring.NET 则填补了这个空白——不管使用哪种事务管理技术：ADO.NET，还是（最常用的）System.Transactions，都可以使用声明式的事务管理。另外，企业服务也有自己的问题，举例来说，如果需要为查询/读取操作和创建/更新/删除操作设置不同的隔离等级，就必须将这两组操作分隔在不同的类中实现，因为在企业服务中，声明式事务的元数据只对类有效。但总的来说，企业服务，特别是在 XP sp2 和 Server 2003 中新出现的“无组件服务”，还有在应用程序进程内驻留的功能都是相当不错的。不过，虽然有这些优点，企业服务仍尚未在开发社区中引起很大的关注。

Spring.NET 事务管理的宗旨，就是要减轻 FCL 或第三方数据访问技术给开发人员带来的这些“痛苦”。它支持声明式事务管理，可以用配置方式获取事务选项的元数据——目前支持两种声明方式：在代码中用.NET 特性声明；在 IoC 容器中用 XML 声明。

最后，Spring.NET 事务管理还允许在同一事务中使用不同的数据访问技术——例如混合使用 ADO.NET 和 NHibernate。

好了，再讲估计就有点烦人了，现在我们来看代码。

14.3. 核心接口

Spring.NET 事务管理的核心概念是事务策略。事务策略由 Spring.Transaction.IPlatformTransactionManager 接口定义，该接口如下：

```
public interface IPlatformTransactionManager {

  ITransactionStatus GetTransaction( ITransactionDefinition
definition );

  void Commit( ITransactionStatus transactionStatus );
```

```
  void Rollback( ITransactionStatus transactionStatus );

}
```

虽然该接口可以通过编程方式使用，但它本质上是一个“SPI”（Service Provider Interface）。注意，IPlatformTransactionManager 是一个接口，以 Spring.NET 的观点来看，必要时很容易为其创建一个模拟实现。IPlatformTransactionManager 实现类的对象在 IoC 容器中的定义方式与其它对象没有区别。在 Spring.NET 中，提供了以下实现类：

- AdoPlatformTransactionManager- 基于本地 ADO.NET 的事务。
- ServiceDomainPlatformTransactionManager- 由企业服务提供的分布式事务管理器。
- TxScopePlatformTransactionManager- 由 System.Transactions 提供的本地/分布式的事务管理器。

这些类在后台调用了以下 FCL 方法：AdoPlatformTransactionManager 调用了 Transaction.Begin()、Transaction.Commit()和 Transaction.Rollback()方法。ServiceDomainPlatformTransactionManager 是一种“无组件式服务（Services without Components）”，所以我们不需要继承 ServicedComponent。ServiceDomainPlatformTransactionManager 会调用 ServiceDomain.Enter()、ServiceDomain.Leave()方法和 ContextUtil.SetAbort()方法。注意我们可以用 Spring.NET 的企业服务导出类为任何 PONO 创建企业服务代理。最后，TxScopePlatformTransactionManager 调用 TransactionScope 的构造器、TransactionScope.Complete()、TransactionScope.Dispose()以及 Transaction.Current.Rollback()方法。每种事务管理器都可以用自己的属性来指定所使用的数据访问技术。详细信息请参考 API 文档，不过本章的例子可以让您更好的了解一些基础知识。

注意，其它事务管理器如 HibernateTransactionManager，属于 Spring.NET 的模块项目，可以单独下载。

IPlatformTransactionManager 的 GetTransaction(..)方法会根据其参数（为 ITransactionDefinition 类型）返回一个 ITransactionStatus 的对象。返回的 ITransactionStatus 可能是一个新的事务，也可能是一个现有事务（如果在当前调用堆栈中找得到匹配事务的话，也就是说，ITransactionStatus 是和正在执行的逻辑线程相关的。）

ITransactionDefinition 接口封装了以下信息：

- Isolation：该事务对其它事务操作的隔离级别。例如，用来表示某个事务是否能看到其它事务写入的、但尚未提交的信息。
- Propagation：一般情况下，TransactionScope 内的代码都会在为其指定的事务中运行。但是，该属性可用来设置如果某个事务环境已经存在时，

该事务内的方法是否要执行：比如说，是简单的让它在现有事务中继续运行呢（这是一般情况），还是挂起现有事务然后创建一个新事务来运行。

- Timeout：在超时（并且被事务基础框架自动回滚之前）前该事务可以运行多久。
- Read-only 状态：只读的事务不会修改任何数据。在某些情况下（比如使用 NHibernate 时），只读事务能显著提高性能。

这些信息反映了事务处理的标准概念。如果需要的话，请参考一下与事务隔离级别和其它核心概念相关的资源，因为不管是使用 Spring.NET 还是其它事务管理方案，理解这些核心概念都是很有必要的。

通过 ITransactionStatus 接口，事务中的代码可以很方便的控制事务的执行并查询事务的状态。

在 Spring.NET 事务管理中，不管是使用声明方式还是编程方式，选择合适的 IPlatformTransactionManager 都是绝对必要的。根据 Spring 的风格，一般通过依赖注入来管理 IPlatformTransactionManager 的实例。

IPlatformTransactionManager 的各个实现类一般需要对相应的后台环境有所了解，比如 ADO.NET、NHibernate 等等。通过下面的例子，您会了解如何定义一个基于标准 ADO.NET 的 IPlatformTransactionManager。

我们必须先定义一个 Spring.NET 的 IDbProvider 对象，然后将它传递给 AdoPlatformTransactionManager。下一章会详细讨论 IDbProvider 接口。

```
<objects xmlns='http://www.springframework.net'
         xmlns:db="http://www.springframework.net/database">

    <db:dbProvider id="DbProvider"
                provider="SqlServer-1.1"
                connectionString="Data
Source=(local);Database=Spring;User
ID=springqa;Password=springqa;Trusted_Connection=False"/>

    <object id="TransactionManager"
            type="Spring.Data.AdoPlatformTransactionManager,
Spring.Data">
        <property name="DbProvider" ref="DbProvider"/>
    </object>

    . . . other object definitions . . .

</objects>
```

我们也可以用同样简单的方式使用基于 System.Transactions 的事务管理器，如下面的代码所示：

```
    <object id="TransactionManager"
            type="Spring.Data.TxScopeTransactionManager,
Spring.Data">
    </object>
```

HibernateTransactionManager 的用法也与此类似，可以参考 Spring.NET NHibernate 模块的文档。

注意，不管使用哪种 IPlatformTransactionManager，应用程序的代码都不需要更改，因为依赖注入是策略模式的绝配。仅通过修改配置我们就能更改事务管理的后台技术，就算改变之后会从本地事务升级为全局事务（或者相反），也不会对我们的代码造成任何影响。

14.4. 用事务进行资源同步

应用程序代码怎样共享由不同的事务管理器创建/重用/清理的资源（比如连接/事务/会话等）？我们有两种方法：高层次方法和低层次方法。

14.4.1. 高层次方法

Spring.NET 的高层次持久化 API 是我们优先选用的方法。这些 API 并不是现有各种事务 API 的替代品，而是负责在后台管理资源的创建/重用和清理，以及资源的事务同步(如事件通知)和异常映射，客户代码无需关心这些样板式的问题，故可以专注于持久化逻辑。一般来说各种持久化 API 使用的都是同一种控制反转方式：在准备好所有相关资源之后，用回调方法或委托来调用客户代码——比如：在事务选项元数据中设置好 DbCommand 的 Connection 和 Transaction 属性后，将这个 DbCommand 传递给回调方法使用。Spring.NET 中的高层次持久化类一般以某某 Template 来命名，比如 AdoTemplate 和 HibernateTemplate。这些模板类的很多方法都是以回调/IoC 的方式建立的，比如通过让客户代码实现特定的回调接口来进行回调。

14.4.2.低层次方法

TODO. A utility class can be used to directly obtain a connection/transaction pair that is aware of the transactional calling context and returns and pair suitable for that context.

14.5. 声明式事务管理

多数 Spring.NET 用户都会选择使用声明式事务管理。声明式事务管理对代码的影响最小，也符合轻量级容器的非侵入性原则。

虽然 Spring.NET 声明式事务管理是利用 Spring.NET 的 AOP 框架实现的，但因为与事务方面相关的代码都内置在 Spring.NET 中，且用法也比较模式化，所以理解这些代码并不需要具备 AOP 的知识。

最基本的声明方式是在方法级别指定事务行为(或不指定事务行为而单独应用事务，见后面的例子）。如果有需要，还可以在一个事务环境中设置由 IPlatformTransactionManager 返回的 ITransactionStatus 对象的 RollbackOnly 属性，以进行一个只用于回滚的事务。关于 Spring.NET 的声明式事务管理，有些问题需要强调一下：

- 声明式事务管理可以工作于任何环境，包括 ADO.NET，System.Transactions，NHibernate 等等，环境改变后只需要修改配置文件。
- 声明式事务管理可用于任意类型，不需要继承任何与基础框架相关的基类如 ServicedComponent。
- 可以用声明方式定义回滚规则。回滚规则可用声明方式进行控制，允许在事务环境中规定只有特定的异常才能触发回滚。
- Spring.NET 允许通过 AOP 来自定义事务行为。例如，如果需要的话可以在事务回滚中加入自定义的行为，也可以应用任意类型的 AOP 通知。
- Spring.NET 不支持跨远程边界调用的事务环境传播。

注意通过 XML 配置回滚规则的功能还在开发之中。

回滚规则的概念很重要：回滚规则可以规定哪些异常应该引发自动回滚。我们在配置中使用声明方式定义回滚规则，而不会涉及到代码本身。所以，虽然我们仍然可以通过编程方式设置 ITransactionStatus.RollbackOnly 属性来将当前事务回滚，但在多数情况下，我们会用回滚规则来确定当哪些异常抛出时应自动回滚。由此带来的优点是明显的，因为业务对象不需要依赖任何事务底层框架。比如说，一般情况我们不需要在代码中直接使用 Spring.NET 的 API。

14.5.1.理解 Spring.NET 声明式事务管理的实现

本节将为用户揭开声明式事务管理的神秘面纱。通过简单的讲解，希望用户能了解[Transaction]特性及其在 IoC 容器中的配置方法。本节会解释 Spring.NET 声明式事务管理的内部工作原理，如果将来遇到了与事务有关的问题，也能对用户有所帮助。

注意

研究 Spring.NET 源代码是理解 Spring.NET 事务管理的最佳方法。API 文档中这部分相关内容应该也已经完成了。在使用 Spring.NET 开发应用程序时，建议将日志级别设为“DEBUG”以便更好的查看后台执行情况。

关于 Spring.NET 的声明式事务管理，有一个最为重要的概念应该掌握，即：事务管理是通过 AOP 代理实现的，且事务通知由元数据驱动(目前通过 XML 或特性)。将通知与事务元数据相结合，所产生的 AOP 代理利用 TransactionInterceptor 和 IPlatformTransactionManager 接口的特定实现类在方法调用周围驱动事务。

注意

虽然在使用 Spring.NET 声明式事务管理的时候不要求掌握 AOP（特别是 Spring.NET 的 AOP）的知识，但了解 AOP 还是会有帮助的。我们曾在 AOP 的相关章节（第十二章）对 Spring.NET 的 AOP 框架作过全面的阐述。

从概念上讲，事务代理对象的方法调用过程可以用下图来表示：

14.5.2.第一个例子

请看下面的接口。在本例中，您只需专注于事务的用法，不必理会其中与业务领域相关的细节。ITestObjectManager 是一个挺寒酸的业务服务层接口——它的实现类会用到两个 DAO。很明显这个服务层过于简单，其中根本没有任何业务逻辑。该“服务”接口如下：

```
public interface ITestObjectManager
{
  void SaveTwoTestObjects(TestObject to1, TestObject to2);

  void DeleteTwoTestObjects(string name1, string name2);
}
```

下面是 ITestObjectManager 的实现类：

```
public class TestObjectManager : ITestObjectManager
{

   // Fields/Properties ommited

   [Transaction()]
   public void SaveTwoTestObjects(TestObject to1, TestObject to2)
   {
```

```
      TestObjectDao.Create(to1.Name, to1.Age);
      TestObjectDao.Create(to2.Name, to1.Age);
   }

   [Transaction()]
   public void DeleteTwoTestObjects(string name1, string name2)
   {
      TestObjectDao.Delete(name1);
      TestObjectDao.Delete(name2);
   }
}
```

注意其中应用在方法上的 Transaction 特性。该特性可以指定其它选项，比如隔离等级，但在本例中就使用默认的设置。不过，请注意仅在方法上应用 Transaction 特性还不能启用事务行为——Transaction 只是一个由别人使用的元数据：Spring.NET 中某些对象会利用这些元数据给相应的对象配置事务行为。

TestObjectDao 则包含基本的创建、修改、删除和查找方法，用来操作我们的“领域“对象：TestObject。TestObject 定义了一些简单的属性，例如姓名和年龄。

```
public interface ITestObjectDao
{
  void Create(string name, int age);
  void Update(TestObject to);
  void Delete(string name);
  TestObject FindByName(string name);
  IList FindAll();
}
```

Create 和 Delete 方法的实现如下所示。注意其中用到的 AdoTemplate 类会在下一章讨论，请参考：14.4,使用事务进行资源同步中关于 Spring.NET 高层持久化 API 与事务管理相结合的内容。

```
public class TestObjectDao : AdoDaoSupport, ITestObjectDao
{
   public void Create(string name, int age)
   {
       AdoTemplate.ExecuteNonQuery(CommandType.Text,
           String.Format("insert into TestObjects(Age, Name) VALUES
({0}, '{1}')",
           age, name));
   }

   public void Delete(string name)
   {
```

```
        AdoTemplate.ExecuteNonQuery(CommandType.Text,
            String.Format("delete from TestObjects where Name = '{0}'",
            name));
    }
}
```

TestObjectManager 通过标准的依赖注入技术来配置自己的 DAO 对象。客户代码最终会得到一个用 Transactoin 特性的元数据配置好的事务代理，在本例中，客户代码是直接从 IoC 容器中请求 ITestObjectManager 实例的。注意，一般来说 ITestObjectManager 还会通过依赖注入配置给其它更高层的对象，比例 Web 服务等。

客户代码如下所示：

```
IApplicationContext ctx =
   new
XmlApplicationContext("assembly://Spring.Data.Integration.Tests/Sprin
g.Data/autoDeclarativeServices.xml");

ITestObjectManager mgr = ctx["testObjectManager"] as
ITestObjectManager;

TestObject to1 = new TestObject();
to1.Name = "Jack";
to1.Age = 7;

TestObject to2 = new TestObject();
to2.Name = "Jill";
to2.Age = 8;

mgr.SaveTwoTestObjects(to1, to2);

mgr.DeleteTwoTestObjects("Jack", "Jill");
```

下面是 DAO 和事务管理器的对象定义：

```
<objects xmlns='http://www.springframework.net'
         xmlns:db="http://www.springframework.net/database">

    <db:dbProvider id="DbProvider"
                   provider="SqlServer-1.1"
                   connectionString="Data
Source=(local);Database=Spring;User
ID=springqa;Password=springqa;Trusted_Connection=False"/>
```

```
    <object id="transactionManager"
            type="Spring.Data.AdoPlatformTransactionManager,
Spring.Data">

        <property name="DbProvider" ref="DbProvider"/>
    </object>

    <object id="adoTemplate" type="Spring.Data.AdoTemplate,
Spring.Data">

        <property name="DbProvider" ref="DbProvider"/>
    </object>

    <object id="testObjectDao" type="Spring.Data.TestObjectDao,
Spring.Data.Integration.Tests">
        <property name="AdoTemplate" ref="adoTemplate"/>
    </object>


    <!-- The object that performs multiple data access operations -->

    <object id="testObjectManager"
            type="Spring.Data.TestObjectManager,
Spring.Data.Integration.Tests">
        <property name="TestObjectDao" ref="testObjectDao"/>
    </object>


</objects>
```

这是标准的 Spring 配置文件，我们可以很灵活的对连接字符串做参数化处理，也能很容易的更换 DAO 对象。下面的配置为事务管理器创建事务代理。

```
    <!-- The rest of the config file is common no matter how many objects
you add -->
    <!-- that you would like to have declarative tx management applied
to -->

    <object id="autoProxyCreator"

type="Spring.Aop.Framework.AutoProxy.DefaultAdvisorAutoProxyCreator,
Spring.Aop">
```

```
    </object>

    <object id="transactionAdvisor"

type="Spring.Transaction.Interceptor.TransactionAttributeSourceAdviso
r, Spring.Data"
            autowire="constructor">
    </object>


    <!-- Transaction Interceptor -->
    <object id="transactionInterceptor"

type="Spring.Transaction.Interceptor.TransactionInterceptor,
Spring.Data">
        <property name="TransactionManager" ref="transactionManager"/>

        <property name="TransactionAttributeSource"
ref="attributeTransactionAttributeSource"/>
    </object>

    <object id="attributeTransactionAttributeSource"

type="Spring.Transaction.Interceptor.AttributesTransactionAttributeSo
urce, Spring.Data">
    </object>
```

这段配置确实比较冗长，也有点难以理解——不过我们可以把这段代码作为模板应用在多个项目中。其中的对象定义是请求 Spring 在 IoC 容器中查找所有应用了[Transaction]特性的对象，并根据特性中的事务选项来向这些对象应用 AOP 事务通知。在此特性既作为切入点，也用于定义必要的事务选项。

因为这段 XML 代码独立于所有对象定义之外，我们可以用一个单独的文件来保存，并通过<import> 节点来引用该文件。请参考 4.15, 从其它文件中导入对象定义。

Note

将来 Spring.NET 会用专门的事务 schema 来降低这段配置的复杂性。Java 版中已经做到了这一点。如果使用事务 schema，配置文件的后半部分就可以缩减为：

```
<!-- enable the configuration of transactional behavior based on
attributes -->

<tx:attribute-driven transaction-manager="transactionManager"/>
```

NOTE: Custom schema not yet implemented - just a sneak peak ;)

如果对此感兴趣，可以先参考一下 Spring.Java 的文档。

14.5.3.Transaction 特性的设置

Transaction 特性是一种元数据，用来说明类或方法必须运行在事务环境中。该特性的默认设置如下：

- 事务传播默认为 TransactionPropagation.Required
- 隔离级别默认为 IsolationLevel.Unspecified
- 事务默认是可读/可写的
- 事务的超时时间默认为后台事务系统的超时时间，如果事务后台不支持超时则事务也不会超时
- 所有异常都会触发自动回滚

当然，默认的设置是可以更改的；下表就是 Transaction 特性的常用属性：

表 14.1.?Transaction 特性的属性

属性	类型	描述
TransactionPropagation	枚举 Spring.Transaction.TransactionPropagation	可选，事务传播方式，有效值为：Required, Supports, Mandatory, RequiresNew, NotSupported, Never, Nested
Isolation	System.Data.IsolationLevel	可选，隔离级别
ReadOnly	boolean	表明事务是读/写事务还是只读事务
Timeout	int (单位是秒)	事务的超时值
RollbackFor	Type 对象的数组	可选，维护一个异常类型的数组，如果这些异常抛出，必定自动回滚
NoRollbackFor	Type 对象的数组	可选，维护一个异常类型的数组，如果这些异常抛出，必定不会引起回滚

请注意我们计划在 RC1 版中支持为单个方法修改隔离级别。至于该功能何时会出现在 Nightly build 版本中，请留意论坛上的新闻。

如果将某个异常类型设为“NoRollbackFor”，那么在该类异常发生前已完成的工作将提交给数据库。而异常本身仍会在调用代码中传播。

在编码时，我们假定所有操作都能成功完成，并通过抛出异常来表明操作失败。当异常抛出时，事务管理框架根据事务选项决定是否回滚。这种编程模型要比完全依靠编程方式（借助 ContextUtil.MyTransactionVote 属性或 TransactionScope.Complete 方法）为好。回滚规则根据异常类型来处理事务结果，进一步提高了灵活性。.

如果让所有异常都触发回滚，就和在使用.NET 企业服务时向方法应用 AutoComplete 特性后的行为相似。所不同者，使用 AutoComplete 特性就意味着将对象与 ServicedCompent 的生命周期绑定在了一起，因为 AutoComplete 特性会将 ContextUtil.DeactivateOnReturn 属性设为 true。对无状态的 DAO 层来说这算不得什么，但在其它情况下则有可能出现问题。Spring.NET 的事务管理不会影响对象的生命周期。

14.5.4. 通过自动代理使用声明式事务

有两个自动代理类可用来进行声明式事务管理，分别是 ObjectNameAutoProxyCreator 和 DefaultAdvisorAutoProxyCreator。

14.5.4.1. 使用 ObjectNameAutoProxyCreator 创建事务代理

如果要为多个对象创建事务代理，ObjectNameAutoProxyCreator 非常适合。TransactionInterceptor 的对象定义和相关的特性只需定义一次。如果希望为某个对象创建事务代理，只需将其加入 ObjectNameAutoProxyCreator 的引用列表即可。下面是一个例子。另外请参考用 ProxyFactoryObject 创建 TransactionInterceptor 的相关章节。

```
    <object name="autoProxyCreator"

type="Spring.Aop.Framework.AutoProxy.ObjectNameAutoProxyCreator,
Spring.Aop">


        <property name="InterceptorNames"
value="transactionInterceptor"/>
        <property name="ObjectNames">
            <list>
                <idref local="testObjectManager"/>
            </list>
```

```
        </property>

    </object>
```

14.5.4.2. 使用 DefaultAdvisorAutoProxyCreator 创建事务代理

这是配置声明式事务的常用方式，通过这种方式，可以将[Transaction]特性作为切入点，来应用 IoC 容器中为所有对象定义的事务通知。

（按：原文中多次提到第五章的例子，这是 Java 版参考文档的编目，译文暂时删去了相关的内容，目前 Spring.NET 参考文档中并没有相关的例子。）

14.5.5. 通过 TransactionProxyFactoryObject 使用声明式事务

在很多情况下 TransactionProxyFactoryObject 比 ProxyFactoryObject 易用，因为该类可以通过自身属性来指定事务通知和事务特性，所以不需要单独为事务通知定义对象。另外，与使用 ProxyFactoryObject 不同，TransactionProxyFactoryObject 不要求使用方法的全名，只用普通的“短”方法名即可。同时，该类可以对方法名进行 wild card matching，从而强制我们为 DAO 的方法使用统一命名规则。我们通过下面的例子来看 TransactionProxyFactoryObject 的用法。

```
    <object id="testObjectManager"

type="Spring.Transaction.Interceptor.TransactionProxyFactoryObject,
Spring.Data">

        <property name="PlatformTransactionManager"
ref="adoTransactionManager"/>
        <property name="Target">

            <object type="Spring.Data.TestObjectManager,
Spring.Data.Integration.Tests">
                <property name="TestObjectDao" ref="testObjectDao"/>
            </object>
        </property>
        <property name="TransactionAttributes">
            <name-values>

                <add key="Save*" value="PROPAGATION_REQUIRED"/>
                <add key="Delete*" value="PROPAGATION_REQUIRED"/>
            </name-values>
        </property>
```

```
    </object>
```

注意 TestObjectManager 的内联定义可以保证外界无法引用未应用事务通知的 TestObjectManager 对象。

14.5.6. 使用抽象对象定义

将 TransactionProxyFactoryObject 配置为抽象对象定义有助于重用相同的配置信息，避免重复的为每个对象定义类似的 TransactionProxyFactoryObject。被代理的对象一般应该有相同的方法命名模式，比如 Save*，Find*等等。将这些相同点放在一个抽象的 TransactionProxyFactoryObject 对象定义中，让其它 TransactionProxyFactoryObject 对象定义去引用它，并根据需要来修改配置。下面是一个抽象 TransactionProxyFactoryObject 对象定义：

```
   <object id="txProxyTemplate" abstract="true"

type="Spring.Transaction.Interceptor.TransactionProxyFactoryObject,
Spring.Data">



        <property name="PlatformTransactionManager"
ref="adoTransactionManager"/>

        <property name="TransactionAttributes">
            <name-values>
                <add key="Save*" value="PROPAGATION_REQUIRED"/>
                <add key="Delete*" value="PROPAGATION_REQUIRED"/>
            </name-values>

        </property>
    </object>
```

其它 TransactionProxyFactoryObject 对象定义可以引用这个“父”配置，如下：

```
<object id="testObjectManager" parent="txProxyTemplate">
    <property name="Target">
            <object type="Spring.Data.TestObjectManager,
Spring.Data.Integration.Tests">
                <property name="TestObjectDao" ref="testObjectDao"/>

            </object>
    </property>
</object>
```

14.5.7. 通过 ProxyFactoryObject 进行声明式事务管理

使用通用的 ProxyFactoryObject 来声明事务代理，可以对生成的代理有更多的控制权，因为我们可以让 ProxyFactoryObject 引用其它通知，比如日志通知和性能通知等。我们仍以前面的例子来说明如何用 ProxyFactoryObject 声明事务代理。

```
    <object id="testObjectManagerTarget"
type="Spring.Data.TestObjectManager, Spring.Data.Integration.Tests">
        <property name="TestObjectDao" ref="testObjectDao"/>
    </object>

    <object id="testObjectManager"
type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">

        <property name="Target" ref="testObjectManagerTarget"/>
        <property name="ProxyInterfaces">
            <value>Spring.Data.ITestObjectManager</value>
        </property>

        <property name="InterceptorNames">
            <value>transactionInterceptor</value>
        </property>

    </object>
```

ProxyFactoryObject 会为目标对象创建代理，这里目标对象是一个 TestObjectManager 实例。内联对象定义仍用来保证外界无法直接获得未经代理的目标对象。其中引用的拦截器定义如下：

```
    <object id="transactionInterceptor"
type="Spring.Transaction.Interceptor.TransactionInterceptor,
Spring.Data">

        <property name="TransactionManager"
ref="adoTransactionManager"/>

        <!-- note do not have converter from string to this property type
registered -->
        <property name="TransactionAttributeSource"
ref="methodMapTransactionAttributeSource"/>
    </object>

    <object name="methodMapTransactionAttributeSource"
```

```
type="Spring.Transaction.Interceptor.MethodMapTransactionAttributeSou
rce, Spring.Data">
        <property name="MethodMap">

           <dictionary>
              <entry
key="Spring.Data.TestObjectManager.SaveTwoTestObjects,
Spring.Data.Integration.Tests"
                     value="PROPAGATION_REQUIRED"/>
              <entry
key="Spring.Data.TestObjectManager.DeleteTwoTestObjects,
Spring.Data.Integration.Tests"
                     value="PROPAGATION_REQUIRED"/>
           </dictionary>
        </property>
    </object>
```

在对象定义 methodMapTransactionAttributeSource 中，通过一个字典为每个方法指定事务选项，字段的键值为"类型全名.方法名, 程序集名"，值为如下形式：

- <Propagation Behavior>, <Isolation Level>, <ReadOnly>, -Exception, +Exception

其中除 Propagation Behavior 外都是可选的。异常类型前的减号表示如果此类异常抛出就回滚，加号表示如果此类异常抛出就提交。

14.6. 编程方式的事务管理

Spring.NET 提供了两种方法来进行编程式事务管理：

- 使用 TransactionTemplate 类
- 直接使用 IPlatformTransactionManager 的实现类

如果您要使用编程方式进行事务管理，Spring 团队推荐使用第一种方式（即使用 TransactionTemplate 类）。

14.6.1. 使用 TransactionTemplate

TransactionTemplate 的工作方式与 Spring.NET 的其它模板类如 AdoTemplate 和 HibernateTemplate 相同。它使用回调方法将应用程序的代码从样板式的准备工作和资源管理中解放出来。如果使用 System.Transaction 的话就必须自己处

理这些烦人的工作。使用 TransactionTemplate 时，提交是默认的行为，异常则会触发自动回滚，不需要使用 TransactionScope 来手工的进行提交或回滚。

同 Spring.NET 中其它模板类一样,TransactionTemplate 的对象是线程安全的。

如果需要在事务环境中执行，应用程序必须使用如下的代码。注意 ITransactionCallback 可以用来返回某个值：

```
TransactionTemplate tt = new TransactionTemplate(TransactionManager);

string userId = "Stewie";

object result = tt.Execute(delegate {
                              dao.UpdateOperation(userId);
                              return dao.UpdateOperation2();
                            });
```

代码中使用了匿名委托，所以只能在.NET 2.0 中运行，匿名委托可以使回调函数的调用更为优雅，在委托内部，可以引用本地变量，如上例中的 userId。本例的委托定义没有显式使用 ITransactionStatus 参数（delegate 关键字可以自己判断相应委托的方法签名。按：此处的委托类型为 delegate object DoInTransaction(ITransactionStatus status);），我们当然也可以在委托声明中显式使用此参数来获取 ITransactionStatus 的引用，然后设置它的 RollbackOnly 属性来触发回滚——或者抛出一个异常。如下所示：

TODO

在.NET 1.1 中，必须使用完整的 DoInTransaction 委托声明，或者使用 ITransactionCallback 接口的实现类。如下所示：

TODO

在应用程序中，如果一个类要使用事务，应该访问 TransactionTemplate 的 PlatformTransactionManager 属性（一般来说通过依赖注入进行设置）。该属性的类型是 IPlatformTransactionManager，这样就很容易用 IPlatformTransactionManager 的模拟实现进行单元测试。

14.6.2.使用 IPlatformTransactionManager

我们也可以直接使用 IPlatformTransactionManager 的实现类来管理事务。只需将 IPlatformTransactionManager 的引用注入给需要事务的对象，就可以用 TransactionDefinition 和 ITransactionStatus 来初始化事务、回滚和提交了。

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
```

```
def.PropagationBehavior = TransactionPropagation.Required;

ITransactionStatus status = transactionManager.GetTransaction(def);

try
{
  // execute your business logic here
} catch (Exception e)
{
  transactionManager.Rollback(status);
  throw;
}
transactionManager.Commit(status);
```

注意可用 TransactionManagerScope 来执行与 System.Transactions.TransactionScope 相似的操作。未来版本会支持这一功能。

14.7. 选择事务管理的方式

如果只有少量的事务操作，一般可以使用编程方式。例如，如果一个 Web 应用程序只需要为某些更新操作执行事务，估计我们都不大会为此去用 Spring.NET 或其它技术来配置事务代理。此时使用 TransactionTemplate 就很合适。另一方面，如果应用程序中需要执行大量的事务操作，就应该使用声明方式了，这样可以把事务管理移到业务逻辑之外，更何况在 Spring.NET 中配置事务也非常的简单。

第十五章. 数据访问对象

15.1. 简介

在数据访问方面，Spring.NET 大量使用了接口。这些接口封装了必要的数据存取操作，与业务领域相关的对象不需要再引用专门的持久化 API。在分层的系统架构中，一般用服务层响应特定的业务对象请求，并且用实现了这些接口的对象处理所有与持久化有关的行为。这些对象通常叫做数据访问对象（DAO），系统结构中相应的层次一般称为数据访问层（DAL）。

在应用程序中使用 DAO，能够提供跨持久化技术的灵活性及测试的便利性。DAO 可以使测试变得相当简单，因为在 NUnit 中很容易创建数据访问接口的模拟实现类，因此可以在没有数据库的环境下测试数据访问层的功能。对于测试来说，这是很有利的，因为依赖数据库的测试环境一般很难搭建起来，并且，如果需要测试异常行为，使用真实的数据库环境也不切实际。

Spring.NET 数据访问对象的设计目的是用统一的方式简化各种数据访问技术（如 ADO.NET 和 NHibernate）的应用。为此，Spring.NET 提供了两组核心功能。其一是与具体数据库无关的通用异常类；其二，通过数据访问对象的基类，提升了 ADO.NET 操作的抽象层次。通过这些功能，可以很轻松的在各种持久化技术之间切换，且编码时无需关心与具体技术有关的异常处理。

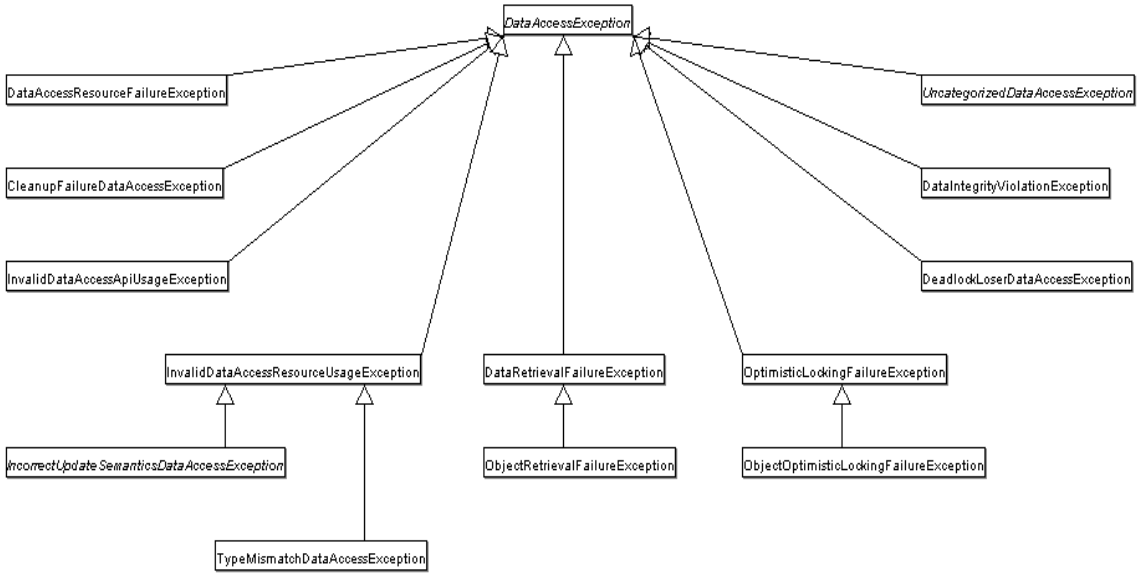
15.2. 统一的异常体系

ADO.NET 本身没有为不同的数据库定义统一的数据库异常，在.NET 1.1 中，FCL 甚至没有为 ADO.NET 定义公共的异常基类。所以，我们必须针对不同的数据库来处理具体的数据库异常，如 System.Data.SqlClient.SqlException 和 System.Data.OracleClient.OracleException 等。.NET 2.0 在这方面作了改进，为数据库异常加入了一个公共基类：System.Data.Common.DbException，但这个类却并不是很灵活，因为用于标识错误信息的错误代码还是与不同数据库相关的。对于同一种错误，比如数据完整性冲突和 SQL 语法错误，不同数据库返回的错误码也不相同。

为帮助开发人员编写灵活且更具可读性的异常处理代码，Spring.NET 将 ADO.NET 中与不同技术相关的异常“翻译”成了一系列自定义的异常类，它们的基类为 Spring.Dao.DataAccessException。这些异常在内部包装了原始的 ADO.NET 异常，所以不必担心会有信息丢失的情况。

除了 ADO.NET，Spring.NET 也对 NHibernate 的异常做了包装。开发人员可以在应用程序的各个层次处理大部分不可恢复的持久化异常，而不需要使用样板式的 catch/throw 语句或异常声明。上面讲过，（与不同数据库方言相关的）ADO.NET 异常被翻译成了 Spring.NET 的异常，所以可以用统一的编程模式来执行 ADO.NET 的常用操作。对于其它基于模板的 ORM 框架，Spring.NET 的处理方式也是一样的。

Spring.NET 使用的数据访问异常如下图所示：



（注意，这仅是 Spring.NET 数据访问异常的一部分）。

异常翻译的基础是 Spring.Data.Support.IAdoExceptionTranslator 接口：

```
public interface IAdoExceptionTranslator
{
    DataAccessException Translate( string task, string sql, Exception
exception );
}
```

其中 Translate 方法的参数分别为：task，为当前尝试执行的任务的描述信息；sql，为引起异常的 SQL 语句；exception，由 ADO.NET 抛出的“原始”异常。task 和 SQL 参数的目的是当异常发生时为开发人员提供一个可读性高的、清晰的错误描述信息。

该接口的默认实现是 ErrorCodeExceptionTranslator 类，该类使用了一个特殊的配置文件 sql-error-codes.xml，其中精确的定义了各种错误代码与 Spring.NET 数据访问异常之间的对应关系。在用 Spring.NET 创建数据访问层时，可以直接使用该类翻译异常。如果在数据访问对象中使用 Spring.NET 的 AdoTemplate 类，那么抛出的异常就会被自动转换。还有一种用法介于这两者之间：使用 ADO.NET2.0 的原生 API 和 Spring.NET 的声明式事务处理，并在自己的异常处理层使用 IAdoExceptionTranslator 接口。

下面列出一些比较常见的异常，其它异常请参考 API 文档.

表 15.1. 常见的数据访问异常

异常	描述

异常	描述
BadSqlGrammarException	当指定的 SQL 语句无效时抛出
DataIntegrityViolationException	在插入或修改数据时，若有完整性冲突，则抛出此异常。例如，插入一个重复的主键。
PermissionDeniedDataAccessException	当底层资源拒绝访问某个元素时，例如某些数据库表，抛出此异常。
DataAccessResourceFailureException	当对底层资源的访问完全失败时抛出，比如无法连接到数据库。

15.3. 抽象的数据访问对象基类

为用统一的方式简化各种数据库技术（如 ADO.NET、NHibernate 和 iBatis.NET 等）的应用，Spring.NET 定义了一系列抽象的 DAO 类，开发人员可以扩展这些类。通过这些抽象类的方法，我们可以获取诸如数据源以及与各种数据库技术相关的所有信息。

数据访问对象基类：：

- AdoDaoSupport：ADO.NET 数据访问对象的基类。子类可以和 DbProvider 及 AdoTemplate 类协作来完成数据操作。请参考下一章的相关内容。详细信息请参见 API 文档。
- HibernateDaoSupport：NHibernate 数据访问对象的基类。子类可以和 ISessionFactory 及 HibernateTemplate 类协作来完成数据操作。也可以直接通过 HibernateTemplate 创建数据访问对象，并重用 HibernateTemplate 的设置，比如 SessionFactory、flush 模式，异常翻译器等。与 NHibernate 相关的类库需要从 Spring.NET 网站上单独下载。

第十六章. DbProvider

16.1. 简介

Spring.NET 用一个通用的工厂类来创建 ADO.NET 各种类型的对象，比如 IDbConnection 和 IDbCommand 等。这个工厂类和.NET 2.0 中引入的工厂类（按：指 DbProviderFactory）很相似，只是加入了一些必要的元数据，以支持 Spring.NET 中 DAO/ADO.NET 框架所提供的功能如异常翻译等。工厂本身使用内嵌在 Spring.Data 程序集中的一个专用 XML 文件来配置，开发人员只需要关心如何使用该工厂类。目前该工厂类支持几个较为流行的数据库，通过扩展也可以支持新的数据库或修改原有的功能。Spring.NET 还针对数据库应用定义了专门的 Schema，可以简化数据库对象定义的配置工作。

与.NET 2.0 中的工厂类相比，DbProvider 的不足之处在于它只返回低层次的接口类型，而非 System.Data.Common 中的抽象基类。不过 DbProvider 仍可以弥补.NET 2.0 本身的一些不足。最明显的是在 ADO.NET 中，DbException 包含的错误信息是远程过程调用的 HRESULT，这样不直观的信息一般不是我们期望的。而 Spring.NET 可将错误码映射为统一的数据访问异常，并在其中包含引发错误的 SQL 语句。这样我们能很轻松的写出可移植的异常处理代码。另外，Spring.NET 的 DbParameter 类也不像用诸如 SqlClientFactory 等强类型工厂创建的 DbParameter 一样包含那么多便利方法。不过，如果我们需要使用 FCL 中抽象或接口类型的数据库对象，仍可通过 DbProvider 类直接创建。最后，对 FCL 数据库类型的这种包装可以让我们很方便的与 Spring.NET 事务管理相集成——因为在用 Spring.NET 创建 DbCommand 对象时，DbCommand 对象的 Connection 和 Transaction 属性已经根据事务调用环境预先设置好了。

16.1.1. IDbProvider 和 DbProviderFactory

IDbProvider 接口如下所示，如果使用过.NET 2.0 的 DbProviderFactory 类，就会发现这个接口与之相似。注意，Spring.NET 的 IDbProvider 接口也适用于.NET 1.1。

```
public interface IDbProvider
{
    IDbCommand CreateCommand();

    object CreateCommandBuilder();

    IDbConnection CreateConnection();

    IDbDataAdapter CreateDataAdapter();

    IDbDataParameter CreateParameter();

    string CreateParameterName(string name);

    IDbMetadata DbMetadata
    {
        get;
    }

    string ConnectionString
    {
        set;
        get;
    }
```

```
    string ExtractError(Exception e);

    bool IsDataAccessException(Exception e);

}
```

ExtractError 方法用于返回一个错误信息，在将异常翻译为 DAO 异常时会用到该信息。IsDataAccessException 方法用于在.NET 1.1 中判断抛出的异常是否与特定的数据库有关，因为.NET 1.1 没有为数据库异常定义公共基类。

DbProviderFactory 类利用给定的 Provider 名称创建 IDbProvider 对象。在为 IDbProvider 对象设置了连接字符串以后，就可以用它来创建 IDbConnection 对象。Provider 名称和对应的数据库如下所示。

- SqlServer-1.1:Microsoft SQL Server，provider V1.0.5000.0，用于.NET 框架 V1.1。
- SqlServer-2.0（System.Data.SqlClient 的别名）:Microsoft SQL Server，provider V2.0.0.0，用于.NET 框架 V2.0。
- OleDb-1.1：OleDb，provider V1.0.5000.0，用于.NET 框架 V1.1。
- OleDb-2.0（System.Data.OleDb 的别名）：OleDb，provider V2.0.0.0，用于.NET 框架 V2.0。
- OracleClient-2.0（System.Data.OracleClient 的别名）：Oracle，Microsoft provider V2.0.0.0。
- OracleODP-2.0（System.DataAccess.Client 的别名）：Oracle, Oracle provider V2.102.2.20。
- MySql：MySQL, MySQL provider 1.0.7.3007（按：Spring.NET 的当前版本（1.1 Preview3）需要 MySqlConnector 1.0.7.30072，您可以从 MySql 的网站上单独下载，当然您也可以修改 Spring.Data 以使用 1.08 RC 或 5.02beta 版。）

DbProviderFactory 的用法如下所示：

```
IDbProvider dbProvider =
DbProviderFactory.GetDbProvider("System.Data.SqlClient");
```

Provider 对象都定义在内嵌的程序集内嵌资源 assembly://Spring.Data/Spring.Data.Common/dbproviders.xml 中（按：参见 src\Spring\Spring.Data\Data\Common\dbproviders.xml 文件）。未来会加入对其它数据库的支持。目前因陋就简，我们倒是有一个方法可以添加新的 Provider，或者为 Provider 对象应用 Spring.NET 的其它功能如 AOP 通知等，方法是将 DbProviderFactory 的 DBPROVIDER_ADDITIONAL_RESOURCE_NAME 属性设置为包含其它 IDbProvider 对象定义的资源路径。该属性的默认值是 file://dbProviders.xml。将来会用 App.config/web.config 中的自定义配置节点来代替这一方法。

16.1.2. XML 配置

在配置文件中定义 DbProvider 的方法如下，一般我们会用它来设置 AdoTemplate 对象的 DbProvider 属性。（按：注意 objects 节点需要引用命名空间 xmlns:d="http://www.springframework.net/database"。）

```
<objects xmlns='http://www.springframework.net'
         xmlns:d="http://www.springframework.net/database">

  <d:dbProvider id="DbProvider"
                provider="System.Data.SqlClient"
                connectionString="Data
Source=(local);Database=Spring;User
ID=springqa;Password=springqa;Trusted_Connection=False"/>

  <object id="adoTemplate" type="Spring.Data.AdoTemplate,
Spring.Data">
    <property name="DbProvider" ref="DbProvider"/>
  </object>

</objects>
```

要使用上面的配置语法，还需要在主程序配置文件中注册一个自定义命名空间。如下面代码所示，注意其中只加入了 parsers 及相关的节点处理器：

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="parsers"
type="Spring.Context.Support.ConfigParsersSectionHandler,
Spring.Core" />
    </sectionGroup>

  </configSections>

  <spring>
    <parsers>
      <parser namespace="http://www.springframework.net/database"
              type="Spring.Data.DatabaseConfigParser, Spring.Data"

schemaLocation="assembly://Spring.Data/Spring.Data/spring-database.xs
d" />
    </parsers>
```

```
  </spring>

</configuration>
```

16.1.3.管理连接字符串

在 Spring.NET 中，有几种方法可用于管理连接字符串。

第一种是使用 IoC 容器的属性替换功能，参见：4.9.1 节,PropertyPlaceholderConfigurer 类。这样我们可以在配置文件中使用变量名作为占位符，并由 PropertyPlaceholderConfigurer 对象统一管理变量的值。下面的例子将连接字符串参数化了，当然也可以直接使用完整的字符串。

请看下面的例子：

```
<configuration>
  <configSections>

    <sectionGroup name="spring">
      <section name='context'
type='Spring.Context.Support.ContextHandler, Spring.Core'/>
    </sectionGroup>

    <section name="databaseSettings"
type="System.Configuration.NameValueSectionHandler, System,
Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
/>

  </configSections>

  <spring>
    <context>
      <resource uri="Aspects.xml" />
      <resource uri="Services.xml" />
      <resource uri="Dao.xml" />
    </context>

  </spring>

  <!-- These properties are referenced in Dao.xml -->
  <databaseSettings>
    <add key="db.datasource" value="(local)" />
    <add key="db.user" value="springqa" />
    <add key="db.password" value="springqa" />
```

```
    <add key="db.database" value="Northwind" />
  </databaseSettings>


</configuration>
```

在 Dao.xml 文件中定义的 DbProvider 对象引用了上面的连接字符串，如下：

```
<objects xmlns='http://www.springframework.net'
         xmlns:d="http://www.springframework.net/database">

  <d:dbProvider id="DbProvider"
                provider="System.Data.SqlClient"

connectionString="${db.datasource};Database=${db.database};User
ID=${db.user};Password=${db.password};Trusted_Connection=False"/>


  <object id="adoTemplate" type="Spring.Data.AdoTemplate,
Spring.Data">
    <property name="DbProvider" ref="DbProvider"/>
  </object>

</objects>
```

TODO: The second option is to use a custom schema specific to your database. The case of using SqlServer is shown below.

第十七章. 使用 ADO.NET 进行数据访问

17.1. 简介

Spring.NET 为使用 ADO.NET 进行数据访问提供了一系列抽象，其特性与优点包括：

- 针对.NET 不同版本（1.1 和 2.0）的全面、一致的 Data Provider 接口
- 与 Spring.NET 的事务管理功能完美集成。
- 可以用模板方式操作 DbCommand，不需要编写传统 ADO.NET 中那些样板式的代码。

- 框架负责执行数据库操作中最通用的部分，开发人员可以关注于 ADO.NET 代码的“实质内容”。
- 轻松创建和管理数据库参数。
- 与具体数据库无关的、包含错误码的数据访问异常，以及高层次的 DAO 异常体系。
- 以集中的方式管理 Connection、Command 和 Data Reader 等对象。
- 简单易用的 DataReader-Object 映射框架。

本章将针对 Spring.NET 数据访问的各项主要功能展开讨论，包括以下几个部分：

- 动机：通过与“原始的”ADO.NET 进行比较，让读者了解为什么可以考虑使用 Spring.NET 的数据访问功能。
- 数据库抽象：简要介绍 Spring.NET 的数据库抽象。
- 在 Spring.NET 中使用 ADO.NET 进行数据访问：讨论两种不同风格的 Spring.NET 数据访问类——基于模板和基于对象的。
- AdoTemplate 简介：介绍 Spring.NET 数据访问核心类型的设计思想和主要方法。
- 异常翻译：介绍 Spring.NET 数据访问的异常体系。
- 参数管理：介绍与参数管理相关的类型和方法。
- 基本的数据访问操作：讨论如何使用 AdoTemplate 类调用 IDbCommand 的 ExcecuteScalar 和 ExecuteNonScalar 方法。
- 数据查询和轻量级的对象映射：使用 AdoTemplate 将结果集映射为对象。
- 操作 DataSet 和 DataTable：使用 AdoTemplate 操作 DataSet 和 DataTable。
- 将 ADO.NET 操作建模为.NET 对象：用面向对象的方式处理数据访问操作。

17.2. 动机

出于多方面的原因，我们需要在 ADO.NET 的基础上创建一套高层次的持久化 API。

首先，我们希望在使用 ADO.NET 时，能把典型的“样板式”任务封装起来。下面是处理一个查询结果集时必需的操作步骤。如果使用 Spring.NET 的数据访问功能，只有斜体字部分是需要编码的。

1. 定义连接参数
2. 打开数据库连接
3. *指定命令的类型和语句*
4. 准备并执行查询语句
5. 设置用于迭代结果集（如果有结果返回）的循环操作
6. *执行所有迭代操作*
7. 处理发生的任何异常
8. 在遇到警告时显示信息或回滚操作
9. 处理事务
10. 关闭数据库连接

Spring.NET 会处理底层的任务，所以开发人员能够专注于 SQL 语句和数据处理的具体工作。这种标准的操作模式被封装进了 AdoTemplate 类中。之所以用 AdoTemplate 这个名字，是因为在用这段流程操作数据的时候，我们一般会创建一个包含资源管理、事务管理和异常管理等任务的模板，然后再将执行数据操作的代码嵌入到模板中。

第二，我们希望能用简单的方式在同一事务中执行多步 ADO.NET 操作，同时遵循数据访问对象（DAO）的设计风格，使事务能在数据访问对象之外（一般是在业务服务层）被初始化。在用"原始"ADO.NET 实现这种设计的时候，通常需要将"事务/连接"对象成对的传给 DAO 对象。这种与主要任务无关的底层操作其实可以"特殊"对待。利用 Spring.NET 的事务管理，可以用很优雅的方式实现这样的设计。Spring.NET 的事务管理可为我们带来很多便利，请参考第十四章,事务管理。

第三，使代码独立于具体的数据库：在.NET 1.1 中，由于种种原因，很难创建与具体数据库无关的代码。最主要的原因是缺乏一种集中的工厂机制为 ADO.NET 中的抽象引用（如 IDbConnection，IDbCommand，DbParameter 等等）创建对象。另外，很多接口的功能有限或是不完整，只能用具体的 Provider 子类将本应简单的任务写成臃肿无聊的代码。最后一点，没有提供一个与具体数据库无关的通用数据访问异常基类。.NET 2.0 在这方面已经改进了很多——所以 Spring.NET 只是提供了必要的补充，以帮助开发人员创建更为灵活的数据访问代码。

第四，资源管理：从代码的角度来说，.NET 中的 using 语句可以使资源管理变的非常优雅。但是，如果为每个数据访问方法嵌套地使用上两三个 using 语句也会让代码变的臃肿，此时不管是直接键入还是“复制—粘贴”，都难免有出错的可能。Spring.NET 将所有资源集中到一起进行管理，开发人员不会因忘记或做错某件事而导致程序异常。

第五，参数管理：在使用 ADO.NET 时，很多情况下都需要用大量代码创建正确的参数。为减少这些工作量，Spring.NET 提供了一个参数“Builder”类，可以简化参数的管理过程。同时，对于存储过程来说，由于参数列表可以从数据库本身获取，参数的创建就可以缩短为仅一行代码。

第六，对象映射：我们经常需要将结果集中的数据转换为某种领域对象。Spring.NET 提供了一个简单的框架来执行这些映射操作，并可以在应用程序中重用映射的结果。

第七,异常处理:当 ADO.NET 代码抛出异常时,标准的处理方法是查看错误代码、在抛出异常的代码处设置断点、重新运行应用程序，然后看看是哪些 SQL 语句或结果值导致了异常。Spring.NET 将不同数据库的错误翻译为一个统一的数据访问异常体系。这种机制能让开发人员快速的理解错误类型并找出代码中的问题。

第八，处理数据库警告：System.Data.Common 命名空间不支持从数据库中获取警告信息，也无法在必要时将警告信息作为回滚的原因。

第九，灵活性：只要有可能，就应该用高层次的数据访问 API 来提高代码的灵活性，Spring.NET 做到了这一点。高层 API 应该能够处理不同数据之间的差异，比如在定义参数时，SqlServer 使用字符@而 Oracle 使用字符:，将这些因具体数据库不同而引起的代码差异交由高层 API 处理，就可以使代码更具可移植性。

注意 Spring.NET 的数据访问框架只是建立在原始 ADO.NET 之上的一个“轻量级”框架，其目的并非是要和其它诸如结果集映射（iBATIS.NET）和 ORM 工具（NHibernate）等高层持久化框架一争高下。（如果前面列举动机或理由都不能引起您的兴趣，那么只能请您原谅了）。通常选择合适的抽象层总是我们在开发前必须要做的工作。在此提醒读者，Spring.NET 确实能和高层持久化抽象框架整合（目前可支持 NHibernate），可以在各种数据库访问技术中使用 Spring.NET 的事务管理，并能在同一事务中混合使用 ORM 和原始 ADO.NET 进行操作。

17.3.数据库抽象

在执行查询之前，必须要先连接到数据库。第十六章,DbProvider 已经详细讨论过这些话题，所以本章我们来看一看如何操作数据库。提高 ADO.NET 应用程序灵活性的一个重要因素是使用 ADO.NET 提供的接口来引用对象，比如 IDbCommand 或 IDbParameter 等等。但是在.NET 1.1 中为这些接口创建对象的惟一方法是直接实例化相关的类型，比如当使用 Sql Server 时：

```
IDbCommand command = new SqlCommand();
```

GoF 在其著作中提到的一个设计模式可以解决这个问题，即抽象工厂模式。.NET2.0 的 DbProviderFactory 类就应用了该模式来创建各种 ADO.NET 对象。另外.NET 2.0 还为所有的 Provider 对象添加了一个抽象基类。相比较原来的 ADO.NET 接口，该抽象基类的功能更为丰富，也可以用统一的方式访问不同的数据库。

Spring.NET 的数据库抽象也提供了类似于 ADO.NET2.0 的 DbProviderFactory 类的功能。数据库抽象的核心接口是 IDbProvider，它包含一系列工厂方法，这些方法的功能与 DbProviderFactory 中的方法类似，所不同的是，它们返回的类型是 ADO.NET 定义的基础接口。注意以 Spring.NET 框架的观点来看，既然 IDbProvider 是一个接口，就能很容易的在必要的时候创建一个模拟实现。该接口的另一个关键要素是 ConnectionString 属性，它可以指定与数据库相连时必要的连接信息。该接口还有一个 IDbMetadata 类型的属性，是给 Spring.NET 数据访问框架中的其它类型使用的，在开发时基本不会用到 DatabaseMetadata 类。

关于如何在 Spring.NET 中配置数据库 Provider，可以参考第十六章,DbProvider。

17.3.1.创建 IDbProvider 类型的实例

每种数据库都有其专门的 IDbProvider 接口实现。Spring.NET 已经为很多数据库创建了实现类，比如 Sql server，Oracle 和 MySql。请参考 Spring.NET 的文档以了解如何配置 Spring.NET 尚未直接支持的数据库。下面的代码用编程方式创建了一个 IDbProvider 类型的对象。

```
IDbProvider dbProvider =
DbProviderFactory.GetDbProvider("System.Data.SqlClient");
```

关于如何在 Spring.NET 的配置文件中创建 IDbProvider 的对象定义，请参考第十六章,DbProvider。

17.4. 命名空间

Spring.NET 的数据访问框架包括一系列命名空间，分别是：Spring.Data，Spring.Data.Generic，Spring.Data.Common，Spring.Data.Support，和 Spring.Data.Object。

Spring.Data 命名空间中包括了常用的接口和类型。

Spring.Data.Generic 命名空间为常用的大部分接口和类型定义了泛型版本，在.NET 2.0 下，可使用该命名空间。

Spring.Data.Common 命名空间包含通用的 Spring.NET 数据库类型和用于创建参数的工具类。

Spring.Data.Object 中的类将数据库查询、更新及存储过程封装为了一系列线程安全、可复用的对象。

最后，Spring.Data.Support 命名空间包含了用来进行异常翻译的接口（如 IAdoExceptionTransactor）和类，以及其它一些工具类型。

17.5. 数据访问方式

Spring.NET 可以用两种方式访问数据。第一种是基于“模板”的方法：创建一个 AdoTemplate 类的实例，然后供其它所有的 DAO 使用。模板类的方法可以用来执行数据访问的操作。DAO 的方法只需要对模板的方法进行调用。另一种方法是将数据库操作建模为对象，这是一种较为面向对象的方式。例如，我们可以通过 AdoQuery 类来封装数据库查询，通过 AdoNonQuery 类来封装创建、更新、删除的操作。同样，存储过程用 StoredProcedure 类进行封装。为使用这些类的功能，开发人员需要创建它们的子类，并在构造器或抽象方法中实现具体的操作。

经验表明，与基于对象的方式相比，使用 AdoTemplate 可以使 DAO 的方法变得更有条理，因为我们一般可以看到操作的细节。但对于存储过程则不然，因为 StoredProcedure 类可以自动从存储过程获取参数，并将一个可变的参数列表传递给“Execute”方法。所以，在应用之前我们可以比较一下这两种方法，以确定哪一种最能适合具体情况。

17.6.AdoTemplate 简介

AdoTemplate 类是 Spring.NET 数据访问框架的核心。该类基于回调方式实现，其核心方法 Execute()会根据回调方法所在的事务环境创建一个 IDbCommand 对象、设置它的 Connection 和 Transaction 属性并传递给回调方法。所有的资源管理工作都由框架负责，开发人员只需专注于如何在回调方法中用传来的 IDbCommand 进行操作。此外，还有一系列建立在 Execute 基础上的其它方法，用以在不同的场合下简化数据访问的过程。

AdoTemplate 有两个实现。位于 Spring.Data.Generic 命名空间下的实现是泛型版本。位于 Spring.Data 命名空间下的实现则是普通版本。这两个实现都要求在构造器中传入一个 IDbProvider 类型的对象：

```
AdoTemplate adoTemplate = new AdoTemplate(dbProvider);
```

AdoTemplate 是线程安全的，所以同一个 AdoTemplate 对象可以在指定的 IDbProvider 上执行所有数据访问操作。AdoTemplate 实现了 IAdoOperations 接口，虽然该接口一般用于测试，但我们也可以用它编码，以免直接使用具体类型。

下面两个小节讲解如何分别在.NET 1.1 和 2.0 中使用 AdoTemplate 的 Execute 方法。

17.6.1. 执行回调

Execute 方法和它的回调函数/接口是 AdoTemplate 的基础，AdoTemplate 的其余方法都建立在 Execute 方法之上。如果 AdoTemplate 类中没有可以直接满足要求的方法，我们还可以借助 Execute 方法执行任何数据库操作。此外，Execute 还能够满足某些特殊的需求，比如使用 XML 数据或 BLOB 字段。

17.6.2. 在.NET 2.0 中执行回调

本例使用 Northwind 数据库，用一个简单的查询返回使用某一邮政编码的客户数量。

```
public int FindCountWithPostalCode(string postalCode)
```

```
{
  return adoTemplate.Execute<int>(delegate(DbCommand command)
         {
             command.CommandText =
                 "select count(*) from Customers where PostalCode =
@PostalCode";

             DbParameter p = command.CreateParameter();
             p.ParameterName = "@PostalCode";
             p.Value = postalCode;
             command.Parameters.Add(p);

             return (int)command.ExecuteScalar();

          });
}
```

当 DbCommand 对象传递到匿名委托内部的时候，已经被 AdoTemplate 设置好了 Connection 属性。并且，AdoTemplate 还根据代码所在的事务环境为 DbCommand 对象设置好了 Transaction 属性。另外请注意匿名委托要访问的 postalCode 变量是定义在匿名委托的"外部"的。

我们可以看到，只需为与数据操作关系最密切的部分进行编码。另外，虽然也可以在回调中显式修改 DbCommand 的 Connection 属性，但不推荐这么做。

Execute 方法另有一个重载可以接受接口形式的回调。用于回调的委托和接口如下所示：

```
public delegate T CommandDelegate<T>(DbCommand command);

public interface ICommandCallback
{
  T DoInCommand<T>(DbCommand command);
}
```

使用委托可以使简化代码，而接口则有利于重用。AdoTemplate 中对应的两个 Execute 重载如下：

```
public class AdoTemplate : AdoAccessor, IAdoOperations
{
   ...

   T Execute<T>(ICommandCallback action);

   T Execute<T>(CommandDelegate<T> del);
```

```
    ...
}
```

17.6.3.?.NET 1.1

AdoTemplate 包括普通版本和泛型版本，这两个版本之间的区别在于 Execute 方法传递给其回调接口或委托的 Command 对象不同：在普通版本中，回调接口或委托获取的是 IDbCommand 接口；而在泛型版本中得到的则是具体的 DbCommand 对象。请参考 API 文档或源代码。因为.NET 1.1 不支持匿名委托，所以需要我们显式的定义委托方法或实现 ICommandCallback 接口。在.NET 1.1 中，查询客户的操作可用以下代码完成。

```
public virtual int FindCountWithPostalCode(string postalCode)
{
  return (int) AdoTemplate.Execute(new
PostalCodeCommandCallback(postalCode));
}
```

回调接口的实现为：

```
private class PostalCodeCommandCallback : ICommandCallback
{
    private string cmdText = "select count(*) from Customer where
PostalCode = @PostalCode";

    private string postalCode;

    public PostalCodeCommandCallback(string postalCode)
    {
        this.postalCode = postalCode;
    }

    public object DoInCommand(IDbCommand command)
    {
        command.CommandText = cmdText;

        IDbDataParameter p = command.CreateParameter();
        p.ParameterName = "@PostalCode";
        p.Value = postalCode;
        command.Parameters.Add(p);

        return command.ExecuteScalar();
```

```
    }
}
```

注意在这个例子中，我们也可以使用更为简单的 AdoTemplate.ExecuteScalar() 方法。

Execute 方法另有一个重载可以接受接口形式的回调。用于回调的委托和接口如下所示：

```
public delegate object CommandDelegate(IDbCommand command);

public interface ICommandCallback
{
  object DoInCommand(IDbCommand command);
}
```

使用委托可以使简化代码，而接口则有利于重用。AdoTemplate 中对应的两个 Execute 重载如下：

```
public class AdoTemplate : AdoAccessor, IAdoOperations
{
   ...

   object Execute(CommandDelegate del);

   object Execute(ICommandCallback action);

   ...

}
```

注意，由 Execute 返回的值需要转型为适当的类型。

17.6.4. AdoTemplate 方法指南

AdoTemplate 类有很多方法，在阅读 SDK 文档时可能会觉得有点不太好掌握。但是使用一段时间后就会发现这个类其实很容易用代码提示来“导航”。下面按名称和数据操作类型对这些方法进行了分组。每个方法都有自己的重载以处理不同的参数类型。

直接调用的普通方法：

- Execute：可以用标准的 DbCommand 对象执行任意数据访问操作。传入给回调方法的 DbCommand 对象的 Connection 和 Transaction 属性已经根据回调所在的事务环境设置好了。

下面两个方法分别对应 DbCommand 对象的同名方法：

- ExecuteNonQuery：用给定的参数执行 DbCommand 的同名方法，并返回此操作影响的行数。
- ExecuteScalar：用给定的参数执行 DbCommand 的同名方法，并返回结果集第一行第一列的值。

将结果集映射到对象：

- QueryWithResultSetExtractor：执行一个查询，利用 IResultSetExtractor 接口的实现类将结果集映射到对象中。
- QueryWithResultSetExtractorDelegate：同 QueryWithResultSetExtractor 一样，但使用 ResultSetExtractorDelegate 委托来执行结果集映射。
- QueryWithRowCallback：执行一个查询，并为结果集中的每一行调用 IRowCallback 接口的实现类。
- QueryWithRowCallbackDelegate：和 QueryWithRowCallback 一样，不同的是为每一行调用 RowCallbackDelegate 委托。
- QueryWithRowMapper：执行一个查询，并利用 IRowMapper 接口的实现类将结果集进行逐行映射。
- QueryWithRowMapperDelegate：同 QueryWithRowMapper 一样，但使用 RowMapperDelegate 委托将结果集的行映射到对象。

将结果集映射到单个对象：

- QueryForObject：执行一个查询，使用 IRowMapper 将结果映射到一个对象中。如果查询返回的结果不是一行，就会抛出异常。

下面的方法使用创建 DbCommand 对象的回调函数进行查询。通常由框架在内部调用这些方法以实现其它功能，例如 Spring.Data.Objects 命名空间的功能：

- QueryWithCommandCreator：执行一个查询，使用 IDbCommandCreator 类型的回调创建 IDbCommand 对象，并用 IRowMapper 或 IResultSetExtractor 将结果集映射到一个对象中。 其重载允许指定多个结果集“处理器”来处理多个结果集并返回 output 参数。

操作 DataTable 和 DataSet：

- DataTableCreate：创建并填充 DataTable。
- DataTableFill：填充一个创建好的 DataTable。

- DataTableUpdateWithCommandBuilder：使用给定的 DataTable、Select 语句和参数更新数据库。
- DataSetCreate：创建并填充 DataSet。
- DataSetFill：填充一个创建好的 DataSet。
- DataSetUpdateWithCommandBuilder：使用给定的 DataSet、Select 语句和参数更新数据库。

注意：泛型版本的 AdoTemplate 类目前不包含这些方法，但可以通过它的 ClassicAdoTemplate 属性来访问。请参考 API 文档。

创建参数的工具方法：

- DeriveParameters： 从存储过程中获取参数集合。

每个方法都有数个重载。除了操作 DataTable 和 DataSet 的方法外，其余方法可能具有下列形式的重载版本：

- MethodName(CommandType cmdType, string cmdText, CallbackInterfaceOrDelegate, parameter setting arguments)

其中包含一个 CallbackInterfaceOrDelegate 类型的参数。另外，“parameter setting arguments”的形式如下：

- MethodName(... string parameterName, Enum dbType, int size, object parameterValue
- MethodName(... IDbParameters parameters)
- MethodName(... ICommandSetter commandSetter)

如果只需要在一次调用中设置一个参数，可以使用第一个重载形式，参数 dbType 的类型为“Enum”，可以是任何与具体数据库相关的枚举，或是通用 DbType 枚举的任意值。实际上这是对类型安全和灵活性的一个折中（注意泛型版本可以提高类型的安全性）。

第二个重载形式可以设置一个参数集合。参数集合的类型是 Spring.NET 的 IDbParameters，稍后会讨论该类。

第三个重载用一个回调接口传递 IDbCommand 对象，开发人员可以直接设置 IDbCommand 对象的 Parameters 属性（或其它属性）。

在.NET 2.0 中，这些方法的委托版本非常好用，因为匿名委托可以使用同一作用域内的变量，所以在 DAO 的方法中，可以将与数据操作有关的定义集中到一起。这样就可以不必象接口回调一样用参数传来传去。这里有个选择的方法：如果不需要在多个 DAO 类型或方法中共享某些数据操作，就尽量使用委托；如果需要重用，就使用接口回调。.NET 2.0 会在适当的时候使用泛型，所以也增强了类型安全性。

17.7. 异常翻译

AdoTemplate 的方法抛出的异常属于数据访问对象（DAO）的异常体系，请参看第十五章,数据访问对象。另外，引起异常的命令文本以及异常的错误信息都会被提取并记录。由于异常并没有和具体的持久化技术紧耦合，所以很容器写出与具体数据库无关的异常处理层。对应 ADO.NET 代码来说，错误消息中所包含的 SQL 语句和错误代码对解决问题是很有帮助的。（下一版本会加入记录参数值的功能。）

17.8. 参数管理

在 ADO.NET 应用程序中，通常会用大量的代码来创建和填充参数。FCL 本身提供的参数接口功能十分有限，而且具体数据库（如 SqlClient）的参数实现类也没有提供多少有用的方法。就算是使用 SqlClient，也得用相当冗长的代码来创建和填充参数集合。Spring.NET 提供了两种方法，可以简化参数的管理工作，对于不同的数据库来说也更为灵活。

17.8.1. IDbParametersBuilder

相对于传统 ADO.NET 需要分别用一行代码负责创建参数、设置参数的类型和大小，IDbParametersBuilder 和 IDbParameter 接口能大幅的减少参数的管理工作量。IDbParamter 接口支持级联的链式调用，下面是一个例子。

```
IDbParametersBuilder builder = CreateDbParametersBuilder();
builder.Create().Name("Country").Type(DbType.String).Size(15).Value(country);
builder.Create().Name("City").Type(DbType.String).Size(15).Value(city);

// now get the IDbParameters collection for use in passing to AdoTemplate methods.
IDbParameters parameters = builder.GetParameters();
```

请注意 IDbParameters 和 IDbParameter 不是 FCL 的一部分，而是由 Spring.Data.Common 命名空间定义的。AdoTemplate 中的方法经常会用到 IDbParameters 集合。

参数名不需要包含参数定义的前缀，比如说 Sql Server 中的“@”。DbProvider 会根据具体的数据库类型确定正确的参数前缀，并由 AdoTemplate 在执行前将其添加到参数名称中。

另外，还可以将 IDbParametersBuilder 对象配置在 IoC 容器中作为生成 IDbParameters 对象的工厂对象。利用 Spring.NET 的表达式求值语言，可以用 XML 文件中的配置来代替上面几行代码。所以，我们可以将参数的定义移到代码之外。通过抽象对象定义和配置文件的导入功能，我们可以用同一套代码支持各种数据库，只需改变配置文件即可。在最终发布版本中将支持这一功能。

17.8.2. IDbParameters

这个接口与 ADO.NET 中的 IDataParameterCollection 接口很相似，但是它定义了很多方法，用以建立参数集合。

下面列举该接口的几个方法。

- int Add(object parameterValue)
- void AddRange(Array values)
- IDbDataParameter AddWithValue(string name, object parameterValue)
- IDbDataParameter Add(string name, Enum parameterType)
- IDbDataParameter AddOut(string name, Enum parameterType)
- IDbDataParameter AddReturn(string name, Enum parameterType)
- void DeriveParameters(string storedProcedureName)

请看一个例子：

```
// inside method has has local variable country and city...

IDbParameters parameters = CreateDbParameters();
parameters.AddWithValue("Country", country).DbType = DbType.String;
parameters.Add("City", DbType.String).Value = city;


// now pass on to AdoTemplate methods.
```

参数名不需要包含参数定义的前缀，也就是 Sql Server 中的“@”。DbProvider 会根据具体的数据库类型确定正确的参数前缀，并由 AdoTemplate 在执行前将其添加到参数名称中。

17.9. 映射空值(DBNull)

在使用 IDataReader 读取数据时，经常需要将 DBNull 值映射为某些默认值，也就是说将其转换为 null 或某个"魔术数字"比如-1。通常我们会用三元操作符来完成这一工作，但这不仅降低了可读性，也增加了出错的可能。Spring.NET 定义了一个 IDataReaderWrapper 接口（扩展了标准的 IDataReader），我们可以通过实现该接口来创建一个能够以统一的、非侵入（对于操作数据的代码而言）

的方式映射 DBNull 值的 DataReader。Spring.NET 已经创建了一个默认的实现类 NullMappingDataReader，我们可以在需要时继承此类或直接实现 IDataReaderWrapper，该接口的定义如下：

```
public interface IDataReaderWrapper : IDataReader
{
    IDataReader WrappedReader
    {
        get;
        set;
    }
}
```

如果将一个 IDataReaderWrapper 对象设置给 AdoTemplate 的 DataReaderWrapperType 属性，那么 AdoTemplate 中所有包含 IDataReader 类型参数的回调接口和委托都会被包装进这个 IDataReaderWrapper。此时，要求该 IDataReaderWrapper 的实现类必须定义默认的无参构造器。

通常在整个应用程序中我们会使用统一的 DBNull 映射方式，所以我们只需要一个 AdoTemplate 和一个 IDataReaderWrapper 实例。如果需要使用多种 DBNull 映射策略，则需要创建多个 AdoTemplate 对象并将它们配置给合适的数据访问对象。

17.10. 基本数据访问操作

AdoTemplate 的 ExecuteNonQuery 和 ExecuteScalar 方法与 DbCommand 对象的同名方法功能相同。

TODO

17.10.1. ExecuteNonQuery

下面是该方法的用法示例：

```
public void CreateCredit(float creditAmount)
{
    AdoTemplate.ExecuteNonQuery(CommandType.Text,
            String.Format("insert into Credits(creditAmount) VALUES
({0})",
            creditAmount));
}
```

TODO

17.10.2. ExecuteScalar

下面是该方法的用法：

```
int iCount = (int)adoTemplate.ExecuteScalar(CommandType.Text, "SELECT COUNT(*) FROM TestObjects");
```

TODO

17.11. 查询和轻量级对象映射

在常规的 ADO.NET 开发中，一般要将数据读取到结果集，再将结果集转换为领域对象。AdoTemplate 中的一系列 QueryWith 方法可以协助完成这些工作。映射操作一般由某个回调接口或委托完成，Spring.NET 定义了三组接口和委托用于回调，开发人员需要自己实现或定义这些回调方法。

- IResultSetExtractor/ResultSetExtractorDelegate：传递一个 IDataReader 对象，开发人员可以利用该对象迭代数据并返回结果对象。
- IRowCallback/RowCallbackDelegate：传递一个 IDataReader 对象，可用于处理当前行。返回值为 void，因此在实现 IRowCallback 接口时一般将类实现为有状态的（按：也就是说实现类需要定义自己的字段来保存回调的结果），在使用匿名委托时，一般用本地变量来保存回调的结果。
- IRowMapper/RowMapperDelegate：传递一个 IDataReader 对象，用以处理当前行并返回一个对应于当前行的对象。

IResultSetExtractor 和 IRowMapper 接口/委托还有对应的泛型版本，相对于普通版本更能保证类型安全。

这里接口或委托的实现和 Spring.Data 中其他情况一样，只需要关心代码的核心逻辑——映射数据，而框架会负责数据迭代和资源管理。

每个 QueryWith 方法都有 4 个重载版本，以处理不同的命令类型。

下一小节将详细讨论 Spring.NET 的轻量级对象映射框架。

17.11.1. ResultSetExtractor

ResultSetExtractor 类可以控制 IDataReader 查询结果的迭代方式。开发人员负责迭代所有结果集并返回对应的结果对象。IResultSetExtractor 的实现类一

般是无状态的，所以只要没有访问有状态的资源就可以一直重用它。框架会负责关闭 IDataReader。

下面是 IResultSetExtractor 接口和 ResultSetExtractorDelegate 委托的泛型版本，定义在 Spring.Data.Generic 命名空间中：

```
public interface IResultSetExtractor<T>
{
  T ExtractData(IDataReader reader);
}

public delegate T ResultSetExtractorDelegate<T>(IDataReader reader);
```

非泛型版本如下：

```
public interface IResultSetExtractor
{
    object ExtractData(IDataReader reader);
}

public delegate object ResultSetExtractorDelegate(IDataReader reader);
```

下面这个例子节选自 Spring.Data 快速入门程序。其中的方法定义在一个 DAO 类中，该类继承了 AdoDaoSupport，AdoDaoSupport 中的 CreateDbParametersBuilder 方法可以用来创建 IDbParametersBuilder 对象。

```
public virtual IList<string>
GetCustomerNameByCountryAndCityWithParamsBuilder(string country,
string city)
{

    IDbParametersBuilder builder = CreateDbParametersBuilder();

builder.Create().Name("Country").Type(DbType.String).Size(15).Value(c
ountry);

builder.Create().Name("City").Type(DbType.String).Size(15).Value(city
);
    return AdoTemplate.QueryWithResultSetExtractor(CommandType.Text,

customerByCountryAndCityCommandText,
                                                    new
CustomerNameResultSetExtractor<List<string>>(),
```

```
builder.GetParameters());
}
```

下面是 IResultSetExtractor 接口的实现类。

```
internal class CustomerNameResultSetExtractor<T> :
IResultSetExtractor<T> where T : IList<string>, new()
{

    public T ExtractData(IDataReader reader)
    {
        T customerList = new T();
        while (reader.Read())
        {
            string contactName = reader.GetString(0);
            customerList.Add(contactName);
        }
        return customerList;
    }
}
```

在框架内部，QueryWithRowCallback 和 QueryWithRowMapper 方法是 ResultSetExtractor 的一个特例。例如，QueryWithRowMapper 方法可以迭代结果集，为其中的每一行调用回调方法 MapRow，并将结果保持到一个 IList 中。如果 QueryWithXXX 方法不能满足某些特殊要求，我们可以创建 AdoTemplate 的子类，用相同的方式创建新的 QueryWithXXX 方法来实现我们的功能。

17.11.2. RowCallback

RowCallback 一般是个有状态的对象，或是用它来填充调用代码中某个有状态的对象。下面的例子节选自 Spring.Data 快速入门程序：

```
public class RowCallbackDao : AdoDaoSupport
{
    private string cmdText = "select ContactName, PostalCode from
Customers";

    public virtual IDictionary<string, IList<string>>
GetPostalCodeCustomerMapping()
    {
```

```
        PostalCodeRowCallback statefullCallback = new
PostalCodeRowCallback();
        AdoTemplate.QueryWithRowCallback(CommandType.Text, cmdText,
                                         statefullCallback);

        // Do something with results in stateful callback...
        return statefullCallback.PostalCodeMultimap;
    }

}
```

下面的 PostalCodeRowCallback 类具有自己的状态，并可以通过 PostalCodeMultimap 属性访问：

```
internal class PostalCodeRowCallback : IRowCallback
{
    private IDictionary<string, IList<string>> postalCodeMultimap =
        new Dictionary<string, IList<string>>();

    public IDictionary<string, IList<string>> PostalCodeMultimap
    {
        get { return postalCodeMultimap; }
    }

    public void ProcessRow(IDataReader reader)
    {
        string contactName = reader.GetString(0);
        string postalCode = reader.GetString(1);
        IList<string> contactNameList;
        if (postalCodeMultimap.ContainsKey(postalCode))
        {
            contactNameList = postalCodeMultimap[postalCode];
        }
        else
        {
            postalCodeMultimap.Add(postalCode, contactNameList = new
List<string>());
        }
        contactNameList.Add(contactName);
    }
}
```

17.11.3. RowMapper

IRowMapper 可以让开发人员专注于如何将结果集中的一行映射为一个对象。框架会负责使用 IDataReader 进行迭代并创建一个 IList 来保存结果对象。下面这个例子节选自 Spring.Data 快速入门程序：

```
    public class RowMapperDao : AdoDaoSupport
    {
        private string cmdText = "select Address, City, CompanyName,
ContactName, " +
                                 "ContactTitle, Country, Fax, CustomerID,
Phone, PostalCode, " +
                                 "Region from Customers";


        public virtual IList<Customer> GetCustomers()
        {
            return AdoTemplate.QueryWithRowMapper(CommandType.Text,
cmdText,
                                                  new
CustomerRowMapper<Customer>());
        }
    }
```

IRowMapper 的实现类如下：

```
    public class CustomerRowMapper<T> : IRowMapper<T> where T : Customer,
new()
    {
        public T MapRow(IDataReader dataReader, int rowNum)
        {
            T customer = new T();
            customer.Address = dataReader.GetString(0);
            customer.City = dataReader.GetString(1);
            customer.CompanyName = dataReader.GetString(2);
            customer.ContactName = dataReader.GetString(3);
            customer.ContactTitle = dataReader.GetString(4);
            customer.Country = dataReader.GetString(5);
            customer.Fax = dataReader.GetString(6);
            customer.Id = dataReader.GetString(7);
            customer.Phone = dataReader.GetString(8);
            customer.PostalCode = dataReader.GetString(9);
            customer.Region = dataReader.GetString(10);
            return customer;
```

```
        }
    }
```

如果映射的逻辑比较少，并且需要使用访问本地变量的时候，使用委托会更加方便：

```
        public virtual IList<Customer> GetCustomersWithDelegate()
        {
            return
AdoTemplate.QueryWithRowMapperDelegate<Customer>(CommandType.Text,
cmdText,
                        delegate(IDataReader dataReader, int rowNum)
                            {
                                Customer customer = new Customer();
                                customer.Address =
dataReader.GetString(0);
                                customer.City =
dataReader.GetString(1);
                                customer.CompanyName =
dataReader.GetString(2);
                                customer.ContactName =
dataReader.GetString(3);
                                customer.ContactTitle =
dataReader.GetString(4);
                                customer.Country =
dataReader.GetString(5);
                                customer.Fax =
dataReader.GetString(6);
                                customer.Id = dataReader.GetString(7);
                                customer.Phone =
dataReader.GetString(8);
                                customer.PostalCode =
dataReader.GetString(9);
                                customer.Region =
dataReader.GetString(10);
                                return customer;
                            });
        }
```

17.11.4. 查询单个对象

如果只需要结果集返回一条记录，可以用 QueryForObject 方法将这条记录映射到一个对象中去，若结果集中的记录不是一条，会抛出

Spring.Dao.IncorrectResultSizeDataAccessException 异常。下面这个例子节选自 Spring.Data 快速入门程序：

```
    public class QueryForObjectDao : AdoDaoSupport
    {
        private string cmdText = "select Address, City, CompanyName,
ContactName, " +
                         "ContactTitle, Country, Fax, CustomerID, Phone,
PostalCode, " +
                         "Region from Customers where ContactName =
@ContactName";

        public Customer GetCustomer(string contactName)
        {
            return AdoTemplate.QueryForObject(CommandType.Text,
cmdText,
                                              new
CustomerRowMapper<Customer>(),
                                              "ContactName",
DbType.String, 30, contactName);
        }
    }
```

17.11.5. Query using a CommandCreator

TODO

17.12. DataTable and DataSet

AdoTemplate 定义了一些方法来操作 DataTable 和 DataSet。目前要了解它们的用法请参考 SDK 文档。

17.12.1. DataTables

TODO

17.12.2. DataSets

TODO

17.13. Deriving Stored Procedure Parameters

TODO

17.14. Database operations as Objects

TODO

17.14.1. AdoNonQuery

TODO

17.14.2. AdoQuery

TODO

17.14.3. MappingAdoQuery

TODO

17.14.4. Stored Procedure

StoredProcedure 类可以用非常少的代码调用存储过程并返回调用结果。由于参数可以用编程方式获取并缓存，所以可以少写很多样板代码。不过如果您喜欢，自然也可以显式的手工定义参数。

AdoTemplate 可以通过三种方式执行存储过程。

- ExecuteScalar
- ExecuteNonQuery
- 使用 Spring.NET 的结果映射框架

这些方法的返回值都是 IDictionary 类型，其中包含输出参数和/或所有从 Spring.NET 结果映射框架返回的值。这些方法的参数或者是变长的——此时参数的顺序必须和存储过程中的参数顺序一致；或者是包含键值对的 IDictionary。使用 IDictionary 参数的方法名都以“ByNamedParamter”为后缀。

我们来看一个例子。下面的 CustOrdersDetailStoredProc 类会调用 Northwind 数据库中的 CustOrdersDetail 存储过程，向其中传入 OrderID 作为参数，并返回 OrderDetails 对象的集合：

```
    public class CustOrdersDetailStoredProc : StoredProcedure
    {
        private static string procedureName = "CustOrdersDetail";

        public CustOrdersDetailStoredProc(IDbProvider dbProvider) :
base(dbProvider, procedureName)
        {
            DeriveParameters();
            AddRowMapper("orderDetailRowMapper", new
OrderDetailRowMapper() );
            Compile();
        }

        public virtual IList GetOrderDetails(int orderid)
        {
            IDictionary outParams = Query(orderid);
            return outParams["orderDetailRowMapper"] as IList;
        }

    }
```

DeriveParameters 方法可以使开发人员不必显式的声明每个参数。通常在用变长参数调用 Query 方法时需要使用 DeriveParameters 方法。如果您不喜欢变长参数这种松散的方式，可以调用 QueryByNamesParameters 方法，传递一个包含参数键值对的 IDictionary 参数。

StoredProcedure 一旦“编译”之后就是线程安全的，注意这里讲的“编译”是指在构造器中的一个操作过程。StoredProcedure 会将参数缓存起来以供所有的 Query 方法调用。用来提取业务对象的 IRowMapper 实现类被“注册”到 StoredProcedure，随后可以通过名称获取它，就象是一个输出参数一样。同样，也可以通过 AddRowCallback 和 AddResultSetExtractor 方法注册 IRowCallback 和 IResultSetExtractor 回调接口。

17.14.5. DataSetOperation

第十八章. ORM 集成

18.1. 简介

与 NHibernate 1.0 和 1.2 进行集成是 Spring.NET 一个独立的模块项目。请在网站上查阅详细信息。

第十九章. Web 框架

19.1. 简介

很多开发人员不喜欢 ASP.NET 是因为它不是“真正的 MVC”（Model-View-Controller），因为控制器的逻辑与页面视图耦合的太过紧密。Page 类的事件处理器就很能说明这个问题：一般来说，ASP.NET 会在事件处理器中直接引用视图元素，比如输入控件等。在此我们不打算从理论上讨论什么是“真正的 MVC”，也不去管使用“看上去更美”的 MVP 或表示层(Presentation)模型将 ASP.NET 这种基于窗体的技术转化为基于请求的传统 MVC 模式是否适合，在最为重要的一点上，我们同意批评家们的看法：控制器的逻辑，比如写在 ASP.NET 页面事件处理器中的代码，不应该依赖于视图元素。

不过话说回来，ASP.NET 其实有很多好的方面。服务端窗体和控件大幅度提高了开发人员的效率，并简化了页面的（HTML）标记。同时，由于每个控件都知道怎样根据用户的浏览器来输出正确的 HTML 标记，所以很多跨浏览器的问题也变得很容易处理。ASP.NET 可以将自定义逻辑与页面的生命周期挂钩，也支持自定义 HTTP 处理管道等等非常强大的功能。最后，ASP.NET 为 Web 开发定义了一个相当有用的抽象层，这使开发人员可以和强类型的服务端控件进行交互，而不必去处理诸如 Form 和 QueryString 这种基于字符串的 HTTP 请求。

由于 ASP.NET 本身的这些优点，我们决定不在 Spring.NET 中重新开发一套“纯粹、真正的 MVC”的 Web 框架，而采用一种更为实际的方式：扩展 ASP.NET，以消除其中的大部分弊端。

前面讲过，code-behind 类中的事件处理器不应该直接处理 ASP.NET 的 UI 控件。事件处理器应该操作页面的表示层模型，比如领域对象或是 ADO.NET 的 DataSet 对象。为此，Spring.NET 实现了一个双向数据绑定的框架，来处理页面控件与数据模型之间的映射关系。数据绑定框架还可以透明的进行数据类型转换和输出的格式化，所以开发人员可以在事件处理器中使用强类型的数据（领域）对象。

在企业级应用中，应用程序的控制流程也是一个需要关注的问题，Spring.NET 的 Web 框架对此也有相应的解决方案。在传统的 ASP.NET 应用程序中，开发人员一般会在某个行为执行后调用 Response.Redirect 或 Server.Transfer 方法跳转到其它页面。一般来说这会导致在页面中对目标 URL 进行硬编码。Spring.NET 使用结果映射（Result Mapping）来解决这一问题，开发人员可以为每个行为结果指定一个别名，再通过别名映射到目标 URL，这一切都配置在外部文件中，很容易修改。将来，Spring.NET 还会实现一个过程管理框架，将结果映射归纳到一个单独的层次，使开发人员可以用非常简单的方法来控制复杂的页面流程。

在 2.0 版之前，ASP.NET 自身的本地化功能相当有限。虽然 VS.NET 2003 可以为每个页面和用户控件都生成一个对应的本地资源文件，但 ASP.NET 基础框架自己却不知道去用它。也就是说，不论何时需要访问本地化资源，开发人员都只能直接使用资源管理器（resource manager）来自己处理，Spring.NET 认为问题不应该这样解决。Spring.NET 的 Web 框架（后文中直接称为 Spring.Web）通过本地资源文件和定义在 IoC 容器内（且供容器使用）的全局资源，为本地化提供了全面的支持。

Spring.Web 能够对 ASP.NET 页面和控件进行依赖注入。也就是说，利用 Spring.NET 的 IoC 容器，开发人员可以很轻松的将服务对象注入到 Web 控件中去。

前面提到的这些功能可以看做是 Spring.Web 的“核心”，除此之外，Spring.Web 也包括一系列可能对大多数开发人员都很有用的“次要功能”，比如在 ASP.NET 1.1 中使用 2.0 的功能（如母版页，即 Master Page），再比如说对向导页面的支持（也就是能处理跨越多次请求和/或表单提交的业务过程的页面）。

为实现上述的功能，Spring.NET 必须扩展（继承）标准 ASP.NET 的 Page 类和 UserControl 类。也就是说，要想充分利用 Spring.Web 的功能（特别是双向数据绑定、本地化和结果映射），Code-behind 中的类型也必须继承 Spring.Web 指定的基类，比如 Spring.Web.UI.Page；不过，对页面进行依赖注入以及其它强大的功能则不需要依赖 Spring.Web 的任何类。之所以强调这一点，是为了让开发人员了解：要想充分利用 Spring.Web 的功能，只能让应用程序的表示层与 Spring.Web 紧耦合。当然，是否要这样做，决定权还是在用户手中。

最后，请留意随 Spring.NET（1.1 版）一同发布的还有几个 Spring.Web 的快速入门程序，以及一个完整的参考程序：SpringAir。简单的快速入门程序是学习 Spring.Web 的最好途径；SpringAir 参考程序则完全使用 Spring.Web 创建前端表示层，其中包含很多 Spring.Web 的最佳开发实践。所以，如果您正在阅读本章内容，别忘了参考该程序的源码（参见第二十九章, SpringAir - 参考程序）。（按：目前第二十九章没有内容，但是 1.1 Preview3 的安装包已经包含了完整的 SpringAir 源码）

19.2. 自动装载应用程序上下文和应用程序上下文嵌套

19.2.1.配置

Spring.Web 自然也是以 Spring.NET IoC 容器为基础的，Spring.Web 在内部大量使用了 IoC 容器的功能。也就是说，如果使用 Spring.Web 建立应用程序，所有控制器（即 ASP.NET 页面）都可以用 Spring.NET 统一的 XML 方式进行配置。Spring.Web 用 PageHandlerFactory 类来（自动）装载和配置 IoC 容器，并从 IoC 容器中获取合适的页面来响应 HTTP 请求。（按：没有在容器中配置的页面则由

ASP.NET 自行管理。如果在 Web.Config 的 httpHandlers 节点中添加了以下 httpHandler：

```
  <add verb="*" path="ContextMonitor.ashx"
type="Spring.Web.Support.ContextMonitor, Spring.Web"/>
```

在调试时，就可以用 http://webdir/ContextMonitor.ashx 查看当前容器内的对象。）

在 Spring.Web 中，IoC 容器的初始化由 PageHandlerFactory 在后台自动进行，这一过程对开发人员来说是完全透明的，不再需要任何形式的显式初始化（比如通过 new 操作符或服务定位器 ContextRegistry）。为使 IoC 容器的自动创建生效，需要将下面的配置加入到 Spring.Web 应用程序根目录的 web.config 文件中（其中 verb 和 path 的属性值当然可以根据实际情况改变）：

```
<system.web>

    <httpHandlers>
        <add verb="*" path="*.aspx"
type="Spring.Web.Support.PageHandlerFactory, Spring.Web"/>
    </httpHandlers>
    ...
</system.web>
```

请注意，这段配置只需要添加到根目录的 web.config 中（也就是 Web 应用程序最顶层虚拟目录中的 web.config 文件）。

这段配置告知 ASP.NET：Spring.Web 的 PageHandlerFactory 类会负责创建.aspx 页面、向页面中注入依赖项（若有需要），然后用此页面来响应 HTTP 请求。

除配置 Spring.Web 的 HttpHandler 之外，还需要在 web.config 文件中定义根应用程序上下文。最终完成的配置文件大概是下面这个样子（当然会随具体情况有所不同）：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

    <configSections>
        <sectionGroup name="spring">
          <section name="context"
type="Spring.Context.Support.WebContextHandler, Spring.Web"/>
        </sectionGroup>
    </configSections>
```

```
    <spring>

        <context>
            <resource uri="~/Config/CommonObjects.xml"/>
            <resource uri="~/Config/CommonPages.xml"/>

            <!-- TEST CONFIGURATION -->
            <!--
            <resource uri="~/Config/Test/Services.xml"/>

            <resource uri="~/Config/Test/Dao.xml"/>
            -->

            <!-- PRODUCTION CONFIGURATION -->

            <resource uri="~/Config/Production/Services.xml"/>
            <resource uri="~/Config/Production/Dao.xml"/>

        </context>

    </spring>

    <system.web>
        <httpHandlers>
            <add verb="*" path="*.aspx"
type="Spring.Web.Support.PageHandlerFactory, Spring.Web"/>
        </httpHandlers>
    </system.web>

</configuration>
```

在这段配置中，有几点非常重要：

1. 必须配置<spring>和<context>节点的处理器（按：请参考第四章的相关内容）。如果需要在同一台服务器上运行多个 Spring.Web 应用程序，也可以将名为 spring 的整个<sectionGroup>节点放在 machine.config 文件中。
2. 必须显式指定<context>节点的 type 属性值为 "Spring.Context.Support.WebApplicationContext, Spring.Web"。这样才能保证 Spring.Web 功能的正常运行（比如处理请求和会话范围内的对象定义）。（按：这点与 Windows 应用的容器配置不同，要特别注意）
3. 必须在<spring>节点中定义一个根上下文，如果对象定义保存在独立的配置文件(比如包含服务和业务层对象定义的 XML 文件)中，需要在<context>节点的<resource>子节点中注册这些文件。文件的路径可以是完整路径或

URL，也可以象上例中一样使用相对路径。容器会将相对路径中的配置文件按照默认的资源类型来处理，对 WebApplicationContext 来说，默认资源类型就是 WebResource 类型。

4. 请注意，同一容器中的对象定义不必保存在同一种资源中（即不需要都是 file://、或都是 http://、或都是 assembly://）。也就是说，容器可以先从一个程序集内嵌资源（assembely://）中载入对象定义，然后再从网络资源中载入其它对象定义。

19.2.2.上下文嵌套

ASP.NET 本身也提供了配置数据的继承机制，开发人员可以用 Web 应用程序子目录中的配置数据覆盖上层目录中的配置数据。

比如说，Web 应用程序根目录下的 web.config 文件可以覆盖 machine.config 文件中的设置，而 Web 应用程序子目录下的 web.config 文件又可以用相同的方式覆盖根目录下的 web.config 文件。子目录的 web.config 也可以添加上层目录未曾出现过的配置内容。

Spring.Web 利用 ASP.NET 的这一机制来进行应用程序上下文的嵌套。可以在子目录的 web.config 中配置新的对象定义，也可以覆盖上层 web.config 中已有的对象定义（覆盖的行为仅在当前子目录中有效）。

对于开发人员来说，可以以虚拟目录为单位将应用程序分为不同的组件，每一个组件都可以在自己虚拟目录中的 web.config 中创建自己的应用程序上下文。位于下层的应用程序上下文一般只包括本组件内部的对象定义，并且可以覆盖父上下文的部分定义（例如菜单）。（按：也就是说，在 Spring.Web 应用中，一个应用程序上下文，或者说一个组件是由一个虚拟目录自然反映的，而应用程序上下文之间的继承关系则是由虚拟目录的层次隐式维系的：在根目录 web.config 中配置的是整个应用的根容器，子目录中配置的就是子容器；在同一虚拟目录中只能配置一个应用程序上下文，而不能象 Windows 应用那样出现嵌套的<context>节点。）

因为子组件的对象定义一般较少，所以建议开发人员将子组件中的对象定义直接写在 web.config 文件中，不必再专门使用外部资源。这样，一个组件的 web.config 文件就和下面的配置差不多：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context"
type="Spring.Context.Support.WebContextHandler, Spring.Web"/>
```

```
      <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <context type="Spring.Context.Support.WebApplicationContext,
Spring.Web">
        <resource uri="config://spring/objects"/>
    </context>

    <objects xmlns="http://www.springframework.net">
        <object type="MyPage.aspx" parent="basePage">
              <property name="MyRootService"
ref="myServiceDefinedInRootContext"/>
              <property name="MyLocalService"
ref="myServiceDefinedLocally"/>
              <property name="Results">
                 <!-- ... -->
              </property>
        </object>
        <object id="myServiceDefinedLocally"
type="MyCompany.MyProject.Services.MyServiceImpl, MyAssembly"/>
    </objects>
  </spring>
</configuration>
```

<context>节点中的<resource>节点告知 IoC 容器从配置文件的 spring/objects 节点中读取对象定义。

当然,我们可以(也应该)将<configSections>节点中和 spring 有关的<section>节点移到上层（或根）web.config 文件中；甚至，如果需要在同一台服务器上运行多个 Spring.Web 应用程序，就应该移到 machine.config 文件中。

有一点非常重要:这些组件级的容器可以引用其父容器中的对象定义。一般来说，如果在当前容器中找不到某个对象定义，Spring.NET 会在所有的父容器中进行查找，直到找到为止（若最终没能找到就抛出异常）。

19.3. 为 ASP.NET 页面进行依赖注入

Spring.Web 以 ASP.NET 为的功能基础：Spring.Web 使用 code-behind 文件中的类型作为 MVC 模式中的控制器，就能说明这一点。在以 MVC 为基础的（Web）应用程序中，控制器一般都是对一或多个服务对象的瘦包装器（thin wrapper）；在开发 Spring.Web 时，Spring.NET 团队就意识到了向页面控制器中注入服务对

象的重要性。因此，Spring.Web 为 ASP.NET 页面的依赖注入提供了一流的支持。开发人员可以用标准的 Spring.NET 配置将任何服务对象（实际上，任何对象）注入到页面中，不需要依赖自定义的服务定位器，也不用自行在容器中手工查找服务对象。

一旦配置好了 Spring.Web 的应用程序上下文，开发人员就能轻松的为页面创建对象定义，并将它们组合成一个完整的 Web 应用程序。

```
<objects xmlns="http://www.springframework.net">

  <object id="basePage" abstract="true">
    <property name="MasterPageFile"
value="~/Web/StandardTemplate.master"/>
  </object>

  <object type="Login.aspx">
      <property name="Authenticator" ref="authenticationService"/>
  </object>

  <object type="Default.aspx" parent="basePage"/>

</objects>
```

在本例中，定义了三个对象：

1. 首先是一个抽象定义，在应用程序中，其它页面可以将它作为基页面，从中继承配置数据。在本例中，这个抽象定义只是简单的规定了要引用哪个页面作为 Master 页，但一般来说，可以在这类对象定义中配置与本地化有关的依赖项，以及图像、脚本和 CSS 样式表的根文件夹。
2. 第二个对象定义则定义了一个登录页面，但没有继承上面抽象定义的信息，也没有使用母版页。从这个对象定义中，我们可以看到如何将服务对象注入到页面对象中（假定 authenticationService 已经在其它地方定义了）。
3. 最后一个对象是应用程序的首页，它继承了抽象的 basePage 定义以便使用母版页，除此之外没有定义任何依赖项。

在配置 ASP.NET 页面的对象定义时，与普通的对象稍有不同。从上面的配置中可以看出，type 属性的值实际上是页面的.aspx 文件名，且文件路径是相对于当前容器所在目录的。因为上面配置的容器是根容器，所以 Login.aspx 和 Default.aspx 必须要在应用程序的虚拟根目录下。母版页则是用绝对路径定义的，因为（位于子目录下的）子容器中的对象也需要引用它。

您可能已经发现，Login 和 Default 页面没有 id 或 name 属性。很显然这不满足 Spring.NET 容器的一般要求，因为普通的对象定义是必须指定 id 或 name 的（内嵌对象定义除外）。实际上，这里是故意省略了 id 或 name 属性。因为在 Spring.Web 应用程序中，页面控制器一般都以.aspx 文件名作为页面对象的标识

符。如果页面对象定义没有指定 id，Spring.Web 就会用.aspx 的文件名作为对象的标识符（会截去路径信息和扩展名，只使用文件名）。

当然，开发人员仍然可以显式指定页面对象的 id 或 name；如果需要多次重用某个页面，并且需要为其注入不同的依赖项时，就需要显式指定 id 了。

（按：在 Spring.Web 的应用程序上下文中，如果为 MasterPage（不管是 System.Web.UI.MasterPage 还是 Spring.Web.UI.MasterPage)配置了对象定义，那么这个对象定义会始终被容器认为是抽象的，abstract 属性不起作用，也无法通过 GetObject 方法从容器中获取 MasterPage 对象。如果使用 ASP.NET 2.0 本身的 MasterPage，容器不会为它进行依赖注入，也无法在 MasterPage 中应用 Spring.Web 的任何高级特性，特别是本地化和语言文化管理。

所以如果使用 Spring.Web 建立 Web 应用，应尽量使模版页继承 Spring.Web.UI.MasterPage。在此请注意，如果模版页继承自 Spring.Web.UI.MasterPage，那么它的内容页就必须继承自 Spring.Web.UI.Page，否则会发生运行时错误。

另外，如果使用 ASP.NET 本身的 MasterPage，通过实现 IApplicationContextAware 接口，模版页也可以在自身代码中获取容器的引用。关于 IApplicationContextAware 接口作用和定义,请参考第四章的相关内容。）

19.3.1.为 Web 控件进行依赖注入

Spring.Web 也能够对页面中的 Web 控件（包括用户控件和标准控件）进行依赖注入。注入的方式有两种：以类型为依据进行全局注入；或者对某个页面中的某个控件进行局部注入。

以类型为依据进行全局注入时,需要以控件的类型全名为标识符创建一个抽象的对象定义；注意此处的类型全名*不能*包含程序集名称。

```
<object id="MyProject.MyControl" abstract="true">
   <!-- inject dependencies here... -->
</object>
```

针对单个控件进行局部注入时，需要创建一个以被注入控件的 UniqueID 为 id 属性值的抽象对象定义，如下：

```
<object id="myContainerControl:myTargetControl" abstract="true">
   <!-- inject dependencies here... -->
</object>
```

要确保这两种对象定义都是抽象的（也就是在<object>节点中添加 abstract="true"属性）。

要获取控件 UniqueID，最简单的方式是将应用程序的 Trace 打开，然后在页面 Trace 信息的 control hierarchy 节点中去找某个控件的 UniqueID。这确实是、、、有点糟糕。（按：打开 trace 的方法是在 web.config 的 system.web 节点中添加一个<trace>节点，比如：

```
<trace enabled="true" requestLimit="20" pageOutput="true"
traceMode="SortByTime" localOnly="true"
writeToDiagnosticsTrace="true"/>
```

）

请注意，可以同时为一个控件的对象定义配置这两种方式的注入，这时，如有必要容器会合并对象定义，用局部注入的值将全局注入的值覆盖掉。

考虑到应用程序上下文的作用域，我们需要注意应该将全局控件的对象定义配置在根容器中，以便能在整个 Web 应用程序执行期间对其进行统一的注入。相对于全局控件，局部控件最好还是定义在各个组件的容器中，以避免不同组件的对象间发生命名冲突。

19.4. 在 ASP.NET 1.1 中使用母版页

Spring.Web 为 ASP.NET 1.1 所扩展的母版页功能与 ASP.NET 2.0 是非常相似的。

为使用母版页，需要在 Web 应用程序中定义一个页面作为布局模板，在其中定义内容占位符以便其它页面可以对其进行引用和填充。下面是一个母版页（Master.aspx）的代码：

```
<%@ Control language="c#" Codebehind="MasterLayout.ascx.cs"
AutoEventWireup="false" Inherits="MyApp.Web.UI.MasterLyout" %>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls"
Assembly="Spring.Web" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >

<html>
    <head>
        <title>Master Page</title>
        <link rel="stylesheet" type="text/css" href="<%=
Context.Request.ApplicationPath %>/css/styles.css">
        <spring:ContentPlaceHolder id="head" runat="server"/>
    </head>
    <body>
        <form runat="server">
            <table cellPadding="3" width="100%" border="1">
                <tr>
```

```
                    <td colspan="2">
                        <spring:ContentPlaceHolder id="title"
runat="server">
                            <!-- default title content -->
                        </spring:ContentPlaceHolder>
                    </td>
                </tr>
                <tr>
                    <td>
                        <spring:ContentPlaceHolder id="leftSidebar"
runat="server">
                            <!-- default left side content -->
                        </spring:ContentPlaceHolder>
                    </td>
                    <td>
                        <spring:ContentPlaceHolder id="main"
runat="server">
                            <!-- default main area content -->
                        </spring:ContentPlaceHolder>
                    </td>
                </tr>
            </table>

        </form>
    </body>
</html>
```

从上面的代码可以看出，母版页定义了页面的整体布局，其它页面可以覆盖其中的四个内容占位符。Mater Page 也可以在占位符中定义默认的内容，如果使用它的页面没有覆盖某个占位符，就会显示默认内容。

下面是一个使用了母版页的子页面 (Child.aspx)：

```
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls"
Assembly="Spring.Web" %>
<%@ Page language="c#" Codebehind="Child.aspx.cs"
AutoEventWireup="false" Inherits="ArtFair.Web.UI.Forms.Child" %>
<html>
    <body>

        <spring:Content id="leftSidebarContent"
contentPlaceholderId="leftSidebar" runat="server">
            <!-- left sidebar content -->
        </spring:Content>
```

```
        <spring:Content id="mainContent" contentPlaceholderId="main"
runat="server">
            <!-- main area content -->
        </spring:Content>

    </body>
</html>
```

在这个页面中,<spring:Content/>控件使用 contentPlaceholderId 属性来指定要填充或覆盖母版页上哪个内容占位符。由于子页面没有定义顶部和标题占位符的内容，所以母版页会显示自身的默认内容。

ContentPlaceHolder 和 Content 控件可以定义在任何 ASP.NET 标记内部,包括：HTML、标准 ASP.NET 控件、用户控件等等。

使用 VS.NET 2003 时的问题

一般来说，子页面中的<html>和<body>标签可以省略，因为母版页中已经定义过了。但是如果这两个标签被省略 VS.NET 2003 的代码提示就会不起作用，所以在编辑时使用这两个标签会比较舒服点。在页面显示时，子页面中的<html>和<body>标签将会被忽略。

19.4.1.将子页面与母版页关联

可以用 Spring.Web.UI.Page 类的 Master 属性为页面指定母版页。另外还有一个 MasterFile 属性，可以用文件名指定母版页。

建议使用 IoC 容器来为页面关联母版页。请看下面的配置：

```
<?xml version="1.0" encoding="utf-8" ?>
<objects xmlns="http://www.springframework.net">

 <object id="masterPage" type="~/Master.aspx" />

 <object id="basePage" abstract="true">
 <property name="Master" ref="masterPage"/>
 </object>

 <object type="Child.aspx" parent="basePage">
 <!-- inject other objects that page needs -->
 </object>

</objects>
```

通过修改其中抽象的对象定义，开发人员可以很容易的为 Web 应用程序中多个页面同时更改母版页。当然，对特定的上下文或页面来说，可以直接用 Master 或 MasterFile 属性覆盖抽象定义中设置的母版页。

19.5. 双向数据绑定

目前，ASP.NET 的数据绑定存在一个问题：它是单向的。数据绑定可以将页面上的控件绑定到数据模型并显示其信息，但是无法在表单提交时将控件的值提取出来。Spring.Web 为 ASP.NET 添加了双向数据绑定的功能，开发人员可以为页面定义数据绑定规则，在页面生存期内的适当时机，这些绑定规则就会自动被应用。

ASP.NET 也不支持回传时的模型管理。的确，它可以用 ViewState 管理视图状态，但是 ViewState 只能处理控件的状态，不能处理与控件绑定的表示层模型对象的状态。开发人员一般只能利用 HTTP 的 Session 对象在回传时存储数据模型。这就又导致了很多本不应该出现的样板式代码，Spring.Web 通过一系列简单模型管理方法，可以消除这些样板代码。

请注意，要想使用双向数据绑定，必须将表示层与 Spring.Web 紧耦合；这是因为双向数据绑定要求页面扩展 Spring.Web 的 Page 类（而非常规的 System.Web.UI.Page 类）。

Spring.Web 数据绑定的用法很简单。开发人员只需要覆盖受保护的 InitializeDataBindings 虚方法，在其中为页面定义一系列数据绑定规则。此外，还需要覆盖三个模型管理方法：InitializeModel，LoadModel，SaveModel。我们还是通过 SpringAir 参考程序的代码来了解这些功能。首先请看页面的定义：

```
<%@ Page Language="c#" Inherits="TripForm"
CodeFile="TripForm.aspx.cs" %>

<asp:Content ID="body" ContentPlaceHolderID="body" runat="server">
    <div style="text-align: center">
        <h4><asp:Label ID="caption" runat="server"></asp:Label></h4>
        <table>
            <tr class="formLabel">
                <td> </td>
                <td colspan="3">
                    <spring:RadioButtonGroup ID="tripMode"
runat="server">
                        <asp:RadioButton ID="OneWay" runat="server" />
                        <asp:RadioButton ID="RoundTrip" runat="server"
/>
                    </spring:RadioButtonGroup>
                </td>
            </tr>
```

```
            <tr>
                <td class="formLabel" align="right">
                    <asp:Label ID="leavingFrom" runat="server" /></td>
                <td nowrap="nowrap">
                    <asp:DropDownList ID="leavingFromAirportCode"
runat="server" />
                </td>
                <td class="formLabel" align="right">
                    <asp:Label ID="goingTo" runat="server" /></td>
                <td nowrap="nowrap">
                    <asp:DropDownList ID="goingToAirportCode"
runat="server" />
                </td>
            </tr>
            <tr>
                <td class="formLabel" align="right">
                    <asp:Label ID="leavingOn" runat="server" /></td>
                <td nowrap="nowrap">
                    <spring:Calendar ID="departureDate" runat="server"
Width="75px" AllowEditing="true" Skin="system" />
                </td>
                <td class="formLabel" align="right">
                    <asp:Label ID="returningOn" runat="server" /></td>
                <td nowrap="nowrap">
                    <div id="returningOnCalendar">
                        <spring:Calendar ID="returnDate"
runat="server" Width="75px" AllowEditing="true" Skin="system" />
                    </div>
                </td>
            </tr>
            <tr>
                <td class="buttonBar" colspan="4">
                    <br/>
                    <asp:Button ID="findFlights" runat="server"/></td>
            </tr>
        </table>
    </div>

 </asp:Content>
```

我们先不要管为什么页面中所有 Label 控件都没有定义文本值，稍后讲到本地化的时候会讨论这个问题。现在要关心的是：页面中定义了一系列输入控件：

tripMode 单选按钮组、leavingFromAirportCode 和 goingToAirportCode 下拉列表，以及两个 Spring.NET 的日历控件：departureDate 和 returnDate。

下面我们来看看要绑定到这个窗体上的模型：

```
namespace SpringAir.Domain
{
    [Serializable]
    public class Trip
    {
        // fields
        private TripMode mode;
        private TripPoint startingFrom;
        private TripPoint returningFrom;

        // constructors
        public Trip()
        {
            this.mode = TripMode.RoundTrip;
            this.startingFrom = new TripPoint();
            this.returningFrom = new TripPoint();
        }

        public Trip(TripMode mode, TripPoint startingFrom, TripPoint
returningFrom)
        {
            this.mode = mode;
            this.startingFrom = startingFrom;
            this.returningFrom = returningFrom;
        }

        // properties
        public TripMode Mode
        {
            get { return this.mode; }
            set { this.mode = value; }
        }

        public TripPoint StartingFrom
        {
            get { return this.startingFrom; }
            set { this.startingFrom = value; }
        }
```

```
    public TripPoint ReturningFrom
    {
        get { return this.returningFrom; }
        set { this.returningFrom = value; }
    }
}

[Serializable]
public class TripPoint
{
    // fields
    private string airportCode;
    private DateTime date;

    // constructors
    public TripPoint()
    {}

    public TripPoint(string airportCode, DateTime date)
    {
        this.airportCode = airportCode;
        this.date = date;
    }

    // properties
    public string AirportCode
    {
        get { return this.airportCode; }
        set { this.airportCode = value; }
    }

    public DateTime Date
    {
        get { return this.date; }
        set { this.date = value; }
    }
}

[Serializable]
public enum TripMode
{
    OneWay,
    RoundTrip
}
```

```
}
```

Trip 类使用两个 TripPoint 类型的属性 StartingFrom 和 ReturningFrom 分别表示出发和返程日期。同时用一个 TripMode 枚举类型的属性来表示旅程是单程还是往返。

最后，Code-behind 文件中的类将所有对象组合在一起：

```
public class TripForm : Spring.Web.UI.Page
{
    // model
    private Trip trip;
    public Trip Trip
    {
        get { return trip; }
        set { trip = value; }
    }

    // service dependency, injected by Spring IoC container
    private IBookingAgent bookingAgent;
    public IBookingAgent BookingAgent
    {
        set { bookingAgent = value; }
    }

    // model management methods
    protected override void InitializeModel()
    {
        trip = new Trip();
        trip.Mode = TripMode.RoundTrip;
        trip.StartingFrom.Date = DateTime.Today;
        trip.ReturningFrom.Date = DateTime.Today.AddDays(1);
    }

    protected override void LoadModel(object savedModel)
    {
        trip = (Trip) savedModel;
    }

    protected override object SaveModel()
    {
        return trip;
    }

    // data binding rules
```

```
    protected override void InitializeDataBindings()
    {
        BindingManager.AddBinding("tripMode.Value", "Trip.Mode");

BindingManager.AddBinding("leavingFromAirportCode.SelectedValue",
"Trip.StartingFrom.AirportCode");
        BindingManager.AddBinding("goingToAirportCode.SelectedValue",
"Trip.ReturningFrom.AirportCode");
        BindingManager.AddBinding("departureDate.SelectedDate",
"Trip.StartingFrom.Date");
        BindingManager.AddBinding("returnDate.SelectedDate",
"Trip.ReturningFrom.Date");
    }

    // event handler for findFlights button, uses injected 'bookingAgent'
    // service and model 'trip' object to find flights
    private void SearchForFlights(object sender, EventArgs e)
    {
        FlightSuggestions suggestions =
bookingAgent.SuggestFlights(trip);
        if (suggestions.HasOutboundFlights)
        {
            // redirect to SuggestedFlights page
        }
    }
}
```

这段看似简单的代码背后其实发生了很多事情，很值得我们逐一的讨论一下：

1. 在第一次请求页面时（IsPostback==false），InitializeModel 方法被调用，在其中创建了一个 Trip 对象并初始化了它的属性。在页面显示之前，SaveModel 方法会被调用，它的返回值会保存到 HTTP Session 中。最后，在每次回传时，LoadModal 方法会被调用，传递给它的参数就是上次 SaveModel 方法保存到 Session 中的值。

   对本例来说，这一过程非常简单：我们的模型只有一个 trip 对象，SaveModel 方法也只返回 trip 对象，LoadModel 只要把 savedModel 参数转型为 Trip，然后赋值给页面的 trip 字段。在稍微复杂的场合中，一般需要用 SaveModel 返回一个包含模型对象的字典，并且在 LoadModel 方法中读取字典的值。

2. 通过调用页面属性 BindingManager 的 AddBinding 方法，InitializeDataBindings 方法为窗体上的五个输入控件分别定义了绑定规则。AddBinding 方法有很多重载，除了上面代码中用到的源和目标的*绑定表达式*，还可以指定*绑定方向*和用于格式化绑定值的 *IFormatter 对*

象。稍后我们会简单的讨论这两个可选参数，现在我们来看一下源和目标的绑定表达式：

数据绑定框架使用 Spring.NET 的表达式语言来定义绑定表达式。在大多数情况下，都会象上面的例子一样将源和目标表达式都解析为某个控件或数据模型的属性或字段名，尤其是在使用双向数据绑定的时候，因为此时这两个绑定表达式都是“可设置的”。关于 InitializeDataBindings 方法，有一点很重要(比应该在此方法中定义数据绑定规则这一点还要重要)，就是该方法只会为*每个页面类型*执行一次。基本上，所有的绑定表达式都在页面第一次被初始化的时候解析并缓存，随后由该页面的所有实例共同使用。这样做主要是出于性能方面的考虑，因为在每次回传时都解析绑定表达式是没有必要的，反会增加页面的总体处理时间。

3. 注意 SearchForFlights 事件处理器并没有依赖界面元素，它只是使用注入进来的 bookingAgent 服务和之前绑定到 UI 控件的 trip 对象来获取一个推荐给客户的航班列表。另外，如果在事件处理器中对 trip 对象进行了修改，那么绑定的对象就会在页面显示前被更新。

   这就实现了我们的首要目标：消除页面事件处理器对界面元素的依赖，将控制器与视图解耦。

4. （按：提示 1、在定义页面的数据模型时，虽然从逻辑上一个私有的字段就足够实现所有功能，但请记住这个模型是要给 Spring.NET 的表达式语言访问的，所以请一定为数据模型定义一个公共属性，可以参考上例中 TripForm 页面的 Trip 属性；提示 2、InitializeModel()方法的调用时机是当 IsPostBack 为 false 时的 Page_Init 事件之前，请注意它和页面事件触发时间的先后。）

我们已经对使用 Spring.NET 在 Web 应用程序中进行数据绑定和模型管理有了一个总体的了解，下面，我们来深入讨论一些细节问题，比如数据绑定在后台究竟是怎样实现的？它都扩展了哪些部分？以及其它一些在实际开发很有用的功能。

19.5.1.数据绑定的后台实现

Spring.NET 的数据绑定框架主要是围绕两个接口来实现的：IBinding 和 IBindingContainer。其中 IBinding 最为重要，所有的绑定类型都要实现它。该接口定义了几个方法，为方便使用，其中一些是有重载的：

```
public interface IBinding
{
    void BindSourceToTarget(object source, object target,
ValidationErrors validationErrors);
    void BindSourceToTarget(object source, object target,
ValidationErrors validationErrors, IDictionary variables);
```

```
    void BindTargetToSource(object source, object target,
ValidationErrors validationErrors);
    void BindTargetToSource(object source, object target,
ValidationErrors validationErrors, IDictionary variables);

    void SetErrorMessage(string messageId, params string[]
errorProviders);
}
```

从这些方法的名称上我们大致可以看出它们的用途。BindSourceToTarget 方法用于从源对象提取绑定的值并复制给目标对象；BindTargetToSource 的功能则与之相反。这两个方法的命名方式很通用，连参数都是常用的类型——这是因为，数据绑定框架实际上可以将任意两个对象绑定起来，虽然最常见的情况是将 Web 窗体绑定到模型对象，而且 Spring.NET 已经将该功能紧密集成在 Web 框架中，但这只是数据绑定框架诸多应用中的一例而已。

我们需要单独说明一下 validationErrors 参数。虽然数据绑定框架没有和验证框架绑定在一起，但它们实际上还是有一定关系的。比如说，数据验证框架最适于根据业务规则验证赋值给模型的数据，但在绑定的过程中，数据绑定框架则更适合对数据的类型进行验证。虽然这两种验证是不同的，但都应该用统一的方式将错误信息显示给用户。为此，Spring.Web 传递给绑定方法的参数和传递给验证对象的参数是同一个 ValidationErrors 实例。这就保证了所有的错误信息都能被统一管理，并通过 Spring.Web 的验证错误显示控件以统一的方式显示给最终用户。

IBinding 接口的最后一个方法是 SetErrorMessage，在绑定时如果发生错误，开发人员可以用这个方法指定一个错误信息的资源 ID，以及用来显示错误信息的 Provider 列表。稍后我们会通过一个小例子来学习 SetErrorMessage 的用法。

IBindingContainer 接口扩展了 IBinding，添加了以下成员：

```
public interface IBindingContainer : IBinding
{
    bool HasBindings { get; }

    IBinding AddBinding(IBinding binding);
    IBinding AddBinding(string sourceExpression, string
targetExpression);
    IBinding AddBinding(string sourceExpression, string
targetExpression, BindingDirection direction);
    IBinding AddBinding(string sourceExpression, string
targetExpression, IFormatter formatter);
    IBinding AddBinding(string sourceExpression, string
targetExpression, BindingDirection direction, IFormatter formatter);
```

```
}
```

可以看到，这个接口定义了一组重载的 AddBinding 方法。其中 AddBinding(IBinding binding)是最通用的一个，可以向 IBindingContainer 中添加任意的绑定类型。其余四个重载是为了使用方便，可以用它们来添加最常用的绑定类型：SimpleExpressionBinding。SimpleExpressionBinding 就是我们在本节一开始的例子中用来将 Trip 对象和 Web 窗体绑定在一起的类型。该类使用 Spring.NET 的表达式语言从源对象提取绑定值，并赋值给目标对象。之前我们已经讨论过 AddBinding 方法的 sourceExpression 和 targetExpression 参数，现在来看另外两个。

19.5.1.1.绑定方向

参数 direction 用来确定数据绑定是单向还是双向。默认情况下，所有绑定都是双向的，除非将 direction 参数设为 BindingDirection.SourceToTarget 或 BindingDirection.TargetToSource，此时，绑定只在某一个方向上的方法被调用时进行，另一个方向上的方法调用会被忽略。在将模型绑定到非输入控件比如 Label 时，单向绑定比较适合。

如果 Web 窗体和表示层模型之间不是简单一对一关系，单向绑定也是很有用的。在前面的例子中，我们刻意将表示层模型（trip 对象）设计为简单的一对一映射。为了方便讨论，现在我们添加一个 Airport 类，并修改原来的 TripPoint 类：

```
namespace SpringAir.Domain
{
    [Serializable]
    public class TripPoint
    {
        // fields
        private Airport airport;
        private DateTime date;

        // constructors
        public TripPoint()
        {}

        public TripPoint(Airport airport, DateTime date)
        {
            this.airport = airport;
            this.date = date;
        }

        // properties
```

```
        public Airport Airport
        {
            get { return this.airport; }
            set { this.airport = value; }
        }

        public DateTime Date
        {
            get { return this.date; }
            set { this.date = value; }
        }
    }

    [Serializable]
    public class Airport
    {
        // fields
        private string code;
        private string name;

       // properties
        public string Code
        {
            get { return this.code; }
            set { this.code = value; }
        }

        public string Name
        {
            get { return this.name; }
            set { this.name = value; }
        }
    }
}
```

除了字符串属性AirportCode,TripPoint类又添加了一个Airport类型的属性：Airport。该类型的定义见上面的代码。现在我们有一个问题：先前AirportCode只是简单的字符串到字符串的绑定，窗体中下拉框的列表项和TripPoint.AirportCode之间可以直接赋值；现在变成了字符串到Airport类之间的绑定，所以我们需要解决这个类型不兼容的问题：

首先，从模型到控件的绑定还是很顺利的。我们只需要将绑定设置为从模型到控件的单向绑定：

```
protected override void InitializeDataBindings()
```

```
{
    BindingManager.AddBinding("leavingFromAirportCode.SelectedValue",
"Trip.StartingFrom.Airport.Code", BindingDirection.TargetToSource);
    BindingManager.AddBinding("goingToAirportCode.SelectedValue",
"Trip.ReturningFrom.Airport.Code", BindingDirection.TargetToSource);
    ...
}
```

我们只需要将机场的代码从 Trip.StartingFrom.Airport.Code 中提取出来赋值给控件，注意这次并不是 Trip.StartingFrom.AirportCode。但是很遗憾，用同样的方式再从控件绑定到模型就不行了：虽然我们也可以把 Airport 的 Code 属性直接（双向）绑定到控件，但是这会使 Airport 的 Name 属性无法被赋值。其实，我们要做的是根据机场的代码查找一个 Airport 对象，并把它赋值给 TripPoint.Airport 属性。好在 Spring.NET 的数据绑定能很容易的实现这一点：在 Spring.Air 中，我们已经在 IoC 容器内定义了一个 airportDao 对象，通过它的 GetAirport(string airportCode)方法就可以查找机场数据。我们只要再定义一个从源到目标（模型，即 Trip.StartingFrom.Airport 属性）的单向绑定规则，并将源定义为表达式——一个调用 airportDao 对象的 GetAirport(string airportCode)方法、从而根据控件值获取 Airport 对象的表达式。下面代码为两个下拉列表定义了完整的绑定规则：

```
protected override void InitializeDataBindings()
{

BindingManager.AddBinding("@(airportDao).GetAirport(#root.leavingFrom
AirportCode.SelectedValue)", "Trip.StartingFrom.Airport",
BindingDirection.SourceToTarget);
    BindingManager.AddBinding("leavingFromAirportCode.SelectedValue",
"Trip.StartingFrom.Airport.Code", BindingDirection.TargetToSource);


BindingManager.AddBinding("@(airportDao).GetAirport(#root.goingToAirp
ortCode.SelectedValue)", "Trip.ReturningFrom.Airport",
BindingDirection.SourceToTarget);
    BindingManager.AddBinding("goingToAirportCode.SelectedValue",
"Trip.ReturningFrom.Airport.Code", BindingDirection.TargetToSource);
    ...
}
```

这就行了——定义两个使用不同表达式的单向绑定规则，并利用表达式从 IoC 容器中获取对象的功能，我们就可以解决这个很不一般的数据绑定问题。

按：注意上面的绑定表达式中的#root，此处必须在控件名称前添加#root 来引用表达式所在的“根环境”，原文例子有错误。请参见第十章相关内容。

19.5.1.2. Formatters

AddBinding 方法最后一个参数是 formatter。我们可以通过这个参数指定一个格式，在将控件值绑定到模型前，用指定的格式来解析（控件的）字符串值；并在将模型值绑定到控件前，用来格式化（模型的）强类型值。

一般情况下，我们可以使用 Spring.Globalization.Formatters 命名空间中的 Formatter 类。如果标准的 Formatter 不能满足要求，也可以创建自己的 Formatter 类——只要实现 IFormatter 接口即可：

```
public interface IFormatter
{
    string Format(object value);
    object Parse(string value);
}
```

Spring.NET 提供的标准 Formatter 包括：CurrencyFormatter、DateTimeFormatter、FloatFormatter、IntegerFormatter、NumberFormatter 和 PercentFormatter，应该可以满足大多数需要了。

19.5.1.3.类型转换

因为数据绑定框架和 IoC 容器都使用相同的表达式求值引擎，所以能够利用所有已注册的类型转换器来执行数据绑定操作。Spring.NET 本身已经自动注册了很多类型转换器（参见 Spring.Objects.TypeConverters 命名空间），不过如果需要，我们还可以实现自定义的类型转换器，并通过标准的 Spring.NET 类型转换器注册机制将它们注册到 Spring.NET 中。

19.5.1.4. 数据绑定事件

Spring.Web 的 Page 类添加了两个与数据绑定相关的页面事件——DataBound 和 DataUnbound。

DataUnbound 事件在数据模型被控件值更新后触发，且仅当 PostBack 时在 Load 事件之后触发，因为使用控件的初始值更新数据模型是没有意义的。

DataBound 事件在控件值更新为数据模型值后触发。触发的时机是在 PreRender 事件之前。

数据模型的更新发生在 Load 事件之后，这样能确保 DataUnbound 事件处理器使用的是更新以后的数据模型；控件值的更新在 PreRender 事件之前，这样能确保数据模型的变化能够立即反映到控件中。

（按：请注意，双向数据绑定的主要目的是将 MVC 的各部分解耦，它管理的是 UI 控件值和模型对象属性之间（一对一）的映射关系，而不是控件状态的初始化。双向绑定无法完成诸如向列表控件绑定 DataSource 这样的功能，在页面类中，这种纯粹的表示层逻辑还是要通过 DataSource 属性和 DataBind()方法完成。读者可以参考 SpringAir 项目的相关代码。）

19.6. 本地化

虽然.NET 框架对本地化的支持相对出色，但 ASP.NET 1.x 在这方面却存在缺憾。

在 ASP.NET 1.1 项目中，每个.aspx 页面都有相关联的资源文件，但 ASP.NET 1.1 却没有去用这些资源文件。ASP.NET 2.0 更正了这一问题，允许开发人员使用页面的本地资源。同时，Spring.Web 对此也提供了支持。（按：原文撰写时 2.0 还是“将来”的事情，译文有些微改动。）

Spring.Web 支持多种本地化方式，必要时可以混合使用。Spring.Web 的本地化支持推模型和拉模型，并且在找不到本地资源时还可以退而查找全局资源。同时，Spring.Web 也支持用户语言文化信息管理和图像本地化，稍后会讨论这些内容。（按：资源的“全局”和“本地”是指其在项目中的适用范围，本地资源仅和某个页面相关联。）

> **提示**
>
> 请参考：Globalization Architecture for ASP.NET 和 Localization Practices for ASP.NET 2.0，这是两篇关于 ASP.NET 国际化和本地化技术的介绍性材料，作者：Michele Leroux Bustamante.

19.6.1.使用 Localizer 进行自动本地化（“推”模型的本地化）

“推”模型本地化的核心思想是：由开发人员在资源文件中为页面指定本地化资源，框架自动将这些资源应用到页面的控件中去。比如，开发人员创建了下面的页面 UserRegistration.aspx...

```
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls"
Assembly="Spring.Web" %>

<%@ Page language="c#" Codebehind="UserRegistration.aspx.cs"
    AutoEventWireup="false"
Inherits="ArtFair.Web.UI.Forms.UserRegistration" %>
<html>
    <body>
        <spring:Content id="mainContent" contentPlaceholderId="main"
runat="server">
            <div align="right">
```

```
                <asp:LinkButton ID="english" Runat="server"
CommandArgument="en-US">English</asp:LinkButton> 
                <asp:LinkButton ID="serbian" Runat="server"
CommandArgument="sr-SP-Latn">Srpski</asp:LinkButton>
            </div>
            <table>
                <tr>
                    <td><asp:Label id="emailLabel"
Runat="server"/></td>
                    <td><asp:TextBox id="email" Runat="server"
Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="passwordLabel"
Runat="server"/></td>
                    <td><asp:TextBox id="password" Runat="server"
Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="passwordConfirmationLabel"
Runat="server"/></td>
                    <td><asp:TextBox id="passwordConfirmation"
Runat="server" Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="nameLabel"
Runat="server"/></td>
                    <td><asp:TextBox id="name" Runat="server"
Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="street1Label"
Runat="server"/></td>
                    <td><asp:TextBox id="street1" Runat="server"
Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="street2Label"
Runat="server"/></td>
                    <td><asp:TextBox id="street2" Runat="server"
Width="150px"/></td>
                </tr>
                <tr>
```

```
                    <td><asp:Label id="cityLabel"
Runat="server"/></td>
                    <td><asp:TextBox id="city" Runat="server"
Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="stateLabel"
Runat="server"/></td>
                    <td><asp:TextBox id="state" Runat="server"
Width="30px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="postalCodeLabel"
Runat="server"/></td>
                    <td><asp:TextBox id="postalCode" Runat="server"
Width="60px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="countryLabel"
Runat="server"/></td>
                    <td><asp:TextBox id="country" Runat="server"
Width="150px"/></td>
                </tr>
                <tr>
                    <td colspan="2">
                        <asp:Button id="saveButton"
Runat="server"/> 
                        <asp:Button id="cancelButton" Runat="server"/>
                    </td>
                </tr>
            </table>
        </spring:Content>
    </body>
</html>
```

仔细看看上面的代码，我们发现其中所有的 Label 或 Button 控件都没有设置 Text 属性。这些控件的 Text 属性值都保存在页面的本地资源文件中，并且用下面这种格式的字符串来标识：

```
$this.controlId.propertyName
```

对应的本地资源文件 UserRegistration.aspx.resx 如下所示：

```
<root>
  <data name="$this.emailLabel.Text">
```

```
    <value>Email:</value>
  </data>
  <data name="$this.passwordLabel.Text">
    <value>Password:</value>
  </data>
  <data name="$this.passwordConfirmationLabel.Text">
    <value>Confirm password:</value>
  </data>
  <data name="$this.nameLabel.Text">
    <value>Full name:</value>
  </data>
  <data name="$this.street1Label.Text">
    <value>Street 1:</value>
  </data>
  <data name="$this.street2Label.Text">
    <value>Street 2:</value>
  </data>
  <data name="$this.cityLabel.Text">
    <value>City:</value>
  </data>
  <data name="$this.stateLabel.Text">
    <value>State:</value>
  </data>
  <data name="$this.postalCodeLabel.Text">
    <value>ZIP:</value>
  </data>
  <data name="$this.countryLabel.Text">
    <value>Country:</value>
  </data>
  <data name="$this.saveButton.Text">
    <value>$messageSource.save</value>
  </data>
  <data name="$this.cancelButton.Text">
    <value>$messageSource.cancel</value>
  </data>
</root>
```

其中最后两项需要特别解释一下。有时候需要将某些资源定义为全局资源。在本例中，我们会在整个应用程序中使用 Save 和 Cancel 按钮，所以将它们的资源定义为全局资源比较合适。

在本地资源文件中定义全局资源的方法，是将它们的<value>属性设置为包含以下前缀的资源重定位表达式：

```
$messageSource.
```

Loclizer 会使用 value 值中的“save”和“cancel”作为查询键值，从全局的消息源中获取最终的文本。有一点要记住的是，资源重定位的目标值只需要定义一次，一般来说要定义在固定语言文化（Invariant culture）资源文件中——所有重定位资源的查询操作都会以固定语言文化为依据，所以能够确保使用正确的语言文化进行全局消息源查询。

提示

要在 VS.NET 中查看页面的.resx 文件，需要点击“Project/Show All Files”。该菜单项启用之后，就可以看到.resx 文件象页面文件的一个“子节点”一样显示在解决方案浏览器中。

VS.NET 创建的.resx 文件会包含一个 xds:schemaElement 和数个 reshead 节点。data 节点出现在 reshead 节点之后。如果要修改.resx 文件，可以在 Windows 资源管理器的右键菜单中选择"Open With"，然后选择适当的源码编辑器。

19.6.2.使用 Localizer

要想自动应用资源，需要向每个页面中注入一个 Localizer 对象（可以配置在一个抽象的页面对象定义中，然后让其它页面对象定义继承它）。注入给页面的 Localizer 会在页面第一次被请求时检查资源文件、将其中所有以“$this”开头的资源项的值加入缓存，并在页面显示前将这些值应用到页面的控件上。

Localizer 是一个实现了 Spring.Globalization.ILocalizer 接口的对象。
Spring.Net 中有一个抽象基类以供继承，
Spring.Globalization.AbstractLocalizer：该类包含一个抽象方法 LoadResources。在实现类中，这个方法必须读取并返回所有需要自动应用的资源的列表。

Spring.Net 还提供了一个具体的实现类，
Spring.Globalization.Localizers.ResourceSetLocalizer，该类可以从本地资源文件中返回资源列表。将来 Spring.NET 还会提供其它实现类，以便从 XML 文件甚至是包含资源 name-value 对的文本文件中读取资源，这样开发人员就可以将资源保存到 Web 应用程序的文件中，而不需要嵌入到程序集内。当然，如果需要将资源保存在数据库中，也可以自己实现 ILocalizer。

前面提到过，一般我们会在一个抽象页面对象定义中配置 Localizer，再由需要它的页面继承，如下：

```
<object id="localizer"
type="Spring.Globalization.Localizers.ResourceSetLocalizer,
Spring.Core"/>
<object id="basePage" abstract="true">
    <description>
```

```
        Pages that reference this definition as their parent
        (see examples below) will automatically inherit following
properties.
    </description>
    <property name="Localizer" ref="localizer"/>
</object>
```

当然，开发人员完全可以为每个页面单独配置 Localizer；在任何情况下都可以用单独配置的 Localizer 覆盖抽象定义中的 Localizer。另外，如果不需要自动应用资源，就可以不配置 Localizer。

最后要注意，Spring.Web 的 UserControl 会从它们所在的页面上继承 Localizer 和其它本地化的设置，但同样可以通过显式的依赖注入将其覆盖。

19.6.3.手动应用资源（“拉”模型的本地化）

虽然自动资源本地化在多数情况下都能满足要求，但对迭代控件中的内嵌控件却是无能为力，因为这些控件的 ID 是不确定的。同样，如果需要在同一页面中多次使用同一资源，自动方式也无法解决问题。例如，SpringAir 示例程序中航班列表的标题列。（请参见：第二十九章,SpringAir - 参考程序）

在这些情况下，需要借助拉模型的本地化功能，方式很简单：调用 GetMessage 方法，如下：

```
<asp:Repeater id="outboundFlightList" Runat="server">
  <HeaderTemplate>
    <table border="0" width="90%" cellpadding="0" cellspacing="0"
align="center" class="suggestedTable">
      <thead>
        <tr class="suggestedTableCaption">
          <th colspan="6">
            <%= GetMessage("outboundFlights") %>
          </th>
        </tr>
        <tr class="suggestedTableColnames">
          <th><%= GetMessage("flightNumber") %></th>
          <th><%= GetMessage("departureDate") %></th>
          <th><%= GetMessage("departureAirport") %></th>
          <th><%= GetMessage("destinationAirport") %></th>
          <th><%= GetMessage("aircraft") %></th>
          <th><%= GetMessage("seatPlan") %></th>
        </tr>
      </thead>
      <tbody>
```

```
  </HeaderTemplate>
```

Spring.Web.UI.Page 和 Spring.Web.UI.UserControls 类中都实现了 GetMessage 方法，如果在本地资源中找不到所请求的项，该方法会自动去全局消息源中查找。

19.6.4.在 Web 应用程序中进行图像本地化

Spring.Web 支持在 Web 应用程序中用简单（且统一）的方式进行图像本地化。在一般的 Web 应用程序中，图像资源与文本资源不同，文本资源可以保存在内嵌的资源文件、XML 文件甚至数据库中，而图像资源一般保存在文件系统的文件中。在 Spring.Web 的支持下，通过一个自定义控件，并结合目录命名规范，本地化图像资源就象本地化文本资源一样简单。

Spring.Web 的 Page 类有一个 ImageRoot 属性，用于定义存放图像文件的根目录。该属性的默认值是“Images”，也就是说 Localizer 会在应用程序根目录的 Images 文件夹中查找图像资源。不过在页面定义中可以将该属性改为任意路径名。

为进行图像本地化，需要在 ImageRoot 属性指向的目录下为每种要支持的语言文化定义一个子目录，比如：

```
/MyApp
   /Images
      /en
      /en-US
      /fr
      /fr-CA
      /sr-SP-Cyrl
      /sr-SP-Latn
      ...
```

目录层次建立之后，就只需将图像文件放置在各个子目录中了，同一图像在不同子目录中的文件名要相同。要在页面中进行图像本地化，需要定义一个 <spring.LocalizedImage>控件：

```
<%@ Page language="c#" Codebehind="StandardTemplate.aspx.cs"
                AutoEventWireup="false"
Inherits="SpringAir.Web.StandardTemplate" %>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls"
Assembly="Spring.Web" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <body>
```

```
    <spring:LocalizedImage id="logoImage"
imageName="spring-air-logo.jpg" borderWidth="0" runat="server" />
  </body>
</html>
```

该控件会使用标准的本地化规则和用户的语言文化信息在相应目录下查找指定的图片文件。例如，如果用户的语言文化是“en-US”，Localizer 就会在 Images/en-US 目录中查找 spring-air-logo.jpg 文件，如果找不到，转而搜索 Images/en 目录，如果仍找不到，就会在 Images 目录下查找（该文件夹一般用于存放通用的文件）。

19.6.5.全局资源

全局资源是（以应用程序上下文为单位的、）定义在容器中的、以保留字“messageSource”命名的普通对象定义。开发人员可以在容器的配置文件中添加下面的对象定义。

```
<object id="messageSource"
type="Spring.Context.Support.ResourceSetMessageSource, Spring.Core">
    <property name="ResourceManagers">
        <list>
            <value>MyApp.Web.Resources.Strings, MyApp.Web</value>
        </list>
    </property>
</object>
```

全局资源会被 IApplicationContext 缓存，并可以通过 IMessageSource 接口访问。

Spring.Web 的 Page 和 UserControl 类都定义了相应的属性来引用它们所在的应用程序上下文，以及与上下文关联的 IMessageSource。这样以来，如果 Page 或 UserControl 在本地资源中找不到某个资源，就会自动去全局资源中查找。

目前，ResourceSetMessageSource 是惟一随 Spring.NET 一同发布的 IMessageSource 实现类。

19.6.6.用户语言文化管理

除了全局和本地资源管理，Spring.Web 也支持用户语言文化管理，在 Spring.Web 的 Page 和 UserControl 类中，都可用 UserCulture 属性访问的当前的 CultureInfo 值。

UserCulture 属性将区域信息的解析工作代理给 Spring.Globalization.ICultureResolver 接口的实现类。开发人员可以在页面的对象定义中配置要使用的语言文化解析器类，如下：

```
<object id="BasePage" abstract="true">
    <property name="CultureResolver">
        <object
type="Spring.Globalization.Resolvers.CookieCultureResolver,
Spring.Web"/>
    </property>
</object>
```

Spring.Web 提供了几个很有用的 ICultureResolver 实现类，开发人员不需要自行实现该接口。不过如果的确需要创建自己的实现类，过程也是很简单的，因为只有两个方法需要实现。下面几节讨论 Spring.Web 提供的几个 ICultureResolver 实现类。

19.6.6.1. DefaultWebCultureResolver

该类在 Spring.Web 中是默认的 ICultureResolver 实现类。如果没有为页面对象显式配置语言文化解析器，就会自动使用该类。有时候显式配置该类也是十分有用的，比如说，有时需要设置它的 DefaultCulture 属性来指定一个默认的语言文化。

DefaultWebCultureResolver 会首先检查自己的 DefaultCulture 属性，如果不为空，就直接返回它的值。若 DefaultCulture 属性值为空就会去检查 HTTP 请求的头信息，最后，如果头信息中没有包括“Accept-Lang”，就会返回当前线程的 UI 语言文化。

19.6.6.2. RequestCultureResolver

该类的工作方法与 DefaultWebCultureResolver 类似，只是它会先检查 HTTP 请求的头信息，再检查 DefaultCulture 属性的值，最后使用与当前线程相关的语言文化。

19.6.6.3. SessionCultureResolver

该类会在用户会话中查找语言文化信息，如果找到就返回，如果找不到，其行为就和 DefaultWebCultureResolver 一样。

19.6.6.4. CookieCultureResolver

该类会在 cookie 中查找语言文化信息，如果找到就返回，如果找不到，其行为就和 DefaultWebCultureResolver 一样。

> **警告**
>
> 如果使用 localhost 作为 URL 来请求页面，CookieCultureResolver 就不起作用，因为一般情况下会认为这是在开发环境中发起的请求。
>
> 为克服这一局限，应该在开发时使用 SessionCultureResolver，而在布署前改为 CookieCultureResolver。在 Spring.Web 中这是很容易做到的（通过修改容器的配置），但还是请注意到这一点。

19.6.7.更改语言文化

要想更改语言文化，必须选用支持该功能的语言文化解析器（按：即选择的解析器必须能够保存新的 CultureInfo 值），例如 SessionCultureResolver 或 CookieCultureResolver。如果需要将语言文化信息保存到数据库中作为用户配置的一部分，则可以创建自己的 ICultureResolver 实现类。

只要选定了合适的语言文化解析器，剩下的就是在页面显示之前将 UserCulture 属性值改为新的 CultureInfo 对象了。在下面的例子中，页面上有两个 LinkButton 用于改变用户的语言，在事件处理器中，只需用下面的代码就可以完成语言文化的更改。这段代码摘自 UserRegistration.aspx.cs：

```
protected override void OnInit(EventArgs e)
{
    InitializeComponent();

    this.english.Command += new CommandEventHandler(this.SetLanguage);
    this.serbian.Command += new CommandEventHandler(this.SetLanguage);

    base.OnInit(e);
}

private void SetLanguage(object sender, CommandEventArgs e)
{
    this.UserCulture = new CultureInfo((string) e.CommandArgument);
}
```

（按：原文的这段描述不甚详细，在此请注意两点：1、只有当向页面注入了 CultureResolver 对象之后，才能对 UserCulture 属性赋值，另外，别忘了注入

Localizer；2、UserCulture 被改变之后，CultureResolver 对象会将新的 UserCulture 值保存到相应位置。若要对其它页面自动产生影响，需要向这些页面注入同一种类的 CultureResovler，这样，当这些页面被请求时，CultureResolver 会在页面显示前先读出已存的 CultureInfo 对象，并用它改变当前页面的 UserCulture 属性，随后，注入给页面的 Localizer 对象就可以利用该属性到正确的资源文件中读取资源。读者可以参考 ICultureResolver 接口实现类（如 SessionCultureResolver）的源码。

还有一个问题需要特别注意：Spring.Web.UI.MasterPage 是没有 CultureResolver 属性的，如果不做其它处理，不可能在 MasterPage 中直接修改 UserCulture 属性！但是，除非有特殊要求，语言文化的修改选项必定设计在 MasterPage 上，这个问题该怎么解决？先提示一下读者——打开 SpringAir 参考项目，看一下 Config/Web.xml 文件中名为 standardPage 的对象定义。然后我们再来说明一下，若要在 MasterPage 中更改 UserCulture，必须要向其内容页中：1、注入合适的 ICultureResolver 对象。2、注入 Localizer。3、也是关键所在，必须在内容页的对象定义中显式设置 MasterPageFile 属性值为 MasterPage 的页面名称。这样，在内容页被创建之后，MasterPage 才能对语言文化进行设置。注意，如果有某个内容页没有显式设置 MasterPageFile 属性，那么再请求这个页面时，MasterPage 中修改 UserCulture 的代码同样会出错，所以我们需要为每个内容页的对象定义都设置 MasterPageFile。最好的方式是将 CultureResolver、Localizer 和 MasterPageFile 集中定义在同一个父对象定义中，让其它页面对象定义去继承它，如同 SpringAir 中的 standardPage 对象定义一样。）

19.7. 结果映射

ASP.NET 一个很显著的问题是没有任何手段将应用程序的流程控制移出代码之外。最常用的方式是在页面的事件处理器中硬编码，调用 Response.Redirect 和 Server.Transer 方法将页面重新定向。

这种方法是有问题的，程序流程上的任何变化都会导致代码的变化（以及随之而来的重新编译、测试、重新布署等等）。比较好的方法是将操作结果和目标页面的映射关系移到程序外部，这种方式已被很多基于 MVC（Model-View-Controller）的 Web 框架证明是成功的。

Spring.Web 支持这种功能，开发人员可以在页面的对象定义中配置结果映射集，然后在事件处理器中使用结果映射的逻辑名称来控制应用程序的流程。

Spring.Web 使用 Result 类来封装并定义逻辑结果，通过 Result 类，可以象配置普通对象一样配置结果映射集：

```
<objects xmlns="http://www.springframework.net">
```

```
    <object id="homePageResult" type="Spring.Web.Support.Result,
Spring.Web">
        <property name="TargetPage" value="~/Default.aspx"/>
        <property name="Mode" value="Transfer"/>
        <property name="Parameters">
            <dictionary>
                <entry key="literal" value="My Text"/>
                <entry key="name" value="${UserInfo.FullName}"/>
                <entry key="host" value="${Request.UserHostName}"/>
            </dictionary>
        </property>
    </object>

    <object id="loginPageResult" type="Spring.Web.Support.Result,
Spring.Web">
        <property name="TargetPage" value="Login.aspx"/>
        <property name="Mode" value="Redirect"/>
    </object>

    <object type="UserRegistration.aspx" parent="basePage">
        <property name="UserManager" ref="userManager"/>
        <property name="Results">
            <dictionary>
                <entry key="userSaved" value-ref="homePageResult"/>
                <entry key="cancel" value-ref="loginPageResult"/>
            </dictionary>
        </property>
    </object>

</objects>
```

在 Result 的对象定义中，必须配置 TargetPage 属性的值。Mode 属性值可以是 Transfer 或 Redirect，默认为 Transfer。

如果目标页面需要参数，可以通过 Result 的 Parameters 属性定义。参数的值既可以是文本值，也可以是对象表达式；页面所在的容器会对表达式进行求值；在本例中，任何使用 homePageResult 的页面都必须定义一个 UserInfo 属性。

根据 Result 对象 Mode 属性值的不同，参数的处理方式也不同。对于 Redirect 模式的 Result 对象来说，所有参数都会先转换为字符串，然后将 URL 编码，最后附加在重定向的请求字符串上。而对于 Transfer 模式的 Result 对象，参数会在请求被转发给目标页面前被添加到 HttpContext.Items 集合中。也就是说 Transfer 模式可以在页面间传递任何对象，所以更为灵活。同时 Transfer 模式也更为高效，因为此时不需要在服务器和客户端之间产生额外的往返过程，所以建议优先 Transfer 模式（这也是它是默认模式的原因）。

上例中，Result 的对象定义是独立的，可以被所有页面引用，比如主页和登录页面。如果 Result 对象只被一个页面使用，应该将 Result 内嵌在页面对象定义的内部，此时可以用内嵌对象定义，也可以用专门的 Result 快捷语法（见下面代码中名为 userSaved 的 Results 值）。

```
<object type="~/UI/Forms/UserRegistration.aspx" parent="basePage">
        <property name="UserManager">
            <ref object="userManager"/>
        </property>
        <property name="Results">
            <dictionary>
                <entry key="userSaved"
value="redirect:UserRegistered.aspx?status=Registration
Successful,user=${UserInfo}"/>
                <entry key="cancel" value-ref="homePageResult"/>
            </dictionary>
        </property>
</object>
```

如果使用快捷方式定义内嵌的 Result，为页面 Results 属性添加的字典项值（即 value 属性的值）必须满足下面的格式...

```
[<mode>:]<targetPage>[?param1,param2,...,paramN]
```

其中<mode>的值前面讲过，就是...

```
redirect
transfer
```

要注意此处使用逗号来分隔参数，而不像 URL 中那样使用&符号，以免需要在 XML 中进行转义。如果一定要用&符号，应该使用&的实体引用（按：即“&”）。

定义了 Result 对象之后，就很容易在页面的事件处理器中使用了（下面代码取自 UserRegistration.aspx.cs）...

```
private void SaveUser(object sender, EventArgs e)
{
    UserManager.SaveUser(UserInfo);
    SetResult("userSaved");
}

public void Cancel(object sender, EventArgs e)
{
    SetResult("cancel");
}
```

```
protected override void OnInit(EventArgs e)
{
    InitializeComponent();

    this.saveButton.Click += new EventHandler(this.SaveUser);
    this.cancelButton.Click += new EventHandler(this.Cancel);

    base.OnInit(e);
}
```

当然我们可以在上面的代码中使用常量来代替字符串。如果一个逻辑结果（比如“home”）会被多个页面引用，使用常量就比较合适。

19.8. 客户端脚本

通过 Page 类的 RegisterClientScriptBlock 和 RegisterStartupScript 方法，ASP.NET 对客户端脚本的支持还算马马虎虎。

但是，这两个方法都不能向页面的<head>节点中添加脚本，而很多情况下我们的确需要这么做。

19.8.1.在 HTML 的 head 节点中注册客户端脚本

Spring.Web 为 Spring.Web.UI.Page 类增加了个方法，以增强对客户端脚本的支持，其中包括 RegisterHeadScriptBlock 和 RegisterHeadScriptFile，这两个方法都有自己的重载。在 Page 或用户控件中，可以用这两个方法将客户端脚本代码或脚本文件添加到 HTML 页面的<head>节点中。

为此，惟一要做的特殊工作就是需要用服务端控件<spring:Head>来定义页面的<head>节点，而不能用标准 HTML 的<head>节点。请看下面的例子：

```
<%@ Page language="c#" Codebehind="StandardTemplate.aspx.cs"
                AutoEventWireup="false"
Inherits="SpringAir.Web.StandardTemplate" %>

<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls"
Assembly="Spring.Web" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <spring:Head runat="server" id="Head1">
    <title>
      <spring:ContentPlaceHolder id="title" runat="server">
```

```
        <%= GetMessage("default.title") %>

      </spring:ContentPlaceHolder>
    </title>
    <LINK href="<%= CssRoot %>/default.css" type="text/css"
rel="stylesheet">
    <spring:ContentPlaceHolder id="head"
runat="server"></spring:ContentPlaceHolder>
  </spring:Head>

  <body>
  ...
  </body>
</html>
```

本例可以看到如何向 Master 页面添加<head>节点，以便子页面能用<spring:ContentPlaceholder>控件改变页面的标题，并向<head>节点中添加其它元素。其中，惟一与前文提到的 Register*方法有关的是<srping:Head>控件。

TODO : insert example

19.8.2.向<head>节点中添加 CSS 定义

使用前面提到的两个方法，我们也可以向页面中添加 CSS 文件。或者使用更为直接的方式，调用 Spring.Web.UI.Page 类的 RegisterStyle 和 RegisterStyleFile 方法将 CSS 定义直接添加到<head>节点中。RegisterStyleFile 方法可以引用外部的 CSS 文件，RegisterStyle 方法则可以向页面添加内嵌的样式表，在最终的 HTML 文档中，用 RegisterStyle 方法注册的样式表表现为一个内嵌的<style>节点。

TODO : insert example

19.8.3.全局目录（Well-Known Directories）

为简化客户端脚本文件、CSS 文件和图像文件的引用过程，Spring.Web.UI.Page 类提供了几个属性，使得开发人员能够通过绝对路径引用这些文件。如果开发人员喜欢常规的（目录式的）Web 应用程序结构，这些属性会为他们提供很大的便利。

这几个属性是 ScriptsRoot、CssRoot 和 ImagesRoot。它们的默认值分别为“Scripts”、“CSS”和“Images”，如果在 Web 应用程序的根目录下创建了这几个目录，就可以直接使用默认值。但是，如果喜欢将这些文件放在其它位置，

也可以随时在页面对象定义中修改这些默认值（同样，一般将这些值设置在抽象的页面定义中，再由其它页面定义继承）。下面是一个例子：

```
<object id="basePage" abstract="true">
    <description>
        Convenience base page definition for all the pages.
        Pages that reference this definition as their parent (see the
examples below)
        will automatically inherit following properties....
    </description>
    <property name="CssRoot" value="Web/CSS"/>
    <property name="ImagesRoot" value="Web/Images"/>
</object>
```

第二十章. .NET Remoting

20.1. 简介

Spring.Services 命名空间的目的是为业务服务提供位置的透明性。我们相信使用普通的接口和.NET 类，用户应该可以用最简单的方式实现服务。我们也认为在将某个服务发布给客户端的时，应该只关心服务的配置，而无需关心服务的实现。

在 Spring.Services 命名空间的支持下，可以用 IoC 容器中的服务导出对象将任一个普通对象发布为 web 服务、企业服务组件或远程对象。这里说的“普通对象”是指不继承或应用基础框架中任何特殊的基类（如 MarshalByRefObject）或特性（如 WebMethod）的对象。目前 Spring.NET 要求要发布的对象必须实现一个业务接口（按：即要发布的类必须实现某个接口，但这并非是限制，毕竟面向接口无论在什么时候都是好的编程实践）。Spring.NET 的 Remoting 导出类会自动为其创建一个继承自 MarshalByRefObject 代理类。在服务端，可以将 SAO 类型注册为 SingleCall 或 Singleton 模式，也可以为每个对象配置生存期和租赁时间。另外，还可以将应用了 AOP 通知的对象发布为 SAO。在客户端，可以用面向接口的最佳编程方式获取 CAO 的代理对象。Spring.NET 还为 Remoting 创建了专门的 schema 来简化 xml 配置，不过我们仍然可以使用标准的 schema 来创建对象定义。

本章多次引用的例子，存放在"examples\Spring\Spring.Examples.Calculator"目录下。

20.2. 在服务端发布 SAO

在.NET 中，有两种方式可以发布 Singleton 模式的 SAO 服务。一是调用 RemotingConfiguration.RegisterWellKnownServiceType 方法，通过编程方法完成，这种方法的局限是必须使用远程对象的默认构造器，且在运行期不容易配置远程对象的 singleton 状态，因为它是按需创建的。第二种方法是调用 RemotingServices.Marshal 方法。这个方法没有前一种的局限。下面的代码是在服务端将一个有初始状态的对象发布为 Singleton SAO。

```
AdvancedMBRCalculator calc = new AdvancedMBRCalculator(217);
RemotingServices.Marshal(calc, "MyRemotedCalculator");
```

其中的 AdvancedMBRCalculator 继承了 MarshalByRefObject。

如果需要配置 singleton SAO 的远程属性，或者需要使用含参的构造器，可以用 Spring.NET 的 IoC 容器来创建、配置 SAO 实例并将其发布为远程对象。SaoExporter 类可以完成这项工作，而且最重要的是，如果要发布的业务对象没有继承自 MarshalbyRefObject，该类会为其自动创建一个继承自 MarshalbyRefObject 的代理类。下面这段 XML 代码取自示例程序 Remoting 快速入门。

20.2.1. SAO Singleton

```
<object id="singletonCalculator"
type="Spring.Calculator.Services.AdvancedCalculator,
Spring.Calculator.Services">

  <constructor-arg type="int" value="217"/>
</object>

<!-- Registers the calculator service as a SAO in 'Singleton' mode. -->
<object name="saoSingletonCalculator"
type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="singletonCalculator" />
  <property name="ServiceName" value="RemotedSaoSingletonCalculator"
/>
</object>
```

这段代码将对象 singletonCalculator 发布为 URI 路径为 "RemotedSaoSingletonCalculator"的 SOA（完整的 URI 为 tcp://localhost:8005/RemotedSaoSingleCallCalculator，使用的是标准.NET 通道配置，稍后讨论）。AdvancedCalculator 类实现了一个业务接口 IAdvancedCalculator。目前 Spring.NET 创建代理的方式要求业务对象必须实现

一个接口。Spring.NET 生成的远程代理类只会包含由接口定义的方法。在上面的对象定义中，通过构造器将远程计算器对象的初始值设为 217。注意 Services.AdvancedCalcualtor 类并没有继承 MarshalByRefObject。属性 Infinite 将设置远程对象的租赁期设为无限，这样远程对象就不会在 5 分钟（默认的租赁时间）之后被垃圾回收。其它与生命周期有关的属性可以通过 InitialLeaseTime，RenewOnCallTime 和 SponsorshipTimeout 属性来设置。

我们可以用自定义的 Remoting schema 来简化对象定义，并且可以在 XML 编辑器中用来进行代码提示，这样对象定义就如下所示：

```
<objects xmlns="http://www.springframework.net"
         xmlns:r="http://www.springframework.net/remoting">

  <r:saoExporter targetName="singletonCalculator"
                 serviceName="RemotedSaoSingletonCalculator" />

  ... other object definitions

</objects>
```

请参考本章最后的内容以了解 Remoting schema 的详细信息。

20.2.2. SAO SingleCall

下面代码发布一个 SingleCall 模式的 SAO：

```
<object id="prototypeCalculator"
type="Spring.Calculator.Services.AdvancedCalculator,
Spring.Calculator.Services"
        singleton="false">
  <constructor-arg type="int" value="217"/>
</object>

<object name="saoSingleCallCalculator"
type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="prototypeCalculator" />
  <property name="ServiceName" value="RemotedSaoSingleCallCalculator"
/>
</object>
```

注意我们修改的是目标对象定义的 singleton 属性，SaoExporter 对象没有做任何改变。SaoExporter 要发布的对象（即 TargetName 属性所引用的对象）也可以是一个 AOP 代理。例如，如果要为计算器对象应用一个简单的 logging 通知，可以用下面的代码：

```
<object id="singletonCalculatorWeaved"
type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
  <property name="target" ref="singletonCalculator"/>
  <property name="interceptorNames">
    <list>
      <value>Log4NetLoggingAroundAdvice</value>
    </list>

  </property>
</object>

<object name="saoSingletonCalculatorWeaved"
type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="singletonCalculatorWeaved" />
  <property name="ServiceName"
value="RemotedSaoSingletonCalculatorWeaved" />
</object>
```

注意

按照.NET Remoting 的要求，远程对象的方法参数必须是可序列化的。

20.2.1. 控制台应用程序配置

在使用 SaoExporter 类时，仍可用.NET 标准的配置节点来配置通道，如下代码所示：

```
<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="tcp" port="8005" />
    </channels>
  </application>
</system.runtime.remoting>
```

在代码中，需要先调用 RemotingConfiguration.Configure("RemoteServer.exe.config")方法来启动.NET Remotion 基础框架（因为本例是在 config 文件中注册的通道），然后再启动 Spring.NET 的应用程序上下文。如下所示：

```
RemotingConfiguration.Configure("RemoteApp.exe.config");

IApplicationContext ctx = ContextRegistry.GetContext();

Console.Out.WriteLine("Server listening...");
```

```
Console.ReadLine();
```

我们也可以在容器中配置一个 Spring.Remoting.RemotingConfigurer 对象，这样 IoC 容器就会在初始化的时候去调用 RemotingConfiguration。RemotingConfigurer 实现了 IObjectFactoryPostProcessor 接口，在所有对象定义被装载之后且初始化方法被调用之前，容器会调用此接口的方法（参见：4.9 节,使用 ObjectFactoryPostProcessor 自定义对象工厂）。RemotingConfigurer 有两个属性可以配置。一个是 Filename，用以指定.NET Remoting 配置文件的位置（如果为 null 则使用默认文件名）；另一个是 EnsureSecurity，用于确保通道是加密的（只适用.NET 2.0）。为方便起见，可以用 Spring.NET 的 Remoting Schema 来配置对象，请看下面的例子（节选自 Remoting 快速入门）：

```
<objects xmlns="http://www.springframework.net"
         xmlns:r="http://www.springframework.net/remoting">

  <r:configurer filename="Spring.Calculator.RemoteApp.exe.config" />

</objects>
```

ReadLine()方法可以防止控制台程序的退出。读者可以参考 Remoting 快速入门中 RemoteServer 的代码。发布 Windows 服务的过程与此很相似，只是在最后不用调用 ReadLine，因为服务进程不会退出。

20.2.3. IIS 应用程序配置

如果要在 IIS 中布署 Remoting 应用程序，可以参考示例程序。基本思想是在 Global.asax 的 Application_Start 方法中初始化 Spring.NET 的 IoC 容器，如下所示：

```
 void Application_Start(object sender, EventArgs e)
 {
     // Code that runs on application startup

     // Ensure Spring has loaded configuration registering context
     Spring.Context.IApplicationContext ctx = new
Spring.Context.Support.XmlApplicationContext(

HttpContext.Current.Server.MapPath("Spring.Config"));
     Spring.Context.Support.ContextRegistry.RegisterContext(ctx);
 }
```

在这个例子中，Spring.NET 的配置文件名为“Spring.Config”。在 Web.config 文件中添加一个标准的<system.runtime.remoting>节点。注意在此不需要指定通道的端口号，因为使用的是 HTTP 端口。如果显式指定端口反而会导致很多莫名其妙的后果。另外，为使 IIS 能识别 Remoting 请求，在导出远程对象时，应该以".rem"或".soap"作为后缀名，这样 IIS 才能做出正确的响应。

20.3. 在客户端访问 SAO

在.NET 中，客户端通过 Administrative 类型注册可以很容易的获得 SAO 的客户端引用。当某个远程类型在客户端注册之后，使用 new 操作符或反射 API 会获得一个远程对象的代理，而非本地对象的引用。在客户端的配置文件中，可以通过 wellknown 节点来注册远程 SAO 对象。但这种方法需要客户端了解 SAO 类型的实现，实际上就是要在客户端引用服务端的程序集，这是一种公认的糟糕的编程方式。如果基于业务接口开发远程服务，就可以消除这种依赖性。即使不从 Remoting 的角度考虑，将接口和实现分离也是设计 OO 系统时一个好习惯。对于 Remoting 来说，这样的设计能使客户可以只通过接口程序集引用远程对象的代理。所以，为将客户端和服务端解耦，我们需要创建一个单独的、只包含接口定义的程序集，并在客户端和服务端共享它。

当远程对象为 SAO 时，这种设计很容易实现。调用 Activator.GetObject 会在客户端初始化一个 SAO 代理。对于 CAO 来说，使用的则是另一种机制，后面会讨论它。下面的代码用于在客户端获取 SAO 代理。

```
ICalculator calc = (ICalculator)Activator.GetObject (
  typeof (ICalculator),
  "tcp://localhost:8005/MyRemotedCalculator");
```

如果用 Spring.NET 构建客户端应用，可以使用 SaoFactoryObject 类在 IoC 容器内获取 SAO 代理的引用。下面的代码取自 Remoting 快速入门，可以参考其中的用法：

```
<object id="calculatorService" type="Spring.Remoting.SaoFactoryObject,
Spring.Services">
  <property name="ServiceInterface"
value="Spring.Calculator.Interfaces.IAdvancedCalculator,
Spring.Calculator.Contract" />
  <property name="ServiceUrl"
value="tcp://localhost:8005/RemotedSaoSingletonCalculator" />
</object>
```

ServiceInterface 属性用来指定要创建的代理的类型（按：即远程对象实现的接口）。ServiceUrl 属性则是远程对象的 URI。

IoC 容器内依赖 ICalculator 对象的其它对象现在就可以直接通过 calculatorService 来引用 ICalculator 接口的远程实现了。由于对象的依赖关系由 IoC 容器管理，所以可以很方便的更换 ICalculator 对象：如果要用一个本地对象替换远程对象，只需修改配置，而不用修改代码。基于接口的编程方式也使客户端的单元测试变得很简单，因为我们可以用接口的模拟实现类进行测试。同时，客户端和服务端也可独立开发。如果客户端和服务端由两个团队同时开发，就可以提高生产力。两个团队在开发前就接口达成共识，客户端团队可以独立开发出一个简单但可用的实现，一旦服务端实现完成，就可与之整合。

20.4. CAO 最佳实践

在.NET 中，我们可以通过 Administrative 类型注册来进行客户端激活对象（CAO）的创建（编程方式或标准的.NET Remoting 配置）。注册以后就能够用 new 操作符创建远程对象的代理，但需要将远程对象的实现部署在客户端。前面讲到过，在开发分布式系统时，这并不是好的做法。最好的方法是创建一个服务端激活的工厂类，由它来向客户端返回 CAO。在使用 IoC 容器时，我们已经知道不必为每个类型都创建工厂类，可以使用 Spring.NET 提供的的通用工厂对象（按：参见第四章有关 IFactoryObject 的章节）；对于远程对象来说，我们同样也可以用一个通用的（SAO 类型的）工厂对象来返回容器中的 CAO 对象。这个工厂的类型就是 CaoExporter。在客户端，可以用 CaoFacotryObject 获取一个（CAO 的）引用。CaoFacotryObject 可以使用含参构造器来远程创建 CAO 对象。这样既避免了需要为每个远程类型创建工厂类的繁复工作，也不必在客户端或服务端为任何 CAO 对象进行注册。因为在跨远程边界返回 MarshalByRefObject 的子类对象后（按：此处是指前文提到的服务端激活的 CaoExporter 工厂对象），就可以通过它请求 CAO 的引用。有关此最佳实践的详细信息，请参考本章最后一节 20.8,参考资源一节中的链接资源。

20.5. 在服务端注册 CAO

要在服务端发布 CAO，应该在容器中将 CaoExporter 的目标对象定义声明为“prototype”，也就是说 singleton 属性应该为 false。这样，每次向容器请求都会返回一个新的对象。在 CaoExporter 类内部，将一个 CaoRemoteFactory 类（ICaoRemoteFactory 接口的实现类）的实例传递给 RemotingServices.Marshal 方法，将目标对象发布为 CAO。CaoRemoteFactory 类从 IoC 容器请求目标对象并动态的为其创建远程代理。注意远程对象的默认租赁期为无限（也就是说 InitializeLifetimeService()方法的返回值为 null）。

我们通过一个例子来说明问题，下面是一个简单的计算器对象：

```
<object id="prototypeCalculator"
type="Spring.Calculator.Services.AdvancedCalculator,
Spring.Calculator.Services"
        singleton="false">
```

```
  <constructor-arg type="int" value="217" />
</object>
```

为将这个对象发布为 CAO，我们可以在容器中配置一个 CaoExporter 对象，如下所示：

```
<object id="caoCalculator" type="Spring.Remoting.CaoExporter,
Spring.Services">
  <property name="TargetName" value="prototypeCalculator" />
  <property name="Infinite" value="false" />
  <property name="InitialLeaseTime" value="2m" />
  <property name="RenewOnCallTime" value="1m" />
</object>
```

注意“TargetName”属性的值是一个非 singleton 的 AdvancedCalculator 对象名，而非其引用。

此外，我们也可以用 Remoting 的 schema 来配置 CAO 对象定义，如下：

```
<r:caoExporter targetName="prototypeCalculator" infinite="false">
  <r:lifeTime initialLeaseTime="2m" renewOnCallTime="1m" />
</r:caoExporter>
```

20.5.1. 向 CAO 对象应用 AOP 通知

CaoExporter 也可以将应用了 AOP 通知的对象发布为 CAO。下面的代码也取自 Remoting 快速入门，为计算器对象应用了日志环绕通知：

```
<object id="prototypeCalculatorWeaved"
type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">

  <property name="targetSource">
    <object type="Spring.Aop.Target.PrototypeTargetSource,
Spring.Aop">
      <property name="TargetObjectName" value="prototypeCalculator" />
    </object>
  </property>
  <property name="interceptorNames">

    <list>
      <value>ConsoleLoggingAroundAdvice</value>
    </list>
  </property>
</object>
```

如果您觉得这段配置难以理解，可以参考第十二章,使用 Spring.NET 进行面向方面的开发。随后，CaoExporter 对象就可以使用 prototypeCalculatorWeaved 对象名了：

```
<r:caoExporter targetName="prototypeCalculatorWeaved"
infinite="false">
  <r:lifeTime initialLeaseTime="2m" renewOnCallTime="1m" />
</r:caoExporter>
```

20.6. 在客户端访问 CAO

在客户端，可以用 CaoFactoryObject 获取 CAO 的引用，如下：

```
<object id="calculatorService" type="Spring.Remoting.CaoFactoryObject,
Spring.Services">
  <property name="RemoteTargetName" value="prototypeCalculator" />
  <property name="ServiceUrl" value="tcp://localhost:8005" />
</object>
```

这个对象定义可以获取上一节导出的远程计算器对象。RemoteTargetName 属性的值就是在服务端导出远程对象时 CaoExporter.TargetName 属性的值。通过这种方法，我们可以用标准的依赖注入技术来获取实现了 IAdvancedCalculator 接口的远程对象引用。（当然，这并不是说客户端应该将远程对象当作进程内的本地对象看待。）

另外，也可以用 Reomting 的 Schema 来简化对象定义，并可利用该 schema 在编辑时进行代码提示。

```
<r:caoFactory id="calculatorService"
              remoteTargetName="prototypeCalculator"
              serviceUrl="tcp://localhost:8005" />
```

20.6.1. 向客户端的 CAO 对象应用 AOP 通知

向客户端的 CAO 对象应用 AOP 通知与向普通对象应用 AOP 通知是一样的。只要将 CaoFactoryObject 创建的对象作为目标对象即可，比如上面配置中的 calculatorService。

20.7. Remoting Schema

请按照第二十四章,与 Visual Studio.NET 集成中讲到的步骤将 XSD 文件安装进 VS.NET。

20.8. 参考资源

可以参考两篇论文,MSDN 网站的 Implementing Broker with .NET Remoting Using Client-Activated Objects 和 Glacial Components 网站的 Step by Step guide to CAO creation through SAO class factories，内容都是关于如何创建一个标准 SAO 工厂用以返回 CAO 的。

第二十一章. .NET 企业服务

21.1. 简介

Spring.Services 命名空间的目的是为业务服务提供位置的透明性。我们相信使用普通的接口和.NET 类，用户应该可以用最简单的方式实现服务。我们也认为在将某个服务发布给客户端的时，应该只关心服务的配置，而无需关心服务的实现。

在 Spring.Services 命名空间的支持下,可以用容器中的服务导出对象将*任意*的普通对象发布为 web 服务，企业服务组件或远程对象。

我们相信这也是迁移到 Indigo 最简单的方式。只要在设计时使用了接口并将其实现为普通的.NET 类型，那么一旦 Indigo 发布，我们就应该能实现 Indigo 的导出类。

目前，使用现有的导出类和 Spring.NET 配置文件，我们可以将实现了某个接口的任意对象发布为企业服务。

21.2. 服务组件

.NET 的服务组件允许我们使用 COM+服务提供的功能，比如声明式的分布式事务管理、基于角色的安全性验证、对象池消息等等。使用这些服务的类型必须要继承自 System.EnterpriseServices.ServicedComponent 类,类型和程序集也需要应用相关的特性,并且需要将服务组件注册到 COM+目录以配置应用程序。在.NET 中，访问和使用 COM+服务的相关内容称为.NET 企业服务。

如果使用 Spring.NET 的远程框架，很多服务类都不需要从 ServicedComponent 继承。您可能对如何将一个普通的对象发布为服务组件，以及如何使客户端以位置透明的方式访问服务组件很感兴趣。使用 Spring.NET 的 ServicedComponentExporter、EnterpriseServicesExporter 和 ServicedComponentfactory 类，就可以很方便的完成这些工作，服务对象不需要继承 ServicedComponent 类，并且为程序集进行强命名和使用 RegSvcs.exe 工具这样的手工工作也可以自动完成。

注意本章的内容并没有对.NET 企业服务做深入的讨论，有兴趣的话可以参考 Christian Nagel 的著作《Enterprise Services with the .NET Framework》。

21.3. 服务端

在服务端导出服务组件时，最大的问题是组件必须要位于文件系统的某个物理程序集内，这样才能将之与 COM+服务注册。同时，要想成功注册，程序集必须使用强命名，这使注册过程变得更加复杂了。

Spring.NET 提供了两个类来处理这些工作：

- Spring.Enterprise.ServicedComponentExporter 负责将一个类导出为服务组件，并确保它继承了 ServicedComponent。同时也允许为组件应用类型级和方法级的特性，以便定义事务行为和消息队列。
- Spring.Enterprise.EnterpriseServicesExporter 则相当于一个 COM+应用程序，可以用它来指定要在应用程序中发布的组件列表（按：即由 ServicedComponentExporter 导出的服务，见下文例子）、应用程序名称和其它程序集级的特性。

现在，假如我们有一个简单的服务接口和一个实现类，如下：

```
namespace MyApp.Services
{
    public interface IUserManager
    {
        User GetUser(int userId);
        void SaveUser(User user);
    }

    public class SimpleUserManager : IUserManager
    {
        private IUserDao userDao;
        public IUserDao UserDao
        {
            get { return userDao; }
            set { userDao = value; }
```

```
        }

        public User GetUser(int userId)
        {
            return UserDao.FindUser(userId);
        }

        public void SaveUser(User user)
        {
            if (user.IsValid)
            {
                UserDao.SaveUser(user);
            }
        }
    }
}
```

在配置文件中，相应的对象定义如下：

```
<object id="userManager" type="MyApp.Services.SimpleUserManager">
    <property name="UserDao" ref="userDao"/>
</object>
```

假设我们需要将 userManager 发布为一个服务组件以便使用事务功能。首先需要用 ServicedComponentExporter 导出我们的服务，如下：

```
<object id="MyApp.EnterpriseServices.UserManager"
type="Spring.Enterprise.ServicedComponentExporter, Spring.Services">

    <property name="TargetName" value="userManager"/>
    <property name="TypeAttributes">
        <list>
            <object
type="System.EnterpriseServices.TransactionAttribute,
System.EnterpriseServices"/>
        </list>
    </property>

    <property name="MemberAttributes">
        <dictionary>
            <entry key="*">
                <list>
                    <object
type="System.EnterpriseServices.AutoCompleteAttribute,
System.EnterpriseServices"/>
```

```
                </list>

            </entry>
        </dictionary>
    </property>
</object>
```

ServicedComponentExporter 对象会用组合方式为 SimpleUserManager 对象创建一个代理对象，代理对象继承了 ServicedComponent，并且代理 SimpleUserManager 对象上的方法调用。同时，导出类会为代理对象应用 TransactionAtribute 特性、并为其中的所有方法应用 AutoCompleteAttribute 特性。

下面，需要为组件的宿主 COM+程序配置一个导出类对象：

```
<object id="MyComponentExporter"
type="Spring.Enterprise.EnterpriseServicesExporter, Spring.Services">

    <property name="ApplicationName" value="My COM+ Application"/>
    <property name="Description" value="My enterprise services
application."/>
    <property name="AccessControl">
        <object
type="System.EnterpriseServices.ApplicationAccessControlAttribute,
System.EnterpriseServices">
            <property name="AccessChecksLevel"
value="ApplicationComponent"/>
        </object>

    </property>
    <property name="Roles">
        <list>
            <value>Admin : Administrator role</value>
            <value>User : User role</value>

            <value>Manager : Administrator role</value>
        </list>
    </property>
    <property name="Components">
        <list>

            <ref object="MyApp.EnterpriseServices.UserManager"/>
        </list>
    </property>
    <property name="Assembly" value="MyComPlusApp"/>
```

```
</object>
```

这个导出类对象会将 Components 属性中列出的所有代理类放入以 Assembly 属性值为名的程序集中，为其设置强命名，并注册为以 ApplicationName 属性值为名的 COM+服务应用。如果指定的应用尚未存在，就会用配置中的 Description、AccessControl 和 Roles 属性值新建并配置该应用程序。

21.4. 客户端

由于服务组件类是动态生成与注册的，所以客户端不能在代码中使用 new 操作符来初始化服务对象，而要通过 Spring.Enterprise.ServicedComponentFactory 获取。通过指定组件的配置模板或远程服务组件的名称，就可以用该类获得服务对象，如下：

```
<object id="enterpriseUserManager"
type="Spring.Enterprise.ServicedComponentFactory, Spring.Services">
    <property name="Name"
value="MyApp.EnterpriseServices.UserManager"/>
    <property name="Template" value="userManager"/>
</object>
```

随后，在 IoC 容器中就可以将这个对象注入到任何需要使用 IUserManager 接口的对象中，就和使用普通的 SimpleUserManager 对象没有区别。我们在例子中也看到，使用实现了某个接口的普通.NET 类来创建服务，我们确实可以通过配置使服务做到位置透明性。

第二十二章. Web 服务

虽然目前.NET 对 web 服务支持的非常好，Spring.NET 认为还是有几个方面可以改进。

22.1. 服务端

首先，.NET 在.asmx 文件中保存 Web 服务请求和服务对象的关联关系，这些.asmx 文件不管有用没用都得放在那儿。

第二，Spring.NET 希望能通过 IoC 容器对 web 服务进行依赖注入。一般说来 web 服务总会依赖其它服务对象，所以，如果能用配置方式来选择服务对象，这个功能就相当强大了。

最后，目前在.NET 中 Web 服务的创建完全是一个实现（特定类型）的过程。多数服务（虽不能说是全部）都应实现为使用粗粒度服务接口的普通类型，并且，

某个对象能否发布为远程对象、web 服务还是企业（COM+）组件应该只与配置有关，而不应该取决于它的实现方式。

21.1.1.消除对.asmx 文件的依赖

ASP.NET 用.aspx 文件来保存表示层代码，用 code-behind 文件中的类保存应用逻辑，.aspx 与类代码文件各有分工；但 web 服务却不同，Web 服务的逻辑完全是在 code-behind 的类中实现的。.asmx 文件并没有什么真正的用途，实际上，这个文件既没有必要存在、也不应该存在。

在将 WebServiceFactoryHandler 类注册为响应*.asmx 请求的 HTTP Handler 之后，开发人员就可以在 IoC 容器中用标准的 Spring.NET 对象定义来发布 Web 服务。

下面通过一个例子来说明这个过程，首先看下面的 web 服务...

```
namespace MyComany.MyApp.Services
{
    [WebService(Namespace="http://myCompany/services")]
    public class HelloWorldService
    {
        [WebMethod]
        public string HelloWorld()
        {
            return "Hello World!";
        }
    }
}
```

这是一个普通的类型，其中应用了两个.NET 特性：方法级的[WebMethod]及类型级的[WebService]。开发人员可以在 VS.NET 中创建这个服务类。

要想将这个对象发布为 web 服务，只需依次作以下操作：

1.在 web.config 文件中，将 Spring.Web.Services.WebServiceFactoryHandler 注册为响应.asmx 请求的 HTTP Handler。*

```
<system.web>
    <httpHandlers>
        <add verb="*" path="*.asmx"
type="Spring.Web.Services.WebServiceHandlerFactory, Spring.Web"/>
    </httpHandlers>
</system.web>
```

当然，也可以为*.asmx 请求注册其它的 Http Handler 来处理意外情况，不过一般没这个必要，因为如果 WebServiceFactoryHandler 无法在容器中找到与某个请求相匹配的对象定义，就会求助于.NET 标准的 Http Handler 去查找.asmx 文件。

2.为 web 服务创建 XML 对象定义

```
<object name="HelloWorld.asmx"
type="MyComany.MyApp.Services.HelloWorldService, MyAssembly"
abstract="true"/>
```

注意，为 web 服务创建的对象定义并不要求一定是抽象的（通过设置 abstract="true"），但设为抽象可防止创建不必要的服务实例，因为.NET 基础框架会在后台自动为每次请求创建新的服务实例，Spring.NET 需要做的只是提供服务的类型名，即便标记为抽象，也能从容器中获取服务对象的实例。所以 Spring.NET 建议将 Web 服务的对象定义显示标记为抽象。

现在就能以对象定义中 name 属性的值为服务名来访问这个 Web 服务了：

```
http://localhost/MyWebApp/HelloWorld.asmx
```

22.1.2.向 web 服务中注入依赖项

为了方便讨论，我们来修改 HelloWorld 方法的实现，以便返回一个可配置的消息。

当然了，我们可以用某种消息定位器来返回一条合适的信息，但是这个“消息定位器”也是需要自己实现的。如果我们在整个应用程序中都用依赖注入来配置对象，对 web 服务反倒去用服务定位器，难免有些古怪。

理想的情况是，我们在 web 服务类中为消息定义一个属性，然后让 Spring.NET 注入这个属性的值：

```
namespace MyApp.Services
{
    public interface IHelloWorld
    {
        string HelloWorld();
    }

    [WebService(Namespace="http://myCompany/services")]
    public class HelloWorldService : IHelloWorld
    {
        private string message;
```

```
        public string Message
        {
            set { message = value; }
        }

        [WebMethod]
        public string HelloWorld()
        {
            return this.message;
        }
    }
}
```

在这里，Spring.NET 标准的依赖注入机制遇到了一个问题：Web 服务的初始化不受 Spring.NET 控制，而是由.NET 框架在内部进行的，所以很难把握配置对象的时机。

解决的方法是创建一个动态的服务端代理，用以包装和配置 web 服务。这样，.NET 框架从 Spring.NET 获取的实际上是这个代理类的类型，代理对象初始化后，就会向 Spring.NET 的应用程序上下文请求真正的服务对象来处理 web 服务请求。

这种代理方式要求用 Spring.Web.Services.WebServiceExporter 类显式的导出 web 服务。对于本例来说，别忘了配置 Message 属性：

```
<object id="HelloWorld" type="MyApp.Services.HelloWorldService,
MyApp">
    <property name="Message" value="Hello, World!"/>

</object>

<object id="HelloWorldExporter"
type="Spring.Web.Services.WebServiceExporter, Spring.Web">
    <property name="TargetName" value="HelloWorld"/>
</object>
```

WebServiceExporter 类会将服务对象中[WebService]和[WebMethod]特性的值复制到代理类中。请注意这些值可以被 WebServiceExporter 类的属性覆盖。

对接口的需求

要实现某些高级的特性，比如将 AOP 代理发布为 web 服务（允许向 web 方法应用 AOP 通知），Spring.NET 需要这些目标对象实现某个（服务）接口。

只有定义在接口中的方法才会被 WebServiceExporter 类导出。

22.1.3.将 PONO 导出为 web 服务

既然我们为 web 服务创建了服务端代理，就没有必要在 web 服务类中应用诸如 WebMethod 之类的特性了。因为此时.NET 基础框架是看不到“真正”的服务对象的，.NET 真正接触到的是应用在代理类上的特性。

也就是说，我们完全可以从服务类中去掉 WebService 和 WebMethod 特性，只留下一个普通的.NET 对象（一个 PONO）。对于上面的例子来说，去掉这些特性后仍然可以正常工作，因为代理类生成器可以自动的在要导出的接口方法上应用 WebMethod 特性。

不过，这还不是最理想的解决方案。去掉了 WebService 和 WebMethod 特性也就无法再保留它们定义的可选信息，比如服务的命名空间、描述、事务模式等等。但若为这些信息而仍在服务类中应用特性，让代理生成器将这些特性的值复制到代理类中去，又违背了我们的初衷。

更好的方法是在 WebServiceExporter 的对象定义中设置所有必要的值，如下...

```
<object id="HelloWorldExporter"
type="Spring.Web.Services.WebServiceExporter, Spring.Web">
    <property name="TargetName" value="HelloWorld"/>
    <property name="Namespace" value="http://myCompany/services"/>
    <property name="Description" value="My exported HelloWorld web
service"/>
    <property name="MemberAttributes">
        <dictionary>
            <entry key="HelloWorld">
                <object type="System.Web.Services.WebMethodAttribute,
System.Web.Services">
                    <property name="Description" value="My
Spring-configured HelloWorld method."/>
                    <property name="MessageName" value="ZdravoSvete"/>
                </object>
            </entry>
        </dictionary>
    </property>
</object>

// or, once configuration improvements are implemented...
<web:service targetName="HelloWorld"
namespace="http://myCompany/services">
    <description>My exported HelloWorld web service.</description>
    <methods>
        <method name="HelloWorld" messageName="ZdravoSvete">
```

```
            <description>My Spring-configured HelloWorld
method.</description>
        </method>
    </methods>
</web:service>
```

根据上面的配置，Spring.NET 可以为目标对象实现的所有接口生成 web 服务代理，并添加必要的特性。同时，将 web 服务的元数据从类实现移到了配置文件中，并允许将任意类型发布为 web 服务。

通过设置 WebServiceExporter 类的 Interfaces 属性，也可以只发布服务类实现的部分接口...

关于分布式对象

不能因为 Spring.NET 能够把任意对象发布为 web 服务，就想把所有对象都发布为 web 服务。我们仍然要受分布式计算原则的约束，我们要保证所发布的 web 服务是有意义的，同时 web 方法的参数和返回值都是可序列化的。

同样，还是要了解一些这方面的基础知识，当需要选择使用 web 服务（或者更广义的说，远程服务）还是本地服务时，应该能够做出正确的判断。

22.1.4.将 AOP 代理发布为 web 服务

很多时候需要将 AOP 代理发布为 web 服务。比如说，假设我们有一个被 AOP 代理包装好的服务类，打算同时通过本地和远程（使用 web 服务）访问它。本地客户端只需要直接获取 AOP 代理的引用，而远程客户端获取则是的 web 服务代理的引用，由它来代理对 AOP 代理的调用，然后再由 AOP 代理负责对目标对象的调用。

要将 AOP 代理发布为 web 服务实际上是相当简单的；因为 AOP 代理和其它对象也没什么两样，所要做的无非就是把 WebServiceExporter 对象定义的 TargetName 属性值设为 AOP 代理对象的 id（或者 name、或者别名）。请看下面的代码...

```
<object id="DebugAdvice" type="MyApp.AOP.DebugAdvice, MyApp"/>

<object id="TimerAdvice" type="MyApp.AOP.TimerAdvice, MyApp"/>

<object id="MyService" type="MyApp.Services.MyService, MyApp"/>

<object id="MyServiceProxy"
type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
    <property name="TargetName" value="MyService"/>
    <property name="IsSingleton" value="true"/>
    <property name="InterceptorNames">
        <list>
```

```
            <value>DebugAdvice</value>
            <value>TimerAdvice</value>
        </list>
    </property>
</object>

<object id="MyServiceExporter"
type="Spring.Web.Services.WebServiceExporter, Spring.Web">
    <property name="TargetName" value="MyServiceProxy"/>
    <property name="Name" value="MyService"/>
    <property name="Namespace" value="http://myApp/webservices"/>
    <property name="Description" value="My web service"/>
</object>
```

这样，Web 服务方法的每次调用都会被 AOP 代理拦截，然后将配置中的 DebugAdvice 和 TimerAdvice 两个通知应用到方法上。

22.1.5.客户端的问题

在客户端，.NET 本身最大的缺陷是将客户端代码绑定在了代理类而非服务接口上。除非按照 Jubal Lowy 的著作《Programming .NET Components》中提到的方法，手工让代理类实现某个服务接口，否则应用程序代码的灵活性会很低，同时，如果需要使用一个新的、改进过的 Web 服务类，或者打算用一个本地服务替换掉 web 服务时，相应的更换工作会很复杂。

在 Spring.NET 的支持下，可以很方便的为 web 服务创建实现了指定服务接口的客户端代理。

22.2. 客户端

在客户端，.NET 本身最大的缺陷是将客户端代码绑定在了代理类而非服务接口上。除非按照 Jubal Lowy 的著作《Programming .NET Components》中提到的方法，手工让代理类实现某个服务接口，否则应用程序代码的灵活性会很低，同时，如果需要使用一个新的、改进过的 Web 服务类，或者打算用一个本地服务替换掉 web 服务时，相应的更换工作会很复杂。

在 Spring.NET 的支持下，可以很方便的为 web 服务创建实现了指定服务接口的客户端代理。

22.2.1. WebServiceProxyFactory

如果使用 VS.NET 或 WSDL 命令行工具，生成的 web 服务代理类最大的问题是没有实现任何接口。这种客户代码和 Web 服务紧耦合的做法使将来不可能在不修改和重新编译客户代码的前提下更换 web 服务代理类。

Spring.NET 有一个很简单的 IFactoryObject 实现类，可以生成“代理的代理“（乍一听好像挺蠢）。简单来说，
Spring.Web.Services.WebServiceProxyFactory 类会为 VS.NET/WSDL 工具生成的代理再创建一个代理，由这个代理去实现指定的服务接口（这就解决了上一段提到的问题）。

现在用一个例子也许更能说明问题；假如我们打算将下面的接口发布为 web 服务...

```
namespace MyCompany.Services
{
    public interface IHelloWorld
    {
        string HelloWorld();
    }
}
```

在客户端，为了能通过此接口引用 web 服务，需要将下面的对象定义添加到应用程序上下文的配置中：

```
<object id="HelloWorld"
type="Spring.Web.Services.WebServiceProxyFactory, Spring.Services">
    <property name="ProxyType" value="MyCompany.WebServices.HelloWorld,
MyClientApp"/>
    <property name="ServiceInterface"
value="MyCompany.Services.IHelloWorld, MyServices"/>
</object>
```

有一点很重要，上面指定的服务接口不必一定是 IHelloWorld 接口，只要存在签名吻合的方法（一种 duck typing（按：“duck typing”是 ruby 语言专家 Dave Thomas 对动态类型的描述，意思是说，如果某个东西走起来象鸭子，叫起来也象鸭子，那么它可能就是一个鸭子。反映到语言中，就是如果一个对象的方法名和某个类型匹配，那么这个对象就可以看做是那个类型）），Spring.NET 就能正确的创建代理。如果找不到匹配的方法，Spring.NET 就会抛出异常。

如果可以同时控制客户端和服务端的设计，那么最好能确保服务端的 web 服务类实现了某个服务接口，特别是打算用 Spring.NET 的 WebServiceExporter 发布 web 服务时，因为该类要求服务类必须实现一个以上接口。

22.2.2. WebServiceClientFactory

另一个 IFactoryObject 的实现类 WebServiceClientFactory 用以动态生成 Web 服务代理：

```
<object id="HelloWorld"
type="Spring.Web.Services.WebServiceClientFactory, Spring.Services">
    <property name="ServiceUrl"
value="http://www.MyCompany.com/HelloWorld.asmx"/>
    <property name="ServiceInterface"
value="MyCompany.Services.IHelloWorld, MyServices"/>
</object>
```

当通过 Web 服务处理强类型 DataSet 的时候，这种代理相当有用。我们且不论这种代理有什么优缺点，先来看.NET 自己的代理生成行为：在处理强类型 DataSet 时，.NET 会为其创建包装类型（wrapper types）。并且，会为每个使用强类型 DataSet 的 Web 服务都创建包装类型，整个解决方案很快会被许多无关紧要的类搞的一团糟。而 Spring.NET 创建的代理则允许开发人员直接引用强类型 DataSet，从而避免了这些问题。

第二十三章. Windows 后台服务

23.1. 备注

除了本文档，读者还可以参考位于 examples\Spring\Spring.Examples.WindowsService 目录下的示例程序，以便对这部分内容有更好的理解。同时也请关注 Spring.NET 网站上本文档的最新版。

23.2. 简介

开发人员一般都会使用 VS.NET 的向导来创建 Windows 服务。虽然这个过程并不复杂，但却难免重复，也没有将框架的代码（与 Windows 服务相关的代码）与应用程序的逻辑代码分离开来。一般情况下，我们认为这是件“坏事”，当然您也可以不同意这一观点。

既然 Spring.NET 能够显式管理 singleton 对象的生命周期，它就能很自然的管理 Windows 服务的生命周期。所以，我们可以很方便的将 Spring.NET 的应用程序上下文发布为 Windows 服务。开始与停止服务就对应于创建和销毁应用程序上下文及其中的对象。在开发 Windows 服务时，这为我们提供了一种高层次的方法来确定哪些对象需要被创建和销毁。

为此，Spring.NET 需要“安装”一个“物理”的服务（按：即一个物理的程序集），以作为可以响应任意多客户端请求的“逻辑”服务——这些逻辑上独立的服务都运行于请求它们的应用程序域内。默认情况下，只需要将可以执行文件复制到特定的目录中去，即可完成服务的布署和安装。

目前，这个物理服务是 Spring.Services.WindowsService.Process.exe。这个程序大量使用了 Spring.Services.WindowsService.Common.dll 程序集中的类和接口。如果您要实验本章讲述的内容，也应该引用这个程序集。

将框架代码和应用程序代码分离可以为我们带来很多好处。客户端也可以向远程目录中复制新的程序集来安装一个新的服务[5]。

23.3.Spring.Services.WindowsService.Process.exe 应用程序

23.3.1.安装

可通过两种方法安装服务应用程序：使用.NET SDK 的 installutil.exe 工具（按：该工具并非由 SDK 提供，而是随.NET 框架一起发布，位于.NET 框架的安装目录下），或者使用更为简单的 Spring.Services.WindowsService.Installer.exe；只不过前者是标准方式，后者则更为灵活。Spring.NET 提供的工具允许为服务自定义名称及显示名称，也允许用不同的名称多次安装同一程序集。在很多场合这种能力都非常有用，比如在某些时候，我们不打算基于同一个物理服务运行多个逻辑服务时。

注意，服务会安装在系统账户下（将服务安装至特定账户看来是 Windows XP 的一个 bug）。

读者可以在 MSDN 网站上找到 installutil 的*文档*，而 Spring.Services.WindowsService.Installer.exe 命令行的用法如下：

```
Spring.Services.WindowsService.Installer.exe

usage:
 install service-exe-path service-display-name service-name
 uninstall service-name      [i|u] service-exe-path
service-display-name service-name
```

例如，要安装一个服务，可以用如下命令：

```
... install  Spring.Services.WindowsService.Process.exe
"Spring.Service Support" spring-service
```

要卸载这个服务，可以用以下命令：

```
... uninstall  spring-service
```

23.3.2. 配置

Spring.Services.WindowsService.Process.exe 所需要的参数可以配置在标准的.NET 配置文件中（也可以包括 log4net 的设置，建议参考 log4net 的文档）。

下面的配置文件定义了由服务的应用程序上下文：

```
<configuration>

    <configSections>
        <section name="log4net"
type="System.Configuration.IgnoreSectionHandler" />
        <sectionGroup name="spring">
            <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core" />

            <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
        </sectionGroup>
    </configSections>

    <spring>
        <context type="Spring.Context.Support.XmlApplicationContext,
Spring.Core">
            <resource uri="file://~/service-process-definition.xml" />

        </context>
    </spring>

    <system.runtime.remoting>
    <application>
        <channels>
        <channel ref="http" port="1234" />
        </channels>

    </application>
    </system.runtime.remoting>

  <log4net>
    <!-- see
http://logging.apache.org/log4net/release/manual/introduction.html
-->
```

```
    <appender name="RollingFile"
type="log4net.Appender.RollingFileAppender">
      <layout type="log4net.Layout.PatternLayout">

        <conversionPattern value="%d [%t] %-5p %c{1} - %m%n" />
      </layout>
      <file value="logs/Spring.Service.Process.log" />
      <appendToFile value="true" />
      <maximumFileSize value="500KB" />
      <maxSizeRollBackups value="5" />

    </appender>
    <appender name="OutputDebugString"
type="log4net.Appender.OutputDebugStringAppender">
      <layout type="log4net.Layout.PatternLayout">
        <conversionPattern value="%d{HH:mm:ss,fff} %-5p %c{2}(line:%L)
- %m%n" />
      </layout>
    </appender>

    <root>
      <level value="OFF" />
    </root>
      <logger name="Spring.Services">
          <level value="ALL" />
      <appender-ref ref="RollingFile" />

      <appender-ref ref="OutputDebugString" />
      </logger>
  </log4net>

</configuration>
```

从配置中可以看出，对象定义是保存在其它文件中的，我们来考虑一下这些文件中的对象定义。

首先，要注意，要想“定位”该服务（也就是确定服务安装在哪儿，安装目录参照上面文件中所定义的部署目录），需要按以下格式定义一个对象：对象的名称不重要，但它必须是一个 IObjectFactoryPostProcessor，能够被容器自动应用。

```
<!-- provides access to the ${spring.services.process.base.dir} property
-->
<object
    name="localizer"
```

```
    type="Spring.Services.WindowsService.Common.Localizer+ForProcess,
Spring.Services.WindowsService.Common">
    <!-- change this to access the property with another prefix, for
example ${foo.process.base.dir}
    <property name="prefix" value="foo"/>
    -->
</object>
```

其中 Prefix 属性的值是用来替换下面字符串中的占位符的：

```
public static readonly string SpringServicesProcessBaseDirFormat =
"{0}.process.base.dir";
```

但通常并不需要配置该属性，因为它有默认值：

```
public static readonly string DefaultPrefix = "spring.services";
```

应用程序上下文中最重要的对象当然是由服务所运行的对象。在服务对象定义中，可以（也应该可以）配置要部署的路径名；下面的配置使用了前面定义的 localizer 对象的属性，所以可以不指定绝对路径名（例如：C:\Spring\Services）：

（按：localizer 对象为同一容器中的对象提供类似属性值替换的功能，也就是，如果正确定义了 localizer 对象，其它对象就可以用一些变量名——实际是占位符，来表示某些信息，比如服务所在的路径等等。默认情况下，如果不对 localizer 对象的 Prefix 属性赋值，那么它的默认值就是 spring.services，若要访问服务应用程序的完整路径名，就可以用${spring.services.application.fullpath}变量名；若设置了 Prefix 的值，比如说设为 foo，那么访问完整路径名就要使用${foo.application.fullpath}了。请参考后面的 23.4.1.1 节。至于属性值替换的相关内容，请参考 IoC 容器一章。）

```
<object
    name="service"
    type="Spring.Services.WindowsService.Common.DefaultService,
Spring.Services.WindowsService.Common"
    init-method="Start"
    destroy-method="Stop">

    <property name="DeployPath"
value="${spring.services.process.base.dir}/deploy"/>
</object>
```

然后，这个对象就可以用 Spring.NET 的远程工具发布为远程对象（请注意要在配置文件中添加 remoting 的相应节点，如前面的代码所示）。

```
<object name="remoted.service" type="Spring.Remoting.SaoExporter,
Spring.Services">
    <property name="TargetName" value="service"/>

    <property name="ServiceName" value="SpringWindowsService.rem"/>
</object>
```

23.4. 将应用程序上下文发布为 Windows 服务

按照下面的形式和规则将应用程序打包，就能将应用程序上下文发布为 Windows 服务。这些规则和 ASP.NET 差不多，所以很容易掌握。

前面讲过，Spring.NET 的应用程序上下文将在一个特定的应用程序域中运行，这个应用程序域的宿主就是一个以 Windows 服务形式运行的进程：该进程可以同时运行多个应用程序上下文。

一个能运行为服务的完整应用程序由以下内容构成：

- .NET 配置文件 service.config：应用程序上下文就定义在该文件中。另外，CLR 也会使用该文件来配置和运行此程序的应用程序域。该文件的作用如同 ASP.NET 应用程序中的 Web.config 文件一样。
- 该项可选：一个 xml 文件（watcher.xml），定义应用程序的监视器（watcher）。监视器可以控制服务的自动重新部署，稍后将对此展开讨论。
- 推荐：象 ASP.NET 一样，用一个 bin 子目录来放置所有的程序集文件；当然了，完全可以将程序集和 service.config 放在同一目录下，但我们不建议这么做...

23.4.1. service.config

这是一个标准的.NET 配置文件，由服务的宿主应用程序域使用。其地位相当于 ASP.NET 应用程序中的 Web.Config 文件[6]。

我们应该在这个文件中定义 Spring.NET 的应用程序上下文。每当服务启动或停止时，应用程序上下文中所有 singleton 对象的生命周期方法也会相应的被调用。当然，singleton 对象是在服务启动时由容器自动创建的。有关 Spring.NET 容器中对象的生命周期，请参考 4.5.1.生命周期接口。下面是一个例子：

```
<configuration>

    <configSections>
        <sectionGroup name="spring">
```

```
            <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core" />
            <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />

        </sectionGroup>
    </configSections>

    <appSettings>
        <add key="port" value="10"/>
    </appSettings>

    <spring>

        <context type="Spring.Context.Support.XmlApplicationContext,
Spring.Core">
            <resource uri="file://~/service.xml" />
        </context>
    </spring>

</configuration>
```

本例中，应用程序上下文的对象定义（又一次！）定义在了一个单独的文件中（当然这要看个人喜好不同），其中惟一的“服务”是一个名为 echo 的对象（还有一个 PropertyPlaceholderConfigurer 对象，为的是让这个例子更真实一点）：

```
<objects xmlns="http://www.springframework.net"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.net
http://www.springframework.net/xsd/spring-objects.xsd">

    <object name="echo"
      type="Spring.Services.WindowsService.Samples.Echo,
Spring.Services.WindowsService.Tests"
      init-method="Start" destroy-method="Stop">
      <property name="port"><value>${port}</value></property>

    </object>

    <object id="configurer"
type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer,
Spring.Core">
      <property name="locations">
        <list>
```

```
            <value>file://~/service.config</value>

         </list>
      </property>
      <property name="configSections">
         <list>
            <value>appSettings</value>

         </list>
      </property>
   </object>

</objects>
```

23.4.1.1. 让应用程序了解自身的位置

有些信息是在服务的运行期确定的，我们无法预知它们的具体值。还好，在 XML 文件中，可以用 NAnt 风格的语法预定义一些变量，在服务对象定义中，可以用这些变量来查找信息。

这些变量包括：

- spring.services.application.fullpath：该变量在运行期会被应用程序的 AppDomain.BaseDirectory 属性值替换，也就是服务应用程序部署的位置。
- spring.services.application.name：该变量会被应用程序所在的子目录名替换。每个应用程序自然会部署在自己的目录下。

注意，只有在容器中定义了 localizer 对象后，这些变量名才可用：

```
<!-- provides access to the ${spring.services.application.*} properties
-->

<object
    name="localizer"

type="Spring.Services.WindowsService.Common.Localizer+ForApplication,
Spring.Services.WindowsService.Common">
    <!-- change this to access the property with another prefix, for
example ${foo.application.base.dir}
    -->
    <property name="prefix" value="myPrefix"/>
</object>
```

由上面的配置可见，我们可以用 localizer 对象改变这些变量的前缀名，然后用以下的格式使用它们：

```
<object name="simple"
  type="Spring.Services.WindowsService.Samples.Simple,
Spring.Services.WindowsService.Tests"
  init-method="Start" destroy-method="Stop">

  <constructor-arg index="0"
value="${myPrefix.application.name},${myPrefix.application.fullPath}"
/>
  <property name="AppName">
    <value>${myPrefix.application.name}</value>
  </property>
  <property name="AppFullPath">

    <value>${myPrefix.application.fullpath}</value>
  </property>
</object>
```

23.4.2. watcher.xml - 可选的配置

该文件为服务应用程序定义了一个监视器，用于在需要时自动的重新部署应用程序，该文件是可选的。

注意，我们可以为服务应用程序定义自己的监视器，监视器要实现 Spring.Services.WindowsService.Common.Deploy.IApplicationWatcher 接口，且在容器中其对象名应为 watcher。比如，我们可以创建一个监视器来监听文件系统的变化，并配置它的行为，使它只监视部分改变，而忽略其它变化。IApplicationWatcher 接口如下：

```
/// <summary>
Interface defining the contract for an application watcher.

<p>An application watcher is responsible to dispatch an
<see cref="IApplicationWatcherFactory">event</see> whenever it thinks
the
application has been updated.</p>
<p>Usually it should not raise other kind of events
as they are usually raised by the <see
cref="FileSystemApplicationWatcher"/>
that creates the watcher itself</p>
</summary>
```

```
<remarks>Usually instances of this interface need to be
disposed</remarks>
<seealso cref="DeployEventArgs"/>
<seealso cref="IDeployLocation"/>
<seealso cref="DeployEventType.ApplicationUpdated"/>
<seealso cref="DeployEventAggregator"/>
<seealso cref="IDeployLocation"/>
<seealso cref="DeployEventType"/>
public interface IApplicationWatcher : IDisposable
{
    /// <summary>

    /// The watched application
    /// </summary>
    IApplication Application {get; }

    /// <summary>
    /// Start to watch the application, using the given dispatcher to
    /// dispatch deply events
    /// </summary>
    /// <param name="dispatcher">the dispatcher used to raise deploy
events</param>

    void StartWatching (IDeployEventDispatcher dispatcher);

    /// <summary>
    /// Stop to watch the application.
    /// </summary>
    void StopWatching ();

    /// <summary>
    /// If physical events watched by this watcher should be filtered,
this methods
    /// will allow to set filters that allows and disallows the event to
be raised
    /// by the watcher.
    /// </summary>
    /// <param name="allows">the list of allowing filters</param>

    /// <param name="disallows">the list of disallowing filters</param>
    /// <seealso cref="FilteringSupport"/>
    /// <seealso cref="RegularExpressionFilter"/>
    void SetFilters (IList allows, IList disallows);
}
```

请注意该接口目前尚未最终确定，可能会在正式发布前有所改动（可能会影响监视器了解它所监视的应用程序的方式）。

下面是一个典型的 wathcer.xml 文件：

```
<objects>

    <object name='watcher'

type='Spring.Services.WindowsService.Common.Deploy.FileSystem.FileSystemApplicationWatcher'>
      <!--
      we can get access to the IApplication we are asked to monitor
      using a reference like the following
      -->
      <constructor-arg ref='.injected.application'/>

      <!-- sometimes the windows OS will decide to not give you the same case you see in explorer:
           in fact one should consider this OS case-insensitive with regard to file names ...
           The following property, true by default can however be tuned
      <property name="ignoreCase" value="false"/>
      -->

      <property name="includes">
        <list>
          <value>wwwroot/bin/*.*</value>
          <value>service.config</value>
          <value>service.xml</value>
        </list>

      </property>

      <!--
      <property name="excludes">
        <list>
          <value>Db/**/*.*</value>
          <value>Jobs</value>

          <value>Jobs</value>
          <value>**/*.log</value>
        </list>
      </property>
```

```
    -->

  </object>

</objects>
```

从上面代码中可以看出，可以将.injected.application 注入给 watcher 的构造器，来使监视器知道自己所监视的应用程序。.injected.application 所引用的是一个类型为 Spring.Services.WindowsService.Common.IApplication 的对象。

23.4.3. bin 目录 - 可选

该目录用于存放服务应用程序的所有程序集文件，如同 ASP.NET 应用程序中的 bin 目录一样。

将程序集放在单独的目录中不仅仅是一个规则，也是一个良好的编程习惯（这样可以将程序集文件与其它部分分隔开，不过您要愿意，完全可以用一个其它的目录（要相应的修改 service.config 文件），或者直接使用应用程序的根目录（也就是 bin 的父目录））。

请注意，服务应用程序的宿主进程会有自己的 PATH 环境变量。所以，如果 DLL 文件没有存放在服务器的系统路径下，就不能指望用[DllImport]特性来导入它们：因为我们都知道 CLR 的 fusion 算法是不考虑 PATH 环境变量的，所以如果某些程序集使用了非系统路径，那么用起来就可能遇到麻烦（SQLite 和 Firebird 的 ADO.NET provider 就是一个很好的例子）。

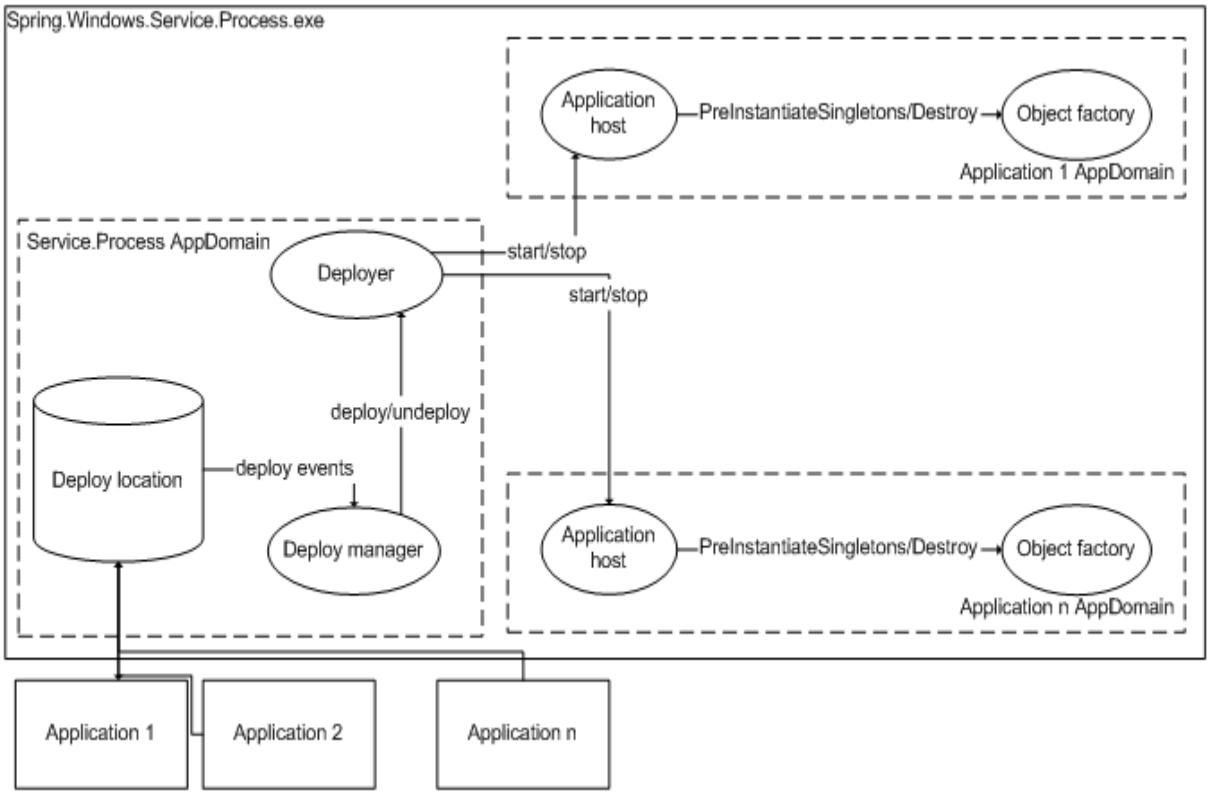
再次说明，可以把程序集放在应用程序路径下的任何子目录中，只要在配置文件（service.config）中写明位置即可：.NET 的程序集探测机制始终是有效的。

请注意，应用程序不需要引用或包含 Spring.NET 的任何程序集：任何程序集中的任何对象，只要它具有生命周期方法，就可以作为服务运行。Spring.NET 对此的支持仍是非侵入性的。

23.5. 自定义或扩展

应该说，对 Windows 服务的支持最初是源自一系列“扩展”的想法，这些想法清晰但是范围有限，主要与服务的部署方式有关：比如部署的位置（文件系统、zip 文件、邮件箱，url 等），（自动）更新的功能等等。

为更好的理解后面的内容，可以参考下面这幅图，这是 Spring.Services.WindowsService.Process.exe 在运行时的内部细节：



23.5.1. .config 文件

Spring.Services.WindowsService.Process.exe 的部分行为可由配置文件进行控制。请注意在使用 Spring.NET 发布 Windows 服务时，这个配置文件是最重要的，在将来的版本中，这个文件的作用可能会更大，也会更加灵活。

对于以标准方式布署的服务（即布署到文件系统中），如果使用了 watcher.xml 文件，更新的功能就可以由监视器进行控制，上文对此已经阐述过了。

然而，Windows 服务还有其它布署方式，比如说通过位于网络中某个位置的、或者通过邮件发送的 zip 文件来布署。

对于这种情况，就应该借助
Spring.Services.WindowsService.Common.Deploy.IDeployLocation 接口来表示布署位置了。>

请注意该接口扩展了 IDisposable 接口[7]，不过这么做是否合适，尚有争议：

```
/// <summary>
/// Interface defining how a deploy location should look like
/// </summary>
```

```
public interface IDeployLocation : IDeployEventSource, IDisposable
{
    /// <summary>
    /// The list of applications deployed at this location
    /// Usually non-valid applications are not listed
    /// </summary>
    /// <seealso cref="Application"/>

    IList Applications { get; }
}
/// <summary>
/// Interface defining the contract for an object acting as the source
of
/// deploy events (application added, removed, updated)
/// </summary>
/// <seealso cref="DeployEventArgs"/>
/// <seealso cref="DeployEventHandler"/>

public interface IDeployEventSource
{
    /// <summary>
    /// The multicaster for deploy events
    /// </summary>
    event DeployEventHandler DeployEvent;
}
```

---

[6] log4net 用户请注意，在处理相对文件路径时，文件 appender 是以应用域代码所在的路径为基础的。所以如果使用 log4net 的话，这种机制对 Spring.NET 来说很方便的，因为我们所指定的日志文件就位于包含服务应用程序的目录的相对路径中。

[7] 这是因为服务所部署的位置可能保存了某些需要释放的资源，比如网络连接，被锁定的文件等等。

第二十四章. 与 Visual Studio.NET 集成

24.1. XML 编辑与验证

(Available in 1.0)

如果您习惯于用自己钟爱的 XML 编辑器编写 XML 文档，那么应该会对本章的大部分内容感兴趣。在运行期，Spring.NET 使用对象定义 schema 验证 XML 数据。对象定义的 XML 数据可以保存在任何 IResource 接口所支持的位置(可以参考 6.1, 简介）。如果要用独立的 XML 文件保存对象定义，在.NET 应用程序配置中，应该这样配置<context>节点：

```
<spring>

  <context>
    <resource uri="file://objects.xml"/>
  </context>

</spring>
```

VS.NET 2005 的 XML 编辑器能利用 xsi:schemaLocation 属性来关联一个 schema 文件，用它来为正在编辑的 XML 文档进行代码提示。VS.NET 2002/2003 不能识别该属性。如果用下面代码中的方法引用 Spring.NET 的 XML schema，在 VS.NET 2005 中编辑配置文件时就能看到代码提示，也可对文档进行验证。如果想在 VS.NET 2002/2003 中使用这些功能，则需要在 VS.NET 中注册这个 schema，或者将它包含到当前的项目当中。

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net
http://www.springframework.net/xsd/spring-objects.xsd">
  <object id="..." type="...">
  ...
  </object>
  <object id="..." type="...">
  ...
  </object>
  ...
</objects>
```

将 schema 文件安装进 VS.NET 会更为方便，对于 VS.NET 2005 来说也是如此，因为这可以避免 Xml 文件过于冗长，也不用每创建一个项目就需要复制 schema 的路径。要将 schema 文件安装到 VS.NET 中，可以将它们复制到以下目录：

VS2003：C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Packages\schemas\xml

或

VS2002：C:\Program Files\Microsoft Visual Studio .NET\Common7\Packages\schemas\xml

对 VS.NET 2005 来说，则是：

C:\Program Files\Microsoft Visual Studio 8\Xml\Schemas

还可以利用 Spring.NET 的 doc/schema 目录下的 NAnt 脚本文件来安装这些 Schema。执行了这个脚本之后，spring-object.xsd 文件就会被复制到正确的位置。

安装了 schema 之后，在 VS.NET 的编辑器中，只要 objects 节点包含 xmlns="http://www.springframework.net"属性，VS 就可以利用这个 schema 来进行代码提示了：

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net">
  <object id="..." type="...">
  ...
  </object>
  <object id="..." type="...">
  ...
  </object>
  ...
</objects>
```

另外，也可以在配置文件的属性中通过 Schemas 来选择 xsd 文件。

在 4.7,与 IObjectFactory 交互一节中曾经提过，可以在.NET 的应用程序配置文件中存放对象定义，如下：

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
```

```
  </configSections>

  <spring>

    <context>
      <resource uri="config://spring/objects"/>
    </context>

    <objects xmlns="http://www.springframework.net">
        ...
    </objects>

  </spring>

</configuration>
```

此时，VS.NET 2002/2003 也能进行代码提示，但是因为没有为整个 App.config 文档提供 schema，所以无法执行全面验证。要进行全面验证需要安装.NET Configuration File schema，并将<spring>和<context>节点也加入到该 schema 中。

验证 schema 是 VS 2005 的新功能，在编辑时，可以在 VS 2005 的错误列表窗口看到文档中的错误。

关于如何存放对象定义的配置数据，好的习惯是保留 App.config 的原型，而在其它兼容 IResource 的位置或文件中存放 Spring.NET 的配置，对于比较重要的项目，可以存放在程序集的内嵌资源中。

24.2. XML Schema 的版本

为了支持泛型，Spring.NET 1.0.2 的 schema 已经与 1.0.1 有所不同。1.0.1 版的 schema 位于 http://www.springframework.net/xsd/1.0.1/。最新版会始终放在 http://www.springframework.net/xsd/。

24.3. 集成 API 文档

在安装过程中，Spring.NET 会将文档注册到 Visual Studio。文档有两个版本，分别针对 VS.NET 2002/2003 和 VS.NET 2005。这两个版本只在格式上有差别，VS.NET 2005 版的文档使用的是新格式，比较酷。哈！

第二十五章. IoC 快速入门

25.1. 简介

本章通过一个综合性的例子向读者讲解 Spring.NET IoC 容器的用法。

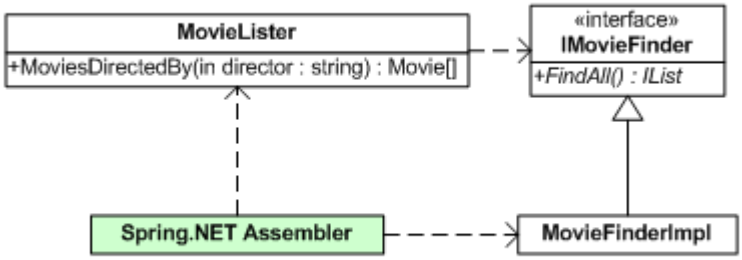
25.2. Movie Finder

本例直接来自 Martin Fowler 的论文，请参考：Inversion of Control Containers and the Dependency Injection pattern。以这篇论文为基础是因为其知名度相当高，很多人在第一次接触 IoC 的时候都曾读过它（用 Google 搜索“IoC”时，这篇文章排在搜索结果的前五位）。

Fowler 用一个电影搜索工具来介绍 IoC 和依赖注入（DI）。文中讲到了一个 MovieLister 对象如何（通过 DI）获取 IMovieFinder 接口类型的对象。

IMovieFinder 接口会返回一个包含所有电影的列表，然后 MovieLister 类再将由某个导演执导的影片从其中过滤出来，并返回一个 Movie 数组。通过本例，读者可以了解 IoC 容器如何为 MovieLister 对象注入一个 IMovieFinder 对象。

MovieFinder 类的源代码在 examples/Spring/Spring.Examples.MovieFinder 目录。



25.2.1.开始建立 MovieFinder 应用程序

MovieFinder 应用程序的起始类是 MovieApp——一个包含入口方法的普通.NET 类型...

```
using System;
namespace Spring.Examples.MovieFinder
{
  public class MovieApp
  {
```

```
    public static void Main ()
    {
    }
  }
}
```

本例试图让读者了解 Spring.NET 容器的用法，所以我们要从 IoC 容器（IApplicatoinContext）中获取一个 MovieFinder 类的实例。创建容器的方法有很多种，在本例中，我们让 Spring.NET 从标准的.NET 配置文件中初始化 IApplicationContext...

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="spring">
            <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core"/>
            <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
        </sectionGroup>
    </configSections>
    <spring>
        <context>
            <resource uri="config://spring/objects"/>
        </context>
        <objects xmlns="http://www.springframework.net">
            <description>An  example that demonstrates simple IoC
features.</description>
        </objects>
    </spring>
</configuration>
```

在配置文件中，通过<objects/>节点的<object/>子节点来配置程序中用到的对象。

现在，我们向 MovieApp 类的 Main 方法添加一些代码...

```
using System;
using Spring.Context;
...
    public static void Main ()
    {
      IApplicationContext ctx = ContextRegistry.GetContext();
    }
...
```

在代码中，我们引用了 Spring.Context 命名空间，以便用 IApplicationContext 接口访问 IoC 容器，其中...

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

...返回的就是一个已经根据<objects/>节点的内容配置好的容器对象。

25.2.2.第一个对象定义

到目前为止我们还没有在配置文件中定义任何对象，现在我们来添加第一个对象定义。下面的 XML 代码就是我们要用到的 MovieLister 对象定义...

```
<objects xmlns="http://www.springframework.net">
    <object name="MyMovieLister"
      type="Spring.Examples.MovieFinder.MovieLister,
Spring.Examples.MovieFinder">
    </object>
</objects>
```

注意，对象定义中的 type 属性值必须是包含程序集名称在内的类型全名。同时，我们给这个对象定义分配了一个惟一的 id：MyMovieLister。在代码中，可以通过这个 id 从 IApplicationContext 中请求所定义的对象...

```
...
    public static void Main ()
    {
      IApplicationContext ctx = ContextRegistry.GetContext();
      MovieLister lister = (MovieLister) ctx.GetObject
("MyMovieLister");
    }
...
```

lister 对象需要同一个 IMovieFinder 对象协作，但我们还没有将这个 IMovieFinder 对象注入给它，所以调用 lister 的 MoviesDirectedBy 方法只能抛出 NullReferenceException 异常。下面是一个 IMovieFinder 实现类的对象定义，该对象会被注入给 lister...

```
<objects xmlns="http://www.springframework.net">
    <object name="MyMovieFinder"
        type="Spring.Examples.MovieFinder.SimpleMovieFinder,
Spring.Examples.MovieFinder"/>
    </object>
</objects>
```

25.2.3.属性注入（设值方法注入）

我们希望把 id 为 MyMoiveFinder 的 IMovieFinder 对象注入给 id 为 MyMovieLister 的 MovieLister 对象，这一工作可通过属性注入来完成，请看下面的 XML 代码...

```
<objects xmlns="http://www.springframework.net">
    <object name="MyMovieLister"
      type="Spring.Examples.MovieFinder.MovieLister,
Spring.Examples.MovieFinder">
        <!-- using setter injection... -->
        <property name="movieFinder" ref="MyMovieFinder"/>
    </object>
    <object name="MyMovieFinder"
        type="Spring.Examples.MovieFinder.SimpleMovieFinder,
Spring.Examples.MovieFinder"/>
    </object>
</objects>
```

当 IoC 容器初始化 MyMovieLister 对象时，就会将 MyMovieFinder 对象注入给它的 MovieFinder 属性。MyMovieLister 对象创建后，应用程序就可以用它来列举某个导演执导的影片。

```
...
    public static void Main ()
    {
      IApplicationContext ctx = ContextRegistry.GetContext();
      MovieLister lister = (MovieLister) ctx.GetObject
("MyMovieLister");
      Movie[] movies = lister.MoviesDirectedBy("Roberto Benigni");
      Console.WriteLine ("\nSearching for movie...\n");
      foreach (Movie movie in movies)
      {
          Console.WriteLine (
              string.Format ("Movie Title = '{0}', Director = '{1}'.",
              movie.Title, movie.Director));
      }
      Console.WriteLine ("\nMovieApp Done.\n\n");
    }
...
```

25.2.4.构造器参数注入

现在，我们在配置文件中为 IMovieFinder 接口的另一个实现类创建对象定义...

```
...
  <object name="AnotherMovieFinder"
      type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder,
Spring.Examples.MovieFinder">
  </object>
...
```

ColonDelimitedMovieFinder 可以将文本文件作为影片数据的来源，文本数据之间用冒号分隔。该类只定义了一个构造器，接受一个 System.IO.FileInfo 类型的参数。假如我们只用上面的对象定义，那么当代码从 IApplicationContext 获取该对象时...

```
IMovieFinder finder = (IMovieFinder) ctx.GetObject
("AnotherMovieFinder");
```

会导致致命异常：Spring.Objects.Factory.ObjectCreationException，因为 Spring.Examples.MovieFinder.ColonDelimitedMovieFinder 类没有无参的默认构造器。（按：这个说法不准确，在 IApplicationContext 中，除非显式指定对象定义的 singleton 属性为 false，或 lazy-init 为 true，否则容器会尝试预先创建对象，所以这个错误不会等到请求对象时才发生。）如果我们想让容器创建它的对象，就必须为其提供必要的构造器参数...

```
...
  <object name="AnotherMovieFinder"
      type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder,
Spring.Examples.MovieFinder">
      <constructor-arg index="0" value="movies.txt"/>
  </object>
...
```

很显然，<constructor-arg/>节点就是用来为构造器提供参数的。IoC 容器使用某个内置的类型转换器将字符串“Movies.txt”转换为 System.IO.FileInfo（关于 Spring.NET 自动类型转换的详细内容，请参见：5.3, 类型转换）。

现在，在配置文件中，我们已经为 IMovieFinder 的两个实现类分别创建了对象定义；如果需要，我们可以更改注入给 MyMovieLister 的 IMovieFinder 实例，如下所示...

```
...
  <object name="MyMovieLister"
```

```
    type="Spring.Examples.MovieFinder.MovieLister,
Spring.Examples.MovieFinder">
      <!-- lets use the colon delimited implementation instead -->
      <property name="movieFinder" ref="AnotherMovieFinder"/>
  </object>
  <object name="MyMovieFinder"
      type="Spring.Examples.MovieFinder.SimpleMovieFinder,
Spring.Examples.MovieFinder"/>
  </object>
  <object name="AnotherMovieFinder"
      type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder,
Spring.Examples.MovieFinder">
      <constructor-arg index="0" value="movies.txt"/>
  </object>
...
```

注意，应用程序不需要重新编译，只要修改配置文件并重新启动，新的 AnotherMovieFinder 即可生效。

25.2.5.总结

这个程序很简单，没有做太多的事情。但它的确讲清楚了如何用 XML 配置来装配对象这一基本功能。这些简单的功能可以应付 80%的需求。诸如工厂方法、惰性初始化以及其它较少用到的特性，Spring.NET 也都支持的非常好（所有的配置方法在第四章,对象、对象工厂和应用程序上下文中都有详细讨论）。

25.2.6.日志

一般来说，第一次学习 Spring.NET 的过程往往也是对 log4net 的入门过程。本节对 log4net 做一个概括性的介绍，以便读者能够快速的起步。关于 log4net，最权威的信息自然来自 log4net 的网站。其它较好的在线资源有 Using log4net (OnDotNet article) 和 Quick and Dirty Guide to Configuring Log4Net For Web Applications。Spring.NET 使用的 log4net 为 1.2.9 版，但很多参考资源都是针对 1.2.0 版的。这两个版本间还是有一定的区别，所以请留意 log4net 网站上的权威信息。另外请注意，Spring.NET 正在开发“通用”的日志类库，使 Spring.NET 不必紧依赖于 log4net，也能够使用其它的日志 API，比如 NLog 和微软企业库的 logging 应用程序块。

log4net 的使用模式是：在 App/Web.config 或独立的 xml 文件中配置 logger、在主应用程序中初始化 log4net、在代码中声明 logger，然后就是 log、log、log（来，咱们一起唱、、、）。在本例中我们用 App.config 来配置 logger。所以，要在 sectionGroup 节点中添加下面这个节点处理器：

```
<section name="log4net"
type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
```

对应的配置节点与如下代码类似：

```
<log4net>
  <appender name="ConsoleAppender"
type="log4net.Appender.ConsoleAppender">
     <layout type="log4net.Layout.PatternLayout">
        <conversionPattern value="%date [%thread] %-5level %logger
- %message%newline" />
     </layout>
  </appender>

  <!-- Set default logging level to DEBUG -->
  <root>
     <level value="DEBUG" />
     <appender-ref ref="ConsoleAppender" />
  </root>

  <!-- Set logging for Spring to INFO.  Logger names in Spring correspond
to the namespace -->
  <logger name="Spring">
     <level value="INFO" />
  </logger>
</log4net>
```

<apperder/>节点定义了日志的输出接收器（output sink），在本例中输出接收器是控制台。log4net 支持多种类型的输出接收器，包括文件、数据库等等，可以参考 log4net 的示例配置。<layout/>节点中的 PatternLayout 定义了日志记录的准确格式。一般来说包括日期、线程信息、日志级别、logger 的名称，最后是要记录的信息。关于如何自定义日志格式，可以参考 PatternLayout 文档。

logger 的名称是由开发人员在 C#代码中声明 logger 时确定的。在本例中，我们用 MovieApp 的类型全名作为 logger 的名称：

```
private static readonly ILog LOG = LogManager.GetLogger(typeof
(MovieApp));
```

另外，为同一逻辑组件或子系统（比如数据访问层）中所有类型定义统一的 logger，用起来会比较方便。在定义日志格式（Layout）时有个小小的技巧，就是使其中的 logger 名称只显示类型全名的最后两部分，以免消息内容被挤得太靠右（就看不到了），因为日志中的其它信息都是放在消息内容前面的。可以用模式字符串"%logger{2}"来只显示 logger 名称的后两部分。

用下面的代码来初始化日志系统：

```
XmlConfigurator.Configure();
```

注意，如果您正在用 1.2.0 版的 log4net，应该调用 DOMConfigurator.Configure() 方法。

<logger>节点将 logger 的名称与日志级别以及 appender 关联起来。此处的灵活性很高，可以混用或匹配不同的 logger 名称、日志级别及 appender。在本例中，我们把根 logger（用专门的 root 节点定义）定义在调试级别，并使用控制台作为接收器。然后，可以用不同的设置定义一些专用的 logger。在本例中，名称以“Spring”开头的 logger 定义在 INFO 级别，同样输出到控制台。如果将此处 logger 的级别由 INFO 改为 DEBUG，就可以看到 IoC 容器在创建时的详细日志信息。由于定义在代码中的 logger 名字包含“Spring”，所以这个 logger 节点可以控制 MainApp 的日志输出级别。现在，我们就可以使用 logger 记录日志了，例如：

```
LOG.Info("Searching for movie...");
```

记录异常也是一个很常见的操作，可以通过 ERROR 级别进行：

```
try {
 //do work
{
catch (Exception e)
{
 LOG.Error("Movie Finder is broken.", e);
}
```

25.3. 应用程序上下文和 IMessageSource 接口

25.3.1.简介

在 Spring.NET 的示例程序中，有一个 Spring.Examples.AppContext 项目，它的内容包括：利用应用程序上下文进行文本本地化、从 ResourceSet 中返回数据，以及如何将资源中内嵌的属性值赋值给某个对象。程序利用一个 windows 窗体来显示这些值。

应用程序上下文的配置中包含一个名为 messageSource 的对象定义，其类型为 Spring.Context.Support.ResourceSetMessageSource， 该类实现了 IMessageSource 接口，这个接口定义了很多方法用于获取本地化资源（如文本和图像），请参见 4.13.2,使用 IMessageSource 接口。在应用程序上下文被创

建时，它就会去查找名为“messageSource”的 IMessageSource 对象，来为自己提供 IMessageSource 的功能。

ResourceSetMessageSource 类的 ResourceManagers 属性用于定义一系列与语言文化相关的资源集会，该属性实际是 ResourceManager 对象的列表。
ResourceManager 对象可以通过两种方式来创建。一是用包含资源名和程序集名的、以逗号分隔的字符串创建。二是用辅助的工厂对象 ResourceManagerFactoryObject 根据资源的 base name 和程序集名来创建。如果需要单独引用 ResourceManager 对象，那么第二种方法比较合适。在示例程序中，用这种方式定义了一个内嵌的资源文件 MyResource.resx 和一个西班牙语的资源文件 MyResources.ex.resx。XML 配置如下：

```
...
 <object name="messageSource"
type="Spring.Context.Support.ResourceSetMessageSource, Spring.Core">
 <property name="ResourceManagers">
 <list>
 <value>Spring.Examples.AppContext.Images,
Spring.Examples.AppContext</value>
 <ref object="myResourceManager"/>
 </list>
 </property>
 </object>

 <object name="myResourceManager"
   type="Spring.Objects.Factory.Config.ResourceManagerFactoryObject,
Spring.Core">
 <property name="BaseName">
 <value>Spring.Examples.AppContext.MyResource</value>
 </property>
 <property name="AssemblyName">
 <value>Spring.Examples.AppContext</value>
 </property>
 </object>
...
```

应用程序首先创建应用程序上下文，然后根据资源键值来获取不同的资源。在此我们用 Keys 类的静态字段来表示资源的键值。资源文件 Image.resx 包含图像数据，其键值为“bubblechamber”（即 Keys.BUBBLECHAMBER 字段的值）。代码：

```
Image image = ctx.GetResourceObject(Keys.BUBBLECHAMBER) as Image;
```

用于将此图像数据从容器中读取出来。MyResource.resx 包含的是文本资源，其中键值“HelloMessage”（Keys.HELLO_MESSAGE）对应的资源值为“Hello {0} {1}”，可以用作格式字符串，下面的代码：

```
string msg = ctx.GetMessage(Keys.HELLO_MESSAGE,
 CultureInfo.CurrentCulture,
 "Mr.", "Anderson");
```

读取该资源值，并用参数中的字符串替换其中的占位符，最终输出为“Hello Mr. Anderson”。这里使用的区域信息为 CurrentCulture，对应的资源文件为 MyResource.resx。如果要用西班牙语：

```
CultureInfo spanishCultureInfo = new CultureInfo("es");
string esMsg = ctx.GetMessage(Keys.HELLO_MESSAGE,
 spanishCultureInfo,
 "Mr.", "Anderson");
```

就会使用资源文件 MyResource.ex.resx。Spring.NET 会把本地化资源文件的选择交给.NET 的 ResourceManager 代理。西班牙语资源文件与英语版的差别是其中 HelloMessage 资源值为“Hola {0} {1}”，所以输出结果为“Hola Mr. Anderson”。

当然，西班牙语没有“Mr.”这一称谓。我们也可以把称谓本身抽象为一个单独的资源项，其键值为 FemaleGreeting（Keys.FEMALE_GREETING）。随后，利用 DefaultMessageResolvable 类，消息参数{0}的替换值就可以自动依不同的区域信息来确定。请看以下代码：

```
string[] codes = {Keys.FEMALE_GREETING};
DefaultMessageResolvable dmr = new DefaultMessageResolvable(codes,
null);

msg = ctx.GetMessage(Keys.HELLO_MESSAGE,
 CultureInfo.CurrentCulture,
 dmr, "Anderson");
```

变量 msg 的值为"Hello Mrs. Adnderson"，因为在资源文件 MyResource.res 中，键 FemaleGreeting 对应的值是“Mrs.”。同样的，以下代码：

```
esMsg = ctx.GetMessage(Keys.HELLO_MESSAGE,
 spanishCultureInfo,
 dmr, "Anderson");
```

变量 esMsg 的值为“Hola Senora Anderson”，因为在资源文件 MyResource.ex.resx 中，键 FemaleGreeting 对应的值是“Senora”。

本地化也可应用于其它非字符串类型的对象。.NET 1.1 的工具类 ComponentResourceManager 就可以很高效的将资源值赋值给对象属性（VS 2005 在生成 WinForm 应用程序时大量的使用了这个类）。本例有一个简单的 Person 类，包含一个整型属性 Age 和一个字符串属性 Name。在资源文件 Person.resx

中，键值以 person.<PropertyName>格式命名，即：person.Name 和 person.Age。以下代码将这两个资源的值赋值给 person 对象的相应属性：

```
Person p = new Person();
ctx.ApplyResources(p, "person", CultureInfo.CurrentUICulture);
```

当然，也可以用 Spring.NET 的容器来为对象属性赋值，但容器本身不会去考虑简单属性的本地化问题。不过，将这两种方式结合起来使用是很方便的：可以用容器配置对象的依赖项，然后在 AfterPropertiesSet 方法（参见 IInitializingObject 接口）中用上面的方法设置与语言文化相关的属性值。

25.4. 应用程序上下文和 IEventRegistry 接口

25.4.1.简介

我们来通过 Spring.Examples.EventRegistry 来说明应用程序上下文如何用松耦合的方式装配.NET 事件。

松耦合的事件模型一般与面向消息的中间件(MOM,Message Oriented Middleware)有关，通常是由一个后台进程来担当其它进程的消息中介。通过向消息中介发送消息，进程间就可以实现间接通信，通信的发起者称为发布者，接收消息的进程则称为订阅者。利用各种中间件的 API，进程可以把自己注册为发布者或订阅者。发布者和订阅者之间的通信是松耦合的，因为它们不需要直接去引用对方，消息中介可以作为它们之间的媒介。IEventRegistry 接口就为.NET 事件提供了类似于消息中介的功能。在此，发布者是指触发.NET 事件的类，而订阅者是指注册并接收这些事件的类，在发布者和订阅者之间传递的信息保存在 System.EventArgs 类型的实例中。IEventRegistry 接口的具体实现类决定了事件通知的类型以及订阅者和发布者之间的耦合方式。

IApplicationContext 接口扩展了 IEventRegistry 接口，并且 IApplicationContext 的实现类将事件注册的功能交由 Spring.Objects.Events.Support.EventRegistry 来代理。IEventRegistry 是一个简单的接口，包括一个发布方法和两个订阅方法。可以参考 4.13.4,松耦合事件模型。Spring.Objects.Events.Support.EventRegistry 的目的本质上只是将发布者和订阅者之间的事件装配过程解耦，在事件装配完成之后，发布者仍是通过标准的.NET 事件机制和订阅者直接耦合的。其它还有一些 IEventRegistry 接口的实现类，可以利用事件注册中心（event registry）来订阅事件，再由事件注册中心负责通知订阅者，从而进一步削弱发布者和订阅者的耦合程度。

在本例中，MyClientEventArgs 是 System.EventArgs 的子类，其中定义了一个字符串属性 EventMessage。MyEventPublisher 类则定义了一个公有的事件，事件委托为：

```
void SimpleClientEvent( object sender, MyClientEventArgs args )
```

并由 void ClientMethodThatTriggersEvent1()方法负责触发该事件。订阅者类 MyEventSubscriber 包括一个事件处理方法 HanldeClientEvents 和一个布尔类型的属性，如果事件处理方法被调用，就将该属性值设为 true。

本例将这两个发布者和订阅者对象都定义在了应用程序上下文中，但并非只有定义在容器中的对象才能使用事件注册功能。应用程序类 EventRegistryApp 类创建应用程序上下文，并从中请求 MyEventPublisher 的对象。然后调用：

```
ctx.PublishEvents( publisher )
```

来将发布者注册到事件中心。事件中心会保留对发布者的引用，稍后会为它注册能够处理其事件的所有订阅者（订阅者需要包含与发布者事件签名匹配的处理方法）。然后，程序创建了两个订阅者对象。其中一个用 ctx.Subscribe(subscriber, typeof(MyEventPublisher))方法注册了发布者的事件，在方法中指定发布者的类型是要限制订阅者只响应 MyEventPublisher 类型的相应事件，实际上是一种过滤订阅者的机制。

随后，发布者触发了一个标准的.NET 事件，订阅者的事件处理方法随之被调用。订阅者在控制台中打印 一行消息，并设置一个状态变量，已标识自己已对事件做出了响应。最后，程序将两个订阅者的状态变量的值打印处理，可以看到，只有一个订阅者的事件处理方法被调用了（也就是注册到事件中心的那个）。

25.5. 对象池示例

本节准备以 QueuedExceutor 对象池为基础建立一个 executor 示例，目的是介绍 Spring.NET 低层次/高质量的可重用线程及对象池功能。executor 具备并发执行的能力（在本例中，用于 grep，类似于文件扫描）。注意本例使用了 Spring.Threading 命名空间下的类，该命名空间计划在 Spring.NET 1.1 中发布，所以 1.0.0 版本中不包含本例的代码。如果需要本例的代码，可以从 CVS（*使用说明*）或 Spring.NET 网站的下载页面下载。

以下是有关 QueuedExecutor 的一些信息，会对理解（或者推翻）该例的实现方式有所帮助。读者可以利用这些知识来建立自己的对象池。

QueueExector 是一个 executor 对象，会通过工作线程顺序执行多个 IRunnable 对象。在用 QueuedExecutor 执行操作时，所有请求都会被排队；稍后的某个时刻，这些请求就会被工作线程执行；如果遇到错误，工作线程就会终止，但 executor 会根据需要重新创建工作线程。

最后，executor 可以通过几种不同的方法来关闭（请参考 Spring.NET 的 SDK 文档）。executor 很简单，但功能非常强大。

示例项目 Spring.Examples.Pool 利用 n 个 Spring.Threading.QueuedExecutor 实例实现了一个支持池的 executor：虽然 Spring.Threading 命名空间下已经有一个 PooledExecutor 类，但在这儿，我们的目的是使用 QueuedExecutor 对象池。

我们会利用这个 executor 来实现一个执行并行递归操作、类似 grep 的控制台应用程序。（按：grep 是 UNIX 中执行文件内字符串查找的工具。）

25.5.1.实现 Spring.Pool.IPoolableObjectFactory

要想使用 SimplePool 类，必须先实现 IPoolableObjectFactory 接口。实现了该接口的类用于创建要被放入对象池的对象。SimplePool 会在对象池被创建、对象被借出和归还、以及对象池被销毁时调用 IPoolableOjbectFactory 接口的生命周期方法（即 MakeObject、ActivateObject、ValidateObject、PassivateObject 和 DestroyObject 这 5 个方法）。

前面提到过，我们打算实现 QueuedExecutor 的对象池。所以，我们需要一个能创建 QueuedExecutor 对象的 IPoolableObjectFactory 实现类，下面是该类的声明：

```
public class QueuedExecutorPoolableFactory : IPoolableObjectFactory
{
```

IPoolableObjectFactory 实现类的第一个任务是创建对象，即实现 MakeObject()方法：

```
object IPoolableObjectFactory.MakeObject()
{
 // to actually make this work as a pooled executor
 // use a bounded queue of capacity 1.
 // If we don't do this one of the queued executors
 // will accept all the queued IRunnables as, by default
 // its queue is unbounded, and the PooledExecutor
 // will happen to always run only one thread ...
 return new QueuedExecutor(new BoundedBuffer(1));
}
```

同时也要能够销毁对象，即实现 DestroyObject(object o)方法：

```
void IPoolableObjectFactory.DestroyObject(object o)
{
 // ah, self documenting code:
 // Here you can see that we decided to let the
 // executor process all the currently queued tasks.
 QueuedExecutor executor = o as QueuedExecutor;
```

```
 executor.ShutdownAfterProcessingCurrentlyQueuedTasks();
}
```

当客户从池中请求对象时，该对象可能需要被激活。所以我们也可以实现下面的方法：

```
void IPoolableObjectFactory.ActivateObject(object o)
{
 QueuedExecutor executor = o as QueuedExecutor;
 executor.Restart();
}
```

如果 QueuedExecutor 对象可以根据需要自行重启，那么也可以让这个方法为空，不去实现它。

激活之后，在将对象池中的对象返回给客户端代码之前，需要验证（对象如果无效就会被丢弃；这可能会导致一个空的、不可用的池 [8]）。在这里，我们检查工作线程是否存在：

```
bool IPoolableObjectFactory.ValidateObject(object o)
{
 QueuedExecutor executor = o as QueuedExecutor;
 return executor.Thread != null;
}
```

钝化（passivation）是激活的反义词，是指当对象被归还到池的时候可能要做的处理。在本例中，我们把 PassivateObject 方法留空：

```
void IPoolableObjectFactory.PassivateObject(object o)
{
}
```

至此，创建一个 SimplePool 对象就创建了一个对象池。

```
pool = new SimplePool(new QueuedExecutorPoolableFactory(), size);
```

25.5.2. 使用池中的对象

在 C#的代码中，利用 using 关键字是很重要的，所以我们实现了一个很简单的辅助类（PooledObjectHolder），可以让我们利用 using 关键字使用对象池中的对象：

```
using (PooledObjectHolder holder = PooledObjectHolder.UseFrom(pool))
{
```

```
 QueuedExecutor executor = (QueuedExecutor) holder.Pooled;
 executor.Execute(runnable);
}
```

这样，我们就无需自己费心去从池中获取和归还对象了。

下面是 PooledObjectHolder 的实现：

```
public class PooledObjectHolder : IDisposable
{
 IObjectPool pool;
 object pooled;

 /// <summary>
 /// Builds a new <see cref="PooledObjectHolder"/>
 /// trying to borrow an object form it
 /// </summary>
 /// <param name="pool"></param>
 private PooledObjectHolder(IObjectPool pool)
 {
 this.pool = pool;
 this.pooled = pool.BorrowObject();
 }

 /// <summary>
 /// Allow to access the borrowed pooled object
 /// </summary>
 public object Pooled
 {
 get
 {
 return pooled;
 }
 }

 /// <summary>
 /// Returns the borrowed object to the pool
 /// </summary>
 public void Dispose()
 {
 pool.ReturnObject(pooled);
 }

 /// <summary>
 /// Creates a new <see cref="PooledObjectHolder"/> for the
```

```
 /// given pool.
 /// </summary>
 public static PooledObjectHolder UseFrom(IObjectPool pool)
 {
 return new PooledObjectHolder(pool);
 }
}
```

别忘了要在使用之后将池中所有的对象销毁。在 PooledQueuedExecutor 类中，用类似下面的代码：

```
public void Stop ()
{
 // waits for all the grep-task to have been queued ...
 foreach (ISync sync in syncs)
 {
 sync.Acquire();
 }
 pool.Close();
}
```

25.5.3.利用 executor 执行并行的 grep

刚创建的 executor 在用法上是非常简单的，但如果要充分利用对象池的功能，还需要一点技巧：

```
private PooledQueuedExecutor executor;

public ParallelGrep(int size)
{
 executor = new PooledQueuedExecutor(size);
}

public void Recurse(string startPath, string filePattern, string
regexPattern)
{
 foreach (string file in Directory.GetFiles(startPath, filePattern))
 {
 executor.Execute(new Grep(file, regexPattern));
 }
 foreach (string directory in Directory.GetDirectories(startPath))
 {
 Recurse(directory, filePattern, regexPattern);
 }
```

```
}

public void Stop()
{
 executor.Stop();
}
public static void Main(string[] args)
{
 if (args.Length < 3)
 {
 Console.Out.WriteLine("usage: {0} regex directory file-pattern
[pool-size]",
    Assembly.GetEntryAssembly().CodeBase);
 Environment.Exit(1);
 }

 string regexPattern = args[0];
 string startPath = args[1];
 string filePattern = args[2];
 int size = 10;
 try
 {
 size = Int32.Parse(args[3]);
 }
 catch
 {
 }
 Console.Out.WriteLine ("pool size {0}", size);

 ParallelGrep grep = new ParallelGrep(size);
 grep.Recurse(startPath, filePattern, regexPattern);
 grep.Stop();
}
```

25.6. AOP

请参考第二十六章, AOP 指南。

---

[8] 您可能认为我们可以实现一个更为智能的对象池。但是如果一个旧的对象池变的不可用，再创建一个新的是很简单的。您也许不喜欢这种方式，但它确实利用了简单原则和对象的不变性。

第二十六章. AOP 指南

26.1. 简介

本章是 Spring.NET AOP 框架的简单指南，向读者介绍如何使用 Spring.NET 进行面向方面的编程（AOP）。

本指南不要求读者有使用 Spring.NET AOP 框架的经验。但是，确实需要读者熟悉 AOP 的基本概念。如果您已经读过（或至少浏览过）本文档 AOP 的相关章节，那么可能会对理解本章的内容有所帮助，如果您已经了解：a）什么是 AOP，b）AOP 能够解决哪些问题，c）AOP 中通知、切入点和连接点的概念，本章就不需要再花时间来解释这些名词了。另外，如果您喜欢从例子中学习和实验，那么完全可以先从本章看起，然后在需要时参考前面的内容（请看 12.1.1, AOP 基本概念）。

*本章的例子**刻意**做得很简单。主要是让读者能尽快看到 Spring.NET 的 AOP 框架能为我们带来什么好处。为了学习 AOP 而强迫读者必须去理解某个对象模型，其实对掌握 AOP 框架没有一点儿帮助。读者可以在读完本章之后自己复习 AOP 的相关概念，并把它们应用到自己的代码中。本章包括几个菜谱式的 AOP 范例，向读者介绍如何在实际开发中应用 AOP 框架；另外，Spring.NET 附带的参考程序也大量使用了 AOP 框架，特别是在领域对象模型中（请参考第二十九章, SpringAir - 参考程序）。*

26.2. 基础知识

本节介绍一些 AOP 的基础知识，读者将了解如何定义和应用简单的**通知**。

26.2.1. 应用通知

我们来看一个非常简单的例子。通过下面的代码，我们学习如何应用一个简单的通知，该通知用于将目标方法的详细调用信息打印到控制台。老实说这个例子不是很吸引人，简直毫无实用价值。但是通过本例读者可以学到如何用自定义的通知完成更为有用的工作（比如事务管理，验证，安全性，线程安全等）。

在开始看 AOP 的相关代码之前，我们先看一下作为通知目标的领域对象（在 Spring.NET AOP 术语中，这样的对象称为目标对象）。

```
public interface ICommand
{
    object Execute(object context);
}
```

```
    public class ServiceCommand : ICommand
    {
        public object Execute(object context)
        {
            Console.Out.WriteLine("Service implementation : [{0}]",
context);
            return null;
        }
    }
```

下面这个通知将要应用到 ServiceCommand 类的 Excute(object context)方法上。可以看出，这是一个环绕通知(参见 12.3.2,通知类型).

```
    public class ConsoleLoggingAroundAdvice : IMethodInterceptor
    {
        public object Invoke(IMethodInvocation invocation)
        {
            Console.Out.WriteLine("Advice executing; calling the

advised method...");

            object returnValue = invocation.Proceed();
            Console.Out.WriteLine("Advice executed; advised method

returned " + returnValue);

            return returnValue;
        }
    }
```

在控制台打印一行信息，表示通知正在执行。

目标方法被调用。

返回值被捕获，赋值给 returnValue 变量。

returnValue 的值被打印出来。

返回 returnValue 的值。

到目前为止,我们有三个类型:ICommand 接口,该接口的实现类(ServiceCommand)和一个（没什么用的）通知（封装在 ConsoleLoggingAroundAdvice 类中）。剩

下的工作就是要将 ConsoleLoggingAroundAdvice 通知应用到 ServiceCommand 类的 Execute 方法上。我们看看如何通过编程方法做到这一点：

```
ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvice(new ConsoleLoggingAroundAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute("This is the argument");
```

上面代码的输出如下：

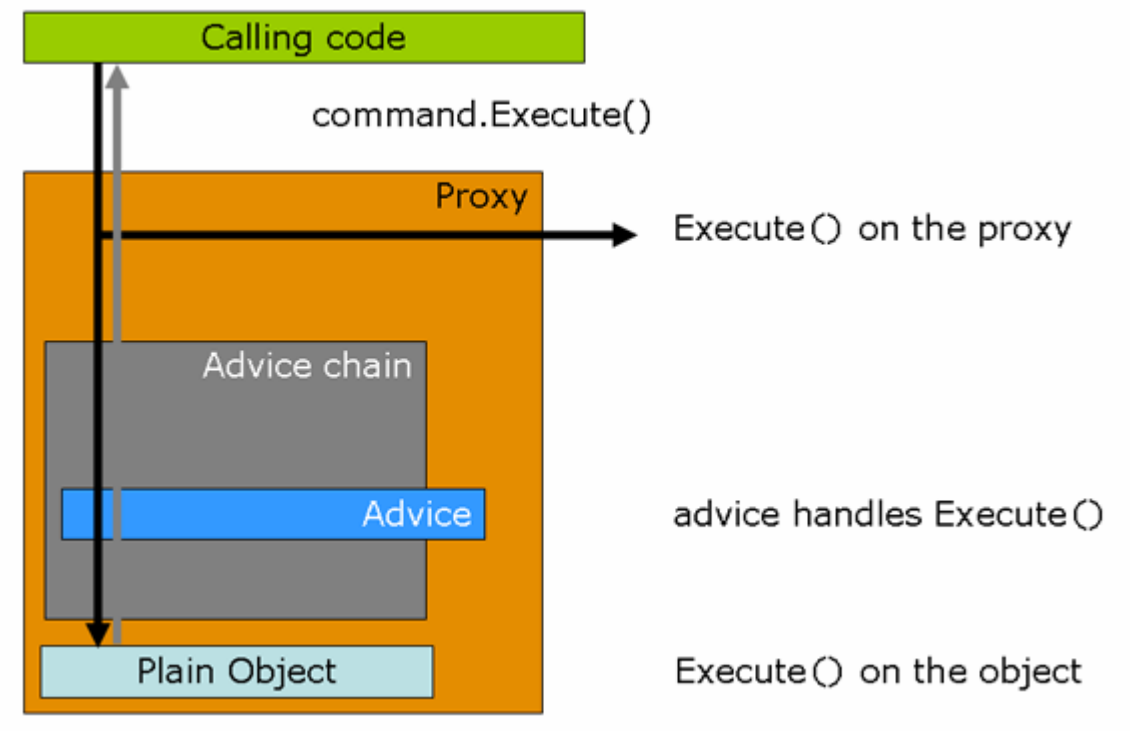
```
Advice executing; calling the advised method...
Service implementation : [This is the argument]
Advice executed; advised method returned
```

输出结果显示：在目标方法调用的前后，ConsoleLogginAroundAdvice 类中的 Console.WriteLine 语句也被调用了。

那么这又是怎么回事呢？代码中使用的 ProxyFactory 类可以帮助我们了解其中的奥妙。ProxyFactory 的构造器以目标对象作为参数（在本例中是 ServiceCommand 类的一个实例）。然后，通过 ProxyFactory 的 AddAdvice()方法向这个 ServiceCommand 实例添加了一个通知（一个 ConsoleLoggingAroundAdvice 通知）。（按：在 12.7.管理目标对象中讲到，若向代理工厂中直接添加拦截器或其它通知类型，Spring.NET 会将其连同一个始终返回 true 的 IPointcut 一起包装进一个 Advisor 中返回。所以，这个通知会被应用到被通知类型的所有方法上去。）随后，通过调用 ProxyFactory 实例的 GetProxy()方法获得一个代理——也就是目标对象（ServiceCommand 的实例）的 AOP 代理。当调用代理的 Execute 方法时。通知就被“应用了”（或者说被执行了），如同上面的输出一样。

<!--[endif]-->

下图是 Spring.NET AOP 代理的执行流程：



这里要注意的是，通过 ProxyFacotry.GetProxy()方法返回的对象同样实现了目标对象实现的（所有）接口，所以它可以转型为目标对象所实现的任一接口。这一点非常重要，目前 Spring.NET 的 AOP 框架要求目标对象必须实现一个以上接口。也就是说，要想让 AOP 框架支持某个类，该类必须实现至少一个接口。实际上这个限制并没有乍听起来那么麻烦：在任何情况下，面向接口编程都是一个好的习惯。（未来版本的 Spring.NET AOP 框架已经计划支持代理没有实现任何接口的类。）

本指南的剩余部分将逐步深入 AOP 框架的细节，但基本来说，都围绕这个例子来讲解。

作为深入学习的第一个例子，下面讨论如何使用配置文件实现与编程方法相同的功能；应该说，这种声明式方法要优于编程方法：

```
    <object id="consoleLoggingAroundAdvice"

type="Spring.Examples.AopQuickStart.ConsoleLoggingAroundAdvice"/>
    <object id="myServiceObject"
type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="Target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="InterceptorNames">
```

```
            <list>
                <value>consoleLoggingAroundAdvice</value>
            </list>
        </property>
    </object>
    ICommand command = (ICommand) ctx["myServiceObject"];
    command.Execute();
```

需要对上面的 XML 配置作些说明。首先，注意 ConsoleLoggingAroundAdvice 本身也是一个普通对象，可以象其它任何对象一样定义在容器中，如果通知本身需要进行依赖注入，也可以象普通对象一样配置。

第二，注意在 IoC 容器中，对应于领域对象（ServiceCommand 实例）的对象定义是 ProxyFactoryObject。ProxyFactoryObject 实现了 IFactoryObject 接口；IFactoryObject 接口的实现类会被 IoC 容器特殊对待：在本例中，该对象定义并不会返回 ProxyFactoryObject 对象，而是返回 ProxyFactoryObject 创建的对象，也就是说一个被应用了 AOP 通知的 ServiceCommand 对象（即 AOP 代理）。

第三，注意 ProxyFactoryObject 的 Target 属性值是一个内联的 ServiceCommand 对象定义，这是它的目标对象（也就是说，该对象的方法调用将被拦截）。该对象被定义为一个内联对象，在使用 ProxyFactoryObject 时这是个较好的做法，因为其它对象无法直接获得原始对象的引用，只能获得 AOP 代理。

最后，注意 ProxyFactoryObject 的 Interceptors 属性是通过名称来引用通知的。本例中只有一个通知，也就是名为 consoleLoggingAroundAdvice 的对象定义。使用对象名称列表而不使用对象引用的原因在第十二章（12.5.3 节）曾经讨论过：

“如果 ProxyFactoryObject 的 IsSingleton 属性被设为 false，它必须能够在每次调用 GetObject 时返回独立的代理对象。若所需的 Advisor 中有任何一个本身是 prototype，它就会在每次请求时返回新的实例。所以 ProxyFactoryObject 必须得从容器中获取 prototype 的对象，这样以来，保存引用的效率就不高了。”

26.2.2.使用切入点－基本概念

上节例子中的通知是没有选择性的，只是简单的拦截目标对象的所有方法（注意，是所有定义在接口中的方法，对于目标对象本身定义的方法则不会去拦截）。

对于简单的例子来说，这么做还可以，但是，如果想要只代理目标对象的部分方法，这个通知就无能为力了。例如，您可能只希望通知在调用以“Start”开头的方法时被执行；或者您可能希望只去代理某些在运行时以某个参数值调用的方法；又或者，您可能只希望代理应用了 Lockable 特性的方法。

在 Spring.NET 的 AOP 框架中，通知的应用条件（即切入点，比如当拦截到方法调用时）由 IPointcut 接口来封装（参见 12.4,Spring.NET 中的切入点）。Spring.NET 提供了许多 IPointcut 的实现类，如果不显式指定（如第一个例子）切入点，AOP 框架就会使用 TruePointcut 类：如其名称所示，这个切入点会始终匹配，所以会代理所有接口的所有方法。

现在来修改一下上面的例子，让通知只应用于名称包含“Do”的方法上，同时修改 ICommand 接口和它的实现类如下：

```
    public interface ICommand
    {
        void Execute();

        void DoExecute();
    }

    public class ServiceCommand : ICommand
    {
        public void Execute()
        {
            Console.Out.WriteLine("Service implementation :
Execute()...");
        }

        public void DoExecute()
        {
            Console.Out.WriteLine("Service implementation :
DoExecute()...");
        }
    }
```

请注意，通知本身（即 ConsoleLogginAroundAdvice 类）不需要修改；我们要修改的是通知应用的条件（地点），而不是通知本身。

可以用编程方式将通知应用于目标对象中名称包含“Do”的方法，如下：

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvisor(new DefaultPointcutAdvisor(
        new SdkRegularExpressionMethodPointcut("Do"),
        new ConsoleLoggingAroundAdvice()));
    ICommand command = (ICommand) factory.GetProxy();
    command.DoExecute();
```

上面代码的输出为：

```
Intercepted call : about to invoke next item in chain...
Service implementation...
Intercepted call : returned
```

结果显示通知应用到了 DoExcute 方法周围，因为该方法名中包含字符“Do”。如果改为调用 Execute 方法：

```
ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvisor(
    new DefaultPointcutAdvisor(
        new SdkRegularExpressionMethodPointcut("Do"),
        new ConsoleLoggingAroundAdvice()));
ICommand command = (ICommand) factory.GetProxy();

// note that there is no 'Do' in this method name
command.Execute();
```

再次运行这段代码，会发现通知没有被应用，因为 Execute 方法与定义的切入点不匹配，所以结果为：

```
Service implementation...
```

同样可用 XML 配置完成以上功能：

```
<object id="consoleLoggingAroundAdvice"

type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor">
        <property name="Pattern" value="Do"/>
        <property name="Advice">
            <object
type="Spring.Examples.AopQuickStart.ConsoleLoggingAroundAdvice"/>
        </property>
    </object>
    <object id="myServiceObject"
            type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="Target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="InterceptorNames">
            <list>
                <value>consoleLoggingAroundAdvice</value>
            </list>
        </property>
    </object>
```

到此，对切入点的这种处理引入了一个新概念：Advisor（可参见 12.4,Spring.NET 中的 Advisor）。Advisor 是切入点和通知的组合，其中切入点用来标识通知的应用条件，而通知则是在条件满足时要采取的行为。上面配置中名为 consoleLoggingAroundAdvice 的对象定义不再直接是 ConsoleLoggingAroundAdvice 类型，而是一个 RegularExpressionMethodPointcutAdvisor 对象，这个 Advisor 包含原有的通知对象和一个可以匹配方法名的正则表达式切入点，用于将通知应用到所有名称包含“Do”的方法上。其中，用来匹配方法名称的正则表达式直接以字符串形式赋值给 RegularExpressMethodPointcutAdvisor 对象的 Pattern 属性。

26.3. 深入探讨

我们在前面的小节中讨论了定义通知的基础知识，并了解了如何使用切入点来应用通知。当然，在 Spring.NET AOP 中，远不止上文提到的那几个简单通知和切入点类型。本节继续探讨 Spring.NET 的 AOP 框架，读者将了解到其它各种类型的通知以及可用的切入点（Spring.NET 提供了许多类型的通知和切入点）。

26.3.1.其它类型的通知

前文例子中的通知名为“环绕通知”。之所以叫做“环绕通知”，是因为这种通知应用在目标方法调用的前后，如同将方法调用环绕起来一样。前文定义的 ConsoleLoggingAroundAdvice 通知通过 IMethodInvocation 接口来获得被调用方法的信息，并分别在目标方法调用前后向控制台输入一行信息。通知包围着目标方法，或者说通知“环绕”在目标方法周围，“环绕通知”由此得名。

“环绕通知”使得我们有机会在目标方法调用开始前或者结束后进行某些操作；同时，也可以检查（甚至修改）目标方法的返回值。

但有时我们并不需要使用功能如此强大的通知。仍考虑 ConsoleLoggingAroundAdvice 通知，如果只需要记录某个方法即将被调用这一信息呢？此时不需要在目标方法调用后采取任何操作，也不需要访问方法调用的返回值。实际上我们只需要在目标方法调用前进行日志操作（本例中仅在控制台打印出方法的详细信息）。我们应尽量采用能满足要求且功能最少的通知类型——这是经验之谈，如果只需要在方法调用前进行某些操作，为什么还要费事去调用 Proceed()方法呢？最好的解决方法是使用“前置通知”。

26.3.1.1.前置通知

前置通知就是只在方法调用前运行的通知。前置通知不需要使用 IMethodInvocation 接口来访问目标方法本身的信息，也不能有返回值——这是件好事，因为我们不会因为忘记调用 Proceed()方法，或者因为忘记返回目标方

法的返回值而出错。如果不需要检查或改变目标方法的返回值，甚至不需要在目标方法成功返回之后做任何操作，那么“前置通知”就是最合适的通知类型。

在 Spring.NET 中，前置通知类型需要实现 Spring.Aop 命名空间下的 IMethodBeforeAdvice 接口。我们仍使用前面的例子，但这次要把实现变的简单一点。首先，定义一个“前置通知”类。

```
    public class ConsoleLoggingBeforeAdvice : IMethodBeforeAdvice
    {
        public void Before(MethodInfo method, object[] args, object
target)
        {
            Console.Out.WriteLine("Intercepted call to this method : "
+ method.Name);
            Console.Out.WriteLine("    The target is               : "
+ target);
            Console.Out.WriteLine("    The arguments are           :
");
            if(args != null)
            {
                foreach (object arg in args)
                {
                    Console.Out.WriteLine("\t: " + arg);
                }
            }
        }
    }
```

我们来将这个通知应用到 ServiceCommand 类的 Execute 方法上。下面是编程实现的方式，可以看到和前面的例子很相似，唯一的区别是在添加通知时使用了新的 ConsoleLoggingBeforeAdvice 对象：

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvice(new ConsoleLoggingBeforeAdvice());
    ICommand command = (ICommand) factory.GetProxy();
    command.Execute();
```

上面代码的输出为：

```
    Intercepted call to this method : Execute
    The target is                   :
Spring.Examples.AopQuickStart.ServiceCommand
    The arguments are               :
```

输出结果表明通知已经被应用在目标方法调用之前。注意和环绕通知不同，使用前置通知不会因忘记调用 Proceed()方法而出错，因为它根本不访问 IMethodInvocation；同样，也不会因忘记返回值而出错。

前置通知作为一个相对简单的编程模型，减少了出错的可能性。

下面是通过 XML 配置应用前置通知的方式：

```
    <object id="beforeAdvice"

type="Spring.Examples.AopQuickStart.ConsoleLoggingBeforeAdvice"/>

    <object id="myServiceObject"
        type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>beforeAdvice</value>
            </list>
        </property>
    </object>
```

26.3.1.2. 后置通知

与前置通知在目标方法调用前执行相反，后置通知是在目标方法执行后执行的通知。

在 Spring.NET 中，后置通知类型需要实现 Spring.Aop 命名空间下的 IAfterReturningAdvice 接口。我们再对前面的例子进行修改，首先定义后置通知类型：

```
    public class ConsoleLoggingAfterAdvice : IAfterReturningAdvice
    {
        public void AfterReturning(
            object returnValue, MethodInfo method, object[] args, object
target)
        {
            Console.Out.WriteLine("This method call returned
successfully : " + method.Name);
            Console.Out.WriteLine("    The target
was                    : " + target);
```

```
            Console.Out.WriteLine("    The arguments
were                    : ");
            if(args != null)
            {
                foreach (object arg in args)
                {
                    Console.Out.WriteLine("\t: " + arg);
                }
            }
            Console.Out.WriteLine("    The return value
is                  : " + returnValue);
        }
    }
```

然后,将 ConsoleLoggingAfterAdvice 通知应用到 ServiceCommand 类的 Execute 方法上。下面是编程实现的方式；可看出和使用前置通知的方法很类似（同样和使用环绕通知的配置也没什么大的差异）——唯一的区别是在添加通知时使用了新的 ConsoleLoggingAfterAdvice 对象：

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvice(new ConsoleLoggingAfterAdvice());
    ICommand command = (ICommand) factory.GetProxy();
    command.Execute();
```

结果输出为：

```
    This method call returned successfully : Execute
    The target was                         :
Spring.Examples.AopQuickStart.ServiceCommand
    The arguments were                     :
    The return value is                    : null
```

输出结果表明通知被应用在了目标方法调用之后。当然在实际开发中应该定义一个更加有用的后置通知类型，本例仅作演示用。注意与环绕通知不同，使用后置通知也不需要调用 Proceed 方法，因为后置通知和前置通知类似，也不访问 IMethodInvocation——并且，虽然后置通知会访问目标方法的返回值，但不需要返回它，不过倒是可以改变返回值的状态（当返回值是引用类型的时候），通常是通过设置返回值的属性，或者调用其方法来修改它的状态。

使用后置通知的原则和前置通知一样，就是说，如果后置通知能够满足要求，就不要使用环绕通知。后置通知同样提供了一个相对简单的编程模型，减少了出错的可能性。

后置通知的一个应用场合是对目标方法的返回值执行访问控制;考虑一下这个用例：有一个服务，用来返回一个文档 URI 列表，根据调用此服务的用户的身份不

同，服务可将用户无权访问的项从列表中删除。这只是实际应用场景之一，在实际开发中，还能发现很多地方可以应用后置通知。

下面是应用后置通知的 XML 配置：

```
    <object id="afterAdvice"

type="Spring.Examples.AopQuickStart.ConsoleLoggingAfterAdvice"/>

    <object id="myServiceObject"
        type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>afterAdvice</value>
            </list>
        </property>
    </object>
```

26.3.1.3. 异常通知

到目前为止我们已经讲过环绕通知、前置通知和后置通知，如果不需要使用 AOP 的全部功能，这些通知已经尽可以满足要求了。不过，Spring.NET 还提供了另外一种通知，“异常通知”。

顾名思义，异常通知是在目标方法抛出异常时执行的通知。异常通知的应用方法与前面讲过的通知类型并无二致。如果在应用程序执行过程中没有抛出异常，那么异常通知就不会执行。但是，如果在运行期间应用程序抛出了异常，异常通知就会介入并执行。可以使用异常通知为应用程序的不同对象应用统一的异常处理策略，或者在目标方法抛出异常时执行日志操作，或者在发生某个重要异常时通知支持团队（比如通过邮件）——其应用场合自然是不胜枚举的。

在 Spring.NET 中，异常通知类型要实现 Spring.AOP 命名空间下的 IThrowsAdvice 接口。基本上，我们只需要在异常通知的实现类中定义要处理的异常类型。先来看一下 IThrowsAdvice 接口：

```
    public interface IThrowsAdvice : IAdvice
    {
    }
```

没错，这就是 IThrowsAdvice 接口——一个没有定义任何方法的标识接口。那么 Spring.NET 怎么才能知道要调用异常通知的什么方法呢？下面的例子也许能说明这一点。首先，定义一个异常通知类型：

```
public class ConsoleLoggingThrowsAdvice : IThrowsAdvice
{
    public void AfterThrowing(Exception ex)
    {
        Console.Out.WriteLine("Advised method threw this exception :
" + ex);
    }
}
```

同时修改 ServiceCommand 类的 Execute 方法，让它抛出一个异常。这样当异常抛出时上面定义的 ConsoleLoggingThrowsAdvice 通知就会介入：

```
public class ServiceCommand : ICommand
{
    public void Execute()
    {
        throw new UnauthorizedAccessException();
    }
}
```

下面使用编程方式将 ConsoleLoggingThrowsAdvice 通知应用到 ServiceCommand 的 Execute 方法上：

```
ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvice(new ConsoleLoggingThrowsAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute();
```

执行结果如下：

```
Advised method threw this exception :
System.UnauthorizedAccessException:
Attempted to perform an unauthorized operation.
```

从输出中可以看出，当目标方法抛出异常时 ConsoleLoggingThrowsAdvice 通知被触发。关于这个通知类型有几件要注意的事情，我们逐一讨论。

在 Spring.NET 中，要实现异常通知，必须实现 IThrowsAdvice 接口。然后，针对要处理的每一种异常定义一个相应的处理方法，方法必须以 AfterThrowing 命名：

```
    void AfterThrowing(Exception ex)
```

异常处理方法必须以 AfterThrowing 命名。这一点很重要，如果异常处理方法不以此为名，异常通知就无法执行。目前，这一命名约束还无法通过配置体现出来（未来版本可能会支持）。

异常处理方法至少需要一个参数来指定异常的类型，异常类型可以是 Exception，也可以是其任何子类。我们应尽量将异常参数声明为要处理的具体异常类型，也就是说，如果要将异常通知应用到一个只抛出 ArgumentException 异常的方法上，那么异常处理方法的参数就应该是 ArgumentException 类型，如下：

```
    void AfterThrowing(ArgumentException ex)
```

注意异常处理方法可以返回任何值，但都会被 AOP 框架忽略，所以一般都返回 void。

下面是应用异常通知的 XML 配置：

```
    <object id="throwsAdvice"

type="Spring.Examples.AopQuickStart.ConsoleLoggingThrowsAdvice"/>

    <object id="myServiceObject"
        type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>throwsAdvice</value>
            </list>
        </property>
    </object>
```

异常通知无法抑制异常（exception swallowing）。无法在异常通知中抑制异常并阻止它在调用堆栈中冒泡。最接近抑制异常的处理方式是在异常处理方法中将异常包装到另一个异常中，然后将其抛出。可以用此功能实现某种异常翻译或者异常消除（scrubbing）策略，以将指定异常（例如数据访问对象抛出的 SqlException 或 OracleException 异常）替换为一个可由业务层服务对象处理的业务异常。下面举一个这种异常通知的例子，只为演示用，没什么实际意义：

```
    public class DataAccessExceptionScrubbingThrowsAdvice :
IThrowsAdvice
    {
```

```
        public void AfterThrowing (SqlException ex)
        {
            // business objects in higher level service layer need only
deal with PersistenceException...
            throw new PersistenceException ("Cannot access persistent
storage.", ex.StackTrace);
        }
    }
```

Spring.NET 的数据访问类库已经实现了类似的功能，当然，比这个例子要复杂的多。

这里我们对异常通知及其实现的讨论相当简单，还有很多的特性没有涉及，比如可以在异常处理方法中访问抛出异常的原始对象、方法及其参数等信息。（详细内容可参考 12.3.2.3,异常通知）

26.3.1.4. 引入（Introductions 或 mixins）

简单的说，引入就是在运行时透明的向任意对象添加新的状态和行为。引入（也称为混入（mixins））可以模仿多重继承，特别是在将横切（crosscutting）状态或操作同时应用到不在同一继承体系内的多个对象时，引入通知的作用非常之大。

26.3.1.5.分层通知（Layering advice，应该是指通知链）

在上面的例子中，仅为目标对象应用了一个通知。如果 Spring.NET 只能做到这一步，那就没什么意思了：Spring.NET 支持为一个目标对象应用多个通知。例如，可能需要同时为一个服务对象应用事务和安全访问检查等许多通知。

为使本节不至过于冗长，我们来将上文实现的所有通知类型应用到同一个目标对象上——我们只使用默认切入点来代理所有连接点，然后看一下不同通知类型的执行顺序。

请参考前文例子中对各种通知类型的定义，在此我们把 ConsoleLoggingAroundAdvice，ConsoleLoggingBeforeAdvice，ConsoleLoggingAfterAdvice，和 ConsoleLoggingThrowsAdvice 通知应用到同一个 ServiceCommand 实例上去：

```
ProxyFactory factory = new ProxyFactory(new ServiceCommand());
factory.AddAdvice(new ConsoleLoggingBeforeAdvice());
factory.AddAdvice(new ConsoleLoggingAfterAdvice());
factory.AddAdvice(new ConsoleLoggingThrowsAdvice());
factory.AddAdvice(new ConsoleLoggingAroundAdvice());
```

```
    ICommand command = (ICommand) factory.GetProxy();
    command.Execute();
```

下面是使用 IoC 容器时的配置文件：

```
    <object id="throwsAdvice"

type="Spring.Examples.AopQuickStart.ConsoleLoggingThrowsAdvice"/>
    <object id="afterAdvice"

type="Spring.Examples.AopQuickStart.ConsoleLoggingAfterAdvice"/>
    <object id="beforeAdvice"

type="Spring.Examples.AopQuickStart.ConsoleLoggingBeforeAdvice"/>
    <object id="aroundAdvice"

type="Spring.Examples.AopQuickStart.ConsoleLoggingAroundAdvice"/>

    <object id="myServiceObject"
        type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>throwsAdvice</value>
                <value>afterAdvice</value>
                <value>beforeAdvice</value>
                <value>aroundAdvice</value>
            </list>
        </property>
    </object>
```

26.3.1.6. 配置通知

请记住通知只是一个普通的.NET 对象（一个 PONO）；通知对象可以随意定义构造器，也可以拥有属性。所以，我们可以用 Spring.NET 的 IoC 容器来管理通知，并可充分利用依赖注入的优点。

考虑一个需要向在线支持中心报告致命异常的异常通知。该通知可以通过属性引用一个报告服务，IoC 容器可在通知对象创建时对这个属性进行依赖注入，所依赖的报告服务可以是一个 Log4Net 包装类，或者一个 Windows EventLog 包装类，或者是一个用于将异常详细信息通过邮件发送给支持团队的自定义服务对象。

同时，请牢记在 Spring.NET 中，AOP 和 IoC 是相互独立的。在上文的例子中也可以看出这一点：每个例子都有相关的编程方式和声明方式（通过 IoC 的 XML 配置机制）。

26.3.2. 使用特性定义切入点

26.4.The Spring.NET AOP Cookbook

前面讲的 AOP 例子都比较简单，主要目的是为了介绍 Spring.NET AOP 的概念。本节则要讨论一些应用 Spring.NET AOP 的真实例子。

26.4.1. 缓存

本节讨论 AOP 最常用的功能之一：缓存。

我们来考虑一下这个场景：在应用程序中，有一些在整个运行周期内始终要维护的静态引用数据。在整个应用程序的运行期，这些数据很少改变，并且在数据库中只用来满足不同表之间的引用完整性。这种静态引用数据(通常是不会改变的)的一个例子就是国家对象（由国家名称和代码组成）集合。我们要做的就是从集合中读出这些数据，并将它们“固定”在缓存中。这样就不必在每次引用国家数据时都要到数据库中读取数据了（例如，使用国家数据在一个 WinForm 窗口中填充下拉控件）。

用于读取国家对象集合的数据访问对象（DAO）是 AdoCountryDao（一个名为 ICountryDao 的 DAO 接口的实现类）。AdoCountryDao 的实现相当简单，其中只有一个方法 FindAllCountries。该方法调用时会通过 IDataReader 查询国家数据并返回。

```
public class AdoCountryDao : ICountryDao
{
    public IList FindAllCountries ()
    {
        // implementation elided for clarity...
        return countries;
    }
}
```

理想状态下，我们希望 FindAllCountries 方法在第一次调用时将结果缓存起来。同时也希望以非侵入的方式做到这一点，因为在应用程序的不同地方都可能用到缓存，也就是说，缓存就是一个所谓的横切关注点，所以，我们可以用 Spring.NET 的 AOP 框架来实现。

在本例中，我们用一个.NET 特性来标识某个地方需要应用缓存。Spring.NET 中包含一系列通用的.NET 特性，其中一个就是 CacheAttribute。在此，我们只要向 FindAllCountries 实例方法应用 CacheAttribute 特性：

```
public class AdoCountryDao : ICountryDao
{
    [Cache]
    public IList FindAllCountries ()
    {
        // implementation elided for clarity...
        return countries;
    }
}
```

SpringAir 是随 Spring.NET 一起发布的参考程序，其中包括一个使用 Spring.NET AOP 缓存机制的例子（参见第二十九章,SpringAir - 参考程序）。

26.4.2. 性能监视

我们来看一下在应用程序中进行性能监视是多么的容易。在实现中，我们使用一或多个 Windows 性能计数器来显示和跟踪性能数据。

26.4.3. 重试规则（Retry Rules）

最后一个例子，我们来讨论一个简单（但很有用）的方面——重试。使用 Spring.NET 的 AOP 框架，可以很容易的将一个（可配置的）方面应用到某个操作上（比如一个打开数据连接的方法），以便在操作失败时能够按期望的次数尝试重新操作。

26.5. Spring.NET AOP 最佳实践

Spring.NET AOP 是一个 80%的 AOP 解决方案，因为在典型的企业应用中，有 80%的问题适合应用 AOP，Spring.NET 为这 80%的问题提供了解决方案。最后这一节讲述哪些情况下适合使用 Spring.NET AOP（80%），哪些情况不适用（另外 20%）。

第二十七章. .NET Remoting 快速入门

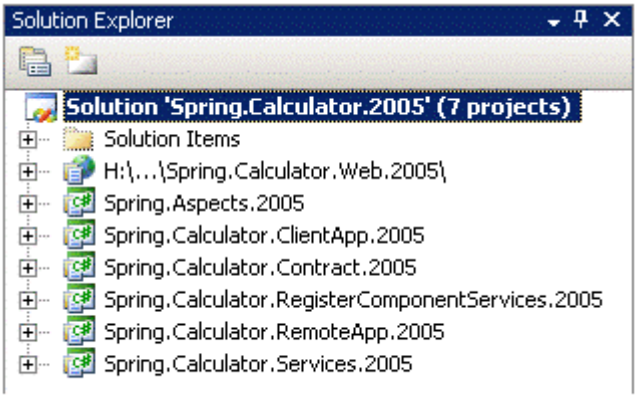
27.1. 简介

本章讨论 Spring.NET 中 Remoting 基础框架的用法。相关的类型定义在 Spring.Services 程序集的 Spring.Services.Remoting 命名空间下。在服务端，将.NET 对象导出为 CAO 或 SAO 的主要策略是使用 CaoExporter 或 SaoExporter 类，在客户端则使用 CaoFactoryObject 或 SaoFactoryObject 获取它们的引用。本章的例子假设读者熟悉.NET Remoting 的基本概念。如果没有接触过.NET Remoting，可以参考本章末尾列出的资源。

27.2. Remoting 示例程序

本例和 Spring.NET 的其它例子一样，刻意做的非常简单。我们会在例子中定义一个能够远程访问的简单计算器类，并将其导出为不同模式的远程对象（CAO，SAO-SingleCall，SAO-Singleton），同时，也将讨论如何向 SAO 添加 AOP 通知。

本章的示例程序位于 examples\Spring\Spring.Examples.Calculator 目录下，其中包含多个项目。



Spring.Calculator.Contract 项目包括两个接口，ICalculator 接口定义了计算器的基本操作，IAdvancedCalculator 接口则在此基础上加入了将计算结果保存至内存的功能。（哈，功能强大，HP-12C 可要当心了（按：HP-12C 是惠普出品的高级金融计算器））。Spring.Calculator.Services 项目中包含这两个接口的实现类，Calculator 和 AdvancedCalculator。AdvancedCalculator 类的目的是为了向读者说明如何设置 SAO-Singleton 对象的状态。注意这些实现类都没有继承 MarshalByRefObject。Spring.Calculator.ClientApp 项目是客户端应用程序。Spring.Calculator.RemoteApp 则是一个控制台应用程序，作为 AdvanceCalculator 远程对象的宿主程序。Spring.Calculator.Aspects 项目定义了一些日志通知，用来说明如何向远程 SAO 应用 AOP 通知。Spring.Calculator.RegisterComonentServices 和 Spring.Calculator.Web 分

别用于导出企业服务和 Web 服务，与本章内容无关。下面是 ICalculator 接口的代码：

```
public interface ICalculator
{
    int Add(int n1, int n2);

    int Substract(int n1, int n2);

    DivisionResult Divide(int n1, int n2);

    int Multiply(int n1, int n2);
}

[Serializable]
public class DivisionResult
{
          private int _quotient = 0;
          private int _rest = 0;

          public int Quotient
          {
                  get { return _quotient; }
                  set { _quotient = value; }
          }

          public int Rest
          {
                  get { return _rest; }
                  set { _rest = value; }
          }
}
```

下面是 IAdvancedCalculator 接口的代码，它扩展了 ICalculator 接口，加入了保存计算结果的功能：

```
public interface IAdvancedCalculator : ICalculator
{
        int GetMemory();

        void SetMemory(int memoryValue);

        void MemoryClear();
```

```
        void MemoryAdd(int num);
}
```

在这个解决方案中，我们用接口在.NET Remoting 客户端和服务器间共享信息。这样做的优点是客户端不需要引用包含实现类的程序集。不让客户端引用实现类的原因有很多。一个是为安全性考虑，因为 IL 代码很容易被反编译，所以客户端就有可能获取实现的源代码。更重要的是，客户端和服务端是解耦的，服务端可以更换接口的实现而不致对客户端造成影响，即客户端不需要修改代码。另外，抛开.NET Remoting 不谈，用接口来发布服务合同也是很好的 OO 设计。客户端完全可以采用一个跟 Remoting 无关的实现，比如实现了同一接口的本地类、测试类或 Web 服务等。使用 Spring.NET 最大的好处，就是能将“基于接口编程”的开销降为最低。这样，.NET Remoting 所提倡的最佳编程方式正好和 Spring.NET 建议的开发方式吻合。OK，现在已经不存在 OO 设计上的障碍了，我们来进一步看看如何实现。

27.3. 实现

Spring.Calculator.Services 项目中的 Calculator 类都相当简单。惟一值得一提的是其中处理内存存储的方法，我们用构造器参数注入来显式配置内存存储的状态。实现类的部分代码如下所示：

```
public class Calculator : ICalculator
{

    public int Add(int n1, int n2)
    {
        return n1 + n2;
    }

    public int Substract(int n1, int n2)
    {
        return n1 - n2;
    }

    public DivisionResult Divide(int n1, int n2)
    {
        DivisionResult result = new DivisionResult();
        result.Quotient = n1 / n2;
        result.Rest = n1 % n2;
        return result;
    }

    public int Multiply(int n1, int n2)
    {
```

```
        return n1 * n2;
    }

}

public class AdvancedCalculator : Calculator, IAdvancedCalculator
{

  private int memoryStore = 0;

  public AdvancedCalculator()
  {}

  public AdvancedCalculator(int initalMemory)
  {
    memoryStore = initalMemory;
  }

  public int GetMemory()
  {
    return memoryStore;
  }

  // other methods omitted in this listing...

}
```

Spring.Calculator.RemotedApp 项目用一个控制台应用程序作为远程对象的宿主。代码也是非常简单的，如下所示：

```
public static void Main(string[] args)
{
    try
    {
        // initialization of Spring.NET's IoC container
        IApplicationContext ctx = ContextRegistry.GetContext();

        Console.Out.WriteLine("Server listening...");
    }
    catch (Exception e)
    {
       Console.Out.WriteLine(e);
    }
    finally
```

```
    {
        Console.Out.WriteLine("--- Press <return> to quit ---");
        Console.ReadLine();
    }
}
```

.NET Remoting 的通道在配置文件（App.config）中用标准的 system.runtime.remoting 节点配置。在本例中，我们使用 tcp 通道，8005 端口：

```
<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="tcp" port="8005" />
    </channels>
  </application>
</system.runtime.remoting>
```

Spring.NET 的应用程序上下文配置如下。其中引用的各个资源文件将对象发布为不同的远程对象。本例使用的 AOP 通知是一个简单的、使用 log4net 记录日志的环绕通知。

```
<configSections>
        <sectionGroup name="spring">
        <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core" />
        <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
        <section name="parsers"
type="Spring.Context.Support.ConfigParsersSectionHandler,
Spring.Core" />
        </sectionGroup>
        <section name="log4net"
type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
</configSections>

<spring>
        <parsers>
        <parser namespace="http://www.springframework.net/remoting"
        type="Spring.Remoting.RemotingConfigParser, Spring.Services"
        schemaLocation="assembly://Spring.Services/Spring.Remoting/sp
ring-remoting.xsd" />
        </parsers>
        <context>
        <resource uri="config://spring/objects" />
        <resource
```

```
uri="assembly://RemoteServer/RemoteServer.Config/cao.xml" />
        <resource
uri="assembly://RemoteServer/RemoteServer.Config/saoSingleCall.xml" />
        <resource
uri="assembly://RemoteServer/RemoteServer.Config/saoSingleCall-aop.xm
l" />
        <resource
uri="assembly://RemoteServer/RemoteServer.Config/saoSingleton.xml" />
        <resource
uri="assembly://RemoteServer/RemoteServer.Config/saoSingleton-aop.xml
" />
        </context>
        <objects xmlns="http://www.springframework.net">
       <description>Definitions of objects to be exported.</description>

 <object type="Spring.Remoting.RemotingConfigurer, Spring.Services">
 <property name="Filename"
value="Spring.Calculator.RemoteApp.exe.config" />
 </object>

        <object id="Log4NetLoggingAroundAdvice"
             type="Spring.Aspects.Logging.Log4NetLoggingAroundAdvice,
Spring.Aspects">
        <property name="Level" value="Debug" />
        </object>

        <object id="singletonCalculator"
             type="Spring.Calculator.Services.AdvancedCalculator,
Spring.Calculator.Services">
        <constructor-arg type="int" value="217"/>
        </object>

        <object id="singletonCalculatorWeaved"
type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
        <property name="target" ref="singletonCalculator"/>
        <property name="interceptorNames">
        <list>
        <value>Log4NetLoggingAroundAdvice</value>
        </list>
        </property>
        </object>

        <object id="prototypeCalculator"
```

```
type="Spring.Calculator.Services.AdvancedCalculator,Spring.Calculator
.Services" singleton="false">
        <constructor-arg type="int" value="217"/>
        </object>

        <object id="prototypeCalculatorWeaved"
type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
        <property name="targetSource">
        <object type="Spring.Aop.Target.PrototypeTargetSource,
Spring.Aop">
       <property name="TargetObjectName" value="prototypeCalculator"/>
        </object>
        </property>
        <property name="interceptorNames">
        <list>
        <value>Log4NetLoggingAroundAdvice</value>
        </list>
        </property>
        </object>
        </objects>
</spring>
```

其中计算器类的对象定义（比如说 singletonCalculator）和普通对象定义的配置方法一样。在将这些对象发布为远程对象的时候，可以用 Spring.Remoting.CaoExporter 将其发布为 CAO，或用 Spring.Remoting.SaoExporter 将其发布为 SAO。这两个导出类的对象定义都需要用 TargetName 属性引用目标对象的名称。如果要导出 SingleCall 模式的 SAO 或 CAO,要将目标对象定义为“prototype“（即 singleton=false)。SaoExporter 类的 ServiceName 属性值是远程对象的名称，会包含在远程对象的 URL 中供客户端访问。另外，如果需要将远程对象的租赁期设为无限，可以将 Infinite 的属性值设为 true。

导出 Singleton 模式 SAO 的配置如下：

```
<objects
        xmlns="http://www.springframework.net"
        xmlns:r="http://www.springframework.net/remoting">

        <description>Registers the calculator service as a SAO in
'Singleton' mode.</description>

        <r:saoExporter
        targetName="singletonCalculator"
        serviceName="RemotedSaoSingletonCalculator" />
</objects>
```

这是用 Spring.NET 的 Remoting Schema 配置的，当然也可以用标准的对象定义，如下：

```
<object name="saoSingletonCalculator"
type="Spring.Remoting.SaoExporter, Spring.Services">
 <property name="TargetName" value="singletonCalculator" />
 <property name="ServiceName" value="RemotedSaoSingletonCalculator" />
</object>
```

这样就可以发布一个 URL 为 tcp://localhost:8005/RemotedSaoSingletonCalculator 的远程对象。使用 SaoExporter 和 CaoExporter 发布其它类型远程对象的方法与此类似，可以参考 RemoteServer 项目的配置文件。

客户端应用程序会连接到远程计算器服务，请求它的当前内存值，该值被预设为 217，然后执行一个简单的加法。在服务端，已经用.NET Remoting 的标准配置将通道配置好了，如下：

```
<system.runtime.remoting>
    <application>
        <channels>
                <channel ref="tcp"/>
        </channels>
    </application>
</system.runtime.remoting>
```

客户端实现的代码如下：

```
public static void Main(string[] args)
{
        try
        {
                Pause();

                IApplicationContext ctx = ContextRegistry.GetContext();

                Console.Out.WriteLine("Get Calculator...");
                IAdvancedCalculator firstCalc = (IAdvancedCalculator)
ctx.GetObject("calculatorService");
    Console.WriteLine("Divide(11, 2) : " + firstCalc.Divide(11, 2));
                Console.Out.WriteLine("Memory = " +
firstCalc.GetMemory());
                firstCalc.MemoryAdd(2);
                Console.Out.WriteLine("Memory + 2 = " +
firstCalc.GetMemory());
```

```
                Console.Out.WriteLine("Get Calculator...");
                IAdvancedCalculator secondCalc = (IAdvancedCalculator)
ctx.GetObject("calculatorService");
                Console.Out.WriteLine("Memory = " +
secondCalc.GetMemory());
        }
        catch (Exception e)
        {
                Console.Out.WriteLine(e);
        }
        finally
        {
                Pause();
        }
}
```

注意，客户端代码并不知道自己使用的是一个远程对象。其中的 pause()方法只是让客户端程序等待用户按下回车键再开始运行，以免在服务端初始化之前就请求远程对象。.NET Remoting 基础框架要在 Spring.NET 的 IoC 容器之前初始化。在客户端的配置中，可以很方便的更换从容器获取的 calculatorService 引用。在复杂点的应用中，计算器服务可能会依赖程序中的其它对象，比如说工作流程处理层中的对象。下面的代码是客户端使用本地实现和远程实现的配置。同一个计算器对象还可以发布为 web 服务或服务组件（企业服务），本章不讨论这些内容。

```
<spring>
  <context>
    <resource uri="config://spring/objects" />

    <!-- Only one at a time ! -->

    <!-- ================================== -->
    <!-- In process (local) implementations -->
    <!-- ================================== -->
    <resource
uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.InProcess/inProcess.xml" />

    <!-- ======================== -->
    <!-- Remoting implementations -->
    <!-- ======================== -->
    <!-- Make sure 'RemoteApp' console application is running and
listening. -->
```

```
    <!-- <resource

uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.Remoting/cao.xml" /> -->
    <!-- <resource

uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.Remoting/cao-ctor.xml" /> -->
    <!-- <resource

uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.Remoting/saoSingleton.xml" /> -->
    <!-- <resource

uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.Remoting/saoSingleton-aop.xml" /> -->
    <!-- <resource

uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.Remoting/saoSingleCall.xml" /> -->
    <!-- <resource

uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.Remoting/saoSingleCall-aop.xml" /> -->

    <!-- ============================ -->
    <!-- Web Service implementations -->
    <!-- ============================ -->
    <!-- Make sure 'http://localhost/SpringCalculator/' web application
is running -->
    <!-- <resource

uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.WebServices/webServices.xml" /> -->
    <!-- <resource

uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.WebServices/webServices-aop.xml" /> -->

    <!-- ================================= -->
    <!-- EnterpriseService implementations -->
    <!-- ================================= -->
    <!-- Make sure you register components with
'RegisterComponentServices' console application.  -->
```

```
    <!-- <resource

uri="assembly://Spring.Calculator.ClientApp/Spring.Calculator.ClientA
pp.Config.EnterpriseServices/enterpriseServices.xml" /> -->
  </context>
</spring>
```

inProcess.xml 配置文件直接定义了一个 AdvancedCalculator 对象：

```
<objects xmlns="http://www.springframework.net">

  <description>inProcess</description>

  <object id="calculatorService"
     type="Spring.Calculator.Services.AdvancedCalculator,
Spring.Calculator.Services" />

</objects>
```

在定义远程对象的客户端引用时，我们使用了工厂对象。在引用 SAO 时，使用 SaoFactoryObject 类；引用 CAO 对象则使用 CaoFactoryObject 类。下面的配置用来获取前面导出的 Singleton-SAO 的客户端引用：

```
<objects xmlns="http://www.springframework.net">

        <description>saoSingleton</description>

        <object id="calculatorService"
type="Spring.Remoting.SaoFactoryObject, Spring.Services">
                <property name="ServiceInterface"

value="Spring.Calculator.Interfaces.IAdvancedCalculator,
Spring.Calculator.Contract" />
                <property name="ServiceUrl"
value="tcp://localhost:8005/RemotedSaoSingletonCalculator" />
        </object>

</objects>
```

TODO: Show use of custom .net remoting schema.

在 SaoFactoryObject 的定义中，必须指定 ServiceInterface 和 ServiceUrl 属性。在此可以利用 Spring.NET 的属性值替换功能（按：请参考第四章），利用环境变量、标准.NET 配置节点或外部配置文件来简化 URL 的配置。这样也很容

易为测试、QA 和产品等不同阶段切换不同的环境。比如说，可以用下面这种方式来定义 ServiceUrl 的值...

```
<property name="ServiceUrl"
value="${protocol}://${host}:${port}/RemotedSaoSingletonCalculator"
/>
```

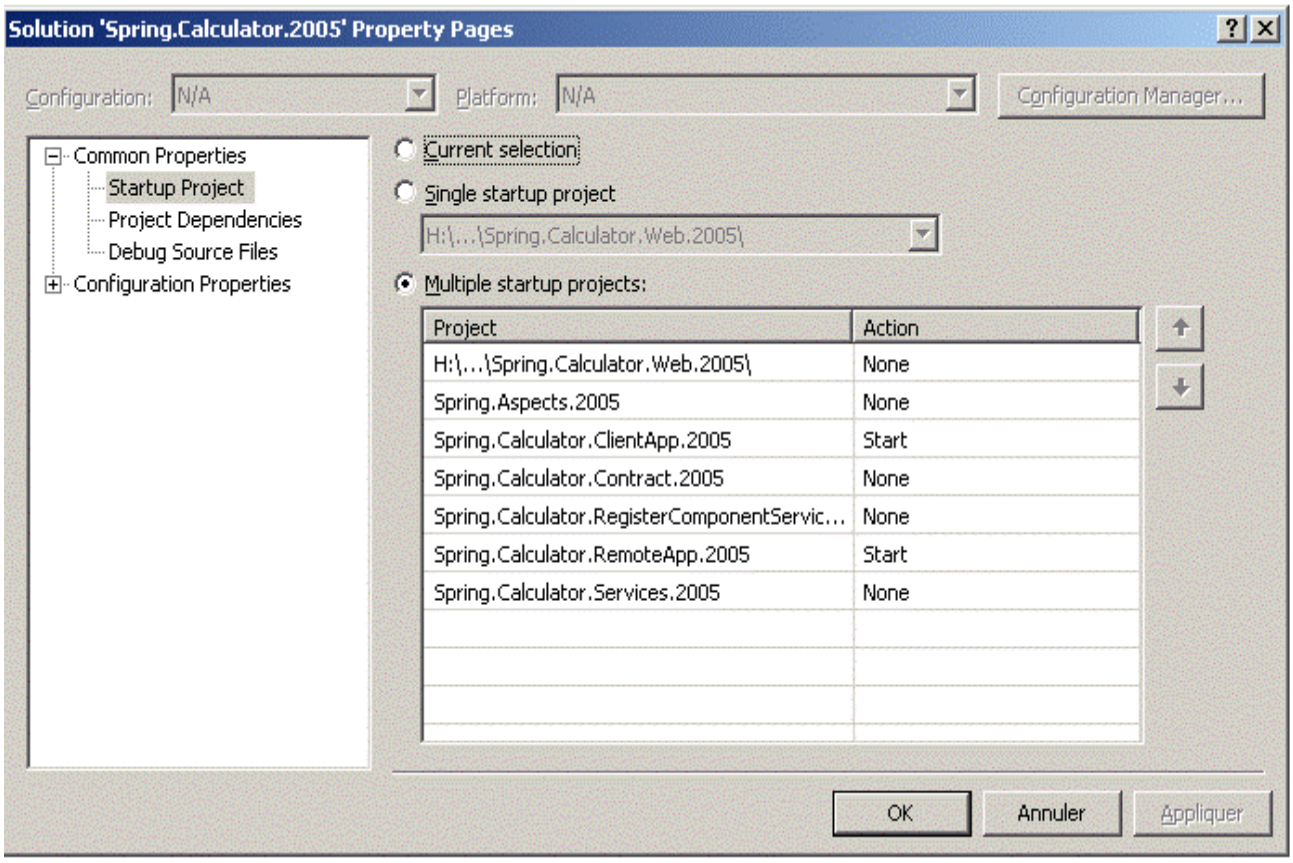
其中的具体值是在其它地方定义的，关于这方面的内容可以参考 4.9.1,PropertyPlaceholderConfigurer。前面提到过，使用配置文件的灵活性就在于，仅修改配置文件就可以更换实现了同一接口的不同的对象（远程或本地的）。

在客户端，可用以下配置获取前面发布的 CAO 的引用：

```
<objects xmlns="http://www.springframework.net">

        <description>cao</description>

        <object id="calculatorService"
type="Spring.Remoting.CaoFactoryObject, Spring.Services">
                <property name="RemoteTargetName"
value="prototypeCalculator" />
                <property name="ServiceUrl"
value="tcp://localhost:8005" />
        </object>

</objects>
```

27.4. 运行程序

我们已经把示例程序的完整实现看了一遍，现在可以运行它了。注意可以设置 VS.NET 让它同时运行两个程序，如下：



程序运行以后，服务端的输出如下... TO BE DONE...

27.5. Remoting Schema

Doc 目录下的 spring-remoting.xsd 文件可允许开发人员用较简短的语法配置 Remoting 的对象定义。可以用 Doc 目录下的 NAnt 脚本“install-shcema”将该文件安装到 VS.NET 环境中去。具体步骤可以参考第二十四章。

RemoteApp 和 ClientApp 项目下的配置文件都使用了 Remoting schema 来定义远程对象。下面的代码节选自配置文件，您会发现使用这种 Schema 定义远程对象感觉还是不错的。

```
<!-- Calculator definitions -->
<object id="singletonCalculator"
   type="Spring.Calculator.Services.AdvancedCalculator,
Spring.Calculator.Services">
        <constructor-arg type="int" value="217" />
</object>

<object id="prototypeCalculator"
   type="Spring.Calculator.Services.AdvancedCalculator,
```

```
Spring.Calculator.Services" singleton="false">
        <constructor-arg type="int" value="217" />
</object>

<!-- CAO object -->
<r:caoExporter targetName="prototypeCalculator" infinite="false">
  <r:lifeTime initialLeaseTime="2m" renewOnCallTime="1m"/>
</r:caoExporter>

<!-- SAO Single Call -->
<r:saoExporter
  targetName="prototypeCalculator"
  serviceName="RemotedSaoSingleCallCalculator"/>

<!-- SAO Singleton -->
<r:saoExporter
  targetName="singletonCalculator"
  serviceName="RemotedSaoSingletonCalculator" />
```

注意，远程对象的 Singleton 模式是由目标对象定义决定的。上例中，因为“prototypeCalculator”的 singleton 属性为 false，所以用它发布的 SAO 对象就是一个 Single-Call 的 SAO，每次调用远程对象的方法时，都会创建新的对象。

TODO: Use Document X! to document the schema.

27.6. 参考资源

.NET Remoting 是一个很庞大的课题。MSDN 上可以找到些介绍性的文章。在这方面，Ingo Rammer 是一个权威，由其维护的 Remoting 参考资源相当丰富：

- An Introduction to Microsoft .NET Remoting Framework
- Microsoft .NET Remoting: A Technical Overview
- Advanced .NET Remoting (authored by Ingo Rammer)
- .NET Remoting FAQ

第二十八章. Web 框架快速入门

28.1.简介

Web 快速入门解决方案用 Spring.Web 实现了一个基本的“HelloWorld”程序。您可以从这个程序开始看起，然后再研究 SpringAir 应用程序中使用的高级特性。

第二十九章. SpringAir - 参考程序

29.1. 简介

本章介绍一个使用 Spring.NET 的参考性应用程序。

29.2. 架构

...

29.3. 实现

...

29.3.1. 领域层

...

29.3.2. 服务层

...

29.3.3. Web 层

...

29.4. 总结

...

第三十章. 数据访问快速入门

30.1. 简介

开发人员可以通过数据访问快速入门程序了解 AdoTemplate（包括泛型和非泛型版本)的用法以及如果使用 Spring.Data.Objects 命名空间中的类进行基于对象

的数据访问的方法。该程序位于 examples/DataAccessQuickStart 目录下，使用 Northwind 数据库。

该程序包含一个假的 DAO 对象和一组 NUnit 测试，目的是供练习，并非一个完整的应用程序。要在 VS.NET 中运行这些测试可以安装 TestDriven.NET、ReSharper 或其它类似的工具。程序中用到的 DAO 类及其演示的用法如下所示：

- CommandCallbackDao——ICommandCallback 和 CommandCallbackDelegate 的用法。
- ResultSetExtractorDao——IResultSetExtractor 和 ResultSetExtractorDelegate 的用法。
- RowCallbackDao——IRowCallback 和 RowCallbackDelegate 的用法。
- RowMapperDao——IRowMapper 和 RowMapperDelegate 的用法。
- QueryForObject——QueryForObject 的用法。
- StoredProcDao——Spring.Data.Objects.StoredProcedure 的用法。

Spring.DataQuickStart.Domain 命名空间定义了几个简单的领域对象，上面 DAO 的方法一般都返回这些对象的集合。

第三十一章. 事务快速入门

31.1. 简介

读者可以通过事务快速入门程序了解 Spring.NET 事务管理的用法。本例的数据库包括两个简单的表，credit 和 debit，其中包含 Identifier 和 Amount 字段。该程序向读者介绍使用特性进行声明式事务管理的用法，以及如何在不修改任何代码的情况下，只通过配置文件更改事务管理器（本地或分布式的）的用法。

31.2. 应用程序概述

例程的设计非常简单，包括两个逻辑层：位于 Spring.TxQuickStart.Services 命名空间的业务服务层和位于 Spring.TxQuickStart.Dao 命名空间的 DAO 层。业务逻辑就是一个简单的银行转账，这是直接从 Sahil Malik 的《Pro ADO.NET》一书中拿来的。“存款”由数据库中的 creadit 和 debit 表保存。可以用 CreditsDebitsSchema.sql 在 Sql Server 中创建这两个表。转账需要在这两个表间进行 ACID 操作。credit 表的 DAO 操作由 IAccountCreditDao 接口定义，debit 表的 DAO 操作由 IAccountDebitDao 接口定义。基于 AdoTemplate 的实现位于 Spring.TxQuickStart.Dao.Ado 命名空间中，请注意 Spring.NET 的事务管理框架允许在同一事务中混合使用各种数据访问技术，比如 ORM 和 ADO.NET。下次发布该示例程序时，会包含这一功能。

31.2.1. 接口

IAccountManager 和两个 DAO 接口如下所示：

```
public interface IAccountManager
{
void DoTransfer(float creditAmount, float debitAmount);
}


public interface IAccountCreditDao
{
void CreateCredit(float creditAmount);
}

public interface IAccountDebitDao
{
void DebitAccount(float debitAmount);
}
```

DOH! - Computer crash - lost work. Please look on the web for the latest updates.

第三十二章. Java 开发人员必读

32.1. 简介

本章意在使 Java 版 Spring 的开发人员能快速的熟悉 Spring.NET。这里并不打算对.NET 版和 Java 版做完整的比较，而是把重点放在了 Spring.NET 与 Java 版的主要不同点上。

32.2. Beans 和 Objects

首先，在名词上有些许的不同。从前随处可见的“bean”，在 Spring.NET 中则使用“object”。下面用一个简单的配置文件来比较一下这两个名词，首先是 Java 版 MovieFinder 应用程序的 application.xml 文件：

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
 <bean id="MyMovieLister" class="MovieFinder.MovieLister">
```

```
 <property name="finder" ref="MyMovieFinder"/>
 </bean>
 <bean id="MyMovieFinder" class="MovieFinder.SimpleMovieFinder"/>
</beans>
```

然后是 Spring.NET 的配置文件：

```
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net
http://www.springframework.net/xsd/spring-objects.xsd">
 <object name="MyMovieLister"
 type="Spring.Examples.MovieFinder.MovieLister,
Spring.Examples.MovieFinder">
 <property name="MovieFinder" ref="MyMovieFinder"/>
 </object>
 <object name="MyMovieFinder"
 type="Spring.Examples.MovieFinder.SimpleMovieFinder,
Spring.Examples.MovieFinder"/>
</objects>
```

可以看到，<beans>和<bean>节点在 Spring.NET 中被替换成了<objects>和<object>节点。Spring.Java 的对象定义要求使用类型全名。Spring.NET 也要求使用类型全名，但同时需要指定类型所在程序集的名称。因为.NET 没有类似"classpath"的概念，所以程序集名称是必需的。.NET 的程序集名称最多可以用四个部分来精确的描述版本信息。

对于其它节点，除了键值对的定义以外，Spring.NET 和 Spring.Java 是完全一致的。在 Java 版中，键值对由 java.util.Properties 类表示，对应的 xml 节点名称为<props>，如下所示：

```
<property name="people">
 <props>
 <prop key="PennAndTeller">The magic property</prop>
 <prop key="GeorgeCarlin">The funny property</prop>
 </props>
</property>
```

而.NET 中的键值对是由 System.Collections.Specialized.NameValueCollection 定义的，对应的 xml 节点为<name-values>。同时，按照.NET 配置文件的习惯，使用<add>节点及其 key、value 属性向键值对中添加元素。如下：

```
<property name="People">
 <name-values>
```

```
 <add key="PennAndTeller" value="The magic property"/>
 <add key="GeorgeCarlin" value="The funny property"/>
 </name-values>
</property>
```

32.3. PropertyEditor 和 TypeConverter

在 Java 中，使用 java.beans 包中的 PropertyEditor 类进行字符串和 Java 类型之间的转换。例如，当使用以逗号分隔的字符串来设置一个字符串数组属性时，可以用 StringArrayPropertyEditor 类来转型。在.NET 中，类型转换的功能是由 System.ComponentModel 命名空间下的 TypeConverter 类型提供的。同时.NET 也允许将类型转换器显式的注册给某个类型，以便能对复杂对象进行自动的转换。但是.NET 却没有提供某些经常在 Spring.Java 中用到的转换方式，比如前面提到的用逗号分隔的字符串和字符串数组之间的转换。而在 Spring.NET 中，Spring.Objects.TypeConverters 命名空间下的 StringArrayConverter 类可完成这类转换。如同 Spring.Java 一样，Spring.NET 允许用户注册自定义的类型转换器。但是，对于自定义的.NET 类型来说，最好是使用.NET 标准的类型转换器注册机制。

32.4. ResourceBundle 和 ResourceManager

32.5. 异常

在 Java 中，异常可以是已检查或未检查的。.NET 只支持未检查异常。Spring.Java 一般也都会使用未检查异常，所以经常会将已检查异常转换为未检查异常。在这方面，Spring.Java 的行为和.NET 的默认行为实际上是很相似的。

32.6. 应用程序配置

Spring.Java 经常会从外部 XML 配置文件创建 ObjectFactory 或 ApplicationContext。Spring.NET 同样也提供了此功能。不过，在.NET 中，可以用 System.Configuration 命名空间来管理应用程序的配置信息。这些功能要使用某些以特殊名称命名的配置文件：对 ASP.NET 来说是 web.config，对 Windows 窗体和控制台应用来说是<MyExe>.exe.config。其中<MyExe>是可执行文件的名字。在编译过程中，如果项目的根目录下有以 App.config 命名的文件，编译器会将其更名为<MyExe>.exe.config 并与可执行文件放置在同一目录中。

配置文件是基于 XML 的，.NET 的配置系统可以根据其中节点的名称获取自定义的配置对象。为使.NET 知道如何根据节点创建自定义配置对象，需要注册一个 IConfigurationSectionHandler 接口的实现类。Spring.NET 本身包括该接口的两个实现类，一个用于从<context>节点创建应用程序上下文，另一个则根据

<objects>节点中的对象定义配置应用程序上下文。<context>节点的功能强大且非常重要，它可以引用以 uri 语法表示的所有 IResource 资源位置，同时，也可以配置嵌套的上下文，但不用象 Spring.Java 当前的版本一样要通过编码或冗长的 XML 配置来实现。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

    <configSections>
        <sectionGroup name="spring">
            <section name="context"
type="Spring.Context.Support.ContextHandler, Spring.Core"/>
            <section name="objects"
type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
        </sectionGroup>
    </configSections>

    <spring>

        <context>
            <resource uri="config://spring/objects"/>
        </context>

        <objects>
            <description>An  example that demonstrates simple IoC
features.</description>
            <object name="MyMovieLister"
type="Spring.Examples.MovieFinder.MovieLister, MovieFinder">
                <property name="movieFinder" ref="AnotherMovieFinder"/>
            </object>
            <object name="MyMovieFinder"
type="Spring.Examples.MovieFinder.SimpleMovieFinder, MovieFinder"/>
            <!--
            An IMovieFinder implementation that uses a text file as it's
movie source...
            -->
            <object name="AnotherMovieFinder"
type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder,
MovieFinder">
                <constructor-arg index="0" value="movies.txt"/>
            </object>
        </objects>

    </spring>
```

```
</configuration>
```

<configSections>和<section>是.NET 配置文件的标准节点，用于注册能够处理其它 XML 节点的 IConfigurationSecitonHandler 实现类，在上面的代码中，注册的两个类型分别是用来处理<context>和<objects>节点的。

下面的代码利用.NET 本身提供的类从应用程序配置文件中返回一个 IApplicationContext 实例：

```
IApplicationContext ctx
 = ConfigurationUtils.GetSection("spring/context") as
IApplicationContext;
```

为了充分利用 spring/context 节点的功能，最好是使用 ContextRegistry 类对应用程序上下文进行初始化。

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

32.7. AOP 框架

32.7.1.不能在 ProxyFactoryObject 的 InterceptorNames 属性中指定目标对象名

配置 ProxyFactoryObject 对象（或对象定义）的 InterceptorNames 属性时，不能将目标对象（即被代理的对象）的名称作为列表的最后一项。在 Spring.Java 中，这么做是可以的，ProxyFactoryBean 会自动检测该属性的值，并将最后一项作为 ProxyFactoryBean 的目标对象。下面的配置对 Spring.Java 来说是有效的（当然，不考虑其中节点名称的变化），但在 Spring.NET 中是不正确的（所以不要这么做）。

```
<?xml version="1.0" encoding="utf-8" ?>
 <objects xmlns="http://www.springframework.net">
 <object id="target" type="Spring.Objects.TestObject">
 <property name="Name" value="Bingo"/>
 </object>

 <object id="nopInterceptor"
type="Spring.Aop.Interceptor.NopInterceptor"/>

 <object id="prototypeTarget"
type="Spring.Aop.Framework.ProxyFactoryObject">
```

```
 <property name="InterceptorNames" value="nopInterceptor,target"/>
<!-- not valid! -->
 </object>
 </objects>
```

在 Spring.NET 中，ProxyFactoryObject 类的 InterceptorNames 属性*只能*包含拦截器名称。目标对象应使用 TargetName 属性来指定。

Spring.NET 之所以不支持这种格式的配置，是因为在 Spring.Java 中，这种“功能“是 Rod Johnson 在最初实现 Spring AOP 时加入的，现在出于向后兼容的原因才将其保留了下来。