

完成以下两项工作：

① 对使用占位符的<bean>元素标签进行解析，得到最终的配置值。这意味着对一些半成品式的 BeanDefinition 对象进行加工处理并得到成品的 BeanDefinition 对象。

② 对 BeanDefinitionRegistry 中的 BeanDefinition 进行扫描，通过 Java 反射机制找出所有属性编辑器的 Bean（实现 java.beans.PropertyEditor 接口的 Bean），并自动将它们注册到 Spring 容器的属性编辑器注册表中（PropertyEditorRegistry）。

(4) Spring 容器从 BeanDefinitionRegistry 中取出加工后的 BeanDefinition，并调用 InstantiationStrategy 着手进行 Bean 实例化的工作。

(5) 在实例化 Bean 时，Spring 容器使用 BeanWrapper 对 Bean 进行封装。BeanWrapper 提供了很多以 Java 反射机制操作 Bean 的方法，它将结合该 Bean 的 BeanDefinition 及容器中的属性编辑器，完成 Bean 属性注入工作。

(6) 利用容器中注册的 Bean 后处理器（实现 BeanPostProcessor 接口的 Bean）对已经完成属性设置工作的 Bean 进行后续加工，直接装配出一个准备就绪的 Bean。

Spring 容器堪称一部设计精密的机器，其内部拥有众多的组件和装置。Spring 的高明之处在于，它使用众多接口描绘出了所有装置的协作蓝图，构建好 Spring 的骨架，继而通过继承体系层层推演、不断丰富，最终让 Spring 成为有血有肉的完整的框架。所以在查看 Spring 框架的源码时，有两条清晰可见的脉络：

(1) 接口层描述了容器的重要组件及组件间的协作关系。

(2) 继承体系逐步实现组件的各项功能。

接口层清晰地勾勒出 Spring 框架的高层功能，框架脉络呼之欲出。有了接口层抽象的描述后，不但 Spring 自己可以提供具体的实现，任何第三方组织也可以提供不同的实现，可以说 Spring 完善的接口层使框架的扩展性得到了很好的保证。纵向继承体系的逐步扩展，分步骤地实现框架的功能，这种实现方案保证了框架功能不会堆积在某些类身上，从而造成过重的代码逻辑负载，框架的复杂度被完美地分解开了。

Spring 组件按其所承担的角色可以划分为两类。

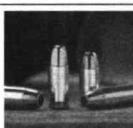
(1) 物料组件：Resource、BeanDefinition、PropertyEditor 及最终的 Bean 等，它们是加工流程中被加工、被消费的组件，就像流水线上被加工的物料一样。

(2) 设备组件：ResourceLoader、BeanDefinitionReader、BeanFactoryPostProcessor、InstantiationStrategy 及 BeanWrapper 等。它们就像流水线上不同环节的加工设备，对物料组件进行加工处理。

第 4 章介绍了 Resource 和 ResourceLoader 两个组件，本章将对其他组件进行讲解。



轻松一刻



尼古拉斯·凯奇主演的《军火之王》具有《教父》般的磅礴气势。片头很有创意，镜头跟着一颗流水线上的子弹不断前进，经过一道道工序，最终子弹走下流水线，打包、装箱、运输、卸货、开包、上膛、发射，最

后击中了目标。对 Spring 容器进行深入分析后，读者也能感受到这种按部就班、有条不紊地预设流程的魅力。

6.1.2 BeanDefinition

`org.springframework.beans.factory.config.BeanDefinition` 是配置文件`<bean>`元素标签在容器中的内部表示。`<bean>`元素标签拥有 `class`、`scope`、`lazy-init` 等配置属性，`BeanDefinition` 则提供了相应的 `beanClass`、`scope`、`lazyInit` 类属性，`BeanDefinition` 就像 `<bean>` 的镜中人，二者是一一对应的。`BeanDefinition` 类的继承结构如图 6-2 所示。

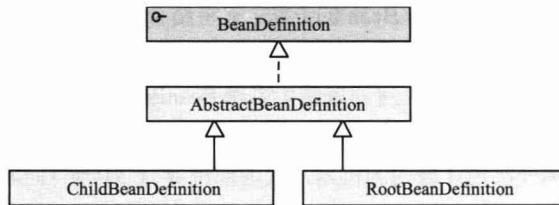


图 6-2 BeanDefinition 类继承结构

`RootBeanDefinition` 是最常用的实现类，它对应一般性的`<bean>`元素标签。我们知道，在配置文件中可以定义父`<bean>`和子`<bean>`，父`<bean>`用 `RootBeanDefinition` 表示，子`<bean>`用 `ChildBeanDefinition` 表示，而没有父`<bean>`的`<bean>`则用 `RootBeanDefinition` 表示。`AbstractBeanDefinition` 对二者共同的类信息进行抽象。

Spring 通过 `BeanDefinition` 将配置文件中的`<bean>`配置信息转换为容器的内部表示，并将这些 `BeanDefinition` 注册到 `BeanDefinitionRegistry` 中。Spring 容器的 `BeanDefinitionRegistry` 就像 Spring 配置信息的内存数据库，后续操作直接从 `BeanDefinitionRegistry` 中读取配置信息。一般情况下，`BeanDefinition` 只在容器启动时加载并解析，除非容器刷新或重启，这些信息不会发生变化。当然，如果用户有特殊的需求，也可以通过编程的方式在运行期调整 `BeanDefinition` 的定义。

创建最终的 `BeanDefinition` 主要包括两个步骤。

(1) 利用 `BeanDefinitionReader` 读取承载配置信息的 `Resource`，通过 XML 解析器解析配置信息的 `DOM` 对象，简单地为每个`<bean>`生成对应的 `BeanDefinition` 对象。但是这里生成的 `BeanDefinition` 可能是半成品，因为在配置文件中，可能通过占位符变量引用外部属性文件的属性，这些占位符变量在这一步里还没有被解析出来。

(2) 利用容器中注册的 `BeanFactoryPostProcessor` 对半成品的 `BeanDefinition` 进行加工处理，将以占位符表示的配置解析为最终的实际值，这样半成品的 `BeanDefinition` 就成为成品的 `BeanDefinition`。

**提示**

Spring 框架源码类包层次结构清晰，但包名很长。在 IDE 环境下，这不是问题，但却给书面表达带来了困难。为了行文方便，常常将 Spring 类的包名省略。如果用户希望了解类位于哪个具体的包中，在 IDEA 中，可以按 Ctrl+N 组合键，输入类名，IDEA 将自动查找出这个类。推荐另一个小工具：Search and Replace，它被称为文本搜索工具中的“倚天剑”。它能深入到 ZIP、JAR 等类型的文件中进行搜索，用户可以通过这个工具轻易地找出类所在的位置。

6.1.3 InstantiationStrategy

`org.springframework.beans.factory.support.InstantiationStrategy` 负责根据 BeanDefinition 对象创建一个 Bean 实例。Spring 之所以将实例化 Bean 的工作通过一个策略接口进行描述，是为了可以方便地采用不同的实例化策略，以满足不同的应用需求，如通过 CGLib 类库为 Bean 动态创建子类再进行实例化。`InstantiationStrategy` 类的继承结构如图 6-3 所示。

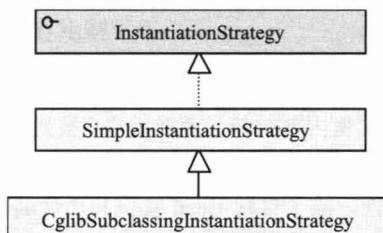


图 6-3 InstantiationStrategy 类继承结构

`SimpleInstantiationStrategy` 是最常用的实例化策略，该策略利用 Bean 实现类的默认构造函数、带参构造函数或工厂方法创建 Bean 的实例。

`CglibSubclassingInstantiationStrategy` 扩展了 `SimpleInstantiationStrategy`，为需要进行方法注入的 Bean 提供了支持。它利用 CGLib 类库为 Bean 动态生成子类，在子类中生成方法注入的逻辑，然后使用这个动态生成的子类创建 Bean 的实例。

`InstantiationStrategy` 仅负责实例化 Bean 的操作，相当于执行 Java 语言中 new 的功能，它并不会参与 Bean 属性的设置工作。所以由 `InstantiationStrategy` 返回的 Bean 实例实际上是一个半成品的 Bean 实例，属性填充的工作留待 `BeanWrapper` 来完成。

6.1.4 BeanWrapper

`org.springframework.beans.BeanWrapper` 是 Spring 框架中重要的组件类。`BeanWrapper` 相当于一个代理器，Spring 委托 `BeanWrapper` 完成 Bean 属性的填充工作。在 Bean 实例被 `InstantiationStrategy` 创建出来之后，容器主控程序将 Bean 实例通过 `BeanWrapper` 包

装起来，这是通过调用 BeanWrapper#setWrappedInstance(Object obj)方法完成的。BeanWrapper 类的继承结构如图 6-4 所示。

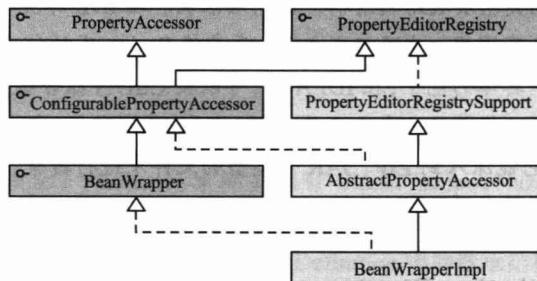


图 6-4 BeanWrapper 类继承结构

通过图 6-4 可以看出，BeanWrapper 还有两个顶级类接口，分别是 PropertyAccessor 和 PropertyEditorRegistry。PropertyAccessor 接口定义了各种访问 Bean 属性的方法，如 setPropertyValue(String, Object)、setPropertyValues(PropertyValues pvs)等；而 PropertyEditorRegistry 是属性编辑器的注册表。所以 BeanWrapper 实现类 BeanWrapperImpl 具有三重身份：

- (1) Bean 包裹器。
- (2) 属性访问器。
- (3) 属性编辑器注册表。

一个 BeanWrapperImpl 实例内部封装了两类组件：被封装的待处理的 Bean，以及一套用于设置 Bean 属性的属性编辑器。

要顺利地填充 Bean 属性，除了目标 Bean 实例和属性编辑器外，还需要获取 Bean 对应的 BeanDefinition，它从 Spring 容器的 BeanDefinitionRegistry 中直接获取。Spring 主控程序从 BeanDefinition 中获取 Bean 属性的配置信息 PropertyValue，并使用属性编辑器对 PropertyValue 进行转换以得到 Bean 的属性值。对 Bean 的其他属性重复这样的步骤，就可以完成 Bean 所有属性的注入工作。BeanWrapperImpl 在内部使用 Spring 的 BeanUtils 工具类对 Bean 进行反射操作，设置属性。下一节将详细介绍属性编辑器的原理，并讲解如何通过配置的方式注册自定义的属性编辑器。

6.2 属性编辑器

在 Spring 配置文件里，往往通过字面值为 Bean 各种类型的属性提供设置值：不管是 double 类型还是 int 类型，在配置文件中都对应字符串类型的字面值。BeanWrapper 在填充 Bean 属性时如何将这个字面值正确地转换为对应的 double 或 int 等内部类型呢？我们可以隐约地感觉到一定有一个转换器在“暗中相助”，这个转换器就是属性编辑器。

“属性编辑器”这个名字可能会让人误以为是一个带用户界面的输入器，其实属性编辑器不一定非得有用户界面，任何实现 java.beans.PropertyEditor 接口的类都是属性编

辑器。属性编辑器的主要功能就是将外部的设置值转换为 JVM 内部的对应类型，所以属性编辑器其实就是一个类型转换器。

PropertyEditor 是 JavaBean 规范定义的接口，JavaBean 规范中还有其他一些 PropertyEditor 配置的接口。为了彻底地理解属性编辑器，必须对 JavaBean 中有关属性编辑器的规范进行学习，相信这些知识对学习和掌握 Spring 中的属性编辑器会大有帮助。

6.2.1 JavaBean 的编辑器

Sun 所制定的 JavaBean 规范很大程度上是为 IDE 准备的——它让 IDE 能够以可视化的方式设置 JavaBean 的属性。如果在 IDE 中开发一个可视化的应用程序，则需要通过属性设置的方式对组成应用的各种组件进行定制，IDE 通过属性编辑器让开发人员使用可视化的方式设置组件的属性。

一般的 IDE 都支持 JavaBean 规范所定义的属性编辑器，当组件开发商发布一个组件时，它往往将组件对应的属性编辑器捆绑发行，这样开发者就可以在 IDE 环境下方便地利用属性编辑器对组件进行定制工作。

JavaBean 规范通过 `java.beans.PropertyEditor` 定义了设置 JavaBean 属性的方法，通过 `BeanInfo` 描述了 JavaBean 的哪些属性是可定制的，此外还描述了可定制属性与 `PropertyEditor` 的对应关系。

`BeanInfo` 与 JavaBean 之间的对应关系通过二者之间的命名规范确立。对应 JavaBean 的 `BeanInfo` 采用如下的命名规范：`<Bean>BeanInfo`。如 `ChartBean` 对应的 `BeanInfo` 为 `ChartBeanBeanInfo`；`Car` 对应的 `BeanInfo` 为 `CarBeanInfo`。当 JavaBean 连同其属性编辑器一同注册到 IDE 中后，在开发界面中对 JavaBean 进行定制时，IDE 就会根据 JavaBean 规范找到对应的 `BeanInfo`，再根据 `BeanInfo` 中的描述信息找到 JavaBean 属性描述（是否开放、使用哪个属性编辑器），进而为 JavaBean 生成特定的开发编辑界面。

JavaBean 规范提供了一个管理默认属性编辑器的管理器 `PropertyEditorManager`，该管理器内保存着一些常见类型的属性编辑器。如果某个 JavaBean 的常见类型属性没有通过 `BeanInfo` 显式指定属性编辑器，那么 IDE 将自动使用 `PropertyEditorManager` 中注册的对应默认属性编辑器。

由于 JavaBean 对应的属性编辑器等 IDE 环境相关的资源和组件需要动态加载，所以在纯 Java 的 IDE 中开发基于组件的应用时，总会感觉 IDE 反应很迟钝，不像 Delphi、C++Builder 一样灵敏快捷。但在 IDEA 开发环境中，设计包括可视化组件的应用时却很快捷，原因是 IDEA 没有使用 Java 的标准用户界面组件库，当然也就没有按照 JavaBean 的规范开发设计 GUI 组件。

1. PropertyEditor

`PropertyEditor` 是属性编辑器的接口，它规定了将外部设置值转换为内部 JavaBean

属性值的转换接口方法。PropertyEditor 主要的接口方法说明如下。

- Object getValue(): 返回属性的当前值，基本类型被封装成对应的封装类实例。
- void setValue(Object newValue): 设置属性的值，基本类型以封装类传入。
- String getAsText(): 将属性对象用一个字符串表示，以便外部的属性编辑器能以可视化的方式显示。默认返回 null，表示该属性不能以字符串表示。
- void setAsText(String text): 用一个字符串去更新属性的内部值，这个字符串一般从外部属性编辑器传入。
- String[] getTags(): 返回表示有效属性值的字符串数组（如 boolean 属性对应的有效 Tag 为 true 和 false），以便属性编辑器能以下拉框的方式显示出来。默认返回 null，表示属性没有匹配的字符值有限集合。
- String getJavaInitializationString(): 为属性提供一个表示初始值的字符串，属性编辑器以此值作为属性的默认值。

可以看出，PropertyEditor 接口方法是内部属性值和外部设置值的沟通桥梁。此外，可以很容易地发现该接口的很多方法是专为 IDE 中的可视化属性编辑器提供的，如 getTags()、getJavaInitializationString() 及另外一些未曾介绍的接口方法。

Java 为 PropertyEditor 提供了一个方便类 PropertyEditorSupport，该类实现了 PropertyEditor 接口并提供了默认实现。一般情况下，用户可以通过扩展这个方便类设计自己的属性编辑器。

2. BeanInfo

BeanInfo 主要描述了 JavaBean 的哪些属性可以编辑及对应的属性编辑器，每个属性对应一个属性描述器 PropertyDescriptor。PropertyDescriptor 的构造函数有两个入参：PropertyDescriptor(String propertyName, Class beanClass)，其中 propertyName 为属性名，beanClass 为 JavaBean 对应的 Class。此外，PropertyDescriptor 还有一个 setPropertyEditorClass (Class propertyEditorClass) 方法，用于为 JavaBean 属性指定编辑器。BeanInfo 接口最重要的方法就是 PropertyDescriptor[] getPropertyDescriptors()，该方法返回 JavaBean 的属性描述器数组。

BeanInfo 接口有一个常用的实现类 SimpleBeanInfo，一般情况下，可以通过扩展 SimpleBeanInfo 实现自己的功能。

3. 一个实例

下面来看一个具体的属性编辑器实例，该实例根据 *Core Java II* 中的一个例子改编而成。

ChartBean 是一个可定制图表组件，允许通过属性设置定制图表的样式，以得到满足各种不同使用场合要求的图表。我们忽略 ChartBean 的其他属性，仅关注其中的两个属性，如代码清单 6-2 所示。

代码清单 6-2 ChartBean

```
public class ChartBean extends JPanel{
    private int titlePosition = CENTER;
    private boolean inverse;
    //省略get/setter方法
}
```

下面为 titlePosition 属性提供一个属性编辑器。我们不去直接实现 PropertyEditor，而是通过扩展 PropertyEditorSupport 这个方便类来定义属性编辑器，如代码清单 6-3 所示。

代码清单 6-3 TitlePositionEditor

```
import java.beans.*;
public class TitlePositionEditor extends PropertyEditorSupport{
    private String[] options = { "Left", "Center", "Right" };

    //①代表可选属性值的字符串标识数组
    public String[] getTags() { return options; }

    //②代表属性初始值的字符串
    public String getJavaInitializationString() { return "" + getValue(); }

    //③将内部属性值转换为对应的字符串表示形式，供属性编辑器显示之用
    public String getAsText(){
        int value = (Integer) getValue();
        return options[value];
    }

    //④将外部设置的字符串转换为内部属性的值
    public void setAsText(String s){
        for (int i = 0; i < options.length; i++){
            if (options[i].equals(s)){
                setValue(i);
                return;
            }
        }
    }
}
```

①处通过 getTags() 方法返回一个字符串数组，因此在 IDE 中该属性对应的编辑器将自动提供一个下拉框，下拉框中包含 3 个可选项：“Left”、“Center”、“Right”。而③和④处的两个方法分别完成属性值到字符串的双向转换功能。ChartBean 的 inverse 属性也有一个相似的编辑器 InverseEditor，我们忽略不讲。

下面编写 ChartBean 对应的 BeanInfo。根据 JavaBean 的命名规范，这个 BeanInfo 应该命名为 ChartBeanBeanInfo，它负责将属性编辑器和 ChartBean 的属性联系起来，如代码清单 6-4 所示。

代码清单 6-4 ChartBeanBeanInfo

```
import java.beans.*;
public class ChartBeanBeanInfo extends SimpleBeanInfo{
```

```

public PropertyDescriptor[] getPropertyDescriptors()
{
    try{
        PropertyDescriptor titlePositionDescriptor
            = new PropertyDescriptor("titlePosition", ChartBean.class);
        titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class);
        //① 将 TitlePositionEditor 绑定到
        //ChartBean 的 titlePosition 属性中
        PropertyDescriptor inverseDescriptor
            = new PropertyDescriptor("inverse", ChartBean.class);
        inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
        return new PropertyDescriptor[]{titlePositionDescriptor, inverseDescriptor};
    }
    catch (IntrospectionException e){
        e.printStackTrace();
        return null;
    }
}
}

```

在 ChartBeanBeanInfo 中分别为 ChartBean 的 titlePosition 和 inverse 属性指定对应的属性编辑器。将 ChartBean 连同属性编辑器及 ChartBeanBeanInfo 打成 JAR 包，使用 IDE 组件扩展管理功能注册到 IDE 中。这样就可以像使用 TextField、Checkbox 等组件一样对 ChartBean 进行可视化的开发设计工作了。下面是 ChartBean 在 NetBeans IDE 中的属性编辑器效果图，如图 6-5 所示。

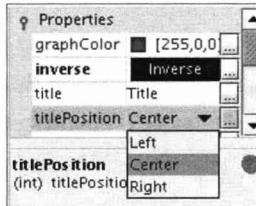


图 6-5 ChartBean 的属性编辑器

ChartBean 可设置的属性都列在属性查看器中。在单击 titlePosition 属性时，下拉框中列出了我们提供的 3 个选项。

6.2.2 Spring 默认属性编辑器

Spring 的属性编辑器和传统的用于 IDE 开发时的属性编辑器不同，它们没有 UI 界面，仅负责将配置文件中的文本配置值转换为 Bean 属性的对应值，所以 Spring 的属性编辑器并非传统意义上的 JavaBean 属性编辑器。

Spring 为常见的属性类型提供了默认的属性编辑器。从图 6-4 中可以看出，BeanWrapperImpl 类扩展了 PropertyEditorRegistrySupport 类，Spring 在 PropertyEditor RegistrySupport 中为常见属性类型提供了默认的属性编辑器，这些“常见的类型”共 32 个，可分为三大类，总结如表 6-1 所示。

表 6-1 Spring 提供的默认属性编辑器

类别	说明
基础数据类型	分为几个小类： (1) 基本数据类型，如 boolean、byte、short、int 等。 (2) 基本数据类型封装类，如 Long、Character、Integer 等。 (3) 两个基本数据类型的数组，即 char[] 和 byte[]。 (4) 大数类，即 BigDecimal 和 BigInteger
集合类	为 5 种类型的集合类 Collection、Set、SortedSet、List 和 SortedMap 提供了编辑器
资源类	用于访问外部资源的 8 个常见类，即 Class、Class[]、File、InputStream、Locale、Properties、Resource[] 和 URL

PropertyEditorRegistrySupport 中有两个用于保存属性编辑器的 Map 类型变量。

- defaultEditors：用于保存默认属性类型的编辑器，元素的键为属性类型，值为对应的属性编辑器实例。
- customEditors：用于保存用户自定义的属性编辑器，元素的键值和 defaultEditors 相同。

PropertyEditorRegistrySupport 通过类似以下的代码定义默认属性编辑器：

```
defaultEditors.put(char.class, new CharacterEditor(false));
defaultEditors.put(Character.class, new CharacterEditor(true));
defaultEditors.put(Locale.class, new LocaleEditor());
defaultEditors.put(Properties.class, new PropertiesEditor());
```

这些默认的属性编辑器用于解决常见属性类型的注册问题。如果用户的应用包括一些特殊类型的属性，且希望在配置文件中以字面值提供配置，那么就需要编写自定义属性编辑器并注册到 Spring 容器中。这样，Spring 才能将配置文件中的属性配置值转换为对应的属性类型值。

6.2.3 自定义属性编辑器

Spring 的大部分默认属性编辑器都直接扩展于 java.beans.PropertyEditorSupport 类，开发者也可以通过扩展 PropertyEditorSupport 实现自己的属性编辑器。相对于用于 IDE 环境的属性编辑器来说，Spring 环境下使用的属性编辑器的功能非常单一，仅需将配置文件中的字面值转换为属性类型的对象即可，并不需要提供 UI 界面，因此仅需简单覆盖 PropertyEditorSupport 的 setAsText()方法即可。

1. 一个实例

继续使用第 5 章中 Boss 和 Car 的例子。假设现在希望在配置 Boss 时不通过引用 Bean 的方式注入 Boss 的 car 属性，而希望直接通过字面值提供配置。为了方便阅读，这里再次列出 Car 和 Boss 类的简要代码，如代码清单 6-5 和 6-6 所示。

代码清单 6-5 Car

```
package com.smart.editor;
public class Car {
    private int maxSpeed;
    public String brand;
    private double price;
    //省略get/setter
}
```

代码清单 6-6 Boss

```
package com.smart.editor;
public class Boss {
    private String name;
    private Car car = new Car();
    //省略get/setter
}
```

Boss 有两个属性：name 和 car，分别对应 String 类型和 Car 类型。Spring 拥有 String 类型的默认属性编辑器，因此，我们无须关心 String 类型的属性。但 Car 类型是我们自定义的类型，要配置 Boss 的 car 属性，有两种方案。

(1) 在配置文件中为 car 专门配置一个<bean>，然后在 Boss 的<bean>中通过 ref 引用 car Bean，这正是第 5 章中所用的方法。

(2) 为 Car 类型提供一个自定义的属性编辑器，这样就可以通过字面值为 Boss 的 car 属性提供配置值。

第一种方案是常用的方法，但在某些情况下，这种方式需要将属性对象一步步肢解为最终可以用基本类型配置的 Bean，使配置文件变得不够清晰。直接为属性类提供一个对应的自定义属性编辑器可能会是更好的替代方案。

现在来为 Car 编写一个自定义的属性编辑器，如代码清单 6-7 所示。

代码清单 6-7 CustomCarEditor

```
package com.smart.editor;
import java.beans.PropertyEditorSupport;

public class CustomCarEditor extends PropertyEditorSupport {

    //①将字面值转换为属性类型对象
    public void setAsText(String text) {
        if(text == null || text.indexOf(",") == -1) {
            throw new IllegalArgumentException("设置的字符串格式不正确");
        }
        String[] infos = text.split(",");
        Car car = new Car();
        car.setBrand(infos[0]);
        car.setMaxSpeed(Integer.parseInt(infos[1]));
        car.setPrice(Double.parseDouble(infos[2]));
    }

    //②调用父类的setValue()方法设置转换后的属性对象
    setValue(car);
}
```

CustomCarEditor 很简单，它仅覆盖 PropertyEditorSupport 便利类的 setAsText(String text)方法，该方法负责将配置文件以字符串提供的字面值转换为 Car 对象。字面值采用逗号分隔格式字符串的同时为 brand、maxSpeed 和 price 属性值提供设置值，setAsText()方法解析这个字面值并生成对应的 Car 对象。由于并不需要将 Boss 内部的 car 属性回显到属性编辑器的 UI 界面中，因此不需要覆盖 getAsText()方法。

2. 注册自定义的属性编辑器

在 IDE 环境下，自定义属性编辑器在使用之前必须通过扩展组件功能进行注册，在 Spring 环境下也需要通过一定的方法注册自定义的属性编辑器。

如果使用 BeanFactory，则用户需要手工调用 registerCustomEditor(Class requiredType, PropertyEditor propertyEditor) 方法注册自定义的属性编辑器；如果使用 ApplicationContext，则只需在配置文件中通过 CustomEditorConfigurer 注册即可。CustomEditorConfigurer 实现了 BeanFactoryPostProcessor 接口，因而是一个 Bean 工厂后处理器。我们知道，Bean 工厂后处理器在 Spring 容器中加载配置文件并生成 BeanDefinition 半成品后就会被自动执行。因此，CustomEditorConfigurer 在容器启动时有机会注入自定义的属性编辑器。下面的配置片段定义了一个 CustomEditorConfigurer：

```
<!--①配置自动注册属性编辑器的CustomEditorConfigurer -->
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <!--② 属性编辑器对应的属性类型-->
            <entry key="com.smart.editor.Car"
                   value="com.smart.editor.CustomCarEditor"/>

            </entry>
        </map>
    </property>
</bean>

<bean id="boss" class="com.smart.editor.Boss">
    <property name="name" value="John"/>
    <!--③该属性将使用②处的属性编辑器完成属性填充操作-->
    <property name="car" value="红旗CA72,200,20000.00"/>
</bean>
```

在①处定义了用于注册自定义属性编辑器的 CustomEditorConfigurer，Spring 容器将通过反射机制自动调用这个 Bean。CustomEditorConfigurer 通过一个 Map 属性定义需要自动注册的自定义属性编辑器。在②处为 Car 类型指定了对应的属性编辑器 CustomCarEditor，其中键是属性类型，值是属性编辑器的类名。

最精彩的当然是③处的配置。原来通过一个<bean>元素标签配置好 car Bean，然后在 Boss 的<bean>中通过 ref 引用 car Bean，而现在直接通过 value 为 car 属性提供配置。BeanWrapper 在设置 Boss 的 car 属性时，将检索自定义属性编辑器的注册表，当发现 Car 属性类型拥有对应的属性编辑器 CustomCarEditor 时，就会利用 CustomCarEditor 将“红旗 CA72,200,20000.00”转换为 Car 对象。

**提示**

按照 JavaBean 的规范，JavaBean 的基础设施会在 JavaBean 的相同类包下查找是否存在<JavaBean>Editor 的规范类。如果存在，则自动使用<JavaBean>Editor 作为该 JavaBean 的 PropertyEditor。

如 com.smart.domain.UserEditor 会自动成为 com.smart.domain.User 对应的 PropertyEditor。Spring 也支持这个规范，如果采用这种规约命令 PropertyEditor，就无须显式在 CustomEditorConfigurer 中注册了，Spring 将自动查找并注册这个 PropertyEditor。

6.3 使用外部属性文件

在进行数据源或邮件服务器等资源的配置时，用户可以直接在 Spring 配置文件中配置用户名/密码、链接地址等信息。但一种更好的做法是将这些配置信息独立到一个外部属性文件中，并在 Spring 配置文件中通过形如 \${user}、\${password} 的占位符引用属性文件中的属性项。这种配置方式拥有两个明显的好处。

- 减少维护的工作量：资源的配置信息可以被多个应用共享，在多个应用使用同一资源的情况下，如果资源用户名/密码、链接地址等配置信息发生更改，则用户只需调整独立的属性文件即可。
- 使部署更简单：Spring 配置文件主要描述应用工程中的 Bean，这些配置信息在开发完成后就基本固定下来了。但数据源、邮件服务器等资源配置信息却需要在部署时根据现场情况确定。如果通过一个独立的属性文件存放这些配置信息，则部署人员只需调整这个属性文件即可，根本不需要关注结构复杂、信息量大的 Spring 配置文件。这不仅给部署和维护带来了方便，也降低了出错的概率。

Spring 提供了一个 PropertyPlaceholderConfigurer，它能够使 Bean 在配置时引用外部属性文件。PropertyPlaceholderConfigurer 实现了 BeanFactoryPostProcessorBean 接口，因而也是一个 Bean 工厂后处理器。

6.3.1 PropertyPlaceholderConfigurer 属性文件

1. 使用 PropertyPlaceholderConfigurer 属性文件

回忆一下在 2.3.4 节中定义的数据源，如下：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost:3306/sampledb"
    p:userName="root"
    p:password="123456"/>
```

驱动器类名、JDBC 的 URL 地址及数据库用户名/密码都直接写在 XML 文件中，部

署人员在部署应用时，必须先找出这个 Bean 配置 XML 文件，再找出数据源 Bean 定义的代码段进行调整，给部署工作带来了很大的困难。

根据实际应用中的最佳实战，可以将这些需要调整的配置信息抽取到一个配置文件中。这里使用一个名为 `jdbc.properties` 的配置文件，如代码清单 6-8 所示。

代码清单 6-8 `jdbc.properties`

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/sampledb
userName=root
password=123456
```

属性文件可以定义多个属性，每个属性都由一个属性名和一个属性值组成，二者用“`=`”隔开。下面通过 `PropertyPlaceholderConfigurer` 引入 `jdbc.properties` 属性文件，调整数据源 Bean 的配置。

```
<!--①引入jdbc.properties属性文件-->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
    p:location="classpath:com/smart/placeholder/jdbc.properties"
    p:fileEncoding="utf-8"/>

<!--②通过属性名引用属性值-->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${driverClassName}"
    p:url="${url}"
    p:username="${userName}"
    p:password="${password}"/>
```

在①处通过 `PropertyPlaceholderConfigurer` 的 `location` 属性引入属性文件，这样，在 Bean 定义的时候就可以引用属性文件中的属性了，如②处的粗体部分所示。通过这样的调整后，部署人员在部署本应用时，仅需关注 `jdbc.properties` 这个配置文件即可，无须关心 Spring 的配置文件。

2. PropertyPlaceholderConfigurer 的其他属性

除了 `location` 属性外，`PropertyPlaceholderConfigurer` 还有一些常用的属性，在一些高级应用中可能会用到。

- `locations`: 如果只有一个属性文件，则直接使用 `location` 属性指定就可以了；如果有多个属性文件，则可以通过 `locations` 属性进行设置。可以像配置 `List` 一样配置 `locations` 属性，参见 5.4.6 节。
- `fileEncoding`: 属性文件的编码格式。Spring 使用操作系统默认编码读取属性文件。如果属性文件采用了特殊编码，则需要通过该属性显式指定。
- `order`: 如果配置文件中定义了多个 `PropertyPlaceholderConfigurer`，则通过该属性指定优先顺序。
- `placeholderPrefix`: 在上面的例子中，通过 `${属性名}`引用属性文件中的属性项，其中“ `${}`”为默认的占位符前缀，可以根据需要改为其他的前缀符。
- `placeholderSuffix`: 占位符后缀，默认为“ `}`”。

3. 使用<context:property-placeholder>引用属性文件

可以使用 context 命名空间定义属性文件，相比于传统的 PropertyPlaceholderConfigurer 配置，这种方法更加优雅。

```
<context:property-placeholder
    location="classpath:com/smart/placeholder/jdbc.properties" />
```

以上配置就相当于在 Spring 容器中定义了一个 PropertyPlaceholderConfigurer 的 Bean。虽然这种方式比较优雅，但如果希望自定义一些额外的高级功能，如属性加密、使用数据库表来保存配置信息等，则必须使用扩展 PropertyPlaceholderConfigurer 的类并使用 Bean 的配置方式（参见 6.3.2 节）。

4. 在基于注解及基于 Java 类的配置中引用属性

在基于 XML 的配置文件中，通过“\${propName}”形式引用属性值。类似的，基于注解配置的 Bean 可以通过@Value 注解为 Bean 的成员变量或方法入参自动注入容器已有的属性，如下：

```
package com.smart.placeholder;
import org.apache.commons.lang.builder.ToStringBuilder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class MyDataSource {

    @Value("${driverClassName}")
    private String driverClassName;

    @Value("${url}")
    private String url;

    @Value("${userName}")
    private String userName;

    @Value("${password}")
    private String password;

    public String toString(){
        return ToStringBuilder.reflectionToString(this);
    }
}
```

@Value 注解可以为 Bean 注入一个字面值，也可以通过@Value("\${propName}")的形式根据属性名注入属性值。由于标注@Configuration 的类本身就相当标注了@Component，所以在标注@Configuration 的类中引用属性的方式和基于注解配置的引用方式是完全一样的，此处不再赘述。

使用@Value 注解来引用属性值带来很大的便利，但在使用过程中，一定要确保所引用的属性值在属性文件中已经存在且数值匹配，否则会造成 Bean 创建错误，引发意想不到的异常。

6.3.2 使用加密的属性文件

对于那些不敏感的属性信息，以明文形式出现在属性文件中是合适的。但如果属性信息是数据库用户名/密码等敏感信息，一般情况下则希望以密文的方式保存。虽然 Web 应用系统的客户端用户看不到服务器端的属性文件，但允许登录到 Web 应用系统所在服务器的内部人员却可以轻易查看到属性文件的内容。如果属性文件以明文形式保存着访问数据库的用户名/密码等信息，那么任何拥有服务器登录权限的人都可能查看到这些机密信息，容易造成数据库访问权限的泄露。

对于一些对安全要求特别高的应用系统（如电信、银行、公安的重点人口库等）来说，这些敏感信息应该只掌握在少数特定维护人员的手中，而不是毫无保留地对所有可以进入部署机器的人员开放。这就要求对应用程序配置文件的某些属性进行加密，让 Spring 容器在读取属性文件后，在内存中对属性进行解密，然后再将解密后的属性值赋给目标对象。

`PropertyPlaceholderConfigurer` 继承自 `PropertyResourceConfigurer` 类，后者有几个有用的 `protected` 方法，用于在属性使用之前对属性列表中的属性进行转换。

- ❑ `void convertProperties(Properties props)`: 属性文件中的所有属性值都封装在 `props` 中，覆盖此方法，可以对所有的属性值进行转换处理。
- ❑ `String convertProperty(String propertyName, String PropertyValue)`: 在加载属性文件并读取文件中的每个属性时，都会调用此方法进行转换处理。
- ❑ `String convertPropertyValue(String originalValue)`: 和上一个方法类似，只不过没有传入属性名。

在默认情况下，这 3 个方法内部都是空的，即不会对属性值进行任何转换。可以扩展 `PropertyPlaceholderConfigurer`，覆盖相应的属性转换方法，就可以支持加密版的属性文件了。

1. DES 加密解密工具类

信息的加密可分为对称和非对称两种方式，前者表示加密后的信息可以解密成原值，而后者则不能根据加密后的信息还原为原值。MD5 属于非对称加密，而 DES 属于对称加密。先用 DES 对属性值进行加密，在读取到属性值时，再用 DES 进行解密。下面是支持 DES 加密解密的工具类，如代码清单 6-9 所示。

代码清单 6-9 DESUtils.java：DES加密解密工具类

```
package com.smart.placeholder;

import java.security.Key;
import java.security.SecureRandom;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import sun.misc.BASE64Decoder;
import sun.misc.BASE64Encoder;
```

```

public class DESUtils {

    //①指定DES加密解密所用的密钥
    private static Key key;
    private static String KEY_STR = "myKey";
    static {
        try {
            KeyGenerator generator = KeyGenerator.getInstance("DES");
            generator.init(new SecureRandom(KEY_STR.getBytes()));
            key = generator.generateKey();
            generator = null;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    //②对字符串进行DES加密，返回BASE64编码的加密字符串
    public static String getEncryptString(String str) {
        BASE64Encoder base64en = new BASE64Encoder();
        try {
            byte[] strBytes = str.getBytes("UTF8");
            Cipher cipher = Cipher.getInstance("DES");
            cipher.init(Cipher.ENCRYPT_MODE, key);
            byte[] encryptStrBytes = cipher.doFinal(strBytes);
            return base64en.encode(encryptStrBytes);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    //③对BASE64编码的加密字符串进行解密，返回解密后的字符串
    public static String getDecryptString(String str) {
        BASE64Decoder base64De = new BASE64Decoder();
        try {
            byte[] strBytes = base64De.decodeBuffer(str);
            Cipher cipher = Cipher.getInstance("DES");
            cipher.init(Cipher.DECRYPT_MODE, key);
            byte[] decryptStrBytes = cipher.doFinal(strBytes);
            return new String(decryptStrBytes, "UTF8");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    //④对入参的字符串进行加密，打印出加密后的串
    public static void main(String[] args) throws Exception {
        if (args == null || args.length < 1) {
            System.out.println("请输入要加密的字符，用空格分隔。");
        } else {
            for (String arg : args) {
                System.out.println(arg + ":" + getEncryptString(arg));
            }
        }
    }
}

```

} } }

上述示例使用 DES 算法打造加密解密工具类。DES 加密解密的关键是加密密钥，在①处设置了一个值为“myKey”的加密密钥。②处的方法可以获取明文字符串的加密串，它以 BASE64 进行编码。BASE64 编码是以大小写字母、数字及其他几个字符组成的编码串，给书写表达带来了方便。③处的方法可以将加密后的字符串解密出来。④处的方法可以将 main()方法入参的值加密并打印出来。

2. 使用密文版的属性文件

可以通过 DESUtils 类对数据库用户名和密码进行加密。在编译好 DESUtils 工具类后，在 DOS 窗口下通过命令获取用户名及密码的密文，如图 6-6 所示。

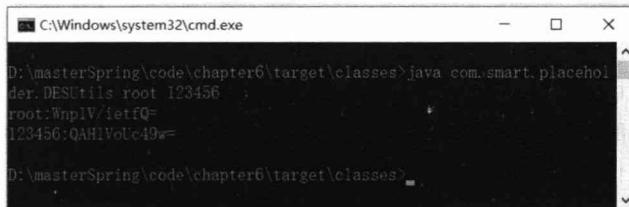


图 6-6 使用 DESUtils 通过命令方式对信息进行加密

在 DOS 窗口下输入 `java com.smart.placeholder.DESUtils root 123456` 命令，`DESUtils` 将使用 DES 对 `root` 及 `123456` 进行加密并返回加密后的密文。用此密文更改 `jdbc.properties` 的相关属性值，如下：

```
driverClassName=com.mysql.jdbc.Driver  
url=jdbc:mysql://localhost:3306/sampledb  
userName=WnplV/ietfQ=  
password=QAH1VoUc49w=
```

`PropertyPlaceholderConfigurer` 本身不支持密文版的属性文件，不过我们扩展该类，覆盖 `String convertProperty(String propertyName, String PropertyValue)` 方法，对 `userName` 及 `password` 的属性值进行解密转换，如代码清单 6-10 所示。

代码清单 6-10 EncryptPropertyPlaceholderConfigurer.java

```
package com.smart.placeholder;
import org.springframework.beans.factory.config.PropertyPlaceholderConfigurer;

//①继承PropertyPlaceholderConfigurer定义支持密文版属性的属性配置器
public class EncryptPropertyPlaceholderConfigurer extends
PropertyPlaceholderConfigurer {
    private String[] encryptPropNames = {"userName", "password"};

    //②对特定属性的属性值进行转换
    @Override
    protected String convertProperty(String propertyName, String PropertyValue)

        if(isEncryptProp(propertyName)) {
```

```

        String decryptValue = DESUtils.getDecryptString(propertyValue);
        System.out.println(decryptValue);
        return decryptValue;
    }else{
        return propertyValue;
    }
}

//⑧判断是否是需要解密的属性
private boolean isEncryptProp(String propertyName){
    for(String encryptPropName:encryptPropNames){
        if(encryptPropName.equals(propertyName)){
            return true;
        }
    }
    return false;
}
}

```

EncryptPropertyPlaceholderConfigurer 使用前面的 DESUtils 类，对已经加密的属性进行解密转换，以便属性的最终读者可以获取解密版的内容。

使用自定义的属性加载器后，就无法使用<context:property-placeholder>引用属性文件了，必须通过传统的配置方式引用加密版的属性文件。

```

<bean class="com.smart.placeholder.EncryptPropertyPlaceholderConfigurer"
      p:location="classpath:com/smart	placeholder/jdbc.properties"
      p:fileEncoding="utf-8"/>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${driverClassName}"
      p:url="${url}"
      p:username="${userName}"
      p:password="${password}" />

```

属性文件加载器的实现类改为 EncryptPropertyPlaceholderConfigurer，其他配置和原来的相同。

6.3.3 属性文件自身的引用

Spring 既允许在 Bean 定义中通过\${propName}引用属性值，也允许在属性文件中使用\${propName}实现属性之间的相互引用。

```

dbName=sampledB
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/${dbName}

```

在上面的属性文件定义中，url 属性的值通过\${dbName}引用了另一个属性的值。对于一些复杂的属性，可以通过这种方式将属性变化的部分抽取出，实现配置的最小化。

**提示**

如果一个属性值太长，一行写不下，则可以通过在行后添加“\”将属性值划分为多行，如：

```
desc=desc content desc content desc content\
desc content desc content
```

6.4 引用 Bean 的属性值

将应用系统的配置信息放在配置文件中并非总是最适合的。如果应用系统是以集群方式部署的，或者希望在运行期动态调整应用系统的某些配置，这时，将配置信息放到数据库中不但方便集中管理，而且可以通过应用系统的管理界面动态维护，有效增强应用系统的可维护性。

在早期版本中，要在配置文件中使 Bean 引用另一个 Bean 的属性值是比较麻烦的，Spring 3.0 则提供了优雅的解决方案。在 Spring 3.0 中，可以通过类似#{beanName.beanProp} 的方式方便地引用另一个 Bean 的值，如代码清单 6-11 所示。

代码清单 6-11 SysConfig.java：提供配置信息的类

```
package com.smart.beanprop;

public class SysConfig {
    private int sessionTimeout;
    private int maxTabPageNum;
    private DataSource dataSource;

    //①模拟从数据库中获取配置值并设置相应的属性
    public void initFromDB() {
        ...
        //模拟从数据库获取配置值
        this.sessionTimeout = 30;
        this.maxTabPageNum = 10;
    }

    public int getSessionTimeout() {
        return sessionTimeout;
    }

    public int getMaxTabPageNum() {
        return maxTabPageNum;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

在①处的方法中可以更改代码，从数据库中获取相应的属性信息。为了简化代码，

仅采用直接设置属性值的方式演示属性引用的过程。

在 XML 配置文件中，先将 SysConfig 定义为一个 Bean，这样在定义数据源时即可通过#{beanName.propName}的方式引用 Bean 的属性值，如代码清单 6-12 所示。

代码清单 6-12 beans.xml：引用Bean的属性值

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/spring-beans-4.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.0.xsd">

    <context:property-placeholder
        location="classpath:com/smarter/placeholder/jdbc.properties"/>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close"
          p:driverClassName="${driverClassName}"
          p:url="${url}"
          p:username="${userName}"
          p:password="${password}" />

    <!--① 通过initFromDB方法从数据源中获取配置属性值 -->
    <bean id="sysConfig" class="com.smarter.beanprop.SysConfig"
          init-method="initFromDB"
          p:dataSource-ref="dataSource"/>

    <!--② 引用Bean的属性值-->
    <bean class="com.smarter.beanprop.ApplicationManager"
          p:maxTabPageNum="#{sysConfig.maxTabPageNum}"
          p:sessionTimeout="#{sysConfig.sessionTimeout}"/>
</beans>
```

访问数据库获取配置属性值的前提是连接到数据库中，为此，需要使用外部属性文件配置数据库的连接信息。在生产环境下，一般通过 JNDI 引用应用容器的数据源，此时属性文件仅提供数据源 JNDI 名即可。然后通过 sysConfig 的 initFromDB() 方法访问数据库，获取应用系统的配置信息，并将其保存在 sysConfig 的属性中。

其他需要访问应用系统配置信息的 Bean 即可通过#{beanName.propName}表达式引用 sysConfig Bean 的属性值，如②处所示。

在基于注解和基于 Java 类配置的 Bean 中，可以通过 @Value("#{beanName.propName}") 的注解形式引用 Bean 的属性值。

```
package com.smarter.beanprop;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
@Component
```

```

public class ApplicationManager {

    @Value("#{sysConfig.sessionTimeout}")
    private int sessionTimeout;

    @Value("#{sysConfig.maxTabPageNum}")
    private int maxTabPageNum;
    ...
}

```

6.5 国际化信息

假设我们正在开发一个支持多国语言的 Web 应用程序，要求系统能够根据客户端系统的语言类型返回对应的界面：英文的操作系统返回英文界面，而中文的操作系统则返回中文界面——这便是典型的 i18n 国际化问题。对于有国际化要求的应用系统，我们不能简单地采用硬编码的方式编写用户界面信息、报错信息等内容，而必须为这些需要国际化的信息进行特殊处理。简单来说，就是为每种语言提供一套相应的资源文件，并以规范化命名的方式保存在特定的目录中，由系统自动根据客户端语言选择适合的资源文件。

6.5.1 基础知识

“国际化信息”也称为“本地化信息”，一般需要两个条件才可以确定一个特定类型的本地化信息，分别是“语言类型”和“国家/地区类型”。如中文本地化信息既有中国大陆地区的中文，又有中国台湾、中国香港地区的中文，还有新加坡地区的中文。Java 通过 `java.util.Locale` 类表示一个本地化对象，它允许通过语言参数和国家/地区参数创建一个确定的本地化对象。

语言参数使用 ISO 标准语言代码表示，这些代码是由 ISO-639 标准定义的，每种语言由两位小写字母表示。在许多网站上都可以找到这些代码的完整列表，下面的网址提供了标准语言代码的信息：http://www.loc.gov/standards/iso639-2/php/English_list.php。

国家/地区参数也由标准的 ISO 国家/地区代码表示，这些代码是由 ISO-3166 标准定义的，每个国家/地区由两个大写字母表示。用户可以从以下网址查看 ISO-3166 的标准代码：http://www.iso.org/iso/country_codes。

表 6-2 给出了一些语言和国家/地区的标准代码。

表 6-2 语言和国家/地区代码示例

语 言	代 码	国家/地区代码	代 码
中文	zh	中国大陆	CN
英语	en	中国台湾	TW

续表

语 言	代 码	国家/地区代码	代 码
法语	Fr	中国香港	HK
德语	de	英国	EN
日语	ja	美国	US
韩语	ko	加拿大	CA

1. Locale

java.util.Locale 是表示语言和国家/地区信息的本地化类，它是创建国际化应用的基础。下面给出几个创建本地化对象的示例：

```
//①带有语言和国家/地区信息的本地化对象
Locale locale1 = new Locale("zh", "CN");
//②只有语言信息的本地化对象
Locale locale2 = new Locale("zh");
//③等同于Locale("zh", "CN")
Locale locale3 = Locale.CHINA
//④等同于Locale("zh")
Locale locale4 = Locale.CHINESE
//⑤获取本地系统默认的本地化对象
Locale locale5 = Locale.getDefault()
```

用户既可以同时指定语言和国家/地区参数定义一个本地化对象（见①处），也可以仅通过语言参数定义一个泛本地化对象（见②处）。Locale 类中通过静态常量定义了一些常用的本地化对象，③和④处直接通过引用常量返回本地化对象。此外，用户还可以获取系统默认的本地化对象，如⑤处所示。



提示

在测试时，如果希望改变系统默认的本地化设置，则可以在启动 JVM 时通过命令参数指定：java -Duser.language=en -Duser.region=US MyTest。

2. 本地化工具类

JDK 的 java.util 包中提供了几个支持本地化的格式化操作工具类，如 NumberFormat、DateFormat、MessageFormat。下面分别通过实例了解它们的用法，如代码清单 6-13~6-15 所示。

代码清单 6-13 NumberFormat

```
Locale locale = new Locale("zh", "CN");
NumberFormat currFmt = NumberFormat.getCurrencyInstance(locale);
double amt = 123456.78;
System.out.println(currFmt.format(amt));
```

上面的实例通过 NumberFormat 按本地化的方式对货币金额进行格式化操作。运行实例，输出以下信息：

¥123,456.78

代码清单 6-14 DateFormat

```
Locale locale = new Locale("en", "US");
Date date = new Date();
DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM, locale);
System.out.println(df.format(date));
```

通过 `DateFormat.getDateInstance(int style, Locale locale)` 方法按本地化的方式对日期进行格式化操作。该方法的第一个入参为时间样式，第二个入参为本地化对象。运行以上代码，输出以下信息：

Jan 8, 2007

`MessageFormat` 在 `NumberFormat` 和 `DateFormat` 的基础上提供了强大的占位符字符串的格式化功能，支持时间、货币、数字及对象属性的格式化操作。下面的实例演示了一些常见的格式化功能，如代码清单 6-15 所示。

代码清单 6-15 MessageFormat

```
//①格式化信息串
String pattern1 = "{0}, 你好! 你于{1}在工商银行存入{2} 元。";
String pattern2 = "At {1,time,short} On{1,date,long}, {0} paid {2,number, currency}.";

//②用于动态替换占位符的参数
Object[] params = {"John", new GregorianCalendar().getTime(), 1.0E3};

//③使用默认的本地化对象格式化信息
String msg1 = MessageFormat.format(pattern1, params);

//④使用指定的本地化对象格式化信息
MessageFormat mf = new MessageFormat(pattern2, Locale.US);
String msg2 = mf.format(params);
System.out.println(msg1);
System.out.println(msg2);
```

`pattern1` 是简单形式的格式化信息串，通过`{n}`占位符指定动态参数的替换位置索引，`{0}`表示第一个参数，`{1}`表示第二个参数，以此类推。

`pattern2` 格式化信息串比较复杂一些，除参数位置索引外，还指定了参数的类型和样式。从 `pattern2` 中可以看出格式化信息串的语法是很灵活的，一个参数甚至可以出现在两个地方。如`{1,time,short}`表示从第二个入参中获取时分秒部分的值，显示为短样式时间；而`{1,date,long}`表示从第二个入参中获取日期部分的值，显示为长样式时间。关于 `MessageFormat` 更详细的使用方法，请参见 JDK 的 Javadoc 文档。

在②处定义了用于替换格式化占位符的动态参数，这里用到了 JDK 6.0 自动装包的语法，否则必须采用封装类表示基本类型的参数值。

在③处通过 `MessageFormat` 的 `format()` 方法格式化信息串。它使用了系统默认的本地化对象，由于是中文平台，因此默认为 `Locale.CHINA`。而在④处显式指定了 `MessageFormat` 的本地化对象。

运行上面的代码，输出以下信息：

```
John, 你好! 你于16-3-15 下午5:46在工商银行存入1,000 元。
At 5:46 PM OnMarch 15, 2016, John paid $1,000.00.
```

3. ResourceBoundle

如果应用系统中的某些信息需要支持国际化功能，则必须为期望支持的不同本地化类型分别提供对应的资源文件，并以规范的方式进行命名。国际化资源文件的命名规范规定资源名称采用以下方式进行命名：

```
<资源名>_<语言代码>_<国家/地区代码>.properties
```

其中，语言代码和国家/地区代码都是可选的。`<资源名>.properties` 命名的国际化资源文件是默认的资源文件，即某个本地化类型在系统中找不到对应的资源文件，就采用这个默认的资源文件。`<资源名>_<语言代码>.properties` 命名的国际化资源文件是某一语言默认的资源文件，即某个本地化类型在系统中找不到精确匹配的资源文件，就采用相应语言默认的资源文件。

举个例子：假设资源名为“resource”，语言为英文，国家/地区为美国，则与其对应的本地化资源文件命名为 `resource_en_US.properties`。信息在资源文件以属性名/值的方式表示，如下：

```
greeting.common=How are you!
greeting.morning = Good morning!
greeting.afternoon = Good Afternoon!
```

对应语言为中文、国家/地区为中国大陆的本地化资源文件则命名为 `resource_zh_CN.properties`，资源文件内容如下：

```
greeting.common=\u60a8\u597d\uff01
greeting.morning=\u65e9\u4e0a\u597d\uff01
greeting.afternoon=\u4e0b\u5348\u597d\uff01
```

本地化不同的同一资源文件，虽然属性值各不相同，但属性名却是相同的，这样应用程序就可以通过 `Locale` 对象和属性名精确定位到某个具体的属性值。

读者可能已经注意到，上面中文的本地化资源文件内容采用了特殊的编码形式，这是因为资源文件对文件内容有严格的要求：只能包含 ASCII 字符。所以必须将非 ASCII 字符的内容转换为 Unicode 代码的表示方式。如上面中文的 `resource_zh_CN.properties` 资源文件的 3 个属性值分别是“您好！”、“早上好！”和“下午好！”这 3 个中文字串所对应的 Unicode 代码串。

在应用开发时，直接采用 Unicode 编码编辑资源文件是很不方便的。所以，通常情况下直接使用正常的方式编写资源文件，在测试或部署时再采用工具进行转换。JDK 在 bin 目录下提供了一个完成此项功能的 `native2ascii` 工具，它可以将中文字符的资源文件转换为 Unicode 编码格式的文件，命令格式如下：

```
native2ascii [-reverse] [-encoding 编码] [输入文件 [输出文件]]
```

`resource_zh_CN.properties` 包含中文字符并且以 UTF-8 进行编码。假设将该资源文件放到 d:\ 目录下，通过下面的命令就可以将其转换为 Unicode 编码的形式：

```
D:\>native2ascii -encoding utf-8 d:\resource_zh_CN.properties
D:\resource_zh_CN_1.properties
```

由于原资源文件采用 UTF-8 编码，所以必须显式地通过`-encoding` 指定编码格式。



实战经验

通过 native2ascii 命令手工转换资源文件，不但在操作上不方便，转换后资源文件中的属性内容由于采用了 ASCII 编码，阅读起来也不方便。很多 IDE 开发工具都有属性文件编辑器的插件，插件会自动将资源文件内容转换为 ASCII 形式的编码，同时以正常的方式阅读和编辑资源文件的内容，这给开发和维护工作带来了很大的便利。IntelliJ IDEA 即支持这种透明化编辑资源文件的功能，不过默认未启用，可通过如下方式开启：Setting → Editor → File Encoding → 勾选“Transparent native-to-ascii conversion”复选框；对于 MyEclipse 来说，可以使用 MyEclipse Properties Editor 编辑资源属性文件。

如果应用程序中拥有大量的本地化资源文件，则直接通过传统的 File 操作资源文件显然太过笨拙。Java 提供了用于加载本地化资源文件的方便类 java.util.ResourceBundle。ResourceBundle 为加载及访问资源文件提供了便捷的操作。下面的语句从相对于类路径的目录中加载一个名为 resource 的本地化资源文件：

```
 ResourceBundle rb = ResourceBundle.getBundle("com/smart/i18n/resource", locale)
```

通过以下代码即可访问资源文件的属性值：

```
rb.getString("greeting.common")
```

来看下面的实例，如代码清单 6-16 所示。

代码清单 6-16 ResourceBundle

```
 ResourceBundle rb1 = ResourceBundle.getBundle("com/smart/i18n/resource", Locale.US);
 ResourceBundle rb2 = ResourceBundle.getBundle("com/smart/i18n/resource", Locale.CHINA);
 System.out.println("us:" + rb1.getString("greeting.common"));
 System.out.println("cn:" + rb2.getString("greeting.common"));
```

rb1 加载了对应美国英语本地化的 resource_en_US.properties 资源文件；而 rb2 加载了对应中国大陆中文的 resource_zh_CN.properties 资源文件。运行上面的代码，将输出以下信息：

```
us:How are you!
```

```
cn:您好！
```

在加载资源文件时，如果不指定本地化对象，则将使用本地系统默认的本地化对象。所以，在中文系统中，ResourceBundle.getBundle("com/smart/i18n/resource")语句也将返回和代码清单 6-16 中 rb2 相同的本地化资源。

ResourceBundle 在加载资源时，如果指定的本地化资源文件不存在，则按以下顺序尝试加载其他资源：本地系统默认本地化对象对应的资源→默认的资源。在上面的例子中，假设使用 ResourceBundle.getBundle("com/smart/i18n/resource", Locale.CANADA)加载资源，由于不存在 resource_en_CA.properties 资源文件，它将尝试加载 resource_zh_CN.properties 资源文件。假设 resource_zh_CN.properties 资源文件也不存在，它将继续尝试加载 resource.properties 资源文件。如果这些资源文件都不存在，则将抛出 java.util.MissingResourceException 异常。

4. 在资源文件中使用格式化串

在上面的资源文件中，属性值都是一个普通的字符串，它们不能结合运行时的动态参数构造出灵活的信息，而这种需求是很常见的。要解决这个问题很简单，只需使用带占位符的格式化串作为资源文件的属性值并结合使用 MessageFormat 就可以满足要求。

在上面的例子中，仅向用户提供了一般性问候。下面对资源文件进行改造，通过格式化串让问候语更具个性化，如下：

```
greeting.common=How are you!{0},today is {1}
greeting.morning = Good morning!{0},now is {1 time short}
greeting.afternoon = Good Afternoon!{0} now is {1 date long}
```

将该资源文件保存在 fmt_resource_en_US.properties 中，按照同样的方式编写对应的中文本地化资源文件 fmt_resource_zh_CN.properties。

下面联合使用 ResourceBundle 和 MessageFormat 得到“接地气”的问候语，如代码清单 6-17 所示。

代码清单 6-17 资源文件格式化串处理

```
//①加载本地化资源
ResourceBundle rb1 =
    ResourceBundle.getBundle("com/smart/i18n/fmt_resource", Locale.US);
ResourceBundle rb2 =
    ResourceBundle.getBundle("com/smart/i18n/fmt_resource", Locale.CHINA);
Object[] params = {"John", new GregorianCalendar().getTime()};

用本地化对象进行格式化
②
String str1 = new MessageFormat(rb1.getString("greeting.common"), Locale.US).format(params);
String str2 = new MessageFormat(rb2.getString("greeting.morning"), Locale.CHINA).format(params);
String str3 = new MessageFormat(rb2.getString("greeting.afternoon"), Locale.CHINA).format(params);
System.out.println(str1);
System.out.println(str2);
System.out.println(str3);
```

运行以上代码，将输出以下信息：

```
How are you!John,today is 1/9/07 4:11 PM
早上好！John，现在是下午4:11
下午好！John，现在是2016年1月9日
```

6.5.2 MessageSource

Spring 定义了访问国际化信息的 MessageSource 接口，并提供了若干个易用的实现类。首先来了解一下该接口的几个重要方法。

- String getMessage(String code, Object[] args, String defaultMessage, Locale locale):
code 表示国际化信息中的属性名；args 用于传递格式化串占位符所用的运行期参数；当在资源中找不到对应的属性名时，返回 defaultMessage 参数指定的默认信息；locale 表示本地化对象。

- `String getMessage(String code, Object[] args, Locale locale) throws NoSuchMessageException`: 与上面的方法类似，只不过在找不到资源中对应的属性名时，直接抛出 `NoSuchMessageException` 异常。
- `String getMessage(MessageSourceResolvable resolvable, Locale locale) throws NoSuchMessageException`: `MessageSourceResolvable` 将属性名、参数数组及默认信息封装起来，它的功能和第一个接口方法相同。

1. MessageSource 的类结构

`MessageSource` 分别被 `HierarchicalMessageSource` 和 `ApplicationContext` 接口扩展，这里主要看一下 `HierarchicalMessageSource` 接口的几个实现类，如图 6-7 所示。

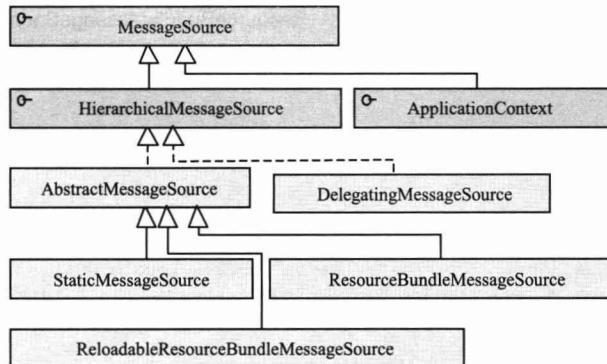


图 6-7 MessageSource 的类结构

`HierarchicalMessageSource` 接口添加了两个方法，建立父子层级的 `MessageSource` 结构，类似于前面介绍的 `HierarchicalBeanFactory`。该接口的 `setParentMessageSource(MessageSource parent)` 方法用于设置父 `MessageSource`，而 `getParentMessageSource()` 方法用于返回父 `MessageSource`。

`HierarchicalMessageSource` 接口最重要的两个实现类是 `ResourceBundleMessageSource` 和 `ReloadableResourceBundleMessageSource`。它们基于 Java 的 `ResourceBundle` 基础类实现，允许仅通过资源名加载国际化信息。`ReloadableResourceBundleMessageSource` 提供了定时刷新功能，允许在不重启系统的情况下更新资源的信息。`StaticMessageSource` 主要用于程序测试，允许通过编程的方式提供国际化信息。而 `DelegatingMessageSource` 是为方便操作父 `MessageSource` 而提供的代理类。

2. ResourceBundleMessageSource

该实现类允许用户通过 `beanName` 指定一个资源名（包括类路径的全限定资源名），或通过 `beanNames` 指定一组资源名。在代码清单 6-17 中通过 JDK 的基础类完成了本地化操作，下面使用 `ResourceBundleMessageSource` 来完成相同任务，如代码清单 6-18 所示。读者可以比较两者的使用差别，并体会 Spring 所提供的国际化处理功能所带来的好处。

代码清单 6-18 通过 ResourceBundleMessageSource 配置资源

```
<bean id="myResource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
    <!--①通过基名指定资源，相对于类根路径-->
    <property name="basenames">
      <list>
        <value>com/smart/i18n/fmt_resource</value>
      </list>
    </property>
  </bean>
```

启动 Spring 容器，并通过 MessageSource 访问配置的国际化信息，如代码清单 6-19 所示。

代码清单 6-19 访问国际化信息：ResourceBundleMessageSource

```
String[] configs = {"com/smart/i18n/beans.xml"};
ApplicationContext ctx = new ClassPathXmlApplicationContext(configs);

//①获取 MessageSource 的 Bean
MessageSource ms = (MessageSource)ctx.getBean("myResource");
Object[] params = {"John", new GregorianCalendar().getTime()};

//②获取格式化的国际化信息
String str1 = ms.getMessage("greeting.common",params,Locale.US);
String str2 = ms.getMessage("greeting.morning",params,Locale.CHINA);
String str3 = ms.getMessage("greeting.afternoon",params,Locale.CHINA);
System.out.println(str1);
System.out.println(str2);
System.out.println(str3);
```

分析代码清单 6-19 中的代码，发现最主要的区别在于无须分别加载不同语言、不同国家/地区的本地化资源文件，仅仅通过资源名就可以加载整套国际化信息资源文件。此外，无须显式使用 MessageFormat 操作国际化信息，仅通过 MessageSource#getMessage()方法就可以完成操作。这段代码的运行结果与代码清单 6-17 的运行结果完全相同。

3. ReloadableResourceBundleMessageSource

前面提到该实现类相比于 ResourceBundleMessageSource 的唯一区别在于它可以定时刷新资源文件，以便在应用程序不重启的情况下感知资源文件的变化。很多生产系统都需要长时间地持续运行，系统重启会给运行带来很大的负面影响。这时，通过该实现类就可以解决国际化信息更新的问题。请看如代码清单 6-20 所示的配置。

代码清单 6-20 通过 ReloadableResourceBundleMessageSource 配置资源

```
<bean id="myResource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>com/smart/i18n/fmt_resource</value>
      </list>
    </property>
  </bean>
```

```

</property>
<property name="cacheSeconds" value="5" /> ① ← 刷新资源文件的
</bean> 周期, 以秒为单位

```

在上面的配置中, 通过 cacheSeconds 属性让 ReloadableResourceBundleMessageSource 每 5 秒钟刷新一次资源文件 (在真实的应用中, 刷新周期不能太短, 否则频繁刷新将带来性能上的负面影响, 一般建议不小于 1 分钟)。cacheSeconds 默认值为 -1, 表示永不刷新, 此时, 该实现类的功能就蜕化为 ResourceBundleMessageSource 的功能。

编写一个测试类对上面配置的 ReloadableResourceBundleMessageSource 进行测试, 如代码清单 6-21 所示。

代码清单 6-21 刷新资源: ReloadableResourceBundleMessageSource

```

String[] configs = {"com/smart/i18n/beans.xml"};
ApplicationContext ctx = new ClassPathXmlApplicationContext(configs);

MessageSource ms = (MessageSource)ctx.getBean("myResource");
Object[] params = {"John", new GregorianCalendar().getTime()};

for (int i = 0; i < 2; i++) {
    String str1 = ms.getMessage("greeting.common",params,Locale.US);
    System.out.println(str1);
    Thread.currentThread().sleep(20000); ① ← 模拟程序应用, 在此期间, 更改
}                                            资源文件

```

在①处让程序睡眠 20 秒, 在此期间, 将 fmt_resource_zh_CN.properties 资源文件的 greeting.common 键值调整为:

```
--How are you!{0},today is {1}---
```

可以看到两次输出的格式化信息分别对应更改前后的内容, 即本地化资源文件的调整自动生效了。

```
How are you!John,today is 1/9/07 4:55 PM
---How are you!John,today is 1/9/07 4:55 PM---
```

6.5.3 容器级的国际化信息资源

在如图 6-7 所示的 MessageSource 类结构中, 我们发现 ApplicationContext 实现了 MessageSource 的接口。也就是说, ApplicationContext 的实现类本身也是一个 MessageSource 对象。

将 ApplicationContext 和 MessageSource 整合起来,乍一看令人费解, Spring 这样设计的意图究竟是什么呢?原来 Spring 认为,在一般情况下,国际化信息资源应该是容器级的。一般不会将 MessageSource 作为一个 Bean 注入其他的 Bean 中,相反,MessageSource 作为容器的基础设施向容器中所有的 Bean 开放。只要考察一下国际化信息的实际消费场所,就更能理解 Spring 这样设计的用意了。国际化信息资源一般在系统输出信息时使用,如 Spring MVC 的页面标签、控制器 (Controller) 等,不同的模块都可能通过这些组件访问国际化信息资源,因此 Spring 将国际化信息资源作为容器的公共基础设施对所有组件开放。

既然一般情况下不会直接通过引用 MessageSource Bean 使用国际化信息资源，那么如何声明容器级的国际化信息资源呢？其实在 6.1.1 节讲解 Spring 容器的内部工作机制时已经埋下了伏笔：在介绍容器启动过程时，通过代码清单 6-1 对 Spring 容器启动时的步骤进行了剖析，④处的 initMessageSource()方法所执行的工作就是初始化容器中的国际化信息资源，它根据反射机制从 BeanDefinitionRegistry 中找出名为 **messageSource** 且类型为 **org.springframework.context.MessageSource** 的 Bean，将这个 Bean 定义的信息资源加载为容器级的国际化信息资源。请看如代码清单 6-22 所示的配置。

代码清单 6-22 容器级资源的配置

```
<!--①注册资源 Bean，其 Bean 名称只能为 messageSource -->
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>com/smart/i18n/fmt_resource</value>
      </list>
    </property>
</bean>
```

下面通过 ApplicationContext 直接访问国际化信息资源，如代码清单 6-23 所示。

代码清单 6-23 通过 ApplicationContext 访问国际化信息资源

```
String[] configs = {"com/smart/i18n/beans.xml"};
ApplicationContext ctx = new ClassPathXmlApplicationContext(configs);
//①直接通过容器访问国际化信息资源
Object[] params = {"John", new GregorianCalendar().getTime()};

String str1 = ctx.getMessage("greeting.common", params, Locale.US);
String str2 = ctx.getMessage("greeting.morning", params, Locale.CHINA);
System.out.println(str1);
System.out.println(str2);
```

运行以上代码，输出以下信息：

```
How are you!John,today is 1/9/07 5:24 PM
早上好！John，现在是下午5:24
```

假设 MessageSource Bean 没有被命名为“messageSource”，那么以上代码将抛出 NoSuchMessageException 异常。

6.6 容器事件

Spring 的 ApplicationContext 能够发布事件并且允许注册相应的事件监听器，因此，它拥有一套完善的事件发布和监听机制。我们知道，Java 通过 java.util.EventObject 类和 java.util.EventListener 接口描述事件和监听器，某个组件或框架如需事件发布和监听机制，都需要通过扩展它们进行定义。在事件体系中，除了事件和监听器外，还有另外 3 个重要的概念。

- 事件源：事件的产生者，任何一个 EventObject 都必须拥有一个事件源。
- 事件监听器注册表：组件或框架的事件监听器不可能飘浮在空中，而必须有所依存。也就是说组件或框架必须提供一个地方保存事件监听器，这便是事件监听器注册表。一个事件监听器注册到组件或框架中，其实就是保存在事件监听器注册表中。当组件和框架中的事件源产生事件时，就会通知这些位于事件监听器注册表中的监听器。
- 事件广播器：它是事件和事件监听器沟通的桥梁，负责把事件通知给事件监听器。

通过图 6-8 可以看出这几个角色是如何各司其职的。

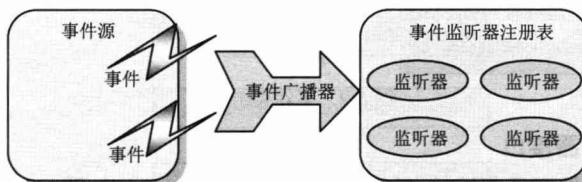


图 6-8 事件体系

事件源、事件监听器注册表和事件广播器这 3 个角色有时可以由同一个对象承担，如 java.swing 包中的 JButton、JCheckBox 等组件，它们分别集以上 3 个角色于一身。

在分析了事件体系后，我们会发现事件体系其实是观察者模式的一种具体实现方式，它并没有任何神秘之处。之所以组件或框架的事件会让一些开发者觉得神奇，就是因为组件或框架通过观察者模式很好地封装了事件模型并透明地提供给使用者，使用者只需按其设定的方式定义并注册事件监听器，事件体系就可以正常工作，因而我们很少会关注它的内部实现机理。

6.6.1 Spring 事件类结构

1. 事件类

首先来了解一下 Spring 的事件类结构。目前 Spring 框架本身仅定义了几个事件，如图 6-9 所示。

ApplicationEvent 的唯一构造函数是 ApplicationEvent(Object source)，通过 source 指定事件源，它有两个子类。

- ApplicationContextEvent：容器事件，它拥有 4 个子类，分别表示容器启动、刷新、停止及关闭的事件。
 - RequestHandleEvent：这是一个与 Web 应用相关的事件，当一个 HTTP 请求被处理后，产生该事件。只有在 web.xml 中定义了 DispatcherServlet 时才会产生该事件。它拥有两个子类，分别代表 Servlet 及 Portlet 的请求事件。
- 也可以根据需要扩展 ApplicationEvent 定义自己的事件，完成其他特殊的功能。

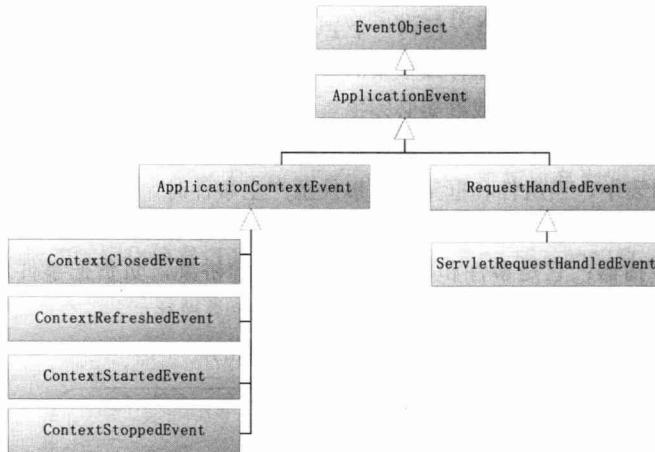


图 6-9 事件类结构

2. 事件监听器接口

Spring 的事件监听器都继承自 ApplicationListener 接口，如图 6-10 所示。

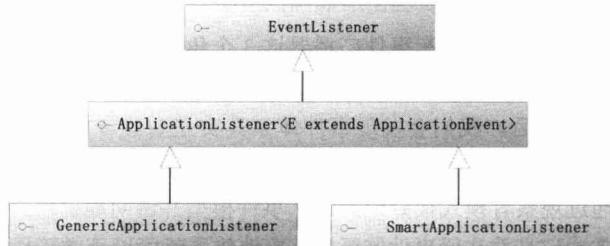


图 6-10 事件监听器接口

ApplicationListener 接口只定义了一个方法：onApplicationEvent(E event)，该方法接收 ApplicationEvent 事件对象，在该方法中编写事件的响应处理逻辑。而 SmartApplicationListener 接口是 Spring 3.0 新增的，它定义了两个方法。

- boolean supportsEventType(Class<? extends ApplicationEvent> eventType): 指定监听器支持哪种类型的容器事件，即它只会对该类型的事件做出响应。
- boolean supportsSourceType(Class<?> sourceType): 指定监听器仅对何种事件源对象做出响应。

其中，GenericApplicationListener 接口是 Spring 4.2 新增的。与 SmartApplicationListener 接口不同的是，它增强了对泛型事件类型的支持，supportsEventType()方法的参数不再仅限于 ApplicationEvent 子类型，而是采用可解析类型 ResolvableType。ResolvableType 是 Spring 4.0 提供的一个更加简单易用的泛型操作支持类。通过 ResolvableType 可以很容易地获取到泛型的实际类型信息，包括获取类级、字段级别、方法返回值、构造器参数及数组组件类型的泛型信息。在 Spring 4.0 框架中，很多核心类内部涉及的泛型操作都替换为 ResolvableType 类进行处理（如 BeanWrapperImpl、GenericTypeResolver 等）。

GenericApplicationListener 定义了两个方法。

- boolean supportsEventType(ResolvableType eventType): 指定监听器是否实际支持给定的事件类型，即它只会对该类型的事件做出响应。
- boolean supportsSourceType(Class<?> sourceType): 指定监听器仅对何种事件源对象做出响应。

3. 事件广播器

当发生容器事件时，容器主控程序将调用事件广播器将事件通知给事件监听器注册表中的事件监听器，事件监听器分别对事件进行响应。Spring 为事件广播器定义了接口，并提供了实现类，如图 6-11 所示。

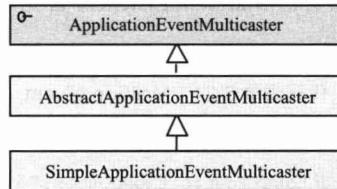


图 6-11 事件广播器类结构

6.6.2 解构 Spring 事件体系的具体实现

Spring 在 ApplicationContext 接口的抽象实现类 AbstractApplicationContext 中完成了事件体系的搭建。AbstractApplicationContext 拥有一个 applicationEventMulticaster 成员变量，applicationEventMulticaster 提供了容器监听器的注册表。AbstractApplicationContext 在 refresh()这个容器启动方法中通过以下 3 个步骤搭建了事件的基础设施。代码清单 6-1 中列出了 refresh()内部的整个过程，为了阅读方便，这里再次给出和事件体系有关的代码，如下：

```

//⑤初始化应用上下文事件广播器
initApplicationEventMulticaster();
...
//⑦注册事件监听器
registerListeners();
...
//⑧完成刷新并发布容器刷新事件
finishRefresh();
  
```

首先，在⑤处，Spring 初始化事件的广播器。用户可以在配置文件中为容器定义一个自定义的事件广播器，只要实现 ApplicationEventMulticaster 即可，Spring 会通过反射机制将其注册成容器的事件广播器。如果没有找到配置的外部事件广播器，则 Spring 自动使用 SimpleApplicationEventMulticaster 作为事件广播器。

在⑦处，Spring 根据反射机制，从 BeanDefinitionRegistry 中找出所有实现 org.springframework.context.ApplicationListener 的 Bean，将它们注册为容器的事件监听

器，实际操作就是将其添加到事件广播器所提供的事件监听器注册表中。

在⑨处，容器启动完成，调用事件发布接口向容器中所有的监听器发布事件。在 publishEvent() 内部可以看到，Spring 委托 ApplicationEventMulticaster 将事件通知给事件监听器。

6.6.3 一个实例

本节通过一个实例讲解事件发布和事件监听的整体过程。这个例子包括一个模拟的邮件发送器 MailSender，它在向目的地发送邮件时，将产生一个 MailSendEvent 事件，容器中注册了监听该事件的监听器 MailSendListener。首先来看一下 MailSendEvent 的代码，如代码清单 6-24 所示。

代码清单 6-24 MailSendEvent

```
package com.smart.event;

import org.springframework.context.ApplicationContext;
import org.springframework.context.event.ApplicationContextEvent;

public class MailSendEvent extends ApplicationContextEvent {
    private String to;

    public MailSendEvent(ApplicationContext source, String to) {
        super(source);
        this.to = to;
    }

    public String getTo() {
        return this.to;
    }
}
```

它直接扩展 ApplicationContextEvent，事件对象除 source 属性外，还具有一个代表发送目的地的 to 属性。

事件监听器 MailSenderListener 负责监听 MailSendEvent 事件，它的代码如下所示：

```
package com.smart.event;
import org.springframework.context.ApplicationListener;
public class MailSendListener implements ApplicationListener<MailSendEvent> {

    //①对MailSendEvent事件进行处理
    public void onApplicationEvent(MailSendEvent event) {
        MailSendEvent mse = (MailSendEvent) event;
        System.out.println("MailSendListener:向" + mse.getTo() + "发送完一封邮件");
    }
}
```

MailSenderListener 直接实现 ApplicationListener 接口，在接口方法中通过 instanceof 操作符判断事件的类型，仅对 MailSendEvent 类型的事件进行处理。

MailSender 要拥有发布事件的能力，就必须实现 ApplicationContextAware 接口，如下：

```

package com.smart.event;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class MailSender implements ApplicationContextAware {
    private ApplicationContext ctx;

    //①ApplicationContextAware的接口方法，以便容器启动时注入容器实例
    public void setApplicationContext(ApplicationContext ctx)
        throws BeansException {
        this.ctx = ctx;
    }

    public void sendMail(String to) {
        System.out.println("MailSender:模拟发送邮件...");
        MailSendEvent mse = new MailSendEvent(this.ctx,to);
        //②向容器中的所有事件监听器发送事件
        ctx.publishEvent(mse);
    }
}

```

在 sendMail()方法中，首先模拟发送邮件，然后产生一个 MailSendEvent 事件，并通过容器句柄 ctx 向容器中的所有事件监听器发送事件。

在 Spring 的配置文件中，仅需进行如下配置：

```

<bean class="com.smart.event.MailSendListener"/>
<bean id="mailSender" class="com.smart.event.MailSender"/>

```

下面的代码启动容器并调用 mailSender Bean 发送一封邮件：

```

ApplicationContext ctx = new ClassPathXmlApplicationContext("com/smart/event/beans.xml");
MailSender mailSender = (MailSender)ctx.getBean("mailSender");
mailSender.sendMail("aaa@bbb.com");

```

当容器启动时，它会根据配置文件自动注册 MailSendListener 这个事件监听器。运行以上代码，输出以下信息：

```

MailSender:模拟发送邮件...
MailSendListener:向 aaa@bbb.com 发送完一封邮件

```

6.7 小结

在本章中，我们对 Spring 容器进行了深度剖析，不但分析了 Spring 容器的运行流程，还深入 Spring 的内部探究其实现机理，并介绍了组成 Spring 容器的重要组件。Spring 不但只是一个优秀而实用的开发框架，该框架本身也是经典的程序设计范本，我们希望通过对其内部设计的深入分析，让读者从中吸取设计思想的精华并应用到自己的开发实践中去，而不仅仅使用 Spring 框架的功能。

此外，还介绍了 Spring 的属性编辑器、外部属性文件、国际化信息及容器事件的知识。在介绍这些知识前，我们学习了相关主题的 Java 知识，这些知识可以帮助读者加深对 Spring 相关知识的理解。

第 7 章

Spring AOP 基础

Spring AOP 是 AOP 技术在 Spring 中的具体实现，它是构成 Spring 框架的另一个重要基石。Spring AOP 构建于 IoC 之上，和 IoC “浑然天成”，统一于 Spring 容器之中。本章将从 Spring AOP 的底层实现技术入手，一步步深入 Spring AOP 的内核，分析它的底层结构和实现。

本章主要内容：

- ◆ AOP 概述
- ◆ Spring AOP 所涉及的 Java 基础知识
- ◆ Spring AOP 的增强类型
- ◆ Spring AOP 的切面类型
- ◆ 通过自动代理技术创建切面

本章亮点：

- ◆ 通过 Spring AOP 所涉及的底层 Java 知识的学习深刻理解 Spring AOP 的具体实现
- ◆ 深入 Spring AOP 的内核分析其组成和结构
- ◆ AOP 疑难问题剖析

7.1 AOP 概述

编程语言的终极目标就是能以更自然、更灵活的方式模拟世界，从原始机器语言到过程语言再到面向对象语言，编程语言一步步地用更自然、更灵活的方式编写软件。AOP 是软件开发思想发展到一定阶段的产物，但 AOP 的出现并不是要完全替代 OOP，而仅作为 OOP 的有益补充。虽然 AOP 作为一项编程技术已经有多年的历史，但长时间停留在学术领域，直到近几年，AOP 才作为一项真正的实用技术在应用领域开疆拓土。需要

指出的是，AOP是有特定的应用场合的，它只适合那些具有横切逻辑的应用场合，如性能监测、访问控制、事务管理及日志记录（虽然有很多文章用日志记录作为讲解AOP的实例，但很多人认为很难用AOP编写实用的程序日志，笔者对此观点非常认同）。不过，这丝毫不影响AOP作为一种新的软件开发思想在软件开发领域所占有的地位。

7.1.1 AOP到底是什么

AOP是Aspect Oriented Programming的简称，最初被译为“面向方面编程”，这个翻译向来为人所诟病，但是由于先入为主的效应，受众广泛，所以这个翻译依然被很多人使用。但我们更倾向于用“面向切面编程”的译法，因为它更加达意。

按照软件重构思想的理念，如果多个类中出现相同的代码，则应该考虑定义一个父类，将这些相同的代码提取到父类中。比如Horse、Pig、Camel这些对象都有run()和eat()方法，通过引入一个包含这两个方法的抽象的Animal父类，Horse、Pig、Camel就可以通过继承Animal复用run()和eat()方法。通过引入父类消除多个类中重复代码的方式在大多数情况下是可行的，但世界并非永远这样简单，请看如代码清单7-1所示的论坛管理业务类的代码。

代码清单7-1 ForumService

```
package com.smart.concept;
public class ForumService {
    private TransactionManager transManager;
    private PerformanceMonitor pmonitor;
    private TopicDao topicDao;
    private ForumDao forumDao;

    public void removeTopic(int topicId) {
        pmonitor.start();
        transManager.beginTransaction();
        topicDao.removeTopic(topicId); //①
        transManager.commit();
        pmonitor.end();
    }
    public void createForum(Forum forum) {
        pmonitor.start();
        transManager.beginTransaction();
        forumDao.create(forum); //②
        transManager.commit();
        pmonitor.end();
    }
    ...
}
```

代码清单7-1中斜体的代码是方法性能监视代码，它在方法调用前启动，在方法调用返回前结束，并在内部记录性能监视的结果信息。而黑色粗体的代码是事务开始和事务提交的代码。我们发现①、②处的业务代码淹没在重复化非业务性的代码之中，性能

监视和事务管理这些非业务性代码葛藤缠树般包围着业务性代码。

如图 7-1 所示，假设将 ForumService 业务类看成一段圆木，将 removeTopic() 和 createForum() 方法分别看成圆木的一截，会发现性能监视和事务管理的代码就好像一个年轮，而业务代码是圆木的树心，这也正是横切代码概念的由来。

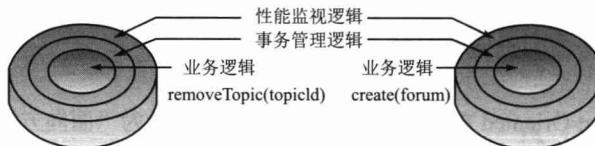


图 7-1 横切逻辑示意图

我们无法通过抽象父类的方式消除如上所示的重复性横切代码，因为这些横切逻辑依附在业务类方法的流程中，它们不能转移到其他地方去。

AOP 独辟蹊径，通过横向抽取机制为这类无法通过纵向继承体系进行抽象的重复性代码提供了解决方案。对于习惯了纵向抽取的开发者来说，可能不太容易理解横向抽取方法的工作机制，因为 Java 语言本身不直接提供这种横向抽取的能力。暂把具体实现放在一旁，先通过图解的方式归纳出 AOP 的解决思路，如图 7-2 所示。

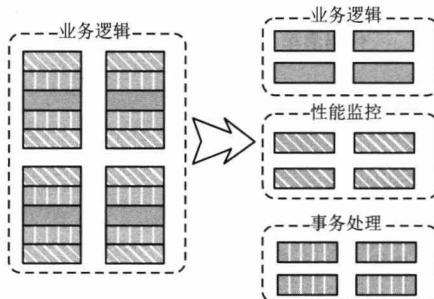


图 7-2 横向抽取

从图 7-2 中可以看出，AOP 希望将这些分散在各个业务逻辑代码中的相同代码通过横向切割的方式抽取到一个独立的模块中，还业务逻辑类一个清新的世界。

当然，将这些重复性的横切逻辑独立出来是很容易的，但如何将这些独立的逻辑融合到业务逻辑中以完成和原来一样的业务流程，才是事情的关键，这也正是 AOP 要解决的主要问题。



轻松一刻

现在常用“雁过拔毛”来形容某人爱贪便宜，对每件经手的事情都要获得一些好处。“雁过拔毛”就是现实生活中 AOP 的一个很形象的例子。其实“雁过拔毛”的原意是形容武艺高超，大雁飞过时也能拔下它的毛来。

7.1.2 AOP 术语

如学习电学要先学习电阻、电压、电容等专业术语一样，AOP 也有一些自己的行话。为了方便后面的学习，先来了解一下 AOP 的几个重要术语。

1. 连接点（Joinpoint）

特定点是程序执行的某个特定位置，如类开始初始化前、类初始化后、类的某个方法调用前/调用后、方法抛出异常后。一个类或一段程序代码拥有一些具有边界性质的特定点，这些代码中的特定点就被称为“连接点”。Spring 仅支持方法的连接点，即仅能在方法调用前、方法调用后、方法抛出异常时及方法调用前后这些程序执行点织入增强。我们知道，黑客攻击系统需要找到突破口，没有突破口就无法进行攻击。从某种程度上来说，AOP 也可以看成一个黑客（因为它要向目标类中嵌入额外的代码逻辑），连接点就是 AOP 向目标类打入楔子的候选锚点。

连接点由两个信息确定：一是用方法表示的程序执行点；二是用相对位置表示的方位。如在 Test.foo() 方法执行前的连接点，执行点为 Test.foo()，方位为该方法执行前的位置。Spring 使用切点对执行点进行定位，而方位则在增强类型中定义。

2. 切点（Pointcut）

每个程序类都拥有多个连接点，如一个拥有两个方法的类，这两个方法都是连接点，即连接点是程序类中客观存在的事物。但在为数众多的连接点中，如何定位某些感兴趣的连接点呢？AOP 通过“切点”定位特定的连接点。借助数据库查询的概念来理解切点和连接点的关系再合适不过了：连接点相当于数据库中的记录，而切点相当于查询条件。切点和连接点不是一对一的关系，一个切点可以匹配多个连接点。

在 Spring 中，切点通过 org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件，Spring AOP 的规则解析引擎负责解析切点所设定的查询条件，找到对应的连接点。确切地说，应该是执行点而非连接点，因为连接点是方法执行前、执行后等包括方位信息的具体程序执行点，而切点只定位到某个方法上，所以如果希望定位到具体的连接点上，还需要提供方位信息。

3. 增强（Advice）

增强是织入目标类连接点上的一段程序代码，是不是觉得 AOP 越来越像黑客了，这不是往业务类中装入木马吗？我们大可按照这一思路去理解增强，因为这样更形象易懂。在 Spring 中，增强除用于描述一段程序代码外，还拥有另一个和连接点相关的信息，这便是执行点的方位。结合执行点的方位信息和切点信息，就可以找到特定的连接。正因为增强既包含用于添加到目标连接点上的一段执行逻辑，又包含用于定位连接点的方位信息，所以 Spring 所提供的增强接口都是带方位名的，如 BeforeAdvice、AfterReturningAdvice、ThrowsAdvice 等。BeforeAdvice 表示方法调用前的位置，而 AfterReturningAdvice 表示访问返回后的位置。所以只有结合切点和增强，才能确定特定的连接点并实施增强逻辑。

有很多书籍和文章将 Advice 译为“通知”，就像将“how old are you?”译为“怎么老是你”一样，明显是一种“望文生义”的译法。来看几个使用“通知”的语境：银行向张三发出了一个催款通知；班主任通知学生明天大扫除。从这些语境中可以知道，通知者只是把某个消息传达给被通知者，并不会替被通知者做任何事情。而 Spring 的 Advice 必须嵌入类的某连接点上，并完成一段附加的执行逻辑，这明显是去“增强”目标类的功能。当然，我们不能对这个翻译有过多的微词，毕竟 Advice 这个英文单词本身就有不知所云，如果将其改为 Enhancer，相信理解起来会更容易一些。



轻松一刻

早期国外普遍采用韦氏拼音来拼写中国的人名、地名，一些译者在翻译国外作品时由于未注意到这一背景，闹出了不少笑话，如将蒋介石（Chiang Kai-shek）译为常凯申，将孟子（Mencius）译为门修斯，这些留洋归来的名人着实“洋气”了一把。

4. 目标对象（Target）

增强逻辑的织入目标类。如果没有 AOP，那么目标业务类需要自己实现所有的逻辑，就如代码清单 7-1 中的 ForumService 所示。在 AOP 的帮助下，ForumService 只实现那些非横切逻辑的程序逻辑，而性能监视和事务管理等这些横切逻辑则可以使用 AOP 动态织入特定的连接点上。

5. 引介（Introduction）

引介是一种特殊的增强，它为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口，通过 AOP 的引介功能，也可以动态地为该业务类添加接口的实现逻辑，让业务类成为这个接口的实现类。

6. 织入（Weaving）

织入是将增强添加到目标类的具体连接点上的过程。AOP 就像一台织布机，将目标类、增强或者引介天衣无缝地编织到一起。我们不能不说“织入”这个词太精辟了。根据不同的实现技术，AOP 有 3 种织入方式。

- (1) 编译期织入，这要求使用特殊的 Java 编译器。
- (2) 类装载期织入，这要求使用特殊的类装载器。
- (3) 动态代理织入，在运行期为目标类添加增强生成子类的方式。

Spring 采用动态代理织入，而 AspectJ 采用编译期织入和类装载期织入。

7. 代理（Proxy）

一个类被 AOP 织入增强后，就产生了一个结果类，它是融合了原类和增强逻辑的代理类。根据不同的代理方式，代理类既可能是和原类具有相同接口的类，也可能就是原类的子类，所以可以采用与调用原类相同的方式调用代理类。

8. 切面 (Aspect)

切面由切点和增强（引介）组成，它既包括横切逻辑的定义，也包括连接点的定义。Spring AOP 就是负责实施切面的框架，它将切面所定义的横切逻辑织入切面所指定的连接点中。

AOP 的工作重心在于如何将增强应用于目标对象的连接点上。这里包括两项工作：第一，如何通过切点和增强定位到连接点上；第二，如何在增强中编写切面的代码。本章大部分内容都将围绕这两点展开。

7.1.3 AOP 的实现者

AOP 工具的设计目标是把横切的问题（如性能监视、事务管理）模块化。使用类似 OOP 的方式进行切面的编程工作。位于 AOP 工具核心的是连接点模型，它提供了一种机制，可以定位到需要在哪里发生横切。

1. AspectJ

AspectJ 是语言级的 AOP 实现，2001 年由 Xerox PARC 的 AOP 小组发布，目前版本已经更新到 1.8.9。AspectJ 扩展了 Java 语言，定义了 AOP 语法，能够在编译期提供横切代码的织入，所以它有一个专门的编译器用来生成遵守 Java 字节编码规范的 Class 文件。其主页位于 <http://www.eclipse.org/aspectj>。

2. AspectWerkz

AspectWerkz 是基于 Java 的简单、动态、轻量级的 AOP 框架，该框架于 2002 年发布，由 BEA Systems 提供支持。它支持运行期或类装载期织入横切代码，所以它拥有一个特殊的类装载器。现在，AspectJ 和 AspectWerkz 项目已经合并，以便整合二者的力量和技术创建统一的 AOP 平台。它们合作的第一个发布版本是 AspectJ 5：扩展 AspectJ 语言，以基于注解的方式支持类似 AspectJ 的代码风格。

3. JBoss AOP

JBoss AOP 于 2004 年作为 JBoss 应用程序服务器框架的扩展功能发布，读者可以从以下地址了解到 JBoss AOP 的更多信息：<http://www.jboss.org/products/aop>。

4. Spring AOP

Spring AOP 使用纯 Java 实现，它不需要专门的编译过程，也不需要特殊的类装载器，它在运行期通过代理方式向目标类织入增强代码。Spring 并不尝试提供最完整的 AOP 实现，相反，它侧重于提供一种和 Spring IoC 容器整合的 AOP 实现，用以解决企业级开发中的常见问题。在 Spring 中可以无缝地将 Spring AOP、IoC 和 AspectJ 整合在一起。

7.2 基础知识

Spring AOP 使用动态代理技术在运行期织入增强的代码，为了揭示 Spring AOP 底层的工作机理，有必要学习涉及的 Java 知识。Spring AOP 使用了两种代理机制：一种是基于 JDK 的动态代理；另一种是基于 CGLib 的动态代理。之所以需要两种代理机制，很大程度上是因为 JDK 本身只提供接口的代理，而不支持类的代理。

7.2.1 带有横切逻辑的实例

下面通过具体化代码实现 7.1 节所介绍的例子的性能监视横切逻辑，并通过动态代理技术对此进行改造。在调用每一个目标类方法时启动方法的性能监视，在目标类方法调用完成时记录方法的花费时间，如代码清单 7-2 所示。

代码清单 7-2 ForumService：包含性能监视横切代码

```
package com.smart.proxy;
public class ForumServiceImpl implements ForumService {
    public void removeTopic(int topicId) {
        //①-1 开始对该方法进行性能监视
        PerformanceMonitor.begin(
            "com.smart.proxy.ForumServiceImpl.removeTopic");
        System.out.println("模拟删除Topic记录："+topicId);
        try {
            Thread.currentThread().sleep(20);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        //①-2 结束对该方法的性能监视
        PerformanceMonitor.end();
    }

    public void removeForum(int forumId) {
        //②-1 开始对该方法进行性能监视
        PerformanceMonitor.begin(
            "com.smart.proxy.ForumServiceImpl.removeForum");
        System.out.println("模拟删除Forum记录："+forumId);
        try {
            Thread.currentThread().sleep(40);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        //②-2 结束对该方法的性能监视
    }
}
```

```

    PerformanceMonitor.end();
}
}

```

在代码清单 7-2 中，粗体表示的代码就是具有横切逻辑特征的代码，每个 Service 类和每个业务方法体的前后都执行相同的代码逻辑：方法调用前启动 PerformanceMonitor；方法调用后通知 PerformanceMonitor 结束性能监视并记录性能监视结果。

PerformanceMonitor 是性能监视的实现类，下面给出一个非常简单的实现版本，如代码清单 7-3 所示。

代码清单 7-3 PerformanceMonitor

```

package com.smart.proxy;
public class PerformanceMonitor {

    //①通过一个 ThreadLocal 保存与调用线程相关的性能监视信息
    private static ThreadLocal<MethodPerformance> performanceRecord =
        new ThreadLocal<MethodPerformance>();

    //②启动对某一目标方法的性能监视
    public static void begin(String method) {
        System.out.println("begin monitor...");
        MethodPerformance mp = new MethodPerformance(method);
        performanceRecord.set(mp);
    }

    public static void end() {
        System.out.println("end monitor...");
        MethodPerformance mp = performanceRecord.get();
    }

    //③打印出方法性能监视的结果信息
    mp.printPerformance();
}
}

```

ThreadLocal 是将非线程安全类改造为线程安全类的“法宝”（11.2 节将详细介绍）。PerformanceMonitor 提供了两个方法：通过调用 begin(String method) 方法开始对某个目标类方法的监视，其中 method 为目标类方法的全限定名；而通过调用 end() 方法结束对目标类方法的监视，并给出性能监视信息。这两个方法必须配套使用。

用于记录性能监视信息的 MethodPerformance 类的代码如代码清单 7-4 所示。

代码清单 7-4 MethodPerformance

```

package com.smart.proxy;
public class MethodPerformance {
    private long begin;
    private long end;
    private String serviceMethod;
    public MethodPerformance(String serviceMethod) {
        this.serviceMethod = serviceMethod;
        this.begin = System.currentTimeMillis(); ① ← 记录目标类方法开始
                                                执行点的系统时间
    }

    public void printPerformance() {
        end = System.currentTimeMillis(); ② ← 获取目标类方法执行完成后的系统时
        long elapse = end - begin;      间，进而计算出目标类方法的执行时间
    }
}

```

```

        System.out.println(serviceMethod+"花费"+elapse+"毫秒。"); ③ ↗
    }
}

报告目标类方法的执行时间

```

通过下面的代码测试拥有性能监视能力的 ForumServiceImpl 业务方法：

```

package com.smart.proxy;
public class TestForumService {
    public static void main(String[] args) {
        ForumService forumService = new ForumServiceImpl();
        forumService.removeForum(10);
        forumService.removeTopic(1012);
    }
}

```

得到以下输出信息：

```

begin monitor...
模拟删除Forum记录:10
end monitor...
com.smart.proxy.ForumServiceImpl.removeForum花费47毫秒。

begin monitor...
模拟删除Topic记录:1012
end monitor...
com.smart.proxy.ForumServiceImpl.removeTopic花费26毫秒。

```

正如代码清单 7-2 所示，当某个方法需要进行性能监视时，必须调整方法代码，在方法体前后分别添加开启性能监视和结束性能监视的代码。这些非业务逻辑的性能监视代码破坏了 ForumServiceImpl 业务逻辑的纯粹性。我们希望通过代理的方式将业务类方法中开启和结束性能监视的横切代码从业务类中完全移除，并通过 JDK 或 CGLib 动态代理技术将横切代码动态织入目标方法的相应位置。

7.2.2 JDK 动态代理

自 Java 1.3 以后，Java 提供了动态代理技术，允许开发者在运行期创建接口的代理实例。在 Sun 刚推出动态代理时，还很难想象它有多大的实际用途，现在终于发现动态代理是实现 AOP 的绝好底层技术。

JDK 的动态代理主要涉及 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。其中，InvocationHandler 是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态地将横切逻辑和业务逻辑编织在一起。

而 Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例，生成目标类的代理对象。这样描述一定很抽象，我们马上着手使用 Proxy 和 InvocationHandler 这两个“魔法戒”对 7.2.1 节中的性能监视代码进行革新。

首先从业务类 ForumServiceImpl 中移除性能监视的横切代码，使 ForumServiceImpl 只负责具体的业务逻辑，如代码清单 7-5 所示。

代码清单 7-5 ForumServiceImpl：移除性能监视横切代码

```

package com.smart.proxy;

public class ForumServiceImpl implements ForumService {

    public void removeTopic(int topicId) {
        //PerformanceMonitor.begin(...) ① ←
        System.out.println("模拟删除Topic记录:"+topicId);
        try {
            Thread.currentThread().sleep(20);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        //PerformanceMonitor.end();① ←
    }

    public void removeForum(int forumId) {
        //PerformanceMonitor.begin(...)② ←
        System.out.println("模拟删除Forum记录:"+forumId);
        try {
            Thread.currentThread().sleep(40);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        //PerformanceMonitor.end();② ←
    }
}

```

在代码清单 7-5 中的①和②处，原来的性能监视代码被移除了，只保留了真正的业务逻辑。

从业务类中移除性能监视横切代码后，必须为它找到一个安身之所，InvocationHandler 就是横切代码的“安家乐园”。将性能监视横切代码安置在 PerformanceHandler 中，如代码清单 7-6 所示。

代码清单 7-6 PerformanceHandler

```

package com.smart.proxy;
import java.lang.reflect.InvocationHandler;           实现 InvocationHandler
import java.lang.reflect.Method;

public class PerformanceHandler implements InvocationHandler { //① ←
    private Object target;
    public PerformanceHandler(Object target){//② ←
        this.target = target;                           target 为目标业务类
    }
    public Object invoke(Object proxy, Method method, Object[] args) //③
        throws Throwable {
        PerformanceMonitor.begin(
            target.getClass().getName()+"."+method.getName()); //③-1
        Object obj = method.invoke(target, args); //③-2 ←
        PerformanceMonitor.end();//③-1
        return obj;
    }
}

```

③处 `invoke()`方法中粗体所示部分的代码为性能监视的横切代码，我们发现，横切代码只出现一次，而不是像原来那样散落各处。③-2 处的 `method.invoke()`语句通过 Java 反射机制间接调用目标对象的方法，这样 `InvocationHandler` 的 `invoke()`方法就将横切逻辑代码（③-1）和业务类方法的业务逻辑代码（③-2）编织到一起，所以，可以将 `InvocationHandler` 看成一个编织器。下面对这段代码作进一步的说明。

首先实现 `InvocationHandler` 接口，该接口定义了一个 `invoke(Object proxy, Method method, Object[] args)`方法，其中，`proxy` 是最终生成的代理实例，一般不会用到；`method` 是被代理目标实例的某个具体方法，通过它可以发起目标实例方法的反射调用；`args` 是被代理实例某个方法的入参，在方法反射调用时使用。

其次，在构造函数里通过 `target` 传入希望被代理的目标对象，如②处所示；在 `InvocationHandler` 接口方法 `invoke(Object proxy, Method method, Object[] args)`里，将目标实例传递给 `method.invoke()`方法，并调用目标实例的方法，如③处所示。

下面通过 `Proxy` 结合 `PerformanceHandler` 创建 `ForumService` 接口的代理实例，如代码清单 7-7 所示。

代码清单 7-7 `ForumServiceTest`: 创建代理实例

```
package com.smart.proxy;
import java.lang.reflect.Proxy;
import org.testng.annotations.*;
public class ForumServiceTest {
    @Test
    public void proxy() {
        ForumService target = new ForumServiceImpl(); //①
        PerformanceHandler handler = new PerformanceHandler(target); //②
        ForumService proxy = (ForumService) Proxy.newProxyInstance( //③
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            handler);
        proxy.removeForum(10); //④
        proxy.removeTopic(1012);
    }
}
```

上面的代码完成了业务类代码和横切代码的编织工作并生成了代理实例。在②处，让 `PerformanceHandler` 将性能监视横切逻辑编织到 `ForumService` 实例中，然后在③处，通过 `Proxy` 的 `newProxyInstance()`静态方法为编织了业务类逻辑和性能监视逻辑的 `handler` 创建一个符合 `ForumService` 接口的代理实例。该方法的第一个入参为类加载器；第二个入参为创建代理实例所需实现的一组接口；第三个入参是整合了业务逻辑和横切逻辑的编织器对象。

按照③处的设置方式，这个代理实例实现了目标业务类的所有接口，即 `Forum`

ServiceImpl 的 ForumService 接口。这样就可以按照调用 ForumService 接口实例相同的方式调用代理实例，如④处所示。运行以上代码，输出以下信息：

```
begin monitor...
模拟删除Forum记录:10
end monitor...
com.smart.proxy.ForumServiceImpl.removeForum花费47毫秒。
```

```
begin monitor...
模拟删除Topic记录:1012
end monitor...
com.smart.proxy.ForumServiceImpl.removeTopic花费26毫秒。
```

我们发现，程序的运行效果和直接在业务类中编写性能监视逻辑的效果一致，但在这里，原来分散的横切逻辑代码已经被抽取到 PerformanceHandler 中。当其他业务类（如 UserService、SystemService 等）的业务方法也需要使用性能监视时，只要按照与代码清单 7-7 相似的方式分别为它们创建代理对象即可。下面通过时序图描述通过创建代理对象进行业务方法调用的整体逻辑，以进一步认识代理对象的本质，如图 7-3 所示。

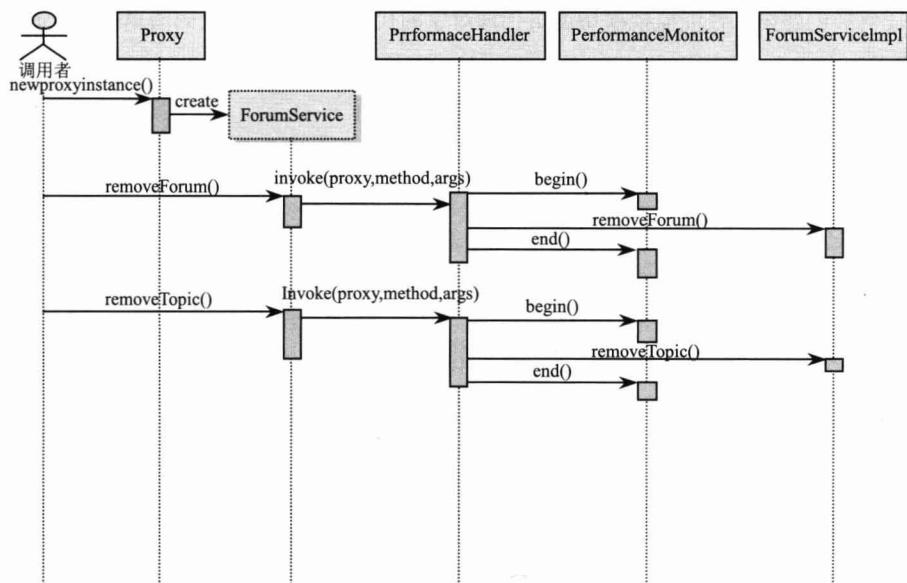


图 7-3 代理实例方法调用的时序图

在图 7-3 中使用虚线的方式对通过 Proxy 创建的 ForumService 代理实例加以突显，ForumService 代理实例内部利用 PerformanceHandler 整合横切逻辑和业务逻辑。调用者调用代理对象的 removeForum() 和 removeTopic() 方法时，图 7-3 所示的内部调用时序清晰地告诉我们实际上后台所发生的一切。

7.2.3 CGLib 动态代理

使用 JDK 创建代理有一个限制，即它只能为接口创建代理实例，这一点可以从 Proxy 的接口方法 newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h) 中看得很清楚：第二个入参 interfaces 就是需要代理实例实现的接口列表。虽然面向接口编程的思想被很多大师级人物（包括 Rod Johnson）所推崇，但在实际开发中，许多开发者也对此深感困惑：难道对一个简单业务表的操作也需要老老实实地创建 5 个类（领域对象类、DAO 接口、DAO 实现类、Service 接口和 Service 实现类）吗？难道不能直接通过实现类构建程序吗？对于这个问题，很难给出一个孰优孰劣的准确判断，但仍有很多不使用接口的项目也取得了非常好的效果。

对于没有通过接口定义业务方法的类，如何动态创建代理实例呢？JDK 动态代理技术显然已经黔驴技穷，CGLib 作为一个替代者，填补了这项空缺。

CGLib 采用底层的字节码技术，可以为一个类创建子类，在子类中采用方法拦截的技术拦截所有父类方法的调用并顺势织入横切逻辑。下面采用 CGLib 技术编写一个可以为任何类创建织入性能监视横切逻辑代理对象的代理创建器，如代码清单 7-8 所示。

代码清单 7-8 CglibProxy

```
package com.smart.proxy;
import java.lang.reflect.Method;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class CglibProxy implements MethodInterceptor {
    private Enhancer enhancer = new Enhancer();
    public Object getProxy(Class clazz) { ← 设置需要创建子类的类
        enhancer.setSuperclass(clazz); //① ← 通过字节码技术动态
        enhancer.setCallback(this); ← 创建子类实例
        return enhancer.create(); //② ← 拦截父类所有方法的调用
    }
    public Object intercept(Object obj, Method method, Object[] args, //③
                           MethodProxy proxy) throws Throwable {
        PerformanceMonitor.begin(obj.getClass().getName() + ". " + method.getName()); //③-1
        Object result=proxy.invokeSuper(obj, args); //③-2 ←
        PerformanceMonitor.end(); //③-1 ← 通过代理类调用
        return result; ← 父类中的方法
    }
}
```

在上面的代码中，用户可以通过 getProxy(Class clazz) 方法为一个类创建动态代理对象，该代理对象通过扩展 clazz 实现代理。在这个代理对象中，织入性能监视的横切逻辑（粗体部分）。intercept(Object obj, Method method, Object[] args, MethodProxy proxy) 是 CGLib 定义的 Interceptor 接口方法，它拦截所有目标类方法的调用。其中，obj 表示目标类的实例；method 为目标类方法的反射对象；args 为方法的动态入参；proxy 为代理类实例。

下面通过 CglibProxy 为 ForumServiceImpl 类创建代理对象，并测试代理对象的方法，如代码清单 7-9 所示。

代码清单 7-9 ForumServiceTest：测试CGLib创建的代理类

```
package com.smart.proxy;
import java.lang.reflect.Proxy;
import org.testng.annotations.*;
public class ForumServiceTest {
    @Test
    public void proxy() {
        CglibProxy proxy = new CglibProxy();
        ForumServiceImpl forumService = //①
            (ForumServiceImpl )proxy.getProxy(ForumServiceImpl.class);
        forumService.removeForum(10);
        forumService.removeTopic(1023);
    }
}
```

在①处通过 CglibProxy 为 ForumServiceImpl 动态创建了一个织入性能监视逻辑的代理对象，并调用代理类的业务方法。运行上面的代码，输出以下信息：

```
begin monitor...
模拟删除 Forum 记录:10
end monitor...
com.smart.proxy.ForumServiceImpl$$EnhancerByCGLIB$$2a9199c0.removeForum 花费 47 毫秒。
begin monitor...
模拟删除 Topic 记录:1023
end monitor...
com.smart.proxy.ForumServiceImpl$$EnhancerByCGLIB$$2a9199c0.removeTopic 花费 16 毫秒。
```

观察以上输出，除了发现两个业务方法中都织入了性能监控的逻辑外，还发现代理类的名字变成 com.smart.proxy.ForumServiceImpl\$\$EnhancerByCGLIB\$\$ 2a9199c0，这个特殊的类就是 CGLib 为 ForumServiceImpl 动态创建的子类。

值得一提的是，由于 CGLib 采用动态创建子类的方式生成代理对象，所以不能对目标类中的 final 或 private 方法进行代理。

7.2.4 AOP 联盟

AOP 联盟 (<http://aopalliance.sourceforge.net>) 是众多开源 AOP 项目的联合组织，该组织的目的是为了制定一套规范描述 AOP 的标准，定义标准的 AOP 接口，以便各种遵守标准的具体实现可以相互调用。

这种标准的制定本应当由 Sun 来做，但是因为 Sun 运作迟缓，AOP 联盟便捷足先登，而且它的影响力越来越大。现在大部分的 AOP 实现都采用 AOP 联盟的标准，所以 AOP 联盟制定的规范已经成为事实上的标准。

7.2.5 代理知识小结

Spring AOP 的底层就是通过使用 JDK 或 CGLib 动态代理技术为目标 Bean 织入横切逻辑的。这里对动态创建代理对象作一个小结。

虽然通过 PerformanceHandler 或 CglibProxy 实现了性能监视横切逻辑的动态织入，但这种实现方式存在 3 个明显需要改进的地方。

(1) 目标类的所有方法都添加了性能监视横切逻辑，而有时这并不是我们所期望的，我们可能只希望对业务类中的某些特定方法添加横切逻辑。

(2) 通过硬编码的方式指定了织入横切逻辑的织入点，即在目标类业务方法的开始和结束前织入代码。

(3) 手工编写代理实例的创建过程，在为不同类创建代理时，需要分别编写相应的创建代码，无法做到通用。

以上 3 个问题在 AOP 中占用重要的地位，因为 Spring AOP 的主要工作就是围绕以上 3 点展开的：Spring AOP 通过 Pointcut（切点）指定在哪些类的哪些方法上织入横切逻辑，通过 Advice（增强）描述横切逻辑和方法的具体织入点（方法前、方法后、方法的两端等）。此外，Spring 通过 Advisor（切面）将 Pointcut 和 Advice 组装起来。有了 Advisor 的信息，Spring 就可以利用 JDK 或 CGLib 动态代理技术采用统一的方式为目标 Bean 创建织入切面的代理对象了。

JDK 动态代理所创建的代理对象，在 Java 1.3 下，性能差强人意。虽然在高版本的 JDK 中动态代理对象的性能得到了很大的提高，但有研究表明，CGLib 所创建的动态代理对象的性能依旧比 JDK 所创建的动态代理对象的性能高不少（大概 10 倍）。但 CGLib 在创建代理对象时所花费的时间却比 JDK 动态代理多（大概 8 倍）。对于 singleton 的代理对象或者具有实例池的代理，因为无须频繁地创建代理对象，所以比较适合采用 CGLib 动态代理技术；反之则适合采用 JDK 动态代理技术。

7.3 创建增强类

Spring 使用增强类定义横切逻辑，同时由于 Spring 只支持方法连接点，增强还包括在方法的哪一点加入横切代码的方位信息，所以增强既包含横切逻辑，又包含部分连接点的信息。

7.3.1 增强类型

AOP 联盟为增强定义了 org.aopalliance.aop.Advice 接口，Spring 支持 5 种类型的增强，先来了解一下增强接口继承关系图，如图 7-4 所示。

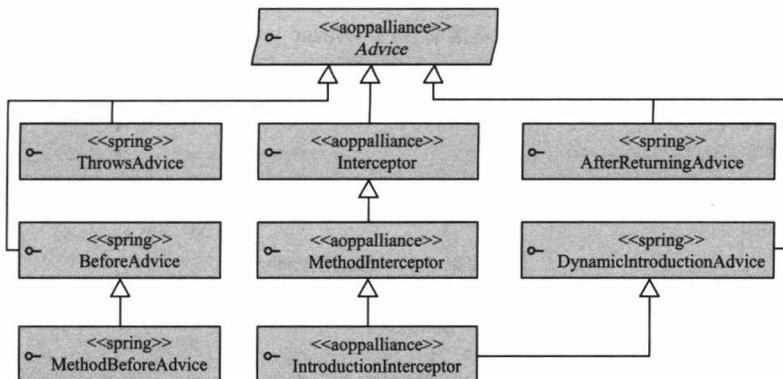


图 7-4 增强接口继承关系图

带<<spring>>标识的接口是 Spring 所定义的扩展增强接口；带<<aopalliance>>标识的接口则是 AOP 联盟定义的接口。按照增强在目标类方法中的连接点位置，可以分为以下 5 类。

- 前置增强：org.springframework.aop.BeforeAdvice 代表前置增强。因为 Spring 只支持方法级的增强，所以 MethodBeforeAdvice 是目前可用的前置增强，表示在目标方法执行前实施增强，而 BeforeAdvice 是为了将来版本扩展需要而定义的。
- 后置增强：org.springframework.aop.AfterReturningAdvice 代表后置增强，表示在目标方法执行后实施增强。
- 环绕增强：org.aopalliance.intercept.MethodInterceptor 代表环绕增强，表示在目标方法执行前后实施增强。
- 异常抛出增强：org.springframework.aop.ThrowsAdvice 代表抛出异常增强，表示在目标方法抛出异常后实施增强。
- 引介增强：org.springframework.aop.IntroductionInterceptor 代表引介增强，表示在目标类中添加一些新的方法和属性。

这些增强接口都有一些方法，通过实现这些接口方法，并在接口方法中定义横切逻辑，就可以将它们织入目标类方法的相应连接点位置。

7.3.2 前置增强

“热情待客、礼貌服务”已经成为服务行业的基本经营理念，下面通过前置增强对服务生的服务用语进行强制规范。假设服务生只做两件事：第一，欢迎顾客；第二，对顾客提供服务。

1. 保证使用礼貌用语的实例

来看一个保证使用礼貌用语的实例，如代码清单 7-10 所示。

代码清单 7-10 Waiter

```
package com.smart.advice;
public interface Waiter {
    void greetTo(String name);
    void serveTo(String name);
}
```

现在来看一个训练不足的服务生的服务情况，如代码清单 7-11 所示。

代码清单 7-11 NaiveWaiter

```
package com.smart.advice;
public class NaiveWaiter implements Waiter {
    public void greetTo(String name) {
        System.out.println("greet to "+name+"...");
    }
    public void serveTo(String name) {
        System.out.println("serving "+name+"...");
    }
}
```

NaiveWaiter 只是简单地向顾客打招呼，闷不作声地走到顾客跟前，直接提供服务。下面对 NaiveWaiter 的服务行为进行规范，让他们在打招呼和提供服务之前，必须先对顾客使用礼貌用语，如代码清单 7-12 所示。

代码清单 7-12 GreetingBeforeAdvice

```
package com.smart.advice;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class GreetingBeforeAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object obj) throws Throwable { //①
        String clientName = (String)args[0];
        System.out.println("How are you! Mr." + clientName + ".");   在目标类方法
                                                                调用前执行
    }
}
```

BeforeAdvice 是前置增强的接口，方法前置增强的 MethodBeforeAdvice 接口是其子类。Spring 目前只提供方法调用的前置增强，在以后的版本中可能会看到 Spring 提供的其他类型的前置增强，这正是 BeforeAdvice 接口存在的意义。MethodBeforeAdvice 接口仅定义了唯一的方法：before(Method method, Object[] args, Object obj) throws Throwable。其中，method 为目标类的方法；args 为目标类方法的入参；而 obj 为目标类实例。当该方法发生异常时，将阻止目标类方法的执行。

礼貌用语的前置增强制定好后，下面着手强制在服务生队伍中应用这个规定，来看具体的实施情况，如代码清单 7-13 所示。

代码清单 7-13 BeforeAdviceTest

```
package com.smart.advice;
import org.springframework.aop.BeforeAdvice;
import org.springframework.aop.framework.ProxyFactory;
import org.testng.annotations.*;
```

```

public class BeforeAdviceTest {
    @Test
    public void before() {
        Waiter target = new NaiveWaiter();
        BeforeAdvice advice = new GreetingBeforeAdvice();

        //①Spring 提供的代理工厂
        ProxyFactory pf = new ProxyFactory();

        //②设置代理目标
        pf.setTarget(target);

        //③为代理目标添加增强
        pf.addAdvice(advice);

        //④生成代理实例
        Waiter proxy = (Waiter)pf.getProxy();
        proxy.greetTo("John");
        proxy.serveTo("Tom");
    }
}

```

运行上面的代码，可以看到以下输出信息：

```

How are you! Mr.John! ① ← 通过前置增强
greet to John...
How are you! Mr.Tom! ② ← 通过前置增强
serving Tom...

```

引入的礼貌用语
引入的礼貌用语

正如我们期望看到的一样，礼貌待客的优质服务理念得到了坚决、彻底的贯彻。

2. 解剖 ProxyFactory

在 BeforeAdviceTest 中，使用 org.springframework.aop.framework.ProxyFactory 代理工厂将 GreetingBeforeAdvice 的增强织入目标类 NaiveWaiter 中。回想一下，前面介绍的 JDK 和 CGLib 动态代理技术是否有一些相似之处？不错，ProxyFactory 内部就是使用 JDK 或 CGLib 动态代理技术将增强应用到目标类中的。

Spring 定义了 org.springframework.aop.framework.AopProxy 接口，并提供了两个 final 类型的实现类，如图 7-5 所示。

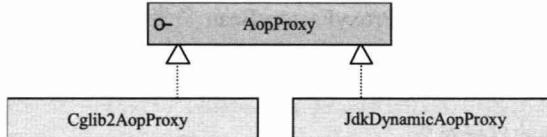


图 7-5 AopProxy 类结构

其中，Cglib2AopProxy 使用 CGLib 动态代理技术创建代理，而 JdkDynamicAopProxy 使用 JDK 动态代理技术创建代理。如果通过 ProxyFactory 的 setInterfaces(Class[] interfaces) 方法指定目标接口进行代理，则 ProxyFactory 使用 JdkDynamicAopProxy；如果是针对类的代理，则使用 Cglib2AopProxy。此外，还可以通过 ProxyFactory 的 setOptimize(true) 方法让 ProxyFactory 启动优化代理方式，这样，针对接口的代理也会使用 Cglib2AopProxy。

值得注意的一点是，在使用 CGLib 动态代理技术时，必须引入 CGLib 类库。

现在回过头来对代码清单 7-13 进行分析。BeforeAdviceTest 使用的是 CGLib 动态代理技术，当我们指定针对接口进行代理时，将使用 JDK 动态代理技术。

```

...
ProxyFactory pf = new ProxyFactory();
pf.setInterfaces(target.getClass().getInterfaces()); //①
pf.setTarget(target);
pf.addAdvice(advice);
...

```

如果指定启用代理优化，则 ProxyFactory 还将使用 Cglib2AopProxy 代理。

```

ProxyFactory pf = new ProxyFactory();
pf.setInterfaces(target.getClass().getInterfaces()); //① ← 指定对接口进行代理
pf.setOptimize(true); //② ← 启用优化
pf.setTarget(target);
pf.addAdvice(advice);

```

读者可能已经注意到，ProxyFactory 通过 addAdvice(Advice)方法添加一个增强，用户可以使用该方法添加多个增强。多个增强形成一个增强链，它们的调用顺序和添加顺序一致，可以通过 addAdvice(int, Advice)方法将增强添加到增强链的具体位置（第一个位置为 0）。

3. 在 Spring 中配置

使用 ProxyFactory 比直接使用 CGLib 或 JDK 动态代理技术创建代理省了很多事，如大家预想的一样，可以通过 Spring 的配置以“很 Spring 的方式”声明一个代理，如代码清单 7-14 所示。

代码清单 7-14 通过ProxyFactoryBean配置代理

```

<bean id="greetingAdvice" class="com.smart.advice.GreetingBeforeAdvice"/>①
<bean id="target" class="com.smart.advice.NaiveWaiter"/> ②
<bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:proxyInterfaces="com.smart.advice.Waiter" ③ ← 指定代理的接口，如果是多个
    p:interceptorNames="greetingAdvice" ④ ← 接口，请使用<list>元素
    p:target-ref="target" ⑤ ← 指定使用的增强 (①处)
/>          指定对哪个 Bean 进行代理 (②处)

```

ProxyFactoryBean 是 FactoryBean 接口的实现类，5.9 节专门介绍了 FactoryBean 的功用，它负责实例化一个 Bean。ProxyFactoryBean 负责为其他 Bean 创建代理实例，它在内部使用 ProxyFactory 来完成这项工作。下面进一步了解一下 ProxyFactoryBean 的几个常用的可配置属性。

- target：代理的目标对象。
- proxyInterfaces：代理所要实现的接口，可以是多个接口。该属性还有一个别名属性 interfaces。
- interceptorNames：需要织入目标对象的 Bean 列表，采用 Bean 的名称指定。这些 Bean 必须是实现了 org.aopalliance.intercept.MethodInterceptor 或 org.springframework.aop.Advisor 的 Bean，配置中的顺序对应调用的顺序。
- singleton：返回的代理是否是单实例，默认为单实例。

- optimize: 当设置为 true 时, 强制使用 CGLib 动态代理。对于 singleton 的代理, 我们推荐使用 CGLib; 对于其他作用域类型的代理, 最好使用 JDK 动态代理。原因是虽然 CGLib 创建代理时速度慢, 但其创建出的代理对象运行效率较高; 而使用 JDK 创建代理的表现正好相反。
- proxyTargetClass: 是否对类进行代理 (而不是对接口进行代理)。当设置为 true 时, 使用 CGLib 动态代理。

运行如代码清单 7-15 所示的测试代码。

代码清单 7-15 测试增强

```
String configPath = "com/smart/advice/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter)ctx.getBean("waiter");
waiter.greetTo("John");
```

输出以下信息:

```
How are you! Mr.John.
greet to John...
```

这时, ProxyFactoryBean 使用了 JDK 动态代理技术。可以调整配置, 使用 CGLib 动态代理技术通过动态创建 NaiveWaiter 的子类来代理 NaiveWaiter 对象, 如下:

```
<bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:interceptorNames="greetingAdvice"
    p:target-ref="target"
    p: proxyTargetClass ="true"/>
...

```

将 proxyTargetClass 设置为 true 后, 无须再设置 proxyInterfaces 属性, 即使设置也会被 ProxyFactoryBean 忽略。

7.3.3 后置增强

后置增强在目标类方法调用后执行。假设服务生在每次服务后也需要使用规范的礼貌用语, 则可以通过一个后置增强来实施这一要求, 如代码清单 7-16 所示。

代码清单 7-16 GreetingAfterAdvice

```
package com.smart.advice;
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
public class GreetingAfterAdvice implements AfterReturningAdvice {
    //①在目标类方法调用后执行
    public void afterReturning(Object returnObj, Method method, Object[] args,
        Object obj) throws Throwable {
        System.out.println("Please enjoy yourself!");
    }
}
```

通过实现 AfterReturningAdvice 来定义后置增强的逻辑, AfterReturningAdvice 接口也仅定义了唯一的方法 afterReturning(Object returnObj, Method method, Object[])

args, Object obj) throws Throwable。其中，returnObj 为目标实例方法返回的结果；method 为目标类的方法；args 为方法的入参；而 obj 为方法的实例。假设在后置增强中抛出异常，如果该异常是目标方法声明的异常，则该异常归并到目标方法中；如果不是目标方法所声明的异常，则 Spring 将其转为运行期异常抛出。

下面将这个后置增强添加到上面的实例中，如代码清单 7-17 所示。

代码清单 7-17 添加后置增强

```
...
<bean id="greetingBefore" class="com.smart.advice.GreetingBeforeAdvice"/>
<bean id="greetingAfter" class="com.smart.advice.GreetingAfterAdvice"/>
<bean id="target" class="com.smart.advice.NaiveWaiter"/>
<bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"
  p:proxyInterfaces="com.smart.advice.Waiter"
  p:target-ref="target"
  p:interceptorNames="greetingBefore,greetingAfter"/>
```

运行代码清单 7-15 中的代码，将输出以下信息：

How are you! Mr.John.① ← 前置增强引入的逻辑
greet_to John...
Please enjoy yourself! ② ← 后置增强引入的逻辑

可见，前置、后置增强中的逻辑都成功地织入目标类 NaiveWaiter 方法所对应的连接点上。



实战经验

interceptorNames 是 String[] 类型的，它接收增强 Bean 的名称而非增强 Bean 的实例。这是因为 ProxyBeanFactory 内部在生成代理类时，需要使用增强 Bean 的类，而非增强 Bean 的实例，以织入增强类中所写的横切逻辑代码，因而可以说增强是类级别的。

对于属性是字符数组类型且数组元素是 Bean 名称的配置，我们最好使用 <idref local="xxx"> 标签，这样在一般的 IDE 环境下编辑 Spring 配置文件时，IDE 会即时发现配置错误并给出报警，以便开发者及早消除配置错误，如下：

```
<property name="interceptorNames">
  <list>
    <idref local="greetingBefore"/>
    <idref local="greetingAfter"/>
  </list>
</property>
```

当然，对于希望尽量简化配置文件的开发者来说，也可以采用逗号、分号或空格分隔的方式进行配置（字符串数组编辑器支持这种配置），如下：

```
<property name="interceptorNames" value="greetingBefore,greetingAfter">
```

7.3.4 环绕增强

介绍完前置、后置增强，环绕增强的作用就显而易见了。环绕增强允许在目标类方

法调用前后织入横切逻辑，它综合实现了前置、后置增强的功能。下面用环绕增强同时实现前礼貌用语和后礼貌用语。

```
package com.smart.advice;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class GreetingInterceptor implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable { //①
        Object[] args = invocation.getArguments(); //目标方法入参
        String clientName = (String)args[0];
        System.out.println("How are you! Mr." + clientName + "."); ②
        Object obj = invocation.proceed(); //③
        System.out.println("Please enjoy yourself!"); //④
        return obj;
    }
}
```

截获目标类方法的执行，并在前后添加横切逻辑

在目标方法执行前调用

通过反射机制调用目标方法

在目标方法执行后调用

Spring 直接使用 AOP 联盟所定义的 MethodInterceptor 作为环绕增强的接口。该接口拥有唯一的接口方法 Object invoke(MethodInvocation invocation) throws Throwable。MethodInvocation 不但封装了目标方法及其入参数组，还封装了目标方法所在的实例对象，通过 MethodInvocation 的 getArguments() 方法可以获取目标方法的入参数组，通过 proceed() 方法反射调用目标实例相应的方法，如③处所示。通过在实现类中定义横切逻辑，可以很容易地实现方法前后的增强。

下面使用环绕增强替换前置和后置增强，如代码清单 7-18 所示。

代码清单 7-18 环绕增强配置

```
<bean id="greetingAround" class="com.smart.advice.GreetingInterceptor"/>
<bean id="target" class="com.smart.advice.NaiveWaiter"/>
<bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:proxyInterfaces="com.smart.advice.Waiter"
    p:target-ref="target"
    p:interceptorNames="greetingAround"/>
```

运行代码清单 7-15 中的代码，将看到以下输出信息：

```
How are you! Mr.John.
greet to John...
Please enjoy yourself!
```

可见，环绕增强达到了前置和后置增强的联合效果。

7.3.5 异常抛出增强

异常抛出增强最适合的应用场景是事务管理，当参与事务的某个 DAO 发生异常时，事务管理器就必须回滚事务。在这里，仅仅给出一个模拟性的实例，用于说明异常抛出增强的使用方法，如代码清单 7-19 所示。

代码清单 7-19 ForumService

```

package com.smart.advice;
import java.sql.SQLException;
public class ForumService {
    public void removeForum(int forumId) {
        // do sth...
        throw new RuntimeException("运行异常。");
    }
    public void updateForum(Forum forum) throws Exception{
        // do sth...
        throw new SQLException("数据更新操作异常。");
    }
}

```

在模拟业务类 ForumService 中定义了两个业务方法，removeForum()抛出运行期异常，而 updateForum()抛出 SQLException。下面试图通过 TransactionManager 这个异常抛出增强对业务方法进行增强处理，统一捕捉抛出的异常并回滚事务，如代码清单 7-20 所示。

代码清单 7-20 TransactionManager

```

package com.smart.advice;
import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;
public class TransactionManager implements ThrowsAdvice {

    //①定义增强逻辑
    public void afterThrowing(Method method, Object[] args, Object target,
        Exception ex) throws Throwable {
        System.out.println("-----");
        System.out.println("method:" + method.getName());
        System.out.println("抛出异常:" + ex.getMessage());
        System.out.println("成功回滚事务。");
    }
}

```

ThrowsAdvice 异常抛出增强接口没有定义任何方法，它是一个标签接口，在运行期 Spring 使用反射机制自行判断，必须采用以下签名形式定义异常抛出的增强方法：

```
void afterThrowing([Method method, Object[] args, Object target], Throwable);
```

方法名必须为 afterThrowing，方法入参规定如下：前 3 个入参 Method method、Object[] args、Object target 是可选的（3 个入参数要么提供，要么不提供），而最后一个入参是 Throwable 或其子类。如以下方法都是合法的：

- afterThrowing(SQLException e)。
 - afterThrowing(RuntimeException e)。
 - afterThrowing(Method method, Object[] args, Object target, RuntimeException e)。
- 而以下方法是非法的：
- afterThrowing(Object[] args, Object target, RuntimeException e)：缺少 Method。
 - solveThrowing(SQLException e)：方法名非法。

可以在同一个异常抛出增强中定义多个 afterThrowing()方法，当目标类方法抛出异常时，Spring 会自动选用最匹配的增强方法。假设在增强中定义了两个方法：

- afterThrowing(SQLException e)。
- afterThrowing(Throwable e)。

当目标方法抛出一个 SQLException 时，将调用 afterThrowing(SQLException e)而非 afterThrowing(Throwable e)进行增强。在类的继承树上，两个类的距离越近，就说这两个类的相似度越高。目标方法抛出异常后，优先选取拥有异常入参和抛出的异常相似度最高的 afterThrowing()方法。



提示

标签接口是没有任何方法和属性的接口，它不对实现类有任何语义上的要求，仅仅表明它的实现类属于一个特定的类型。它非常类似于 Web 2.0 中 TAG 的概念，Java 使用它标识某一类对象。它主要有两个用途：第一，通过标签接口标识同一类型的类，这些类本身可能并不具有相同的方法，如 Advice 接口；第二，通过标签接口使程序或 JVM 采取一些特殊处理，如 java.io.Serializable，它告诉 JVM 对象可以被序列化。

在 Spring 中对这个异常抛出增强进行配置，如下：

```
<bean id="transactionManager" class="com.smart.advice.TransactionManager"/>
<bean id="forumServiceTarget" class="com.smart.advice.ForumService"/>
<bean id="forumService" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:interceptorNames="transactionManager"
    p:target-ref="forumServiceTarget"          因 ForumService 是类,
    p:proxyTargetClass="true"/>①             使用 CGLib 代理
```

采用类似于代码清单 7-15 的代码测试这个异常抛出增强，将得到以下输出信息：

```
-----
method:removeForum
抛出异常：运行异常。
成功回滚事务。
-----
method:updateForum
抛出异常：数据更新操作异常。
成功回滚事务。
```

可见，ForumService 的两个方法所抛出的异常都被 TransactionManager 这个异常抛出增强捕获并成功处理。这样 ForumService 就从事务管理繁复的代码中解放出来，历史揭开了崭新的一页！

7.3.6 引介增强

引介增强是一种比较特殊的增强类型，它不是在目标方法周围织入增强，而是为目标类创建新的方法和属性，所以引介增强的连接点是类级别的，而非方法级别的。通过引介增强，可以为目标类添加一个接口的实现，即原来目标类未实现某个接口，通过引

介增强可以为目标类创建实现某接口的代理。这种功能富有吸引力，因为它能够在横向 上定义接口的实现方法，思考问题的角度发生了很大的变化。

Spring 定义了引介增强接口 `IntroductionInterceptor`，该接口没有定义任何方法， Spring 为该接口提供了 `DelegatingIntroductionInterceptor` 实现类。一般情况下，通过扩展 该实现类定义自己的引介增强类。

回到本章前面性能监视的例子，我们对所有的业务类都织入了性能监视的增强。由于 性能监视会影响业务系统的性能，所以是否启用性能监视应该是可控的，即维护人员 可以手工打开或关闭性能监视的功能。但原来的例子只简单地添加了运行性能监视逻 辑，未提供任何控制的功能，现在可以用引介增强来实现这一诱人的功能。

首先定义一个用于标识目标类是否支持性能监视的接口，如代码清单 7-21 所示。

代码清单 7-21 Monitorable

```
package com.smart.introduce;
public interface Monitorable {
    void setMonitorActive(boolean active);
}
```

该接口仅包括一个 `setMonitorActive(boolean active)` 方法，我们期望通过该接口方法 控制业务类性能监视功能的激活和关闭状态。

下面通过扩展 `DelegatingIntroductionInterceptor` 为目标类引入性能监视的可控功能， 如代码清单 7-22 所示。

代码清单 7-22 ControllablePerformanceMonitor

```
package com.smart.introduce;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.support.DelegatingIntroductionInterceptor;
public class ControllablePerformanceMonitor
    extends DelegatingIntroductionInterceptor
    implements Monitorable {
    private ThreadLocal<Boolean> MonitorStatusMap =new ThreadLocal <Boolean>();①
    public void setMonitorActive(boolean active) {②
        MonitorStatusMap .set(active);
    }
    //③拦截方法
    public Object invoke(MethodInvocation mi) throws Throwable {
        Object obj = null;
        //④对于支持性能监视可控代理，通过判断其状态决定是否开启性能监控功能
        if (MonitorStatusMap.get() != null && MonitorStatusMap.get()) {
            PerformanceMonitor.begin(mi.getClass().getName() + "."
                + mi.getMethod().getName());
            obj = super.invoke(mi);
            PerformanceMonitor.end();
        } else {
            obj = super.invoke(mi);
        }
    }
}
```

```

        return obj;
    }
}

```

ControllablePerformanceMonitor 在扩展 DelegatingIntroductionInterceptor 的同时，还必须实现 Monitorable 接口，提供接口方法的实现。在①处定义了一个 ThreadLocal 类型的变量，用于保存性能监视开关状态。之所以使用 ThreadLocal 变量，是因为这个控制状态使代理类变成了非线程安全的实例，为了解决单实例线程安全的问题，通过 ThreadLocal 让每个线程单独使用一个状态。

在③处覆盖了父类中的 invoke()方法，该方法用于拦截目标类方法的调用，根据监视开关的状态有条件地对目标实例方法进行性能监视。④处的粗体代码所示部分可能有点难以理解，它使用了 Java 5.0 的自动拆包功能，MonitorStatusMap.get()方法返回的 Boolean 被自动拆包为 boolean 类型的值。

下面通过 Spring 的配置，将这个引介增强织入业务类 ForumService 中，具体配置如代码清单 7-23 所示。

代码清单 7-23 配置引介增强

```

<bean id="pmonitor" class="com.smart.introduce.Controllable
PerformanceMonitor"/>
<bean id="forumServiceTarget" class="com.smart.introduce.ForumService"/>
<bean id="forumService" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:interfaces="com.smart.introduce.Monitorable" ① ← 引介增强所实现的接口
    p:target-ref="forumServiceTarget"
    p:interceptorNames="pmonitor"
    p:proxyTargetClass="true" />② ← 由引介增强一定要通过创建子类
                                来生成代理，所以需要强制使用
                                CGLib，否则会报错

```

引介增强的配置与一般的配置有较大的区别：首先，需要指定引介增强所实现的接口，如①处所示，这里的引介增强实现了 Monitorable 接口；其次，由于只能通过为目标类创建子类的方式生成引介增强的代理，所以必须将 proxyTargetClass 设置为 true。

如果没有对 ControllablePerformanceMonitor 进行线程安全的特殊处理，就必须将 singleton 属性设置为 true，让 ProxyFactoryBean 产生 prototype 作用域类型的代理。这就带来了一个严重的性能问题，因为 CGLib 动态创建代理的性能很低，而每次通过 getBean() 方法从容器中获取作用域类型为 prototype 的 Bean 时都将返回一个新的代理实例，所以这种性能的影响是巨大的，这也是为什么在代码中通过 ThreadLocal 对 ControllablePerformanceMonitor 的开关状态进行线程安全化处理的原因。通过线程安全化处理后，就可以使用默认的 singleton Bean 作用域，这样创建代理的动作仅发生一次。

代码清单 7-24 所示的代码对织入性能监视控制接口业务类方法的调用情况进行测试。

代码清单 7-24 IntroduceTest：测试引介增强

```

package com.smart.introduce;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.testng.annotations.*;

```

```

public class IntroduceTest {
    @Test
    public void introduce() {
        String configPath = "com/smart/introduce/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        ForumService forumService = (ForumService)ctx.getBean("forumService");
        // 默认情况下，未开启性能监视功能
        forumService.removeForum(10); | ① ↘
        forumService.removeTopic(1022); | ② ↗ 开启性能监视功能
        Monitorable moniterable = (Monitorable)forumService; | ③ ↗ 在性能监视功能开启的情况下，再次调用业务方法
        moniterable.setNeedMonitor(true);

        forumService.removeForum(10); | ③ ↗ 在性能监视功能开启的情况下，再次调用业务方法
        forumService.removeTopic(1022);
    }
}

```

注意②处的(Monitorable)forumService 代码，强制性地将 forumService 转换为 Monitorable 类型。代码的成功执行表示从 Spring 容器中返回的代理确实引入了 Monitorable 接口方法的实现。

执行以上代码，可以看到以下输出信息：

```

模拟删除Forum记录:10 ① ← 未激活监视功能
模拟删除Topic记录:1022

begin monitor... ② ← 激活监视功能
模拟删除Forum记录:10
end monitor...
org.springframework.aop.framework.Cglib2AopProxy$CglibMethodInvocation. removeForum
花费47毫秒。

begin monitor...
模拟删除Topic记录:1022
end monitor...
org.springframework.aop.framework.Cglib2AopProxy$CglibMethodInvocation. removeTopic
花费16毫秒。

```

在①处，只有业务逻辑被执行，性能监视功能没有被执行；而在②处，性能监视功能正常启用，两个业务方法都启用了性能监视功能。



提示

在 Spring 4.0 中，基于 CGLib 的类代理不再要求目标类必须有无参构造函数。这是一个不错的特性，这样在使用 CGLib 类时，不再需要特别关注目标类是否有无参构造函数。取消这个限制后，增强的目标 Bean 就可以使用构造函数注入了。Spring 到底如何实现这个功能？这就要归根于 Spring 内联了 objenesis 类库，感兴趣的读者可到其官网 (<http://objenesis.org>) 查看。

7.4 创建切面

在介绍增强时，读者可能会注意到一个问题：增强被织入目标类的所有方法中。假设我们希望有选择地织入目标类的某些特定方法中，就需要使用切点进行目标连接点的定位。描述连接点是进行 AOP 编程最主要的工作，为了突出强调这一点，再次给出 Spring AOP 如何定位连接点。

增强提供了连接点方位信息，如织入到方法前面、后面等，而切点进一步描述了织入哪些类的哪些方法上。

Spring 通过 org.springframework.aop.Pointcut 接口描述切点，Pointcut 由 ClassFilter 和 MethodMatcher 构成，它通过 ClassFilter 定位到某些特定类上，通过 MethodMatcher 定位到某些特定方法上，这样 Pointcut 就拥有了描述某些类的某些特定方法的能力。可以简单地用 SQL 复合查询条件来理解 Pointcut 的功用。Pointcut 类关系图如图 7-6 所示。

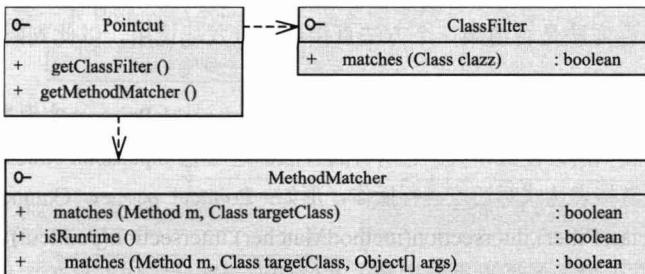


图 7-6 Pointcut 类关系图

可以看到 ClassFilter 只定义了一个方法 matches(Class clazz)，其参数代表一个被检测类，该方法判别被检测的类是否匹配过滤条件。

Spring 支持两种方法匹配器：静态方法匹配器和动态方法匹配器。所谓静态方法匹配器，仅对方法名签名（包括方法名和入参类型及顺序）进行匹配；而动态方法匹配器会在运行期检查方法入参的值。静态匹配仅会判别一次，而动态匹配因为每次调用方法的入参都可能不一样，所以每次调用方法都必须判断，因此，动态匹配对性能的影响很大。一般情况下，动态匹配不常使用。方法匹配器的类型由 isRuntime() 方法的返回值决定，返回 false 表示是静态方法匹配器，返回 true 表示是动态方法匹配器。

此外，Spring 2.0 还支持注解切点和表达式切点，前者通过 Java 5.0 的注解定义切点，而后者通过字符串表达式定义切点，二者都使用 AspectJ 的切点表达式语言。

7.4.1 切点类型

Spring 提供了 6 种类型的切点，下面分别对它们的用途进行介绍。

- 静态方法切点：org.springframework.aop.support.StaticMethodMatcherPointcut 是静态方法切点的抽象基类，默认情况下它匹配所有的类。StaticMethodMatcherPointcut 包括两个主要的子类，分别是 NameMatchMethodPointcut 和 AbstractRegexpMethodPointcut，前者提供简单字符串匹配方法签名，而后者使用正则表达式匹配方法签名。
- 动态方法切点：org.springframework.aop.support.DynamicMethodMatcherPointcut 是动态方法切点的抽象基类，默认情况下它匹配所有的类。
- 注解切点：org.springframework.aop.support.annotation.AnnotationMatchingPointcut 实现类表示注解切点。使用 AnnotationMatchingPointcut 支持在 Bean 中直接通过 Java 5.0 注解标签定义的切点。
- 表达式切点：org.springframework.aop.support.ExpressionPointcut 接口主要是为了支持 AspectJ 切点表达式语法而定义的接口。
- 流程切点：org.springframework.aop.support.ControlFlowPointcut 实现类表示控制流程切点。ControlFlowPointcut 是一种特殊的切点，它根据程序执行堆栈的信息查看目标方法是否由某一个方法直接或间接发起调用，以此判断是否为匹配的连接点。
- 复合切点：org.springframework.aop.support.ComposablePointcut 实现类是为创建多个切点而提供的方便操作类。它所有的方法都返回 ComposablePointcut 类，这样就可以使用链接表达式对切点进行操作，形如：Pointcut pc=new ComposablePointcut().union(classFilter).intersection(methodMatcher).intersection(pointcut)。

本章仅对其中的 4 类切点进行讲解，注解切点和表达式切点将在下一章讲解。

7.4.2 切面类型

由于增强既包含横切代码，又包含部分连接点信息（方法前、方法后主方位信息），所以可以仅通过增强类生成一个切面。但切点仅代表目标类连接点的部分信息（类和方法的定位），所以仅有切点无法制作出一个切面，必须结合增强才能制作出切面。Spring 使用 org.springframework.aop.Advisor 接口表示切面的概念，一个切面同时包含横切代码和连接点信息。切面可以分为 3 类：一般切面、切点切面和引介切面，可以通过 Spring 所定义的切面接口清楚地了解切面的分类，如图 7-7 所示。

- Advisor：代表一般切面，仅包含一个 Advice。因为 Advice 包含了横切代码和连接点信息，所以 Advice 本身就是一个简单的切面，只不过它代表的横切的连接点是所有目标类的所有方法，因为这个横切面太宽泛，所以一般不会直接使用。
- PointcutAdvisor：代表具有切点的切面，包含 Advice 和 Pointcut 两个类，这样就可以通过类、方法名及方法方位等信息灵活地定义切面的连接点，提供更具适用性的切面。

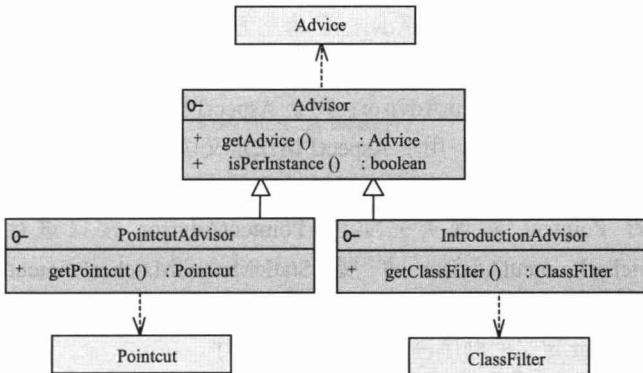


图 7-7 切面类继承关系

- **IntroductionAdvisor:** 代表引介切面。7.3.6 节介绍了引介增强类型，引介切面是对应引介增强的特殊的切面，它应用于类层面上，所以引介切点使用 ClassFilter 进行定义。

下面再来看一下 PointcutAdvisor 的主要实现类体系，如图 7-8 所示。

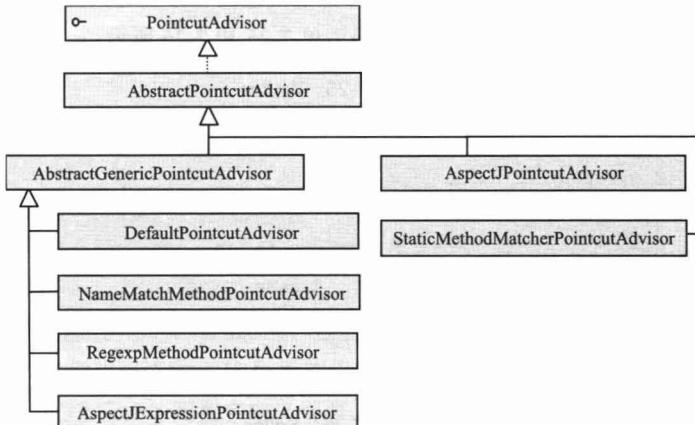


图 7-8 PointcutAdvisor 实现类体系

PointcutAdvisor 主要有 6 个具体的实现类，分别介绍如下。

- **DefaultPointcutAdvisor:** 最常用的切面类型，它可以通过任意 Pointcut 和 Advice 定义一个切面，唯一不支持的是引介的切面类型，一般可以通过扩展该类实现自定义的切面。
- **NameMatchMethodPointcutAdvisor:** 通过该类可以定义按方法名定义切点的切面。
- **RegexpMethodPointcutAdvisor:** 对于按正则表达式匹配方法名进行切点定义的切面，可以通过扩展该实现类进行操作。RegexpMethodPointcutAdvisor 允许用户以正则表达式模式串定义方法匹配的切点，其内部通过 JdkRegexpMethodPointcut 构造出正则表达式方法名切点。

- **StaticMethodMatcherPointcutAdvisor**: 静态方法匹配器切点定义的切面，默认情况下匹配所有的目标类。
- **AspectJExpressionPointcutAdvisor**: 用于 AspectJ 切点表达式定义切点的切面。
- **AspectJPointcutAdvisor**: 用于 AspectJ 语法定义切点的切面。

这些 Advisor 的实现类都可以在 Pointcut 中找到对应物，实际上，它们都是通过扩展对应的 Pointcut 实现类并实现 PointcutAdvisor 接口进行定义的。如 StaticMethodMatcherPointcutAdvisor 扩展 StaticMethodMatcherPointcut 类并实现 PointcutAdvisor 接口。此外，Advisor 都实现了 org.springframework.core.Ordered 接口，Spring 将根据 Advisor 定义的顺序决定织入切面的顺序。

在了解了切点和切面的知识后，下面将通过具体实例进一步了解它们的具体用法。

7.4.3 静态普通方法名匹配切面

StaticMethodMatcherPointcutAdvisor 代表一个静态方法匹配切面，它通过 StaticMethodMatcherPointcut 来定义切点，并通过类过滤和方法名来匹配所定义的切点。来看下面的 Waiter 和 Seller 业务类，如代码清单 7-25 和 7-26 所示。

代码清单 7-25 Waiter

```
package com.smart.advisor;
public class Waiter {
    public void greetTo(String name) {
        System.out.println("waiter greet to "+name+"...");
    }
    public void serveTo(String name){
        System.out.println("waiter serving "+name+"...");
    }
}
```

Waiter 有两个方法，分别是 greetTo() 和 serveTo()。

代码清单 7-26 Seller

```
package com.smart.advisor;
public class Seller {
    public void greetTo(String name) {
        System.out.println("seller greet to "+name+"...");
    }
}
```

Seller 拥有一个和 Waiter 相同名称的方法 greetTo()。现在，我们希望通过 StaticMethodMatcherPointcutAdvisor 定义一个切面，在 Waiter#greetTo() 方法调用前织入一个增强，即连接点为 Waiter#greetTo() 方法调用前的位置。具体的切面类的实现如代码清单 7-27 所示。

代码清单 7-27 GreetingAdvisor

```
package com.smart.advisor;
import java.lang.reflect.Method;
```

```

import org.springframework.aop.ClassFilter;
import org.springframework.aop.support.StaticMethodMatcherPointcutAdvisor;
public class GreetingAdvisor extends StaticMethodMatcherPointcutAdvisor {
    public boolean matches(Method method, Class clazz) { //① ← 切点方法匹配规则:
        return "greetTo".equals(method.getName());                                方法名为greetTo
    }
    public ClassFilter getClassFilter() { //② ← 切点类匹配规则:为
        return new ClassFilter() {                                              Waiter 的类或子类
            public boolean matches(Class clazz) {
                return Waiter.class.isAssignableFrom(clazz);
            }
        };
    }
}

```

StaticMethodMatcherPointcutAdvisor 抽象类唯一需要定义的是 matches()方法。在默认情况下，该切面匹配所有的类，这里通过覆盖 getClassFilter()方法，让它仅匹配 Waiter 类及其子类。

当然，Advisor 还需要一个增强类的配合，我们定义一个前置增强，如代码清单 7-28 所示。

代码清单 7-28 GreetingBeforeAdvice

```

package com.smart.advisor;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class GreetingBeforeAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object obj) throws Throwable {
        System.out.println(obj.getClass().getName()+"."+method.getName()); //①
        String clientName = (String)args[0];
        System.out.println("How are you! Mr."+clientName+".");
    }
}

```

输出切点

当然，可以直接使用 ProxyFactory，通过手工编码的方式织入切面生成代理类，有兴趣的读者可以参考代码清单 7-13 所示的代码，在③处使用 addAdvisor()方法添加切面就可以了。下面使用 Spring 配置来定义切面，如代码清单 7-29 所示。

代码清单 7-29 配置切面：静态方法匹配切面

```

<bean id="waiterTarget" class="com.smart.advisor.Waiter"/>
<bean id="sellerTarget" class="com.smart.advisor.Seller"/>
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice"/>
<bean id="greetingAdvisor" class="com.smart.advisor.GreetingAdvisor"
      p:advice-ref="greetingAdvice"/>① ← 向切面注入一个前置增强
                                         通过一个父<bean>
<bean id="parent" abstract="true" ② ← 定义公共的配置信息
      class="org.springframework.aop.framework.ProxyFactoryBean"
      p:interceptorNames="greetingAdvisor"
      p:proxyTargetClass="true"/>
<bean id="waiter" parent="parent" p:target-ref="waiterTarget"/>③ ← waiter 代理
<bean id="seller" parent="parent" p:target-ref="sellerTarget"/>④ ← seller 代理

```

在①处，将 greetingAdvice 增强装配到 greetingAdvisor 切面中。StaticMethodMatcherPointcutAdvisor 除具有 advice 属性外，还可以定义另外两个属性。

- classFilter: 类匹配过滤器，在 GreetingAdvisor 中用编码的方式设定了 classFilter。
- order: 切面织入时的顺序，该属性用于定义 Ordered 接口表示的顺序。

由于需要分别为 waiter 和 seller 两个 Bean 定义代理器，且二者有很多公共的配置信息，所以使用了一个父<bean>简化配置，如②处所示。在③和④处，通过引用父<bean>轻松地定义了两个织入切面的代理。

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter)ctx.getBean("waiter");
Seller seller = (Seller)ctx.getBean("seller");
waiter.greetTo("John");
waiter.serveTo("John");
seller.greetTo("John");
```

运行以上代码，输出以下信息：

```
com.smart.advisor.Waiter.greetTo ①
How are you! Mr.John.
waiter greet to John...
waiter serving John...
seller greet to John...
```

这两行为切面
引入的增强逻辑

可见切面只织入 Waiter.greetTo() 方法调用前的连接点上，Waiter.serveTo() 和 Seller.greetTo()方法没有织入切面。

7.4.4 静态正则表达式方法匹配切面

在 StaticMethodMatcherPointcutAdvisor 中，仅能通过方法名定义切点，这种描述方式不够灵活。假设目标类中有多个方法，且它们都满足一定的命名规范，使用正则表达式进行匹配描述就要灵活多了。RegexpMethodPointcutAdvisor 是正则表达式方法匹配的切面实现类，该类已经是功能齐备的实现类，一般情况下无须扩展该类。

1. 具体实例

下面直接使用 RegexpMethodPointcutAdvisor，通过配置的方式为 Waiter 目标类定义一个切面，如代码清单 7-30 所示。

代码清单 7-30 通过正则表达式定义切面

```
<bean id="regexpAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
      p:advice-ref="greetingAdvice">
    <property name="patterns">①
      <list>
        <value>.*greet.*</value>②
      </list>
    </property>
</bean>
<bean id="waiter1" class="org.springframework.aop.framework.ProxyFactoryBean"
      p:interceptorNames="regexpAdvisor"
      p:target-ref="waiterTarget"
      p:proxyTargetClass="true"/>
```

用正则表达式定义目标类全
限定方法名的匹配模式串
匹配模式串

在②处定义了一个匹配模式串“.*greet.*”，该模式串匹配 Waiter.greetTo()方法。值得注意的是，匹配模式串匹配的是目标类方法的全限定名，即带类名的方法名。

运行下列测试代码：

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter)ctx.getBean("waiter1");
waiter.greetTo("John");
waiter.serveTo("John");
```

输出以下信息：

```
com.smart.advisor.Waiter.greetTo ①
How are you! Mr.John.
waiter greet to John...
waiter serving John...
```

这两行为切面
引入的增强逻辑

可见，Waiter.greetTo()方法被织入了切面，而 Waiter.serveTo()方法没有被织入切面。除了例子中所使用的 patterns 和 advice 属性外，还有另外两个属性，分别介绍如下。

- pattern：如果只有一个匹配模式串，则可以使用该属性进行配置。patterns 属性用于定义多个匹配模式串，这些匹配模式串之间是“或”的关系。
- order：切面在织入时对应的顺序。

2. 正则表达式语法

正则表达式的语法内容较多，这里仅对常见的正则表达式知识进行简单介绍（见表 7-1），相信这些知识足以应付 AOP 配置的日常所需。

表 7-1 正则表达式符号

符 号	说 明	实 例
.	匹配除换行符外的所有单个的字符	.n 匹配 nay, an apple is on the tree 中的 an 和 on, 但不匹配 nay
*	匹配*前面的字符 0 次或 n 次	bo* 匹配 A ghost booooed 中的 boooo 或 A bird warbled 中的 b, 但不匹配 Agoat g runted 中的任何字符
+	匹配+前面的字符 1 次或 n 次。等价于{1,}	a+ 匹配 candy 中的 a 和 caaaaaandy. 中的所有 a
^	表示匹配的字符必须在最前边	^A 不匹配 an A 中的 A, 但匹配 An A. 中最前面的 A
\$	与^类似，匹配最末的字符	t\$ 不匹配 eater 中的 t, 但匹配 eat 中的 t
?	匹配?前面的字符 0 次或 1 次	e?le? 匹配 angel 中的 el 和 angle 中的 le
xy	匹配 x 或者 y	green red 匹配 green apple 中的 green 和 red apple. 中的 red
[xyz]	一张字符列表，匹配列表中的任一字符。 可以通过连字符“-”指出一个字符范围	[abc] 跟 [a-c] 一样。它们匹配 brisket 中的 b 及 ache 中的 a 和 c
{n}	这里的 n 是一个正整数。匹配前面的 n 个字符	a{2} 不匹配 candy 中的 a, 但匹配 caandy 中的两个 a
{n,}	这里的 n 是一个正整数。匹配至少 n 个前面的字符	a{2,} 不匹配 candy 中的 a, 但匹配 caandy 中的所有 a 和 caaaaaandy. 中的所有 a
{n,m}	这里的 n 和 m 都是正整数。匹配至少 n 个最多 m 个前面的字符	a{1,3} 不匹配 cndy 中的任何字符, 但匹配 candy 中的 a 和 caandy 中的前面两个 a 和 caaaaaandy. 中前面的 3 个 a。注意, 即使 caaaaaandy. 中有很多个 a, 但只匹配前面的 3 个 a, 即 aaa

续表

符 号	说 明	实 例
\	将下一个字符标记为一个特殊字符	例如, n 匹配字符 n。\\n 匹配一个换行符。语法中的特殊字符需要通过转义符表示, 如\\.表示., 而\\{表示{
转义字符		
\d	匹配一个数字字符。等价于 [0-9]	
\D	匹配一个非数字字符。等价于 [^0-9]	
\f	匹配一个换页符。等价于\x0c 和\cL	
\n	匹配一个换行符。等价于\x0a 和\cJ	
\r	匹配一个回车符。等价于\x0d 和\cM	
\s	匹配任何空白字符, 包括空格、制表符、换页符等。等价于[\f\n\r\t\v]	
\S	匹配任何非空白字符。等价于[^ \f\n\r\t\v]	
\t	匹配一个制表符。等价于\x09 和\cI	
\v	匹配一个垂直制表符。等价于\x0b 和\cK	
\w	匹配包括下划线的任何单词字符。等价于[A-Za-z0-9_]	
\W	匹配任何非单词字符。等价于[^A-Za-z0-9_]	

下面举几个例子, 进一步认识正则表达式在配置匹配方法上的具体应用。

示例 1: .*set.*表示所有类中以 set 为前缀的方法, 如 com.smart.Waiter.setSalary()、Person.setName()等。

示例 2: com\smart\advisor\..* 表示 com.smart.advisor 包下所有类的所有方法。

示例 3: com\smart\service\..*Service\..* 匹配 com.smart.service 包下所有类名以 Service 结尾的类的所有方法, 如 com.smart.service.UserService.save(User user)、com.smart.service.ForumService.update(Forum forum)等。

示例 4: com\smart\service\..*\..save.+ 匹配所有以 save 为前缀的方法, 该方法后还必须拥有至少一个字符, 且这些方法位于 com.smart.service 包中以 Service 为后缀的类中。如匹配 com.smart.service.UserService 类的 saveUser()和 saveLoginLog()方法, 但不匹配该类的 save()方法。

只要程序的类包具有良好的命名规范, 就可以使用简单的正则表达式描述出目标方法。由于需要使用全限定名来定义方法名, 所以不但方法名需要具有良好的规范性, 包名也需要具体良好的规范性。对包名、类名、方法名按其功用进行规范命名并不是一件坏事, 相反, 规范命名可以增强程序的可读性和团队开发的协作性, 降低沟通成本, 是值得实践和提倡的编程方法。



实战经验

在编写正则表达式时, 通过一些好用的工具可以取得事半功倍的效果。RegexBuddy 是使用正则表达式时最好的助手, 借助这款小巧的工具可以很容易地创建符合用户要求的正则表达式。读者可以通过 <http://www.regexbuddy.com> 了解更多关于这款软件的信息。

7.4.5 动态切面

在低版本中，Spring 提供了用于创建动态切面的 DynamicMethodMatcherPointcut Advisor 抽象类，因为该类在功能上和其他类有重叠，会给开发者造成选择上的困惑，因此在 Spring 2.0 中已经过期。可以使用 DefaultPointcutAdvisor 和 DynamicMethod MatcherPointcut 来完成相同的功能。

DynamicMethodMatcherPointcut 是一个抽象类，它将 isRuntime() 标识为 final 且返回 true，这样其子类就一定是一个动态切点。该抽象类默认匹配所有的类和方法，因此需要通过扩展该类编写符合要求的动态切点，如代码清单 7-31 所示。

代码清单 7-31 GreetingDynamicPointcut

```
package com.smart.advisor;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;
import org.springframework.aop.ClassFilter;
import org.springframework.aop.support.DynamicMethodMatcherPointcut;
public class GreetingDynamicPointcut extends DynamicMethodMatcherPointcut {
    private static List<String> specialClientList = new ArrayList<String>();
    static {
        specialClientList.add("John");
        specialClientList.add("Tom");
    }
    public ClassFilter getClassFilter() { //① 对类进行静态切点检查
        return new ClassFilter() {
            public boolean matches(Class clazz) {
                System.out.println("调用getClassFilter()对" + clazz.getName() + "做静态检查.");
                return Waiter.class.isAssignableFrom(clazz);
            }
        };
    }
    public boolean matches(Method method, Class clazz) { //② 对方法进行静态切点检查
        System.out.println("调用matches(method,clazz)" + clazz.getName() + "."
                           + method.getName() + "做静态检查.");
        return "greetTo".equals(method.getName());
    }
    public boolean matches(Method method, Class clazz, Object[] args) { //③ 对方法进行动态切点检查
        System.out.println("调用matches(method,clazz)" + clazz.getName() + "."
                           + method.getName() + "做动态检查.");
        String clientName = (String) args[0];
        return specialClientList.contains(clientName);
    }
}
```

GreetingDynamicPointcut 类既可用于静态切点检查的方法，又有用于动态切点检查的方法。由于动态切点检查会对性能造成很大的影响，所以应当尽量避免在运行时每次都对目标类的各个方法进行动态检查。Spring 采用这样的机制：在创建代理时对目标类的每个连接点使用静态切点检查，如果仅通过静态切点检查就可以知道连接点是不匹配

的，则在运行时就不再进行动态检查；如果静态切点检查是匹配的，则在运行时才进行动态切点检查。

在动态切点类中定义静态切点检查的方法可以避免不必要的动态检查操作，从而极大地提高运行效率，这一点在稍后的运行测试中再作进一步分析。

在 GreetingDynamicPointcut 类中，在③处通过 matches(Method method, Class clazz, Object[] args) 定义了动态切点检查的方法，只对目标方法为 greetTo(clientName) 且 clientName 为特殊客户的方法启用增强，通过 specialClientList 模拟特殊的客户名单。

在编写好动态切点后，就可以着手在 Spring 配置文件中装配出一个动态切面，如代码清单 7-32 所示。

代码清单 7-32 动态切面配置

```
<bean id="waiterTarget" class="com.smart.advisor.Waiter"/>
<bean id="dynamicAdvisor"
      class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut">
      <bean class="com.smart.advisor.GreetingDynamicPointcut"/>①
    </property>
    <property name="advice">
      <bean class="com.smart.advisor.GreetingBeforeAdvice"/>
    </property>
  </bean>
<bean id="waiter2" class="org.springframework.aop.framework.ProxyFactoryBean"
      p:interceptorNames="dynamicAdvisor"
      p:target-ref="waiterTarget"
      p:proxyTargetClass="true"/>
```

动态切面的配置和静态切面的配置没有什么区别。使用 DefaultPointcutAdvisor 定义切面，在①处使用内部 Bean 方式注入动态切点 GreetingDynamicPointcut，增强依旧使用前面定义的 GreetingBeforeAdvice。此外，DefaultPointcutAdvisor 还有一个 order 属性，用于定义切面的织入顺序。

一切准备就绪后，开始编写一个测试类，这样就可以看到动态切面的“庐山真面目”了，如代码清单 7-33 所示。

代码清单 7-33 动态切面测试代码

```
package com.smart.advisor;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.testng.annotations.*;
public class DynamicAdvisorTest {

    @Test
    public void dynamic() {
        String configPath = "com/smart/advisor/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter) ctx.getBean("waiter2");
        waiter.serveTo("Peter");
        waiter.greetTo("Peter");
        waiter.serveTo("John");
    }
}
```

```

        waiter.greetTo("John"); //① ←
    }
}

```

John 是特殊客户

运行以上代码，在控制台上输出以下信息：

以下 10 行输出信息反映了在织入切面前 Spring 对目标类中所有方法进行的静态切点检查。
 ① ← 调用 getClassFilter() 对 com.smart.advisor.Waiter 做静态检查。
 调用 matches(method, clazz) 对 com.smart.advisor.Waiter.serveTo 做静态检查。
 调用 getClassFilter() 对 com.smart.advisor.Waiter 做静态检查。
 调用 matches(method, clazz) 对 com.smart.advisor.Waiter.greetTo 做静态检查。
 调用 getClassFilter() 对 com.smart.advisor.Waiter 做静态检查。
 调用 matches(method, clazz) 对 com.smart.advisor.Waiter.hashCode 做静态检查。
 调用 getClassFilter() 对 com.smart.advisor.Waiter 做静态检查。
 调用 matches(method, clazz) 对 com.smart.advisor.Waiter.clone 做静态检查。
 调用 getClassFilter() 对 com.smart.advisor.Waiter 做静态检查。
 调用 matches(method, clazz) 对 com.smart.advisor.Waiter.toString 做静态检查。

② ← 对应 waiter.serveTo("Peter")：第一次
 调用 serveTo() 方法时，执行静态切点检查
 调用 getClassFilter() 对 com.smart.advisor.Waiter 做静态检查。
 调用 matches(method, clazz) 对 com.smart.advisor.Waiter.serveTo 做静态检查。
 waiter serving Peter...

③ ← 以下 4 行对应 waiter.greetTo("Peter")：第一次
 调用 greetTo() 方法时，执行静态、动态切点检查
 调用 getClassFilter() 对 com.smart.advisor.Waiter 做静态检查。
 调用 matches(method, clazz) 对 com.smart.advisor.Waiter.greetTo 做静态检查。
 调用 matches(method, clazz) 对 com.smart.advisor.Waiter.greetTo 做动态检查。
 waiter greet to Peter...

④ ← 对应 waiter.serveTo("John")：第二次调用
 serveTo() 方法时，不再执行静态切点检查
 waiter serving John...

⑤ ← 以下 4 行对应 waiter.greetTo("John")：第二
 次调用 greetTo() 方法时，只执行动态切点检查
 调用 matches(method, clazz) 对 com.smart.advisor.Waiter.greetTo 做动态检查。
 com.smart.advisor.Waiter.greetTo
 How are you! Mr.John.
 waiter greet to John...

通过以上输出信息，对照 DynamicMethodMatcherPointcut 切点类，可以很容易发现，Spring 会在创建代理织入切面时，对目标类中的所有方法进行静态切点检查；在生成织入切面的代理对象后，第一次调用代理类的每一个方法时都会进行一次静态切点检查，如果本次检查就能从候选者列表中将该方法排除，则以后对该方法的调用就不再执行静态切点检查；对于那些在静态切点检查时匹配的方法，在后续调用该方法时，将执行动态切点检查。

在例子中，切点匹配的规则是：目标类为 com.smart.advisor.Waiter 或其子类；方法名为 greetTo；动态入参 clientName 必须是特殊名单中的客户。

基于这条规则，serveTo() 及从 Object 中继承而来的 toString()、hashCode() 和 clone() 等方法通过静态切点检查就可以排除在候选者之外，只有 greetTo() 方法是动态切点检查的候选者，每次调用都会进行动态切点检查。

如果将 GreetingDynamicPointcut 类的 getClassFilter() 和 matches(Method method,

Class clazz)方法注释掉，重新运行代码清单 7-33 所示的测试代码，则将得到以下输出信息：

```
调用matches(method,clazz)对com.smart.advisor.Waiter.serveTo做动态检查。
waiter serving Peter...
调用matches(method,clazz)对com.smart.advisor.Waiter.greetTo做动态检查。
waiter greet to Peter...
调用matches(method,clazz)对com.smart.advisor.Waiter.serveTo做动态检查。
waiter serving Peter...
调用matches(method,clazz)对com.smart.advisor.Waiter.greetTo做动态检查。
com.smart.advisor.Waiter.greetTo
How are you! Mr. John.
waiter greet to John...
```

可以发现，每次调用代理对象的任何一个方法，都会执行动态切点检查，这将导致很大的性能问题。所以，在定义动态切点时，切勿忘记同时覆盖 getClassFilter() 和 matches(Method method, Class clazz) 方法，通过静态切点检查排除大部分方法。



提示

7.2 节介绍了动态代理的概念，在这里又碰到了动态切面的概念，这两个概念很容易混淆。其实动态代理的“动态”是相对于那些编译期生成代理和类加载期生成代理而言的。动态代理是运行时动态产生的代理。在 Spring 中，不管是静态切面还是动态切面，都是通过动态代理技术实现的。所谓静态切面，是指在生成代理对象时就确定了增强是否需要织入目标类的连接点上；而动态切面是指必须在运行期根据方法入参的值来判断增强是否需要织入目标类的连接点上。

7.4.6 流程切面

Spring 的流程切面由 DefaultPointcutAdvisor 和 ControlFlowPointcut 实现。流程切点代表由某个方法直接或间接发起调用的其他方法。来看下面的实例，假设通过一个 WaiterDelegate 类代理 Waiter 所有的方法，如代码清单 7-34 所示。

代码清单 7-34 WaiterDelegate

```
package com.smart.advisor;
public class WaiterDelegate {
    private Waiter waiter;
    public void service(String clientName) { //① ←— waiter 的方法通过
        waiter.greetTo(clientName);           —该方法发起调用
        waiter.serveTo(clientName);
    }
    public void setWaiter(Waiter waiter) {
        this.waiter = waiter;
    }
}
```

如果希望所有由 WaiterDelegate#service() 方法发起调用的其他方法都织入 GreetingBeforeAdvice 增强，就必须使用流程切面来完成目标。下面使用

DefaultPointcutAdvisor 配置一个流程切面来完成这一需求，如代码清单 7-35 所示。

代码清单 7-35 配置控制流程切面

```
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
<bean id="waiterTarget" class="com.smart.advisor.Waiter"/>
<bean id="controlFlowPointcut"
    class="org.springframework.aop.support.ControlFlowPointcut">
    <constructor-arg type="java.lang.Class"
        value="com.smart.advisor.WaiterDelegate" />① ← 指定流程切点的类
    <constructor-arg type="java.lang.String"
        value="service" />② ← 指定流程切点的方法
</bean>
<bean id="controlFlowAdvisor"
    class="org.springframework.aop.support.DefaultPointcutAdvisor"
    p:pointcut-ref="controlFlowPointcut"
    p:advice-ref="greetingAdvice" />

<bean id="waiter3" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:interceptorNames="controlFlowAdvisor"
    p:target-ref="waiterTarget"
    p:proxyTargetClass="true"/>
```

ControlFlowPointcut 有两个构造函数，分别是 ControlFlowPointcut(Class clazz) 和 ControlFlowPointcut(Class clazz, String methodName)。第一个构造函数指定一个类作为流程切点；而第二个构造函数指定一个类和某一个方法作为流程切点。

在这里，指定 com.smart.advisor.WaiterDelegate#service() 方法作为切点，表示所有通过该方法直接或间接发起的调用匹配切点。说到这里，可能还有一些模糊的地方，下面通过测试代码观察流程切面的运行效果，如代码清单 7-36 所示。

代码清单 7-36 流程切面测试

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter3");
WaiterDelegate wd = new WaiterDelegate();
wd.setWaiter(waiter);
waiter.serveTo("Peter");
waiter.greetTo("Peter");
wd.service("Peter");
```

运行上面的代码，在控制台上输出以下信息：

```
waiter serving Peter... ① ← 对应 waiter.serveTo("Peter")
waiter greet to Peter... ② ← 对应 waiter.greetTo("Peter")
③ ← 对应 wd.service("Peter")
com.smart.advisor.Waiter.greetTo
How are you! Mr.Peter.
waiter greet to Peter...
com.smart.advisor.Waiter.serveTo
How are you! Mr.Peter.
waiter serving Peter...
```

①处和②处的信息是直接通过 waiter 调用 serveTo() 和 greetTo() 方法的输出，此时增强没有起作用；③处是通过 WaiterDelegate#service() 调用 Waiter 的 serveTo() 和 greetTo() 方法的输出，这时发现 Waiter 的两个方法都织入了增强。

流程切面和动态切面从某种程度上说可以算是一类切面，因为二者都需要在运行期判断动态的环境。对于流程切面来说，代理对象在每次调用目标类方法时，都需要判断方法调用堆栈中是否有满足流程切点要求的方法。因此，和动态切面一样，流程切面对性能的影响也很大。在 JVM 1.4 上，流程切点通常比别的切点要慢 5 倍，在 JVM 1.3 上要慢 10 倍。

7.4.7 复合切点切面

在前面的例子中定义的切面仅有一个切点，有时，一个切点可能难以描述目标连接点的信息。比如在前面流程切面的例子中，假设我们希望由 WaiterDelegate# service() 发起调用且被调用的方法是 Waiter#greetTo() 时才织入增强，这个切点就是复合切点，因为它由两个单独的切点共同确定，第一个切点是代码清单 7-35 所定义的流程切点，而另一个切点是方法名为“greetTo”的方法名切点。

当然，用户也可以只通过一个切点描述同时满足以上两个匹配条件的连接点，而更好的方法是使用 Spring 提供的 ComposablePointcut 把两个切点组合起来，通过切点的复合运算表示。ComposablePointcut 可以将多个切点以并集或交集的方式组合起来，提供了切点之间复合运算的功能。

ComposablePointcut 本身也是一个切点，它实现了 Pointcut 接口。下面先来了解一下 ComposablePointcut 的构造函数。

- ❑ ComposablePointcut(): 构造一个匹配所有类所有方法的复合切点。
- ❑ ComposablePointcut(ClassFilter classFilter): 构造一个匹配特定类所有方法的复合切点。
- ❑ ComposablePointcut(MethodMatcher methodMatcher): 构造一个匹配所有类特定方法的复合切点。
- ❑ ComposablePointcut(ClassFilter classFilter, MethodMatcher methodMatcher): 构造一个匹配特定类特定方法的复合切点。

ComposablePointcut 提供了 3 个交集运算的方法。

- ❑ ComposablePointcut intersection(ClassFilter filter): 将复合切点和一个 ClassFilter 对象进行交集运算，得到一个结果复合切点。
- ❑ ComposablePointcut intersection(MethodMatcher mm): 将复合切点和一个 MethodMatcher 对象进行交集运算，得到一个结果复合切点。
- ❑ ComposablePointcut intersection(Pointcut other): 将复合切点和一个切点对象进行交集运算，得到一个结果复合切点。

ComposablePointcut 提供了两个并集运算的方法。

- ❑ ComposablePointcut union(ClassFilter filter): 将复合切点和一个 ClassFilter 对象进行并集运算，得到一个结果复合切点。

- ComposablePointcut union(MethodMatcher mm): 将复合切点和一个 MethodMatcher 对象进行交并集运算，则得到一个结果复合切点。

ComposablePointcut 没有提供直接对两个切点进行交并集运算的方法，如果需要对两个切点进行交并集运算，可以使用 Spring 提供的 org.springframework.aop.support.Pointcuts 工具类，该工具类中有两个非常好用的静态方法。

- Pointcut intersection(Pointcut a, Pointcut b): 对两个切点进行交集运算，返回一个结果切点，该切点即 ComposablePointcut 对象的实例。
- Pointcut union(Pointcut a, Pointcut b): 对两个切点进行并集运算，返回一个结果切点，该切点即 ComposablePointcut 对象的实例。

下面通过 ComposablePointcut 创建一个流程切点和方法名切点的相交切点，程序代码如代码清单 7-37 所示。

代码清单 7-37 GreetingComposablePointcut

```
package com.smart.advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.support.ComposablePointcut;
import org.springframework.aop.support.ControlFlowPointcut;
import org.springframework.aop.support.NameMatchMethodPointcut;
public class GreetingComposablePointcut {
    public Pointcut getIntersectionPointcut(){           创建一个复合切点
        ComposablePointcut cp = new ComposablePointcut(); //①←          创建一个流程切点
        Pointcut pt1 = new ControlFlowPointcut(WaiterDelegate.class, "service"); //②←          创建一个方法名切点
        NameMatchMethodPointcut pt2 = new NameMatchMethodPointcut(); //③←
        pt2.addMethodName("greetTo");
        return cp.intersection(pt1).intersection(pt2); //④←          将两个切点进行交集操作
    }
}
```

通过 GreetingComposablePointcut#getIntersectionPointcut() 方法，即可得到一个相交的复合切点。配置复合切点的切面和配置其他切点一样，如代码清单 7-38 所示。

代码清单 7-38 配置复合切点切面

```
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
<bean id="gcp" class="com.smart.advisor.GreetingComposablePointcut" />
<bean id="composableAdvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor"
    p:pointcut="#{gcp.intersectionPointcut}" ①          引用 gcp.getIntersectionPointcut()
    p:advice-ref="greetingAdvice"/>                    方法所返回的复合切点
<bean id="waiter4" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:interceptorNames="composableAdvisor" ②          使用复合切点切面
    p:target-ref="waiterTarget"
    p:proxyTargetClass="true"/>
```

在①处使用 util 命名空间的标签引用另一个 Bean 的属性(关于 util 命名空间标签的详细信息,请参看 5.4.6 节)。

下面编写相应的测试代码:

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter4");
WaiterDelegate wd = new WaiterDelegate();
wd.setWaiter(waiter);
waiter.serveTo("Peter");
waiter.greetTo("Peter");
wd.service("Peter");
```

运行以上代码,输出以下信息:

```
waiter serving Peter... ① ← 对应 waiter.serveTo("Peter")
waiter greet to Peter... ② ← 对应 waiter.greetTo("Peter")
com.smart.advisor.Waiter.greetTo ③ ← 对应通过 wd.service("Peter")
How are you! Mr.Peter.          调用的 Waiter.greetTo() 方法
waiter greet to Peter...       对应通过 wd.service("Peter")
waiter serving Peter... ④ ← 调用的 Waiter.serveTo() 方法
```

通过以上输出信息发现,只有通过 WaiterDelegate#service()方法调用 Waiter#greetTo()方法时才织入了增强,而这正是复合交集切点所描述的接连点。

7.4.8 引介切面

7.3.6 节介绍了引介增强的知识,引介切面是引介增强的封装器,通过引介切面,可以更容易地为现有对象添加任何接口的实现。图 7-9 是引介切面的类继承关系图。

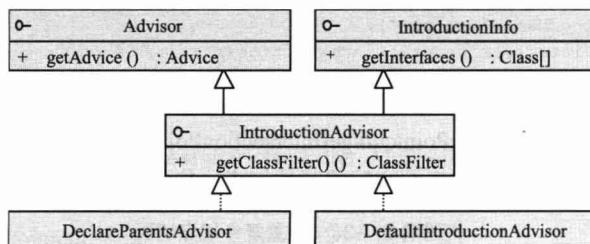


图 7-9 引介切面类继承关系图

从图 7-9 中可以看出,IntroductionAdvisor 接口同时继承 Advisor 和 IntroductionInfo 接口,IntroductionInfo 接口描述了目标类需要实现的新接口。IntroductionAdvisor 和 PointcutAdvisor 接口不同,它仅有一个类过滤器 ClassFilter 而没有 MethodMatcher,这是因为引介切面的切点是类级别的,而 Pointcut 的切点是方法级别的。

IntroductionAdvisor 有两个实现类,分别是 DefaultIntroductionAdvisor 和 DeclareParentsAdvisor,前者是引介切面最常用的实现类,后者用于实现使用 AspectJ 语言的 DeclareParent 注解表示的引介切面。

DefaultIntroductionAdvisor 拥有 3 个构造函数。

- DefaultIntroductionAdvisor(Advice advice): 通过一个增强创建的引介切面，引介切面将为目标对象新增增强对象中所有接口的实现。
- DefaultIntroductionAdvisor(DynamicIntroductionAdvice advice, Class clazz): 通过一个增强和一个指定的接口类创建引介切面，仅为目标对象新增 clazz 接口的实现。
- DefaultIntroductionAdvisor(Advice advice, IntroductionInfo introductionInfo): 通过一个增强和一个 IntroductionInfo 创建引介切面，目标对象需要实现哪些接口由 introductionInfo 对象的 getInterfaces()方法表示。

下面通过 DefaultIntroductionAdvisor 为代码清单 7-22 中的引介增强配置切面，会发现这种方式比前面的方式要更简洁、更清晰，如代码清单 7-39 所示。

代码清单 7-39 配置引介切面

```
<bean id="introduceAdvisor" class="org.springframework.aop.support.  
DefaultIntroductionAdvisor">  
    <constructor-arg>①← ControllablePerformanceMonitor 是一个 Advice 对象  
        <bean class="com.smart.introduce.ControllablePerformanceMonitor"/>  
    </constructor-arg>  
</bean>  
<bean id="forumServiceTarget" class="com.smart.introduce.ForumService"/>  
<bean id="forumService" class="org.springframework.aop.framework.ProxyFactoryBean"  
    p:interceptorNames="introduceAdvisor"  
    p:target-ref="forumServiceTarget"  
    p:proxyTargetClass="true"/>
```

虽然引介切面和其他切面有很大的不同，但却可以采用相似的 Spring 配置方式配置引介切面。通过这种方式配置的代码在本质上和代码清单 7-23 中配置的效果是一样的。

7.5 自动创建代理

在前面所有的例子中，都通过 ProxyFactoryBean 创建织入切面的代理，每个需要被代理的 Bean 都需要使用一个 ProxyFactoryBean 进行配置，虽然可以使用父子<bean>进行改造，但还是很麻烦。对于小型系统，可以将就使用，但对于由拥有众多需要代理 Bean 的系统，原来的配置显然不尽如人意。

幸运的是，Spring 提供了自动代理机制，让容器自动生成代理，把开发人员从繁琐的配置工作中解放出来。在内部，Spring 使用 BeanPostProcessor 自动完成这项工作。

7.5.1 实现类介绍

这些基于 BeanPostProcessor 的自动代理创建器的实现类，将根据一些规则自动在容

器实例化 Bean 时为匹配的 Bean 生成代理实例。这些代理创建器可以分为 3 类。

- 基于 Bean 配置名规则的自动代理创建器：允许为一组特定配置名的 Bean 自动创建代理实例的代理创建器，实现类为 BeanNameAutoProxyCreator。
- 基于 Advisor 匹配机制的自动代理创建器：它会对容器中所有的 Advisor 进行扫描，自动将这些切面应用到匹配的 Bean 中（为目标 Bean 创建代理实例），实现类为 DefaultAdvisorAutoProxyCreator。
- 基于 Bean 中 AspectJ 注解标签的自动代理创建器：为包含 AspectJ 注解的 Bean 自动创建代理实例，实现类为 AnnotationAwareAspectJAutoProxyCreator。

图 7-10 是自动代理创建器实现类的类继承图。

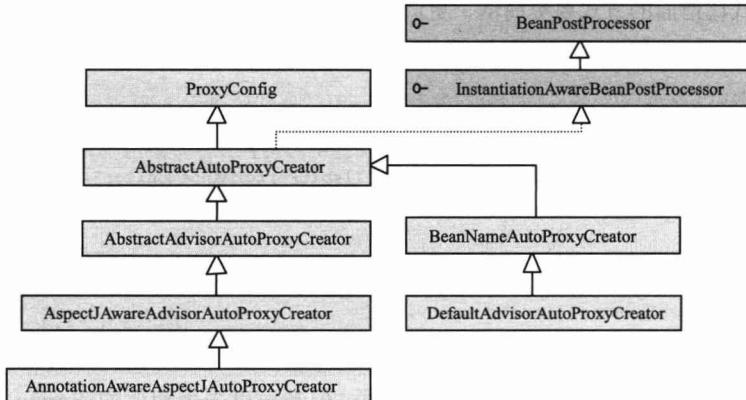


图 7-10 自动代理创建器实现类的类继承图

从图 7-10 中可以清楚地看到所有的自动代理创建器类都实现了 BeanPostProcessor，在容器实例化 Bean 时，BeanPostProcessor 将对它进行加工处理，所以，自动代理创建器有机会对满足匹配规则的 Bean 自动创建代理对象。

本章暂不涉及 AspectJ 的内容，下面只讲解 BeanNameAutoProxyCreator 和 DefaultAdvisorAutoProxyCreator 的用法。

7.5.2 BeanNameAutoProxyCreator

在代码清单 7-29 中通过配置两个 ProxyFactoryBean 分别为 waiter 和 seller 的 Bean 创建代理对象。下面通过 BeanNameAutoProxyCreator 以更优雅、更便捷的方式完成相同的功能，如代码清单 7-40 所示。

代码清单 7-40 使用 Bean 名进行自动代理

```

<bean id="waiter" class="com.smart.advisor.Waiter" />
<bean id="seller" class="com.smart.advisor.Seller" />
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator"
  
```

```
p:beanNames="*er" ① ← 由于只有一个 Bean 名称，所以直接使用
p:interceptorNames="greetingAdvice" value 属性进行配置，可以通过<list>子元
p:optimize="true"/> 素设定多个 Bean 名称，或通过逗号、空格、
分号的方式设定多个 Bean 名称
```

BeanNameAutoProxyCreator 有一个 beanNames 属性，它允许用户指定一组需要自动代理的 Bean 名称，Bean 名称可以使用*通配符。假设 Spring 容器中除 waiter 和 seller 外还有其他的 Bean，如果以“er”为后缀的规则可以将这两个 Bean 和容器中其他的 Bean 区分开，就可以通过将 beanNames 属性设定为 “*er” 使 waiter 和 seller 这两个 Bean 被自动代理。当然，使用通配符会带来一定的风险，在例子中，假设一个其他的 Bean 名称也以 “er” 结尾，则自动代理创建器也会为该 Bean 创建代理。所以为了保险起见，用户可以使用下面的方式配置 beanNames 属性：value="waiter,seller"。

一般情况下不会为 FactoryBean 的 Bean 创建代理，如果刚好有这样一个需求，则需要在 beanNames 中指定添加\$的 Bean 名称，如<property name="beanNames" value="\$waiter" />等。

BeanNameAutoProxyCreator 的 interceptorNames 属性指定一个或多个增强 Bean 的名称。此外，还有一个常用的 optimize 属性，如果将此属性设置为 true，则将强制使用 CGLib 动态代理技术。

通过这样的配置后，容器在创建 waiter 和 seller Bean 的实例时，就会自动为它们创建代理对象。而这一操作对于使用者来说是完全透明的，可以通过以下测试证实这一点：

```
String configPath = "com/smart/autoproxy/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter"); ①
Seller seller = (Seller) ctx.getBean("seller"); ②
waiter.greetTo("John");
seller.greetTo("Tom");
```

运行以上代码，在控制台上输出以下信息：

```
com.smart.advisor.Waiter.greetTo ① ← 被自动织入的增强逻辑
How are you! Mr. John.
waiter greet to John...
com.smart.advisor.Seller.greetTo ② ← 被自动织入的增强逻辑
How are you! Mr. Tom.
seller greet to Tom...
```

通过输出信息可以知道，从容器中返回的 waiter Bean 的 greetTo()方法及 seller Bean 的 greetTo()方法都被织入了增强，可见①处返回的 waiter Bean 和②处返回的 seller Bean 都是被代理过的对象。

7.5.3 DefaultAdvisorAutoProxyCreator

我们知道切面 Advisor 是切点和增强的复合体，Advisor 本身已经包含了足够的信息，如横切逻辑（要织入什么）及连接点（织入哪里）。

DefaultAdvisorAutoProxyCreator 能够扫描容器中的 Advisor，并将 Advisor 自动织入匹配的目标 Bean 中，即为匹配的目标 Bean 自动创建代理。

在代码清单 7-30 中通过 `ProxyFactoryBean` 为 `waiter` 配置了代理，在这里，引入 `DefaultAdvisorAutoProxyCreator` 为容器中所有带“greet”方法名的目标 Bean 自动创建代理。

```
<bean id="waiter" class="com.smart.advisor.Waiter" />
<bean id="seller" class="com.smart.advisor.Seller" />
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
<bean id="regexpAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
      p:patterns=". *greet.*"
      p:advice-ref="greetingAdvice" />

<bean class="org.springframework.aop.framework.autoproxy.
    DefaultAdvisorAutoProxyCreator"/>①
```

在①处，用 `DefaultAdvisorAutoProxyCreator` 定义了一个 Bean，它负责将容器中的 Advisor 织入匹配的目标 Bean 中。通过下面的测试代码检测一下自动代理创建器是否正常工作：

```
String configPath = "com/smart/autoproxy/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter");
Seller seller = (Seller) ctx.getBean("seller");
waiter.serveTo("John");
waiter.greetTo("John");
seller.greetTo("Tom");
```

运行以上代码，输出以下信息：

```
waiter serving John...
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...
com.smart.advisor.Seller.greetTo
How are you! Mr.Tom.
seller greet to Tom...
```

`Waiter#serveTo()`方法没有被织入增强，而 `Waiter` 和 `Seller` 的 `greetTo()`方法都被织入了增强，由此可见，增强被正确地织入匹配的连接点中。

7.5.4 AOP 无法增强疑难问题剖析

大家在使用 Spring AOP 时，或多或少会碰到一些方法无法被增强的问题，有时同一个类里面的方法有的可以被增强，有的却无法被增强。要分析其原因，首先要从 Spring AOP 的实现机制入手。从上文 Spring AOP 基础知识的学习可以知道，AOP 底层实现有两种方法：一种是基于 JDK 动态代理；另一种是基于 CGLib 动态代理。

在 JDK 动态代理中通过接口来实现方法拦截，所以必须确保要拦截的目标方法在接口中有定义，否则将无法实现拦截。

在 CGLib 动态代理中通过动态生成代理子类来实现方法拦截，所以必须确保要拦截的目标方法可被子类访问，也就是目标方法必须定义为非 final，则非私有实例方法。

是否注意了上面几点，就可以万事大吉了？要回答这个问题，先来看一个实例。首先对7.5.3节的示例进行简单的修改，如代码清单7-41所示。

代码清单7-41 增强问题测试

```
public class AopAwareTest {
    @Test
    public void autoProxy() {
        String configPath = "com/smart/autoproxy/beans-aware.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter) ctx.getBean("waiter");
        waiter.serveTo("John");
        waiter.greetTo("John"); //①
    }
}
```

在这里，引入DefaultAdvisorAutoProxyCreator为容器中所有带“To”方法名的目标Bean自动创建代理，如代码清单7-42所示。

代码清单7-42 beans-aware.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
...
http://www.springframework.org/schema/context/spring-context.xsd">
    <context:component-scan base-package="com.smart.aop" />
    <bean id="waiter" class="com.smart.advisor.Waiter" />
    <bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
    <!--通过Advisor自动创建代理-->
    <bean id="regexpAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
        p:patterns=".*To.*" p:advice-ref="greetingAdvice" />
    <bean
        class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAuto
        ProxyCreator"
        p:proxyTargetClass="true" />
</beans>
```

为了方便测试，这里只保留服务生。运行以上代码，输出以下信息：

```
com.smart.advisor.Waiter.serveTo
How are you! Mr.John.
waiter serving John...
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...
```

从运行结果来看，Waiter#serveTo()和Waiter#greetTo()方法都被织入了增强，由此可见增强被正确地织入匹配的连接点中。现在改造一下Waiter#serveTo()方法，让Waiter#serveTo()方法内部直接调用Waiter#greetTo()方法，并把代码清单7-41中①处的方法注释掉。模拟测试在一个类中一个可织入的方法serveTo()调用另一个可织入的方法greetTo()时，这两个方法是否都可以被织入增强，如代码清单7-43所示。

代码清单 7-43 增强问题测试

```

public class AopAwareTest {
    @Test
    public void autoProxy() {
        String configPath = "com/smart/autoproxy/beans-aware.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter) ctx.getBean("waiter");
        waiter.serveTo("John");
    }
}

public class Waiter {
    ...
    public void serveTo(String name) {
        System.out.println("waiter serving "+name+"...");
        greetTo(name);
    }

    public void greetTo(String name) {
        System.out.println("waiter greet to "+name+"...");
    }
}

```

运行以上代码，输出以下信息：

```

com.smart.advisor.Waiter.serveTo
How are you! Mr.John.
waiter serving John...
waiter greet to John...

```

从运行结果来看，只有 Waiter#serveTo()方法被织入了增强，而 Waiter#greetTo()方法没有被织入增强。原本希望这两个方法都被织入增强，但现在只有从外部调用的方法 serveTo()被增强，在同一个类的内部方法之间调用的方法 greetTo()无法被增强。笔者在实际的项目中也碰到过这个问题，经过反复的测试研究发现，原来在方法内部之间调用的时候，不会使用被增强的代理类，而是直接调用未被增强原类的方法，这也就是方法 greetTo()无法被织入增强的原因。

在实际项目中存在内部两个方法调用的问题，同时又希望它们都能够被增强，例如在一个 Service 类的两个方法 bus1()和 bus2()中都使用了缓存注解@Cacheable，也就是希望这两个方法都启用缓存。如果在业务处理过程中，bus2()方法只被 bus1()所调用，则 bus2()方法永远无法启用缓存，这在无形中给系统留下了隐患，而且不易被发现。

笔者为此研究了一套解决办法，就是想办法在内部方法调用时，让其通过代理类调用内部的方法。因此，需要让原来的 Waiter 实现一个可注入自身代理类的接口 BeanSelfProxyAware，如代码清单 7-44 所示。

代码清单 7-44 BeanSelfProxyAware 接口

```

public interface BeanSelfProxyAware {
    void setSelfProxy(Object object); //织入自身代理类接口
}

```

有了这个代理类接口之后，需要对所有实现了 BeanSelfProxyAware 接口的 Bean 执

行自身代理 Bean 的注入，设计一个可复用的注入装配器 BeanSelfProxyAwareMounter，如代码清单 7-45 所示。

代码清单 7-45 BeanSelfProxyAwareMounter

```
package com.smart.aop;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;
...
@Component
public class BeanSelfProxyAwareMounter implements SystemBootAddon,
ApplicationContextAware {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private ApplicationContext applicationContext;

    //①注入Spring容器
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }

    public void onReady() { //②实现系统启动器接口中的装配就绪方法

        //②-1从容器中获取所有注入的自动代理Bean
        Map<String, BeanSelfProxyAware> proxyAwareMap =
            applicationContext.getBeansOfType(BeanSelfProxyAware.class);
        if(proxyAwareMap!=null){
            for (BeanSelfProxyAware beanSelfProxyAware : proxyAwareMap.values()) {
                beanSelfProxyAware.setSelfProxy(beanSelfProxyAware);
                if (logger.isDebugEnabled()) {
                    logger.debug("{}注册自身被代理的实例。");
                }
            }
        }
    }

    public int getOrder() {
        return Ordered.HIGHEST_PRECEDENCE;
    }
}
```

在①处通过实现 ApplicationContextAware#setApplicationContext()接口方法来注入 Spring 容器上下文。在②处实现了系统启动器 SystemBootAddon#onReady()接口方法，此接口方法在下文中进行详细说明。这里的实现逻辑是从 Spring 容器中获取所有实现自动代理织入接口 BeanSelfProxyAware 的 Bean，循环迭代遍历这些 Bean，并调用 setSelfProxy()方法将自身代理类注入自身。

接下来需要编写一个系统所有组件都装载完成后、准备就绪前调用的插件接口 SystemBootAddon，用于在 Spring 容器启动完成之后触发调用注入装配器 BeanSelfProxy AwareMounter，如代码清单 7-46 所示。

代码清单 7-46 SystemBootAddon接口

```
public interface SystemBootAddon extends Ordered{
    //在系统就绪后调用的方法
    void onReady();
}
```

这个接口比较简单，只设计了一个接口方法 `onReady()`，用于在系统就绪后调用。同时实现了 Bean 加载顺序接口 `Ordered`，在插件中可以实现 `Ordered#getOrder` 方法，返回一个整型数字来指定插件的执行顺序。值越小优先被加载处理，值越大最后被加载处理。

最后需要设置一个启动管理器，告诉 Spring 什么时候触发 `BeanSelfProxyAwareMounter` 装配器，如代码清单 7-47 所示。

代码清单 7-47 设置启动管理器

```
...
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextRefreshedEvent;
import org.springframework.core.OrderComparator;
import org.springframework.stereotype.Component;
import org.springframework.util.Assert;

@Component
public class SystemBootManager implements ApplicationListener<ContextRefreshedEvent> {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private List<SystemBootAddon> systemBootAddons = Collections.EMPTY_LIST;
    private boolean hasRunOnce = false;

    //①注入所有SystemBootAddon插件
    @Autowired(required = false)
    public void setSystemBootAddons(List<SystemBootAddon> systemBootAddons) {
        Assert.notEmpty(systemBootAddons);
        OrderComparator.sort(systemBootAddons);
        this.systemBootAddons = systemBootAddons;
    }

    //②触发所有插件
    public void onApplicationEvent(ContextRefreshedEvent event) {
        if (!hasRunOnce) {
            for (SystemBootAddon systemBootAddon : systemBootAddons) {
                systemBootAddon.onReady();
                if (logger.isDebugEnabled()) {
                    logger.debug("执行插件:" + systemBootAddon.getClass().getCanonicalName());
                }
            }
            hasRunOnce = true;
        } else {
            if (logger.isDebugEnabled()) {
                logger.debug("已执行过容器启动插件集，本次忽略之。");
            }
        }
    }
}
```

}

在①处通过自动注入方式注入所有实现 SystemBootAddon 接口的插件，在②处通过监听 Spring 容器的 ContextRefreshedEvent 事件调用容器中所有注册的 SystemBootAddon 插件。至此，设计的注入 Bean 自身代理类的通用型插件已经全部完成，接下来只需在需要注入自身代理类的 Bean 中实现 BeanSelfProxyAware 接口即可，如本示例中的 Waiter 类。下面对代码清单 7-43 中的 Waiter 类进行简单改造，如代码清单 7-48 所示。

代码清单 7-48 增强问题测试

```
public class Waiter implements BeanSelfProxyAware {  
    ...  
    private Waiter waiter;  
  
    public void setSelfProxy(Object object) {  
        waiter = (Waiter)object;  
    }  
  
    public void serveTo(String name){  
        System.out.println("waiter serving "+name+"...");  
        greetTo(name);  
    }  
  
    public void greetTo(String name) {  
        System.out.println("waiter greet to "+name+"...")  
    }  
}
```

重新运行代码清单 7-43 中的 AopAwareTest 测试，输出以下信息：

```
com.smart.advisor.Waiter.serveTo  
How are you! Mr.John.  
waiter serving John...  
com.smart.advisor.Waiter.greetTo  
How are you! Mr.John.  
waiter greet to John...
```

从运行结果来看，Waiter#serveTo()和 Waiter#greetTo()方法都被织入了增强，由此可见设计的注入 Bean 自身代理类的通用型插件有效。如果读者在实际项目中遇到同样的问题，则可以借鉴或使用笔者设计的这个插件。

如果不想使用上述方式，则也可以考虑使用 AspectJ，因为其支持编译期织入且不需要生成代理类，也就避免了因生成代理类产生的一些制约（static 和 final 方法不能被覆盖等问题）。关于 AspectJ 的用法，将在下一章进行详细讲解。

7.6 小结

AOP 是 OOP 的延伸，它为程序开发提供了一个崭新的思考角度，可以将重复性的

横切逻辑抽取到统一的模块中。通过 OOP 的纵向抽象和 AOP 的横向抽取，程序才可以真正解决重复性代码问题。

Spring 采用 JDK 动态代理和 CGLib 动态代理技术在运行期织入增强，所以不需要装备特殊的编译器或类装载器就可以使用 AOP 的功能。要使用 JDK 动态代理，目标类必须实现接口，而 CGLib 不对目标类作任何限制，它通过动态生成目标类子类的方式提供代理。JDK 在创建代理对象时的性能高于 CGLib，而生成的代理对象的运行性能却比 CGLib 的低。如果是 singleton 的代理，则推荐使用 CGLib 动态代理。

Spring 只能在方法级别上织入增强，Spring 提供了 4 种类型的方法增强，分别是前置增强、后置增强、环绕增强和异常抛出增强，此外还有一种特殊的引介增强。引介增强是类级别的，它为目标类织入新的接口实现。从广义上说，增强其实就是一种最简单的切面，它既包括横切代码又包括切点信息，只不过它的切点只是简单的方法相对位置信息。所以增强一般需要和切点联合才可以表示一个更具实用性的切面。

在 Spring 中，普通的切点通过目标类名和方法名描述连接点的信息。流程切点是比较特殊的切点，它通过方法调用堆栈的运行环境信息来决定连接点。有时需要通过切点的交叉或合并描述一个最终的切点，这时可以使用 `ComposablePointcut` 的复合切点。

切面是增强和切点的联合体，可以很方便地通过 Spring 提供的 `ProxyBeanFactory` 将切面织入不同的目标类中。当然，为每个目标类手工配置一个切面是比较烦琐的，Spring 利用 `BeanPostProcessor` 可干涉 Bean 生命周期的机制，提供了一些可以自动创建代理、织入切面的自动代理创建器，其中 `DefaultAdvisorAutoProxyCreator` 是功能强大的自动代理创建器，它可以将容器中的所有 Advisor 自动织入目标 Bean 中。

第 8 章

基于@AspectJ 和 Schema 的 AOP

在上一章的学习中我们发现，在 Spring 中定义一个切面是比较烦琐的，需要实现专门的接口，并进行一些较为复杂的配置，Spring AOP 的配置是被批评最多的地方。Spring 听到了这方面的声音，下决心解决这一问题，并取得了很好的突破。如今，Spring AOP 已经焕然一新，用户可以使用 @AspectJ 注解非常容易地定义一个切面，而不需要实现任何接口。对于没有使用 Java 5.0 的项目，可以通过基于 Schema 的配置定义切面，其方便程度和基于 @AspectJ 注解的配置相差无几。此外，Spring 还可以使用 AspectJ 语言编写切面，发扬二者的长处：用 AspectJ 编织切面，并让 Spring 容器管理这些切面。

本章主要内容：

- ◆ Java 5.0 注解知识
- ◆ 通过 @AspectJ 定义切面
- ◆ 切点函数讲解
- ◆ 绑定连接点参数
- ◆ 基于 Schema 配置定义切面
- ◆ Spring LTW

本章亮点：

- ◆ 对切点表达式函数进行深入分析
- ◆ 深入讲解集成 AspectJ 的过程

8.1 Spring 对 AOP 的支持

Spring 在新版本中对 AOP 功能进行了重要的增强，主要表现在以下几个方面：

- 新增了基于 Schema 的配置支持，为 AOP 专门提供了 aop 命名空间。
- 新增了对 AspectJ 切点表达式语言的支持。@AspectJ 是 AspectJ 1.5 新增的功能，它通过 Java 5.0 的注解技术，允许开发者在 POJO 中定义切面。Spring 使用和 @AspectJ 相同风格的注解，并通过 AspectJ 提供的注解库和解析库处理切点。当然，由于 Spring 只支持方法级的切点，所以仅对 @AspectJ 提供了有限的支持。
- 可以无缝地集成 AspectJ。AspectJ 提供了语言级切面的实现，Spring 无意开发一个重复的东西，Spring 对开源世界里一切优秀的东西向来采取兼收并蓄的态度。

这里所说的 Spring AOP 包括基于 XML 配置的 AOP 和基于 @AspectJ 注解的 AOP，这两种方法虽然在配置切面时的表现方式不同，但底层都采用了动态代理技术（JDK 或 CGLib 动态代理）。Spring 可以集成 AspectJ，但 AspectJ 本身并不属于 Spring AOP 的范畴。

一般情况下，对于开发 JAVA EE 企业应用的开发者而言，Spring AOP 已经可以满足使用的要求。虽然 AspectJ 提供对 AOP 更为细致的实现，但像实例化切面、属性访问切面、条件切面等功能，在实际应用中并不常用。

如果是基于 Java 5.0 的项目，推荐使用 Spring 提供的 @AspectJ 配置方式，因为它能以更简单、更直接的方式应用切面。

8.2 Java 5.0 注解知识快速进阶

在进入本章学习之前，有必要对 Java 5.0 新增的注解（Annotation）技术进行简单的学习。因为 Spring 支持 @AspectJ，而 @AspectJ 本身就是基于 Java 5.0 的注解技术，所以学习 Java 5.0 的注解知识有助于更好地理解和掌握 Spring 的 AOP 技术。

8.2.1 了解注解

对于 Java 开发人员来说，在编写代码时，除源程序外，还会使用 Javadoc 标签对类、方法或成员变量进行注释，以便使用 Javadoc 工具生成和源码配套的 Javadoc 文档。这些 @param、@return 等 Javadoc 标签就是注解标签，它们为第三方工具提供了描述程序代码的注释信息。使用过 Xdoclet 的读者对此更有感触，像 Struts、Hibernate 都提供了 Xdoclet 标签，使用它们可以快速地生成对应程序代码的配置文件。

Java 5.0 注解可以看作 Javadoc 和 Xdoclet 标签的延伸与发展。在 Java 5.0 中可以自定义这些标签，并通过 Java 语言的反射机制获取类中标注的注解，完成特定的功能。

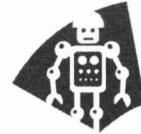
注解是代码的附属信息，它遵循一个基本原则：注解不能直接干扰程序代码的运行，无论增加或删除注解，代码都能够正常运行。Java 语言解释器会忽略这些注解，而由第三方工具负责对注解进行处理。第三方工具可以利用代码中的注解间接控制程序代码的

运行，它们通过 Java 反射机制读取注解的信息，并根据这些信息更改目标程序的逻辑，而这正是 Spring AOP 对 @AspectJ 提供支持所采取的方法。



轻松一刻

很多事物的设计都必须遵循最基本的原则。为了防止机器人伤害人类，科幻作家阿西莫夫于 1940 年提出了“机器人三原则”：第一，机器人不能伤害人类；第二，机器人应遵守人类的命令，与第一条违背的命令除外；第三，机器人应能保护自己，与第一条违背的命令除外。这是给机器人赋予的伦理性纲领，机器人学术界一直将这 3 条原则作为机器人开发的准则。



8.2.2 一个简单的注解类

通常情况下，第三方工具不但负责处理特定的注解，其本身还提供了这些注解的定义，所以通常仅需关注如何使用注解即可。但定义注解类本身并不困难，Java 提供了定义注解的语法。下面着手编写一个简单的注解类，如代码清单 8-1 所示。

代码清单 8-1 NeedTest注解类

```
package com.smart.aspectj.anno;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME) //①声明注解的保留期限
@Target(ElementType.METHOD) //②声明可以使用该注解的目标类型
public @interface NeedTest { //③定义注解
    boolean value() default true; //④声明注解成员
}
```

Java 新语法规规定使用 @interface 修饰符定义注解类，如③处所示。一个注解可以拥有多个成员，成员声明和接口方法声明类似，这里仅定义了一个成员，如④处所示。成员声明有以下几点限制：

- 成员以无入参、无抛出异常的方式声明，如 boolean value(String str)、boolean value() throws Exception 等方式是非法的。
- 可以通过 default 为成员指定一个默认值，如 String level() default "LOW_LEVEL"、int high() default 2 是合法的，当然也可以不指定默认值。
- 成员类型是受限的，合法的类型包括原始类型及其封装类、String、Class、enums、注解类型，以及上述类型的数组类型，如 ForumService value()、List foo() 是非法的。

在①和②处所看到的注解是 Java 预定义的注解，称为元注解（Meta-Annotation），它们被 Java 编译器使用，会对注解类的行为产生影响。@Retention(RetentionPolicy.

RUNTIME)表示 NeedTest 这个注解可以在运行期被 JVM 读取，注解的保留期限类型在 java.lang.annotation.Retention 类中定义，介绍如下。

- SOURCE：注解信息仅保留在目标类代码的源码文件中，但对应的字节码文件将不再保留。
- CLASS：注解信息将进入目标类代码的字节码文件中，但类加载器加载字节码文件时不会将注解加载到 JVM 中，即运行期不能获取注解信息。
- RUNTIME：注解信息在目标类加载到 JVM 后依然保留，在运行期可以通过反射机制读取类中的注解信息。

Target(ElementType.METHOD)表示 NeedTest 这个注解只能应用到目标类的方法上，注解的应用目标在 java.lang.annotation.ElementType 类中定义，介绍如下。

- TYPE：类、接口、注解类、Enum 声明处，相应的注解称为类型注解。
- FIELD：类成员变量或常量声明处，相应的注解称为域值注解。
- METHOD：方法声明处，相应的注解称为方法注解。
- PARAMETER：参数声明处，相应的注解称为参数注解。
- CONSTRUCTOR：构造函数声明处，相应的注解称为构造函数注解。
- LOCAL_VARIABLE：局部变量声明处，相应的注解称为局部变量注解。
- ANNOTATION_TYPE：注解类声明处，相应的注解称为注解类注解，ElementType.TYPE 包括 ElementType.ANNOTATION_TYPE。
- PACKAGE：包声明处，相应的注解称为包注解。

如果注解只有一个成员，则成员名必须取名为 value()，在使用时可以忽略成员名和赋值号 (=)，如 @NeedTest(true)。当注解类拥有多个成员时，如果仅对 value 成员进行赋值，则也可不使用赋值号；如果同时对多个成员进行赋值，则必须使用赋值号，如 DeclareParents (value = "NaiveWaiter", defaultImpl = SmartSeller.class)。注解类可以没有成员，没有成员的注解称为标识注解，解释程序以标识注解存在与否进行相应的处理；此外，所有的注解类都隐式继承于 java.lang.annotation.Annotation，但注解不允许显式继承于其他的接口。

我们希望使用 NeedTest 注解对业务类的方法进行标注，以便测试工具可以根据注解情况激活或关闭对业务类的测试。在编写好 NeedTest 注解类后，就可以在其他类中使用它了。

8.2.3 使用注解

在 ForumService 中使用 NeedTest 注解，标注业务方法是否需要测试，如代码清单 8-2 所示。

代码清单 8-2 ForumService：使用注解

```

package com.smart.aspectj.anno;
public class ForumService {
    @NeedTest(value=true) //①标注注解
    public void deleteForum(int forumId){
        System.out.println("删除论坛模块: "+forumId);
    }

    @NeedTest(value=false) //②标注注解
    public void deleteTopic(int postId){
        System.out.println("删除论坛主题: "+postId);
    }
}

```

如果注解类和目标类不在同一个包中，则需要通过 import 引用注解类。在①和②处，使用 NeedTest 分别对 deleteForum() 和 deleteTopic() 方法进行标注。在标注注解时，可以通过以下格式对注解成员进行赋值：

```
@<注解名>(<成员名 1>=<成员值 1>,<成员名 2>=<成员值 2>,...)
```

如果成员是数组类型，则可以通过 {} 进行赋值，如 boolean 数组的成员可以设置为 {true, false, true}。下面是几个注解标注的例子。

示例 1：多成员的注解。

```
@AnnoExample(id= 2868724, synopsis = "Enable time-travel",
engineer = "Mr. Peabody",date = "4/1/2007")
```

示例 2：一个成员的注解，成员名为 value。可以省略成员名和赋值符号。

```
@Copyright("2011 bookegou.com All Right Reserved")
```

示例 3：无成员的注解。

```
@Override
```

示例 4：成员为字符串数组的注解。

```
@SuppressWarnings(value={"unchecked","fallthrough"})
```

示例 5：成员为注解数组类型的注解。

```
@Reviews({@Review(grade=Review.Grade.EXCELLENT,reviewer="df"),
@Review(grade=Review.Grade.UNSATISFACTORY,reviewer="eg",
comment="This method needs an @Override annotation")})
```

@Reviews 注解拥有一个@Review 注解数组类型的成员。@Review 注解类型有 3 个成员，其中 reviewer、comment 都是 String 类型的成员，但 comment 有默认值，而 grade 是枚举类型的成员。

由于 NeedTest 注解的保留限期是 RetentionPolicy.RUNTIME 类型，因此，当 ForumService 被加载到 JVM 时，仍可通过反射机制访问到 ForumService 各个方法的注解信息。

8.2.4 访问注解

前面提到过，注解不会直接影响程序的运行，但是第三方程序或工具可以利用代码

中的注解完成特殊的任务，间接控制程序的运行。对于 RetentionPolicy.RUNTIME 保留期限的注解，可以通过反射机制访问类中的注解。

在 Java 5.0 中，Package、Class、Constructor、Method 及 Field 等反射对象都新增了访问注解信息的方法：`<T extends Annotation>T getAnnotation(Class<T> annotationClass)`，该方法支持通过泛型直接返回注解对象。

下面通过反射来访问注解，得出 ForumService 类中通过@NeedTest 注解所承载的测试需求，如代码清单 8-3 所示。

代码清单 8-3 ToolTest：访问代码中的注解

```
package com.smart.aspectj.anno;
import java.lang.reflect.Method;
import org.testng.annotations.*;
public class ToolTest {
    @Test
    public void tool() {
        //①得到ForumService对应的Class对象
        Class clazz = ForumService.class;
        //②得到 ForumService 对应的 Method 数组
        Method[] methods = clazz.getDeclaredMethods();
        System.out.println(methods.length);
        for (Method method : methods) {
            //③获取方法上所标注的注解对象
            NeedTest nt = method.getAnnotation(NeedTest.class);
            if (nt != null) {
                if (nt.value()) {
                    System.out.println(method.getName() + "()需要测试");
                } else {
                    System.out.println(method.getName() + "()不需要测试");
                }
            }
        }
    }
}
```

在③处通过方法的反射对象，获取了方法上所标注的 NeedTest 注解对象，接着就可以访问注解对象的成员，从而得到 ForumService 类方法的测试需求。运行以上代码，输出以下信息：

```
deleteForum()需要测试
deleteTopic()不需要测试
```

8.3 着手使用@AspectJ

第 7 章分别使用 Pointcut 和 Advice 接口描述切点和增强，并用 Advisor 整合二者描

述切面，@AspectJ则采用注解来描述切点、增强，二者只是表述方式不同，描述内容的本质是完全相同的，这就好比一个用中文、一个用英文讲述同一则伊索寓言。

8.3.1 使用前的准备

在使用@AspectJ之前，首先必须保证所使用的Java是5.0及以上版本，否则无法使用注解技术。

Spring在处理@Aspect注解表达式时，需要将Spring的asm模块添加到类路径中。asm是轻量级的字节码处理框架，因为Java的反射机制无法获取入参名，Spring就利用asm处理@AspectJ中所描述的方法入参名。

此外，Spring采用AspectJ提供的@AspectJ注解类库及相应的解析类库，需要在pom.xml文件中添加aspectj.weaver和aspectj.tools类包的依赖。

8.3.2 一个简单的例子

在做好准备工作后，就可以开始编写一个基于@AspectJ的切面。首先来看一个简单的例子，以便对@AspectJ有一个切身的体会。

@AspectJ采用不同的方式对AOP进行描述，依旧使用NaiveWaiter的例子进行讲解。为了方便阅读，再次给出NaiveWaiter类的代码，如下：

```
package com.smart;
public class NaiveWaiter implements Waiter {
    public void greetTo(String clientName) {
        System.out.println("NaiveWaiter:greet to "+clientName+"...");
    }
    public void serveTo(String clientName) {
        System.out.println("NaiveWaiter:serving "+clientName+"...");
    }
}
```

下面使用@AspectJ注解定义一个切面，如代码清单8-4所示。

代码清单8-4 PreGreetingAspect：切面

```
package com.smart.aspectj.aspectj;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect //①通过该注解将PreGreetingAspect标识为一个切面
public class PreGreetingAspect {

    @Before("execution(* greetTo(..))") //②定义切点和增强类型
    public void beforeGreeting(){ //③增强的横切逻辑
        System.out.println("How are you");
    }
}
```

我们“惊奇”地发现这个切面没有实现任何特殊的接口，它只是一个普通的 POJO。它特殊的地方在于使用了@AspectJ 注解。

首先，在 PreGreetingAspect 类定义处标注了@AspectJ 注解，这样，第三方处理程序就可以通过类是否拥有@AspectJ 注解判断其是否为一个切面，如①处所示。

其次，在 beforeGreeting()方法定义处标注了@Before 注解，并为该注解提供了成员值“execution(* greetTo(..))”，如②处所示。②处的注解提供了两个信息：@Before 注解表示该增强是前置增强，而成员值是一个@AspectJ 切点表达式。它的意思是：在目标类的 greetTo()方法上织入增强，greetTo()方法可以带任意的入参和任意的返回值。

最后，在③处的 beforeGreeting()方法是增强所使用的横切逻辑，该横切逻辑在目标方法前调用，可以通过图 8-1 描述这种关系。

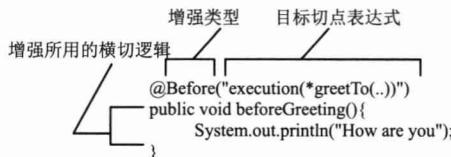


图 8-1 切面的信息构成

PreGreetingAspect 类通过注解和代码，将切点、增强类型和增强的横切逻辑糅合到一个类中，使切面的定义浑然天成。PreGreetingAspect 类相当于第 7 章中 BeforeAdvice、NameMatchMethodPointcut 及 DefaultPointcutAdvisor 三者联合表达的信息，在掌握 @AspectJ 定义切面的技术后，再去使用第 7 章中基于接口的切面技术，一定会顿生“马后桃花马前雪”的感慨。不过，通过基于接口切面的学习可以更加深刻地了解 Spring AOP 的内核技术。

下面通过 AspectJProxyFactory 为 NaiveWaiter 生成织入 PreGreetingAspect 切面的代理，如代码清单 8-5 所示。

代码清单 8-5 AspectJProxyTest

```

package com.smart.aspectj.example;
import org.springframework.aop.aspectj.annotation.AspectJProxyFactory;
import com.smart.NaiveWaiter;
import com.smart.Waiter;
public class AspectJProxyTest {
    Waiter target = new NaiveWaiter();
    AspectJProxyFactory factory = new AspectJProxyFactory();

    //①设置目标对象
    factory.setTarget(target);

    //②添加切面类
    factory.addAspect(PreGreetingAspect.class);

    //③生成织入切面的代理对象
    Waiter proxy = factory.getProxy();
}

```

```

    proxy.greetTo("John");
    proxy.serveTo("John");
}
}
}

```

第7章通过ProxyFactory织入基于接口描述的切面，此处则使用AspectJProxyFactory织入基于@AspectJ的切面。在①处设置了目标对象；在②处添加了一个切面类，该类必须是带@AspectJ注解的类；在③处就可以获取织入了切面的代理对象。

接下来直接通过代理对象调用greetTo()和serveTo()方法，将产生以下输出信息：

How are you ①

greet to John...

serving John...

通过①处的输出信息，可以知道代理对象的greetTo()方法已经被织入了切面类所定义的增强逻辑。

细心的读者可能会发现，在PreGreetingAspect切面类中定义的前置增强方法和通过BeforeAdvice接口定义的前置增强方法不一样，一个明显的区别就是PreGreetingAspect切面类的beforeGreeting()方法没有任何入参，而BeforeAdvice的接口方法before(Method method, Object[] args, Object obj)的入参提供了一条访问目标对象方法和入参的途径。难道使用@AspectJ定义的切面就没有办法访问目标对象连接点的信息了吗？当然不是，在本章后面的内容中，读者将学到相关的知识。

8.3.3 如何通过配置使用@AspectJ切面

虽然可以通过编程的方式织入切面，但在一般情况下，都是通过Spring的配置完成切面织入工作的。

```

<!-- ①目标Bean -->
<bean id="waiter" class="com.smart.NaiveWaiter" />
<!-- ②使用了@AspectJ注解的切面类 -->
<bean class="com.smart.aspectj.example.PreGreetingAspect" />
<!-- ③自动代理创建器，自动将@AspectJ注解切面类织入目标Bean中-->
<bean class="org.springframework.aop.aspectj.annotation.
  <!--AnnotationAwareAspectJAutoProxyCreator"/>

```

7.5.1节介绍了几个自动代理创建器，其中AnnotationAwareAspectJAutoProxyCreator能够将@AspectJ注解切面类自动织入目标Bean中。在这里，PreGreetingAspect是使用@AspectJ注解描述的切面类，而NaiveWaiter是匹配切点的目标类。

如果使用基于Schema的aop命名空间进行配置，那么事情就更简单了，如下：

```

<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop" ①
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://www.springframework.org/schema/aop ②
    http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">

```

```
<!--③基于@AspectJ切面的驱动器-->
<aop:aspectj-autoproxy />
<bean id="waiter" class="com.smart.NaiveWaiter" />
<bean class="com.smart.aspectj.example.PreGreetingAspect" />
</beans>
```

首先在配置文件中引入 aop 命名空间，如①和②处所示；然后通过 aop 命名空间的<aop:aspectj-autoproxy/>自动为 Spring 容器中那些匹配@AspectJ 切面的 Bean 创建代理，完成切面织入。当然，Spring 在内部依旧采用 AnnotationAwareAspectJAutoProxyCreator 进行自动代理的创建工作，但具体的实现细节已经被<aop:aspectj-autoproxy />隐藏起来。

<aop:aspectj-autoproxy/>有一个 proxy-target-class 属性，默认为 false，表示使用 JDK 动态代理技术织入增强；当配置为<aop:aspectj-autoproxy proxy-target-class="true"/>时，表示使用 CGLib 动态代理技术织入增强。不过即使 proxy-target-class 设置为 false，如果目标类没有声明接口，则 Spring 将自动使用 CGLib 动态代理。

8.4 @AspectJ 语法基础

@AspectJ 使用 Java 5.0 注解和正规的 AspectJ 的切点表达式语言描述切面，由于 Spring 只支持方法的连接点，所以 Spring 仅支持部分 AspectJ 的切点语言。本节将学习 AspectJ 切点表达式语言，以 AspectJ 5.0 版本为准。

8.4.1 切点表达式函数

AspectJ 5.0 的切点表达式由关键字和操作参数组成，如切点表达式 execution (* greetTo(..))，execution 为关键字，而“* greetTo(..)”为操作参数。在这里，execution 代表目标类执行某一方法，而“* greetTo(..)”描述目标方法的匹配模式串，二者联合起来表示目标类 greetTo()方法的连接点。为了描述方便，将 execution()称作函数，而将匹配串“* greetTo(..)”称作函数的入参。

Spring 支持 9 个@AspectJ 切点表达式函数，它们用不同的方式描述目标类的连接点。根据描述对象的不同，可以大致分为 4 种类型。

- 方法切点函数：通过描述目标类方法的信息定义连接点。
- 方法入参切点函数：通过描述目标类方法入参的信息定义连接点。
- 目标类切点函数：通过描述目标类类型的信息定义连接点。
- 代理类切点函数：通过描述目标类的代理类的信息定义连接点。

这 4 种类型的切点函数通过表 8-1 进行说明。

表 8-1 切点函数

类别	函数	入参	说明
方法切点 函数	execution()	方法匹配模式串	表示满足某一匹配模式的所有目标类方法连接点。如 execution(* greetTo(..))表示所有目标类中的 greetTo()方法
	@annotation()	方法注解类名	表示标注了特定注解的目标类方法连接点。如 @annotation(com.smart.anno.NeedTest) 表示任何标注了 @NeedTest 注解的目标类方法
方法入参 切点函数	args()	类名	通过判别目标类方法运行时入参对象的类型定义指定连接点。如 args(com.smart.Waiter) 表示所有有且仅有一个按类型匹配于 Waiter 入参的方法
	@args()	类型注解类名	通过判别目标类方法运行时入参对象的类是否标注特定注解来指定连接点。如 @args(com.smart.Monitorable) 表示任何这样的一个目标方法：它有一个入参且入参对象的类标注 @Monitorable 注解
目标类 切点函数	within()	类名匹配串	表示特定域下的所有连接点。如 within(com.smart.service.*) 表示 com.smart.service 包中的所有连接点，即包中所有类的所有方法；而 within(com.smart.service.*Service) 表示在 com.smart.service 包中所有以 Service 结尾的类的所有连接点
	target()	类名	假如目标类按类型匹配于指定类，则目标类的所有连接点匹配这个切点。如通过 target(com.smart.Waiter) 定义的切点、Waiter 及 Waiter 实现类 NaiveWaiter 中的所有连接点都匹配该切点
	@within()	类型注解类名	假如目标类按类型匹配于某个类 A，且类 A 标注了特定注解，则目标类的所有连接点匹配这个切点。如 @within(com.smart.Monitorable) 定义的切点，假如 Waiter 类标注了 @Monitorable 注解，则 Waiter 及 Waiter 实现类 NaiveWaiter 的所有连接点都匹配这个切点
	@target()	类型注解类名	假如目标类标注了特定注解，则目标类的所有连接点都匹配该切点。如 @target(com.smart.Monitorable)，假如 NaiveWaiter 标注了 @Monitorable，则 NaiveWaiter 的所有连接点都匹配这个切点
代理类 切点函数	this()	类名	代理类按类型匹配于指定类，则被代理的目标类的所有连接点都匹配该切点。这个函数比较难以理解，这里暂不举例，留待后面详解

除了表 8-1 中所列的函数外，@AspectJ 还有 call()、initialization()、preinitialization()、staticinitialization()、get()、set()、handler()、adviceexecution()、withincode()、cflow()、cflowbelow()、if()、@this() 及 @withincode() 等函数，这些函数在 Spring 中不能使用，否则会抛出 IllegalArgumentException 异常。在不特别声明的情况下，本书所讲的 @AspectJ 函数均指表 8-1 中所列的函数。

8.4.2 在函数入参中使用通配符

有些函数的入参可以接受通配符，@AspectJ 支持 3 种通配符。

- *：匹配任意字符，但它只能匹配上下文中的一个元素。
- ..：匹配任意字符，可以匹配上下文中的多个元素，但在表示类时，必须和*联合使用，而在表示入参时则单独使用。

- `+`: 表示按类型匹配指定类的所有类, 必须跟在类名后面, 如 `com.smart.Car+`。继承或扩展指定类的所有类, 同时还包括指定类本身。
- `@AspectJ` 函数按其是否支持通配符及支持的程度, 可以为以下 3 类。
 - 支持所有通配符: `execution()` 和 `within()`, 如 `within(com.smart.*)`、`within(com.smart.service..*.*Service+)` 等。
 - 仅支持“`+`”通配符: `args()`、`this()` 和 `target()`, 如 `args(com.smart.Waiter+)`、`target(java.util.List+)` 等。虽然这 3 个函数可以支持“`+`”通配符, 但其意义不大, 因为对于这些函数来说, 使用和不使用“`+`”都是一样的, 如 `target(com.smart.Waiter+)` 和 `target(com.smart.aspectj.Waiter)` 是等价的。
 - 不支持通配符: `@args()`、`@within()`、`@target()` 和 `@annotation()`, 如 `@args(com.smart.anno.NeedTest)` 和 `@within(com.smart.anno.NeedTest)`。

此外, `args()`、`this()`、`target()`、`@args()`、`@within()`、`@target()` 和 `@annotation()` 这 7 个函数除了可以指定类名外, 也可以指定变量名, 并将目标对象中的变量绑定到增强的方法中。关于参数绑定的内容将在 8.6 节介绍, 而函数的其他内容将在 8.5 节详解。

8.4.3 逻辑运算符

切点表达式由切点函数组成, 切点函数之间还可以进行逻辑运算, 组成复合切点。Spring 支持以下切点运算符。

- `&&`: 与操作符, 相当于切点的交集运算。如果在 Spring 的 XML 配置文件中使用切点表达式, 由于`&`是 XML 特殊字符, 所以使用转义字符`& &`表示。为了使用方便, Spring 提供了一个等效的运算符“`and`”。如 `within(com.smart..*) and args(String)` 表示在 `com.smart` 包下所有类(当前包及子孙包)拥有一个 `String` 入参的方法。
- `||`: 或操作符, 相当于切点的并集运算, `or` 是等效的操作符。如 `within(com. smart..*) || args(String)` 表示在 `com.smart` 包下所有类的方法, 或者所有拥有一个 `String` 入参的方法。
- `!`: 非操作符, 相当于切点的反集运算, `not` 是等效的操作符。如 `!within(com. smart.*)` 表示所有不在 `com.smart` 包下的方法。

在标准的`@AspectJ` 中并不提供 `and`、`or` 和 `not` 操作符, 它们是 Spring 为了在 XML 配置文件中方便定义切点表达式而特意添加的等价操作符。有意思的是, 和一般的语法表达不一样, 在 Spring 中使用 `and`、`or` 和 `not` 操作符时, 允许不在前后添加空格, 如 `within(com.smart..*) and not args(String)` 和 `within(com.smart..*) and not args(String)` 拥有相同的效果。虽然 Spring 接受这种表示方式, 但为了保证程序的可读性, 最好还是采用传统的习惯, 在操作符的前后添加空格。

**提示**

如果 not 位于切点表达式的开头，则必须在开头添加一个空格，否则将产生解析错误。如“not within(com.smart..*)”将产生解析错误，这应该是 Spring 解析的一个 Bug，在表达式开头添加空格后则可以通过解析，即“__not within (com.smart..*)”。

8.4.4 不同增强类型

第7章使用不同的接口描述各种增强类型，@AspectJ也为各种增强类型提供了不同的注解类，它们位于 org.aspectj.lang.annotation.*包中。这些注解类拥有若干个成员，可以通过这些成员完成定义切点信息、绑定连接点参数等操作。此外，这些注解的存留期限都是 RetentionPolicy.RUNTIME，标注目标都是 ElementType.METHOD。下面逐一学习@AspectJ所提供的几个增强注解。

1. @Before

前置增强，相当于 BeforeAdvice。Before 注解类拥有两个成员。

- value：该成员用于定义切点。
- argNames：由于无法通过 Java 反射机制获取方法入参名，所以如果在 Java 编译时未启用调试信息，或者需要在运行期解析切点，就必须通过这个成员指定注解所标注增强方法的参数名（注意二者名字必须完全相同），多个参数名用逗号分隔。

2. @AfterReturning

后置增强，相当于 AfterReturningAdvice。AfterReturning 注解类拥有 4 个成员。

- value：该成员用于定义切点。
- pointcut：表示切点的信息。如果显式指定 pointcut 值，那么它将覆盖 value 的设置值，可以将 pointcut 成员看作 value 的同义词。
- returning：将目标对象方法的返回值绑定给增强的方法。
- argNames：如前所述。

3. @Around

环绕增强，相当于 MethodInterceptor。Around 注解类拥有两个成员。

- value：该成员用于定义切点。
- argNames：如前所述。

4. @AfterThrowing

抛出增强，相当于 ThrowsAdvice。AfterThrowing 注解类拥有 4 个成员。

- value：该成员用于定义切点。
- pointcut：表示切点的信息。如果显式指定 pointcut 值，那么它将覆盖 value 的设置值。可以将 pointcut 成员看作 value 的同义词。

- throwing: 将抛出的异常绑定到增强方法中。
- argNames: 如前所述。

5. @After

Final 增强，不管是抛出异常还是正常退出，该增强都会得到执行。该增强没有对应的增强接口，可以把它看成 ThrowsAdvice 和 AfterReturningAdvice 的混合物，一般用于释放资源，相当于 try{}finally{}的控制流。After 注解类拥有两个成员。

- value: 该成员用于定义切点。
- argNames: 如前所述。

6. @DeclareParents

引介增强，相当于 IntroductionInterceptor。DeclareParents 注解类拥有两个成员。

- value: 该成员用于定义切点，它表示在哪个目标类上添加引介增强。
- defaultImpl: 默认的接口实现类。

除引介增强外，其他增强都很容易理解，将在本章后续内容中统一讲述。引介增强的使用比较特别，为此特别在 8.4.5 节准备了一个实例。

8.4.5 引介增强用法

请看以下两个接口及其实现类，如图 8-2 所示。

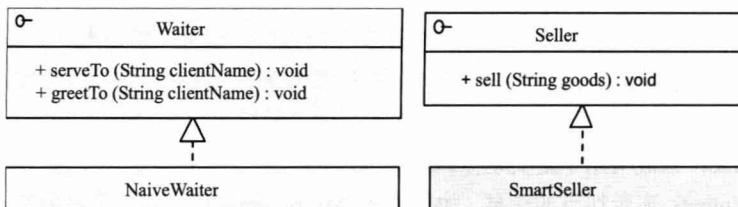


图 8-2 Waiter 和 Seller

假设希望 `NaiveWaiter` 能够同时充当售货员的角色，即通过切面技术为 `NaiveWaiter` 新增 `Seller` 接口的实现。在第 7 章中已经做过相似的工作，因此这里不在概念上作更多的解释，下面着手用`@AspectJ` 的引介增强来实现这一功能，如代码清单 8-6 所示。

代码清单 8-6 EnableSellerAspect

```

package com.smart.aspectj.basic;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;
import com.smart.Seller;
import com.smart.SmartSeller;

@Aspect
public class EnableSellerAspect {

    @DeclareParents(value="com.smart.NaiveWaiter", //①为NaiveWaiter添加接口实现
    
```

```

    defaultImpl=SmartSeller.class) //②默认的接口实现类
public Seller seller; //③要实现的目标接口
}

```

在 EnableSellerAspect 切面中，通过 @DeclareParents 为 NaiveWaiter 添加了一个需要实现的 Seller 接口，并指定其默认实现类为 SmartSeller，然后通过切面技术将 SmartSeller 融合到 NaiveWaiter 中，这样 NaiveWaiter 就实现了 Seller 接口。

在 Spring 配置文件中配置好切面和 NaiveWaiter Bean，如下：

```

<aop:aspectj-autoproxy/>
<bean id="waiter" class="com.smart.NaiveWaiter"/>
<bean class="com.smart.aspectj.basic.EnableSellerAspect"/>

```

运行以下测试代码：

```

package com.smart.aspectj.basic;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.smart.Seller;
import com.smart.Waiter;
public class DeclaredParentsTest {
    public static void main(String[] args) {
        String configPath = "com/smart/aspectj/basic/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter)ctx.getBean("waiter");
        waiter.greetTo("John");
        Seller seller = (Seller)waiter; //①可以成功地进行强制类型转换
        seller.sell("Beer", "John");
    }
}

```

代码成功执行，并输出以下信息：

```

NaiveWaiter:greet to John...
SmartSeller: sell Beer to John...

```

可见，NaiveWaiter 已经成功地新增了 Seller 接口的实现。

8.5 切点函数详解

切点函数是 AspectJ 表达式语言的核心，也是使用 @AspectJ 进行切面定义的难点，本节通过具体实例来学习切点函数的相关知识。为了方便讲解，假设可供被增强的目标类包括 7 个类，这些目标类都位于 com.smart.* 包中，如图 8-3 所示。

在这些类中，除 SmartSeller#showGoods() 方法是 protected 外，其他的方法都是 public。

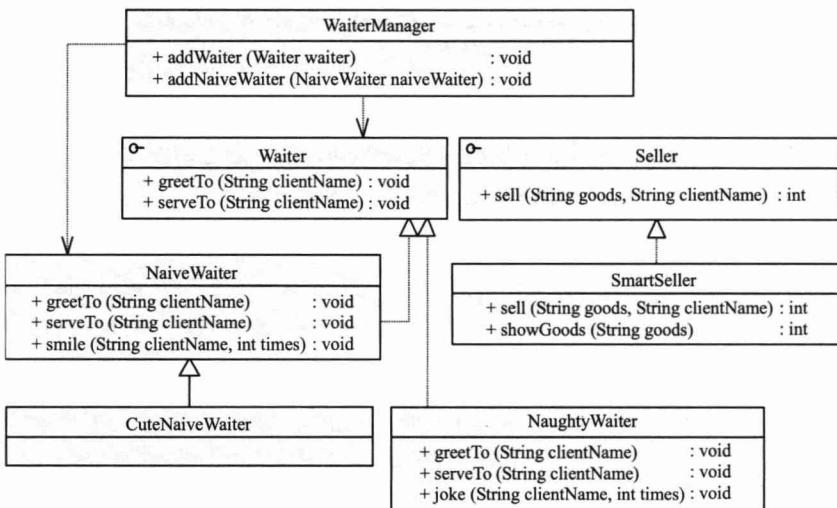


图 8-3 Waiter 和 Seller 类图

8.5.1 @annotation()

@annotation 表示标注了某个注解的所有方法。通过一个实例说明@annotation()的用法。TestAspect 定义了一个后置增强切面，该增强将应用到标注了 NeedTest 的目标方法中。

```

package com.smart.aspectj.fun;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class TestAspect {
    @AfterReturning("@annotation(com.smart.anno.NeedTest)") //①后置增强切面
    public void needTestFun() {
        System.out.println("needTestFun() executed!");
    }
}
  
```

假设 NaughtyWaiter#greetTo()方法标注了@NeedTest 注解，而 NaiveWaiter#greetTo()方法没有标注@NeedTest 注解，如代码清单 8-7 所示。

代码清单 8-7 标注了@NeedTest注解的NaughtyWaiter

```

package com.smart;
import com.smart.anno.NeedTest;
public class NaughtyWaiter implements Waiter {
    @NeedTest
    public void greetTo(String clientName) {
        System.out.println("NaughtyWaiter:greet to "+clientName+"...");
    }
    ...
}
  
```

通过 Spring 配置自动应用切面。

```
<aop:aspectj-autoproxy />
<bean id="naiveWaiter" class="com.smart.NaiveWaiter" />
<bean id="naughtyWaiter" class="com.smart.NaughtyWaiter" />
<bean class="com.smart.aspectj.fun.TestAspect" />
```

运行代码清单 8-8 中的代码。

代码清单 8-8 PointcutFunTest：测试代码

```
package com.smart.aspectj.fun;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.smart.Seller;
import com.smart.Waiter;
public class PointcutFunTest{
    @Test
    public void pointcut {
        String configPath = "com/smart/aspectj/fun/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
        Waiter naughtyWaiter = (Waiter) ctx.getBean("naughtyWaiter");
        naiveWaiter.greetTo("John"); //①该方法未被织入增强
        naughtyWaiter.greetTo("Tom"); //②该方法被织入增强
    }
}
```

输出以下信息：

```
NaiveWaiter:greet to John...
NaughtyWaiter:greet to Tom... ① ← 对应NaughtyWaiter 的greetTo()方法
needTestFun() executed!
```

从以上信息中可以获知，切面被正确地织入 NaughtyWaiter#greetTo()方法中。

8.5.2 execution()

execution()是最常用的切点函数，其语法如下：

```
execution(<修饰符模式>? <返回类型模式> <方法名模式>(<参数模式>) <异常模式>?)
```

除了返回类型模式、方法名模式和参数模式外，其他项都是可选的。与其直接讲解该方法的使用规则，还不如通过一个个具体的例子来理解。下面给出各种使用 execution() 函数的实例。

1. 通过方法签名定义切点

- execution(public * *(..)): 匹配所有目标类的 public 方法，但不匹配 SmartSeller 和 protected void showGoods()方法。第一个*代表返回类型；第二个*代表方法名；而..代表任意入参的方法。
- execution(* *To(..)): 匹配目标类所有以 To 为后缀的方法，它匹配 NaiveWaiter 类和 NaughtyWaiter 的 greetTo()和 serveTo()方法。第一个*代表返回类型；而*To 代表任意以 To 为后缀的方法。

2. 通过类定义切点

- execution(* com.smart.Waiter.*(..)): 匹配 Waiter 接口的所有方法，它匹配 NaiveWaiter 和 NaughtyWaiter 类的 greetTo() 和 serveTo() 方法。第一个 * 代表返回任意类型；com.smart.Waiter.* 代表 Waiter 接口中的所有方法。
- execution(* com.smart.Waiter+.*(..)): 匹配 Waiter 接口及其所有实现类的方法，它不但匹配 NaiveWaiter 和 NaughtyWaiter 类的 greetTo() 和 serveTo() 这两个 Waiter 接口定义的方法，同时还匹配 NaiveWaiter#smile() 和 NaughtyWaiter#joke() 这两个不在 Waiter 接口中定义的方法。

3. 通过类包定义切点

在类名模式串中，“.*” 表示包下的所有类，而 “..*” 表示包、子孙包下的所有类。

- execution(* com.smart.*(..)): 匹配 com.smart 包下所有类的所有方法。
- execution(* com.smart..*(..)): 匹配 com.smart 包、子孙包下所有类的所有方法，如 com.smart.dao、com.smart.service 及 com.smart.dao.user 包下所有类的所有方法都匹配。当 “..” 出现在类名中时，后面必须跟 “*”，表示包、子孙包下的所有类。
- execution(* com..*.*Dao.find*(..)): 匹配包名前缀为 com 的任何包下类名后缀为 Dao 的方法，方法名必须以 find 为前缀。如 com.smart.UserDao#findByUserId()、com.smart.dao.ForumDao#findById() 方法都匹配切点。

4. 通过方法入参定义切点

切点表达式中的方法入参部分比较复杂，可以使用 “*” 和 “..” 通配符。其中，“*” 表示任意类型的参数；而 “..” 表示任意类型的参数且参数个数不限。

- execution(* joke(String,int)): 匹配 joke(String,int) 方法，且 joke() 方法的第一个入参是 String，第二个入参是 int。它匹配 NaughtyWaiter#joke(String,int) 方法。如果方法中的入参类型是 java.lang 包下的类，则可以直接使用类名；否则必须使用全限定类名，如 joke(java.util.List,int)。
- execution(* joke(String,*)): 匹配目标类中的 joke() 方法，该方法的第一个入参为 String，第二个入参可以是任意类型，如 joke(String s1, String s2) 和 joke(String s1, double d2) 都匹配，但 joke(String s1, double d2, String s3) 不匹配。
- execution(* joke(String..)): 匹配目标类中的 joke() 方法，该方法的第一个入参为 String，后面可以有任意个入参且入参类型不限，如 joke(String s1)、joke(String s1, String s2) 和 joke(String s1, double d2, String s3) 都匹配。
- execution(* joke(Object+)): 匹配目标类中的 joke() 方法，方法拥有一个入参，且入参是 Object 类型或该类的子类。它匹配 joke(String s1) 和 joke(Client c)。如果定义的切点是 execution(* joke(Object))，则只匹配 joke(Object object)，而不匹配 joke(String cc) 或 joke(Client c)。

8.5.3 args()和@args()

args()函数的入参是类名，而@args()函数的入参必须是注解类的类名。虽然 args()允许在类名后使用“+”通配符，但该通配符在此处没有意义，添加和不添加效果都一样。

1. args()

该函数接收一个类名，表示目标类方法入参对象是指定类（包含子类）时，切点匹配，如下面的例子：

```
args(com.smart.Waiter)
```

表示运行时入参是 Waiter 类型的方法，它和 execution(* *(com.smart.Waiter)) 的区别在于后者是针对类方法的签名而言的，而前者则针对运行时的入参类型而言。如 args(com.smart.Waiter) 既匹配 addWaiter(Waiter waiter)，又匹配 addNaiveWaiter(NaiveWaiter naiveWaiter)；而 execution(* *(com.smart.Waiter)) 只匹配 addWaiter(Waiter waiter)。实际上，args(com.smart.Waiter) 等价于 execution(* *(com.smart.Waiter+))，当然也等价于 args(com.smart.Waiter+)。

2. @args()

该函数接收一个注解类的类名，当方法的运行时入参对象标注了指定的注解时，匹配切点。这个切点函数的匹配规则不太容易理解，通过图 8-4 对此进行详细讲解。

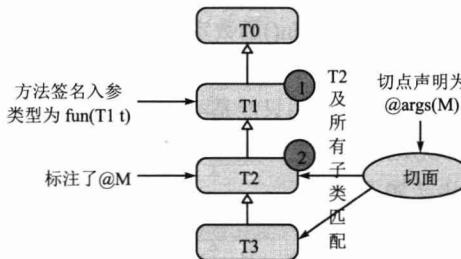


图 8-4 @arg(M) 匹配示意图 (1)

T0、T1、T2、T3 具有如图 8-4 所示的继承关系，假设目标类方法的签名为 fun(T1 t)，它的入参为 T1，而切面的切点定义为 @args(M)，T2 类标注了 @M。当 fun(T1 t) 的传入对象是 T2 或 T3 时，方法匹配 @args(M) 声明所定义的切点。

假设方法签名为 fun(T1 t)，入参为 T1，而标注 @M 的类是 T0，当 fun(T1 t) 传入 T1、T2、T3 的实例时，均不匹配切点 @args(M)。

在类的继承树中，①处为方法签名中入参类型在类继承树中的位置，称之为入参类型点；而②处为标注了 @M 注解的类在类继承树中的位置，称之为注解点。判断方法在运行时是否匹配 @args(M) 切点，可以根据①和②在类继承树中的相对位置来判别。

(1) 如果在类继承树中注解点②高于入参类型点①，则该目标方法不可能匹配切点 @args(M)，如图 8-5 所示。

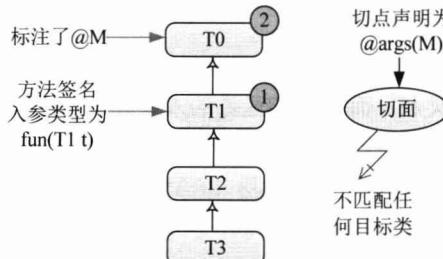


图 8-5 @arg(M) 匹配示意图 (2)

(2) 如果在类继承树中注解点②低于入参类型点①，则注解点所在类及其子孙类作为方法入参时，该方法匹配切点@args(M)，如图 8-4 所示。

下面举一个具体的例子。假设定义这样的切点：@args(com.smart.Monitorable)，如果 NaiveWaiter 标注了@Monitorable，则对于 WaiterManager#addWaiter(Waiter w)方法来说，如果入参是 NaiveWaiter 或其子类对象，则该方法匹配切点；如果入参是 NaughtyWaiter 对象，则不匹配切点。如果 Waiter 标注了@Monitorable，但 NaiveWaiter 未标注@Monitorable，则 WaiterManager#addNaiveWaiter(NaiveWaiter w)不匹配切点，这是因为注解点（Waiter）高于入参类型点（NaiveWaiter）。

8.5.4 within()

通过类匹配模式串声明切点，within()函数定义的连接点是针对目标类而言的，而非针对运行期对象的类型而言，这一点和 execution()是相同的。但和 execution()函数不同的是，within()所指定的连接点最小范围只能是类，而 execution()所指定的连接点可以大到包，小到方法入参。所以从某种意义上说，execution()函数的功能涵盖了 within()函数的功能。within()函数的语法如下：

```
within(<类匹配模式>)
```

形如 within(com.smart.NaiveWaiter)，是 within()函数所能表达的最小粒度。如果试图用 within()匹配方法级别的连接点，如 within(com.smart.NaiveWaiter.greet*)，那么将会产生解析错误。

下面是一些使用 within()函数的实例。

- within(com.smart.NaiveWaiter): 匹配目标类 NaiveWaiter 的所有方法。如果切点调整为 within(com.smart.Waiter)，则 NaiveWaiter 和 NaughtyWaiter 中的所有方法都不匹配。而 Waiter 本身是接口，不可能实例化，所以 within(com.smart.Waiter)的声明是无意义的。
- within(com.smart.*): 匹配 com.smart 包中的所有类，但不包括子孙包，所以 com.smart.service 包中类的方法不匹配这个切点。
- within(com.smart..*): 匹配 com.smart 包及子孙包中的类，所以 com.smart.service、com.smart.dao 及 com.smart.service.fourm 等包中所有类的方法都匹配这个切点。

8.5.5 @within()和@target()

除@annotation()和@args()函数外，还有另外两个用于注解的切点函数，分别是@target()和@within()。和@annotation()及@args()函数一样，它们也只接受注解类名作为入参。其中，@target(M)匹配任意标注了@M 的目标类，而@within(M)匹配标注了@M 的类及子孙类。

@target(M)切点的匹配规则如图 8-6 所示。

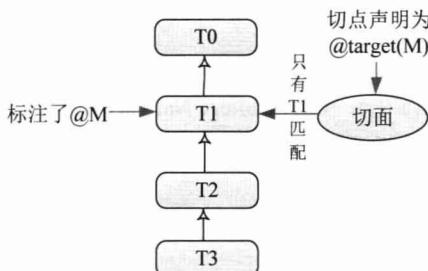


图 8-6 @target(M) 匹配目标类示意图

假设 NaiveWaiter 标注了@Monitorable，而其子类 CuteNaiveWaiter 没有标注@Monitorable，则@target(com.smart.Monitorable)匹配 NaiveWaiter 类的所有方法，但不匹配 CuteNaiveWaiter 类的方法。

@within(M)切点的匹配规则如图 8-7 所示。

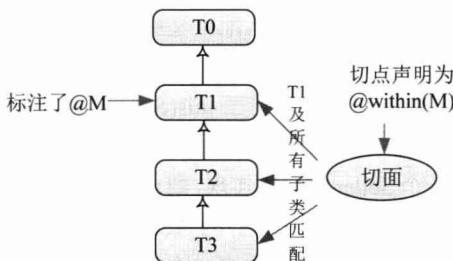


图 8-7 @within(M) 匹配目标类示意图

假设 NaiveWaiter 标注了@Monitorable，而其子类 CuteNaiveWaiter 没有标注@Monitorable，则@within(com.smart.Monitorable)不但匹配 NaiveWaiter 类中的所有方法，也匹配 CuteNaiveWaiter 类中的所有方法。

有一个特别值得注意的地方是，如果标注@M 注解的是一个接口，则所有实现该接口的类并不匹配@within(M)。假设 Waiter 标注了@Monitorable 注解，但 NaiveWaiter、NaughtyWaiter 及 CuteNaiveWaiter 这些接口实现类都没有标注@Monitorable，则@within(com.smart.Monitorable)和@target(com.smart.Monitorable)都不匹配 NaiveWaiter、NaughtyWaiter 及 CuteNaiveWaiter。这是因为@within()、@target()及@annotation()函数都是针对目标类而言的，而非针对运行时的引用类型而言的，这点区别需要在开发中特别注意。

8.5.6 target()和 this()

target()切点函数通过判断目标类是否按类型匹配指定类来决定连接点是否匹配，而this()函数则通过判断代理类是否按类型匹配指定类来决定是否和切点匹配。二者都仅接受类名的入参，虽然类名可以带“+”通配符，但对于这两个函数来说，使用与不使用“+”通配符，效果完全相同。

1. target()

target(M)表示如果目标类按类型匹配于 M，则目标类的所有方法都匹配切点。下面通过一些例子理解 target(M)的匹配规则。

- target(com.smart.Waiter): NaiveWaiter、NaughtyWaiter 及 CuteNaiveWaiter 的所有方法都匹配切点，包括那些未在 Waiter 接口中定义的方法，如 NaiveWaiter#simple() 和 NaughtyWaiter#joke()方法。
- target(com.smart.Waiter+): 与 target(com.smart.Waiter)是等价的。

2. this()

根据 Spring 的官方文档，this()函数判断代理对象的类是否按类型匹配于指定类，如果匹配，则代理对象的所有连接点匹配切点。但通过实验发现，实际情况和文档有所出入。如声明一个切点 this(com.smart.NaiveWaiter)，如果不使用 CGLib 动态代理，则生成的代理对象属于 Waiter 类型，而非 NaiveWaiter 类型，这一点可以简单地通过 instanceof 操作符进行判断。但是，我们发现 NaiveWaiter 中所有的方法都被织入了增强。

一般情况下，使用 this()和 target()来匹配定义切点，二者是等效的。

- (1) target(com.smart.Waiter)等价于 this(com.smart.Waiter)。
- (2) target(com.smart.NaiveWaiter)等价于 this(com.smart.NaiveWaiter)。

二者的区别体现在通过引介切面产生代理对象时的具体表现。如果通过 8.4.5 节的方法为 NaiveWaiter 引介一个 Seller 接口的实现，则 this(com.smart.Seller)匹配 NaiveWaiter 代理对象的所有方法，包括 NaiveWaiter 本身 greetTo()、serverTo()方法，以及通过 Seller 接口引入的 sell()方法；而 target(com.smart.Seller)不匹配通过引介切面产生的 NaiveWaiter 代理对象。

下面通过具体的实例来了解这一微妙的区别。EnableSellerAspect 是为 NaiveWaiter 添加 Seller 接口实现的引介切面，如下：

```
package com.smart.aspectj.fun;
...
@Aspect
public class EnableSellerAspect {
    @DeclareParents(value = "com.smart.NaiveWaiter",
                   defaultImpl = SmartSeller.class)
    public static Seller seller;
}
```

TestAspect 是通过判断运行期代理对象所属类型来定义切点的切面，如代码清单 8-9 所示。

代码清单 8-9 TestAspect：通过this()指定切点

```
package com.smart.aspectj.fun;
...
@Aspect
public class TestAspect {
    //①后置增强，织入任何运行期对象为Seller类型的Bean中
    @AfterReturning("this(com.smart.Seller)")
    public void thisTest(){
        System.out.println("thisTest() executed!");
    }
}
```

在 Spring 中配置这两个切面和 NaiveWaiter，如下：

```
<aop:aspectj-autoproxy>
<bean id="naiveWaiter" class="com.smart.NaiveWaiter" />
<bean class="com.smart.aspectj.fun.EnableSellerAspect"/>
<bean class="com.smart.aspectj.fun.TestAspect" />
```

EnableSellerAspect 切面为 NaiveWaiter 引介 Seller 接口产生一个实现 Seller 接口的代理对象，TestAspect 在判断出 NaiveWaiter 这个代理对象实现 Seller 接口后，就将其切面织入这个代理对象中，所以最终 NaiveWaiter 的代理对象其实共织入了两个切面。在这里，我们忽略了多切面织入顺序的问题，读者将在本章后续的内容中了解到这方面的知识。

运行以下测试代码：

```
String configPath = "com/smart/aspectj/fun/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
naiveWaiter.greetTo("John");
naiveWaiter.serveTo("John");
((Seller)naiveWaiter).sell("Beer", "John");
```

输出以下信息：

```
NaiveWaiter:greet to John...
thisTest() executed!
NaiveWaiter:serving John...
thisTest() executed!
SmartSeller: sell Beer to John...
thisTest() executed!
```

可见代理对象的 3 个方法都织入了代码清单 8-9 中通过 this() 函数定义的切面。

8.6 @AspectJ 进阶

@AspectJ 可以使用切点函数定义切点，还可以使用逻辑运算符对切点进行复合运算得到复合切点。为了在切面中重用切点，还可以对切点进行命名，以便在其他地方引

用定义过的切点。当一个连接点匹配多个切点时，需要考虑织入顺序的问题，另外一个重要问题是如何在增强中访问连接点上下文的信息。

8.6.1 切点复合运算

使用切点复合运算符，将拥有强大而灵活的切点表达能力。以下是一个使用了复合切点的切面，如代码清单 8-10 所示。

代码清单 8-10 TestAspect：切点复合运算

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class TestAspect {
    @After("within(com.smart.*) " +
           "+ && execution(* greetTo(..)))") //①与运算
    public void greeToFun() {
        System.out.println("--greetToFun() executed!--");
    }

    @Before(" !target(com.smart.NaiveWaiter) " +
            "+ && execution(* serveTo(..))") //②非与运算
    public void notServeInNaiveWaiter() {
        System.out.println("--notServeInNaiveWaiter() executed!--");
    }

    @AfterReturning("target(com.smart.Waiter) || " +
                   " target(com.smart.Seller)") //③或运算
    public void waiterOrSeller() {
        System.out.println("--waiterOrSeller() executed!--");
    }
}
```

在①处，通过`&&`运算符定义了一个匹配 com.smart 包中所有 `greetTo()` 方法的切点；在②处，通过`!`和`&&`运算符定义了一个匹配所有 `serveTo()` 方法并且该方法不位于 `NaiveWaiter` 目标类的切点；在③处，通过`||`运算符定义了一个匹配 `Waiter` 和 `Seller` 接口实现类所有连接点的切点。

8.6.2 命名切点

在前面所举的例子中，切点直接声明在增强方法处，这种切点声明方式称为匿名切点，匿名切点只能在声明处使用。如果希望在其他地方重用一个切点，则可以通过 `@Pointcut` 注解及切面类方法对切点进行命名。以下是一个具体的实例，如代码清单 8-11 所示。

代码清单 8-11 TestNamePointcut

```

package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Pointcut;
public class TestNamePointcut {
    @Pointcut("within(com.smart.*)") ①
    private void inPackage(){}
    @Pointcut("execution(* greetTo(..))") ②
    protected void greetTo(){}
    @Pointcut("inPackage() and greetTo()") ③
    public void inPkgGreetTo(){}
}

```

通过注解方法 `inPackage()` 对该切点进行命名，方法可视域修饰符为 `private`，表明该命名切点只能在本切面类中使用

通过注解方法 `greetTo()` 对该切点进行命名，方法可视域修饰符为 `protected`，表明该命名切点可以在当前包中的切面类、子切面类中使用

引用命名切点定义的切点，本切点也是命名切点，它对应的可视域为 `public`

在代码清单 8-11 中定义了 3 个命名切点，命名切点的使用类方法作为切点的名称，此外方法的访问修饰符还控制了切点的可引用性，这种可引用性和类方法的可访问性相同，如 `private` 的切点只能在本类中引用，`public` 的切点可以在任何类中引用。命名切点仅利用方法名及访问修饰符的信息，所以习惯上方法的返回类型为 `void`，并且方法体为空。可以通过图 8-8 更直观地了解命名切点的结构。

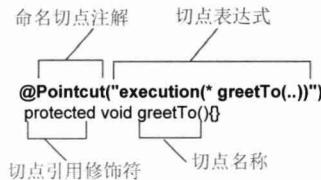


图 8-8 命名切点的结构

在③处，`inPkgGreetTo()`的切点引用了同类中的 `greetTo()`切点，而 `inPkgGreetTo()`切点可以被任何类引用。用户还可以扩展 `TestNamePointcut` 类，通过类的继承关系定义更多的切点。

命名切点定义好后，就可以在定义切面时通过名称引用切点，请看下面的实例：

```

package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class TestAspect {
    @Before("TestNamePointcut.inPkgGreetTo()") //①
    public void pkgGreetTo(){
        System.out.println("--pkgGreetTo() executed!--");
    }
    @Before("!target(com.smart.NaiveWaiter) && TestNamePointcut.inPkgGreetTo()") //②
    public void pkgGreetToNotNaiveWaiter(){
        System.out.println("--pkgGreetToNotNaiveWaiter() executed!--");
    }
}

```

在①处，引用了 `TestNamePointcut.inPkgGreetTo()`切点；而在②处，在复合运算中使用了命名切点。

8.6.3 增强织入的顺序

一个连接点可以同时匹配多个切点，切点对应的增强在连接点上的织入顺序是如何安排的呢？这个问题需要分 3 种情况讨论。

- 如果增强在同一个切面类中声明，则依照增强在切面类中定义的顺序进行织入。
- 如果增强位于不同的切面类中，且这些切面类都实现了 org.springframework.core.Ordered 接口，则由接口方法的顺序号决定（顺序号小的先织入）。
- 如果增强位于不同的切面类中，且这些切面类没有实现 org.springframework.core.Ordered 接口，则织入的顺序是不确定的。

可以通过图 8-9 描述这种织入的规则。切面类 A 和 B 都实现了 Ordered 接口，切面类 A 对应序号为 1，切面类 B 对应序号为 2，切面类 A 按顺序定义了 3 个增强，切面类 B 按顺序定义两个增强，这 5 个增强对应的切点都匹配某个目标类的连接点，则增强织入的顺序为图中虚线所示。

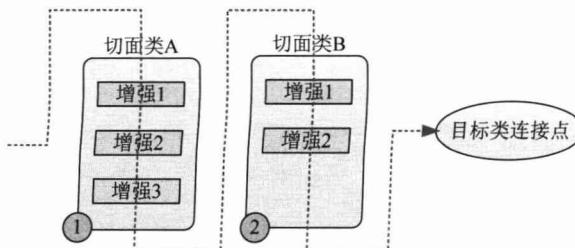


图 8-9 增强织入顺序

8.6.4 访问连接点信息

AspectJ 使用 org.aspectj.lang.JoinPoint 接口表示目标类连接点对象。如果是环绕增强，则使用 org.aspectj.lang.ProceedingJoinPoint 表示连接点对象，该类是 JoinPoint 的子接口。任何增强方法都可以通过将第一个入参声明为 JoinPoint 访问连接点上下文信息。先来了解一下这两个接口的主要方法。

1. JoinPoint

- java.lang.Object[] getArgs(): 获取连接点方法运行时的入参列表。
- Signature getSignature(): 获取连接点的方法签名对象。
- java.lang.Object getTarget(): 获取连接点所在的目标对象。
- java.lang.Object getThis(): 获取代理对象本身。

2. ProceedingJoinPoint

ProceedingJoinPoint 继承于 JoinPoint 子接口，它新增了两个用于执行连接点方法的方法。

- `java.lang.Object proceed() throws java.lang.Throwable`: 通过反射执行目标对象的连接点处的方法。
 - `java.lang.Object proceed(java.lang.Object[] args) throws java.lang.Throwable`: 通过反射执行目标对象连接点处的方法，不过使用新的入参替换原来的入参。
- 来看一个具体的实例，如代码清单8-12所示。

代码清单8-12 TestAspect: 访问连接点对象

```

package com.smart.aspectj.advanced;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class TestAspect {
    @Around("execution(* greetTo(..)) && target(com.smart.NaiveWaiter)") ①
    public void joinPointAccess(ProceedingJoinPoint pjp) throws Throwable{ ②
        System.out.println("-----joinPointAccess-----");
        System.out.println("args[0]:"+pjp.getArgs()[0]);
        System.out.println("signature:"+pjp.getTarget().getClass());
        pjp.proceed(); ④ ←
        System.out.println("-----joinPointAccess-----");
    }
}

```

在①处声明了一个环绕增强；在②处增强方法的第一个入参声明为 `PreceedingJoinPoint` 类型（注意一定要在第一个位置）；在③处通过连接点对象 `pjp` 访问连接点信息；在④处通过连接点调用目标对象的方法。

执行以下测试代码：

```

String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
naiveWaiter.greetTo("John");

```

输出以下信息：

```

-----joinPointAccess-----
args[0]:John
signature:class com.smart.NaiveWaiter
NaiveWaiter:greet to John... ① ← 对应pjp.proceed();
-----joinPointAccess-----

```

8.6.5 绑定连接点方法入参

在介绍切点函数时说过 `args()`、`this()`、`target()`、`@args()`、`@within()`、`@target()` 和 `@annotation()` 这 7 个函数除了可以指定类名外，还可以指定参数名，将目标对象连接点上的方法入参绑定到增强的方法中。

其中，`args()`用于绑定连接点方法的入参；`@annotation()`用于绑定连接点方法的注解

对象；而@args()用于绑定连接点方法入参的注解。来看一个 args()绑定参数的实例，如代码清单 8-13 所示。

代码清单 8-13 TestAspect：绑定连接点参数

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before; 绑定连接点参数，首先，args(name, num,...)根据
@Aspect ②处的增强方法入参找到 name 和 num 对应的类型，以得到真实的切点表达式：target(com.smart.
public class TestAspect { 在该增强方法织入目标连接点时，增强方法可以通过
    @Before("target(com.smart.NaiveWaiter) && args(String,int,...)") ①← num 和 name 访问到连接点的方法入参
        public void bindJoinPointParams(int num, String name) (②←
            System.out.println("----bindJoinPointParams()----");
            System.out.println("name:" + name);
            System.out.println("num:" + num);
            System.out.println("----bindJoinPointParams()----");
        }
    }
}
```

在①处，通过 args(name,num,...)进行连接点参数的绑定。和前面讲述的方式不同，当 args()函数入参为参数名时，共包括两方面的信息。

(1) 连接点匹配规则信息：连接点方法的第一个入参是 String 类型，第二个入参是 int 类型。

(2) 连接点方法入参和增强方法入参的绑定信息：连接点方法的第一个入参绑定到增强方法的 name 参数上，第二个入参绑定到增强方法的 num 参数上。

切点匹配和参数绑定的过程是这样的：首先，args()根据参数名称在增强方法中查到名称相同的入参并获知对应的类型，这样就知道了匹配连接点方法的入参类型；其次，连接点方法入参类型所在的位置则由参数名在 args()函数中声明的位置决定。代码清单 8-13 中 args(name,num) 匹配的目标类方法的第一个入参必须是 String 类型，第二个入参必须是 int 类型，如 smile(String name,int times)匹配，而 smile(int times ,String name)不匹配。可以通过图 8-10 详细了解这一有趣的匹配过程。

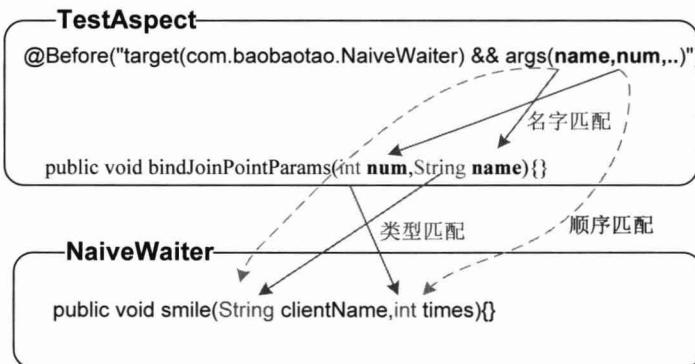


图 8-10 切点匹配和参数绑定过程

和 args()一样，其他可以绑定连接点参数的切点函数（如@args()和 target()等），当指定参数名时，就同时具有匹配切点和绑定参数双重功能。

运行下面的测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
NaiveWaiter naiveWaiter = (NaiveWaiter) ctx.getBean("naiveWaiter");
naiveWaiter.smile("John",2);
```

将看到以下输出信息：

```
----bindJoinPointParams()----
name:John
num:2
----bindJoinPointParams()----
NaiveWaiter:smile to John2times...
```

可见，增强方法按预期绑定了 NaiveWaiter.smile(String name,int times)方法的运行期入参。



提示

为了保证实例能成功执行，必须启用 CGLib 动态代理：`<aop:aspectj-autoproxy proxy-target-class="true" />`。因为该实例需要对 NaiveWaiter 类进行代理（因为 NaiveWaiter#smile()方法不是 Waiter 接口的方法），所以必须使用 CGLib 动态代理生成子类的代理方法。

8.6.6 绑定代理对象

使用 this()或 target()函数可绑定被代理对象实例，在通过类实例名绑定对象时，依然具有原来连接点匹配的功能，只不过类名是通过增强方法中同名入参的类型间接决定罢了。这里通过 this()函数来了解对象绑定的用法。

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import com.smart.Waiter;
@Aspect
public class TestAspect {
    @Before("this(waiter)") ①
    public void bindProxyObj(Waiter waiter){ ②
        System.out.println("----bindProxyObj()----");
        System.out.println(waiter.getClass().getName());
        System.out.println("----bindProxyObj()----");
    }
}
```

通过②处查找出waiter对应的类型为 Waiter，因而切点表达式为 this(Waiter)。当增强方法织入目标连接点时，增强方法通过 waiter 入参绑定目标对象

①处的切点表达式首先按类变量名查找②处增强方法的入参列表，进而获取类变量名对应的类为 com.smart.Waiter，这样就知道了切点的定义为 this(com.smart.Waiter)，即所有代理对象为 Waiter 类的所有方法匹配该切点。②处的增强方法通过 waiter 入参绑定目标对象。

可见 NaiveWaiter 的所有方法都匹配①处的切点。运行以下测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
naiveWaiter.greetTo("John");
```

可以看到如下输出信息：

```
----bindProxyObj()----
com.smart.NaiveWaiter$$EnhancerByCGLIB$$6758891b
----bindProxyObj()----
NaiveWaiter:greet to John...
```

按相似的方法使用 target() 函数进行绑定。

8.6.7 绑定类注解对象

@within() 和 @target() 函数可以将目标类的注解对象绑定到增强方法中，下面通过 @within() 函数演示注解绑定的操作。

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import com.smart.Monitorable;
@Aspect
public class TestAspect {
    @Before("@within(m)") ① ←
    public void bindTypeAnnoObject(Monitorable m) { ②
        System.out.println("----bindTypeAnnoObject()----");
        System.out.println(m.getClass().getName());
        System.out.println("----bindTypeAnnoObject()----");
    }
}
```

通过②处查找出 m 对应 Monitorable 类型的注解，因而真实的切点表达式为 @within (Monitorable)。当增强方法织入目标连接点时，增强方法通过 m 入参可以引用到连接点

在 NaiveWaiter 类中标注了 @Monitorable 注解，所有的 NaiveWaiter Bean 都匹配切点，其 Monitorable 注解对象将绑定到增强方法中。运行以下代码，即可查看绑定注解对象：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
((NaiveWaiter)naiveWaiter).greetTo("John");
```

输出以下信息：

```
----bindTypeAnnoObject()----
$Proxy3
----bindTypeAnnoObject()----
NaiveWaiter:greet to John...
```

从输出信息中我们发现了一个秘密，即使用 CGLib 代理 NaiveWaiter 时，其类的注解 Monitorable 对象也被代理了。

8.6.8 绑定返回值

在后置增强中，可以通过 returning 绑定连接点方法的返回值，如下：

```
@AfterReturning(value="target(com.smart.SmartSeller)",returning="retVal") //①
public void bingReturnValue(int retVal){ //②
    System.out.println("----bindException()----");
    System.out.println("returnValue:"+retVal);
    System.out.println("----bindException()----");
}
```

①处和②处的名字必须相同，此外，②处 retVal 的类型必须和连接点方法的返回值类型匹配。运行下面的测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
SmartSeller seller = (SmartSeller) ctx.getBean("seller");
seller.sell("Beer", "John");
```

可以看到以下输出信息：

```
SmartSeller: sell Beer to John...
----bingReturnValue()----
returnValue:100
----bingReturnValue()----
```

可见目标连接点 Seller#sell()方法返回的入参被成功绑定到增强方法中。

8.6.9 绑定抛出的异常

和通过切点函数绑定连接点信息不同，连接点抛出的异常必须使用 AfterThrowing 注解的 throwing 成员进行绑定，如代码清单 8-14 所示。

代码清单 8-14 TestAspect：绑定异常对象

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class TestAspect {
    @AfterThrowing(value="target(com.smart.SmartSeller)",throwing="iae") //①
    public void bindException(IllegalArgumentException iae){ //②
        System.out.println("----bindException()----");
        System.out.println("exception:"+iae.getMessage());
        System.out.println("----bindException()----");
    }
}
```

①处 throwing 指定的异常名和②处入参的异常名相同，这个异常增强只在连接点抛出异常 instanceof IllegalArgumentException 时才匹配，增强方法通过 iae 参数可以访问抛出的异常对象。

在 SmartSeller 中添加一个抛出异常的测试方法，如下：

```
package com.smart;
public class SmartSeller implements Seller {
```

```

public void checkBill(int billId){
    if(billId == 1) throw new IllegalArgumentException("iae Exception");
    else throw new RuntimeException("re Exception");
}
}

```

当 billId 为 1 时抛出 `IllegalArgumentException`, 否则抛出 `RuntimeException`。运行以下测试代码:

```

String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
SmartSeller seller = (SmartSeller) ctx.getBean("seller");
seller.checkBill(1); ①————— 运行该方法将抛出 IllegalArgumentException

```

将看到以下输出信息:

```

----bindException()----
exception:iae Exception
----bindException()----
Exception in thread "main" java.lang.IllegalArgumentException: iae Exception
...

```

可见, 当 `seller.checkBill(1)` 抛出异常后, 异常增强起作用, 处理完成后, 再向外抛出 `IllegalArgumentException`。如果将①处的代码调整为 `seller.checkBill(2)`, 再次运行代码, 将直接抛出 `RuntimeException`, 异常增强并没有起作用。这是因为 `RuntimeException` 不按类型匹配于 `IllegalArgumentException`, 切点不匹配。

8.7 基于 Schema 配置切面

如果读者的项目不能使用 Java 5.0, 那么就无法使用基于`@AspectJ`注解的切面。但是使用 AspectJ 切点表达式的大门依旧向我们敞开着, 因为 Spring 提供了基于 Schema 配置的方法, 它完全可以替代基于`@AspectJ`注解声明切面的方式。

这是很容易理解的, 因为基于`@AspectJ`注解的切面, 本质上是将切点、增强类型的信息使用注解进行描述, 现在把这两个信息移到 Schema 的 XML 配置文件中, 只是配置信息所在的地方不一样, 表达的信息可以完全相同。XML 和注解, 一个是“金刚”, 一个是“罗汉”, 就像那句俗语“猪往前拱, 鸡往后刨”, 做的是同一件事, 只是方法形式不同而已。

使用基于 Schema 的切面定义后, 切点、增强类型的注解信息从切面类中剥离出来, 原来的切面类也就蜕变为真正意义上的 POJO。

8.7.1 一个简单切面的配置

首先来配置一个基于 Schema 的切面, 它使用了 `aop` 命名空间。为了更准确地了解整个配置文件的全貌, 给出一个结构上完整的切面示例, 如代码清单 8-15 所示。

代码清单 8-15 基于 Schema 配置的切面示例

```

<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">
    <aop:config proxy-target-class="true">
        <aop:aspect ref="adviceMethods"> ① ← 引用④处的adviceMethods
            <aop:before pointcut="target(com.smart.NaiveWaiter) and execution (* greetTo(..))"②
                method="preGreeting"/> ③ ← 增强方法使用adviceMethods Bean中的
                                         preGreeting方法
        </aop:aspect>
    </aop:config>
</beans>

```

① 处：引用④处的 adviceMethods
② 处：声明切点表达式
③ 处：增强方法所在的 Bean
④ 处：增强方法所在的 Bean

使用一个[<aop:aspect>](#)元素标签定义切面，其内部可以定义多个增强。在[<aop:config>](#)元素中可以定义多个切面。在①处，切面引用了 adviceMethods Bean，该 Bean 是增强方法所在的类。通过[<aop:before>](#)声明了一个前置增强，并通过 pointcut 属性定义切点表达式，切点表达式的语法和@AspectJ 中所用的语法完全相同，由于&&在 XML 中使用不便，所以一般用 and 操作符代替之。③处通过 method 属性指定增强的方法，该方法应该是 adviceMethods Bean 中的方法。

[<aop:config>](#)拥有一个 proxy-target-class 属性，当设置为 true 时，表示其中声明的切面均使用 CGLib 动态代理技术；当设置为 false 时，使用 Java 动态代理技术。一个配置文件可以同时定义多个[<aop:config>](#)，不同的[<aop:config>](#)可以采取不同的代理技术。

AdviceMethods 是增强方法所在的类，它是一个普通的 Java 类，没有任何特殊的地方。

```

package com.smart.schema;
public class AdviceMethods{ ① ← 该方法通过配置被
    public void preGreeting() {
        System.out.println("--how are you!--");
    }
}

```

通过以下代码测试这个使用 Schema 配置的切面：

```

String configPath = "com/smart/schema/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
Waiter naughtyWaiter = (Waiter) ctx.getBean("naughtyWaiter");
naiveWaiter.greetTo("John");
naughtyWaiter.greetTo("Tom")

```

输出以下信息：

```
--how are you!-- ①
NaiveWaiter:greet to John...
NaughtyWaiter:greet to Tom...
```

可见，切面被正确地实施到目标 Bean 中。

8.7.2 配置命名切点

在代码清单 8-15 的②处通过 pointcut 属性声明的切点是匿名切点，不能被其他增强或其他切面引用。Spring 提供了命名切点的配置方式，如代码清单 8-16 所示。

代码清单 8-16 命名切点配置

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods" >
        <aop:pointcut id="greetToPointcut" ①           定义切点，该切点的命名
            expression="target(com.smart.NaiveWaiter)"   为 greetToPointcut
            and execution(* greetTo(..))" />
        <aop:before method="preGreeting" pointcut-ref="greetToPointcut"/>②
    </aop:aspect>
</aop:config>
```

在①处，使用[<aop:pointcut>](#)定义了一个切点，并通过 id 属性进行了命名；在②处，通过 pointcut-ref 引用这个命名的切点。和[<aop:before>](#)一样，除引介增强外，其他任何增强类型的元素都拥有 pointcut、pointcut-ref 及 method 这 3 个属性。

[<aop:pointcut>](#)如果位于[<aop:aspect>](#)元素中，则命名切点只能被当前[<aop:aspect>](#)内定义的元素访问到。为了能被整个[<aop:config>](#)元素中定义的所有增强访问，必须在[<aop:config>](#)元素下定义切点。

```
<aop:config proxy-target-class="true">
    <aop:pointcut id="greetToPointcut" ①           当前<aop:config>下的所有
        expression="target(com.smart.NaiveWaiter) and execution(* greetTo(..))" />  增强均可以访问该切点

    <aop:aspect ref="adviceMethods" >
        <aop:before method="preGreeting" pointcut-ref="greetToPointcut" /> ②
    </aop:aspect>
    <aop:aspect ref="adviceMethods" >
        <aop:after method="postGreeting" pointcut-ref="greetToPointcut" /> ③
    </aop:aspect>
</aop:config>
```

在①处定义的命名切点分别被②和③处不同的切面增强所访问。如果在[<aop:config>](#)元素下直接定义[<aop:pointcut>](#)，则必须保证[<aop:pointcut>](#)在[<aop:aspect>](#)之前定义。在[<aop:config>](#)元素下还可以定义[<aop:advisor>](#)（后面介绍），三者在[<aop:config>](#)中的配置有先后顺序的要求：首先是[<aop:pointcut>](#)，然后是[<aop:advisor>](#)，最后是[<aop:aspect>](#)。而在[<aop:aspect>](#)中定义的[<aop:pointcut>](#)则没有先后顺序的要求，可以在任何位置定义。可以通过 spring-aop-4.0.xsd Schema 样式定义文件了解关于 config 元素的配置规则。



实战经验

Schema 样式定义文件比 DTD 样式定义文件具有更强的文档样式定义能力。但由于 Schema 样式定义文件使用基于 XML 的文档样式描述语言定义文档格式，所以样式定义文件本身的内容比较复杂，可读性不高。为了快速理解 Spring 所提供的 Schema 样式定义文件（如 spring-aop-4.0.xsd 等）所描述的文档规则，可以借助一些可视化工具。Altova XMLSpy 就是这样的一款 Schema 可视化工具，图 8-11 就是从 XMLSpy 中导出的。

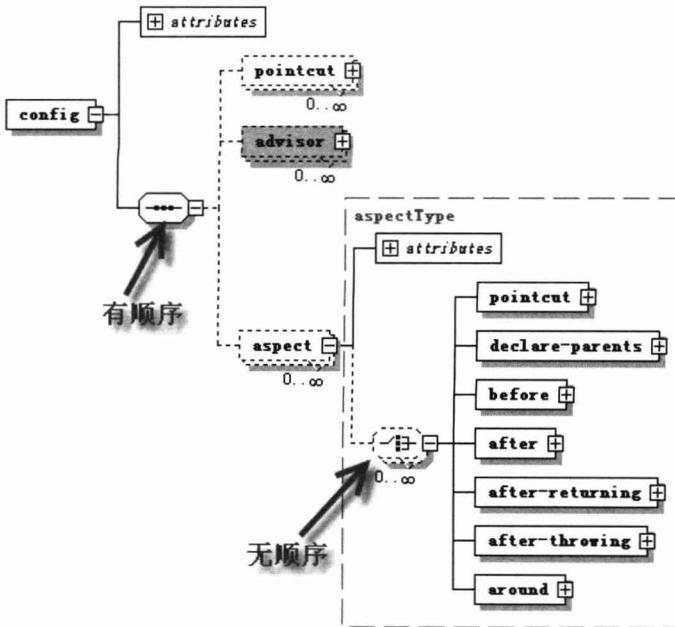


图 8-11 config 元素的 Schema 样式定义

8.7.3 各种增强类型的配置

基于 Schema 定义的切面和基于@AspectJ 定义的切面内容大抵一致，只是在表现形式上存在差异罢了。在上一节中介绍了前置增强，本节来学习如何通过 Schema 配置其余的增强类型。

1. 后置增强

通过<aop:after-returning>配置后置增强。

```

<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods" >
        <aop:after-returning method="afterReturning"
            pointcut="target(com.smart.SmartSeller)" returning= "retVal"/> ①
    </aop:aspect>

```

```
</aop:config>
...

```

returning 属性必须和增强方法的入参名一致。下面是后置增强对应的方法：

```
package com.smart.schema;
public class AdviceMethods {
    public void afterReturning(int retVal){ ① ← 增强方法, retVal 和配置文件中的 returning 属性值相同
        ...
    }
}
```

如果增强方法不希望接收返回值，将配置处的<aop:after-returning>的 returning 属性和增强方法的对应入参去除即可。

2. 环绕增强

通过<aop:around>配置环绕增强。

```
<aop:config proxy-target-class="true">
<aop:aspect ref="adviceMethods" >
    <aop:around method="aroundMethod"
        pointcut="execution(* serveTo(..) and within(com. smart.Waiter))"/>
    </aop:aspect>
</aop:config>
...

```

和前面介绍的一样，环绕增强对应的方法，可以将第一个入参声明为 ProceedingJoinPoint。

```
package com.smart.schema;
import org.aspectj.lang.ProceedingJoinPoint;
public class AdviceMethods {
    public void aroundMethod(ProceedingJoinPoint pjp){ ① ← 环绕增强的方法, pjp 可以访问到环绕增强的连接点信息
        ...
    }
}
```

3. 抛出异常增强

通过<aop:after-throwing>匹配抛出异常的增强。

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods" >
        <aop:after-throwing method="afterThrowingMethod"
            pointcut="target(com.smart.SmartSeller) and execution(* checkBill(..))"
            throwing="iae"/> ① ← 通过 iae 查找增强方法对应名字的入参，进而获取需要增强的连接点的匹配异常类型为
    </aop:aspect>
</aop:config>
...

```

通过 throwing 属性声明需要绑定的异常对象，指定的异常名必须和增强方法对应的入参名一致。

```
package com.smart.schema;
public class AdviceMethods {
    public void afterThrowingMethod(IllegalArgumentException iae){
        ...
    }
}
```

4. Final增强

通过[<aop:after>](#)配置Final增强。

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods" >
        <aop:after method="afterMethod"
            pointcut="execution(* com..*.Waiter.greetTo(..))"/>
    </aop:aspect>
</aop:config>
...
```

对应的Final增强方法如下：

```
package com.smart.schema;
public class AdviceMethods {
    public void afterMethod(){
        ...
    }
}
```

5. 引介增强

通过[<aop:declare-parents>](#)配置引介增强。引介增强和其他类型的增强不同，它没有method、pointcut和pointcut-ref属性。

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods" >
        <aop:declare-parents
            implement-interface="com.smart.Seller" ① ← 要引介实现的接口
            default-impl="com.smart.SmartSeller" ② ← 默认的实现类
            types-matching="com.smart.Waiter+" /> ③ ← 哪些类需要引介
                                                接口的实现
    </aop:aspect>
</aop:config>
...
```

[<aop:declare-parents>](#)通过implement-interface属性声明要实现的接口，通过default-impl属性指定默认的接口实现类，通过types-matching属性以AspectJ切点表达式语法指定哪些Bean需要引介Seller接口的实现。可以通过以下代码查看到NaiveWaiter已经实施了引介增强：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
((Seller)naiveWaiter).sell("Beer", "John"); ① ← 强制类型转换成功
```

需要注意的是，虽然[<aop:declare-parents>](#)没有method属性指定增强方法所在的Bean，但[<aop:aspect ref="adviceMethods" >](#)中的ref属性依然要指定一个增强Bean。

8.7.4 绑定连接点信息

基于Schema配置的增强方法绑定连接点信息和基于@AspectJ绑定连接点信息所使用的方式没有什么区别。

- 第一，所有增强类型对应的方法的第一个入参都可以声明为JoinPoint（环绕增强可以声明为ProceedingJoinPoint）访问连接点信息。

□ 第二，<aop:after-returning>（后置增强）可以通过 returing 属性绑定连接点方法的返回值，<aop:after-throwing>（抛出异常增强）可以通过 throwing 属性绑定连接点方法所抛出的异常。

□ 第三，所有增强类型都可以通过可绑定参数的切点函数绑定连接点方法的入参。

第一、第二种绑定参数的访问已经在上一节进行了介绍，下面通过一个实例来了解第三种绑定参数的方法，如代码清单 8-17 所示。

代码清单 8-17 绑定连接点参数到增强方法

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods">
        <aop:before method="bindParams" ①
            pointcut="target(com.smart.NaiveWaiter) and args(name, num,..)" /> ②
        </aop:before>
    </aop:aspect>
</aop:config>
```

在②处，在切点表达式中通过 args(name,num,..)绑定了连接点的两个参数，对应的增强函数如代码清单 8-18 所示。

代码清单 8-18 AdviceMethods 绑定参数的增强方法

```
package com.smart.schema;
import org.aspectj.lang.ProceedingJoinPoint;
public class AdviceMethods {
    public void bindParams(int num, String name){ //①
        System.out.println("----bindParams()----");
        System.out.println("name:" + name);
        System.out.println("num:" + num);
        System.out.println("----bindParams()----");
    }
}
```

①处的 bindParams(int num, String name)和代码清单 8-17 中的切点函数 args(name,num,..)声明的参数名必须相同。运行以下代码：

```
String configPath = "com/smart/schema/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
((NaiveWaiter)naiveWaiter).smile("John", 2);
```

输出以下信息：

```
----bindParams()----
name:John
num:2
----bindParams()----
NaiveWaiter:smile to John2times...
```

可见，目标类 NaiveWaiter 的连接点方法 smile("John", 2)的入参被正确地绑定到增强方法中。

8.7.5 Advisor 配置

在第 7 章中学习了 Advisor 的知识，它是 Spring 中切面概念的对应物，是切点和增

强的复合体，不过它仅包含一个切点和一个增强。在 AspectJ 中没有对应的等价物，在 aop Schema 配置样式中，可以通过<aop:advisor>配置一个 Advisor。通过 advice-ref 属性引用基于接口定义的增强，通过 pointcut 定义切点表达式，或者通过 pointcut-ref 引用一个命名的切点。来看下面的例子：

```
<aop:config proxy-target-class="true">
    <aop:advisor advice-ref="testAdvice" ①             引用③处的增强
        pointcut="execution(* com..*.Waiter.greetTo(..))"/> ②           切点表达式
    </aop:advisor>
</aop:config>
<bean id="testAdvice"                                前置增强
    class="com.smart.schema.TestBeforeAdvice"/> ③
```

在③处定义了一个前置增强 Bean，而在①处引用这个增强，在②处使用切点表达式配置切点。TestBeforeAdvice 是一个实现了 MethodBeforeAdvice 接口的增强类，如下：

```
package com.smart.schema;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class TestBeforeAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        System.out.println("-----TestBeforeAdvice-----");
        System.out.println("clientName:"+args[0]);
        System.out.println("-----TestBeforeAdvice-----");
    }
}
```

8.8 混合切面类型

现在我们已经掌握了 4 种定义切面的方式：

- (1) 基于@AspectJ注解的方式。
- (2) 基于<aop:aspect>的方式。
- (3) 基于<aop:advisor>的方式。
- (4) 基于 Advisor 类的方式。

作为开发者，可能会觉得 Spring 在同一个问题上提供了太多的选择，在选择实现方案时反倒让人陷入了困境。其实 Spring 并不是在做一件吃力不讨好的事情，开发人员完全可以根据项目的具体情况做出选择：如果项目采用 Java 5.0，则可以优先考虑使用 @AspectJ；如果项目只能使用低版本的 JDK，则可以考虑使用<aop:aspect>；如果正在升级一个基于低版本 Spring AOP 开发的项目，则可以考虑使用<aop:advisor>复用已经存在的 Advice 类；如果项目只能使用低版本的 Spring，那么就只能使用 Advisor 了。此外，值得注意的是，一些切面只能使用基于 API 的 Advisor 方式进行构建，如基于 ControlFlowPointcut 的流程切面。

8.8.1 混合使用各种切面类型

Spring 虽然提供了 4 种定义切面的方式，但其底层的实现技术却是一样的，那就是基于 CGLib 和 JDK 动态代理，所以在同一个 Spring 项目中可以混合使用 Spring 所提供的各种切面定义方式，如代码清单 8-19 所示。

代码清单 8-19 各种切面类型混合使用

```

<bean id="controlFlowAdvisor" ① ← 使用 Advisor API 方式实现的流程控制切面
    class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut">
        <bean class="org.springframework.aop.support.ControlFlowPointcut">
            <constructor-arg type="java.lang.Class"
                value="com.smart.advisor.WaiterDelegate"/>
            <constructor-arg type="java.lang.String" value="service"/>
        </bean>
    </property>
    <property name="advice">
        <bean class="com.smart.advisor.GreetingBeforeAdvice"/>
    </property>
</bean>                                使用 @AspectJ 方式定义的切面

<aop:aspectj-autoproxy />② ←

<bean class="com.smart.aspectj.example.PreGreetingAspect" />
                                使用基于 Schema 配置方式定义的切面

<aop:config proxy-target-class="true"> ③ ←
    <aop:advisor advice-ref="testAdvice" pointcut="execution(* com..*.Waiter.
greetTo(..))"/>④ ← 使用<aop:advisor>配置方式定义的切面

    <aop:aspect ref="adviceMethods"> ⑤ ← 使用<aop:aspect>配置方式定义的切面
        <aop:before pointcut="target(com.smart.NaiveWaiter) and execution (* greetTo(..))"
            method="preGreeting"/>
    </aop:aspect>
</aop:config>
<bean id="adviceMethods" class="com.smart.schema.AdviceMethods" />⑥ ← POJO 的增强类
<bean id="testAdvice" class="com.smart.schema.TestBeforeAdvice"/>⑦ ← 基于特定增强
接口的增强类

```

虽然 Spring 可以混合使用各种切面类型达到统一的效果，但在一般情况下并不会在一个项目中同时使用。毕竟项目开发不是时装表演，也不是多军种联合演习，我们应该尽量根据项目的实际需要采用单一的实现方式，以保证技术的单一性。

8.8.2 各种切面类型总结

在学习完 Spring 的 4 种切面定义方式后，有必要对这 4 种实现方式进行比较，它们的本质是相同的，都是定义切点和增强，不同的只是在表现形式上，如表 8-2 所示。

表8-2 切面不同定义方式具体实现比较

	@AspectJ	<aop:aspect>	Advisor	<aop:advisor>
增强类型	前置增强 @Before	<aop:before>	MethodBeforeAdvice	同 Advisor
	后置增强 @AfterReturning	<aop:after-returning>	AfterReturningAdvice	同 Advisor
	环绕增强 @Around	<aop:around>	MethodInterceptor	同 Advisor
	抛出异常增强 @AfterThrowing	<aop:after-throwing>	ThrowsAdvice	同 Advisor
	final增强 @After	<aop:after>	无对应接口	同 Advisor
	引介增强 @DeclareParents	<aop:declare-parents>	IntroductionInterceptor	同 Advisor
切点定义	支持 AspectJ 切点表达式语法, 可以通过@Pointcut注解定义命名切点	支持 AspectJ 切点表达式语法, 可以通过<aop:pointcut>定义命名切点	直接通过基于 Pointcut 的实现类定义切点	基本上和<aop:aspect>相同, 不过切点函数不能绑定参数
连接点方法入参绑定	(1) 使用 JoinPoint、Proceeding JoinPoint 连接点对象; (2) 使用切点函数指定参数名绑定	同@AspectJ <aop:after-returning>	通过增强接口方法入参绑定	同 Advisor
连接点方法返回值或抛出异常绑定	(1) 在后置增强中, 使用 @AfterReturning 的 returning 成员绑定方法返回值; (2) 在抛出异常增强中, 使用@AfterThrowing 的 throwing 成员绑定方法抛出的异常	(1) 在后置增强中, 使用 <aop:after-returning> 的 returning 属性绑定方法返回值; (2) 在抛出异常增强中, 使用 <aop:after-throwing> 的 throwing 属性绑定方法抛出的异常	通过增强接口方法入参绑定	同 Advisor

从表8-2中可以看出, <aop:advisor>其实是<aop:aspect>和 Advisor的“混血儿”, 它的切点表示方式和<aop:aspect>相同, 增强定义方式则和 Advisor相同。连接点方法入参的绑定方式和 Advisor一样, 通过增强接口方法入参进行调用, 所以<aop:advisor>在切点表达式中不能使用切点函数绑定连接点方法入参, 否则会产生错误。

在内部, Spring 使用 AspectJExpressionPointcut 为 @AspectJ、<aop:aspect> 及 <aop:advisor> 提供具体的切点实现。

8.9 其他

8.9.1 JVM Class 文件字节码转换基础知识

到目前为止, 我们所接触到的 AOP 切面织入都是在运行期通过 JDK 或 CGLib 动态代理的方式实现的。我们知道, 除了运行期织入切面的方式外, 还可以在类加载期通过字节码编辑技术将切面织入目标类中, 这种织入方式称为 LTW (Load Time Weaving)。

AspectJ LTW 使用 Java 5.0 所提供的代理功能（agent）完成加载期切面织入工作。JDK 的代理功能能够让代理器访问到 JVM 的底层部件，借此向 JVM 注册类文件转换器，在类加载时对类文件的字节码进行转换。AspectJ LTW 由于基于 JDK 动态代理技术工作，而 JDK 动态代理的作用范围是整个 JVM，所以这种工作方式比较粗放，对于单一 JVM 多个应用的情况尤其不适合。

Spring 为 LTW 的过程提供了细粒度的控制，它支持在单个 ClassLoader 范围内实施类文件转换，且配置更为简单。

在进行 Spring LTW 的学习之前，先来了解一下 Java 5.0 中引入的 Instrument 的相关知识，为后面的学习打好基础。

java.lang.instrument 包的工作原理

Java 5.0 新增了一个 java.lang.instrument 包，该包中有两个能对 JVM 底层组件进行访问的类。具体地说，就是通过 JVM 的 -javaagent 代理参数在启动时获取 JVM 内部组件的引用，以便在后续流程中使用。借助 JDK 动态代理，可以在 JVM 启动时装配并应用 ClassTransformer，对类字节码进行转换，实现 AOP 的功能。

java.lang.instrument 包中定义了两个重要的接口。

- ClassFileTransformer：Class 文件转换器接口，该接口有一个唯一的方法，如下：

```
byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
ProtectionDomain protectionDomain, byte[] classfileBuffer)
```

该接口对 Class 文件的字节码进行转换，classfileBuffer 是类文件对应的字节码数组，返回的 byte[] 为转换后的字节码。如果返回 null，则表示不进行字节码处理（并非将类的字节码数据置空）。

- Instrumentation：代表 JVM 内部的一个构件。这个术语不好翻译，它有“仪表、仪器、设备”等意思，我们不妨将其称为“组件”。

可以通过该接口的方法向 JVM 的内部“组件”注册一些 ClassFileTransformer，注册转换器的接口方法为 void addTransformer(ClassFileTransformer transformer)。

当 ClassFileTransformer 实例注册到 JVM 中后，JVM 在加载 Class 文件时，会先调用这个 ClassFileTransformer 的 transform() 方法对 Class 文件的字节码进行转换。如果向 JVM 中注册多个 ClassFileTransformer，它们将按注册的顺序组成链式的调用。这样 ClassFileTransformer 的实现者就可以从 JVM 层面截获所有类的字节码，并引入希望添加的逻辑，如让每个类拥有性能监视的能力、织入特殊用途的增强代码等。

图 8-12 描述了拥有多个转换器的 JVM 从加载类到最终生成对应的类字节码的过程。

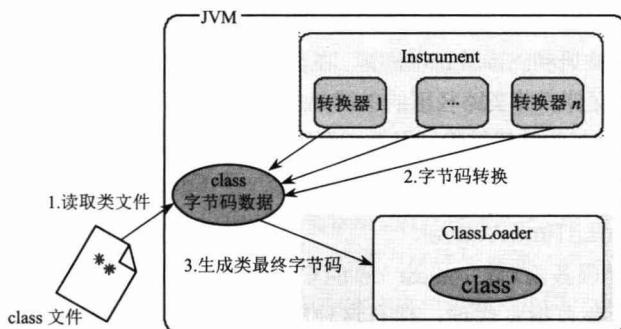


图 8-12 带转换器的 JVM 加载 Class 文件的处理过程

**提示**

目前已经有很多字节码处理的开源项目，ObjectWeb 的 ASM、Apache 的 BCEL 及 SourceForge 和 SERP 是其中优秀的代表者。TopLink JPA 就使用 ASM 编写它的 ClassTransformer。

8.9.2 使用 LTW 织入切面

Spring 的 LTW 仅支持 AspectJ 定义的切面，既可以是直接采用 AspectJ 语法定义的切面，也可以是采用基于@AspectJ 注解，通过 Java 类定义的切面。Spring LTW 直接采用了与 AspectJ LTW 相同的基础结构，即它利用类路径下的 META-INF/aop.xml 配置文件找到切面定义及切面所要实施的候选目标类的信息，通过 LoadTimeWeaver 在 ClassLoader 加载类文件时将切面织入目标类中。工作原理如图 8-13 所示。

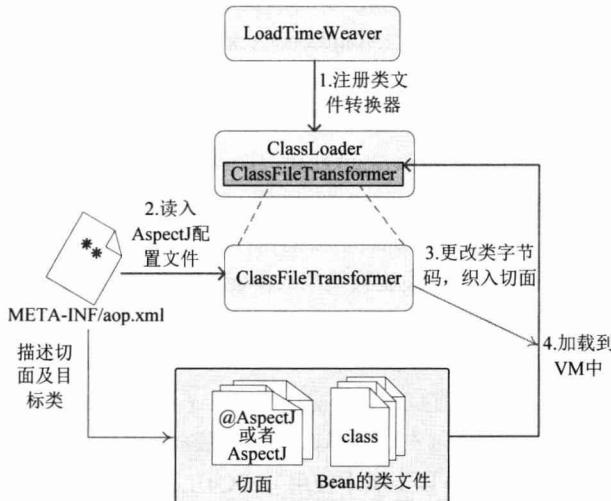


图 8-13 Spring LTW 工作原理

Spring 利用特定 Web 容器的 ClassLoader，通过 LoadTimeWeaver 将 Spring 提供的 ClassFileTransformer 注册到 ClassLoader 中。在类加载期，注册的 ClassFileTransformer 读取 AspectJ 的配置文件，即类路径下的 META-INF/aop.xml 文件，获取切面，对加载到 VM 中的 Bean 类进行字节码转换，织入切面。Spring 容器初始化 Bean 实例时，采用的 Bean 类就是已经被织入了切面的类。下面来了解一下 Spring 的 LoadTimeWeaver。

1. Spring 的 LoadTimeWeaver

大多数 Web 应用服务器（除 Tomcat 外）的 ClassLoader 都支持直接访问 Instrument，无须通过 javaagent 参数指定代理，拥有这种能力的 ClassLoader 称为“组件使能”（instrumentation-capable）。通过“组件使能”功能，可以非常方便地访问 ClassLoader 的 Instrument。Spring 利用 Web 应用服务器类加载器的这个特性，为它们分别提供了专门的 LoadTimeWeaver，以便向特定的 ClassLoader 注册 ClassFileTransformer，对类进行字节码转换，实施切面的织入。

Spring 的 org.springframework.instrument.classloading.LoadTimeWeaver 接口规定了类加载期织入器的高层协议。该接口有 3 个方法。

- void addTransformer(ClassFileTransformer transformer): 添加一个 ClassFileTransformer 到加载期织入器中。
- ClassLoader getInstrumentableClassLoader(): 我们知道，JVM 拥有 Instrumentation 组件，但这是 JVM 级别的。Spring 对 ClassLoader 进行扩展，让它也具有 Instrumentation 组件，以便只对 ClassLoader 中的类应用 ClassFileTransformer。
- ClassLoader getThrowawayClassLoader(): 返回一个“丢弃”的 ClassLoader，目的是使 Instrumentation 的作用范围仅局限在本 ClassLoader 中，而不影响父 ClassLoader。Spring 为 LoadTimeWeaver 提供了多个实现类。
- InstrumentationLoadTimeWeaver: 该装载期织入器使用-javaagent JVM 启动参数注册转换器，该类必须和 org.springframework.instrument.InstrumentationSavingAgent 结合使用。InstrumentationSavingAgent 是代理类，它获取 JVM 的 Instrumentation 后，以静态变量的方式保存这个引用，这样其他类就可以通过 getInstrumentation() 方法从 InstrumentationSavingAgent 中获取 JVM 的 Instrumentation。InstrumentationLoadTimeWeaver 即利用 InstrumentationSavingAgent 持有的 Instrumentation 引用完成添加转换器的操作。
- SimpleLoadTimeWeaver: 该装载期织入器能为当前的 ClassLoader 创建一个相应的 SimpleInstrumentableClassLoader（简单组件使能的 ClassLoader 实现类），一般在测试或 IDE 环境下使用。
- 特定 Web 服务器的织入器: JBossLoadTimeWeaver（在 JBoss AS 5.0 以上版本使用）、GlassFishLoadTimeWeaver（在 GlassFish 3.0 以下版本使用）、WebLogicLoadTimeWeaver（在 BEA WebLogic 10.0 以上版本使用）、OC4JLoadTimeWeaver（在 Oracle OC4J 10.1.3.1 以上版本使用）。

- ReflectiveLoadTimeWeaver: 有一些应用服务器不是组件使能的 ClassLoader, 或者虽然是组件使能的, 但却无法获取应用服务器的类加载器实例, 用户只能以 ClassLoader 类型获取这些类加载器的句柄。在这种情况下, Spring 也没有办法将转换器添加到 ClassLoader 中。这时, 只得对目标的 ClassLoader 进行改造, 让它实现一些特殊的方法, 并由 ReflectiveLoadTimeWeaver 在运行期通过反射的机制调用接口方法, 以便注册 ClassFileTransformer。可以通过扩展需要改造的 ClassLoader 类, 并实现以下两个特殊方法达到目的。
 - public void addTransformer(java.lang.instrument.ClassFileTransformer): 向 ClassLoader 中注册转换器。
 - public ClassLoader getThrowawayClassLoader(): 返回需要丢弃的 ClassLoader。

Spring 只需在配置文件中添加一行配置就可以启用 LoadTimeWeaver, 如下:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <!--①启用Spring装载期织入功能 -->
    <context:load-time-weaver/>
</beans>
```

通过①处的配置, 向 Spring 容器中添加一个 LoadTimeWeaver, 它负责向运行期的 ClassLoader 注册多个 ClassFileTransformer, 以便实施 LTW 的功能。

2. 在 Tomcat 下的配置

Tomcat 服务器的 ClassLoader 不是组件使能的 ClassLoader, Spring 专门为 Tomcat 提供了一个 TomcatInstrumentableClassLoader, 它扩展于 Tomcat 服务器的 org.apache.catalina.loader.WebappClassLoader, 并实现了 LoadTimeWeaver 要求的两个特殊方法。

可以按照以下步骤使 Tomcat Web 应用启动 Spring 的 LoadTimeWeaver。

- (1) 在 Spring 配置文件中定义<context:load-time-weaver/>。
- (2) 将 org.springframework.instrument.tomcat-{version}.jar 复制到<TOMCAT_HOME>/lib 下, 该 JAR 包中包含了 TomcatInstrumentableClassLoader 类。
- (3) 编写以下配置片段, 让 Tomcat 服务器使用 TomcatInstrumentableClassLoader 作为其 ClassLoader。

```
<Context path="/myWebApp" docBase="/my/webApp/location">
  <Loader loaderClass="org.springframework.instrument.classloading.tomcat.
    TomcatInstrumentableClassLoader"
         useSystemClassLoaderAsParent="false"/>
</Context>
```

如何让以上配置片段生效呢? 对于 Tomcat 6.x 和 Tomcat 7.x 来说, 可以使用以下 4 种方式。

(1) 打开<TOMCAT_HOME>/conf/server.xml 文件，将以上配置片段添加到 server.xml 文件的相应位置上。

(2) 将以上配置片段的内容保存为 context.xml 文件，并将其复制到<TOMCAT_HOME>/conf/context.xml 中。

(3) 将以上配置片段的内容保存为<webapp_name>.xml 文件，并将其复制到<TOMCAT_HOME>/conf/Catalina/localhost/<webapp_name>.xml 中。

(4) 将以上配置片段的内容保存为 context.xml 文件，并将其和 Web 应用程序一起打包在 WAR 包中，放在 WAR 包的 META-INF 目录下：<webapp_name>.war/META-INF/context.xml。

前两种配置方式会产生全局，也就是说，Tomcat 应用中所有 Web 应用的类加载器都会调整为 TomcatInstrumentableClassLoader；而后两种配置方式只会对相应的 Web 应用产生影响，其他的 Web 应用依旧使用 Tomcat 默认的 WebappClassLoader。从部署的简易性到范围的影响性，最后一种配置方式是最佳选择。

3. 在其他 Web 应用服务器下的配置

前面说过，有 4 种类型的 Web 服务器的类加载器支持组件使能功能。因此，它们既无须使用 javaagent 参数，也无须更改任何 Web 服务器的配置文件，只需在 Spring 的配置文件中配置<context:load-time-weaver/>，就可以使用 Spring 的 LTW。现将这 4 种 Web 服务器及其版本列出。

- BEA WebLogic（10.0 以上版本）。
- Oracle Containers for Java EE（OC4J 10.1.3.1 以上版本）。
- Resin（3.1 以上版本）。
- JBoss（5.x 以上版本）。

8.10 小结

在本章中，首先介绍了 Java 5.0 的注解知识，它是学习@AspectJ 的基础。使用 @AspectJ 定义切面比基于接口定义切面更加直观、更加简洁，成为 Spring 所推荐的切面定义方式。

掌握切点表达式语法和切点函数是学习@AspectJ 的重心，我们分别对 9 个切点函数进行了详细的讲述。切点表达式非常灵活，拥有强大的切点表达能力，读者可以使用通配符、切点函数及切点运算符定义切点。

如果项目因某种原因无法使用 Java 5.0，则可以采用基于 Schema 配置的方式继续使用 AspectJ 的切点表达式和增强定义，基于 Schema 的配置采用<aop:aspect>描述@Aspect 类所描述的相同信息，只是换了一种方法而已。此外，还可以通过<aop:advisor>复用旧系统已有的 Advice，并配合使用 AspectJ 的切点表达式。

在切点表达式中，大多数切点函数都可以绑定连接点方法的入参，以便增强方法访问连接点信息。此外，用户也可以简单地将增强方法的第一个入参定义为 JoinPoint 访问连接点的上下文。

Spring AOP 虽然提供了 4 种切面定义方式，但其底层实现却是相同的，也就是说，表象不同，本质归一。所以如果需要，则可以放心地混合使用这些不同的定义方式。

Spring 还支持 LTW 的功能，允许通过 AspectJ 定义切面，在类加载期通过类文件转换器织入切面。

第 9 章

Spring SpEL

Spring 动态表达式语言（简称 SpEL）是一个支持运行时查询和操作对象图的强大动态语言。其语法类似于 EL 表达式，具有诸如显式方法调用和基本字符串模板函数等特性。本章对 JVM 动态语言进行简要概述，并对 SpEL 动态语言特性及其表达式的用法进行详细介绍，通过实例逐步揭开 SpEL 表达式的层层外衣。通过本章的学习，读者不仅可以运用 SpEL 表达式解决实际项目中的一些通用性表达式需求，还可以应用 SpEL 提供的扩展点来解决一些高阶问题。

本章主要内容：

- ◆ JVM 动态语言
- ◆ SpEL 表达式概述
- ◆ SpEL 核心接口
- ◆ SpEL 基础表达式
- ◆ 在 Spring 中使用 SpEL

本章亮点：

- ◆ 对 Java 动态表达式发展的总结
- ◆ 对 SpEL 各种用法进行讲解

9.1 JVM 动态语言

Java 是一门强类型的静态语言，所有代码在运行之前都必须进行严格的类型检查并编译成 JVM 字节码；因此虽然在安全、性能方面得到了保障，但牺牲了灵活性。这个特征就决定了 Java 在语言层面无法直接进行表达式语句的动态解析。而动态语言恰恰相反，其显著的特点是在程序运行时可以改变程序结构或变量类型。典型的动态语言有

Ruby、Python、JavaScript、Perl 等。这些动态语言能够被广泛应用于许多领域，得益于其动态、简单、灵活等特性。因为它们无须编译，即可被解释执行。它们可以在运行时动态改变表达式语句，非常适合编写复杂的动态表达式。

Java 在实现复杂业务系统、大型商业系统、分布式系统及中间件等方面有着非常强的优势。但在开发这些系统的过程中，有时需要引用动态语言的一些特性，以弥补其在动态性方面的不足。特别是在解决一些商业系统中动态规则的解析需求，如积分规则、各类套餐计费、活动促销等规则时，在 Java 的早期版本中，开发人员一般会使用 Rhino、BeanShell、MVEL 等类库实现。

为了简化在 Java 中使用动态脚本语言的难度，Java 6.0 开始提供对 JSR-223 规范的全面支持。JSR-223 中规范了在 Java 虚拟机上运行的动态脚本语言与 Java 程序之间的交互方式，并在 Java 6.0 中内置集成了 Mozilla Rhino 的 JavaScript 解析引擎，因此可以很方便地在 Java 中调用 JavaScript 编写的动态脚本，并通过 JavaScript 解释执行的特性来编写相关动态业务逻辑。

下例在 Java 中使用 Rhino 快速实现一个动态求和函数 sum，只需在脚本引擎中注册一个标准的 JavaScript 函数，就可以在 Java 应用上下文中调用注册的 JavaScript 函数，如代码清单 9-1 所示。

代码清单 9-1 ScriptSample：脚本实现动态函数

```
package com.smart.js;
import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class ScriptSample {

    public static void main(String[] args) throws Exception{
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("JavaScript");
        String scriptText = "function sum(a,b) { return a+b; } ";
        engine.eval(scriptText);
        Invocable invocable = (Invocable) engine;
        Object result = invocable.invokeFunction("sum", 100, 99);
        System.out.println(result); //打印199
    }
}
```

上例只用几行代码就实现了一个动态求和函数。首先使用 ScriptEngineManager 创建一个脚本引擎管理器，并通过其创建脚本引擎，调用 ScriptEngine#eval()方法注册 JavaScript 求和函数脚本。接下来就可以使用 Invocable#invokeFunction()方法调用注册的 sum 函数，第一个参数就是要调用的自定义函数名 sum，第二个参数开始对应自定义函数名 sum 的参数列表。

虽然目前在 JVM 中支持很多脚本语言（如 JavaScript、JRuby、Jython 等），但在使用的时候，还是需要对其进行相应的封装。对于仅仅需要一些简单的表达式需求的场景，使用脚本语言显得有些“笨重”，这也就是下文要重点介绍 SpEL 表达式的原因。

9.2 SpEL 表达式概述

Spring 动态语言（简称 SpEL）是一个支持运行时查询和操作对象图的强大的动态语言。其语法类似于 EL 表达式，具有诸如显式方法调用和基本字符串模板函数等特性。

同其他的 Java 动态语言相比（如国外的 OGNL、MVEL 和 JBoss EL，国内的 Aviator、IKEExpression 和 FastEL 等），SpEL 不但提供上述表达式的类似功能，而且更加简洁、灵活。加之出自 Spring 社区之手，与 Spring 框架及其子项目的结合显得更加顺畅自然。

SpEL 表达式的语言特性都是经过 Spring 框架及其子项目的需求一步步提炼而成的。在使用方面，IDE 工具可提供很好的代码自动补全功能。在集成方面，SpEL 抽象了一个通用表达式操作 API，因此可以很好地与其他动态语言进行集成。

SpEL 作为 Spring 家族中表达式求值的基础，它不直接依赖于 Spring 框架，可独立使用。只是在基于 Spring 的框架中使用 SpEL 更加便捷，一般情况下，无须手工调用 SpEL 提供的 API。因为 Spring 框架已经提供了许多直接使用 SpEL 表达式的方法，并且在框架层面屏蔽了表达式的运行设施创建过程，直接使用即可。例如，在 Bean 配置定义中，可以直接通过 “#{ }” 编写 SpEL 表达式。

为了更好地学习 SpEL 动态语言，本章中的示例将 SpEL 作为一个独立的动态语言来使用，通过手工调用 SpEL API 来讲解其各种表达式的用法。下面先通过一个简单的示例来认识一下 SpEL。在使用 SpEL 之前，需要在 pom.xml 文件中添加 spring-expression 模块依赖，如代码清单 9-2 所示。

代码清单 9-2 在 pom.xml 文件中添加表达式模块依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.smart</groupId>
    <artifactId>chapter9</artifactId>
    <version>1.0</version>
    <name>Spring4.x第九章实例</name>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>
</project>
```

SpEL 被设计成一个可独立运行的模块，可以脱离 Spring 容器直接运行，因此只需引入 SpEL 的模块 spring-expression 即可，无须引入 Spring 框架的其他模块。接下来，在代码中就可以使用 SpEL 提供表达式解析类，如代码清单 9-3 所示。

代码清单 9-3 SpelHello示例

```

package com.smart.spel;
import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpELExpressionParser;
public class SpelHello {
    public static void main(String[] args) {
        ExpressionParser parser = new SpELExpressionParser();
        Expression exp = parser.parseExpression("'Hello'+' World'");
        String message = (String) exp.getValue();
        System.out.println(message);
    }
}

```

在 SpEL 中，使用表达式非常简单，只需创建一个 SpELExpressionParser 实例即可。也就是首先创建一个 SpEL 表达式解析器，然后调用 ExpressionParser#parseExpression() 方法对表达式进行解析。其中，单引号表示 String 类型，如示例中的'Hello'、'World'。



轻松一刻

山东话与河南话的发音大致相同，但也有小小的差别。比如山东人称馒头为干粮，河南人称馒头为馍，发音与山东话的“么”相似。一个河南人在山东上大学，去食堂打饭，向大师傅说：“来个馍。”大师傅听了很是茫然，问：“你要么？”河南人答道：“对呀，我要馍。”大师傅听后就更糊涂了，这怎么来打饭的还不知道自己要什么的，就问道：“你到底要什么啊？”河南人有点生气了，心想：你都知道我要什么了怎么还问啊。于是不耐烦地说：“你到底有没有馍？我就是要个馍啊！！”

9.3 SpEL 核心接口

上一节介绍了 SpEL 的简单用法，接下来逐步讲解 SpEL 的核心接口。SpEL 的所有类与接口都定义在 org.springframework.expression 包及其子包，以及 spel.support 中。

ExpressionParser 接口用来解析表达式字符串，表达式字符串是一个用单引号标注或者用转义的双引号标注的字符串。Expression 接口用来计算表达式字符串的值，当调用 ExpressionParser#parseExpression() 和 Expression#getValue() 方法时可能抛出解析异常 ParseException 和求值异常 EvaluationException。因此，在手工调用 API 时，需要处理这些可能的异常。

SpEL 支持一系列功能特性，如方法调用、属性调用及构造器调用等。下例是调用字符串类型的 concat()方法：

```
ExpressionParser parser = new SpELExpressionParser();
```

```
Expression exp=parser.parseExpression("'HelloWorld'.concat('!'))";
String message=(String)exp.getValue(); // message的值为“Hello World!”
```

在上述代码中, Expression#getValue()方法返回的是一个 Object 对象, 需要进行显式类型转换, 使用起来感觉有些不够优雅。可以使用泛型方法 public <T> T getValue(Class<T> desiredResultType), 这个方法取值的时候无须进行显式类型转换。但需要注意的是, 如果返回的类型不能被转换为泛型 T 或者被已注册的类型转换器所转换时, 则会抛出求值 EvaluationException。

```
String message= exp.getValue(String.class);
```

在 SpEL 中更常见的用途是针对特定实例对象的属性进行求值。在下面的例子中, 我们通过表达式获取 User 实例的 userName 属性。

```
User user = new User();
user.setUserName("tom");
user.setCredits(100);
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = new StandardEvaluationContext(user);
String username = (String)parser.parseExpression("userName").getValue(context);
```

在创建 StandardEvaluationContext 实例时, 指定一个根对象作为求值目标对象, 这样在求值表达式中就可以引用根对象属性。在求值内部可以使用反射机制从注册对象中获取相应的属性值。

在单独使用 SpEL 时, 需要创建一个 ExpressionParser 解析器, 并提供一个 EvaluationContext 求值上下文。在普通的基于 Spring 的应用开发中, 这些 API 一般很少会涉及, 仅需编写 SpEL 表达式字符串即可。例如, 在 Bean 定义的时候使用 SpEL 表达式, 只需写好相应的表达式字符串即可, 至于解析器、求值上下文、根对象和其他预定义变量等基础设施, Spring 都会创建。

9.3.1 EvaluationContext 接口

EvaluationContext 接口提供了属性、方法、字段解析器及类型转换器。默认实现类 StandardEvaluationContext 的内部使用反射机制来操作对象。为了提高性能, 在其内部会对已获取的 Method、Field 和 Constructor 实例进行缓存。

StandardEvaluationContext 类可以通过构造函数传递求值根对象或通过 setRootObject() 方法设置求值根对象, 通过 setVariable()方法设置相关变量, 通过 registerFunction()方法注册自定义函数。还可以向 StandardEvaluationContext 类注册自定义构造函数转换器 ConstructorResolvers、方法转换器 MethodResolvers 及属性转换器 PropertyAccessors 来解析 SpEL 表达式。

默认 SpEL 表达式使用的类型转换器是引用 Spring 核心包中的 ConversionService, ConversionService 转换服务会自动根据源类型与目标类型选择合适的转换器。Spring 内置了一系列常用的类型转换器, 如 StringToBooleanConverter、NumberToCharacterConverter 等。如果内置的类型转换器未能满足需求, 则可以编写并注册自定义类型转换器。当处

理表达式中涉及泛型时，SpEL会尝试将当前值转换为对应的目标类型。例如，在表达式中通过 setValue()方法来设置 List 类型的元素时，如果目标类型是 List<Boolean>，那么 SpEL 会自动尝试将当前设置值转换为布尔类型，如下：

```
class Simple{
    public List<Boolean> booleanList= new ArrayList<Boolean>();
}
Simple simple= new Simple();
simple.booleanList.add(true);

//创建求值上下文
StandardEvaluationContext simpleContext= new StandardEvaluationContext(simple);

//自动将 "false" 转换为布尔类型
parser.parseExpression("booleanList[0]").setValue(simpleContext, "false");

//b将被设置为false
Boolean b=simple.booleanList.get(0);
```

在上面的示例中，表达式 “booleanList[0]” 是一个 List<Boolean> 类型。将一个字符串类型的 “false” 值设置给 booleanList 集合的第一个元素，由于目标类型是 Boolean，因而在 SpEL 内部会自动将原类型为 “false”的字符串值转换为目标布尔类型。如果将 “false” 改为 “false1”，再次运行的时候，则会报一个转换异常 ConversionFailedException，因为 “false1” 字符串无法转换为布尔类型。

9.3.2 SpEL 编译器

在默认情况下，SpEL 表达式只有在求值时才进行表达式计算，所以表达式可以在运行时进行动态修改。但对于同一个表达式而言，如果每次取值都要进行动态解析，则势必影响表达式的执行效率。对于一些调用频率不高的场景，对性能的影响不是很明显；反之，如果同一个表达式重复调用比较频繁，那么可能会对性能产生较大的影响。

SpelCompiler 编译器就是为了解决这个问题而诞生的，它可以将表达式直接编译成字节码，从而避免每次调用时进行语法解析所产生的时间消耗，有效提高执行效率。由于已经将表达式编译成字节码，如果在后续运行时表达式发生变化，则必须重新编译。因此，SpelCompiler 适用于表达式不经常发生变动且重复调用频率较高的场景。下面是基于 SpEL 编码的示例，如代码清单 9-4 所示。

代码清单 9-4 CompilerSample 编译表达式示例

```
package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.spel.SpelCompilerMode;
import org.springframework.expression.spel.SpelParserConfiguration;
import org.springframework.expression.spel.standard.SpelExpression;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
```

```

public class CompilerSample {

    public static void main(String[] args) {
        User user = new User();
        //①创建解析配置
        SpelParserConfiguration configuration = new SpelParserConfiguration(
                SpelCompilerMode.IMMEDIATE,
                CompilerSample.class.getClassLoader());
        //②创建解析器
        SpelExpressionParser parser = new SpelExpressionParser(configuration);

        //③创建取值上下文
        EvaluationContext context = new StandardEvaluationContext(user);

        //④表达式
        String expression = "isVipMember('tom') && isVipMember('jony')";

        //⑤解析表达式
        SpelExpression spelExpression = parser.parseRaw(expression);

        //⑥通过表达式求值
        System.out.println(spelExpression.getValue(context)); //第一次调用
        System.out.println(spelExpression.getValue(context)); //第二次调用
    }
}

```

在 SpEL 中启用编译模式，需要创建一个 SpelParserConfiguration 配置解析类，并指定编译模式和类加载器。其中，编译模式 SpEL 提供了一个枚举类型 SpelCompilerMode。该枚举共有 3 个值，分别是 SpelCompilerMode.OFF、SpelCompilerMode.MIXED 和 SpelCompilerMode. IMMEDIATE。

编译模式默认采用 SpelCompilerMode.OFF 值，表示不启用编译。想要全局开启，则可以通过配置类路径下的 spring.properties 配置文件，然后 SpelParserConfiguration 会在类加载时读取 spring.expression.compiler.mode 属性来进行配置。

SpelCompilerMode.MIXED 表示混合模式，表示在解释型和编译型之间进行转换；前面几次执行表达式取值采用解释型处理，直到达到 SpEL 的一个阈值（100）之后，才启用编译处理，也就是对表达式进行编译。

SpelCompilerMode. IMMEDIATE 表示立即启用编译。实际上，在 SpEL 内部不会立即启用编译，而是在第二次执行表达式取值时才会启用编译。

在②处通过配置解析对象创建解析器；在③处通过根对象实例 user 创建取值上下文；在④处编写一个求值表达式 “isVipMember('tom') && isVipMember('jony')”，其中 isVipMember 是 User 对象中声明的方法，根据用户名判断是否是 VIP 会员，“&&” 表示逻辑与操作；在⑤处通过调用 SpelExpressionParser#parseRaw()方法创建 SpelExpression 对象；在⑥处调用 Expression#getValue()方法对表达式求值。

9.4 SpEL 基础表达式

9.4.1 文本字符解析

文本表达式支持字符串、日期、数字（正数、实数及十六进制数）、布尔类型及 null，其中字符串需使用单引号或反斜杠+双引号包含起来，如“'Hello World'”、“\"Hello World\\\"”，而单引号字符可使用 “\\" 表示。来看一个具体的示例，如代码清单 9-5 所示。

代码清单 9-5 LiteralExprSample 文本解析

```
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.*;

public class LiteralExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        //①解析字符串
        String helloWorld = (String) parser.parseExpression("\\"Hello World\\\"").getValue();

        //②解析双精度浮点型
        double doubleNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

        //③解析整型
        int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

        //④解析整型布尔型
        boolean trueValue = (Boolean) parser.parseExpression("true").getValue();

        //⑤解析空值
        Object nullValue = parser.parseExpression("null").getValue();
        ...
    }
}
```

本示例演示了如何使用 SpEL 表达式将文本字面值解析为相应的数据类型。数据类型支持负数、小数、科学计数法及八进制、十六进制数等。默认情况下采用 Double.parseDouble() 方法进行数值转换。

9.4.2 对象属性解析

在 SpEL 中，可使用类似 “xxx.yyy.zzz” 的对象属性路径轻松地访问对象属性值，如代码清单 9-6 所示。

代码清单 9-6 PropertyExprSample 属性解析

```

package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.Date;

public class PropertyExprSample {
    public static void main(String[] args) {

        //①构造一个对象
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100); //设置积分
        user.setPlaceOfBirth(new PlaceOfBirth("中国", "厦门")); //设置出生地

        //②构造SpEL解析上下文
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //③基本属性值获取
        String username = (String)parser.parseExpression("userName").getValue(context);
        int credits = (Integer) parser.parseExpression("credits +
10").getValue(context);

        //④嵌套对象属性值获取
        String city = (String)parser.parseExpression("placeOfBirth.city").getValue
(context);
        ...
    }
}

```

从上面的示例中可以看出，对象属性解析与文本字符解析有所不同，对象属性解析需要在取值的时候传递一个计算上下文参数 EvaluationContext。在这里，将 User 实例作为上下文的根对象传递给 EvaluationContext，这样 SpEL 表达式解析器就可以根据属性路径表达式获取上下文中根对象的属性值。

9.4.3 数组、集合类型解析

在 SpEL 中，支持数组、集合类型（Map、List）的解析。数组支持标准 Java 语言创建数组的方法，如“new int[]{1,2,3}”。List 支持大括号括起来的内容，数据项之间用逗号隔开，如“{1,2,3,4}”、“{{'a','b'},{'x','y'}}”。目前 SpEL 还不支持多维数组初始化，如“new int[2][3]{{1,2,3},{4,5,6}}”。Map 采用如下方式表达：{userName:'tom',credits:100}，如代码清单 9-7 所示。

代码清单 9-7 CollectionExprSample 属性解析

```

package com.smart.spel;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.List;
import java.util.Map;

public class CollectionExprSample {
    public static void main(String[] args) {
        ...
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①数组表达式解析
        int[] array1 = (int[]) parser.parseExpression("new int[]{1,2,3}").getValue
            (context);
        int[][] array2 = (int[][][]) parser.parseExpression("new int[2][3]").getValue
            (context);

        //目前不支持多维数组初始化，以下语句将报错
        int[][] array3= (int[][][]) parser.parseExpression("new
            int[2][3]{{1,2,3},{4,5,6}}").getValue(context);

        //②List 表达式解析
        List list = (List) parser.parseExpression("{1,2,3,4}").getValue(context);
        List listOfLists = (List)
parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);

        //③列表字符串解析
        Map userInfo = (Map) parser.parseExpression("{userName:'tom',credits:100 }").
            getValue(context);
        List userInfo2 = (List) parser.parseExpression("{{userName:'tom',credits:100 },
            {userName:'tom',credits:100 }}").getValue(context);

        //④从数组、List、Map 中取值
        String interest1 =
            (String)parser.parseExpression("interestsArray[0]").getValue(context);
        String interest2 =
            (String)parser.parseExpression("interestsList[0]"). getValue(context);
        String interest3 =
            (String)parser.parseExpression("interestsMap ['interest1']").getValue
            (context);
        ...
    }
}

```

在 SpEL 中，要从解析器获取指定属性值，需要将根对象存储到 EvaluationContext 上下文中，如示例中的“new StandardEvaluationContext(user)”。

数组与 List 取值通过指定括号内的索引获取，如示例中的“interestsArray[0]”、“interestsList[0]”，其中 interestsArray、interestsList 均为 User 根对象的属性。

Map 取值通过键名获取，如示例中的“interestsMap['interest1']”，其中 interestsMap 也是 User 对象的属性，“interest1”为 Map 存储数据的键值。

9.4.4 方法解析

在 SpEL 中，方法调用支持 Java 可访问的方法，包括对象方法、静态方法，并支持可变方法参数；还可以调用 String 类型的所有可访问的方法，如 String#substring()，如代码清单 9-8 所示。

代码清单 9-8 MethodExprSample方法解析

```
package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class MethodExprSample {
    public static void main(String[] args) {
        User user = new User();
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①调用String方法
        String substring = parser.parseExpression(
            "'Spring SpEL'.substring(7)").getValue(String.class);
        Integer index = parser.parseExpression(
            "'Spring SpEL'.indexOf('SpEL')").getValue(Integer.class);

        //②调用实例方法
        boolean isCorrect = parser.parseExpression("validatePassword('123456')")
            .getValue(context, Boolean.class);
        //③调用私有方法，将发生错误
        isCorrect = parser.parseExpression("validatePassword2('123456')").getValue(
            context, Boolean.class);
        //④调用静态方法
        isCorrect = parser.parseExpression("validatePassword3('123456')").getValue(
            context, Boolean.class);

        //⑤调用对象方法，可变参数列表
        parser.parseExpression("addInterests('Js','C')").getValue(context, Boolean.class);
    }
}
```

在①处，在表达式中直接使用 String 提供的方法对文本字符进行各种操作。在②处，调用自定义类 User 的方法 User#validatePassword()。需要注意的是，私有方法是不能调用的，如示例中③处执行时将会报错。在④处，调用自定义类 User 的静态方法 User#validatePassword3()。在⑤处，调用自定义类 User 的可变参数方法 User# addInterests()。

9.4.5 操作符解析

SpEL 提供了丰富的操作符解析，支持关系操作符、逻辑操作符、算术运算操作符及正则表达式匹配等。

1. 关系操作符

在 SpEL 表达式中，可以使用标准 Java 操作符号，支持关系操作符：等于、不等于、小于、小于或等于、大于、大于或等于、正则表达式及 instanceof 操作符，如代码清单 9-9 所示。

代码清单 9-9 OperatorExprSample 关系操作符解析

```
package com.smart.spel;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class OperatorExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        //①关系操作符
        boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);
        boolean falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

        //②字符串关系比较
        trueValue = parser.parseExpression("\"black\" < \"block\"").getValue(Boolean.class);

        //③instanceof运算符
        falseValue = parser.parseExpression( "'xyz' instanceof
                                              T(int)").getValue(Boolean.class);

        //④正则匹配运算，前面为字符串，后面为正则表达式
        trueValue = parser.parseExpression(
            "'5.00' matches '^-[0-9]{2}\\.\\d{2})?\\$'").getValue(Boolean.class);
        falseValue = parser.parseExpression(
            "'5.0067' matches '\\-[0-9]{2}\\.\\d{2})?\\$'").getValue(Boolean.class);
    }
}
```

Java 语言本身并未提供字符串关系比较操作符，但 SpEL 则支持字符串关系比较，如示例中②处的 “\"black\" < \"block\"”。 instanceof 运算符是用来在运行时指出文本字符是否是特定类的一个实例，如示例中的 “'xyz' instanceof T(int)” 所示。需要注意的是 instanceof 操作符后面的类型表达格式：T(Java 类型)，如整型 T(int)、T(Integer)，字符类型 T(String)。在④处的表达式中使用 matches 关键字来对文本字符进行正则匹配运算，matches 后面是标准的 Java 正则表达式。

2. 逻辑操作符

逻辑操作符支持与操作（and 或&&）、或操作（or 或||）、非操作（!）。需要指出的是，在 SpEL 表达式中，既可以采用标准 Java 逻辑操作符，还可以采用“and”和“or”这两个操作符，如代码清单 9-10 所示。

代码清单 9-10 MethodExprSample逻辑操作符解析

```

package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class MethodExprSample {
    public static void main(String[] args) {
        User user = new User();
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①与操作, 结果为false
        boolean falseValue = parser.parseExpression("true && false").getValue(Boolean.class);

        String expression = "isVipMember('tom') and isVipMember('jony')";
        boolean trueValue = parser.parseExpression(expression).getValue(context, Boolean.class);

        //②或操作, 结果为true
        trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

        //③取非操作, 结果为false
        falseValue = parser.parseExpression("!true").getValue(Boolean.class);
    }
}

```

在 SpEL 表达式中，通过关系操作符关键字进行与操作，关键字的前后运算结果必须是 Java 标准的布尔类型。如下列表表达式是非法的：true and 0；因为 and 操作符后面的“0”是整型值，而不是一个布尔值。

3. 算术运算操作符

SpEL 表达式支持 Java 标准运算操作，其中加法运算符可用于数字、字符串和日期，减法运算符可用于数字和日期，乘法和除法运算符仅可用于数字。其他支持的数学运算包括取模（%）和指数幂（^），使用 Java 标准的运算符优先级规则。如代码清单 9-11 所示。

代码清单 9-11 OperatorExprSample算术运算操作符解析

```

package com.smart.spel;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class OperatorExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        //①加法操作, 运行结果等于1
        int two = parser.parseExpression("1 + 1").getValue(Integer.class);
    }
}

```

```

String testString = parser.parseExpression(
    "\"test\" + ' ' + \"string\")").getValue(String.class);

//②减法操作, 运行结果等于4
int four = parser.parseExpression("1 - -3").getValue(Integer.class);

//减法操作, 运行结果等于-9000
double d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class);

//③乘法操作, 运行结果等于6
int six = parser.parseExpression("-2 * -3").getValue(Integer.class);

//乘法操作, 运行结果等于24.0
double twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getValue(Double.class);

//④除法操作, 运行结果等于-2
int minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class);

//除法操作, 运行结果等于1.0
double one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class);

//⑤求余操作, 运行结果等于3
int three = parser.parseExpression("7 % 4").getValue(Integer.class);

// 求余操作, 运行结果等于1
one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class);

//⑥优先级算术运算, 运行结果等于-21
int minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Integer.class);
}

}

```

9.4.6 安全导航操作符

安全导航操作符来源于 Groovy 语言, 它避免了空指针异常。通常在访问对象方法或属性时需要验证该对象是否为空。为了避免烦琐的空对象验证, 可采用安全导航操作符, 它只会返回 `null` 而不是抛出异常。其格式是在获取对象属性操作符“.”前面添加一个“?”, 如代码清单 9-12 所示。

代码清单 9-12 SafeExprSample 安全导航操作符解析

```

package com.smart.spel;
import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

import java.util.Date;

```

```

public class SafeExprSample {
    public static void main(String[] args) {
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100);
        user.setPlaceOfBirth(new PlaceOfBirth("中国", "厦门"));

        ExpressionParser parser = new SpelExpressionParser();
        StandardEvaluationContext context = new StandardEvaluationContext(user);
        String city = parser.parseExpression("PlaceOfBirth?.City").getValue(context,
String.class);
        System.out.println(city); //打印“厦门”

        //①将PlaceOfBirth对象设置为null
        user.setPlaceOfBirth(null);

        //②不会抛出异常
        city = parser.parseExpression("PlaceOfBirth?.City").getValue(context,
String.class);
        System.out.println(city); //打印“null”
    }
}

```

在上面的示例中，使用了 SpEL 提供的安全导航操作符，如 “PlaceOfBirth?.City”，在获取 PlaceOfBirth 对象属性的时候，会先判断 PlaceOfBirth 对象是否为空；如果为空则直接返回 null，如果不为空则获取其属性值。在 Java 8.0 中提供了 java.util.Optional 来支持类似的功能。

9.4.7 三元操作符

可以在表达式内使用 if-then-else 条件逻辑三元操作符，也就是 Java 标准的条件表达式：<表达式 1>?<表达式 2>:<表达式 3>。如表达式 “true?'value1': 'value2'”，将返回 “value1”。一个真实的例子如代码清单 9-13 所示。

代码清单 9-13 IfThenElseExprSample 三元操作符解析

```

package com.smart.spel;

import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

import java.util.Date;

public class IfThenElseExprSample {
    public static void main(String[] args) {

```

```

User user = new User();
user.setUserName("tom");
user.setLastVisit(new Date());
user.setCredits(100);
user.setPlaceOfBirth(new PlaceOfBirth("中国", "厦门"));

ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext(user);

//三元操作表达式
String expression = "UserName == 'tom'? Credits+10:Credits";

Integer credits = parser.parseExpression(expression).getValue(context,
Integer.class);
System.out.println(credits); // 输出“110”
}
}

```

在示例中，创建了一个用户名为“tom”的用户实例“user”，并设置其积分为 100。在三元操作表达式中判断当前是否存在一个“tom”用户，如果存在，就添加 10 个积分。

9.4.8 Elvis 操作符

Elvis 操作符是 Groovy 语言中使用的三元操作符的缩写。在三元运算符中通常要重复变量两次，例如：

```

String name = "tom";
String displayName = name!=null?name:"Unknown";

```

可以使用 Elvis 操作符替代：name?:"Unknown"，代码非常简洁。下面来看一下具体的实例，如代码清单 9-14 所示。

代码清单 9-14 ElvisExprSample 操作符解析

```

package com.smart.spel;

import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.Date;

public class ElvisExprSample {
    public static void main(String[] args) {
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100);
        user.setPlaceOfBirth(new PlaceOfBirth("中国", "厦门"));

        ExpressionParser parser = new SpelExpressionParser();
        StandardEvaluationContext context = new StandardEvaluationContext(user);
    }
}

```

```

String userName = parser.parseExpression("UserName?:'用户名为空'").getValue
        (context, String.class);
System.out.println(userName); //输出 "tom"
user.setUserName(null);
userName = parser.parseExpression("UserName?:'用户名为空'").getValue(context,
        String.class);
System.out.println(userName); //输出 "用户名为空"
}
}
}

```

SpEL 提供的 Elvis 操作符非常简洁，其格式为“<var>?:<value>”。如果左边变量取值为 null，就取 value 值；否则就取 var 变量自身的值。

9.4.9 赋值、类型、构造器、变量

1. 赋值

属性设置是通过使用赋值运算符完成的，相当于调用 Expression#setValue()方法，也可以通过调用 Expression#getValue()方法赋值，如代码清单 9-15 所示。

代码清单 9-15 ObjectExprSample 赋值

```

package com.smart.spel;
import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.Date;

public class ObjectExprSample {
    public static void main(String[] args) {
        User user = new User();
        user.setUserName("tom");

        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①通过setValue赋值
        parser.parseExpression("userName").setValue(context, "jony");
        System.out.println(user.getUserName()); //输出 "jony"

        //②通过表达式赋值
        parser.parseExpression("userName='anyli'").getValue(context);
        System.out.println(user.getUserName()); //输出 "anyli"
    }
}

```

在①处通过 Expression 接口提供的 setValue()方法为表达式上下文对象属性赋值；在②处在调用 getValue()方法时通过赋值表达式直接赋值，如示例中的“userName='anyli'”。

2. 类型

T 操作符是一个非常实用的类型操作符，通过它可以从类路径加载指定类名称

(全限定名) 的 Class 对象，操作表达式为“T(全限定类名)”，其功能类似于 ClassLoader#loadClass()方法，如代码清单 9-16 所示。

代码清单 9-16 ObjectExprSample 类型

```
package com.smart.spel;
import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class ObjectExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        //①加载java.lang.String
        Class stringClass=parser.parseExpression("T(java.lang.String)").getValue
(Class.class);
        System.out.println(stringClass==java.lang.String.class); //输出 true

        //②加载com.smart.User
        Class userClass=parser.parseExpression("T(com.smart.User)").getValue(Class.
class);
        System.out.println(userClass == com.smart.User.class); //输出 true
    }
}
```

在①处通过 T 操作符加载 java.lang.String 类，返回类 String 的 Class 对象；在②处通过 T 操作符加载自定义类 User 的 Class 对象。在 SpEL 内部，通过 StandardTypeLocator#findType()方法加载类。上述类的名称都使用全限定类名。如果加载的目标类位于 java.lang 包下，则可以不带包名；对于其他包下的类，则必须带上完整的包名。

T 操作符还可以直接调用类静态方法，操作表达式为“T(全限定类名).静态方法”。例如，可以使用 T 操作符直接调用 Math# random()方法，获取一个随机数，如下：

```
Object randomValue = parser.parseExpression("T(java.lang.Math).random()").getValue();
System.out.println(randomValue); //返回一个随机数
```

通过 T 操作符可以在 SpEL 表达式中调用工具类中的任意静态方法。这是一个非常实用的特性，如编写一个日期工具类 DateUtils，假设其中有一个解析日期的静态方法 parseDate()，则可以通过 T 操作符直接调用“T(com.smart.DateUtils). parseDate('20160501')”。

3. 构造器

可以使用 new 运算符调用构造器来创建一个对象实例。除了基本类型（如整型、浮点型）和字符串，其他类需要使用全限定类名，如代码清单 9-17 所示。

代码清单 9-17 ObjectExprSample 构造器

```

package com.smart.spel;
...
public class ObjectExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        User user = parser.parseExpression("new com.smart.User('tom')").getValue(User.class);
        System.out.println(user.getUserName()); //输出“tom”
    }
}

```

4. 变量

变量可以在表达式中使用语法“#变量名”引用，变量设置使用 EvaluationContext#setVariable(var, value)方法，其中 var 为变量名称，value 为变量对应的值，如代码清单 9-18 所示。

代码清单 9-18 ObjectExprSample 变量解析

```

package com.smart.spel;
...
public class ObjectExprSample {
    public static void main(String[] args) {
        User user = new User("tom");
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①为newUserName变量设置新值
        context.setVariable("newUserName", "jony");

        //②取变量值，并赋值
        parser.parseExpression("userName=#newUserName").getValue(context);
        System.out.println(user.getUserName()); //输出“jony”

        //③ this变量值使用
        List<Integer> credits = new ArrayList<Integer>();
        credits.addAll(Arrays.asList(150, 100, 90, 50, 110, 130, 70));
        context.setVariable("credits", credits);
        List<Integer> creditsGreater100 =
            (List<Integer>)parser.parseExpression("#credits.?[#this>100]").getValue(context);
    }
}

```

在①处通过调用 EvaluationContext#setVariable()方法在动态上下文中定义一个新的变量“newUserName”。在②处的表达式中通过“#变量名”访问这个变量的值，并将取到的值赋给 User 对象的 userName 属性。在③处创建了一个积分列表 credits，并将积分列表设置到动态上下文中。

9.4.10 集合过滤

集合过滤是动态语言的一个强大功能，它允许通过一个过滤条件获取原集合的子集。集合过滤的语法为“?[selectExpression]”。例如，从一个积分列表中找到大于 100 的积分，代码如下：

```
List<Integer> creditsGreater100=
    (List<Integer>)parser.parseExpression("#credits.? [#this>100]").getValue(context);
```

对于 List 或 Set 来说，过滤条件是针对集合内的每个元素进行比较的；而对于 Map 来说，则是针对每一个条目项（Java 类型的 Map.Entry 对象）进行运算的，如代码清单 9-19 所示。

代码清单 9-19 ObjectExprSample 集合选择

```
package com.smart.spel;
...
public class ObjectExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext();
        Map<String, Integer> creditsMap = new HashMap();
        creditsMap.put("Tom", 95);
        creditsMap.put("Jony", 110);
        creditsMap.put("Morin", 85);
        creditsMap.put("Mose", 120);
        creditsMap.put("Morrow", 60);
        context.setVariable("credits", creditsMap);

        Map<String, Integer> creditsGreater100=
            (Map<String, Integer>)parser.parseExpression("#credits.? [value>90]").getValue(context);
    }
}
```

“#credits.? [value>90]” 表达式将返回一个新的 Map，即过滤出值大于 100 的条目。默认返回所有匹配的结果，也可以仅返回第一个或最后一个匹配项：用`^ [...]`获得第一个匹配值，如`#credits.^ [value>90]`；用`$ [...]`获得最后一个匹配值，如`#credits. $ [value>90]`。

9.4.11 集合转换

集合转换允许使用一个算子对集合内的元素进行运算，得到一个新的集合。集合转换的语法为“! [projectionExpression]”。来看一个具体的例子，如代码清单 9-20 所示。

代码清单 9-20 ObjectExprSample 集合转换

```
package com.smart.spel;
...
public class ObjectExprSample {
    public static void main(String[] args) {
```

```

ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = new StandardEvaluationContext();

List<Integer> credits= new ArrayList<Integer>();
credits.addAll(Arrays.asList(150,100,90,50,110,130,70));
context.setVariable("credits",credits);

List<Boolean> creditsGreater100=
    (List<Boolean>)parser.parseExpression("#credits.![#this>100]").getValue
(context);
}
}

```

在示例中，对 List 集合元素进行转换，得到一个判断积分是否大于 100 的结果集合：[true, false, false, true, true, false]。类似的，Map 也可以进行转换，此时转换表达式是对 Map 中的每个项目运用算子进行求值。

9.5 在 Spring 中使用 SpEL

在 XML 配置方式或注解配置方式中都可以使用 SpEL 表达式进行一些高级配置，两种方式都采用统一的语法使用 SpEL 表达式：#{<expression string>}。

9.5.1 基于 XML 的配置

在 Bean 配置中，可以使用 SpEL 表达式为 Bean 属性或构造函数入参注入动态值。下面以 Bean 属性使用 SpEL 表达式为例，如下：

```

<bean id="numberGuess" class="org.springframework.samples.NumberGuess"
    p:randomNumber="#{T(java.lang.Math).random()*100.0}"
/>

```

在示例中，通过 SpEL 提供的 T 类型操作符，直接调用 java.lang.Math 的静态方法来生成一个随机数，并把生成的随机数乘以 100.0 后赋值给 NumberGuess 的 randomNumber 属性。还可以通过 systemProperties 获取各个系统环境变量，如下：

```

<bean id="systemPropertyBean" class="com.smart.spel.SystemPropertyBean"
    p:osName="#{systemProperties['os.name']}"
    p:javaHome="#{systemProperties['os.name']}"
    p:classPath="#{systemProperties['java.class.path']}"
    p:javaVersion="#{systemProperties['java.class.path']}"
/>

```

可以在 Bean 配置时引用任何一个在 Spring 容器中已定义的 Bean 的属性（属性提供了相应的 get 方法），通过“Bean 名称.属性名”的方式来引用其他 Bean 的属性，如下：

```

<bean id="numberGuess" class="org.springframework.samples.NumberGuess"
    p:randomNumber="#{T(java.lang.Math).random()*100.0}"
/>
<bean id="shapeGuess" class="org.springframework.samples.ShapeGuess"

```

```
p:initialShapeSeed = "#{numberGuess.randomNumber}"
/>>
```

在上述代码中，定义了 NumberGuess 和 ShapeGuess 两个 Bean，ShapeGuess 中的 initialShapeSeed 属性通过表达式引用 NumberGuess 中定义的 randomNumber 属性。

9.5.2 基于注解的配置

@Value 注解可以标注在类的属性、方法及构造器函数上，用于从配置文件中加载一个参数值。下面是一个设置属性示例：

```
@Component
public class MyDataSource {

    @Value("#{properties['driverClassName']}")
    private String driverClassName;

    @Value("#{properties['url']}")
    private String url;

    @Value("#{properties['userName']}")
    private String userName;

    @Value("#{properties['password']}")
    private String password;

}
```

上述代码通过@Value 注解自动注入属性配置文件中的属性选项值，其中 properties 是定义的加载配置文件的 Bean 的名称。要让上述代码能够正常运行，需要在 Spring 中引入 util 工具命名空间，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       ...
       http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util-4.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context- 4.0.xsd">
    <util:properties id="properties" location="classpath:jdbc.properties" />
</beans>
```

上述 jdbc.properties 配置文件的配置选项如下：

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/dbName
userName=root
password=1234
```

细心的读者可能会发现，类似 “#{properties['password']}

 这种写法比较容易出错。Spring 提供了一种更简便的写法——属性占位符，只要在 Spring 的配置方法中添加一个 “**property-placeholder**”，就可以在表达式中使用 “\${属性}”，如下：

```

<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       ...
       http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util-4.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <util:properties id="properties" location="classpath:jdbc.properties" />
    <context:property-placeholder properties-ref="properties" />
</beans>

```

可以把上述代码改为如下形式：

```

@Component
public class MyDataSource {

    @Value("${driverClassName}")
    private String driverClassName;

    @Value("${url}")
    private String url;
    ...
}

```

在 Bean 的方法及构造器函数上使用@Value 注解与类属性的使用方法类似，这里不再赘述。

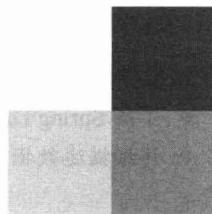
9.6 小结

在实际项目开发中，对于动态表达式的需求场景还是比较多的。早期，大家可能会在 Java 中引入一些动态脚本语言如 Rhino、Jython、JRuby 等类库来实现。虽然这些动态脚本语言可以实现各类复杂的表达式需求，但需要自行封装实现，工作量很大。

SpEL 表达式的出现为我们提供了一个轻量级的表达式框架，它实现了一套丰富的操作表达式，支持文本、对象、集合、方法表达式解析，并提供了丰富的表达式操作，可以满足大多数表达式场景需求。SpEL 已深入整合到 Spring 框架的 Bean 配置中，使用 SpEL 可以完成众多高级的 Bean 配置问题。

第3篇

数 据 篇



第 10 章

Spring 对 DAO 的支持

随着持久化技术的持续发展，各种持久化框架已趋于成熟，Oracle 也发布了几个持久化规范。Spring 对多个持久化技术提供了集成支持，包括 Hibernate、MyBatis、JPA、JDO；此外，还提供一个简化 JDBC API 操作的 Spring JDBC 框架。Spring 面向 DAO 制定了一个通用的异常体系，屏蔽具体持久化技术的异常，使业务层和具体的持久化技术实现解耦。另外，Spring 提供了模板类简化各种持久化技术的使用。通用的异常体系及模板类是 Spring 整合各种持久化技术的不二法门，Spring 不但借此实现了对多种持久化技术的整合，还可以整合潜在的各种持久化框架，体现了开发模式中“开-闭原则”的经典应用。

本章主要内容：

- ◆ Spring DAO 异常体系
- ◆ Spring 数据访问模板
- ◆ 配置数据源

本章亮点：

- ◆ 了解 Spring DAO 层的设计思想
- ◆ 详细描述数据源的配置

10.1 Spring 的 DAO 理念

DAO（Data Access Object）是用于访问数据的对象，虽然在大多数情况下将数据保存在数据库中，但这并不是唯一的选择，也可以将数据存储到文件中或 LDAP 中。DAO 不但屏蔽了数据存储的最终介质的不同，也屏蔽了具体的实现技术的不同。

早期，JDBC 是访问数据库的主流选择。近几年，数据持久化技术获得了长足的发

展, Hibernate、MyBatis、JPA、JDO 成为持久层中争放异彩的实现技术。只要为数据访问定义好 DAO 接口, 并使用具体的技术实现 DAO 接口的功能, 就可以在不同的实现技术间平滑地切换。

图 10-1 是一个典型的 DAO 应用实例, 在 UserDao 中定义访问 User 数据对象的接口方法, 业务层通过 UserDao 操作数据, 并使用具体的持久化技术实现 UserDao 接口方法, 这样业务层和具体的持久化技术就实现了解耦。

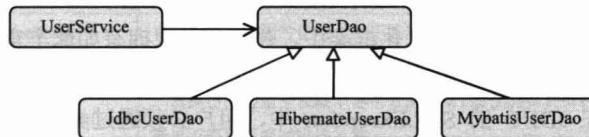


图 10-1 业务层通过 DAO 接口访问数据

提供 DAO 层的抽象可以带来一些好处: 首先, 可以很容易地构造模拟对象, 方便单元测试的开展; 其次, 在使用切面时会有更多的选择, 既可以使用 JDK 动态代理, 又可以使用 CGLib 动态代理。

Spring 本质上希望以统一的方式整合底层的持久化技术, 即以统一的方式进行调用及事务管理, 避免让具体的实现侵入到业务层的代码中。由于每种持久化技术都有各自的异常体系, 所以 Spring 提供了统一的异常体系, 使不同异常体系的阻抗得以消弭, 方便定义出和具体实现技术无关的 DAO 接口, 以及整合到相同的事务管理体系中。

10.2 统一的异常体系

统一的异常体系是整合不同的持久化技术的关键。Spring 提供了一套和实现技术无关的、面向 DAO 层语义的异常体系, 并通过转换器将不同持久化技术的异常转换成 Spring 的异常。

10.2.1 Spring 的 DAO 异常体系

在很多正统 API 或框架中, 检查型异常被过多地使用, 以至在使用 API 时, 代码里充斥着大量 try/catch 样板式的代码。在很多情况下, 除在 try/catch 中记录异常信息外, 并没有做多少实质性的工作。引发异常的问题往往是不可恢复的, 如数据连接失败、SQL 语句存在语法错误等。强制捕捉的检查型异常除限制开发人员的自由外, 并没有提供什么有价值的东西。因此, Spring 的异常体系都是建立在运行期异常的基础上的, 开发者可以根据需要捕捉感兴趣的异常。

JDK 的很多 API 之所以难用, 一个很大的原因就是检查型异常的泛滥, 如 JavaMail、EJB、JDBC 等。使用这些 API, 一堆堆异常处理的代码喧宾夺主地侵入到业务代码中,

破坏了代码的整洁和优雅。

Spring 在 org.springframework.dao 包中提供了一套完备优雅的 DAO 异常体系，这些异常都继承于 `DataAccessException`，而 `DataAccessException` 本身又继承于 `NestedRuntimeException`，`NestedRuntimeException` 异常以嵌套的方式封装了源异常。因此，虽然不同持久化技术的特定异常被转换到 Spring 的 DAO 异常体系中，但原始的异常信息并不会丢失；只要用户愿意，就可以方便地通过 `getCause()` 方法获取原始的异常信息。

Spring 的 DAO 异常体系并不和具体的实现技术相关，它从 DAO 概念的抽象层面定义了异常的目录树。在所有的持久化框架中，并没有发现拥有如此丰富语义的异常体系的框架。Spring 的这种设计无疑是独具匠心的，它使得开发人员关注某一特定语义的异常变得很容易。在 JDBC 的 `SQLException` 中，用户必须通过异常的 `getErrorCode()` 或 `getSQLState()` 方法获取错误代码，然后根据这些代码判断错误原因。这种过于底层的 API 不但带来了代码编程上的难度，而且也使代码的移植变得困难，因为 `getErrorCode()` 方法是数据库相关的。

Spring 以分类手法建立了异常分类目录，对于大部分应用来说，这个异常分类目录对异常类型的划分具有适当的颗粒度。一方面，使开发者从底层细如针麻的技术细节中脱离出来；另一方面，可以从这个语义丰富的异常体系中选择感兴趣的异常加以处理。图 10-2 列出了那些位于 Spring DAO 异常体系第一层次的异常类，每个异常类下可能拥有众多的子异常类。

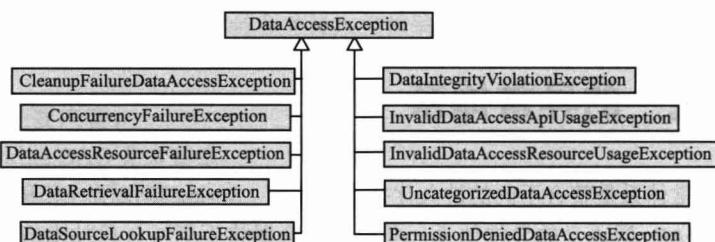


图 10-2 Spring DAO 异常体系

Spring DAO 异常体系类非常丰富，这里仅列出 `DataAccessException` 异常类下的子类。可以很容易地通过异常类的名字了解异常所代表的语义。下面通过表 10-1 对这些异常进行简单的描述。

表 10-1 Spring DAO 异常体系类

异常	说明
CleanupFailureDataAccessException	DAO 操作成功执行，但在释放数据资源时发生异常，如关闭 Connection 时发生异常等
ConcurrencyFailureException	表示在进行并发数据操作时发生异常，如乐观锁无法获取、悲观锁无法获取、死锁引发的失败等
DataAccessResourceFailureException	访问数据资源时失败，如无法获取数据连接、无法获取 Hibernate 的会话等
DataRetrievalFailureException	获取数据失败，如找不到对应主键的数据、使用了错误的列索引等

续表

异常	说 明
DataSourceLookupFailureException	无法从 JNDI 中查找到数据源
DataIntegrityViolationException	当数据操作违反了数据一致性限制时抛出的异常，如插入重复的主键、引用不存在的外键等
InvalidDataAccessApiUsageException	不正确地调用某一持久化技术时抛出的异常，如在 Spring JDBC 中查询对象，在调用前必须进行编译操作，如果忘记这项操作则会产生该异常。这种异常不是由底层数据资源产生的，而是由不正确地使用持久化技术产生的
InvalidDataAccessResourceUsageException	在访问数据源时使用了不正确的办法所抛出的异常，如 SQL 语句错误将抛出该异常
PermissionDeniedDataAccessException	数据访问时由于权限不足引发的异常，如仅拥有只读权限的用户试图进行数据更改操作时将抛出该异常
UncategorizedDataAccessException	其他未分类的异常都归到该异常中

为了进一步细化错误的问题域，Spring 对一级异常类进行了子类的细分，如 InvalidDataAccessResourceUsageException 就拥有十多个子类，下面是其中的两个子类，如图 10-3 所示。

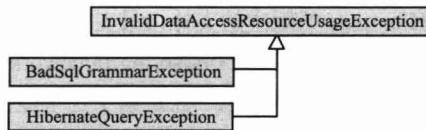


图 10-3 一级异常类的细分

对于 InvalidDataAccessResourceUsageException 异常，不同的持久化技术均有对应的子异常类。如 BadSqlGrammarException 对应 JDBC 实现技术的 SQL 语句语法错误异常，而 HibernateQueryException 对应 Hibernate 实现技术的查询语法异常。

Spring 的这个异常体系具有高度的可扩展性，当 Spring 需要对一种新的持久化技术提供支持时，只要为其定义一个对应的子异常就可以了，这种更改完全满足设计模式中的“开-闭原则”。

虽然 Spring 定义了如此丰富的异常类，但作为开发人员，仅需对感兴趣的异常进行处理即可。假设某个项目要求在发生乐观锁异常时，尝试再次获取乐观锁，而不是直接返回错误；那么，只需在代码中显式捕捉 ConcurrencyFailureException 异常，然后在 catch 代码块中编写满足需求的逻辑即可。其他众多的异常则可以简单地交由框架自动处理，如发生运行期异常时自动回滚事务。

10.2.2 JDBC 的异常转换器

传统的 JDBC API 在发生几乎所有的数据操作问题时都会抛出相同的 SQLException，它将异常的细节性信息封装在异常属性中。所以，如果希望了解异常的具体原因，则必须分析异常对象的信息。

SQLException 拥有两个代表异常具体原因的属性：错误码和 SQL 状态码。前者是数据库相关的，可通过 getErrorCode()方法返回，其值的类型是 int；而后者是一个标准的错误代码，可通过 getSQLState()方法返回，是一个 String 类型的值，由 5 个字符组成。

Spring 根据错误码和 SQL 状态码信息将 SQLException 译成 Spring DAO 的异常体系所对应的异常。在 org.springframework.jdbc.support 包中定义了 SQLExceptionTranslator 接口，该接口的两个实现类 SQLErrorCodeSQLExceptionTranslator 和 SQLStateSQLExceptionTranslator 分别负责处理 SQLException 中错误码和 SQL 状态码的翻译工作。将 SQLException 翻译成 Spring DAO 异常体系的工作是比较困难的，但 Spring 框架替我们完成了这项艰巨的工作并保证了转换的正确性，我们有充分的理由依赖这个转换的正确性。

10.2.3 其他持久化技术的异常转换器

由于各种框架级的持久化技术都拥有一个语义明确的异常体系，所以将这些异常转换为 Spring DAO 的体系相对轻松一些。下面将学习不同持久化技术的异常转换器。

Spring 4.0 移除了对 Hibernate 低版本的支持，只支持 Hibernate 3.6 之后的版本。另外，Spring 4.0 移除了对 TopLink 的支持。在 org.springframework.orm 包中，分别为 Spring 所支持的 ORM 持久化技术定义了一个子包，在这些子包中提供相应 ORM 技术的整合类。Spring 为各种 ORM 持久化技术所提供的异常转换器在表 10-2 中说明。

表 10-2 各ORM持久化技术异常转换器

ORM 持久化技术	异常转换器
Hibernate X.0（X 可为 3,4,5，下同）	org.springframework.orm.hibernateX.SessionFactoryUtils
JPA	org.springframework.orm.jpa.EntityManagerFactoryUtils
JDO	org.springframework.orm.jdo.PersistenceManagerFactoryUtils

这些工具类除了具有异常转换的功能，在进行事务管理时，还提供了从事务上下文中返回相同会话的功能（将在第 11 章深入讲解事务管理的知识）。

Spring 也支持 MyBatis ORM 持久化技术，由于 MyBatis 抛出的异常是和 JDBC 相同的 SQLException 异常，所以直接采用和 JDBC 相同的异常转换器。

10.3 统一数据访问模板

到一个餐馆用餐，大抵会经历这样一个流程：进入餐馆→迎宾小姐问候并引到适合的位置→拿起菜单点菜→用餐→埋单→离开餐馆。之所以我们喜欢时不时到餐馆用餐，就是因为我们只要点菜→用餐→埋单就可以了，幕后的烹饪制作、刷锅洗盘等工作完全不用关心，一切已经由餐馆服务人员按照服务流程按部就班、有条不紊地执行了。衡量

一个餐馆服务质量好坏的一个重要标准是我们无须关心他们所负责的流程：不用催问菜为什么还没有上好（不但快而且服务态度佳），不用关心盘子为什么不干净（不但干净而且已经进行了消毒）。

从某种角度看，与其说餐馆为我们提供了服务，还不如说我们参与到餐馆的流程中：不管什么顾客点的菜都由相同的厨师烹制，不管什么顾客都按单付钱。在幕后，餐馆拥有一个服务模板，模板中定义的流程可以用于应付所有的顾客，只要为顾客提供几个专有需求（点的菜可不一样，座位可以自由选择），其他一切都按模板化的方式处理。

在直接使用具体的持久化技术时，大多需要处理整个流程，并没有享受餐馆用餐式的便捷。Spring 为支持的持久化技术分别提供了模板访问的方式，降低了使用各种持久化技术的难度，因此可以大幅度地提高开发效率。

10.3.1 使用模板和回调机制

下面是一段使用 JDBC 进行数据访问操作的简单代码，我们已经尽可能地简化了整个过程的处理，但以下步骤都是不可或缺的，如代码清单 10-1 所示。

代码清单 10-1 JDBC 数据访问

```
public void saveCustomer(Customer customer) throws Exception {
    Connection con=null;
    PreparedStatement stmt=null;
    try {
        //①获取资源
        con=getConnection();
        //②启动事务
        con.setAutoCommit(false);
        //③具体的数据访问操作和处理
        stmt=con.prepareStatement("insert into CUSTOMERS(ID,NAME) values(?,?)");
        stmt.setLong(1,customerId);
        stmt.setString(2,customer.getName());
        stmt.execute();
        ...
        stmt.execute();
        //④提交事务
        con.commit();
    }catch(Exception e){
        try{
            //⑤回滚事务
            con.rollback();
        }catch(SQLException sqlex){
            sqlex.printStackTrace(System.out);
        }
        throw e;
    }
}
```

```
    }finally{
        //⑥释放资源
        try{
            stmt.close();
            con.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

如上述代码所示，JDBC 数据访问操作按以下流程进行：

- (1) 准备资源。
 - (2) 启动事务。
 - (3) 在事务中执行具体的数据访问操作。
 - (4) 提交/回滚事务。
 - (5) 关闭资源，处理异常。

按照传统的方式，在编写任何带事务的数据访问程序时，都需要重复编写上面的代码，而其中只有粗体部分所示的代码是业务相关的，其他代码都是在例行公事，因而导致大量八股文式的代码充斥着整个程序。



轻松一刻

八股文的每篇文章均由一定的格式、字数构成，即由破题、承题、起讲、入手、起股、中股、后股、束股八部分组成。破题是用两句话将题目的意义破开；承题是承接破题的意义而说明之；起讲为议论的开始，首二字用“意味”、“若曰”、“以为”、“且夫”、“尝思”等开端；入手为起讲；起股、中股、后股、束股才是正式议论，以中股为全篇重心。在这四股中，每股又有两股排比对偶的文字，合共八股，故名八股文。



Spring 将这个相同的数据访问流程固化到模板类中，并将数据访问中固定和变化的部分分开，同时保证模板类是线程安全的，以便多个数据访问线程共享同一个模板实例。固定的部分在模板类中已经准备好，而变化的部分通过回调接口开放出来，用于定义具体数据访问和结果返回的操作。图 10-4 描述了模板类拆分固定和变化部分的逻辑。

这样，只要编写好回调接口，并调用模板类进行数据访问，就可以得到预想的结果：数据访问成功执行，前置和后置的样板化工作也按顺序正确执行，在提高开发效率的同时保证了资源使用的正确性，彻底消除了因忘记进行资源释放而引起的资源泄露问题。

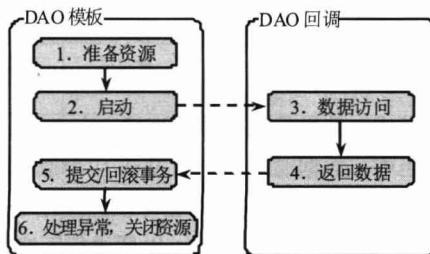


图 10-4 Spring DAO 模板和回调

10.3.2 Spring 为不同持久化技术所提供的模板类

Spring 为各种支持的持久化技术都提供了简化操作的模板和回调，在回调中编写具体的数据操作逻辑，使用模板执行数据操作，在 Spring 中，这是典型的数据操作模式。下面来了解一下 Spring 为不同的持久化技术所提供的模板类，如表 10-3 所示。关于模板类的具体使用，将在本书后续章节介绍。

表 10-3 不同持久化技术对应的模板类

ORM 持久化技术	模 板 类
JDBC	org.springframework.jdbc.core.JdbcTemplate
Hibernate X.0	org.springframework.orm.hibernateX.HibernateTemplate
JPA	org.springframework.orm.jpa.JpaTemplate
JDO	org.springframework.orm.jdo.JdoTemplate

如果直接使用模板类，则一般需要在 DAO 中定义一个模板对象并提供数据资源。Spring 为每种持久化技术都提供了支持类，支持类中已经完成了这样的功能。这样，只需扩展这些支持类，就可以直接编写实际的数据访问逻辑，因此更加方便。

不同持久化技术的支持类如表 10-4 所示。

表 10-4 持久化技术的支持类

ORM 持久化技术	支 持 类
JDBC	org.springframework.jdbc.core.JdbcDaoSupport
Hibernate X.0	org.springframework.orm.hibernateX.HibernateDaoSupport
JPA	org.springframework.orm.jpa.JpaDaoSupport
JDO	org.springframework.orm.jdo.JdoDaoSupport

这些支持类都继承于 `dao.support.DaoSupport` 类，`DaoSupport` 类实现了 `InitializingBean` 接口，在 `afterPropertiesSet()` 接口方法中检查模板对象和数据源是否被正确设置，否则将抛出异常。

所有的支持类都是 `abstract` 的，其目的是希望被继承使用，而非直接使用。

10.4 数据源

不管采用何种持久化技术，都必须拥有数据连接。在 Spring 中，数据连接是通过数据源获得的。在以往的应用中，数据源一般是由 Web 应用服务器提供的。在 Spring 中，不但可以通过 JNDI 获取应用服务器的数据源，也可以直接在 Spring 容器中配置数据源。此外，还可以通过代码的方式创建一个数据源，以便进行无容器依赖的单元测试。

10.4.1 配置一个数据源

Spring 在第三方依赖包中包含了两个数据源的实现类包：其一是 Apache 的 DBCP；其二是 C3P0。可以在 Spring 配置文件中利用二者中的任何一个配置数据源。

1. DBCP 数据源

DBCP 是一个依赖 Jakarta commons-pool 对象池机制的数据库连接池，所以在类路径下还必须包括 commons-pool 的类包。下面是使用 DBCP 配置 MySQL 数据源的片段：

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"
      p: driverClassName="com.mysql.jdbc.Driver"
      p: url="jdbc:mysql://localhost:3309/sampled"
      p: username="root"
      p: password="1234"/>
```

BasicDataSource 提供了 close()方法关闭数据源，所以必须设定 destroy-method="close" 属性，以便 Spring 容器关闭时，数据源能够正常关闭。

假设数据库是 MySQL，如果数据源配置不当，则将可能发生经典的“8 小时问题”。原因是 MySQL 在默认情况下如果发现一个连接的空闲时间超过 8 小时，则将会在数据库端自动关闭这个连接。而数据源并不知道这个连接已经被数据库关闭了，当它将这个无用的连接返回给某个 DAO 时，DAO 就会报无法获取 Connection 的异常。

除以上必需的数据源属性外，还有一些常用的属性，如表 10-5 所示。

表 10-5 DBCP 设置参数说明

分类	属性	默认值	说 明
事务属性	defaultAutoCommit	true	连接池创建的连接的默认 auto-commit 状态
	defaultReadOnly	驱动默认	连接池创建的连接的默认 read-only 状态。如果没有设置，则 setReadOnly()方法将不会被调用
	defaultTransactionIsolation	驱动默认	连接池创建的连接的默认的 TransactionIsolation 状态，可选值包括：NONE、READ_COMMITTED、READ_UNCOMMITTED、REPEATABLE_READ 和 SERIALIZABLE

续表

分类	属性	默认值	说 明
数据源连接数量	initialSize	0	初始化连接：连接池启动时创建的初始化连接数量
	maxActive	8	最大活动连接：连接池在同一时间能够分配的最大活动连接数量。如果设置为非正数，则表示不限制
	maxIdle	8	最大空闲连接：连接池中允许保持空闲状态的最大连接数量，超过的空闲连接将被释放。如果设置为负数，则表示不限制
	minIdle	0	最小空闲连接：连接池中允许保持空闲状态的最小连接数量，低于这个数量将创建新的连接。如果设置为 0，则不创建
	maxWait	无限	最大等待时间：当没有可用连接时，连接池等待连接被归还的最大时间（以毫秒计数），超过时间则抛出异常。如果设置为 -1，则表示无限等待
连接健康情况维护和检测	validationQuery	无默认值	SQL 查询语句。在将连接返回给调用者之前，用此 SQL 验证从连接池取出的连接是否可用。如果指定，则查询必须是一个 SQL SELECT，并且必须返回至少一行记录。在 MySQL 中可以设置为“select 1”，在 Oracle 中可以设置为“select 1 from dual”
	testOnBorrow	true	指明是否在从连接池中取出连接前进行检验，如果检验失败，则从连接池中去除该连接并尝试取出另一个新的连接。注意：设置为 true 后如果要生效，则 validationQuery 参数必须正确设置
	testOnReturn	false	指明是否在归还到连接池中前进行检验。注意：设置为 true 后如果要生效，则 validationQuery 参数必须正确设置
	testWhileIdle	false	指明连接是否被空闲连接回收器（如果有）进行检验。如果检测失败，则连接将被从连接池中去除。注意：设置为 true 后如果要生效，则 validationQuery 参数必须正确设置
	timeBetweenEvictionRunsMillis	-1	空闲连接回收器线程运行的周期，以毫秒为单位。如果设置为非正数，则不运行空闲连接回收器线程。注意：启用该参数时，validationQuery 参数必须正确设置
缓存语句	numTestsPerEvictionRun	3	在每次空闲连接回收器线程（如果有）运行时检查的连接数量
	minEvictableIdleTimeMillis	1000 × 60 × 30	连接在可被空闲连接回收器回收前已经在连接池中的空闲时间，以毫秒为单位
连接泄露回收	poolPreparedStatements	false	开启连接池的 prepared statement 池功能。当设置为 true 时，所有 CallableStatement 和 PreparedStatement 都会被缓存起来
	maxOpenPreparedStatements	无限制	PreparedStatement 池能够同时分配的打开的 statements 的最大数量。如果设置为 0，则表示不限制
连接泄露回收	removeAbandoned	false	标记是否删除泄露的连接。如果 removeAbandoned 设置为 true，那么“存在泄露嫌疑”的连接可能被连接池主动清除。这个机制在 $(getNumIdle() < 2) \text{ and } (getNumActive() > getMaxActive() - 3)$ 条件满足时被触发。举例来说：当 maxActive=20，活动连接为 18，空闲连接为 1 时，可以触发 removeAbandoned 动作。但是活动连接只有在未被使用的时间超过 removeAbandonedTimeout 时才被回收，默认为 300 秒。如果应用需要一个进行长操作的连接，则需要考虑将 removeAbandonedTimeout 设置得更长一些，否则可能发生正常连接被强制清除的情况

续表

分类	属性	默认值	说 明
连接 泄露 回收	removeAbandonedTimeout	300	泄露的连接可以被回收的超时值，单位为秒
	logAbandoned	false	标记当 Statement 或连接被泄露时是否打印程序的 stack traces 日志。被泄露的 Statements 和连接的日志添加在每个连接打开或者生成新的 Statement，因为需要生成 stack trace

如果采用 DBCP 的默认配置，由于 testOnBorrow 属性的默认值为 true，数据源在将连接将交给 DAO 前，会事先检测这个连接是否是好的，如果连接有问题（在数据库端被关闭），则会取一个其他的连接给 DAO，所以不会有“8 小时问题”。如果每次将连接交给 DAO 时都检测连接的有效性，那么在高并发的应用中将会带来性能问题，因为它会需要更多的数据库访问请求。

一种推荐的高效方式是：将 testOnBorrow 设置为 false，而将 testWhileIdle 设置为 true，再设置好 timeBetweenEvictionRunsMillis 值。这样，DBCP 将通过一个后台线程定时地对空闲连接进行检测，当发现无用的空闲连接（那些被数据库关闭的连接）时，就会将它们清除掉。只要将 timeBetweenEvictionRunsMillis 值设置为小于 8 小时，那些被 MySQL 关闭的空闲连接就可以被清除出去，从而避免了“8 小时问题”。

当然，MySQL 本身可以通过调整 interactive-timeout（以秒为单位）配置参数，更改空闲连接的过期时间。所以，在设置 timeBetweenEvictionRunsMillis 值时，必须首先获知 MySQL 空闲连接的最大过期时间。

2. C3P0 数据源

C3P0 是一个开放源码的 JDBC 数据源实现项目，实现了 JDBC3 和 JDBC2 扩展规范说明的 Connection 和 Statement 池。下面使用 C3P0 配置一个 Oracle 数据源，代码如下：

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close"
      p:driverClass="oracle.jdbc.driver.OracleDriver"
      p:jdbcUrl="jdbc:oracle:thin:@localhost:1521:ora9i"
      p:use="admin"
      p:password="1234"/>
```

ComboPooledDataSource 和 BasicDataSource 一样提供了一个用于关闭数据源的 close()方法，这样就可以保证 Spring 容器关闭时数据源能够被成功释放。

C3P0 拥有比 DBCP 更丰富的配置属性，通过这些属性，可以对数据源进行各种有效的控制。

- acquireIncrement：当连接池中的连接用完时，C3P0 一次性创建新连接的数目。
- acquireRetryAttempts：定义在从数据库获取新连接失败后重复尝试获取的次数，默认为 30。
- acquireRetryDelay：尝试获取连接的间隔时间，单位为毫秒，默认为 1000。
- autoCommitOnClose：连接关闭时默认将所有未提交的操作回滚。默认为 false。

- **automaticTestTable:** C3P0 将创建一张名为 Test 的空表，并使用其自带的查询语句进行测试。如果定义了这个参数，那么属性 preferredTestQuery 将被忽略。用户不能在这张 Test 表上进行任何操作，它仅为 C3P0 测试所用。默认为 null。
- **breakAfterAcquireFailure:** 获取连接失败将会引起所有等待获取连接的线程抛出异常。但是数据源仍有效保留，并在下次调用 getConnection()方法时继续尝试获取连接。如果设为 true，那么在尝试获取连接失败后，该数据源将声明已断开并永久关闭。默认为 false。
- **checkoutTimeout:** 当连接池用完时，客户端调用 getConnection()方法后等待获取新连接的时间，超时后将抛出 SQLException，如果设为 0 则无限期等待。单位为毫秒，默认为 0。
- **connectionTesterClassName:** 通过实现 ConnectionTester 或 QueryConnectionTester 的类来测试连接，类名需要设置为全限定名。默认为 com.mchange.v2.C3P0.impl.DefaultConnectionTester。
- **idleConnectionTestPeriod:** 间隔多少秒检查所有连接池中的空闲连接。默认为 0，表示不检查。
- **initialPoolSize:** 初始化时创建的连接数，应在 minPoolSize 与 maxPoolSize 之间取值。默认为 3。
- **maxIdleTime:** 最大空闲时间，超过空闲时间的连接将被丢弃。为 0 或负数则永不丢弃。默认为 0。
- **maxPoolSize:** 连接池中保留的最大连接数。默认为 15。
- **maxStatements:** JDBC 的标准参数，用以控制数据源内加载的 PreparedStatement 数量。但由于预缓存的 Statement 属于单个 Connection 而不是整个连接池，所以设置这个参数需要考虑多方面的因素，如果 maxStatements 与 maxStatementsPerConnection 均为 0，则缓存被关闭。默认为 0。
- **maxStatementsPerConnection:** 连接池内单个连接所拥有的最大缓存 Statement 数。默认为 0。
- **numHelperThreads:** C3P0 是异步操作的，缓慢的 JDBC 操作通过帮助进程完成。扩展这些操作可以有效地提升性能，通过多线程实现多个操作同时被执行。默认为 3。
- **preferredTestQuery:** 定义所有连接测试都执行的测试语句。在使用连接测试的情况下，这个参数能显著提高测试速度。测试的表必须在初始数据源的时候就存在。默认为 null。
- **propertyCycle:** 用户修改系统配置参数执行前最多等待的秒数。默认为 300。
- **testConnectionOnCheckout:** 因性能消耗大，请只在需要的时候使用它。如果设为 true，那么在每个 connection 提交的时候都将校验其有效性。建议使用

idleConnection TestPeriod 或 automaticTestTable 等方法来提升连接测试的性能。默认为 false。

- testConnectionOnCheckin: 如果设为 true, 那么在取得连接的同时将校验连接的有效性。默认为 false。

对于连接有效性的检测, 请参照我们推荐的 DBCP 配置方式。

3. 使用属性文件

数据源的配置信息有可能经常需要改动, 同时可能被其他工程复用。此外, 用户名/密码等信息比较敏感, 可能需要使用特别的安全措施。所以一般将数据源的配置信息独立到一个属性文件中, 通过<context:property-placeholder>引入属性文件, 以\${xxx}的方式引用属性。示例代码如下:

```
<context:property-placeholder
    location="/WEB-INF/jdbc.properties "/>
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p: driverClassName="${jdbc.driverClassName}"
    p: url="${jdbc.url}"
    p: username="${jdbc.username}"
    p: password="${jdbc.password}"/>
```

在 jdbc.properties 属性文件中定义属性值, 如下:

```
jdbc.driverClassName= com.mysql.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3309/sampledb
jdbc.username=root
jdbc.password=1234
```

这里, 属性文件的内容是以明文方式存放的。如果需要对属性文件的内容进行加密, 请参见 6.3 节。



提示

经常有开发者在\${xxx}前后不小心输入一些空格, 这些空格字符将和变量合并后作为属性的值。如 p: username=" \${jdbc.username} " 的属性配置项, 在前后都有空格, 被解析后, username 的值为 “1234”, 这将造成最终的错误, 因此需要特别小心。

10.4.2 获取 JNDI 数据源

如果应用配置在高性能的应用服务器(如 WebLogic 或 WebSphere 等)上, 则可能更希望使用应用服务器本身提供的数据源。应用服务器的数据源使用 JNDI 开放调用者使用, Spring 为此专门提供了引用 JNDI 数据源的 JndiObjectFactoryBean 类。下面是一个简单的配置:

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean"
    p: jndiName="java:comp/env/jdbc/bbt"/>
```

通过 jndiName 指定引用的 JNDI 数据源名称。

Spring 为获取 Java EE 资源提供了一个 jee 命名空间，通过 jee 命名空间，可以有效地简化 Java EE 资源的引用。下面是使用 jee 命名空间引用 JNDI 数据源的配置：

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
                           http://www.springframework.org/schema/jee
                           http://www.springframework.org/schema/jee/spring-jee-4.0.xsd">
    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/bbt"/>
</beans>
```

10.4.3 Spring 的数据源实现类

Spring 本身也提供了一个简单数据源实现类 DriverManagerDataSource，它位于 org.springframework.jdbc.datasource 包中。这个类实现了 javax.sql.DataSource 接口，但它并没有提供池化连接的机制；每次调用 getConnection() 方法获取新连接时，只是简单地创建一个新的连接。因此，这个数据源类比较适合在单元测试或简单的独立应用中使用，因为它不需要额外的依赖类。

下面来看一下 DriverManagerDataSource 类的简单使用，如下：

```
DriverManagerDataSource ds = new DriverManagerDataSource ();
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUrl("jdbc:mysql://localhost:3309/sampledb");
ds.setUsername("root");
ds.setPassword("1234");
Connection actualCon = ds.getConnection();
```

当然，也可以通过配置的方式直接使用 DriverManagerDataSource 类。

10.5 小结

Spring 支持目前大多数常用的数据持久化技术。Spring 定义了一套面向 DAO 层的异常体系，并为各种支持的持久化技术提供了异常转换器。这样，在设计 DAO 接口时，就可以抛开具体的实现技术，定义统一的接口。

不管采用何种持久化技术，访问数据的流程是相对固定的。Spring 将数据访问流程划分为固定和变化两部分，并以模板的方式定义好流程，用回调接口将变化的部分开放出来，留给开发者自行定义。这样，仅需提供业务相关的逻辑就可以完成整体的数据访问。Spring 为了进一步简化持久化模板类的使用，为各种持久化技术提供了便捷的支持。

类。支持类不但包含数据访问模板，还包含数据源或会话等内容。通过扩展支持类定义自己的数据访问类是最简单的数据访问方式。

不管采用何种持久化技术，都需要定义数据源。在实际部署时，可能会直接采用应用服务器本身提供的数据源，这时可以通过 JndiObjectFactoryBean 或 jee 命名空间引用 JNDI 数据源。

第 11 章

Spring 的事务管理

在使用 Spring 开发应用时, Spring 的事务管理可能是被使用最多、应用最广的功能。对于大多数应用者来说, Spring 事务管理所带来的好处是实实在在的。学习 Spring 而忽略了事务管理, 就像到了黄山不登天都峰、讲隋唐演义不讲李元霸一样, 最精彩的部分被忽略了。Spring 不但提供了和底层事务源无关的事务抽象, 还提供了声明性事务的功能, 可以让程序从事务代码中解放出来。而这正是 EJB 自鸣得意的地方, 但现在我们有了更好、更便捷的替代方案。

本章主要内容:

- ◆ 事务属性值的实际意义
- ◆ ThreadLocal 的工作机制
- ◆ Spring 事务管理的体系结构
- ◆ 基于 XML 的事务配置
- ◆ 基于注解的事务配置

本章亮点:

- ◆ 介绍数据库事务的基础知识, 它们是掌握事务配置的基础
- ◆ 深入学习 ThreadLocal 知识, 进而揭示 Spring 事务同步管理器的工作原理
- ◆ Spring 注解 AspectJ 织入的过程

11.1 数据库事务基础知识

Spring 虽然提供了灵活方便的事务管理功能, 但这些功能都是基于底层数据库本身的事務处理机制工作的。要深入了解 Spring 的事务管理和配置, 有必要先对数据库事务的基础知识进行学习。

11.1.1 何为数据库事务

“一荣俱荣，一损俱损”这句话很能体现事务的思想，很多复杂的事物要分步进行，但它们组成了一个整体，要么整体生效，要么整体失效。这种思想反映到数据库上，就是多条 SQL 语句，要么所有执行成功，要么所有执行失败。

数据库事务有严格的定义，它必须同时满足 4 个特性：原子性（Atomic）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），简称为 ACID。下面是对每个特性的说明。

- 原子性：表示组成一个事务的多个数据库操作是一个不可分割的原子单元，只有所有的操作执行成功，整个事务才提交。事务中的任何一个数据库操作失败，已经执行的任何操作都必须撤销，让数据库返回到初始状态。
- 一致性：事务操作成功后，数据库所处的状态和它的业务规则是一致的，即数据不会被破坏。如从 A 账户转账 100 元到 B 账户，不管操作成功与否，A 账户和 B 账户的存款总额是不变的。
- 隔离性：在并发数据操作时，不同的事务拥有各自的数据空间，它们的操作不会对对方产生干扰。准确地说，并非要求做到完全无干扰。数据库规定了多种事务隔离级别，不同的隔离级别对应不同的干扰程度，隔离级别越高，数据一致性越好，但并发性越弱。
- 持久性：一旦事务提交成功后，事务中所有的数据操作都必须被持久化到数据库中。即使在提交事务后，数据库马上崩溃，在数据库重启时，也必须保证能够通过某种机制恢复数据。

在这些事务特性中，数据“一致性”是最终目标，其他特性都是为达到这个目标而采取的措施、要求或手段。

数据库管理系统一般采用重执行日志来保证原子性、一致性和持久性。重执行日志记录了数据库变化的每一个动作，数据库在一个事务中执行一部分操作后发生错误退出，数据库即可根据重执行日志撤销已经执行的操作。此外，对于已经提交的事务，即使数据库崩溃，在重启数据库时也能够根据日志对尚未持久化的数据进行相应的重执行操作。

和 Java 程序采用对象锁机制进行线程同步类似，数据库管理系统采用数据库锁机制保证事务的隔离性。当多个事务试图对相同的数据进行操作时，只有持有锁的事务才能操作数据，直到前一个事务完成后，后面的事务才有机会对数据进行操作。Oracle 数据库还使用了数据版本的机制，在回滚段为数据的每个变化都保存一个版本，使数据的更改不影响数据的读取。

11.1.2 数据并发的问题

一个数据库可能拥有多个访问客户端，这些客户端都可用并发的方式访问数据库。数据库中的相同数据可能同时被多个事务访问，如果没有采取必要的隔离措施，就会导致各种并发问题，破坏数据的完整性。这些问题可以归结为 5 类，包括 3 类数据读问题（脏读、不可重复读和幻象读）及 2 类数据更新问题（第一类丢失更新和第二类丢失更新）。下面分别通过实例讲解引发问题的场景。

1. 脏读（dirty read）

A 事务读取 B 事务尚未提交的更改数据，并在这个数据的基础上进行操作。如果恰好 B 事务回滚，那么 A 事务读到的数据根本是不被承认的。来看取款事务和转账事务并发时引发的脏读场景。

时间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4		取出 500 元，把余额改为 500 元
T5	查询账户余额为 500 元（脏读）	
T6		撤销事务，余额恢复为 1000 元
T7	汇入 100 元，把余额改为 600 元	
T8	提交事务	

在这个场景中，B 希望取款 500 元，而后又撤销了动作，而 A 往相同的账户中转账 100 元，就因为 A 事务读取了 B 事务尚未提交的数据，因而造成账户白白丢失了 500 元。在 Oracle 数据库中，不会发生脏读的情况。



轻松一刻

一个有结巴的人在饮料店柜台前转悠，老板很热情地迎上来说：“喝一瓶？”结巴连忙说：“我…喝…喝…”老板麻利地打开易拉罐递给结巴，结巴终于憋出了他的那句话：“我…喝…喝…喝不起啊！”

2. 不可重复读（unrepeatable read）

不可重复读是指 A 事务读取了 B 事务已经提交的更改数据。假设 A 在取款事务的过程中，B 往该账户转账 100 元，A 两次读取账户的余额发生不一致。

时间	取款事务 A	转账事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元

续表

时 间	取款事务 A	转账事务 B
T4	查询账户余额为 1000 元	
T5		取出 100 元，把余额改为 900 元
T6		提交事务
T7	查询账户余额为 900 元（和 T4 读取的不一致）	

在同一事务中，T4 时间点和 T7 时间点读取的账户存款余额不一致。

3. 幻象读 (phantom read)

A 事务读取 B 事务提交的新增数据，这时 A 事务将出现幻象读的问题。幻象读一般发生在计算统计数据的事务中。举个例子，假设银行系统在同一个事务中两次统计存款账户的总金额，在两次统计过程中，刚好新增了一个存款账户，并存入 100 元，这时，两次统计的总金额将不一致。

时 间	统计金额事务 A	转账事务 B
T1		开始事务
T2	开始事务	
T3	统计总存款数为 10000 元	
T4		新增一个存款账户，存款为 100 元
T5		提交事务
T6	再次统计总存款数为 10100 元（幻象读）	

如果新增数据刚好满足事务的查询条件，那么这个新数据就进入了事务的视野，因而产生了两次统计结果不一致的情况。

幻象读和不可重复读是两个容易混淆的概念，前者是指读到了其他已经提交事务的新增数据，而后者是指读到了已经提交事务的更改数据（更改或删除）。为了避免这两种情况，采取的对策是不同的：防止读到更改数据，只需对操作的数据添加行级锁，阻止操作中的数据发生变化；而防止读到新增数据，则往往需要添加表级锁——将整张表锁定，防止新增数据（Oracle 使用多版本数据的方式实现）。

4. 第一类丢失更新

A 事务撤销时，把已经提交的 B 事务的更新数据覆盖了。这种错误可能造成很严重的问题，通过下面的账户取款转账就可以看出来。

时 间	取款事务 A	转账事务 B
T1	开始事务	
T2		开始事务
T3	查询账户余额为 1000 元	
T4		查询账户余额为 1000 元
T5		汇入 100 元，把余额改为 1100 元
T6		提交事务
T7	取出 100 元，把余额改为 900 元	

续表

时 间	取款事务 A	转账事务 B
T8	撤销事务	
T9	余额恢复为 1000 元 (丢失更新)	

A 事务在撤销时，“不小心”将 B 事务已经转入账户的金额给抹去了。

5. 第二类丢失更新

A 事务覆盖 B 事务已经提交的数据，造成 B 事务所做操作丢失。

时 间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4	查询账户余额为 1000 元	
T5		取出 100 元，把余额改为 900 元
T6		提交事务
T7	汇入 100 元	
T8	提交事务	
T9	把余额改为 1100 元 (丢失更新)	

在上面的例子中，由于支票转账事务覆盖了取款事务对存款余额所做的更新，导致银行最后损失了 100 元。相反，如果转账事务先提交，那么用户账户将损失 100 元。

11.1.3 数据库锁机制

数据并发会引发很多问题，在一些场合下有些问题是允许的，但在另一些场合下可能是致命的。数据库通过锁机制解决并发访问的问题，虽然不同的数据库在实现细节上存在差别，但原理基本上是一样的。

按锁定的对象的不同，一般可以分为表锁定和行锁定。前者对整张表进行锁定，而后者对表中的特定行进行锁定。从并发事务锁定的关系上看，可以分为共享锁定和独占锁定。共享锁定会防止独占锁定，但允许其他的共享锁定。而独占锁定既防止其他的独占锁定，也防止其他的共享锁定。为了更改数据，数据库必须在进行更改的行上施加行独占锁定，INSERT、UPDATE、DELETE 和 SELECT FOR UPDATE 语句都会隐式采用必要的行锁定。下面介绍一下 Oracle 数据库常用的 5 种锁定。

- 行共享锁定：一般通过 SELECT FOR UPDATE 语句隐式获得行共享锁定，在 Oracle 中用户也可以通过 LOCK TABLE IN ROW SHARE MODE 语句显式获得行共享锁定。行共享锁定并不防止对数据行进行更改操作，但是可以防止其他会话获取独占性数据表锁定。允许进行多个并发的行共享和行独占锁定，还允许进行数据表的共享或者采用共享行独占锁定。

- 行独锁定：通过一条 INSERT、UPDATE 或 DELETE 语句隐式获取，或者通过一条 LOCK TABLE IN ROW EXCLUSIVE MODE 语句显式获取。这种锁定可以防止其他会话获取一个共享锁定、共享行独占锁定或独占锁定。
- 表共享锁定：通过 LOCK TABLE IN SHARE MODE 语句显式获得。这种锁定可以防止其他会话获取行独占锁定（INSERT、UPDATE 或 DELETE），或者防止其他表共享行独占锁定或表独占锁定，但它允许在表中拥有多个行共享和表共享锁定。该锁定可以让会话具有对表事务级一致性访问，因为其他会话在用户提交或者回滚该事务并释放对该表的锁定之前不能更改这张被锁定的表。
- 表共享行独占锁定：通过 LOCK TABLE IN SHARE ROW EXCLUSIVE MODE 语句显式获得。这种锁定可以防止其他会话获取一个表共享、行独占或者表独占锁定，但允许其他行共享锁定。这种锁定类似于表共享锁定，只是一次只能对一张表放置一个表共享行独占锁定。如果 A 会话拥有该锁定，则 B 会话可以执行 SELECT FOR UPDATE 操作；但如果 B 会话试图更新选择的行，则需要等待。
- 表独占锁定：通过 LOCK TABLE IN EXCLUSIVE MODE 语句显式获得。这种锁定可以防止其他会话对该表的任何其他锁定。

11.1.4 事务隔离级别

尽管数据库为用户提供了锁的 DML 操作方式，但直接使用锁管理是非常麻烦的，因此数据库为用户提供了自动锁机制。只要用户指定会话的事务隔离级别，数据库就会分析事务中的 SQL 语句，然后自动为事务操作的数据资源添加适合的锁。此外，数据库还会维护这些锁，当一个资源上的锁数目太多时，自动进行锁升级以提高系统的运行性能，而这一过程对用户来说完全是透明的。

ANSI/ISO SQL 92 标准定义了 4 个等级的事务隔离级别，在相同的数据环境下，使用相同的输入，执行相同的工作，根据不同的隔离级别，可能导致不同的结果。不同的事务隔离级别能够解决的数据并发问题的能力是不同的，如表 11-1 所示。

表 11-1 事务隔离级别对并发问题的解决情况

隔离级别	脏 读	不可重复读	幻 象 读	第一类丢失更新	第二类丢失更新
READ UNCOMMITTED	允许	允许	允许	不允许	允许
READ COMMITTED	不允许	允许	允许	不允许	允许
REPEATABLE READ	不允许	不允许	允许	不允许	不允许
SERIALIZABLE	不允许	不允许	不允许	不允许	不允许

事务的隔离级别和数据库并发性是对立的。一般来说，使用 READ UNCOMMITTED 隔离级别的数据库拥有最高的并发性和吞吐量，而使用 SERIALIZABLE 隔离级别的数据库并发性最低。

SQL 92 定义 READ UNCOMMITTED 主要是为了提供非阻塞读的能力。Oracle 虽然也支持 READ UNCOMMITTED，但它不支持脏读；因为 Oracle 使用多版本机制彻底解决了在非阻塞读时读到脏数据的问题并保证读的一致性，所以，Oracle 的 READ COMMITTED 隔离级别就已经满足了 SQL 92 标准的 REPEATABLE READ 隔离级别。

SQL 92 推荐使用 REPEATABLE READ 以保证数据的读一致性，不过用户可以根据应用的需要选择适合的隔离等级。

11.1.5 JDBC 对事务的支持

并不是所有的数据库都支持事务，即使支持事务的数据库也并非支持所有的事务隔离级别。用户可以通过 Connection#getMetaData()方法获取 DatabaseMetaData 对象，并通过该对象的 supportsTransactions()、supportsTransactionIsolationLevel(int level)方法查看底层数据库的事务支持情况。

Connection 默认情况下是自动提交的，即每条执行的 SQL 语句都对应一个事务。为了将多条 SQL 语句当成一个事务执行，必须先通过 Connection#setAutoCommit(false)阻止 Connection 自动提交，并通过 Connection#setTransactionIsolation()设置事务的隔离级别。Connection 中定义了对应 SQL 92 标准 4 个事务隔离级别的常量。通过 Connection#commit()提交事务，通过 Connection#rollback()回滚事务。下面是典型的 JDBC 事务数据操作的代码，如代码清单 11-1 所示。

代码清单 11-1 JDBC 事务代码

```
Connection conn ;
try{
    conn = DriverManager.getConnection(); ① ← 获取数据连接
    conn.setAutoCommit(false); ② ← 关闭自动提交机制
    conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE); ③ ← 设置事务隔离级别
    Statement stmt = conn.createStatement();

    int rows = stmt.executeUpdate( "INSERT INTO t_topic VALUES(1,'tom') " );
    rows = stmt.executeUpdate( "UPDATE t_user set topic_nums = topic_nums +1 "+
        "WHERE user_id = 1" );

    conn.commit(); ④ ← 提交事务
} catch (Exception e){
    ...
    conn.rollback(); ⑤ ← 回滚事务
} finally{
    ...
}
```

在 JDBC 2.0 中，事务最终只能有两个操作：提交和回滚。但是，有些应用可能需要对事务进行更多的控制，而不是简单地提交或回滚。JDBC 3.0（Java 1.4 及以后的版本）引入了一个全新的保存点特性，Savepoint 接口允许用户将事务分割为多个阶段，用户可以指定回滚到事务的特定保存点，而并非像 JDBC 2.0 一样只能回滚到开始事务的点，如图 11-1 所示。

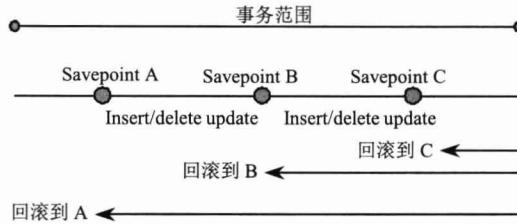


图 11-1 带 Savepoint 的事务

下面的代码使用了保存点功能，在发生特定问题时，回滚到指定的保存点，而非回滚整个事务，如代码清单 11-2 所示。

代码清单 11-2 使用保存点的事务代码

```
...
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate( "INSERT INTO t_topic VALUES(1,'tom')");
Savepoint svpt = conn.setSavepoint("savePoint1"); ① ← 设置一个保存点
rows = stmt.executeUpdate( "UPDATE t_user set topic_nums = topic_nums +1 "+
                           "WHERE user_id = 1");
...
conn.rollback(svpt); ② ← 回滚到①处的 savePoint1, ①之前的 SQL 操作, 在整个事务提交后依然提交, 但①到②之间的 SQL 操作被撤销了
...
conn.commit(); ③ ← 提交事务
...
```

并非所有数据库都支持保存点功能，用户可以通过 `DatabaseMetaData#supportsSavepoints()` 方法查看是否支持。

11.2 ThreadLocal 基础知识

在第 10 章中我们知道，Spring 通过各种模板类降低了开发者使用各种数据持久化技术的难度。这些模板类都是线程安全的，也就是说，多个 DAO 可以复用同一个模板实例而不会发生冲突。使用模板类访问底层数据，根据持久化技术的不同，模板类需要绑定数据连接或会话的资源。但这些资源本身是非线程安全的，也就是说它们不能在同一时刻被多个线程共享。虽然模板类通过资源池获取数据连接或会话，但资源池本身解决的是数据连接或会话的缓存问题，并非数据连接或会话的线程安全问题。

按照传统经验，如果某个对象是非线程安全的，在多线程环境下，对对象的访问必须采用 `synchronized` 进行线程同步。但模板类并未采用线程同步机制，因为线程同步会降低并发性，影响系统性能。此外，通过代码同步解决线程安全的挑战性很大，可能会增加几倍的实现难度。那么，模板类究竟仰仗何种“魔法神功”，可以在无须线程同步的情况下就化解线程安全的难题呢？答案就是 `ThreadLocal`！

`ThreadLocal` 在 Spring 中发挥着重要的作用，在管理 `request` 作用域的 Bean、事务管

理、任务调度、AOP 等模块中都出现了它的身影。要想了解 Spring 事务管理的底层技术，ThreadLocal 是必须攻克的“山头堡垒”。

11.2.1 ThreadLocal 是什么

早在 Java 1.2 版本中就提供了 `java.lang.ThreadLocal`。`ThreadLocal` 为解决多线程程序的并发问题提供了一种新的思路，使用这个工具类可以很简洁地编写出优美的多线程程序。线程局部变量并不是 Java 的新发明，很多语言（如 IBM XL、FORTRAN）在语法层面就提供了线程局部变量。在 Java 中没有提供语言级支持，而是以一种变通的方法，通过 `ThreadLocal` 的类提供支持。所以，在 Java 中编写线程局部变量的代码相对来说要笨拙一些，这也是为什么线程局部变量没有在 Java 开发者中得到很好普及的原因。

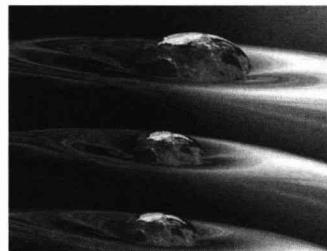
`ThreadLocal`，顾名思义，它不是一个线程，而是保存线程本地化对象的容器。当运行于多线程环境的某个对象使用 `ThreadLocal` 维护变量时，`ThreadLocal` 为每个使用该变量的线程分配一个独立的变量副本。所以每个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本。从线程的角度看，这个变量就像线程专有的本地变量，这也是类名中“Local”所要表达的意思。

`InheritableThreadLocal` 继承于 `ThreadLocal`，它自动为子线程复制一份从父线程那里继承而来的本地变量：在创建子线程时，子线程会接收所有可继承的线程本地变量的初始值。当必须将本地线程变量自动传送给所有创建的子线程时，应尽可能地使用 `InheritableThreadLocal`，而非 `ThreadLocal`。



轻松一刻

`ThreadLocal` 很像平行宇宙（Parallel Universes）的概念。平行宇宙来自量子力学，是指多元宇宙中所包含的各个宇宙，一个宇宙从某个宇宙中分离出来，与原宇宙平行存在着既相似又不同的其他宇宙。它们可能处于同一空间体系，但时间体系不同，就好像同在一条铁路线上疾驰的先后两列火车。2011 年，邓肯·琼斯执导的《源代码》就是一部关于平行宇宙的科幻影片。



11.2.2 ThreadLocal 的接口方法

`ThreadLocal` 类接口很简单，只有 4 个方法，先来了解一下。

- ❑ `void set(Object value)`: 设置当前线程的线程局部变量的值。
- ❑ `public Object get()`: 返回当前线程所对应的线程局部变量。

- `public void remove()`: 将当前线程局部变量的值删除，目的是为了减少内存的占用。该方法是 Java 5.0 新增的。需要指出的是，当线程结束后，对该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必需的操作，但它可以加快内存回收的速度。
- `protected Object initialValue()`: 返回该线程局部变量的初始值。该方法是一个 `protected` 的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第一次调用 `get()` 或 `set(Object)` 时才执行，并且仅执行一次。`ThreadLocal` 中的默认实现直接返回一个 `null`。

值得一提的是，在 Java 5.0 中，`ThreadLocal` 已经支持泛型，该类的类名已经变为 `ThreadLocal<T>`。API 方法也相应进行了调整，新版本的 API 方法分别是 `void set(T value)`、`T get()` 及 `T initialValue()`。

`ThreadLocal` 是如何做到为每个线程维护一份独立的变量副本呢？其实实现的思路很简单：在 `ThreadLocal` 类中有一个 `Map`，用于存储每个线程的变量副本，`Map` 中元素的键为线程对象，值为对应线程的变量副本。下面提供一个简单的实现版本，如代码清单 11-3 所示。

代码清单 11-3 SimpleThreadLocal

```
public class SimpleThreadLocal {
    private Map<Object, Object> valueMap = Collections.synchronizedMap(new HashMap<>());
    public void set(Object newValue) {
        valueMap.put(Thread.currentThread(), newValue); ①
    }
    public Object get() {
        Thread currentThread = Thread.currentThread();
        Object o = valueMap.get(currentThread); ②
        if (o == null && !valueMap.containsKey(currentThread)) { ③
            o = initialValue();
            valueMap.put(currentThread, o);
        }
        return o;
    }
    public void remove() {
        valueMap.remove(Thread.currentThread());
    }
    public Object initialValue() {
        return null;
    }
}
```

The code listing is annotated with the following comments:

- Annotation ①: "键为线程对象, 值为本线程的变量副本" (Key is thread object, value is variable copy for this thread) points to the line `valueMap.put(Thread.currentThread(), newValue);`.
- Annotation ②: "返回本线程对应的变量" (Return variable for this thread) points to the line `Object o = valueMap.get(currentThread);`.
- Annotation ③: "如果在 Map 中不存在, 则放到 Map 中保存起来" (If not in Map, put it in) points to the conditional block starting with `if (o == null && !valueMap.containsKey(currentThread)) {`.

虽然代码清单 11-3 中 `ThreadLocal` 的实现版本显得比较幼稚，但它和 JDK 所提供的 `ThreadLocal` 类在实现思路上是非常相近的。

11.2.3 一个 ThreadLocal 实例

下面通过一个具体的实例了解一下 `ThreadLocal` 的具体使用方法，如代码清单 11-4 所示。

代码清单 11-4 SequenceNumber

```

package com.smart.basic;

public class SequenceNumber {
    private static ThreadLocal<Integer> seqNum = new ThreadLocal<Integer>() {①
        public Integer initialValue() {
            return 0;
        }
    };

    public int getNextNum(){② ← 获取下一个序列值
        seqNum.set(seqNum.get()+1);
        return seqNum.get();
    }

    public static void main(String[ ] args)
    {
        SequenceNumber sn = new SequenceNumber();

        TestClient t1 = new TestClient(sn); ③
        TestClient t2 = new TestClient(sn);
        TestClient t3 = new TestClient(sn); ④ ← 3 个线程共享 sn, 各自产生序列号
        t1.start();
        t2.start();
        t3.start();
    }
}

private static class TestClient extends Thread
{
    private SequenceNumber sn;
    public TestClient(SequenceNumber sn) {
        this.sn = sn;
    }
    public void run()
    {
        for (int i = 0; i < 3; i++) {⑤ ← 每个线程打出 3 个序列值
            System.out.println("thread["+Thread.currentThread().getName()+
                "] sn["+sn.getNextNum()+"]");
        }
    }
}
}

```

可通过匿名内部类的方式定义 ThreadLocal 的子类，提供初始的变量值，如①处所示。TestClient 线程产生一组序列号，在③处，生成 3 个 TestClient，它们共享同一个 SequenceNumber 实例。运行以上代码，在控制台上输出以下信息：

```

thread[Thread-2] sn[1]
thread[Thread-0] sn[1]
thread[Thread-1] sn[1]
thread[Thread-2] sn[2]
thread[Thread-0] sn[2]
thread[Thread-1] sn[2]
thread[Thread-2] sn[3]

```

```
thread[Thread-0] sn[3]
thread[Thread-1] sn[3]
```

考查输出的结果信息，发现每个线程所产生的序号虽然都共享同一个 SequenceNumber 实例，但它们并没有相互干扰，而是各自产生独立的序列号，这是因为通过 ThreadLocal 为每个线程提供了单独的副本。

11.2.4 与 Thread 同步机制的比较

ThreadLocal 和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。那么，ThreadLocal 和线程同步机制相比有什么优势呢？

在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序缜密地分析什么时候对变量进行读/写、什么时候需要锁定某个对象、什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

而 ThreadLocal 从另一个角度来解决多线程的并发访问。ThreadLocal 为每个线程提供了一个独立的变量副本，从而隔离了多个线程对访问数据的冲突。因为每个线程都拥有自己的变量副本，因而也就没有必要对该变量进行同步。ThreadLocal 提供了线程安全的对象封装，在编写多线程代码时，可以把不安全的变量封装进 ThreadLocal。

由于 ThreadLocal 中可以持有任何类型的对象，低版本 JDK 所提供的 get()方法返回的是 Object 对象，需要强制类型转换。但 Java 5.0 通过泛型很好地解决了这个问题，在一定程度上简化了 ThreadLocal 的使用，代码清单 11-2 就使用了 Java 5.0 新的 ThreadLocal<T>版本。

概括而言，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式：访问串行化，对象共享化；而 ThreadLocal 采用了“以空间换时间”的方式：访问并行化，对象独享化。前者仅提供一份变量，让不同的线程排队访问；而后者为每个线程都提供了一份变量，因此可以同时访问而互不影响。

11.2.5 Spring 使用 ThreadLocal 解决线程安全问题

我们知道，在一般情况下，只有无状态的 Bean 才可以在多线程环境下共享。在 Spring 中，绝大部分 Bean 都可以声明为 singleton 作用域。正是因为 Spring 对一些 Bean（如 RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder 等）中非线程安全的“状态性对象”采用 ThreadLocal 进行封装，让它们也成为线程安全的“状态性对象”，因此，有状态的 Bean 就能够以 singleton 的方式在多线程中正常工作。

一般的 Web 应用划分为展现层、服务层和持久层 3 个层次，在不同的层中编写对应的逻辑，下层通过接口向上层开放功能调用。在一般情况下，从接收请求到返回响应所经过的所有程序调用都同属于一个线程，如图 11-2 所示。

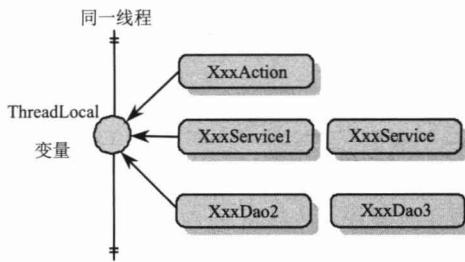


图 11-2 同一线程贯通三层

这样用户就可以根据需要，将一些非线程安全的变量以 ThreadLocal 存放，在同一次请求响应的调用线程中，所有对象所访问的同一 ThreadLocal 变量都是当前线程所绑定的。

下面的实例能够体现 Spring 对有状态 Bean 的改造思路，如代码清单 11-5 所示。

代码清单 11-5 TopicDao：非线程安全

```
public class TopicDao {
    //①一个非线程安全变量
    private Connection conn;

    public void addTopic() {
        //②引用非线程安全变量
        Statement stat = conn.createStatement();
        ...
    }
}
```

由于①处的 conn 是非线程安全的成员变量，因此 addTopic()方法也是非线程安全的，所以必须在使用时创建一个新的 TopicDao 实例（非 singleton）。下面使用 ThreadLocal 对 conn 这个非线程安全的“状态”进行改造，如代码清单 11-6 所示。

代码清单 11-6 TopicDao：线程安全

```
import java.sql.Connection;
import java.sql.Statement;
public class TopicDao {
    //①使用ThreadLocal保存Connection变量
    private static ThreadLocal<Connection> connThreadLocal = new
    ThreadLocal<Connection>();
    public static Connection getConnection(){

        //②如果connThreadLocal没有本线程对应的Connection，则创建一个新的Connection，
        //并将其保存到线程本地变量中
        if (connThreadLocal.get() == null) {
            Connection conn = ConnectionManager.getConnection();
            connThreadLocal.set(conn);
            return conn;
        }else{
    }
```

```

    //③直接返回线程本地变量
    return connThreadLocal.get();
}
}

public void addTopic() {
    //④从ThreadLocal中获取线程对应的Connection
    Statement stat = getConnection().createStatement();
}
}

```

不同的线程在使用 TopicDao 时，先判断 connThreadLocal.get() 是否为 null，如果为 null，则说明当前线程还没有对应的 Connection 对象，这时创建一个 Connection 对象并添加到本地线程变量中；如果不为 null，则说明当前线程已经拥有了 Connection 对象，直接使用就可以了。这样就保证了不同的线程使用自己独立的 Connection，而不会使用其他线程的 Connection。因此，这个 TopicDao 就可以做成 singleton 的 Bean 了。

当然，这个例子本身很粗糙，将 Connection 的 ThreadLocal 直接放在 DAO 中只能做到本 DAO 的多个方法共享 Connection 时不发生线程安全问题，但无法和其他 DAO 共用同一个 Connection。要做到同一事务多 DAO 共享同一个 Connection，必须在一个共同的外部类使用 ThreadLocal 保存 Connection。但这个实例基本上说明了 Spring 对所有状态类线程安全化的解决思路。在本章后面的内容中，将详细说明 Spring 如何通过 ThreadLocal 解决事务管理的问题。

11.3 Spring 对事务管理的支持

Spring 为事务管理提供了一致的编程模板，在高层次建立了统一的事务抽象。也就是说，不管是选择 Spring JDBC、Hibernate、JPA 还是选择 MyBatis，Spring 都可以让用户用统一的编程模型进行事务管理。

像 Spring DAO 为不同的持久化实现提供了模板类一样，Spring 事务管理继承了这一风格，也提供了事务模板类 TransactionTemplate。通过 TransactionTemplate 并配合使用事务回调 TransactionCallback 指定具体的持久化操作，就可以通过编程方式实现事务管理，而无须关注资源获取、复用、释放、事务同步和异常处理等操作。

Spring 事务管理的亮点在于声明式事务管理。Spring 允许通过声明方式，在 IoC 配置中指定事务的边界和事务属性，Spring 自动在指定的事务边界上应用事务属性。声明式事务是 EJB 焰赫一时的技术，Spring 让这种技术平民化，甚至可以说，Spring 的声明式事务比 EJB 的更为强大。

EJB 事务建立在 JTA 的基础上，而 JTA 又必须通过 JNDI 获取。这意味着不管用户的应用是跨数据源的应用，还是单数据源的应用，EJB 都要求使用全局事务的方式加以处理，即基于 EJB 的应用无法脱离应用服务器所提供的容器环境。这种不加区分一概而论的做法无异于杀鸡杀牛都用一把刀。

Spring 深刻地认识到，大部分应用都是基于单数据源的，只有为数不多的应用需要使用多数据源的 JTA 事务。因此，在单数据源的情况下，Spring 直接使用底层的数据源管理事务。只有在面对多数据源的应用时，Spring 才寻求 Java EE 应用服务器的支持，通过引用应用服务器中的 JNDI 资源完成 JTA 事务。Spring 让人印象深刻的地方在于不管用户使用何种持久化技术，也不管用户是否使用了 JTA 事务，都可以采用相同的事务管理模型。这种统一的处理方式所带来的好处是不可估量的，用户完全可以抛开事务管理的问题编写程序，并在 Spring 中通过配置完成事务的管理工作。

11.3.1 事务管理关键抽象

在 Spring 事务管理 SPI (Service Provider Interface) 的抽象层主要包括 3 个接口，分别是 PlatformTransactionManager、TransactionDefinition 和 TransactionStatus，它们位于 org.springframework.transaction 包中。通过图 11-3 可以描述这三者的关系。

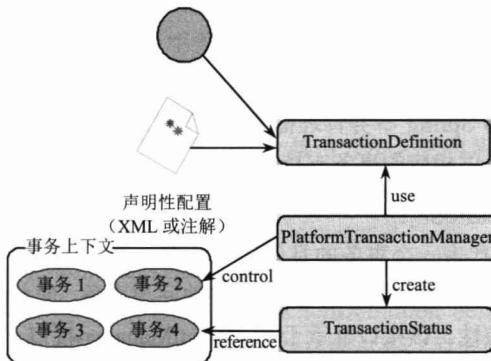


图 11-3 Spring 事务管理 SPI 抽象

TransactionDefinition 用于描述事务的隔离级别、超时时间、是否为只读事务和事务传播规则等控制事务具体行为的事务属性，这些事务属性可以通过 XML 配置或注解描述提供，也可以通过手工编程的方式设置。

PlatformTransactionManager 根据 TransactionDefinition 提供的事务属性配置信息创建事务，并用 TransactionStatus 描述这个激活事务的状态。下面分别来了解这些 SPI 接口内部的组成。

1. TransactionDefinition

TransactionDefinition 定义了 Spring 兼容的事务属性，这些属性对事务管理控制的若干方面进行配置。

- **事务隔离：**当前事务和其他事务的隔离程度。在 TransactionDefinition 接口中，定义了和 java.sql.Connection 接口中同名的 4 个隔离级别：ISOLATION_READ_UNCOMMITTED、ISOLATION_READ_COMMITTED、ISOLATION_REPEATABLE_

READ 和 ISOLATION_SERIALIZABLE（实际上它直接使用 Connection 的同名常量进行赋值），这些常量分别对应于 11.1.4 节所描述的隔离级别。此外，`TransactionDefinition` 还定义了一个默认的隔离级别——ISOLATION_DEFAULT，它表示使用底层数据库的默认隔离级别。

- **事务传播：**通常在一个事务中执行的所有代码都会运行于同一事务上下文中。但是 Spring 也提供了几个可选的事务传播类型，例如，简单地参与到现有的事务中，或者挂起当前的事务，创建一个新的事务。在 Spring 事务管理中，传播行为是一个重要的概念，将单独在 11.3.4 节详细讲述。Spring 提供了 EJB CMT 所支持的事务传播类型。
- **事务超时：**事务在超时前能运行多久，超过时间后，事务被回滚。有些事务管理器不支持事务过期的功能，这时，如果设置 TIMEOUT_DEFAULT 之外的其他值，则将抛出异常。
- **只读状态：**只读事务不修改任何数据，资源事务管理者可以针对可读事务应用一些优化措施，提高运行性能。只读事务在某些情况下（如使用 Hibernate 时）是一种非常有用的优化，试图在只读事务中更改数据将引发异常。

Spring 允许通过 XML 或注解元数据的方式为一个有事务要求的服务类方法配置事务属性，这些信息作为 Spring 事务管理框架的“输入”，Spring 将自动按事务属性信息的指示，为目标方法提供相应的事务支持。

2. TransactionStatus

`TransactionStatus` 代表一个事务的具体运行状态。事务管理器可以通过该接口获取事务运行期的状态信息，也可以通过该接口间接地回滚事务，它相比于在抛出异常时回滚事务的方式更具可控性。该接口继承于 `SavepointManager` 接口，`SavepointManager` 接口基于 JDBC 3.0 保存点的分段事务控制能力提供了嵌套事务的机制。

`SavepointManager` 接口拥有以下几个方法。

- `Object createSavepoint()`: 创建一个保存点对象，以便在后面可以利用 `rollbackToSavepoint(Object savepoint)` 方法使事务回滚到特定的保存点上，也可以通过 `releaseSavepoint()` 方法释放一个已经不用的保存点。
- `void rollbackToSavepoint(Object savepoint)`: 将事务回滚到特定的保存点上，被回滚的保存点将自动释放。
- `void releaseSavepoint(Object savepoint)`: 释放一个保存点。如果事务提交，则所有的保存点会被自动释放，无须手工清除。

这 3 个方法在底层的资源不支持保存点时，都将抛出 `NestedTransactionNotSupportedException` 异常。

`TransactionStatus` 扩展了 `SavepointManager` 接口，并提供了以下几个方法。

- `boolean hasSavepoint()`: 判断当前事务是否在内部创建了一个保存点，该保存点是为了支持 Spring 的嵌套事务而创建的。

- boolean isNewTransaction(): 判断当前事务是否是一个新的事务, 如果返回 false, 则表示当前事务是一个已经存在的事务, 或者当前操作未运行在事务环境中。
- boolean isCompleted(): 判断当前事务是否已经结束 (已经提交或回滚)。
- boolean isRollbackOnly(): 判断当前事务是否已经被标识为 rollback-only。
- void setRollbackOnly(): 将当前事务设置为 rollback-only。通过该标识通知事务管理器只能将事务回滚, 事务管理器将通过显式调用回滚命令或抛出异常的方式回滚事务。

3. PlatformTransactionManager

通过 JDBC 的事务管理知识可以知道, 事务只能被提交或回滚 (或回滚到某个保存点后提交), Spring 高层事务抽象接口 org.springframework.transaction.PlatformTransactionManager 很好地描述了事务管理这个概念, 如代码清单 11-7 所示。

代码清单 11-7 PlatformTransactionManager

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

PlatformTransactionManager 只定义了 3 个接口方法, 它们是 SPI (Service Provider Interface) 高层次的接口方法。这些访问都没有和 JNDI 绑定在一起, 可以像 Spring 容器中普通的 Bean 一样对待 PlatformTransactionManager 实现者。

下面来了解一下 PlatformTransactionManager 接口方法的功能。

- TransactionStatus getTransaction(TransactionDefinition definition): 该方法根据事务定义信息从事务环境中返回一个已存在的事务, 或者创建一个新的事务, 并用 TransactionStatus 描述这个事务的状态。
- commit(TransactionStatus status): 根据事务的状态提交事务。如果事务状态已经被标识为 rollback-only, 则该方法将执行一个回滚事务的操作。
- rollback(TransactionStatus status): 将事务回滚。当 commit()方法抛出异常时, rollback()方法会被隐式调用。

11.3.2 Spring 的事务管理器实现类

Spring 将事务管理委托给底层具体的持久化实现框架来完成。因此, Spring 为不同的持久化框架提供了 PlatformTransactionManager 接口的实现类, 如表 11-2 所示。

表 11-2 不同持久化技术对应的事务管理器实现类

事 务	说 明
org.springframework.orm.jpa.JpaTransactionManager	使用 JPA 进行持久化时, 使用该事务管理器

续表

事 务	说 明
org.springframework.orm.hibernateX.HibernateTransactionManager	使用 Hibernate X.0 (X 可为 3,4,5, 下同) 版本进行持久化时, 使用该事务管理器
org.springframework.jdbc.datasource.DataSource TransactionManager	使用 Spring JDBC 或 MyBatis 等基于 DataSource 数据源的持久化技术时, 使用该事务管理器
org.springframework.orm.jdo.JdoTransactionManager	使用 JDO 进行持久化时, 使用该事务管理器
org.springframework.transaction.jta.JtaTransactionManager	具有多个数据源的全局事务使用该事务管理器 (不管采用何种持久化技术)

这些事务管理器都是对特定事务实现框架的代理, 这样就可以通过 Spring 所提交的高级抽象对不同种类的事务实现使用相同的方式进行管理, 而不用关心具体的实现。

要实现事务管理, 首先要在 Spring 中配置好相应的事务管理器, 为事务管理器指定数据资源及一些其他事务管理控制属性。下面来看一下几个常见的事务管理器的配置。

1. Spring JDBC 和 MyBatis

如果使用 Spring JDBC 或 MyBatis, 由于它们都基于数据源的 Connection 访问数据库, 所以可以使用 DataSourceTransactionManager, 只要在 Spring 中进行以下配置就可以了, 如代码清单 11-8 所示。

代码清单 11-8 基于数据源的数据管理器

```
...
<bean id="dataSource" ① 配置一个数据源
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${jdbc.driverClassName}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}"/>
<bean id="transactionManager" ② 基于数据源的事务管理器
    class="org.springframework.jdbc.datasource. DataSourceTransactionManager"
    p:dataSource-ref="dataSource" ③ 引用数据源
/>
</bean>
```

在幕后, DataSourceTransactionManager 使用 DataSource 的 Connection 的 commit()、rollback() 等方法管理事务。

2. JPA

JPA 通过 javax.persistence.EntityTransaction 管理 JPA 的事务, EntityTransaction 对象可以通过 javax.persistence.EntityManager#getTransaction() 方法获得, 而 EntityManager 又通过一个工厂类方法 javax.persistence.EntityManagerFactory#createEntityManager() 获取。

在底层, JPA 依然通过 JDBC 的 Connection 的事务方法完成最终的控制。因此, 要配置一个 JPA 事务管理器, 必须先提供一个 DataSource, 然后配置一个 EntityManagerFactory, 最后才配置 JpaTransactionManager, 如代码清单 11-9 所示。

代码清单 11-9 JPA 事务管理器配置

```

...
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
    p:dataSource-ref="dataSource"/>① ← 指定一个数据源
...
</bean>
<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager"
    p:entityManagerFactory-ref="entityManagerFactory"/>② ← 指定实体管理器

```

3. Hibernate

Spring 4.0 已取消了对 Hibernate 3.6 之前版本的支持，并全面支持 Hibernate 5.0；因此，只为 Hibernate 3.6+ 提供了事务管理器。下面以 Hibernate 4.0 为例，如代码清单 11-10 所示。

代码清单 11-10 Hibernate 4.0 事务管理器配置

```

...
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean" ) ← 指定一个数据源
    p:dataSource-ref="dataSource" ① ← 指定 Hibernate
    p:mappingResources="classpath:bbtForum.hbm.xml" ② ← 配置文件

    <property name="hibernateProperties" > ③ ← Hibernate 其他配置属性
        <props>
            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.generate_statistics">true</prop>
        </props>
    </property>
</bean>
<bean id="transactionManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager"
    p:sessionFactory-ref="sessionFactory"/>④ ← 注入会话工厂

```

大部分 ORM 框架都拥有自己事务管理的 API，它们对 DataSource 和 Connection 进行了封装。Hibernate 使用 org.hibernate.Session 封装 Connection，所以需要一个能够创建 Session 的 SessionFactory（更多内容请参见第 14 章）。

4. JTA

如果希望在 Java EE 容器里使用 JTA，则将通过 JNDI 和 Spring 的 JtaTransactionManager 获取一个容器管理的 DataSource。JtaTransactionManager 不需要知道 DataSource 和其他特定的资源，因为它引用容器提供的全局事务管理，如代码清单 11-11 所示。

代码清单 11-11 JTA 事务管理器

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"

```

```

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-4.0.xsd">

<jee:jndi-lookup id="accountDs" jndi-name="java:comp/env/jdbc/account"/>①
<jee:jndi-lookup id="orderDs" jndi-name="java:comp/env/jdbc/order"/>
    ↗ 通过 jee 命名空间获取
    ↗ Java EE 应用服务器容
    ↗ 器中的数据源

<bean id="transactionManager"           指定 JTA 事务管理器
      class="org.springframework.transaction.jta.JtaTransactionManager"/>②
</beans>

```

这里使用了 jee 命名空间从 Java EE 容器中返回 JNDI 管理的资源。①处的配置相当于：

```

<bean id="simple"
  class="org.springframework.jndi.JndiObjectFactoryBean"
  p:jndiName = "java:comp/env/jdbc/account "/>

```

显然，使用 jee 命名空间的配置更加简洁。jee 命名空间还可以获取 EJB 本地或远程的无状态 Session Bean（更多信息参见 Spring 参考手册的附录）。

11.3.3 事务同步管理器

Spring 将 JDBC 的 Connection、Hibernate 的 Session 等访问数据库的连接或会话对象统称为资源，这些资源在同一时刻是不能多线程共享的。为了让 DAO、Service 类可能做到 singleton，Spring 的事务同步管理器类 org.springframework.transaction.support.Transaction SynchronizationManager 使用 ThreadLocal 为不同事务线程提供了独立的资源副本，同时维护事务配置的属性和运行状态信息。事务同步管理器是 Spring 事务管理的基石，不管用户使用的是编程式事务管理，还是声明式事务管理，都离不开事务同步管理器。

Spring 框架为不同的持久化技术提供了一套从 TransactionSynchronizationManager 中获取对应线程绑定资源的工具类，如表 11-3 所示。

表 11-3 线程绑定资源获取工具

持久化技术	线程绑定资源获取工具
Spring JDBC 或 MyBatis	org.springframework.jdbc.datasource.DataSourceUtils
Hibernate X.0	org.springframework.orm.hibernateX.SessionFactoryUtils
JPA	org.springframework.orm.jpa.EntityManagerFactoryUtils
JDO	org.springframework.orm.jdo.PersistenceManagerFactoryUtils

这些工具类都提供了静态的方法，通过这些方法可以获取和当前线程绑定的资源，如 DataSourceUtils.getConnection(DataSource dataSource) 方法可以从指定的数据源中获取和当前线程绑定的 Connection，而 Hibernate 的 SessionFactoryUtils.getSession(SessionFactory

sessionFactory, boolean allowCreate)方法则可以从指定的 SessionFactory 中获取和当前线程绑定的 Session。

当需要脱离模板类，手工操作底层持久化技术的原生 API 时，就需要通过这些工具类获取线程绑定的资源，而不应该直接从 DataSource 或 SessionFactory 中获取。因为后者不能获得与本线程相关的资源，因此无法让数据操作参与到与本线程相关的事务环境中。

这些工具类还有另外一个重要的用途：将特定异常转换为 Spring 的 DAO 异常，已经在第 10 章的表 10-2 中对此进行了说明。

Spring 为不同的持久化技术提供了模板类，模板类在内部通过资源获取工具类间接访问 TransactionSynchronizationManager 中的线程绑定资源。所以，如果 DAO 使用模板类进行持久化操作，这些 DAO 就可以配置成 singleton。如果不使用模板类，也可以直接通过资源获取工具类访问线程相关的资源。

下面，让我们揭开 TransactionSynchronizationManager 的层层面纱，探寻其中的奥秘。

```
public abstract class TransactionSynchronizationManager {
    //①用于保存每个事务线程对应的Connection或Session等类型的资源
    private static final ThreadLocal resources = new ThreadLocal();

    //②用于保存每个事务线程对应事务的名称
    private static final ThreadLocal currentTransactionName = new ThreadLocal();

    //③用于保存每个事务线程对应事务的read-only状态
    private static final ThreadLocal currentTransactionReadOnly = new ThreadLocal();

    //④用于保存每个事务线程对应事务的隔离级别
    private static final ThreadLocal currentTransactionIsolationLevel = new ThreadLocal();
    ThreadLocal();

    //⑤用于保存每个事务线程对应事务的激活态
    private static final ThreadLocal actualTransactionActive = new ThreadLocal();
    ...
}
```

TransactionSynchronizationManager 将 DAO、Service 类中影响线程安全的所有“状态”统一抽取到该类中，并用 ThreadLocal 进行替换，从此 DAO（必须是基于模板类或资源获取工具类创建的 DAO）和 Service（必须采用 Spring 事务管理机制）摘掉了非线程安全的帽子，完成了脱胎换骨式的身份转变。

11.3.4 事务传播行为

当我们调用一个基于 Spring 的 Service 接口方法（如 UserService#addUser()）时，它将运行于 Spring 管理的事务环境中，Service 接口方法可能会在内部调用其他的 Service 接口方法以共同完成一个完整的业务操作，因此就会产生服务接口方法嵌套调用的情

况，Spring 通过事务传播行为控制当前的事务如何传播到被嵌套调用的目标服务接口方法中。事务传播是 Spring 进行事务管理的重要概念，其重要性怎么强调都不为过。但是事务传播行为也是被误解最多的地方，本节将详细分析不同事务传播行为的表现形式，掌握它们之间的区别。

Spring 在 TransactionDefinition 接口中规定了 7 种类型的事务传播行为，它们规定了事务方法和事务方法发生嵌套调用时事务如何进行传播，如表 11-4 所示。

表 11-4 事务传播行为类型

事务传播行为类型	说 明
PROPAGATION_REQUIRED	如果当前没有事务，则新建一个事务；如果已经存在一个事务，则加入到这个事务中。这是最常见的选择
PROPAGATION_SUPPORTS	支持当前事务。如果当前没有事务，则以非事务方式执行
PROPAGATION_MANDATORY	使用当前的事务。如果当前没有事务，则抛出异常
PROPAGATIONQUIRES_NEW	新建事务。如果当前存在事务，则把当前事务挂起
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作。如果当前存在事务，则把当前事务挂起
PROPAGATION_NEVER	以非事务方式执行。如果当前存在事务，则抛出异常
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行；如果当前没有事务，则执行与 PROPAGATION_REQUIRED 类似的操作

在使用 PROPAGATION_NESTED 时，底层的数据源必须基于 JDBC 3.0，并且实现者需要支持保存点事务机制。

11.4 编程式的事務管理

在实际应用中，很少需要通过编程来进行事务管理。即便如此，Spring 还是为编程式事务管理提供了模板类 org.springframework.transaction.support.TransactionTemplate，以满足一些特殊场合的需要。

TransactionTemplate 和那些持久化模板类一样是线程安全的，因此，可以在多个业务类中共享 TransactionTemplate 实例进行事务管理。TransactionTemplate 拥有多个设置事务属性的方法，如 setReadOnly(boolean readOnly)、setIsolationLevel(int isolationLevel)等。

虽然很少需要手工编写事务管理的程序，但作为深入认识 Spring 事务管理的一种方式，了解一下编程式事务也不是坏事。TransactionTemplate 有两个主要的方法：

- void setTransactionManager(PlatformTransactionManager transactionManager): 设置事务管理器。
- Object execute(TransactionCallback action): 在 TransactionCallback 回调接口中定义需要以事务方式组织的数据访问逻辑。

TransactionCallback 接口只有一个方法：Object doInTransaction(TransactionStatus status)。如果操作不会返回结果，则可以使用 TransactionCallback 的子接口 TransactionCallbackWithoutResult，如代码清单 11-12 所示。

代码清单 11-12 采用编程式的事务管理

```

...
@Service
public class ForumService1 {
    public ForumDao forumDao;
    public TransactionTemplate template;

    @Autowired          ① ← 通过 AOP 主动注入
    public void setTemplate(TransactionTemplate template) {
        this.template = template;
    }

    public void addForum(final Forum forum) {
        template.execute(new TransactionCallbackWithoutResult() {
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                forumDao.addForum(forum); ② ← 需要在事务环境中执行的代码
            }
        });
    }
}
...
}

```

由于 Spring 事务管理基于 TransactionSynchronizationManager 进行工作，所以，如果在回调接口方法中需要显式访问底层数据连接，则必须通过资源获取工具类得到线程绑定的数据连接。这是 Spring 事务管理的底层协议，不容违反。如果 ForumDao 是基于 Spring 提供的模板类构建的，由于模板类已经在内部使用了资源获取工具类获取数据连接，所以用户就不必关心底层数据连接的获取问题。

11.5 使用 XML 配置声明式事务

大多数 Spring 用户选择声明式事务管理的功能，这种方式对代码的侵入性最小，可以让事务管理代码完全从业务代码中移除，非常符合非侵入式轻量级容器的理念。

Spring 的声明式事务管理是通过 Spring AOP 实现的，通过事务的声明性信息，Spring 负责将事务管理增强逻辑动态织入业务方法的相应连接点中。这些逻辑包括获取线程绑定资源、开始事务、提交/回滚事务、进行异常转换和处理等工作。

Spring 提供了和 EJB CMT 相似的声明式事务管理，这不但体现在二者的最终执行效果上，还体现在二者事务声明的语法上。但二者也存在明显的不同，下面通过表 11-5 对二者进行比较。

表 11-5 Spring 和 EJB CMT 声明事务的不同

比较项	EJB CMT	Spring
是否绑定 JTA	绑定在 JTA 上，即使单数据源也是如此。所以 EJB 不能脱离容器运行	可以在任何环境下使用，包括直接在 Spring 中声明的数据源或在应用服务器 JNDI 中的 JTA 数据源
持久化技术支持	采用非开放的 EJB 自制持久化技术	通过少量配置即可和 JDBC、JDO、Hibernate 等持久化技术一起工作
目标类要求	必须是实现特定接口的特殊类	可以是任何 POJO，不过在内部必须使用资源获取工具类操作数据连接或会话。如果 DAO 使用模板类进行构建，则这种要求将自动得到满足
回滚规则	没有提供	提供声明式的回滚规则
开放性控制	使用 EJB CMT，除了使用 setRollbackOnly() 方法，没有办法影响容器的事务管理	Spring 允许用户通过 AOP 定制事务行为。例如，如果需要，用户可以在事务回滚中插入定制的行为，也可以增加任意的增强，就和任何 AOP 的增强一样
分布式事务	支持分布式事务。一般应用并不需要使用这样的功能	Spring 不直接支持高端应用服务器所提供的跨越远程调用的事务上下文传播。此时，可以通过 Spring 的 Java EE 服务集成来提供。此外，在 Spring 中集成 JOTM 后，Spring 也可以提供 JTA 事务的功能

回滚规则的概念比较重要，它使用户能够指定什么样的异常导致自动回滚、什么样的异常不影响事务提交，这些规则可以在配置文件中通过声明的方式指定；同时，仍然可以通过调用 TransactionStatus#setRollbackOnly() 方法编程式地回滚当前事务。通常我们定义一条规则，如声明 MyApplicationException 必须总是导致事务回滚。这种方式带来了显著的好处，它使用户的业务对象不必依赖于事务设施。典型的例子是用户不必在代码中导入 Spring API、事务代码等。

对 EJB 来说，默认的行为是 EJB 容器在遇到系统异常（通常指运行时异常）时自动回滚当前事务。EJB 遇到应用异常（通常指检查型异常）时并不会自动回滚。默认情况下，Spring 的声明式事务管理和 EJB 的相同（只在遇到运行期异常时自动回滚）。



轻松一刻

在讲解 Spring 时，经常会涉及的一个问题是“框架的侵入性”。从广义上讲，“侵入性”就是对应用编程行为的影响，从这个意义上说，任何框架都逃脱不了“侵入性”的控诉。不过反过来一想：框架不对应用产生任何影响，我们还需要框架做什么呢？关键问题是这种影响是怎样施加的。EJB 以一种强权政治的方式强力扭转程序员的固有习惯，让应用开发人员的日子很难过；而 Spring 则以一种春风化雨、喜闻乐见的方式引入它的“侵入性”——这不禁让我们想起了那则著名的“太阳和风”的寓言。



11.5.1 一个将被实施事务增强的服务接口

BbtForum 拥有 4 个方法，我们希望 addTopic() 和 updateForum() 方法拥有写事务的能力，而其他两个方法只需要拥有读事务的能力就可以了。以下是 BbtForum 的实现代码，如代码清单 11-13 所示。

代码清单 11-13 BbtForum

```
package com.smart.service;
...
@Service
@Transactional
public class BbtForum {
    public ForumDao forumDao;
    public TopicDao topicDao;
    public PostDao postDao;
    public void addTopic(Topic topic) {
        topicDao.addTopic(topic);
        postDao.addPost(topic.getPost());
    }
    public Forum getForum(int forumId) {
        return forumDao.getForum(forumId);
    }
    public void updateForum(Forum forum) {
        forumDao.updateForum(forum);
    }
    public int getForumNum() {
        return forumDao.getForumNum();
    }
    ...
}
```

BbtForum 是一个 POJO，只是简单地使用持久层的多个 DAO 类，通过它们的协作实现业务功能。在这里，我们看不到任何事务操作的代码，所以如果直接使用 BbtForum，那么这些方法都将以无事务的方式运行。现在，我们的任务是通过 Spring 声明式事务配置让这些业务方法拥有适合的事务功能。

11.5.2 使用原始的 TransactionProxyFactoryBean

在 Spring 的早期版本中，用户必须通过 TransactionProxyFactoryBean 代理类对需要事务管理的业务类进行代理，以便实施事务功能的增强。在 Spring 2.0 后，由于可以通过 aop/tx 命名空间声明事务，因此通过该代理类实施声明式事务的方式已经不被推荐。

1. 声明式事务配置

从循序渐进的学习角度来看，了解 TransactionProxyFactoryBean 有助于我们更直观地理解 Spring 实施声明式事务的内部工作原理。所以我们不妨把使用这个代理类作为学习 Spring 事务管理的起始站，后面的章节会陆续介绍 Spring 的配置方式，如代码清单 11-14 所示。

代码清单 11-14 使用 TransactionProxyFactoryBean 配置

```

<!--①引入DAO和DataSource的配置文件-->
<import resource="classpath:applicationContext-dao.xml" />

<!--②声明事务管理器-->
<bean id="txManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<!--③需要实施事务增强的目标业务Bean -->
<bean id="bbtForumTarget"
      class="package com.smart.service.BbtForum"
      p:forumDao-ref="forumDao"
      p:topicDao-ref="topicDao"
      p:postDao-ref="postDao"/>

<bean id="bbtForum" ④———— 使用事务代理工厂类为目标业务 Bean 提供事务增强
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
      p:transactionManager-ref="txManager" ④-1———— 指定事务管理器
      p:target-ref="bbtForumTarget"> ④-2———— 指定目标业务 Bean
      <property name="transactionAttributes">④-3———— 事务属性配置
        <props>
          <prop key="get*>">PROPAGATION_REQUIRED,readOnly</prop>④-4———— 只读事务
          <prop key="*">>PROPAGATION_REQUIRED</prop>④-5———— 可写事务
        </props>
      </property>
</bean>

```

当然，必须首先配置好 DAO Bean、数据源等基础设施，这些信息统一在 applicationContext-dao.xml 配置文件中定义，如①处所示（读者可以参看本书配套网盘中的具体配置）。在这个示例中，我们使用 Spring JDBC 的持久化技术，所以它的资源池是 JDBC 数据源，而事务管理器是 DataSourceTransactionManager。如果使用其他持久化技术，则需要对基础设施和事务管理器进行相应的调整。

按照约定的习惯，需要事务增强的业务类一般将 id 取名为 xxxTarget，如③处所示，这可以在字面上表示该 Bean 是要被代理的目标 Bean。

通过 TransactionProxyFactoryBean 对业务类进行代理，织入事务增强功能，如④处所示。首先，需要为该代理类指定事务管理器，这些事务管理器实现了 PlatformTransactionManager 接口；其次，通过 target 属性指定需要代理的目标 Bean；最后，为业务 Bean 的不同方法配置事务属性。Spring 允许通过键值配置业务方法的事务属性信息，键可以使用通配符，如 get*代表目标类中所有以 get 为前缀的方法，它匹配 BbtForum 的 getForum(int forumId) 和 getForumNum() 方法；而 key="*" 匹配 BbtForum 接口的所有方法。