1 Spring 基础

在本部分，将介绍 Spring 框架的两个核心特性：反向控制（IoC）和面向切面编程（AOP）。

- 首先，简单介绍 Spring 中 IoC 和 AOP；
- 其次，装配 Bean，介绍如何利用 IoC 实现系统对象间的松耦合关联，如何使用 XML 在 Spring 容器中定义系统对象，装配其依赖类。
- 创建切面，介绍 Spring 的 AOP 把系统级服务（如安全和监控）从被服务对象中解耦出来

1.1 Spring 简介

1.1.1 Spring 特点

Spring 是一个轻量级的 IoC 和 AOP 容器框架。

- 轻量级：从大小及系统开支上说。且 Spring 是非侵入式的（基于 Spring 开发的系统中对象一般不依赖于 Spring 的类）
- 反向控制：使用 IoC 对象是被动接收依赖类而不是主动去找（容器在实例化对象时主动将其依赖类注入给它）
- 面向切面：将业务逻辑从系统服务中分离，实现内聚开发。系统对象只做其该做的业务逻辑不负责其他系统问题（如日志和事务支持）。
- 容器：包含且管理系统对象的生命周期和配置，通过配置设定 Bean 是单一实例还是每次请求产生一个实例，并设定 Bean 之间的关联关系
- 框架：使用简单组件配置组合成一个复杂的系统，系统中的对象是通过 XML 文件配置组合起来的，且 Spring 提供了很多基础功能（事务管理、持久层集成等）

1.1.2 Spring 模块

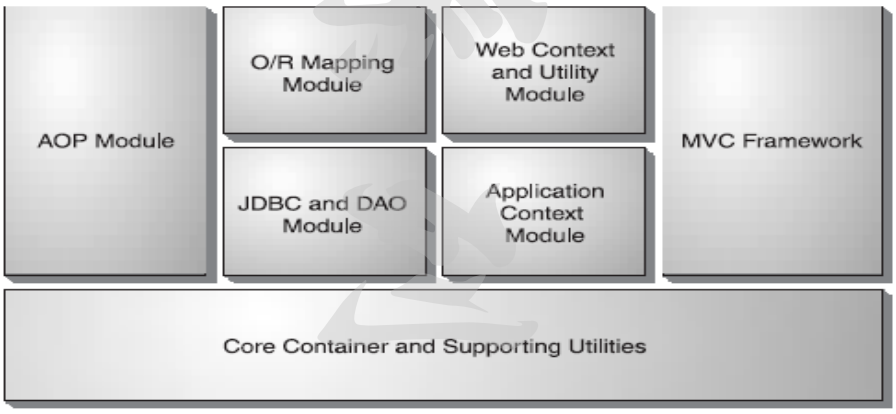


图 1-1

Spring 框架由 7 个模块组成（如图 1-1）：

- 核心容器：提供了基础功能。包含 BeanFactory 类（Spring 框架的核心，采用工厂

模式实现 IoC）

- 应用上下文模块：扩展了 BeanFactory，添加了对 I18N（国际化）、系统生命周期事件及验证的支持，并提供许多企业级服务，如电子邮件服务、JNDI 访问、EJB 集成、远程调用及定时服务，并支持与模板框架（如 Velocity 和 FreeMarker）的集成
- AOP 模块：对面向切面提供了丰富的支持，是 Spring 应用系统开发切面的基础；并引入 metadata 编程
- JDBC 和 DAO 模块：
- O/R 映射模块：
- Web 模块：建立在应用上下文模块的基础上，提供了适合 web 系统的上下文，另外，该模块支持多项面向 web 的任务，如透明处理多文件上传请求，自动将请求参数绑定到业务对象中等
- MVC 框架：

所有模块都是建立在核心容器之上的，容器规定如何创建、配置和管理 Bean，以及其它细节

1.2 示例

1.2.1 Spring 简单示例

程序清单 1.1 GreetingService 接口将实现从接口中分离出来

```
package com.springinaction.chapter01.hello;
public interface GreetingService {
    public void sayGreeting();
}
```

程序清单 1.2 GreetingServiceImpl.java 负责打印问候语

```
package com.springinaction.chapter01.hello;

public class GreetingServiceImpl implements GreetingService {
    private String greeting;
    public GreetingServiceImpl() {}
    public GreetingServiceImpl(String greeting) {
        this.greeting = greeting;
    }
    public void sayGreeting() {
        System.out.println(greeting);
    }
    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }
}
```

程序清单 1.3 在 Spring 中配置 Hello World

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="greetingService"
            class="com.springinaction.chapter01.hello.GreetingServiceImpl">
        <property name="greeting">
            <value>Buenos Dias!</value>
        </property>
    </bean>
</beans>
```

程序清单 1.4 Hello World 示例的主类

```
package com.springinaction.chapter01.hello;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
public class HelloApp {
    public static void main(String[] args) throws Exception {
        Resource resource = new FileSystemResource("hello.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        GreetingService greetingService =
            (GreetingService) factory.getBean("greetingService");
        greetingService.sayGreeting();
    }
}
```

1.2.2 IoC 示例

使用 IoC，对象的依赖都是在对象创建时由负责协调系统中各个对象的外部实体提供的。

减少耦合的一个通常做法是将具体实现隐藏在接口下，使得具体实现类的替换不会影响到引用类。

程序清单 1.5 Quest.java（使用接口解耦合）

```
package com.springinaction.chapter01.knight;
public interface Quest {
    public abstract Object embark() throws QuestException;
}
```

程序清单 1.6 HolyGrailQuest.java（Quest 的实现类）

```
package com.springinaction.chapter01.knight;

public class HolyGrailQuest implements Quest{
    public HolyGrailQuest() {}
    public Object embark() throws QuestException {
        //   Do whatever it means to embark on a quest
        return new HolyGrail();
    }
}
```

程序清单 1.7 Knight.java（使用接口解耦合）

```
package com.springinaction.chapter01.knight;

public interface Knight {
    public Object embarkOnQuest() throws QuestException;
    public String getName();
}
```

程序清单 1.8 KnightOfTheRoundTable.java（Knight 的实现类，骑士被动获取探险任务）

```
package com.springinaction.chapter01.knight;

public class KnightOfTheRoundTable implements Knight {
    private String name;
    private Quest quest;

    public KnightOfTheRoundTable(String name) {
        this.name = name;
    }
    public Object embarkOnQuest() throws QuestException {
        System.out.println("ceshi");
        return quest.embark();
    }
    public void setQuest(Quest quest) {
        this.quest = quest;
    }
    public String getName() {
        return this.name;
    }
}
```

程序清单 1.9 knight.xml（将一个探险任务装配给一个骑士）

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframeworkorg/dtd/spring-beans.dtd">
<beans>
    <!--定义一个探险任务-->
    <bean id="quest" class= "com.springinaction.chapter01.knight.HolyGrailQuest"/>
    <!—定义一个骑士-->
    <bean id="knight"
        class="com.springinaction.chapter01.knight.KnightOfTheRoundTable">
        <constructor-arg>
            <value>Bedivere</value><!—设置骑士的名字-->
        </constructor-arg>
        <property name="quest">
            <ref bean="quest"/><!—给予骑士一个探险任务-->
        </property>
    </bean>
</beans>
```

程序清单 1.10 KnightApp.java（运行 Knight 例子）

```
package com.springinaction.chapter01.knight;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
public class KnightApp {
    public static void main(String[] args) throws Exception {
        Resource resource = new FileSystemResource("knight.xml");
        BeanFactory factory = new XmlBeanFactory(resource) ;

        Knight knight = (Knight)factory.getBean("knight");
        knight.embarkOnQuest();
    }
}
```

程序清单 1.11 KnightOfTheRoundTableTest.java（KnightOfTheRoundTable 的测试类）

```
package com.springinaction.chapter01.knight;
import junit.framework.TestCase;

public class KnightOfTheRoundTableTest extends TestCase {
    public void testEmbarkOnQuest() {
        KnightOfTheRoundTable knight = new KnightOfTheRoundTable("Bedivere");
        Quest quest = new HolyGrailQuest();
```

```
        knight.setQuest(quest);
        HolyGrail grail;
        try {
            grail = (HolyGrail)knight.embarkOnQuest();
            assertNotNull(grail);
            assertTrue(grail.isHoly());
        } catch (QuestException e) {
            e.printStackTrace();
        }
    }
}
```

1.2.3 AOP

1.2.3.1AOP 简介

AOP 常被定义为一种编程结束，用来在系统中提升业务的分离。

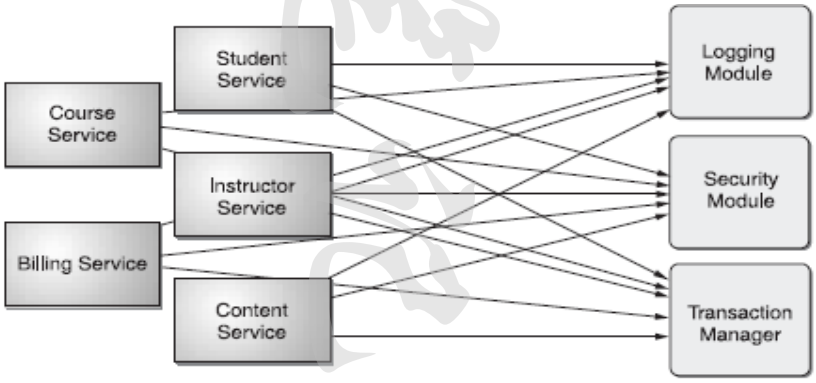


图 1-4 系统业务（如日志和安全）的调用常分散在各个模块中，而这些业务并不是该模块的主要业务

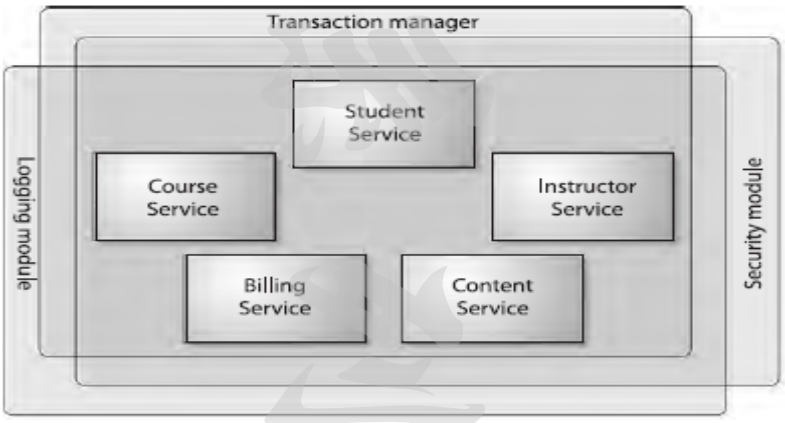


图 1-5 使用 AOP，系统业务覆盖了它影响到的组件

1.2.3.2AOP 示例

程序清单 1.12 MinstrelAdvice.java（面向切面的吟游诗人）

```
package com.springinaction.chapter01.knight;
import java.lang.reflect.Method;
import org.apache.log4j.Logger;
import org.springframework.aop.MethodBeforeAdvice;
public class MinstrelAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object target)
            throws Throwable {
        Knight knight = (Knight) target;
        //得到目标类的logger
        Logger song = Logger.getLogger(target.getClass());
        song.debug("Brave " + knight.getName() + " did " + method.getName());
    }
}
```

修改 knight.xml 编织切面，将 MinstrelAdvice 装配到 Knight 中

程序清单 1.13 knight.xml（将 MinstrelAdvice 装配到 Knight 中）

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="quest"
        class="com.springinaction.chapter01.knight.HolyGrailQuest" />
    <bean id="knightTarget"
        class="com.springinaction.chapter01.knight.KnightOfTheRoundTable"
        singleton="false">
        <constructor-arg>
            <value>Bedivere</value>
        </constructor-arg>
        <property name="quest">
            <ref bean="quest" />
        </property>
    </bean>
    <!--创建Minstrel实例-->
    <bean id="minstrel"
        class="com.springinaction.chapter01.knight.MinstrelAdvice" />
    <bean id="knight"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <list>
                <value>
                    <!--拦截Knight的方法-->
```

```
                    com.springinaction.chapter01.knight.Knight
                </value>
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>minstrel</value><!--先让Minstrel处理请求-->
            </list>
        </property>
        <property name="target">
            <ref bean="knightTarget" /><!--然后Knight处理请求-->
        </property>
    </bean>
</beans>
```

图 1-7 面向切面的吟游诗人覆盖了骑士，在骑士没有察觉的情况下记录骑士的行动

1.3 Spring 比较

学习比较

1.3.1 比较 Spring 和 EJB

表 1.1 Spring 和 EJB 功能比较

特征	EJB	Spring
事务管理	● 必须使用 JTA 事务管理器 ● 支持跨越远程调用的事务	● 通过 PlatformTransactionManager 接口 ● 自身不支持分布式事务－需使用 JTA
声明式事务支持	● 可使用部署文件声明式定义事务 ● 可通配符*对每个方法或每个类定义事务行为 ● 不能声明式定义回滚动作－只能采用编码方式实现	● 可通过 Spring 配置文件或类元数据定义声明式事务 ● 可显示或使用正则表达式为方法定义事务行为 ● 可声明式地为每个方法和每个异常类型定义回滚动作
持久化	● 支持编码式 BMP 和声明式 CMP	● 提供一个集成多种持久化技术的框架（JDBC、Hibernate、JDO 和 iBATIS）
声明式安全	● 支持基于用户和角色的声明式安全管理，对用户和角色的管理和实现是容器特定的 ● 在部署描述中声明安全配置	● 没有自己的安全实现 ● Acegi，建立在 Spring 基础之上的开源安全框架，提供了通过配置文件和类元数据配置的声明式安全服务
分布计算	● 提供容器管理的远程方法调用	● 通过 RMI、JAX-RPC 和 WebService 提供远程调用代理

1.3.2 关于其他轻量级容器

表 1.2 IoC 类型

类型	名称	描述
类型 1	接口依赖	Beans 必须实现指定的接口，这样它的依赖类才能被容器管理
类型 2	设值注入	依赖类和属性通过 Bean 的 setter 方法注入
类型 3	构造注入	依赖类和属性通过 Bean 的构造方法注入

2 装配 bean

Spring 提供两种不同的容器：

- Bean 工厂（由 org.springframework.beans.factory.BeanFactory 接口定义）是最简单的容器，提供基础的依赖注入支持
- 应用上下文（由 org.springframework.context.ApplicationContext 接口定义）建立在 bean 工厂基础之上，提供系统构架服务。

2.1 简介

2.1.1 BeanFactory 介绍学习比较

2.1.2 应用上下文

应用上下文不仅载入 Bean 定义信息，装配 Bean，根据需要分发 Bean，还提供如下功能：

- 文本信息解析工具，包括对国际化的支持
- 载入文件资源的通用方法
- 向注册为监听器的 Bean 发送事件

常用实现类如下：

- ClassPathXmlApplicationContext：从类路径中 XML 文件载入上下文定义信息，把上下文定义文件当成类路径资源；（在类路径中寻找）
- FileSystemXmlApplicationContext：从文件系统中的 XML 文件载入上下文定义信息（在指定的路径中寻找）
- XmlWebApplicationContext：从 web 系统中的 XML 文件载入上下文定义信息

2.1.3 Bean 的生命

如图 2.1 所示，在 BeanFactory 中 Bean 的生命周期：

1. 容器寻找 Bean 的定义信息并将其实例化
2. 使用依赖注入，Spring 按照 Bean 定义信息配置 Bean 的所有属性

3. 如果 Bean 实现了 BeanNameAware 接口，工厂调用 Bean 的 setBeanName()方法传递 Bean 的 ID
4. 如果 Bean 实现了 BeanFactoryAware 接口，工厂调用 setBeanFactory()方法传入工厂自身

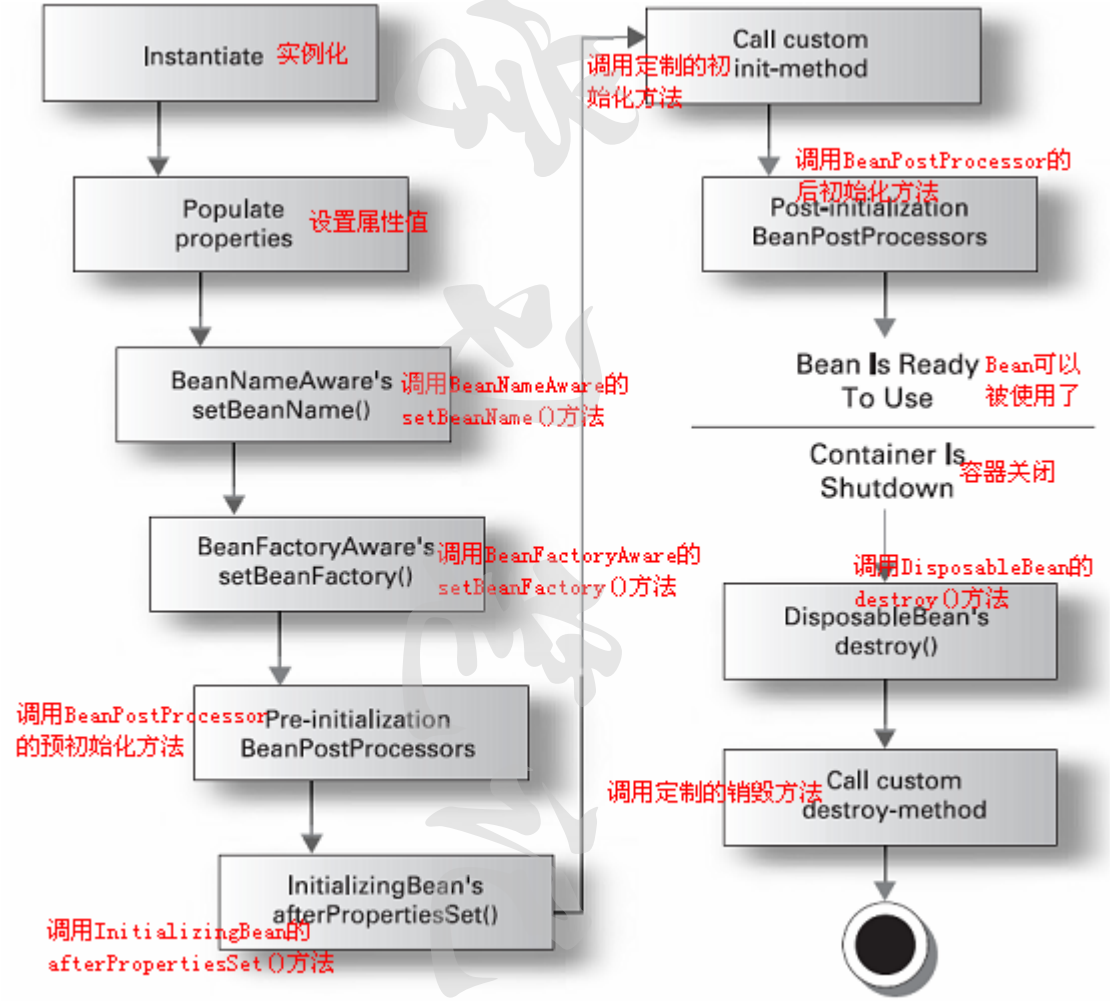


Figure 2.1 The life cycle of a bean within a Spring bean factory container

5. 如果有 BeanPostProcessor 和 Bean 关联，那么其 postProcessBeforeInitialization()方法将被调用
6. 如果 Bean 指定了 init-method 方法，将被调用
7. 最后，如果有 BeanPostProcessor 和 Bean 关联，那么其 postProcessAfterInitialization()方法将被调用

此时，Bean 已经可以被应用系统使用，并将被保留在 BeanFactory 中直到它不在被需要。有两种方法可将其从 BeanFactory 中删除掉。

1. 如 Bean 实现了 DisposableBean 接口，destroy()方法被调用
2. 如指定了定制的销毁方法，就调用这个方法

如图 2.2 所示，Bean 在 Spring 应用上下文中的生命周期与在 BeanFactory 中的生命周期只有一点不同：如 Bean 实现了 ApplicationContextAware 接口，setApplicationContext()方法会被调用。

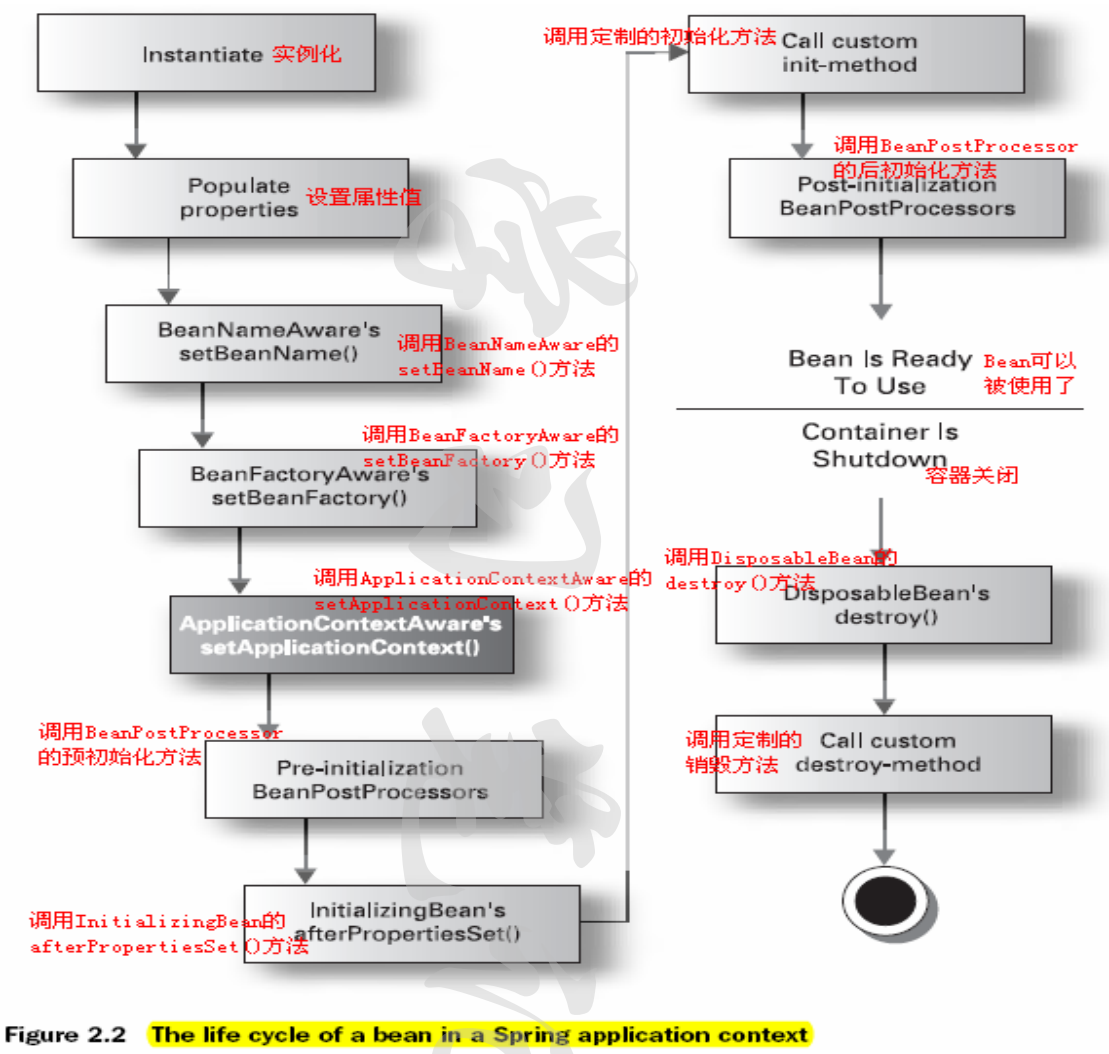


Figure 2.2 The life cycle of a bean in a Spring application context

2.2 基本装配

2.2.1 示例

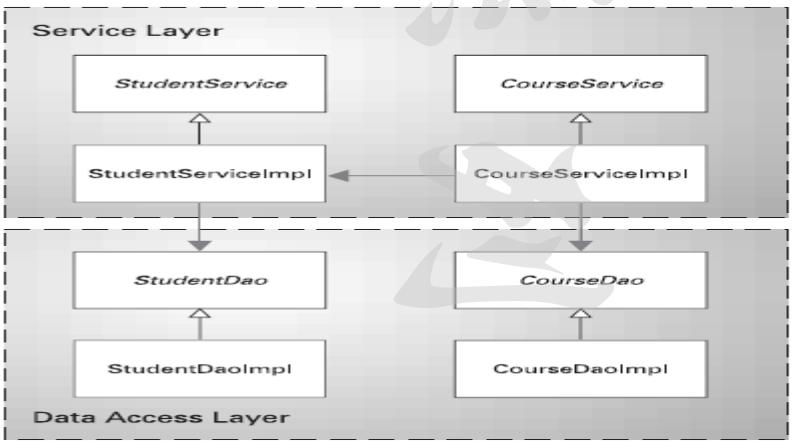


Figure 2.3 The beans that make up the service layer of the Spring Training application

程序清单 2.1 StudentService.java

```
package chapter02;

public interface StudentService {
    public Student getStudent(String id);
    public void createStudent(Student student);
    public java.util.Set getCompletedCourses(Student student);
}
```

程序清单 2.2 CourseService.java

```
package chapter02;

public interface CourseService {
    public Course getCourse(String id);
    public void createCourse(Course course);
    public java.util.Set getAllCourses();
    public void enrollStudentInCourse(Course course,Student student) throws
CourseException;
}
```

程序清单 2.3 StudentServiceImpl.java(StudentService 处理有关学生的功能)

```
package chapter02;

import java.util.Set;
public class StudentServiceImpl implements StudentService {
    private StudentDao studentDao;
    public StudentServiceImpl(StudentDao studentDao){//通过构造方法注入
        this.studentDao = studentDao;
    }
    public void setStudentDao(StudentDao studentDao) {//或通过setter方法注入
        this.studentDao = studentDao;
    }
    public void createStudent(Student student) {
        studentDao.create(student);
    }
    public Set getCompletedCourses(Student student) {
        return studentDao.getCompletedCourses(student);
    }
    public Student getStudent(String id) {
        return studentDao.findById(id);
    }
}
```

程序清单 2.4 StudentDao.java（StudentServiceImpl 将其很多职责委托给了 StudentDao）

```
package chapter02;
import java.util.Set;
/**
 * 处理与数据库的交互来读写学生信息
 */
public interface StudentDao {
    Student findById(String id);
    void create(Student student);
    Set getCompletedCourses(Student student);
}
```

程序清单 2.5 CourseServiceImpl.java

```
package chapter02;
import java.util.Iterator;
import java.util.Set;
public class CourseServiceImpl implements CourseService {
    private CourseDao courseDao;
    private StudentService studentService;
    private int maxStudents;
    public CourseServiceImpl(CourseDao dao) {//通过构造方法注入设置CourseDao
        this.courseDao = dao;
    }
    public void setStudentService(StudentService service) {//通过setter方法设置
        this.studentService = service;
    }
    public void setMaxStudents(int maxStudents) {
        this.maxStudents = maxStudents;
    }
    public int getMaxStudents() {
        return maxStudents;
    }
    public Course getCourse(String id) {
        return courseDao.findById(id);
    }
    public void createCourse(Course course) {
        courseDao.create(course);
    }
    public void enrollStudentInCourse(Course course, Student student)
            throws CourseException {
        //在学生可以登记一门课之前，必须已修完所有预修课程
        if (course.getStudents().size() >= maxStudents) {
            throw new CourseException("Course is full");
        }
```

```
        //判断学生是否满足先决条件
        enforcePrerequisites(course, student);
        course.getStudents().add(student);
        courseDao.update(course);
    }
    private void enforcePrerequisites(Course course, Student student)
            throws CourseException {
        Set completed = studentService.getCompletedCourses(student);
        Set prereqs = course.getPrerequisites();
        for (Iterator iter = prereqs.iterator(); iter.hasNext();) {
            if (!completed.contains(iter.next())) {
                //如果学生不满足先决条件，抛出异常
                throw new CourseException("Prerequisites are not met.");
            }
        }
    }
    public Set getAllCourses() {
        return courseDao.getAllCourses();
    }
}
```

程序清单 2.6 CourseDao.java（CourseServiceImpl 将其很多职责委托给了 CourseDao）

```
package chapter02;
import java.util.Set;
public interface CourseDao {
    void update(Course course);
    Course findById(String id);
    void create(Course course);
    Set getAllCourses();
}
```

程序清单 2.7 training-dao.xml（向容器中装配 DAO Bean）

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--注册courseDao bean-->
    <bean id="courseDao" class="chapter02.CourseDaoImpl"/>
    <!--注册studentDao bean-->
    <bean id="studentDao" class="chapter02.StudentDaoImpl"/>
</beans>
```

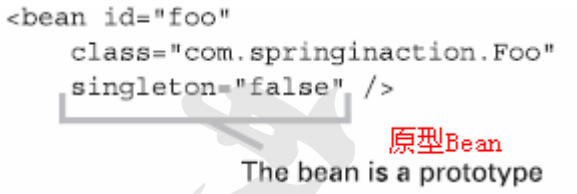
程序清单 2.8 training-service.xml（向容器中装配逻辑层 Bean）

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--注册studentService bean-->
    <bean id="studentService" class="chapter02.StudentServiceImpl">
        <constructor-arg>
            <ref bean="studentDao" />
        </constructor-arg>
    </bean>
    <!--注册courseService bean-->
    <bean id="courseService" class="chapter02.CourseServiceImpl">
        <constructor-arg>
            <ref bean="courseDao" />
        </constructor-arg>

        <property name="maxStudents">
            <!--通过子元素<value>设置基本类型属性-->
            <value>30</value>
        </property>
        <property name="studentService">
            <!--通过子元素<ref>设置指向其他Bean的属性-->
            <ref bean="studentService" />
        </property>
    </bean>
</beans>
```

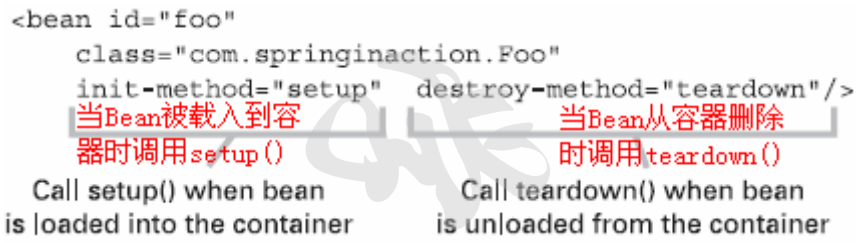
2.2.2 其他说明

2.2.2.1原型与单实例



Spring 中的 Bean 缺省情况下是单实例模式。<bean>的 singleton 属性告诉上下文该 Bean 是否为单实例 Bean，缺省是 true，为 false 表示该 Bean 为原型 Bean。

2.2.2.2实例化与销毁



Bean 实例化时，有时需要做些初始化工作，然后才能使用；同样，当 Bean 从容器删除时需要按顺序做些清理工作。

在 Bean 的定义中设置 init-method，此方法在 Bean 被实例化时被调用；同样，也可设置 destory-method，该方法在 Bean 从容器删除之前被调用。

2.2.2.3内部 Bean

程序清单 2.9 training-nest.xml（使用内部 Bean 设置属性）

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--注册courseService bean-->
    <bean id="courseService" class="chapter02.CourseServiceImpl">
        <constructor-arg>
            <ref bean="courseDao" />
        </constructor-arg>

        <property name="maxStudents">
            <!--通过子元素<value>设置基本类型属性-->
            <value>30</value>
        </property>
        <property name="studentService">
            <!--注册内部Bean:studentService-->
            <bean id="studentService"
class="chapter02.StudentServiceImpl">
                <constructor-arg>
                    <ref bean="studentDao" />
                </constructor-arg>
            </bean>
        </property>
    </bean>
</beans>
```

2.2.2.4装配集合

表 2.1 Spring 装配支持的集合类型

XML	类型
<list>	java.awt.List,arrays
<set>	java.awt.Set
<map>	java.awt.Map
<props>	java.awt.Properties

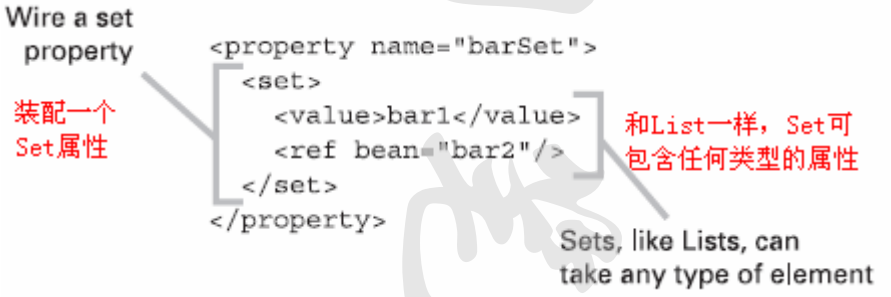
2.2.2.4.1示例

程序清单 2.10（装配集合）

```
    <bean id="sessionFactoryStats"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSourceStats"/>
        <property name="mappingResources">
            <list>
                <value>example/ex1.hbm.xml</value>
                <value>example/ex2.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    ${hibernate.dialect}
                </prop>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.jdbc.batch_size">20</prop>
                <prop key="hibernate.jdbc.fetch_size">20</prop>
                <prop key="hibernate.generate_statistics">true</prop>
            </props>
        </property>
        <property name="eventListeners">
            <map>
                <entry key="merge">
                    <bean
class="org.springframework.orm.hibernate3.support.IdTransferringMerge
EventListener"/>
                </entry>
            </map>
        </property>
    </bean>
```

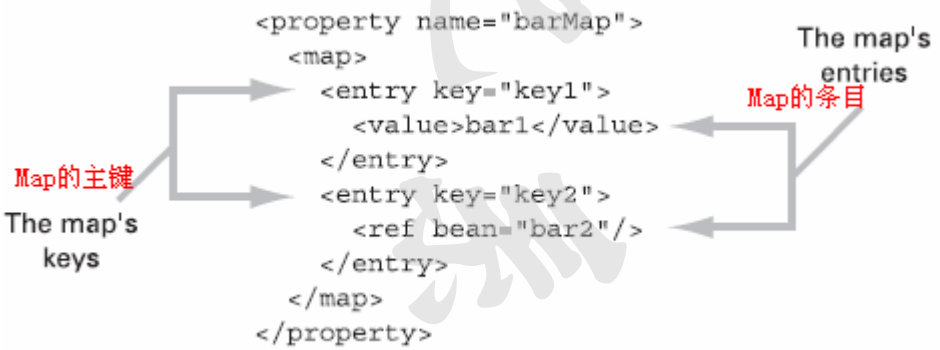
2.2.2.4.2 装配 List 和数组

2.2.2.4.3 装配 Set



学习比较

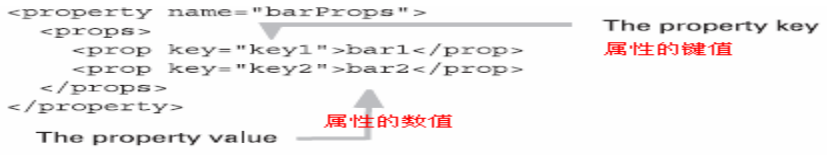
2.2.2.4.4 装配 Map



Map 中的每条条目是由一个主键和一个数值组成的，用<entry>元素来定义一条条目。Map 中的<entry>的数值和<list>及<set>的一样，可以是任何有效的属性元素，包括<value>、<ref>、<list>、<map>等；注意的是，配置<entry>时，属性 key 的值只能是 String。

2.2.2.4.5 装配 Properties

Java.util.Properties 集合是最后一个能在 Spring 中装配的集合类，使用<props>元素来装配。使用<prop>元素表示每条熟悉。但<prop>的值只能是 String 型的。



程序清单 2.11（使用<props>进行 URL 映射）

```
<property name="mappings">
    <props>
        <prop key="/viewCourseDetails.htm">
            viewCourseController
        </prop>
    </props>
</property>
```

2.2.2.5设置 null

程序清单 2.12（将一个属性设置为 null）

```
<property name="foo"><null/><property>
```

2.2.2.6解决构造方法参数的不确定性

学习比较

解决构造方法参数的不确定性有两种方法：

- 序号：<constructor-arg>元素有一个 index 属性，可用其来指定构造方法的顺序。

程序清单 2.13（使用 index 解决构造方法不确定性）

```
<bean id="foo" class="com.springinaction.Foo">
    <constructor-arg index="1">
        <value>http://www.manning.com</value>
    </constructor-arg>
    <constructor-arg index="0">
        <value>http://www.springinaction.com</value>
    </constructor-arg>
</bean>
```

- 类型。

程序清单 2.14（使用 type 解决构造方法不确定性）

```
<bean id="foo" class="com.springinaction.Foo">
    <constructor-arg type="java.lang.String">
        <value>http://www.manning.com</value>
    </constructor-arg>
    <constructor-arg type="java.net.URL">
        <value>http://www.springinaction.com</value>
    </constructor-arg>
</bean>
```

2.3 自动装配

只要设置需要自动装配的<bean>中的 autowire 属性即可让 Spring 自动装配。

```
<bean id="foo"
    class-"com.springinaction.Foo"
    autowire="autowire type"
/>
```

自动装配该Bean的属性
Auto-wire this bean's properties

有 4 种自动装配类型：

- byName：在容器中寻找和需要自动装配的属性名相同的 Bean（或 ID），如没有找到相符的 Bean，该属性就没有被装配上。
- byType：在容器中寻找一个与需要自动装配的属性类型相同的 Bean；如没有找到相符的 Bean，该属性就没有被装配上，如找到超过一个相符的 Bean 抛出异常 org.springframework.beans.factory.UnsatisfiedDependencyException
- constructor：在容器中查找与需要自动装配的 Bean 的构造方法参数一致的一个或多个 Bean。如存在不确定 Bean 或构造方法，容器会抛出异常 org.springframework.beans.factory.UnsatisfiedDependencyException。
- autodetect：首先尝试使用 constructor 来自动装配，然后使用 byType 方式。不确定性的处理与 constructor 和 byType 方式一样。

2.4 使用 Spring 的特殊 Bean

学习比较

特殊 Bean 的作用：

- 通过配置后加工 Bean，涉及到 Bean 和 Bean 工厂的生命周期中
- 从外部配置文件中加载配置信息
- 改变 Spring 的依赖注入，使其在设置 Bean 属性时，自动将字符串转换成其他类型。
- 从属性文件中加载文本信息，包括国际化信息
- 监听并处理由其他 Bean 以及 Spring 容器发布的系统消息
- 知道其在 Spring 容器的唯一标识

2.4.1 对 Bean 进行后处理

后处理是在 Bean 实例化以及装配完成之后发生的。在 Bean 被创建以及装配之后，BeanPostProcessor 接口提供两次修改 Bean 的机会。

程序清单 2.15 BeanPostProcessor.java

```
package org.springframework.beans.factory.config;
import org.springframework.beans.BeansException;
public interface BeanPostProcessor {
    Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException;
    Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException;
}
```

其中 postProcessBeforeInitialization()方法在 Bean 初始化（即调用 afterPropertiesSet()及 Bean 指定的 initmethod 方法）之前被调用；同样，postProcessAfterInitialization()方法在初始化之后立即被调用。

2.4.1.1示例

程序清单 2.16 Fuddifier.java（后处理 Bean，转换字符串属性）

```
package chapter02.other;
import java.lang.reflect.Field;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
public class Fuddifier implements BeanPostProcessor {
    /**
     * 循环Bean的所有属性，寻找java.lang.String类型的属性。
     * 对于每个String属性，将其传递给fuddify(String)方法，进行转换
     */
    public Object postProcessAfterInitialization(Object bean, String name)
            throws BeansException {
        Field[] fields = bean.getClass().getDeclaredFields();
        try {
            for (int i = 0; i < fields.length; i++) {
                if (fields[i].getType().equals(java.lang.String.class)) {
                    fields[i].setAccessible(true);
                    String original = (String) fields[i].get(bean);
                    fields[i].set(bean, fuddify(original));
                }
            }
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        return bean;
    }
    private String fuddify(String orig) {
        if (orig == null)
            return orig;
        return orig.replaceAll("(r|l)", "w").replaceAll(" (R|L) ", "W");
    }
    public Object postProcessBeforeInitialization(Object bean, String name)
            throws BeansException {
        return bean;
    }
}
```

编写后处理 Bean，还需要注册后处理 Bean：

- Bean 工厂中注册：

程序清单 2.17 Bean 工厂中注册后处理 Bean 部分代码

```
XmlBeanFactory factory = new XmlBeanFactory(
        new FileSystemResource("/training-service.xml"));
BeanPostProcessor fuddifier = new Fuddifier();
factory.addBeanPostProcessor(fuddifier);
```

- 应用上下文中注册：

程序清单 2.18 应用上下文中注册后处理 Bean 部分代码

```
<bean id=" fuddifier" class="chapter02.other.Fuddifier"/>
```

2.4.2 对 Bean 工厂进行后处理

BeanFactoryPostProcessor 在 Bean 工厂载入所有 Bean 的定义后，实例化 Bean 之前，对 Bean 工程做一些后处理工作。

程序清单 2.19 BeanFactoryPostProcessor.java

```
package org.springframework.beans.factory.config;

import org.springframework.beans.BeansException;
public interface BeanFactoryPostProcessor {
    void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
throws BeansException;
}
```

2.4.2.1示例

程序清单 2.20 BeanCounter.java（BeanFactoryPostProcessor 实现类，统计容器中 Bean 的个数）

```
package chapter02.other;
import org.apache.log4j.Logger;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import
org.springframework.beans.factory.config.ConfigurableListableBeanFactory;

public class BeanCounter implements BeanFactoryPostProcessor {
    private Logger LOGGER = Logger.getLogger(BeanCounter.class);
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
factory)
            throws BeansException {
        LOGGER.debug("BEAN COUNT: " + factory.getBeanDefinitionCount());
    }
}
```

Bean 工厂中无法使用 BeanFactoryPostProcessor，在应用上下文中使用只需在 XML 中添

加如下配置：

程序清单 2.21 注册 BeanFactoryPostProcessor 实现类

```
    <bean id="beanCounter" class="chapter02.other.BeanCounter"/>
```

2.4.3 分散配置

在应用上下文中，使用 PropertyPlaceholderConfigurer 从外部属性文件装载配置信息：

程序清单 2.22 应用上下文中从单个外部属性文件装载配置信息

```
    <bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigure
r">
        <property name="location" value="WEB-INF/jdbc.properties"/>
    </bean>
```

其中，location 属性允许使用单个配置文件，可使用其 locations 属性设置配置文件列表：

程序清单 2.23 应用上下文中从多个外部属性文件装载配置信息

```
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigure
r">
    <property name="locations">
        <list>
            <value>jdbc.properties</value>
            <value>security.properties</value>
            <value>application.properties</value>
        </list>
    </property>
</bean>
```

这样，就可用占位符变量代替 Bean 配置文件中硬编码配置了。语法上，占位符变量采用${variable}的形式，如下：

程序清单 2.24 使用占位符代替硬编码配置

```
<bean id="dataSourceDBA"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username.dba}"/>
    <property name="password" value="${jdbc.password.dba}"/>
</bean>
```

2.4.4 定制属性编辑器

JavaBean 的接口 java.beans.PropertyEditor 提供将字符串值映射成非 String 类型的方法（其直接实现类为 PropertyEditorSupport）：

- getAsText（）：返回一个属性值的字符串

- setAsText(String text)：将传递进来的字符串赋给 Bean 的属性

Spring 提供了几个建立在 PropertyEditorSupport 之上的定制编辑器：

- URLEditor：将字符串与 java.net.URL 相互转换
- ClassEditor：使用包含全称类名的字符串设置 java.lang.Class 属性
- CustomDateEditor：使用某种 java.text.DateFormat 对象将字符串设置给 java.util.Date 属性
- FileEditor：使用包含文件路径的字符串设置 java.io.File 属性
- LocaleEditor：使用包含地域信息（如“en_US”）的字符串设置 java.util.Local 属性
- StringArrayPropertyEditor：将一个包含逗号的 String 转换成 String 数组属性
- StringTrimmerEditor：自动修正字符串属性，可选择将字符串转变成 null

2.4.4.1示例

程序清单 2.25 PhoneEditor.java（继承 PropertyEditorSupport 实现自己的编辑器）

```
package chapter02.phone;
import java.beans.PropertyEditorSupport;
public class PhoneEditor extends PropertyEditorSupport {
    public void setAsText(String textValue) {
        String stripped = stripNonNumeric(textValue);
        String areaCode = stripped.substring(0, 3);
        String prefix = stripped.substring(3, 6);
        String number = stripped.substring(6);
        PhoneNumber phone = new PhoneNumber(areaCode, prefix, number);
        setValue(phone);
    }
    private String stripNonNumeric(String original) {
        StringBuffer allNumeric = new StringBuffer();
        for (int i = 0; i < original.length(); i++) {
            char c = original.charAt(i);
            if (Character.isDigit(c)) {
                allNumeric.append(c);
            }
        }
        return allNumeric.toString();
    }
}
```

实现 PhoneEditor 之后需要注册 PhoneEditor：

- Bean 工厂中：

程序清单 2.26 注册自定义编辑器

```
        String prop = "chapter02/phone/phone.xml";
        Resource res = new ClassPathResource(prop);
        PhoneEditor pe = new PhoneEditor();
        XmlBeanFactory beanFactory = new XmlBeanFactory(res);
```

```
        //注册自定义编辑器
        beanFactory.registerCustomEditor(PhoneNumber.class,pe );
```

- 应用上下文中：

程序清单 2.27 注册自定义编辑器

```
<bean id="customEditorConfigurer"
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="chapter02.phone.PhoneNumber">
                <bean id="phoneEditor" class="chapter02.phone.PhoneEditor" />
            </entry>
        </map>
    </property>
</bean>
```

使用自定义编辑器的应用上下文配置文件：

程序清单 2.28 使用了自定义编辑器的 Spring 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="customEditorConfigurer"
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
        <property name="customEditors">
            <map>
                <entry key="chapter02.phone.PhoneNumber">
                    <bean id="phoneEditor" class="chapter02.phone.PhoneEditor"
/>
                </entry>
            </map>
        </property>
    </bean>
    <bean id="contact" class="chapter02.phone.Contact">
        <property name="phoneNumber">
            <value>888-555-1212</value>
        </property>
    </bean>
</beans>
```

程序清单 2.29 Contact.java

```
package chapter02.phone;
public class Contact {
    private PhoneNumber phoneNumber;
    public void setPhoneNumber(PhoneNumber phoneNumber) {
```

```
        this.phoneNumber = phoneNumber;
    }
    public PhoneNumber getPhoneNumber() {
        return phoneNumber;
    }
}
```

程序清单 2.30 PhoneNumber.java

```
package chapter02.phone;
public class PhoneNumber {
    private String areaCode;
    private String prefix;
    private String number;
    public PhoneNumber() {
    }
    public PhoneNumber(String areaCode, String prefix, String number) {
        super();
        this.areaCode = areaCode;
        this.prefix = prefix;
        this.number = number;
    }
    public String getAreaCode() {
        return areaCode;
    }
    public String getNumber() {
        return number;
    }
    public String getPrefix() {
        return prefix;
    }
}
```

2.4.5 解析文本信息

Spring 的应用上下文通过 MessageSource 接口为容器提供参数化信息支持：

程序清单 2.31 MessageSource.java

```
package org.springframework.context;
import java.util.Locale;
public interface MessageSource {
    String getMessage(String code, Object[] args, String defaultMessage,
Locale locale);
    String getMessage(String code, Object[] args, Locale locale) throws
NoSuchMessageException;
```

```
    String getMessage(MessageSourceResolvable resolvable, Locale locale)
throws NoSuchMessageException;
}
```

Srping 提供一个该接口的实现 ResourceBundleMessageSource，如需使用，向配置文件加入代码：

程序清单 2.32 使用 ResourceBundleMessageSource 的配置，id 必须为 messageSource

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename">
        <value>trainingtext</value>
    </property>
</bean>
```

2.4.6 事件

2.4.6.1监听事件

应用系统生命周期中，应用上下文会发布很多事件，系统事件有（这样事件都是抽象类 org.springframework.context.ApplicationEvent 的子类）：

- ContextClosedEvent：应用上下文关闭时发布
- ContextRefreshedEvent：应用上下文初始化或刷新时发布
- RequestHandledEvent：在 web 应用上下文中，当请求被处理后发布

如需对系统事件做出回应，则需实现 org.springframework.context.ApplicationListener 接口（此接口要求实现方法 onApplicationEvent(ApplicationEvent event)处理系统事件），如下：

程序清单 2.33 RefreshListener.java

```
package chapter02.phone;
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;
public class RefreshListener implements ApplicationListener {
    public void onApplicationEvent(ApplicationEvent event) {
        // TODO Auto-generated method stub
    }
}
```

程序清单 2.34 注册 RefreshListener

```
<bean id="refreshListener" class="com.springinaction.foo.RefreshListener"/>
```

2.4.6.2发布事件

首先，自定义一个事件（继承 ApplicationEvent），如程序清单 2.35

然后，发布事件（应用上下文的方法 publishEvent()发布事件），如程序清单 2.36

程序清单 2.35 CourseFullEvent.java

```
package chapter02;
import org.springframework.context.ApplicationEvent;
public class CourseFullEvent extends ApplicationEvent {
    private Course course;
    public CourseFullEvent(Object source, Course course) {
        super(source);
        this.course = course;
    }
    public Course getCourse() {
        return course;
    }
}
```

程序清单 2.36 发布事件

```
        String prop = "chapter02/phone/phone.xml";
        ApplicationContext context = new ClassPathXmlApplicationContext(prop);
        Course course = new Course();
        //发布事件
        context.publishEvent(new CourseFullEvent(this,course));
```

2.4.7 感知其他 Bean 学习比较

三个接口：

- BeanNameAware：知道自身的名称
- BeanFactoryAware：知道所处的 BeanFactory
- ApplicationContextAware：知道所处的 ApplicationContext

3 创建切面（AOP）

3.1 AOP 介绍

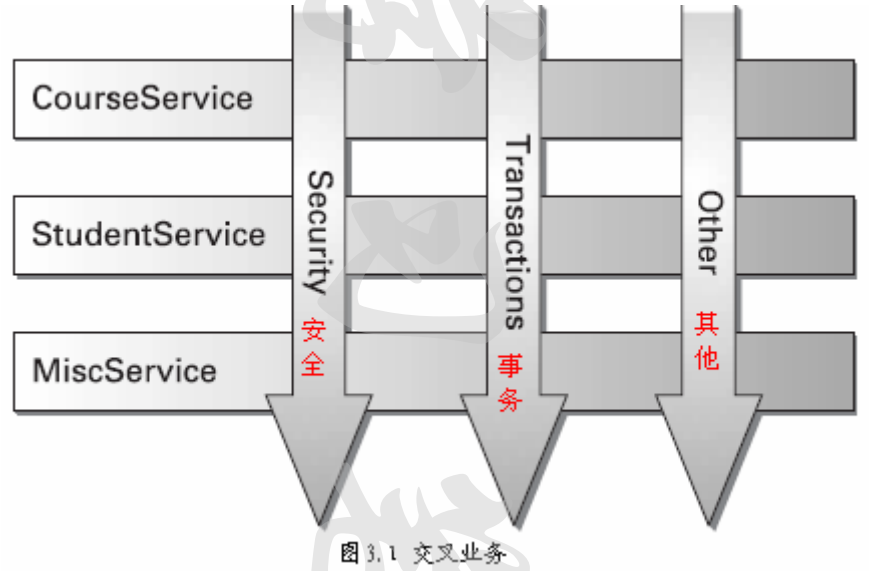


图 3.1 交叉业务

如图 3.1 所示，系统中存在交叉业务，可将这些业务模块化到特定对象切面中，如此有如两个好处：

- 每个业务逻辑放在一个地方，而不是分散到代码的各个角落
- 服务模块更加清晰

3.1.1 定义 AOP 术语

- 切面（Aspect）：需要实现的交叉功能
- 连接点（Joinpoint）：应用程序执行过程中插入切面的地点；此处可是方法调用、异常抛出甚至是要修改的字段
- 通知（Advice）：通知切面的实际实现。它通知应用系统新的行为，通知在连接点插入到应用系统中
- 切入点（Pointcut）：定义了通知应该应用在哪些连接点。通常通过指定类名和方法名，或匹配类名和方法名式样的正则表达式来指定切入点。
- 引入（Introduction）：允许为已存在类添加新方法和属性
- 目标对象（Target）：被通知对象。既可是编写的类也可是添加定制行为的第三方类。
- 代理（Proxy）：将通知应用到目标对象后创建的对象。
- 织入（Weaving）：将切面应用到目标对象从而创建一个新的代理对象的过程。切面在指定接入点被织入到目标对象中，织入发生在目标对象生命周期的多个点上：
  - 编译器：切面在目标对象编译时织入（需特殊的编译器）
  - 类装载期：切面在目标对象被载入到 JVM 时织入（需特殊的类载入器，在类的载入到应用系统之前增强目标对象的字节码）
  - 运行期：切面在应用系统运行时织入（通常，AOP 容器将在织入切面时动态

生成委托目标对象的代理对象）

通知包括需要应用的交叉行为；连接点是通知要在应用系统需要应用的所有切入点；切入点定义了通知要在哪些连接点应用（即哪些连接点要被通知）。

3.1.2 Spring 的 AOP 实现

关键点：

- 用 Java 编写 Spring 通知
- Spring 的运行时通知对象
- Spring 实现了 AOP 联盟接口
- Spring 只支持方法连接点

Srping 的两种代理创建方式：

- 如果目标对象实现了一个（或多个）接口暴露的方法，Spring 将使用 JDK 的 java.lang.reflect.Proxy 类创建代理。此类动态产生一个新类，该类实现了所需的接口，织入通知并代理对目标对象的所有请求
- 若目标对象未实现任何接口，Spring 使用 CGLIB 库生成目标对象的子类；在创建该子类时，织入通知并将对目标对象的调用委托给该子类。此种代理方式需注意：
  - 对接口创建代理优于对类创建代理
  - 标记为 final 的方法不能被通知

3.2 创建通知

学习比较

Table 3.1 Advice types in Spring 表3.1 Spring中通知类型

Advice type 通知类型	Interface 接口	Description 描述
Around	`org.aopalliance.intercept.MethodInterceptor`	Intercepts calls to the target method 拦截对目标对象方法调用
Before	`org.springframework.aop.BeforeAdvice`	Called before the target method is invoked 在目标方法被调用之前调用
After	`org.springframework.aop.AfterReturningAdvice`	Called after the target method returns 在目标方法被调用之后调用
Throws	`org.springframework.aop.ThrowsAdvice`	Called when target method throws an exception 当目标方法抛出异常时调用

下面以一个例子进行演示：

程序清单 3.1 KwikEMart.java

```
package chapter03.kwik;
public interface KwikEMart {
    Squishee buySquishee(Customer customer) throws KwikEMartException;
}
```

程序清单 3.2 ApuKwikEMart.java（KwikEMart 接口的实现）

```
package chapter03.kwik;
```

```
public class ApuKwikEMart implements KwikEMart {
    private boolean squisheeMachineEmpty;
    public Squishee buySquishee(Customer customer) throws KwikEMartException {
        if (customer.isBroke()) {
            throw new CustomerIsBrokeException();
        }
        if (squisheeMachineEmpty) {
            throw new NoMoreSquisheesException();
        }
        return new Squishee();
    }
}
```

3.2.1 前置通知

前置通知需实现 MethodBeforeAdvice 接口：

程序清单 3.3 MethodBeforeAdvice.java

```
package org.springframework.aop;
import java.lang.reflect.Method;
public interface MethodBeforeAdvice extends BeforeAdvice {
    void before(Method method, Object[] args, Object target) throws Throwable;
}
```

程序清单 3.4 WelcomeAdvice.java（前置通知，在客户买果酱之前向客户打招呼）

```
package chapter03.kwik;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class WelcomeAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object target) {
        Customer customer = (Customer) args[0];//将第一个参数转换为Customer
        System.out.println("Hello " + customer.getName()
                + ". How are you doing?");//和Customer打招呼
    }
}
```

前置通知返回类型为 void，其唯一能阻止目标方法被调用的途径是抛出异常（或调用 System.exit()）。

程序清单 3.5 kwikemart.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 创建代理目标对象 -->
    <bean id="kwikEMartTarget" class="chapter03.kwik.ApuKwikEMart" />
```

```
    <!-- 创建通知 -->
    <bean id="welcomeAdvice" class="chapter03.kwik.WelcomeAdvice" />

    <!-- 创建代理Bean 开始 -->
    <bean id="kwikEMart"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>chapter03.kwik.KwikEMart</value>
        </property>
        <property name="interceptorNames">
            <list>
                <value>welcomeAdvice</value>
            </list>
        </property>
        <property name="target">
            <ref bean="kwikEMartTarget" />
        </property>
    </bean>
    <!-- 创建代理Bean 结束 -->
</beans>
```

说明：ProxyFactoryBean 类被 BeanFactory 和 ApplicationContext 用于创建代理；配置创建代理 Bean 时以如下创建：

- 实现 KwikEMart 接口
- 应用 WelcomeAdvice（id 为 welcomeAdvice）通知对象到所有调用
- 像目标对象那样使用 ApuKwikEMart Bean

3.2.2 后置通知

后置通知需实现 AfterReturningAdvice 接口：

程序清单 3.6 AfterReturningAdvice.java

```
package org.springframework.aop;
import java.lang.reflect.Method;
import org.aopalliance.aop.Advice;
public interface AfterReturningAdvice extends Advice {
    void afterReturning(Object returnValue, Method method, Object[] args, Object
target) throws Throwable;
}
```

程序清单 3.7 ThankYouAdvice.java（后置通知，在客户买完果酱后表示谢谢）

```
package chapter03.kwik;
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
```

```
public class ThankYouAdvice implements AfterReturningAdvice {
    public void afterReturning(Object returnValue, Method method,
            Object[] arg2, Object target) throws Throwable {
        System.out.println("Thank you. Come again!");
    }
}
```

3.2.3 环绕通知

环绕通知需实现 MethodInterceptor 接口：

程序清单 3.8 MethodInterceptor.java（实现两种通知）

```
public interface MethodInterceptor extends Interceptor {
    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

与前面两种通知的区别：

- MethodInterceptor 能控制目标方法是否真的被调用。通过调用 MethodInvocation 的 proceed()方法来调用目标方法
- MethodInterceptor 能控制返回的对象

程序清单 3.9 OnePerCustomerInterceptor.java（限制一个客户只能买一果酱）

```
package chapter03.kwik;
import java.util.HashSet;
import java.util.Set;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class OnePerCustomerInterceptor implements MethodInterceptor {
    private Set customers = new HashSet();//定义包含用户的集合
    public Object invoke(MethodInvocation invocation) throws Throwable {
        Customer customer = (Customer) invocation.getArguments()[0];//获取当前
用户
        if (customers.contains(customer)) {
            //如果有重复客户则抛出异常
            throw new KwikEMartException("One per customer.");
        }
        Object squishee = invocation.proceed();//调用目标方法
        customers.add(customer);//添加用户
        return squishee;//返回目标方法结果
    }
}
```

3.2.4 异常通知

需实现 ThrowsAdvice 接口：

程序清单 3.10 ThrowsAdvice.java（）

```
package org.springframework.aop;
import org.aopalliance.aop.Advice;
/**
 * There aren't any methods on this interface, as methods are invoked by reflection.
 * Implementing classes should implement methods of the form
 * afterThrowing([Method], [args], [target], Throwable subclass)
 * The first three arguments are optional, and only useful if
 * we want further information about the joinpoint, as in AspectJ
 * after throwing advice.
 *
 * @author Rod Johnson
 */
public interface ThrowsAdvice extends Advice {
}
```

程序清单 3.11（实现 ThrowsAdvice 接口的类至少要有一个如下形式的方法）

```
    /**
     * @param throwable--Throwable,需抛出的异常
     */
    void afterThrowing(Throwable throwable)
    /**
     * @param method--Method,被调用的方法
     * @param args--Object[],被调用的方法参数
     * @param target--Object,目标对象
     * @param throwable--Throwable,需抛出的异常
     */
    void afterThrowing(Method method, Object[] args, Object target,Throwable
throwable)
```

程序清单 3.12 KwikEMartExceptionAdvice.java（ThrowsAdvice 接口的实现类）

```
package chapter03.kwik;
import org.springframework.aop.ThrowsAdvice;
public class KwikEMartExceptionAdvice implements ThrowsAdvice {
    public void afterThrowing(NoMoreSquisheesException e) {
        //本方法只捕获NoMoreSquisheesException进行处理
    }
    public void afterThrowing(CustomerIsBrokeException e) {
        //本方法只捕获CustomerIsBrokeException进行处理
    }
}
```

3.2.5 引入通知

引入通知给目标对象添加新的方法（以及属性），而其他通知是在目标对象的方法被调用的周围织入。

3.3 切入点

3.3.1 定义切入点

Spring 根据需要织入通知的类和方法来定义切入点。通知是根据其特性织入目标类和方法。Spring 的切入点框架的核心接口是 Pointcut。

程序清单 3.13 Pointcut.java（）

```
package org.springframework.aop;
public interface Pointcut {
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
    // could add getFieldMatcher() without breaking most existing code
    /**
     * Canonical Pointcut instance that always matches.
     */
    Pointcut TRUE = TruePointcut.INSTANCE;
}
```

程序清单 3.13 ClassFilter.java（类过滤切面，决定一个类是否符合通知的要求）

```
package org.springframework.aop;
public interface ClassFilter {
    boolean matches(Class clazz);
    ClassFilter TRUE = TrueClassFilter.INSTANCE;
}
```

其中 TRUE 适用于创建只根据方法决定时符合要求的切入点。

程序清单 3.14 MethodMatcher.java（方法过滤切面）

```
package org.springframework.aop;
import java.lang.reflect.Method;
public interface MethodMatcher {
    boolean matches(Method method, Class targetClass);
    boolean isRuntime();
    boolean matches(Method method, Class targetClass, Object[] args);
    MethodMatcher TRUE = TrueMethodMatcher.INSTANCE;
}
```

说明：接口 MethodMatcher 有三个方法，用在被代理对象生命周期的特定时期。

- matches(Method,Class)方法根据目标类和方法决定一个方法是否被通知，在 AOP

代理被创建时调用一次，其结果决定通知是否被织入；

- 若 matches(Method,Class)返回 true，isRuntime()被调用来决定 MethodMatcher 的类型：静态和动态（静态切入点通知总是被执行，如一个切入点是静态的，isRuntime()方法应返回 false；动态切入点根据运行时方法的参数值决定通知是否执行，若一个切入点是动态的，isRuntime()方法应返回 true），在代理类创建时调用一次。
- 如一个切入点是静态的，matches(Method,Class,Object[])方法永远不会被调用。对于动态切入点，目标对象方法每次被调用时，matches(Method,Class,Object[])方法被调用。

3.3.2 理解 Advisor

Advisor 把通知和切入点组合到一个对象中，接口为 PointcutAdvisor。

3.3.3 静态切入点

如创建自制的静态切入点继承抽象类 StaticMethodMatcherPointcut
常用的静态切入点：NameMatchMethodPointcut 和 JdkRegexpMethodPointcut

3.3.4 动态切入点

内置 ControlFlowPointcut。 学习比较

3.3.5 切入点实施

为了在应用系统中重复使用切入点，进行合并与交叉操作，Spring 提供如下两个类：

- ComposablePointcut：通过将已有的 ComposablePointcut、切入点、MethodMatcher 及 ClassFilter 对象进行合并或交叉，组装成一个新的 ComposablePointcut 对象；调用其实例的 intersection()或 union()方法实现
- Pointcuts：可对两个 Pointcut 对象进行合并

程序清单 3.15 UnionPointcut.java（实现通过配置创建切入点合并）

```
package chapter03;
import java.util.List;
import org.springframework.aop.ClassFilter;
import org.springframework.aop.MethodMatcher;
import org.springframework.aop.Pointcut;
import org.springframework.aop.framework.AopConfigException;
import org.springframework.aop.support.Pointcuts;
public class UnionPointcut implements Pointcut {
    private Pointcut delegate;//声明合并的Pointcut实例
    public ClassFilter getClassFilter() {//委托给Pointcut的方法
        return getDelegate().getClassFilter();
```

```
    }
    public MethodMatcher getMethodMatcher() {//委托给Pointcut的方法
        return getDelegate().getMethodMatcher();
    }
    private Pointcut getDelegate() {//如果没有配置则抛出异常
        if (delegate == null) {
            throw new AopConfigException("No pointcuts have been configured.");
        }
        return delegate;
    }
    public void setPointcuts(List pointcuts) {//创建组合切入点
        if (pointcuts == null || pointcuts.size() == 0) {
            throw new AopConfigException("Must have at least one Pointcut.");
        }
        delegate = (Pointcut) pointcuts.get(0);
        for (int i = 1; i < pointcuts.size(); i++) {
            Pointcut pointcut = (Pointcut) pointcuts.get(i);
            delegate = Pointcuts.union(delegate, pointcut);
        }
    }
}
```

学习比较

3.4 引入

引入通知影响整个类，通过给需要消息的类添加方法和属性来实现，既，可用已存在的类实现另外的接口，维持另外的状态（又叫混合）

3.4.1 IntroductionInterceptor

Spring 通过 MethodInterceptor 接口的子接口 IntroductionMethodInterceptor 实现引入，其添加一个方法：boolean implementsInterface (Class intf);

程序清单 3.16 Auditable.java（记录最近一次业务对象的修改时间）

```
package chapter03.training;
import java.util.Date;
public interface Auditable {
    void setLastModifiedDate(Date date);
    Date getLastModifiedDate();
}
```

程序清单 3.17 AuditableMixin.java（实现 IntroductionInterceptor）

```
package chapter03.training;
import java.util.Date;
```

```
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.IntroductionInterceptor;
public class AuditableMixin implements IntroductionInterceptor, Auditable {
    public boolean implementsInterface(Class intf) {//实现Auditable
        return intf.isAssignableFrom(Auditable.class);
    }
    public Object invoke(MethodInvocation m) throws Throwable {
        if (implementsInterface(m.getMethod().getDeclaringClass())) {
            //调用引入的方法
            return m.getMethod().invoke(this, m.getArguments());
        } else {//委托其他方法
            return m.proceed();
        }
    }
    /*===========以下为实现混合逻辑==================*/
    private Date lastModifiedDate;
    public Date getLastModifiedDate() {
        return lastModifiedDate;
    }
    public void setLastModifiedDate(Date lastModifiedDate) {
        this.lastModifiedDate = lastModifiedDate;
    }
}
```

Spring 提供的实现类：DelegatingIntroductionInterceptor

程序清单 3.18 `AuditableMixin.java`（继承 DelegatingIntroductionInterceptor）

```
package chapter03.training;
import java.util.Date;
import org.springframework.aop.support.DelegatingIntroductionInterceptor;
public class AuditableMixin extends DelegatingIntroductionInterceptor
    implements Auditable {
    private Date lastModifiedDate;
    public Date getLastModifiedDate() {
        return lastModifiedDate;
    }
    public void setLastModifiedDate(Date lastModifiedDate) {
        this.lastModifiedDate = lastModifiedDate;
    }
}
```

若混合体需改变目标对象方法的行为，则需实现 invoke()方法，如下的例子中：有一个接口 `Immutable` 接口，只需改变对其一个方法引入（对象是不可变的，其状态是不可变）：

程序清单 3.18 `AuditableMixin.java`（继承 DelegatingIntroductionInterceptor）

```
package chapter03.training;
import org.aopalliance.intercept.MethodInvocation;
```

```
import org.springframework.aop.support.DelegatingIntroductionInterceptor;
public class ImmutableMixin
    extends DelegatingIntroductionInterceptor implements Immutable {
    private boolean immutable;
    public void setImmutable(boolean immutable) {//设置immutable
        this.immutable = immutable;
    }
    public Object invoke(MethodInvocation mi) throws Throwable {
        //如果调用set方法则抛出异常
        String name = mi.getMethod().getName();
        if (immutable && name.indexOf("set") == 0) {
            throw new IllegalModificationException();
        }
        return super.invoke(mi);
    }
}
```

3.4.2 引入 Advisor

接口：IntroductionAdvisor；实现类：DefaultIntroductionAdvisor；配置实例：

```
<beans>
    <bean id="courseTarget"
        class="com.springinaction.training.model.Course" singleton="false" />
    <bean id="auditableMixin"
        class="com.springinaction.training.advice.AuditableMixin"
        singleton="false" />
    <bean id="auditableAdvisor"
        class="org.springframework.aop.support.DefaultIntroductionAdvisor"
        singleton="false">
        <constructor-arg>
            <ref bean="auditableMixin" />
        </constructor-arg>
    </bean>
    <bean id="course"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyTargetClass">
            <value>true</value>
        </property>
        <property name="singleton">
            <value>false</value>
        </property>
        <property name="proxyInterfaces">
            <value>com.springinaction.training.advice.Auditable</value>
```

```
        </property>
        <property name="auditableAdvisor">
            <list>
                <value>servletAdvisor</value>
            </list>
        </property>
        <property name="target">
            <value ref="courseTarget"/>
        </property>
    </bean>
</beans>
```

3.5 ProxyFactoryBean

表 3.3 ProxyFactoryBean 属性

属性	使用
target	代理的目标对象
proxyInterfaces	代理应实现的接口列表
interceptorNames	需应用到目标对象上的通知 Bean 的名字。可是拦截器、Advisor 或其他通知类型的名字。该属性须按照在 BeanFactory 中使用的顺序设置
singleton	每次调用 getBean 时，工厂是否返回的是同一个代理实例。若使用有状态通知，应设置为 false
aopProxyFactory	使用的 ProxyFactoryBean 实现。Spring 两种实现：JDK 动态代理和 CGLIB。通常不需使用该属性
exposeProxy	目标对象是否需得到当前的代理。通过调用 AopContext.getCurrentProxy 实现。
frozen	一旦工厂被创建，是否可修改代理的通知。当设置为 true 时，在运行时不能修改 ProxyFactoryBean。通常不需使用该属性
optimize	是否对创建的代理进行优化（只适用于 CGLIB）
proxyTargetClass	是否代理目标类而不是实现接口。只能在使用 CGLIB 时使用

3.6 自动代理

自动代理的两个类：BeanNameAutoProxyCreator 和 DefaultAdvisorAutoProxyCreator。

3.6.1 BeanNameAutoProxyCreator

为匹配一系列的 Bean 自动创建代理。

示例如下：为所有服务 Bean 添加 PerformanceThresholdInterceptor（跟踪每个服务方法调用的执行时间，且当执行时间超出给定阀值采取措施）；配置 BeanNameAutoProxyCreator，使所有名字以 Service 结尾的类应用该拦截器。

程序清单 3.19 PerformanceThresholdInterceptor.java（）

```
package chapter03.training.advice;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class PerformanceThresholdInterceptor implements MethodInterceptor {
    private final long thresholdInMillis;//阀值
    public PerformanceThresholdInterceptor(long thresholdInMillis) {//设置阀值
        this.thresholdInMillis = thresholdInMillis;
    }
    public Object invoke(MethodInvocation invocation) throws Throwable {
        long t = System.currentTimeMillis();
        Object o = invocation.proceed();
        t = System.currentTimeMillis() - t;
        if (t > thresholdInMillis) {
            warnThresholdExceeded();
        }
        return o;
    }
    private void warnThresholdExceeded() {
        System.out.println("Danger! Danger!");
    }
}
```

程序清单 3.19（配置 BeanNameAutoProxyCreator，使以 Service 结尾的类应用拦截器）

```
    <bean id="performanceThresholdInterceptor"
        class="chapter03.training.advice.PerformanceThresholdInterceptor">
        <constructor-arg>
            <value>5000</value>
        </constructor-arg>
    </bean>
    <bean id="preformanceThresholdProxyCreator"
    class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyProxy
Creator">
        <bean>
            <property name="beanNames">
                <list>
                    <value>*Service</value>
                </list>
            </property>
            <property name="interceptorNames">
                <!-- interceptorNames属性值是拦截器、通知或Advisor的名字 -->
                <value>performanceThresholdInterceptor</value>
            </property>
        </bean>
    </bean>
```

3.6.2 DefaultAdvisorAutoProxyCreator

为了使用该类，需在配置中定义一个该类的 Bean，该类实现了 BeanPostProcessor 接口，当 ApplicationContext 读入所有 Bean 的配置信息后，该类将扫描上下文，寻找所有的 Advisor，将 Advisor 应用到所有符合 Advisor 切入点的 Bean 中。且该类只能与 Advisor 配合使用。

程序清单 3.20 （配置 DefaultAdvisorAutoProxyCreator，使以 Service 结尾的类应用拦截器）

```
    <bean id="performanceThresholdInterceptor"
        class="chapter03.training.advice.PerformanceThresholdInterceptor">
        <constructor-arg>
            <value>5000</value>
        </constructor-arg>
    </bean>
    <bean id="advisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <bean class="performanceThresholdInterceptor" />
        </property>
        <property name="pattern">
            <value>.+Service\..+</value>
        </property>
    </bean>
    <bean id="autoProxyCreator"
    class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProx
yCreator" />
```

3.6.3 元数据自动代理

代理配置是通过源代码属性而不是外部配置文件获得的。常见的是声明式事务支持。

4 数据库支持

4.1 Spring 的 DAO 理念

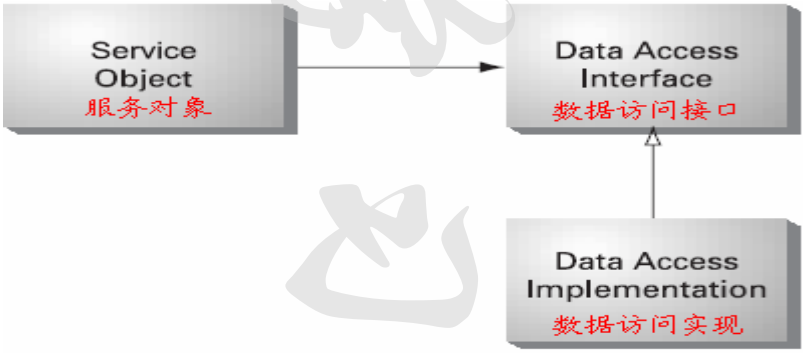


图4.1 服务对象应依赖接口访问数据

DAO：数据访问对象（Data Access Object）

4.1.1 DataAccessException

表 4.1 Spring 的 DAO 异常层次

学习比较

异常	何时抛出
CleanupFailureDataAccessException	一项操作成功执行但释放数据库资源时发生异常（如关闭 Connection）
DataAccessResourceFailureException	数据访问资源彻底失败，如不能连接数据库
DataIntegrityViolationException	Insert 或 Update 数据时违反了完整性，如违反唯一性限制
DataRetrievalFailureException	某些数据不能被检索到，如不能通过关键字找到一条记录
DeadlockLoserDataAccessException	当前操作因死锁而失败
IncorrectUpdateSemanticsDataAccessException	Update 时发生某些没有预料到的情况，如更改超过预期的记录数。当这个异常被抛出时，执行着的事务不会被回滚
InvalidDataAccessApiUsageException	一个数据访问的 Java API 没有正确使用，如必须在执行前编译好的查询编译失败
InvalidDataAccessResourceUsageException	错误使用数据访问资源，如用错误的 SQL 语法访问关系型数据库
OptimisticLockingFailureException	乐观锁的失败。将由 ORM 工具或用户的 DAO 实现抛出
TypeMismatchDataAccessException	Java 类型和数据类型不匹配，如试图将 String 类型插入到数据库的数值型字段中
UncategorizedDataAccessException	有错误发生，但无法归类到某一更为具体的异常中

Spring 的 DAO 框架抛出的异常都是该异常的子类。该异常是 RuntimeException，也是 Spring 的 NestedRuntimeException 的子类，可通过 NestedRuntimeException 的 getCause()方法获得导致该异常的另一个异常。

4.1.2 使用 DataSource

程序清单 4.1（从 JDNI 得到 DataSource）

```
<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
        <value>java:comp/env/jdbc/myDatasource</value>
    </property>
</bean>
```

程序清单 4.2（创建 DataSource 连接池）

```
<bean id="ds" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName"
        value="${jdbc.stats.driverClassName}" />
    <property name="url" value="${jdbc.stats.url}" />
    <property name="username" value="${jdbc.stats.username}" />
    <property name="password" value="${jdbc.stats.password}" />
</bean>
```

4.1.3 一致的 DAO 支持

模板设计模式（Template Method Pattern）。

数据访问固定步骤：数据库建立连接、操作完释放资源。

Spring 把数据访问流程中固定部分和可变部分分开，分别映射为两个不同的类：模板（Template）和回调（Callback）。模板管理流程的固定部分，回调处填写实现细节。

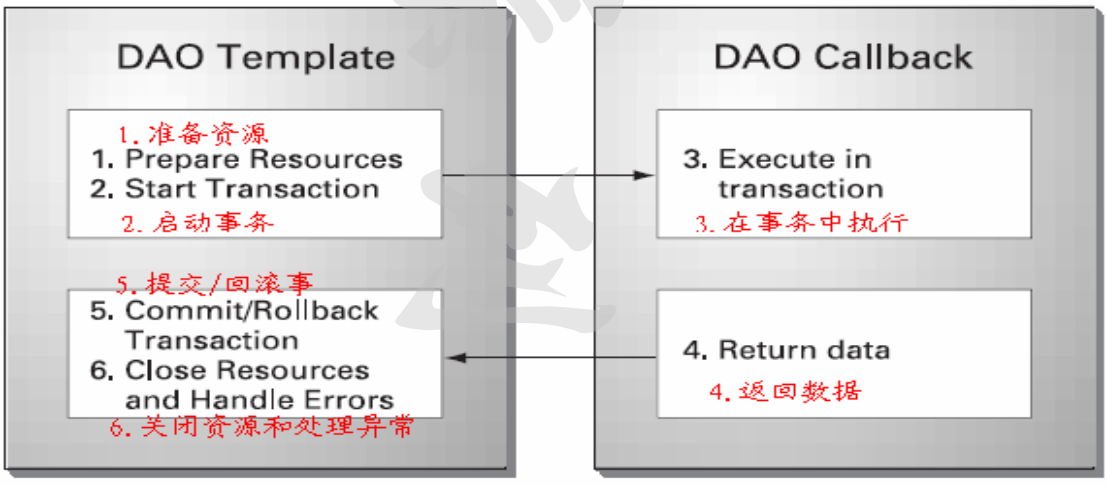


图4.2 Spring的DAO模板和回调类的职责

如图 4.2 所示，Spring 的模板类处理数据访问的不变部分：事务控制、资源管理、异常处理。回调接口的实现定义了特定于应用的部分：创建 statement、绑定参数、整理结果集（Result Set）。

在模板——回调（template-callback）之上提供一个支撑类，以便应用的数据访问类继承。图 4.3 描述了应用的数据访问类、支撑类和模板类之间的关系：

图4.3 持久化API、模板类、DAO支持类和应用DAO类之间的关系

4.2 Spring 中使用 JDBC

4.2.1 JdbcTemplate

程序清单 4.3（将 JdbcTemplate 连接到 DAO bean 中）

```
<bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource">
        <ref bean="dataSource" />
    </property>
</bean>
<bean id="studentDao" class="StudentDaoJdbc">
    <property name="jdbcTemplate">
        <ref bean="jdbcTemplate" />
    </property>
</bean>
<bean id="courseDao" class="CourseDaoJdbc">
    <property name="jdbcTemplate">
        <ref bean="jdbcTemplate" />
    </property>
</bean>
```

4.2.1.1写数据

一些相关接口：

- PreparedStatementCreator：接口的实现负责创建 PreparedStatement
- SqlProvider：接口的实现向 JdbcTemplaye 类提供 SQL 语句字符串
- PreparedStatementSetter：接口的实现接收 PreparedStatement，并负责设置参数

- BatchPreparedStatementSetter：接口的实现接收 PreparedStatement，并负责批量设置参数

程序清单 4.4（使用 JdbcTemplate.execute 方法插入数据）

```
public int insertPerson(Person person) {
    //创建SQL
    String sql = "insert into person(id,firstName,lastName) values(?,?,?)";
    //设置参数
    Object[] params = new Object[] { person.getId(), person.getFirstName(),
            person.getLastName() };
    //设置数据类型
    int[] types = new int[] { Types.INTEGER, Types.VARCHAR, Types.VARCHAR };
    //执行statement
    return jdbcTemplate.update(sql, params, types);
}
```

程序清单 4.5（使用 BatchPreparedStatementCreator 插入多个对象）

```
public int[] updatePersons(final List persons) {
    // 创建SQL
    String sql = "insert into person(id,firstName,lastName) values (?,?,?)";
    BatchPreparedStatementSetter setter = null;
    setter = new BatchPreparedStatementSetter() {
        // 定义批量语句的数量
        public int getBatchSize() {
            return persons.size();
        }
        // 设置参数
        public void setValues(PreparedStatement ps, int index)
                throws SQLException {
            Person person = (Person) persons.get(index);
            ps.setInt(0, person.getId().intValue());
            ps.setString(1, person.getFirstName());
            ps.setString(2, person.getLastName());
        }
    };
    // 执行statement
    return jdbcTemplate.batchUpdate(sql, setter);
}
```

4.2.1.2读数据

相关接口：

- RowCallbackHandler
- RowMapper

- RowMapperResultSetExtractor

程序清单 4.6（使用 RowCallbackHandler 执行一个查询）

```
    public Person getPerson(final Integer id) {
        // 创建SQL
        String sql = "select id, first_name, last_name from person where id = ?";
        // 创建查询结果对象
        final Person person = new Person();
        // 创建查询参数
        final Object[] params = new Object[] { id };
        jdbcTemplate.query(sql, params, new RowCallbackHandler() {
            // 处理查询结果
            public void processRow(ResultSet rs) throws SQLException {
                person.setId(new Integer(rs.getInt("id")));
                person.setFirstName(rs.getString("first_name"));
                person.setLastName(rs.getString("last_name"));
            }
        });
        return person;// 返回查询结果对象
    }
```

程序清单 4.7（把 ResultSet 中的一条记录映射成一个对象）

```
class PersonRowMapper implements RowMapper {
    public Object mapRow(ResultSet rs, int index) throws SQLException {
        List all = new ArrayList();
        Person person = new Person();
        person.setId(new Integer(rs.getInt("id")));
        person.setFirstName(rs.getString("first_name"));
        person.setLastName(rs.getString("last_name"));
        all.add(person);
        return all;
    }
}
```

程序清单 4.8（使用自定义的 RowMapper）

```
    public List getAllPersons() {
        String sql = "select id, first_name, last_name from person";
        return (List) jdbcTemplate.query(sql, new RowMapperResultSetExtractor(
                new PersonRowMapper()));
    }
```

程序清单 4.9（使用自定义的 RowMapper）

```
    public Person getPerson(final Integer id) {
        String sql = "select id, first_name, last_name from person where id = ?";
```

```
        final Object[] params = new Object[] { id };
        List list = (List) jdbcTemplate.query(sql, params,
                new RowMapperResultSetExtractor(new PersonRowMapper()));
        return (Person) list.get(0);
    }
```

4.2.1.3调用存储过程

程序清单 4.10（通过 CallableStatementCallback 执行存储过程）

```
    public void archiveStudentData() {
        CallableStatementCallback cb = new CallableStatementCallback() {
            public Object doInCallableStatement(CallableStatement cs)
                    throws SQLException {
                cs.execute();
                return null;
            }
        };
        jdbcTemplate.execute("{ ARCHIVE_STUDENTS }", cb);
    }
```

说明：ARCHIVE_STUDENTS 为存储过程名字

学习比较

4.2.2 把操作创建成对象

4.2.2.1SqlUpdate

程序清单 4.11 InsertPerson.java（数据库操作对象，插入 Person）

```
package chapter04;
import java.sql.Types;
import javax.sql.DataSource;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;
public class InsertPerson extends SqlUpdate {
    public InsertPerson(DataSource ds) {
        setDataSource(ds);//设置DataSource以创建JdbcTemplate
        setSql("insert into person(id,firstName,lastName) values (?, ?, ?)");
        //为每个参数调用declareParameter()且顺序与SQL中一致
        declareParameter(new SqlParameter(Types.NUMERIC));
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.VARCHAR));
        //数据库操作对象必须在其使用前编译好
        compile();
```

```
    }
    public int insert(Person person) {
        Object[] params = new Object[] { person.getId(), person.getFirstName(),
                person.getLastName() };
        return update(params);
    }
}
```

程序清单 4.12（使用数据库操作对象 InsertPerson 插入数据）

```
    private InsertPerson insertPerson;
    public int insertPerson(Person person) {
        return insertPerson.insert(person);
    }
```

4.2.2.2MappingSqlQuery 查询数据库

程序清单 4.13 PersonByIdQuery.java（数据库操作对象，查询数据）

```
package chapter04;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;
import javax.sql.DataSource;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.MappingSqlQuery;
public class PersonByIdQuery extends MappingSqlQuery {
    public PersonByIdQuery(DataSource ds) {
        super(ds, "select id, first_name, last_name from person where id = ?");
        declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }
    public Object mapRow(ResultSet rs, int rowNumber) throws SQLException {
        Person person = new Person();
        person.setId((Integer) rs.getObject("id"));
        person.setFirstName(rs.getString("first_name"));
        person.setLastName(rs.getString("last_name"));
        return person;
    }
}
```

程序清单 4.14（使用数据库操作对象查询数据）

```
    private PersonByIdQuery personByIdQuery;
    public Person getPerson(Integer id) {
```

```
        Object[] params = new Object[] { id };
        return (Person) personByIdQuery.execute(params).get(0);
    }
```

4.2.3 自增键

接口：DataFieldMaxValueIncrementer。

程序清单 4.15（使用数据库操作对象查询数据）

```
    private DataFieldMaxValueIncrementer incrementer;
    public void setIncrementer(DataFieldMaxValueIncrementer incrementer) {
        this.incrementer = incrementer;
    }
    public void insertPerson(Person person) {
        Integer id = new Integer(incrementer.nextIntValue());
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        String sql = "insert into person(id,firstName,lastName) values (?,?,?)";
        Object[] params = new Object[] { id, person.getFirstName(),
                person.getLastName() };
        jdbcTemplate.update(sql, params);
        // everything was successful
        person.setId(id);
    }
```

4.3 Spring 的 ORM 框架支持

ORM 框架的部分功能：

- 延迟加载（Lazy loading）：
- 主动抓取（Eager fetching）：
- 缓存（Caching）：
- 级联（Cascading）：

Spring 附加的服务：

- 整合事务管理
- 异常处理
- 线程安全，轻量级模板类
- 便利的支持类
- 资源管理

4.3.1 整合 Hibernate

4.3.1.1Hibernate 简介

程序清单 4.16（使用数据库操作对象查询数据）

```
package chapter04;
import java.util.Set;
public class Student {
    private Integer id;
    private String firstName;
    private String lastName;
    private Set courses;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public Set getCourses() {
        return courses;
    }
    public void setCourses(Set courses) {
        this.courses = courses;
    }
}
```

程序清单 4.17（Hibernate 映射文件 Student.hbm.xml）

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
```

```
<hibernate-mapping>
    <!-- 定义需映射的类 -->
    <class name="org.springinaction.training.model.Student">
        <!-- 映射主键 -->
        <id name="id">
            <generator class="assigned" />
        </id>
        <!-- 映射属性 -->
        <property name="sex" />
        <property name="weight" />
        <!-- 映射关系 -->
        <set name="courses" table="transcript">
            <key column="student_id" />
            <many-to-many column="course_id"
                class="org.springinaction.training.model.Course" />
        </set>
    </class>
</hibernate-mapping>
```

程序清单 4.18（通过主键取得一个 Student 对象的代码）

```
    private SessionFactory sessionFactory;
    public Student getStudent(Integer id) throws HibernateException {
        Session session = sessionFactory.openSession();
        Student student = (Student) session.load(Student.class, id);
        session.close();
        return student;
    }
```

4.3.1.2管理 Hibernate 资源

相关类：LocalSessionFactoryBean

程序清单 4.19（配置 SessionFactory）

```
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="dataSource">
            <ref bean="dataSource" />
        </property>
    </bean>
```

程序清单 4.20（配置 SessionFactory，详细示例）

```
    <!-- Hibernate SessionFactory -->
    <bean id="sessionFactory"
```

```
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="lobHandler" ref="oracleLobHandler"/>
        <property name="mappingResources">
            <list>
                <value>stat2/user/domain/User.hbm.xml</value>
                <value>stat2/common/message/domain/Message.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">${hibernate.dialect}</prop>
                <prop key="hibernate.show_sql">false</prop>
                <prop key="hibernate.jdbc.batch_size">20</prop>
                <prop key="hibernate.jdbc.fetch_size">20</prop>
                <prop key="hibernate.generate_statistics">true</prop>
            </props>
        </property>
        <property name="eventListeners">
            <map>
                <entry key="merge">
                    <bean
class="org.springframework.orm.hibernate3.support.IdTransferringMergeEventLi
stener"/>
                </entry>
            </map>
        </property>
    </bean>
```

4.3.1.3使用 HibernateTemplate

Hibernate 中的模板回调机制包含 HibernateTemplate 和回调接口 HibernateCallback。

程序清单 4.21（配置 HibernateTemplate）

```
    <bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate.HibernateTemplate">
        <property name="sessionFactory">
            <ref bean="sessionFactory" />
        </property>
    </bean>
```

程序清单 4.22（使用 HibernateTemplate 和 HibernateCallback 访问数据库）

```
    public Student getStudent(final Integer id) {
        return (Student) hibernateTemplate.execute(new HibernateCallback() {
```

```
            public Object doInHibernate(Session session)
                    throws HibernateException {
                return session.load(Student.class, id);
            }
        });
    }
```

程序清单 4.23（使用 HibernateTemplate 简洁代码）

```
    public Student getStudent(Integer id) {
        return (Student) hibernateTemplate.load(Student.class, id);
    }
```

4.3.1.4HibernateDaoSupport

5 事务管理

5.1 理解事务

5.1.1 事务特性

学习比较

ACID：

- 原子性（Atomic）：要么都发生，要么都不发生
- 一致性（Consistent）：事务结束后，系统所处的状态与其业务规则一致
- 隔离性（Isolated）：
- 持久性（Durable）：事务完成，结果应持久化

5.1.2 Spring 的事务管理器

事务管理器实现	目标
org.springframework.jdbc.datasource.DataSourceTransactionManager	在单一的 JDBC DataSource 中管理事务
org.springframework.orm.hibernate.HibernateTransactionManager	当持久化机制是 Hibernate 时，用其管理事务
org.springframework.orm.jdo.JdoTransactionManager	当 JDO 用作持久化时，用其管理事务
org.springframework.transaction.jta.JtaTransactionManager	使用 JTA 实现事务。在事务跨越多个资源时必须使用
org.springframework.orm.ojb.PersistenceBrokerTransactionManager	当 Apache 的 OJB 用作持久化时，用其管理事务

每种事务管理器是对特定平台的事务实现的代理（图 5.2）

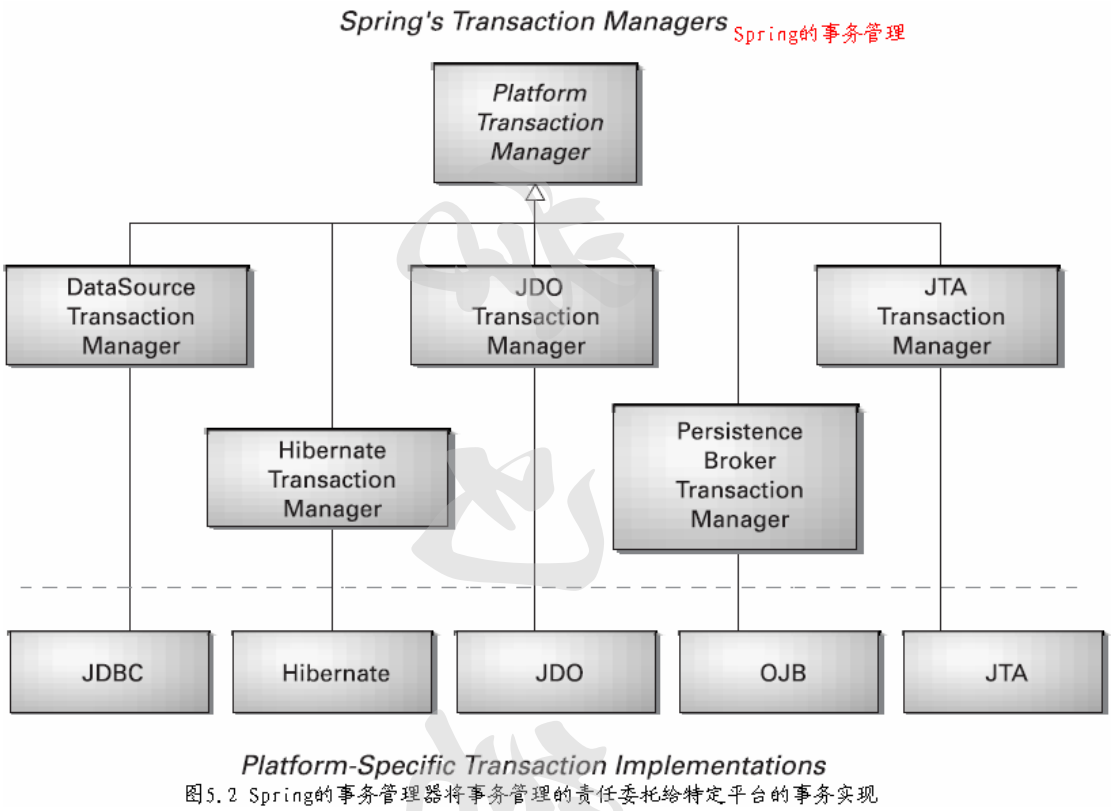


图5.2 Spring的事务管理器将事务管理的责任委托给特定平台的事务实现

程序清单 5.1（JDBC 事务）

```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource">
        <ref bean="dataSource" />
    </property>
</bean>
```

程序清单 5.2（Hibernate 事务）

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref bean="sessionFactory" />
    </property>
</bean>
```

程序清单 5.3（JDO 事务）

```
<bean id="transactionManager"
    class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory">
        <ref bean="persistenceManagerFactory" />
    </property>
</bean>
```

程序清单 5.4（OJB 事务）

```
<bean id="transactionManager"
class="org.springframework.orm.ojb.PersistenceBrokerTransactionManager">
</bean>
```

程序清单 5.5（JTA 事务）

```
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManagerName">
        <value>java:/TransactionManager</value>
    </property>
</bean>
```

5.2 编程事务

程序清单 5.6（在 enrollStudentInCourse()方法中的程序控制事务）

```
public void enrollStudentInCourse() {
    transactionTemplate.execute(new TransactionCallback() {
        public Object doInTransaction(TransactionStatus ts) {
            try {
                // do stuff 在doInTransaction()方法中运行
            } catch (Exception e) {
                ts.setRollbackOnly();//调用setRollbackOnly()回滚
            }
            return null;//如果成功，事务被提交
        }
    });
}
```

程序清单 5.6（注入 transactionTemplate）

```
<bean id="transactionTemplate"
    class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager">
        <ref bean="transactionManager" />
    </property>
</bean>
<bean id="courseService"
    class="com.springinaction.training.service.CourseServiceImpl">
    <property name=" transactionTemplate">
        <ref bean=" transactionTemplate" />
    </property>
</bean>
```

5.3 声明式事务

程序清单 5.7（为服务创建代理，以便进行事务处理）

```
    <bean id="courseServiceTarget"
        class="com.springinaction.training.service.CourseServiceImpl">
        <property name=" transactionTemplate">
            <ref bean=" transactionTemplate" />
        </property>
    </bean>
    <bean id="courseService"
    class="org.springframework.transaction.interceptor.TransactionProxyFactor
yBean">
        <property name="proxyInterfaces">
            <list>
                <value>
                    <!-- 代理所实现的接口 -->
                    com.springinaction.training.service.CourseService
                </value>
            </list>
        </property>
        <property name="target">
            <!-- 被代理的对象 -->
            <ref bean="courseServiceTarget" />
        </property>
        <property name="transactionManager">
            <!-- 事务管理器 -->
            <ref bean="transactionManager" />
        </property>
        <property name="transactionAttributeSource">
            <!-- 事务属性源 -->
            <ref bean="attributeSource" />
        </property>
    </bean>
```

5.3.1 事务属性

事务属性是对事务策略如何应用到方法的描述，包括如下：

- 传播行为
- 隔离级别
- 只读提示
- 事务超时间隔

5.3.1.1传播行为

表 5.2 Spring 的传播规则

传播行为	意义
PROPAGATION_MANDATORY	支持当前事务，如果当前没有事务，就抛出异常。
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行。被嵌套的事务可从当前事务中单独地提交或回滚。如果当前没有事务，则进行与 PROPAGATION_REQUIRED 类似的操作
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。如使用 JTATransactionManager，则需访问 TransactionManager
PROPAGATION_REQUIRED	支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。如使用 JTATransactionManager，则需访问 TransactionManager
PROPAGATION_SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行。

学习比较

5.3.1.2隔离级别

并发存在的问题：

- 脏读（Dirty read）：一个事务读取了另一个事务修改但未提交的数据
- 不可重复读（Nonrepeatable read）：一个事务执行相同的查询两次或两次，但每次查询结果都不相同时。
- 幻读（Phantom read）：一个事务（T1）读取几行记录后，另一并发事务（T2）插入一些记录时

表 5.3 Spring 的事务隔离级别

隔离级别	意义
ISOLATION_DEFAULT	使用后端数据库默认的隔离级别
ISOLATION_READ_UNCOMMITTED	允许读取未提交的改变了的数据。可能导致脏读、不可重复读、幻读
ISOLATION_READ_COMMITTED	允许在并发事务已经提交后读取。可防止脏读，但不可重复读、幻读仍可发生
ISOLATION_REPEATABLE_READ	对相同字段的多次读取的结果是一致的，除非数据被事务本身改变。可防止脏读、不可重复读，但幻读仍可发生
ISOLATION_SERIALIZABLE	完全服从 ACID 的隔离级别，确保不发生脏读、不可重复读、幻读，但是最慢的。典型的通过完全锁定在事务中涉及到的数据表来完成的

其他说明：

- 将可能启动新事务的传播行为（PROPAGATION_REQUIRED，PROPAGATION_REQUIRES_NEW，PROPAGATION_NESTED）的事务设置为只读或事务超时可进行优化。
- Hibernate 设置只读：将 Hibernate 的 flush 模式设置为 FLUSH_NEVER

5.3.2 简单示例

相关接口和类：

- TransactionProxyFactoryBean：参照一个方法的事务属性，决定如何在该方法上执行事务策略
- TransactionAttributeSource：作为在方法上查找事务属性的一个参考

事务属性如果没有明确说明，默认为：PROPAGATION_REQUIRED 和 ISOLATION_DEFAULT。

程序清单 5.8（改变默认事务属性）

```
    <!-- 定义定制的事务属性 -->
    <bean id="myTransactionAttribute"
    class="org.springframework.transaction.interceptor.DefaultTransactionAttr
ibute">
        <property name="propagationBehaviorName">
            <!-- 设置传播行为 -->
            <value>PROPAGATION_REQUIRES_NEW</value>
        </property>
        <property name="isolationLevelName">
            <!-- 设置隔离级别 -->
            <value>ISOLATION_REPEATABLE_READ</value>
        </property>
    </bean>
    <bean id="transactionAttributeSource"
    class="org.springframework.transaction.interceptor.MatchAlwaysTransaction
AttributeSource">
        <property name="transactionAttribute">
            <!-- 改变默认事务属性 -->
            <ref bean="myTransactionAttribute" />
        </property>
    </bean>
```

5.4 通过方法名声明事务

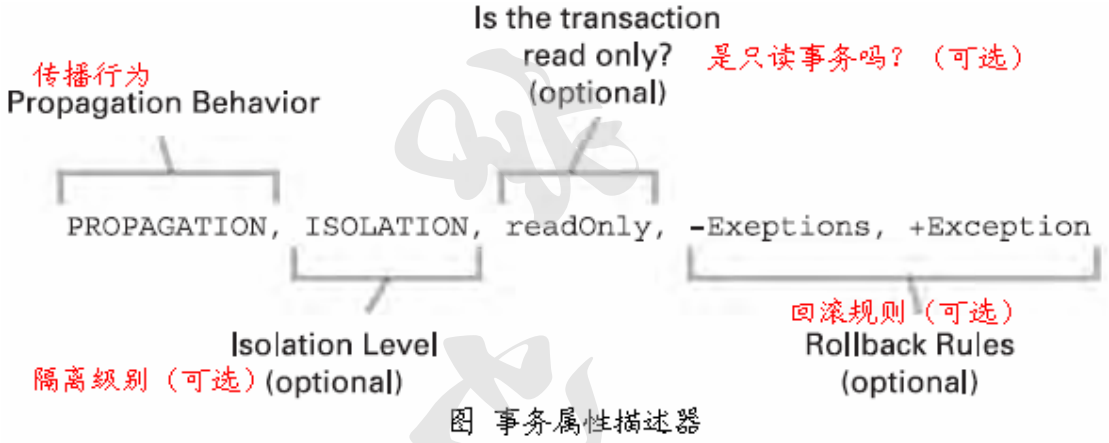


图 事务属性描述器

相关接口或类：NameMatchTransactionAttributeSource。

程序清单 5.9（使用 NameMatchTransactionAttributeSource）

```
    <bean id="transactionAttributeSource"
    class="org.springframework.transaction.interceptor.NameMatchTransactionAt
tributeSource">
        <property name="properties">
            <props>
                <prop key="enrollStudentInCourse">
                    PROPAGATION_REQUIRES_NEW,ISOLATION_REPEATABLE_READ,
                    -CourseException
                </prop>
            </props>
        </property>
    </bean>
```

程序清单 5.10（使用通配符匹配）

```
    <bean id="transactionAttributeSource"
    class="org.springframework.transaction.interceptor.NameMatchTransactionAt
tributeSource">
        <property name="properties">
            <props>
                <prop key="get*">PROPAGATION_SUPPORTS</prop>
            </props>
        </property>
    </bean>
```

程序清单 5.11（名称匹配事务的捷径）

```
    <bean id="courseService"
    class="org.springframework.transaction.interceptor.TransactionProxyFactor
yBean">
```

```
        <property name="transactionProperties">
            <props>
                <prop key="enrollStudentInCourse">
                    PROPAGATION_REQUIRES_NEW
                </prop>
            </props>
        </property>
    </bean>
```

5.5 用元数据声明事务

5.6 修剪事务声明

程序清单 5.12（）

```
    <bean id="courseService"
    class="org.springframework.transaction.interceptor.TransactionProxyFactor
yBean">
        <property name="target">
            <ref bean="courseServiceTarget" />
        </property>
        <property name="transactionManager">
            <ref bean="transactionManager" />
        </property>
        <property name="transactionAttributeSource">
            <ref bean="attributeSource" />
        </property>
    </bean>
    <bean id="courseServiceTarget"
        class="com.springinaction.training.service.CourseServiceImpl">
    </bean>
```

两种方法：

- Bean 继承
- AOP 自动代理

5.6.1 继承父 TransactionProxyFactoryBean

程序清单 5.13（首先在 XML 配置中增加 TransactionProxyFactoryBean 的 abstract 声明）

```
    <bean id="abstractTxDefinition"
    class="org.springframework.transaction.interceptor.TransactionProxyFactor
yBean"
        lazy-init="true">
```

```
        <property name="transactionManager">
            <ref bean="transactionManager" />
        </property>
        <property name="transactionAttributeSource">
            <ref bean="attributeSource" />
        </property>
    </bean>
```

说明：lazy-init 属性设置为 true 告诉容器不用创建 bean，除非应用请求；可把该 bean 变成抽象的

程序清单 5.14（创建子 bean）

```
    <bean id="studentService" parent="abstractTxDefinition">
        <property name="target">
            <bean
    class="com.springinaction.training.service.StudentServiceImpl" />
        </property>
    </bean>
```

5.6.2 自动代理事务

程序清单 5.15（首先配置 DefaultAdvisorAutoProxyCreator 实例）

```
    <bean id="autoproxy"
    class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProx
yCreator">
    </bean>
```

程序清单 5.16（首先配置 TransactionAttributeSourceAdvisor 实例，根据事务属性源确定方法是否和事务属性关联）

```
    <bean id="transactionAdvisor"
    class="org.springframework.transaction.interceptor.TransactionAttributeSo
urceAdvisor">
        <constructor-arg>
            <ref bean="transactionInterceptor" />
        </constructor-arg>
    </bean>
```

程序清单 5.17（拦截器）

```
    <bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor
">
        <property name="transactionManager">
            <ref bean="transactionManager" />
        </property>
```

```
        <property name="transactionAttributeSource">
            <ref bean="transactionAttributeSource" />
        </property>
    </bean>
```

程序清单 5.18（为自动代理选择属性源）

```
    <bean id="transactionAttributeSource"
    class="org.springframework.transaction.interceptor.NameMatchTransactionAt
tributeSource">
        <property name="properties">
            <props>
                <prop key="get*v">PROPAGATION_SUPPORTS</prop>
            </props>
        </property>
    </bean>
```

说明：使用程序清单 5.18 配置时，所有名字以“get”开始的方法（应用上下文中所有 Bean）被代理。可使用程序清单 5.19 进行具体化。

程序清单 5.19（为自动代理选择属性源）

```
    <bean id="transactionAttributeSource"
    class="org.springframework.transaction.interceptor.MethodMapTransactionAt
tributeSource">
        <property name="methodMap">
            <map>
                <entry
            key="com.springinaction.training.service.CourseServiceImpl.get*">
                    <value>PROPAGATION_SUPPORTS</value>
                </entry>
            </map>
        </property>
    </bean>
```