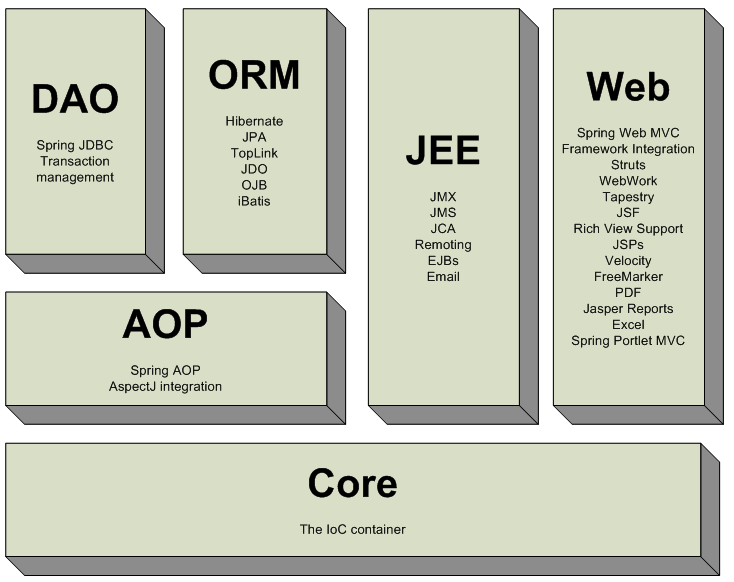
Spring课程

# Spring基本特征



Spring是一个非常活跃的开源框架；它是一个基于Core来构架多层JavaEE系统的框架,它的主要目地是简化企业开发.

Spring以一种非侵入式的方式来管理你的代码，Spring提倡”最少侵入”，这也就意味着你可以适当的时候安装或卸载Spring

# 开发spring所需要的工具

## Spring的jar包

到http://www.springsource.org/download下载spring，然后进行解压缩，在解压目录中找到下面jar文件，拷贝到类路径下

--spring的核心类库 在spring文档的dist下

dist\spring.jar

--引入的第三方类库 都spring文档的lib下

lib\jakarta-commons\commons-logging.jar

如果使用了切面编程(AOP),还需要下列jar文件

lib/aspectj/aspectjweaver.jar和aspectjrt.jar

lib/cglib/cglib-nodep-2.1\_3.jar

如果使用了JSR-250中的注解,如@Resource/@PostConstruct/@PreDestroy,还需要下列jar文件

lib\j2ee\common-annotations.jar

注:JSR(Java 规范请求)是指向JCP(Java Community Process)提出新增一个标准化技术规范的正式请求。任何人都可以提交JSR(Java 规范请求)，以向Java平台增添新的API和服务。JSR已成为Java界的一个重要标准

## Spring配置文件

默认情况下是applicationContext.xml文件。可以建立很多xml文件，工程中一般都是这样配置的。

# Spring基本功能详解

## SpringIOC

Spring的控制反转：把对象的创建、初始化、销毁等工作交给spring容器来做。由spring容器控制对象的生命周期。

步骤：

1. 启动spring容器
2. 在类路径下寻找配置文件来实例化容器

ApplicationContext *ctx =* new ClassPathXmlApplicationContext(new String[]{"beans.xml"});可以在整个类路径中寻找xml文件

\* 通过这种方式加载。需要将spring的配置文件放到当前项目的classpath路径下

\* classpath路径指的是当前项目的src目录，该目录是java源文件的存放位置。

1. 在文件系统路径下寻找配置文件来实例化容器

ApplicationContext *ctx =* new FileSystemXmlApplicationContext(new String[]{“d:\\beans.xml“});Spring的配置文件可以指定多个，可以通过String数组传入。

注：经常用第一种方法启动容器

1. 从spring容器中提取对象

## 别名

<beans>

<alias name="person" alias="p"/>

<bean name="person" class="cn.itcast.aliasspring.Person"/>

</beans>

通过这样的配置，可以达到在一个地方命名，在多个地方使用不同的名字的效果。

## Spring容器内部对象

### 创建对象的方式

#### 无参构造函数

<bean id=“personService" class="cn.itcast.bean.impl.PersonServiceImpl"/>

#### 静态工厂

<bean id="personService" class="com.itcast.factory.PersonServiceFactory" factory-method="createPersonService" />

public class PersonServiceFactory {

public static PersonService createPersonService(){

return new PersonServiceImpl();

}

}

#### 实例工厂

记住概念即可

### 对象的scope

#### singleton(默认值)

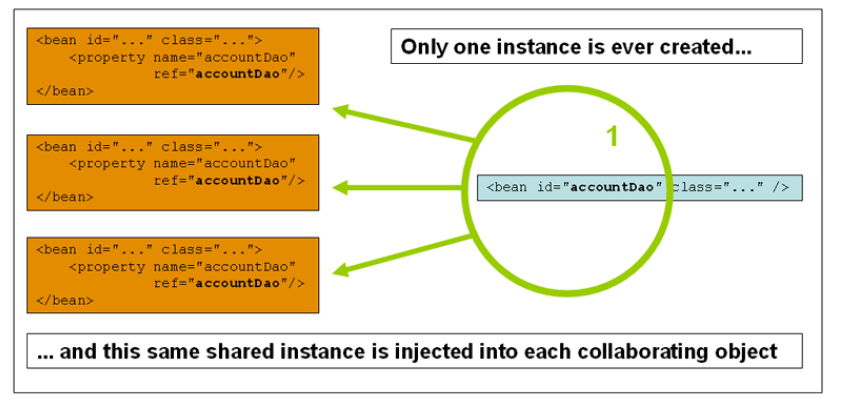
在每个Spring IoC容器中一个bean定义只有一个对象实例(共享)。

默认情况下会在容器启动时初始化bean，但我们可以指定Bean节点的lazy-init=“true”来延迟初始化bean，这时候，只有第一次获取bean会才初始化bean。如：

<bean id="xxx" class="cn.itcast.OrderServiceBean" lazy-init="true"/>

如果想对所有bean都应用延迟初始化，可以在根节点beans设置default-lazy-init=“true“，如下：

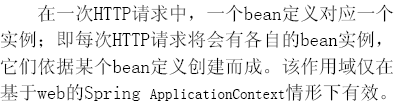
<beans default-lazy-init="true“ ...>



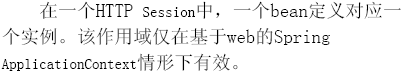
#### prototype

允许bean可以被多次实例化(使用一次就创建一个实例) . Spring不能对一个prototype bean的整个生命周期负责.这就意味着清楚prototype作用域的对象并释放任何prototype bean所持有的昂贵资源都是客户端的责任。

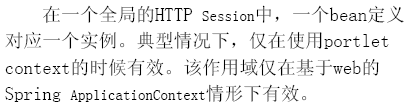
#### Request



#### Session



#### Global session



### 初始化bean时机

Spring默认在启动时将所有singleton bean提前进行实例化。提前实例化意味着作为初始化的一部分，ApplicationContext会自动创建并配置所有的singleton bean.通常情况下这是件好事。因为这样在配置中有任何错误能立即发现。

Lazy-init=”true or false”

Lazy-init 为false,spring容器将在启动的时候报错（比较好的一种方式）

Lazy-init 为true,spring容器将在调用该类的时候出错。

### init、destroy

Spring初始化bean或销毁bean时，有时需要作一些处理工作，因此spring可以在创建和拆卸bean的时候调用bean的两个生命周期方法。

<bean id=“foo” class=“...Foo”

init-method=“setup”

destory-method=“teardown”/>

当foo被载入到Spring容器中时调用init-method方法。当foo从容器中删除时调用destory-method（scope = singleton有效）

## 依赖注入(DI)

### xml形式

#### 使用构造器注入

使用xml的注入方式

1. 通过参数的顺序

<constructor-arg index="0">

<value>张三</value>

</constructor-arg>

<constructor-arg index="1">

<value>56</value>

</constructor-arg>

1. 通过参数的类型

<constructor-arg type="java.lang.Integer">

<value>56</value>

</constructor-arg>

<constructor-arg type="java.lang.String">

<value>张三</value>

</constructor-arg>

#### 使用属性setting方法进行注入

使用xml的注入方式：

1. 简单Bean的注入

简单Bean包括两种类型：包装类型和String

<bean id="personService" class="com.itcast.bean.impl.PersonServiceImpl">

<!-- 基本类型,string类型 -->

<property name="age" value="20"></property>

<property name="name" value="张无忌"></property> </bean>

1. 引用其他Bean

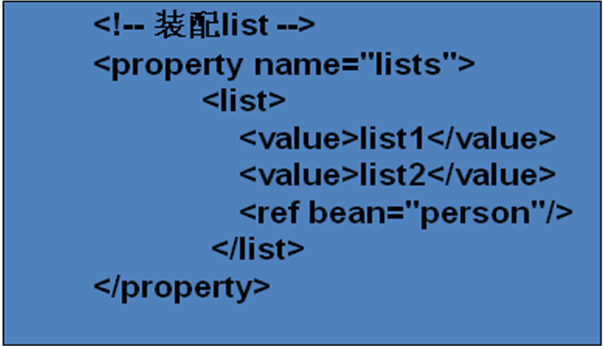
<bean id="person" class="com.itcast.bean.Person" />

<bean id="personService" class="com.itcast.bean.impl.PersonServiceImpl">

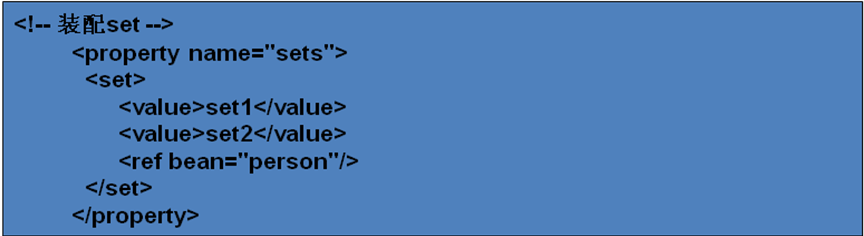
<property name="person" ref="person" />

</bean>

#### 装配list集合



#### 装配set集合



#### 装配map

<property name="maps">

<map>

<entry key="01">

<value>map01</value>

</entry>

<entry key="02">

<value>map02</value>

</entry>

</map>

</property>

map中的<entry>的数值和<list>以及<set>的一样，可以使任何有效的属性元

素，需要注意的是key值必须是String的。

#### 装配Properties

<property name="props">

<props>

<prop key="01">prop1</prop>

<prop key="02">prop2</prop>

</props>

</property>

### 注解

步骤：

1. 在配置文件中，引入context命名空间

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

**xmlns:context="http://www.springframework.org/schema/context"**

xsi:schemaLocation="http://www.springframework.org/schema/beans

<http://www.springframework.org/schema/beans/spring-> beans-2.5.xsd

**http://www.springframework.org/schema/context**

**<http://www.springframework.org/schema/context/spring-context-2.5.xsd>**">

1. 在配置文件中加入context:annotation-config标签

<context:annotation-config/>

这个配置隐式注册了多个对注释进行解析处理的处理器

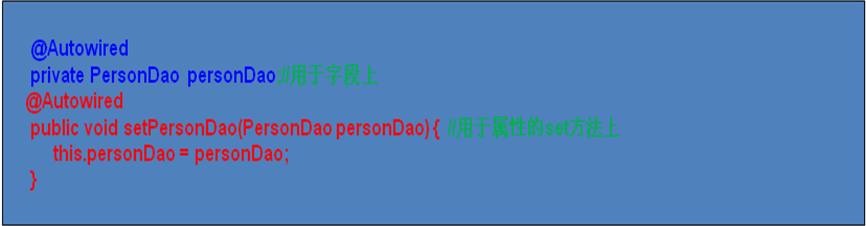
AutowiredAnnotationBeanPostProcessor，CommonAnnotationBeanPostProcessor，

PersistenceAnnotationBeanPostProcessor，RequiredAnnotationBeanPostProcessor

注： @Resource注解在spring安装目录的lib\j2ee\common-annotations.jar

#### @Autowired

这两个注解的区别是：@Autowired 默认按类型装配，@Resource默认按名称装配，当找不到与名称匹配的bean才会按类型装配。



@Autowired注解是按类型装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它required属性为false。



#### @Qualifier

如果我们想使用按名称装配，可以结合@Qualifier注解一起使用。如下：



#### @Resource

1. @Resource注解和@Autowired一样，也可以标注在字段或属性的setter方法上.
2. @Resource注解默认按名称装配。

名称可以通过@Resource的name属性指定，如果没有指定name属性，

* 当注解标注在字段上，即默认取字段的名称作为bean名称寻找依赖对象
* 当注解标注在属性的setter方法上，即默认取属性名作为bean名称寻找依赖对象。



* 注意：如果没有指定name属性，并且按照默认的名称找不到依赖对象时， @Resource注解会回退到按类型装配。但一旦指定了name属性，就只能按名称装配了。

#### @PostConstruct

指定Bean的初始化方法

#### @PreDestroy

指定Bean的销毁方法

## 扫描

前面的例子我们都是使用XML的bean定义来配置组件。在一个稍大的项目中，通常会有上百个组件，如果这些组件采用xml的bean定义来配置，显然会增加配置文件的体积，查找及维护起来也不太方便。spring2.5为我们引入了组件自动扫描机制，它可以在类路径底下寻找标注了@Component、@Service、@Controller、@Repository注解的类，并把这些类纳入进spring容器中管理。它的作用和在xml文件中使用bean节点配置组件是一样的。要使用自动扫描机制，我们需要打开以下配置信息:

**1、引入context命名空间 需要在xml配置文件中配置以下信息：**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

**xmlns:context="http://www.springframework.org/schema/context"**

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

**http://www.springframework.org/schema/context**

**http://www.springframework.org/schema/context/spring-context-2.5.xsd**">

**<context:component-scan base-package="cn.itcast"/>**

</beans>

**2、在配置文件中添加context:component-scan标签**

**<context:component-scan base-package="cn.itcast"/>**

其中base-package为需要扫描的包(含子包)。

**注**:

1、在使用组件扫描元素时，AutowiredAnnotationBeanPostProcessor 和CommonAnnotationBeanPostProcessor会隐式地被包括进来。 也就是说，连个组件都会被自动检测*并*织入 - 所有这一切都不需要在XML中提供任何bean配置元数据。

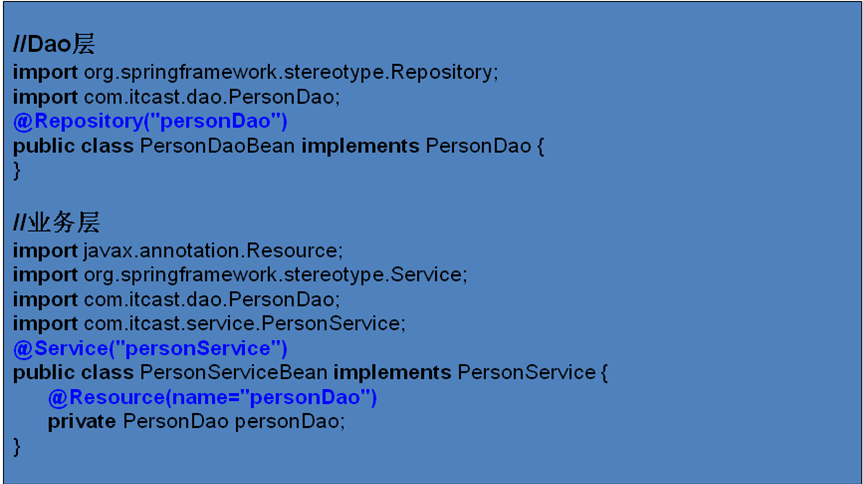
2、功能介绍

@Service用于标注业务层组件、

@Controller用于标注控制层组件（如struts中的action）、

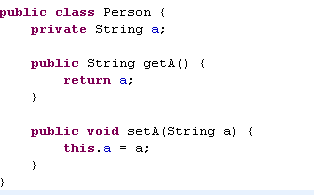
@Repository用于标注数据访问组件，即DAO组件。

而@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。



## spring中的继承

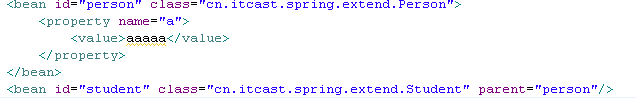
Person类如图所示：



Student类如图所示：



配置文件中：



图中的配置文件中，parent为student在容器中继承person.如果去掉person是不行的。

## IOC和DI的意义

# 4 面向切面编程

## 4.1 代理模式

代理模式的英文叫做Proxy或Surrogate，中文都可译为”代理“，所谓代理，就是一个人或者一个机构代表另一个人或者另一个机构采取行动。在一些情况下，一个客户不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用

1. 抽象主题角色

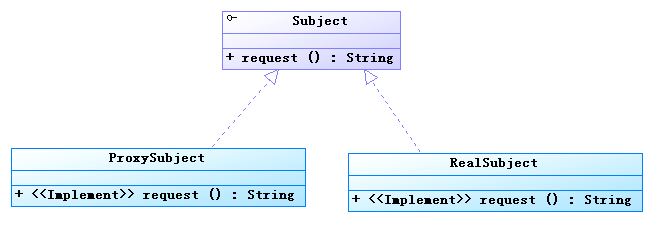
声明了真实主题和代理主题的共同接口，这样一来在任何可以使用真实主题的地方都可以是使用代理主题

1. 代理主题（Proxy）角色：

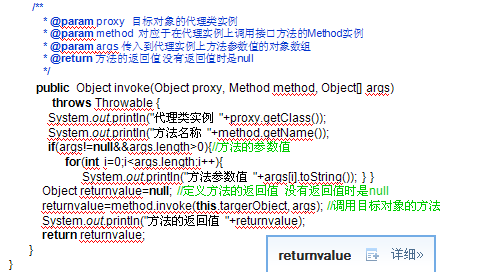
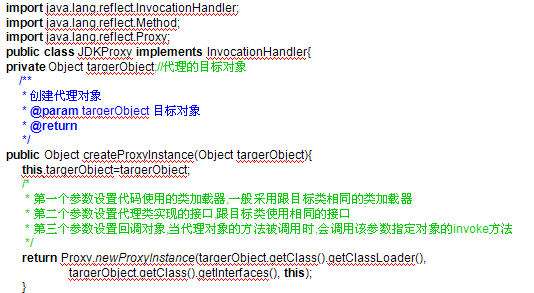
代理主题角色内部含有对真实主题的引用，从而可以在任何时候操作真实主题对象；代理主题角色提供一个与真实主题角色相同的接口，以便可以在任何时候都可以替代真实主题控制对真实主题的引用，负责在需要的时候创建真实主题对象（和删除真实主题对象）；代理角色通常在将客户端调用传递给真实的主题之前或之后，都要执行某个操作，而不是单纯地将调用传递给真实主题对象。

1. 真实主题角色

定义了代理角色所代表地真实对象



### 4.1.1 JDK动态代理



JDK的动态代理必须具备四个条件：

目标接口

目标类

拦截器

代理类

总结：1、因为利用JDKProxy生成的代理类实现了接口，所以目标类中所有的方法在代理类中都有。

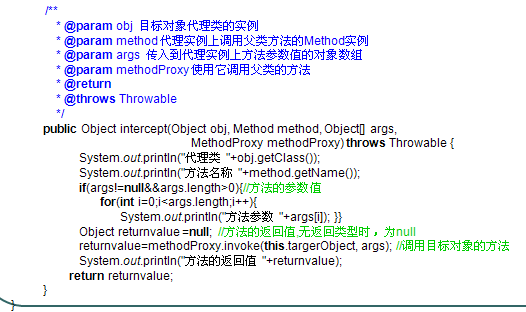
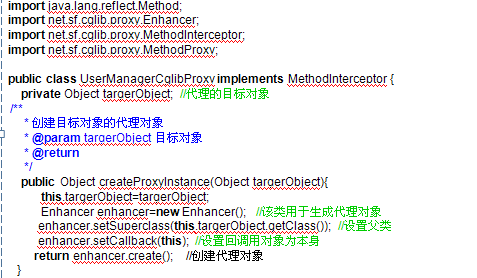
2、生成的代理类的所有的方法都拦截了目标类的所有的方法。而拦截器中invoke方法的内容正好就是代理类的各个方法的组成体。

3、利用JDKProxy方式必须有接口的存在。

4、invoke方法中的三个参数可以访问目标类的被调用方法的API、被调用方法的参数、被调用方法的返回类型。

### 4.1.2 CGLIB做代理

1. CGlib是一个强大的,高性能,高质量的Code生成类库。它可以在运行期扩展Java类与实现Java接口。
2. 用CGlib生成代理类是目标类的子类。
3. 用CGlib生成 代理类不需要接口
4. 用CGLib生成的代理类重写了父类的各个方法。
5. 拦截器中的intercept方法内容正好就是代理类中的方法体



spring有两种代理方式：

1. 若目标对象实现了若干接口，spring使用JDK的java.lang.reflect.Proxy类代理。

优点：因为有接口，所以使系统更加松耦合

缺点：为每一个目标类创建接口

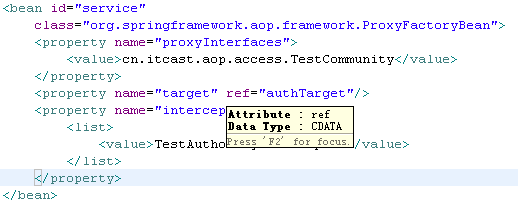
1. 若目标对象没有实现任何接口，spring使用CGLIB库生成目标对象的子类。

优点：因为代理类与目标类是继承关系，所以不需要有接口的存在。

缺点：因为没有使用接口，所以系统的耦合性没有使用JDK的动态代理好。

### 4.1.3 Spring的动态代理

1. 拦截器必须实现MethodInterceptor接口
2. 在spring中的配置



总结：不管采用JDK动态代理生成代理类还是采用CGLIB生成动态代理类。目标类中的所有方法都被拦截下来。而在哪个方法里做比如权限的判断、安全性的检查等一系列工做必须在拦截器中作相应的判断。但是这样的编程形式给程序的编写带来了一定的麻烦。

1. 在拦截器中控制哪些方法将被做权限判断、安全性检查等是一件比较困难的事情。
2. 采取这样的配置目标类只能是一个，所以如果用这种方法做权限控制，得写很多代理，这样给代码的书写造成了困难。
3. 每一个类中的每一个方法如果都有不同的权限（实际的系统往往都是这样的），在拦截器中的判断代码书写会很困难。
4. 这样的代码也会导致硬编码，也就是说我们必须在拦截器中写一些权限判断等事情，会导致拦截器中代码量的增大，造成维护的麻烦。

## 4.2 AOP编程

### 4.2.1概念：

1. Aspect(切面)

比如说事务、权限等，与业务逻辑没有关系的部分

1. joinpoint(连接点)

目标类的目标方法。（由客户端在调用的时候决定）

1. Pointcut(切入点)

所谓切入点是指我们要对那些拦截的方法的定义.

被纳入spring aop中的目标类的方法。

1. Advice(通知)

所谓通知是指拦截到joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(**切面要完成的功能**)

1. Target(目标对象):

代理的目标对象

1. Weaving(织入)

是指把切面应用到目标对象来创建新的代理对象的过程.切面在指定的连接点织入到目标对象

|  |  |
| --- | --- |
| JDKProxy代理 | SpringAop |
| 目标对象 | 目标对象 |
| 拦截器类 | 切面 |
| 拦截器类中的方法 | 通知 |
| 被拦截到的目标类中方法的集合 | 切入点 |
| 在客户端调用的方法（目标类目标方法） | 连接点 |
| 代理类 | AOP代理 |
| 代理类的代理方法生成的过程 | 织入 |

通知根据拦截目标类中的目标方法的位置不一样可以分为：前置通知、后置通知、最终通知、环绕通知、异常通知

### 4.2.2 AOP实现的两种模式

#### xml形式

1. 前置通知

在spring配置文件中声明切面



在spring配置文件中声明目标类



定义切面、切入点、通知



注：见6.2.3.4



说明：

1、在切面类中，没有必要实现接口，但方法名称要与<aop:before method=”checkSecurity” 中的checkSecurity一样。

2、checkSecurity方法中通过JoinPoint参数可以获得目标类的目标方法名称、参数值等信息。

1. 后置通知
2. 没有特殊说明的地方和前置通知是一样的。
3. 在spring配置文件中



1. 在拦截器中的方法要和checkSecurity方法一样，有两个参数

JoinPoint point 可以获得目标方法和参数值

Object val 这里的名字要和returning=”val”中保持一致，指的是方法的返回值。

1. returning=”val”时，通知里可以有返回参数，这个参数只能决定通知里能不能拿到方法的返回值，和客户端没有关系。



1. 在执行目标类的目标方法中遇到异常，则不执行后置通知。
2. 异常通知
3. 没有特殊说明的地方和前置通知一样
4. 在spring配置文件中



其中throwing指定了传递异常的参数名称

1. 在异常通知中（拦截器）中，必须是checkSecurity方法。方法中有两个参数

JoinPoint point 可以获得方法的名称、参数

Throwable ex 利用ex.getMessage()可以获得异常信息

1. 最终通知
2. 没有特殊说明，和前置通知的配置一样。
3. 在spring配置文件里：



说明：在最终通知中不受异常的影响。也就是说不论目标方法执行的过程中是否抛出异常，最终通知都将执行。

1. 环绕通知
2. 没有特殊说明和前置通知的配置保持一致。
3. 在spring文件中

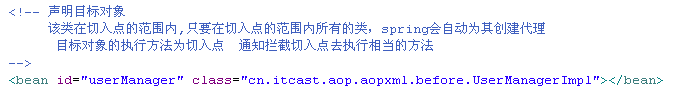


1. 在环绕通知中，方法名称为checkSecurity。参数 类型 为ProceedingJoinPoint。

ProceedingJoinPoint的proceed方法相当于invoke方法，调用目标类的目标方法。ProceedingJoinPoint继承了JoinPoint类

1. 能在方法执行的前后加入额外的代码。

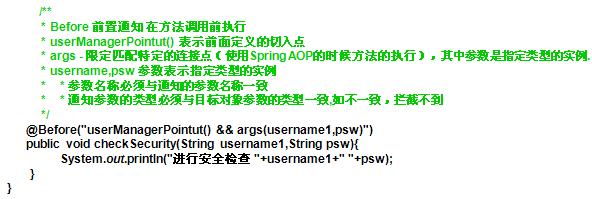
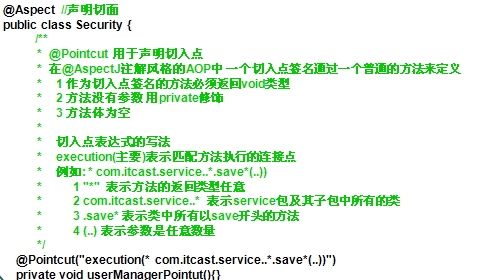
说明：



#### 4.2.2.2Aop注解形式

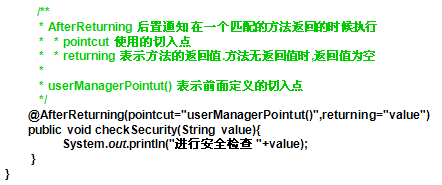
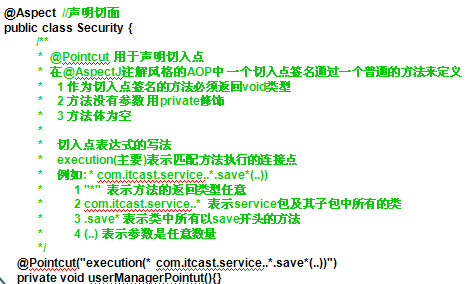


1. 前置通知

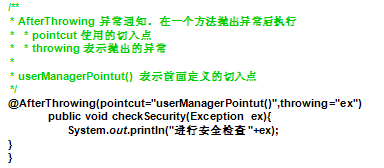
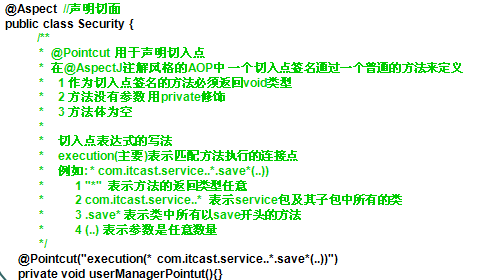


注意：@Aspectj是按照类型匹配的。

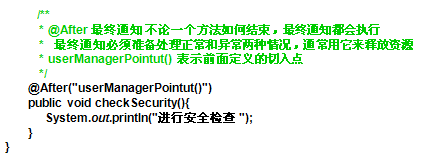
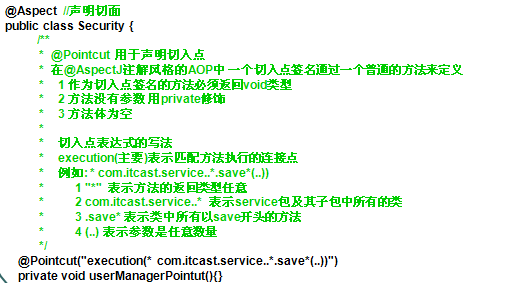
1. 后置通知



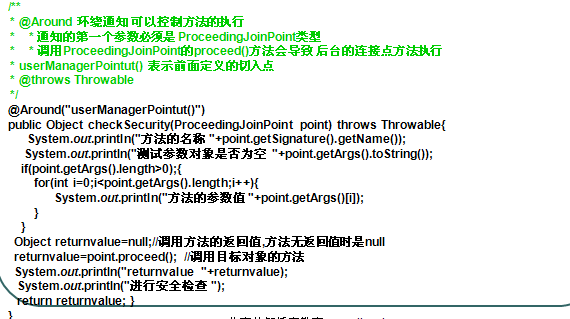
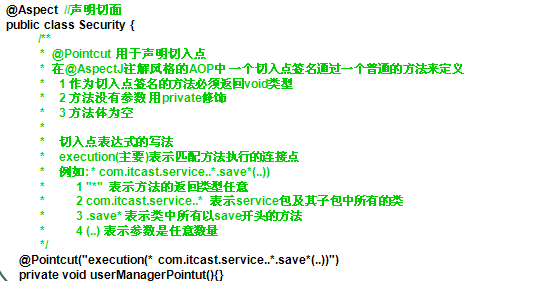
1. 异常通知



1. 最终通知



1. 环绕通知



# 5 Spring数据库

## 5.1 Spring+JDBC

### 5.1.1 Jdbc编程特点

静态代码+动态变量 = jdbc编程。在spring中动态变量可以用注入的形式给予。这样的编程方式适合包装成模板。静态代码构成了模板，而动态变量则是需要传入的参数。

### 5.1.2引入DataSource

在spring中注入DataSource

### 5.1.3 核心类JdbcTemplate

1. 基于模板的设置（为什么可以设置成基于模板的形式）
2. 完成了资源的创建和释放的工作
3. 简化为我们对JDBC的操作
4. 完成了对JDBC的核心流程的工作，包括SQL语句的创建和执行
5. 仅需要传递DataSource就可以把它实例化
6. JdbcTemplate只需要创建一次
7. JdbcTemplate是线程安全类

### 5.1.4 使用JdbcTemplate

在Dao类中，用JdbcTemplate作为属性，用spring对JdbcTemplate进行注入。再对JdbcTemplate进行DataSource注入。

注：为什么只要对JdbcTemplate注入DataSource就可以了？

### 5.1.5 继承JdbcDaoSupport

在Dao类中，继承JdbcDaoSupport。因为JdbcDaoSupport已经有了JdbcTemplate的引用，所以只要继承JdbcDaoSupport就相当于有了JdbcTemplate属性。

### 5.1.6 使用properties文件

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>jdbc.properties</value>

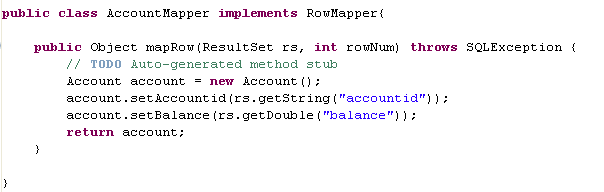
</list>

</property>

</bean>

### 5.1.7 RowMapper的使用

1. 产生原因：在Jdbc的操作中，有很多情况下是要将ResultSet里的数据封装到一个持久化Bean里，再把持久化Bean封装到集合中。这样会造成大量的代码的重复，不利于代码重用。而RowMapper正好解决了这个问题。
2. 使用：
3. 写一个类实现RowMapper接口



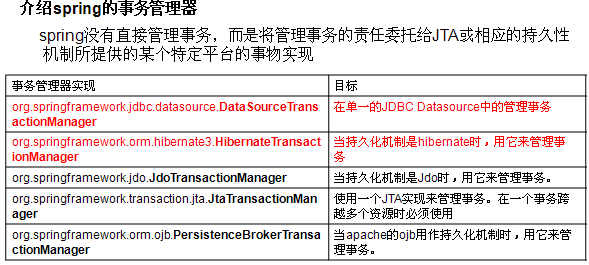
1. 在回调接口中，作为参数进行传入即可。



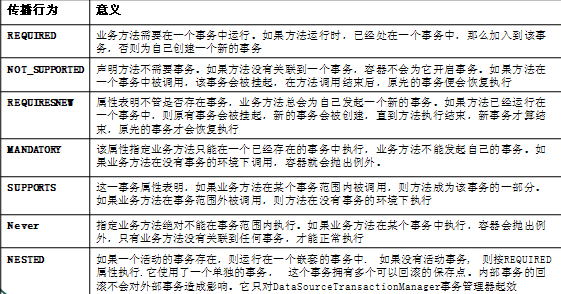
### 5.1.8声明式事务管理



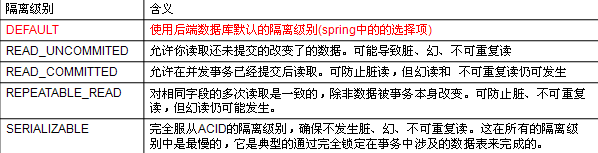
#### 5.1.8.1Spring的事务管理器



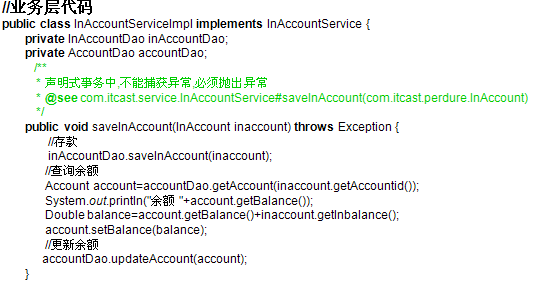
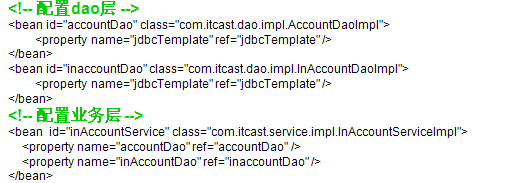
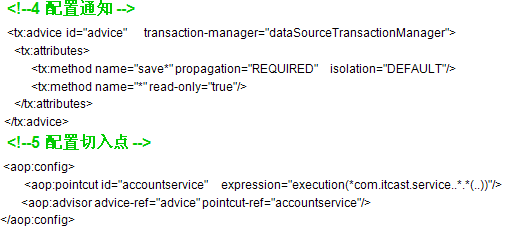
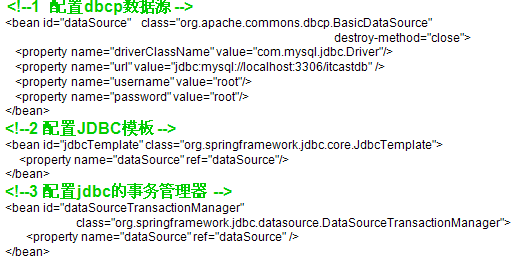
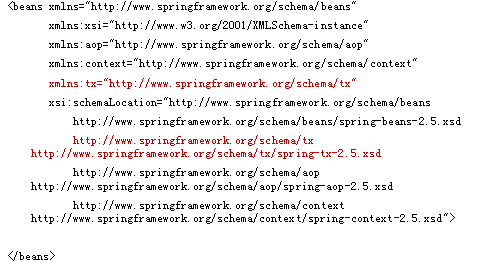
#### 5.1.8.2Spring事务的传播属性



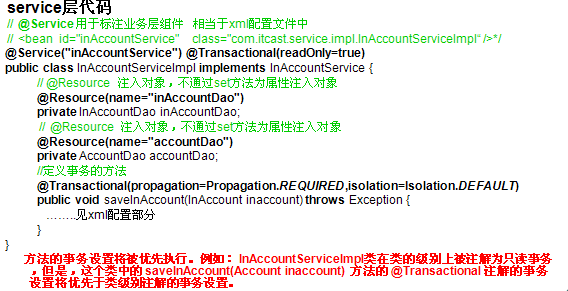
#### 5.1.8.3Spring事务的隔离级别



#### 5.1.8.4以XML配置的 形式



#### 5.1.8.5以注解方式配置



### 5.1.9使用CGLIB以XML形式配置事务

1. 业务逻辑类不能实现接口
2. 客户端调用返回的是目标类（代理类的父类）

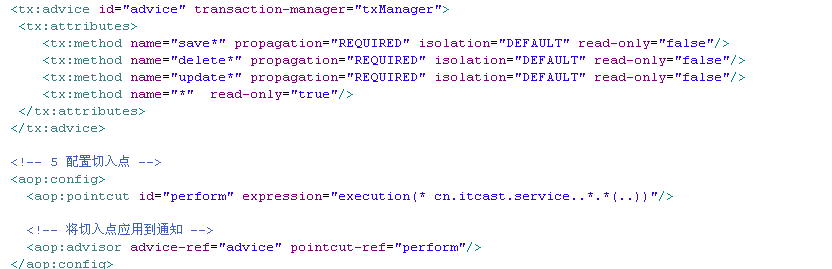
## 5.2 Spring+Hibernate

### 5.2.1 HibernateTemplate模板

1. 如果一个DAO类继承了HibernateDaoSupport，只需要在spring配置文件中注入SessionFactory就可以了。
2. 如果一个DAO类没有继承HibernateDaoSupport,需要有一个SessionFactory的属性，并且在配置文件中进行注入。

### 5.2.2 声明式事务

#### 配置XML文件



#### 注解形式：

1. 在配置文件中引用spring的自动扫描机制。



1. 在配置文件中引入注解解析器



1. 在service层通过@Transaction进行注解

# 6 Struts2+spring+hibernate

## 6.1 需要添加的jar包

## 6.2 Spring融合web服务器

1、 在web.xml文件中，添加如下配置：



说明：

在web.xml中加载applicationContext.xml文件有几种方式：

如果spring配置文件被命名为applicationContext.xml，并且放在WEB-INF目录下，则不需要配置<context-param>，因为ContextLoaderListener默认在WEB-INF目录下寻找名为applicationContext.xml的文件。若存在多个Spring配置文件，则在<param-value>中依次列出，之间以逗号隔开。

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/classes/applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

1. 也可以利用上图中的方式来配置：

利用classpath来配置

1. 我们一般采用classpath的方式来配置

上面的配置说明在web容器启动的时候，spring容器也启动了。

## 6.3 struts.xml文件

在struts.xml文件中，添加struts2的常量：

struts.objectFactory = spring



在spring中的文件配置：



## 6.4 OpenInSessionView

由于使用的是spring的声明式事务处理方式，所以在调用this.getHibernateTemplate().load方法时，使用了hibernate的懒加载技术。当把一个实体Bean从数据库中加载完以后，只能加载其ID值。这个时候spring的声明式事务处理方式已经把session给关闭掉了。所以当值在页面输出时会产生异常。

处理方式为：OpenSessionInview模式。

