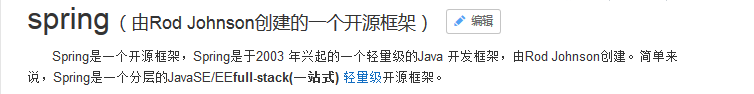


# Spring的概述

## 什么是Spring :



Spring是一个开源框架，Spring是于2003 年兴起的一个轻量级的Java 开发框架，由[Rod Johnson](http://baike.baidu.com/view/2192255.htm)在其著作《Expert one-on-one J2EE Development without EJB》，中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为 [J2EE](http://baike.baidu.com/view/1507.htm) 应用程序开发提供集成的框架。Spring使用基本的[JavaBean](http://baike.baidu.com/view/28155.htm)来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于[服务器](http://baike.baidu.com/view/899.htm)端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。Spring的核心是[控制反转](http://baike.baidu.com/view/1486379.htm)（[IoC](http://baike.baidu.com/view/146665.htm)）和面向切面（[AOP](http://baike.baidu.com/view/73626.htm)）。简单来说，**Spring是一个分层的JavaSE/EEfull-stack(一站式)** [轻量级](http://baike.baidu.com/subview/1318763/16205192.htm)**开源框架。**Rod Johnson成为一个改变Java世界的大师级人物。

**EE开发分成三层结构:**

WEB层:Spring MVC.

业务层:Bean管理:(IOC)

持久层:Spring的JDBC模板.ORM模板用于整合其他的持久层框架.

Expert One-to-One J2EE Design and Development :J2EE的设计和开发:(2002.EJB)

对Java EE 系统框架臃肿、低效、脱离现实的种种现状提出了质疑，并积极寻求探索革新

Expert One-to-One J2EE Development without EJB :J2EE不使用EJB的开发.

## Spring的版本

**Spring 3.X 和 Spring4.X**

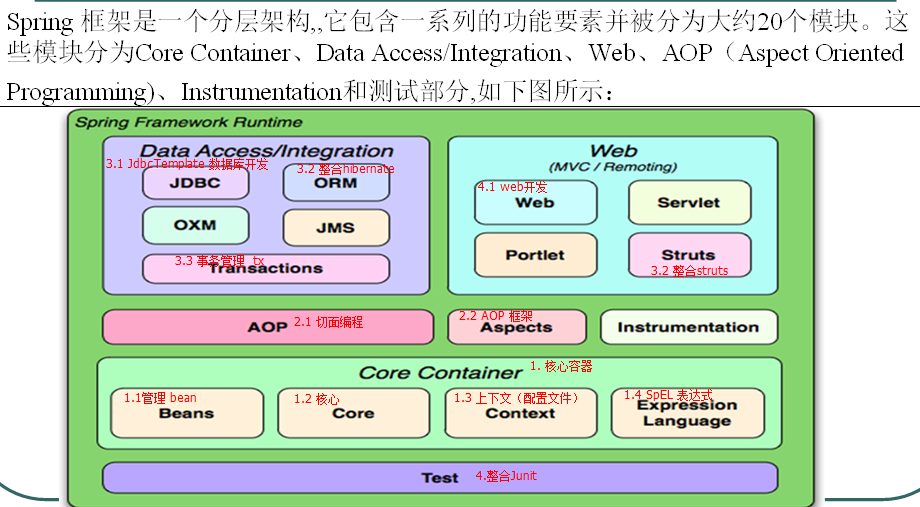
## spring核心

* Spring的核心是**控制反转（IoC）**和**面向切面（AOP）**

## spring优点

|  |
| --- |
| * 方便解耦，简化开发 （高内聚低耦合）   + Spring就是一个大工厂（容器），可以将所有对象创建和依赖关系维护，交给Spring管理   + spring工厂是用于生成bean * AOP编程的支持   + Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能 * 声明式事务的支持   + 只需要通过配置就可以完成对事务的管理，而无需手动编程 * 方便程序的测试   + Spring对Junit4支持，可以通过注解方便的测试Spring程序 * 方便集成各种优秀框架   + Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz等）的直接支持 * 降低JavaEE API的使用难度   + Spring 对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低 |

## spring体系结构



**核心容器：beans、core、context、expression**

# 入门案例：IoC（）【掌握】

IOC:Inversion of Control的缩写

#### 步骤一:下载Spring的开发包:

官网：http://spring.io/

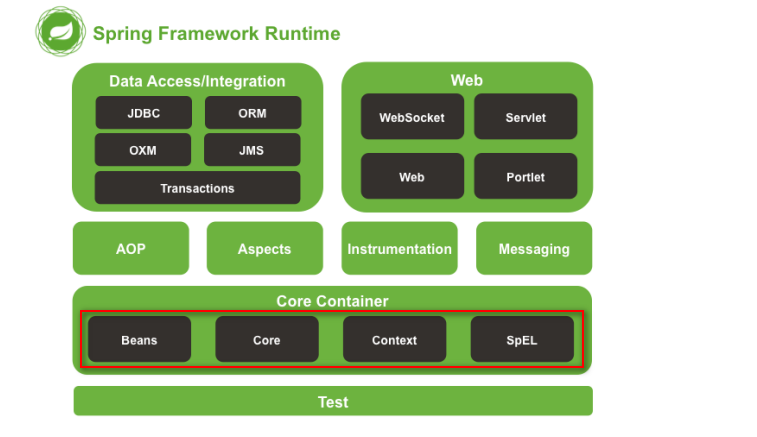
下载地址：http://repo.springsource.org/libs-release-local/org/springframework/spring解压:(Spring目录结构:)

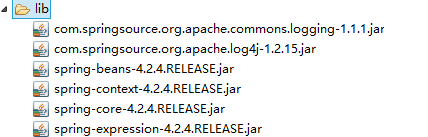
\* docs :API和开发规范.

\* libs :jar包和源码.

\* schema :约束.

#### 步骤二:创建web项目,引入Spring的开发包:





4个核心包2个日志包

#### 步骤三创建目标类

* 提供UserService接口和实现类
* 获得UserService实现类的实例

之前开发中，直接new一个对象即可。

学习spring之后，将由Spring创建对象实例--> IoC 控制反转（Inverse of Control）

之后需要实例对象时，从spring工厂（容器）中获得，需要将实现类的全限定名称配置到xml文件中

|  |
| --- |
| **public** **interface** UserService {    **public** **void** addUser();  } |
| **public** **class** UserServiceImpl **implements** UserService {  @Override  **public** **void** addUser() {  System.*out*.println("add user");  }  } |

#### 步骤四编写配置文件

**引入相关配置文件: applicationContext.xml和log4j.properties日志配置文件**

**Spring的配置文件：applicationContext.xml：**

* 位置：任意，开发中一般在classpath下（src）
* 名称：任意，开发中常用applicationContext.xml
* 内容：添加schema约束

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean id=*"userServiceId"* class=*"com.qf.ioc.userServiceImpl"></bean>*  </beans> |

#### 步骤五编写测试类

**ApplicationContext接口（了解）**

\* 使用ApplicationContext工厂的接口，使用该接口可以获取到具体的Bean对象

\* 该接口下有两个具体的实现类

\* ClassPathXmlApplicationContext -- 加载类路径下的Spring配置文件

\* FileSystemXmlApplicationContext

@Test

// Spring的方式：

**public** **void** demo2(){

// 创建Spring的工厂类：

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//获得内容 --不需要自己new，都是从spring容器获得

UserService userService = (UserService) applicationContext.getBean("userServiceId");

userService.addUser();

}

注意：Spring工厂默认调用无参数的构造函数来实例化对象。

# 入门案例：DI【掌握】

* DI Dependency Injection ,依赖注入

is a ：是一个，继承。

has a：有一个，成员变量，依赖。

class B {

private A a; //B类依赖A类

}

依赖：一个对象需要使用另一个对象

注入：通过setter方法进行另一个对象实例设置。

* 例如：

class BookServiceImpl{

//之前开发：接口 = 实现类 （service和dao耦合）

//private BookDao bookDao = new BookDaoImpl();

//spring之后 （解耦：service实现类使用dao接口，不知道具体的实现类）

private BookDao bookDao;

setter方法

}

模拟spring执行过程

创建service实例：BookService bookService = new BookServiceImpl() -->IoC

创建dao实例：BookDao bookDao = new BookDaoImple() -->IoC

将dao设置给service：bookService.setBookDao(bookDao); -->DI

### dao

|  |
| --- |
| **public** **interface** BookDao {    **public** **void** addBook();  } |
| **public** **class** BookDaoImpl **implements** BookDao {  @Override  **public** **void** addBook() {  System.*out*.println(" add book");  }  } |

### Service

|  |
| --- |
| **public** **interface** BookService {  **public** **abstract** **void** addBook();  } |
| **public** **class** BookServiceImpl **implements** BookService {    // 方式1：之前，接口=实现类  // private BookDao bookDao = new BookDaoImpl();  // 方式2：接口 + setter  **private** BookDao bookDao;  **public** **void** setBookDao(BookDao bookDao) {  **this**.bookDao = bookDao;  }    @Override  **public** **void** addBook(){  **this**.bookDao.addBook();  }  } |

### 配置文件

|  |
| --- |
| <!-- 创建service -->  <bean id=*"bookServiceId"* class=*"com.qf.BookServiceImpl"*>  <property name=*"bookDao"* ref=*"bookDaoId"*></property>  </bean>    <!-- 创建dao实例 -->  <bean id=*"bookDaoId"* class=*"com.qf. di.BookDaoImpl"></bean>* |

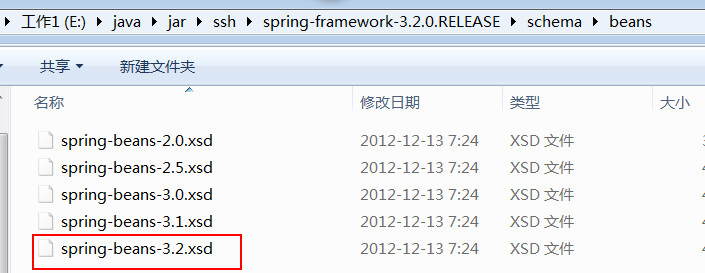
### 测试

|  |
| --- |
| @Test  **public** **void** demo01(){  //从spring容器获得  String xmlPath = "com/qf/di/beans.xml";  ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext(xmlPath);  BookService bookService = (BookService) applicationContext.getBean("bookServiceId");    bookService.addBook();    } |

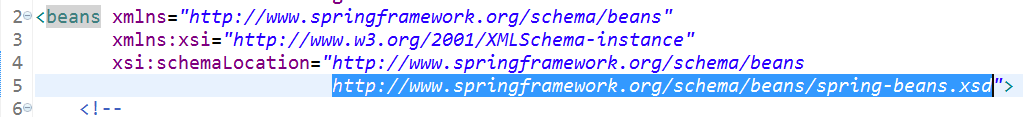
# schema xml提示

## 步骤一：确定xsd文件位置

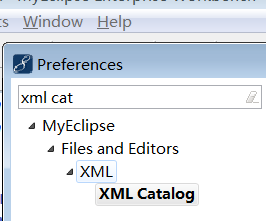
spring-framework-3.2.0.RELEASE\schema\beans



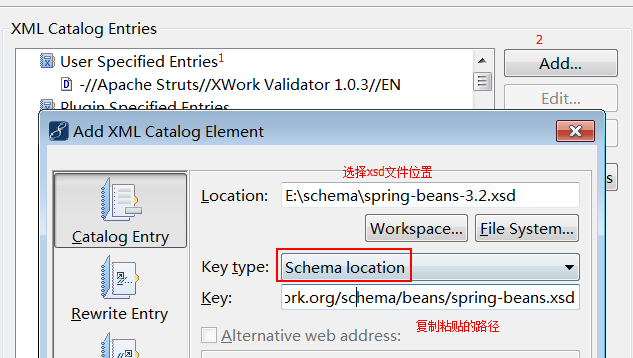
## 步骤二：复制路径



## 步骤三：搜索“xml catalog”

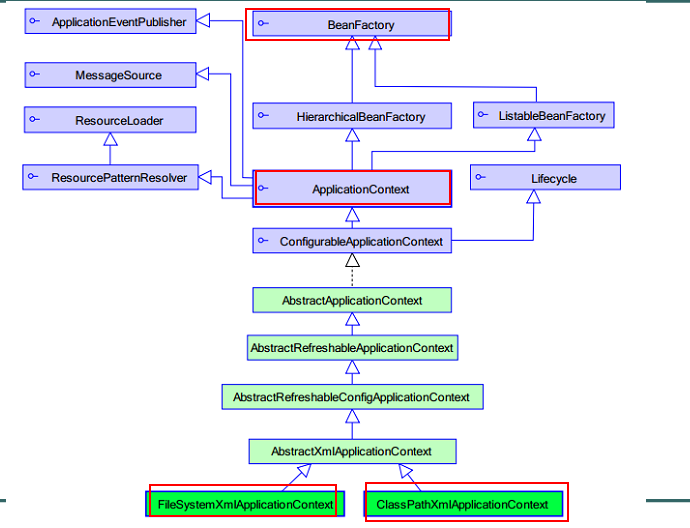


## 步骤四：添加约束提示



# 核心API

* api整体了解，之后不使用，在学习过程需要。



* BeanFactory ：这是一个工厂，用于生成任意bean。

采取延迟加载，第一次getBean时才会初始化Bean

* ApplicationContext：是BeanFactory的子接口，功能更强大。（国际化处理、事件传递、Bean自动装配、各种不同应用层的Context实现）。当配置文件被加载，就进行对象实例化。

ClassPathXmlApplicationContext 用于加载classpath（类路径、src）下的xml

加载xml运行时位置 --> /WEB-INF/classes/...xml

FileSystemXmlApplicationContext 用于加载指定盘符下的xml

加载xml运行时位置 --> /WEB-INF/...xml

通过java web ServletContext.getRealPath() 获得具体盘符

# Spring的相关配置:

id :Bean起个名字. 在约束中采用ID的约束:唯一. 必须以字母开始，可以使用字母、数字、连字符、下划线、句话、冒号 id:不能出现特殊字符.

<bean id=”bookAction”>

# 装配Bean 基于XML

## 实例化方式

* 3种bean实例化方式：默认构造、静态工厂、实例工厂

### 默认构造

|  |
| --- |
| <bean id="" class=""> 必须提供默认构造 |

### 静态工厂

* 常用与spring整合其他框架（工具）
* 静态工厂：用于生成实例对象，所有的方法必须是static

|  |
| --- |
| <bean id="" class="工厂全限定类名" factory-method="静态方法"> |

工厂

|  |
| --- |
| **public** **class** MyBeanFactory {    /\*\*  \* 创建实例  \* **@return**  \*/  **public** **static** UserService createService(){  **return** **new** UserServiceImpl();  }  } |

### 实例工厂

|  |
| --- |
| <!-- 用spring创建工厂实例 单例的 -->  <bean id="facId" class="com.qf.c\_xml.factory.MyFactory" scope="singleton"></bean>      <!-- 配置bean来获得service实例 -->  <bean id="userServieId" factory-bean="facId" factory-method="getInstance"></bean> |

## Spring框架中<bean>标签的配置

* 作用域：用于确定spring创建bean实例个数



* 取值：

singleton 单例，默认值。

prototype 多例，每执行一次getBean将获得一个实例。例如：struts整合spring，配置action多例。

* 配置信息

|  |
| --- |
| <bean id="" class="" scope=""> |

|  |
| --- |
| 1. id属性的区别  \* id -- Bean起个名字，在约束中采用ID的约束，唯一  \* 取值要求：必须以字母开始，可以使用字母、数字、连字符、下划线、句话、冒号 id:不能出现特殊字符    2. class属性 -- Bean对象的全路径  3. scope属性 -- scope属性代表Bean的作用范围  \* singleton -- 单例（默认值）  \* prototype -- 多例，在Spring框架整合Struts2框架的时候，Action类也需要交给Spring做管理，配置把Action类配置成多例！！  \* request -- 应用在Web项目中,每次HTTP请求都会创建一个新的Bean  \* session -- 应用在Web项目中,同一个HTTP Session 共享一个Bean  \* globalsession -- 应用在Web项目中,多服务器间的session    4. Bean对象的创建和销毁的两个属性配置（了解）  \* 说明：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  \* init-method -- 当bean被载入到容器的时候调用init-method属性指定的方法  \* destroy-method -- 当bean从容器中删除的时候调用destroy-method属性指定的方法  \* 想查看destroy-method的效果，有如下条件  \* scope= singleton有效  \* web容器中会自动调用，但是main函数或测试用例需要手动调用（需要使用ClassPathXmlApplicationContext的close()方法） |

## 生命周期

### 初始化和销毁

* 目标方法执行前后执行后，将进行初始化或销毁。

|  |
| --- |
| <bean id="" class="" init-method="初始化方法名称" destroy-method="销毁的方法名称"> |

#### 目标类

|  |
| --- |
| **public** **class** UserServiceImpl **implements** UserService {  @Override  **public** **void** addUser() {  System.*out*.println("add user");  }    **public** **void** myInit(){  System.*out*.println("初始化");  }  **public** **void** myDestroy(){  System.*out*.println("销毁");  }  } |

#### spring配置

|  |
| --- |
| <!--  init-method 用于配置初始化方法,准备数据等  destroy-method 用于配置销毁方法,清理资源等  -->  <bean id=*"userServiceId"* class=*"com.qf.UserServiceImpl"*  init-method=*"myInit"* destroy-method=*"myDestroy"* ></bean> |

#### 测试

|  |
| --- |
| @Test  **public** **void** demo02() **throws** Exception{  //spring 工厂  String xmlPath = "com/qf/beans.xml";  ClassPathXmlApplicationContext applicationContext = **new** ClassPathXmlApplicationContext(xmlPath);  UserService userService = (UserService) applicationContext.getBean("userServiceId");  userService.addUser();      applicationContext.close();    } |

# 属性依赖注入

### 构造方法注入

#### 目标类

|  |
| --- |
| **public** **class** User {    **private** Integer uid;  **private** String username;  **private** Integer age;    **public** User(Integer uid, String username) {  **super**();  **this**.uid = uid;  **this**.username = username;  }    **public** User(String username, Integer age) {  **super**();  **this**.username = username;  **this**.age = age;  } |

#### spring配置

|  |
| --- |
| <!-- 构造方法注入  \* <constructor-arg> 用于配置构造方法一个参数argument  name ：参数的名称  value：设置普通数据  ref：引用数据，一般是另一个bean id值    index ：参数的索引号，从0开始 。如果只有索引，匹配到了多个构造方法时，默认使用第一个。  type ：确定参数类型  例如：使用名称name  <constructor-arg name="username" value="jack"></constructor-arg>  <constructor-arg name="age" value="18"></constructor-arg>  例如2：【类型type 和 索引 index】  <constructor-arg index="0" type="java.lang.String" value="1"></constructor-arg>  <constructor-arg index="1" type="java.lang.Integer" value="2"></constructor-arg>  -->  <bean id=*"userId"* class=*"com.qf.constructor.User"* >  <constructor-arg index=*"0"* type=*"java.lang.String"* value=*"1"*></constructor-arg>  <constructor-arg index=*"1"* type=*"java.lang.Integer"* value=*"2"*></constructor-arg>  </bean> |

### setter方法注入

|  |
| --- |
| <!-- setter方法注入  \* 普通数据  <property name="" value="值">  等效  <property name="">  <value>值  \* 引用数据  <property name="" ref="另一个bean">  等效  <property name="">  <ref bean="另一个bean"/>    -->  <bean id=*"personId"* class=*"com.qf.setter.Person"*>  <property name=*"pname"* value=*"小苍"*></property>  <property name=*"age"*>  <value>1234</value>  </property>    <property name=*"homeAddr"* ref=*"homeAddrId"*></property>  <property name=*"companyAddr"*>  <ref bean=*"companyAddrId"*/>  </property>  </bean>    <bean id=*"homeAddrId"* class=*"com.qf.setter.Address"*>  <property name=*"addr"* value=*"北京"*></property>  <property name=*"tel"* value=*"****110*** *"*></property>  </bean>  <bean id=*"companyAddrId"* class=*"com.qf.setter.Address"*>  <property name=*"addr"* value=*"北京海淀"*></property>  <property name=*"tel"* value=*"120"*></property>  </bean> |

### P命令空间[了解]

**名称空间p的属性注入的方式:Spring2.x版本后提供的方式.**

* 对“setter方法注入”进行简化，替换<property name="属性名">，而是在

<bean p:属性名="普通值" p:属性名-ref="引用值">

* p命名空间使用前提，必须添加命名空间

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

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

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

xsi:schemaLocation="

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

|  |
| --- |
| <bean id=*"personId"* class=*"com.qf.p.Person"*  p:pname=*"小泽"* p:age=*"22"*  p:homeAddr-ref=*"homeAddrId"* p:companyAddr-ref=*"companyAddrId"*>  </bean>    <bean id=*"homeAddrId"* class=*"com.qf.p.Address"*  p:addr=*"东莞"* p:tel=*"666666"*>  </bean>  <bean id=*"companyAddrId"* class=*"com.qf.p.Address"*  p:addr=*"日本"* p:tel=*"333333"*>  </bean> |

### SpEL的方式的属性注入[了解]

**Spring3.x版本后提供的方式.**

* 对<property>进行统一编程，所有的内容都使用value

<property name="" value="#{表达式}">

#{123}、#{'jack'} ： 数字、字符串

#{beanId} ：另一个bean引用

#{beanId.propName} ：操作数据

#{beanId.toString()} ：执行方法

#{T(类).字段|方法} ：静态方法或字段

|  |
| --- |
| <!--  <property name="cname" value="#{'jack'}"></property>  <property name="cname" value="#{customerId.cname.toUpperCase()}"></property>  通过另一个bean，获得属性，调用的方法  <property name="cname" value="#{customerId.cname?.toUpperCase()}"></property>  ?. 如果对象不为null，将调用方法  -->  <bean id=*"customerId"* class=*"com.qf.spel.Customer"* >  <property name=*"cname"* value=*"#{customerId.cname?.toUpperCase()}"*></property>  <property name=*"pi"* value=*"#{T(java.lang.Math).PI}"*></property>  </bean> |

### 集合注入

|  |
| --- |
| <!--  集合的注入都是给<property>添加子标签  数组：<array>  List：<list>  Set：<set>  Map：<map> ，map存放k/v 键值对，使用<entry>描述  Properties：<props> <prop key=""></prop> 【】    普通数据：<value>  引用数据：<ref>  -->  <bean id=*"collDataId"* class=*"com.qf.coll.CollData"* >  <property name=*"arrayData"*>  <array>  <value>hello wolrd</value>  <value>jack</value>  <value>rose</value>  </array>  </property>    <property name=*"listData"*>  <list>  <value>芙蓉</value>  <value>菊花</value>  <value>如花</value>  </list>  </property>    <property name=*"setData"*>  <set>  <value>芙蓉</value>  <value>菊花</value>  <value>如花</value>  </set>  </property>    <property name=*"mapData"*>  <map>  <entry key=*"jack"* value=*"杰克"*></entry>  <entry key=*"rose"* value=*"肉丝"*></entry>  </map>  </property>    <property name=*"propsData"*>  <props>  <prop key=*"高富帅"*>冠希哥</prop>  <prop key=*"白富美"*>阿凤姐</prop>  <prop key=*"男屌丝"*>菊花信</prop>  </props>  </property>  </bean> |

# Spring的分配置文件的开发

一种:创建工厂的时候加载多个配置文件:

ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml","applicationContext2.xml");

二种:在一个配置文件中包含另一个配置文件：

<import resource="applicationContext2.xml"></import>

# 技术分析之Spring框架的IOC功能之注解的方式

## Spring框架的IOC之注解方式的快速入门

### 1. 步骤一：导入注解开发所有需要的jar包

引入IOC容器必须的6个jar包

多引入一个：Spring框架的AOP的jar包，spring-aop的jar包

### 2. 步骤二：创建对应的包结构，编写Java的类

UserService -- 接口

UserServiceImpl -- 具体的实现类

### 3. 步骤三：引入约束

在src的目录下，创建applicationContext.xml的配置文件，然后引入约束。注意：因为现在想使用注解的方式，那么引入的约束发生了变化

\* 需要引入context的约束，具体的约束如下

<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.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"> <!-- bean definitions here -->

</beans>

### 4. 步骤四：在applicationContext.xml配置文件中开启组件扫描

Spring的注解开发:组件扫描

<context:component-scan base-package="com.qf.demo1"/>

注意：可以采用如下配置

<context:component-scan base-package="com.qf"/> 这样是扫描com.qf包下所有的内容

5. 步骤五：在UserServiceImpl的实现类上添加注解

@Component(value="userService") -- 相当于在XML的配置方式中 <bean id="userService" class="...">

6. 步骤六：编写测试代码

public class SpringDemo1 {

@Test

public void run1(){

ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");

UserService us = (UserService) ac.getBean("userService");

us.save();

}

}

## Spring框架中Bean管理的常用注解

**1. @Component:组件.(作用在类上)**

**2. Spring中提供@Component的三个衍生注解:(功能目前来讲是一致的)**

@Controller -- 作用在WEB层

@Service -- 作用在业务层

@Repository -- 作用在持久层

说明：这三个注解是为了让标注类本身的用途清晰，Spring在后续版本会对其增强

3. 属性注入的注解(说明：使用注解注入的方式,可以不用提供set方法)

如果是注入的普通类型，可以使用value注解

@Value -- 用于注入普通类型

如果注入的是对象类型，使用如下注解

@Autowired -- 默认按类型进行自动装配

如果想按名称注入

@Qualifier -- 强制使用名称注入

@Resource -- 相当于@Autowired和@Qualifier一起使用

强调：Java提供的注解

属性使用name属性

**Bean的作用范围和生命周期的注解**

1. Bean的作用范围注解

注解为@Scope(value="prototype")，作用在类上。值如下：

singleton -- 单例，默认值

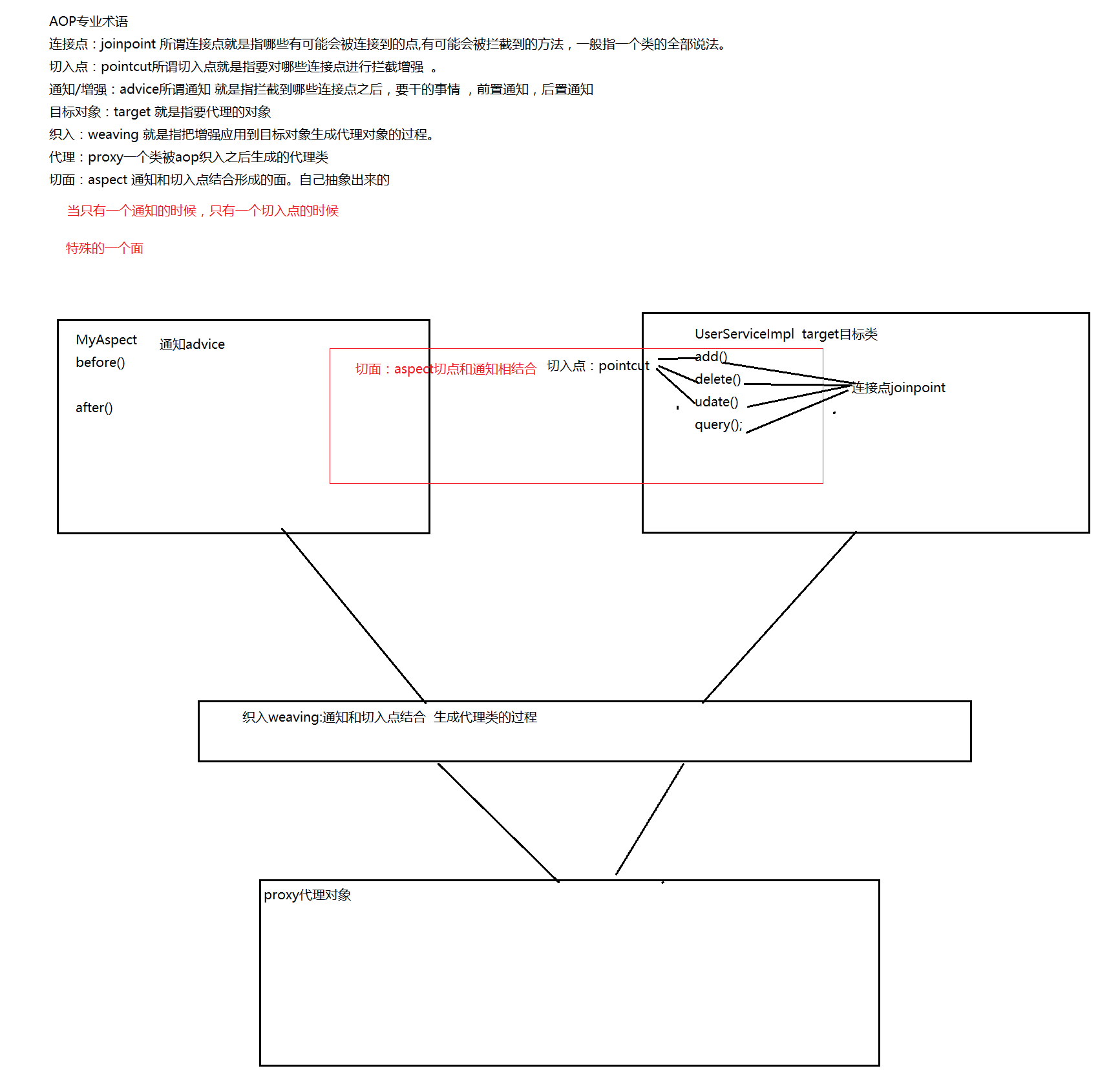
prototype -- 多例

2. Bean的生命周期的配置（了解）

注解如下：

@PostConstruct -- 相当于init-method

@PreDestroy -- 相当于destroy-method



# Spring框架整合JUnit单元测试

**1. 为了简化了JUnit的测试，使用Spring框架也可以整合测试**

**2. 具体步骤**

要求：必须先有JUnit的环境（即已经导入了JUnit4的开发环境）！！

步骤一：在程序中引入:spring-test.jar

步骤二：在具体的测试类上添加注解

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:configs/applicationContext.xml")

public class SpringDemo1 {

@Resource(name="userService")

private UserService userService;

@Test

public void demo2(){

userService.save();

}

}

# AOP

## 什么是AOP

1、在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP（面向对象编程）的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

1. AOP采取**横向抽取**机制，取代了传统**纵向继承**体系重复性代码
2. 经典应用：事务管理、性能监视、安全检查、缓存 、日志等
3. Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类织入增强代码对程序进行增强（不修改源码的情况下）
4. AspectJ是一个基于Java语言的**AOP框架**，Spring2.0开始，Spring AOP引入对Aspect的支持，AspectJ扩展了Java语言，提供了一个专门的编译器，在编译时提供横向代码的织入

## AOP的底层实现:

1. Srping框架的AOP技术底层也是采用的代理技术，代理的方式提供了两种

1. 基于JDK的动态代理

必须是面向接口的，只有实现了具体接口的类才能生成代理对象

2. 基于CGLIB动态代理

对于没有实现了接口的类，也可以产生代理，产生这个类的子类的方式

2. Spring的传统AOP中根据类是否实现接口，来采用不同的代理方式

1. 如果实现类接口，使用JDK动态代理完成AOP

2. 如果没有实现接口，采用CGLIB动态代理完成AOP

# 动态代理

## 代理模式

什么是代理模式及其作用

Proxy Pattern（即：代理模式），23种常用的面向对象软件的设计模式之一

代理模式的定义：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，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。

**优点:**

(1).职责清晰

真实的角色就是实现实际的业务逻辑，不用关心其他非本职责的事务，通过后期的代理完成一件完成事务，附带的结果就是编程简洁清晰。

(2).代理对象可以在客户端和目标对象之间起到中介的作用，这样起到了的作用和保护了目标对象的作用。

(3).高扩展性

**结构**

一个是真正的你要访问的对象(目标类)，另一个是代理对象,真正对象与代理

对象实现同一个接口,先访问代理类再访问真正要访问的对象。

代理模式简单案例

## JDK动态代理

Java.lang.reflect.Proxy类可以直接生成一个代理对象

**Proxy.newProxyInstance():**产生代理类的实例。仅能代理实现至少一个接口的类

ClassLoader：类加载器。固定写法，和被代理类使用相同的类加载器即可。

Class[] interface：代理类要实现的接口。固定写法，和被代理类使用相同的接口即可。

InvocationHandler：策略（方案）设计模式的应用。如何代理？

**InvocationHandler中的invoke方法：**调用代理类的任何方法，此方法都会执行

Object proxy:代理对象本身的引用。一般用不着。

Method method:当前调用的方法。

Object[] args:当前方法用到的参数

1.目标类：接口 + 实现类

2.工厂类：编写工厂生成代理

4.测试

**1目标类**

|  |
| --- |
| **public** **interface** UserService {    **public** **void** addUser();  **public** **void** updateUser();  **public** **void** deleteUser();  } |

**2、切面类**

|  |
| --- |
| **public** **class** MyAspect {    **public** **void** before(){  System.*out*.println(" 前");  }    **public** **void** after(){  System.*out*.println("后");  }  } |

**3、工厂**

|  |
| --- |
| **public** **class** MyBeanFactory {    **public** **static** UserService createService(){  //1 目标类  **final** UserService userService = **new** UserServiceImpl();  //2切面类  **final** MyAspect myAspect = **new** MyAspect();  /\* 3 代理类：将目标类（切入点）和 切面类（通知） 结合 --> 切面  \* Proxy.newProxyInstance  \* 参数1：loader ，类加载器，动态代理类 运行时创建，任何类都需要类加载器将其加载到内存。  \* 一般情况：当前类.class.getClassLoader();  \* 目标类实例.getClass().get...  \* 参数2：Class[] interfaces 代理类需要实现的所有接口  \* 目标类实例.getClass().getInterfaces() ;注意：只能获得自己接口，不能获得父元素接口  \*  \* 参数3：InvocationHandler 处理类，接口，必须进行实现类，一般采用匿名内部  \* 提供 invoke 方法，代理类的每一个方法执行时，都将调用一次invoke  \* 参数：Object proxy ：代理对象  \* 参数：Method method : 代理对象当前执行的方法的描述对象（反射）  \* 执行方法名：method.getName()  \* 执行方法：method.invoke(对象，实际参数)  \* 参数：Object[] args :方法实际参数  \*  \*/  UserService proxService = (UserService)Proxy.*newProxyInstance*(  MyBeanFactory.**class**.getClassLoader(),  userService.getClass().getInterfaces(),  **new** InvocationHandler() {    @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {    //前执行  myAspect.before();    //执行目标类的方法  Object obj = method.invoke(userService, args);    //后执行  myAspect.after();    **return** obj;  }  });    **return** proxService;  }  } |

**4、测试**

|  |
| --- |
| @Test  **public** **void** demo01(){  UserService userService = MyBeanFactory.*createService*();  userService.addUser();  userService.updateUser();  userService.deleteUser();  } |

## CGLIB字节码增强

有spring的核心包就行啦

核心类：Enhancer

* 没有接口，只有实现类。
* 采用字节码增强框架 cglib，在运行时 创建目标类的子类，从而对目标类进行增强。

1. 获得代理类的核心类Enhancer对象
2. 设置父类（目标类），setSuperclass（）方法，底层是创建目标类的子类
3. 设置回调函数enhancer.setCallbac(new MethodInterceptor())
4. 创建代理对象 create()方法

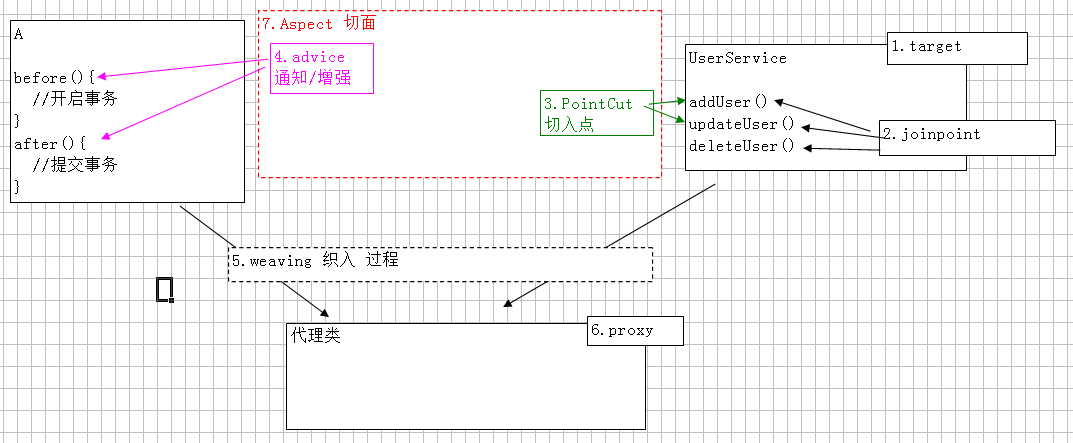
**工厂类**

|  |
| --- |
| **public** **class** MyBeanFactory {    **public** **static** UserServiceImpl createService(){  //1 目标类  **final** UserServiceImpl userService = **new** UserServiceImpl();  //2切面类  **final** MyAspect myAspect = **new** MyAspect();  // 3.代理类 ，采用cglib，底层创建目标类的子类  //3.1 核心类  Enhancer enhancer = **new** Enhancer();  //3.2 确定父类  enhancer.setSuperclass(userService.getClass());  /\* 3.3 设置回调函数 , MethodInterceptor接口 等效 jdk InvocationHandler接口  \* intercept() 等效 jdk invoke()  \* 参数1、参数2、参数3：以invoke一样  \* 参数4：methodProxy 方法的代理  \*  \*  \*/  enhancer.setCallback(**new** MethodInterceptor(){  @Override  **public** Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) **throws** Throwable {    //前  myAspect.before();    //执行目标类的方法  Object obj = method.invoke(userService, args);  // \* 执行代理类的父类 ，执行目标类 （目标类和代理类 父子关系）  methodProxy.invokeSuper(proxy, args);    //后  myAspect.after();    **return** obj;  }  });  //3.4 创建代理  UserServiceImpl proxService = (UserServiceImpl) enhancer.create();    **return** proxService;  }  } |

# AOP的开发中的相关术语

|  |
| --- |
| 1. Joinpoint(连接点) -- 所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点  2. Pointcut(切入点) -- 所谓切入点是指我们要对哪些Joinpoint进行拦截的定义  3. Advice(通知/增强) -- 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)  4. Introduction(引介) -- 引介是一种特殊的通知在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field  5. Target(目标对象) -- 代理的目标对象  6. Weaving(织入) -- 是指把增强应用到目标对象来创建新的代理对象的过程  7. Proxy（代理） -- 一个类被AOP织入增强后，就产生一个结果代理类  8. Aspect(切面) -- 是切入点和通知的结合，以后咱们自己来编写和配置的 |

看图理解：



# AOP联盟通知类型

|  |
| --- |
| * AOP联盟为通知Advice定义了org.aopalliance.aop.Advice * Spring按照通知Advice在目标类方法的连接点位置，可以分为5类   + 前置通知 org.springframework.aop.MethodBeforeAdvice     - 在目标方法执行前实施增强   + 后置通知 org.springframework.aop.AfterReturningAdvice     - 在目标方法执行后实施增强   + 环绕通知 org.aopalliance.intercept.MethodInterceptor     - 在目标方法执行前后实施增强   + 异常抛出通知 org.springframework.aop.ThrowsAdvice     - 在方法抛出异常后实施增强   + 引介通知 org.springframework.aop.IntroductionInterceptor     - 在目标类中添加一些新的方法和属性 |

|  |
| --- |
| 环绕通知，必须手动执行目标方法  try{  //前置通知  //执行目标方法  //后置通知  } catch(){  //抛出异常通知  } |

# spring aop编程：【掌握】

导包：spring的传统AOP的开发的包

\* spring-aop-4.2.4.RELEASE.jar

\* com.springsource.org.aopalliance-1.0.0.jar

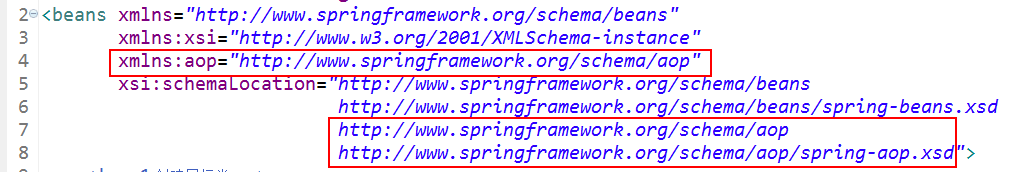
\* aspectJ的开发包

\* com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

\* spring-aspects-4.2.4.RELEASE.jar

* 从spring容器获得目标类，如果配置aop，spring将自动生成代理。
* 要确定目标类，aspectj 切入点表达式，导入jar包

### spring配置



|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd"*>  <!-- 1 创建目标类 -->  <bean id=*"userServiceId"* class=*"com.qf.c\_spring\_aop.UserServiceImpl"*></bean>  <!-- 2 创建切面类（通知） -->  <bean id=*"myAspectId"* class=*"com.qf.c\_spring\_aop.MyAspect"*></bean>  <!-- 3 aop编程  3.1 导入命名空间  3.2 使用 <aop:config>进行配置  proxy-target-class="true" 声明时使用cglib代理  <aop:pointcut> 切入点 ，从目标对象获得具体方法  <aop:advisor> 特殊的切面，只有一个通知 和 一个切入点  advice-ref 通知引用  pointcut-ref 切入点引用  3.3 切入点表达式  execution(\* com.qf.c\_spring\_aop.\*.\*(..))  选择方法 返回值任意 包 类名任意 方法名任意 参数任意    -->  <aop:config proxy-target-class=*"true"*>  <aop:pointcut expression=*"execution(\* com.qf.c\_spring\_aop.\*.\*(..))"* id=*"myPointCut"*/>  <aop:advisor advice-ref=*"myAspectId"* pointcut-ref=*"myPointCut"*/>  </aop:config>  </beans> |

### 测试

|  |
| --- |
| @Test  **public** **void** demo01(){  String xmlPath = "com/qf/c\_spring\_aop/beans.xml";  ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext(xmlPath);    //获得目标类  UserService userService = (UserService) applicationContext.getBean("userServiceId");  userService.addUser();  userService.updateUser();  userService.deleteUser();  } |

# AspectJ

## 介绍

* AspectJ是一个基于Java语言的AOP框架
* Spring2.0以后新增了对AspectJ切点表达式支持
* @AspectJ 是AspectJ1.5新增功能，通过JDK5注解技术，允许直接在Bean类中定义切面

新版本Spring框架，建议使用AspectJ方式来开发AOP

* 主要用途：自定义开发

## 切入点表达式【掌握】

1.execution() 用于描述方法 【掌握】

语法：execution(修饰符 返回值 包.类.方法名(参数) throws异常)

修饰符，一般省略

public 公共方法

\* 任意

返回值，不能省略

void 返回没有值

String 返回值字符串

\* 任意

包，[省略]

com.qf.crm 固定包

com.qf.crm.\*.service crm包下面子包任意 （例如：com.qf.crm.service）

com.qf.crm.. crm包下面的所有子包（含自己）

com.qf.crm.\*.service.. crm包下面任意子包，固定目录service，service目录任意包

类，[省略]

UserServiceImpl 指定类

\*Impl 以Impl结尾

User\* 以User开头

\* 任意

方法名，不能省略

addUser 固定方法

add\* 以add开头

\*Do 以Do结尾

\* 任意

(参数)

() 无参

(int) 一个整型

(int ,int) 两个

(..) 参数任意

throws ,可省略，一般不写。

**execution(\* com.qf.crm.\*.service..\*.\*(..))**

## AspectJ 通知类型

* aop联盟定义通知类型，具有特性接口，必须实现，从而确定方法名称。
* aspectj 通知类型，只定义类型名称。和方法格式。
* 个数：6种，知道5种，掌握1中。

before:前置通知(应用：各种校验)

在方法执行前执行，如果通知抛出异常，阻止方法运行

afterReturning:后置通知(应用：常规数据处理)

方法正常返回后执行，如果方法中抛出异常，通知无法执行

必须在方法执行后才执行，所以可以获得方法的返回值。

around:环绕通知(应用：十分强大，可以做任何事情)

方法执行前后分别执行，可以阻止方法的执行

必须手动执行目标方法

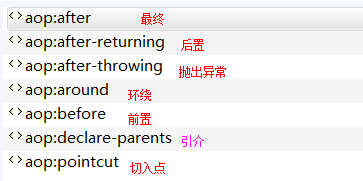
afterThrowing:抛出异常通知(应用：包装异常信息)

方法抛出异常后执行，如果方法没有抛出异常，无法执行

after:最终通知(应用：清理现场)

方法执行完毕后执行，无论方法中是否出现异常

|  |
| --- |
| 环绕  try{  //前置：before  //手动执行目标方法  //后置：afterRetruning  } catch(){  //抛出异常 afterThrowing  } finally{  //最终 after  } |



## 基于xml

### 1. 步骤一：创建JavaWEB项目，引入具体的开发的jar包

\* 先引入Spring框架开发的基本开发包

\* 再引入Spring框架的AOP的开发包

\* spring的传统AOP的开发的包

\* spring-aop-4.2.4.RELEASE.jar

\* com.springsource.org.aopalliance-1.0.0.jar

\* aspectJ的开发包

\* com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

\* spring-aspects-4.2.4.RELEASE.jar

### 2. 步骤二：创建Spring的配置文件，引入具体的AOP的schema约束

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"> |

### 步骤三：编写切面类

|  |
| --- |
| /\*\*  \* 切面类，含有多个通知  \*/  **public** **class** MyAspect {    **public** **void** myBefore(JoinPoint joinPoint){  System.*out*.println("前置通知 ： " + joinPoint.getSignature().getName());  }    **public** **void** myAfterReturning(JoinPoint joinPoint,Object ret){  System.*out*.println("后置通知 ： " + joinPoint.getSignature().getName() + " , -->" + ret);  }    **public** Object myAround(ProceedingJoinPoint joinPoint) **throws** Throwable{  System.*out*.println("前");  //手动执行目标方法  Object obj = joinPoint.proceed();    System.*out*.println("后");  **return** obj;  }    **public** **void** myAfterThrowing(JoinPoint joinPoint,Throwable e){  System.*out*.println("抛出异常通知 ： " + e.getMessage());  }    **public** **void** myAfter(JoinPoint joinPoint){  System.*out*.println("最终通知");  }  } |

### 步骤四：spring配置

**编写目标类，切面类 aop的配置**

|  |
| --- |
| <!-- 1 创建目标类 -->  <bean id=*"userServiceId"* class=*"com.qf.d\_aspect.a\_xml.UserServiceImpl"*></bean>  <!-- 2 创建切面类（通知） -->  <bean id=*"myAspectId"* class=*"com.qf.d\_aspect.a\_xml.MyAspect"*></bean>  <!-- 3 aop编程  <aop:aspect> 将切面类 声明“切面”，从而获得通知（方法）  ref 切面类引用  <aop:pointcut> 声明一个切入点，所有的通知都可以使用。  expression 切入点表达式  id 名称，用于其它通知引用  -->  <aop:config>  <aop:aspect ref=*"myAspectId"*>  <aop:pointcut expression=*"execution(\* com.qf.d\_aspect.a\_xml.UserServiceImpl.\*(..))"* id=*"myPointCut"*/>    <!-- 3.1 前置通知  <aop:before method="" pointcut="" pointcut-ref=""/>  method : 通知，及方法名  pointcut :切入点表达式，此表达式只能当前通知使用。  pointcut-ref ： 切入点引用，可以与其他通知共享切入点。  通知方法格式：public void myBefore(JoinPoint joinPoint){  参数1：org.aspectj.lang.JoinPoint 用于描述连接点（目标方法），获得目标方法名等  例如：  <aop:before method="myBefore" pointcut-ref="myPointCut"/>  -->    <!-- 3.2后置通知 ,目标方法后执行，获得返回值  <aop:after-returning method="" pointcut-ref="" returning=""/>  returning 通知方法第二个参数的名称  通知方法格式：public void myAfterReturning(JoinPoint joinPoint,Object ret){  参数1：连接点描述  参数2：类型Object，参数名 returning="ret" 配置的  例如：  <aop:after-returning method="myAfterReturning" pointcut-ref="myPointCut" returning="ret" />  -->    <!-- 3.3 环绕通知  <aop:around method="" pointcut-ref=""/>  通知方法格式：public Object myAround(ProceedingJoinPoint joinPoint) throws Throwable{  返回值类型：Object  方法名：任意  参数：org.aspectj.lang.ProceedingJoinPoint  抛出异常  执行目标方法：Object obj = joinPoint.proceed();  例如：  <aop:around method="myAround" pointcut-ref="myPointCut"/>  -->  <!-- 3.4 抛出异常  <aop:after-throwing method="" pointcut-ref="" throwing=""/>  throwing ：通知方法的第二个参数名称  通知方法格式：public void myAfterThrowing(JoinPoint joinPoint,Throwable e){  参数1：连接点描述对象  参数2：获得异常信息，类型Throwable ，参数名由throwing="e" 配置  例如：  <aop:after-throwing method="myAfterThrowing" pointcut-ref="myPointCut" throwing="e"/>  -->  <!-- 3.5 最终通知 -->  <aop:after method=*"myAfter"* pointcut-ref=*"myPointCut"*/>        </aop:aspect>  </aop:config> |

### 步骤五：完成测试

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**)  @ContextConfiguration("classpath:com/qf/test/beans.xml")  **public** **class** TestAop {  @Resource(name="userDao")  **private** UserDao userDao;  @Test  **public** **void** run1(){  userDao.add();  }  } |

## AOP基于注解

### S 引入相关的jar包:

\* spring的传统AOP的开发的包

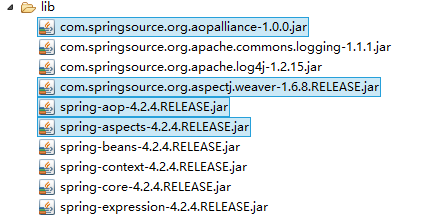
spring-aop-4.2.4.RELEASE.jar

com.springsource.org.aopalliance-1.0.0.jar

\* aspectJ的开发包:

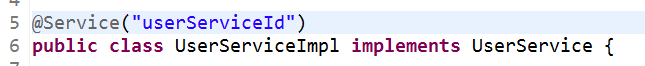
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

spring-aspects-4.2.4.RELEASE.jar



### 替换bean

|  |
| --- |
| <!-- 1 创建目标类 -->  <bean id=*"userServiceId"* class=*"com.qf.d\_aspect.b\_anno.UserServiceImpl"*></bean>  <!-- 2 创建切面类（通知） -->  <bean id=*"myAspectId"* class=*"com.qf.d\_aspect.b\_anno.MyAspect"*></bean> |





* 注意：扫描

|  |
| --- |
| <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"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd"*>  <!-- 1.扫描 注解类 -->  <context:component-scan base-package=*"com.qf.d\_aspect.b\_anno"*></context:component-scan> |

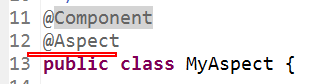
### 替换aop

* 必须进行aspectj 自动代理

|  |
| --- |
| <!-- 2.确定 aop注解生效 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy> |

* 声明切面

|  |
| --- |
| <aop:aspect ref=*"myAspectId"*> |



* 替换前置通知

|  |
| --- |
| <aop:before method=*"myBefore"* pointcut=*"execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))"*/> |

|  |
| --- |
| //切入点当前有效  @Before("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **public** **void** myBefore(JoinPoint joinPoint){  System.*out*.println("前置通知 ： " + joinPoint.getSignature().getName());  } |

* 替换 公共切入点

|  |
| --- |
| <aop:pointcut expression=*"execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))"* id=*"myPointCut"*/> |

|  |
| --- |
| //声明公共切入点  @Pointcut("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **private** **void** myPointCut(){  } |

* 替换后置

|  |
| --- |
| <aop:after-returning method=*"myAfterReturning"* pointcut-ref=*"myPointCut"* returning=*"ret"* /> |

|  |
| --- |
| @AfterReturning(value="myPointCut()" ,returning="ret")  **public** **void** myAfterReturning(JoinPoint joinPoint,Object ret){  System.*out*.println("后置通知 ： " + joinPoint.getSignature().getName() + " , -->" + ret);  } |

* 替换环绕

|  |
| --- |
| <aop:around method=*"myAround"* pointcut-ref=*"myPointCut"*/> |

|  |
| --- |
| @Around(value = "myPointCut()")  **public** Object myAround(ProceedingJoinPoint joinPoint) **throws** Throwable{  System.*out*.println("前");  //手动执行目标方法  Object obj = joinPoint.proceed();    System.*out*.println("后");  **return** obj;  } |

* 替换抛出异常

|  |
| --- |
| <aop:after-throwing method=*"myAfterThrowing"* pointcut=*"execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))"* throwing=*"e"*/> |

|  |
| --- |
| @AfterThrowing(value="execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))" ,throwing="e")  **public** **void** myAfterThrowing(JoinPoint joinPoint,Throwable e){  System.*out*.println("抛出异常通知 ： " + e.getMessage());  } |

### 切面类

|  |
| --- |
| /\*\*  \* 切面类，含有多个通知  \*/  @Component  @Aspect  **public** **class** MyAspect {    //切入点当前有效  // @Before("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **public** **void** myBefore(JoinPoint joinPoint){  System.*out*.println("前置通知 ： " + joinPoint.getSignature().getName());  }    //声明公共切入点  @Pointcut("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **private** **void** myPointCut(){  }    // @AfterReturning(value="myPointCut()" ,returning="ret")  **public** **void** myAfterReturning(JoinPoint joinPoint,Object ret){  System.*out*.println("后置通知 ： " + joinPoint.getSignature().getName() + " , -->" + ret);  }    // @Around(value = "myPointCut()")  **public** Object myAround(ProceedingJoinPoint joinPoint) **throws** Throwable{  System.*out*.println("前");  //手动执行目标方法  Object obj = joinPoint.proceed();    System.*out*.println("后");  **return** obj;  }    // @AfterThrowing(value="execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))" ,throwing="e")  **public** **void** myAfterThrowing(JoinPoint joinPoint,Throwable e){  System.*out*.println("抛出异常通知 ： " + e.getMessage());  }    @After("myPointCut()")  **public** **void** myAfter(JoinPoint joinPoint){  System.*out*.println("最终通知");  }  } |

### AOP注解启动配置

|  |
| --- |
| <!-- 1.扫描 注解类 -->  <context:component-scan base-package=*"com.qf.d\_aspect.b\_anno"*></context:component-scan>    <!-- 2.确定 aop注解生效 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy> |

### aop注解总结

@Aspect 声明切面，修饰切面类，从而获得 通知。

通知

@Before 前置

@AfterReturning 后置

@Around 环绕

@AfterThrowing 抛出异常

@After 最终

切入点

@PointCut ，修饰方法 private void xxx(){} 之后通过“方法名”获得切入点引用

# Spring框架的JDBC模板技术

## Spring框架的JDBC模板技术概述

**1. Spring框架中提供了很多持久层的模板类来简化编程，使用模板类编写程序会变的简单**

**2. 提供了JDBC模板，Spring框架提供的**

**JdbcTemplate类**

**3. Spring框架可以整合Hibernate框架，也提供了模板类**

**HibernateTemplate类**

## JDBC的模板类

**1. 步骤一：创建数据库的表结构**

create database spring\_day03;

use spring\_day03;

create table t\_account(

id int primary key auto\_increment,

name varchar(20),

money double

);

**2. 引入开发的jar包**

先引入IOC基本的6个jar包

再引入Spring-aop的jar包

最后引入JDBC模板需要的jar包

MySQL数据库的驱动包

Spring-jdbc.jar spring整合jdbc的包

Spring-tx.jar spring整合事务的包

**3. 编写测试代码（自己来new对象的方式）**

@Test

public void run1(){

// 创建连接池，先使用Spring框架内置的连接池

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring\_day03");

dataSource.setUsername("root");

dataSource.setPassword("root");

// 创建模板类

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

// 完成数据的添加

jdbcTemplate.update("insert into t\_account values (null,?,?)", "测试",10000);

}

## 使用Spring框架来管理模板类

**1. 刚才编写的代码使用的是new的方式，应该把这些类交给Spring框架来管理。**

2. 修改的步骤如下

**步骤一：Spring管理内置的连接池**

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql:///spring\_day03"/>

<property name="username" value="root"/>

<property name="password" value="root"/>

</bean>

步骤二：Spring管理模板类

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

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

</bean>

步骤三：编写测试程序

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext.xml")

public class Demo2 {

@Resource(name="jdbcTemplate")

private JdbcTemplate jdbcTemplate;

@Test

public void run2(){

jdbcTemplate.update("insert into t\_account values (null,?,?)", "测试2",10000);

}

}

## Spring框架管理开源的连接池

1. 管理DBCP连接池

先引入DBCP的2个jar包

com.springsource.org.apache.commons.dbcp-1.2.2.osgi.jar

com.springsource.org.apache.commons.pool-1.5.3.jar

编写配置文件

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql:///spring\_day03"/>

<property name="username" value="root"/>

<property name="password" value="root"/>

</bean>

2. 管理C3P0连接池

先引入C3P0的jar包

com.springsource.com.mchange.v2.c3p0-0.9.1.2.jar

\* 编写配置文件

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"/>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_day03"/>

<property name="user" value="root"/>

<property name="password" value="root"/>

</bean>

## Spring框架的JDBC模板的简单操作

**1. 增删改查的操作**

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext.xml")

public class SpringDemo3 {

@Resource(name="jdbcTemplate")

private JdbcTemplate jdbcTemplate;

@Test

// 插入操作

public void demo1(){

jdbcTemplate.update("insert into account values (null,?,?)", "二狗子",1000d);

}

@Test

// 修改操作

public void demo2(){

jdbcTemplate.update("update account set name=?,money =? where id = ?", "jack",10000d,5);

}

@Test

// 删除操作

public void demo3(){

jdbcTemplate.update("delete from account where id = ?", 5);

}

@Test

// 查询一条记录

public void demo4(){

Account account = jdbcTemplate.queryForObject("select \* from account where id = ?", new BeanMapper(), 1);

System.out.println(account);

}

@Test

// 查询所有记录

public void demo5(){

List<Account> list = jdbcTemplate.query("select \* from t\_account", new BeanMapper());

for (Account account : list) {

System.out.println(account);

}

}

}

class BeanMapper implements RowMapper<Account>{

public Account mapRow(ResultSet rs, int arg1) throws SQLException {

Account account = new Account();

account.setId(rs.getInt("id"));

account.setName(rs.getString("name"));

account.setMoney(rs.getDouble("money"));

return account;

}

}

# Spring框架的事务管理

## 事务的回顾

#### 1. 什么是事务

事务就是逻辑上的一组操作，组成事务的各个执行单元，操作要么全都成功，要么全都失败.

**转账**的例子：赵信给狐狸转钱，扣钱，加钱。两个操作组成了一个事情！

#### 2. 事务的特性 ACID

原子性（Atomicity） -- 事务不可分割. 整体

在事务中的多条sql语句是一个整体，要么就全部执行成功，有一条执行失败，都不执行。

一致性 （Consistency）-- 事务执行的前后数据的完整性保持一致. 数据的完整等性。赵信给狐狸转账2000块。能量守恒

隔离性 （Isolation）-- 一个事务执行的过程中,不应该受到其他的事务的干扰. 关于事务的隔离性数据库提供了多种隔离级别

持久性 （Durability）-- 事务一旦提交,数据就永久保持到数据库中.

#### 3. 如果不考虑隔离性:引发一些读的问题

脏读 -- 一个事务读到了另一个事务未提交的数据.

**张三和李四做交易，张三给李四转账，但是没有提交事务，李四这个时候去查询数据，这个时候读到了张三还有提交的数据，张三回滚事务，导致李四在查询钱少了。**

不可重复读 -- 一个事务读到了另一个事务已经提交的update数据,导致多次查询结果不一致.

张三在取钱的时候，第一查询余额是1万，这个事务李四取走了1万块，导致张三在取钱的时候，取不出来。张三在一个事务中，两次查询的结果不一样，因为中间被另外一个事务做了update操作。

幻读/虚读 -- 一个事务读到了另一个事务已经提交的insert数据,导致多次查询结构不一致.

#### 4. 通过设置数据库的隔离级别来解决上述读的问题

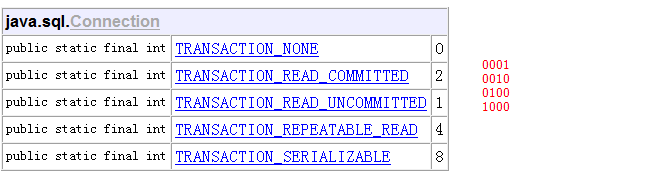
数据库为我们提供的四种隔离级别：

　　 Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。 8

　　 Repeatable read (可重复读)：可避免脏读、不可重复读的发生。 4

　　 Read committed (读已提交)：可避免脏读的发生。 2

　　 Read uncommitted (读未提交)：最低级别，任何情况都无法保证。 1



#### 5. 如果想在Hibernate的框架中来设置隔离级别，需要在hibernate.cfg.xml的配置文件中通过标签来配置

通过：hibernate.connection.isolation = 4 来配置

取值

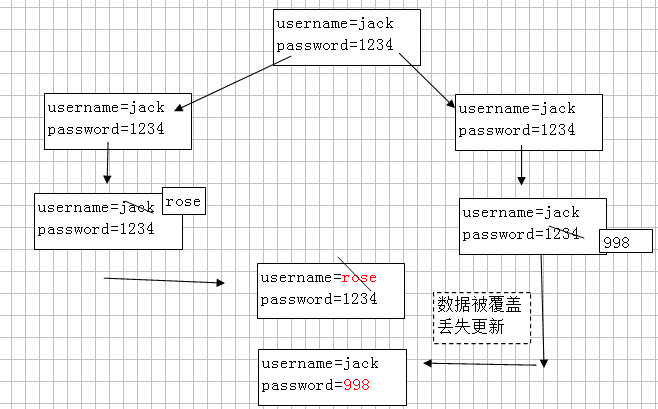
1—Read uncommitted isolation

2—Read committed isolation

4—Repeatable read isolation

8—Serializable isolation

### lost update 丢失更新



* 悲观锁：

顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如 读锁，写锁等，都是在做操作之前先上锁。

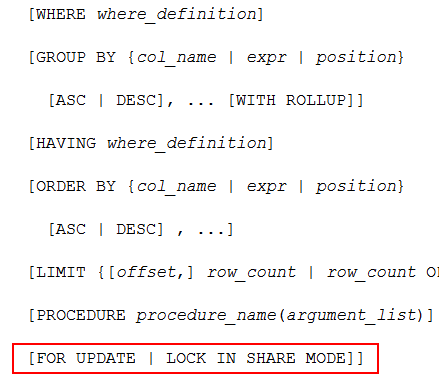
采用数据库锁机制。

读锁：共享锁。该锁可以共享

select .... from ... lock in share mode;

写锁：排他锁。（独占） 只能有一个线程拥有该锁。

select ... from .... for update



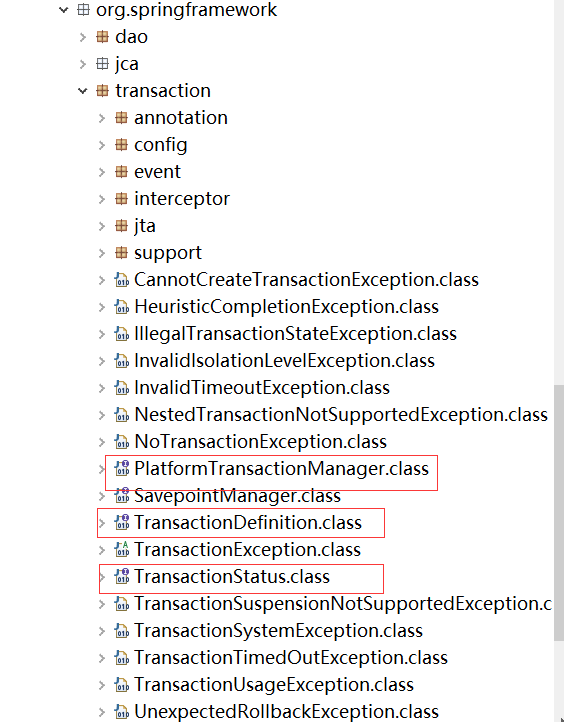
* 乐观锁：

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。

在表中提供一个字段（版本字段），用于标识记录。如果版本不一致，不允许操作。

在数据库变里面整个version字段，0，

## Spring框架的事务管理相关的类和API



**1、 PlatformTransactionManager 平台事务管理器，spring要管理事务，必须使用事务管理器**

**进行事务配置时，必须配置事务管理器**。

 **2、TransactionDefinition：事务详情（事务定义、事务属性），spring用于确定事务具体详情，**

**例如：隔离级别、是否只读、超时时间 等**

**进行事务配置时，必须配置详情。spring将配置项封装到该对象实例。**

**3、TransactionStatus：事务状态，spring用于记录当前事务运行状态。例如：是否有保存点，事务是否完成。** spring底层根据状态进行相应操作。

4. 总结：上述对象之间的关系：平台事务管理器真正管理事务对象.根据事务定义的信息TransactionDefinition 进行事务管理，在管理事务中产生一些状态.将状态记录到TransactionStatus中

**5. PlatformTransactionManager接口中实现类和常用的方法**

1. 接口的实现类

\* 如果使用的Spring的JDBC模板或者MyBatis框架，需要选择DataSourceTransactionManager实现类

\* 如果使用的是Hibernate的框架，需要选择HibernateTransactionManager实现类

2. 该接口的常用方法

\* void commit(TransactionStatus status)

\* TransactionStatus getTransaction(TransactionDefinition definition)

\* void rollback(TransactionStatus status)

6. TransactionDefinition

1. 事务隔离级别的常量

\* static int ISOLATION\_DEFAULT -- 采用数据库的默认隔离级别

\* static int ISOLATION\_READ\_UNCOMMITTED

\* static int ISOLATION\_READ\_COMMITTED

\* static int ISOLATION\_REPEATABLE\_READ

\* static int ISOLATION\_SERIALIZABLE

2. 事务的传播行为常量（不用设置，使用默认值）

\* 先解释什么是事务的传播行为：解决的是业务层之间的方法调用！！

\* PROPAGATION\_REQUIRED（默认值） -- A中有事务,使用A中的事务.如果没有，B就会开启一个新的事务,将A包含进来.(保证A,B在同一个事务中)，默认值！！

\* PROPAGATION\_SUPPORTS -- A中有事务,使用A中的事务.如果A中没有事务.那么B也不使用事务.

\* PROPAGATION\_MANDATORY -- A中有事务,使用A中的事务.如果A没有事务.抛出异常.

\* PROPAGATION\_REQUIRES\_NEW（记）-- A中有事务,将A中的事务挂起.B创建一个新的事务.(保证A,B没有在一个事务中)

\* PROPAGATION\_NOT\_SUPPORTED -- A中有事务,将A中的事务挂起.

\* PROPAGATION\_NEVER -- A中有事务,抛出异常.

\* **PROPAGATION\_NESTED**（记） -- 嵌套事务.当A执行之后,就会在这个位置设置一个保存点.如果B没有问题.执行通过.如果B出现异常,运行客户根据需求回滚(选择回滚到保存点或者是最初始状态)

## 搭建事务管理转账案例的环境（强调：简化开发，以后DAO可以继承JdbcDaoSupport类）

1. 步骤一：创建WEB工程，引入需要的jar包

\* IOC的6个包

\* AOP的4个包

\* C3P0的1个包

\* MySQL的驱动包

\* JDBC目标2个包

\* 整合JUnit测试包

2. 步骤二：引入配置文件

\* 引入配置文件

\* 引入log4j.properties

\* 引入applicationContext.xml

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"/>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_day03"/>

<property name="user" value="root"/>

<property name="password" value="root"/>

</bean>

3. 步骤三：创建对应的包结构和类

\* com.qf.demo1

\* AccountService

\* AccountServlceImpl

\* AccountDao

\* AccountDaoImpl

4. 步骤四:引入Spring的配置文件,将类配置到Spring中

<bean id="accountService" class="com.qf.demo1.AccountServiceImpl">

</bean>

<bean id="accountDao" class="com.qf.demo1.AccountDaoImpl">

</bean>

5. 步骤五：在业务层注入DAO ,在DAO中注入JDBC模板（强调：简化开发，以后DAO可以继承JdbcDaoSupport类）

<bean id="accountService" class="com.qf.demo1.AccountServiceImpl">

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

</bean>

<bean id="accountDao" class="com.qf.demo1.AccountDaoImpl">

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

</bean>

6. 步骤六：编写DAO和Service中的方法

public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {

public void outMoney(String out, double money) {

this.getJdbcTemplate().update("update t\_account set money = money = ? where name = ?", money,out);

}

public void inMoney(String in, double money) {

this.getJdbcTemplate().update("update t\_account set money = money + ? where name = ?", money,in);

}

}

7. 步骤七：编写测试程序.

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext.xml")

public class Demo1 {

@Resource(name="accountService")

private AccountService accountService;

@Test

public void run1(){

accountService.pay("冠希", "美美", 1000);

}

}

## Spring框架的事务管理的分类

1. Spring的事务管理的分类

1. Spring的编程式事务管理（不推荐使用）

通过手动编写代码的方式完成事务的管理（不推荐）

2. Spring的声明式事务管理（底层采用AOP的技术）

通过一段配置的方式完成事务的管理（重点掌握注解的方式）

## Spring框架的事务管理之编程式的事务管理（了解）

1. 说明：Spring为了简化事务管理的代码:提供了模板类 TransactionTemplate，所以手动编程的方式来管理事务，只需要使用该模板类即可！！

2. 手动编程方式的具体步骤如下：

1. 步骤一:配置一个事务管理器，Spring使用PlatformTransactionManager接口来管理事务，所以咱们需要使用到他的实现类！！

<!-- 配置事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

2. 步骤二:配置事务管理的模板

<!-- 配置事务管理的模板 -->

<bean id="transactionTemplate" class="org.springframework.transaction.support.TransactionTemplate">

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

</bean>

3. 步骤三:在需要进行事务管理的类中,注入事务管理的模板.

<bean id="accountService" class="com.qf.demo1.AccountServiceImpl">

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

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

</bean>

4. 步骤四:在业务层使用模板管理事务:

// 注入事务模板对象

private TransactionTemplate transactionTemplate;

public void setTransactionTemplate(TransactionTemplate transactionTemplate) {

this.transactionTemplate = transactionTemplate;

}

public void pay(final String out, final String in, final double money) {

transactionTemplate.execute(new TransactionCallbackWithoutResult() {

protected void doInTransactionWithoutResult(TransactionStatus status) {

// 扣钱

accountDao.outMoney(out, money);

int a = 10/0;

// 加钱

accountDao.inMoney(in, money);

}

});

}

## Spring框架的事务管理之声明式事务管理

即通过配置文件来完成事务管理(AOP思想)

1. 声明式事务管理又分成两种方式

基于AspectJ的XML方式（重点掌握）

基于AspectJ的注解方式（重点掌握）

### Spring框架的事务管理之基于AspectJ的XML方式（重点掌握）

1. 步骤一:恢复转账开发环境

2. 步骤二:引入AOP的开发包

3. 步骤三:配置事务管理器

<!-- 配置事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

4. 步骤四:配置事务增强

<!-- 配置事务增强 -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<!--

name ：绑定事务的方法名，可以使用通配符，可以配置多个。

propagation ：传播行为

isolation ：隔离级别

read-only ：是否只读

timeout ：超时信息

rollback-for：发生哪些异常回滚.

no-rollback-for：发生哪些异常不回滚.

-->

<!-- 哪些方法加事务 -->

<tx:method name="pay" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

5. 步骤五:配置AOP的切面

<!-- 配置AOP切面产生代理 -->

<aop:config>

<aop:advisor advice-ref="myAdvice" pointcut="execution(\* com.qf.demo2.AccountServiceImpl.pay(..))"/>

</aop:config>

**注意：如果是自己编写的切面，使用<aop:aspect>标签，如果是系统制作的，使用<aop:advisor>标签。**

6. 步骤六:编写测试类

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext2.xml")

public class Demo2 {

@Resource(name="accountService")

private AccountService accountService;

@Test

public void run1(){

accountService.pay("冠希", "美美", 1000);

}

}

## Spring框架的事务管理之基于AspectJ的注解方式（重点掌握，最简单的方式）

1. 步骤一:恢复转账的开发环境

2. 步骤二:配置事务管理器

<!-- 配置事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

3. 步骤三:开启注解事务

<!-- 开启注解事务 -->

<tx:annotation-driven transaction-manager="transactionManager"/>

4. 步骤四:在业务层上添加一个注解:@Transactional

5. 编写测试类

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext3.xml")

public class Demo3 {

@Resource(name="accountService")

private AccountService accountService;

@Test

public void run1(){

accountService.pay("冠希", "美美", 1000);

}

}

# SSH框架开发的基本回顾

## 技术分析之SSH三大框架需要的jar包

1. Struts2框架

\* struts-2.3.24\apps\struts2-blank\WEB-INF\lib\\*.jar -- Struts2需要的所有jar包

\* struts2-spring-plugin-2.3.24.jar ---Struts2整合Spring的插件包

2. Hibernate框架

\* hibernate-release-5.0.7.Final\lib\required\\*.jar -- Hibernate框架需要的jar包

\* slf4j-api-1.6.1.jar -- 日志接口

\* slf4j-log4j12-1.7.2.jar -- 日志实现

\* mysql-connector-java-5.1.7-bin.jar -- MySQL的驱动包

3. Spring框架

\* IOC核心包

\* AOP核心包

\* JDBC模板和事务核心包

\* Spring整合JUnit测试包

\* Spring整合Hibernate核心包

\* Spring整合Struts2核心包

## SSH三大框架需要的配置文件

1. Struts2框架

\* 在web.xml中配置核心的过滤器

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

\* 在src目录下创建struts.xml，用来配置Action

2. Hibernate框架

\* 在src目录创建hibernate.cfg.xml配置文件

\* 在JavaBean所在的包下映射的配置文件

3. Spring框架

\* 在web.xml配置整合WEB的监听器

<listener>

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

</listener>

<context-param>

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

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

\* 在src目录下创建applicationContext.xml

\* 在src目录下log4j.proerties

## Spring框架整合Struts2框架

1. 导入CRM项目的UI页面，找到添加客户的页面，修改form表单，访问Action

2. 编写CustomerAction接收请求，在struts.xml中完成Action的配置

<package name="crm" extends="struts-default" namespace="/">

<action name="customer\_\*" class="com.qf.web.action.CustomerAction" method="{1}">

</action>

</package>

3. 在Action中获取到service（开发不会使用，因为麻烦）

\* 可以通过 WebApplicationContextUtils.getWebApplicationContext(ServletActionContext.getServletContext()); 来获取，但是这种方式编写代码太麻烦了！！

### 4. Spring整合Struts2框架的第一种方式（Action由Struts2框架来创建）

\* 因为导入的struts2-spring-plugin-2.3.24.jar 包自带一个配置文件 struts-plugin.xml ，该配置文件中有如下代码

\* <constant name="struts.objectFactory" value="spring" /> 开启一个常量，如果该常量开启，那么下面的常量就可以使用

\* struts.objectFactory.spring.autoWire = name，该常量是可以让Action的类来自动装配Bean对象！！

### 5. Spring整合Struts2框架的第二种方式（Action由Spring框架来创建）（推荐大家来使用的）

\* 把具体的Action类配置文件applicatonContext.xml的配置文件中，但是注意：struts.xml需要做修改

\* applicationContext.xml

\* <bean id="customerAction" class="com.qf.web.action.CustomerAction" scope="prototype">

\* struts.xml中的修改，把全路径修改成ID值

\* <action name="customer\_\*" class="customerAction" method="{1}">

\* 第二种方式需要有两个注意的地方

\* Spring框架默认生成CustomerAction是单例的，而Struts2框架是多例的。所以需要配置 scope="prototype"

\* CustomerService现在必须自己手动注入了

**strust2交给spring管理之后，自动注入就失效了**

## Spring框架整合Hibernate框架

### （带有hibernate.cfg.xml的配置文件。强调：不能加绑定当前线程的配置）

**用LocalSessionFactoryBean加载hibernate.cfg.xml配置文件**

1. 编写CustomerDaoImpl的代码，加入配置并且在CustomerServiceImpl中完成注入

2. 编写映射的配置文件，并且在hibernate.cfg.xml的配置文件中引入映射的配置文件

3. 在applicationContext.xml的配置文件，配置加载hibernate.cfg.xml的配置

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

<property name="configLocation" value="classpath:hibernate.cfg.xml"/>

</bean>

4. 在CustomerDaoImpl中想完成数据的添加，Spring框架提供了一个HibernateDaoSupport的工具类，以后DAO都可以继承该类！！

public class CustomerDaoImpl extends HibernateDaoSupport implements CustomerDao {

public void save(Customer c) {

System.out.println("持久层...");

this.getHibernateTemplate().save(c);

}

}

<bean id="customerDao" class="com.qf.dao.CustomerDaoImpl">

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

</bean>

5. 开启事务的配置

\* 先配置事务管理器，注意现在使用的是Hibernate框架，所以需要使用Hibernate框架的事务管理器

<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">

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

</bean>

\* 开启注解事务

<tx:annotation-driven transaction-manager="transactionManager"/>

\* 在Service类中添加事务注解

@Transactional

## Spring框架整合Hibernate框架

### （不带有hibernate.cfg.xml的配置文件）\*\*

1. Hibernate配置文件中

\* 数据库连接基本参数（4大参数）

\* Hibernate相关的属性

\* 连接池

\* 映射文件

2. 开始进行配置

\* 先配置连接池相关的信息

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"/>

<property name="jdbcUrl" value="jdbc:mysql:///xxx"/>

<property name="user" value="root"/>

<property name="password" value="root"/>

</bean>

\* 修改 LocalSessionFactoryBean 的属性配置，因为已经没有了hibernate.cfg.xml的配置文件，所以需要修改该配置，注入连接池

<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">

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

</bean>

\* 继续在 LocalSessionFactoryBean 中配置，使用hibernateProperties属性继续来配置其他的属性，注意值是properties属性文件

<!-- 配置其他的属性 -->

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>

<prop key="hibernate.show\_sql">true</prop>

<prop key="hibernate.format\_sql">true</prop>

<prop key="hibernate.hbm2ddl.auto">update</prop>

</props>

</property>

<!-- 配置映射 -->

<property name="mappingResources">

<list>

<value>com/qf/domain/Customer.hbm.xml</value>

</list>

</property>

## Hibernate的模板的常用的方法

1. 增删改的操作:

\* 添加:

\* save(Object obj);

\* 修改:

\* update(Object obj);

\* 删除:

\* delete(Object obj);

2. 查询的操作:

\* 查询一条记录:

\* Object get(Class c,Serializable id);

\* Object load(Class c,Serializable id);

3. 查询多条记录:

\* List find(String hql,Object... args);

## 延迟加载问题

1. 使用延迟加载的时候，再WEB 层查询对象的时候程序会抛出异常！

\* 原因是延迟加载还没有发生SQL语句，在业务层session对象就已经销毁了，所以查询到的JavaBean对象已经变成了托管态对象！

\* 注意：一定要先删除javassist-3.11.0.GA.jar包（jar包冲突了）

2. 解决办法非常简单，Spring框架提供了一个过滤器，让session对象在WEB层就创建，在WEB层销毁。只需要配置该过滤器即可

\* 但是：要注意需要在struts2的核心过滤器之前进行配置

<filter>

<filter-name>OpenSessionInViewFilter</filter-name>

<filter-class>org.springframework.orm.hibernate5.support.OpenSessionInViewFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>OpenSessionInViewFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

## 最终的：

44包

Spring整合struts2框架，action由spring创建。

Spring整合web的包，struts2整合spring的包

Spring整合hibarnate

不要hibernate.cfg.xml配置文件

Spring-orm.jar

一定要先删除javassist-3.11.0.GA.jar包（jar包冲突了）

配置OpenSessionInViewFilter过滤器，在struts2的核心过滤器之前配置，解决懒加载问题。