**-Spring设计思想**

dao

注入装配反转反射

Xml文件

Bean

Dbhelper

阿瑟大时代按时打算大声的

sever

(方法调用处理)

Entity

Userdao

Implement

多种方式实现了userdao

Userdao

（接口类）

Spring容器

**装配**

通过加载xm文件，或者annotation，在程序载入的时候便创建对象，要的时候可以直接调用，不必在用new关键字创建

<bean id=*"u"* class=*"com.bjsxt.dao.impl.UserDAOImpl"* />

**注入：**

在xml文件中或者annotation便指定两个类见对象的关系，（当一个类调用了另一个类的对象作为参数）在程序载入的时候便完成初始化，也就是在一个类里面有其他类的对象的时候，spring容器可以帮我们初始化这个被调用的对象（通过get，set方法，和注入）

下面的意思是，创建名字为userSever的*com.bjsxt.service.UserService*的对象，然后调用setuserDao方法，将*com.bjsxt.dao.impl.UserDAOImpl*的一个对象u作为参数执行setuserDao方法，从而初始化userDAO

<bean id=*"userService"*class=*"com.bjsxt.service.UserService"* >

<property name=*"userDAO"* ref=*"u"*/>//注入

</bean>

**反射：**

**在applicationContext中他会自动浏览xml文件，然后通过里面的类名和prototy里面的name值，通过解析，然后得到类名和方法名，然后创建对象**

1. 通过Class对象的newInstance()方法来创建Class对象对应类的实例。这个方法是使用Class对象对应类的默认构造器创建对象，这就要求Class对象对应类必须要有默认构造器。
2. 在spring这些框架中，会根据配置文件自动创建类的实例并注入到依赖此类的类中。这时候用的最多的就是默认构造器。像是在spring的配置文件中，我们提供的是某个类的全类名，这种情况要创建类的实例，就必须使用反射了。

**一些知识点：（重要）**

**接口不可以通过new关键字实例化创建对象，但是可以通过传入实现它的类的对象实现实例化。**

假设有个接口Userdao，有三个实现它的类Userdaoimp1，Userdaoimp2，Userdaoimp3，Userdaoimp4

那么

Userdao userdao = new Userdao（）

错误：接口不能用new实例化

Userdao userdao1 = new Userdaoimp1（）；

Userdao userdao2 = new Userdaoimp2（）；

Userdao userdao3 = new Userdaoimp3（）；

正确：用实现它的类的对象去初始化接口，这样的话接口便很灵活，可以在Userdaoimp4里面同时写上上面的三句，便可以通过三个对象调用Userdaoimp1，Userdaoimp2，Userdaoimp3

三个实现类的方法，还可以扩展自己的代码，实现多继承

**DI：dependcy inject （依赖注入）**

在spring中生成对象，和建立对象间的关系（一个类的里引用了别的类的对象，那么这两个类的对象就有关系）都依靠xml配置文件来实现，这样在配置文件里创造对象叫做注入，而spring的对象都是依赖于这种方法来生成，所以说spring依赖注入

**IOC：inverse of control（控制反转）**

* 1. 实例化具体bean
  2. 动态装配

解释1：

在以前的编程中我们自己new对象并且使用，资源控制权完完全全在程序员中，在运用spring后变成了对象由spring创建，资源分配spring容器控制

解释2

原来程序员在写程序的时候写实现，控制实现方式，现在变成了操控接口，是一个抽象的概念，由具体的实现到抽象的接口，叫做控制反转

解释3

原来具体的实现掌握在程序员的手中现在掌握在容器的手中

**注入的类型**

程序员在某个类中需要依赖其它类的方法，则通常是new一个依赖类再调用类实例的方法，这种开发存在的问题是new的类实例不好统一管理，spring提出了依赖注入的思想，即依赖类不由程序员实例化，而是通过spring容器帮我们new指定实例并且将实例注入到需要该对象的类中。依赖注入的另一种说法是“控制反转”，通俗的理解是：平常我们new一个实例，这个实例的控制权是我们程序员，而控制反转是指new实例工作不由我们程序员来做而是交给spring容器来做。

1. **set注入：**

在原来的方法里建立一个set方法作为入口，然后在配置文件里用注入的方法将两个类的对象联系起来，然后用get方法取出常（最常用，例子见上面的注案例）

1. **构造方法注入 ：**

2.1首先在构造方法里将要初始化的对象进行初始化

2.2在xml文件里进行装配

2.3案例

构造方法

**public** UserService(UserDAO userDAO) {

**super**();

**this**.userDAO = userDAO;

}

**装配**

<bean id=*"userService"*class=*"com.bjsxt.service.UserService"*>

//将u作为参数传入*userService*的构造方法进行初始化

<constructor-arg>

<ref bean=*"u"*/>

</constructor-arg>

</bean>

**小知识点**

**<bean id = “” class = “ ”> </bean>表示重新创建一个对象**

**<ref bean = “ ”>代表引用上面已经创建的对象**

id可以用 name替代，唯一区别是name里面可以包含一些特殊字符

Id和name同时存在的时候，id为他的标识，name为别名，

当只有name的时候，第一个name就是他的标识符

**字段注入**

**在对象上加上@autorwire**

**简单属性注入：**

在xml文件里创建对象和通过get和set方法给对象的属性赋值（一般不用）

**public** **class** UserDAOImpl **implements** UserDAO {

/\* (non-Javadoc)

\* @see com.bjsxt.dao.UserDAO#save(com.bjsxt.model.User)

\*/

**private** **int** daoId;

**private** String daoStatus;

**public** **int** getDaoId() {

**return** daoId;

}

**public** **void** setDaoId(**int** daoId) {

**this**.daoId = daoId;

}

**public** String getDaoStatus() {

**return** daoStatus;

}

**public** **void** setDaoStatus(String daoStatus) {

**this**.daoStatus = daoStatus;

}

**public** **void** save(User user) {

System.***out***.println("user saved!");

}

@Override

**public** String toString() {

**return** **this**.daoId + ":" + **this**.daoStatus;

}

}

Xml

<bean name=*"userDAO"* class=*"com.bjsxt.dao.impl.UserDAOImpl"*>

<property name=*"daoId"* value=*"8"*></property>//简单属性注入

<property name=*"daoStatus"* value=*"good"*></property>

</bean>

**Bean的作用范围：scope**

<bean id=*"userService"* class=*"com.bjsxt.service.UserService"* scope=*"prototype"*>

1. scope = singleton 单例模式，每次都用同一个对象
2. Scope = prototype 原型模式，每次调用都创建一个新的对象
3. Scope = session
4. Scope = application
5. Scope = request

//后面这三个与网站有关，马老师没讲

**Collection类型的注入**

在xml里面给collection装值，

（直接看eclipse上的程序）

自动装配:autowire ：

通过自动装配，可以无需手动注入，spring容器会将通过autowire自动将所需的属性装配

**1、通过name来装配**

Autowire = “byName

”,他会自动调用set方法然后再文件里找一个和对象属性同名的对象然后装配进来

**2、通过属性装配**

Autowire = “byType” 他会根据类型来找文件里是否有和它需要的类型一样的对象，然后装配进来

1. 写在最外围的beans里面，表示下面所有的bean都采用这个autowire方式

**Bean的生命周期？？？？？？重要（百度）**

Lazy-init = “true”:把这句话加入到某个bean里面，然后他在spring初始化容器的时候先不会初始化这个对象

<bean id="u" class="com.bjsxt.dao.impl.UserDAOImpl" lazy-init = “true” >

init-method,和destory-method

表示这个对象在spring容器创建然后自动初始化这个对象时候先执行init-method的方法，然后在结束的时候执行destory-method所指的方法，一般不和prototype连用

<bean id="u" class="com.bjsxt.dao.impl.UserDAOImpl" init-method = “init” destory-method = “destory” >

**非常重要@component和@resource**

**使用前先在xml加上这句话**

**<context:component-scan base-package="com.bjsxt"/>**

**则spring会扫描文件，然后找@component和@resource**

**@Component：自动初始化，不必在xml文件里再写<bean>来让spring容器初始化需要的对象，将component写在类的上面，那么spring容器在初始化的时候会自动生成一个他的对象**

@Component("u")

public class UserDAOImpl implements UserDAO {

public void save(User user) {

//Hibernate

//JDBC

//XML

//NetWork

System.out.println("user saved!");

}

}

//在spring容器初始化的时候自动生成UserDAOImpl的一个对象，然后起名为u

@resource（“value”）：一般写在set或者构造方法上面

表示在spring初始化这个项目的时候查找有没有一个叫“value”的对象，如果找不到，就按类型寻找有没有要求的类型的数据，如果找到了其中的一种就调用set方法注入，以componemt一起使用

@Resource(name="u")

**public** **void** setUserDAO( UserDAO userDAO) {

**this**.userDAO = userDAO;

}

//表示找到u，并且调用set方法注入对象

@qualiter

在使用按类型注入的时候可能会出现相同类型的对象不止一个，可以用这个标注要注入具体的哪个对象

**@Scope("prototype"):**

写在类的上面表示每次调用生成一个新的对象，不写的时候默认为

@Scope("singleton"):表示每次都使用同一个对象

**@PostConstruct**

当在某个类的方法上的时候，表示每次生成这个类的对象的时候，先执行这个类的这个方法。

@PostConstruct

public void init() {

System.out.println("init");

}

//每次生成init方法所对应的类的对象的时候先执行init方法

**@PreDestroy**

当在某个类的方法上的时候，表示每次销毁这个类的对象的时候，先执行这个类的这个方法。

@PreDestroy

public void destroy() {

System.out.println("destroy");

}

////每次销毁destory方法所对应的类的对象的时候先执行destory方法

**Spring容器执行后的返回值**

Spring执行后生成的**是一**个map，key为对象的名字，value为真实的对象

**组合的编程思想：**

当我们完成一个项目的编写并且已经提交了源代码，也就是源代码已经无法修改的情况下如何扩展某一个类

1、采用编写一个新的类继承需要扩展的类，然后调用super先执行父类方法，在增加新的业务逻辑

**这样的方式不够灵活不推荐**

**2、采用组合的方法**

在组合方式里，以接口为最顶级，然后业务逻辑写在他的实现它的类里面，当需要业务拓展的时候，

1、生成第一个实现类的对象然后调用这个类的方法，那么就完成了对第一个类的保存

1. 编写新的业务逻辑实现拓展

3、当再一次需要业务拓展的时候，可以同时生成接口一和接口二实现类的对象，然后调用各自的方法，从而快速实现业务拓展，但是继承却无法整合两个子类的东西，因为只能继承一个

**这样做的话就很方便并且很灵活的实现了业务拓展，因为接口比继承灵活**

**Aop面向切面的编程（**Aspect-Oriented-Programming：层面导向编程

1. 在我们以前的编程中采用直线思维，很有逻辑的做每一件事，这样做的话对于一些权限处理，错误处理、等等很多事情却要我们向横向编程，导致代码沉重，不好添加和删除，面向切面的编程在直线编程的基础上，将横向的部分抽出来，在一个方法前后动态地添加业务逻辑，和一些其他代码，，实现拓展和横向编程，这样做既灵活又高效，可以动态的添加和删除在切面上的逻辑而不影响原来的执行代码

**动态代理**：**在已经完成的类里面动态的插入其他代码，实现业务扩展**

1、在jdk提供的类里**必须是实现接口的类**才可以动态代理

2、用proxy和invorcationHander实现动态代理

**在原生的jdk中用静态方法Proxy和实现InvocationHandler接口来的类**

Proxy象一个传送带，将代码和参数传递到处理类

实现InvocationHandler接口来的类像一个工厂，给传过来的方前后添加方法

1. 处理过程：

1、userDAOProxy 是UserDAO的一个实现类的对象，当他调用UserDAO接口里面的方法的时候，就相当于调用了实现了InvocationHandler接口的类的invoke方法，然后通过li对象吧你调用的方法和参数都传递过来，你可以在invoke方法里先执行想要动添加的代码，然后执行原来的方法，从而实现动态添加的效果

**package** spring\_shangguigu\_8\_Proxy\_动态代理;

**import** java.lang.reflect.\*;

**import** java.lang.reflect.Proxy;

**import** java.net.\*;

**public** **class** getcatulateLogging{

**private** caculate target;

**public** getcatulateLogging(caculate target){

**this**.target = target;

}

**public** caculate getcatulateLoggingproxy(){

caculate proxy;

ClassLoader loader = target.getClass().getClassLoader();

Class[] inteface = **new** Class[]{caculate.**class**};

//处理方法

InvocationHandler h = **new** InvocationHandler() {

//第一个参数proxy不使用

//method 表示当前正在执行的方法

//args是方法的参数的数组

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

String methodname = method.getName();

//日志

System.***out***.println("the method "+methodname +"has start");

//利用invoke执行原方法

Object result = method.invoke(target,args); //利用原来对象执行方法

//日志

System.***out***.println("the method "+methodname +"has end");

System.***out***.println("result ="+result);

**return** result;

}

};

proxy =(caculate)Proxy.*newProxyInstance*(loader, inteface, h);

**return** proxy; //返回代理对象

}

}

**AspectJ：一个专门的面向切面的框架语言，也是专门面向aop编程的框架，spring使用了他**

1. 概念：
   1. JoinPoint 织入点
   2. PointCut 织入点集合
   3. Aspect（切面）织入的那个类
   4. Advice 织入类里面的方法
   5. Target 代理对象
   6. Weave 织入

**采用xml方式配置aop**

<bean id="logInterceptor" class="com.bjsxt.aop.LogInterceptor"></bean> //初始化生成处理类对象

<aop:config>

<aop:aspect id="logAspect" ref="logInterceptor">

<aop:before method="before" pointcut="execution(public \* com.bjsxt.service..\*.add(..))" />

</aop:aspect>

//利用logInterceptor在 com.bjsxt.service的任意类下面的add方法执行前加入要执行before方法

</aop:config>

**连接池DataSource**

DataSource是sun定下的一种接口，返回的是一个数据库连接，具体是那种数据库的连接，可以自己设定，

可以在xml配置文件里面建立多种连接不同数据库的方式，每个方式都返回一个DataSource数据，这样就可以实现连接不同的数据库

|  |  |  |
| --- | --- | --- |
| 参数 | 配置 | 说明 |
| initialSize | 3 | 初始化配置 |
| minIdle | 3 | 最小连接数 |
| maxIdle | 15 | 最大空闲连接 |
| maxTotal | 15 | 最大连接数 |
| maxWaitMillis | 5000 | 获取连接超时时间(单位ms) |

**在xml里面用dbcp（一个连接池的包）和IOC连接数据库**

1. **在xml文件里面配置数据库连接**

1-1：

Spring容器在初始化的时候会创建BasicDataSource的一个对象dataSource，通过这个对象就可以获得数据库连接

<bean id=*"dataSource"* destroy-method=*"close"*

class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"*

value=*"com.mysql.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/spring"* />

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

<property name=*"password"* value=*"123456"* />

</bean>

1-2

在需要连接数据库的类里面通过IOC注入拿到dataSource对象，

**private** DataSource data;

**public** DataSource getData() {

**return** data;

}

@Resource(name = "dataSource") //annotation注入

**public** **void** setData(DataSource data) {

**this**.data = data;

}

1-3

连接数据库进行操作

**public** **void** save(User user) {

**try**{

Connection conn = data.getConnection();

conn.createStatement().executeUpdate("insert into us values('15251102130','123456');");

conn.close();

}**catch**(Exception e){

e.printStackTrace();

}

}

总结：

1. IOC的概念
2. DI
3. @compenent和@resource
4. Aop
5. 面向切面的编程
6. 动态代理
7. 组合编程
8. 面向接口的编程
9. 装配
10. 注入
11. 各种标签

**Spring之尚硅谷**

1. **spring是什么**

**Spring是一个优秀的开源框架，spring为简化企业级开发而生，使用spring可以简单实现javabean，**

**Spring是一个IOC和AOP容器框架**

**特点：**

**轻量级：Spring 是非侵入性的 - 基于 Spring 开发的应用中的对象可以不**

**依赖于 Spring 的 API**

**依赖注入(DI --- dependency injection、IOC)**

**面向切面编程(AOP --- aspect oriented programming)**

**容器: Spring 是一个容器, 因为它包含并且管理应用对象的生命周期**

**框架: Spring 实现了使用简单的组件配置组合成一个复杂的应用. 在 Spring 中可以使用 XML 和 Java 注解组合这些对象**

**一站式：在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库 （实际上 Spring 自身也提供了展现层的 SpringMVC 和 持久层的 Spring JDBC）**

**1、1spring容器**

SpringIOC创建bean之前需要对spring进行实例

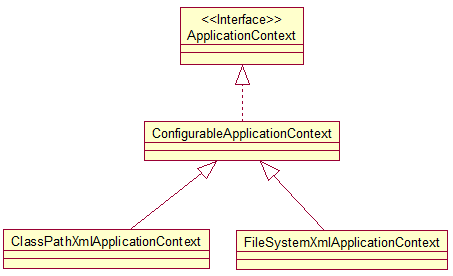
实例SpringIOC容器有两种方式

BeanFactory: IOC 容器的基本实现.

**ApplicationContext: 提供了更多的高级特性. 是 BeanFactory 的子接口**.

**BeanFactory 是 Spring 框架的基础设施，面向 Spring 本身；ApplicationContext 面向使用 Spring 框架的开发者，几乎所有的应用场合都直接使用 ApplicationContext 而非底层的 BeanFactory**

**1、2：ApplictionContext接口的树级结构**



ApplicationContext

在初始化上下文的时候就实例化所有单例的bean

ConfigurableApplicationContext

扩展于ApplictionContext的子接口，新增两个方法，refresh（）和close（），让ApplicationContext具有启动、刷新和关闭上下文的功能。

ClassPathXmlApplicationContext

ConfigurableApplicationContext的一个实现类，功能为从类路径下加载xml配置文件，完成bean初始化

FileSystemXmlApplicationContext

ConfigurableApplicationContext的一个实现类，表示从文件系统加载xml配置文件，完成bean初始化

1. **IOC**

**IOC(Inversion of Control)：其思想是反转资源获取的方向. 传统的资源查找方式要求组件向容器发起请求查找资源. 作为回应, 容器适时的返回资源. 而应用了 IOC 之后, 则是容器主动地将资源推送给它所管理的组件, 组件所要做的仅是选择一种合适的方式来接受资源. （反转资源控制的方式）**

**3、DI**

**DI(Dependency Injection) — IOC 的另一种表述方式：即组件以一些预先定义好的方式(例如: setter 方法)接受来自如容器的资源注入.可以应对集合、常量、对象的注入（把一个类的对象通过set方法注入到另一个类）**

1. **bean的配置方式一：xml，全类名配置方式（反射）**

**Exp：**

<bean id=*"helloword"* class=*"spring\_beans.Helloword"*> <property name=*"name"* value=*"zzx"*></property>

</bean>

4-1

在这种配置方式里面需要用到全类名“*spring\_beans.Helloword*”

通**过反射的方式调用无参数的构造方法在IOC容器里面创造bean，**要求在这个类里面必须要有一个**无参数的构造方法（默认构造器）**

**4-2**

Id是这个对象（bean）的标识，所以**id必须是唯一**的，容器将id和对象（bean）以键值对的方式存入map里面，取出的时候也是依靠getBean（id）来得到这个对象（bean）

1. **获取bean的方式**

**5-1：以id获取**

Helloword hello = (Helloword)ctx.getBean("helloword");

5-2：以类型获取

Helloword hello = ctx.getBean(Helloword.**class**);

这样做的时候可能会应因为相同类型的bean有多个，导致获取失败，需要通过@qualiter来指定获取哪个，比较麻烦

**6、注入的方法**

6-1：属性的注入

通过set方法注入常用数据类型，集合、属性，引用等（引用注入比较多）

exp：见马士兵版笔记

<bean id=*"u"* class=*"com.bjsxt.dao.impl.UserDAOImpl"* />

引用注入，使用的是上面已经创造好的

<bean id=*"userService"*class=*"com.bjsxt.service.UserService"* >

<property name=*"userDAO"* ref=*"u"*/>//注入

</bean>

内部bean注入，这个bean不能被外部调用,只能被usersever使用，是单独创造的bean

<bean id=*"userService"*class=*"com.bjsxt.service.UserService"* >

<property name=*"userDAO"* >//注入

<bean id=*"a"* class=*"com.bjsxt.dao.impl.UserDAOImpl"* />

<property/>

</bean>

在第二个对象userService里注入u，实现第二个对象可以获得第一个对象，换句话说就是userService可以调用u所在的类的方法

6-2构造方法注入（构造器注入）

用 <constructor-arg value = "asdasd"></constructor-arg>按顺序注入

Exp：见上面马士兵版笔记

6-3工厂方法注入

一般不用，记得有这么回事就好

6-4字段注入

1. **xml中的null为<null/>**
2. **给级联属性赋值（就是给被引用的对象的属性赋值）**

<bean id=*"userService"*class=*"com.bjsxt.service.UserService"* >

<property name=*"userDAO"* ref=*"u"*/>//注入

<property name=*"userDAO.name"* value=*"zzx"*/>//给级联属性赋值

</bean>

**给级联属性赋值必须要这个对象已经创建，**

**Id与name的区别**

1、每个Bean可以有一个id属性，并可以根据该id在IoC容器中查找该Bean，该id属性值必须在IoC容器中唯一；

2、可以不指定id属性，只指定全限定类名，如：

3、如果不指定id，只指定name，那么name为Bean的标识符，并且需要在容器中唯一；

4、同时指定name和id，此时id为标识符，而name为Bean的别名，两者都可以找到目标Bean；

5、可以指定多个name，之间可以用分号（“；”）、空格（“ ”）或逗号（“，”）分隔开，如果没有指定id，那么第一个name为标识符，其余的为别名；若指定了id属性，则id为标识符，所有的name均为别名。如：

6、<prototype name = “” ref=“”></prototype>标签里面只能用name，不用id

1. **给创建集合bean,下面是xml配置的经典案例**

<!-- 创建三辆车 -->

<bean id= *"baoma"* class = *"spring\_beans.car"*>

<property name=*"price"* value=*"1000000"*></property>

<property name=*"maxspeed"* value = *"250"*></property>

<property name=*"color"* value=*"green"*></property>

</bean>

<bean id= *"luhu"* class = *"spring\_beans.car"*>

<property name=*"price"* value=*"2000000"*></property>

<property name=*"maxspeed"* value = *"350"*></property>

<property name=*"color"* value=*"red"*></property>

</bean>

<bean id= *"falali"* class = *"spring\_beans.car"*>

<property name=*"price"* value=*"3000000"*></property>

<property name=*"maxspeed"* value = *"450"*></property>

<property name=*"color"* value=*"orange"*></property>

</bean>

<!-- 创建三个房子 -->

<bean id= *"house1"* class = *"spring\_beans.house"*>

<property name=*"price"* value= *"100000000"*></property>

<property name=*"area"* value= *"100m"*></property>

</bean>

<bean id= *"house2"* class = *"spring\_beans.house"*>

<property name=*"price"* value= *"200000000"*></property>

<property name=*"area"* value= *"200m"*></property>

</bean>

<bean id= *"house3"* class = *"spring\_beans.house"*>

<property name=*"price"* value= *"300000000"*></property>

<property name=*"area"* value= *"300m"*></property>

</bean>

<!--一只狗 -->

<bean id =*"dog"* class=*"spring\_beans.dog"*>

<property name=*"high"* value=*"100"*></property>

<property name=*"color"* value=*"blue"*></property>

</bean>

<!--创建car的集合类型的bean，（先在namespace里引入util） -->

<util:list id = *"cars"*>

<ref bean = *"baoma"*/>

<ref bean = *"falali"*/>

<ref bean = *"luhu"*/>

</util:list>

<!--创建house类型的map -->

<util:map id=*"houses"*>

<entry key=*"beijing"* value-ref=*"house1"*></entry>

<entry key=*"shanghai"* value-ref=*"house2"*></entry>

<entry key=*"shenzhen"* value-ref=*"house3"*></entry>

</util:map>

<!-- 往人里面注入常量name list<car> map<house> 和一个dog对象 -->

<bean id= *"preson"* class = *"spring\_beans.person"* >

<property name=*"name"* value=*"zzx"*></property>

<property name=*"cars"* ref=*"cars"*></property>

<property name=*"houses"* ref=*"houses"*></property>

<property name=*"dog"* ref=*"dog"*></property>

</bean>

1. **autowire自动装配**

<bean id= *"preson"* class = *"spring\_beans\_autowire.person"* autowire=*"byName"* >

<property name=*"name"* value=*"zzx"*></property>

</bean>

自动装配有两种

1autowire="byName"

按照名字来装配，在xml文件里面找有没有与spring\_beans\_autowire.person类里面有相同名字的属性的bean，如果有就调用set方法自动装配

2autowire="byType"(默认的注入行为，也可不写)

按照属性来装配，在xml文件里面找有没有与spring\_beans\_autowire.person类里面有相同类型的属性的bean，如果有就调用set方法自动装配

但是如果有两个相同类型的bean会导致自动装配失败

（详细见Eclipse--》spring\_shangguigu\_1\_helloword--》spring\_beans\_autowire）

**11、bean之间的继承关系（**parent=*""***）**

11-1一个bean可以继承另一个bean的属性，也可以重写实现对父bean的覆盖

<bean id= *"baoma"* class = *"spring\_beans\_继承与依赖.car"*>

<property name=*"price"* value=*"1000000"*></property>

<property name=*"maxspeed"* value = *"250"*></property>

<property name=*"color"* value=*"green"*></property>

</bean>

<!-- 继承了宝马并且重写了他的color属性 -->

<bean id= *"luhu"* parent=*"baoma"*>

<property name=*"color"* value=*"red"*></property>

</bean>

11-2 抽象bean（abstract="true"）只能继承，不能实例化

如果一个bean没有指定他的类，则这个bean为抽象bean（也要写abstract="true"），或者可以通过 abstract="true"来指定这个bean

是抽象bean抽象bean不能实例化只能被继承

<bean id= *"aodi"* abstract=*"true"*>

<property name=*"price"* value=*"2000000"*></property>

<property name=*"maxspeed"* value = *"350"*></property>

<property name=*"color"* value=*"red"*></property>

</bean>

<bean id= *"falali"* class = *"spring\_beans\_继承与依赖.car"* parent=*"aodi"*>

</bean>

1. bean的依赖关系（depends-on=*""）*

用depends-on=""来指定bean的依赖关系，表示当前bean实例化之前spring容器里面必须有被依赖的bean否则会报错

<bean id=*"dog"* class=*"spring\_beans\_继承与依赖.dog"*>

<property name=*"high"* value=*" 100"*></property>

<property name=*"color"* value=*" orange"*></property>

</bean>

<bean id = *"person1"* class=*"spring\_beans\_继承与依赖.person"* depends-on=*"dog"*>

<property name=*"dog"* ref=*"dog"*></property>

<property name=*"name"* value=*"jack"*></property>

</bean>

//person1初始化之前必须有dog，否则报错，相当一指定了他的dog不能为null

**12、Bean的作用范围：scope**

<bean id=*"userService"* class=*"com.bjsxt.service.UserService"* scope=*"prototype"*>

**scope = singleton 单例模式，每次都用同一个对象，这个对象在spring初始化的时候**

**已经创建**，这中单例模型可以节省内存，但是webwork的Action不是线程安全的，要求在多线程环境下必须是一个线程对应一个独立的实例，不能使用singleton。所以，我们在Spring配置Webwork Action Bean时，需要加上属性scope=”prototype”或singleton=”false”。

       singleton模式指的是对某个对象的完全共享，包括代码空间和数据空间，说白了，如果一个类是singleton的，假如这个类有成员变量，那么这个成员变量的值是各个线程共享的（有点类似于static的样子了），当线程A往给变量赋了一个值以后，线程B就能读出这个值。因此，对于前台Action，肯定不能使用singleton的模式，必须是一个线程请求对应一个独立的实例。推而广之，只要是带数据成员变量的类，为了防止多个线程混用数据，就不能使用singleton

1. Scope = prototype **原型模式，每次调用的时候都创建一个新的对象,在这种模式下，在容器初始化的时候不创键bean的实例，而是在每次初始化的时候创建一个新的bean**
2. Scope = session
3. Scope = application
4. Scope = request

**13、外部属性文件式连接数据库**

将数据库的属性字段写在另一个文件里，与xml文件分离，方便以后去做修改

引入c3p0和jdbcmysql

13-1编写属性文件db.properties

jdbc.mysql.driverclass=com.mysql.jdbc.Driver

jdbc.mysql.url=jdbc:mysql://localhost:3306/spring

jdbc.mysql.user=root

jdbc.mysql.password=123456

13-2在xml导入属性文件

<context:property-placeholder location=*"classpath:db.properties"*/>

13-3：连接数据库，并且获取DAtaSource的bean

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

<property name=*"driverClass"* value=*"${jdbc.mysql.driverclass}"* />

<property name=*"jdbcUrl"* value=*"${jdbc.mysql.url}"* />

<property name=*"user"* value=*"${jdbc.mysql.user}"* />

<property name=*"password"* value=*"${jdbc.mysql.password}"* />

</bean>

**14使用spel完成一些bean的赋值在#{}里面的就可以解析java代码**

Spel是IOC里的一项语言，通过这个spel语言可以实现对bena的属性的赋值，他的语法结构是 value = “#{}”,在{}中写上值

整数赋值：

<property name="count" value="#{5}"/>

字符串：

<property name=“name” value="#{'Chuck'}"/>

对象：

<property name=“name” value="#{house1}"/>

调用其他对象的属性来赋值：

<property name=*"address"* value=*"{house1.area}"*></property>

调用静态方法的赋值（这里调用math的PI计算轮胎周长）

<property name=*"luntai"* value=*"#{T(java.lang.Math).PI\*80}"*></property>

调用方法：

<property name=*"address"* value=*"{house1.toString()}"*></property>

在里面使用各种运算符（如多目运算）

<property name=*"info"* value=*"#{baoma.price*>300000? '金领':'白领'}"></property>

1. **bean的生命周期和bean的后置处理器**

**从bean被创建到销毁，叫做bean的生命周期**

15-1：init—method 和destory-method

<bean id = *"car"* class=*"spring\_postProcess.car"*

init-method=*"init"* destroy-method=*"destory"*>

Init-method：

在通过构造器或工厂方法创建 Bean 实例，和**完成属性赋值后**调用init-method所指定方法

Destory-method

在容器销毁（这个bean被销毁）的时候执行他所指定的方法

15-2BeanPostProcessor 接口

这个接口的实现类，可以影响每一个bean的声明周期，利用动态代理，可作为bean的拦截器，过滤器，重要须知道

首先创建BeanPostProcessor 接口的实现类，在里面做出你想要的修改

**实例：**

**public** **class** mybeanProcessor **implements** BeanPostProcessor {

**public** Object postProcessBeforeInitialization(Object bean, String beanName)

**throws** BeansException {

System.***out***.println("postProcessBefore"+bean +beanName);

**return** bean;

}

**public** Object postProcessAfterInitialization(Object bean, String beanName)

**throws** BeansException {

System.***out***.println("postProcessAfter"+bean +beanName);

**return** bean;

}

}

//第一个参数是xml里面的其中一个bean（谁执行就是谁），第二个参数是所对应的id

这个类相当于一个工具，可以被调用

然后:在xml文件里面创建这个接口实现类的bean，那么，spring容就会让每个bean创建时都执行它，这个bean对所有的Bean都会起作用，因此无需设置id

<bean class=*"spring\_postProcess.mybeanProcessor"*></bean>

15-3如果在同时存在init-thread和destory-method和BeanPostProcessor的接口实现类，那么他们的执行顺序如下

1. 通过配置类或者xml文件读取类信息反射创建bean
2. 通过beanNameAware BeanfoctoryAwary BeanapplicationAwary让bean可以知道自己的名字，工厂，容器等信息

3、将 Bean 实例传递给 Bean 前置处理器的 postProcessBeforeInitialization 方法

4、调用 Bean 的初始化方法method-init

5、将 Bean 实例传递给 Bean 后置处理器的 postProcessAfterInitialization方法

6、Bean 可以使用了

7、当容器关闭时, 调用 Bean 的销毁方法destory

15-4：BeanPostProcessor的接口实现类的权力相当大，可以将传过来的spring容器生成的bean做出修改，也可以直接创建一个新的bean然后返回，实现偷梁换柱

**16创建bean的方法**

16-1spring容器内部反射（常用）

16-2静态工厂方法

16-3实例工厂方法

（见12尚硅谷）

16-4：用spring提供的factory<T>接口来生成bean

1. 创建一个类实现spring提供的factory<T>接口，这个实现类就像一个工厂,可以创建出对象，并且返回

Exp：

**public** **class** carfactory **implements** FactoryBean<car> {

//返回bean对象,xml就是通过它来获得对象

**public** car getObject() **throws** Exception {

**return** **new** car("bmw", 300000);

}

//返回bean类型

**public** Class<?> getObjectType() {

**return** car.**class**;

}

//返回是不是单例

**public** **boolean** isSingleton() {

**return** **true**;

}

}

1. 在xml里面创建carfactory的对象，得到的就是getObject()方法返回的对象，因而从工厂得到了bean

<bean id=*"car"* class=*"spring\_factorybaen.carfactory"* > </bean>

**17、classpath 扫描自动加载出类的对象，完成注入和装配**

基本的四个注解，四个注解效果一样，只是约定了某层用什么

@Component: 基本注解, 标识了一个受 Spring 管理的组件

@Respository（仓库：储藏室）: 标识持久层组件 dao层接口和实现类

@Service: 标识服务层(业务层)组件

@Controller: 标识表现层组件 控制层主要实现他只是用来处理简单的参数

给springmvc使用

验证啊、或者简单查询什么的，所有业务逻辑不要写在controller里面

1. 在xml文件里面设置（component-scan）为自动加载并且创建注解所对应的类的bean

利用包的名字初始化所有子包的类，即装配所有以spring\_classpath\_scran.开头的类下面的有特定注解bean

<context:component-scan base-package="spring\_classpath\_scran"/>

2、 resource-pattern加载指定的包

<context:component-scan base-package="spring\_classpath\_scran" resource-pattern="Service/\*.class"></context:component-scan>

即装配除了以spring\_classpath\_scran.Service的类以外的所有以spring\_classpath\_scran.开头的类下面的有特定注解bean，如果正使用注解的时候没有指定名字，那么，他会按类名第一个字母小写，生成对应的bean

如UserSever生成bean名字为userSever，但是如果类名为按规定起，本身就是usersever，那么他生成的bean的名字是usersever。

1. 使用context:exclude-filter子组件来排除某个注解，和扫描哪些注解

如以后让spring只扫描除@controller以外注解的类，让springmvc值扫描

Controller的注解，可以这样做

//让spring值管理处@controller以外的注解的类

<context:component-scan base-package=*"zzx"*>

<context:exclude-filter type=*"annotation"* expression=*"org.springframework.stereotype.Controller"*/>

</context:component-scan>

1. 用 @Autowired （默认按属性装配）和 @Resource（默认按名字装配） 、@Inject完成自动装配，形成bean之间的相互引用，这三个注解的效果一样，所以我们一般使用@Autowire就可以
2. @Autowired可以放在**字段上，构造方法，引用类型，或者set方法上**完成对应的自动装配

如：放在一个引用上面，也可以完成装配

@Autowired

**private** userController controller;

1. **@Autowired在自动装配的时候是按照属性来装配的，也就是按类型装配，在出现多个相同属性的时候，可能会出现无法装配，此时可以通过@qualiter来指定装配哪一个**

**如useRespository有两个实现类的对象在spring容器里面，然后用@qualiter来指定装配名字为respositoryimp的bean**

**@Autowired**

**@Qualifier("respositoryimp")**

**private useRespository useRespository;**

1. @autowire标注的时候，如果spring容器里面没有这个类的对象，会报错，如果要指定这个对象可以为空可以这样写

**@Autowired（require=false）**

**private useRespository useRespository;**

1. @autowire放在数组上面

他会将容器里面所有匹配类型的bean装配进来

@Autowired

savecar[] cars;

1. 放在集合上面（List，非arraylist等实现类，否则会报错）

Spring会读取集合信息，把适合的bean都装配进来

@Autowired

List<savecar> carList;

1. 放在map上面（map，非hashmap或其他实现类，否则报错）

Spring会将适合的bean装配进来，以bean的名字作为key，引用作为value

@Autowired

Map<String, savecarimpl> carmap;

**18泛型类、泛型接口等**

**public** **class** BaseRepository<T> {}

泛型类:

1、在这个类里面可以像普通的类一样操定义各个类型的函数、属性、对象

2、T是一个任意类型的东西，比如String、Integer、或则某一个类如User

他的参数就是一个类型，或者说是一个类。

1. 在泛型类里面可以操作T类型的东西，如定义T类型的一个对象，创造方法 等各种操作

如T t，表示定义一个T类型的对象t，如果T为一个类User，那么t就是User的一个对象。

1. 泛型类在实例化，或者继承他的类必须指明T是什么类型
2. 泛型是对参数类型的增强，同时也能消除许多强制类型转换，因为一开始T

的类型没有指定，而是在他的对象或者实现类或者子类里面指定，灵活性更高，比如突然想把本来写的int改为doule，只需在实现类里面重新指定一下T就好了

1. **泛型的依赖注入完成自动装配（spring\_3\_fanxing）**

泛型就像一个模板，因为泛型的类型没有指定，所以他就像一个接口，他的功能函数都是围绕泛型来定义，在泛型的类型没有指明之前，他的函数不可使用

所以继承他的类必须指明泛型的类型，然后就可以使用父类（泛型类定义的方法）

在泛型类里面使用@autowire的时候，会被子类继承，并且注入的将会是他对应的子类的对象，因为子类的对象才是可用的

泛型是对参数类型的增强，同时也能消除许多强制类型转换，因为一开始T

的类型没有指定，而是在他的对象或者实现类或者子类里面指定，灵活性更高，比如突然想把本来写的int改为doule，只需在实现类里面重新指定一下T就好了

**20(补)、动态代理**

动态代理是aop的底层原理，过程如下

1. 编写接口以及实现类
2. 利用接口和实现类来实现动态代理
3. proxy为代理对象，
4. 利用Proxy.newProxyInstance(loader, interfaces, h)，来产生代理 对象，第一个参数为指定类加载器，第二个为接口方法，第三个参数为InvocationHandler接口的实现类的对象。
5. InvocationHandler方法里面有个invoke方法，实现对对象的动态代理，实现前置方法，后置方法等，利用method.invoke(target, args)来实现对元方法的调用，在打debug的时候再这里进入方法可以进入到原来的类和方法
6. **AOP面向切面的编程**

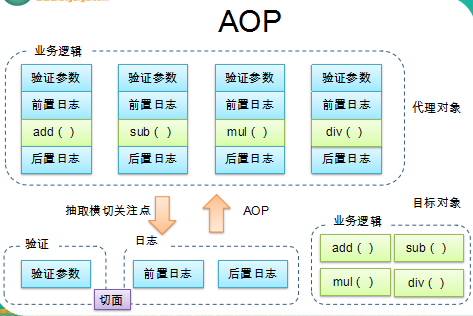
我们在编程的时候经常需要加上日志，需要验证参数是否合法（拦截器）等非业务处理的代码，导致我们的代码混乱，代码分散维护性低，aop将采用与动态代理（动态代理见代码）相似的原理，将非业务处理的代码从原来的类里面抽出来，写在一个单独的类里面，这样做就完美地解决了上面的代码混乱，代码分散，代码重复，可维护性低的问题，aop是对面向对象编程的一个补充

在应用 AOP 编程时, 仍然需要定义公共功能, 但可以明确的定义这个功能在哪里, 以什么方式应用, 并且不必修改受影响的类. 这样一来横切关注点就被模块化到特殊的对象(切面)里.

Aop的好处

每个事物的逻辑位于一个位置，代码不分散，便于维护和升级

业务模块更简洁，只包含核心代码



Aop中的一些术语

切面：横切关注点(跨越应用程序多个模块的功能) 是一个实现通知的类

如：上面有验证切面、日志切面

通知(Advice): 切面必须要完成的工作（方法）

目标(Target): 被通知的对象

代理(Proxy): 向目标对象应用通知之后创建的对象

连接点（Joinpoint）：程序执行的某个特定位置，是一个物理点

如：某个方法调用前，异常出现后

切点（pointcut）：每个类都拥有多个连接点：例如 ArithmethicCalculator 的所有方法实际上都是连接点，即连接点是程序类中**客观存在的事务**。AOP 通过切点定位到特定的连接点。类比：连接点相当于数据库中的记录，切点相当于查询条件。切点和连接点不是一对一的关系，一个切点匹配多个连接点，

1. **AspectJ ：Spring里面用来实现面向切面编程的一个框架，可以用注解或者xml文件配置来实现面向切面的编程**

**1、采用注解的方式实现aop**

1、在xml文件里加入aop自动注解的标签（详细见eclipse里尚硅谷9）

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

2、建立切面类将他用@compenemt注入容器，然后用@Aspect将他设置为切面类

3、利用@before@after等通知注解表明什么时候执行，和哪些方法需要执行

**2、spring里面存在的通知注解有哪些**

@Before: 前置通知, 在目标方法执行之前执行

@After: 后置通知, 在目标方法执行之后执行（无论是否异常都执行）

但是在后置通知里面不允许访问目标方法的结果，相当于finally里面

@AfterRunning: 返回通知, 在方法返回结果之后执行

只有在方法正常执行之后才会执行

而且可以访问原方法的返回值

@AfterThrowing: 异常通知, 在方法抛出异常之后

在方法出现异常的时候才执行，并且可以访问到异常

对象，并且还可以指定出现什么异常才执行

@Around: 环绕通知, 围绕着方法执行

就像动态代理可以同时加上前置后置返回异常通知，功能强大

但是不常用

1. **指定切面的优先级（@order）**

可以在切面类上面加上@order（int）来指定切面的优先级，而int越小，切面的优先级越高，优先级高的切面先于优先级低的切面执行

1. **将切点封装成方法，方便调用和修改**

1、将切入点字符串封装成方法，declarePoincut()的返回值即使切入点字符串

@Pointcut("execution(public \* spring\_shangguigu\_p\_aspectj.caculateimp.\*(..))")

**public** **void** declarePoincut(){}

2、调用方法declarePoincut()去替代

execution(public \* spring\_shangguigu\_p\_aspectj.caculateimp.\*(..))

@Before("declarePoincut()")

1. **在xml文件里面配置aop**

**配置切面类的bean**

**配置切点**

**配置装配切面**

**见尚硅谷10**

1. **建立通通知类**
2. **将类是扫描进入spring**
3. **在xml文件里面配置aop**
4. **配置pointcut**
5. **配置切面，包括引用什么切面类的对象，通知类型**

**<context:component-scan base-package="aop\_byxml"></context:component-scan>**

**<aop:config proxy-target-class="true">**

**<aop:pointcut expression="execution(\* aop\_byxml.\*.\*(..))" id="pointcut"/>**

**<!-- 任意返回值（\*和aop间有空格），aop\_byxml包下的任意类的任意方法，（..）表示任意参数 -->**

**<aop:aspect ref="loggingAspect">**

**<aop:before method="beforeMethod" pointcut-ref="pointcut"/>**

**</aop:aspect>**

**<!-- 切面2-->**

**<aop:aspect ref="vlidationAspect">**

**<aop:before method="beforeMethod" pointcut-ref="pointcut"/>**

**</aop:aspect>**

**</aop:config>**

1. **使用jdbcTemplate操作数据库**

**JdbcTemplate 类被设计成为线程安全的, 所以可以再 IOC 容器中声明它的单个实例, 并将这个实例注入到所有的 DAO 实例中.**

1. **通过c3p0 得到数据库对象dataSource**
2. **用dataSource对象作为参数得到jdbcTemplate对象**
3. **使用jdbcTemplate操作数据库**

**（详细见spring4）**

**23具名参数（在操作数据库的时候让参数再用？占位，而是用数据库里的字段名占位，这样可以在参数中多的时候不混乱，便于维护）**

**1、通过c3p0 得到数据库对象dataSource**

**2、用dataSource对象作为参数得到namedParameterJdbcTemplate对象**

**3、使用namedParameterJdbcTemplate操作数据库**

**（详细见11）**

1. **事务**
2. **简介**

**事务就是一系列的动作, 它们被当做一个单独的工作单元. 这些动作要么全部完成, 要么全部不起作用**

**具有**

原子性：事务的所有动作为一个原子，不可分割，要么全部完成，要么全部不完成

一致性：一旦事务被提交，数据和资源处于满足业务规则的一致性状态

隔离性：可能多个事务处理相同的数据，这是候，数据应该相互隔离

持久性：事务完成后，不再受任何因素的影响

1. **spring中的事务**

Spring 既支持编程式事务管理, 也支持声明式的事务管理.

**编程式事务管理:** 将事务管理代码嵌入到业务方法中来控制事务的提交和回滚. 在编程式管理事务时, 必须在每个事务操作中包含额外的事务管理代码.

**声明式事务管理**: 大多数情况下比编程式事务管理更好用. 它将事务管理代码从业务方法中分离出来, 以声明的方式来实现事务管理. 事务管理作为一种横切关注点, 可以通过 AOP 方法模块化. Spring 通过 Spring AOP 框架支持声明式事务管理.

1. **spring利用注解使用声明式事务（创建、启用、添加）**

**分为创建事务的bean，启用事务，在合适的地方添加事务注解三部**

3-1：在xml文件里创建事务处理的bean

<!-- 配置事务的bean-->

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

<!--因为是管理数据的，所以需要数据源作为参数->

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

</bean>

3-2：启用

<!-- 启用事务注解-->

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

3-3：在需要事务的地方加上@transitional注解

@Transactional

**public** **void** testTransiton(){

Bookshopdao.findbookpricebyisbn("1001");

Bookshopdao.updatebookstock("1001");

Bookshopdao.updateUserAccount("aa",100);

}

1. 事务的传播行为

当一个事务里面嵌套了多个子事务时，事务改如何进行

如一个人买两本书，买书本身是事务，每买一本书也是一个事务

买书

买第二本书

买一本书

4-1用@transition（propagation = REQUEST）设置事务的传播行为

4-2propagation = REQUIRED

事务的默认行为，在嵌套事务里面，只要其中一个失败，撤销全部已经完成的子事务，回到第一个事务开始前，就像买书的事务覆盖了子事务，子事务不起作用

（如买两本书，第一本成功、第二本失败，那么事务会回滚两本书的事务，两本均购买失败）

4-3propagation = REQUIRES\_NEW

采用子事务覆盖母事务的方式，母事务失效

（如买两本书，第一本成功、第二本失败，那么第一本会交易成功，只回滚第一本书的事务）

4-4一些事务里的概念

脏读: 对于两个事物 T1, T2, T1 读取了已经被 T2 更新但 还没有被提交的字段. 之后, 若 T2 回滚, T1读取的内容就是临时且无效的.

不可重复读:对于两个事物 T1, T2, T1 读取了一个字段, 然后 T2 更新了该字段. 之后, T1再次读取同一个字段, 值就不同了.

幻读:对于两个事物 T1, T2, T1 从一个表中读取了一个字段, 然后 T2 在该表中插入了一些新的行. 之后, 如果 T1 再次读取同一个表, 就会多出几行.

4-5事务的隔离级别

理论上事务应该在较高的隔离级别上运行，那么就可以避免4-4的问题，但是为了高效运行，事务一般运作在较低的隔离级别，那么可以运行自己设置隔离级别（isolation=””）的方式来提高事务隔离级别（默认的为read\_comment，一般不用设置）

4-6设置事务超时时间

Timeout = 3

表示只能占用数据源三秒，三秒没成功将自动回滚

4-7设置只读

readOnly=true

表示事物对数据库进行读取而不进行更新，这样有利于优化数据库

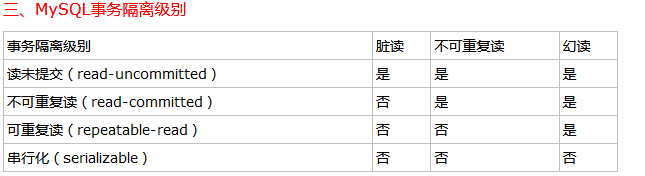
4-8事务的隔离级别

读可以提交

不可重复读

可重复读

序列化



**26在xml文件里面配置事物**

**1配置事物的bean**

**2配置需要事物的方法的属性**

**3配置切入点**

**4将方法的属性和切入点联系起来**

<!-- 配置事物处理的bean -->

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

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

</bean>

<!--配置事物的每个方法的属性 -->

<tx:advice id=*"txadv"* transaction-manager=*"dataSourceTransactionManager"*>

<tx:attributes>

//配置方法的属性

<tx:method name=*"sever"* propagation=*"REQUIRES\_NEW"*/>

<tx:method name=*"\*"*/>

</tx:attributes>

</tx:advice>

<!-- 配置事物的切入点 和将方法的属性装配进来-->

<aop:config>

<aop:pointcut expression=*"execution(\* spring\_shangguigu\_jdbc.transation.sever.\*(..))*

*"* id=*"txpoinCut"*/>

//配置切入点和每个方法的属性（属性在txdiv里为每个方法配置好了）

<aop:advisor advice-ref=*"txadv"* pointcut-ref=*"txpoinCut"*/>

</aop:config>

**27Spring在web上的应用**

**1). 需要额外加入的 jar 包:**

**spring-web-4.0.0.RELEASE.jar**

**spring-webmvc-4.0.0.RELEASE.jar**

**2). Spring 的配置文件, 没有什么不同**

1. **. 如何创建 IOC 容器 ?**

**在非web项目的时候可以在main方法直接手动创建**

**在web项目时则在服务器加载（servletContext创建）的时候，在载入 ServletContextListener#contextInitialized(ServletContextEvent sce) 方法的时候创建 IOC 容器.，然后将创建出来的ioc容器通过serAttribute方法放入servletContext（application）里面，然后需要的时候则直接从里面取出**

**首先在servletContext里面创建ioc的配置方法如下：**

**<!-- 配置 Spring 配置文件的名称和位置 -->**

**<context-param>**

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

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

**</context-param>**

**<!-- 启动 IOC 容器的 ServletContextListener -->**

**<listener>**

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

**</listener>**

1. **在ioc装入servletcontext后在web中得到ioc容器**

**4-1在jsp中得到**

<%

//直接在application里面拿到ioc容器 ctx

ApplicationContext ctx =WebApplicationContextUtils.getRequiredWebApplicationContext(application);

//取得bean

preson preson =(preson) ctx.getBean(preson.class);

//调用方法

preson.hello();

%>

**4-2在servlet中得到ioc容器**

//1从ServletContext中拿到ioc容器对象

ServletContext application = **this**.getServletContext();

//2从ServletContext中拿到ioc容器对象

WebApplicationContext wac = WebApplicationContextUtils.*getRequiredWebApplicationContext*(application);

重新开始

1. spring容器初始化的过程会调用空的构造器通过反射创建对象，调用set方法进行注入,如果没有指定创建的bean全部都是单例的

2、内部bean只能在bean的内部使用，不能再外部调用