# --spring优点：

1 IOC/DI 方便程序解耦和 Inversion of Control / Dependency Injection

2 AOP 方便程序的扩展 Aspect Oriented Programming

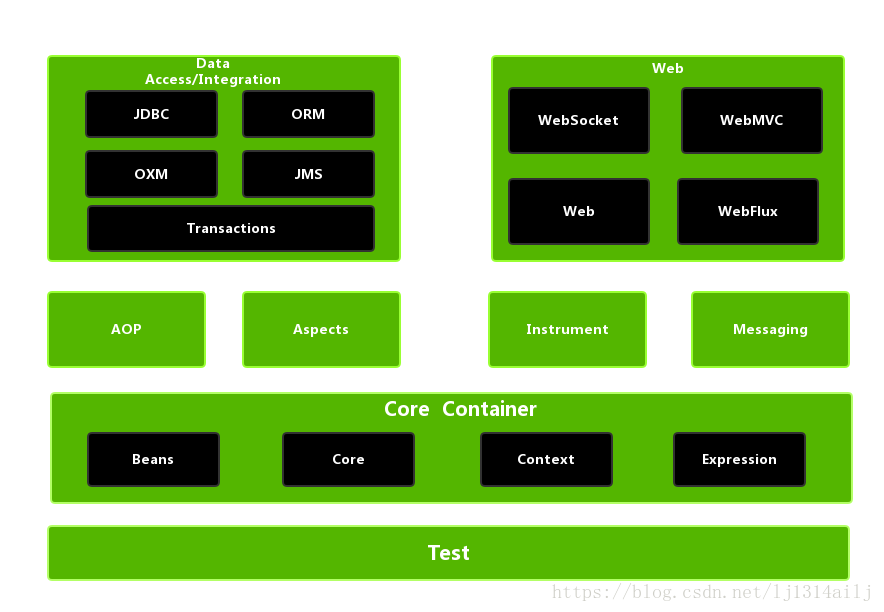
3 方便整合其他框架

4 轻量级，代码入侵低。

5 一站式框架：web层/service层/dao层

# Spring5 系统架构

**一，系统架构**

  Spring 总共大约有 20 个模块，由 1300 多个不同的文件构成。而这些组件被分别整合在核心容器 （Core Container）、AOP（Aspect Oriented Programming）和设备支持（Instrmentation）、 数据访问及集成（Data Access/Integeration）、Web、报文发送（Messaging）、Test，6 个模块 集合中。以下是 Spring 5 的模块结构图：   
   
组成 Spring 框架的每个模块集合或者模块都可以单独存在，也可以一个或多个模块联合实现。每个模 块的组成和功能如下：

**1.1 核心容器：由 spring-beans、spring-core、spring-context 和 spring-expression（Spring Expression Language, SpEL） 4 个模块组成。**

  spring-beans 和 spring-core 模块是 Spring 框架的核心模块，包含了控制反转（Inversion of Control, IOC）和依赖注入（Dependency Injection, DI）。

BeanFactory 接口是 Spring 框架中 的核心接口，它是工厂模式的具体实现。BeanFactory 使用控制反转将应用程序的配置和依赖性规范与 实际的应用程序代码进行了分离。但 BeanFactory 容器实例化后并不会自动实例化 Bean，只有当 Bean 被使用时 BeanFactory 容器才会对该 Bean 进行实例化与依赖关系的装配。

  spring-context 模块构架于核心模块之上，他扩展了 BeanFactory，为她添加了 Bean 生命周期控制、框架事件体系以及资源加载透明化等功能。此外该模块还提供了许多企业级支持，如邮件访问、 远程访问、任务调度等，

ApplicationContext 是该模块的核心接口，她是 BeanFactory 的超类，与   
BeanFactory 不同，ApplicationContext 容器实例化后会自动对所有的单实例 Bean 进行实例化与依赖关系的装配，使之处于待用状态。

  spring-expression 模块是统一表达式语言（EL）的扩展模块，可以查询、管理运行中的对象， 同时也方便的可以调用对象方法、操作数组、集合等。

它的语法类似于传统 EL，但提供了额外的功能， 最出色的要数函数调用和简单字符串的模板函数。这种语言的特性是基于 Spring 产品的需求而设计， 他可以非常方便地同 Spring IOC 进行交互。

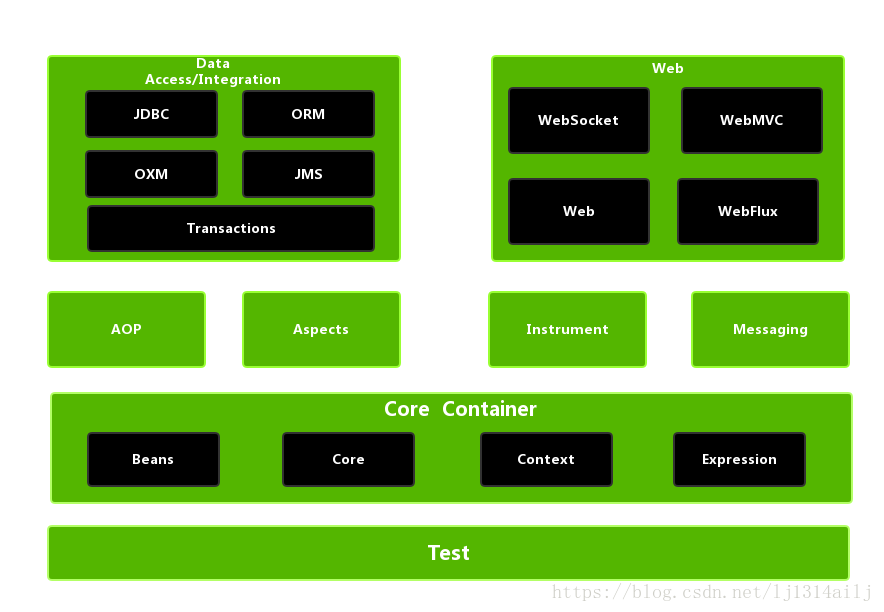
**1.2 AOP 和设备支持：由 spring-aop、spring-aspects 和 spring-instrument 3 个模块组成。**

  spring-aop 是 Spring 的另一个核心模块，是 AOP 主要的实现模块。作为继 OOP 后，对程序员影响最大的编程思想之一，AOP 极大地开拓了人们对于编程的思路。

在 Spring 中，他是以 JVM 的动态代理技术为基础，设计出了一系列的 AOP 横切实现，比如前置通知、返回通知、异常通知等。同时，使用 Pointcut 接口来匹配切入点，可以使用现有的切入点来设计横切面，也可以扩展相关方法根据需求进行切入。

  spring-aspects 模块集成自 AspectJ 框架，主要是为 Spring AOP 提供多种 AOP 实现方法。   
spring-instrument 模块是基于 JAVA SE 中的”java.lang.instrument”进行设计的，应该算是 AOP 的一个支援模块，主要作用是在 JVM 启用时，生成一个代理类，程序员通过代理类在运行时修改类的字节，从而改变一个类的功能，实现 AOP 的功能。

在分类里，我把他分在了 AOP 模块下，在 Spring 官方文档里对这个地方也有点含糊不清,这里是纯个人观点。



**1.3 数据访问及集成：由spring-jdbc、spring-tx、spring-orm、spring-jms 和 spring-oxm 5 个模块组成。**

  spring-jdbc 模块是 Spring提供的 JDBC 抽象框架的主要实现模块，用于简化JDBC编程。

主要是以提供 JDBC 模板方式、关系数据库对象化方式、SimpleJdbc 方式、事务管理来简化 JDBC 编程，主要实现类是JdbcTemplate，SimpleJdbcTemplate 以及NamedParameterJdbcTemplate。

  spring-tx 模块是 Spring JDBC 事务控制实现模块。使用 Spring 框架，它对事务做了很好的封装， 通过它的 AOP 配置，可以灵活的配置在任何一层；

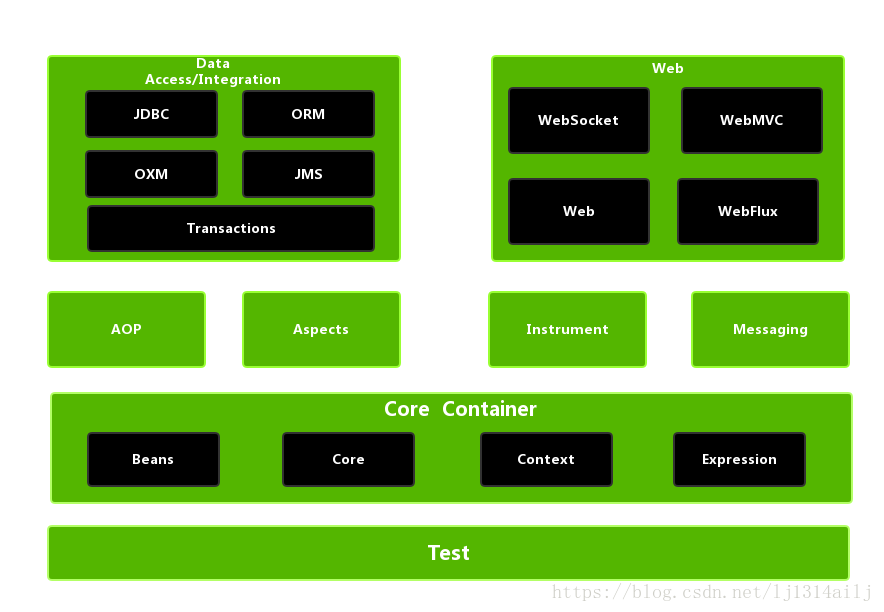
但是在很多的需求和应用，直接使用 JDBC 事务控制 还是有其优势的。其实，事务是以业务逻辑为基础的；一个完整的业务应该对应业务层里的一个方法； 如果业务操作失败，则整个事务回滚；

所以，事务控制是绝对应该放在业务层的；但是，持久层的设计 则应该遵循一个很重要的原则：保证操作的原子性，即持久层里的每个方法都应该是不可以分割的。所以，在使用 Spring JDBC 事务控制时，应该注意其特殊性。

  spring-orm 模块是 ORM 框架支持模块，主要集成 Hibernate, Java Persistence API (JPA) 和Java Data Objects (JDO) ，用于资源管理、数据访问对象(DAO)的实现和事务策略。

  spring-jms 模块（Java Messaging Service）能够发送和接受信息，自 Spring Framework 4.1 以后，他还提供了对 spring-messaging 模块的支撑。

  spring-oxm 模块主要提供一个抽象层以支撑 OXM（OXM 是 Object-to-XML-Mapping 的缩写，它是一个 O/M-mapper，将 java 对象映射成 XML 数据，或者将 XML 数据映射成 java 对象），例如：JAXB,Castor, XMLBeans, JiBX 和 XStream 等。



**1.4 Web：由 spring-web、spring-webmvc、spring-websocket 和 spring-webflux 4 个模块组成.**

  spring-web 模块为 Spring 提供了最基础 Web 支持，主要建立于核心容器之上，通过 Servlet 或者 Listeners 来初始化 IOC 容器，也包含一些与 Web 相关的支持。

  spring-webmvc 模 块 众 所 周 知 是 一 个 的 Web-Servlet 模 块 ， 实 现 了 Spring MVC （model-view-Controller）的 Web 应用。

  spring-websocket 模块主要是与 Web 前端的全双工通讯的协议。（资料缺乏，这是个人理解）

  spring-webflux 是一个新的非堵塞函数式 Reactive Web 框架，可以用来建立异步的，非阻塞，事件驱动的服务，并且扩展性非常好。

**1.5 报文发送：即 spring-messaging 模块。**

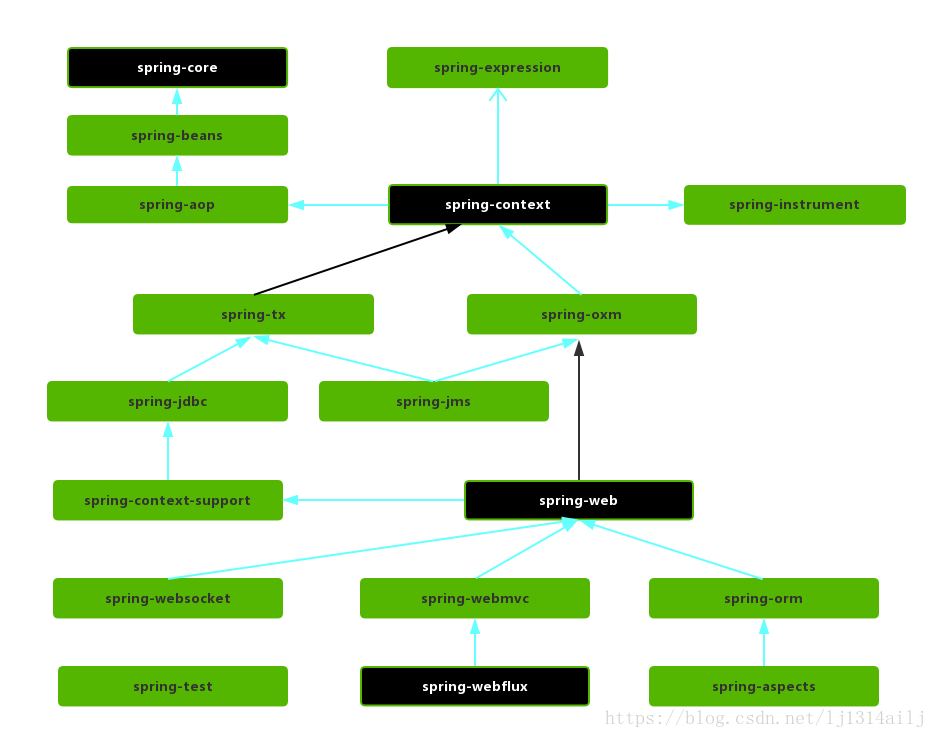
  spring-messaging 是从 Spring4 开始新加入的一个模块，主要职责是为 Spring 框架集成一些基础的报文传送应用。

**1.6 Test：即 spring-test 模块。**

  spring-test 模块主要为测试提供支持的，毕竟在不需要发布（程序）到你的应用服务器或者连接 到其他企业设施的情况下能够执行一些集成测试或者其他测试对于任何企业都是非常重要的。

**1.7 Spirng 各模块之间的依赖关系**

  该图是 Spring5 的包结构，可以从中清楚看出 Spring 各个模块之间的依赖关系。

   
如果想学习 Spring 源码的学习，建议是从 spring-core 入手，其次是 spring-beans 和   
spring-aop，随后是 spring-context，再其次是 spring-tx 和 spring-orm，最后是 spring-web和其他部分

# 1 重要的高级思想：IOC/DI AOP

# 2 Spring是一套SE/EE的一站式框架。

一站式框架：

WEB层：SpringMVC

Service层：Spring管理Beans/Spring管理事务

Dao层：Spring中的JDBCTemplate（JDBC模板），和Spring中的ORMM模块（整 合其他ORM框架）

# 第一大部分---IOC/DI

# 3 Spring中beans管理：

## 3.1使用控制反转，

将新建对象的权限交给（反转给）Spring容器。让Spring去管理对象的创建、自动装配和销毁。

好处：方便程序的解耦合。

### 3.1.1 传统方式，

需要我们自己在类中去创建需要依赖的对象。如：

public class UserDaoImpl implements UserDao{

private JDBCTemplate template = new JDBCTemplate();

}

这种方式，当以后需要实现一个MyBatis的dao操作时，就需要到源码中去直接修改创建依赖对象。当项目庞大时，非常的不易于维护。

### 3.1.2 Spring管理Beans

我们不去创建对象，但是在Spring配置文件中配置需要创建对象的描述，以及哪些服务需要哪些依赖对象。这样，让Spring容器去管理对象的创建、装配、销毁。

public class UserDaoImpl implements UserDao{

@Autowired

private JDBCTemplate template； }

}  
这种方式，并没有在源码中将依赖对象new出写死，而是由Spring根据配置文件，在运行时创建、装配管理。所以，后期修改维护时，只需要修改该配置文件即可。

## 3.2 Spring怎么根据配置文件创建Beans？（IOC底层实现，工厂+反射+配置文件）

### 3.2.1 了解IOC之前，先了解一下平时我们都怎么获取一个对象。

开闭原则（OCP，Open Close Principle）是面向对象设计中“可复用设计”的基石，是面向对象设计中最重要的原则之一，其它很多的设计原则都是实现开闭原则的一种手段。

遵循开闭原则设计出的模块具有两个主要特征：

（1）对于扩展是开放的（Open for extension）。这意味着模块的行为是可以扩展的。当应用的需求改变时，我们可以对模块进行扩展，使其具有满足那些改变的新行为。也就是说，我们可以改变模块的功能。

（2）对于修改是关闭的（Closed for modification）。对模块行为进行扩展时，不必改动模块的源代码或者二进制代码。模块的二进制可执行版本，无论是可链接的库、DLL或者.EXE文件，都无需改动。

最基本的方法：直接new出对象

UserDao dao = new UserDao( );

UserDao dao = new UserDaoImpl( ); new UserMybatisImpl();

面向接口的方法：可以实现多态，父类引用指向子类对象

一个好的程序要实现OCP设计原则：就是在尽量不修改源代码的基础上，实现对程序的扩展。

由此：工厂模式诞生了

接口和实现类有耦合（联系过紧）

切换底层需求时，需要修改源代码

接口：UserDao

实现类：UserDaoImpl

工厂：BeanFactory

class BeanFactory{

public static UserDao getBean(){

return new UserDaoImpl();

}

public static UserMybatis getBean(){

return new UserMyBatisDaoImpl();

}

}

这种方法解决了接口和实现类的耦合，但是由造成了接口和工厂直接的耦合。

当底层需要切换时，虽然不需要去大量的修改源代码，但是仍需要在源代码种对一个地方修改：工厂。

如何：能够做到，真正的切换底层而不动源代码呢？

由此：IOC诞生了—工厂+反射+配置文件

new MyHibernaeDaoImpl();

IOC底层实现原理：工厂+反射+配置文件（beans.xml）或注解 实现程序解耦和

// 1 BeanFactory工厂

class BeanFactory{

public static UserDao getBean(String id){

//2 解析XML配置文件，根据id获取类名程

…

String className = 解析方法（id）；

//3 反射获取实例

Class clazz = Class.forName( className);

UserDao dao = clazz.newInstance();

return dao;

}

}

<bean id=”userDao” name=”userDao” class=”cn.htb.UserDaoImpl” ><bean/>

这样以来，当底层需求切换时，就不再需要改动源代码，只需要对配置文件进行修改即可。

而且，这一系列操作Spring的底层已经实现了，也就是说我们不再自己创建对象，只需要配置文件即可，创建对象的事由Spring去做，这也就是控制反转思想的实现。

另外，在结合其依赖注入思想的实现就能完全实现对beans的管理。

## 3.3如何把创建的对象放置到需要的位置？🡪依赖注入？

### 3.3.1知识扩展：OOP种类（对象）的关系

1）依赖

class A{ }

class B{

public void doit(A a){ }

}

2）继承

class A{ }

class B extends A{ }

3）聚合：松散聚合 紧密聚合

### 3.3.2 依赖注入

IOC 控制反转；将对象的创建权反转给了Spring

DI 依赖注入；前提必须有IOC环境，Spring管理这个类的时候，将这个类依赖的属性注入（设置）进来。

如：属性注入方法，在类中要有依赖的set方法。

当spring通过IOC创建对象时，在配置文件中发现其有依赖的配置，如下：

<property name=”userDao” ref=”userDao”/>

<property name=”name” ref=”张三/>

则会在通过IOC创建成功后，在通过反射调用该对象依赖属性的set方法，对其进行设置。

## 3.4 了解了控制反转和依赖注入，那么是谁来操纵这一切的呢？

### 3.4.1 spring中的bean工厂

#### 3.4.1.1 BeanFactory

老版本的工厂类接口。

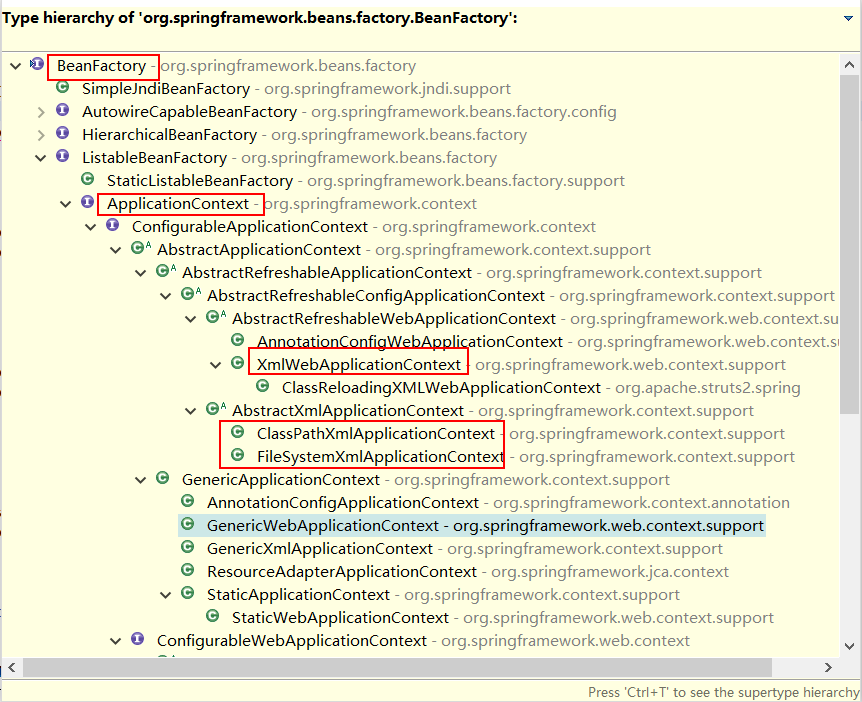
#### 3.4.1.2 ApplicationContext

新版本的工厂类接口，继承了BeanFactory，有着比BeanFactory更强大的功能（信息文件解析/支持国际化I18N/还有什么通知）。

两个常用实现类：

ClassPathXmlApplicationContext:加载类路径下的配置文件（src下，编译后WEB-INF下）

FileSystemXmlApplicationContext:加载文件系统下的配置文件（也就是磁盘上的配置文件）



#### 3.4.1.3 两个版本的工厂接口区别：

最大区别：BeanFactory是在调用getBean()的时候，才会生成相应的实例。

ApplictionContext是在加载配置文件时，就生成了文件中对应的所有实例（单例模式下）。

ApplictionContext applictionContext =

new ClassPathXmlApplictionContext(“applicationContext.xml”)

UserDao userDao = (UserDao) applictionContext.getBean(“userDao”);

## 3.5 Spring中bean的生命周期

四大阶段：实例化阶段🡪初始化阶段🡪被使用阶段🡪销毁阶段

Spring IoC容器允许BeanFactoryPostProcessor（后置处理程序）在容器实例化任何bean之前读取bean的定义(配置元数据)，并可以修改它。同时可以定义多个BeanFactoryPostProcessor，通过设置'order'属性来确定各个BeanFactoryPostProcessor执行顺序。

注册一个BeanFactoryPostProcessor实例需要定义一个Java类来实现BeanFactoryPostProcessor接口，并重写该接口的postProcessorBeanFactory方法。通过beanFactory可以获取bean的定义信息，并可以修改bean的定义信息。这点是和BeanPostProcessor最大区别。

@FunctionalInterface  
public interface BeanFactoryPostProcessor {  
  
 */\*\*  
 \* Modify the application context's internal bean factory after its standard  
 \* initialization. All bean definitions will have been loaded, but no beans  
 \* will have been instantiated yet. This allows for overriding or adding  
 \* properties even to eager-initializing beans.  
 \** ***@param*** *beanFactory the bean factory used by the application context  
 \** ***@throws*** *org.springframework.beans.BeansException in case of errors  
 \*/* void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;  
  
}

生命周期中主要涉及的类：

实例化：

BeanFactoryPostProcessor bean工厂后置处理器->postProcessBeanFactory(beanFactory)

获取并修改bean的定义信息

BeanPostProcessor bean后置处理器

InstantiationAwareBeanPostProcessor ->postProcessBeforeInstantiation()

构造方法实例化

->postProcessPropertyValue()

BeanNameAware

BeanFactoryAware

ApplicationContextAware

初始化：

BeanPostProcessor ->postProcessBeforeInitialization ()

init-method

InitializingBean ->afterPropertiesSet()

BeanPostProcessor ->postProcessAfterInitialization ()

销毁：

 DiposiableBean -destory()

destroy-method

Spring容器加载配置文件调用构造器进行实例化（BeanFactory/ApplicationContext）

实例化BeanFactoryPostProcessor的实现类. postProcessBeanFactory()

实例化BeanPostProcessor的实现类

实例化InstantiatioAwareBeanPostProcessor实现类

执行InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation()

调用构造器

执行InstantiationAwareBeanPostProcessor.postProcessPropertyValues()

注入属性

BeanNameAware.setBeanName();

BeanFactoryAware.setBeanFactory();

ApplicationContextAware.setApplicationContext()

使用阶段:容器初始化成功，执行业务代码后，下面开始销毁容器

 BeanPostProcessor.postProcessBeforeInitialization（）

init-method

InitializingBean.afterPropertiesSet()

BeanPostProcessor.postProcessAfterInitialization()

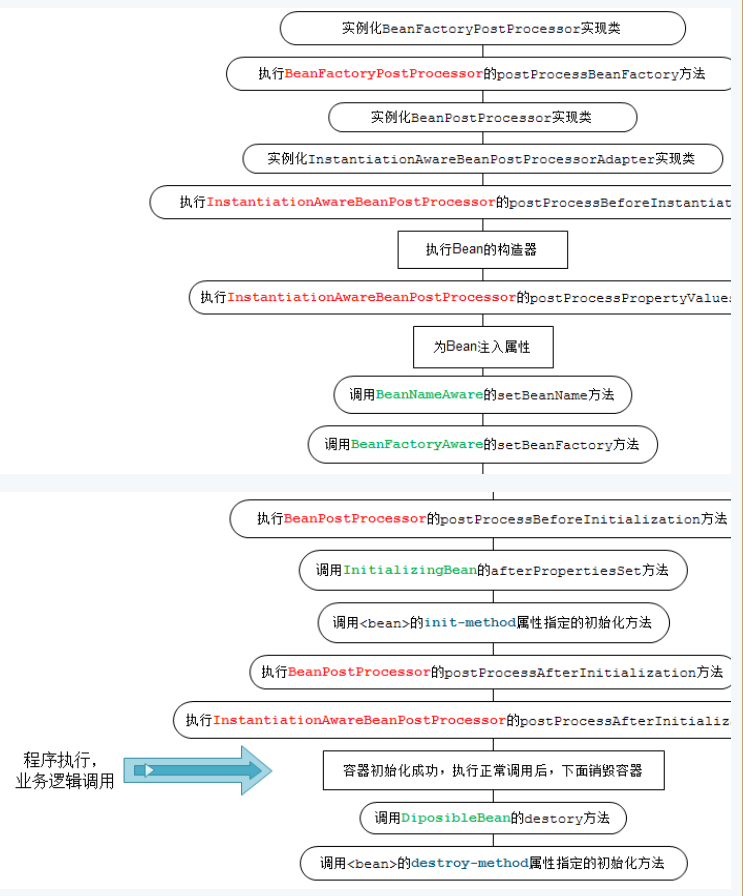
实例化阶段

初始化阶段

 DiposiableBean.destory()

destroy-method

销毁阶段



# 4 bean的配置文件是什么样的呢？

## 4.1 文件约束：使用的是schema约束

## 4.2 <bean>标签的id和name配置

id ：使用了约束只不过是唯一约束。里面不能出现特殊字符。

name: 没有使用唯一约束（所以理论上是可以重复的，但是实际开发中不允许重复），里面可以有特殊字符。

## 4.3 Bean的生命周期的配置

<bean>标签中有

init-method: bean被初始化时调用的方法。

destroy-method bean被销毁时调用的方法（bean是单例创建，工厂关闭时销毁调用。如果时是多例的，则由于存在多个对象，不知道对哪个对象调用，则不调用）

<bean name=”userDao” class=”cn.htb.UserDaoImpl” init-method=”setUp” destroy-method=”destroy”> <bean/>

public class UserDaoImpl implements UserDao{  
 public void setup(){

Systom.out.println(“被初始化”)；

}

public void destroy(){

Systom.out.println(“被初销毁”)；

}

}

class test{

// ApplicationContext接口本身没有关闭方法，但是其实现类有关闭方法

ClassPathXmlApplictionContext applictionContext =

new ClassPathXmlApplictionContext(“applicationContext.xml”)

UserDao userDao = (UserDao) applictionContext.getBean(“userDao”);

applicationContext.close();

}

## 4.4 bean的作用范围的配置

<bean>标签中的scope配置：控制bean的作用范围

singleton：默认的，Spring会采用单例模式创建这个对象。只创建一次，处理多个需求。

prototype：多例模式，每次需要时都创建一个新的对象。（和struts2整合时使用）

request：应用在Web项目中，Spring创建这个对象后，放置在request范围中。

session：应用在Web项目中，Spring创建这个对象后，放置在session范围中。

globalsession：应用在Web项目中，只能在porlet环境下使用（全局session），多用于有子系统的项目中，比如：QQ 和QQ空间，当在QQ中登录后，就不需要再次在QQ空间中登录。如果没有在porlet环境下，则相当于session。

# 5 bean的实例化方式：

## 5.1 使用类的无参构造创建（重点）

一个PIJO：

public class Car{

private String name;

private String price;

private Car(){ }

}

配置文件配置：

<bean name=”car” class=”cn.htb.Car”><bean/>

测试获取实例对象：

ApplictionContext applictionContext =

new ClassPathXmlApplictionContext(“applicationContext.xml”)

Car car = (Car) applictionContext.getBean(“car”);

## 5.2 使用静态工厂创建

创建类POJO：

public class Car{ }

创建静态工厂类：

public class MyBeanFactory{

//静态方法

public static Car getBean(){

return new Car();

}

}

配置文件配置：

<bean name=”car” class=”cn.htb.MyBeanFactory” factory-method=”getBean”><bean/>

测试获取实例：

ApplictionContext applictionContext =

new ClassPathXmlApplictionContext(“applicationContext.xml”)

Car car = (Car) applictionContext.getBean(“car”);

## 5.3 使用实例工厂创建

创建类POJO：

public class Car{ }

创建实例化工厂类：

public class MyBeanFactory{

//非静态方法

public Car getBean(){

return new Car();

}

}

配置文件配置：

<bean name=”myBeanFactory” class=”cn.htb.MyBeanFactory” ><bean/>

<bean name=”car” factory-bean=” myBeanFactory” factory-method=”getBean”><bean/>

测试获取实例：

ApplictionContext applictionContext =

new ClassPathXmlApplictionContext(“applicationContext.xml”)

Car car = (Car) applictionContext.getBean(“car”);

静态工厂类与非静态工厂类的区别:，

前者不需要创建对象，直接可以调用静态方法创建bean；

后者则要先创建对象，然后再通过对象调用其方法创建bean

# 6 bean的属性注入（依赖注入）

## 6.1 先了解通常情况下属性注入有哪些方法？

构造方法注入—set方法注入—接口注入

构造方法注入：

public class Cat{

private String name;

private String price;

public Cat(String name,String price){

this.name = name;

this.price = price;

}

}

//注入

Cat cat = new Cat(“猫”，“100”)；

set方法注入：

public class Cat{

private String name;

private String price;

public void setName(){ this.name = name }

public void setPrice(){ this.price = price }

}

}

//注入

Cat cat = new Cat();

cat.setName(“猫”);

cat.setPrice(“100”);

接口方法注入：

public interface Injectson{

public void setName(String name);

public void setPrice(String price);

}

实现接口：

public class Cat implements Injectson{

private String name;

private String price;

public void setName(){ this.name = name }

public void setPrice(){ this.price = price }

}

//注入

Cat cat = new Cat();

cat.setName(“猫”);

cat.setPrice(“100”);

## 6.2 但是在Spring中主要支持四种属性注入方法：

#### 0 注解注入

#### 1 set方法注入（最常用）：需要给类属性（依赖）设置set方法

#### 2 构造方法注入。需要给类设置有参构造方法

#### 3 P名称空间属性注入（Spring2.5以后）

#### 4 SpEL属性注入（Spring3以后）

set方法注入：

//创建对象

public class Cat{

private String name;

private String price;

public void setName(){ this.name = name }

public void setPrice(){ this.price = price }

}

//配置文件

<bean name=”cat” class=”cn.htb.Cat”>

<property name=”name” value=”宝马” />

< property name=”price” value=”10000000” />

<bean/>

/注入对象使用：<property name=”name” ref=”其他bean的id或name” />

构造方法注入：

//创建对象

public class Cat{

private String name;

private String price;

public Cat(String name,String price){

this.name = name;

this.price = price;

}

}

//配置文件

<bean name=”cat” class=”cn.htb.Cat”>

<constructor-arg name=”name” value=”宝马” />

<constructor-arg name=”price” value=”10000000” />

<bean/>

/注入对象使用：< constructor-arg name=”name” ref=”其他bean的id或name” />

p名称空间注入：

约束名称空间修改：修改后就能用了

<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"*>

//创建对象

public class Cat{

private String name;

private String price;

public void setName(){ this.name = name }

public void setPrice(){ this.price = price }

}

//配置文件

//普通属性

<bean name=”cat” class=”cn.htb.Cat” p:name=“宝马” p:price=”100000“><bean/>

//对象依赖

<bean name=”cat” class=”cn.htb.Cat” p:mouser-ref =其他bean的id或name”><bean/>

SpEL注入：

//创建对象

public class Cat{

private String name;

private String price;

public void setName(){ this.name = name }

public void setPrice(){ this.price = price }

}

//配置文件

//普通属性

<bean name=”cat1” class=”cn.htb.Cat” >

<property name=”name” value=”#{‘宝马’}”>

<property name=”price” value=”#{‘10000’}”>

<bean/>

//对象依赖

<bean name=”cat2” class=”cn.htb.Cat” >

<property name=”price” value=”#{ cat1}”>

<bean/>

//使用别的对象的属性

<bean name=”cat2” class=”cn.htb.Cat” >

<property name=”price” value=”#{ cat1.name}”>

<property name=”price” value=”#{ cat1.eat（） }”>//别的对象的方法

<bean/>

## 6.3 复杂类型的注入：数组/集合等list set array map props（这个只能存字符串）

<bean name=”cat2” class=”cn.htb.Cat” >

<property name=”price”>

<list>

<value><value/>

<ref />

<list/>

<property/>

<bean/>

<bean name=”cat2” class=”cn.htb.Cat” >

<property name=”price”>

<set>

<value><value/>

<ref />

<set/>

<property/>

<bean/>

<bean name=”cat2” class=”cn.htb.Cat” >

<property name=”price”>

<map>

<entry key=”” value=””>

<entry key-ref=”” value-ref=””>

<map/>

<property/>

<bean/>

# 7 分模块开发

## 5.1 加载配置文件时，加载多个文件

ApplictionContext applictionContext =

new ClassPathXmlApplictionContext(“applicationContext.xml”， “applicationContext2.xml”)

## 5.2 在一个配置文件中，引入多个其他的配置文件

<import resources=”cn/htb/applicationContest2.xml”>

<import resources=”cn/htb/applicationContest3.xml”>

# 8 IOC注解开发

7个基本jar包

core.jar beans.jar context.jar expression.jar common-logging.jar log4j.jar aop.jar

## 8.1添加约束context

### 8.1.1配置文件：打开注解扫描

组件扫描<context:component-scan base-package=””>

## 8. 2类注解一个Bean：

@Component(“userDao”):相当于，配置了一个name=“userDao”的<bean>

@Controller（“userController”）

@Service（”userService”）

@Repository（”userDao”）

当注解没有写name时，则默认时注释类的名称首字母小写

@Component(“userService”) //相当于<bean name=”userDao” clas=””><bean/>

public class UserServiceImpl implements UserService{

}

## 8.3注解属性：

### 8.3.1普通属性：有set方法和没有set方法

@Component(“userService”)

public class UserServiceImpl implements UserService{

private String name;

@Value(“张三”) //可以直接注入值：张三

public void setName(){

this.name = name;

}

}

@Component(“userService”)

public class UserServiceImpl implements UserService{

@Value(“张三”) //可以直接注入值：张三

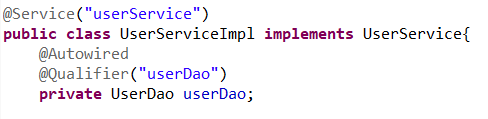
private String name;

}

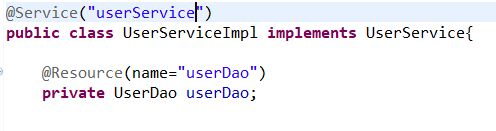
### 8.3.2对象属性注入

@Autowired按照对象类型完成注入

@Qualifier（“userDao”）与@Autowired结合使用，按照对象名称完成注入。@Qualifier不可以单独使用



@Resource按照对象名称完成属性注入（不属于spring而是javax.annotation.Resource;）@Resource(name="userDao")



## 8.4其他注解

### 8.4.1 bean的生命周期注解

@PostConstruct 初始化方法注解初始化之前调用）

@PreDestroy 销毁方法注解（容器销毁时调用，比如调用工厂的close()方法）

### 8.4.2 bean的作用域的注解

@Scope

参数：

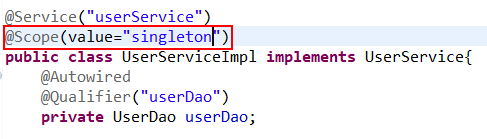
singleton：默认单例

prototype：多例

request

session

globalsession



# 9 IOC 的XML配置开发和注解开发比较

方式：

## 9.1 XML开发：可以适用于任何场景

结构清晰，易于维护。（切换底层时只需要修改配置文件即可）

## 9.2 注解开发：有些地方不能使用，比如：不是自己提供的类，源码包中的类。

开发速度快，但不易于维护（切换底层时需要修改源代码中的注解信息）

## 9.3 XML和注解的整合开发

使用XML配置管理bean。

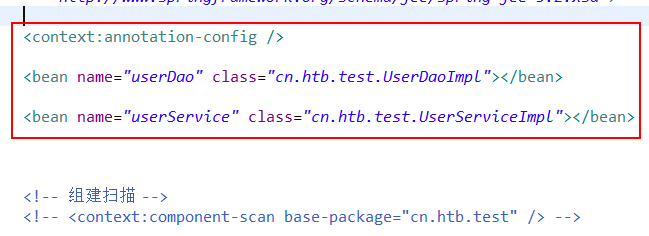
使用注解管理属性注入。

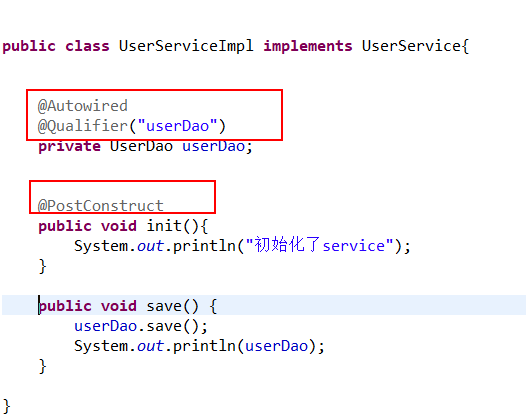
1 此时，不需要在XML中开组件扫描<context:component-scan package=”” />,

该扫描是用于扫描的是类上的注解（@Component @Controller @Service @Repository @Scope）

2 只需要打开注解配置即可 <context:annotation-config />

以允许在类中使用注解（@Resource @Autowired @Qualifier @PostConstruct @PreDestroy）





# 第二大部分---AOP

# 10 AOP面向切面编程

4个基本jar包：

aspectJ.jar aopaliance联盟 spring-aop.jar spring-aspects.jar test.jar

## 10.1 AOP概念

可用于权限校验、日志记录、事务管理、性能监控（就是运行时间的计算）等。

（推荐使用）系统的多个模块存在交叉业务（比如安全检验、事务管理、日志记录等），多个模块中相同的交叉业务连在一起就形成了系统的一个切面，AOP就是将交叉业务模块化，使之可以独立管理，降低和逻辑业务的耦合性。

同时结合声明式预编译和动态代理技术，方便的增加系统的业务功能。

AOP为Aspect Oriented Programming的缩写，意为：面向切面编程。是对OOP编程的扩展和延申，用于解决OOP开发中遇到的问题。

针对业务处理过程中的切面进行横切抽取，它所面对的是处理过程中的某个步骤或阶段，并使之模块化,以降低逻辑过程中各部分之间[耦合性](https://baike.so.com/doc/5725915-5938652.html)，让组件更专注入核心业务逻辑。

最后再通过预编译方式和运行期动态代理将其织入需要管理的Bean对象中，来实现系统功能的增强和统一维护，是一种思想，而非技术。

（比如：将程序的权限校验和日志记录等系统功能， 和他的普通业务功能，如增删改查进行隔离，也就降低了耦合、提高了可重用性、还利于维护）

AOP通过预编译和动态代理+配置的思想，来完成多模块相同功能的增强和维护。

预编译和动态代理，能够完成方法和功能增强；

配置则能够降低程序的耦合度，便于管理。

对程序不同组件的核心业务以外的系统功能。

AOP是OOP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

AOP:针对业务处理过程进行横切提取，它所面对的是处理过程中的某个步骤或者阶段,以获得逻辑过程中各部分之间低耦合的隔离效果。AOP做到了代码块的重用。

  通常，系统由很多组件组成，每个组件负责一部分功能，然而，这些组件也经常带有一些除了核心功能之外的附带功能 。系统服务如日志、事务管理和安全经常融入到一些其他功能模块中。这些系统服务通常叫做交叉业务，这是因为它们总是分布在系统的很多组件中。通过将这些业务分布在多个组件中，给我们的代码引入了双重复杂性。

(1) 实现系统级业务的代码在多个组件中复制。这意味着如果你要改变这些业务逻辑，你就必须到各个模块去修改。就算把这些业务抽象成一个独立模块，其它模块只是调用它的一个方法，但是这个方法调用也还是分布在很多地方。

(2) 组件会因为那些与自己核心业务无关的代码变得杂乱。一个向地址录中添加条目的方法应该只关心如何添加地址，而不是关心它是不是安全或支持事务的。

此时，我们该怎么办呢？

这正是AOP用得着的地方。AOP帮助我们将这些服务模块化，并把它们声明式地应用在需要它们的地方，使得这些组件更加专注于自身业务，完全不知道其它涉及到的系统服务。

## 10.2 AOP原理

理解一：

　spring用代理类包裹切面，把他们织入到Spring管理的bean中。也就是说代理类伪装成目标类，它会截取对目标类中方法的调用，让调用者对目标类的调用都先变成调用伪装类，伪装类中就先执行了切面，再把调用转发给真正的目标bean。

　　现在可以自己想一想，怎么搞出来这个伪装类，才不会被调用者发现（过JVM的检查，JAVA是强类型检查，哪里都要检查类型）。

　　1.实现和目标类相同的接口，我也实现和你一样的接口，反正上层都是接口级别的调用，这样我就伪装成了和目标类一样的类（实现了同一接口，咱是兄弟了），也就逃过了类型检查，到java运行期的时候，利用多态的后期绑定（所以spring采用运行时），伪装类（代理类）就变成了接口的真正实现，而他里面包裹了真实的那个目标类，最后实现具体功能的还是目标类，只不过伪装类在之前干了点事情（写日志，安全检查，事物等）。

　　这就好比，一个人让你办件事，每次这个时候，你弟弟就会先出来，当然他分不出来了，以为是你，你这个弟弟虽然办不了这事，但是他知道你能办，所以就答应下来了，并且收了点礼物（写日志），收完礼物了，给把事给人家办了啊，所以你弟弟又找你这个哥哥来了，最后把这是办了的还是你自己。但是你自己并不知道你弟弟已经收礼物了，你只是专心把这件事情做好。

　　顺着这个思路想，要是本身这个类就没实现一个接口呢，你怎么伪装我，我就压根没有机会让你搞出这个双胞胎的弟弟，那么就用第2种代理方式，创建一个目标类的子类，生个儿子，让儿子伪装我

　　2.生成子类调用，这次用子类来做为伪装类，当然这样也能逃过JVM的强类型检查，我继承的吗，当然查不出来了，子类重写了目标类的所有方法，当然在这些重写的方法中，不仅实现了目标类的功能，还在这些功能之前，实现了一些其他的（写日志，安全检查，事物等）。

这次的对比就是，儿子先从爸爸那把本事都学会了，所有人都找儿子办事情，但是儿子每次办和爸爸同样的事之前，都要收点小礼物（写日志），然后才去办真正的事。当然爸爸是不知道儿子这么干的了。这里就有件事情要说，某些本事是爸爸独有的(final的)，儿子学不了，学不了就办不了这件事，办不了这个事情，自然就不能收人家礼了。

　　前一种兄弟模式，spring会使用JDK的java.lang.reflect.Proxy类，它允许Spring动态生成一个新类来实现必要的接口，织入通知，并且把对这些接口的任何调用都转发到目标类。

　　后一种父子模式，spring使用CGLIB库生成目标类的一个子类，在创建这个子类的时候，spring织入通知，并且把对这个子类的调用委托到目标类。

　　相比之下，还是兄弟模式好些，他能更好的实现松耦合，尤其在今天都高喊着面向接口编程的情况下，父子模式只是在没有实现接口的时候，也能织入通知，应当做一种例外。

以一个权限校验为例：

public class UserDao{

public void save(){

checkAuthority();

保存操作

}

public static void checkAuthority(){

校验规则。。。

}

}

这样每个Dao都要写校验方法，复用性低，难以维护。

给多个Dao设置权限校验：保存操作前权限校验

1传统方法1:给每个Dao添加校验方法，并调用

public class BaseDao{

public static void checkAuthority(){

校验规则。。。

}

}

public class UserDao extends BaseDao{

public void save(){

checkAuthority();

保存操作

}

}

这样，就可以只需要对BaseDao的一个方法进行维护即可，但是如果某天不再需要该校验了，这又是个问题，难道去一个个修改Dao不再继承吗？

2传统方法2:设置BaseDao，让别的dao继承，纵向继承

public class UserDao{

public void save(){

//保存操作

}

}

动态代理方式

UserDaoProxy

生成该Dao的代理对象，在代理对象中完成校验工作。

1 只需要维护该代理对象

2 不需要时直接删除该代理对象即可

动态代理的方法：

JDK动态代理：基于接口的代理，只能对接口的实现类生成代理对象

Cglib动态代理：对没有实现接口的类产生代理，通过生成目标类的子类来完成代理功能。

AOP

Spring中的AOP底层机制就是使用的动态代理实现的。（横向抽取）

只不过Spring底层已经封装好了代理方法，而且会自动切换JDK或者Cglib

### 10.2.1JDK代理案例

关键类：Proxy proxy. newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)

InvocationHandler

#### 精准分析：

1 目标对象必须实现接口

2 示例：

创建UserDAO接口

创建UserDaoImpl实现类

创建代理类：MyProxy

package cn.proxy.jdk;

public interface UserDao {

public void save();

}

package cn.proxy.jdk;

public class UserDaoImpl implements UserDao{

@Override

public void save() {

System.*out*.println("保存了数据。。。。");

}

}

创建代理类：

public class MyProxy{

public UserDao getProxy(){

Proxy.newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h);

//观察该方法参数：他需目标类的类 加载器，和他实现的接口数组，以及有一个InvocationHandler对象

//所以要在MyProxy类中传入参数属性UserDao

}

}

也就是如下的代码：

public class MyProxy{

public UserDao getProxy(UserDao userDao){

Proxy.newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h);

}

}

之后再分别获取参数并填入方法中，另外InvocationHandler是接口，所以用匿名内部类来写。

package cn.proxy.jdk;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.InvocationHandler;

import net.sf.cglib.proxy.Proxy;

public class MyProxy {

public Object getProxy(UserDao userDao){

UserDao proxy = (UserDao) Proxy.*newProxyInstance*(userDao.getClass().getClassLoader(), userDao.getClass().getInterfaces(), new InvocationHandler() {

@Override

public Object invoke(Object arg0, Method arg1, Object[] arg2) throws Throwable {

System.*out*.println("前面的处理");

Object obj = arg1.invoke(userDao, arg2);

System.*out*.println("后面的处理");

return obj;

}

});

return proxy;

}

}

测试代码：

public static void main(String[] args) {

//创建目标类对象

UserDao userDao = new UserDaoImpl();

//创建代理类

MyProxy proxy = new MyProxy();

//获取代理对象

UserDao user = (UserDao) proxy.getProxy(userDao);

user.save();

}

通用处理：

package cn.proxy.jdk;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.InvocationHandler;

import net.sf.cglib.proxy.Proxy;

public class MyProxy {

public Object getProxy(Object obj){

Object proxy = Proxy.*newProxyInstance*(obj.getClass().getClassLoader(), obj.getClass().getInterfaces(), new InvocationHandler() {

@Override

public Object invoke(Object arg0, Method arg1, Object[] arg2) throws Throwable {

System.*out*.println("前面的处理");

Object o = arg1.invoke(obj, arg2);

System.*out*.println("后面的处理");

return o;

}

});

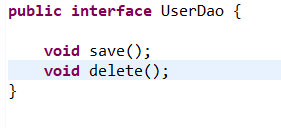
return proxy;

}

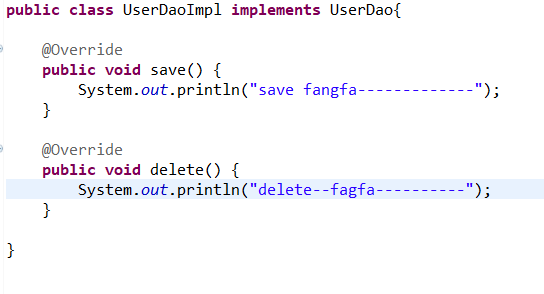
}

#### 示例二：

接口

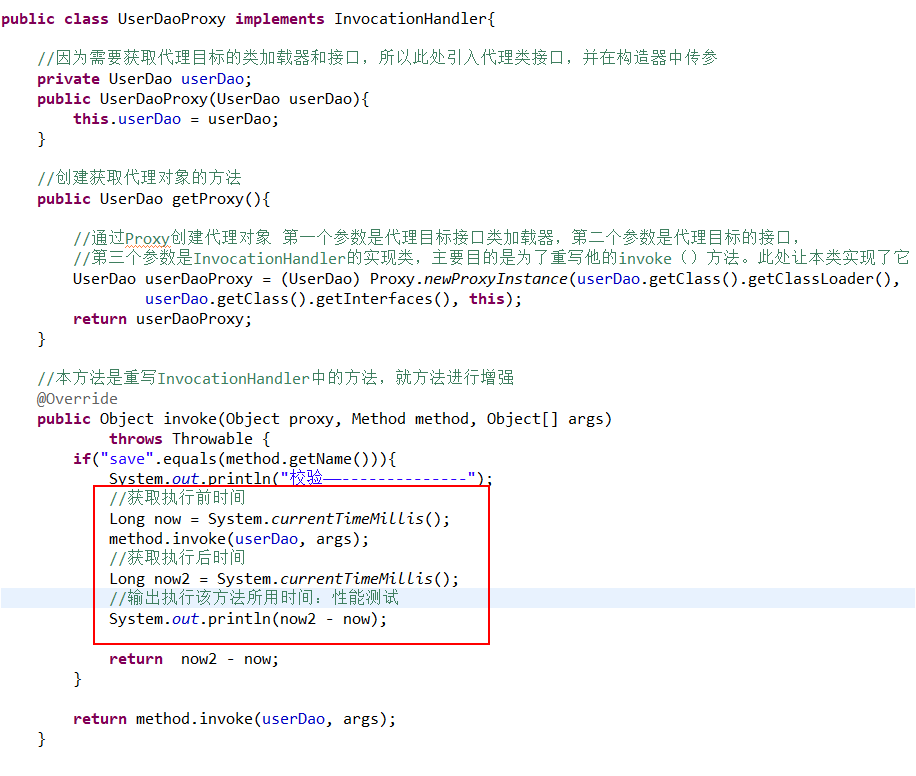


实现类



代理对象获取：

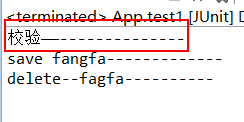




测试类：



效果：



### 10.2.2Cglib动态代理

关键类：

Enhancer

enhancer.setSuperclass(userDao.getClass());

enhancer.setCallback(**new** MethodInterceptor(){})；

enhancer.create();

CallBack 的子类：MethodInterceptor

精准分析：

package cn.proxy.cglib;

import java.lang.reflect.Method;

import org.springframework.cglib.proxy.Enhancer;

import org.springframework.cglib.proxy.MethodInterceptor;

import org.springframework.cglib.proxy.MethodProxy;

public class MyProxy {

public Object getProxy(UserDao userDao){

Enhancer enhancer =new Enhancer();

//设置目标类

enhancer.setSuperclass(userDao.getClass());

//设置回调方法，类似InvocationHandler

enhancer.setCallback(new MethodInterceptor() {

@Override

public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {

System.*out*.println("qian...");

//使用代理方法，入参为代理对象和参数

Object obj = methodProxy.invokeSuper(proxy, args);

//或者使用method，入参为目标对象和参数

//Object obj = method.invoke(userDao, args);

System.*out*.println("hhou...");

return obj;

}

});

//创建并返回代理对象

return enhancer.create();

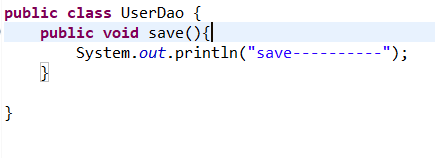
}

}

1 Cglib代理是通过创建目标类的子类来完成增强功能的，所以目标类不能被finally修饰，因为被final修饰的类无法创建子类。

2 要被增强的方法，也不能被final修饰，因为无法重写。

实现类：



代理类：



测试类：



## 10.3 AOP详解

AOP最初是由一个叫做AOP联盟的组织提出，后被Spring引入。

### 10.3.1Spring有两条AOP方法：

spring自身的AOP方法，比较繁琐（弃用）

引入的AspectJ方法，现在使用的方法。

### 10.3.2 spring述语了解

public void userDao{

void save();

void delete();

void update()

void query()

}

JoinPoint:连接点。目标类中所有可以被拦截到的点（方法）；

此处的增删改查，都可以被拦截和增强，都是连接点。

PointCut切入点。真正被拦截增强的点。

在实际开发中，真正被拦截增强的的点，如对save()方法增强校验。

Advice通知/增强。对切入点进行增强的方法。方法层面的增强。

如；对save方法进行权限校验，则这个权限校验(checkAuthority）就是一个通知。

Intorduction引介。类层面的增强。

Target：被增强的对象。

如对UserDao增强，则UserDao就是目标。

Weaving织入：将通知（Advice）应用到目标（Target）的过程。

Proxy代理对象。

Aspect切面，多个通知和多个切入点的组合。

所以配置文件中的，设置切面就是设置通知和切入点。

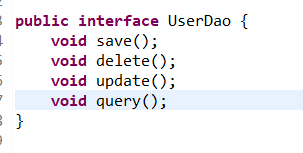
（Advice和Pointcut）

）

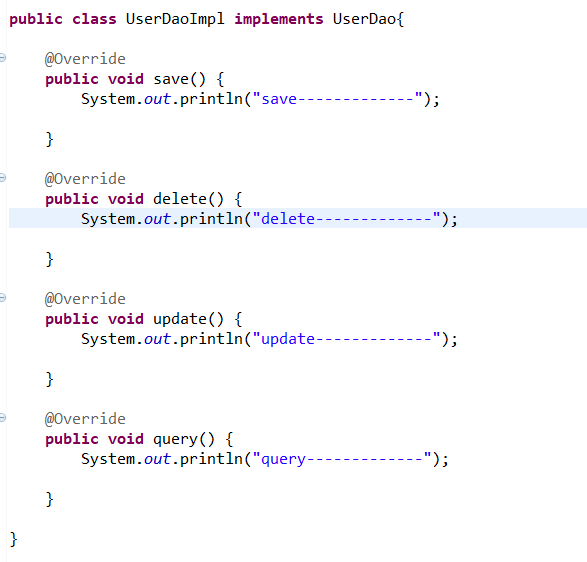
## 10.4 XML配置AOP切面示例

### 10.4.1简单示例

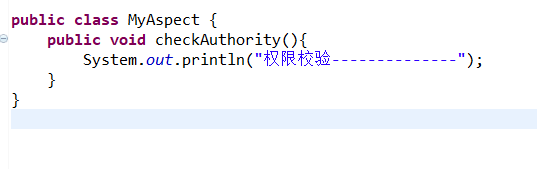
接口：



实现类：



切面类：



xml配置：





切面：就是 多个切入点 和 多个通知 的组合

测试类：



### 10.4.2 AOP中的通知类型：就是增强方法执行相对于目标方法的顺序

切面代码：



10.5.1前置通知（before）: 在切入点之前执行.可以获取切入点信息（Joinpoint joinpoint）



10.5.2后置通知（after-returning）：在切入点之后执行。可以获取返回值（Object resut）



10.5.3环绕通知(around)：在切入点前后执行，还可以阻止切入点执行

（ProceedingJoinPoint proceeding） proceeding处理 🡪处理连接点



10.5.4异常抛出通知(after-throwing)：切入点抛出异常后执行.可以获取异常（Throwable e）

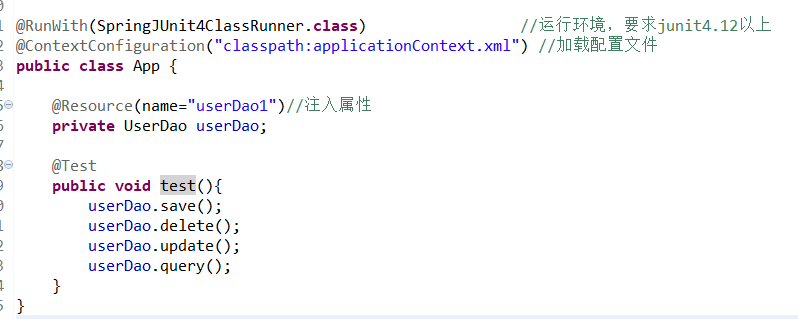


10.5.5最终通知(after)：无论切入点是否异常，都会执行。相当于finally内的内容



10.5.6引介通知

测试代码：



### 10.4.3切入点公式

<aop:pointcut expression=*"execution(\* cn.htb.aoptest.UserDaoImpl.query(..))"* id=*"pointcut"*/>

1 写在execution（）中

2 第一种完整的：public void cn.htb.aoptest.UserDaoImpl.query(..)

3 第二种：\* cn.htb.aoptest.UserDaoImpl.query(..)

4 第三种：\* cn.htb..UserDaoImpl.query(..)

//cn.htb下的所有包内的所有UserDaoImpl类的query方法

5 第四种：\* cn.htb..UserDaoImpl+.query(..)

// cn.htb下的所有包内的所有UserDaoImpl类以及其子类的query方法

6 第五种：\* cn.htb..\*.query(..)

//cn.htb下所有包所有类的query方法

## 10.5 AOP注解开发

注解类型：

@Before : 前置通知

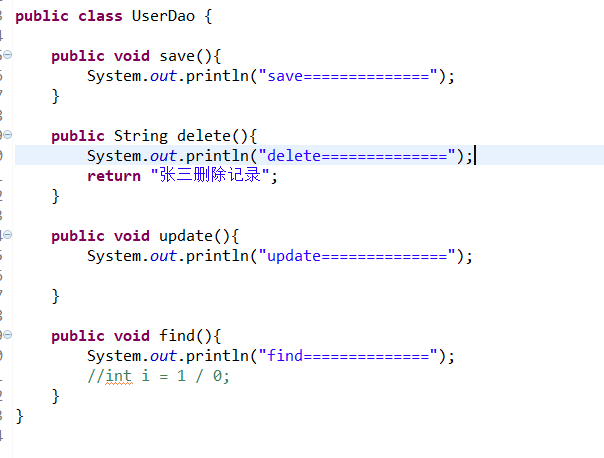
@After-returning ：后置通知

@Around ：环绕通知

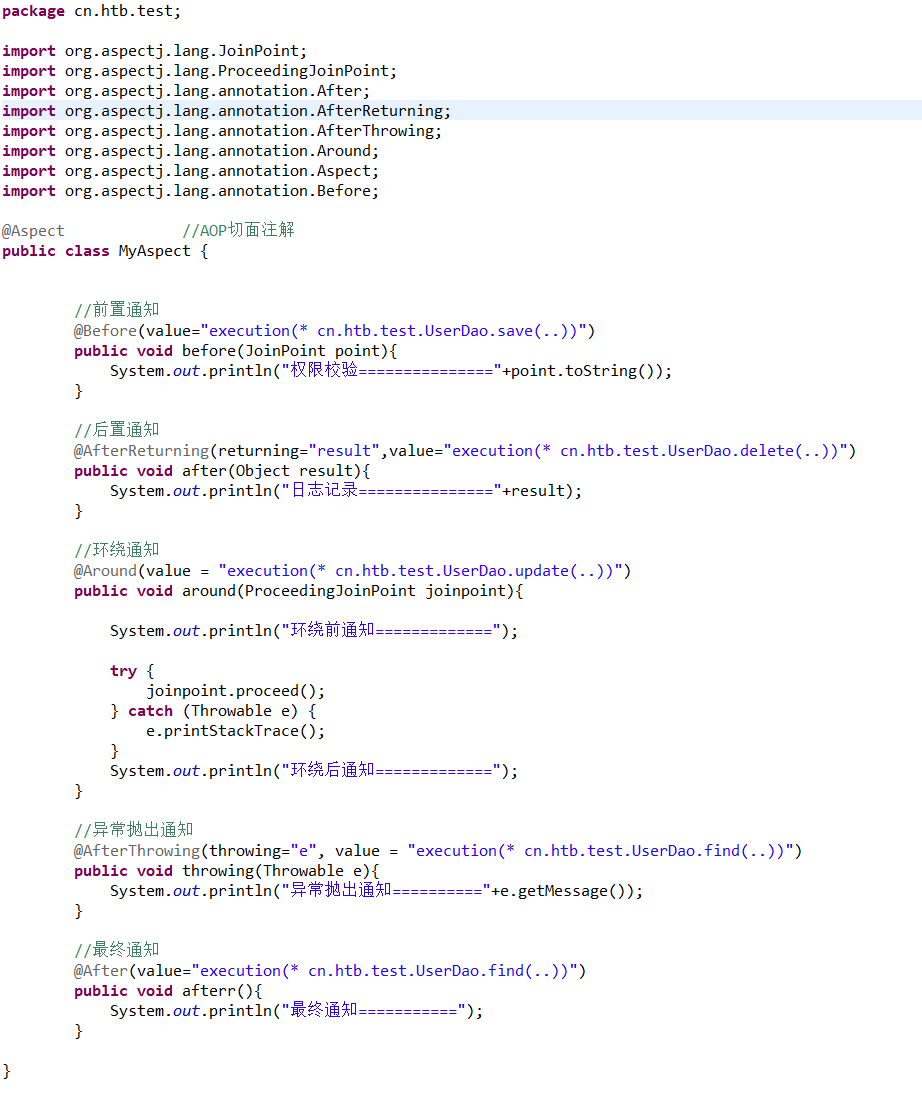
@After-throwing ：异常抛出通知

@After ：最终通知

实现类



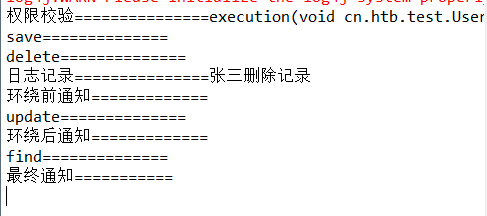
切面类



测试类



效果



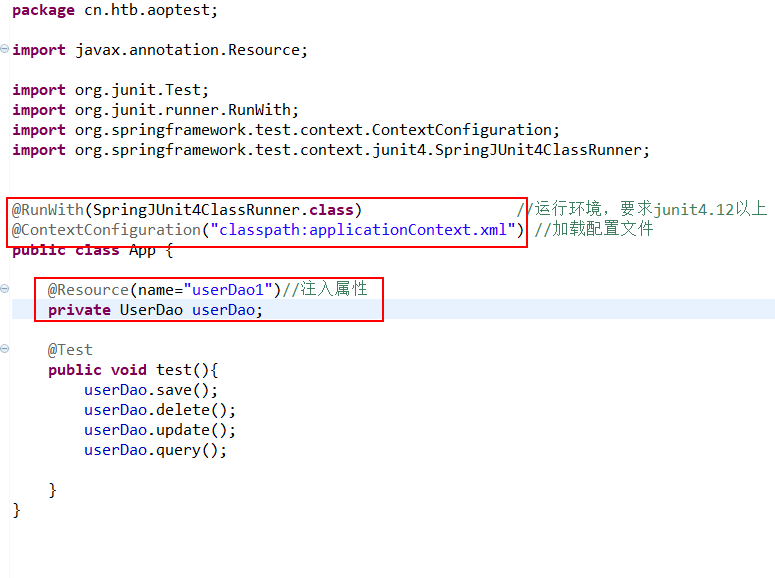
## 10.6切入点注解开发



AOP注解开发中可以配合切入点开发，使得稍微方便一点。

但是与xml相比，还是xml结构更清晰，更易于维护。

# 11 spring整合junit



# 第三大部分---DAO

# 12 Spring中JDBCTemplate使用

## 12.1 spring是EE开发的一站式框架，针对每个开发层都有一定的解决方法，在Dao层，它提供了JDBC模板和ORM框架。

## 12.2 JDBC模板的使用

需要jar包：

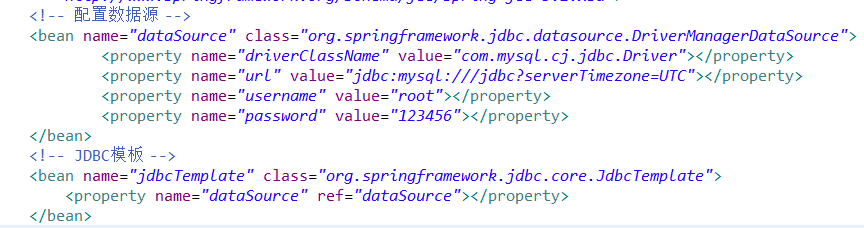
spring-jdbc.jar mysql-connector,jar aop.jar(部分依赖在这里) tx.jar(事务管理)

## 12.3数据库连接方式：

### 12.3.1jdbc连接

spring-jdbc.jar mysql-connector,jar aop.jar(部分依赖在这里) tx.jar(事务管理)

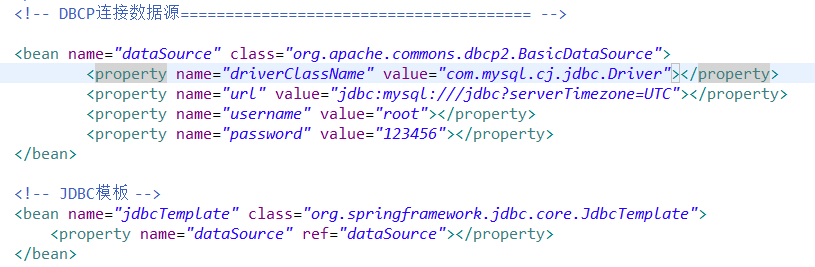
驱动类：DriverManagerDataSource



### 12.3.2DBCP连接

jar包：dbcp.jar pool.jar

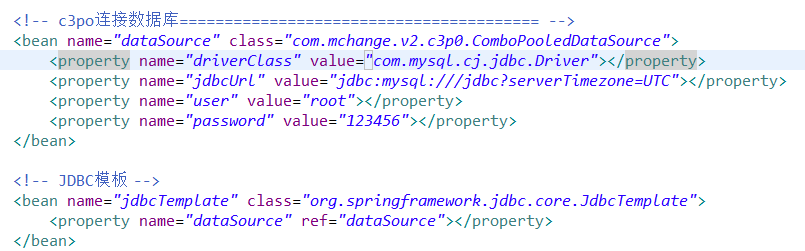
驱动类：BasicDataSource



### 12.3.3 c3p0连接

jar包：c3po.jar mchange-common.jar

驱动类：ComboPooledDataSource

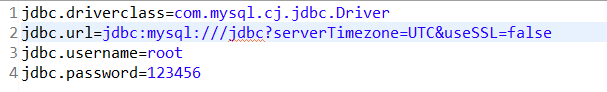


### 12.3.4引入外部属性文件

属性文件中使用&符号，不需要用amp;转义。

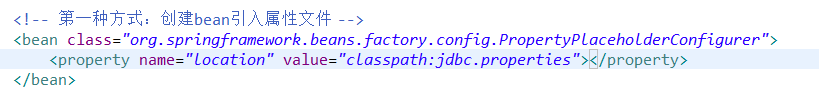
但是XML文件中，需要使用。

创建属性文件：



引入属性文件：

方式一：



方式二：常用



<!-- 开启注解 -->

<context:annotation-config/>

<!-- 引入属性文件 -->

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

<!-- 配置数据源 -->

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

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

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

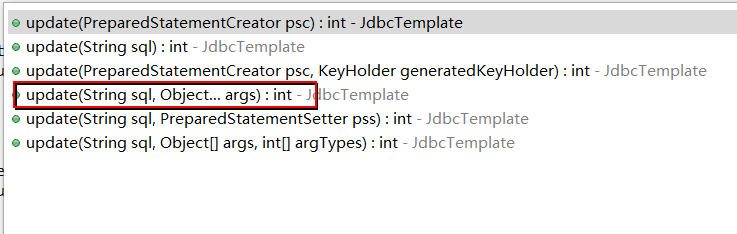
<property name=*"username"* value=*"${jdbc.username}"*></property>

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

</bean>

### 12.3.4JdbcTemplate代码操作

1 jdbcTemplate中的增删改都是使用update方法操作



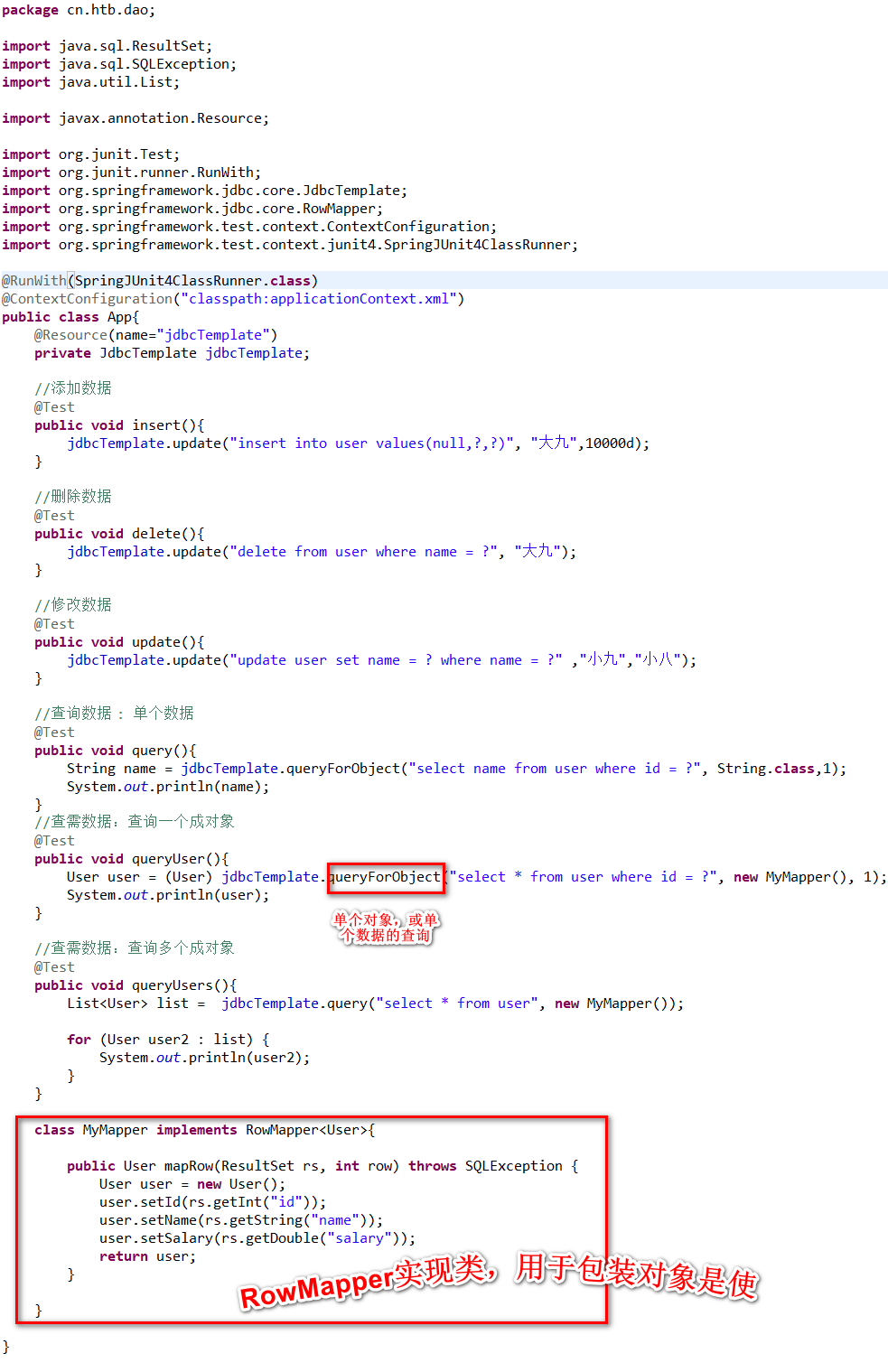
测试代码：

主要用到三个方法

update(sql. params) :增删改

queryForObject(sql, requiredType, params)查询单个数据，返回值Object

query(sql, requiredType, params)：查询多个数据，返回值List



# 第四大部分--- 事务管理

# 13 事务管理

## 13.1 事务简介

### 13.1.1 事务：

逻辑上的一组操作，组成这组操作的各个单元，要么全都成功，要么全都失败。

### 13.1.2 特性：

原子性：事务不可分割（不能一部分操作了，而其他的不操作）；

一致性：事务执行前后数据完整性一致；

隔离性：一个事务的执行不应该受到其他事务的干扰；

持久性：一旦事务结束，数据持久化到数据库。

### 13.1.3 如果不考虑隔离性会引发安全性问题

#### 读问题：

脏读：一个事务读到另一个事务未提交的数据。

幻读（虚读）：一个事务读到另一个事务已经提交的insert的数据，导致一个事务中多次查询的结果不一致。

不可重复读：一个事务读到另一个事务已经提交的update的数据，导致一个事务中多次查询的结果不一致。

#### 写问题：

丢失更新

### 13.1.4隔离级别（解决读问题）

Read uncommitted: 读未提交》解决不了任何问题。

Read comminted： 读已提交》 解决脏读问题，但是幻读和不可重复读可能发生 （orcle）

Repeatable Read: 可重复度》解决脏读和不可重复度。幻读可能发生。 （msql）

Serializable: 解决所有问题。

## 13.2 Spring的事务管理的API

### 13.2.1 PlatformTransactionManager平台事务管理器（接口）

DataSourceTransactionManager :底层使用JdbcTemplate管理事务

HibernateTransactionManager :底层使用Hibernate管理事务

### 13.2.2 TranscationDefinition 事务定义信息（接口）

事务定义：用于定义事务的相关的信息，如：隔离级别、传播行为、超时信息、是否只读

### 13.2.3 TransactionStatus : 事务的状态（接口）

事务状态：用于记录在事务管理过程中，事务的状态的对象。如：事务是提交了/是否回滚了

### 13.2.4 事务管理的API间的关系

Spring进行事务管理时，首先 平台事务管理器 会根据 事务定义信息 进行事务的管理；在事务管理过程中，会产生各种信息，将这些状态信息记录到 事务的状态 中。

## 13.3 事务的传播行为（三大类）

Service1{

void A(){

dao1();

dao2();

}

}

Service2{

void B(){

Service1.A();

dao3();

dao4();

}

}

如果遇到了特别复杂的业务逻辑，有可能会出现业务层之间的方法相互调用。

事务的传播行为主要用来解决业务层方法相互调用的问题。

### 13.3.1保证多个操作在同一个事务中

propagation\_required :默认值，如果A中有事务，则使用A的事务，如果A中没有事务，则创建一个新的事务，将操作包含起来。

propagation\_supports ： 支持事务，如果A中有事务，则使用A事务，如果没有事务，则不适用事务。

propagation\_mandatory ：如果A中有事务，使用A事务，如果没事务，则抛出异常。

### 13.3.2保证多个操作不在同一个事务中

propagation\_requiredNews：如果A中有事务，将A中事务挂起（暂停），创建新的事务，只包含自身的操作。如果A中没有事务，则创建一个新的事务，包含自身操作。

propagation\_notSupports ：如果A中有事务，将A的事务挂起，不使用事务管理。

propagation\_never ：如果A中有事务，报异常。

### 13.3.3嵌套式事务

propagation\_nested ：嵌套事务，如果A中有事务，按照A的事务执行，执行完成后，设置一个保存点，执行B事务。

如果没有异常，则执行通过；

如果有异常，则可以选择回滚到最初点（A事务前），也可以选择回滚到保存点位置（B事务之前）。

## 13.4 事务管理的两大类方式（编程式-声明式）

无论哪种事务管理方式，都需要先在xml文件中配置事务平台管理器。

### 13.4.1编程式事务管理

#### 13.4.1.1 配置数据源

<!-- 开启注解 -->

<context:annotation-config/>

<!-- 引入属性文件 -->

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

<!-- 配置数据源 -->

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

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

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

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

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

</bean>

#### 13.4.1.2 配置JdbcTemplate模板

注入daoSource: 因为dao继承了JdbcDaoSupport，其代码中包含了JdbcTempalte的set/get方法，所以直接在次注入模板即可

<!-- 配置dao -->

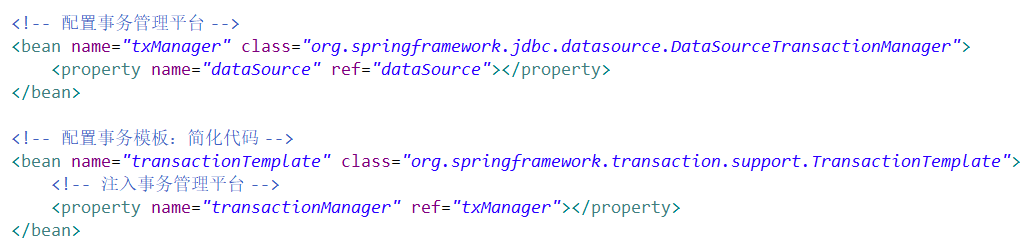
<bean name=*"userDao"* class=*"cn.htn.test.UserDaoImpl"*>

<!-- 注入daoSource: 因为dao继承了JdbcDaoSupport，其代码中包含了JdbcTempalte的set/get方法，所以直接在次注入模板即可 -->

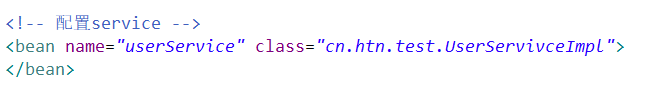
<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

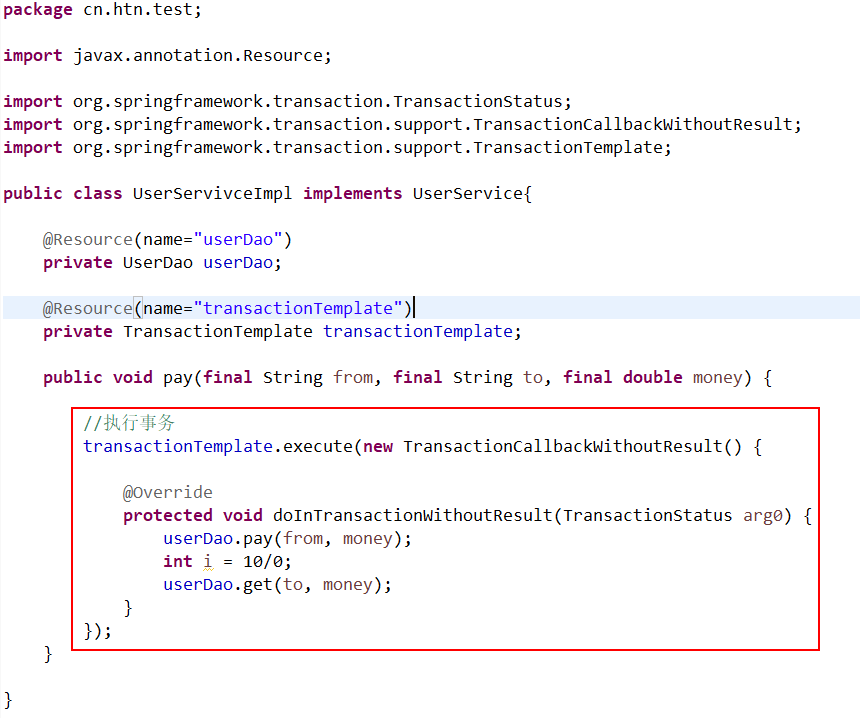
#### 13.4.1.3 配置事务管理平台



#### 13.4.1.4在service中注入事务管理模板



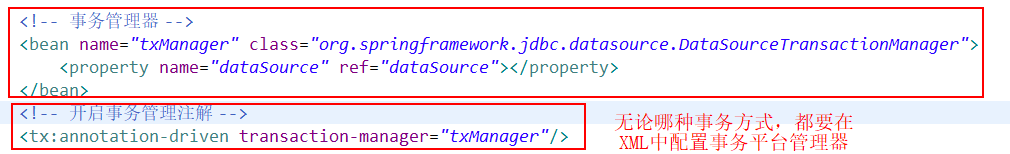
#### 13.4.1.5 编程式事务代码



### 13.4.2XML声明式事务管理



### 13.4.3 注解式声明式事务管理



# 14Hibernate延迟加载问题

哪些地方会出现延迟加载？

## 14.1使用Load()方法时。

## 14.2查询关联关系时。

查到某个对象后，显示其关联对象。

## 14.3 问题

1 问题描述：

在hibernate中开启延迟加载后，service层获取数据（Teacher），如果Web层展示数据（Students），就会抛出异常no-session.

2 问题原因：

主要原因是，开启延迟加载后，在service层中开启session并通过dao获取数据Teacher，但是因为开启了延迟加载，所以没有查询与其关联的Students，而且查询结束后关闭session。

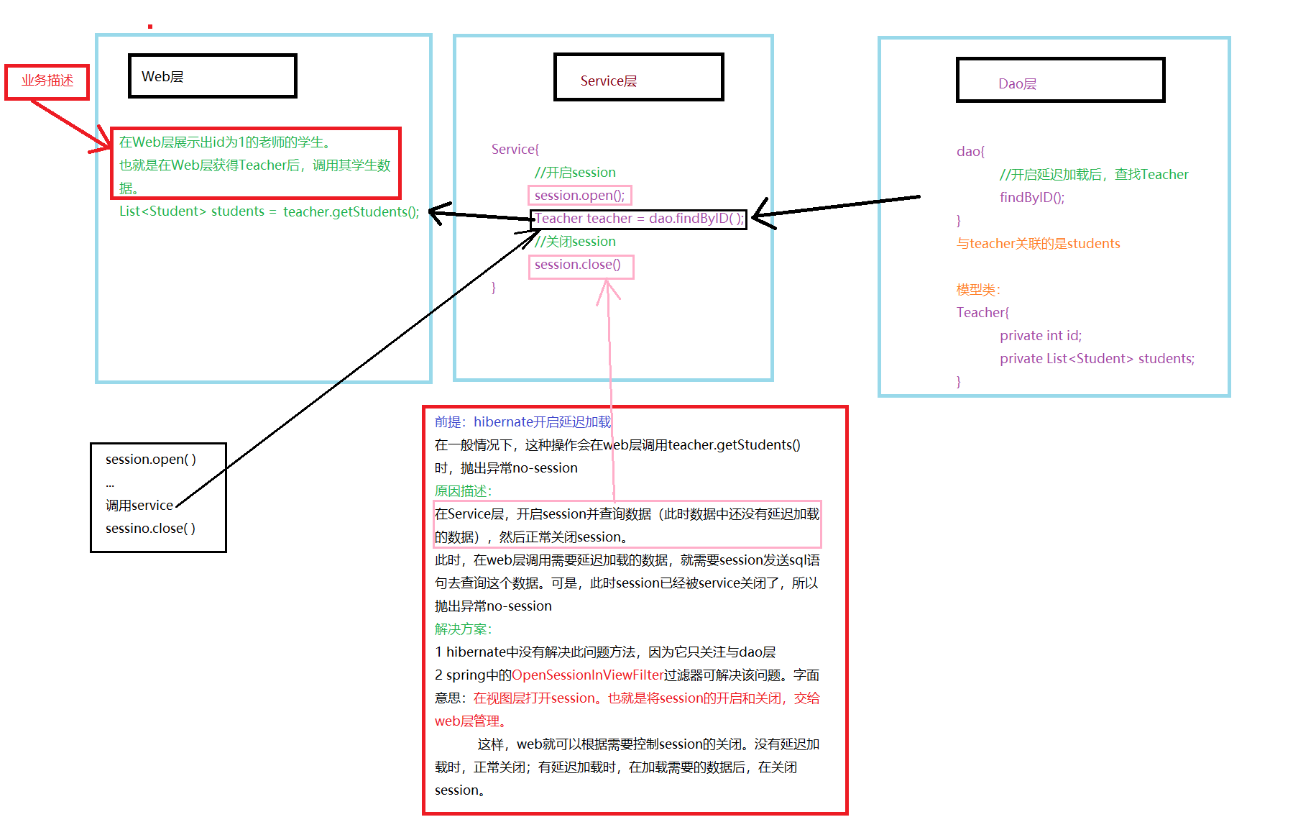
然而当我们在web层使用Teacher的Students时，需要session再次发送sql获取数据，可是此时session已经关闭了，无法发送sql，就会报错no-session。

3 解决方法：

首先，hibernate中并没有提供解决改问题的方法。但是改问题在spring中提供了解决方案。

spring中的OpenSessionInViewFilter过滤器 ：通过字面意思理解，在视图层打开session，也就是将session的管理（打开/关闭），从service层交给web控制。

这样，web层控制了session的生命，就可以在需要延迟加载的地方，先加载延迟数再关闭session；不需要延迟加载的地方就正常关闭，



# 第一次复习：整体回顾

1 两大核心思想：IOC/DI

IOC inversion Of Control 控制反转，将对象的创建权交给Spring容器管理，完成对象的创建、装配和销毁。

Dependence injection 依赖注入：被spring容器管理的对象，会在需要的地方自动完成装配。

IOC/DI原理：

这一切都是由BeanFactory /ApplicationContext控制的。配置文件+反射+工厂

## 2 Spring就是通过IOC来完成堆bean的管理

== bean的生命周期：

实例化-初始化-使用阶段-销毁阶段

实例化阶段：

工厂类BeanFactory/Application加载配置文件

BeanFactoryPostProcessor.postProcessBeanFactor()

BeanPostProcessor

InstantiationAwarePostProcessor.postProcessBeforeInstantiation()

调用构造方法

.postProcessPropertyValues()

BeanNameAware

BeanFactoryAware

ApplicationContextAware

初始化阶段：

BeanPostProcessor.postProcessBeforeInitilize（）

自定义初始化方法

InitlizingBean.afterPropertiesSet()

BeanPostProcessor.postProcessAfterInitilize()

使用阶段：自定装配

销毁阶段：

DispoableBean.destory()

自定义销毁方法

== 如何自定义初始化方法和销毁方法：

方法一：xml配置中

<bean init-method />

<bean destroy-method/>

方法二：注解开发

@PostConstract

@PreDestory

3 获取bean实例的方法：

== 构造方法获取

== 静态工厂获取

== 实例化工厂获取

4 此处提到了工厂模式：

== 简单工厂模式（又叫静态工厂模式：Spring使用）

== 工厂方法模式

== 抽象工厂模式

## 5 DI依赖注入

== spring中依赖注入的方法：

set/get方法注入（常用）

构造参数方法注入

P名称空间注入 （需要修改约束）

SpEL方法注入

## 6 AOP

AOP aspect oritened programming 面向切面编程。是对oop思想的补充，

主要是对服务的逻辑处理过程切面进行横切抽取，并使之模块化，以提高其复用性、降低服务各逻辑过程间的耦合性.让组件更加关注于核心业务的开发。

最后在通过预编译方法和动态代理技术，将其织入到需要管理的bean对象中，已完成对系统功能的增强和维护。

7 动态代理：

JDK动态代理：兄弟级 InvocationHandler Proxy

CGlib动态代理：父子级MethodInteceptor Enchancer

8 增强相关述语：

代理对象

目标对象

连接点

切入点

通知

切面

织入

引入

9 增强类型：

前置通知

后置通知

环绕通知

异常抛出通知

最终通知

## 10 dao JDBCTemplate模板

主要方法：

update（） ：增删改

queryForObject()

query():

涉及类：RowMapper

## 11 事务管理：

== 事务的四大特性：

原子性 一致性 隔离性 持久性

== 不关注隔离性造成的问题：

读：脏读 幻读 不可重复读

== 隔离级别：

读未提交

读已提交

可重复读

Serialiable

== 事务管理的实现：三大API

平台事务管理器 ： PlatformTransactionManager

事务定义: TransactionDefiniton关于事务的设置：传播行为、隔离级别、超时时间、是否只读

事务状态:TransactionStatus 是否提交/是否回滚

== 三者关系：

平台事务管理器根据，事务的定义信息进行事务管理，在管理过程中产生的消息记录到事务状态中。

== 三者中最为重要的是PlatformTransactionManager

他有两个重要子类：

DataSourceTransactionManager

HibernateTransactionManager