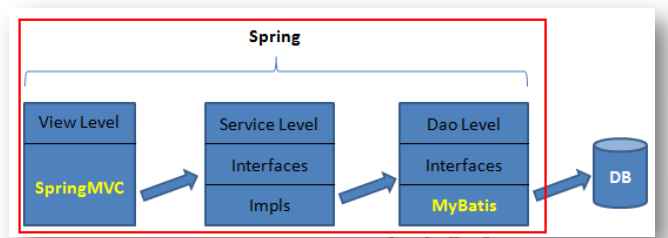
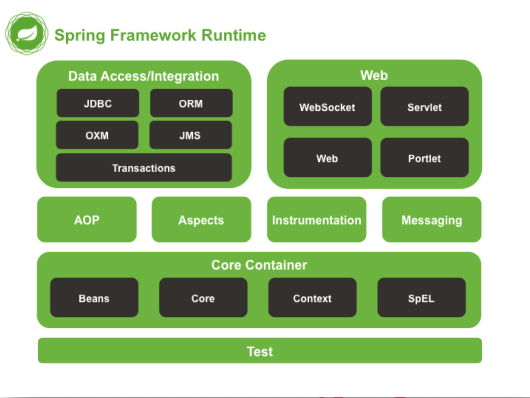
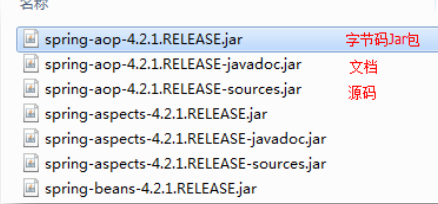
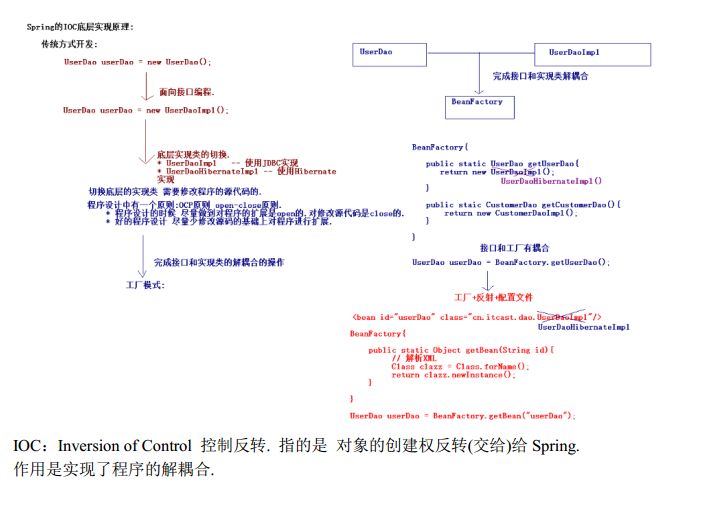
# Spring架构图







## IOC底层实现原理



### ========== demo01 ==========

导包(4+2)

一个最基本的Spring程序需要以下包(4+2)

E:\01\_JAVA\101\_software&doc\15\_spring\spring-framework-4.2.4.RELEASE-dist\spring-framework-4.2.4.RELEASE\libs

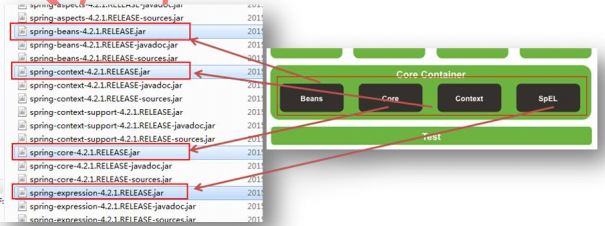
-核心容器包4个

-spring-beans-4.2.4.RELEASE.jar

-spring-context-4.2.4.RELEASE.jar

-spring-core-4.2.4.RELEASE.jar

-spring-expression-4.2.4.RELEASE.jar



E:\01\_JAVA\101\_software&doc\15\_spring\spring-framework-3.0.2.RELEASE-dependencies\org.apache.commons\com.springsource.org.apache.commons.logging\1.1.1

-日志记录包1个

-com.springsource.org.apache.commons.logging-1.1.1.jar

E:\01\_JAVA\101\_software&doc\15\_spring\spring-framework-3.0.2.RELEASE-dependencies\org.apache.log4j\com.springsource.org.apache.log4j\1.2.15

-日志记录依赖包1个

com.springsource.org.apache.log4j-1.2.15.jar

### ========== demo01 ==========

1.定义接口：IStudentService

2.定义实体类：StudentServiceImpl

3.新建Spring配置文件：在src下，命名为application.xml(测试用，使用application-demo1.xml)

从file:///E:/01\_JAVA/101\_software&doc/15\_spring/spring-framework-4.2.4.RELEASE-dist/spring-framework-4.2.4.RELEASE/docs/spring-framework-reference/html/xsd-configuration.html

拷贝xml的文件头

4.定义bean元素

<?xml version="1.0" encoding="UTF-8"?>

<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 definitions here -->

<!--

id是bean实例的唯一标识，程序通过id访问bean，bean之间的依赖关系通过id关联

class指定bean所属的类

-->

<bean id="studentService" class="demo1.StudentServiceImpl"></bean>

</beans>

5.定义测试类TestDemo

(Eclipse中类名上按Ctrl+T可以查看继承关系)

获取容器有2个接口可以实现：ApplicationContext(开发中使用)和BeanFactory

-分析ApplicationContext(extends HierarchicalBeanFactory -> extends BeanFactory)

-有2个实现类

-ClassPathXmlApplicationContext

-配置文件在项目的类路径下 new ClassPathXmlApplicationContext("application-demo1.xml")

-FileSystemXmlApplicationContext

-配置文件在本地磁盘目录下 new FileSystemXmlApplicationContext("d:/application-demo1.xml")

-配置文件在和src同级的目录下 new FileSystemXmlApplicationContext("application-demo1.xml")

-分析BeanFactory

-使用XmlBeanFactory实例化，接收参数为Resource接口对象

-Resource接口 用于读取Spring配置文件

-2个实现子类

-ClassPathResource

-FileySystemResource

-获取容器：获取类路径下的配置文件

BeanFactory bf = new XmlBeanFactory(new ClassPathResource("application-demo1.xml"))

-获取容器：获取当前项目根目录下的配置文件

BeanFactory bf = new XmlBeanFactory(new FileySystemResource("application-demo1.xml"))

-ApplicationContext

-容器对象初始化之后，所有对象都装配好，程序调用直接从内存中取出即可，执行效率高，占用内存大

-BeanFactory

-调用getBean()的时候才正式装配对象

## bean元素详解

### ========== ba01 ==========

-getBean()

-从容器获取指定的bean实例，传入application.xml中定义的bean id

-首先调用bean类的无参构造

### ========== ba02 ==========

-动态工厂Bean

-定义工厂类

package beanAssemble.ba02;

public class ServiceFactory {

/\*\*

\* IUserService工厂类

\* @return

\*/

public IUserService getUserService() {

return new UserServiceImpl();

}

}

-xml配置

<!--

id是bean实例的唯一标识，程序通过id访问bean，bean之间的依赖关系通过id关联

class指定bean所属的类

-->

<bean id="factory" class="beanAssemble.ba02.ServiceFactory"></bean>

<!--

动态工厂实现

factory-bean 指定对应的工厂Bean

factory-method 指定创建对象所用的方法

-->

<bean id="userService" factory-bean="factory" factory-method="getUserService"></bean>

### ========== ba03 ==========

-静态工厂Bean

-无需工厂实例，工厂类使用static方法

-工厂类

package beanAssemble.ba03;

public class ServiceFactory {

/\*\*

\* IUserService静态工厂类

\* @return

\*/

public static IUserService getUserService() {

return new UserServiceImpl();

}

}

-xml配置

<!-- bean definitions here -->

<!--

id是bean实例的唯一标识，程序通过id访问bean，bean之间的依赖关系通过id关联

class指定bean所属的类

-->

<!--

静态工厂实现

factory-method 指定创建对象所用的方法

-->

<bean id="userService" class="beanAssemble.ba03.ServiceFactory" factory-method="getUserService"></bean>

### ========== ba04 ==========

-Bean的作用域(xml属性scope)，一共5个

-singleton 单例(默认) -容器创建时就装配好

-prototype 多例 -代码中使用到这个bean才进行装配

--以下3个是Web应用中使用Spring才有效

-request 每次请求产生一个

-session 每个不同HTTP session产生一个

-global session

-例子：

-xml配置

<!-- bean definitions here -->

<!--

id是bean实例的唯一标识，程序通过id访问bean，bean之间的依赖关系通过id关联

class指定bean所属的类

-->

<!--

scope bean作用域

-singleton 单例(默认) -容器创建时就装配好

-prototype 多例 -代码中使用到这个bean才进行装配

-->

<bean id="userService1" class="beanAssemble.ba04.UserServiceImpl" scope="singleton"></bean>

<bean id="userService2" class="beanAssemble.ba04.UserServiceImpl" scope="prototype"></bean>

### ========== ba05 ==========

-Bean后处理器(BeanPostProcessor类)

-Bean初始化，自动执行后处理类的2个方法，用于增强Bean中的方法

-在目标初始化完毕前，容器调用该方法

-public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException{}

-在目标初始化结束后，容器调用该方法

-public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException{}

-xml配置

<bean id="studentService" class="beanAssemble.ba05.StudentServiceImpl"></bean>

<bean id="teacherService" class="beanAssemble.ba05.TeacherServiceImpl"></bean>

<!-- Bean后处理器 -->

<bean id="beanProcessor" class="beanAssemble.ba05.MyBeanPostProcessor" />

### ========== ba06 ==========

-bean创建后行为，销毁前行为

-实体类

@Override

public void setUp() {

System.out.println("初始化完毕后的行为");

}

@Override

public void tearDown() {

System.out.println("销毁前的行为");

}

-xml配置

-destroy-method要能看到效果，2个条件

-单例

-容器关闭

<bean id="userService" class="beanAssemble.ba06.UserServiceImpl"

init-method="setUp" destroy-method="tearDown"></bean>

-测试程序

public void test() {

ApplicationContext ac = new ClassPathXmlApplicationContext("beanAssemble/application-ba06.xml");

// 从容器获取Service

IUserService userService = (IUserService) ac.getBean("userService");

userService.doSome();

userService.doOther();

// 关闭容器对象

((ClassPathXmlApplicationContext) ac).close();

}

### ========== ba07 ==========

-bean的生命周期

-1 调用无参构造，实例化

-2 调用参数的setter，为属性注入值

-3 (可选)如果bean实现了BeanNameAware接口，则会执行接口方法setBeanAname(String beanid),bean可以获取其在容器中的id

-4 (可选)如果bean实现了BeanFactoryAware接口，则会执行接口方法setBeanFactory(BeanFactory factory),Bean可以获取到BeanFactory对象

-5 (可选)如果定义了bean后处理器，执行 postProcessBeforeInitialization()

-6 (可选)如果bean实现了InitializingBean接口，则执行接口方法afterPropertiesSet(),这个方法在Bean的所有属性的set方法执行完毕后执行，是Bean初始化结束的标志(重要)

-7 (可选)如果设置了init-method方法，则执行

-8 (可选)如果定义了bean后处理器，执行 postProcessAfterInitialization()

-9 执行业务方法

-10 (可选)如果bean实现了DisposableBean接口，则执行接口方法destroy()

-11 (可选)如果设置了destroy-method方法，则执行

-测试程序输出：

getBean()首先调用的是无参构造

为name属性注入值

BeanNameAware接口的setBeanName()可以获得在XML中配置的Bean ID 是studentService

BeanFactoryAware接口的setBeanFactory()可以获得BeanFactory的对象是：org.springframework.beans.factory.support.DefaultListableBeanFactory@59ec2012: defining beans [beanProcessor,studentService,teacherService]; root of factory hierarchy

InitializingBean接口的afterPropertiesSet() 对象已经初始化完毕了！

init-method xml中的创建后行为

BeanPostProcessor接口的postProcessAfterInitialization()方法

BeanPostProcessor接口的postProcessAfterInitialization()方法

BeanPostProcessor接口的方法增强，下一步调用业务

StudentServiceImpldoSome() 真实业务

DisposableBean接口的destroy()

destroy-method xml的销毁前行为

# Spring IOC

## 基于XML注入

### ========== di01\_1 ==========

基于XML的DI（依赖注入）

-设值注入

-基本类型

-引用类型

实体类1：

public class Student {

private String name;

private Integer age;

private School school;

}

实体类2：

public class School {

private String schoolName;

}

xml配置：

<!-- 值类型注入 -->

<bean id="student" class="di01\_1.Student">

<property name="name" value="xiaoming"></property>

<property name="age" value="19"></property>

<!-- 引用类型注入，注入下方的xx大学 -->

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

</bean>

<bean id="school" class="di01\_1.School">

<property name="schoolName" value="xx大学"></property>

</bean>

测试类：

public class TestDemo {

@Test

public void test() {

ApplicationContext ac = new ClassPathXmlApplicationContext("di01\_1/application.xml");

Student student = (Student) ac.getBean("student");

System.out.println(student);

}

}

### ========== di01\_2 ==========

-构造注入

实体类1 Student增加构造方法：

//构造注入使用的构造方法

**public** Student(String name, Integer age, School school) {

**this**.name = name;

**this**.age = age;

**this**.school = school;

}

xml配置：

<bean id=*"school"* class=*"di01\_2.School"*>

<property name=*"schoolName"* value=*"xx大学"*></property>

</bean>

<!-- 构造注入 -->

<bean id=*"student2"* class=*"di01\_2.Student"*>

<!-- name属性: 构造函数的参数名 -->

<!-- index属性: 构造函数的参数索引 -->

<!-- type属性: 构造函数的参数类型-->

<!-- <constructor-arg name="name" index="0" type="java.lang.Integer" value="999" ></constructor-arg> -->

<constructor-arg name=*"name"* value=*"xiaoming"*></constructor-arg>

<constructor-arg name=*"age"* value=*"18"*></constructor-arg>

<constructor-arg name=*"school"* ref=*"school"*></constructor-arg>

</bean>

测试程序：

**public** **class** TestDemo {

@Test

**public** **void** test() {

ApplicationContext ac = **new** ClassPathXmlApplicationContext("di01\_2/application.xml");

Student student = (Student) ac.getBean("student2");

System.***out***.println(student);

}

}

### ========== di02\_1 ==========

-命名空间注入（了解即可）

p命名空间注入，采用值类型注入，需要有setter和无参构造

实体类1 Student

**package** di02\_1;

**public** **class** Student {

**private** String name;

**private** Integer age;

**private** School school;

**public** Student() {

}

//构造注入使用的构造方法

**public** Student(String name, Integer age, School school) {

**this**.name = name;

**this**.age = age;

**this**.school = school;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

@Override

**public** String toString() {

**return** "Student [name=" + name + ", age=" + age + ", school=" + school + "]";

}

**public** School getSchool() {

**return** school;

}

**public** **void** setSchool(School school) {

**this**.school = school;

}

}

xml配置

<!-- p命名空间注入，采用值类型注入，需要有setter和无参构造 -->

<bean id=*"school"* class=*"di02\_1.School"*>

<property name=*"schoolName"* value=*"xx大学"*></property>

</bean>

<bean id=*"student"* class=*"di02\_1.Student"*

p:name=*"xiaoming"* p:age=*"18"* p:school-ref=*"school"*

></bean>

### ========== di02\_2 ==========

c命名空间注入，采用构造注入，需要有构造器

xml配置：

<!-- c命名空间注入，采用构造注入，需要构造方法 -->

<bean id=*"school"* class=*"di02\_2.School"*>

<property name=*"schoolName"* value=*"xx大学"*></property>

</bean>

<bean id=*"student"* class=*"di02\_2.Student"*

c:name=*"xiaoming"* c:age=*"18"* c:school-ref=*"school"*

></bean>

### ========== di03 ==========

集合属性注入

包含这些集合

**private** String[] strs;

**private** List<Student> students;

**private** Set<String> mySet;

**private** Map<String,Integer> myMap;

**private** Properties myPro;

xml配置

<bean id=*"school"* class=*"di03.School"*>

<property name=*"schoolName"* value=*"xx大学"*></property>

</bean>

<bean id=*"student1"* class=*"di03.Student"*>

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

<property name=*"age"* value=*"18"*></property>

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

</bean>

<bean id=*"student2"* class=*"di03.Student"*>

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

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

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

</bean>

<!-- 集合注入 -->

<bean id=*"myCollections"* class=*"di03.MyCollections"*>

<!-- 数组注入 -->

<property name=*"strs"*>

<array>

<value>a</value>

<value>b</value>

<value>c</value>

</array>

</property>

<!-- List注入 -->

<property name=*"students"*>

<list>

<ref bean=*"student1"*></ref>

<ref bean=*"student2"*></ref>

</list>

</property>

<!-- set注入 -->

<property name=*"mySet"*>

<set>

<value>小学</value>

<value>中学</value>

<value>大学</value>

</set>

</property>

<!-- Map注入 -->

<property name=*"myMap"*>

<map>

<entry key=*"height"* value=*"180"*></entry>

<entry key=*"weight"* value=*"80"*></entry>

</map>

</property>

<!-- Properties注入 -->

<property name=*"myPro"*>

<props>

<prop key=*"tel"*>110</prop>

<prop key=*"address"*>guangzhou</prop>

</props>

</property>

</bean>

### ========== di04\_1 ==========

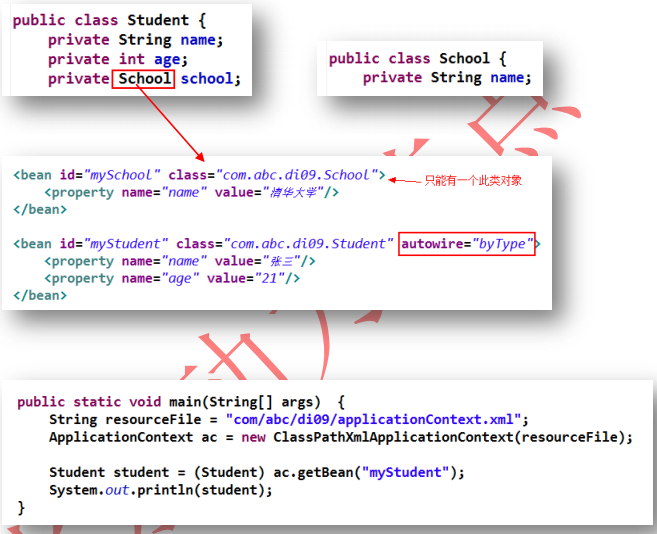
域属性自动注入

-byName：根据名称自动注入

对比id自动注入



-byType：根据类型自动注入



### ========== di05 ==========

使用SPEL注入

xml配置

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

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

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

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 definitions here -->

<!-- SPEL表达式 -->

<bean id=*"school"* class=*"di05.School"*>

<property name=*"schoolName"* value=*"xxUNIVERCITY"*></property>

</bean>

<bean id=*"person"* class=*"di05.Person"*>

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

<property name=*"age"* value=*"#{T(java.lang.Math).random()\*30}"* />

</bean>

<bean id=*"student"* class=*"di05.Student"*>

<!-- 引用person的属性 -->

<property name=*"stuName"* value=*"#{person.name}"* />

<property name=*"stuAge"* value=*"#{person.age}"* />

<!-- 引用school对象注入属性 -->

<property name=*"stuSchool"* value=*"#{school}"*></property>

<!-- 指定属性指定其方法 -->

<!-- <property name="schoolName" value="#{school.name.toLowerCase()}"></property> -->

</bean>

</beans>

### ========== di06 ==========

使用内部Bean注入

xml配置

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

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

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

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 definitions here -->

<!-- 使用内部Bean注入 -->

<bean id=*"student"* class=*"di06.Student"*>

<!-- 引用person的属性 -->

<property name=*"stuName"* value=*"xiaoming"* />

<property name=*"stuAge"* value=*"18"* />

<!-- 使用内部Bean注入 -->

<property name=*"stuSchool"*>

<bean class=*"di06.School"*>

<property name=*"schoolName"* value=*"xx小学"*></property>

</bean>

</property>

</bean>

</beans>

### ========== di07 ==========

同类抽象bean注入

多个bean实例属于一个类，实例的属性值有相同值，可以使用抽象bean

XML配置：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

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

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

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 definitions here -->

<!-- 使用同类抽象Bean注入 -->

<bean id=*"abase"* class=*"di07.ABase"* abstract=*"true"*>

<property name=*"department"* value=*"计算机系"* />

<property name=*"school"* value=*"xx大学"* />

</bean>

<bean id=*"student1"* class=*"di07.Student"* parent=*"abase"*>

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

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

</bean>

<bean id=*"student2"* class=*"di07.Student"* parent=*"abase"*>

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

<property name=*"score"* value=*"10"*></property>

</bean>

</beans>

### ========== di08==========

使用异类抽象bean注入

不同类对象具有相同的属性，且值一样，可以使用异类抽象bean注入

xml配置

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

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

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

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 definitions here -->

<!-- 使用同类抽象Bean注入 -->

<bean id=*"abase"* class=*"di08.ABase"* abstract=*"true"*>

<property name=*"department"* value=*"计算机系"* />

<property name=*"school"* value=*"xx大学"* />

</bean>

<bean id=*"student"* class=*"di08.Student"* parent=*"abase"*>

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

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

</bean>

<bean id=*"teacher"* class=*"di08.Teacher"* parent=*"abase"*>

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

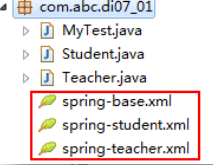
<property name=*"age"* value=*"30"*></property>

</bean>

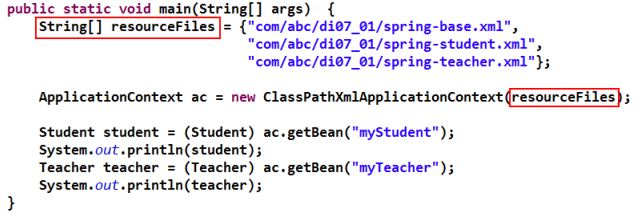
</beans>

### 多个Spring配置文件

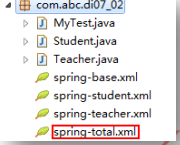
1. 平等关系的



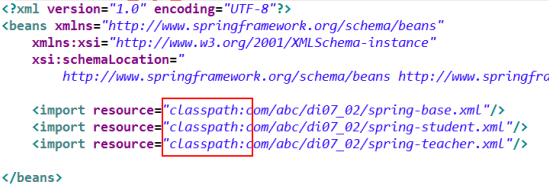
调用的时候传入一个数组



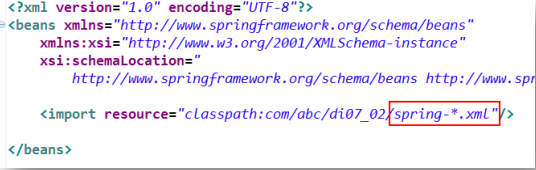
1. 包含关系



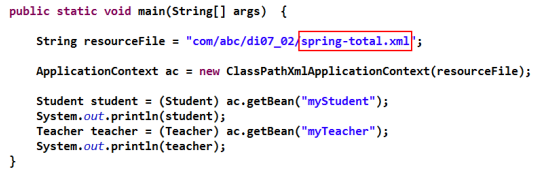
spring-total.xml配置



可以使用\*通配符（注意这里，不能命名为spring-total.xml，否则会递归调用导致死循环）



调用的时候直接传入汇总的即可



## 基于注解注入

### ==========adi01==========

环境搭建：

1. 导入AOP的jar包，注解的后台实现使用到了AOP编程



1. 修改xml约束头

<?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: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>

1. 配置组件扫描器，在指定的基本包中扫描注解

xml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: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 -->

<context:component-scan base-package=*"adi01"* />

</beans>

### ==========adi01==========

类上使用注解：

@Component(“student”)

下面这3个和Component是等效的，使用场景不同，方便扩展功能

@Repository --- 用于DAO实现类

@Service ---用于Service实现类

@Controller --- 用于Controller实现类

Bean的作用域

@Scope(“singleton”)

实体类Student配置

package adi01;

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

import javax.annotation.Resource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

//指定bean的id

@Component("student")

/\*

下面这3个和Component是等效的，使用场景不同，方便扩展功能

@Repository --- 用于DAO实现类

@Service ---用于Service实现类

@Controller --- 用于Controller实现类

\*/

//指定bean的作用范围

@Scope("singleton")

public class Student {

//基本类型注入，直接在属性上注入不安全，破坏了封装性，开发中不使用此种方式

//@value("xiaoming")

private String name;

private Integer age;

//引用类型注入，这里写在属性，开发中写在setter方法上

/\*

\* 按照类型注入域属性@Autowired

\* 如果有多个同样类型的，就无法注入

\* @Autowired(required=false) 表示匹配失败，设置为null

\*/

@Autowired

/\*

\* 按照类型注入域属性@Qualifier

\* 指定名称注入

\*/

@Qualifier("school")

/\*

\* 域属性注解@Resource

\* 1. @Resource 不带参数，表示按照School去匹配Bean

\* 2. @Resource(name="school")，查找name为school的属性注入

\*/

@Resource(name="school")

private School school;

@Override

public String toString() {

return "Student [name=" + name + ", age=" + age + "]";

}

public String getName() {

return name;

}

//基本类型注入，在setter上使用

@Value("xiaoming")

public void setName(String name) {

this.name = name;

}

public Integer getAge() {

return age;

}

//基本类型注入，在setter上使用

@Value("18")

public void setAge(Integer age) {

this.age = age;

}

//初始化方法和销毁方法

@PostConstruct

public void setUp() {

System.out.println("Bean初始化完成后执行");

}

@PreDestroy

public void tearDown() {

System.out.println("Bean对象销毁前执行");

}

}

实体类School类

package adi01;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;

@Component("school")

public class School {

private String schoolName;

@Value("xx小学")

public void setSchoolName(String schoolName) {

this.schoolName = schoolName;

}

}

### 使用JUnit4测试Spring

### ==========adi02==========

代替创建容器，和getBean方法

ApplicationContext ac = **new** ClassPathXmlApplicationContext("di08/application.xml");

Student student = (Student) ac.getBean("student");

导入jar包



定义被测试类

定义spring配置文件

定义测试类

//帮我们创建容器

@RunWith(SpringJUnit4ClassRunner.**class**)

//指定创建容器时使用哪个配置文件

@ContextConfiguration("classpath:adi02/application.xml")

## XML配置优先级高于注解，需要setter或者构造器支持

# Spring AOP

Spring的AOP实现

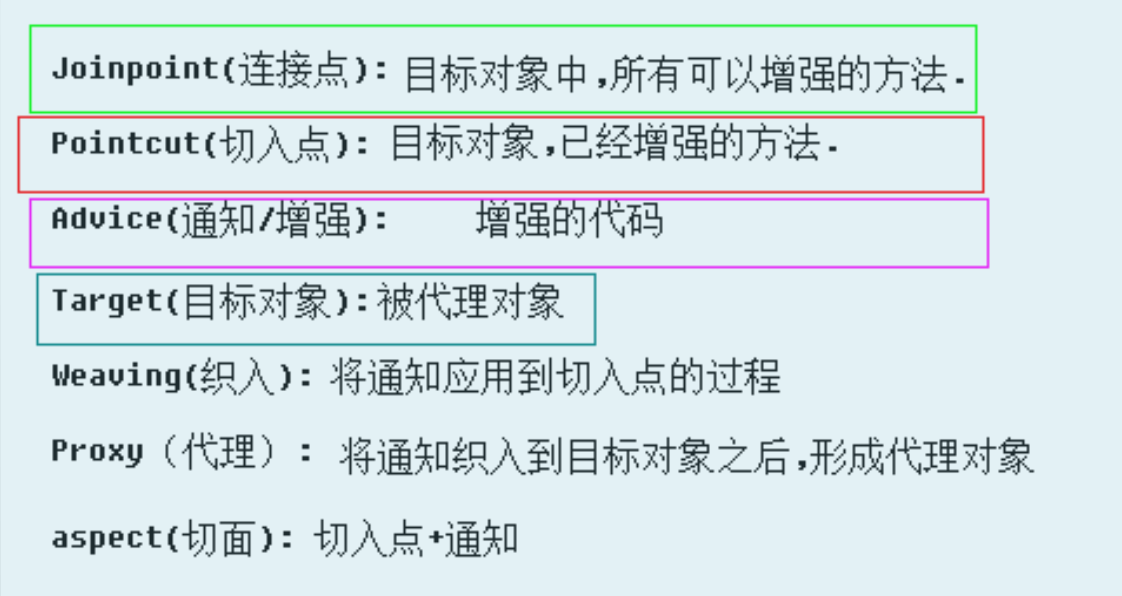
-JDK动态代理（接口）（优先采用）

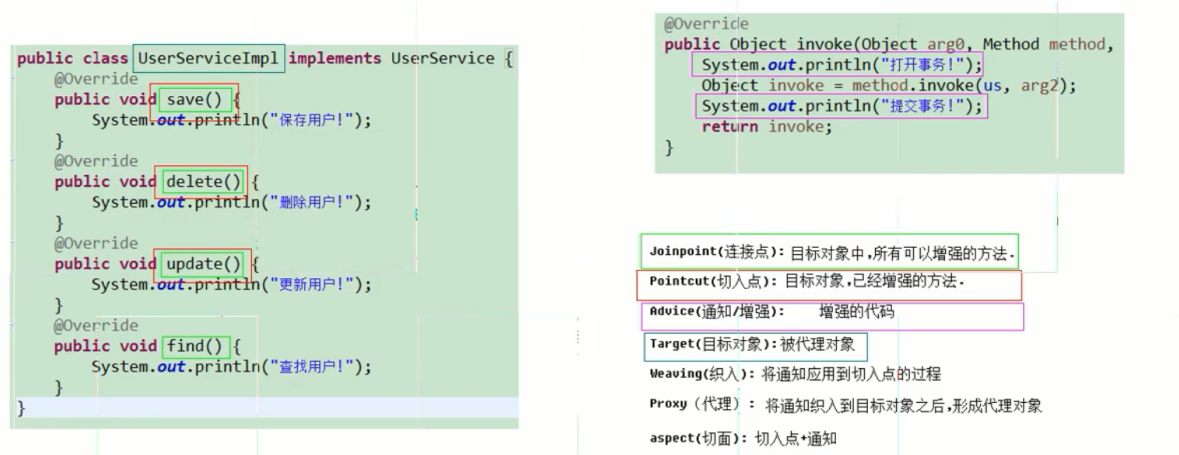
被代理对象必须要实现接口,才能产生代理对象.如果没有接口将不能使用动态代理技术

-cglib动态代理（任何类）

第三方代理技术,cglib代理.可以对任何类生成代理.代理的原理是对目标对象进行继承代理. 如果目标对象被final修饰.那么该类无法被cglib代理.

AOP编程术语：



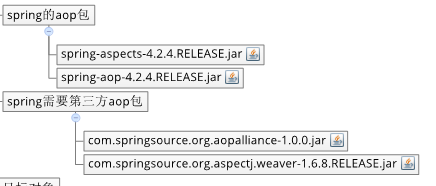


环境搭建

导入包：

E:\01\_JAVA\101\_software&doc\15\_spring\spring-framework-3.0.2.RELEASE-dependencies\org.aopalliance\com.springsource.org.aopalliance\1.0.0\com.springsource.org.aopalliance-1.0.0.jar

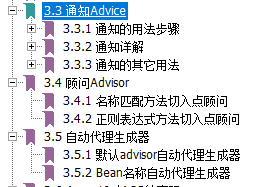
E:\01\_JAVA\101\_software&doc\15\_spring\spring-framework-3.0.2.RELEASE-dependencies\org.aspectj\com.springsource.org.aspectj.weaver\1.6.8.RELEASE\com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar



xml配置用最基本的头部即可

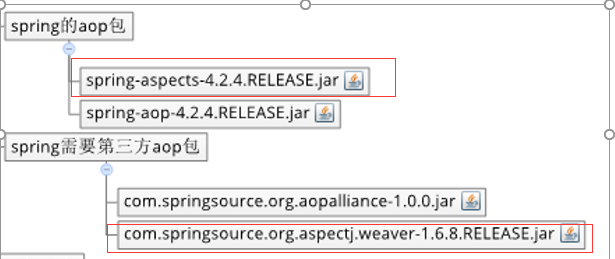
=====================准备通知=====================

跳过下面部分（开发中使用AspectJ基于XML和注解的AOP实现）



## AspectJ基于XML和注解的AOP实现

导入包：



导入约束



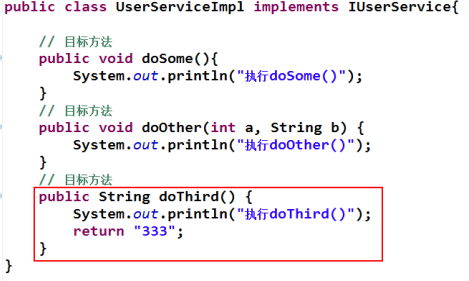
## Aspect的XML实现

### ==========aop01==========

步骤

1. 定义业务接口和实现类

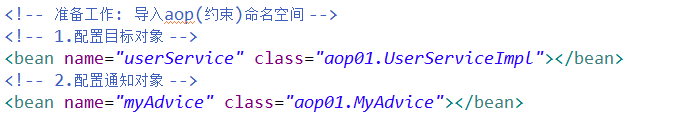




1. 定义切面POJO类（通知类），若干普通方法



1. 注册目标对象和通知类



1. 定义AOP配置

XML配置

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

xmlns=*"http://www.springframework.org/schema/beans"* 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-4.2.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.2.xsd "*>

<!-- 准备工作: 导入aop(约束)命名空间 -->

<!-- 1.配置目标对象 -->

<bean name=*"userService"* class=*"aop01.UserServiceImpl"*></bean>

<!-- 2.配置通知对象 -->

<bean name=*"myAdvice"* class=*"aop01.MyAdvice"*></bean>

<!-- 3.配置将通知织入目标对象 -->

<aop:config>

<!-- 配置切入点

这几种写法都是一样的

public void aop01.UserServiceImpl.save()

void aop01.UserServiceImpl.save()

\* aop01.UserServiceImpl.save()

\* aop01.UserServiceImpl.\*()

\* aop01.\*ServiceImpl.\*(..) 开发中使用这个，代表返回值为任意，aop01包下以及子包下的以包含ServiceImpl.java的任意方法，参数任意

\* aop01..\*ServiceImpl.\*(..)

-->

<aop:pointcut expression=*"execution(\* aop01.\*ServiceImpl.\*(..))"*

id=*"pc"* />

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

<!-- 指定名为before方法作为前置通知 -->

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

<!-- 后置 -->

<aop:after-returning method=*"afterReturning"*

pointcut-ref=*"pc"* />

<!-- 环绕通知 -->

<aop:around method=*"around"* pointcut-ref=*"pc"* />

<!-- 异常拦截通知 -->

<aop:after-throwing method=*"afterException"*

pointcut-ref=*"pc"* />

<!-- 后置 -->

<aop:after method=*"after"* pointcut-ref=*"pc"* />

</aop:aspect>

</aop:config>

</beans>

## Aspect的注解实现

### ==========aop02==========

步骤

1. 定义业务接口和实现类
2. 定义通知类

通知类加上@Aspect注解

@Aspect

**public** **class** MyAdvice {

1. 通知类的普通方法上加通知注解

//前置通知

@Before("execution(\* aop01.\*ServiceImpl.\*(..))")

**public** **void** before(){

System.***out***.println("这是前置通知!!");

}

//后置通知

@AfterReturning("execution(\* aop01.\*ServiceImpl.\*(..))")

**public** **void** afterReturning(){

System.***out***.println("这是后置通知(如果出现异常不会调用)!!");

}

//环绕通知

@Around("execution(\* aop01.\*ServiceImpl.\*(..))")

**public** Object around(ProceedingJoinPoint pjp) **throws** Throwable {

System.***out***.println("这是环绕通知之前的部分!!");

Object proceed = pjp.proceed();//调用目标方法

System.***out***.println("这是环绕通知之后的部分!!");

**return** proceed;

}

//异常通知

@AfterThrowing("execution(\* aop01.\*ServiceImpl.\*(..))")

**public** **void** afterException(){

System.***out***.println("出事啦!出现异常了!!");

}

//后置通知

@After("execution(\* aop01.\*ServiceImpl.\*(..))")

**public** **void** after(){

System.***out***.println("这是后置通知(出现异常也会调用)!!");

}

1. 注册目标对象和通知类

<!-- 1.配置目标对象 -->

<bean name=*"userService"* class=*"aop01.UserServiceImpl"*></bean>

<!-- 2.配置通知对象 -->

<bean name=*"myAdvice"* class=*"aop01.MyAdvice"*></bean>

1. 注册AspectJ自动代理

<!-- 准备工作: 导入aop(约束)命名空间 -->

<!-- 1.配置目标对象 -->

<bean name=*"userService"* class=*"aop01.UserServiceImpl"*></bean>

<!-- 2.配置通知对象 -->

<bean name=*"myAdvice"* class=*"aop01.MyAdvice"*></bean>

<!-- 3.开启使用注解完成织入 -->

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

# Spring+DAO

包括JDBC模板 --- IOC的应用

Spring的事务管理 --- AOP的应用

## JDBC

### dao\_jdbcTemplate

导包

JDBC jar包



事务jar包



mysql驱动包，C3P0的jar包

演示用例用C3P0

1. 定义实体类User，包含ID/用户名/年龄，生成构造，getter/setter和toString
2. 定义数据库和表，bjtest/t\_user
3. 定义IUserDAO

**package** dao\_jdbcTemplate;

**import** java.util.List;

**public** **interface** IUserDao {

//增加

**void** insert(User user);

//删除

**void** delete(Integer id);

//修改

**void** update(User user);

//根据ID查询用户名

String selectUsernameByID(Integer id);

//根据年龄查询用户名

List<String> selectUsernameByAge(Integer age);

//根据ID查询用户记录

User selectUserById(Integer id);

//查询所有记录

List<User> selectAllUsers();

}

1. 定义UserDaoImpl，继承IUserDAO，方法暂时不覆写
2. 定义IUserService

**package** dao\_jdbcTemplate;

**import** java.util.List;

**public** **interface** IUserService {

//增加用户

**void** add(User user);

//删除用户

**void** remove(Integer id);

// 修改用户

**void** modify(User user);

// 根据ID查找用户名

String findUsernameById(Integer id);

// 根据年龄查找用户名

List<String> findUsernameByAge(Integer age);

// 根据ID查找用户记录

User findUserById(Integer id);

// 返回所有用户信息

List<User> findAllUser();

}

1. 定义UserServiceImpl

**package** dao\_jdbcTemplate;

**import** java.util.List;

**public** **class** UserServiceImpl **implements** IUserService {

**private** IUserDao dao;

**public** IUserDao getDao() {

**return** dao;

}

**public** **void** setDao(IUserDao dao) {

**this**.dao = dao;

}

@Override

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

dao.insert(user);

}

@Override

**public** **void** remove(Integer id) {

dao.delete(id);

}

@Override

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

dao.update(user);

}

@Override

**public** String findUsernameById(Integer id) {

**return** dao.selectUsernameByID(id);

}

@Override

**public** List<String> findUsernameByAge(Integer age) {

**return** dao.selectUsernameByAge(age);

}

@Override

**public** User findUserById(Integer id) {

**return** dao.selectUserById(id);

}

@Override

**public** List<User> findAllUser() {

**return** dao.selectAllUsers();

}

}

1. 导入数据源（3选一）

-Spring默认数据源

-DriverManagerDataSource

<!-- 配置Spring默认数据源 -->

<bean id=*"myDataSource"* class=*"org.springframework.jdbc.datasource.DriverMangerDataSource"*>

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

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

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

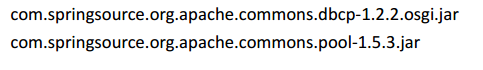
<property name=*"password"* value=*"123456"*></property>

</bean>

-DBCP数据源

-BasicDataSource

需要导入2个包



<!-- 配置DBCP数据源 -->

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

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

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

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

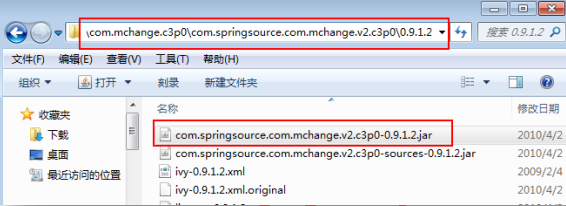
<property name=*"password"* value=*"123456"*></property>

</bean>

-C3P0数据源

-ComboPooledDataSource

需要导入1个包



<!-- 配置C3P0数据源 -->

<bean id=*"myC3P0DataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

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

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

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

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

</bean>

1. 从jdbc.properties读取数据库连接信息

放在src目录下

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/crm?characterEncoding=utf-8

jdbc.username=root

jdbc.password=123456

1. 修改XML

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd "*>

<!-- 指定spring读取db.properties配置 -->

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

<bean id=*"myC3P0DataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

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

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

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

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

</bean>

</beans>

1. 配置JDBC模板

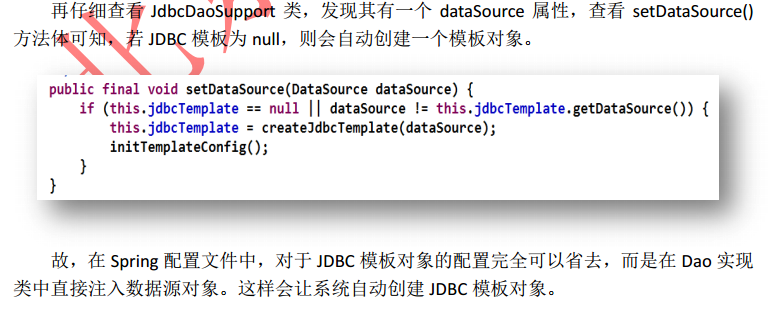
<!-- 2.将JDBCTemplate放入spring容器 -->

<bean name=*"myJdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"* >

<property name=*"dataSource"* ref=*"myC3P0DataSource"* ></property>

</bean>

1. DAO实现类继承JdbcDaoSupport类，可以使用setJdbcTemplate()方法，有JDBC模板属性



最终XML的配置

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd "*>

<!-- 指定spring读取db.properties配置 -->

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

<!-- 配置Spring默认数据源 -->

<!--

<bean id="myDataSource"

class="org.springframework.jdbc.datasource.DriverMangerDataSource">

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

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

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

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

</bean> -->

<!-- 配置DBCP数据源 -->

<!--

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

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

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

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

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

</bean> -->

<!-- 1.配置C3P0数据源 -->

<bean id=*"myC3P0DataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

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

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

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

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

</bean>

<!-- 2.将JDBCTemplate放入spring容器 -->

<bean name=*"myJdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"* >

<property name=*"dataSource"* ref=*"myC3P0DataSource"* ></property>

</bean>

<!-- 3.将UserDao放入spring容器 -->

<bean name=*"userDao"* class=*"dao\_jdbcTemplate.UserDaoImpl"* >

<!-- 会自动创建模板对象，因此不需要手动创建模板 -->

<!-- <property name="jdbcTemplate" ref="myJdbcTemplate" /> -->

<!-- 直接注入数据源，系统会自动创建JDBC模板对象 -->

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

</bean>

</beans>

### JDBC模板

DAO实现类使用继承自JdbcDaoSupport的getTemplate方法，获取JDBC模板对象

增删改，通过update方向实现

public int update(String sql)

public int update(String sql,Object... args)

package dao\_jdbcTemplate;

import org.junit.Before;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestDemo {

private IUserService service;

@Before

public void setUp() {

ApplicationContext ac = new ClassPathXmlApplicationContext("dao\_jdbcTemplate/applicationContext.xml");

service = (IUserService) ac.getBean("service");

}

@Test

public void testInsert() {

User user = new User("xiaoxiao",24);

service.add(user);

}

@Test

public void testDelete() {

service.remove(4);

}

@Test

public void testUpdate() {

User user = new User("xiaoming",24);

service.add(user);

user.setId(5);

user.setAge(100);

service.modify(user);

}

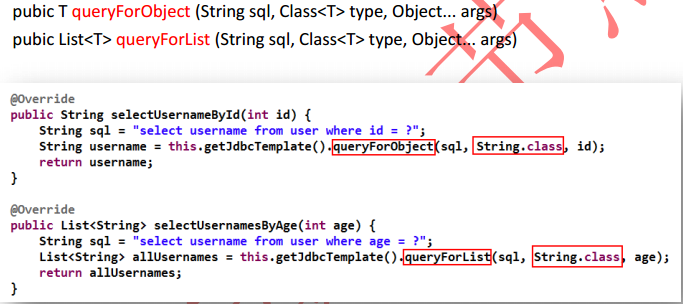
}

查询操作：

-简单对象查询

-queryForObject()

-queryForList()



-自定义对象查询

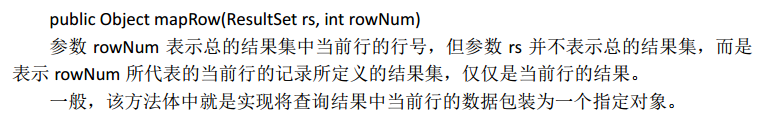
-queryForObject()

-query ()



RowMapper为记录映射接口，用于将查询结果的每一条记录包装到指定对象

这个接口有一个方法要实现



测试代码太多，不贴

JDBC模板对象是多例的，每次使用都要getJdbcTemplate()方法获取

## Spring AOP事务

事务提升到service层

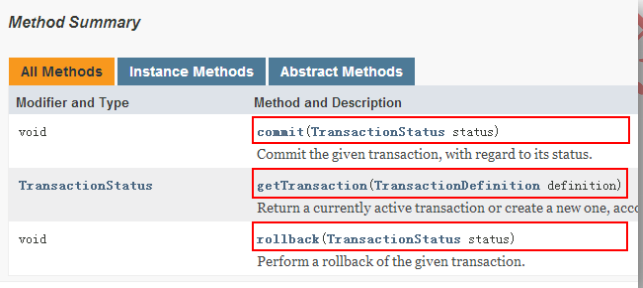
Spring有3种方式

-事务代理工厂管理事务

-事务注解管理事务

-AspectJ的AOP配置管理事务

事务管理器接口PlatformTransactionManager



常用的2个实现类

-DataSourceTransactionManager：使用JDBC或者iBatis进行持久化数据使用

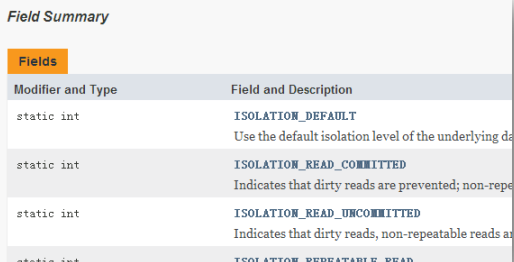
-HibernateTransactionManager：使用Hibernate进行持久化数据时使用

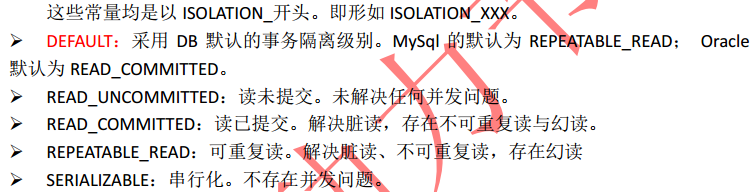
Spring的回滚方式：发生运行时异常回滚，发生受查异常（编译时异常）时提交（可以手动指定）

事务定义接口：TransactionDefinition

有3类常量：

-事务隔离级别（5个）

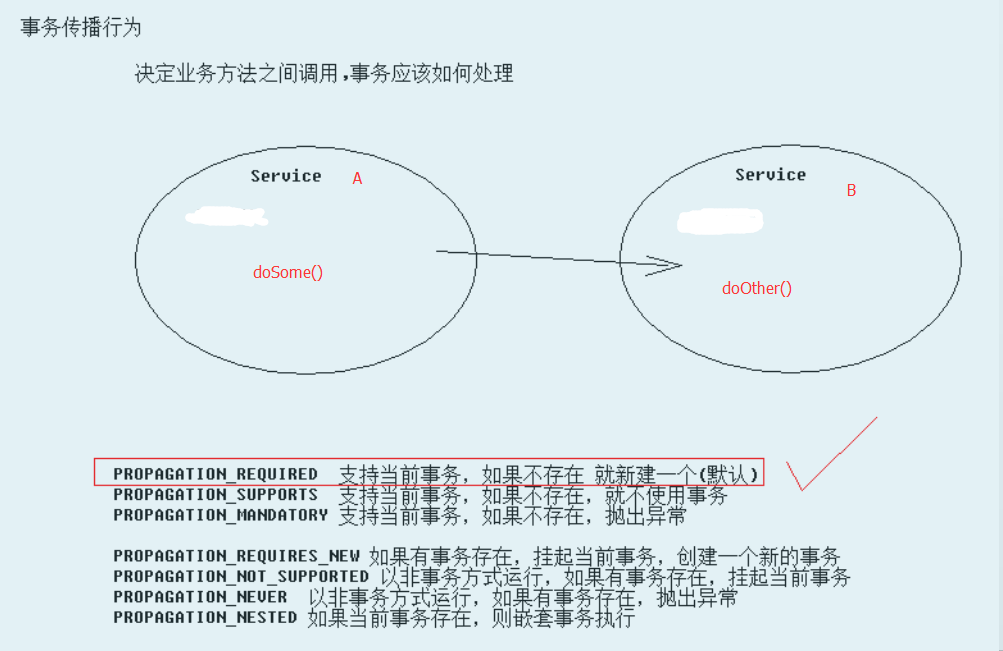


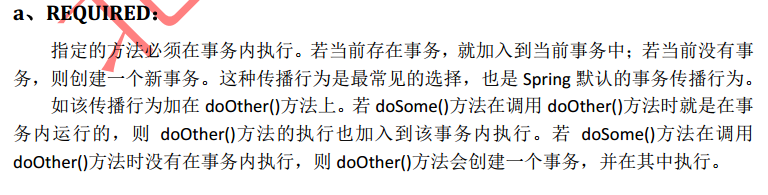


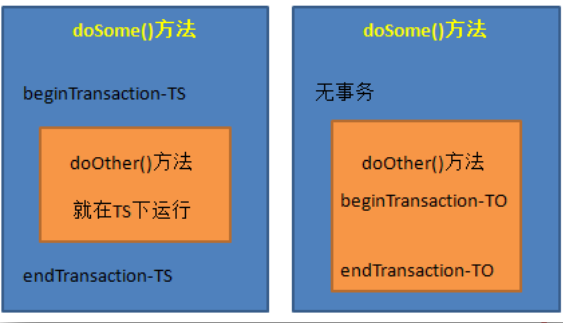
-事务传播行为（7个）

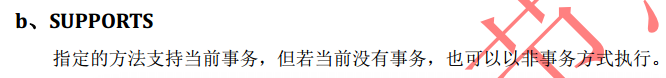
加在方法上的

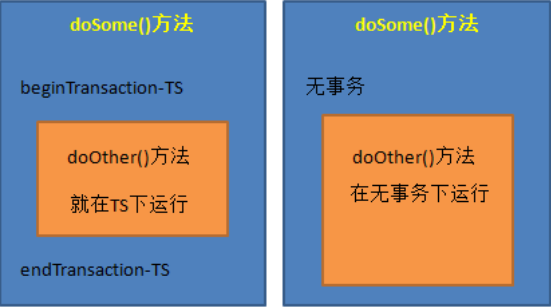
不同事务间的方法调用

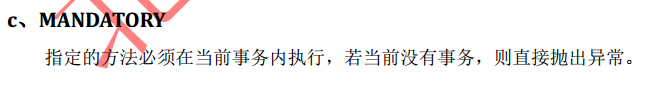


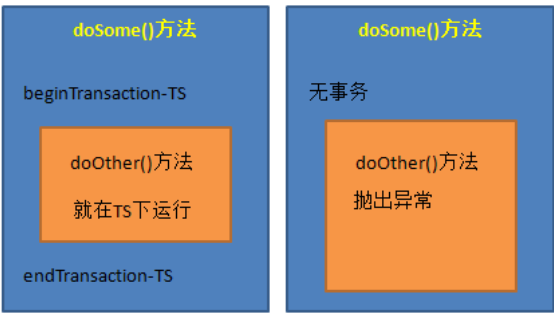


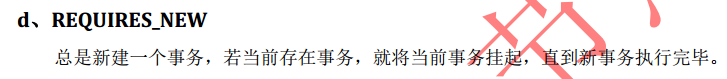


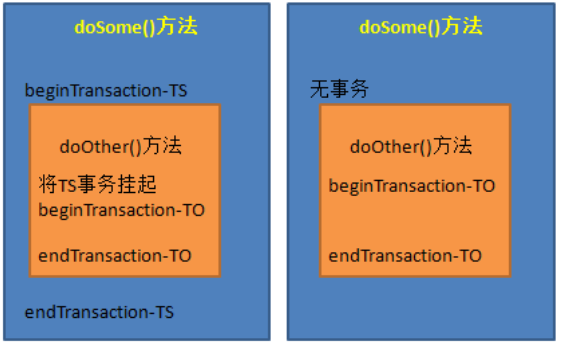


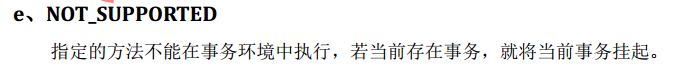


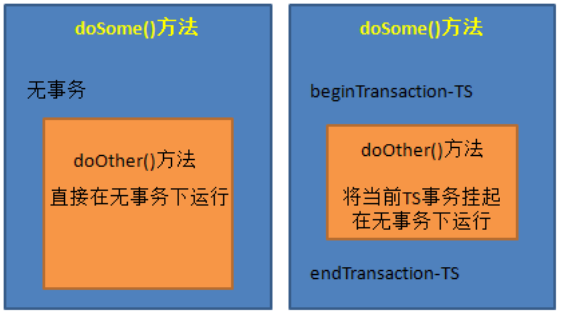


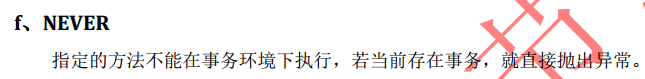


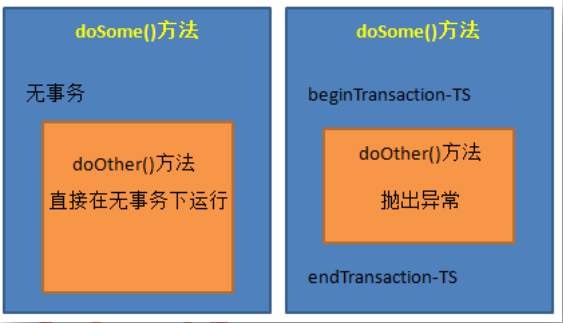


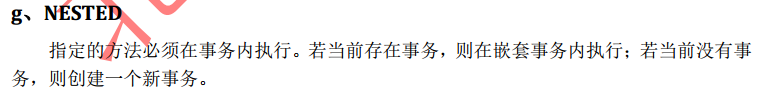


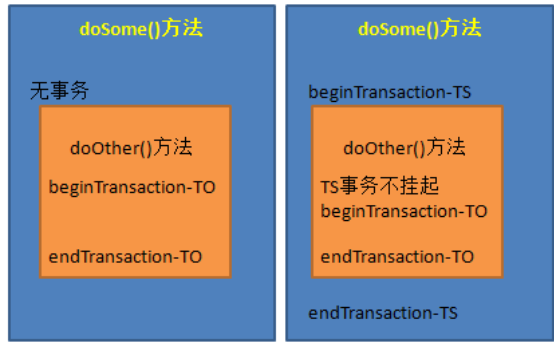








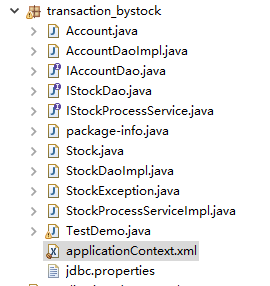




-默认事务超时时限TIMEOUT\_DEFAULT（使用默认值即可）

### transaction\_bystock

代码比较多，不贴



管理事务的3种方式实现

### Spring代理工厂（不推荐，硬编码）

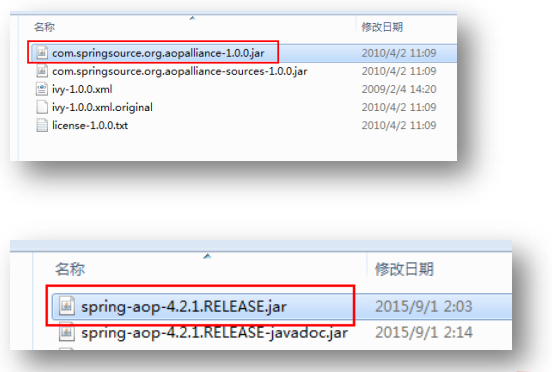
#### transaction\_proxy

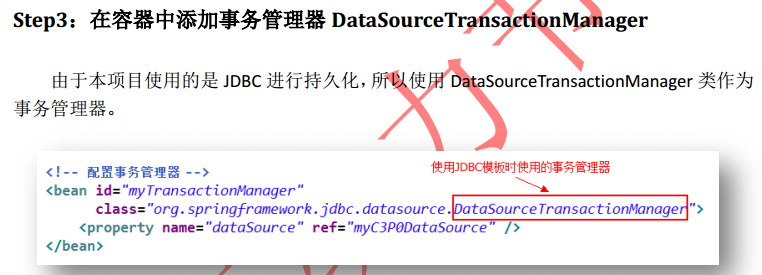
为目标类，也就是service类创建事务代理

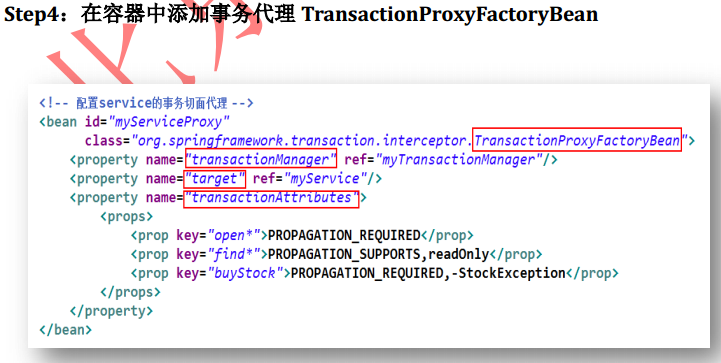
使用到TransactionProxyFactoryBean类，初始化需要3个属性

1. transactionManager 事务管理器
2. target 目标对象，service实现类对象
3. transactionAttributes 事务属性设置

需要2个包





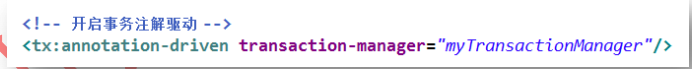


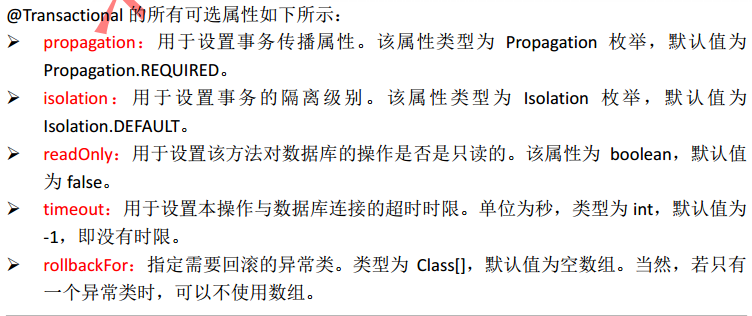


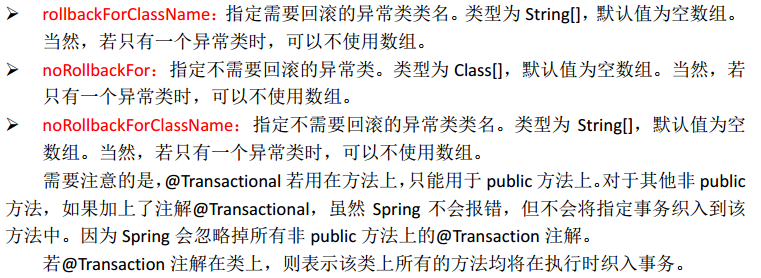
代码测试通过

### Spring注解配置AOP事务

@Transactional注解方式，把事务织入到对应方法中







代码测试通过。

## 使用AspectJ的AOP配置管理事务（重点）

### transaction\_advisor

代码测试通过