# 框架FrameWork

**内参保密不得外传**

**达内大数据方向培优阶段**

**陈子枢 主讲**

**QQ: 1920-9520-78**

**http://big.tedu.cn**

**2017**

# 第一天：管理对象IoC+管理依赖DI

## 初识Spring框架

### 时代变迁

原始时代我们用一个jsp搞定一切，但如此开发大型项目时我们遇到问题，前端美化的代码和后端的代码交织，代码中又有html、js、css样式，又有业务逻辑和数据库访问代码，杂乱不清晰，美工和开发打架。

于是mvc分层架构封建时代出现，把我们写代码的地方硬性分成3个地方，Model层封装数据，View视图层页面展现，Controller控制层访问转发。代码之间的耦合度降低。概念有了，需要实际干活的。于是随着mvc分层概念的深入人心，业界涌现出很多实现框架，最著名的莫过于struts1和struts2。随着前端框架的成熟，后端框架也应运而生如：dbutils、jdbcTemplate、hibernate、ibatis、mybatis。

一个前端WEB层框架有了，一个后端数据库层访问框架有了，那中间呢？谁来胜任？spring石破天惊，横空出世。

### Spring的野心

了解了历史，有个问题值得我们去深思？spring到底想干什么？

它想把全球最好的技术组合到一起为企业提供高质量的企业级的应用程序框架，减轻开发者开发的难度，减少重复的代码。

目标很宏大，那如何下手呢？如果是你你会怎么实现？第一步要干什么呢？

我们拿经典的框架来举例子。

struts2作为WEB框架深受企业爱戴，它会自己管理action，来创建其实例，这样在程序中就可以访问action的资源。hibernate作为持久层优秀的框架它也自己管理持久对象。可以看到，各个诸侯都自己管理对象，而要想让它们对象复用，那真是繁琐。前面就有失败者WebService，为了管理不同的开发语言的对象而层层包装转换，辛苦制定的规则，还借着J2EE规范之名，也推广不开。

如何破局呢？要想发号施令其它人听，最好的解决办法就是扼住他们的咽喉。在java的世界里最重要的无疑就是对象的生命周期管理。于是spring以此为切入点，实现自己的统治。宣布所有对象由我来管理，struts2你不再管理对象，由我来管理，你要用从我这拿。hibernate你也不再管理对象，由我来管理，你要用从我这拿。你说管就管吗？这两个征战数年战功赫赫的大将军会听一个初出茅庐乳臭未干野小子的话？他们当然不会听，spring的话可以不听，但他们都要听开发者的。开发一个完整的系统有四个核心，WEB层支持、业务逻辑层、持久层支持、事务支持。而这就是它们的软肋，这就是它们的命门所在，它们只能完成一部分工作，不是一个整体解决方案。而spring并没有抹杀它们，而是依然高官厚禄，承认它们的市场地位，还赠与一个事务管理。一边打压一边拉拢，它们两位看看大势已去，只能俯首称臣。于是兵不血刃，一场变革悄然兴起，一个经典的三层框架诞生SSH （Strut2+Spring+Hibernate）。

故事很传奇，听的人很开心。可spring真就这么简单吗？如果这样想，你就大错特错了。例如：spring怎么来实现对象的管辖？怎么让不同技术之间能简单的互相配合？这才是spring的决胜之处。

为实现这些spring创新的形成了一套新的理论体系，可谓前无古人后无来者。其中最核心的是：IoC控制反转、DI依赖注入、Bean工厂、SpringAOP面向切面编程、事务控制。

并且spring并没有停止不前，这只是统治地球的第一步，随着spring占领市场后，开始对有功之臣进行清洗，struts2不再优秀，致命bug层出不穷，最终被springmvc替代。hibernate想用jdbcTemplate和jdo替代，却被mybatis超越，目前还未统一。世界又达到新的平衡，经典的新三大框架诞生，SSM（SpringMVC+Spring+MyBatis）。目前新的趋势又在蔓延springboot+springcloud已经开始逐鹿中原，大有一统江湖之势！

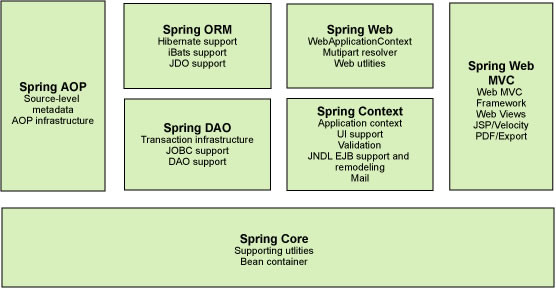
### 框架组成



Spring官网： http://spring.io

Spring是一个开源框架，是为了解决企业应用程序开发复杂性而创建的。Spring框架的不光是技术牛，而是它的核心思想更牛，它不重复发明轮子，而是把业界做的最好的技术黏合起来形成一个强大的企业级的应用框架。

Spring 框架是一个分层架构，由7个定义良好的模块组成。Spring 模块构建在核心容器之上，核心容器定义了创建、配置和管理 bean 的方式，如下图所示：



组成 Spring 框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。每个模块的功能如下：

|  |  |  |
| --- | --- | --- |
|  | **模块** | **说明** |
|  | 核心容器Spring Core | 核心容器提供Spring框架的基本功能。核心容器的主要组件是BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转（IOC）模式，将应用程序的配置和依赖性规范与实际的应用程序代码分开。 |
|  | Spring上下文  Spring Context | Spring上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。 |
|  | Spring AOP | 通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。可以很容易地使 Spring框架管理的任何对象支持AOP。Spring AOP模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。 |
|  | Spring DAO | JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。 |
|  | Spring ORM | Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括JDO、Hibernate和iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。 |
|  | Spring Web | Web上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以Spring 框架支持与 Jakarta Struts的集成。Web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。 |
|  | Spring MVC框架 | MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。 |

Spring 框架的功能可以用在任何J2EE服务器中，大多数功能也适用于不受管理的环境。Spring 的核心要点是：支持不绑定到特定J2EE服务的可重用业务和数据访问对象。毫无疑问，这样的对象可以在不同J2EE环境（Web或EJB）、独立应用程序、测试环境之间重用。

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

### 核心概念

|  |  |  |
| --- | --- | --- |
|  | **模块** | **说明** |
|  | BeanFactory | Spring内部使用，创建bean的工厂 |
|  | ApplicationContext | 外部应用程序调用，也成为spring容器 |
|  | IoC控制反转  Inversion of Control | 开发者在无需自己new对象，无需关心对象的创建过程  User user = new User(); 手动创建对象  User user = context.getBean(user); 容器创建对象 |
|  | DI依赖注入  Dependency Injection | 松耦合方式实现对象直接的依赖 |
|  | AOP面向切面编程 | 补充java面向对象的不足 |

### 主要jar组成

|  |  |  |
| --- | --- | --- |
|  | **模块** | **说明** |
|  | org.springframework.core | 核心工具包，其他包依赖此包 |
|  | org.springframework.beans | 核心，包括：配置文件，创建和管理bean等 |
|  | org.springframework.aop | 面向切面编程，提供AOP的实现 |
|  | org.springframework.context | 提供IoC功能上的扩展服务，此外还提供许多企业级服务的支持，邮件、任务调度、JNDI定位、EJB集成、远程访问、缓存以及多种视图层框架的支持 |
|  | org.springframework.web.mvc | 包含SpringMVC应用开发时所需的核心类 |
|  | org.springframework.transaction | 为JDBC、Hibernate、JDO、JPA提供一致的声明式和编程式事务管理 |
|  | org.springframework.web | 包含Web应用开发时所需支持类 |
|  | org.springframework.aspects | 提供对AspectJ框架的支持 |
|  | org.springframework.test | 对junit等测试框架的简单封装 |
|  | org.springframework.asm | 3.0后提供自己独立的 |
|  | org.springframework.context.support | Context的扩展支持，用于mvc方面 |
|  | org.springframework.expression | Spring表达式语言 |
|  | org.springframework.instument | 对服务器的代理接口 |
|  | org.springframework.jdbc | 对jdbc的简单封装 |
|  | org.springframework.jms | 为简化jms api的使用而做的简单封装 |
|  | org.springframework.orm | 整合第三方orm，如hibernate/mybatis |
|  | org.springframework.web.servlet | 增强servlet |

### 第三方框架集成

|  |  |  |
| --- | --- | --- |
|  | **框架** | **说明** |
|  | 权限 ApacheShiro | java的安全框架，认证、授权、加密、会话管理、缓存 |
|  | 缓存 Ehcache | java进程内缓存框架，快速、精干，Hibernate默认缓存 |
|  | 缓存 redis | 基于C语言编写、支持分布式内存数据库、kv数据库 |
|  | 持久层框架Hibernate | 对jdbc轻量级封装，将pojo和数据库建立映射关系，全ORM映射框架 |
|  | 持久层框架Mybatis | 支持SQL级别，半ORM映射框架 |
|  | 定时任务Quartz | 可以快速完成任务调度的工作 |
|  | 校验框架Hibernate | 用来验证bean的字段，基于注解，方便快速高效 |

## IoC控制反转创建对象例子

Inversion of Control 控制反转方式来实现对象的创建。

### 创建java工程

创建lib目录

导入7个jar包

commons-logging-1.0.4.jar

spring-beans-3.2.3.RELEASE.jar

spring-context-3.2.3.RELEASE.jar

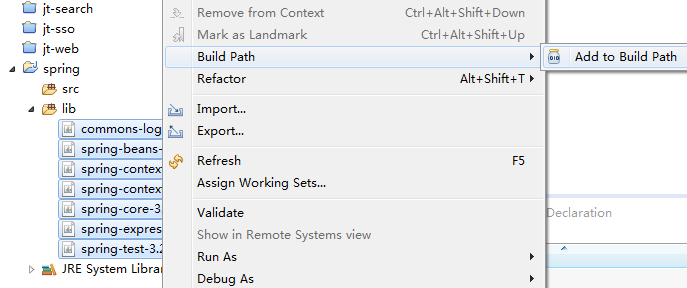
spring-context-support-3.2.3.RELEASE.jar

spring-core-3.2.3.RELEASE.jar

spring-expression-3.2.3.RELEASE.jar

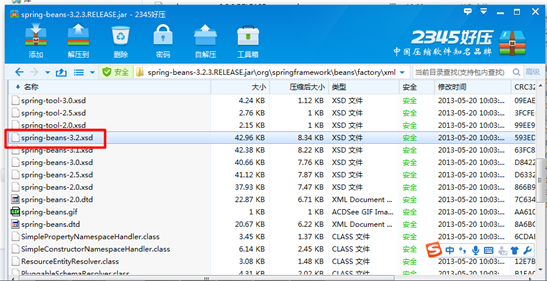
spring-test-3.2.3.RELEASE.jar

Build Path

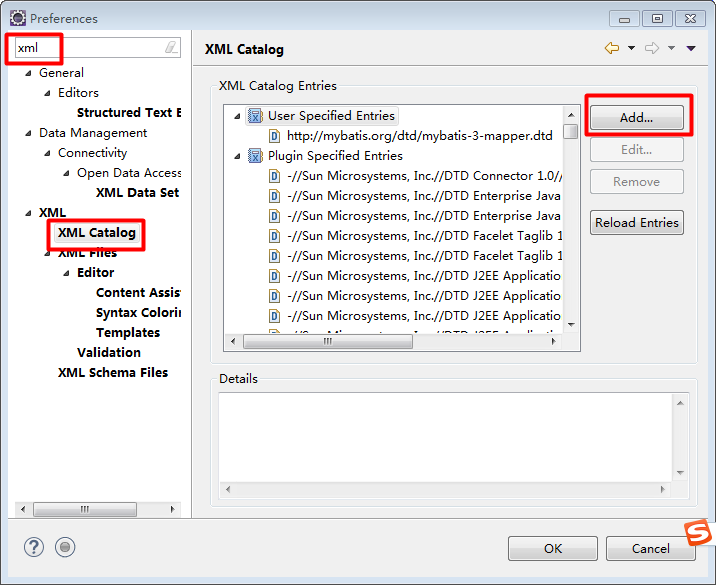


### 导入约束自动提示

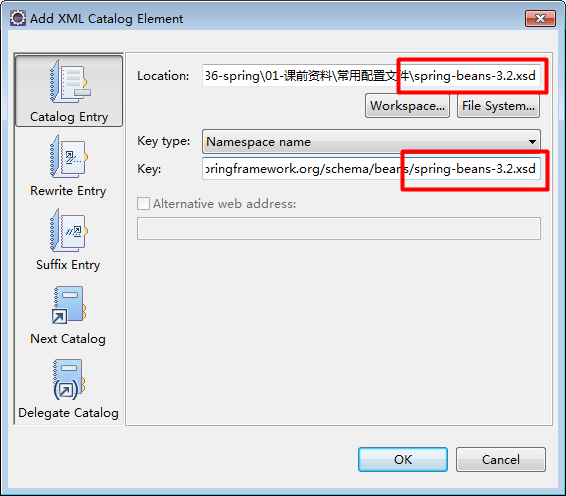
找到jar包中的xsd文件



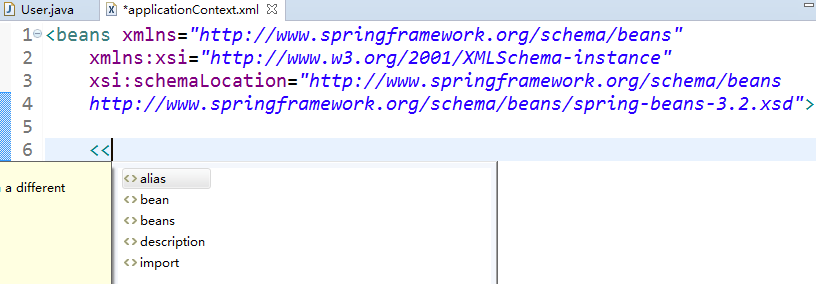
自定义xsd提示



浏览xsd文件，复制拼接到下面的链接中



关闭xml文件，再次打开，提示出现



### 创建User对象

package hello;

public class User {

public void sayHi(){

System.out.println("hi");

}

}

### spring配置文件中配置userbean

<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-3.2.xsd">

<!--

bean spring管理的类都称为bean

id 唯一标识，不能重复，驼峰规则首字母小写

class 完整的类路径

-->

<bean id="user" class="hello.User"/>

</beans>

### 测试类

package hello;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("hello/applicationContext.xml");

User user = (User) context.getBean("user");

user.sayHi();

}

}

### 小结

这就是spring框架的IoC，控制反转。之前我们自己new出新类。new User();变成由一个初始化的xml配置文件来创建，也就是由spring容器来创建。遍历xml配置文件，读取到<bean>，获取到class属性的类的全路径，利用反射创建这个类。

在java范畴中万物皆Object,zai Spring中万物皆Bean。Bean是Spring的核心、基础、根源。

## 创建过程说明

### 加载过程

首先加载applicationContext.xml，然后遍历里面的<bean>标签，读取class属性找到类，创建对象实例，当getBean获取对象，强转后保存到user对象中，这样用户就可以调用。

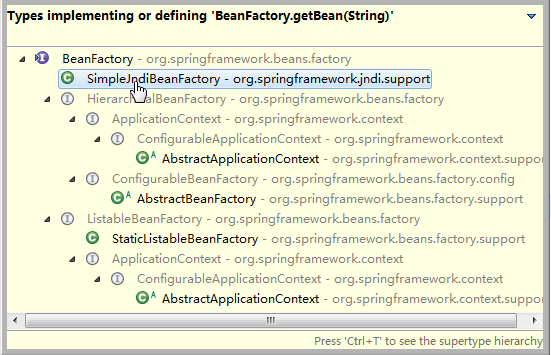
### spring容器内部如何存这些对象呢？

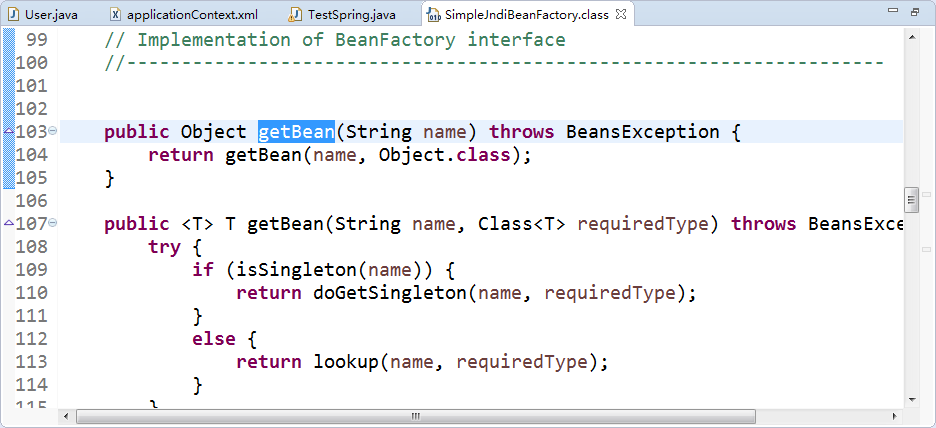
根据bead的id属性getBean("user")去获取这个对象，

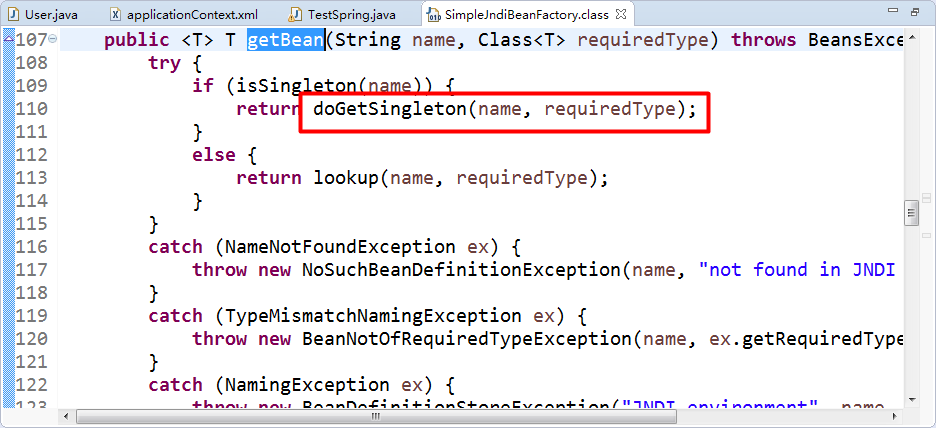
spring初始化时把bean放入到一个map中维护，map<id,Object>

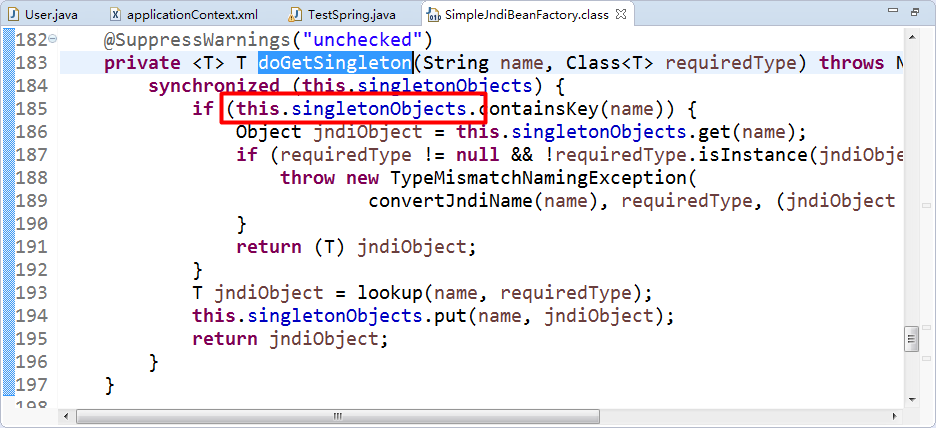
### getBean源码剖析

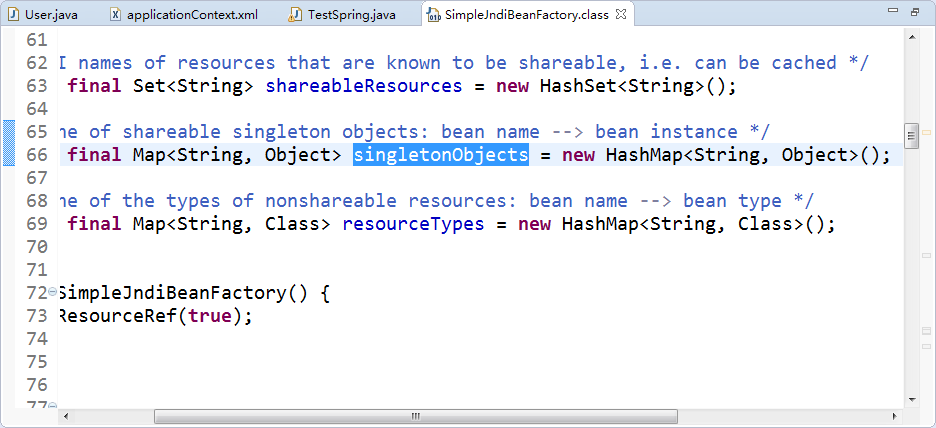
跟踪getBean方法，找其实现类，最终可以看到bean被存放在Map中。











### 多实例

#### applicationContext.xml

<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-3.2.xsd">

<!--

bean spring管理的类都称为bean

id 唯一标识，不能重复，驼峰规则首字母小写

class 完整的类路径

-->

<bean id="user" class="hello.User"/>

<bean id="usera" class="hello.User"/>

</beans>

#### TestSpring.java

package hello;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("hello/applicationContext.xml");

User user = (User) context.getBean("user");

user.sayHi();

System.out.println(user);

User usera = (User) context.getBean("usera");

usera.sayHi();

System.out.println(usera);

}

}

#### 执行结果

hi

hello.User@3f130f01

hi

[hello.User@7c3e2b00](mailto:hello.User@7c3e2b00)

可以看到仅加行<bean>就又创建一个实例，而且默认spring创建的bean是单例。

## 创建对象的4种方式

### 默认构造

增加默认构造

package hello;

public class User {

public User() {

System.out.println("我是默认无参构造");

}

public void sayHi(){

System.out.println("hi");

}

}

执行结果

我是默认无参构造

hi

hello.User@7c3e2b00

spring底层是通过反射来创建对象，那如果是反射，就需要无参构造。可我们没有加，怎么可以呢？java编译器在编译时，如果发现类没有构造方法会自动添加构造方法。

修改代码

package hello;

public class User {

public User(String name) {

System.out.println("我是默认无参构造");

}

public void sayHi(){

System.out.println("hi");

}

}

我们手动添加带参构造，执行后结果报错。

Caused by: org.springframework.beans.BeanInstantiationException: Could not instantiate bean class [hello.User]: No default constructor found; nested exception is java.lang.NoSuchMethodException: hello.User.<init>()

at org.springframework.beans.factory.support.SimpleInstantiationStrategy.instantiate(SimpleInstantiationStrategy.java:83)

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.instantiateBean(AbstractAutowireCapableBeanFactory.java:1000)

... 36 more

Caused by: java.lang.NoSuchMethodException: hello.User.<init>()

at java.lang.Class.getConstructor0(Class.java:2892)

at java.lang.Class.getDeclaredConstructor(Class.java:2058)

at org.springframework.beans.factory.support.SimpleInstantiationStrategy.instantiate(SimpleInstantiationStrategy.java:78)

说明：如果以后开发过程中，编写了构造方法，那么切记要先加默认构造方法！

### 静态工厂设计模式

抽象类怎么实例化？可以通过静态工厂设计模式（也成为简单工厂模式）。

#### StaticFactory.java

package staticfactory;

import java.util.Calendar;

public class StaticFactory {

public static Calendar getCalendar(){

return Calendar.getInstance();

}

}

这直接就可以获取对象实例，那如果要从容器中获取对象，怎么做？

#### applicationContext.xml

<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-3.2.xsd">

<bean id="cal" class="staticfactory.StaticFactory" factory-method="getCalendar"/>

</beans>

#### 测试类

package staticfactory;

import java.util.Calendar;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("staticfactory/applicationContext.xml");

Calendar calendar = (Calendar) context.getBean("cal");

System.out.println(calendar.getTime());

}

}

#### 问题：上面调用静态方法有没创建类？

StaticFactory.java

package staticfactory;

import java.util.Calendar;

public class StaticFactory {

public StaticFactory() {

System.out.println("我是一个静态工厂类");

}

public static Calendar getCalendar(){

return Calendar.getInstance();

}

}

打印结果可以看到，没有打印出默认构造中的日志，说明静态方法不需要创建类。

#### 问题：去掉static方法，那还能创建对象吗？

当然不行，提示你需要一个静态方法

org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'cal' defined in class path resource [staticfactory/applicationContext.xml]: No matching factory method found: factory method 'getCalendar()'. Check that a method with the specified name exists and that it is static.

没有静态方法就无法直接调用。

什么时候使用工厂？有些类不能直接实例化，就不能直接通过new关键字来创建，类似这样的对象我们要通过静态工厂模式来创建对象。

### 实例工厂设计模式

#### MethodFactory.java

package methodfactory;

import java.util.Calendar;

public class MethodFactory {

public MethodFactory() {

System.out.println("我是方法工厂默认构造");

}

public Calendar getCalendar(){

return Calendar.getInstance();

}

}

#### applicationContext.xml

<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-3.2.xsd">

<!-- 先创建工程对象，再调用工程对象的方法 -->

<bean id="methodFacotry" class="methodfactory.MethodFactory"/>

<bean id="cal" factory-bean="methodFacotry" factory-method="getCalendar"/>

</beans>

#### 测试类

package methodfactory;

import java.util.Calendar;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("methodfactory/applicationContext.xml");

Calendar calendar = (Calendar) context.getBean("cal");

System.out.println(calendar.getTime());

}

}

两个模式的差异：

静态工厂设计模式不会创建类对象，直接调用静态方法。而工厂方法设计模式会创建类，然后通过xml中配置去访问其方法。

如果日常开发，很少用工厂设计模式，如果要用到这两种方式，你的类的抽象层次已经比较高了。一般这些都是架构师在构建架构时会使用。但学完你至少要看得懂。

### Spring工厂

这是spring框架自身提供的，它需要实现FactoryBean接口，实现代码就必须写在getObject()方法中。

#### SpringFactory.java

package springfactory;

import java.util.Calendar;

import org.springframework.beans.factory.FactoryBean;

public class SpringFactory implements FactoryBean<Calendar>{

public SpringFactory() {

System.out.println("我是一个spring工厂类");

}

public Calendar getObject() throws Exception {

return Calendar.getInstance();

}

public Class<?> getObjectType() {

return Calendar.class;

}

public boolean isSingleton() {

return false;

}

}

#### applicationContext.xml

<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-3.2.xsd">

<bean id="cal" class="springfactory.SpringFactory"/>

</beans>

#### 测试类

package springfactory;

import java.util.Calendar;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("springfactory/applicationContext.xml");

Calendar calendar = (Calendar) context.getBean("cal");

System.out.println(calendar.getTime());

}

}

那用spring工厂有什么好处呢？实现一个接口就可以了，简单方便。

## Spring创建对象是单例还是多例

什么是单例多例？

* 单例就是在整个spring容器中只有一个对象
* 多例就是在内存中有多个对象

### 单例

配置文件

<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-3.2.xsd">

<!--

bean spring管理的类都称为bean

id 唯一标识，不能重复，驼峰规则首字母小写

class 完整的类路径

-->

<bean id="user" class="hello.User"/>

</beans>

测试类

package hello;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("hello/applicationContext.xml");

User user = (User) context.getBean("user");

user.sayHi();

System.out.println(user);

User user1 = (User) context.getBean("user");

user1.sayHi();

System.out.println(user1);

}

}

执行结果

我是默认无参构造

hi

hello.User@55794cb9

hi

hello.User@55794cb9

我们从spring容器中获取对象，实例化成两个对象，两个对象地址相同说明对象是单例的。

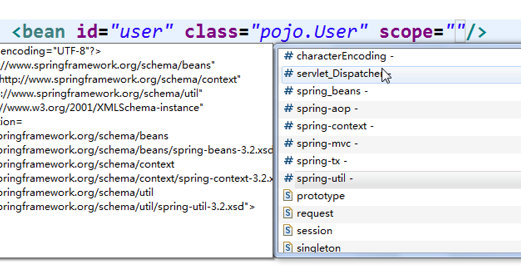
为什么是一样的呢？

之前我们看过最终spring把所有的bean都放在一个map中，getBean("user")，这个user就是map中的key。那key相同，你说它对象返回是不是一个呢？当然是一个对象了。

总结：默认条件下，spring中的bean创建的对象是单例。

单例对象有好有坏：好处，spring创建后，那就不会再频繁创建，缓存在map中。

### 多例



默认scope=singleton 单例，可以选择prototype 多例（原型）

<bean id="user" class="hello.User" scope="prototype"/>

执行结果

我是默认无参构造

hi

hello.User@26bfdd7d

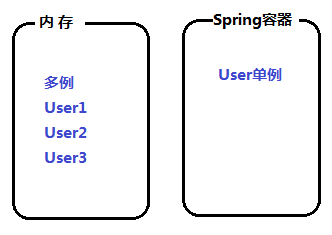
我是默认无参构造

hi

hello.User@7e664d6a

可以看到对象地址不同，说明是多例。无参构造也执行了两次，说明对象创建了两个。

### 单例多例的存储



默认情况下，spring中创建的对象都是单例，并且维护其生命周期。单例对象的生命周期与spring容器共命运，同生共死。

但如果对象是多例的，那么Spring容器只负责对象的创建，不负责维护其生命周期，也就是说如果容器关闭，对象并未销毁，需要用户自行关闭。

### 懒加载

#### 单例+懒加载

<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-3.2.xsd">

<!--

bean spring管理的类都称为bean

id 唯一标识，不能重复，驼峰规则首字母小写

class 完整的类路径

-->

<bean id="user1" class="hello.User" scope="singleton" lazy-init="true"/>

<bean id="user2" class="hello.User" scope="singleton" lazy-init="true"/>

<bean id="user3" class="hello.User" scope="singleton" lazy-init="true"/>

<bean id="user4" class="hello.User" scope="singleton" lazy-init="true"/>

<bean id="user5" class="hello.User" scope="singleton" lazy-init="true"/>

<bean id="user6" class="hello.User" scope="singleton" lazy-init="true"/>

<bean id="user7" class="hello.User" scope="singleton" lazy-init="true"/>

<bean id="user9" class="hello.User" scope="singleton" lazy-init="true"/>

<bean id="user0" class="hello.User" scope="singleton" lazy-init="true"/>

<bean id="user" class="hello.User" scope="singleton" lazy-init="true"/>

</beans>

如果我们的<bean>很多会有什么问题呢？spring启动时是不是越来越慢啊？那一个大型项目中上万个类，启动不就太漫长了，那怎么解决呢?

一开始我们没调用bean所以即使spring创建了对象，没有业务调用也是多余的。那怎么做好呢？能否调用时再创建？

spring也提供了这种方式：懒加载

#### 单例+非懒加载

单例默认都是直接创建对象

<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-3.2.xsd">

<!--

bean spring管理的类都称为bean

id 唯一标识，不能重复，驼峰规则首字母小写

class 完整的类路径

-->

<bean id="user1" class="hello.User" scope="singleton" lazy-init="false"/>

<bean id="user2" class="hello.User" scope="singleton" lazy-init="false"/>

<bean id="user3" class="hello.User" scope="singleton" lazy-init="false"/>

<bean id="user4" class="hello.User" scope="singleton" lazy-init="false"/>

<bean id="user5" class="hello.User" scope="singleton" lazy-init="false"/>

<bean id="user6" class="hello.User" scope="singleton" lazy-init="false"/>

<bean id="user7" class="hello.User" scope="singleton" lazy-init="false"/>

<bean id="user9" class="hello.User" scope="singleton" lazy-init="false"/>

<bean id="user0" class="hello.User" scope="singleton" lazy-init="false"/>

<bean id="user" class="hello.User" scope="singleton" lazy-init="false"/>

</beans>

#### 全局配置

<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-3.2.xsd"

default-lazy-init="true">

<bean id="user1" class="hello.User" scope="singleton" lazy-init="default"/>

#### 多例

<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-3.2.xsd"

default-lazy-init="true">

<bean id="user1" class="hello.User" scope="prototype" lazy-init="default"/>

可以看到，无论是开启还是关闭懒加载配置，都是一样的结果，懒加载，怎样会这样呢？

多例对象都是懒加载。那为何Spring这样设计呢？这样没有用到时就不会创建对象，就不会占用内存空间。

## 对象的初始化和销毁

Spring会创建声明的对象，那只会创建对象吗？创建完对象还需要做些什么呢？需要对对象进行初始化，那有初始化当然就有销毁，那它是如何实现的呢？

### 初始化

#### User.java

增加初始化方法init和消化方法destory

package hello;

public class User {

public User() {

System.out.println("我是默认无参构造");

}

public void init(){

System.out.println("我是对象初始化方法");

}

public void sayHi(){

System.out.println("hi");

}

public void destroy(){

System.out.println("我是对象摧毁方法");

}

}

#### applicationContext.xml

<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-3.2.xsd

">

<bean id="user" class="hello.User" init-method="init"/>

</beans>

#### 测试类

package hello;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("hello/applicationContext.xml");

User user = (User) context.getBean("user");

user.sayHi();

System.out.println(user);

}

}

#### 执行结果

我是默认无参构造

我是对象初始化方法

hi

hello.User@404a12b8

### 销毁

#### 单例

applicationContext.xml

<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-3.2.xsd

">

<bean id="user" class="hello.User" init-method="init" destroy-method="destroy"/>

</beans>

怎么没有调用销毁方法呢？现在对象是单例的，受spring容器管理，spring容器没有关闭，那这个对象会销毁吗？当然不会，所有我们看不到销毁方法。

package hello;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ClassPathXmlApplicationContext context =

new ClassPathXmlApplicationContext("hello/applicationContext.xml");

User user = (User) context.getBean("user");

user.sayHi();

System.out.println(user);

//ApplicationContext接口方法少

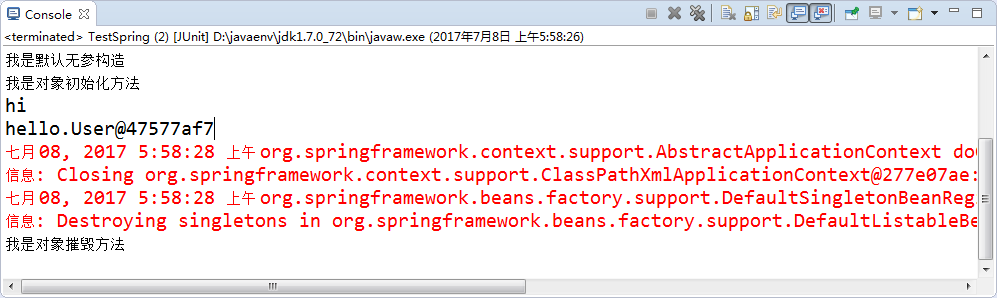
//换成ClassPathXmlApplicationContext对象，才有close方法，关闭容器

context.close();

}

}

执行结果：



当spring容器关闭时触发销毁方法。

#### 多例

修改成多例的：

<bean id="user" class="hello.User"

scope="prototype"

init-method="init" destroy-method="destroy"/>

可以看到，创建的多例对象，spring就不管其销毁了。验证了我们之前提到的spring对单例和多例处理方式的不同。

## DI依赖注入属性

### 为什么需要DI依赖注入？

IoC反转控制只完成了单个对象的创建，那对象初始化如何实现呢，特别对关联对象的初始化？

例如：User对象需要Dept对象属性。

这就需要配合DI依赖注入来实现。

### 两种注入方式

对象由spring创建，只后再由spring给属性赋值

spring提供两种方式设置属性值：

* set方法注入
* 构造方法注入（spring源码大都使用这种方式）

## Set方法属性注入

### 单值属性

#### User.java

package hello;

public class User {

private String name;

private Integer age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public String toString() {

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

}

}

#### applicationContext.xml

<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-3.2.xsd

">

<bean id="user" class="hello.User">

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

</bean>

</beans>

注意：ref和value不能同时存在

#### 执行结果

User [name=tony, age=null]

### 对象属性

#### Dog.java

package hello;

public class Dog {

public void eat(){

System.out.println("主人我饿了");

}

}

#### User.java

package hello;

public class User {

private Dog dog;

private String name;

private Integer age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Dog getDog() {

return dog;

}

public void setDog(Dog dog) {

this.dog = dog;

}

@Override

public String toString() {

return "User [dog=" + dog + ", name=" + name + ", age=" + age + "]";

}

}

#### applicationContext.xml

<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-3.2.xsd

">

<bean id="user" class="hello.User">

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

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

</bean>

<bean id="dog" class="hello.Dog"/>

</beans>

#### 测试类

package hello;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("hello/applicationContext.xml");

User user = (User) context.getBean("user");

System.out.println(user);

Dog dog = user.getDog();

dog.eat();

}

}

这有什么意义呢？今天我能注入一只狗，明天我就能注入DAO，后台我就能注入Service，这就解决了关联对象的注入。这个方式解决了对象直接的关联问题，从而奠基了spring的地位。

### 集合属性

支持List、Set、Map、Property

#### User.java

package hello;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.Set;

public class User {

private Dog dog;

private String name;

private Integer age;

private List list;

private Set set;

private Map map;

private Properties dbParams;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Dog getDog() {

return dog;

}

public void setDog(Dog dog) {

this.dog = dog;

}

public List getList() {

return list;

}

public void setList(List list) {

this.list = list;

}

public Set getSet() {

return set;

}

public void setSet(Set set) {

this.set = set;

}

public Map getMap() {

return map;

}

public void setMap(Map map) {

this.map = map;

}

public Properties getDbParams() {

return dbParams;

}

public void setDbParams(Properties dbParams) {

this.dbParams = dbParams;

}

@Override

public String toString() {

return "User [dog=" + dog + ", name=" + name + ", age=" + age + ", list=" + list + ", set=" + set + ", map="

+ map + ", dbParams=" + dbParams + "]";

}

}

#### applicationContext.xml

<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-3.2.xsd

">

<bean id="user" class="hello.User">

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

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

<property name="list">

<list>

<value>严嵩</value>

<value>徐阶</value>

<value>高供</value>

<value>张居正</value>

<ref bean="dog"/>

</list>

</property>

<property name="set">

<set>

<value>神童</value>

<value>编撰</value>

<value>首辅</value>

</set>

</property>

<property name="map">

<map>

<entry key="历史朝代" value="明朝"/>

<entry key="皇帝" value="万历"/>

</map>

</property>

<property name="dbParams">

<props>

<prop key="username">root</prop>

<prop key="password">root</prop>

</props>

</property>

</bean>

<bean id="dog" class="hello.Dog"/>

</beans>

#### 执行结果

User [dog=hello.Dog@545e1b34, name=tony, age=null, list=[严嵩, 徐阶, 高供, 张居正, hello.Dog@545e1b34], set=[神童, 编撰, 首辅], map={历史朝代=明朝, 皇帝=万历}, dbParams={password=root, username=root}]

主人我饿了

## 构造方法属性注入

### User.java

package constructor;

public class User {

private String name;

private Integer age;

public User(String name, Integer age){

this.name = name;

this.age = age;

}

@Override

public String toString() {

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

}

}

### applicationContext.xml

<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-3.2.xsd

">

<bean id="user" class="constructor.User">

<constructor-arg index="0" value="tina"/>

<constructor-arg index="1" value="18"/>

</bean>

</beans>

### 测试类

package constructor;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("constructor/applicationContext.xml");

User user = (User) context.getBean("user");

System.out.println(user);

}

}

## 常见异常

### xml路径找不到

java工程默认是到bin目录下找

org.springframework.beans.factory.BeanDefinitionStoreException: IOException parsing XML document from class path resource [applicationContext.xml]; nested exception is java.io.FileNotFoundException: class path resource [applicationContext.xml] cannot be opened because it does not exist

at org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:341)

at org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:302)

at org.springframework.beans.factory.support.AbstractBeanDefinitionReader.loadBeanDefinitions(AbstractBeanDefinitionReader.java:174)

at org.springframework.beans.factory.support.AbstractBeanDefinitionReader.loadBeanDefinitions(AbstractBeanDefinitionReader.java:209)

at org.springframework.beans.factory.support.AbstractBeanDefinitionReader.loadBeanDefinitions(AbstractBeanDefinitionReader.java:180)

at org.springframework.beans.factory.support.AbstractBeanDefinitionReader.loadBeanDefinitions(AbstractBeanDefinitionReader.java:243)

at org.springframework.context.support.AbstractXmlApplicationContext.loadBeanDefinitions(AbstractXmlApplicationContext.java:127)

at org.springframework.context.support.AbstractXmlApplicationContext.loadBeanDefinitions(AbstractXmlApplicationContext.java:93)

at org.springframework.context.support.AbstractRefreshableApplicationContext.refreshBeanFactory(AbstractRefreshableApplicationContext.java:130)

at org.springframework.context.support.AbstractApplicationContext.obtainFreshBeanFactory(AbstractApplicationContext.java:537)

at org.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicationContext.java:451)

at org.springframework.context.support.ClassPathXmlApplicationContext.<init>(ClassPathXmlApplicationContext.java:139)

at org.springframework.context.support.ClassPathXmlApplicationContext.<init>(ClassPathXmlApplicationContext.java:83)

at hello.TestSpring.test(TestSpring.java:12)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:606)

at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)

at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)

at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)

at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)

at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)

at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78)

at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57)

at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)

at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)

at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)

at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)

at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)

at org.junit.runners.ParentRunner.run(ParentRunner.java:363)

at org.eclipse.jdt.internal.junit4.runner.JUnit4TestReference.run(JUnit4TestReference.java:86)

at org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:38)

at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:459)

at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:675)

at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.java:382)

at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.java:192)

Caused by: java.io.FileNotFoundException: class path resource [applicationContext.xml] cannot be opened because it does not exist

at org.springframework.core.io.ClassPathResource.getInputStream(ClassPathResource.java:157)

at org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:328)

... 36 more

## 继承和抽象

### 继承

spring的继承和java的继承不同。多个dao对象，每个都需要数据源，数据代码就都重复出现，如何能少写点代码呢？

#### DataSource.java

package parent;

public class DataSource {

private String conn;

public String getConn() {

return conn;

}

public void setConn(String conn) {

this.conn = conn;

}

}

#### BaseDao.java

package parent;

public class BaseDao {

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

public DataSource getDataSource() {

return dataSource;

}

}

#### CartDao.java

package parent;

public class CartDao extends BaseDao{

public void add(){

System.out.println(super.getDataSource().getConn());

}

}

#### OrderDao.java

package parent;

public class OrderDao extends BaseDao{

public void add(){

System.out.println(super.getDataSource().getConn());

}

}

#### applicationContext.xml

<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-3.2.xsd

">

<bean id="dataSource" class="parent.DataSource">

<property name="conn" value="jdbc:mysql://127.0.0.1:3306/jtdb"/>

</bean>

<bean id="baseDao" class="parent.BaseDao">

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

</bean>

<bean id="cartDao" class="parent.CartDao"/>

</beans>

#### 测试类

package parent;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("parent/applicationContext.xml");

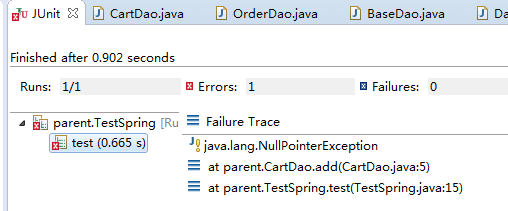
CartDao cartDao = (CartDao) context.getBean("cartDao");

cartDao.add();

}

}

#### 上面执行完成会报null指针



为何呢？

BaseDao baseDao = (BaseDao) context.getBean("baseDao");

System.out.println(baseDao);

执行结果

[parent.BaseDao@4921bae7](mailto:parent.BaseDao@4921bae7)

baseDao对象创建了啊？

cartDao、orderDao不都继承了baseDao吗？baseDao都可以获取到，怎么报nullpoint呢？大家想想cartDao会实例化，它的父类spring会去绑定关系吗？对，不会，为何不会呢？大家想如果某个类有父类我就绑定关系，那这个类也有父类呢？如果继承层级很多，那不是初始化不完了。如果new呢？就可以，说明spring和jdk实现方式是不同的。

问题的根源知道了，也就是父类spring并没有初始化，它创建的baseDao和cartDao继承的BaseDao并没有关联起来。那现在的问题怎么解决呢？就需要告诉spring创建对象时，去维护这个关系。通过parent属性，那spring就会内部去引用baseDao，并把它注入进去。

<bean id="cartDao" class="parent.CartDao" parent="baseDao" />

结论：spring中的继承和java中的继承是不同的。

### 抽象类

BaseDao单独实例化没有意义，可以进一步的变成抽象类

#### BaseDao.java

package parent;

public abstract class BaseDao {

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

public DataSource getDataSource() {

return dataSource;

}

}

执行报错

Caused by: org.springframework.beans.BeanInstantiationException: Could not instantiate bean class [parent.BaseDao]: Is it an abstract class?; nested exception is java.lang.InstantiationException

抽象类怎么还去实例化了呢？spring还需配置下

<bean id="baseDao" class="parent.BaseDao" abstract="true">

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

</bean>

#### applicationContext.xml

<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-3.2.xsd

">

<bean id="dataSource" class="parent.DataSource">

<property name="conn" value="jdbc:mysql://127.0.0.1:3306/jtdb"/>

</bean>

<bean id="baseDao" class="parent.BaseDao" abstract="true">

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

</bean>

<bean id="cartDao" class="parent.CartDao" parent="baseDao" />

</beans>

这样spring就不会对其进行实例化

package parent;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("parent/applicationContext.xml");

BaseDao baseDao = (BaseDao) context.getBean("baseDao");

System.out.println(baseDao);

CartDao cartDao = (CartDao) context.getBean("cartDao");

cartDao.add();

}

}

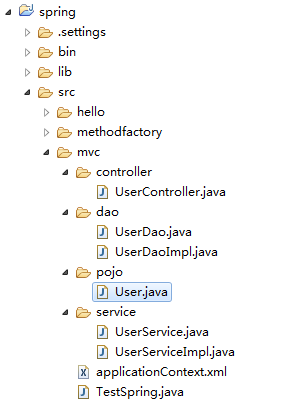
#### 执行报错

获取baseDao就会报错

Caused by: org.springframework.beans.factory.BeanIsAbstractException: Error creating bean with name 'dataSource': Bean definition is abstract

说明BaseDao Spring没有创建实例，去掉BaseDao的测试即可。

## Spring的三层架构



### User.java

package mvc.pojo;

public class User {

private String name;

private Integer age;

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;

}

}

### UserDao.java

package mvc.dao;

public interface UserDao {

public void addUser();

}

### UserDaoImpl.java

package mvc.dao;

import mvc.pojo.User;

public class UserDaoImpl implements UserDao {

private User user;

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}

public void addUser() {

System.out.println("新增用户："+user.getName());

}

}

### UserService.java

package mvc.service;

public interface UserService {

public void addUser();

}

### UserServiceImpl.java

package mvc.service;

import mvc.dao.UserDao;

public class UserServiceImpl implements UserService{

private UserDao userDao;

public void setUserDao(UserDao userDao) {

this.userDao = userDao;

}

public void addUser() {

userDao.addUser();

}

}

### UserController.java

package mvc.controller;

import mvc.service.UserService;

public class UserController {

private UserService userService;

public void setUserService(UserService userService) {

this.userService = userService;

}

public void add(){

userService.addUser();

}

}

### applicationContext.xml

<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-3.2.xsd

">

<bean id="user" class="mvc.pojo.User">

<property name="name" value="万历首辅张居正"/>

<property name="age" value="58"/>

</bean>

<bean id="userDao" class="mvc.dao.UserDaoImpl">

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

</bean>

<bean id="userService" class="mvc.service.UserServiceImpl">

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

</bean>

<bean id="userController" class="mvc.controller.UserController">

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

</bean>

</beans>

### TestSpring.java

package mvc;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import mvc.controller.UserController;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("mvc/applicationContext.xml");

UserController u = (UserController) context.getBean("userController");

u.add();

}

}

# 第二天：自动装配+包扫描

## 属性自动装配

### 传统方式

#### User.java

package autowired.old;

public class User {

private Cat cat;

private Car car;

public Cat getCat() {

return cat;

}

public void setCat(Cat cat) {

this.cat = cat;

}

public Car getCar() {

return car;

}

public void setCar(Car car) {

this.car = car;

}

}

#### Cat.java

package autowired.old;

public class Cat {

public void play(){

System.out.println("cat play");

}

}

#### Car.java

package autowired.old;

public class Car {

public void go(){

System.out.println("car go");

}

}

#### applicationContext.xml

<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-3.2.xsd

">

<bean id="user" class="autowired.old.User">

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

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

</bean>

<bean id="cat" class="autowired.old.Cat"/>

<bean id="car" class="autowired.old.Car"/>

</beans>

#### TestSpring.java

package autowired.old;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("autowired/old/applicationContext.xml");

User user = (User) context.getBean("user");

user.getCar().go();

user.getCat().play();

}

}

### byName

上面的方式这些bean的声明满天飞，我们工作量并未减少，感觉还很麻烦。而且xml中无法对其检查，运行加载时报错，其实只是路径问题、字母缺漏或者大小写的问题。开发效率低。

<bean id="user" class="autowired.old.User">

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

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

</bean>

修改为

<bean id="user" class="autowired.old.User" autowire="byName"/>

真的好简单，一下代码看着清爽很多。

byName的处理过程是，先去找所有的set方法，setXyz去掉set，首字母变小写xyz，然后去找对应的属性，通过属性名称去配置文件中找id=xyz的，通过id再找到具体的类，然后进行set设置完成自动注入。

注意：byName是按bean的id名称对应，其实叫byId才更准确。byName容易误以为是<bean name="cat"/>。

### byType

修改id名称

<bean id="catA" class="autowired.old.Cat"/>

执行时报空指针java.lang.NullPointerException。因为按byName规则找不对应set方法，真正的setCat就没执行，对象就没有初始化，所以调用时就报了空指针错误。那如果我们有这种需求，id名称就是和规则不同呢？spring为我们提供了另一种方式byType按class装配。

<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-3.2.xsd

">

<bean id="user" class="autowired.old.User" autowire="byType"/>

<bean id="catA" class="autowired.old.Cat"/>

<bean id="car" class="autowired.old.Car"/>

</beans>

修改成autowire="byType"后又可以使用了。

<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-3.2.xsd

">

<bean id="user" class="autowired.old.User" autowire="byType"/>

<bean class="autowired.old.Cat"/>

<bean class="autowired.old.Car"/>

</beans>

狠一点，把cat和car的id都去掉，依然可以。

通常spring使用byName多些。

### autowire

如果要配置很多呢？

<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-3.2.xsd

" default-autowire="byName">

<bean id="user" class="autowired.old.User" autowire="default"/>

<bean id="user1" class="autowired.old.User" autowire="default"/>

<bean id="user2" class="autowired.old.User" autowire="default"/>

<bean id="user3" class="autowired.old.User" autowire="default"/>

<bean id="user4" class="autowired.old.User" autowire="default"/>

<bean id="cat" class="autowired.old.Cat"/>

<bean id="car" class="autowired.old.Car"/>

</beans>

### 属性注解@Autowired

如果配置都写在配置文件中，会造成配置文件特别长，不易维护。通过注解方式直接注入属性。jdk1.5引入注解，spring2.5开始全面支持注解。

#### 引入spring的context文件头

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

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

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-3.2.xsd

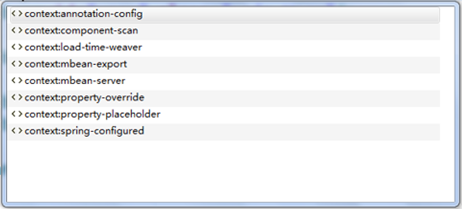
http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

#### Eclipse导入xsd约束，否则不会出现自动提示

敲入context标签，就有下面提示



#### 配置开启属性注解

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

<context:annotation-config/>

<bean id="user" class="annotation.User"/>

<bean id="car" class="annotation.Car"/>

<bean id="cat" class="annotation.Cat"/>

</beans>

#### 增加注解

去掉set方法，在属性上加入注解@Autowired

package annotation;

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

public class User {

@Autowired

private Cat cat;

@Autowired

private Car car;

public Cat getCat() {

return cat;

}

public Car getCar() {

return car;

}

}

#### 测试

package annotation;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("annotation/applicationContext.xml");

User user = (User) context.getBean("user");

user.getCar().go();

user.getCat().play();

}

}

#### 工作原理

可以看到autowired就方便了，对象的注入变得更加简单，可以省略set方法。

autowired到底怎么注入的？

根据注解获取到类名和bean的class进行匹配，匹配到设置其实例给属性。

<bean id="user" class="annotation.User"/>

<bean id="carA" class="annotation.Car"/>

<bean id="catB" class="annotation.Cat"/>

改名结果依然正常，说明其是按byType进行匹配。

### @Qualifier

实际中用的不多

@Autowired不支持id匹配，如果按byName匹配怎么做？就要用到@Qualifier注解。

#### User.java

package annotation;

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

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

public class User {

@Autowired

private Cat cat;

@Autowired @Qualifier(value="carA") //可以写在一行上也可以分开

private Car car;

public Cat getCat() {

return cat;

}

public Car getCar() {

return car;

}

}

#### applicationContext.xml

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

<context:annotation-config/>

<bean id="user" class="annotation.User"/>

<bean id="carA" class="annotation.Car"/>

<bean id="catB" class="annotation.Cat"/>

</beans>

只有@Qualifier可以看到也不行，必须两个注解一块使用。

### @Resource

就想填入一个注解，byType、byName自由选择呢？

spring没有给我们，jdk给了我们，jdk的@Resource注解。

#### User.java

package annotation;

import javax.annotation.Resource;

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

public class User {

@Autowired

private Cat cat;

@Resource(name="carA")

private Car car;

public Cat getCat() {

return cat;

}

public Car getCar() {

return car;

}

}

#### applicationContext.xml

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

<context:annotation-config/>

<bean id="user" class="annotation.User"/>

<bean id="carA" class="annotation.Car"/>

<bean id="catB" class="annotation.Cat"/>

</beans>

@Resource是按byType方式装配

@Resource(name="carA")是按byName方式装配

但需要说明的是@Resource虽然更加规范不会依赖spring框架，但如果使用spring框架的话还是推荐使用@Autowired，实际开发中会有细小差异。

### 面试：@Autowired与@Resource的区别

1. @Autowired与@Resource都可以用来装配bean。都可以写在字段上，或写在setter方法上。

2. @Autowired默认按类型装配（属于spring），默认情况下必须要求依赖对象必须存在，如果要允许null 值，可以设置它的required属性为false，如：@Autowired(required=false) ，如果我们想使用名称装配可以结合@Qualifier注解进行使用。

3. @Resource（属于J2EE），默认按照名称进行装配，名称可以通过name属性进行指定。如果没有指定name属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在setter方法上默认取属性名进行装配。 当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

它们的作用相同都是用注解方式注入对象，但执行顺序不同。@Autowired先byType，@Resource先byName。

## 类自动装配

上面我们只是少写了set方法而已，对代码量来说最头大的是xml的配置声明，没有校验很易写错。能否把这部分配置代码干掉呢？

### 包扫描

spring退出了包扫描，这下代码量几乎为零，号称“xml零配置”。

#### User.java

package annotation;

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

import org.springframework.stereotype.Component;

@Component

public class User {

@Autowired

private Cat cat;

@Autowired

private Car car;

public Cat getCat() {

return cat;

}

public Car getCar() {

return car;

}

}

#### Car.java

package annotation;

import org.springframework.stereotype.Component;

@Component

public class Car {

public void go(){

System.out.println("car go");

}

}

#### Cat.java

package annotation;

import org.springframework.stereotype.Component;

@Component

public class Cat {

public void play(){

System.out.println("cat play");

}

}

#### applicationContext.xml

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

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

</beans>

#### 测试类

package annotation;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("annotation/applicationContext.xml");

User user = (User) context.getBean("user");

user.getCar().go();

user.getCat().play();

}

}

执行过程：加载spirng核心配置文件，遍历到<context-component:scan，去指定的目录下找到所有含有@Component的类，按之前<bean声明的规则创建对象，保存在map中，key=类名首字母小写（实例名称），value=创建的对象。这样就无需大量的xml的配置，而且非常自动化，只要在规定的路径下添加新的类。

### 我是谁？类的id是？

spring提供了接口BeanNameAware可以查看类的id名称。

#### CIA.java

package inspect;

import org.springframework.beans.factory.BeanNameAware;

import org.springframework.stereotype.Component;

@Component

public class CIA implements BeanNameAware{

public void kill(String name){

System.out.println("Kill " + name);

}

public void setBeanName(String name) {

System.out.println("当前的类的id：" + name);

}

}

#### applicationContext.xml

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

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

</beans>

#### 测试类

package inspect;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("inspect/applicationContext.xml");

CIA x = (CIA) context.getBean("CIA");

x.kill("you");

}

}

类目和ID对应表：

类名： CIA cIA CiA ciA

ID: CIA cIA ciA ciA

分析：类名的关键在于第二个字母，第二个字母大写，首字母大写小写无所谓。如果第二个字母小写，那第一个必须小写。第二个字母之后大小写没关系。

### 自定义bean的id

#### CIA.java

package inspect;

import org.springframework.beans.factory.BeanNameAware;

import org.springframework.stereotype.Component;

@Component(value="cia")

public class CIA implements BeanNameAware{

public void kill(String name){

System.out.println("Kill " + name);

}

public void setBeanName(String name) {

System.out.println("当前的类的id：" + name);

}

}

#### 测试类

package inspect;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("inspect/applicationContext.xml");

CIA x = (CIA) context.getBean("cia");

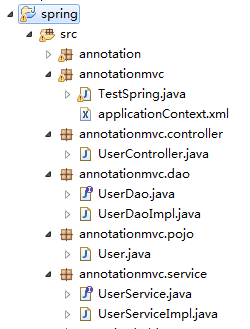
x.kill("you");

}

}

## Spring的三层架构

注入方式实现三层架构



### User.java

package annotationmvc.pojo;

public class User {

private String name;

private Integer age;

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;

}

}

### UserDao.java

package annotationmvc.dao;

public interface UserDao {

public void addUser();

}

### UserDaoImpl.java

package annotationmvc.dao;

import org.springframework.stereotype.Repository;

@Repository(value="userDao")

public class UserDaoImpl implements UserDao {

public void addUser() {

System.out.println("新增用户：");

}

}

### UserService.java

package annotationmvc.service;

public interface UserService {

public void addUser();

}

### UserServiceImpl.java

package annotationmvc.service;

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

import org.springframework.stereotype.Service;

import annotationmvc.dao.UserDao;

@Service(value="userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

public void addUser() {

userDao.addUser();

}

}

### UserController.java

package annotationmvc.controller;

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

import org.springframework.stereotype.Controller;

import annotationmvc.service.UserService;

@Controller

public class UserController {

@Autowired

private UserService userService;

public void add(){

userService.addUser();

}

}

### applicationContext.xml

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

<context:component-scan base-package="annotationmvc.dao,annotationmvc.service,annotationmvc.controller"/>

</beans>

### 测试类

package annotationmvc;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import annotationmvc.controller.UserController;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("annotationmvc/applicationContext.xml");

UserController u = (UserController) context.getBean("userController");

u.add();

}

}

### 三步走

第一步：声明资源类型

@Repository 声明dao

@Service 声明service

@Controller 声明controller

注意名称只是方便开发者理解，人为区分，计算机无需区分都是@Compnent。

第二步：在实现类中加注解@Autowired注入资源

如UserServiceImpl中注入UserDao，UserController中注入UserService。

第三步：包扫描路径

<context:component-scan base-package="annotationmvc.dao,annotationmvc.service,annotationmvc.controller"/>

<context:component-scan base-package="annotationmvc.\*"/>

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

多个包直接用逗号隔开，annotationmvc.\*当前目录下，\*所有包路径下。

\*是可以，但其它不需要的文件不需要的目录都检查，例如工具类，抽象类不用扫描去创建对象，如果都处理，就会浪费执行时间，性能就会变慢。

## 自动装配问题

### 包扫描错误导致bean未被定义

包扫描最大一个问题就是扫描路径问题，如果配置的扫描路径不正确，有些bean没放在扫描的目录，spring就不会创建这个bean，这样getBean时就报错。这种错误非常隐晦。所以如果报bean没有定义。要检查两个地方，一个是bean名称是否正确，一个就是扫描路径。

例如：

一个类放在home目录下，包扫描配置的是home.\*。

这个bean就不会创建，因为.\*的意思是当前目录下的子目录，不包含当前目录，改成home就可以，它代表home及其所有子目录。

配置文件：

<context:component-scan base-package="proxy.statica.\*"/>

执行报错：

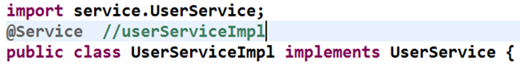
org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'staticProxy' is defined

解决方法：

<context:component-scan base-package="proxy.statica"/>

### @Autowired是按byName还是按byType注入的？

前面我们说注解@Autowired是按byType方式，但没有证明，下面我们就来证明这个问题。



再做一个实现类，就改个名字

package annotationmvc.dao;

import org.springframework.stereotype.Repository;

@Repository

public class UserDaoImpl2 implements UserDao {

public void addUser() {

System.out.println("新增用户：");

}

}

执行测试类，报错

org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'userServiceImpl': Injection of autowired dependencies failed; nested exception is org.springframework.beans.factory.BeanCreationException: Could not autowire field: private annotationmvc.dao.UserDao annotationmvc.service.UserServiceImpl.userDao; nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type [annotationmvc.dao.UserDao] is defined: expected single matching bean but found 2: userDaoImpl,userDaoImpl2

什么错误呢？发现了两个匹配的，不知道该去调用哪个？从错误提示就可看出，spring是按byType去查找了。两个实现类名称不同id，还报错说明不是按名称去找的。@Autowire是按接口的类去查找，结果实现类有两个不唯一，没法选择了于是报错。

### 一个接口多个实现类怎么装配？

那现在就有多个实现类，如何解决呢？

UserDaoImpl.java

package annotationmvc.dao;

import org.springframework.stereotype.Repository;

@Repository(value="userDao") //设置value值进行区分

public class UserDaoImpl implements UserDao {

public void addUser() {

System.out.println("新增用户：1");

}

}

UserDaoImpl2.java

package annotationmvc.dao;

import org.springframework.stereotype.Repository;

@Repository(value="userDao2")

public class UserDaoImpl2 implements UserDao {

public void addUser() {

System.out.println("新增用户：2");

}

}

UserServiceImpl.java

package annotationmvc.service;

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

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

import org.springframework.stereotype.Service;

import annotationmvc.dao.UserDao;

@Service

public class UserServiceImpl implements UserService{

@Autowired //调用的实现类1

private UserDao userDao;

public void addUser() {

userDao.addUser();

}

}

执行结果

新增用户：1

那怎么调用实现类2呢？

强制指定按byName的注解Qualifier

UserServiceImpl.java

package annotationmvc.service;

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

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

import org.springframework.stereotype.Service;

import annotationmvc.dao.UserDao;

@Service

public class UserServiceImpl implements UserService{

@Autowired

@Qualifier(value="userDao2")

private UserDao userDao;

public void addUser() {

userDao.addUser();

}

}

执行结果

新增用户：2

## 代理设计模式

AOP 实现的关键就在于AOP框架自动创建的AOP代理，AOP代理则可分为静态代理和动态代理两大类，其中静态代理是指使用AOP框架提供的命令进行编译，从而在编译阶段就可生成AOP代理类，因此也称为编译时增强；而动态代理则在运行时借助于JDK、CGLIB动态代理等在内存中“临时”生成AOP动态代理类，因此也被称为运行时增强。

静态代理有：编译时织入（特殊编译器实现）、类加载时织入（特殊的类加载器实现）。

动态代理有：jdk动态代理（基于接口来实现）、CGLIB（基于类实现）。

### 什么叫代理？

#### Host.java

package proxy.old;

//房东

public class Host {

public void rent(){

System.out.println("房子出租");

}

}

#### Rent.java

package proxy.old;

public interface Rent {

public void rent();

}

#### Proxy.java

package proxy.old;

public class Proxy implements Rent{

Host host = new Host();

@Override

public void rent() {

System.out.println("带客户看房");

host.rent();

System.out.println("收取中介费");

}

}

#### Client.java

package proxy.old;

public class Client {

public static void main(String[] args) {

Proxy proxy = new Proxy();

proxy.rent();

}

}

### 代理模式的特点

* 核心要完成本质工作
* 在完成本职工作的基础上要完成额外的工作

### 业务：控制数据库事务

#### TransactionManager.java

package proxy.tx;

import org.springframework.stereotype.Component;

@Component

public class TransactionManager {

public void begin(){

System.out.println("事务开启");

}

public void commit(){

System.out.println("事务提交");

}

public void rollback(){

System.out.println("事务回滚");

}

}

#### UserDao.java

package proxy.tx.dao;

public interface UserDao {

public void addUser();

public void updateUser();

}

#### UserDaoImpl.java

package proxy.tx.dao;

import org.springframework.stereotype.Repository;

@Repository(value="userDao")

public class UserDaoImpl implements UserDao{

public void addUser() {

System.out.println("新增用户");

}

public void updateUser() {

System.out.println("修改用户");

}

}

#### UserService.java

package proxy.tx.service;

public interface UserService {

public void addUser();

public void updateUser();

}

#### UserServiceImpl.java

package proxy.tx.service;

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

import org.springframework.stereotype.Service;

import proxy.tx.TransactionManager;

import proxy.tx.dao.UserDao;

@Service(value="userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

@Autowired

private TransactionManager tx;

public void addUser() {

try {

tx.begin();

userDao.addUser();

tx.commit();

} catch (Exception e) {

tx.rollback();

}

}

public void updateUser() {

try {

tx.begin();

userDao.updateUser();

tx.commit();

} catch (Exception e) {

tx.rollback();

}

}

}

#### applicationContext.xml

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

<context:component-scan base-package=" proxy.tx.service,proxy.tx.web,proxy.tx"/>

</beans>

#### 测试类

package proxy.tx;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import proxy.tx.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("proxy/tx/applicationContext.xml");

UserService userService = (UserService) context.getBean("userService");

userService.addUser();

userService.updateUser();

}

}

上面的代码看看有什么问题？

业务代码中夹杂了tx.begin()事务开始tx.commit()提交的代码。当前的业务层变得不纯粹了，业务代码和事务代码交织。

解决方案：把业务代码和事务代码分开，别干扰我，让我纯粹。

## 代理实现的三种方式

### 静态代理

实现业务代码和事务代码分离。

#### TransactionManager.java

package proxy.staticp;

import org.springframework.stereotype.Component;

@Component

public class TransactionManager {

public void begin(){

System.out.println("事务开启");

}

public void commit(){

System.out.println("事务提交");

}

public void rollback(){

System.out.println("事务回滚");

}

}

#### UserDao.java

package proxy.staticp.dao;

public interface UserDao {

public void addUser();

public void updateUser();

}

#### UserDaoImpl.java

package proxy.staticp.dao;

import org.springframework.stereotype.Repository;

@Repository(value="userDao")

public class UserDaoImpl implements UserDao{

public void addUser() {

System.out.println("新增用户");

}

public void updateUser() {

System.out.println("修改用户");

}

}

#### UserService.java

package proxy.staticp.service;

public interface UserService {

public void addUser();

public void updateUser();

}

#### UserServiceImpl.java

package proxy.staticp.service;

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

import org.springframework.stereotype.Service;

import proxy.staticp.dao.UserDao;

@Service(value="userSerivce")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

public void addUser() {

userDao.addUser();

}

public void updateUser() {

userDao.updateUser();

}

}

#### StaticProxy.java

package proxy.staticp;

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

import org.springframework.stereotype.Component;

import proxy.staticp.service.UserService;

@Component

public class StaticProxy implements UserService{

@Autowired

private UserService target;

@Autowired

private TransactionManager tx;

@Override

public void addUser() {

try {

tx.begin();

target.addUser();

tx.commit();

} catch (Exception e) {

tx.rollback();

}

}

@Override

public void updateUser() {

try {

tx.begin();

target.updateUser();

tx.commit();

} catch (Exception e) {

tx.rollback();

}

}

}

注意静态代理不是指的类静态或者方法静态，而是指在运行之前已经形成，也就是编译时已经代码固定。

#### applicationContext.xml

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

<context:component-scan base-package="proxy.staticp.service,proxy.staticp.web,proxy.staticp"/>

</beans>

#### 测试类

package proxy.staticp;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("proxy/staticp/applicationContext.xml");

StaticProxy proxy = (StaticProxy) context.getBean("staticProxy");

proxy.addUser();

proxy.updateUser();

}

}

可以看到这样业务层代码非常干净，事务层和业务层分离，业务层就只做业务层的代码。耦合性就消除了。这就实现了架构的松耦合。

很显著也看到它的缺点：

一个静态代理类智能代理一个类，代码复用度不高。

# 第三天：创建对象+静态代理+动态代理+AOP

## 代理实现的三种方式

### jdk动态代理

动态代理有种实现方式：一个是jdk提供的，一个是cglib提供的。

动态代理三要素：proxy代理类、InvocationHandler接口、Invoke方法。

#### TransactionManager.java

package proxy.jdk;

import org.springframework.stereotype.Component;

@Component

public class TransactionManager {

public void begin(){

System.out.println("事务开启");

}

public void commit(){

System.out.println("事务提交");

}

public void rollback(){

System.out.println("事务回滚");

}

}

#### UserDao.java

package proxy.jdk.dao;

public interface UserDao {

public void addUser();

public void updateUser();

}

#### UserDaoImpl.java

package proxy.jdk.dao;

import org.springframework.stereotype.Repository;

@Repository(value="userDao")

public class UserDaoImpl implements UserDao{

public void addUser() {

System.out.println("新增用户");

}

public void updateUser() {

System.out.println("修改用户");

}

}

#### UserService.java

package proxy.jdk.service;

public interface UserService {

public void addUser();

public void updateUser();

}

#### UserServiceImpl.java

package proxy.jdk.service;

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

import org.springframework.stereotype.Service;

import proxy.jdk.dao.UserDao;

@Service(value="userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

public void addUser() {

userDao.addUser();

}

public void updateUser() {

userDao.updateUser();

}

}

#### DynamicProxy.java

步骤：

1. 写一个获取代理方法，把被代理对象和事务对象作为参数传入
2. 通过反射创建实例 Proxy.newProxyInstance，参数：类加载器、接口
3. 匿名内部类new InvocationHandler，参数：代理对象、方法，参数
4. 回调 result = method.invoke，参数：目标对象、参数

package proxy.jdk;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public class DynamicProxy {

// 创建代理对象，传入真实对象和事务对象

public static Object getProxy(final Object target, final TransactionManager tx) {

Object proxy = Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(),

// 采用匿名内部类方式，参数必须final

new InvocationHandler() {

// 代理对象，真实对象的方法，真实对象的参数

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

Object result = null;

Long beginTime = System.currentTimeMillis();

try {

tx.begin();

// 通过反射调用真实对象的方法

result = method.invoke(target, args);

tx.commit();

} catch (Exception e) {

tx.rollback(); // 异常后返回null

}

Long endTime = System.currentTimeMillis();

System.out.println("执行时间：" + (endTime - beginTime) + " 毫秒");

// 必须设置返回值，否则下面过程就无法获取到执行结果

return result;

}

});

return proxy;

}

}

#### applicationContext.xml

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

<context:component-scan base-package="proxy.jdk.dao,

proxy.jdk.service.UserService,

proxy.jdk.service,proxy.jdk"/>

</beans>

#### 测试类

package proxy.jdk;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import proxy.jdk.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("proxy/jdk/applicationContext.xml");

UserService target = (UserService) context.getBean("userService");

TransactionManager tx = (TransactionManager) context.getBean("transactionManager");

UserService proxy = (UserService)DynamicProxy.getProxy(target, tx);

System.out.println(proxy.getClass());

proxy.addUser();

}

}

#### 执行结果

class com.sun.proxy.$Proxy7 //代理对象

事务开启

新增用户

事务提交

执行时间：0 毫秒 //没有真正的业务所以执行很快

**Proxy代理类**

Proxy是JDK中用来创建代理对象的类，其中使用最多的是newProxyInstance方法来创建代理对象。

public static Object newProxyInstance(

ClassLoader loader, //由哪个类加载器加载生成代理对象

Classs<?>[] interfaces, //接口对象数组，必须传入目标对象的接口

InvocationHandler h //表示一个invocationHandler对象，当代理对象执行方法时，会调用改InvocatonHandler对象

) throws IllegalArgumentException

### 不支持实现类

#### PersonServiceImpl.java

实现类没有接口

package proxy.jdk.service;

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

import org.springframework.stereotype.Service;

import proxy.jdk.dao.UserDao;

@Service

public class PersonServiceImpl {

@Autowired

private UserDao userDao;

public void addUser() {

userDao.addUser();

}

public void updateUser() {

userDao.updateUser();

}

}

#### 测试类

package proxy.jdk;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import proxy.jdk.service.PersonServiceImpl;

import proxy.jdk.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("proxy/jdk/applicationContext.xml");

UserService target = (UserService) context.getBean("userService");

PersonServiceImpl impl = (PersonServiceImpl) context.getBean("personServiceImpl");

TransactionManager tx = (TransactionManager) context.getBean("transactionManager");

UserService proxy = (UserService)DynamicProxy.getProxy(target, tx);

System.out.println(proxy.getClass());

proxy.addUser();

PersonServiceImpl proxyImpl = (PersonServiceImpl)DynamicProxy.getProxy(impl, tx);

System.out.println(proxyImpl.getClass());

proxyImpl.addUser();

}

}

#### 执行报错

没有接口，则报错。

java.lang.ClassCastException: com.sun.proxy.$Proxy8 cannot be cast to proxy.jdk.service.PersonServiceImpl

at proxy.jdk.TestSpring.test(TestSpring.java:27)

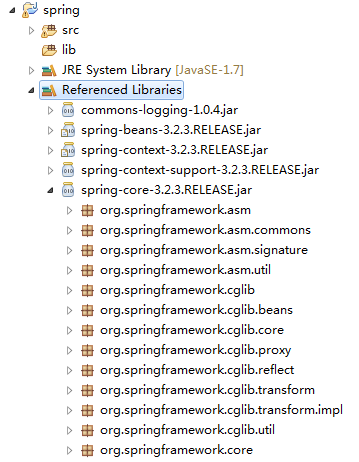
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:606)

### cglib动态代理



CODE GENERLIZE LIBRARY，Spring框架底层已经加入cglib的jar包

#### TransactionManager.java

package proxy.cglib;

import org.springframework.stereotype.Component;

@Component

public class TransactionManager {

public void begin(){

System.out.println("事务开启");

}

public void commit(){

System.out.println("事务提交");

}

public void rollback(){

System.out.println("事务回滚");

}

}

#### UserDao.java

package proxy.cglib.dao;

public interface UserDao {

public void addUser();

public void updateUser();

}

#### UserDaoImpl.java

package proxy.cglib.dao;

import org.springframework.stereotype.Repository;

@Repository(value="userDao")

public class UserDaoImpl implements UserDao{

public void addUser() {

System.out.println("新增用户");

}

public void updateUser() {

System.out.println("修改用户");

}

}

#### UserService.java

package proxy.cglib.service;

public interface UserService {

public void addUser();

public void updateUser();

}

#### UserServiceImpl.java

package proxy.cglib.service;

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

import org.springframework.stereotype.Service;

import proxy.cglib.dao.UserDao;

@Service(value="userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

public void addUser() {

userDao.addUser();

}

public void updateUser() {

userDao.updateUser();

}

}

#### DynamicProxy.java

步骤：

1. 创建Enhancer对象
2. 设置接口、父类、回调拦截器
3. 回调参数：Object代理对象，Method为上下文中实体类所调用的被代理的方法引用，Object[]为参数列表，MethodProxy为生成的代理类对方法的代理引用
4. enhancer.create 返回对象

package 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;

import proxy.cglib.service.UserService;

public class DynamicProxy {

// 创建代理对象，传入真实对象和事务对象

public static Object getProxy(final UserService target, final TransactionManager tx) {

Enhancer enhancer = new Enhancer(); //创建增强器

//没有接口也要设置，内部会自动判断

enhancer.setInterfaces(target.getClass().getInterfaces());

enhancer.setSuperclass(target.getClass()); //设置父类，cglib创建的类都是目标对象的子类

enhancer.setCallback(new MethodInterceptor() {

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

try {

tx.begin();

method.invoke(target, args);

tx.commit();

} catch (Exception e) {

tx.rollback();

}

return null;

}

});

return enhancer.create();

}

}

#### applicationContext.xml

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

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

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

">

<context:component-scan base-package="proxy.cglib.dao,

proxy.cglib.service,

proxy.cglib"/>

</beans>

#### 测试类

package proxy.cglib;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import proxy.cglib.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("proxy/cglib/applicationContext.xml");

UserService target = (UserService) context.getBean("userService");

TransactionManager tx = (TransactionManager) context.getBean("transactionManager");

UserService proxy = (UserService)DynamicProxy.getProxy(target, tx);

System.out.println(proxy.getClass());

proxy.addUser();

}

}

#### 执行结果

class proxy.cglib.service.UserServiceImpl$$EnhancerByCGLIB$$f06de5e5

事务开启

新增用户

事务提交

### 支持实现类

#### PersonServiceImpl.java

package proxy.cglib.service;

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

import org.springframework.stereotype.Service;

import proxy.cglib.dao.UserDao;

@Service

public class PersonServiceImpl {

@Autowired

private UserDao userDao;

public void addUser() {

userDao.addUser();

}

public void updateUser() {

userDao.updateUser();

}

}

#### 测试类

package proxy.cglib;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import proxy.cglib.service.PersonServiceImpl;

import proxy.cglib.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("proxy/cglib/applicationContext.xml");

UserService target = (UserService) context.getBean("userService");

PersonServiceImpl impl = (PersonServiceImpl) context.getBean("personServiceImpl");

TransactionManager tx = (TransactionManager) context.getBean("transactionManager");

UserService proxy = (UserService)DynamicProxy.getProxy(target, tx);

System.out.println(proxy.getClass());

proxy.addUser();

PersonServiceImpl proxyImpl = (PersonServiceImpl)DynamicProxy.getProxy(impl, tx);

System.out.println(proxyImpl.getClass());

proxyImpl.addUser();

}

}

#### 执行结果

class proxy.cglib.service.UserServiceImpl$$EnhancerByCGLIB$$aab2b748

事务开启

新增用户

事务提交

class proxy.cglib.service.PersonServiceImpl$$EnhancerByCGLIB$$88152cda

事务开启

新增用户

事务提交

上面是接口支持，下面是实现类支持。因为cglib是分析编译后的二进制文件，所以无需接口也可以实现。

### 三种方式比较

代理设计模式最强大的在于，在代理类中包装一个被代理对象，然后影响被代理对象的行为！

|  |  |
| --- | --- |
| **代理实现方式** | **说明** |
| **静态代理** | 一个类一个代理，重复代码仍然较多，复用度低。  静态代理：事先写好代理对象类，在程序发布前就已经存在了。  调用真实业务方法 |
| **jdk**  **动态代理** | 动态代理：应用程序发布后，通过动态创建代理对象。  通过反射进行invoke回调  要求业务类必须有接口，没有接口无法生成代理对象  class com.sun.proxy.$Proxy7 |
| **cglib**  **动态代理** | 不要求业务类必须有接口，有实现类即可  底层采用asm字节码生成框架生成代理类的字节码，它创建慢，但一旦创建要比jdk方式快。  cglib使用继承，因此父类最好不要有final属性，final阻止继承和多态。  class proxy.cglib.service.UserServiceImpl$$EnhancerByCGLIB$$f06de5e5 |
| **ASM**  **字节码生成框架** | ASM是一个Java字节码生成框架，它能够以二进制形式修改已有类，动态生成类。ASM可也直接产生二进制class文件，也可以在类被加载如JVM前动态改变类行为，ASM从类文件中读入信息后，能够改变类行为，分析类信息，甚至能够根据用户要求生成新类。不过ASM在创建class字节码的过程中，操纵的级别是底层JVM的汇编指令级别。这要求就高了，对开发者要对class的结构和JVM指令要有一定了解。 |

## SpringAOP

### 动态代理缺点

从上述示例中我们可以看出，静态代理太死板，所以实际中使用不多。而动态代理非常灵活，有接口通常使用jdk动态代理方式，没有接口通常采用cglib方式。虽然动态代理已经很不错，代码分离，实现松耦合，比较通用，那还有没有改进的地方呢？

不知道大家有没有发现，两种动态代理非常霸道，如果一个类被它代理，则接口或者类的所有方法都被强迫执行。在实际开发中组织类不是按事务需求，而是按业务划分，那必然有些方法需要事务，有些方法不需要，动态代理可不管这么多。再者我们还需要写一个动态代理类来完成，对开发者暴露了部分开发的内容，封装性不是很彻底。

### 面试：有了OOP为什么还要AOP？

上面从编程细节可以看出动态代理控制粒度太粗，能否更加细致呢？OOP做不到，AOP可以。那什么是AOP呢？

AOP（Aspect-Oriented Programming，面向切面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象瀑布层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能，日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码。

在OOP设计中，它导致了上面这些类型的代码大量重复，不利于各个模块的重用。而AOP技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即切面。

所谓“切面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向切面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“切面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹。使用“横切”技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。AOP 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。简而言之，AOP的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

OOP实现了纵向复用，而AOP实现了横向复用，纵向+横向=无敌！

### 实现AOP的技术分为两大类

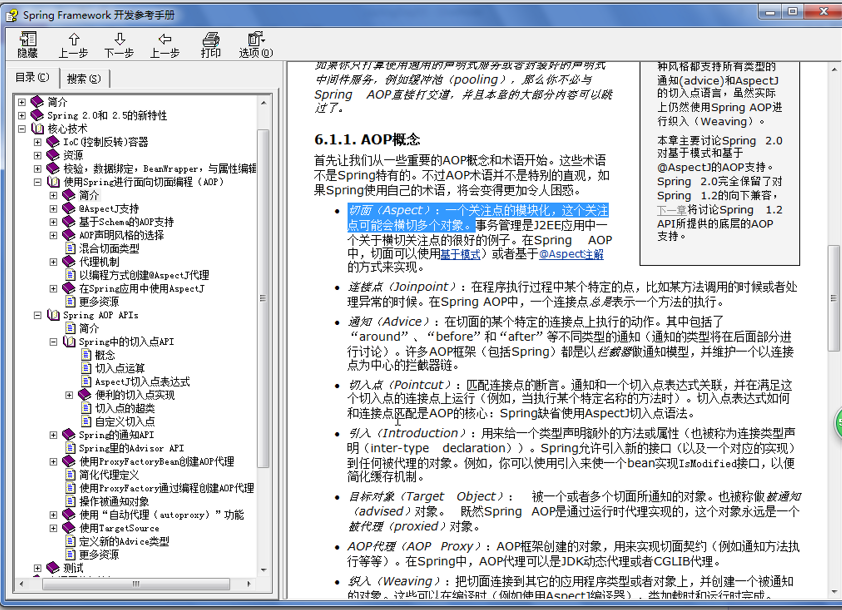
一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；

二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

Spring提供了两种方式来生成代理对象: JDKProxy和Cglib，具体使用哪种方式生成由AopProxyFactory根据AdvisedSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，如果目标对象没有实现接口，则默认会采用CGLIB代理。如果目标对象实现了接口，可以强制使用CGLIB实现代理。（添加CGLIB库，并在spring配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）

### AOP概念

AOP面向切面编程，Spring2.5-中文参考手册.chm



### 织入Weaving

上面的例子我们使用了一个aop:before，它怎么就自动会在业务代码方法之前执行呢？就因为它叫before？当然不是，它怎么就跑到了业务代码之前？这就要靠一种叫织入Weaving的技术实现。那什么叫织入呢？

#### Java织入

提供3个织入点：

编译期：在Java编译期，采用特殊的编译器，将切面织入到Java类中。

类加载期：通过特殊的类加载器，在类字节码加载到JVM时，织入切面。

运行期：采用CGLIB或JDK动态代理进行切面的织入。

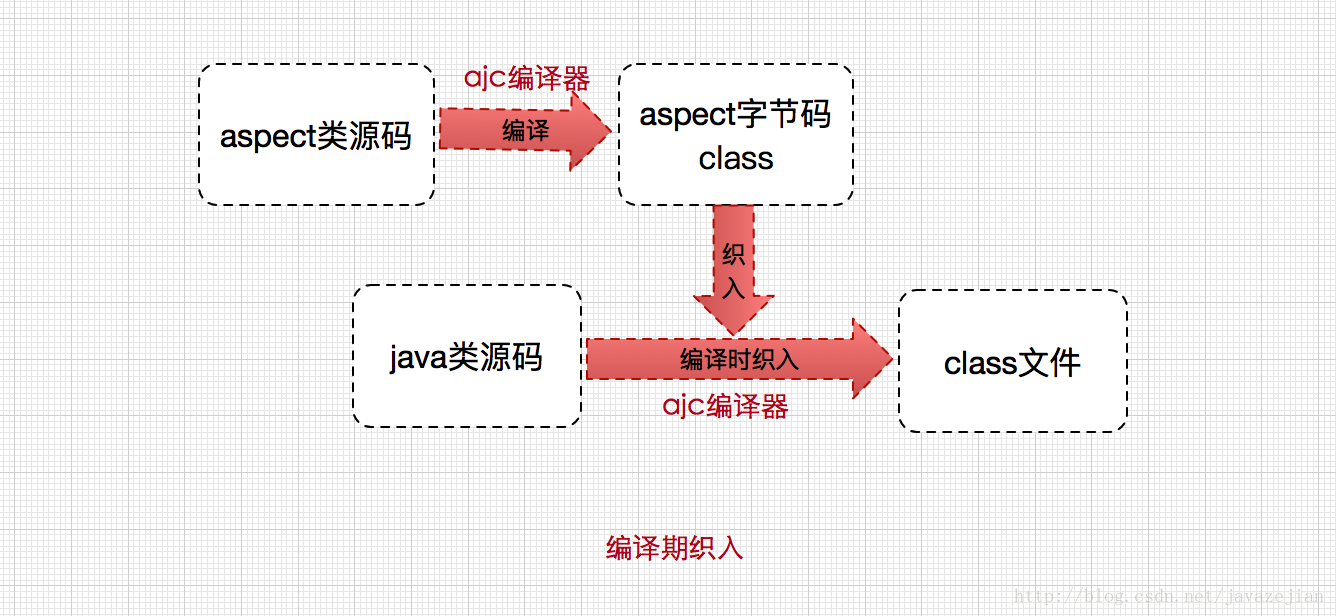
#### AspectJ织入

提供3个织入点：

编译期：把aspect类（切面）和目标类放在一起用ajc（特殊编译器类似javac编译器）编译。

后编译期：目标类可能已经打成jar包，这时候也可以用ajc命令将jar在织入一次。

类加载期：Load-Time Weaving LTW，在jvm加载类的时候，做字节码的替换。



其中前2个时间点，可以理解为静态织入，因为在class文件生成后，就已经织入好了。类加载期织入，可以理解为动态织入，因为这个类替换是在jvm加载类的时候完成的。

类加载期织入又有3种方式：agent方式、启动时加参数替换默认的系统类加载器、自定义类加载器。

### AOP的五大金刚

|  |  |  |
| --- | --- | --- |
|  | **名词** | **说明** |
|  | 切面 Aspect | 很好理解，就是一个面，java在这之前关注都是每个类，而AOP出现，关注点提升到更高的层面，由关注某个点提升到关注某个面，所以面向切面编程是面向对象编程的补充。其具体体现为一个类。 |
|  | 切入点 Pointcut | 进行规则匹配，不同的匹配，执行不同的内容。 |
|  | 目标对象 Target Object | 被代理的对象。 |
|  | 连接点 JoinPoint | 程序和面交汇的点叫做连接点。它是一个抽象的概念，在实现AOP时，并不需要去定义一个join point。 |
|  | 通知 Advice | 在连接点上触发的事件，从而可以在很多位置去执行额外的动作。如前置通知、后置通知、异常通知、环绕通知。 |

## SpringAOP例子

AOP有两种方式一种xml配置方式，一种注解配置方式，我们先来了解xml配置方式。

开发步骤：

* 创建java工程，导入aop支持jar包
* 编写接口
* 编写业务实现类
* 创建切面类
* 创建通知方法
* 配置文件，增加aop支持，配置aop配置、切点、切面、通知
* 编写测试类进行测试

### 创建java工程，AOP引入jar包

新增jar，支持对aop的处理

aopalliance-1.0.jar

aspectjrt-1.7.4.jar

aspectjweaver-1.7.4.jar

spring-aop-3.2.3.RELEASE.jar

spring-aop-3.2.3.RELEASE-sources.jar

spring-aspects-3.2.3.RELEASE.jar

spring-aspects-3.2.3.RELEASE-sources.jar

commons-logging-1.0.4.jar

spring-beans-3.2.3.RELEASE.jar

spring-context-3.2.3.RELEASE-sources.jar

spring-context-3.2.3.RELEASE.jar

spring-context-support-3.2.3.RELEASE-sources.jar

spring-context-support-3.2.3.RELEASE.jar

spring-core-3.2.3.RELEASE-sources.jar

spring-core-3.2.3.RELEASE.jar

spring-expression-3.2.3.RELEASE-sources.jar

spring-expression-3.2.3.RELEASE.jar

spring-test-3.2.3.RELEASE-sources.jar

spring-test-3.2.3.RELEASE.jar

### UserService.java

package aop.service;

public interface UserService {

public void addUser();

}

### UserServiceImpl.java

package aop.service;

import org.springframework.stereotype.Service;

@Service(value="userService")

public class UserServiceImpl implements UserService{

public void addUser() {

System.out.println("add user.");

}

}

### TxAspect.java

package aop.aspect;

import org.springframework.stereotype.Component;

@Component //切面

public class TxAspect {

//通知

public void before(){

System.out.println("我是一个通知方法 before。");

}

}

### applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

<context:component-scan base-package="aop.service,aop.aspect"/>

<!-- AOP配置 -->

<aop:config>

<!-- 切点表达式进行匹配判断，within对哪个类进行处理 -->

<aop:pointcut expression="within(aop.service.UserServiceImpl)" id="pc"/>

<!-- 切面触发的执行类 -->

<aop:aspect ref="txAspect">

<!-- 切面触发的执行类的before方法 -->

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

</aop:aspect>

</aop:config>

</beans>

### 测试类

package aop;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import aop.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("aop/applicationContext.xml");

UserService userService = (UserService) context.getBean("userService");

System.out.println(userService.getClass());

userService.addUser();

}

}

### 执行结果

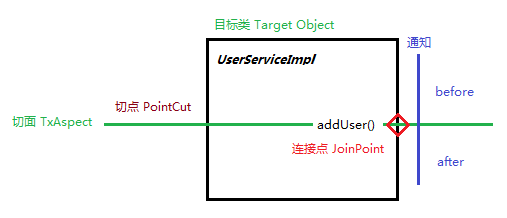
class com.sun.proxy.$Proxy6

我是一个通知方法 before。

add user.

## SpringAOP的强大

### 调用过程



我们获取对象，要么是对象地址，要么是代理，我们看是什么？

System.out.println(userService.getClass());

我们一点一点来梳理下，前面我们已经学过动态代理实现的两种方式，一种jdk动态代理，一种cglib。从打印结果我们可知，目前使用的是jdk动态代理。

UserService userService = (UserService) context.getBean("userService");

执行时spring已经把UserService变成代理对象。在执行userService.addUser();业务方法时就会被invoke方法拦截，就被切点表达式拦截。within(aop.service

.UserServiceImpl)匹配上了实现类，就会执行切面中通知。

<aop:aspect ref="txAspect">

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

</aop:aspect>

就找到TxAspect类，找到before方法，并调用。

也就是说只要程序满足切点表达式，就会调用切面的方法。最终我们就看到它不是真实的对象，而是一个代理对象。由代理对象去执行就会响应对象的新闻，去执行额外的通知方法。

### 松耦合

整个过程可以看到用写代码吗？不用写，只要满足切面要求就会自动调用。那意味着什么呢？意味着松耦合，业务代码和增强代码完全分离。首先不影响程序的正常运行，如果有某些特定的任务，或者操作。可以通过切面进行编辑，在切面中编写特定的代码完成功能，同时不影响代码的执行。再者，说不要就不要，只要去掉within中的UserServicImpl，不匹配就不会进行aop的处理。

PersonServiceImpl.java

package aop.service;

import org.springframework.stereotype.Service;

@Service(value="personService")

public class PersonServiceImpl{

public void addUser() {

System.out.println("add person.");

}

}

修改配置文件applicationContext.xml

expression="within(aop.service.PersonServiceImpl)"

执行结果

class aop.service.UserServiceImpl

add user.

说明切点不匹配则不会创建代理，获得的对象就是真实对象。

### 代理对象的生成策略

UserServiceImpl实现了UserService接口

PersonServiceImpl只是一个实现类，没有接口

#### 测试类

package aop;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import aop.service.PersonServiceImpl;

import aop.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("aop/applicationContext.xml");

UserService userService = (UserService) context.getBean("userService");

System.out.println(userService.getClass());

userService.addUser();

PersonServiceImpl person = (PersonServiceImpl) context.getBean("personService");

System.out.println(person.getClass());

person.addUser();

}

}

#### 执行结果

class com.sun.proxy.$Proxy6

我是一个通知方法 before。

add user.

class aop.service.PersonServiceImpl$$EnhancerByCGLIB$$26c24b6d

我是一个通知方法 before。

add person.

#### 小结

可以看到有接口spring采用jdk动态代理方式创建代理对象，没有接口采用cglib动态代理方式创建代理对象，如果不匹配就是真实对象。

当从容器中获取对象时，如果获取的对象满足切入点表达式，那么就会为其创建代理对象，代理对象指定方法就会执行与切入点绑定的通知方法。

进入切面的前提条件是必须有代理对象。

## AspectJ切点表达式

AspectJ是AOP一种实现，主要原理是用asm做字节码替换来达到AOP的目的。

### 切入点指示符

用来指示切入点表达式目的，SpringAOP支持的AspectJ切入点指示符如下：

|  |  |  |
| --- | --- | --- |
|  | **指示符** | **作用** |
|  | within | 用于匹配指定类型内的方法执行 |
|  | execution | 用于匹配方法执行的连接点 |
|  | this | 用于匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口 |
|  | target | 用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口 |
|  | args | 用于匹配当前执行的方法传入的参数为指定类型的执行方法 |
|  | @within | 用于匹配所以持有指定注解类型内的方法 |
|  | @target | 用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解 |
|  | @args | 用于匹配当前执行的方法传入的参数持有指定注解的执行 |
|  | @annotation | 用于匹配当前执行方法持有指定注解的方法 |
|  | bean | Spring AOP扩展的，AspectJ没有对于指示符，用于匹配特定名称的Bean对象的执行方法 |
|  | reference pointcut | 表示引用其他命名切入点，只有@ApectJ风格支持，Schema风格不支持 |

AspectJ切入点支持的切入点指示符还有： call、get、set、preinitialization、staticinitialization、initialization、handler、adviceexecution、withincode、cflow、cflowbelow、if、@this、@withincode；但Spring AOP目前不支持这些指示符，使用这些指示符将抛出IllegalArgumentException异常。这些指示符Spring AOP可能会在以后进行扩展。

AspectJ切点表达式非常丰富很多方式，这里我们只简单了解，最常用的两种方式。

### within表达式

within类级别，粗粒度的控制：

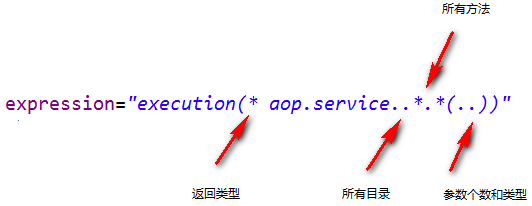
|  |  |
| --- | --- |
| **within(aop.service.UserServiceImpl))** | **指定类，只能指定一个类** |
| **within(aop.service.\*))** | 只包括当前目录下的类 |
| **within(aop.service.\*.\*))** | 只包括当前目录下的类 |
| **within(aop.service..\*))** | 指定当前目录包含所有子目录中的类 |

### execution表达式

execution方法级别，细粒度的控制：

语法：execution(返回值类型 包名.类名.方法名(参数列表))

|  |  |
| --- | --- |
| **execution(void aop.service.UserServiceImpl.addUser())** | **匹配方法** |
| **execution(void aop.service.PersonServiceImpl.addUser(String))** | 方法参数必须为字符串 |
| **execution(\* aop.service..\*.\*(..))** | 万能配置 |



匹配更加灵活多样，匹配五部分：返回类型、包路径、类、方法、参数个数和类型，所以项目中普遍都使用这种方式。

### 问题：如果有方法匹配有方法不匹配会创建代理类吗？

PersonServiceImpl.java

package aop.service.p.q;

import org.springframework.stereotype.Service;

@Service(value="personService")

public class PersonServiceImpl{

public String addUser(String name) {

System.out.println("add person."+name);

return name;

}

public void updateUser(){

System.out.println("update person");

}

}

配置只扫描addUser方法

expression="execution(\* aop.service..\*.addUser(..))"

显然addUser匹配，updateUser就不匹配。

执行结果：

class com.sun.proxy.$Proxy7

我是一个通知方法 before。

add user.

class aop.service.p.q.PersonServiceImpl$$EnhancerByCGLIB$$651e9f41

我是一个通知方法 before。

add person.tony

update person

但我们发现这个对象不是真实对象，而是代理对象，那这个现象说明什么？

一个类有多个方法，虽然updateUser不满足，但addUser满足，所以这个对象依然是代理对象，但是执行updateUser时和切点表达式不匹配，所以它虽然是代理对象，但不会执行通知方法。

所以代理对象是按类创建，不是根据方法来创建的。

## 通知Advice

### 五种通知类型

Spring和其他JavaAOP框架一样，在运行时完成织入。

通知类型：

|  |  |  |
| --- | --- | --- |
|  | **指示符** | **作用** |
|  | 前置通知 before | 在业务代码执行前执行 |
|  | 后置通知  after returning | 在业务代码执行后执行，并且不抛出异常，正常返回 |
|  | 异常通知  after throwing | 在业务代码执行时异常 |
|  | 最终通知  after (finally) | 在业务代码执行后，不论正常还是异常 |
|  | 环绕通知 around | 这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行业务或直接返回它自己的返回值或抛出异常来结束执行。自主权力非常大。 |

前四种方式不能影响目标方法的执行，而环绕通知可以改变这一切！

### 生命周期

* 前置通知在调用方法前，执行方法时就失效了。
* 后置通知在调用方法后，执行方法前它还无法生效。
* 异常通知只在异常时才有效，最终通知只在方法执行完才有效。

### 环绕通知

#### TxAspect.java

package aop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.stereotype.Component;

@Component //切面

public class TxAspect {

//前置通知

public void before(){

System.out.println("我是一个通知方法 before。");

}

//环绕通知

public Object around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

//没有下面的语句，没有返回值会怎样？ 业务执行被改变

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

return result;

}

}

#### applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

<context:component-scan base-package="aop.service,aop.aspect"/>

<!-- AOP配置 -->

<aop:config>

<!-- 切点表达式进行匹配判断，within对哪个类进行处理 -->

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

<!-- 切面触发的执行类 -->

<aop:aspect ref="txAspect">

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

</aop:aspect>

</aop:config>

</beans>

#### 测试类

package aop;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import aop.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("aop/applicationContext.xml");

UserService userService = (UserService) context.getBean("userService");

System.out.println(userService.getClass());

userService.addUser();

}

}

#### 执行结果

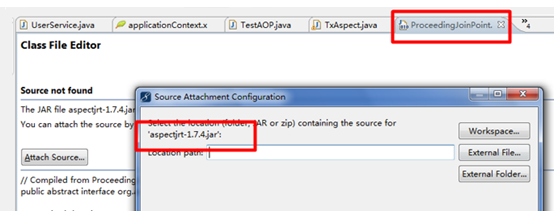
class com.sun.proxy.$Proxy7

环绕通知开始

add user.

环绕通知结束

### ProceedingJoinPoint和JoinPoint的区别



public interface ProceedingJoinPoint extends JoinPoint {

#### 只支持环绕通知

TxAspect.java

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.stereotype.Component;

@Component //切面

public class TxAspect {

//前置通知，不能是ProceedingJoinPoint，其只能用于环绕通知

public void before(JoinPoint joinPoint){

System.out.println("我是一个通知方法 before。");

//获取目标对象的类型

Class targetClass = joinPoint.getTarget().getClass();

System.out.println("目标对象："+targetClass);

//获取目标方法

String methodName = joinPoint.getSignature().getName();

System.out.println("目标方法："+methodName);

}

//环绕通知

public Object around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

return result;

}

}

applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

<context:component-scan base-package="aop.service,aop.aspect"/>

<!-- AOP配置 -->

<aop:config>

<!-- 切点表达式进行匹配判断，within对哪个类进行处理 -->

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

<!-- 切面触发的执行类 -->

<aop:aspect ref="txAspect">

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

</aop:aspect>

</aop:config>

</beans>

执行结果

class com.sun.proxy.$Proxy7

我是一个通知方法 before。

目标对象：class aop.service.UserServiceImpl

目标方法：addUser

add user.

执行报错

如果使用ProceedingJoinPoint而不是JoinPoint报下列错误

Caused by: org.springframework.beans.BeanInstantiationException: Could not instantiate bean class [org.springframework.aop.aspectj.AspectJPointcutAdvisor]: Constructor threw exception; nested exception is java.lang.IllegalArgumentException: ProceedingJoinPoint is only supported for around advice

at org.springframework.beans.BeanUtils.instantiateClass(BeanUtils.java:163)

### 异常通知

增加修改方法调用，故意抛出错误

#### UserService.java

package aop.service;

public interface UserService {

public void addUser();

public void updateUser();

}

#### UserServiceImpl.java

package aop.service;

import org.springframework.stereotype.Service;

@Service(value="userService")

public class UserServiceImpl implements UserService{

public void addUser() {

System.out.println("add user.");

}

public void updateUser(){

System.out.println("update user");

int i = 1/0;

}

}

#### TxAspect.java

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.stereotype.Component;

@Component //切面

public class TxAspect {

//前置通知，不能是ProceedingJoinPoint，其只能用于环绕通知

public void before(JoinPoint joinPoint){

System.out.println("我是一个通知方法 before。");

//获取目标对象的类型

Class targetClass = joinPoint.getTarget().getClass();

System.out.println("目标对象："+targetClass);

//获取目标方法

String methodName = joinPoint.getSignature().getName();

System.out.println("目标方法："+methodName);

}

//环绕通知

public Object around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

return result;

}

//异常通知

public void afterThrowing(JoinPoint joinPoint, Throwable throwable){

System.out.println("异常通知");

System.out.println("异常类："+throwable.getClass());

System.out.println("异常信息："+throwable.getMessage());

}

}

#### applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

<context:component-scan base-package="aop.service,aop.aspect"/>

<!-- AOP配置 -->

<aop:config>

<!-- 切点表达式进行匹配判断，within对哪个类进行处理 -->

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

<!-- 切面触发的执行类 -->

<aop:aspect ref="txAspect">

<aop:after-throwing method="afterThrowing" throwing="throwable" pointcut-ref="pc"/>

</aop:aspect>

</aop:config>

</beans>

注意：

* 必须加throwing属性，名称必须和TxAspect中方法的参数名称一致。
* JoinPoint 必须是第一个参数

异常都是java.lang.IllegalArgumentException: error at ::0 formal unbound in pointcut。

#### TestSpring.java

package aop;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import aop.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("aop/applicationContext.xml");

UserService userService = (UserService) context.getBean("userService");

System.out.println(userService.getClass());

userService.addUser();

userService.updateUser();

}

}

#### 执行结果

class com.sun.proxy.$Proxy7

add user.

update user

异常通知

异常类：class java.lang.ArithmeticException

异常信息：/ by zero

# 第四天：通知+AOP

## 通知Advice

### 后置通知

#### UserService.java

package aop.service;

public interface UserService {

public void addUser();

public void updateUser();

public String getUsername();

}

#### UserServiceImpl.java

package aop.service;

import org.springframework.stereotype.Service;

@Service(value="userService")

public class UserServiceImpl implements UserService{

public void addUser() {

System.out.println("add user.");

}

public void updateUser(){

System.out.println("update user");

int i = 1/0;

}

public String getUsername() {

return "tony";

}

}

#### TxAspect.java

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.stereotype.Component;

@Component //切面

public class TxAspect {

//前置通知，不能是ProceedingJoinPoint，其只能用于环绕通知

public void before(JoinPoint joinPoint){

System.out.println("我是一个通知方法 before。");

//获取目标对象的类型

Class targetClass = joinPoint.getTarget().getClass();

System.out.println("目标对象："+targetClass);

//获取目标方法

String methodName = joinPoint.getSignature().getName();

System.out.println("目标方法："+methodName);

}

//环绕通知

public Object around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

return result;

}

//异常通知

public void afterThrowing(JoinPoint joinPoint, Throwable throwable){

System.out.println("异常通知");

System.out.println("异常类："+throwable.getClass());

System.out.println("异常信息："+throwable.getMessage());

}

//后置通知

public String afterReturning(String msg){

System.out.println("后置通知");

System.out.println("目标对象返回结果："+msg);

return msg;

}

}

#### applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

<context:component-scan base-package="aop.service,aop.aspect"/>

<!-- AOP配置 -->

<aop:config>

<!-- 切点表达式进行匹配判断，within对哪个类进行处理 -->

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

<!-- 切面触发的执行类 -->

<aop:aspect ref="txAspect">

<aop:after-returning method="afterReturning" returning="msg" pointcut-ref="pc"/>

</aop:aspect>

</aop:config>

</beans>

注意：设置return属性，名称和切面中的方法参数名称相同。

#### TestSpring.java

package aop;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import aop.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("aop/applicationContext.xml");

UserService userService = (UserService) context.getBean("userService");

System.out.println(userService.getClass());

String name = userService.getUsername();

System.out.println(name);

}

}

#### 执行结果

class com.sun.proxy.$Proxy7

后置通知

目标对象返回结果：tony

tony

### 环绕加后置

#### TxAspect.java

环绕时不返回值

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.stereotype.Component;

@Component //切面

public class TxAspect {

//前置通知，不能是ProceedingJoinPoint，其只能用于环绕通知

public void before(JoinPoint joinPoint){

System.out.println("我是一个通知方法 before。");

//获取目标对象的类型

Class targetClass = joinPoint.getTarget().getClass();

System.out.println("目标对象："+targetClass);

//获取目标方法

String methodName = joinPoint.getSignature().getName();

System.out.println("目标方法："+methodName);

}

//环绕通知

public void around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

}

//异常通知

public void afterThrowing(JoinPoint joinPoint, Throwable throwable){

System.out.println("异常通知");

System.out.println("异常类："+throwable.getClass());

System.out.println("异常信息："+throwable.getMessage());

}

//后置通知

public String afterReturning(String msg){

System.out.println("后置通知");

System.out.println("目标对象返回结果："+msg);

return msg;

}

}

#### applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

<context:component-scan base-package="aop.service,aop.aspect"/>

<!-- AOP配置 -->

<aop:config>

<!-- 切点表达式进行匹配判断，within对哪个类进行处理 -->

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

<!-- 切面触发的执行类 -->

<aop:aspect ref="txAspect">

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

<aop:after-returning method="afterReturning" returning="msg" pointcut-ref="pc"/>

</aop:aspect>

</aop:config>

</beans>

注意：环绕在前后置再后，环绕不返回则影响后置取值；如果环绕在后置之后呢？那就不影响其取值。

#### 执行结果

环绕在前后置在后

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

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

class com.sun.proxy.$Proxy7

环绕通知开始

环绕通知结束

后置通知

目标对象返回结果：null

null

后置在前环绕在后

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

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

class com.sun.proxy.$Proxy7

环绕通知开始

后置通知

目标对象返回结果：tony

环绕通知结束

null

### 两个环绕通知

最大的问题是，每个环绕通知中都会调用业务的代码，那两个环绕通知，会不会调用两次呢？如果是，业务被执行两次，这样符合逻辑吗？

#### AroundA.java

package aop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.stereotype.Component;

@Component //切面

public class AroundA {

public Object arounda(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("around a start");

Object result = joinPoint.proceed();

System.out.println("around a stop");

return result;

}

}

#### AroundB.java

package aop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.stereotype.Component;

@Component //切面

public class AroundB {

public Object aroundb(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("around b start");

Object result = joinPoint.proceed();

System.out.println("around b stop");

return result;

}

}

#### applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

<context:component-scan base-package="aop.service,aop.aspect"/>

<!-- AOP配置 -->

<aop:config>

<!-- 切点表达式进行匹配判断，within对哪个类进行处理 -->

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

<aop:aspect ref="aroundA">

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

</aop:aspect>

<aop:aspect ref="aroundB">

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

</aop:aspect>

</aop:config>

</beans>

#### 执行结果

class com.sun.proxy.$Proxy7

around a start

around b start

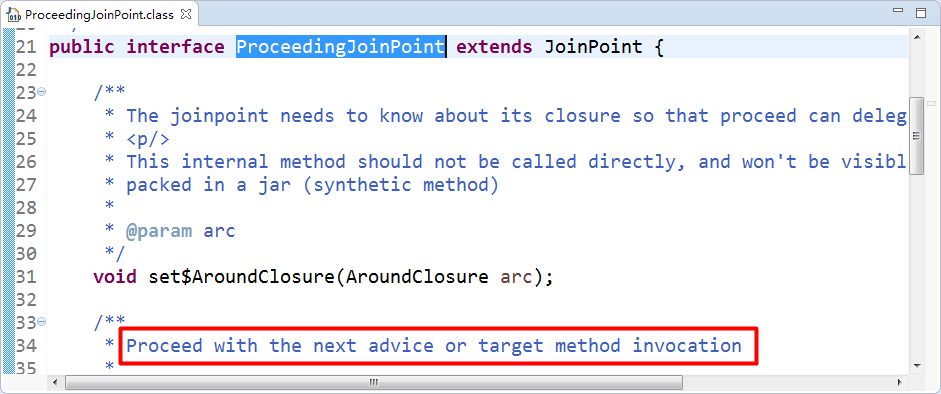
add user.

around b stop

around a stop

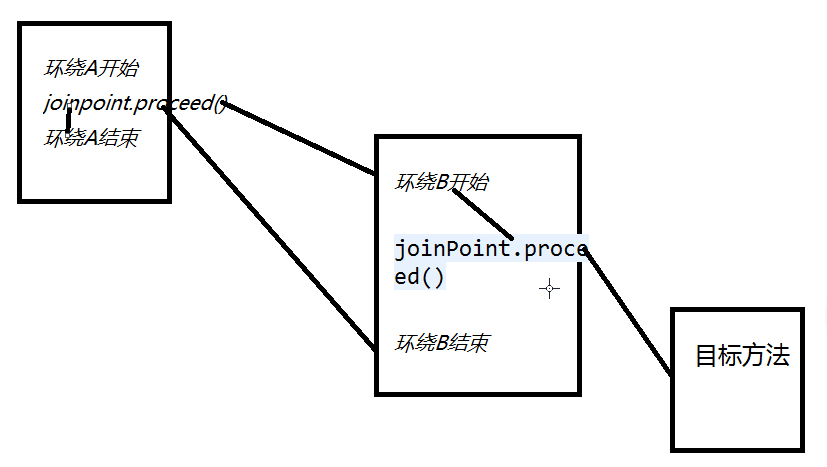
什么顺序呢？先执行A再执行B，那目标方法执行几回？Spring只调用了一次。如果真执行两次就乱套了。那它是如何做到的呢？

查看ProceedingJoinPoint源码34行可以看的非常清晰



原来如此，Spring非常聪明，非常智能，它再执行业务方法调用时会先判断如果还有通知，本通知就不调用业务方法，知道是最里面的一个通知时才会调用业务方法。

知道了这个原则，那我们再加C，再加D，它们就只是嵌套执行，而不会多次执行业务方法。



### 最终通知

#### UserServiceImpl.java

package aop.service;

import org.springframework.stereotype.Service;

@Service(value="userService")

public class UserServiceImpl implements UserService{

public void addUser() {

System.out.println("add user.");

int i = 1/0; //故意抛错，最终通知依然执行

}

public void updateUser(){

System.out.println("update user");

int i = 1/0;

}

public String getUsername() {

return "tony";

}

}

#### TxAspect.java

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.stereotype.Component;

@Component //切面

public class TxAspect {

//前置通知，不能是ProceedingJoinPoint，其只能用于环绕通知

public void before(JoinPoint joinPoint){

System.out.println("我是一个通知方法 before。");

//获取目标对象的类型

Class targetClass = joinPoint.getTarget().getClass();

System.out.println("目标对象："+targetClass);

//获取目标方法

String methodName = joinPoint.getSignature().getName();

System.out.println("目标方法："+methodName);

}

//环绕通知

public void around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

}

//异常通知

public void afterThrowing(JoinPoint joinPoint, Throwable throwable){

System.out.println("异常通知");

System.out.println("异常类："+throwable.getClass());

System.out.println("异常信息："+throwable.getMessage());

}

//后置通知

public String afterReturning(String msg){

System.out.println("后置通知");

System.out.println("目标对象返回结果："+msg);

return msg;

}

//最终通知

public void after(){

System.out.println("最终通知");

}

}

#### applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

<context:component-scan base-package="aop.service,aop.aspect"/>

<!-- AOP配置 -->

<aop:config>

<!-- 切点表达式进行匹配判断，within对哪个类进行处理 -->

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

<aop:aspect ref="txAspect">

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

</aop:aspect>

</aop:config>

</beans>

注意：环绕在前后置再后，环绕不返回则影响后置取值；如果环绕在后置之后呢？那就不影响其取值。

#### 执行结果

class com.sun.proxy.$Proxy7

add user.

最终通知

而junit已经抛出错误

## AOP注解方式

在类和方法上使用：

* @Before 前置通知
* @Around 环绕通知
* @AfterThrowing 异常通知
* @AfterReturning 后置通知
* @After 最终通知

### 准备环境

创建aopannotation java工程

创建lib目录，拷贝jar包，所有jar BuilderPath引入jar包

aopalliance-1.0.jar

aspectjrt-1.7.4.jar

aspectjweaver-1.7.4.jar

commons-logging-1.0.4.jar

spring-aop-3.2.3.RELEASE-sources.jar

spring-aop-3.2.3.RELEASE.jar

spring-aspects-3.2.3.RELEASE-sources.jar

spring-aspects-3.2.3.RELEASE.jar

spring-beans-3.2.3.RELEASE.jar

spring-context-3.2.3.RELEASE-sources.jar

spring-context-3.2.3.RELEASE.jar

spring-context-support-3.2.3.RELEASE-sources.jar

spring-context-support-3.2.3.RELEASE.jar

spring-core-3.2.3.RELEASE-sources.jar

spring-core-3.2.3.RELEASE.jar

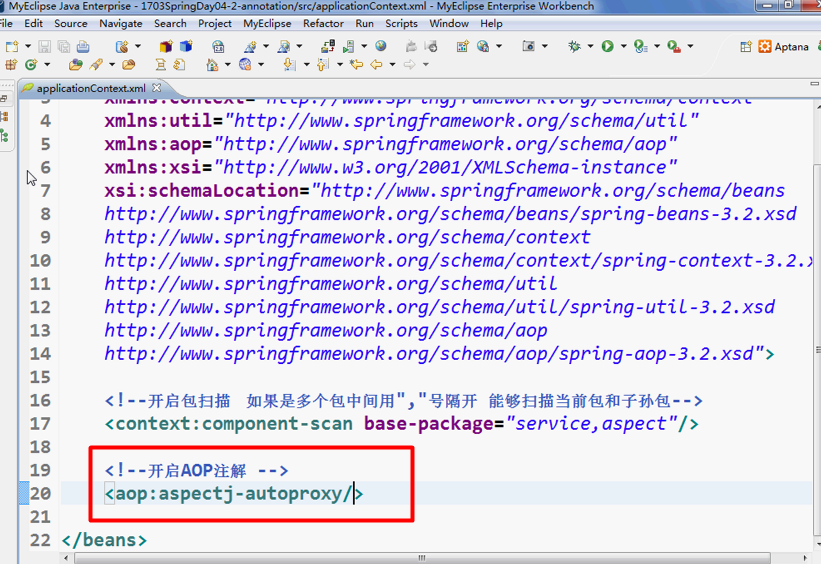
spring-expression-3.2.3.RELEASE-sources.jar

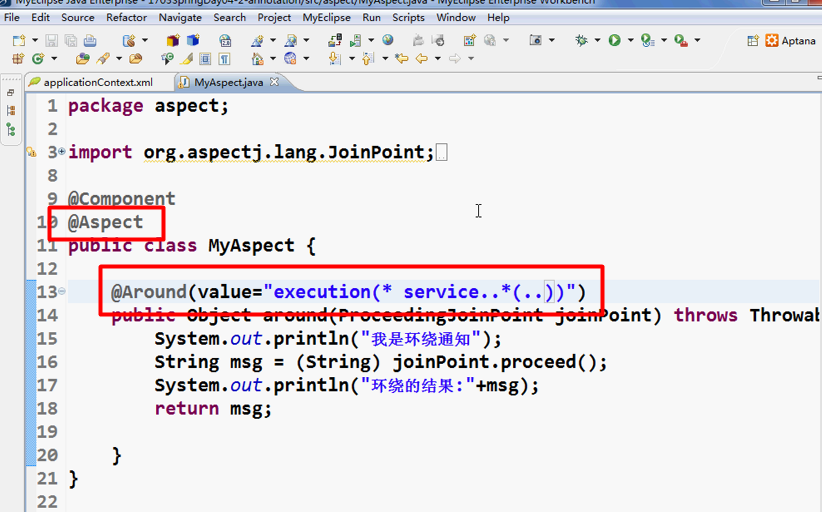
spring-expression-3.2.3.RELEASE.jar

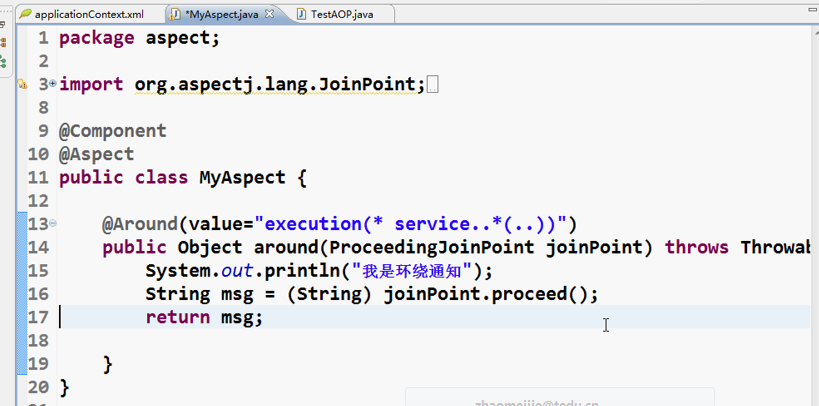
spring-test-3.2.3.RELEASE-sources.jar

spring-test-3.2.3.RELEASE.jar

### 环绕通知









#### UserService.java

package aop.service;

public interface UserService {

public void addUser();

public void updateUser();

}

#### UserServiceImpl.java

package aop.service;

import org.springframework.stereotype.Service;

@Service(value="userService")

public class UserServiceImpl implements UserService{

public void addUser() {

System.out.println("add user.");

}

public void updateUser(){

System.out.println("update user");

int i = 1/0;

}

}

#### TxAspect.java

package aop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.springframework.stereotype.Component;

@Component

@Aspect //切面

public class TxAspect {

@Around(value="execution(\* aop.service..\*(..))") //环绕通知

public Object around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

return result;

}

}

#### applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

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

<!-- 开启AOP注解 -->

<aop:aspectj-autoproxy/>

</beans>

几乎实现xml零配置

#### aspectj-autoproxy

通过aop命名空间的<aop:aspectj-autoproxy />声明自动为spring容器中那些配置@aspectJ切面的bean创建代理，织入切面。当然，spring在内部依旧采用AnnotationAwareAspectJAutoProxyCreator进行自动代理的创建工作，但具体实现的细节已经被<aop:aspectj-autoproxy />隐藏起来了<aop:aspectj-autoproxy />有一个proxy-target-class属性，默认为false，表示使用jdk动态代理织入增强，当配为<aop:aspectj-autoproxy poxy-target-class="true"/>时，表示使用CGLib动态代理技术织入增强。不过即使proxy-target-class设置为false，如果目标类没有声明接口，则spring将自动使用CGLib动态代理。

#### TestSpring.java

package aop;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import aop.service.UserService;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("aop/applicationContext.xml");

UserService userService = (UserService) context.getBean("userService");

System.out.println(userService.getClass());

userService.addUser();

}

}

业务调用方式未变

#### 执行结果

class com.sun.proxy.$Proxy9

环绕通知开始

add user.

环绕通知结束

### 前置通知

#### TxAspect.java

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.springframework.stereotype.Component;

@Component

@Aspect //切面

public class TxAspect {

@Around(value="execution(\* aop.service..\*(..))") //环绕通知

public Object around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

return result;

}

@Before(value="execution(\* aop.service..\*(..))") //前置通知

public void before(JoinPoint joinPoint){

System.out.println("我是一个通知方法 before。");

//获取目标对象的类型

Class targetClass = joinPoint.getTarget().getClass();

System.out.println("目标对象："+targetClass);

//获取目标方法

String methodName = joinPoint.getSignature().getName();

System.out.println("目标方法："+methodName);

}

}

#### 执行结果

class com.sun.proxy.$Proxy10

环绕通知开始

我是一个通知方法 before。

目标对象：class aop.service.UserServiceImpl

目标方法：addUser

add user.

环绕通知结束

### 异常通知

#### UserServiceImpl.java

package aop.service;

import org.springframework.stereotype.Service;

@Service(value="userService")

public class UserServiceImpl implements UserService{

public void addUser() {

System.out.println("add user.");

int i = 1/0; //故意抛出异常

}

public void updateUser(){

System.out.println("update user");

}

public String getUsername() {

return "tony";

}

}

#### TxAspect.java

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

import org.springframework.stereotype.Component;

@Component

@Aspect //切面

public class TxAspect {

@AfterThrowing(value="execution(\* aop.service..\*(..))", throwing="throwable") //异常通知

public void afterThrowing(JoinPoint joinPoint, Throwable throwable){

System.out.println("异常通知");

System.out.println("异常类："+throwable.getClass());

System.out.println("异常信息："+throwable.getMessage());

}

}

#### 执行结果

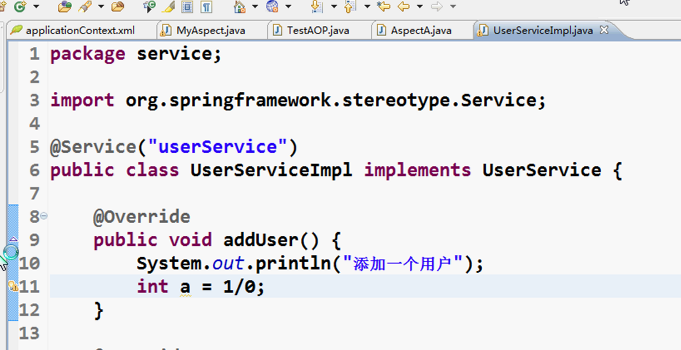
class com.sun.proxy.$Proxy9

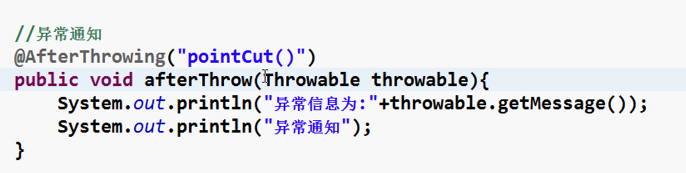
add user.

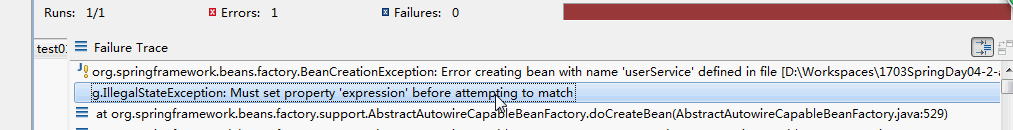
异常通知

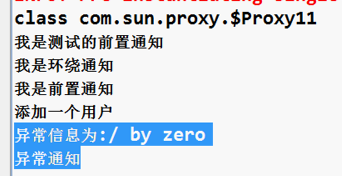
异常类：class java.lang.ArithmeticException

异常信息：/ by zero







### 全局切入点

#### TxAspect.java

下面代码每个方法都写个切点表达式太麻烦，这样写还没aop的xml配置方式简单，那注解方式有没有简单的写法？

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.springframework.stereotype.Component;

@Component

@Aspect //切面

public class TxAspect {

@Around(value="execution(\* aop.service..\*(..))") //环绕通知

public Object around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

return result;

}

@Before(value="execution(\* aop.service..\*(..))") //前置通知

public void before(JoinPoint joinPoint){

System.out.println("我是一个通知方法 before。");

//获取目标对象的类型

Class targetClass = joinPoint.getTarget().getClass();

System.out.println("目标对象："+targetClass);

//获取目标方法

String methodName = joinPoint.getSignature().getName();

System.out.println("目标方法："+methodName);

}

@AfterThrowing(value="execution(\* aop.service..\*(..))", throwing="throwable") //异常通知

public void afterThrowing(JoinPoint joinPoint, Throwable throwable){

System.out.println("异常通知");

System.out.println("异常类："+throwable.getClass());

System.out.println("异常信息："+throwable.getMessage());

}

}

#### 修改TxAspect.java

新增一个方法使用@Pointcut注解标识，后面都通过函数名称引用pointcut()。

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.stereotype.Component;

@Component

@Aspect //切面

public class TxAspect {

//全局切入点

@Pointcut(value="execution(\* aop.service..\*(..))")

public void pointcut(){

}

@Around(value="pointcut()") //环绕通知

public Object around(ProceedingJoinPoint joinPoint) throws Throwable{

System.out.println("环绕通知开始");

Object result = joinPoint.proceed(); //执行目标方法

System.out.println("环绕通知结束");

return result;

}

@Before(value="pointcut()") //前置通知

public void before(JoinPoint joinPoint){

System.out.println("我是一个通知方法 before。");

//获取目标对象的类型

Class targetClass = joinPoint.getTarget().getClass();

System.out.println("目标对象："+targetClass);

//获取目标方法

String methodName = joinPoint.getSignature().getName();

System.out.println("目标方法："+methodName);

}

@AfterThrowing(value="pointcut()", throwing="throwable") //异常通知

public void afterThrowing(JoinPoint joinPoint, Throwable throwable){

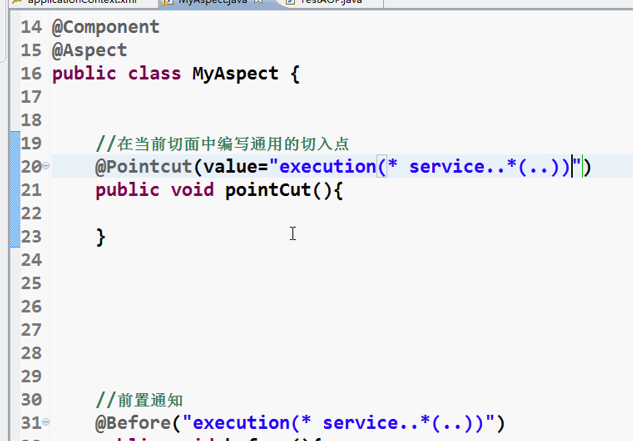
System.out.println("异常通知");

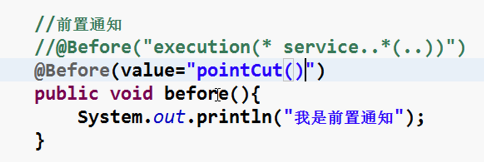
System.out.println("异常类："+throwable.getClass());

System.out.println("异常信息："+throwable.getMessage());

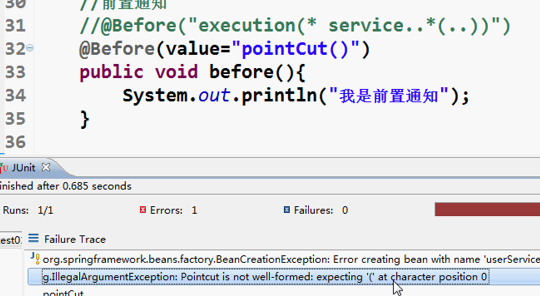
}

}





必须是pointCut()，如果去掉括号报错



从上面写法我们也能知道，这个方式只是在本类里共享，如果再有类，还得写这段话。那如果多个怎么办？那就写个父类，用继承方式来解决。

## 自定义注解

Java注解是附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。包含在 java.lang.annotation包中。元注解是指注解的注解。包括 @Retention @Target @Document @Inherited四种。

### 元注解

@Document 说明该注解将被包含在javadoc中

@Inherited 说明子类可以继承父类中的该注解

@Retention 定义注解的保留策略

@Target 定义注解的作用目标

### @Retention

@Retention(RetentionPolicy.SOURCE) //注解仅存在于源码中，在class字节码文件中不包含

@Retention(RetentionPolicy.CLASS) // 默认的保留策略，注解会在class字节码文件中存在，但运行时无法获得

@Retention(RetentionPolicy.RUNTIME) // 注解会在class字节码文件中存在，在运行时可以通过反射获取到

### @Target

@Target(ElementType.TYPE) 接口、类、枚举、注解

@Target(ElementType.FIELD) 字段、枚举的常量

@Target(ElementType.METHOD) 方法

@Target(ElementType.PARAMETER) 方法参数

@Target(ElementType.CONSTRUCTOR) 构造函数

@Target(ElementType.LOCAL\_VARIABLE) 局部变量

@Target(ElementType.ANNOTATION\_TYPE) 注解

@Target(ElementType.PACKAGE) 包

由以上的源码可以知道，他的elementType 可以有多个，一个注解可以为类的，方法的，字段的等等

### 示例

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface PrivilegeInfo {

public String name();

}

## AOP应用场景

### 异常

异常处理类和业务逻辑类完全松耦合。时刻捕获生产生产环境中所有的错误，实时监控该系统，异常收集。

需求：记录程序异常，哪个类，出现的什么异常。

#### UserDao.java

package aop.dao;

public interface UserDao {

public void addUser();

public void updateUser();

public String getUsername();

}

#### UserDaoImpl.java

package aop.dao;

import org.springframework.stereotype.Repository;

@Repository("userDao")

public class UserDaoImpl implements UserDao{

public void addUser() {

System.out.println("新增用户");

}

public void updateUser() {

System.out.println("修改用户");

}

public String getUsername() {

return "tony";

}

}

#### UserService.java

package aop.service;

public interface UserService {

public void addUser();

public void updateUser();

public String getUsername();

}

#### UserServiceImpl.java

package aop.service;

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

import org.springframework.stereotype.Service;

import aop.dao.UserDao;

@Service(value="userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

public void addUser() {

userDao.addUser();

int i = 1/0;

}

public void updateUser(){

userDao.updateUser();

}

public String getUsername() {

return userDao.getUsername();

}

}

#### UserController.java

package aop.controller;

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

import org.springframework.stereotype.Controller;

import aop.service.UserService;

@Controller

public class UserController {

@Autowired

private UserService userService;

public void addUser(){

userService.addUser();

}

}

#### ThrowAspect.java

package aop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.stereotype.Component;

@Component

@Aspect //切面

public class ThrowAspect {

//全局切入点

@Pointcut(value="execution(\* aop.service..\*(..))")

public void pointcut(){

}

@AfterThrowing(value="pointcut()", throwing="throwable") //异常通知

public void afterThrowing(JoinPoint joinPoint, Throwable throwable){

System.out.println("异常通知");

System.out.println("异常类："+throwable.getClass());

System.out.println("异常信息："+throwable.getMessage());

}

}

#### applicationContext.xml

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

">

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

<!-- 开启AOP注解 -->

<aop:aspectj-autoproxy/>

</beans>

#### TestSpring.java

package aop;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import aop.controller.UserController;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("aop/applicationContext.xml");

UserController user = (UserController) context.getBean("userController");

System.out.println(user.getClass());

user.addUser();

}

}

### 性能监控

需求：计算service的各个类的各个方法的执行时间，也就是耗时

打印信息包括：类名、方法名、耗时

分析得出这个需求只能使用环绕通知的方式，要在业务执行前记录开始时间，在业务执行后记录完成时间，两个时间相减就是耗时，耗时过长的业务可能就存在性能问题。

#### UserDaoImpl.java

先去掉异常

package aop.service;

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

import org.springframework.stereotype.Service;

import aop.dao.UserDao;

@Service(value="userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

public void addUser() {

userDao.addUser();

//int i = 1/0;

}

public void updateUser(){

userDao.updateUser();

}

public String getUsername() {

return userDao.getUsername();

}

}

#### TimeAspect.java

注意只对service层方法监控

package aop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.stereotype.Component;

@Component

@Aspect

public class TimeAspect {

@Pointcut(value="execution(\* aop.service.\*.\*(..))")

public void pointcut(){

}

@Around(value="pointcut()")

public Object time(ProceedingJoinPoint joinPoint) throws Throwable{

long startTime = System.currentTimeMillis();

Object result = joinPoint.proceed();

long stopTime = System.currentTimeMillis();

System.out.println("执行对象："+joinPoint.getTarget().getClass());

System.out.println("执行方法："+joinPoint.getSignature().getName());

System.out.println("执行时间："+(stopTime - startTime)+"毫秒数");

return result;

}

}

#### 执行结果

class aop.controller.UserController

新增用户

执行对象：class aop.service.UserServiceImpl

执行方法：addUser

执行时间：1毫秒数

#### 对所有的类进行监控

修改配置

@Pointcut(value="execution(\* aop..\*(..))")

执行结果

class aop.controller.UserController$$EnhancerByCGLIB$$32304be2

新增用户

执行对象：class aop.dao.UserDaoImpl

执行方法：addUser

执行时间：0毫秒数

执行对象：class aop.service.UserServiceImpl

执行方法：addUser

执行时间：5毫秒数

执行对象：class aop.controller.UserController

执行方法：addUser

执行时间：64毫秒数

#### 只监控controller类

修改配置

@Pointcut(value="execution(\* aop.controller..\*(..))")

获取两个UserControllre

package aop;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import aop.controller.UserController;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("aop/applicationContext.xml");

UserController user = (UserController) context.getBean("userController");

System.out.println(user.getClass());

user.addUser();

UserController user1 = (UserController) context.getBean("userController");

System.out.println(user1.getClass());

user1.addUser();

}

}

执行结果

class aop.controller.UserController$$EnhancerByCGLIB$$30aa270b

新增用户

执行对象：class aop.controller.UserController

执行方法：addUser

执行时间：28毫秒数

class aop.controller.UserController$$EnhancerByCGLIB$$30aa270b

新增用户

执行对象：class aop.controller.UserController

执行方法：addUser

执行时间：0毫秒数

第一次慢这么多，什么原因？因为第一次要创建这个对象所有慢。在实际系统中，类在容器启动时已经创建，那在调用就没有这么慢，而主要是业务执行时间。

### 权限控制

需求：对类的方法进行控制，允许的才能访问，不允许的就提示权限不足。分析得出先需要有权限控制列表，然后在切面中获取执行的方法的注解，判断其是否在允许列表中，允许则业务继续执行，不允许则不能执行业务，同时提示权限不足。

这个需求也只能使用环绕通知，因为只有环绕通知才能改变执行的行为。

#### PrivilegeInfo.java

package aop.annotation;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

//注解在方法上使用

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface PrivilegeInfo {

public String name();

}

#### AllowList.java

package aop;

import java.util.List;

//权限列表

public class AllowList {

private static List<String> allow;

public static List<String> getAllow() {

return allow;

}

public static void setAllow(List<String> allow) {

AllowList.allow = allow;

}

}

#### UserServiceImpl.java

package aop.service;

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

import org.springframework.stereotype.Service;

import aop.annotation.PrivilegeInfo;

import aop.dao.UserDao;

@Service(value="userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

//设置权限

@PrivilegeInfo(name="add")

public void addUser() {

userDao.addUser();

//int i = 1/0;

}

@PrivilegeInfo(name="update")

public void updateUser(){

userDao.updateUser();

}

public String getUsername() {

return userDao.getUsername();

}

}

#### PrivilegeAspect.java

package aop.aspect;

import java.lang.reflect.Method;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.springframework.stereotype.Component;

import aop.AllowList;

import aop.annotation.PrivilegeInfo;

@Component

@Aspect

public class PrivilegeAspect {

@Around(value="execution(\* aop..\*(..))")

public Object around(ProceedingJoinPoint joinPoint) throws Throwable{

//获取目标对象

Class targetClass = joinPoint.getTarget().getClass();

//获取方法名

String methodName = joinPoint.getSignature().getName();

System.out.println("访问："+targetClass+"."+methodName);

//获取参数列表

Object[] args = joinPoint.getArgs();

//由参数获取其类

Class[] clsArgs = new Class[args.length];

for(int i=0;i<args.length;i++){

clsArgs[i] = args[i].getClass();

}

//通过反射，获取方法

Method method = targetClass.getMethod(methodName, clsArgs);

//判断方法上是否有权限注解

if(method.isAnnotationPresent(PrivilegeInfo.class)){

//获取注解对象

PrivilegeInfo info = method.getAnnotation(PrivilegeInfo.class);

String pName = info.name(); //获取注解的name属性

//是否拥有此权限

if(AllowList.getAllow().contains(pName)){

System.out.println("恭喜，你有权限访问!");

Object result = joinPoint.proceed();

return result;

}else{

System.out.println("抱歉，你权限不足!");

return null;

}

}else{ //无需校验的直接通过

System.out.println("无需权限，直接访问！");

Object result = joinPoint.proceed();

return result;

}

}

}

#### TestSpring.java

package aop;

import java.util.ArrayList;

import java.util.List;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import aop.controller.UserController;

public class TestSpring {

@Test

public void test(){

//启动spring容器

ApplicationContext context =

new ClassPathXmlApplicationContext("aop/applicationContext.xml");

//权限集合

List<String> allow = new ArrayList<String>();

allow.add("add");

allow.add("delete");

allow.add("get");

AllowList.setAllow(allow);

UserController user = (UserController) context.getBean("userController");

System.out.println(user.getClass());

user.addUser();

user.updateUser();

}

}

#### 执行结果

class aop.controller.UserController$$EnhancerByCGLIB$$c3ae10d

访问：class aop.controller.UserController.addUser

无需权限，直接访问！

访问：class aop.service.UserServiceImpl.addUser

恭喜，你有权限访问!

访问：class aop.dao.UserDaoImpl.addUser

无需权限，直接访问！

新增用户

访问：class aop.controller.UserController.updateUser

无需权限，直接访问！

访问：class aop.service.UserServiceImpl.updateUser

抱歉，你权限不足!

### 事务

#### Transaction.java

package aop.annotation;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Transaction {

}

#### UserServiceImpl.java

package aop.service;

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

import org.springframework.stereotype.Service;

import aop.annotation.PrivilegeInfo;

import aop.annotation.Transaction;

import aop.dao.UserDao;

@Service(value="userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

//设置权限

@PrivilegeInfo(name="add")

@Transaction

public void addUser() {

userDao.addUser();

int i = 1/0;

}

@PrivilegeInfo(name="update")

@Transaction

public void updateUser(){

userDao.updateUser();

}

public String getUsername() {

return userDao.getUsername();

}

}

#### TxAspect.java

package aop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

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

import org.springframework.stereotype.Component;

import aop.annotation.Transaction;

import aop.tx.TransactionManager;

@Component

@Aspect

public class TxAspect {

@Autowired

private TransactionManager tx;

@Around(value="execution(\* aop..\*(..)) && @annotation(transaction)")

public Object around(ProceedingJoinPoint joinPoint, Transaction transaction) throws Throwable{

try {

tx.begin();

Object result = joinPoint.proceed();

tx.commit();

return result;

} catch (Exception e) {

tx.rollback();

}

return null;

}

}

#### 执行结果

class aop.controller.UserController

开启事务

新增用户

事务回滚

开启事务

修改用户

事务提交

## AOP表达式高级用法

实现更加细粒度的控制。

可以通过切点表达式直接获取方法上的注解，注入过程，根据切点表达式首先会匹配满足条件的类的方法，再根据注解标签@annotation找到满足条件的方法，要求注解的类型必须与参数列表中的类型匹配，如果不匹配，则切入点不满足。

### 匹配注解

@Around(value="execution(\* aop..\*(..)) and @annotation(transaction)")

public Object around(ProceedingJoinPoint joinPoint, Transaction transaction) throws Throwable{

可以用&&符号也可以用and符号。

### 获取目标对象

@Around(value="execution(\* aop..\*(..)) and @annotation(transaction) and target(t)")

public Object around(ProceedingJoinPoint joinPoint, Transaction transaction, Object t) throws Throwable{

### 匹配参数

只有addUser(name,age)匹配，而updateUser就没有执行切面方法。

package aop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

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

import org.springframework.stereotype.Component;

import aop.annotation.Transaction;

import aop.tx.TransactionManager;

@Component

@Aspect

public class TxAspect {

@Autowired

private TransactionManager tx;

@Around(value="execution(\* aop..\*(..)) and @annotation(transaction) and target(t) and args(name,age)")

public Object around(ProceedingJoinPoint joinPoint, Transaction transaction, Object t, String name, Integer age) throws Throwable{

System.out.println(t.getClass());

System.out.println("参数："+name+" "+age);

try {

tx.begin();

Object result = joinPoint.proceed();

tx.commit();

return result;

} catch (Exception e) {

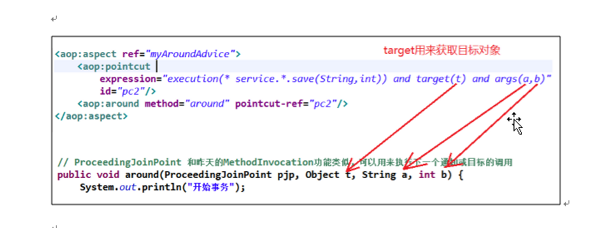
tx.rollback();

}

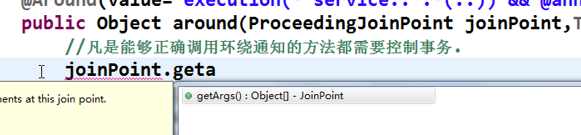
return null;

}

}



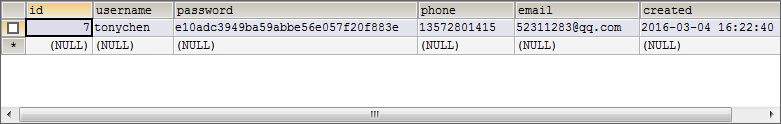
这个JoinPoint中就有



# 第五天：数据库支持+JDBCTemplate+事务

## 数据库访问

### User表



### 建表SQL语句

DROP TABLE IF EXISTS `user\_c`;

CREATE TABLE `user\_c` (

`id` varchar(40) NOT NULL,

`user\_name` varchar(30) DEFAULT NULL,

`age` int(11) DEFAULT NULL,

`address` varchar(200) DEFAULT NULL,

`dept\_id` int(11) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

/\*Data for the table `user\_c` \*/

insert into `user\_c`(`id`,`user\_name`,`age`,`address`,`dept\_id`) values ('1','夏言',73,'桂州村',1),('2','严嵩',87,'分宜县城介桥村',1),('3','徐阶',80,'明松江府华亭县',2),('4','高拱',66,'河南省新郑市高老庄村',2),('5','张居正',58,'江陵',2);

### JDBC

JDBC（Java DataBase Connectivity java数据库连接）是一种用于执行SQL语句的Java API，可以为多种关系数据库提供统一访问，它由一组用Java语言编写的类和接口组成。JDBC提供了一种基准，据此可以构建更高级的工具和接口，使数据库开发人员能够编写数据库应用程序。

#### 创建java工程jdbc

导入jar包，BuilderPath

c3p0-0.9.1.2.jar

mysql-connector-java-5.1.28-bin.jar

spring-jdbc-3.2.3.RELEASE-sources.jar

spring-jdbc-3.2.3.RELEASE.jar

spring-tx-3.2.3.RELEASE-sources.jar

spring-tx-3.2.3.RELEASE.jar

aopalliance-1.0.jar

aspectjrt-1.7.4.jar

aspectjweaver-1.7.4.jar

commons-logging-1.0.4.jar

spring-aop-3.2.3.RELEASE-sources.jar

spring-aop-3.2.3.RELEASE.jar

spring-aspects-3.2.3.RELEASE-sources.jar

spring-aspects-3.2.3.RELEASE.jar

spring-beans-3.2.3.RELEASE.jar

spring-context-3.2.3.RELEASE-sources.jar

spring-context-3.2.3.RELEASE.jar

spring-context-support-3.2.3.RELEASE-sources.jar

spring-context-support-3.2.3.RELEASE.jar

spring-core-3.2.3.RELEASE-sources.jar

spring-core-3.2.3.RELEASE.jar

spring-expression-3.2.3.RELEASE-sources.jar

spring-expression-3.2.3.RELEASE.jar

spring-test-3.2.3.RELEASE-sources.jar

spring-test-3.2.3.RELEASE.jar

#### jdbc.properties

jdbc.driver=com.mysql.jdbc.Driver

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

jdbc.username=root

jdbc.password=root

#### applicationContext.xml

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

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util-3.2.xsd">

<!--引入外部配置文件 -->

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

<!--配置数据源 -->

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

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

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

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

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

</bean>

</beans>

#### TestJdbc.java

package jdbc;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import javax.sql.DataSource;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import jdbc.pojo.User;

public class TestJdbc {

@Test

public void select() throws SQLException{

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

DataSource ds = (DataSource)context.getBean("dataSource");

Connection cn = ds.getConnection();

String sql = "select \* from user\_c";

PreparedStatement ps = cn.prepareStatement(sql);

ResultSet rs = ps.executeQuery();

while(rs.next()){

User user = new User();

user.setId(rs.getString(1));

user.setUserName(rs.getString(2));

user.setAge(rs.getInt(3));

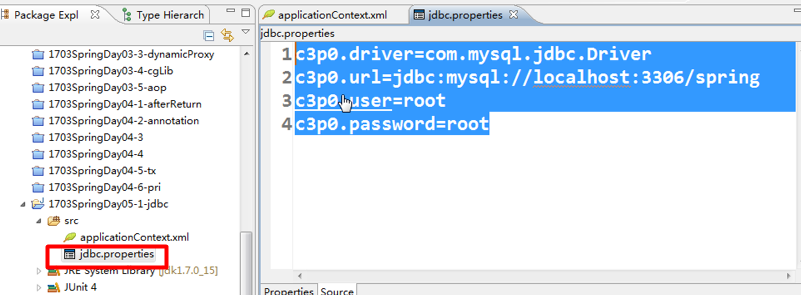
user.setAddress(rs.getString("address"));

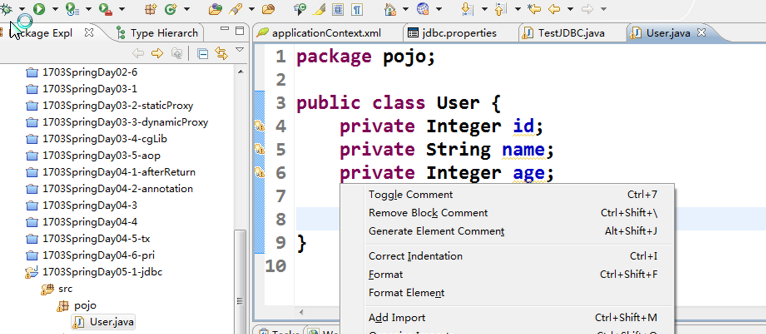
System.out.println(user);

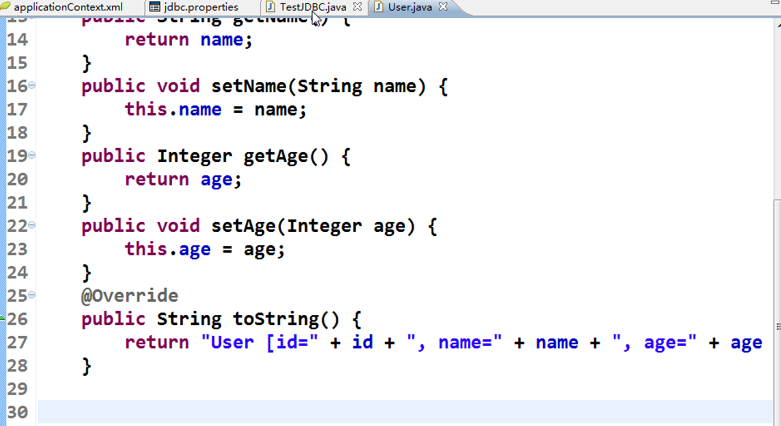
}

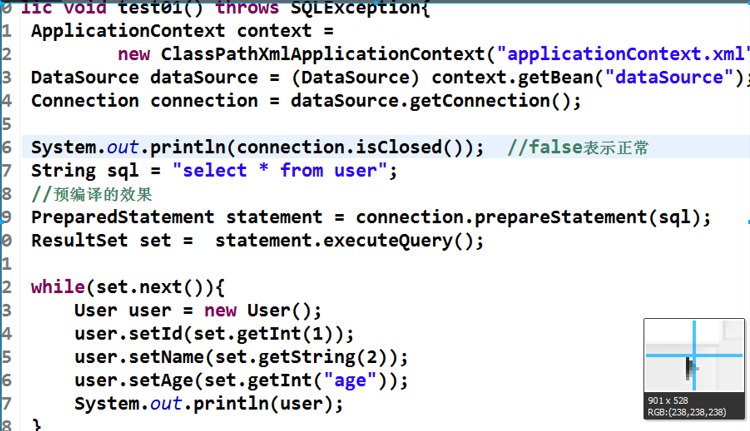
}

}











### ORM映射

#### 手动

写一个对象转换类

package jdbctemplate;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import jdbc.pojo.User;

public class UserMapper implements RowMapper<User>{

//手动映射：将返回的rs对象手动转换为User对象

public User mapRow(ResultSet rs, int index) throws SQLException {

User user = new User();

user.setId(rs.getString(1));

user.setUserName(rs.getString(2));

user.setAge(rs.getInt(3));

user.setAddress(rs.getString("address"));

return user;

}

}

#### 自动

RowMapper<User> userMapper = new BeanPropertyRowMapper<User>(User.class);

它通过反射实现，没有上面直接手写UserMapper执行速度快。

### jdbcTemplate

Spring对数据库的操作在jdbc上面做了深层次的封装，使用spring的注入功能，可以把DataSource注册到JdbcTemplate之中。

JdbcTemplate位于中spring-jdbc-3.2.3.RELEASE.jar其全限定命名为org.springframework.jdbc.core.JdbcTemplate。要使用JdbcTemlate还需包含事务和异常控制spring-tx-3.2.3.RELEASE.jar。

#### UserMapper.java

package jdbctemplate;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import jdbc.pojo.User;

public class UserMapper implements RowMapper<User>{

//手动映射：将返回的rs对象手动转换为User对象

public User mapRow(ResultSet rs, int index) throws SQLException {

User user = new User();

user.setId(rs.getString(1));

user.setUserName(rs.getString(2));

user.setAge(rs.getInt(3));

user.setAddress(rs.getString("address"));

return user;

}

}

#### applicationContext.xml

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

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util-3.2.xsd">

<!--引入外部配置文件 -->

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

<!--配置数据源 -->

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

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

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

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

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

</bean>

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

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

</bean>

</beans>

#### TestJdbcTemplate.java

数据库表的CRUD操作，包括：手动对象转换和自动对象转换。

package jdbc;

import java.util.List;

import org.junit.Before;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.springframework.jdbc.core.BeanPropertyRowMapper;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import jdbc.pojo.User;

import jdbctemplate.UserMapper;

public class TestJdbcTemplate {

private JdbcTemplate jdbcTemplate;

@Before //执行测试方法之前执行，初始化

public void init(){

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

this.jdbcTemplate = (JdbcTemplate) ac.getBean("jdbcTemplate");

}

@Test //查询所有记录

public void selectList(){

String sql = "select \* from user\_c";

UserMapper userMapper = new UserMapper();

List<User> userList = jdbcTemplate.query(sql, userMapper);

for(User u : userList){

System.out.println(u);

}

}

@Test //手动转换

public void selectOne(){

String sql = "select \* from user\_c where id = ?";

UserMapper userMapper = new UserMapper();

User user = jdbcTemplate.queryForObject(sql, userMapper, 5);

System.out.println(user);

}

@Test //自动转换

public void selectListRowMapper(){

String sql = "select \* from user\_c";

RowMapper<User> userMapper = new BeanPropertyRowMapper<User>(User.class);

List<User> userList = jdbcTemplate.query(sql, userMapper);

for(User u : userList){

System.out.println(u);

}

}

@Test //自动转换

public void selectOneRowMapper(){

String sql = "select \* from user\_c where id = ?";

RowMapper<User> userMapper = new BeanPropertyRowMapper<User>(User.class);

User user = jdbcTemplate.queryForObject(sql, userMapper, 5);

System.out.println(user);

}

@Test //新增

public void insert(){

String sql = "insert into user\_c (id,user\_name,age,address) values(?,?,?,?)";

jdbcTemplate.update(sql, "10","tony",18,"北京");

}

@Test //修改

public void update(){

String sql = "update user\_c set age=? where id=?";

jdbcTemplate.update(sql, "16","10");

}

@Test //删除

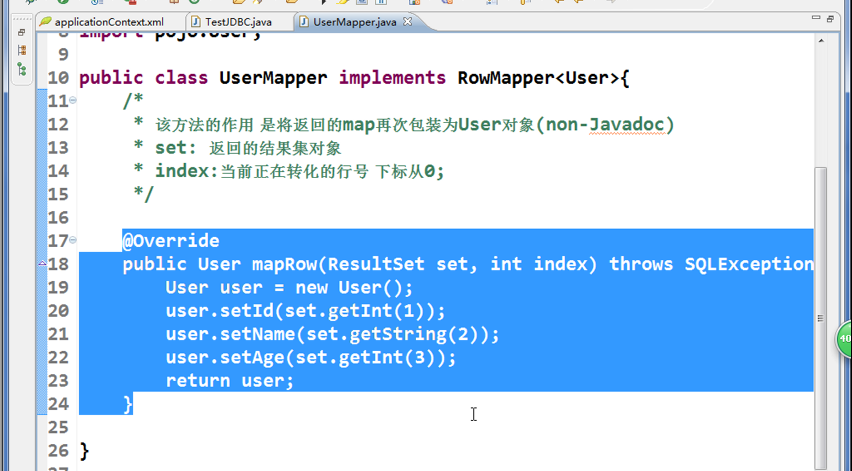
public void delete(){

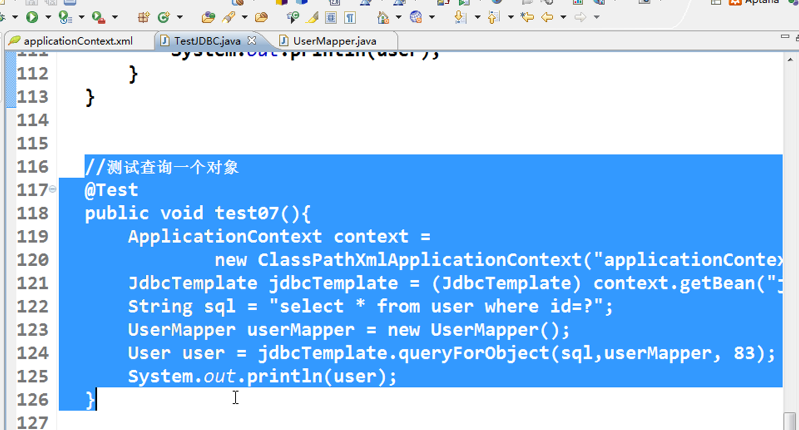
String sql = "delete from user\_c where id=?";

jdbcTemplate.update(sql, "10");

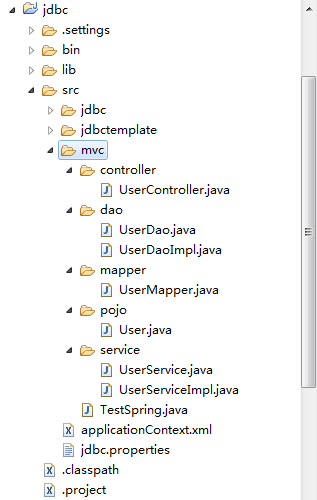
}

}





## CRUD操作



### User.java

package mvc.pojo;

public class User {

private String id;

private String userName;

private Integer age;

private String address;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public Integer getAge() {

return age;

}

public void setAge(Integer age) {

this.age = age;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

@Override

public String toString() {

return "User [id=" + id + ", userName=" + userName + ", age=" + age + ", address=" + address + "]";

}

}

### UserMapper.java

package mvc.mapper;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.stereotype.Component;

import mvc.pojo.User;

@Component

public class UserMapper implements RowMapper<User>{

//手动映射：将返回的rs对象手动转换为User对象

public User mapRow(ResultSet rs, int index) throws SQLException {

User user = new User();

user.setId(rs.getString(1));

user.setUserName(rs.getString(2));

user.setAge(rs.getInt(3));

user.setAddress(rs.getString("address"));

return user;

}

}

### UserDao.java

package mvc.dao;

import java.io.Serializable;

import java.util.List;

import mvc.pojo.User;

public interface UserDao {

public List<User> find();

public User get(Serializable id);

public void save(User user);

public void update(User user);

public void deleteById(Serializable id);

}

### UserDaoImpl.java

package mvc.dao;

import java.io.Serializable;

import java.util.List;

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

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.stereotype.Repository;

import mvc.mapper.UserMapper;

import mvc.pojo.User;

@Repository("userDao")

public class UserDaoImpl implements UserDao{

@Autowired

private JdbcTemplate jdbcTemplate;

@Autowired

private UserMapper userMapper;

public List<User> find() {

String sql = "select \* from user\_c";

return jdbcTemplate.query(sql, userMapper);

}

public User get(Serializable id) {

String sql = "select \* from user\_c where id = ?";

return jdbcTemplate.queryForObject(sql, userMapper, id);

}

public void save(User user) {

String sql = "insert into user\_c (id,user\_name,age,address) values(?,?,?,?)";

jdbcTemplate.update(sql, user.getId(),user.getUserName(),user.getAge(),user.getAddress());

}

public void update(User user) {

String sql = "update user\_c set user\_name=?,age=?,address=? where id=?";

jdbcTemplate.update(sql, user.getUserName(),user.getAge(),user.getAddress(),user.getId());

}

public void deleteById(Serializable id) {

String sql = "delete from user\_c where id=?";

jdbcTemplate.update(sql, id);

}

}

### UserService.java

package mvc.service;

import java.io.Serializable;

import java.util.List;

import mvc.pojo.User;

public interface UserService {

public List<User> find();

public User get(Serializable id);

public void save(User user);

public void update(User user);

public void deleteById(Serializable id);

}

### UserServiceImpl.java

package mvc.service;

import java.io.Serializable;

import java.util.List;

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

import org.springframework.stereotype.Service;

import mvc.dao.UserDao;

import mvc.pojo.User;

@Service("userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

public List<User> find() {

return userDao.find();

}

public User get(Serializable id) {

return userDao.get(id);

}

public void save(User user) {

userDao.save(user);

}

public void update(User user) {

userDao.update(user);

}

public void deleteById(Serializable id) {

userDao.deleteById(id);

}

}

### UserController.java

package mvc.controller;

import java.util.List;

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

import org.springframework.stereotype.Controller;

import mvc.pojo.User;

import mvc.service.UserService;

@Controller

public class UserController {

@Autowired

private UserService userService;

public void find(){

List<User> userList = userService.find();

for(User u : userList){

System.out.println(u);

}

}

public void save(){

User user = new User();

user.setId("10");

user.setUserName("tony");

user.setAge(16);

user.setAddress("北京亦庄");

userService.save(user);

}

}

### jdbc.properties

jdbc.driver=com.mysql.jdbc.Driver

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

jdbc.username=root

jdbc.password=root

### applicationContext.xml

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

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util-3.2.xsd">

<!--引入外部配置文件 -->

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

<!-- 包扫描 -->

<context:component-scan base-package="mvc.\*"/>

<!--配置数据源 -->

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

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

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

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

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

</bean>

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

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

</bean>

</beans>

### TestSpring.java

package mvc;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import mvc.controller.UserController;

public class TestSpring {

@Test //查询所有记录

public void selectList(){

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

UserController userController = (UserController) ac.getBean("userController");

userController.find();

userController.save();

}

}

## 声明式事务

### JDBC事务缺点

只对sql有效，很多重复代码，和SQL耦合性高

spring事务就可以解决，一旦开启spring事务，JDBC事务就不起作用。

缺点：不好理解，因为无法直接看到相应的代码。利用环绕通知+TranactionManager实现事务控制，Spring事务就是利用AOP来实现的。

### UserServiceImpl.java

执行完新增出现异常，按理说应该回滚，可数据却已经添加到数据库中。这显然不合理。

package mvc.service;

import java.io.Serializable;

import java.util.List;

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

import org.springframework.stereotype.Service;

import mvc.dao.UserDao;

import mvc.pojo.User;

@Service("userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

public List<User> find() {

return userDao.find();

}

public User get(Serializable id) {

return userDao.get(id);

}

public void save(User user) {

userDao.save(user);

int i = 1/0;

}

public void update(User user) {

userDao.update(user);

}

public void deleteById(Serializable id) {

userDao.deleteById(id);

}

}

### applicationContext.xml

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

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

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-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util-3.2.xsd">

<!--引入外部配置文件 -->

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

<!-- 包扫描 -->

<context:component-scan base-package="mvc.\*"/>

<!--配置数据源 -->

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

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

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

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

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

</bean>

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

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

</bean>

<!-- 定义事务管理器 -->

<bean id="transactionManager"

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

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

</bean>

<!-- 定义事务策略 -->

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

<tx:attributes>

<tx:method name="save\*" propagation="REQUIRED"/>

<tx:method name="update\*" propagation="REQUIRED"/>

<tx:method name="delete\*" propagation="REQUIRED"/>

<tx:method name="find\*" read-only="true"/>

<tx:method name="get\*" read-only="true"/>

</tx:attributes>

</tx:advice>

<aop:config>

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

<aop:advisor advice-ref="txAdvice" pointcut-ref="pc"/>

</aop:config>

</beans>

执行可以看到，错误依然报错，但是数据库的记录已经不会插入到数据库中，spring会自动促使数据库事务回滚。

### 问题：update事务只读会怎样？

执行报错：

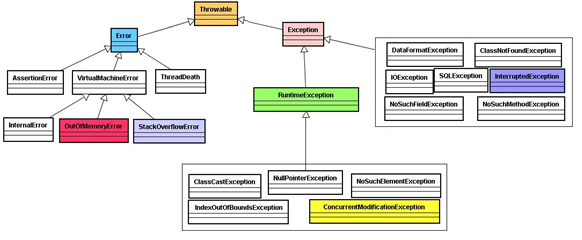
Caused by: java.sql.SQLException: Connection is read-only. Queries leading to data modification are not allowed

### 检查型异常

声明式事务中Spring规定默认对非检查型异常和运行时异常进行事务回滚，而对检查型异常则不进行回滚操作。

那么什么是非检查型异常？什么又是检查型异常？

最简单的判断方法是：继承自error或runtimeexception的是非检查型异常，而继承自exception的则是检查型异常。



public class ArithmeticException extends RuntimeException

非检查型异常，自动回滚

public class SQLException extends java.lang.Exception

检查型异常，不会自动回滚，你不理，我也不理

#### UserService.java

package mvc.service;

import java.io.Serializable;

import java.sql.SQLException;

import java.util.List;

import mvc.pojo.User;

public interface UserService {

public List<User> find();

public User get(Serializable id);

public void save(User user) throws SQLException ;

public void update(User user);

public void deleteById(Serializable id);

}

#### UserServiceImpl.java

package mvc.service;

import java.io.Serializable;

import java.sql.SQLException;

import java.util.List;

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

import org.springframework.stereotype.Service;

import mvc.dao.UserDao;

import mvc.pojo.User;

@Service("userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

public void save(User user) throws SQLException {

userDao.save(user);

throw new SQLException();

//int i=1/0;

}

}

#### UserController.java

package mvc.controller;

import java.sql.SQLException;

import java.util.List;

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

import org.springframework.stereotype.Controller;

import mvc.pojo.User;

import mvc.service.UserService;

@Controller

public class UserController {

@Autowired

private UserService userService;

public void save() throws SQLException{

User user = new User();

user.setId("10");

user.setUserName("tony");

user.setAge(16);

user.setAddress("北京亦庄");

userService.save(user);

}

}

TestSpring.java

package mvc;

import java.sql.SQLException;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import mvc.controller.UserController;

public class TestSpring {

@Test //查询所有记录

public void selectList() throws SQLException{

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

UserController userController = (UserController) ac.getBean("userController");

userController.save();

}

}

可以发现事务没有回滚，虽然出错，记录插入到数据库中。

### 手动修改事务

#### rollback-for

spring支持手动修改，但尽量不要修改：

注意必须是类的全路径java.sql.SQLException

<tx:method name="save\*" propagation="REQUIRED" rollback-for="java.sql.SQLException"/>

执行测试类，这次看到就没有添加数据，事务回滚了。

#### no-rollback-for

相反也可以让非检查型异常不回滚

<tx:method name="save\*" propagation="REQUIRED" no-rollback-for="java.lang.RuntimeException"/>

抛出异常

public void save(User user) throws SQLException {

userDao.save(user);

//throw new SQLException();

int i=1/0;

}

记录还是进来，事务没有回滚。

### spirng中的try-catch

#### UserServiceImpl.java

public void save(User user){

try {

userDao.save(user);

//throw new SQLException();

int i=1/0;

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

看到自己加try/catch不是我们非常好的吗？怎么事务失败，数据还是加入了呢？

报错了，spring怎么还能正常运行呢？

问题的关键不在try/catch，不能引入spring后，业务就不能写try/catch了。而在于上面的异常Exception。这个把所有的Exception都获取了。那spring就无法知道你报错了。不知道，我还怎么给你维护事务呢？

所以我们实际开发中这边一般放的是自定义异常，Exception就不能用了。

#### MyException.java

package mvc.exception;

//注意不能继承Exception，否则事务不生效

public class MyException extends RuntimeException{

private static final long serialVersionUID = 1L;

public MyException(String msg){

}

}

#### applicationContext.xml

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

<tx:attributes>

<tx:method name="save\*" propagation="REQUIRED"/>

<tx:method name="update\*" propagation="REQUIRED"/>

<tx:method name="delete\*" propagation="REQUIRED"/>

<tx:method name="find\*" read-only="true"/>

<tx:method name="get\*" read-only="true"/>

</tx:attributes>

</tx:advice>

#### UserServiceImpl.java

public void save(User user){

try {

userDao.save(user);

int i=1/0;

} catch (MyException e) {

System.out.println(e.getMessage());

}

}

通过自定义异常实现用户自行处理，只要发现自己有try-catch，spring事务失效。

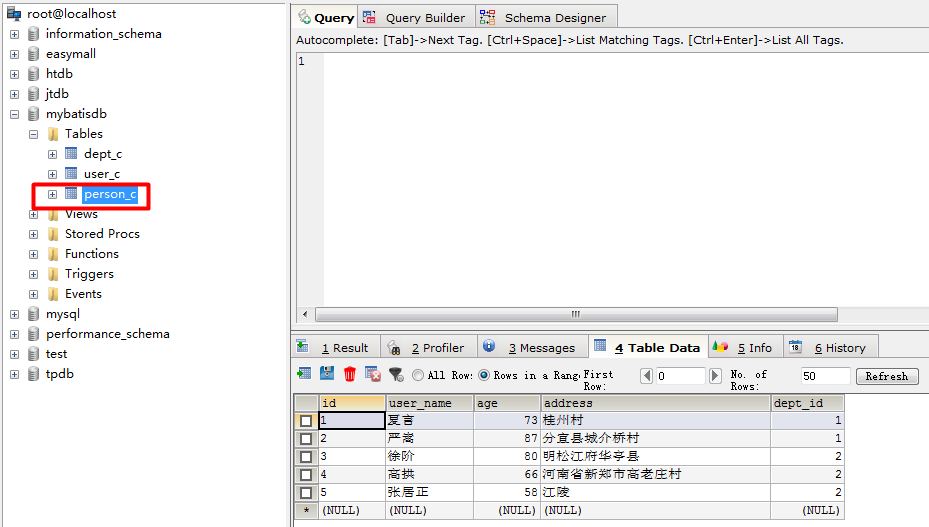
小结

如果在编程中石油try-catch，建议不要随便加Exception，因为Exception的捕获范围太大，如果异常成功捕获，那么Spring容器不知道有异常，则事务不能起作用，所以以后编写try-catch时，异常捕获写的是自己能够处理的异常信息，如果异常不能处理则交给Spring容器。

## 多表操作

### 不在一起

#### 复制user\_c表为person\_c



#### PersonDao.java

package mvc.dao;

import mvc.pojo.Person;

public interface PersonDao {

public void save(Person person);

}

#### PersonDaoImpl.java

package mvc.dao;

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

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.stereotype.Repository;

import mvc.pojo.Person;

@Repository("personDao")

public class PersonDaoImpl implements PersonDao{

@Autowired

private JdbcTemplate jdbcTemplate;

public void save(Person person) {

String sql = "insert into person\_c (id,user\_name,age,address) values(?,?,?,?)";

jdbcTemplate.update(sql, person.getId(),person.getUserName(),person.getAge(),person.getAddress());

}

}

#### PersonService.java

package mvc.service;

import mvc.pojo.Person;

public interface PersonService {

public void save(Person person);

}

#### PersonServiceImpl.java

package mvc.service;

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

import org.springframework.stereotype.Service;

import mvc.dao.PersonDao;

import mvc.pojo.Person;

@Service("personService")

public class PersonServiceImpl implements PersonService{

@Autowired

private PersonDao personDao;

public void save(Person person) {

personDao.save(person);

}

}

#### UserController.java

public void save() throws SQLException{

User user = new User();

user.setId("10");

user.setUserName("tony");

user.setAge(16);

user.setAddress("北京亦庄");

userService.save(user);

int i = 1/0;

Person person = new Person();

person.setId("10");

person.setUserName("tony");

person.setAge(16);

person.setAddress("北京亦庄");

personService.save(person);

}

#### TestSpring.java

package mvc;

import java.sql.SQLException;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import mvc.controller.UserController;

public class TestSpring {

@Test //查询所有记录

public void selectList() throws SQLException{

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

UserController userController = (UserController) ac.getBean("userController");

userController.save();

}

}

执行时，user插入，person失败，不合理，没有形成一个事务，没有回滚。

### 在一起

#### UserService.java

package mvc.service;

import java.io.Serializable;

import java.util.List;

import mvc.pojo.Person;

import mvc.pojo.User;

public interface UserService {

public List<User> find();

public User get(Serializable id);

public void save(User user, Person person);

public void update(User user);

public void deleteById(Serializable id);

}

#### UserServiceImpl.java

package mvc.service;

import java.io.Serializable;

import java.util.List;

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

import org.springframework.stereotype.Service;

import mvc.dao.PersonDao;

import mvc.dao.UserDao;

import mvc.pojo.Person;

import mvc.pojo.User;

@Service("userService")

public class UserServiceImpl implements UserService{

@Autowired

private UserDao userDao;

@Autowired

private PersonDao personDao;

public List<User> find() {

return userDao.find();

}

public User get(Serializable id) {

return userDao.get(id);

}

public void save(User user, Person person){

userDao.save(user);

int i = 1/0;

personDao.save(person);

}

public void update(User user) {

userDao.update(user);

}

public void deleteById(Serializable id) {

userDao.deleteById(id);

}

}

#### UserController.java

public void save() throws SQLException{

User user = new User();

user.setId("10");

user.setUserName("tony");

user.setAge(16);

user.setAddress("北京亦庄");

Person person = new Person();

person.setId("10");

person.setUserName("tony");

person.setAge(16);

person.setAddress("北京亦庄");

userService.save(user, person);

}

抛错，记录都没有插入成功，事务回滚。

所以Spring事务非常简单，在service把多个表的操作方法放在一个service的方法中，Spring会自动给它加上事务保护。