**Spring概述**

[Spring](http://localhost:9093/itour/work/detail?id=101)

阅读  最后发布于 2022-08-05 21:11

**1.什么是Spring**

从微观上来说，Spring即spring-framework，是开源的轻量级的一站式JAVAEE框架。Spring在JAVAEE三层结构中分别提供了不同的解决技术，web提供了SpringMvc，Service提供了IOC, 数据访问层提供了JdbcTemplate；

从宏观上来说，Spring指的是Spring构建的整个Spring生态圈。Spring的整个生态圈覆盖了web开发、数据访问、安全控制、分布式、消息服务、移动开发、批处理等领域，为其提供了对应的解决方案；

**2.Spring的核心模块**

a.AOP(面向切面编程):面向切面，在不修源员代码的情况下进行功能的增强（拓展功能时不修改源代码）。如日志、权限等功能。

b.IOC(控制反转):控制反转，把创建对象的过程交给Spring进行;如有一个类，在类里面有个方法（非静态方法），调用类里面的方法， 需要创建类对象，使用对象调用方法，创建对象的过程需要new出来， 把对象的创建方式不是通 过new方式实现（程序代码创建对象），而是交给spring创建类对象,将创建对象的控制权交给Spring;

**3.Spring的特点**

1）方便解耦，简化开发，降低企业应用开发的复杂度；

2）AOP编程支持

3）方便与其他框架整合

4）方便进行事务操作；

**4.Spring的版本**

Spring Framework4.X需要jdk1.6及以上的支持;

Spring Framework5.X需要jdk1.8及以上的支持;

# Spring IOC操作

[Spring](http://localhost:9093/itour/work/detail?id=117)

阅读  最后发布于 2022-07-30 10:02

## 1.什么是IOC?

IOC即控制反转，把对象的创建和对象之间的调用过程，交个Spring进行管理，从达到降低耦合度的目的；

## 2.IOC的底层原理

IOC底层主要使用XML解析、工厂设计模式、反射等技术；

### 2.1发展过程

1)原始方式2）工厂模式3）IOC方式

### 2.2.IOC接口（BeanFactory）

IOC思想基于IOC容器完成，IOC容器底层就是对象工厂；Spring提供了IOC容器实现两种方式（两个接口）

1）BeanFactory：IOC容器基本实现，是Spring内部的使用接口，一般不提供给开发人员使用；

2）ApplicationContext：BeanFactory接口的子接口，提供了更多更强大的功能，一般是由开发人员使用的；

### 2.3.BeanFactory和ApplicationContext的区别

1）BeanFactory加载配置文件的时候不会创建对象，在获取对象（使用）的时候才会去创建对象；

2）ApplicationContext加载配置文件的时候就会把配置文件中的对象进行创建；

3）当Spring与web项目集成时，一般把耗时、好资源的操作都在项目启动的时候进行处理更合适。

### 2.4.ApplicationContext接口实现类

ApplicationContext有两个主要的实现类，FileSystemXmlApplicationContext和ClassPathXmlApplicationContext；

FileSystemXmlApplicationContext:文件对应盘符的路径；

ClassPathXmlApplicationContext:src路径下的类路径；

## 3.IOC操作Bean管理

Bean管理是指使用Spring创建对象和注入属性两个操作。Spring IOC提供了两种管理Bean的方式

1）基于XML配置文件方式实现

2）基于注解方式实现;

### 一、基于XML方式Bean管理

1）在Spring配置文件中，使用bean标签，在bean标签里添加对应的属性，就可以实现对象创建；

2）bean标签中常用的属性

1.id:唯一标识

2.class:创建对象的类全路径（包类路径）

#### 1.基于XML方式创建对象方式

##### 1）使用无参构造

创建对象时，默认执行无参构造方法完成对象的创建,如果类中没有无参构造会出现异常；

<bean id="emp" class="com.itour.Emp"></bean>

#### 2.基于XML方式注入属性

DI：依赖注入，就是注入属性，DI是IOC中的一种具体实现，表示依赖注入，需要在创建对象的基础上完成；

#### 3.基于xml方式实现依赖注入方式：

##### 1）使用set方法进行注入；

1.创建类，定义属性和对应的set方法；  
2.在Spring配置文件中配置对象创建，配置属性注入；

<bean id="book" class="com.itour.Book">

<!--使用property属性完成属性注入，name：类里面的属性名称;value:向属性中注入的值-->

<property name="bname" value="欧亨利小说集"></property>

</bean>

##### 2）使用有参构造函数注入;

1.创建类，定义属性，创建属性对应有参构造函数；

2.在Spring配置文件中constructor-arg标签进行注入

<bean id="book" class="com.itour.Book">

<constructor-arg name="id" value="1"></constructor-arg>

<constructor-arg name="bname" value="欧亨利小说集"></constructor-arg>

</bean>

##### 3)p命名空间注入

1.使用P名称空间注入，可以简化基于xml配置方式（调用的是无参构造函数）

1）添加P名称空间在配置文件中

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

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

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

2)进行属性注入，在bean标签里面进行操作；

<bean id="book" class="com.itour.Book" p:bname="朱自清散文集" p:id="1">

</bean>

#### 4.XML方式注入属性

##### 1.)字面量

1.null值（使用null标签）

<property name="bname">

<null/>

</property>

##### 2.)属性值包含特殊符号

1）把<>进行转义;

<property name="bname">

<value>&lt;&lt;朱自清散文集&gt;&gt;</value>

</property>

2)把待特殊符合内容写在CDATA结构中；

<property name="bname">

<value><![CDATA[<<朱自清散文集>>]]></value>

</property>

##### 3.)注入属性-外部Bean

1.创建Service和dao对象，在Service中注入dao；

2.在Spring配置文件中配置

<bean id="bookService" class="com.itour.BookService">

<!--注入bookDao对象

name:类里面属性名称；

ref：创建bookDao对象bean标签的id值

-->

<property name="bookDao" ref="bookDaoImpl"> </property>

</bean>

<bean id="bookDaoImpl" class="com.itour.BookDao"></bean>

##### 4.)注入属性-内部Bean

Emp对象中有一个dept对象（一个员工对应一个部门）；

<bean id="emp" class="com.itour.Emp">

<!--设置普通属性-->

<property name="empno" value="1"> </property>

<property name="ename" value="zhangsan"> </property>

<!--设置对象类型属性-->

<property name="dept" >

<bean id="dept" class="com.itour.Dept">

<property name="id" value="1"> </property>

<property name="dname" value="技术部"> </property>

</bean>

</property>

</bean>

##### 5.)注入属性-级联赋值

1）级联赋值第一种写法

<bean id="emp" class="com.itour.Emp">

<!--设置普通属性-->

<property name="empno" value="1"> </property>

<property name="ename" value="zhangsan"> </property>

<!--级联赋值-->

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

</bean>

<bean id="dept" class="com.itour.Dept">

<property name="id" value="1"> </property>

<property name="dname" value="技术部"> </property>

</bean>

1）级联赋值第二种写法（就近原则），需要生成对应对象的（dept）的get方法

<bean id="emp" class="com.itour.Emp">

<!--设置普通属性-->

<property name="empno" value="1"> </property>

<property name="ename" value="zhangsan"> </property>

<!--级联赋值-->

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

<property name="dept.id" value="1"> </property>

<property name="dept.dname" value="财务部"> </property>

</bean>

<bean id="dept" class="com.itour.Dept">

<property name="id" value="1"> </property>

<property name="dname" value="技术部"> </property>

</bean>

##### 6.）xml方式注入集合属性

1.创建类，定义数组,list,set,map类型属性，生成对应的set方法；

2.在Sprig配置文件中配置

###### 1)注入数组类型属性

使用array标签完成

<bean id="dept" class="com.itour.Emp">

<property name="arr">

<array>

<value>游泳</value>

<value>打球</value>

</array>

</property>

</bean>

###### 2)注入List集合属性

使用list标签完成

<bean id="dept" class="com.itour.Emp">

<property name="list">

<list>

<value>游泳</value>

<value>打球</value>

</list>

</property>

</bean>

###### 3)注入Set集合属性

使用set标签完成

<bean id="dept" class="com.itour.Emp">

<property name="sets">

<set>

<value>游泳</value>

<value>打球</value>

</set>

</property>

</bean>

###### 4）注入Map集合属性

使用entry标签来完成

<bean id="dept" class="com.itour.Emp">

<property name="maps">

<map>

<entry key="travel" value="旅行"></entry>

<entry key="food" value="美食"></entry>

</map>

</property>

</bean>

###### 5）在集合里面注入对象类型值

一个部门对应多个员工

<bean id="dept" class="com.itour.Dept">

<property name="empList">

<!--使用List和ref标签完成对List<javaBean>注入-->

<list>

<ref bean="employee1"></ref>

<ref bean="employee2"></ref>

</list>

</property>

</bean>

<bean id="employee1" class="com.itour.Emp">

<property name="empno" value="1"></property>

<property name="ename" value="mike"></property>

</bean>

<bean id="employee2" class="com.itour.Emp">

<property name="empno" value="2"></property>

<property name="ename" value="joe"></property>

</bean>

###### 6）把集合注入部分提取出来

1）在spring中引入名称空间util

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

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

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

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

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

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

2)抽取list集合类型属性注入

<util:list id="empoyeeList">

<ref bean="employee1"></ref>

<ref bean="employee2"></ref>

</util：list>

<bean id="employee1" class="com.itour.Emp">

<property name="empno" value="1"></property>

<property name="ename" value="mike"></property>

</bean>

<bean id="employee2" class="com.itour.Emp">

<property name="empno" value="2"></property>

<property name="ename" value="joe"></property>

</bean>

3)提取出来的list集合类型属性注入使用

<bean id="dept" class="com.itour.Dept">

<property name="empList" ref="empoyeeList"></property>

</bean>

#### 5.Bean管理FactoryBean

1、Spring 有两种类型bean,一种普通 bean，另外一种工厂 bean (FactoryBean) 。

2、普通 bean：在配置文件中定义 bean 类型就是返回类型。

<bean id="person" class="com.stu.spring.Person"></bean>

3、工厂bean：在配置文件定义bean 类型可以和返回类型不一样。

第一步 创建类，让这个类作为工厂bean，实现接口 FactoryBean。

第二步 实现接口里面的方法，在实现的方法中定义返回的 bean 类型。

public class Person<T> implements FactoryBean<T> {

private Long id;

private String name;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

//定义返回bean

@Override

public T getObject() throws Exception {

// TODO Auto-generated method stub

Student stu = new Student();

stu.setName("mike");

return (T) stu;

}

@Override

public Class<?> getObjectType() {

// TODO Auto-generated method stub

return null;

}

}

<bean id="person" class="com.stu.spring.Person"></bean>

public class Test {

@org.junit.Test

public void test1() {

ApplicationContext context = new ClassPathXmlApplicationContext("conf/bean1.xml");

Student bean = context.getBean("person",Student.class);

System.out.println(bean.getName());

}

}

#### 6.Bean的作用域

在Spring里面，Bean的作用域指设置创建Bean实例是单实例还是多实例；单实例在整个程序中bean实例只有一个，多实例每次创建对象都会创建一个新的对象；

##### 1.)如何设置Bean是单例还是多实例？

1）在Spring配置文件中Bean标签里面有属性(scope)用于设置是单实例还是多实例；

2）scope的属性值singleton表示单实例对象，是默认值，prototype表示多实例对象；

3）scope的属性值为request时，每次创建的对象会放入Request对象中；

4）scope的属性值为session时，每次创建的对象会放入Session对象中；

##### 2.)singleton和prototype的区别

1.singleton是单实例，prototype是多实例；

2.设置scope的值为singleton的时候，加载Spring配置文件的时候就会创建单例对象； 设置socpe的值为prototype的时候，在加载spring配置文件的时候不会创建对象，在调用getBean方法的时候会创建多实例对象；

#### 7.Bean的生命周期

生命周期是指从对象创建到对象销毁的过程；Bean的声明周期包含了：

1.通过构造器创建 bean 实例（无参数构造）

2.把Bean实例传递Bean后置处理器的方法postProcessBeforeInitialization中；

3.为bean的属性设置值和对其他bean引用（调用 set 方法）

4.把Bean实例传递Bean后置处理器的方法postProcessAfterInitialization中；

5.调用bean的初始化的方法（需要进行配置）

6.bean的使用（对象获取）

7.当容器关闭时候，调用bean 的销毁的方法（需要进行配置销毁的方法）

实例  
1.创建类；

public class Person {

private String name;

public Person() {

System.out.println("1.通过构造器创建 bean 实例");

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

System.out.println("2.为bean的属性设置值和对其他bean引用");

}

public void initMethod() {

System.out.println("3.调用bean的初始化的方法");

}

public void destoryMethod() {

System.out.println("5.调用销毁方法");

}

}

2.在xml中配置

<bean id="myBeanPostProcessor" class="com.stu.spring.MyBeanPostProcessor"></bean>

<bean id="person" class="com.stu.spring.Person" init-method="initMethod" destroy-method="destoryMethod">

<property name="name" >

<value>朱自清</value>

</property>

</bean>

3.创建后置处理器

public class MyBeanPostProcessor implements BeanPostProcessor {

@Override

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

// TODO Auto-generated method stub

System.out.println("postProcessBeforeInitialization");

return bean;

}

@Override

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

// TODO Auto-generated method stub

System.out.println("postProcessAfterInitialization");

return bean;

}

}

4.测试

public class TestSpring {

@Test

public void test1() {

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("conf/bean1.xml");

Person bean = context.getBean("person",Person.class);

System.out.println("4.获取Bean的实例");

context.close();//手动让bean实例销毁

}

}

#### 8.Bean自动装配

根据指定装配规则（属性名或者属性类型），Spring自动将匹配的属性值进行注入；  
实现自动装配可通过bean标签中的autowire，配置自动装配。autowire有两个值：  
byName：根据属性名称注入，注入的bean的id必须和类属性名词一致；  
byType:根据属性类型注入；

手动注入：

<bean id="emp" class="com.itour.Emp">

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

</bean>

<bean id="dept" class="com.itour.Dept"></bean>

自动注入  
根据属性名称注入，要求实体类中的属性名称，和要注入的id的名称必须一致；

<bean id="emp" class="com.itour.Emp" autowire="byName"></bean>

<bean id="dept" class="com.itour.Dept"></bean>

根据类型注入，如果配置了多个类型相同的Bean，会报错；

<bean id="emp" class="com.itour.Emp" autowire="byType"></bean>

<bean id="dept" class="com.itour.Dept"></bean>

#### 9.Bean引入外部属性文件

当创建对象时，对象中的属性发生变化后需要修改Spring的xml配置文件，不是很方便，可以把一些固定的值或相关的值放入属性文件中（.properties），然后在xml配置文件中引入属性文件，然后在xml中读取属性文件中的内容；例如在操作数据库时,将数据相关的属性值放入属性文件中，在Spring的配置文件中引入对应的属性文件，然后读取属性文件中的内容； 1.直接配置数据库信息

‹bean id="dataSource"class="com. alibaba druid. pool. DruidDataSource">

<property name="driverClassName" value="com.mysq1.jdbc.Driver"></property>

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

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

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

</bean>

2.引入外部属性文件配置数据库信息  
1）创建外部属性文件，properties格式文件

prop.driverClass=com.mysql.jdbc.Driver

prop.url=jdbc:mysql://localhost:3306/userDb

prop.userName=root

prop.password-root

1. 把外部的properties属性文件引入到Spring配置文件中 2.1引入context空间

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

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

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

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd"

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

2.2添加连接池对应的依赖或jar，引入properties文件，并读取文件中的内容

＜!--引入外部屬性文件--〉

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

＜!--配置连接池--〉

‹bean id="dataSource"class="com. alibaba druid.pool.DruidDataSource">

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

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

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

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

</bean>

### 二、基于注解方式Bean管理

#### 1.什么是注解

1.注解是代码特殊标记，格式：@注解名称(属性名称=属性值，属性名称=属性值...)

2.注解可以作用在类、方法、属性上；

3.使用注解的目的是简化xml的配置；

#### 2.Spring针对Bean管理提供了对应的注解

1.@Component:一般用于实体层对象创建

2.@Service:一般用于Service层对象创建；

3.@Controller:一帮用于controller层对象创建

4.@Repository:一般拥有dao层对象创建；

总结：四个注解功能一样，都可以创建Bean的实例；

#### 3.基于注解方式实现对象创建

1.引入Spring基本依赖及Spring-aop的依赖；

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.16</version>

</dependency>

2.在Spring配置文件中开启组件扫描

<!-- 开启组件扫描

1.如果扫描多个包，多个包使用逗号分隔开；

2.扫描包的上层目录；

3.默认扫描base-package指定包下的所有类

-->

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

开启组件扫描细节配置

<!-- 扫描com.itour包下使用@Controller注解标记的类

use-default-filters="false":表示不使用默认的filter,使用自己配置的filter;

context:include-filter:设置要扫描哪些内容；

-->

<context:component-scan base-package="com.itour" use-default-filters="false">

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

</context:component-scan>

<!--扫描com.itour包下没有使用@Controller注解标记的类

context:exclude-filter：设置哪些哪些内容不要扫描

-->

<context:component-scan base-package="com.itour" >

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

</context:component-scan>

3.创建类，在类上面添加创建对象注解

//1.在注解中，value属性值可以省略不写，默认值是类名称，首字母小写；

@Component（value="emp"）

public class Emp {

@Value("Mike")

private String name;

@Autowired

private Dept dept;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Dept getDept() {

return dept;

}

public void setDept(Dept dept) {

this.dept = dept;

}

@Override

public String toString() {

return "Emp [name=" + name + ", dept=" + dept + "]";

}

}

3)测试

@Test

public void test2() {

ApplicationContext context = new ClassPathXmlApplicationContext("conf/bean1.xml");

Emp bean = context.getBean("emp",Emp.class);

System.out.println(bean);

}

#### 4.基于注解方式实现属性注入

1.@Autowired：根据属性类型进行自动装配；

2.@Qualifier：根据属性名称进行注入；  
@Qualifier注解的使用，需要和@Autowired一起使用； 当有多个实现类时，可以指定具体需要注入的实现类

3.@Resource:可以根据类型注入，也可以根据名称注入；

@Resource(name="userDao") 默认根据类型注入，指定name属性是根据名称注入； @Resource属于javax.annotation.Resource包下，是java提供的注解，不属于spring提供的注解，spring建议使用@Autowired和@Qualifier；

4.@Value:注入普通类型属性；

#### 5.完全注解开发

1）创建配置类，替代xml配置文件；

@Configuration

@ComponentScan(basePackages = "com.itour")

public class SpringConfig {

}

2)创建类，添加对应的注解

@Component

public class Emp {

@Value("Mike")

private String name;

@Autowired

private Dept dept;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Dept getDept() {

return dept;

}

public void setDept(Dept dept) {

this.dept = dept;

}

@Override

public String toString() {

return "Emp [name=" + name + ", dept=" + dept + "]";

}

}

3)测试

@Test

public void test3() {

ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);

Emp bean = context.getBean("emp",Emp.class);

System.out.println(bean);

}

## 总结

1）控制反转(Spring IOC)操作Bean管理主要是包含了创建对象和依赖注入两个操作；

2）控制反转(Spring IOC)操作底层主要使用了XML解析、工厂模式、反射技术；

3）Spring IOC提供了两种管理Bean的方式,基于xml方式和注解方式。基于xml方式使用bean标签创建对象，依赖注入可使用set、有参构造函数、P命名空间注入等方式实现。 基于注解的方式可以使用@Component、@Service、@Controller、@Repository创建Bean实例，依赖注入可以使用@Autowired、@Qualifier、@Resource、@Value实现；

4)只有将对象的创建交给了Spring管理，Spring才能帮我们完成依赖注入；

# Spring AOP

## 什么是AOP？

AOP(面向切面编程):利用AOP可以对业务逻辑的各个部分进行隔离，将与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了业务模块间的耦合度，同时提高了系统的可维护性。简单说就是拓展功能（权限认证、日志、事务处理）不需要修改源代码。

## AOP的底层原理

发展过程AOP原理Spring AOP 底层使用了动态代理，使用动态代理来增强类中某个方法的功能。在有接口的情况下，使用JDK的动态代理，通过创建接口实现类代理对象来增强类的方法； 没有接口的情况下，使用CGLIB动态代理，通过创建子类的代理对象来增强类的方法；

## 3.JDK动态代理和CGLIB动态代理

### 1.JDK动态代理实现

使用JDK动态代理，利用Proxy类里面的newProxyInstance方法创建代理对象；

1.创建接口，定义方法

public interface HouseProvider {

void buyHouse();

}

2.创建接口实现类，实现方法

public class HouseProviderImpl implements HouseProvider {

@Override

public void buyHouse() {

// TODO Auto-generated method stub

System.out.println("房屋交易业务");

}

}

3.使用Proxy类创建代理对象

public class ProxyFactory {

//目标对象(被代理对象)

private Object target;

public ProxyFactory(Object target) {

this.target = target;

}

/\*\*

\* public static Object newProxyInstance(ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h)

\* loader:指定目标对象使用的类加载器，获取加载器的方式固定;

\* interfaces:目标对象实现的接口类型,使用泛型方法确认类型

\* h:事情处理:执行目标对象的方法时，会触发事情处理器方法(增强)

\*/

public Object getProxyInstance() {

return Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), new InvocationHandler() {

//增强逻辑

@Override

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

// TODO Auto-generated method stub

System.out.println("增强的方法执行前执行");

//执行被增强的方法

Object result = method.invoke(target, args);

System.out.println("增强的方法执行后执行");

return result;

}

});

}

}

4.测试

@Test

public void testClient() {

HouseProvider houseProviderImpl = new HouseProviderImpl();

HouseProvider proxyInstance = (HouseProvider)new ProxyFactory(houseProviderImpl).getProxyInstance();

proxyInstance.buyHouse();

}

### 2.CGLIB动态代理

使用CGLIB动态代理，通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。

1.创建被增强的类（被代理的类）

public class HouseProvider {

public void show() {

System.out.println("你好，房屋出售");

}

}

2.在子类中采用方法拦截父类方法的调用，实现增强

public class ProxyFactory implements MethodInterceptor {

//目标对象(被增强对象)

private Object target;

public ProxyFactory(Object target) {

super();

this.target = target;

}

public Object getProxyInstance() {

//1.创建一个工具类

Enhancer enhancer = new Enhancer();

//2.设置父类

enhancer.setSuperclass(target.getClass());

//3.设置回调函数

enhancer.setCallback(this);

//4.创建子类对象,即代理对象

return enhancer.create();

}

//增强逻辑

@Override

public Object intercept(Object arg0, Method method, Object[] arg2, MethodProxy arg3) throws Throwable {

// TODO Auto-generated method stub

System.out.println("Cglib代理-->被增强方法之前执行");

//调用被增强方法

Object result = method.invoke(target, arg2);

System.out.println("Cglib代理-->被增强方法之后执行");

return result;

}

}

4.测试

@Test

public void testClient() {

HouseProvider provider = new HouseProvider();

HouseProvider proxyInstance = (HouseProvider)new ProxyFactory(provider).getProxyInstance();

proxyInstance.show();

}

## 4.Spring AOP术语

1.)通知/增强(Advice):增强的逻辑称为增强;比如拓展日志功能，这个日志功能被称为增强;分为五种类型：

I.)前置通知：在方法之前执行;

II.)后置通知：在方法之后执行;

III.)异常通知: 在方法出现异常时执行;

IV.)最终通知: 在后置之后执行；

V.)环绕通知: 在方法之前和之后执行;

2.)切入点(Pointcut)：在类里面可以有许多方法被增强，比如实际的操作中我们只增强了add() 和update(),实际增强的方法就叫切入点;

3.)连接点(Joinpoint)：类里面那些方法可以被增强,这些方法被称为连接点;

4.)切面(Aspect):把增强应用到具体的方法上面的过程称为切面（把增强应用到切入点的过程，例如把权限认证用到登录功能上的过程）

## 5.Spring AOP操作

在Spring里面进行AOP的操作，基于aspectj实现提供了两种实现方式：

1.基于aspectj的xml方式；

2.基于aspectj的注解方式；

### 5.1基于Aspectj的XML方式

在Spring里面进行AOP的操作，使用aspectj实现，Aspectj不是spring的部分，和spring一起使用进行AOP的操作，Spring2.0新增了对Aspectj的支持。

#### 1.)Aspectj简介

1.Aspectj是一个面向切面的框架,它拓展了Java语言,Aspectj定义了AOP的基本语法，所以有一个专门的编译器用来生成遵守Java字节编码的Class文件;

2.Aspectj是一个基于java的AOP框架;

3.Spring2.0之后新增了对Aspectj切点的表达式的支持；

4.@Aspectj是Aspectj1.5新增的功能，通过JDK5注解技术,允许直接在Bean类中定义切面；

5.新版的Spring框架建议使用Apectj方式开发Aop；

6.使用Aspectj需要导入Spring AOP和Aspectj相关jar包;

#### 2.)基于Aspectj的XML方式实现AOP

1.引入Spring基本的依赖，及AOP相关依赖；

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>5.3.16</version>

</dependency>

2.编写增强逻辑及增强类

//增强类

public class UserAdvice {

public void before1() {

System.out.println("前置通知--before");

}

public void after1(){

System.out.println("后置置增强---after");

}

public Object round1(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{

System.out.println("环绕-方法之前--around start");

Object proceed = proceedingJoinPoint.proceed();

System.out.println("环绕-方法之后--around end");

return proceed;

}

public void returing1() throws Throwable{

System.out.println("最终通知---after-returning");

}

public void error1() throws Throwable{

System.out.println("异常通知--after-throwing");

}

}

//被增强类

public class UserService {

public String add() {

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

System.out.println(1/0);

return "success";

}

}

3.在Spring配置文件中引入aop的约束

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

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

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

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

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

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

xsi:schemaLocation="

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.1.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.1.xsd

http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc-4.1.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.1.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.1.xsd">

4.在Spring中配置增强类和被增强类，使用表达式配置切入点和切面

<!-- 增强类 -->

<bean id="userAdvice" class="com.stu.service.UserAdvice"></bean>

<!-- 被增强类 -->

<bean id="userService" class="com.stu.service.UserService"></bean>

<!-- 配置AOP

excution(访问修饰符 返回类型 包.类名.方法名 参数)

访问修饰符: 一般省略，\*表示任意，public表示公共方法；

返回类型：不可以省略，\*表示任意，String表示字符串类型，void表示没有返回值

包名：a.固定包名：com.itour.service

b.子包：com.itour.service..（包含com.itour.service）

c.子包：com.itour.\*.service,com.itour.\*.service..

方法名: \*表示任意，\*batch表示以batch结尾的方法，insert\*表示以insert结尾的方法；

-->

<aop:config>

<!-- 配置切入点(增强的方法) -->

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

<!-- 配置切面(把增强用到方法上) -->

<aop:aspect ref="userAdvice">

<!-- 配置增强类型方法:增强类里面使用哪个方法作为前置

before:前置通知;after:后置通知;around:环绕通知;after-returning:最终（返回）通知;after-throwing:异常通知

执行顺序:1.before 2.around 调用proceedingJoinPoint.proceed() 3.after 4.returning 5.around 6.after,如果出现异常

-->

<aop:before method="before1" pointcut-ref="pointcut"/>

<aop:after method="after1" pointcut-ref="pointcut"/>

<aop:around method="round1" pointcut-ref="pointcut"/>

<aop:after-returning method="returing1" pointcut-ref="pointcut"/>

<aop:after-throwing method="error1" pointcut-ref="pointcut"/>

</aop:aspect>

</aop:config>

5.测试

public class TestUser {

@Test

public void test2() {

ApplicationContext context = new ClassPathXmlApplicationContext("conf/bean1.xml");

UserService bean = context.getBean("userService",UserService.class);

System.out.println("add "+bean.add());

}

}

#### 5.2基于Aspectj的注解方式实现AOP

1.导入相关jar,创建增强类和被增强类;

//被增强类

@Service

public class UserService {

public String add() {

return "添加成功";

}

}

@Component

public class UserAdvice {

}

基于xml创建对象,需要在Spring配置文件中配置bean

<bean id="userAdvice" class="com.itour.UserProxy"></bean>

<bean id="userService" class="com.itour.service.UserService"></bean>

基于注解创建对象，需要开启注解扫描

<!-- 开启组件扫描

1.如果扫描多个包，多个包使用逗号分隔开；

2.扫描包的上层目录；

3.默认扫描base-package指定包下的所有类

-->

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

2.使用@Aspect标识增强类，在作为通知方法上面添加通知类型注解，使用切入点表达式配置切入点完成方法的增强

@Component

@Aspect

public class UserAdvice {

@Before(value="execution(\* com.study.service.\*.\*(..))")

public void before1() {

System.out.println("前置通知");

}

@After(value="execution(\* com.study.service.\*.\*(..))")

public void after1(){

System.out.println("后置置增强");

}

@Around(value="execution(\* com.study.service.\*.\*(..))")

public Object round1(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{

System.out.println("方法之前");

Object proceed = proceedingJoinPoint.proceed();

System.out.println("方法之后");

return proceed;

}

}

3.在spring配置文件中开启Aspectj生成代理对象，Spring会去找@Aspect标记的类，创建对应的代理对象

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

4.测试

//测试

public class TestUser {

@Test

public void test2() {

ApplicationContext context = new ClassPathXmlApplicationContext("conf/bean1.xml");

UserService bean = context.getBean("userService",UserService.class);

System.out.println("add "+bean.add());

}

}

5.如何实现相同切入点抽取 使用@Pointcut注解抽取出相同切入点

//增强类

@Component

@Aspect

public class UserAdvice {

//相同切入点抽取

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

public void point() {

}

@Before(value="point()")

public void before1() {

System.out.println("前置通知--before");

}

@After(value="point()")

public void after1(){

System.out.println("后置置增强---after");

}

@Around(value="point()")

public Object round1(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{

System.out.println("环绕-方法之前--around start");

Object proceed = proceedingJoinPoint.proceed();

System.out.println("环绕-方法之后--around end");

return proceed;

}

@AfterReturning(value="point()")

public void returing1() throws Throwable{

System.out.println("最终通知---after-returning");

}

@AfterThrowing(value="point()")

public void error1() throws Throwable{

System.out.println("异常通知--after-throwing");

}

#### 5.3完全注解

1.使用注解方式创建增强类和被增强类

2.使用@Aspect标记增强类且在作为通知方法上面添加通知类型注解，使用切入点表达式是配置完成方法的增强

3.使用配置类配置，开启注解扫描和Aspectj生成代理对象

@Configuration //标记为配置类

@ComponentScan(basePackages = "com.stu")//开启注解扫描

@EnableAspectJAutoProxy//开启Aspectj生成代理对象

public class SpringConfig {

}

# Spring Jdbc

[Spring](http://localhost:9093/itour/work/detail?id=123)

阅读  最后发布于 2022-08-05 21:14

## 1.什么是JdbcTemplate？

Spring对Jdbc进行了封装,使用JdbcTemplate方便对数据库的操作；

## 2.如何使用JdbcTemplate？

### 2.1基于XML方式

1.引入相关依赖

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>5.3.16</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.30</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.2.11</version>

</dependency>

2.在spring配置文件中,引入bean命名空间，配置数据库连接池

<!-- 数据库连接池 -->

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" destroy-method="close">

<property name="url" value="jdbc:mysql://1.116.226.147:3307/itour?serverTimezone=Asia/Shanghai"></property>

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

<property name="password" value="mysql@958958"></property>

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

</bean>

3.在spring配置文件中,配置JdbcTemplate对象，注入DataSource；

<!-- 配置JdbcTemplate -->

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

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

</bean>

4.创建Service类，创建dao类，在dao中注入JdbcTemplate对象；

1）创建Service类和dao类

public class AccountService {

AccountDao accountDao;

public void setAccountDao(AccountDao accountDao) {

this.accountDao = accountDao;

}

public List<Account> query() {

List<Account> queryAll = accountDao.queryAll();

return queryAll;

}

}

public class AccountDao {

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public List<Account> queryAll(){

String sql = "select \* from account";

List<Account> query = jdbcTemplate.query(sql,new BeanPropertyRowMapper<Account>(Account.class));

return query;

}

}

2）通过spring IOC创建Service和dao对象完成对象注入；

<!-- 配置Dao,注入JdbcTemplate,需要对应的set方法支持-->

<bean id="accountDao" class="com.stu.dao.AccountDao">

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

</bean>

<!-- 配置AccountService,注入AccountDao ,需要对应的set方法支持-->

<bean id="accountService" class="com.stu.service.AccountService">

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

</bean>

<bean id="account" class="com.stu.dao.Account"></bean>

3)测试

@Test

public void testClient() {

ApplicationContext context = new ClassPathXmlApplicationContext("classpath:conf/bean2.xml");

AccountService bean = context.getBean("accountService",AccountService.class);

List<Account> query = bean.query();

System.out.println(query);

}

### 2.2基于注解方式

1）引入相关依赖；

2）在spring配置文件中,引入bean命名空间，配置数据库连接池

<!-- 数据库连接池 -->

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" destroy-method="close">

<property name="url" value="jdbc:mysql://1.116.226.147:3307/itour?serverTimezone=Asia/Shanghai"></property>

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

<property name="password" value="mysql@958958"></property>

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

</bean>

3）在spring配置文件中,配置JdbcTemplate对象，注入DataSource，开启注解扫描；

<!-- 配置JdbcTemplate -->

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

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

</bean>

<!-- 开启注解扫描 -->

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

4）通过spring IOC注解方式创建Service和dao对象完成对象注入（需要开启注解扫描）

@Service

public class AccountService {

@Autowired

AccountDao accountDao;

public List<Account> query() {

List<Account> queryAll = accountDao.queryAll();

return queryAll;

}

}

@Repository

public class AccountDao {

@Autowired

private JdbcTemplate jdbcTemplate;

public List<Account> queryAll(){

String sql = "select \* from account";

List<Account> query = jdbcTemplate.query(sql,new BeanPropertyRowMapper<Account>(Account.class));

return query;

}

}

5)测试

@Test

public void testClient() {

ApplicationContext context = new ClassPathXmlApplicationContext("classpath:conf/bean2.xml");

AccountService bean = context.getBean("accountService",AccountService.class);

List<Account> query = bean.query();

System.out.println(query);

}

### 2.3完全注解方式

1）引入相关依赖

2）使用配置类开启注解扫描、配置数据库连接池和JdbcTemplate

@Configuration

@ComponentScan(basePackages = "com.stu")

public class SpringConfig {

@Bean

public DruidDataSource dataSource() {

DruidDataSource dataSource = new DruidDataSource();

dataSource.setUrl("jdbc:mysql://1.116.226.147:3307/itour?serverTimezone=Asia/Shanghai");

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

dataSource.setUsername("root");

dataSource.setPassword("mysql@958958");

return dataSource;

}

@Bean

public JdbcTemplate jdbcTemplate(DruidDataSource dataSource) {

JdbcTemplate jdbcTemplate = new JdbcTemplate();

jdbcTemplate.setDataSource(dataSource);

return jdbcTemplate;

}

}

3)构建对应的service和dao

@Service

public class AccountService {

@Autowired

AccountDao accountDao;

public List<Account> query() {

List<Account> queryAll = accountDao.queryAll();

return queryAll;

}

}

@Repository

public class AccountDao {

@Autowired

private JdbcTemplate jdbcTemplate;

public List<Account> queryAll(){

String sql = "select \* from account";

List<Account> query = jdbcTemplate.query(sql,new BeanPropertyRowMapper<Account>(Account.class));

return query;

}

}

4)测试

@Test

public void testClient() {

ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);

AccountService bean = context.getBean("accountService",AccountService.class);

List<Account> query = bean.query();

System.out.println(query);

}

## 3.使用JdbcTemplate操作数据库

### 1.添加、修改和删除操作

调用JdbcTemplate中update方法实现添加操作 sql:sql语句  
args:sql语句值

update(String sql,Object... args)

### 2.查询返回某个值

调用JdbcTemplate中queryForObject方法

queryForObject(String sql,Class<T> requiredType)

### 3.查询返回对象

调用JdbcTemplate中queryForObject方法  
RowMapper:是接口，返回不同类型的数据，使用这个接口的实现类完成数据的封装；

queryForObject(String sql,RowMapper<T> rowMapper,Object... args);

实例

public Book queryById(Long id){

String sql = "select \* from t\_book where id= ?";

Book book = jdbcTemplate.queryForObject(sql,new BeanPropertyRowMapper<Book>(Book.class),id);

return book;

}

### 4.查询返回集合

调用JdbcTemplate中query方法

query(String sql,RowMapper<T> rowMapper,Object... args);

实例

public List<Book> queryList(){

String sql = "select \* from t\_book";

List<Book> book = jdbcTemplate.query(sql,new BeanPropertyRowMapper<Book>(Book.class));

return book;

}

### 5.批量添加、修改、删除

调用JdbcTemplate中batchUpdate方法

batchUpdate(String sql,List<Object[]> batchArgs)

实例：

public void batchUpdate(List<Object[]> batchArgs){

String sql="增删改sql";

jdbcTemplate.batchUpdate(sql,batchArgs);

}

# Spring如何管理事务

## 1.事务概念

### 1.）什么是事务？

事务是逻辑上的一组操作,组成这组操作的各个逻辑单元，要么都成功，要么都都失败。

如银行转账，从第一个账户划出款项，将款项存入第二个账户；在这个过程中，两个环节是关联的。第一个账户划出款项必须保证正确的存入第二个账户，如果第二个环节没有完成，整个的过程都应该取消，否则就会发生丢失款项的问题。整个交易过程，可以看作是一个事务，成功则全部成功，失败则需要全部撤消

### 2.）事务的四个特征(ACID)?

1)原子性：事务中的全部操作在数据库中是不可分割的，要么全部完成，要么全部不执行。

2)一致性：在事务开始之前和事务结束以后，数据必须保持一致(符合所有预设规则);

3)隔离性：事务的执行不受其他事务的干扰，一个事务会不会读取到另一个未提交的事务修改的数据。

4)持久性：一个事务一旦提交，数据库中数据的改变是永久性的；

## 2.搭建事务操作环境（转账模拟)

1.创建数据库表，添加记录

create table account(

id INT PRIMARY KEY AUTO\_INCREMENT,

username varchar(20),

money FLOAT

)

2.使用配置类或者xml方式开启注解扫描，配置数据源和JdbcTemplate

@Configuration

@ComponentScan(basePackages = "com.stu")

public class SpringConfig {

@Bean

public DruidDataSource dataSource() {

DruidDataSource dataSource = new DruidDataSource();

dataSource.setUrl("jdbc:mysql://1.116.226.147:3307/itour?serverTimezone=Asia/Shanghai");

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

dataSource.setUsername("root");

dataSource.setPassword("mysql@958958");

return dataSource;

}

@Bean

public JdbcTemplate jdbcTemplate(DruidDataSource dataSource) {

JdbcTemplate jdbcTemplate = new JdbcTemplate();

jdbcTemplate.setDataSource(dataSource);

return jdbcTemplate;

}

}

3.构建Service ，dao完成对象的创建和注入并在dao实现类中实现账户划出和存入，在Service层中调用dao实现转账功能；

@Service

public class AccountService {

@Autowired

AccountDao accountDao;

public String transfer(Account from ,Account to) {

String transfer = accountDao.transfer (from,to);

return transfer;

}

}

@Repository

public class AccountDao {

@Autowired

private JdbcTemplate jdbcTemplate;

public String transfer(Account from,Account to){

String sql = "update account set money = money -? where id=?";

int update = jdbcTemplate.update(sql, from.getMoney(),from.getId());

String sql\_to = "update account set money = money +? where id=?";

update = jdbcTemplate.update(sql\_to, to.getMoney(),to.getId());

return "sucess";

}

}

4.测试

@Test

public void testClient() {

ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);

AccountService bean = context.getBean("accountService",AccountService.class);

Account from = new Account();

from.setId(1L);

from.setMoney(100F);

Account to = new Account();

to.setId(2L);

to.setMoney(100F);

String result = bean.transfer(from,to);

System.out.println(result);

}

5.总结

1）在没有事务的情况下，代码正常执行没有问题。

2）如果执行过程中出现异常，如当A账号转B账号100时，从A账号划出100元时出现了异常（断电），没有执行向B账号存入100的操作，就会导致A账户款项丢失。此时需要我们使用事务来解决该问题；

## 3.事务操作的过程

1.开启事务操作

2.进行业务操作

3.没有异常，提交事物

4.出现异常，事务回滚

## 4.spring事务管理

1.Spring 事务的本质是数据库对事务的支持，没有数据库的事务支持，Spring 是无法提供事务功能的。Spring 只提供统一事务管理接口，具体实现都是由各数据库自己实现，数据库事务的提交和回滚是通过数据库自己的事务机制实现。在使用时，事务一般需要添加到javaEE三层结构里的service层（业务逻辑层）；

### 1)Spring事务分类

在Spring中事务的管理有两种方式：编程式和声明式事务管理，开发中一般使用声明式事务管理方式。

#### 1.编程是事务管理

编程式事务管理使用 TransactionTemplate，需要显式执行事务。

#### 2.声明式事务管理

1.声明式事务管理建立在 AOP 之上的。其本质是通过 AOP 功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前启动一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

2.优点是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过 @Transactional 注解的方式，便可以将事务规则应用到业务逻辑中，减少业务代码的污染。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

## 5.声明式事务管理实现

在Spring进行声明式事物管理，底层使用的Spring AOP原理。Spring提供了一个接口，代表事务管理器，这个接口针对不同的框架提供了不同的实现类Spring。声明式事务管理方式有两种基于注解方式和基于xml配置文件方式，实际开发中一般使用注解方式；

### 1.）基于注解方式实现声明式事务管理

1.搭建事务操作环境  
使用注解或xml方式创建Service,dao对象及注入操作，在spring中配置文件中配置注解扫描、数据库连接池、JdbcTemplate（操作数据库）；  
2.在spring配置文件中配置对应的事务管理器

<!-- 创建事务管理器 -->

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

<!-- 注入数据源 -->

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

</bean>，

3.在spring配置文件中开启事务注解  
1)在Spring配置文件中引入命名空间tx

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

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

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

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

xsi:schemaLocation="

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.1.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.1.xsd

http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc-4.1.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.1.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.1.xsd">

2)开启事务注解

<!-- 开启事务注解

transaction-manager：事务管理器的Id

-->

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

4.在Service类上面或者方法上添加事务注解  
1）@Transactional,这个注解可以添加在类上，也可以添加在方法上；  
2）如果把@Transactional注解添加在类上，该类里面的所有方法都会添加事务；  
3）如果把@Transactional注解添加在方法上，只会在该方法上添加事务；

@Service

public class AccountService {

@Autowired

AccountDao accountDao;

@Transactional

public String transfer(Account from ,Account to) {

String transfer = accountDao.transfer (from,to);

return transfer;

}

}

5)测试

@Test

public void testClient() {

ApplicationContext context = new ClassPathXmlApplicationContext("classpath:conf/Bean2.xml");

AccountService bean = context.getBean("accountService",AccountService.class);

Account from = new Account();

from.setId(1L);

from.setMoney(100F);

Account to = new Account();

to.setId(2L);

to.setMoney(100F);

String result = bean.transfer(from,to);

System.out.println(result);

}

### 2.）使用完全注解实现声明式事务管理

1.配置配置类

@Configuration

@ComponentScan(basePackages = "com.stu")//开启注解扫描

@EnableTransactionManagement//开启事务注解

public class SpringConfig {

//配置数据源

@Bean

public DruidDataSource dataSource() {

DruidDataSource dataSource = new DruidDataSource();

dataSource.setUrl("jdbc:mysql://1.116.226.147:3307/itour?serverTimezone=Asia/Shanghai");

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

dataSource.setUsername("root");

dataSource.setPassword("mysql@958958");

return dataSource;

}

//配置JdbcTemplate

@Bean

public JdbcTemplate jdbcTemplate(DruidDataSource dataSource) {

JdbcTemplate jdbcTemplate = new JdbcTemplate();

jdbcTemplate.setDataSource(dataSource);

return jdbcTemplate;

}

//配置事务管理器

@Bean

public DataSourceTransactionManager transactionManager(DruidDataSource dataSource) {

DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();

transactionManager.setDataSource(dataSource);

return transactionManager;

}

}

2.在Servic层添加注解

@Service

public class AccountService {

@Autowired

AccountDao accountDao;

@Transactional

public String transfer(Account from ,Account to) {

String transfer = accountDao.transfer (from,to);

return transfer;

}

}

3.测试

@Test

public void testClient() {

ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);

AccountService bean = context.getBean("accountService",AccountService.class);

Account from = new Account();

from.setId(1L);

from.setMoney(100F);

Account to = new Account();

to.setId(2L);

to.setMoney(100F);

String result = bean.transfer(from,to);

System.out.println(result);

}

### 3.）XML方式实现声明式事务管理

1.搭建事务操作环境  
使用注解或xml方式创建Service,dao对象及注入操作，在spring中配置文件中配置注解扫描、数据库连接池、JdbcTemplate；

2.在Spring配置文件中配置

1）配置事务管理器

<!-- 创建事务管理器 -->

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

<!-- 注入数据源 -->

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

</bean>

2）配置通知、切入点和切面

<!-- 配置通知 -->

<tx:advice id="txadvice">

<!-- 配置事务参数 -->

<tx:attributes>

<tx:method name="insert\*" propagation="REQUIRED"/>

<tx:method name="update\*" propagation="REQUIRED"/>

<tx:method name="delete\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

<!-- 配置切入点和切面 -->

<aop:config>

<!-- 配置切入点 -->

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

<!-- 配置切面 -->

<aop:advisor advice-ref="txadvice" pointcut-ref="pt"/>

</aop:config>

3)测试

@Service

public class AccountService {

@Autowired

AccountDao accountDao;

public String updateAccount(Account from ,Account to) {

String transfer = accountDao.transfer (from,to);

return transfer;

}

}

@Test

public void testClient() {

ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);

AccountService bean = context.getBean("accountService",AccountService.class);

Account from = new Account();

from.setId(1L);

from.setMoney(100F);

Account to = new Account();

to.setId(2L);

to.setMoney(100F);

String result = bean.updateAccount(from,to);

System.out.println(result);

}

### 4）声明式事务实现原理

SPring 声明式事务底层通过AOP/动态代理来实现；

在Bean初始化阶段创建代理对象：Spring容器在初始化每个单例bean的时候，会遍历容器中的所有BeanPostProcessor实现类，并执行其postProcessAfterInitialization方法，在执行AbstractAutoProxyCreator类的postProcessAfterInitialization方法时会遍历容器中所有的切面，查找与当前实例化bean匹配的切面，这里会获取事务属性切面，查找@Transactional注解及其属性值，然后根据得到的切面创建一个代理对象，默认是使用JDK动态代理创建代理，如果目标类是接口，则使用JDK动态代理，否则使用Cglib。

在执行目标方法时进行事务增强操作：当通过代理对象调用Bean方法的时候，会触发对应的AOP增强拦截器，声明式事务是一种环绕增强，对应接口为MethodInterceptor，事务增强对该接口的实现为TransactionInterceptor，事务拦截器TransactionInterceptor在invoke方法中，通过调用父类TransactionAspectSupport的invokeWithinTransaction方法进行事务处理，包括开启事务、事务提交、异常回滚。

## 6.声明式事务管理参数配置

### 1.事务的传播行为（propagation）

当一个事务方法被另外一个事务方法调用时，这个事务方法（被调用方法）如何进行事务处理。传播机制是针对被调用者，控制它是否被传播或者被怎样传播。事务传播行为可以由传播属性指定，Spring中定义了7种类传播行为：REQUIRED、REQUIRES\_NEW、SUPPORTS、MANDATORY、NOT\_SUPPORTED、NEVER、NESTED。

|  |  |
| --- | --- |
| **被调用者方法中事务传播类型** | **说明** |
| REQUIRED | REQUIRED是Spring默认的传播行为。如果当前没有事务，则自己新建一个事务，如果当前存在事务则加入这个事务。 |
| REQUIRES\_NEW | 创建一个新事务，如果存在当前事务，则挂起该事务。 |
| SUPPORTS | 当前存在事务，则加入当前事务，如果当前没有事务，就以非事务方法执行。 |
| NESTED | 如果当前事务存在，则在嵌套事务中执行，否则新建一个事务。 |
| MANDATORY | 当前存在事务，则加入当前事务，如果当前事务不存在，则抛出异常。 |
| NOT\_SUPPORTED | 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起(务挂起之后，任何操作都不在该事务的控制之下） |
| NEVER | 以非事务方式执行，如果当前存在事务，则抛出异常 |

#### 1)required：

1.当调用者方法（当前方法）上没有事务，被调用方法会创建一个新的事务；

2.当调用者方法(当前方法)上有事务，被调用方法会加入到调用者方法的事务中； 总结：只创建一个事务，

#### 2)required\_new:

不管调用者方法(当前方法)中有没有事务，被调用者方法都会创建一个新事务。如果调用者方法中存在事务，则会挂起当前事务(事务挂起之后，任何操作都不在该事务的控制之下)，老事务的回滚，不会影响新事务的提交。

#### 3)nested：

1.如果调用者方法中(当前方法）存在事务，则被调用的方法会嵌套在当前事务内执行。

2.如果调用者方法中(当前方法）不存在事务,被调用者方法会创建一个新事务。

#### 4)supports：

1.如果调用者方法中(当前方法)存在事务，被调用者方法会加入到调用者方法的事务中；

2.如果调用者方法中(当前方法)不存在事务，被调用者方法以非事务的方式执行；

总结：supports不创建事务，当前方法有事务加入到当前事务，当前方法没有事务以非事务方式运行；

#### 5)not\_supported：

1.如果调用者方法中(当前方法)存在事务，将当前事务挂起；

2.如果调用者方法中(当前方法)不存在事务，以非事务方式执行；

总结：不支持事务；

#### 6)mandatory:

1.如果调用者方法中(当前方法)存在事务,被调用者方法会加入到调用者方法的事务中；

2.如果调用者方法中(当前方法)不存在事务,抛出异常；

总结：强制，必须使用事务；

#### 7)never：

1.如果调用者方法中存在事务，则抛出异常；

2.如果调用者方法中不存在事务，以非事务方式执行；

总结：禁止事务，必须在一个没有事务中执行，否则报错。

### 2.isolation:事务隔离级别

事务隔离级别是针对数据库访问的并发性问题而言的，而数据库访问的并发性问题指多个事务可以同时访问数据库中的数据，而当多个事务在数据库中并发执行（同时执行）时，数据的一致性可能受到破坏，从而导致数据出现问题。

#### 1.数据库访问的并发性带来的问题

##### 1.更新丢失:

1.)如果多个事务同时（并发）对数据库表中的同一条记录进行修改，那么后修改的记录将会覆盖前面修改的记录，前面的修改就丢失掉了，这就叫做更新丢失。

2)更新丢失和多线程同步很相似，所以解决方法也是一样的，那就是对行加锁，同时只允许一个事务访问数据库。即就是加锁以后，两个事务即使同时访问数据库，也只允许加锁的事务先访问，另一个未加锁的在外等待，直到释放锁后才能访问数据库。

##### 2.脏读

1.)所谓脏读就是一个事务A读取另一个事务B修改但尚未提交的数据并在此基础上操作，而事务 B 又执行事务回滚（也就是撤销了事务），那么事务 A 读取到的数据就是脏数据。

2.)解决办法就是在第一个事务提交前，任何其他事务不可读取其修改过的值，则可以避免该问题。

##### 3.不可重复读

1.)所谓不可重复读就是一个事务对同一行数据重复读取两次，但是却得到了不同的结果。事务T1读取某一数据后，事务T2对其做了修改，当事务T1再次读该数据时得到与前一次不同的值.

2.)解决办法:如果只有在修改事务完全提交之后才可以读取数据，则可以避免该问题。

##### 4.幻读

1.)所谓幻象读指两次执行同一条 select 语句会出现不同的结果。即当第一次执行 select 语句后，接着另一个事务执行了 insert 语句（也就是插入了一条记录），这时第二次执行相同的 select 语句，返回的结果自然与第一次不同，这就是幻象读。

2.）解决办法就是如果在操作事务完成数据处理之前，任何其他事务都不可以添加新数据，则可避免该问题。

#### 2.事务的隔离级别

为了解决以上各种数据库访问的并发性问题（更新丢失、脏读、不可重复读、幻象读），为此数据库提供了4种隔离级别。Spring的接口TransactionDefinition中定义了表示隔离级别的常量，当然其实主要还是对应数据库的事务隔离级别：

1.ISOLATION\_DEFAULT:数据库默认的隔离界别,MySQL 默认可重复读，Oracle 默认读已提交。

2.ISOLATION\_READ\_UNCOMMITTED：读未提交。

如果一个事务A已经开始写数据，则另外一个事务B则不允许同时进行写操作，但允许读此行数据。该隔离级别可以通过“排他写锁”实现。这样就避免了更新丢失，却可能出现脏读。也就是说事务B读取到了事务A未提交的数据。

3.ISOLATION\_READ\_COMMITTED：读已提交

读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行。该隔离级别避免了脏读，但是却可能出现不可重复读。事务A事先读取了数据，事务B紧接了更新了数据，并提交了事务，而事务A再次读取该数据时，数据已经发生了改变。

4.ISOLATION\_REPEATABLE\_READ：可重复读

可重复读是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，即使第二个事务对数据进行修改，第一个事务两次读到的的数据是一样的。这样就发生了在一个事务内两次读到的数据是一样的，因此称为是可重复读。读取数据的事务将会禁止写事务（但允许读事务），写事务则禁止任何其他事务。这样避免了不可重复读取和脏读，但是有时可能出现幻象读。（读取数据的事务）这可以通过“共享读锁”和“排他写锁”实现。

5.ISOLATION\_SERIALIZABLE：序列化

提供严格的事务隔离。它要求事务序列化执行，事务只能一个接着一个地执行，但不能并发执行。如果仅仅通过“行级锁”是无法实现事务序列化的，必须通过其他机制保证新插入的数据不会被刚执行查询操作的事务访问到。序列化是最高的事务隔离级别，同时代价也花费最高，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻像读。

6.总结：

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。 对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为 Read Committed（授权读取、读提交）。它能够避免脏读取，而且具有较好的并发性能。尽管它会导致不可重复读、幻读和丢失更新这些并发性问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制。

#### 3.悲观锁和乐观锁

虽然数据库的隔离级别可以解决大多数问题，但是灵活度较差，为此又提出了悲观锁和乐观锁的概念。

##### 1）悲观锁

悲观锁就是某事务在更新数据过程中将数据锁定，其他任何事务都不能读取或修改，必须修改完成后才能访问数据（类似于Java的线程同步机制）。悲观锁的特点是具有排他性，通常依赖于数据库的锁机制，一般适合短事务处理。当一个事务加了悲观锁，其他任何事务是不能读取或修改数据，也就是只能在外面等待，什么事也干不了，直到悲观锁被释放为止。如果有很多事务都要访问数据库（高并发的情况），加了悲观锁就意味所有事务需要排着长长的队，一个一个访问数据库，那么访问数据库的效率非常低，所以叫悲观锁。

##### 2）乐观锁

乐观锁相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以只会在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则返回用户错误的信息，让用户决定如何去做。乐观锁的特点是并发性较好，事务修改数据时，其他事务仍可以修改数据。

实现乐观锁一般来说有以下2种方式：

1.使用版本号

使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。

2.使用时间戳

乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

说白了，乐观锁其实根本不是一种数据库锁机制，而是一种冲突检测机制，这种冲突检测机制是依赖软件或应用程序实现的。乐观在它的并发性比悲观锁好，一个事务在修改数据时，其他事务仍然可以修改数据。

##### 3）悲观锁与乐观锁的优缺点及使用场景

悲观锁的优点是可以保障数据库的数据是绝对安全的，它是依赖数据库的锁机制，能很好的解决数据库访问的并发性问题，但是缺点就是会导致数据库访问性能低下，所以适合短事务（也就是事务执行时间很短）的情况。如果事务执行时间很长，那么后面的事务就得一直排队等待嘛。它的使用场景是对数据安全性要求非常高的场景，比如银行系统、金融系统等。

乐观锁的优点是可以保障并发性比较好，也就是数据库访问性能可以，它是依赖软件的冲突检测机制实现的，但是缺点就是并没彻底解决数据库访问的并发性问题，所以数据库的数据不是绝对安全的。它的使用场景是对数据安全性要求不高而对性能要求很高的场景，比如各种信息管理系统等。

相关博客：https://www.jb51.net/article/226652.htm

### 3.timeout：超时时间

### 4.readOnly:是否只读

### 5.rollbackFor：回滚

### 6.noRollbackFor：不回滚

## 7.声明式事务在哪些情况下会失效？

### 1.@Transactional 应用在非 public 修饰的方法上

因为在Spring AOP 代理时,事务拦截器在目标方法执行前后进行拦截，获取Transactional 注解的事务配置信息时，会检查目标方法的修饰符是否为 public，不是 public则不会获取@Transactional 的属性配置信息

### 2.@Transactional 注解属性 propagation 设置错误

1)SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。

2)NOT\_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。

3)NEVER：以非事务方式运行，如果当前存在事务，则抛出异常

### 3.@Transactional 注解属性 rollbackFor 设置错误

rollbackFor 可以指定能够触发事务回滚的异常类型。Spring默认抛出了未检查unchecked异常（继承自 RuntimeException的异常）或者 Error才回滚事务，其他异常不会触发回滚事务。

### 4.同一个类中方法调用，导致@Transactional失效

由于使用Spring AOP代理造成的，因为只有当事务方法被当前类以外的代码调用时，才会由Spring生成的代理对象来管理。

相关博客： http://cloud.laidianping.cn/?article/990121