

Spring框架

Spring框架

理论概念

入门使用

IOC

概念

IOC的底层原理

IOC接口

IOC操作Bean管理

基于xml方式

基于xml方式创建对象

基于xml方式注入属性

p名称空间注入（了解）

xml注入其他类型属性

一、 字面量

二、注入属性--外部bean

三、注入属性--内部bean和级联赋值

xml注入集合属性

IOC操作Bean管理（工厂bean）

IOC操作Bean管理（bean作用域）

IOC操作Bean管理（bean生命周期）

bean的后置处理器

IOC操作Bean管理（自动装配）

IOC操作Bean管理（外部属性文件）

IOC操作Bean管理（基于注解方法）

注解概念

Spring针对Bean管理中创建对象提供注解

基于注解方法实现创建对象

组件扫描配置

基于注解方式的属性注入

纯注解开发

AOP

底层原理

动态代理

AOP术语

AOP(JDK动态代理)

一、使用JDK动态代理，使用Proxy类里面的方法创建代理对象

二、编写JDK动态代理代码

AOP操作

准备

AspectJ注解

AspectJ配置文件

完全注解

JdbcTemplate

概念

准备工作

JdbcTemplate操作数据库

添加

修改

删除

查询（查询返回某个值）

查询（查询返回集合）

批量添加

- 批量修改
- 事务
 - 事务操作（搭建事务操作环境）
 - 事务操作
 - Spring 事务管理介绍
 - 事务操作（注解声明式事务管理）
 - 事务操作（声明式事务管理参数配置）
 - 事务操作（XML声明式事务管理）
 - 事务操作（完全注解方式）

理论概念

轻量级的开源的JavaEE框架，可以解决企业应用开发的复杂性。AOP编程支持，方便程序测试，方便与其他框架尽情整合，方便进行事务操作，降低API开发难度。

减轻对项目模块之间的管理，类和类之间的管理，帮关注开发人员创建对象，管理对象之间的关系。

Spring核心技术ioc，aop。能实现模块之间，类之间的解耦合。

依赖：类A中使用类B的属性或者方法，叫做类A依赖类B

入门使用

1.创建类

```
public class User {  
    public void add(){  
        System.out.println("add....");  
    }  
}
```

2.配置bean.XML文件

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
        xsi:schemaLocation="http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
    <!-- 配置user类的创建-->  
    <bean id="user" class="com.qiangliu8.spring5.User"></bean>  
</beans>
```

3.单元测试

IOC

概念

控制反转，是一个理论，概念，思想。

把对象的创建，赋值，管理工作都交给代码之外的容器实现，也就是对象的创建是有其他外部资源完成。

控制：创建对象，对象的属性赋值，对象之间的关系管理。

反转：把原来的开发人员管理、创建对象的权限转移给代码之外的容器实现，由容器代替开发人员管理对象。创建对象，给属性赋值。

正转：由开发人员在代码中，使用new构造方法创建对象，开发人员主动管理对象。

容器：是一个服务器软件，一个框架。

为什么要使用IOC：目的是减少对代码的改动，也能实现不同的功能。实现解耦合。

Java中创建方法的方式

1. 构造方法，new XX()
2. 反射
3. 序列化
4. 克隆
5. ioc:容器创建对象
6. 动态代理

IOC的体现：

servlet：

1. 创建类继承HttpServlet
2. 在web.xml注册servlet，使用myservlet

MyServlet

3. 没有创建Servlet对象，没有 MyServlet myservlet = new MyServlet()
4. Servlet是Tomcat服务器他帮你创建的。

Tomcat作为容器：里面存放的有Servlet对象，Listener,Filter对象

IOC的技术实现：

DI（依赖注入）是IOC的技术实现，只需要在程序中提供要使用的对象名称就可以了，至于对象如何在容器中创建，赋值，查找都是容器内部实现的。

Spring底层创建对象，使用的是反射机制。

IOC的底层原理

1. XML解析

xml配置文件，配置创建对象

```
<bean id="user" class="com.qiangliu8.spring5.User"></bean>
```

2. 工厂模式

创建工厂类

```

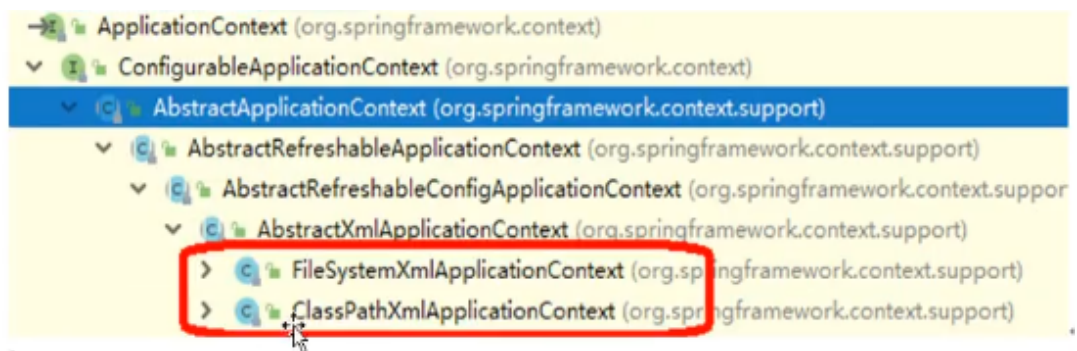
public UserFactory{
    public static UserDao getDao(){
        String classValue = class属性值//xml解析
        Class class = Class.forName(ClassValue)//通过反射创建对象
        return (UserDao)class.newInstance();
    }
}

```

3. 反射

IOC接口

1. IOC思想基于IOC容器完成，底层就是对象工厂。
2. Spring提供IOC容器实现的两个方式：
 1. **BeanFactory**：IOC容器的基本实现,内部使用的接口，一般不使用。加载配置文件时不会创建对象，在获取对象使用才会去创建对象，
 2. **ApplicationContext**：BeanFactory的子接口，提供更多更强大的功能，一般开发人员使用。加载配置文件时创建对象，
3. ApplicationContext接口有实现类



```

public class SpringTest {
    @Test
    public void testAdd(){
        //1.加载Spring的配置文件
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean1.xml");
        //2.获取配置创建的对象
        User user = context.getBean("user", User.class);
        user.add();
    }
}

```

IOC操作Bean管理

两种操作

1. Spring创建对象
2. Spring注入属性

两种方式

1. 基于xml配置文件方式实现

2. 基于注解方式实现

基于xml方式

基于xml方式创建对象

在spring配置文件中，使用bean标签，标签里面添加对应属性，就可以实现对象创建

```
<bean id="user" class="com.qiangliu8.User"></bean>
```

常用属性	
id	唯一标识
class	创建对象所在包类的路径
name	和id一样，但是可以支持一些特殊字符

创建对象时，默认执行对象的无参数构造函数

基于xml方式注入属性

DI：依赖注入，就是注入属性。

第一种注入方式--SET()

```
public class Book {  
    private String name;  
    private String sex;  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getSex() {  
        return sex;  
    }  
    public void setSex(String sex) {  
        this.sex = sex;  
    }  
}
```

```
<bean id="book" class="com.qiangliu8.spring5.Book">  
    <!-- 使用property定成属性注入-->  
    <!-- name:类里面属性名称-->  
    <!-- value:向属性注入的值-->  
    <property name="name" value="易经晶"></property>  
    <property name="sex" value="男"></property>  
</bean>
```

第二种注入方式--有参构造注入

```
public class Order {
    private String oname;
    private String address;
    public Order(String oname, String address) {
        this.oname = oname;
        this.address = address;
    }
}
```

```
<bean id="orders" class="com.qiangliu8.spring5.Order">
    <constructor-arg name="oname" value="No.1"></constructor-arg>
    <constructor-arg name="address" value="包河花园"></constructor-arg>
</bean>
```

p名称空间注入 (了解)

使用p名称空间注入，可以简化基于xml配置方式

1. 第一步添加p名称空间配置

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

2. 使用set方法注入

```
<bean id="book1" class="com.qiangliu8.spring5.Book" p:name="九阳神功"
p:sex="女"></bean>
```

xml注入其他类型属性

一、字面量

1. null值

```
<bean id="book" class="com.qiangliu8.spring5.Book">
    <property name="address">
        <null></null>
    </property>
</bean>
```

2. 特殊字符

把<>话转义或者特殊符号内容写到CDATA

```
<bean id="book2" class="com.qiangliu8.spring5.Book">
    <property name="name">
        <value><![CDATA[<<南京>>]]></value>
    </property>
</bean>
```

二、注入属性--外部bean

1. 创建两个类service类和dao类

UserDaoImpl类

```
public class UserDaoImpl implements UserDao{
    @Override
    public void update() {
        System.out.println("UserDaoImpl--update");
    }
}
```

UserService类

```
public class UserService {

    //创建UserDao属性，生成set方法
    private UserDao userDao;

    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

    public void add(){
        System.out.println("service-add");
        userDao.update();
    }
}
```

2. 在service调用dao方法

如上

3. 在spring配置文件中配置

```
<!--service对象创建-->
<bean id="userService" class="com.qiangliu8.service.UserService">
    <!--注入userDao的对象 name: 类里面属性的名称 ref: 创建userDao对象bean标签id
    值-->
    <property name="userDao" ref="userDao1"></property>
</bean>
<!--dao对象创建-->
<bean id="userDao1" class="com.qiangliu8.dao.UserDaoImpl"></bean>
```

4. 单元测试

```
public class BeanTest {
    @Test
    public void testOrders(){
        ApplicationContext applicationContext = new
        ClassPathXmlApplicationContext("bean2.xml");
        UserService userService = (UserService)
        applicationContext.getBean("userService");
        userService.add();
    }
}
```

三、注入属性--内部bean和级联赋值

Dept类

```
public class Dept {  
    private String dname;  
  
    public void setDname(String dname) {  
        this.dname = dname;  
    }  
  
    @Override  
    public String toString() {  
        return "Dept{" +  
            "dname='" + dname + '\'' +  
            '}';  
    }  
}
```

Emp类

```
public class Emp {  
    private String Ename;  
    private String gender;  
    private Dept dept;  
  
    public void setDept(Dept dept) {  
        this.dept = dept;  
    }  
  
    public void setEname(String ename) {  
        Ename = ename;  
    }  
  
    public void setGender(String gender) {  
        this.gender = gender;  
    }  
  
    @Override  
    public String toString() {  
        return "Emp{" +  
            "Ename='" + Ename + '\'' +  
            ", gender='" + gender + '\'' +  
            ", dept=" + dept +  
            '}';  
    }  
}
```



```

<!--内部bean-->
<bean id="emp" class="com.qiangliu8.bean.Emp">
    <!--设置普通属性-->
    <property name="ename" value="刘强"></property>
    <property name="gender" value="男"></property>
    <!--设置对象类型属性-->
    <property name="dept">
        <bean id="dept" class="com.qiangliu8.bean.Dept">
            <property name="dname" value="财务部"></property>
        </bean>
    </property>
</bean>

```

内部bean

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--内部bean-->
    <bean id="emp" class="com.qiangliu8.bean.Emp">
        <!--设置普通属性-->
        <property name="ename" value="刘强"></property>
        <property name="gender" value="男"></property>
        <!--设置对象类型属性-->
        <property name="dept">>
            <bean id="dept" class="com.qiangliu8.bean.Dept">
                <property name="dname" value="财务部"></property>
            </bean>
        </property>
    </bean>
</beans>

```

级联赋值

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--内部bean-->
    <bean id="emp" class="com.qiangliu8.bean.Emp">
        <!--设置普通属性-->
        <property name="ename" value="刘强"></property>
        <property name="gender" value="男"></property>
        <!--设置对象类型属性-->
        <property name="dept" ref="dept"></property>
        <property name="dept.dname" value="社联部1"></property>
    </bean>

    <bean id="dept" class="com.qiangliu8.bean.Dept">
        <property name="dname" value="社联部"></property>
    </bean>
</beans>

```

xml注入集合属性

1. 注入数组、注入List集合类型属性、注入Map集合类型属性、注入Set集合类型属性

```
public class Stu {  
    //数组类型属性  
    private String[] courses;  
    //List集合类型属性  
    private List<String> list;  
    //Map集合类型属性  
    private Map<String,String> map;  
    //Set集合类型属性  
    private Set<String> set;  
  
    public void setList(List<String> list) { this.list = list; }  
  
    public void setSet(Set<String> set) { this.set = set; }  
  
    public void setMap(Map<String, String> map) { this.map = map; }  
  
    public void setCourses(String[] courses) { this.courses = courses; }  
}
```

2. 创建类，定义数组、list、map、set类型属性，生成对应set方法
3. 在Spring配置文件中配置

```
<bean id="stu" class="com.qiangliu8.collectiontype.Stu">  
    <!--数组类型注入-->  
    <property name="courses">  
        <array>  
            <value>java课程</value>  
            <value>数据库课程</value>  
        </array>  
    </property>  
    <!--list类型属性注入-->  
    <property name="list">  
        <list>  
            <value>list1</value>  
            <value>list2</value>  
            <value>list3</value>  
        </list>  
    </property>  
    <!--map类型属性注入-->  
    <property name="map">  
        <map>  
            <entry key="key1" value="value2"></entry>  
            <entry key="key2" value="value2"></entry>  
        </map>  
    </property>  
    <!--set类型属性注入-->  
    <property name="set">  
        <set>  
            <value>set1</value>  
        </set>  
    </property>  
</bean>
```

```

        <value>set2</value>
    </set>
</property>
</bean>

```

4. 注入属性是List集合，里面是对象时

```

<bean id="stu" class="com.qiangliu8.collectiontype.Stu">
    <!-- 注入list集合，但值是对象-->
    <property name="courseList">
        <list>
            <ref bean="course1"></ref>
            <ref bean="course2"></ref>
        </list>
    </property>
</bean>

<!-- 创建多个course对象-->
<bean id="course1" class="com.qiangliu8.collectiontype.Course">
    <property name="cname" value="Spring5"></property>
</bean>
<bean id="course2" class="com.qiangliu8.collectiontype.Course">
    <property name="cname" value="MyBatis"></property>
</bean>

```

5. 把集合注入部分提取出来

1. 在spring配置文件中引入名称空间

```

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

```

2. 创建Book类

```

public class Book {
    private List<String> list;

    public void setList(List<String> list) {
        this.list = list;
    }
}

```

3. 提取list集合类型属性注入

```

<util:list id="bookList">
    <value>语文</value>
    <value>数学</value>
    <value>英语</value>
</util:list>

<bean id="book" class="com.qiangliu8.collectiontype.Book">
    <property name="list" ref="bookList"></property>
</bean>

```

IOC操作Bean管理 (工厂bean)

Spring有两种类型Bean,一种普通Bean, 一种工厂Bean (FactoryBean)

普通bean:在配置文件中bean定义的类型就是返回类型

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FactotyBean

```

public class MyBean implements FactoryBean<Course> {
    @Override
    public Course getObject() throws Exception {
        Course course = new Course();
        course.setName("大数据");
        return course;
    }

    @Override
    public Class<?> getObjectType() {
        return null;
    }

    @Override
    public boolean isSingleton() {
        return false;
    }
}

```

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

```

ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("bean3.xml");
Course course = applicationContext.getBean("myBean", Course.class);
System.out.println(course);

```

Spring配置

```

<bean id="myBean" class="com.qiangliu8.collectiontype.factorybean.MyBean">
</bean>

```

IOC操作Bean管理 (bean作用域)

1spring里面，设置创建bean实例时单实例还是多实例

默认情况下，bean是单实例对象，scope属性值为singleton

```
ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("bean2.xml");
Book book1 = applicationContext.getBean("book",Book.class);
Book book2 = applicationContext.getBean("book",Book.class);
System.out.println(book1==book2);//true, 对象是一样的
```

scope有两个值:

1. 默认值，singleton，表示是单实例对象。加载spring配置文件时创建对象，在调用getBean方法时创建多实例对象

```
<bean id="book" class="com.qiangliu8.collectiontype.Book"
scope="singleton">
```

2. prototype，表示是多实例对象，不是在加载spring配置文件时创建对象，在调用getBean方法时创建多实例对象

```
<bean id="book" class="com.qiangliu8.collectiontype.Book"
scope="prototype">
```

3. request、session

IOC操作Bean管理 (bean生命周期)

对象从创建到销毁的过程。

1. 通过构造器创建bean实例（无参数构造）
2. 为bean的属性设置值和对其他bean引用（调用set方法）
3. 调用bean初始化的方法（需要进行配置）
4. bean可以使用（对象获取到了）
5. 容器关闭后，调用bean的销毁方法（需要进行配置的销毁方法）

演示部分

```
public class Order {
    private String oname;

    public Order() {
        System.out.println("第一步，执行无参构造函数创建bean实例!");
    }

    public void setOname(String oname) {
        this.oname = oname;
        System.out.println("第二步，调用set方法设置属性的值");
    }
    //创建执行的初始化方法
    public void initMethod(){
        System.out.println("第三步 执行初始化的方法");
    }
    public void destoryMethod(){
```

```

        System.out.println("第五步，执行销毁的方法");
    }
}

```

单元测试

```

@Test
public void test3(){
    ApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("bean4.xml");
    Order order = applicationContext.getBean("order",Order.class);
    System.out.println("第四步，获取创建实例对象");
    System.out.println(order);
    //手动让bean实例销毁 applicationContext父类没有close方法，需要强转实现类的close方法
    ((ClassPathXmlApplicationContext)applicationContext).close();
}

```

Spring配置

```

<bean id="order" class="com.qiangliu8.collectiontype.bean.Order" init-
method="initMethod" destroy-method="destoryMethod">
    <property name="oname" value="小米11"></property>
</bean>

```

bean的后置处理器

加上手指处理器，生命周期有**七步**

1. 通过构造器创建bean实例（无参数构造）
2. 为bean的属性设置值和对其他bean引用（调用set方法）
3. **把bean实例传递bean后置处理器的方法postProcessBeforeInitialization（）**
4. 调用bean初始化的方法（需要进行配置）
5. **把bean实例传递bean后置处理器的方法postProcessAfterInitialization**
6. bean可以使用（对象获取到了）
7. 容器关闭后，调用bean的销毁方法（需要进行配置的销毁方法）

实现过程

1. 创建类，实现接口BeanPostProcessor

```

public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String
    beanName) throws BeansException {

        System.out.println("在初始化之前执行的方法");
        return null;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String
    beanName) throws BeansException {
        System.out.println("在初始化之后执行的方法");
        return null;
    }
}

```

```
}
```

2. Spring配置后置处理器，自动就给beans里的其他bean对象添加了后置处理器

```
<bean id="order" class="com.qiangliu8.collectiontype.bean.Order" init-  
method="initMethod" destroy-method="destoryMethod">  
    <property name="oname" value="小米11"></property>  
</bean>  
  
<!--      配置后置处理器-->  
<bean id="myBeanPostProcessor"  
class="com.qiangliu8.collectiontype.bean.MyBeanPostProcessor"></bean>
```

IOC操作Bean管理（自动装配）

手动装配：在xml通过value name值 配置属性

自动装配：根据指定装配规则（属性名称或者属性类型），Spring自动将匹配的属性值进行注入。

演示：

Emp 和Dept类

```
public class Emp {  
    private Dept dept;  
  
    public void setDept(Dept dept) {  
        this.dept = dept;  
    }  
  
    @Override  
    public String toString() {  
        return "Emp{" +  
            "dept=" + dept +  
            '}';  
    }  
    public void test(){  
        System.out.println(dept);  
    }  
}
```

spring配置

```
<!--      实现自动装配-->  
<!--      bean标签属性autowire,配置自动装配-->  
<!--      autowire有两个值: -->  
<!--          byName根据属性名称注入 注入bean的id值和类属性名称一样 -->  
<!--byType根据属性类型注入-->  
<bean id="emp" class="com.qiangliu8.collectiontype.autowire.Emp"  
autowire="byName"></bean>  
<bean id="dept" class="com.qiangliu8.collectiontype.autowire.Dept"></bean>
```

注意使用byType时，相同类的bean不能有多个

IOC操作Bean管理（外部属性文件）

两种方法:

1. 直接配置数据库信息

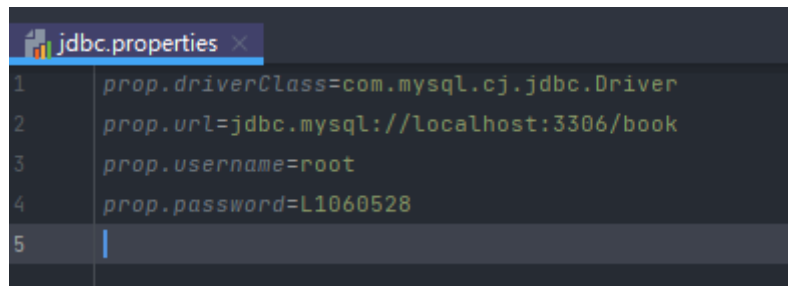
1.配置德鲁伊连接池，引入 druid-1.1.9.jar 包

2.spring配置

```
<!-- 直接配置连接池-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver">
</property>
    <property name="url" value="jdbc:mysql://localhost:3306/book">
</property>
    <property name="username" value="root"></property>
    <property name="password" value="Lq060528"></property>
</bean>
```

2. 引入外部属性文件配置数据库连接池

1. 创建外部属性文件，properties格式文件，写数据库信息



```
jdbc.properties
1 prop.driverClass=com.mysql.cj.jdbc.Driver
2 prop.url=jdbc:mysql://localhost:3306/book
3 prop.username=root
4 prop.password=L1060528
5
```

2. 把外部properties属性文件引入到spring配置文件中

引入名称空间context

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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

    <!-- 引入外部文件-->
    <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="${prop.password}">
    </property>
        <property name="password" value="${prop.password}">
    </property>
    </bean>
</beans>

```

IOC操作Bean管理（基于注解方法）

注解概念

1. 代码的特殊标记，格式：@注解名称（属性名称=属性值，属性名称=属性值）
2. 使用注解，注解作用在类上面，方法上面，属性上面
3. 注解目的:简化xml配置

Spring针对Bean管理中创建对象提供注解

1. @Component
2. @Service
3. @Controller
4. @Repository

上面4个注解功能一样，都可以用来创建bean实例

基于注解方法实现创建对象

第一步，引入依赖 `spring-aop-5.2.5.RELEASE.jar`

第二步，开启组件扫描

```

<!--      开启组件扫描-->
<!--      如果扫描多个包，多个包使用逗号隔开-->
<!--      扫描上级包-->
<context:component-scan base-package="com.qiangliu8.demo"></context:component-
scan>

```

第三步，创建类

```

//在注解里面value属性值可以省略不计
//默认值是类名称，首字母小写
@Component(value = "userService") //<bean id="userService" class="..">
public class UserService {
    public void add(){
        System.out.println("service add...");
    }
}

```

第四步，单元测试

```

@Test
public void test1(){
    ApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("bean1.xml");
    UserService userService =
    applicationContext.getBean("userService",UserService.class);
    System.out.println(userService);
    userService.add();
}

```

组件扫描配置

普通设置

```

<context:component-scan base-package="com.qiangliu8.demo"></context:component-
scan>

```

指定设置扫描哪一类注解

```

<!-- use-default-filters="false"表示现在不使用默认filter,自己配置filter-->
<!-- context:include-filter设置扫描哪些内容-->
<!-- type="annotation" 表示按照注解进行扫描
expression="org.springframework.stereotype.Controller"表示扫描Controller注解的类-->
<context:component-scan base-package="com.qiangliu8.demo" use-default-
filters="false" >
    <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

```

指定设置扫描哪一类注解不扫描，其他全扫描

```

<!-- context:exclude-filter表示除了这个注解之外其他都扫描-->
<context:component-scan base-package="com.qiangliu8.demo">
    <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

```

基于注解方式的属性注入

1. @AutoWired：根据属性类型进行自动装配

第一步把service和dao对象创建，在service和dao对象加上注解

```

@Repository
public class userDaoImpl implements userDao{

```

```

@Service
public class UserService {

```

第二步在service注入dao对象，在service类添加dao类型属性，在属性上面使用注解

```

@Service
public class UserService {
    //定义userDao属性 不需要添加set方法
    //添加注入属性注解
    @Autowired
    private com.qiangliu8.demo.dao.userDao userDao;

    public void add(){
        System.out.println("service add...");
        userDao.add();
    }
}

```

第三步单元测试

2. @Qualifier: 根据属性名称进行注入

类型会重复，所以加上value根据名称注入

需要和上面@Autowired一起使用

```

@Repository(value = "userDaoImpl1")
public class userDaoImpl implements userDao{

```

```

@Autowired
@Qualifier(value = "userDaoImpl1")
private com.qiangliu8.demo.dao.userDao userDao;

```

3. @Resource: 可以根据类型注入，可以根据名称注入

```

@Resource()//根据类型注入
//或者
@Resource(value="userDaoImpl1")//根据名称注入

```

4. @Value: 注入普通属性

```

@Value(value = "刘强")
private String name;

```

纯注解开发

1. 创建配置类，代替XML文件

```

@Configuration //作为配置类，代替xml配置文件
@ComponentScan(basePackages = {"com.qiangliu8.demo"})
public class SpringConfig {
    ....
}

```

2. AnnotationConfigApplicationContext注解 单元测试

```

@Test
public void test2(){
    ApplicationContext applicationContext = new
    AnnotationConfigApplicationContext(SpringConfig.class);
    UserService userService =
    applicationContext.getBean("userService",UserService.class);
    userService.add();
}

```

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，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

通俗：不通过修改源代码方式添加新功能，在主干功能里面添加新功能

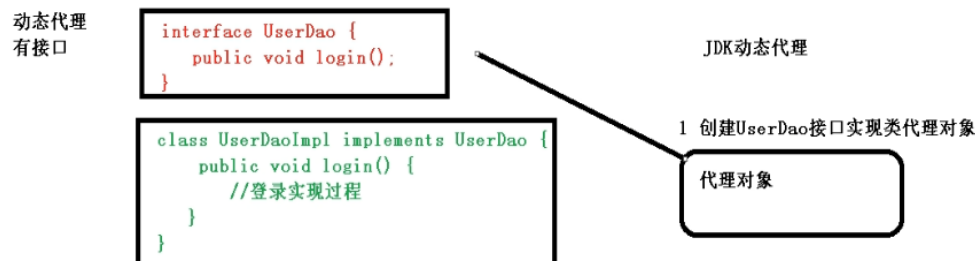
底层原理

AOP底层使用动态代理

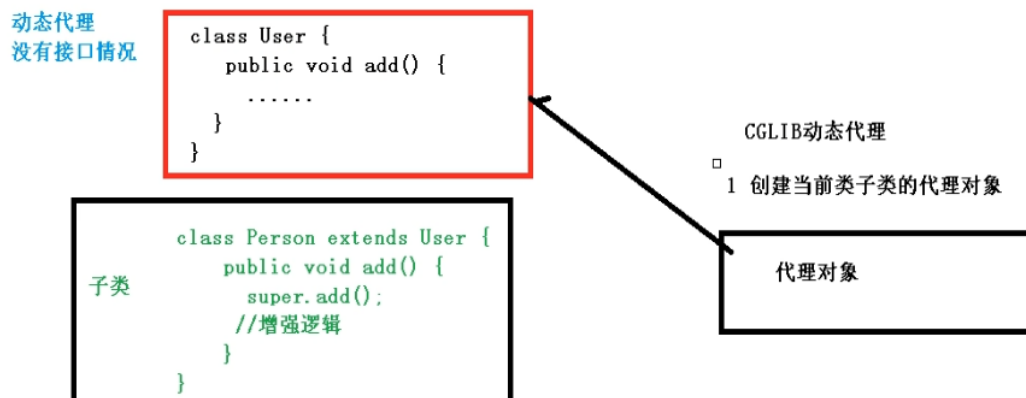
动态代理

有两种情况动态代理

1. 有接口情况，使用JDK动态代理



2. 无接口情况，使用CGLIB动态代理



AOP术语

1. 连接点

类里面哪些方法可以被增强，这些方法成为连接点

2. 切入点

实际被真正增强的方法，称为切入点

3. 通知（增强）

实际增强的逻辑部分被称为通知（增强）

通知有多种类型：前置通知，后置通知，环绕通知，异常通知，最终通知

4. 切面

是动作：把通知应用到切入点的过程

AOP(JDK动态代理)

一、使用JDK动态代理，使用Proxy类里面的方法创建代理对象

调用newProxyInstance方法

```
static Object newProxyInstance(ClassLoader loader, 类<?>[] interfaces, InvocationHandler h)
```

返回指定接口的代理类的实例，该接口将方法调用分派给指定的调用处理程序。

方法里有三个参数：

第一参数：类加载器

第二参数，增强方法所在类，这个类实现的接口，支持多个接口

第三参数，实现这个接口InvocationHandler，创建代理对象，写增强方法。

二、编写JDK动态代理代码

1. 创建接口，定义方法

```
public interface UserDao {  
    public int add(int a,int b);  
    public String update(String id);  
}
```

2. 创建接口实现类，实现方法

```
public class UserDaoImpl implements UserDao{  
    @Override  
    public int add(int a, int b) {  
        return a+b;  
    }  
    @Override  
    public String update(String id) {  
        return id;  
    }  
}
```

3. 使用Proxy类创建接口代理

```
public class JDKProxy {  
    public static void main(String[] args) {  
        //创建接口实现类代理对象  
        Class[] interfaces={UserDao.class};  
  
        UserDao userDao = new UserDaoImpl();  
        UserDao dao = (UserDao)  
        Proxy.newProxyInstance(JDKProxy.class.getClassLoader(), interfaces, new  
        UserDaoProxy(userDao));  
    }  
}
```

```

        System.out.println(dao.add(1,2));
    }
}

//创建代理对象代码
class UserDaoProxy implements InvocationHandler {
    //把创建的是谁的代理对象，把谁传递过来
    //有参构造传递
    private Object object;
    public UserDaoProxy(Object obj){
        this.object = obj;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
        //方法之前
        System.out.println("方法之前执行..." + method.getName() + ": 传递的参数..." + Arrays.toString(args));

        //被增江的方法执行
        Object object1 = method.invoke(object, args);

        if ("add".equals(method.getName())){
            System.out.println("add方法执行了");
        }else{
            System.out.println("update方法执行了");
        }
        //方法之后
        System.out.println("方法之后执行..." + object);
        return object1;
    }
}

```

AOP操作

准备

1. Spring框架一般是基于AspectJ实现AOP操作

AspectJ不是Spring组成部分，独立AOP框架，一般把AspectJ和Spring框架一起使用，进行AOP操作。

2. 基于AspectJ实现AOP操作

1. 基于xml配置文件实现
2. 基于注解方式实现

3. 在项目工程里面引入AOP相关依赖

```

com.springsource.net.sf.cglib-2.2.0.jar
com.springsource.org.aopalliance-1.0.0.jar
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
commons-logging-1.1.1.jar
druid-1.1.9.jar
spring-aop-5.2.6.RELEASE.jar
spring-aspects-5.2.6.RELEASE.jar
spring-beans-5.2.6.RELEASE.jar
spring-context-5.2.6.RELEASE.jar
spring-core-5.2.6.RELEASE.jar
spring-expression-5.2.6.RELEASE.jar

```

4. 切入点表达式

切入点表达式作用：知道对哪个类里面的哪个方法进行增强

语法结构：

```
execution([权限修饰符][返回类型][类全路径][方法名称][参数列表])
```

举例：对com.qiangliu8.demo.dao.UserDao类里面的add进行增强

```
exexction(* com.qiangliu8.demo.dao.UserDao.add(..));
```

对com.qiangliu8.demo.dao.UserDao类所有的方法进行增强

```
execution(* com.qiangliu8.demo.dao.UserDao.*(..));
```

对com.qiangliu8.demo.dao.包里面所有类所有的方法进行增强

```
execution(* com.qiangliu8.demo.*.UserDao.*(..));
```

AspectJ注解

1. 创建类，在类里定义方法

```

//被增强类
public class User {
    //被增强方法
    public void add(){
        System.out.println("User..add.");
    }
}

```

2. 创建增强类（编写增强逻辑）

1. 在增强类里面，创建方法，让不同方法代表不同通知类型

```
//增强类
public class UserProxy {
    //前置通知
    public void before(){
        System.out.println("前置方法。。。");
    }
}
```

3. 进行通知的配置

1. 在Spring配置文件中，开始注解扫描

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"

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

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

    //开启注解扫描
    <context:component-scan base-
package="com.qiangliu8.demo.aopanno">
        </context:component-scan>
</beans>
```

2. 使用注解创建User和UserProxy

```
//被增强类
@Component
public class User {
    //被增强方法
    public void add(){
        System.out.println("User..add.");
    }
}
```

```
//增强类
@Component
public class UserProxy {
    //前置通知
    public void before(){
        System.out.println("前置方法。。。");
    }
}
```

3. 在增强类上添加注解@Aspect


```
//增强类
@Component
@Aspect
public class UserProxy {
    . . . .
}
```

4. 在Spring配置文件中开启生成代理对象

```
//开启Aspect生成代理对象
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

4. 配置不同类型的通知

在增强类里面，在作为通知方法上添加通知类型注解，使用切入点表达式配置。

```
//增强类
@Component
@Aspect
public class UserProxy {
    //前置通知
    //@Before注解表示作为前置通知
    @Before(value="execution(* com.qiangliu8.demo.aopanno.User.add(..))")
    public void before(){
        System.out.println("前置方法。。。");
    }
    //后置通知（返回通知）
    @AfterReturning(value = "execution(* com.qiangliu8.demo.aopanno.*.*(..))")
    public void afterReturn(){
        System.out.println("AfterReturning。。。");
    }
    @After(value = "execution(* com.qiangliu8.demo.aopanno.User.add(..))")
    public void after(){
        System.out.println("后置方法。。。");
    }
    @AfterThrowing(value = "execution(* com.qiangliu8.demo.aopanno.User.add(..))")
    public void afterThrow(){
        System.out.println("AfterThrowing。。。");
    }
    //环绕通知
    @Around(value = "execution(* com.qiangliu8.demo.aopanno.User.add(..))")
    public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        System.out.println("这是环绕之前");
        //被增强方法执行
        proceedingJoinPoint.proceed();
        System.out.println("这是环绕之后");
    }
}
```

5. 单元测试

```
@Test
public void testAopAnno(){
    ApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("bean1.xml");
    User user = applicationContext.getBean("user", User.class);
    user.add();
}
```

无异常发生的打印：

这是环绕之前
前置方法。。。
User..add.
这是环绕之后
后置方法。。。
AfterReturning。。。

异常发生的打印：

这是环绕之前
前置方法。。。
后置方法。。。
AfterThrowing。。。

6. 相同切入点抽取

```
//相同切入点抽取
@Pointcut(value = "execution(* com.qiangliu8.demo.aopanno.User.add(..))")
public void pointcut(){
}

@Before(value = "pointcut()")
public void before(){
    System.out.println("前置方法。。。");
}
```

7. 有多个增强类多同一个方法进行增强，设置增强类优先级

在增强类上面添加注解@Order(数字类型值)，数字类型值越小优先级越高

```
@Component
@Aspect
@Order(2)
public class PersonProxy {
    @Before(value = "execution(*
com.qiangliu8.demo.aopanno.User.add(..))")
    public void before(){
        System.out.println("另一个增强类");
    }
}
```

AspectJ配置文件

1. 创建两个类，一个增强类，一个被增强类，创建方法。
2. 在spring配置文件中创建两个类对象

```
<bean id="book" class="com.qiangliu8.demo.aopxml.Book"></bean>

<bean id="bookProxy" class="com.qiangliu8.demo.aopxml.BookProxy"></bean>
```

3. 在spring配置文件中配置

```
<!--配置aop增强-->
<aop:config>
    <!-- 切入点-->
    <aop:pointcut id="p" expression="execution(*
com.qiangliu8.demo.aopxml.Book.buy(..))"/>
    <!--配置切面-->
    <aop:aspect ref="bookProxy">
        <!--增强作用在具体方法上-->
        <aop:before method="before" pointcut-ref="p"/>
    </aop:aspect>
</aop:config>
```

完全注解

```
@Configuration
@ComponentScan(basePackages = {"com.qiangliu8.demo"})
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class ConfigAop {
}
```

不需要再常见xml了 直接替代以下xml

```
<!--//开启注解扫描-->
<context:component-scan base-package="com.qiangliu8.demo.aopanno">
</context:component-scan>

<!-- //开启Aspect生成代理对象-->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

JdbcTemplate

概念

Spring框架对jdbc进行封装，使用JdbcTemplate方便实现对数据库进行操作。

准备工作

1. 引入jar包

```

com.springsource.net.sf.cglib-2.2.0.jar
com.springsource.org.aopalliance-1.0.0.jar
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
commons-logging-1.1.1.jar
druid-1.1.9.jar
mysql-connector-java-8.0.22.jar
spring-beans-5.2.6.RELEASE.jar
spring-context-5.2.6.RELEASE.jar
spring-core-5.2.6.RELEASE.jar
spring-expression-5.2.6.RELEASE.jar
spring-orm-5.2.6.RELEASE.jar
spring-tx-5.2.6.RELEASE.jar

```

2. spring配置数据库连接池

```

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="url" value="jdbc:mysql:///user_db?
serverTimezone=UTC"></property>
    <property name="username" value="root"></property>
    <property name="password" value="Lq060528"></property>
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver">
</property>
</bean>

```

3. 配置JdbcTemplate对象，注入DataSource

```

<!--JdbcTemplate对象-->
<bean id="JdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
    <!--注入dataSource-->
    <property name="dataSource" ref="dataSource"></property>
</bean>

```

4. 创建service类，创建dao类，在dao注入JdbcTemplate对象

```

@Repository
public class UserDaoImpl implements UserDao{
    @Autowired
    private JdbcTemplate jdbcTemplate;
}

```

```

@Service
public class UserService {
    @Autowired
    private BookDao bookDao;
}

```

JdbcTemplate操作数据库

添加

1. 对用数据库创建实体类

```
public class User {  
    private String userId;  
    private String username;  
    private String ustatus;  
    .....  
}
```

2. 编写service和dao

1. 在dao进行数据库添加操作
2. 调用JdbcTemplate对象里面update方法实现添加操作

```
update(String sql, Object... args)
```

第一个参数: sql语句

第二个参数: 可变参数, sql语句的值

```
@Override  
public void addUser(User user) {  
    //1. 创建sql语句  
    String sql = "insert into t_user (userId, username, ustatus)  
values(?, ?, ?)";  
    //调用方法实现  
    Object [] args = {user.getUserId(), user.getUsername(),  
user.getUstatus()};  
    int update = jdbcTemplate.update(sql, args);  
    System.out.println(update);  
}
```

3. 测试类

```
public class TestUser {  
    @Test  
    public void testJdbcTemplate(){  
        ApplicationContext applicationContext = new  
ClassPathXmlApplicationContext("bean1.xml");  
        UserService userService =  
applicationContext.getBean("userService", UserService.class);  
        User user = new User("1102644615", "刘强", "1");  
        userService.addUser(user);  
    }  
}
```

修改

1. 对用数据库创建实体类

上图不变

2. 编写service和dao

1. 在dao进行数据库添加操作

2. 调用JdbcTemplate对象里面update方法实现添加操作

```
update(String sql, Object... args)
```

第一个参数: sql语句

第二个参数:可变参数, sql语句的值

```
@Override
public void updateUser(User user) {
    String sql = "update t_user set username = ?,ustatus=? where
userId = ?";
    Object[] args =
{user.getUsername(),user.getUstatus(),user.getUserId()};
    int update = jdbcTemplate.update(sql,args);
    System.out.println(update);
}
```

3. 测试类

```
@Test
public void testJdbcTemplate_update(){
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("bean1.xml");
    UserService userService =
applicationContext.getBean("userService",UserService.class);
    User user = new User("1102644615","新刘强","0");
    userService.updateUser(user);
}
```

删除

1. 对用数据库创建实体类

上图不变

2. 编写service和dao

1. 在dao进行数据库添加操作

2. 调用JdbcTemplate对象里面update方法实现添加操作

```
update(String sql, Object... args)
```

第一个参数: sql语句

第二个参数:可变参数, sql语句的值

```
@Override
public void deleteUser(String id) {
    String sql = "delete from t_user where userId = ?";
    Object[] args = {id};
    int update = jdbcTemplate.update(sql,args);
    System.out.println(update);
}
```

3. 测试类

```

@Test
public void testJdbcTemplate_delete(){
    ApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("bean1.xml");
    UserService userService = applicationContext.getBean("userService",
    UserService.class);
    userService.deleteUser("1102644615");
}

```

查询（查询返回某个值）

1. 查询表里面有多少条记录，返回是某个值
2. 调用JdbcTemplate对象实现查询返回某个值代码

```
queryForObject(String sql, Class<T> requiredType)
```

第一个参数: sql语句

第二个参数:返回值类型Class

```

@Override
public int selectCount() {
    String sql = "select count(*) from t_user";
    int count = jdbcTemplate.queryForObject(sql,Integer.class);
    return count;
}

```

查询（查询返回某个对象）

1. 查询表里面某个具体的记录
2. JdbcTemplate实现查询返回对象

![image-20210202184508722](C:\Users\11026\AppData\Roaming\Typora\typora-user-images\image-20210202184508722.png)

第一个参数: sql语句

第二个参数:RowMapper，是接口，返回不同类型数据，使用这个接口里面实现类

第三个参数, sql语句参数

```

```java
@Override
public User findUserInfo(String id) {
 String sql = "select * from t_user where UserId = ?";
 User user = jdbcTemplate.queryForObject(sql,new
 BeanPropertyRowMapper<User>(User.class),id);
 return user;
}

```

## 查询 (查询返回集合)

1. 查询表里面某些记录的集合
2. JdbcTemplate实现查询返回对象

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

第一个参数: sql语句

第二个参数: RowMapper, 是接口, 返回不同类型数据, 使用这个接口里面实现类

第三个参数, sql语句参数

```
@Override
public User findUserInfo(String id) {
 String sql = "select * from t_user where UserId = ?";
 User user = jdbcTemplate.queryForObject(sql, new
 BeanPropertyRowMapper<User>(User.class), id);
 return user;
}
```

## 批量添加

1. 操作表里面多条记录
2. JdbcTemplate实现批量添加操作

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

第一个参数: sql语句

第二个参数: List集合, 添加多条记录数据

```
@Override
public void batchAdd(List<Object[]> batchArgs) {
 String sql = "insert into t_user (userId, username, ustatus)
 values(?, ?, ?)";
 int[] ints = jdbcTemplate.batchUpdate(sql, batchArgs);
 System.out.println(Arrays.toString(ints));
}
```

3. 单元测试

```
@Test
public void testJdbcTemplate_batchAdd() {
 ApplicationContext applicationContext = new
 ClassPathXmlApplicationContext("bean1.xml");
 UserService userService = applicationContext.getBean("userService",
 UserService.class);
 List<Object[]> list = new ArrayList<Object[]>();
 list.add(new String[]{"1", "新刘强1", "1"});
 list.add(new String[]{"2", "新刘强2", "1"});
 list.add(new String[]{"3", "新刘强3", "0"});
 userService.batchAdd(list);
}
```



## 批量修改

1. 操作表里面多条记录
2. JdbcTemplate实现批量添加操作

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

第一个参数: sql语句

第二个参数: List集合, 添加多条记录数据

```
@Override
public void batchAdd(List<Object[]> batchArgs) {
 String sql = "insert into t_user (userId,username,ustatus)
values(?,?,?)";
 int[] ints = jdbcTemplate.batchUpdate(sql,batchArgs);
 System.out.println(Arrays.toString(ints));
}
```

3. 单元测试

```
@Test
public void testJdbcTemplate_batchUpdate(){
 ApplicationContext applicationContext = new
 ClassPathXmlApplicationContext("bean1.xml");
 UserService userService = applicationContext.getBean("userService",
 UserService.class);
 List<Object[]> list = new ArrayList<Object[]>();
 list.add(new String[]{"1新刘强", "11", "1"});
 list.add(new String[]{"2新刘强", "22", "2"});
 list.add(new String[]{"3新刘强", "33", "3"});
 userService.batchUpdate(list);
}
```

## 事务

事务时数据库操作最基本单元, 逻辑上一组操作, 要么成功, 要么都失败。

**四大特性:**

1. 原子性
2. 一致性
3. 隔离性
4. 持久性

## 事务操作 (搭建事务操作环境)

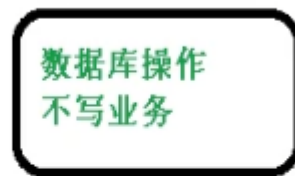
## Service



创建转账的方法

(1) 调用dao两个的方法

## Dao



创建两个方法

(1) 少钱的方法

(2) 多钱的方法

第一步：创建数据库表，添加记录。

	id	username	money
	1	刘强	1000
▶	2	俞文竹	1000

第二步：创建service,搭建dao，完成对象创建和注入关系。

1. service注入dao,在dao注入JdbcTemplate,在JdbcTemplate注入DataSource.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

 <context:component-scan base-package="com.qiangliu8.shiwu">
</context:component-scan>

 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
 <property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"></property>
 <property name="url" value="jdbc:mysql:///user_db?
serverTimezone=UTC"></property>
 <property name="username" value="root"></property>
 <property name="password" value="Lq060528"></property>
 </bean>

 <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
 <property name="dataSource" ref="dataSource"></property>
 </bean>
</beans>
```

```
@Repository
public class UserDaoImpl implements UserDao{
 @Autowired
 private JdbcTemplate jdbcTemplate;
}
```

```
@Service
public class UserService {
 @Autowired
 private UserDao userDao;
}
```

2. dao中创建少钱和多钱的两个方法，service创建转账方法。

**dao:**

```
@Repository
public class UserDaoImpl implements UserDao{
 @Autowired
 private JdbcTemplate jdbcTemplate;

 @Override
 public void addMoney(int money,String id) {
 String sql = "update t_account set money = money-? where id = ?";
 jdbcTemplate.update(sql,money,id);
 }

 @Override
 public void reduceMoney(int money,String id) {
 String sql = "update t_account set money = money+? where id = ?";
 jdbcTemplate.update(sql,money,id);
 }
}
```

**service:**

```
@Service
public class UserService {
 @Autowired
 private UserDao userDao;
 public void accountMoney(String id1,String id2,int money){
 //id号加money
 userDao.addMoney(money,id1);
 //id号减money
 userDao.reduceMoney(money,id2);
 }
}
```

3. 单元测试

```

@Test
public void testAccount(){
 ApplicationContext applicationContext = new
 ClassPathXmlApplicationContext("bean2.xml");
 UserService userService = applicationContext.getBean("userService",
 UserService.class);
 userService.accountMoney("1", "2", 100);
}

```

4. 如果上面代码出现了问题，需要用事务去解决。

1. 第一步：开启事务
2. 第二步：进行业务操作
3. 第三步：没有发生异常，提交事务
4. 第四步：出现异常，事务回滚

## 事务操作

### Spring 事务管理介绍

1. 事务一般添加到JavaEE三层结构里面Service层（业务逻辑层）
2. 在spring进行事务管理操作

有两种方式：**编程式事务管理**和**声明式事务管理**（建议使用）

3. 声明式事务管理

1. **基于注解方式**
2. **基于xml配置文件方式**

4. 在Spring进行声明式事务管理，**底层使用AOP**
5. Spring事务管理API

1. 提供一个接口，代表事务管理器，这个接口针对不同的框架提供不同的实现类、

```

PlatformLoggingMBean

java.lang.management
public interface PlatformLoggingMBean
extends management.PlatformManagedObject

The management interface for the logging facility.
There is a single global instance of the PlatformLoggingMBean. The ManagementFactory.getPlatformLoggingMBean()
can be used to obtain the PlatformLoggingMBean object as follows:

 PlatformLoggingMBean logging = ManagementFactory.getPlatformMBean(PlatformLoggingMBean.class);

The PlatformLoggingMBean object is also registered with the platform MBeanServer. The ObjectName
identifying the PlatformLoggingMBean within an MBeanServer is:

 java.util.logging:type=Logging

Since: 1.7

```

### 事务操作（注解声明式事务管理）

1. 在Spring配置文件配置事务管理器

```

<!--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. 在Spring配置文件中，开启事务注解

### 1. 在spring配置文件引入名称空间tx

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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

```

### 2. 开启事务注解

```

<!-- 开启事务注解-->
<tx:annotation-driven transaction-
manager="transactionManager"></tx:annotation-driven>

```

### 3. 在Service类上面添加事务注解

@Transactional可以类上面，表明这个类所有方法都添加了事务  
添加方法上，表明这个方法添加了事务。

```

@Service
@Transactional
public class UserService {
 . . .
}

```

## 事务操作（声明式事务管理参数配置）

在Service类上面添加@Transactional，在这个注解里面可以配置事务相关参数

```

Propagation propagation() default org.springframework.transaction.annotation
Isolation isolation() default org.springframework.transaction.annotation.Isolatic
in timeout() default -1
boolean readOnly() default false
Class<? extends Throwable> rollbackFor() default {}
String[] rollbackForClassName() default {}
Class<? extends Throwable> noRollbackFor() default {}
String[] noRollbackForClassName() default {}

```

**propagation:**事务传播行为

多事务方法直接进行调用，这个过程中事务是如何进行管理的

事务方法：对数据库表数据进行变化的操作。

事务方法：对数据库表数据进行变化的操作

```

@Transactional
public void add() {
 //调用update方法
 update();
}

```

```

public void update() {

}

```

Spring框架事务传播行为有7种

**REQUIRED** 如果add方法本身有事务，调用update方法之后，update使用当前add方法里面事务  
如果add方法本身没有事务，调用update方法之后，创建新事务

**REQUIRED\_NEW** 使用add方法调用update方法，如果add无论是否有事务，都创建新的事务

事务的传播行为可以由传播属性指定。Spring 定义了 7 种类传播行为。

传播属性	描述
REQUIRED	如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在它自己的事务内运行
REQUIRED_NEW	当前的方法必须启动新事务，并在它自己的事务内运行。如果有事务正在运行，应该将它挂起
SUPPORTS	如果有事务在运行，当前的方法就在这个事务内运行。否则它可以不运行在事务中。
NOT_SUPPORTED	当前的方法不应该运行在事务中。如果有运行的事务，将它挂起
MANDATORY	当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常
NEVER	当前的方法不应该运行在事务中。如果有运行的事务，就抛出异常
NESTED	如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行。否则，就启动一个新的事务，并在它自己的事务内运行。

```
8 @Service
9 @Transactional(propagation = Pr
0 public class UserService {
1 @Autowired
2 private UserDao userDao;
3 public void accountMo
4 userDao.addMoney(
5 userDao.reduceMon
6 }
7 }
8
```

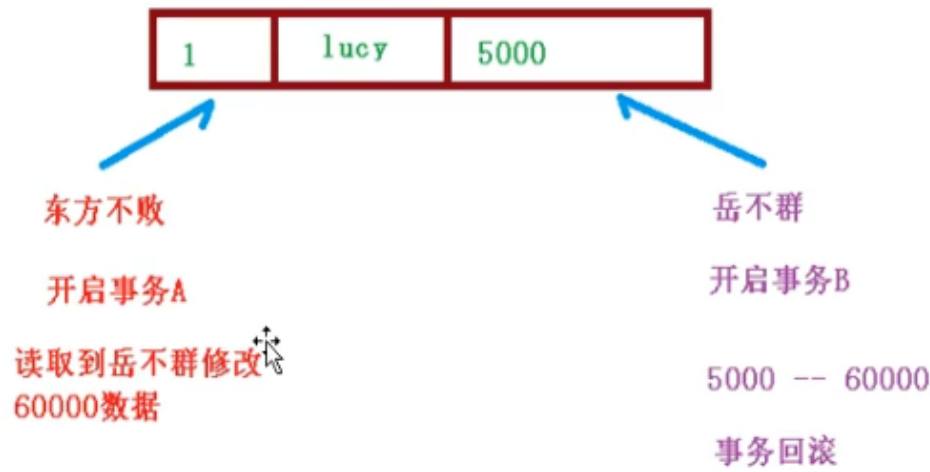
Propagation.MANDATORY (org.springframework.transaction. annotation) Propagatio  
Propagation.NESTED (org.springframework.transaction. annotation)  
Propagation.NEVER (org.springframework.transaction. annotation) Propagatio  
Propagation.REQUIRED (org.springframework.transaction. annotation) Propagatio  
Propagation.SUPPORTS (org.springframework.transaction. annotation) Propagatio  
Propagation org.springframework.transaction. annotation  
Propagation.NOT\_SUPPORTED (org.springframework.transaction. annotation) Propagatio  
Propagation.REQUIRES\_NEW (org.springframework.transaction. annotation) Propagatio  
Process java.lang  
ProcessBuilder java.lang  
ProcessHandle java.lang  
ReportedException org.springframework.beans  
Press Ctrl+ to choose the selected (or first) suggestion and insert a dot afterwards. Next Tip

**isolation:**事务隔离级别

事务由特性成为隔离性，多事物操作之间不会产生影响，不考虑隔离性会产生三个问题：脏读，不可重复读，虚（幻）读。

**脏读:**1个未提交事务读取到了另一个未提交的事务的数据。即读取了还没回滚的无效脏数据的操作。

脏读



不可重复读：一个未提交事务读取到另一个提交事务修改数据

不可重复读



虚读：一个未提交的事务读取到了另一个已提交事务增加的数据。

通过设置事务隔离性，可以解决读问题。

	脏读	不可重复读	幻读
READ UNCOMMITTED (读未提交)	有	有	有
READ COMMITTED (读已提交)	无	有	有
REPEATABLE READ (可重复读)	无	无	有
SERIALIZABLE (串行化)	无	无	无

```
@Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.REPEATABLE_READ)
```

timeout:超时时间



事务需要在一定时间内提交，如果不提交进行回滚

默认值是-1，设置时间以秒为单位。

**readOnly:**是否只读

读：查询操作，写：添加修改删除操作

默认值为false,表示可以查询，可以添加删除操作。true表示只能查询

**rollbackFor:**回滚

设置查询哪些异常进行事务回滚

**noRollbackFor:** 不回滚

设置出现哪些异常不进行事务回滚

## 事务操作（XML声明式事务管理）

### 1. 在spring配置文件中进行配置

第一步配置事务管理器

```
<!--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>
```

第二步配置通知

```
<!-- 配置通知-->
<tx:advice id="txadvice">
 <!-- 配置事务参数-->
 <tx:attributes>
 <!-- 指定那种规则的商法上面添加事务-->
 <tx:method name="accountMoney" isolation="REPEATABLE_READ"/>
 <tx:method name="account*" propagation="REQUIRED"/>
 </tx:attributes>
</tx:advice>
```

第三步配置切入点和切面

```
<!-- 配置切入点和切面-->
<aop:config>
 <!-- 配置切入点-->
 <aop:pointcut id="pt" expression="execution(*
com.qiangliu8.shiwu.service.UserService.*(..))"/>
 <!-- 配置切面-->
 <aop:advisor advice-ref="txadvice" pointcut-ref="pt"></aop:advisor>
</aop:config>
```

## 事务操作（完全注解方式）

```
@Configuration//配置类
@ComponentScan(basePackages = "com.qiangliu8.shiwu")//开启组件扫描
@EnableTransactionManagement//开启事务
public class TxConfig {
 //创建数据库连接池
 @Bean
 public DruidDataSource getDruidDataSource(){
 DruidDataSource druidDataSource = new DruidDataSource();
 druidDataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
 druidDataSource.setUrl("jdbc:mysql:///user_db?serverTimezone=UTC");
 druidDataSource.setUsername("root");
 druidDataSource.setPassword("Lq060528");
 return druidDataSource;
 }
 //创建JdbcTemplate对象
 @Bean
 public JdbcTemplate getJdbcTemplate(DataSource dataSource){
 //到ioc容器中根据类型找到dataSource
 JdbcTemplate jdbcTemplate = new JdbcTemplate();
 jdbcTemplate.setDataSource(dataSource);
 return jdbcTemplate;
 }
 //创建事务管理器对象
 @Bean
 public DataSourceTransactionManager
 getDataSourceTransactionManager(DataSource dataSource){
 DataSourceTransactionManager dataSourceTransactionManager = new
 DataSourceTransactionManager();
 dataSourceTransactionManager.setDataSource(dataSource);
 return dataSourceTransactionManager;
 }
}
```