**Spring5框架**

课程内容介绍 Introduction to course content

|  |
| --- |
| 第一章 Spring框架概述  **Chapter One Spring framework overview**  第二章 IOC容器  **Chapter Two IOC container**  第三章 AOP（Aspect-Oriented Programming）  **Chapter Three AOP（Aspect-Oriented Programming）**  第四章 JdbcTemplate  **Chapter Four JdbcTemplate**  第五章 事务管理  **Chapter Five transaction management** |

Spring框架概述（Spring framework overview）

|  |
| --- |
| 1. Sping是轻量级的开源JavaEE框架，解决企业应用开发的复杂性。   Spring is a lightweight open source Java EE framework that solves the complexity of enterprise application development.   1. Spring有两个核心部分：IOC和AOP   Spring has two core parts: IOC and AOP.   1. IOC：控制反转（Inversion of Control），把创建对象过程交给spring进行管理。   **IOC means Inversion of Control, which hands the process of creating objects to spring for management.**   1. AOP：面向切面编程（Aspect-Oriented Programming），不修改源代码进行功能增强   **AOP means Aspect-Oriented Programming, which enhances functions without modifying the source code**   1. Spring特点（Spring features） 2. 方便解耦，简化开发 **Convenient decoupling and simplified development** 3. Aop编程支持 AOP programming support 4. 方便程序的测试 Convenient program testing 5. 方便集成各种优秀框架 Convenient integration of various excellent frameworks 6. 方便进行事务操作 Facilitate transaction operations 7. 降低API开发难度 Reduce the difficulty of API development 8. 现在课程中，选取Spring版本5.x   Now in the course, we select spring version 5. X |

Spring5入门案例（Spring5 starter case）

|  |
| --- |
| 1、download Spring5 URL：spring.io   1. Use the latest stable version of spring 5.3.21       选择GA（稳定）版本。GA 是 General Availability 的缩写，翻译过来代表稳定可用于生产的版本。  We select GA (stable) version. GA is the abbreviation of general availability, which translates to a stable version available for production.   1. Download address:   https://github.com/spring-projects/spring-framework         1. 可以通过maven中央仓库下载。<https://mvnrepository.com/>   You can download through Maven central warehouse    2、打开idea工具，创建普通Java工程。  Open the idea tool to create a common java project        3、导入Spring5相关jar包  **Import spring 5 related jar package**  Spring5模块      4、创建普通类，在这个类创建普通方法  **Create a common class and create a common method in this class**  public class User {  public void add() {  System.out.println("add...");  }  }  5、创建Spring配置文件，在配置文件配置创建的对象  **Create a spring configuration file and configure the created object in the configuration file**   1. Spring configuration file bean1.xml     <?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对象 configure User object-->  <bean id="user" class="com.qixin.User"></bean>  </beans>  6、Testing  public class TestSpring5 {  @Test  public void testAdd() {  //1、Load spring configuration file  ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");  //2、Get the object created by the configuration  User user = context.getBean("user", User.class);  System.out.println(user);  user.add();  }  } |

IOC（概念和原理 Concept and principle）

|  |
| --- |
| When we were learning Spring, we may have heard two words（DI and IOC）.DI means Dependency injection, and IOC means Inversion of Control  DI与IOC含义相同，只不过这两个称呼是从两个角度描述同一个概念。下面我们通过简单的语言来描述这两个概念。  DI and IOC have the same meaning, but these two names describe the same concept from two perspectives. Below we describe these two concepts in simple language.  当某个Java对象（调用者）需要调用另一个Java对象（被调用者，即被依赖对象）时，在传统模式下，调用者通常会采用“new 被调用者”的代码方式来创建对象，如下图所示。这种方式会导致调用者与被调用者之间的耦合性增加，不利于后期项目的升级和维护。  When a Java object (the caller) needs to call another Java object (the callee, that is the dependent object), in the traditional mode, the caller usually uses the "new callee" code method to create the object, as shown in the following figure. This method will increase the coupling between the caller and the callee, which is not conducive to the upgrade and maintenance of the later project.    在使用Spring框架之后，对象的实例不再由调用者来创建，而是由Spring容器来创建。Spring容器会负责控制程序之间的关系，而不是由调用者的程序代码直接控制。这样，控制权由应用代码转移到了Spring容器，控制权发生了反转，这就是Spring的控制反转（IOC）。  After using the Spring framework, the instance of the object is no longer created by the caller, but by the Spring container. The Spring container is responsible for controlling the relationship between programs, rather than being directly controlled by the caller's program code. In this way, the control is transferred from the application code to the Spring container, and the control is reversed, which is Spring's Inversion of Control (IOC).  从Spring容器的角度来看，Spring容器负责将被依赖对象赋值给调用者的成员变量，这相当于为调用者注入了它依赖的实例，这就是Spring的依赖注入（DI），如下图所示。  From the perspective of the Spring container, the Spring container is responsible for assigning the dependent object to the caller's member variable, which is equivalent to injecting the caller's dependent instance, which is Spring's dependency injection (DI), as shown in the following figure .     1. 1、什么是IOC What is IOC 2. 控制反转，把对象创建和对象之间的调用过程， 交给Spring进行管理   Inversion of Control , Spring manages object creation and calls between objects   1. 使用IOC目的：为了耦合度降低   The purpose of using IOC is to reduce the coupling.   1. IOC底层原理（IOC underlying principle）   xml解析、工厂模式、反射  Spring using XML parsing, factory pattern and reflection to implement IOC   1. 讲解IOC底层原理（Explain the basic principle of IOC）   class UserService {  execute() {  UserDao dao = new UserDao();  dao.add();  }  }  Class UserDao {  add() {  …  }  }  传统方式：耦合度太高了（Traditional mode: coupling is too high）  编程倡导的是高内聚、低耦合（Programming emphasizes high cohesion and low coupling）  工厂模式（Factory Pattern）  class UserService {  execute() {  UserDao dao = UserFactory.getUserDao();  dao.add();  }  }  class UserDao {  add() {  …  }  }  class UserFactory {  public static UserDao getUserDao() {  return new UserDao();  }  }  目的：通过工厂模式降低耦合程度。这并不是最终方案。  Objective: to reduce the coupling degree through factory pattern. But this is not the final plan.   1. IOC实现过程（IOC implementation process）   第一步：xml配置文件，配置创建的对象  Step 1: XML configuration file to configure the created object  <bean id="dao" class="com.qixin.UserDao"></bean>  第二步：创建工厂类  Step 2: create factory class  We only talk about ideas, not concrete implementation.  Class UserFactory {  public static UserDao getDao() {  String classValue = class属性值; // 1.xml parsing  Class clazz = Class.forName(classValue); //2. using reflection technology to create objects  return (UserDao)clazz.newInstance();  }  }  进一步降低耦合度。只需要更改xml配置文件中class属性的值即可。  Further reducing the coupling. You only need to change the value of the class attribute in the XML configuration file. |

IOC接口（IOC interface）

|  |
| --- |
| 1. **IOC思想基于IOC容器完成，IOC容器底层就是对象工厂**   **The idea of IOC is based on the IOC container, and the bottom layer of the IOC container is the object factory**   1. **Spring提供IOC容器实现两种方式：（两个接口）**   **Spring provides two ways to implement the IOC container: (through two interfaces)**     1. BeanFactory：IOC容器基本实现，是Spring内部使用的接口，不提供开发人员进行使用   BeanFactory is the basic implementation of the IOC container. It is an interface used internally by Spring and is not provided for developers to use.   * 加载配置文件的时候不会创建对象，在获取对象（使用）时才去创建对象   Objects are not created when the configuration file is loaded, and objects are created when the object is obtained.  @Test  public void testAdd() {  //1、loading spring configuration file  BeanFactory context = new ClassPathXmlApplicationContext("bean1.xml");  //2、get the object created by the configuration  User user = context.getBean("user", User.class);  System.out.println(user);  user.add();  }   1. ApplicationContext：BeanFactory接口的子接口，提供更多更强大的功能，一般由开发人员进行使用   ApplicationContext is a sub-interface of the BeanFactory interface, which provides more and more powerful functions and is generally used by developers   * 加载配置文件的时候就会把配置文件中的对象进行创建   When the configuration file is loaded, the objects defined in the configuration file are created  @Test  public void testAdd() {  //1、loading spring configuration file  ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");  //2、get the object created by the configuration  User user = context.getBean("user", User.class);  System.out.println(user);  user.add();  }   1. **ApplicationContext接口有两个实现类（Ctrl+H）**   **The ApplicationContext interface has two implementation classes**     * FileSystemXmlApplicationContext   ApplicationContext context = new FileSystemXmlApplicationContext("D:\\代码工厂\\spring5\\spring5\_demo1\\src\\bean1.xml");   * ClassPathXmlApplicationContext   ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml"); |

IOC操作中的Bean管理（Bean management in IOC operations）

|  |
| --- |
| 1. **什么是Bean管理（概念） What’s the bean management?**   Bean管理包含两个操作：  Bean management consists of two operations   1. Spring创建对象 Spring creates objects 2. Spring注入属性 Spring injection properties 3. **Bean管理操作有两种方式 There are two ways for Bean management operations** 4. 基于xml配置文件方式实现 Implementation based on xml configuration file 5. 基于注解方式实现 Implementation based on annotations |

IOC操作中的Bean管理（基于xml方式）Bean management in IOC operations（Base on xml configuration file）

|  |
| --- |
| **1、基于xml方式创建对象 Create objects based on xml**  <bean id="user" class="com.qixin.User"></bean>   1. 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   In the spring configuration file, use the bean tag and add the corresponding attribute in the tag to realize object creation   1. 在bean标签里有很多属性，下面是几个常用的属性   There are many attributes in the bean tag, the following are several commonly used attributes   * id属性：唯一标识   id attribute: unique identification   * class属性：类全路径   class attribute: class full path（The path containing packages and classes）   1. 创建对象的时候，默认是使用无参构造方法完成对象创建   When creating an object, the default is to use the no-argument constructor to complete the object creation  **2、基于xml方式注入属性 Inject attributes based on xml**   1. DI：依赖注入，就是注入属性。在创建对象基础之上完成的。   DI: Dependency Injection, is to inject properties. DI is done on the basis of creating objects  传统方式 Traditional way  第一种方式：使用set方法进行注入  The first way: use the set method for injection  **public class** Book {  **private** String **name**;  **public void** setName(String name) {  **this**.**name** = name;  }  **public static void** main(String[] args) {  Book book = **new** Book();  book.setName(**"Java Programing"**);  } }  第二种方式：使用有参构造方法进行注入  The second way: use the parameterized construction method for injection  **public class** Book {  **private** String **name**;  **public void** setName(String name) {  **this**.**name** = name;  }  **public** Book(String name) {  **this**.**name** = name;  }  **public static void** main(String[] args) {  Book book = **new** Book(**"Java Programing"**);  } }   1. **第一种注入方式：使用set方式进行注入**   The first injection method: use the set method for injection   1. 创建类，定义属性和对应的set方法   Create a class, define properties and corresponding set methods  **public class** Book {  **private** String **name**;  **private** String **author**;  **private** String **address**;   **public void** setName(String name) {  **this**.**name** = name;  }  **public void** setAuthor(String author) {  **this**.**author** = author;  }  **public void** setAddress(String address) {  **this**.**address** = address;  }  **public void** output() {  System.***out***.println(**name** + **":"** + **author** + **":"** + **address**);  } }   1. 在spring配置文件配置对象创建，配置属性注入   Configure object creation and property injection in the spring configuration file  <**bean id="book" class="com.qixin.Book"**>  <**property name="name" value="Java programming"**/>  <**property name="author" value="qixin"**/>  <**property name="address" value="湖北"**/> </**bean**>   1. 测试 testing   @Test **public void** testBook() {ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);Book book = context.getBean(**"book"**, Book.**class**);  book.output(); }   1. **第二种注入方式：使用有参构造进行注入**   The second injection method: use parameterized construction for injection   1. 创建类，定义属性，创建属性所对应的有参构造方法   Create classes, define properties, and create parameterized constructors corresponding to properties  **public class** Order {  **private** String **name**;  **private** String **address**;  **public** Order(String name, String address) {  **this**.**name** = name;  **this**.**address** = address;  }  **public void** output() {  System.***out***.println(**name** + **":"** + **address**);  } }  （2）在spring配置文件中进行配置 Config in spring configuration file  *<!--* *Injecting properties through parameterized constructors -->* <**bean id="order" class="com.qixin.Order"**>  <**constructor-arg name="name" value="computer"**/>  <**constructor-arg name="address" value="huat"**/> </**bean**>  （3）Testing  @Test **public void** testOrder() {ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);Order order = context.getBean(**"order"**, Order.**class**);  order.output(); }  **5、p名称空间注入（了解） p namespace injection (understand)**  使用p名称空间注入，可以简化基于xml配置方式  Using p namespace injection, xml-based configuration can be simplified  第一步 在配置文件中添加p名称空间  The first step is to add the p namespace to the configuration file    第二步 进行属性注入，在bean标签里面进行操作  The second step is to perform attribute injection and set the attribute value in the bean tag  <**bean id="book" class="com.qixin.Book" p:name="Java" p:author="qixin" p:address="hubei"**> </**bean**> |

IOC操作中的Bean管理（注入其他类型属性） Bean management in IOC operations（Inject other type properties）

|  |
| --- |
| 1. 1、字面量 literal 2. null value   <bean id="book" class="com.qixin.Book">  <property name="name" value="Java programming"/>  <property name="author" value="qixin"/>  <!--null-->  <property name="address">  <null/>  </property>  </bean>   1. 属性值包含特殊符号 Property value contains special symbols   <**bean id="book" class="com.qixin.Book"**>  <**property name="name" value="Java programming"**/>  <**property name="author" value="qixin"**/>  *<!--* *Property value contains special symbols <>  1 escape the symbol <> to &lt; &gt;  2 Write the contents with special symbols to CDATA  -->* <**property name="address"**>  <**value**><![CDATA[<hubei>]]></**value**>  </**property**> </**bean**>   1. 2、注入属性-外部bean Inject properties - external beans 2. 创建两个类service类和dao类 create service and dao class   public interface UserDao {  public void update();  }  public class UserDaoImpl implements UserDao{  @Override  public void update() {  System.out.println("dao update ......");  }  }  public class UserService {  // Create UserDao property and generate set method  private UserDao userDao;  public void setUserDao(UserDao userDao) {  this.userDao = userDao;  }  public void add() {  System.out.println("service add ......");  userDao.update();  }  }   1. 在service调用dao里面的方法 Call the method of dao in the service 2. 在spring配置文件中进行配置 Configure in spring configuration file bean2.xml   <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  <!--1 create service and dao objects-->  <bean id="userService" class="com.qixin.service.UserService">  <!—injecti userDao object  name property：the property name in class  ref property：Refers to the id value of the userDao bean tag  -->  <property name="userDao" ref="userDao"></property>  </bean>  <bean id="userDao" class="com.qixin.dao.UserDaoImpl"></bean>  </beans>   1. Testing   public class TestBean {  @Test  public void testAdd() {  ApplicationContext context = new ClassPathXmlApplicationContext("bean2.xml");  UserService userService = context.getBean("userService", UserService.class);  userService.add();  }  }  3、注入属性-内部bean Inject properties - inner beans  （1）一对多关系：部门和员工 One-to-many relationship: department and employee  A department has multiple employees, and an employee belongs to a department  （2）在实体类之间表示一对多关系 Represent a one-to-many relationship in the entity class  //department class  public class Dept {  private String name;  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "Dept{" +  "name='" + name + '\'' +  '}';  }  }  //employee class  public class Emp {  private String name;  private String gender;  //员工属于某一个部门，使用对象形式表示  //The employee belongs to a certain department, which is represented in the form of an object  private Dept dept;  public void setName(String name) {  this.name = name;  }  public void setGender(String gender) {  this.gender = gender;  }  public void setDept(Dept dept) {  this.dept = dept;  }  public void output() {  System.out.println(name + ":" + gender + ":" + dept);  }  }  （3）在spring配置文件中进行配置 Configure in spring configuration file( bean3.xml)  <bean id="emp" class="com.qixin.bean.Emp">  <property name="name" value="tom"/>  <property name="gender" value="male"/>  <!-- set object properties -->  <property name="dept">  <bean id="dept" class="com.qixin.bean.Dept">  <property name="name" value=" Development Department "></property>  </bean>  </property>  </bean>  （4）Testing  @Test  public void testEmp() {  ApplicationContext context = new ClassPathXmlApplicationContext("bean3.xml");  Emp emp = context.getBean("emp", Emp.class);  emp.output();  }  4、注入属性-级联赋值 Injecting properties - cascading assignment  （1）the first way  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean id="emp" class="com.qixin.bean.Emp">  <property name="name" value="tom"/>  <property name="gender" value="male"/>  <!-- Cascading assignment -->  **<property name="dept" ref="dept"/>**  </bean>  <bean id="dept" class="com.qixin.bean.Dept">  <property name="name" value=" Development Department "/>  </bean>  </beans>  （2）the second way  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean id="emp" class="com.qixin.bean.Emp">  <property name="name" value="tom"/>  <property name="gender" value="male"/>  <!--级联赋值-->  <property name="dept" ref="dept"/>  **<property name="dept.name" value="** **Development Department "/>**  </bean>  <bean id="dept" class="com.qixin.bean.Dept"></bean>  </beans>  注意：需要在Emp类中生成dept的get方法  Note: The get method of dept needs to be generated in the Emp class  public class Emp {  private String name;  private String gender;  private Dept dept;  //生成dept的get方法  **public Dept getDept() {**  **return dept;**  **}**  **……**  } |

IOC操作中的Bean管理（注入集合属性）Bean management in IOC operations（Inject collection properties）

|  |
| --- |
| 复制并创建新的项目spring5\_demo2  Copy and create new project spring5\_demo2  1、注入数组类型属性 Inject array type properties  **2、注入List集合类型属性 Inject list collection properties**  **3、注入Map集合类型属性 Inject map collection type properties**  （1）创建类，定义数组、List、Map、Set  Create class, define array、List、Map、Set  public class Student {  //1 array type property  private String[] courses;  //2 list collection type property  private List<String> list;  //3 map collection type property  private Map<String, String> map;  //4 set collection type property  private Set<String> sets;    public void setCourses(String[] courses) {  this.courses = courses;  }  public void setList(List<String> list) {  this.list = list;  }  public void setMap(Map<String, String> map) {  this.map = map;  }  public void setSets(Set<String> sets) {  this.sets = sets;  }  public void output() {  System.out.println(Arrays.toString(courses));  System.out.println(list);  System.out.println(map);  System.out.println(sets);  }  }  （2）在spring配置文件中进行配置 config in spring configuration file bean1.xml  <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">  <!--1 collection type inject-->  <bean id="student" class="com.qixin.collection.Student">  <!—inject array type property数组类型属性注入-->  <property name="courses">  <array>  <value>Java</value>  <value>Database</value>  </array>  </property>  <!--inject list type property-->  <property name="list">  <list>  <value>张三</value>  <value>小三</value>  </list>  </property>  <!—inject map type property-->  <property name="map">  <map>  <entry key="Java" value="java"></entry>  <entry key="Database" value="database"></entry>  </map>  </property>  <!—inject set type property-->  <property name="sets">  <set>  <value>MySQL</value>  <value>Redis</value>  </set>  </property>  </bean>  </beans>  （3）Testing  public class TestStudent {  @Test  public void testStudent() {  ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");  Student student = context.getBean("student", Student.class);  student.output();  }  }  再看其他特殊情况 Let’s see other situation.  **4、在集合里面设置对象类型值 Set the object type value in the collection**  **一个学生可以上很多门课 A student can take many courses**  （1）创建Course类 create Course class  public class Course {  private String name; // course name  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "Course{" +  "name='" + name + '\'' +  '}';  }  }  （2）在Student类中添加集合属性 Add collection type property in Student class  // student selected course  private List<Course> courseList;  public void setCourseList(List<Course> courseList) {  this.courseList = courseList;  }  （3）在配置文件中创建多个course对象 Create multiple course objects in the configuration file  *<!— create mutiple course objects -->* <**bean id="course1" class="com.qixin.collection.Course"**>  <**property name="name" value="Java"**/> </**bean**> <**bean id="course2" class="com.qixin.collection.Course"**>  <**property name="name" value="Spring"**/> </**bean**>  *<!— inject list collection type -->* <**property name="courseList"**>  <**list**>  <**ref bean="course1"**></**ref**>  <**ref bean="course2"**></**ref**>  </**list**> </**property**>  （4）Testing  **public void** output() {  System.***out***.println(Arrays.*toString*(**courses**));  System.***out***.println(**list**);  System.***out***.println(**map**);  System.***out***.println(**sets**);  System.***out***.println(**courseList**);  }  **5、把集合注入部分提取出来 Extract the collection injection part**  （1）创建新的Book类 create a new class named Book  **public class** Book {  **private** List<String> **list**;  **public void** setList(List<String> list) {  **this**.**list** = list;  }  **public void** output() {  System.***out***.println(**list**);  } }  （2）在spring配置文件中引入名称空间util  create a new spring configuration file bean2.xml, and Introduce util namespace  <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  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"**>  （3）使用util标签完成list集合注入提取 Extract list collection injection using util tag  *<!--1 Extract list collection injection using util tag -->* <**util:list id="bookList"**>  <**value**>Java</**value**>  <**value**>Database</**value**>  <**value**>MySQL</**value**> </**util:list**>  *<!--2 Inject the extracted list collection -->* <**bean id="book" class="com.qixin.collection.Book"**>  <**property name="list" ref="bookList"**/> </**bean**>  （4）编写测试方法  @Test **public void** testBook() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean2.xml"**);  Book book = context.getBean(**"book"**, Book.**class**);  book.output(); } |

IOC操作中的Bean管理（bean作用域）Bean management in IOC operations（bean scope）

|  |
| --- |
| 1、在Spring里面，可以设置创建的bean实例是单实例还是多实例  In Spring, you can set whether the created bean instance is a single instance or a multi-instance  2、在Spring里面，默认情况下，bean是单实例对象  In Spring, beans are single-instance objects by default  **public class** TestBook {  @Test  **public void** testBook() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean2.xml"**);  Book book1 = context.getBean(**"book"**, Book.**class**);  Book book2 = context.getBean(**"book"**, Book.**class**);  System.***out***.println(book1);  System.***out***.println(book2);  } }    3、如何设置单实例还是多实例 How to set up single instance or multi-instance?  （1）在spring配置文件bean标签里面有属性（scope）用于设置单实例还是多实例  There is a property (scope) in the bean tag of the spring configuration file to set single instance or multi-instance  （2）scope属性值 scope property have two value  第一个值 默认值 singleton，表示是单实例对象  第二个值 prototype，表示是多实例对象  The first value：The default value is singleton, which means it is a single instance object  The second value：prototype, indicates that it is a multi-instance object  <**bean id="book" class="com.qixin.collection.Book" scope="prototype"**>  <**property name="list" ref="bookList"**/> </**bean**>    （3）singleton和prototype区别  The difference between singleton and prototype  设置scope值是single时候，加载spring配置文件的时候就会创建单实例对象  When the scope value is set to single, a single instance object will be created when the spring configuration file is loaded  设置scope值是prototype时候，不是在加载spring配置文件的时候创建多实例对象，而是在调用getBean方法时候创建多实例对象  When the scope value is set to prototype, the multi-instance objects are not created when the spring configuration file is loaded, but they are created when the getBean method is called |

IOC操作中的Bean管理（bean生命周期）Bean management in IOC operations（bean life cycle）

|  |
| --- |
| 1、什么是bean的生命周期 what’s the bean life cycle?  从对象创建到对象销毁的过程  The bean life cycle is the process from object creation to object destruction  2、bean生命周期 bean life cycle  （1）通过构造器创建bean实例（无参数构造） Create a bean instance through the constructor (no parameter construction)  （2）为bean的属性设置值和对其他bean的引用（调用set方法）Set values for bean properties and references to other beans (call the set method)  （3）调用bean的初始化方法（需要配置初始化的方法）Call the initialization method of the bean (the initialization method needs to be configured)  （4）bean可以使用了（对象获取到了）The bean is ready to use (the object is obtained)  （5）当容器关闭时候，调用bean的销毁的方法（需要配置销毁的方法）When the container is closed, the destroy method of the bean is called (the destroy method needs to be configured)  3、演示bean的生命周期 Demonstrate the life cycle of a bean  **package** com.qixin.bean;  **public class** Order {  *//no parameter construction* **public** Order() {  System.***out***.println(**"The first step is to create a bean instance through no parameter construction"**);  }   **private** String **name**;   **public void** setName(String name) {  **this**.**name** = name;  System.***out***.println(**"The second step is to call the set method to set the bean’s property"**);  }   *//* *Create a method that performs initialization* **public void** initMethod() {  System.***out***.println(**"The third step is to call the initialization method of the bean"**);  }   *//* *Create a method that performs destruction* **public void** destroyMethod() {  System.***out***.println(**"The fifth step is to execute the method of destruction"**);  } }  **define bean4.xml**  <**bean id="order" class="com.qixin.bean.Order" init-method="initMethod" destroy-method="destroyMethod"**>  <**property name="name" value="computer"**/> </**bean**>  **public class** TestOrder {  @Test  **public void** testOrder() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean4.xml"**);  Order order = context.getBean(**"order"**, Order.**class**);  System.***out***.println(**"The fourth step is to get the created bean instance object"**);  System.***out***.println(order);   *//* *Manually call the close method to destroy the bean instance* ((ClassPathXmlApplicationContext)context).close();  } }    4、bean的后置处理器，bean生命周期有7步  （1）通过构造器创建bean实例（无参数构造） Create a bean instance through the constructor (no parameter construction)  （2）为bean的属性设置值和对其他bean的引用（调用set方法）Set values for bean properties and references to other beans (call the set method)  （3）调用bean的初始化方法（需要配置初始化的方法）Call the initialization method of the bean (the initialization method needs to be configured)  **（4）把bean实例传递给bean后置处理器的方法postProcessBeforeInitialization**  **Pass the bean instance to the method postProcessBeforeInitialization of the bean post processor**  （5）bean可以使用了（对象获取到了）The bean is ready to use (the object is obtained)  **（6）把bean实例传递给bean后置处理器的方法postProcessAfterInitialization**  **Pass the bean instance to the method postProcessAfterInitialization of the bean post processor**  （7）当容器关闭时候，调用bean的销毁的方法（需要配置销毁的方法）When the container is closed, the destroy method of the bean is called (the destroy method needs to be configured)  5、演示添加后置处理器效果 Demonstrate adding post processor effects  （1）创建类，实现接口BeanPostProcessor  create class MyBeanPost implements interface BeanPostProcessor  **package** com.qixin.bean;  **public class** MyBeanPost **implements** BeanPostProcessor {   @Override  **public** Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println(**"The method executed before initialization "**);  **return** BeanPostProcessor.**super**.postProcessBeforeInitialization(bean, beanName);  }   @Override  **public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println(**"The method executed after initialization "**);  **return** BeanPostProcessor.**super**.postProcessAfterInitialization(bean, beanName);  } }  *<!—config post processor-->* <**bean id="myBeanPost" class="com.qixin.bean.MyBeanPost"**></**bean**> |

IOC操作中的Bean管理（xml自动装配）Bean management in IOC operations（xml autowire）

|  |
| --- |
| 1、什么是自动装配 What is autowire?  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  According to the specified assembly rules (property name or property type), Spring automatically injects matching property values  2、演示自动装配过程 Demonstrate the automatic assembly process  （1）创建类 create class Emp and Dept  **public class** Emp {  **private** Dept **dept**;   **public void** setDept(Dept dept) {  **this**.**dept** = dept;  }   @Override  **public** String toString() {  **return "Emp{"** +  **"dept="** + **dept** +  **'}'**;  }   **public void** output() {  System.***out***.println(**dept**);  } }  **public class** Dept {  @Override  **public** String toString() {  **return "Dept{}"**;  } }  创建配置文件bean5.xml create configuration file bean5.xml  在bean标签中使用autowire属性来实现自动装配，autowire属性有两个值：  Autowiring using the autowire property in the bean tag，autowire property have two value：   * byName根据属性名称注入，注入值bean的id值和类属性名称一样   Inject according to the property name, the id value of the injected bean is the same as the class property name   * byType根据属性类型注入，相同类型的bean不能有多个   Inject according to the property type, there cannot be more than one bean of the same type  （2）根据属性名称自动注入 Automatic injection based on property name<**bean id="emp" class="com.qixin.autowire.Emp" autowire="byName"**>  *<!--<property name="dept" ref="dept"/>-->* </**bean**> <**bean id="dept" class="com.qixin.autowire.Dept"**></**bean**>  （3）根据属性类型自动注入 Automatic injection based on property type<**bean id="emp" class="com.qixin.autowire.Emp" autowire="byType"**>  *<!--<property name="dept" ref="dept"/>-->* </**bean**> <**bean id="dept" class="com.qixin.autowire.Dept"**></**bean**>  （4）Testing  **public class** TestAutowire {  @Test  **public void** testAutowire() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean5.xml"**);  Emp emp = context.getBean(**"emp"**, Emp.**class**);  System.***out***.println(emp);  } } |

IOC操作中的Bean管理（引入外部属性文件）Bean management in IOC operations（import external properties file）

|  |
| --- |
| 1、直接配置数据信息 Config database information directly  In addition to using jdbc to operate the database, we often use database connection pool to improve database access performance. Druid is an implementation of a database connection pool on Alibaba's open source platform.  （1）配置druid连接池 Config druid connection pool  （2）引入druid连接池Jar包 import druid connection pool package    （3）创建配置文件bean6.xml Create configuration file bean6.xml  *<!--* *Configure the connection pool directly -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"**>  <**property name="driverClassName" value="com.mysql.cj.jdbc.Driver"**/>  <**property name="url" value="jdbc:mysql://localhost:3306/demo?characterEncoding=UTF-8&amp;serverTimezone=Asia/Shanghai"**/>  <**property name="username" value="root"**/>  <**property name="password" value="1234"**/> </**bean**>  2、通过引入外部属性文件配置数据连接池 Configure the data connection pool by introducing an external properties file  （1）创建外部属性文件，properties格式文件，写数据库信息  Create an external properties file named jdbc.properties,write the database information in it.    （2）把外部properties属性文件引入到spring配置文件中  Import the properties property file into the spring configuration file   * 引入context名称空间 import context namespace   <**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"**>   * 在spring配置文件使用标签引入外部属性文件 import external properties file in spring configuration file   *<!—import exteranl properties file-->* <**context:property-placeholder location="classpath:jdbc.properties"**/> *<!—config connection pool-->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"**>  <**property name="driverClassName" value="${driverClassName}"**/>  <**property name="url" value="${url}"**/>  <**property name="username" value="${username}"**/>  <**property name="password" value="${password}"**/> </**bean**> |

IOC操作中的Bean管理（基于注解方式）Bean management in IOC operations（Base on annotation）

|  |
| --- |
| 1、什么是注解 what’s annotation?  （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)  Annotation is a special code tag, format: @ annotation name (property name=property value, property name=property value)  （2）注解作用在类上面，方法上面，属性上面  Annotations work on classes, methods, and properties  （3）使用注解目的：为了简化xml配置  The purpose of using annotations: to simplify xml configuration  2、Spring针对Bean管理中创建对象提供了以下四个注解  Spring provides the following four annotations for creating objects in Bean management  （1）@Component  （2）@Service  （3）@Controller  （4）@Repository  \*上面四个注解功能是一样的，都可以用来创建bean实例，习惯将不同注解用在不同的层上  The above four annotations have the same function and can be used to create bean instances. Usually, the above annotations are used on different layers.  3、基于注解方式的创建对象 base on annotation to create objects  Copy and create the new project named spring5\_demo3  第一步 引入依赖 the first step import dependencies    第二步 开启组件扫描 The second step enable component scan  *<!—enable component scan  1 如果扫描多个包，多个包之间使用逗号隔开 If scanning multiple packages, separate multiple packages with commas  2 扫描包上层目录 Scan the upper directory of the package -->* <**context:component-scan base-package="com.qixin"**/> *<!--<context:component-scan base-package="com.qixin.dao,com.qixin.service"/>-->*  第三步 创建类，在类上面添加创建对象注解 The third step Create a class and add annotations to the class  *//在注解里面value属性值可以省略不写 In the annotation, the property value can be omitted or not written //默认值是类名称，首字母小写 The default value is the class name, with the first letter lowercase //UserService --> userService //@Service(value = "userService")* @Service **public class** UserService {  **public void** add() {  System.***out***.println(**"service add......"**);  } }  第四步 创建测试类 The fourth step create testing class  **public class** TestDemo {  @Test  **public void** testDemo1() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  UserService userService = context.getBean(**"userService"**, UserService.**class**);  System.***out***.println(userService);  userService.add();  } }    4、开启组件扫描细节配置 Enable component scan details configuration  *<!—example 1  use-default-filters="false" 表示现在不使用默认filter，自己配置filter Indicates that the default filter is not used now, and the filter is configured by yourself  context:include-filter，设置扫描哪些注解 Set which annotations to scan -->* <**context:component-scan base-package="com.qixin" use-default-filters="false"**>  <**context:include-filter type="annotation" expression="org.springframework.stereotype.Service"**/> </**context:component-scan**>  *<!—example 2  下面配置扫描包所有注解 Configure all annotations of the scan package  context:exclude-filter 设置哪些注解不进行扫描 Set which annotations are not scanned -->* <**context:component-scan base-package="com.qixin"**>  <**context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"**/> </**context:component-scan**>  5、基于注解方式实现属性注入 Property injection based on annotation  （1）@Autowired：根据属性类型进行自动装配 Autowiring based on property type  第一步 创建service和dao对象创建，并在对象上添加相应的注解  The first step Create service and dao objects, and add corresponding annotations to the objects  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  The second step inject UserDao object in UserService class, add userDao property in UserService class, use @Autowired annotation above property.  ---------------------------------------------------------------------------------------------------------  @Service **public class** UserService {   *//定义dao类型属性 define userDao property   //不需要添加set方法 No need to add set method  //添加注入属性注解 add @Autowired annotation* @Autowired  **private** UserDao **userDao**;   **public void** add() {  System.***out***.println(**"service add......"**);  **userDao**.add();  } }  ---------------------------------------------------------------------------------------------------------  **public interface** UserDao {  **public void** add(); }  ---------------------------------------------------------------------------------------------------------  @Repository **public class** UserDaoImpl **implements** UserDao{  @Override  **public void** add() {  System.***out***.println(**"dao add......"**);  } }  ---------------------------------------------------------------------------------------------------------  **public class** TestDemo {  @Test  **public void** testDemo1() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  UserService userService = context.getBean(**"userService"**, UserService.**class**);  System.***out***.println(userService);  userService.add();  } }    （2）@Qualifier：根据属性名称进行注入 Inject based on property name  @Qualifier注解的使用，要配合@Autowired一起使用  The use of @Qualifier annotation should be used together with @Autowired  一个dao接口可以有多个实现类，那就不能通过类型注入，必须通过属性名称来注入  A dao interface can have multiple implementation classes, so it cannot be injected by type, but must be injected by property name  ---------------------------------------------------------------------------------------------------------  @Repository **public class** UserDaoImpl1 **implements** UserDao{  @Override  **public void** add() {  System.***out***.println(**"dao1 add......"**);  } }  ---------------------------------------------------------------------------------------------------------  @Repository **public class** UserDaoImpl2 **implements** UserDao{  @Override  **public void** add() {  System.***out***.println(**"dao2 add......"**);  } }  ---------------------------------------------------------------------------------------------------------  @Autowired @Qualifier(value = **"userDaoImpl1"**) *//根据名称进行注入 Inject by name* **private** UserDao **userDao**;  （3）@Resource：可以根据类型注入，也可以根据名称注入 Can be injected by type or by name  \*注意：@Resource 是javax.annotation.Resource包中的注解  \*Note: @Resource is an annotation in the javax.annotation.Resource package  *//@Resource //根据类型进行注入 Inject by type* @Resource(name = **"userDaoImpl1"**) *//根据名称进行注入 Inject by name* **private** UserDao **userDao**;  （4）@Value：注入普通类型属性  @Value(value = **"abc"**) **private** String **name**;  ---------------------------------------------------------------------------------------------------------  **public void** add() {  System.***out***.println(**"service add......"**);  System.***out***.println(**"name = "** + **name**);  **userDao**.add(); }    6、纯注解实现 Full annotation implementation  （1）创建配置类，替代xml配置文件 Create a configuration class to replace the xml configuration file  @Configuration *//作为配置类，替代xml配置文件 As a configuration class, instead of the xml configuration file* @ComponentScan(basePackages = {**"com.qixin"**}) **public class** SpringConfig {  }  （2）编写测试类 create testing code  @Test **public void** testDemo2() {  *//加载配置类 load configuration class* ApplicationContext context = **new** AnnotationConfigApplicationContext(SpringConfig.**class**);  UserService userService = context.getBean(**"userService"**, UserService.**class**);  System.***out***.println(userService);  userService.add(); } |

AOP概念 AOP Concept

|  |
| --- |
| 1、什么是AOP（Aspect-Oriented Programming）What is AOP?  （1）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。  Aspect-oriented programming: AOP can be used to isolate various parts of business logic, thereby reducing the coupling between various parts of business logic, improving program reusability, and improving development efficiency.  （2）通俗描述：不通过修改源代码方式，在主干功能里面添加新功能。  Popular description: Add new functions to the main function without modifying the source code.  （3）使用登陆例子说明AOP  Use the login example to illustrate AOP    提出需求，在登陆功能基础之上添加权限判断的功能。  Now we put forward a demand: add the function of authority judgment on the basis of the login function.  原始方式：需要修改源代码实现  Use the original way, we need to modify the source code to implement the function  if adminstrator  ……  else if user  ……  AOP方式：不需要修改源代码来添加新的功能  Use the AOP way, we add new functions to the main function without modifying the source code.  我们只需要添加新的权限判断模块，然后将它配置到程序当中。  We only need to add a new authority judgment module, and then config it into the program. |

AOP底层原理 The underlying principle of AOP

|  |
| --- |
| 1、AOP底层使用动态代理 AOP bottom layer uses dynamic proxy  这里提到一个词——动态代理，什么是动态代理呢？  Here is a word – dynamic proxy，what’s the dynamic proxy?  动态代理就是在程序运行期间，创建目标对象的代理对象，并对目标对象中的方法进行功能性增强的一种技术。在生成代理对象的过程中，目标对象不变，代理对象中的方法是目标对象方法的增强方法。  Dynamic proxy is a technology that creates a proxy object of the target object and enhances the functions of the methods in the target object during the running of the program. In the process of generating the proxy object, the target object remains unchanged, and the method in the proxy object is the enhancement method of the target object method.  （1）动态代理有两种情况 There are two cases of dynamic proxy    第一种情况 目标对象有接口，使用JDK动态代理  In the first case, the target object has an interface and uses JDK dynamic proxy   * 创建接口实现类代理对象，增强类的方法 Create a proxy object of the interface implementation class and enhance the method of the class     第二种情况 目标对象没有接口，使用CGLIB动态代理  In the second case, the target object has no interface and uses CGLIB dynamic proxy   * 创建子类的代理对象，增强类的方法 Create a proxy object of the subclass to enhance the method of the class |

AOP底层原理（JDK动态代理）The underlying principle of AOP（JDK dynamic proxy）

|  |
| --- |
| 1、使用JDK动态代理，使用Proxy类里面的方法创建代理对象  <https://www.matools.com/api/java8>  <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/reflect/Proxy.html>  java.lang.reflect package  public class Proxy  extends Object  implements Serializable  Proxy提供了用于创建对象的静态方法，这些对象充当接口实例但允许自定义方法调用。  Proxy provides static methods for creating objects that act like instances of interfaces but allow for customized method invocation.  （1）调用newProxyInstance方法 call the newProxyInstance method  public static Object newProxyInstance(ClassLoader loader,  Class<?>[] interfaces,  InvocationHandler h)  返回指定接口的代理实例，该接口将方法调用分派给指定的调用处理程序。  Returns a proxy instance for the specified interfaces that dispatches method invocations to the specified invocation handler.  方法里面有三个参数：this method have three parameter.   * 第一个参数：类加载器 the class loader to define the proxy class * 第二个参数：增强方法所在的类，这个类实现的接口，支持多个接口 the list of interfaces for the proxy class to implement * 第三个参数：调度方法被分派的调用处理程序the invocation handler to dispatch method invocations to   第三个参数非常重要，我们要实现这个接口InvocationHandler，创建代理对象，写增强的方法。  The third parameter is very important. We need to implement this interface InvocationHandler, create proxy objects, and write enhanced methods.  2、实现JDK动态代理代码 Implement JDK dynamic proxy code  复制并创建新的项目 spring5\_demo4 copy and create a new project named spring5\_demo4  （1）创建接口，定义方法 Create interfaces, define methods  **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) {  System.***out***.println(**"add方法执行了......"**);  **return** a+b;  }   @Override  **public** String update(String id) {  System.***out***.println(**"update方法执行了......"**);  **return** id;  } }  （3）使用Proxy类创建接口代理对象 Use the Proxy class to create interface proxy objects  **public class** JDKProxy {  **public static void** main(String[] args) {  *//创建接口实现类的代理对象* Class[] interfaces = {UserDao.**class**};  UserDao userDao = **new** UserDaoImpl();  UserDao dao = (UserDao) Proxy.*newProxyInstance*(  JDKProxy.**class**.getClassLoader(),  interfaces,  **new** UserDaoProxy(userDao)  );  **int** result = dao.add(1, 2);  System.***out***.println(result);  } }  *//创建代理对象代码 Create proxy object code* **class** UserDaoProxy **implements** InvocationHandler {  *//1 通过有参构造方法把目标对象传递过来 Pass the target object through the constructor with parameters* **private** Object **obj**;  **public** UserDaoProxy(Object obj) {  **this**.**obj** = obj;  }  *//增强的逻辑代码 Enhanced logic code* @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *//方法之前 before method* System.***out***.println(**"execute before method......"** + method.getName() + **": passed parameters..."** + Arrays.*toString*(args));  *//执行被增强的方法 execute the enhanced method* Object res = method.invoke(**obj**, args);  *//方法之后 after method* System.***out***.println(**"execute after method......"** + **obj**);  **return** res;  } } |

AOP术语 AOP terminology

|  |
| --- |
| **public class** User {  **public void** add() {  }  **public void** delete() {  }  **public void** update() {  }  **public void** select() {  } }  1、连接点（Joinpoint）  类里面哪些方法可以被增强，这些方法称为连接点 Which methods in the class can be enhanced, these methods are called join points  2、切入点（Pointcut）  实际被真正增强的方法，称为切入点 The method that is actually enhanced, called the pointcut  3、通知（Advice）  （1）实际增强的逻辑代码称为通知（增强）The logic code that actually enhances is called notification (enhancement)  （2）通知有五种类型 advice have five types   * 前置通知 Before advice @Before * 后置通知 AfterReturning advice @AfterReturning * 环绕通知 Around advice @Around * 异常通知 AfterThrowing advice @AfterThrowing * 最终通知 After advice @After   4、切面（Aspect）  把通知应用到切入点的过程叫做切面 The process of applying a notification to a pointcut is called an aspect |

AOP操作准备 Prepare for AOP Operation

|  |
| --- |
| 1、Spring框架一般都是基于AspectJ实现AOP操作 The Spring framework generally implements AOP operations based on AspectJ  （1）什么是AspectJ what’s the AspectJ?  AspectJ不是Spring组成部分，它是基于Java语言的AOP框架，一般把AspectJ和Spring框架一起使用，进行AOP操作  AspectJ is not a part of Spring. It is an AOP framework based on the Java language. Generally, AspectJ and Spring framework are used together to perform AOP operations.  2、基于AspectJ实现AOP操作 AOP operation based on AspectJ  （1）基于xml配置文件实现 Implementation based on xml configuration file  （2）基于注解方法实现（推荐） Implementation based on annotation (recommend)  3、在项目工程里面引入AOP相关依赖 Introduce AOP related dependencies into the project    4、切入点表达式 pointcut expression  （1）切入点表达式作用是知道对类里面的哪个方法进行增强 The pointcut expression is used to know which method in the class to enhance  （2）语法结构：syntax  execution([权限修饰符][返回类型][类全路径][方法名称]([参数列表]))  execution([permission modifier][return type][class full path][method name]([parameter list]))  举例1：对com.qixin.dao.BookDao类里面的add方法进行增强  Example 1：Enhance the add method in the com.qixin.dao.BookDao class  execution(\* com.qixin.dao.BookDao.add(..))  举例2：对com.qixin.dao.BookDao类里面所有方法进行增强  Example 2：Enhance all methods in the com.qixin.dao.BookDao class  execution(\* com.qixin.dao.BookDao.\*(..))  举例3：对com.qixin.dao包里面所有类的所有方法进行增强  Example 3：Enhance all methods of all classes in the com.qixin.dao package  execution(\* com.qixin.dao.\*.\*(..)) |

AOP操作（AspectJ注解）AOP operations (AspectJ annotations)

|  |
| --- |
| 1、创建包和类，在类里面定义方法 create package aop and User class, define methods in it  *package com.qixin.aop.annotation;*  *//目标类 target class* **public class** User {  **public void** add() {  *//int i = 10/0;*  System.***out***.println(**"add......"**);  } }  2、创建增强类（编写增强逻辑）Create enhancement class (write enhancement logic)  （1）在增强类里面，创建方法，让不同方法代表不同通知类型 In the enhanced class, create methods and let different methods represent different notification types  *//增强类 Enhanced class* **public class** UserProxy {  *//前置通知，在add方法之前执行 before advice, execute before the add method* **public void** before() {  System.***out***.println(**"before......"**);  } }  3、配置通知 Config advice  （1）在spring配置文件bean1.xml中，开启注解扫描 In the spring configuration file bean1.xml, enable annotation scanning  <**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"  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 enable annotation scanning -->* <**context:component-scan base-package="com.qixin.aop.annotation"**></**context:component-scan**> </**beans**>  （2）使用注解创建User和UserProxy对象 use annotation to create User and UserProxy objects@Component **public class** User {  ---------------------------------------------------------------------------------------------------------@Component**public class** UserProxy {  （3）在增强类上面添加注解@Aspect Add the annotation @Aspect to the enhanced class@Component @Aspect *//生成代理对象 Generate proxy object* **public class** UserProxy {  （4）在spring配置文件中开启Aspect生成代理对象 Turn on Aspect in the spring configuration file to generate proxy objects  *<!--开启Aspect生成代理对象 Turn on Aspect to generate proxy objects -->* <**aop:aspectj-autoproxy**></**aop:aspectj-autoproxy**>  4、配置不同类型的通知 Configure different types of advices  Do you remember what types of advices there are?  （1）在增强类的里面，在通知方法上面添加通知类型注解，使用切入点表达式配置  In the enhanced class, add a advice annotation to the advice method and configure it using a pointcut expression@Component @Aspect**public class** UserProxy {  *//前置通知 Before advice* @Before(value = **"execution(\* com.qixin.aop.annotation.User.add(..))"**)  **public void** before() {  System.***out***.println(**"before......"**);  }   *//后置通知（返回通知）AfterReturning advice* @AfterReturning(value = **"execution(\* com.qixin.aop.annotation.User.add(..))"**)  **public void** afterReturning() {  System.***out***.println(**"afterReturning......"**);  }   *//最终通知 After advice* @After(value = **"execution(\* com.qixin.aop.annotation.User.add(..))"**)  **public void** after() {  System.***out***.println(**"after......"**);  }   *//异常通知 AfterThrowing advice* @AfterThrowing(value = **"execution(\* com.qixin.aop.annotation.User.add(..))"**)  **public void** afterThrowing() {  System.***out***.println(**"afterThrowing......"**);  }   *//环绕通知 Around advice* @Around(value = **"execution(\* com.qixin.aop.annotation.User.add(..))"**)  **public void** around(ProceedingJoinPoint point) **throws** Throwable {  System.***out***.println(**"before arround......"**);  *//被增强的方法执行 Enhanced method execution* point.proceed();  System.***out***.println(**"after arround......"**);  } }  5、Testing  **package** com.qixin.test;  **public class** TestAop {  @Test  **public void** testAopAnnotation() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  User user = context.getBean(**"user"**, User.**class**);  user.add();  } }    6、相同切入点抽取 Same pointcut extraction@Pointcut(value = "execution(\* com.qixin.aop.annotation.User.add(..))") public void pointdemo() { }@Before(value = "pointdemo()") public void before() {  System.*out*.println("before......"); }  7、有多个增强类，对同一个方法进行增强，设置增强优先级 If there are multiple enhancement classes to enhance the same method, the enhancement priority can be set  （1）在增强类上面添加注解@Order(数字类型值)，值越小优先级越高  Add the annotation @Order (value of numeric type) to the enhanced class, the smaller the value, the higher the priority  @Component @Aspect@Order(1) **public class** PersonProxy {  @Pointcut(value = **"execution(\* com.qixin.aop.annotation.User.add(..))"**)  **public void** pointdemo() {  }  @Before(value = **"pointdemo()"**)  **public void** before() {  System.***out***.println(**"person before......"**);  } }  再次测试，输出如下结果。Test again and output the following results.    8、使用完全注解开发 Develop with fully annotated  （1）创建配置类，不需要创建xml配置文件 Create a configuration class without using an xml configuration file  @Configuration @ComponentScan(basePackages = {**"com.qixin"**}) @EnableAspectJAutoProxy(proxyTargetClass = **true**) **public class** ConfigAop { }  （2）创建测试方法 create testing method  @Test **public void** testAopAnnotation2() {  ApplicationContext context = **new** AnnotationConfigApplicationContext(ConfigAop.**class**);  User user = context.getBean(**"user"**, User.**class**);  user.add(); } |

AOP操作（AspectJ配置文件）AOP operations (AspectJ configuration file)

|  |
| --- |
| 1、创建两个类，目标类和被增强类，创建方法 Create two classes, the target class and the enhanced class, and create methods  **package com.qixin.aop.xml;**  **public class** Book {  **public void** buy() {  System.***out***.println(**"buy......"**);  } }  ---------------------------------------------------------------------------------------------------------  **public class** BookProxy {  **public void** before() {  System.***out***.println(**"before......"**);  }  **public void** around(ProceedingJoinPoint point) **throws** Throwable {  System.***out***.println(**"before around......"**);point.proceed();  System.***out***.println(**"after around......"**);  } }  2、在Spring配置文件bean2.xml中创建两个类对象 Create two class objects in Spring configuration file named bean2.xml  *<!--创建对象 create objects-->* <**bean id="book" class="com.qixin.aop.xml.Book"**></**bean**> <**bean id="bookProxy" class="com.qixin.aop.xml.BookProxy"**></**bean**>  3、在Spring配置文件中配置切入点 Configure pointcuts in Spring configuration files  *<!--配置aop增强 Configure aop enhancements -->* <**aop:config**>  *<!--配置切入点 Configure pointcut -->* <**aop:pointcut id="point" expression="execution(\* com.qixin.aop.xml.Book.buy(..))"**/>  *<!--配置切面 Configure Aspects -->* <**aop:aspect ref="bookProxy"**>  *<!--配置增强作用在具体的方法上 Configuration enhancements work on specific methods -->* <**aop:before method="before" pointcut-ref="point"**/>  <**aop:around method="around" pointcut-ref="point"**/>  </**aop:aspect**> </**aop:config**>  4、Testing  @Test **public void** testAopXml() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean2.xml"**);  Book book = context.getBean(**"book"**, Book.**class**);  book.buy(); } |

JdbcTemplate（概念和准备）JdbcTemplate（Concept and preparation）

|  |
| --- |
| 1、什么是JdbcTemplate What is JdbcTemplate？  Spring框架对JDBC进行封装，使用JdbcTemplate方便实现对数据库操作  The Spring framework encapsulates JDBC and uses JdbcTemplate to facilitate database operations  2、准备工作 Preparation  Copy and create new project named spring5\_demo5.  （1）引入相关jar包 Import related jar packages    （2）在spring配置文件中配置数据库连接池 Configure database connection pool in spring configuration file  *<!-- 数据库连接池 database pool-->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"  destroy-method="close"**>  <**property name="url" value="jdbc:mysql://localhost:3306/user\_db?characterEncoding=UTF-8&amp;serverTimezone=Asia/Shanghai"** />  <**property name="username" value="root"** />  <**property name="password" value="1234"** />  <**property name="driverClassName" value="com.mysql.cj.jdbc.Driver"** /> </**bean**>  （3）配置JdbcTemplate对象，注入DataSource属性 Configure the JdbcTemplate object and inject the DataSource property  *<!—JdbcTemplate object-->* <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  *<!—inject dataSource-->* <**property name="dataSource" ref="dataSource"**/> </**bean**>  （4）创建service和dao类，在dao注入jdbcTemplate对象 Create service and dao classes and inject jdbcTemplate objects in dao   * 配置文件 Configuration file   *<!--开启组件扫描 Enable component scan -->* <**context:component-scan base-package="com.qixin"**></**context:component-scan**>   * Dao   **public interface** BookDao { }  @Repository **public class** BookDaoImpl **implements** BookDao{  *//inject jdbcTemplate* @Autowired  **private** JdbcTemplate **jdbcTemplate**; }   * Service   @Service **public class** BookService {  *//inject dao* @Autowired  **private** BookDao **bookDao**; } |

JdbcTemplate操作数据库（添加）JdbcTemplate operation database (add)

|  |
| --- |
| 1、对应数据库创建实体类 Create entity class corresponding to the database      **public class** Book {  **private** String **userId**;  **private** String **username**;  **private** String **status**;  ……  2、在service和dao层编写代码 Write code at the service and dao layers  （1）在dao层进行数据库添加操作 Add database operations at the dao layer  （2）调用JdbcTemplate对象里面update方法实现添加操作 Call the update method in the JdbcTemplate object to implement the add operation    有两个参数 have two parameters   * 第一个参数：sql语句 the first parameter: sql statement * 第二个参数：可变参数，设置sql语句值 the second parameter: Variable parameter, set sql statement value   **public interface** BookDao {**void** add(Book book);  }  @Repository **public class** BookDaoImpl **implements** BookDao{  *//inject jdbcTemplate* @Autowired  **private** JdbcTemplate **jdbcTemplate**;   *//add method* @Override  **public void** add(Book book) {  *//create sql* String sql = **"insert into t\_book values(?,?,?)"**;  *//call the jdbcTemplate update method* Object[] args = {book.getUserId(), book.getUsername(), book.getStatus()};  **int** n = **jdbcTemplate**.update(sql, args);  System.***out***.println(n);  } }  @Service **public class** BookService {@Autowired  **private** BookDao **bookDao**;   **public void** addBook(Book book) {  **bookDao**.add(book);  }  }  3、测试类  **public class** TestBook {  @Test  **public void** testAdd() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  BookService bookService = context.getBean(**"bookService"**, BookService.**class**);  Book book = **new** Book();  book.setUserId(**"1"**);  book.setUsername(**"Java"**);  book.setStatus(**"T"**);  bookService.addBook(book);  } } |

JdbcTemplate操作数据（修改和删除） JdbcTemplate operation database (update and delete)

|  |
| --- |
| 1、修改 update  @Override **public void** update(Book book) {String sql = **"update t\_book set username=?,status=? where user\_id=?"**;Object[] args = {book.getUsername(), book.getStatus(), book.getUserId()};  **int** n = **jdbcTemplate**.update(sql, args);  System.***out***.println(n); }  @Test **public void** testUpdate() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  BookService bookService = context.getBean(**"bookService"**, BookService.**class**);  Book book = **new** Book();  book.setUserId(**"1"**);  book.setUsername(**"Java"**);  book.setStatus(**"F"**);  bookService.updateBook(book); }  2、删除 delete  @Override **public void** delete(String id) {String sql = **"delete from t\_book where user\_id=?"**;**int** n = **jdbcTemplate**.update(sql, id);  System.***out***.println(n); }  @Test **public void** testDelete() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  BookService bookService = context.getBean(**"bookService"**, BookService.**class**);  bookService.deleteBook(**"1"**); } |

JdbcTemplate操作数据库（查询-返回某个值） JdbcTemplate operation database (Query - returns a value)

|  |
| --- |
| 1、查询表里面的总记录，返回某个值 Query the total records in the table and return a certain value    有两个参数 have two parameters   * 第一个参数：sql语句 the first parameter: sql statement * 第二个参数：返回类型的Class return type of Class   @Override **public int** selectCount() {String sql = **"select count(\*) from t\_book"**;**return jdbcTemplate**.queryForObject(sql, Integer.**class**); }  ----------------------------------------------------------  **public int** findCount() {  **return bookDao**.selectCount(); }  3、Testing  @Test **public void** testSelectCount() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  BookService bookService = context.getBean(**"bookService"**, BookService.**class**);  **int** count = bookService.findCount();  System.***out***.println(count); } |

JdbcTemplate操作数据库（查询-返回对象）JdbcTemplate operation database (Query - return object)

|  |
| --- |
| 1、查询图书详情 Querying Book Details    有三个参数 have three parameters   * 第一个参数：sql语句 the first parameter: sql statement * 第二个参数：RowMapper接口，返回不同类型数据，使用这个接口的实现类完成数据封装 the second parameter: RowMapper interface, returns different types of data, use the implementation class of this interface to complete data encapsulation * 第三个参数：sql语句值 the third parameter: sql statement value   @Override **public** Book findBookInfo(String id) {  String sql = **"select \* from t\_book where user\_id=?"**;Book book = **jdbcTemplate**.queryForObject(sql, **new** BeanPropertyRowMapper<Book>(Book.**class**), id);  **return** book; }  --------------------------------------------------------  **public** Book findOne(String id) {  **return bookDao**.findBookInfo(id); }  3、Testing  @Test **public void** testSelectBook() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  BookService bookService = context.getBean(**"bookService"**, BookService.**class**);  Book book = bookService.findOne(**"1"**);  System.***out***.println(book); } |

JdbcTemplate操作数据库（查询-返回集合）JdbcTemplate operation database (Query – return collection)

|  |
| --- |
| 1、查询图书列表 Query book list  2、调用JdbcTemplate方法实现查询返回集合 Call the JdbcTemplate method to implement the query and return collection    @Override **public** List<Book> findAllBook() {  String sql = **"select \* from t\_book"**;  List<Book> bookList = **jdbcTemplate**.query(sql, **new** BeanPropertyRowMapper<Book>(Book.**class**));  **return** bookList; }  ------------------------------------------------------  **public** List<Book> findAll() {  **return bookDao**.findAllBook(); }  **3、Testing**  @Test **public void** testSelectAll() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  BookService bookService = context.getBean(**"bookService"**, BookService.**class**);  List<Book> bookList = bookService.findAll();  **for** (Book book : bookList) {  System.***out***.println(book);  } } |

JdbcTemplate操作数据库（批量操作）JdbcTemplate operation database (batch operation)

|  |
| --- |
| 1、批量操作：操作表里面多条记录 Batch Operations: Manipulate multiple records in a table  2、JdbcTemplate实现批量添加操作 JdbcTemplate implements batch add operations    有两个参数：have two parameters   * 第一个参数：sql语句 The first parameter: sql statment * 第二个参数：List集合，添加多条记录数据 The second parameter: List colletion, add multiple data   @Override **public void** batchAddBook(List<Object[]> batchArgs) {  String sql = **"insert into t\_book values(?,?,?)"**;  **int**[] ints = **jdbcTemplate**.batchUpdate(sql, batchArgs);  System.***out***.println(Arrays.*toString*(ints)); }  ------------------------------------------------------------------------------------------------------------------  **public void** batchAdd(List<Object[]> batchArgs) {  **bookDao**.batchAddBook(batchArgs); }  ------------------------------------------------------------------------------------------------------------------  @Test **public void** testBatchAdd() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  BookService bookService = context.getBean(**"bookService"**, BookService.**class**);  List<Object[]> bactchArgs = **new** ArrayList<>();  Object[] o1 = {**"3"**,**"C"**,**"F"**};  Object[] o2 = {**"4"**,**"C++"**,**"T"**};  Object[] o3 = {**"5"**,**"Redis"**,**"F"**};  bactchArgs.add(o1);  bactchArgs.add(o2);  bactchArgs.add(o3);  bookService.batchAdd(bactchArgs); }  3、JdbcTemplate实现批量修改操作 JdbcTemplate implements batch modification operations  @Override **public void** batchUpdateBook(List<Object[]> batchArgs) {  String sql = **"update t\_book set username=?,status=? where user\_id=?"**;  **int**[] ints = **jdbcTemplate**.batchUpdate(sql, batchArgs);  System.***out***.println(Arrays.*toString*(ints)); }  ------------------------------------------------------------------------------------------------------------------  **public void** batchUpdate(List<Object[]> batchArgs) {  **bookDao**.batchUpdateBook(batchArgs); }  ------------------------------------------------------------------------------------------------------------------  @Test **public void** testBatchUpdate() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  BookService bookService = context.getBean(**"bookService"**, BookService.**class**);  List<Object[]> bactchArgs = **new** ArrayList<>();  Object[] o1 = {**"C"**,**"T"**,**"3"**};  Object[] o2 = {**"C++"**,**"T"**,**"4"**};  Object[] o3 = {**"Redis"**,**"T"**,**"5"**};  bactchArgs.add(o1);  bactchArgs.add(o2);  bactchArgs.add(o3);  bookService.batchUpdate(bactchArgs); }  4、JdbcTemplate实现批量删除操作 JdbcTemplate implements batch delete operations  @Override **public void** batchDeleteBook(List<Object[]> batchArgs) {  String sql = **"delete from t\_book where user\_id=?"**;  **int**[] ints = **jdbcTemplate**.batchUpdate(sql, batchArgs);  System.***out***.println(Arrays.*toString*(ints)); }  ------------------------------------------------------------------------------------------------------------------  **public void** batchDelete(List<Object[]> batchArgs) {  **bookDao**.batchDeleteBook(batchArgs); }  ------------------------------------------------------------------------------------------------------------------  @Test **public void** testBatchDelete() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  BookService bookService = context.getBean(**"bookService"**, BookService.**class**);  List<Object[]> bactchArgs = **new** ArrayList<>();  Object[] o1 = {**"3"**};  Object[] o2 = {**"4"**};  Object[] o3 = {**"5"**};  bactchArgs.add(o1);  bactchArgs.add(o2);  bactchArgs.add(o3);  bookService.batchDelete(bactchArgs); } |

事务概念 Transaction Concept

|  |
| --- |
| 1、什么是事务 what is a transaction?  （1）事务是数据库操作最基本单元，逻辑上一组操作，要么都成功，如果有一个失败所有操作都失败 Transaction is the most basic unit of database operation. Logically a group of operations, either all succeed or one fails, then all operations fail  （2）典型场景：银行转账 Typical scene: bank transfer  lucy转账100元给mary lucy少100，mary多100  Lucy transfers 100 yuan to Mary  Lucy's account decreased by 100 yuan, Mary's account increased by 100 yuan  2、事务的四个特性（ACID）Four properties of transactions（ACID）  （1）原子性：不可分割，要么都成功，要么都失败 Atomicity: indivisible, either all succeed or all fail  （2）一致性：操作之前和操作之后总量保持不变 Consistency: the total amount remains the same before and after the operation  （3）隔离性：多事务之间操作时彼此之间不会产生影响 Isolation: Multi-transaction operations will not affect each other  （4）持久性：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失Durability: After the transaction is completed, the modification to the data is permanent and will not be lost even if the system fails |

事务操作（搭建事务操作环境）Transaction operation (build a transaction operation environment)

|  |
| --- |
| 1、创建数据库表，添加记录 create table and add some records    2、复制和创建新的项目，创建service和dao，完成对象创建和注入  Copy and create new project named spring5\_demo6, create service and dao, Complete object creation and injection  （1）配置文件 configuration file  <**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"**>   *<!--开启组件扫描 enable component scan-->* <**context:component-scan base-package="com.qixin"**></**context:component-scan**>   *<!-- 数据库连接池 database pool -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"  destroy-method="close"**>  <**property name="url" value="jdbc:mysql://localhost:3306/user\_db?characterEncoding=UTF-8&amp;serverTimezone=Asia/Shanghai"** />  <**property name="username" value="root"** />  <**property name="password" value="1234"** />  <**property name="driverClassName" value="com.mysql.cj.jdbc.Driver"** />  </**bean**>   *<!—JdbcTemplate object-->* <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  *<!—inject dataSource-->* <**property name="dataSource" ref="dataSource"**/>  </**bean**> </**beans**>  （2）service注入dao，在dao注入JdbcTemplate，在JdbcTemplate注入DataSource  Inject dao in service, inject JdbcTemplate in dao, inject DataSource in JdbcTemplate  @Service **public class** UserService {  *//注入dao* @Autowired  **private** UserDao **userDao**; }  -------------------------------------------------------------------------------------------------------------  @Repository **public class** UserDaoImpl **implements** UserDao {   @Autowired  **private** JdbcTemplate **jdbcTemplate**; }  3、在dao创建两个方法：多钱和少钱的方法，在service创建转账方法  Create two methods in dao class, the less money and more money methods, then create transfer methods in service class.  @Override **public void** addMoney() {  String sql = **"update t\_account set money=money+? where username=?"**;  **jdbcTemplate**.update(sql, 100, **"mary"**); } *//lucy转账100给mary Lucy transfers 100 to mary* @Override **public void** reduceMoney() {  String sql = **"update t\_account set money=money-? where username=?"**;  **jdbcTemplate**.update(sql, 100, **"lucy"**); }  -------------------------------------------------------------------------------------------------------------  *//转账的方法 method of transfer* **public void** accountMoney() {  *//lucy少100* **userDao**.reduceMoney();  *//mary多100* **userDao**.addMoney(); }  -------------------------------------------------------------------------------------------------------------  **public class** TestAccount {  @Test  **public void** testAccountAnnotation() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  UserService userService = context.getBean(**"userService"**, UserService.**class**);  userService.accountMoney();  }  }  4、上面代码，如果正常执行没有问题，但是如果上面代码执行过程中出现异常，有问题  The above code, if there is no problem with normal execution, but if there is an exception during the execution of the above code, there is a problem  *//转账的方法* **public void** accountMoney() {  *//lucy少100* **userDao**.reduceMoney();  *//模拟异常 Simulate exceptions, make some mistakes manually* **int** i = 10/0;  *//mary多100* **userDao**.addMoney(); }  （1）上面问题如何解决呢？ How to solve the above problem?  \*使用事务进行解决 We can use transactions to solve this problem  （2）事务操作流程 transaction operation process  **public void** accountMoney() {  **try** {  *//第一步 开启事务 the first step enable transaction   //第二步 执行业务操作 the second step perform business operations* **userDao**.reduceMoney();**int** i = 10/0;**userDao**.addMoney();   *//第三步 没有发生异常，提交事务 the third step if no exception occurred, commit the transaction* } **catch** (Exception e) {  *//第四步 出现异常，事务回滚 the fourth step if an exception occurs, the transaction is rolled back*  } } |

事务操作（Spring事务管理介绍）Transaction operation (Introduction to Spring Transaction Management)

|  |
| --- |
| 1、事务管理添加到JavaEE三层结构里面的Service层（业务逻辑层）  Transaction management is added to the Service layer (business logic layer) in the JavaEE three-layer structure.  2、在Spring进行事务管理操作 Transaction management operations in Spring  （1）有两种方式：编程式事务管理（了解）和声明式事务管理（推荐）  There are two ways: programmatic transaction management (understanding) and declarative transaction management (recommended)  通过代码来编写就叫编程式事务管理，像上一节的例子，这种方式比较麻烦。  我们通常使用Spring的声明式事务管理。  We write it through code, which is called programmatic transaction management. Like the example in the previous section, this method is more troublesome.  So we usually use Spring's declarative transaction management.  3、Spring声明式事务管理 Spring declarative transaction management  （1）基于注解方式（推荐）Base on annotation (recommended)  （2）基于xml配置文件方式（了解）Base on xml configuration file (understanding)  4、在Spring进行声明式事务管理，底层使用AOP  Declarative transaction management in Spring, using AOP at the bottom layer  5、Spring事务管理API Spring transaction management API  （1）Spring针对事务管理，提供一个接口（PlatformTransactionManager），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  For transaction management, Spring provides an interface (PlatformTransactionManager), which represents the transaction manager. This interface provides different implementation classes for different frameworks. |

事务操作（注解声明式事务管理）Transaction operations (annotated declarative transaction management)

|  |
| --- |
| 1、在Spring配置文件中配置事务管理器 Configure the transaction manager in the Spring configuration file  *<!--创建事务管理器 create transaction manager-->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!--注入数据源 Inject datasource-->* <**property name="dataSource" ref="dataSource"**/> </**bean**>  2、在Spring配置文件，开启事务注解 Enable transaction annotations in the Spring configuration file  *<?***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: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"**>  *<!--开启事务注解 enable transaction annotation-->* <**tx:annotation-driven transaction-manager="transactionManager"**></**tx:annotation-driven**>  3、在service类上面（或者service类里面方法上面）添加事务注解 Add transaction annotations on the service class (or above the method in the service class)  （1）@ Transactional，这个注解添加到类上面，也可以添加方法上面 This annotation can be added to both classes and methods  （2）如果把这个注解添加类上面，这个类里面所有方法都添加事务 If you add this annotation to the class, all methods in this class will add transactions  （3）如果把这个注解添加方法上面，为这个方法添加事务 If you add this annotation to a method, add a transaction to this method  @Service @Transactional **public class** UserService {  **4、Testing**  继续运行上述错误代码，则事务会回滚，观察数据结果。Continue to run the above error code, the transaction will be rolled back, observe the data results.  **public class** TestAccount {  @Test  **public void** testAccountAnnotation() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean1.xml"**);  UserService userService = context.getBean(**"userService"**, UserService.**class**);  userService.accountMoney();  }  } |

事务操作（声明式事务管理参数配置）Transaction operations (Declarative transaction management parameter configuration)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1、在service类上面添加注解@Transactional，在这个注解里面可以配置事务相关参数  Add the annotation @Transactional to the service class, in which you can configure transaction-related parameters    2、propagation：事务传播行为 Transaction propagation behavior  事务传播行为指的就是当一个事务方法（当前事务）被另一个事务方法（调用者）调用时，这个事务方法（当前事务）对另一个事务方法（调用者）的态度。  Transaction propagation behavior refers to the attitude of a transaction method (current transaction) to another transaction method (caller) when it is called by another transaction method (current transaction).  简单说就是：事务方法A调用了事务方法B，B需要怎么应对。  To put it simply: transaction method A calls transaction method B, and how B needs to respond.    Spring框架事务传播行为有7中，其中以下两种最为常用。  There are 7 types of Spring Framework transaction propagation behaviors, of which the following two are the most commonly used.  **REQUIRED**   * 如果add方法本身有事务，调用update方法之后，update使用当前add方法里面的事务   If the add method itself has a transaction, after calling the update method, the update uses the transaction in the current add method   * 如果add方法本身没有事务，调用update方法之后，创建新事务   If the add method itself has no transaction, after calling the update method, create a new transaction  **REQUIRED\_NEW**  使用add方法调用update方法，无论add方法是否有事务，都创建新的事务  Use the add method to call the update method, regardless of whether the add method has a transaction, create a new transaction   |  |  | | --- | --- | | 事务传播行为  transaction propagation behaviors | 说明  Description | | **REQUIRED（default）** | 表示当前方法必须在事务中运行。  如果调用者有事务，则当前方法加入到调用者事务中运行。  如果调用者没有事务，则当前方法自己新开启一个事务运行。 | | **REQUIRES\_NEW** | 表示当前方法必须在事务中运行。  如果调用者有事务，则当前方法自己新开启一个事务运行。  如果调用者没有事务，则当前方法当前方法自己新开启一个事务运行。 | | SUPPORTS | 表示当前方法不必在事务中运行。  如果调用者有事务，则当前方法加入到调用者事务中运行。  如果调用者没有事务，则当前方法以非事务的形式运行。 | | MANDATORY | 表示当前方法必须在调用者事务中运行。  如果调用者有事务，则当前方法加入到调用者事务中运行。  如果调用者没有事务，则当前方法抛出异常。 | | NESTED | 表示当前方法必须在事务中运行。  如果调用者有事务，则当前方法以“嵌套事务”的形式加入到调用者事务中运行。  如果调用者没有事务，则当前方法自己新开启一个事务运行。 | | NEVER | 表示调用者必须以非事务形式运行。  如果调用者有事务，则抛出异常。  如果调用者没有事务，则当前方法以非事务的形式运行。 | | NOT\_SUPPORTED | 表示当前方法不支持在事务中运行。  如果调用者有事务，则挂起调用者的事务，当前方法以非事务的形式运行。  如果调用者没有事务，则当前方法以非事务的形式运行。 |   @Service @Transactional(propagation = Propagation.***REQUIRED***) **public class** UserService {  3、isolation：事务隔离级别 Transaction isolation level  当两个事务对同一个数据库的记录进行操作时，那么，他们之间的影响是怎么样的呢?这就出现了事务隔离级别的概念。数据库的隔离性与并发控制有很大关系。数据库的隔离级别是数据库的事务特性ACID的一部分，ACID，即原子性(atomicity)、一致性(consistency)、隔离性(isolation)和持久性(durability)。  When two transactions operate on the records of the same database, what is the impact between them? This is where the concept of transaction isolation level emerges. The isolation of the database has a lot to do with concurrency control. The isolation level of the database is part of the transaction characteristics of the database ACID.  （1）当两个事务对同一个数据库的记录进行操作时，有三个读的问题：When two transactions operate on records in the same database, there are three read problems  ① 脏读 dirty read  A事务对一条记录进行修改，尚未提交，B事务已经看到了A的修改结果。若A发生回滚，B读到的数据就是错误的，这就是脏读。  Transaction A modifies a record, but it has not yet been committed, and transaction B has seen the modification result of A. If A rolls back, the data read by B is wrong, which is a dirty read.  ② 不可重复读 non-repeatable read  A事务对一条记录进行修改，尚未提交，B事务第一次查询该记录，看到的是修改之后的结果，此时A发生回滚，B事务又一次查询该记录，看到的是回滚后的结果。同一个事务内，B两次查询结果不一致，这就是不可重复读。  Transaction A modifies a record and has not yet submitted it. Transaction B queries the record for the first time and sees the result after the modification. At this time, A rollback occurs, and transaction B queries the record again, and sees the rollback the result after. Within the same transaction, B's two query results are inconsistent, which is non-repeatable read.  ③ 幻读 phantom reading  A事务对所有记录进行修改，尚未提交，此时B事务创建了一条新记录，A、B都提交。A查看所有数据，发现有一条数据没有被修改，因为这是B事务新增的，就想看到了幻象一样，这就是幻读。  Transaction A modifies all records and has not yet submitted them. At this time, transaction B creates a new record, and both A and B are submitted. A looks at all the data and finds that one piece of data has not been modified, because it is newly added by transaction B, and it wants to see an illusion. This is a phantom read.  （2）Spring的事务隔离级别有四个：READ\_UNCOMMITTED、READ\_COMMITTED、REPEATABLE\_READ和SERIALIZABLE。MySQL默认是REPEATABLE\_READ。  Spring has four transaction isolation levels: READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, and SERIALIZABLE. MySQL defaults to REPEATABLE\_READ.  ① READ\_UNCOMMITTED  READ\_UNCOMMITTED：一个事务可以读取到另一个事务未提交的事务记录。A transaction can read the data that is still uncommitted by other transactions。  这是Spring事务最弱的隔离级别。见下面的图，事务A开启，写入一条记录，这时候，事务B读入数据，读到了这条记录，但是，之后事务A回滚。因此，事务B读到的数据不是有效的(the database is in an invalid state)。这种情况称为脏读(dirty read)。除了脏读的问题，READ\_UNCOMMITTED还可能出现non-repeatable read(不可重复读)和phantom read(幻读)的问题。  This is the weakest isolation level for Spring transactions. See the figure below, transaction A opens and writes a record. At this time, transaction B reads data and reads this record, but then transaction A rolls back. Therefore, the data read by transaction B is not valid (the database is in an invalid state). This situation is called a dirty read. In addition to the problem of dirty reads, READ\_UNCOMMITTED may also have problems with non-repeatable read (non-repeatable read) and phantom read (phantom read).    ② READ\_COMMITTED  READ\_COMMITTED：一个事务只能读取到已经提交的记录，不能读取到未提交的记录。A transaction can only read the committed data, and it can’t read the uncommitted data.  因此，dirty read的情况不再发生，但可能会出现其他问题。见下图。  So the dirty read situation no longer occurs, but other problems may arise. See below.    在事务A两次读取的过程之间，事务B修改了那条记录并进行提交。因此，事务A前后两次读取的记录不一致。这个问题称为non-repeatable read(不可重复读)。(两次读取的记录不一致，重复读取就会发现问题。)  Between transaction A's two reads, transaction B modifies that record and commits. Therefore, the records read before and after transaction A are inconsistent. This problem is called non-repeatable read (non-repeatable read). (The records read twice are inconsistent, and repeated reading will find problems.)  除了non-repeatable read的问题，READ\_COMMITTED还可能发生phantom read的问题。  In addition to the problem of non-repeatable read, the problem of phantom read may also occur with READ\_COMMITTED  ③ REPEATABLE\_READ  REPEATABLE\_READ：一个事务可以多次从数据库读取某条记录，而且多次读取的那条记录都是一致的，相同的。  A transaction can read a record from the database multiple times, and the record read multiple times is consistent and the same.  这个隔离级别可以避免dirty read和non-repeatable read的问题，但可能发生phantom read的问题。如下图。  This isolation level can avoid the problem of dirty read and non-repeatable read, but the problem of phantom read may occur. As shown below.    事务A两次从数据库读取一系列记录，期间，事务B插入了某条记录并提交。事务A第二次读取时，会读取到事务B刚刚插入的那条记录。在事务期间，事务A两次读取的一系列记录不一致，这个问题称为phantom read。  Transaction A reads a series of records from the database twice, during which transaction B inserts a record and commits. When transaction A reads for the second time, it will read the record just inserted by transaction B. During a transaction, a series of records read twice by transaction A are inconsistent, this problem is called phantom read.  ④ SERIALIZABLE  SERIALIZABLE是Spring最强的隔离级别。事务执行时，会在所有级别上加锁，比如read和write时都会加锁，仿佛事务是以串行的方式进行的，而不是一起发生的。这会防止dirty read、non-repeatable read和phantom read的出现，但是，会带来性能的下降。  SERIALIZABLE is Spring's strongest isolation level. When a transaction is executed, locks are applied at all levels, such as read and write, as if the transactions were performed in a serial fashion, rather than all at once. This will prevent dirty reads, non-repeatable reads, and phantom reads from occurring, however, it will bring performance degradation.  （3）通过设置事务隔离性，解决读问题 Solve read problems by setting transaction isolation  @Service @Transactional(propagation = Propagation.***REQUIRED***, isolation = Isolation.***REPEATABLE\_READ***) **public class** UserService {  4、timeout：超时时间  （1）事务需要在一定时间内进行提交，如果不提交，事务就会回滚 The transaction needs to be committed within a certain period of time. If it is not committed, the transaction will be rolled back  （2）默认值是-1，设置时间以秒为单位进行计算 The default value is -1, the set time is calculated in seconds  5、readOnly：是否只读  （1）读：查询操作，写：添加删除修改操作 Read: query operations, write: add, delete, modify operations  （2）readOnly默认值false，表示可以查询，可以添加删除修改操作 The default value of readOnly is false, which means that you can query, add, delete, and modify operations.  （3）readOnly设置成true之后，只能查询 After readOnly is set to true, you can only query  6、rollbackFor：回滚  设置出现哪些异常进行事务回滚 Set which exceptions occur for transaction rollback  7、noRollbackFor：不回滚  设置出现哪些异常不进行事务回滚 Set which exceptions do not perform transaction rollback |

事务操作（xml声明式事务管理）Transaction operations (xml declarative transaction management)

|  |
| --- |
| 1、在Spring配置文件中进行配置 Configure in Spring configuration file bean2.xml  第一步 配置事务管理器 the first step Configure Transaction Manager  第二步 配置通知 the second step Configure advice  第三步 配置切入点和切面 the third step Configure pointcuts and aspects  *<!--1 创建事务管理器 create transaction manager-->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!--注入数据源 inject datasource-->* <**property name="dataSource" ref="dataSource"**/> </**bean**>  *<!--2 配置通知 Configure advice-->* <**tx:advice id="txAdvice"**>  *<!--配置事务参数 Configure transaction parameters -->* <**tx:attributes**>  *<!--指定哪种规则的方法上面添加事务 Specifies which rule's method to add the transaction to -->* <**tx:method name="accountMoney" propagation="REQUIRED"**/>  *<!--<tx:method name="account\*"/>-->* </**tx:attributes**> </**tx:advice**>  *<!--3 配置切入点和切面 Configure pointcuts and aspects -->* <**aop:config**>  *<!--配置切入点 Configure pointcuts -->* <**aop:pointcut id="point" expression="execution(\* com.qixin.service.UserService.\*(..))"**/>  *<!—配置切面 Configure aspects -->* <**aop:advisor advice-ref="txAdvice" pointcut-ref="point"**/> </**aop:config**>  **2、Testing**  @Test **public void** testAccountXml() {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean2.xml"**);  UserService userService = context.getBean(**"userService"**, UserService.**class**);  userService.accountMoney(); } |

事务操作（完全注解声明式事务管理）Transaction operations (Fully annotated declarative transaction management)

|  |
| --- |
| 1、创建配置类，使用配置类替代xml配置文件 Create a configuration class and use the configuration class to replace the xml configuration file  **package** com.qixin.config;  @Configuration *//配置类 configuration class* @ComponentScan(basePackages = **"com.qixin"**) *//组件扫描 component scan* @EnableTransactionManagement *//开启事务 enable transaction* **public class** TxConfig {   *//创建数据库连接池 Create a database connection pool* @Bean  **public** DruidDataSource getDruidDataSource() {  DruidDataSource dataSource = **new** DruidDataSource();  dataSource.setDriverClassName(**"com.mysql.cj.jdbc.Driver"**);  dataSource.setUrl(**"jdbc:mysql://localhost:3306/user\_db?characterEncoding=UTF-8&serverTimezone=Asia/Shanghai"**);  dataSource.setUsername(**"root"**);  dataSource.setPassword(**"1234"**);  **return** dataSource;  }   *//创建JdbcTemplate对象 create JdbcTemplate object* @Bean  **public** JdbcTemplate getJdbcTemplate(DataSource dataSource) {  JdbcTemplate jdbcTemplate = **new** JdbcTemplate();  *//到IOC容器中根据类型找到dataSource进行注入 Find the dataSource according to the type in the IOC container for injection* jdbcTemplate.setDataSource(dataSource);  **return** jdbcTemplate;  }   *//创建事务管理器 Create transaction manager* @Bean  **public** DataSourceTransactionManager getDataSourceTransactionManager(DataSource dataSource) {  DataSourceTransactionManager transactionManager = **new** DataSourceTransactionManager();  transactionManager.setDataSource(dataSource);  **return** transactionManager;  } }  2、Testing  @Test **public void** testAccountAnnotation2() {  ApplicationContext context = **new** AnnotationConfigApplicationContext(TxConfig.**class**);  UserService userService = context.getBean(**"userService"**, UserService.**class**);  userService.accountMoney(); } |

计算机专业综合实训 Computer professional comprehensive training 15-19周