1. Spring基本概念：
2. spring是轻量级的开源的javaEE框架
3. Spring可以解决企业应用开发的复杂性
4. Spring有两个核心部分：Ioc，Aop
5. Ioc 控制反转，把创建对象的过程交给spring来完成
6. Aop面向切面，不修改源代码进行功能的增强
7. Spring的特点：
8. 方便解耦，简化开发：降低程序的耦合性

注：耦合性(Coupling)，也叫耦合度，是对模块间关联程度的度量。耦合的强弱取决于模块间接口的复杂性、调用模块的方式以及通过界面传送数据的多少。模块间的耦合度是指模块之间的依赖关系，包括控制关系、调用关系、数据传递关系。模块间联系越多，其耦合性越强，同时表明其独立性越差( 降低耦合性，可以提高其独立性)。耦合性存在于各个领域，而非软件设计中独有的，但是我们只讨论软件工程中的耦合。 在软件工程中，耦合指的就是就是对象之间的依赖性。对象之间的耦合越高，维护成本越高。因此对象的设计应使类和构件之间的耦合最小。软件设计中通常用耦合度和内聚度作为衡量模块独立程度的标准。划分模块的一个准则就是高内聚低耦合。

1. Aop编程支持
2. 方便程序的测试
3. 方便和其他框架进行整合
4. 方便进行事务操作
5. 降低API开发难度

入门案例：

1.准备（下载Spring5）

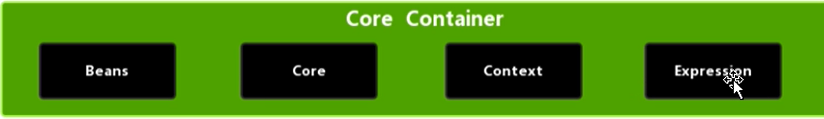
选择5.2.6 GA版本即稳定版本

下载地址：

<https://repo.spring.io/artifactory/release/org/springframework/spring/>

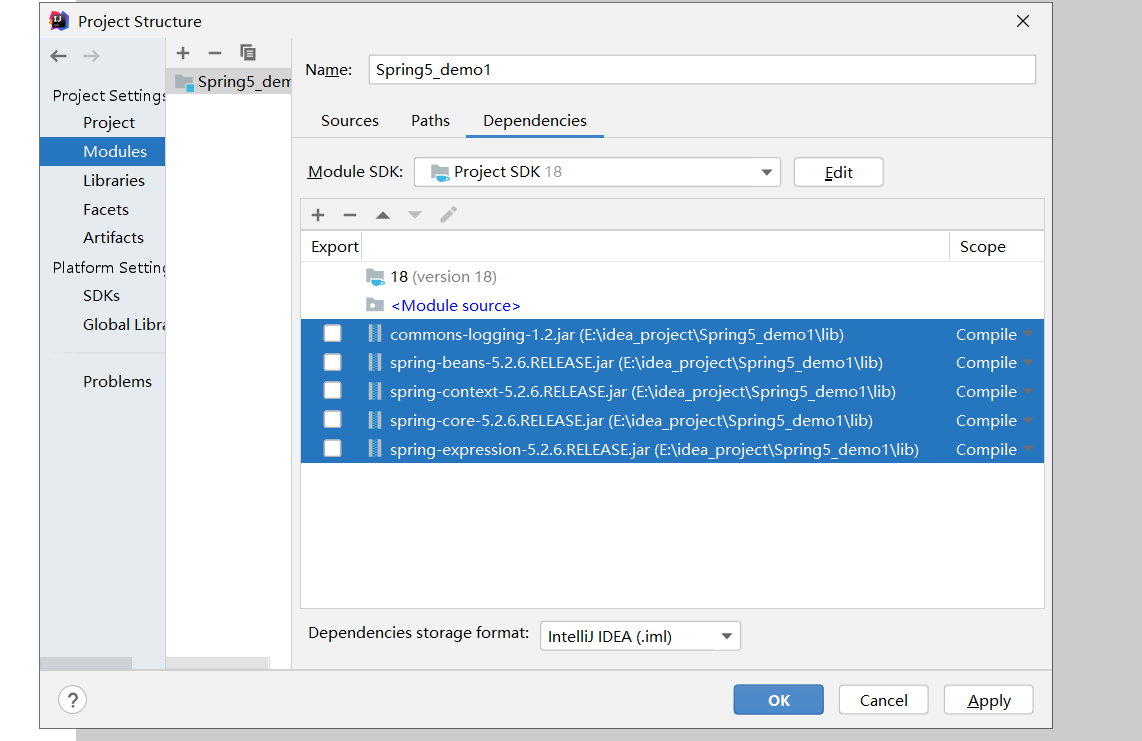
1. 新建一个普通的java工程
2. 导入Spring的相关jar包

基础部分需要的jar包如下：外加一个日志jar包



将这些jar包在新建的lib文件夹下粘贴，然后点开project Strecture 的Modules：

Dependencies 的加号，添加导入的这五个jar包：



1. 创建一个类，在类里创建一个方法

public class User {

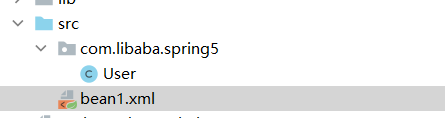
public void add(){

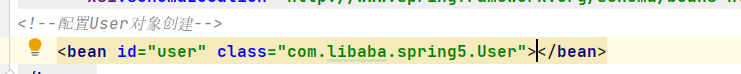
System.out.println("add……");

}

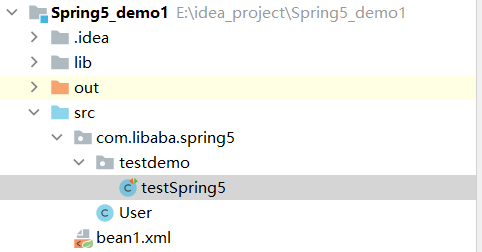
}

1. 创建Spring的配置文件，在配置文件中配置创建的对象
2. spring中配置文件的格式是XML，注意，XML配置文件必须写在src目录下，不然会出现错误





1. 进行代码测试



import com.libaba.spring5.User;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class testSpring5 {

@Test

public void testAdd(){

//加载配置文件

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

//获取配置创建的对象

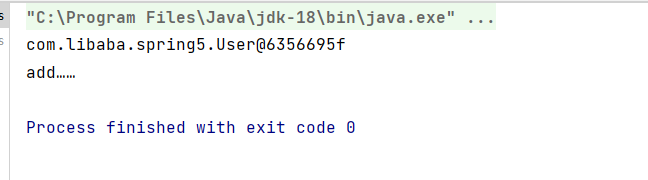
User user=context.getBean("user", User.class);

System.out.println(user);

user.add();

}

}



1. IOC容器
2. IOC底层原理

1）什么是IOC：控制反转，把对象的创建和对象之间的调用，交给spring进行管理，能够降低代码之间的耦合度

通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体将其所依赖的对象的引用传递给它。也可以说，依赖被注入到对象中。

1. IOC底层原理：

Xml解析+反射+工厂模式

反射：得到类的字节码文件

首先工厂模式举例：

//普通方法

**class** User1{

**public** **void** test1(){

User2 user2=**new** User2();

user2.add();

}

}

**class** User2{

**public** **void** add(){

System.***out***.println("test2");

}

}

//工厂模式

**class** User1{

**public** **void** test1(){

User2 user2=UserFactory.testuser2();

user2.add();

}

}

**class** User2{

**public** **void** add(){

System.out.println("test2");

}

}

**class** UserFactory{

**public** **static** **User2** testuser2() {

**return** **new** User2();

}

}

由上可得，工厂模式并不能把耦合降到最低，所以配合xml+反射

过程：

1.Xml配置文件，配置创建的对象

<**bean id="user" class="com.libaba.spring5.User"**></**bean**>

1. 创建工厂类(结合前边的三个类，前边的User1，User2类不变，更新UserFactory类4)

Class UserFactory{

public static User2 testuser2(){

String classValue=class属性值 ;//xml解析得到上面的User1类路径

Class clazz=Class.forName(classValue);//反射得到字节码文件，就是将前边那个java文件转为class文件

return (User2)clazz.newInstance();//该方法创建对象

}

}

上边进一步降低了耦合

1. IOC接口（BeanFactory）

1）IOC思想基于IOC容器完成，IOC容器底层就是对象工厂

2）Spring提供了IOC容器实现的两种方式：（两个接口）

BeanFactory：

IOC容器基本实现，是Spring内部的使用接口，不提供给开发人员使用

\*加载配置文件时不会创建对象，在获取对象或使用时才去创建对象

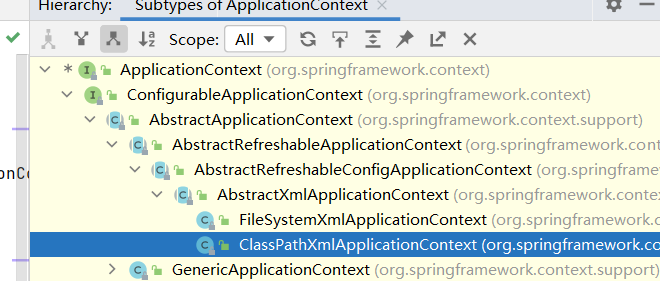
ApplicationContext：

BeanFactory接口的子接口，提供更多更加强大的功能，一般有开发人员使用

\*加载配置文件时就会把配置文件中的对象进行创建

1. ApplicationContext接口的实现类：

对ApplicationContext ctrl+H，得到他的方法：



FileSystemXmlApplicationContext:你的磁盘里的路径

1. IOC操作Bean管理（基于xml，基于注解方式）

什么是Bean管理：两个操作，如下

1）Spring创建对象

2）Spring注入属性（参数）

Bean管理操作有两种实现方式：

1. 基于xml配置文件方式实现

基于xml方式创建对象：



在Spring配置文件中使用bean标签，标签里面设置属性

下面介绍一下bean标签里面的属性：

\*id属性 ：唯一标识

\*class属性：创建对象的类的全路径（包加类）

\*name属性：与id差不多，但id中不能加特殊符号

创建对象时，也是默认实现无参的构造方法

基于xml方式注入属性：

DI：依赖注入，就是注入属性/传入参数

两种注入方式：

使用set方法和使用有参的构造方法：

普通实现：

public class Book {

private String bname;

//set方式注入

public void setBname(String bname) {

this.bname = bname;

}

//有参构造方式注入

public Book(String bname){

this.bname = bname;

}

public static void main(String[] args){

Book book=new Book("abc");

book.setBname("abc");

}

}

**使用set方式，在配置文件中配置对象创建，配置属性注入：**

**第一步：**

**//使用set方式进行注入属性**

**public class Book {**

**//创建属性**

**private String bauthor;**

**private String bname;**

**//创建属性对应的set方法**

**public void setBauthor(String bauthor) {**

**this.bauthor = bauthor;**

**}**

**public void setBname(String bname) {**

**this.bname = bname;**

**}**

**public void testDemo(){**

**System.out.println(bname+"::"+bauthor);**

**}**

**}**

**第二步：**

**<!--set方式注入属性-->**

**<bean id="book" class="com.libaba.spring5.Book">**

**<!--使用property标签完成属性注入**

**name:类里面属性的值**

**value：向属性里注入的值**

**-->**

**<property name="bname" value="易筋经"></property>**

**<property name="bauthor" value="少林和尚"></property>**

**</bean>**

**</beans>**

**第三步：**

**写测试类得到结果：**



**使用有参的构造方法，通过配置文件，配置属性注入：**

**第一步：**

**//使用有参构造方式**

**public class Orders {**

**//创建属性**

**private String oname;**

**private String address;**

**//创建有参构造方法，注入属性**

**public Orders(String oname,String address){**

**this.oname=oname;**

**this.address=address;**

**}**

**public void ordersTest(){**

**System.out.println(oname+"::"+address);**

**}**

**}**

**第二步：**

**<!--有参构造注入属性-->**

**<bean id="orders" class="com.libaba.spring5.Orders">**

**<!--下面这个标签是可以调用有参的构造方法，默认是无参的**

**name:类中的属性**

**value：注入的值**

**index="0" ,标识注入的第一个属性**

**-->**

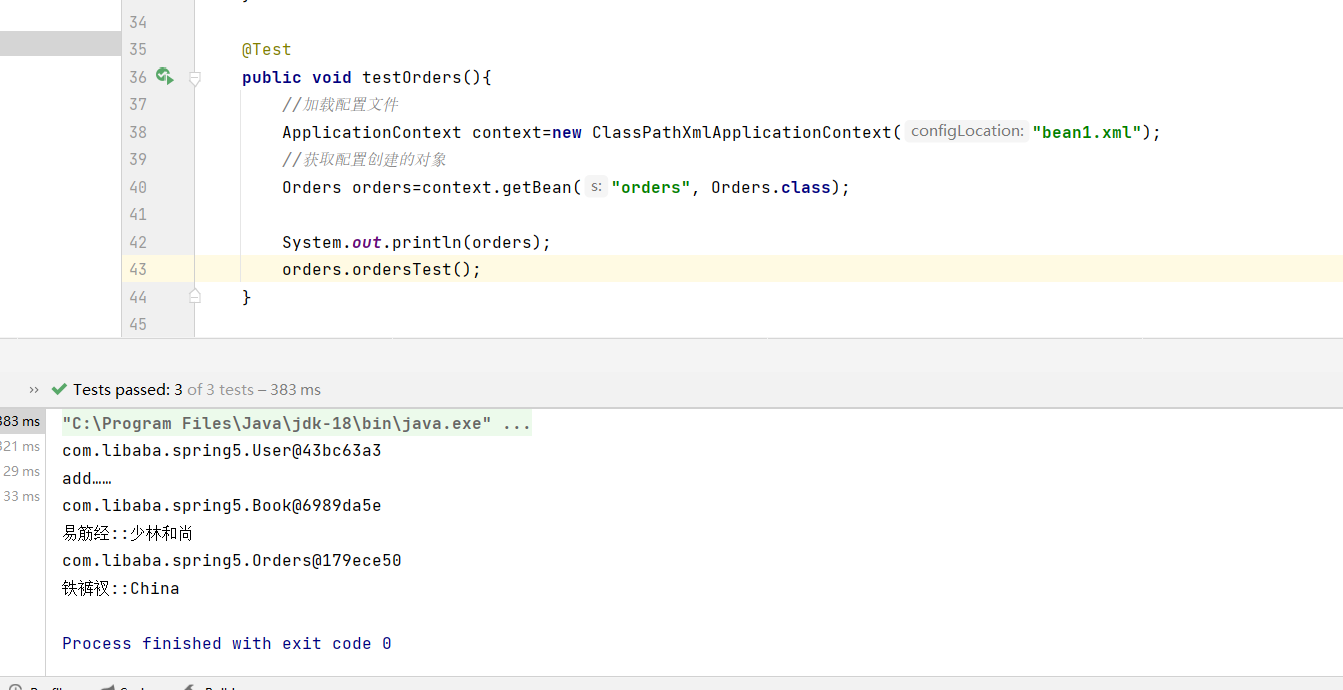
**<constructor-arg name="oname" value="铁裤衩"></constructor-arg>**

**<constructor-arg name="address" value="China"></constructor-arg>**

**</bean>**

**第三步：**

写测试函数后：

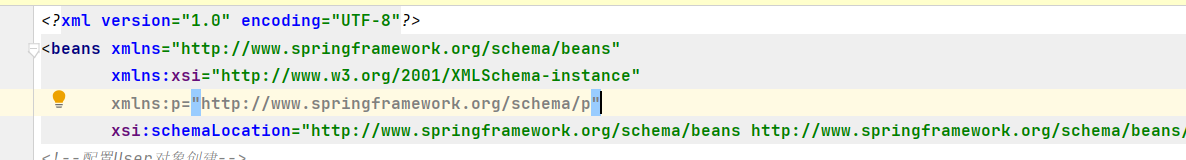


**P名称空间注入：**

**就是可以简化基于xml的配置方式，了解**

**步骤如下：**

**第一步：添加p名称空间在配置文件头**



**第二步：**

**进行属性注入，不需要其他标签，直接在bean标签里写：**

**<bean id="book" class="com.libaba.spring5.Book" p:bname="降龙十八掌" p:bauthor="狗剩子">**

Xml注入其他属性：

1. 字面量：（参数）
2. null值

<!--null值属性-->

<property name="address">

<null/>

</property>

1. 属性值包含特殊符号

<!--特殊符号 -->

<property name="address">

<value><![CDATA[<<南京>>]]></value>

</property>

1. 注入外部bean

1. 创建两个类service，dao
2. 在service中调用dao类

//创建UserDao类型属性，生成set方法

private UserDao userDao;

public void setUserDao(UserDao userDao){

this.userDao=userDao;

}

1. 在spring配置文件里进行配置

<!--service和Dao对象创建-->

<bean id="userService" class="com.libaba.spring5.service.UserService">

<!-- 注入userDao对象

name 属性值，类里面的属性名称

ref属性：创建userDao对象bean标签id值,就是下面那个标签的id值

-->

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

</bean>

<bean id="userDaoImpl" class="com.libaba.spring5.dao.UserDaoImpl"></bean>

写测试类：

测试函数进行测试。

1. 注入内部bean和级联赋值
2. 一对多的关系，举例实现内部bean

部门和员工，一对多关系

写两个实体类：

public class Emp {

private String ename;

private String gender;

//员工属于某一个部门,使用对象形式表示

private Dept dept;

public void setDept(Dept dept) {

this.dept = dept;

}

public void setEname(String ename) {

this.ename = ename;

}

public void setGender(String gender) {

this.gender = gender;

}

}

public class Dept {//部门类

private String dname;

public void setDname(String dname) {

this.dname = dname;

}

}

然后在spring配置文件中进行配置：

<!--内部bean-->

<bean id="emp" class="com.libaba.spring5.bean.Emp">

<!--先设置两个普通的属性-->

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

<property name="gender" value="女"></property>

<!-- 设置对象类型属性-->

<property name="dept">

<bean id="dept" class="com.libaba.spring5.bean.Dept">

<property name="dname" value="保安部"></property>

</bean>

</property>

</bean>

写测试方法：

、、、、、、、、、、、、、、

1. 注入属性——级联赋值

写法一：

<!--级联赋值-->

<bean id="emp" class="com.libaba.spring5.bean.Emp">

<!--先设置两个普通的属性-->

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

<property name="gender" value="女"></property>

<!-- 级联赋值 -->

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

</bean>

<bean id="dept" class="com.libaba.spring5.bean.Dept">

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

</bean>

写法二：

注意，必须写Dept的get方法，不然会报错

<!--级联赋值-->

<bean id="emp" class="com.libaba.spring5.bean.Emp">

<!--先设置两个普通的属性-->

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

<property name="gender" value="女"></property>

<!-- 级联赋值 -->

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

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

</bean>

<bean id="dept" class="com.libaba.spring5.bean.Dept">

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

</bean>

4.注入集合属性

前边注入了String类型，对象类型

1. 注入数组：
2. 注入List集合类型属性
3. 注入Map集合类型属性
4. 注入Set集合类型属性

先在类中定义各个属性：

public class Stu {

//数组类型的属性

private String[] courses;

//创建list集合类型属性

private List<String> list;

//创建map集合类型属性

private Map<String,String> maps;

//创建set集合类型属性

private Set<String> sets;

public void setSets(Set<String> sets) {

this.sets = sets;

}

public void setList(List<String> list) {

this.list = list;

}

public void setMaps(Map<String, String> maps) {

this.maps = maps;

}

public void setCourses(String[] courses) {

this.courses = courses;

}

}

然后在Spring配置文件中进行配置：

<!--集合类型对象配置-->

<bean id="stu" class="com.libaba.spring5.collectiontype.Stu">

<!-- 数组类型属性配置-->

<property name="courses">

<array>

<value>Java课程</value>

<value>英语四级</value>

<value>自动控制原理</value>

</array>

</property>

<!-- List类型属性配置-->

<property name="list">

<list>

<value>张三</value>

<value>李四</value>

</list>

</property>

<!-- Map类型属性配置-->

<property name="maps">

<map>

<entry key="JAVA" value="java"></entry>

<entry key="JPH" value="jph"></entry>

</map>

</property>

<!-- Set类型属性配置-->

<property name="sets>">

<set>

<value>Mysql</value>

<value>libaba</value>

</set>

</property>

</bean>

然后进行测试。。。

1. 在集合中注入对象类型的值

新建一个类Course

在Stu类里面创建一个集合List，其中的内容是Course类的对象

然后写List集合属性的set方法

配置Spring配置文件：

<!-- 注入list对象类型-->

<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.libaba.spring5.collectiontype.test2.Course">

<property name="cname" value="Spring5框架"></property>

</bean>

<bean id="course2" class="com.libaba.spring5.collectiontype.test2.Course">

<property name="cname" value="Mabatis框架"></property>

</bean>

然后写测试类进行测试。。。。

1. 把集合注入部分提取出来，作为公共部分

在Spring配置文件中引入名称空间util（复制修改就行）

<?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: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">

使用util标签完成

配置文件注入：

使用util标签来将list集合的属性值进行抽取，使其成为公共部分，可以在后边的任何对象创建时使用：：

<!--1.提取list集合类型属性的注入-->

<util:list id="bookList">

<!-- 引入对象用ref-->

<value>九阳</value>

<value>九阴</value>

<value>大刀</value>

</util:list>

<!--2.提取list集合类型属性注入使用-->

<bean id="book" class="com.libaba.spring5.collectiontype.test2.Book">

<property name="list" ref="bookList">

</property>

</bean>

IOC操作Bean管理（FactoryBean）

Spring中的两种Bean，一种普通Bean，另外一种工厂Bean

普通Bean就是自己定义的Bean，定义的是什么类型，返回的就是定义的类型

工厂Bean，配置文件中定义的bean和返回的类型可以不一样

做法：

创建类，让该类作为工厂bean，实现接口FactorBean

实现接口里面的方法，在实现的方法中返回debean类型

public class Mybean implements FactoryBean<Course> {

//定义返回的bean

@Override

public Course getObject() throws Exception {

Course course=new Course();

course.setCname("abc");

return course;

}

@Override

public Class<?> getObjectType() {

return null;

}

@Override

public boolean isSingleton() {

return FactoryBean.super.isSingleton();

}

IOC操作Bean管理（bean的作用域）

在Spring中设置bean实例是单实例还是多实例（获取多次对象实例，看他的地址是否相同）

在默认情况下，bean是单实例

设置单实例还是多实例对象（用标签里的scope属性设置）

scope属性值讲解：

singleton, 单实例，默认值

prototype 多实例

request,session



二者的区别，singleton在加载配置文件时创建单实例对象，prototype在调用getBean方法的时候才创建多实例对象

IOC操作Bean管理（bean的生命周期）

1. 生命周期指从生成到销毁

1. bean生命周期：步骤如下

通过构造器创建bean实例（无参构造）

为bean的属性设置值和对其他bean引用（调用set方法）

调用bean的初始化方法

Bean可以使用了

容器关闭时，调用bean的销毁方法，（需要配置销毁方法）

public class Orders {

private String oname;

//无参构造

public Orders(){

System.out.println("第一步，执行无参构造创建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 执行初始化方法");

}

//创建执行的销毁方法

public void destroyMethod(){

System.out.println("第五步 执行销毁方法");

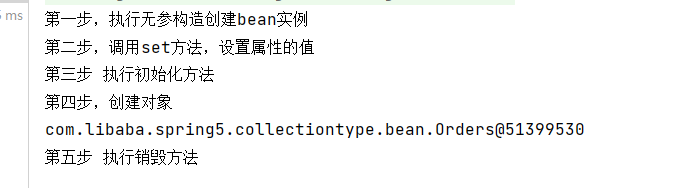
}

}

<bean id="orders" class="com.libaba.spring5.collectiontype.bean.Orders" init-method="initMethod" destroy-method="destroyMethod">

<property name="oname" value="手机"></property>

</bean>



Bean的后置处理器加上后，七部操作：

通过构造器创建bean实例（无参构造）

为bean的属性设置值和对其他bean引用（调用set方法）

把bean实例传递给bean后置处理器的方法postProcessBeforeInitialization

调用bean的初始化方法

把bean实例传递给bean后置处理器的方法postProcessAfterInitialization

Bean可以使用了

容器关闭时，调用bean的销毁方法，（需要配置销毁方法）

创建一个类，实现BeanPostProcessor接口，实现里面的方法

public class MyBeanPost implements BeanPostProcessor {

@Override

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

System.out.println("在初始化之前");

return bean;

}

@Override

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

System.out.println("在初始化之后");

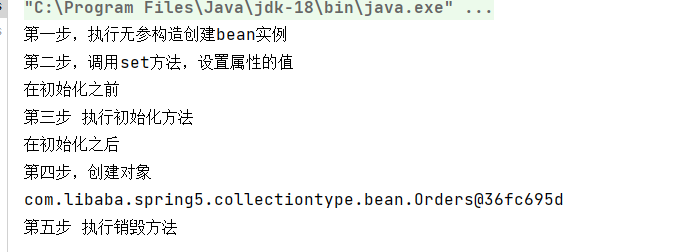
return bean;

}

}

<!--配置后置处理器-->

<bean id="mybeanPost" class="com.libaba.spring5.collectiontype.bean.MyBeanPost"></bean>



IOC操作Bean管理（自动装配）

自动装配是什么？前边的所有配置是手动装配，根据指定的装配规则（属性名称或者属性类型），Spring自动将匹配的属性值进行注入

根据属性的名称进行注入：

根据属性类型进行注入：

<!--实现自动装配

bean中的属性autowire 配置自动装配

autowire 中的值，byName，根据属性的名称进行注入， （id和类中属性的名称完全相同就能完成自动装配）

，byType 根据属性的类型注入 （bean中的类型和类中的类型相同的完成自动装配）

-->

<bean id="emp" class="com.libaba.spring5.collectiontype.autowire.Emp" autowire="byType">

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

</bean>

<bean id="dept" class="com.libaba.spring5.collectiontype.autowire.Dept"></bean>

</beans>

IOC操作Bean管理（引入外部属性文件）：以数据库连接池的配置为例子

1）直接配置数据库信息

配置德鲁伊连接池

引入德鲁伊连接池依赖

<!--直接配置连接池-->

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

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

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

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

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

</bean>

2）引入外部属性文件配置数据库连接池

创建外部属性文件，properties格式文件，写数据库信息

prop.driverClass=com.mysql.jdbc.Driver

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

username=root

password=libaba

将外部属性文件引入到spring配置文件中

引入一个context名称空间 与前边的util一样

<?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:util="http://www.springframework.org/schema/util"

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

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

在spring配置文件中使用标签引入外部属性文件：

<!--引入外部属性文件-->

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

<!-- 配置连接池

${}里面的内容是外部属性文件中的表达式左边的值

-->

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

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

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

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

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

</bean>

1. 基于注解方式实现

IOC操作Bean管理

什么是注解？

注解是代码特殊格式，@注解名称（属性名称=属性值，……）

注解可以作用在类，属性，方法上面

注解简化xml

Spring针对Bean管理中创建对象提供注解

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

\*上面四个注解功能一样，分别用在不同层上

基于注解方式实现对象创建：（会出现错误，可以尝试调节Jdk版本）

1）引入依赖Aop依赖（jar包）

2）开启组件扫描 让spring容器知道要扫描那个类，引入context名称空间，标签写扫描的包

<!--开启组件扫描

base-package后边的多个包名可以用逗号隔开

-->

<context:component-scan base-package="com.libaba.spring5.collectiontype"></context:component-scan>

1. 创建类，在类上面添加创建对象注解

@Component(value="userService") //相当于bean标签中id里的值，如果注解里面不写value的值，默认为类名首字母小写

public class UserService {

public void add(){

System.out.println("service test1111");

}

}

4）创建测试类进行测试

注意：开启组件扫描中的细节问题

举例如下：

<!--实例1

自己写扫描的规则，只扫描带Controller注解的类

-->

<context:component-scan base-package="com.libaba.spring5.collectiontype" use-default-filters="false">

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

</context:component-scan>

<!--实例2

context:exclude-filter:设置不去扫描的内容

-->

<context:component-scan base-package="com.libaba.spring5.collectiontype" >

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

</context:component-scan>

基于注解方式实现属性的注入：

1. @AutoWired:根据属性类型进行自动装配

把service和dao对象创建，类上添加创建对象的注解

service中注入dao对象，在service类中添加dao类型属性，在属性上使用注解，不用set方法

1. @Qualifier：根据属性名称进行注入

配合AutoWired使用

@Autowired

@Qualifier(value="userDaoImpl1") //必须和Autowired一起使用，当那个接口有多个实现类时，根据类型无法完成注入，可以根据名称进行注入

上边的是注入对象类型

1. @Resource：可以根据类型注入，也可以根据名称注入

//根据类型进行注入

// @Resource

//根据名称注入

@Resource(name="UserDao1")

private UserDao userDao;

1. @Value：注入普通类型属性

@Value(value = "张三")

private String name;

完全注解方式：

1. 创建配置类，替代配置文件

@Configuration //将这个类作为配置类的注解

@ComponentScan(basePackages = {"com.libaba"}) //相当于配置文件中的扫描部分

public class SpringConfig {

}

1. 编写测试类

@Test

public void testService2(){

//纯注解实现配置类

ApplicationContext context=

new AnnotationConfigApplicationContext(SpringConfig.class);

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

System.out.println(userService);

userService.add();

}

1. AOP
2. 基本概念：

什么是AOP：

面向切面编程，通过预编译方式和运行期间动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，

AOP可以干什么：

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。

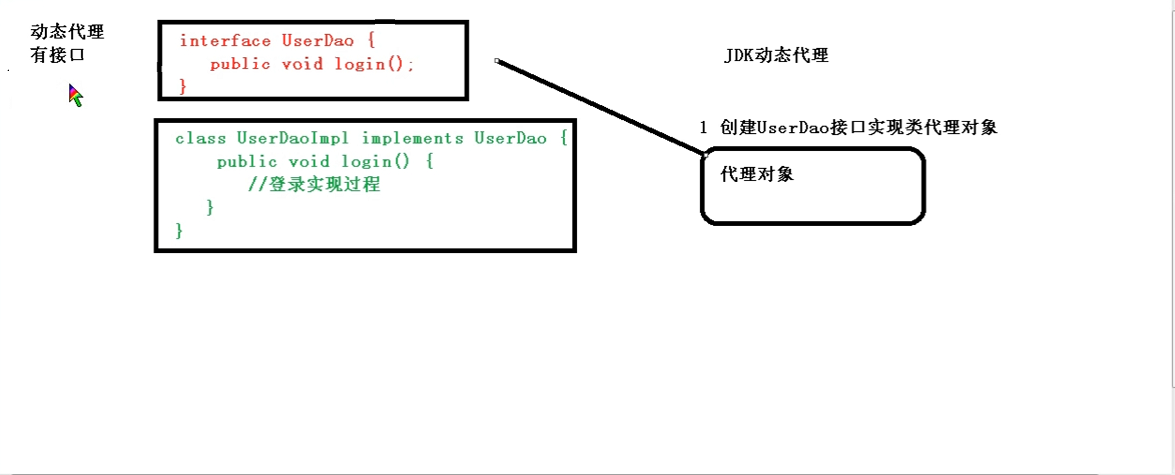
AOP也就是，将一个业务分为各个模块，如果要修改就只需添加新的模块，或者修改特定的模块，不需要修改所有的源代码

1. 底层原理
2. AOP底层使用动态代理

有两种情况的代理，

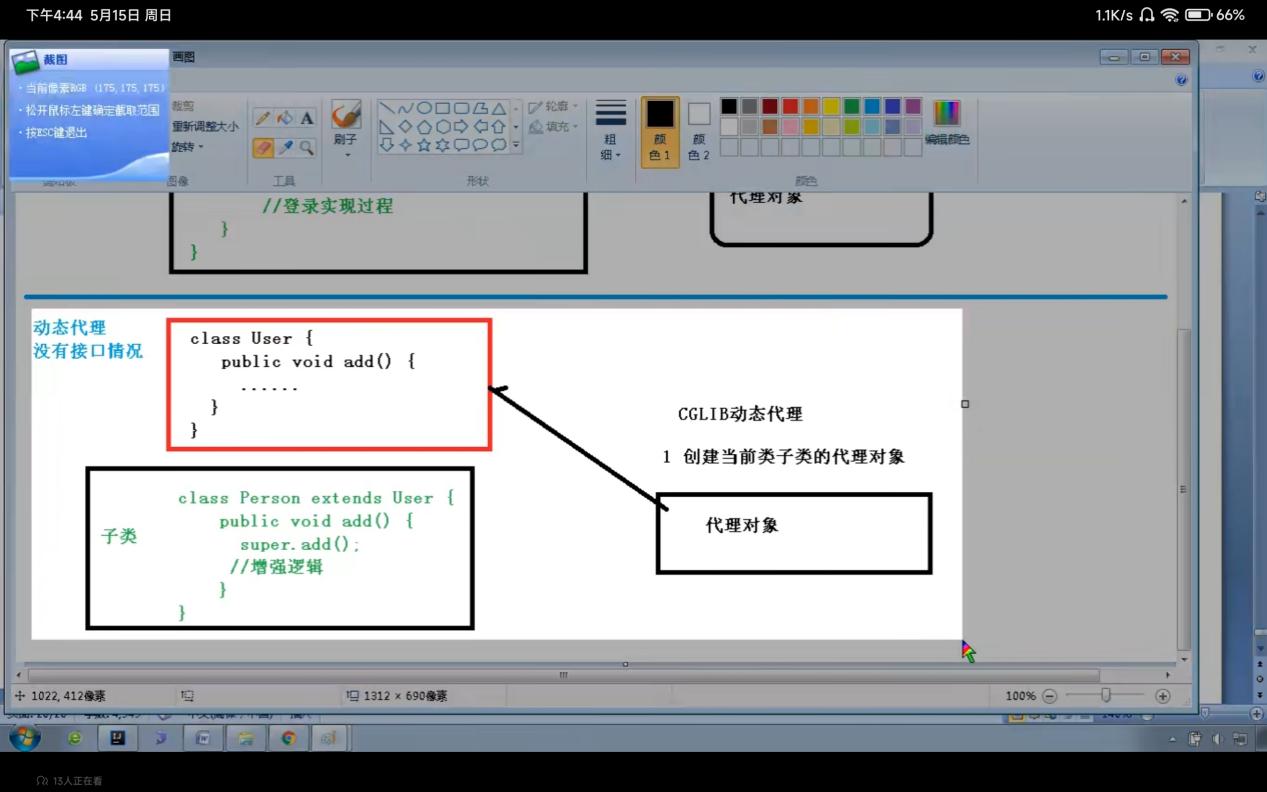
有接口情况：使用JDK动态代理

JDK动态代理：就是创建一个接口的实现类代理对象，在里面原来实现类的方法基础上新增要添加的功能



没有接口情况：使用CGLIB代理

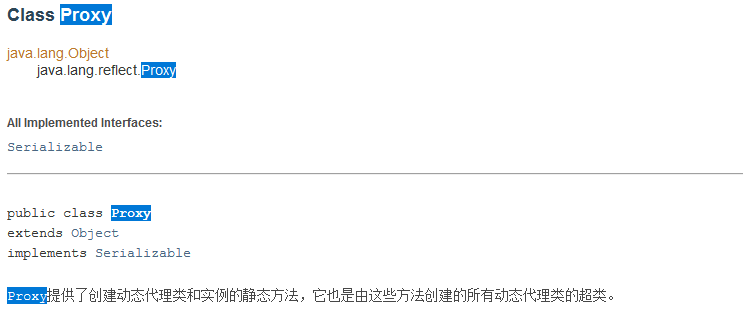
CGLIB代理，创建子类的代理对象进行增强

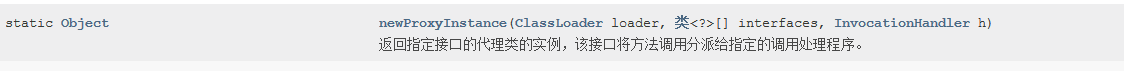


举例：使用有接口情况，JDK动态代理

1）使用java.lang包里的Proxy类的newProxyInstance方法返回接口的代理对象

该方法中的三个参数，1累加载器，2增强方法所在类实现的接口，3实现这个接口InvocationHandler，创建代理对象，写增强的方法





1. 写代码

创建接口，定义方法， 创建实现类

public interface UserDao {

public int add(int a,int b);

public String update(String id);

}

public class UserDaoImpl implements UserDao{

@Override

public int add(int a, int b) {

System.out.println("add方法执行");

return a+b;

}

@Override

public String update(String id) {

System.out.println("update方法执行");

return id;

}

}

使用Proxy类来创建接口代理对象

public class JDKProxy { //测试类

public static void main(String[] args) {

//创建接口实现类代理对象

Class[] interfaces={UserDao.class}; //interfaces就是UserDao接口，即实现类实现的接口,可以用UserDaoImpl.class.getInterfaces()代替

//这个接口InvocationHandler的实现类，用过有参构造方法，传入接口实现类的对象，返回接口的代理对象

// Proxy.newProxyInstance(JDKProxy.class.getClassLoader(), interfaces, new InvocationHandler() {

// @Override

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

// return null;

// }

// });

UserDaoImpl 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:"+result);

}

}

//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 obj;//接受接口实现类的对象

public UserDaoProxy(Object obj){

this.obj=obj;

}

//增强的逻辑

@Override

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

//方法之前

System.out.println("在方法之前执行"+method.getName()+"传递的参数："+ Arrays.toString(args));

//被增强的方法执行

Object res=method.invoke(obj,args);

//方法之后

System.out.println("在方法之后执行"+obj);

return res;

}

}

AOP原理总结：

1.首先在测试类中写

UserDao dao=(UserDao)Proxy.newProxyInstance

(JDKProxy.class.getClassLoader(),UserDaoImpl.class.getInterfaces(),new UserDaoProxy(userDao));

注意这个函数Proxy.newProxyInstance(1,2,3)中的三个参数，1表示当前类的构造器，写法是：

当前测试类名称.class.getClassLoader()... 2表示要返回的代理的接口是哪个，写法是：

接口实现类名称.class.getInterfaces()..... 3 表示一个类UserDaoProxy的对象，要向这个对象中传入的参数是接口实现类的对象

2.如上边的1.3详解，这个类要实现InvocationHandler接口，该接口内的具体写法是AOP的核心，首先必须使用有参的构造方法来获取传入的接口实现类的对象，写法为：

private Object obj;//接受接口实现类的对象

public UserDaoProxy(Object obj){

this.obj=obj;

}

然后执行要增强的方法，method.invoke(obj,args);

1. AOP术语
2. 连接点

能够被进行增强的方法就是连接点

1. 切入点

实际上真正增强了的方法就是切入点

1. 通知（增强）

实际上增强的逻辑部分就是通知

通知的类型有多种：

前置通知

在方法之前增强

后置通知

在方法之后增强

环绕通知

在方法前后都增强

异常通知

方法出现异常进行增强

最终通知

异常处理中，无论出不出现异常都增强

4.切面

是一个动作，把通知应用到切入点的过程

1. AOP操作具体实现
2. 准备

Spring框架中一般都基AspectJ实现AOP操作

AspectJ是什么：

不是Spring的部分，是独立的AOP框架，一般把AspectJ和Spring结合操作

1. 基于AspectJ实现AOP操作

基于xml配置文件实现

基于注解方式实现

1. 在项目工程中引入AOP相关依赖

4个jar包

1. 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]))

例子：对com.libaba.dao.UserDao 的add()方法进行增强

写他的切入点表达式：execution(\* com.libaba.dao.UserDao.add(..) ) //\*表示可以是任意

对com.libaba.dao.UserDao所有方法进行增强

execution(\* com.libaba.dao.UserDao.\*(..) )

对com.libaba.dao包里的所有类的所有方法进行增强

execution(\*com.libaba.dao.\*.\*(..))

1. 注解方式实现AOP
2. 创建类，定义方法
3. 创建一个增强类，写增强逻辑

在增强类里面创建方法，让不同方法代表不同通知类型

1. 进行通知配置

在Spring配置文件中，开启注解扫描

<?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: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

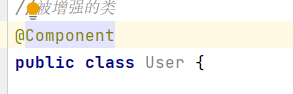
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd

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

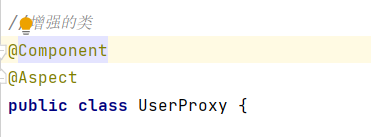
<!--开启注解扫描-->

<context:component-scan base-package="com.libaba.spring5.collectiontype.aopanno"></context:component-scan>

使用注解创建User和UserProxy对象



在增强类上面添加注解，@Aspect



在spring配置文件中开启生成代理

<!--开启Aspect生成代理对象-->

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

配置不同类型的通知

在增强类里面，在作为通知方法上面添加通知类型注解，使用切入表达式配置

//前置通知,Before注解表示在增强方法之前执行

@Before(value="execution(\*com.libaba.spring5.collectiontype.aopanno.User.add(..)")

//增强的类

@Component

@Aspect //代理对象

public class UserProxy {

//前置通知,Before注解表示在增强方法之前执行

@Before( "execution(\* com.libaba.spring5.collectiontype.aopanno.User.add())")

public void before(){

System.out.println("before^^^^");

}

//后置通知，有异常也执行

@After("execution(\* com.libaba.spring5.collectiontype.aopanno.User.add())")

public void after(){

System.out.println("after");

}

//返回通知，有异常不执行

@AfterReturning("execution(\* com.libaba.spring5.collectiontype.aopanno.User.add())")

public void afterreturning(){

System.out.println("afterreruning");

}

//出现异常执行

@AfterThrowing("execution(\* com.libaba.spring5.collectiontype.aopanno.User.add())")

public void afterthrowing(){

System.out.println("afterthrowing");

}

//环绕通知

@Around("execution(\* com.libaba.spring5.collectiontype.aopanno.User.add())")

public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

System.out.println("环绕之前");

//被增强的方法,通过这个参数可以执行被增强的方法

proceedingJoinPoint.proceed();

System.out.println("环绕之后");

}

}

1. 细节问题

公共切入点抽取

对切入点的公共部分抽取出来

//相同切入点抽取

@Pointcut("execution(\* com.libaba.spring5.collectiontype.aopanno.User.add())")

public void pointdemo(){

}

//前置通知,Before注解表示在增强方法之前执行

@Before( "pointdemo()")

public void before(){

System.out.println("before^^^^");

}

1. 出现多个增强类对用一个方法进行增强，可以设置增强类的优先级

在增强类上添加一个注解。@Order(数字) ，数字越小，优先级越高

1. Xml配置文件方式实现AOP

1）（了解）

创建两个类，增强类和被增强类，创建方法

在spring配置文件中创建两个类对象

在spring配置文件中配置切入点，配置切面（把增强的方法应用到切入点上）

<aop:config>

<!-- 切入点-->

<aop:pointcut id="p" expression="execution(\* com.libaba.spring5.collectiontype.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:before>

</aop:aspect>

</aop:config>

1. 完全注解方式实现

创建一个配置类，不需要xml文件，在其中写：

@Configuration

@ComponentScan(basePackages = {"com.libaba"}) //代替开启注解扫描

@EnableAspectJAutoProxy(proxyTargetClass = true)//代替开启AspectJ扫描

public class ConfigAop {

}

1. JDBCTemplate

什么是JDBCTemplate

Spring对JDBC的封装，试用JDBCTemplate 更加方便对数据库进行操作

准备：

1. 引入相关依赖

Mysql,orm,tx,jdbc,druid

1. 在Spring配置文件中配置连接池

<!-- 数据库连接池 -->

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

destroy-method="close">

<property name="url" value="jdbc:mysql:///user\_db" />

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

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

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

</bean>

1. 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>

1. 创建service类，dao类，在dao类里面注入jdbcTemplate对象

配置文件中

<!-- 开启组件扫描 -->

<context:component-scan base-package="com.libaba.spring5.collectiontype"></context:component-scan>

Service:

@Service

public class BookService {

//注入dao

@Autowired

private BookDao bookDao;

}

Dao:

@Repository

public class BookDaoImpl implements BookDao{

//注入JdbcTemplate

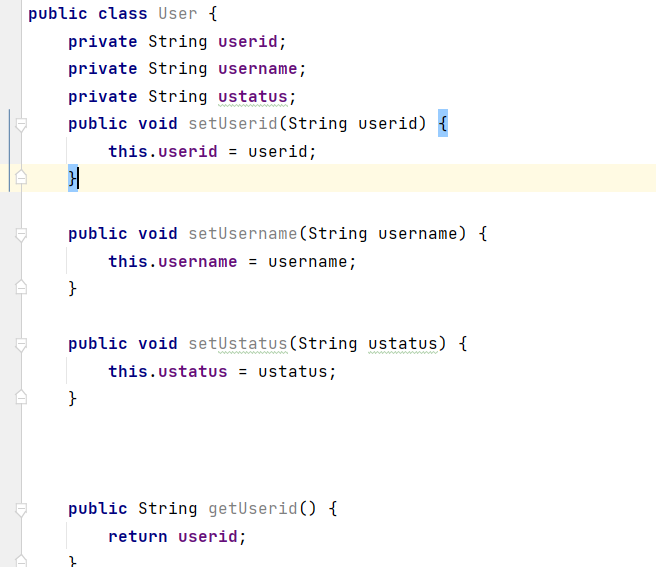
@Autowired

private JdbcTemplate jdbcTemplate;

}

操作数据库：

1. 对应数据库中的表创建出实体类



1. 编写service和dao

在dao里进行数据库添加操作

Update方法，实现添加操作，两个参数，1sql语句，2可变参数，设置sql语句值

@Repository

public class BookDaoImpl implements BookDao{

//注入JdbcTemplate

@Autowired

private JdbcTemplate jdbcTemplate;

@Override

public void add(Book book) {

String sql="insert into t\_book values(?,?,?)";

//update第二个参数，可变，根据实际需要传入的情况确定多少个，内容是啥

int update=jdbcTemplate.update(sql,book.getUserid(),book.getUsername(),book.getUstatus());

System.out.println("受影响的行数："+update);

}

}

编写测试类：

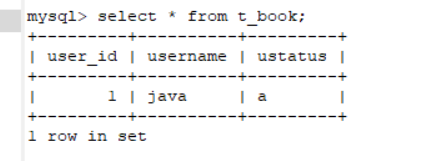
这个过程中出现错误：

java.sql.SQLException: Unknown initial character set index '255' received from server. Initial client character set can be forced via the 'characterEncoding' property. at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1055)

解决方法：

在数据库连接池配置时这样写：

<property name="url" value="jdbc:mysql:///user\_db?characterEncoding=utf8" />



修改和删除数据库操作：

//修改

public void updateBook(Book book){

bookDao.updateBook(book);

}

//删除

public void deleteBook(String id){

bookDao.delete(id);

}

//修改

@Override

public void updateBook(Book book) {

String sql="update t\_book set username=?,ustatus=? where user\_id=?";

int type=jdbcTemplate.update(sql,book.getUsername(),book.getUstatus(),book.getUserid());

System.out.println("受影响的行数："+type);

}

//删除

@Override

public void delete(String id) {

String sql="delete from t\_book where user\_id=?";

int type=jdbcTemplate.update(sql,id);

System.out.println("受影响的行数："+type);

}

写测试类。。。

数据库查询操作：

1）查询返回某个值：

使用函数是

jdbcTemplate.queryForObject(sql,Integer.class);

这个函数的两个参数是sql语句，和返回类型class

2）查询返回对象：

函数jdbcTemplate.queryForObject（），三个参数：

1 Sql语句，2 RowMapper接口，返回不同类型数据，使用这个接口实现类完成数据封装

3sql语句值

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

第二个参数就能返回一个Book类型的对象

3）查询返回集合：

查询列表分页：

query函数，三个参数：和上边相同

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

对数据库的批量操作：

下面三种操作使用的方法是：



两个参数：

第二个参数，List集合，表示添加的记录多条数据

批量添加：

Service类：

//批量添加

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

bookDao.batchAddBook(batchArgs);

}

Dao类：

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));

}

批量删除：

与后边的差不多

批量修改：

具体做法：

1. 先在Service类里添加一个方法bookUpdate1()

//批量修改

public void batchUpdate1(List<Object[]> batchArgs){

bookDao.batchUpdateBook(batchArgs);

}

1. 在BookDao接口中创建batchUpdateBook方法，然后在实现类里面写实现方法：

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

String sql="update t\_book set username=?,ustatus=? where user\_id=?";

int[] ints=jdbcTemplate.batchUpdate(sql,batchArgs);

System.out.println(Arrays.toString(ints));

}

3.写测试类，进行测试

//批量修改

List<Object[]> batchArgs=new ArrayList<>();

Object[] o1={"jaav","b","3",};

Object[] o2={"c++","d","5"};

Object[] o3={"c##","b","4"};

batchArgs.add(o1);

batchArgs.add(o2);

batchArgs.add(o3);

bookService.batchUpdate1(batchArgs);

1. 事务操作

事务概念：

1）什么是事务：事务是数据库操作最基本单元，逻辑上1一组操作，要么全部成功，如果有一个失败，全部失败

举例：银行转账，A给B 100，A没了，B 200

2）事务的四个特性：（ACID）

原子性：

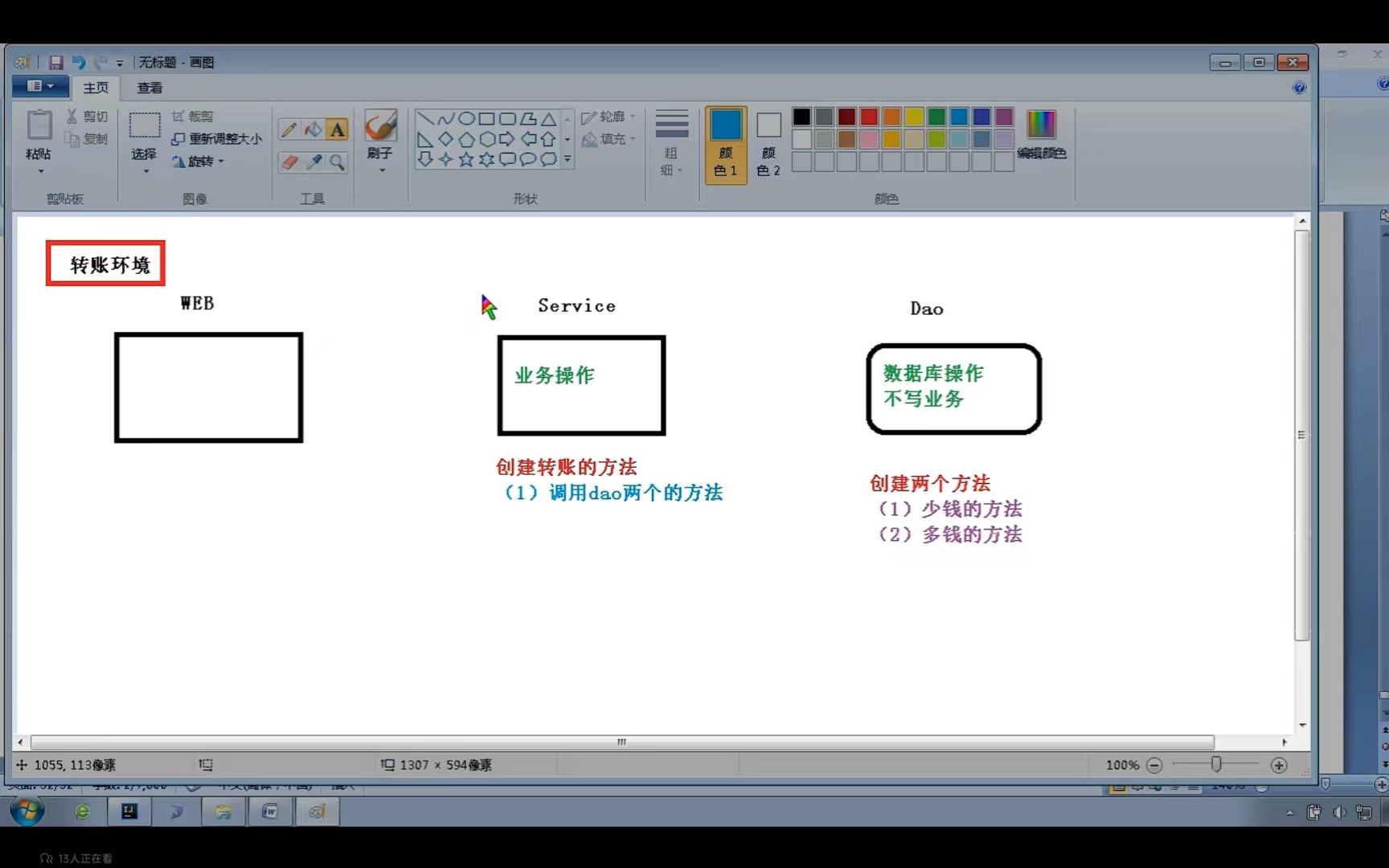
一致性：

隔离性：

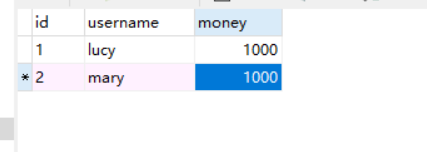
持久性：

事务操作：

搭建环境：



1. 创建一个表，添加记录



1. 创建Service，dao，完成对象的创建和注入关系

Service中注入dao，dao里面注入jdbc模板，jdbc模板中注入datasource数据源

1. 在Dao里面创建两个方法，一个多钱，一个少钱，在service中创建转账方法
2. 当出现问题时：

//转账方法

public void accountMoney(){

//lucy少100

userDao.reduceMoney();

//模拟异常

int i=10/0;

//mary多100

userDao.addMoney();

}

结果会减少，但增加的不会增加

使用事务解决。。。

事务操作过程：

开启事务操作，

进行业务操作，

没有出现异常进行提交事务

出现异常事务回滚

将事务一般加在Service层（业务逻辑层）上

在Spring中进行事务管理操作：

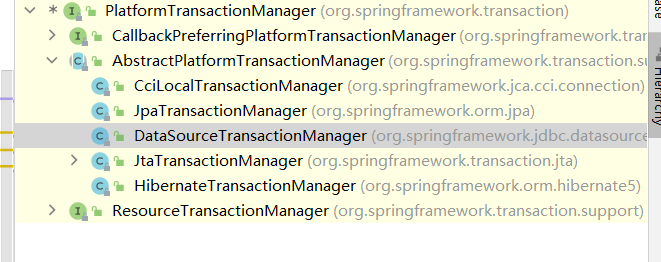
编程式事务管理（很不方便）和声明式事务管理

声明式事务管理：基于注解方式和基于XML配置文件

在Spring中进行声明式事务管理操作，底层使用AOP原理

Spring事务管理API：

PlatformTransactionManager提供接口，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



Spring中进行声明式事务管理操作（基于注解方式）：

1. 在Spring中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>

1. 在配置文件中配置开启事务的注解

引入名称空间tx

开启事务的注解

<!--开启事务注解-->

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

1. 在Service类上边或者方法上面加事务注解：

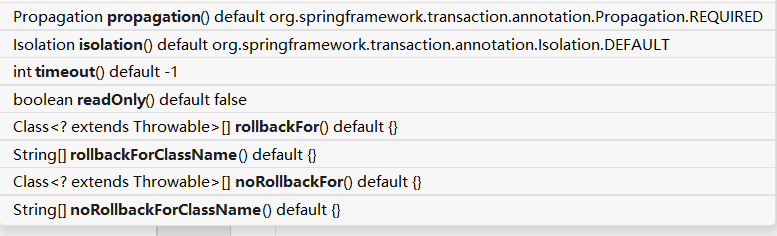
@Transactional

加到类上边表示类的所有方法都添加事务

加到方法上，只在方法上添加事务

添加事务后，测试，出现异常不会改变减少和增加的值

参数配置：



黑体标出的就是@Transactional 里面要配置的参数

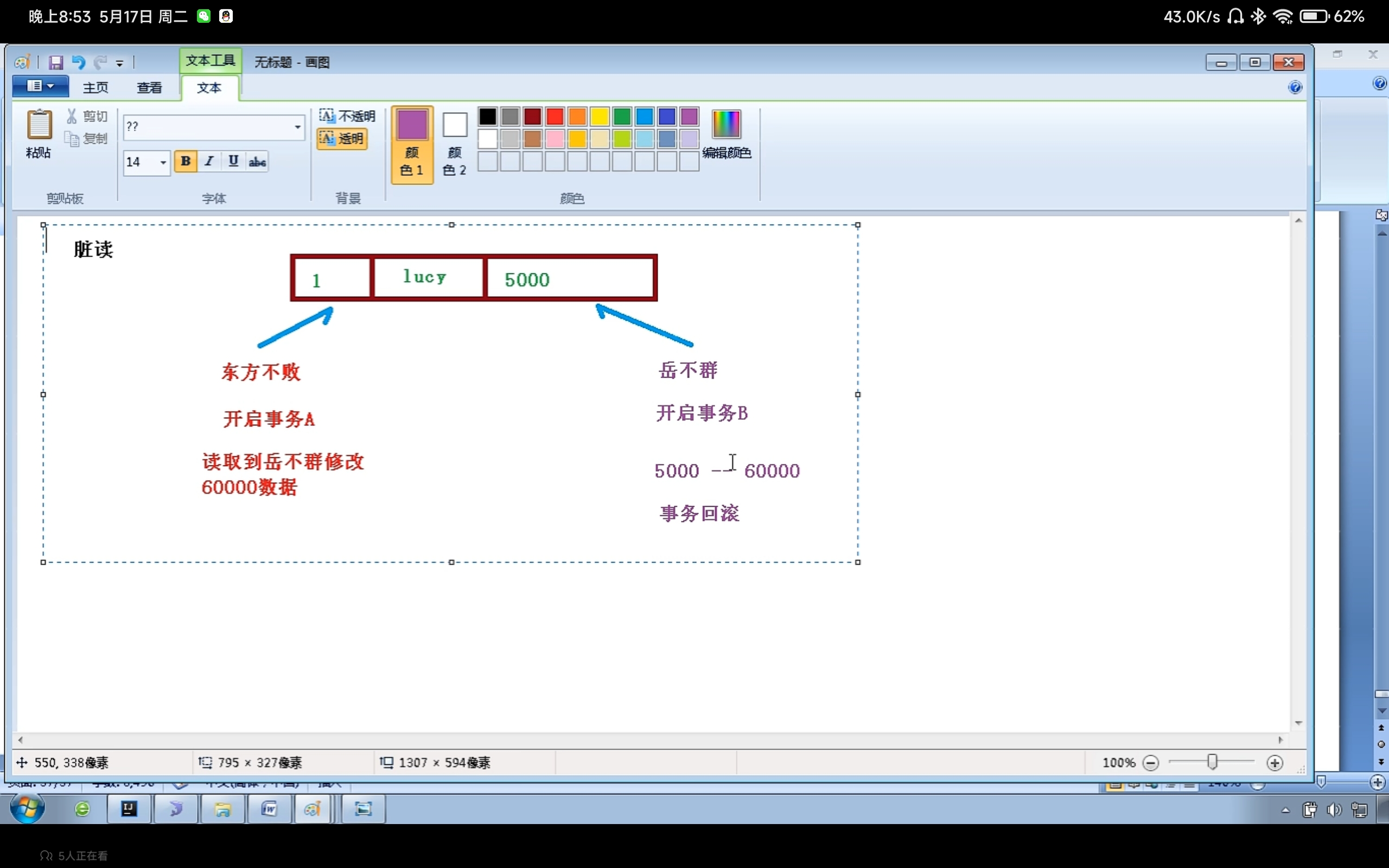
propagation 表示事务的传播行为,也就是多个方法相互调用时，方法对应的事务应该怎么处理



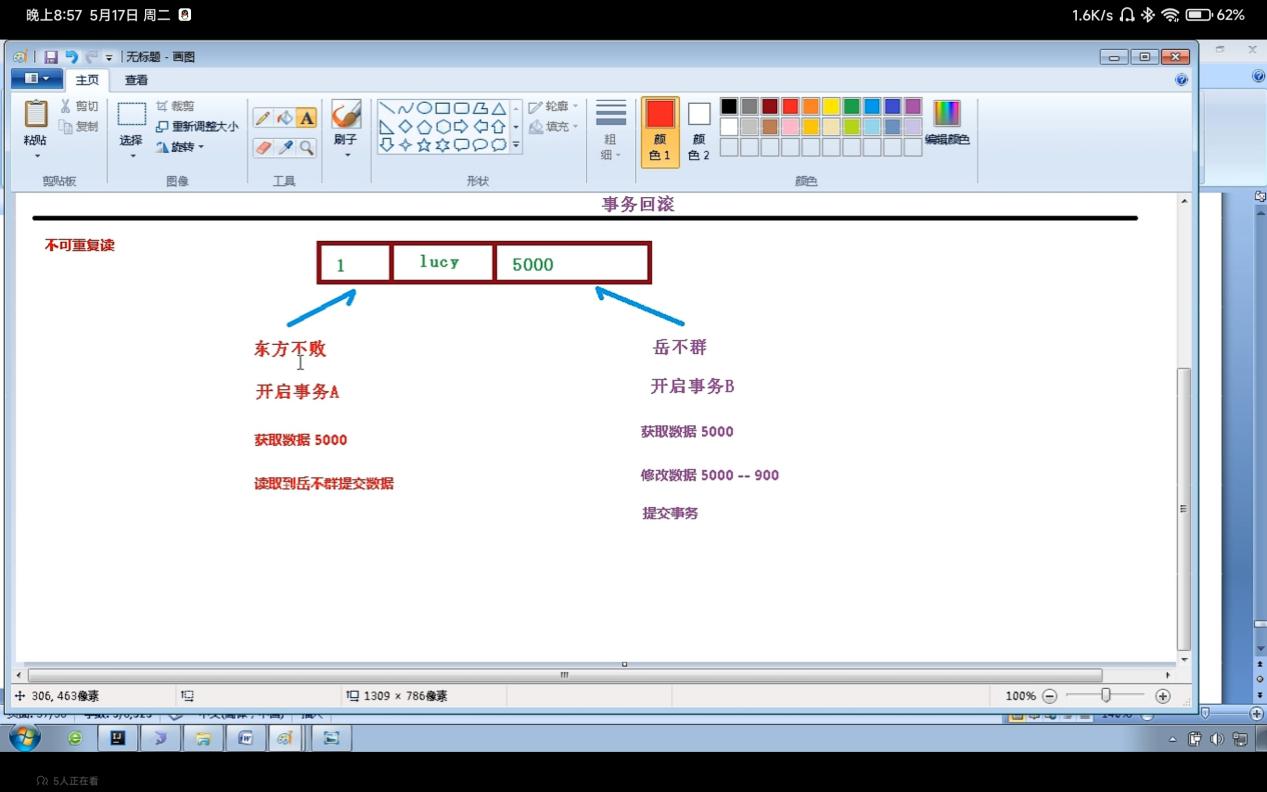
isolation 隔离级别

不考虑隔离，会出现三个问题：脏读，不可重复读，虚读（幻读）

脏读：一个未提交的事务读取到了另一个未提交的事务的数据



不可重复读:



虚读（幻读）：一个未提交事务读取到另一个提交事务添加数据

通过设置事务的隔离性，解决上面的三个问题：

隔离级别：



@Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.REPEATABLE\_READ)

timeout 超时时间

事务在一定时间内完成提交，如果不提交回滚，默认是-1，可以设置时间以秒为单位

readonly 是否只读

读指 查询操作

readOnly 默认值是false，表示可以查询，也可以进行添加修改删除操作

设置为true之后，只能进行查询操作

rollbackFor 回滚

设置出现了哪些异常进行回滚

noRollbackFor 不回滚

设置出现了哪些异常不进行回滚

Spring中进行声明式事务管理操作（基于XML配置文件方式）：

操作过程：

在Spring配置文件中进行配置

第一步，配置事务管理器；第二步，配置通知；第三步，配置切入点和切面

也就是先配置事务，然后配置方法，然后将事务放到方法上

<!--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>

<!--配置通知（事务）-->

<tx:advice id="txadvice">

<!--配置事务参数-->

<tx:attributes>

<!--在方法上面添加事务,后边加\*表示以account开头的方法-->

<tx:method name="accountmoney" propagation="REQUIRED"/>

<!-- <tx:method name="account\*"/>-->

</tx:attributes>

</tx:advice>

<!-- 配置切入点和切面-->

<aop:config>

<!-- 配置切入点-->

<aop:pointcut id="pt" expression="execution(\* com.libaba.spring5.collectiontype.service.UserService.\*(..))"/>

<!-- 配置切面-->

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

</aop:config>

Spring中进行声明式事务管理操作（完全注解方式）：

写一个配置类：

@Configuration //配置类

@ComponentScan(basePackages = "com.libaba") //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 dataSource=new DruidDataSource();

dataSource.setDriverClassName("com.alibaba.druid.pool.DruidDataSource");

dataSource.setUrl("jdbc:mysql:///user\_db");

dataSource.setUsername("root");

dataSource.setPassword("libaba");

return dataSource;

}

//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();

//注入datasource

jdbcTemplate.setDataSource(dataSource);

return jdbcTemplate;

}

//创建事务管理器对象

@Bean

public DataSourceTransactionManager getDataSourceTransactionManager(DataSource dataSource){

DataSourceTransactionManager transactionManager=new DataSourceTransactionManager();

transactionManager.setDataSource(dataSource);

return transactionManager;

}

}