# Spring学习笔记

# Spring框架概述

## Spring框架概述

1. Spring是轻量级的开源的javaEE框架
2. Spring可以解决企业应用开发的复杂性
3. Spring有两个核心部分：IOC和Aop
4. IOC：控制翻转，把创建对象交给Spring框架
5. Aop：面向切面，不修改源代码进行功能增强
6. Spring特点：
7. 方便解耦，简化开发
8. Aop编程的支持
9. 方便程序的测试
10. 方便和其他框架进行整合
11. 方便进行事物操作
12. 降低API开发的难度
13. Java源码是经典学习规范
14. Spring配置文件使用xml文件

## IOC（概念和原理）

1. 什么是IOC
2. 控制反转，把对象的创建和对象之间的调用过程，交给Spring进行管理
3. 使用IOC的目的：为了耦合度降低
4. IOC底层原理“
5. xml解析，工厂的设计模式，反射
6. IOC过程
7. Xml配置文件，配置创建的对象

|  |
| --- |
| <bean id=”dao” class =”com.ly.UserDao”></bean> |

1. 有service类和dao类，创建工厂类

|  |
| --- |
| Class UserFactory{  Public static UserDao getDao(){  String classValue=class属性值；//xml解析  //2.通过反射创建对象  Class clazz=Class.forName(classValue);  return Clazz.newInstance();  }  } |

## IOC(接口)

1. IOC思想基于IOC容器完成，IOC的容器底层就是对象工厂
2. Spring提供IOC实现的两种方式：（两个重要的接口）
3. BeanFactory：IOC容器基本实现方式，是Spring内部的使用接口，不提供开发人员进行使用

\*加载配置文件时候不会创建对象，在获取（使用）对象的时候才去创建对象

1. ApplicationContext：BeanFactory接口的子接口，提供更多更强大的功能，一般是面向开发人员进行使用

\*加载配置文件时候就会把在配置文件对象进行创建

3.ApplicationContext接口有实现类：

FileSystemXmlApplicationContext:把盘里面的路径加载出来

ClassPathXmlApplicatiobContext:在类的路径加载出来

# IOC操作Bean管理

## Bean管理：

1. Spring创建对象
2. Spring注入属性

## Bean管理操作有两种方式

1. 基于xml文件方式实现
2. 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
3. Bean标签有很多属性
4. id属性：唯一标识
5. class属性：类全路径（包类路径）
6. 创建对象时候，默认也是执行无参数的构造方法

### 3.基于注解方式实现

（1）DI：依赖注入，就是注入属性

第一种注入方式：使用set方法进行注入

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

第二种注入方式：使用有参数构造进行注入

1. 创建类，定义属性，创建属性对应有参构造方法
2. 在spring配置文件中，进行配置

### 4.名称空间注入

1. 使用p名称空间注入，可以简化基于xml配置方式

第一步添加p名称空间在配置文件中

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

第二步;进行属性注入，在bean标签里面进行操作

## IOC操作Bean管理（xml注入其他属性）

### 1.字面量

1. null值
2. <property name="address">  
    <null></null>  
   </property>
3. 属性值包含特殊符号

<!--属性值中包含特殊符号

1把<>进行转义

2.把带特殊符号内容写到CDATA

-->

<property name="address">

<value><![CDATA[<<南京>>]]>

</value>

#### （4）注入属性-外部bean

1. 创建两个类service类和dao类
2. 在service调用dao里面的方法
3. 在spring配置文件中进行配置

#### （5）注入属性-内部bean级联赋值

1. 一对多关系：部门和员工

一个部门有多个员工，一个员工属于一个部门

部门是一，员工是多

1. 在实体类之间表示一对多关系，员工表示所属部门，使用对象类型属性进行表示
2. 在spring配置文件中进行配置

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

<!--内部bean-->

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

<!--设置两个基本属性-->

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

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

<!--设置对象类型属性-->

<property name="dept">

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

<property name="dname" value="人事部"></property>

</bean>

</property>

#### （6）注入属性-级联赋值

（1）第一种写法

|  |
| --- |
| <bean id="emp" class="com.ly.spring5.bean.Emp">  <!--设置两个基本属性-->  <property name="gender" value="女"></property>  <property name="ename" value="ZMK"></property>  <!--级联赋值-->  <property name="dept" ref="dept"></property> </bean> <bean id="dept" class="com.ly.spring5.bean.Dept">  <property name="dname" value="财务部"></property> </bean> |

1. 第二种写法

首先，要生成对应的get方法

|  |
| --- |
| <bean id="emp" class="com.ly.spring5.bean.Emp">  <!--设置两个基本属性-->  <property name="gender" value="女"></property>  <property name="ename" value="ZMK"></property>  <!--级联赋值-->  <property name="dept" ref="dept"></property>  <property name="dept.dname" value="总经理"></property> </bean> <bean id="dept" class="com.ly.spring5.bean.Dept">  <property name="dname" value="财务部"></property> </bean> |

## IOC操作Bean管理（xml注入集合属性）

1. 注入数组类型属性
2. 注入List集合类型属性
3. 注入Map集合类型属性
4. 创建类，定义数组,list,map,set类型属性，生成对应的set方法
5. 在spring配置文件中进行配置

|  |
| --- |
| 1. <bean id="stu" class="com.ly.collectiontype.Stu"> <!--完成对数组类型的注入-->  <property name="course">  <array>  <value>JAVA课程</value>  <value>数据库课程</value>  </array>  </property>  <!--完成对list数据类型的注入-->  <property name="list ">  <list>  <value>张三</value>  <value>小三</value>  </list>  </property>  <!--完成对map类型注入-->  <property name="maps">  <map>  <entry key="JAVA" value="JAVA"></entry>  <entry key="PHP" value="php"></entry>  </map>  </property>  <!--set类型注入-->  <property name="sets">  <set>  <value>MYSQL</value>  <value>Redis</value>  </set>  </property>   </bean> |

1. 在集合中设置对象类型值

|  |
| --- |
| 1. <!--注入list集合类型，值是对象-->  <property name="coursesList ">  <list>  <ref bean="course1"></ref>  <ref bean="course2"></ref>  </list>  </property> <!-- <bean id></bean>-->  </bean>  <!--创建对个course对象-->  <bean id="course1" class="com.ly.collectiontype.Course">  <property name="cname" value="spring5框架"></property>  </bean>  <bean id="course2" class="com.ly.collectiontype.Course">  <property name="cname" value="Mybabits框架"></property>  </bean> |

1. 把集合注入部分提取出来
2. 在spring配置文件中引入名称空间 util
3. 使用util标签完成list集合注入提取

|  |
| --- |
| 1. <!--1.提取list集合类型属性注入-->  <util:list id="bookList">  <value>java</value>  <value>mysql</value>  <value>c++</value>  </util:list> <!--2.提取list集合类型属性使用--> <bean id="book" class="com.ly.collectiontype.Book">  <property name="list" ref="bookList">    </property> </bean> |

IOC操作Bean管理（FactoryBean）

1. Spring有两种类型的bean，一种普通bean，另外一种工厂bean（FactoryBean）
2. 普通bean：在spring的配置文件中定义bean类型就是返回类型
3. 工厂bean：在配置文件定义bean类型可以和返回的类型不一样
4. 第一步：创建类，让这个类作为工厂bean，实现接口Factorybean
5. 第二步：实现接口里面的方法，在实现的方法中定义返回的bean类型

|  |
| --- |
| 1. public class MyBean implements FactoryBean<Course> {  //定义返回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();  } } |

|  |
| --- |
| @Test public void test3(){  ApplicationContext context=new ClassPathXmlApplicationContext("bean3.xml");  Course course = context.getBean("myBean", Course.class);  System.*out*.println(course); } |

## IOC操作Bean管理（bean作用域）

|  |
| --- |
| 1. 在Spring里面，设置创建bean实例是单实例还是多实例 2. 在Spring里面，默认情况下，bean是单实例对象， 3. 如何设置是单实例还是多实例 4. 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 5. Scope属性值：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   <bean id="book" class="com.ly.collectiontype.Book" scope="prototype">  <property name="list" ref="bookList">  </property> </bean> |

|  |
| --- |
| Singleton和prototype区别   1. Singleton单实例，prototype多实例 2. 设置scope值是singleton时候，加载spring配置文件时候就会创建单实例对象 3. 设置scope值是prototype时候，不是在加载spring配置文件时候创建对象，在调用getBean方法时候创建多实例对象== |

## IOC操作Bean管理（bean生命周期）

|  |
| --- |
| 1. 生命周期： 2. 从对象创建到对象销毁的过程 3. bean生命周期： 4. 通过构造器创建bean实例（无参数构造） 5. 为bean的属性设置值和对其他bean引用（调用set方法） 6. 调用bean的初始化的方法（需要进行配置初始化的方法） 7. Bean可以调用了 8. 当容器关闭时候，调用bean的销毁的放啊发（需要进行配置销毁的方法） 9. bean的后置处理器，bean生命周期有七步   （5）通过构造器创建bean实例（无参数构造）   1. 为bean的属性设置值和对其他bean引用（调用set方法） 2. 把bean实例传递bean后置处理器的方法（postProcessBeforeInitialization） 3. 调用bean的初始化的方法（需要进行配置初始化的方法） 4. 把bean实例传递bean后置处理器的方法（postProcessAfterInitialization） 5. Bean可以调用了 6. 当容器关闭时候，调用bean的销毁的放啊发（需要进行配置销毁的方法） |

## IOC操作Bean管理（xml自动装配）

1. 什么是自动装配
2. 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
3. 根据属性的名称进行注入：

|  |
| --- |
| 1. <!--实现自动装配  bean标签属性autowire：配置自动装配  autowire常用属性：byName根据属性名称注入 :注入bean的id值和类属性名称一样  byType：根据属性类型注入    -->  <bean id="emp" class="com.ly.autowire.Emp" autowire="byName"> <!-- //手动转配--> <!-- <property name="dept" ref="dept"></property>-->  </bean>  <bean id="dept" class="com.ly.autowire.Dept">   </bean> |

1. 根据属性类型进行注入

|  |
| --- |
| 1. <!--实现自动装配  bean标签属性autowire：配置自动装配  autowire常用属性：byName根据属性名称注入 :注入bean的id值和类属性名称一样  byType：根据属性类型注入   -->  <bean id="emp" class="com.ly.autowire.Emp" autowire="byType"> <!-- //手动转配--> <!-- <property name="dept" ref="dept"></property>-->  </bean>  <bean id="dept" class="com.ly.autowire.Dept">   </bean> |

## IOC操作Bean管理（外部属性文件）

1. 直接配置数据库信息
2. 配置德鲁伊连接池
3. 引入德鲁伊连接池依赖jar包

|  |
| --- |
| 1. <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="123456"></property> </bean> |

1. 通过引入外部属性文件配置数据库连接池
2. 创建外部属性文件，properties文件，写入数据库的配置信息

|  |
| --- |
| 1. prop.driverClass=com.mysql.jdbc.Driver prop.url=jdbc:mysql://localhost:3306/userDb prop.userName=root prop.password=123456 |

1. 把外部properties属性文件引入到spring配置文件中

\*引入context名称空间

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

1. 在spring配置文件使用标签引入外部属性文件

|  |
| --- |
| 1. <!--引入外部属性文件-->  <context:property-placeholder location="classpath:jdbc.properties"></context:property-placeholder>  <!--直接配置连接池--> <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  <property name="driverClassName" value="${prop.driverClass}"></property>  <property name="url" value="${prop.url}"></property>  <property name="username" value="${prop.userName}"></property>  <property name="password" value="${prop.password}"></property> </bean> |

## IOC操作Bean管理（基于注解方式）

1. 什么是注解
2. 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……）
3. 使用注解，注解作用在类上面，方法上面，属性上面
4. 使用注解目的：简化xml配置
5. Bean管理（创建对象）提供注解

（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

3.基于注解方式实现对象创建

第一步操作：引入依赖



第二步：开启组件扫描

|  |
| --- |
| <!--开启组件扫描 1.如果扫描多个包，多个包之间使用逗号隔开 2.扫描包上层目录   --> <context:component-scan base-package="com.ly"></context:component-scan> |

第三步：创建类，在类上面添加创建对象注解

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

1. 开启组件扫描中细节配置

|  |
| --- |
| 1. <!--实例1  use-default-filters="false":表示现在不适用默认filter，自己配置filter  context:include-filter:设置扫描那些内容  -->  <context:component-scan base-package="com.ly" use-default-filters="false">  <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>  </context:component-scan>  <!--实例2  下面配置扫描包所有内容  context:exclude-filter：设置那些内容不扫描 表示除了Controller都扫描    -->  <context:component-scan base-package="com.ly">  <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>  </context:component-scan> |

1. 基于注解方式实现属性注入
2. @AutoWired:根据属性类型进行自动装配

第一步：把service和dao对象创建，在service和dao类添加创建对象注解

第二步：在service注入dao对象，在service类中添加dao类型属性，在属性上面使用注解

|  |
| --- |
| @Service(value = "userService") //<bean id="userService" class="..."> public class UserService {  //定义属性  //不需要添加set方法  //添加注入属性注解  @Autowired  private UserDao userDao;  public void add(){  System.*out*.println("service addd.....");  userDao.add();  } |

1. @Qualifier：根据属性的名称进行注入

@ Qualifier要和@AutoWired一起使用

|  |
| --- |
| //定义属性 //不需要添加set方法 //添加注入属性注解 @Autowired//根据类型进行注入 @Qualifier(value = "userDaoImpl1")//根据名称进行注入  private UserDao userDao; public void add(){  System.*out*.println("service addd.....");  userDao.add(); } |

1. @Resource：可以根据类型注入，可以根据名称注入

|  |
| --- |
| 1. // private UserDao userDao; // @Resource //根据类型进行注入  @Resource(name = "userDaoImpl1")//根据名称进行注入  private UserDao userDao;  public void add(){  System.*out*.println("service addd.....");  userDao.add();  } |

Resource：是javax包（java扩展包中的）

1. @Value：注入普通类型属性

|  |
| --- |
| 1. @Value(value = "ZMK") private String name; |

1. 完全注解开发
2. 创建配置类，替代xml配置文件

|  |
| --- |
| 1. @Configuration //作为配置类，替换xml配置文件 @ComponentScan (basePackages ={"com.ly"}) public class SpringConfig {   } |

1. 编写测试类

|  |
| --- |
| 1. public void testService2(){  //加载配置类  ApplicationContext context=new AnnotationConfigApplicationContext(SpringConfig.class);  UserService userService = context.getBean("userService", UserService.class);  System.*out*.println(userService);  userService.add(); } |

# AOP

## 什么是AOP

1. 面向切面编程（方面），利用AOP可以对业务逻辑的各个部分进行分隔，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底层原理

1. 有两种情况动态代理

第一种：有接口情况，使用JDK动态代理

创建接口实现类代理对象，增强类的方法

第二种情况：没有接口情况，使用CGLIB动态代理

创建当前类子类的代理对象，增强类的方法

## AOP（JDK动态代理）

1. 使用JDK动态代理，使用Proxy类里面的方法创建代理对象
2. 调用newProxyInstance方法 newProxyInstance（ClassLoader loader，类<?>[] interfaces,InvocationHandler h） 返回指定接口的代理类的实例，该接口将方法调用分派给指定的调用处理程序

方法里面有三个参数：

第一个参数：ClassLoader，类加载器

第二个参数，interfaces 增强方法所在的类，这个类实现的接口，支持多个接口

第三个参数，实现这个接口InvocationHandler，创建代理对象，写增强的方法

1. 编写JDK动态代理代码
2. 创建接口，定义方法

|  |
| --- |
| 1. package com.ly;  public interface UserDao {  public int add(int a,int b);  public String update(String id); } |

1. 创建接口实现类，实现方法

|  |
| --- |
| 1. package com.ly;  public class UserDaoImpl implements UserDao{   @Override  public int add(int a, int b) {  return a+b;  }   @Override  public String update(String id) {  return id;  } } |

（5）使用Proxy类创建接口代理对象

|  |
| --- |
| package com.ly;  import org.junit.internal.RealSystem;  import java.lang.reflect.InvocationHandler; import java.lang.reflect.Method; import java.lang.reflect.Proxy; import java.util.Arrays;  public class JDKProxy {  public static void main(String[] args) {  //创建接口实现类代理对象  Class[] interfaces={UserDao.class};  /\* 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, 12);  System.*out*.println(result);  } } //创建代理对象代码 class UserDaoProxy implements InvocationHandler{  //1.把创建的是谁的代理对象，把谁传递过来  //有参数的构造传递  private Object obj;  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("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("方法之后执行...."+obj);  return res;  }  } |

## AOP（操作术语）

1. 连接点：类里面那些方法可以被增强，这些方法被称为连接点
2. 切入点：实际被增强的方法，成为切入点
3. 通知（增强）：
4. 实际被增强的逻辑部分成为通知（增强）
5. 通知有多种类型

\*前置通知

\*后置通知

\*环绕通知

\*异常通知

\*最终通知

1. 切面

是动作

1. 把通知应用到切入点过程叫做通知

## AOP操作（准备）

1. Spring框架一般基于AspectJ实现AOP操作
2. 什么是AspectJ

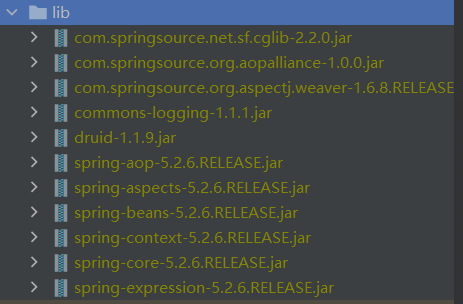
\*AspectJ不是Spring组成部分，独立AOP框架，一般把AspectJ和Spring框架一起使用，进行AOP操作

（2）基于Aspectj实现AOP操作

（1）基于xml配置文件实现

（2）基于注解方式实现（一般使用）

3.在项目工程里面引入AOP相关依赖



1. 切入点表达式
2. 切入点表达式作用：可以知道对哪个类里面的哪个方法进行增强
3. 语法结构：

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

举例1：对com.atguigu.dao.BookDao类里面的add进行增强

execution(\* com.atguigu.dao.BookDao.add(..))

## AOP操作（Aspectj注解）

1. 创建类，在类里面定义方法

|  |
| --- |
| 1. package com.ly.aopanno;  public class User {  public void add(){  System.*out*.println("add......");  } } |

1. 创建增强类（编写增强逻辑）
2. 在增强类里面，创建方法，让不同方法代表不同通知类型

|  |
| --- |
| 1. */\*\*  \* 增强的类  \*/* public class UserProxy {  //前置通知  public void before(){  System.*out*.println("before......");  } } |

1. 进行通知的配置
2. 在spring配置文件中，开启注解扫描

|  |
| --- |
| 1. <!--开启注解扫描--> <conetext:component-scan base-package="com.ly.aopanno"></conetext:component-scan> |

1. 使用注解创建User和UserProxy对象

|  |
| --- |
| 1. @Component public class User {  public void add(){  System.*out*.println("add......");  } }   @Component public class UserProxy {  //前置通知  public void before(){  System.*out*.println("before......");  } } |

1. 在增强类上面添加注解@Aspect

|  |
| --- |
| 1. @Component @Aspect //生成代理对象 public class UserProxy {  //前置通知 |

1. 在spring配置文件中开启生成代理对象

|  |
| --- |
| 1. <!--开启AspectJ生成代理对象--> <aop:aspectj-autoproxy></aop:aspectj-autoproxy> |

1. 配置不同类型的通知
2. 在增强类的里面，在作为通知方法上面添加通知类型注解，使用切入点表达式配置

|  |
| --- |
| 1. //前置通知 //@Before注解表示作为前置通知，使用切入点表达是 @Before(value = "execution(\* com.ly.aopanno.User.add(..))") public void before(){  System.*out*.println("before......"); } |

1. 相同的切入点进行抽取

|  |
| --- |
| 1. //相同切入点进行抽取 @Pointcut(value = "execution(\* com.ly.aopanno.User.add(..))") public void pointdemo(){   } //前置通知 //@Before注解表示作为前置通知，使用切入点表达是 @Before(value = "pointdemo()") public void before(){  System.*out*.println("before......"); } |

1. 有多个增强类对同一个方法进行增强，设置增强类优先级

（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

## AOP操作（Aspectj配置文件）

1. 创建两个类，增强类和被增强类，创建方法
2. 在spring配置文件中创建两个类对象

|  |
| --- |
| 1. <!--创建对象--> <bean id="book" class="com.ly.aopxml.Book"> </bean> <bean id="bookProxy" class="com.ly.aopxml.BookProxy"></bean> |

1. 在spring配置文件中配置切入点

|  |
| --- |
| 1. <!--创建对象--> <bean id="book" class="com.ly.aopxml.Book"> </bean> <bean id="bookProxy" class="com.ly.aopxml.BookProxy"></bean> <!--配置aop增强--> <aop:config>  <!--切入点-->  <aop:pointcut id="p" expression="execution(\* com.ly.aopxml.Book.buy(..))"/>  <!--配置切面-->  <aop:aspect ref="bookProxy">  <!--增强作用在具体的方法上-->  <aop:before method="before" pointcut-ref="p"></aop:before>  </aop:aspect> </aop:config> |

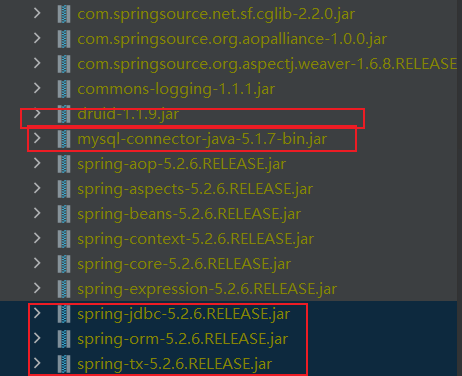
1. 完全使用注解开发
2. 创建配置类，不需要配置xml配置类

|  |
| --- |
| 1. @Configuration @ComponentScan(basePackages = {"com.ly"}) @EnableAspectJAutoProxy(proxyTargetClass = true) public class Config {  } |

# JdbcTemplate

## 概念和准备工作

1. 什么是jdbcTemplate
2. Spring框架对JDBC进行封装，使用JDBCTempalte方便实现对数据库操作
3. 准备工作
4. 引入相关jar包



1. 在spring的配置文件中，配置数据库的连接池

|  |
| --- |
| 1. <!-- 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="123456" />  <property name="driverClassName" value="com.mysql.jdbc.Driver" /> </bean> |

1. 配置jdbcTemplate对象，注入DataSource

|  |
| --- |
| 1. <!--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，创建dao类，在dao注入jdbcTemplate对象

\*配置文件中

<!--开启组件的扫描-->  
 <conetext:component-scan base-package="com.ly"></conetext:component-scan>

\*在service中

|  |
| --- |
| @Service public class BookService {  //注入dao  @Autowired  private BookDao bookDao; } |

\*在dao中

|  |
| --- |
| @Repository public class BookDaoImpl implements BookDao{  //注入jdbcTemplate  @Autowired  private JdbcTemplate jdbcTemplate; } |

## jdbcTemplate操作数据库

### 添加数据

1.对应数据库表创建实体类

|  |
| --- |
| 1. package com.ly.entity;  public class User {  private String userId;  private String username;  private String ustatus;   public String getUserId() {  return userId;  }   public void setUserId(String userId) {  this.userId = userId;  }   public String getUsername() {  return username;  }   public void setUsername(String username) {  this.username = username;  }   public String getUstatus() {  return ustatus;  }   public void setUstatus(String ustatus) {  this.ustatus = ustatus;  } } |

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



有两个参数

第一个参数：sql语句

第二个参数：可变参数，设置sql语句中值

|  |
| --- |
| @Repository public class BookDaoImpl implements BookDao{  //注入jdbcTemplate  @Autowired  private JdbcTemplate jdbcTemplate;  //添加方法  @Override  public void add(Book book) {  //创建sql语句  String sql="insert into t\_book values(?,?,?)";  //2.调用方法实现  Object[] args={book.getUserId(),book.getUsername(),book.getUstatus()};  int update = jdbcTemplate.update(sql,args);  System.*out*.println(update);  } } |

测试类

### 修改和删除

|  |
| --- |
| //修改 @Override public void updateBook(Book book) {  String sql="update t\_book set username=?,ustatus=?where user\_id=?";  Object[] args={book.getUsername(),book.getUstatus(),book.getUserId()};  int update = jdbcTemplate.update(sql, args);  System.*out*.println(update); }  //删除 @Override public void delete(String id) {  String sql="delete from t\_book where id=?";  int update = jdbcTemplate.update(sql, id);  System.*out*.println(update); } |

### JdbcTemplate操作数据库（查询操作）

#### 查询返回某个值

1. 查询表里面有多少条记录，返回某个值
2. 使用jdbcTemplate实现查询返回某个值代码



1. 有两个参数

第一个参数：sql语句

第二个参数：返回类型Class

|  |
| --- |
| //查询表中记录数 @Override public int selectCount() {  String sql="select count(\*) from t\_book";  Integer count = jdbcTemplate.queryForObject(sql, Integer.class);  return count; } |

#### 查询返回对象

1. jdbcTemplate实现查询返回对象



有三个参数

第一个参数：sql语句

第二个参数：RowMapper，是一个接口，返回不同类型数据，使用这个接口里面的实现类完成对数据的封装，

第三个参数：sql语句值

|  |
| --- |
| @Override public Book findBookInfo(String id) {  String sql="select \* from t\_book where user\_id=?";  //调用jdbcTemplate模板  Book book = jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<Book>(Book.class), id);  return book; } |

#### 查询返回集合

1. 调用jdbcTemplate方法实现查询返回集合



有三个参数

第一个参数：sql语句

第二个参数：RowMapper，是一个接口，返回不同类型数据，使用这个接口里面的实现类完成对数据的封装，

第三个参数：sql语句值

|  |
| --- |
| @Override public List<Book> findAllBook() {  String sql="select \* from t\_book";  List<Book> bookList = jdbcTemplate.query(sql, new BeanPropertyRowMapper<Book>(Book.class));  return bookList; } |

### JdbcTemplate操作数据库（批量操作）

1. 批量操作：操作表里面多条记录
2. JdbcTemplate实现批量添加操作



有两个参数：

第一个参数：sql语句

第二个参数：List集合，添加多条记录数据

|  |
| --- |
| @Override public void batchAddBook(List<Object[]> batchArgs) {  String sql="insert into t\_book values(?,?,?)";  int[] ints = jdbcTemplate.batchUpdate(sql, batchArgs);  System.*out*.println(Arrays.*toString*(ints)); } |
|  |
| //批量添加 List<Object[]> batchArgs=new ArrayList<>(); Object[] o1={"6","java","a"}; Object[] o2={"7","c++","b"}; Object[] o3={"8","MySQL","c"}; batchArgs.add(o1); batchArgs.add(o2); batchArgs.add(o3); //调用批量添加方法 bookService.batchAdd(batchArgs); |

### JdbcTemplate实现批量修改操作

|  |
| --- |
| @Override 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)); } |

### JdbcTemplate实现批量删除操作

|  |
| --- |
| //批量删除 @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(ints); } |

# 事务操作

## 事务概念

### 什么是事务

1. 事务是数据库操作最基本单元，逻辑上一组操作，如果有一个失败所有操作都失败

### 事务特性(ACID)

1. 原子性：原子性是指事务是一个不可分割的工作单位，事务中的操作要么全部成功，要么全部失败。比如在同一个事务中的SQL语句，要么全部执行成功，要么全部执行失败。
2. 一致性: 事务必须使数据库从一个一致性状态变换到另外一个一致性状态。换一种方式理解就是：事务按照预期生效，数据的状态是预期的状态。
3. 隔离性: 事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。
4. 持久性: 持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

### 事务操作（搭建事务操作环境）

1. 创建数据库表，添加记录



1. 创建service，搭建dao，完成对象的创建和注入关系
2. service注入dao，在dao内注入jdbcTemplate，在jdbcTemplate注入DataSource

|  |
| --- |
| 1. @Service public class UserService { //注入Dao  @Autowired  private UserDao userDao;  }   @Repository public class UserDaoImpl implements UserDao {  //注入JdbcTemplate  @Autowired  private JdbcTemplate jdbcTemplate; } |

1. 在dao创建两个方法：多钱和少钱的方法，在service创建方法（转账的方法）

|  |
| --- |
| 1. @Repository public class UserDaoImpl implements UserDao {  //注入JdbcTemplate  @Autowired  private JdbcTemplate jdbcTemplate;  //lucy转账100给mary  //多钱  @Override  public void addMoney() {  String sql="update t\_account set money=money-? where username=?";  jdbcTemplate.update(sql,100,"lucy");  }  //少钱  @Override  public void reduceMoney() {  String sql="update t\_account set money=money+? where username=?";  jdbcTemplate.update(sql,100,"mary");  } |

在service中创建转账的方法

|  |
| --- |
| @Service public class UserService { //注入Dao  @Autowired  private UserDao userDao;  //转账的方法  public void accountMoney(){  //lucy少100  userDao.reduceMoney();  //mary多100  userDao.addMoney();  } } |

4使用事务解决

1. 事务操作过程

|  |
| --- |
| 1. @Service public class UserService { //注入Dao  @Autowired  private UserDao userDao;   //第一步 开启事务   //第二步：进行业务操作   //转账的方法  public void accountMoney(){  try {  //lucy少100  userDao.reduceMoney();  int i=10/0;  //mary多100  userDao.addMoney();  //如果没有异常，提交事务   }catch (Exception e){  //出现异常，事务回滚   }  } } |

### 事务操作（Spring事务管理介绍）

1. 首先，事物添加到JAVAEE三层结构里面Service层（业务）
2. 在Spring进行事务管理操作

有两种方式：

1. 编程式事务管理
2. 声明式事务管理
3. 声明式事务管理
4. 基于注解方式实现（使用）
5. 基于xml配置文件方式实现
6. 在Spring进行声明式事务管理，底层使用AOP原理
7. Spring事务管理API
8. 提供一个接口，代表事务管理器，这个接口针对不同的框架提供不同的实现类

### 事务操作（注解声明式事务管理）

1. 在spring配置文件配置事务管理器

|  |
| --- |
| 1. <!--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spring配置文件中，开启事务注解
2. 在spring配置文件引入名称空间tx

|  |
| --- |
| 1. <?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:conetext="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd"> |

1. 开启事务注解

|  |
| --- |
| 1. <!--开启事务注解--> <tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-driven> |

1. 在service类上面（或者service类里面方法上面）添加事务注解

（1）@Transactional，这个注解可以添加到类上面，也可以添加到方法上面

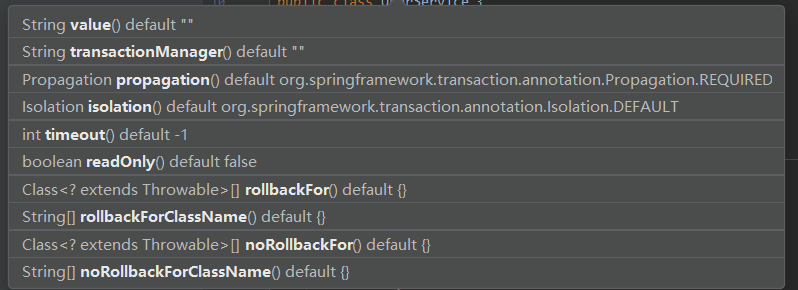
（2）如果把这个注解添加类上面，这个类里面所有的方法都添加事务

（3）如果把这个注解添加方法上面，只是为该方法添加注解

|  |
| --- |
| @Service @Transactional//可以加到类上面，也可以加到方法上面 public class UserService { |

### 事务操作（声明式事务管理参数配置）

1. 在service类上面添加注解@Transactional，在这个注解里面可以配置相关参数



1. propagation()：表示事务传播行为
2. 多事务方法直接进行调用，这个过程中事务是如何进行管理的
3. 事务方法：对数据库数据进行变化的操作
4. Spring框架事务传播行为有7中：①REQUIRED：如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务内运行②REQUIRED\_NEW：当前的方法必须启动新事物，并在他自己的事务内运行，如果有事务正在运行，应该将它挂起③：SUPPORTS：如果有事务在运行，当前的方法就在这个事务内运行，否则它可以不运行在事务中④：NOT\_SUPPORTED:当前的方法不应该运行在事务中，如果有运行的事务，将它挂起⑤：MANDATORY：当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常⑥：NEVER：当前的方法不应该运行在事务内中，如果没有运行的事务，就抛出异常⑦：MESTED：如果有事务在运行，当前的方法就应该在这个事务的嵌套事物内运行，否则，就启动一个新的事务，并在他自己的事务内运行
5. isolation：事务的隔离级别
6. 事务有特性成为隔离性，多事务操作之间不会产生影响，不考虑隔离性产生很多问题
7. 脏读，不可重复读，幻读，
8. 脏读：一个未提交事务读取到另一个未提交事务的数据
9. 不可重复读：一个未提交事务读取到另一个提交事务修改数据
10. 幻读：一个未提交事物读取到另一个提交事务添加数据
11. 通过设置事务隔离性，解决读问题
12. @Transactional(propagation = Propagation.*REQUIRED*,isolation = Isolation.*REPEATABLE\_READ*)/
13. timeout：超时时间
14. 事务需要在一定时间内进行提交，如果不提交进行回滚
15. 默认值是-1，设置时间以秒为1单位1进行计算
16. readOnly：是否只读
17. 读：查询操作，写：添加修改删除操作
18. Readonly默认值false，表示可以查询，可以添加修改删除操作
19. 设置readOnly值是true，设置成true之后，只能查询操作
20. rollbackFor：回滚

设置哪些异常进行事务回滚

1. noRollbackFor：不回滚

设置哪些异常不进行回滚

### 事务操作（XML声明式事务管理）

1. 在spring配置文件中进行配置

第一步：配置事务管理器

第二步：配置通知

第三步：配置切入点和切面

|  |
| --- |
| <!--开启组件的扫描-->  <conetext:component-scan base-package="com.ly"></conetext:component-scan>  <!-- 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="123456" />  <property name="driverClassName" value="com.mysql.jdbc.Driver" />  </bean>  <!--jdbcTemplate对象-->  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <!--注入dataSource-->  <property name="dataSource" ref="dataSource"></property>  </bean>  <!--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-->  <tx:advice id="txadvice">  <!--配置事务的一些相关参数-->  <tx:attributes>  <!--z-->  <tx:method name="accountMoney" propagation="REQUIRED"/>  </tx:attributes>  </tx:advice>  <!--配置切入点和切面-->  <aop:config>  <aop:pointcut id="pt" expression="execution(\* com.ly.service.UserService.\*(..)"/>  <aop:advisor advice-ref="txadvice" pointcut-ref="pt"></aop:advisor>   </aop:config> |

### 事务操作（完全注解开发声明式事务管理）

1.创建一个配置类，使用配置类代替xml配置文件

|  |
| --- |
| @Configuration//配置类 @ComponentScan(basePackages = "com.ly")//开启组件的扫描 @EnableTransactionManagement //开启事务 public class TxConfig {  //创建数据库的连接池  @Bean  public DruidDataSource geDruidDataSource(){  DruidDataSource dataSource=new DruidDataSource();  dataSource.setDriverClassName("com.mysql.jdbc.Driver");  dataSource.setUrl("mysql://ldbc:3306///user\_db");  dataSource.setUsername("root");  dataSource.setPassword("123456");  return dataSource;  }  //创建jdbcTemplate对象  @Bean  public JdbcTemplate getJdbcTemplate(DataSource dataSource){  //到IOC容器中根据类型找到要注入的数据  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 dataSourceTransactionManager=new DataSourceTransactionManager();  dataSourceTransactionManager.setDataSource(dataSource);  return dataSourceTransactionManager;  } } |

# Spring5框架新功能

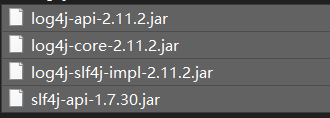
1. 整个Spring5框架的代码基于java8,运行时候兼容JDK9，许多不建议使用的类和方法在代码库中删除

## Spring5.0框架自带了通用的日志封装

（1）Spring5已经移除了Log4jConfigListener，官方建议用Log4j2

（2）Spring5框架整合Log4j2

第一步：引入jar包



第二步：创建Log4j2.xml文件配置文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <!--日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL --> <!--Configuration后面的status用于设置log4j2自身内部的信息输出，可以不设置，当设置成trace时，可以看到log4j2内部各种详细输出--> <configuration status="INFO">  <!--先定义所有的appender-->  <appenders>  <!--输出日志信息到控制台-->  <console name="Console" target="SYSTEM\_OUT">  <!--控制日志输出的格式-->  <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>  </console>  </appenders>  <!--然后定义logger，只有定义了logger并引入的appender，appender才会生效-->  <!--root：用于指定项目的根日志，如果没有单独指定Logger，则会使用root作为默认的日志输出-->  <loggers>  <root level="info">  <appender-ref ref="Console"/>  </root>  </loggers> </configuration> |

## Spring5框架核心容器支持@Nullable注解

（1）@Nullable注解可以使用在方法上面，属性上面，参数上面，表示方法返回可以为空，属性值可以为空，参数值可以为空

(2)注解用在方法上，方法返回值可以为空



1. 注解使用在方法参数里面，表示参数可以为空



1. 使用在属性上面，表示属性值可以为空
2. @Nullable  
   private String bookName;

## Spring5核心容器支持函数式风格GenericApplicationContext

|  |
| --- |
| @Test  public void testGenericApplicationContext(){  //创建GenericApplicationContext对象  GenericApplicationContext genericApplicationContext=new AnnotationConfigApplicationContext();  //2.调用context的方法对象注册  genericApplicationContext.refresh();  genericApplicationContext.registerBean("user1",User.class,()->new User());  //3.获取在spring注册的对象  Object bean = genericApplicationContext.getBean("user1");  System.*out*.println(bean);  }  } |

## Spring5支持整合Junit5

### 整合junit4

第一步 引入Spring相关针对测试依赖



第二步：创建测试类，使用注解方式完成

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class)//测试单元框架 @ContextConfiguration("classpath:bean1.xml")//加载配置文件 public class JTest4 {  @Autowired  private UserService userService;  @Test  public void test1(){  userService.accountMoney();  } } |

### 整合Junit5

第一步：引入1Junit5的jar包



第二步：创建测试类，使用注解方式完成

|  |
| --- |
| @ExtendWith(SpringExtension.class)//注解引入 @ContextConfiguration("classpath:bean1.xml")//加载配置文件 public class JTest5 {  @Autowired  private UserService userService;  @Test  public void test2(){  userService.accountMoney();  } } |

使用一个符合注解代替上面两个注解完成整合

|  |
| --- |
| @SpringJUnitConfig(locations = "classpath:bean1.xml") public class JTest5 {  @Autowired  private UserService userService;  @Test  public void test2(){  userService.accountMoney();  } } |

## Spring框架新功能：SpringWebflux

### SpringWebflux介绍

* 是Spring5添加的模块，用于web开发，功能与SpringMVC类似的，前一种比较流行响应式编程
* 使用传统web框架，比如SpringMVC,这些基于Servlet容器，Webflux是一种异步非阻塞的框架，异步非阻塞的框架在Servlet3.1以后才支持，核心是基于Reactor的相关API实现的
* 异步非阻塞 异步和同步针对调用者：调用者发送请求，如果等着对方回应之后才去做其他的事情就是同步，如果发送完请求之后不等着对方回应就去做其他的事情就是异步
* 阻塞和非阻塞：针对被调用者而言，被调用者受到请求之后，做完请求之后才给出反馈就是阻塞，如果受到请求之后马上给对方做出反馈然后再去做其他的事情就是非阻塞
* Webflux特点：

1. 非阻塞式：在我们有限的资源下，提高系统吞吐量和伸缩性
2. 函数式编程：Spring5框架基于java8，Webflux使用java8函数式编程方式实现路由请求
3. 比较SpringMVC和SpringWebflux

第一：两个框架都可以使用注解方式，都运行在Tomcat中

第二：SpringMVC采用命令式编程，Webflux采用异步响应式编程

### 响应式编程

1. 什么是响应式编程：响应式编程是面向数据流和变化传播的编程范式，这意味着可以在编程语言中很方便地表达静态或者动态的数据流，而相关的计算机模型会自动将变化的信道通过数据流进行传播

### Webflux执行流程和核心API

### SpringWebflux（基于注解编程模型）

### SpringWebflux（基于函数式编程模型）