# Spring学习笔记

spring xml配置文件，注解，都是与容器交流的语言。

让容器知道自己可以管理什么，怎么管理，制定好容器可以理解的规则，让容器去按照规则去执行。容器提供了规则的规范，我们按照这个规范去编写这个规则。

## spring的核心

IOC和AOP

IOC：依赖倒置，控制反转。把创建对象的过程（原来由使用的类进行创建）和对象之间的调用过程交给spring容器管理。降低代码之间的耦合度。

AOP: 面向切面（编程）。不修改的前提下，对代码功能进行增强。

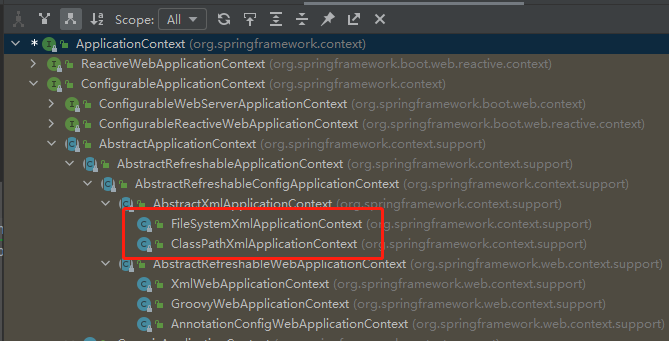
## 特点

1. 方便解耦，简化开发。
2. 方便程序的测试。
3. 方便继承各种框架。
4. 降低java API的使用难度。
5. 方便进行事务的操作。
6. Java源码的学习范例。

## 1：IOC底层原理

1. xml解析，工厂模式，反射：通过得到字节码（class）文件来实例化类。
2. IOC底层，就是用来创建对象的工厂。
3. Spring提供了IOC容器实现的两种方式：（两个接口）
   1. BeanFactory:IOC容器的基本实现，是spring内部自己使用的接口，不提供给开发人员使。
      1. 加载配置文件时不会创建对象，在获取（使用）对象的时候才去创建对象。
   2. ApplicationContext：BeanFactory接口的一个子接口，提供了更多更强大的功能，一般是提供给开发人员使用的。
      1. 加载配置的时候，就会创建配置文件中的对象。

C）ApplicationContext接口有实现类：



标红处为常用的容器入口，还有其他的容器入口。

## 1：容器怎么实例化类？怎么生成对象？

## 2：IOC操作，Bean管理

### 什么是Bean管理

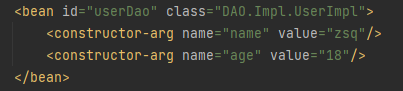
Bean管理指的是两个操作：

1：对象的创建。2：属性的注入。

## 2：类（bean）的scope属性

<bean id="userDao" class="DAO.Impl.UserImpl" scope="singleton"/>

默认使用无参的构造函数。（PS：为什么，代码里是怎么做的？）



为有参构造函数指定参数值。

|  |
| --- |
| public class UserImpl implements IUserDAO {  public String name;  public String age;   public UserImpl(String name, String age) {  this.name = name;  this.age = age;  System.*out*.println(this.toString());  }  } |
|  |

用有参构造函数实例化类

字面量：值为定值的变量。如int a=10;

singleton:单例模式

<bean id="userDao" class="DAO.Impl.UserImpl" scope="prototype"/>

prototype：多例模式

|  |  |  |
| --- | --- | --- |
| 取值范围 | 说明 | 备注 |
| singleton | 默认值，单例的 |  |
| prototype | 多例的 |  |
| request | Web项目中，spring创建一个Bean对象，将对象存到request域中 | 为什么ideal只是自动提示前两种，是不是后三种已经过时了 |
| session | Web项目中，spring创建一个Bean对象，将对象存到session域中 | 为什么ideal只是自动提示前两种，是不是后三种已经过时了 |
| globalSession | Web项目中，应用在Portlet环境(PS:什么是Portlet环境)，如果没有Portlet环境，那么globalSession,相当于session | 为什么ideal只是自动提示前两种，是不是后三种已经过时了 |

## 3：singleton和prototype的类，生命周期不一样

scope=”singleton”时，容器加载后就已经实例。容器销毁，实例销毁

scope=”prototype”时，需要获得类的实例时才会实例化。不再被引用，实例被销毁（PS:JVM的垃圾回收机制）。

## 4: bean的生命周期

1. 执行无参构造函数创建类的实例
2. 调用set方法设置属性值
3. （增加了Bean的后置处理器的话）把bean的实例传递给bean的后置处理器的方法
4. 执行bean的初始化方法
5. 增加了Bean的后置处理器的话）把bean的实例传递给bean的后置处理器的方法
6. 获取创建好的Bean的是实例。
7. 当容器关闭的时候，调用bean的销毁的方法（如果配置了销毁方法的话）。

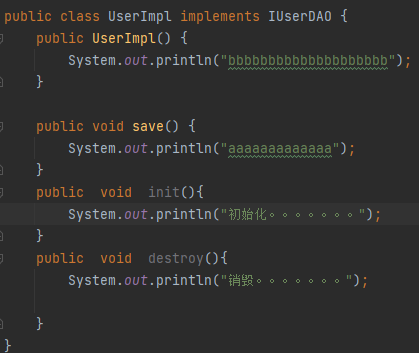
### Bean的后置处理器

增加bean的后置处理器后，bean的生命周期有7步。

不加Bean的后置处理器，bean的生命周期有5步。

|  |  |
| --- | --- |
| 属性 | 说明 |
| init-method | 指定类的初始化方法名称(PS:在构造函数后执行) |
| destroy-method | 指定类的销毁方法名称 |





## 5：容器的生命周期

context.close();//主动的销毁容器自己。可以销毁改销毁的东西。

## 6: bean实例化的三种方式

1. 无参构造方法实例化
2. 工厂静态方法实例化
3. 工厂实例方法实例化 （PS:设计模式->工厂方法）

PS:为啥ideal的Ctrl+H出现的 继承关系图，有的完整，有的不完整。

## 7：Spring注解

Spring 种针对Bean管理中创建对象提供的注解：

1. @Component :spring容器中一种普通的组件，使用它都可以实现将依赖交给容器
2. @Service：该注解使用在业务层的类上。
3. @Controller：一般使用在web层上
4. @Repository：一般使用在DAO层或者持久层

以上注解都可以将对象实例交给容器创建，做名字的区分是在逻辑上更加清晰，更加利于开发。

|  |
| --- |
| *//@Component(value = "userDao") //@Controller(value = "userDao") //@Service(value = "userDao")* @Repository(value = "userDao")  后三个注解，等同于@Component注解，功能上表示是一个组件，名称是为了逻辑清晰。  在注解里，value值可以省略不写，默认是类的名称，首字母小写。 |

|  |  |
| --- | --- |
| 注解 | 说明 |
| @AutoWired | 根据类型注入属性，首字母小写 |
| @Qualifier | 根据名称注入属性，和@AutoWired一起使用 |
| @Resource | 可以根据类型（@Resource）也可以根据名称(@Resource(name=”asdf”))注入.(PS:spring官方不推荐使用) |
| @Value("") | 给普通属性（字段）注入值。字面量 |
| @Configuration | 表示该类作为配置类，代替Application.xml等配置文件 |

Xml配置开发

|  |
| --- |
| <bean name="userDao" class="DAO.Impl.UserImpl"/>  *<!-- use-default-filters=“false”,表示现在不使用默认的filter,自己配置filter. context:include-filter：设置扫描那些内容 context:exclude-filter：设置不扫描那些内容 -->* <context:component-scan base-package="DAO" use-default-filters="false">  <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>  <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Service"/> </context:component-scan> |

纯注解开发

|  |
| --- |
| @Configuration*//表示该类是一个配置类* @ComponentScan(basePackages = {"DAO"})*//配置扫描的基础包* public class ServerConfig { }  使用该类获取容器上下文，来获得类的实例对象。  AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ServerConfig.class); |

# AOP（底层原理）

1. AOP使用动态代理

有两种情况的动态代理

1. 有接口情况，使用JDK的动态代理
2. 没有接口情况，使用CGLIB的动态代理

Java反射

|  |
| --- |
| **反射机制的概念:**  指在运行状态中,对于任意一个类,都能够知道这个类的所有属性和方法,对于任意一个对象,都能调用它的任意一个方法.这种动态获取信息,以及动态调用对象方法的功能叫java语言的反射机制.  **反射机制的应用:**  生成动态代理,面向切片编程(在调用方法的前后各加栈帧).  **反射机制的原理:**  1 首先明确的概念: 一切皆对象----类也是对象.  2 然后知道类中的内容 :modifier constructor field method.  3 其次明白加载: 当Animal.class在硬盘中时,是一个文件,当载入到内存中,可以认为是一个对象,是java.lang.class的对象.  **java 获取反射常使用的三种方式：**  1.通过new对象实现反射机制  2.通过路径实现反射机制  3.通过类名实现反射机制   1. 通过Object类的getClass方法来获取 2. 使用.class的方式 3. 使用Class.forName方法 |

建立一个学生类

|  |
| --- |
| public class Student {  private int id;  String name;  protected boolean sex;  public float score;  } |

通过反色机制获取对象

|  |
| --- |
| public class Get {  //获取反射机制三种方式  public static void main(String[] args) throws ClassNotFoundException {  //方式一(通过建立对象)  Student stu = new Student();  Class classobj1 = stu.getClass();  System.out.println(classobj1.getName());    //方式二（所在通过路径-相对路径）  Class classobj2 = Class.forName("fanshe.Student");  System.out.println(classobj2.getName());    //方式三（通过类名）  Class classobj3 = Student.class;  System.out.println(classobj3.getName());  }  } |

### JDK反射实现代理模式

### 有接口的，用JKD动态代理实现

首先创建一个接口

|  |
| --- |
| public interface IUserDAO {  public int save(int a,int b); } |

实现接口的类，被代理的类。

|  |
| --- |
| @Service(value = "userDao")public class UserImpl implements IUserDAO {  public UserImpl() {  System.*out*.println("bbbbbbbbbbbbbbbbbbbb");  }  public int save(int a, int b) {  System.*out*.println("aaaaaaaaaaaaa");  return a + b;  } } |

来写代理类

|  |
| --- |
| public class UserProxy implements InvocationHandler {  Object user;*//被代理对象*  *//有参构造函数为被代理对象实例化* public UserProxy(Object user) {  this.user = user;  } */\*\*  \* 实现 接口InvocationHandler必须重写的invoke()方法  \* @param proxy ,真实的代理对象  \* @param method ,要执行的方法信息  \* @param args ,执行方法需要的参数  \* @return 返回被代理对象执行方法的结果  \*/* @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  System.*out*.println("执行代理方法前的增强操作");  Object a = method.invoke(user, args);*//调用被代理对象的方法* System.*out*.println("执行代理方法后的增强操作");  return a;*//返回被代理对象方法执行的返回结果* }   public static void main(String[] args) {  Class[] interfaces = {IUserDAO.class};  IUserDAO user = (IUserDAO) Proxy.*newProxyInstance*(UserProxy.class.getClassLoader(), interfaces, new UserProxy(new UserImpl()));  System.*out*.println(user.save(1, 2));  } } |
|  |

|  |
| --- |
| public static void main(String[] args) {  Class[] interfaces = {IUserDAO.class};  IUserDAO user = (IUserDAO) Proxy.*newProxyInstance*(UserProxy.class.getClassLoader(), IUserDAO.class.getInterfaces(), new UserProxy(new UserImpl()));  System.*out*.println(user.save(1, 2)); } |

获得接口数组的方式

|  |  |
| --- | --- |
| 第一种 | Class[] interfaces = {IUserDAO.class}; |
| 第二种 | IUserDAO.class.getInterfaces() |

无接口的CGLIB动态代理实现思想

1：被代理的类

|  |
| --- |
| public class User{  public void add(){} } |

2：代理类

|  |
| --- |
| public class People extends User {  public User user;  public People(User user) {  this.user = user;  }  public void add() {  *//增强方法* super.add();  } } |

# InvocationHandler接口invoke（）方法的第一个参数proxy的真正含义

上边内容详细的讲解了创建代理类的调用处理程序（实现InvocationHandler接口的类），获得代理对象的Proxy类，但是就发现InvocationHandler中的invoke方法中的第一个参数proxy好像从来没有用过，所以就开始在网上查询proxy的用途，

## 1.讲解前我们先列一下我们要说明的问题

* proxy代表什么意思
* proxy参数怎么用及什么时候用
* proxy参数运行时的类型是什么
* 为什么不用this代替proxy

## 2.proxy代表什么意思

proxy是真实对象的真实代理对象，invoke方法可以返回调用代理对象方法的返回结果，也可以返回对象的真实代理对象（com.sun.proxy.$Proxy0）。

## 3.proxy参数怎么用及什么时候用

proxy参数是invoke方法的第一个参数，通常情况下我们都是返回真实对象方法的返回结果，但是我们也可以将proxy返回，proxy是真实对象的真实代理对象，我们可以通过这个返回对象对真实的对象做各种各样的操作。

* 创建一个接口People，包含一个work方法，方法的返回对象是它本身

|  |
| --- |
| public interface People {   public People work(String workName);  public String time(); } |

* 创建一个接口People的实现类Student

|  |
| --- |
| public class Student implements People{  @Override  public People work(String workName) {  System.*out*.println("工作内容是"+workName);  return this;  }  @Override  public String time() {  return "2018-06-12";  } } |

* 创建一个代理类的调用处理程序WorkHandler

|  |
| --- |
| public class WorkHandler implements InvocationHandler{  private Object obj;  public WorkHandler(Object obj) {  this.obj = obj;  }  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  System.*out*.println("before 动态代理...");  System.*out*.println(proxy.getClass().getName());  System.*out*.println(this.obj.getClass().getName());  if(method.getName().equals("work")) {  method.invoke(this.obj, args);  System.*out*.println("after 动态代理...");  return proxy;  } else {  System.*out*.println("after 动态代理...");  return method.invoke(this.obj, args);  }  } } |

* 客户端实例创建代理对象并输出结果

|  |
| --- |
| public class Test {  public static void main(String[] args) {  People people = new Student();  InvocationHandler handler = new WorkHandler(people);   People proxy = (People)Proxy.*newProxyInstance*(people.getClass().getClassLoader(), people.getClass().getInterfaces(), handler);  People p = proxy.work("写代码").work("开会").work("上课");  System.*out*.println("打印返回的对象");  System.*out*.println(p.getClass());  String time = proxy.time();  System.*out*.println(time);  } } |

我们可以看到上面的代理类调用处理程序打印了proxy参数对象，并且返回了proxy对象。

运行结果：

|  |
| --- |
| after 动态代理...  before 动态代理...  com.sun.proxy.$Proxy0  com.test.Application.Student  工作内容是上课  after 动态代理...  打印返回的对象  class com.sun.proxy.$Proxy0  class com.sun.proxy.$Proxy0  before 动态代理...  com.sun.proxy.$Proxy0  com.test.Application.Student  after 动态代理... |

我们可以看到WorkHandler代理调用处理程序打印proxy参数输出的结果是com.sun.proxy.$Proxy0，这也说明proxy参数是代理类的真实代理对象；Proxy类生成的代理对象可以调用work方法并且返回真实的代理对象，也可以通过反射来对真实的代理对象进行操作。

## 4.proxy参数运行时的类型是什么

上面我们已经打印出了proxy的类型是：com.sun.proxy.$Proxy0真实的代理对象

## 5.为什么不用this替代

因为this代表的是InvocationHandler接口实现类本身，并不是真实的代理对象。

# AOP术语

* 1. 连接点：类里边哪些方法可以被增强，这些方法称为连接点。
  2. 切入点：实际被增强的方法，称为切入点。
  3. 通知（增强 ）：增强的逻辑部分称为通知（增强）。
     + 通知的多种类型
       1. 前置通知
       2. 后置通知
       3. 环绕通知：切入点执行签后都执行
       4. 异常通知
       5. 最终通知：类似finally,无论如何，最终都会执行。
  4. 切面：是一种动作，把通知应用到切入点的过程。

# 什么是AspectJ

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

## 基于AspectJ实现AOP操作

1. 基于xml配置文件
2. 基于注解方式实现

切入点表达式

作用：知道对哪个类里的哪个方法进行增强。

语法：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：对类里的add方法进行增强

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

举例2：对类里的所有方法进行增强。

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

举例3：对com.dao包里所有的类的所有的方法进行增强

Execution( \* com.dao.\*.\*(..) )

增强配置步骤

1：开启AspectJ的自动代理

|  |
| --- |
| @Configuration*//表示该类是一个配置类* @ComponentScan(basePackages = {"DAO"})*//配置扫描的基础包* @EnableAspectJAutoProxy *//开启AspectJ 的自动代理* public class ServerConfig { } |

2：编写增强类

|  |
| --- |
| @Component @Aspect *//生成代理对象* public class AdviceConfig {  */\*\*  \* 前置通知  \*/*  public void before(){  System.*out*.println("对方法进行前置增强");  } } |

3:设置通知类型和切点表达式

|  |
| --- |
| @Component @Aspect *//生成代理对象* public class AdviceConfig {  */\*\*  \* 前置通知  \*/* @Before(value = "execution(\* DAO.\*.\* (..))")  public void before(){  System.*out*.println("对方法进行前置增强");  } } |

被增强类

|  |
| --- |
| @Component(value = "userDao") public class UserImpl implements IUserDAO {  public UserImpl() {  }  public int save(int a, int b) {  return a + b;  }  } |

4：运行代码，查看结果

|  |
| --- |
| public class Test {  @org.junit.Test  public void test1(){  { AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ServerConfig.class);  IUserDAO userDao =(IUserDAO) context.getBean("userDao");  System.*out*.println(userDao.save(1,2));  }  }  } |

运行结果：

|  |
| --- |
| 对方法进行前置增强  3 |

全部5种增强

|  |  |  |
| --- | --- | --- |
| 注解 | 说明 | 执行顺序 |
| @Before |  |  |
| @After |  |  |
| @Around | 焕然增强，比较特殊，需要有参数，且可以有返回对象 |  |
| @AfterReturning | 返回后增强 |  |
| @AfterThrowing | 抛错后增强 |  |

|  |
| --- |
| @Component @Aspect *//生成代理对象* public class AdviceConfig {  @Before(value = "execution(\* DAO.\*.\* (..))")  public void before() {  System.*out*.println("对方法进行前置增强");  }   @After(value = "execution(\* DAO.\*.\* (..))")  public void after() {  System.*out*.println("对方法进行后置增强");  }   @Around(value = "execution(\* DAO.\*.\* (..))")  public Object around(ProceedingJoinPoint joinPoint) throws Throwable {  System.*out*.println("对方法进行环绕增强前");  Object obj = joinPoint.proceed();*//执行被增强的方法* System.*out*.println("对方法进行环绕增强后：" + obj);  return obj;  }   @AfterReturning(value = "execution(\* DAO.\*.\* (..))")  public void affterReturn() {  System.*out*.println("对方法进行返回后增强");  }   @AfterThrowing(value = "execution(\* DAO.\*.\* (..))")  public void afterThrow() {  System.*out*.println("对方法进行抛错后增强");   } } |

所有增强的执行结果(没有抛错的情况下)

|  |  |
| --- | --- |
| 对方法进行环绕增强前  对方法进行前置增强  对方法进行返回后增强  对方法进行后置增强  对方法进行环绕增强后：3 | 1:@Around  2:@Before  3:@ AfterReturning  4:@Around  5:@ After |

所有增强的执行结果(没有抛错的情况下)（PS:抛错会打断环绕增强？）

|  |  |
| --- | --- |
| 对方法进行环绕增强前  对方法进行前置增强  对方法进行抛错后增强  对方法进行后置增强  java.lang.ArithmeticException: / by zero | 1:@Around  2:@Before  3:@AfterThrowing  4:@After |

公共切入点抽取

|  |
| --- |
| @Component @Aspect *//生成代理对象* public class AdviceConfig {  */\*\*  \* 公共切入点抽取  \*/* **@Pointcut(value = "execution(\* DAO.\*.\* (..))")  public void execut(){  }**  @Before(value = "execut()")  public void before() {  System.*out*.println("对方法进行前置增强");  }   @After(value = "execut()")  public void after() {  System.*out*.println("对方法进行后置增强");  }   @Around(value = "execut()")  public Object around(ProceedingJoinPoint joinPoint) throws Throwable {  System.*out*.println("对方法进行环绕增强前");  Object obj = joinPoint.proceed();*//执行被增强的方法* System.*out*.println("对方法进行环绕增强后：" + obj);  return obj;  }   @AfterReturning(value = "execut()")  public void affterReturn() {  System.*out*.println("对方法进行返回后增强");  }   @AfterThrowing(value = "execut()")  public void afterThrow() {  System.*out*.println("对方法进行抛错后增强");   } } |

## 为增强类设置优先级

1．有多个增强类对一个方法进行增强，可以通过类上加注解@Order(数字类型值)来设置优先级。

数字值越小，优先级越高。值从0开始。

# Spring 整合jdbc

返回对象

|  |
| --- |
| public User getUser(int id){  String sql="select \* from userinfo where id=?";  User user=jdbcTemplate.queryForObject(sql,new BeanPropertyRowMapper<User>(User.class), BigDecimal.*valueOf*(id));  return user; } |

返回集合

|  |
| --- |
| public List<User> getUsers(){  String sql="select \* from userinfo ";  List<User> users=jdbcTemplate.query(sql,new BeanPropertyRowMapper<User>(User.class));  return users; } |

批量操作数据库：不是原子操作

|  |
| --- |
| public void batchAdd(List<Object[]> batchArgs){  String sql="insert into userinfo values(?,?,?);";  int[] ints=jdbcTemplate.batchUpdate(sql,batchArgs);  System.*out*.println(String.*valueOf*(ints)); } |

# Jdbc的事务

## 事务的四个特性（ACID）

* 1. 原子性
  2. 一致性
  3. 隔离性
  4. 持久性

1：事务一般添加到Service层

2：在spring 中进行事务管理操作

1），有两种方式：编程式事务管理和声明式事务管理（推荐）。

· 2)，声明式事务管理

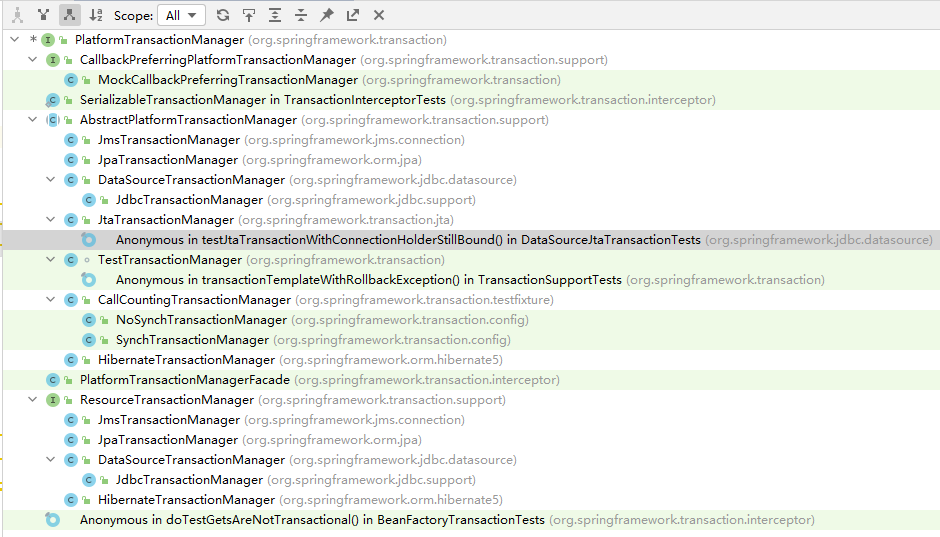
基于注解的方式。

基于xml配置文件的方式。

## 3：在Spring中进行声明式事务管理，底层使用AOP。

## 4：Spring中事务管理的API.

1),提供一个接口，代表事务管理器，这个接口根据不通的框架提供不同的实现类。



## 事务操作（注解实现声明式事务管理）

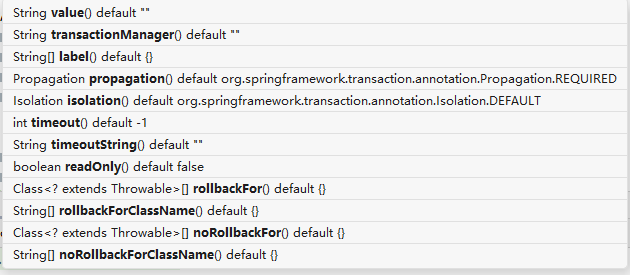
|  |
| --- |
| 1，在配置文件中配置事务管理器  *<!--创建事务管理器-->* <bean id="transaction" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  *<!--注入数据源-->* <property name="dataSource" ref="dataSource"/> </bean>  2，在spring配置文件中，开启事务注解  1）,在spring的配置文件中，引入事务的名称空间tx.  2),开启事务管理器注解。  *<!--开启事务注解-->* <tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-driven>   1. 在Service层的类上边或者类里的方法上边添加事务注解   @Transactional |

## 事务操作（声明式事务管理参数配置）(PS:ctrl+p,自动出现可配置参数)

@Transactional 注解详解

### value 和 transactionManager 属性

它们两个是一样的意思。当配置了多个事务管理器时，可以使用该属性指定选择哪个事务管理器。



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| propagation:事务传播行为  多事务方法同时进行调用，这个过程事务是如何进行管理的。  解决的问题：  1：当一个事务方法（添加了@Transactional注解的方法），调用一个没有添加事务注解的方法。如果事务方法（add()）报错了，没有添加事务注解的update()方法，如何处理？  2：当一个没有事务方法，调用了有事务方法，其中一个报错了，该 如何处理？  3：当两个方法都有事务管理，其中一个报错了，该如何处理？    Spring框架中，有7种事务传播行为   |  |  |  | | --- | --- | --- | | 序号 | 传播属性 | 说明 | | 1 | REQUIRED 默认使用该行为！ | 如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新事务，并在自己的事务内运行。（PS：这俩事务的执行顺序是什么？） | | 2 | REQUIRED\_NEW | 当前的方法必须启动新事务，并在它自己的事务内运行，如果有事务在运行，应该将它挂起。（PS：这俩事务的执行顺序是什么？） | | 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。（PS：事务和方法的执行顺序是什么？） | | 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中，如果有运行的事务，就抛出异常。 | | 7 | NESTED | 如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行，否则，就启动一个新的事务，并在它自己的事务内运行。PS：这俩事务的执行顺序是什么？） |   1：REQUIRED  如果add方法本身有事务，调用update方法后，update使用当前add方法里的事务。  如果add方法本身没有事务，调用update方法后，update创建一个新的的事务使用。  2：REQUIRED\_NEW  使用add方法调用update方法，无论add方法是否有事务，都创建新的事务执行。如果add有事务，将add事务先挂起（PS：是这样吗？）。  事务传播行为的配置方式：  @Transactional(propagation = Propagation.*REQUIRED*,isolation = Isolation.*DEFAULT*)  isolation:事务的隔离级别  解决多并发可能产生的三个问题：脏读，不可重复读，虚（幻）读。  脏读：（PS：是一个致命问题）一个未提交的事务读取到另一个未提交事务的数据。    不可重复读：（PS:是一种现象）一个未提交的事务，读取了一个已经提交事务的数据，很可能导致每次读的数据都不一样。  "不可重复读" 是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。   虚（幻）读： 幻觉读是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。  脏读、幻读和不可重复读的相关资料：  <https://www.cnblogs.com/itcomputer/articles/5133254.html>  <https://zhuanlan.zhihu.com/p/150107974>   |  |  | | --- | --- | | 隔离级别 | 说明 | | DEFAULT |  |     配置方式：  @Transactional(propagation = Propagation.*REQUIRED*,isolation = Isolation.*DEFAULT*)  timeout:超时时间  1），事务需要在超时时间内进行提交，如果超时没有提交，则进行回滚。  2），默认值是-1，也就是不超时。设置时间以秒为单位进行计算。  注解配置方式：  @Transactional(timeout = 10)  readOnly:是否只读  1），读：查询操作。写：添加、修改、更新操作。  2），readOnly默认值为false,表示可以增删查改。  3），设置readOnly值为true，表示只可以查。  注解配置方式：  @Transactional(readOnly = true)  rollbackFor:回滚  1） 设置出现哪些异常可以进行回滚  2）注解的配置方式  @Transactional(rollbackFor = {Exception.class})  noRollbackFor:不回滚  1）设置出现哪些异常可以不进行回滚  2）注解的配置方式  @Transactional(noRollbackFor = {IndexOutOfBoundsException.class})  （PS：IndexOutOfBoundsException.class:只是为了演示配置使用方式，不作为使用建议）  rollbackFor和noRollbackFor的相关资料  <https://www.cnblogs.com/clwydjgs/p/9317849.html> |

### Xml配置方式配置事务

|  |
| --- |
| *<!-- 配置通知 -->* <tx:advice id="userAdvice">  <tx:attributes>  *<!--在方法上配置事务-->* <tx:method name="getUser" propagation="REQUIRES\_NEW"/>  <tx:method name="batchAdd" propagation="REQUIRES\_NEW"/>  </tx:attributes> </tx:advice> *<!--配置AOP-->* <aop:config>  *<!--配置切点-->* <aop:pointcut id="pt" expression="execution( \* DAO.service.UserService.getUser(..))"/>  *<!--配置通知者-->* <aop:advisor advice-ref="userAdvice" pointcut-ref="pt"/> </aop:config> |

### @Bean注解

|  |
| --- |
| @Bean 的用法 <https://www.cnblogs.com/feiyu127/p/7700090.html>  @Bean是一个方法级别上的注解，主要用在@Configuration注解的类里，也可以用在@Component注解的类里。添加的bean的id为方法名 定义bean 下面是@Configuration里的一个例子  @Configuration public class AppConfig {  @Bean  public TransferService transferService() {  return new TransferServiceImpl();  } }  这个配置就等同于之前在xml里的配置bean的依赖  <beans> <bean id="transferService" class="com.acme.TransferServiceImpl"/> </beans>  @bean 也可以依赖其他任意数量的bean，如果TransferService 依赖 AccountRepository，我们可以通过方法参数实现这个依赖  @Configuration public class AppConfig {  @Bean  public TransferService transferService(AccountRepository accountRepository) {  return new TransferServiceImpl(accountRepository);  } } 接受生命周期的回调 任何使用@Bean定义的bean，也可以执行生命周期的回调函数，类似@PostConstruct and @PreDestroy的方法。用法如下  public class Foo {  public void init() {  *// initialization logic* } }  public class Bar {  public void cleanup() {  *// destruction logic* } }  @Configuration public class AppConfig {   @Bean(initMethod = "init")  public Foo foo() {  return new Foo();  }   @Bean(destroyMethod = "cleanup")  public Bar bar() {  return new Bar();  }  }  默认使用javaConfig配置的bean，如果存在close或者shutdown方法，则在bean销毁时会自动执行该方法，如果你不想执行该方法，则添加@Bean(destroyMethod="")来防止出发销毁方法 指定bean的scope使用@Scope注解 你能够使用@Scope注解来指定使用@Bean定义的bean  @Configuration public class MyConfiguration {  @Bean  @Scope("prototype")  public Encryptor encryptor() {  } }  @Scope and scoped-proxy  spring提供了scope的代理，可以设置@Scope的属性proxyMode来指定，默认是ScopedProxyMode.NO， 你可以指定为默认是ScopedProxyMode.INTERFACES或者默认是ScopedProxyMode.TARGET\_CLASS。 以下是一个demo，好像用到了（没看懂这块）  *// an HTTP Session-scoped bean exposed as a proxy* @Bean @SessionScope public UserPreferences userPreferences() {  return new UserPreferences(); }  @Bean public Service userService() {  UserService service = new SimpleUserService();  *// a reference to the proxied userPreferences bean* service.setUserPreferences(userPreferences());  return service; } 自定义bean的命名 默认情况下bean的名称和方法名称相同，你也可以使用name属性来指定@Configuration public class AppConfig {  @Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })  public DataSource dataSource() {  *// instantiate, configure and return DataSource bean...* } } bean的别名 bean的命名支持别名，使用方法如下 bean的描述 有时候提供bean的详细信息也是很有用的，bean的描述可以使用 @Description来提供  @Configuration public class AppConfig {  @Bean  @Description("Provides a basic example of a bean")  public Foo foo() {  return new Foo();  } } |

# Spring 5 新功能

<https://cntofu.com/book/95/33-what-new-in-the-spring-framework.md>

1：整个Spring 5框架的代码基于JAVA 8，运行时兼容JDK9，许多不建议使用的类和方法在代码库种删除。

2：Spring 5 框架自带了通用的日志封装.

1): Spring 5已经移除了Log4jConfigListener, Log4jConfigListener用来进行log4j日志整合.现在官方建议使用log4j2的版本.

## 整合log4j2日志

Pom.xml引入 依赖

|  |
| --- |
| <dependency>  <groupId>org.apache.logging.log4j</groupId>  <artifactId>log4j-api</artifactId>  <version>2.11.0</version> </dependency> <dependency>  <groupId>org.apache.logging.log4j</groupId>  <artifactId>log4j-core</artifactId>  <version>2.11.0</version> </dependency> |

Resource目录下创建固定名称的文件:log4j2.xml

|  |
| --- |
| *<?*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，这个用于设置log4j2自身内部的信息输出，可以不设置，当设置成trace时，你会看到log4j2内部各种详细输出-->  <!--monitorInterval：Log4j能够自动检测修改配置 文件和重新配置本身，设置间隔秒数-->* <configuration status="ALL" monitorInterval="30">  *<!--先定义所有的appender-->* <appenders>  *<!--输出控制台的配置-->* <console name="Console" target="SYSTEM\_OUT">  *<!--输出日志的格式-->* <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>  </console>   *<!--文件会打印出所有信息，这个log每次运行程序会自动清空，由append属性决定，这个也挺有用的，适合临时测试用-->* <File name="log" fileName="log/test.log" append="false">  <PatternLayout pattern="%d{HH:mm:ss.SSS} %-5level %class{36} %L %M - %msg%xEx%n"/>  </File>   *<!-- 这个会打印出所有的info及以下级别的信息，每次大小超过size，则这size大小的日志会自动存入按年份-月份建立的文件夹下面并进行压缩，作为存档-->* <RollingFile name="RollingFileInfo" fileName="${sys:user.home}/logs/info.log"  filePattern="${sys:user.home}/logs/$${date:yyyy-MM}/info-%d{yyyy-MM-dd}-%i.log">  *<!--控制台只输出level及以上级别的信息（onMatch），其他的直接拒绝（onMismatch）-->* <ThresholdFilter level="debug" onMatch="ACCEPT" onMismatch="DENY"/>  <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>   <Policies>  <TimeBasedTriggeringPolicy/>  <SizeBasedTriggeringPolicy size="10 MB"/>  </Policies>  </RollingFile>   <RollingFile name="RollingFileWarn" fileName="${sys:user.home}/logs/warn.log"  filePattern="${sys:user.home}/logs/$${date:yyyy-MM}/warn-%d{yyyy-MM-dd}-%i.log">   <ThresholdFilter level="warn" onMatch="ACCEPT" onMismatch="DENY"/>  <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>  <Policies>  <TimeBasedTriggeringPolicy/>  <SizeBasedTriggeringPolicy size="100 MB"/>  </Policies>  *<!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7个文件，这里设置了20 -->* <DefaultRolloverStrategy max="20"/>  </RollingFile>   <RollingFile name="RollingFileError" fileName="${sys:user.home}/logs/error.log"  filePattern="${sys:user.home}/logs/$${date:yyyy-MM}/error-%d{yyyy-MM-dd}-%i.log">  <ThresholdFilter level="error" onMatch="ACCEPT" onMismatch="DENY"/>  <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>  <Policies>  <TimeBasedTriggeringPolicy/>  <SizeBasedTriggeringPolicy size="100 MB"/>  </Policies>  </RollingFile>   </appenders>   *<!--然后定义logger，只有定义了logger并引入的appender，appender才会生效-->* <loggers>   *<!--过滤掉spring和mybatis的一些无用的DEBUG信息-->* <logger name="org.springframework" level="INFO"/>  <logger name="org.mybatis" level="INFO"/>   <root level="all">  *<!--输出到控制台-->* <appender-ref ref="Console"/>  *<!--Info输出到文件-->* <appender-ref ref="RollingFileInfo"/>  *<!--Warn输出到文件-->* <appender-ref ref="RollingFileWarn"/>  *<!--Error输出到文件-->* <appender-ref ref="RollingFileError"/>  </root>   </loggers>  </configuration> |

# Spring web flux

## 说明

SpringMVC方式实现，同步阻塞的方式，基于SpringMVC+Servlet+Tomcat

SpringWebFlux方式实现，异步非阻塞方式，基于SpringWebFlux+Reactor+Netty

SpringWebFluc 比Spring MVC 同样的机器配置可以处理更多的数据。

SpringWebFlux（基于函数式编程模型）

1).在使用函数式编程模型操作的时候，需要自己初始化服务器。

2）基于函数式编程模型的时候，有两个核心接口：RouterFunction(实现路由功能，请求转发给对应的handler)和HandlerFunction(处理请求，处理响应的函数)。核心任务：定义两个函数式接口的实现并且启动需要的服务器。

3）.SpringWebFlux请求和响应不再是ServletRequset和ServletResponse. 而是ServerRequest和ServerResponse.

初始化服务器，编写Router.

User.java

|  |
| --- |
| public class User {  private String name;  private String gender;  private Integer age;   public User(String name, String gender, Integer age) {  this.name = name;  this.gender = gender;  this.age = age;  }   public String getName() {  return name;  }   public void setName(String name) {  this.name = name;  }   public String getGender() {  return gender;  }   public void setGender(String gender) {  this.gender = gender;  }   public Integer getAge() {  return age;  }   public void setAge(Integer age) {  this.age = age;  } } |

UserHandler.java

|  |
| --- |
| public class UserHandler {  private final UserService userService;   public UserHandler(UserService userService) {  this.userService = userService;  }   public Mono<ServerResponse> getUserById(ServerRequest request) {  *//获取Id值* int userId = Integer.*valueOf*(request.pathVariable("id"));  *//空值处理* Mono<ServerResponse> responseMono = ServerResponse.*notFound*().build();  *//调用service方法得到数据* Mono<User> userMono = this.userService.getUserById(userId);  *//把serMono进行转换返回  //使用Reactory操作符 flatMap* return userMono  .flatMap(person ->  ServerResponse.*ok*().contentType(MediaType.*APPLICATION\_JSON*)  .body(*fromObject*(person)))  .switchIfEmpty(responseMono);  }   *//查询所有* public Mono<ServerResponse> getAllUsers(ServerRequest request) {  *//调用service得到结果* Flux<User> users = this.userService.getAllUsers();  return ServerResponse.*ok*().contentType(MediaType.*APPLICATION\_JSON*).body(users, User.class);  }   *//添加* public Mono<ServerResponse> saveUser(ServerRequest request) {   *//得到user对象* Mono<User> userMono = request.bodyToMono(User.class);  return ServerResponse.*ok*().build(this.userService.saveUserInfo(userMono));  }  } |

UserServiceImpl.java

|  |
| --- |
| */\*\*  \* 该类模拟数据库存储  \*/* @Repository public class UserServiceImpl implements UserService {  *//创建一个Map集合，来模拟数据库操作* private final Map<Integer,User> users=new HashMap<>();   public UserServiceImpl() {  this.users.put(1,new User("q2we1","男",12));  this.users.put(2,new User("q2we2","男",12));  this.users.put(3,new User("q2we3","男",12));  }   @Override  public Mono<User> getUserById(int id) {  return Mono.*justOrEmpty*(this.users.get(id));  }   @Override  public Flux<User> getAllUsers() {  return Flux.*fromIterable*(this.users.values());  }   @Override  public Mono<Void> saveUserInfo(Mono<User> userMono) {  return userMono.doOnNext(person->{  *//向Map集合李放值* int id=this.users.size()+1;  users.put(id,new User("123","女",15));  }).thenEmpty(Mono.*empty*());  } } |

Router服务端，Server.java

|  |
| --- |
| public class Server {  */\*  创建Router路由  \*/* public RouterFunction<ServerResponse> routingFunction() {  UserService userService = new UserServiceImpl();  UserHandler handler = new UserHandler(userService);  *//设置路由* return RouterFunctions.*route*(  *GET*("/user/{id}").and(*accept*(*APPLICATION\_JSON*)), handler::getUserById)  .andRoute(*GET*("/users").and(*accept*(*APPLICATION\_JSON*)), handler::getAllUsers);  }  */\*  创建服务器完成适配  \*/* public void createReactoryServer() {  *//路由和handler适配* RouterFunction<ServerResponse> route = routingFunction();  HttpHandler httpHandler = *toHttpHandler*(route);  ReactorHttpHandlerAdapter handlerAdapter = new ReactorHttpHandlerAdapter(httpHandler);  *//创建服务器* HttpServer httpServer = HttpServer.*create*();  httpServer.handle(handlerAdapter).bindNow();  }   public static void main(String[] args) throws IOException {  Server server=new Server();  server.createReactoryServer();  System.*out*.println("enter to exit!");  System.*in*.read();  }  } |

WebClient 的调用方式：

|  |
| --- |
| public class Client {  public static void main(String[] args) {  *//调用服务器的地址* WebClient webClient = WebClient.create("http://localhost:63547");*//Server 的IP和地址   //根据Id 查询* String id = "1";  User user = webClient.get().uri("/user/{id}", id).accept(MediaType.APPLICATION\_JSON)  .retrieve().bodyToMono(User.class).block();  System.out.println(user.toString());  Flux<User> userFlux = webClient.get().uri("/users").accept(MediaType.APPLICATION\_JSON)  .retrieve().bodyToFlux(User.class);   userFlux.map(stu -> stu.toString()).buffer().doOnNext(System.out::println).blockFirst();  } } |

# Java是传值调用还是传引用调用

**java里边，参数传参的类型是传值调用而不是传引用调用  
 PS:即使是传对象的引用，也只是传了引用的副本，对引用本身的操作如果没有人接受的话，不可传递。  
 如果通过引用的副本，对指向的对象进行了操作，是可以通过其他引用访问时发现的。**