**一、Spring框架的概述、IOC(文件方式)、单例、多例、懒加载、DI(文件方式)**

1、框架的概述

所谓的框架其实就是程序的架子，在这个程序的架子中，搭建起程序的基本的骨架，针对程序的通用问题给出了便捷的解决方案，可以使开发人员基于框架快速开发具体的应用程序。

就像盖房子，老家盖楼自己一旦一点的打地基慢慢盖。

现在大楼大厦这种，都是搭建架子，然后再填水泥…

小程序用Servlet+JSP

大程序用框架SSM

2、常用的框架

早期框架 SSH

Struts2 🡪 Servlet (不安全)

Spring 🡪javabean

Hibernate 🡪jdbc（设计理念好，性能低，复杂/难）

现在框架 SSM

Spring 🡪 javabean

SpringMVC 🡪Servlet

Mybatis 🡪jdbc

3、Spring框架的概述

Spring是一个开源框架，可以整合许多其他的框架进行工作。

Spring的主要技术是**IOC(DI)**和**AOP**

IOC(DI) -- 控制反转（依赖注入）

简单说让IOC完成对对象的创建和销毁

以前Person p=new Person();

IOC Person p=null; 告诉IOC我要用p即可，IOC负责创

建具体的对象

以前的做法不能解耦合，现在可以使用IOC管理，解决耦合问题

AOP – 面向切面编程

也是为了解耦合，层与层之间加入切面代码

4、IOC（DI）的概念

所谓的IOC称之为控制反转，简单来说就是将对象的创建的权力及对象的生命周期的管理过程交给由Spring框架来处理，从此再开发过程中不再需要关注对象的创建和生命周期的管理，而是再需要时由Spring框架提供，这个由Spring框架管理对象和生命周期的机制称之为控制反转。

而在创建对象的过程中Spring可以依据配置对对象的属性进行设置，这个过程称之为依赖注入，也成为DI。

Book b=new Book(); --- IOC

b.setId();

b.setProductName(); --DI

5、IOC的第一个案例—环境的搭建

IOC控制反转，管理javabean

(1)下载Spring开发包 <https://spring.io>

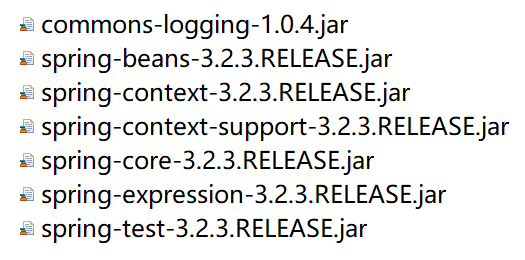
docs文档 libs 开发用的jar包 schema 约束

（2）创建一个普通的Java项目就可以（Spring因为是管理模型层的javabean，

没必要非得创建web项目）

（3）导入spring开发需要用的jar包（开发IOC的例子，使用如下7个jar

包）



（4）spring开发需要创建一个专门的配置文件，文件名和位置任意，一般定

义再在src目录下，文件名一般定义为applicationContext.xml

（5）spring的spring-beans-3.2.xsd约束引入进来

Step1: step2 step3……

6、IOC的第一个案例2

类Person --> IOC创建、管理Person

7、IOC的实现原理

在初始化一个Spring容器时，Spring会解析指定的xml文件，当解析到其中的<bean>标签时，会根据该标签的class属性中指定的类的全路径名，通过反射该类的对象，将该对象存入内置的Map中管理。

Class.forName(“com.lddx.bean.Person”) --->Person类

（通过反射获得的类对象等价于new Person获得的对象）

Map.put(“person”,Person);

以上都是Spring容器底层完成的

当通过context.getBean(“person”)方法从容器中获取对象时，相当于底层

Map.get(“person”) --->Person

由此可以得出几个推论：

（1）默认情况下，多次获取同一个id的bean，得到的将是同一个对象

（2）即使是同一个类，如果配置多个<bean>标签具有不同的id，每个id都会

在内置的Map中由一对键值对，其中的值就是这个类创建的不同对象

（3）同一个<beans>标签下，不允许配置多个相同id的<bean>标签，如果相

同会报错。

8、IOC获取对象的方式 -- SpringN01\_03\_IOCDemo02

**（1）通过bean标签的id属性来获取对象**

（2）通过bean标签的class属性来获取对象

（有潜在风险，在配置中的<bean>标签可能会出现多个一样的class属性）

（3）通过别名来获取对象（前提需要使用alias标签定义别名）

9、IOC创建对象的方式 SpringN01\_04\_IOCCreate

**（1）通过类的无参构造方式创建对象**

（2）通过静态工厂的方式创建对象

很多时候，我们面对的类是无法通过无参构造去创建的，例如该类没有

无参构造，是抽象类等情况，此时无法要求Spring的IOC通过无参构造来创建对象了，可以使用静态工厂的方式来创建。

（简单来说就是把创建对象的过程封装到了静态工厂中）

（3）通过实例工厂的方式创建对象

实例工厂也可以理解为类是无法通过无参构造来创建的问题，解决的思

路和静态工厂的方式类似，只不过实例工厂方式提供的方法是非静态方

法。

（4）通过Spring工厂的方式创建对象

Spring内置了一个工厂接口，是FactoryBean。

可以通过实现该接口来开发Spring工厂，来完成对对象的创建

10、单例和多例

Spring容器管理的对象（javabean对象）在默认情况下都是单例模式，也就是说一个javabean对象只会创建一次，存在内置的map中，之后无论获取多少次该对象，都返回的是同一个对象。

<bean id=”person” class=”com.lddx.bean.Person”></bean>

底层：**Person p=new Person() 单例模式会只创建一次**

Map.put(“person”,p);

Context.getBean(“person”) -->Map.get(“person”);

Spring默认采用单例方式，减少了对象的创建，从而减少了内存的消耗。

但是在实际开发中是存在很多多例模式的需求的，Spring框架也提供了可以设置多例模式的方式。（当当网中设计的购物车---多例）

多例模式是对象每次使用都会创建一个新的对象，不会像单例只会创建一次。

使用scope=”singleton/prototype”来设置是单例模式还是多例模式

总结：

（1）单例模式

Bean在单例模式下，spring容器启动时解析xml，发现bean标签后，直接创建bean的对象，并存入Map中保存。此后无论调用多少次getBean()获取对象，都是从Map中获取的同一个对象。

（2）多例模式

Bean在多例模式下，spring容器启动时解析xml，发现bean标签后，只是将bean进行管理，并不会创建该对象。此后每次使用getBean()获取对象的时候，Spring的IOC才会创建该对象并保存到Map中，所以getBean()获取的对象每一次都是一个新对象。

11、懒加载

Spring默认会在容器初始化的过程中，解析xml文件，并将单例模式的bean创建并保存到Map中，这种机制在bean比较少的情况下问题不大，一旦bean非常多的时候，spring的初始化过程就需要花费大量的时间来创建bean，花费大量的空间存储bean，但是这些bean可能很久都用不上，这种在启动时时间和空间上的浪费显得非常不值得。

Spring提供了懒加载机制。所谓的懒加载机制就是规定bean不在xml解析时立即创建，而是在后续第一次用到是才创建，而从减轻在Spring初始化过程中对时间和空间的消耗浪费。因为是单例模式，还是会创建，只不过懒加载机制把创建的时间往后推迟了而已，推迟第一次用的时候才创建。

（强调：懒加载机制只对单例模式起作用，对于多例模式设置懒加载没有意义）

lazy-init=””属性设置是否懒加载，true表示懒加载，false不懒加载（默认）

在<beans>标签中设置default-lazy-init=””是否懒加载

总结：1如果lazy-init和default-lazy-init同时对bean标签设置，lazy-init的优

先级更高；

2、如果lazy-init=””设置的值为default，相当于是引用default-lazy-init设

置的值。

12、Spring容器的初始化和销毁

Spring容器初始化的时候做一些创建、连接等操作；

Spring容器销毁时做一些关闭、释放资源等操作。

init-method=”” Spring容器初始化的时候执行设置的方法

destory-method=”” Spring销毁的时候执行设置的方法

13、依赖注入—DI

IOC（DI）控制反转（依赖注入）

在IOC创建对象的过程中，Spring可以依据相关配置对对象的属性赋值，对对象的属性赋值的过程称之为依赖注入，也叫DI。

以前的做法—不使用Spring框架

Person p=new Person();

p.setName(“zs”);

p.setAge(20);

p.setAddress(“山东…”);

或者

Person p1=new Person(“zs”,20,”山东…”);

现在Spring框架帮我们把Person创建，也会把Person中的属性进行赋值，这就是IOC和DI。

**DI依赖注入的两种方式：**

**（1）set方法注入**

**（2）构造方法注入**

14、DI-set方法注入 SpringN01\_06\_DI\_set

15、自动装配 -- autowire SpringN01\_07\_DI\_set\_autowire

使用set方式依赖注入的时候，如果一个<bean>中需要引入另外一个<bean>,配置的代码<property name="dog" ref="d"></property>，如果需要引入的<bean>特别多，需要写<property name="dog" ref="d"></property>的配置就会很多，可以使用自动装配，使用自动装配，那么<property name="dog" ref="d"></property>配置就可以省略不写了。

autowire=”byName” 保证javabean类中私有属性的名字和<bean>标签中的id的名字一致，否则自动装配不成功。一般使用byName进行自动装配。

autowire=”byType” 通过javabean类的类型来进行自动装配，

保证javabean类中的私有属性的类型要和<bean>标签中的class的内容一致。

此方法**有风险，**不可以出现多个<bean>标签中的class同名的情况，如果有同名的情况，通过byTpye的方式自动装配就会出错。

16、构造方法的依赖注入

对象中的私有属性的另外一种方式是在对象创建的过程中，通过构造方法传入并设置对象的属性。Spring也可以通过这样的构造方法的方式实现属性的注入。

**二、注解、IOC(注解方式)、DI(注解的方式)**

1、注解的概念

注释：给人看的提示信息，程序不看也不执行。

注解：主要给程序看的信息，程序会执行。

通常用来在某种程度上，尤其是框架上，注解用来代替配置文件，实

现轻量级配置。

注解是从JDK5.0开始提供的特性。

2、几个常见的注解

（1）@Override 子类继承父类，重写父类方法的注解

（2）@Deprecated 过时注解

（3）@SuppressWarnings 消除黄色警告

3、自定义注解

自定义一个注解的过程类似于定义一个接口

可以通过@interface来定义一个注解

该注解可以在一个类中的任何地方使用

元注解：可以通过元注解来修饰注解，控制自定义注解的特征

（元注解都是sun公司提供好的）

（1）@Target -- 声明当前定义的注解可以用在什么位置

ElementType.CONSTRUCTOR 构造方法中

ElementType.FIELD 成员变量

ElementType.LOCAL\_VARIABLE 局部变量上

ElementType.METHOD 方法上

ElementType.PACKAGE 包

ElementType.PARAMETER 方法参数

ElementType.TYPE 类

（2）@Retention 了解即可，偏底层设计

声明当前定义的注解被保留到什么阶段

A.java源文件 ---> A.class字节码文件 --->JVM

编译 运行

RetentionPolicy.SOURCE 保留到源码阶段，在编译过程中丢失，主要是

给编译器在编译代码的时候看到。

RetentionPolicy.CLASS 保留到.java和..class阶段，加载到jvm虚拟机

运行时就丢失了。

RetentionPolicy.RUNTIME 全阶段

（3）@Documented 文档注解

指定当前注解是否会被文档提取工具提取到自动生成的文档中

（4）注解中声明属性

1）在注解中声明属性的过程类似于接口中定义方法；

2）在注解声明中定义的属性需要在使用注解时为属性赋值（相当于接口中

的方法必须重写，注解中的属性必须赋值）；

3）在注解中声明的属性必须是public，public可以省略；

4）在注解中声明的属性的类型必须是八大基本数据类型，String类型、Class类型，枚举类型(enum)，其它注解类型，或者以上类型的一维数组。

5）在给注解中的属性赋值时，可以在使用注解的后面使用小括号，小括号里采用**属性名=赋值内容**的方式赋值，多个值之间用**逗号分隔开**。

6）可以在注解中声明属性时，使用default给属性赋一个默认值

7）如果注解中只有一个属性需要被赋值，且该属性的名字叫value，则在赋值的时候value=是可以省略的

8）如果注解中某个属性是一个一维数组类型，且给该属性赋值时，赋值的内容只有一个值，则赋值时大括号是可以省略的。

如果赋值的内容是多个值，则赋值的内容必须放在大括号里

4、注解的反射：基于注解实现控制程序执行

RetentionPolicy.RUNTIME级别的注解会保留到程序运行时，可以通过反射技术获取，从而根据是否有注解或注解属性值的不同控制程序按照不同方式进行。

boolean isAnnotationPresent() 判断是否有注解

getAnnotation() 获得注解

5、Spring注解方式实现IOC

Spring除了默认的使用xml配置文件的方式实现之外，也支持使用注解的方

式来实现配置，注解的方式效率更高，配置信息更清晰，修改更方便，推荐

使用。

前期准备工作：

**（1）需要导入一个新的jar包 spring-aop...jar包**

**（2）引入context约束文件**

案例：使用配置文件的方式和注解的方式分别配置javabean类

配置文件的方式配置javabean类的步骤

（1）在applicationContext.xml文件中配置

<bean id=”” class=””></bean>

**注解的方式配置javabean类步骤：**

**（1）在applicationContext.xml文件中开启全包扫描**

**<context:compoment-scan base-package=”” >**

**（2）在javabean类上添加注解@Component**

**在使用@Component注解时，id的推断规则：**

（1）如果明确指定了id的值，则使用指定的id值

（2）如果没有明确指定id的值，则自动推断id的值：

自动推断id时，看类名的第二个字母

如果类名的第二个字母是小写，则首字母转小写组成id的值；

如果类名的第二个字母是大写，则首字母保持不变组成id的值。

6、Spring注解方式实现工厂模式（实例工厂）

（1）对实例工厂加@Component注解

（2）对实例工厂中的方法添加@Bean注解

7、Spring注解方式实现静态工厂模式

（1）对静态工厂加注解@Component注解

（2）对静态方法添加@Bean注解

Spring框架 IOC（**配置文件的方式**、**注解的方式**）

DI（**配置文件的方式**、**注解的方式**）

8、Spring注解方式实现DI（依赖注入）

（1）注解实现DI—常规属性 String int double… @Value

Hero.java文件

（2）注解实现DI--@Value注解读取properties文件中的内容

@Value(${xxx})

MySQLConnection.java文件

（3）注解实现DI—集合和Properties对象

数据写在applicationContext.xml文件中，用util约束

使用@Value(#{@id值})获取数据给属性赋值

Person.java文件

（4）一个javabean类包含另外一个javabean类的情况

使用注解@Autowired来完成自动装配

**自动装配的实现原理：**

会先按照类型进行匹配，如果找到唯一的类型就进行依赖注入；

如果找不到类型会报错误；

如果找到的为多个类型，则再开始按照id进行查找；如果找到唯一的id就实现依赖注入，如果找不到报错误。

**正常情况：**

**代码运行没有任何异常错误**

**@Autowired**

**private Dog dog;**

**先按照类型（Dog类型）找到了3个，Dog类，HSQDog和JMDog，按照类型找是不唯一的；**

**再按照id查找匹配，根据属性dog匹配id的值，能找到唯一的Dog类**

**测试1：去掉Dog类上的@Component，运行程序，报错误**

**先按照Dog类型查找，找到2个，分别是HSQDog和JMDog，按照类型找是不唯一的**

**再按照id查找，根据属性dog匹配id的值，因为HSQDog和JMDog的默认id都不是dog，所以id没有匹配到**

**所以程序运行报错误了**

**测试2：去掉Dog类上的@Component，再去掉HSQDog类上的@Component，运行正常。**

**先按照Dog类型查找，只查找到了一个，JMDog,按照类型能够找到唯一的。**

**测试3：HSQDog，JMDog这两个类加@Component，Dog类还是去掉@Component，同时给HSQDog手动**

**的添加id的值为dog。**

**先按照Dog类型查找，找到了2个，HSDQdog和JMDog，按照类型找到的是不唯一的；**

**然后再按照id查找，因为HSQDog的id设置的就是dog，根据属性dog就能找到id的dog。**

**运行正常**

使用@Qualifier注解配置@Autowired注解一起使用

自动装配时不再按照类型匹配查找了，直接按照@Qualifier定义的内容直接

进行id匹配查找

9、其它注解（单例，多例，懒加载，spring容器的初始化和销毁）

@Scope 设置单例、多例的注解

@Scope(value=”prototype”) 多例

@Scope(“singleton”) 单例

@Lazy 懒加载注解

@PostConstruct Spring容器初始化的时候执行注解的方法

@PreDestory Spring容器销毁的时候执行注解的方法

10、案例—软件开发分层、彻底解耦合

@Controller @Service @Repository @Component

这四个注解的功能是完全相同的，都是用来修饰类的，会将类修饰为Spring

容器管理的javabean对象。

其中@Component一般认为是通用注解

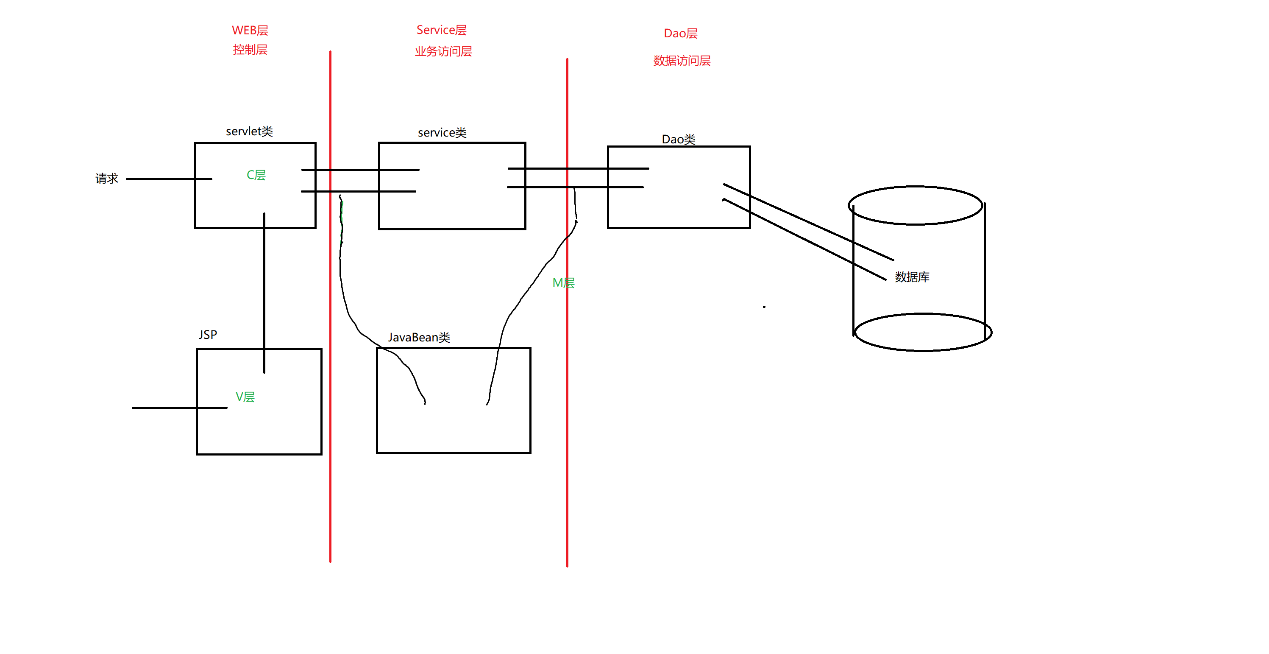
@Controller一般用在软件分层中的控制层，一般用在web层

@Service一般用在软件分层中的业务访问层，一般用在service层

@Repository一般用在软件分层中的数据访问层，一般用在dao层

Model模型层 ~~View视图层~~ Controller控制层

JaveEE的经典三层架构



**在JavaEE的经典三层架构中，将Model模型层的功能进一步细分，JavaBean只负责数据的封装，Service负责业务逻辑处理，Dao负责数据库的访问。**

**三、AOP面向切面编程**

1、AOP概述

AOP面向切面编程，就是在层与层之间加入代码，成为切面代码，主要目的

是在某些情况下，让层与层之间的耦合度进一步降低。

使用接口+IOC+DI已经能够很好的进行解耦合了。

AOP比较难理解的，需要准备一些场景代码讲解。

2、修改Service层的代码，增加功能性代码

**public** **void** addUser() {

System.*out*.println("#记录日志");

System.*out*.println("#权限控制");

System.*out*.println("#开启事务");

System.*out*.println("调用Dao层的方法，完成新增用户...");

System.*out*.println("#提交事务");

}

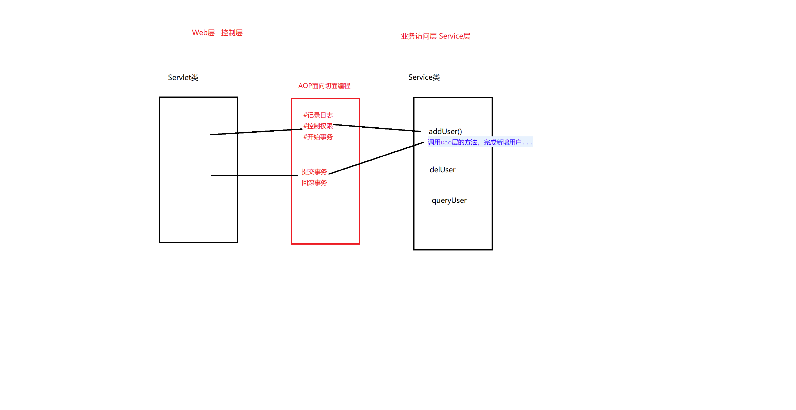
业务代码和功能代码混淆在一块了，比较乱，混杂。相当于把日记的内容和上课记笔记的内容写一个一起。

企业开发要保证六字真言：**高内聚，低耦合**。

高内聚：每一个组件，每一个层只干自己的事情，绝不越界。 AOP

低耦合：每个组件，每一个层之间耦合度降到最低。接口+Spring IOC+DI

3、AOP的概念，解决功能代码和业务代码的分离



4、代理模式 --AOP底层的核心实现技术思想

代理模式的核心思想：在不改变一个类的情况下增强该类中的方法

（1）静态代理

可以解决代码混杂的问题，无法解决代码重复的问题

（2）动态代理 --java提供Proxy类

既可以解决代码混杂的问题，也可以解决代码重复的问题

缺陷：基于接口的方式实现的，不在接口中存在的方法是访问不了的

（3）第三方动态代理 -- cglib代理

既可以解决代码混杂的问题，也可以解决代码重复的问题

cglib代码是基于继承实现的，对于非接口的方法依然是可以访问的

5、使用代理模式对项目进行重构

静态代理 -- SpringN03\_03\_project\_staticProxy

Java动态代理 --- SpringN03\_04\_project\_javaProxy

Cglib动态代理 -- SpringN03\_05\_project\_cglibProxy

6、SpringAOP基本概念

**专属名词：连接点、切入点、切面/切面类、通知方法、目标对象、目标方法**

连接点：层与层之间方法的调用过程

切入点：根据切入点决定对哪个连接点进行功能的增强

切入点=连接点+切入点表达式

切面：也叫切面类，用来写增强代码的功能

通知方法：简称通知，用来完成增强代码功能的方法

目标对象：增强代码执行完后回到的目标对象

目标方法：增强代码执行完后回到的目标方法

**专属动词：织入**

将切面类中的通知方法的结果返回到目标对象和目标方法的过程称之为织入

7、Spring的AOP入门案例

Step1：先准备开发jar包，一个12个jar包

Step2：在applicationContext文件中，导入aop约束

Step3：创建切面类

Step4：在切面类中定义通知方法

Step5：在applicationContext文件中配置切面类，通知方法以及切入点规则

Step6：执行

8、切入点表达式

（1）within()表达式

通过类名进行匹配目标对象，粗粒度的切入点表达式，只能控制到类这个级别。

A within()表达式可以使用\*号匹配符，匹配指定包下所有的类。

注意：只匹配当前包，不包括当前包的子孙包

B 在within()表达式中也可以使用\*号匹配包

注意：多个星号，最后一个星表示所有类，前面的星表示任意包

C 在within()表达式中使用..\*，匹配指定包下及其子孙包下所有的类

（2）execution()表达式

细粒度的切入点表达式，可以以方法为单位定义切入点规则

语法：execution(返回值类型 包名.类名.方法名(参数类型))

9、五大通知

（1）前置通知

1）基本使用：在目标方法之前执行的通知

2）配置方法：<aop:before />

3）前置通知执行的方法中可以定义一个参数JoinPoint参数

该参数代表当前连接点，通过该参数可以获取目标对象、目标方法的相关信息；

（2）环绕通知

1）基本使用：在目标方法执行之前和之后都可以执行的通知

2）配置方法：<aop:around />

3）环绕通知必须写参数，必须写返回值

（3）后置通知

1）基本使用：在目标方法成功执行之后执行的通知

2）配置方法：<aop:after-returning />

（4）异常通知

1）基本使用：在目标方法抛出异常时执行的通知

2）配置方法：<aop:after-throwing />

（5）最终通知

1）基本使用：是在目标方法执行之后执行的通知

2）配置方法：<aop:after />

3）最终通知和后置通知的区别：

后置通知是在目标方法正常执行后执行的通知，如果目标方法有异常，后置通知是不会执行的；

最终通知无论如何都会在目标方法执行完执行，即使目标方法有异常也会执行；

后置通知可以配置得到目标方法的返回结果，而最终通知无法获取。

注意：

JoinPoint可以在以上几个通知中都使用，如果有多个参数，要保证JoinPoint

参数在第一个。

10、五大通知的顺序问题

当一个切面类中同时配置了多个通知时，执行顺序如下：

前置、环绕前 一定在目标方法执行之前；

后置、环绕后、最终 一定在目标方法执行之后执行。

在符合以上的前提下，而他们之间的执行顺序取决于配置的顺序，配置越往前，执行越优先。

11、多个切面的执行顺序问题

12、注解方式AOP的开发

（1）浅谈SpringAOP的原理

Spring在创建bean时，除了创建目标对象bean之外，会根据aop的配置，生成目标对象的代理对象（助理）。

SpringAOP使用的代理是JavaProxy的动态代理和cglib的动态代理。默认情况下，如果目标对象实现过接口，则采用的JavaProxy代理机制。

如果目标对象没有实现过接口，则采用的是cglib代理机制。

proxy-target-class=”” 设置动态代理的机制，true是cglib代理

Spring框架 IOC DI AOP

配置文件方式、注解方式

（2）AOP的注解方式的实现

Spring也支持注解方式实现AOP，相对于配置文件方式，注解配置更加

轻量级，配置、修改更加方便，是目前最流行的方式。

AOP的注解方式的实现步骤

①在applicationContext.xml文件中开启注解方式AOP

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

②配置切面/切面类 @Aspect

③配置通知/通知方法 @Before @Around… 五大通知

@Pointcut 配置切入点表达式的注解

@Aspect 配置切面的注解

@Before 前置通知的注解

@Around 环绕通知的注解

@AfterReturning 后置通知的注解

@AfterThrowing 异常通知的注解

@After 最终通知的注解

@Order 配置切面的执行顺序

SSM

Spring框架 ok

SpringMVC框架 servlet

MyBatis框架 jdbc