## Spring 核心功能演示 + 面试题

Spring Framework 简称 Spring，是 Java 开发中最常用的框架，地位仅次于 Java API，就连近几年比较流行的微服务框架 SpringBoot，也是基于 Spring 实现的，SpringBoot 的诞生是为了让开发者更方便地使用 Spring，因此 Spring 在 Java 体系中的地位可谓首屈一指。  
当然，如果想要把 Spring 所有功能都讲的一清二楚，远远不是一两篇文章能够做到的，但幸运的是，Spring 的基础资料可以很轻易的搜索到，那么我们本讲主要的目的就是把 Spring 中的核心知识点和常见面试题分享给大家，希望对大家能有所帮助。

### Spring 介绍

Spring 是一个开源框架，为了解决企业应用程序开发复杂性而创建的，Spring 的概念诞生于 2002 年，于 2003 年正式发布第一个版本 Spring Framework 0.9。下面一起来看 Spring 各个版本的更新特性和它的发展变化吧。

#### Spring 1.x

此版本主要是为了解决企业应用程序开发复杂性而创建的，当时 J2EE 应用的经典架构是分层架构：表现层、业务层、持久层，最流行的组合就是 SSH（Struts、Spring、Hibernate）。  
Spring 1.x 仅支持基于 XML 的配置，确保用户代码不依赖 Spring，它主要包含了以下功能模块：aop、beans、ejb、jdbc、jndi、orm、transation、validation、web 等。

#### Spring 2.x

Spring 2.x 的改动并不是很大，主要是在 Spring 1.x 的基础上增加了几个新模块，如 ehcache、jms、jmx、scripting、stereotype 等。

#### Spring 3.x

Spring 3.x 开始不止支持 XML 的配置，还扩展了基于 Java 类的配置，还增加了 Expression、Instructment、Tomcat、oxm 等组件，同时将原来的 Web 细分为：Portlet、Servlet。

#### Spring 4.x

Spring 4.x 扩充了 Groovy、Messaging、WebMvc、Tiles2、WebSocket 等功能组件，同时 Spring 还适配了 Java 版本，全面支持 Java 8.0、Lambda 表达式等。随着 RESTful 架构风格被越来越多的用户所采用，Spring 4.x 也提供了 RestController 等注解新特性。

#### Spring 5.x

Spring 5.x 紧跟 Java 相关技术的更新迭代，不断适配 Java 的新版本，同时不断重构优化自身核心框架代码，支持函数式、响应式编程模型等。

### Spring 核心

Spring 核心包括以下三个方面：

* 控制反转（Ioc）
* 依赖注入（DI）
* 面向切面编程（AOP）

下面分别来看它的这些特性。

#### 控制反转（IoC）

控制反转（Inversion of Control，IoC），顾名思义所谓的控制反转就是把创建对象的权利交给框架去控制，而不需要人为地去创建，这样就实现了可插拔式的接口编程，有效地降低代码的耦合度，降低了扩展和维护的成本。  
比如，你去某地旅游不再用自己亲自为订购 A 酒店还是 B 酒店而发愁了，只需要把住店的需求告诉给某个托管平台，这个托管平台就会帮你订购一个既便宜又舒适的酒店，而这个帮你订购酒店的行为就可以称之为控制反转。

#### 依赖注入（DI）

依赖注入（Dependency Injection，DI），是组件之间依赖关系由容器在运行期决定，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

##### IoC 和 DI 的关系

IoC 是 Spring 中一个极为重要的概念，而 DI 则是实现 IoC 的方法和手段。  
接下来，我们来看依赖注入的常见实现方式有哪些？  
依赖注入的常见实现方式如下：

1. setter 注入
2. 构造方法注入
3. 注解注入

##### 1）setter 注入

Java 代码：

public class UserController {  
 // 注入 UserService 对象  
 private UserService userService;  
 public void setUserService(UserService userService){  
 this.userService = userService;  
 }  
}

XML 配置：

<bean name="userController" class="com.learning.controller.UserController">   
 <!-- 依赖注入 -->   
 <property name="userService" ref="userService"></property>   
</bean>   
<bean name="userService" class="com.learning.dao.impl.UserServiceImpl"></bean>

**Bean 标签的常用属性说明：**

* id：为实例化对象起名称，根据 id 值可以得到我们配置的实例化对象，id 属性的名称原则上可以任意命名，但是能包含任何特殊符号；
* class：创建对象所在类的全路径；
* name：功能和 id 属性一样，但是现在一般不用；与 id 的区别在于：name 属性值里可以包含特殊符号，但是 id 不可以；
* scope：一般最常用的有两个值： Singleton：单例模式，整个应用程序，只创建 bean 的一个实例；Prototype：原型模式，每次注入都会创建一个新的 bean 实例，Spring 默认的是单例模式。

##### 2）构造方法注入

Java 代码：

public class UserController {  
 private UserService userService;  
 public UserController(UserService userService){  
 this.userService = userService;  
 }  
}

XML 配置：

<bean name="userController" class="com.learning.controller.UserController">   
 <!-- 依赖注入 -->   
 <constructor-arg ref="userService"></constructor-arg>   
</bean>   
<bean name="userService" class="com.learning.dao.impl.UserServiceImpl"></bean>

##### 3）注解注入

@Controller  
public class UserController {  
 // 使用注解自动注入  
 @Autowired()  
 private UserService userService;  
 // do something  
}  
// 创建依赖对象  
@Service  
public class UserService {  
 // do something   
}

创建依赖对象的常见注解：@Component、@Controller、@Service、@Repository。

**总结** ：可以看出注解的方式要比传统的 XML（setter 和构造器注入）实现注入更为方便，同时注解方式也是官方力推的依赖注入最佳使用方式。

#### 面向切面编程（AOP）

面向切面编程（Aspect Oriented Programming，AOP），它就好比将系统按照功能分类，每一个类别就是一个“切面”，我们再针对不同的切面制定相应的规则，类似开发模式被称为面向切面编程。

##### AOP 使用场景

* 日志系统
* 安全统一效验

##### AOP 优点

* 集中处理某一类问题，方便维护
* 逻辑更加清晰
* 降低模块间的耦合度

##### AOP 相关概念

* Join point：连接点，程序执行期间的某一个点，例如执行方法或处理异常时候的点，在 Spring AOP 中，连接点总是表示方法的执行。
* Advice：通知，通知分为方法执行前通知，方法执行后通知、环绕通知等。许多 AOP 框架（包括 Spring）都将通知建模为拦截器，在连接点周围维护一系列拦截器（形成拦截器链），对连接点的方法进行增强。
* Pointcut：切点，匹配连接点（Join point）的表达式，是 AOP 的核心，并且 Spring 默认使用 AspectJ 作为切入点表达式语言。
* Aspect：切面，是一个跨越多个类的模块化的关注点，它是通知（Advice）和切点（Pointcut）合起来的抽象，它定义了一个切点（Pointcut）用来匹配连接点（Join point），也就是需要对需要拦截的那些方法进行定义。
* Target object：目标对象，被一个或者多个切面（Aspect）通知的对象，也就是需要被 AOP 进行拦截对方法进行增强（使用通知）的对象，也称为被通知的对象。由于在 AOP 里面使用运行时代理，因而目标对象一直是被代理的对象。
* AOP proxy：AOP 代理，为了实现切面（Aspect）功能使用 AOP 框架创建一个对象，在 Spring 框架里面一个 AOP 代理指的是 JDK 自身的动态代理或 CGLIB 实现的动态代理。
* Weaving：把切面加入到对象，并创建出代理对象的过程。
* Advisor：一个 Advisor 相当于一个小型的切面，不同的是它只有一个通知（Advice），Advisor 在事务管理里面会经常遇到。

##### AOP 代码实现

AOP 的示例我们就以开车为例，开车的完成流程是这样的：巡视车体及周围情况 → 发动 → 开车 → 熄火 → 锁车。

当然我们的主要目的是“开车”，但在开车之前和开完车之后，我们要做一些其他的工作，这些“其他”的工作，可以理解为 AOP 编程。

##### 1）创建类和方法

package com.learning.aop;  
import org.springframework.stereotype.Component;  
  
@Component("person")  
public class Person {  
 public void drive() {  
 System.out.println("开车");  
 }  
}

##### 2）创建 AOP 拦截

package com.learning.aop;  
  
import org.aspectj.lang.annotation.After;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.springframework.stereotype.Component;  
  
@Component  
@Aspect  
public class CarAop {  
 @Before("execution(\* com.learning.aop.Person.drive())")  
 public void before() {  
 System.out.println("巡视车体及周围情况");  
 System.out.println("发动");  
 }  
 @After("execution(\* com.learning.aop.Person.drive())")  
 public void after() {  
 System.out.println("熄火");  
 System.out.println("锁车");  
 }  
}

##### 3）XML 配置注入扫描包路径

<?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">  
 <context:component-scan base-package="com.learning"/>  
 <aop:aspectj-autoproxy/>  
</beans>

##### 4）创建测试类

package com.learning.aop;  
  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class PersonTest {  
 public static void main(String[] args) {  
 ApplicationContext context =  
 new ClassPathXmlApplicationContext("applicationContext.xml");  
 Person landlord = context.getBean("person", Person.class);  
 landlord.drive();  
 }  
}

运行测试代码，执行结果如下：

巡视车体及周围情况

发动

开车

熄火

锁车

##### AspectJ 注解说明：

* @Before — 前置通知，在连接点方法前调用；
* @Around — 环绕通知，它将覆盖原有方法，但是允许你通过反射调用原有方法；
* @After — 后置通知，在连接点方法后调用；
* @AfterReturning — 返回通知，在连接点方法执行并正常返回后调用，要求连接点方法在执行过程中没有发生异常；
* @AfterThrowing — 异常通知，当连接点方法异常时调用。

### 相关面试题

#### 1.@Value 注解的作用是什么？

答：基于 @Value 的注解可以读取 properties 配置文件，使用如下:

@Value(“#{configProperties[‘jdbc.username’]}”)

private String userName;

以上为读取 configProperties 下的 jdbc.username 配置。

#### 2.Spring 通知类型有哪些？

答：Spring 通知类型总共有 5 种：前置通知、环绕通知、后置通知、异常通知、最终通知。

* 前置通知（Before advice）：在目标方法执行之前执行的通知。在某连接点（ join point ）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。
* 环绕通知（Around Advice）：在目标方法执行之前和之后都可以执行额外代码的通知，也可以选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。
* 后置通知（After (finally) advice）：目标方法执行之后（某连接点退出的时候）执行的通知（不论是正常返回还是异常退出）。
* 异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。
* 最终通知（After returning advice）：在某连接点（join point）正常完成后执行的通知，例如，一个方法没有抛出任何异常，正常返回。

#### 3.怎么理解 Spring 中的 IOC 容器？

答：Spring IOC 就是把创建对象的权利交给框架去控制，而不需要人为的去创建，这样就实现了可插拔式的接口编程，有效地降低代码的耦合度，降低了扩展和维护的成本。

比如，去某地旅游不再用自己亲自为订购 A 酒店还是 B 酒店而发愁了，只需要把住店的需求告诉给某个托管平台，这个托管平台就会帮你订购一个既便宜又舒适的酒店，而这个帮你订购酒店的行为就可以称之为控制反转。

#### 4.怎么理解 Spring 中的依赖注入？

答：依赖注入是指组件之间的依赖关系由容器在运行期决定，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

#### 5.IoC 和 DI 有什么关系？

答：IoC 是 Spring 中一个极为重要的概念，提供了对象管理的功能，从而省去了人为创建麻烦，而 DI 正是实现 IoC 的方法和手段。

#### 6.@Component 和 @Bean 有什么区别？

答：它们的作用对象不同：@Component 作用于类，而 @Bean 注解作用于方法。

@Component 通常是通过类路径扫描来自动侦测和装配对象到 Spring 容器中，比如 @ComponentScan 注解就是定义扫描路径中的类装配到 Spring 的 Bean 容器中；@Bean 注解是告诉 Spring 这是某个类的实例，当我需要用它的时把它给我，@Bean 注解比 @Component 注解自定义性更强，很多地方我们只能通过 @Bean 注解来注册 Bean，比如当我们引用第三方库中的类需要装配到 Spring容器时，则只能通过 @Bean 来实现，比如以下示例，只能通过 @Bean 注解来实现：

public class WireThirdLibClass {  
 @Bean  
 public ThirdLibClass getThirdLibClass() {  
 return new ThirdLibClass();  
 }  
}

#### 7.Spring 中 bean 的作用域有几种类型？

答：Spring 中 bean 的作用域有四种类型，如下列表：

* 单例（Singleton）：整个应用程序，只创建 bean 的一个实例；
* 原型（Prototype）：每次注入都会创建一个新的 bean 实例；
* 会话（Session）：每个会话创建一个 bean 实例，只在 Web 系统中有效；
* 请求（Request）：每个请求创建一个 bean 实例，只在 Web 系统中有效。

Spring 默认的是单例模式。

#### 8.什么是 Spring 的内部 bean？

答：当一个 bean 仅被用作另一个 bean 的属性时，它能被声明为一个内部 bean，为了定义 inner Bean，在 Spring 的基于 XML 的配置元数据中，可以在 <property/> 或 <constructor-arg/> 元素内使用 <bean/> 元素，内部 bean 通常是匿名的，它们的 Scope 一般是 prototype。

#### 9.Spring 注入方式有哪些？

答：Spring 的注入方式包含以下五种：

* setter 注入
* 构造方法注入
* 注解注入
* 静态工厂注入
* 实例工厂注入

其中最常用的是前三种，官方推荐使用的是注解注入，相对使用更简单，维护成本更低，更直观。

#### 10.在 Spring 中如何操作数据库？

答：在 Spring 中操作数据库，可以使用 Spring 提供的 JdbcTemplate 对象，JdbcTemplate 类提供了很多便利的方法，比如把数据库数据转变成基本数据类型或对象，执行自定义的 SQL 语句，提供了自定义的数据错误处理等，JdbcTemplate 使用示例如下：

@Autowired  
private JdbcTemplate jdbcTemplate;  
// 新增  
@GetMapping("save")  
public String save(){  
 String sql = "INSERT INTO USER (USER\_NAME,PASS\_WORD) VALUES ('laowang','123')";  
 int rows = jdbcTemplate.update(sql);  
 return "执行成功，影响" + rows + "行";  
}  
// 删除  
@GetMapping("del")  
public String del(int id){  
 int rows= jdbcTemplate.update("DELETE FROM USER WHERE ID = ?",id);  
 return "执行成功，影响" + rows + "行";  
}  
// 查询  
@GetMapping("getMapById")  
public Map getMapById(Integer id){  
 String sql = "SELECT \* FROM USER WHERE ID = ?";  
 Map map= jdbcTemplate.queryForMap(sql,id);  
 return map;  
}

#### 11.Spring 的 JdbcTemplate 对象和 JDBC 有什么区别？

答：Spring 的 JdbcTemplate 是对 JDBC API 的封装，提供更多的功能和更便利的操作，比如 JdbcTemplate 拥有：

* JdbcTemplate 是线程安全的；
* 实例化操作比较简单，仅需要传递 DataSource；
* 自动完成资源的创建和释放工作；
* 创建一次 JdbcTemplate，到处可用，避免重复开发。

#### 12.Spring 有几种实现事务的方式？

答：Spring 实现事务有两种方式：编程式事务和声明式事务。  
编程式事务，使用 TransactionTemplate 或 PlatformTransactionManager 实现，示例代码如下：

private final TransactionTemplate transactionTemplate;  
public void add(User user) throws Exception{  
 // Spring编码式事务，回调机制  
 transactionTemplate.execute(new TransactionCallback<Object>() {  
 @Override  
 public Object doInTransaction(TransactionStatus status) {  
 try {  
 userMapper.insertSelective(user);  
 } catch (Exception e) {  
 // 异常，设置为回滚  
 status.setRollbackOnly();  
 throw e;  
 }  
 return null;  
 }  
 });  
}

如果有异常，调用 status.setRollbackOnly() 回滚事务，否则正常执行 doInTransaction() 方法，正常提交事务。  
如果事务控制的方法不需要返回值，就可以使用 TransactionCallbackWithoutResult（TransactionCallback 接口的抽象实现类）示例代码如下：

public void add(User user) throws Exception {  
 // Spring编码式事务，回调机制  
 transactionTemplate.execute(new TransactionCallbackWithoutResult() {  
 @Override  
 protected void doInTransactionWithoutResult(TransactionStatus status) {  
 try {  
 userMapper.insertSelective(user);  
 } catch (Exception e) {  
 // 异常，设置为回滚  
 status.setRollbackOnly();  
 throw e;  
 }  
 }  
 });  
}

声明式事务，底层是建立在 Spring AOP 的基础上，在方式执行前后进行拦截，并在目标方法开始执行前创建新事务或加入一个已存在事务，最后在目标方法执行完后根据情况提交或者回滚事务。  
声明式事务的优点：不需要编程，减少了代码的耦合，在配置文件中配置并在目标方法上添加 @Transactional 注解来实现，示例代码如下：

@Transactional  
public void save() {  
 User user = new User("laowang");  
 userMapper.insertSelective(user);  
 if (true) {  
 throw new RuntimeException("异常");  
 }  
}

抛出异常，事务会自动回滚，如果方法正常执行，则会自动提交事务。

#### 13.Spring 事务隔离级别有哪些？

答：Spring 的注入方式包含以下五种：

* ISOLATION\_DEFAULT：用底层数据库的设置隔离级别，数据库设置的是什么我就用什么；
* ISOLATIONREADUNCOMMITTED：未提交读，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）；
* ISOLATIONREADCOMMITTED：提交读，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读），SQL server 的默认级别；
* ISOLATIONREPEATABLEREAD：可重复读，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读），MySQL 的默认级别；
* ISOLATION\_SERIALIZABLE：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。

默认值为 ISOLATION\_DEFAULT 遵循数据库的事务隔离级别设置。

#### 14.Spring 声明式事务无效可能的原因有哪些？

答：可能的原因如下：

* MySQL 使用的是 MyISAM 引擎，而 MyISAM 是不支持事务的；
* @Transactional 使用在非 public 方法上，@Transactional 注解只能支持 public 级别，其他类型声明的事务不会生效；
* @Transactional 在同一个类中无事务方法 A() 内部调用有事务方法 B()，那么此时 B() 事物不会生效。Spring 中的 AOP 的底层实现原理是什么？

答：Spring AOP 的底层实现原理就是动态代理。Spring AOP 的动态代理有两种实现方式，对于接口使用的是 JDK 自带的动态代理来实现的，而对比非接口使用的是 CGLib 来实现的，关于动态代理的详细内容，可参考前面【反射和动态代理】的那篇文章。

#### 15.Spring 中的 Bean 是线程安全的吗？

答：Spring 中的 Bean 默认是单例模式，Spring 框架并没有对单例 Bean 进行多线程的封装处理，因此默认的情况 Bean 并非是安全的，最简单保证 Bean 安全的举措就是设置 Bean 的作用域为 Prototype（原型）模式，这样每次请求都会新建一个 Bean。

#### 16.说一下 Spring 中 Bean 的生命周期？

答：Spring 中 Bean 的生命周期如下：

* ① 实例化 Bean：对于 BeanFactory 容器，当客户向容器请求一个尚未初始化的 Bean 时，或初始化 Bean 的时候需要注入另一个尚未初始化的依赖时，容器就会调用 createBean 进行实例化。对于 ApplicationContext 容器，当容器启动结束后，通过获取 BeanDefinition 对象中的信息，实例化所有的 Bean；
* ② 设置对象属性（依赖注入）：实例化后的对象被封装在 BeanWrapper 对象中，紧接着 Spring 根据 BeanDefinition 中的信息以及通过 BeanWrapper 提供的设置属性的接口完成依赖注入；
* ③ 处理 Aware 接口：Spring 会检测该对象是否实现了 xxxAware 接口，并将相关的 xxxAware 实例注入给 Bean：
* 如果这个 Bean 已经实现了 BeanNameAware 接口，会调用它实现的 setBeanName(String BeanId) 方法，此处传递的就是 Spring 配置文件中 Bean 的 id 值；
* 如果这个 Bean 已经实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory() 方法，传递的是 Spring 工厂自身；
* 如果这个 Bean 已经实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext) 方法，传入 Spring 上下文；
* ④ BeanPostProcessor：如果想对 Bean 进行一些自定义的处理，那么可以让 Bean 实现了 BeanPostProcessor 接口，那将会调用 postProcessBeforeInitialization(Object obj, String s) 方法；
* ⑤ InitializingBean 与 init-method：如果 Bean 在 Spring 配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法；
* ⑥ 如果这个 Bean 实现了 BeanPostProcessor 接口，将会调用 postProcessAfterInitialization(Object obj, String s) 方法；由于这个方法是在 Bean 初始化结束时调用的，因而可以被应用于内存或缓存技术；

以上几个步骤完成后，Bean 就已经被正确创建了，之后就可以使用这个 Bean 了。

* ⑦ DisposableBean：当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 这个接口，会调用其实现的 destroy() 方法；
* ⑧ destroy-method：最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的销毁方法。

#### 17.Spring 有哪些优点?

答：Spring 优点如下：

* 开源免费的热门框架，稳定性高、解决问题成本低；
* 方便集成各种优秀的框架；
* 降低了代码耦合性，通过 Spring 提供的 IoC 容器，我们可以将对象之间的依赖关系交由 Spring 进行控制，避免硬编码所造成的过度程序耦合；
* 方便程序测试，在 Spring 里，测试变得非常简单，例如：Spring 对 Junit 的支持，可以通过注解方便的测试 Spring 程序；
* 降低 Java EE API 的使用难度，Spring 对很多难用的 Java EE API（如 JDBC、JavaMail、远程调用等）提供了一层封装，通过 Spring 的简易封装，让这些 Java EE API 的使用难度大为降低。

#### 18.Spring 和 Struts 的区别？

答：Spring 和 Struts 区别如下：  
Spring 特性如下：

* 具备 IOC/DI、AOP 等通用能力，提高研发效率
* 除了支持 Web 层建设以外，还提供了 J2EE 整体服务
* 方便与其他不同技术结合使用，如 Hibernate、MyBatis 等
* Spring 拦截机制是方法级别

Struts 特性如下：

* 是一个基于 MVC 模式的一个 Web 层的处理
* Struts 拦截机制是类级别

#### 19.Spring、SpringBoot、SpringCloud 的区别是什么？

答：它们的区别如下：

* Spring Framework 简称 Spring，是整个 Spring 生态的基础。
* Spring Boot 是一个快速开发框架，让开发者可以迅速搭建一套基于 Spring 的应用程序，并且将常用的 Spring 模块以及第三方模块，如 MyBatis、Hibernate 等都做了很好的集成，只需要简单的配置即可使用，不需要任何的 XML 配置文件，真正做到了开箱即用，同时默认支持 JSON 格式的数据，使用 Spring Boot 进行前后端分离开发也非常便捷。
* Spring Cloud 是一套整合了分布式应用常用模块的框架，使得开发者可以快速实现微服务应用。作为目前非常热门的技术，有关微服务的话题总是在各种场景下被大家讨论，企业的招聘信息中也越来越多地出现对于微服务架构能力的要求。

#### 20.Spring 中都是用了哪些设计模式？

答：Spring 中使用的设计模式如下：

* 工厂模式：通过 BeanFactory、ApplicationContext 来创建 bean 都是属于工厂模式；
* 单例、原型模式：创建 bean 对象设置作用域时，就可以声明 Singleton（单例模式）、Prototype（原型模式）；
* 察者模式：Spring 可以定义一下监听，如 ApplicationListener 当某个动作触发时就会发出通知；
* 责任链模式：AOP 拦截器的执行；
* 策略模式：在创建代理类时，如果代理的是接口使用的是 JDK 自身的动态代理，如果不是接口使用的是 CGLIB 实现动态代理。

### 总结

通过本节内容我们充分的了解了 Spring 的核心：IoC、DI、AOP，也是用代码演示了 Spring 核心功能的示例，其中可以发现的是 Spring 正在从之前的 XML 配置编程变为 Java 注解编程，注解编程让 Spring 更加轻量化简单化了，这一点在我们后面介绍 SpringBoot 的时候，会让你更加感同身受。对于开发者来说，只有真正掌握了 Spring，才能称得上是一名合格的 Java 工程师。当然，学习的目的是为了更好的应用，因此现在就一起动手实践起来吧。

[点击此处下载本文源码](https://github.com/vipstone/java-%20interview/tree/master/springlearning)

更多java学习资料如下：

[Java高级架构师课程(总共122门课程，1460GB)](https://www.consultdog.com/index.php/front/44/)

[最强java面试视频课程(21门课程，126GB)](https://www.consultdog.com/index.php/front/78/)