spring相关面试题

Bean的作用域

1、singleton：单例，Spring中的bean默认都是单例的。

2、prototype：每次请求都会创建一个新的bean实例。

3、request：每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。

4、session：每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP session内有效。

5、global-session：全局session作用域。

Bean的生命周期

1.对Bean进行实例化

2.依赖注入

3.如果Bean实现了 BeanNameAware 接口，Spring将调用 setBeanName ()，设置 Bean 的 id（xml文件中bean标签的id）

4.如果Bean实现了 BeanFactoryAware 接口，Spring将调用 setBeanFactory()

5.如果Bean实现了 ApplicationContextAware 接口，Spring容器将调用 setApplicationContext()

6.如果存在 BeanPostProcessor ，Spring将调用它们的 postProcessBeforeInitialization （预初始化）方法，在Bean初始化前对其进行处理

7.如果Bean实现了 InitializingBean 接口，Spring将调用它的 afterPropertiesSet 方法，然后调用xml定义的 init-method 方法，两个方法作用类似，都是在初始化 bean 的时候执行

8.如果存在 BeanPostProcessor ，Spring将调用它们的 postProcessAfterInitialization （后初始化）方法，在Bean初始化后对其进行处理

9.Bean初始化完成，供应用使用，直到应用被销毁

10.如果Bean实现了 DisposableBean 接口，Spring将调用它的 destory 方法，然后调用在xml中定义的 destory-method 方法，这两个方法作用类似，都是在Bean实例销毁前执行。

```
public interface BeanPostProcessor {
	@Nullable
	default Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
		return bean;
	}

	@Nullable
	default Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
		return bean;
	}

}

public interface InitializingBean {
	void afterPropertiesSet() throws Exception;
}
```

IOC容器初始化过程？

ioc 容器初始化过程：BeanDefinition 的资源定位、解析和注册。

1. 从XML中读取配置文件。
2. 将bean标签解析成 BeanDefinition，如解析 property 元素， 并注入到 BeanDefinition 实例中。
3. 将 BeanDefinition 注册到容器 BeanDefinitionMap 中。
4. BeanFactory 根据 BeanDefinition 的定义信息创建实例化和初始化 bean。

单例bean的初始化以及依赖注入一般都在容器初始化阶段进行，只有懒加载（lazy-init为true）的单例bean是在应用第一次调用getBean()时进行初始化和依赖注入。

```
// AbstractApplicationContext
// Instantiate all remaining (non-lazy-init) singletons.
finishBeanFactoryInitialization(beanFactory);
```

多例bean 在容器启动时不实例化，即使设置 lazy-init 为 false 也没用，只有调用了getBean()才进行实例化。

loadBeanDefinitions 采用了模板模式，具体加载 BeanDefinition 的逻辑由各个子类完成。

JDK动态代理和CGLIB动态代理的区别？

Spring AOP中的动态代理主要有两种方式：JDK动态代理和CGLIB动态代理。

JDK动态代理

如果目标类实现了接口，Spring AOP会选择使用JDK动态代理目标类。代理类根据目标类实现的接口动态生成，不需要自己编写，生成的动态代理类和目标类都实现相同的接口。JDK动态代理的核心是 `InvocationHandler` 接口和 `Proxy` 类。

缺点：目标类必须有实现的接口。如果某个类没有实现接口，那么这个类就不能用JDK动态代理。

CGLIB来动态代理

通过继承实现。如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library）可以在运行时动态生成类的字节码，动态创建目标类的子类对象，在子类对象中增强目标类。

CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为 `final` ，那么它是无法使用CGLIB做动态代理的。

优点：目标类不需要实现特定的接口，更加灵活。

什么时候采用哪种动态代理？

1. 如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP
2. 如果目标对象实现了接口，可以强制使用CGLIB实现AOP
3. 如果目标对象没有实现了接口，必须采用CGLIB库

Spring 用到了哪些设计模式?

1、简单工厂模式： BeanFactory 就是简单工厂模式的体现，根据传入一个唯一标识来获得 Bean 对象。

```
@Override
public Object getBean(String name) throws BeansException {
    assertBeanFactoryActive();
    return getBeanFactory().getBean(name);
}
```

2、工厂方法模式： FactoryBean 就是典型的工厂方法模式。spring在使用 getBean() 调用获得该bean时，会自动调用该bean的 getObject() 方法。每个 Bean 都会对应一个 FactoryBean ，如 SqlSessionFactory 对应 SqlSessionFactoryBean 。

3、单例模式：一个类仅有一个实例，提供一个访问它的全局访问点。Spring 创建 Bean 实例默认是单例的。

4、适配器模式：SpringMVC中的适配器 HandlerAdatper 。由于应用会有多个Controller实现，如果需要直接调用Controller方法，那么需要先判断是由哪一个Controller处理请求，

然后调用相应的方法。当增加新的 Controller，需要修改原来的逻辑，违反了开闭原则（对修改关闭，对扩展开放）。

为此，Spring提供了一个适配器接口，每一种 Controller 对应一种 HandlerAdapter 实现类，当请求过来，SpringMVC会调用 getHandler() 获取相应的Controller，然后获取该Controller对应的 HandlerAdapter ，最后调用 HandlerAdapter 的 handle() 方法处理请求，实际上调用的是Controller的 handleRequest() 。每次添加新的 Controller 时，只需要增加一个适配器类就可以，无需修改原有的逻辑。

常用的处理器适配器： SimpleControllerHandlerAdapter ， HttpRequestHandlerAdapter ， AnnotationMethodHandlerAdapter 。

```
// Determine handler for the current request.
mappedHandler = getHandler(processedRequest);

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

// Actually invoke the handler.
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

public class HttpRequestHandlerAdapter implements HandlerAdapter {

   @Override
   public boolean supports(Object handler) {//handler是被适配的对象，这里使用的是对象的适配器模式
      return (handler instanceof HttpRequestHandler);
   }

   @Override
   @Nullable
   public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
      throws Exception {

      ((HttpRequestHandler) handler).handleRequest(request, response);
      return null;
   }
}
```

5、代理模式：spring 的 aop 使用了动态代理，有两种方式 JdkDynamicAopProxy 和 Cglib2AopProxy 。

6、观察者模式：spring 中 observer 模式常用的地方是 listener 的实现，如 ApplicationListener 。

7、模板模式： Spring 中 jdbcTemplate 、 hibernateTemplate 等，就使用到了模板模式。

有哪些事务传播行为?

在TransactionDefinition接口中定义了七个事务传播行为：

1. PROPAGATION_REQUIRED 如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。如果嵌套调用的两个方法都加了事务注解，并且运行在相同线程中，则这两个方法使用相同的事务中。如果运行在不同线程中，则会开启新的事务。
2. PROPAGATION_SUPPORTS 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。
3. PROPAGATION_MANDATORY 如果已经存在一个事务，支持当前事务。如果不存在事务，则抛出异常 IllegalTransactionStateException 。
4. PROPAGATION_REQUIRES_NEW 总是开启一个新的事务。需要使用JtaTransactionManager作为事务管理器。
5. PROPAGATION_NOT_SUPPORTED 总是非事务地执行，并挂起任何存在的事务。需要使用JtaTransactionManager作为事务管理器。
6. PROPAGATION_NEVER 总是非事务地执行，如果存在一个活动事务，则抛出异常。
7. PROPAGATION_NESTED 如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务, 则按PROPAGATION_REQUIRED 属性执行。

PROPAGATION_NESTED 与PROPAGATION_REQUIRES_NEW的区别:

使用 PROPAGATION_REQUIRES_NEW 时，内层事务与外层事务是两个独立的事务。一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。

使用 PROPAGATION_NESTED 时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。