

大家好，很高兴我们可以继续学习交流Java高频面试题。本小节主要交流学习Spring框架的相关知识点。Spring是目前流行的一站式框架，包括SpringMVC和SpringBoot都给我们搭建Web系统提供了便利。

做为一名Java开发工程师，在日常的工作中，我们必不可少的会使用到Spring框架。有的同学对Spring的使用比较熟练，但是对其技术原理却不甚了解，而这正是面试中所需要具备的能力。接下来，让我们一起来交流学习Spring相关的技术原理吧~

## (1) 说一下Spring中的控制反转（IOC）吧。

**答：**IOC也叫**控制反转**，将对象间的依赖关系交给Spring容器，使用配置文件来创建所依赖的对象，由**主动**创建对象改为了**被动**方式，实现解耦合。可以通过注解**@Autowired**和**@Resource**来注入对象，被注入的对象必须被下边的四个注解之一标注：

- **@Controller**
- **@Service**
- **@Repository**
- **@Component**

在Spring配置文件中配置 **<context:annotation-config/>**元素开启注解。还有一个概念是**DI（依赖注入）**，和控制反转是同一个概念的不同角度的描述，即应用程序在运行时依赖IOC容器来动态注入对象需要的外部资源（对象等）。

**解析：**

Spring是一个**轻量级的IOC和AOP容器**框架。是为Java应用程序提供基础服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。

Spring的**核心模块**如下所示：

- **Spring Core：**是核心类库，提供IOC服务；
- **Spring Context：**提供框架式的Bean访问方式，以及企业级功能（JNDI、定时任务等）；
- **Spring AOP：**提供AOP服务；
- **Spring DAO：**对JDBC进行了抽象，简化了数据访问异常等处理；
- **Spring ORM：**对现有的ORM持久层框架进行了支持；
- **Spring Web：**提供了基本的面向Web的综合特性；
- **Spring MVC：**提供面向Web应用的Model-View-Controller实现。

## Spring的优点有哪些呢？

- Spring的**依赖注入**将对象之间的依赖关系交给了框架来处理，减小了各个组件之间的耦合性；
- **AOP面向切面编程**，可以将通用的任务抽取出来，复用性更高；
- Spring对于其余**主流框架都提供了很好的支持**，代码的侵入性很低。

简单阐述了Spring框架，并且介绍了其关键技术IOC之后，我们一起来看看Spring的另一个关键技术AOP（面向切面编程）吧~

## (2) Spring中的AOP面向切面编程有了解吗?

答：AOP，面向切面编程是指当需要在某一个方法之前或者之后做一些额外的操作，比如说日志记录，权限判断，异常统计等，可以利用AOP将功能代码从业务逻辑代码中分离出来。

13

AOP中有如下的操作术语：

- **Joinpoint(连接点)**：类里面可以被增强的方法，这些方法称为连接点
- **Pointcut(切入点)**：所谓切入点是指我们要对哪些Joinpoint进行拦截的定义
- **Advice(通知/增强)**：所谓通知是指拦截到Joinpoint之后所要做的事情就是通知。
- **Aspect(切面)**：是切入点和通知（引介）的结合
- **Introduction(引介)**：引介是一种特殊的通知在不修改类代码的前提下，Introduction可以在运行期为类动态地添加一些方法或属性
- **Target(目标对象)**：代（dai）理的目标对象(要增强的类)
- **Weaving(织入)**：是把增强应用到目标的过程，把advice 应用到 target的过程
- **Proxy（代(dai)理）**：一个类被AOP织入增强后，就产生一个结果代（dai）理类

我们来简单理解下重要概念。**切入点就是在类里边可以有很多方法被增强**，比如实际操作中，只是增强了个别方法，则定义实际被增强的某个方法为切入点；**通知/增强 就是指增强的逻辑**，比如扩展日志功能，这个日志功能称为增强；**切面就是把增强应用到具体方法上面的过程称为切面**。

Spring中的AOP主要有两种实现方式：

- **使用JDK动态代（dai）理实现**，使用java.lang.reflection.Proxy类来处理
- **使用cglib来实现**

两种实现方式的不同之处：

**JDK动态代（dai）理**，只能对实现了接口的类生成代（dai）理，而不是针对类，该目标类型实现的接口都将被代（dai）理。原理是通过在运行期间创建一个接口的实现类来完成对目标对象的代（dai）理。**实现步骤大概如下：**

- 定义一个实现接口InvocationHandler的类
- 通过构造函数，注入被代（dai）理类
- 实现invoke（Object proxy, Method method, Object[] args）方法
- 在主函数中获得被代（dai）理类的类加载器
- 使用Proxy.newProxyInstance（）产生一个代（dai）理对象
- 通过代（dai）理对象调用各种方法

**cglib主要针对类实现代（dai）理，对是否实现接口无要求**。原理是对指定的类生成一个子类，覆盖其中的方法，因为是继承，所以被代（dai）理的类或方法不可以声明为final类型。**实现步骤大概如下：**

- 定义一个实现了MethodInterceptor接口的类
- 实现其 intercept（）方法，在其中调用proxy.invokeSuper（）

接下来，我们看分别给出JDK动态代（dai）理和cglib实现动态代（dai）理的Demo。

## JDK动态代（dai）理Demo:

```
1 package com.package1;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5 import java.lang.reflect.Proxy;
6
7 public class DynamicProxy {
8
9     public static void main(String[] args){
10         //定义一个people作为被代（dai）理的实例
11         IPeople ple=new People();
12         //定义一个handler
13         InvocationHandler handle=new MyHandle(ple);
14
15         //获得类加载器
16         ClassLoader cl=ple.getClass().getClassLoader();
17
18         //动态产生一个代（dai）理，下边两种方法均可
19         // IPeople p=(IPeople) Proxy.newProxyInstance(cl, new Class[]{IPeople.class}, handle)
20         IPeople p=(IPeople) Proxy.newProxyInstance(cl, ple.getClass().getInterfaces(), handl
21
22         //执行被代（dai）理者的方法。
23         p.func();
24     }
25
26
27
28 }
29
30 class MyHandle implements InvocationHandler{
31
32
33     //被代（dai）理的实例
34     Object obj=null;
35
36     //我要代（dai）理谁
37     public MyHandle(Object obj){
38         this.obj=obj;
39     }
40
41
42     @Override
43     public Object invoke(Object proxy, Method method, Object[] args)
44         throws Throwable {
45         Object result=method.invoke(this.obj, args);
46         return result;
47     }
48
49 }
50
51 interface IPeople{
52
53     public void fun();
54
55     public void func();
56 }
57
58 //实际被代（dai）理类
59 class People implements IPeople{
60
61     @Override
62     public void fun() {
63         System.out.println("这是fun方法");
64     }
65 }
```

13

```

66
67     @Override
68     public void func() {
69         System.out.println("这是func方法");
70     }
71
72
73 }

```

13

### cglib实现动态代理(dai) Demo:

```

1  package com.pak;
2
3  import net.sf.cglib.proxy.*;
4
5  import java.lang.reflect.Method;
6
7  public class TestCglib {
8      public static void main(String[] args) {
9
10         // 定义一个回调接口的数组
11         Callback[] callbacks = new Callback[] {
12             new MyApiInterceptor(), new MyApiInterceptorForPlay()
13         };
14
15         Enhancer enhancer = new Enhancer();
16         enhancer.setSuperclass(Person.class); // 设置要代理(dai)的父类
17         enhancer.setCallbacks(callbacks); // 设置回调的拦截器数组
18         enhancer.setCallbackFilter(new CallbackFilterImpl()); // 设置回调选择器
19
20         Person person = (Person) enhancer.create(); // 创建代理(dai)对象
21
22         person.eat();
23         System.out.println("-----");
24         person.play();
25     }
26 }
27
28 class MyApiInterceptor implements MethodInterceptor {
29
30     @Override
31     public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws
32         System.out.println("吃饭前我会先洗手"); // 此处可以做一些操作
33         Object result = proxy.invokeSuper(obj, args);
34         System.out.println("吃完饭我会先休息会儿"); // 方法调用之后也可以进行一些操作
35         return result;
36     }
37 }
38 class MyApiInterceptorForPlay implements MethodInterceptor {
39
40     @Override
41     public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws
42         System.out.println("出去玩我会先带好玩具"); // 此处可以做一些操作
43         Object result = proxy.invokeSuper(obj, args);
44         System.out.println("玩一个小时我就回家了"); // 方法调用之后也可以进行一些操作
45         return result;
46     }
47 }
48
49 class CallbackFilterImpl implements CallbackFilter {
50     @Override
51     public int accept(Method method) {
52         if (method.getName().equals("play"))
53             return 1;

```

```
54         else
55             return 0;
56     }
57 }
58
59
60 // 创建一个普通类做为代（dai）理类
61 class Person {
62     // 代（dai）理类中由普通方法
63     public void eat() {
64         System.out.println("我要开始吃饭咯...");
65     }
66
67     public void play() {
68         System.out.println("我要出去玩耍了,,,");
69     }
70 }
```

13

### Spring AOP对这两种代（dai）理方式的选择：

- 如果目标对象实现了接口，默认情况下会采用JDK的动态代（dai）理实现AOP，也可以强制使用cglib实现AOP；
- 如果目标对象没有实现接口，必须采用cglib库，Spring会自动在JDK动态代（dai）理和cglib之间转换。

#### 解析：

IOC和AOP都是Spring中的关键技术，在上边的介绍中，我们简单给出了JDK动态代（dai）理和cglib的实现步骤，并且给出了两种动态代（dai）理的实现Demo。

在日常开发中，IOC和AOP也是必不可少的关键技能，IOC技术常用来注入对象，而AOP则是用来在接口上设置拦截器进行一些权限判断等，希望大家可以切实理解IOC和AOP技术。

### (3) IOC容器的初始化过程：

答：IOC容器的初始化主要包括**Resource定位**，**载入和注册**三个步骤，接下来我们依次介绍。

- **Resource资源定位：**

Resource定位是指BeanDefinition的资源定位，也就是IOC容器找数据的过程。Spring中使用外部资源来描述一个Bean对象，IOC容器第一步就是需要定位Resource外部资源。由ResourceLoader通过统一的Resource接口来完成定位。

- **BeanDefinition的载入：**

载入过程就是把定义好的Bean表示成IOC容器内部的数据结构，即BeanDefinition。在配置文件中每一个Bean都对应着一个BeanDefinition对象。

通过BeanDefinitionReader读取，解析Resource定位的资源，将用户定义好的Bean表示成IOC容器的内部数据结构BeanDefinition。

在IOC容器内部维护着一个BeanDefinitionMap的数据结构，通过BeanDefinitionMap，IOC容器可以对Bean进行更好的管理。

- **BeanDefinition的注册:**

注册就是将前面的BeanDefition保存到Map中的过程，通过BeanDefinitionRegistry接口来实现注册。

13

**解析:**

**IOC容器的初始化过程就是对Bean定义资源的定位、载入和注册**，此时容器对Bean的依赖注入并没有发生。接下来，我们看下**依赖注入的发生时刻吧**。

ApplicationContext默认会在容器启动的时候创建我们配置好的各个Bean，我们的Bean配置如下：

```
1 | <bean id="oneBean" class="com.nowcoder.oneBean">
```

这里边的隐藏属性是lazy-init，即上边的配置和下边的是一样的：

```
1 | <bean id="oneBean" class="com.nowcoder.oneBean" lazy-init="false">
```

**lazy-init=false**表示不开启延迟加载，在容器启动的时候即创建该Bean。对应的，我们还可以配置lazy-init=true表示开启延迟加载，那么该Bean的创建发生在应用程序**第一次向容器索取Bean**时，通过getBean()方法的调用完成。

介绍了IOC容器的初始化以及Bean的注入发生时刻，我们再来看以下几个常见的Spring面试题吧~

## BeanFactory和FactoryBean的区别：

- **BeanFactory**：Bean工厂，是一个工厂(Factory)，是Spring IOC容器的最顶层接口，它的作用是**管理Bean，即实例化、定位、配置**应用程序中的对象及建立这些对象间的依赖。
- **FactoryBean**：工厂Bean，**是一个Bean，作用是产生其他Bean实例**，需要提供一个工厂方法，该方法用来返回其他Bean实例。

## BeanFactory和ApplicationContext有什么区别？

BeanFactory是Spring里面最顶层的接口，包含了各种Bean的定义，读取Bean配置文档，管理Bean的加载、实例化，控制Bean的生命周期，维护Bean之间的依赖关系。

**ApplicationContext**接口是BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

- 继承了MessageSource，支持国际化。
- 提供了统一的资源文件访问方式。
- 提供在Listener中注册Bean的事件。
- 提供同时加载多个配置文件的功能。
- 载入多个（有继承关系）上下文，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

## ApplicationContext 三种常见的实现方式:

- **FileSystemXmlApplicationContext**：此容器从一个XML文件中加载Bean的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。
- **ClassPathXmlApplicationContext**：此容器也从一个XML文件中加载Bean的定义，需要正确设置classpath因为这个容器将在classpath里找Bean配置。
- **WebXmlApplicationContext**：此容器加载一个XML文件，定义了一个WEB应用的所有Bean。

### 在创建Bean和内存占用方面的区别：

- **BeanFactory采用的是延迟加载形式来注入Bean的**，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，就不能发现一些存在于**Spring配置**中的问题。如果Bean的某一个属性没有注入，BeanFactory加载后，直至第一次使用调用getBean方法才会抛出异常。
- **ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。**这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例Bean，确保当需要的时候，可以直接获取。
- 相对于基本的BeanFactory，ApplicationContext不足之处是占用内存空间。当应用程序配置Bean较多时，程序启动较慢，因为其一次性创建了所有的Bean。

### BeanFactory和ApplicationContext的优缺点分析：

#### BeanFactory的优缺点：

**优点：**应用启动的时候占用资源很少，对资源要求较高的应用，比较有优势；

**缺点：**运行速度会相对来说慢一些。而且有可能出现空指针异常的错误，而且通过Bean工厂创建的Bean生命周期会简单一些。

#### ApplicationContext的优缺点：

**优点：**所有的Bean在启动的时候都进行了加载，系统运行的速度快；在系统启动的时候，可以发现系统中的配置问题。

**缺点：**把费时的操作放到系统启动中完成，所有的对象都可以预加载，缺点就是内存占用较大。

## (4) Spring中Bean的作用域有哪几种？

**答：**Spring框架支持以下五种Bean的作用域：

- **singleton** : Bean在每个Spring IOC 容器中只有一个实例，默认作用域。
- **prototype** : 一个Bean的定义可以有多个实例。
- **request** : 每次http请求都会创建一个Bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。
- **session** : 在一个HTTP Session中，一个Bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
- **global-session** : 在一个全局的HTTP Session中，一个Bean对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

#### 解析：

Spring中Bean的作用域常使用的是singleton，也就是单例作用域。**那么单例Bean是线程安全的吗？**

**答：**单例Bean和线程安全与否没有必然的关系。多个线程在多个工作内存和主内存交互的时候会出现不一致的地方，那么就不是线程安全的。大部分的Spring Bean并没有可变的状态(比如Service类和DAO类)，所以一定程度上可以说Spring的单例Bean是线程安全的。如果你的Bean有多种状态的话(比如 View Model 对象)，就需要自行保证线程安全。在一般情况下，只有无状态的Bean才可以在多线程环境下共享。

#### 循环依赖：



我们再来看一个关于**循环依赖**的问题吧，面试官会这么问，“如果A对象创建的过程需要使用到B对象，但是B对象创建的时候也需要A对象，也就是构成了循环依赖的现象，那么Spring会如何解决？”

答：这是一种**构造器循环依赖**，通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出 `BeanCurrentlyInCreationException` 异常表示循环依赖。

13

## (5) Spring的事务有了解吗？

答：Spring支持编程式事务管理和声明式事务管理两种方式：

- **编程式事务管理**：使用 `TransactionTemplate` 实现。
- **声明式事务管理**：建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

### 声明式事务的优点：

就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过 `@Transactional` 注解的方式，便可以将事务规则应用到业务逻辑中。

### 事务选择：

声明式事务管理要优于编程式事务管理，这正是Spring倡导的**非侵入式的开发方式**，使业务代码不受污染，只要**加上注解**就可以获得完全的事务支持。唯一不足之处是声明式事务的最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

当多个Spring事务存在的时候，Spring定义了下边的7个传播行为来处理这些事务行为：

- **PROPAGATION\_REQUIRED**：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，**该设置是最常用的设置**。
- **PROPAGATION\_SUPPORTS**：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。
- **PROPAGATION\_MANDATORY**：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。
- **PROPAGATION\_REQUIRES\_NEW**：创建新事务，无论当前存不存在事务，都创建新事务。
- **PROPAGATION\_NOT\_SUPPORTED**：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
- **PROPAGATION\_NEVER**：以非事务方式执行，如果当前存在事务，则抛出异常。
- **PROPAGATION\_NESTED**：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

解析：

Spring事务的考察，我们主要是掌握**声明式和编程式事务管理的区别与选择**。事务的传播行为也要有一个了解，另外，Spring事务的隔离级别和MySQL事务的隔离级别类似，依然存在**读未提交，读已提交，可重复读以及串行**四种隔离级别。我们在MySQL章节会进行详细介绍。

## (6) SpringMVC的消息处理流程有了解吗？

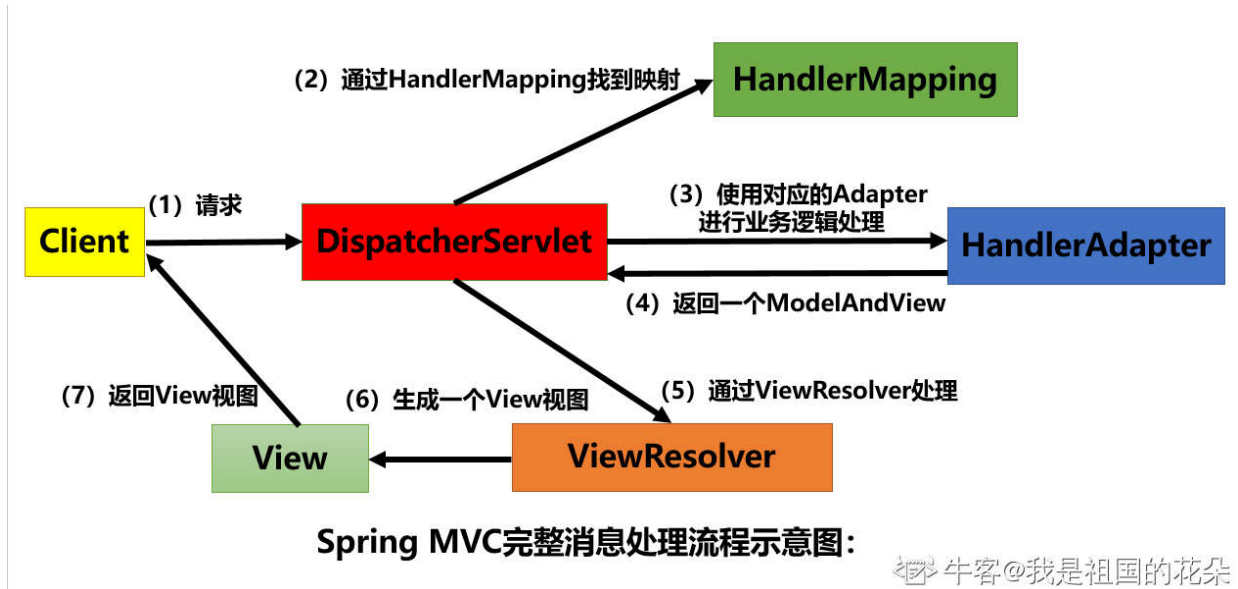


答：SpringMVC是一种轻量级的Web层框架，是一个基于请求驱动的Web框架，使用了“**前端控制器(DispatcherServlet)**”模型来进行设计，再根据请求映射规则分发给相应的页面控制器进行处理。消息处理依次经过的组件如下：

**DispatcherServlet、HandlerMapping、HandlerAdapter，返回一个ModelAndView逻辑视图名、ViewResolver、View**

13

**具体流程可以概括为：**通过前端控制器DispatcherServlet来接收并且分发请求，然后通过HandlerMapping和HandlerAdapter找到具体可以处理该请求的Handler，经过逻辑处理，返回一个ModelAndView，经过ViewResolver处理，最后生成了一个View视图返回给了客户端。可以参考如下的示意图：



牛客@我是祖国的花朵

## (7) 简单说下SpringBoot吧。

答：Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件**一站式解决方案**，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了**各种启动器starter**，开发者能快速上手。

SpringBoot的优点包括可以独立运行，简化了配置，可以实现**自动配置**，无代码生成以及XML配置，并且可以进行应用监控。

**注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心**，实现了SpringBoot项目的自动配置。

### SpringBoot的核心注解：

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

- **@SpringBootConfiguration**：组合了 @Configuration 注解，实现配置文件的功能。
- **@EnableAutoConfiguration**：打开自动配置的功能，也可以关闭某个自动配置的选项。如关闭数据源自动配置功能：@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。
- **@ComponentScan**：Spring组件扫描。

### SpringBoot项目启动分析：

Application类是通过**SpringApplication**类的静态run方法来启动应用的。打开这个静态方法，该静态方法真正执行的是两部分：**new SpringApplication()**并且**执行对象run()**方法。

解析:

SpringBoot框架在平时使用较多，面试中也会考察其自动配置的实现原理以及项目的启动流程等知识点。


总结:

该小节中，我们对Spring相关知识点进行了简单的交流和学习。Spring相关知识点，在面试中的考察可深可浅。对于校招来说，一般情况下需要掌握Spring的基本技术原理。对于社招来说，理应对Spring相关的源码进行分析，比如SpringBoot的相关启动源码等。限于文章篇幅，我们仅仅做了简单的介绍与交流，关于Spring框架相关的知识点，大家着重掌握与巩固IOC以及AOP相关的基础知识点，有能力和精力的同学可以进一步深入研究Spring相关技术原理。

限于作者水平，文章中难免会有不妥之处。大家在学习过程中遇到我没有表达清楚或者表述有误的地方，欢迎随时在文章下边指出，我会及时关注，随时改正。另外，大家有任何话题都可以在下边留言，我们一起交流探讨。

讨论

评论



我是祖国的花朵

N


作者

1#

大家好，\*\*\*两字触发了牛客的敏感字，所以打不出来，目前使用代（dai）理来显示，后续应该会修复该问题。

发表于 2019-12-15 16:21:00

赞(2) 回复(0)



还年轻多学习

N

2#

你好，关于循环依赖的问题，set注入的方式会出现吗？就你所说的场景中会有可能是set依赖吗？

发表于 2019-12-25 21:31:33

赞(0) 回复(1)

我是祖国的花朵

N

作者

： 你好，文中所述的是构造器循环依赖，解决方式就是抛异常；你说的属于set循环依赖，Spring中是有解决办法的。

2020-01-18 17:28:13

赞(0) 回复(0)

请输入你的观点

回复




刘畅201904211606816

3#

打卡

发表于 2020-01-04 21:56:27

赞(0) 回复(0)



zsy78006605

N

4#

你好，Bean的lazy-init那里是不是说反了？

发表于 2020-01-17 03:32:38

赞(1) 回复(1)

我是祖国的花朵

N

作者

： 已经修复完善了这块逻辑，感谢指出。加油~

2020-01-18 17:22:17

赞(0) 回复(0)

请输入你的观点

回复

13



柳杰201905011049420

5#

\*\*\*的两种方式看的不是很懂

发表于 2020-01-17 16:32:23

赞(0) 回复(0)



牧水s

6#

打卡

发表于 2020-02-14 21:02:42

赞(0) 回复(0)



ZQ.stu

7#

如果没有spring使用经验, 校招中需要对其原理了解到什么程度?

发表于 2020-02-15 20:22:59

赞(0) 回复(1)

我是祖国的花朵 : 重点掌握IOC和AOP的相关知识点, 不必到源码级别, 主要是其实现方式等概念性问题。

2020-02-16 11:45:57

赞(0) 回复(0)

请输入你的观点

回复



小源20190118151956

8#

打卡

发表于 2020-02-18 22:40:04

赞(0) 回复(0)



牛客ID: 9047559

9#

楼主您好, 有时间的话可以总结下springboot自动配置的实现原理以及项目的启动流程吗? 谢谢

发表于 2020-03-06 17:27:37

赞(2) 回复(0)



人见人爱的OFFER收割机

10#

楼主您好, 有时间的话可以总结下springboot自动配置的实现原理以及项目的启动流程吗? 谢谢

发表于 2020-03-30 10:30:22

赞(1) 回复(0)



小小一只

11#

Spring事务传播方式, 没用过, 根本记不住

发表于 2020-03-31 22:06:51

赞(0) 回复(1)

我是祖国的花朵 : 哈哈, 记住常用的前几个就可以啦

请输入你的观点

回复

13



赚多多

12#

你好，请问一下SpringBoot和Kafaka在校招中考察的深度怎么样

发表于 2020-05-13 09:37:03

赞(0) 回复(1)

啦啦啦啦201905171741501： 很好理解，就是假如两个方法都用上了事物，那么其中一个方法去调用另一个方法，这个时候你需要告诉系统应该去遵守那个方法的事物，spring给出了所有的可能，这些可能性就是事物的传播机制

2020-05-14 01:42:49

赞(0) 回复(0)

请输入你的观点

回复



啦啦啦啦201905171741501

13#

感谢，在网上东找西找，你这总结的非常好

发表于 2020-05-14 01:39:26

赞(1) 回复(1)

我是祖国的花朵 作者： 加油~

2020-05-15 09:28:43

赞(0) 回复(0)

请输入你的观点

回复