面试真题

【美团】SpringBoot框架有什么优势？核心注解有哪些？

**Spring Boot框架优势：**

1. **自动配置**：通过@EnableAutoConfiguration和@SpringBootApplication等注解，自动配置Spring组件，减少了手动配置的复杂度。
2. 内嵌Web服务器：如Tomcat、Jetty、Undertow，无需外部容器即可运行应用。
3. **微服务支持**：与Spring Cloud集成，适用于微服务架构，支持服务注册、发现、配置管理等功能。
4. **生产环境友好**：集成了Spring Boot Actuator，可轻松获取应用健康检查、监控和管理信息。

**核心注解：**

1. **@SpringBootApplication**：核心注解，标记为Spring Boot应用，包含了
   1. **@Configuration**：标记配置类，Spring会根据该类进行应用上下文的配置。
   2. **@EnableAutoConfiguration**：启用自动配置，Spring Boot根据项目的依赖自动配置适当的Spring组件。
   3. **@ComponentScan**：指定自动扫描的包，Spring会自动扫描并注册带有@Component、@Service、@Repository、@Controller等注解的类。

【美团】springboot的自动装配有了解吗？

Spring Boot的自动装配（Auto Configuration）是指根据应用的环境和配置自动配置所需的Spring Beans。

**自动装配的工作流程：**

1. **@EnableAutoConfiguration、@AutoConfigurationPackage**：通过该注解，Spring Boot启用自动装配功能、自动扫描并注册需要自动配置的类。该注解通常与@SpringBootApplication一起使用。
2. Spring Boot启动时，会加载spring.factories文件（包含自动装配相关的配置），读取里面的自动配置类。
3. **@Conditional**系列注解：自动配置类根据环境、类路径等条件，判断是否需要配置相关的Bean。
4. 如果条件满足，自动装配就会生效，相关的Bean会被注册到Spring上下文中。

**自动装配的常见场景：**

1. **数据库连接池**：Spring Boot自动根据配置文件中的数据库URL、用户名、密码等信息配置数据源。
2. **Web服务器**：如果没有显式指定Web服务器，Spring Boot会自动使用内嵌的Tomcat、Jetty或Undertow。
3. **缓存**：Spring Boot根据类路径中的依赖，自动配置缓存（如Redis、Ehcache等）。
4. **消息中间件**：自动配置Kafka、RabbitMQ等消息中间件。

【美团】@AutoWire和@Resource区别？

都是依赖注入、**自动装配Bean**的注解

* **@Autowired：**默认**按类型（byType）** 、Spring 特有、 支持@Autowired(required = false)依赖项可选，即没有匹配到时注入 null不抛异常。 支持结合 @Qualifier 注解指定注入的 Bean 名称。
* **@Resource：** 默认**按名称（byName）** 、JSR-250 标准（更适合标准化和跨框架）、 如果未指定名称且没有匹配的 Bean，会按类型注入。

【美团】springboot启动流程？

1. **创建SpringApplication实例**：
   * SpringApplication.run(Application.class, args)是启动Spring Boot应用的入口。SpringApplication类负责初始化应用的配置和上下文。
2. **设置环境（Environment）**：
   * Spring Boot会创建一个Environment对象，它包含了应用的配置信息（如application.properties或application.yml中的内容）。
   * Environment还会包括系统属性、命令行参数等。
3. **应用上下文（ApplicationContext）初始化**：
   * Spring Boot自动选择合适的ApplicationContext类型，通常为AnnotationConfigApplicationContext或GenericWebApplicationContext（如果是Web应用）。
   * Spring容器加载所有的@Configuration类及其Bean。
4. **自动装配（Auto Configuration）**：
   * 通过@EnableAutoConfiguration和spring.factories文件，Spring Boot根据当前环境和配置文件，自动装配所需的组件（如数据库连接池、Web服务器、消息中间件等）。
5. **Bean的创建和注入**：
   * Spring容器根据自动装配和手动配置的规则创建应用所需的Bean并进行注入。
6. **执行CommandLineRunner或ApplicationRunner**：
   * 如果应用实现了CommandLineRunner或ApplicationRunner接口，在Spring Boot启动完成后，会执行它们的run()方法，通常用于执行一些启动时的初始化逻辑。
7. **启动内嵌Web服务器（如果是Web应用）**：
   * 如果是Web应用，Spring Boot会启动一个内嵌的Web服务器（如Tomcat、Jetty或Undertow），并将应用部署到该服务器上。
8. **应用启动完成**：
   * 启动完成后，Spring Boot应用开始监听端口，等待外部请求。

面向简历

SpringBoot

SSM

AOP

IOC

知识框架

Spring 基础

Spring的重要模块（4）

**Core Container（核心容器）**

* **Spring Core**：提供了IoC和DI，是其他模块的基础。
* **Spring Beans**：负责管理Bean及其生命周期。
* **Spring Context**：基于Core和Beans的高级容器，ApplicationContext主要提供的5大功能：
  + 核心容器 BeanFactory
  + 国际化 MessageSource：根据浏览器请求头携带的信息翻译为某一特定的语言。
  + 资源获取 ResourceLoader：context.getResources("file:xxx");
  + 环境信息 EnvironmentCapable：context.getEnvironment().getProperty("java\_home");
  + 事件发布 ApplicationEventPublisher
* **Spring Expression Language（SpEL）**：一个强大的表达式语言，用于在运行时查询和操作对象的值。

**AOP（面向切面编程）**

在方法执行前后或抛出异常时，**动态插入额外的逻辑**，比如**日志记录、权限验证、事务管理**等。

**Data Access（数据访问）**

* **Spring JDBC**：简化了原生JDBC的管理连接、资源释放和异常处理。
* **Spring ORM**：支持与主流ORM框架（如Hibernate、JPA、MyBatis等）集成，简化持久层开发。
* **Spring Transaction**：提供声明式和编程式的事务管理机制，与数据库操作密切结合。

**Web层**

* **Spring Web**：提供基础的Web开发支持，包括Servlet API的集成，适用于构建MVC架构。
* **Spring MVC**：实现了Model-View-Controller（MVC）模式的框架，用于构建基于HTTP请求的Web应用。它是一个常用的模块，支持注解驱动的Web开发。
* **Spring WebFlux**：提供基于Reactive Streams的响应式编程模型，专为高并发的异步非阻塞请求设计。

**扩展模块**

* **Spring Batch**：批处理，支持大规模数据的处理与分块执行。
* **Spring Integration**：提供消息驱动的应用程序集成方案，适用于构建企业集成架构。
* **Spring Cloud**：用于构建微服务架构的模块集合，支持分布式系统中的服务注册、配置管理、服务调用等功能。

Spring的启动过程

1. **加载配置，初始化容器，实例化容器** 读取配置文件，配置数据库连接、事务管理、AOP 配置等。

创建容器 ApplicationContext，在容器启动阶段实例化 **BeanFactory**，并加载容器中的 BeanDefinitions。

1. **解析 BeanDefinitions，Bean的生命周期**

Spring 容器会解析配置文件中的 BeanDefinitions（包括 Bean 的作用域、依赖关系等信息）。

实例化 Bean、依赖注入、初始化

1. **发布事件**
2. **完成启动**

**BeanFactory、ApplicationContext、FactoryBean的区别**

* **BeanFactory**是 是 Spring 的基础 IoC 容器，提供 Bean 的创建与管理功能，延迟加载 Bean。
* **ApplicationContext（推荐）** ：是 BeanFactory 的扩展，添加了**国际化、事件发布、AOP** 等高级功能，**默认预加载所有单例 Bean**。ClassPathXmlApplicationContext#refresh。
* **FactoryBean**：通过自定义的 getObject() 方法创建对象。AOP 使用 FactoryBean 生成代理对象。

**Spring、SpringMVC、SpringBoot的区别**

1. Spring（SpringFramework）：包含多个核心模块、简化了J2EE（Java 2 平台企业版）的开发
2. SpringMVC：多个功能模块之一，基于Spring提供的IoC、AOP 等功能来实现 Web 层的处理
3. SpringBoot：自动配置常用Spring模块，简化初始化和配置工作。

Spring 基础 扩展知识

**Spring框架中如何实现国际化（i18n）和本地化（l10n）？**

Spring 框架通过 **MessageSource 接口**，结合 **LocaleResolver 和 @RequestMapping**，可**根据用户的语言偏好或请求参数**，**动态切换语言**，**动态加载消息资源文件**。

🍃Spring IoC

控制反转 IoC

控制反转 IoC是【**思想**】：指将**对象的创建和管理**的控制权从**程序代码**转移到 **Spring 容器**。

* **导入和配置。** 导入坐标spring-context后新建配置文件applicationContext.xml并配置bean（包括id，class）
* **获取IoC容器。** new ClassPathXmlApplicationContext("applicationContext.xml")
* **获取bean。** ctx.getBean("bookDao")

依赖注入DI

依赖注入 DI是【**实现方式**】：指容器**根据配置**将**所需的依赖**注入到**对象**中

依赖注入的3种方式

1. **Setter 注入**
   * 适用于可选依赖的注入、自己开发的模块的注入
   * 如果只有Setter方法，没有注入，会导致null对象出现。
   * 配置<bean>用<property>标签的value或ref属性。
2. **构造器注入（推荐）**
   * 适用于强制依赖的注入、第三方不含setter的模块的注入
   * 构造器注入结合 @Qualifier 明确依赖关系，可以避免多个依赖注入混乱。
   * 配置<bean>用<constructor-arg>标签的value或ref属性。
3. **自动装配（不推荐）**
   * 不推荐原因：**隐式依赖关系**导致的**可测试性差**和不易维护。 具体来说，不能通过构造器或 setter 方法修改依赖关系，因为 Spring 会在创建 Bean 时自动完成依赖注入，测试代码无法显式地控制 Mock （模仿）对象的注入，只能使用 @InjectMocks 和 @Mock 等注解来让 Mockito 和 Spring 协作进行注入，或者采用构造器注入来手动控制 Mock 对象的注入。
   * 配置<bean>用autowire="xx"。
4. 接口回调注入
   * 例如，实现BeanFactoryAware进行注入

属性注入的3种情况

1. **简单类型（如int、String）**。用<property>标签的value属性
2. **引用类型**。用<property>标签的ref属性
3. **集合**。用<property>标签的<array>``<list>``<set>``<map>``<props>子标签

配置文件的注入

@PropertySource按路径加载配置文件

@Value注入值

Environment对象注入值

Bean的配置

**Spring Bean**： **Spring 容器管理的 Java 对象** 。

注册到容器

1. 基于 XML 的配置
2. 基于 @Component 注解及其衍生注解
   1. **@Component**：通用注解，用于没有明确职责的类或通用组件，比如工具类、任务调度器等。
   2. **@Controller**：处理 Web 层请求。
   3. **@Service**：处理服务层逻辑。
   4. **@Repository**：处理持久层数据访问对象。@Repository 注解会自动将**低层次的数据库异常（如 SQLException）**转换为 **Spring 统一的 DataAccessException**。
3. 基于 @Configuration声明配置类 和 @Bean
   1. @Bean + @Configuration：方法级别、手动注册、用于第三方库或无法修改的类、更灵活
   2. @Component + @ComponentScan：类级别、自动扫描、用于自定义类、自动化更强
4. 基于 @Import 注解将普通类导入到 Spring 容器中，这些类会自动被注册为 Bean。

自动装配

1. no（默认）：不自动装配，需要显式地定义依赖。
2. byType：通过 Bean 类型进行自动装配。
3. byName：通过 Bean 名称进行自动装配。
4. constructor：通过构造函数进行自动装配。

**@Autowired 和 @Resource 的区别**

都是**自动装配Bean**的注解

* **@Autowired：** 默认**按类型（byType）** 、Spring 特有、 支持@Autowired(required = false)依赖项可选，即没有匹配到时注入 null不抛异常。 支持结合 @Qualifier 注解指定注入的 Bean 名称。
* **@Resource：** 默认**按名称（byName）** 、JSR-250 标准（更适合标准化和跨框架）、 如果未指定名称且没有匹配的 Bean，会按类型注入。

**@Qualifier 和 @Primary 的区别**

都是**解决 Bean 注入时的歧义问题**的注解

* **@Qualifier**：指定**名称**选择对应的实现 Bean。
* **@Primary** ：指定**默认**注入哪个 Bean。

注：@Qualifier 可以覆盖 @Primary 的默认行为。

条件配置

**@Profile 指定环境**

@Profile 用于定义Bean 的配置文件所属的环境，比如 dev 通常表示开发环境，prod 表示生产环境。

@Configuration  
@Profile("dev") // 这个配置类只在 "dev" profile激活时加载  
public class DevConfig {  
    @Bean  
    public DataSource dataSource() {  
        // 返回开发环境的DataSource  
        return new EmbeddedDatabaseDataSource();  
   }  
}  
​  
@Configuration  
@Profile("prod") // 这个配置类只在 "prod" profile激活时加载  
public class ProdConfig {  
    @Bean  
    public DataSource dataSource() {  
        // 返回生产环境的DataSource  
        return new SomeProductionDataSource();  
   }  
}

**@Conditional条件加载**

* @ConditionalOnProperty 根据配置文件的属性值来决定是否装配。常用于启用或禁用某些功能模块。
* @ConditionalOnClass
* @ConditionalOnMissingBean 避免重复注册 Bean。
* @ConditionalOnBean 当容器中存在某个类型的 Bean 时，才装配当前的 Bean。

Bean 的作用域

**作用域（2+4）**

**2 基础作用域 + 4 仅Web 应用可用：**

1. **singleton（默认）：** IoC 容器中只有唯一的 bean 实例。适用于无状态的共享资源。
2. **prototype：** 每次获取bean都创建一个新实例。适用于短期使用的有状态且非线程安全的对象。 应用：唯一标识符（UUID 、验证码、Token）、用户临时数据（表单数据缓存、文件上传临时存储）
3. **request：** 每个 HTTP 请求创建一个实例，如表单数据处理。
4. **session：** 每个会话创建一个实例，如用户信息缓存。
5. **application/global-session：** 在Web应用启动时创建一个实例，如统计数据。
6. **websocket：** 每个 WebSocket 会话创建一个实例，适用于WebSocket 连接状态管理。

**作用域与生命周期的关系**

* **singleton**：Bean 的生命周期与 Spring 容器的生命周期一致。在容器启动时创建，在容器关闭时销毁。
* **prototype**：容器只负责创建，不管理其生命周期、不调用销毁方法，客户端决定何时销毁。
* 其他：Bean 的生命周期分别与 HTTP 请求、会话、应用或 WebSocket 的生命周期一致。

**有状态的单例bean是非线程安全的**

解决：

1. **避免在单例 Bean 中使用可变状态**
2. **使用prototype作用域**
3. **加锁**: 利用 synchronized 或 ReentrantLock
4. **使用ThreadLocal保存变量**

Bean 的生命周期及其应用（543）

1. **实例化：**Spring 容器通过**反射**根据**配置文件或注解**实例化 Bean 对象。
2. **依赖注入**
3. **初始化**
   1. **Aware注入**：Bean 可以实现 Aware 接口获取BeanFactory、ApplicationContext 等容器资源
      * BeanNameAware
      * BeanClassLoaderAware
      * BeanFactoryAware
   2. **BeanPostProcessor#postProcessBeforeInitialization**（应用：创建动态代理，实现AOP）
   3. 初始化Initialization（应用：缓存初始化）
      * @PostConstruct
      * InitializingBean#afterPropertiesSet
      * initMethod
   4. **BeanPostProcessor#postProcessAfterInitialization**
4. **使用**
5. **销毁@PreDestroy：** DisposableBean#destroy。 常用于进行资源释放（关闭数据库连接、文件句柄、线程池）、会话管理（清理用户会话或缓存） 注意：容器关闭时才调用销毁逻辑，所以看不到，要想看到destroy-method执行：
   1. 暴力手动关闭：ClassPathXmlApplication#close
   2. 注册关闭钩子：ClassPathXmlApplication#registerShutdownHook()

🍃Spring AOP

AOP核心思想

AOP（Aspect-Oriented Programming，面向切面编程）通过切面将通用功能（如**权限、日志、事务、性能**等）模块化，避免代码重复。通俗理解，通过代理的方式，先拦截穿插、再调用真正方法实现。

AOP相关术语

**切面(Aspect) = 切点(Pointcut) + 通知(Advice)**

* **切面（Aspect）：** **横切关注点的模块**，封装了不同模块共享的功能
* **切点（Pointcut）：** 切点是一个表达式，通过这个表达式可以找到想要织入的哪些方法
  + **连接点（Join Point）：** 目标对象的所属类中，定义的所有方法均为连接点
  + 切点是特殊的连接点
* **通知（Advice）：** 拦截到目标对象的连接点之后**要执行的操作**
  + **目标（Target）：** 被切面增强的对象
  + **代理（Proxy）：** 代理对象包含目标对象的原始方法和增强逻辑
  + **织入（Weaving）：** 将切面应用到目标对象上的过程，Spring AOP 是运行时织入

**常见的通知类型**

* **前置通知（Before）**
* **后置通知（After）**
* **环绕通知（Around）**
* **异常通知（AfterThrowing）**
* **返回通知（AfterReturning）**

Spring4版本

1. 环绕之前通知
2. 前置通知Before
3. 被增强的方法
4. **环绕之后通知 / 无（异常时）**
5. 后置通知After
6. **AfterReturning 返回通知 / AfterThrowing 异常通知（异常时）**

Spring5版本

1. 环绕之前通知
2. 前置通知Before
3. 被增强的方法
4. **AfterReturning 返回通知 / AfterThrowing 异常通知（异常时）**
5. 后置通知After
6. **环绕之后通知 / 无（异常时）**

AOP的工作机制

**JDK动态代理和CGLIB 动态代理的区别**

**JDK 动态代理**（SpringFramework的默认实现）

* 基于**接口**实现，通过 Java 的**反射**机制实现
* 适合实现**接口**的的类，没有接口的话会有报错

**CGLIB 动态代理**（SpringBoot2的默认实现）

* 基于**类继承**，通过**ASM 字节码生成工具**生成继承目标类的子类。
* 不能代理 final 类和 final 方法。
* 适合**没有接口的类**。

Spring AOP 和 AspectJ AOP 的区别

**Spring AOP**：

* 基于动态代理，仅支持运行时AOP
* 更轻量、更方便，适合大部分业务场景，但仅适用于 Spring 容器管理的对象

**AspectJ AOP**：

* 支持编译时、类加载时、运行时AOP
* 更灵活、性能高，适合日志、监控

🍃Spring MVC

Model、View、Controller

* **Model（模型）：** 负责封装数据，POJO、DTO 对象。
* **View（视图）：** 负责展示数据，通常是 JSP、Thymeleaf 等模板引擎。
* **Controller（控制器）：** 处理请求，调用服务层处理业务逻辑，最终返回数据和视图。

MVC的工作流程

1. **请求**：客户端发起请求，到达DispatcherServlet
2. **（拦截）映射、适配处理、返回（拦截）**
   * **HandlerInterceptor#preHandle()拦截**：返回true继续处理，用于权限、日志、异常、性能
   * **HandlerMapping映射**：根据请求的 URL、HTTP 方法映射合适的Controller
   * **HandlerAdapter适配处理**：用HandlerAdapter调用Controller处理
   * **返回**
     + 返回 **ModelAndView 对象（数据模型、视图名称）**
     + 返回 JSON 对象。
   * **HandlerInterceptor#postHandle()拦截**
3. **解析、渲染、响应**
   * **ViewResolver解析**
     + 对于视图：将逻辑视图名称解析为实际的视图（如 JSP、Thymeleaf ）
     + 对于 JSON ：将对象序列化为 JSON返回给客户端
   * **视图渲染引擎渲染**：根据 Model 中的数据渲染HTML页面
   * **响应**：将渲染的视图或 JSON 数据响应给客户端。

**拦截器（HandlerInterceptor）**

用于权限、日志、异常、性能。

1. 实现 HandlerInterceptor 接口并重写其三个核心方法：
   * preHandle()：请求到达控制器之前的预处理。
   * postHandle()：控制器执行之后但视图渲染之前的后处理。
   * afterCompletion()：整个请求结束之后的回调。
2. 通过 WebMvcConfigurer 或 xml 配置来**注册拦截器**并指定拦截的路径。

**拦截链及其工作原理**

拦截链是一系列拦截器（如 AOP 切面、过滤器、拦截器）的统称。

1. 根据 @Before、@After等注解，往集合里面加入对应的 **MethodInterceptor** 实现
2. 通过 CglibMethodInvocation 对象封装集合后调用**CglibMethodInvocation#proceed** 具体来说，借助 **currentInterceptorIndex 下标**，递归顺序地执行集合里面的 MethodInterceptor
3. 完成了拦截链的调用。

**拦截器、过滤器、切面的区别**

* **HandlerInterceptor（拦截器）**：基于 Spring MVC ，通常用于**应用程序级别的日志记录、权限验证**等。
* **Filter（过滤器）**：基于 Servlet，过滤所有的 HTTP 请求，通常用于**全局的跨域请求处理、编码转换**等。
* **AOP 拦截链（切面）**：切面中的 @Before、@After、@Around 等注解用于控制拦截的执行顺序。

MVC的数据绑定和校验

**数据绑定（@RequestParam、@RequestBody、@ModelAttribute）**

* @RequestParam 用于绑定简单参数，用于单个请求参数
* @RequestBody：用于表单数据是以 **JSON** 格式在请求体提交的，映射为 Java 对象。
* @ModelAttribute 用于绑定复杂对象，将表单字段与 **Java 对象**的属性进行绑定

**数据类型转换（PropertyEditor、@InitBinder）**

自定义 PropertyEditor 用于扩展 Spring 的数据绑定机制，适配特殊的数据类型转换。实现步骤：

1. **实现 PropertyEditorSupport 子类**：定义转换逻辑。
2. **注册 PropertyEditor**：通过 @InitBinder 方法将自定义的 PropertyEditor 注册到 WebDataBinder 中。
3. **使用自定义类型**：自动将请求参数绑定到目标对象的复杂类型字段。

**数据校验（@Valid、@Validated）**

@Valid 注解用于触发表单对象的验证，BindingResult 用于检查验证结果。 Spring MVC 支持使用 @NotNull、@Size、@Email 等定义验证规则。 如果表单验证失败，控制器可以根据错误结果返回错误页面。

* **@Valid**：适用于单一或嵌套对象
* **@Validated**：支持分组验证（groups = XX.class），可以根据不同的场景执行不同的验证规则

MVC的异常处理机制

**局部异常处理 @ExceptionHandler**

用于局部的异常处理，通常定义在Controller类中，捕获特定的异常，并返回自定义的错误信息或视图。

**优先级高：**当控制器类中有 @ExceptionHandler 注解的方法时，Spring 会优先调用该局部异常处理方法。如果没有找到局部的异常处理方法，Spring 会调用全局的异常处理器。

// 该方法可以处理 UserNotFoundException 和 InvalidUserException 两种类型的异常。  
@ExceptionHandler({UserNotFoundException.class, InvalidUserException.class})  
public String handleMultipleExceptions(Exception ex, Model model) {  
    model.addAttribute("errorMessage", ex.getMessage());  
    return "errorPage";  
}

**全局异常处理 @ControllerAdvice**

通过 @ControllerAdvice 和 @ExceptionHandler 定义全局异常处理，适用于所有控制器，避免重复代码。

// GlobalExceptionHandler 可以处理所有控制器中抛出的异常，包括 UserNotFoundException 和其他类型的异常。  
@ControllerAdvice  
public class GlobalExceptionHandler {  
    @ExceptionHandler(Exception.class)  
    public ResponseEntity<String> handleGlobalException(Exception ex) {  
        return new ResponseEntity<>("Global error: " + ex.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);  
   }  
​  
    @ExceptionHandler(UserNotFoundException.class)  
    public ResponseEntity<String> handleUserNotFoundException(UserNotFoundException ex) {  
        return new ResponseEntity<>("User not found: " + ex.getMessage(), HttpStatus.NOT\_FOUND);  
   }  
}

**设置HTTP 响应状态码（@ResponseStatus、@ResponseEntity）**

@ResponseStatus用于定义异常类对应的 HTTP 状态码和原因短语。

@ResponseStatus(value = HttpStatus.NOT\_FOUND, reason = "面试鸭未找到")  
public class UserNotFoundException extends RuntimeException {  
}

@ResponseEntity自定义响应结构 ResponseEntity来返回标准化的错误格式

// 自定义响应结构 `ResponseEntity`   
@ExceptionHandler(UserNotFoundException.class)  
@ResponseBody  
public ResponseEntity<?> handleUserNotFoundException(UserNotFoundException ex) {  
    Map<String, String> errorResponse = new HashMap<>();  
    errorResponse.put("error", "Mianshiya User not found");  
    errorResponse.put("message", ex.getMessage());  
    return new ResponseEntity<>(errorResponse, HttpStatus.NOT\_FOUND);  
}

// 此时，客户端会收到类似这样的 JSON 响应  
{  
    "error": "Mianshiya User not found",  
    "message": "The user with ID 123 was not found"  
}

Rest风格

**REST（Representational State Transfer, （资源的）表述性状态转移）** **用 URL 定位资源，用 HTTP 动词来描述操作**，使得客户端和服务器之间的交互更加简单、清晰和高效。

@RequestMapping 请求映射

@GetMapping、@PostMapping、@PutMapping、@DeleteMapping

假设有一个用户资源，URI 为 /users，RESTful API 设计如下：  
   GET /users：获取所有用户。  
   GET /users/{id}：获取指定 ID 的用户。  
   POST /users：创建新用户。  
   PUT /users/{id}：更新指定 ID 的用户。  
   DELETE /users/{id}：删除指定 ID 的用户。

**幂等性：**无论请求执行多少次，其结果应该相同。

* **GET**、**PUT**、**DELETE**：这些操作应该是幂等的。
* **POST**：通常是非幂等的，因为每次调用 POST 可能都会创建新的资源。

**@PathVariable 与 @RequestParam 的区别**

* **@PathVariable**：从 **URL 路径**中提取动态数据，常用于 **RESTful URL 模式**。如 /users/{id}。
* **@RequestParam**：从**请求的查询参数**中提取数据，通常用于处理**表单数据或查询参数**。如 /users?id=1。

**@RequestHeader 、 @CookieValue 、@SessionAttribute 获取值**

@RequestHeader 注解用于提取 HTTP 请求头中的值，并将其注入到控制器方法的参数中。例如访问 Accept、Content-Type、User-Agent 等请求头信息。

@GetMapping("/header-info")  
public String getHeaderInfo(@RequestHeader("User-Agent") String userAgent) {  
    // 使用 userAgent 进行业务处理  
    return "headerInfoView";  
}  
​

@CookieValue 注解用于从 HTTP 请求的 Cookie 中提取值，并将其注入到控制器方法的参数中。

@GetMapping("/cookie-info")  
public String getCookieInfo(@CookieValue("sessionId") String sessionId) {  
 // 使用 sessionId 进行业务处理  
 return "cookieInfoView";  
}

@SessionAttribute 是 Spring MVC 中的注解，用于从当前 HTTP 会话中获取属性值并将其绑定到控制器方法的参数上，而无需手动从 HttpSession 获取。

// Spring 从会话中提取名为 "loggedInUser" 的属性值，并将其绑定到 User 对象中，传递给控制器方法。  
@GetMapping("/profile")  
public String getUserProfile(@SessionAttribute("loggedInUser") User user) {  
 return "User Profile: " + user.getName();  
}

@RequestBody 和 @ResponseBody 请求和响应

* **@RequestBody**：将 **请求体中的数据**绑定到**方法参数**上。Spring 会将 JSON格式的请求体映射为 Java 对象。
* **@ResponseBody**：将**方法的返回结果**写入 **HTTP 响应体**中。通常用于返回 JSON 格式的数据。
  + @RestController 是 @Controller 和 @ResponseBody 的组合注解，表示该类的所有控制器方法都默认返回 JSON 数据，而不需要为每个方法单独加 @ResponseBody。
* Spring 根据请求头中的 Content-Type、响应头中的 Accept，选择合适的消息转换器，实现 JSON和 Java 对象的相互转换。

**常见的消息转换器（HttpMessageConverter）**

* **MappingJackson2HttpMessageConverter**：实现 JSON 数据和 Java 对象互相转换。
* **Jaxb2RootElementHttpMessageConverter**：实现 XML 数据和 Java 对象互相转换。
* **StringHttpMessageConverter**：处理 text/plain 类型的请求和响应，将请求体转换为字符串。

**MVC、MVVM、WebFlux的区别**

* MVC（Model, View, Controller）：View 和 Model 直接交互，由 Controller 调节，适合后端 Web 框架。
* MVVM（Model, View, ViewModel）：依赖 ViewModel 双向绑定自动更新，适合前端框架（Vue、React）
* MVC：基于 Servlet API，同步阻塞、每个请求对应一个线程，适用于大多数 Web 应用。
* WebFlux：基于 Reactor 库，响应式、异步非阻塞的，适用于 I/O 操作频繁、高并发、低延迟的应用。

🍃Spring 循环依赖

什么是循环依赖

循环依赖是指多个 Bean **循环引用**导致 Spring 容器**无法正常初始化**它们。

例如，A 要依赖 B，发现 B 还没创建，于是开始创建 B ，创建的过程发现 B 要依赖 A， 而 A 还没创建好呀，因为它要等 B 创建好，就这样它们俩就搁这卡 bug 了。

通过三级缓存解决循环依赖

**三级缓存**

1. **一级缓存（Singleton Objects Map 单例对象）**：存放已经实例化、属性填充、初始化的 Bean。
2. **二级缓存（Early Singleton Objects Map 早期单例对象）**：存放已实例化，但尚未完全初始化的 Bean。也就是三级缓存中ObjectFactory产生的对象，与三级缓存配合使用的，可以防止 AOP 的情况下，每次调用ObjectFactory.getObject()都是会产生新的代理对象的。
3. **三级缓存（Singleton Factories Map 单例工厂）**：存放对象工厂ObjectFactory，ObjectFactory的getObject()方法最终调用getEarlyBeanReference()方法创建早期Bean（特别是为了支持AOP代理对象的创建）。

**Spring创建Bean流程**

1. 当 Spring 创建 A 时发现 A 依赖了 B ，又去创建 B，B 依赖了 A ，又去创建 A；
2. 在 B 创建 A 的时候， A 此时还没有初始化完成，因此在 一二级缓存 中肯定没有 A；
3. 那么此时就去三级缓存中调用 getObject() 方法最终调用getEarlyBeanReference() 方法去生成并获取 A 的前期暴露的对象。
4. 然后就将这个 ObjectFactory 从三级缓存中移除，并且将前期暴露对象放入到二级缓存中，那么 B 就将它注入到依赖，来支持循环依赖。

**只用两级缓存够不够**

* 在没有 AOP 的情况下，确实可以只使用一级和三级缓存来解决循环依赖问题。
* **当涉及到 AOP 时，二级缓存非常重要，避免了同一个 Bean 有多个代理对象的问题。**
  + 直接使用二级缓存不做任何处理会导致我们拿到的 Bean 是未代理的原始对象。
  + 如果二级缓存内存放的都是代理对象，则违反了 Bean 的生命周期。
* getEarlyBeanReference会判断这个对象是否需要代理，如果否则直接返回，如果是则返回代理对象。

**ObjectFactory 和 Provider 的区别**

**ObjectFactory 的使用场景**

* **懒加载 Bean**：当某个 Bean 的创建过程可能耗时较长、依赖的资源较重、或需要在运行时决定动态 Bean 使用时，可以通过 ObjectFactory 进行懒加载，避免容器启动时不必要的 Bean 创建。
* **避免循环依赖**：在某些情况下，两个 Bean 可能相互依赖，导致循环依赖问题。通过使用 ObjectFactory，可以延迟其中一个 Bean 的创建，避免循环依赖。在 Spring 的循环依赖的三级缓存的 map 里面存储的就是 ObjectFactory，用于延迟代理对象的创建。

**ObjectFactory和Provider 的区别**

二者在功能上非常相似，都提供了惰性获取 Bean 实例的机制。

* ObjectFactory 是 Spring 内部的接口。
* Provider 则是 Java 标准中的一部分，适用于更通用的场景。

通过@Lazy解决循环依赖

@Lazy 延迟加载

* **没有 @Lazy 的情况下**：在 Spring 容器初始化 A 时会立即尝试创建 B，而在创建 B 的过程中又会尝试创建 A，最终导致循环依赖。
* **使用 @Lazy 的情况下**： A 的构造器上添加 @Lazy 注解之后（延迟 Bean B 的实例化），Spring 不会立即创建 B，而是会注入一个 B 的代理对象。由于此时 B 仍未被真正初始化，A 的初始化可以顺利完成。等到 A 实例实际调用 B 的方法时，代理对象才会触发 B 的真正初始化，在注入 B 中的 A 属性时，此时 A 已经创建完毕了，就可以将 A 给注入进去。

Spring 事务

数据一致性问题

* **脏读（Dirty Read）**：读取了**尚未提交的事务的数据**，如果回滚，则不一致。
* **不可重复读（Non-repeatable Read）**：多次读取前后**数据**不一致，因为其他事务**修改并提交了**该数据。
* **幻读（Phantom Read）**：多次查询的**结果集**不同，因为其他事务**插、删了数据**。

事务隔离级别（Isolation）（5）

Spring 提供了五种事务隔离级别@Transactional(isolation = Isolation.XXX)：

1. **DEFAULT：**通常默认为 READ\_COMMITTED。
2. **READ\_UNCOMMITTED（读未提交）：**最低的隔离级别，允许事务读取尚未提交的数据，可能导致脏读、不可重复读和幻读。
3. **READ\_COMMITTED（读已提交，默认）：**仅允许读取已经提交的数据，**避免了脏读**，但可能会出现不可重复读和幻读问题。
4. **REPEATABLE\_READ（可重复读）：**确保在同一个事务内的多次读取结果一致，**避免脏读和不可重复读**，但可能会有幻读问题。
5. **SERIALIZABLE（可串行化）：**最高的隔离级别，通过强制事务按顺序执行，**完全避免脏读、不可重复读和幻读**，代价是性能显著下降。

事务传播行为（Propagation）（7）

@Transactional(propagation = Propagation.REQUIRES\_NEW)

1. **PROPAGATION\_REQUIRED（默认）：** 如果当前存在事务，则用当前事务
2. **PROPAGATION\_SUPPORTS：** 支持当前事务，如果不存在，则以**非事务**方式执行
3. **PROPAGATION\_MANDATORY：** 支持当前事务，如果不存在，则**抛出异常**
4. **PROPAGATION\_REQUIRES\_NEW：** 创建一个新事务，如果存在当前事务，则挂起当前事务
   * 应用场景：**日志记录、通知服务**等。即使主事务失败，独立事务的操作也应该成功执行。
5. **PROPAGATION\_NOT\_SUPPORTED：** 不支持当前事务，**始终以非事务**方式执行
   * 应用场景：需要明确禁止事务的场景，比如**读取配置信息、不需要事务控制的数据查询**。
6. **PROPAGATION\_NEVER：** 不支持当前事务，**如果当前存在事务，则抛出异常**
   * 应用场景：需要保证绝对没有事务的场景，比如某些**不允许在事务中执行的数据库操作**。
7. **PROPAGATION\_NESTED：** 如果当前事务存在，则在嵌套事务中执行，**内层事务依赖外层事务**，如果外层失败，则会回滚内层，内层失败不影响外层。
   * 应用场景：需要部分回滚或局部事务的业务逻辑。比如，**订单中的部分操作可能会失败，但不希望整个订单回滚**。

事务失效的情况（8）

如果使用了 @Transactional 注解在以下几个情况下会导致事务失效：

1. **异常未被回滚。**rollbackFor 没设置对，比如默认没有任何设置（RuntimeException 或者 Error 才能捕获），则方法内抛出 IOException 则不会回滚，需要配置 @Transactional(rollbackFor = Exception.class)。
2. **异常被捕获。**异常被捕获了，比如代码抛错，但是被 catch 了，仅打了 log 没有抛出异常，这样事务无法正常获取到错误，因此不会回滚。
3. **同类方法直接调用。**同一个类中方法调用，因此事务是基于动态代理实现的，同类的方法调用不会走代理方法，因此事务自然就失效了。
4. **非 public 方法。**事务仅适用于公共方法。@Transactional 应用在非 public 修饰的方法上，Spring 事务管理器判断非公共方法则不应用事务。
5. **final 或 static 方法。**@Transactional 应用在 final 和 static 方法上，因为 aop （Spring Boot2.x版本默认是 cglib，Spring 自身默认是 jdk，一般现在用的都是 SpringBoot）默认是 cglib 代理，无法对 final 方法子类化。static 是静态方法，属于类，不属于实例对象，无法被代理！
6. **事务隔离导致数据不一致。**propagation 传播机制配置错误，例如以下的代码（伪代码，忽略同一个类中的方法调用影响代理的情况）因为配置了 Propagation.REQUIRES\_NEW，是新起了一个事务，即 addAddress 的事务和 addUserAndAddress 其实不是一个事务，因此两个事务之间当然就无法保证数据的一致性了。
7. **线程上下文无法同步事务。**多线程环境，因为 @Transactional 是基于 ThreadLocal 存储上下文的，多线程情况下每个线程都有自己的上下文，那么之间如何保持事务同步？保持不了，因此事务失效。
8. **不支持事务的引擎。**用的是MyISAM，这个引擎本身不支持事务!

Spring 事务 扩展知识

**JTA（Java Transaction API）**

JTA（Java Transaction API）是一种标准的事务管理 API，用于分布式事务管理。

配置步骤：

1. 配置 JTA 事务管理器（如Atomikos、Bitronix）。
2. 设置 JtaTransactionManager 并整合数据源。
3. 在代码中通过注解（@Transactional）或编程方式管理分布式事务。

Spring 设计模式

* **工厂模式（IoC）：** BeanFactory，整个IOC 就是一个工厂。
* **单例模式（Bean）：**默认情况下 Bean 都是单例的。
* **代理模式（AOP）：**AOP 整个都是代理模式。
* **责任链模式（MVC拦截器）：**多个拦截器串联起来就形成了责任链。
* **适配器模式（MVC适配器）：**在 Spring MVC 中提到的 handlerAdapter 其实就是适配器。
* **模板方法（Template）：**例如 JdbcTemplate、RestTemplate，名字是 xxxTemplate 的都是模板。
* **观察者模式（监听器）：**在 Spring 中的监听器实现。

Spring Data JPA（重在实战）

JPA 是 Java 官方定义的**持久化规范**，用于将 **Java 对象**映射到**数据库表**中。

JPA 的主要功能

1. **Repository 接口。**定义接口继承 JpaRepository 或 CrudRepository，自动提供常见的 CRUD 操作实现。
2. **方法名称推导查询。**根据接口方法名称推导出相应的查询语句，无需手动编写查询逻辑。
3. **动态查询。**通过 @Query 注解或 Criteria API 实现自定义查询。

JPA 的关键特性

* **实体映射**：将 Java 类映射为数据库表。
* **查询**：通过 JPQL（Java Persistence Query Language）编写查询语句。
* **缓存管理**：提供一级缓存的支持。
* **事务管理**：与数据库事务集成。

JPA 的主要实现包括 Hibernate、EclipseLink 等。

JPA 和 Hibernate 的区别

JPA是规范，Hibernate 是一个完整的 ORM 解决方案，JPA 可以使用 Hibernate 作为其底层 实现。

Hibernate 的优势和功能扩展

1. **HQL（Hibernate Query Language）。**Hibernate 提供了 HQL（Hibernate Query Language），它是一种面向对象的查询语言，支持通过实体类字段进行查询。HQL 比 JPQL 提供了更多的功能，并且直接与 Hibernate 的底层机制结合。
2. **延迟加载。**Hibernate 支持延迟加载，即当访问到某个实体的关联对象时才从数据库中加载该对象的数据，提升性能。JPA 也支持延迟加载，但 Hibernate 提供了更多控制选项。
3. **缓存机制。**Hibernate 提供了一级和二级缓存的支持。一级缓存是默认开启的，并且是与当前会话绑定的。二级缓存可以配置为应用级别的缓存，以提升查询性能。
4. **批量操作和批量获取。**Hibernate 提供了批量插入、更新和删除操作的支持，帮助优化大批量数据操作的性能。开发者可以通过配置批量大小来优化这些操作。
5. **自动生成数据库架构。**Hibernate 可以根据实体类的定义自动生成数据库表和外键关系，帮助开发者快速构建数据库结构。通过 hibernate.hbm2ddl.auto 配置项可以控制自动生成表结构的行为。

**Spring Data JPA 扩展知识**

* 如何使用 JPA 在数据库中非持久化一个字段？
* JPA 的审计功能是做什么的？有什么用？
* 实体之间的关联关系注解有哪些？

**Spring Data JPA 扩展知识**

* 控制请求访问权限的方法
* hasRole 和 hasAuthority 的区别
* 如何对密码进行加密？
* 如何优雅更换系统使用的加密算法？

Spring Security

Spring 扩展知识

Spring父子容器

**父容器（管理Service层、DAO层）**：通常是 Spring 应用上下文（ApplicationContext），如 ContextLoaderListener 加载的根容器。它主要用于管理应用程序的全局 Bean，如服务层（Service）、数据访问层（DAO）等。

**子容器（管理Web层）**：每个 DispatcherServlet 实例都会创建一个子容器，用于管理 Web 层（如控制器和拦截器）中的 Bean。

**关系**：子容器可以访问父容器的 Bean，反之不能。

**意义：**

* **Service层与 Web 层分离**：全局的 Bean，如数据库连接池、事务管理等，可以放在父容器中，而与 Web 相关的控制器放在子容器中。
* **避免 Bean 重复定义**：例如，DAO的配置可以在父容器中定义，Web 层的配置可以在子容器中定义，这样可以防止冲突。
* **分模块管理**

@Scheduled ？

**主要作用：**

* **定时任务执行**：@Scheduled 注解允许开发者定义一个方法，该方法会按照指定的时间规则定期执行。
* **支持多种时间配置**：支持固定延迟、固定速率以及基于 Cron 表达式的任务调度。

**使用场景：**

* 自动备份数据或定时清理系统资源。
* 定时发送通知或报告。
* 定时同步数据，如从第三方服务中定期拉取数据。

@Scheduled(fixedRate = 5000) // 每隔 5 秒执行一次  
public void performTask() {  
 System.out.println("Task executed at: " + new Date());  
}

@Cacheable、@CachePut、@CacheEvict？

* **@Cacheable**：方法执行前检查缓存，若存在缓存数据则直接返回，否则执行方法并将结果缓存。
* **@CachePut**：每次执行方法并将结果更新到缓存中。
* **@CacheEvict**：用于移除缓存数据，可清除单个或多个缓存条目。

如何配置Spring Cache？

**配置 Spring Cache：**

1. 启用缓存：使用 @EnableCaching。
2. 配置缓存实现（如 ConcurrentMap、Redis、EhCache）。
3. 使用注解标记缓存逻辑。

@EventListener ？

用于监听和处理事件。通过标注在方法上，@EventListener 可以使方法自动监听特定类型的事件，并在事件发布时触发执行。

* 当系统中某个状态变化时触发特定操作。
* 日志记录、监控系统、通知系统等需要响应事件的模块。

// 当 MyCustomEvent 事件被发布时，handleEvent() 方法会自动触发来处理该事件。.  
@Component  
public class MyEventListener {  
    @EventListener  
    public void handleEvent(MyCustomEvent event) {  
        System.out.println("Handling event: " + event.getMessage());  
   }  
}