Spring全家桶

注意, 框架这里不是面试重点, 故只给出了一些常考的通用题型, 大家不要在这里花太多时间.

Spring

IOC (高)

IOC控制反转就是将对象的控制权交给IOC容器

- 我们只需要在Spring的配置文件中配置对应的bean, 在Spring容器启动是, Spring会读取配置文件, 利用反射来初始化对应的bean. 当需要调用某些对象时, Spring就会从容器中取出对应的bean分配给你.

IOC中有另外一个比较重要的概念, 就是DI, Dependency Injection, 依赖注入.

- 就是Spring在运行期由容器将依赖关系注入到组件之中。
- 比如说某个XXXController中有个XXXService属性, 则Spring会通过空参构造创建XXXController对象, 然后使用set方法注入所依赖的XXXService对象

Spring的IOC用的是简单工厂模式+配置文件的方式实现解耦的.

IOC优点

- IOC优点是可以实现解耦.
- 也可以避免重复创建对象带来的内存占用. (不懂JVM可以不聊这一点)

    - 如果创建完对象使用完就销毁, 那么对象频繁的创建销毁, 必然带来频繁的gc
    - 无论是CMS还是G1都会有STW时间, 频繁gc必然会使性能降低.

IOC缺点

- 创建对象复杂了
- 反射创建对象慢, 也就导致了Springboot启动慢.

AOP (高)

AOP面向切面编程, 其实就是动态地对类方法做增强. 可以在方法执行的前后做一些事情, 能够将通用操作（例如事务处理、日志管理、权限控制等）封装起来.

AOP的底层是代理模式.

- 代理模式一般有三种, 静态代理, 动态代理. 动态代理分为JDK动态代理和CGlib动态代理.
- Spring的AOP用的就是动态代理. 有接口的情况下, 使用JDK动态代理. JDK动态代理通过动态创建接口的实现类来代理对象.
- 在没有接口的情况下, Spring使用CGlib动态代理. CGlib的动态代理通过动态创建被代理类的子类来代理对象.

Spring框架中都用到了哪些设计模式？（中）

(不要硬背自己不了解的设计模式就不用说)

- 工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；
- 单例模式：Bean默认为单例模式。

- 代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；
- 模板方法：用来解决代码重复的问题。比如.RestTemplate, RabbitTemplate, JpaTemplate。
- 适配器模式: spring MVC 的 HandlerAdapter
- 观察者模式: Spring Event事件驱动

Bean的注入方式有哪些 (中)

- 构造函数注入：通过类的构造函数来注入依赖项。
- Setter注入：通过类的Setter方法来注入依赖项。
- Field(字段）注入：直接在类的字段上使用注解（如@Autowired或@Resource)来注入依赖

Spring官方推荐构造函数注入，这种注入方式的优势如下：

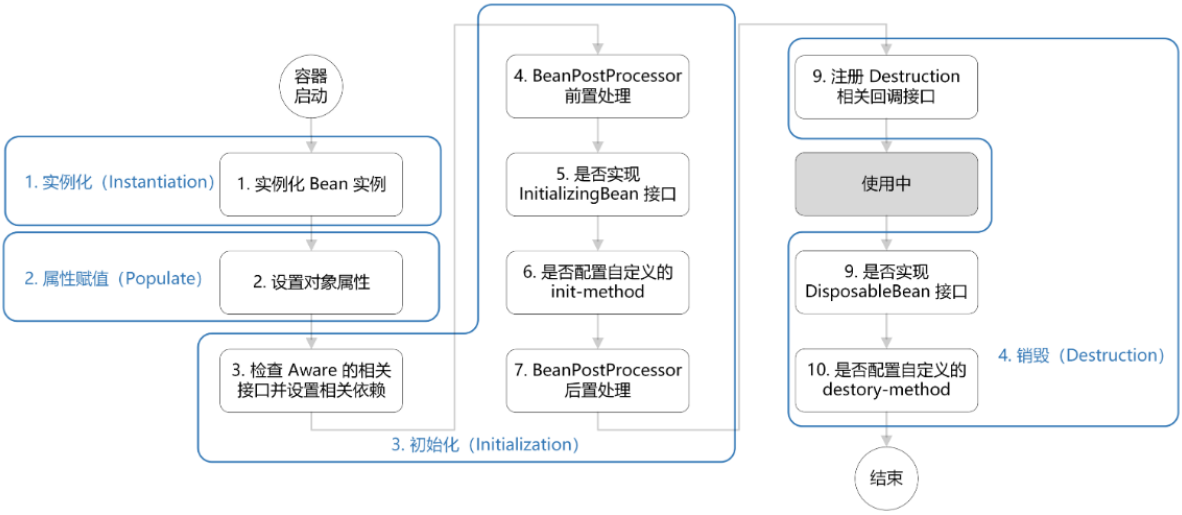
- 依赖完整性：确保所有必需依赖在对象创建时就被注入，避免了空指针异常的风险。
- 不可变性：有助于创建不可变对象，提高了线程安全性。
- 初始化保证：组件在使用前已完全初始化，减少了潜在的错误。
- 测试便利性：在单元测试中，可以直接通过构造函数传入模拟的依赖项，而不必依赖Spring容器进行注入。

bean的生命周期 (中)

(一些细节不明白建议网上查查, 不要死记硬背, 理解记忆)

1. 创建Bean的实例：Bean容器首先会使用反射来创建Bean的实例

2. Bean属性赋值：把@Autowired等注解注入的对象, 通过反射调用set方法赋值给bean
3. Bean初始化：
    - 检查Aware的相关接口并设置相关依赖
    - 执行初始化前置处理
        - 如果有和加载这个Bean的Spring容器相关的BeanPostProcessor对象，执行postProcessBeforeInitialization()方法
    - 初始化操作
        - 如果Bean实现了InitializingBean接口，执行afterPropertiesSet()方法。
        - 如果Bean在配置文件中的定义包含init-method属性，执行指定的方法。
    - 初始化后置处理
        - 如果有和加载这个Bean的Spring容器相关的BeanPostProcessor对象，执行postProcessAfterInitialization()方法。
4. 销毁Bean:
    - 如果Bean实现了DisposableBean接口，执行destroy()方法。
    - 如果Bean在配置文件中的定义包含destroy-method属性，执行指定的Bean销毁方法。或者，也可以直接通过@PreDestroy注解标记Bean销毁之前执行的方法



Spring 循环依赖 (中)

循环依赖是指 Bean 对象循环引用，A依赖B, B又依赖A. 那创建对象时, 创建A对象需要用到B对象, 此时尝试去创建B对象, 发现又要用到A对象.

Spring是通过三级缓存来解决循环依赖问题的

- 首先第1级缓存缓存的是一个完全初始化好的bean
- 第2级缓存缓存的是一个创建好了，但没有初始化好的bean
- 第3季缓存存的是创建bean的工厂对象

当我们获取一个对象的时候， Spring首先会从第1级缓存找有没有对应的bean，如果没有然后会去第2层缓存找，如果还是没有那么会从第3层缓存去找对应的工厂对象，然后去创建这个bean，然后将其放入到第2级的缓存中就可以用了。

举一个简单的例子， A和b循环依赖，现在要创建A对象

- 如果说spring在一二级缓存中找不到a，那么就会在第3级缓存去创建a的对象，将尚未初始化的a对象放入到第2级缓存中。
- 然后发现a依赖于b，那么spring会优先初始化被依赖的对象，然后第3级缓存创建了B对象，放入二级缓存中。

- 此时发现B依赖于a，然后从二级缓存中又找到了这个a对象，注入到b对象中。然后b对象初始化完成，将b对象放入一级缓存。
- 回过头来发现，a对象所依赖的b对象在一级缓存中已经有了，那么就直接将这个b对象注入给a。
- 然后循环依赖就解决了。

这种方式解决的前提是a和B都是单例的情况下，如果是非单例的情况, 循环依赖是无法解决的

Spring事务管理方式 (中)

- **编程式事务**：通过 `TransactionTemplate`或者 `TransactionManager` 手动管理事务
- **声明式事务**：@Transactiona, 通过aop实现的

Spring的事务失效的几种情况？ (低)

使用@Transactional注解来控制事务, 可能会出现以下几种失效的情况:

- 未捕获异常：如果一个事务方法中发生了异常，并且异常未被处理或传播到事务边界之外，那么事务会失效，所有的数据库操作会回滚。
- 非受检异常：默认情况下，Spring对非受检异常（RuntimeException或其子类）进行回滚处理，这意味着当事务方法中抛出这些异常时，事务会回滚。
- 事务传播属性设置不当：如果在多个事务之间存在事务嵌套，且事务传播属性配置不正确，可能导致事务失效。特别是在方法内部调用有@Transactional注解的方法时要特别注意。

- 多数据源的事务管理：如果在使用多数据源时，事务管理没有正确配置或者存在多个@Transactional注解时，可能会导致事务失效。
- 跨方法调用事务问题：如果一个事务方法内部调用另一个方法，而这个被调用的方法没有@Transactional注解，这种情况下外层事务可能会失效。
- 事务在非公开方法中失效：如果@Transactional注解标注在私有方法上或者非public方法上，事务也会失效。
- 使用this直接调用某方法: 因为Spring事务是通过代理对象来控制的，只有通过代理对象的方法调用才会应用事务管理的相关规则。当使用this直接调用时，是绕过了Spring的代理机制，因此事务会失效.

Spring中的Bean都是单例的吗 (低)

- Spring中的Bean默认都是单例的。每个Bean的实例只会被创建一次，并且会被存储在Spring容器中，以便重复使用
- Spring也支持将Bean设置为多例模式，每次请求都会创建一个新的Bean。设置scope为"prototype"可以将Bean设置为多例模式.

SpringMVC

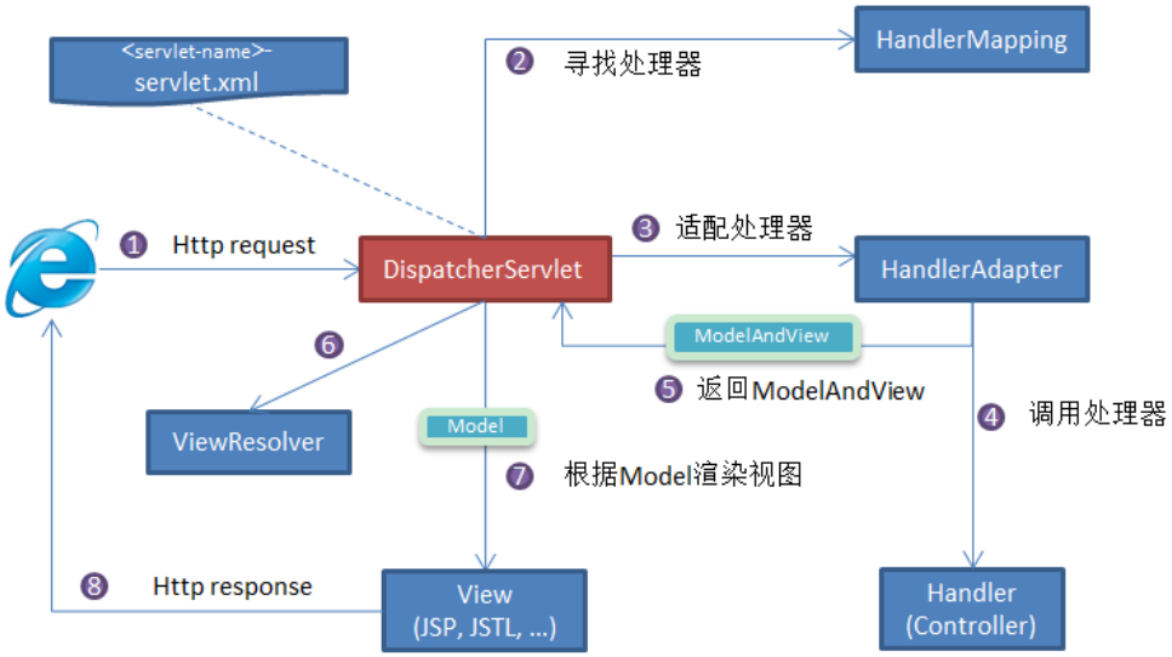
MVC分层是什么 (中)

MVC全名是Model View Controller, 模型(model), 视图(view), 控制器(controller). 它是一种软件设计规范，可以用一种业务逻辑、数据、界面显示分离的方法组织代码.

- 视图（view): 为用户提供使用界面，与用户直接进行交互。

- 模型（model): 模型分为两类, 一类称为数据承载Bean, 一类称为业务处理Bean。所谓数据承载Bean是指实体类（如：User类），专门为用户承载业务数据的；业务处理Bean则是指Service或Dao对象，专门用于处理用户提交请求的。
- 控制器（controller): 用于将用户请求转发给相应的Model进行处理，并根据Model的计算结果向用户提供相应响应。它使视图与模型分离。

springmvc工作原理/springmvc请求处理流程 (中)



1. 客户端（浏览器）发送请求，Dispatcherservlet拦截请求
2. Dispatcherservlet根据请求信息调用HandlerMapping。HandlerMapping根据URL去匹配查找能处理的Handler(这个Handler指的是我们自己写的Controller)
3. DispatcherServlet调用HandlerAdapter适配器执行Handler(这里涉及到了适配器模式)
4. Handler完成对用户请求的处理后，会返回一个ModelAndView对象给DispatcherServlet

    - ModelAndView顾名思义，包含了数据模型以及相应的视图的信息
5. ViewResolver视图解析去会根据逻辑View查找实际的View
6. DispaterServlet把返回的Model传给View(视图渲染）。
7. 把View返回给请求者（浏览器）

上述流程是传统开发模式（JSP,Thymeleaf等）的工作原理。

现在主流的开发方式是前后端分离，后端不会在管View了(不再负责渲染页面). View由前端框架（Vue,React等）来处理. 后端只会给前端返回JSON数据. 前端渲染数据到View上.

故前后端分离的请求处理流程如下

1. 客户端（浏览器）发送请求，Dispatcherservlet拦截请求
2. Dispatcherservlet根据请求信息调用HandlerMapping。HandlerMapping根据URL去匹配查找能处理的Handler(这个Handler指的是我们自己写的Controller)
3. DispatcherServlet调用HandlerAdapter适配器执行Handler(这里涉及到了适配器模式)
4. Handler完成对用户请求的处理后，会将返回转为JSON给DispatcherServlet
5. DispatcherServlet把JSON给前端, 前端框架（Vue,React等）来渲染数据.

Mybatis

为什么Mapper只需要声明接口和编写xml, 不需要实现? (中)

- Mapper接口会被动态代理, 然后根据对应的xml文件生成具体的代理类对象, 然后执行xml中的sql.

Mybatis里的#{}和${}的区别？ (中)

- Mybatis在处理#{}时，会创建预编译的SQL语句，将SQL中的#{}替换为？号，在执行SQL时会为预编译SQL中的占位符(?)赋值，调用PreparedStatement的set方法来赋值，预编译的SQL语句执行效率高，并且可以防止SQL注入，提供更高的安全性，适合传递参数值。
- Mybatis在处理${}时，只是创建普通的SQL语句，然后在执行SQL语句时MyBatis将参数直接拼入到SQL里，不能防止SQL注入，因为参数直接拼接到SQL语句中，如果参数未经过验证、过滤，可能会导致安全问题。

所以推荐使用#{}.

SpringBoot

Spring Boot自动装配 (中)

(这里加载spring.factories配置文件涉及到了spring的spi机制. 熟悉spi可以围绕spi聊, 不熟悉那就理解下面这段话即可)

- Spring Boot通过@EnableAutoConfiguration注解开启自动配置
- 然后会加载spring.factories中注册的各种AutoConfiguration类

- 当某个AutoConfiguration类满足其注解@Conditional指定的生效条件时，会去创建AutoConfiguration类中定义的 Bean，并注入Spring容器
- 后续我们使用时就可以直接注入自动配置的bean

对于一些实习生, 或者初学者来说, Springboot这里, 还会提问一些注解的作用. 诸如 `@Configuration` 或者 `@RequesttMapping` 等等, 但这都是开发常用注解, 自己按理解回答即可, 不多赘述.

SpringCloud

SpringCloud不是校招/实习的重点内容, 了解即可, 不需要太熟.

springcloud和springboot的区别 (中)

- Spring Boot是用于构建单个Spring应用的框架，而Spring Cloud则是用于构建分布式系统中的微服务架构的工具，Spring Cloud提供了服务注册与发现、负载均衡、断路器、网关等功能。
- SpringCloud像是一个大城市, Springboot是城市中的一个个居民房. SpringCloud城市中提供了发电站, 水利局, 医院, 警察局等公共设施, 方便居民楼使用.

微服务的特点/优缺点 (中)

微服务最大的特点就是解决了单点故障问题

优点:

- 解决了单体项目复杂性的问题
- 每个服务都可以由单独的团队进行开发
- 每个服务都可以使用单独的技术栈进行开发
- 每个服务都是独立地进行部署和维护
- 每个服务都可以进行独立的扩展

缺点:

- "微"的粒度不好把握 (高内聚低耦合)
- 微服务架构是一个分布式系统, 虽然单个服务本身变得简单了, 但是服务之间需要相互调用, 整个服务架构变得复杂了
- 微服务架构需要依赖分布式数据库架构
- 微服务的单元测试和调用变得更为复杂
- 部署基于微服务架构的应用程序变得更加复杂
- 微服务开发的技术成本更高

常见的微服务组件有哪些 (中)

- **注册中心：注册中心的作用是对新节点的注册与状态维护，解决了如何发现新节点以及检查各节点的运行状态的问题。**
    - 微服务节点在启动时会将自己的服务名称IP、端口等信息在注册中心登记
    - 注册中心会采用心跳机制定时检查该节点的运行状态
- **负载均衡：负载均衡解决了请求应该打到哪个节点上的问题.**
    - 因服务高可用的要求，服务调用者会从注册中心中发现有多个节点可供调用，必须要从中进行选择
    - 因此服务调用者一端必须内置负载均衡器，通过负载均衡策略选择合适的节点发起实质性的通信请求

- **服务通信：服务通信组件解决了服务间如何进行消息通信的问题**
  - 一般采用Feign或者Dubbo进行服务调用.
- **配置中心：配置中心主要解决了如何集中管理各节点配置文件的问题.**
  - 在微服务架构下，所有的微服务节点都包含自己的各种配置文件。如果将这些配置文件分散存储在节点上，发生配置更改就需要逐个节点调整，将给运维人员带来巨大的压力
  - 可以将各节点配置文件从服务中剥离，集中转存到配置中心。一般配置中心都有U界面，方便实现大规模集群配置调整
- **分布式链路追踪：分布式链路追踪解决了如何直观的了解各节点间的调用链路的问题**
  - 系统中一个复杂的业务流程，可能会出现连续调用多个微服务，我们需要了解完整的业务逻辑涉及的每个微服务的运行状态，通过可视化链路图展现，可以帮助开发人员快速分析系统瓶颈及出错的服务
- **服务保护：服务保护主要是解决了如何对系统进行链路保护，避免服务雪崩的问题**
  - 在业务运行时，微服务间互相调用支撑，如果某个微服务出现高延迟导致线程池满载，或是业务处理失败就很容易发生服务雪崩问题
  - 所以需要一些组件能提供熔断 降级 限流等功能保护服务.

负载均衡的意义 (中)

负载平衡能优化资源使用，最大化吞吐量，并避免任何单一资源的过载.

简单来说, 某个服务器只能承受100并发, 另外一个能承受200并发. 那么请求被转发到第一台服务器和第二台服务器的比例应该是1:2. 第二台服务器应该接受更多请求, 避免第一台服务接受更多请求产生过载.

负载均衡常见实现策略 (中)

- 简单轮询：将请求按顺序分发给后端服务器上，不关心服务器当前的状态，比如后端服务器的性能、当前的负载
- 加权轮询：根据服务器自身的性能给服务器设置不同的权重，将请求按顺序和权重分发给后端服务器，可以让性能高的机器处理更多的请求
- 简单随机：将请求随机分发给后端服务器上，请求越多，各个服务器接收到的请求越平均
- 加权随机：根据服务器自身的性能给服务器设置不同的权重，将请求按各个服务器的权重随机分发给后端服务器
- 一致性哈希：根据请求的客户端ip、或请求参数通过哈希算法得到一个数值，利用该数值取模映射出对应的后端服务器，这样能保证同一个客户端或相同参数的请求每次都使用同一台服务器
- 最小活跃数：统计每台服务器上当前正在处理的请求数，也就是请求活跃数，将请求分发给活跃数最少的后台服务器

服务熔断、服务降级是什么, 为什么需要熔断降级 (中)

在复杂的分布式系统中，微服务之间的相互调用，比如A调B, B调C, C调D. 如果D此时出现问题, 一直阻塞, 那么A B C都会阻塞住, 需要一直等D的结果. 此时请求量比较大的话, 就会造成A B C不可用, 这就是服务雪崩问题. 所以需要配置降级熔断措施.

服务降级：用户请求A服务，A服务调用B服务，当B服务出现故障或者在特定的时间段内不能给A服务响应，为了避免A服务因等待B服务而产生阻塞，A服务就不等B服务的结果了，直接给用户一个降级响应 (比如给默认值, 或者给一个信息"服务器开小差了")

服务熔断：用户请求A服务，A服务调用B服务，当B服务出现故障的频率过高达到特定间值时，当用户再请求A服务时，A服务将不再调用B服务，直接给用户一个降级响应 (比如给默认值, 或者给一个信息"服务器开小差了")