**框架相关**

**SpringMVC的Controller是如何将参数和前端传来的数据一一对应的**

​ 其实还是用到了servlet的取值方式，如果参数中有一个strring类型的name属性，那么它就会从request作用域中获取一个name的属性，如果获取到了就封装成一个String类型的，然后绑定到参数中

**Mybatis如何找到指定的Mapper的，如何完成查询的**

​ 通过他的namespace ，也就是他的命名空间，其实我们可以不写Mapper接口的，也能直接调用xml中的方法，通过他的sqlSessionFactory工厂中加载xml，然后找到他的namespace，然后调用指定的方法，但是我们用Mapper接口进行映射，并且将它交给Spring容器进行管理，简化了开发效率，并且降低代码的耦合度

**Spring的拦截器过滤器以及监听器的区别**

​ 拦截器（Spring）

​ 通过实现HandlerInterceptor接口进行使用，可以拦截请求并且处理，true同行，false拒绝

​ postHandle()：这个方法在handler执行后，但是DispatcherServlet 向客户端返回响应前被调用，在该方法中对用户请求request进行处理。

​ afterCompletion()：这个方法在 DispatcherServlet 完全处理完请求后被调用，可以在该方法中进行一些资源清理的操作。view渲染完成、dispatcherServlet返回之前执行

​ 过滤器

​ 通过实现java定义好的javax.servlet.Filter接口 进行使用，他是基于Servlet的，

​ doFilter(ServletRequest, ServletResponse, FilterChain)：这是一个完成过滤行为的方法。这同样是上游过滤器调用的方法。引入的FilterChain对象提供了后续过滤器所要调用的信息。如果该过滤器是过滤器链中的最后一个过滤器，则将请求交给被请求资源。也可以直接给客户端返回响应信息。init(FilterConfig)：由Web容器来调用完成过滤器的初始化工作。它保证了在第一次doFilter()调用前由容器调用。您能获取在 web.xml 文件中指定的初始化参数。destroy()：由Web容器来调用来释放资源，doFilter()中的所有活动都被该实例终止后，调用该方法。

​ 监听器

​ 实现ServletContextListener 接口

​ spring容器的加载要在servlet之后，因此在有些过滤器当中需要提前用到spring bean的时候，就需要改成 Listener 的方式

​ org.springframework.web.context.ContextLoaderListener

​ 自动装配ApplicationContext的配置信息。因为它实现了ServletContextListener这个接口，注：如果在xml中配置了初始化加载，然后在代码中再次地实现了ContextLoaderListener ，那么它会初始化两次

​ 区别：

​ 拦截器是Spring所封装的

​ 过滤器和监听器是SpringMVC封装的Servlet的监听和过滤器

​

**Spring使用了哪些设计模式**

​ 那么就先举几个比较常用的spring的设计模式吧（不是全部！）

​ 1、简单工厂模式（Sample Factory）

​ 这个体现在BeanFactory上使用了工厂模式

​ 2、代理模式（Proxy Pattern）

​ 代理模式主要体现在AOP的动态代理上

​ 3、单例模式（Single Pattern）

​ 单例模式就是在spring的容器对象中体现

​ 4、模版方法模式（Template Method）

​ 模板方法就是通过反射拿到Bean的类模板

​ 5、观察者模式（Oberver Pattern）

​ 这个体现在Spring的监听器中

**Spring的IOC有什么优势**

​ 控制反转，将一个对象，通过反射，再使用单例将对象加载到Spring的容器中，大大减少频繁new对象，并且频繁new对象所造成的内存其实是非常浪费的，例如每次一个人来访问，就会new一个实现类对象，然后调用其中的方法，这样多个人调用他就会产生多个内存对象，造成资源浪费，而且他们所调用的都是一样的方法，这样就会产生耦合，在Spring中是不被允许的，因为开发的代码质量要注重高内聚低耦合的思想，所以使用IOC来管理对象一是节约了内存空间。二是降低了代码的耦合度，三是实现了控制的反转，以前是需要一个对象就new一个，但是我们将他放到容器中要用从容器中拿出来直接用即可

**Spring的bean的生命周期**

​ 1、instantiate bean对象的实例化

​ 2、populate properties封装属性

​ 3、如果Bean实现BeanNameWare执行setBeanName

​ 4、如果Bean实现BeanFactoryAware 或者 ApplicationContextAware 设置工厂 setBeanFactory 或者上下文对象 setApplicationContext

​ 5、如果存在类实现 BeanPostProcessor（后处理Bean） ，执行postProcessBeforeInitialization，BeanPostProcessor接口提供钩子函数，用来动态扩展修改Bean。(程序自动调用后处理Bean)

​ 6、如果Bean实现InitializingBean 执行 afterPropertiesSet

​ 7、调用<bean init-method="init">指定初始化方法 init

​ 8、如果存在类实现 BeanPostProcessor（处理Bean） ，执行postProcessAfterInitialization

​ 9、执行业务处理

​ 10、如果Bean实现 DisposableBean 执行 destroy

​ 11、调用<bean destroy-method="customerDestroy"> 指定销毁方法 customerDestroy

**Spring的BeanFacoty和ApplicationContext的区别**

​ Spring核心工厂是BeanFactory ,BeanFactory采取延迟加载，第一次getBean时才会初始化Bean,

​ ApplicationContext是对BeanFactory扩展，ApplicationContext是会在加载配置文件时初始化Bean。它可以进行国际化处理、事件传递和bean自动装配以及各种不同应用层的Context实现

​

**Spring是什么呢？**

​ 是一个IOC，APO框架，是一个生态体系，是一个一站式Java框架，也是一个容器

**SpringMVC的运行流程**

​ 1、用户发送请求至前端控制器DispatcherServlet

​ 2、DispatcherServlet收到请求调用HandlerMapping处理器映射器。

​ 3、处理器映射器找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

​ 4、DispatcherServlet调用HandlerAdapter处理器适配器

​ 5、HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)

​ 6、Controller执行完成返回ModelAndView

​ 7、HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet

​ 8、DispatcherServlet将ModelAndView传给ViewReslover视图解析器

​ 9、ViewReslover解析后返回具体View

​ 10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）

​ 11、DispatcherServlet响应用户

**SpringMVC的核心**

​ DispatcherServlet 核心控制器

​ HandlerMapping 处理器映射器

​ HandlerAdapter 处理器适配器

​ ModelAndView 数据视图模型对象

​ ViewReslover 视图解析器

**Spring的IOC和AOP以及DI注入**

**IOC**

​ 核心：控制反转

​ “控制反转”，不是什么技术，而是一种设计思想 Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。

​

**AOP**

​ 核心：面向切面

​ 面向切面也是一种设计思想，他是为了解决代码混乱，代码冗余

​ AOP的关键点：

​ 关注点：

​ 切面(Aspect)：是一个类，里面定义了通知与切点。

​ 切点(PointCut)：表达式。就是告诉程序要在执行哪些核心业务的时候，执行非核心的业务。

​ 通知（advice）:

五种通知方式：

前置通知 [Before advice]： 在连接点前面执行，前置通知不会影响连接点的执行，除非此处抛出异常。

​

正常返回通知[After returning advice]：在连接点正常执行完成后执行，如果连接点抛出异常，则不会执行。

​

异常返回通知[After throwing advice]：在连接点抛出异常后执行。

返回通知[After (finally) advice]：在连接点执行完成后执行，不管是正常执行完成，还是抛出异常，都会执行返回通知中的内容。

环绕通知[Around advice]：环绕通知围绕在连接点前后，比如一个方法调用的前后。这是最强大的通知类型，能在方法调用前后自定义一些操作。环绕通知还需要负责决定是继续处理join point(调用ProceedingJoinPoint的proceed方法)还是中断执行。

**DI**

​ 依赖注入(DI),DI是IOC的实现方式,依赖注入(DI)和控制反转(IOC)是从不同的角度的描述的同一件事情，就是指通过引入IOC容器，利用依赖关系注入的方式，实现对象之间的解耦。 所谓依赖注入，是指程序运行过程中，如果需要调用另一个对象协助时，无须在代码中创建被调用者，而是依赖于外部的注入。Spring的依赖注入对调用者和被调用者几乎没有任何要求，完全支持对POJO之间依赖关系的管理

​

**Mybatis是什么呢？**

​ 其实，它底层就是一个JDBC封装的组件 ,MyBatis的前身是Ibatis ,也是一个半自动的ORM关系映射框架

**Mybatis和Hibernate的区别是什么？**

​ Mybatis是半自动ORM映射框架，而Hibernate是一个全自动映射框架，Hibernate映射了数据库和实体类的关系，而Mybatis映射了实体类和SQL语句的关系

**Mybatis核心组件有哪些**

​ SqlSessionFactoryBuilder（构造器）

​

​ SqlSessionFactory（工厂接口）

​ SqlSession（会话接口）

​  作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能

​ SQL Mapper（映射器）

​

​ Executor（执行器）

​ MyBatis执行器，是MyBatis 调度的核心，负责SQL语句的生成和查询缓存的维护

​ StatementHandler（封装了JDBC Statement操作 ）

​ 封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数、将Statement结果集转换成List集合。

​ ParameterHandler（参数转换器）

​   负责对用户传递的参数转换成JDBC Statement 所需要的参数，

​ ResultSetHandler（结果返回）

​ 负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；

​ TypeHandler（转换实体类与数据库的映射）

​   负责java数据类型和jdbc数据类型之间的映射和转换

​ MappedStatement（维护增删改查节点）

​ MappedStatement维护了一条<select|update|delete|insert>节点的封装，

​ SqlSource（sql源生成器）

​  负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回

​ BoundSql（生成的sql）

​ 表示动态生成的SQL语句以及相应的参数信息

​ Configuration（Mybatis的所有配置信息）

​  MyBatis所有的配置信息都维持在Configuration对象之中。

**Mybatis的运行流程**

​ 首先SqlSessionFactoryBuilder，根据传入的数据流生成Configuration对象，

​ 然后根据Configuration对象创建默认的SqlSessionFactory实例

​ 然后从SqlSessionFactory中获取SqlSession

​ 然后根据SqlSession去执行Executor执行器

​ 在执行的过程中StatementHandler封装参数以及JDBC原生操作

​ ParameterHandler是在StatementHandler封装的参数

​ TypeHandler负责转换实体类和数据库字段的关系

**微服务相关**

**Ribbon的默认负载均衡算法有几种**

RoundRobinRule     轮询

RandomRule         随机  
​  
AvailabilityFilteringRule 会先过滤多次访问故障而处于断路器跳闸状态的服务  
​  
WeightedResponseTimeRule 根据平均响应时间计算所有的服务权重，响应时间越快服务权重越大就越容易被选中，刚启动时信息统计不足，会先使用轮询，等到响应数据有了之后在进行响应时间进行均衡负载

RetryRule 先按照轮询的方式，如果获取到服务失败等就会在指定的时间内进行重试，然后获取下一个可用的服务  
​  
BastAvailableRule 会先过滤掉由于多次访问故障而处于断路器跳闸的服务，然后选择一个并发量最小的服务  
​  
ZoneAvoidanceRule     默认规则，符合判断服务所在的区域的性能和服务的可用性选择服务器

**Eureka和Zookeeper的区别**

著名的CAP理论指出，一个分布式系统不可能同时满足C(一致性)、A(可用性)和P(分区容错性) 。

Zookeeper保证CP

​ 所以Zookeeper的一致性比较强，但是他的可用性并不高，体现在如果一个Zookeeper集群挂掉了父节点，那么他会重新进行选举，而在这个选举的过程中他是无法提供服务的

Eureka保证AP

​

​ 在Eureka的集群中，他保证了可用性，也就是当一个注册中心挂掉了之后他会带着服务去另一个注册中心，这个并不影响他们的可用性，只要还有一口气他的服务就能提供，除非所有的注册中心都挂掉了

**Eureka的运行流程和实现原理**

Register（服务注册） ----》 Renew（服务续约 ----》 Cancel（服务下线）

服务注册一般是发生在服务启动的时候，后面如果服务自身检测认为 Down，也会来更新服务状态的。

服务续约，主要是用来告诉 Eureka Server Service Provider 还活着，避免服务被剔除掉。

服务下线一般在 Service Provider shut down 的时候调用，用来把自身的服务从 Eureka Server 中删除，以防客户端调用不存在的服务。

小结：服务的注册表示服务发现，将自己的信息注册到Eureka上面，并且不定期的发送检测机制（心跳检测），如果有相应就进行服务续约，如果发现一个服务响应超时或者说访问不了，那么他就会将服务剔除掉，这就是服务下线

实现原理：

​ 其实主要还是使用Servlet实现的，它使用 Jersey ，框架实现自身的 RESTful HTTP接口，peer之间的同步与服务的注册全部通过 HTTP 协议实现 ，他的心跳机制等都是通过定时任务去进行实现的

**Eureka的核心**

**心跳检测机制**

什么是心跳检测机制？

​ 就是当你的服务再运行的过程中他会时不时地去检测你当前的服务是否健康如果响应超时或者访问不到它就会进行服务剔除，默认是90秒，这个是可以修改的，但是如果Eureka和服务之间出现了网络故障本来就是由于自己的网络状态所引起的那么他就会进入到自我保护状态

**自我保护机制**

​ 自我保护机制的工作机制是如果在15分钟内超过85%的客户端节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，Eureka Server自动进入自我保护机制，此时会出现以下几种情况：

1、Eureka Server不再从注册列表中移除因为长时间没收到心跳而应该过期的服务。

2、Eureka Server仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上，保证当前节点依然可用。

3、当网络稳定时，当前Eureka Server新的注册信息会被同步到其它节点中。

这里我们可以看到哪怕他是出现了故障的时候他都会保证当前节点的可用性。以及自己的容错性，所以在个Zookeeper的对比中他做到了AP(可用性和分区容错性)

**Eureka服务端配置**

心跳检测，毫秒，改为10秒，如果10秒没有收到服务的信息，那么就会将服务剔除掉，默认60\*1000,60秒  
eureka.server.eviction-interval-timer-in-ms=10000  
​  
是否开启自我保护，默认true  
eureka.server.enable-self-preservation=true  
​  
是否将自己注册到注册中心上以供其他服务发现，默认true  
eureka.client.register-with-eureka=false  
​  
是否从注册中心上，获取注册表的信息  
eureka.client.fetch-registry=false  
​

**Eureka客户端配置**

#eureka客户端需要多长时间发送心跳给eureka服务器，表明他仍然或者，默认30秒  
eureka.instance.lease-renewal-interval-in-seconds=30  
​  
#eureka服务器在接受到实力的最后一次发出的心跳后，需要等待多久才可以将此实例删除，默认90秒  
eureka.instance.lease-expiration-duration-in-seconds=30  
​  
#表示eureka client间隔多久去拉取服务器注册信息,默认为30秒  
eureka.client.registry-fetch-interval-seconds=18  
​  
#表示eureka的注册中心的地址  
eureka.client.serviceUrl.defaultZone=http://127.0.0.1:8081/eureka/

**Feign客户端负载均衡以及熔断**

**Feign客户端的实现原理**

​ Feign 的英文表意为“假装，伪装，变形”， 是一个http请求调用的轻量级框架，可以以Java接口注解的方式调用Http请求，而不用像Java中通过封装HTTP请求报文的方式直接调用。 他是基于HttpClient进行调用的，

**项目方面**

**介绍一下自己**

​ 首先简短的介绍自己：

​ 你好我叫XXX，今年XX岁，面试的职位是XXX，以前在XXXX公司工作过X年，（根据实际情况，说学校以及）

**讲讲项目经历**

​ 结合实际项目或者虚拟项目，划分模块2-3个，每一个模块所应用到的核心技术（如：Redis，Elasticsearch，MongoDB，FastDFS等等），然后是就是项目整体架构（例如：垂直结构（SSM），分布式架构（Dubbo），还有微服务架构（SpringCloud）），然后就是核心的功能，例如搜索，如何去进行搜索，有哪些比较有特点的代码，在做这个搜索模块中所遇到的问题，还有优化和容灾是如何进行，如果让你来设计这个搜索你会如何设计这个模块

**你们项目中微服务是怎么划分的，划分粒度怎么确定？**

​ 答：这个是根据整体项目的架构大小来确认的，如果功能特别多特别复杂以及业务上面的区分来进行划分，例如：汽车销售平台中的和汽车相关的，在项目比较小的情况下并且业务划分不太多，比如汽车卖出去之后和汽车销售的统计，还有一些不太多的小功能（例如：规格参数和海报），那么这个就完全可以写到一个模块中，就叫做汽车模块，但是另一种场景，他需要汽车表的基本功能，还有统计，然后他还有一个需求，是保险方便的，车险需要购置保险，并且如果还有合同等等，这样的话就需要拆分成两个模块了，一个汽车模块。一个售后附加模块，更多的是根据业务来进行划分的

**那在实践微服务架构中，有遇到什么问题么？**

**Docker：**

​ 1、在使用Docker部署的时候一些特定不开放端口的服务，我们重新启动了Docker，这样会破坏Centos7的firewalld防火墙规则，这样这个端口还是会被访问的，例如我们先启动防火墙然后启动docker进行端口映射这样会造成规则破坏，导致端口号无法被防火墙拦截，造成安全隐患那么怎么解决呢（Centos7中会出现），原因是因为Docker在使用端口

​ 解决博客

​ <https://myforever.cn/blog/538872310925361152/550244603433844736/551125169477779456>

**数据库：**

​

**项目：**

​ 1、Eureka的心跳机制问题，在项目中关闭了Eureka的心跳机制

​

**服务器：**

​

**前后端交互：**

​ 1、分布式id前端精度丢失问题，在数据库中使用bigint数据类型，在项目实体类中使用Long类型，并且进行分布式id的生成，在前后端的数据传递过程中，由于前段long类型只能保存16位，而分布式id至少有19位，从而造成精度的丢失，而引起的数据不一致性

​ 解决方案：

​ 将实体类中的分布式字段id改成String类型，并且将分布式id生成改为idwork-str的String类型，这样就能解决了

**你们在关于微服务间数据一致性问题，是如何解决的？**

**数据库方面：**

​ 在数据库主从复制的时候，所引起的主从延迟问题造成的数据库不一致，那么这个如何解决呢？

​

**自己做过哪些调优？JVM调优、数据库调优都行**

​ 数据库调优，主要是体现在数据库的索引优化上，对一些比较查询频繁的表和数据量比较大的表进行索引的优化，通过结合代码进行索引的创建，并且通过Mysql性能分析进行检查代码性能

​ **方案一：忽略**

​ 任何脱离业务的架构设计都是耍流氓，绝大部分业务，例如：百度搜索，淘宝订单，QQ消息，58帖子都允许短时间不一致。如果业务能接受，最推崇此法。别把系统架构搞得太复杂。

​ **方案二：选择性主读**

​ 将哪个库，哪个表，哪个主键三个信息拼装一个key设置到cache里，这条记录的超时时间，设置为“主从同步时延” ，画外音：key的格式为“db:table:PK”，假设主从延时为1s，这个key的cache超时时间也为1s。 这是要读哪个库，哪个表，哪个主键的数据呢，也将这三个信息拼装一个key，到cache里去查询， 在cache里记录哪些记录发生过写请求，来路由读主还是读从

**平时会用到哪些数据结构？**

​ 线型：

​ 数组

​ 队列

​ 栈

​ 树型：

​ B+树

​ 红黑树

**RPC和HTTP的关系是什么？**

​ RPC（即Remote Procedure Call，远程过程调用）

​ HTTP（HyperText Transfer Protocol，超文本传输协议）

​ 他们最本质的区别，就是RPC主要工作在TCP协议之上，而HTTP服务主要是工作在HTTP协议之上，我们都知道HTTP协议是在传输层协议TCP之上的，所以效率来 说RPC肯定是要比HTTP效率更高得

**谈一谈你对微服务架构的理解**

​ 主要是因为SOA（流式计算架构），微服务架构我们可以看成流式计算，并且将相应的功能模块拆分成一个个细微的服务，服务与服务之间可以相互调用，并且微服务可以很好的实现扩容缩容，Cloud也提供了一系列的技术解决方案，能够协调这些服务进行项目的构建以及运维

SpringCloudAlibaba

**你用过哪些RPC框架，讲讲他们优缺点**

​ Dubbo，只支持JAVA语言，不能支持其他语言的拓展，Dubbo只是实现了服务治理，其他组件需要另外整合以实现对应的功能，比如，分布式配置：可以使用淘宝的diamond、百度的disconf来实现分布式配置管理。服务跟踪：可以使用京东开源的Hydra，批量任务：可以使用当当开源的Elastic-Job，Dubbo的RPC来实现服务间调用的一些痛点，服务提供方与调用方接口依赖方式太强：调用方对提供方的抽象接口存在强依赖关系，需要严格的管理版本依赖，才不会出现服务方与调用方的不一致导致应用无法编译成功等一系列问题 ，服务对平台敏感，难以简单复用：通常我们在提供对外服务时，都会以REST的方式提供出去，这样可以实现跨平台的特点。 在Dubbo中我们要提供REST接口时，不得不实现一层代理，用来将RPC接口转换成REST接口进行对外发布。

​ 总结：

​ 只支持Java不易于拓展

​ 组件少，整合繁琐，对项目的构建来说难度会比较大

​ 对接口的依赖高，并且需要严格的管理版本依赖

​ 服务对平台敏感，难以简单复用：通常我们在提供对外服务时通常以REST方式提供

**用过docker么，对容器了解多少**

​ 简单的了解和使用过，docker是一个容器虚拟化技术，他的三个核心是，镜像，容器，仓库，从仓库中下载镜像再通过镜像虚拟化容器到宿主机中，解决了开发环境的不一致的问题，并且他轻巧方便，并且有极高的稳定性

**如何定位JAVA生产卡顿或卡死问题，如何定位解决问题**

​ 首先在Linux当中定位问题

​ 首先查询是不是Java的问题导致的占用内存过高

top

​ 那么我们查看例如4087（他只占用了0.7的CPU，但是生产的时候定位的时候肯定不是，模拟环境）

​ 我们就定位到了4087他占用CPU就是这个Java进程，然后我们分析这个进程的线程

ps -mp 4087 -o THREAD,tid,time

​ 我们可以看到线程的tid，我们找到例如4185这个线程，那么在内存中使用的是16进制，所以我们需要转换为16，把4185用计算器或者其他的转换为16进制，那么4185转换为16进制就是1059 ，但是在分析的时候需要使用字母必须为小写

jstack 4087 | grep 1059 -A60

然后通过进程id再跟踪线程tid进行分析

​

这样就能找到对应的线程以及对应的类，以及错误行数了，然后进行代码的修改或者是优化