# Spring

spring原理

Bean 刚刚初始化的时候做一些操作

Spring Bean 的生命周期是什么样子

AOP 里面的 Proxy 都是基于什么原理实现，有什么优缺点。

Spring AOP的实现原理和场景？

Spring bean的作用域和生命周期；

Spring IOC是什么？优点是什么？

Spring如何实现事务

# SpringMVC

## springMVC和struts2的区别有哪些?

（1）springmvc的入口是一个servlet即前端控制器（DispatchServlet），而struts2入口是一个filter过虑器（StrutsPrepareAndExecuteFilter）。

（2）springmvc是基于方法开发(一个url对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例。

（3）Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，springmvc通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。

## Spring MVC的异常处理 ？

答：可以将异常抛给Spring框架，由Spring框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

## SpringMVC的控制器是不是单例模式,如果是,有什么问题,怎么解决？

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

## SpringMvc里面拦截器是怎么写的

有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在SpringMvc的配置文件中配置拦截器即可：

## SpringMVC的流程？

（1）用户发送请求至前端控制器DispatcherServlet；

（2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handle；

（3）处理器映射器根据请求url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 具体处理器(Handler，也叫后端控制器)；

（6）Handler执行完成返回ModelAndView；

（7）HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet；

（8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析；

（9）ViewResolver解析后返回具体View；

（10）DispatcherServlet对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。

# SpringBoot

**什么是 Spring Boot？**

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，开发者能快速上手。

**Spring Boot解决哪些问题？**

Spring Boot使编码，配置，部署，监控变简单。

**Spring Boot 优点**

独立运行，简化配置，自动配置，无代码生成和XML配置，应用监控，上手容易

**Spring Boot 的核心配置文件有哪几个？它们的区别是什么？**

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 主要用于 Spring Boot 项目的自动化配置。

bootstrap 配置文件有以下几个应用场景。

使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；

一些固定的不能被覆盖的属性；

一些加密/解密的场景；

**Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？**

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。

**开启 Spring Boot 特性有哪几种方式？**

1）继承spring-boot-starter-parent项目

2）导入spring-boot-dependencies项目依赖

**Spring Boot 自动配置原理是什么？**

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

**你如何理解 Spring Boot 中的 Starters？**

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成 Spring 及其他技术，而不需要到处找示例代码和依赖包。如你想使用 Spring JPA 访问数据库，只要加入 spring-boot-starter-data-jpa 启动器依赖就能使用了。

**如何在 Spring Boot 启动的时候运行一些特定的代码？**

可以实现接口 ApplicationRunner 或者 CommandLineRunner，这两个接口实现方式一样，它们都只提供了一个 run 方法

**Spring Boot 有哪几种读取配置的方式？**

Spring Boot 可以通过 @PropertySource,@Value,@Environment, @ConfigurationProperties 来绑定变量

**SpringBoot 实现热部署有哪几种方式？**

主要有两种方式：Spring Loaded，Spring-boot-devtools

**你如何理解 Spring Boot 配置加载顺序？**

在 Spring Boot 里面，可以使用以下几种方式来加载配置。

1）properties文件；

2）YAML文件；

3）系统环境变量；

4）命令行参数；

## SpringBoot启动流程

当执行SpringApplication类的静态run方法时

1、创建SpringApplication实例：

将传入的启动类"com.example.ms.DemoApplication"放入Set集合中

判断是否为Web环境：存在（javax.servlet.Servlet && org.springframework.web.context.ConfigurableWebApplicationContext ）类则是

创建并初始化ApplicationInitializer列表 （spring.factories）

创建并初始化ApplicationListener列表  （spring.factories）

初始化主类mainApplicatioClass    (DemoApplication)

2、启动run方法

创建计时器StopWatch

配置awt系统属性

获取SpringApplicationRunListeners

启动SpringApplicationRunListeners

创建ApplicationArguments

创建并初始化ConfigurableEnvironment

打印Banner

创建ConfigurableApplicationContext

准备ConfigurableApplicationContext

刷新ConfigurableApplicationContext

容器刷新后动作

SpringApplicationRunListeners发布finish事件

计时器停止计时

Spring Boot、Spring MVC 和 Spring 有什么区别？

Spring 是一个“引擎”;

Spring MVC 是基于Spring的一个 MVC 框架;

Spring Boot 是基于Spring4的条件注册的一套快速开发整合包。

# SpringCloud

## SpringCloud生态

Spring Cloud Config：配置管理工具包，让你可以把配置放到远程服务器，集中化管理集群配置，目前支持本地存储、Git以及Subversion。

Spring Cloud Bus：事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。

Eureka：云端服务发现，一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。

Hystrix：熔断器，容错管理工具，旨在通过熔断机制控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。

Ribbon：提供云端负载均衡，有多种负载均衡策略可供选择，可配合服务发现和断路器使用。

Zuul：Zuul 是在云平台上提供动态路由,监控,弹性,安全等边缘服务的框架。Zuul 相当于是设备和 Netflix 流应用的 Web 网站后端所有请求的前门。

Feign：Feign是一种声明式、模板化的HTTP客户端。

Spring Cloud Task：提供云端计划任务管理、任务调度。

Spring Cloud Security：基于spring security的安全工具包，为你的应用程序添加安全控制。

Archaius：配置管理API，包含一系列配置管理API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。

Consul：封装了Consul操作，consul是一个服务发现与配置工具，与Docker容器可以无缝集成。

Spring Cloud for Cloud Foundry：通过Oauth2协议绑定服务到CloudFoundry，CloudFoundry是VMware推出的开源PaaS云平台。

Spring Cloud Sleuth：日志收集工具包，封装了Dapper和log-based追踪以及Zipkin和HTrace操作，为SpringCloud应用实现了一种分布式追踪解决方案。

Spring Cloud Data Flow：大数据操作工具，作为Spring XD的替代产品，它是一个混合计算模型，结合了流数据与批量数据的处理方式。

Spring Cloud Zookeeper：操作Zookeeper的工具包，用于使用zookeeper方式的服务发现和配置管理。

Spring Cloud Stream：数据流操作开发包，封装了与Redis,Rabbit、Kafka等发送接收消息。

Spring Cloud CLI：基于 Spring Boot CLI，可以让你以命令行方式快速建立云组件。

Turbine：Turbine是聚合服务器发送事件流数据的一个工具，用来监控集群下hystrix的metrics情况。

Spring Cloud Connectors：便于云端应用程序在各种PaaS平台连接到后端，如：数据库和消息代理服务。

Spring Cloud Cluster：提供Leadership选举，如：Zookeeper, Redis, Hazelcast, Consul等常见状态模式的抽象和实现。

Spring Cloud Starters：Spring Boot式的启动项目，为Spring Cloud提供开箱即用的依赖管理。

## Spring Cloud微服务如何设计异常处理机制

面向外部提供的服务接口，会通过服务网关（如使用Zuul提供的apiGateway）面向公网提供服务，如给App客户端提供的用户登陆、注册等服务接口。在编写面向外部的服务接口时，服务端所有的异常处理我们都要进行相应地捕获，并在controller层映射成相应地错误码和错误信息，返回给调用方。

而面向内部的服务接口，则是在进行微服务拆分后由于各个微服务系统的边界划定问题所导致的功能逻辑分散。在基于Spring Cloud的微服务体系中，微服务提供方会提供相应的客户端SDK代码，而客户端SDK代码则是通过FeignClient的方式进行服务调用，Feign内部会集成像Ribbon和Hystrix这样的框架来实现客户端服务调用的负载均衡和服务熔断功能。

**最佳实践设计**

首先，无论是内部还是外部的微服务，在服务端我们都应该设计一个全局异常处理类，用来统一封装系统在抛出异常时面向调用方的返回信息。而实现这样一个机制，我们可以利用Spring提供的注解@ControllerAdvice来实现异常的全局拦截和统一处理功能。从最佳实践上考虑，我们一般会为内部和外部接口分别设计一个统一面向调用方的异常对象。

FeignClient为我们提供了异常解码机制

## 原生Spring-Cloud-Config的配置中心的缺点

**Git的权限控制是个坑**：Git的权限管理是说控制用户能不能Push或者Delete分支，或者能不能Push代码，而不是能不能访问某个目录的文件。

**粒度问题**：你每次对一条配置发生crud的操作，其带来的影响是整个文件发生变动。当时我们最理想的存储介质就是数据库。

**Spring-Cloud-Config的刷新机制是个坑**：因为一个配置中心应该要能够做到，配置发生改动的时候，项目能够自动感知，自动更新配置才对。在Spring-Cloud-Config中，这套机制是借助一些代码仓库（SVN、Github等）提供的Webhook机制加上Spring-Cloud-Bus来实现的。

那么问题又来了！配置数据放在数据库中，数据库里没有Webhook这种东西啊，怎么做到实时刷新？

**长轮询，短轮询**

在页面上要实时显示后台的库存数量！比如库存减少了，用户不需要进行刷新，页面上的数字自己会变化。

如果采取短轮询就是在客户端(js)中不断访问后台，后台接到请求马上返回最新的库存数，然后刷新到这个页面当中。

长轮询是客户端(js)依然是不断的去请求。但是呢，服务端不是马上返回。而是等待库存数量变化了再返回。

怎么实现：我们在项目中采用Spring的DeferredResult来实现。

## 如何有效快速的监听出配置表的数据发生了变动？

因为我们用的是mysql。这里有一个Mysql的自定义函数叫mysql-udf-http。具有http\_get()、http\_post()、http\_put()、http\_delete()四个函数，可以在MySQL数据库中利用HTTP协议进行REST相关操作。

然后再和mysql的触发器结合起来用，可以实现在配置表发生变动的时候，主动通知我们的配置中心服务端。让服务端明白配置发生了变动！

# MySQL

## ？事务介绍，分布式事物的理解，常见的解决方案有哪些，什么事两阶段提交、三阶段提交？

## ？MySQL记录binlog的方式主要包括三种模式？每种模式的优缺点是什么？

？JDBC如何实现事务、嵌套事务实现、分布式事务实现

## 谈谈我经历的三个公司的数据库操作

你们数据库的高可用架构是怎么样的？

卓望公司：使用oracle为主库，再使用工具将数据同步到mysql数据库中，运维会实时监控数据库是否宕机或其它异常。

顺丰科技：将订单号作为分库分表的依据，将该订单相关的数据都放到一个库中，然后基础数据冗余，有三个表。缺点是做统计的时候比较麻烦，做不了实时统计。冗余数据只写一个表，然后用etl同步工具同步。

瑞宝麟：贷前系统采用单一的oracle数据库，报表系统采用分表策略，每天一个表

自我感觉在数据库这一方面比较薄弱。听说最近有款分库分表的中间件叫ShardingSphere，还没来得及去看。

数据库主从复制的依据是：

Binlog日志

## 数据库集群如何保证其每个数据库的数据一致性?

1.半同步复制：简单的说就是: 主库发生增删改操作的时候,会等从库及时复制了并且通知了主库, 才会把这个操作叫做成功。专业的讲，半同步复制，是等待其中一个从库也接收到Binlog事务并成功写入Relay Log之后，才返回Commit操作成功给客户端。优点：保证数据一致性，优点：保证数据一致性。

2.数据库中间件：所有的读写请求都走中间件，然后写的请求路由到主库，读的请求路由到从库当有一个写请求过来时候，生成一个key A ，马上路由写到主库，然后立马有一个读请求过来。 如果时间是在1s内的，就对应的key继续路由到主库。如果在1s以后的，就路由到从库。说白了，中间件就是给个同步时间，给你同步，在同步时间内，所有的请求都落在主库

3 缓存记录写key法:将某个库上的某个key要发生写操作，记录在cache里，并设置“经验主从同步时间”的cache超时时间，例如1s。然后修改数据库。

用mysql过程中，有遇到什么问题么？

因为字段超长报错这些就不说了吧，比较棘手又常见的是慢查询问题

## 乐观锁的业务场景及实现方式

一般在并发比较小的时候使用乐观锁。

相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本。

实现数据版本有两种方式，第一种是使用版本号，第二种是使用时间戳。

## SQL解析，SQL执行过程

客户端把语句发给服务器端执行，服务器会先在数据库的高速缓存中去查找,是否存在相同语句的执行计划。如果在

数据高速缓存中,则服务器进程就会直接执行这个 SQL 语句,省去后续的工作。如果没有就会走下一步，进行语句合法性检查(data dict cache)，语言含义检查(data dict cache)，当语法、语义都正确后,系统就会对我们需要查询的对象加锁。并且检查所连接的用户是否有这个数据访问的权限。当语句与语法都没有问题,权限也匹配的话,服务器进程还是不会直接对数据库文件进行查询。服务器进程会根据一定的规则,对这条语句进行优化。然后再是语义执行，提取数据并返回。

## 说一下关系型数据库和非关系型数据库的区别

非关系型数据库的优势：

性能：NOSQL是基于键值对的，可以想象成表中的主键和值的对应关系，而且不需要经过SQL层的解析，所以性能非常高

可扩展性：同样也是因为基于键值对，数据之间没有耦合性，所以非常容易水平扩展。

使用场景：日志、埋点、论坛、博客等

关系型数据库的优势：

复杂查询：可以用SQL语句方便的在一个表以及多个表之间做非常复杂的数据查询

事务支持:使得对于安全性能很高的数据访问要求得以实现。

使用场景：所有有逻辑关系的数据存储

## 说一下Innodb和MySIAM的区别

MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持。MyISAM类型的表强调的是性能，其执行数度比InnoDB类型更快，但是不提供事务支持，而InnoDB提供事务支持以及外部键等高级数据库功能。

InnoDB不支持FULLTEXT类型的索引

InnoDB 中不保存表的具体行数，也就是说，执行select count(\*) from table时，InnoDB要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count(\*)语句包含 where条件时，两种表的操作是一样的。

对于AUTO\_INCREMENT类型的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中，可以和其他字段一起建立联合索引。

DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除。

LOAD TABLE FROM MASTER操作对InnoDB是不起作用的，解决方法是首先把InnoDB表改成MyISAM表，导入数据后再改成InnoDB表，但是对于使用的额外的InnoDB特性(例如外键)的表不适用。

# Mybatis

## mybatis 的 dao 接口跟 xml 文件里面的sql 是如何建立关系的？

Mybatis在初始化SqlSessionFactoryBean的时候，找到mapperLocations路径去解析里面所有的XML文件。Mybatis会把每个SQL标签封装成SqlSource对象（根据SQL语句的不同，又分为动态SQL和静态SQL。其中，静态SQL包含一段String类型的sql语句；而动态SQL则是由一个个SqlNode组成）。然后会将SqlSource与全限定类名+方法名组成的ID封装再一个MappedStatement对象中，并将它缓存到Configuration#mappedStatements中，每一个SQL标签就对应一个MappedStatement对象。

当使用@Autowired注入这个Dao接口的时候，spring会通过JDK动态代理，返回了一个Dao接口的代理对象，这个代理对象的处理器是MapperProxy对象。调用Dao接口方法的时候，实际调用到代理对象的invoke方法，在这里，实际上调用的就是SqlSession里面的东西了。SqlSession通过statement全限定类型+方法名拿到MappedStatement 对象，然后

MappedStatement调用执行器Executor去具体SQL并返回。

## 对于有实现的dao接口，mapper还会用代理么？

答案是肯定，只要你配置了MapperScan，它就会去扫描，然后生成代理。但是，如果你的dao接口有实现类，并且这个实现类也是一个Spring Bean，那就要看你在Autowired的时候，去注入哪一个了。如果没有显示注入哪个实现类，这种情况会在启动时就报错。

## 如果有两个XML文件和这个Dao建立关系，岂不是冲突了？

不管有几个XML和Dao建立关系，只要保证namespace+id唯一即可。

# Redis

## Redis 底层数据结构有一下数据类型：

**简单动态字符串（simple dynamic string）SDS；链表；字典；跳跃表；整数集合；压缩列表；对象**

**字典**，又称为符号表（symbol table）、关联数组（associative array）或映射（map），是一种用于保存键值对的抽象数据结构。在字典中，一个键（key）可以和一个值（value）进行关联，字典中的每个键都是独一无二的。在C语言中，并没有这种数据结构，但是Redis 中构建了自己的字典实现。比如SET msg "hello world"创建这样的键值对（“msg”，“hello world”）在数据库中就是以字典的形式存储。Redis 字典所使用的哈希表由 dict.h/dictht 结构定义.Redis 中采用了连地址法（separate chaining）来解决键冲突。

**跳跃表**：跳跃表（skiplist）是一种有序数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。

**整数集合**：整数集合是集合建的底层实现之一，当一个集合中只包含整数，且这个集合中的元素数量不多时，redis就会使用整数集合intset作为集合的底层实现。我们可以这样理解整数集合，他其实就是一个特殊的集合，里面存储的数据只能够是整数，并且数据量不能过大。

**压缩列表**：压缩列表是列表键和哈希键的底层实现之一。当一个列表键只把汗少量列表项，并且每个列表项要么就是小整数，要么就是长度比较短的字符串，那么Redis 就会使用压缩列表来做列表键的底层实现。

## 知道动态字符串sds的优缺点么？

**SDS与C字符串区别**

**C字符串：**

**缺点：**

获取字符串长度的复杂度为O(N)

API不安全，可能会造成缓存区溢出（对于一个字符串A后面拼接字符串B时，需要考虑A字符串的容量问题）

修改字符串长度N次必然需要执行N次内存重分配（即无论是缩小还是扩大C字符串的大小都需要进行内存分配）

只能保存文本数据（即C字符串是以'\0'结尾的，如果想保存'\0'这个特殊字符，则会导致数组提前被截取了）

**优点：**

可以使用所有<String.h>库中的函数

**SDS**

**优点：**

获取字符串长度的复杂度为O(1)

API是安全的，不会造成缓存溢出问题

修改字符串长度N次最多需要执行N次内存重新分配（由于SDS结构中有free字段的存在，因此缩小时，会把没有的char算到free中（惰性空间释放），而放大（空间预分配）时如果len长度<1M，则扩大后的len=原有len\*2，如果len长度>=1M，则扩大后的len=原有len+1M）

可以保存文本或者二进制数据（因为SDS获取字符串时是根据len长度算的，而不是根据'\0'结尾之前的字符串算）

可以使用一部分<String.h>库中函数（不需要重新实现）

**缺点：**

很多函数以值的形式返回新字符串，由于有时SDS要求创建一个占用更多空间的新字符串，所以大多数SDS的API调用像这样：s = sdscat(s,"Some more data");你可以看到s被用来作为sdscat的输入，但也被设为SDS API调用返回的值，因为我们不知道此调用是否会改变了我们传递的SDS字符串，还是会重新分配一个新的字符串。忘记将sdscat或者类似函数的返回值赋回到存有SDS字符串的变量的话，就会引起bug。

如果一个SDS字符串在你的程序中多个地方共享，当你修改字符串的时候，你必须修改所有的引用。但是，大多数时候，当你需要共享SDS字符串时，将字符串封装成一个结构体，并使用一个引用计数会更好，否则很容易导致内存泄露。

## redis的单线程特性有什么优缺点？

**Redis快的主要原因是**：

完全基于内存

数据结构简单，对数据操作也简单

使用多路 I/O 复用模型：这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。

**单进程单线程好处**

代码更清晰，处理逻辑更简单

不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

不存在多进程或者多线程导致的切换而消耗CPU

**单进程单线程弊端**

无法发挥多核CPU性能，不过可以通过在单机开多个Redis实例来完善；

不适合大数据传输

**其他一些优秀的开源软件采用的模型**

多进程单线程模型：Nginx

单进程多线程模型：Memcached

## Redis分布式锁操作的原子性，Redis内部是如何实现的？

因为redis是单线程的。

## ？Redis跳跃表的问题？

## ？redis里面的哈希表？

happen-before原则（前一个操作的执行结果必须对后一个操作可见）

# Kafka

## kafka 的架构，包含了哪些角色？

**Broker**:一台 kafka 服务器就是一个 broker。一个集群由多个broker组成。

**Producer**:消息生产者。

**Consumer**:消息消费者。

**Topic**：Kafka的悄息通过 Topic 进行分类。 主题就好比数据库的表，或者文件系统里的文件夹。

**Partition**：一个非常大的 topic 可以分布到多个 broker（即服务器）上，一个topic 可以分为多个 partition.消息以追加的方式写入partition，后以先先出的顺序读取。每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序的id（offset）。kafka 只保证按一个 partition 中的顺序将消息发给 consumer，不保证一个 topic 的整体（多个partition间）的顺序；

**Consumer Group**（同一个消费者组中的消费者，一个分区只能被一个消费者消费）一个 Consumer Group 是多个 consumer 的组合, 作为一个整体存在.同一个消费者组内的消费者读取消息的时候, 不会读取同一个分区内的消息.但是组与组之间不受任何影响.

**Offset**：偏移量.kafka 的存储文件都是按照 offset.kafka 来命名用 offset 做名字的好处是方便查找。例如你想找位于 2049 的位置，只要找到2048.kafka 的文件即可。当然the first offset就是00000000000.kafka

## kafka ack 机制？集群中的ack 是怎么实现的？

Kafka producer有三种ack机制：

0 （意味着producer不等待broker同步完成的确认）

1（意味着producer要等待leader成功收到数据并得到确认）

-1（意味着producer得到follwer确认）

## kafka 的最小工作单元？

生产者、消费者、主题、偏移量

kafka 消息重复消费的问题？幂等怎么做的？

数据版本号，还有数据库唯一索引

产生重复消费的情况，比如说投递的时候重复了，消费的时候由于 offset 没处理好等等问题导致的话，我想可能会更好。

## 其它疑问

你们用的是Kafka？那你说说Kafka的底层架构原理，磁盘上数据如何存储的，整体分布式架构是如何实现的？

再说说Kafka是如何保证数据的高容错性的？零拷贝等技术是如何运用的？高吞吐量下如何优化生产者和消费者的性能？

看过Kafka的源码没有。如果看过，说说你对Kafka源码的理解？

kafka集群

项目中用到了 Kafka 这个框架，那么面试官就会问 Kafka 和 RocketMQ 之间的区别是什么

kafka为什么这么优秀(一) https://mp.weixin.qq.com/s/JPKYPazb6zJVwzVnncEdEg

kafka为什么这么优秀(二) https://mp.weixin.qq.com/s/jgd3e\_XnMw2Aluj9olJIFQ

# Netty NIO

## NIO和传统的IO有什么区别呢？

传统的IO，需要为每个连接创建一个线程，当并发的连接数量非常巨大时，线程所占用的栈内存和CPU线程切换的开销将非常巨大。使用NIO，不再需要为每个线程创建单独的线程，可以用一个含有限数量线程的线程池，甚至一个线程来为任意数量的连接服务。

IO是面向流的，一次可以读取一个或多个字节。NIO是缓冲区的，数据是先被 读/写到buffer中的，你可以控制读取什么位置的数据。

IO是阻塞式IO，当一条线程执行read()或者write()方法时，这条线程会一直阻塞知道读取到了一些数据或者要写出去的数据已经全部写出，在这期间这条线程不能做任何其他的事情。NIO在非阻塞式模式下，允许一条线程从channel中读取数据，通过返回值来判断buffer中是否有数据，如果没有数据，NIO不会阻塞。

Java NIO的selectors允许一条线程去监控多个channels的输入，你可以向一个selector上注册多个channel，然后调用selector的select()方法判断是否有新的连接进来或者已经在selector上注册时channel是否有数据进入。

NIO缺点：

需要额外做的工作是检查你需要的数据是否已经全部到了buffer中，你还需要保证当有更多的数据进入buffer中时，buffer中未处理的数据不会被覆盖

总结

NIO允许你用一个单独的线程或几个线程管理很多个channels（网络的或者文件的），代价是程序的处理和处理IO相比更加复杂

如果你需要同时管理成千上万的连接，但是每个连接只发送少量数据，例如一个聊天服务器，用NIO实现会更好一些，相似的，如果你需要保持很多个到其他电脑的连接，例如P2P网络，用一个单独的线程来管理所有出口连接是比较合适的

如果你只有少量的连接但是每个连接都占有很高的带宽，同时发送很多数据，传统的IO会更适合

## 那么NIO和IO各适用的场景是什么呢？

如果需要管理同时打开的成千上万个连接，这些连接每次只是发送少量的数据，例如聊天服务器，这时候用NIO处理数据可能是个很好的选择。

而如果只有少量的连接，而这些连接每次要发送大量的数据，这时候传统的IO更合适。使用哪种处理数据，需要在数据的响应等待时间和检查缓冲区数据的时间上作比较来权衡选择。

## ?netty的线程模型么？

Netty使用的是Reactor线程模型，Netty使用EventLoop来处理连接上的读写事件，而一个连接上的所有请求都保证在一个EventLoop中被处理，一个EventLoop中只有一个Thread，所以也就实现了一个连接上的所有事件只会在一个线程中被执行。一个EventLoopGroup包含多个EventLoop，可以把一个EventLoop当做是Reactor线程模型中的一个线程，而一个EventLoopGroup类似于一个ExecutorService，

# Dubbo

## ？Dubbo完整的一次调用链路介绍；

## dubbo的服务暴露过程

Dubbo会在Spring实例化完bean之后，在刷新容器最后一步发布ContextRefreshEvent事件的时候，通知实现了ApplicationListener的ServiceBean类进行回调onApplicationEvent 事件方法，dubbo会在这个方法中调用ServiceBean父类ServiceConfig的export方法，而该方法真正实现了服务的（异步或者非异步）发布。

由服务配置类 ServiceConfig 进行初始化工作及服务暴露入口，首先进去执行该类的export()方法。

export方法先判断是否需要延迟暴露（这里我们使用的是不延迟暴露），然后执行doExport方法。

doExport方法先执行一系列的检查方法，然后调用doExportUrls方法。检查方法会检测dubbo的配置是否在Spring配置文件中声明，没有的话读取properties文件初始化。

doExportUrls方法先调用loadRegistries获取所有的注册中心url，然后遍历调用doExportUrlsFor1Protocol方法。对于在标签中指定了registry属性的Bean，会在加载BeanDefinition的时候就加载了注册中心。

原文：https://blog.csdn.net/yanpenglei/article/details/80261762

## Dubbo Provider服务提供者要控制执行并发请求上限，具体怎么做？

在Provider上配置的Consumer端属性：actives，消费者端，最大并发调用限制，即当Consumer对一个服务的并发调用到上限后，新调用会Wait直到超时

Provider上配置的Provider端属性： executes，一个服务提供者并行执行请求上限，即当Provider对一个服务的并发调用到上限后，新调用会Wait（Consumer可能到超时）。在方法上配置（dubbo:method ）则并发限制针对方法，在接口上配置（dubbo:service），则并发限制针对服务。

**Dubbo启动的时候支持几种配置方式？**

根据 DUBBO 官方文档，配置 DUBBO 有 4 种方式，分别是：

XML 配置文件方式

properties 配置文件方式

annotation 配置方式

API 配置方式

**配置的优先级:**

1. 方法级配置别优于接口级别，即小Scope优先

2. Consumer端配置优于Provider配置，优于全局配置

3. Dubbo Hard Code的配置值（默认）

如：在dubbo的provider和consumer的配置文件中，如果都配置了timeout的超时时间，dubbo默认以consumer中配置的时间为准

## Dubbo框架

服务层：

Dubbo最大的特点就是按照分层的方式来架构，使用这种方式可以实现各个层之间最大限度地松耦合。Dubbo的框架设计一共划分为10层，有服务接口层，配置层，服务代理层，集群层，监控层，服务注册层，远程调用层，信息交换层，网络传输层，数据序列化层。其中我觉得最主要的就是服务代理层，服务注册层与集群层，服务代理层会将根据你写的接口生成一个代理类，用于处理请求，而服务注册层则会将服务地址暴露给注册中心，请求过来后再由底层转发给各个代理实现类去处理，并响应。

服务接口层（Service）：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。

配置层（Config）：对外配置接口，以ServiceConfig和ReferenceConfig为中心，可以直接new配置类，也可以通过spring解析配置生成配置类。

服务代理层（Proxy）：服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。

服务注册层（Registry）：封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory、Registry和RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。

集群层（Cluster）：封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster、Directory、Router和LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方来透明，只需要与一个服务提供方进行交互。

监控层（Monitor）：RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory、Monitor和MonitorService。

远程调用层（Protocol）：封将RPC调用，以Invocation和Result为中心，扩展接口为Protocol、Invoker和Exporter。Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。Invoker是实体域，它是Dubbo的核心模型，其它模型都向它靠扰，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。

信息交换层（Exchange）：封装请求响应模式，同步转异步，以Request和Response为中心，扩展接口为Exchanger、ExchangeChannel、ExchangeClient和ExchangeServer。

网络传输层（Transport）：抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel、Transporter、Client、Server和Codec。

数据序列化层（Serialize）：可复用的一些工具，扩展接口为Serialization、 ObjectInput、ObjectOutput和ThreadPool。

# Zookeeper

## Zookeeper在选举的过程中，还能对外提供服务么？

在Zookeeper运行期间，如果Leader节点挂了，那么整个Zookeeper集群将暂停对外服务，进入新一轮Leader选举。

## ZooKeeper作为发现服务的问题

在分布式系统领域有个著名的CAP定理（C-数据一致性；A-服务可用性；P-服务对网络分区故障的容错性)ZooKeeper是个CP的，即任何时刻对ZooKeeper的访问请求能得到一致的数据结果，同时系统对网络分割具备容错性；但是它不能保证每次服务请求的可用性

作为一个分布式协同服务，ZooKeeper非常好，但是对于Service发现服务来说就不合适了；因为对于Service发现服务来说就算是返回了包含不实的信息的结果也比什么都不返回要好；

## 对Paxos算法了解多少？

Paxos是一种分布式一致性算法，

Paxos算法定义了三种角色：Acceptor、Proposer和Learner，其中Proposer是提议发起者，负责接收客户端请求，并将客户端的请求发送到Paxos集群中，以便决定这个值是否可以被批准；Acceptor是提议批准者，负责处理接收到的提议；Learner是学习者，只能学习到已经被批准的值，不能学习没有被批准的值。每一个进程都可以扮演其中任意一种角色。

Paxos完整算法核心过程包含prepare和accept两个阶段，可大致描述为：

1) prepare阶段：Proposer向Acceptor发起提议权申请请求（Prepare消息），Acceptor负责批准Proposer申请的提议权，如果同意就回复Promise消息

2) accept阶段：Proposer一旦获得提议权即可进一步提交变量取值（Accept消息），Acceptor负责批准Proposer提交的变量取值，如果接受就回复Accepted消息

# Dump文件分析

# 分布式系统所有问题解决思路概括

分布式和Dubbo的问题

这是一个老生常谈的话题，也是这几年比较火的话题，说起分布式就一定和 Dubbo 有关系，但是不能仅仅就理解到 Dubbo。首先我们需要思考它解决的问题，为什么要引入 Dubbo这个概念。随着业务的发展、用户量的增长，系统数量增多，调用依赖关系也变得复杂，为了确保系统高可用、高并发的要求，系统的架构也从单体时代慢慢迁移至服务SOA时代，应运而生的 Dubbo 出现了，它作为 RPC 的出现使得我们搭建微服务项目变得简单，但是我们不仅仅要思考Dubbo带来的框架支撑。同时需要思考服务的幂等、分布式事务、服务之间的 Trace 定位、分布式日志、数据对账、重试机制等，与此同时考虑 MQ 对系统的解耦和压力的分担、数据库分布式部署和分库分表、限流、熔断等机制。所以最终总结是不仅仅要看 Dubbo 的使用、原理同时还要思考上下游和一些系统设计的问题，这块相对的知识点较多，可以针对上面抛出来的点各个击破。

分布式一致性session

分布式锁

分布式事务

分布式事务是指事务的参与者，支持事务的服务器，资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。

分库分表中间件及部署

# 服务器

**Nginx 并发模型**

Nginx 的并发模型是一个多进程并发模型，它的 Master 进程在绑定监听地址端口后 fork 出了多个 Slave 进程共同竞争处理这个服务端套接字接收到的很多客户端连接。

# 开放题

## 如何访问链表中间节点

对于这个问题，我们首先能够想到的就是先遍历一遍整个的链表，然后计算出链表的长度，进而遍历第二遍找出中间位置的数据。这种方式非常简单。

若题目要求只能遍历一次链表，那又当如何解决问题？

可以采取建立两个指针，一个指针一次遍历两个节点，另一个节点一次遍历一个节点，当快指针遍历到空节点时，慢指针指向的位置为链表的中间位置，这种解决问题的方法称为快慢指针方法。

## 有限内存的海量数据处理的题目

一般这类题目的解答无非是以下几种：

分治，hash映射，堆排序，双层桶划分，Bloom Filter，bitmap，数据库索引，mapreduce等。