# Spring

# SpringBoot与SpringCloud

# Mybatis

# 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？

IO是面向流的，NIO是面向块（缓冲区）的。

IO面向流的操作一次一个字节地处理数据。一个输入流产生一个字节的数据，一个输出流消费一个字节的数据。，导致了数据的读取和写入效率不佳。

NIO面向块的操作在一步中产生或者消费一个数据块。按块处理数据比按(流式的)字节处理数据要快得多，同时数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。通俗来说，NIO采取了“预读”的方式，当你读取某一部分数据时，他就会猜测你下一步可能会读取的数据而预先缓冲下来。

## 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。