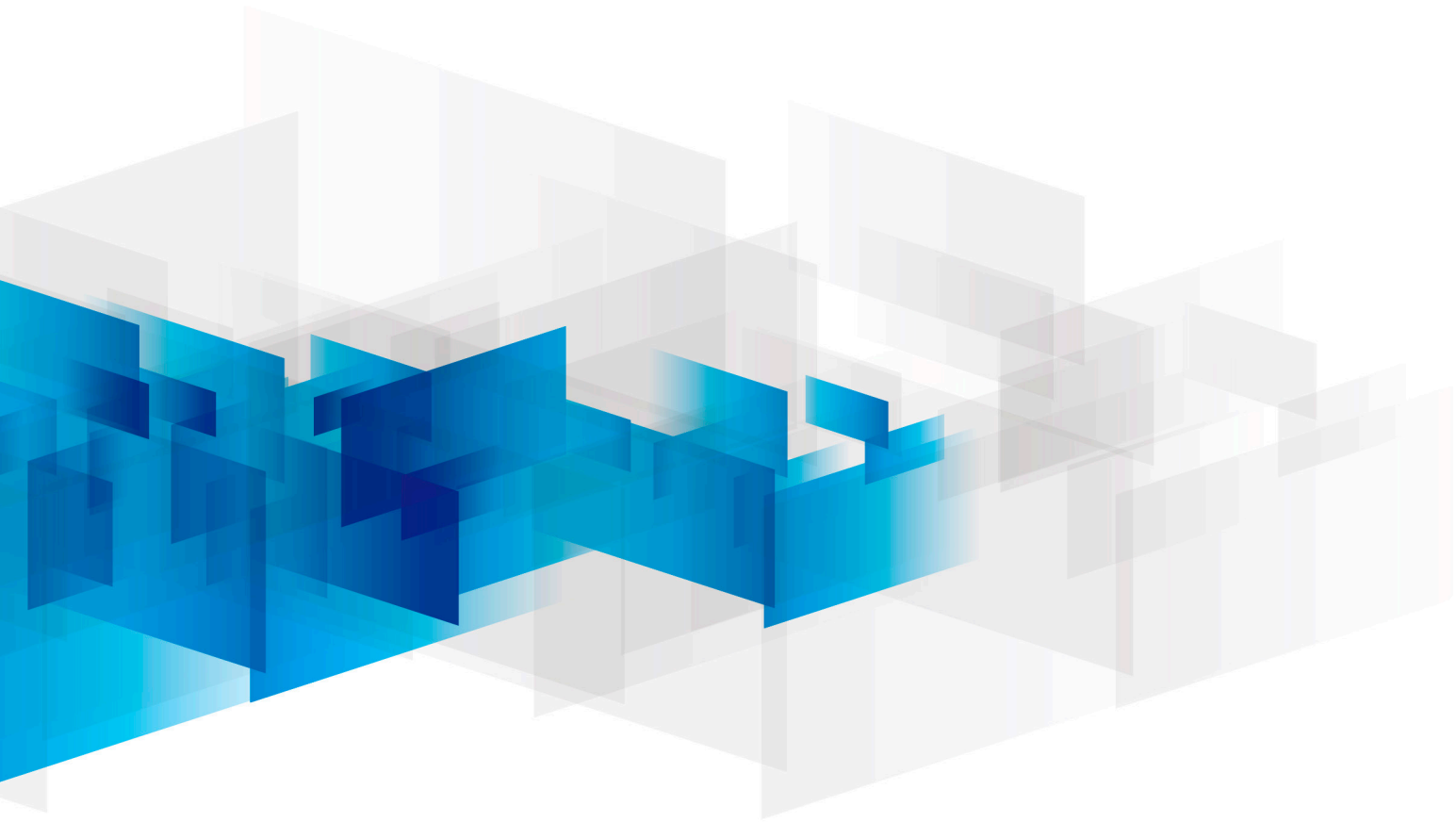
kaikeba 开课吧

题库

大厂面试

Java高频100题



DA-CHANG-GAO-PIN-MIAN-SHI-100-TI

Dubbo

Q1：服务调用超时问题怎么解决

A：

消费者调用服务超时会引起服务降级的发生，即从发出调用请求到获取到提供者的响应结果这个时间超出了设定的时限。默认服务调用超时时限为 1 秒。可以在消费者端与提供者端设置超时时限来解决。

总的来说还是要设计好业务代码来减少调用时长，设置准确 RPC 调用的超时时间才能更好的解决这个问题。

Q2：Dubbo 支持哪些序列化方式?

A：

默认使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。

Q3：Dubbo 和 SpringCloud 的关系?

A：

Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流量监控和熔断。而 SpringCloud 诞生于微服务架构时代，考虑的是微服务治理的方方面面，另外由于依托了 Spirng、SpirngBoot 的优势之上，两个框架在开始目标就不一致，Dubbo 定位服务治理、SpirngCloud 是一个生态。

Q4：Dubbo 的架构设计?

A：

Dubbo 框架设计一共划分了 10 个层：

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。

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。

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。

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。

监控层（Monitor）：RPC 调用次数和调用时间监控。

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。
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。
网络传输层（Transport）：抽象 mina 和 netty 为统一接口，以 Message 为中心。

Q5：Dubbo 的默认集群容错方案?

A：

FailoverCluster

Q6：Dubbo 使用的是什么通信框架？

A：

默认使用 NIO Netty 框架

Q7：Dubbo 的主要应用场景?

A：

透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何 API 侵入。软负载均衡及容错机制，可在内网替代 F5 等硬件负载均衡器，降低成本，减少单点。
服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的 IP 地址，并且能够平滑添加或删除服务提供者。

Q8：Dubbo 服务注册与发现的流程?

A：

流程说明：
Provider(提供者) 绑定指定端口并启动服务·指供者连接注册中心，并发本机 IP、端口、应用信息和提供服务信息发送至注册中心存储
Consumer(消费者），连接注册中心，并发送应用信息、所求服务信息至注册中心·注册中心根据消费者所求服务信息匹配对应的提供者列表发送至 Consumer 应用缓存。
Consumer 在发起远程调用时基于缓存的消费者列表择其一发起调用。
Provider 状态变更会实时通知注册中心、在由注册中心实时推送至
Consumer
设计的原因：
Consumer 与 Provider 解偶，双方都可以横向增减节点数。
注册中心对本身可做对等集群，可动态增减节点，并且任意一台宕掉后，将自动切换到另一台去中心化，双方不直接依懒注册中心，即使注册中心全部宕机短时间内也不会影响服务的调用服务提供者无状态，任意一台宕掉后，不影响使用

Q9：Dubbo 的负载均衡是如何实现的?

A:

无论是 Dubbo 还是 Spring Cloud，负载均衡原理都是相同的。消费者从注册中心中获取到当前其要消费服务的所有提供者，然后对这些提供者按照指定的负载均衡策略获取到其中的一个提供者，然后进行调用。当调用失败时，会再按照容错机制进行处理。

不同的是，Dubbo 中还可以在负载均衡之前先根据设定好的路由规则对所有可用提供者进行路由，从中筛选掉一部分不符合规则的提供者，对剩余提供者再进行负载均衡。

Dubbo 中默认支持四种负载均衡策略：加权随机策略、加权最小活跃度调度策略、双权重轮询策略，及一致性 hash 策略。

Spring Cloud(H 版) 一般使用 Ribbon 实现负载均衡，其默认支持五种负载均衡策略：轮询策略、随机策略、重试策略、最可用策略，及可用过滤策略。

Q10：Dubbo 的四大组件

A:

1、Provider：服务提供者。

2、Consumer：服务消费者。

3、Registry：服务注册与发现的中心，提供目录服务，亦称为服务注册中心

4、Monitor：统计服务的调用次数、调用时间等信息的日志服务，并可以对服务设置权限、降级处理等，称为服务管控中心

Q11：Dubbo 在安全机制方面是如何解决的

A:

Dubbo 通过 Token 令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。Dubbo 还提供服务黑白名单，来控制服务所允许的调用方。

Q12：Dubbo 和 SpringCloud 中的 OpenFeign 组件的区别?

A:

最大的区别：Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于 TCP 协议传输的，配合以 Hession 序列化完成 RPC 通信。

而 SpringCloud 是基于 Http 协议 +Rest 接口调用远程过程的通信，相对来说，Http 请求会有更大的报文，占的带宽也会更多。但是 REST 相比 RPC 更为灵活。

Q13：Dubbo 支持哪些协议，每种协议的应用场景，优缺点?

A：

dubbo 协议：
Dubbo 默认传输协议
连接个数：单连接
连接方式：长连接
协议：TCP
传输方式：NIO 异步传输
适用范围：传入传出参数数据包较小（建议小于 100K），消费者比提供者个数多，单一 消费者无法压满提供者，尽量不要用 dubbo 协议传输大文件或超大字符串。
rmi：采用 JDK 标准的 rmi 协议实现，传输参数和返回参数对象需要实现 Serializable 接口，使用 java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。多个短连接，TCP 协议传输，同步传输，适用常规的远程服务调用和 rmi 互操作。在依赖低版本的 Common-Collections 包，java 序列化存在安全漏洞；
webservice：基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；
http：基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用；hessian：集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hession 服务互操作。多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；
memcache：基于 memcached 实现的 RPC 协议
redis：基于 redis 实现的 RPC 协议

Q14：Dubbo 的核心功能有哪些?

A：

主要就是如下 3 个核心功能：
Remoting：网络通信框架，提供对多种 NIO 框架抽象封装，包括“同步转异步”和“请求 - 响应”模式的信息交换方式。
Cluster：服务框架，提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。
Registry：服务注册，基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

Q15：Dubbo 的注册中心集群挂掉，发布者和订阅者之间还能通信么?

A:

可以的，启动 dubbo 时，消费者会从 zookeeper 拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用。

Q16：Dubbo 的集群容错方案有哪些?

A:

FailoverCluster：失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。

FailfastCluster：快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。FailsafeCluster：失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

FailbackCluster：失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

ForkingCluster：并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks=”2″来设置最大并行数。

BroadcastCluster：广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信

Q17：Dubbo 集群的负载均衡有哪些策略

A:

Dubbo 提供了常见的集群策略实现，并预扩展点予以自行实现。

A、 random： 随机算法，是 Dubbo 默认的负载均衡算法。存在服务堆积问题。

B、 roundrobin： 轮询算法。按照设定好的权重依次进行调度。

C、 leastactive： 最少活跃度调度算法。即被调度的次数越少，其优选级就越高，被调度到的机率就越高。

D、 consistenthash： 一致性 hash 算法。对于相同参数的请求，

其会被路由到相同的提供者。

Q18：为什么需要服务治理?

A:

服务治理是主要针对分布式服务框架的微服务，处理服务调用之间的关系、服务发布和发现、故障监控与处理，服务的参数配置、服务降级和熔断、服务使用率监控等。

原因:

过多的服务 URL 配置困难
负载均衡分配节点压力过大的情况下也需要部署集群
服务依赖混乱，启动顺序不清晰
过多服务导致性能指标分析难度较大，需要监控
故障定位与排查难度较大

Q19：Dubbo 超时时间怎样设置?

A:

Dubbo 超时时间设置有两种方式：
服务提供者端设置超时时间，在 Dubbo 的用户文档中，推荐如果能在服务端多配置就尽量多配置，因为服务提供者比消费者更清楚自己提供的服务特性。
服务消费者端设置超时时间，如果在消费者端设置了超时时间，以消费者端为主，即优先级更高。因为服务调用方设置超时时间控制性更灵活。如果消费方超时，服务端线程不会定制，会产生警告。

Q20：Dubbo 框架源码最重要的设计原则是什么？从架构设计角度谈一下你对这个设计原则的理解。

A:

Dubbo 在设计时具有两大设计原则：
“微内核 + 插件”的设计模式。内核只负责组装插件（扩展点），Dubbo 的功能都是由插件实现的，也就是 Dubbo 的所有功能点都可被用户自定义扩展类所替换。Dubbo 的高扩展性、开放性在这里被充分体现。
采用 URL 作为配置信息的统一格式，所有扩展点都通过传递 URL 携带配置信息。简单来说就是，在 Dubbo 中，所有重要资源都是以 URL 的形式来描述的。

Q21：为什么 Dubbo 使用 URL，而不使用 JSON，使用 URL 的好处是什么?

A:

关于这个问题，官方是没有相关说明的，下面我谈两点我个人的看法：
首先，Dubbo 是将 URL 作为公共契约出现的，即希望所有扩展点都要遵守的约定。既然是约定，那么可以这样约定，也可以那样约定。只要统一就行。所以，在 Dubbo 创建之初，也许当时若采用了 JSON 作为这个约定也是未偿不可的。
其次，单从 JSON 与 URL 相比而言，都是一种简洁的数据存储格式。但在简洁的同时，URL 与 Dubbo 应用场景的契合度更高些。因为 Dubbo 中 URL 的所有应用场景都与通信有关，

都会涉及到通信协议、通信主机、端口号、业务接口等信息。其语义性要强于 JSON，且对于这些数据就无需再给出相应的 key 了，会使传输的数据量更小。

Q22：请简述一下 Dubbo 四大组件间的关系。

A：

Dubbo 的四大组件为：Consuer、Provider、Registry 与 Monitor。它们间的关系可以描述为如下几个

过程：

start：Dubbo 服务启动，Spring 容器首先会创建服务提供者。

register：服务提供者创建好后，马上会注册到服务注册中心 Registry，这个注册过程称为服务暴露。服务暴露的本质是将服务名称（接口）与服务提供者主机写入到注册中心 Registry 的服务映射表中。注册中心充当着“DNS 域名服务器”的角色。

subscribe：服务消费者启动后，首先会向服务注册中心订阅相关服务。

notify：消费者可能订阅的服务在注册中心还没有相应的提供者。当相应的提供者在注册中心注册后，注册中心会马上通知订阅该服务的消费者。但消费者在订阅了指定服务后，在没有收到注册中心的通知之前是不会被阻塞的，而是可以继续订阅其它服务。

invoke：消费者以同步或异步的方式调用提供者提供的请求。消费者通过远程注册中心获取到提供者列表，然后消费者会基于负载均衡算法选一台提供者处理消费者的请求。

count：每个消费者对各个服务的累计调用次数、调用时间；每个提供者被消费者调用的累计次数和时间，消费者与调用者都会定时发送到监控中心，由监控中心记录。这些统计数据可以在 Dubbo 的可视化界面看到。

Q23：什么是 SPI？请简单描述一下 SPI 要解决的问题。

A：

SPI，Service Provider Interface，服务提供者接口，是一种服务发现机制。其主要是解决面向抽象编程中上层对下层接口实现类的依赖问题，可以实现这两层间的解耦合。

Q24：JDK 的 SPI 机制存在什么问题?

A：

JDK 的 SPI 机制将所有配置文件中的实现类全部实例化，无论是否能够用到，浪费了宝贵的系统资源。

Q25：简述 Dubbo 的 Wrapper 机制

A：

Wrapper 机制，即扩展类的包装机制。就是对扩展类中的 SPI 接口方法进行增强，进行包装，

是 AOP 思想的体现，是 Wrapper 设计模式的应用。一个 SPI 可以包含多个 Wrapper，即可以通过多个 Wrapper 对同一个扩展类进行增强，增强不出现的功能。Wrapper 机制不是通过注解实现的，而是通过一套 Wrapper 规范实现的。

Q26：Dubbo 的 Wrapper 类是否属于扩展类?

A：

wrapper 类仅仅是对现有的扩展类功能上的增强，并不是一个独立的扩展类，所以其不属于扩展类范畴。

Q27：简述 Dubbo 的 Active 机制

A：

Activate 机制，即扩展类的激活机制。通过指定的条件来实现一次激活多个扩展类的目的。激活机制没有增强扩展类，也没有增加扩展类，其仅仅是为原有的扩展类添加了更多的识别标签，而不像之前的，每个扩展类仅有一个“功能性扩展名”识别标签。其是通过 @Active 注解实现的。

Q28：Dubbo 的 Activate 类是否属于扩展类?

A：

Activate 机制仅用于为扩展类添加激活标识的，其是通过在扩展类上添加 @Activate 注解来实现的，所以 Activate 类本身就是扩展类。

Q29：简述 Dubbo 中配置中心与注册中心的关系

A：

Dubbo 中的注册中心是用于完成服务发现的，而配置中心是用于完成配置信息的统一管理的。若没有专门设置配置中心，系统会默认将注册中心服务器作为配置中心服务器。

Q30：Dubbo 内核工作原理的四个构成机制间的关系是怎样的？或者说，一个扩展类实例获取过程是怎样的?

A：

获取一个扩展类实例，一般需要经过这样几个环节：

获取到该 SPI 接口的 ExtensionLoader。而这个获取的过程会将该 SPI 接口的所有扩展类（四类）加载并缓存。

通过 extensionLoader 获取到其自适应实例。通常 SPI 接口的自适应实例都是由 Adaptive 方法自动生成的，所以需要对这个自动生成的 Adaptive 类进行动态编译。

在通过自适应实例调用自适应的业务方法时，才会获取到其真正需要的扩展类实例。所以说，

一个扩展类实例一般情况下是在调用自适应方法时才创建。
在获取这个真正的扩展类实例时，首先会根据要获取的扩展类实例的“功能性扩展名”，从扩展类缓存中找到其对应的扩展类，然后调用其无参构造器，创建扩展类实例 instance。
通过 injectExtension(instance) 方法，调用 instance 实例的 setter 完成初始化。
遍历所有该 SPI 的 Wrapper，逐层包装这个 setter 过的 instance。此时的这个 instance，即 wrapper 实例就是我们需要获取的扩展类实例。

ElasticSearch

Q31：你们公司的 ES 集群，一个 node 一般会分配几个分片?

A：

我们遵循官方建议，一个 Node 最好不要多于三个 shards。

Q32：Elasticsearch 是如何实现 Master 选举的?

A：

Elasticsearch 的选主是 ZenDiscovery 模块负责的，主要包含 Ping（节点之间通过这个 RPC 来发现彼此）和 Unicast（单播模块包含一个主机列表以控制哪些节点需要 ping 通）这两部分；
对所有可以成为 master 的节点（node.master: true）根据 nodeId 字典排序，每次选举每个节点都把自己所知道节点排一次序，然后选出第一个（第 0 位）节点，暂且认为它是 master 节点。
如果对某个节点的投票数达到一定的值（可以成为 master 节点数 n/2+1）并且该节点自己也选举自己，那这个节点就是 master。否则重新选举一直到满足上述条件。

Q33：你是如何做写入调优的?

A：

1）写入前副本数设置为 0；
2）写入前关闭 refresh_interval 设置为 -1，禁用刷新机制；
3）写入过程中：采取 bulk 批量写入；
4）写入后恢复副本数和刷新间隔；
5）尽量使用自动生成的 id。

Q34：如何避免脑裂?

A：

可以通过设置最少投票通过数量（discovery.zen.minimum_master_nodes）超过所有候

选节点一半以上来解决脑裂问题。

Q35：Elasticsearch 对于大数据量（上亿量级）的聚合如何实现?

A:

Elasticsearch 提供的首个近似聚合是 cardinality 度量。它提供一个字段的基数，即该字段的 distinct 或者 unique 值的数目。它是基于 HLL 算法的。HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。

其特点是：

可配置的精度，用来控制内存的使用（更精确 = 更多内存）；

小的数据集精度是非常高的；

我们可以通过配置参数，来设置去重需要的固定内存使用量。无论数千还是数十亿的唯一值，内存使用量只与你配置的精确度相关 。

Q36：ES 主分片数量可以在后期更改吗？为什么?

A:

不可以，因为根据路由算法 shard = hash(document_id) % (num_of_primary_shards)，当主分片数量变化时会影响数据被路由到哪个分片上。

Q37：如何监控集群状态?

A:

Marvel 件让你可以很简单的通过 Kibana 监控 Elasticsearch。你可以实时查看你的集群健康状态和性能，也可以分析过去的集群、索引和节点指标。

Q38：ElasticSearch 中的副本是什么?

A:

一个索引被分解成碎片以便于分发和扩展。副本是分片的副本。一个节点是一个属于一个集群的 ElasticSearch 的运行实例。一个集群由一个或多个共享相同集群名称的节点组成。

Q39：ES 更新数据的执行流程?

A:

(1) 将原来的 doc 标识为 deleted 状态，然后新写入一条数据。

(2)buffer 每 refresh 一次，就会产生一个 segmentfile，所以默认情况下是 1s 一个

segmentfile，

segmentfile 会越来越多，此时会定期执行 merge。
(3) 每次 merge 时，会将多个 segmentfile 合并成一个，同时这里会将标识为 deleted 的 doc 给物理删除掉，然后将新的 segmentfile 写入磁盘，这里会写一个 commitpoint，标识所有新的 segmentfile，然后打开 segmentfile 供搜索使用，同时删除旧的 segmentfile。

Q40：shard 里面是什么组成的?

A：

是多个 segment 组成的。

Q41：ElasticSearch 中的分析器是什么?

A：

在 ElasticSearch 中索引数据时，数据由为索引定义的 Analyzer 在内部进行转换。 分析器由一个 Tokenizer 和零个或多个 TokenFilter 组成。编译器可以在一个或多个 CharFilter 之前。分析模块允许您在逻辑名称下注册分析器，然后可以在映射定义或某些 API 中引用它们。Elasticsearch 附带了许多可以随时使用的预建分析器。或者，您可以组合内置的字符过滤器，编译器和过滤器器来创建自定义分析器。

Q42：客户端在和集群连接时，如何选择特定的节点执行请求的?

A：

TransportClient 利用 transport 模块远程连接一个 elasticsearch 集群。它并不加入到集群中，只是简单的获得一个或者多个初始化的 transport 地址，并以 轮询 的方式与这些地址进行通信。

Q43：Elasticsearch 中的倒排索引是什么?

A：

倒排索引是搜索引擎的核心。搜索引擎的主要目标是在查找发生搜索条件的文档时提供快速搜索。倒排索引是一种像数据结构一样的散列图，可将用户从单词导向文档或网页。它是搜索引擎的核心。其主要目标是快速搜索从数百万文件中查找数据。

Q44：什么是脑裂?

A：

一个正常 es 集群中只有一个主节点，主节点负责管理整个集群，集群的所有节点都会选择同一个节点作为主节点所以无论访问那个节点都可以查看集群的状态信息。而脑裂问题的出现就是因为从节点在选择主节点上出现分歧导致一个集群出现多个主节点从而使集群分裂，

使得集群处于异常状态。

Q45：什么是索引?

A:

索引（名词）一个索引 (index) 就像是传统关系数据库中的数据库，它是相关文档存储的地方，index 的复数是 indices 或 indexes。

索引（动词） 「索引一个文档」表示把一个文档存储到索引（名词）里，以便它可以被检索或者查询。这很像 SQL 中的 INSERT 关键字，差别是，如果文档已经存在，新的文档将覆盖旧的文档。

Q46：详细描述一下 Elasticsearch 更新和删除文档的过程

A:

删除和更新也都是写操作，但是 Elasticsearch 中的文档是不可变的，因此不能被删除或者改动以展示其变更；

磁盘上的每个段都有一个相应的 .del 文件。当删除请求发送后，文档并没有真的被删除，而是在 .del 文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并时，在 .del 文件中被标记为删除的文档将不会被写入新段。

在新的文档被创建时，Elasticsearch 会为该文档指定一个版本号，当执行更新时，旧版本的文档在 .del 文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是会在结果中被过滤掉。

JVM°

Q47：JVM 参数主要有几种分类

A:

标准参数

标准参数，顾名思义，标准参数中包括功能以及输出的结果都是很稳定的，基本上不会随着 JVM 版本的变化而变化。标准参数以 - 开头，如：java -version、java -jar 等，通过 java -help 可以查询所有的标准参数。

非标准参数

非标准参数以 -X 开头，是标准参数的扩展。对应前面讲的标准化参数，这是非标准化参数。表示在将来的 JVM 版本中可能会发生改变，但是这类以 -X 开始的参数变化的比较小。

不稳定参数

这是我们日常开发中接触到最多的参数类型。这也是非标准化参数，相对来说不稳定，随着 JVM 版本的变化可能会发生变化，主要用于 JVM 调优和 debug。

不稳定参数以 -XX 开头，此类参数的设置很容易引起 JVM 性能上的差异，使 JVM 存在极大

的不稳定性。
如果此类参数设置合理将大大提高 JVM 的性能及稳定性。
不稳定参数分为三类：
性能参数：用于 JVM 的性能调优和内存分配控制，如内存大小的设置；
行为参数：用于改变 JVM 的基础行为，如 GC 的方式和算法的选择；
调试参数：用于监控、打印、输出 jvm 的信息；

Q48：Java 中会存在内存泄漏吗，简述一下

A：

理论上 Java 因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是 Java 被广泛使用于服务器端编程的一个重要原因）；然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被 GC 回收，因此也会导致内存泄露的发生。例如 hibernate 的 Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。下面例子中的代码也会导致内存泄露。

Q49：Java 中都有哪些引用类型

A：

强引用：发生 gc 的时候不会被回收。new
软引用：有用但不是必须的对象，在发生内存溢出之前会被回收。SoftReference
弱引用：有用但不是必须的对象，在下一次 GC 时会被回收。WeakReference
虚引用（幽灵引用 / 幻影引用）：无法通过虚引用获得对象，用 PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知。

Q50：在 Java 中，对象什么时候可以被垃圾回收?

A：

首先先由可达性算法去判断对象是否可回收
二次标记，相当于二次审判。finalize 方法
其次再去根据 GC 的回收机制，择时回收

Q51：StackOverflow 异常有没有遇到过？一般你猜测会在什么情况下被触发?

A：

答：栈内存溢出，一般由栈内存的局部变量过爆了，导致内存溢出。出现在递归方法，参数个数过多，递归过深，递归没有出口。

Q52：堆空间分哪几个部分？以及如何设置各个部分大小？

A：

Java 堆被所有线程共享，在 Java 虚拟机启动时创建。是虚拟机管理最大的一块内存。
Java 堆是垃圾回收的主要区域，而且主要采用分代回收算法。堆进一步划分主要是为了更好的回收内存或更快的分配内存。
Java 虚拟机规范的描述是：所有的对象实例以及数组都要在堆上分配。
堆内存空间在物理上可以不连续，逻辑上连续即可。
堆大小 = 新生代 + 老年代。
堆的大小可通过参数 –Xms（堆的初始容量）、-Xmx（堆的最大容量） 来指定。

其中，新生代 (Young) 被细分为 Eden 和 两个 Survivor 区域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分。
默认的，Eden : from : to = 8 : 1 : 1 。(可以通过参数 –XX:SurvivorRatio 来设定 。)

即： Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。
JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块 Survivor 区域是空闲着的。
新生代实际可用的内存空间为 9/10 (即 90%) 的新生代空间。
-Xmn：至于这个参数则是对 -XX:newSize、-XX:MaxnewSize 两个参数的同时配置，也就是说如果通过 -Xmn 来配置新生代的内存大小，

那么 -XX:newSize = -XX:MaxnewSize = -Xmn，虽然会很方便，但需要注意的是这个参数是在 JDK1.4 版本以后才使用的。

Q53：什么是栈帧？栈帧存储了什么？

A：

虚拟机栈也是线程私有，而且生命周期与线程相同，每个 Java 方法在执行的时候都会创建一个栈帧（Stack Frame）。栈帧 (Stack Frame) 是用于支持虚拟机进行方法调用和方法执行的数据结构。栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用至执行完成的过程，都对应着一个栈帧在虚拟机栈里从入栈到出栈的过程。

Q54：如何设置参数生成 GC 日志

A：

设置 JVM GC 格式日志的主要参数包括如下 8 个：
1. -XX:+PrintGC 输出简要 GC 日志

2. -XX:+PrintGCDetails 输出详细 GC 日志
3. -Xloggc:gc.log 输出 GC 日志到文件
4. -XX:+PrintGCTimeStamps 输出 GC 的时间戳（以 JVM 启动到当期的总时长的时间戳形式）
5. -XX:+PrintGCDateStamps 输出 GC 的时间戳（以日期的形式，如 2020-04-

26T21:53:59.234+0800）
6. -XX:+PrintHeapAtGC 在进行 GC 的前后打印出堆的信息
7. -verbose:gc : 在 JDK 8 中， -verbose:gc 是 -XX:+PrintGC 一个别称，日志格式等价与： -XX:+PrintGC 。不过在 JDK 9 中 -XX:+PrintGC 被标记为 deprecated。
-verbose:gc 是一个标准的选项，-XX:+PrintGC 是一个实验的选项，建议使用 -verbose:gc 替代 -XX:+PrintGC

8. -XX:+PrintReferenceGC 打印年轻代各个引用的数量以及时长
开启 GC 日志

多种方法都能开启 GC 的日志功能，其中包括：使用 -verbose:gc 或 -XX:+PrintGC 这两个标志中的任意一个能创建基本的 GC 日志（这两个日志标志实际上互为别名，默认情况下的 GC 日志功能是关闭的）使用
XX:+PrintGCDetails 标志会创建更详细的 GC 日志推荐使用 -XX:+PrintGCDetails 标志（这个标志默认情况下也是关闭的）；通常情况下使用基本的 GC 日志很难诊断垃圾回收时发生的问题。
开启 GC 时间提示
除了使用详细的 GC 日志，我们还推荐使用 -XX:+PrintGCTimeStamps 或者 -XX:+PrintGCDateStamps，便于我们更精确地判断几次 GC 操作之间的时间。这两个参数之间的差别在于时间戳是相对于 0（依据 JVM 启动的时间）的值，而日期戳（date stamp）是实际的日期字符串。由于日期戳需要进行格式化，所以它的效率可能会受轻微的影响，不过这种操作并不频繁，它造成的影响也很难被我们感知。
指定 GC 日志路径
默认情况下 GC 日志直接输出到标准输出，不过使用 -Xloggc:filename 标志也能修改输出到某个文件。除非显式地使用 -PrintGCDetails 标志，否则使用 -Xloggc 会自动地开启基本日志模式。使用日志循环（Logrotation）标志可以限制保存在 GC 日志中的数据量；对于需要长时间运行的服务器而言，这是一个非常有用的标志，否则累积几个月的数据很可能会耗尽服务器的磁盘。
开启日志滚动输出
通过 -XX:+UseGCLogfileRotation -XX:NumberOfGCLogfiles=N -XX:GCLogfileSize=N 标志可以控制日志文件的循环。默认情况下，UseGCLogfileRotation 标志是关闭的。它负责打开或关闭 GC 日志滚动记录功能的。要求必须设置 -Xloggc 参数开启 UseGCLogfileRotation

标志后，默认的文件数目是 0（意味着不作任何限制），默认的日志文件大小是 0（同样也是不作任何限制）。因此，为了让日志循环功能真正生效，我们必须为所有这些标志设定值。需要注意的是：
The size of the log file at which point the log will be rotated, must be >= 8K. 设置滚动日志文件的大小，必须大于 8k。当前写日志文件大小超过该参数值时，日志将写入下一个文件设置滚动日志文件的个数，必须大于等于 1 必须设置 -Xloggc 参数

Q55：GC 是什么？为什么要有 GC？

A：

GC 是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。
Java 程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或 Runtime.getRuntime().gc() ，但 JVM 可以屏蔽掉显示的垃圾回收调用。
垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在 Java 诞生初期，垃圾回收是 Java 最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今 Java 的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得 iOS 的系统比 Android 系统有更好的用户体验，其中一个深层次的原因就在于 Android 系统中垃圾回收的不可预知性。
第 3 行中生成的 object 在第几行执行后成为 garbage collection 的对象？

```
public class MyClass {
public StringBuffer aMethod() {
StringBuffer sf = new StringBuffer(“Hello”);
StringBuffer[] sf_arr = new StringBuffer[1];
sf_arr[0] = sf;
sf = null;
sf_arr[0] = null;
return sf;
}
}
```

A：

第 7 行

Q56：Java 虚拟机是如何判定两个 Java 类是相同的?

A：

① 类的全限定名是否相等

② 类加载器是否相等

即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。比如一个 Java 类 com.example.Sample，编译之后生成了字节代码文件 Sample.class。两个不同的类加载器 ClassLoaderA 和 ClassLoaderB 分别读取了这个 Sample.class 文件，并定义出两个 java.lang.Class 类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。

Q57：使用过哪些 jdk 命令，并说明各自的作用是什么

A：

jps

jps:Java Virtual Machine Process Status Tool

查看 Java 进程 ，相当于 Linux 下的 ps 命令，只不过它只列出 Java 进程。

jstat

jstat:JVM Statistics Monitoring Tool

jstat 可以查看 Java 程序运行时相关信息，可以通过它查看堆信息的相关情况

jinfo

jinfo：Java Configuration Info

jinfo 可以用来查看正在运行的 java 程序的扩展参数，甚至支持运行时，修改部分参数

jmap

jmap:Memory Map

jmap 用来查看堆内存使用状况，一般结合 jhat 使用。

jstack

jstack：Java Stack Trace，jstack 是 java 虚拟机自带的一种堆栈跟踪工具。jstack 用于生成 java 虚拟机当前时刻的线程快照。线程快照是当前 java 虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。 线程出现停顿的时候通过 jstack 来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。 如果 java 程序崩溃生成 core 文件，jstack 工具可以用来获得 core 文件的 java stack 和 native stack 的信息，从而可以轻松地知道 java 程序是如何崩溃和在程序何处发生问题。另外，jstack 工具还可以附属到正在运行的 java 程序中，看到当时运行的 java 程序的 javastack 和 native stack 的信息 , 如果现在运行的 java 程序呈现 hung 的状态，jstack 是非常有用的。

jconsole
Jconsole:Java Monitoring and Management Console，Java 5 引入，一个内置 Java 性能分析器，
可以从命令行或在 GUI shell 中运行。您可以轻松地使用 JConsole 来监控 Java 应用程序性能和跟踪 Java 中的代码。

Q58：JVM 运行时数据区区域分为哪几部分?

A:

线程共享：堆、方法区
线程私有：虚拟机栈、本地方法栈、程序计数器
jdk1.7 之前，HotSpot 虚拟机对于方法区的实现称之为“永久代”， Permanent Generation。
jdk1.8 之后，HotSpot 虚拟机对于方法区的实现称之为“元空间”， Meta Space。
方法区是 Java 虚拟机规范中的定义，是一种规范，而永久代和元空间是 HotSpot 虚拟机不同版本的两种实现。

Q59：是否了解类加载器双亲委派模型机制和破坏双亲委派模型?

A:

双亲委派模型机制：皇子的例子
破坏双亲委派模型：JDK1.0 的时候写好了一些类和类加载器，但是 JDK1.2 的时候才出现了双亲委派模型。比如说 DriverManager 去加载 Driver 的时候，就是破坏了双亲委派模型。
DriverManager 相当于是皇上去处理
Driver 实现类（第三方提供）相当于皇子去处理

Q60：逃逸分析有几种类型?

A:

逃逸分析 (Escape Analysis) 是目前 Java 虚拟机中比较前沿的优化技术。这是一种可以有效减少 Java 程序中同步负载和内存堆分配压力的跨函数全局数据流分析算法。通过逃逸分析，Java Hotspot 编译器能够分析出一个新的对象的引用的使用范围从而决定是否要将这个对象分配到堆上。
逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他地方中，称为方法逃逸。
逃逸分析包括：
全局变量赋值逃逸
方法返回值逃逸
实例引用发生逃逸
线程逃逸 : 赋值给类变量或可以在其他线程中访问的实例变量

Q61：-Xms 这些参数的含义是什么?

A：

堆内存分配：

JVM 初始分配的内存由 -Xms 指定，默认是物理内存的 1/64。

JVM 最大分配的内存由 -Xmx 指定，默认是物理内存的 1/4。

Q62：你知道哪几种垃圾收集器，各自的优缺点，重点讲下 cms 和 G1，包括原理，流程，优缺点。

A：

几种垃圾收集器：

Serial 收集器：单线程的收集器，收集垃圾时，必须 stoptheworld，使用复制算法。

ParNew 收集器：Serial 收集器的多线程版本，也需要 stoptheworld，复制算法。

ParallelScavenge 收集器：新生代收集器，复制算法的收集器，并发的多线程收集器，目标是达到一个可控的吞吐量。如果虚拟机总共运行 100 分钟，其中垃圾花掉 1 分钟，吞吐量就是 99%。

SerialOld 收集器：是 Serial 收集器的老年代版本，单线程收集器，使用标记整理算法。

ParallelOld 收集器：是 ParallelScavenge 收集器的老年代版本，使用多线程，标记 - 整理算法。

CMS(ConcurrentMarkSweep) 收集器：

是一种以获得最短回收停顿时间为目标的收集器，标记清除算法，运作过程：初始标记，并发标记，重新标记，并发清除，收集结束会产生大量空间碎片。

G1 收集器：标记整理算法实现，运作流程主要包括以下：初始标记，并发标记，最终标记，筛选标记。不会产生空间碎片，可以精确地控制停顿。

CMS 收集器和 G1 收集器的区别：CMS 收集器是老年代的收集器，可以配合新生代的 Serial 和 ParNew 收集器一起使用；G1 收集器收集范围是老年代和新生代，不需要结合其他收集器使用；CMS 收集器以最小的停顿时间为目标的收集器；G1 收集器可预测垃圾回收的停顿时间 CMS 收集器是使用“标记 - 清除”算法进行的垃圾回收，容易产生内存碎片 G1 收集器使用的是“标记 - 整理”算法，进行了空间整合，降低了内存空间碎片。

Q63：JVM 的内存结构，Eden 和 Survivor 比例是多少?

A：

Eden 区是一块，Survivor 区是两块。Eden 区和 Survivor 区的比例是 8：1：1。

多线程 / 高并发

Q64：负载平衡的意义什么?

A：

在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

Q65：请说出同步线程及线程调度相关的方法?

A：

1）wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
2）sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理 InterruptedException 异常；
3）notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待
状态的线程，而是由 JVM 确定唤醒哪个线程，而且与优先级无关；
4）notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；注意：java5 通过 Lock 接口提供了显示的锁机制，Lock 接口中定义了加锁（lock()）方法）和解锁（unLock() 方法），增强了多线程编程的灵活性及对线程的协调

Q66：关于 epoll 和 select 的区别，哪些说法是正确的？（多选）

A：

A. epoll 和 select 都是 I/O 多路复用的技术，都可以实现同时监听 多个 I/O 事件的状态。
B. epoll 相比 select 效率更高，主要是基于其操作系统支持的 I/O 事件通知机制，而 select 是基于轮询机制。
C. epoll 支持水平触发和边沿触发两种模式。
D. select 能并行支持 I/O 比较小，且无法修改。
A . B . C

Q67：启动一个线程是调用 run() 方法还是 start() 方法?

A：

启动一个线程是调用 start() 方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由 JVM 调度并执行，这并不意味着线程就会立即运行。run() 方法是线程启动后要进行回调（callback）的方法。

Q68：如何确保 N 个线程可以访问 N 个资源同时又不导致死锁？

A：

使用多线程的时候，一种非常简单的避免死锁的方式就是：指定获取锁的顺序，并强制线程按照指定的顺序获取锁。因此，如果所有的线程都是以同样的顺序加锁和释放锁，就不会出现死锁了。

Q69：编写多线程程序的几种实现方式（换个问法：创建多线程的方式）？

A：

（1）通过继承 Thread 类

（2）通过实现 Runnable 接口（推荐使用，因为 Java 中是单继承，一个类只有一个父类，若继承了 Thread 类，就无法在继承其它类，显然实现 Runnable 接口更为灵活

（3）通过实现 Callable 接口（Java5 之后）

Q70：线程和进程的区别？

A：

进程：具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配和调度的一个独立单位线程：是进程的一个实体，是 cpu 调度和分派的基本单位，是比进程更小的可以独立运行的基本单位

特点：1）线程的划分尺度小于进程，这使多线程程序拥有高并发性；2）进程在运行时各自内存单元相互独立，线程之间内存共享，这使多线程编程可以拥有更好的性能和用户体验。

Q71：什么是线程池，有哪些常用线程池？

A：

就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

Q72：什么是死锁？

A：

两个线程都在等待对方执行完毕才能继续往下执行的时候就发生了死锁。结果就是两个进程都陷入了无限的等待中。

Q73：怎么保证缓存和数据库数据的一致性?

A：

合理设置缓存的过期时间。

新增、更改、删除数据库操作时同步更新 Redis，可以使用事物机制来保证数据的一致性。

消息中间件

Q74：消费者获取消息有几种模式?

A：

消费者获取消息有两种模式：推送模式和拉取模式。

Q75：RocketMQ 的特点有哪些?

A：

支持严格的消息顺序支持 Topic 与 Queue 两种模式亿级消息堆积能力比较友好的分布式特性同时支持 Push 与 Pull 方式消费消息

Q76：kafka 同时设置了 7 天和 10G 清除数据，到第五天的时候消息达到了 10G，这个时候 kafka 将如何处理?

A：

这个时候 kafka 会执行数据清除工作，时间和大小不论那个满足条件，都会清空数据。

Q77：为何需要 Kafka 集群

A：

本地开发，一台 Kafka 足够使用。在实际生产中，集群可以跨服务器进行负载均衡，再则可以使用复制功能来避免单独故障造成的数据丢失。同时集群可以提供高可用性。

Q78：Kafka 数据存储设计

A：

partition 的数据文件（offset，MessageSize，data）

partition 中的每条 Message 包含了以下三个属性：offset，essageSize，data，其中 offset 表示 Message 在这个 partition 中的偏移量，offset 不是该 Message 在 partition 数据文件中的实际存储位置，而是逻辑上一个值，它唯一确定了 partition 中的一条 Message，可以认为 offset 是 partition 中 Message 的 id；MessageSize 表示消息内容 data 的大小；data 为 Message 的具体内容。

数据文件分段 segment（顺序读写、分段命令、二分查找）partition 物理上由多个 segment 文件组成，每个 segment 大小相等，顺序读写。每个 segment 数据文件以该段中最小的 offset 命名，文件扩展名为 .log。这样在查找指定 offset 的 Message 的时候，用二分查找就可以定位到该 Message 在哪个 segment 数据文件中。

数据文件索引（分段索引、稀疏存储）Kafka 为每个分段后的数据文件建立了索引文件，文件名与数据文件的名字是一样的，只是文件扩展名为 .index。index 文件中并没有为数据文件中的每条 Message 建立索引，而是采用了稀疏存储的方式，每隔一定字节的数据建立一条索引。这样避免了索引文件占用过多的空间，从而可以将索引文件保留在内存中

Q79：Kafka 如何判断一个节点是否存活?

A：

（1）节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每个节点的连接

（2）如果节点是个 follower, 他必须能及时的同步 leader 的写操作，延时不能太久

80.kafka 消息发送的可靠性机制有几种

生产者向 kafka 发送消息时，可以选择需要的可靠性级别。通过 acks 参数的值进行设置。

（1） 0 值 异步发送。生产者向 kafka 发送消息而不需要 kafka 反馈成功 ack。该方式效率最高，但 可靠性最低。其可能会存在消息丢失的情况。

（2） 1 值 同步发送，默认值。生产者发送消息给 kafka，broker 的 partition leader 在收到消息后 马上发送成功 ack（无需等待 ISR 中的 follower 同步完成），生产者收到后知道消息发送成功， 然后会再发送消息。如果一直未收到 kafka 的 ack，则生产者会认为消息发送失败，会重发消息。

（3） -1 值 同步发送。其值等同于 all。生产者发送消息给 kafka，kafka 收到消息后要等到 ISR 列表中的所有副本都同步消息完成后，才向生产者发送成功 ack。如果一直未收到 kafka 的 ack， 则认为消息发送失败，会自动重发消息。

该方式存在 follower 重复接收的情况。注意，重复接收，与重复消费是两个概念

Q80：请详细说一下推送模式和拉取模式。

A：

1. PushConsumer

推送模式（虽然 RocketMQ 使用的是长轮询）的消费者。消息的能及时被消费。使用非常简单，内部已处理如线程池消费、流控、负载均衡、异常处理等等的各种场景。

2. PullConsumer

拉取模式的消费者。应用主动控制拉取的时机，怎么拉取，怎么消费等。主动权更高。但要自己处理各种场景。

Q81：Kafka 与传统消息系统之间有三个关键区别

A：

(1).Kafka 持久化日志，这些日志可以被重复读取和无限期保留

(2).Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性

(3).Kafka 支持实时的流式处理

Q82：RocketMQ 由哪些角色组成?

A：

生产者（Producer）：负责产生消息，生产者向消息服务器发送由业务应用程序系统生成的消息。

消费者（Consumer）：负责消费消息，消费者从消息服务器拉取信息并将其输入用户应用程序。

消息服务器（Broker）：是消息存储中心，主要作用是接收来自 Producer 的消息并存储，Consumer 从这里取得消息。

名称服务器（NameServer）：用来保存 Broker 相关 Topic 等元信息并给 Producer ，提供 Consumer 查找 Broker 信息。

Q83：Kafka 的消费者如何消费数据

A：

答：消费者每次消费数据的时候，消费者都会记录消费的物理偏移量（offset）的位置等到下次消费时，他会接着上次位置继续消费

Q84：Kafka 的优点

A：

多生产者和多消费者

基于磁盘的数据存储，换句话说，Kafka 的数据天生就是持久化的。

高伸缩性，Kafka 一开始就被设计成一个具有灵活伸缩性的系统，对在线集群的伸缩丝毫不影响整体系统的可用性。

高性能，结合横向扩展生产者、消费者和 broker，Kafka 可以轻松处理巨大的信息流，同时保证亚秒级的消息延迟。

Q85：Kafka 的设计时什么样的呢?

A：

Kafka 将消息以 topic 为单位进行归纳
将向 Kafka topic 发布消息的程序成为 producers.
将预订 topics 并消费消息的程序成为 consumer.
Kafka 以集群的方式运行，可以由一个或多个服务组成，每个服务叫做一个 broker.
producers 通过网络将消息发送到 Kafka 集群，集群向消费者提供消息

Q86：说说你对 Consumer 的了解?

A：

1、获得 Topic-Broker 的映射关系。Consumer 启动时需要指定 Namesrv 地址，与其中一个 Namesrv 建立长连接。消费者每隔 30 秒从 Namesrv 获取所有 Topic 的最新队列情况，Consumer 跟 Broker 是长连接，会每隔 30 秒发心跳信息到 Broker.
2、消费者端的负载均衡。根据消费者的消费模式不同，负载均衡方式也不同。

Q87：Kafka 新建的分区会在哪个目录下创建

A：

在启动 Kafka 集群之前，我们需要配置好 log.dirs 参数，其值是 Kafka 数据的存放目录，这个参数可以配置多个目录，目录之间使用逗号分隔，通常这些目录是分布在不同的磁盘上用于提高读写性能。当然我们也可以配置 log.dir 参数，含义一样。只需要设置其中一个即可。如果 log.dirs 参数只配置了一个目录，那么分配到各个 Broker 上的分区肯定只能在这个目录下创建文件夹用于存放数据。但是如果 log.dirs 参数配置了多个目录，那么 Kafka 会在哪个文件夹中创建分区目录呢?
答案是：Kafka 会在含有分区目录最少的文件夹中创建新的分区目录，分区目录名为 Topic 名 + 分区 ID。注意，是分区文件夹总数最少的目录，而不是磁盘使用量最少的目录！也就是说，如果你给 log.dirs 参数新增了一个新的磁盘，新的分区目录肯定是先在这个新的磁盘上创建直到这个新的磁盘目录拥有的分区目录不是最少为止。

Q88：说一下 kafka 消费者消费过程

A：

生产者将消息发送到 topic 中，消费者即可对其进行消费，其消费过程如下：
1) consumer 向 broker 集群提交连接请求，其所连接上的任意 broker 都会向其发送 broker
controller 的通信 URL，即 broker controller 主机配置文件中的 listeners 地址

2) 当 consumer 指定了要消费的 topic 后，其会向 broker controller 发送 poll 请求
3) broker controller 会为 consumer 分配一个或几个 partition leader，并将该 partitioin 的当 前 offset 发送给 consumer
4) consumer 会按照 broker controller 分配的 partition 对其中的消息进行消费
5) 当消费者消费完该条消息后，消费者会向 broker 发送一个该消息已被消费的反馈，即 该消息的 offset
若为手动提交：可以是消费完一条消息就提交一个 offset，也可以是消费完这一批消息 后，提交最后一个消息的 offset。关键看代码怎么写。
若为自动提交：提交最后一个消息的 offset。
6) 当 broker 接到消费者的 offset 后，会更新到相应的 _consumeroffset 中
7) 以上过程一直重复，直到消费者停止请求消息
8) 消费者可以重置 offset，从而可以灵活消费存储在 broker 上的消息

Q89：介绍下 Kafka

A：

答：Kafka 是一种高吞吐量、分布式、基于发布 / 订阅的消息系统，最初由 LinkedIn 公司开发，使用 Scala 语言编写，目前是 Apache 的开源项目。
broker：Kafka 服务器，负责消息存储和转发
topic：消息类别，Kafka 按照 topic 来分类消息
partition：topic 的分区，一个 topic 可以包含多个 partition，topic 消息保存在各个 partition 上
offset：消息在日志中的位置，可以理解是消息在 partition 上的偏移量，也是代表该消息的唯一序号
Producer：消息生产者
Consumer：消息消费者
ConsumerGroup：消费者分组，每个 Consumer 必须属于一个 group
Zookeeper：保存着集群 broker、topic、partition 等 meta 数据；另外，还负责 broker 故障发现，partitionleader 选举，负载均衡等功能

Q90：什么情况会导致 kafka 运行变慢?

A：

cpu 性能瓶颈
磁盘读写瓶颈
网络瓶颈

Spring Cloud°

Q91：你曾阅读过 Spring Cloud 的源码吗?

A：

我们知道，Spring Cloud 是通过 Spring Boot 集成了很多第三方框架构成的。现在准备解析
Spring Cloud 中某子框架的源码，若还没有找到合适的入手位置，那么从哪里开始解析可能是一个不错的选择?
我自己曾阅读过 Spring Cloud 中的 Eureka、OpenFeign、Ribbon 等源码。对于一个未曾阅读过的子框架源码，我认为从自动配置类开始解析可能是一个不错的选择。
我们知道 Spring Cloud 是通过 Spring Boot 将其它第三方框架集成进来的。Spring Boot 最大的特点就是自动配置，我们可以通过导入相关 Starter 来实现需求功能的自动配置、相关核心业务类实例的创建等。也就是说，核心业务类都是集中在自动配置类中的。所以从这里下手分析应该是个不错的选择。
那么从哪里可以找到这个自动配置类呢? 从导入的 starter 依赖工程的 META-INF 目录中的 spring.factory 文件中可以找到。该文件的内容为 key-value 对，查找 EnableAutoConfiguration 的全限定性类名作为 key 的 value，这个 value 就是我们要找到的自动配置类。

Q92：@EnableConfigurationProperties 注解对于 Starter 的定义很重要，请谈一上你对这个注解的认识。

A：

@EnableConfigurationProperties 注解在 Starter 定义时主要用于读取 application.yml 配置文件中相关的属性，并封装到指定类型的实例中，以备 Starter 中的核心业务实例使用。
具体来说，它就是开启了对 @ConfigurationProperties 注解的 Bean 的自动注册，注解到 Spring 容器中。这种 Bean 有两种注册方式：在配置类使用 @Bean 方法注册，或直接使用该注解的 value 属性进行注册。若在配置类中使用 @Bean 注册，则需要在配置类中定义一个 @Bean 方法，该方法的返回值为“使用 @ConfigurationProperties 注解标注”的类。若直接使用该注解的 value 属性进行注册，则需要将这个“使用 @ConfigurationProperties 注解标注”的类作为 value 属性值出现即可。

Q93：Spring Boot 中定义了很多条件注解，这些注解一般用于对配置类的控制。在这些条件注解中有一个@ConditionalOnMissingBean 注解，你了解过嘛？请谈一下你对它的认识。

A：

@ConditionalOnMissingBean 注解是 Spring Boot 提供的众多条件注册中的一个。其表示的意义是，当容器中没有指定名称或指定类型的 Bean 时，该条件为 true。不过，这里需要强调一点的是，这里要查找的“容器”是可以指定的。通过 search 属性指定。其 search 的范围有三种：仅搜索当前配置类容器；搜索所有层次的父类容器，但不包含当前配置类容器；搜索当前配置类容器及其所有层次的父类容器，这个是默认搜索范围。

Q94：Spring Cloud 中默认情况下对于 Eureka Client 实例的创建中，@RefreshScope 注解是比较重要的，请谈一下你对这个注解的认识。

A：

@RefreshScope 注解是 Spring Cloud 中定义的一个注解。该注解用于配置类，可以添加在配置类上，也可以添加在 @Bean 方法上。其表示的意思是，该 @Bean 方法会以多例的形式生成会自动刷新的 Bean 实例。这种方式就等价于在 Spring 的 xml 配置文件中指定标签的 scope 属性值为 refresh。当然，若一个配置类上添加了该注解，则表示该配置类中的所有 @Bean 方法创建的实例都是 @RefreshScope 的。

Q95：Spring Cloud 中默认情况下对于 Eureka Client 实例的创建是在 EurekaClient 的自动配置类中通过 @Bean 方法完成的。但在源码中，这个 @Bean 方法上同时出现了 @RefreshScope、@ConditionalOnMissionBean，与 @Lazy 注解，从这些注解的意义来分析，是否存在矛盾呢？它们联合使用又是什么意思呢？请谈一下你的看法。

A：

首先来说，这三个注解的意义都是比较复杂的。

@RefreshScope 注解是 Spring Cloud 中定义的一个注解。其表示的意思是，该 @Bean 方法会以多例的形式生成会自动刷新的 Bean 实例。

@ConditionalOnMissionBean 注解表示的意思是，只有当容器中没有 @Bean 要创建的实例时才会创建新的实例，即这里创建的 @Bean 实例是单例的。

@Lazy 注解表示延迟实例化。即在当前配置类被实例化时并不会调用这里的 @Bean 方法去

创建实例，而是在代码执行过程中，真正需要这个 @Bean 方法的实例时才会创建。
这三个注解的联用不存在矛盾，其要表达的意思是，这个 @Bean 会以延迟实例化的形式创建一个单例的对象，而该对象具有自动刷新功能。

Q96：Spring Cloud 中默认情况下对于 Eureka Client 实例的创建是在 EurekaClient 的自动配置类中通过 @Bean 方法完成的。但在源码中，这个 @Bean 方法上同时出现了 @RefreshScope、@ConditionalOnMissionBean，与 @Lazy 注解，从这些注解的意义来分析，是否存在矛盾呢？它们联合使用又是什么意思呢？请谈一下你的看法。

A：

首先来说，这三个注解的意义都是比较复杂的。
@RefreshScope 注解是 Spring Cloud 中定义的一个注解。其表示的意思是，该 @Bean 方法会以多例的形式生成会自动刷新的 Bean 实例。
@ConditionalOnMissionBean 注解表示的意思是，只有当容器中没有 @Bean 要创建的实例时才会创建新的实例，即这里创建的 @Bean 实例是单例的。
@Lazy 注解表示延迟实例化。即在当前配置类被实例化时并不会调用这里的 @Bean 方法去创建实例，而是在代码执行过程中，真正需要这个 @Bean 方法的实例时才会创建。
这三个注解的联用不存在矛盾，其要表达的意思是，这个 @Bean 会以延迟实例化的形式创建一个单例的对象，而该对象具有自动刷新功能。

Q97：Spring Cloud 中大量地使用了条件注解，其中 @ConditionalOnRefreshScope 注解对于 Eureka Client 的创建非常重要。请谈一下你对这个注解的认识。

A：

首先，关于条件注解，实际是 Spring Boot 中出现的内容，其一般应用于配置类中。表示只有当该条件满足时才会创建该实例。而您提到的 @ConditionalOnRefreshScope 注解，其实际是 Eureka Client 的自动配置类中的一个内部注解。该注解不同于 Spring Boot 中的一般性注解的是，其是一个复合条件注
解，其复合的条件有三个：
在当前类路径下具有 RefreshScope 类
在容器中要具有 RefreshAutoConfiguration 类的实例指定的 eureka.client.refresh.enable 属性值为 true。不过，其缺省值就是 true。这也就是为什么我们的配置文件默认支持自动更新的原因。
只有当这个复合注解中的三个条件均成立时，@ConditionalOnRefreshScope 注解才满足

条件。此时才有可能会调用创建 Eureka Client 的 @Bean 方法。所以，该注解对于 Eureka

Q98：你刚才已经谈过了对 @ConditionalOnRefreshScope 注解的认识，非常不错。不过，与这个注解相对应的另一个注解 @ConditionalOnMissingRefreshScope，你是否了解？若关注过，谈一下你的认识。

A：

@ConditionalOnMissingRefreshScope 我也曾了解过。这个注解就像 @ConditionalOnRefreshScope 注解一样，也是一个复合条件注解，其也包含了三个条件。不同的是，这个注解中的条件是或的关系，只要满足其中一条这个注解就匹配上了。而 @ConditionalOnRefreshScope 注解中的三个条件是与的关系，必须所有条件均满足其才能匹配上。

这个或的关系是通过让一个复合条件类继承自一个能够表示或关系的复合条件父类 AnyNestedCondition 实现的。这样的话，这个复合条件类中定义的多个内部条件类中，只要有一个匹配上，那么这个复合条件类就算匹配上了。

Q99：Spring Cloud 中 Eureka Client 的源码中有一个非常重要的类 Applications，其被称为客户端注册表。请谈一下你对它的认识。

A：

Applications 类实例中封装了来自于 Eureka Server 的所有注册信息，通常称其为“客户端注册表”。只所以要强调“客户端”是因为，服务端的注册表不是这样表示的，是一个 Map。

该类中封装着一个非常重要的 Map 集合，key 为微服务名称，而 Value 则为 Application 实例。Application 类中封装了一个 Set 集合，集合元素为“可以提供该微服务的所有主机的 InstanceInfo”。也就是说，Applications 中封装着所有微服务的所有提供者信息。

Q100：Spring Cloud 中 Eureka Client 与 Eureka Server 的通信，及 Eureka Server 间的通信是如何实现的？请简单介绍一下。

A：

Spring Cloud 中 Eureka Client 与 Eureka Server 的通信，及 Eureka Server 间的通信，均采用的是 Jersey 框架。

Jersey 框架是一个开源的 RESTful 框架，实现了 JAX-RS 规范。该框架的作用与 SpringMVC 是相同的，其也是用户提交 URI 后，在处理器中进行路由匹配，路由到指定的后台业务。这个路由功能同样也是通过处理器完成的，只不过这里的处理器不叫 Controller，而叫 Resource

Kaikeba
开课吧

关注公众号
了解更多内容