

微服务条目	落地的技术
服务开发	SpringBoot、Spring、SpringMVC
服务配置管理	Netflix公司的Archaius、阿里的Diamond等
服务注册与发现	Eureka、Consul、Zookeeper
服务调用	RPC、Rest、gRPC
服务熔断器	Hystrix、Envoy等
负载均衡	Nginx、Ribbon
服务接口调用（客户端调用服务的简化工具）	Feign
消息队列	Kafka、RabbitMQ、ActiveMQ等
服务配置中心配置管理	SpringCloudConfig、Chaos等
服务路由（API网关）	Zuul
服务监控	Zabbix、Nagios、Metrics、Spectator等
全链路追踪	Zipkin、Brave、Dapper等
服务部署	Docker、OpenStack、Kubernetes等
数据流操作开发包	SpringCloud Stream
事件消息总线	Spring Cloud Bus

35、Eureka和zookeeper都可以提供服务注册与发现的功能，请说说两个的区别？

Zookeeper保证了CP（C：一致性，P：分区容错性），Eureka保证了AP（A：高可用）

1.当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的信息，但不能容忍直接down掉不可用。也就是说，服务注册功能对高可用性要求比较高，但zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新选leader。问题在于，选取leader时间过长，30~120s，且选取期间zk集群都不可用，这样就会导致选取期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够恢复，但是漫长的选取时间导致的注册长期不可用是不能容忍的。

2.Eureka保证了可用性，Eureka各个节点是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点仍然可以提供注册和查询服务。而Eureka的客户端向某个Eureka注册或发现时发生连接失败，则会自动切换到其他节点，只要有一台Eureka还在，就能保证注册服务可用，只是查到的信息可能不是最新的。除此之外，Eureka还有自我保护机制，如果在15分钟内超过85%的节点没有正常的心跳，那么Eureka就认为客户端与注册中心发生了网络故障，此时会出现以下几种情况：

- ①、Eureka不在从注册列表中移除因为长时间没有收到心跳而应该过期的服务。
- ②、Eureka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他节点上（即保证当前节点仍然可用）
- ③、当网络稳定时，当前实例新的注册信息会被同步到其他节点。

因此，Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像Zookeeper那样使整个微服务瘫痪。

RabbitMQ面试题

1、什么是rabbitmq

采用AMQP高级消息队列协议的一种消息队列技术，最大的特点就是消费并不需要确保提供方存在，实现了服务之间的高度解耦。

2、为什么要使用rabbitmq

- 1、在分布式系统下具备异步、削峰、负载均衡等一系列高级功能；
- 2、拥有持久化的机制，进程消息，队列中的信息也可以保存下来。
- 3、实现消费者和生产者之间的解耦。
- 4、对于高并发场景下，利用消息队列可以使得同步访问变为串行访问达到一定量的限流，利于数据库的操作。

5. 可以使用消息队列达到异步下单的效果，排队中，后台进行逻辑下单。

3、使用 rabbitmq 的场景

- 1、服务间异步通信
- 2、顺序消费
- 3、定时任务
- 4、请求削峰

4、如何确保消息正确地发送至 RabbitMQ？如何确保消息接收方消费了消息？

发送方确认模式

将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。

一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。

如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack (not acknowledged, 未确认) 消息。发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

接收方确认机制

接收方消息确认机制

消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。这里并没有用到超时机制，RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。保证数据的最终一致性；

下面罗列几种特殊情况

如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。

（可能存在消息重复消费的隐患，需要去重）如果消费者接收到消息却没有确认消息，连接也未断开，则 RabbitMQ 认为该消费者繁忙，将不会给该消费者分发更多的消息。

5、如何避免消息重复投递或重复消费？

在消息生产时，MQ 内部针对每条生产者发送的消息生成一个 inner-msg-id，作为去重的依据（消息投递失败并重传），避免重复的消息进入队列；

在消息消费时，要求消息体中必须要有一个 bizId（对于同一业务全局唯一，如支付 ID、订单 ID、帖子 ID 等）作为去重的依据，避免同一条消息被重复消费。

6、消息基于什么传输？

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制

7、消息如何分发？

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。

通过路由可实现多消费的功能

8、消息怎么路由？

消息提供方->路由->一至多个队列

消息发布到交换器时，消息将拥有一个路由键（routing key），在消息创建时设定。

通过队列路由键，可以把队列绑定到交换器上。

消息到达交换器后，RabbitMQ 会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）；

常用的交换器主要分为一下三种

fanout：如果交换器收到消息，将会广播到所有绑定的队列上

direct：如果路由键完全匹配，消息就被投递到相应的队列

topic：可以使来自不同源头的消息能够到达同一个队列。使用 topic 交换器时，可以使用通配符

9、如何确保消息不丢失？

消息持久化，当然前提是队列必须持久化

RabbitMQ 确保持久性消息能从服务器重启中恢复的方式是，将它们写入磁盘上的一个持久化日志文件，当发布一条持久性消息到持久交换器上时，Rabbit 会在消息提交到日志文件后才发送响应。

一旦消费者从持久队列中消费了一条持久化消息，RabbitMQ 会在持久化日志中把这条消息标记为等待垃圾收集。如果持久化消息在被消费之前 RabbitMQ 重启，那么 Rabbit 会自动重建交换器和队列（以及绑定），并重新发布持久化日志文件中的消息到合适的队列。

10、使用 RabbitMQ 有什么好处？

- 1、服务间高度解耦
- 2、异步通信性能高
- 3、流量削峰

11、RabbitMQ 的集群

镜像集群模式

你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，然后每次你写消息到 queue 的时候，都会自动把消息到多个实例的 queue 里进行消息同步。

好处在于，你任何一个机器宕机了，没事儿，别的机器都可以用。坏处在于，第一，这个性能开销也太大了吧，消息同步所有机器，导致网络带宽压力和消耗很重！第二，这么玩儿，就没有扩展性可言了，如果某个 queue 负载很重，你加机器，新增的机器也包含了这个 queue 的所有数据，并没有办法线性扩展你的 queue

12、mq 的缺点

系统可用性降低

系统引入的外部依赖越多，越容易挂掉，本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整？MQ 挂了，整套系统崩溃了，你不就完了么。

系统复杂性提高

硬生生加个 MQ 进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？头大头大，问题一大堆，痛苦不已

一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉，最好之后，你会发现，妈呀，系统复杂度提升了一个数量级，也许是复杂了 10 倍。但是关键时刻，用，还是得用的

13、Kafka、ActiveMQ、RabbitMQ、RocketMQ 都有什么区别？

对于吞吐量来说 kafka 和 RocketMQ 支撑高吞吐，ActiveMQ 和 RabbitMQ 比他们低一个数量级。对于延迟量来说 RabbitMQ 是最低的。

1.从社区活跃度

按照目前网络上的资料，RabbitMQ、activeM、ZeroMQ 三者中，综合来看，RabbitMQ 是首选。

2.持久化消息比较

ActiveMq 和 RabbitMq 都支持。持久化消息主要是指我们机器在不可抗力因素等情况下挂掉了，消息不会丢失的机制。

3.综合技术实现

可靠性、灵活的路由、集群、事务、高可用的队列、消息排序、问题追踪、可视化管理工具、插件系统等等。

RabbitMq / Kafka 最好，ActiveMq 次之，ZeroMq 最差。当然 ZeroMq 也可以做到，不过自己必须手动写代码实现，代码量不小。尤其是可靠性中的：持久性、投递确认、发布者证实和高可用性。

4.高并发

毋庸置疑，RabbitMQ 最高，原因是它的实现语言是天生具备高并发高可用的 erlang 语言。

5.比较关注的比较，RabbitMQ 和 Kafka

RabbitMq 比 Kafka 成熟，在可用性上，稳定性上，可靠性上，RabbitMq 胜于 Kafka（理论上）。另外，Kafka 的定位主要在日志等方面，因为 Kafka 设计的初衷就是处理日志的，可以看做一个日志（消息）系统一个重要组件，针对性很强，所以如果业务方面还是建议选择 RabbitMq。还有就是，Kafka 的性能（吞吐量、TPS）比 RabbitMq 要高出来很多

14、如何保证高可用的？

RabbitMQ 是比较有代表性的，因为是基于主从（非分布式）做高可用性的，我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。

单机模式，就是 Demo 级别的，一般就是你本地启动了玩玩儿的？没人生产用单机模式普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读写操作。

镜像集群模式：这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！RabbitMQ 一个 queue 的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个 queue 的完整数据

Kafka 一个最基本的架构认识：由多个 broker 组成，每个 broker 是一个节点；你创建一个 topic，这个 topic 可以划分为多个 partition，每个 partition 可以存在于不同的 broker 上，每个 partition 就放一部分数据。这就是天然的分布式消息队列，就是说一个 topic 的数据，是分散放在多个机器上的，每个机器就放一部分数据。Kafka 0.8 以后，提供了 HA 机制，就是 replica（复制品）副本机制。每个 partition 的数据都会同步到其它机器上，形成自己的多个 replica 副本。所有 replica 会选举一个 leader 出来，那么生产和消费都跟这个 leader 打交道，然后其他 replica 就是 follower。写的时候，leader 会负责把数据同步到所有 follower 上去，读的时候就直接读 leader 上的数据即可。只能读写 leader？很简单，要是你可以随意读写每个 follower，那么就要 care 数据一致性的问题，系统复杂度太高，很容易出问题。

Kafka 会均匀地将一个 partition 的所有 replica 分布在不同的机器上，这样才可以提高容错性。因为如果某个 broker 宕机了，没事儿，那个 broker 上面的 partition 在其他机器上都

有副本的，如果这上面有某个 partition 的 leader，那么此时会从 follower 中重新选举一个新的 leader 出来，大家继续读写那个新的 leader 即可。这就是所谓的高可用性了。写数据的时候，生产者就写 leader，然后 leader 将数据落地写本地磁盘，接着其他 follower 自己主动从 leader 来 pull 数据。一旦所有 follower 同步好数据了，就会发送 ack 给 leader，leader 收到所有 follower 的 ack 之后，就会返回成功的消息给生产者。（当然，这只是其中一种模式，还可以适当调整这个行为）消费的时候，只会从 leader 去读，但是只有当一个消息已经被所有 follower 都同步成功返回 ack 的时候，这个消息才会被消费者读到

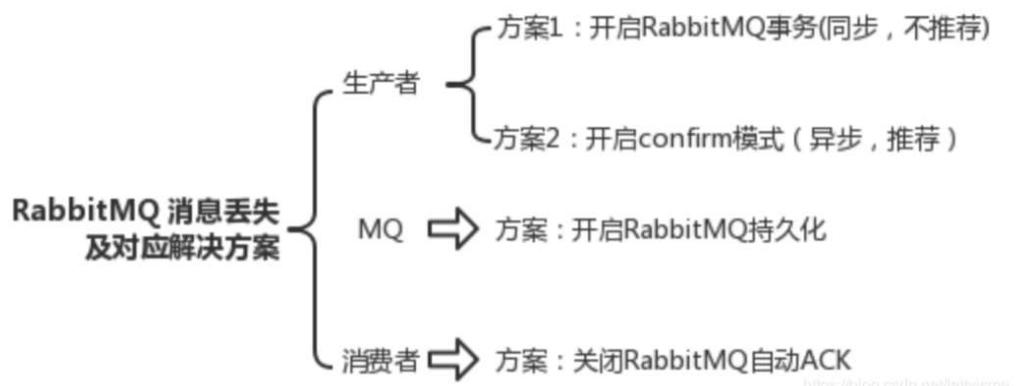
15、如何保证消息的可靠传输？如果消息丢了怎么办

数据的丢失问题，可能出现在生产者、MQ、消费者中

生产者丢失：生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络问题啥的，都有可能。此时可以选择用 RabbitMQ 提供的事务功能，就是生产者发送数据之前开启 RabbitMQ 事务 channel.txSelect，然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务 channel.txRollback，然后重试发送消息；如果收到了消息，那么可以提交事务 channel.txCommit。吞吐量会下来，因为太耗性能。所以一般来说，如果你要确保说写 RabbitMQ 的消息别丢，可以开启 confirm 模式，在生产者那里设置开启 confirm 模式之后，你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中，RabbitMQ 会给你回传一个 ack 消息，告诉你说这个消息 ok 了。如果 RabbitMQ 未能处理这个消息，会回调你一个 nack 接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。事务机制和 noconfirm 机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是 confirm 机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息 RabbitMQ 接收了之后会异步回调你一个接口通知你这个消息接收到。所以一般在生产者这块避免数据丢失，都是用 confirm 机制的

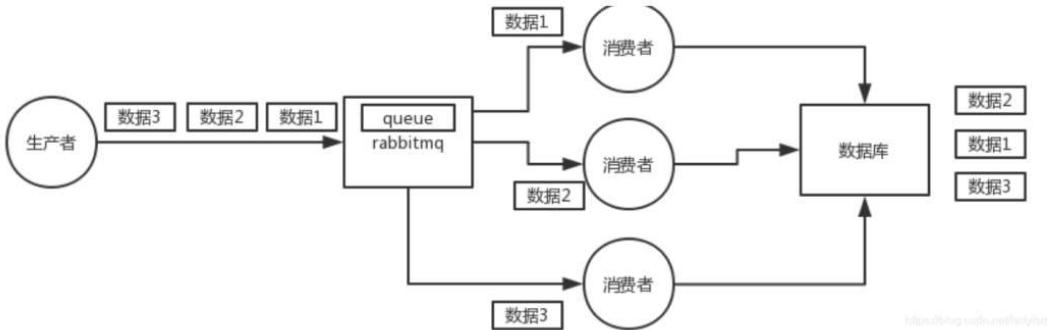
MQ 中丢失：就是 RabbitMQ 自己弄丢了数据，这个你必须开启 RabbitMQ 的持久化，就是消息写入之后会持久化到磁盘，哪怕是 RabbitMQ 自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。设置持久化有两个步骤：创建 queue 的时候将其设置为持久化，这样就可以保证 RabbitMQ 持久化 queue 的元数据，但是不会持久化 queue 里的数据。第二个是发送消息的时候将消息的 deliveryMode 设置为 2，就是将消息设置为持久化的，此时 RabbitMQ 就会将消息持久化到磁盘上去。必须要同时设置这两个持久化才行，RabbitMQ 哪怕是挂了，再次重启，也会从磁盘上重启恢复 queue，恢复这个 queue 里的数据。持久化可以跟生产者那边的 confirm 机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者 ack 了，所以哪怕是在持久化到磁盘之前，RabbitMQ 挂了，数据丢了，生产者收不到 ack，你也是可以自己重发的。注意，哪怕是你给 RabbitMQ 开启了持久化机制，也有一种可能，就是这个消息写到了 RabbitMQ 中，但是还没来得及持久化到磁盘上，结果不巧，此时 RabbitMQ 挂了，就会导致内存里的一点点数据丢失

消费端丢失：你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。这个时候得用 RabbitMQ 提供的 ack 机制，简单来说，就是你关闭 RabbitMQ 的自动 ack，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再在程序里 ack 一把。这样的话，如果你还没处理完，就不就没有 ack？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的



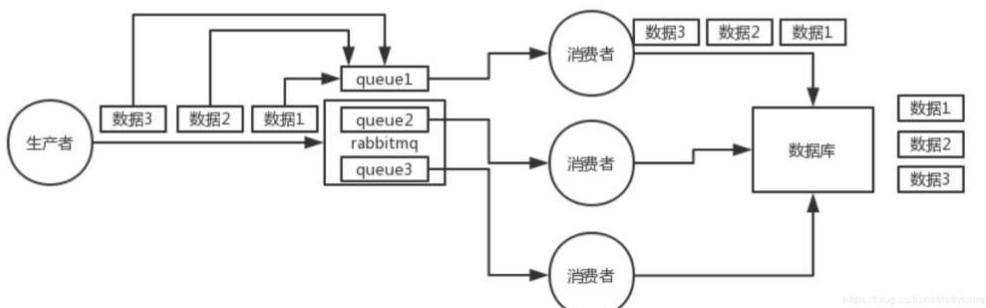
16、如何保证消息的顺序性

先看看顺序会错乱的场景：RabbitMQ：一个 queue，多个 consumer，这不明显乱了；



解决：

拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。



17、如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决

消息积压处理办法：临时紧急扩容：

先修复 consumer 的问题，确保其恢复消费速度，然后将现有 consumer 都停掉。新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。

接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息。MQ 中消息失效：假设你用的是 RabbitMQ，RabbitMQ 是可以设置过期时间的，也就是 TTL。如果消息在 queue 中积压超过一定的时间就会被 RabbitMQ 给清理掉，这个数据就没了。这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是大量的数据会直接丢弃。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上 12 点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。假设 1 万个订单积压在 mq 里面，没有处理，其中 1000 个订单都丢了，你只能手动写程序把那 1000 个订单给查出来，手动发到 mq 里去再补一次。

mq 消息队列块满了：如果消息积压在 mq 里，你很长时间都没有处理掉，此时导致 mq 都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

18、设计MQ的思路

比如说这个消息队列系统，我们从以下几个角度来考虑一下：

首先这个 mq 得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下 kafka 的设计理念，broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。如果现在资源不够了，简单啊，给 topic 增加 partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

其次你得考虑一下这个 mq 的数据要不要落地磁盘吧？那肯定要了，落磁盘才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是 kafka 的思路。

其次你考虑一下你的 mq 的可用性啊？这个事儿，具体参考之前可用性那个环节讲解的 kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。能不能支持数据 0 丢失啊？可以的，参考我们之前说的那个 kafka 数据零丢失方案。

19、什么是Message？

消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括 routing-key (路由键) 、 priority (相对于其他消息的优先权) 、 delivery-mode (指出该消息可能需要持久性存储) 等。

20、什么是Publisher ?

消息的生产者，也是一个向交换器发布消息的客户端应用程序。

21、什么是Exchange (将消息路由给队列)

交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列

22、什么是Binding (消息队列和交换器之间的关联)

绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表

23、什么是Queue ?

消息队列，用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走

24、什么是Connection ?

网络连接，比如一个 TCP 连接。

25、什么是Channel ?

信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的 TCP 连接内地虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接

26、什么是Consumer ?

消息的消费者，表示一个从消息队列中取得消息的客户端应用程序

27、什么是Virtual Host ?

虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。

28、什么是Broker ?

表示消息队列服务器实体

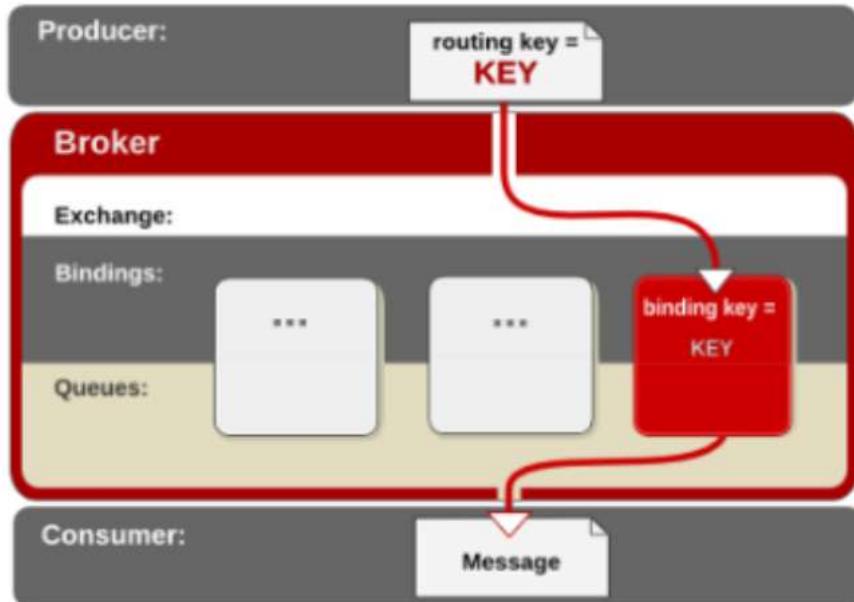
29、Exchange 类型 ?

Exchange 分发消息时根据类型的不同分发策略有区别，目前共四种类型：direct、fanout、topic、headers。headers 匹配 AMQP 消息的 header 而不是路由键，此外 headers 交换器和 direct 交换器完全一致，但性能差很多，目前几乎用不到了。

30、Direct 键 (routing key) 分布 ?

Direct：消息中的路由键 (routing key) 如果和 Binding 中的 binding key 一致，交换器就将消息发到对应的队列中。它是完全匹配、单播的模式。

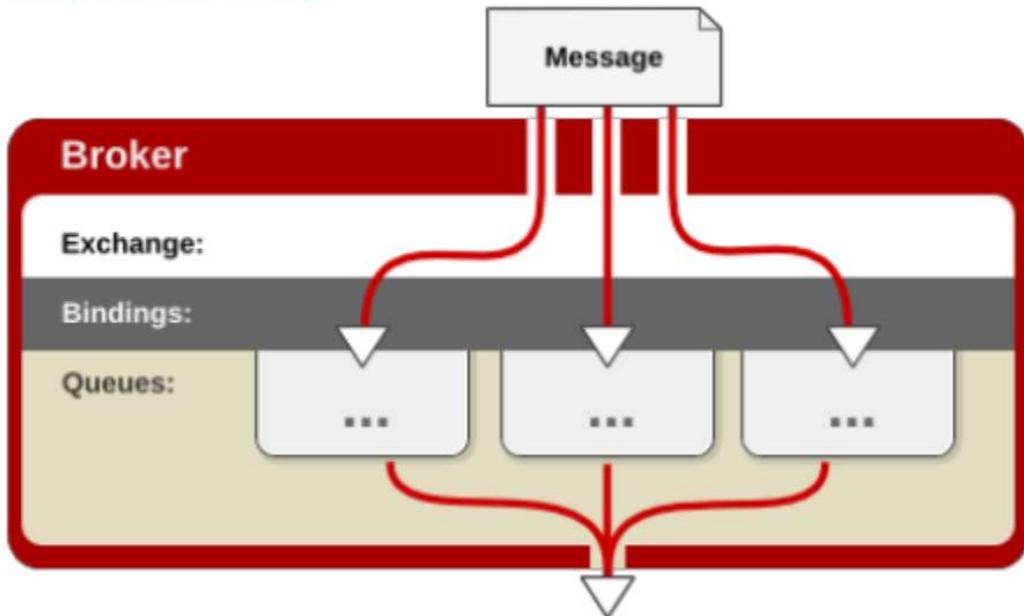
Direct Exchange



31、Fanout (广播分发) ?

Fanout : 每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去。很像子网广播，每台子网内的主机都获得了一份复制的消息。fanout 类型转发消息是最快的。

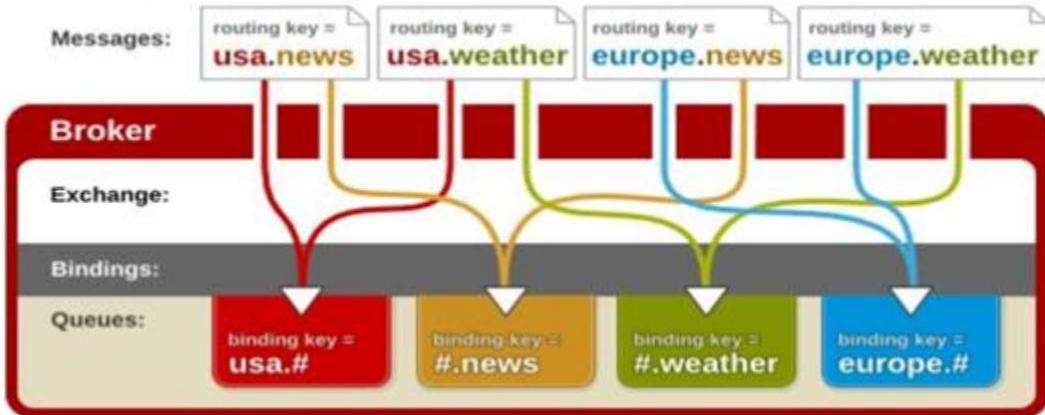
Fanout Exchange



32、topic 交换器 (模式匹配) ?

topic 交换器 : topic 交换器通过模式匹配分配消息的路由键属性，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。它将路由键和绑定键的字符串切分成单词，这些单词之间用点隔开。它同样也会识别两个通配符：符号“#”和符号“”。#匹配 0 个或多个单词，匹配不多不少一个单词。

Topic Exchange



Dubbo 面试题

1、为什么要用 Dubbo ?

随着服务化的进一步发展，服务越来越多，服务之间的调用和依赖关系也越来越复杂，诞生了面向服务的架构体系(SOA)，也因此衍生出了一系列相应的技术，如对服务提供、服务调用、连接处理、通信协议、序列化方式、服务发现、服务路由、日志输出等行为进行封装的服务框架。就这样为分布式系统的服务治理框架就出现了，Dubbo 也就这样产生了。

2、Dubbo 的整体架构设计有哪些分层？

接口服务层 (Service)：该层与业务逻辑相关，根据 provider 和 consumer 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 和服务端的 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和发现，以服务 URL 为中心，扩展接口为 RegistryFactory、Registry、RegistryService

路由层 (Cluster)：封装多个提供者的路由和负载均衡，并桥接注册中心，以Invoker 为中心，扩展接口为 Cluster、Directory、Router 和 LoadBlancce

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)：封装 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，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

3、默认使用的是什么通信框架，还有别的选择吗？

默认也推荐使用 netty 框架，还有 mina

4、服务调用是阻塞的吗？

默认是阻塞的，可以异步调用，没有返回值的可以这么做。
Dubbo 是基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小，异步调用会返回一个 Future 对象。

5、一般使用什么注册中心？还有别的选择吗？

推荐使用 Zookeeper 作为注册中心，还有 Redis、Multicast、Simple 注册中心，但不推荐。

6、默认使用什么序列化框架，你知道的还有哪些？

推荐使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。

7、服务提供者能实现失效踢出是什么原理？

服务失效踢出基于 zookeeper 的临时节点原理。

8、服务上线怎么不影响旧版本？

采用多版本开发，不影响旧版本。

9、如何解决服务调用链过长的问题？

可以结合 zipkin 实现分布式服务追踪。

10、说说核心的配置有哪些？

配置	配置说明
dubbo:service	服务配置
dubbo:reference	引用配置
dubbo:protocol	协议配置
dubbo:application	应用配置
dubbo:module	模块配置
dubbo:registry	注册中心配置
dubbo:monitor	监控中心配置
dubbo:provider	提供方配置
dubbo:consumer	消费方配置
dubbo:method	方法配置
dubbo:argument	参数配置

11、Dubbo 推荐用什么协议？

- dubbo:// (推荐)
- rmi://
- hessian://
- http://
- webservice://
- thrift://
- memcached://
- redis://
- rest://

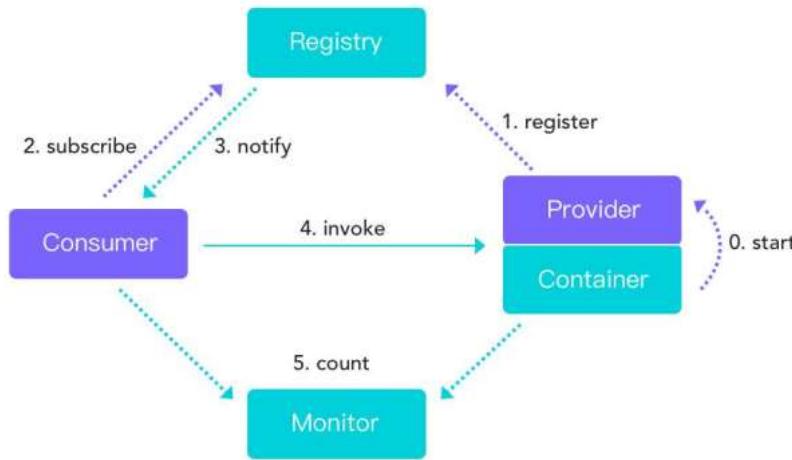
12、同一个服务多个注册的情况下可以直连某一个服务吗？

可以点对点直连，修改配置即可，也可以通过 telnet 直接某个服务。

13、画一画服务注册与发现的流程图？

Dubbo Architecture

..... init > async —> sync



14、Dubbo 集群容错有几种方案？

集群容错方案	说明
Failover Cluster	失败自动切换，自动重试其它服务器（默认）
Faileast Cluster	快速失败，立即报错，只发起一次调用
Failsafe Cluster	失败安全，出现异常时，直接忽略
Failback Cluster	失败自动恢复，记录失败请求，定时重发
Forking Cluster	并行调用多个服务器，只要一个成功即返回
Broadcast Cluster	广播逐个调用所有提供者，任意一个报错则报错

15、Dubbo 服务降级，失败重试怎么做？

可以通过 dubbo:reference 中设置 mock="return null"。mock 的值也可以修改为 true，然后再跟接口同一个路径下实现一个 Mock 类，命名规则是“接口名称+Mock”后缀。然后在 Mock 类里实现自己的降级逻辑

16、Dubbo 使用过程中都遇到了些什么问题？

在注册中心找不到对应的服务，检查 service 实现类是否添加了@Service 注解无法连接到注册中心，检查配置文件中的对应的测试 ip 是否正确

17、Dubbo Monitor 实现原理？

Consumer 端在发起调用之前会先走 filter 链；provider 端在接收到请求时也是先走 filter 链，然后才进行真正的业务逻辑处理。
默认情况下，在 consumer 和 provider 的 filter 链中都会有 Monitorfilter。

- 1、MonitorFilter 向 DubboMonitor 发送数据
- 2、DubboMonitor 将数据进行聚合后（默认聚合 1min 中的统计数据）暂存到 ConcurrentMap<Statistics, AtomicReference> statisticsMap，然后使用一个含有 3 个线程（线程名字：DubboMonitorSendTimer）的线程池每隔 1min 钟，调用 SimpleMonitorService 遍历发送 statisticsMap 中的统计数据，每发送完毕一个，就重置当前的 Statistics 的 AtomicReference
- 3、SimpleMonitorService 将这些聚合数据塞入 BlockingQueue queue 中（队列大写为 100000）
- 4、SimpleMonitorService 使用一个后台线程（线程名为：DubboMonitorAsyncWriteLogThread）将 queue 中的数据写入文件（该线程以死循环的形式来写）
- 5、SimpleMonitorService 还会使用一个含有 1 个线程（线程名字：DubboMonitorTimer）的线程池每隔 5min 钟，将文件中的统计数据画成图表

18、Dubbo 用到哪些设计模式？

Dubbo 框架在初始化和通信过程中使用了多种设计模式，可灵活控制类加载、权限控制等功能。

工厂模式

Provider 在 export 服务时，会调用 ServiceConfig 的 export 方法。ServiceConfig 中有个字段：

```
private static final Protocol protocol =  
ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();
```

Dubbo 里有很多这种代码。这也是一种工厂模式，只是实现类的获取采用了 JDKSPI 的机制。这么实现的优点是可扩展性强，想要扩展实现，只需要在 classpath 下增加个文件就可以了，代码零侵入。另外，像上面的 Adaptive 实现，可以做到调用时动态决定调用哪个实现，但是由于这种实现采用了动态代理，会造成代码调试比较麻烦，需要分析出实际调用的实现类。

装饰器模式

Dubbo 在启动和调用阶段都大量使用了装饰器模式。以 Provider 提供的调用链为例，具体的调用链代码是在 ProtocolFilterWrapper 的 buildInvokerChain 完成的，具体是将注解中含有 group=provider 的 Filter 实现，按照 order 排序，最后的调用顺序是：

```
EchoFilter -> ClassLoaderFilter -> GenericFilter -> ContextFilter ->  
ExecuteLimitFilter -> TraceFilter -> TimeoutFilter -> MonitorFilter ->  
ExceptionFilter
```

更确切地说，这里是装饰器和责任链模式的混合使用。例如，EchoFilter 的作用是判断是否是回声测试请求，是的话直接返回内容，这是一种责任链的体现。而像 ClassLoaderFilter 则只是在主功能上添加了功能，更改当前线程

的 ClassLoader，这是典型的装饰器模式。

观察者模式

Dubbo 的 Provider 启动时，需要与注册中心交互，先注册自己的服务，再订阅自己的服务，订阅时，采用了观察者模式，开启一个 listener。注册中心会每 5 秒定时检查是否有服务更新，如果有更新，向该服务的提供者发送一个 notify 消息，provider 接受到 notify 消息后，即运行 NotifyListener 的 notify 方法，执行监听器方法。

动态代理模式

Dubbo 扩展 JDK SPI 的类 ExtensionLoader 的 Adaptive 实现是典型的动态代理实现。Dubbo 需要灵活地控制实现类，即在调用阶段动态地根据参数决定调用哪个实现类，所以采用先生成代理类的方法，能够做到灵活的调用。生成代理类的代码是 ExtensionLoader 的 createAdaptiveExtensionClassCode 方法。代理类的主要逻辑是，获取 URL 参数中指定参数的值作为获取实现类的 key。

19、Dubbo 配置文件是如何加载到 Spring 中的？

Spring 容器在启动的时候，会读取到 Spring 默认的一些 schema 以及 Dubbo 自定义的 schema，每个 schema 都会对应一个自己的 NamespaceHandler，NamespaceHandler 里面通过 BeanDefinitionParser 来解析配置信息并转化为需要加载的 bean 对象！

20、Dubbo SPI 和 Java SPI 区别？

JDK SPI

JDK 标准的 SPI 会一次性加载所有的扩展实现，如果有的扩展吃内存很耗时，但也没用上，很浪费资源。
所以只希望加载某个的实现，就不现实了

DUBBO SPI

- 1，对 Dubbo 进行扩展，不需要改动 Dubbo 的源码
- 2，延迟加载，可以一次只加载自己想要加载的扩展实现。
- 3，增加了对扩展点 IOC 和 AOP 的支持，一个扩展点可以直接 setter 注入其它扩展点。
- 4，Dubbo 的扩展机制能很好的支持第三方 IoC 容器，默认支持 Spring Bean。

21、Dubbo 支持分布式事务吗？

目前暂时不支持，可与通过 tcc-transaction 框架实现
介绍：tcc-transaction 是开源的 TCC 补偿性分布式事务框架
Git 地址：<https://github.com/changmingxie/tcc-transaction>
TCC-Transaction 通过 Dubbo 隐式传参的功能，避免自己对业务代码的入侵。

22、Dubbo 可以对结果进行缓存吗？

为了提高数据访问的速度。Dubbo 提供了声明式缓存，以减少用户加缓存的工作量

```
<dubbo:reference cache="true" />
```

其实比普通的配置文件就多了一个标签 cache="true"

23、服务上线怎么兼容旧版本？

可以用版本号（version）过渡，多个不同版本的服务注册到注册中心，版本号不同的服务相互间不引用。这个和服务分组的概念有一点类似。

24、Dubbo 必须依赖的包有哪些？

Dubbo 必须依赖 JDK，其他为可选。

25、Dubbo telnet 命令能做什么？

dubbo 服务发布之后，我们可以利用 telnet 命令进行调试、管理。Dubbo2.0.5 以上版本服务提供端口支持 telnet 命令

连接服务

telnet localhost 20880 //键入回车进入 Dubbo 命令模式。

查看服务列表

```
dubbo>ls  
com.test.TestService  
dubbo>ls com.test.TestService  
create  
delete  
query
```

□ ls (list services and methods)

□ ls : 显示服务列表。

□ ls -l : 显示服务详细信息列表。

□ ls XxxService : 显示服务的方法列表。

□ ls -l XxxService : 显示服务的方法详细信息列表。

26、Dubbo 支持服务降级吗？

以通过 dubbo:reference 中设置 mock="return null"。mock 的值也可以修改为 true，然后再跟接口同一个路径下实现一个 Mock 类，命名规则是 “接口名称+Mock” 后缀。然后在 Mock 类里实现自己的降级逻辑

27、Dubbo 如何优雅停机？

Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果使用 kill -9 PID 等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID 时，才会执行。

28、Dubbo 和 Dubbox 之间的区别？

Dubbox 是继 Dubbo 停止维护后，当当网基于 Dubbo 做的一个扩展项目，如加了服务可 Restful 调用，更新了开源组件等。

29、Dubbo 和 Spring Cloud 的区别？

根据微服务架构在各方面的要素，看看 Spring Cloud 和 Dubbo 都提供了哪些支持。

	Dubbo	Spring Cloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务网关	无	Spring Cloud Netflix Zuul
断路器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task
.....

使用 Dubbo 构建的微服务架构就像组装电脑，各环节我们的选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心，但是如果你是一名高手，那这些都不是问题；而 Spring Cloud 就像品牌机，在 Spring Source 的整合下，做了大量的兼容性测试，保证了机器拥有更高的稳定性，但是如果要在使用非原装组件外的东西，就需要对其基础有足够的了解。

30、你还了解别的分布式框架吗？

别的还有 spring 的 spring cloud，facebook 的 thrift，twitter 的 finagle 等

31、Dubbo是什么？

Dubbo是阿里巴巴开源的基于 Java 的高性能 RPC 分布式服务框架，现已成为 Apache 基金会孵化项目。

面试官问你如果这个都不清楚，那下面的就没必要问了。

官网：<http://dubbo.apache.org>

32、Dubbo默认使用什么注册中心，还有别的选择吗？

推荐使用 Zookeeper 作为注册中心，还有 Redis、Multicast、Simple 注册中心，但不推荐。

33、Dubbo有哪几种配置方式？

- 1) Spring 配置方式
- 2) Java API 配置方式

34、在 Provider 上可以配置的 Consumer 端的属性有哪些？

- 1) timeout : 方法调用超时
- 2) retries : 失败重试次数，默认重试 2 次
- 3) loadbalance : 负载均衡算法，默认随机
- 4) actives 消费者端，最大并发调用限制

35、Dubbo启动时如果依赖的服务不可用会怎样？

Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，默认 check="true"，可以通过 check="false" 关闭检查。

36、Dubbo推荐使用什么序列化框架，你知道的还有哪些？

推荐使用Hessian序列化，还有Duddo、FastJson、Java自带序列化。

37、Dubbo有哪几种负载均衡策略，默认是哪种？

负载均衡策略	说明
Random LoadBalance	随机，按权重设置随机概率（默认）
RoundRobin LoadBalance	轮询，按公约后的权重设置轮询比率
LeastActive LoadBalance	最少活跃调用数，相同活跃数的随机
ConsistentHash LoadBalance	一致性 Hash，相同参数的请求总是发到同一提供者

38、注册了多个同样的服务，如果测试指定的某一个服务呢？

可以配置环境点对点直连，绕过注册中心，将以服务接口为单位，忽略注册中心的提供者列表。

39、Dubbo支持服务多协议吗？

Dubbo 允许配置多协议，在不同服务上支持不同协议或者同一服务上同时支持多种协议。

40、当一个服务接口有多种实现时怎么做？

当一个接口有多种实现时，可以用 group 属性来分组，服务提供方和消费方都指定同一个 group 即可。

41、服务上线怎么兼容旧版本？

可以用版本号（version）过渡，多个不同版本的服务注册到注册中心，版本号不同的服务相互间不引用。这个和服务分组的概念有一点类似。

42、Dubbo可以对结果进行缓存吗？

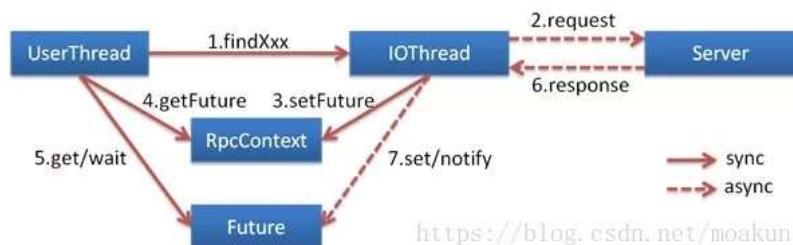
可以，Dubbo 提供了声明式缓存，用于加速热门数据的访问速度，以减少用户加缓存的工作量。

43、Dubbo服务之间的调用是阻塞的吗？

默认是同步等待结果阻塞的，支持异步调用。

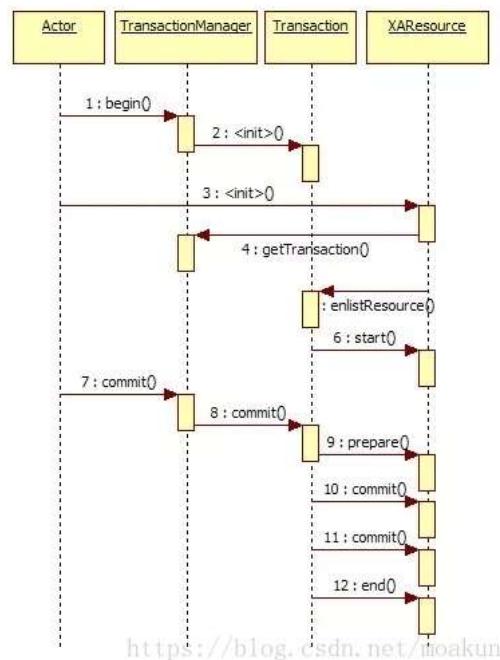
Dubbo 是基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小，异步调用会返回一个 Future 对象。

异步调用流程图如下。



44、Dubbo支持分布式事务吗？

目前暂时不支持，后续可能采用基于 JTA/XA 规范实现，如以图所示。



45、Dubbo支持服务降级吗？

Dubbo 2.2.0 以上版本支持。

46、Dubbo如何优雅停机？

Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果使用 kill -9 PID 等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID 时，才会执行。

47、服务提供者能实现失效踢出是什么原理？

服务失效踢出基于 Zookeeper 的临时节点原理。

29、如何解决服务调用链过长的问题？

Dubbo 可以使用 Pinpoint 和 Apache Skywalking(Incubator) 实现分布式服务追踪，当然还有其他很多方案。

30、服务读写推荐的容错策略是怎样的？

读操作建议使用 Failover 失败自动切换，默认重试两次其他服务器。

写操作建议使用 Failfast 快速失败，发一次调用失败就立即报错。

31、Dubbo必须依赖的包有哪些？

Dubbo 必须依赖 JDK，其他为可选。

32、Dubbo的管理控制台能做什么？

管理控制台主要包含：路由规则，动态配置，服务降级，访问控制，权重调整，负载均衡，等管理功能。

33、说说 Dubbo 服务暴露的过程。

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。

34、Dubbo 停止维护了吗？

2014 年开始停止维护过几年，17 年开始重新维护，并进入了 Apache 项目。

35、Dubbo 和 Dubbox 有什么区别？

Dubbox 是继 Dubbo 停止维护后，当当网基于 Dubbo 做的一个扩展项目，如加入了服务可 Restful 调用，更新了开源组件等。

36、你还了解别的分布式框架吗？

别的还有 Spring cloud、Facebook 的 Thrift、Twitter 的 Finagle 等。

37、Dubbo 能集成 Spring Boot 吗？

可以的，项目地址如下。

<https://github.com/apache/incubator-dubbo-spring-boot-project>

38、在使用过程中都遇到了些什么问题？

Dubbo 的设计目的是为了满足高并发小数据量的 rpc 调用，在大数据量下的性能表现并不好，建议使用 rmi 或 http 协议。

39、你读过 Dubbo 的源码吗？

要了解 Dubbo 就必须看其源码，了解其原理，花点时间看下吧，网上也有很多教程，后续有时间我也会在公众号上分享 Dubbo 的源码。

40、你觉得用 Dubbo 好还是 Spring Cloud 好？

扩展性的问题，没有好坏，只有适合不适合，不过我好像更倾向于使用 Dubbo, Spring Cloud 版本升级太快，组件更新替换太频繁，配置太繁琐，还有很多我觉得是没有 Dubbo 顺手的地方.....

MyBatis 面试题

1、什么是 Mybatis ?

- 1、Mybatis 是一个半 ORM (对象关系映射) 框架，它内部封装了 JDBC，开发时只需要关注 SQL 语句本身，不需要花费精力去处理加载驱动、创建连接、创建statement 等繁杂的过程。程序员直接编写原生态 sql，可以严格控制 sql 执行性能，灵活度高。
- 2、MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO 映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。3、通过 xml 文件或注解的方式将要执行的各种 statement 配置起来，并通过 java 对象和 statement 中 sql 的动态参数进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射为 java 对象并返回。（从执行 sql 到返回 result 的过程）。

2、Mybatis 的优点

- 1、基于 SQL 语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在 XML 里，解除 sql 与程序代码的耦合，便于统一管理；提供 XML 标签，支持编写动态 SQL 语句，并可重用。
- 2、与 JDBC 相比，减少了 50% 以上的代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接；
- 3、很好的与各种数据库兼容（因为 MyBatis 使用 JDBC 来连接数据库，所以只要 JDBC 支持的数据库 MyBatis 都支持）。
- 4、能够与 Spring 很好的集成；
- 5、提供映射标签，支持对象与数据库的 ORM 字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

3、MyBatis 框架的缺点

- 1、SQL 语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写 SQL 语句的功底有一定要求。
- 2、SQL 语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

4、MyBatis 框架适用场合

- 1、MyBatis 专注于 SQL 本身，是一个足够灵活的 DAO 层解决方案。
- 2、对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis 将是不错的选择。

5、MyBatis 与 Hibernate 有哪些不同？

- 1、Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句。
- 2、Mybatis 直接编写原生态 sql，可以严格控制 sql 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，因为这类软件需求变化频繁，一旦需求变化要求迅速输出成果。但是灵活的前提是 mybatis 无法做到数据库无关性，如果需要实现支持多种数据库的软件，则需要自定义多套 sql 映射文件，工作量大。
- 3、Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件，如果用 hibernate 开发可以节省很多代码，提高效率。

6、#{ } 和 \${ } 的区别是什么？

#{} 是预编译处理，\${} 是字符串替换。
Mybatis 在处理#{} 时，会将 sql 中的#{} 替换为?号，调用 PreparedStatement 的 set 方法来赋值；
Mybatis 在处理\${} 时，就是把 \${} 替换成变量的值。
使用#{} 可以有效的防止 SQL 注入，提高系统安全性

7、当实体类中的属性名和表中的字段名不一样，怎么办？

第 1 种：通过在查询的 sql 语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

```

<select id="selectorder" parameterType="int" resultType="me.gacl.domain.order">
    select order_id id, order_no orderno ,order_price price from
    orders where order_id=#{id};
</select>

```

第2种：通过来映射字段名和实体类属性名的——对应的关系。

```

<select id="getOrder" parameterType="int" resultMap="orderresultmap">
    select * from orders where order_id=#{id}
</select>
<resultMap type="me.gacl.domain.order" id="orderresultmap">
    <!-用 id 属性来映射主键字段->
    <id property="id" column="order_id">
        <!-用 result 属性来映射非主键字段，property 为实体类属性名，column 为数据表中的属性->
        <result property = "orderno" column ="order_no"/>
        <result property="price" column="order_price" />
    </resultMap>

```

8、模糊查询 like 语句该怎么写？

第1种：在 Java 代码中添加 sql 通配符。

```

string wildcardname = "%smi%";
List<Name> names = mapper.selectlike(wildcardname);

<select id="selectlike">
    select * from foo where bar like #{value}
</select>

```

第2种：在 sql 语句中拼接通配符，会引起 sql 注入

```

String wildcardname = "smi";
List<Name> names = mapper.selectlike(wildcardname);

<select id="selectlike">
    select * from foo where bar like "%"#{value}%""
</select>

```

9、通常一个 Xml 映射文件，都会写一个 Dao 接口与之对应，请问，这个 Dao 接口的工作原理是什么？Dao 接口里的方法，参数不同时，方法能重载吗？

Dao 接口即 Mapper 接口。接口的全限名，就是映射文件中的 namespace 的值；接口的方法名，就是映射文件中 Mapper 的 Statement 的 id 值；接口方法内的参数，就是传递给 sql 的参数。Mapper 接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为 key 值，可唯一定位一个 MapperStatement。在 Mybatis 中，每一个

```

<insert id="insertname">
    insert into names (name) values (#{value})
</insert>

```

然后在 java 代码中像下面这样执行批处理插入:

```
list < string > names = new ArrayList();
names.add("fred");
names.add("barney");
names.add("betty");
names.add("wilma");
// 注意这里 executorType.batch
SqlSession sqlSession =
sqlSessionFactory.openSession(executorType.batch);
try {
    NameMapper mapper = sqlSession.getMapper(NameMapper.class);
    for (String name: names) {
        mapper.insertName(name);
    }
    sqlSession.commit();
} catch (Exception e) {
    e.printStackTrace();
    sqlSession.rollback();
    throw e;
}
finally {
    sqlSession.close();
}
```

13、如何获取自动生成的(主)键值?

insert 方法总是返回一个 int 值，这个值代表的是插入的行数。
如果采用自增长策略，自动生成的键值在 insert 方法执行完后可以被设置到传入的参数对象中。

示例：

```

<insert id="insertname" usegeneratedkeys="true" keyproperty="id">
insert into names (name) values (#{name})
</insert>

name name = new name();
name.setName("fred");
int rows = mapper.insertname(name);
// 完成后, id 已经被设置到对象中
System.out.println("rows inserted = " + rows);
System.out.println("generated key value = " + name.getId());

```

14、在 mapper 中如何传递多个参数？

1、第一种：
DAO 层的函数

```
public UserselectUser(String name, String area);
```

对应的 xml, #{} 代表接收的是 dao 层中的第一个参数, #{1} 代表 dao 层中第二个参数, 更多参数一致往后加即可。

```

<select id="selectUser" resultMap="BaseResultMap">
select * from user_user_t where user_name = #{0} and user_area = #{1}
</select>

```

2、第二种：使用 @param 注解

```

public interface UserMapper {
User selectUser(@Param("username") String username,
@Param("hashedpassword") String hashedpassword);
}

```

然后就可以在 xml 像下面这样使用(推荐封装为一个 map, 作为单个参数传递给 mapper)

```

<select id="selectUser" resultType="User">
select id, username, hashedpassword from some_table
where username = #{username}
and hashedpassword = #{hashedpassword}
</select>

```

3、第三种：多个参数封装成 map

```

try {
    // 映射文件的命名空间.SQL 片段的 ID, 就可以调用对应的映射文件中的 SQL
    // 由于我们的参数超过了两个, 而方法中只有一个 Object 参数收集, 因此我们使用 Map 集合来装载我们的参数
    Map<String, Object> map = new HashMap();
    map.put("start", start);
    map.put("end", end);
    return sqlSession.selectList("StudentID.pagination", map);
} catch (Exception e) {
    e.printStackTrace();
    sqlSession.rollback();
    throw e;
}
finally {
    MybatisUtil.closeSqlSession();
}

```

15、Mybatis 动态 sql 有什么用？执行原理？有哪些动态 sql？

Mybatis 动态 sql 可以在 Xml 映射文件内, 以标签的形式编写动态 sql, 执行原理是根据表达式的值完成逻辑判断并动态拼接 sql 的功能。Mybatis 提供了 9 种动态 sql 标签 : trim | where | set | foreach | if | choose | when | otherwise | bind。

16、Xml 映射文件中，除了常见的 select|insert|update|delete 标签之外，还有哪些标签？

答：、、、，加上动态 sql 的 9 个标签，其中为 sql 片段标签，通过标签引入 sql 片段，为不支持自增的主键生成策略标签。

17、Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复？

不同的 Xml 映射文件，如果配置了 namespace，那么 id 可以重复；如果没有配置 namespace，那么 id 不能重复；

原因就是 namespace+id 是作为 Map<String, MapperStatement> 的 key 使用的，如果没有 namespace，就剩下 id，那么，id 重复会导致数据互相覆盖。有了 namespace，自然 id 就可以重复，namespace 不同，namespace+id 自然也就不同。

18、为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里？

Hibernate 属于全自动 ORM 映射工具，使用 Hibernate 查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而 Mybatis 在查询关联对象或关联集合对象时，需要手动编写 sql 来完成，所以，称之为半自动 ORM 映射工具。

19、一对一、一对多的关联查询？

```
<mapper namespace="com.lcb.mapping.userMapper">
    <!--association 一对一关联查询-->
    <select id="getClass" parameterType="int" resultMap="ClassesResultMap">
        select * from class c,teacher t where c.teacher_id=t.t_id and
        c.c_id=#{id}
    </select>
    <resultMap type="com.lcb.user.Classes" id="ClassesResultMap">
        <!-- 实体类的字段名和数据表的字段名映射 -->
        <id property="id" column="c_id"/>
        <result property="name" column="c_name"/>
        <association property="teacher">
            javaType="com.lcb.user.Teacher"
            <id property="id" column="t_id"/>
            <result property="name" column="t_name"/>
        </association>
    </resultMap>
    <!--collection 一对多关联查询-->
    <select id="getClass2" parameterType="int"
        resultMap="ClassesResultMap2">
        select * from class c,teacher t,student s where c.teacher_id=t.t_id
        and c.c_id=s.class_id and c.c_id=#{id}
    </select>
    <resultMap type="com.lcb.user.Classes" id="ClassesResultMap2">
        <id property="id" column="c_id"/>
        <result property="name" column="c_name"/>
        <association property="teacher">
            javaType="com.lcb.user.Teacher"
            <id property="id" column="t_id"/>
            <result property="name" column="t_name"/>
        </association>
        <collection property="student" ofType="com.lcb.user.Student">
            <id property="id" column="s_id"/>
            <result property="name" column="s_name"/>
        </collection>
    </resultMap>
</mapper>
```

20、MyBatis 实现一对一有几种方式？具体怎么操作的？

有联合查询和嵌套查询，联合查询是几个表联合查询，只查询一次，通过在 resultMap 里面配置 association 节点配置一对一的类就可以完成；

嵌套查询是先查一个表，根据这个表里面的结果的外键 id，去再另外一个表里面查询数据，也是通过 association 配置，但另外一个表的查询通过 select 属性配置

21、MyBatis 实现一对多有几种方式，怎么操作的？

有联合查询和嵌套查询。联合查询是几个表联合查询，只查询一次，通过在 resultMap 里面的 collection 节点配置一对多的类就可以完成；嵌套查询是先查一个表，根据这个表里面的结果的外键 id，去再另外一个表里面查询数据，也是通过配置 collection，但另外一个表的查询通过 select 节点配置。

22、Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？

答：Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否启用延迟加载 lazyLoadingEnabled=true|false。

它的原理是，使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 a.getB().getName()，拦截器 invoke() 方法发现 a.getB() 是 null 值，那么就会单独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 a.setB(b)，于是 a 的对象 b 属性就有值了，接着完成 a.getB().getName() 方法的调用。这就是延迟加载的基本原理。

当然了，不光是 Mybatis，几乎所有的包括 Hibernate，支持延迟加载的原理都是一样的。

23、Mybatis 的一级、二级缓存

1) 一级缓存：基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。

2) 二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态)，可在它的映射文件中配置；

3) 对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

24、什么是 MyBatis 的接口绑定？有哪些实现方式？

接口绑定，就是在 MyBatis 中任意定义接口，然后把接口里面的方法和 SQL 语句绑定。我们直接调用接口方法就可以，这样比起原来用了 SqlSession 提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式，一种是通过注解绑定，就是在接口的方法上面加上 @Select、@Update 等注解，里面包含 SQL 语句来绑定；另外一种就是通过 XML 里面写 SQL 来绑定，在这种情况下，要指定 XML 映射文件里面的 namespace 必须为接口的全路径名。当 SQL 语句比较简单时候，用注解绑定；当 SQL 语句比较复杂时候，用 XML 绑定，一般用 XML 绑定的比较多。

25、使用 MyBatis 的 mapper 接口调用时有哪些要求？

- 1、Mapper 接口方法名和 mapper.xml 中定义的每个 SQL 的 id 相同；
- 2、Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 SQL 的 parameterType 的类型相同；
- 3、Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 SQL 的 resultType 的类型相同；
- 4、Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

26、Mapper 编写有哪几种方式？

第一种：接口实现类继承 SqlSessionDaoSupport：使用此种方法需要编写 mapper 接口，mapper 接口实现类、mapper.xml 文件。

1、在 sqlMapConfig.xml 中配置 mapper.xml 的位置

```
<mappers>
<mapper resource="mapper.xml 文件的地址" />
<mapper resource="mapper.xml 文件的地址" />
</mappers>
```

2、定义 mapper 接口

3、实现类集成 SqlSessionDaoSupport，mapper 方法中可以 this.getSqlSession() 进行数据增删改查。

4、Spring 配置

```
<bean id="" class="mapper 接口的实现">
<property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
```

第二种：使用 org.mybatis.spring.mapper.MapperFactoryBean：

1、在 sqlMapConfig.xml 中配置 mapper.xml 的位置，如果 mapper.xml 和 mapper 接口的名称相同且在同一个目录，这里可以不用配置

```
<mappers>
<mapper resource="mapper.xml 文件的地址" />
<mapper resource="mapper.xml 文件的地址" />
</mappers>
```

2、定义 mapper 接口：

- 1、mapper.xml 中的 namespace 为 mapper 接口的地址
- 2、mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致

3、Spring 中定义

```
<bean id="" class="org.mybatis.spring.mapper.MapperFactoryBean">
<property name="mapperInterface" value="mapper 接口地址" />
<property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

第三种：使用 mapper 扫描器：

1、mapper.xml 文件编写：

mapper.xml 中的 namespace 为 mapper 接口的地址；

mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致；

如果将 mapper.xml 和 mapper 接口的名称保持一致，则不用在 sqlMapConfig.xml 中进行配置。

2、定义 mapper 接口：

注意 mapper.xml 的文件名和 mapper 的接口名称保持一致，且放在同一个目录

3、配置 mapper 扫描器：

```
<bean class="org.mybatis.spring.mapper.MappersScannerConfigurer">
<property name="basePackage" value="mapper 接口包地址" />
<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />
</bean>
```

4、使用扫描器后从 Spring 容器中获取 mapper 的实现对象。

27、简述 MyBatis 的插件运行原理，以及如何编写一个插件。

答：Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件，Mybatis 使用 JDK 的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke() 方法，当然，只会拦截那些你指定需要拦截的方法。

编写插件：实现 Mybatis 的 Interceptor 接口并复写 intercept() 方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

28、MyBatis实现一对一有几种方式?具体怎么操作的 ?

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次,通过在resultMap里面配置association节点配置一对一的类就可以完成;
嵌套查询是先查一个表,根据这个表里面的结果的外键id,去再另外一个表里面查询数据,也是通过association配置,但另外一个表的查询通过select属性配置

ZooKeeper 面试题

1、什么是Zookeeper?

ZooKeeper 是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

2、Zookeeper 如何保证了分布式一致性特性？

- 1、顺序一致性
- 2、原子性
- 3、单一视图
- 4、可靠性
- 5、实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于写请求，这些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。因此，随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。有序性是 zookeeper 中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为 xid (Zookeeper Transaction Id)。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个

zookeeper 最新的 xid

3、ZooKeeper 提供了什么？

- 1、文件系统
- 2、通知机制

4、Zookeeper 文件系统

Zookeeper 提供一个多层级的节点命名空间（节点称为 znode）。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。

Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得 Zookeeper 不能用于存放大量的数据，每个节点的存放数据上限为 1M。

5、ZAB 协议？

ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复的原子广播协议。

ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。

当整个 zookeeper 集群刚刚启动或者 Leader 服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首先选举产生新的 Leader 服务器，然后集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步，当集群中超过半数机器与该 Leader 服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader 服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

6、四种类型的数据节点 Znode

1、PERSISTENT-持久节点

除非手动删除，否则节点一直存在于 Zookeeper 上

2、EPHEMERAL-临时节点

临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。

3、PERSISTENT_SEQUENTIAL-持久顺序节点

基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

4、EPHEMERAL_SEQUENTIAL-临时顺序节点

基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

7、Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper 允许客户端向服务端的某个 Znode 注册一个 Watcher 监听，当服务端的一些指定事件触发了这个 Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据 Watcher 通知状态和事件类型做出业务上的改变。

工作机制：

- 1、客户端注册 watcher
- 2、服务端处理 watcher
- 3、客户端回调 watcher

Watcher 特性总结

1、一次性

无论是服务端还是客户端，一旦一个 Watcher 被触发，Zookeeper 都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。

2、客户端串行执行

客户端 Watcher 回调的过程是一个串行同步的过程。

3、轻量

3.1、Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。

3.2、客户端向服务端注册 Watcher 的时候，并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用 boolean 类型属性进行了标记。

4、watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务端之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不同时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee，即客户端监听事件后，才会感知它所监视 znode 发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。Zookeeper 只能保证最终的一致性，而无法保证强一致性。

5、注册 watcher getData、exists、getChildren

6、触发 watcher create、delete、setData

7、当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当 client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch 可能会丢失：对于一个未创建的 znode 的 exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。

8、客户端注册 Watcher 实现

- 1、调用 getData()/getChildren()/exist()三个 API，传入 Watcher 对象
- 2、标记请求 request，封装 Watcher 到 WatchRegistration
- 3、封装成 Packet 对象，发服务端发送 request
- 4、收到服务端响应后，将 Watcher 注册到 ZKWatcherManager 中进行管理
- 5、请求返回，完成注册。

9、服务端处理 Watcher 实现

1、服务端接收 Watcher 并存储

接收到客户端请求，处理请求判断是否需要注册 Watcher，需要的话将数据节点的节点路径和 ServerCnxn（ServerCnxn 代表一个客户端和服务端的连接，实现了 Watcher 的 process 接口，此时可以看成一个 Watcher 对象）存储在 WatcherManager 的 WatchTable 和 watch2Paths 中去。

2、Watcher 触发

以服务端接收到 setData() 事务请求触发 NodeDataChanged 事件为例：

2.1 封装 WatchedEvent 将通知状态（SyncConnected）、事件类型（NodeDataChanged）以及节点路径封装成一个 WatchedEvent 对象

2.2 查询 Watcher 从 WatchTable 中根据节点路径查找 Watcher

2.3 没找到；说明没有客户端在该数据节点上注册过 Watcher

2.4 找到；提取并从 WatchTable 和 Watch2Paths 中删除对应 Watcher（从这里可以看出 Watcher 在服务端是一次性的，触发一次就失效了）

3、调用 process 方法来触发 Watcher

这里 process 主要就是通过 ServerCnxn 对应的 TCP 连接发送 Watcher 事件通知

10、客户端回调 Watcher

客户端 SendThread 线程接收事件通知，交由 EventThread 线程回调 Watcher。客户端的 Watcher 机制同样是一次性的，一旦被触发后，该 Watcher 就失效了。

11、ACL 权限控制机制

UGO (User/Group/Others)

目前在 Linux/Unix 文件系统中使用，也是使用最广泛的权限控制方式。是一种粗粒度的文件系统权限控制模式。

ACL (Access Control List) 访问控制列表包括三个方面：

权限模式 (Scheme)

- 1、IP：从 IP 地址粒度进行权限控制
- 2、Digest：最常用，用类似于 username:password 的权限标识来进行权限配置，便于区分不同应用来进行权限控制
- 3、World：最开放的权限控制方式，是一种特殊的 digest 模式，只有一个权限标识“world:anyone”
- 4、Super：超级用户

授权对象

授权对象指的是权限赋予的用户或一个指定实体，例如 IP 地址或是机器名。

权限 Permission

- 1、CREATE：数据节点创建权限，允许授权对象在该 Znode 下创建子节点
- 2、DELETE：子节点删除权限，允许授权对象删除该数据节点的子节点
- 3、READ：数据节点的读取权限，允许授权对象访问该数据节点并读取其数据内容或子节点列表等
- 4、WRITE：数据节点更新权限，允许授权对象对该数据节点进行更新操作
- 5、ADMIN：数据节点管理权限，允许授权对象对该数据节点进行 ACL 相关设置操作

12、Chroot 特性

3.2.0 版本后，添加了 Chroot 特性，该特性允许每个客户端为自己设置一个命名空间。如果一个客户端设置了 Chroot，那么该客户端对服务器的任何操作，都将会被限制在其自己的命名空间下。

通过设置 Chroot，能够将一个客户端应用于 Zookeeper 服务端的一颗子树相对应，在那些多个应用公用一个 Zookeeper 集群的场景下，对实现不同应用间的相互隔离非常有帮助。

13、会话管理

分桶策略：将类似的会话放在同一区块中进行管理，以便于 Zookeeper 对会话进行不同区块的隔离处理以及同一区块的统一处理。

分配原则：每个会话的“下次超时时间点” (ExpirationTime)

计算公式：

```
ExpirationTime_ = currentTime + sessionTimeout  
ExpirationTime = (ExpirationTime_ / ExpirationInterval + 1) *  
ExpirationInterval , ExpirationInterval 是指 Zookeeper 会话超时检查时间  
间隔，默认 tickTime
```

14、服务器角色

Leader

- 1、事务请求的唯一调度和处理器，保证集群事务处理的顺序性
- 2、集群内部各服务的调度者

Follower

- 1、处理客户端的非事务请求，转发事务请求给 Leader 服务器
- 2、参与事务请求 Proposal 的投票
- 3、参与 Leader 选举投票

Observer

- 1、3.0 版本以后引入的一个服务器角色，在不影响集群事务处理能力的基础上提升集群的非事务处理能力
- 2、处理客户端的非事务请求，转发事务请求给 Leader 服务器
- 3、不参与任何形式的投票

15、Zookeeper 下 Server 工作状态

服务器具有四种状态，分别是 LOOKING、FOLLOWING、LEADING、OBSERVING。

- 1、LOOKING：寻找 Leader 状态。当服务器处于该状态时，它会认为当前集群中没有 Leader，因此需要进入 Leader 选举状态。
- 2、FOLLOWING：跟随者状态。表明当前服务器角色是 Follower。
- 3、LEADING：领导者状态。表明当前服务器角色是 Leader。
- 4、OBSERVING：观察者状态。表明当前服务器角色是 Observer。

16、数据同步

整个集群完成 Leader 选举之后，Learner (Follower 和 Observer 的统称) 回向 Leader 服务器进行注册。当 Learner 服务器向 Leader 服务器完成注册后，进入数据同步环节。

数据同步流程：（均以消息传递的方式进行）

Learner 向 Leader 注册

数据同步

同步确认

Zookeeper 的数据同步通常分为四类：

- 1、直接差异化同步（ DIFF 同步）
- 2、先回滚再差异化同步（ TRUNC+DIFF 同步）
- 3、仅回滚同步（ TRUNC 同步）
- 4、全量同步（ SNAP 同步）

在进行数据同步前，Leader 服务器会完成数据同步初始化：

peerLastZxid :

□ 从 learner 服务器注册时发送的 ACKEPOCH 消息中提取 lastZxid (该 Learner 服务器最后处理的 ZXID)

minCommittedLog :

□ Leader 服务器 Proposal 缓存队列 committedLog 中最小 ZXID

maxCommittedLog :

□ Leader 服务器 Proposal 缓存队列 committedLog 中最大 ZXID

直接差异化同步（ DIFF 同步）

□ 场景：peerLastZxid 介于 minCommittedLog 和 maxCommittedLog 之间

先回滚再差异化同步（ TRUNC+DIFF 同步）

□ 场景：当新的 Leader 服务器发现某个 Learner 服务器包含了一条自己没有的事务记录，那么就需要让该 Learner 服务器进行事务回滚-回滚到 Leader 服务器上存在的，同时也是最接近于 peerLastZxid 的 ZXID

仅回滚同步（ TRUNC 同步）

□ 场景：peerLastZxid 大于 maxCommittedLog 全量同步（ SNAP 同步）

□ 场景一：peerLastZxid 小于 minCommittedLog

□ 场景二：Leader 服务器上没有 Proposal 缓存队列且 peerLastZxid 不等于 lastProcessZxid

17、zookeeper 是如何保证事务的顺序一致性的？

zookeeper 采用了全局递增的事务 Id 来标识，所有的 proposal (提议) 都在被提出的时候加上了 zxid , zxid 实际上是一个 64 位的数字，高 32 位是 epoch (时期; 纪元; 世; 新时代) 用来标识 leader 周期，如果有新的 leader 产生出来，epoch 会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

18、zk 节点宕机如何处理？

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；

如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

ZK 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 ZK 节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。

所以

3 个节点的 cluster 可以挂掉 1 个节点 (leader 可以得到 2 票 > 1.5)

2 个节点的 cluster 就不能挂掉任何 1 个节点了 (leader 可以得到 1 票 <= 1)

19、zookeeper 负载均衡和 nginx 负载均衡区别

zk 的负载均衡是可以调控，nginx 只是能调权重，其他需要可控的都需要自己写插件；但是 nginx 的吞吐量比 zk 大很多，应该说按业务选择用哪种方式。

20、分布式集群中为什么会有 Master ?

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行 leader 选举。

21、Zookeeper 有哪几种几种部署模式？

部署模式：单机模式、伪集群模式、集群模式

22、集群最少要几台机器，集群规则是怎样的？

集群规则为 $2N+1$ 台， $N>0$ ，即 3 台

23、集群支持动态添加机器吗？

其实就是水平扩容了，Zookeeper 在这方面不太好。两种方式：

全部重启：关闭所有 Zookeeper 服务，修改配置之后启动。不影响之前客户端的会话。

逐个重启：在过半存活即可用的原则下，一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。

3.5 版本开始支持动态扩容

24、Zookeeper 对节点的 watch 监听通知是永久的吗？为什么不是永久的？

不是。官方声明：一个 Watch 事件是一个一次性的触发器，当被设置了 Watch 的数据发生了改变的时候，则服务器将这个改变发送给设置了 Watch 的客户端，以便通知它们。

为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况下，每次变动都要通知到所有的客户端，给网络和服务器造成很大压力。一般是客户端执行 getData("/节点 A",true)，如果节点 A 发生了变更或删除，客户端会得到它的 watch 事件，但是在之后节点 A 又发生了变更，而客户端又没有设置 watch 事件，就不再给客户端发送。

在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我只要最新的数据即可。

25、Zookeeper 的 java 客户端都有哪些？

java 客户端：zk 自带的 zkclient 及 Apache 开源的 Curator。

26、chubby 是什么，和 zookeeper 比你怎么看？

chubby 是 google 的，完全实现 paxos 算法，不开源。zookeeper 是 chubby 的开源实现，使用 zab 协议，paxos 算法的变种。

27、说几个 zookeeper 常用的命令。

常用命令：ls get set create delete 等。

28、ZAB 和 Paxos 算法的联系与区别？

相同点：

- 1、两者都存在一个类似于 Leader 进程的角色，由其负责协调多个 Follower 进程的运行
- 2、Leader 进程都会等待超过半数的 Follower 做出正确的反馈后，才会将一个提案进行提交
- 3、ZAB 协议中，每个 Proposal 中都包含一个 epoch 值来代表当前的 Leader 周期，Paxos 中名字为 Ballot

不同点：

ZAB 用来构建高可用的分布式数据主备系统（Zookeeper），Paxos 是用来构建分布式一致性状态机系统。

29、Zookeeper 的典型应用场景

Zookeeper 是一个典型的发布/订阅模式的分布式数据管理与协调框架，开发人员可以使用它来进行分布式数据的发布和订阅。

通过对 Zookeeper 中丰富的数据节点进行交叉使用，配合 Watcher 事件通知机制，可以非常方便的构建一系列分布式应用中都会涉及的核心功能，如：

- 1、数据发布/订阅
- 2、负载均衡
- 3、命名服务
- 4、分布式协调/通知
- 5、集群管理
- 6、Master 选举
- 7、分布式锁
- 8、分布式队列

30、数据发布/订阅

数据发布/订阅系统，即所谓的配置中心，顾名思义就是发布者发布数据供订阅者进行数据订阅。

目的

动态获取数据（配置信息）

实现数据（配置信息）的集中式管理和数据的动态更新

设计模式

Push 模式

Pull 模式

数据（配置信息）特性

- 1、数据量通常比较小
- 2、数据内容在运行时会动态更新
- 3、集群中各机器共享，配置一致

如：机器列表信息、运行时开关配置、数据库配置信息等

基于 Zookeeper 的实现方式

- 数据存储：将数据（配置信息）存储到 Zookeeper 上的一个数据节点
- 数据获取：应用在启动初始化节点从 Zookeeper 数据节点读取数据，并在该节点上注册一个数据变更 Watcher
- 数据变更：当变更数据时，更新 Zookeeper 对应节点数据，Zookeeper 会将数据变更通知发到各客户端，客户端接到通知后重新读取变更后的数据即可。

31、zk 的命名服务

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

32、分布式通知和协调

对于系统调度来说：操作人员发送通知实际是通过控制台改变某个节点的状态，然后 zk 将这些变化发送给注册了这个节点的 watcher 的所有客户端。对于执行情况汇报：每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据，这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

33、zk 的命名服务（文件系统）

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

34、zk 的配置管理（文件系统、通知机制）**

程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的 znode 下，当有配置发生改变时，也就是 znode 发生变化时，可以通过改变 zk 中某个目录节点的内容，利用 watcher 通知给各个客户端，从而更改配置。

35、Zookeeper 集群管理（文件系统、通知机制）**

所谓集群管理无在乎两点：是否有机器退出和加入、选举 master。对于第一点，所有机器约定在父目录下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它上船了。

新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount 又有了，对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为 master 就好。

36、Zookeeper 分布式锁（文件系统、通知机制**）

有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode 的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 distribute_lock 节点就释放出锁。

对于第二类，/distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，编号最小的获得锁，用完删除，依次方便。

37、Zookeeper 队列管理（文件系统、通知机制）

两种类型的队列：

- 1、同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。
- 2、队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建 PERSISTENT_SEQUENTIAL 节点，创建成功时 Watcher 通知等待的队列，队列删除序列号最小的节点用以消费。此场景下 Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。

38、Zookeeper 角色

Zookeeper 集群是一个基于主从复制的高可用集群，每个服务器承担如下三种角色中的一种

Leader

1. 一个 Zookeeper 集群同一时间只会有一个实际工作的 Leader，它会发起并维护与各 Follower 及 Observer 间的心跳。
2. 所有的写操作必须要通过 Leader 完成再由 Leader 将写操作广播给其它服务器。只要有超过半数节点（不包括 observer 节点）写入成功，该写请求就会被提交（类 2PC 协议）。

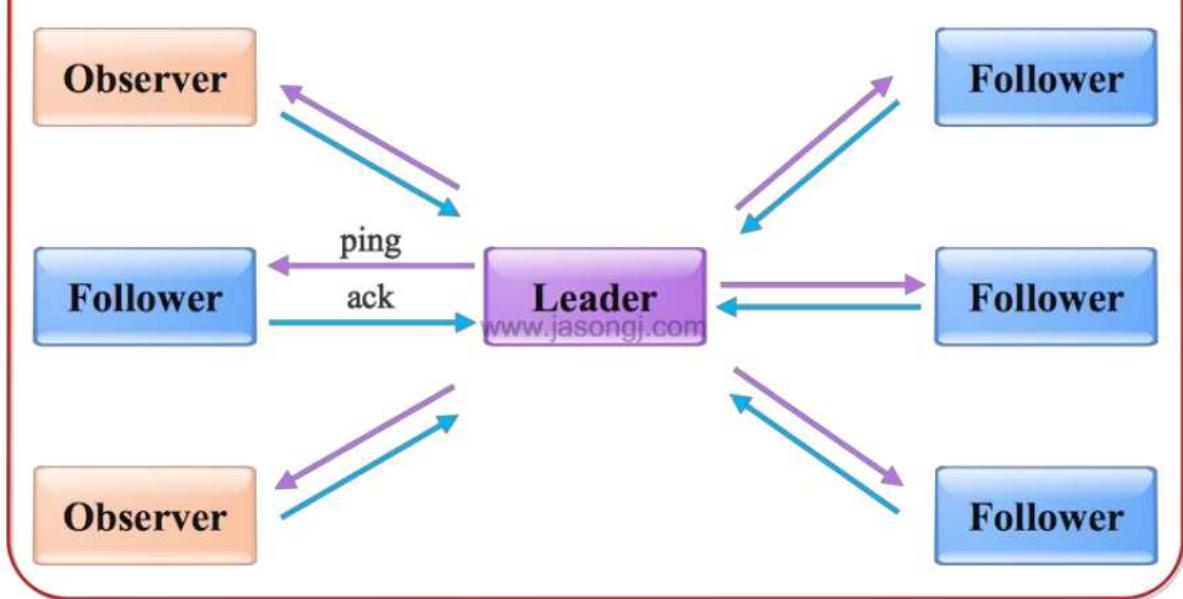
Follower

1. 一个 Zookeeper 集群可能同时存在多个 Follower，它会响应 Leader 的心跳，
2. Follower 可直接处理并返回客户端的读请求，同时会将写请求转发给 Leader 处理，
3. 并且负责在 Leader 处理写请求时对请求进行投票。

Observer

角色与 Follower 类似，但是无投票权。Zookeeper 需保证高可用和强一致性，为了支持更多的客户端，需要增加更多 Server；Server 增多，投票阶段延迟增大，影响性能；引入 Observer，Observer 不参与投票；bservers 接受客户端的连接，并将写请求转发给 leader 节点；加入更多 Observer 节点，提高伸缩性，同时不影响吞吐率。

Zookeeper Ensemble



39、事务编号 Zxid (事务请求计数器+ epoch)

在ZAB (ZooKeeper Atomic Broadcast, ZooKeeper 原子消息广播协议) 协议的事务编号 Zxid设计中 , Zxid 是一个 64 位的数字 , 其中低 32 位是一个简单的单调递增的计数器 , 针对客户端每一个事务请求 , 计数器加 1 ; 而高 32 位则代表 Leader 周期 epoch 的编号 , 每个当选产生一个新的 Leader 服务器 , 就会从这个 Leader 服务器上取出其本地日志中最大事务的 ZXID , 并从中读取 epoch 值 , 然后加 1 , 以此作为新的 epoch , 并将低 32 位从 0 开始计数。 Zxid (Transaction id) 类似于 RDBMS 中的事务 ID , 用于标识一次更新操作的 Proposal (提议)

ID。为了保证顺序性 , 该 zkid 必须单调递增。

40、epoch

epoch : 可以理解为当前集群所处的年代或者周期 , 每个 leader 就像皇帝 , 都有自己的年号 , 所以每次改朝换代 , leader 变更之后 , 都会在前一个年代的基础上加 1 。这样就算旧的 leader 崩溃恢复之后 , 也没有人听他的了 , 因为 follower 只听从当前年代的 leader 的命令。

41、Zab 协议有两种模式-恢复模式（选主）、广播模式（同步）

Zab 协议有两种模式 , 它们分别是恢复模式 (选主) 和广播模式 (同步) 。当服务启动或者在领导者崩溃后 , Zab 就进入了恢复模式 , 当领导者被选举出来 , 且大多数 Server 完成了和 leader 的状态同步以后 , 恢复模式就结束了。状态同步保证了 leader 和 Server 具有相同的系统状态。

42、Leader election (选举阶段-选出准 Leader)

Leader election (选举阶段) : 节点一开始都处于选举阶段 , 只要有一个节点得到超半数节点的票数 , 它就可以当选准 leader 。只有到达广播阶段 (broadcast) 准 leader 才会成为真正的 leader 。这一阶段的目的是就是为了选出一个准 leader , 然后进入下一个阶段

43、Discovery (发现阶段-接受提议、生成 epoch、接受 epoch)

Discovery (发现阶段) : 在这个阶段 , followers 跟准 leader 进行通信 , 同步 followers 最近接收的事务提议。这个一阶段的主要目的是发现当前大多数节点接收的最新提议 , 并且准 leader 生成新的 epoch , 让 followers 接受 , 更新它们的 accepted Epoch

一个 follower 只会连接一个 leader , 如果有一个节点 f 认为另一个 follower p 是 leader , f 在尝试连接 p 时会被拒绝 , f 被拒绝之后 , 就会进入重新选举阶段

44、Synchronization (同步阶段-同步 follower 副本)

Synchronization (同步阶段) : 同步阶段主要是利用 leader 前一阶段获得的最新提议历史 , 同步集群中所有的副本。只有当大多数节点都同步完成 , 准 leader 才会成为真正的 leader 。 follower 只会接收 zxid 比自己的 lastZxid 大的提议。

45、Broadcast (广播阶段-leader 消息广播)

Broadcast (广播阶段) : 到了这个阶段 , Zookeeper 集群才能正式对外提供事务服务 , 并且 leader 可以进行消息广播。同时如果有新的节点加入 , 还需要对新节点进行同步

46、ZAB 协议 JAVA 实现 (FLE-发现阶段和同步合并为 Recovery Phase (恢复阶段))

协议的 Java 版本实现跟上面的定义有些不同，选举阶段使用的是 Fast Leader Election (FLE)，它包含了选举的发现职责。因为 FLE 会选举拥有最新提议历史的节点作为 leader，这样就省去了发现最新提议的步骤。实际的实现将发现阶段和同步合并为 Recovery Phase (恢复阶段)。所以，ZAB 的实现只有三个阶段：Fast Leader Election；Recovery Phase；Broadcast Phase。

47、投票机制

每个 sever 首先给自己投票，然后用自己的选票和其他 sever 选票对比，权重大的胜出，使用权重较大的更新自身选票箱。具体选举过程如下：

1. 每个 Server 启动以后都询问其它的 Server 它要投票给谁。对于其他 server 的询问，server 每次根据自己的状态都回复自己推荐的 leader 的 id 和上一次处理事务的 zxid (系统启动时每个 server 都会推荐自己)
2. 收到所有 Server 回复以后，就计算出 zxid 最大的哪个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server。
3. 计算这过程中获得票数最多的的 sever 为获胜者，如果获胜者的票数超过半数，则改 server 被选为 leader。否则，继续这个过程，直到 leader 被选举出来
4. leader 就会开始等待 server 连接
5. Follower 连接 leader，将最大的 zxid 发送给 leader
6. Leader 根据 follower 的 zxid 确定同步点，至此选举阶段完成。
7. 选举阶段完成 Leader 同步后通知 follower 已经成为 up-to-date 状态
8. Follower 收到 up-to-date 消息后，又可以重新接受 client 的请求进行服务了

目前有 5 台服务器，每台服务器均没有数据，它们的编号分别是 1,2,3,4,5,按编号依次启动，它们的选择过程如下：

1. 服务器 1 启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器 1 的状态一直属于 Looking。
2. 服务器 2 启动，给自己投票，同时与之前启动的服务器 1 交换信息，由于服务器 2 的编号大所以服务器 2 胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是 LOOKING。
3. 服务器 3 启动，给自己投票，同时与之前启动的服务器 1,2 交换信息，由于服务器 3 的编号最大所以服务器 3 胜出，此时投票数正好大于半数，所以服务器 3 成为领导者，服务器 1,2 成为小弟。
4. 服务器 4 启动，给自己投票，同时与之前启动的服务器 1,2,3 交换信息，尽管服务器 4 的编号大，但之前服务器 3 已经胜出，所以服务器 4 只能成为小弟。
5. 服务器 5 启动，后面的逻辑同服务器 4 成为小弟

48、Zookeeper 工作原理 (原子广播)

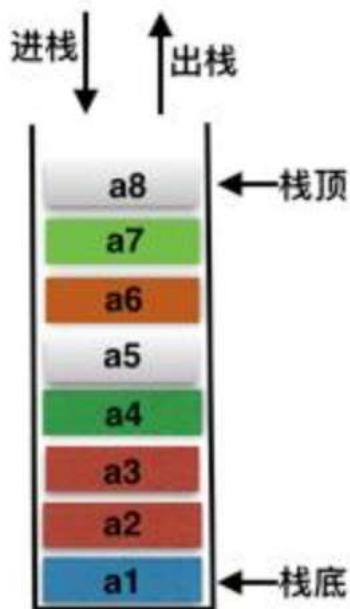
1. Zookeeper 的核心是原子广播，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式和广播模式。
2. 当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 server 的完成了和 leader 的状态同步以后，恢复模式就结束了。
3. 状态同步保证了 leader 和 server 具有相同的系统状态
4. 一旦 leader 已经和多数的 follower 进行了状态同步后，他就可以开始广播消息了，即进入广播状态。这时候当一个 server 加入 zookeeper 服务中，它会在恢复模式下启动，发现 leader，并和 leader 进行状态同步。待到同步结束，它也参与消息广播。Zookeeper 服务一直维持在 Broadcast 状态，直到 leader 崩溃了或者 leader 失去了大部分的 followers 支持。
5. 广播模式需要保证 proposal 被按顺序处理，因此 zk 采用了递增的事务 id 号(zxid)来保证。所有的提议(proposal)都在被提出的时候加上了 zxid。
6. 实现中 zxid 是一个 64 位的数字，它高 32 位是 epoch 用来标识 leader 关系是否改变，每次一个 leader 被选出来，它都会有一个新的 epoch。低 32 位是个递增计数。
7. 当 leader 崩溃或者 leader 失去大多数的 follower，这时候 zk 进入恢复模式，恢复模式需要重新选举出一个新的 leader，让所有的 server 都恢复到一个正确的状态

49、Znode 有四种形式的目录节点

1. PERSISTENT：持久的节点。
2. EPHEMERAL：暂时的节点。
3. PERSISTENT_SEQUENTIAL：持久化顺序编号目录节点。
4. EPHEMERAL_SEQUENTIAL：暂时化顺序编号目录节点。

1、栈 (stack)

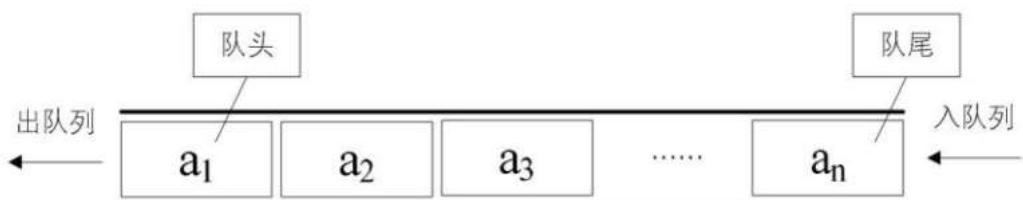
栈 (stack) 是限制插入和删除只能在一个位置上进行的表，该位置是表的末端，叫做栈顶 (top)。它是后进先出 (LIFO) 的。对栈的基本操作只有 push (进栈) 和 pop (出栈) 两种，前者相当于插入，后者相当于删除最后的元素。



图：栈的存储结构示意图

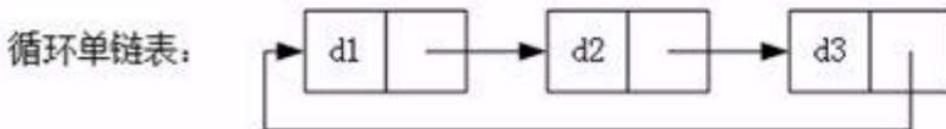
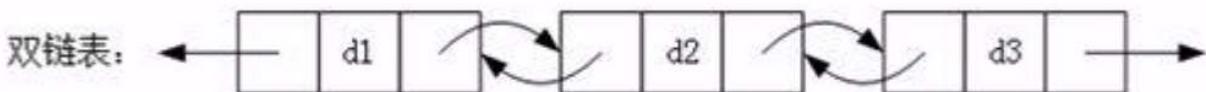
2、队列 (queue)

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端 (front) 进行删除操作，而在表的后端 (rear) 进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。



3、链表 (Link)

链表是一种数据结构，和数组同级。比如，Java 中我们使用的 ArrayList，其实现原理是数组。而LinkedList 的实现原理就是链表了。链表在进行循环遍历时效率不高，但是插入和删除时优势明显。



4、散列表 (Hash Table)

散列表 (Hash table , 也叫哈希表) 是一种查找算法 , 与链表、树等算法不同的是 , 散列表算法在查找时不需要进行一系列和关键字 (关键字是数据元素中某个数据项的值 , 用以标识一个数据元素) 的比较操作。

散列表算法希望能尽量做到不经过任何比较 , 通过一次存取就能得到所查找的数据元素 , 因而必须要在数据元素的存储位置和它的关键字 (可用 key 表示) 之间建立一个确定的对应关系 , 使每个关键字和散列表中一个唯一的存储位置相对应。因此在查找时 , 只要根据这个对应关系找到给定关键字在散列表中的位置即可。这种对应关系被称为散列函数 (可用 $h(key)$ 表示)。

常用的构造散列函数的方法有 :

1) 直接定址法 : 取关键字或关键字的某个线性函数值为散列地址。

即 : $h(key) = key$ 或 $h(key) = a * key + b$, 其中 a 和 b 为常数。

(2) 数字分析法

(3) 平方取值法 : 取关键字平方后的中间几位为散列地址。

(4) 折叠法 : 将关键字分割成位数相同的几部分 , 然后取这几部分的叠加和作为散列地址。

(5) 除留余数法 : 取关键字被某个不大于散列表表长 m 的数 p 除后所得的余数为散列地址 ,

即 : $h(key) = key \text{ MOD } p$ $p \leq m$

(6) 随机数法 : 选择一个随机函数 , 取关键字的随机函数值为它的散列地址 ,

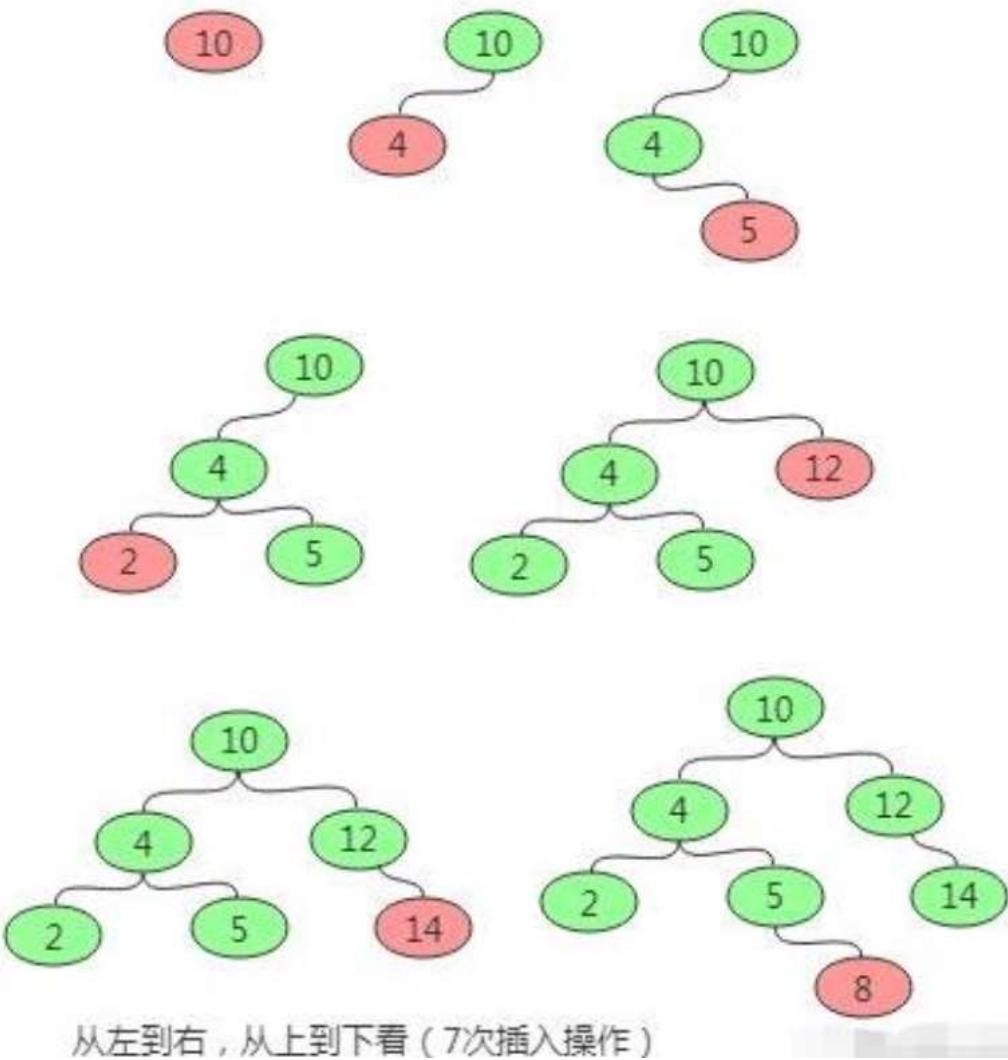
即 : $h(key) = \text{random}(key)$

5. 排序二叉树

首先如果普通二叉树每个节点满足 : 左子树所有节点值小于它的根节点值 , 且右子树所有节点值大于它的根节点值 , 则这样的二叉树就是排序二叉树。

插入操作

首先要从根节点开始往下找到自己要插入的位置 (即新节点的父节点) ; 具体流程是 : 新节点与当前节点比较 , 如果相同则表示已经存在且不能再重复插入 ; 如果小于当前节点 , 则到左子树中 寻找 , 如果左子树为空则当前节点为要找的父节点 , 新节点插入到当前节点的左子树即可 ; 如果大于当前节点 , 则到右子树中寻找 , 如果右子树为空则当前节点为要找的父节点 , 新节点插入到当前节点的右子树即可。



从左到右 , 从上到下看 (7 次插入操作)

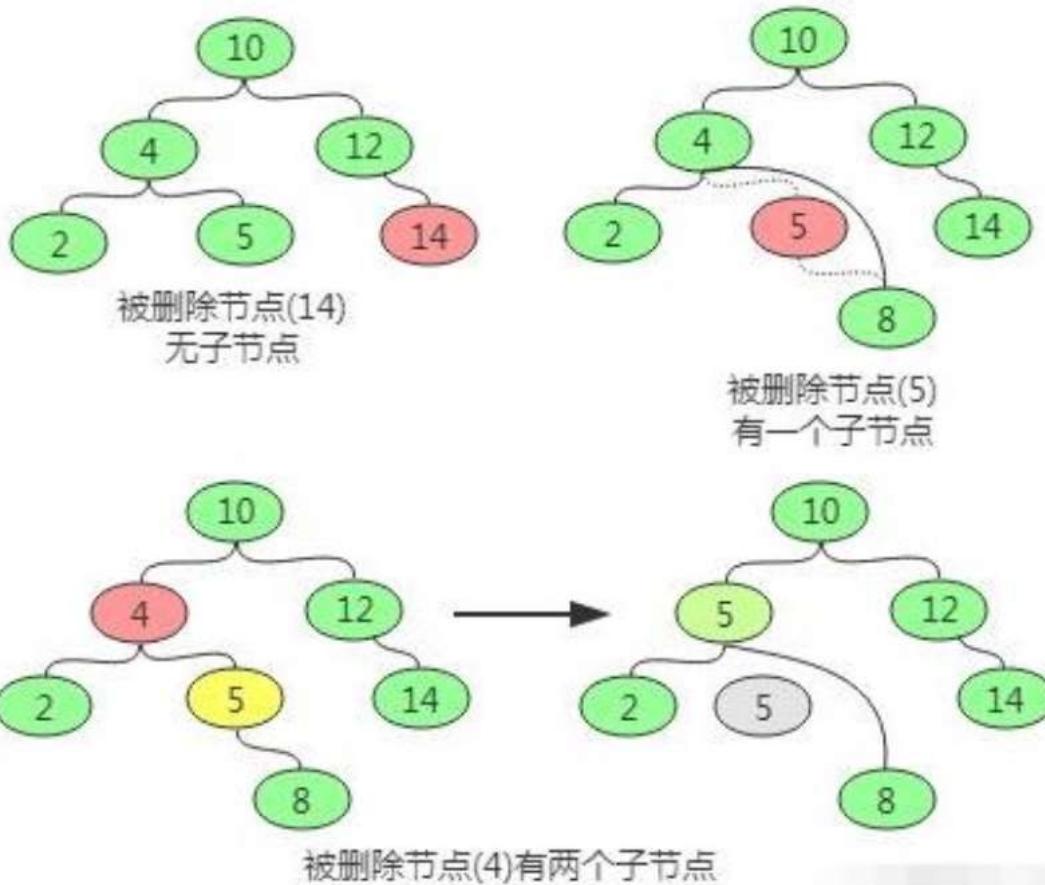
删除操作

删除操作主要分为三种情况 , 即要删除的节点无子节点 , 要删除的节点只有一个子节点 , 要删除的节点有两个子节点。

1. 对于要删除的节点无子节点可以直接删除 , 即让其父节点将该子节点置空即可。

2. 对于要删除的节点只有一个子节点 , 则替换要删除的节点为其子节点。

3. 对于要删除的节点有两个子节点 , 则首先找该节点的替换节点 (即右子树中最小的节点) , 接着替换要删除的节点为替换节点 , 然后删除替换节点。

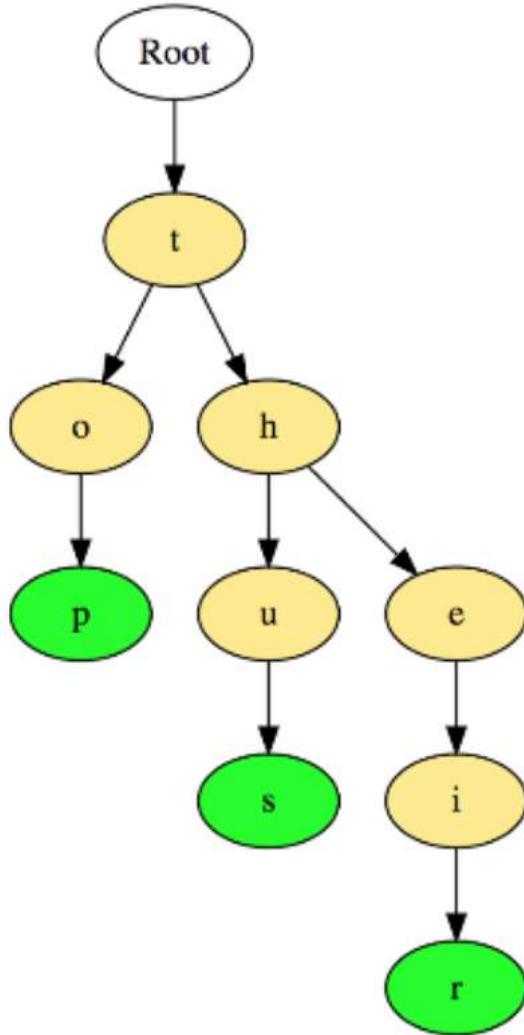


查询操作

查找操作的主要流程为：先和根节点比较，如果相同就返回，如果小于根节点则到左子树中递归查找，如果大于根节点则到右子树中递归查找。因此在排序二叉树中可以很容易获取最大（最右最深子节点）和最小（最左最深子节点）值。

6、前缀树

前缀树(Prefix Trees 或者 Trie)与树类似，用于处理字符串相关的问题时非常高效。它可以实现快速检索，常用于字典中的单词查询，搜索引擎的自动补全甚至 IP 路由。
下图展示了“top”，“thus”和“their”三个单词在前缀树中如何存储的：



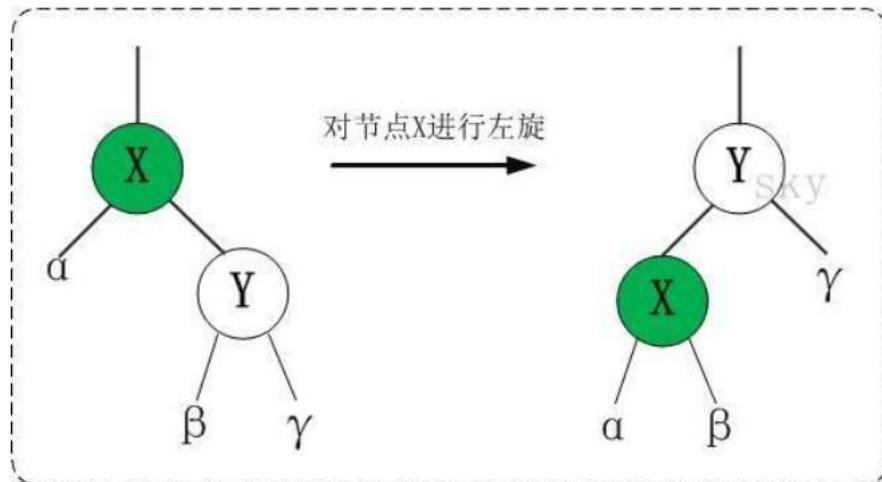
7、红黑树

红黑树的特性

- (1) 每个节点或者是黑色，或者是红色。
- (2) 根节点是黑色。
- (3) 每个叶子节点 (NIL) 是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]
- (4) 如果一个节点是红色，则它的子节点必须是黑色的。
- (5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

左旋

对 x 进行左旋，意味着，将“ x 的右孩子”设为“ x 的父亲节点”；即，将 x 变成了一个左节点(x 成为了 z 的左孩子)！。因此，左旋中的“左”，意味着“被旋转的节点将变成一个左节点”。



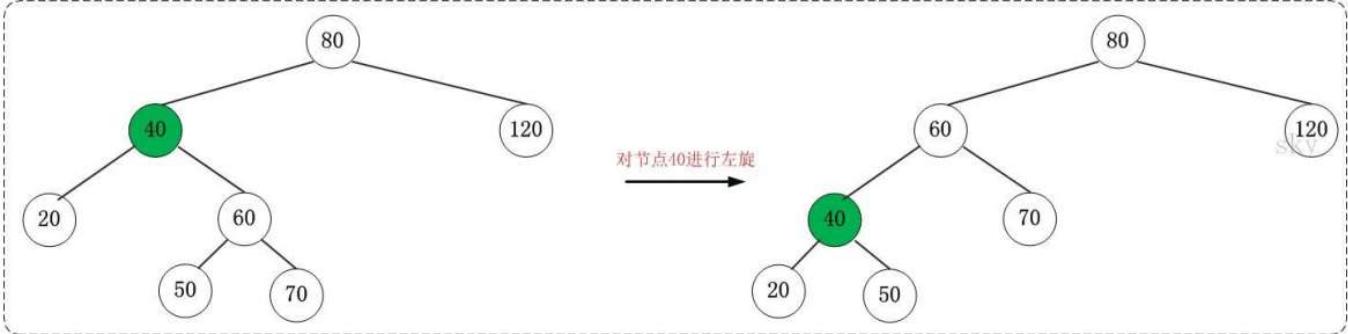
```

LEFT-ROTATE(T, x)
y ← right[x] // 前提: 这里假设 x 的右孩子为 y。下面开始正式操作
right[x] ← left[y] // 将 "y 的左孩子" 设为 "x 的右孩子"，即 将 β 设为 x 的右孩子
p[left[y]] ← x // 将 "x" 设为 "y 的左孩子的父亲"，即 将 β 的父亲设为 x
    
```

```

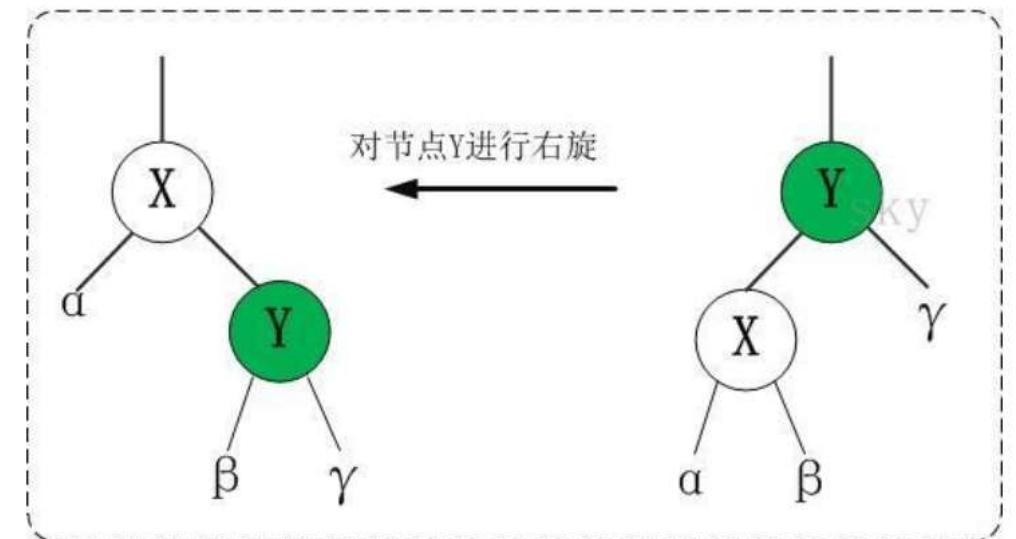
p[y] ← p[x] // 将 "x" 的父亲" 设为 "y" 的父亲"
if p[x] = nil[T]
then root[T] ← y // 情况 1: 如果 "x" 的父亲" 是空节点, 则将 y 设为根节点
else if x = left[p[x]]
then left[p[x]] ← y // 情况 2: 如果 x 是它父节点的左孩子, 则将 y 设为"x" 的父节点的左孩子
else right[p[x]] ← y // 情况 3: (x 是它父节点的右孩子) 将 y 设为"x" 的父节点的右孩子
left[y] ← x // 将 "x" 设为 "y" 的左孩子
p[x] ← y // 将 "x" 的父节点" 设为 "y"

```



右旋

对 x 进行右旋，意味着，将“x 的左孩子”设为“x 的父亲节点”；即，将 x 变成了一个右节点(x成了为 y 的右孩子)！因此，右旋中的“右”，意味着“被旋转的节点将变成一个右节点”。



```

RIGHT-ROTATE(T, y)
x ← left[y] // 前提: 这里假设 y 的左孩子为 x。下面开始正式操作
left[y] ← right[x] // 将 "x" 的右孩子" 设为 "y" 的左孩子", 即 将β设为 y 的左孩子
p[right[x]] ← y // 将 "y" 设为 "x" 的右孩子的父亲", 即 将β的父亲设为 y
p[x] ← p[y] // 将 "y" 的父亲" 设为 "x" 的父亲"
if p[y] = nil[T]
then root[T] ← x // 情况 1: 如果 "y" 的父亲" 是空节点, 则将 x 设为根节点
else if y = right[p[y]]
then right[p[y]] ← x // 情况 2: 如果 y 是它父节点的右孩子, 则将 x 设为"y" 的父节
点的左孩子"
else left[p[y]] ← x // 情况 3: (y 是它父节点的左孩子) 将 x 设为"y" 的父节点的左孩
子"
right[x] ← y // 将 "y" 设为 "x" 的右孩子"
p[y] ← x // 将 "y" 的父节点" 设为 "x"

```

添加

第一步: 将红黑树当作一颗二叉查找树，将节点插入。

第二步：将插入的节点着色为“红色”。

根据被插入节点的父节点的情况，可以将“当节点 z 被着色为红色节点，并插入二叉树”划分为三种情况来处理。

① 情况说明：被插入的节点是根节点。

处理方法：直接把此节点涂为黑色。

② 情况说明：被插入的节点的父节点是黑色。

处理方法：什么也不需要做。节点被插入后，仍然是红黑树。

③ 情况说明：被插入的节点的父节点是红色。这种情况下，被插入节点是一定存在非空祖父节点的；进一步的讲，被插入节点也一定存在叔叔节点(即使叔叔节点为空，我们也视之为存在，空节点本身就是黑色节点)。理解这点之后，我们依据“叔叔节点的情况”，将这种情况进一步划分为 3 种情况(Case)

	现象说明	处理策略
Case 1	当前节点的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色。	(01) 将“父节点”设为黑色。 (02) 将“叔叔节点”设为黑色。 (03) 将“祖父节点”设为“红色”。 (04) 将“祖父节点”设为“当前节点”（红色节点）；即，之后继续对“当前节点”进行操作。
Case 2	当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子	(01) 将“父节点”作为“新的当前节点”。 (02) 以“新的当前节点”为支点进行左旋。
Case 3	当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的左孩子	(01) 将“父节点”设为“黑色”。 (02) 将“祖父节点”设为“红色”。 (03) 以“祖父节点”为支点进行右旋。

第三步：通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。

删除

第一步：将红黑树当作一颗二叉查找树，将节点删除。

这和“删除常规二叉查找树中删除节点的方法是一样的”。分 3 种情况：

- ① 被删除节点没有儿子，即为叶节点。那么，直接将该节点删除就 OK 了。
- ② 被删除节点只有一个儿子。那么，直接删除该节点，并用该节点的唯一子节点顶替它的位置。
- ③ 被删除节点有两个儿子。那么，先找出它的后继节点；然后把“它的后继节点的内容”复制给“该节点的内容”；之后，删除“它的后继节点”。

第二步：通过“旋转和重新着色”等一系列来修正该树，使之重新成为一棵红黑树。

因为“第一步”中删除节点之后，可能会违背红黑树的特性。所以需要通过“旋转和重新着色”来修正

该树，使之重新成为一棵红黑树。

选择重着色 3 种情况。

① 情况说明：x 是“红+黑”节点。

处理方法：直接把 x 设为黑色，结束。此时红黑树性质全部恢复。

② 情况说明：x 是“黑+黑”节点，且 x 是根。

处理方法：什么都不做，结束。此时红黑树性质全部恢复。

③ 情况说明：x 是“黑+黑”节点，且 x 不是根。

处理方法：这种情况又可以划分为 4 种子情况。这 4 种子情况如下表所示：

	现象说明	处理策略
Case 1	x 是“黑+黑”节点，x 的兄弟节点是红色。（此时x的父节点和x的兄弟节点的子节点都是黑节点）。	(01) 将x的兄弟节点设为“黑色”。 (02) 将x的父节点设为“红色”。 (03) 对x的父节点进行左旋。 (04) 左旋后，重新设置x的兄弟节点。
Case 2	x 是“黑+黑”节点，x 的兄弟节点是黑色，x 的兄弟节点的两个孩子都是黑色。	(01) 将x的兄弟节点设为“红色”。 (02) 设置“x的父节点”为“新的x节点”。
Case 3	x 是“黑+黑”节点，x 的兄弟节点是黑色；x 的兄弟节点的左孩子是红色，右孩子是黑色的。	(01) 将x兄弟节点的左孩子设为“黑色”。 (02) 将x兄弟节点设为“红色”。 (03) 对x的兄弟节点进行右旋。 (04) 右旋后，重新设置x的兄弟节点。
Case 4	x 是“黑+黑”节点，x 的兄弟节点是黑色；x 的兄弟节点的右孩子是红色的，x 的兄弟节点的左孩子任意颜色。	(01) 将x父节点颜色 赋值给 x的兄弟节点。 (02) 将x父节点设为“黑色”。 (03) 将x兄弟节点的右子节设为“黑色”。 (04) 对x的父节点进行左旋。 (05) 设置“x”为“根节点”。

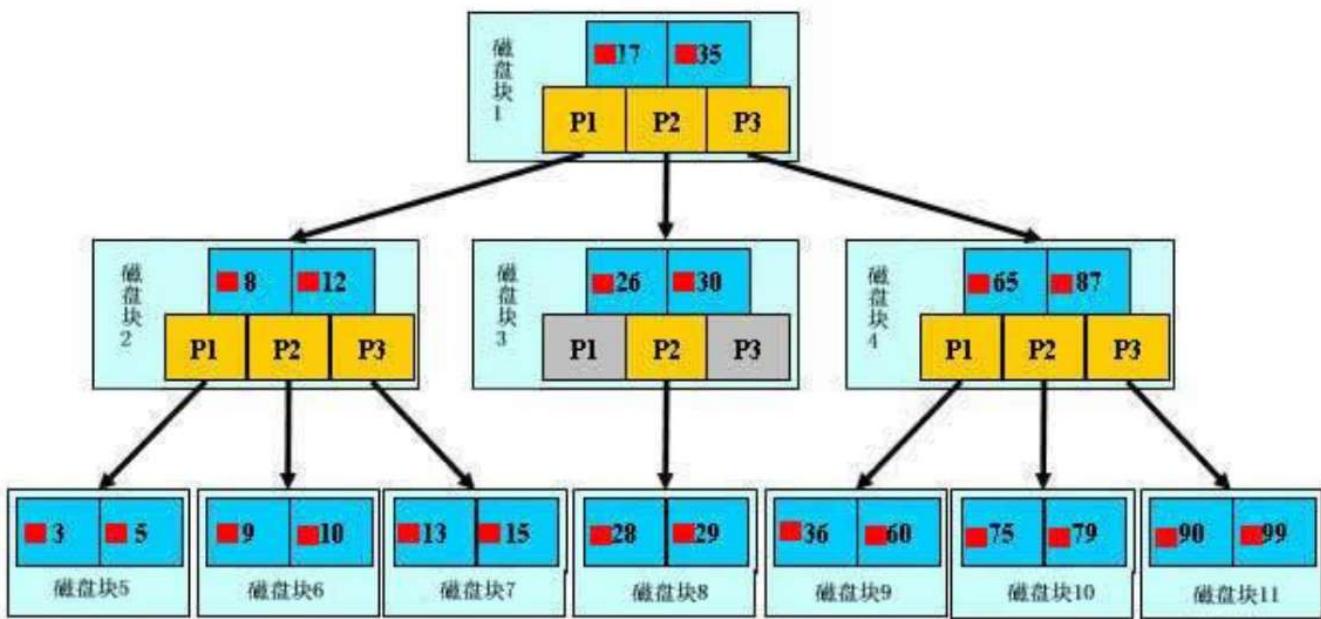
参考：<https://www.jianshu.com/p/038585421b73>

代码实现：<https://www.cnblogs.com/skywang12345/p/3624343.html>

8、B-TREE

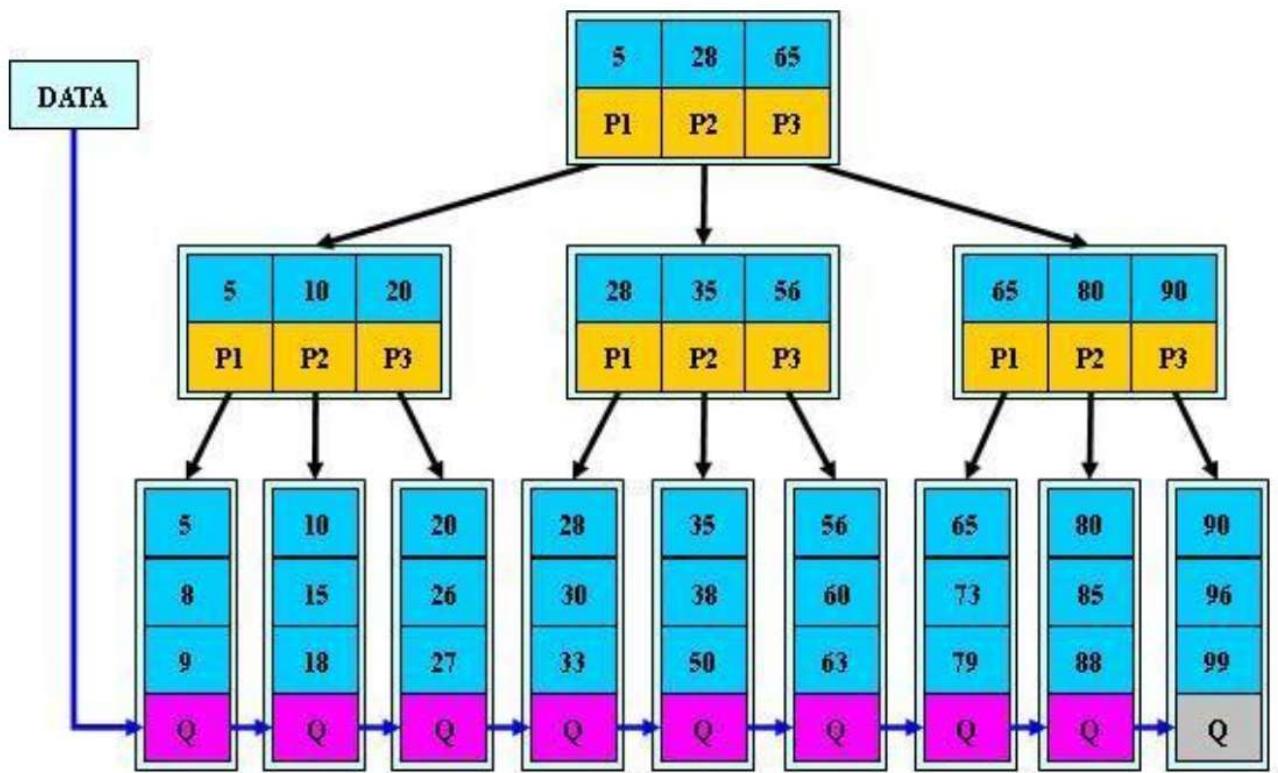
B-tree 又叫平衡多路查找树。一棵 m 阶的 B-tree (m 叉树)的特性如下 (其中 ceil(x)是一个取上限的函数) :

1. 树中每个结点至多有 m 个孩子；
2. 除根结点和叶子结点外，其它每个结点至少有有 ceil(m / 2)个孩子；
3. 若根结点不是叶子结点，则至少有 2 个孩子（特殊情况：没有孩子的根结点，即根结点为叶子结点，整棵树只有一个根节点）；
4. 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息(可以看做是外部结点或查询失败的结点，实际上这些结点不存在，指向这些结点的指针都为 null)；
5. 每个非终端结点中包含有 n 个关键字信息：(n , P0 , K1 , P1 , K2 , P2 , , Kn , Pn)。其中：
 - a) Ki (i=1...n)为关键字，且关键字按顺序排序 K(i-1) < Ki。
 - b) Pi 为指向子树根的接点，且指针 P(i-1)指向子树种所有结点的关键字均小于 Ki，但都大于 K(i-1)。
 - c) 关键字的个数 n 必须满足：ceil(m / 2)-1 <= n <= m-1。



一棵 m 阶的 B+tree 和 m 阶的 B-tree 的差异在于：

- 1.有 n 棵子树的结点中含有 n 个关键字；(B-tree 是 n 棵子树有 n-1 个关键字)
- 2.所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。(B-tree 的叶子节点并没有包括全部需要查找的信息)
- 3.所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。(B-tree 的非终节点也包含需要查找的有效信息)



参考：<https://www.jianshu.com/p/1ed61b4cca12>

9、位图

位图的原理就是用一个 bit 来标识一个数字是否存在，采用一个 bit 来存储一个数据，所以这样可以大大的节省空间。bitmap 是很常用的数据结构，比如用于 Bloom Filter 中；用于无重复整数的排序等等。bitmap 通常基于数组来实现，数组中每个元素可以看成是一系列二进制数，所有元素组成更大的二进制集合。

<https://www.cnblogs.com/polly333/p/4760275.html>

算法面试题

1、数据里有{1,2,3,4,5,6,7,8,9}，请随机打乱顺序，生成一个新的数组（请以代码实现）

```
import java.util.Arrays;
//打乱数组
public class Demo1 {
    //随机打乱
    public static int[] strand(int[] a) {
        int[] b = new int[a.length];
        for(int i = 0; i < a.length; i++) {
            //随机获取下标
            int tmp = (int)(Math.random()*(a.length - i)); //随机数:[ 0 ,
            a.length - i )
            b[i] = a[tmp];
            //将此时a[tmp]的下标移动到靠后的位置
            int change = a[a.length - i - 1];
            a[a.length - i - 1] = a[tmp];
            a[tmp] = change;
        }
        return b;
    }
    public static void main(String[] args) {
        int[] a = {1,2,3,4,5,6,7,8,9};
        System.out.println(Arrays.toString(strand(a)));
    }
}
```

2、写出代码判断一个整数是不是2的阶次方（请代码实现，谢绝调用API方法）

```
import java.util.Scanner;
//判断整数是不是2的阶次方
public class Demo2 {
    public static boolean check(int sum) {
        boolean flag = true; //判断标志
        while(sum > 1) {
            if (sum % 2 == 0) {
                sum = sum/2;
            } else {
                flag = false;
                break;
            }
        }
        return flag;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入一个整数:");
        int sum = scanner.nextInt();
        System.out.println(sum + " 是不是2的阶次方: " + check(sum));
    }
}
```

3、假设今日是2015年3月1日，星期日，请算出13个月零6天后是星期几，距离现在多少天（请用代码实现，谢绝调用API方法）

```
i
import java.util.Scanner;
//算出星期几
public class Demo4 {
    public static String[] week = {"星期日", "星期一", "星期二", "星期三", "星期四", "星期五", "星期六"};
    public static int i = 0;
    public static int[] monthday1 = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    public static int[] monthday2 = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    //查看距离当前天数的差值
    public static String distance(int year, int month, int day, int newMonth, int newDay) {
        int sum = 0; //设定初始距离天数
        if (month + newMonth >= 12) {
            if ((year + 1) % 4 == 0 && (year + 1) % 100 != 0 || (year + 1) % 400 == 0) {
                sum += 366 + newDay;
                for (int i = 0; i < newMonth - 12; i++) {
                    sum += monthday1[month + i];
                }
            } else {
                sum += 365 + newDay;
                for (int i = 0; i < newMonth - 12; i++) {
                    sum += monthday1[month + i];
                }
            }
        } else {
            for (int i = 0; i < newMonth; i++) {
                sum += monthday1[month + i];
            }
            sum += newDay;
        }
        return week[sum % 7];
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入当前年份");
        int year = scanner.nextInt();
        System.out.println("请输入当前月份");
        int month = scanner.nextInt();
        System.out.println("请输入当前天数");
        int day = scanner.nextInt();
        System.out.println("请输入当前是星期几：以数字表示，如：星期天 为 0");
        int index = scanner.nextInt();
        System.out.println("今天是：" + year + "-" + month + "-" + day + " " +
week[index]);
        System.err.println("请输入相隔月份");
        int newMonth = scanner.nextInt();
        System.out.println("请输入剩余天数");
        int newDay = scanner.nextInt();
        System.out.println("经过" + newMonth + "月" + newDay + "天后，是" +
distance(year, month, day, newMonth, newDay));
    }
}
```

4、有两个篮子，分别为A 和 B，篮子A里装有鸡蛋，篮子B里装有苹果，请用面向对象的思想实现两个篮子里的物品交换（请用代码实现）

```
//面向对象思想实现篮子物品交换
public class Demo5 {
    public static void main(String[] args) {
        //创建篮子
        Basket A = new Basket("A");
        Basket B = new Basket("B");
        //装载物品
        A.load("鸡蛋");
        B.load("苹果");
        //交换物品
        A.change(B);
        A.show();
        B.show();
    }
}

class Basket{
    public String name; //篮子名称
    private Goods goods; //篮子中所装物品
    public Basket(String name) {
        // TODO Auto-generated constructor stub
        this.name = name;
        System.out.println(name + "篮子被创建");
    }
}
```

```

    }
    //装物品函数
    public void load(String name) {
        goods = new Goods(name);
        System.out.println(this.name + "装载了" + name + "物品");
    }
    public void change(Basket B) {
        System.out.println(this.name + "和" + B.name + "中的物品发生了交换");
        String tmp = this.goods.getName();
        this.goods.setName(B.goods.getName());
        B.goods.setName(tmp);
    }
    public void show() {
        System.out.println(this.name + "中有" + goods.getName() + "物品");
    }
}

class Goods{
    private String name; //物品名称
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Goods(String name) {
        // TODO Auto-generated constructor stub
        this.name = name;
    }
}

```

5、二分查找

又叫折半查找，要求待查找的序列有序。每次取中间位置的值与待查关键字比较，如果中间位置的值比待查关键字大，则在前半部分循环这个查找的过程，如果中间位置的值比待查关键字小，则在后半部分循环这个查找的过程。直到查找到了为止，否则序列中没有待查的关键字。

```

public static int biSearch(int []array,int a){
    int lo=0;
    int hi=array.length-1;
    int mid;
    while(lo<=hi){
        mid=(lo+hi)/2;//中间位置
        if(array[mid]==a){
            return mid+1;
        }else if(array[mid]<a){ //向右查找
            lo=mid+1;
        }else{ //向左查找
            hi=mid-1;
        }
    }
    return -1;
}

```

6、冒泡排序算法

- (1) 比较前后相邻的两个数据，如果前面数据大于后面的数据，就将这两个数据交换。
- (2) 这样对数组的第 0 个数据到 N-1 个数据进行一次遍历后，最大的一个数据就“沉”到数组第 N-1 个位置。
- (3) N=N-1，如果 N 不为 0 就重复前面二步，否则排序完成。

```

public static void bubbleSort1(int [] a, int n){
    int i, j;
    for(i=0; i<n; i++){//表示 n 次排序过程。
        for(j=1; j<n-i; j++){
            if(a[j-1] > a[j]){//前面的数字大于后面的数字就交换
                //交换 a[j-1] 和 a[j]
                int temp;
                temp = a[j-1];
                a[j-1] = a[j];
                a[j]=temp;
            }
        }
    }
}

```

7、插入排序算法

通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应的位置并插入。插入排序非常类似于整扑克牌。在开始摸牌时，左手是空的，牌面朝下放在桌上。接着，一次从桌上摸起一张牌，并将它插入到左手一把牌中的正确位置上。为了找到这张牌的正确位置，要将它与手中已有的牌从右到左地进行比较。无论什么时候，左手中的牌都是排好序的。如果输入数组已经是排好序的话，插入排序出现最佳情况，其运行时间是输入规模的一个线性函数。如果输入数组是逆序排列的，将出现最坏情况。平均情况与最坏情况一样，其时间代价是 $O(n^2)$ 。



[初始关键字]:	(49)	38	65	97	76	13	27	<u>49</u>
i=2:	(38)	(38 49)	65	97	76	13	27	<u>49</u>
i=3:	(65)	(38 49 65)	97	76	13	27	<u>49</u>	
i=4:	(97)	(38 49 65 97)	76	13	27	<u>49</u>		
i=5:	(76)	(38 49 65 76 97)	13	27	<u>49</u>			
i=6:	(13)	(13 38 49 65 76 97)	27	<u>49</u>				
i=7:	(27)	(13 27 38 49 65 76 97)	<u>49</u>					
i=8:	(49)	(13 27 38 49 49 65 76 97)						
			↑ 监视哨L.r[0]					

```
public void sort(int arr[]){
    for(int i = 1; i<arr.length;i++)
    {
        //插入的数
        int insertVal = arr[i];
        //被插入的位置(准备和前一个数比较)
        int index = i-1;
        //如果插入的数比被插入的数小
        while(index>=0&&insertVal<arr[index])
        {
            //将把 arr[index] 向后移动
            arr[index+1]=arr[index];
            //让 index 向前移动
            index--;
        }
        //把插入的数放入合适位置
        arr[index+1]=insertVal;
    }
}
```

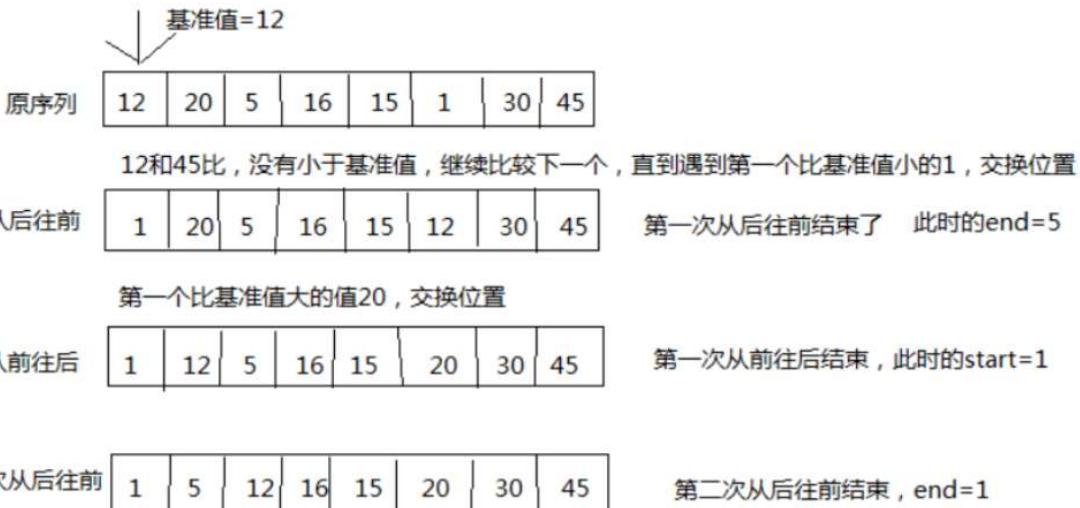
8、快速排序算法

快速排序的原理：选择一个关键值作为基准值。比基准值小的都在左边序列（一般是无序的），比基准值大的都在右边（一般是无序的）。一般选择序列的第一个元素。

一次循环：从后往前比较，用基准值和最后一个值比较，如果有继续比较下一个，直到找到第一个比基准值小的值才交换。找到这个值之后，又从前往后开始比较，如果有比基准值大的，交换位置，如果没有继续比较下一个，直到找到第一个比基准值大的值才交换。直到从前往后的比较索引>从后往前比较的索引，结束第一次循环，此时，对于基准值来说，左右两边就是有序的了。

```
public void sort(int[] a,int low,int high){
    int start = low;
    int end = high;
    int key = a[low];
    while(end>start){
        //从后往前比较
        while(end>start&&a[end]>=key)
        //如果没有比关键值小的，比较下一个，直到有比关键值小的交换位置，然后又从前往后比较
        end--;
        if(a[end]<=key){
            int temp = a[end];
            a[end] = a[start];
            a[start] = temp;
        }
        //从前往后比较
        while(end>start&&a[start]<=key){
            //如果没有比关键值大的，比较下一个，直到有比关键值大的交换位置
            start++;
            if(a[start]>=key){
                int temp = a[start];
                a[start] = a[end];
                a[end] = temp;
            }
        }
        //此时第一次循环比较结束，关键值的位置已经确定了。左边的值都比关键值小，右边的值都比关键值大，但是两边的顺序还有可能是不一样的，进行下面的递归调用
    }
    //递归
    if(start>low) sort(a,low,start-1); //左边序列。第一个索引位置到关键值索引-1
    if(end<high) sort(a,end+1,high); //右边序列。从关键值索引+1 到最后一个
}
```

```
}
```

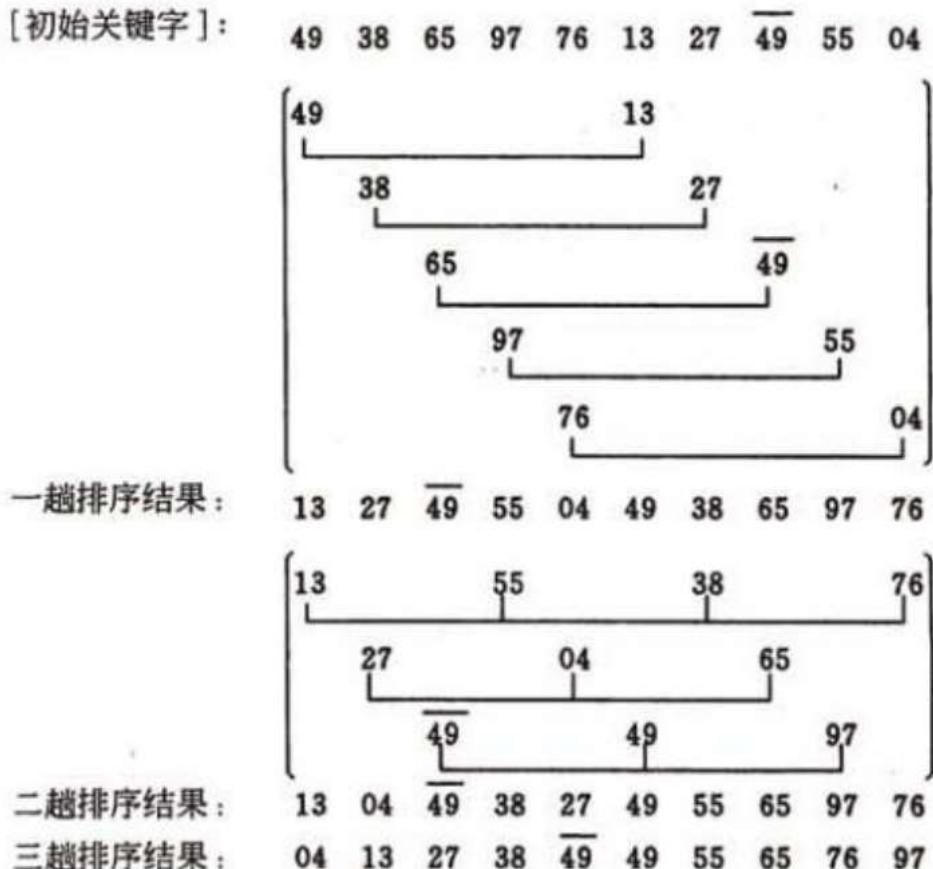


上面的循环之后得到了基准值左右两边的序列了。注意：交换的是起始和结束位置的值，不是基准值

9、希尔排序算法

基本思想：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

- 操作方法：选择一个增量序列 t_1, t_2, \dots, t_k ，其中 $t_i > t_j, t_k = 1$ ；
- 按增量序列个数 k ，对序列进行 k 趟排序；
- 每趟排序，根据对应的增量 t_i ，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1时，整个序列作为一个表来处理，表长度即为整个序列的长度。



```
private void shellSort(int[] a) {
    int dk = a.length/2;
    while( dk >= 1 ) {
```

```

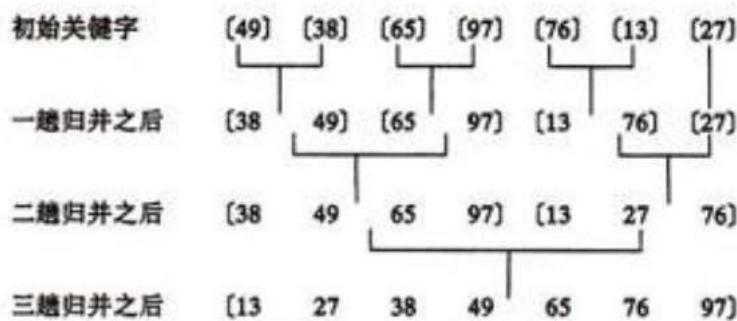
        ShellInsertSort(a, dk);
        dk = dk/2;
    }
}

private void shellInsertSort(int[] a, int dk) {
    //类似插入排序，只是插入排序增量是 1，这里增量是 dk,把 1 换成 dk 就可以了
    for(int i=dk;i<a.length;i++){
        if(a[i]<a[i-dk]){
            int j;
            int x=a[i];//x 为待插入元素
            a[i]=a[i-dk];
            for(j=i-dk; j>=0 && x<a[j];j=j-dk){
                //通过循环，逐个后移一位找到要插入的位置。
                a[j+dk]=a[j];
            }
            a[j+dk]=x;//插入
        }
    }
}

```

10、归并排序算法

归并 (Merge) 排序法是将两个 (或两个以上) 有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。



```

public class MergeSortTest {
    public static void main(String[] args) {
        int[] data = new int[] { 5, 3, 6, 2, 1, 9, 4, 8, 7 };
        print(data);
        mergeSort(data);
        System.out.println("排序后的数组: ");
        print(data);
    }

    public static void mergeSort(int[] data) {
        sort(data, 0, data.length - 1);
    }

    public static void sort(int[] data, int left, int right) {
        if (left >= right)
            return;
        // 找出中间索引
        int center = (left + right) / 2;
        // 对左边数组进行递归
        sort(data, left, center);
        // 对右边数组进行递归
        sort(data, center + 1, right);
        // 合并
        merge(data, left, center, right);
        print(data);
    }

    /**
     * 将两个数组进行归并，归并前面 2 个数组已有序，归并后依然有序
     *
     * @param data
     * 数组对象
     * @param left
     * 左数组的第一个元素的索引
     * @param center
     * 左数组的最后一个元素的索引， center+1 是右数组第一个元素的索引
     * @param right
     * 右数组最后一个元素的索引
     */
    public static void merge(int[] data, int left, int center, int right) {
        // 临时数组
        int[] tmpArr = new int[data.length];
        // 右数组第一个元素索引
        int mid = center + 1;
        // third 记录临时数组的索引
        int third = left;
        // 缓存左数组第一个元素的索引
        int first = left;
    }
}

```

```

int tmp = left;
while (left <= center && mid <= right) {
    // 从两个数组中取出最小的放入临时数组
    if (data[left] <= data[mid]) {
        tmpArr[third++] = data[left++];
    } else {
        tmpArr[third++] = data[mid++];
    }
}
// 剩余部分依次放入临时数组（实际上两个 while 只会执行其中一个）
while (mid <= right) {
    tmpArr[third++] = data[mid++];
}
while (left <= center) {
    tmpArr[third++] = data[left++];
}
// 将临时数组中的内容拷贝回原数组中
// (原 left-right 范围的内容被复制回原数组)
while (tmp <= right) {
    data[tmp] = tmpArr[tmp++];
}
}

public static void print(int[] data) {
    for (int i = 0; i < data.length; i++) {
        System.out.print(data[i] + "\t");
    }
    System.out.println();
}
}

```

11、桶排序算法

桶排序的基本思想是：把数组 arr 划分为 n 个大小相同子区间（桶），每个子区间各自排序，最后合并。计数排序是桶排序的一种特殊情况，可以把计数排序当成每个桶里只有一个元素的情况。

1. 找出待排序数组中的最大值 max、最小值 min
2. 我们使用动态数组 ArrayList 作为桶，桶里放的元素也用 ArrayList 存储。桶的数量为 $(\max - \min) / arr.length + 1$
3. 遍历数组 arr，计算每个元素 arr[i] 放的桶
4. 每个桶各自排序

```

public static void bucketSort(int[] arr){
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for(int i = 0; i < arr.length; i++){
        max = Math.max(max, arr[i]);
        min = Math.min(min, arr[i]);
    }
    // 创建桶
    int bucketNum = (max - min) / arr.length + 1;
    ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketNum);
    for(int i = 0; i < bucketNum; i++){
        bucketArr.add(new ArrayList<Integer>());
    }
    // 将每个元素放入桶
    for(int i = 0; i < arr.length; i++){
        int num = (arr[i] - min) / (arr.length);
        bucketArr.get(num).add(arr[i]);
    }
    // 对每个桶进行排序
    for(int i = 0; i < bucketArr.size(); i++){
        Collections.sort(bucketArr.get(i));
    }
}

```

12、基数排序算法

将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

```

public class radixSort {
    int a[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,101,56,17,18,23,34,15,35,25,53,51};
    public radixSort(){
        sort(a);
        for(int i=0;i<a.length;i++){
            System.out.println(a[i]);
        }
    }
    public void sort(int[] array){
        //首先确定排序的趟数;
        int max=array[0];
        for(int i=1;i<array.length;i++){

```

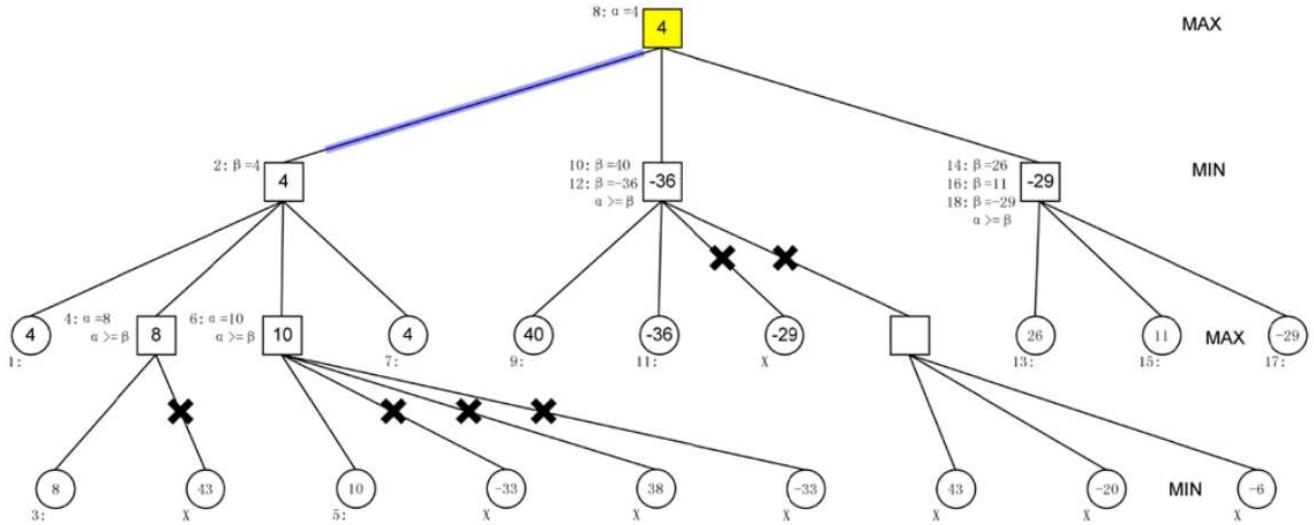
```

if(array[i]>max){
    max=array[i];
}
int time=0;
//判断位数;
while(max>0){
    max/=10;
    time++;
}
//建立 10 个队列;
List<ArrayList> queue=newArrayList<ArrayList>();
for(int i=0;i<10;i++){
    ArrayList<Integer> queue1=new ArrayList<Integer>();
    queue.add(queue1);
}
//进行 time 次分配和收集;
for(int i=0;i<time;i++){
    //分配数组元素;
    for(int j=0;j<array.length;j++){
        //得到数字的第 time+1 位数;
        int x=array[j]%(int)Math.pow(10,i+1)/(int)Math
        ArrayList<Integer> queue2=queue.get(x);
        queue2.add(array[j]);
        queue.set(x, queue2);
    }
    int count=0;//元素计数器;
    //收集队列元素;
    for(int k=0;k<10;k++){
        while(queue.get(k).size()>0){
            ArrayList<Integer> queue3=queue.get(k);
            array[count]=queue3.get(0);
            queue3.remove(0);
            count++;
        }
    }
}
}
}

```

13、剪枝算法

在搜索算法中优化中，剪枝，就是通过某种判断，避免一些不必要的遍历过程，形象的说，就是剪去了搜索树中的某些“枝条”，故称剪枝。应用剪枝优化的核心问题是设计剪枝判断方法，即确定哪些枝条应当舍弃，哪些枝条应当保留的方法。



14、回溯算法

回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。

15、最短路径算法

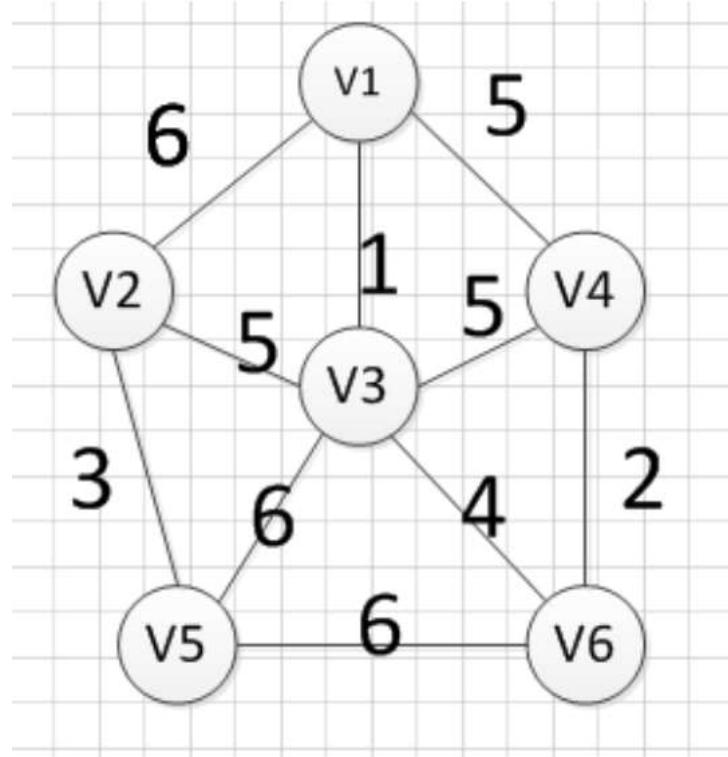
从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径叫做最短路径。解决最短路的问题有以下算法，Dijkstra 算法，Bellman-Ford 算法，Floyd 算法和 SPF 算法等。

16、最小生成树算法

现在假设有一个很实际的问题：我们要在 n 个城市中建立一个通信网络，则连通这 n 个城市需要布置 $n-1$ 条通信线路，这个时候我们需要考虑如何在成本最低的情况下建立这个通信网？

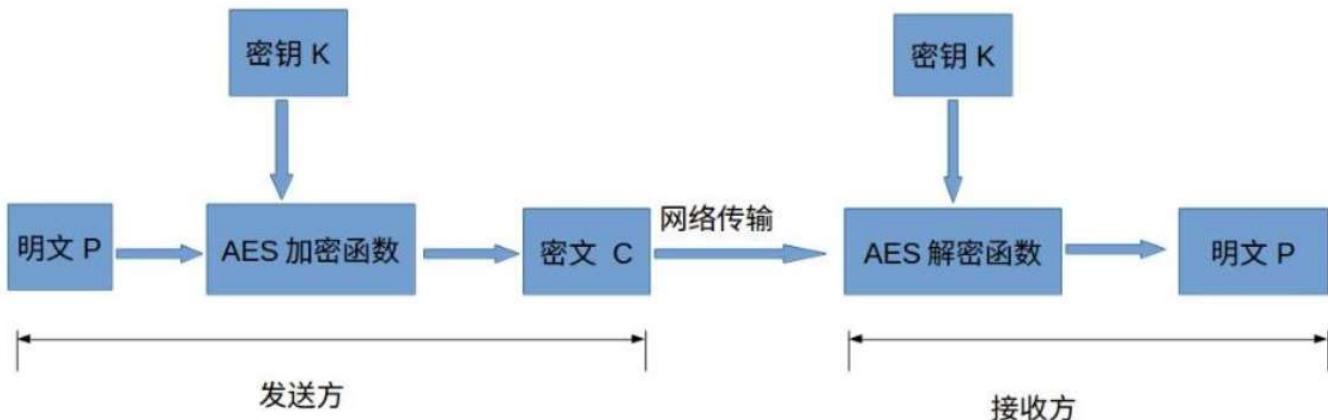
于是我们就可以引入连通图来解决我们遇到的问题， n 个城市就是图上的 n 个顶点，然后，边表示两个城市的通信线路，每条边上的权重就是我们搭建这条线路所需要的成本，所以现在我们有 n 个顶点的连通网可以建立不同的生成树，每一颗生成树都可以作为一个通信网，当我们构造这个连通网所花的成本最小时，搭建该连通网的生成树，就称为最小生成树。

构造最小生成树有很多算法，但是他们都是利用了最小生成树的同一种性质：MST 性质（假设 $N=(V,E)$ 是一个连通网， U 是顶点集 V 的一个非空子集，如果 (u, v) 是一条具有最小权值的边，其中 u 属于 U ， v 属于 $V-U$ ，则必定存在一颗包含边 (u, v) 的最小生成树），下面就介绍两种使用 MST 性质生成最小生成树的算法：普里姆算法和克鲁斯卡尔算法。



17、AES

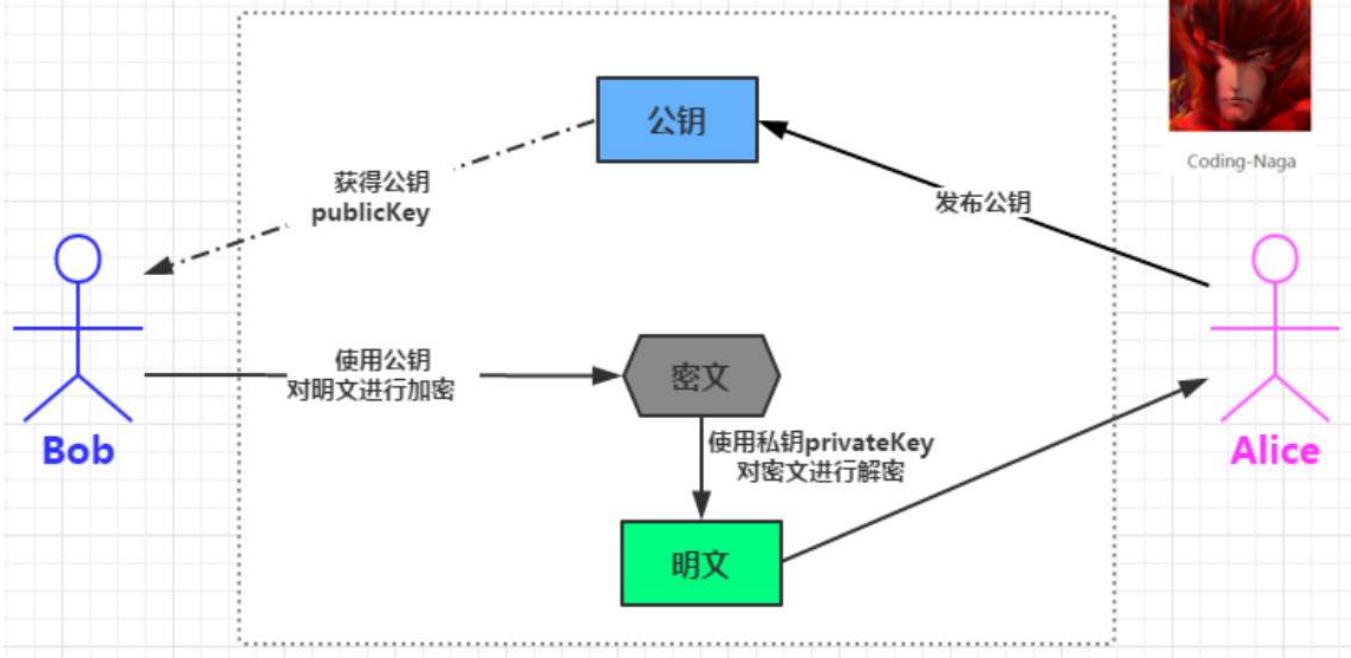
高级加密标准(AES,Advanced Encryption Standard)为最常见的对称加密算法(微信小程序加密传输就是用这个加密算法的)。对称加密算法也就是加密和解密用相同的密钥，具体的加密流程如下图：



18、RSA

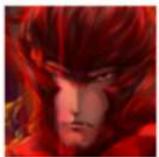
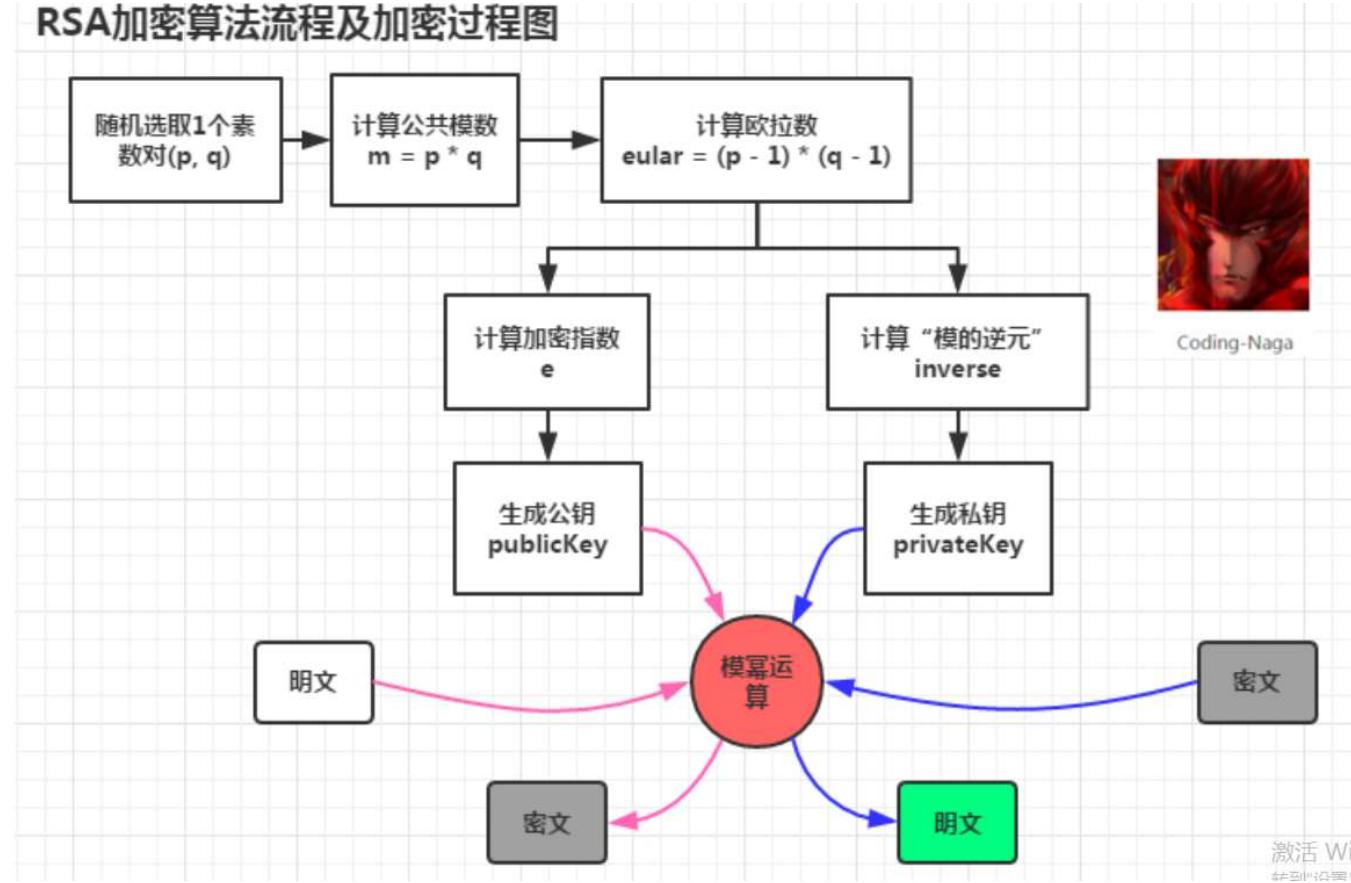
RSA 加密算法是一种典型的非对称加密算法，它基于大数的因式分解数学难题，它也是应用最广泛的非对称加密算法。
非对称加密是通过两个密钥（公钥-私钥）来实现对数据的加密和解密的。公钥用于加密，私钥用于解密。

非对称加密算法流程图



Coding-Naga

RSA加密算法流程及加密过程图



Coding-Naga

19、CRC

循环冗余校验(Cyclic Redundancy Check, CRC)是一种根据网络数据包或电脑文件等数据产生简短固定位数校验码的一种散列函数，主要用来检测或校验数据传输或者保存后可能出现的错误。它是利用除法及余数的原理来作错误侦测的。

20、MD5

MD5 常常作为文件的签名出现，我们在下载文件的时候，常常会看到文件页面上附带一个扩展名为.MD5 的文本或者一行字符，这行字符就是就是把整个文件当作原数据通过 MD5 计算后的值，我们下载文件后，可以用检查文件 MD5 信息的软件对下载到的文件在进行一次计算。两次结果对比就可以确保下载到文件的准确性。另一种常见用途就是网站敏感信息加密，比如用户名密码，支付签名等等。随着 https 技术的普及，现在的网站广泛采用前台明文传输到后台，MD5 加密（使用偏移量）的方式保护敏感数据保护站点和数据安全。

激活 Wi-Fi
并启动“扫描器”

21、更多算法练习

更多算法练习题，请访问 <https://leetcode-cn.com/problemset/algorithms/>

Elasticsearch 面试题

1、elasticsearch 了解多少，说说你们公司 es 的集群架构，索引数据大小，分片有多少，以及一些调优手段。

面试官：想了解应聘者之前公司接触的 ES 使用场景、规模，有没有做过比较大规模的索引设计、规划、调优。

解答：

如实结合自己的实践场景回答即可。

比如：ES 集群架构 13 个节点，索引根据通道不同共 20+ 索引，根据日期，每日递增 20+，索引：10 分片，每日递增 1 亿+ 数据，每个通道每天索引大小控制：150GB 之内。

仅索引层面调优手段：

1.1. 设计阶段调优

- 1、根据业务增量需求，采取基于日期模板创建索引，通过 roll over API 滚动索引；
- 2、使用别名进行索引管理；
- 3、每天凌晨定时对索引做 force_merge 操作，以释放空间；
- 4、采取冷热分离机制，热数据存储到 SSD，提高检索效率；冷数据定期进行 shrink 操作，以缩减存储；
- 5、采取 curator 进行索引的生命周期管理；
- 6、仅针对需要分词的字段，合理的设置分词器；
- 7、Mapping 阶段充分结合各个字段的属性，是否需要检索、是否需要存储等。……

1.2. 写入调优

- 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。

1.3. 查询调优

- 1、禁用 wildcard；
- 2、禁用批量 terms（成百上千的场景）；
- 3、充分利用倒排索引机制，能 keyword 类型尽量 keyword；
- 4、数据量大时候，可以先基于时间敲定索引再检索；
- 5、设置合理的路由机制。

1.4. 其他调优

部署调优，业务调优等。

上面的提及一部分，面试者就基本对你之前的实践或者运维经验有所评估了。

2、elasticsearch 的倒排索引是什么

面试官：想了解你对基础概念的认知。

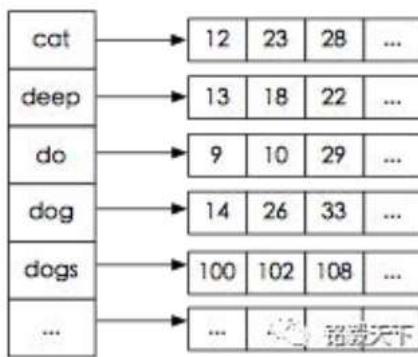
解答：通俗解释一下就可以。

传统的我们的检索是通过文章，逐个遍历找到对应关键词的位置。

而倒排索引，是通过分词策略，形成了词和文章的映射关系表，这种词典+映射表即为倒排索引。

有了倒排索引，就能实现 $O(1)$ 时间复杂度的效率检索文章了，极大的提高了检索效率

term字典 倒排docId表



学术的解答方式：

倒排索引，相反于一篇文章包含了哪些词，它从词出发，记载了这个词在哪些文档中出现过，由两部分组成——词典和倒排表。

加分项：倒排索引的底层实现是基于：FST (Finite State Transducer) 数据结构。

lucene 从 4+版本后开始大量使用的数据结构是 FST。FST 有两个优点：

1、空间占用小、通过对词典中单词前缀和后缀的重复利用，压缩了存储空间；

2、查询速度快。O(len(str))的查询时间复杂度。

3、elasticsearch 索引数据多了怎么办，如何调优，部署

面试官：想了解大数据量的运维能力。

解答：索引数据的规划，应在前期做好规划，正所谓“设计先行，编码在后”，这样才能有效的避免突如其来的数据激增导致集群处理能力不足引发的线上客户检索或者其他业务受到影响。

如何调优，正如问题 1 所说，这里细化一下：

3.1 动态索引层面

基于模板+时间+rollover api 滚动创建索引，举例：设计阶段定义：blog 索引的模板格式为：blog_index_时间戳的形式，每天递增数据。

这样做的好处：不至于数据量激增导致单个索引数据量非常大，接近于上线 2 的 32 次幂-1，索引存储达到了 TB+甚至更大。

一旦单个索引很大，存储等各种风险也随之而来，所以要提前考虑+及早避免。

3.2 存储层面

冷热数据分离存储，热数据（比如最近 3 天或者一周的数据），其余为冷数据。对于冷数据不会再写入新数据，可以考虑定期 force_merge 加 shrink 压缩操作，节省存储空间和检索效率。

3.3 部署层面

一旦之前没有规划，这里就属于应急策略。结合 ES 自身的支持动态扩展的特点，动态新增机器的方式可以缓解集群压力，注意：如果之前主节点等规划合理，不需要重启集群也能完成动态新增的。

4、elasticsearch 是如何实现 master 选举的

面试官：想了解 ES 集群的底层原理，不再只关注业务层面了。

解答：

前置前提：

- 只有候选主节点 (master : true) 的节点才能成为主节点。
- 最小主节点数 (min_master_nodes) 的目的是防止脑裂。

这个我看了各种网上分析的版本和源码分析的书籍，云里雾里。

核对了一下代码，核心入口为 findMaster，选择主节点成功返回对应 Master，否则返回 null。选举流程大致描述如下：

第一步：确认候选主节点数达标，elasticsearch.yml 设置的值

discovery.zen.minimum_master_nodes ;

第二步：比较：先判定是否具备 master 资格，具备候选主节点资格的优先返回；若两节点都为候选主节点，则 id 小的值会主节点。注意这里的 id 为 string 类型。题外话：获取节点 id 的方法。

1GET /_cat/nodes?v&h=ip,port,heapPercent,heapMax,id,name

2ip port heapPercent heapMax id name

5、详细描述一下 Elasticsearch 索引文档的过程

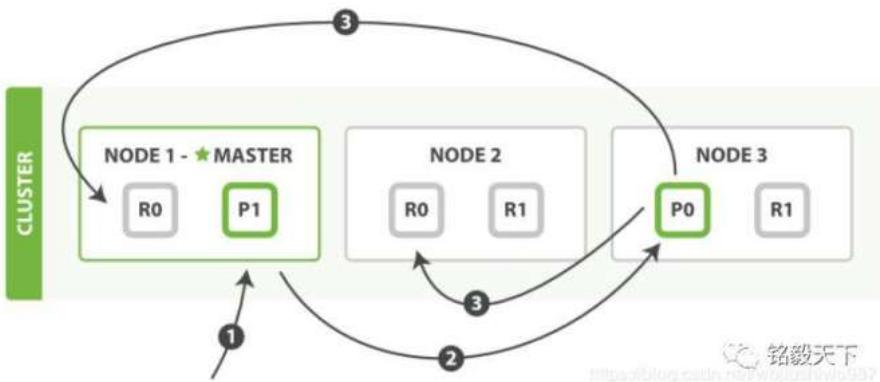
面试官：想了解 ES 的底层原理，不再只关注业务层面了。

解答：

这里的索引文档应该理解为文档写入 ES，创建索引的过程。

文档写入包含：单文档写入和批量 bulk 写入，这里只解释一下：单文档写入流程。

记住官方文档中的这个图。



第一步：客户写集群某节点写入数据，发送请求。（如果没有指定路由/协调节点，请求的节点扮演路由节点的角色。）

第二步：节点 1 接受到请求后，使用文档_id 来确定文档属于分片 0。请求会被转到另外的节点，假定节点 3。因此分片 0 的主分片分配到节点 3 上。

第三步：节点 3 在主分片上执行写操作，如果成功，则将请求并行转发到节点 1 和节点 2 的副本分片上，等待结果返回。所有的副本分片都报告成功，节点 3 将向协调节点（节点 1）报告成功，节点 1 向请求客户端报告写入成功。

如果面试官再问：第二步中的文档获取分片的过程？

回答：借助路由算法获取，路由算法就是根据路由和文档 id 计算目标的分片 id 的过程。

```
1shard = hash(_routing) % (num_of_primary_shards)
```

6、详细描述一下 Elasticsearch 搜索的过程？

面试官：想了解 ES 搜索的底层原理，不再只关注业务层面了。

解答：

搜索拆解为“query then fetch”两个阶段。

query 阶段的目的：定位到位置，但不取。

步骤拆解如下：

- 1、假设一个索引数据有 5 主+1 副本 共 10 分片，一次请求会命中（主或者副本分片中）的一个。
 - 2、每个分片在本地进行查询，结果返回到本地有序的优先队列中。
 - 3、第 2) 步骤的结果发送到协调节点，协调节点产生一个全局的排序列表。fetch 阶段的目的：取数据。
- 路由节点获取所有文档，返回给客户端。

7、Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法

面试官：想了解对 ES 集群的运维能力。

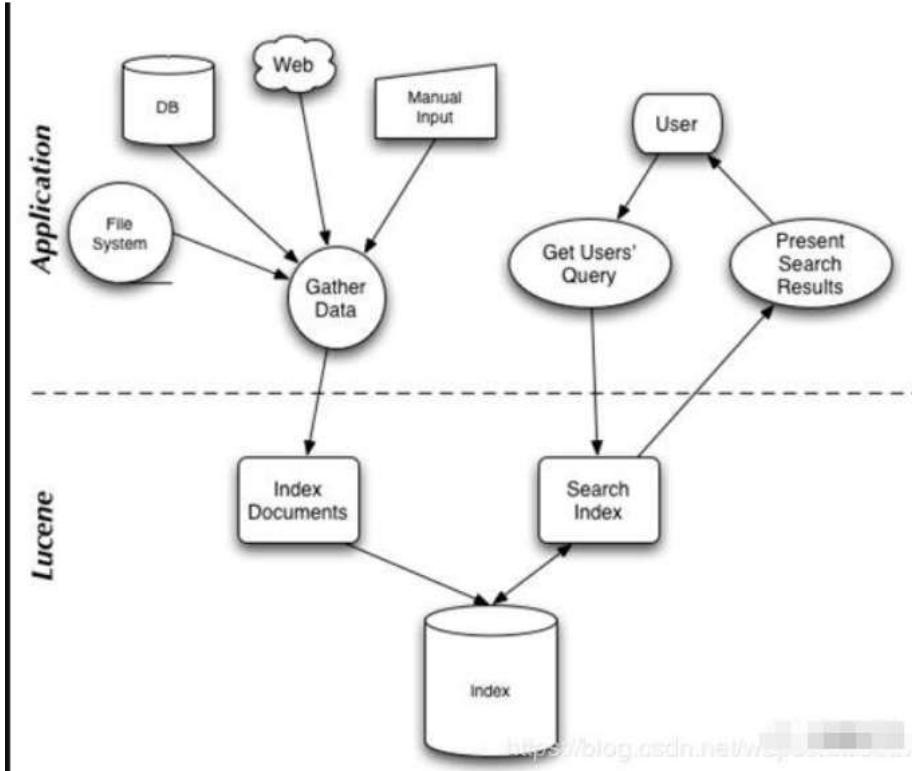
解答：

- 1、关闭缓存 swap；
- 2、堆内存设置为：Min (节点内存/2, 32GB)；
- 3、设置最大文件句柄数；
- 4、线程池+队列大小根据业务需要做调整；
- 5、磁盘存储 raid 方式——存储有条件使用 RAID10，增加单节点性能以及避免单节点存储故障。

8、Lucence 内部结构是什么？

面试官：想了解你的知识面的广度和深度。

解答：



Lucene 是有索引和搜索的两个过程，包含索引创建，索引，搜索三个要点。可以基于这个脉络展开一些。

最近面试一些公司，被问到的关于 Elasticsearch 和搜索引擎相关的问题，以及自己总结的回答。

9、Elasticsearch 是如何实现 Master 选举的？

- 1、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；
- 2、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。
- 3、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。
- 4、补充：master 节点的职责主要包括集群、节点和索引的管理，不负责文档级别的管理；data 节点可以关闭 http 功能*。

10、Elasticsearch 中的节点（比如共 20 个），其中的 10 个选了一个 master，另外 10 个选了另一个 master，怎么办？

- 1、当集群 master 候选数量不小于 3 个时，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；
- 2、当候选数量为两个时，只能修改为唯一的一个 master 候选，其他作为 data 节点，避免脑裂问题

11、客户端在和集群连接时，如何选择特定的节点执行请求的？

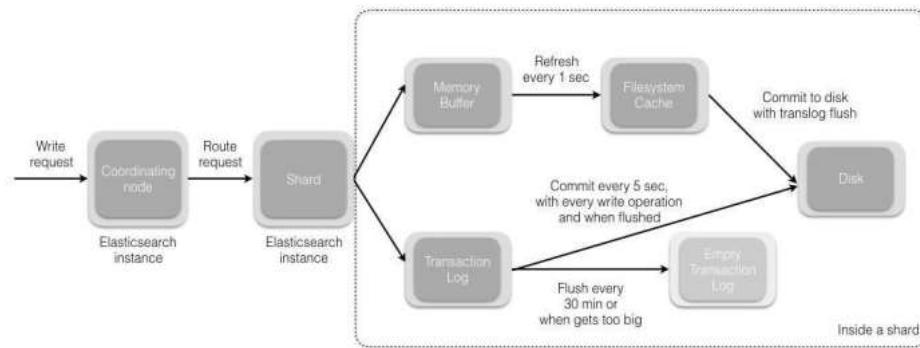
- 1、TransportClient 利用 transport 模块远程连接一个 elasticsearch 集群。它并不加入到集群中，只是简单的获得一个或者多个初始化的 transport 地址，并以 轮询 的方式与这些地址进行通信。

12、详细描述一下 Elasticsearch 索引文档的过程。

协调节点默认使用文档 ID 参与计算（也支持通过 routing），以便为路由提供合适的分片。

```
shard = hash(document_id) % (num_of_primary_shards)
```

- 1、当分片所在的节点接收到来自协调节点的请求后，会将请求写入到 Memory Buffer，然后定时（默认是每隔 1 秒）写入到 Filesystem Cache，这个从 MemoryBuffer 到 Filesystem Cache 的过程就叫做 refresh；
- 2、当然在某些情况下，存在 Memory Buffer 和 Filesystem Cache 的数据可能会丢失，ES 是通过 translog 的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到 translog 中，当 Filesystem cache 中的数据写入到磁盘中时，才会清除掉，这个过程叫做 flush；
- 3、在 flush 过程中，内存中的缓冲将被清除，内容被写入一个新段，段的 fsync 将创建一个新的提交点，并将内容刷新到磁盘，旧的 translog 将被删除并开始一个新的 translog。
- 4、flush 触发的时机是定时触发（默认 30 分钟）或者 translog 变得太大（默认为 512M）时



补充：关于 Lucene 的 Segement：

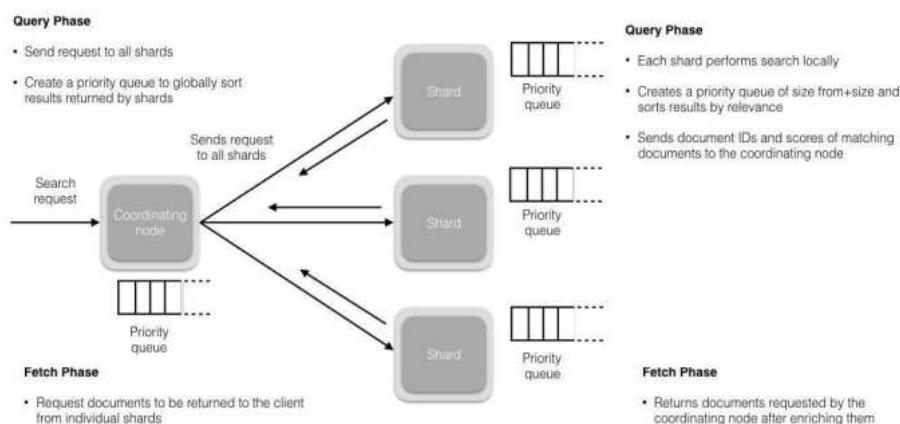
- 1、Lucene 索引是由多个段组成，段本身是一个功能齐全的倒排索引。
- 2、段是不可变的，允许 Lucene 将新的文档增量地添加到索引中，而不用从头重建索引。
- 3、对于每一个搜索请求而言，索引中的所有段都会被搜索，并且每个段会消耗CPU 的时钟周、文件句柄和内存。这意味着段的数量越多，搜索性能会越低。
- 4、为了解决这个问题，Elasticsearch 会合并小段到一个较大的段，提交新的合并段到磁盘，并删除那些旧的小段。

13、详细描述一下 Elasticsearch 更新和删除文档的过程。

- 1、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；
- 2、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。
- 3、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。

14、详细描述一下 Elasticsearch 搜索的过程

- 1、搜索被执行成一个两阶段过程，我们称之为 Query Then Fetch；
- 2、在初始查询阶段时，查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。每个分片在本地执行搜索并构建一个匹配文档的大小为 from + size 的优先队列。PS：在搜索的时候是会查询 Filesystem Cache 的，但是有部分数据还在 Memory Buffer，所以搜索是近实时的。
- 3、每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。
- 4、接下来就是取回阶段，协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分片加载并丰富文档，如果有需要的话，接着返回文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。
- 5、补充：Query Then Fetch 的搜索类型在文档相关性打分的时候参考的是本分片的数据，这样在文档数量较少的时候可能不够准确，DFS Query Then Fetch 增加了一个预查询的处理，询问 Term 和 Document frequency，这个评分更准确，但是性能会变差。*



15、在 Elasticsearch 中，是怎么根据一个词找到对应的倒排索引的？

SEE :

- Lucene 的索引文件格式(1)
- Lucene 的索引文件格式(2)

16、Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法？

- 1、64 GB 内存的机器是非常理想的，但是 32 GB 和 16 GB 机器也是很常见的。少于 8 GB 会适得其反。
- 2、如果你要在更快的 CPUs 和更多的核心之间选择，选择更多的核心更好。多个内核提供的额外并发远胜过稍微快一点点的时钟频率。
- 3、如果你负担得起 SSD，它将远远超出任何旋转介质。基于 SSD 的节点，查询和索引性能都有提升。如果你负担得起，SSD 是一个好的选择。
- 4、即使数据中心们近在咫尺，也要避免集群跨越多个数据中心。绝对要避免集群跨越大的地理距离。
- 5、请确保运行你应用程序的 JVM 和服务器的 JVM 是完全一样的。在 Elasticsearch 的几个地方，使用 Java 的本地序列化。

- 6、通过设置 gateway.recover_after_nodes、gateway.expected_nodes、gateway.recover_after_time 可以在集群重启的时候避免过多的分片交换，这可能会让数据恢复从数个小时缩短为几秒钟。
- 7、Elasticsearch 默认被配置为使用单播发现，以防止节点无意中加入集群。只有在同一台机器上运行的节点才会自动组成集群。最好使用单播代替组播。
- 8、不要随意修改垃圾回收器（CMS）和各个线程池的大小。
- 9、把你的内存的（少于）一半给 Lucene（但不要超过 32 GB！），通过 ES_HEAP_SIZE 环境变量设置。
- 10、内存交换到磁盘对服务器性能来说是致命的。如果内存交换到磁盘上，一个 100 微秒的操作可能变成 10 毫秒。再想想那么多 10 微秒的操作时延累加起来。不难看出 swapping 对于性能是多么可怕。
- 11、Lucene 使用了大量的文件。同时，Elasticsearch 在节点和 HTTP 客户端之间进行通信也使用了大量的套接字。所有这一切都需要足够的文件描述符。你应该增加你的文件描述符，设置一个很大的值，如 64,000。

补充：索引阶段性能提升方法

- 1、使用批量请求并调整其大小：每次批量数据 5–15 MB 大是个不错的起始点。
- 2、存储：使用 SSD
- 3、段和合并：Elasticsearch 默认值是 20 MB/s，对机械磁盘应该是个不错的设置。如果你用的是 SSD，可以考虑提高到 100–200 MB/s。如果你在做批量导入，完全不在意搜索，你可以彻底关掉合并限流。另外还可以增加 index.translog.flush_threshold_size 设置，从默认的 512 MB 到更大一些的值，比如 1 GB，这可以在一次清空触发的时候在事务日志里积累出更大的段。
- 4、如果你的搜索结果不需要近实时的准确度，考虑把每个索引的 index.refresh_interval 改到 30s。
- 5、如果你在做大批量导入，考虑通过设置 index.number_of_replicas: 0 关闭副本。

17、对于 GC 方面，在使用 Elasticsearch 时要注意什么？

- 1、SEE：<https://elasticsearch.cn/article/32>
- 2、倒排词典的索引需要常驻内存，无法 GC，需要监控 data node 上 segment memory 增长趋势。
- 3、各类缓存，field cache, filter cache, indexing cache, bulk queue 等等，要设置合理的大小，并且要根据最坏的情况来看 heap 是否够用，也就是各类缓存全部占满的时候，还有 heap 空间可以分配给其他任务吗？避免采用 clear cache 等“自欺欺人”的方式来释放内存。
- 4、避免返回大量结果集的搜索与聚合。确实需要大量拉取数据的场景，可以采用 scan & scroll API 来实现。
- 5、cluster stats 驻留内存并无法水平扩展，超大规模集群可以考虑分拆成多个集群通过 tribe node 连接。
- 6、想知道 heap 够不够，必须结合实际应用场景，并对集群的 heap 使用情况做持续的监控。

18、Elasticsearch 对于大数据量（上亿量级）的聚合如何实现？

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。

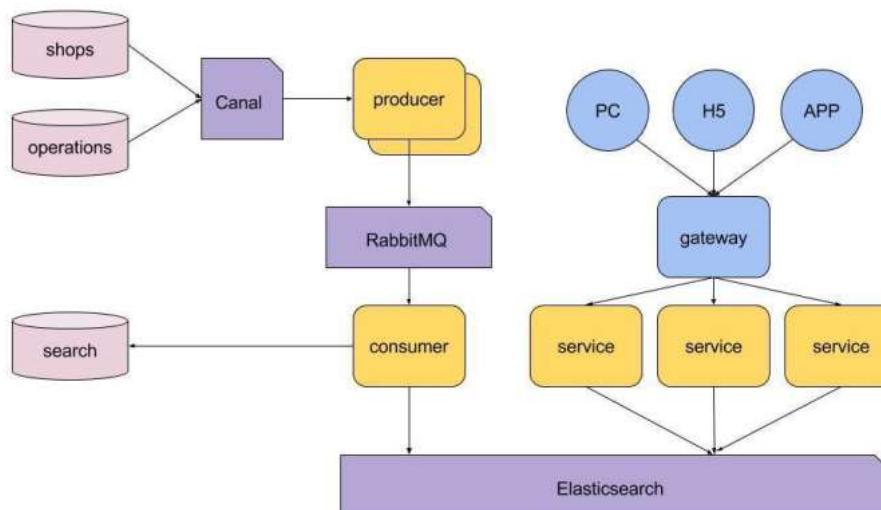
19、在并发情况下，Elasticsearch 如何保证读写一致？

- 1、可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；
- 2、另外对于写操作，一致性级别支持 quorum/one/all，默认认为 quorum，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。
- 3、对于读操作，可以设置 replication 为 sync（默认），这使得操作在主分片和副本分片都完成后才会返回；如果设置 replication 为 async 时，也可以通过设置搜索请求参数 _preference 为 primary 来查询主分片，确保文档是最新版本。

20、如何监控 Elasticsearch 集群状态？

Marvel 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。

21、介绍下你们电商搜索的整体技术架构



22、介绍一下你们的个性化搜索方案？

SEE 基于 word2vec 和 Elasticsearch 实现个性化搜索

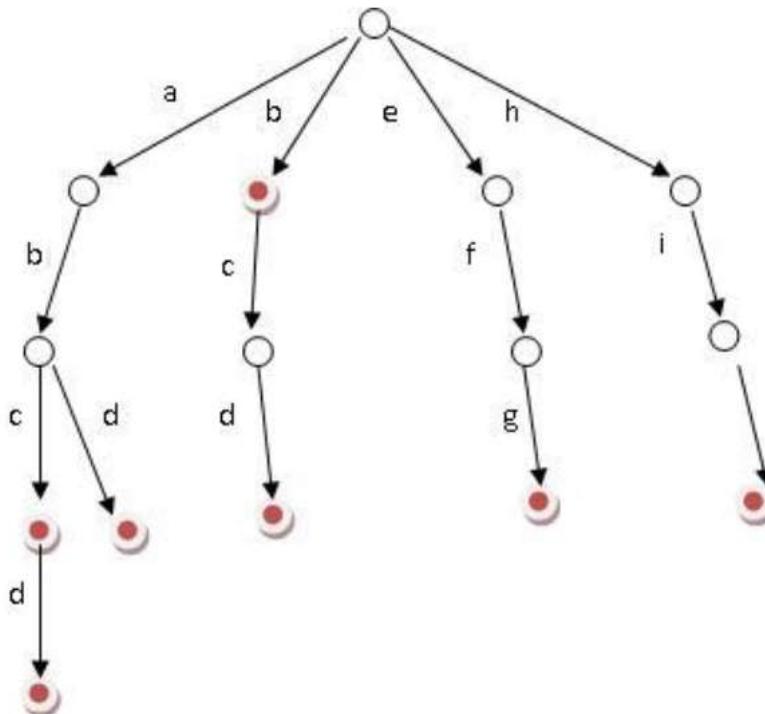
23、是否了解字典树？

常用字典数据结构如下所示

数据结构	优缺点
排序列表Array/ArrayList	使用二分法查找，不平衡
HashMap/TreeMap	性能高，内存消耗大，几乎是原始数据的三倍
Skip List	跳跃表，可快速查找词语，在lucene、redis、Hbase等均有实现。相对于TreeMap等结构，特别适合高并发场景（Skip List介绍）
Trie	适合英文词典，如果系统中存在大量字符串且这些字符串基本没有公共前缀，则相应的trie树将非常消耗内存（数据结构之trie树）
Double Array Trie	适合做中文词典，内存占用小，很多分词工具均采用此种算法（深入双数组Trie）
Ternary Search Tree	三叉树，每一个node有3个节点，兼具省空间和查询快的优点（Ternary Search Tree）
Finite State Transducers (FST)	一种有限状态转移机，Lucene 4有开源实现，并大量使用

Trie 的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。它有 3 个基本性质：

- 1、根节点不包含字符，除根节点外每一个节点都只包含一个字符。
- 2、从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
- 3、每个节点的所有子节点包含的字符都不相同。



- 1、可以看到，trie 树每一层的节点数是 26^i 级别的。所以为了节省空间，我们还可以用动态链表，或者用数组来模拟动态。而空间的花费，不会超过单词数×单词长度。
- 2、实现：对每个结点开一个字母集大小的数组，每个结点挂一个链表，使用左儿子右兄弟表示法记录这棵树；
- 3、对于中文的字典树，每个节点的子节点用一个哈希表存储，这样就不用浪费太大的空间，而且查询速度上可以保留哈希的复杂度 O(1)

24、拼写纠错是如何实现的？

- 1、拼写纠错是基于编辑距离来实现；编辑距离是一种标准的方法，它用来表示经过插入、删除和替换操作从一个字符串转换到另外一个字符串的最小操作步数；
- 2、编辑距离的计算过程：比如要计算 batyu 和 beauty 的编辑距离，先创建一个 7×8 的表（batyu 长度为 5，coffee 长度为 6，各加 2），接着，在如下位置填入黑色数字。其他格的计算过程是取以下三个值的最小值：
如果最上方的字符等于最左方的字符，则为左上方的数字。否则为左上方的数字
+1。（对于 3,3 来说为 0）
左方数字+1（对于 3,3 格来说为 2）
上方数字+1（对于 3,3 格来说为 2）
最终取右下角的值即为编辑距离的值 3。

		b	e	a	u	t	y
	0	1	2	3	4	5	6
b	1	0	1	2	3	4	5
a	2	1	1	1	2	3	4
t	3	2	2	2	2	2	3
y	4	3	3	3	3	3	2
u	5	4	4	4	3	4	3

对于拼写纠错，我们考虑构造一个度量空间（Metric Space），该空间内任何关系满足以下三条基本条件：

$d(x,y) = 0$ -- 假如 x 与 y 的距离为 0，则 $x=y$

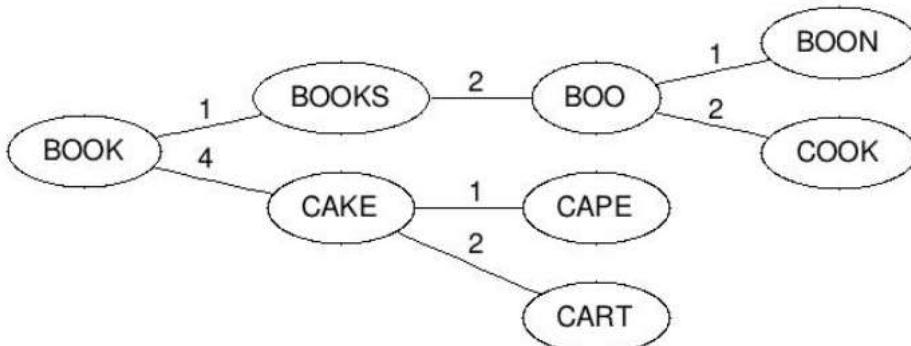
$d(x,y) = d(y,x)$ -- x 到 y 的距离等同于 y 到 x 的距离

$d(x,y) + d(y,z) \geq d(x,z)$ -- 三角不等式

1、根据三角不等式，则满足与 query 距离在 n 范围内的另一个字符转 B ，其与 A 的距离最大为 $d+n$ ，最小为 $d-n$ 。

2、BK 树的构造过程如下：每个节点有任意个子节点，每条边有个值表示编辑距离。所有子节点到父节点的边上标注 n 表示编辑距离恰好为 n 。比如，我们有棵树父节点是“book”和两个子节点“cake”和“books”，“book”到“books”的边标号 1，“book”到“cake”的边标号 4。从字典里构造好树后，无论何时你想插入新单词时，计算该单词与根节点的编辑距离，并且查找数值为 $d(\text{newword}, \text{root})$ 的边。递归得与各子节点进行比较，直到没有子节点，你就可以创建新的子节点并将新单词保存在那。比如，插入“boo”到刚才上述例子的树中，我们先检查根节点，查找 $d(\text{"book"}, \text{"boo"}) = 1$ 的边，然后检查标号为 1 的边的子节点，得到单词“books”。我们再计算距离 $d(\text{"books"}, \text{"boo"}) = 2$ ，则将新单词插在“books”之后，边标号为 2。

3、查询相似词如下：计算单词与根节点的编辑距离 d ，然后递归查找每个子节点标号为 $d-n$ 到 $d+n$ （包含）的边。假如被检查的节点与搜索单词的距离 d 小于 n ，则返回该节点并继续查询。比如输入 cape 且最大容忍距离为 1，则先计算和根的编辑距离 $d(\text{"book"}, \text{"cape"}) = 4$ ，然后接着找和根节点之间编辑距离为 3 到 5 的，这个就找到了 cake 这个节点，计算 $d(\text{"cake"}, \text{"cape"}) = 1$ ，满足条件所以返回 cake，然后再找和 cake 节点编辑距离是 0 到 2 的，分别找到 cape 和 cart 节点，这样就得到 cape 这个满足条件的结果。

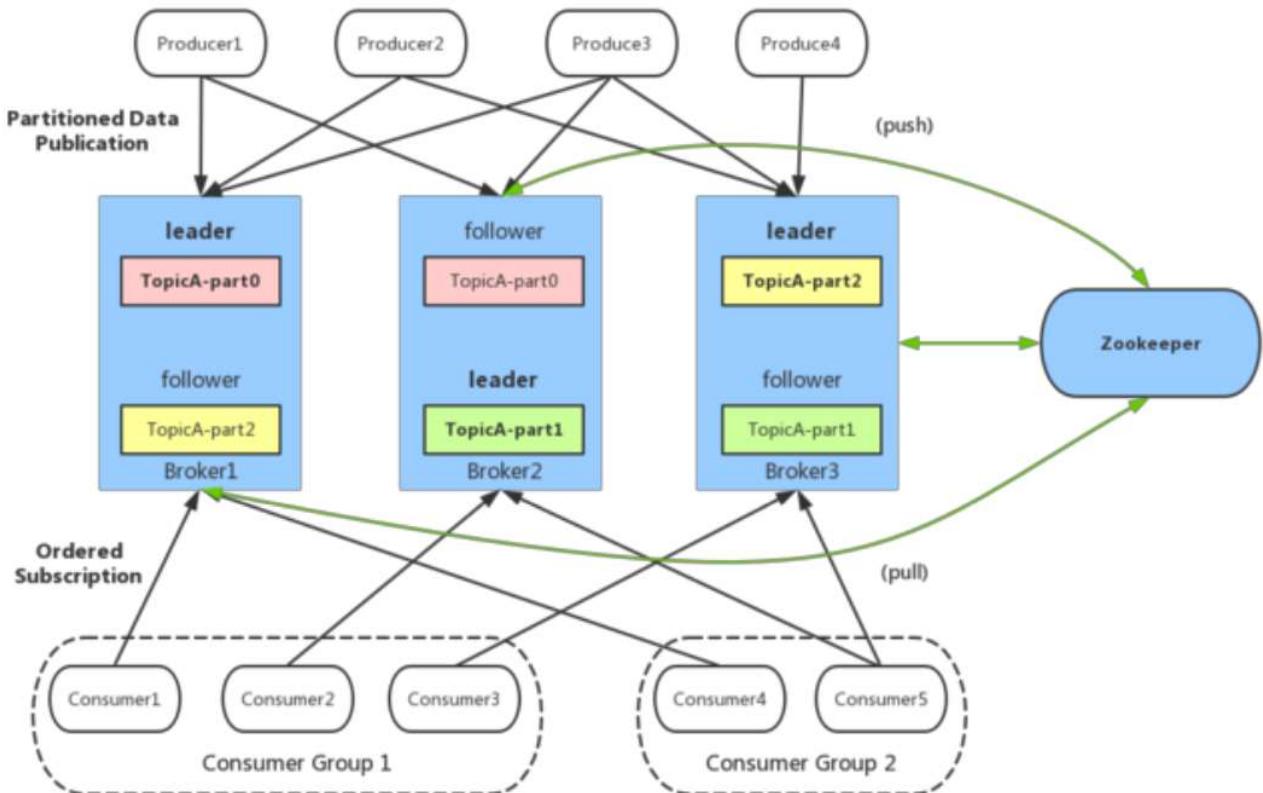


Kafka 面试题

1、Kafka 是什么

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。

1. broker : Kafka 服务器，负责消息存储和转发
2. topic : 消息类别，Kafka 按照 topic 来分类消息
3. 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
4. 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
5. Producer : 消息生产者
6. Consumer : 消息消费者
7. Consumer Group : 消费者分组，每个 Consumer 必须属于一个 group
8. Zookeeper : 保存着集群 broker、topic、partition 等 meta 数据；另外，还负责 broker 故障发现，partition leader 选举，负载均衡等功能

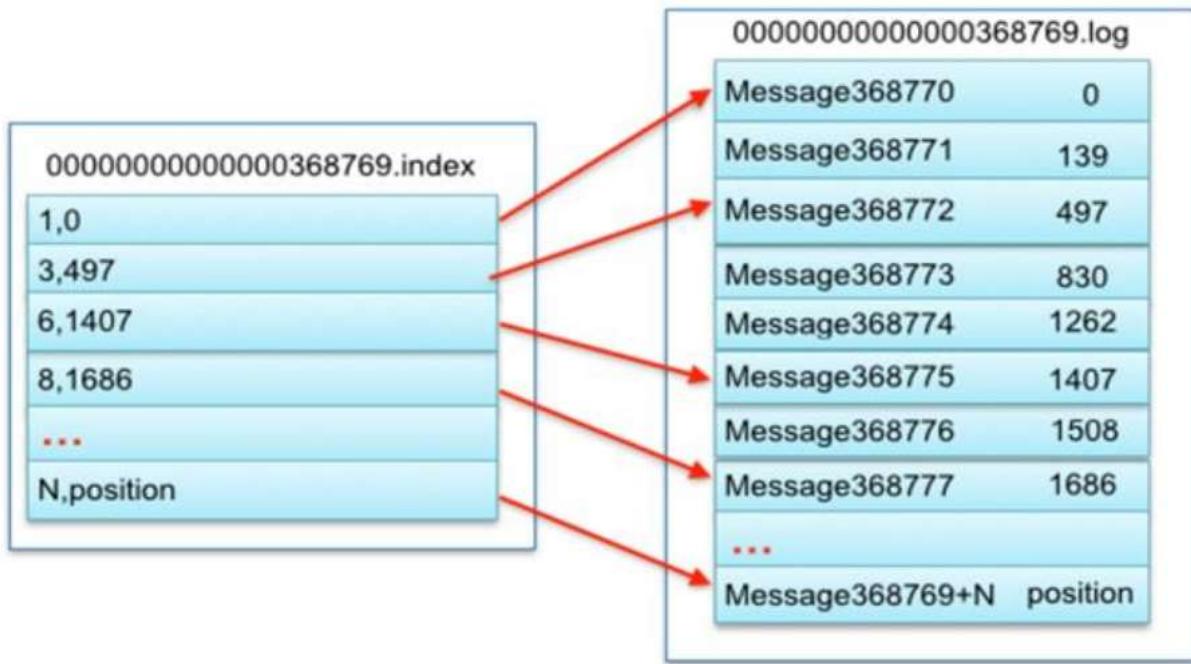


2、partition 的数据文件 (offset , MessageSize , data)

partition 中的每条 Message 包含了以下三个属性：offset，MessageSize，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。

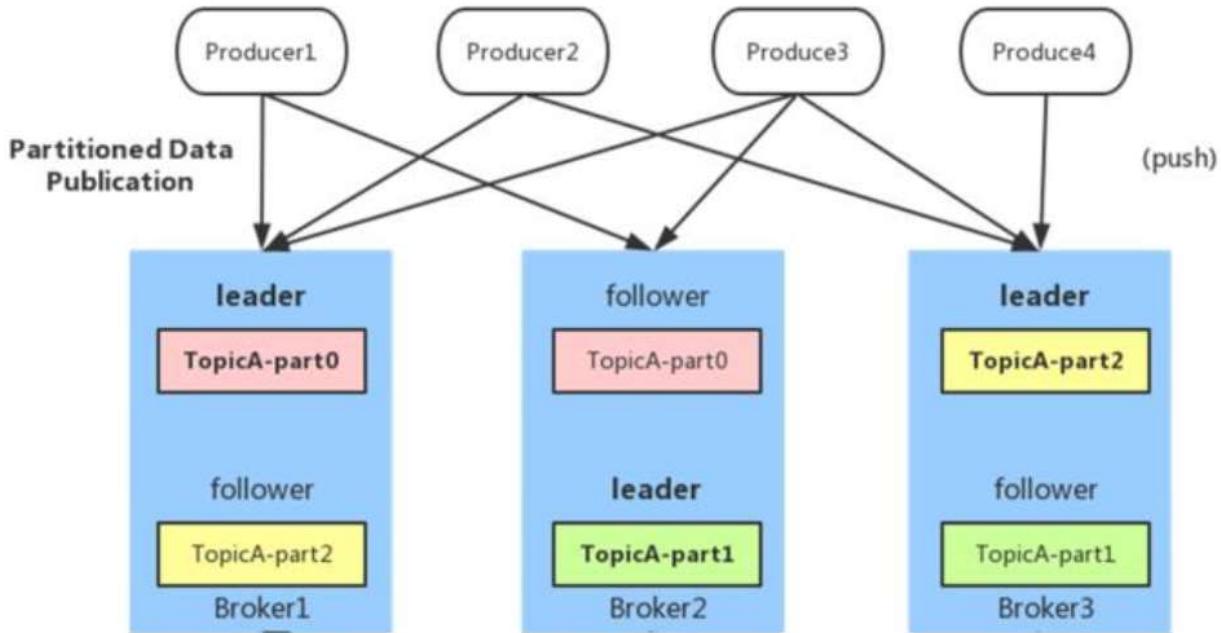
3、数据文件分段 segment (顺序读写、分段命令、二分查找)

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。



4、负载均衡 (partition 会均衡分布到不同 broker 上)

由于消息 topic 由多个 partition 组成，且 partition 会均衡分布到不同 broker 上，因此，为了有效利用 broker 集群的性能，提高消息的吞吐量，producer 可以通过随机或者 hash 等方式，将消息平均发送到多个 partition 上，以实现负载均衡。



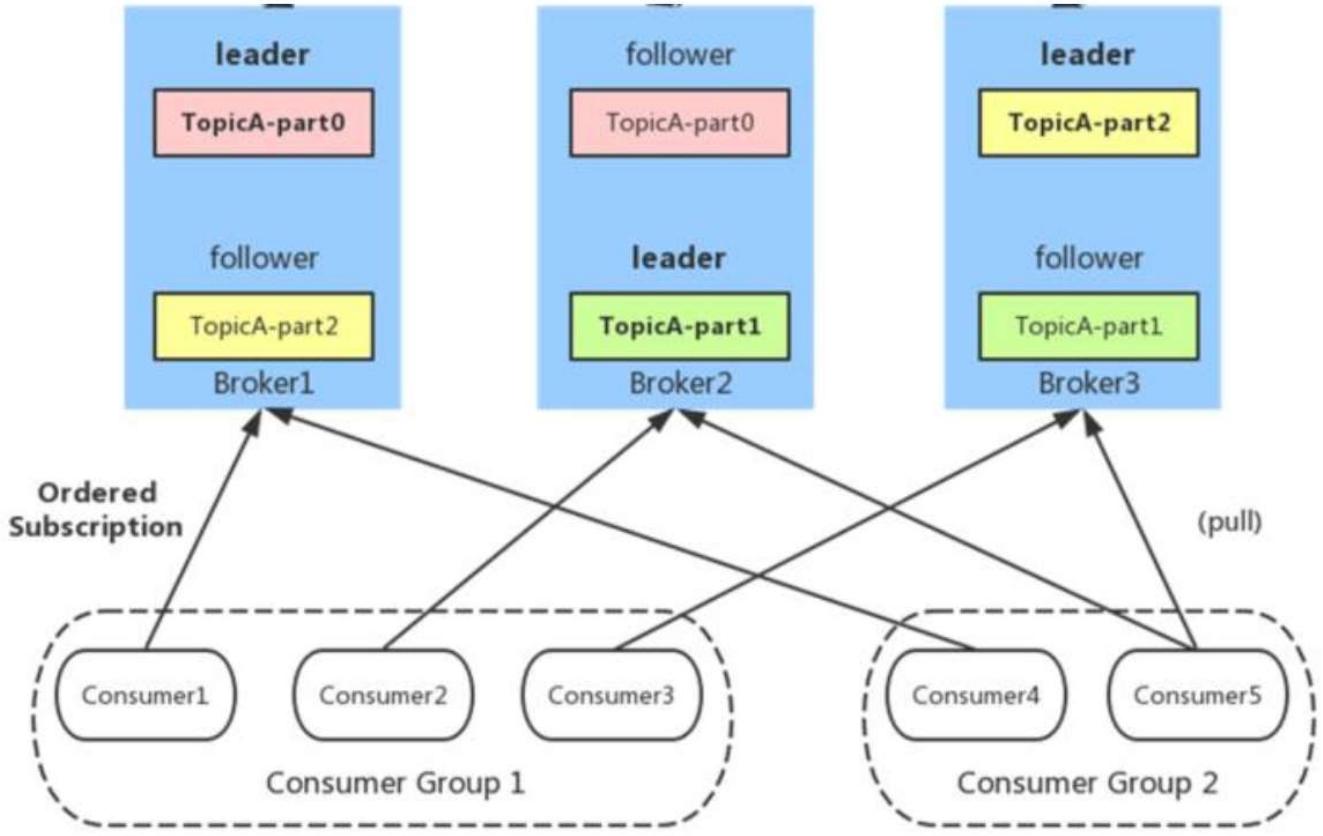
5、批量发送

是提高消息吞吐量重要的方式，Producer 端可以在内存中合并多条消息后，以一次请求的方式发送了批量的消息给 broker，从而大大减少 broker 存储消息的 IO 操作次数。但也一定程度上影响了消息的实时性，相当于以时延代价，换取更好的吞吐量。

6、压缩 (GZIP 或 Snappy)

Producer 端可以通过 GZIP 或 Snappy 格式对消息集合进行压缩。Producer 端进行压缩之后，在Consumer 端需进行解压。压缩的好处就是减少传输的数据量，减轻对网络传输的压力，在对大数据处理上，瓶颈往往体现在网络上而不是 CPU (压缩和解压会耗掉部分 CPU 资源)。

7、消费者设计



8、Consumer Group

同一 Consumer Group 中的多个 Consumer 实例，不同时消费同一个 partition，等效于队列模式。partition 内消息是有序的，Consumer 通过 pull 方式消费消息。Kafka 不删除已消费的消息对于 partition，顺序读写磁盘数据，以时间复杂度 O(1)方式提供消息持久化能力。

9、如何获取 topic 主题的列表

```
bin/kafka-topics.sh --list --zookeeper localhost:2181
```

10、生产者和消费者的命令行是什么？

生产者在主题上发布消息：

```
bin/kafka-console-producer.sh --broker-list 192.168.43.49:9092 --topic
Hello-Kafka
```

注意这里的 IP 是 server.properties 中的 listeners 的配置。接下来每个新行就是输入一条新消息。

消费者接受消息：

```
bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic
Hello-Kafka --from-beginning
```

11、consumer 是推还是拉？

Kafka 最初考虑的问题是，customer 应该从 brokers 拉取消息还是 brokers 将消息推送到 consumer，也就是 pull 还 push。在这方面，Kafka 遵循了一种大部分消息系统共同的传统设计：producer 将消息推送到 broker，consumer 从 broker 拉取消息。

一些消息系统比如 Scribe 和 Apache Flume 采用了 push 模式，将消息推送到下游的 consumer。这样做有好处也有坏处：由 broker 决定消息推送的速率，对于不同消费速率的 consumer 就不太好处理了。消息系统都致力于让 consumer 以最大的速率最快速的消费消息，但不幸的是，push 模式下，当 broker 推送的速率远大于 consumer 消费的速率时，consumer 恐怕就要崩溃了。最终 Kafka 还是选取了传统的 pull 模式。

Pull 模式的另外一个好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。Push 模式必须在不知道下游 consumer 消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免 consumer 崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull 模式下，consumer 就可以根据自己的消费能力去决定这些策略。

Pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询，直到新消息到 t 达。为了避免这点，Kafka 有个参数可以让 consumer 阻塞知道新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发送)。

12、讲讲 kafka 维护消费状态跟踪的方法

大部分消息系统在 broker 端的维护消息被消费的记录：一个消息被分发到 consumer 后 broker 就马上进行标记或者等待 customer 的通知后进行标记。这样也可以在消息在消费后立马就删除以减少空间占用。

但是这样会不会有什么问题呢？如果一条消息发送出去之后就立即被标记为消费过的，一旦 consumer 处理消息时失败了（比如程序崩溃）消息就丢失了。为了解决这个问题，很多消息系统提供了另外一个个功能：当消息被发送出去之后仅仅被标记为已发送状态，当接到 consumer 已经消费成功的通知后才标记为已被消费的状态。这虽然解决了消息丢失的问题，但产生了新问题，首先如果 consumer 处理消息成功了但是向 broker 发送响应时失败了，这条消息将被消费两次。第二个问题时，broker 必须维护每条消息的状态，并且每次都要先锁住消息然后更改状态然后释放锁。这样麻烦又来了，且不说要维护大量的状态数据，比如如果消息发送出去但没有收到消费成功的通知，这条消息将一直处于被锁定的状态，Kafka 采用了不同的策略。Topic 被分成了若干分区，每个分区在同一时间只被一个 consumer 消费。这意味着每个分区被消费的消息在日志中的位置仅仅是一个简单的整数：offset。这样就很容易标记每个分区消费状态就很容易了，仅仅需要一个整数而已。这样消费状态的跟踪就很简单了。这带来了另外一个好处：consumer 可以把 offset 调成一个较老的值，去重新消费老的消息。这对传统的消息系统来说看起来有些不可思议，但确实是非常有用的，谁规定了一条消息只能被消费一次呢？

13、讲一下主从同步

<https://blog.csdn.net/honglei915/article/details/37565289>

14、为什么需要消息系统，mysql 不能满足需求吗？

1.解耦：

允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

2.冗余：

消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的“插入-获取-删除”范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

3.扩展性：

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。

4.灵活性 & 峰值处理能力：

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果以为能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

5.可恢复性：

系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

6.顺序保证：

在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。（Kafka 保证一个 Partition 内的消息的有序性）

7.缓冲：

有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。

8.异步通信：

很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

15、Zookeeper 对于 Kafka 的作用是什么？

Zookeeper 是一个开放源码的、高性能的协调服务，它用于 Kafka 的分布式应用。Zookeeper 主要用于在集群中不同节点之间进行通信

在 Kafka 中，它被用于提交偏移量，因此如果节点在任何情况下都失败了，它都可以从之前提交的偏移量中获取

除此之外，它还执行其他活动，如：leader 检测、分布式同步、配置管理、识别新节点何时离开或连接、集群、节点实时状态等等。

16、数据传输的事务定义有哪三种？

和 MQTT 的事务定义一样都是 3 种。

(1) 最多一次：消息不会被重复发送，最多被传输一次，但也有可能一次不传输

(2) 最少一次：消息不会被漏发送，最少被传输一次，但也有可能被重复传输。

(3) 精确的一次 (Exactly once)：不会漏传输也不会重复传输，每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的

16、Kafka 判断一个节点是否还活着有那两个条件？

(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

17、Kafka 与传统 MQ 消息系统之间有三个关键区别

(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

18、讲一讲 kafka 的 ack 的三种机制

request.required.acks 有三个值 0 1 -1(all)

0：生产者不会等待 broker 的 ack，这个延迟最低但是存储的保证最弱当 server 挂掉的时候就会丢数据。

1：服务端会等待 ack 值 leader 副本确认接收到消息后发送 ack 但是如果 leader 挂掉后他不确保是否复制完成新 leader 也会导致数据丢失。

-1(all)：服务端会等所有的 follower 的副本受到数据后才会受到 leader 发出的 ack，这样数据不会丢失

19、消费者如何不自动提交偏移量，由应用提交？

将 auto.commit.offset 设为 false，然后在处理一批消息后 commitSync() 或者异步提交 commitAsync()
即：

```
ConsumerRecords< ConsumerRecord> records = consumer.poll();
for (ConsumerRecord< ConsumerRecord> record : records){
    ...
try{
    consumer.commitSync()
} .
.
}
```

20、消费者故障，出现活锁问题如何解决？

出现“活锁”的情况，是它持续的发送心跳，但是没有处理。为了预防消费者在这种情况下一直持有分区，我们使用 max.poll.interval.ms 活跃检测机制。在此基础上，如果你调用的 poll 的频率大于最大间隔，则客户端将主动地离开组，以便其他消费者接管该分区。发生这种情况时，你会看到 offset 提交失败（调用 commitSync() 引发的 CommitFailedException）。这是一种安全机制，保障只有活动成员能够提交 offset。所以要留在组中，你必须持续调用 poll。

消费者提供两个配置设置来控制 poll 循环：

max.poll.interval.ms：增大 poll 的间隔，可以为消费者提供更多的时 间去处理返回的消息（调用 poll(long) 返回的消息，通常返回的消息都是一批）。缺点是此值越大将会延迟组重新平衡。

max.poll.records：此设置限制每次调用 poll 返回的消息数，这样可以更容易的预测每次 poll 间隔要处理的最大值。通过调整此值，可以减少 poll 间隔，减少重新平衡分区的

对于消息处理时间不可预测的情况，这些选项是不够的。处理这种情况的推荐方法是将消息处理移到另一个线程中，让消费者继续调用 poll。但是必须注意确保已提交的 offset 不超过实际位置。另外，你必须禁用自动提交，并只有在线程完成处理后才为记录手动提交偏移量（取决于你）。还要注意，你需要 pause 暂停分区，不会从 poll 接收到新消息，让线程处理完之前返回的消息（如果你的处理能力比拉取消息的慢，那创建新线程将导致你机器内存溢出）。

21、如何控制消费的位置

kafka 使用 seek(TopicPartition, long) 指定新的消费位置。用于查找服务器保留的最早和最新的 offset 的特殊的方法也可用（seekToBeginning(Collection) 和 seekToEnd(Collection)）

22、kafka 分布式（不是单机）的情况下，如何保证消息的顺序消费？

Kafka 分布式的单位是 partition，同一个 partition 用一个 write ahead log 组织，所以可以保证 FIFO 的顺序。不同 partition 之间不能保证顺序。但是绝大多数用户都可以通过 message key 来定义，因为同一个 key 的 Message 可以保证只发送到同一个 partition。

Kafka 中发送 1 条消息的时候，可以指定(topic, partition, key) 3 个参数。partition 和 key 是可选的。如果你指定了 partition，那就是所有消息发往同 1 个 partition，就是有序的。并且在消费端，Kafka 保证，1 个 partition 只能被 1 个 consumer 消费。或者你指定 key（比如 order id），具有同 1 个 key 的所有消息，会发往同 1 个 partition。

23、kafka 的高可用机制是什么？

这个问题比较系统，回答出 kafka 的系统特点，leader 和 follower 的关系，消息读写的顺序即可。

<https://www.cnblogs.com/qingyunzong/p/9004703.html>

<https://www.tuicool.com/articles/BNRza2E>

<https://yq.aliyun.com/articles/64703>

24、kafka 如何减少数据丢失

<https://www.cnblogs.com/huxi2b/p/6056364.html>

25、kafka 如何不消费重复数据？比如扣款，我们不能重复的扣。

其实还是得结合业务来思考，我这里给几个思路：

比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，update 一下好吧。

比如你是写 Redis，那没问题了，反正每次都是 set，天然幂等性。

比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的 id，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。

比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据

微服务 面试题

微服务，又称微服务架构，是一种架构风格，它将应用程序构建为以业务领域为模型的小型自治服务集合。

通俗地说，你必须看到蜜蜂如何通过对齐六角形蜡细胞来构建它们的蜂窝状物。他们最初从使用各种材料的小部分开始，并继续从中构建一个大型蜂箱。这些细胞形成图案，产生坚固的结构，将蜂窝的特定部分固定在一起。这里，每个细胞独立于另一个细胞，但它也与其他细胞相关。这意味着对一个细胞的损害不会损害其他细胞，因此，蜜蜂可以在不影响完整蜂箱的情况下重建这些细胞。

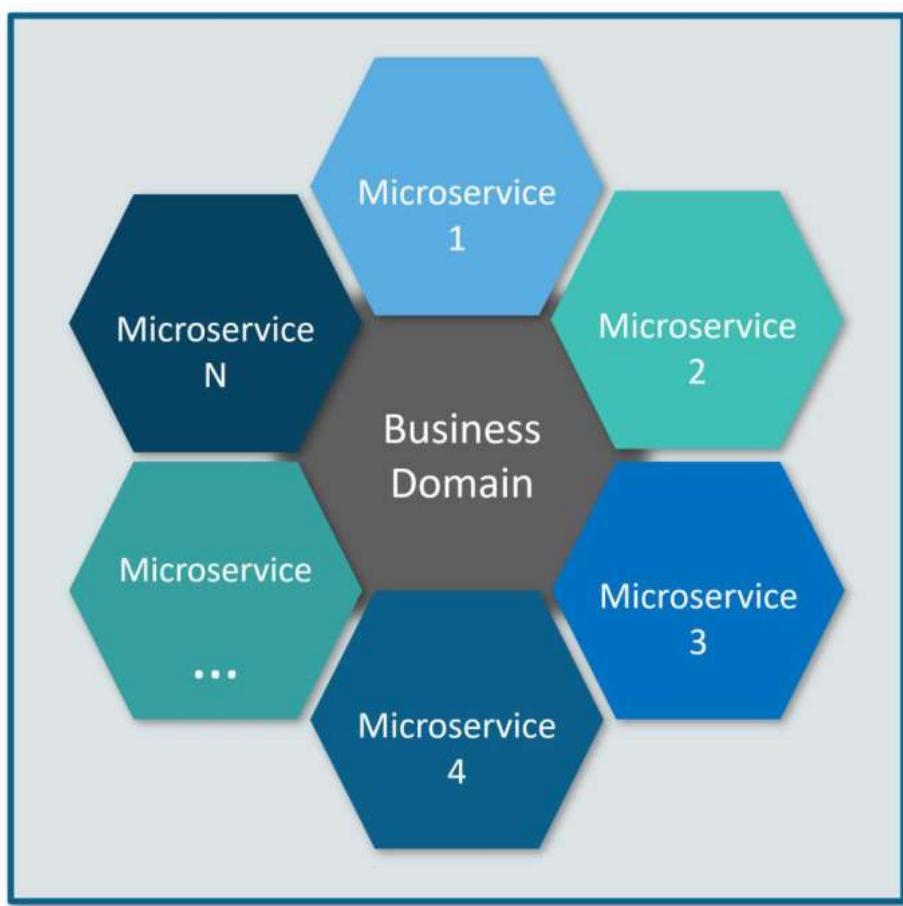


图 1：微服务的蜂窝表示 – 微服务访谈问题

请参考上图。这里，每个六边形形状代表单独的服务组件。与蜜蜂的工作类似，每个敏捷团队都使用可用的框架和所选的技术堆栈构建单独的服务组件。就像在蜂箱中一样，每个服务组件形成一个强大的微服务架构，以提供更好的可扩展性。此外，敏捷团队可以单独处理每个服务组件的问题，而对整个应用程序没有影响或影响最小。

2、微服务架构有哪些优势？

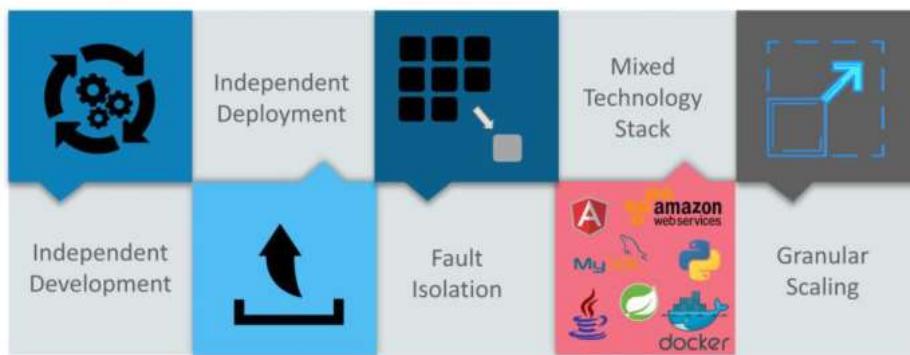


图 2 : 微服务的优点 - 微服务访谈问题

- 独立开发 - 所有微服务都可以根据各自的功能轻松开发
- 独立部署 - 基于其服务，可以在任何应用程序中单独部署它们
- 故障隔离 - 即使应用程序的一项服务不起作用，系统仍可继续运行
- 混合技术堆栈 - 可以使用不同的语言和技术来构建同一应用程序的不同服务
- 粒度缩放 - 单个组件可根据需要进行缩放，无需将所有组件缩放在一起

3、微服务有哪些特点？



图 3 : 微服务的特点 - 微服务访谈问题

- 解耦 - 系统内的服务很大程度上是分离的。因此，整个应用程序可以轻松构建，更改和扩展
- 组件化 - 微服务被视为可以轻松更换和升级的独立组件
- 业务能力 - 微服务非常简单，专注于单一功能
- 自治 - 开发人员和团队可以彼此独立工作，从而提高速度
- 持续交付 - 通过软件创建，测试和批准的系统自动化，允许频繁发布软件
- 责任 - 微服务不关注应用程序作为项目。相反，他们将应用程序视为他们负责的产品
- 分散治理 - 重点是使用正确的工具来做正确的工作。这意味着没有标准化模式或任何技术模式。开发人员可以自由选择最有用的工具来解决他们的问题
- 敏捷 - 微服务支持敏捷开发。任何新功能都可以快速开发并再次丢弃

4、设计微服务的最佳实践是什么？

以下是设计微服务的最佳实践：

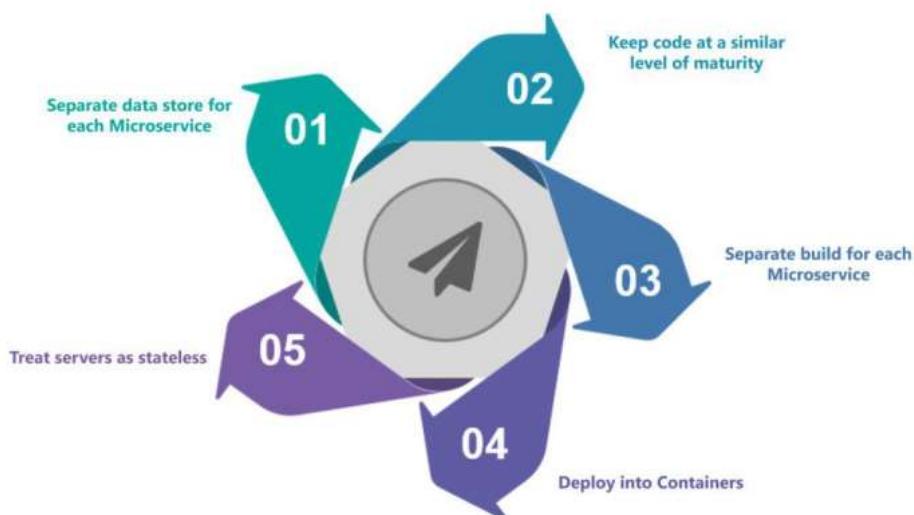


图 4 : 设计微服务的最佳实践 – 微服务访谈问题

5、微服务架构如何运作？

微服务架构具有以下组件：

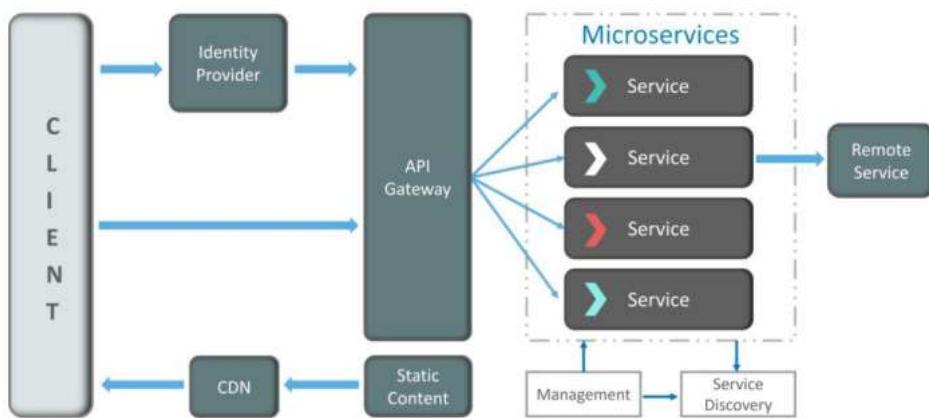


图 5：微服务 架构 - 微服务面试问题

- 客户端 - 来自不同设备的不同用户发送请求。
- 身份提供商 - 验证用户或客户身份并颁发安全令牌。
- API 网关 - 处理客户端请求。
- 静态内容 - 容纳系统的所有内容。
- 管理 - 在节点上平衡服务并识别故障。
- 服务发现 - 查找微服务之间通信路径的指南。
- 内容交付网络 - 代理服务器及其数据中心的分布式网络。
- 远程服务 - 启用驻留在 IT 设备网络上的远程访问信息。

6、微服务架构的优缺点是什么？

微服务架构的优点	微服务架构的缺点
自由使用不同的技术	增加故障排除挑战
每个微服务都侧重于单一功能	由于远程呼叫而增加延迟
支持单个可部署单元	增加了配置和其他操作的工作量
允许经常发布软件	难以保持交易安全
确保每项服务的安全性	艰难地跨越各种边界跟踪数据
多个服务是并行开发和部署的	难以在服务之间进行编码

7、单片，SOA 和微服务架构有什么区别？

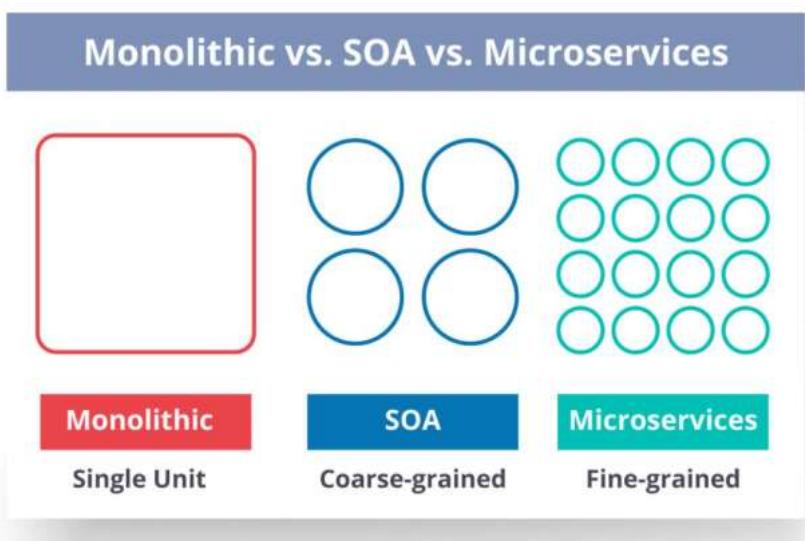


图 6：单片 SOA 和微服务之间的比较 - 微服务访谈问题

- 单片架构类似于大容器，其中应用程序的所有软件组件组装在一起并紧密封装。

- 一个面向服务的架构是一种相互通信服务的集合。通信可以涉及简单的数据传递，也可以涉及两个或多个协调某些活动的服务。
- 微服务架构是一种架构风格，它将应用程序构建为以业务域为模型的小型自治服务集合。

8、在使用微服务架构时，您面临哪些挑战？

- 开发一些较小的微服务听起来很容易，但开发它们时经常遇到的挑战如下。
- 自动化组件：难以自动化，因为有许多较小的组件。因此，对于每个组件，我们必须遵循 Build，Deploy 和 Monitor 的各个阶段。
 - 易感性：将大量组件维护在一起变得难以部署，维护，监控和识别问题。它需要在所有组件周围具有很好的感知能力。
 - 配置管理：有时在各种环境中维护组件的配置变得困难。
 - 调试：很难找到错误的每一项服务。维护集中式日志记录和仪表板以调试问题至关重要。

9、SOA 和微服务架构之间的主要区别是什么？

SOA 和微服务之间的主要区别如下：

SOA	微服务
遵循“尽可能多的共享”架构方法	遵循“尽可能少分享”的架构方法
重要性在于 业务功能 重用	重要性在于“有界背景”的概念
他们有 共同的 治理 和标准	他们专注于 人们的 合作 和其他选择的自由
使用 企业服务总线（ESB） 进行通信	简单的消息系统
它们支持 多种消息协议	他们使用 轻量级协议，如 HTTP / REST 等。
多线程， 有更多的开销来处理I / O.	单线程 通常使用Event Loop功能进行非锁定I / O 处理
最大化应用程序服务可重用性	专注于 解耦
传统的关系数据库 更常用	现代 关系数据库 更常用
系统的变化需要修改整体	系统的变化是创造一种新的服务
DevOps / Continuous Delivery正在变得流行，但还不是主流	专注于DevOps /持续交付

10、微服务有什么特点？

您可以列出微服务的特征，如下所示：



图 7：微服务的特征 - 微服务访谈问题

11、什么是领域驱动设计？

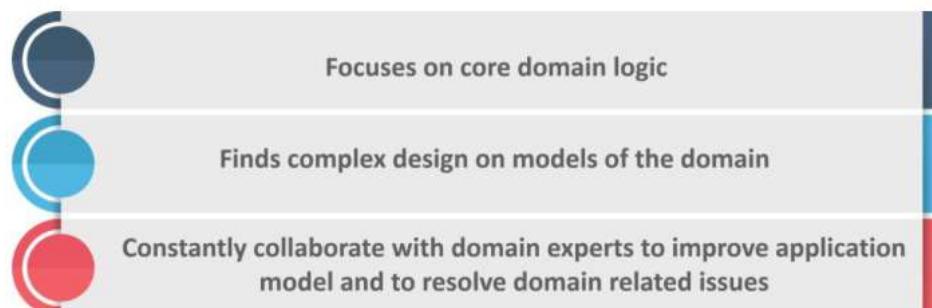


图 8：DDD 原理 - 微服务面试问题

12、为什么需要域驱动设计（DDD）？

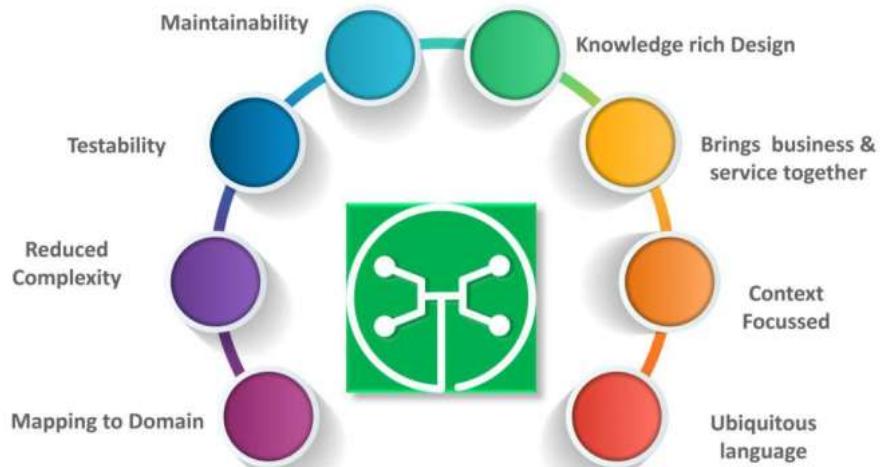


图 9 : 我们需要 DDD 的因素 - 微服务面试问题

13、什么是无所不在的语言？

如果您必须定义泛在语言（UL），那么它是特定域的开发人员和用户使用的通用语言，通过该语言可以轻松解释域。无处不在的语言必须非常清晰，以便它将所有团队成员放在同一页面上，并以机器可以理解的方式进行翻译。

14、什么是凝聚力？

模块内部元素所属的程度被认为是凝聚力。

15、什么是耦合？

组件之间依赖关系强度的度量被认为是耦合。一个好的设计总是被认为具有高内聚力和低耦合性。

16、什么是 REST / RESTful 以及它的用途是什么？

Representational State Transfer (REST) / RESTful Web 服务是一种帮助计算机系统通过 Internet 进行通信的架构风格。这使得微服务更容易理解和实现。微服务可以使用或不使用 RESTful API 实现，但使用 RESTful API 构建松散耦合的微服务总是更容易。

17、你对 Spring Boot 有什么了解？

事实上，随着新功能的增加，弹簧变得越来越复杂。如果必须启动新的 spring 项目，则必须添加构建路径或添加 maven 依赖项，配置应用程序服务器，添加 spring 配置。所以一切都必须从头开始。Spring Boot 是解决这个问题的方法。使用 spring boot 可以避免所有样板代码和配置。因此，基本上认为自己就好像你正在烘烤蛋糕一样，春天就像制作蛋糕所需的成分一样，弹簧靴就是你手中的完整蛋糕。



图 10 : Spring Boot 的因素 - 微服务面试问题

18、什么是 Spring 引导的执行器？

Spring Boot 执行程序提供了 restful Web 服务，以访问生产环境中运行应用程序的当前状态。在执行器的帮助下，您可以检查各种指标并监控您的应用程序。

19、什么是 Spring Cloud？

根据 Spring Cloud 的官方网站，Spring Cloud 为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智能路由，领导选举，分布式会话，集群状态）。

20、Spring Cloud 解决了哪些问题？

在使用 Spring Boot 开发分布式微服务时，我们面临的问题很少由 Spring Cloud 解决。

- 与分布式系统相关的复杂性 - 包括网络问题，延迟开销，带宽问题，安全问题。
- 处理服务发现的能力 - 服务发现允许集群中的进程和服务找到彼此并进行通信。
- 解决冗余问题 - 冗余问题经常发生在分布式系统中。
- 负载平衡 - 改进跨多个计算资源（例如计算机集群，网络链接，中央处理单元）的工作负载分布。
- 减少性能问题 - 减少因各种操作开销导致的性能问题。

21、在 Spring MVC 应用程序中使用 WebMvcTest 注释有什么用处？

在测试目标只关注 Spring MVC 组件的情况下，WebMvcTest 注释用于单元测试 Spring MVC 应用程序。在上面显示的快照中，我们只想启动 ToTestController。执行此单元测试时，不会启动所有其他控制器和映射。

22、你能否给出关于休息和微服务的要点？

虽然您可以通过多种方式实现微服务，但 REST over HTTP 是实现微服务的一种方式。REST 还可用于其他应用程序，如 Web 应用程序，API 设计和 MVC 应用程序，以提供业务数据。
微服务是一种体系结构，其中系统的所有组件都被放入单独的组件中，这些组件可以单独构建，部署和扩展。微服务的某些原则和最佳实践有助于构建弹性应用程序。简而言之，您可以说 REST 是构建微服务的媒介。

23、什么是不同类型的微服务测试？

在使用微服务时，由于有多个微服务协同工作，测试变得非常复杂。因此，测试分为不同的级别。

- 在底层，我们有面向技术的测试，如单元测试和性能测试。这些是完全自动化的。
- 在中间层面，我们进行了诸如压力测试和可用性测试之类的探索性测试。
- 在顶层，我们的验收测试数量很少。这些验收测试有助于利益相关者理解和验证软件功能。

24、您对 Distributed Transaction 有何了解？

分布式事务是指单个事件导致两个或多个不能以原子方式提交的单独数据源的突变的任何情况。在微服务的世界中，它变得更加复杂，因为每个服务都是一个工作单元，并且大多数时候多个服务必须协同工作才能使业务成功。

25、什么是 Idempotence 以及它在哪里使用？

幂等性是能够以这样的方式做两次事情的特性，即最终结果将保持不变，即好像它只做了一次。
用法：在远程服务或数据源中使用 Idempotence，这样当它多次接收指令时，它只处理指令一次。

26、什么是上下文？

上下文是域驱动设计的核心模式。DDD 战略设计部门的重点是处理大型模型和团队。DDD 通过将大型模型划分为不同的上下文并明确其相互关系来处理大型模型。

27、什么是双因素身份验证？

双因素身份验证为帐户登录过程启用第二级身份验证。



图 11：双因素认证的表示 - 微服务访谈问题

因此，假设用户必须只输入用户名和密码，那么这被认为是单因素身份验证。

28、双因素身份验证的凭据类型有哪些？

这三种凭证是：



图 12：双因素认证的证书类型 - 微服务面试问题

29、什么是客户证书？

客户端系统用于向远程服务器发出经过身份验证的请求的一种数字证书称为客户端证书。客户端证书在许多相互认证设计中起着非常重要的作用，为请求者的身份提供了强有力的保证。

30、PACT 在微服务架构中的用途是什么？

PACT 是一个开源工具，允许测试服务提供者和消费者之间的交互，与合同隔离，从而提高微服务集成的可靠性。

微服务中的用法

- 用于在微服务中实现消费者驱动的合同。
- 测试微服务的消费者和提供者之间的消费者驱动的合同。

查看即将到来的批次

31、什么是 OAuth？

OAuth 代表开放授权协议。这允许通过在 HTTP 服务上启用客户端应用程序（例如第三方提供商 Facebook，GitHub 等）来访问资源所有者的资源。因此，您可以在不使用其凭据的情况下与另一个站点共享存储在一个站点上的资源。

32、康威定律是什么？

“任何设计系统的组织（广泛定义）都将产生一种设计，其结构是组织通信结构的副本。” – Mel Conway

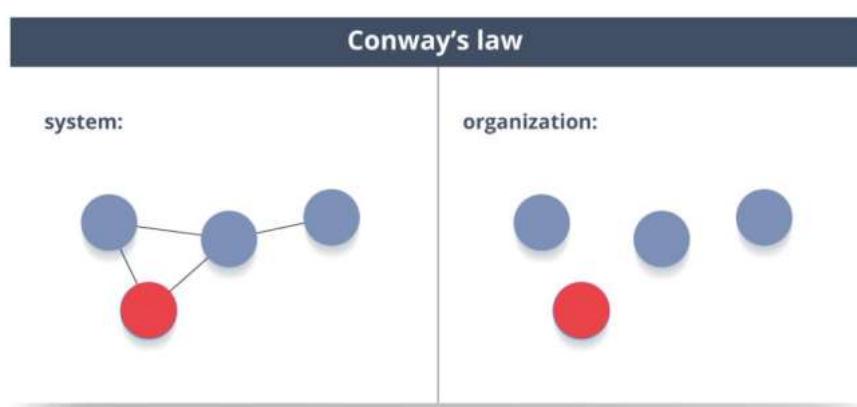


图 13：Conway 定律的表示 - 微服务访谈问题

该法律基本上试图传达这样一个事实：为了使软件模块起作用，整个团队应该进行良好的沟通。因此，系统的结构反映了产生它的组织的社会边界。

33、合同测试你懂什么？

根据 Martin Flower 的说法，合同测试是在外部服务边界进行的测试，用于验证其是否符合消费服务预期的合同。

此外，合同测试不会深入测试服务的行为。更确切地说，它测试该服务调用的输入 & 输出包含所需的属性和所述响应延迟，吞吐量是允许的限度内

34、什么是端到端微服务测试？

端到端测试验证了工作流中的每个流程都正常运行。这可确保系统作为一个整体协同工作并满足所有要求。

通俗地说，你可以说端到端测试是一种测试，在特定时期后测试所有东西。

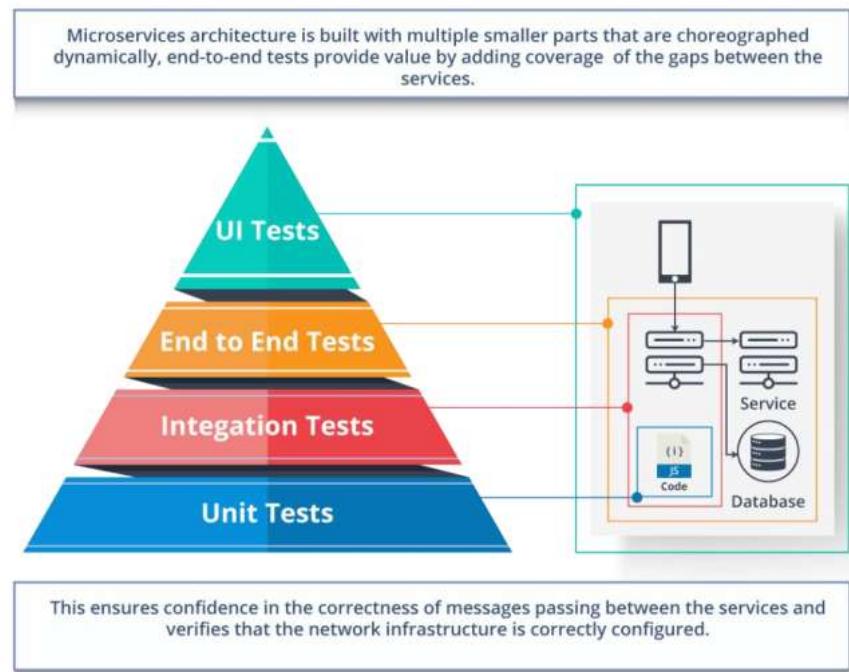


图 14：测试层次 - 微服务面试问题

35、Container 在微服务中的用途是什么？

容器是管理基于微服务的应用程序以便单独开发和部署它们的好方法。您可以将微服务封装在容器映像及其依赖项中，然后可以使用它来滚动按需实例的微服务，而无需任何额外的工作。

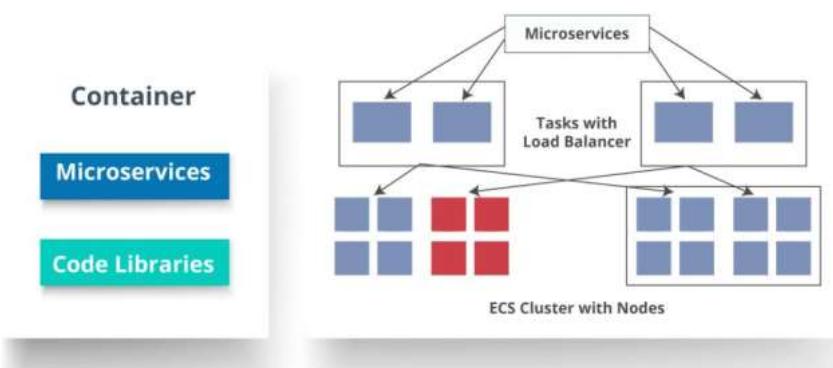


图 15：容器的表示及其在微服务中的使用方式 - 微服务访谈问题

36、什么是微服务架构中的 DRY ?

DRY 代表不要重复自己。它基本上促进了重用代码的概念。这导致开发和共享库，这反过来导致紧密耦合。

37、什么是消费者驱动的合同 (CDC) ?

这基本上是用于开发微服务的模式，以便它们可以被外部系统使用。当我们处理微服务时，有一个特定的提供者构建它，并且有一个或多个使用微服务的消费者。通常，提供程序在 XML 文档中指定接口。但在消费者驱动的合同中，每个服务消费者都传达了提供商期望的接口。

38、Web , RESTful API 在微服务中的作用是什么？

微服务架构基于一个概念，其中所有服务应该能够彼此交互以构建业务功能。因此，要实现这一点，每个微服务必须具有接口。这使得 Web API 成为微服务的一个非常重要的推动者。RESTful API 基于 Web 的开放网络原则，为构建微服务架构的各个组件之间的接口提供了最合理的模型。

39、您对微服务架构中的语义监控有何了解？

语义监控，也称为 综合监控，将自动化测试与监控应用程序相结合，以检测业务失败因素。

40、我们如何进行跨功能测试？

跨功能测试是对非功能性需求的验证，即那些无法像普通功能那样实现的需求。

41、我们如何在测试中消除非决定论？

非确定性测试（NDT）基本上是不可靠的测试。所以，有时可能会发生它们通过，显然有时它们也可能失败。当它们失败时，它们会重新运行通过。

从测试中删除非确定性的一些方法如下：

- 1、隔离
- 2、异步
- 3、远程服务
- 4、隔离
- 5、时间
- 6、资源泄漏

42、Mock 或 Stub 有什么区别？

存根

- 一个有助于运行测试的虚拟对象。
- 在某些可以硬编码的条件下提供固定行为。
- 永远不会测试存根的任何其他行为。

例如，对于空堆栈，您可以创建一个只为 empty（）方法返回 true 的存根。因此，这并不关心堆栈中是否存在元素。

嘲笑

- 一个虚拟对象，其中最初设置了某些属性。
- 此对象的行为取决于 set 属性。
- 也可以测试对象的行为。

例如，对于 Customer 对象，您可以通过设置名称和年龄来模拟它。您可以将 age 设置为 12，然后测试 isAdult（）方法，该方法将在年龄大于 18 时返回 true。因此，您的 Mock Customer 对象适用于指定的条件。

43、您对 Mike Cohn 的测试金字塔了解多少？

Mike Cohn 提供了一个名为 Test Pyramid 的模型。这描述了软件开发所需的自动化测试类型。

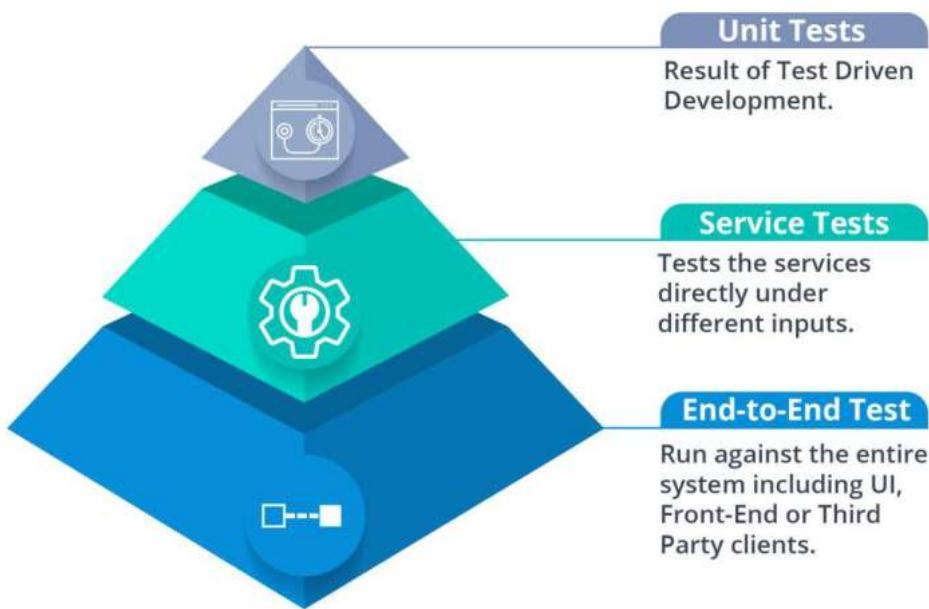


图 16：Mike Cohn 的测试金字塔 – 微服务面试问题

根据金字塔，第一层的测试数量应该最高。在服务层，测试次数应小于单元测试级别，但应大于端到端级别。

44、Docker 的目的是什么？

Docker 提供了一个可用于托管任何应用程序的容器环境。在此，软件应用程序和支持它的依赖项紧密打包在一起。因此，这个打包的产品被称为 Container，因为它是由 Docker 完成的，所以它被称为 Docker 容器！

45、什么是金丝雀释放？

Canary Releasing 是一种降低在生产中引入新软件版本的风险的技术。这是通过将变更缓慢地推广到一小部分用户，然后将其发布到整个基础架构，即将其提供给每个人来完成的。

46、什么是持续集成 (CI) ?

持续集成 (CI) 是每次团队成员提交版本控制更改时自动构建和测试代码的过程。这鼓励开发人员通过在每个小任务完成后将更改合并到共享版本控制存储库来共享代码和单元测试。

47、什么是持续监测？

持续监控深入监控覆盖范围，从浏览器内前端性能指标，到应用程序性能，再到主机虚拟化基础架构指标。

48、架构师在微服务架构中的角色是什么？

微服务架构中的架构师扮演以下角色：

- 决定整个软件系统的布局。
- 帮助确定组件的分区。因此，他们确保组件相互粘合，但不紧密耦合。
- 与开发人员共同编写代码，了解日常生活中面临的挑战。
- 为开发微服务的团队提供某些工具和技术的建议。
- 提供技术治理，以便技术开发团队遵循微服务原则。

49、我们可以用微服务创建状态机吗？

我们知道拥有自己的数据库的每个微服务都是一个可独立部署的程序单元，这反过来又让我们可以创建一个状态机。因此，我们可以为特定的微服务指定不同的状态和事件。例如，我们可以定义 Order 微服务。订单可以具有不同的状态。Order 状态的转换可以是 Order 微服务中的独立事件。

50、什么是微服务中的反应性扩展？

Reactive Extensions 也称为 Rx。这是一种设计方法，我们通过调用多个服务来收集结果，然后编译组合响应。这些调用可以是同步或异步，阻塞或非阻塞。Rx 是分布式系统中非常流行的工具，与传统流程相反。希望这些微服务面试问题可以帮助您进行微服务架构师访谈。

翻译来源：

<https://www.edureka.co/blog/interview-questions/microservices-interview-questions/>

Linux面试题

1、绝对路径用什么符号表示？当前目录、上层目录用什么表示？主目录用什么表示？切换目录用什么命令？

答案：

绝对路径：如/etc/init.d

当前目录和上层目录：./..

主目录：~/

切换目录：cd

2、怎么查看当前进程？怎么执行退出？怎么查看当前路径？

答案：
查看当前进程：ps
执行退出：exit
查看当前路径：pwd

3、怎么清屏？怎么退出当前命令？怎么执行睡眠？怎么查看当

前用户 id ? 查看指定帮助用什么命令 ?
答案：
清屏：clear
退出当前命令：ctrl+c 彻底退出
执行睡眠：ctrl+z 挂起当前进程 fg 恢复后台 查看当前用户 id：“id”：查看显示目前登陆账户的 uid 和 gid 及所属分组及用户名
查看指定帮助：如 man adduser 这个很全而且有例子；adduser --help 这个告诉你一些常用参数；info adduser；

4、Ls 命令执行什么功能？可以带哪些参数，有什么区别？

答案：
ls 执行的功能：列出指定目录中的目录，以及文件哪些参数以及区别：a 所有文件 | 详细信息，包括大小字节数，可读可写可执行的权限等

5、查看文件有哪些命令

vi 文件名 #编辑方式查看，可修改
cat 文件名 #显示全部文件内容
more 文件名 #分页显示文件内容
less 文件名 #与 more 相似，更好的是以前翻页
tail 文件名 #仅查看尾部，还可以指定行数
head 文件名 #仅查看头部，还可以指定行数

6、列举几个常用的Linux命令

列出文件列表：ls 【参数 -a -l】
创建目录和移除目录：mkdir rmdir
用于显示文件后几行内容：tail，例如：tail -n 1000：显示最后1000行
打包：tar -xvf
打包并压缩：tar -zcvf
查找字符串：grep
显示当前所在目录：pwd
创建空文件：touch
编辑器：vim vi

7、你平时是怎么查看日志的？

Linux查看日志的命令有多种：tail、cat、tac、head、echo等，本文只介绍几种常用的方法。

1、tail
最常用的一种查看方式
命令格式：tail[必要参数][选择参数][文件]
-f 循环读取
-q 不显示处理信息
-v 显示详细的处理信息
-c<数目> 显示的字节数
-n<行数> 显示行数
-q, --quiet, --silent 从不输出给出文件名的首部
-s, --sleep-interval=S 与-f合用，表示在每次反复的间隔休眠S秒

例如：

```
tail -n 10 test.log 查询日志尾部最后10行的日志；  
tail -n +10 test.log 查询10行之后的所有日志；  
tail -fn 10 test.log 循环实时查看最后1000行记录(最常用的)
```

一般还会配合着grep搜索用，例如：

```
tail -fn 1000 test.log | grep '关键字'
```

如果一次性查询的数据量太大，可以进行翻页查看，例如：

```
tail -n 4700 aa.log | more -1000 可以进行多屏显示(ctrl + f 或者 空格键可以快捷键)
```

2、head
跟tail是相反的head是看前多少行日志

```
head -n 10 test.log 查询日志文件中的头10行日志；  
head -n -10 test.log 查询日志文件除了最后10行的所有日志；
```

head其他参数参考tail

3. cat

cat 是由第一行到最后一行连续显示在屏幕上
一次显示整个文件：

```
$ cat filename
```

从键盘创建一个文件：

```
$ cat > filename
```

将几个文件合并为一个文件：

```
$ cat file1 file2 > file 只能创建新文件，不能编辑已有文件
```

将一个日志文件的内容追加到另外一个：

```
$ cat -n textfile1 > textfile2
```

清空一个日志文件：

```
$ cat : >textfile2
```

注意：> 意思是创建，>>是追加。千万不要弄混了。

cat其他参数参考tail

4. more

more命令是一个基于vi编辑器文本过滤器，它以全屏幕的方式按页显示文本文件的内容，支持vi中的关键字定位操作。more名单中内置了若干快捷键，常用的有H（获得帮助信息），Enter（向下翻滚一行），空格（向下滚动一屏），Q（退出命令）。more命令从前向后读取文件，因此在启动时就加载整个文件。

该命令一次显示一屏文本，满屏后停下来，并且在屏幕的底部出现一个提示信息，给出至今已显示的该文件的百分比：-More- (XX%)

more的语法：more 文件名

Enter 向下n行，需要定义，默认为1行

Ctrl f 向下滚动一屏

空格键 向下滚动一屏

Ctrl b 返回上一屏

= 输出当前行的行号

:f 输出文件名和当前行的行号

v 调用vi编辑器

!命令 调用Shell，并执行命令

q退出more

5. sed

这个命令可以查找日志文件特定的一段，根据时间的一个范围查询，可以按照行号和时间范围查询按照行号

```
sed -n '5,10p' filename 这样你就可以只查看文件的第5行到第10行。
```

按照时间段

```
sed -n '/2014-12-17 16:17:20/,/2014-12-17 16:17:36/p' test.log
```

6. less

less命令在查询日志时，一般流程是这样的

```
less log.log  
shift + G 命令到文件尾部 然后输入 ? 加上你要搜索的关键字例如 ? 1213  
按 n 向上查找关键字  
shift+ctrl+n 反向查找关键字  
Tess与more类似，使用Tess可以随意浏览文件，而more仅能向前移动，不能向后移动，而且 Tess 在查看之前不会加载整个文件。  
less log2013.log 查看文件  
ps -ef | less ps查看进程信息并通过less分页显示  
history | less 查看命令历史使用记录并通过less分页显示  
less log2013.log log2014.log 浏览多个文件
```

常用命令参数：

```
less与more类似，使用less可以随意浏览文件，而more仅能向前移动，不能向后移动，而且 less 在查看之前不会加载整个文件。  
less log2013.log 查看文件  
ps -ef | less ps查看进程信息并通过less分页显示  
history | less 查看命令历史使用记录并通过less分页显示  
less log2013.log log2014.log 浏览多个文件
```

常用命令参数：
-b <缓冲区大小> 设置缓冲区的大小
-g 只标志最后搜索的关键词
-i 忽略搜索时的大小写
-m 显示类似more命令的百分比
-N 显示每行的行号
-o <文件名> 将less 输出的内容在指定文件中保存起来
-Q 不使用警告音
-S 显示连续空行为一行
/字符串：向下搜索“字符串”的功能
?字符串：向上搜索“字符串”的功能
n: 重复前一个搜索（与 / 或 ? 有关）
N: 反向重复前一个搜索（与 / 或 ? 有关）
b 向后翻一页
h 显示帮助界面
q 退出less 命令

一般人查日志配合应用的其他命令

```
history // 所有的历史记录  
history | grep xxx // 历史记录中包含某些指令的记录  
history | more // 分页查看记录  
history -c // 清空所有的历史记录  
!! 重复执行上一个命令  
查询出来记录后选中 : !323
```

8、建立软链接(快捷方式), 以及硬链接的命令

```
软链接: ln -s slink source  
硬链接: ln link source
```

9、目录创建用什么命令？创建文件用什么命令？复制文件用什么命令？

创建目录：mkdir
创建文件：典型的如 touch , vi 也可以创建文件，其实只要向一个不存在的文件输出，都会创建文件
复制文件：cp 7. 文件权限修改用什么命令？格式是怎么样的？
文件权限修改：chmod
格式如下：
chmodu+xfile 给 file 的属主增加执行权限 chmod 751 file 给 file 的属主分配读、写、执行(7)的权限，给 file 的所在组分配读、执行(5)的权限，给其他用户分配执行(1)的权限
chmodu=rwx,g=rx,o=xfile 上例的另一种形式 chmod =r file 为所有用户分配读权限
chmod444file 同上例 chmod a-wx,a+r file 同上例
\$ chmod -R u+r directory 递归地给 directory 目录下所有文件和子目录的属主分配读的权限

10、查看文件内容有哪些命令可以使用？

```
vi 文件名 #编辑方式查看，可修改  
cat 文件名 #显示全部文件内容  
more 文件名 #分页显示文件内容  
less 文件名 #与 more 相似，更好的是可以往前翻页  
tail 文件名 #仅查看尾部，还可以指定行数  
head 文件名 #仅查看头部，还可以指定行数
```

11、随意写文件命令？怎么向屏幕输出带空格的字符串，比如“hello world”？

写文件命令：vi
向屏幕输出带空格的字符串:echo hello world

12、终端是哪个文件夹下的哪个文件？黑洞文件是哪个文件夹下的哪个命令？

终端 /dev/tty
黑洞文件 /dev/null

13、移动文件用哪个命令？改名用哪个命令？

mv mv

14、复制文件用哪个命令？如果需要连同文件夹一块复制呢？如果有提示功能呢？

cp cp -r ? ? ? ?

15、删除文件用哪个命令？如果需要连目录及目录下文件一块删除呢？删除空文件夹用什么命令？

rm rm -r rmdir

16、Linux 下命令有哪几种可使用的通配符？分别代表什么含义？

? "可替代单个字符。
* "可替代任意多个字符。
方括号 "[charset]" 可替代 charset 集中的任何单个字符，如 [a-z]，[abABC]

17、用什么命令对一个文件的内容进行统计？(行号、单词数、字节数)

wc 命令 -c 统计字节数 -l 统计行数 -w 统计字数。

18、Grep 命令有什么用？如何忽略大小写？如何查找不含该串的行？

是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。

grep [string] filename grep [^string] filename

19、Linux 中进程有哪几种状态？在 ps 显示出来的信息中分别用什么符号表示的？

- 1、不可中断状态：进程处于睡眠状态，但是此刻进程是不可中断的。不可中断，指进程不响应异步信号。
 - 2、暂停状态/跟踪状态：向进程发送一个 SIGSTOP 信号，它就会因响应该信号而进入 TASK_STOPPED 状态；当进程正在被跟踪时，它处于 TASK_TRACED 这个特殊的状态。正被“跟踪”指的是进程暂停下来，等待跟踪它的进程对它进行操作。
 - 3、就绪状态：在 run_queue 队列里的状态
 - 4、运行状态：在 run_queue 队列里的状态
 - 5、可中断睡眠状态：处于这个状态的进程因为等待某些事件的发生（比如等待 socket 连接、等待信号量），而被挂起
 - 6、zombie 状态（僵尸）：父亲没有通过 wait 系列的系统调用会顺便将子进程的尸体（task_struct）也释放掉
 - 7、退出状态
- D 不可中断 Uninterruptible (usually IO)
R 正在运行，或在队列中的进程
S 处于休眠状态
T 停止或被追踪
Z 僵尸进程
W 进入内存交换（从内核 2.6 开始无效）
X 死掉的进程

20、怎么使一个命令在后台运行？

一般都是使用 & 在命令结尾来让程序自动运行。（命令后可以不追加空格）

21、利用 ps 怎么显示所有的进程？怎么利用 ps 查看指定进程的信息？

```
ps -ef (system v 输出)
ps -aux bsd 格式输出
ps -ef | grep pid
```

22、哪个命令专门用来查看后台任务？

job -l

23、把后台任务调到前台执行使用什么命令？把停下的后台任务在后台执行起来用什么命令？

把后台任务调到前台执行 fg
把停下的后台任务在后台执行起来 bg

24、终止进程用什么命令？带什么参数？

kill [-s <信息名称或编号>][程序] 或 kill [-l <信息编号>]
kill-9 pid

25、怎么查看系统支持的所有信号？

kill -l

26、搜索文件用什么命令？格式是怎么样的？

find <指定目录> <指定条件> <指定动作>
whereis 加参数与文件名
locate 只加文件名
find 直接搜索磁盘，较慢。
find / -name "string*"

27、查看当前谁在使用该主机用什么命令？查找自己所在的终端信息用什么命令？

查找自己所在的终端信息：who am i
查看当前谁在使用该主机：who

28、使用什么命令查看用过的命令列表？

history

29、使用什么命令查看磁盘使用空间？空闲空间呢？

df -h1

文件系统 容量 已用 可用 已用% 挂载点

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/hda2	45G	19G	24G	44%	/
/dev/hda1	494M	19M	450M	4%	/boot

30、使用什么命令查看网络是否连通？

netstat

31、使用什么命令查看 ip 地址及接口信息？

ifconfig

32、查看各类环境变量用什么命令？

查看所有 env
查看某个，如 home：env \$HOME

33、通过什么命令指定命令提示符？

- \u : 显示当前用户账号
- \h : 显示当前主机名
- \W : 只显示当前路径最后一个目录
- \w : 显示当前绝对路径（当前用户目录会以~代替）
- \$PWD : 显示当前全路径
- \\$: 显示命令行\$或者#符号
- # : 下达的第几个命令
- \d : 表示日期，格式为 week day month date，例如："MonAug1"
- \t : 显示时间为 24 小时格式，如：HH : MM : SS
- \T : 显示时间为 12 小时格式
- \A : 显示时间为 24 小时格式：HH : MM
- \v : BASH 的版本信息 如 export PS1='[\u@\h\w#\]\$'

34、查找命令的可执行文件是去哪查找的? 怎么对其进行设置及添加?

whereis [-bfmsu][-B <目录>...][-M <目录>...][-S <目录>...][文件 ...]

补充说明：whereis 指令会在特定目录中查找符合条件的文件。这些文件的类型应属于原始代码，二进制文件，或是帮助文件。

-b 只查找二进制文件。

-B <目录> 只在设置的目录下查找二进制文件。-f 不显示文件名前的路径名称。

-m 只查找说明文件。

-M <目录> 只在设置的目录下查找说明文件。-s 只查找原始代码文件。

-S <目录> 只在设置的目录下查找原始代码文件。-u 查找不包含指定类型的文件。

w -h ich 指令会在 PATH 变量指定的路径中，搜索某个系统命令的位置，并且返回第一个搜索结果。

-n 指定文件名长度，指定的长度必须大于或等于所有文件中最长的文件名。

-p 与-n 参数相同，但此处的包括了文件的路径。-w 指定输出时栏位的宽度。

-V 显示版本信息

35、通过什么命令查找执行命令?

which 只能查可执行文件

whereis 只能查二进制文件、说明文档，源文件等

36、怎么对命令进行取别名？

```
alias la='ls -a'
```

37、du 和 df 的定义，以及区别？

du 显示目录或文件的大小

df 显示每个<文件>所在的文件系统的空间使用情况，默认是显示所有文件系统。（文件系统分配其中的一些磁盘块用来记录它自身的一些数据，如 i 节点，磁盘分布图，间接块，超级块等。这些数据对大多数用户级的程序来说是不可见的，通常称为 Meta Data。）du 命令是用户级的程序，它不考虑 Meta Data，而 df 命令则查看文件系统的磁盘分配图并考虑 Meta Data。

df 命令获得真正的文件系统数据，而 du 命令只查看文件系统的部分情况。

38、awk 详解。

```
awk '{pattern + action}' {filenames}
#cat /etc/passwd |awk -F ':' '{print 1"\t"7}' // -F 的意思是用':'分隔 root
/bin/bash
daemon /bin/sh 搜索/etc/passwd 有 root 关键字的所有行
#awk -F: '/root/' /etc/passwd root:x:0:0:root:/root:/bin/bash
```

39、当你需要给命令绑定一个宏或者按键的时候，应该怎么做呢？

可以使用 bind 命令，bind 可以很方便地在 shell 中实现宏或按键的绑定。

在进行按键绑定的时候，我们需要先获取到绑定按键对应的字符序列。

比如获取 F12 的字符序列获取方法如下：先按下 Ctrl+V,然后按下 F12 .我们就可以得到 F12 的字符序列 \[24~。

接着使用 bind 进行绑定

```
[root@localhost ~]# bind '\e[24~":"date''
```

注意：相同的按键在不同的终端或终端模拟器下可能会产生不同的字符序列。

【附】也可以使用 showkey -a 命令查看按键对应的字符序列。

40、如果一个 linux 新手想要知道当前系统支持的所有命令的列表，他需要怎么做？

使用命令 compgen -c，可以打印出所有支持的命令列表。

```
[root@localhost ~]# compgen -c
1.
11
ls
which
if
then
else
elif
fi
```

```
case  
esac  
for  
select  
while  
until  
do  
done  
...
```

41、如果你的助手想要打印出当前的目录栈，你会建议他怎么做？

使用 Linux 命令 dirs 可以将当前的目录栈打印出来。

```
[root@localhost ~]# dirs  
/usr/share/x11
```

42、你的系统目前有许多正在运行的任务，在不重启机器的条件下，有什么方法可以把所有正在运行的进程移除呢？

使用 linux 命令 'disown -r' 可以将所有正在运行的进程移除。

43、bash shell 中的 hash 命令有什么作用？

linux 命令'hash'管理着一个内置的哈希表，记录了已执行过的命令的完整路径.用该命令可以打印出你所使用过的命令以及执行的次数。

```
[root@localhost ~]# hash  
hits command  
2 /bin/ls  
2 /bin/su
```

44、哪一个 bash 内置命令能够进行数学运算。

bash shell 的内置命令 let 可以进行整型数的数学运算

```
#!/bin/bash  
... ... le  
t c=a+b  
... ...
```

45、怎样一页一页地查看一个大文件的内容呢？

通过管道将命令"cat file_name.txt" 和 'more' 连接在一起可以实现这个需要

```
[root@localhost ~]# cat file_name.txt | more
```

46、数据字典属于哪一个用户的？

数据字典是属于'SYS'用户的，用户'SYS'和'SYSEM'是由系统默认自动创建的

47、怎样查看一个 linux 命令的概要与用法？假设你在/bin 目录中偶然看到一个你从没见过的命令，怎样才能知道它的作用和用法呢？

使用命令 whatis 可以先显示出这个命令的用法简要，比如，你可以使用 whatis zcat 去查看'zcat'的介绍以及使用简要。

```
[root@localhost ~]# whatis zcat  
zcat [gzip] (1) - compress or expand files
```

48、使用哪一个命令可以查看自己文件系统的磁盘空间配额呢？

使用命令 repquota 能够显示 出一个文件系统的配额信息

【附】只有 root 用户才能够查看其它用户的配额。

