链滴

个人整理 - Java 后端面试题 - 架构篇

作者：valarchie
原文链接：https://ld246.com/article/1583985732697
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

标 ★ 号为重要知识点

<h2 id="id全局唯一且自增-如何实现-">id 全局唯一且自增，如何实现？</h2>

Redis 的 incr 和 increby 自增原子命令
统一数据库的 id 发放
美团 Leaf Leaf——美团点评分布式 ID 生成系统（批发号段）
Twitter 的 snowflake 算法
UUID

<h2 id="-如何设计算法压缩一段URL-">★ 如何设计算法压缩一段 URL？</h2>
通过发号策略，给每一个过来的长地址，发一个号即可，小型系统直接用 mysql 的自增索引就定了。如果是大型应用，可以考虑各种分布式 key-value 系统做发号器。不停的自增就行了。第一个用这个服务的人得到的短地址是 http://xx.xx/0 第二个是 http://xx.xx/1 第 11 个是 http://xx.xx/a 依次往后，相当于实现了一个 62 进制的自增字段即可。
常用的 url 压缩算法是短地址映射法。具体步骤是：

将长网址用 md5 算法生成 32 位签名串，分为 4 段,，每段 8 个字符；
对这 4 段循环处理，取每段的 8 个字符, 将他看成 16 进制字符串与 0x3fffffff(30 位 1)的位与操，超过 30 位的忽略处理；
将每段得到的这 30 位又分成 6 段，每 5 位的数字作为字母表的索引取得特定字符，依次进行获 6 位字符串；
这样一个 md5 字符串可以获得 4 个 6 位串，取里面的任意一个就可作为这个长 url 的短 url 地。

<h2 id="-Dubbo负载均衡策略-">★Dubbo 负载均衡策略？</h2>
随机、轮询、最少使用、一致性哈希（除了一致性哈希外，都有加权）
<h2 id="负载均衡算法-">负载均衡算法？</h2>

常见 6 种负载均衡算法：轮询，随机，源地址哈希，加权轮询，加权随机，最小连接数。

nginx5 种负载均衡算法：轮询，weight 轮询，ip_hash，fair（响应时间），url_hash

dubbo 负载均衡算法：随机，轮询，最少活跃调用数，一致性 Hash

<h2 id="Dubbo中Zookeeper做注册中心-如果注册中心集群都挂掉-发布者和订阅者之间还能通信么">Dubbo 中 Zookeeper 做注册中心，如果注册中心集群都挂掉，发布者和订阅者之间还能通信么？/h2>
可以，因为 dubbo 在注册中心挂掉之后，会从原先的缓存中读取连接地址。
<h2 id="-Dubbo完整的一次调用链路介绍-">★Dubbo 完整的一次调用链路介绍？</h2>
调用方：

将方法名方法参数传入 InvokerInvocationHandler 的 invoke 方法中，对于 Object 中的方法 toring, hashCode, equals 直接调用 invoker 的对应方法。
然后进入（故障转移集群）MockClusterInvoker.invoke()方法中。三种调用策略：① 不需要 mok， 直接调用 FailoverClusterInvoker。② 强制 mock，调用 mock。③ 先调 FailoverClusterInvoke，调用失败在 mock.
FailoverClusterInvoker 默认调用策略。① 通过目录服务查找到所有订阅的服务提供者的 Invoke 对象。② 路由服务根据策略（比如：容错策略）来过滤选择调用的 Invokers。③ 通过负载均衡策略

oadBalance 来选择一个 Invoker
执行选择的 Invoker.invoker(invocation),经过监听器链，经过过滤器链,执行到远程调用的 Dubb
Invoker。
DubboInvoker 根据 url 也就是根据服务提供者的长连接，这里封装成交互层对象 ExchangeClie
t 供这里调用，判断远程调用类型同步，异步还是 oneway 模式。ExchangeClient 发起远程调用。<b
>
6.获取调用结果：①Oneway 返回空 RpcResult② 异步，直接返回空 RpcResult, ResponseFuture
调 ③ 同步， ResponseFuture 模式同步转异步，等待响应返回

消费方：

通过 Invocation 获取服务名和端口组成 serviceKey=com.alibaba.dubbo.demo.DemoService:
0880, 从 DubboProtocol 的 exproterMap 中获取暴露服务的 DubboExporter, 在从 dubboExporte
获取 invoker 返回
经过过滤器链。
经过监听器链。
到达执行真正调用的 invoker， 这个 invoker 由代理工厂 ProxyFactory.getInvoker(demoServi
e, DemoService.class, registryUrl)创建，具体请看代理那部分介绍。
调用 demoService 实例方法，将结果封装成 RpcResult 返回。

<h2 id="-SpringCloud和Dubbo有什么不一样-">★SpringCloud 和 Dubbo 有什么不一样？</h2>
1.dubbo 采用 RPC 的方式交互，SpringCloud 采用 Http,restful 协议进行交互。
2.dubbo 依赖 zookeeper 进行服务注册，Springloud 自己拥有自己的服务注册中心。
3.dubbo 需要强依赖，需要持有相同的类或者 jar 包，springcloud 弱依赖，但需要通过接口文档进
约束。
4.C 数据一致性，A 服务可用性，P 服务对网络分区故障的容错性，Zookeeper 保证的是 CP，euraka
保证的是 AP。
<h2 id="使用Redis如何实现分布式锁-">使用 Redis 如何实现分布式锁？</h2>
setnx 指令，设置锁的有效时间防止死锁。设置一个随机值来标识锁的持有人，利用这个随机值
释放锁。
<h2 id="Tomcat如何优化-">Tomcat 如何优化？</h2>

虚拟机参数：

server 模式。
最大堆最小堆大小。
年轻代和老年代的比例。
开启优化。
使用偏向锁。
gc 年龄。
合适的 gc

tomcat 参数：

maxThread。
minThread。
acceptCount。
connectionTimeout。
maxProcessors 与 minProcessors。

<h2 id="-幂等的处理方式-">★ 幂等的处理方式？</h2>

查询与删除操作是天然幂等
唯一索引，防止新增脏数据
token 机制，防止页面重复提交
悲观锁 for update
乐观锁（通过版本号/时间戳实现， 通过条件限制 where avai_amount-#subAmount# >=
）
分布式锁
状态机幂等（如果状态机已经处于下一个状态，这时候来了一个上一个状态的变更，理论上是不够变更的，这样的话，保证了有限状态机的幂等。）
select + insert（并发不高的后台系统，或者一些任务 JOB，为了支持幂等，支持重复执行）</li

redis 存储订单号，如果已存在的话说明已经处理过

<h2 id="后台系统怎么防止用户恶意频繁访问-">后台系统怎么防止用户恶意频繁访问？</h2>
设计一个数据结构，有用户 id,当前秒数,调用次数。每次请求时对比当前秒数和该对象内的是否致，一致的话累加调用次数。不一致的话，将当前秒数替换成新的，调用次数清 0。
<h2 id="-请谈谈单点登录原理-">★ 请谈谈单点登录原理？</h2>

同域下的单点登录，只需共享 session 即可。

登录业务系统，跳转至 SSO 服务器，判断用户名密码正确，在 sso 域下种下 cookie，在 session 中标记为登录，返回一个 ticket，跳转到业务系统，业务系统再拿这个 ticket 跑去 SSO 服务器验证 ticket 是否有效，有效的话，在业务系统 session 中设置为已登录即可。

相比于单系统登录，sso 需要一个独立的认证中心，只有认证中心能接受用户的用户名密码等安信息，其他系统不提供登录入口，只接受认证中心的间接授权。间接授权通过令牌实现，sso 认证中验证用户的用户名密码没问题，创建授权令牌，在接下来的跳转过程中，授权令牌作为参数发送给各子系统，子系统拿到令牌，即得到了授权，可以借此创建局部会话，局部会话登录方式与单系统的登方式相同。这个过程，也就是单点登录的原理，用下图说明

单点登录自然也要单点注销，在一个子系统中注销，所有子系统的会话都将被销毁，用下面的图说明

<h2 id="-谈谈项目中分布式事务应用场景-">★ 谈谈项目中分布式事务应用场景？</h2>

学勤系统与账户余额系统
招生系统与账户余额系统

<h2 id="-MQ和数据库的一致性问题-MQ消息最终一致性-">★MQ 和数据库的一致性问题，MQ 消最终一致性。</h2>

事务消息与普通消息的区别就在于消息生产环节，生产者首先预发送一条消息到 MQ(这也被称发送 half 消息)

MQ 接受到消息后，先进行持久化，则存储中会新增一条状态为待发送的消息

然后返回 ACK 给消息生产者，此时 MQ 不会触发消息推送事件

生产者预发送消息成功后，执行本地事务

执行本地事务，执行完成后，发送执行结果给 MQ

MQ 会根据结果删除或者更新消息状态为可发送

如果消息状态更新为可发送，则 MQ 会 push 消息给消费者，后面消息的消费和普通消息是一的

<h2 id="-正在处理的队列突然断电怎么办-">★ 正在处理的队列突然断电怎么办？ </h2>

正在处理的实现事务功能，下次自动回滚。
队列实现持久化储存，下次启动自动载入。
添加标志位，未处理 0，处理中 1，已处理 2。每次启动的时候，把所有状态为 1 的，置为 0。 </i>
关键性的应用就给电脑配个 UPS。

<h2 id="-服务限流的方式">★ 服务限流的方式</h2>

漏桶：水(请求)先进入到漏桶里,漏桶以一定的速度出水(接口有响应速率),当水流入速度过大会直溢出(访问频率超过接口响应速率),然后就拒绝请求。

令牌桶算法：系统会按恒定 1/QPS 时间间隔(如果 QPS=100,则间隔是 10ms)往桶里加入 Toke，如果桶已经满了就不再加了.新请求来临时,会各自拿走一个 Token,如果没有 Token 就拒绝服务。 </>

基于 redis 实现的限流：假设每分钟访问次数不能超过 10 次，在 Redis 中创建一个键，过期 60 秒，对此服务接口的访问就把键值加 1，在 60 秒内增加到 10 的时候，禁止访问服务接口。

计数器，滑动窗口（假设窗口为 10s,则建立一个大小为 10 的数组，然后每次让当前秒数除 10 落到哪个格子就累加，每一时刻数组的和就是窗口的数值）

令牌桶可以应对突发的大流量

漏斗算法用于请求恒定速率通过

<h2 id="RabbitMQ消息堆积怎么处理-">RabbitMQ 消息堆积怎么处理？ </h2>
新建一个 topic，partition 是原来的 10 倍；然后写一个临时的分发数据的 consumer 程序，这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

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 数据；等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的 consumer 机器来消费消息
<h2 id="-kafka消息会不会丢失-">★kafka 消息会不会丢失？ </h2>
Kafka 消息发送分同步(sync)、异步(async)两种方式。默认是使用同步方式，可通过 producer.type 属性进行配置；Kafka 保证消息被安全生产，有三个选项分别是 0,1,-1。

通过 request.required.acks 属性进行配置：
0 代表：不进行消息接收是否成功的确认(默认值)；
1 代表：当 Leader 副本接收成功后，返回接收成功确认信息；
-1 代表：当 Leader 和 Follower 副本都接收成功后，返回接收成功确认信息；

网络异常：
acks 设置为 0 时，不和 Kafka 集群进行消息接受确认，当网络发生异常等情况时，存在消息丢的可能；
客户端异常：
异步发送时，消息并没有直接发送至 Kafka 集群，而是在 Client 端按一定规则缓存并批量发送在这期间，如果客户端发生死机等情况，都会导致消息的丢失；
缓冲区满了：
异步发送时，Client 端缓存的消息超出了缓冲池的大小，也存在消息丢失的可能；
Leader 副本异常：
acks 设置为 1 时，Leader 副本接收成功，Kafka 集群就返回成功确认信息，而 Follower 副本能还在同步。这时 Leader 副本突然出现异常，新 Leader 副本(原 Follower 副本)未能和其保持一致就会出现消息丢失的情况；
以上就是消息丢失的几种情况，在日常应用中，我们需要结合自身的应用场景来选择不同的配置
想要更高的吞吐量就设置：异步、ack=0；想要不丢失消息数据就选：同步、ack=-1 策略
<h2 id="-RabbitMQ的消息丢失解决方案-">★RabbitMQ 的消息丢失解决方案？ </h2>

消息持久化：Exchange 设置持久化：durable:true；
Queue 设置持久化；
Message 持久化发送。
ACK 确认机制：消息发送确认；消息接收确认。

<h2 id="-kafka的leader副本选举-">★kafka 的 leader 副本选举？ </h2>

如果某个分区 patition 的 Leader 挂了,那么其它跟随者将会进行选举产生一个新的 leader,之后有的读写就会转移到这个新的 Leader 上,在 kafka 中,其不是采用常见的多数选举的方式进行副本的 Leder 选举,而是会在 Zookeeper 上针对每个 Topic 维护一个称为 ISR（in-sync replica，已同步的副）的集合,显然还有一些副本没有来得及同步。只有这个 ISR 列表里面的才有资格成为 leader(先使用 IR 里面的第一个，如果不行依次类推，因为 ISR 里面的是同步副本，消息是最完整且各个节点都是一的)。
通过 ISR,kafka 需要的冗余度较低，可以容忍的失败数比较高。假设某个 topic 有 f+1 个副本，kfka 可以容忍 f 个不可用,当然,如果全部 ISR 里面的副本都不可用,也可以选择其他可用的副本,只是存数据的不一致。

<h2 id="kafka消息的检索-">kafka 消息的检索？ </h2>
其实很简单主要是用二分查找算法,比如我们要查找一条 offest=10000 的文件,kafka 首先会在应分区下的 log 文件里采用二分查看定位到某个记录该 offest
=10000 这条消息的 log,然后从相应的 index 文件定位其偏移量,然后拿着偏移量到 log 里面直接获。这样就完成了一个消息的检索过程。
<h2 id="-RabbitMQ-集群方式-">★RabbitMQ 集群方式？ </h2>

普通集群：

以两个节点（rabbit01、rabbit02）为例来进行说明。rabbit01 和 rabbit02 两个节点仅有相同元数据，即队列的结构，但消息实体只存在于其中一个节点 rabbit01（或者 rabbit02）中。
当消息进入 rabbit01 节点的 Queue 后，consumer 从 rabbit02 节点消费时，RabbitMQ 会临在 rabbit01、rabbit02 间进行消息传输，把 A 中的消息实体取出并经过 B 发送给 consumer。所以onsumer 应尽量连接每一个节点，从中取消息。即对于同一个逻辑队列，要在多个节点建立物理 Que。否则无论 consumer 连 rabbit01 或 rabbit02，出口总在 rabbit01，会产生瓶颈。当 rabbit01点故障后，rabbit02 节点无法取到 rabbit01 节点中还未消费的消息实体。如果做了消息持久化，那得等 rabbit01 节点恢复，然后才可被消费；如果没有持久化的话，就会产生消息丢失的现象。

<ol start="2">
镜像集群：

在普通集群的基础上，把需要的队列做成镜像队列，消息实体会主动在镜像节点间同步，而不是客户端取数据时临时拉取，也就是说多少节点消息就会备份多少份。该模式带来的副作用也很明显，了降低系统性能外，如果镜像队列数量过多，加之大量的消息进入，集群内部的网络带宽将会被这种步通讯大大消耗掉。所以在对可靠性要求较高的场合中适用
由于镜像队列之间消息自动同步，且内部有选举 master 机制，即使 master 节点宕机也不会影整个集群的使用，达到去中心化的目的，从而有效的防止消息丢失及服务不可用等问题

<h2 id="-kafka高性能的原因-">★kafka 高性能的原因？</h2>

Broker NIO 异步消息处理，实现了 IO 线程与业务线程分离；

磁盘顺序写；

零拷贝（跳过用户缓冲区的拷贝，建立一个磁盘空间和内存的直接映射，数据不再复制到用户态冲区）；

分区/分段（每次文件操作都是对一个小文件的操作，非常轻便，同时也增加了并行处理能力）；/p>

批量发送 (可以指定缓存的消息达到某个量的时候就发出去，或者缓存了固定的时间后就发送出，大大减少服务端的 I/O 次数)

数据压缩

<h2 id="-ZooKeeper分布式高可用">★ZooKeeper 分布式高可用</h2>

ZooKeeper 运行期间，集群中至少有过半的机器保存了最新数据。集群超过半数的机器能够正工作，集群就能够对外提供服务。

zookeeper 可以选出 N 台机器作主机，它可以实现 M:N 的备份；keepalive 只能选出 1 台机

作主机，所以 keepalive 只能实现 M:1 的备份。

通常有以下两种部署方案：双机房部署（一个稳定性更好、设备更可靠的机房，这个机房就是主机房，而另外一个机房则更加廉价一些，例如，对于一个由 7 台机器组成的 ZooKeeper 集群，通常主要机房中部署 4 台机器，剩下的 3 台机器部署到另外一个机房中）；三机房部署（无论哪个机房发了故障，剩下两个机房的机器数量都超过半数。在三个机房中都部署若干个机器来组成一个 ZooKeeper 集群。假设机器总数为 N，各机房机器数：N1 = (N-1)/2 ， N2=1~(N-N1)/2 ， N3 = N - N1 - N2 ）。

水平扩容就是向集群中添加更多机器，Zookeeper2 种方式（不完美），一种是集群整体重启，外一种是逐台进行服务器的重启。

<h2 id="-如何设计秒杀">★ 如何设计秒杀</h2>

对于大促时候的秒杀活动，一般运营会配置静态的活动页面，配置静态活动页面主要有两个目的方面是为了便于在各种社交媒体分发，另一方面是因为秒杀活动页的流量是大促期间最大的，通过配成静态页面可以将页面发布在公有云上动态的横向扩展；

将秒杀活动的静态页面提前刷新到 CDN 节点，通过 CDN 节点的页面缓存来缓解访问压力和公网络带宽，CDN 上缓存 js、css 和图片；

将活动 H5 页面部署在公有云的 web server 上，使用公有云最大的好处就是能够根据活动的火程度动态扩容而且成本较低，同时将访问压力隔离在公司系统外部；

在提供真正商品秒杀业务功能的 app server 上，需要进行交易限流、熔断控制，防止因为秒杀易影响到其他正常服务的提供，我们在限流和熔断方面使用了 hystrix，在核心交易的 controller 层过 hystrix 进行交易并发限流控制，当交易流量超出我们设定的限流最大值时，会对新交易进行熔断理固定返回静态失败报文。

服务降级处理，除了上面讲到的限流和熔断控制，我们还设定了降级开关，对于首页、购物车、单查询、大数据等功能都会进行一定程度的服务降级，例如我们会对首页原先动态生成的大数据页面局降级为所有人看到的是一样的页面、购物车也会降级为不在一级页面的 tabbar 上的购物车图标上示商品数量、历史订单的查询也会提供时间周期较短的查询、大数据商品推荐也会提供一样的商品推，通过这样的降级处理能够很好的保证各个系统在大促期间能够正常的提供最基本的服务，保证用户够正常下单完成付款。

上面介绍的都是如何保证能扛住高并发，下面介绍下整个方案中如何防止超卖现象的发生，我们常的下单过程中防止超卖一般是通过在数据库上实施乐观锁来完成，使用乐观锁虽然比 for update 种悲观锁方式性能要好很多，但是还是无法满足秒杀的上万并发需求，我们的方案其实也很简单实时存的扣减在缓存中进行，异步扣减数据库中的库存，保证缓存中和数据库中库存的最终一致性。

在这个方案中我们使用的分布式缓存是 redis，使用了 codis 集群方案稳定性和高可用方面还是较有保证的，因为 redis 是单线程写，所以也不用担心线程安全的问题，redis 自身就能够保证数据

强一致性，在下单的事务中包含了实时扣减缓存中的库存和异步发送队列，由队列处理器再异步从队中取出订单根据订单信息扣减库存系统数据库中的商品数量。

<h2 id="高性能统计UV的方式-">高性能统计 UV 的方式？</h2>
（1）使用 redis 的 set 集合
（2）使用 redis 的 bitmap（注意内存消耗）
<h2 id="-缓存击穿的解决办法">★ 缓存击穿的解决办法</h2>

加载 DB 时同步，其他则等待；DB 端做 SQL 合并，Queue 合并排队处理；
部分缓存设置为永不过期；
读取数据时候则等待 500ms，500ms 缓存应该已经加载完成；

<h2 id="-后台系统怎么防止请求重复提交-">★ 后台系统怎么防止请求重复提交。</h2>
前端 js，控制按钮。前端放置令牌。
数据库唯一索引。redis 看 key 是否存在。或者数据库字段状态。
<h2 id="有没有遇到进线上GC-出现的症状是什么样的-怎么解决的-">有没有遇到进线上 GC，出现症状是什么样的，怎么解决的？</h2>
利用堆快照，查看到底是哪些对象占用大量内存导致经常 gc
<h2 id="-假如你的项目出现性能瓶颈了-你觉得可能会是哪些方面-怎么解决问题-">★ 假如你的项目现性能瓶颈了，你觉得可能会是哪些方面，怎么解决问题。</h2>
DB 层面，有可能是 sql，索引，表过大，数据库压力。
缓存层面：有可能缓存命中率差,redis 性能瓶颈，需要扩容
服务器压力：服务器处理瓶颈
Java 层面:代码写法
前端层面：cdn 压力，页面压力
<h2 id="情景题-如果一个外卖配送单子要发布-现在有200个骑手都想要接这一单-如何保证只有一骑手接到单子-">情景题：如果一个外卖配送单子要发布，现在有 200 个骑手都想要接这一单，如何证只有一个骑手接到单子？</h2>
分布式锁，或者幂等接口，CAS 乐观锁
<h2 id="场景题-美团首页每天会从10000个商家里面推荐50个商家置顶-每个商家有一个权值-你如来推荐-第二天怎么更新推荐的商家-">场景题：美团首页每天会从 10000 个商家里面推荐 50 个商家顶，每个商家有一个权值，你如何来推荐？第二天怎么更新推荐的商家？</h2>
可以借鉴下 stackoverflow，视频网站等等的推荐算法。
<h2 id="场景题-微信抢红包问题">场景题：微信抢红包问题</h2>
悲观锁，乐观锁，存储过程放在 mysql 数据库中。
<h2 id="场景题-1000个任务-分给10个人做-你怎么分配-先在纸上写个最简单的版本-然后优化-">景题：1000 个任务，分给 10 个人做，你怎么分配，先在纸上写个最简单的版本，然后优化。</h2>
全局队列，把 1000 任务放在一个队列里面，然后每个人都是取，完成任务。
分为 10 个队列，每个人分别到自己对应的队列中去取务。
<h2 id="分布式服务如何跟踪-">分布式服务如何跟踪？</h2>
调用可以实现跟踪系统，可以在业务日志中添加调用链 ID，各个环节 RPC 均添加调用时延,QPS 等。
非业务组件应该少加入业务代码，服务调用采用买点，也会采用配置采样率方式，买点即当前节的上下文信息，包含 TraceId，RPCId，开始结束时间，类型，协议，调用方 IP，端口，服务名等，及其他异常信息，报文等扩展，日志采用离线 + 实时的如 flume 结合 kafka 等，应按照 TraceId 汇日志后按 RPCId 顺序整理。
<h2 id="Sentinel-工作原理-">Sentinel 工作原理？</h2>

每个 Sentinel 以每秒钟一次的频率向它所知的 Master，Slave 以及其他 Sentinel 实例发送一个 ING 命令；
2 如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel 标记为主观下线；
如果一个 Master 被标记为主观下线，则正在监视这个 Master 的所有 Sentinel 要以每秒一次的率确认 Master 的确进入了主观下线状态；

当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认 Master 的确入了主观下线状态，则 Master 会被标记为客观下线；
在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有 Master，Slave 发送 INO 命令；当 Master 被 Sentinel 标记为客观下线时，Sentinel 向下线的 Master 的所有 Slave 发送 IFO 命令的频率会从 10 秒一次改为每秒一次；
若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的客观下线状态就会被移除；
若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的主观下线状态就会被移除。/li>

<h2 id="redis的主从-">redis 的主从？</h2>
监控（Monitoring）： Redis Sentinel 实时监控主服务器和从服务器运行状态；
自动故障转移：如果一个 master 不正常运行了，哨兵可以启动一个故障转移进程，将一个 slave 升成为 master，其他的 slave 被重新配置使用新的 master，并且应用程序使用 Redis 服务端通知的新址；
<h2 id="讲讲分布式唯一ID-">讲讲分布式唯一 ID。</h2>
雪花算法：

1 bit：不用，为啥呢？因为二进制里第一个 bit 为如果是 1，那么都是负数，但是我们生成的 id 都是正数，所以第一个 bit 统一都是 0。
41 bit：表示的是时间戳，单位是毫秒。41 bit 可以表示的数字多达 2^41 - 1，也就是可以标识 ^41 - 1 个毫秒值，换算成年就是表示 69 年的时间。
10 bit：记录工作机器 id，代表的是这个服务最多可以部署在 2^10 台机器上哪，也就是 1024 机器。但是 10 bit 里 5 个 bit 代表机房 id，5 个 bit 代表机器 id。意思就是最多代表 2^5 个机房（3 个机房），每个机房里可以代表 2^5 个机器（32 台机器）。
12 bit：这个是用来记录同一个毫秒内产生的不同 id，12 bit 可以代表的最大正整数是 2^12 - 1 = 4096，也就是说可以用这个 12 bit 代表的数字来区分同一个毫秒内的 4096 个不同的 id。

<h2 id="-什么是一致性hash">★ 什么是一致性 hash</h2>
利用哈希环进行一致性哈希
<h2 id="-如何使用redis和zookeeper实现分布式锁-有什么区别优缺点-会有什么问题-分别适用什场景--延伸-如果知道redlock-讲讲他的算法实现-争议在哪里-">★ 如何使用 redis 和 zookeeper 实分布式锁？有什么区别优缺点，会有什么问题，分别适用什么场景。（延伸：如果知道 redlock，讲他的算法实现，争议在哪里）</h2>
Redis 实现比较复杂，流程如下：

根据 lockKey 区进行 setnx（set not exist，顾名思义，如果 key 值为空，则正常设置，返回 1 否则不会进行设置并返回 0）操作，如果设置成功，表示已经获得锁，否则并没有获取锁。

如果没有获得锁，去 Redis 上拿到该 key 对应的值，在该 key 上我们存储一个时间戳（用毫秒示，t1），为了避免死锁以及其他客户端占用该锁超过一定时间（5 秒），使用该客户端当前时间戳与存储的时间戳作比较。

如果没有超过该 key 的使用时限，返回 false，表示其他人正在占用该 key，不能强制使用；如已经超过时限，那我们就可以进行解锁，使用我们的时间戳来代替该字段的值。

但是如果在 setnx 失败后，get 该值却无法拿到该字段时，说明操作之前该锁已经被释放，这个候，最好的办法就是重新执行一遍 setnx 方法来获取其值以获得该锁。

缺点：有可能 master 崩溃，导致多节点获取到锁。

从实现难度上来说，Zookeeper 实现非常简单，实现分布式锁的基本逻辑：

客户端调用 create()方法创建名为“locknode/guid-lock-”的节点，需要注意的是，这里节点创建类型需要设置为 EPHEMERAL_SEQUENTIAL。
客户端调用 getChildren(“locknode”)方法来获取所有已经创建的子节点。
客户端获取到所有子节点 path 之后，如果发现自己在步骤 1 中创建的节点是所有节点中序号最的，那么就认为这个客户端获得了锁。
如果创建的节点不是所有节点中需要最小的，那么则监视比自己创建节点的序列号小的最大的节，进入等待。直到下次监视的子节点变更的时候，再进行子节点的获取，判断是否获取锁。

区别：

Redis 分布式锁，需要自己不断去尝试获取锁，比较消耗性能

ZooKeeper 分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开较小

如果 Redis 获取锁的那个客户端挂了，那么只能等待超时时间之后才能释放锁

而对于 ZooKeeper，因为创建的是临时 znode，只要客户端挂了，znode 就没了，此时就自动放锁

redlock 算法实现：
假设有 5 个完全独立的 redis 主服务器

获取当前时间戳

client 尝试按照顺序使用相同的 key,value 获取所有 redis 服务的锁，在获取锁的过程中的获取间比锁过期时间短很多，这是为了不要过长时间等待已经关闭的 redis 服务。并且试着获取下一个 red s 实例。比如：TTL 为 5s,设置获取锁最多用 1s，所以如果一秒内无法获取锁，就放弃获取这个锁，而尝试获取下个锁

client 通过获取所有能获取的锁后的时间减去第一步的时间，这个时间差要小于 TTL 时间并且至有 3 个 redis 实例成功获取锁，才算真正的获取锁成功

如果成功获取锁，则锁的真正有效时间是 TTL 减去第三步的时间差 的时间；比如：TTL 是 5s,获所有锁用了 2s,则真正锁有效时间为 3s(其实应该再减去时钟漂移);

如果客户端由于某些原因获取锁失败，便会开始解锁所有 redis 实例；因为可能已经获取了小于 3 个锁，必须释放，否则影响其他 client 获取锁
<pre><code class="highlight-chroma"> redlock的争议点：(fgc导致的问题)

 对于提升效率的景下，RedLock 太重。
 对于对正确性要极高的场景下，RedLock 并不能保证正确性。
</code></pre>

<h2 id="2pc-3pc-的区别-解决了哪些问题-">2pc 3pc 的区别，解决了哪些问题。</h2>
3pc 将 2pc 中的一阶段拆为 canCommit 和 prepareCommit
二阶段提交有几个缺点：

同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他三方节点访问公共资源不得不处于阻塞状态。单点故障。由于协调者的重要性，一旦协调者发生故障参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解因为协调者宕机导致的参与者处于阻塞状态的问题）数据不一致。在二阶段提交的阶段二中，当协调向参与者发送 commit 请求之后，发生了局部网络异常或者在发送 commit 请求过程中协调者发生故障，这回导致只有一部分参与者接受到了 commit 请求。而在这部分参与者接到 commit 请求之就会执行 commit 操作。但是其他部分未接到 commit 请求的机器则无法执行事务提交。于是整个布式系统便出现了数据部一致性的现象。
二阶段无法解决的问题：协调者再发出 commit 消息之后宕机，而唯一接收到这条消息的参与者同时宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道务是否被已经提交。

3pc 比 2pc 减少事务阻塞范围 。3pc 在超时后会自动提交。相对于 2PC，3PC 主要解决的单点障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行 commit

<h2 id="什么是paxos算法--什么是zab协议-">什么是 paxos 算法， 什么是 zab 协议。</h2>
<pre><code class="highlight-chroma">paxos算法的推导，
如果只有一个人投的话，那么每个人必须接受第一个提议。
这样又会导致三个分别投不同的票，形不成大多数。
推导出可以选多次，但多次票有可能投自己又投别人，形成多个大多数，所以又加了限制条件，只能同意之前同意过的

再推导出当两个机重连的话，机子必须接受第一个发给他的提案，这样就违背了之前选好的。
所以当服务器重连时候，必须发给他之前同意好的提案。
https://blog.51cto com/12615191/2086264
</code></pre>
zab 协议：
原子广播：

ZAB 协议的消息广播过程使用的是一个原子广播协议，类似一个 二阶段提交过程。对于客户端送的写请求，全部由 Leader 接收，Leader 将请求封装成一个事务 Proposal，将其发送给所有 Follw

r，然后，根据所有 Follwer 的反馈，如果超过半数成功响应，则执行 commit 操作（先提交自己，发送 commit 给所有 Follwer）。
崩溃恢复：

针对这些问题，ZAB 定义了 2 个原则：

ZAB 协议确保那些已经在 Leader 提交的事务最终会被所有服务器提交。

ZAB 协议确保丢弃那些只在 Leader 提出/复制，但没有提交的事务。

如果让 Leader 选举算法能够保证新选举出来的 Leader 服务器拥有集群总所有机器编号（即 ZXI 最大）的事务，那么就能够保证这个新选举出来的 Leader 一定具有所有已经提交的提案。

当 Follower 链接上 Leader 之后，Leader 服务器会根据自己服务器上最后被提交的 ZXID 和 Fol ower 上的 ZXID 进行比对，比对结果要么回滚，要么和 Leader 同步。

<h2 id="-Dubbo的原理-有看过源码么-数据怎么流转的-怎么实现集群-负载均衡-服务注册和发现-试转发-快速失败的策略是怎样的--">★Dubbo 的原理，有看过源码么，数据怎么流转的，怎么实现群，负载均衡，服务注册和发现，重试转发，快速失败的策略是怎样的 。</h2>

第一层：service 层，接口层，给服务提供者和消费者来实现的
第二层：config 层，配置层，主要是对 dubbo 进行各种配置的
第三层：proxy 层，服务代理层，无论是 consumer 还是 provider，dubbo 都会给你生成代理代理之间进行网络通信
第四层：registry 层，服务注册层，负责服务的注册与发现
第五层：cluster 层，集群层，封装多个服务提供者的路由以及负载均衡，将多个实例组合成一服务
第六层：monitor 层，监控层，对 rpc 接口的调用次数和调用时间进行监控
第七层：protocal 层，远程调用层，封装 rpc 调用
第八层：exchange 层，信息交换层，封装请求响应模式，同步转异步
第九层：transport 层，网络传输层，抽象 mina 和 netty 为统一接口
第十层：serialize 层，数据序列化层

<h2 id="-一次RPC请求的流程是什么-">★ 一次 RPC 请求的流程是什么。</h2>

服务消费方（client）调用以本地调用方式调用服务；
client stub 接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体；
client stub 找到服务地址，并将消息发送到服务端；
server stub 收到消息后进行解码；
server stub 根据解码结果调用本地的服务；
本地服务执行并将结果返回给 server stub；
server stub 将返回结果打包成消息并发送至消费方；
client stub 接收到消息，并进行解码；
服务消费方得到最终结果。

<h2 id="-解释什么是MESI协议-缓存一致性--">★ 解释什么是 MESI 协议(缓存一致性)。</h2>
MESI 协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当 CPU 写数时，如果发现操作的变量是共享变量，即在其他 CPU 中也存在该变量的副本，会发出信号通知其他 C

U 将该变量的缓存行置为无效状态，因此当其他 CPU 需要读取这个变量时，发现自己缓存中缓存该量的缓存行是无效的，那么它就会从内存重新读取。
（另外一种硬件层面的解决是总线锁）
<h2 id="Zookeeper的用途-选举的原理是什么-适用场景-">Zookeeper 的用途，选举的原理是什，适用场景。</h2>
用途：类似文件系统的分布式协调服务。
选举原理：

Zookeeper 集群中只有超过半数以上的服务器启动，集群才能正常工作；
在集群正常工作之前，myid 小的服务器给 myid 大的服务器投票，直到集群正常工作，选出 Leaer；
选出 Leader 之后，之前的服务器状态由 Looking 改变为 Following，以后的服务器都是 Followr。
适用场景：
命名服务
配置管理
集群管理
分布式锁
队列管理

<h2 id="Zookeeper-watch机制原理-">Zookeeper watch 机制原理。</h2>

客户端注册 Watcher 到服务端;
服务端发生数据变更;
服务端通知客户端数据变更;
客户端回调 Watcher 处理变更应对逻辑;

<h2 id="什么叫数据一致性-你怎么理解数据一致性-">什么叫数据一致性，你怎么理解数据一致性。/h2>

一致性又可以分为强一致性与弱一致性。
强一致性可以理解为在任意时刻，所有节点中的数据是一样的。同一时间点，你在节点 A 中获取 key1 的值与在节点 B 中获取到 key1 的值应该都是一样的。
弱一致性包含很多种不同的实现，目前分布式系统中广泛实现的是最终一致性。
所谓最终一致性，就是不保证在任意时刻任意节点上的同一份数据都是相同的，但是随着时间的移，不同节点上的同一份数据总是在向趋同的方向变化。也可以简单的理解为在一段时间后，节点间数据会最终达到一致状态。

<h2 id="请思考一个方案-实现分布式环境下的countDownLatch-">请思考一个方案，实现分布式境下的 countDownLatch。</h2>
zookeeper，判断某个节点下的子节点到达一定数目后，则执行，否则等待。
<h2 id="-用过哪些MQ-怎么用的-和其他mq比较有什么优缺点-MQ的连接是线程安全的吗">★ 用哪些 MQ，怎么用的，和其他 mq 比较有什么优缺点，MQ 的连接是线程安全的吗</h2>
<table>
<thead>
<tr>
<th>特性</th>
<th>ActiveMQ</th>
<th>RabbitMQ</th>
<th>RocketMQ</th>
<th>Kafka</th>
</tr>
</thead>
<tbody>

<tr>
<td>单机吞吐量</td>
<td>万级，比 RocketMQ、Kafka 低一个数量级</td>
<td>同 ActiveMQ</td>
<td>10 万级，支撑高吞吐</td>
<td>10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景</td>
</tr>
<tr>
<td>topic 数量对吞吐量的影响</td>
<td></td>
<td></td>
<td>topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，同等机器下，可以支撑大量的 topic</td>
<td>topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量要过多，如果要支撑大规模的 topic，需要增加更多的机器资源</td>
</tr>
<tr>
<td>时效性</td>
<td>ms 级</td>
<td>微秒级，这是 RabbitMQ 的一大特点，延迟最低</td>
<td>ms 级</td>
<td>延迟在 ms 级以内</td>
</tr>
<tr>
<td>可用性</td>
<td>高，基于主从架构实现高可用</td>
<td>同 ActiveMQ</td>
<td>非常高，分布式架构</td>
<td>非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用</td>
</tr>
<tr>
<td>消息可靠性</td>
<td>有较低的概率丢失数据</td>
<td>基本不丢</td>
<td>经过参数优化配置，可以做到 0 丢失</td>
<td>同 RocketMQ</td>
</tr>
<tr>
<td>功能支持</td>
<td>MQ 领域的功能极其完备</td>
<td>基于 erlang 开发，并发能力很强，性能极好，延时很低</td>
<td>MQ 功能较为完善，还是分布式的，扩展性好</td>
<td>功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使
</td>
</tr>
</tbody>
</table>
<h2 id="MQ系统的数据如何保证不丢失">MQ 系统的数据如何保证不丢失</h2>
发送消息后和接收消息后 确认机制 加上持久化
<h2 id="MQ有可能发生重复消费-如何避免-如何做到幂等-">MQ 有可能发生重复消费，如何避免如何做到幂等。</h2>
唯一主键，或者使用 redis 做 id，
<h2 id="异步模式的用途和意义-">异步模式的用途和意义。</h2>
避免阻塞

<h2 id="使用kafka有没有遇到什么问题-怎么解决的-">使用 kafka 有没有遇到什么问题，怎么解决。</h2>
https://www.cnblogs.com/leaves1024/p/11073191.html
https://blog.csdn.net/chizizhixin/article/details/78563595
https://blog.csdn.net/lsh2366254/article/details/84910011
<h2 id="如何保证消息的有序性-消息处理的有序性-">如何保证消息的有序性。消息处理的有序性。</h2>
使用同一个 queue
<h2 id="消息的重发补发策略">消息的重发补发策略</h2>

实时队列采用双队列模式，生产者将行为记录写入 Queue1，worker 服务从 Queue1 消费新鲜数据，如果异常则写入 Queue2（主要保存异常数据），RetryWorker 会监听 Queue2，消费异常数据，如果还未处理成功按照一定的策略等待或者将异常数据再写入 Queue2，如果数据发生积压可以调整 worker 的消费游标，从最新数据重新开始消费，保证了最新 data 得到处理，中间未处理的一段则可启动 backupWorker 指定起止游标在消费完指定区间的数据后，backupWorker 会自动停止。

DB 降级开关后，可直接写入 redis（storm），同时将数据写入一份到 Retry 队列，在开启 DB 级开关后消费 Retry 队列中的数据，从而把数据写入到 mysql 中，达到最终一致性。MYSQL 切分为片为 2 的 N 次方，例如原来分为两个库 d0 和 d1 均放在 s0 服务器上，s0 同时有备机 s1，扩容只几步骤：确保 s0 到 s1 服务器同步顺利，没有明显延迟；s0 暂时关闭读写权限；确保 s1 已经完全同到 s0 更新；s1 开放读写权限；d1 的 dns 由 s0 切换到 s1；s0 开放读写权限。

转自我的 github
<h5 id="技术讨论群QQ-1398880">技术讨论群 QQ:1398880</h5>