

分布式面试题

分布式基础

集群、分布式、SOA、微服务的概念及区别

集群：不同服务器部署同一套应用服务对外提供访问，实现服务的负载均衡或者互备(热备，主从等)，指同一种组件的多个实例，形成的逻辑上的整体。单个节点可以提供完整服务。集群是物理形态

分布式：服务的不同模块部署在不同的服务器上，单个节点不能提供完整服务，需要多节点协调提供服务(也可以是相同组件部署在不同节点、但节点间通过交换信息协作提供服务)，分布式强调的是工作方式

SOA：面向服务的架构，一种设计方法，其中包含多个服务，服务之间通过相互依赖最终提供一系列的功能。一个服务通常以独立的形式存在于操作系统进程中。各个服务之间通过网络调用。

- 中心化实现：ESB(企业服务总线)，各服务通过ESB进行交互，解决异构系统之间的连通性，通过协议转换、消息解析、消息路由把服务提供者的数据传送到服务消费者。很重，有一定的逻辑，可以解决一些公用逻辑的问题。
- 去中心化实现：微服务

微服务：在 SOA 上做的升华，微服务架构强调的一个重点是业务需要彻底的组件化和服务化，原有的单个业务系统会拆分为多个可以独立开发、设计、运行的小应用。这些小应用之间通过服务完成交互和集成

服务单一职责

轻量级通信：去掉ESB总线，采用restapi通信

简述CAP理论

- 数据一致性(consistency)：如果系统对一个写操作返回成功，那么之后的读请求都必须读到这个新数据；如果返回失败，那么所有读操作都不能读到这个数据，对调用者而言数据具有强一致性(strong consistency)
- 服务可用性(availability)：所有读写请求在一定时间内得到响应，可终止、不会一直等待
- 分区容错性(partition-tolerance)：在网络分区的情况下，被分隔的节点仍能正常对外服务

如果选择了 CA 而放弃了 P，那么当发生分区现象时，为了保证 C，系统需要禁止写入，当有写入请求时，系统返回 error（例如，当前系统不允许写入），这又和 A 冲突了，因为 A 要求返回 no error 和 no timeout。因此，分布式系统理论上不可能选择 CA 架构，只能选择 CP 或者 AP 架构。

反证：

如果 CAP 三者可同时满足，由于允许 P 的存在，则一定存在节点之间的丢包，如此则不能保证 C

因为允许分区容错，写操作可能在节点 1 上成功，在节点 2 上失败，这时候对于 Client 1 (读取节点1) 和 Client 2(读取节点2)，就会读取到不一致的值，出现不一致的情况。如果保持一致性，写操作必须同时失败，也就是降低系统的可用性。

简述Base理论

cap理论的一种妥协，由于cap只能二取其一，base理论降低了发生分区容错时对可用性和一致性的要求

- 1、基本可用：允许可用性降低（可能响应延长、可能服务降级），
- 2、软状态：指允许系统中的数据存在中间状态，并认为该中间状态不会影响系统整体可用性。
- 2、最终一致性：节点数据同步可以存在时延），但在一定的期限后必须达成数据的一致，状态变为最终状态

数据一致性模型有哪些

强一致性：当更新操作完成之后，任何多个后续进程的访问都会返回最新的更新过的值，这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据 CAP 理论，这种实现需要牺牲可用性。

弱一致性：系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。用户读到某一操作对系统数据的更新需要一段时间，我们称这段时间为“不一致性窗口”。

最终一致性：最终一致性是弱一致性的特例，强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。到达最终一致性的时间，就是不一致窗口时间，在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。

最终一致性模型根据其提供的不同保证可以划分为更多的模型，包括因果一致性和会话一致性等。

因果一致性：要求有因果关系的操作顺序得到保证，非因果关系的操作顺序则无所谓。

进程 A 在更新完某个数据项后通知了进程 B，那么进程 B 之后对该数据项的访问都应该能够获取到进程 A 更新后的最新值，并且如果进程 B 要对该数据项进行更新操作的话，务必基于进程 A 更新后的最新值。

在微博或者微信进行评论的时候，比如你在朋友圈发了一张照片，朋友给你评论了，而你对朋友的评论进行了回复，这条朋友圈的显示中，你的回复必须在朋友之后，这是一个因果关系，而其他没有因果关系的数据，可以允许不一致。

会话一致性：将对系统数据的访问过程框定在了一个会话当中，约定了系统能保证在同一个有效的会话中实现“读己之所写”的一致性，就是在你的一次访问中，执行更新操作之后，客户端能够在同一个会话中始终读取到该数据项的最新值。实际开发中有分布式的 Session 一致性问题，可以认为是会话一致性的一个应用。

选举算法Quorum 机制、WARO

waro：一种简单的副本控制协议，写操作时、只有当所有的副本都更新成功之后，这次写操作才算成功，否则视为失败。优先保证读、任何节点读到的数据都是最新数据，牺牲了更新服务的可用性、只要有一个副本宕机了，写服务就不会成功。但只要有一个节点存活、仍能提供读服务

Quorum 机制：10个副本，一次成功更新了三个，那么至少需要读取八个副本的数据，可以保证读到了最新的数据。无法保证强一致性，也就是无法实现任何时刻任何用户或节点都可以读到最近一次成功提交的副本数据。需要配合一个获取最新成功提交的版本号的 metadata 服务，这样可以确定最新已经成功提交的版本号，然后从已经读到的数据中就可以确认最新写入的数据

简述paxos算法

Paxos算法解决的是一个分布式系统如何就某个值（决议）达成一致。一个典型的场景是，在一个分布式数据库系统中，如果各个节点的初始状态一致，每个节点执行相同的操作序列，那么他们最后能够得到一个一致的状态。为了保证每个节点执行相同的操作序列，需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令一致。在Paxos算法中，有三种角色：Proposer (提议者), Acceptor (接受者), Learners (记录员)

- Proposer提议者：只要Proposer发的提案Propose被半数以上的Acceptor接受，Proposer就认为该提案的value被选定了。
- Acceptor接受者：只要Acceptor接受了某个提案，Acceptor就认为该提案的value被选定了
- Learner记录员：Acceptor告诉Learner哪个value被选定，Learner就认为哪个value被选定。

Paxos算法分为两个阶段，具体如下：

阶段一（prepare）：

(a) Proposer 收到client请求或者发现本地有未提交的值，选择一个提案编号 N，然后向半数以上的 Acceptor 发送编号为 N 的 Prepare 请求。

(b) Acceptor 收到一个编号为 N 的 Prepare 请求，如果该轮paxos

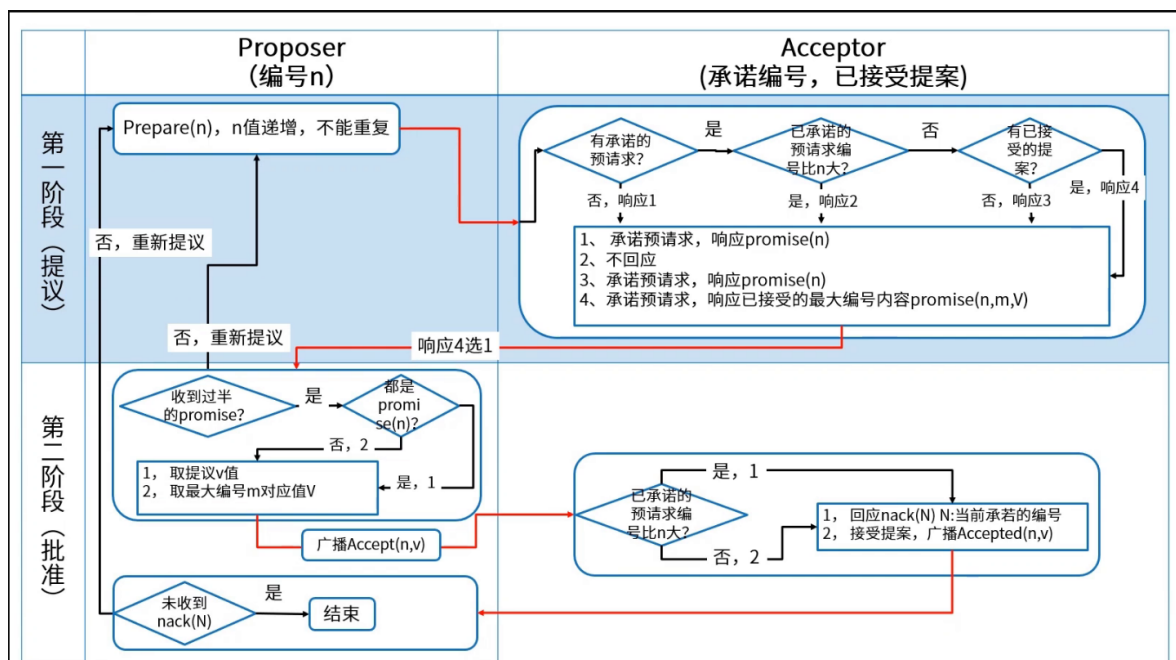
- 本节点已经有已提交的value记录，对比记录的编号和N，大于N则拒绝回应，否则返回该记录value及编号
- 没有已提交记录，判断本地是否有编号N1，N1>N、则拒绝响应，否则将N1改为N（如果没有N1，则记录N），并响应prepare

阶段二（accept）：

(a) 如果 Proposer 收到半数以上 Acceptor 对其发出的编号为 N 的 Prepare 请求的响应，那么它就会发送一个针对[N,V]提案的 Accept 请求给半数以上的 Acceptor。V 就是收到的响应中编号最大的value，如果响应中不包含任何value，那么V就由 Proposer 自己决定。

(b) 如果 Acceptor 收到一个针对编号为 N 的提案的 Accept 请求，Acceptor对比本地的记录编号，如果小于等于N，则接受该值，并提交记录value。否则拒绝请求

Proposer 如果收到的大多数Acceptor响应，则选定该value值，并同步给leaner，使未响应的Acceptor达成一致



活锁: accept时被拒绝, 加大N, 重新accept, 此时另外一个proposer也进行相同操作, 导致accept一致失败, 无法完成算法

multi-paxos: 区别于paxos只是确定一个值, multi-paxos可以确定多个值, 收到accept请求后, 则一定时间内不再accept其他节点的请求, 以此保证后续的编号不需要在经过preprae确认, 直接进行accept操作。此时该节点成为了leader, 直到accept被拒绝, 重新发起prepare请求竞争leader资格。

简述raft算法

概念:

- 分布式一致性算法: raft会先选举出leader, leader完全负责replicated log的管理。leader负责接受所有客户端更新请求, 然后复制到follower节点, 并在“安全”的时候执行这些请求。如果leader故障, follower会重新选举出新的leader
- 三种状态: 一个节点任一时刻处于三者之一
 - leader: 处理所有的客户端请求 (如果客户端将请求发给了Follower, Follower将请求重定向给Leader)
- follower: 不会发送任何请求, 只会简单地响应来自Leader或Candidate的请求
- candidate: 用于选举产生新的leader(候选人)
- term: 任期, leader产生到重新选举为一任期, 每个节点都维持着当前的任期号
 - term是递增的, 存储在log日志的entry中, 代表当前entry是在哪一个term时期写入
 - 每个任期只能有一个leader或者没有(选举失败)
 - 每次rpc通信时传递该任期号, 如果RPC收到任期号大于本地的、切换为follower, 小于本地任期号则返回错误信息
- 两个RPC通信:
 - RequestVote RPC: 负责选举, 包含参数lastIndex, lastTerm
 - AppendEntries RPC: 负责数据的交互。
- 日志序列: 每一个节点上维持着一份持久化Log, 通过一致性协议算法, 保证每一个节点中的Log保持一致, 并且顺序存放, 这样客户端就可以在每一个节点中读取到相同的数据

- 状态机：日志序列同步到多数节点时，leader将该日志提交到状态机，并在下一次心跳通知所有节点提交状态机（携带最后提交的lastIndex）

何时触发选举：

- 集群初始化时，都是follower，随机超时，变成candidate，发起选举
- 如果follower在election timeout内没有收到来自leader的心跳，则主动触发选举

选举过程：发出选举的节点角度

- 1、增加节点本地的term，切换到candidate状态
- 2、投自己一票

其他节点投票逻辑：每个节点同一任期最多只能投一票，候选人知道的信息不能比自己少（通过副本日志和 安全机制保障），先来先得

- 3、并行给其他节点发送RequestVote RPCs(选举请求)、包含term参数
- 4、等待回复

4.1、收到majority(大多数)的投票，赢得选举，切换到leader状态，立刻给所有节点发心跳消息

4.2、被告知别人当选，切换到follower状态。（原来的leader对比term，比自己的大，转换到follower状态）

4.3、一段时间没收到majority和leader的心跳通知，则保持candidate、重新发出选举

日志序列同步：日志需要存储在磁盘持久化，崩溃可以从日志恢复

- 1、客户端发送命令给Leader。
 - 2、Leader把日志条目加到自己的日志序列里。
 - 3、Leader发送AppendEntries RPC请求给所有的follower。携带了prevLogIndex, prevLogTerm
- follower收到后，进行日志序列匹配

- 匹配上则追加到自己的日志序列
- 匹配不上则拒绝请求，leader将日志index调小，重新同步直至匹配上，follower将leader的日志序列覆盖到本地

一旦新的日志序列条目变成majority的了，将日志序列应用到状态机中

- Leader在状态机里提交自己日志序列条目，然后返回结果给客户端
- Leader下次发送AppendEntries RPC时，告知follower已经提交的日志序列条目信息(lastIndex)
- follower收到RPC后，提交到自己的状态机里

提交状态机时，如果term为上一任期，必须与当前任期数据一起提交，否则可能出现覆盖已提交状态机的日志

新选举出的leader一定拥有所有已提交状态机的日志条目

- leader在当日志序列条目已经复制到大多数follower机器上时，才会提交日志条目。

- 而选出的leader的logIndex必须大于等于大多数节点，因此leader肯定有最新的日志

安全原则：

- 选举安全原则：对于一个给定的任期号，最多只会有一人被选举出来
- 状态机安全原则：如果一个leader已经在给定的索引值位置的日志条目应用到状态机中，那么其他任何的服务器在这个索引位置不会提交一个不同的日志
- 领导人完全原则：如果某个日志条目在某个任期号中已经被提交，那么这个条目必然出现在更大任期号的所有领导人中
- 领导人只附加原则：领导人绝对不会删除或者覆盖自己的日志，只会增加
- 日志匹配原则：如果两个日志在相同的索引位置的日志条目的任期号相同，那么我们就认为这个日志从头到这个索引位置之间全部完全相同

简述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，实现分布式数据一致性

所有客户端的请求都是写入到 Leader 进程中，然后，由 Leader 同步到其他节点，称为 Follower。在集群数据同步的过程中，如果出现 Follower 节点崩溃或者 Leader 进程崩溃时，都会通过 Zab 协议来保证数据一致性

ZAB 协议包括两种基本的模式：**崩溃恢复和消息广播**。

消息广播：

集群中所有的事务请求都由 Leader 节点来处理，其他服务器为 Follower，Leader 将客户端的事务请求转换为事务 Proposal，并且将 Proposal 分发给集群中其他所有的 Follower。

完成广播之后，Leader 等待 Follower 反馈，当有过半数的 Follower 反馈信息后，Leader 将再次向集群内 Follower 广播 Commit 信息，Commit 信息就是确认将之前的 Proposal 提交。

Leader 节点的写入是一个两步操作，第一步是广播事务操作，第二步是广播提交操作，其中过半数指的是反馈的节点数 $\geq N/2+1$ ，N 是全部的 Follower 节点数量。

崩溃恢复：

- 初始化集群，刚刚启动的时候
- Leader 崩溃，因为故障宕机
- Leader 失去了半数的机器支持，与集群中超过一半的节点断连

此时开启新一轮 Leader 选举，选举产生的 Leader 会与过半的 Follower 进行同步，使数据一致，当与过半的机器同步完成后，就退出恢复模式，然后进入消息广播模式

整个 ZooKeeper 集群的一致性保证就是在上面两个状态之前切换，当 Leader 服务正常时，就是正常的消息广播模式；当 Leader 不可用时，则进入崩溃恢复模式，崩溃恢复阶段会进行数据同步，完成以后，重新进入消息广播阶段。

Zxid 是 Zab 协议的一个事务编号，Zxid 是一个 64 位的数字，其中低 32 位是一个简单的单调递增计数器，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 周期年代的编号。

Leader 周期（epoch），可以理解为当前集群所处的年代或者周期，每当有一个新的 Leader 选举出现时，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，以此作为新的周期 ID。高 32 位代表了每代 Leader 的唯一性，低 32 位则代表了每代 Leader 中事务的唯一性。

zab节点的三种状态：

following：服从leader的命令

leading：负责协调事务

election/looking：选举状态

负载均衡策略有哪些

1、轮询法

将请求按顺序轮流地分配到后端服务器上，它均衡地对待后端的每一台服务器，而不关心服务器实际的连接数和当前的系统负载。

2、加权轮询法

不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，让其处理更多的请求；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。

3、随机法

通过系统的随机算法，根据后端服务器的列表大小值来随机选取其中的一台服务器进行访问。由概率统计理论可以得知，随着客户端调用服务端的次数增多，其实际效果越来越接近于平均分配调用量到后端的每一台服务器，也就是轮询的结果。

4、加权随机法

与加权轮询法一样，加权随机法也根据后端机器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。

5、源地址哈希法

源地址哈希的思想是根据获取客户端的IP地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客户端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。

6、最小连接数法

最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它是根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前的请求，尽可能地提高后端服务的利用效率，将负责合理地分流到每一台服务器。

分布式系统的设计目标

可扩展性：通过对服务、存储的扩展，来提高系统的处理能力，通过对多台服务器协同工作，来完成单台服务器无法处理的任务，尤其是高并发或者大数据量的任务。

高可用：单点不影响整体，单点故障指系统中某个组件一旦失效，会让整个系统无法工作

无状态：无状态的服务才能满足部分机器宕机不影响全部，可以随时进行扩展的需求。

可管理：便于运维，出问题能不能及时发现定位

高可靠：同样的请求返回同样的数据；更新能够持久化；数据不会丢失

分布式事务

分布式事务有哪些解决方案

基于XA协议的：两阶段提交和三阶段提交，需要数据库层面支持

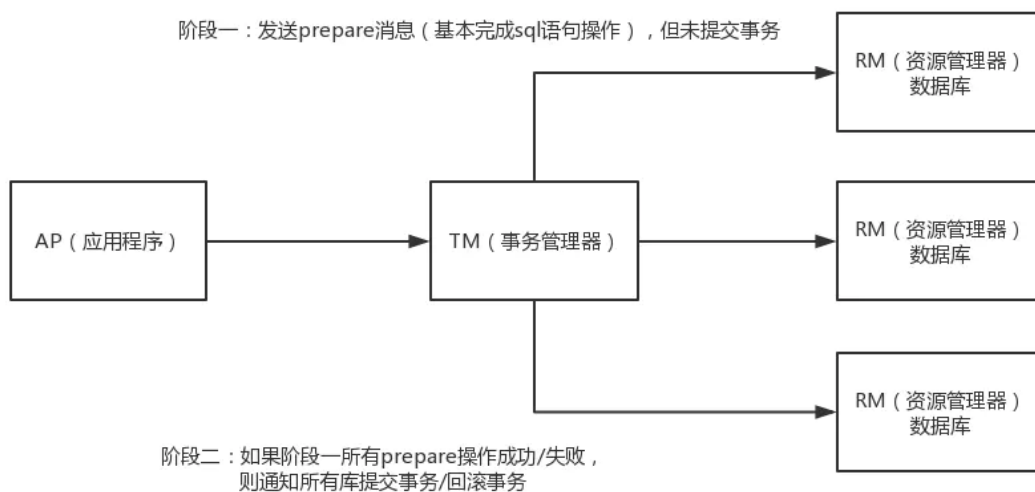
基于事务补偿机制的：TCC，基于业务层面实现

本地消息表：基于本地数据库+mq，维护本地状态（进行中），通过mq调用服务，完成后响应一条消息回调，将状态改成完成。需要配合定时任务扫表，重新发送消息调用服务，需要保证幂等

基于事务消息：mq

对比两阶段，三阶段有哪些改进

两阶段协议：



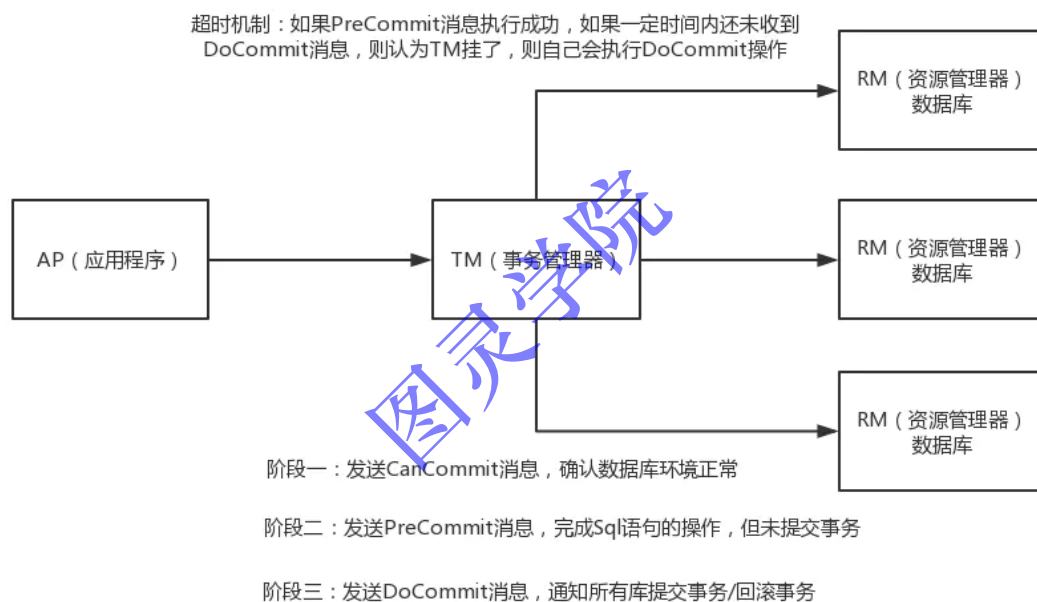
第一阶段（prepare）：每个参与者执行本地事务但不提交，进入 ready 状态，并通知协调者已经准备就绪。

第二阶段（commit）当协调者确认每个参与者都 ready 后，通知参与者进行 commit 操作；如果有参与者 fail，则发送 rollback 命令，各参与者做回滚。

问题：

- 单点故障：一旦事务管理器出现故障，整个系统不可用（参与者都会阻塞住）
- 数据不一致：在阶段二，如果事务管理器只发送了部分 commit 消息，此时网络发生异常，那么只有部分参与者接收到 commit 消息，也就是说只有部分参与者提交了事务，使得系统数据不一致。
- 响应时间较长：参与者和协调者资源都被锁住，提交或者回滚之后才能释放
- 不确定性：当协事务管理器发送 commit 之后，并且此时只有一个参与者收到了 commit，那么当该参与者与事务管理器同时宕机之后，重新选举的事务管理器无法确定该条消息是否提交成功。

三阶段协议：主要是针对两阶段的优化，解决了2PC单点故障的问题，但是性能问题和不一致问题仍然没有根本解决



引入了超时机制解决参与者阻塞的问题，超时后本地提交，2pc只有协调者有超时机制

- 第一阶段：CanCommit阶段，协调者询问事务参与者，是否有能力完成此次事务。
 - 如果都返回yes，则进入第二阶段
 - 有一个返回no或等待响应超时，则中断事务，并向所有参与者发送abort请求
- 第二阶段：PreCommit阶段，此时协调者会向所有的参与者发送PreCommit请求，参与者收到后开始执行事务操作。参与者执行完事务操作后（此时属于未提交事务的状态），就会向协调者反馈“Ack”表示我已经准备好提交了，并等待协调者的下一步指令。
- 第三阶段：DoCommit阶段，在阶段二中如果所有的参与者节点都返回了Ack，那么协调者就会从“预提交状态”转变为“提交状态”。然后向所有的参与者节点发送“doCommit”请求，参与者节点在收到提交请求后就会各自执行事务提交操作，并向协调者节点反馈“Ack”消息，协调者收到所有参与者的Ack消息后完成事务。相反，如果有一个参与者节点未完成PreCommit的反馈或者反馈超时，那么协调者都会向所有的参与者节点发送abort请求，从而中断事务。

简述TCC事务模型

TCC (补偿事务) : Try、Confirm、Cancel

针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作

Try操作做业务检查及资源预留，Confirm做业务确认操作，Cancel实现一个与Try相反的操作既回滚操作。TM首先发起所有的分支事务的try操作，任何一个分支事务的try操作执行失败，TM将会发起所有分支事务的Cancel操作，若try操作全部成功，TM将会发起所有分支事务的Confirm操作，其中Confirm/Cancel操作若执行失败，TM会进行重试。

TCC模型对业务的侵入性较强，改造的难度较大，每个操作都需要有 `try`、`confirm`、`cancel` 三个接口实现

TCC 中会添加事务日志，如果 Confirm 或者 Cancel 阶段出错，则会进行重试，所以这两个阶段需要支持幂等；如果重试失败，则需要人工介入进行恢复和处理等。

分布式服务

如何理解RPC

远程过程调用

RPC要求在调用方中放置被调用的方法的接口。调用方只要调用了这些接口，就相当于调用了被调用方的实际方法，十分易用。于是，调用方可以像调用内部接口一样调用远程的方法，而不用封装参数名和参数值等操作。

包含

1. 动态代理，封装调用细节
2. 序列化与反序列化，数据传输与接收
3. 通信，可以选择七层的http，四层的tcp/udp
4. 异常处理等

首先，调用方调用的是接口，必须得为接口构造一个假的实现。显然，要使用动态代理。这样，调用方的调用就被动态代理接收到了。

第二，动态代理接收到调用后，应该想办法调用远程的实际实现。这包括下面几步：

- 识别具体要调用的远程方法的IP、端口
- 将调用方法的入参进行序列化
- 通过通信将请求发送到远程的方法中

这样，远程的服务就接收到了调用方的请求。它应该：

- 反序列化各个调用参数
- 定位到实际要调用的方法，然后输入参数，执行方法
- 按照调用的路径返回调用的结果

zk的初始化选举和崩溃选举过程

zxId: 事务id, sId: 节点id

先对比zxId, 再对比sId, 先投自己, 选票内容 (zxId, sId) , 遇强改投

投票箱: 每个节点在本地维护自己和其他节点的投票信息, 改投时需要更新信息, 并广播

节点状态:

- LOOKING, 竞选状态。
- FOLLOWING, 随从状态, 同步leader状态, 参与投票。
- OBSERVING, 观察状态, 同步leader状态, 不参与投票。
- LEADING, 领导者状态。

初始化: 没有历史数据, 5个节点为例

- 节点1启动, 此时只有一台服务器启动, 它发出去请求没有任何响应, 所以它的选举状态一直是LOOKING状态
- 节点2启动, 它与节点1进行通信, 互相交换自己的选举结果, 由于两者都没有历史数据, 所以serverId值较大的服务器2胜出, 但是由于没有达到半数以上, 所以服务器1,2还是继续保持LOOKING状态
- 节点3启动, 与1、2节点通信交互数据, 服务器3成为服务器1,2,3中的leader, 此时有三台服务器选举了3, 所以3成为leader
- 节点4启动, 理论上服务器4应该是服务器1,2,3,4中最大的, 但是由于前面已经有半数以上的服务器选举了服务器3, 所以它只能切换为follower
- 节点5启动, 同4一样

崩溃选举:

- 变更状态, leader故障后, follower进入looking状态
- 各节点投票, 先投自己 (zxId, sId) , 再广播投票,
- 接收到投票, 对比zxId和sId, 如果本节点小、则将票改为接收的投票信息, 并记录投票信息, 重新广播。否则本节点大、则可不作处理
- 统计本地投票信息, 超过半数, 则切换为leading状态并广播

zk的数据模型

ZK的数据模型是一种树形结构, 具有一个固定的根节点 (/) , 可以在根节点下创建子节点, 并在子节点下继续创建下一级节点。每一层级用 / 隔开, 且只能用绝对路径 (get /work/task1) 的方式查询ZK节点, 而不能用相对路径

- 持久节点:
将节点创建为持久节点, 该数据节点会一直存储在ZK服务器上, 即使创建该节点的客户端与服务端的会话关闭了, 该节点依然不会被删除, 除非显式调用delete函数进行删除操作。
- 临时节点:
如果将节点创建为临时节点, 那么该节点数据不会一直存储在ZK服务器上。当创建该临时节点的

客户端会话因超时或发生异常而关闭时，该节点也相应地在ZK服务器上被删除。也可以主动调用delete删除。

- 有序节点：

有序节点并不算一种单独种类的节点，而是在持久节点和临时节点的基础上，增加一个节点有序的性质。创建有序节点的时候，ZK服务器会自动使用一个单调递增的数字作为后缀，追加到创建的节点后边。例如一个客户端创建了一个路径为 works/task- 的有序节点，那么 ZooKeeper 将会生成一个序号并追加到该节点的路径后，最后该节点的路径为 works/task-1。

节点内容：一个二进制数组（byte data[]），用来存储节点的数据、ACL访问控制、子节点数据（因为临时节点不允许有子节点，所以其子节点字段为null），记录自身状态信息的stat。

stat + 节点路径可以查看状态信息

czxid：创建节点的事务id

mzxid：最后一次被更新的事务id

pxid：子节点最后一次被修改的事务id

ctime：创建时间

mtime：最后更新时间

version：版本号、表示的是对节点数据内容，子节点信息或ACL信息的修改次数

可以避免并发更新问题，使用之前获取的版本进行CAS操作更新

cversion：子节点版本号

aversion：acl的版本号

ephemeralOwner：创建节点的sessionId，如果是持久节点、值为0

dataLenght：数据内容长度

numChildren：子节点个数

zk的watch机制实现原理

```
new ZooKeeper(String connectionString, int sessionTimeout, Watcher watcher)
这个watcher将作为整个ZooKeeper会话期间的上下文，一直被保存在客户端ZKWatcher的defaultwatcher
```

也可以动态添加watcher：getData(),exists,getChildren。

分布式环境下的观察者模式：通过客户端和服务端分别创建有观察者的信息列表。客户端调用相应接口时，首先将对应的 Watch 事件放到本地的 ZKWatcher 中进行管理。服务端在接收到客户端的请求后根据请求类型判断是否含有 Watch 事件，并将对应事件放到 WatchManager 中进行管理。在事件触发的时候服务端通过节点的路径信息查询相应的 Watch 事件通知给客户端，客户端在接收到

通知后，首先查询本地的 ZKWatchManager 获得对应的 Watch 信息处理回调操作。这种设计不但实现了一个分布式环境下的观察者模式，而且通过将客户端和服务端各自处理 Watch 事件所需要的额外信息分别保存在两端，减少彼此通信的内容。大大提升了服务的处理性能

客户端实现过程：

- 标记该会话是一个带有Watch事件的请求
- 通过 DataWatchRegistration 类来保存 watcher 事件和节点的对应关系
- 客户端向服务器发送请求，将请求封装成一个 Packet 对象，并添加到一个等待发送队列 outgoingQueue 中
- 调用负责处理队列outgoingQueue 的 SendThread 线程类中的 readResponse 方法接收服务端的回调，并在最后执行 finishPacket () 方法将 Watch 注册到 ZKWatchManager，sendThread 通过发送 path 路径和 watcher 为 true，到 server 注册 watch 事件。

ZKWatchManager保存了 Map<String,Set> dataWatchers、Map<String,Set> existsWatchers、Map<String,Set> childrenWatchers 三个集合，客户端会在 dataWatchers 中会添加一个 key 为 path 路径的本地事件

服务端实现过程：

- 解析收到的请求是否带有Watch注册事件，通过 FinalRequestProcessor 类中的 processRequest 函数实现的。当 getDataRequest.getWatch() 值为 True 时，表明该请求需要进行 Watch 监控注册。
- 将对应的Watch事件存储到WatchManager，通过 zks.getZKDatabase().getData 函数实现，WatchManger 该类中有 HashMap<String,HashSet> watchTable，key为path，Watcher 是一个客户端网络连接封装，当节点变化时会通知对应的连接（连接通过心跳保持）

服务端触发过程：

- 调用 WatchManager中的方法触发数据变更事件
- 封装了一个具有会话状态、事件类型、数据节点 3 种属性的 WatchedEvent 对象。之后查询该节点注册的 Watch 事件，如果为空说明该节点没有注册过 Watch 事件。如果存在 Watch 事件则添加到定义的 Wathcers 集合中，并在 WatchManager 管理中删除。最后，通过调用 process 方法向客户端发送通知

客户端回调过程：

- 使用 SendThread.readResponse() 方法来统一处理服务端的相应
- 将收到的字节流反序列化转换成 WatcherEvent 对象。调用 eventThread.queueEvent() 方法将接收到的事件交给 EventThread 线程进行处理
- 从 ZKWatchManager 中查询注册过的客户端 Watch 信息。查询到后，会将其从 ZKWatchManager 的管理中删除。因此客户端的 Watcher 机制是一次性的，触发后就会被删除
- 将查询到的 Watcher 存储到 waitingEvents 队列中，调用 EventThread 类中的 run 方法循环取出在 waitingEvents 队列中等待的 Watcher 事件进行处理

zk分布式锁实现原理

- 上来直接创建一个锁节点下的一个接一个的临时顺序节点
- 如果自己不是第一个节点，就对自己上一个节点加监听器
- 只要上一个节点释放锁，自己就排到前面去了，相当于是一个排队机制。

而且用临时顺序节点，如果某个客户端创建临时顺序节点之后，自己宕机了，zk感知到那个客户端宕机，会自动删除对应的临时顺序节点，相当于自动释放锁，或者是自动取消自己的排队。解决了惊群效应

Zookeeper 的典型应用场景

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会涉及的核心功能，如：

- (1) 数据发布/订阅：配置中心
- (2) 负载均衡：提供服务者列表
- (3) 命名服务：提供服务名到服务地址的映射
- (4) 分布式协调/通知：watch机制和临时节点，获取各节点的任务进度，通过修改节点发出通知
- (5) 集群管理：是否有机器退出和加入，选举 master
- (7) 分布式锁
- (8) 分布式队列

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。

zk的数据同步原理

根据这三个参数的大小对比结果，选择对应的数据同步方式。

- peerLastZxid: Learner服务器 (Follower或observer) 最后处理的zxid。
- minCommittedLog: Leader服务器proposal缓存队列committedLog中的最小的zxid。
- maxCommittedLog: Leader服务器proposal缓存队列committedLog中的最大的zxid。

Zookeeper中数据同步一共有四类，如下。

- DIFF: 直接差异化同步
peerlastZxid介于minCommittedLog和maxCommittedLog之间
- TRUNC+DIFF: 先回滚再差异化同步
当Leader服务器发现某个Learner包含了一条自己没有的事务记录,那么就需要让该Learner进行事务回滚到Leader服务器上存在的
- TRUNC: 仅回滚同步
peerlastZxid大于maxCommittedLog, Leader会要求Learner回滚到ZXID值为maxCommittedLog对应的事务操作
- SNAP: 全量同步
peerLastZxid 小于minCommittedLog

在初始化阶段, Leader服务器会优先初始化以全量同步方式来同步数据

learner先向leader注册, 上报peerLastZxid

zk中一个客户端修改了某个节点的数据，其他客户端能够马上获取到这个最新数据吗？

不能确保任何客户端能够获取（即Read Request）到一样的数据，除非客户端自己要求，方法是客户端在获取数据之前调用函数sync，再调用getData()

ZK客户端A对节点的内容从 v1->v2, 但是ZK客户端B对内容获取，依然得到的是 v1. 这个是实际存在的现象，存在同步时延

请谈谈ZooKeeper对事务性的支持

ZooKeeper对于事务性的支持主要依赖于四个函数，zoo_create_op_init, zoo_delete_op_init, zoo_set_op_init以及zoo_check_op_init。

每一个函数都会在客户端初始化一个operation，客户端程序有义务保留这些operations。当准备好一个事务中的所有操作后，可以使用zoo_multi来提交所有的操作，由zookeeper服务来保证这一系列操作的原子性。也就是说只要其中有一个操作失败了，相当于此次提交的任何一个操作都没有对服务端的数据造成影响。Zoo_multi的返回值是第一个失败操作的状态信号。

简述zk中的观察者机制

```
peerType=observer
server.1:localhost:2181:3181:observer
```

观察者的设计是希望能动态扩展zookeeper集群又不会降低写性能。

如果扩展节点是follower，则写入操作提交时需要同步的节点数会变多，导致写入性能下降，而follower又是参与投票的、也会导致投票成本增加

observer是一种新的节点类型，解决扩展问题的同时，不参与投票、只获取投票结果，同时也可以处理读写请求，写请求转发给leader。负责接收leader同步过来的提交数据，observer的节点故障也不会影响集群的可用性，

跨数据中心部署。把节点分散到多个数据中心可能因为网络的延迟会极大拖慢系统。使用observer的话，更新操作都在一个单独的数据中心来处理，并发送到其他数据中心，让其他数据中心的节点消费数据。

无法完全消除数据中心之间的网络延迟，因为observer需要把更新请求转发到另一个数据中心的leader，并处理同步消息，网络速度极慢的话也会有影响，它的优势是为本地读请求提供快速响应。

zk的会话管理机制

客户端连接zk，有zk分配一个全局唯一的sessionId，客户端需要配置超时时间timeOut并传到zk，zk会据此计算会话下一次超时的时间点，zk根据这个时间点按照分桶策略进行分开存放，zk会给sessionId设置一个isClosing属性、如果检测到超时会将该属性标记为关闭

会话状态：CONNECTING、CONNECTED、RECONNECTING、RECONNECTED、CLOSE

SessionTracker：zk中的会话管理器，负责会话的创建、管理和清理

- sessionsWithTimeout：一个ConcurrentHashMap，用来管理会话的超时时间
- sessionsById：HashMap，维护sessionId到session的映射
- sessionsSets：HashMap，会话超时后进行归档，便于恢复和管理

$\text{ExpirationTime} = \text{CurrentTime} + \text{SessionTimeout}$

SessionTracker根据ExpirationTime 将session进行分桶管理，同时按照一定的时间间隔进行定期检查，

客户端读写请求都可以将session的超时时间重置，SessionTracker会将session进行分桶迁移，如果没有读写请求，客户需要发送ping心跳链接，否则session超时会被清除

会话清理：

- 标记isClosing为关闭，此时该会话有新的请求也无法处理
- 发起会话关闭请求，同步到整个集群，使用提交的方式
- 收集需要清理的临时节点，先获取内存数据库中会话对应的临时节点集合，如果此时有删除节点请求到达，将请求对应的节点路径从集合中移除，避免重复删除，如果有创建节点请求到达、则将

请求中的路径添加到集合中

- 添加删除事务变更，将节点删除事务添加到outstandingChanges中，触发watch
- 删除临时节点
- 移除会话
- 关闭连接

连接断开后客户端可以重连zk，如果该session未过期，session重新变为CONNECTED

如果时间超过 `sessionTimeout`，服务器则会进行会话的清理工作，如果此时ZK客户端才恢复连接，则会收到State为Expired的 `WatchedEvent`，并断开与服务器的连接。

重连：断开后更换服务器链接，RECONNECTING状态，会将会话迁移到新连接的服务器上

当一个客户端发一个心跳请求个服务端，但是网络延时，导致服务端没有收到，过一会后，客户端连接上了另一个新的服务端，在这之后，之前的心跳被旧的服务端收到了，这时候旧的服务端会被提醒，当前session已经被转移了，然后旧的服务端会关闭这个连接。客户端一般不会感知到这个异常，因为旧连接一般都会被关闭。但是还有一个特殊情况，两个客户端同时使用保存这的session id+密码来重新连接服务端，第一个连接成功，紧着第二个又连接成功，这会导致第一个连接被关闭，然后就是这两个客户端无限重连了

zk和eureka的对比

- 1、zk保证的是CP，即分区容错和一致性，eureka保证的是AP、即可用性和分区容错
- 2、zk的master如果挂掉，需要进行选举，此时zk无法使用。

eureka的各个节点是平等的。只要有一台是正常的就能提供服务，

eureka还有一种自我保护机制，如果15分钟内超过85%的节点没有正常心跳会开启保护机制。eureka不再从注册列表移除长时间没有心跳的服务、eureka仍能接受新的服务注册和查询，只不过不会同步到其他节点。网络稳定后再同步到其他节点。同时eureka的客户端也会缓存服务信息。eureka对于服务注册发现来说是非常好的选择

Dubbo是什么？能做什么？

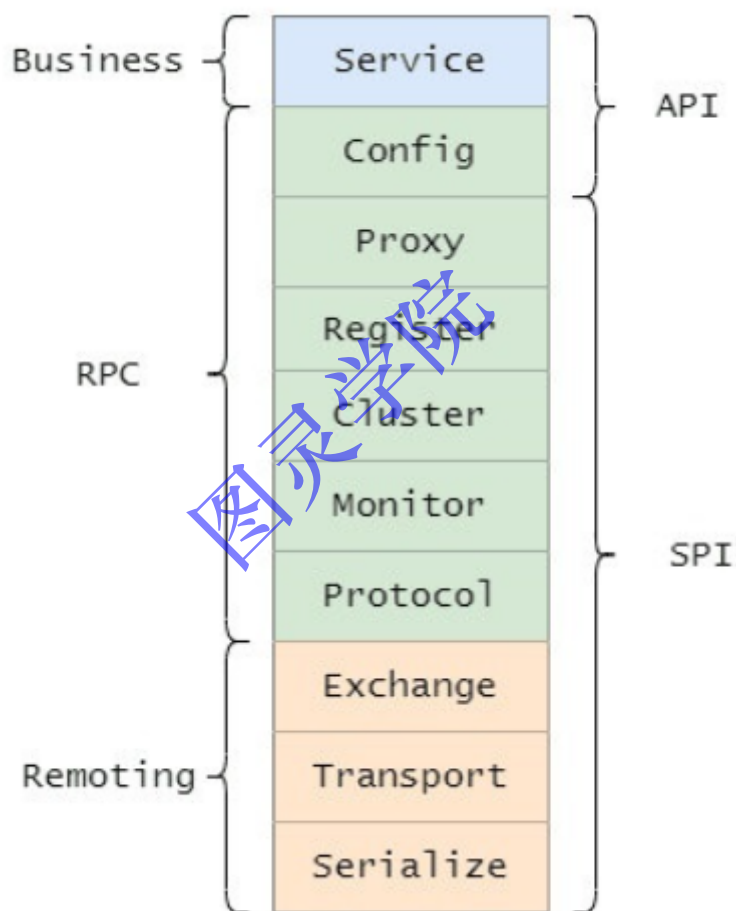
Dubbo是阿里巴巴开源的基于Java的高性能RPC分布式服务框架，现已成为Apache基金会孵化项目。致力于提供高性能和透明化的RPC远程服务调用方案，以及SOA服务治理方案。

简单的说，dubbo就是个服务框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有dubbo这样的分布式服务框架的需求，本质上是个远程服务调用的分布式框架

其核心部分包含：

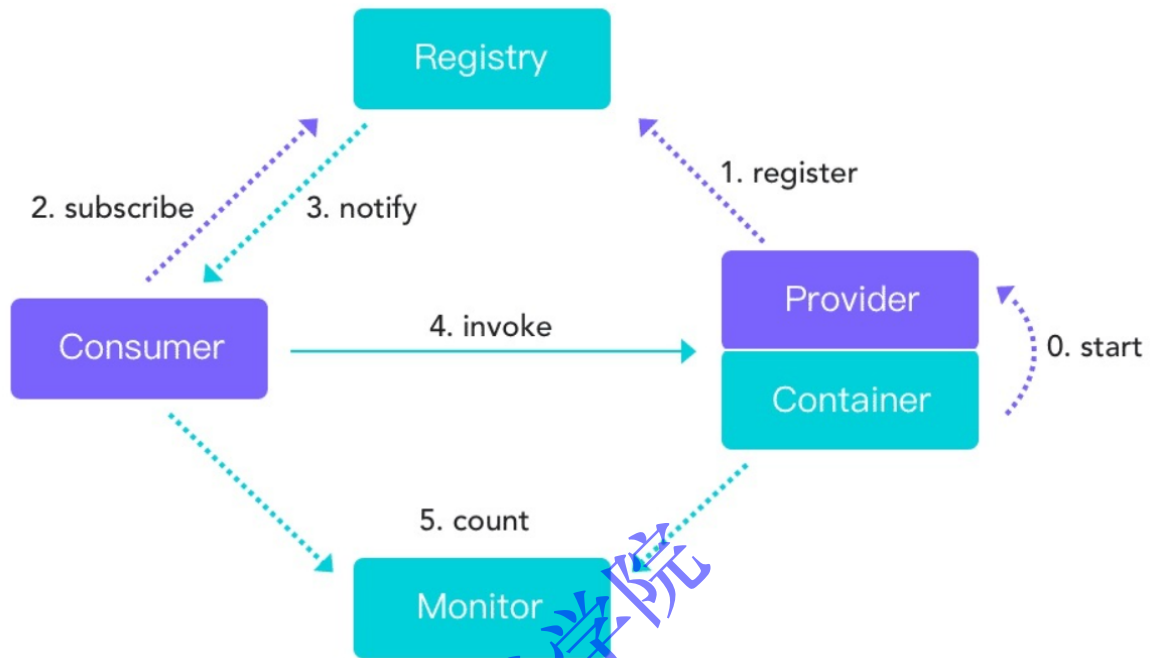
1. 远程通讯：提供对多种基于长连接的NIO框架抽象封装，包括多种线程模型，序列化，以及“请求-响应”模式的信息交换方式，透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。
2. 集群容错：提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持，可在内网替代F5等硬件负载均衡器，降低成本，减少单点。
3. 自动发现：基于注册中心目录服务，服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。

简述dubbo的分层设计



- Service，业务层，就是咱们开发的业务逻辑层。
- Config，配置层，主要围绕 ServiceConfig 和 ReferenceConfig，初始化配置信息。
- Proxy，代理层，服务提供者还是消费者都会生成一个代理类，使得服务接口透明化，代理层做远程调用和返回结果。
- Register，注册层，封装了服务注册和发现。
- Cluster，路由和集群容错层，负责选取具体调用的节点，处理特殊的调用要求和负责远程调用失败的容错措施。
- Monitor，监控层，负责监控统计调用时间和次数。
- Portocol，远程调用层，主要是封装 RPC 调用，主要负责管理 Invoker。
- Exchange，信息交换层，用来封装请求响应模型，同步转异步。
- Transport，网络传输层，抽象了网络传输的统一接口，Netty、Mina 等。
- Serialize，序列化层，将数据序列化成二进制流，以及反序列化。

能说下Dubbo的工作流程吗



1. Start: 启动Spring容器时,自动启动Dubbo的Provider
2. Register: Dubbo的Provider在启动后会去注册中心注册内容.注册的内容包括: IP、端口、接口列表(接口类、方法)、版本、Provider的协议.
3. Subscribe: 订阅.当Consumer启动时,自动去Registry获取到所已注册的服务的信息.
4. Notify: 通知.当Provider的信息发生变化时,自动由Registry向Consumer推送通知.
5. Invoke: Consumer 调用Provider中方法
 - 5.1 同步请求.消耗一定性能.但是必须是同步请求,因为需要接收调用方法后的结果
6. Count: 次数, 每隔2分钟,Provider和Consumer自动向Monitor发送访问次数.Monitor进行统计.

dubbo服务暴露过程

Dubbo 采用 URL 的方式来作为约定的参数类型。

```
protocol://username:password@host:port/path?key=value&key=value
```

protocol: 指的是 dubbo 中的各种协议, 如: dubbo thrift http

username/password: 用户名/密码

host/port: 主机/端口

path: 接口的名称

parameters: 参数键值对

ServiceBean实现了ApplicationListener，监听ContextRefreshedEvent时间，在 Spring IOC 容器刷新完成后调用 onApplicationEvent方法，服务暴露的启动点。根据配置得到 URL，再利用 Dubbo SPI 机制根据 URL 的参数选择对应的实现类，实现扩展。

通过 javassist 动态封装服务实现类，统一暴露出 Invoker 使得调用方便、屏蔽底层实现细节，然后封装成 exporter 存储起来，等待消费者的调用，并且会将 URL 注册到注册中心，使得消费者可以获取服务提供者的信息。

一个服务如果有多个协议那么就都需要暴露，比如同时支持 dubbo 协议和 hessian 协议，那么需要将这个服务用两种协议分别向多个注册中心（如果有多个的话）暴露注册。

- 1、检测配置，如果有些配置空的话会默认创建，并且组装成 URL。
- 2、根据URL进行服务暴露、创建代理类invoker、根据URL得知具体的协议，根据Dubbo SPI选取实现类实现exporter。
- 3、如果只是本地暴露，将exporter 存入ServiceConfig的缓存
- 4、远程暴露，先通过 registry 协议找到 RegistryProtocol 进行 export，将URL中export=dubbo://...先转换成 exporter，然后获取注册中心的相关配置，如果需要注册则向注册中心注册，并且在 ProviderConsumerRegTable 这个表格中记录服务提供者，其实就是往一个 ConcurrentHashMap 中将塞入 invoker，key 就是服务接口全限定名，value 是一个 set，set 里面会存包装过的 invoker，根据URL上 Dubbo 协议暴露出 exporter，打开 Server调用NettyServer 来监听服务

图灵学院

dubbo服务引入过程

饿汉式是通过调用 ReferenceBean的 afterPropertiesSet方法时引入服务。

懒汉式是只有当这个服务被注入到其他类中时启动引入流程，也就是说用到了才会开始服务引入。默认使用懒汉式，如果需要使用饿汉式，可通过配置dubbo:reference 的 init 属性开启。

ReferenceBean实现了FactoryBean接口，当对任意服务 Interface 进行自动注入或者 getBean 获取时，就会触发 getObject()函数的服务引用过程。

- 本地引入走 injvm 协议，到服务暴露的缓存中取exporter。
- 直连远程引入服务，测试的情况下用，不需要启动注册中心，由 Consumer 直接配置写死 Provider 的地址，然后直连即可。
- 注册中心引入远程服务，Consumer 通过注册中心得知 Provider 的相关信息，然后进行服务的引入

获取注册中心实例，向注册中心注册自身，并订阅providers、configurators、routers节点，触发DubboInvoker的生成，cluster将多个服务调用者进行封装，返回一个invoker，

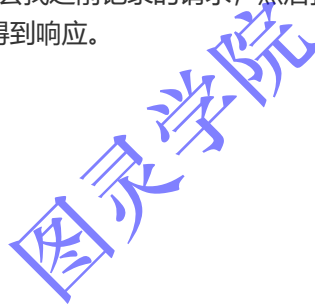
通过配置构建一个map，然后利用map来构建URL，再通过URL上的协议利用自适应扩展机制调用对应的protocol.refer得到相应的invoker。然后再构建代理，封装invoker返回服务引用，之后Consumer调用这个代理类

dubbo服务调用过程

调用某个接口的方法会调用之前生成的代理类，然后会从cluster中经过路由的过滤、负载均衡机制选择一个invoker发起远程调用，此时会记录此请求和请求的ID等待服务端的响应。

服务端接受请求之后会通过参数找到之前暴露存储的map，得到相应的exporter，然后最终调用真正的实现类，再组装好结果返回，这个响应会带上之前请求的ID。

消费者收到这个响应之后会通过ID去找之前记录的请求，然后找到请求之后将响应塞到对应的Future中，唤醒等待的线程，最后消费者得到响应。



Dubbo支持的注册中心有哪些？

Zookeeper(官方推荐)

- 优点：支持分布式
- 缺点：受限于Zookeeper的特性

Multicast：组播协议允许将一台主机发送的数据通过网络路由器和交换机复制到多个加入此组播的主机，是一种一对多的通讯方式。每一台服务提供方和服务消费方都可以看作是注册中心的一部分

- 优点：去中心化,不需要单独安装软件.
- 缺点：Provider和Consumer和Registry不能跨机房(路由)

不需要启动任何中心节点，只要广播地址一样，就可以互相发现，组播受网络结构限制，只适合小规模应用或开发阶段使用。

Redis

- 优点:支持集群,性能高
- 缺点:要求服务器时间同步.否则可能出现集群失败问题

Simple

- 优点: 标准RPC服务.没有兼容问题
- 缺点: 不支持集群

dubbo集群容错策略有哪些

Failover Cluster失败自动切换：dubbo的默认容错方案，当调用失败时自动切换到其他可用的节点，具体的重试次数和间隔时间可通过引用服务的时候配置，默认重试次数为1也就是只调用一次。

Failback Cluster失败自动恢复：在调用失败，记录日志和调用信息，然后返回空结果给consumer，并且通过定时任务每隔5秒对失败的调用进行重试

Failfast Cluster快速失败：只会调用一次，失败后立刻抛出异常

Failsafe Cluster失败安全：调用出现异常，记录日志不抛出，返回空结果

Forking Cluster并行调用多个服务提供者：通过线程池创建多个线程，并发调用多个provider，结果保存到阻塞队列，只要有一个provider成功返回了结果，就会立刻返回结果

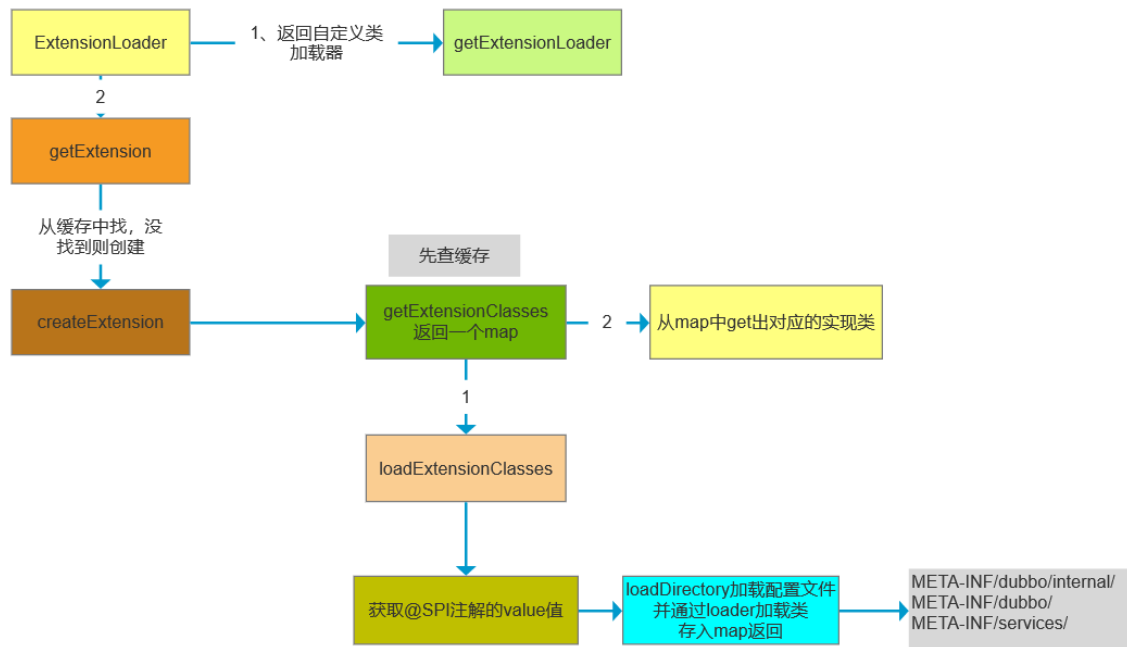
Broadcast Cluster广播模式：逐个调用每个provider，如果其中一台报错，在循环调用结束后，抛出异常。

图灵学院

dubbo的SPI机制

spi: service provider interface 服务发现机制

通过接口全限定名找到指定目录下对应的文件，获取具体的实现类然后加载即可，做到了灵活的替换具体的实现类。



```

@SPI("dubbo")
public interface Protocol {

    int getDefaultPort();

    @Adaptive
    <T> Exporter<T> export(Invoker<T> invoker) throws RpcException;

    @Adaptive
    <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;

    void destroy();
}

```

Dubbo的jar包，在 /META_INF/dubbo/internal/com.alibaba.dubbo.rpc.Protocol 文件中

```

dubbo=com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol
http=com.alibaba.dubbo.rpc.protocol.http.HttpProtocol
hessian=com.alibaba.dubbo.rpc.protocol.hessian.HessianProtocol

```

Protocol接口@Adaptive注解，在运行时会针对Protocol生成代理类，该代理类的方法中会有代理代码，代理代码会在运行时动态根据url中的protocol来获取key(默认是dubbo)，也可以自己指定，如果指定了别的key，那么就会获取别的实现类的实例，通过这个url中的参数不同，就可以控制动态使用不同的组件实现类

为什么 Dubbo 不用 JDK 的 SPI，而是要自己实现？

java spi缺点：

- 1、需要遍历所有实现并实例化，假设一个实现类初始化过程比较消耗资源且耗时，但是你的代码里面又用不上它，这就产生了资源的浪费。也无法准确引用
- 2、没有使用缓存每次load都需要重新加载

Dubbo SPI：

- 1、给每个实现类配了个名字，通过名字去文件里面找到对应的实现类全限定名然后加载实例化，按需加载。
- 2、增加了缓存存储实例，提高读取性能。
- 3、提供了对IOC和AOP等高级功能的支持，以实现更多类型的扩展。

dubbo和springcloud对比

- 1、dubbo由于是二进制的传输，占用带宽会更少
- 2、springCloud是http协议传输，带宽会比较多，同时使用http协议一般会使用JSON报文，消耗会更大
- 3、dubbo的开发难度较大，原因是dubbo的jar包依赖问题很多大型工程无法解决
- 4、springcloud的接口协议约定比较自由且松散，需要有强有力的行政措施来限制接口无序升级
- 5、dubbo只是springcloud的一个子集，解决的是分布式中的服务调用问题，而springcloud提供了全套的解决方案

dubbo中Zk集群挂掉，发布者和订阅者还能通信么？

可以。

因为当启动dubbo容器时，消费者会去zookeeper拉取注册的生产者地址列表，并将其缓存在本地。每次发起调用时，都会按照本地的地址列表，以负载均衡的策略去进行调用。但是zookeeper挂掉则后续新的生产者无法被消费者发现。

- 注册中心对等集群，任意一台宕掉后，会自动切换到另一台
- 注册中心全部宕掉，服务提供者和消费者仍可以通过本地缓存通讯
- 服务提供者无状态，任一台宕机后，不影响使用
- 服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复

zk实际是如何存储生产者和消费者信息

服务提供者启动时: 向 `/dubbo/com.foo.BarService/providers` 目录下写入自己的 URL 地址

服务消费者启动时: 订阅 `/dubbo/com.foo.BarService/providers` 目录下的提供者 URL 地址。并向 `/dubbo/com.foo.BarService/consumers` 目录下写入自己的 URL 地址

监控中心启动时: 订阅 `/dubbo/com.foo.BarService` 目录下的所有提供者和消费者 URL 地址。

dubbo支持的协议有哪些

Dubbo协议(官方推荐协议)

- 采用NIO复用单一长连接, 并使用线程池并发处理请求, 减少握手和加大并发效率, 性能较好 (推荐使用)
- 大文件上传时,可能出现问题(不使用Dubbo文件上传)

RMI(Remote Method Invocation)协议

- JDK自带的能力。可与原生RMI互操作, 基于TCP协议、短连接

Hessian协议

- 可与原生Hessian互操作, 基于HTTP协议, 对于数据包比较大的情况比较友好
- 需Hessian.jar支持, Http短连接的开销大, 它的参数和返回值都需要实现Serializable接口

http协议

Webservice: 基于CXF的Frontend-simple和Transports-http实现; 基于WebService的远程调用协议。

- 序列化: SOAP文本序列化
- 适用场景: 系统集成, 跨语言调用。

Thrift: Thrift是Facebook捐给Apache的一个RPC框架,

- 语言中立
- 平台中立

分布式存储

如何实现分库分表

将原本存储于单个数据库上的数据拆分到多个数据库，把原来存储在单张数据表的数据拆分到多张数据表中，实现数据切分，从而提升数据库操作性能。分库分表的实现可以分为两种方式：垂直切分和水平切分。

水平：将数据分散到多张表，涉及分区键，

- 分库：每个库结构一样，数据不一样，没有交集。库多了可以缓解io和cpu压力
- 分表：每个表结构一样，数据不一样，没有交集。表数量减少可以提高sql执行效率、减轻cpu压力

垂直：将字段拆分为多张表，需要一定的重构

- 分库：每个库结构、数据都不一样，所有库的并集为全量数据
- 分表：每个表结构、数据不一样，至少有一列交集，用于关联数据，所有表的并集为全量数据

存储拆分后如何解决唯一主键问题

UUID：简单、性能好，没有顺序，没有业务含义，存在泄漏mac地址的风险

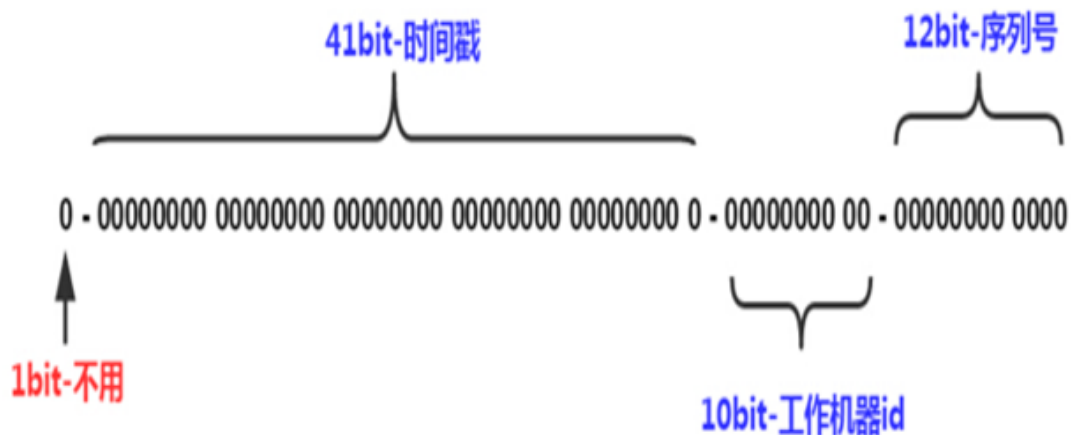
数据库主键：实现简单，单调递增，具有一定的业务可读性，强依赖db、存在性能瓶颈，存在暴露业务信息的风险

redis，mongodb，zk等中间件：增加了系统的复杂度和稳定性

雪花算法

雪花算法原理

snowflake-64bit



第一位符号位固定为0，41位时间戳，10位workId，12位序列号，位数可以有不同实现
优点：

- 每个毫秒值包含的ID值很多，不够可以变动位数来增加，性能佳（依赖workId的实现）。
- 时间戳值在高位，中间是固定的机器码，自增的序列在低位，整个ID是趋势递增的。
- 能够根据业务场景数据库节点布置灵活调整bit位划分，灵活度高。

缺点：

- 强依赖于机器时钟，如果时钟回拨，会导致重复的ID生成，所以一般基于此的算法发现时钟回拨，都会抛异常处理，阻止ID生成，这可能导致服务不可用。

如何解决不使用分区键的查询问题

- 映射：将查询条件的字段与分区键进行映射，建一张单独的表维护(使用覆盖索引)或者在缓存中维护
- 基因法：分区键的后x个bit位由查询字段进行hash后占用，分区键直接取x个bit位获取分区，查询字段进行hash获取分区，适合非分区键查询字段只有一个的情况
- 冗余：查询字段冗余存储

session的分布式方案

- 1、采用无状态服务，抛弃session

2、存入cookie（有安全风险）

3、服务器之间进行 Session 同步，这样可以保证每个服务器上都有全部的 Session 信息，不过当服务器数量比较多的时候，同步是会有延迟甚至同步失败；

4、IP 绑定策略

使用 Nginx（或其他复杂均衡软硬件）中的 IP 绑定策略，同一个 IP 只能在指定的同一个机器访问，但是这样做失去了负载均衡的意义，当挂掉一台服务器的时候，会影响一批用户的使用，风险很大；

5、使用 Redis 存储

把 Session 放到 Redis 中存储，虽然架构上变得复杂，并且需要多访问一次 Redis，但是这种方案带来的好处也是很大的：

- 实现了 Session 共享；
- 可以水平扩展（增加 Redis 服务器）；
- 服务器重启 Session 不丢失（不过也要注意 Session 在 Redis 中的刷新/失效机制）；
- 不仅可以跨服务器 Session 共享，甚至可以跨平台（例如网页端和 APP 端）。

分布式缓存

如何避免缓存穿透、缓存击穿、缓存雪崩？

缓存雪崩是指缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案：

- 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
- 给每一个缓存数据增加相应的缓存标记，记录缓存是否失效，如果缓存标记失效，则更新数据缓存。
- 缓存预热
- 互斥锁

缓存穿透是指缓存和数据库中都没有的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案：

- 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；
- 从缓存取不到的数据，在数据库中也未取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击
- 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力

缓存击穿是指缓存中没有但数据库中有数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。和缓存雪崩不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案

- 设置热点数据永远不过期。
- 加互斥锁

分布式系统中常用的缓存方案有哪些

客户端缓存：页面和浏览器缓存，APP缓存，H5缓存，localStorage 和 sessionStorage

CDN缓存：内容存储：数据的缓存，内容分发：负载均衡

nginx缓存：静态资源

服务端缓存：本地缓存，外部缓存

数据库缓存：持久层缓存（mybatis，hibernate多级缓存），mysql查询缓存

操作系统缓存：Page Cache、Buffer Cache

如何保证数据库与缓存的一致性？

由于缓存和数据库是分开的，无法做到原子性的同时进行数据修改，可能出现缓存更新失败，或者数据库更新失败的情况，这时候会出现数据不一致，影响前端业务

- 先更新数据库，再更新缓存。缓存可能更新失败，读到老数据
- 先删缓存，再更新数据库。并发时，读操作可能还是会将旧数据读回缓存
- 先更新数据库，再删缓存。也存在缓存删除失败的可能

最经典的缓存+数据库读写的模式，Cache Aside Pattern。

读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。

更新的时候，先更新数据库，然后再删除缓存。

为什么是删除而不是更新？

删除更加轻量，延迟加载的一种实现，更新可能涉及多个表、比较耗时

延时双删：先删除缓存，再更新数据库，休眠1s、再次删除缓存。写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据，并发还是可能读到旧值覆盖缓存

终极方案：

将访问操作串行化

1. 先删缓存，将更新数据库的操作放进有序队列中
2. 从缓存查不到的查询操作，都进入有序队列

会面临的问题：

1. 读请求积压，大量超时，导致数据库的压力：限流、熔断
2. 如何避免大量请求积压：将队列水平拆分，提高并行度。
3. 保证相同请求路由正确。

缓存过期都有哪些策略？

定时过期：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量

惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，但是很消耗内存、许多的过期数据都还存在于内存中。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。

定期过期：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key（是随机的），并清除其中已过期的key。该策略是定时过期和惰性过期的折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

分桶策略：定期过期的优化，将过期时间点相近的key放在一起，按时间扫描分桶。

常见的缓存淘汰算法

FIFO（First In First Out，先进先出），根据缓存被存储的时间，离当前最远的数据优先被淘汰；

LRU（Least Recently Used，最近最少使用），根据最近被使用的时间，离当前最远的数据优先被淘汰；

LFU（Least Frequently Used，最不经常使用），在一段时间内，缓存数据被使用次数最少的会被淘汰。

布隆过滤器原理，优缺点

位图：int[10]，每个int类型的整数是4*8=32个bit，则int[10]一共有320 bit，每个bit非0即1，初始化时都是0

添加数据时，将数据进行hash得到hash值，对应到bit位，将该bit改为1，hash函数可以定义多个，则一个数据添加会将多个（hash函数个数）bit改为1，多个hash函数的目的是减少hash碰撞的概率

查询数据：hash函数计算得到hash值，对应到bit中，如果有一个为0，则说明数据不在bit中，如果都为1，则该数据可能在bit中

优点：

- 占用内存小
- 增加和查询元素的时间复杂度为： $O(K)$ ，(K为哈希函数的个数，一般比较小)，与数据量大小无关
- 哈希函数相互之间没有关系，方便硬件并行运算
- 布隆过滤器不需要存储元素本身，在某些对保密要求比较严格的场合有很大优势
- 数据量很大时，布隆过滤器可以表示全集
- 使用同一组散列函数的布隆过滤器可以进行交、并、差运算

缺点：

- 误判率，即存在假阳性(False Position)，不能准确判断元素是否在集合中
- 不能获取元素本身
- 一般情况下不能从布隆过滤器中删除元素

图灵学院

分布式缓存寻址算法

- hash算法：根据key进行hash函数运算、结果对分片数取模，确定分片
适合固定分片数的场景
扩展分片或者减少分片时，所有数据都需要重新计算分片、存储
- 一致性hash：将整个hash值得区间组织成一个闭合的圆环，计算每台服务器的hash值、映射到圆环中。使用相同的hash算法计算数据的hash值，映射到圆环，顺时针寻找，找到的第一个服务器就是数据存储的服务器。
新增及减少节点时只会影响节点到他逆时针最近的一个服务器之间的值
存在hash环倾斜的问题，即服务器分布不均匀，可以通过虚拟节点解决
- hash slot：将数据与服务器隔离开，数据与slot映射，slot与服务器映射，数据进行hash决定存放的slot
新增及删除节点时，将slot进行迁移即可

redis的持久化机制

RDB: Redis DataBase 将某一个时刻的内存快照 (Snapshot) , 以二进制的方式写入磁盘。

手动触发:

- save命令, 使 Redis 处于阻塞状态, 直到 RDB 持久化完成, 才会响应其他客户端发来的命令, 所以在生产环境一定要慎用
- bgsave命令, fork出一个子进程执行持久化, 主进程只在fork过程中有短暂的阻塞, 子进程创建之后, 主进程就可以响应客户端请求了

自动触发:

- save m n: 在 m 秒内, 如果有 n 个键发生改变, 则自动触发持久化, 通过bgsave执行, 如果设置多个、只要满足其一就会触发, 配置文件有默认配置(可以注释掉)
- flushall: 用于清空redis所有的数据库, flushdb清空当前redis所在库数据(默认是0号数据库), 会清空RDB文件, 同时也会生成dump.rdb、内容为空
- 主从同步: 全量同步时会自动触发bgsave命令, 生成rdb发送给从节点

优点:

- 1、整个Redis数据库将只包含一个文件 dump.rdb, 方便持久化。
- 2、容灾性好, 方便备份。
- 3、性能最大化, fork 子进程来完成写操作, 让主进程继续处理命令, 所以是 IO 最大化。使用单独子进程来进行持久化, 主进程不会进行任何 IO 操作, 保证了redis的高性能
- 4.相对于数据集大时, 比 AOF 的启动效率更高。

缺点:

- 1、数据安全性低。RDB 是间隔一段时间进行持久化, 如果持久化之间 redis 发生故障, 会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)
- 2、由于RDB是通过fork子进程来协助完成数据持久化工作的, 因此, 如果当数据集较大时, 可能会导致整个服务器停止服务几百毫秒, 甚至是1秒钟。会占用cpu

AOF: Append Only File 以日志的形式记录服务器所处理的每一个写、删除操作, 查询操作不会记录, 以文本的方式记录, 可以打开文件看到详细的操作记录, 调操作系统命令进程刷盘

- 1、所有的写命令会追加到 AOF 缓冲中。
- 2、AOF 缓冲区根据对应的策略向硬盘进行同步操作。
- 3、随着 AOF 文件越来越大, 需要定期对 AOF 文件进行重写, 达到压缩的目的。
- 4、当 Redis 重启时, 可以加载 AOF 文件进行数据恢复。

同步策略:

- 每秒同步: 异步完成, 效率非常高, 一旦系统出现宕机现象, 那么这一秒钟之内修改的数据将会丢失
- 每修改同步: 同步持久化, 每次发生的数据变化都会被立即记录到磁盘中, 最多丢一条
- 不同步: 由操作系统控制, 可能丢失较多数据

• 优点:

- 1、数据安全

2、通过 append 模式写文件，即使中途服务器宕机也不会破坏已经存在的内容，可以通过 redis-check-aof 工具解决数据一致性问题。

3、AOF 机制的 rewrite 模式。定期对AOF文件进行重写，以达到压缩的目的

- 缺点：

1、AOF 文件比 RDB 文件大，且恢复速度慢。

2、数据集大的时候，比 rdb 启动效率低。

3、运行效率没有RDB高

AOF文件比RDB更新频率高，优先使用AOF还原数据。

AOF比RDB更安全也更大

RDB性能比AOF好

如果两个都配了优先加载AOF

Redis单线程为什么这么快

Redis基于Reactor模式开发了网络事件处理器、文件事件处理器 file event handler。它是单线程的，所以 Redis 才叫做单线程的模型，它采用IO多路复用机制来同时监听多个Socket，根据Socket上的事件类型来选择对应的事件处理器来处理这个事件。可以实现高性能的网络通信模型，又可以跟内部其他单线程的模块进行对接，保证了 Redis 内部的线程模型的简单性。

文件事件处理器的结构包含4个部分：多个Socket、IO多路复用程序、文件事件分派器以及事件处理器（命令请求处理器、命令回复处理器、连接应答处理器等）。

多个 Socket 可能并发的产生不同的事件，IO多路复用程序会监听多个 Socket，会将 Socket 放入一个队列中排队，每次从队列中有序、同步取出一个 Socket 给事件分派器，事件分派器把 Socket 给对应的事件处理器。

然后一个 Socket 的事件处理完之后，IO多路复用程序才会将队列中的下一个 Socket 给事件分派器。文件事件分派器会根据每个 Socket 当前产生的事件，来选择对应的事件处理器来处理。

1、Redis启动初始化时，将连接应答处理器跟AE_READABLE事件关联。

2、若一个客户端发起连接，会产生一个AE_READABLE事件，然后由连接应答处理器负责和客户端建立连接，创建客户端对应的socket，同时将这个socket的AE_READABLE事件和命令请求处理器关联，使得客户端可以向主服务器发送命令请求。

3、当客户端向Redis发请求时（不管读还是写请求），客户端socket都会产生一个AE_READABLE事件，触发命令请求处理器。处理器读取客户端的命令内容，然后传给相关程序执行。

4、当Redis服务器准备好给客户端的响应数据后，会将socket的AE_WRITABLE事件和命令回复处理器关联，当客户端准备好读取响应数据时，会在socket产生一个AE_WRITABLE事件，由对应命令回复处理器处理，即将准备好的响应数据写入socket，供客户端读取。

5、命令回复处理器全部写完到 socket 后，就会删除该socket的AE_WRITABLE事件和命令回复处理器的映射。

单线程快的原因：

- 1) 纯内存操作
- 2) 核心是基于非阻塞的IO多路复用机制
- 3) 单线程反而避免了多线程的频繁上下文切换带来的性能问题

说一下你知道的redis高可用方案

主从

哨兵模式：

sentinel，哨兵是 redis 集群中非常重要的一个组件，主要有以下功能：

- 集群监控：负责监控 redis master 和 slave 进程是否正常工作。
- 消息通知：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- 故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。
- 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。

哨兵用于实现 redis 集群的高可用，本身也是分布式的，作为一个哨兵集群去运行，互相协同工作。

- 故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举
- 即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的
- 哨兵通常需要 3 个实例，来保证自己的健壮性。
- 哨兵 + redis 主从的部署架构，是不保证数据零丢失的，只能保证 redis 集群的高可用性。
- 对于哨兵 + redis 主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。

Redis Cluster是一种服务端Sharding技术，3.0版本开始正式提供。采用slot(槽)的概念，一共分成16384个槽。将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行

方案说明

- 通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值)区间的数据，默认分配了16384个槽位
- 每份数据分片会存储在多个互为主从的多节点上

- 数据写入先写主节点，再同步到从节点(支持配置为阻塞同步)
- 同一分片多个节点间的数据不保持强一致性
- 读取数据时，当客户端操作的key没有分配在该节点上时，redis会返回转向指令，指向正确的节点
- 扩容时需要把旧节点的数据迁移一部分到新节点

在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 6379，另外一个就是 加1w 的端口号，比如 16379。

16379 端口号是用来进行节点间通信的，也就是 cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。cluster bus 用了另外一种二进制的协议，gossip 协议，用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间。

优点

- 无中心架构，支持动态扩容，对业务透明
- 具备Sentinel的监控和自动Failover(故障转移)能力
- 客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可
- 高性能，客户端直连redis服务，免去了proxy代理的损耗

缺点

- 运维也很复杂，数据迁移需要人工干预
- 只能使用0号数据库
- 不支持批量操作(pipeline管道操作)
- 分布式逻辑和存储模块耦合等

Redis Sharding是Redis Cluster出来之前，业界普遍使用的多Redis实例集群方法。其主要思想是采用哈希算法将Redis数据的key进行散列，通过hash函数，特定的key会映射到特定的Redis节点上。Java redis客户端驱动jedis，支持Redis Sharding功能，即ShardedJedis以及结合缓存池的ShardedJedisPool

优点

优势在于非常简单，服务端的Redis实例彼此独立，相互无关联，每个Redis实例像单服务器一样运行，非常容易线性扩展，系统的灵活性很强

缺点

由于sharding处理放到客户端，规模进一步扩大时给运维带来挑战。

客户端sharding不支持动态增删节点。服务端Redis实例群拓扑结构有变化时，每个客户端都需要更新调整。连接不能共享，当应用规模增大时，资源浪费制约优化

redis 主从同步机制

1、从节点执行slaveof masterIp port，保存主节点信息

- 2、从节点中的定时任务发现主节点信息，建立和主节点的socket连接
- 3、从节点发送信号，主节点返回，两边能互相通信
- 4、连接建立后，主节点将所有数据发送给从节点（数据同步）
- 5、主节点把当前的数据同步给从节点后，便完成了复制过程。接下来，主节点就会持续的把写命令发送给从节点，保证主从数据一致性。

runId: 每个redis节点启动都会生成唯一的uuid，每次redis重启后，runId都会发生变化。

offset: 主从节点各自维护自己的复制偏移量offset，当主节点有写入命令时， $offset = offset + \text{命令的字节长度}$ 。从节点在收到主节点发送的命令后，也会增加自己的offset，并把自己的offset发送给主节点。主节点同时保存自己的offset和从节点的offset，通过对比offset来判断主从节点数据是否一致。

repl_backlog_size: 保存在主节点上的一个固定长度的先进先出队列，默认大小是1MB。

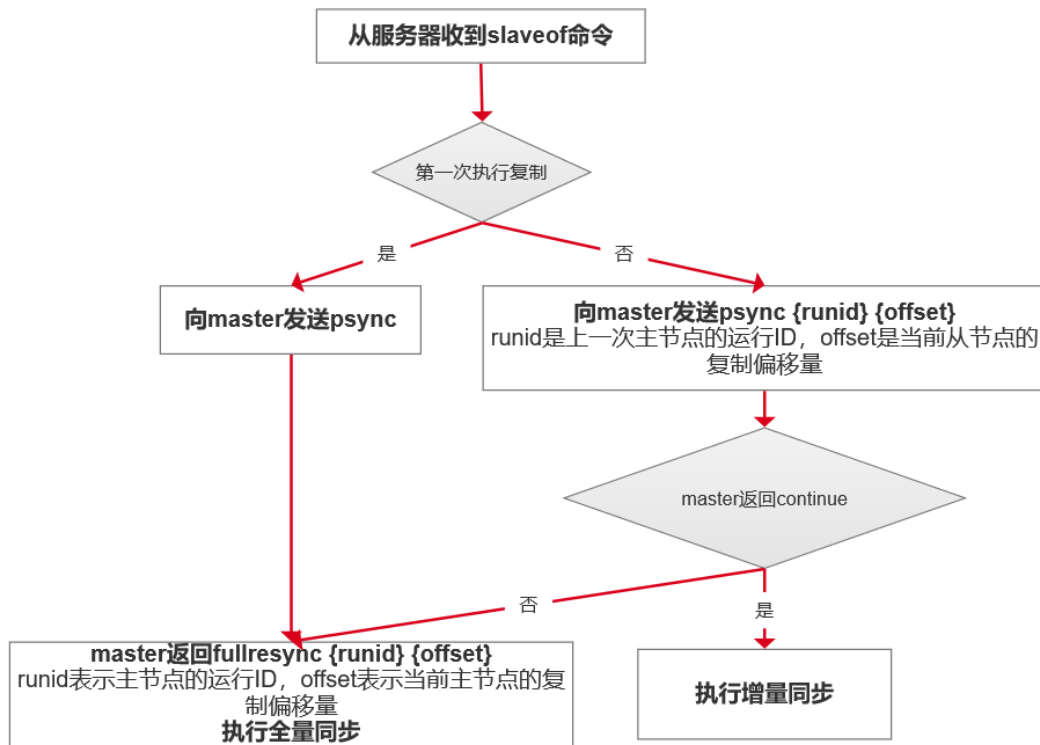
全量复制:

- 从节点发送psync命令，psync runid offset（由于是第一次，runid为?，offset为-1）
- 主节点返回FULLRESYNC runid offset，runid是主节点的runid，offset是主节点目前的offset。从节点保存信息
- 主节点启动bgsave命令fork子进程进行RDB持久化
- 主节点将RDB文件发送给从节点，到从节点加载数据完成之前，写命令写入缓冲区
- 从节点清理本地数据并加载RDB，如果开启了AOF会重写AOF

部分复制:

1. 复制偏移量: psync runid offset
2. 复制积压缓冲区: 当主从节点offset的差距过大超过缓冲区长度时，将无法执行部分复制，只能执行全量复制。
 - 如果从节点保存的runid与主节点现在的runid相同，说明主从节点之前同步过，主节点会继续尝试使用部分复制(到底能不能部分复制还要看offset和复制积压缓冲区的情况);
 - 如果从节点保存的runid与主节点现在的runid不同，说明从节点在断线前同步的Redis节点并不是当前的主节点，只能进行全量复制。

过程原理:



简述Redis事务实现

1、事务开始

`MULTI`命令的执行，标识着一个事务的开始。`MULTI`命令会将客户端状态的 `flags` 属性中打开 `REDIS_MULTI` 标识来完成的。

2、命令入队

当一个客户端切换到事务状态之后，服务器会根据这个客户端发送来的命令来执行不同的操作。如果客户端发送的命令为`MULTI`、`EXEC`、`WATCH`、`DISCARD`中的一个，立即执行这个命令，否则将命令放入一个事务队列里面，然后向客户端返回 `QUEUED` 回复

- 如果客户端发送的命令为 `EXEC`、`DISCARD`、`WATCH`、`MULTI` 四个命令的其中一个，那么服务器立即执行这个命令。
- 如果客户端发送的是四个命令以外的其他命令，那么服务器并不立即执行这个命令。
首先检查此命令的格式是否正确，如果不正确，服务器会在客户端状态（`redisClient`）的 `flags` 属性关闭 `REDIS_MULTI` 标识，并且返回错误信息给客户端。
如果正确，将这个命令放入一个事务队列里面，然后向客户端返回 `QUEUED` 回复

事务队列是按照FIFO的方式保存入队的命令

3、事务执行

客户端发送 `EXEC` 命令，服务器执行 `EXEC` 命令逻辑。

- 如果客户端状态的 flags 属性不包含 REDIS_MULTI 标识，或者包含 REDIS_DIRTY_CAS 或者 REDIS_DIRTY_EXEC 标识，那么就直接取消事务的执行。
- 否则客户端处于事务状态（flags 有 REDIS_MULTI 标识），服务器会遍历客户端的事务队列，然后执行事务队列中的所有命令，最后将返回结果全部返回给客户端；

redis 不支持事务回滚机制，但是它会检查每一个事务中的命令是否正确。

Redis 事务不支持检查那些程序员自己逻辑错误。例如对 String 类型的数据库键执行对 HashMap 类型的操作！

- WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set（CAS）行为。可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。
- MULTI命令用于开启一个事务，它总是返回OK。MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。
- EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。当操作被打断时，返回空值 nil。
- 通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务，并且客户端会从事务状态中退出。
- UNWATCH命令可以取消watch对所有key的监控。

简述redis数据结构

String：字符串

List：列表

Hash：哈希表

Set：无序集合

Sorted Set：有序集合

bitmap：布隆过滤器

GeoHash：坐标，借助Sorted Set实现，通过zset的score进行排序就可以得到坐标附近的其它元素，通过将score还原成坐标值就可以得到元素的原始坐标

HyperLogLog：统计不重复数据，用于大数据基数统计

Streams：内存版的kafka

数据库实现分布式锁的问题及解决方案

利用唯一约束键存储key，insert成功则代表获取锁成功，失败则获取失败，操作完成需要删除锁

问题：

- 非阻塞，锁获取失败后没有排队机制，需要自己编码实现阻塞，可以使用自旋，直到获取锁
- 不可重入，如果加锁的方法需要递归，则第二次插入会失败，可以使用记录线程标识解决重入问题
- 死锁，删除锁失败、则其他线程没办法获取锁，可以设置超时时间、使用定时任务检查
- 数据库单点故障，数据库高可用

redis分布式锁实现

setnx+setex：存在设置超时时间失败的情况，导致死锁

set(key,value,nx,px)：将setnx+setex变成原子操作

问题：

- 任务超时，锁自动释放，导致并发问题。使用redisson解决(看门狗监听，自动续期)
 - 以及加锁和释放锁不是同一个线程的问题。在value中存入uuid(线程唯一标识)，删除锁时判断该标识（使用lua保证原子操作）
 - 不可重入，使用redisson解决（实现机制类似AQS，计数）
 - 异步复制可能造成锁丢失，使用redLock解决
1. 顺序向五个节点请求加锁
 2. 根据一定的超时时间来推断是不是跳过该节点
 3. 三个节点加锁成功并且花费时间小于锁的有效期
 4. 认定加锁成功

分布式高可用

什么是服务降级、什么是熔断

降级是解决系统资源不足和海量业务请求之间的矛盾

在暴增的流量请求下，对一些非核心流程业务、非关键业务，进行有策略的放弃，以此来释放系统资源，保证核心业务的正常运行，尽量避免这种系统资源分配的不平衡，打破二八策略，让更多的机器资源，承载主要的业务请求。服务降级不是一个常态策略，而是应对非正常情况下的应急策略。服务降级的结果，通常是对一些业务请求，返回一个统一的结果，可以理解为是一种 FailOver 快速失败的策略。一般通过配置中心配置开关实现开启降级

熔断模式保护的是业务系统不被外部大流量或者下游系统的异常而拖垮。

如果开启了熔断，订单服务可以在下游调用出现部分异常时，调节流量请求，比如在出现 10% 的失败后，减少 50% 的流量请求，如果继续出现 50% 的异常，则减少 80% 的流量请求；相应的，在检测的下游服务正常后，首先恢复 30% 的流量，然后是 50% 的流量，接下来是全部流量

高并发场景下如何实现系统限流？

限流一般需要结合容量规划和压测来进行。当外部请求接近或者达到系统的最大阈值时，触发限流，采取其他的手段进行降级，保护系统不被压垮。常见的降级策略包括延迟处理、拒绝服务、随机拒绝等。

计数器法：

- 1、将时间划分为固定的窗口大小，例如1s
- 2、在窗口时间段内，每来一个请求，对计数器加1。
- 3、当计数器达到设定限制后，该窗口时间内的之后的请求都被丢弃处理。
- 4、该窗口时间结束后，计数器清零，从新开始计数。

滑动窗口计数法：

1. 将时间划分为细粒度的区间，每个区间维持一个计数器，每进入一个请求则将计数器加一。
2. 多个区间组成一个时间窗口，每流逝一个区间时间后，则抛弃最老的一个区间，纳入新区间。
3. 若当前窗口的区间计数器总和超过设定的限制数量，则本窗口内的后续请求都被丢弃。

漏桶算法：如果外部请求超出当前阈值，则会在容易里积蓄，一直到溢出，系统并不关心溢出的流量。从出口处限制请求速率，并不存在计数器法的临界问题，请求曲线始终是平滑的。无法应对突发流量，相当于一个空桶+固定处理线程

令牌桶算法：假设一个大小恒定的桶，这个桶的容量和设定的阈值有关，桶里放着很多令牌，通过一个固定的速率，往里边放入令牌，如果桶满了，就把令牌丢掉，最后桶中可以保存的最大令牌数永远不会超过桶的大小。当有请求进入时，就尝试从桶里取走一个令牌，如果桶里是空的，那么这个请求就会被拒绝。

消息中间件

消息队列的优缺点，使用场景

优点：

- 1、解耦，降低系统之间的依赖
- 2、异步处理，不需要同步等待
- 3、削峰填谷，将流量从高峰期引到低谷期进行处理

缺点：

- 1、增加了系统的复杂度，幂等、重复消费、消息丢失等问题的带入
- 2、系统可用性降低，mq的故障会影响系统可用
- 3、一致性，消费端可能失败

场景：日志采集、发布订阅等

如何保证消息不被重复消费

幂等：一个数据或者一个请求，重复来多次，确保对应的数据是不会改变的，不能出错。

思路：

- 如果是写 redis，就没问题，反正每次都是 set，天然幂等性
- 生产者发送消息的时候带上一个全局唯一的id,消费者拿到消息后，先根据这个id去 redis里查一下，之前有没消费过，没有消费过就处理，并且写入这个 id 到 redis，如果消费过了，则不处理。
- 基于数据库的唯一键

Kafka、ActiveMQ、RabbitMQ、RocketMQ 对比

ActiveMQ：JMS规范，支持事务、支持XA协议，没有生产大规模支撑场景、官方维护越来越少

RabbitMQ：erlang语言开发、性能好、高并发，支持多种语言，社区、文档方面有优势，erlang语言不利于java程序员二次开发，依赖开源社区的维护和升级，需要学习AMQP协议、学习成本相对较高

以上吞吐量单机都在万级

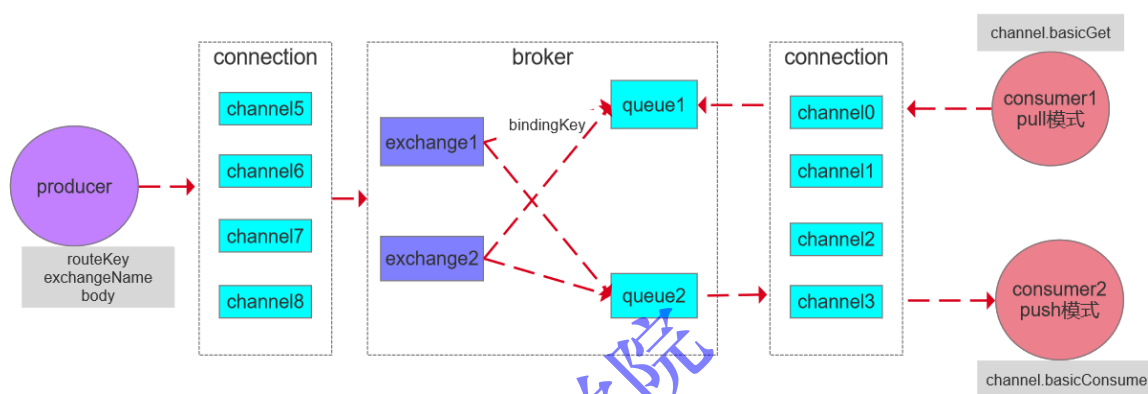
kafka：高性能，高可用，生产环境有大规模使用场景，单机容量有限（超过64个分区响应明显变长）、社区更新慢

吞吐量单机百万

rocketmq: java实现, 方便二次开发、设计参考了kafka, 高可用、高可靠, 社区活跃度一般、支持语言较少

吞吐量单机十万

RabbitMQ架构设计



Broker: rabbitmq的服务节点

Queue: 队列, 是RabbitMQ的内部对象, 用于存储消息。RabbitMQ中消息只能存储在队列中。生产者投递消息到队列, 消费者从队列中获取消息并消费。多个消费者可以订阅同一个队列, 这时队列中的消息会被平均分摊(轮询)给多个消费者进行消费, 而不是每个消费者都收到所有的消息进行消费。(注意: RabbitMQ不支持队列层面的广播消费, 如果需要广播消费, 可以采用一个交换器通过路由Key绑定多个队列, 由多个消费者来订阅这些队列的方式。

Exchange: 交换器。生产者将消息发送到Exchange, 由交换器将消息路由到一个或多个队列中。如果路由不到, 或返回给生产者, 或直接丢弃, 或做其它处理。

RoutingKey: 路由Key。生产者将消息发送给交换器的时候, 一般会指定一个RoutingKey, 用来指定这个消息的路由规则。这个路由Key需要与交换器类型和绑定键(BindingKey)联合使用才能最终生效。在交换器类型和绑定键固定的情况下, 生产者可以在发送消息给交换器时通过指定RoutingKey来决定消息流向哪里。

bindingKey: 通过绑定将交换器和队列关联起来, 在绑定的时候一般会指定一个绑定键, 这样RabbitMQ就可以指定如何正确的路由到队列了。

交换器和队列实际上是多对多关系。就像关系数据库中的两张表。他们通过BindingKey做关联(多对多关系表)。在投递消息时, 可以通过Exchange和RoutingKey(对应BindingKey)就可以找到相对应的队列。

信道: 信道是建立在Connection 之上的虚拟连接。当应用程序与Rabbit Broker建立TCP连接的时候, 客户端紧接着可以创建一个AMQP 信道(Channel), 每个信道都会被指派一个唯一的ID。RabbitMQ 处理的每条AMQP 指令都是通过信道完成的。信道就像电缆里的光纤束。一条电缆内含有许多光纤束, 允许所有的连接通过多条光线束进行传输和接收。

vhost: 虚拟主机, 每一个应用可以指定不同的vhost, 此时对于应用来说、vhost就是broker

简述RabbitMq的交换机类型

交换机分发会先找出绑定的队列，然后再判断 `routekey`，来决定是否将消息分发到某一个队列中

```
channel channel = connection.createChannel();    //在rabbitmq中创建一个信道
channel.exchangeDeclare("exchangeName", "direct"); //创建一个type为direct的交换机
channel.queueDeclare("queueName");    //创建一个队列
channel.queueBind("queueName", "exchangeName", "zhangsna");    //绑定并设置路由键
channel.queueBind("queueName", "exchangeName", "lisi");    //绑定并设置路由键
channel.queueBind("queueName", "exchangeName", "wangwu"); //绑定并设置路由键
```

fanout：扇形交换机，不再判断routekey，直接将消息分发到所有绑定的队列

direct：判断routekey的规则是完全匹配模式，即发送消息时指定的routekey要等于绑定的routekey

topic：判断routekey的规则是模糊匹配模式

header：绑定队列与交换机的时候指定一个键值对，当交换机在分发消息的时候会先解开消息体里的 `headers` 数据，然后判断里面是否有所设置的键值对，如果发现匹配成功，才将消息分发到队列中；这种交换机类型在性能上相对来说较差，在实际工作中很少会用到

rabbitmq可以直连队列么？

生产者和消费者使用相同的参数声明队列。重复声明不会改变队列

```
//生产者
channel.queueDeclare(QueueName, false, false, false, null);
//发送10条消息，依次在消息后面附加1-10个点
for (int i = 1; i <= 10; i++)
{
    String message = "hello" + i;
    channel.basicPublish("", QueueName, null, message.getBytes());
}

//消费者
channel.queueDeclare(QueueName, false, false, false, null);
QueueingConsumer consumer = new QueueingConsumer(channel);
// 指定消费队列
channel.basicConsume(QueueName, true, consumer);
while (true)
{
}
```

```

QueueingConsumer.Delivery delivery = consumer.nextDelivery();
String message = new String(delivery.getBody());
dowork(message);
}

```

```
channel.queueDeclare(queue, durable, exclusive, autoDelete, arguments) ;
```

queue: 队列名字

durable: 队列持久化标志, **true** 为持久化队列

exclusive: **exclusive:** 排他队列, 仅对创建的链接可见、链接中的 **channel** 都可见, 其他链接不能重复声明, 链接关闭队列会被自动删除

autoDelete: 自动删除, 如果该队列没有任何订阅的消费者的话, 该队列会被自动删除。这种队列适用于临时队列。

arguments: Map 类型, 队列参数设置

x-message-ttl: 数字, 消息队列中消息的存活时间, 超过会被删除

x-expires: 数字, 队列自身的空闲存活时间, 指定时间内没有被访问, 就会被删除

x-max-length 和 **x-max-length-bytes:** 队列最大长度和空间, 超出会删除老的数据

x-dead-letter-exchange 和 **x-dead-letter-routing-key:** 设置死信

x-max-priority: 队列支持的优先级别, 需要生产者在发送消息时指定, 消息按照优先级从高到底分发给消费者

```
channel.basicPublish(exchange, routingKey, mandatory, immediate,
basicProperties, body);
```

exchange: 交换机名

routingKey: 路由键

mandatory: 为 **true** 时, 如果 **exchange** 根据自身类型和消息 **routeKey** 无法找到一个符合条件的 **queue**, 那么会调用 **basic.return** 方法将消息返还给生产者, **channel.addReturnListener** 添加一个监听器, 当 **broker** 执行 **basic.return** 方法时, 会回调 **handleReturn** 方法, 这样就可以处理变为死信的消息了; 设为 **false** 时, 出现上述情形 **broker** 会直接将消息扔掉;

immediate: 3.0 以前这个标志告诉服务器如果该消息关联的 **queue** 上有消费者, 则马上将消息投递给它, 如果所有 **queue** 都没有消费者, 直接把消息返还给生产者, 不用将消息入队列等待消费者了。3.0 之后取消了该参数

basicProperties: 消息的详细属性, 优先级别、持久化、到期时间等, **headers** 类型的 **exchange** 要用到的是其中的 **headers** 字段。

body: 消息实体, 字节数组。

QueueingConsumer: 一个已经实现好了的 **Consumer**, 相比于自己实现 **Consumer** 接口, 这是个比较安全快捷的方式。该类基于 **jdk** 的 **BlockingQueue** 实现, **handleDelivery** 方法中将收到的消息封装成 **Delivery** 对象, 并存放到 **BlockingQueue** 中, 这相当于消费者本地存放了一个消息缓存队列。**nextDelivery()** 方法底层调用的 **BlockingQueue** 的阻塞方法 **take()**。

```
channel.basicConsume(queue, autoAck, consumer);
```

queue: 队列名。

autoAck: 自动应答标志, **true** 为自动应答。

consumer: 消费者对象, 可以自己实现 **Consumer** 接口, 建议使用 **QueueingConsumer**。

rabbitmq的持久化机制

- 1、交换机持久化：exchange_declare创建交换机时通过参数指定
- 2、队列持久化：queue_declare创建队列时通过参数指定
- 3、消息持久化：new AMQPMessage创建消息时通过参数指定

append的方式写文件，会根据大小自动生成新的文件，rabbitmq启动时会创建两个进程，一个负责持久化消息的存储，另一个负责非持久化消息的存储(内存不够时)

消息存储时会在ets表中记录消息在文件中的映射以及相关信息（包括id、偏移量，有效数据，左边文件，右边文件），消息读取时根据该信息到文件中读取、同时更新信息

消息删除时只从ets删除，变为垃圾数据，当垃圾数据超出比例（默认50%），并且文件数达到3个，触发垃圾回收，锁定左右两个文件，整理左边文件有效数据、将右边文件有效数据写入左边，更新文件信息，删除右边，完成合并。当一个文件的有用数据等于0时，删除该文件。

写入文件前先写buffer缓冲区，如果buffer已满，则写入文件(此时只是操作系统的页存)

每隔25ms刷一次磁盘，不管buffer满没满，都将buffer和页存中的数据落盘

每次消息写入后，如果没有后续写入请求，则直接刷盘

RabbitMQ事务消息

通过对信道设置实现

1. channel.txSelect(); 通知服务器开启事务模式；服务端会返回Tx.Select-Ok
2. channel.basicPublish; 发送消息，可以是多条，可以是消费消息提交ack
3. channel.txCommit()提交事务；
4. channel.txRollback()回滚事务；

消费者使用事务：

1. autoAck=false，手动提交ack，以事务提交或回滚为准；
2. autoAck=true，不支持事务的，也就是说你即使在收到消息之后在回滚事务也是于事无补的，队列已经把消息移除了

如果其中任意一个环节出现问题，就会抛出IOException异常，用户可以拦截异常进行事务回滚，或决定要不要重复消息。事务消息会降低rabbitmq的性能

RabbitMQ如何保证消息的可靠性传输

- 1、使用事务消息

2、使用消息确认机制

发送方确认：

- channel设置为confirm模式，则每条消息会被分配一个唯一id
- 消息投递成功，信道会发送ack给生产者，包含了id，回调ConfirmCallback接口
- 如果发生错误导致消息丢失，发生nack给生产者。回调ReturnCallback接口
- ack和nack只有一个触发，且只有一次，异步触发。可以继续发送消息

接收方确认：

- 声明队列时，指定noack=false，broker会等待消费者手动返回ack、才会删除消息，否则立刻删除
- broker的ack没有超时机制，只会判断链接是否断开，如果断开、消息会被重新发送

rabbitmq的死信队列、延迟队列原理

死信消息：

1. 消息被消费方否定确认，使用 `channel.basicNack` 或 `channel.basicReject`，并且此时 `requeue` 属性被设置为 `false`。
2. 消息在队列的存活时间超过设置的TTL时间。
3. 消息队列的消息数量已经超过最大队列长度。

那么该消息将成为死信消息。如果配置了死信队列信息，那么该消息将会被丢进死信队列中，如果没有配置，则该消息将会被丢弃

为每个需要使用死信的业务队列配置一个死信交换机，同一个项目的死信交换机可以共用一个，然后为每个业务队列分配一个单独的routeKey，死信队列只不过是绑定在死信交换机上的队列，死信交换机也不是什么特殊的交换机，只不过是用来接受死信的交换机，所以可以为任何类型【Direct、Fanout、Topic】

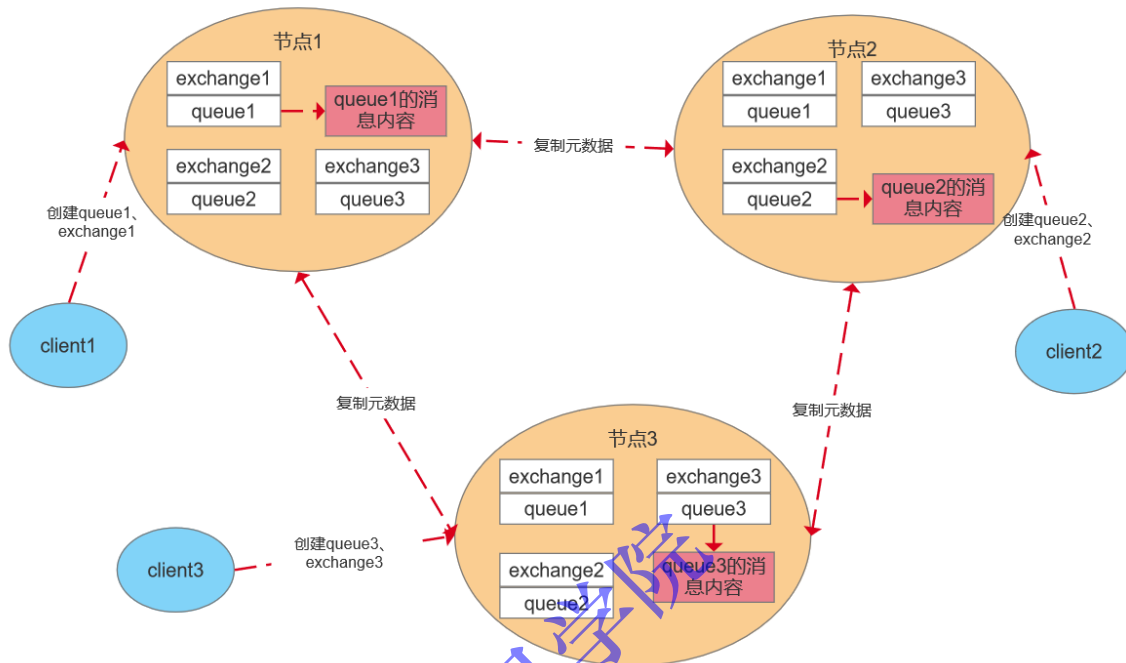
TTL：一条消息或者该队列中的所有消息的最大存活时间

如果一条消息设置了TTL属性或者进入了设置TTL属性的队列，那么这条消息如果在TTL设置的时间内没有被消费，则会成为“死信”。如果同时配置了队列的TTL和消息的TTL，那么较小的那个值将会被使用。

只需要消费者一直消费死信队列里的消息

```
arguments.put("x-dead-letter-exchange", "dlx.exchange");
channel.queueDeclare(queueName, true, false, false, arguments);
channel.queueBind(queueName, exchangeName, routingKey);
channel.exchangeDeclare("dlx.exchange", "topic", true, false, null);
channel.queueDeclare("dlx.queue", true, false, false, null);
channel.queueBind("dlx.queue", "dlx.exchange", "#");
```


rabbitmq的普通集群原理



元数据：

- 队列元数据：队列名称和它的属性
- 交换器元数据：交换器名称、类型和属性
- 绑定元数据：一张简单的表格展示了如何将消息路由到队列
- vhost元数据：为vhost内的队列、交换器和绑定提供命名空间和安全属性

为什么只同步元数据：

- 存储空间，每一个节点都保存全量数据，影响消息堆积能力
- 性能，消息的发布者需要将消息复制到每一个集群节点

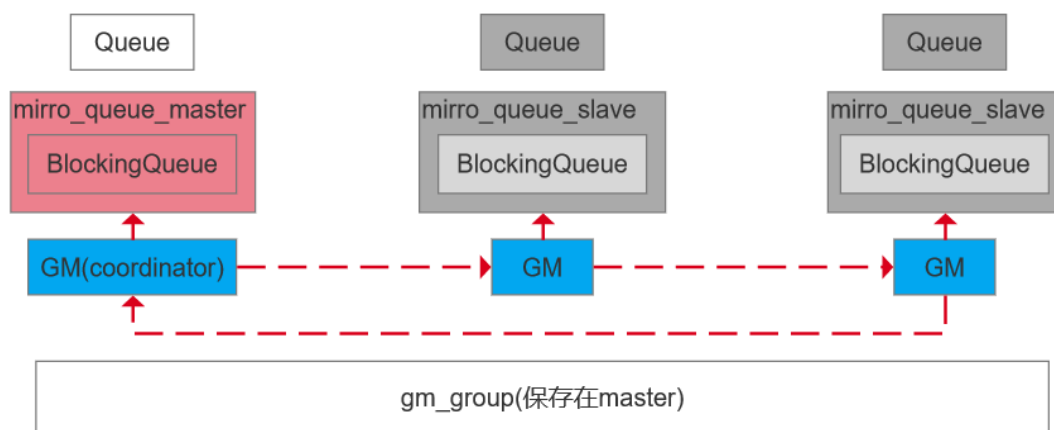
客户端连接的是非队列数据所在节点：则该节点会进行路由转发，包括发送和消费

集群节点类型：

- 磁盘节点：将配置信息和元信息存储在磁盘上。
- 内存节点：将配置信息和元信息存储在内存中。性能优于磁盘节点。依赖磁盘节点进行持久化

RabbitMQ要求集群中至少有一个磁盘节点，当节点加入和离开集群时，必须通知磁盘节点（如果集群中唯一的磁盘节点崩溃了，则不能进行创建队列、创建交换器、创建绑定、添加用户、更改权限、添加和删除集群节点）。如果唯一磁盘的磁盘节点崩溃，集群是可以保持运行的，但不能更改任何东西。因此建议在集群中设置两个磁盘节点，只要一个可以，就能正常操作。

rabbitmq的镜像队列原理

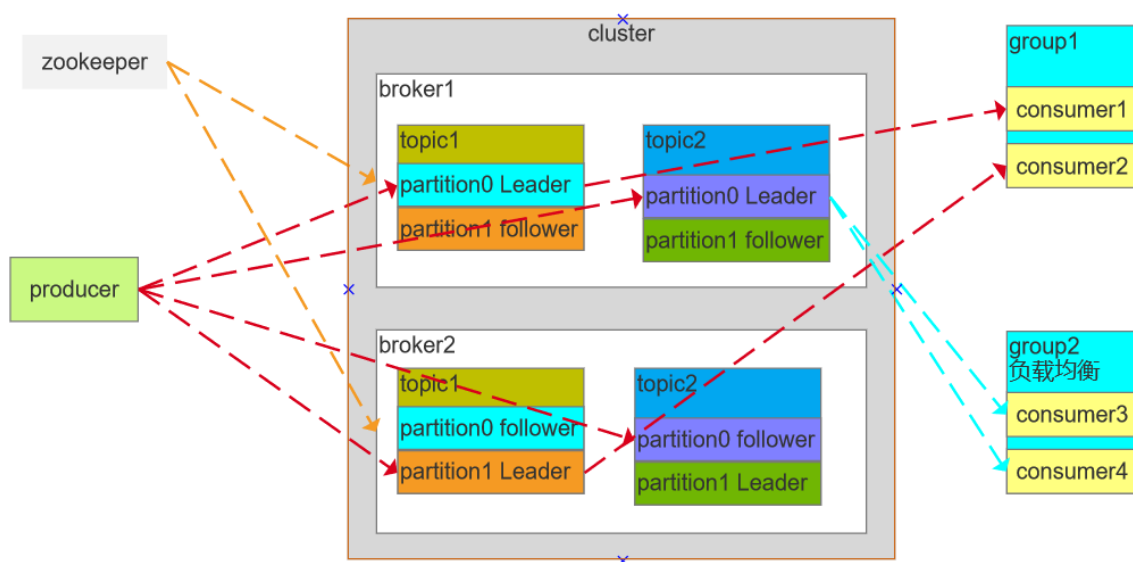


GM负责消息的广播，所有的GM组成gm_group，形成链表结构，负责监听相邻节点的状态，以及传递消息到相邻节点，master的GM收到消息时代表消息同步完成

mirror_queue_master/slave负责消息的处理，操作blockingQueue，Queue负责AMQP协议（commit、rollback、ack等）

master处理读写

kafka架构设计



Consumer Group: 消费者组，消费者组内每个消费者负责消费不同分区的数据，提高消费能力。逻辑上的一个订阅者。

Topic: 可以理解为一个队列，Topic 将消息分类，生产者和消费者面向的是同一个 Topic。

Partition: 为了实现扩展性，提高并发能力，一个Topic 以多个Partition的方式分布到多个 Broker 上，每个 Partition 是一个有序的队列。一个 Topic 的每个Partition都有若干个副本（Replica），一个 Leader 和若干个 Follower。生产者发送数据的对象，以及消费者消费数据的对象，都是 Leader。Follower负责实时从 Leader 中同步数据，保持和 Leader 数据的同步。Leader 发生故障时，某个 Follower 还会成为新的 Leader。

Zookeeper: Kafka 集群能够正常工作，需要依赖于 Zookeeper，Zookeeper 帮助 Kafka 存储和管理集群信息。

kafka高性能高吞吐的原因

1、磁盘顺序读写：保证了消息的堆积

- 顺序读写，磁盘会预读，预读即在读取的起始地址连续读取多个页面，主要时间花费在了传输时间，而这个时间两种读写可以认为是一样的。
- 随机读写，因为数据没有在一起，将预读浪费掉了。需要多次寻道和旋转延迟。而这个时间可能是传输时间的许多倍。

2、零拷贝：避免 CPU 将数据从一块存储拷贝到另外一块存储的技术

- 传统的数据复制：
 - 1、读取磁盘文件数据到内核缓冲区
 - 2、将内核缓冲区的数据copy到用户缓冲区
 - 2、将用户缓冲区的数据copy到socket的发送缓冲区
 - 3、将socket发送缓冲区中的数据发送到网卡、进行传输
- 零拷贝：
磁盘文件->内核空间读取缓冲区->网卡接口->消费者进程

3、分区分段+索引

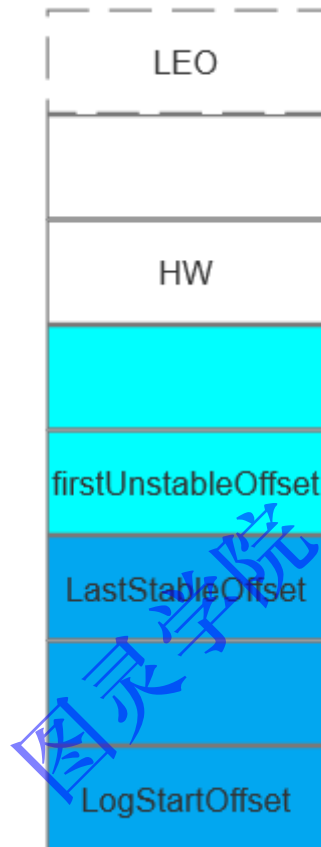
Kafka的message消息实际上是分布式存储在一个一个小的segment中的，每次文件操作也是直接操作的segment。为了进一步的查询优化，Kafka又默认为分段后的数据文件建立了索引文件，就是文件系统上的.index文件。这种分区分段+索引的设计，不仅提升了数据读取的效率，同时也提高了数据操作的并行度

4、批量压缩：多条消息一起压缩，降低带宽

5、批量读写

6、直接操作page cache，而不是JVM、避免GC耗时及对象创建耗时，且读写速度更高，进程重启、缓存也不会丢失

kafka的副本同步机制



LEO: 下一条待写入位置

firstUnstableOffset: 第一条未提交数据

LastStableOffset: 最后一条已提交数据

LogStartOffset: 起始位置

isolation.level=read_committed: 只能消费到LastStableOffset, read_committed可以消费到HW的上一条

一个partition对应的ISR中最小的LEO作为分区的HW, consumer最多只能消费到HW所在的位置

leader收消息后会更新本地的LEO, leader还会维护follower的LEO即remote LEO, follower发出fetch同步数据请求时(携带自身的LEO)、leader会更新remote LEO, 更新分区的HW, 然后将数据响应给follower、follower更新自身HW(取响应中的HW和自身的LEO中的较小值), LEO+1

ISR: 如果一个follower落后leader不超过某个时间阈值, 那么则则ISR中, 否则将放在OSR中。

同步副本时，follower获取leader的LEO和LogStartOffset，与本地对比、如果本地的LogStartOffset超出了leader的值，则超过这个值的数据删除，再进行同步，如果本地的小于leader的、则直接同步

Kafka消息高可靠解决方案

消息发送：

- ack: 0、不重试，1、lead写入成功就返回了，all/-1、等待ISR同步完再返回
- unclean.leader.election.enable : false，禁止选举ISR以外的follower为leader
- tries > 1，重试次数
- min.insync.replicas > 1：同步副本数，没满足该值前、不提供读写服务、写操作会异常

消费：

手工提交offset

broker：减小刷盘间隔

事务消息

kafka的rebalance机制

consumer group中的消费者与topic下的partition重新匹配的过程

何时会产生rebalance：

- consumer group中的成员个数发生变化
- consumer消费超时
- group订阅的topic个数发生变化
- group订阅的topic的分区数发生变化

coordinator：通常是partition的leader节点所在的broker，负责监控group中consumer的存活，consumer维持到coordinator的心跳，判断consumer的消费超时

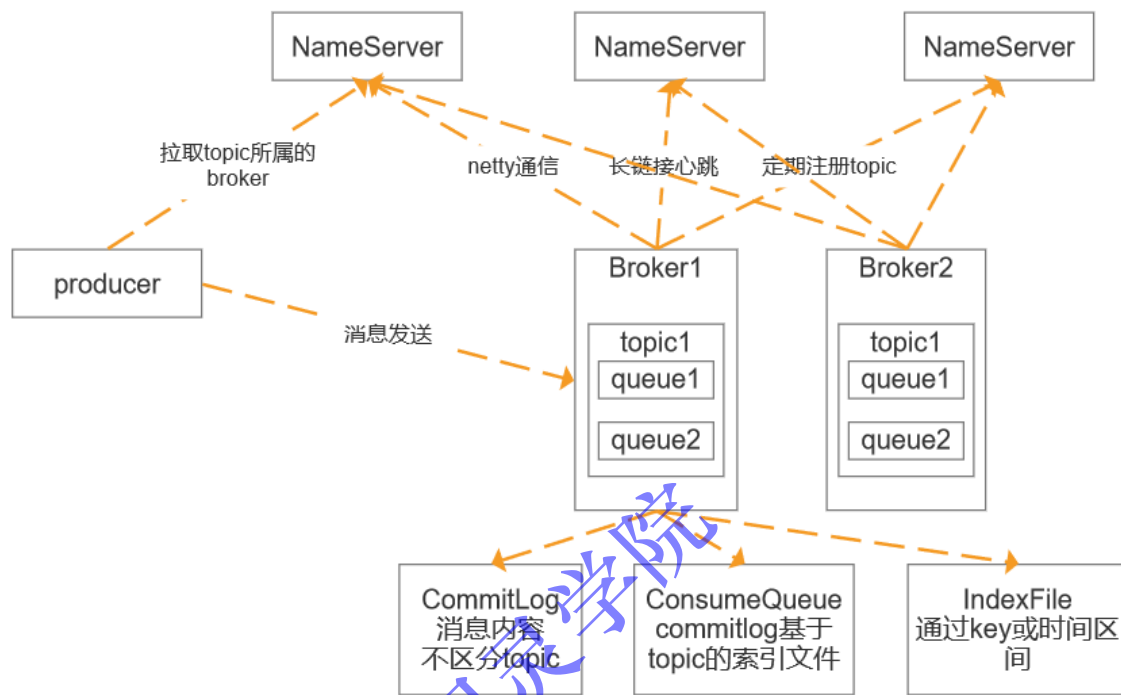
- coordinator通过心跳返回通知consumer进行rebalance
- consumer请求coordinator加入组，coordinator选举产生leader consumer
- leader consumer从coordinator获取所有的consumer，发送syncGroup(分配信息)给到coordinator
- coordinator通过心跳机制将syncGroup下发给consumer
- 完成rebalance

leader consumer监控topic的变化，通知coordinator触发rebalance

如果C1消费消息超时，触发rebalance，重新分配后、该消息会被其他消费者消费，此时C1消费完成提交offset、导致错误

解决：coordinator每次rebalance，会标记一个Generation给到consumer，每次rebalance该Generation会+1，consumer提交offset时，coordinator会比对Generation，不一致则拒绝提交

RocketMQ 架构设计



路由信息是包括了：BrokerServer，Topic和ConsumeQueueID等信息。

RocketMQ 事务消息原理

依赖于TransactionListener接口

- executeLocalTransaction方法会在发送消息后调用，用于执行本地事务，如果本地事务执行成功，rocketmq再提交消息
- checkLocalTransaction用于对本地事务做检查，rocketmq依赖此方法做补偿

通过两个内部的topic来实现对消息的两阶段支持，

prepare：将消息（消息上带有事务标识）投递到一个名为RMS_SYS_TRANS_HALF_TOPIC的topic中，而不是投递到真正的topic中。

commit/rollback: producer再通过TransactionListener的executeLocalTransaction方法执行本地事务, 当producer的localTransaction处理成功或者失败后, producer会向broker发送commit或rollback命令, 如果是commit, 则broker会将投递到RMQ_SYS_TRANS_HALF_TOPIC中的消息投递到真实的topic中, 然后再投递一个表示删除的消息到RMQ_SYS_TRANS_OP_HALF_TOPIC中, 表示当前事务已完成; 如果是rollback, 则没有投递到真实topic的过程, 只需要投递表示删除的消息到RMQ_SYS_TRANS_OP_HALF_TOPIC。最后, 消费者和消费普通的消息一样消费事务消息

- 第一阶段 (prepare) 失败: 给应用返回发送消息失败
- 事务失败: 发送回滚命令给broker, 由broker执行消息的回滚
- Commit或rollback失败: 由broker定时向producer发起事务检查, 如果本地事务成功, 则提交消息事务, 否则回滚消息事务

事务状态的检查有两种情况:

- commit/rollback: broker会执行相应的commit/rollback操作
- 如果是TRANSACTION_NOT_TYPE, 则一段时间后再次检查, 当检查的次数超过上限 (默认15次) 则丢弃消息

RocketMQ 顺序消息原理

默认是不能保证的, 需要程序保证发送和消费的是同一个queue, 多线程消费也无法保证

发送顺序: 发送端自己业务逻辑保证先后, 发往一个固定的queue, 生产者可以在消息体上设置消息的顺序

发送者实现MessageQueueSelector接口, 选择一个queue进行发送, 也可使用rocketmq提供的默认实现

- SelectMessageQueueByHash: 按参数的hashcode与可选队列进行求余选择
- SelectMessageQueueByRandom: 随机选择

mq: queue本身就是顺序追加写, 只需保证一个队列统一时间只有一个consumer消费, 通过加锁实现, consumer上的顺序消费有一个定时任务、每隔一定时间向broker发送请求延长锁定

消费端:

pull模式: 消费者需要自己维护需要拉取的queue, 一次拉取的消息都是顺序的, 需要消费端自己保证顺序消费

push模式: 消费实例实现自MQPushConsumer接口, 提供注册监听的方法消费消息, registerMessageListener、重载方法

- MessageListenerConcurrently: 并行消费
- MessageListenerOrderly: 串行消费, consumer会把消息放入本地队列并加锁, 定时任务保证锁的同步

简述RocketMQ 持久化机制

- commitLog: 日志数据文件, 被所有的queue共享, 大小为1G, 写满之后重新生成, 顺序写
- consumeQueue: 逻辑queue, 消息先到达commitLog、然后异步转发到consumeQueue, 包含queue在CommitLog中的物理位置偏移量 Offset, 消息实体内容的大小和 Message Tag 的 hash 值。大小约为 600W 个字节, 写满之后重新生成, 顺序写
- indexFile: 通过 key 或者时间区间来查找 CommitLog 中的消息, 文件名以创建的时间戳命名, 固定的单个 IndexFile 大小为 400M, 可以保存 2000W 个索引

所有队列共用一个日志数据文件, 避免了kafka的分区数过多、日志文件过多导致磁盘IO读写压力较大造成性能瓶颈, rocketmq的queue只存储少量数据、更加轻量化, 对于磁盘的访问是串行化避免磁盘竞争, 缺点在于: 写入是顺序写, 但读是随机的, 先读ConsumeQueue, 再读 CommitLog, 会降低消息读的效率

消息发送到broker后, 会被写入commitLog, 写之前加锁, 保证顺序写入。然后转发到consumeQueue

消息消费时先从consumeQueue读取消息在CommitLog中的起始物理偏移量 Offset, 消息大小、和消息 Tag 的 HashCode 值。在从CommitLog 读取消息内容

- 同步刷盘, 消息持久化到磁盘才会给生产者返回ack, 可以保证消息可靠、但是会影响性能
- 异步刷盘: 消息写入pageCache就返回ack给生产者, 刷盘采用异步线程, 降低读写延迟提高性能和吞吐

RocketMQ 如何保证不丢消息

生产者:

- 同步阻塞的方式发送消息, 加上失败重试机制, 可能broker存储失败, 可以通过查询确认
- 异步发送需要重写回调方法, 检查发送结果
- ack机制, 可能存储CommitLog, 存储ConsumerQueue失败, 此时对消费者不可见

broker: 同步刷盘、集群模式下采用同步复制、会等待slave复制完成才会返回确认

消费者:

- offset手动提交, 消息消费保证幂等

定时任务实现原理

优先队列：基于小顶堆实现，每次新增任务需要进行堆化，取任务时取堆顶元素、调整堆架构，时间复杂度是 $O(\log N)$

时间轮算法：是一个环形队列，按照时间的单位区分，每个时间单位里面是一个链表、用来存储定时任务，像时钟一样轮询环形队列，取出链表中的任务执行，如果超出了环形队列的时间粒度、可以使用多级时间轮，即使用不同维度的时间单位，就跟时钟或者水表一样，这一层的走了一圈，下一层的才走了一格，时间复杂度为 $O(1)$

图灵学院