万字总结：分布式系统的38个知识点

OSC开源社区 2022-08-12 20:36 发表于广东

以下文章来源于一灰灰blog ，作者小灰灰blog

一灰灰blog

Java常用技术探索，分享编程技巧，技术博文

天天说分布式分布式，那么我们是否知道什么是分布式，分布式会遇到什么问题，有哪些理论支撑，有哪些经典的应对方案，业界是如何设计并保证分布式系统的高可用呢？

1.架构设计

这一节将从一些经典的开源系统架构设计出发，来看一下，如何设计一个高质量的分布式系统；

而一般的设计出发点，无外乎

- 冗余：简单理解为找个备胎，现任挂掉之后，备胎顶上
- 拆分：不能让一个人承担所有的重任，拆分下，每个人负担一部分，压力均摊

1.1 主备架构

给现有的服务搭建一个备用的服务，两者功能完全一致，区别在于平时只有主应用对外提供服务能力；而备应用则只需要保证与主应用能力一致，随时待机即可，并不用对外提供服务；当主应用出现故障之后，将备应用切换为主应用，原主应用下线；迅速的主备切换可以有效的缩短故障时间

基于上面的描述，主备架构特点比较清晰

- 采用冗余的方案，加一台备用服务
- 缺点就是资源浪费

其次就是这个架构模型最需要考虑的则是如何实现主备切换？

- 人工

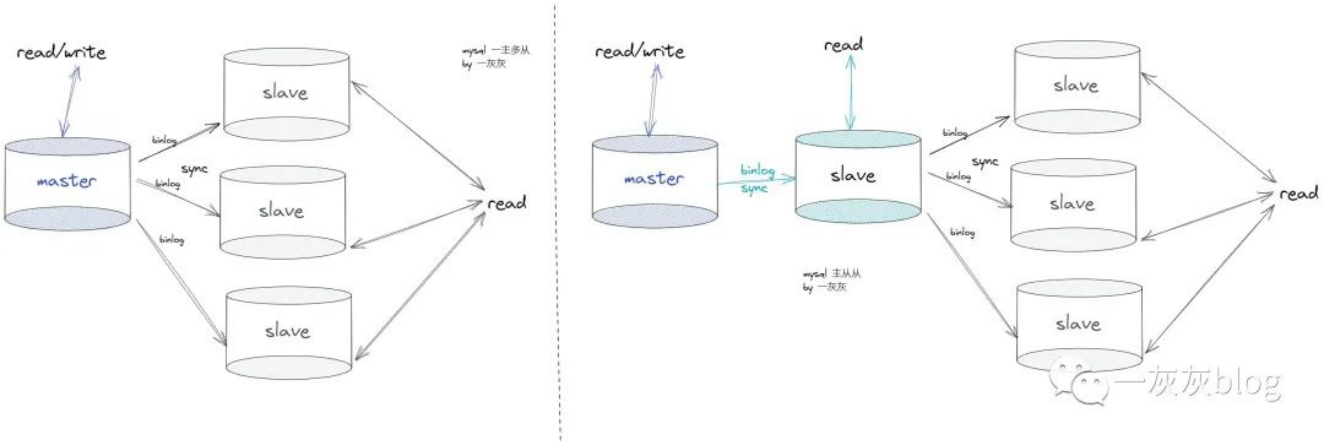
- VIP(虚拟ip) + keepalived 机制

1.2 主从架构

主从一般又叫做读写分离，主提供读写能力，而从则只提供读能力

鉴于当下的互联网应用，绝大多数都是读多写少的场景；读更容易成为性能瓶颈，所以采用读写分离，可以有效的提高整个集群的响应能力

主从架构可以区分为：一主多从 + 一主一从再多从，以mysql的主从架构模型为例进行说明



MySql主从

主从模式的主要特点在于

- 添加从，源头依然是数据冗余的思想
- 读写分离：主负责读写，从只负责读，可以视为负载均衡策略
- 从需要向主同步数据，所若有的从都同步与主，对主的压力依然可能很大；所以就有了主从从的模式

关键问题则在于

- 主从延迟
- 主的写瓶颈
- 主挂之后如何选主

1.3 多主多从架构

一主多从面临单主节点的瓶颈问题，那就考虑多主多从的策略，同样是主负责提供读写，从提供读；

但是这里有一个核心点在于多主之间的数据同步，如何保证数据的一致性是这个架构模型的重点

如MySql的双主双从可以说是一个典型的应用场景，在实际使用的时候除了上面的一致性之外，还需要考虑主键id冲突的问题

1.4 普通集群模式

无主节点，集群中所有的应用职能对等，没有主次之分（当下绝大多数的业务服务都属于这种），一个请求可以被集群中任意一个服务响应；

这种也可以叫做去中心化的设计模式，如redis的集群模式，eureka注册中心，以可用性为首要目标

对于普通集群模式而言，重点需要考虑的点在于

- 资源竞争：如何确保一个资源在同一时刻只能被一个业务操作
  - 如现在同时来了申请退款和货物出库的请求，如果不对这个订单进行加锁，两个请求同时响应，将会导致发货又退款了，导致财货两失
- 数据一致性：如何确保所有的实例数据都是一致的，或者最终是一致的
  - 如应用服务使用jvm缓存，那么如何确保所有实例的jvm缓存一致?
  - 如Eureka的分区导致不同的分区的注册信息表不一致

1.5 数据分片架构

> 这个分片模型的描述可能并不准确，大家看的时候重点理解一下这个思想

前面几个的架构中，采用的是数据冗余的方式，即所有的实例都有一个全量的数据，而这里的数据分片，则从数据拆分的思路来处理，将全量的数据，通过一定规则拆分到多个系统中，每个系统包含部分的数据，减小单个节点的压力，主要用于解决数据量大的场景

比如redis的集群方式，通过hash槽的方式进行分区

如es的索引分片存储

1.6 小结

这一节主要从架构设计层面对当前的分布式系统所采用的方案进行了一个简单的归类与小结，并不一定全面，欢迎各位大佬留言指正

基于冗余的思想：

- 主备
- 主从
- 多主多从
- 无中心集群

基于拆分的思想：

- 数据分片

> 对于拆分这一块，我们常说的分库分表也体现的是这一思想

2.理论基础

这一小节将介绍分布式系统中的经典理论，如广为流程的CAP/BASE理论，一致性理论基础paxios,raft，信息交换的Gossip协议，两阶段、三阶段等

本节主要内容参考自

- 一致性算法-Gossip协议详解 - 腾讯云开发者社区-腾讯云[1]
- P2P 网络核心技术：Gossip 协议 - 知乎[2]
- 从Paxos到Raft，分布式一致性算法解析_mb5fdb0a87e2fa1的技术博客_51CTO博客[3]
- 【理论篇】浅析分布式中的 CAP、BASE、2PC、3PC、Paxos、Raft、ZAB - 知乎[4]

2.1 CAP定理

CAP 定理指出，分布式系统 **不可能** 同时提供下面三个要求：

- Consistency：一致性
  - 操作更新完成并返回客户端之后，所有节点数据完全一致
- Availability：可用性
  - 服务一直可用
- Partition tolerance：分区容错性

- 分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足**一致性**和**可用性**的服务

通常来讲P很难不保证，当服务部署到多台实例上时，节点异常、网络故障属于常态，根据不同业务场景进行选择

对于服务有限的应用而言，首选AP，保证高可用，即使部分机器异常，也不会导致整个服务不可用；如绝大多数的前台应用都是这种

对于数据一致性要求高的场景，如涉及到钱的支付结算，CP可能更重要了

对于CAP的三种组合说明如下

选择	说明
CA	放弃分区容错性，加强一致性和可用性，其实就是传统的单机场景
AP	放弃一致性（这里说的一致性是强一致性），追求分区容错性和可用性，这是很多分布式系统设计时的选择，例如很多NoSQL系统就是如此
CP	放弃可用性，追求一致性和分区容错性，基本不会选择，网络问题会直接让整个系统不可用

2.2 BASE理论

base理论作为cap的延伸，其核心特点在于放弃强一致性，追求最终一致性

- Basically Available: 基本可用
  - 指分布式系统在出现故障的时候，允许损失部分可用性，即保证核心可用
  - 如大促时降级策略
- Soft State：软状态
  - 允许系统存在中间状态，而该中间状态不会影响系统整体可用性
  - MySql异步方式的主从同步，可能导致的主从数据不一致
- Eventual Consistency：最终一致性
  - 最终一致性是指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态

基于上面的描述，可以看到BASE理论适用于大型高可用可扩展的分布式系统

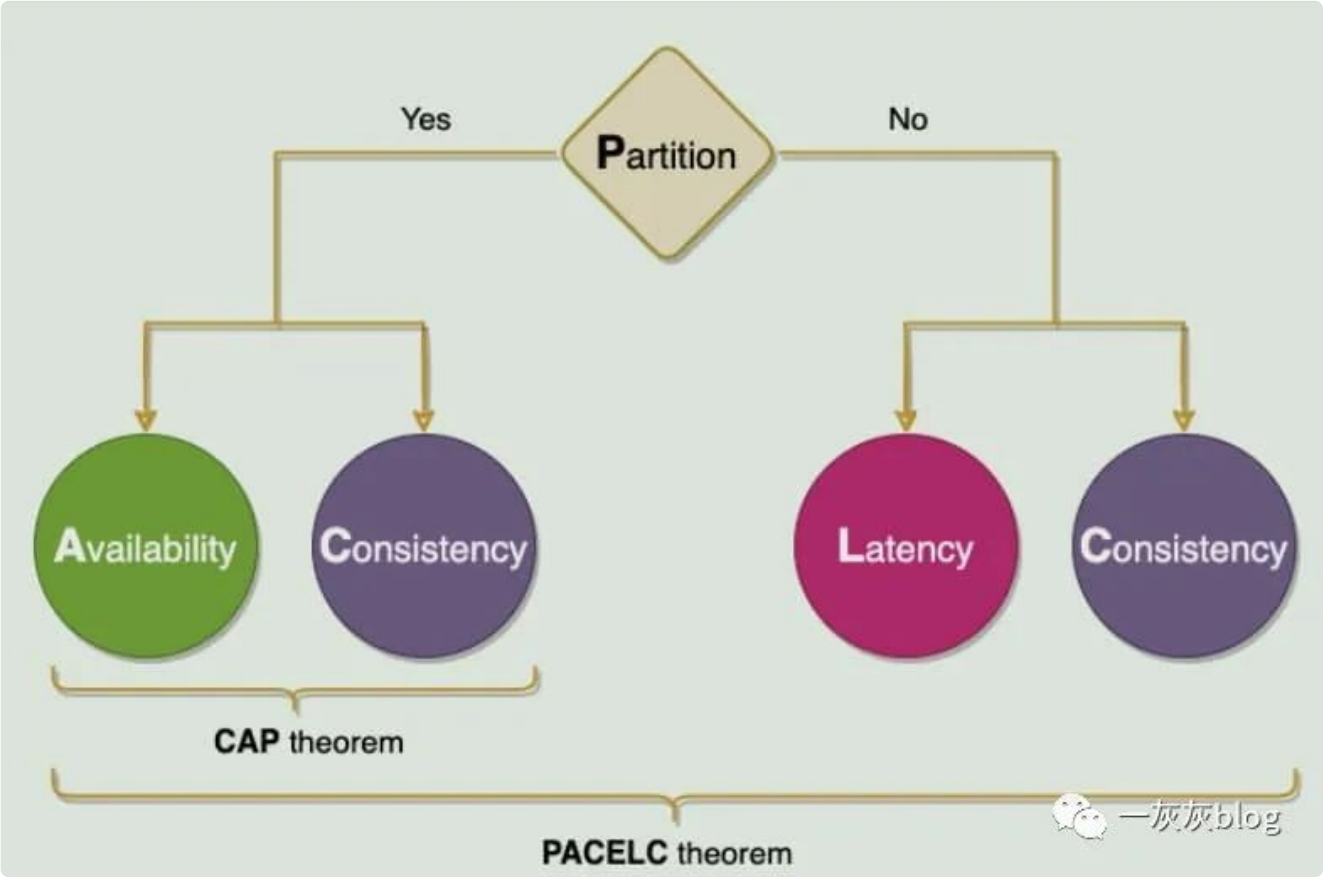
注意其不同于ACID的强一致性模型，而是通过牺牲强一致性 来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态

2.3 PACELEC 定理

> 这个真没听说过，以下内容来自：

> - Distributed System Design Patterns | by Nishant | Medium[5]

- 如果有一个分区（'P'），分布式系统可以在可用性和一致性（即'A'和'C'）之间进行权衡;
- 否则（'E'），当系统在没有分区的情况下正常运行时，系统可以在延迟（'L'）和一致性（'C'）之间进行权衡。



定理（PAC）的第一部分与CAP定理相同，ELC是扩展。整个论点假设我们通过复制来保持高可用性。因此，当失败时，CAP定理占上风。但如果没有，我们仍然必须考虑复制系统的一致性和延迟之间的权衡。

2.4 Paxos共识算法

> Paxos算法解决的问题是分布式共识性问题，即一个分布式系统中的各个进程如何就某个值（决议）通过共识达成一致

基于上面这个描述，可以看出它非常适用于选举；其工作流程

- 一个或多个提议进程 (Proposer) 可以发起提案 (Proposal)，

- Paxos算法使所有提案中的某一个提案，在所有进程中达成一致。系统中的多数派同时认可该提案，即达成了一致

角色划分：

- Proposer: 提出提案Proposal，包含编号 + value
- Acceptor: 参与决策，回应Proposers的提案；当一个提案，被半数以上的Acceptor接受，则该提案被批准
  - 每个acceptor只能批准一个提案
- Learner: 不参与决策，获取最新的提案value

2.5 Raft算法

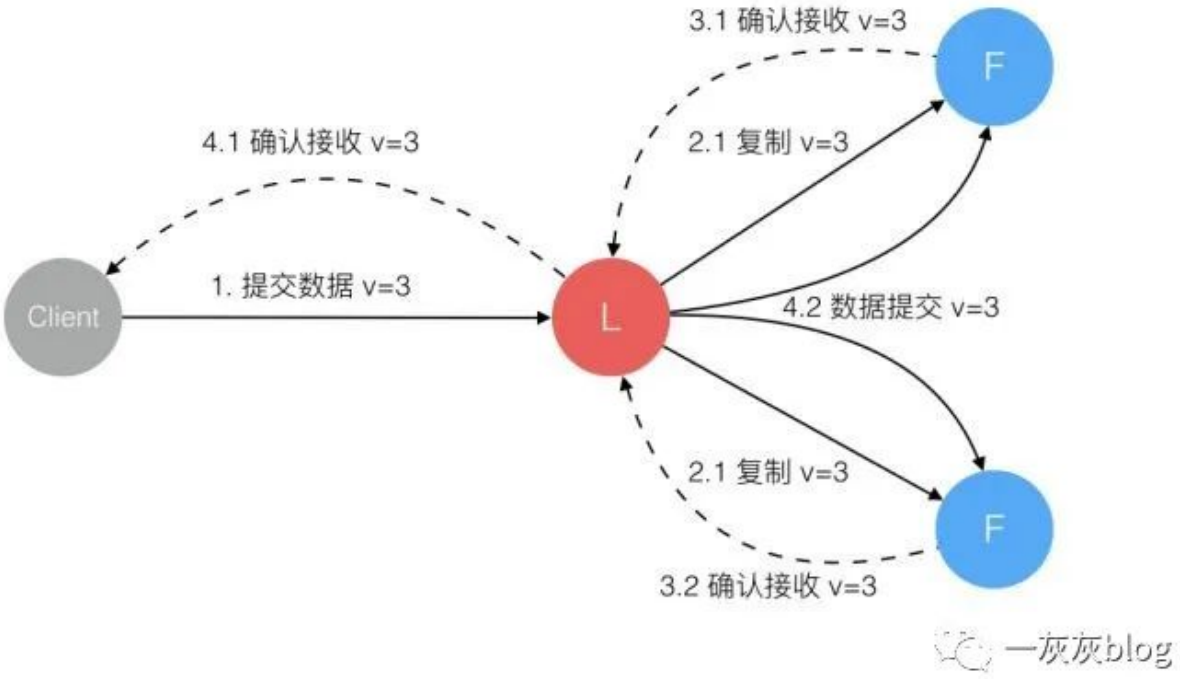
> 推荐有兴趣的小伙伴，查看
>
> - Raft 算法动画演示[6]
> - Raft算法详解 - 知乎[7]

为了解决paxos的复杂性，raft算法提供了一套更易理解的算法基础，其核心流程在于：

leader接受请求，并转发给follow，当大部分follow响应之后，leader通知所有的follow提交请求、同时自己也提交请求并告诉调用方ok

角色划分：

- Leader：领导者，接受客户端请求，并向Follower同步请求，当数据同步到大多数节点上后告诉Follower提交日志
- Follow: 接受并持久化Leader同步的数据，在Leader告之日志可以提交之后，提交
- Candidate：Leader选举过程中的临时角色，向其他节点拉选票，得到多数的晋升为leader，选举完成之后不存在这个角色



raft共识流程

2.6 ZAB协议

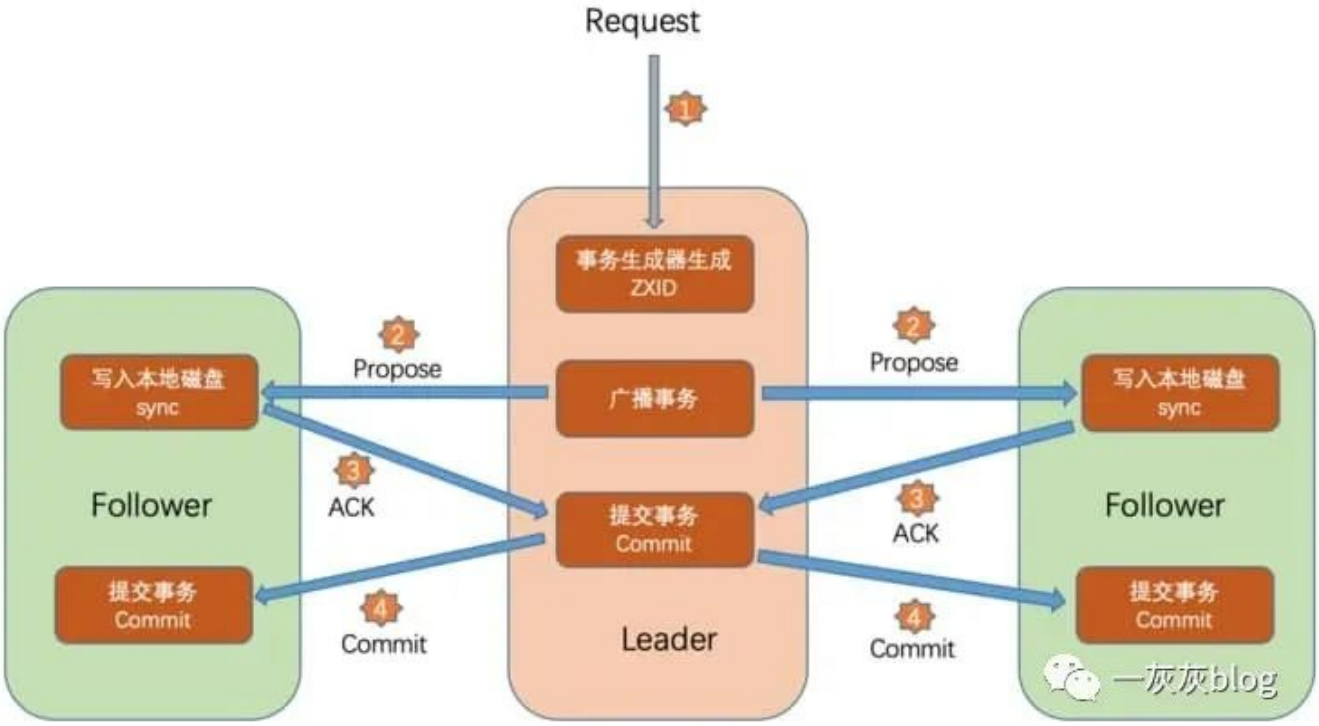
> ZAB(Zookeeper Atomic Broadcast) 协议是为分布式协调服务ZooKeeper专门设计的一种支持崩溃恢复的一致性协议，基于该协议，ZooKeeper 实现了一种 主从模式的系统架构来保持集群中各个副本之间的数据一致性。
>
> - zookeeper核心之ZAB协议就这么简单！[8]

主要用于zk的数据一致性场景，其核心思想是Leader再接受到事务请求之后，通过给Follower，当半数以上的Follower返回ACK之后，Leader提交提案，并向Follower发送commit信息

角色划分

- Leader: 负责整个Zookeeper 集群工作机制中的核心
  - 事务请求的唯一调度和处理者，保证集群事务处理的顺序性
  - 集群内部各服务器的调度者
- Follower：Leader的追随者
  - 处理客户端的非实物请求，转发事务请求给 Leader 服务器
  - 参与事务请求 Proposal 的投票

- 参与 Leader 选举投票
- Observer：是 zookeeper 自 3.3.0 开始引入的一个角色，
  - 它不参与事务请求 Proposal 的投票，
  - 也不参与 Leader 选举投票
  - 只提供非事务的服务（查询），通常在不影响集群事务处理能力的前提下提升集群的非事务处理能力。



ZAB消息广播

2.7 2PC协议

> two-phase commit protocol，两阶段提交协议，主要是为了解决强一致性，中心化的强一致性协议

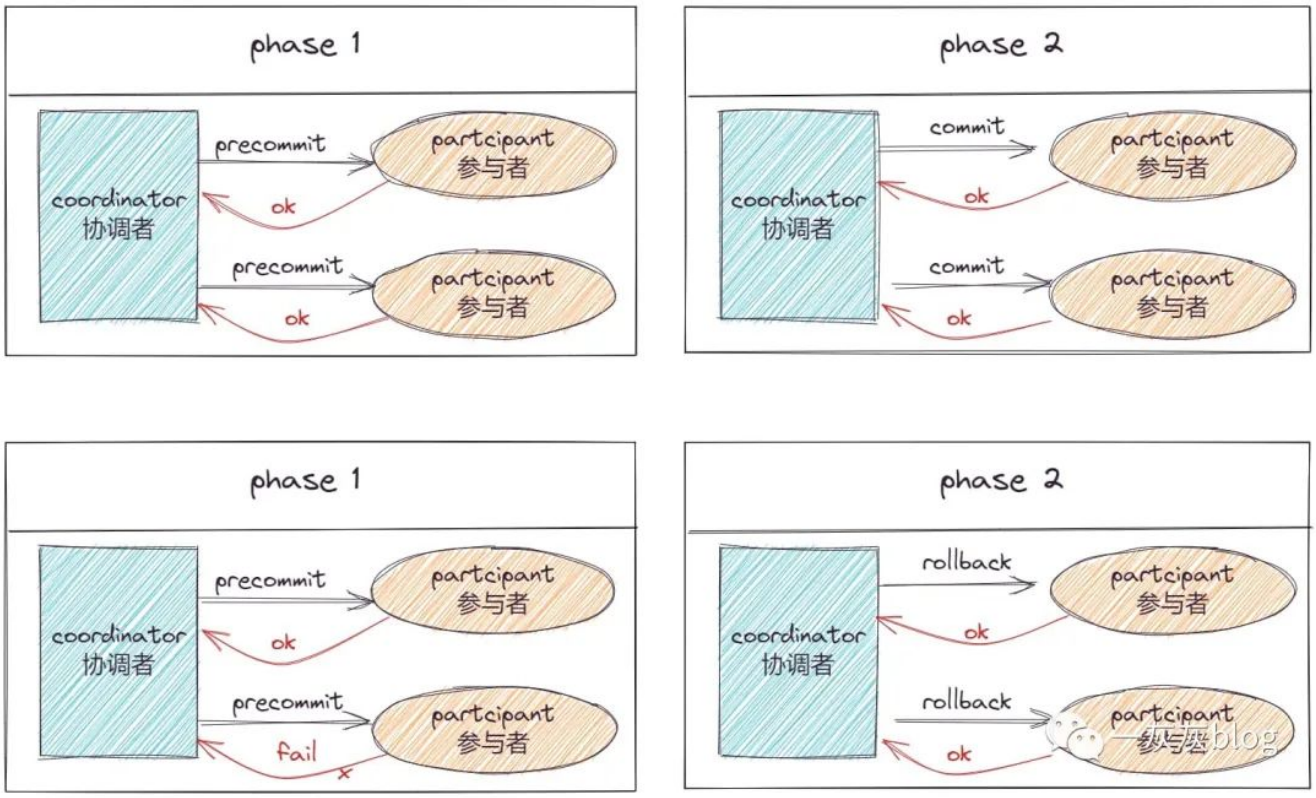
角色划分

- 协调节点(coordinator)：中心化
- 参与者节点(partcipant)：多个

执行流程

协调节点接收请求，然后向参与者节点提交 `precommit` ，当所有的参与者都回复ok之后，协调节点再给所有的参与者节点提交 `commit` ，所有的都返回ok之后，才表明这个数据确认提交

当第一个阶段，有一个参与者失败，则所有的参与者节点都回滚



2pc流程

特点

优点在于实现简单

缺点也很明显

- 协调节点的单点故障
- 第一阶段全部ack正常，第二阶段存在部分参与者节点异常时，可能出现不一致问题

2.8 3PC协议

> 分布式事务：两阶段提交与三阶段提交 - SegmentFault 思否[9]

在两阶段的基础上进行扩展，将第一阶段划分两部，cancommit + precommit，第三阶段则为 docommit

第一阶段 cancommit

该阶段协调者会去询问各个参与者是否能够正常执行事务，参与者根据自身情况回复一个预估值，相对于真正的执行事务，这个过程是轻量的

第二阶段 precommit

本阶段协调者会根据第一阶段的询盘结果采取相应操作，若所有参与者都返回ok，则协调者向参与者提交事务执行(单不提交)通知；否则通知参与者abort回滚

第三阶段 docommit

如果第二阶段事务未中断，那么本阶段协调者将会依据事务执行返回的结果来决定提交或回滚事务，若所有参与者正常执行，则提交；否则协调者+参与者回滚

在本阶段如果因为协调者或网络问题，导致参与者迟迟不能收到来自协调者的 commit 或 rollback 请求，那么参与者将不会如两阶段提交中那样陷入阻塞，而是等待超时后继续 commit，相对于两阶段提交虽然降低了同步阻塞，但仍然无法完全避免数据的不一致

特点

- 降低了阻塞与单点故障：
  - 参与者返回 CanCommit 请求的响应后，等待第二阶段指令，若等待超时/协调者宕机，则自动 abort，降低了阻塞；
  - 参与者返回 PreCommit 请求的响应后，等待第三阶段指令，若等待超时/协调者宕机，则自动 commit 事务，也降低了阻塞；
- 数据不一致问题依然存在
  - 比如第三阶段协调者发出了 abort 请求，然后有些参与者没有收到 abort，那么就会自动 commit，造成数据不一致

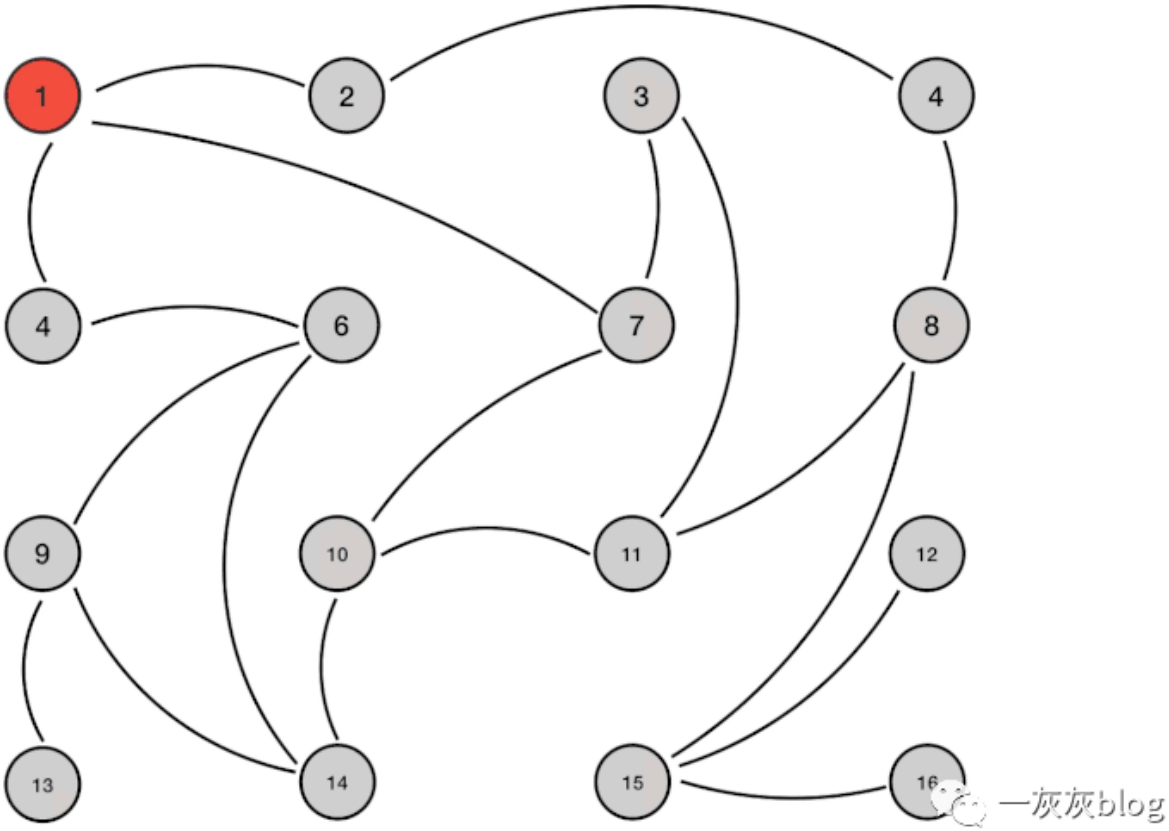
2.9 Gossip协议

> Gossip 协议，顾名思义，就像流言蜚语一样，利用一种随机、带有传染性的方式，将信息传播到整个网络中，并在一定时间内，使得系统内的所有节点数据一致。Gossip 协议通过上面的特性，可以保证系统能在极端情况下（比如集群中只有一个节点在运行）也能运行
>
> - P2P 网络核心技术：Gossip 协议 - 知乎[10]

主要用在分布式数据库系统中各个副本节点同步数据之用，这种场景的一个最大特点就是组成的网络的节点都是对等节点，是非结构化网络

工作流程

- 周期性的传播消息，通常周期时间为1s
- 被感染的节点，随机选择n个相邻节点，传播消息
- 每次传播消息都选择还没有发送过的节点进行传播
- 收单消息的节点，不会传播给向它发送消息的节点



Gossip传播示意图

特点

- 扩展性：允许节点动态增加、减少，新增的节点状态最终会与其他节点一致
- 容错：网络中任意一个节点宕机重启都不会影响消息传播
- 去中心化：不要求中心节点，所有节点对等，任何一个节点无需知道整个网络状况，只要网络连通，则一个节点的消息最终会散播到整个网络
- 一致性收敛：协议中的消息会以一传十、十传百一样的指数级速度在网络中快速传播，因此系统状态的不一致可以在很快的时间内收敛到一致。消息传播速度达到了 logN
- 简单：Gossip 协议的过程极其简单，实现起来几乎没有太多复杂性

缺点

- 消息延迟：节点只会随机向少数几个节点发送消息，消息最终是通过多个轮次的散播而到达全网的，因此使用 Gossip 协议会造成不可避免的消息延迟
- 消息冗余：节点会定期随机选择周围节点发送消息，而收到消息的节点也会重复该步骤，导致消息的冗余

2.10 小结

本节主要介绍的是分布式系统设计中的一些常见的理论基石，如分布式中如何保障一致性，如何对一个提案达成共识

- BASE，CAP，PACELEC理论：构建稳定的分布式系统应该考虑的方向
- paxos,raft共识算法
- zab一致性协议
- gossip消息同步协议

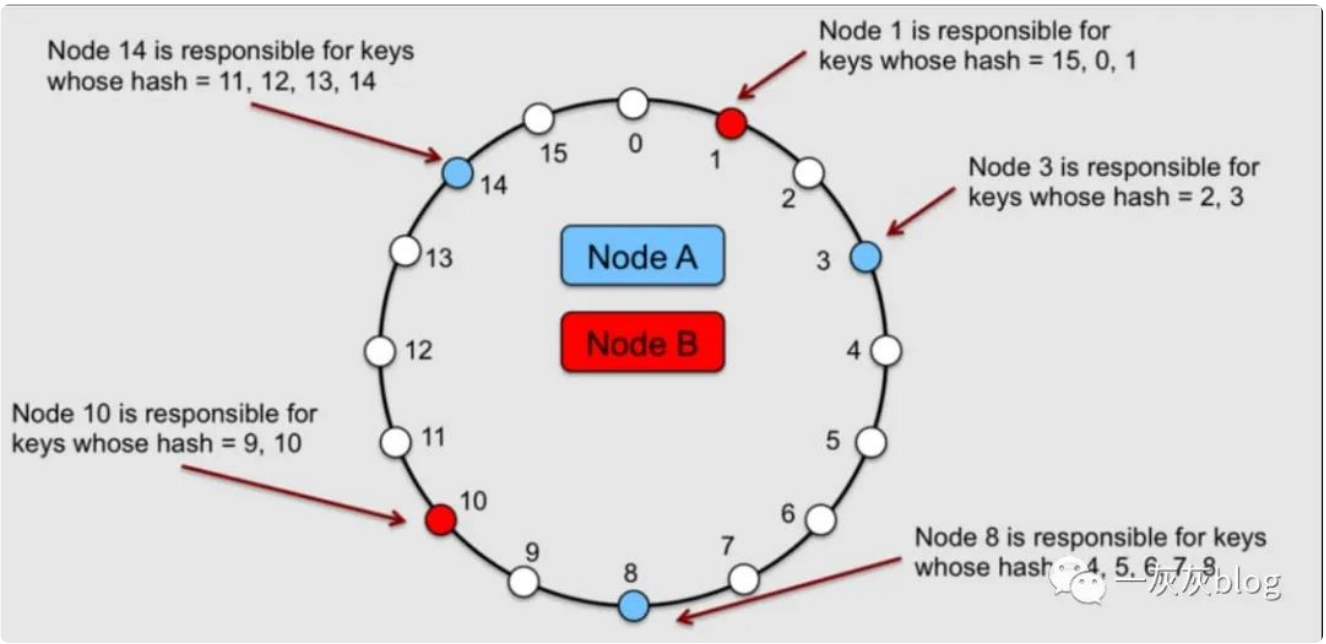
3.算法

这一节将主要介绍下分布式系统中的经典的算法，比如常用于分区的一致性hash算法，适用于一致性的Quorum NWR算法，PBFT拜占庭容错算法，区块链中大量使用的工作量证明PoW算法等

3.1 一致性hash算法

一致性hash算法，主要应用于数据分片场景下，有效降低服务的新增、删除对数据复制的影响

通过对数据项的键进行哈希处理映射其在环上的位置，然后顺时针遍历环以查找位置大于该项位置的第一个节点，将每个由键标识的数据分配给hash环中的一个节点



一致性hash算法

一致散列的主要优点是增量稳定性；节点添加删除，对整个集群而言，仅影响其直接邻居，其他节点不受影响。

注意：

- redis集群实现了一套hash槽机制，其核心思想与一致性hash比较相似

3.2 Quorum NWR算法

> 用来保证数据冗余和最终一致性的投票算法，其主要数学思想来源于鸽巢原理
>
> - 分布式系统之Quorum （NRW）算法-阿里云开发者社区[11]

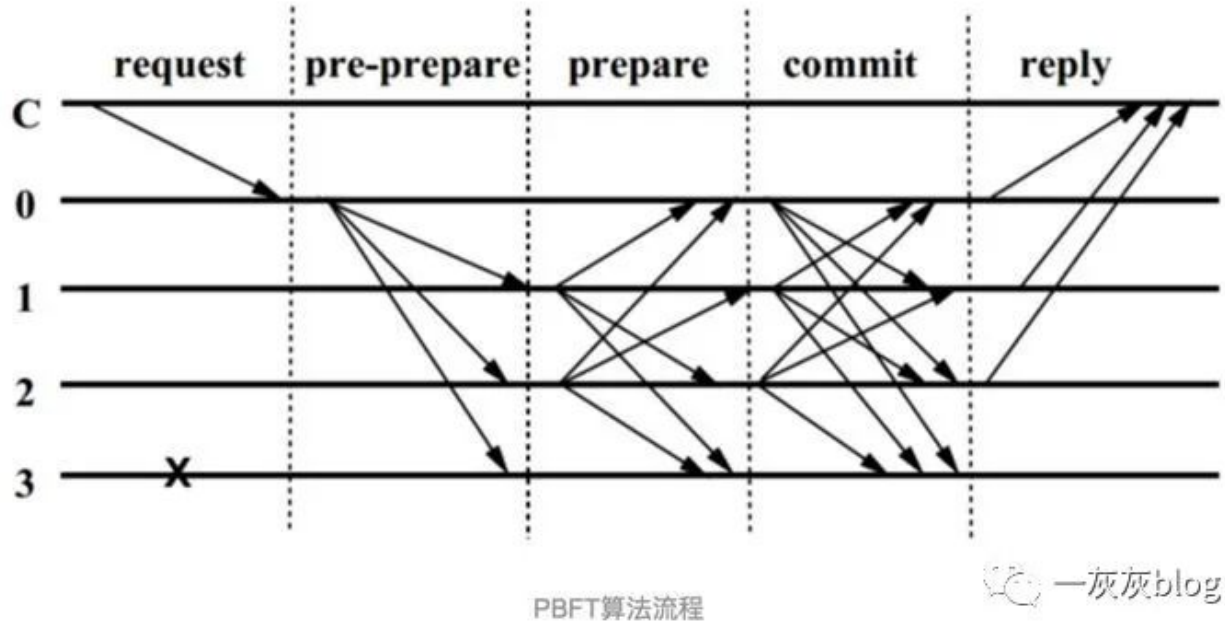
- N 表示副本数，又叫做复制因子（Replication Factor）。也就是说，N 表示集群中同一份数据有多少个副本
- W，又称写一致性级别（Write Consistency Level），表示成功完成 W 个副本更新写入，才会视为本次写操作成功
- R 又称读一致性级别（Read Consistency Level），表示读取一个数据对象时需要读 R 个副本, 才会视为本次读操作成功

Quorum NWR算法要求每个数据拷贝对象 都可以投1票，而每一个操作的执行则需要获取最小的读票数，写票数；通常来讲写票数W一般需要超过N/2，即我们通常说的得到半数以上的票才表示数据写入成功

事实上当W=N、R=1时，即所谓的WARO(Write All Read One)。就是CAP理论中CP模型的场景

3.3 PBFT拜占庭算法

拜占庭算法主要针对的是分布式场景下无响应，或者响应不可信的情况下的容错问题，其核心分三段流程，如下



拜占庭算法

假设集群节点数为 N，f个故障节点(无响应)和f个问题节点(无响应或错误响应),f+1个正常节点，即 3f+1=n

- 客户端向主节点发起请求，主节点接受请求之后，向其他节点广播 pre-prepare 消息
- 节点接受pre-prepare消息之后，若同意请求，则向其他节点广播 prepare 消息；
- 当一个节点接受到2f+1个prepare新消息，则进入commit阶段，并广播commit消息
- 当收到 2f+1 个 commit 消息后（包括自己），代表大多数节点已经进入 commit 阶段，这一阶段已经达成共识，于是节点就会执行请求，写入数据

相比 Raft 算法完全不适应有人作恶的场景，PBFT 算法能容忍 (n 1)/3 个恶意节点 (也可以是故障节点)。另外，相比 PoW 算法，PBFT 的优点是不消耗算 力。PBFT 算法是O(n ^ 2) 的消息复杂度的算法，所以以及随着消息数 的增加，网络时延对系统运行的影响也会越大，这些都限制了运行 PBFT 算法的分布式系统 的规模，也决定了 PBFT 算法适用于中小型分布式系统

3.4 PoW算法

工作量证明 (Proof Of Work，简称 PoW)，同样应用于分布式下的一致性场景，区别于前面的raft, pbft, paxos采用投票机制达成共识方案，pow采用工作量证明

客户端需要做一定难度的工作才能得出一个结果，验证方却很容易通过结果来检查出客户端是不是做了相应的工作，通过消耗一定工作浪，增加消息伪造的成本，PoW以区块链中广泛应用而广为人知，下面以区块链来简单说一下PoW的算法应用场景

以BTC的转账为例，A转n个btc给B，如何保证不会同时将这n个币转给C?

- A转账给B，交易信息记录在一个区块1中
- A转账给C，交易信息被记录在另一个区块2中
- 当区块1被矿工成功提交到链上，并被大多数认可（通过校验区块链上的hash值验证是否准确，而这个hash值体现的是矿工的工作量），此时尚未提交的区块2则会被抛弃
- 若区块1被提交，区块2也被提交，各自有部分人认可，就会导致分叉，区块链中采用的是优选最长的链作为主链，丢弃分叉的部分（这就属于区块链的知识点了，有兴趣的小伙伴可以扩展下相关知识点，这里就不展开了）

PoW的算法，主要应用在上面的区块提交验证，通过hash值计算来消耗算力，以此证明矿工确实有付出，得到多数认可的可以达成共识

3.5 小结

本节主要介绍了下当前分布式下常见的算法，

- 分区的一致性hash算法: 基于hash环，减少节点动态增加减少对整个集群的影响；适用于数据分片的场景
- 适用于一致性的Quorum NWR算法: 投票算法，定义如何就一个提案达成共识
- PBFT拜占庭容错算法: 适用于集群中节点故障、或者不可信的场景
- 区块链中大量使用的工作量证明PoW算法: 通过工作量证明，认可节点的提交

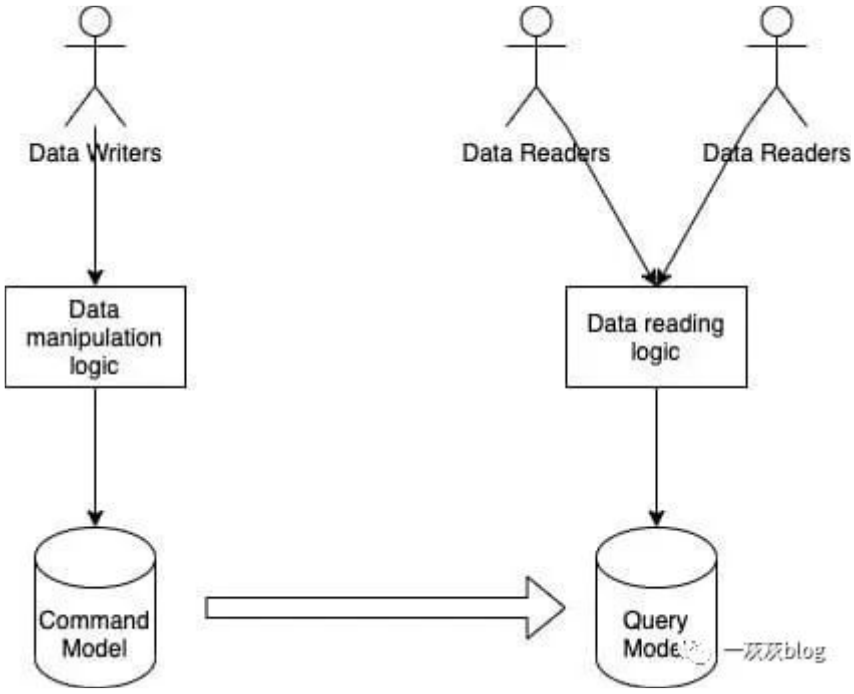
4.技术思想

这一节的内容相对前面几个而言，并不太容易进行清晰的分类；主要包含一些高质量的分布式系统的实践中，值得推荐的设计思想、技术细节

4.1 CQRS

> - DDD 中的那些模式 — CQRS - 知乎[12]
> - 详解CQRS架构模式_架构_Kislay Verma_InfoQ精选文章[13]

Command Query Responsibility Segregation 即我们通俗理解的读写分离，其核心思想在于将两类不同操作进行分离，在独立的服务中实现



cqrs

用途在于将领域模型与查询功能进行分离，让一些复杂的查询摆脱领域模型的限制，以更为简单的 DTO 形式展现查询结果。同时分离了不同的数据存储结构，让开发者按照查询的功能与要求更加自由的选择数据存储引擎

4.2 复制负载平衡服务

- 分布式系统设计:服务模式之复制负载平衡服务 - 知乎[14]
- 负载均衡调度算法大全 | 菜鸟教程[15]

复制负载平衡服务(Replication Load Balancing Service, RLBS)，可以简单理解为我们常说的负载均衡，多个相同的服务实例构建一个集群，每个服务都可以响应请求，负载均衡器负责请求的分发到不同的实例上，常见的负载算法

算法	说明	特点
轮询	请求按照顺序依次分发给对应的服务器	优点简单，缺点在于未考虑不同服务器的实际性能情况
加权轮询	权重高的被分发更多的请求	优点：充分利用机器的性能

算法	说明	特点
最少连接数	找连接数最少的服务器进行请求分发,若所有服务器相同的连接数，则找第一个选择的	目的是让优先让空闲的机器响应请求
少连接数慢启动时间	刚启动的服务器，在一个时间段内，连接数是有限制且缓慢增加	避免刚上线导致大量的请求分发过来而超载
加权最少连接	平衡服务性能 + 最少连接数	
基于代理的自适应负载均衡	载主机包含一个自适用逻辑用来定时监测服务器状态和该服务器的权重	
源地址哈希法	获取客户端的IP地址，通过哈希函映射到对应的服务器	相同的来源请求都转发到相同的服务器上
随机	随机算法选择一台服务器	
固定权重	最高权重只有在其他服务器的权重值都很低时才使用。然而，如果最高权重的服务器下降，则下一个最高优先级的服务器将为客户端服务	每个真实服务器的权重需要基于服务器优先级来配置
加权响应	服务器响应越小其权重越高，通常是基于心跳来判断机器的快慢	心跳的响应并不一定非常准确反应服务情况

4.3 心跳机制

在分布式环境里中，如何判断一个服务是否存活，当下最常见的方案就是心跳

比如raft算法中的leader向所有的follow发送心跳，表示自己还健在，避免发生新的选举；

比如redis的哨兵机制，也是通过ping/pong的心跳来判断节点是否下线，是否需要选新的主节点；

再比如我们日常的业务应用得健康监测，判断服务是否正常

4.4 租约机制

租约就像一个锁，但即使客户端离开，它也能工作。客户端请求有限期限的租约，之后租约到期。如果客户端想要延长租约，它可以在租约到期之前续订租约。

租约主要是了避免一个资源长久被某个对象持有，一旦对方挂了且不会主动释放的问题；在实际的场景中，有两个典型的应用

case1 分布式锁

业务获取的分布式锁一般都有一个有效期，若有效期内没有主动释放，这个锁依然会被释放掉，其他业务也可以抢占到这把锁；因此对于持有锁的业务方而言，若发现在到期前，业务逻辑还没有处理完，则可以续约，让自己继续持有这把锁

典型的实现方式是redisson的看门狗机制

case2 raft算法的任期

在raft算法中，每个leader都有一个任期，任期过后会重新选举，而Leader为了避免重新选举，一般会定时发送心跳到Follower进行续约

4.5 Leader & Follow

这个比较好理解，上面很多系统都采用了这种方案，特别是在共识算法中，由领导者负责代表整个集群做出决策，并将决策传播到所有其他服务器

领导者选举在服务器启动时进行。每个服务器在启动时都会启动领导者选举，并尝试选举领导者。除非选出领导者，否则系统不接受任何客户端请求

4.6 Fencing

在领导者-追随者模式中，当领导者失败时，不可能确定领导者已停止工作，如慢速网络或网络分区可能会触发新的领导者选举，即使前一个领导者仍在运行并认为它仍然是活动的领导者

Fencint是指在以前处于活动状态的领导者周围设置围栏，使其无法访问集群资源，从而停止为任何读/写请求提供服务

- 资源屏蔽：系统会阻止以前处于活动状态的领导者访问执行基本任务所需的资源。
- 节点屏蔽：系统会阻止以前处于活动状态的领导者访问所有资源。执行此操作的常见方法是关闭节点电源或重置节点。

4.7 Quorum法定人数

法定人数，常见于选举、共识算法中，当超过Quorum的节点数确认之后，才表示这个提案通过(数据更新成功)，通常这个法定人数为 = 半数节点 + 1

4.8 High-Water mark高水位线

高水位线，跟踪Leader（领导者）上的最后一个日志条目，且该条目已成功复制到>quorum（法定人数）的Follow（跟谁者），即表示这个日志被整个集群接受

日志中此条目的索引称为高水位线索引。领导者仅公开到高水位线索引的数据。

如Kafka：为了处理非可重复读取并确保数据一致性，Kafka broker会跟踪高水位线，这是特定分区的最大偏移量。使用者只能看到高水位线之前的消息。

4.9 Phi 累计故障检测

Phi Accrual Failure Detection,使用历史检测信号信息使阈值自适应

通用的应计故障检测器不会判断服务器是否处于活动状态，而是输出有关服务器的可疑级别。

如Cassandra（Facebook开源的分布式NoSql数据库）使用 Phi 应计故障检测器算法来确定群集中节点的状态

4.10 Write-ahead Log预写日志

预写日志记录是解决操作系统中文件系统不一致的问题的高级解决方案，当我们提交写到操作系统的文件缓存，此时业务会认为已经提交成功；但是在文件缓存与实际写盘之间会有一个时间差，若此时机器宕机，会导致缓存中的数据丢失，从而导致完整性缺失

为了解决这个问题，如mysql，es等都采用了预写日志的机制来避免这个问题

MySql：

- 事务提交的流程中，先写redolog precommit， 然后写binlog，最后再redolog commit；当redolog记录成功之后，才表示事务执行成功；
- 因此当出现上面的宕机恢复时，则会加载redologo，然后重放对应的命令，来恢复未持久化的数据

ElasticSearch:

- 在内存中数据生成段写到操作系统文件缓存前，会先写事务日志，出现异常时，也是从事务日志进行恢复

4.11 分段日志

将日志拆分为多个较小的文件，而不是单个大文件，以便于操作。

单个日志文件在启动时读取时可能会增长并成为性能瓶颈。较旧的日志会定期清理，并且很难对单个大文件执行清理操作。

单个日志拆分为多个段。日志文件在指定的大小限制后滚动。使用日志分段，需要有一种将逻辑日志偏移量（或日志序列号）映射到日志段文件的简单方法。

这个其实也非常常见，比如我们实际业务应用配置的log，一般都是按天、固定大小进行拆分，并不会把所有的日志都放在一个日志文件中

再比如es的分段存储，一个段就是一个小的存储文件

4.12 checksum校验

在分布式系统中，在组件之间移动数据时，从节点获取的数据可能会损坏。

计算校验和并将其与数据一起存储。

要计算校验和，请使用 MD5、SHA-1、SHA-256 或 SHA-512 等加密哈希函数。哈希函数获取输入数据并生成固定长度的字符串（包含字母和数字）;此字符串称为校验和。

当系统存储某些数据时，它会计算数据的校验和，并将校验和与数据一起存储。当客户端检索数据时，它会验证从服务器接收的数据是否与存储的校验和匹配。如果没有，则客户端可以选择从另一个副本检索该数据。

HDFS和Chubby将每个文件的校验和与数据一起存储。

4.13 小结

这一节很多内容来自下面这篇博文，推荐有兴趣的小伙伴查看原文

- Distributed System Design Patterns | by Nishant | Medium[16]

这一节主要简单的介绍了下分布式系统中应用到的一些技术方案，如有对其中某个技术有兴趣的小伙伴可以留言，后续会逐一进行补全

5.分布式系统解决方案

最后再介绍一些常见的分布式业务场景及对应的解决方案，比如全局唯一的递增ID-雪花算法，分布式系统的资源抢占-分布式锁，分布式事务-2pc/3pc/tcc ，分布式缓存等

5.1 缓存

缓存实际上并不是分布式独有的，这里把它加进来，主要是因为实在是应用得太广了，无论是应用服务、基础软件工具还是操作系统，大量都可以见到缓存的身影

缓存的核心思想在于：借助更高效的IO方式，来替代代价昂贵的IO方式

如：

- redis的性能高于mysql
- 如内存的读写，远高于磁盘IO，文件IO
- 磁盘顺序读写 > 随机读写

用好缓存可以有效提高应用性能，下面以一个普通的java前台应用为例说明

- JVM缓存 -> 分布式缓存(redis/memcache) -> mysql缓存 -> 操作系统文件缓存 -> 磁盘文件

缓存面临的核心问题，则在于

- 一致性问题：缓存与db的一致性如何保障（相信大家都听说过或者实际处理过这种问题）
- 数据完整性：比如常见的先写缓存，异步刷新到磁盘，那么缓存到磁盘刷新这段时间内，若宕机导致数据丢失怎么办?
  - TIP: 上面这个问题可以参考mysql的redolog

5.2 全局唯一ID

在传统的单体架构中，业务id基本上是依赖于数据库的自增id来处理；当我们进入分布式场景时，如我们常说的分库分表时，就需要我们来考虑如何实现全局唯一的业务id了，避免出现在分表中出现冲突

全局唯一ID解决方案：

- uuid
- 数据库自增id表
- redis原子自增命令
- 雪花算法 (原生的，扩展的百度UidGenerator, 美团Leaf等)
- Mist 薄雾算法

5.3 分布式锁

常用于分布式系统中资源控制，只有持有锁的才能继续操作，确保同一时刻只会有一个实例访问这个资源

常见的分布式锁有

- 基于数据库实现分布式锁
- Redis实现分布式锁（应用篇） | 一灰灰Learning[17]
- 从0到1实现一个分布式锁 | 一灰灰Learning[18]
- etcd实现分布式锁
- 基于consul实现分布式锁

5.4 分布式事务

事务表示一组操作，要么全部成功，要么全部不成功；单机事务通常说的是数据库的事务；而分布式事务，则可以简单理解为多个数据库的操作，要么同时成功，要么全部不成功

更确切一点的说法，分布式事务主要是要求事务的参与方，可能涉及到多个系统、多个数据资源，要求它们的操作要么都成功，要么都回滚；

一个简单的例子描述下分布式事务场景：

下单扣库存

- 用户下单，付钱
- 此时订单服务，会生成订单信息
- 支付网关，会记录付款信息，成功or失败
- 库存服务，扣减对应的库存

一个下单支付操作，涉及到三个系统，而分布式事务则是要求，若支付成功，则上面三个系统都应该更新成功；若有一个操作失败，如支付失败，则已经扣了库存的要回滚（还库存），生成的订单信息回滚（删掉--注：现实中并不会去删除订单信息，这里只是用于说明分布式事务，请勿带入实际的实现方案）

分布式事务实现方案：

- 2PC: 前面说的两阶段提交，就是实现分布式事务的一个经典解决方案
- 3PC: 三阶段提交
- TCC：补偿事务，简单理解为应用层面的2PC
- SAGA事务
- 本地消息表
- MQ事务方案

5.5 分布式任务

分布式任务相比于我们常说单机的定时任务而言，可以简单的理解为多台实例上的定时任务，从应用场景来说，可以区分两种

- 互斥性的分布式任务
  - 即同一时刻，集群内只能有一个实例执行这个任务
- 并存式的分布式任务
  - 同一时刻，所有的实例都可以执行这个任务
  - 续考虑如何避免多个任务操作相同的资源

分布式任务实现方案：

- Quartz Cluster
- XXL-Job
- Elastic-Job

- 自研：
  - 资源分片策略
  - 分布式锁控制的唯一任务执行策略

5.6 分布式Session

> Session一般叫做会话，Session技术是http状态保持在服务端的解决方案，它是通过服务器来保持状态的。我们可以把客户端浏览器与服务器之间一系列交互的动作称为一个 Session。是服务器端为客户端所开辟的存储空间，在其中保存的信息就是用于保持状态。因此，session是解决http协议无状态问题的服务端解决方案，它能让客户端和服务端一系列交互动作变成一个完整的事务。

单机基于session/cookie来实现用户认证，那么在分布式系统的多实例之间，如何验证用户身份呢？这个就是我们说的分布式session

分布式session实现方案：

- session stick：客户端每次请求都转发到同一台服务器(如基于ip的hash路由转发策略)
- session复制: session生成之后，主动同步给其他服务器
- session集中保存：用户信息统一存储，每次需要时统一从这里取(也就是常说的redis实现分布式session方案)
- cookie: 使用客户端cookie存储session数据，每次请求时携带这个

5.7 分布式链路追踪

分布式链路追踪也可以叫做全链路追中，而它可以说是每个开发者的福音，通常指的是一次前端的请求，将这个请求过程中，所有涉及到的系统、链路都串联起来，可以清晰的知道这一次请求中，调用了哪些服务，有哪些IO交互，瓶颈点在哪里，什么地方抛出了异常

当前主流的全链路方案大多是基于google的 `Dapper` 论文实现的

全链路实现方案

- zipkin
- pinpoint
- SkyWalking
- CAT
- jaeger

5.8 布隆过滤器

Bloom过滤器是一种节省空间的概率数据结构，用于测试元素是否为某集合的成员。

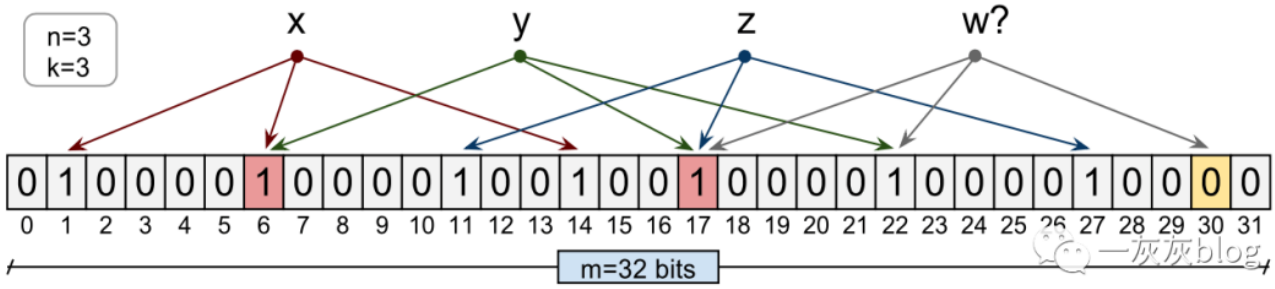
布隆过滤器由一个长度为 m 比特的位数组（bit array）与 k 个哈希函数（hash function）组成的数据结构。

原理是当一个元素被加入集合时，通过 K 个散列函数将这个元素映射成一个位数组中的 K 个点，把它们置为 1。

检索时，我们只要看看这些点是不是都是 1 就大约知道集合中有没有它了，也就是说，如果这些点有任何一个 0 ，则被检元素一定不在；如果都是 1 ，则被检元素很可能在。

关于布隆过滤器，请牢记一点

- 判定命中的，不一定真的命中
- 判定没有命中的，则一定不在里面



布隆过滤器

常见的应用场景，如

- 防止缓存穿透
- 爬虫时重复检测

5.9 一灰灰的小结

分布式系统的解决方案当然不局限于上面几种，比如分布式存储、分布式计算等也属于常见的场景，当然在我们实际的业务支持过程中，不太可能需要让我们自己来支撑这种大活；而上面提到的几个点，基本上或多或少会与我们日常工作相关，这里列出来当然是好为了后续的详情做铺垫

6.总结

6.1 综述

这是一篇概括性的综述类文章，可能并没有很多的干货，当然也限于“一灰灰”我个人的能力，上面的总结可能并不准确，如有发现，请不吝赐教

全文总结如下

常见的分布式架构设计方案：

- 主备，主从，多主多从，普通无中心集群，数据分片架构

分布式系统中的理论基石：

- CAP， BASE, PACELEC
- 共识算法：paxos, raft, zab
- 一致性协议：2pc, 3pc
- 数据同步：gossip

分布式系统中的算法：

- 分区的一致性hash算法: 基于hash环，减少节点动态增加减少对整个集群的影响；适用于数据分片的场景
- 适用于一致性的Quorum NWR算法: 投票算法，定义如何就一个提案达成共识
- PBFT拜占庭容错算法: 适用于集群中节点故障、或者不可信的场景
- 区块链中大量使用的工作量证明PoW算法: 通过工作量证明，认可节点的提交

分布式系统解决方案：

- 分布式缓存
- 全局唯一ID
- 分布式锁
- 分布式事务
- 分布式任务
- 分布式会话
- 分布式链路追踪
- 布隆过滤器

References

[1] 一致性算法-Gossip协议详解 - 腾讯云开发者社区-腾讯云: *https://cloud.tencent.com/developer/article/1662426*
[2] P2P 网络核心技术：Gossip 协议 - 知乎: *https://zhuanlan.zhihu.com/p/41228196*
[3] 从Paxos到Raft，分布式一致性算法解析_mb5fdb0a87e2fa1的技术博客_51CTO博客: *https://blog.51cto.com/u_15060467/2678779*
[4] 【理论篇】浅析分布式中的 CAP、BASE、2PC、3PC、Paxos、Raft、ZAB - 知乎: *https://zhuanlan.zhihu.com/p/338628717*
[5] Distributed System Design Patterns | by Nishant | Medium: *https://medium.com/@nishantparmar/distributed-system-design-patterns-2d20908fecfc*
[6] Raft 算法动画演示: *http://thesecretlivesofdata.com/raft/*
[7] Raft算法详解 - 知乎: *https://zhuanlan.zhihu.com/p/32052223*
[8] zookeeper核心之ZAB协议就这么简单！: *https://segmentfault.com/a/1190000037550497*
[9] 分布式事务：两阶段提交与三阶段提交 - SegmentFault 思否: *https://segmentfault.com/a/1190000012534071*
[10] P2P 网络核心技术：Gossip 协议 - 知乎: *https://zhuanlan.zhihu.com/p/41228196*
[11] 分布式系统之Quorum （NRW）算法-阿里云开发者社区: *https://developer.aliyun.com/article/53498*
[12] DDD 中的那些模式 — CQRS - 知乎: *https://zhuanlan.zhihu.com/p/115685384*
[13] 详解CQRS架构模式_架构_Kislay Verma_InfoQ精选文章: *https://www.infoq.cn/article/wdlpjosudoga34jutys9*
[14] 分布式系统设计:服务模式之复制负载平衡服务 - 知乎: *https://zhuanlan.zhihu.com/p/34191846*
[15] 负载均衡调度算法大全 | 菜鸟教程: *https://www.runoob.com/w3cnote/balanced-algorithm.html*
[16] Distributed System Design Patterns | by Nishant | Medium: *https://medium.com/@nishantparmar/distributed-system-design-patterns-2d20908fecfc*
[17] Redis实现分布式锁（应用篇） | 一灰灰Learning: *https://hhui.top/spring-db/09.%E5%AE%9E%E4%BE%8B/20.201030-springboot%E7%B3%BB%E5%88%97%E6%95%99%E7%A8%8Bredis%E5%AE%9E%E7%8E%B0%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81%E5%BA%94%E7%94%A8%E7%AF%87/*
[18] 从0到1实现一个分布式锁 | 一灰灰Learning: *https://hhui.top/spring-middle/03.zookeeper/02.210415-springboot%E6%95%B4%E5%90%88zookeeper%E4%BB%8E0%E5%88%B01%E5%AE%9E%E7%8E%B0%E4%B8%80%E4%B8%AA%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81/*

往期推荐

Astro 1.0发布，现代化静态站点生成器

Redis回击Dragonfly：13年后，Redis的架构依然是同类最佳

建议退役！JSON之父：JavaScript已阻碍进步

这里有最新开源资讯、软件更新、技术干货等内容

点这里 ↓↓↓ 记得 关注✔ 标星⭐ 哦~

OSC开源社区

开源中国，为开发者服务

2250篇原创内容

公众号

阅读原文

喜欢此内容的人还喜欢

如何优雅做好项目管理?

OSC开源社区

为何黑客很少攻击赌博网站?

黑客IT

学习Node.js框架和库：构建高效Web应用的利器

码农啊彪