分布式基础

学习目的

> 了解分布式理论，然后对各类分布式框架的特性和适用场景有全面的掌握，最后结合实际应用场景落地

集中式

是把所有的功能集中到一台主机上，对外提供服务。

> 优点：便于维护、操作简单
>
> 缺点：单点故障；单机性能瓶颈，横向扩展困难

分布式

> 分布式系统是若干独立计算机的集合，这些计算机对于用户来说就像是单个相关系统。

通过网络通信、为了完成共同任务而协调工作的计算机节点（Node）组成的系统，分布式系统的设计目标

> 解决集中式系统的不足，实现整个系统的高性能、高可用、可扩展

存在的问题

> 数据一致性、节点间通讯方式、节点可用性等

由此，诞生了如CAP、BASE、一致性Hash等基本理论。

分布式一致性

CAP

- 背景

  为提高可用性，必然会引入多节点做冗余，如何保证各个节点之间数据的一致性?
- 定义

  Robert Greiner的描述：

  > 在一个分布式系统（指互相连接并共享数据的节点的集合）中，当涉及读写操作时，只能保证一致性（Consistence）、可用性（Availability）、分区容错性（Partition Tolerance）三者中的两个，另一个必须被牺牲

  - **一致性 (C)**

    > 在写操作发生之后，对某个指定的客户端，读操作必须返回最新的写操作结果

    注：此一致性并不是某个时间点整个系统所有节点的数据一致性【最终一致性】，而是【瞬时局部一致性】
  - **可用性 (A)**

    > 非故障节点，接收到请求就必须返回合理的结果【其实就是允许部分不可用】

注：1）节点必须是正常的 2）必须响应，即：不是超时和错误 3）合理即：在一定场景下具有合理性，不是严格的正确

- **分区容错性 (P)**

  大多数分布式系统分布在多个子网络，每个子网络可称为一个‘分区’

  > 要求子网络间通信失败后，仍能提供服务

  一般来说，分布式环境中，节点之间依靠网络通信，而网络环境不可能100%可靠，所以一定存在分区问题；当发生‘分区’错误时，如果系统不能继续提供服务，将失去分布式的基本意义，所以说，分布式系统中P总应当被满足。

- CA/CP/AP选择

1. CA

   前部对**P**的解释中，已阐明分布式环境中P总需要被满足，所以理论上CAP一般只能取CP或AP，而CA只存在于集中式应用中。

2. C和A的矛盾

   假如有2个节点：N1、N2，共享数据V

   > **CP**
   >
   > 客户端向N1发起写入请求（V0->V1），当向N2发起读请求时，如果要保证数据一致性(读到V1)，那么N1必须在处理写操作时，同时锁定N2的读、写操作，在数据同步后，才可重新放开N2的读写，锁定期间N2不可用（比如返回error信息，告知客户端‘服务暂不可用’），即：可用性不满足

   > **AP**
   >
   > 如果要保证N2可用，N2返回的值可能是还是V0，那一致性就不能保证；此时V0虽不是一个‘正确’的值，但牺牲一致性的前提下，仍是‘合理’的值，并不是错误

   所以，C和A只可选一；实际应用中，多数场景并不需要很强的实时一致性

BASE

- 思想

Dan Pritchett 提出了BASE理论

> 即使无法做到强一致性（Strong Consistency，CAP 的一致性就是强一致性），但分布式系统可以采用适合的方式达到最终一致性（Eventual Consitency）

- 定义

BASE理论是对CAP中**一致性（C）**和**可用性（P）**权衡的结果

1. **基本可用（ Basically Available）**

   在BASE理论中，*基本可用* 概念与CAP中类似，指在出现故障时，允许损失部分可用性，但不等价于系统不可用。如：

   - 响应时间的损失：在故障时刻，响应时间可能会延长
   - 系统功能的损失：在故障时刻，采取限流或降级手段

2. **软状态（ Soft-state ）**

   指允许系统存在中间状态，而此状态不影响整体的可用性。这里中间状态相当于CAP中的数据不一致

   如：

   - 分布式节点间需要数据同步（存储、元数据...），此过程中存在的延时，即是软状态

3. **最终一致性（Eventually Consistent）**

   指分布式系统无法做到强一致性，但应在一定时限内使各个节点数据最终能够达到一致性的状态，这个时限取决于：

   - 网络延迟
   - 系统负载
   - 方案设计
   - ...

- 总结

BASE理论本质是对CAP理论的补充和延伸，具体来说，是对AP方案（牺牲一致性）的补充：

1. CAP理论是忽略分布式环境必然存在的网络延时，BASE提出最终一致性
2. CAP理论牺牲一致性，是瞬时的，而不是永久放弃一致性，BASE对这一点做了延伸，即在一定时限内，系统应达到最终一致性

Paxos

基于消息传递，解决分布式系统一致性的算法。它要解决的问题是

> **在一个可能发生机器宕机、网络异常的环境下，如何让分布式系统中的所有节点快速准确的对某个数据值达成一致，且不会破坏整个系统的一致性。**

半小时学会什么是分布式一致性算法——Paxos

Raft

通过**选举**达成一致性。最终一致性、去中心化、高可用。共识算法。工作原理概括为

> Raft选举出Leader，Leader全负责 replicated log的管理，负责接收所有客户端的数据请求，然后复制到Follower节点，并在"安全"的状态下执行这些请求；如果Leader发生故障，Followers会重新选举出Leader

- 角色（状态）
  - **Leader 领导者**

    主要负责处理客户端请求，进行日志复制等操作，每一轮选举的目的就是选出一个Leader
  - **Candidate 候选者**

    负责发送拉票请求，当某个节点得票数超过N/2+1（过半），就成为Leader
  - **Follower 选民**

    协议刚开始时，所有节点都是Follower
- 过程
  - **Leader Election 选主**

  1. 初始启动时，所有节点状态都是Follower，并设定一个election timeout（150ms~300ms随机）；如果这段时间内没有收到来自Leader的心跳，节点就发起选举，将自己状态切换为Candidate，向集群其它节点发送请求，询问是否选举自己为Leader，如果该节点还未将自己切换为Candidate，并且没有投票给其它，则接受投票

     注：***多可能性的详解见后续章节***

     约束

     > 任一个term内，只能投决一票，先达先处理
     >
     > Candidate节点的信息不能比投票的Follower旧（比较log index，term）

2. 当收到集群中过半数的接受投票后，节点成为Leader，所有系统的变化都经由此Leader

- **Log Replication 日志复制**

1. 当节点成为Leader后，接收客户端修改请求时，Leader会先将数据写入本地日志，数据是**Uncommitted**状态，Leader向其它节点发起AppendEntries请求，数据在Follower上没有冲突，则写入本地日志，状态也是**Uncommitted**，返回给Leader OK
2. Leader收到过半成功后，将数据状态改为 Committed，并返回给客户端

> 如果不过半，Leader数据状态仍是**Uncommitted**，返回给客户端"错误"。

3. Leader再次给Follower节点发送AppendEntries请求，Follower将本地日志数据状态改为Committed，完成复制过程，数据达成一致状态

约束

> log被复制到过半节点，即是 committed ，保证不会回滚
>
> leader一定包含最新的committed log，因此leader只追加，不删除覆盖
>
> 不同节点，如果某个位置日志相同，则这个位置之前所有日志一定相同

- 复杂选主

在选举阶段有2个超时设置：

> election timeout：Follower---->Candidate，randomized to be between 150ms and 300ms
>
> heartbeat timeout：Leader向Follower发起心跳的间隔时间

- **正常选主**

1. 所有节点初始状态都是Follower，各自有随机不同的election timeout
2. 某一个节点倒计时结束后，这个节点状态变为Candidate，并向其它节点发送选举请求（Request Vote）

> 暂且不考虑多个节点同时结束倒计时，同时成为Candidate

3. 其它节点接收到请求后，若还未将自己切换为Candidate，并且没有投票给其它Candidate，则返回成功，并**重置**自己的election timeout

> 同样不考虑已投票给其它节点

4. Candidate节点收到过半成功请求后，自己成为Leader
5. 完成选举后，Leader向Follower发送*Append Entries*消息，消息是一个特定的间隔时间（heartbeat timeout），Follower响应每个消息，Leader和Follower重置 heartbeat timeout（个人理解），重复此过程

> 直到某个Follower收不到心跳而成为Candidate，将重新选举，即：lost Leader，re-election

问题：1）heartbeat timeout是只在Leader上吗？2）如果Follower收不到请求，需要重新选举，则推论，Follower应该也有heartbeat timeout，否则怎么感知到Leader心跳超时，那么也就是说每个角色都有两个超时时间

- **Leader故障选主**

重点问题在于，选主后，老的Leader复活后怎么处理？

在每一轮选举中，都是有term记录的，复活的原Leader term<当前 term，则原Leader降级为Follower

- **多个Candidate**

  以4个节点描述过程

1. 假如2个节点同时成为Candidate，并发起选举 Request Vote，Candidate间肯定不会投决
2. Follower收到的请求肯定是一个先达，那么当前Follower已完成投票，后续的请求不予投决。坏的情况下，每个Candidate都有2票（votes），此轮获取Leader失败（*split vote*）

   > ...and each reaches a single follower node before the other.

3. 各自节点的election timeout仍在运行，最先timeout的节点发起新一轮的选举
4. 如果在选出Leader之后，心跳还未发出，另一个Candidate timout，则这个选举会正常发起，因Follower已完成了一轮的投票，所以拒绝请求，在Candidate也收到心跳请求时，它的term已低于Leader，将自己转态为Follower

- 网络分区下的复制日志

  以5个节点描述此场景，A为Leader，分区后表现为A、B，C、D、E两部分

  1. 先看Leader A所在的分区（A、B）

     > 当客户端发送数据变更请求R1时，此分区有Leader，还可以收到请求，在同步Follower节点时，因只有B一个节点，不足半数，所以B和Leader节点数据状态都是**Uncommitted**

  2. 再看C、D、E分区，因发生网络分区，无法收到Leader A的心跳，所以此分区重新选主，假如选出Leader为C，

     > 当客户端发送数据变更请求R2到C时，C同步到D、E，满足过半，所以3个节点数据状态都是**Committed**，正常返回给客户端

     注：C作为Leader之后，看似集群有了2个Leader，其实，A和C处于不同的任期（term）

  3. 当网络恢复后，5个节点处于同一个网络状态下

     > Leader C发起*AppendEntries*请求时， A降级为Follower，因为A、B分区的数据是未提交状态，是无效数据，此时可以删除，然后同步C中数据

  问题：1）如果Leader A先发起*AppendEntries*请求，怎么处理？2）C *AppendEntries*请求内容是什么，是所有**Committed**的数据吗？3）Leader A、C同时存在时，客户端怎么请求？集群什么状态？4）Leader A角度看：过半不可用，

- 解惑

> ## stale leader
>
> 在这样的情况下，我们来考虑读写。
>
> 首先，如果客户端将请求发送到了A，A无法将log entry 复制到majority节点，因此不会告诉客户端写入成功，这就不会有问题。
>
> 对于读请求，stale leader可能返回stale data，比如在read-after-write的一致性要求下，客户端写入到了term2任期的leader C，但读请求发送到了A。如果要保证不返回stale data，leader需要check自己是否过时了，办法就是与大多数节点通信一次，这个可能会出现效率问题。另一种方式是使用lease，但这就会依赖物理时钟。
>
> 从raft的论文中可以看到，leader转换成follower的条件是收到来自更高term的消息，如果网络分割一直持续，那么stale leader就会一直存在。而在raft的一些实现或者raft-like协议中，leader如果收不到majority节点的消息，那么可以自己step down，自行转换到follower状态。

- 避免未决

> 用randomized election timeouts来尽量避免平票 （split vote）
>
> 节点的数目都是奇数个，尽量保证过半（majority）

- refer

  国外动画过程演示（推荐）

  基于动画中文解读（不全面，可参考）

  半小时学会分布式一致性算法——Raft

  共识算法：Raft（存疑，不推荐）

  一文搞懂Raft算法（结合上文看，恰好能解释一些不足）

分布式事务

2PC

3PC

TCC

MQ事务

可扩展

一致性Hash

Range Based

全局流水号

高可用

Master-Slave

选举协议

限流

熔断

降级

高性能

分布式锁

分布式MQ

分布式缓存