分布式数据库

key value

分布式计算

map-reduce

/**/

好文章

https://www.cnblogs.com/richaaaard/p/6351705.html

/**/

水平分表和垂直分表

1，水平分割：

比方说qq用户有10亿，分成10个1亿的表。

用户id如果是主key，id%10取模，决定放入10个中的哪个表。

2，垂直分割：

这个就是大学学习数据库理论的时候，分表的办法。通过同一个主键，两个表建立联系。

垂直分割指的是：表的记录并不多，但是字段却很长，

表占用空间很大，检索表的时候需要执行大量的IO，严重降低了性能。

把大的字段拆分到另一个表，并且该表与原表是一对一的关系。

例如学生答题表tt：有如下字段：

Id name 分数 题目 回答

其中题目和回答是比较大的字段，id name 分数比较小。

这就可以使用垂直分割。我们可以把题目单独放到一张表中，通过id与tt表建立一对一的关系，

同样将回答单独放到一张表中。这样我们查询tt中的分数的时候就不会扫描题目和回答了。

/**/

存放图片、文件等大文件用文件系统存储，数据库只存储路径。

图片和文件存放在文件系统，甚至单独存放在一台服务器(图床)。

/**/

/**/

分布式数据库

分布式数据库重要的概念：分布式事务。

分布式事务首先是事务。

对于分布式数据库系统而言， 在保证数据一致性的要求下，进行事务的分发、协同多节点完成业务请求。

需要科学有效的一致性算法来支撑。
/**/

分布式数据库理论基础
CAP理论：

一致性(Consistency)、可用性 (Availability)、分区容忍性 (Partition tolerance)
分区容忍性 (Partition tolerance) 是必须的，网络分区和丢弃的消息已成事实。
系统设计人员必须在一致性和可用性之间进行权衡
/**/
/*

强一致性
Strict Consistency (强一致性) 也称为Atomic Consistency
(原子一致性) 或 Linearizable Consistency(线性一致性)，必须满足以下 两个要求：
1、任何一次读都能读到某个数据的最近一次写的数据。
2、系统中的所有进程，看到的操作顺序，都和全局时钟下的顺序一致。

对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。
简言之，在任意时刻，所有节点中的数据是一样的。

风河项目两个系统用pci连接保持一致，应该就是强一致性。
*/
/**/

BASE理论
BA：Basically Available 基本可用，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。
s：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。
E：Consistency 最终一致性，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。

一致性算法
/**/

Paxos:
Paxos 算法主要解决数据分片的单点问题，目的是让整个集群的结点对某个值的变更达成一致。

Paxos (强一致性) 属于多数派算法 。
任何一个点都可以提出要修改某个数据的提案，是否通过这个提案取决于这个集群中是否有超过半数的结点同意，所以 Paxos 算法需要集群中的结点是单数 。

Raft 算法是简化版的Paxos:
Raft 划分成三个子问题：
一是Leader Election;
二是 Log Replication;
三是Safety。

Raft 定义了三种角色 Leader、Follower、Candidate，
最开始大家都是Follower，当Follower监听不到Leader，就可以自己成为Candidate，发起投票 ，选出新的leader 。

其有两个基本过程：
① Leader选举：每个 C andidate随机经过一定时间都会提出选举方案，最近阶段中 得 票最多者被选为 L eader。
② 同步log：L eader会找到系统中log(各种事件的发生记录)最新的记录，并强制所有的follow来刷新到这个记录。
Raft一致性算法是通过选出一个leader来简化日志副本的管理，例如，日志项(log entry)只允许从leader流向follower。
/**/

raft算法学习：

其实可以理解成raft 协议。
/**
类比操作系统：
也是情景分析，
情景太复杂的情况，然后避免这些情景的发生。
**/

/**/

rpc 原理

作者：洪春涛

远程过程调用带来的新问题在远程调用时，我们需要执行的函数体是在远程的机器上的。

这就带来了几个新问题：

Call ID映射。在RPC中，所有的函数都必须有自己的一个ID。这个ID在所有进程中都是唯一确定的。客户端在做远程过程调用时，必须附上这个ID。然后我们还需要在客户端和服务端分别维护一个 {函数 <--> Call ID} 的对应表。两者的表不一定需要完全相同，但相同的函数对应的Call ID必须相同。当客户端需要进行远程调用时，它就查一下这个表，找出相应的Call ID，然后把它传给服务端，服务端也通过查表，来确定客户端需要调用的函数，然后执行相应函数的代码。

序列化和反序列化。客客户端和服务端使用的都不是同一种语言（比如服务端用C++，客户端用Java或者Python）。这时候就需要客户端把参数先转成一个字节流，传给服务端后，再把字节流转成自己能读取的格式。这个过程叫序列化和反序列化。同理，从服务端返回的值也需要序列化反序列化的过程。

网络传输。网络传输层需要把Call ID和序列化后的参数字节流传给服务端，然后再把序列化后的调用结果传回客户端。
只要能完成这两者的，都可以作为传输层使用。
因此，它所使用的协议其实是不限的，能完成传输就行。
尽管大部分RPC框架都使用TCP协议，但其实UDP也可以，而gRPC干脆就用了HTTP2。

具体过程如下：
// Client端
// int l_times_r = Call(ServerAddr, Multiply, lvalue, rvalue)
1. 将这个调用映射为Call ID。这里假设用最简单的字符串当Call ID的方法
2. 将Call ID，lvalue和rvalue序列化。可以直接将它们的值以二进制形式打包
3. 把2中得到的数据包发送给ServerAddr，这需要使用网络传输层
4. 等待服务器返回结果
5. 如果服务器调用成功，那么就将结果反序列化，并赋给l_times_r

// Server端
1. 在本地维护一个Call ID到函数指针的映射call_id_map，可以用std::map<std::string, std::function<>>
2. 等待请求
3. 得到一个请求后，将其数据包反序列化，得到Call ID
4. 通过在call_id_map中查找，得到相应的函数指针
5. 将lvalue和rvalue反序列化后，在本地调用Multiply函数，得到结果
6. 将结果序列化后通过网络返回给Client

实现这些功能的库：
Call ID映射可以直接使用函数字符串，也可以使用整数ID。
映射表一般就是一个哈希表。

序列化反序列化可以自己写，也可以使用Protobuf或者FlatBuffers之类的。
网络传输库可以自己写socket，或者用asio，ZeroMQ，Netty之类。

当然，这里面还有一些细节可以填充，比如如何处理网络错误，如何防止攻击，如何做流量控制，等等。
但有了以上的架构，这些都可以持续加进去。

最后，有兴趣的可以看我们自己写的一个小而精的RPC库 Remmy（hjk41/Remmy），对于理解RPC如何工作很有好处。

/**/
snapshot 定义
SNIA对快照（Snapshot）的定义是：
关于指定数据集合的一个完全可用拷贝，该拷贝包括相应数据在某个时间点（拷贝开始的时间点）的映像。

快照大致分为2种，一种叫做即写即拷（copy-on-write）快照，通常也会叫作指针型快照，
占用空间小，如果没有备份而原数据盘坏了，数据就无法恢复了；
/*类似h263，相对于以前的变化。如果没有原始数据，就恢复不了*/

镜像型快照实际就是当时数据的全镜像，不过要占用到相等容量的空间原理 (shared virtual array).当然也可以压缩。
/**/
/**/
Raft算法
将其问题分解为

- 领导选举
- 日志复制
- 安全性

领导选举： 对于一个集群只有一个leader（领导）。 领导选举时机和规则。

日志复制：其实就是同步操作数据的过程。leader将操作日志同步到其他节点。

安全性：在不同的情况，我们都能保证一致性，这也就是安全性需要考虑的问题。

/*疑问：客户端如何知道leader，新加入的机器如何参与这个网络*/
/**/

一些基本概念定义：

1、三种角色

Raft是一个用于管理日志一致性的协议。

它将分布式一致性分解为多个子问题：

Leader选举（Leader election）、日志复制（Log replication）、安全性（Safety）、日志压缩（Log compaction）等。

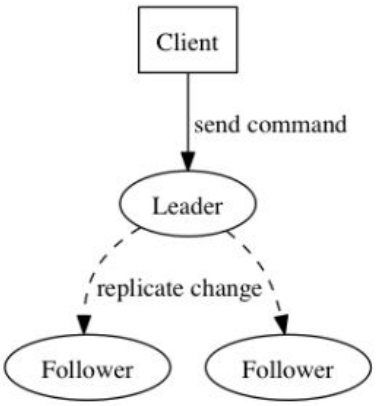
同时，Raft算法使用了更强的假设来减少情景，使之变的易于理解和实现。

Raft将系统中的角色分为领导者（Leader）、跟从者（Follower）和候选者（Candidate）：

Leader：接受客户端请求，并向Follower同步请求日志，当日志同步到大多数节点上后告诉Follower提交日志。

Follower：接受并持久化Leader同步的日志，在Leader告之日志可以提交之后，提交日志。

Candidate：Leader选举过程中的临时角色。



Raft要求系统在任意时刻最多只有一个Leader，正常工作期间只有Leader和Followers。

Raft算法将时间分为一个个的任期（term），每一个term的开始都是Leader选举。

如果Leader选举失败，该term就会因为没有Leader而结束。

在成功选举Leader之后，Leader会在整个term内管理整个集群。

2、Term

Raft 算法将时间划分成为任意不同长度的任期（term）。

任期用连续的数字进行表示。

每一个任期的开始都是一次选举（election），一个或多个候选人会试图成为领导人。

如果一个候选人赢得了选举，它就会在该任期的剩余时间担任领导人。

在某些情况下，有可能没有选出领导人，那么，将会开始另一个任期。

Raft 算法保证在给定的一个任期最多只有一个领导人

3、RPC

Raft 算法中服务器节点之间通信使用远程过程调用（RPC），

并且基本的一致性算法只需要两种类型的 RPC，为了在服务器之间传输快照增加了第三种 RPC。

【RPC有三种】：

RequestVote RPC：候选人在选举期间发起。

AppendEntries RPC：领导人发起的一种心跳机制，复制日志也在该命令中完成。

InstallSnapshot RPC: 领导者使用该RPC来发送快照给太落后的追随者。

/**/

Leader选举

1、Leader选举的过程

Raft 使用心跳（heartbeat）触发Leader选举。

当服务器启动时，初始化为Follower。

Leader向所有Followers周期性发送heartbeat。

如果Follower在选举超时时间内没有收到Leader的heartbeat，就会等待一段随机的时间后发起一次Leader选举。

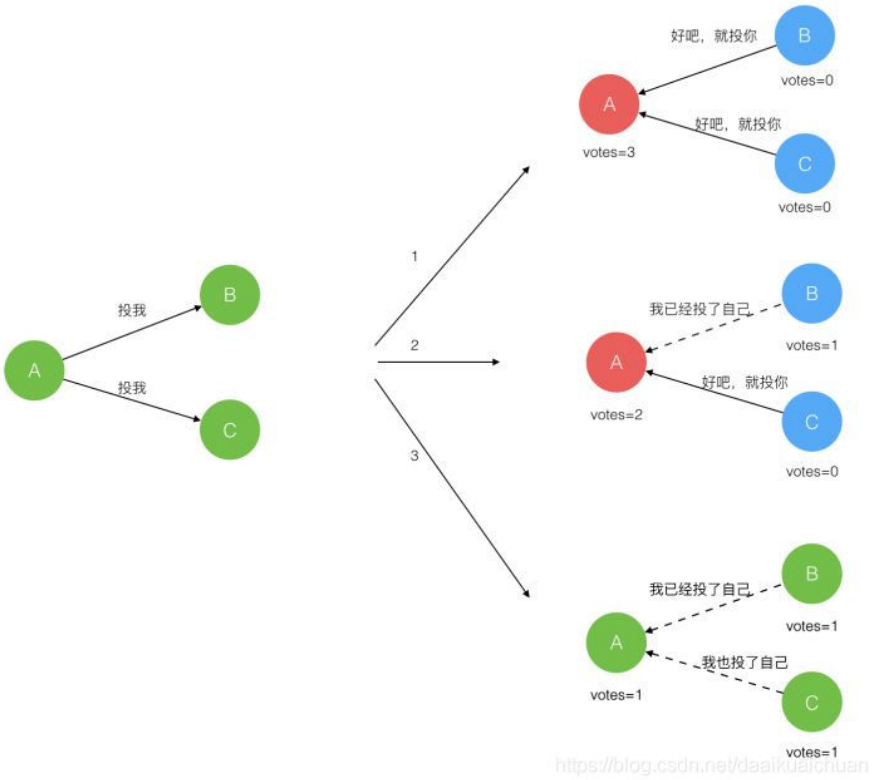
每一个follower都有一个时钟，是一个随机的值，表示的是follower等待成为leader的时间，谁的时钟先跑完，则发起leader选举。

Follower将其当前term加一然后转换为Candidate。它首先给自己投票并且给集群中的其他服务器发送 RequestVote RPC。结果有以下三种情况：

赢得了多数的选票，成功选举为Leader；

收到了Leader的消息，表示有其它服务器已经抢先当选了Leader；

没有服务器赢得多数的选票，Leader选举失败，等待选举时间超时后发起下一次选举。



/**

选举有两个重要的属性：安全（Safety）和可用（Liveness）

安全（Safety） 指的是必须最多只有一个候选者可以在某一任期内赢得领导者地位。Raft 可以保证这件事。每台服务器只给一个候选者投票，一旦它投出选票，它就会拒绝来自其他候选者的任何请求。服务器并不关心它的票到底投给了哪台服务器。为了实现这种机制，服务器需要保证将自己的投票信息存储到磁盘，这样就能在服务器崩溃之后也能恢复到之前的状态。否则就会出现服务器已经作出投票，并在崩溃重启后，在同一任期内将票又投给了另外一个不同服务器的情况。因为每台服务器只能进行一次投票，而且每个候选者都必须获得多数票，也就可以发现，不可能出现两个候选者同时获胜的情况。/*投票之前，保存数据到disk，崩溃后能得到以前状态*/

比方说有三台服务器在某一任期内进行选举，另外两台服务器显然无法获得多数票。不过后面会介绍不同任期间会出现不同候选者获胜的情况，但在某一确定的任期内，只有一个候选者可以被选举为领导者。

可用（Liveness） 需要保证一定有获胜者，这样系统不会永远处于没有领导者的状态。问题在于理论上，会反复出现分票的情况，多个候选者在同一任期内同时开始进行选举，这样就会导致分票，在超时之后，又进行新一轮的选举又再次出现分票，所以从理论上说这样的状态可以无限循环下去。Raft 需要分散出现超时的间隔，每台服务器都会随机的计算下次超时的间隔时间，这个时间间隔在 [T, 2T] 之间。T 代表着选举超时的时间，即服务器可能出现超时的最短时间。通过将超时时间分散，可以降低两台服务器同时开始选举的机率，先启动的那台有足够的时间向其他所有服务器发起请求，并在其他服务器参与竞争之前就完成选举这个过程。当这个超时间隔时间远大于广播投票请求的时间时，这个策略会变得更为有效。这里的广播时间指的是，一台服务器与其他所有服务器通信所需的时间。
**/

Raft发起选举的情况有如下几种：
刚启动时，所有节点都是follower，这个时候发起选举，选出一个leader；
当leader挂掉后，时钟最先跑完的follower发起重新选举操作，选出一个新的leader。
成员变更的时候会发起选举操作。

Leader选举的限制

在Raft协议中，所有的日志条目都只会从Leader节点往Follower节点写入，且Leader节点上的日志只会增加，绝对不会删除或者覆盖。

这意味着Leader节点必须包含所有已经提交的日志，即能被选举为Leader的节点一定需要包含所有的已经提交的日志。

因为日志只会从Leader向Follower传输，所以如果被选举出的Leader缺少已经Commit的日志，那么这些已经提交的日志就会丢失，显然这是不符合要求的。

这就是Leader选举的限制：能被选举成为Leader的节点，一定包含了所有已经提交的日志条目。

也就是说被投票的leader的日志条目号，一定不会比投票者的日志条目号低。

/*从这里就能想到，如果旧的leader，最新的日志条目没有过半，就是没有commit。就会造成新的leader有可能没有这个日志条目，然后这个日志条目就没有了*/

Election Basics

- **Increment current term**
- **Change to Candidate state**
- **Vote for self**
- **Send RequestVote RPCs to all other servers, retry until either:**
  1. Receive votes from majority of servers:
     - Become leader
     - Send AppendEntries heartbeats to all other servers
  2. Receive RPC from valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election