分布式

[1. Raft 1](#_135)

[1.1. server状态 1](#_174)

[1.2. 流程 1](#_140)

[1.2.1. Leader选举 1](#_136)

[1.2.2. 日志复制 2](#_137)

[1.2.3. 日志压缩 2](#_138)

[1.2.4. 成员变更 3](#_139)

[1.3. 特性 3](#_141)

[1.3.1. 选举安全 3](#_142)

[1.3.2. 领导人只附加原则 3](#_143)

[1.3.3. 日志匹配原则 4](#_144)

[1.3.4. 领导人完全特性 4](#_145)

[1.3.5. 状态机安全 4](#_146)

[1.4. 细节讨论 4](#_202)

[1.4.1. Leader崩溃 4](#_203)

[2. CAP 4](#_101)

[2.1. Consistency：一致性 4](#_102)

[2.2. Availability：可用性 4](#_103)

[2.3. Partion Tolerance：分区容错性 5](#_104)

[2.4. CP 5](#_105)

[2.5. AP 5](#_192)

[2.6. CP还是AP依赖于使用场景 5](#_193)

[3. BASE 5](#_106)

[3.1. Basically Available 5](#_107)

[3.2. Soft State 5](#_108)

[3.3. Eventually Consistent：最终一致 5](#_109)

[4. Lease 5](#_110)

[4.1. 场景 5](#_111)

[4.2. 基于lease的读 6](#_112)

[4.3. 基于lease的写 6](#_113)

[5. 更新成功判断 6](#_114)

[5.1. write-all-read-one 6](#_115)

[5.2. Quorum 6](#_116)

[6. 有趣的问题 6](#_184)

[6.1. Paxos、ZAB和Raft的区别 6](#_189)

[6.2. 分布式事务解决方法 7](#_195)

[6.2.1. 刚性事务2PC,3PC等 7](#_196)

[6.2.2. 柔性事务 7](#_197)

[7. 3PC 9](#_120)

[7.1. 流程 9](#_121)

[7.1.1. canCommit 9](#_122)

[7.1.2. PreCommit 9](#_123)

[7.1.3. doCommit 10](#_124)

[7.2. 相比于2PC的改进 10](#_125)

[8. 2PC 11](#_117)

[8.1. 流程 11](#_147)

[8.1.1. 协调者 11](#_118)

[8.1.2. 参与者 11](#_119)

[8.2. 事务中断 11](#_149)

[8.3. 问题 11](#_148)

[8.3.1. 同步阻塞 11](#_150)

[8.3.2. 单点问题 12](#_151)

[8.3.3. 数据不一致 12](#_152)

[8.3.4. 太过保守 12](#_153)

[8.4. 细节 12](#_194)

[9. Paxos 12](#_126)

[9.1. basic-paxos 12](#_186)

[9.1.1. 角色 12](#_127)

[9.1.2. 提案 13](#_154)

[9.1.3. 原则 13](#_155)

[9.1.4. 实现细节 13](#_208)

[9.1.5. 竞争及活锁 14](#_191)

[9.2. multi-paxos 14](#_187)

[9.2.1. 比basic-paxos的优势 14](#_188)

[9.2.2. 实现细节 14](#_156)

[9.3. 工程实践 16](#_131)

[9.3.1. Chubby 16](#_157)

[9.3.2. zookeeper 16](#_132)

# Raft

## server状态

**所有服务器都拥有的状态分为两种**：

1. 持久的

current term：该server的当前任期

votedFor：投给谁了

log[]：日志条目

2. 经常变的

commitIndex：已经提交日志的最大索引

lastApplied：应用到状态机的最大索引

**Leader server还具有另外的状态：**

nextIndex[]：对于每个slave，需要发送给他的下一个日志条目索引

matchIndex[]：对于每个slave，已经复制给他的最大索引

## 流程

### Leader选举

#### RequestVote

##### 请求

包含以下参数：

term：候选人任期号

candidateId：候选人id，如ip+port的组合

lastLogIndex：候选人最后日志条目索引值

lastLogTerm：候选人最后日志条目任期号

##### 回复

包含以下参数：

term：以便候选人更新term

voteGranted：false or true，是否赢得选票

#### 发送者

初始阶段都是Follower，心跳超时后开始选举。

为了防止所有server同时发起选举，通过随机错开选举时间来解决。

#### 接收者

当得到某个候选者的选票时，只有对方的term大于等于自己的，并且当前设定的leader为空或者和候选人一样时，需满足候选人上个日志的任期号和索引都要大于等于自己，才同意该选票

### 日志复制

#### AppendEntries

##### 请求

包含以下参数：

term：领导人任期号

leaderId：领导人身份

prevLogIndex：上个日志索引

prevLogTerm：上个日志任期

entries[]：需要添加的日志

leaderCommit：leader提交日志的最大索引

##### 回复

包含以下参数：

term：任期

success：是否成功

#### Follower

1. 如果请求的term < currentTerm，则拒绝

2. 如果相同prevLogIndex上的日志任期号和请求中的任期号不同，返回false。（其实在暗示Leader递减nextIndex）

3. 如果已经存在的日志条目与新的产生冲突（索引一样任期号不同），则删除这一条及之后所有的日志。

4. 添加日志，并更新leaderCommit

#### commitIndex

Leader的CommitIndex在收到过半Follower确认后才会更新。

Follower的CommitIndex则在收到AppendEntries后更新。

### 日志压缩

图 12 展示了 Raft 中快照的基础思想。每个服务器独立的创建快照，只包括已经被提交的日志。主要的工作包括将状态机的状态写入到快照中。Raft 也包含一些少量的元数据到快照中：最后被包含索引指的是被快照取代的最后的条目在日志中的索引值（状态机最后应用的日志），最后被包含的任期指的是该条目的任期号。保留这些数据是为了支持快照后紧接着的第一个条目的附加日志请求时的一致性检查，因为这个条目需要前一日志条目的索引值和任期号。为了支持集群成员更新（第 6 节），快照中也将最后的一次配置作为最后一个条目存下来。一旦服务器完成一次快照，他就可以删除最后索引位置之前的所有日志和快照了。

### 成员变更

通过单个节点成员变更来保证在同一时刻Leader只有一个。

变更流程如下：

1. 向Leader提交一个成员变更请求，请求的内容为服务节点的是添加还是移除，以及服务节点的地址信息。

2. Leader在收到请求后，会向日志中追加一条ConfChange的日志，其中包含了Cnew，后续这个日志会随着AppendEntries的RPC同步到所有的Follower节点中。

3. 当ConfChange日志被添加到日志中时立即生效（注意：不是等到提交以后才生效）

4. 当ConfChange日志被复制到Cnew的Majority服务器上时，那么就可以对日志进行提交了。

#### 追赶新的服务器

在添加服务器后，如果新的服务器需要花费很长时间来追赶日志，那么这段时间内服务不可用。

解决方法有：

在及群众加入新的角色Learner，Learner只对集群的日志进行复制，并不参与投票和提交决定。

加入一个新phase，这个阶段会在固定时间内尝试追赶日志：在这个阶段会分为多个Rounds（比如10）向Leader同步日志，如果新节点能够正常同步日志，那么每一轮的日志同步时间都会缩短，如果最后一轮同步日志的时间未超时，那么就足够接近于Leader，可以加入集群。

## 特性

### 选举安全

一个任期号，最多只有一个领导人

### 领导人只附加原则

Leader绝对不会删除和覆盖自己的日志，只会增加

### 日志匹配原则

如果两个日志在相同索引任期号一样，那么这个日志从头到索引位置都一样

### 领导人完全特性

如果日志条目在某个任期提交，那么一定会出现在更大任期号的所有领导人中

### 状态机安全

如果领导人已经将给定的索引值的日志应用到状态机，那么其他服务器在这个索引位置不会应用不同的日志

## 细节讨论

### Leader崩溃

#### 日志复制阶段崩溃

此时该次请求将成为一个uncommitted log，也就是领导者成功接收，但并未复制到大多数节点上的log。

该log可能会被覆盖丢弃，也可能最终被commit和apply，到底是哪种结果，取决于哪个节点当选领导者。

如果持有uncommitted log的节点当选领导者，log将被commit和apply

如果非持有uncommitted log的节点当选领导（持有uncommitted log的节点也down掉了），那么log将被覆盖丢弃。

#### 复制成功，发送给客户端响应前崩溃

此时的log是commit Log，也就是复制到大多数节点的log，是不会丢弃的。

#### 发送给客户端响应后崩溃

同样是committed log，所以是不会丢弃的。

# CAP

## Consistency：一致性

## Availability：可用性

## Partion Tolerance：分区容错性

## CP

CP不是简单地放弃A，而是保障CP的硬件条件去追求A，所以产生了过半写入这样非常经典的使用方式。

这里的一致性保证在于，分布式节点可以根据少数服从多数完成数据的一致性要求，因此产生最大效益。

更高可用性：对所有实例不再是全部写入成功才认为是成功。

更快响应性：在使用广播获取过半结果后直接认定结果，依靠补充手段实现数据的一致性。

## AP

允许数据的最终一致性。

## CP还是AP依赖于使用场景

# BASE

当分布式系统出现不可知故障时，允许损失部分可用性。具体表现为：

响应时间的损失：出现故障后，通过延长响应来保障可用性

功能上的损失：流量高峰时，通过服务降级、限流等治理手段提供有损服务。

## Basically Available

基本可用：功能降级，保证核心功能

## Soft State

软状态：允许数据存在中间状态，允许同步存在时延

## Eventually Consistent：最终一致

# Lease

## 场景

在分布式系统中，中心服务器存储、维护着一些数据，这些数据是系统元数据。其他节点通过访问中心服务器读取、修改其上的元数据。每次访问元数据，那么中心服务器节点的性能将成为系统的瓶颈。

中心服务器向各节点发送数据时颁发一个lease，每个lease上的有效期是一个明确时间点。在lease有效期内，中心服务器保证不会修改对应数据的值

## 基于lease的读

判断元数据是否处于本地cache且lease处于有效期：

是：直接返回cache中的元数据

否：向中心服务器请求元数据（同时返回lease，进行再一次缓存）

## 基于lease的写

1. 节点向中心服务器发起元数据请求

2. 中心服务器收到修改请求，阻塞所有的读请求（接收但不回应）

3. 中心服务器等待所有lease失效

4. 中心服务器修改元数据并向客户端节点返回成功

# 更新成功判断

## write-all-read-one

只有在所有副本上更新成功，才认为更新成功，更新的可用性不高

## Quorum

部分副本更新成功，就认为该更新成功

# 有趣的问题

## Paxos、ZAB和Raft的区别

**Paxos**

Paxos可以分为basic-paxos和multi-paxos。

1. basic-paxos

是基础Paxos，Proposer、Follower和Learner，允许多Proposer。存在的问题有：多Proposer同时提出议案，可能会产生冲突，从而导致一次决议的时间变长。

2. multi-paxos

让具有最大proposer-id的Proposer成为Leader，Leader发起的提案才是有效的。但是multi-paxos不保证新的Leader具有所有提交的日志，所以需要特殊的机制来保证日志的同步。相比于Raft强调日志连续性，multi-paxos允许日志有空洞。因此当新Leader产生时，需要通过Paxos阶段一向其他节点学习到缺失的可以被提交的日志（例如将所有日志的确认和学习合并到一轮消息中，减少消息数目等）。

**Raft**

通过明确Leader选举、日志复制等流程大大简化了一致性协议的理解难度。严格来说也是2PC，因为虽然只需要通过一次请求回复就能处理客户端请求，但是Follower CommitIndex的更新是在后续的AppendEntries中。

同时它有一个严格的限制条件，即需要保证日志的连续性。在数据库方面可能给会导致串行的事务提交，影响效率。

**ZAB**

是zookeeper使用的协议，分为崩溃恢复和消息广播。在消息广播阶段就是2PC。

## 分布式事务解决方法

分布式事务触发的场景有以下几种:

跨数据库分布式事务：数据库的物理分割下保障垮库操作的ACID

跨服务分布式事务：服务的网络分割下保障多服务的事务完整性

混合式分布式事务：跨数据库分布式事务+跨服务分布式事务

先来个完整的结论：业务规避 > BASE柔性事务 > CP刚性事务

你好

### 刚性事务2PC,3PC等

### 柔性事务

#### 事务消息

##### 基于DB的本地消息表方案

1. 开启本地事务，进行更新操作并将其他库的操作写入事件表

2. 由定时任务定期读事件表，并发送到消息队列。

3. 其余服务器接收到消息队列中的消息，完成其他库的事务。

业务方：直接利用本地事务，将业务数据和事务消息直接写入数据库。

投递线程：使用专门的投递工作线程进行事务消息投递到MQ，根据投递ACK去删除事务消息表记录

##### 基于MQ的自身消息方案

​事务发起方首先发送prepare消息到MQ；

1. 在发送prepare消息成功后执行本地事务；

2. 根据本地事务执行结果返回commit或者是rollback；

3. 如果消息是rollback, MQ将删除该prepare消息不进行下发，如果是commit消息，MQ将会消息发送给consumer端；

4. 如果执行本地事务过程中，执行端挂掉，或者超时，MQ服务器端将不停的询问producer来获取事务状态；

5. Consumer端的消费成功机制有MQ保证

MQ事务消息方案因为使用了半消息机制，对业务页具有比较大侵入性，有以下注意点：

业务方调用半消息，并提供对应的回查方法；

MQ提要提供半消息机制，并定期扫描长期半消息，对消息生产者进行回查确认事务。

消费方需要进行幂等消费。

当前MQ并不支持事务消息。

#### TCC

￼

TCC 是一种补偿型事务，该模型要求应用的每个服务提供 try、confirm、cancel 三个接口，它的核心思想是通过对资源的预留(提供中间态，如账户状态、冻结金额等)，尽早释放对资源的加锁，如果事务可以提交，则完成对预留资源的确认，如果事务要回滚，则释放预留的资源。

TCC模型完全交由业务实现，每个子业务都需要实现Try-Confirm-Cancel三个接口，对业务侵入大，资源锁定交由业务方。

Try：尝试执行业务，完成所有业务检查(一致性)，预留必要的业务资源(准隔离性)。

Confirm：确认执行业务，不再做业务检查。只使用Try阶段预留的业务资源，Confirm操作满足幂等性。

Cancel：取消执行业务释放Try阶段预留业务资源。

**一个完整的业务活动由一个主业务服务与若干子业务服务组成：**

1. 主业务服务负责发起并完成整个业务活动

2. 业务服务提供TCC型业务操作。

3. 业务活动管理器控制业务活动的一致性，它登记业务活动中的操作，并在业务活动提交时确认所有的TCC型操作的Confirm操作，在业务活动取消时调用所有TCC型操作的Cancel操作。

TCC将事务提交划分成两个阶段，Try即为一阶段，Confirm 和 Cancel 是二阶段并行的两个分支，二选一。从阶段划分上非常像2PC，我们是否可以说TCC是一种2PC或者2PC变种呢？其实不可以，原因如下：

2PC的操作对象在于资源层，对于开发人员无感知；而TCC的操作在于业务层，具有较高开发成本。

2PC是一个整体的长事务，也是刚性事务；而TCC是一组的本地短事务，是柔性事务。

2PC的Prepare(表决阶段)进行了操作表决；而TCC的try并没有表决准备，直接兼备资源操作与准备能力

2PC是全局锁定资源，所有参与者阻塞 交互等待TM通知；而TCC的资源锁定在于Try操作，业务方可以灵活选择业务资源的锁定粒度。

#### 最大努力通知事务

# 3PC

## 流程

### canCommit

1. 事务询问

协调者向所有参与者发送一个包含事务内容的canCommit请求，询问是否可以执行事务提交操作，并开始等待各参与者的响应。

2. 各参与者向协调者反馈事务询问的响应

参与者在接收到来自协调者的canCommit请求，正常情况下，如果自身认为可以顺利执行事务，那么会反馈Yes响应，并进入预备状态，否者反馈No响应。

### PreCommit

加入协调者从所有参与者获得的反馈都是Yes响应，那么就会执行事务预提交。

1. 发送PreCommit请求

协调者向所有参与者发送preCommit请求，并进入Prepared阶段。

2. 事务预提交

参与者接收到preCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。

3. 各参与者向协调者反馈事务执行的响应

如果参与者成功执行了事务操作，那么就会反馈给协调者Ack响应，同时等待最终的指令：提交(commit)和终止（abort)。

中断事务：

如果任何一个参与者向协调者反馈了No响应，或者在等待超时之后，协调者尚无法接收所有参与者的反馈响应，那么就会中断事务。

1. 发送中断请求

协调者向所有参与者节点发出abort请求。

2. 中断事务

无论是收到来自协调者的abort请求，还是在等待协调者请求过程中出现超时，参与者都会中断事务。

### doCommit

该阶段将进行真正的事务提交。

执行提交：

1. 发送提交请求

向参与者发送doCommit请求。

2. 事务提交

参与者收到doCommit请求后，会正式执行事务提交操作，并在完成之后释放整个事务执行期间占用的事务资源。

3. 反馈事务提交结果

参与者在完成事务提交之后，向协调者发送Ack消息。

4. 完成事务

协调者接收到所有参与者反馈的Ack消息，完成事务。

第三阶段的超时，会导致参与者继续事务提交。

## 相比于2PC的改进

降低了参与者的阻塞范围，能够在出现单点故障后继续达成一致。但是仍会出现数据不一致的情况。

# 2PC

## 流程

### 协调者

1. 写本地日志"begin\_commit"，并进入WAIT状态

2. 向所有参与者发送"prepare消息"

3. 等待响应，此时有两种情况：

若收到任何参与者发来的"vote-abort"消息：写本地"global-abort"，进入ABORT状态，然后向参与者发送"global-abort"消息

若收到所有参与者发来的"vote-commit"消息：写本地"global-commit"，进入COMMIT状态，然后向所有参与者发送"global-commit"消息

4. 收到所有参与者对"global-abort"或"global-commit"的响应，写"end-transaction"

### 参与者

1. 写本地日志"init"，进入INIT状态

2. 接收协调者发来的"prepare"消息，开始执行，并将Undo和Redo信息记入事务日志中

若参与者成功执行事务：则写"ready"，进入READY状态，发送"vote-commit"消息，然后等待响应。

若参与者执行事务失败：写本地"abort"，进入ABORT状态，想协调者发送"vote-abort"消息，然后等待"global-abort"消息，并确认。

## 事务中断

1. 任何一个参与者向协调者反馈了"vote-abort"消息。

2. 等待超时后，协调者尚无法收到所有参与者的反馈

## 问题

### 同步阻塞

同步阻塞会极大地限制分布式系统的性能。在二阶段提交的执行过程中，所有参与该事务操作的逻辑都处于阻塞状态。也就是说，各个参与者都在等待其他参与者响应的过程中（等到所有参与者执行结束），将无法进行任何操作。

### 单点问题

一旦协调者出现问题，那么整个二阶段提交流程将无法运转

### 数据不一致

阶段二（执行事务提交的时候），当协调者向所有的参与者发送Commit请求之后，发生了局部网络异常或者协调者在尚未发送完所有Commit请求之前自身发生了崩溃，导致最终只有部分参与者收到了Commit请求。

### 太过保守

如果在协调者指示参与者进行事务提交询问的过程中，参与者出现故障而导致协调者始终无法获取到所有参与者的响应信息的话，这时协调者只能依靠其自身的超时机制来判断是否需要中断事务。

二阶段提交协议没有设计较为完善的容错机制，任意一个节点的失败都会导致整个事务的失败。

## 细节

比如电商下单场景，需要支付服务进行支付，库存服务扣减库存，订单服务进行订单生成，物流服务更新物流信息等。

其中每个服务都可以作为参与者。

# Paxos

## basic-paxos

https://juejin.im/post/6844903817297788942

### 角色

#### Proposer

提案者

1. 向所有的Acceptor发送消息Prepare(b)，b是Paxos的轮数

2. 如果收到任何一个Acceptor发来的Reject(B)，则本轮失败，将b设为B+1,重新开始。

3. 如果收到N/2 + 1以上的Promise(b, v\_i），其中v\_i表示Acceptor最近一次在i轮设置了v

如果Promise(b,v)中的v都为空，则有Proposer选择一个v，广播Accept(b,v)

否则，选择最大i的v，广播Accept(b,v)

4. 如果收到Nack(B),将轮数置为B+1后重新开始

#### Acceptor

批准者

B为该Acceptor接收到的最大轮数

1. 接收Prepare(b)

b>B，设置B=b，并回复Promise(b, V\_B)

否则，回复Reject(B)

2. 接收Accept(b,v)

如果b<B,则回复Nack(B)

设置V为v，批准了v，广播Accepted消息

#### Learner

可以让所有的Acceptor将它们对提案的批准情况，统一发送给一个特定的Learner（主Learner），这样能减少通信次数，但引入了一个新的不稳定因素：主Learner随时可能出现故障（或者发给特定的Learner集合）。

### 提案

将修改某个变量的值为某个值，设置当前primary为某个节点都抽象为value

### 原则

1. 只有被提出的提案才能被选定

2. 只能有一个值被选定

3. 如果某个进程认为某个提案被选定了，那么这个提案必须是真的被选定的那个

### 实现细节

#### Proposal Number

1. 全局唯一：无论是相同的Proposer的不同Proposal，还是不同的Proposer的相同或者不同proposal

2. 单调递增：越大优先级越高

一个简单的做法：Proposal Number由Round Number组合完成，后面拼接server id。maxRound是递增的，并且需要被持久化在磁盘中。

#### 持久化状态

**1. minProposal**

server接收到的最小即将accept的Proposal number，如果是0表示server没有收到任何的Prepare request。

**2. accept proposal**

server最后一次accept的Proposal number，如果是0表示server没有accept任何Proposal。

**3. accept value**

最近server accept的Proposal的value，如果是null表示server还没有accept任何的Proposal。

**4. maxRound**

server锁看见的最大round number。

#### log ID

log id就是Paxos instance，表示一个提案（独一无二）

具体是从整个集群中查询所有的server获得当前最大logID，加1后作为执行本次Paxos instance的唯一标识。

### 竞争及活锁

但在 Paxos 协议过程中，虽然也存在着并发竞争，不会出现上述死锁。这是因为， Paxos 协议引入了轮数的概念，高轮数的 paxos 提案可以抢占低轮数的 paxos 提案。 从而避免了死锁的发生。

然而这种设计却引入了“活锁”的可能，即 Proposer 相互不断以更高的轮数提出议案，使得每轮 Paxos过程都无法最终完成，从而无法批准任何一个 value。

## multi-paxos

### 比basic-paxos的优势

multi-paxos降低basic-paxos出现多个proposer冲突的频率，合并阶段一从而将一次决议的平均代价缩小到最优的两次。

将basic-paxos并行化，能够处理多个提案。

### 实现细节

使用1个主Proposer，也就是proposer-id最大的proposer提出的决议才是有效的。

#### Leader产生

经过一轮的Basic-Paxos，成功得到多数派accept的proposer即为leader。之后可以通过lease机制，保持这个leader的身份，使得其他的proposer不再发起提案，这样就进入了一个leader任期。

在这个任期中，由于没有了并发冲突，这个leader对后续的日志进行投票时，不必每次都向多数派询问logId，也不必执行prepare阶段，直接执行accept阶段。

#### confirm日志优化

在Paxos协议中，对于决议的读取也是需要执行一轮Paxos过程的，在实际工程中做数据恢复时，对每条日志都执行一轮Paxos的代价过大，因此引入需要引入一种被成为confirm的机制，即leader持久化一条日志，得到多数派的accept后，就再写一条针对这条日志的confirm日志，表示这条日志已经确认形成了多数派备份，在回放日志时，判断如果一条日志有对应的confirm日志，则可以直接读取本地内容，而不需要再执行一轮Paxos。confirm日志只要写本地即可，不需要同步到备机，但是出于提示备机及时回放收到日志的考虑（备机收到一条日志后并不能立即回放，需要确认这条日志已经形成多数派备份才能回放），leader也会批量的给备机同步confirm日志。出于性能的考虑，confirm日志往往是延迟的成批写出去，因此仍然会出现部分日志已经形成多数派备份，但是没有对应的confirm日志的情况，对于这些日志，需要在恢复过程中进行重确认。

在实际的工程实践中，可以使用基于logID的滑动窗口机制来限制confirm日志与对应的原始日志的距离，以简化日志回放与查询逻辑

#### Leader对日志的重确认

没有confirm日志的原始日志，需要进行一轮Paxos，进行重确认。即“空洞”需要重确认，重确认的结束位置是当前集群中最大的log ID。

#### 服务器状态

所有Acceptor上面持久化的状态：

lastLogIndex：已经accept的最大的log entry index。

minProposal：最小的接收到的Proposal id（但是未accept），如果是0表示未接收任何Proposal Request。

firstUnchosenIndex：acceptedProposal[index] != '无穷大'的最小log entry index。

每个acceptor上面存放的复制日志，日志index（1 <= index <= lastLogIndex）。每个Log entry包含以下内容：

acceptedProposal[i]：这个log entry最后接受的Proposal id。初始时为0，如果这个日志被提交，则设置为无穷大。

acceptedValue[i]：这个log entry最后接受的value

所有Proposal上面持久化的状态：

maxRound：Proposal见过的最大round number。

在Proposer里经常改变的：

nextIndex：下一个可用的log entry的index

prepared：如果prepared为true，那么proposer就不需要prepare阶段，直接进行accept阶段，否则需要prepare阶段。prepared初始化为false，当收到大多数acceptor的prepare request的相应消息noMoreAccepted为true时，会将prepared设置为true。

## 工程实践

### Chubby

### zookeeper

为分布式应用提供了高效且可靠的分布式协调服务，提供了注入统一命名服务、配置管理和分布式锁等分布式的基础服务。

#### 集群角色

##### Leader

为客户端提供读和写服务

##### Follower

能提供读服务，并参与Leader选举过程，并参与到写操作的"过半写成功"策略。

##### Observer

能提供读服务，但是不参与Leader选举和"过半写成功"策略。

#### 数据节点Znode

Znode能够保存数据内容和一系列属性信息。

##### 持久节点

##### 临时节点

#### 权限

##### create

创建子节点的权限

##### read

获取节点数据和子节点列表的权限

##### write

更新节点数据的权限

##### delete

删除子节点的权限

##### admin

设置节点ACL的权限。

#### ZAB协议

ZooKeeper Atomic Broadcast（ZAB， Zookeeper原子消息广播协议）

##### 崩溃恢复

当Leader服务器出现网络中断、崩溃退出与重启等异常情况时采用该模式。

新Leader要具有集群中所有机器最高编号（zxid最大）的事务Proposal，跟Raft异曲同工之妙。

选出Leader之后，就是要进行同步，将没有被Follower同步的事务Proposal逐个发送给Follow服务器，如果有不同的Proposal，则进行回退操作（Raft无疑了）。

Leader在正式工作前，会首先确认事务日志中的所有Proposal是否已经被集群中过半的服务器commit。

Leader 服务器需要确保所有的 Follower 服务器能够接收到每一条事务的 Proposal ，并且能将所有已经提交的事务 Proposal 应用到内存数据中。等到 Follower 将所有尚未同步的事务 Proposal 都从 Leader 服务器上同步过啦并且应用到内存数据中以后，Leader 才会把该 Follower 加入到真正可用的 Follower 列表中。

##### 消息广播

当Leader选举成功后，同时集群中有过半的机器与该Leader服务器完成了状态同步之后，就会转入消息广播模式。

消息广播使用的是原子广播协议，类似二阶段提交过程。针对客户端的事务请求，Leader服务器会为其生成对应的事务Proposal，并将其发送给集群中其余所有的机器，然后再分别收集各自的选票，最后进行事务提交。

所有的Follewer服务器要么正常反馈Leader提出的事务Proposal，要么抛弃Leader服务器。

在过半Follower服务器反馈Ack之后就可以开始提交事务Proposal

每一个Follower服务器在接收到这个事务Proposal之后，都会首先以事务日志形式写到本地磁盘中，并且在写入成功后反馈给Leader服务器一个Ack响应。Leader服务器收到超过半数的ACK响应，就会广播一个Commit消息给所有Follower服务器通知其进行事务提交（Follower对事务进行提交），同时Leader也会完成对事务的提交。

#### zxid

事务Proposal的标识号：高32位是epoch,低32位是count

#### 完成同步后leader更新epoch，在各个follower上以（epoch + 1, 0）为zxid写一条更新操作（NEW\_LEADER)，然后就开始提供服务