随着Kubernetes成为容器编排领域霸主，etcd也越来越火热。

etcd 作为 Kubernetes的首要数据存储，它存储和复制所有的Kubernetes集群状态。

作为Kubernetes 集群的关键组件，因此必须要有一种**可靠的**方法来进行配置和管理。

ETCD是什么

官方解释：用于存储分布式系统中最关键数据的**分布式、可靠**的**键值存储**。

分布式:

多节点协调，部分节点down掉不影响整个系统运行。---可用性

可靠性:

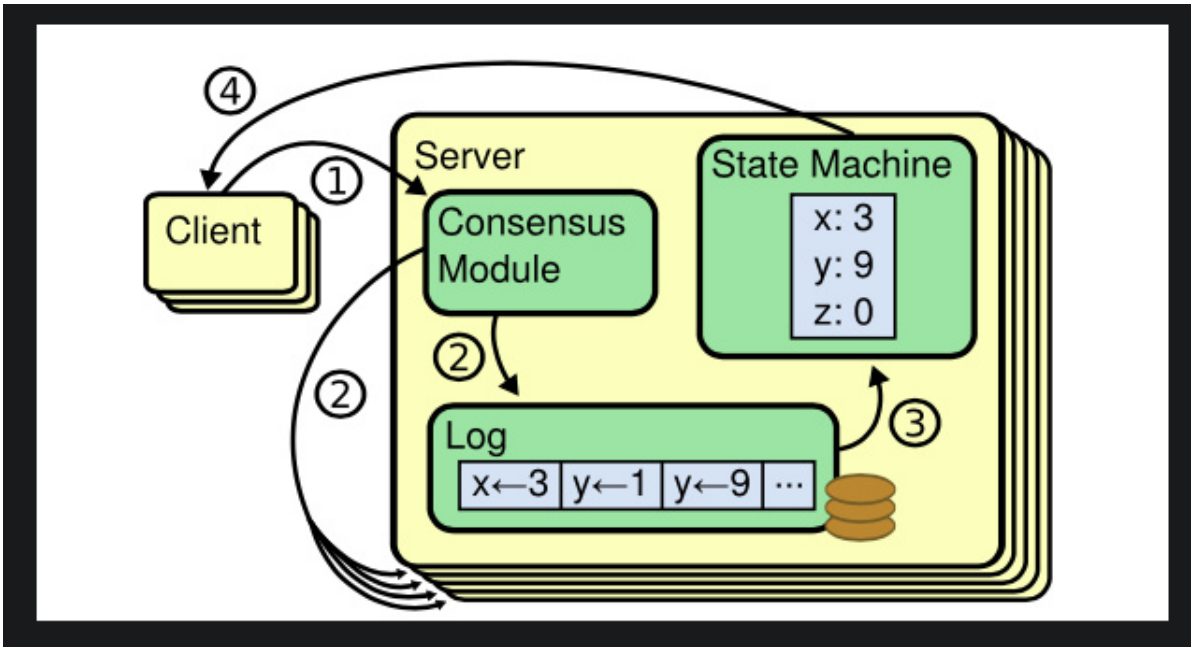
一旦一个写入被确认，后续的所有读取请求都应该能够得到这个值。---一致性

RAFT是什么

如何实现分布式可靠?

答案就是**共识算法**。

RAFT共识算法由共识模块、日志模块、状态机组成。通过共识模块保证各个节点日志的一致性，然后各个节点基于同样的日志、顺序执行指令，最终各个复制状态机的结果实现一致



etcd 的核心原理就是RAFT分布式共识：

- Leader选举，Leader故障后集群能快速选出新Leader；
- 日志复制， 集群只有Leader能写入日志， Leader负责复制日志到Follower节点，并强制Follower节点与自己保持相同；
- 安全性，一个任期内集群只能产生一个Leader、已提交的日志条目在发生Leader选举时，一定会存在更高任期的新Leader日志中、各个节点的状态机应用的任意位置的日志条目内容应一样等。

分布式共识可视化原理

== web ==

raft可视化模拟操作

1.模拟选主

2.模拟日止同步提交，复制虚框，提交实框

3.模拟主宕机X-node，term递增，集群重选主

4.模拟剩余集群运行状态

5.模拟恢复X-node

6.模拟分区，3nodes 宕机

回顾

- Follower，跟随者， 同步从Leader收到的日志，默认为此状态；
- Candidate，竞选者，可以发起Leader选举；(心跳超时)
- Leader，集群领导者， 唯一性，拥有同步日志的特权，需定时广播心跳给Follower节点，以维持领导者身份。
- 法定票数 quorum 一半以上
- 集群最好奇数
- 任期号term单调递增
- 进入Candidate状态的节点，会立即发起选举流程，自增任期号，投票给自己，并向其他节点发送竞选Leader投票消息（MsgVote）
- Raft日志条目同步（两阶段提交）

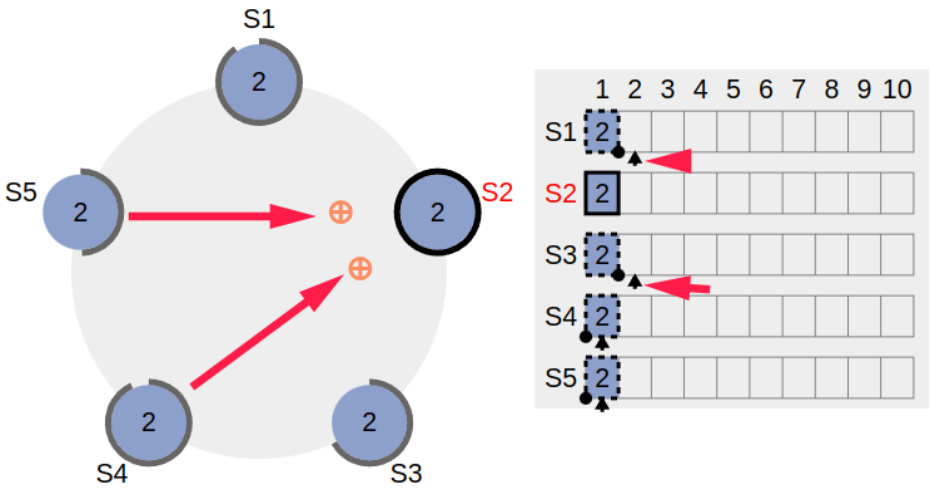
Raft日志条目同步

第一阶段：日志条目复制

1. **客户端请求**：客户端向 Leader 发送写入请求。
2. **Leader 处理**：Leader 将请求封装成一个日志条目，并将其追加到自己的本地日志中。
3. **日志条目发送**：Leader 将这个日志条目发送给所有的 Follower 节点。
4. **Follower 确认**：每个 Follower 收到日志条目后，将其追加到自己的本地日志中，并向 Leader 发送确认消息。

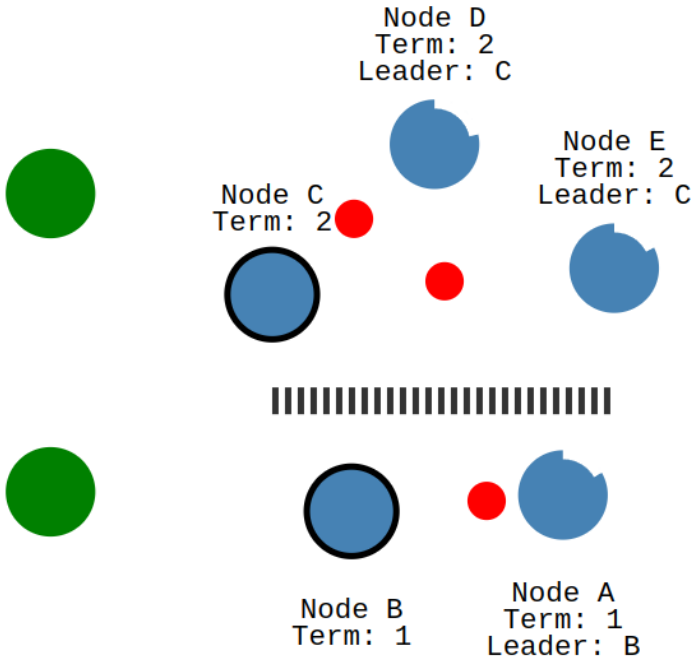
第二阶段：日志条目提交

1. **多数派确认**：Leader 收集来自 Follower 的确认消息。一旦收到来自大多数 Follower 的确认，Leader 就认为这个日志条目已经被大多数节点接受。
2. **提交日志条目**：Leader 提交这个日志条目，并应用到状态机中，更新其本地状态。
3. **通知 Follower**：Leader 向所有 Follower 发送提交消息，通知它们这个日志条目已经被提交。
4. **Follower 应用日志条目**：Follower 收到提交消息后，也会应用这个日志条目到它们的状态机中，更新本地状态。
5. **响应客户端**：Leader 向客户端返回成功响应。



为什么提交分为两个阶段?

如果允许 Follower 直接提交日志条目，那么在网络分区或其他故障情况下，可能会出现多个 Leader 同时存在的情况（**即“脑裂”**）。每个 Leader 都可能提交不同的日志条目，从而导致数据不一致。



可能会出现多个 Leader 同时存在的情况（即“脑裂”）,但数据不会错乱。某种意义上解决了脑裂。

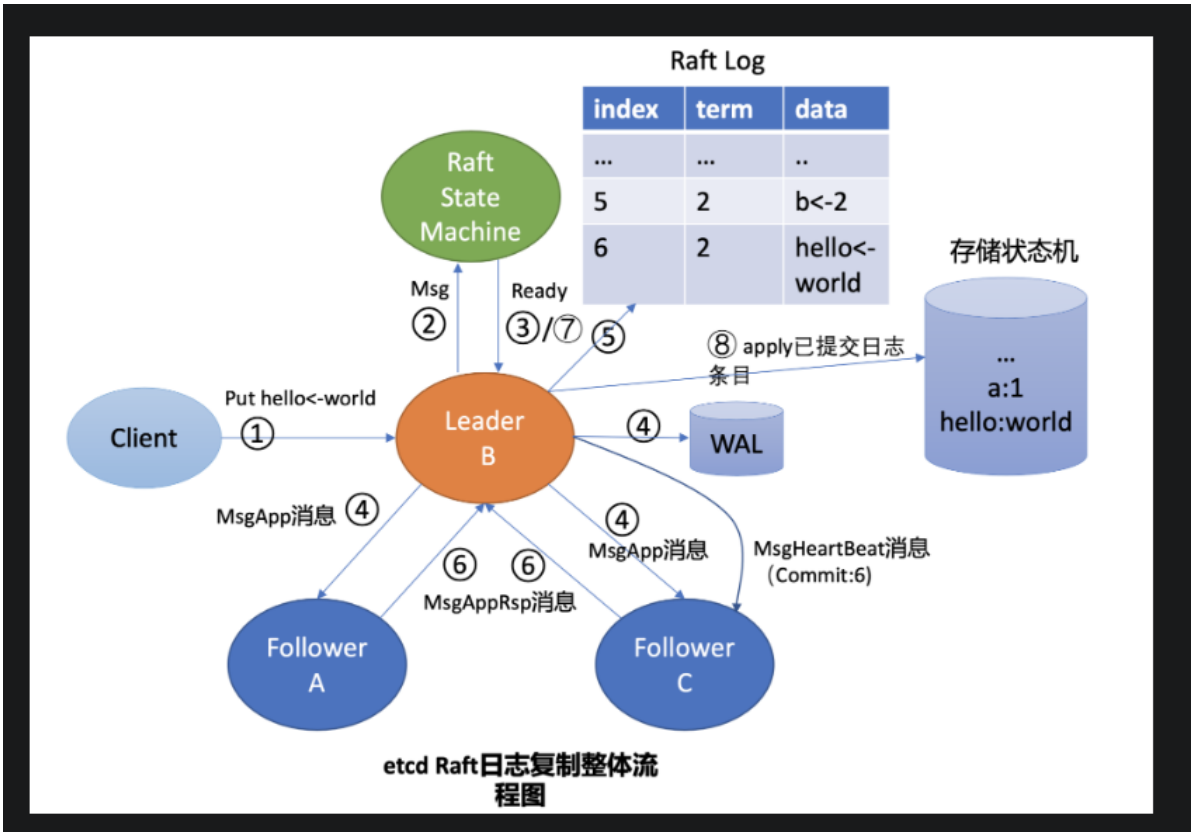
总结:

RAFT**关键特性**:

- **领导选举**：当现有的领导者失效时，系统会选举出一个新的领导者。
- **日志复制**：领导者负责将客户端请求转换为日志条目，并将这些条目复制到跟随者（Follower）节点上。
- **分区容忍**：Raft 能够处理网络分区的情况，尽管在网络分区期间可能会暂时失去可用性。

ETCD RUNNING ON RAFT

etcd使用Write-Ahead Log预写日志 (WAL)，是其数据持久化机制的关键部分，**主要用于确保数据的完整性和一致性**，在任何数据修改操作之前，etcd 会先将这些修改记录到 WAL 文件中。这样即使在写入过程中发生故障（如服务器崩溃），也可以通过回放 WAL 日志来恢复数据。

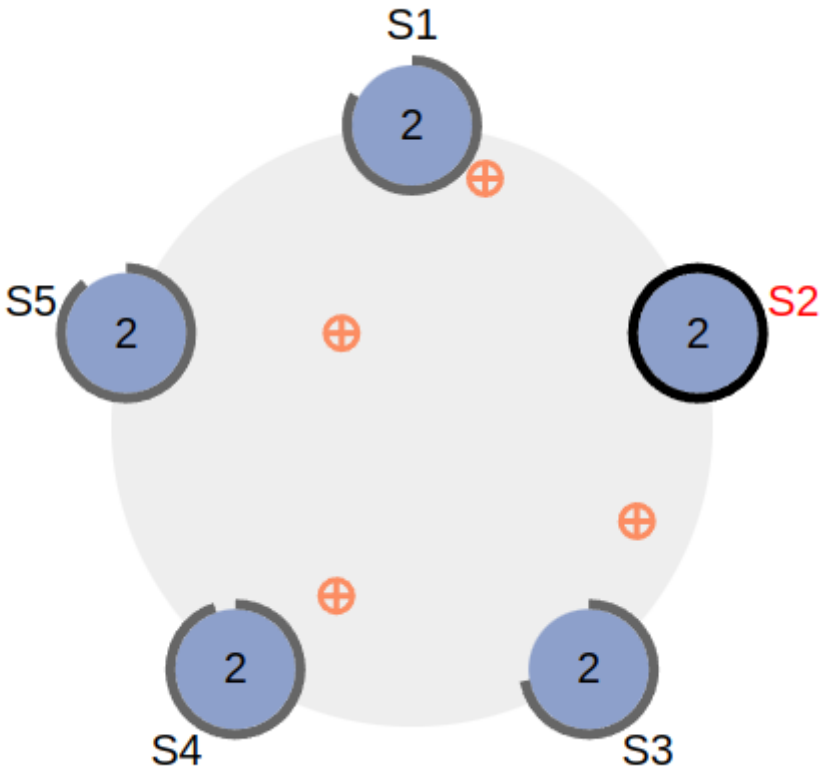


发散:

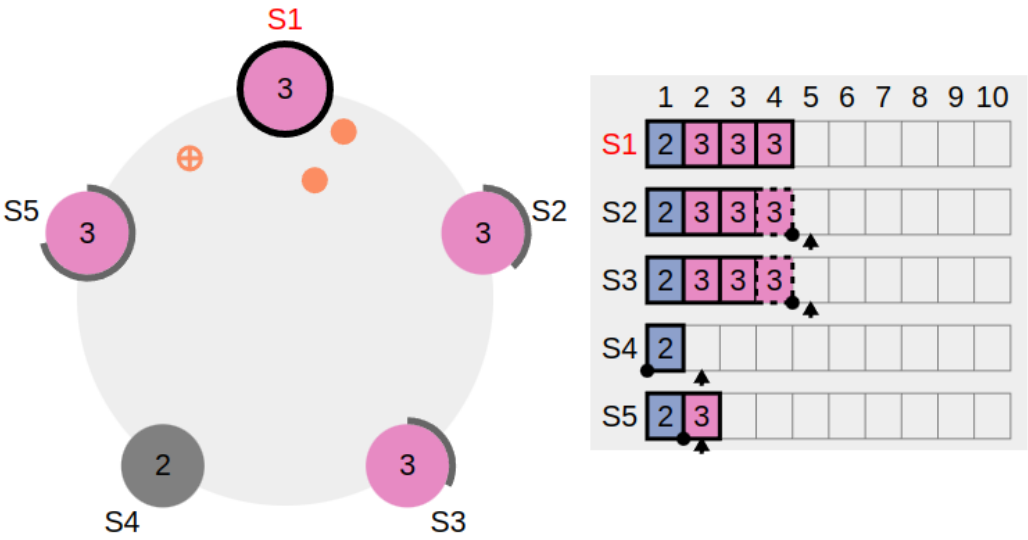
mysql的redo log 也是使用的wal机制。Write-Ahead Log（WAL）并不是etcd专属。

投票选举规则

- etcd默认心跳间隔时间（heartbeat-interval）是100ms， 默认竞选超时时间（election timeout）是1000ms，为了优化选票被瓜分导致选举失败的问题，引入了随机数，每个节点等待发起选举的时间点不一致，优雅的解决了潜在的竞选活锁，同时易于理解。



- 当节点收到选举投票的时候，需检查候选者的最后一条日志中的任期号，若小于自己则拒绝投票。



- 如果任期号相同，日志却比自己短，也拒绝为其投票。

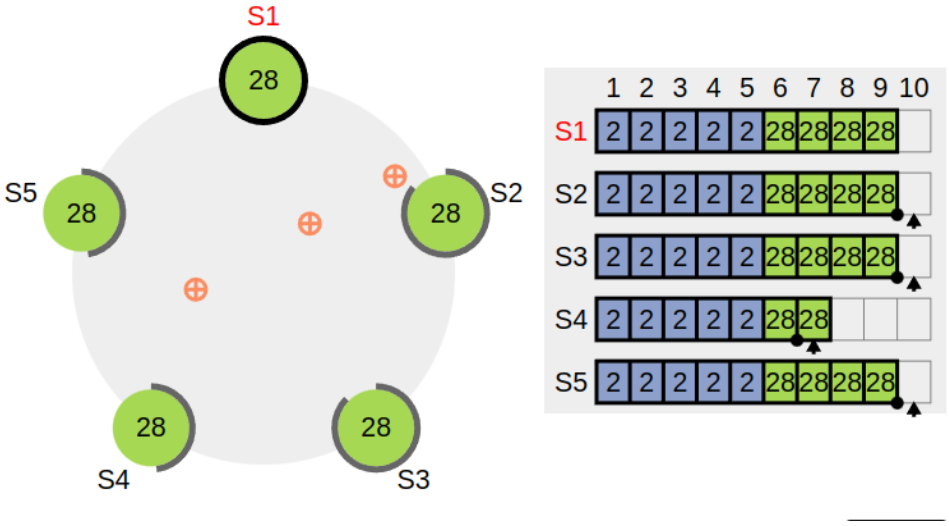
归纳：

客户端是否必须向 etcd 领导者发送请求?

Raft是基于领导者的；领导者处理所有需要集群共识的客户端请求。

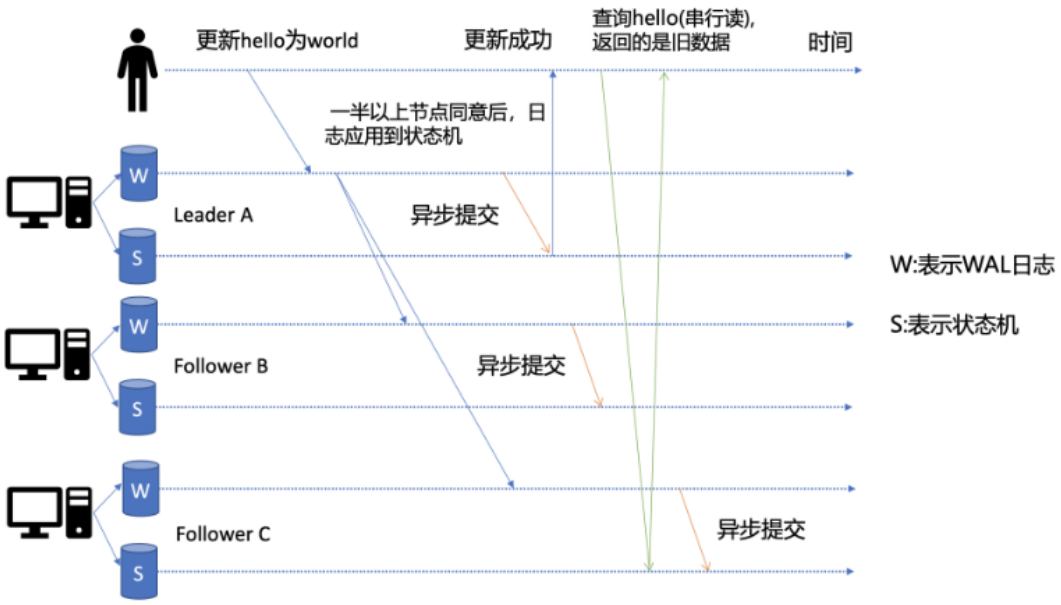
但是，客户端不需要知道哪个节点是领导者。任何需要共识的请求都会自动转发给领导者。

不需要共识的请求（例如序列化读取）可以由任何集群成员处理。



这会不会导致非强一致性?

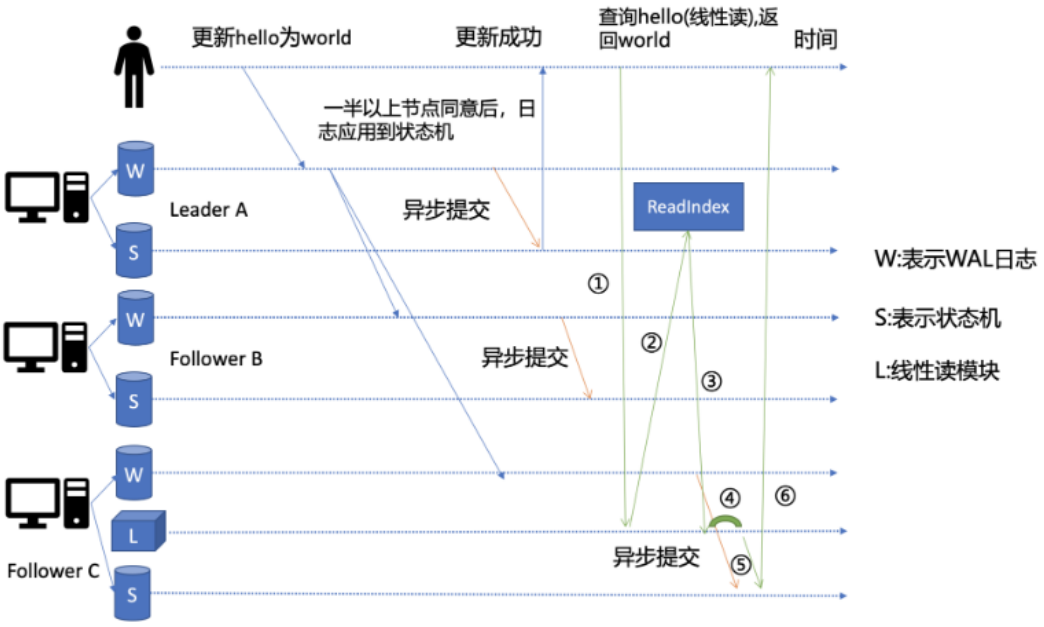
序列化读取 Serializable



client发起一个读取hello的请求，假设此请求直接从状态机中读取， 如果连接到的是C节点，若C节点磁盘I/O出现波动，可能导致它应用已提交的日志条目很慢，则会出现更新hello为world的写命令，在client读hello的时候还未被提交到状态机，因此就可能读取到旧数据。

线性化读取 Linearizable

线性化读取可以理解一旦一个值更新成功，随后任何通过线性读的client都能及时访问到。虽然集群中有多个节点，但client通过线性读就如访问一个节点一样。--强一致性



当收到一个线性读请求时，它首先会从Leader获取集群最新的已提交的日志索引(committed index)，如上图中的流程二所示。

Leader收到ReadIndex请求时，为防止脑裂等异常场景，会**向Follower节点发送心跳确认，一半以上节点确认Leader身份后才能将已提交的索引(committed index)返回给节点C(上图中的流程三)**。

C节点则会**等待**，直到状态机已应用索引(applied index)大于等于Leader的已提交索引时(committed Index)(上图中的流程四)，然后去通知读请求，数据已赶上Leader，你可以去状态机中访问数据了(上图中的流程五)。

源码实现大致流程走读：

```
546     }
547 }
548
549 func (n *node) ReadIndex(ctx context.Context, rctx []byte) error { //todo
550     return n.step(ctx, pb.Message{Type: pb.MsgReadIndex, Entries: []pb.Entry{{Data: rctx}}})
551 }
552
```

代码印证：最多不给答案（节点down），或者晚给答案，不会给错误答案。 ---可靠&强一致性体现

【即使 2 个服务器发生故障，5 个服务器组成的集群也可以继续运行。如果更多服务器发生故障（无主情况），它们将停止进展（但永远不会返回错误的结果）。】

两种读取模式性能对比

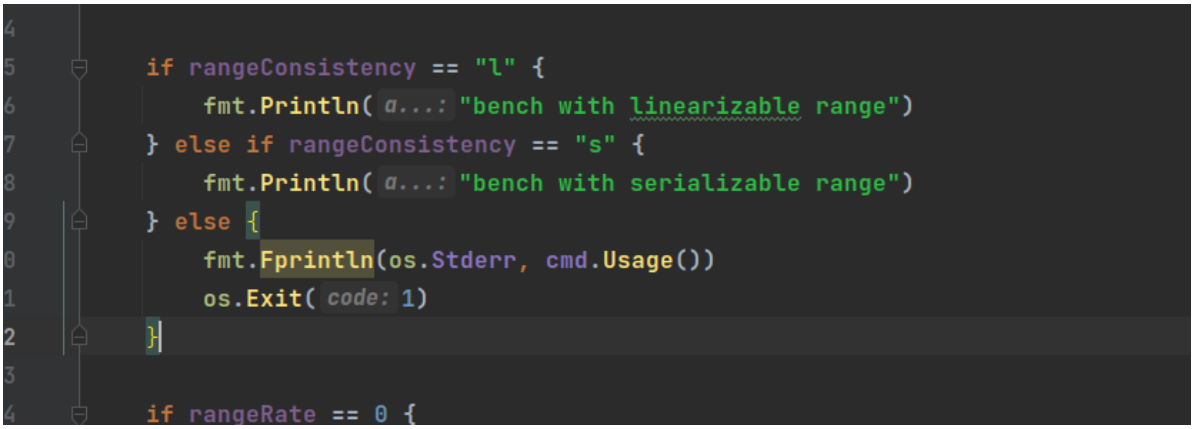
线性化读取请求通过集群成员仲裁来获取最新数据。序列化读取请求比线性化读取更便宜，因为它们由任何单个 etcd 成员提供服务，而不是由成员仲裁提供服务，以换取可能提供过时的数据。etcd 可以读取：

请求数	密钥大小（以字节为单位）	值的大小（以字节为单位）	连接数	客户端数量	一致性	平均读取 QPS	每个请求的平均延迟
10,000	8	256	1	1	线性化	1,353	0.7毫秒
10,000	8	256	1	1	可序列化	2,909	0.3毫秒
100,000	8	256	100	1000	线性化	141,578	5.5毫秒
100,000	8	256	100	1000	可序列化	185,758	2.2毫秒

示例命令为：

```
# Single connection read requests
benchmark --endpoints=${HOST_1},${HOST_2},${HOST_3} --conns=1 --clients=1 \
    range YOUR_KEY --consistency=l --total=10000
benchmark --endpoints=${HOST_1},${HOST_2},${HOST_3} --conns=1 --clients=1 \
    range YOUR_KEY --consistency=s --total=10000

# Many concurrent read requests
benchmark --endpoints=${HOST_1},${HOST_2},${HOST_3} --conns=100 --clients=1000 \
    range YOUR_KEY --consistency=l --total=100000
benchmark --endpoints=${HOST_1},${HOST_2},${HOST_3} --conns=100 --clients=1000 \
    range YOUR_KEY --consistency=s --total=100000
```

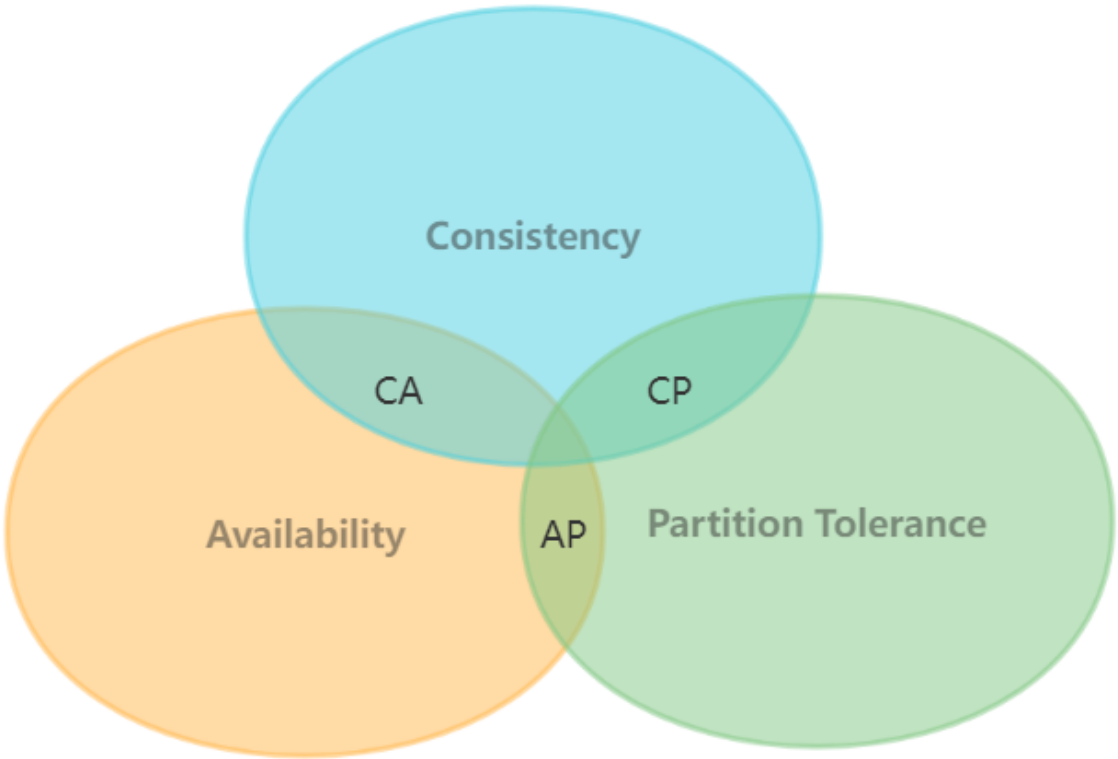


当然，etcd默认使用的是 **线性化读取（Linearizable Reads）**，保证了**强一致性**。

CAP理论

在分布式计算系统中，任何网络化的共享数据系统最多只能同时满足以下三个属性中的两个：

- 一致性（Consistency）
- 可用性（Availability）
- 分区容忍性（Partition tolerance）



根据cap理论分析下mysql集群与etcd分别满足哪些属性

ETCD CP ---------牺牲A（可用性），在网络分区期间，etcd 集群中的一部分节点可能会变得不可用，直到分区恢复。保持数据的一致性远比短暂的不可用更加重要。（eg：K8S）

mysql全同步 C

mysql半同步/异步 AP

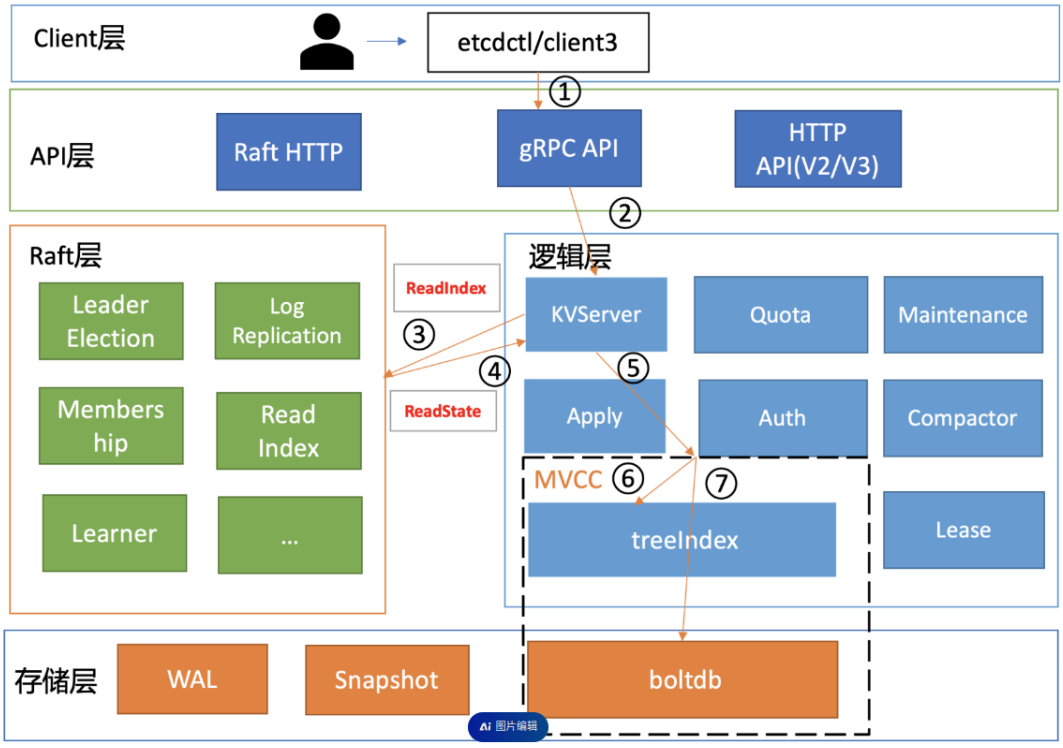
【mysql主从同步为了解决单点故障，从而引入了多副本。但基于复制算法实现的数据库，为了保证服务可用性，大多数提供的是最终一致性，即非强一致性。】

例外：mysql mgr 组复制， 基于 paxos协议（zookeeper使用的也是paxos协议） CP

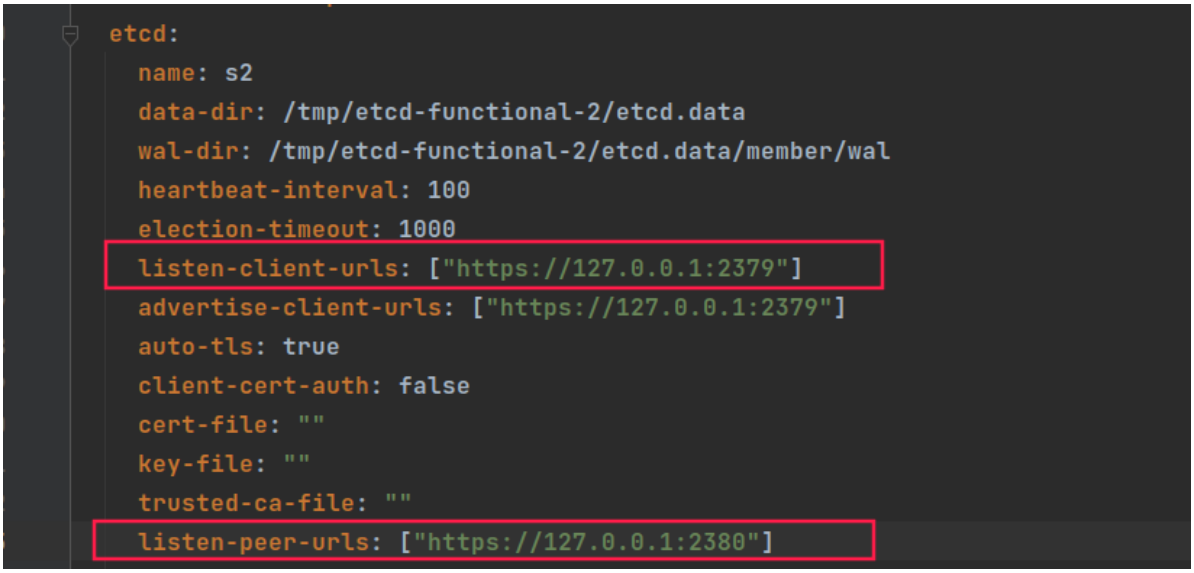
总结：分布式系统的话默认应该有P，最终都在CP与AP之间取舍。

更多引申

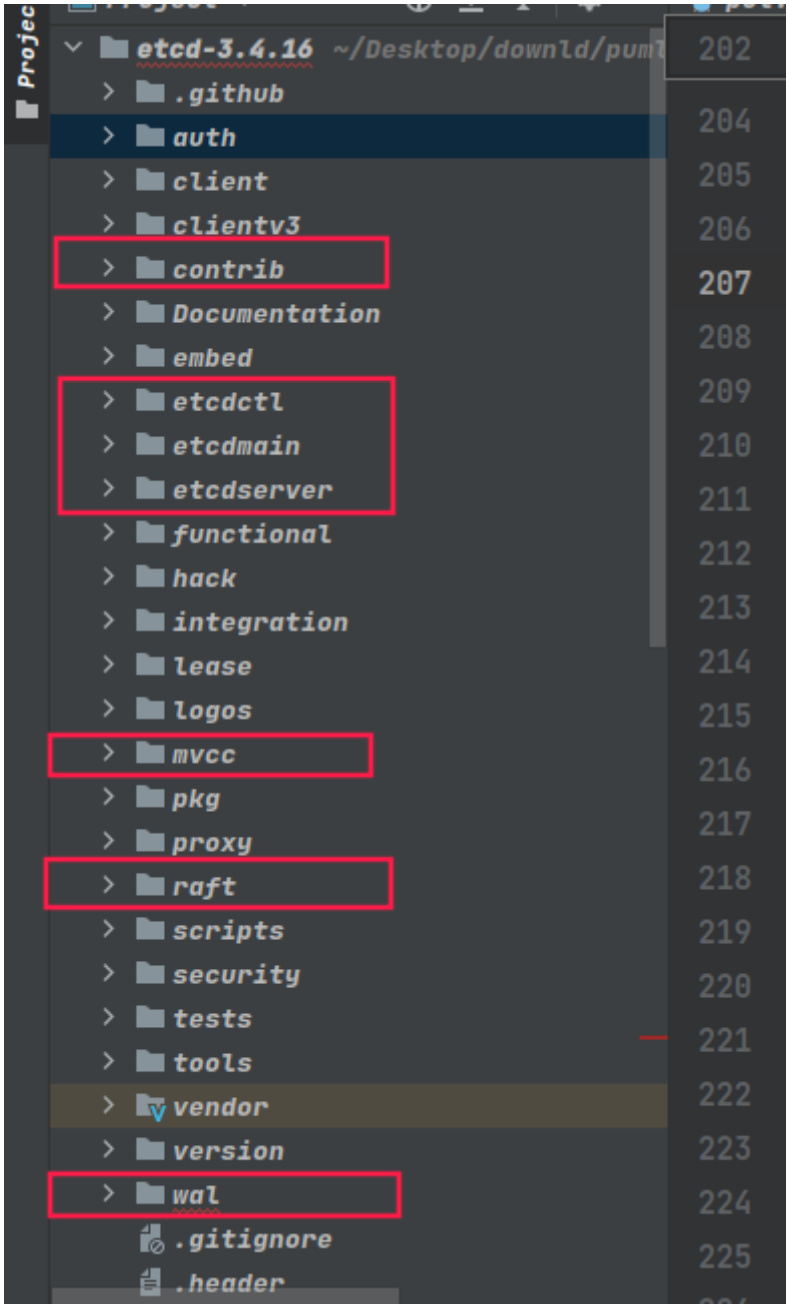
etcd架构



peer节点间基于grpc(http2.0)通信，默认端口 :2380，对外client服务支持grpc/http1.1 默认端口 :2379， **致敬redis 6379 6380**。多少有点致敬的意思。



其他特性，mvcc， lease ，服务注册发现watch等


etcd-3.4.16 ~/Desktop/downld/pum
.github
auth
client
clientv3
contrib
Documentation
embed
etcdctl
etcdmain
etcdserver
functional
hack
integration
lease
logos
mvcc
pkg
proxy
raft
scripts
security
tests
tools
vendor
version
wal
.gitignore
.header
202
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225