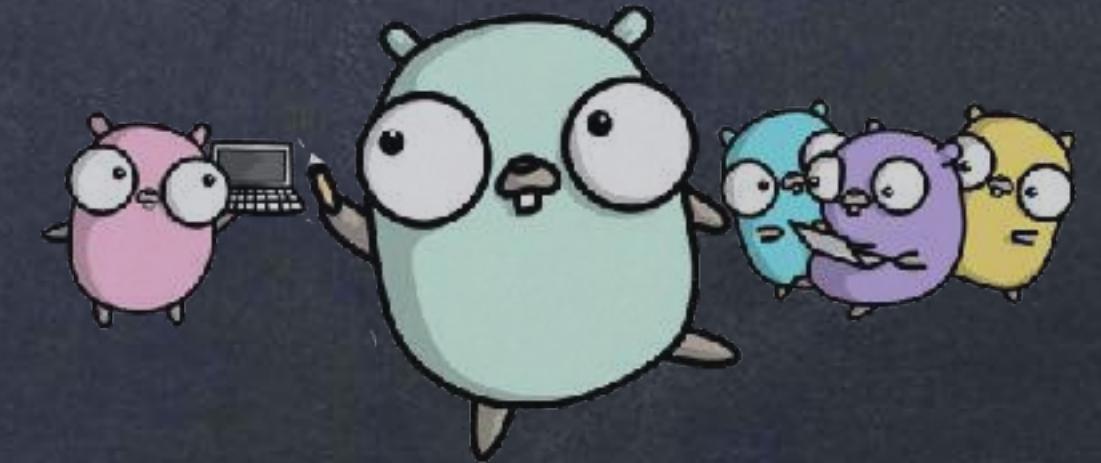


# Etcd的设计与实现

- [xiaorui.cc](http://xiaorui.cc)

- [github.com/rfyiamcool](https://github.com/rfyiamcool)

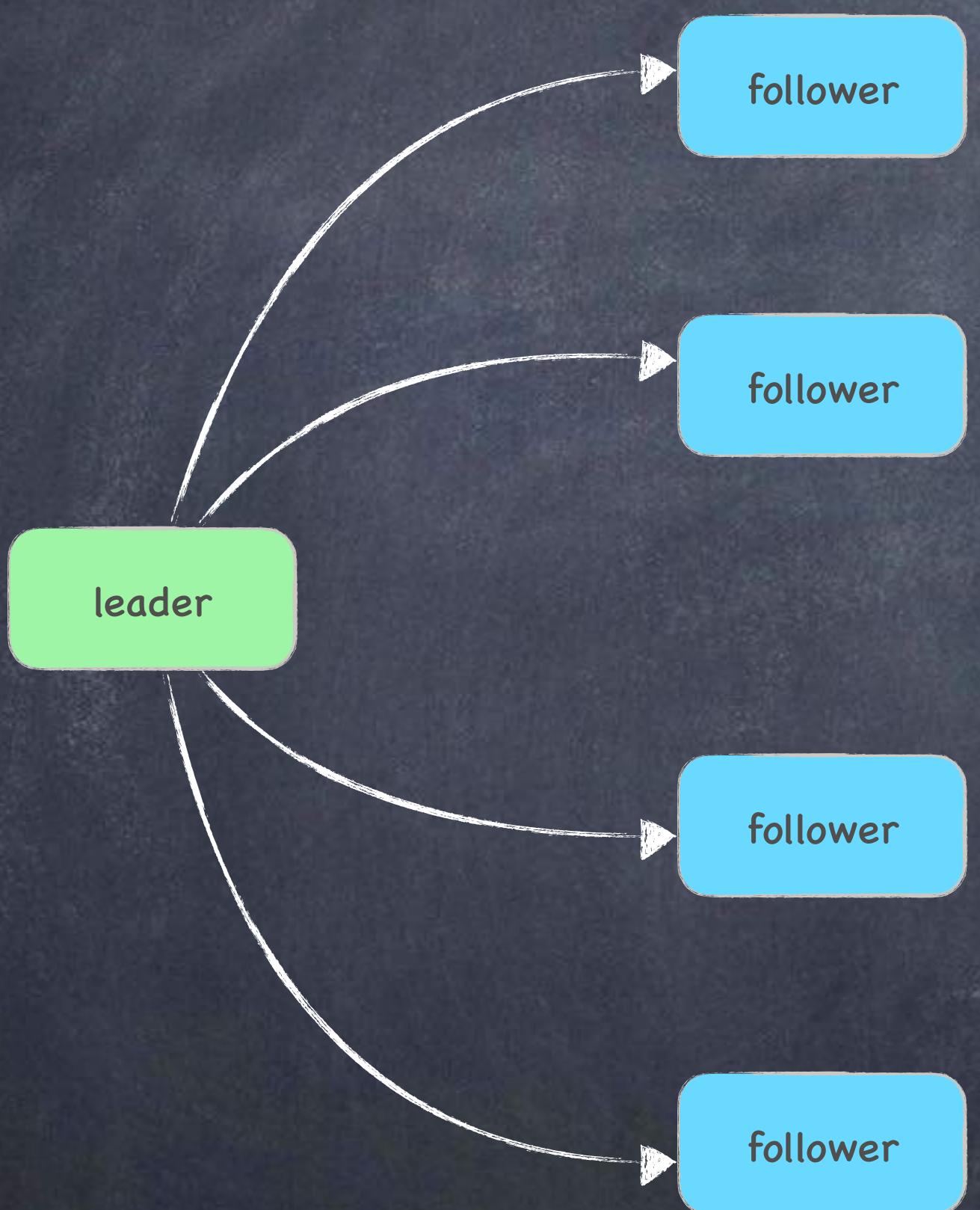




what is etcd ?



# what is etcd ?



etcd

分布式 kv 存储

- feature
- 元数据存储
- 服务发现
- 分布式锁
- 中间件协调
- ...

# different

④ etcd

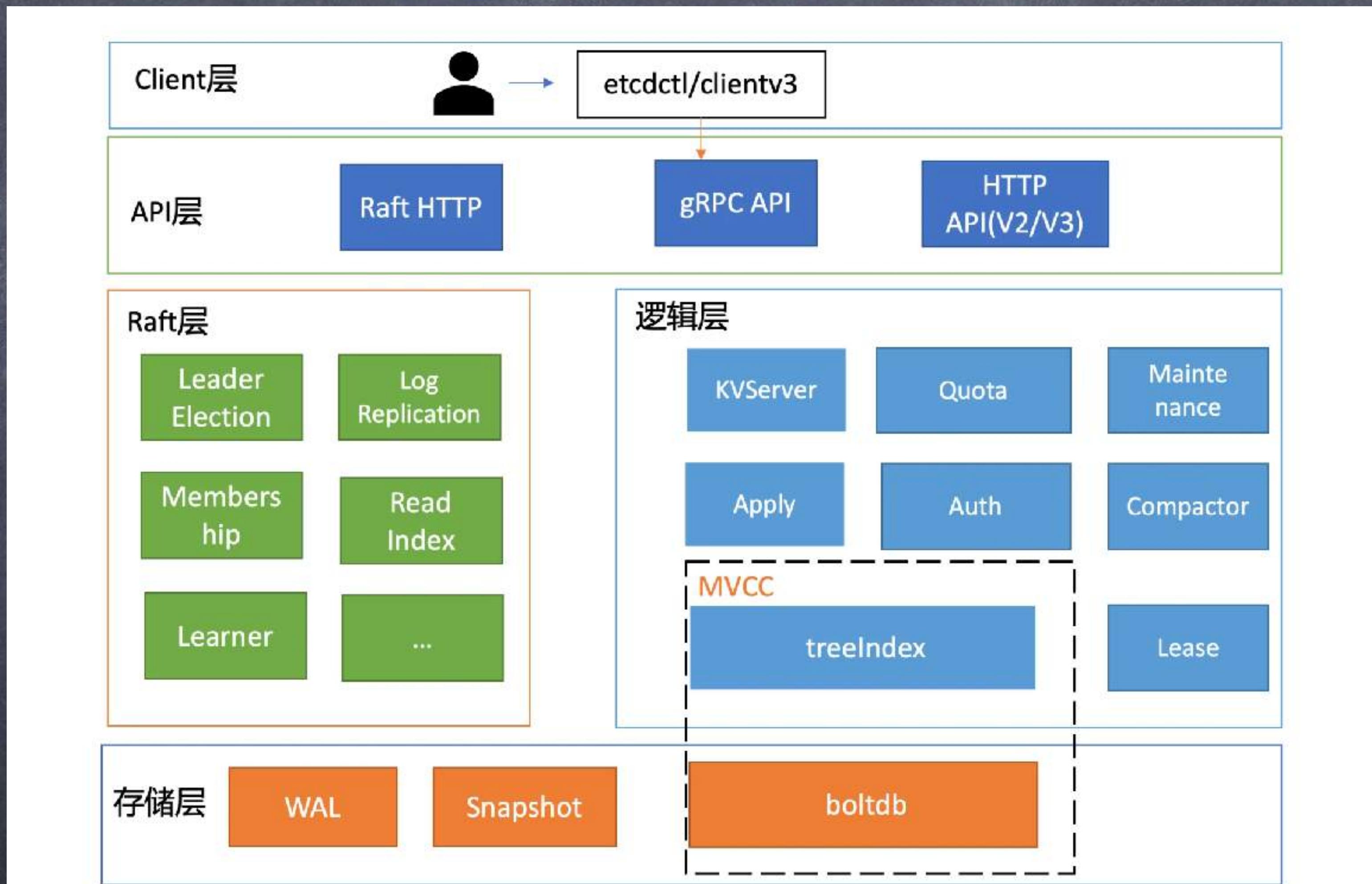
④ consul

④ zookeeper

④ other

		etcd	zookeeper	consul
etcd	线性读	支持	不支持	支持
consul	多版本并发控制	支持	不支持	不支持
zookeeper	事务	对比内容和版本	对比版本	对比内容和版本
other	更新通知	历史和当前键范围	当前键和目录	当前键(支持前缀)
other	HTTP	支持	不支持	支持
other	权限管理	rbac	acl	acl
other	多数据中心	不支持	不支持	支持

# etcd design

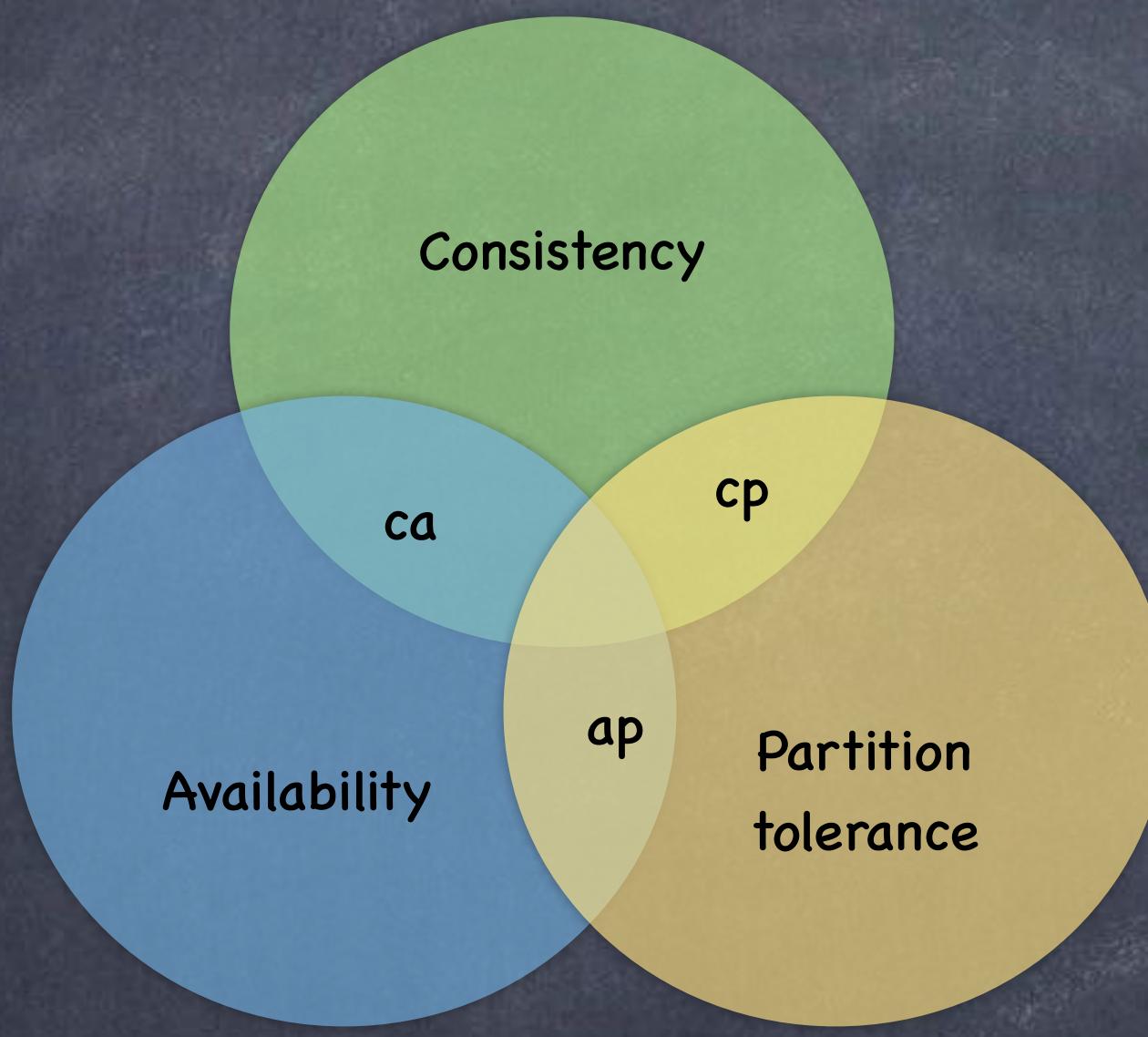




raft algo

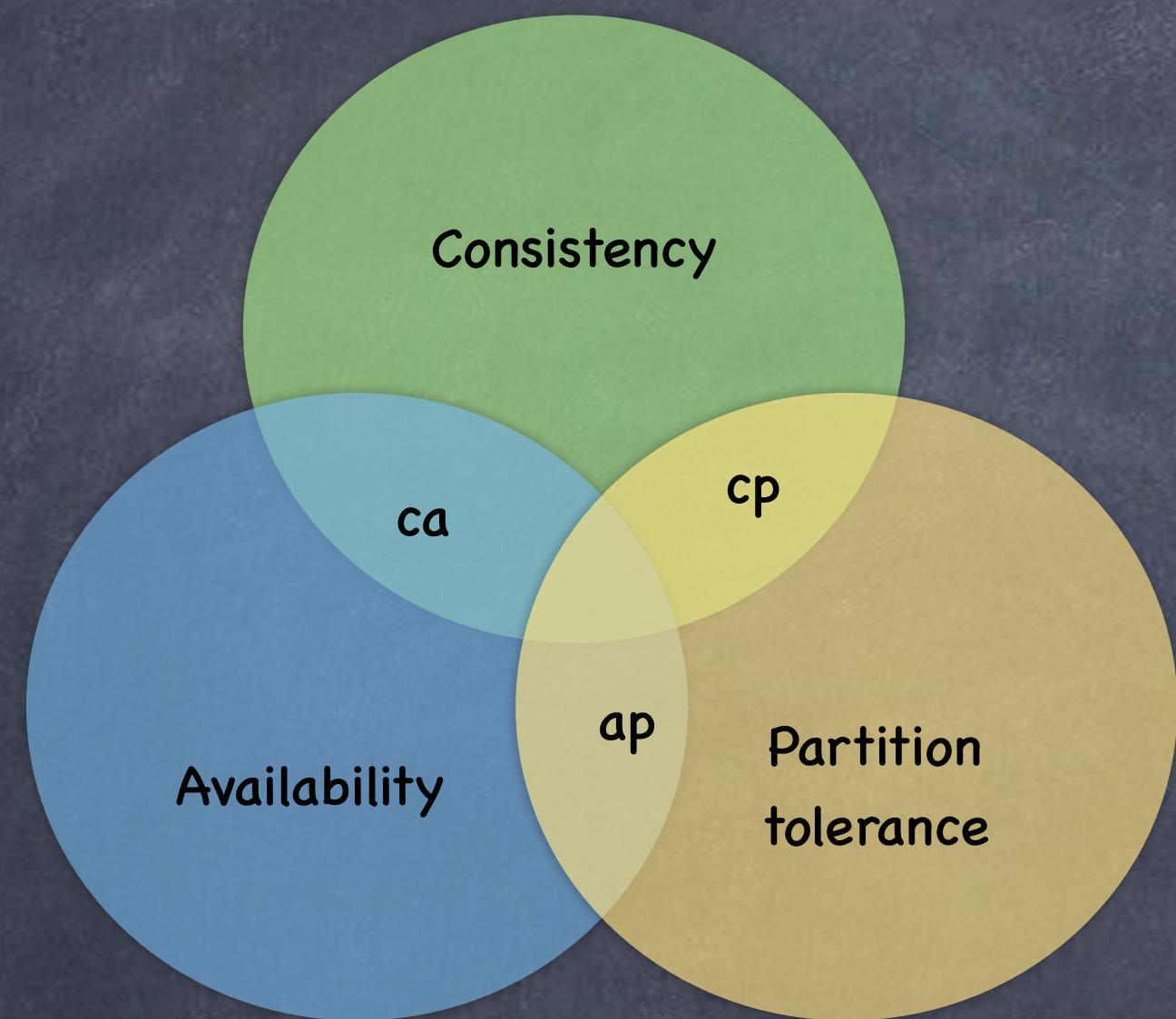


# cap 原则



- **consistency** (一致性)
  - 每个节点返回的数据是一致的；
  - 强一致性
  - 弱一致性（最终一致性）
- **availability** (可用性)
  - 非故障的节点在合理的时间内返回合理的响应；
- **partition tolerance** (分区容错性)
  - 当网络出现分区后，系统依然能够继续服务；

# cap 原则



分布式系统理论上不可能选择**CA**架构，而必须选择**CP**或**AP**架构。

## cp

- zookeeper, etcd, tikv
- 分布式 database 多为 cp

cp 也会尽量保证 a !!!

## ap

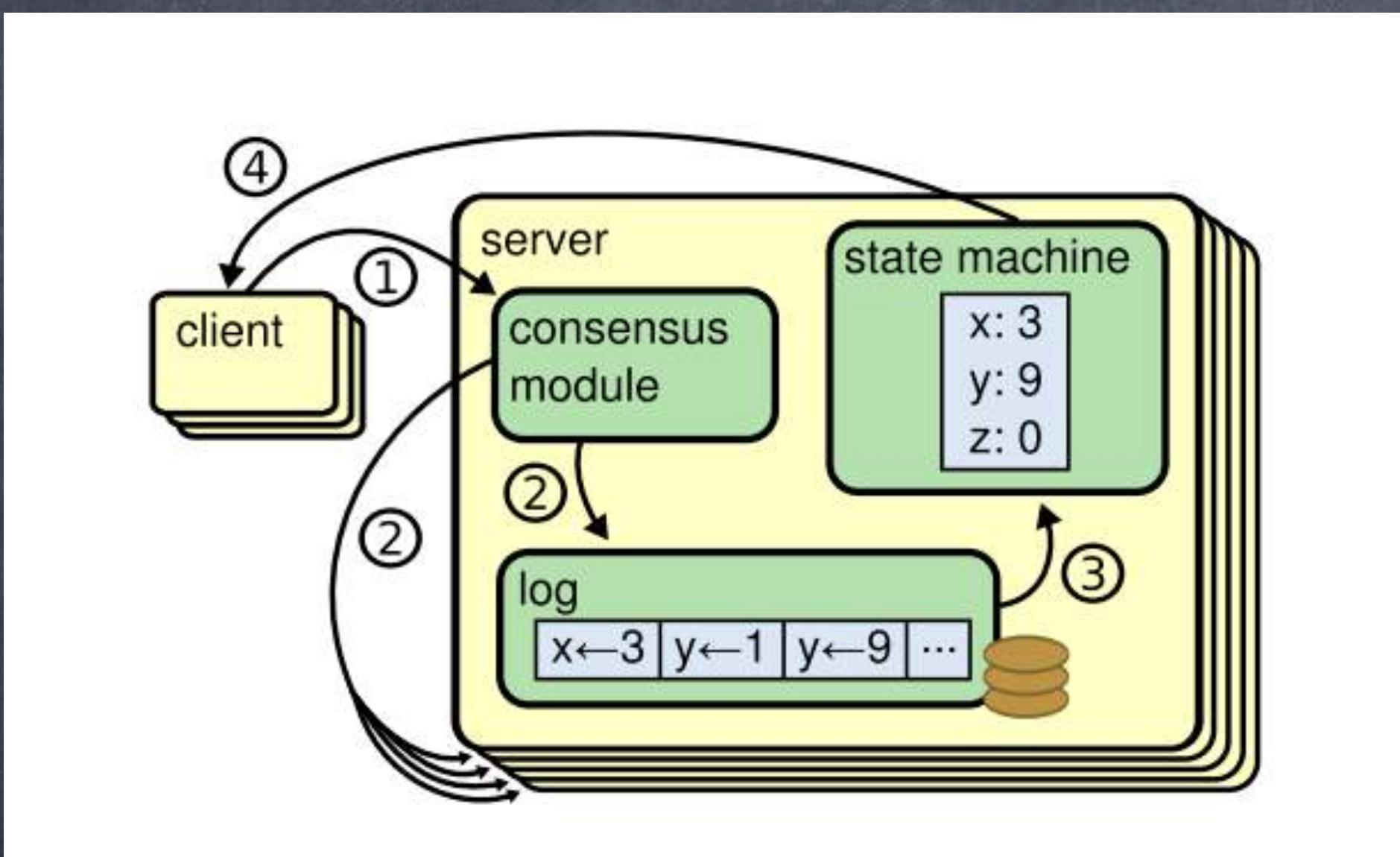
- eureka, cassandra, dynamodb
- 服务发现系统多为 ap

ap 也会尽量保证 c !!!

## ca ???

- 若发生分区现象，为了保证C，系统需要禁止写入，此时就与A发生冲突；
- 如果是为了保证A，则会出现正常的分区可以写入数据，有故障的分区不能写入数据，则与C就冲突了；

# raft design

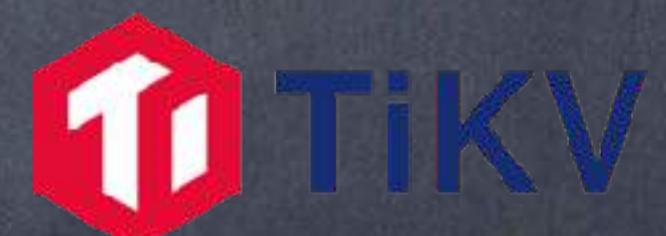


## feature

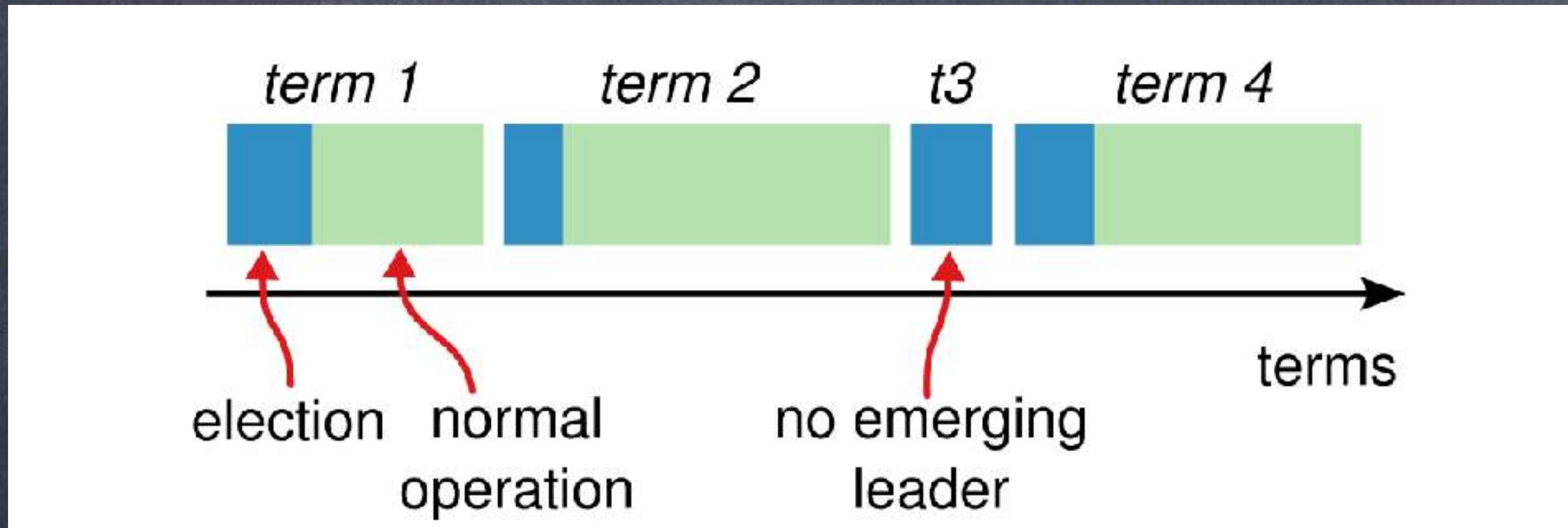
- Leader election
- Log replication
- Safety
- Membership change

## who

- etcd
- consul
- nacos
- tidb
- cockroachDB
- ...



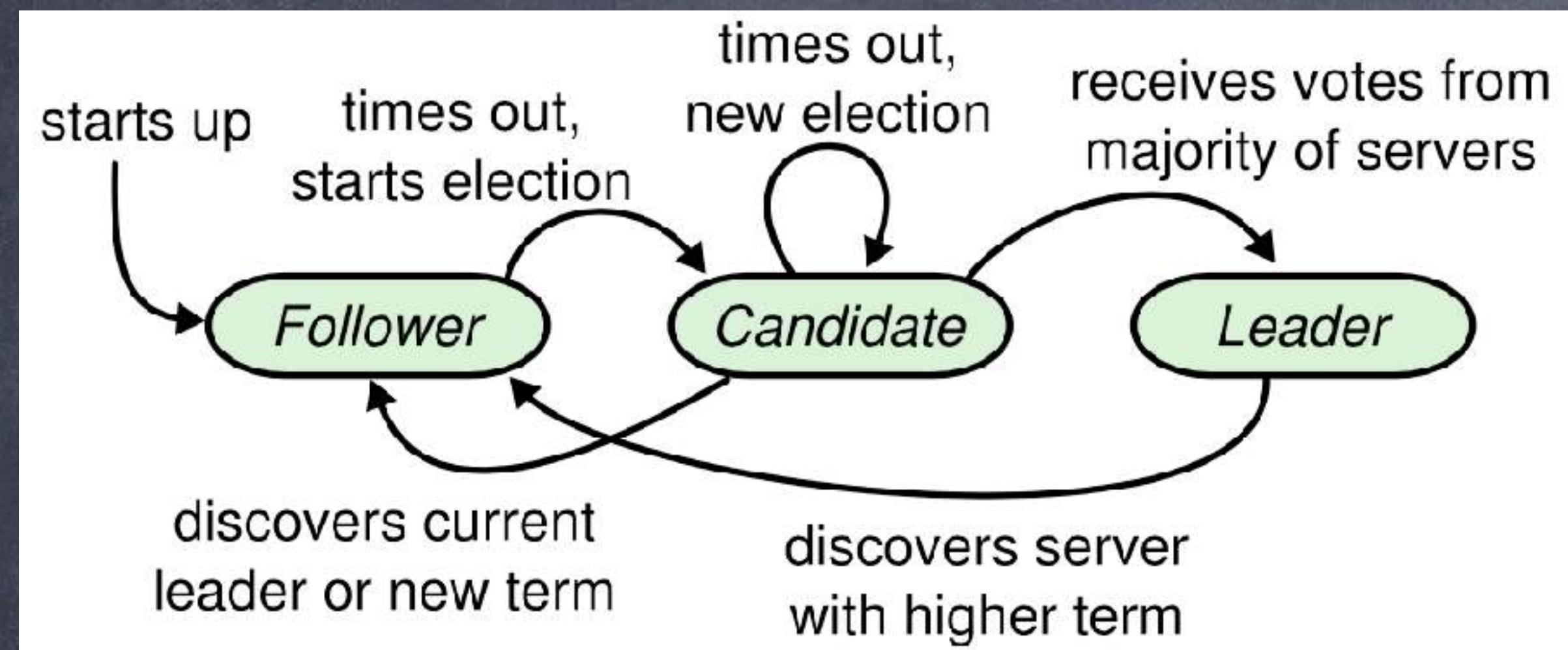
# term



- Raft 将时间划分为任意长度的任期 (**term**);
- Term 是 Raft 集群中的逻辑时钟;
- 如果一个节点的 **currentTerm** 小于对方，就会更新自己的 **currentTerm**;
- 如果 **candidate** 或者 **leader** 发现自己的 **term** 过期了，那么它会立即转换为 **follower** 状态；



# raft role



- **leader** 领导者

- 可读可写
- 处理日志复制
- 保证一致性
- 维护心跳

- **candidate** 候选者

- 集群的候选者，会发起投票试图当选**leader**；

- 通过 **RequestVote** 通知其他节点来投票；

- **follower** 跟随者

- 可串行读本地数据，也可力求一致性的线性读；

- 自身不能处理些请求，可**proxy**到**leader**处理；

- 参与投票，一个**term**任期内只能投一次；

- 当触发 **election timeout** 时，晋升为 **candidate**；

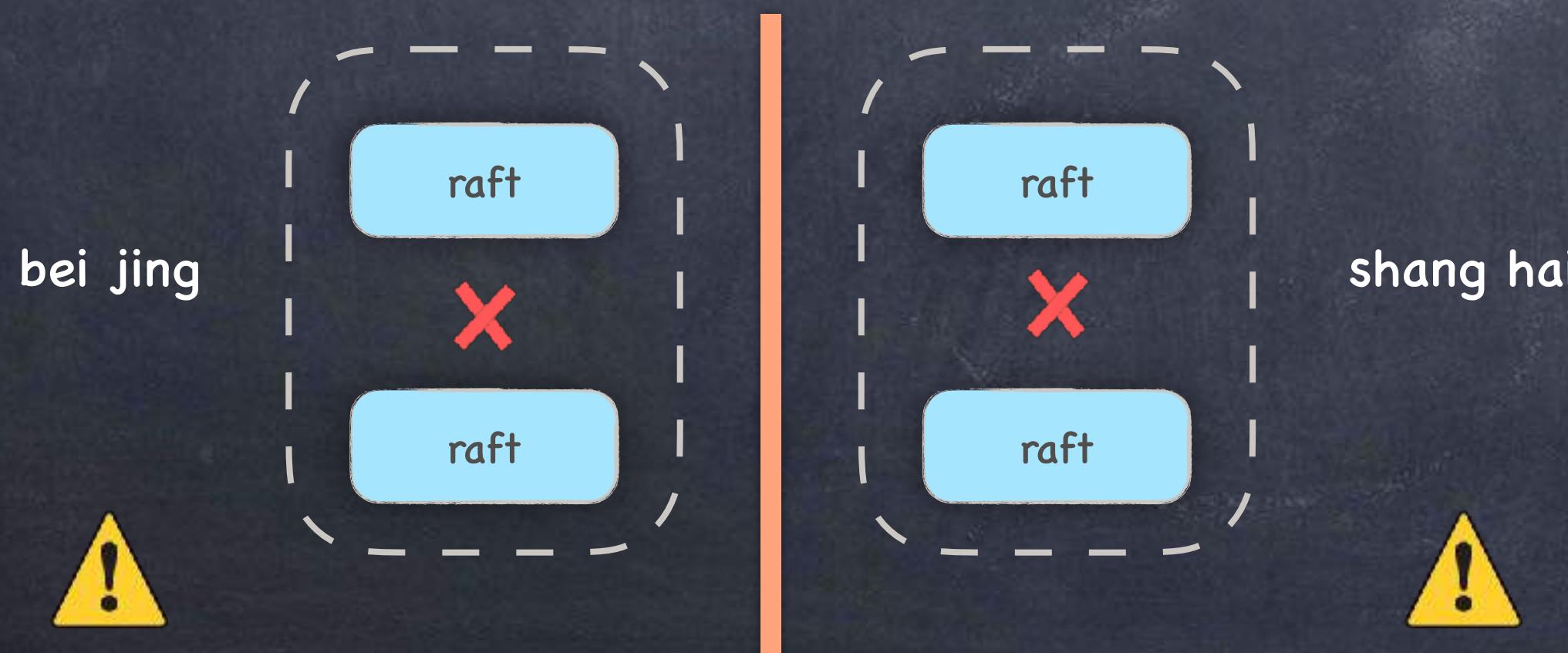


# quorum



total	quorum instance	tolerable instance
2	x	x
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
9	5	4
9	5	4
...	...	...

- ④ max quorum
- ④  $(n + 1) / 2$
- ④ 一般集群个数为 奇数；
- ④ 偶数的quorum的计算公式
- ④  $(n / 2) + 1$
- ④ 偶数会有哪些问题？
- ④ quorum 性价比不高；
- ④ 增加了 平票 概率，可进行再次重试选举；
- ④ 如 bj 和 sh 各两个实例，当网络故障后，两个集群都不可用 !!!



Quorum 法定数 是 大多数节点（比一半多点）



重点!

# leader election condition

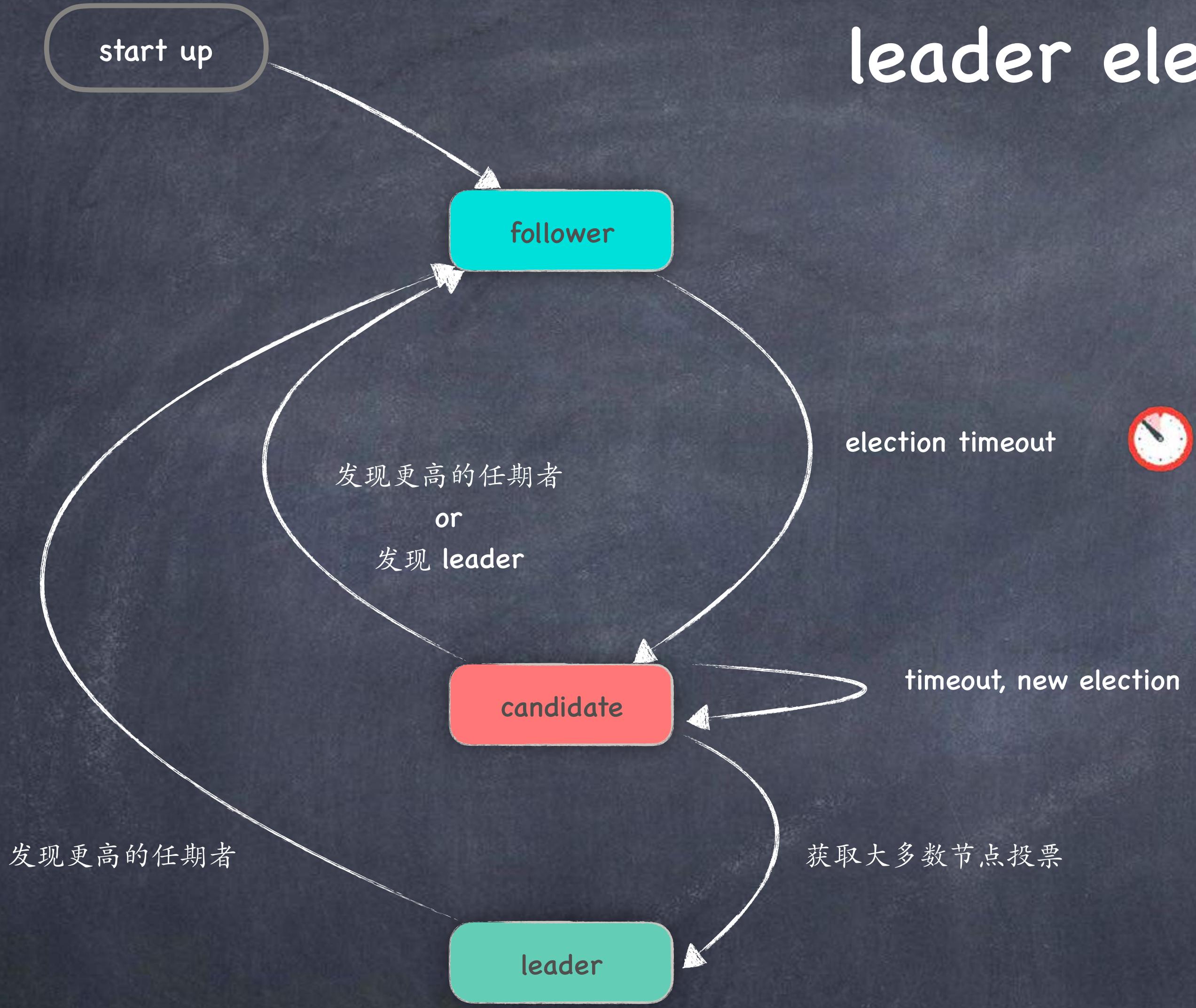
- 能成为 leader 的条件？

- 当前集群无可用 leader；
- 触发 election timeout；
- term 任期最新；
- log 日志最新；
- 获取多数投票；

RequestVote RPC	
Invoked by candidates to gather votes (§5.2).	
<b>Arguments:</b>	
term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)
<b>Results:</b>	
term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote
<b>Receiver implementation:</b>	
1. Reply false if term < currentTerm (§5.1)	
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)	

- candidate 首先毛遂自荐给自己投票；
- candidate 选举时需要携带 term 及日志信息；
- Term 和日志较新的节点才可以选举为 leader；
- follower 发现本地比 candidate 数据新则拒绝投票；
- 同一个 term 任期内只允许一次投票，先到先得；
- 一个 term 任期只有一个可用的 leader；
- 同时出现两个 candidate 且通知其他节点投票，满足  $(n + 1) / 2$  投票就可晋升 leader；
- 如偶数集群下两个 candidate 获取一样的投票数，谁也不让步，则进行等待超时重试；

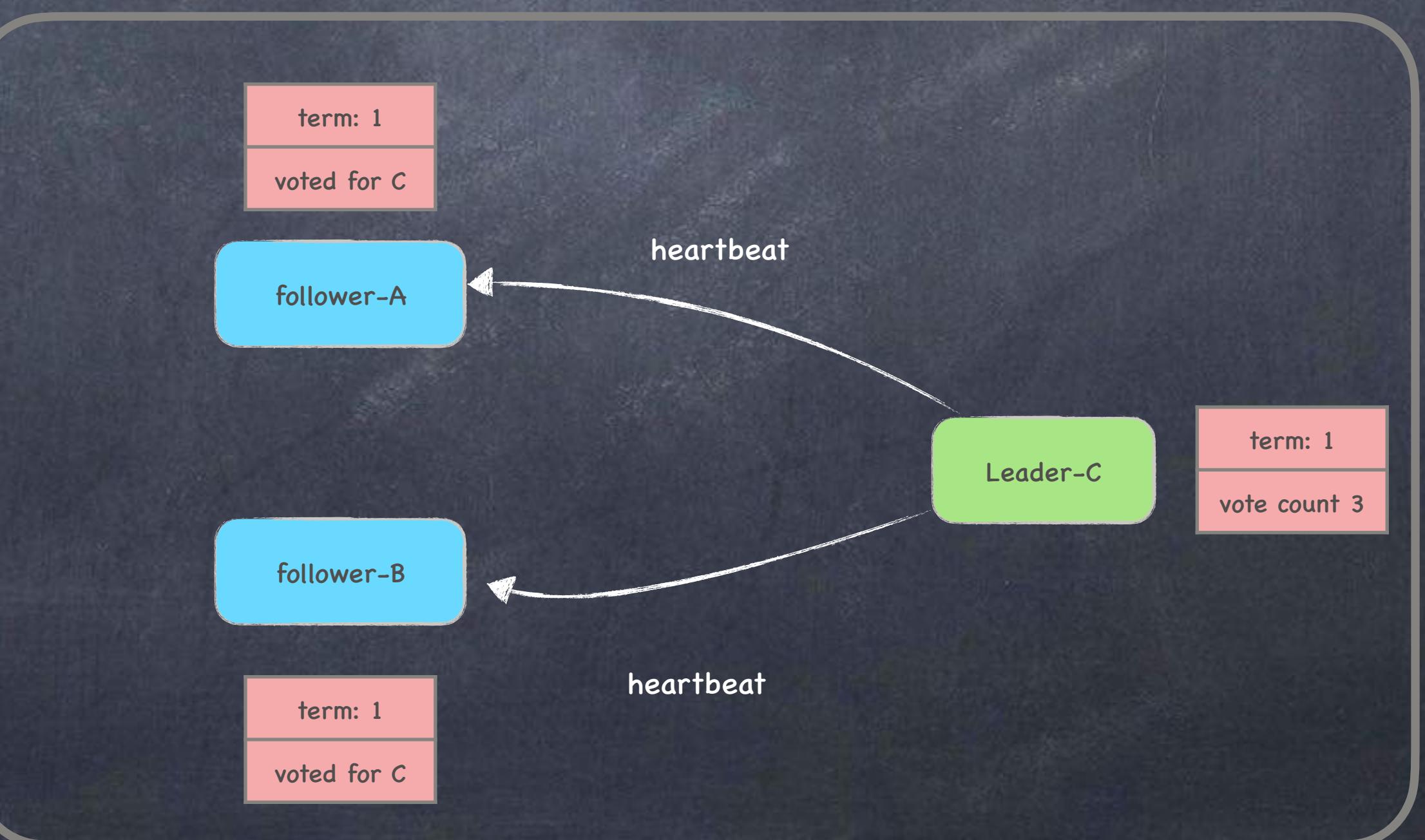
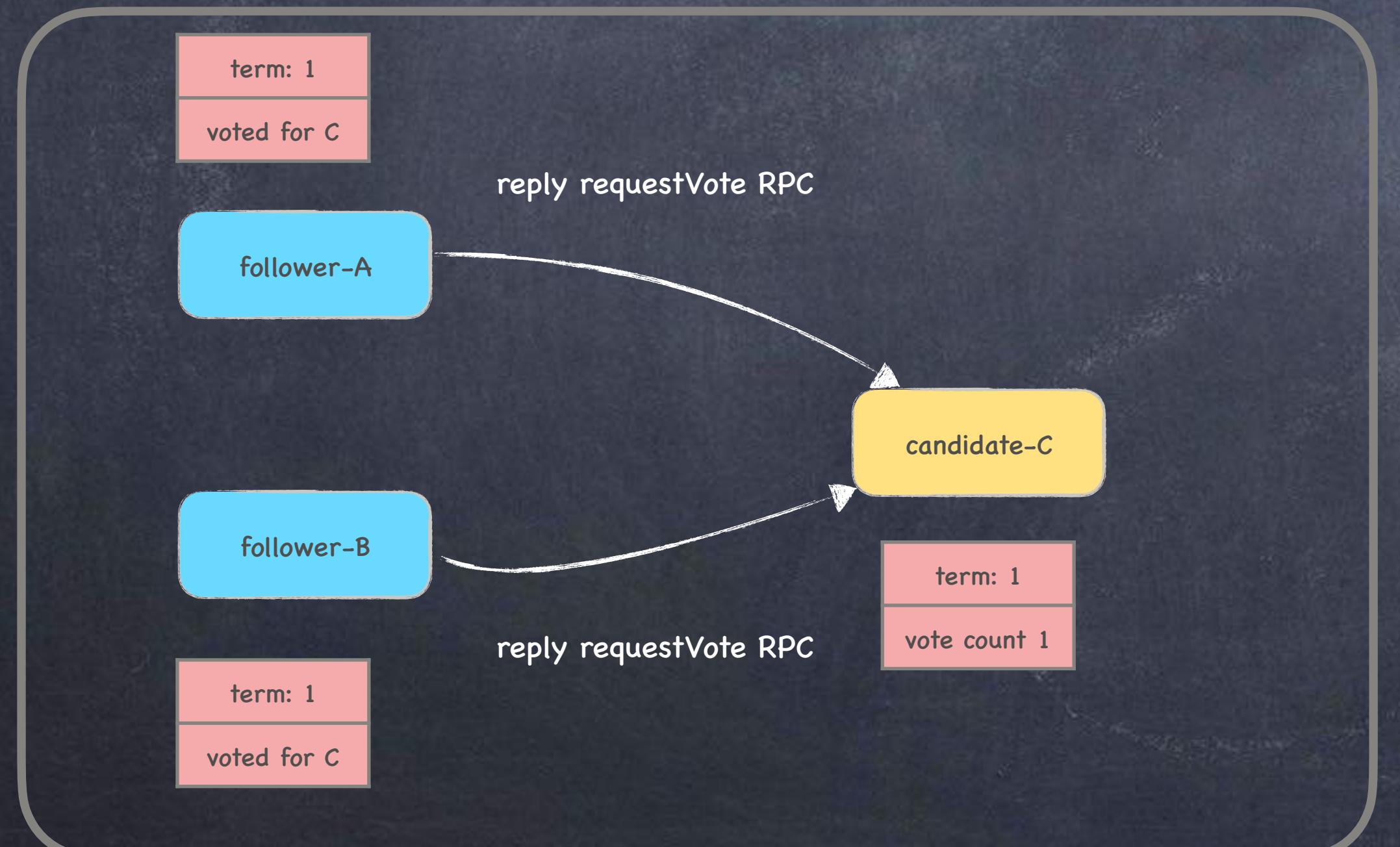
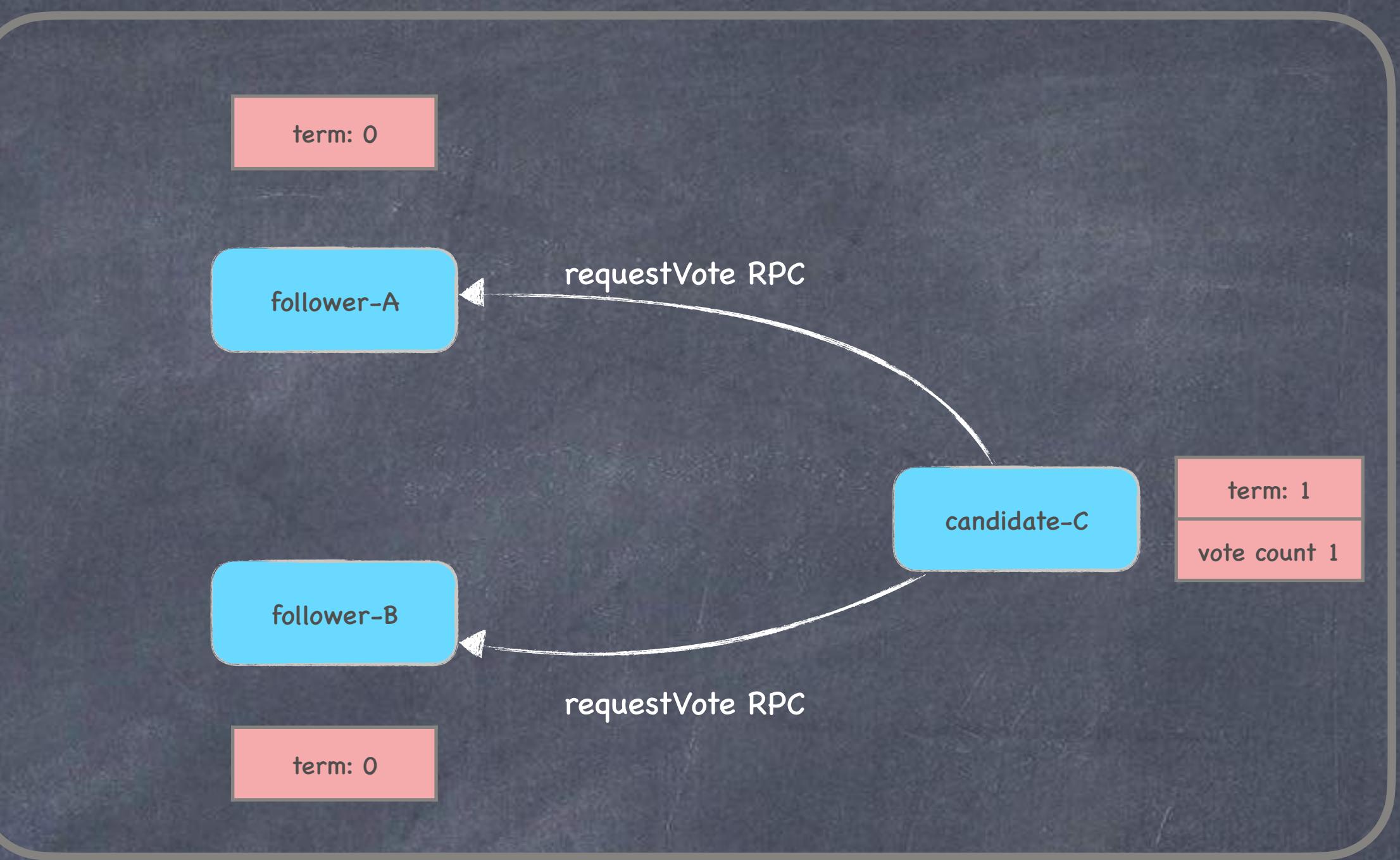
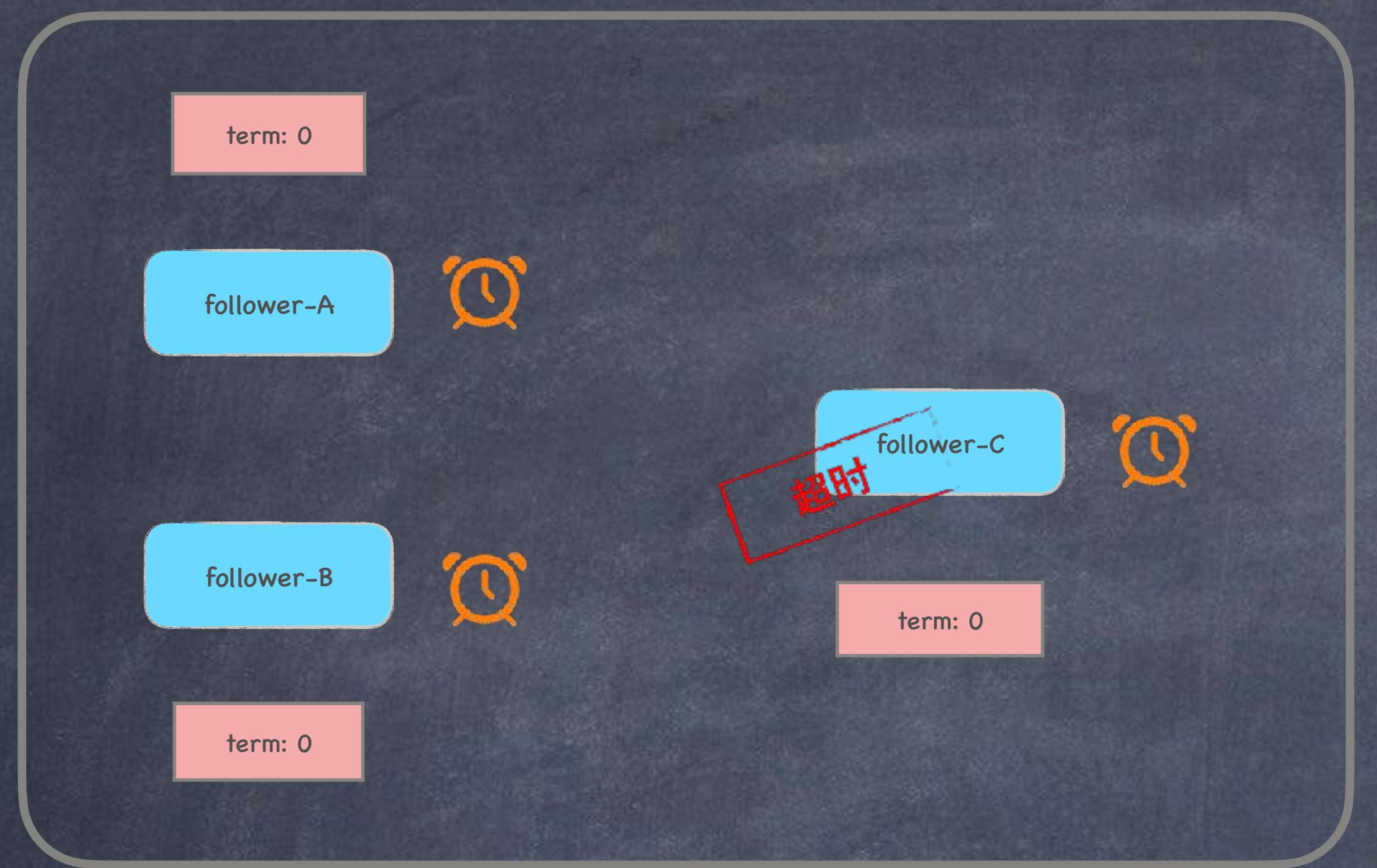
# leader election



- etcd default timeout:
  - heartbeat timeout = 100 ms
    - 发送心跳的超时时间；
  - Leader 给其他节点发送心跳的时间间隔；
- election timeout =  $1000 \text{ ms} + \text{jitter}$ 
  - 如长时间未收到心跳，且超过选举的超时时间
- follower 晋升为 candidate 进行选举；

角色转换图

<http://thesecretlivesofdata.com/raft/>



# leader election

如何尽量避免多个 **candidate** 的产生？

```
func (r *raft) resetRandomizedElectionTimeout() {
    r.randomizedElectionTimeout = r.electionTimeout + globalRand.Intr(r.electionTimeout)
}

func (m *member) ElectionTimeout() time.Duration {
    return time.Duration(m.s.Cfg.ElectionTicks*int(m.s.Cfg.TickMs)) * time.Millisecond
}
```

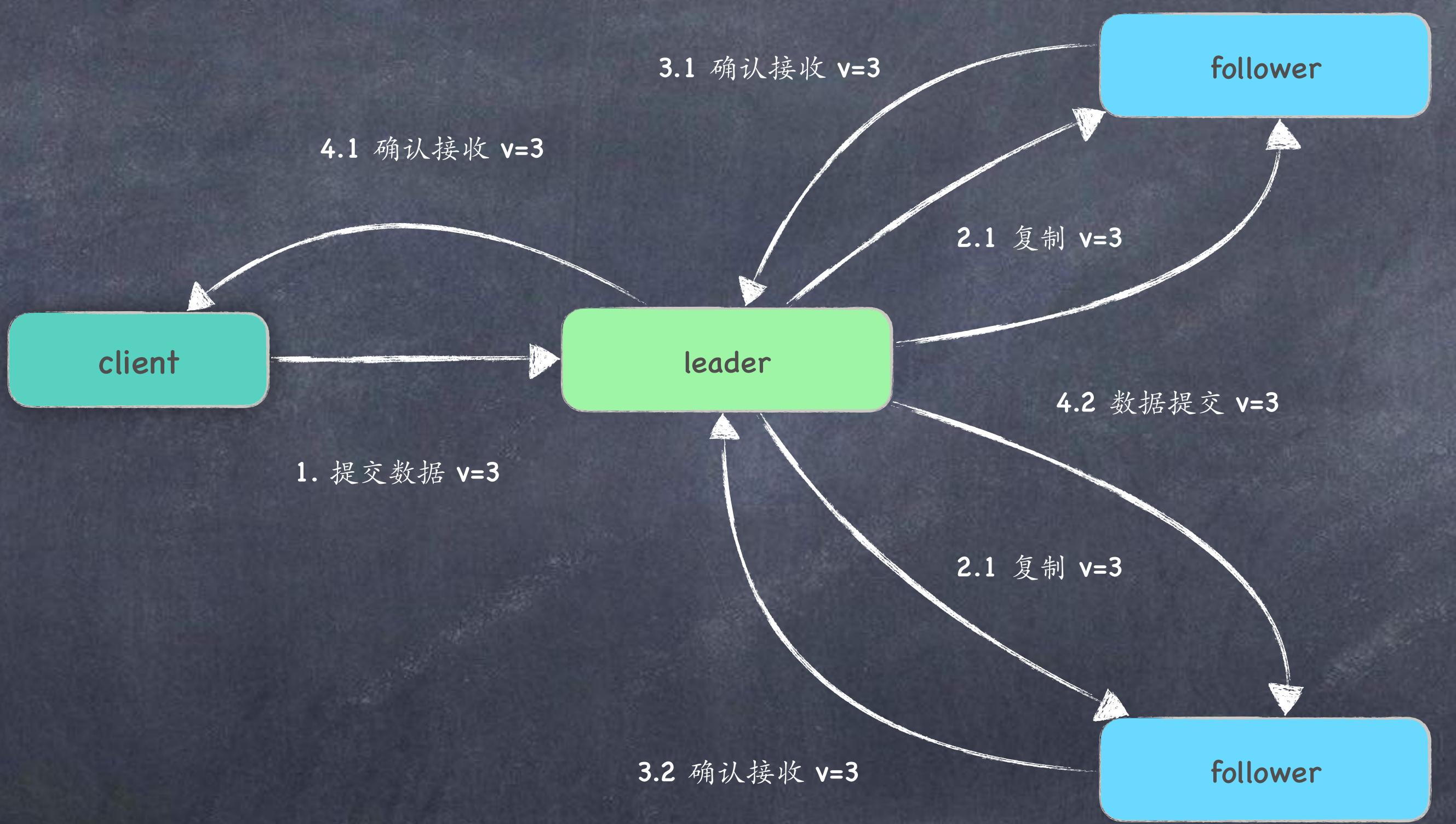
RequestVote RPC

```
message RequestVoteRequest {
    uint64 term = 1;          // 候选人的任期号
    uint64 candidate_id = 2;  // 请求投票的候选人 id
    uint64 last_log_index = 3; // 候选人最新日志条目的索引值
    uint64 last_log_term = 4; // 候选人最新日志条目对应的任期号
}

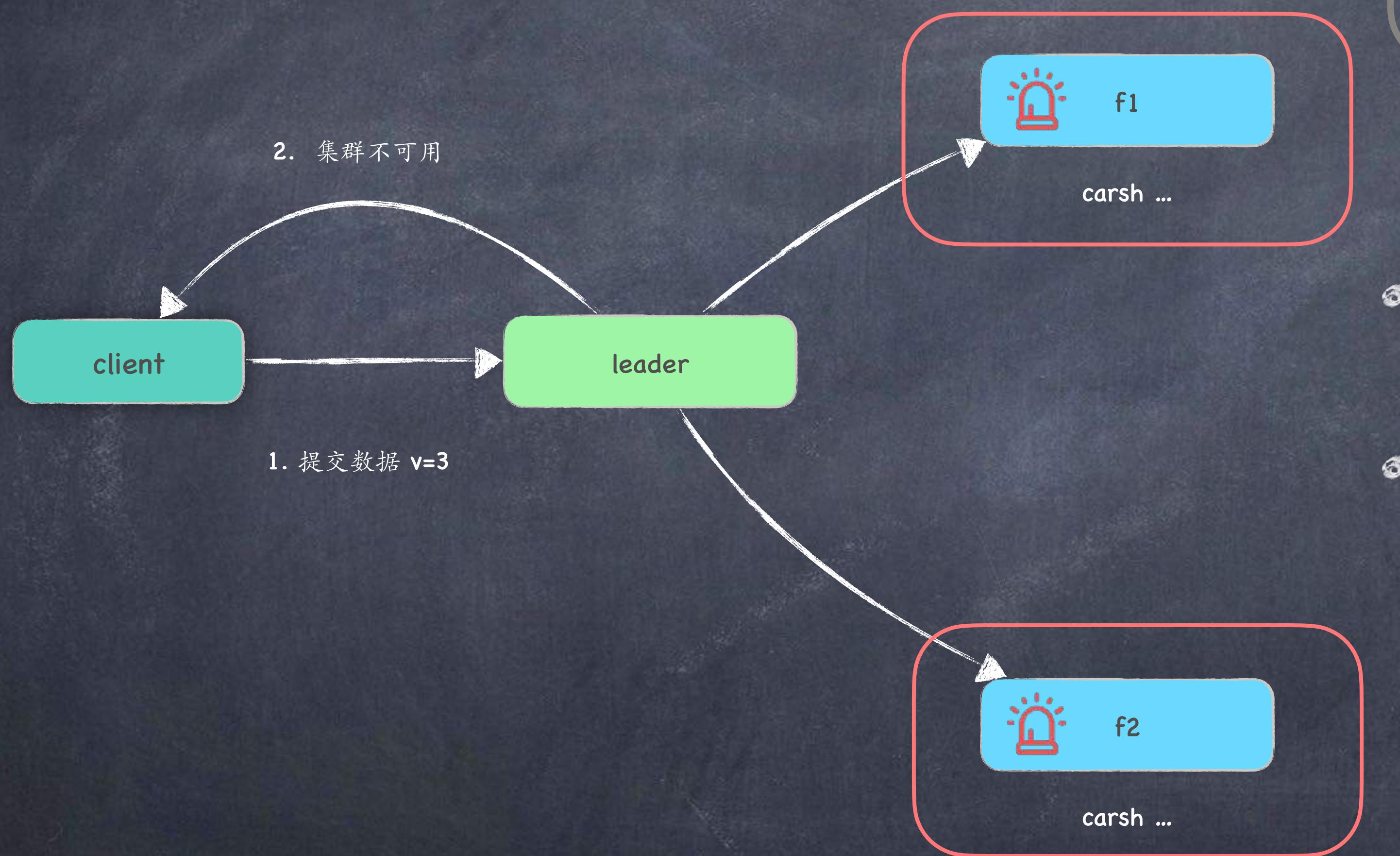
message RequestVoteResponse {
    uint64 term = 1;          // 对方的任期号
    bool vote_granted = 2;    // 如果候选人收到选票为 true
}
```

- ① 一个 **follower** 的竞选过程如下：
- ② 由 **follower** 变为 **candidate**，递增当前的 **term**；
- ③ 给自己投1票，同时向其他所有节点发送拉票请求（**RequestVote RPC**）；
- ④ **candidate** 节点等待投票结果，确认下一步自己的去向：
- ⑤ 获得多数投票，赢得选举，节点状态变为 **Leader**；
- ⑥ 其他的节点数据比当前节点更加新，则退化为 **Follower**；
- ⑦ 第一轮选举未产生结果，节点状态保持为**Candidate**；
- ⑧ 多 **candidate** 的场景下，**candidate** 如收到已成为 **leader** 的 **AppendEntries** 请求，经判断后会选择更变为 **follower**；

# log replication



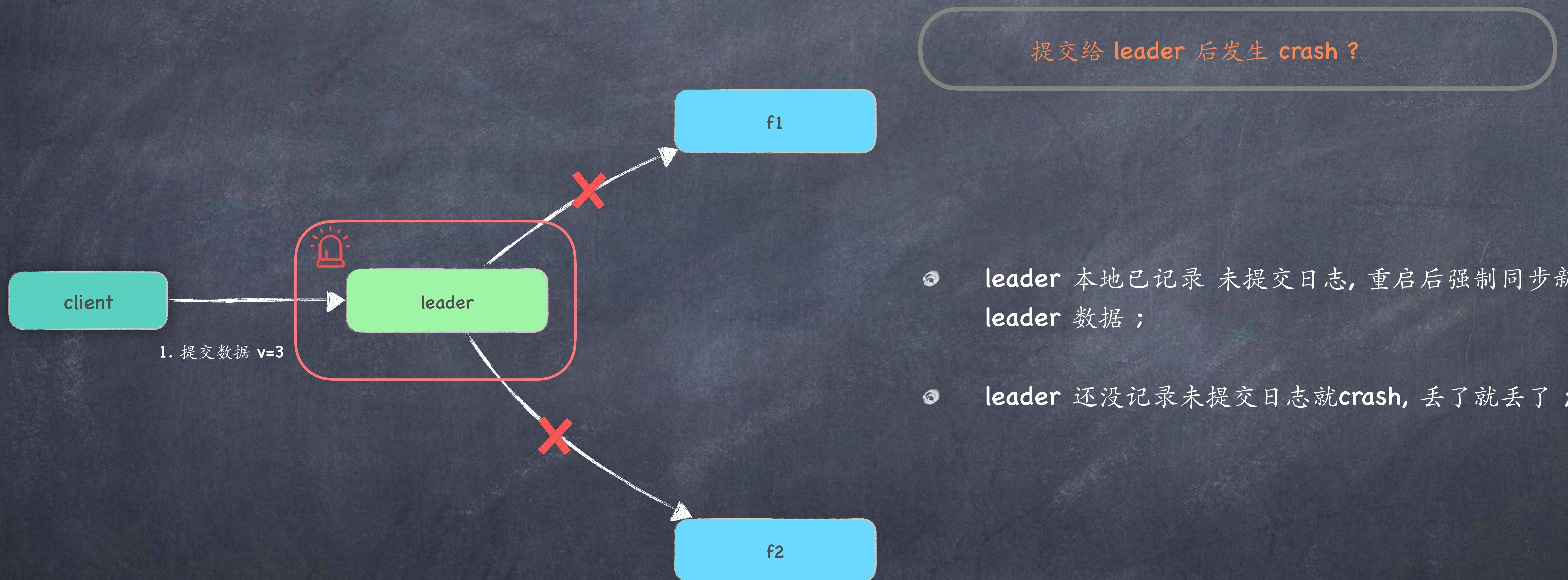
# exception case



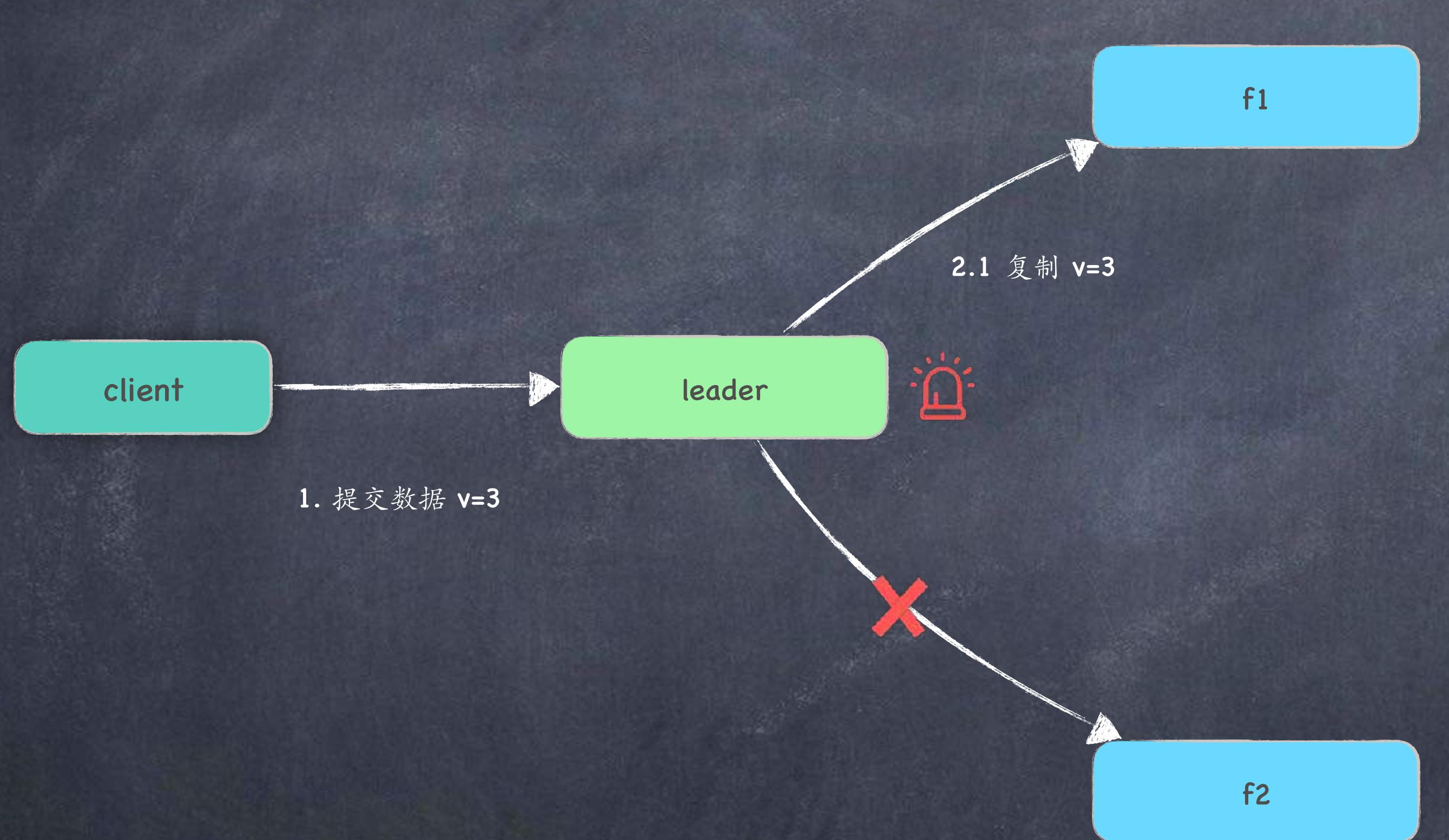
两个 follower 节点不可用, leader 如何处理请求 ?

- ① leader 通过心跳已知 follower 已挂，则直接返回错误；
- ② leader 不知节点已挂，同步数据时得知异常；
  - 触发异常
  - 触发超时

# exception case



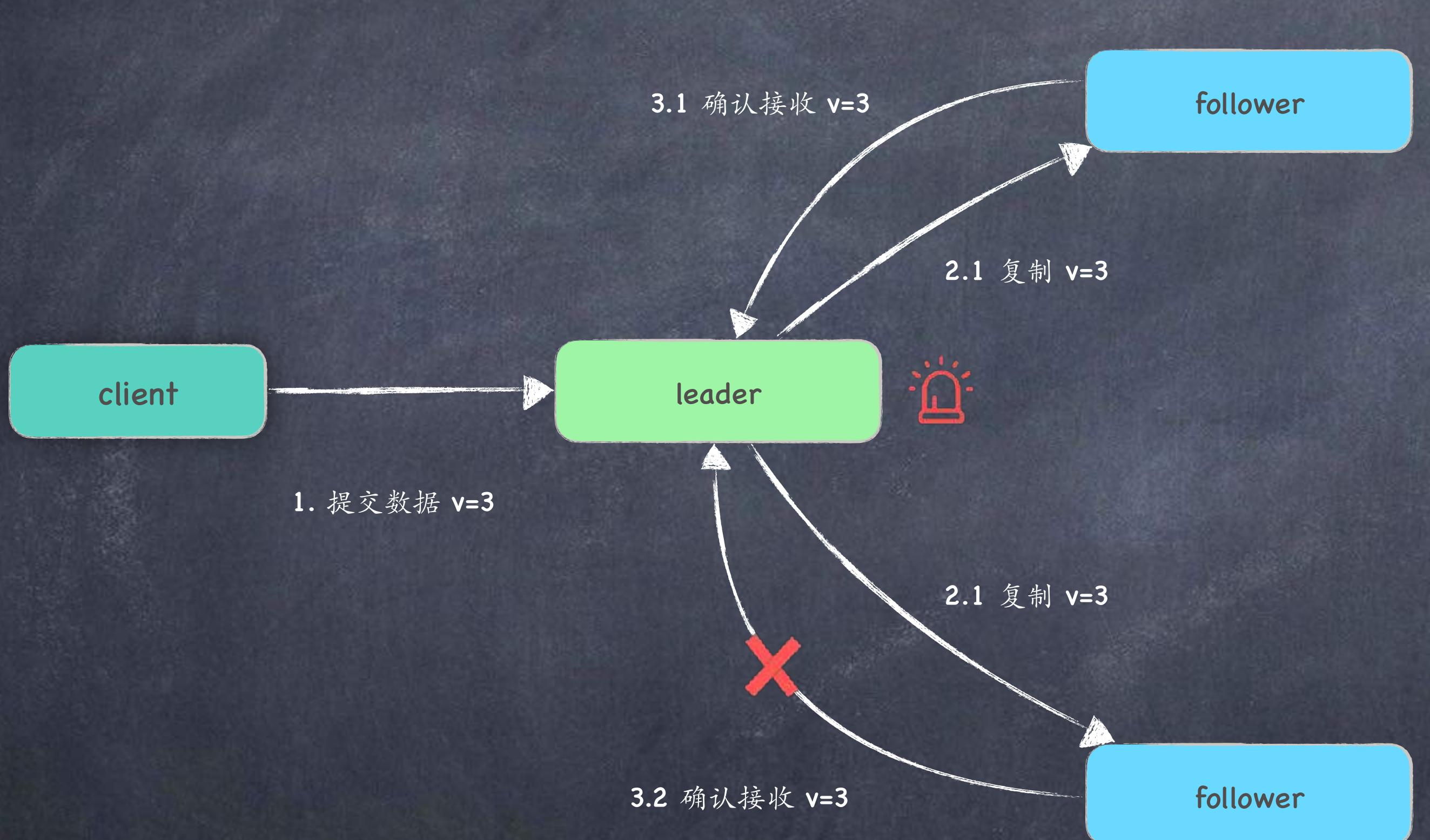
# exception case



复制给 follower-1 后, leader 就发生了 crash ?

- ④ leader 发生了重启，集群触发新的选举；
- ④ 由于 follower-1 的数据较新，那么该节点会晋升 leader；
- ④ follower-1 会把 uncommitted 的 v=3 同步给其他节点；
- ④ follower-1 收到其他节点接收确认后，进行提交日志；
- ④ client 需要实现提交的幂等性；

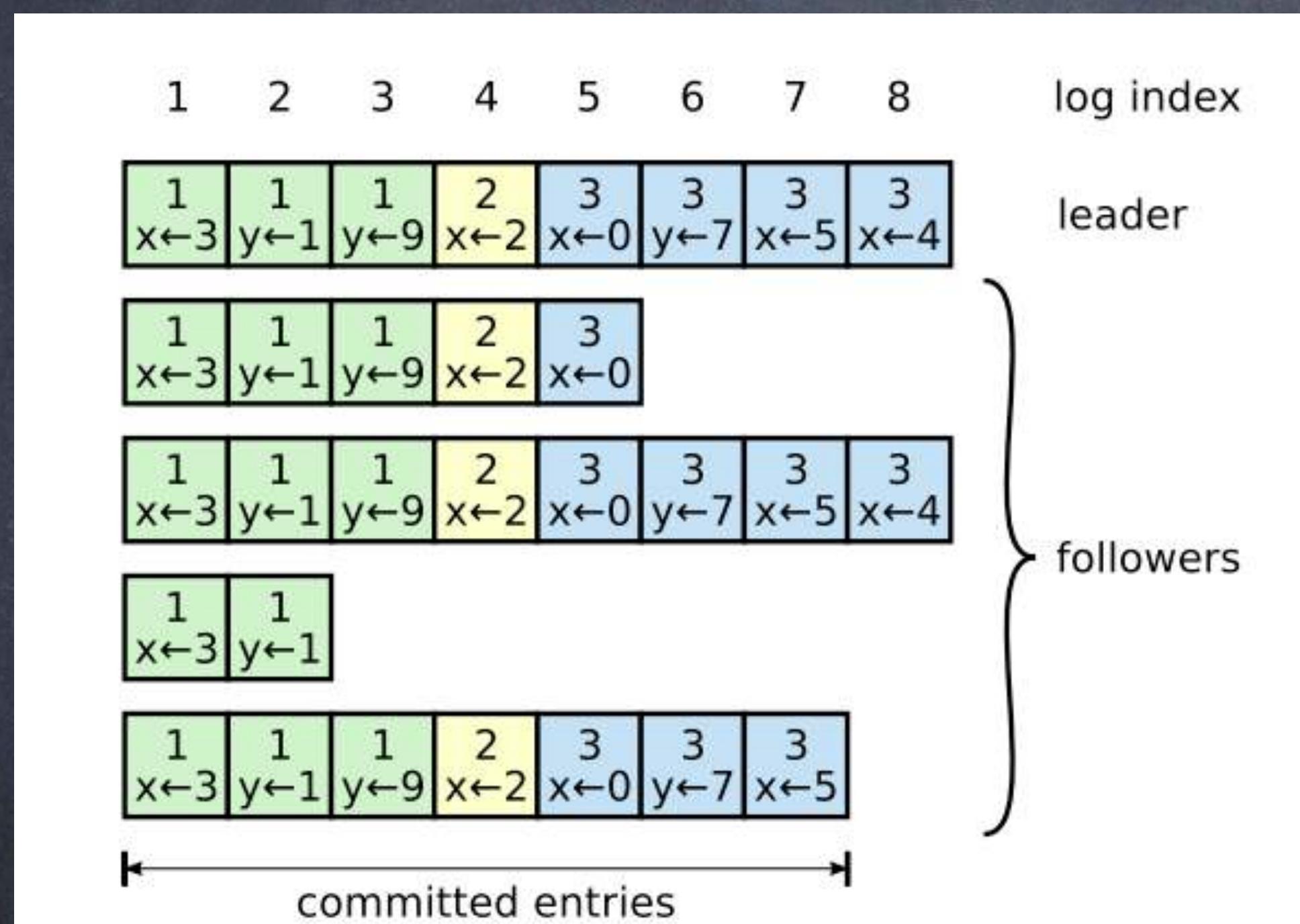
# exception case



follower 返回确认消息时, leader 发生了 crash ?

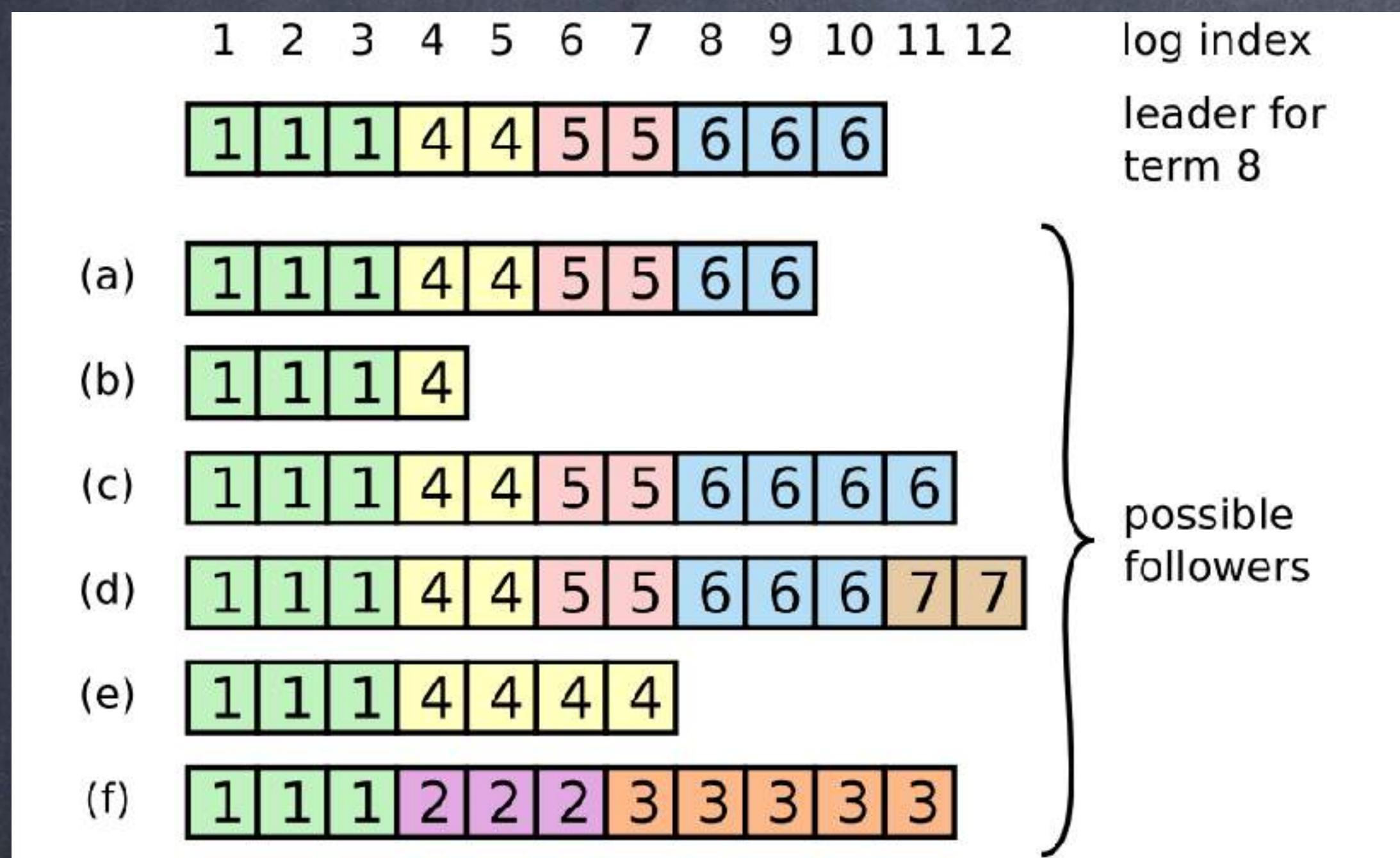
- 三个节点的数据已经一致, 都为 uncommitted v=3 ;
- 这时谁当 Leader 都可以 ;
- 新leader当选后, 需要进行同步提交通知 ;

# log index



- ② raft 中的每个日志记录都带有 **term** 和 **log index** 来唯一标识；
- ③ 日志记录具有两个特性
  - 如果两条不同节点的某两个日志记录具有相同的 **term** 和 **index** 号，则两条记录一定是完全相同的；
  - 如果两条不同节点的某两个日志记录具有相同的 **term** 和 **index** 号，则两条记录之前的所有记录也一定是完全相同的；

# 一致性复制



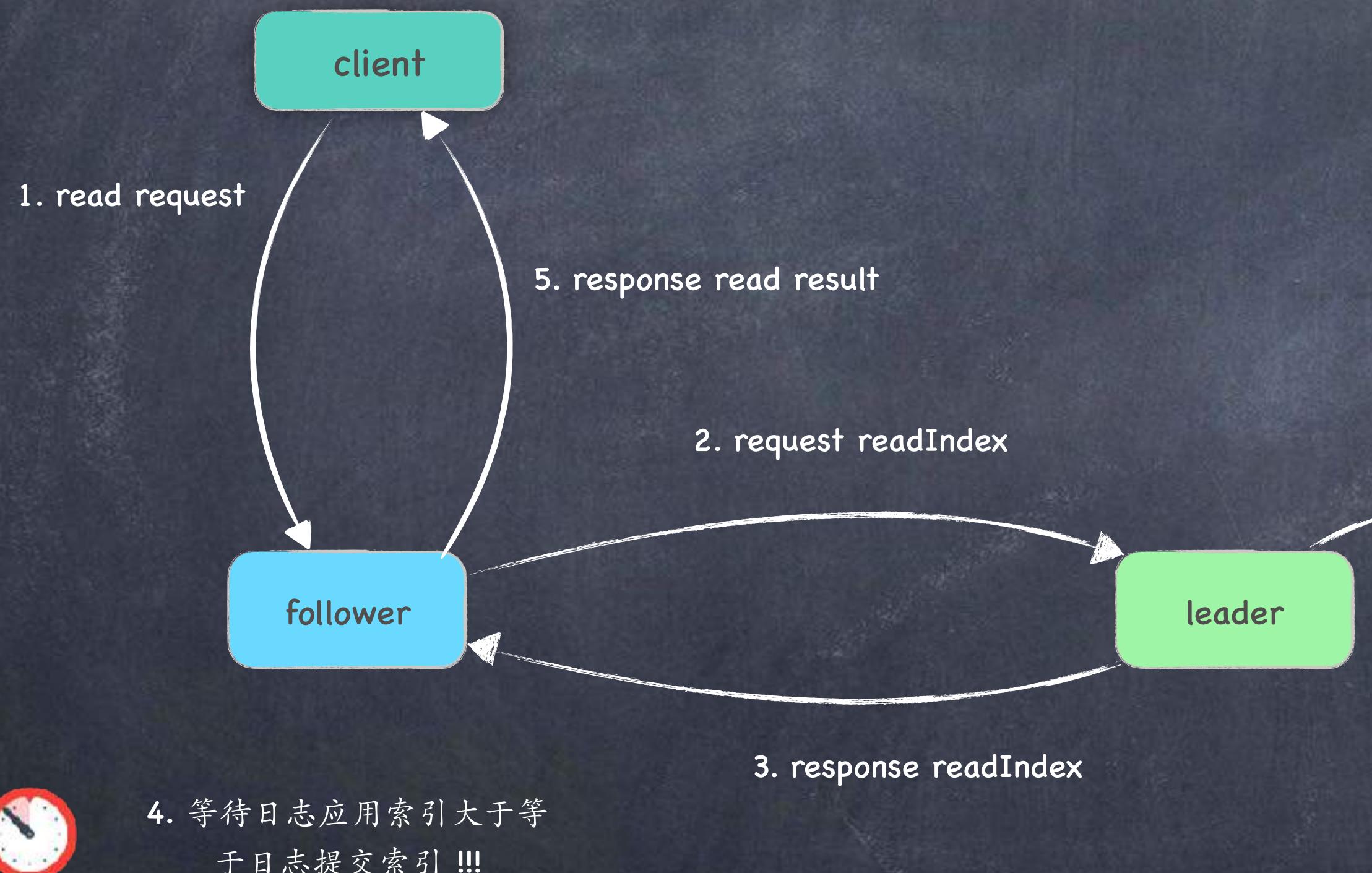
对于一个已选举出的 **leader**, **follower a-f** 都是有可能产生的情形；

- ① **leader** 从来不会覆盖、删除或者修改其日志；
- ② **leader** 会初始化一个数组 **nextIndex[]**, 该结构对应的值表示本 **leader** 将给对应 **follower** 发送的下一条日志 **index**；
- ③ 若 **follower** 对比其前一条 **log** 不一致，则会拒绝 **leader** 发来的请求。此时 **leader** 就将其在 **nextIndex[]** 中的对应值减一；
- ④ **leader** 不断重试直到 **follower** 比对成功，然后 **follower** 接受 **AppendEntries RPC**, 一个个的抛弃所有冲突的日志；
- ⑤ **leader** 按照自身日志顺序将日志正常复制给 **follower**, 并不断将 **nextIndex[]** 对应值 +1，直到对应值“追上”自身日志的 **index** 为止；

**a** 和 **b** 的情况是没有完全收到来自 **leader** 的 **AppendEntries RPC**, 而 **c-f** 则是带有不同时期的未提交的日志（有可能是他们当 **leader** 时产生的，但没有提交就 **crash** 了）。

# 线性一致性读

线性读



串行读

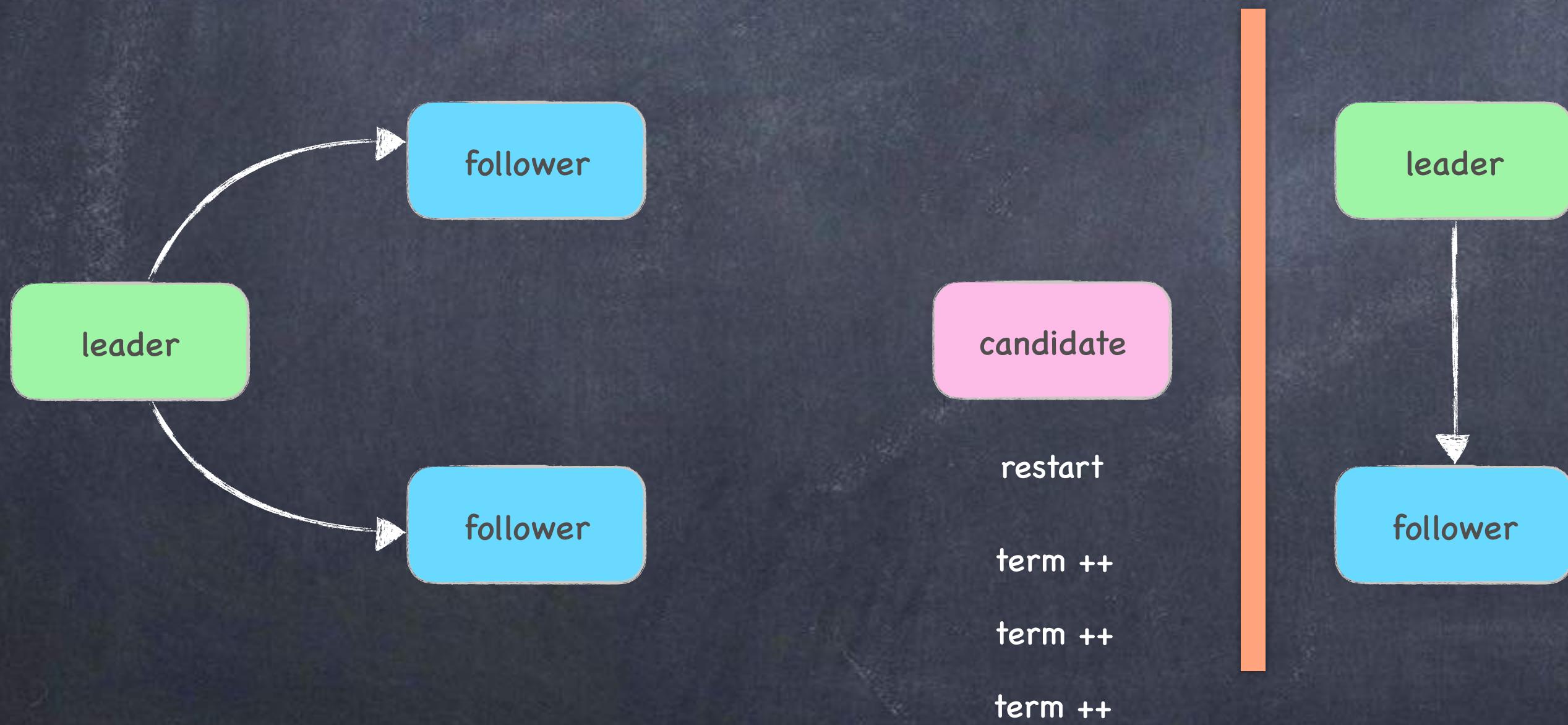


串行读

- 具有低延时、高吞吐量的特点；
- 适合对数据一致性要求不高的场景；
- 线性读
- 吞吐相对差；
- 解决读数据一致性要求高的场景；

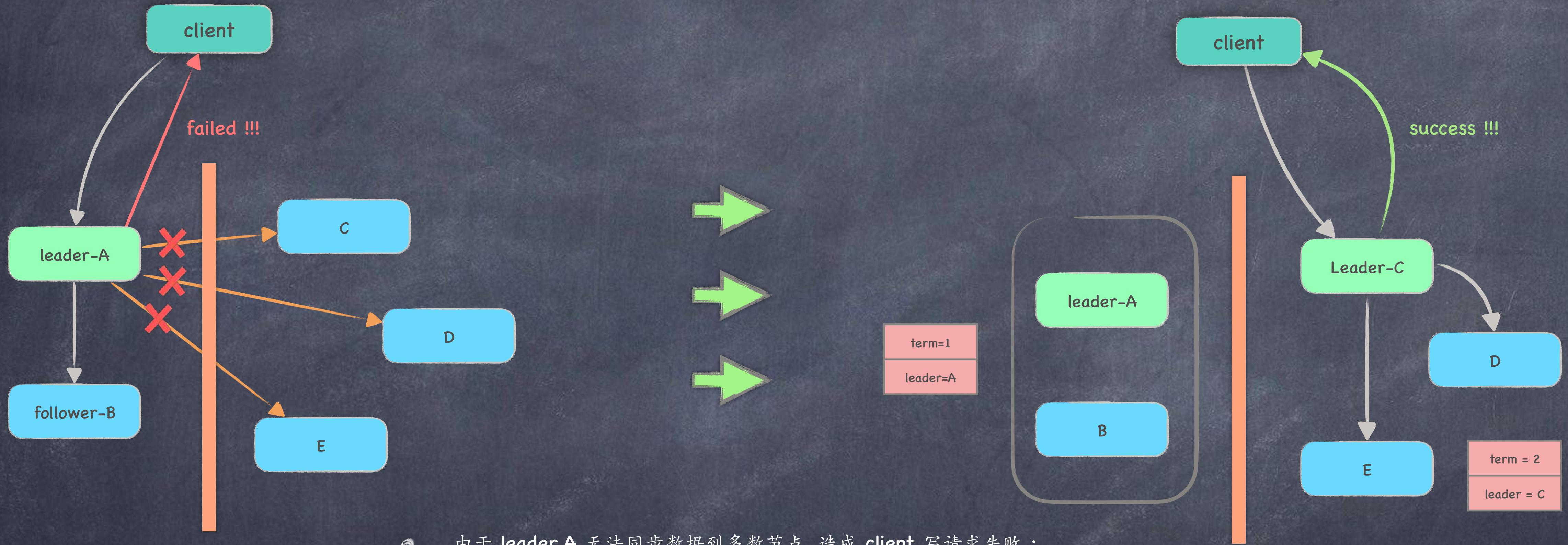
# prevote

避免由于网络分区导致 **candidate** 的 **term** 不断增大 !!!



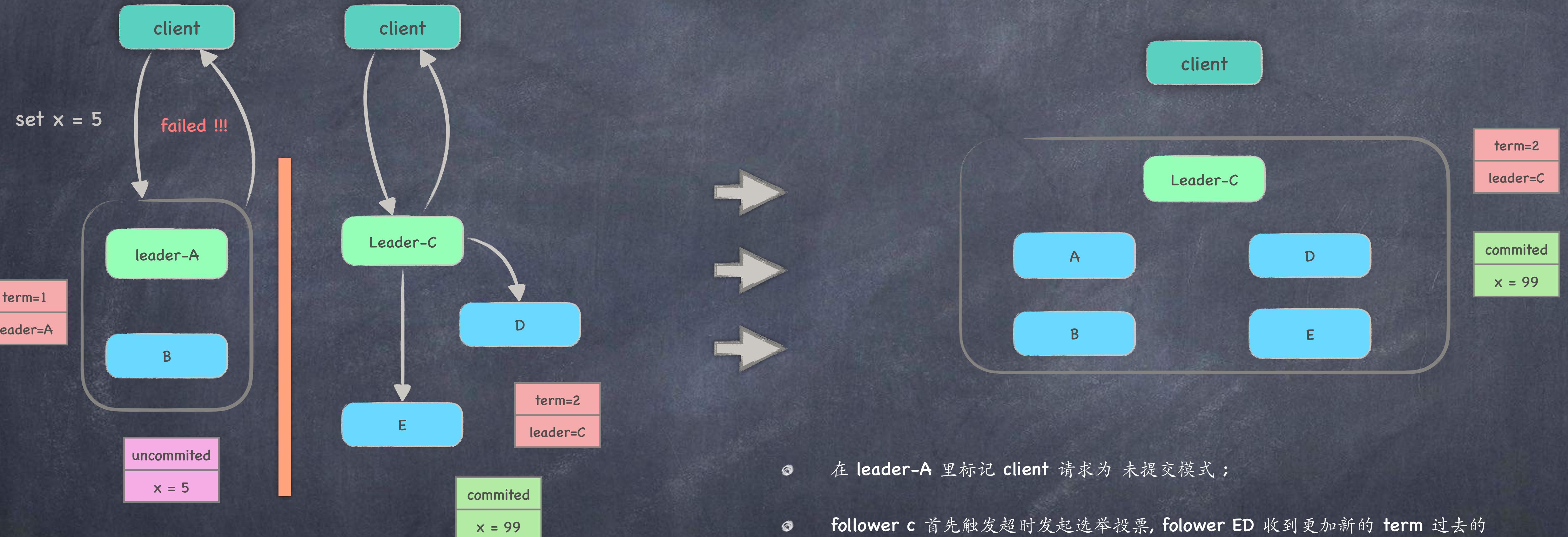
- ④ **Term** 的递增很重要，连 **Leader** 也会因为遇到更新的 **Term** 而退化为 **follower**；
- ⑤ 变为 **candidate** 的条件
- ⑥ 询问其他节点是否有可用 **leader**；
- ⑦ 可连通绝大部分节点；

# 网络分区



- 由于 **leader A** 无法同步数据到多数节点，造成 **client** 写请求失败；
- 网络分区后 **follower c** 长时间未收到心跳，则触发选举且当选；
- client** 按照策略一直尝试跟可用的节点进行请求；
- 当网络恢复后，**leader A B** 将降为 **follower**，并且强制同步 **Leader C** 的数据；

# 脑裂的一致性



leader-A 还未感知异常时, client 发送写请求?

- 在 leader-A 里标记 client 请求为 未提交模式；
- follower c 首先触发超时发起选举投票, folower ED 收到更加新的 term 过去的 requestVote, 则回应同意当选；
- leader C 可以接收用户的请求, 由于可以拿到大多数的回应, 则可以正常提交数据；
- 当网络分区问题解决后, 由于旧集群的Leader A 的term 低于 Leader C , leader A 强制同步 Leader C的数据;

# safety

**Election Safety:** at most one leader can be elected in a given term. §5.2

**Leader Append-Only:** a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

**Log Matching:** if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

**Leader Completeness:** if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

**State Machine Safety:** if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.  
§5.4.3

content in the 5 page

- 选举安全性 (Election Safety)

- 在一个任期内只能存在最多一个leader节点；
- 拒绝 term 和 log 没自己新的 candidate；

- Leader日志为只附加原则 (Leader Append-Only)

- leader节点永远不会删除或者覆盖本节点上面的日志数据；

- 日志匹配性 (Log Matching)

- 如果两个节点上的日志，在日志的某个索引上的日志数据其对应的任期号相同，那么在两个节点在这条日志之前的日志数据完全匹配；

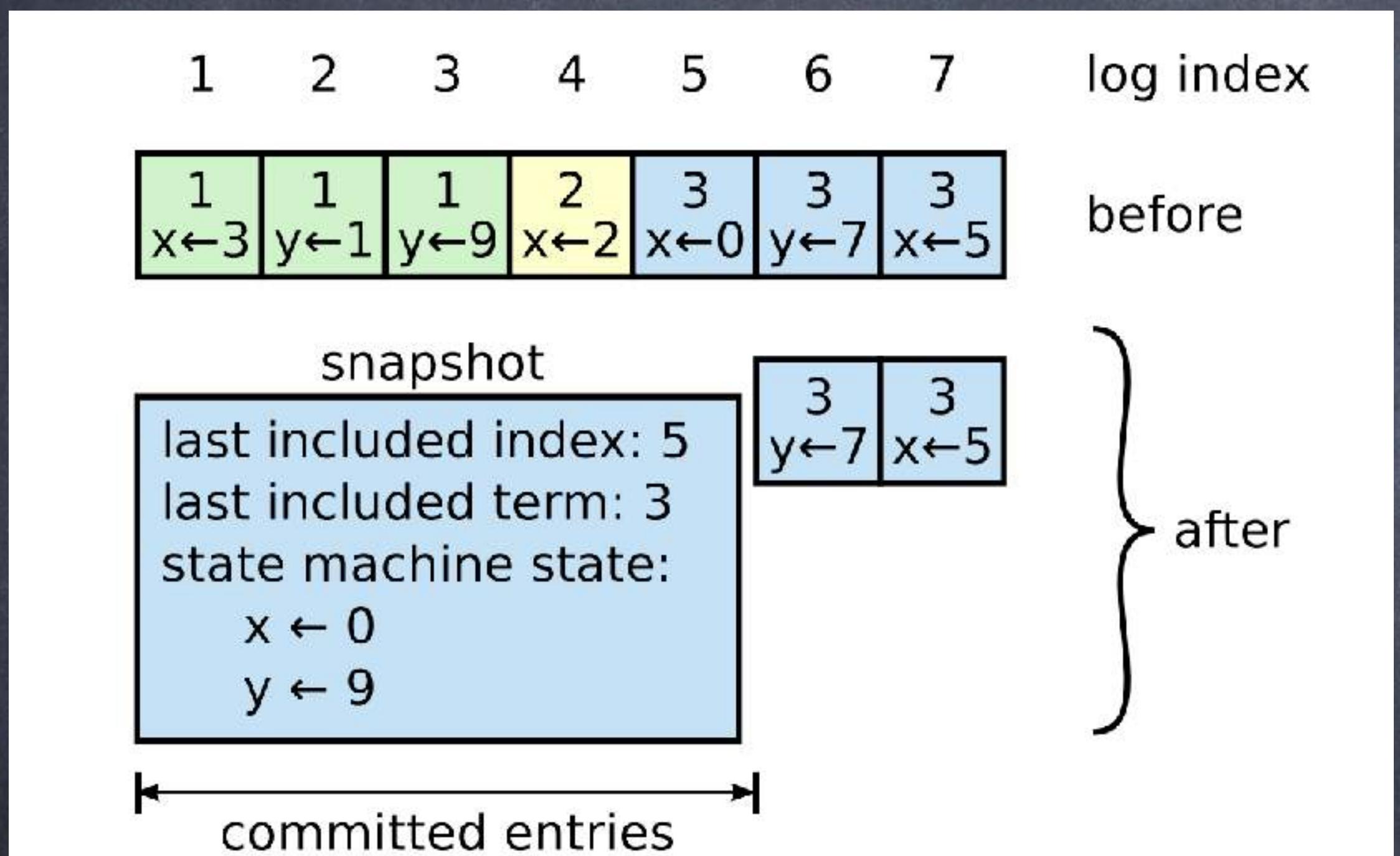
- leader完备性 ( Leader Completeness )

- 如果一条日志在某个任期内已被提交，那么这条日志数据在leader节点上更高任期号的日志数据中都存在；

- 状态机安全性 ( State Machine Safety )

- 如果某个节点已经将一条提交过的数据输入raft状态机执行了，那么其它节点不可能再将相同索引的另一条日志数据输入到raft状态机中执行；

# snapshot & wal



默认10w条日志做一个快照 !

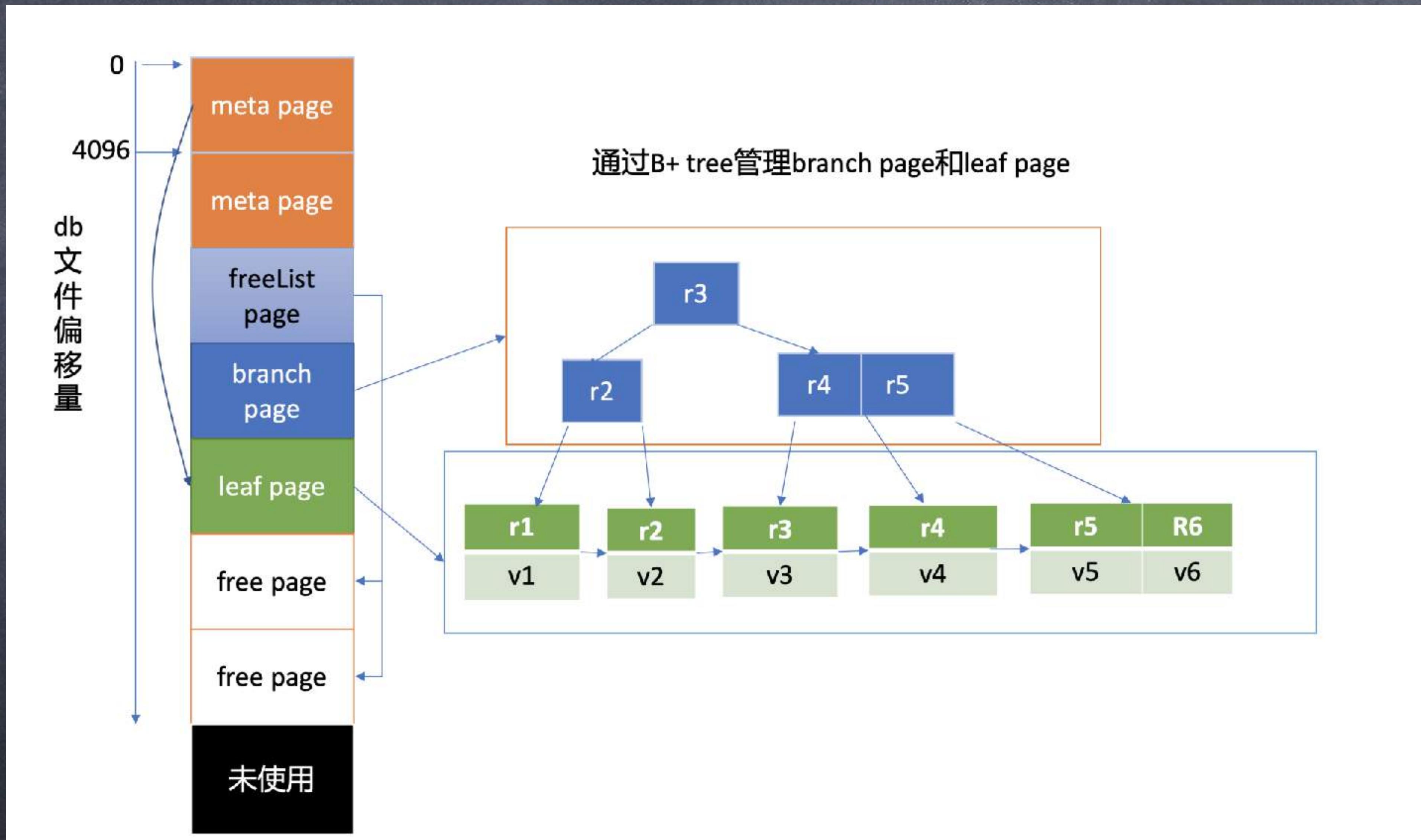
- wal (类似 mysql redo, redis aof)
  - 该 wal 只用于 raft entry 日志；
  - 预写式日志，持久化存储；
  - 每次更改数据前，需要追加写日志，追加写是顺序写io；
  - 如发生 crash 异常，则需重放 wal 日志；
  - 记录整个数据变化的全部历程；
- snapshot
  - 解决 wal 日志占用空间过大；
  - 解决新的节点加入进来后，需要同步和恢复的日志太多；
  - 解决重启回放 wal 的时间过长；
- snapshot 原理？
  - 压缩日志；
  - 保留压缩合并后的 snapshot 及 增量的wal；



boltdb



# boltdb



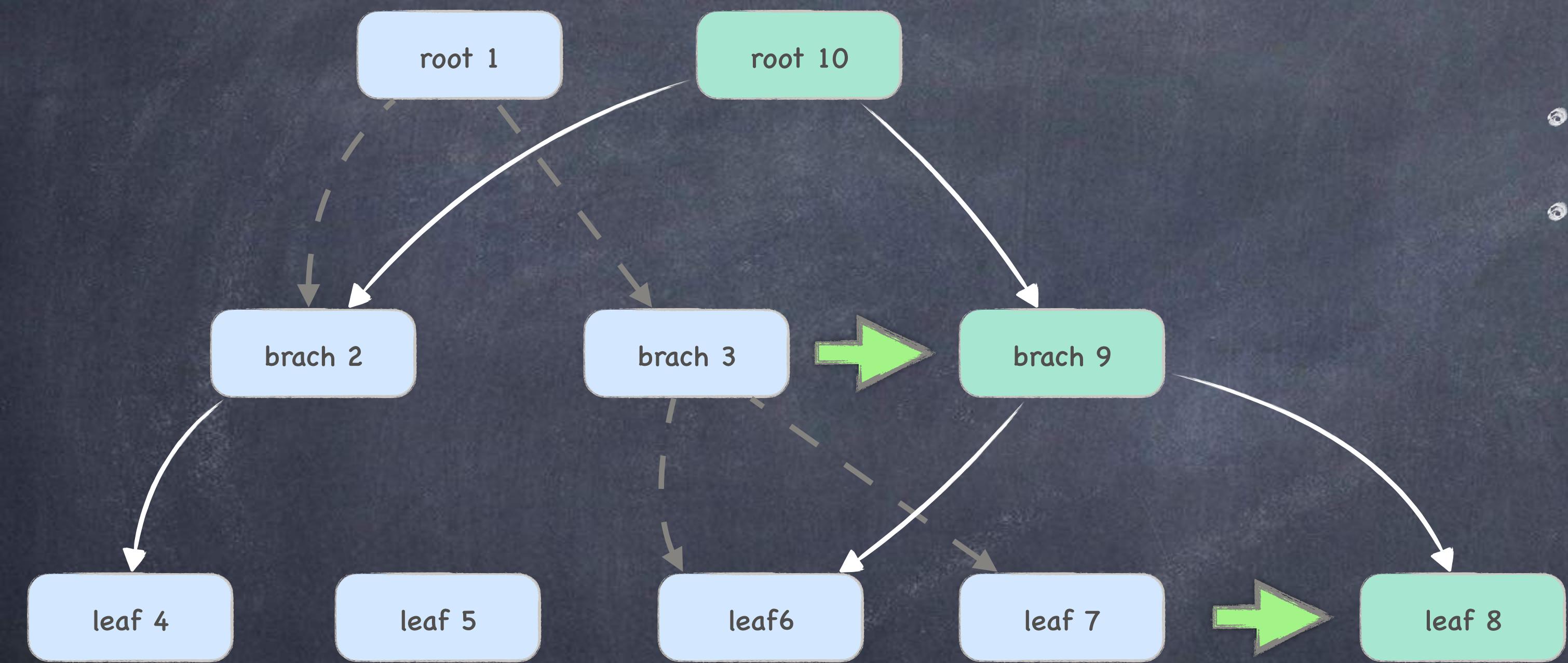
# boltdb basic



重点!

- etcd、consul都有在使用boltdb；
- boltDB为 b+tree 结构
- boltDB数据只是单个 db 文件，该文件通过 mmap 映射到进程的内存空间里；
- boltDB没有 wal 的预写机制，通过cow实现无锁读写并发；
- boltDB不会进行 compact 和 defrag 操作，只是将释放的 page 存入 freelist，下次申请 page 时可从freelist 里获取. freelist ids是pageid有序，所以可应对有申请N个page的需求；
- boltDB的基础单位是page，每个 page 为定长的4KB，一条数据可以跨越多个 page；

# boltcdb write



- 写事务不会原地修改当前的 **page**, 而是对数据进行拷贝, 在新的 **page** 上进行修改;
- 对 **leaf page** 进行修改, 其 **leaf** 所涉及的所有 **page** 都需要拷贝修改;
- 如对 叶子节点 **leaf 7** 进行修改?
  - 拷贝 **meta** 数据, 相当于快照;
  - 需要把 **leaf 7** 的数据拷贝到新的 **leaf 8** 进行修改;
  - leaf 7** 的其父节点们也要随之复制和修改;
  - 更新 **meta page** 数据;

类似 mysql double write 实现

修改过程为 拷贝副本并修改 (RCU)

# atomicity



```
// write writes the meta onto a page.
func (m *meta) write(p *page) {
    ...

    // Page id is either going to be 0 or 1
    // which we can determine by the transaction ID.
    p.id = pgid(m.txid % 2)
    p.flags |= metaPageFlag

    // Calculate the checksum.
    m.checksum = m.sum64()

    m.copy(p.meta())
}

// meta retrieves the current meta page reference.
func (db *DB) meta() *meta {
    metaA := db.meta0
    metaB := db.meta1
    if db.meta1.txid > db.meta0.txid {
        metaA = db.meta1
        metaB = db.meta0
    }

    if err := metaA.validate(); err == nil {
        return metaA
    } else if err := metaB.validate(); err == nil {
        return metaB
    }
}
```

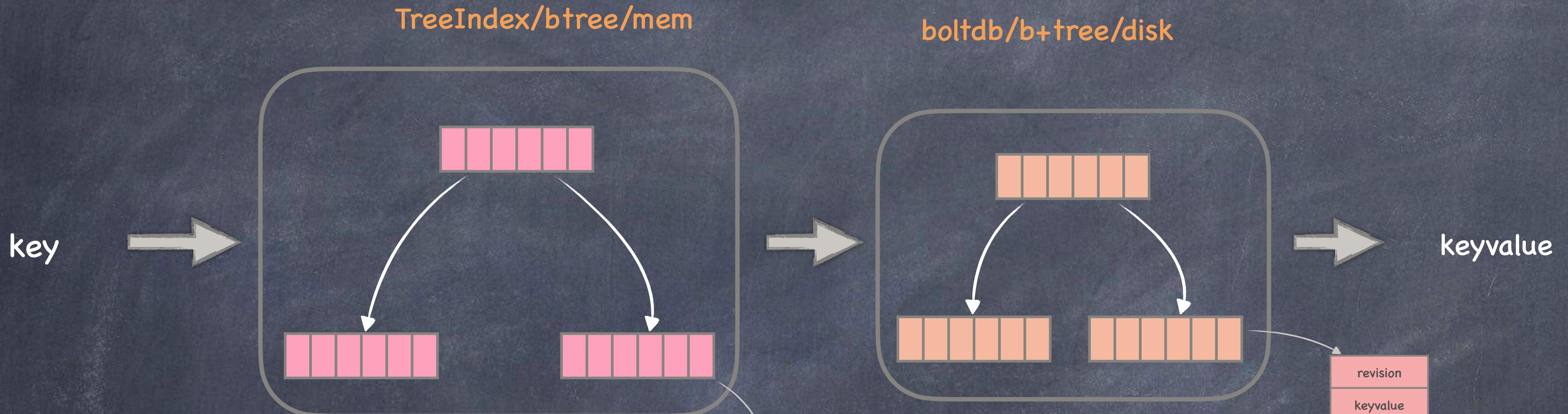
- ④ 若事务未提交时出错，因为 **boltdb** 的操作都是在内存中进行，不会对数据库造成影响；
- ④ 若是在 **commit** 的过程中出错，如写入文件失败或机器崩溃，**boltdb** 写入文件的顺序也保证了不会造成影响：
  - ④ 先写入 **B+** 树数据 和 **freelist** 数据；
  - ④ 后写入 **metadata** 元数据；
- ④ 写磁盘是无法保证其原子性，那么如何安全的写 **meta page**？
  - ④ 两个 **meta page**，一个是新，一个是旧，如启动发生异常，则使用旧 **meta** 元数据；
  - ④ **metadata** 交替保存在文件前2个 **page** 中；
  - ④ **meta.checksum** 检测**meta**是否可用；



## etcd design



# 索引及存储



- ④ treeindex 采用 google btree 库构建；
- ④ treeindex 是内存中的索引数据结构；
- ④ treeIndex 只存 key 和 revision, 而 value 都在 boltdb 中；
- ④ 关系 keyIndex -> generation -> revision；
- ④ boltdb 中的 key 都是 revision；

- ④ 如何获取请求数据？
  - ④ 首先在 TreeIndex 索引结构里通过 key 获取 keyIndex；
  - ④ 通过 keyIndex 找到对应的 generations；
  - ④ 在 generations 里找到对应的 revision；
  - ④ 拿着 revision 到 BoltDB 中获取 keyvalue 数据；
  - ④ 该 keyvalue 其实就是 mvccpb.KeyValue 结构；

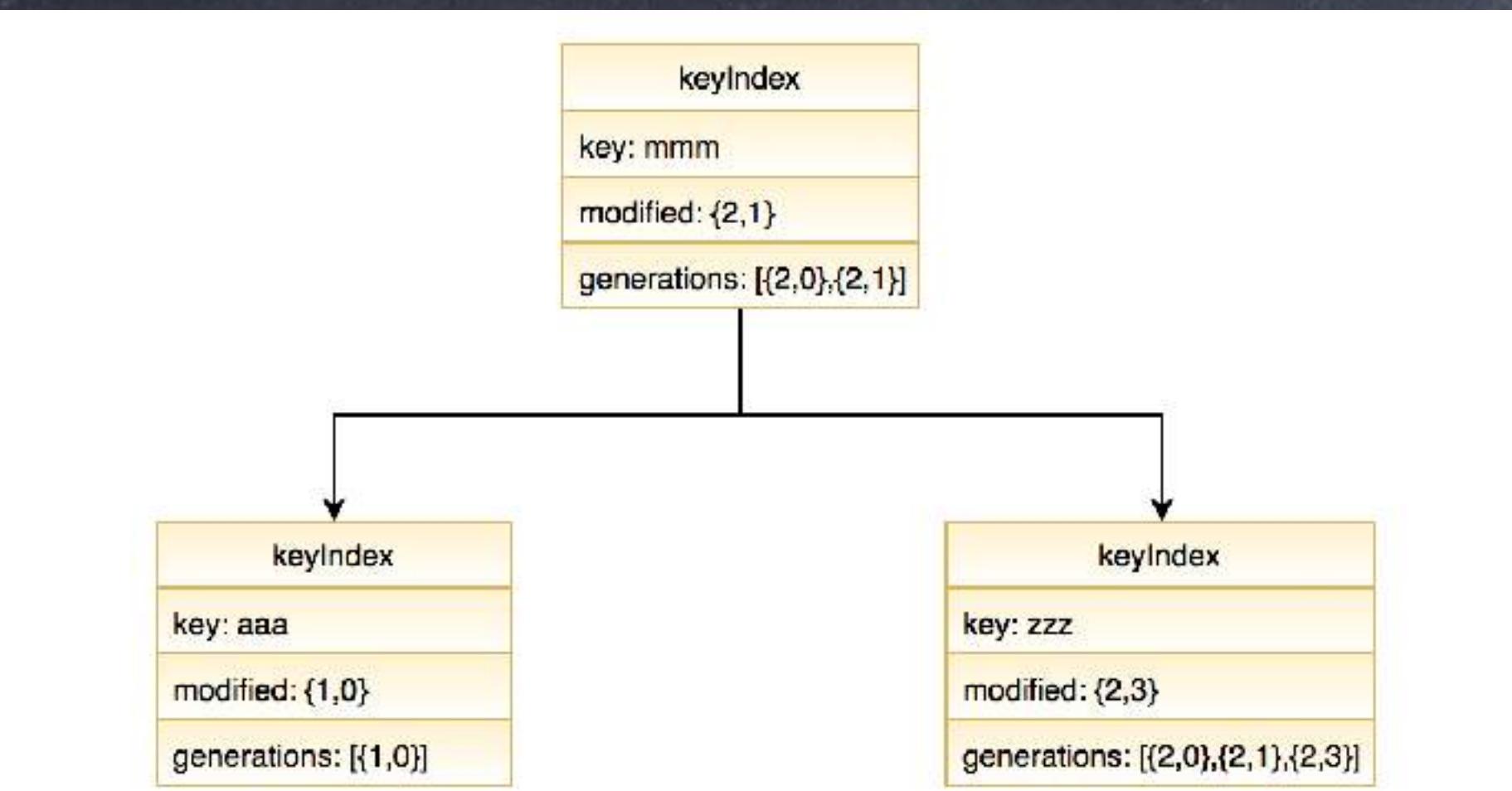
## treeIndex

## struct

```
type keyIndex struct {
    key      []byte
    modified  revision // the main rev of the last modification
    generations []generation
}

// generation contains multiple revisions of a key.
type generation struct {
    ver      int64
    created revision // when the generation is created (put in first revision).
    revs     []revision
}

type revision struct {
    // main is the main revision of a set of changes that happen atomically.
    main int64
    // sub is the sub revision of a change in a set of changes that happen atomically.
    // Each change has different increasing sub revision in that set.
    sub int64
}
```



```
// mvccpb.KeyValue

message KeyValue {
    bytes key = 1;
    int64 create_revision = 2;
    int64 mod_revision = 3;
    int64 version = 4;
    bytes value = 5;           // 真正的数据
    int64 lease = 6;
}
```

# MVCC

```
● ● ●

# 更新key hello为world1
$ etcdctl put hello world1
OK

# 通过指定输出模式为json，查看key hello更新后的详细信息
$ etcdctl get hello -w=json
{
  "kvs": [
    {
      "key": "aGVsbG8=",
      "create_revision": 2,
      "mod_revision": 2,
      "version": 1,
      "value": "d29ybGQx"
    }
  ],
  "count": 1
}

# 再次修改key hello为world2
$ etcdctl put hello world2
OK

# 确认修改成功，最新值为world2
$ etcdctl get hello
hello
world2

# 指定查询版本号，获得了hello上一次修改的值
$ etcdctl get hello --rev=2
hello
world1
# 删除key hello
$ etcdctl del hello
1
```

- ④ 控制锁的粒度，并发读写相互不阻塞，但写写阻塞；
- ④ 由于 **etcd** 都是那种短事务，不好体现；
- ④ **boltdb** 里自带一个大写锁，不好体现；
- ④ 每次写或删除操作都会创建一个新的版本；
- ④ **client** 可以读取历史变更数据；
- ④ 方便 **client watch** 的增量更新通知；

# compact



- ① 最少保留最近的 **revision**；
- ② 默认没有开启自动压缩功能；
- ③ 可按照 **revision count** 来自动 **compact**；
- ④ 可按照 **periodic** 时间来自动 **compact**；
- ⑤ 手动压缩原理
  - 压缩任务放到 **fifo** 队列中异步执行；
  - 遍历并压缩 **treeindex** 的 **keyindex** 索引；
  - 删除 **keyindex** 里 **boltdb** 的 **key**；

```
● ● ●  
# 自动保持一个小时的历史 (etcd 启动参数)  
etcd --auto-compaction-retention=1 --auto-compaction-mode 'periodic'  
  
# 自动保持近1000条的历史 (etcd 启动参数)  
etcd --auto-compaction-retention=1000 --auto-compaction-mode 'revision'  
  
# 手动压缩到修订版本 N  
etcdctl compact N
```

# tombstone

keyIndex
key: hello
modified: {4, 1}
generations[0] -> {2, 0} {3 0} {tombstone}

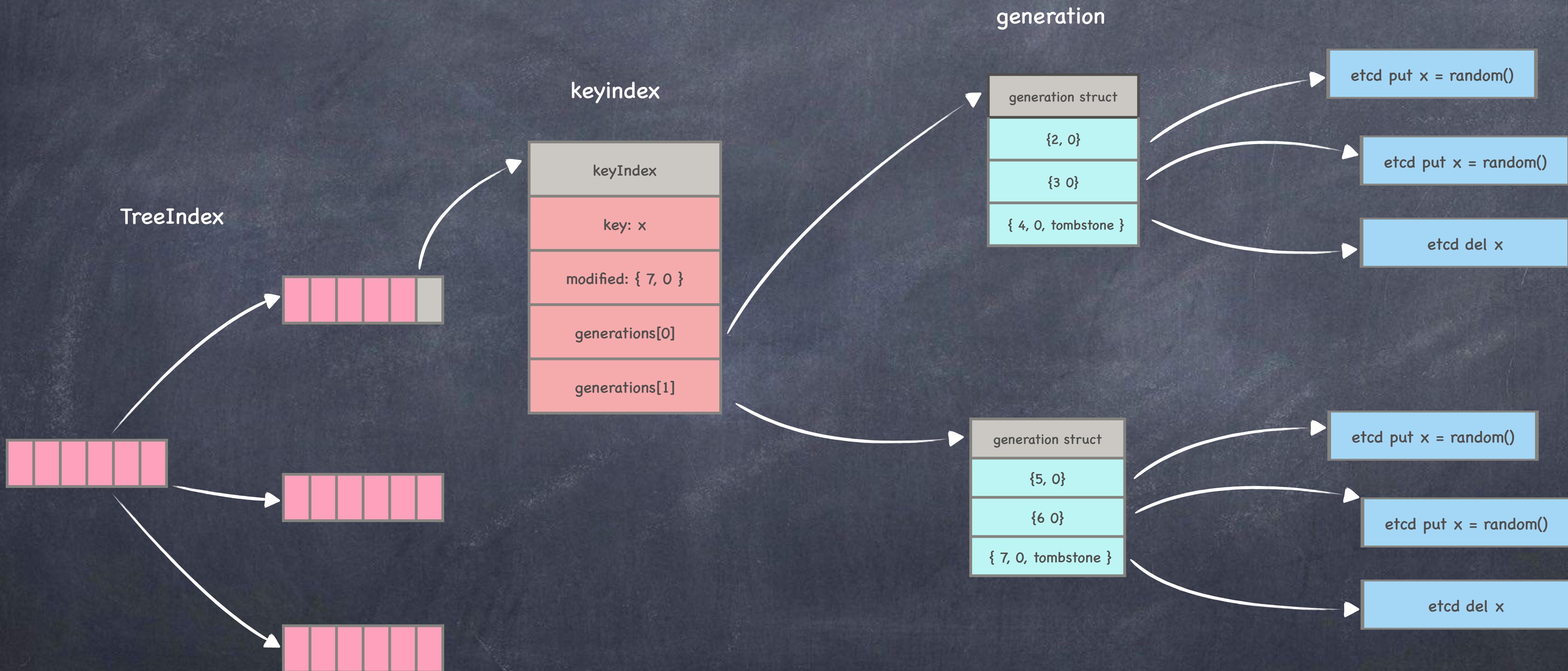
compact( 1000 )

空 generation

command	boltdb key	boltdb value/myccpb.KeyValue
etcdctl put hello world1	{2,0}	{"key":"aGVsbG8=","create_revision":2,"mod_revision":2,"version":1,"value":"d29ybGQx"}
etcdctl put hello world2	{3,0}	{"key":"aGVsbG8=","create_revision":2,"mod_revision":3,"version":2,"value":"d29ybGQy"}
etcdctl del hello	{4,0,t}	{"key":"aGVsbG8="}

- 为何 etcd 采用 **lazy delete** 延迟删除机制 ?
- 最大程度保证 **watcher** 能够获取完整的状态变更 ;
- 实时操作 **boltdb** 开销要比批量操作慢 ;
- 删除操作也会增加一个新版本, 该版本标识为 **tombstone** ;
- 当 **compact** 遇到该字段含有 **tombstone** 标识时才会进行删除清理 **keyindex** ;
- 数据 (**keyindex**) 只有在被标记 **tombstone** 后才能被**compact**给彻底删除 ;

# mvcc && treeindex



# defrag

```
func (b *backend) defrag() error {
    b.mu.Lock()
    defer b.mu.Unlock()

    dir := filepath.Dir(b.db.Path())
    temp, err := ioutil.TempFile(dir, "db.tmp.*")
    if err != nil {
        return err
    }
    tdbp := temp.Name()
    tmpdb, err := bolt.Open(tdbp, 0600, &options)
    if err != nil {
        return err
    }

    dbp := b.db.Path()

    // 把当前db内容读出来 复制到 目标db中.
    err = defragdb(b.db, tmpdb, defragLimit)
    if err != nil {
        tmpdb.Close()
        if rmErr := os.RemoveAll(tmpdb.Path()); rmErr != nil {
        }
        return err
    }

    err = b.db.Close()
    if err != nil {
        b.lg.Fatal("failed to close database", zap.Error(err))
    }
    err = tmpdb.Close()
    if err != nil {
        b.lg.Fatal("failed to close tmp database", zap.Error(err))
    }
    // gofail: var defragBeforeRename struct{}
    err = os.Rename(tdbp, dbp)
    if err != nil {
        b.lg.Fatal("failed to rename tmp database", zap.Error(err))
    }

    return nil
}
```

- **etcdctl defrag** 会阻塞**etcd**的访问请求，其主要用来自收**boltdb**的空间；
- **boltdb**本身没有回收接口，删除的数据其实被标记在 **freelist** 里等待下次分配，也就是说空间不会回收；
- 如想回收空间，则只能是重做一个新的**db**文件.

一句话，把数据顺序写到一个新文件里 !!!

# etcd 内存占用



- **raftlog** 会保留一部分的日志在内存中；
- 如果更新频繁或者 **kv** 太多的话, **treeIndex** 也会随之变大；
- **watcher** 的缓冲区在网络拥塞或事件太多时, 也会占用不少内存；
- **boltdb mmap** 在持续新增数据时引发扩容, 后面删除也不会回收空间, 可使用 **defrag** 来达到缩容效果；

# watch

```
if getResp, err = kv.Get(context.TODO(), "/school/class/students"); err != nil {
    fmt.Println(err)
    return
}

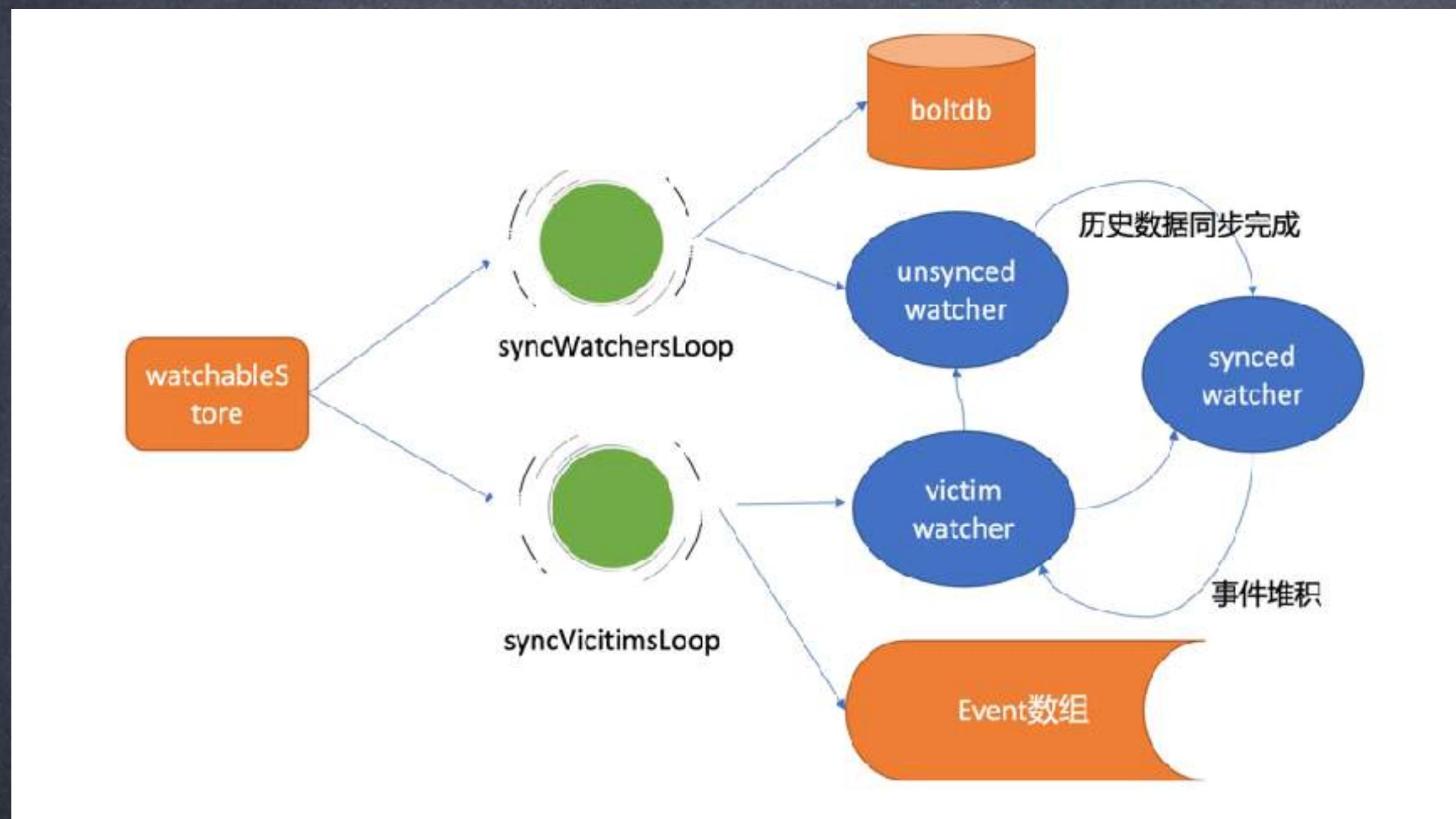
watchStartRevision = getResp.Header.Revision + 1
watcher = clientv3.NewWatcher(client)

watchRespChan = watcher.Watch(ctx, "/school/class/students", clientv3.WithRev(watchStartRevision))
for watchResp = range watchRespChan {
    for _, event = range watchResp.Events {
    }
}
```

1. 首先获取最新的数据及对应 **revision** 版本号；
2. 使用返回最新的 **revision** 进行 **watch** 监听；
3. 如遇到**revision**已被 **etcd** 压缩合并，则重新进行第一步；

```
loginfo x »
2021 @ 08:04:01.632 watch with rev canceled {"error":"etcdserver: mvcc: required revision has been compacted"}
2021 @ 08:04:01.131 watch with rev canceled {"error":"etcdserver: mvcc: required revision has been compacted"}
2021 @ 08:04:01.732 watch with rev canceled {"error":"etcdserver: mvcc: required revision has been compacted"}
2021 @ 08:04:01.517 watch with rev canceled {"error":"etcdserver: mvcc: required revision has been compacted"}
2021 @ 08:04:36.689 watch with rev canceled {"error":"etcdserver: mvcc: required revision has been compacted"}
2021 @ 08:04:36.188 watch with rev canceled {"error":"etcdserver: mvcc: required revision has been compacted"}
2021 @ 08:04:36.689 watch with rev canceled {"error":"etcdserver: mvcc: required revision has been compacted"}
```

# watch



```
func newWatchableStore() ... {
    s := &watchableStore{
        store:   NewStore(lg, b, le, ig),
        victimc: make(chan struct{}, 1),
        unsynced: newWatcherGroup(), // 包含所有未同步的watchers
        synced:  newWatcherGroup(), // 包含与存储进程同步的所有同步了的watchers
        stopc:   make(chan struct{}),
    }

    go s.syncWatchersLoop() // 启动每100ms一次的同步未同步映射中的监听者
    go s.syncVictimsLoop() // 同步预先发送未成功的watchers
}

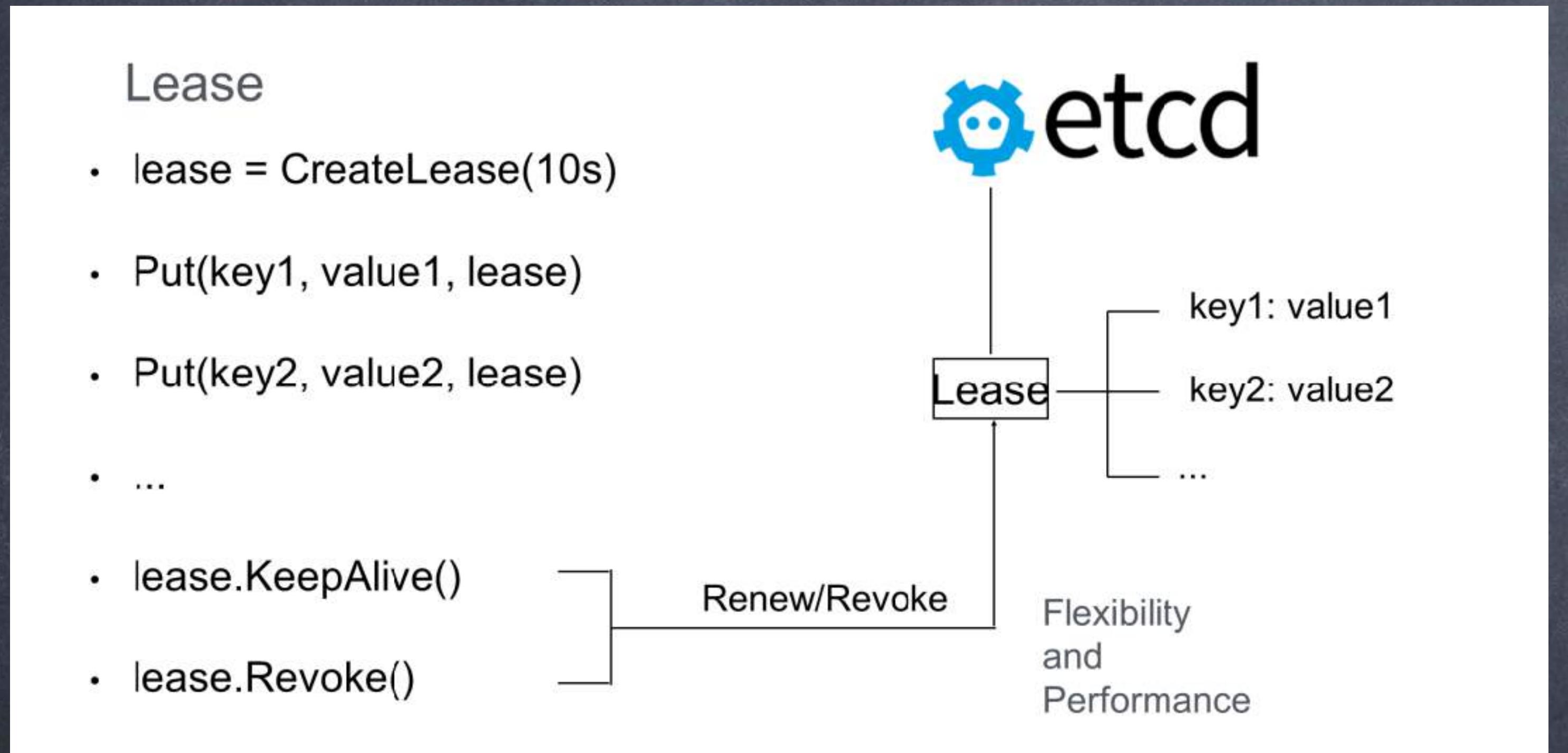
type watcherGroup struct {
    // 通过key可以找到watcher的集合
    keyWatchers watcherSetByKey // hashmap

    // 通过范围和前缀查找watcher的结构
    ranges adt.IntervalTree // 线段树

    // 所有的 watcher 集合
    watchers watcherSet
}
```

- ④ 使用 **hash** 和 **intervalTree** 来组织索引 **watcher**；
- ④ 支持前缀和范围监听；
- ④ **synced watcher** 最新事件的监听；
- ④ **unsynced watcher** 需要推送历史数据；
- ④ **victim watcher** 事件堆积的监听处理；

# lease



- 多个 `key` 可以使用同一个 `lease`；
- 当 `lease` 过期时，对应的多个 `key` 都会被清理掉；

# lease

```
type lessor struct {
    mu sync.RWMutex

    leaseMap      map[LeaseID]*Lease // 通过 id 或者 lease 对象
    leaseHeap     LeaseQueue       // 用来实现定时器的小顶堆
    itemMap       map[LeaseItem]LeaseID // 通过 key 查询关联的 lease id

    minLeaseTTL int64
    expiredC chan []*Lease        // 已经过期的事件
    ...
}

type Lease struct {
    ID          LeaseID
    ttl         int64 // time to live of the lease in seconds
    expiry     time.Time

    // mu protects concurrent accesses to itemSet
    mu      sync.RWMutex
    itemSet map[LeaseItem]struct{}
    revoked chan struct{}}
}

type LeaseItem struct {
    Key string
}

type LeaseID int64
```

```
func (le *lessor) runLoop() {
    defer close(le.doneC)

    for {
        le.revokeExpiredLeases()
        le.checkpointScheduledLeases()

        select {
        case <-time.After(500 * time.Millisecond):
        case <-le.stopC:
            return
        }
    }
}

func (le *lessor) revokeExpiredLeases() {
    var ls []*Lease

    // rate limit
    revokeLimit := leaseRevokeRate / 2

    le.mu.RLock()
    if le.isPrimary() {
        ls = le.findExpiredLeases(revokeLimit)
    }
    le.mu.RUnlock()

    if len(ls) != 0 {
        select {
        case <-le.stopC:
            return
        case le.expiredC <- ls:
        default:
        }
    }
}
```

# txn

```
func handleTrans(etcd *v3.Client, from, to string, amount uint) (bool, error) {
    getResp, err := etcd.Txn(etcd.Ctx()).Then(OpGet(from), OpGet(to)).Commit()
    if err != nil {
        return false, err
    }

    fromKV := getResp.Responses[0].GetRangeResponse().Kvs[0]
    toKV := getResp.Responses[1].GetRangeResponse().Kvs[1]
    fromValue, toValue := toUint64(fromKV.Value), toUint64(toKV.Value)
    if fromValue < amount {
        return false, fmt.Errorf("insufficient value")
    }

    txn := etcd.Txn(etcd.Ctx()).If(
        v3.Compare(v3.ModRevision(from), "=", fromKV.ModRevision),
        v3.Compare(v3.ModRevision(to), "=", toKV.ModRevision))

    txn = txn.Then(
        OpPut(from, fromUint64(fromValue - amount)),
        OpPut(to, fromUint64(toValue - amount)))

    putResp, err := txn.Commit()
    if err != nil {
        return false, err
    }

    return putResp.Succeeded, nil
}
```

Txn().If(cond1, cond2, ...).Then(op1, op2).Else(op1, op2 ...).commit()

- etcd 事务机制实现了多个操作的原子性；
- etcd 的事务为乐观锁 CAS 模式；
- 当所有条件为 true 时，则执行 then 分支逻辑；
- 当为 false 时，执行 else 分支逻辑；
- etcd 的事务机制是 etcd 层 和 boltdb 共同实现的；
- if.then.else 只有在 commit 才一股脑投递给 etcd 处理；

# 存储的大小限制

既然 **boltdb** 中的 **key** 为 **revision** 修订号，如何通过 **key** 找到 **value**？

- **etcd** 会在 内存 中构建了全量索引，内存占用也可观；
- 开机扫描 **db** 来构建 **treeindex** 的开销；
- **boltdb** 在大量存储下造成 **b+tree** 的重平衡和分裂开销；
- **mmap** 把文件映射到进程内存空间，如当前节点内存受限，那么会频繁造成的 **page cache** 缺页中断；

官方建议 **etcd** 的**db**大小不超过 8 GB !!!



other



# 锁

switch by sentinel node



redis 锁

- 简单粗暴，性能好；
- 非公平锁；
- 拿不到锁进行轮询；
- redis主从异常时发生 多锁 故障；

client

lock-k1

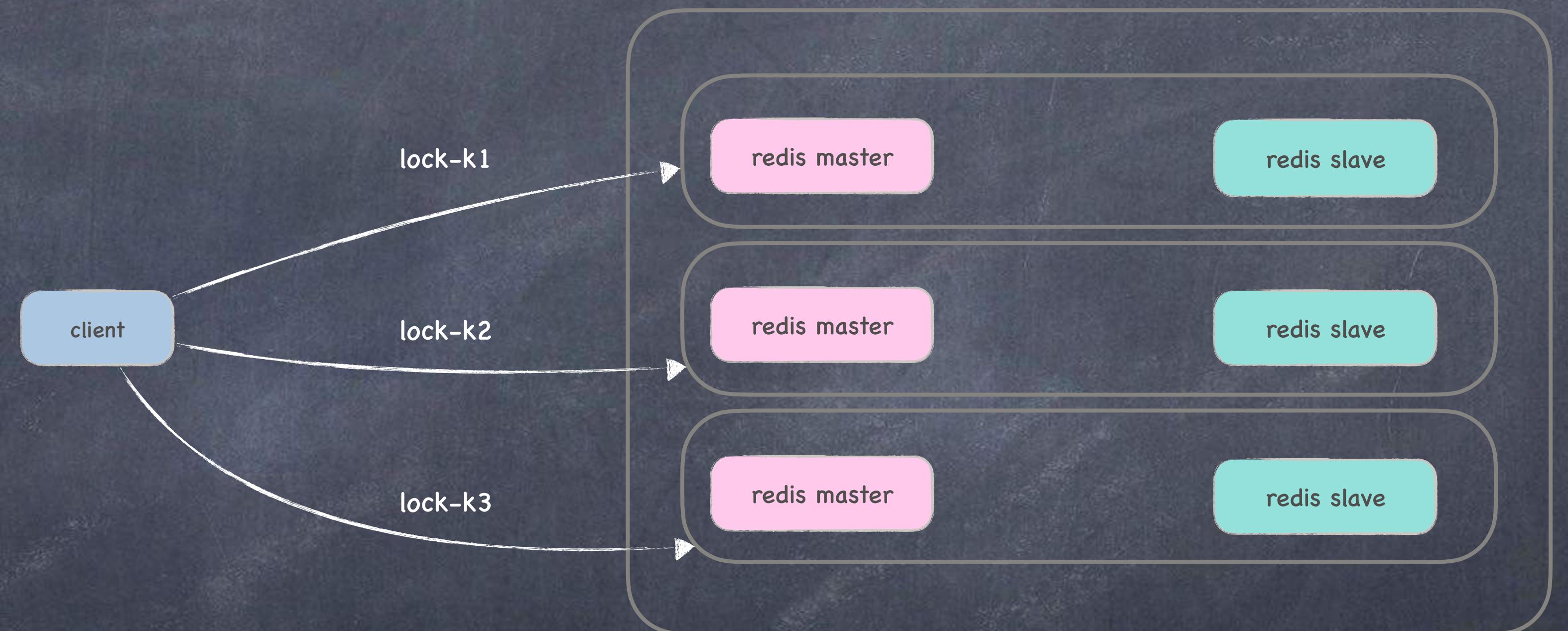
lock-k2

lock-k3

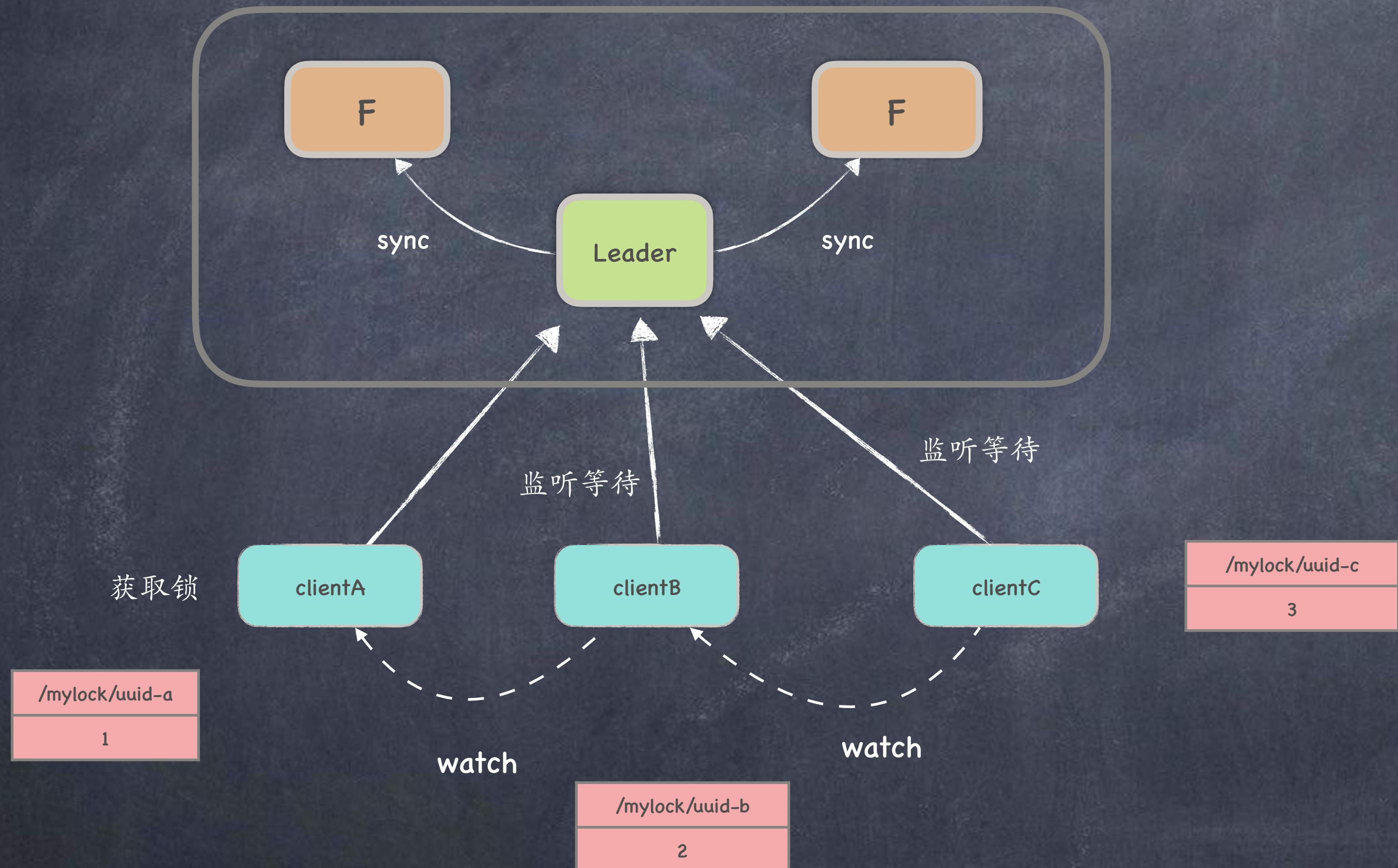
RedLock ???

时钟漂移 ???

redis clusefr

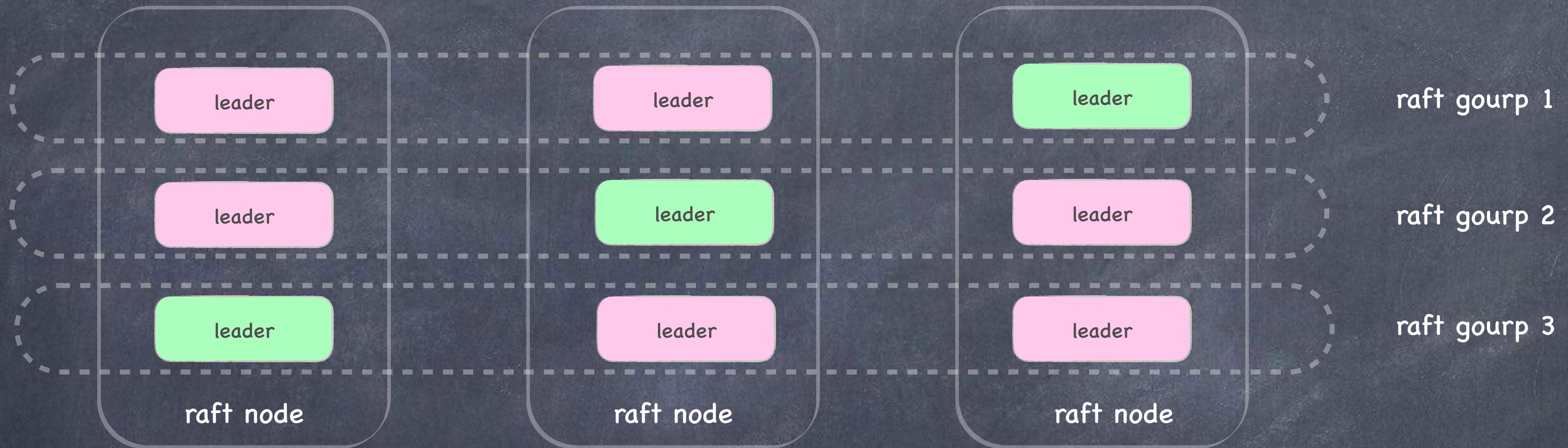


# etcd 锁



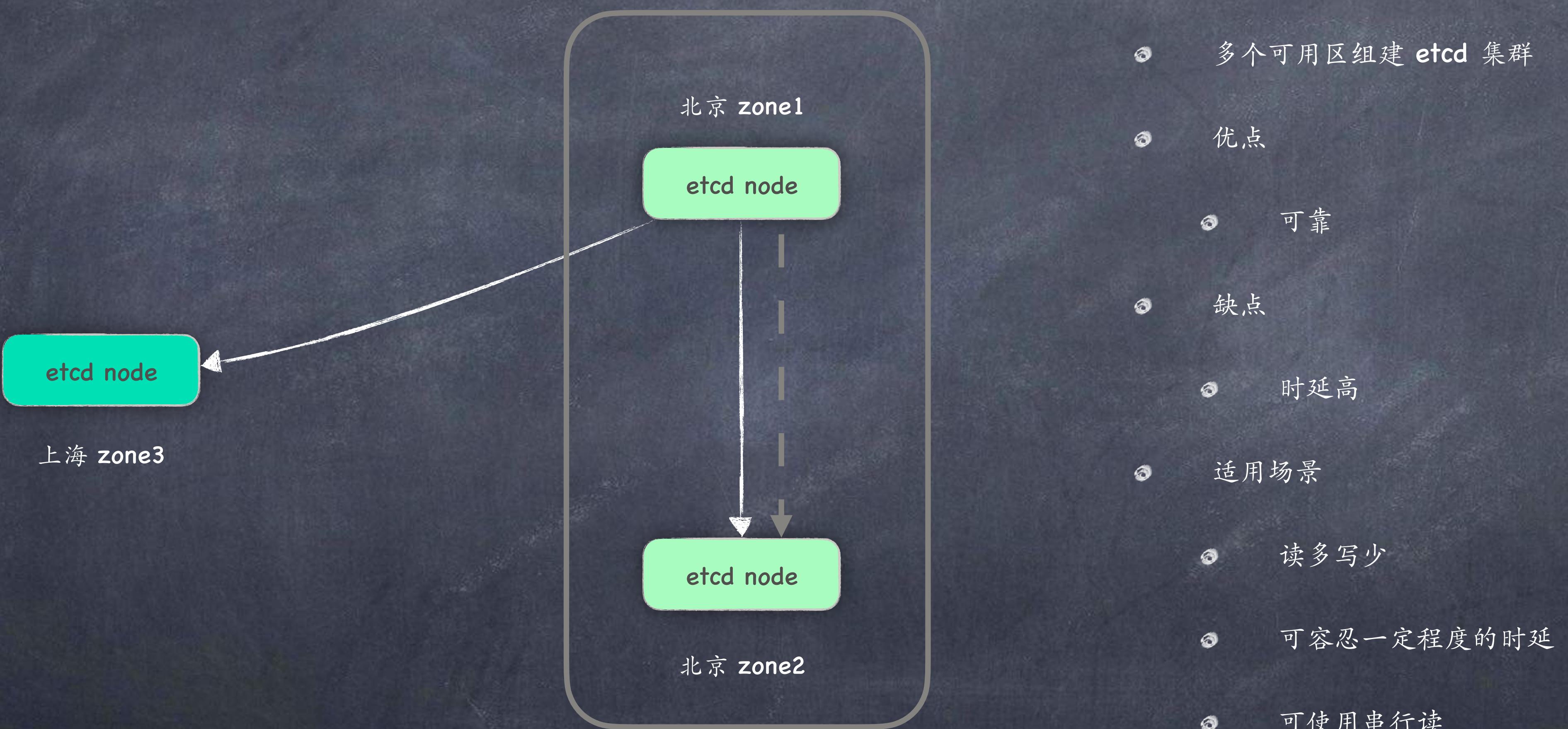
- etcd 实现锁的特点
- 实现了 公平锁 机制
- 避免了 惊群 问题
- 避免了 忙轮询 问题
- 降低了 多锁安全问题 的出现概率
- 网络分区下还是会出现
- 实现机制跟 zookeeper 一样 !!!

# multi raft



- etcd 为单 raft group 模式；
- 一个 raft node 实例只能为一个状态；
- 作为元数据存储的 etcd 性能已足够；

# 多可用区集群



# 读性能

Number of requests	Key size in bytes	Value size in bytes	Number of connections	Number of clients	Consistency	Average read QPS	Average latency per request
10,000	8	256	1	1	Linearizable	1,353	0.7ms
10,000	8	256	1	1	Serializable	2,909	0.3ms
100,000	8	256	100	1000	Linearizable	141,578	5.5ms
100,000	8	256	100	1000	Serializable	185,758	2.2ms

- hardware
  - 8u 16g
  - ssd
  - network latency = 0.1 ms
- benchmark
  - 串行读 = 18.5w
  - 线性读 = 14w
  - 线性读比串行读慢
  - 线性读比串行度时延高一倍
  - 无压力下时延稳定在 < 1ms

串行度 vs 线性读

# 写性能

## ② Benchmark write qps

### ③ SSD

④ WRITE QPS 55000

④ 平均延时 189ms

### ③ 非 SSD

④ WRITE QPS 35000

④ 平均延时 279ms

随着写请求的增大，读性能会显著下降！

Write QPS	Line Read QPS	平均延迟
0	14w	5ms
10	10w	9ms
100	8w	10ms
500	7w	14ms
1000	5.5 w	17 ms
5000	3.5 w	28ms

# tips

- 通过事务来实现批量操作，加快 etcd 的吞吐；
- 如下线主机需要通知 etcd 集群；
- 调整io优先级 ionice -c2 -n0 -p `pgrep etcd`；
- 做好周期性的快照备份；
- quota-backend-bytes 调大 db 空间；



# refer

- <http://thesecretlivesofdata.com/raft/>
- [https://github.com/maemual/raft-zh\\_cn/blob/master/raft-zh\\_cn.md](https://github.com/maemual/raft-zh_cn/blob/master/raft-zh_cn.md)
- <https://github.com/etcd-io/etcd>
- <https://blog.betacat.io/post/2019/08/learn-transaction-isolation-levels-from-etcd/>





# " Q&A "

- [xiaorui.cc](http://xiaorui.cc)

