# Zookeeper: Wait-free coordination for Internet-scale systems 论文阅读笔记

陈超　201922080611

July 15, 2021

## 一致性算法：Zookeeper I

Zookeeper 的一致性保证：

- Linearizable Writes (线性化写入，满足实时性)
  1. Clients send writes to the leader,
     the leader chooses an order, numbered by ”zxid” sends to replicas,
     which all executes in zxid order.
- FIFO client order , each client specifies an order for its operations (reads AND writes) (可以说，单个用户所有的操作，可以看出线性化的，满足实时性。)
  1. Writes: writes appear in the write order in client-specified order .
  2. reads: each read executes at a particular point in the write order .
     a client’s successive reads execute at no-decreasing(非下降的) points int the order.
     a client’s read executes after all previous writes by that client
     a server may block a client’s read to wait for previous write, or sync()

## 一致性算法：Zookeeper II

**Zookeeper 两种基本顺序的总结：**

Linearizable writes: all requests that update the state of Zookeeper are serializable and respect precedence;

FIFO client order: all requests from a given client are executed in the order that they were sent by the client.

## 一致性算法：Zookeeper III

**FAQ:** 为什么尽管是松一致性，Zookeeper 还是这么有用呢?

- Linearizable writes: 线性化的写入，写操作，满足实时性。
- FIFO client order: 单个用户发出的操作是满足可线性化的。单个用户下，能够读到该用户下的写入的最近内容。
- watcher： a ordering guarantee for the notifications。
  - 重点，顺序保证
    - if a client is watching for a change , the client will see the notification event before it sees the new new state of system after the change is made。
- sync() causes subsequent client reads to see preceding writes（之前的写入，比如其它 client 的写入。）useful when a read must see latest data, constitutes slow read。 sync causes a server to apply all pending write requests before processing the read without the overhead of a full write。(轻量的写入操作)
- Writes are well-behaved, e.g. exclusive test-and-set operations. writes really do execute in order, on lastest data.
- Read order rules ensure ”read you own writes”.

## 一致性算法：Zookeeper IV

- Read order rules help reasoning(因果).

# 一致性算法：Zookeeper V

一个例子：

if read sees ”ready” file，subsequent reads see previous writes.(如果一个用户已经看到了”ready” 文件 (有这个前提条件)，那么它必可以看到之前的写入的内容。)

| Write order: | Read order |
|---|---|
| delete(”ready”) | |
| write f1 | |
| write f2 | |
| create(”ready”) | |
| | exists(”ready”) |
| | read f1 |
| | read f2 |

# 一致性算法：Zookeeper VI

watch triggered by a write delivered before reads from subsequence writes.

| Write order: | Read order |
|---|---|
| | exists(”ready”, watch = true) |
| | read f1 |
| delete(”ready”) | |
| write f1 | |
| write f2 | |
| | read f2 |

## 一致性算法：Zookeeper VII

Zookeeper 的一些应用，集群管理、分布式互斥锁、分布式读写锁。

## 一致性算法：Zookeeper VIII

### Group Membership

We take advantage of ephemeral nodes to implement group membership.

- we start by designing a znode, $z_g$ to represent the group.
- when a process member of the group starts, it creates an ephemeral child znode under under $z_g$.
  - if each process has a unique name or identifier, then the name is used as the name of the child znode;
  - the process creates the znode with the SEQUENTIAL flag to obtain a unique name assignment.
- After the child znode is created under $z_g$ the process start normally. It does not need to do anything else. If the process fails or ends, the znode that represents it under $z_g$ is automatically removed.
- Processes can obtain group information by simple listing the children of $z_g$. If a process wants to monitor changes in group membership, the process can set the watch flag to true and refresh the group information(always setting the watch flag to true) when change notifications are received.

## 一致性算法：Zookeeper IX

代码:

```
AddMember:
1: n = create("l + /mem-", EPHEMERAL | SEQUENTIAL)
Monitor:
  //等待l目录下的新member
1. C = getChildren(l, true)
2. //处理问题，rebalance ...
3. goto 1
// rebalance ...
// N 个任务
// M 个节点
//假设 N <= M
int k = M / N;
int r = M % N;
int j = 0;
for(int i = 0; i < M; i++) {
    for(int k = 0; k < N; k++) {
        node[i++] = task[j];
    }
    if(r > 0)
       node[i++] = task[j];
    j++;
}
```

## 一致性算法：Zookeeper X

### 分布式互斥锁 (Simple Lock)

The simplest lock implementation use "lock files", The lock is represented by a node.

- To a acquire a lock,
- a client tries to create the designated znode with the EPHEMERAL flag.
    - If the create succeeds, the client holds the lock.
    - Otherwise, the client can read the znode with the watch flag set to be notified if the current leader dies. A client releases the lock when it dies(session/lease expired) or explicitly deletes the znode.
- Other clients that are waiting for a lock try again to acquire a lock once they observe the znode being deleted.

## 一致性算法：Zookeeper XI

Simple Lock 的问题,

- 一旦释放了 lock，那么等待的所有节点，必然会被唤醒，然后进行竞争，导致 zookeeper 的通信量增加。这样现象被叫做惊群现象 (Herb Effect)

如何解决，让锁请求排队，先到的锁，先被唤醒。

# 一致性算法：Zookeeper XII

## Simple Locks without Herb Effect

- We define a lock znode / to implement such locks.
- we line up all the client requesting the lock (FIFO 排序) and each client obtains the lock in order of request arrival.

## 一致性算法：Zookeeper XIII

实现 Simple Locks without Herb Effect 代码如下：

```
Lock func
//自增序号，创建znode节点
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
//读取l目录下的所有znodes
2 C = getChildren(l, false)
//如果，当前znode是l目录中的最老的，表示获得该互斥锁
3 if n is the lowest znode in C, exit
//找到仅仅排在n的最前面的节点
4 p = znode in C ordered just before n
//等待，被唤醒
5 if exists(p, true) wait for watch event
//被唤醒(主动释放或者节点崩溃)，重新检查，
//是否有更小的序列的锁请求在{等待或者持有锁}。
6 goto 2
Unlock func
1 delete(n)
```

## 一致性算法：Zookeeper XIV

设计的好处：

- The removal of znode only causes one client to wake up, since each znode is waited by exactly one other client, so we do not have the herd effect(惊群);
- There is no polling or timeouts;
- Because of the way we have implemented locking, we can see by browsing the Zookeeper data the amount of lock contention, and debug locking problems.

## 一致性算法：Zookeeper XV

### 分布式读写锁

ReadWriteLock

- the readLock may be held simutaneously by multiple read threads/nodes, so long as there are no writers.
- the writeLock is exclusive.

## 一致性算法：Zookeeper XVI

实现代码，have separate read lock and write lock procedures. the unlock procedure is the same as the global lock case.

```
Write Lock
1. n = create(l + "/write-", EPHEMERAL | SEQUENTIAL)
//得到目录下的所有znodes
2. C = getChildren(l, false)
//如果n是最老的节点， 获得互斥的写锁
3. if n is the lowest znode in C, exit
//找到仅仅排在n的最前面的节点
4. p = znode in C ordered just before n
//等待， 并唤醒
5 if exists(p, true) wait for watch event
//被唤醒(主动释放或者节点崩溃)， 重新检查，
//是否有更小的序列的锁请求在{等待或者持有锁}。
6 goto 2
Read Lock
1. n = create(l + "/read-", EPHEMERAL | SEQUENTIAL)
2. C = getChildren(l, false)
//没有老的writeLock请求在等待锁， 直接获得锁
3. if no write znodes lower than n in C, exit
//得到排在前面的写request znode
4. p = write znode in C ordered just before n
//等待， 并唤醒
5. if exists(p, true) wait for watch event
//被唤醒(主动释放或者节点崩溃)， 重新检查，
//是否有更小的序列的锁请求在{等待或者持有锁}。
6 goto 2
```

# 一致性算法：Zookeeper XVII

设计的好处:

- Since read locks may be sharded, lines 3 and 4 slightly because only earlier write lock znodes prevent the client from obtaining a read lock.
    - write Lock: if n is the lowest znode in C, exit
    - read lock: if no write znodes lower than n in C, exit
- we have "herd effect"
    - p = write znode in C ordered just before n, if exits(p, true) wait fot watch event.

  when there are serveral clients waiting for a read lock and get notified when the "write-" znode with the lower sequence number is deleted; in fact, this is a desired behavior, all those read clients should be released since they may now have the lock.