主讲老师： Fox

```
文档：1. Zookeeper特性与节点数据类型详解.n...
链接：http://note.youdao.com/noteshare?id=f0549278905bb988c831d6910c54143a&sub=67F7F974690A43B49495E6A5ACB6A594
```


CAP&Base理论

- CAP 理论
- BASE 理论

Zookeeper介绍

- 什么是Zookeeper

Zookeeper实战

- Zookeeper安装
- 客户端命令行操作
  - 常见cli命令
- ZooKeeper数据结构
  - 节点分类
  - 节点状态信息
  - 监听通知（watcher）机制
  - Zookeeper 节点特性总结
  - 应用场景
- Zookeeper集群
  - 集群角色
  - 集群架构
  - 三节点Zookeeper集群搭建

Zookeeper四字命令

Zookeeper Leader 选举原理

Zookeeper 数据同步流程

CAP&Base理论

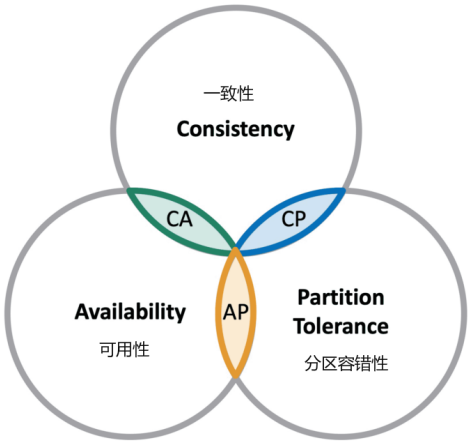
CAP 理论

CAP 理论指出对于一个分布式计算系统来说，不可能同时满足以下三点：

- **一致性**：在分布式环境中，一致性是指数据在多个副本之间是否能够保持一致的特性，等同于所有节点访问同一份最新的数据副本。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作后，应该保证系统的数据仍然处于一致的状态。
- **可用性**：每次请求都能获取到正确的响应，但是不保证获取的数据为最新数据。
- **分区容错性**：分布式系统在遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障。

一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。

在这三个基本需求中，最多只能同时满足其中的两项，P 是必须的，因此只能在 CP 和 AP 中选择，zookeeper 保证的是 CP，对比 spring cloud 系统中的注册中心 eureka实现的是 AP。



思考：zookeeper是强一致性吗?

BASE 理论

BASE 是 Basically Available(基本可用)、Soft-state(软状态) 和 Eventually Consistent(最终一致性) 三个短语的缩写。

- **基本可用：**在分布式系统出现故障，允许损失部分可用性（服务降级、页面降级）。
- **软状态：**允许分布式系统出现中间状态。而且中间状态不影响系统的可用性。这里的中间状态是指不同的 data replication（数据备份节点）之间的数据更新可以出现延时的最终一致性。
- **最终一致性：**data replications 经过一段时间达到一致性。

BASE 理论是对 CAP 中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性。

> 强一致性:又称线性一致性(linearizability)
> 1.任意时刻，所有节点中的数据是一样的,
> 2.一个集群需要对外部提供强一致性，所以只要集群内部某一台服务器的数据发生了改变，那么就需要等待集群内其他服务器的数据同步完成后，才能正常的对外提供服务
> 3.保证了强一致性，务必会损耗可用性
>
> 弱一致性:
> 1.系统中的某个数据被更新后，后续对该数据的读取操作可能得到更新后的值，也可能是更改前的值。
> 2.即使过了不一致时间窗口，后续的读取也不一定能保证一致。
>
> 最终一致性:
> 1.弱一致性的特殊形式,不保证在任意时刻任意节点上的同一份数据都是相同的，但是随着时间的迁移，不同节点上的同一份数据总是在向趋同的方向变化
> 2.存储系统保证在没有新的更新的条件下，最终所有的访问都是最后更新的值
>
> 顺序一致性:
> 1.任何一次读都能读到某个数据的最近一次写的数据。
> 2.对其他节点之前的修改是可见(已同步)且确定的,并且新的写入建立在已经达成同步的基础上。

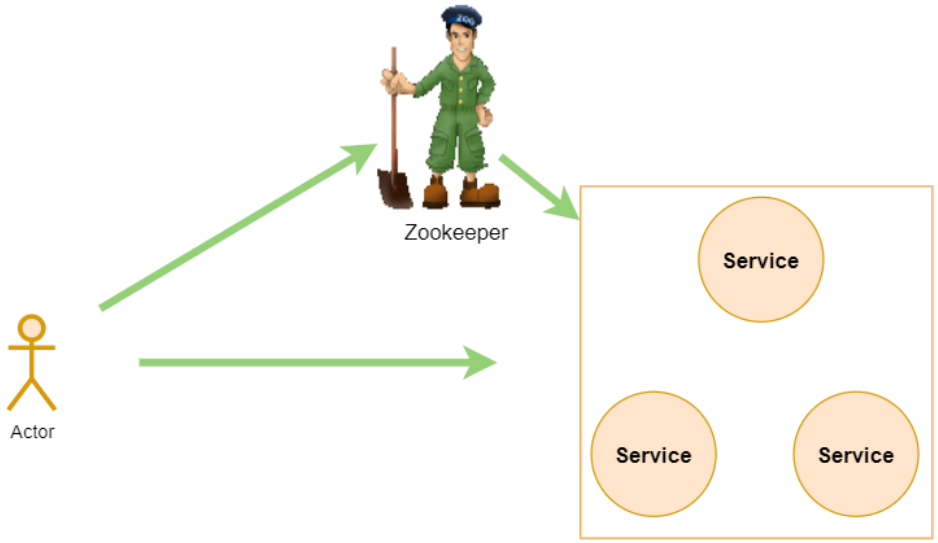
Zookeeper写入是强一致性,读取是顺序一致性。

Zookeeper介绍

什么是Zookeeper

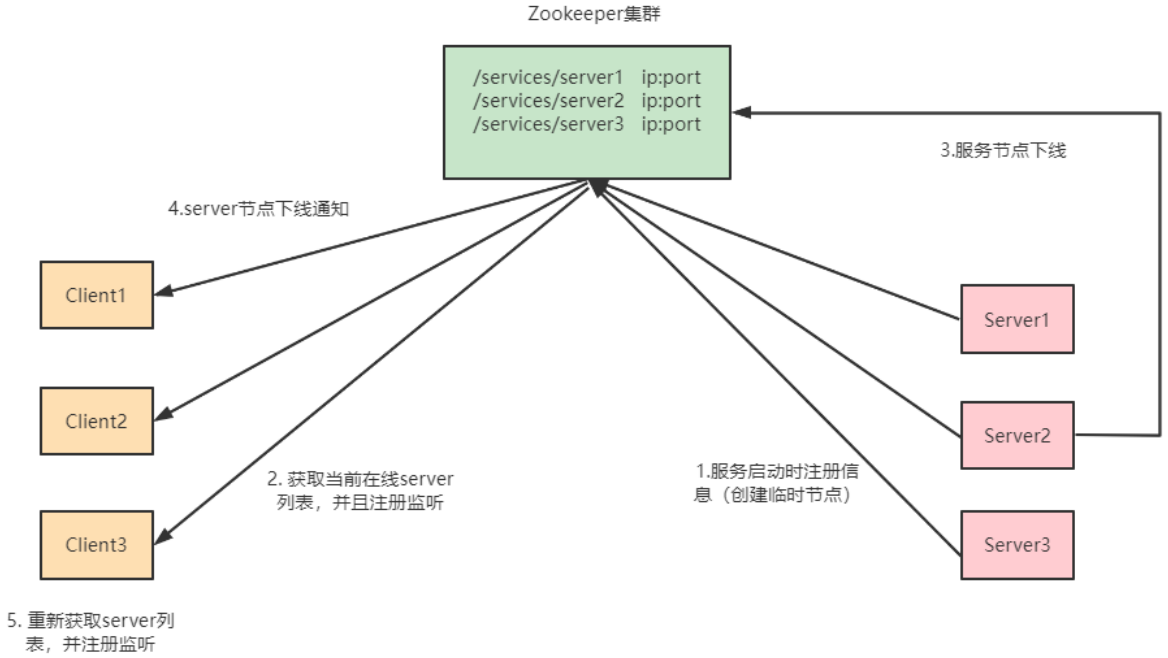
ZooKeeper 是一个开源的分布式协调框架，是Apache Hadoop 的一个子项目，主要用来解决分布式集群中应用系统的一致性问题。Zookeeper 的设计目标是将那些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的接口提供给用户使用。

官方：https://zookeeper.apache.org/



ZooKeeper本质上是一个分布式的小文件存储系统（Zookeeper=文件系统+监听机制）。提供基于类似于文件系统的目录树方式的数据存储，并且可以对树中的节点进行有效管理，从而用来维护和监控存储的数据的状态变化。通过监控这些数据状态的变化，从而可以达到基于数据的集群管理、统一命名服务、分布式配置管理、分布式消息队列、分布式锁、分布式协调等功能。

Zookeeper从设计模式角度来理解：是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后接受观察者的注册，一旦这些数据的状态发生变化，Zookeeper 就将负责通知已经在Zookeeper上注册的那些观察者做出相应的反应。

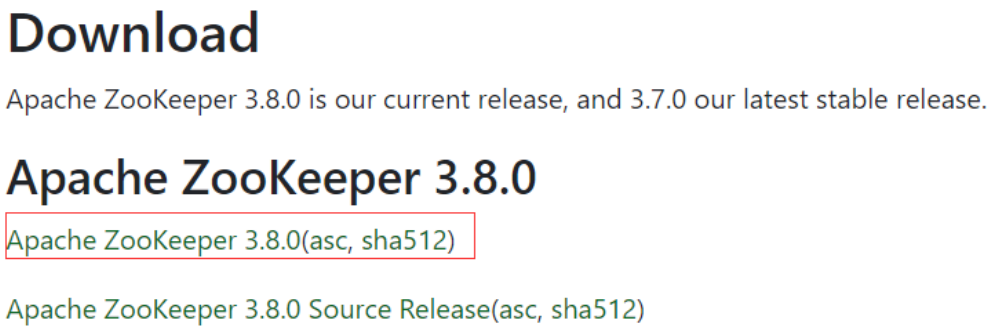


Zookeeper实战

Zookeeper安装

下载地址：https://zookeeper.apache.org/releases.html

运行环境：jdk8



1）修改配置文件

解压安装包后进入conf目录，复制zoo_sample.cfg，修改为zoo.cfg

```
cp zoo_sample.cfg zoo.cfg
```

修改 zoo.cfg 配置文件，将 dataDir=/tmp/zookeeper 修改为指定的data目录

zoo.cfg中参数含义：

```
# zookeeper时间配置中的基本单位 (毫秒)
tickTime=2000
# 允许follower初始化连接到leader最大时长，它表示tickTime时间倍数 即:initLimit*tickTime
initLimit=10
# 允许follower与leader数据同步最大时长,它表示tickTime时间倍数
syncLimit=5
#zookeper 数据存储目录及日志保存目录（如果没有指明dataLogDir，则日志也保存在这个文件中）
dataDir=/tmp/zookeeper
#对客户端提供的端口号
clientPort=2181
#单个客户端与zookeeper最大并发连接数
maxClientCnxns=60
# 保存的数据快照数量，之外的将会被清除
autopurge.snapRetainCount=3
#自动触发清除任务时间间隔，小时为单位。默认为0，表示不自动清除。
autopurge.purgeInterval=1
```

2）启动zookeeper server

```
# 可以通过 bin/zkServer.sh 来查看都支持哪些参数
# 默认加载配置路径conf/zoo.cfg
bin/zkServer.sh start conf/zoo.cfg

# 查看zookeeper状态
bin/zkServer.sh status
```

3）启动zookeeper client连接Zookeeper server

```
bin/zkCli.sh
# 连接远程的zookeeper server
bin/zkCli.sh -server ip:port
```

客户端命令行操作

输入命令 help 查看zookeeper支持的所有命令：

```
[zk: localhost:2181(CONNECTED) 1] help
ZooKeeper -server host:port -client-configuration properties-file cmd args
        addWatch [-m mode] path # optional mode is one of [PERSISTENT, PERSISTENT_RECURSIVE] - default is PERSISTENT_RECURSIVE
        addauth scheme auth
        close
        config [-c] [-w] [-s]
        connect host:port
        create [-s] [-e] [-c] [-t ttl] path [data] [acl]
        delete [-v version] path
        deleteall path [-b batch size]
        delquota [-n|-b|-N|-B] path
        get [-s] [-w] path
        getAcl [-s] path
        getAllChildrenNumber path
        getEphemerals path
        history
        listquota path
        ls [-s] [-w] [-R] path
        printwatches on|off
        quit
        reconfig [-s] [-v version] [[-file path] | [-members serverID=host:port1:port2;port3[,...]*]] | [-add serverId=host:port1
:port2;port3[,...]]* [-remove serverId[,...]*]
        redo cmdno
```

常见cli命令

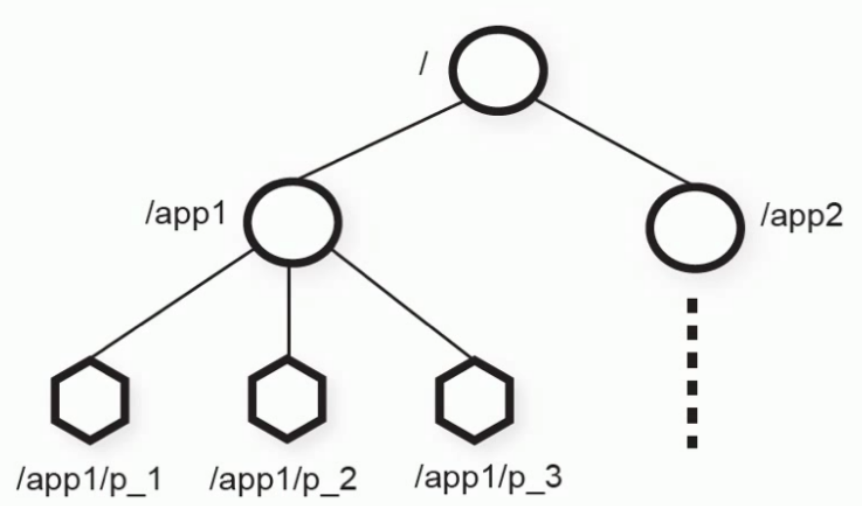
https://zookeeper.apache.org/doc/r3.8.0/zookeeperCLI.html

命令基本语法	功能描述
help	显示所有操作命令
ls [-s] [-w] [-R] path	使用 ls 命令来查看当前 znode 的子节点 [可监听] -w: 监听子节点变化 -s: 节点状态信息（时间戳、版本号、数据大小等） -R: 表示递归的获取
create [-s] [-e] [-c] [-t ttl] path [data] [acl]	创建节点 -s：创建有序节点。 -e：创建临时节点。 -c：创建一个容器节点。 t ttl]：创建一个TTL节点， -t 时间（单位毫秒）。 data：节点的数据，可选，如果不使用时，节点数据就为null。 acl：访问控制
get [-s] [-w] path	获取节点数据信息 -s: 节点状态信息（时间戳、版本号、数据大小等） -w: 监听节点变化
set [-s] [-v version] path data	设置节点数据 -s:表示节点为顺序节点 -v: 指定版本号
getAcl [-s] path	获取节点的访问控制信息 -s: 节点状态信息（时间戳、版本号、数据大小等）
setAcl [-s] [-v version] [-R] path acl	设置节点的访问控制列表 -s:节点状态信息（时间戳、版本号、数据大小等） -v:指定版本号

	-R:递归的设置
stat [-w] path	查看节点状态信息
delete [-v version] path	删除某一节点，只能删除无子节点的节点。 -v： 表示节点版本号
deleteall path	递归的删除某一节点及其子节点
setquota -n|-b val path	对节点增加限制 n:表示子节点的最大个数 b:数据值的最大长度，-1表示无限制

ZooKeeper数据结构

ZooKeeper 数据模型的结构与 Unix 文件系统很类似，整体上可以看作是一棵树，每个节点称做一个 ZNode。



ZooKeeper的数据模型是层次模型，层次模型常见于文件系统。层次模型和key-value模型是两种主流的数据模型。ZooKeeper使用文件系统模型主要基于以下两点考虑:

1. 文件系统的树形结构便于表达数据之间的层次关系
2. 文件系统的树形结构便于为不同的应用分配独立的命名空间(namespace)

ZooKeeper的层次模型称作Data Tree，Data Tree的每个节点叫作Znode。不同于文件系统，每个节点都可以保存数据，每一个 ZNode 默认能够存储 1MB 的数据，每个 ZNode 都可以通过其路径唯一标识，每个节点都有一个版本(version)，版本从0开始计数。

```
public class DataTree {
  private final ConcurrentHashMap<String, DataNode> nodes =
  new ConcurrentHashMap<String, DataNode>();


  private final WatchManager dataWatches = new WatchManager();
  private final WatchManager childWatches = new WatchManager();

}
```

```
10
11 public class DataNode implements Record {
12  byte data[];
13  Long acl;
14  public StatPersisted stat;
15  private Set<String> children = null;
16  }
```

节点分类

一个znode可以使持久性的，也可以是临时性的:

1. 持久节点(PERSISTENT): 这样的znode在创建之后即使发生ZooKeeper集群宕机或者client宕机也不会丢失。

2. 临时节点(EPHEMERAL): client宕机或者client在指定的timeout时间内没有给ZooKeeper集群发消息，这样的znode就会消失。

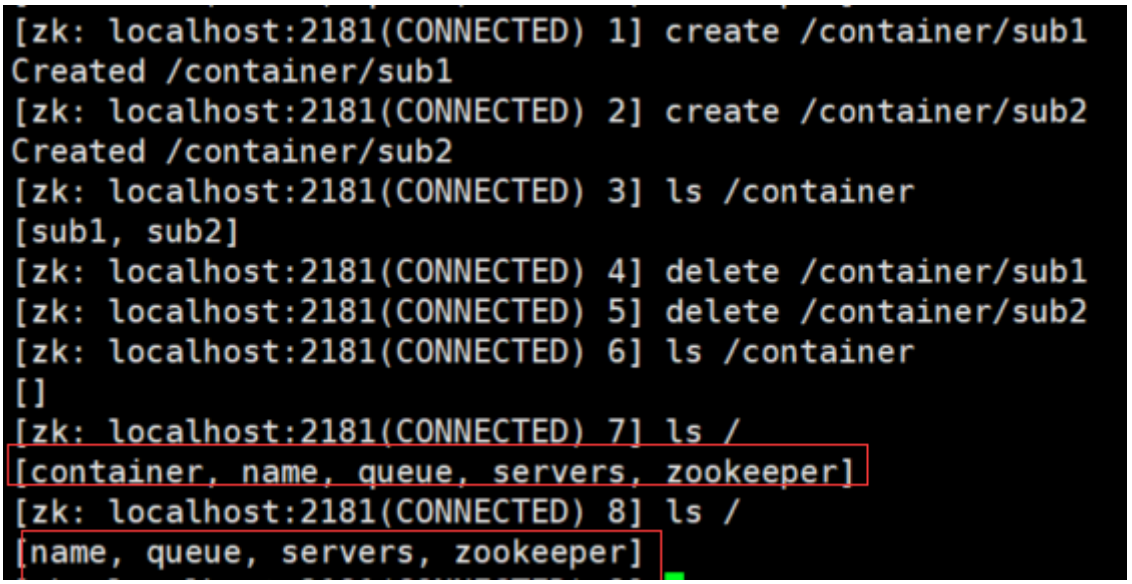
如果上面两种znode具备顺序性，又有以下两种znode :

3. 持久顺序节点(PERSISTENT_SEQUENTIAL): znode除了具备持久性znode的特点之外，znode的名字具备顺序性。

4. 临时顺序节点(EPHEMERAL_SEQUENTIAL): znode除了具备临时性znode的特点之外，zorde的名字具备顺序性。

zookeeper主要用到的是以上4种节点。

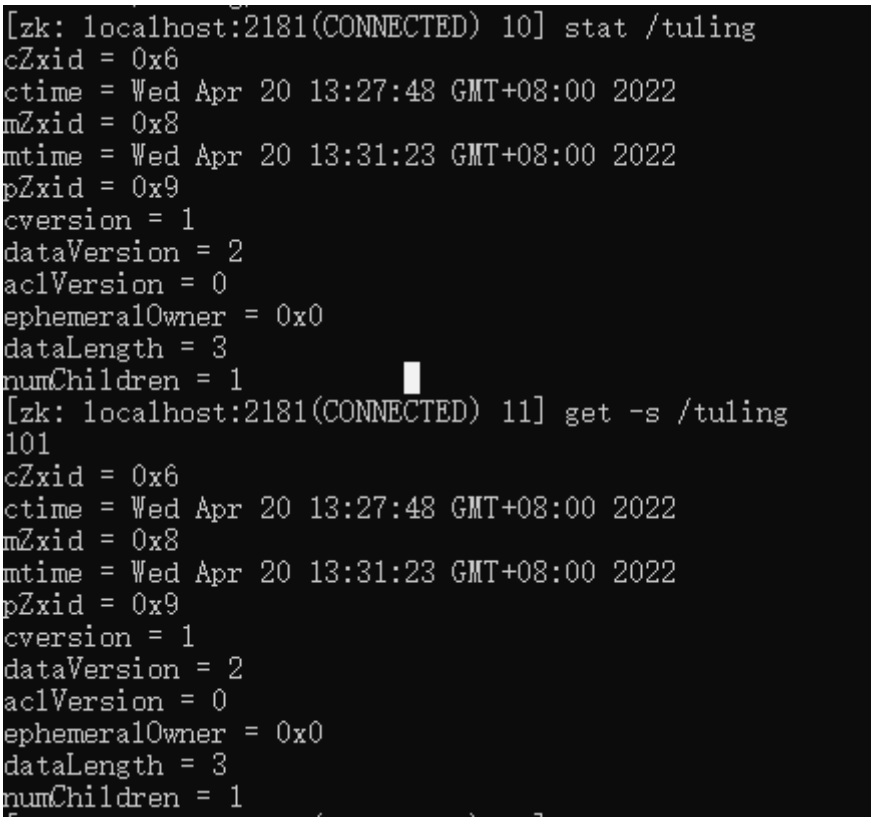
5. Container节点 (3.5.3版本新增)：Container容器节点，当容器中没有任何子节点，该容器节点会被zk定期删除（定时任务默认60s 检查一次)。 和持久节点的区别是 ZK 服务端启动后，会有一个单独的线程去扫描，所有的容器节点，当发现容器节点的子节点数量为 0 时，会自动删除该节点。可以用于 leader 或者锁的场景中。



6. TTL节点：带过期时间节点，默认禁用，需要在zoo.cfg中添加extendedTypesEnabled=true 开启。 注意：ttl不能用于临时节点

```
#创建持久节点
create /servers xxx
#创建临时节点
create -e /servers/host xxx
#创建临时有序节点
create -e -s /servers/host xxx
#创建容器节点
create -c /container xxx
# 创建ttl节点
 create -t 10 /ttl

```

节点状态信息



- cZxid ： Znode创建的事务id。
- ctime：节点创建时的时间戳。
- mZxid ： Znode被修改的事务id，即每次对znode的修改都会更新mZxid。

对于zk来说，每次的变化都会产生一个唯一的事务id，zxid（ZooKeeper Transaction Id），通过zxid，可以确定更新操作的先后顺序。例如，如果zxid1小于zxid2，说明zxid1操作先于zxid2发生，zxid对于整个zk都是唯一的，即使操作的是不同的znode。

- pZxid: 表示该节点的子节点列表最后一次修改的事务ID，添加子节点或删除子节点就会影响子节点列表，但是修改子节点的数据内容则不影响该ID（注意: 只有子节点列表变更了才会变更pzxid，子节点内容变更不会影响pzxid）
- mtime：节点最新一次更新发生时的时间戳.
- cversion ： 子节点的版本号。当znode的子节点有变化时，cversion 的值就会增加1。
- dataVersion：数据版本号，每次对节点进行set操作，dataVersion的值都会增加1（即使设置的是相同的数据），可有效避免了数据更新时出现的先后顺序问题。
- ephemeralOwner:如果该节点为临时节点, ephemeralOwner值表示与该节点绑定的session id。如果不是, ephemeralOwner值为0(持久节点)。

在client和server通信之前,首先需要建立连接,该连接称为session。连接建立后,如果发生连接超时、授权失败,或者显式关闭连接,连接便处于closed状态, 此时session结束。

- dataLength ： 数据的长度

- numChildren ： 子节点的数量（只统计直接子节点的数量）

监听通知（watcher）机制

- 一个Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端，以便通知它们。
- Zookeeper采用了 Watcher机制实现数据的发布订阅功能，多个订阅者可同时监听某一特定主题对象，当该主题对象的自身状态发生变化时例如节点内容改变、节点下的子节点列表改变等，会实时、主动通知所有订阅者。
- watcher机制事件上与观察者模式类似，也可看作是一种观察者模式在分布式场景下的实现方式。

watcher的过程：

1. 客户端向服务端注册watcher
2. 服务端事件发生触发watcher
3. 客户端回调watcher得到触发事件情况

注意：Zookeeper中的watch机制，必须客户端先去服务端注册监听，这样事件发送才会触发监听，通知给客户端。

支持的事件类型：

- None：连接建立事件
- NodeCreated： 节点创建
- NodeDeleted： 节点删除
- NodeDataChanged：节点数据变化
- NodeChildrenChanged：子节点列表变化
- DataWatchRemoved：节点监听被移除
- ChildWatchRemoved：子节点监听被移除

特性	说明
一次性触发	watcher是一次性的，一旦被触发就会移除，再次使用时需要重新注册
客户端顺序回调	watcher回调是顺序串行执行的，只有回调后客户端才能看到最新的数据状态。一个watcher回调逻辑不应该太多，以免影响别的watcher执行
轻量级	WatchEvent是最小的通信单位，结构上只包含通知状态、事件类型和节点路径，并不会告诉数据节点变化前后的具体内容
时效性	watcher只有在当前session彻底失效时才会无效，若在session有效期内快速重连成功，则watcher依然存在，仍可接收到通知；

```
1  #监听节点数据的变化
```

```
get -w path
stat -w path
#监听子节点增减的变化
ls -w path
```

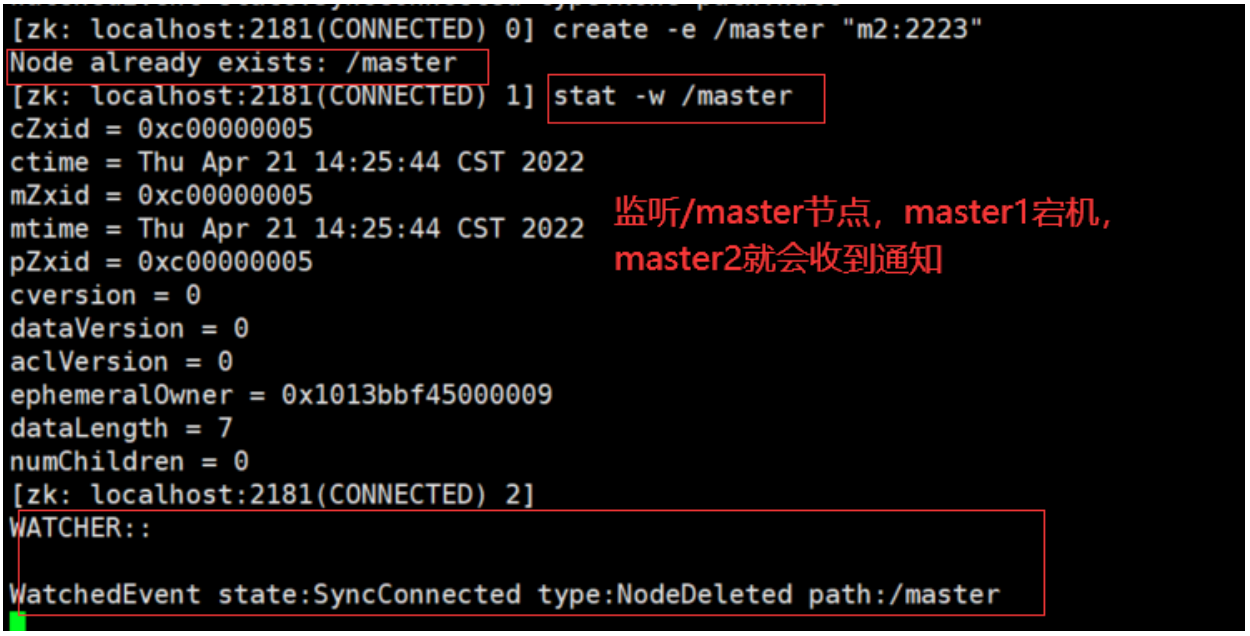
使用案例——协同服务

设计一个master-worker的组成员管理系统，要求系统中只能有一个master，master能实时获取系统中worker的情况。

保证组里面只有一个master的设计思路

```
#master1
create -e /master "m1:2223"

#master2
create -e /master "m2:2223" # /master已经存在，创建失败
Node already exists: /master
#监听/master节点
stat -w /master
#当master2收到/master节点删除通知后可以再次发起创建节点操作
 create -e /master "m2:2223"
```

master-slave选举也可以用这种方式



master监控worker状态的设计思路

```
#master服务
create /workers
#让master服务监控/workers下的子节点
ls -w /workers

#worker1
```

```
create -e /workers/w1 "w1:2224" #创建子节点，master服务会收到子节点变化通知

#master服务
ls -w /workers
#worker2
create -e /workers/w2 "w2:2224" #创建子节点，master服务会收到子节点变化通知

#master服务
ls -w /workers
#worker2
quit #worker2退出，master服务会收到子节点变化通知
```

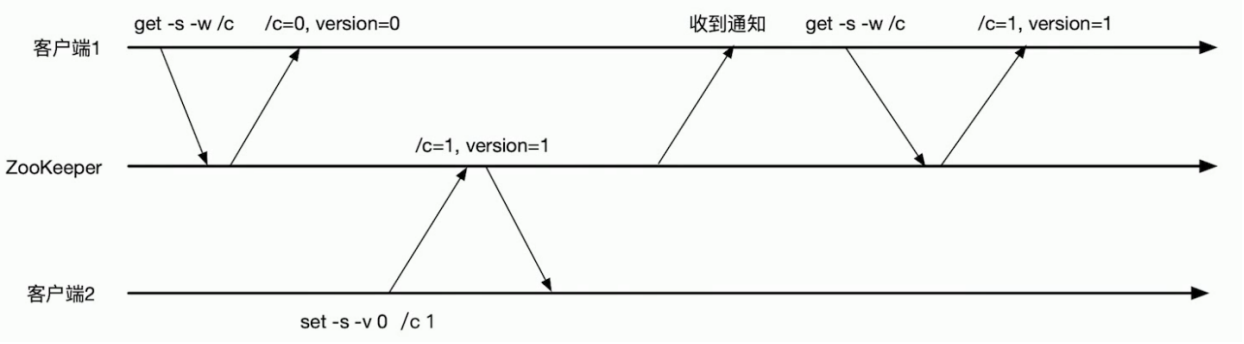
```
[zk: localhost:2181(CONNECTED) 6] create -e /workers/w1 "w1:2224"

WATCHER::

WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/workers
Created /workers/w1
[zk: localhost:2181(CONNECTED) 7]
```

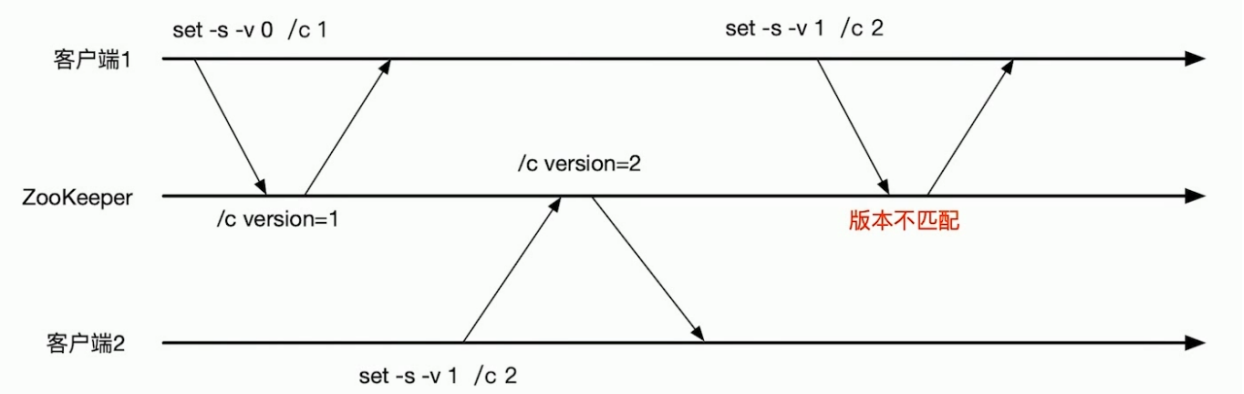
使用场景——条件更新

设想用2 /c实现一个counter，使用set命令来实现自增1操作。条件更新场景：

1. 客户端1把/c更新到版本1，实现/c的自增1。
2. 客户端2把/c更新到版本2，实现/c的自增1。
3. 客户端1不知道/c已经被客户端2 更新过了，还用过时的版本1是去更新/c，更新失败。如果客户端1使用的是无条件更新，/c就会更新为2，没有实现自增1。

使用条件更新可以避免出现客户端基于过期的数据进行数据更新的操作。





Zookeeper 节点特性总结

1. 同一级节点 key 名称是唯一的

```
[zk: localhost:2181(CONNECTED) 1] create /lock
Created /lock
[zk: localhost:2181(CONNECTED) 2] create /lock
Node already exists: /lock
```

已存在/lock节点，再次创建会提示已经存在

2.创建节点时，必须要带上全路径

3.session 关闭，临时节点清除

4.自动创建顺序节点

```
[zk: localhost:2181(CONNECTED) 17] create  /queue
Created /queue
[zk: localhost:2181(CONNECTED) 18] create -e -s /queue/host1
Created /queue/host10000000000
[zk: localhost:2181(CONNECTED) 19] create -e -s /queue/host2
Created /queue/host20000000001
[zk: localhost:2181(CONNECTED) 20] create -e -s /queue/host3
Created /queue/host30000000002
[zk: localhost:2181(CONNECTED) 21] create -e -s /queue/host4
Created /queue/host40000000003
[zk: localhost:2181(CONNECTED) 22] ls -R /queue
/queue
/queue/host10000000000
/queue/host20000000001
/queue/host30000000002
/queue/host40000000003
```

5.watch 机制，监听节点变化

事件监听机制类似于观察者模式，watch 流程是客户端向服务端某个节点路径上注册一个 watcher，同时客户端也会存储特定的 watcher，当节点数据或子节点发生变化时，服务端通知客户端，客户端进行回调处理。特别注意：监听事件被单次触发后，事件就失效了。

6.delete 命令只能一层一层删除。提示：新版本可以通过 deleteall 命令递归删除。

应用场景

ZooKeeper适用于存储和协同相关的关键数据，不适合用于大数据量存储。

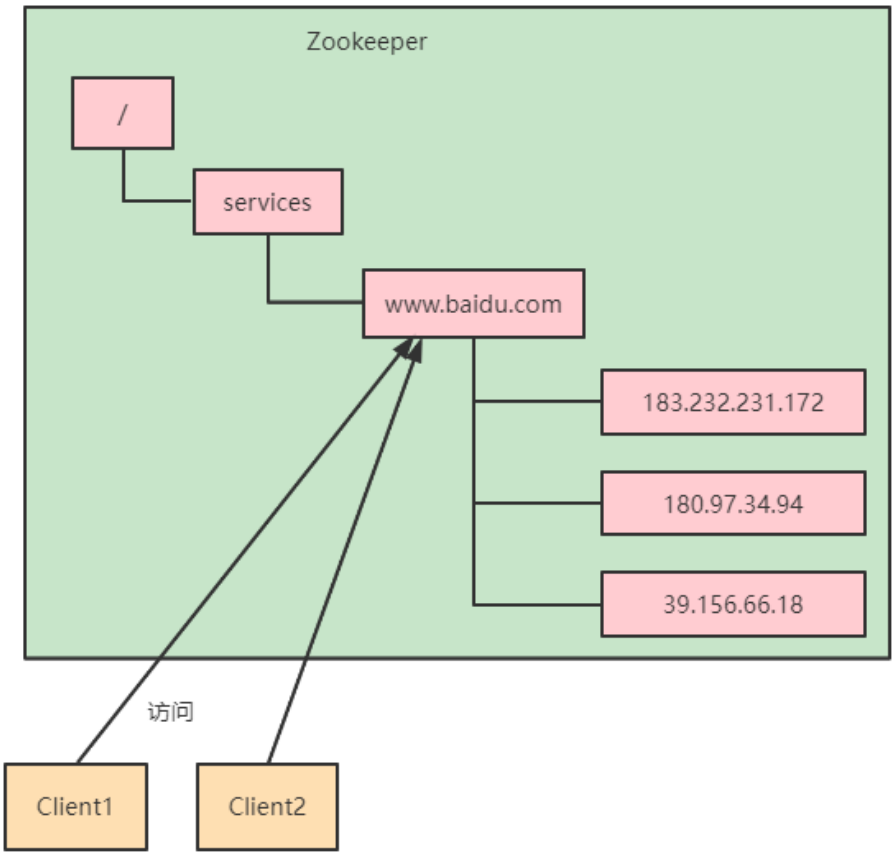
有了上述众多节点特性，使得 zookeeper 能开发不出不同的经典应用场景，比如：

- 注册中心
- 数据发布/订阅（常用于实现配置中心）
- 负载均衡
- 命名服务
- 分布式协调/通知
- 集群管理
- Master选举
- 分布式锁
- 分布式队列

统一命名服务

在分布式环境下，经常需要对应用/服务进行统一命名，便于识别。

例如：IP不容易记住，而域名容易记住。



利用 ZooKeeper 顺序节点的特性，制作分布式的序列号生成器，或者叫 id 生成器。（分布式环境下使用作为数据库 id，另外一种是 UUID（缺点：没有规律）），ZooKeeper 可以生成有顺序的容易理解的同时支持分布式环境的编号。

```
/
└── /order
  ├── /order-date1-000000000000001
  ├── /order-date2-000000000000002
  ├── /order-date3-000000000000003
  ├── /order-date4-000000000000004
  └── /order-date5-000000000000005
```

数据发布/订阅

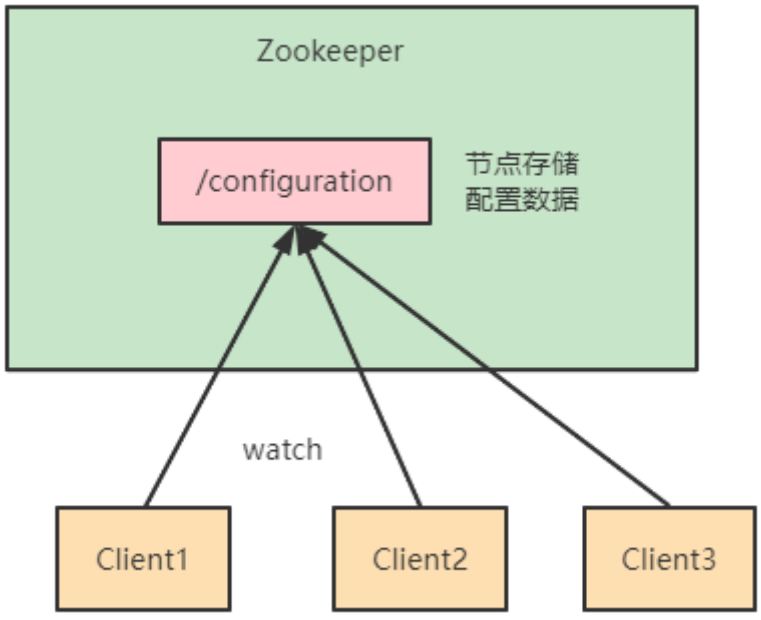
数据发布/订阅的一个常见的场景是配置中心，发布者把数据发布到 ZooKeeper 的一个或一系列的节点上，供订阅者进行数据订阅，达到动态获取数据的目的。

配置信息一般有几个特点:

1. 数据量小的KV
2. 数据内容在运行时会发生动态变化
3. 集群机器共享，配置一致

ZooKeeper 采用的是推拉结合的方式。

1. 推: 服务端会推给注册了监控节点的客户端 Watcher 事件通知
2. 拉: 客户端获得通知后，然后主动到服务端拉取最新的数据

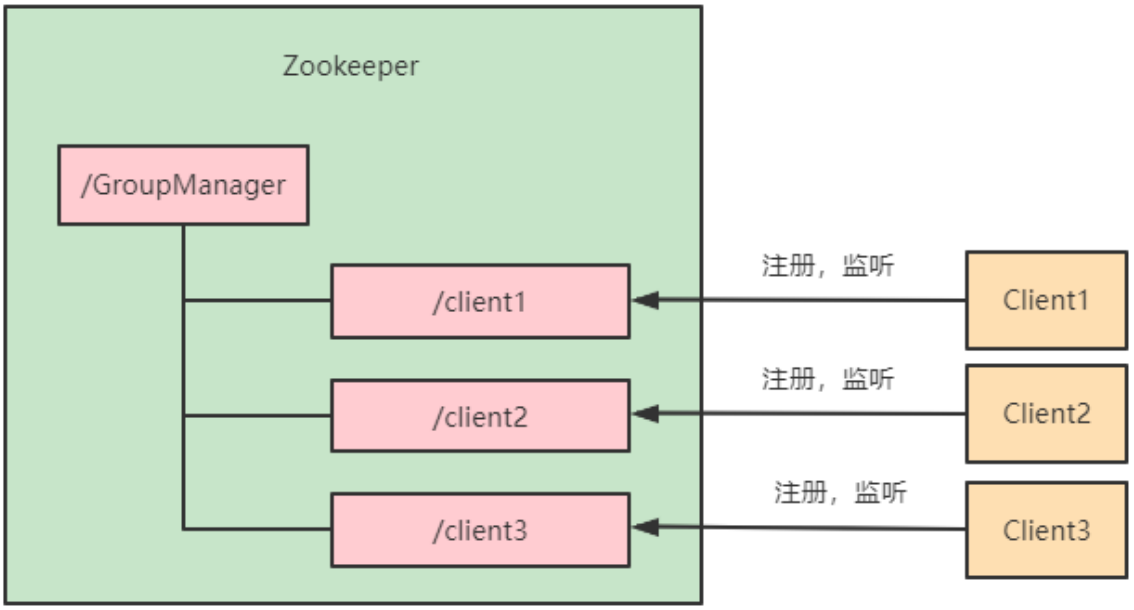


统一集群管理

分布式环境中，实时掌握每个节点的状态是必要的，可根据节点实时状态做出一些调整。

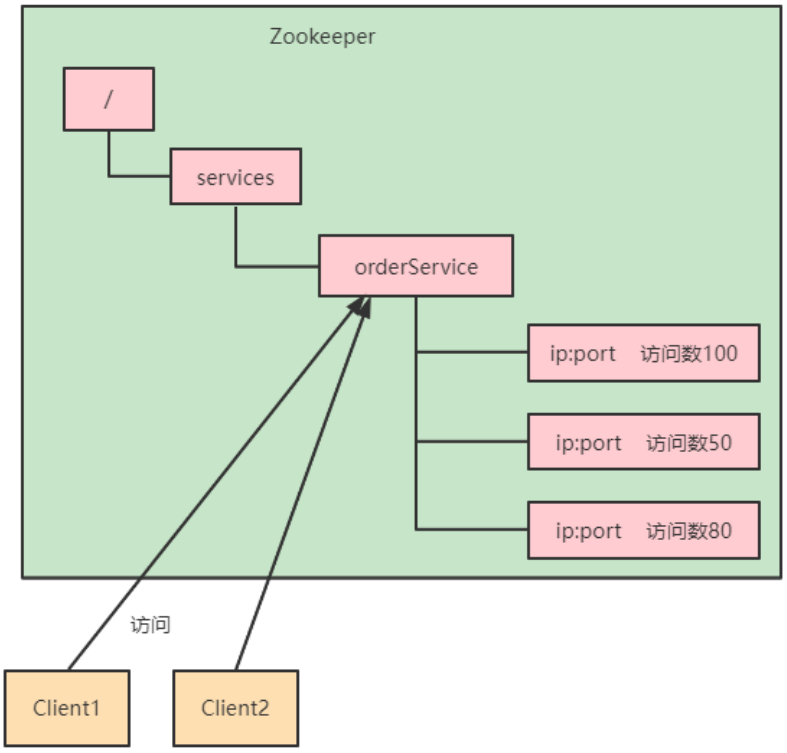
ZooKeeper可以实现实时监控节点状态变化：

- 可将节点信息写入ZooKeeper上的一个ZNode。
- 监听这个ZNode可获取它的实时状态变化。



负载均衡

在Zookeeper中记录每台服务器的访问数，让访问数最少的服务器去处理最新的客户端请求



Zookeeper其他应用场景会在后续课程中详细讲解

补充知识点：

永久性Watch

在被触发之后，仍然保留，可以继续监听ZNode上的变更，是Zookeeper 3.6.0版本新增的功能

```
addWatch [-m mode] path
```

addWatch的作用是针对指定节点添加事件监听，支持两种模式

- PERSISTENT，持久化订阅，针对当前节点的修改和删除事件，以及当前节点的子节点的删除和新增事件。
- PERSISTENT_RECURSIVE，持久化递归订阅，在PERSISTENT的基础上，增加了子节点修改的事件触发，以及子节点的子节点的数据变化都会触发相关事件（满足递归订阅特性）

ACL权限控制

Zookeeper 权限控制 ACL

Zookeeper集群

集群角色

- Leader： 领导者。

事务请求（写操作）的唯一调度者和处理者，保证集群事务处理的顺序性；集群内部各个服务器的调度者。对于create、setData、delete等有写操作的请求，则要统一转发给leader处理，leader需要决定编号、执行操作，这个过程称为事务。

- Follower: 跟随者

处理客户端非事务（读操作）请求（可以直接响应），转发事务请求给Leader；参与集群Leader选举投票。

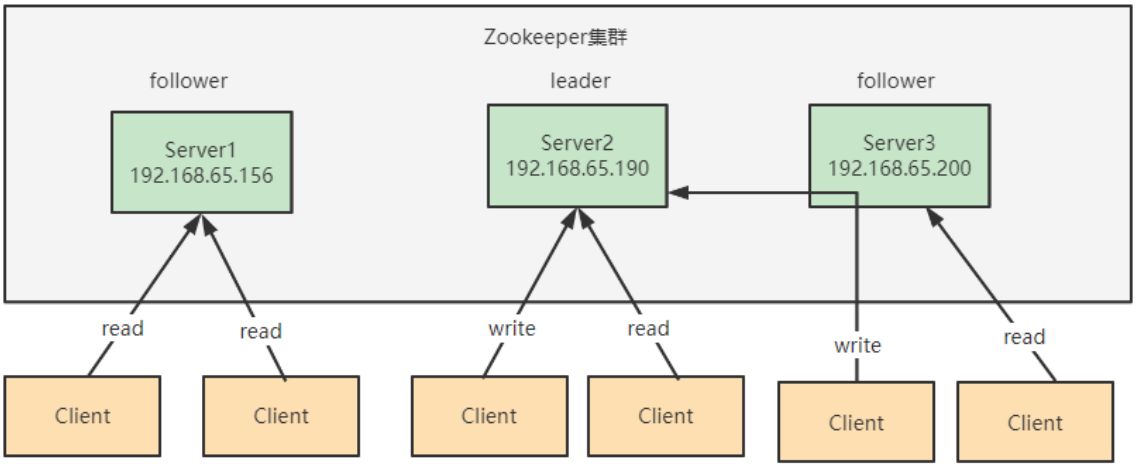
- Observer: 观察者

对于非事务请求可以独立处理（读操作），对于事务性请求会转发给leader处理。Observer节点接收来自leader的inform信息，更新自己的本地存储，不参与提交和选举投票。通常在不影响集群事务处理能力的前提下提升集群的非事务处理能力。

Observer应用场景：

- 提升集群的读性能。因为Observer和不参与提交和选举的投票过程，所以可以通过往集群里面添加observer节点来提高整个集群的读性能。
- 跨数据中心部署。 比如需要部署一个北京和香港两地都可以使用的zookeeper集群服务，并且要求北京和香港客户的读请求延迟都很低。解决方案就是把香港的节点都设置为observer。

集群架构



leader节点可以处理读写请求，follower只可以处理读请求。follower在接到写请求时会把写请求转发给leader来处理。

Zookeeper数据一致性保证：

- 全局可线性化(Linearizable)写入：先到达leader的写请求会被先处理，leader决定写请求的执行顺序。
- 客户端FIFO顺序：来自给定客户端的请求按照发送顺序执行。

三节点Zookeeper集群搭建

环境准备：三台虚拟机

```
192.168.65.156
192.168.65.190
192.168.65.200
```

条件有限也可以在一台虚拟机上搭建zookeeper伪集群

1）修改zoo.cfg配置，添加server节点配置

```
# 修改数据存储目录
dataDir=/data/zookeeper

#三台虚拟机 zoo.cfg 文件末尾添加配置
```

```
server.1=192.168.65.156:2888:3888
server.2=192.168.65.190:2888:3888
server.3=192.168.65.200:2888:3888
```

> server.A=B:C:D
> A 是一个数字，表示这个是第几号服务器； 集群模式下配置一个文件 myid，这个文件在 dataDir 目录下，这个文件里面有一个数据 就是 A 的值，Zookeeper 启动时读取此文件，拿到里面的数据与 zoo.cfg 里面的配置信息比较从而判断到底是哪个server。
> B 是这个服务器的地址；
> C 是这个服务器Follower与集群中的Leader服务器交换信息的端口；
> D 是万一集群中的Leader服务器挂了，需要一个端口来重新进行选举，选出一个新的Leader，而这个端口就是用来执行选举时服务器相互通信的端口。

2）创建 myid 文件，配置服务器编号

在dataDir对应目录下创建 myid 文件，内容为对应ip的zookeeper服务器编号

```
cd /data/zookeeper
# 在文件中添加与 server 对应的编号（注意：上下不要有空行，左右不要有空格）
vim myid
```

注意：添加 myid 文件，一定要在 Linux 里面创建，在 notepad++里面很可能乱码

```
[root@master apache-zookeeper-3.8.0-bin]# cd /data/zookeeper/
[root@master zookeeper]# ls
myid  version-2  zookeeper_server.pid
[root@master zookeeper]# cat myid
1
```

3）启动zookeeper server集群

启动前需要关闭防火墙(生产环境需要打开对应端口)

```
# 分别启动三个节点的zookeeper server
bin/zkServer.sh start
# 查看集群状态
bin/zkServer.sh status
```

```
[root@node01 apache-zookeeper-3.8.0-bin]# bin/zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /usr/local/soft/apache-zookeeper-3.8.0-bin/bin/../conf/zoo.cfg
Client port found: 2181. Client address: localhost. Client SSL: false.
Mode: leader
```

Zookeeper四字命令

用户可以使用Zookeeper四字命令获取 zookeeper 服务的当前状态及相关信息

Zookeeper Leader 选举原理

zookeeper 的 leader 选举存在两个阶段，一个是服务器启动时 leader 选举，另一个是运行过程中 leader 服务器宕机。

在分析选举原理前，先介绍几个重要的参数：

- 服务器 ID(myid)：编号越大在选举算法中权重越大
- 事务 ID(zxid)：值越大说明数据越新，权重越大
- 逻辑时钟(epoch-logicalclock)：同一轮投票过程中的逻辑时钟值是相同的，每投完一次值会增加

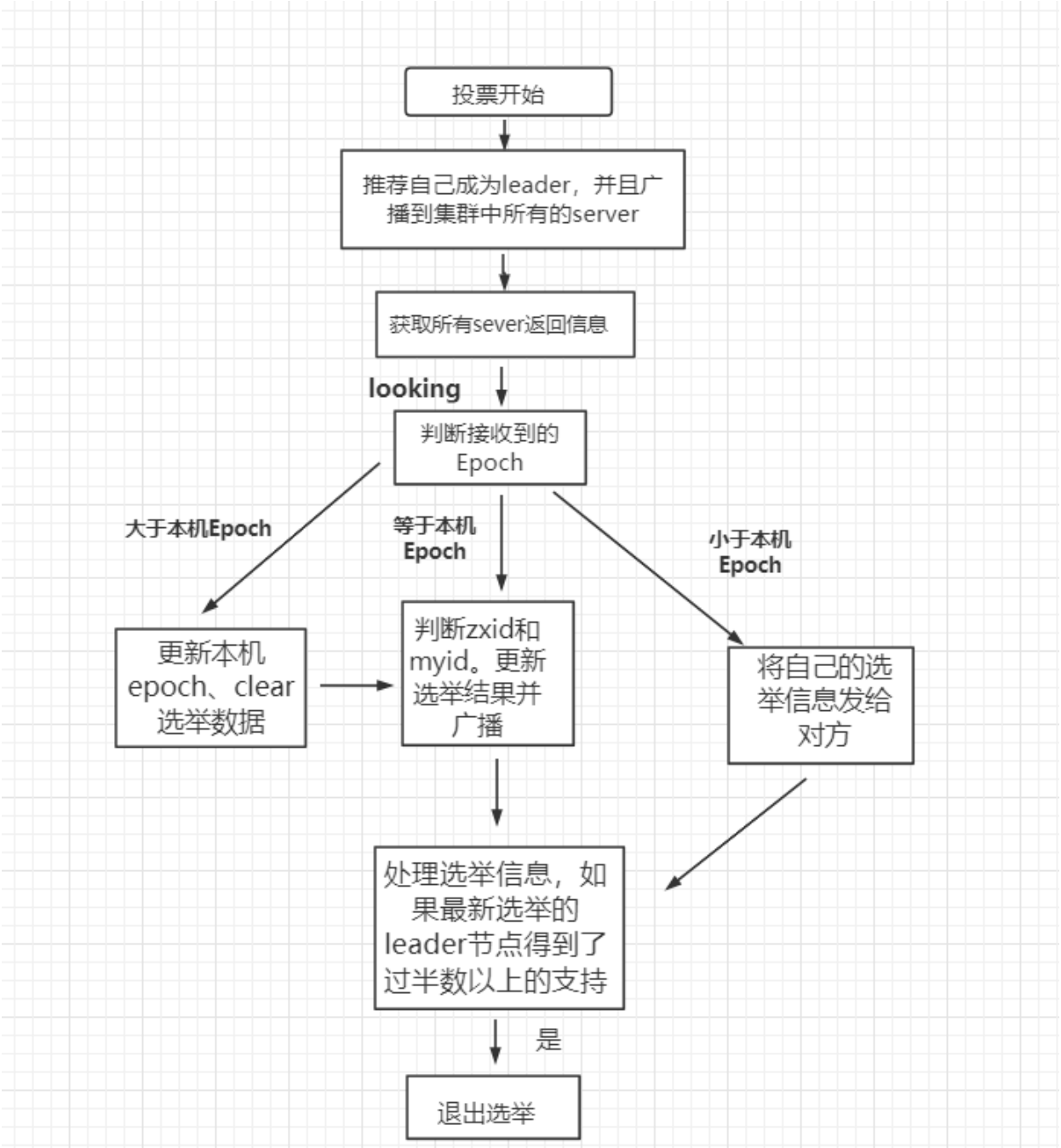
选举状态：

- LOOKING: 竞选状态
- FOLLOWING: 随从状态，同步 leader 状态，参与投票
- OBSERVING: 观察状态，同步 leader 状态，不参与投票
- LEADING: 领导者状态

服务器启动时的 leader 选举

每个节点启动的时候都 LOOKING 观望状态，接下来就开始进行选举主流程。这里选取三台机器组成的集群为例。第一台服务器 server1启动时，无法进行 leader 选举，当第二台服务器 server2 启动时，两台机器可以相互通信，进入 leader 选举过程。

- （1）每台 server 发出一个投票，由于是初始情况，server1 和 server2 都将自己作为 leader 服务器进行投票，每次投票包含所推举的服务器myid、zxid、epoch，使用（myid，zxid）表示，此时 server1 投票为（1,0），server2 投票为（2,0），然后将各自投票发送给集群中其他机器。
- （2）接收来自各个服务器的投票。集群中的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票（epoch）、是否来自 LOOKING 状态的服务器。
- （3）分别处理投票。针对每一次投票，服务器都需要将其他服务器的投票和自己的投票进行对比，对比规则如下：
    - a. 优先比较 epoch
    - b. 检查 zxid，zxid 比较大的服务器优先作为 leader
    - c. 如果 zxid 相同，那么就比较 myid，myid 较大的服务器作为 leader 服务器
- （4）统计投票。每次投票后，服务器统计投票信息，判断是都有过半机器接收到相同的投票信息。server1、server2 都统计出集群中有两台机器接受了（2,0）的投票信息，此时已经选出了 server2 为 leader 节点。
- （5）改变服务器状态。一旦确定了 leader，每个服务器响应更新自己的状态，如果是 follower，那么就变更为 FOLLOWING，如果是 Leader，变更为 LEADING。此时 server3继续启动，直接加入变更自己为 FOLLOWING。



运行过程中的 leader 选举

当集群中 leader 服务器出现宕机或者不可用情况时，整个集群无法对外提供服务，进入新一轮的 leader 选举。

- （1）变更状态。leader 挂后，其他非 Oberver服务器将自身服务器状态变更为 LOOKING。
- （2）每个 server 发出一个投票。在运行期间，每个服务器上 zxid 可能不同。
- （3）处理投票。规则同启动过程。
- （4）统计投票。与启动过程相同。
- （5）改变服务器状态。与启动过程相同。

Zookeeper 数据同步流程

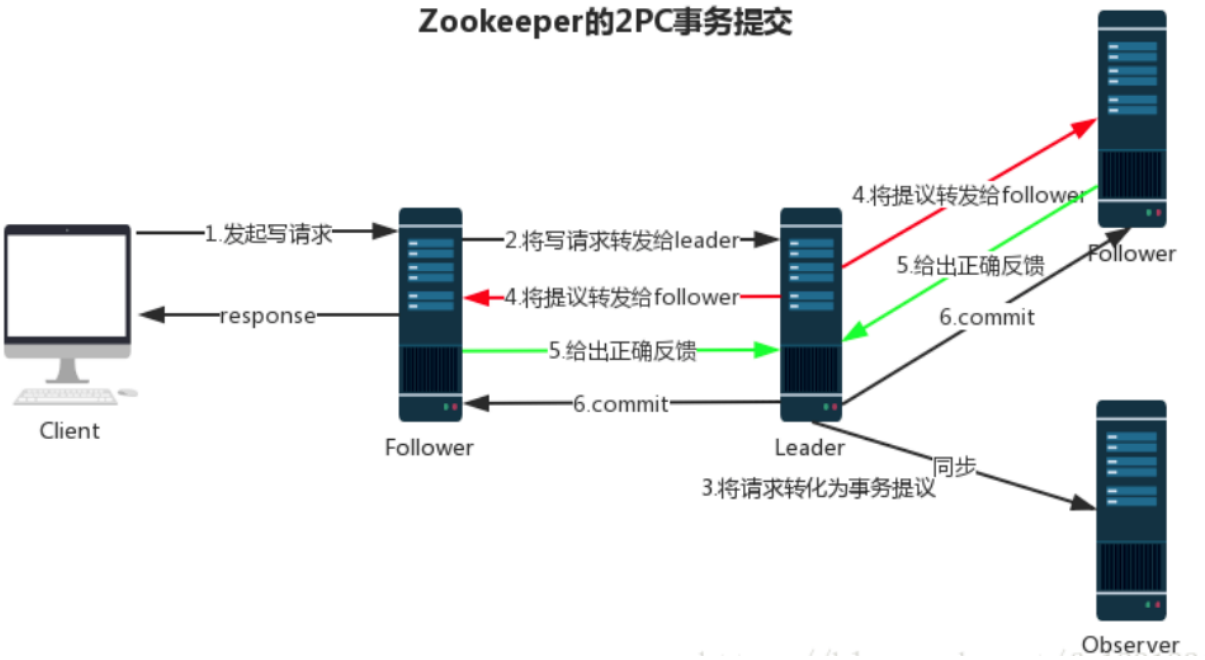
在 Zookeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性。

ZAB 协议分为两部分：

- 消息广播
- 崩溃恢复

消息广播

Zookeeper 使用单一的主进程 Leader 来接收和处理客户端所有事务请求，并采用 ZAB 协议的原子广播协议，将事务请求以 Proposal 提议广播到所有 Follower 节点，当集群中有过半的Follower 服务器进行正确的 ACK 反馈，那么Leader就会再次向所有的 Follower 服务器发送commit 消息，将此次提案进行提交。这个过程可以简称为 2pc 事务提交，整个流程可以参考下图，注意 Observer 节点只负责同步 Leader 数据，不参与 2PC 数据同步过程。



崩溃恢复

在正常情况消息下广播能运行良好，但是一旦 Leader 服务器出现崩溃，或者由于网络原理导致 Leader 服务器失去了与过半 Follower 的通信，那么就会进入崩溃恢复模式，需要选举出一个新的 Leader 服务器。在这个过程中可能会出现两种数据不一致性的隐患，需要 ZAB 协议的特性进行避免。

- Leader 服务器将消息 commit 发出后，立即崩溃
- Leader 服务器刚提出 proposal 后，立即崩溃

ZAB 协议的恢复模式使用了以下策略：

- 选举 zxid 最大的节点作为新的 leader
- 新 leader 将事务日志中尚未提交的消息进行处理