分布式协同管理——Zookeeper

贝毅君

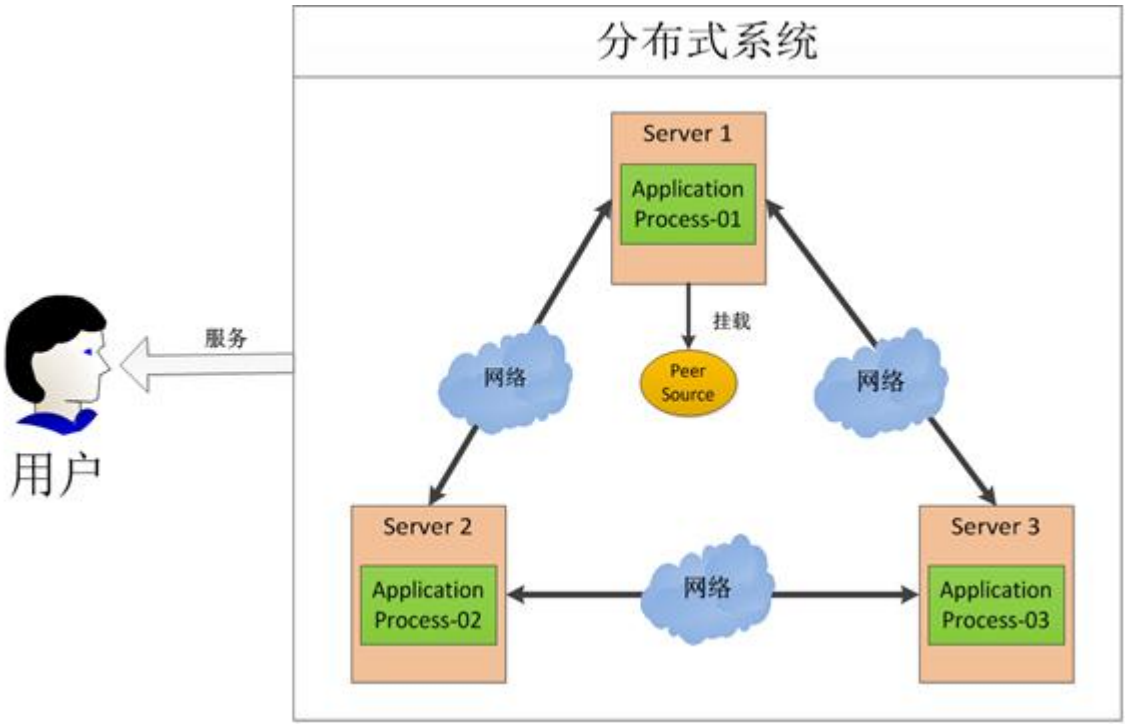
beiyj@zju.edu.cn

分布式协调技术

- 主要用来解决分布式环境当中多个进程之间的同步控制，让它们有序的去访问某种临界资源，防止造成"脏数据"的后果

分布式锁

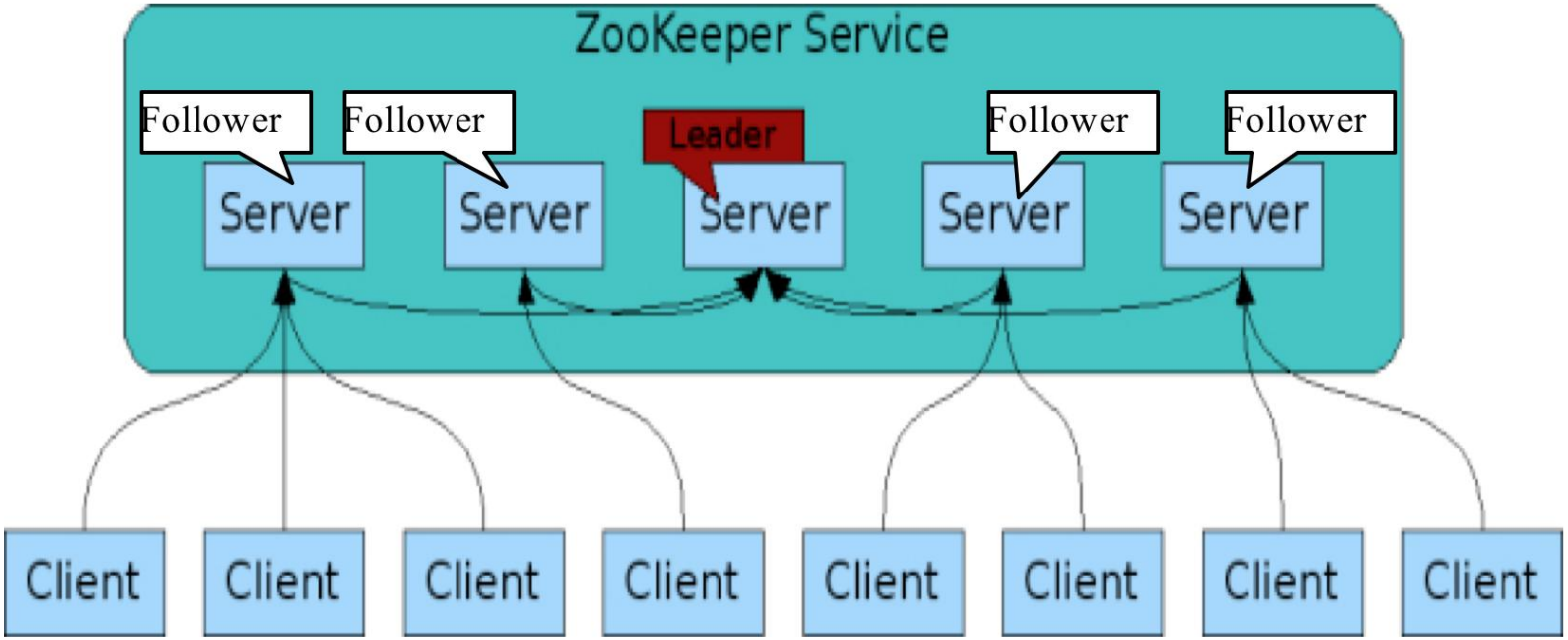
- 保证分布式系统中多个进程能够有序访问某个临界资源
- 分布式环境下的这个锁叫作**分布式锁**



ZooKeeper概述

- Zookeeper 是 Google 的 Chubby一个开源的实现，是 Hadoop 的分布式协调服务
- 它包含一个简单的原语集，分布式应用程序可以基于它实现同步服务，配置维护和命名服务
- 它能提供基于类似于文件系统的目录节点树方式的数据存储，但是 Zookeeper 并不是用来专门存储数据的，它的作用主要是用来维护和监控你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

ZooKeeper概述



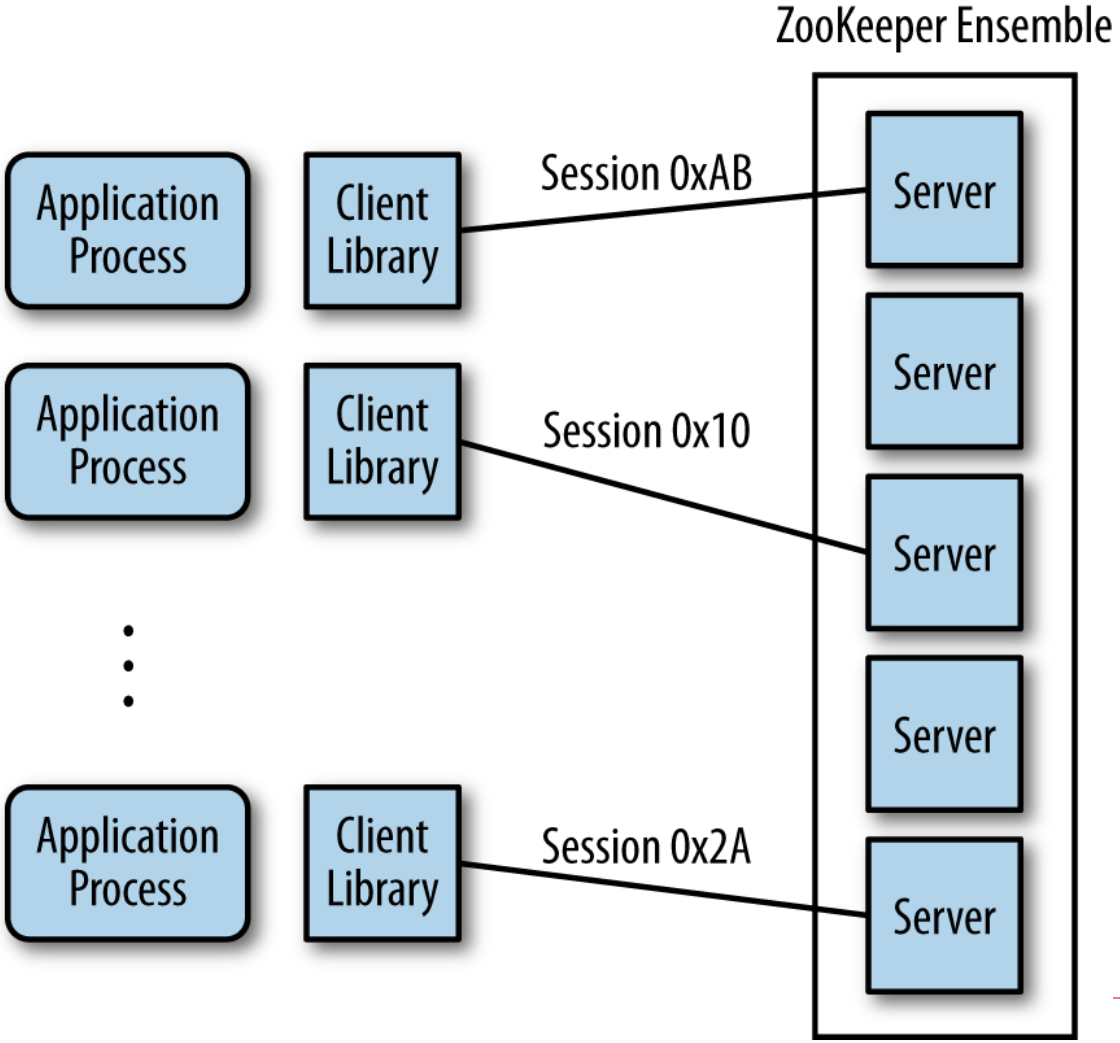
Zookeeper 作为一个分布式的服务框架，主要用来解决分布式集群中应用系统的一致性问题。

Zookeeper的基本概念

- Zookeeper中的角色主要有以下三类，如下表所示：

角色		描述
领导者（Leader）		领导者负责进行投票的发起和决议，更新系统状态
学习者（Learner）	跟随者（Follower）	Follower 用于接收客户请求并向客户端返回结果，在选主过程中参与投票
	观察者（ObServer）	ObServer 可以接收客户端连接，将写请求转发给 leader 节点。但 ObServer 不参加投票过程，只同步 leader 的状态。ObServer 的目的是为了扩展系统，提高读取速度
客户端（Client）		请求发起方

Zookeeper体系架构



ZooKeeper基本流程

- ZooKeeper的基本运转流程：
  1. 选举Leader。
  2. 同步数据。
  3. 选举Leader过程中算法有很多，但要达到的选举标准是一致的。
  4. Leader要具有最高的执行ID，类似root权限。
  5. 集群中大多数的机器得到响应并接受选出的Leader。

ZooKeeper基本应用

- **命名服务** - 按名称标识集群中的节点。
- **配置管理** - 加入节点的最近的和最新的系统配置信息。
- **集群管理** - 实时地监测集群中加入/离开节点。
- **选举算法** - 选举一个节点作为当前leader。
- **锁定和同步服务** - 在修改数据的同时锁定数据。此机制可帮助你在连接其他分布式应用程序（如Apache HBase）时进行自动故障恢复。
- **高度可靠的数据注册表** - 即使在一个或几个节点关闭时也可以获得数据。

ZooKeeper特性

- 结构简单
  - 类似于文件系统的树状结构
- 数据备份
  - 数据一致性，snapshot+WAL
- 有序性
  - 有序的事务编号zxid
- 高效性
  - 所有server都提供读服务

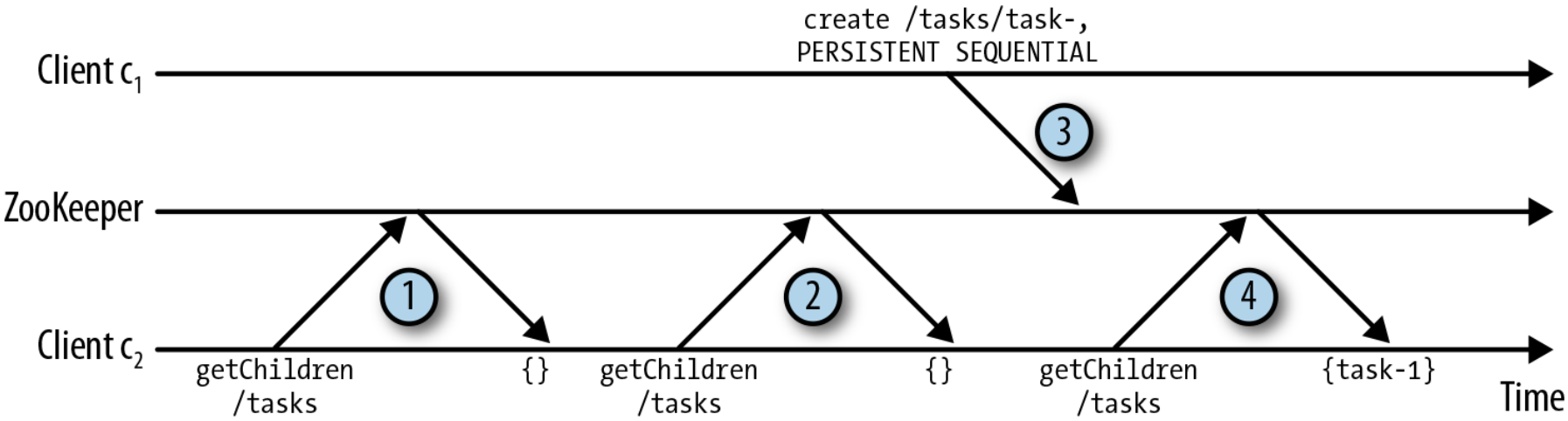
为什么使用Zookeeper？

- 大部分分布式应用需要一个主控、协调器或控制器来管理物理分布的子进程（如资源、任务分配等）
- 目前，大部分应用需要开发私有的协调程序，缺乏一个通用的机制
- 协调程序的反复编写浪费，且难以形成通用、伸缩性好的协调器
- ZooKeeper：提供通用的分布式锁服务，用以协调分布式应用

Zookeeper能帮我们做什么？

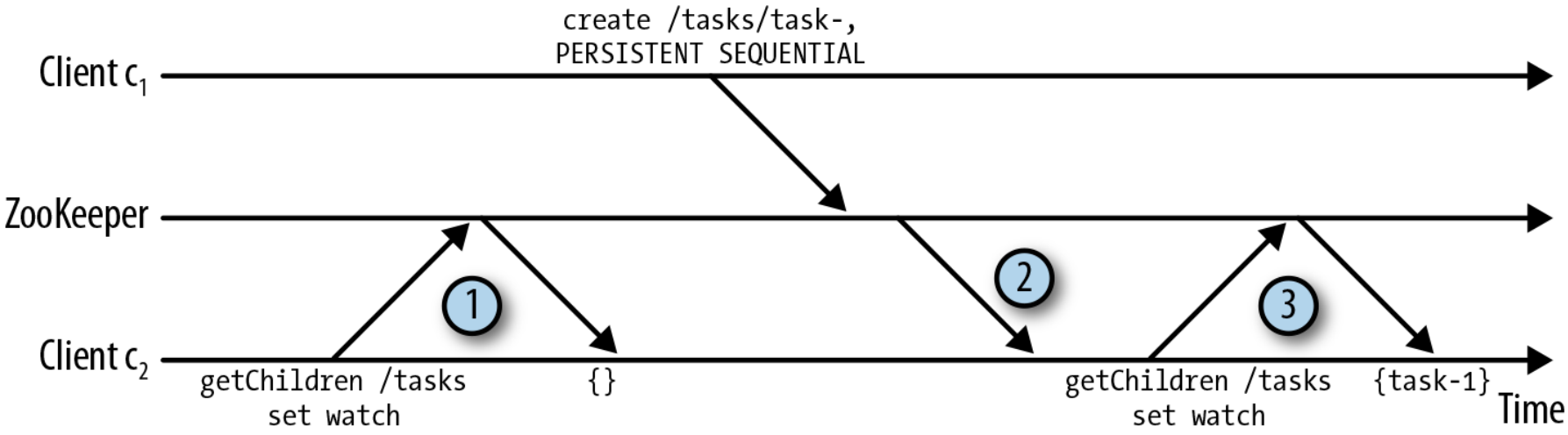
- Hadoop,使用Zookeeper的事件处理确保整个集群只有一个NameNode，存储配置信息等。
- HBase,使用Zookeeper的事件处理确保整个集群只有一个Hmaster，察觉HRegionServer联机和宕机，存储访问控制列表等.

多次重复读-轮询



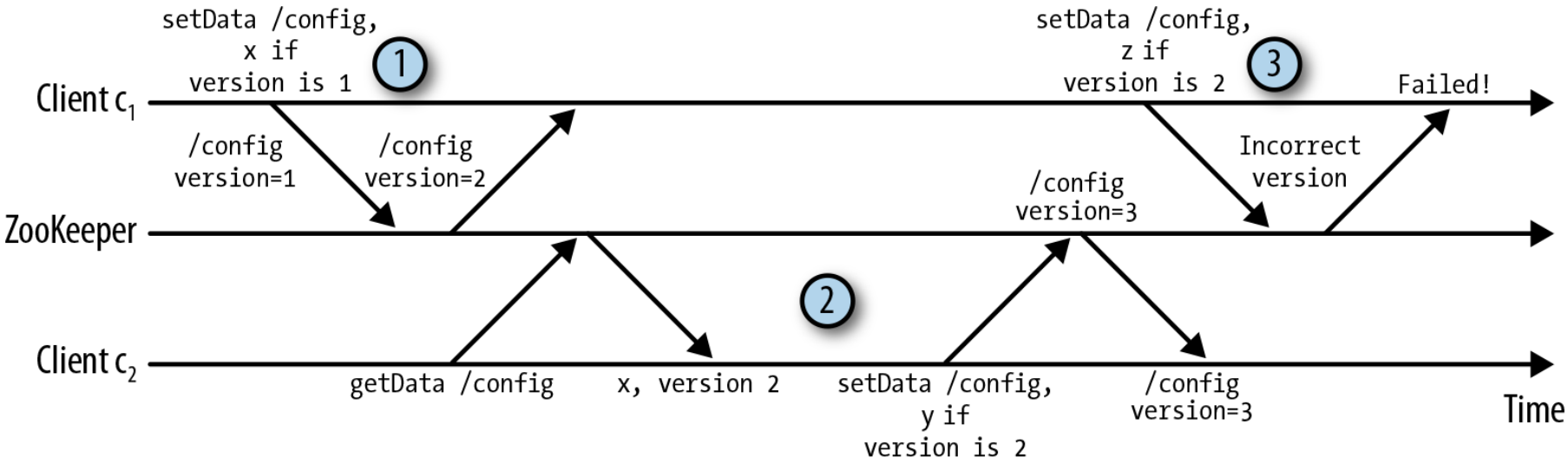
1. Client c_2 reads the list of tasks, initially empty.
2. Client c_2 reads znode again to determine whether there are new tasks.
3. Client c_1 creates a new task.
4. Client c_2 reads again and observes the change.

多次重复读-异步通知



1. Client c_2 reads the list of tasks, initially empty. It sets a watch for changes.
2. When there is a change, the client is notified.
3. Client c_2 reads the children of /tasks and observes the new task.

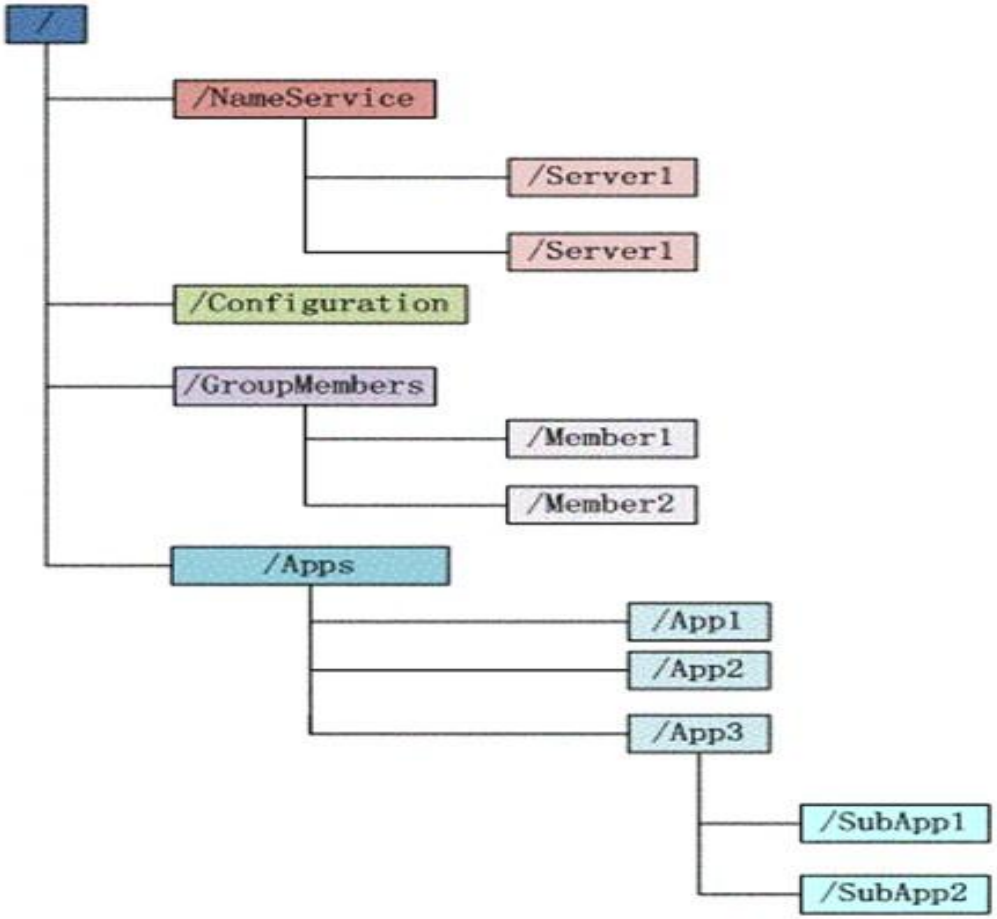
版本保证一致性



1. Client c_1 writes the first version of /config.
2. Client c_2 reads /config and writes the second version.
3. Client c_1 tries to write a change to /config, but the request fails because the version does not match.

ZooKeeper数据模型（1）

- Zookeeper 会维护一个具有层次关系的数据结构，它非常类似于一个标准的文件系统，如下图所示：



ZooKeeper数据模型（2）

- 层次化的目录结构，命名符合常规文件系统规范
- 每个节点在Zookeeper中叫做Znode，并且其有一个唯一的路径标识
- 节点Znode可以包含数据和子节点，但是EPHEMERAL类型的节点不能有子节点
- Znode 的节点名可以自动编号，如 App1 已经存在，再创建的话，将会自动命名为 App2
- Znode中的数据可以有多个版本，比如某一个路径下存有多个数据版本，那么查询这个路径下的数据就需要带上版本
- 客户端应用可以在节点上设置监视器
- 节点不支持部分读写，而是一次性完整读写

ZooKeeper数据节点特性

- ZooKeeper命名空间中的Znode，兼具文件和目录两种特点。既像文件一样维护着数据、元信息、ACL、时间戳等数据结构，又像目录一样可以作为路径标识的一部分。
- 每个Znode由3部分组成:
  - ① stat：此为状态信息, 描述该Znode的版本, 权限等信息
  - ② data：与该Znode关联的数据
  - ③ children：该Znode下的子节点
- 它用来**管理调度数据**，比如分布式应用中的配置文件信息、状态信息、汇集位置等等。这些数据通常以KB为大小单位。ZooKeeper的服务器和客户端都被设计为严格检查并限制每个Znode的数据大小至多1M

Zookeeper数据节点（1）

- Znode有两种类型，短暂的（ephemeral）和持久的（persistent）
- Znode的类型在创建时确定并且之后不能再修改
- 短暂Znode：该节点的生命周期依赖于创建它们的会话的客户端会话结束时，Zookeeper会将该短暂Znode删除，短暂znode不可以有子节点
- 持久Znode：不依赖于客户端会话，只有当客户端明确要删除该持久Znode时才会被删除
- Znode有四种形式的目录节点，PERSISTENT、PERSISTENT_SEQUENTIAL、EPHEMERAL、EPHEMERAL_SEQUENTIAL

Zookeeper数据节点（2）

- 顺序节点
  - 当创建Znode的时候，用户可以请求在ZooKeeper的路径结尾添加一个**递增的计数**。
  - 这个计数**对于此节点的父节点来说**是唯一的，它的格式为"%10d"(10位数字，没有数值的数位用0补充，例如"0000000001")。
  - 顺序节点在锁定和同步中起重要作用。
- 临时节点
  - 支持创建临时节点。临时节点在创建它的会话活动期间存在，会话终止的时候，临时节点被删除。
  - 临时节点在leader选举中起着重要作用。

ZooKeeper数据节点(2)

- 客户端可以在节点上设置watch，称之为**监视器**。
  - 当节点状态发生改变时(Znode的增、删、改)将会触发watch所对应的操作。
  - 当watch被触发时，ZooKeeper将会向客户端发送且仅发送一条通知。
  - 当连接会话过期时，客户端将与服务器断开连接，相关的watches也将被删除。
- 节点数据存取原子性
  - 每个Znode节点里的数据是原子地读取和写入的

ZooKeeper读写数据过程

- 客户端将连接到ZooKeeper集合中的一个节点，可以是leader或follower节点。
  - 服务器节点将向特定客户端分配会话ID，并发送确认。
  - 客户端将向节点发送心跳，以确保连接不会丢失。
- **客户端读取特定的znode数据**
  - 客户端向具有znode路径的节点发送**读取请求。**
  - 节点从其自己的数据库获取返回所请求的znode信息。
- **客户端存储数据在ZooKeeper**
  - 将znode路径和数据发送到服务器。
  - 连接的服务器将该请求转发给leader，leader将向所有的follower发出写入请求。如果大部分节点成功响应，且写入请求成功，则成功返回代码将被发送到客户端。

组件	描述
写入（write）	写入过程由leader节点处理。leader将写入请求转发到所有znode，并等待znode的回复。如果一半的znode回复，则写入过程完成。
读取（read）	读取由特定连接的znode在内部执行，因此不需要与集群进行交互。
复制数据库（replicated database）	它用于在zookeeper中存储数据。每个znode都有自己的数据库，每个znode在一致性的帮助下每次都有相同的数据。
Leader	Leader是负责处理写入请求的Znode。
Follower	follower从客户端接收写入请求，并将它们转发到leader znode。
请求处理器（request processor）	只存在于leader节点。它管理来自follower节点的写入请求。
原子广播（atomic broadcasts）	负责广播从leader节点到follower节点的变化。

Zookeeper的顺序号

- 创建Znode时设置顺序标识，Znode名称后会附加一个值
- 顺序号是一个单调递增的计数器，由父节点维护
- 在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序

Zookeeper的读写机制

- Zookeeper是一个由多个server组成的集群
- 一个leader，多个follower
- 每个server保存一份数据副本
- 全局数据一致
- 分布式读写
- 更新请求转发，由leader实施

Zookeeper的保证

- 更新请求顺序进行，来自同一个client的更新请求按其发送顺序依次执行
- 数据更新原子性，一次数据更新要么成功，要么失败
- 全局唯一数据视图，client无论连接到哪个server，数据视图都是一致的
- 实时性，在一定事件范围内，client能读到最新数据

ZooKeeper中的时间

- 致使ZooKeeper节点状态改变的每一个操作都将使节点接收到一个Zxid格式的时间戳，并且这个时间戳全局有序。如果Zxid1的值小于Zxid2的值，那么Zxid1所对应的事件发生在Zxid2所对应的事件之前。
- 实际 上，ZooKeeper的每个节点维护者三个Zxid值，为别为：cZxid、mZxid、pZxid。
- ① **cZxid**： 是节点的创建时间所对应的Zxid格式时间戳。
- ② **mZxid**：是节点的修改时间所对应的Zxid格式时间戳。

ZooKeeper中的版本

- 对节点的每一个操作都将致使这个节点的版本号增加。每个节点维护着三个版本号，他们分别为：
  - ① **version**：节点数据版本号
  - ② **cversion**：子节点版本号
  - ③ **aversion**：节点所拥有的ACL版本号

ZooKeeper节点属性

属性	描述
czxid	节点被创建的 zxid
mzxid	节点被修改的 zxid
ctime	节点被创建的时间
mtime	节点被修改的 zxid
version	节点被修改的版本号
cversion	节点所拥有的子节点被修改的版本号
aversion	节点的 ACL 被修改的版本号
ephemeralOwner	如果此节点为临时节点，那么他的值为这个节点拥有者的会话 ID；否则，他的值为 0
dataLength	节点数长度
numChildren	节点用的子节点长度
pzxid	最新修改的 zxid，貌似与 mzxid 重合了

ZooKeeper中服务操作

操作	描述
create	创建 Znode（父 Zonde 必须存在）
delete	删除 Znode（Znode 没有子节点）
exists	测试 Znode 是否存在，并获取他的元数据
getACL/ setACL	为 Znode 获取/设置 ACL
getChildren	获取 Znode 所有子节点的列表
getData/setData	获取/设置 Znode 的相关数据
sync	使客户端的 Znode 视图与 ZooKeeper 同步

Zookeeper的主要API接口(1)

- String create(String path, byte[] data, List<ACL> acl, CreateMode createMode)
- Stat exists(String path, boolean watch)
- void delete(String path, int version)
- List<String> getChildren(String path, boolean watch)

Zookeeper的主要API接口(2)

- Stat setData(String path, byte[] data, int version)
- byte[] getData(String path, boolean watch, Stat stat)
- void addAuthInfo(String scheme, byte[] auth)
- Stat setACL(String path, List<ACL> acl, int version)
- List<ACL> getACL(String path, Stat stat)

观察（watcher）

- Watcher 在 ZooKeeper 是一个核心功能，Watcher 可以监控目录节点的数据变化以及子目录的变化，一旦这些状态发生变化，服务器就会通知所有设置在这个目录节点上的 Watcher，从而每个客户端都很快知道它所关注的目录节点的状态发生变化，而做出相应的反应
- 可以设置观察的操作：exists,getChildren,getData
- 可以触发观察的操作：create,delete,setData

写操作与内部事件对应关系

	event For "/path"	event For "/path/child"
create("/path")	EventType.NodeCreated	NA
delete("/path")	EventType.NodeDeleted	NA
setData("/path")	EventType.NodeDataChanged	NA
create("/path/child")	EventType.NodeChildrenChanged	EventType.NodeCreated
delete("/path/child")	EventType.NodeChildrenChanged	EventType.NodeDeleted
setData("/path/child")	NA	EventType.NodeDataChanged

内部事件与watcher对应关系

event For "/path"	defaultWatcher	exists ("/path")	getData ("/path")	getChildren ("/path")
EventType.None	√	√	√	√
EventType.NodeCreated		√	√	
EventType.NodeDeleted		√(不正常)	√	
EventType.NodeDataChanged		√	√	
EventType.NodeChildrenChanged				√

写操作与watcher对应关系

	"/path"			"/path/child"		
	exists	getData	getChildren	exists	getData	getChild
create("/path")	√	√				
delete("/path")	√	√	√			
setData("/path")	√	√				
create("/path/child")			√	√	√	
delete("/path/child")			√	√	√	√
setData("/path /child")				√	√	

ACL(1)

每个znode被创建时都会带有一个ACL列表，用于决定谁可以对它执行何种操作

ACL权限	允许的操作
CREATE	create（子节点）
READ	getChildren getData
WRITE	setData
DELETE	delete（子节点）
ADMIN	setACL

ACL(2)

- 身份验证模式有三种：
- digest:用户名，密码
- host:通过客户端的主机名来识别客户端
- ip： 通过客户端的ip来识别客户端
- new ACL(Perms.READ,new Id("host","example.com"));
  - 这个ACL对应的身份验证模式是host，符合该模式的身份是example.com，权限的组合是：READ

JAVA API

- **connect** - 连接到ZooKeeper
- **create**- 创建znode
- **exists**- 检查znode是否存在及其信息
- **getData** - 从特定的znode获取数据
- **setData** - 在特定的znode中设置数据
- **getChildren** - 获取特定znode中的所有子节点
- **delete** - 删除特定的znode及其所有子项
- **close** - 关闭连接

Zookeeper示例代码(1)


```
// 创建一个与服务器的连接
ZooKeeper zk = new ZooKeeper("localhost:" + CLIENT_PORT,
        ClientBase.CONNECTION_TIMEOUT, new Watcher() {
            // 监控所有被触发的事件
            public void process(WatchedEvent event) {
                System.out.println("已经触发了" + event.getType() + "事件! ");
            }
        });
// 创建一个目录节点
zk.create("/testRootPath", "testRootData".getBytes(), Ids.OPEN_ACL_UNSAFE,
  CreateMode.PERSISTENT);
// 创建一个子目录节点
zk.create("/testRootPath/testChildPathOne", "testChildDataOne".getBytes(),
  Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
System.out.println(new String(zk.getData("/testRootPath",false,null)));
```

Zookeeper示例代码(2)

```
// 取出子目录节点列表
System.out.println(zk.getChildren("/testRootPath",true));
// 修改子目录节点数据
zk.setData("/testRootPath/testChildPathOne","modifyChildDataOne".getBytes(),-1);
System.out.println("目录节点状态: ["+zk.exists("/testRootPath",true)+"]");
// 创建另外一个子目录节点
zk.create("/testRootPath/testChildPathTwo", "testChildDataTwo".getBytes(),
  Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
System.out.println(new String(zk.getData("/testRootPath/testChildPathTwo",true,null)));
// 删除子目录节点
zk.delete("/testRootPath/testChildPathTwo",-1);
zk.delete("/testRootPath/testChildPathOne",-1);
// 删除父目录节点
zk.delete("/testRootPath",-1);
// 关闭连接
zk.close();
```

- 输出的结果如下:
- 已经触发了 None 事件!
- testRootData [testChildPathOne]
- 目录节点状态:
  [5,5,1281804532336,1281804532336,0,1,0,0,12,1,6]
- 已经触发了 NodeChildrenChanged 事件！
- testChildDataTwo
- 已经触发了 NodeDeleted 事件！
- 已经触发了 NodeDeleted 事件！

Zookeeper示例

- 安装
- 命令行
- API应用

谢谢！