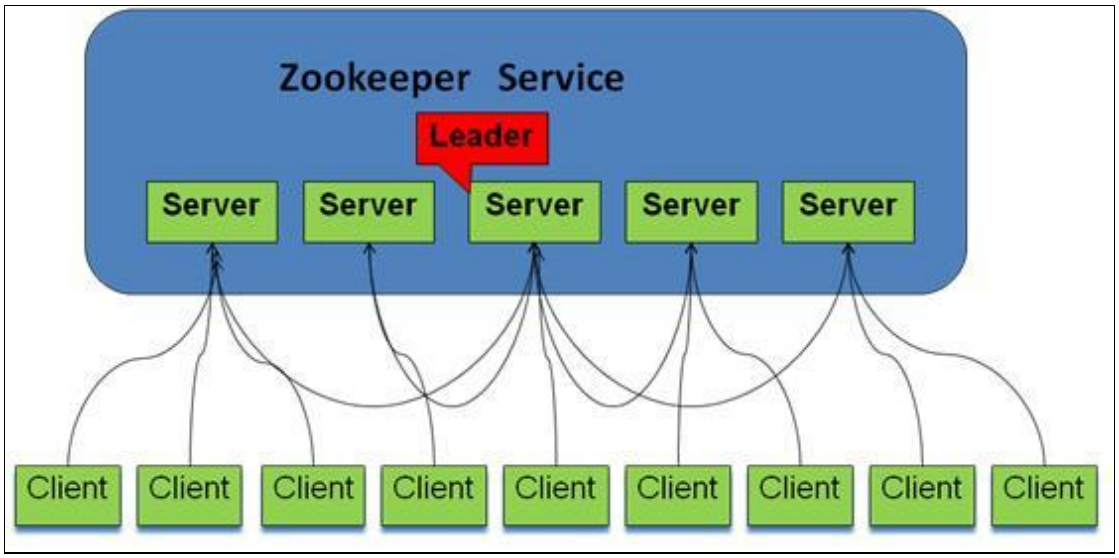
Zookeeper 原理和应用

目录

1、Zookeeper 原理解析

1.1、集群角色描述

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



1.2、Paxos 算法概述（ZAB 协议）

Paxos 算法是莱斯利•兰伯特（英语：Leslie Lamport）于 1990 年提出的一种基于消息传递且具有高度容错特性的一致性算法。

分布式系统中的节点通信存在两种模型：**共享内存（Shared memory）**和**消息传递（Messages passing）**。基于消息传递通信模型的分布式系统，不可避免的会发生以下错误：进程可能会慢、被杀死或者重启，消息可能会延迟、丢失、重复，在基础 Paxos 场景中，先不考虑可能出现消息篡改即拜占庭错误（**Byzantine failure，即虽然有可能一个消息被传递了两次，但是绝对不会出现错误的消息**）的情况。**Paxos 算法解决的问题是在一个可能发生上述异常的分布式系统中如何就某个值达成一致，保证不论发生以上任何异常，都不会破坏决议一致性**。

Paxos 算法使用一个希腊故事来描述，在 Paxos 中，存在三种角色，分别为
Proposer（提议者，用来发出提案 proposal），
Acceptor（接受者，可以接受或拒绝提案），
Learner（学习者，学习被选定的提案，当提案被超过半数的 Acceptor 接受后为被批准）。
下面更精确的定义 Paxos 要解决的问题：
1、决议(value)只有在被 proposer 提出后才能被批准
2、在一次 Paxos 算法的执行实例中，只批准(chose)一个 value
3、learner 只能获得被批准(chosen)的 value

ZooKeeper 的选举算法有两种：一种是基于 **Basic Paxos**（Google Chubby 采用）实现的，另外一种是基于 **Fast Paxos**（ZooKeeper 采用）算法实现的。系统默认的选举算法为 Fast Paxos。并且 ZooKeeper 在 3.4.0 版本后只保留了 FastLeaderElection 算法。

ZooKeeper 的核心是原子广播，这个机制保证了各个 Server 之间的同步。实现这个机制的协议叫做 **ZAB 协议**（**Zookeeper Atomic BrodCast**）。

ZAB 协议有两种模式，它们分别是**崩溃恢复模式（选主）**和**原子广播模式（同步）**。

1、当服务启动或者在领导者崩溃后，ZAB 就进入了恢复模式，当领导者被选举出来，且大多数 Server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 follower 之间具有相同的系统状态。
2、当 ZooKeeper 集群选举出 leader 同步完状态退出恢复模式之后，便进入了原子广播模式。所有的写请求都被转发给 leader，再由 leader 将更新 proposal 广播给 follower

为了保证事务的顺序一致性，zookeeper 采用了递增的事务 id 号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了 zxid。实现中 zxid 是一个 64 位的数字，它高 32 位是 epoch 用来标识 leader 关系是否改变，每次一个 leader 被选出来，它都会有一个新的 epoch，标识当前属于那个 leader 的统治时期。低 32 位用于递增计数。

这里给大家介绍以下 Basic Paxos 流程：

1、选举线程由当前 Server 发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的 Server
2、选举线程首先向所有 Server 发起一次询问(包括自己)
3、选举线程收到回复后，验证是否是自己发起的询问(验证 zxid 是否一致)，然后获取对方的 serverid(myid)，并存储到当前询问对象列表中，最后获取对方提议的 leader 相关信息(serverid,zxid)，并将这些信息存储到当次选举的投票记录表中
4、收到所有 Server 回复以后，就计算出 id 最大的那个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server
5、线程将当前 id 最大的 Server 设置为当前 Server 要推荐的 Leader，如果此时获胜的 Server 获得 n/2 + 1 的 Server 票数， 设置当前推荐的 leader 为获胜的 Server，将根据获胜的 Server 相关信息设置自己的状态，否则，继续这个过程，直到 leader 被选举出来。

通过流程分析我们可以得出：要使 Leader 获得多数 Server 的支持，则 Server 总数必须是奇数 2n+1，且存活的 Server 的数目不得少于 n+1。

每个 Server 启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的 server 还会从磁盘快照中恢复数据和会话信息，zk 会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。

Fast Paxos 流程是在选举过程中，某 Server 首先向所有 Server 提议自己要成为 leader，当其它 Server 收到提议以后，解决 epoch 和 zxid 的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出 Leader

1.2.1、ZooKeeper 的全新集群选主

以一个简单的例子来说明整个选举的过程：假设有五台服务器组成的 zookeeper 集群，它们的 serverid 从 1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来看看会发生什么
1、服务器 1 启动，此时只有它一台服务器启动了，它发出去的报没有任何响应，所以它的选举状态一直是 LOOKING 状态
2、服务器 2 启动，它与最开始启动的服务器 1 进行通信，互相交换自己的选举结果，由于两者都没有历史数据，所以 id 值较大的服务器 2 胜出，但是由于没有达到超过半数以上的

服务器都同意选举它(这个例子中的半数以上是 3)，所以服务器 1、2 还是继续保持 LOOKING 状态
3、服务器 3 启动，根据前面的理论分析，服务器 3 成为服务器 1,2,3 中的老大，而与上面不同的是，此时有三台服务器(超过半数)选举了它，所以它成为了这次选举的 leader
4、服务器 4 启动，根据前面的分析，理论上服务器 4 应该是服务器 1,2,3,4 中最大的，但是由于前面已经有半数以上的服务器选举了服务器 3，所以它只能接收当小弟的命了
5、服务器 5 启动，同 4 一样，当小弟

总结：zookeeper server 的三种工作状态
LOOKING：当前 Server 不知道 leader 是谁，正在搜寻，正在选举
LEADING：当前 Server 即为选举出来的 leader，负责协调事务
FOLLOWING：leader 已经选举出来，当前 Server 与之同步，服从 leader 的命令

1.2.2、ZooKeeper 的非全新集群选主

那么，初始化的时候，是按照上述的说明进行选举的，但是当 zookeeper 运行了一段时间之后，有机器 down 掉，重新选举时，选举过程就相对复杂了。

需要加入数据 version、serverid 和逻辑时钟。
数据 version：数据新的 version 就大，数据每次更新都会更新 version
server id：就是我们配置的 myid 中的值，每个机器一个
逻辑时钟：这个值从 0 开始递增，每次选举对应一个值，也就是说：如果在同一次选举中，那么这个值应该是一致的；逻辑时钟值越大，说明这一次选举 leader 的进程更新，也就是每次选举拥有一个 zxid，投票结果只取 zxid 最新的
选举的标准就变成：

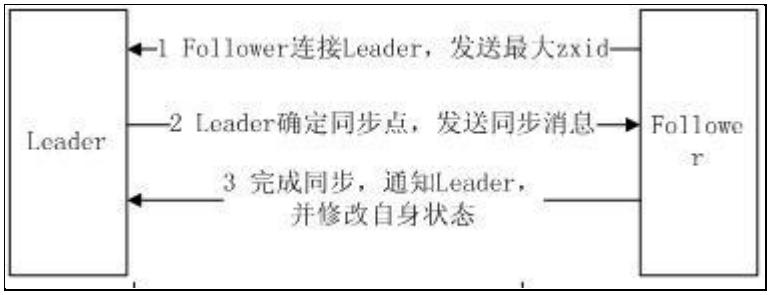
1、逻辑时钟小的选举结果被忽略，重新投票
2、统一逻辑时钟后，数据 version 大的胜出
3、数据 version 相同的情况下，server id 大的胜出

根据这个规则选出 leader。

1.3、数据同步

选完 leader 以后，zk 就进入状态同步过程。
1、leader 等待 server 连接；
2、follower 连接 leader，将最大的 zxid 发送给 leader；
3、leader 根据 follower 的 zxid 确定同步点；
4、完成同步后通知 follower 已经成为 uptodate 状态；
5、follower 收到 uptodate 消息后，又可以重新接受 client 的请求进行服务了。

以下是流程图：



1.4、ZooKeeper 工作流程

1.4.1、Leader 工作流程

Leader 主要有三个功能：

1、恢复数据

2、维持与 Learner 的心跳，接收 Learner 请求并判断 Learner 的请求消息类型

Learner 的消息类型主要：

PING 消息：Learner 的心跳信息

REQUEST 消息：Follower 发送的提议信息，包括读写请求

ACK 消息：Follower 对提议的回复，超过半数的 Follower 通过，则 commit 该提议

REVALIDATE 消息：用来延长 SESSION 有效时间

3、根据不同的消息类型，进行不同的处理。

1.4.2、Follower 工作流程

Follower 主要有四个功能：

1、向 Leader 发送请求（PING 消息、REQUEST 消息、ACK 消息、REVALIDATE 消息）；

2、接收 Leader 消息并进行处理；

3、接收 Client 的请求，如果为写请求，则转发给 Leader；

4、返回 Client 结果。

Follower 的消息循环处理如下几种来自 Leader 的消息：

1、PING 消息： 心跳消息；

2、PROPOSAL 消息：Leader 发起的提案，要求 Follower 投票；

3、COMMIT 消息：服务器端最新一次提案的信息；

4、UPTODATE 消息：表明同步完成；

5、REVALIDATE 消息：根据 Leader 的 REVALIDATE 结果，关闭待 revalidate 的 session 还是允许其接受消息；

6、SYNC 消息：返回 SYNC 结果到客户端，这个消息最初由客户端发起，用来强制得到最新的更新。

1.4.3、Observer 工作流程

Observer 流程和 Follower 的唯一不同的地方就是 Observer 不会参加 Leader 发起的投票，也不会被选举为 Leader，所以不重复描述了。

2、Zookeeper 应用案例

2.1、服务器上下线动态感知

1、需求描述

某分布式系统中，主节点可以有多台，可以动态上下线。任意一台客户端都能实时感知到主节点服务器的上下线

2、设计思路

1、设计服务器端存入服务器上线，下线的信息，比如都写入到 servers 节点下

2、设计客户端监听该 servers 节点，获取该服务器集群的在线服务器列表

3、服务器一上线，就往 zookeeper 文件系统中的一个统一的节点比如 servers 下写入一个临时节点，记录下服务器的信息（思考，该节点最好采用什么类型的节点？）

4、服务器一下线，则删除 servers 节点下的该服务器的信息，则客户端因为监听了该节点的数据变化，所以将第一时间得知服务器的在线状态

3、代码开发

服务器端处理：

```
package com.ghgj.zookeeper.mydemo;

import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooDefs.Ids;
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.data.Stat;

/**
 * 用来模拟服务器的动态上线下线
 * 总体思路就是服务器上线就上 zookeeper 集群创建一个临时节点，然后监听了该数据节点的个数变化的客户端都收到通知
 * 下线，则该临时节点自动删除，监听了该数据节点的个数变化的客户端也都收到通知
 */
```

```
public class DistributeServer {

	private static final String connectStr = "hadoop02:2181,hadoop03:2181,hadoop04:2181";
	private static final int sessionTimeout = 4000;
	private static final String PARENT_NODE = "/server";
	static ZooKeeper zk = null;

	public static void main(String[] args) throws Exception {
		DistributeServer distributeServer = new DistributeServer();
		distributeServer.getZookeeperConnect();
		distributeServer.registeServer("hadoop03");
		Thread.sleep(Long.MAX_VALUE);
	}

	/**
	 * 拿到 zookeeper 进群的链接
	 */
	public void getZookeeperConnect() throws Exception {
		zk = new ZooKeeper(connectStr, sessionTimeout, new Watcher() {
			@Override
			public void process(WatchedEvent event) {

			}
		});
	}

	/**
	 * 服务器上线就注册，掉线就自动删除，所以创建的是临时顺序节点
	 */
	public void registeServer(String hostname) throws Exception{
		Stat exists = zk.exists(PARENT_NODE, false);
		if(exists == null){
		zk.create(PARENT_NODE,"server_parent_node".getBytes(),Ids.OPEN_ACL_UNSAFE,
CreateMode.PERSISTENT);
		}
		zk.create(PARENT_NODE+"/"+hostname, hostname.getBytes(), Ids.OPEN_ACL_UNSAFE,
CreateMode.EPHEMERAL_SEQUENTIAL);
		System.out.println(hostname+" is online, start working.....");
	}
}
```

客户端处理：

```
package com.ghgj.zookeeper.mydemo;
```

```
import java.util.ArrayList;
import java.util.List;

import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;

/**
 * 用来模拟用户端的操作：连上 zookeeper 进群，实时获取服务器动态上下线的节点信息
 * 总体思路就是每次该 server 节点下有增加或者减少节点数，我就打印出来该 server 节点下的所有节点
 */
public class DistributeClient {

	private static final String connectStr = "hadoop02:2181,hadoop03:2181,hadoop04:2181";
	private static final int sessionTimeout = 4000;
	private static final String PARENT_NODE = "/server";
	static ZooKeeper zk = null;

	public static void main(String[] args) throws Exception {
		DistributeClient dc = new DistributeClient();
		dc.getZookeeperConnect();
		Thread.sleep(Long.MAX_VALUE);
	}

	/**
	 * 拿到 zookeeper 进群的链接
	 */
	public void getZookeeperConnect() throws Exception {
		zk = new ZooKeeper(connectStr, sessionTimeout, new Watcher() {
			@Override
			public void process(WatchedEvent event) {
				try {
					// 获取父节点 server 节点下所有子节点，即是所有正上线服务的服务器节点
					List<String> children = zk.getChildren(PARENT_NODE, true);
					List<String> servers = new ArrayList<String>();
					for(String child: children){
						// 取出每个节点的数据，放入到 list 里
						String server = new String(zk.getData(PARENT_NODE+"/"+child, false, null), "UTF-8");
						servers.add(server);
					}
					// 打印 list 里面的元素
```

```
                    System.out.println(servers);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
        System.out.println("Client is online, start Working.....");
    }
}
```

2.2、分布式共享锁

1、需求描述

在我们自己的分布式业务系统中，可能会存在某种资源，需要被整个系统的各台服务器共享访问，但是只允许一台服务器同时访问

2、设计思路

1、设计多个客户端同时访问同一个数据

2、为了同一时间只能允许一个客户端上去访问，所以各个客户端去 zookeeper 集群的一个 znode 节点去注册一个临时节点，定下规则，每次都是编号最小的客户端才能去访问

3、多个客户端同时监听该节点，每次当有子节点被删除时，就都收到通知，然后判断自己的编号是不是最小的，最小的就去执行访问，不是最小的就继续监听。

3、代码开发

服务器端：

```
package com.ghgj.zookeeper.mydemo;

import java.util.Collections;
import java.util.List;
import java.util.Random;

import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.EventType;
import org.apache.zookeeper.ZooDefs.Ids;
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.data.Stat;

/**
 * 需求：多个客户端，需要同时访问同一个资源，但同时只允许一个客户端进行访问。
```

```
 * 设计思路：多个客户端都去父 znode 下写入一个子 znode，能写入成功的去执行访问，写入不成功的等待
 */
public class MyDistributeLock {

    private static final String connectStr = "hadoop02:2181,hadoop03:2181,hadoop04:2181";
    private static final int sessionTimeout = 4000;
    private static final String PARENT_NODE = "/parent_locks";
    private static final String SUB_NODE = "/sub_client";
    static ZooKeeper zk = null;

    private static String currentPath = "";

    public static void main(String[] args) throws Exception {

        MyDistributeLock mdc = new MyDistributeLock();

        // 1、拿到 zookeeper 链接
        mdc.getZookeeperConnect();

        // 2、查看父节点是否存在，不存在则创建
        Stat exists = zk.exists(PARENT_NODE, false);
        if(exists == null){
            zk.create(PARENT_NODE, PARENT_NODE.getBytes(), Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
        }

        // 3、监听父节点
        zk.getChildren(PARENT_NODE, true);

        // 4、往父节点下注册节点，注册临时节点，好处就是，当宕机或者断开链接时该节点自动删除
        currentPath = zk.create(PARENT_NODE+SUB_NODE, SUB_NODE.getBytes(), Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);

        // 5、关闭 zk 链接
        Thread.sleep(Long.MAX_VALUE);
        zk.close();
    }

    /**
     * 拿到 zookeeper 集群的链接
     */
    public void getZookeeperConnect() throws Exception {
```

```
            zk = new ZooKeeper(connectStr, sessionTimeout, new Watcher() {
                @Override
                public void process(WatchedEvent event) {
                    // 匹配看是不是子节点变化，并且监听的路径也要对
                    if(event.getType() == EventType.NodeChildrenChanged &&
event.getPath().equals(PARENT_NODE)){
                        try {
                            // 获取父节点的所有子节点，并继续监听
                            List<String> childrenNodes = zk.getChildren(PARENT_NODE, true);
                            // 匹配当前创建的 znode 是不是最小的 znode
                            Collections.sort(childrenNodes);
                            if((PARENT_NODE+"/"+childrenNodes.get(0)).equals(currentPath)){
                                // 处理业务
                                handleBusiness(currentPath);
                            }
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                }
            });
        }

        public void handleBusiness(String create) throws Exception{
            System.out.println(create+" is working.....");
            Thread.sleep(new Random().nextInt(4000));
            zk.delete(currentPath, -1);
            System.out.println(create+" is done .....");
            currentPath = zk.create(PARENT_NODE+SUB_NODE, SUB_NODE.getBytes(),
Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);
        }
}
```

2.3、Hadoop HA 高可用集群搭建

见搭建文档