ZooKeeper原理与实战

目录

- Zookeeper简介
- ZooKeeper安装
- Curator常用API及其原理
- Zookeeper在PtBalancer中的应用

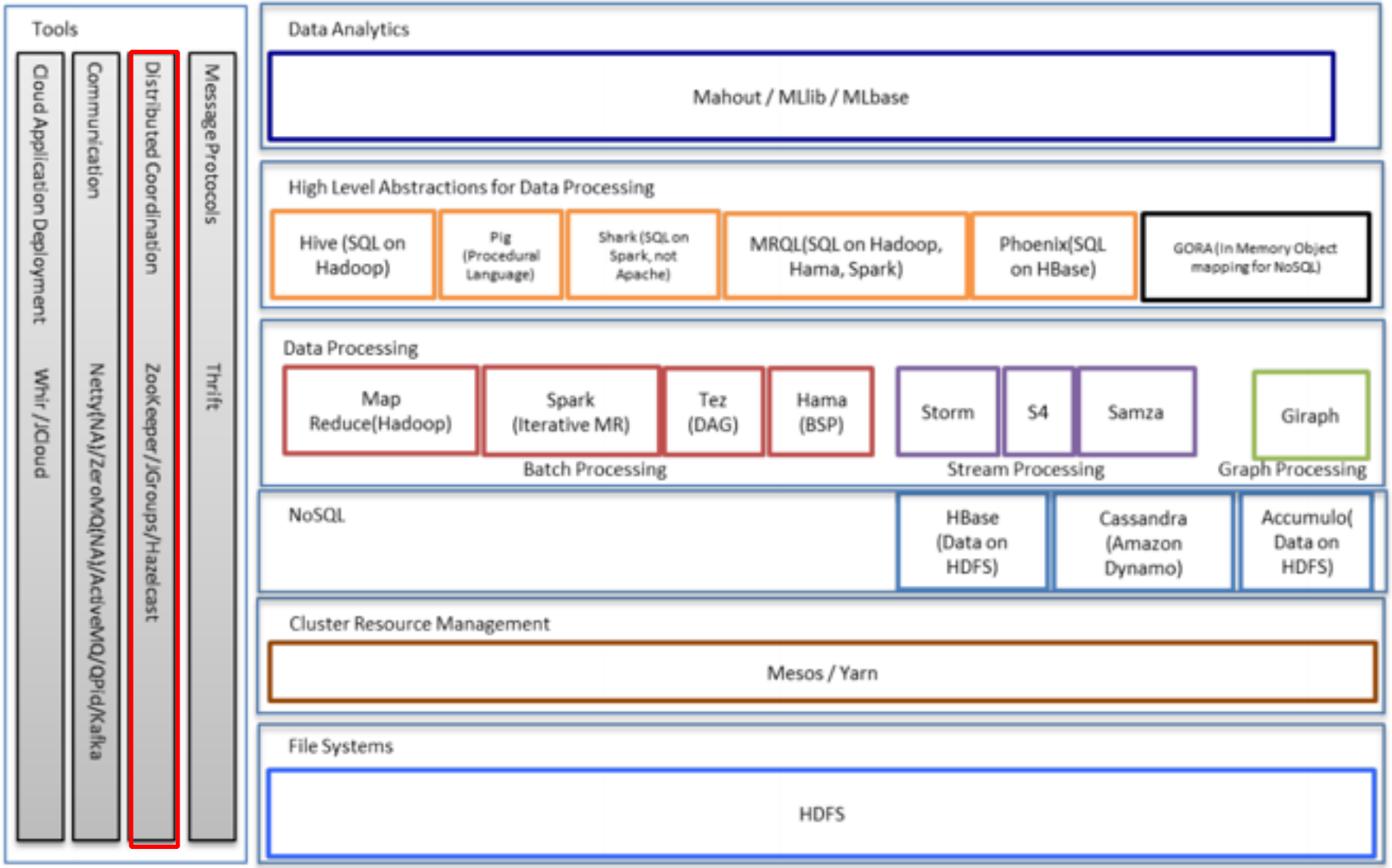
Zookeeper简介

- 背景
- 典型应用场景
- 架构

背景

- 开源的大数据系统，类似于动物园
  - 难以管理


- ZooKeeper
  - 动物管理员
  - 分布式系统协调
  - http://zookeeper.apache.org/



Batch Processing frameworks

Stream Processing frameworks

NoSQL databases and related technology

Graph Processing frameworks

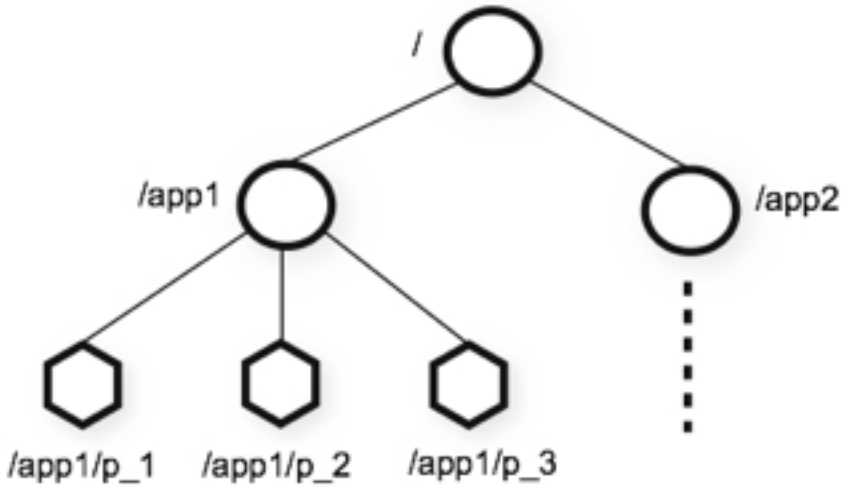
NA Non Apache projects

ZooKeeper在Hadoop生态系统中的位置

典型应用场景

- 分布式通知/协调
  用于分布式系统的任务分发与任务执行结果反馈
- 集群管理
  - 可以监控节点的存活状态
- Master选举
  Master/slave结构的系统中，避免单点故障，需要多个master，其中只有一个active master，ZooKeeper帮助选择active master
- 分布式锁
  - 独占：某一时刻只有一个client能够获得
  - 控制时序：多个客户端的某些过程按照顺序执行
- 分布式队列
- ……

数据结构

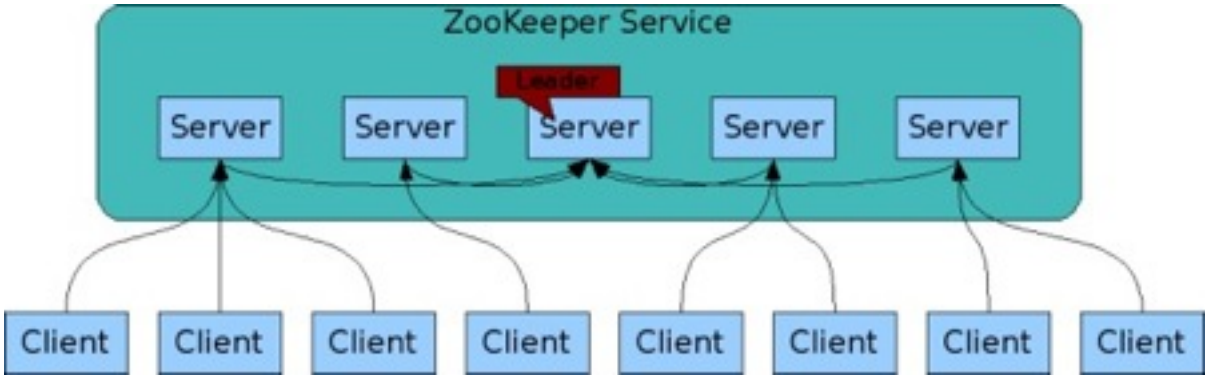


- 类似于文件系统，每一个节点称为ZNode
- 图中：6个ZNode

```
[zk: localhost:2181(CONNECTED) 12] create /app1 app1
Created /app1
[zk: localhost:2181(CONNECTED) 13] create /app1/p_1 p_1
Created /app1/p_1
[zk: localhost:2181(CONNECTED) 14] get /app1
app1
```

- 特点：可以存放数据
- 节点类型
  - 临时节点（API调用）：创建的session失效之后，节点被删除（不存在子节点）
  - 持久化节点：节点一直存在

架构



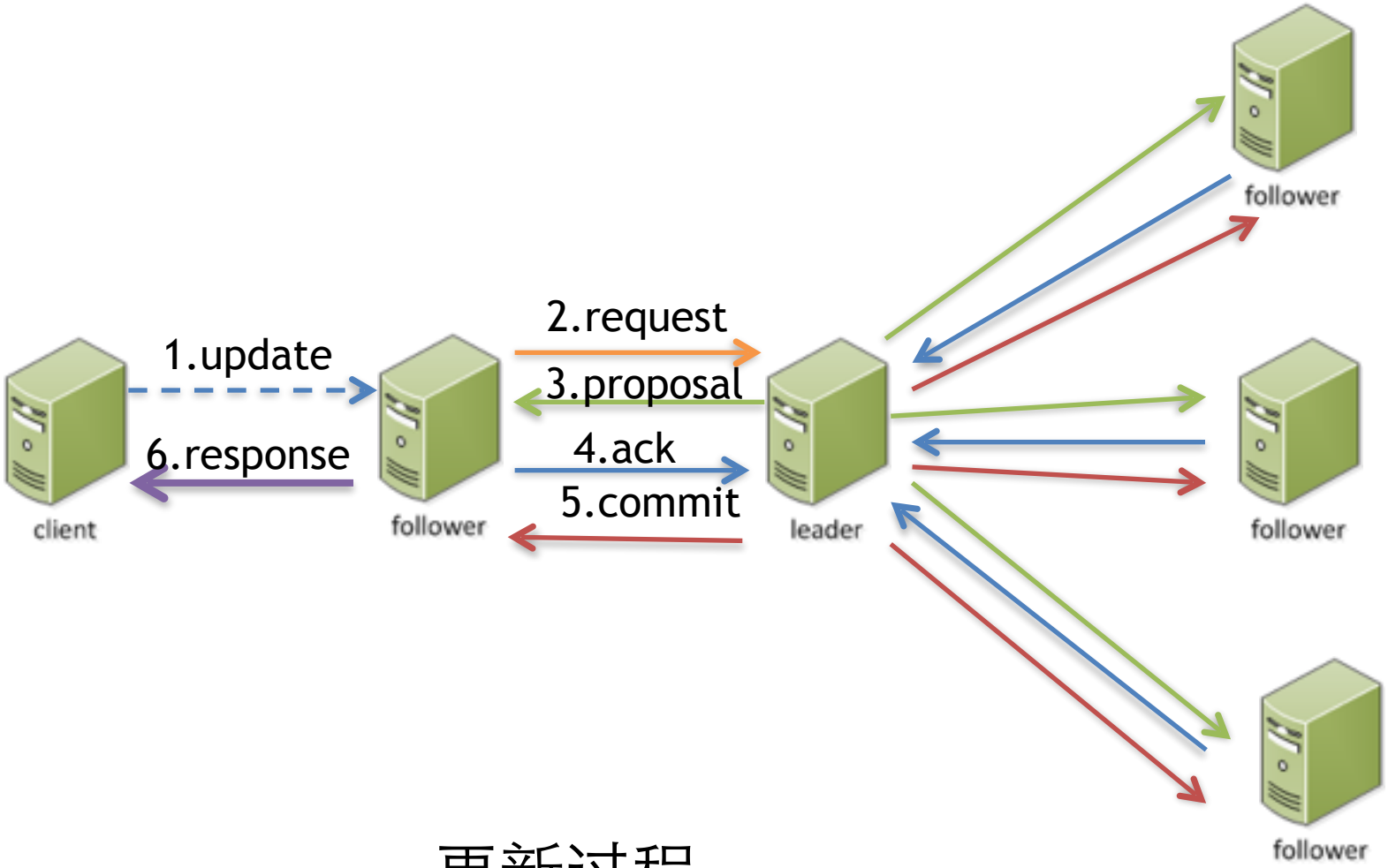
- 节点
  - 2N+1个server组成，N+1个节点可用时，整个系统保持可用
- server角色类型
  - Server：存放数据（in-memory）
  - Leader:不接受client的请求，负责进行投票的发起和决议，最终更新状态
  - Follower:接收客户请求并返回客户结果，参与Leader发起的投票

Zookeeper Leader选择

- 场景
  - 初始化
  - Leader失去Leader的地位时，如节点崩溃
- 使用Paxos算法，大致过程：
  - Step1 每一个server向所有server请求成为Leader
  - Step2 投票数超过一半的成为Leader，否则，重新执行Step1

数据交互

- 读
  - 直接读取client连接的server内存中的数据
- 更新
  1. client向server发送请求
  2. server向leader发送请求
  3. leader发起proposol过程
  4. follower反馈结果
  5. leader接收响应，如果超过一半认为成功，则认为成功，否则认为失败。并将结果反馈给server
  6. server将结果反馈给client


1.update
6.response
client
follower
2.request
3.proposal
4.ack
5.commit
leader
follower
follower
follower

更新过程

ObServer节点

- 问题

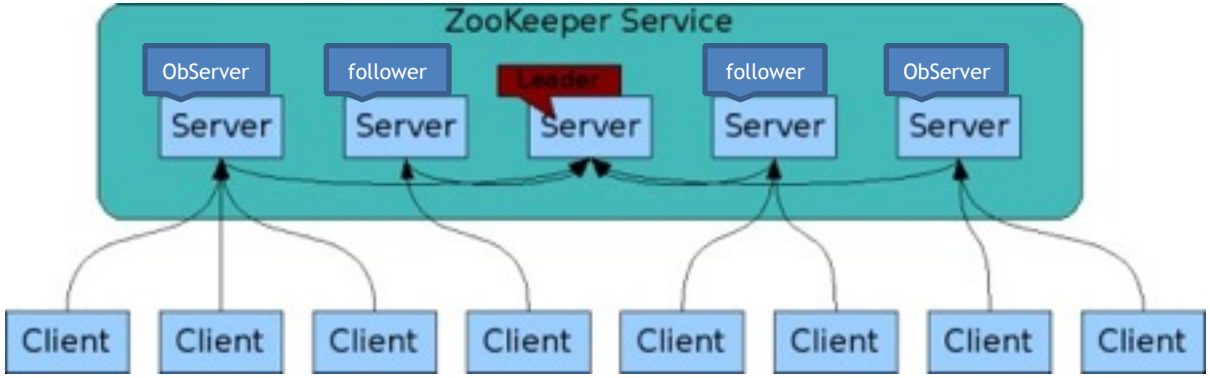
当client变多，ZK集群变大时，节点都为follower时，更新成本较大

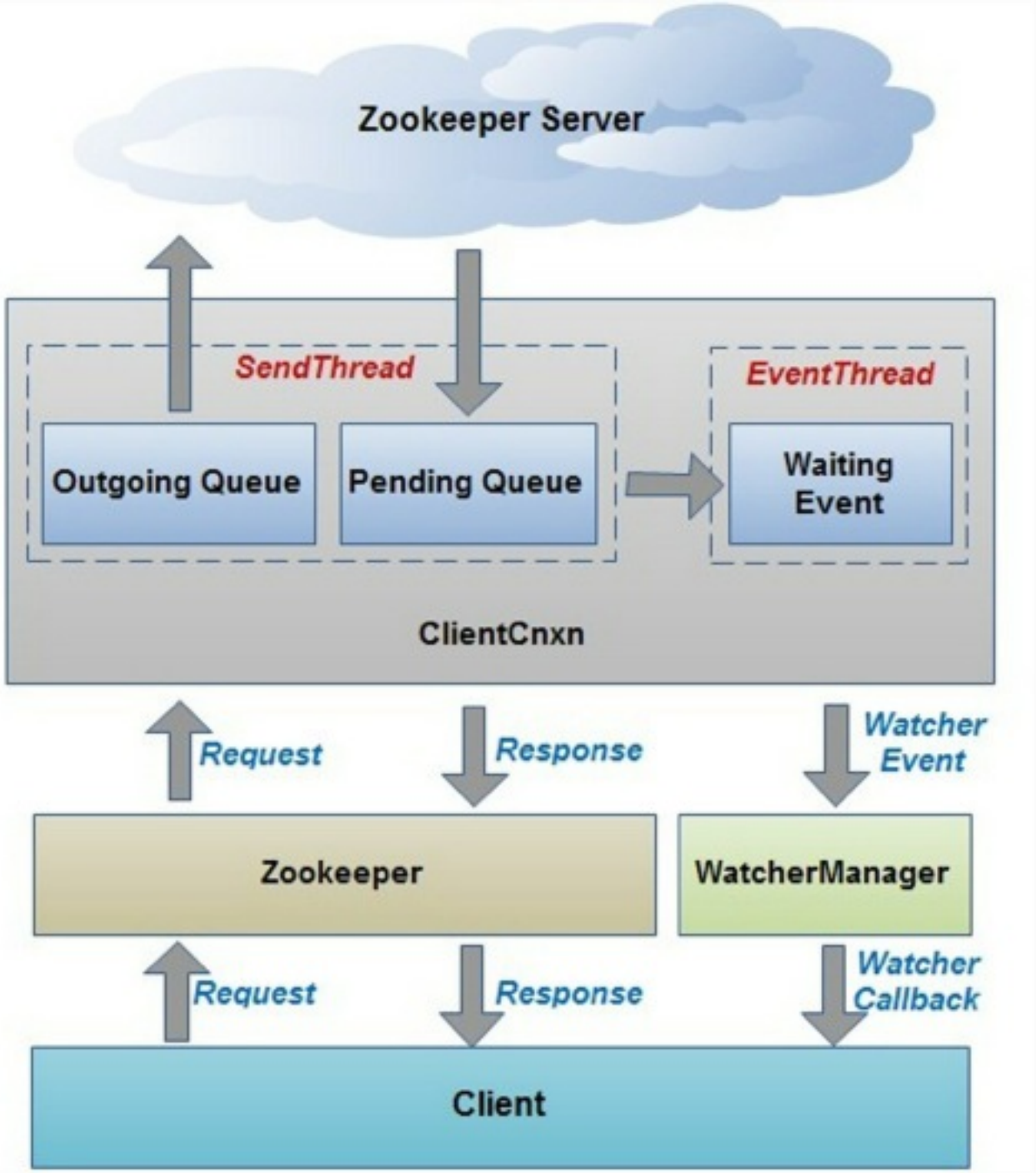
- 解决方案

ObServer节点：和client交互，存有数据的副本，不参与投票

角色

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





client架构

ClientCnxn：管理client和ZooKeeper间的网络连接

WatcherManager：管理Watcher，如：负责对ZNode的监控

Zookeeper：Client交互的主要接口，如创建Znode、更新ZNode

ZooKeeper承诺

- 顺序性

client的update请求都会根据他发出的顺序被顺序的处理

- 原子性

一个update操作要么成功要么失败，没有其他可能的结果

- 强一致性

client无论连接到那个server，展示给它的都是同一个视图

- 可靠性

update一旦成功，就被持久化了，除非另一个update请求更新了当前值

- 实时性

对于每一个client，它的系统视图都是最新的

ZooKeeper安装

- 三个节点为例
- 下载：

http://mirrors.hust.edu.cn/apache/zookeeper/

- 节点

server.1	server.2	server.3
192.168.2.31	*192.168.2.32*	*192.168.2.33*

ZooKeeper安装

1. conf目录下创建zoo.cfg文件，每个节点都要配置

```
maxClientCnxns=50
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/lib/zookeeper
clientPort=2181
server.1=192.168.2.31:2888:3888
server.2=192.168.2.32:2888:3888
server.3=192.168.2.33:2888:3888
```

2. 创建文件/var/lib/zookeeper/myid，3个节点的内容分别为1、2、3

ZooKeeper安装

- 启动：每个节点都要启动

bin/zkServer.sh start

- 查看进程

```
[root@hadoop1 conf]# jps
1291 QuorumPeerMain
```

- 连接

bin/zkCli.sh -server 192.168.2.31:2181

Curator编程

- 对Zookeeper编程进行了封装
- 为了开发者更加友好的使用ZooKeeper
- http://curator.apache.org/

两种异常

- CONNECTION_LOSS（连接丢失）
  - eg:连接的server down掉
- SESSION_EXPIRED（会话失效）
  - 由ZooKeeper集群管理，非client管理
  - ZooKeeper清除和该session有关的信息
  - 正常运行的ZK不会出现这种情况

Curator保证

Curator采用了连接的重试机制，可以保证

1. 所有的操作都会等待，直到连接可用
2. 如果连接失效，Curator采用重连机制
3. 通过重试机制，每一个操作都可以保证在连接失效之后仍能正常运行

常用API

- 参考http://curator.apache.org/curator-framework/index.html
- CuratorFramework
  - 封装有和ZooKeeper进行交互的session
  - 自动重连
  - 线程安全，一个客户端一个CuratorFramework

```
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3)
String
zookeeperConnectionString="192.168.2.31:2181,192.168.2.32:2181,192.168.2.33:2181"
CuratorFramework client =
```

例子中，ZooKeeper每1秒检查一次session状态，如果断掉的话，重新连接，如果连续3次均没连上，则session失效

数据获取及更新

```
//创建ZNode
client.create().creatingParentsIfNeeded().withMode(CreateMode.EPHEMERAL)
                .forPath(“/demo/path”, “demo”.getBytes())

//更新ZNode数据
client.setData().forPath(“/demo/path”, “other data”.getBytes())

//读取ZNode数据
byte[] result=client.getData().forPath(“/demo/path”)

//删除ZNode
client.delete().forPath(“/demo/path”)

//返回值为null表示不存在，否则存在
client.checkExists().forPath(path)

//获取子节点列表
List<String> children=client.getChildren().forPath("/Ptmind");
```

监听器

- NodeCache：监控节点的变化
- PathChildrenCache：监控某个目录的子目录的变化

```
public class ClientHandler {
	PathChildrenCache jobPathCache;
	public ClientHandler(ClientContext context) {
		String path = ZKOperate.ONE.assignTaskPath;
		jobPathCache = new PathChildrenCache(context.client, path, true);
		PathChildrenCacheListener listener = new PathChildrenCacheListener() {
			/*
			 * 顺序执行，不管子节点的变化有多频繁，都能被监听到
			 */
			@Override
			public void childEvent(CuratorFramework client,
					PathChildrenCacheEvent event)
				switch (event.getType()) {
				case CHILD_UPDATED:
				case CHILD_ADDED:
					byte[] data = event.getData().getData();
					dealWithTask(data);
					break;
				case CHILD_REMOVED: {
					break;
				}
				default:
					break;
				}
			}
		};
		jobPathCache.getListenable().addListener(listener);
		jobPathCache.start();
	}
}
```

LeaderLatch

- LeaderLatch用于实现Leader的选举
  - 触发方法isLeader()，表示成为leader
  - 触发notLeader()，表示失去leader权限
  - 场景：多个master中的active master选举
- 原理

  多个client注册同一个路径（/demo/master）,按照顺序注册成临时节点${sessionId}-1 ,${sessionId}-2,${sessionId}-3 ….. 每次选取序列号最小的那个机器作为Leader

```
//LeaderLatchListenerImpl用于active master的选择
public class LeaderLatchListenerImpl implements LeaderLatchListener {
        static String IP=IP=InetAddress.getLocalHost().getHostAddress();
        private final LeaderLatch leaderLatch;
        public LeaderLatchListenerImpl(CuratorFramework client, String path,
                                       String masterID) {
                leaderLatch = new LeaderLatch(client, path, masterID);
                leaderLatch.addListener(this);
                leaderLatch.start();
        }
        @Override
        public void isLeader() {
                System.out.println("become leader，do ... to become leader");
        }
        @Override
```

```
[zk: localhost:2181(CONNECTED) 23] ls /demo/master/_c_

_c_03c8b206-1dcc-44ab-8c4e-cab17d258278-latch-0000000002   _c_32191e2c-823b-479b-9b21-8138c34d2c9d-latch-0000000000
_c_61057fd7-49c5-4c8b-8e96-3f5c2983dd32-latch-0000000003   _c_0328b0a2-935d-4ffc-b1de-8bc66712ced2-latch-0000000001
```

分布式锁

- 保证过程的顺序执行
- 原理
  - 按请求顺序在lockPath下创建临时节点

```
public void doWork(CuratorFramework client, String lockPath) throws Exception  {
            InterProcessSemaphoreMutex lock;
            lock = new InterProcessSemaphoreMutex (client, lockPath);
   /**
    * 1. 等待直到获取锁
    * 2. 使用完成之后必须调用release
    */
            lock.acquire();
            //执行
            lock.release(); // always release the lock in a finally block
}
```

编程注意事项

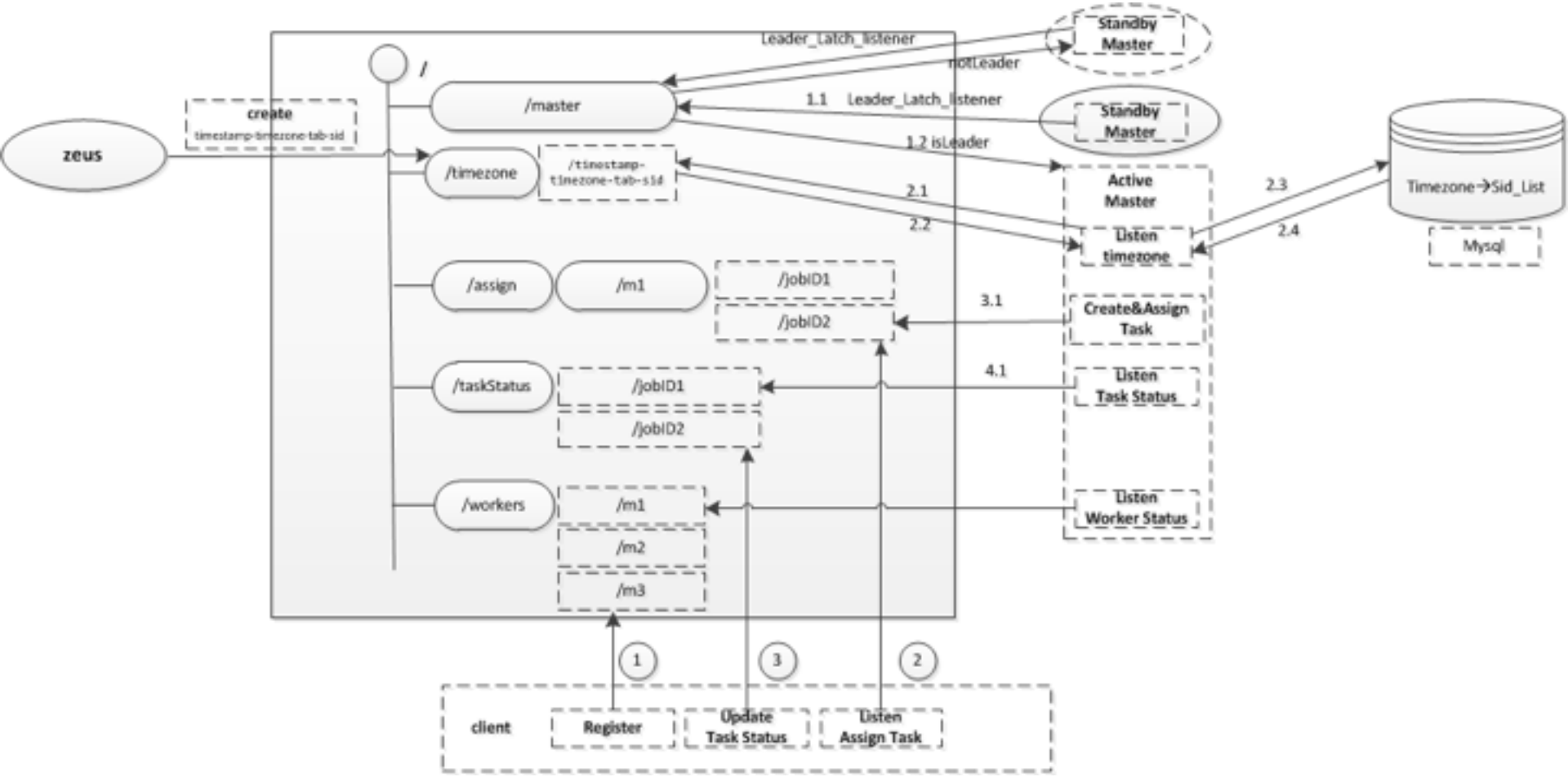
- 子节点数不要太多，否则可能出现程序假死的情况

ZooKeeper在PtBalancer中的应用

- PtBalancer简介
- 消息通信架构

PtBalancer2.0简介

- 功能：执行HDFS向MySQL的导出任务
- 架构：master/slave
  - 解决了master的单点故障
  - 异步通信
- 中间件：ZooKeeper
  - active master选择
  - zeus下发任务
  - master下发任务
  - client的任务执行结果响应


Leader_Latch_listener
Standby Master
notLeader
/
/master
1.1 Leader_Latch_listener
Standby Master
create
timestamp-timezone-tab-sid
zeus
/timezone
/tinostamp-timezone-tab-sid
1.2 isLeader
Active Master
2.1
2.2
2.3
2.4
Timezone→Sid_List
Mysql
Listen timezone
/assign
/m1
/jobID1
/jobID2
3.1
Create&Assign Task
/taskStatus
/jobID1
/jobID2
4.1
Listen Task Status
/workers
/m1
/m2
/m3
Listen Worker Status
1
3
2
client
Register
Update Task Status
Listen Assign Task

消息通信架构

总结

- Zookeeper简介
- ZooKeeper安装
- Curator常用API及其原理
- Zookeeper在PtBalancer中的应用

参考资料

1. 《Hadoop开发者第四期》
2. 《ZooKeeper.Distributed process coordination》
3. http://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/
4. http://nileader.blog.51cto.com/1381108/1040007
5. http://blog.cloudera.com/blog/2009/12/observers-making-zookeeper-scale-even-further/