High Availability in Hadoop

在分布式计算中，选主是一个常见的场景，主节点被选择出来用来控制其他节点或者分配任务，选主算法满足以下特征：

1. 各个节点均衡的获得成功称为主节点的权重，一旦主节点选出，其他节点可以感知谁是主节点，被服从分配
2. 主节点唯一存在
3. 一旦主节点失效，宕机或者断开连接，其他节点能够感知，并且重新进行选主算法

使用Zookeeper来实现Master的基本思路：选择一个根节点，比如/jobMaster，多台机器同时在此节点下面创建一个子节点/jobMaster/lock，Zookeeper保证最终只有一台机器能够创建成功，那么这台机器将成为Master。Zookeepr Curator将上面的思路进行封装，把原始API的节点创建、事件监听和自动选举进行整合封装，提供简单易用的方案，目前提供两种选择算法Leader Latch和Leader Election。此选择功能集中在recipes模块中：

*<dependency>*

*<groupId>org.apache.curator</groupId>*

*<artifactId>curator-recipes</artifactId>*

*<version>${curator-version}</version>*

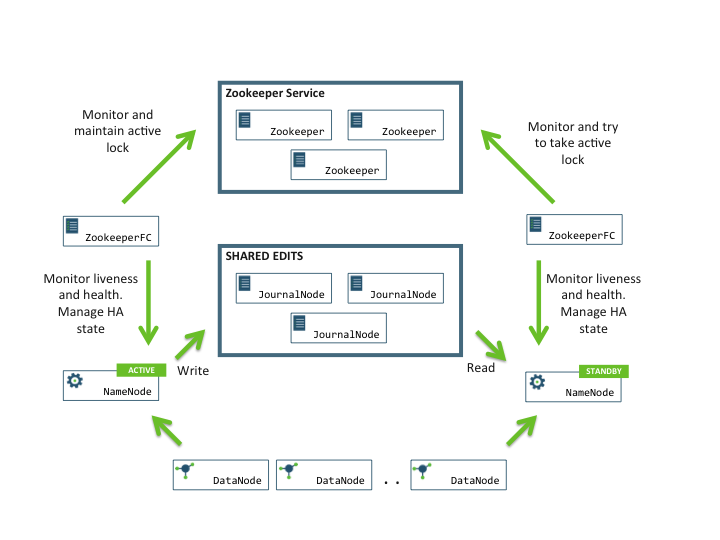
*</dependency>*

Curator Leader有两种实现方式：

1. Leader Latch, 随机从候选中选择一台作为leader，选中之后除非调用close()释放leadship，否则其他的后选择无法成为leader，Spark使用这种方法。
2. Leader Elector，通过LeaderSelectionListener对领导权进行控制，在适当的时候释放领导权，每个节点都有可能获得领导权，而LeaderLatch则一直持有LeaderShip，除非调用close，否则不会释放领导权

# HDFS HA

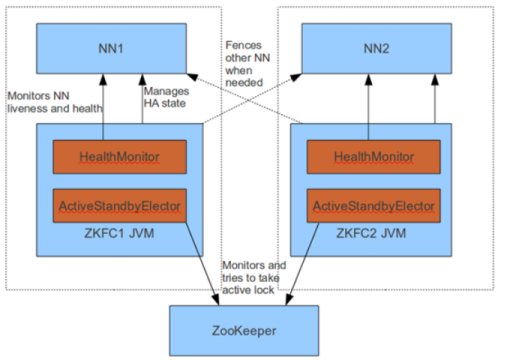
当前HDFS HA是采用QJM/Quorum Journal Manager，基于Paxos算法实现的方案，示意图如下所示：



当发生故障时Active的NN挂掉后，Standby NN会在它成为Active NN前，读取所有JN里面的修改日志，这样就能高可靠的保证与挂掉的NN的目录镜像树一致，然后无缝的接替它的职责，维护来自客户端请求，从而达到一个高可用的目录。下面是对上图中各个角色详细介绍。

## 1.1 ZKFailoverController

ZKFailoverController部署在每个NameNode节点上，作为一个Daemon进程，简称zkfc，示例图如下所示：



主要包括三个组件：

* HealthMonitor，监控NameNode是否处于unavailable或者unhealthy状态，通过RPC调用NN相应的方法完成
* ActiveStandbyElector，管理和监控自己在ZK中的状态
* ZKFailoverController，订阅HealthMonitor和ActiveStandbyElector的事件，并管理NameNode状态

类结构如下图所示：



核心类为ZKFailoverController，其中实现类DFSZKFailoverController作为独立的进程运行在节点上。主要职责如下：

* 健康监测，周期性的通过proxy:HAServiceProtocol向它监控的NN发送健康探测命令，从而来确定某个NN是否处于健康状态，如果机器宕机心跳失败，则ZKFC就会标记它处于一个不健康的状态。
* 会话管理，如果NN监控，zkfc就会在Zookeeper中保持一个打开的会话，如果NN同时是Active状态，那么zkfc会在ZK中占有一个类型为短暂类型的znode，当NN挂掉，则这个znode就会被删除，然后备用的nn将会得到这把锁，称为主NN，同时标记状态为Active
* 当宕机的NN新启动时，会再次注册Zookeeper，发现已经有znode锁，会自动变为Standby状态，如此往复循环，保证高可靠性
* Master选举，通过在zookeeper中维持一个短暂类型的znode，来实现抢占式的锁机制，从而判断那个NameNode为Active状态。

## 1.2 NN Active

当ZKFC确定本主机上的NN称为Active时，调用HAServiceProtocol#becomeActive，即NameNodeRpcServer#becomeActive，源码如下所示：

*public synchronized void transitionToActive(StateChangeRequestInfo req)*

*throws ServiceFailedException, AccessControlException, IOException {*

*checkNNStartup();*

*nn.checkHaStateChange(req);*

*nn.transitionToActive();*

*}*

nn.transitionToActive中将状态变为ACTIVE\_STATE，设置如下：

*ActiveState.setState(context:HAContext,HAState s)*

在设置过程中调用NameNodeHAContext#startActiveServices，源码如下所示：

*public void startActiveServices() throws IOException {*

*try {*

*namesystem.startActiveServices();*

*if(conf.getBoolean(DFSConfigKeys.DFS\_QUOTA\_METRICS\_MONITOR\_ENABLED,*

*DFSConfigKeys.DFS\_QUOTA\_METRICS\_MONITOR\_ENABLED\_DEFAULT)) {*

*quotaMonitor.serviceStart();*

*}*

*startTrashEmptier(conf);*

*} catch (Throwable t) {*

*doImmediateShutdown(t);*

*}*

*}*

对于FSNameSystem.startActiveService，其中会执行：

*editLog.initJournalsForWrite();*

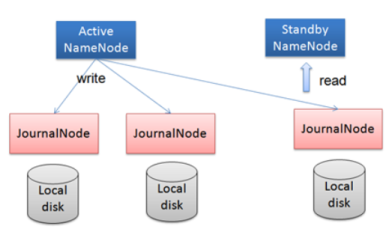
*// May need to recover*

*editLog.recoverUnclosedStreams();*

从JournalNode中获取Editlog。

## 1.3 JournalNode

两个NameNode为了数据同步，通过一组JournalNodes独立进程进行相互通信，当Active状态的NN的NameSpace修改时，会告知大部分的JournalNodes进程。当Standby状态的NN切换成Active时，从JNs中读取Editlog，把EditLog应用于自己的NameSpace，从而达到完全同步。基本原理在2N+1台JN存储Edit Log，每次写操作有大于N+1返回成功则认为该次写成功，数据不会丢失。该操作基于Paxos算法，容忍N台机器挂掉，如果多与N台，则该算法失效，如下图所示：



**1）EditLog的写入**

在NameNode中通过配置指定QuorumJournalManager，如下所示：

*<property>*

*<name>dfs.namenode.edits.journal-plugin.qjournal</name>*

*<value>org.apache.hadoop.hdfs.qjournal.client.QuorumJournalManager</value>*

*</property>*

当要写入Edit Log时，首先获取EditLogStream，在FSEditLog中进行初始化：

*editLogStream = journalSet.startLogSegment(segmentTxId, layoutVersion);*

写入操作如下所示：

*synchronized void logEdit(final int length, final byte[] data) {*

*long start = beginTransaction();*

*try {*

*editLogStream.writeRaw(data, 0, length);*

*} catch (IOException ex) {}*

*endTransaction(start);*

*}*

2) Edit Log的读取

Standby NN在运行过程中从JN中获取Edit Log进行数据的同步。其中EditLogInputStream的初始化如下所示：

*for (RemoteEditLog remoteLog : manifest.getLogs()) {*

*URL url = logger.buildURLToFetchLogs(remoteLog.getStartTxId());*

*EditLogInputStream elis = EditLogFileInputStream.fromUrl(*

*connectionFactory, url, remoteLog.getStartTxId(),*

*remoteLog.getEndTxId(), remoteLog.isInProgress());*

*allStreams.add(elis);*

*}*

* 在运行过程中获取Edit Log，通过FSEditLogLoader来完成，具体操作为loadFSEdits

*private long loadEdits(Iterable<EditLogInputStream> editStreams,*

*FSNamesystem target, StartupOption startOpt, MetaRecoveryContext recovery)*

*FSEditLogLoader loader = new FSEditLogLoader(target, lastAppliedTxId);*

*for (EditLogInputStream editIn : editStreams) {*

*try {*

*loader.loadFSEdits(editIn, lastAppliedTxId + 1, startOpt, recovery);*

*}*

*}*

* Standby切换成Active时获取最新的Edit Log，通过EditLogTailer完成

*void doTailEdits() throws IOException, InterruptedException {*

*………*

*editsLoaded = image.loadEdits(streams, namesystem);}*

## 1.4 Datanode Fencing

Datanode Fencing，确保了只有一个NN能控制DN，其实现Jira为HDFS-1972，核心原理为：

* 每个NN改变状态的时候，向DN发送自己的状态和一个序列号
* DN在运行过程中维护此序列号，当failover时，新的NN在返回DN心跳时会返回自己的Active状态和一个更大的序列号，DN接收到这个返回则认为NN为新的active
* 如果这时原理的active NN恢复，返回给DN的心跳信息包含active状态和原来的序列号，这时DN会拒绝这个NN命令

## 1.5 Client Fencing

客户端Fencing，确保只有一个NN能响应客户端请求，让访问Standby NN的客户端直接失败，在RPC层封装了一次，通过FailoverProxyProvider重试的方式连接NN，配置如下：

*<property>*

*<name>dfs.client.failover.proxy.provider.hadoop-ha</name>*

*<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>*

*</property>*

通过若干次连接NN失败后尝试连接新的NN，对客户端的影响是在重试的时候会增加一定延迟。客户端可以设置重试次数和时间。

http://blog.csdn.net/stark\_summer/article/details/44219095

http://www.cnblogs.com/tgzhu/p/5790565.html

# 2.YARN HA

## 2.1 系统架构图

YARN RM的HA系统架构图如下所示：



RM HA由一对Active/Standby节点构成，通过RMStateStore存储内部数据和主要应用的数据及标记。

## 2.2 源码分析

在Hadoop中YARN 的Leader Selector实现使用了第一种实现，类为CuratorBasedElector

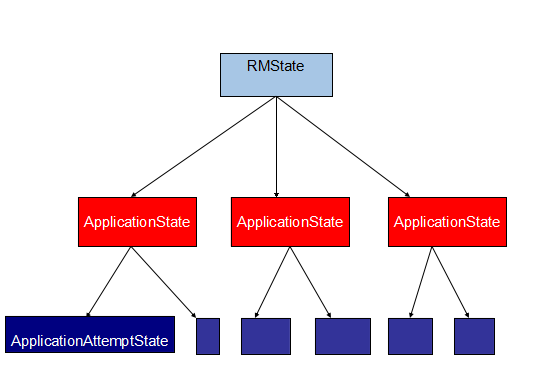
service，类的继承图如下所示：



流程图如下所示：



其中核心是RMStateStore，作用是访问进程异常退出导致的应用状态丢失，重启后无法重跑之前的应用的现象，在RMStateStore中维护RMState信息，如下所示:



类似于MR作业运行的分层执行状态图，RMState中包含若干个ApplicationState的应用状态信息，每个应用状态信息包含很多应用Attempt状态。RMStateStore的实现类如下所示：



* MemoryRMStateStore，状态信息保存在内存中
* FileSystemRMStateStore，状态信息保存在HDFS上，做持久化
* NullRMStateStore，什么都不做，不保存应用状态信息
* LevelDBRMStateStore，状态信息保存在LevelDB中。
* ZKRMStateStore，状态信息保存在Zookeeper中

在YARN中默认配置是FileSystemRMStateStore，配置参数为：

*<property>*

*<name>yarn.resourcemanager.store.class</name>*

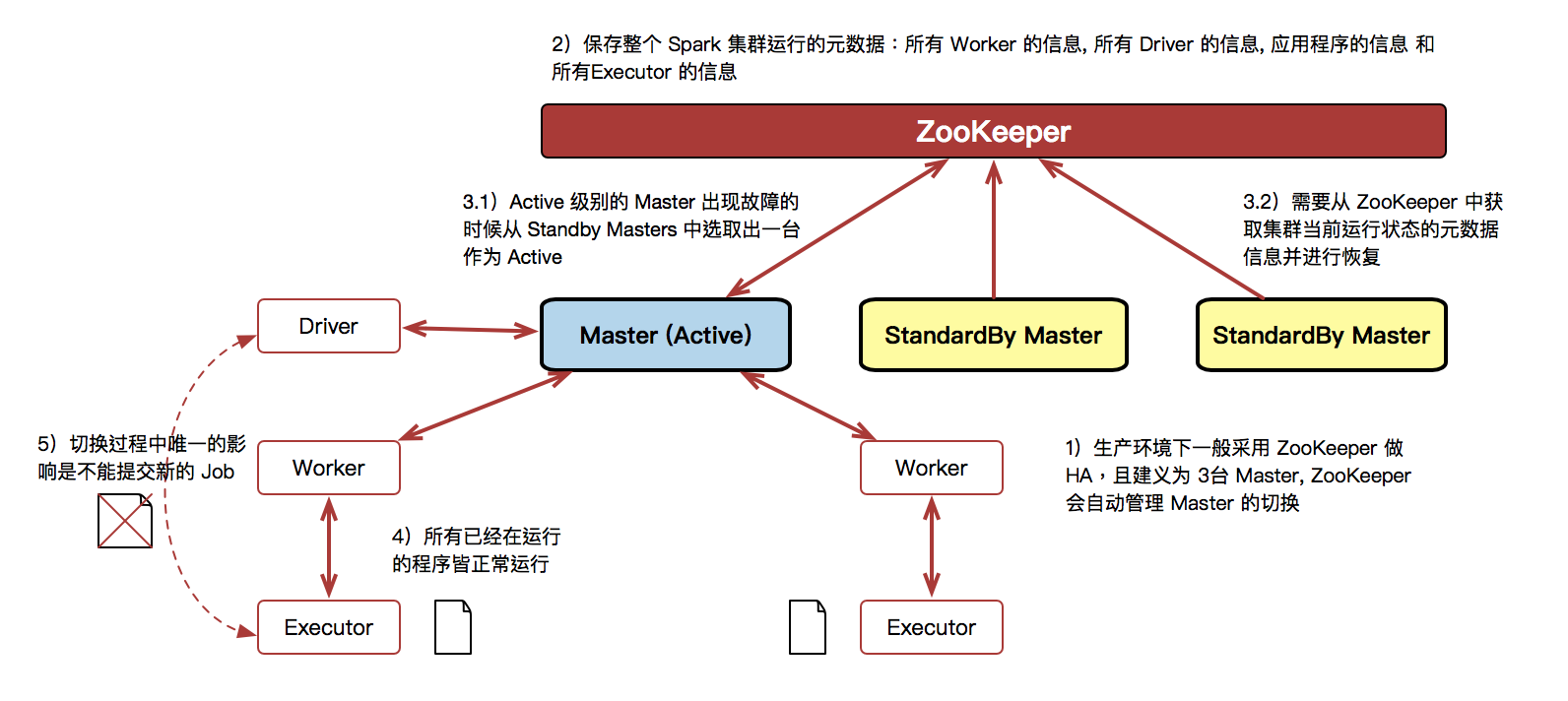
*<value>org.apache.hadoop.yarn.server.resourcemanager.recovery.FileSystemRMStateStore</value>*

*</property>*

# 3.Spark Leader Selector

## 3.1 系统架构及流程

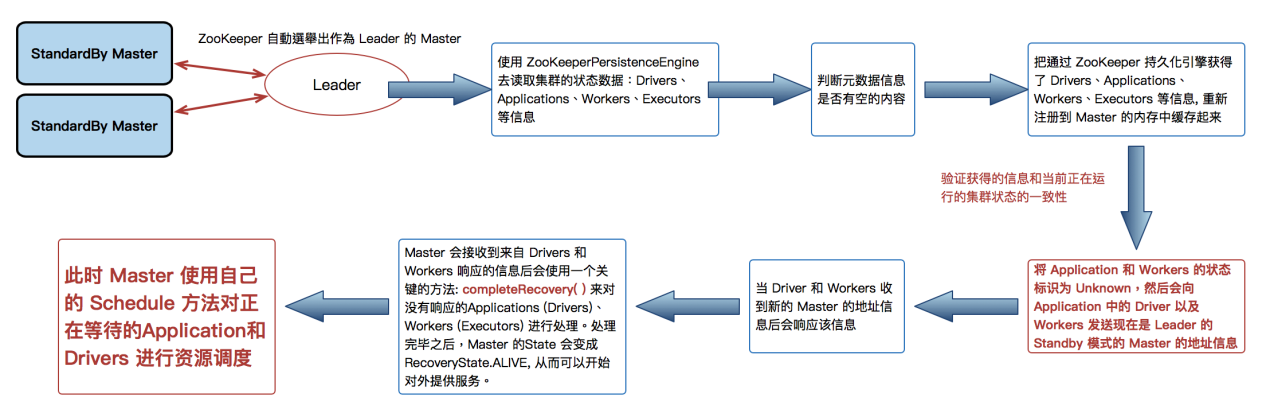
系统架构图如下所示：



使用Zookeeper做HA，Zookeeper会自动管理Master的切换。使用ZookeeperPersistenceEngine存储Spark集群运行的元数据，包括所有Worker、Driver、Application和所有Executor信息。

当Zookeeper遇到当前的Active Master故障时从Standby Masters中选出一台作为Active，该Master从ZookeeperPersistenceEngine中获取集群当前运行状态的元数据信息并进行恢复。在切换过程中，所有在运行的程序正常运行，运行当中的Job本身的调度和处理和Master本身没有任何关系。但是在切换过程中唯一的影响是不能提交新的Job，已经运行的程序也不能因为Action操作触发新的Job请求。

Spark Master HA On Zookeeper切换流程图如下所示：



## 3.2 源码分析

Master HA相关的类图如下所示：



其中Master切换成Leader的源码分析如下所示：

*case ElectedLeader =>*

//1.从persistenceEngine中读取当前运行状态

*val (storedApps, storedDrivers, storedWorkers) = persistenceEngine.readPersistedData(rpcEnv)*

*state = if (storedApps.isEmpty && storedDrivers.isEmpty && storedWorkers.isEmpty) {*

*RecoveryState.ALIVE*

*} else {*

*RecoveryState.RECOVERING*

*}*

*if (state == RecoveryState.RECOVERING) {*

*//2.进行状态的恢复，将状态信息写入Master的进程内存中*

*beginRecovery(storedApps, storedDrivers, storedWorkers)*

*recoveryCompletionTask = forwardMessageThread.schedule(new Runnable {*

*override def run(): Unit = Utils.tryLogNonFatalError {*

*//3.向Driver和Workers发送新的Master地址消息，根据收到的响应信息进行处理*

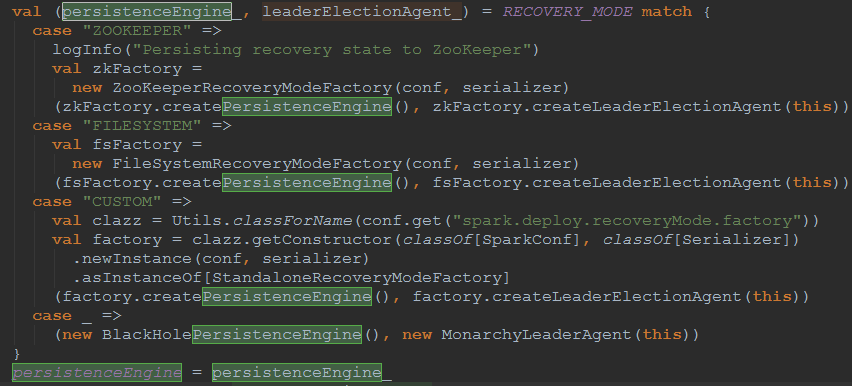
*self.send(CompleteRecovery)*

*}*

*}, WORKER\_TIMEOUT\_MS, TimeUnit.MILLISECONDS)*

*}*

1）从persistenceEngine中读取当前运行状态，当前在Spark中实现了四种PersistenceEngine，如下所示：



* 默认情况下，使用BlockHolePersistenceEngine，不会持久化集群的数据。
* FileSystemPersistenceEngine，将集群元数据保存在Master本地目录中，每个应用或者Worker都保存到一个文件中。

上述两种PersistenceEngine对应的LeaderAgent为MonarchyLeaderAgent，即单节点的Leader，其是实现是直接把传入的Master作为Leader，即相当于没有HA的运行逻辑。

* ZookeeperPersistenceEngine，将集群元数据保存到Zookeeper集群中
* CUSTOM方式允许用户自定义Master HA实现，通过参数spark.deploy.recoveryMode.

Factory参数定义。

目前为了支持Spark HA，ZookeeperPersistenceEngine是常用的元数据存储方式。在PersistenceEngine中重要的方法是使用persists来实现数据持久化。

*/\*\**

*\* Defines how the object is serialized and persisted. Implementation will*

*\* depend on the store used.*

*\*/*

*def persist(name: String, obj: Object): Unit*

通过readPersistedData来获取集群中的元数据：

*/\*\**

*\* Returns the persisted data sorted by their respective ids (which implies that they're*

*\* sorted by time of creation).*

*\*/*

*final def readPersistedData(*

*rpcEnv: RpcEnv): (Seq[ApplicationInfo], Seq[DriverInfo], Seq[WorkerInfo]) = {*

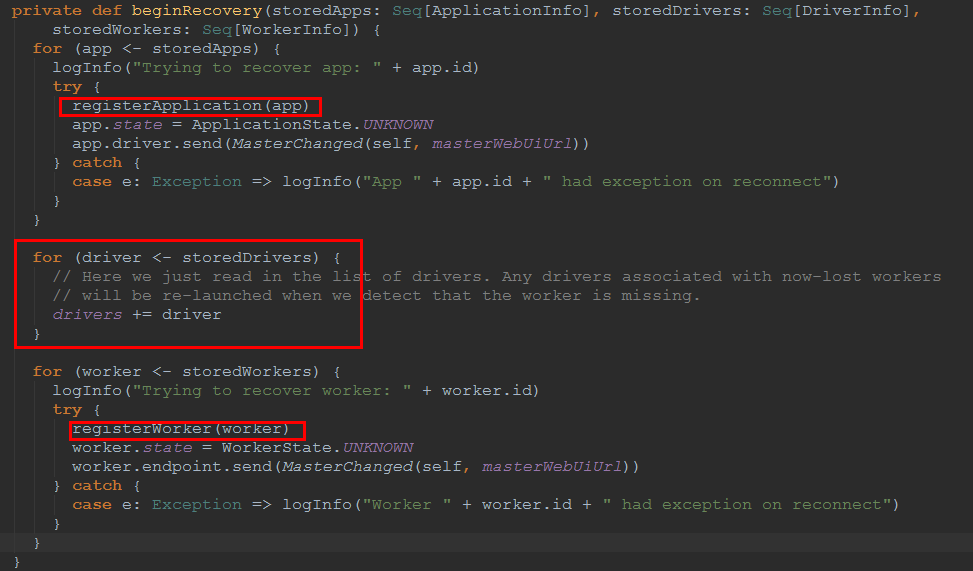
*rpcEnv.deserialize { () =>*

*(read[ApplicationInfo]("app\_"), read[DriverInfo]("driver\_"), read[WorkerInfo]("worker\_"))*

*}*

*}*

2）调用进行beginRecovery进行运行状态的恢复，将状态信息写入Master的进程内存



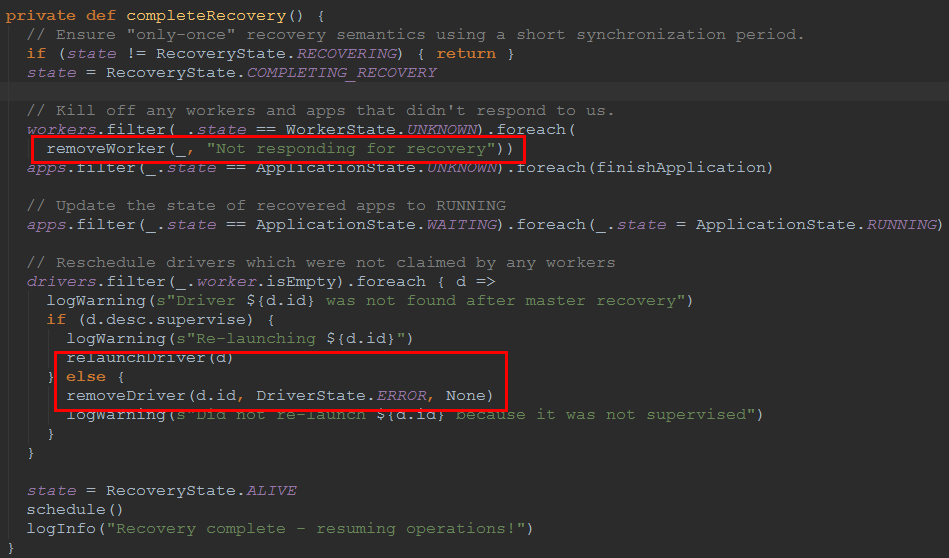
不再详述。将Application和Worker的状态标识为Unkown，然后向Application的Driver和Workers发送新Leader的Master地址信息，在beginRecovery中分别对应的代码为：

*app.driver.send(MasterChanged(self, masterWebUiUrl))*

*......*

*worker.endpoint.send(MasterChanged(self, masterWebUiUrl))*

1. 当Driver和Worker收到新的Master地址信息后会响应该信息，Master会收到来自Driver和Worker的响应信息，调用completeRecovery来对没有响应的Applications和Workers进行处理，处理完毕后，Master的State会变成RecoveryState.ALIVE，从而可以开始对外提供服务。源码截图如下所示：

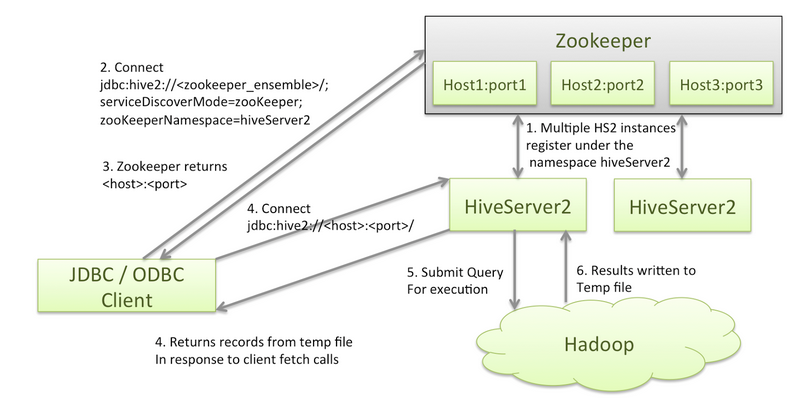


# 4.Hive HA

HA有几种模式，一种是NN/RM/Spark等的Failover Active/Standby切换的HA模型。另外一种则是HiveServer2 HA这种负载均衡的模型。上述已经分析了第一种HA模型，对于HiveServer2 HA模型：

* Server端，在不同的机器上启动多个HiveServer2进程，每个进程在Zookeeper指定的namespace中加载该节点的ThriftServer的host/port信息
* Client端，从ZK指定的ns中随机选择一个ThriftServer的host/port，连接该进程，达到负责均衡的目的
* 对于HA，由ZK的Watch机制完成，发现某台集群的HiveServer2进程挂掉，则对该Znode做标记，这样下次Client端连接就会把该节点自动过滤掉。

如下图所示：



ZK ServiceDiscovery模式是通过下面的参数启用：

*<property>*

*<name>hive.server2.support.dynamic.service.discovery</name>*

*<value>true</value>*

*</property>*

## 4.1 Server端源码分析

在HiveServer2的startHiveServer2方法中，如果启用了service.discovery模式，则会调用addServerInstanceToZookeeper将该ThriftServer的host/port信息加载到Zookeeper中，源码如下所示：

*private static void startHiveServer2() throws Throwable {*

*while (true) {*

*...try {*

*server = new HiveServer2();.....*

*// If we're supporting dynamic service discovery, we'll add the service uri for this*

*// HiveServer2 instance to Zookeeper as a znode.*

*if (hiveConf.getBoolVar(ConfVars.HIVE\_SERVER2\_SUPPORT\_DYNAMIC\_SERVICE\_DISCOVERY)) {*

*server.addServerInstanceToZooKeeper(hiveConf);*

*}....}*

addServerInstanceToZookeeper方法根据传入的HiveConf信息，将配置中的ZK Quorum加载到指定的HIVE\_SERVER2\_ZOOKEEPER\_NAMESPACE中，

*private void addServerInstanceToZooKeeper(HiveConf hiveConf) throws Exception {*

*......*

*String instanceURI = getServerInstanceURI(hiveConf);*

*String pathPrefix =*

*ZooKeeperHiveHelper.ZOOKEEPER\_PATH\_SEPARATOR + rootNamespace*

*+ ZooKeeperHiveHelper.ZOOKEEPER\_PATH\_SEPARATOR + "serverUri=" + instanceURI + ";"*

*+ "version=" + HiveVersionInfo.getVersion() + ";" + "sequence=";*

*znodePath = zooKeeperClient.create().creatingParentsIfNeeded()*

*.withMode(CreateMode.EPHEMERAL\_SEQUENTIAL).forPath(pathPrefix, znodeDataUTF8);*

*setRegisteredWithZooKeeper(true);*

*// Set a watch on the znode*

*if (zooKeeperClient.checkExists().usingWatcher(*

*new DeRegisterWatcher()).forPath(znodePath) == null) {......}*

检查ZK注册的信息，如下所示：

*[zk: localhost:2181(CONNECTED) 0] ls /hiveserver2*

*[serverUri=dw2:10000;version=1.1.0;sequence=0000000006, serverUri=dw1:10000;version=1.1.0-;sequence=0000000007]*

至此注册完成。

## 4.2 Client端源码分析

客户端的JDBC连接URL如下所示：

*jdbc:hive2://***<zookeeper-quorum>***/*<dbName>*;serviceDiscoveryMode=zooKeeper;***zooKeeperNamespace=hiveserver2**

连接成功，则显示如下Shell信息：

*Connecting to jdbc:hive2://dw1:2181,dw2:2181,dw3:2181/default;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2\_zk*

*17/11/30 18:40:52 [main]: INFO jdbc.HiveConnection: Connected to dw1:10000*

*Connected to: Apache Hive (version 1.1.0)....*

*Beeline version 1.1.0-cdh5.9.0 by Apache Hive*

*0: jdbc:hive2://dw1:2181,dw2:2181,dw3:2181/de> show tables;*

*+----------------+--+*

*| tab\_name |*

*+----------------+--+ |*

*| web\_logs | |*

*+----------------+--+*

*5 rows selected (0.147 seconds*

Beeline调用org.apache.hive.jdbc.parseUrl解析jdbc连接字符串，判断serviceDiscoveryMode类型进行解析：

* 如果是Zookeeper，则调用ZookeeperHiveClientHelper解析
* 否则解析出JDBC的host/port

在HiveDriver中调用org.apache.hive.jdbc.Utils#parseURL(uri) 获取JdbcConnectionParams，源码如下所示：

*private static String resolveAuthority(JdbcConnectionParams connParams)*

*throws JdbcUriParseException, ZooKeeperHiveClientException {*

*String serviceDiscoveryMode =*

*connParams.getSessionVars().get(JdbcConnectionParams.SERVICE\_DISCOVERY\_MODE);*

*if ((serviceDiscoveryMode != null)*

*&& (JdbcConnectionParams.SERVICE\_DISCOVERY\_MODE\_ZOOKEEPER*

*.equalsIgnoreCase(serviceDiscoveryMode))) {*

*// Resolve using ZooKeeper*

*return resolveAuthorityUsingZooKeeper(connParams);*

*} else {*

*String authority = connParams.getAuthorityList()[0];*

*URI jdbcURI = URI.create(URI\_HIVE\_PREFIX + "//" + authority);*

*....*

*return jdbcURI.getAuthority();*

*}}*

resloveAuthroityUsingZookeeper中调用ZookeeperHiveClientHelper的getNextServerUri

fromZookeeper方法从ZK NAMESPACE中随机获取hiveServer2的host/port。

在ZookeeperHiveClientHelper#getNextServerUriFromZookeeper(connParams)的源码如下：

*static String getNextServerUriFromZooKeeper(JdbcConnectionParams connParams)*

*throws ZooKeeperHiveClientException {*

*String zooKeeperEnsemble = connParams.getZooKeeperEnsemble();*

*String zooKeeperNamespace =*

*connParams.getSessionVars().get(JdbcConnectionParams.ZOOKEEPER\_NAMESPACE);*

*if ((zooKeeperNamespace == null) || (zooKeeperNamespace.isEmpty())) {*

*zooKeeperNamespace = JdbcConnectionParams.ZOOKEEPER\_DEFAULT\_NAMESPACE;*

*}*

*List<String> serverHosts;*

*Random randomizer = new Random();*

*String serverNode;*

*CuratorFramework zooKeeperClient =*

*CuratorFrameworkFactory.builder().connectString(zooKeeperEnsemble)*

*.retryPolicy(new ExponentialBackoffRetry(1000, 3)).build();*

*zooKeeperClient.start();*

*try {*

*serverHosts = zooKeeperClient.getChildren().forPath("/" + zooKeeperNamespace);*

*// Remove the znodes we've already tried from this list*

*serverHosts.removeAll(connParams.getRejectedHostZnodePaths());*

*….*

*// Now pick a host randomly*

*serverNode = serverHosts.get(randomizer.nextInt(serverHosts.size()));*

*connParams.setCurrentHostZnodePath(serverNode);*

*String serverUri =*

*new String( zooKeeperClient.getData().forPath("/" + zooKeeperNamespace + "/" + serverNode),Charset.forName("UTF-8"));*

*return serverUri;*

*}*

*}*

* 根据conf中获取所有的ZK Quorum节点，从zookeeper nameSpace中获取所有的Server Host对应的的Znode节点
* 去除配置不使用的节点，*getRejectedHostZnodePaths*
* 在剩下的znode节点中随机选出一个Znode节点
* 解析Znode节点的value信息

*zooKeeperClient.getData().forPath("/" + zooKeeperNamespace + "/" + serverNode)*

包含jdbc所需要的host/port。

至此客户端随机选择一个HiveServer2，通过HiveDriver建立了JDBC连接。

## 4.3 HA处理

当某一台host上的HiveServer2进程挂掉的时候，会调用HiveServer2.stop方法，

*if (hiveConf.getBoolVar(ConfVars.HIVE\_SERVER2\_SUPPORT\_DYNAMIC\_SERVICE\_DISCOVERY)) {*

*try {*

*removeServerInstanceFromZooKeeper();*

*} catch (Exception e) {*

*LOG.error("Error removing znode for this HiveServer2 instance from ZooKeeper.", e);*

*}*

*}*

在该方法中会检测是否使用*HIVE\_SERVER2\_SUPPORT\_DYNAMIC\_SERVICE\_DISCOVERY*模式，如果是该模式则会将Zookeeper相应的实例删除掉。

在创建Server的Znode节点，同时为该pathPrefix添加了ZK DeRegisterWatcher，其实现如下所示：

*private class DeRegisterWatcher implements Watcher {*

*@Override*

*public void process(WatchedEvent event) {*

*if (event.getType().equals(Watcher.Event.EventType.NodeDeleted)) {*

*HiveServer2.this.setRegisteredWithZooKeeper(false);*

*// If there are no more active client sessions, stop the server*

*if (cliService.getSessionManager().getOpenSessionCount() == 0) {*

*HiveServer2.this.stop();*

*}*

*}*

*}*

*}*

参考文献：

HDFS HA:http://www.cnblogs.com/tgzhu/p/5790565.html

HiveServer2 HA: https://www.jianshu.com/p/9865fd29ad6b

http://blog.csdn.net/wo541075754/article/details/70216046

http://curator.apache.org/curator-recipes/leader-election.html

Spark Master HA实现：https://www.cnblogs.com/jcchoiling/p/6427862.html