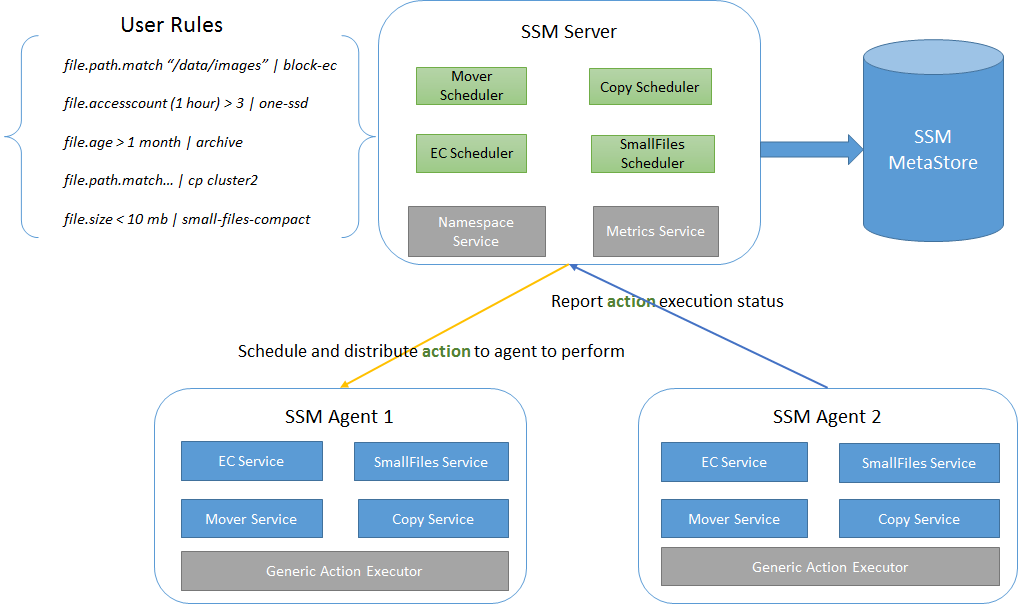
SSM源码分析

SSM系统架构图如下所示：



1. SSM MetaStore，使用数据库存储文件管理policy元数据，支持多种数据库包括MySQL，Embed数据库TiDB/TiKV等
2. SSM Server，从Hadoop集群中获取文件存储及访问信息，并根据MetaStore中定义的文件管理Policy，生成执行命令
3. SSM Agent，接收SSM Server分配的命令并执行
4. Web，还提供基于Zeppelin的UI，方便用户定义policy及文件服务的管理

SSM是典型的Master-Slave架构，并且有内置分布式数据库，用来存储文件状态和用户指定的规则。

SSM的项目目录结构如下：

*# tree -L 1*

*.*

*├── bin //启动脚本*

*├── conf //配置目录*

*├── dist //zeppelin war包*

*├── pom.xml*

*├── smart-action //文件操作action*

*├── smart-admin //Smart 管理RPC*

*├── smart-agent //Smart Aget*

*├── smart-alluxio-support //支持Alluxio的文件系统的管理*

*├── smart-client //Smart RPC 客户端*

*├── smart-common //定义服务通用类及描述*

*├── smart-dist //项目的打包*

*├── smart-engine*

*//Smart的核心module,包括集群的管理，rules与action的解析及生成*

*├── smart-hadoop-support //用于hadoop的访问*

*├── smart-integration*

*├── smart-maven-plugins //项目编译包*

*├── smart-metastore //Metastore Dao，各表的操作*

*├── smart-metrics //用于描述文件访问事件*

*├── smart-rule //rule的定义及解析，使用antrl4描述*

*├── smart-server // Smart Server*

*├── smart-tidb //内嵌使用tidb，作为MetaStore DB*

*├── smart-zeppelin //Smart Zeppelin*

*├── supports*

# SSM Server

SSM Server的主要功能：Rule的管理、收集Hadoop集群信息并根据以上指标生成命令发送给SSM Agent，其模块如下图如所示：



1）SmartZeppelinServer和SmartRPCServer，对外提供的RESTFul API及RPC客户端，Web端通过Zeppelin来提供可操作界面

1. ServerContext，主要封装了MetaStore，用于保存元数据
2. SmartEngine是SmartServer的核心，其中包括：

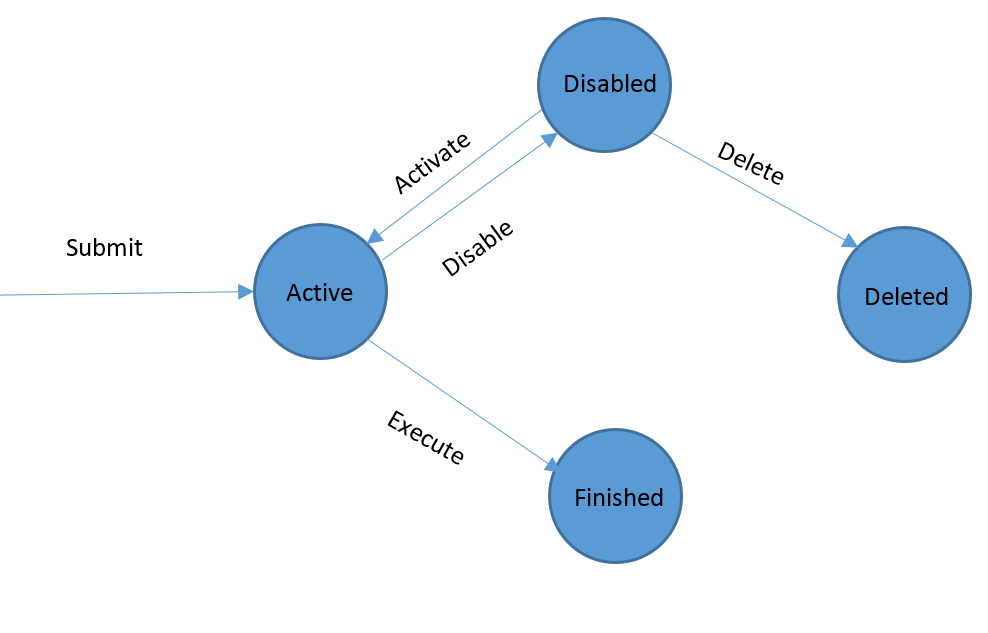
* StateManager，从文件系统中获取文件访问信息来支撑文件操作的命令执行
* RuleManager，接收客户端或者Web UI提交的文件操作Rule，进行持久化及开始执行流程
* CmdletManater，根据StateManager和RuleManager维护的数据，获取文件管理命令
* AgentExecutorServer，维护SmartAgent集群，下发CmdletManager生成的命令

1. HazelcastExecutorService，实现SmartEngine的高可用，通过分布式缓存共享ScheduledCmdlets

下面对核心流程进行源码分析。

## **Rule的生命周期管理**

在SSM中，文件的管理以Rule为核心，系统中Rule有4中状态：active,disabled,finished及deleted，转换图如下所示：



1. Submit，通过RESTFul API或者RPC提交Rule，Rule的定义不再这里介绍

* SmartAdminProtocol，RPC接口，定义如下：



* 通过RESTFul API来提交Rule，接口定义在RuleRestAPI类中

*@POST*

*@Path("/add")*

*public Response addRule(@FormParam("ruleText") String ruleText) {*

*try {*

*//调用RuleManager*

*t = smartEngine.getRuleManager().submitRule(ruleText, RuleState.DISABLED);*

*} ...*

*return new JsonResponse(Response.Status.CREATED, t).build();*

*}*

通过分析，可知Rule Sumit都是通过RuleManager来提交，其相关类如下：



其执行如下：

*public long submitRule(String rule, RuleState initState) throws IOException {*

*//Rule的解析，生成CmdletDescriptor*

*TranslateResult tr = doCheckRule(rule, null);*

*doCheckActions(tr.getCmdDescriptor());*

*//RuleInfo，用于track其执行流程*

*RuleInfo.Builder builder = RuleInfo.newBuilder();*

*builder.setRuleText(rule).setState(initState);*

*RuleInfo ruleInfo = builder.build();*

*RulePluginManager.onAddingNewRule(ruleInfo, tr);*

*//将Rule写入数据库*

*try {metaStore.insertNewRule(ruleInfo); } ......*

*RuleInfoRepo infoRepo = new RuleInfoRepo(ruleInfo, metaStore);*

*mapRules.put(ruleInfo.getId(), infoRepo);*

*//提交Rule，当initState为Active时触发执行*

*submitRuleToScheduler(infoRepo.launchExecutor(this));*

*RulePluginManager.onNewRuleAdded(ruleInfo, tr);*

*return ruleInfo.getId();*

*}*

以rule:

*file:path matches "/tmp/\*" and age > 5min | delete*

为例，解析后sqlStatement为：

*SELECT path FROM file WHERE (((path LIKE '/tmp/%') AND (($NOW - modification\_time) > 300000)))*

CMD描述为：

*CmdletDescriptor {*

*"actionCommon":""*

*"actionNames":"delete",*

*"actionArgs":"",*

*"cmdletString":"delete"}*

1. Active，提交Rule的默认状态为disable，可以通过API将其active，其相关类图：



将Rule解析为RuleInfoRepo、TranslationContext及ExecutionContext后，提交到Scheduler中执行周期Task，如下所示：

*public void addPeriodicityTask(RuleExecutor re) {  
 TimeBasedScheduleInfo si = re.getTranslateResult().getTbScheduleInfo(); //调度时间  
 long now = System.currentTimeMillis();  
 si.setSubScheduleTime(now);  
 long startDelay = si.getStartTime() - now;  
 if (startDelay < 0) {  
 startDelay = 0;  
 }  
 long every = si.getEvery(); //获取执行周期，默认为5000ms，即5秒  
 if (every <= 0) {  
 every = 5000;  
 }  
 service.scheduleAtFixedRate(re, startDelay, every, TimeUnit.MILLISECONDS); //周期执行  
}*

1. disable及delete，其执行过程相对简单，不再详述，

*getCmdletManager().dropRuleCmdlets(ruleID);*

*getCmdletManager().deleteCmdletByRule(ruleID);*

涉及到到CmdletManager中数据的清除和MetaStore中数据清理。

## **RuleExecutor的执行**

在RuleExecutor中封装了调度时间周期及执行的任务，Rule Active后，通过ScheduledExecutorService调度Executor的执行，在执行过程中根据Rule的定义和文件信息，生成cmdlet，转发给CmdletManager进行调度执行，其执行如下：

*@Override*

*public void run() {*

*long startCheckTime = System.currentTimeMillis();*

*boolean doExec = true;*

*//check Rule执行时间信息，是否执行*

*if (doExec) {*

*//获取待处理的文件，文件管理的核心*

*files = executeFileRuleQuery();*

*......*

*}*

*endCheckTime = System.currentTimeMillis();*

*if (doExec) { //后面再分析ExecutorPlugin的作用*

*for (RuleExecutorPlugin plugin : plugins) {*

*files = plugin.preSubmitCmdlet(info, files);*

*}*

*//提交cmdlets*

*numCmdSubmitted = submitCmdlets(info, files);*

*}*

*//更新执行信息*

*ruleManager.updateRuleInfo(rid, null, System.currentTimeMillis(), 1, numCmdSubmitted);*

*.....*

*}*

### 获取执行的文件

RuleExecutor在执行executeFileRuleQuery，根据Rule中定义的文件操作条件获取哪些文件需要处理，在Rule的解析章节已经获取了文件筛选sql，如：

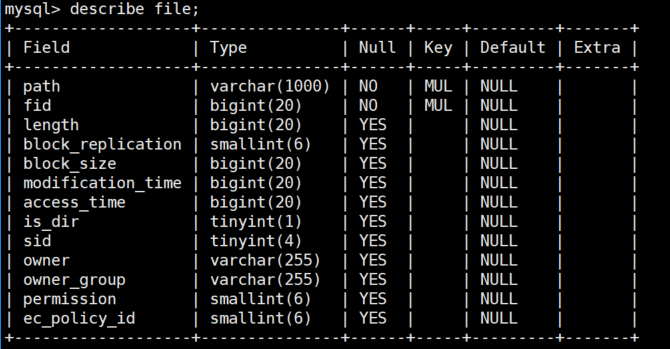
*file:path matches "/tmp/\*" and age > 5min | delete*

中解析的SQL Statement为：

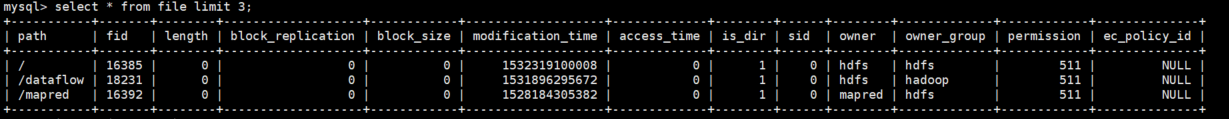
*SELECT path FROM file WHERE (((path LIKE '/tmp/%') AND (($NOW - modification\_time) > 300000)))*

从表File中获取大于5分钟且匹配/tmp/\*文件。

File表结构如下：



字段值示例如下：

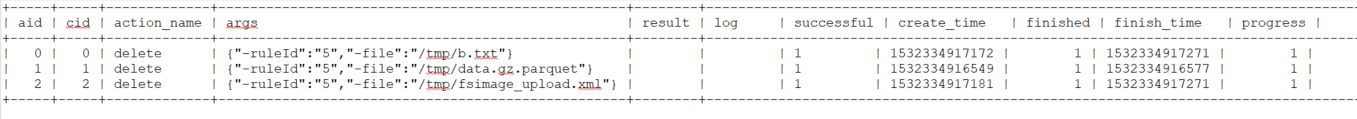


### **向CmdletManager发送执行命令，加入到调度缓存中**

Cmdlet执行的相关信息描述如下：



其执行信息封装在ActionInfo及CmdletInfo中，这些信息会存储到Metastore中，分别对应action和cmdlet表，分别如下：

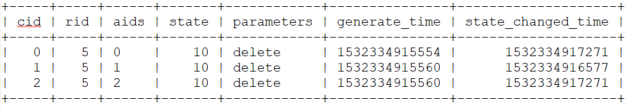


其log示例如下：

*Action starts at Mon Jul 23 16:35:17 CST 2018 : Delete /tmp/fsimage\_upload.xml*

*Delete /tmp/fsimage\_upload.xml*

cmdlet表如下：



将命令执行信息放到CmdletManager的map缓存中等待调度，其执行syncCmdAction如下：

*private void syncCmdAction(CmdletInfo cmdletInfo,  
 List<ActionInfo> actionInfos) throws IOException {  
 lockMovefileActionFiles(actionInfos);  
 for (ActionInfo actionInfo : actionInfos) {  
 idToActions.put(actionInfo.getActionId(), actionInfo);   
 }  
 idToCmdlets.put(cmdletInfo.getCid(), cmdletInfo);  
  
 if (cmdletInfo.getState() == CmdletState.PENDING) {   
 numCmdletsGen.incrementAndGet();  
 cacheCmd.put(cmdletInfo.getCid(), cmdletInfo);  
 synchronized (pendingCmdlet) {  
 pendingCmdlet.add(cmdletInfo.getCid());  
 }  
 } else if (cmdletInfo.getState() == CmdletState.DISPATCHED) {  
 runningCmdlets.add(cmdletInfo.getCid());  
 LaunchCmdlet launchCmdlet = createLaunchCmdlet(cmdletInfo);  
 idToLaunchCmdlet.put(cmdletInfo.getCid(), launchCmdlet);  
 }  
}*

调度通过其执行为ScheduleTask，如下：

*private class ScheduleTask implements Runnable {  
 @Override  
 public void run() {  
 try {  
 int nScheduled;  
 batchSyncCmdAction(); //处理pending的action及cmd，加入待调度map中  
 do {  
 nScheduled = scheduleCmdlet();   
 totalScheduled += nScheduled;  
 } while (nScheduled != 0);  
 } ....}*

### **SchedulerService，补全ActionInfo**

在CmdletManager中调度执行中最终使用的方法是scheduleCmdlet

*private int scheduleCmdlet() throws IOException {*

*......*

*Iterator<Long> it = schedulingCmdlet.iterator();*

*while (it.hasNext() && !shouldStopSchedule()) {*

*//获取待执行的cmdlet*

*CmdletInfo cmdlet = idToCmdlets.get(id);*

*......*

*synchronized (cmdlet) {*

*switch (cmdlet.getState()) { //根据状态进行实际的执行*

*case CANCELLED:*

*case DISABLED:*

*it.remove();*

*break;*

*case PENDING:*

*if (cmdlet.getDeferedToTime() > curr) {*

*break;*

*}*

*//初始化LaunchCmdlet*

*LaunchCmdlet launchCmdlet = createLaunchCmdlet(cmdlet);*

*ScheduleResult result;*

*try { //核心方法*

*result = scheduleCmdletActions(cmdlet, launchCmdlet);*

*} .....*

*}*

scheduleCmdletActions(cmdlet,launchCmdlet)

*private ScheduleResult scheduleCmdletActions(CmdletInfo info,*

*LaunchCmdlet launchCmdlet) {*

*//获取actIds*

*List<Long> actIds = info.getAids();*

*......*

*for (idx = 0; idx < actIds.size(); idx++) {*

*actionInfo = idToActions.get(actIds.get(idx));*

*launchAction = launchCmdlet.getLaunchActions().get(idx);*

*//根据actionName，获取对应的schedulers*

*actSchedulers = schedulers.get(actionInfo.getActionName());*

*......*

*for (schIdx = 0; schIdx < actSchedulers.size(); schIdx++) {*

*ActionScheduler s = actSchedulers.get(schIdx);*

*try {*

*//Scheduler调用LaunchAction*

*scheduleResult = s.onSchedule(actionInfo, launchAction);*

*} ......*

*}*

*}*

*.....*

*//触发通用的action*

*postscheduleCmdletActions(actIds, scheduleResult, idx, schIdx);*

*return scheduleResult;*

*}*

其均通过SchedulerService，进行调度，类图如下：



### **CmdletDispatcher，将命令分发到Agent**

将Pending的Cmdlet进行分发到Agent执行，执行Task为DispatchTask，相关类如下：

****

其执行如下：

*@Override*

*public void run() {*

*.....*

*boolean redisp = launchCmdlet != null;*

*boolean disped;*

*while (resvExecSlot()) { //有空闲Slot*

*disped = false;*

*try {*

*if (launchCmdlet == null) { //获取待执行的launchCmdlet*

*launchCmdlet = getNextCmdletToRun();*

*}*

*.......*

*if (!dispatch(launchCmdlet)) {*

*......*

*}*

dispatch命令的执行为CmdletDispatcher#dispatch(launchCmdlet)，执行如下：

*private boolean dispatch(LaunchCmdlet cmdlet) {*

*//获取srvId，执行类型：LOCAL/REMOTE\_SSM/AGENT*

*do {*

*dispInstIdxs[srvId] = (dispInstIdxs[srvId] + 1) % cmdExecSrvNodeIds.get(srvId).size();*

*//获取执行的nodId*

*nodeId = cmdExecSrvNodeIds.get(srvId).get(dispInstIdxs[srvId]);*

*counter = regNodes.get(nodeId);*

*.....*

*} while (true);*

*//补全所执行的nodeId*

*cmdlet.setNodeId(nodeId);*

*//调用CmdletExecutoService#execute，执行cmdlet*

*selected.execute(cmdlet);*

*dispSucc = true;*

*} .....*

*}*

默认使用AgentExecutorService#execute(LaunchCmdlet)，执行如下：

*AgentMaster#launchCmdlet(cmdlet:LaunchCmdlet)*

执行如下：

*Object askMaster(Object message) throws Exception {*

*//Message为LaunchCmdlet  
 Future<Object> answer = Patterns.ask(master, message, TIMEOUT);  
 return Await.result(answer, TIMEOUT.duration());  
}*

至此SSM Server端文件管理命令的解析、生成及分发基本流程已走完。LaunchCmdlet任务通过Akka框架分发到SSM Agent。

# SSM Agent

AgentMaster通过调用askMaster将LaunchCmdlet文件管理命令发送到SSM Agent，Agent接收到信息进行处理，SSM Agent系统结构图如下：



SmartAgent接收到要执行的命令通过AgentCmdletService#execute来运行，如下：

*@Override  
public void execute(Message message) throws Exception {  
 if (message instanceof LaunchCmdlet) {  
 executor.execute(factory.createCmdlet((LaunchCmdlet) message));  
 }....*

*}*

LaunchCmdlet转换类图如下所示：



Cmdlet为可执行线程，如下：

*private void runAllActions() {*

*state = CmdletState.EXECUTING;*

*stateUpdateTime = System.currentTimeMillis();*

*Iterator<SmartAction> iter = actions.iterator();*

*while (iter.hasNext()) {*

*SmartAction act = iter.next();*

*.....*

*act.init(act.getArguments());*

*act.run();*

*......*

*}*

SmartAction是根据LaunchCmdlet中的action 和参数生成的具体文件操作，以MoveFile为例：

*@Override*

*protected void execute() throws Exception {*

*......*

*if (movePlan.isDir()) {*

*dfsClient.setStoragePolicy(fileName, storagePolicy);*

*appendResult("Directory moved successfully.");*

*return;*

*}*

*int totalReplicas = movePlan.getBlockIds().size();*

*.....*

*//执行move操作*

*int numFailed = move();*

*.....}*

# 文件信息的采集

在第一部分分析，SmartEngine从MetaStore数据库中查询文件访问信息，然后进行操作，在运行过程中SmartServer通过StateManager获取文件信息，其相关类图如下所示：



## **文件访问热度**

1. AccessEventFetcher，启动如下：

*this.scheduledFuture = scheduledExecutorService.scheduleAtFixedRate(  
 fetchTask, toWait, fetchInterval, TimeUnit.MILLISECONDS);*

fetchInterval默认1000ms，周期性执行FetchTask，其执行如下：

*@Override*

*public void run() {*

*try {*

*List<FileAccessEvent> events = this.collector.collect();*

*if (events.size() > 0) {*

*this.manager.onAccessEventsArrived(events);*

*}*

*}......*

*}*

获取文件访问事件，事件的描述如下：



Event的事件源为Reader，执行初始化及执行如下：

*@Override  
public List<FileAccessEvent> collect() throws IOException {  
 try {  
 if (reader.exists(now)) {  
 reader.seekTo(now, false);  
 List<FileAccessEvent> events = new ArrayList<>();  
 while (reader.hasNext()) {  
 Info info = reader.next();  
 events.add(new FileAccessEvent(info.getPath(), info.getTimestamp()));  
 now = info.getTimestamp();  
 }......*

*}*

Reader从hadoop-metrics2.properties配置中获取文件访问信息

*source = conf.getString(SOURCE\_KEY, SOURCE\_DEFAULT);*

将文件访问事件持久化到数据库中

*public void addAccessEvents(List<FileAccessEvent> eventList) {*

*if (this.currentWindow == null && !eventList.isEmpty()) {*

*this.currentWindow = assignWindow(eventList.get(0).getTimestamp());*

*}*

*for (FileAccessEvent event : eventList) {*

*if (!this.currentWindow.contains(event.getTimestamp())) {*

*// New Window occurs*

*this.createTable();*

*this.currentWindow = assignWindow(event.getTimestamp());*

*this.eventBuffer.clear();*

*}.....*

*}*

Table的聚合时间默认为5\*1000ms，初始化见AccessEventAggregator#aggregationGranularity。创建表并将文件访问统计信息写入数据库中，每隔5s中创建一个AccessAccount表，在数据库中如下所示：

|  |  |
| --- | --- |
|  |  |

表中的信息为fid及访问次数，表名中为时间窗口，在该端时间内文件fid被访问了多少次。

1. AccessCountTable的使用

根据表中的统计数据，来判断文件的冷热程度

*public List<FileAccessInfo> getHotFiles(List<AccessCountTable> tables,*

*int topNum) throws IOException {*

*try {*

*if (topNum == 0) {*

*topNum = serverContext.getConf().getInt(SmartConfKeys.SMART\_TOP\_HOT\_FILES\_NUM\_KEY,*

*SmartConfKeys.SMART\_TOP\_HOT\_FILES\_NUM\_DEFAULT);*

*return serverContext.getMetaStore().getHotFiles(tables, topNum);*

*}.....*

*}*

1. AccessCountTable的清理，执行类为TableEvictor，相关类为：



*AccessCountTableDeque dayTableDeque =  
 new AccessCountTableDeque(*

*new CountEvictor(metaStore, NUM\_DAY\_TABLES\_TO\_KEEP)); //默认30个表*

在添加Table时，AccessCountTableManager#addTable进行清理

*public boolean addAndNotifyListener(AccessCountTable table) {*

*.....*

*tableEvictor.evictTables(this, this.size());  
 return true;  
}*

## **文件信息**

HdfsStatesUpdateService用于收集文件信息



其中

* CacheFileFetcher用于获取缓存文件信息
* DataNodeInfoFetcher获取数据节点的存储信息
* InotifyEventFetcher用于获取文件信息

下面以InotifyEventFetcher为例，介绍其文件获取流程

NamespaceFetcher通过启动IngestionTask来获取文件信息，执行如下：

*@Override*

*public void run() {*

*//周期为2秒钟*

*long curr = System.currentTimeMillis();*

*if (curr - lastUpdateTime >= 2000) {*

*.....*

*lastUpdateTime = curr;*

*}*

*....*

*try {*

*//父目录文件信息*

*HdfsFileStatus status = client.getFileInfo(parent);*

*if (status != null && status.isDir()) {*

*if (startAfter == null) {*

*FileInfo internal = convertToFileInfo(status, "");*

*internal.setPath(parent);*

*this.addFileStatus(internal);*

*numDirectoriesFetched.incrementAndGet();*

*}*

*HdfsFileStatus[] children;*

*do {*

*children = listStatus(parent);*

*if (children == null || children.length == 0) {*

*break;*

*}*

*for (HdfsFileStatus child : children) {*

*if (child.isDir()) {*

*this.deque.add(child.getFullName(parent));*

*} else {*

*this.addFileStatus(convertToFileInfo(child, parent));*

*numFilesFetched.incrementAndGet();*

*}*

*}*

*} while (startAfter != null && batches.size() < maxPendingBatches);*

*if (startAfter != null) {*

*pendingParent = parent;*

*}*

*}*

*}*

*}*

FileStatusIngester将文件写入数据库：

*@Override*

*public void run() {*

*//从IngestionTask中获取文件信息*

*FileInfoBatch batch = IngestionTask.pollBatch();*

*try {*

*if (batch != null) {*

*FileInfo[] statuses = batch.getFileInfos();*

*if (statuses.length == batch.actualSize()) {*

*//向数据库中写入文件*

*this.dbAdapter.insertFiles(batch.getFileInfos());*

*IngestionTask.numPersisted.addAndGet(statuses.length);*

*} ......*

# ActionScheduler

5.RulePluginManager