YARN Service Framework源码分析

在YARN中，Service底层复杂的资源管理、调度等逻辑对于用户是透明的，用户只需要通过CLI或者REST API向YARN中提交规范的JSON定义文件，就可以部署和管理运行在YARN上的服务。系统会自动启动、监听容器和其他需要保持应用程序运行的操作，比如自动重启运行识别失败的容器，其执行流程如下图所示：



1. 用户定义JSON格式Service请求，描述服务的规格，如容器内存、CPU核数量、Docker ImageID等
2. 通过YARN CLI或者REST API提交服务
3. RM在接受请求后，启动ApplicationMaster，用于容器编排等功能
4. AM根据用户的资源请求从RM请求资源，然后在分配的NM上启动Container
5. YARN AM会监控容器的健康和执行情况，对容器的故障及异常进行处理。服务生命周期事件和指标写入YARN Timeline Server，同时额外的服务元信息（容器IP及端口）等写入由ZK支持的YARN服务注册表
6. Registry DNS监听Zookeeper中的znode创建和删除，并创建各种DNS记录来提供DNS查询
7. 根据JSON规范和YARN配置中提供的信息，每个容器都会有一个主机名，客户端使用标准的DNS查询容器名从而得到容器IP

# **YARN Service Framework基本概念**

# 服务的定义

服务创建的执行命令：

*yarn app -launch my-sleeper sleeper.json*

服务通过sleeper.json进行定义，定义文件中内容，将其抽象成Service对象，sleeper.json内容如下：

|  |  |
| --- | --- |
| {  *"name": "sleeper-service",*  *"version": "1.0.0",*  *"components" :*  [  {  *"name": "sleeper",*  *"number\_of\_containers": 2,*  *"launch\_command": "sleep 900000",*  *"resource": {*  *"cpus": 1,*  *"memory": "256"*  }  }  ]  } | {  *"name": "sleeper-service",*  *"version": "1.0.0",*  *"components" :*  [  {  *"name": "sleeper",*  *"number\_of\_containers": 2,*  *"artifact":{*  *"id":"hadoop/centos:latest",*  *"type":"DOCKER"*  *}*  *"launch\_command": "sleep 900000",*  *"resource": {*  *"cpus": 1,*  *"memory": "256"*  },  *"retry\_policy":"ON\_FALURE",*  *"configuration":{*  *"env":{*  *"YARN\_CONTAINER\_RUNTIME\_DOCKER\_RUN\_OVERRIDE\_DISABLE":"true"*  *},*  *"properties":{*  *"docker.network":"host"*  *}*  *}*  } ]} |

*Service service =  
 loadAppJsonFromLocalFS(fileName, appName, lifetime, queue); //ApiServiceClient.java*

然后通过REST API发送给RM的Service Framework API进行服务的启动。Service的相关类图如下：



*String buffer = jsonSerDeser.toJson(service);  
ClientResponse response = getApiClient() .post(ClientResponse.class, buffer);*

# **Service Framework剖析**

在YARN Service Framework中ServiceMaster负责服务的统一管理，其启动后与RM交互获取资源，并通过组件编排来启动Container。其同时负责服务注册、组件的监控与组件的扩容、容错等功能。

## ServerMaster内部架构与功能

在YARN Service中，ServerMaster负责所有组件的统一管理和分配，它根据Service定义，生成所有组件及启动顺序。ServerMaster本质上是ApplicationMaster，启动后通过RPC协议向RM注册、申请资源和释放资源；从RM获取资源后，根据组件启动的顺序启动Service的组件；对启动的Service进行监控和管理，并且将事件发送到TimelineV2Server中，以供查询。ServerMaster的内部框架图如下所示：



1. User Service，对外提供ClientRMService，用户可以通过ClientAMProtocol进行Service的管理，比如组件的扩容、重启、停止、升级及查看状态等
2. Component管理模块，主要涉及到：

* ContainerLauchService，通过NMClient与NM交互启动Component
* ServiceMonitor，监控Component的状态

1. 服务注册与监控模块，服务启动后通过YarnRegistryViewForProvider注册到ZK Registry中，其运行流程及指标通过ServiceTimelinePublisher写入Timeline Server
2. ServiceManager，状态机管理模块，使用有限状态机维护Service对象的生命周期
3. 安全管理模块，由ClientAMTokenSecretManager模块来完成

相关类图如下所示：



## **Service事件与事件处理器**

ServiceMaster将Service服务封装成对象和事件处理器，其中封装成对象是为了统一管理，所有的事件服务器处理来自AsyncDispatcher分配的事件，按照事件类型分配给对应的事件处理器。在ServiceScheduler中，将Service管理的对象由对应的状态机进行管理，Service本质上是运行在YARN中的Application，与MapReduce的对比如下表所示：

|  |  |  |
| --- | --- | --- |
| 角色 | YARN Service(ServiceMaster) | MapReduce（MRAppMaster） |
| 应用 | Service | Job |
| Task | Component | Task |
| Task执行 | ComponentInstance | TaskAttempt |

这三种角色在AM中随着程序的执行进行状态的变化。在YARN ServiceMaster包括三种事件调度器，其处理的类型如下图所示：



### Service状态机

在YARN Service Framework中，用数据结果Service来维护服务的生命周期，其内部有Service的状态机，记录Service可能存在的各个状态（ServiceState）以及导致状态见转换的时间（ServiceEventType），当某种类型的事件发生时，会根据实际情况进行状态转移，同时触发一个行为；除了维护状态机之外，还保存了Service基本信息（比如名称、所在队列、启动时间等）和运行信息（当前正在运行的Container数目、状态、quicklinks等）。其状态转换如下图所示：



1. 基本状态（ServiceState），目前有以下状态

* ACCEPTED，初始化状态，提交到RM后Service的状态设置为ACCEPTED，当服务在其他状态下通过客户端进行flex/start/update操作后Service的状态也更新为该状态
* STATED，ServiceMaster在YARN中启动后进入STARTED状态
* STABLE，当Service中所有Component都启动完毕后，Service的状态变为STABLE状态，在该状态下服务对外可用
* FAILED，ServerMaster运行失败所处的状态
* STOPPED，Server的应用被YARN Kill掉后，处于STOPPED
* FLEX，通过客户端触发FLEX进行Component的伸缩，服务的状态变为FLEX
* UPGRADING状态，客户端触发upgrade事件后，Service进行在线升级时更为为该状态。升级完毕后需要进行restart操作来让Service重启

1. 基本事件

* Service状态的变化大部分是通过客户端action进行的改变，改变后调用updateService进行对应的操作，执行如下：

*public Response updateService(@Context HttpServletRequest request,*

*@PathParam(SERVICE\_NAME) String appName, Service updateServiceData) {*

*updateServiceData.setName(appName);*

*if (updateServiceData.getState() != null*

*&& updateServiceData.getState() == ServiceState.FLEX) {*

*return flexService(updateServiceData, ugi);*

*}*

*if (updateServiceData.getState() != null*

*&& updateServiceData.getState() == ServiceState.STOPPED) {*

*return stopService(appName, false, ugi);*

*}....}*

* 服务组件启动Ready后，状态由STARTED转化为STABLE

*boolean isStable = true;*

*for (org.apache.hadoop.yarn.service.api.records.Component comp : scheduler*

*.getApp().getComponents()) {*

*if (comp.getState() !=*

*org.apache.hadoop.yarn.service.api.records.ComponentState.STABLE) {*

*isStable = false;*

*break;*

*}}*

*if (isStable) {*

*scheduler.getApp().setState(ServiceState.STABLE);*

*}*

* 状态机事件，状态机默认状态为State.STABLE，还有一个状态为UPGRADING，该状态机仅对升级事件有作用（是否需要进行扩展到其他事件，如用于Stop后的状态持久化等）

*public enum ServiceEventType {  
 START, //服务的启动  
 UPGRADE, //服务的升级  
 STOP\_UPGRADE //服务的暂停升级  
}*

下图描述了各个事件的来源，主要来自ClientAMService、ApiService及运行的Components



### Component状态机

在Service中，根据服务定义启动多种组件，Component用于维护组件生命周期的数据结构，该该类中维护了Container状态机，其初始化代码如下：

*private void createAllComponents() {*

*long allocateId = 0;*

*Collection<org.apache.hadoop.yarn.service.api.records.Component> sortedComponents =*

*ServiceApiUtil.sortByDependencies(app.getComponents());*

*for (org.apache.hadoop.yarn.service.api.records.Component compSpec : sortedComponents) {*

*Component component = new Component(compSpec, allocateId, context);*

*componentsById.put(allocateId, component);*

*componentsByName.put(component.getName(), component);*

*allocateId++;*

*if (!hasAtLeastOnePlacementConstraint*

*&& compSpec.getPlacementPolicy() != null*

*&& compSpec.getPlacementPolicy().getConstraints() != null*

*&& !compSpec.getPlacementPolicy().getConstraints().isEmpty()) {*

*hasAtLeastOnePlacementConstraint = true;*

*}*

*}*

*}*

在Component中维护了基本信息，包括组件的描述、优先级、id号等，还维护了运行时组件实例信息，包括正在运行的实例、等待运行的实例、组件probe(健康监控)等。其状态转换如下图所示：



1）基本状态，状态类型为ComponentState

*public enum ComponentState {  
 INIT, //初始化  
 FLEXING, //正在进行Flex  
 STABLE, //稳定状态  
 UPGRADING //升级中  
}*

2）基本事件，事件类型为ComponentEventType，包括以下几种：

*public enum ComponentEventType {*

*FLEX, //对组件进行Flex操作，起始及客户端进行Flex操作*

*CONTAINER\_ALLOCATED, //RM分配一个Container*

*CONTAINER\_RECOVERED, //AM启动过程中，恢复Container*

*CONTAINER\_STARTED, //NM启动Container*

*CONTAINER\_COMPLETED, //在Flex过程中，启动Container失败*

*UPGRADE, //客户端进行升级*

*STOP\_UPGRADE //停止升级*

*}*

下图描述了各个事件的来源，主要来自ClientAMService、ApiService及运行的Components



### ComponentInstance状态机

对于每个Component中都可以启动多个ComponentInstance，在Service中对应组件的启动个数，其数据结构主要包括Container描述信息（启动时间、主机、使用资源、artifact等）及运行信息，其状态机转换如下图所示：



1）基本状态

*public enum ComponentInstanceState {*

*INIT, //初始状态，ServiceMaster启动后进行初始化状态*

*STARTED, // RM分配资源后，由AM与NM交互启动Container*

*READY, //Container启动后，在其中运行服务组件完成后对外提供服务*

*UPGRADING //目前还不支持，有Component UPGRADING来完成升级*

*}*

2）基本事件

*public enum ComponentInstanceEventType {  
 START, //启动  
 STOP, //停止  
 BECOME\_READY, //组件实例Ready  
 BECOME\_NOT\_READY //组件实例非ready  
}*

状态机事件源如下所示：



## **用户交互模块**

在ServiceMaster中，ClientAMService负责处理来自普通用户的请求，其是一个Rpc Server，实现了ClientAMProtocol协议的服务，其定义如下：

*public class ClientAMService extends AbstractService*

*implements ClientAMProtocol {*

*private ServiceContext context;*

*private Server server;*

*......*

*@Override protected void serviceStart() throws Exception {*

*Configuration conf = getConfig();*

*YarnRPC rpc = YarnRPC.create(conf);*

*InetSocketAddress address = new InetSocketAddress(0);*

*server = rpc.getServer(ClientAMProtocol.class, this, address, conf,*

*context.secretManager, 1);*

*...}}*

在ClientAMService类中保留了ServiceMaster上下文对象ServiceContext，通过该对象可以获知ServiceMaster中绝大部分信息，包括服务定义、启动组件信息等。ClientRMService可以很容器通过查询ClientAMService中的信息为客户端的请求作出应答。

目前支持的操作定义在ClientAMProtocol中，其实现的接口如下：

*public interface ClientAMProtocol {*

*FlexComponentsResponseProto flexComponents(FlexComponentsRequestProto request)*

*throws IOException, YarnException;*

*GetStatusResponseProto getStatus(GetStatusRequestProto requestProto)*

*throws IOException, YarnException;*

*StopResponseProto stop(StopRequestProto requestProto)*

*throws IOException, YarnException;*

*UpgradeServiceResponseProto upgrade(UpgradeServiceRequestProto request)*

*throws IOException, YarnException;*

*RestartServiceResponseProto restart(RestartServiceRequestProto request)*

*throws IOException, YarnException;}*

# **Service基本执行流程**

在YARN Service Framework中，Service由ServiceMaster和多Component组成，其中ServiceMaster管理整个Service的生命周期，包括组件的编排、部署及监控。而Component负责Service的具体执行，每个Component都可以包括多个Component Instance。下面着重介绍这几个组成部分的执行流程。

## **ServiceMaster的启动**

通过命令行创建Service：

*yarn app -launch my-sleeper sleeper.json*

客户端提交创建任务的执行如下图所示：



通过Jersey Client向RM发送服务创建请求，其处理类为ApiServer，创建服务代码为:

*ServiceClient sc = getServiceClient();  
sc.init(YARN\_CONFIG);  
sc.start();  
ApplicationId applicationId = sc.actionCreate(service);*

调用submitApp，核心代码如下：

*ApplicationId submitApp(Service app) throws IOException, YarnException {*

*//从Service对象中获取service name*

*String serviceName = app.getName();*

*......*

*YarnClientApplication yarnApp = yarnClient.createApplication();*

*//初始化App SubmissionContext*

*ApplicationSubmissionContext submissionContext =*

*yarnApp.getApplicationSubmissionContext();*

*......*

*//生成AM启动命令行*

*String cmdStr = buildCommandLine(app, conf, appRootDir, hasAMLog4j);*

*submissionContext.setResource(Resource.newInstance(YarnServiceConf*

*.getLong(YarnServiceConf.AM\_RESOURCE\_MEM,*

*YarnServiceConf.DEFAULT\_KEY\_AM\_RESOURCE\_MEM, app.getConfiguration(),conf), 1));*

*......*

*//设置Application Tags*

*Set<String> appTags =*

*AbstractClientProvider.createApplicationTags(serviceName, null, null);*

*if (!appTags.isEmpty()) {*

*submissionContext.setApplicationTags(appTags);*

*}*

*//AM Container启动Launch Context*

*ContainerLaunchContext amLaunchContext =*

*Records.newRecord(ContainerLaunchContext.class);*

*amLaunchContext.setCommands(Collections.singletonList(cmdStr));*

*amLaunchContext.setEnvironment(env);*

*amLaunchContext.setLocalResources(localResources);*

*addHdfsDelegationTokenIfSecure(amLaunchContext);*

*submissionContext.setAMContainerSpec(amLaunchContext);*

*//提交执行*

*yarnClient.submitApplication(submissionContext);*

*return submissionContext.getApplicationId();*

*}*

其核心数据是服务定义组件my-sleeper.json和zk定义。 my-sleeper.json在启动之前写到HDFS中，如下：

*ServiceApiUtil.createDirAndPersistApp(fs, appDir, service); //保存到HDFS路径中*

*FSDataOutputStream dataOutputStream = fs.create(path, overwrite)*

*writeJsonAsBytes(instance, dataOutputStream);*

至此可以启动ServiceMaster。

## **Component的生命周期**

ServiceMaster启动并注册到RM后，向RM请求资源，并根据Component定义启动组件，核心组件是ServiceScheduler，其流程如下图所示：



1. Service应用提交后，首先启动ServiceMaster，启动过程中，根据配置文件，生成要启动的组件信息，

*private void createAllComponents() {*

*long allocateId = 0;*

*Collection<org.apache.hadoop.yarn.service.api.records.Component> sortedComponents =*

*ServiceApiUtil.sortByDependencies(app.getComponents());*

*for (org.apache.hadoop.yarn.service.api.records.Component compSpec : sortedComponents){*

*Component component = new Component(compSpec, allocateId, context);*

*componentsById.put(allocateId, component);*

*componentsByName.put(component.getName(), component);*

*allocateId++;*

*if (!hasAtLeastOnePlacementConstraint*

*&& compSpec.getPlacementPolicy() != null*

*&& compSpec.getPlacementPolicy().getConstraints() != null*

*&& !compSpec.getPlacementPolicy().getConstraints().isEmpty()) {*

*hasAtLeastOnePlacementConstraint = true;*

*}*

*}*

*}*

ServiceMaster向RM申请资源后，将其交由Component进行处理，通过allocateId来区分资源用于启动哪类Component

1. 资源分配后，Component根据状态机的转换，调用ContainerLaunchService向NM发送Container请求，启动ComponentInstance

*scheduler.getContainerLaunchService()  
 .launchCompInstance(scheduler.getApp(), instance, container);*

对于Component来说，可以有多个ComponentInstance，在配置中通过numOfContainers来指定

1. Container启动后，通过状态机改变Component的状态，

*private static ComponentState checkIfStable(Component component) {  
 if (component.componentMetrics.containersReady.value() == component  
 .getComponentSpec().getNumberOfContainers()) {  
 component.componentSpec.setState(  
 org.apache.hadoop.yarn.service.api.records.ComponentState.STABLE);  
 return STABLE;  
 } else {  
 component.componentSpec.setState(  
 org.apache.hadoop.yarn.service.api.records.ComponentState.FLEXING);  
 return FLEXING;  
 }*

当Component启动足够的ComponentInstance后，状态变为STABLE，可以对外提供服务。

1. 在启动过程中，将Component最新信息注册到ZK中，在ContainerStartedTransition中调用，

*ComponentStatusRetriever#*

*public void updateContainerStatus(ContainerStatus status) {*

*this.status = status;*

*org.apache.hadoop.yarn.service.api.records.Container container =*

*getCompSpec().getContainer(status.getContainerId().toString());*

*if (container != null) {*

*container.setIp(StringUtils.join(",", status.getIPs()));*

*container.setHostname(status.getHost());*

*if (timelineServiceEnabled) { //Timeline开始时，写入该事件*

*serviceTimelinePublisher.componentInstanceIPHostUpdated(container);*

*}*

*}*

*updateServiceRecord(yarnRegistryOperations, status);*

*}*

更新中的状态是从NM中直接获取

*status = nmClient.getContainerStatus(containerId, nodeId);*

## **Component Instance的启动**

ServiceScheduler向NM提交Container启动请求，其入口如下：

*scheduler.getContainerLaunchService()  
 .launchCompInstance(scheduler.getApp(), instance, container);*

在执行过程中涉及到的类如下：



### 4.3.1 ContainerLaunchContext

NM根据ContainerLaunchContext启动Container，对于Service Framework，其输入源来自于以下几个部分：

1. **Service**，其描述了定义文件，其中指定了配置、quicklinks、资源、artifact等
2. **ConstanceIntance，**其中维护了ServiceContext、tokens及所属的Component定义文件
3. **Container，**分配的资源，启动的NM、containerId、优先级、资源、ExecutionType等

通过ProvderService#buildContainerLauncContext生成ContainerLaunchContext配置信息：

* 处理Service依赖的Artifact，例如Tarball、Docker镜像等

Tarball，将ARCHIVE添加到LocalResource中

Docker，设置Docker Imge/网络/Domain等

* TOKEN包括以下信息 //配置环境变量
* Global Token

*CLUSTER\_ZK\_QUORUM*

*SERVICE\_ZK\_PATH*

*USER*

*KEY\_DNS\_DOMAIN*

*DOMAIN*

*FS\_DEFAULT\_NAM\_KEY*

*CLUSTER\_FS\_URI*

*CLUSTER\_FS\_HOST*

*SERVICE\_HDFS\_DIR*

*SERVICE\_NAME\_LC*

*SERVICE\_NAME*

* Component Token

*CONTAINER\_NAME*

*CONTAINER\_ID*

*COMPONENT\_INSTANCE\_NAME*

*COMPONENT\_NAME\_LC*

*依赖COMPONENT的ip及host，对应COMPONENT\_INSTANCE\_IP*

*COMPONENT\_INSTANCE\_HOST*

这些Token会以ENV的形式添加到LaunchContext中。

* Component中的Configuration，根据定义文件中的配置
* WORK\_DIR，LOG\_DIR，根据当前启动的路径进行配置
* 常量，LANG,LC\_ALL,LANGUAGE

大部分环境变量在ApplicationConstants中定义如下：

*MALLOC\_ARENA\_MAX*

*HADOOP\_HDFS\_HOME*

*HADOOP\_CONF\_DIR*

*CONTAINER\_ID*

*NM\_HOST*

*LOCAL\_DIRS*

*LOG\_DIRS*

*....*

这些环境变量会在launch\_container.sh中以export的形式添加到环境变量。

* 在HDFS中创建组件使用的配置文件
* compInstanceDir，组件对应的文件目录，格式为: *{app}/{comp}/{instance}*
* 配置文件，两种类型：HADOOP\_XML和TEMPLATE，将第一步的配置替换配置文件中的信息，配置在定义文件中的Configuration中，属性名为:files
* 将配置文件放到HDFS对应的组件目录
* 将组件实例需要的资源进行本地化
* 将上一步生成的配置文件下载到本地，*{conf}/{filename}*
* 生成组件启动命令
* 定义文件中的启动命令, launch\_command
* 将launch\_command中的转义字符用token中配置进行替代
* 添加OUT\_FILE,ERR\_FILE
* 设置retry周期及次数

*CONTAINER\_RETRY\_MAX*

*CONTAINER\_RETRY\_INTERVAL*

*CONTAINER\_FAILURE\_VALIDITY\_INTERVAL*

以上是ContainerLaunchContext的全部信息，通过NMClient提交给NM

*instance.getComponent().getScheduler().getNmClient()*

*.startContainerAsync(container, launcher.completeContainerLaunch());*

#completeConainerLaunch的执行是进一步对ContainerLaunchContext进行处理：

* 将commands进行join连接成一个Command
* 设置environments
* 设置serviceData
* 设置localResource
* 设置tokens，执行过程中去掉重复的参数
* 如果是yarn docker，设置:

*YARN\_CONTAINER\_RUNTIME*

*YARN\_CONTAINER\_RUNTIME\_DOCKER\_IMAGE*

*YARN\_CONTAINER\_RUNTIME\_DOCKER\_CONTAINER\_NETWORK*

*YARN\_CONTAINER\_RUNTIME\_DOCKER\_RUN\_PRIVILEGED\_CONTAINER*

*YARN\_CONTAINER\_RUNTIME\_DOCKER\_LOCAL\_RESOURCE\_MOUNTS*

### 4.3.2 Container的启动

NodeManager通过ContainerManagerImpl负责接收并处理该请求，启动流程如下：

1. 资源本地化



根据ContainerLaunchContext的分析，LocalResource资源，主要包括Artifact和Config File，以TarballProviderService为例：

Config File

*for (ConfigFile originalFile : component.getConfiguration().getFiles()) {*

*......*

*LocalResource configResource =  
 fs.createAmResource(remoteFile, LocalResourceType.FILE);*

*launcher.addLocalResource(symlink, configResource,configFile.getDestFile());*

*}*

Artifact:

*Path artifact = new Path(instance.getCompSpec().getArtifact().getId());  
LocalResourceType type = LocalResourceType.ARCHIVE;  
LocalResource packageResource = fileSystem.createAmResource(artifact, type);  
launcher.addLocalResource(APP\_LIB\_DIR, packageResource);*

资源本地化的具体执行类为ContainerExecutor#startLocalizer。其定义文件示例为：

*"artifact" : {*

*"id" : ".yarn/package/LLAP/llap-14Sep2018.tar.gz",*

*"type" : "TARBALL"*

*}*

1. Container的运行

Container的运行由ContainerLauncer服务实现，主要过程是将待运行的Container的环境变量和运行命令写到脚本launch\_container.sh中，并将启动脚本的命令写入到:

*default\_container\_executor.sh*

然后通过该脚本启动Container，其运行过程如下图所示：



ContainerLaunch在执行时生成token及launch\_container.sh启动脚本，

*try (DataOutputStream containerScriptOutStream =  
 lfs.create(nmPrivateContainerScriptPath,EnumSet.of(CREATE, OVERWRITE))) {  
 // 设置环境变量  
 sanitizeEnv(environment, containerWorkDir, appDirs, userLocalDirs,  
 containerLogDirs, localResources, nmPrivateClasspathJarDir,  
 nmEnvVars);*

*//设置ContainerExecutor启动信息：LOCALIZED\_RESOURCE,USER,LOG\_DIRS*

*CONTAINER\_RUN\_CMDS等，在执行过程中调用linuxContainerRuntime.prepraContainer  
 prepareContainer(localResources, containerLocalDirs);  
   
 // 输出环境变量  
 exec.writeLaunchEnv(containerScriptOutStream, environment,  
 localResources, launchContext.getCommands(),  
 containerLogDir, user, nmEnvVars);  
}*

ContainerExecutor将launch\_container.sh和tokens文件复制到工作目录，并重新构造新的default\_container\_executor.sh，启动该脚本，源码如下：

*Path launchDst =  
 new Path(containerWorkDir, ContainerLaunch.CONTAINER\_SCRIPT);  
copyFile(nmPrivateContainerScriptPath, launchDst, user);*

*Shell.CommandExecutor shExec = null;  
try {  
 setScriptExecutable(launchDst, user);  
 setScriptExecutable(sb.getWrapperScriptPath(), user);  
 shExec = buildCommandExecutor(sb.getWrapperScriptPath().toString(),  
 containerIdStr, user, pidFile, container.getResource(),  
 new File(containerWorkDir.toUri().getPath()),  
 container.getLaunchContext().getEnvironment());  
 if (isContainerActive(containerId)) {  
 shExec.execute();  
 }*

至此完成了Component Container的启动。

### 4.3.3 LinuxContainerExecutor

目前在YARN中最常用的是LinuxContainerExecutor，核心包括两个方面：

1. **ResourceHandler**，用于实现资源隔离，在LCE中初始化如下：

*resourceHandlerChain = ResourceHandlerModule  
 .getConfiguredResourceHandlerChain(conf, nmContext);*

在LCE#handleLaunchForLaunchType中资源的隔离调用源码如下：

*resourcesHandler.preExecute(containerId, container.getResource());*

*List<PrivilegedOperation> ops = resourceHandlerChain.preStart(container);*

当前支持Network/Disk/Memory/CPU/Numa/五种ResourceHandler，在ResourcePluginManager中还提供了GPU\_URI和FPGA两种资源的ResourcePlugin，其类图如下所示：



目前支持7种资源的隔离，分成两种类型：

1. 基于CGoups，通过CGroupsHandler来控制，包括网络、cpu、内存、磁盘四种资源，在CGoups中挂载资源使用配置来实现，以内存为例，其核心代码：

*long containerHardLimit = container.getResource().getMemorySize();*

*cGroupsHandler.updateCGroupParam(MEMORY, cgroupId,*

*CGroupsHandler.CGROUP\_PARAM\_MEMORY\_HARD\_LIMIT\_BYTES,*

*String.valueOf(containerHardLimit) + "M");*

基于CGroup的资源隔离使用如下表所示(参数前缀yarn.nodemanager.resource)：

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离资源 | 启用 | 参数 | 使用 |
| CPU | cpu.enabled | cpu-vcores //配置节点虚拟核数  pcores-vcores-multiplier //虚拟核与物理核的比例  cgroups.strict-resource-usage //严格的cpu使用  count-logical-processors-as-cores  //逻辑核是否作为物理核 | cpu.vcores  cpus |
| Memory | memory.enabled | memory //配置节点可使用内存  memory.enforced //严格的内存控制  soft-limit-percentage //宽松的内存使用量比例  swappiness //缓存使用量  vmem-check-enabled //检测vmem使用量  pmem-check-enabled //检测pmem使用量  vmem-pmeb-ratio //虚拟内存与实际内存的比率 | memory-mb  memory |
| DISK | disk.enabled | DEFAULT\_WEIGHT: 500 | 目前不完善 |
| 网络 | network.enabled | network.interface //配置网卡，默认eth0  outbound-bandwith-mbit //网卡写带宽：1000  strict-resource-usage //是否strict  outbound-bandwith-yarn-mbit  //yarn container可以使用的总带宽 | 与container的个数，没有具体配置 |

目前对于Disk和网络资源，不可配置具体使用量

1. 通过ResourceAllocator进行管理，包括GPU/FPAG，以GPU为例

*private Set<GpuDevice> allowedGpuDevices = new TreeSet<>();  
private Map<GpuDevice, ContainerId> usedDevices = new TreeMap<>();*

GpuResourceAllocator使用如上的GPU资源表示形式来进行管理。为请求分配资源如下：

*public synchronized GpuAllocation assignGpus(Container container)*

*throws ResourceHandlerException {*

*Resource requestedResource = container.getResource();*

*ContainerId containerId = container.getContainerId();*

*int numRequestedGpuDevices = getRequestedGpus(requestedResource);*

*// Assign Gpus to container if requested some.*

*if (numRequestedGpuDevices > 0) {*

*....*

*Set<GpuDevice> assignedGpus = new TreeSet<>();*

*for (GpuDevice gpu : allowedGpuDevices) {*

*if (!usedDevices.containsKey(gpu)) {*

*usedDevices.put(gpu, containerId);*

*assignedGpus.add(gpu);*

*if (assignedGpus.size() == numRequestedGpuDevices) {*

*break;*

*}*

*}*

*}*

*.....*

*return new GpuAllocation(assignedGpus,*

*Sets.difference(allowedGpuDevices, assignedGpus));*

*}*

*return new GpuAllocation(null, allowedGpuDevices);*

*}*

在Container中的配置为

*return Long.valueOf(requestedResource.getResourceValue(GPU\_URI)).intValue();*

在Resource中配置通过ResourceInforamation来配置，对于GPU使用yarn.ip/gpu来指定

目前仅支持GPU和FPAG，在resource-plugins中配置参数：

*<property>  
 <name>yarn.nodemanager.resource-plugins</name>  
 <value>yarn-io/gpu,yar-io/fpga</value>  
</property>*

|  |  |  |
| --- | --- | --- |
| 隔离资源 | 参数 | 使用 |
| gpu | gpu.allowed-gpu-devices  gpu.docker-plugin <= nvidia-docker-v1  nvidia-docker-v1.endpoint | yarn.io/gpu |
| fgpa | allowed-fpga-devices  fpga.vendor-plugin.class  fpga.path-to-discovery-executables | yarn.io/fgpa |

1. 其他NumaNodeResource

当配置numa.awareness-enabled为true时，启用NumaResourceHandlerImpl

|  |  |  |
| --- | --- | --- |
| 隔离资源 | 参数 | 使用 |
| numa | read-topology  node-ids  numactl.cmd  /usr/bin/numactl |  |

通过NumaResourceAllocator为Container分配资源

*private NumaResourceAllocation allocate(ContainerId containerId, Resource resource) {  
 for (int index = 0; index < numaNodesList.size(); index++) {  
 NumaNodeResource numaNode = numaNodesList  
 .get((currentAssignNode + index) % numaNodesList.size());  
 if (numaNode.isResourcesAvailable(resource)) {  
 numaNode.assignResources(resource, containerId);  
 currentAssignNode = (currentAssignNode + index + 1)  
 % numaNodesList.size();  
 return new NumaResourceAllocation(numaNode.getNodeId(),  
 resource.getMemorySize(), numaNode.getNodeId(),  
 resource.getVirtualCores());  
 }  
 }*

在Container的启动命令中添加numa的使用参数

*ArrayList<String> args = new ArrayList<>();  
args.add(numaCtlCmd);  
args.add( "--interleave=" + String.join(",", numaAllocation.getMemNodes()));  
args.add("--cpunodebind=" + String.join(",", numaAllocation.getCpuNodes()));  
ret.add(new PrivilegedOperation(OperationType.ADD\_NUMA\_PARAMS, args));*

Numa、GPU及FPGA的使用方式类似，都是命令中添加：

*resources\_options*

参数列表如下：

*--module-gpu*

*--module-fpga*

目前还不支持Numa

1. 资源auto-detection，通过配置下面的参数：

*<property>  
 <name>yarn.nodemanager.resource.detect-hardware-capabilities</name>  
 <value>true</value>  
</property>*

自动检测节点CPU与Mem，当enable后，直接获取主机上可用的资源

* 节点可用内存

*public static long getContainerMemoryMB(Configuration conf) {*

*if (!isHardwareDetectionEnabled(conf)) {*

*return getConfiguredMemoryMB(conf); //默认配置参数：resource.memory-mb*

*}*

*ResourceCalculatorPlugin plugin = //从系统中通过SysInfo获取可用资源*

*ResourceCalculatorPlugin.getResourceCalculatorPlugin(null, conf);*

*if (plugin == null) {*

*return getConfiguredMemoryMB(conf);*

*}*

*return getContainerMemoryMBInternal(plugin, conf); }*

*//配置为-1，则计算公式为：*

*0.8f\*(physicalMemoryMB-2\*hadoopHeapSizeMB)*

*PhysicalMemoryMB <= SysInfo.getPhysicalMemorySize()*

*hadoopHeapSizeMB<=Runtime.getRuntime().maxMemory*

*physicalMemroyMB- reservedMemoryMB //配置system-reserved-memory-mb*

* 节点可用CPU

*public static int getVCores(Configuration conf) {*

*if (!isHardwareDetectionEnabled(conf)) {*

*return getConfiguredVCores(conf); //resource.vcores，默认8*

*}*

*ResourceCalculatorPlugin plugin =*

*ResourceCalculatorPlugin.getResourceCalculatorPlugin(null, conf);*

*if (plugin == null) {*

*return getConfiguredVCores(conf);*

*}*

*return getVCoresInternal(plugin, conf); } //计算公式如下： count-logical-processors-as-cores为true*

*=> percentage-physical-cpu-limit\*SysInfo.getNumProcessors //逻辑核数*

*false*

*=>percentage-physical-cpu-limit\*SysInfo.getNumCores*

1. LinuxContainerRuntime

目前在YARN中支持多种Container类型的启动，其通过ContainerRuntime来实现

*public enum RuntimeType {  
 DEFAULT,  
 DOCKER,  
 JAVASANDBOX;  
}*

* ContainerExecutor中的执行如下：

*private int handleLaunchForLaunchType(ContainerStartContext ctx,*

*ApplicationConstants.ContainerLaunchType type) throws IOException,*

*ConfigurationException {*

*......*

*ContainerRuntimeContext runtimeContext = buildContainerRuntimeContext(*

*ctx, pidFilePath, resourcesOptions, tcCommandFile, numaArgs);*

*if (type.equals(ApplicationConstants.ContainerLaunchType.RELAUNCH)) {*

*linuxContainerRuntime.relaunchContainer(runtimeContext);*

*} else {*

*linuxContainerRuntime.launchContainer(runtimeContext);*

*}*

*}*

* ContainerRuntimeContext中包含以下信息： ContainerExecutor的输入信息

*private ContainerRuntimeContext buildContainerRuntimeContext(*

*ContainerStartContext ctx, Path pidFilePath, String resourcesOptions,*

*String tcCommandFile, List<String> numaArgs) {*

*....*

*}*

包含的属性如下表所示：

|  |  |
| --- | --- |
| 属性 | 描述 |
| numaArgs | *--interleave --cpunodebind* |
| CONTAINER\_LAUNCH\_PREFIX\_COMMANS | */usr/bin/numactl* |
| LOCALIZED\_RESOURCES | *本地资源* |
| RUN\_AS\_USER | *执行用户：USER或者nobody* |
| USER | *程序启动用户* |
| APPID | *程序id* |
| CONTAINER\_ID\_STR | *Container的ID* |
| NM\_PRIVATE\_CONTAINER\_SCRIPT\_PATH | *程序的启动脚本，参数来自于ContaineLaunchContext及环境变量* |
| NM\_PRIVATE\_TOKEN\_PATH | *Token路径* |
| PID\_FILE\_PATH | *pid文件* |
| LOCAL\_DIRS | *执行本地目录* |
| LOG\_DIRS | *日志目录* |
| FILECACHE\_DIRS | *缓存目录* |
| USER\_LOCAL\_DIRS | *用户本地目录* |
| CONTAINER\_LOCAL\_DIRS | *Container执行目录* |
| USER\_CACHE\_DIRS | *用户缓存目录* |
| APPLICATION\_LOCAL\_DIRS | *程序本地目录* |
| CONTAINER\_LOG\_DIRS | *Container执行log目录* |
| RESOURCE\_OPTIONS | *资源的选项，例如gpu:*  *--module-gpu [num]* |

* linuxContainerRuntime，根据CONTAINER的类型选择Runtime，类图如下所示：



执行的Runtime根据Environment中的参数来决定，DelegationLinuxContainerRuntime#

pickContainerRuntime <=  *Container.getLaunContext.getEnvironment*

*LinuxContainerRuntime pickContainerRuntime(  
 Map<String, String> environment) throws ContainerExecutionException {  
 LinuxContainerRuntime runtime;  
 //Sandbox checked first to ensure DockerRuntime doesn't circumvent controls  
 if (javaSandboxLinuxContainerRuntime != null &  
 javaSandboxLinuxContainerRuntime.isSandboxContainerRequested()){   
 runtime = javaSandboxLinuxContainerRuntime;  
 } else if (dockerLinuxContainerRuntime != null &&  
 DockerLinuxContainerRuntime.isDockerContainerRequested(environment)){  
 runtime = dockerLinuxContainerRuntime;  
 } else if (defaultLinuxContainerRuntime != null &&  
 !DockerLinuxContainerRuntime.isDockerContainerRequested(environment)) {  
 runtime = defaultLinuxContainerRuntime;  
 } ...  
 return runtime;  
}*

|  |  |
| --- | --- |
| 参数 | ContainerRuntime类型 |
| runtime.linux.sandbox-mod <=  enforcing,permissive | JavaSandboxLinuxContainerRuntime |
| YARN\_CONTAINER\_RUNTIME\_TYPE  <= docker | DockerLinuxContainerRuntime |
| others | DefaultLinuxContainerRuntime |

* PrivilegedOperationExecutor，具体的执行命令

*String[] fullCommandArray = getPrivilegedOperationExecutionCommand  
 (prefixCommands, operation);  
ShellCommandExecutor exec = new ShellCommandExecutor(fullCommandArray,  
 workingDir, env, 0L, inheritParentEnv);  
exec.execute();*

其内封装了Shell#run来执行

# **YARN Service API**

# **高级特性**

1. 服务的注册与查找
2. 服务的监控
3. 下一步重点工作

yarn.service.framework.path

Service service <= ApiServiceClient#loadAppJsonFromLocalFs

#getApiUrl

#rm.webapp.address

#/app/v1/services

=> RestApiConstants

