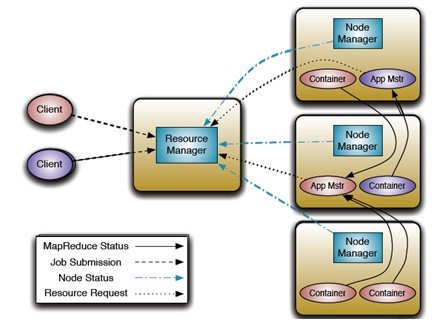
Hadoop Yarn

# 一、简介与总体框架

Hadoop为了解决原MapReduce框架的不足引入Yarn(分布式资源管理系统)。原MapReduce框架，JobTracker是集群事务的集中处理点，存在单点故障，而且JobTracker需要完成的任务太多（维护job及task的状态）造成过多资源消耗。在TaskTracker端，Map/Reduce task作为资源的表示太过简单，没有考虑cpu、内存等资源情况，当把两个需要消耗大内存的task调度到一起，很容易出现内存不足的情况。在资源管理方面，将资源强制划分为map/reduce slot，当只有map task时，reduce slot不能用，只有reduce task时，map slot不能用，容易造成资源利用不足。

Yarn/MR2最基本的思想是将原JobTracker主要的资源管理和job调度/监视功能分开作为两个单独的守护进程，框架如下图所示：



Yarn框架

JobTacker的解决方案中包含两个组件，ResourceManager（RM）和每个应用相关的ApplicationMaster（AM），应用是指一个单独的MapReduce或者DAG作业。RM和NodeManager（NM，每个节点一个）共同组成整个数据计算框架。RM是系统中将资源分配给各个应用的最终决策这，AM实际上是一个具体的框架库，它的任务时与RM协商获取应用所需资源及与NM合作完成执行和监控task的任务。

下面是Yarn的资源调度相关参数

表格1：Yarn资源调度及分配策略

|  |  |  |
| --- | --- | --- |
| 属性名 | 含义 | 默认值 |
| yarn.resourcemanager.scheduler.class | RM使用的资源调度器类（ResourceScheduler），常用的有  CapacityScheduler和FairScheduler，可以根据需要进行定制 | FaiScheduler |
| yarn.scheduler.fair.allocation.file | 用于配置资源调度器，包括队列及配置的资源等 | fair-scheduler.xml |
| yarn.nodemanager.container-executor.class | LCE，用于创建及发布containers，其资源处理机制，为容器绑定特定的cpu和memory节点，分配特定比例的cpu、io时间及限制内存等，并提供独立的namespace。 | DefaultContainerExcutor(一般配置LinuxContainerExuecutor) |
| yarn.nodemanager.  linux-container-excutor.resources-handler.class | 用于协助LinuxContainerExecutor处理资源，当前只有CgroupsLCEResourcesHandler起到作用，其修改Cgroup的配置文件实现创建Cgroup及更新资源配置 | DefaultLCEResourcesHandler(空方法) |
| yarn.nodemanager.linux-container-executor.cgroups.hierarchy | Cgroups以层级形式来管理资源，通过这个参数配置LCE使用的层级，在其中配置资源的限制，当前仅支持Cgroups的cpu管理，以后可能会支持的多些。前提是使用Cgroups  LCEResourceHandler | /hadoop-yarn，（挂载到/cgroup/cpu） |
| yarn.nodemanager.linux-container-executor.cgroups.mout-path | Yarn的LCE mount路径 | 无（配置cgroup的安装路径），Linux 2.6以上版本自带Cgroup，安装路径/cgroup |
| yarn.nodemanager.linux-container-exectuor.cgroups.mount | 在找不到cgroups的情况下，LCE是否尝试挂载cgroup，前提是使用  CgroupsLCEResourceHandler | false |

表格2：资源分配限制

|  |  |  |
| --- | --- | --- |
| 属性名 | 含义 | 默认值 |
| yarn.nodemanager.resource.cpu-vcores | 该参数用于配置CPU虚拟出的虚拟核数目。Yarn允许Admin根据实际需要和Cpu性能设置虚拟cpu个数，个数越多粒度越细 | 8 |
| yarn.scheduler.maximun-allocation-vcores | 每个Container可以分配的最多Vcores数目 | 32 |
| yarn.scheduler.minimum-allocation-vcores | 每个Container分配的最少vcores数目 | 1 |
| Yarn.nodemanager.resouce.memory-mb | 该节点上YARN可使用的物理内存总量（如果节点没有这么多的内存，则需要调小这个值，Yarn不会探测节点内存总量） | 8192 |
| yarn.nodemanager.vmen-pmem-ration | 任务每使用1MB物理内存，最多可使用虚拟内存量 | 2.1 |
| yarn.nodemanager.maximum-allocation-mb | 单个任务可以使用的最大物理内存 | 8192 |
| yarn.nodemanager.minmum-allocation-mb | 单个任务可申请的最小物理内存 | 1024 |
| Yarn.scheduler.fair.max.assign | 如果开启批量分配功能，可指定分配的container数目 | -1，表示不限制 |

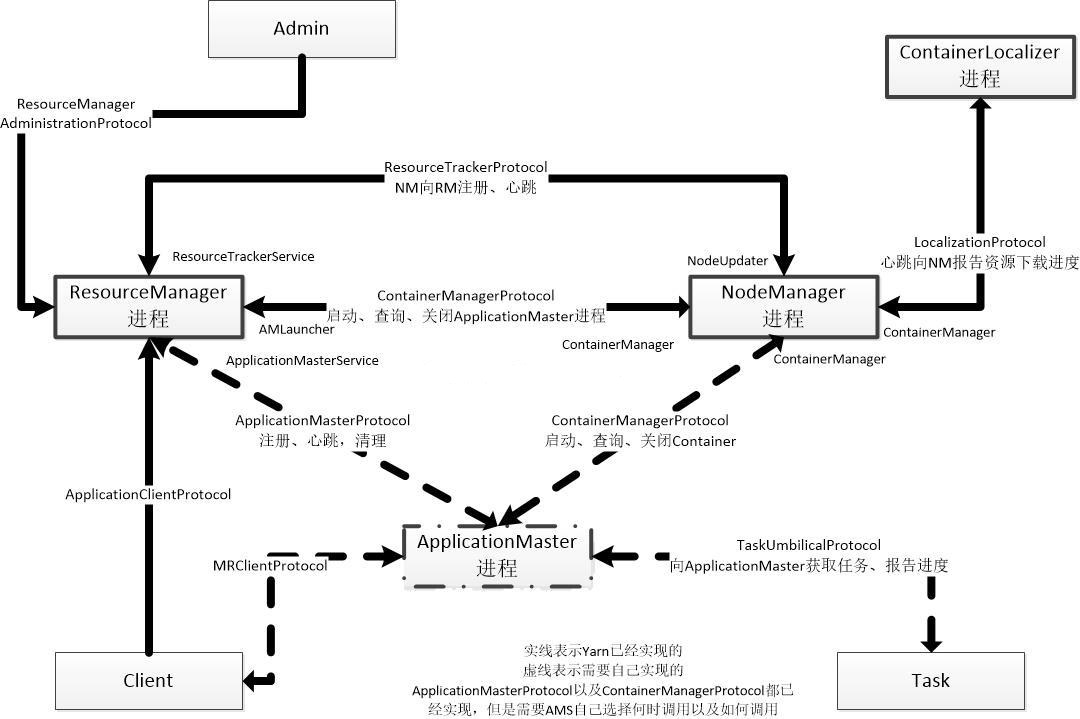
表格3：资源控制是否启动配置

|  |  |  |
| --- | --- | --- |
| yarn.scheduler.fair.user-as-default-queue | 当应用程序未指定队列名时，是否指定用户名作为应用程序所在的队列名，如果false或者未设置，所有位置队列的应用程序提交到default队列中 | True |
| yarn.scheduler.fair.preemption | 是否启用抢占机制，默认值 | false |
| yarn.scheduler.assignmultiple | 是否启用批量分配功能，当一个节点出现大量资源时，可以一次分配完成，也可多次分配完成 | false |
| yarn.nodemanager.pmem-check-enabled | 启动一个线程检查每个任务正使用的物理内存量，如果超过分配值，则直接将其kill掉 | true |
| yarn.nodemanager.vmem-check-enabled | 启动一个线程检查每个任务正使用的虚拟内存量，如果超过分配值，则直接将其kill掉 | true |
| aclSubmitApps | 可向队列中提交应用程序的Linux用户或用户组列表 | \*，任何用户 |
| aclAdministerApps | 该队列的管理员列表，一个队列的管理员可管理该队列中资源和应用程序 |  |

# 二、Yarn组件

在Yarn中，Client向ResourceManager提交应用，每个应用都会启动一个ApplicationMa

ster。AM经过ResourceManager分配资源后，运行在某个Slave节点（NodeManager）的container上，执行计算的Task，也运行在Slave节点的container中。RM,NM,AM及Container之间的通信，都是通过RPC机制来完成的，详细框架如下图所示：



Yarn详细框架

1）通过sbin/yarn命令管理Yarn集群，Yarn Amin对应的默认端口是8033，在RM中对应的类为AdminService等服务，其实现接口RMAdministrationProtocol。Client，可以Yarn提交应用或者获取集群等信息，可以通过Yarn命令，但是实现的接口是AdminClientProtocol，对应的端口为8032等。Client及Admin对Yarn的访问及管理，都是与RM通信来完成。

2）ResourceManager(RM)负责管理调度资源。每个应用的ApplicationMaster与ResourceManager协商资源，并与NodeManager协同工作来执行和管理Task。ResourceManager内部有一个可插入的调度器，负责向各个应用分配资源以满足容量、组等限制，这个调度器不负责管理或追踪应用的状态，也不负责由于硬件错误或应用问题导致的task失败重启问题，调度器只一句应用的资源需求来执行调度工作，调度的内容是一个抽象的Resouce Container，其中包含资源元素，如内存,cpu,网络及磁盘等。

3）NodeManager是每个节点的slave，其负责管理应用的container，管理他们的资源使用，并向ResouceManater汇报整体的资源使用情况。NM会接收并且处理来自AM的container的启动和停止的各种请求。

4）ApplicationMaster，负责向RM的调度器协商合理的Resource Container并追踪他们的状态，管理进度，AM在系统中以Container的形式执行。AM同时与NodeManager进行通信以启动或者停止服务/监控所有任务的运行情况，并在任务失败的情况下，重新为任务申请资源并且重启任务、负责推测任务的执行。

5）Container，Yarn对系统资源的抽象，同时也是系统资源分配的基本单位，封装节点上多维度资源，包括cpu/内存/磁盘和网络等。Yarn会每个任务分配一个Container，并且该任务只能使用该container中所描述的资源。

# 三、Yarn 接口（支持其他计算模型）

# 四、Yarn服务、事件库及状态机

## 4.1 服务库

对于生命周期较长的对象，YARN采用了基于服务的对象管理模型对其进行管理，该模型具有以下几个特点：

1）每个被服务化的对象分为4个状态：NOTINIED（被创建）、INITED（已初始化）、STARTED（已启动）和STOPPED（已停止）

2）任何服务状态变化都可以触发另外一些动作

3）可以通过组合的方式对任意服务进行组合，以便进行统一管理

YARN中所有的服务对象最终实现了接口Service（定义最基本的服务初始化、启动、停止等操作），而AbstractService类提供了一个最基本的Service实现。Yarn中所有对象，如果是非组合服务，直接继承AbstractService即可，否则需要继承CompositeService（组合服务，组合各种服务对象）。例如ResourceManager，是一个组合对象，组合了各种服务对象，包括ClientRMService,ApplicationMasterLauncher、ApplicationMasterService等。

## 4.2 事件库

Yarn采用事件驱动的并发模型，为了构建该模型，Yarn将各种处理逻辑抽象成事件和对应事件调度器，并将每类事件的处理过程分割成多个步骤，用有限状态机表示。

处理过程：将请求作为事件进入系统，由中央异步调度器（Async-Dispatcher）负责传递给相应事件调度器（Event Handler）。该事件调度器可能将该事件转发给另外一个事件调度器，也可能交给带有限状态机的事件处理器，其处理结果以事件的形式输给中央异步调度器。

在Yarn中，所有核心服务实际是都是一个中央异步调度器，包括ResourceManager,Node

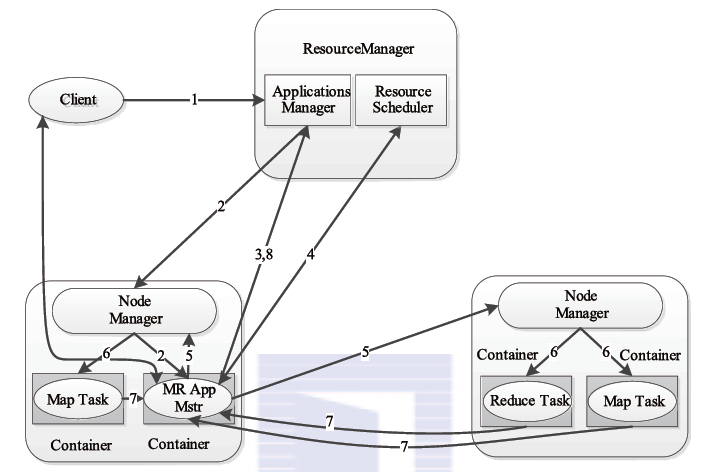
-Manager，MRAppMaster等，它们维护了事先注册的事件和事件处理器，根据接收的事件类型驱动服务的运行。

## 4.3 状态机库

状态机由一组状态组成，这些状态分成三类：初始状态，中间状态和最终状态。在Yarn中，每个状态由一个四元组表示，转换前状态（prestate）、转换后状态（poststate）、事件和回调函数。

# 六、Yarn 应用程序启动及执行

运行在Yarn上的应用程序主要分为两类，端应用程序和长应用程序，其中端应用程序指在一定时间内（可能是秒级、分钟级或小时级等）可运行完成并正常退出的应用程序，如Mapreduce及TEZ DAG作业等；长应用程序是指在指不出意外，不终止的应用程序，一般指一些服务，如Storm Service、HBase Service等。这两类应用程序作用不同，但是运行在Yarn上的流程是相同的，下图是应用程序在Yarn上的执行过程：



Yarn工作流程

## 6.1提交应用及其使用到的资源

用户向YARN中RM提交应用程序，其中包括ApplicationMaster程序，启动ApplicationMaster的命令、用户程序等。客户端将应用程序所需的文件资源（外部字典、jar包及二进制文件）提交到HDFS，流程图如下所示:

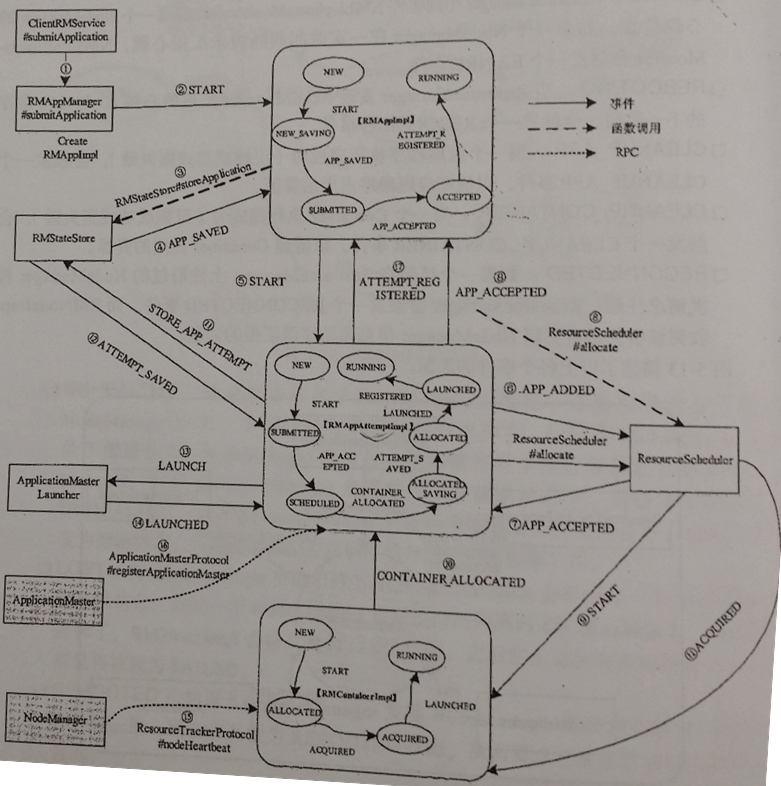


job提交流程图

图中yarnClient(ResourceMgrDelegate)通过RPC协议，提交Application到RM中的ClientRMService。App Context的初始化时根据ClientProtocol（例如YarnRunner传的参数生成）,App Context在YarnRunner中的createApplicationSubmissionContextt()方法生成，其中包括jar,资源,队列等参数的设置。ResourceMgrDelegate作业Yarn Cluster的client。

## 6.2 AppliationMaster的启动

RM会该应用程序分配第一个Container，并与对应的NodeManager通信。要求它在这个Container中启动应用程序的AM。AM与RM通信，请求和获取资源。ApplicationMaster获取资源后，与对应的NodeManager通信以启动任务，则NM首先从HDFS上下载文件缓存到本地，然后启动该任务，流程图如下所示：



ApplicationMaster启动流程（hadoop技术内幕130页）

1. ResourceManager中的ClientRMService实现了ApplicationClientProtocol协议，它处理来自客户端的要求，并调用RMAppManager#submitAppliacation通知其他相关服务作进一步处理。

2. RMAppManager为该应用程序创建了一个RMAppImpl（app的运行状态）对象以维护它的运行状态，并判断系统状态，如果故障重启状态，则向它发送一个RMappEventType

.RECOVER事件，否则发送RMAppEventType.START事件。

3. RMAppImpl收到RMAppEventType.START事件后，会调用RMStateStore#store

Application(RMStateStore是插拔式组件，在不启用RM恢复机制的前提下，默认实现是NullRMStateStore，它不会进行任何保存工作，其他实现还有MemoryRMStateStore,FileSyste

mRMStateStore等)，以日志记录RMAppImpl当前信息，至此RMAppImpl的运行状态由NEW转移为NEW\_SAVING。

4. 日志记录完成后，RMStateStore进一步向RMAppImpl发送RMAppEventType.APP\_

SAVED事件

5. RMAppImpl收到RMAppEventType.APP\_NEW\_SAVED事件后，将触发AddApplication

ToSchedulerTransition，状态改变成RMAppState.SUBMITTED，同时创建一个运行实例对象RMAppAttemptImpl，同时向它发送一个RMAppAttemptEventType.START事件，RMAppImpl状态转为ACCEPTED。

6. RMAppAttemptImpl收到START事件后，进行一些必要的初始化工作（设置初始事件，各种安全Token等等），向ApplicationMasterService注册，然后向ResourceScheduler发送SchedulerEventype.APP\_ADDED事件，至此RMAppAttemptImpl状态由NEW转移到SUBMITTED。

7. ResourceScheduler收到SchedulerEventype.APP\_ADDED事件后，首先进行一些权限检查，然后将应用程序信息保存到内部的数据结构中，并将应用程序信息保存到内部的数据结构中，并向RMAppAttemptImpl发送RMAppAttemptEventType.APP\_ACCEPTED事件。

8. RMAppAttemptImpl收到RMAppAttemptEventType.APP\_ACCEPTED事件后，首先向RMAppImpl发送一个RMAppEventType.APP\_ACCEPTED事件（RMAppImpl收到该事件后直接将状态SUBMITTED转移到ACCEPTED），然后调用ResourceScheduler#allocate为应用程序的ApplicationMaster申请资源，请求格式如下：

ResourceRequest request =BuilderUtils.newResourceRequest( AM\_CONTAINER\_PRIORITY, ResourceRequest.ANY, appAttempt.getSubmissionContext().getResource(), 1);

在请求中，包括优先级，可在任意节点（any）及资源量为x的container。至此RMappAttemp

Impl的状态为SUBMITTED转移为SCHEDULED。

9. ResourceManager为应用程序的ApplicationMaster分配资源后，创建一个RMContainerImpl，并向它发送RMContainerEventType.START事件。

10.RMContainerImpl收到RMContainerEventType.START事件后，直接向RMAppAttempt放一个CONTAINER\_ALLOCATED事件，至此RMContainerImpl状态从NEW转移到ALLOCATED。

11. RMAppAttemptImpl收到RMAppAttemptEventType.CONTAINER\_ALLOCATED事件后，调用ResouceScheduler#allocate获取分配的资源，ResourceScheduler将资源返回给它之前，会向RMContainerImpl发送一个RMContainerEvent.ACQUIRED事件（RMContainerImpl收到该事件，会向ContainerAllocationExpirer注册以启动监控，之后向RMAppAttemptImpl发送Container\_ACQUIRED事件），而RMAppAttemptImpl收到资源后，第一事件向RMStateStore发送MStateStoreEventType.STORE\_APP\_ATTEMPT事件请求记录日志，至此RMAppAttemptImpl状态从SCHEDULED转移到ALLOCATED\_SAVING。

12. 日志记录完后，RMStateStore进一步向RMAppAttemptImpl发送RMAppAttempt

EventType.ATTEMPT\_SAVED事件。

13. RMAppAttemptImpl收到ATTEMPT\_SAVED事件后，向ApplicationMasterLauncher发送LAUCH事件，状态从ALLOCATED\_SAVING转移到ALLOCATED。

14.ApplicationMasterLauncher收到AMLauncherEventType.LAUCH事件后，将该事件放到事件队列中，等待AMLauncher线程池中的线程处理该事件。处理方法是与对应的Node

Manager通信，启动ApplicationMaster，一旦启动成功后，向RMAppAttemptImpl发送LAUNCHED事件后，会向AMLivelinessMonitor注册，以监控运行状态。RMAppAttemptImpl状态从Allocated转移到LAUNCHED。

15.NodeManager通过心跳机制汇报ApplicationMaster所在Container已经成功启动，收到该信息后，ResourceScheduler发送一个RMContainerEventType.LAUCHCED事件，RMContainer收到该事件后，会从ContainerAllocationExpired监控列表中很删除。

16.启动的ApplicationMaster通过RPC函数ApplicationMasterProtocol#registerApplic

actionMaster向ResourceManager注册，RM中的ApplicationMasterService服务接收到该请求后，向RMAppAttemptImpl发送一个RMappAttemptEventType.REGISTERED事件，而RMAppAttemptImpl收到该事件后，首先保存该ApplicationMaster的基本信息（比如host,启动的rpc端口等），然后向RMAppImpl发送一个RMAppEventType.ATTEMPT\_REGISTERED事件。RMAppAttemptImpl状态从LAUNCHED转移到RUNNING。

17.RMAppImpl收到RMAppEventType.ATTEMPT\_REGISTERD事件后，将状态从ACCEPTED转换为诶RUNNING。

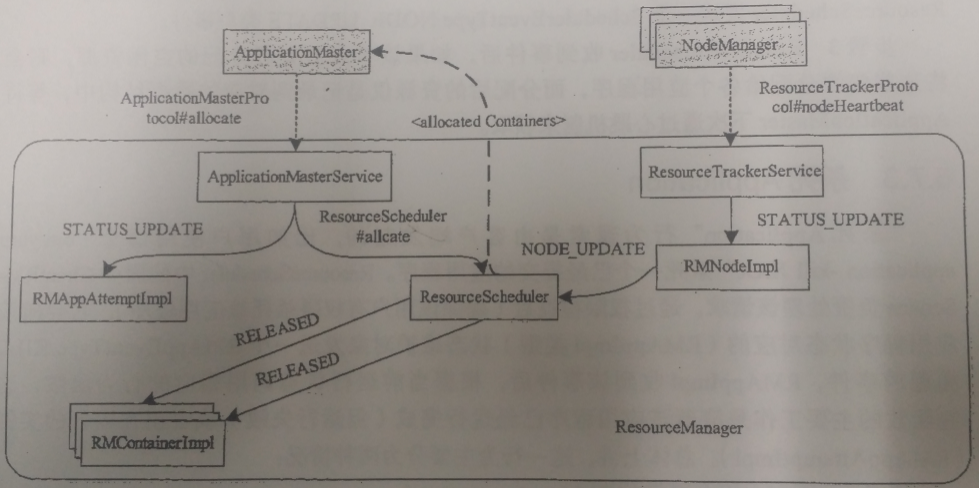
## 6.3 ApplicationMaster的注册

在上节，ApplicationMaster在Container中启动后，通过ApplicationMasterProtocol#

registedService向ResourceManager注册。RM中的ApplicationMasterService服务接收到该请求后，将向RMAppAttemptImp发送一个RMAppAttemptEvent.REGISTERED事件。

## 6.4 任务的资源申请

ApplicationMaster采用轮询的方式通过RPC协议向RM申请和领取资源。AM通过函数AMprotocol#allocated向RM汇报资源需求，包括资源需求描述、待释放的Container列表、请求加入黑名单的节点列表、请求移除黑名单的节点列表。流程如下图所示：



Container分配与申请流程

AM中的ApplicationMasterService负责处理ApplicationMaster的请求，一旦受到该请求，会向RMAppAttemptImpl发送一个RMAppAttemptEventType.STATUS\_UPDATE类型时间，而RMAppAttemptImpl收到该事件后，将更新应用程序执行进度和AMlivenessMonitor中记录的应用程序最近更新时间。

AppliationMasterService调用ResourceScheduler#allocate函数，将该ApplicationMaster资源需求汇报给ResourceScheduler。ResourceScheduler读取待释放的Container列表，依次向对应的RMContainerImpl发送RMContainerEventType.RELEASED类型事件，以杀死正在运行的Container，然后将新的资源需求更新到对应的数据结构中，并返回已经为该应用程序分配的资源。

NodeManager通过RPC函数ResourceTracker#nodeHeartbeat向ResourceManager汇报各个Container运行状态。RM中的ResourceTrackerService负责处理来自NodeManager的请求，一旦收到该请求，会向RMNodeImpl发送一个RMNodeEventType.STATUS\_UPDATED类型事件，而RMNodeImpl收到该事件后，将更新各个Container的运行状态，并进一步向RM发送一个SchedulerEventType.NODE\_UNPDATE类型事件。ResourceScheduler收到事件后，如果该节点有可分配的空闲资源，会将这些资源分配给各个应用程序，而分配后的资源仅是记录到对应的数据结构中，等待ApplicationMaster下次通过心跳机制来领取。

## 6.5 任务的启动与执行

ApplicationMaster申请到资源后，AM就会将启动命令交给NodeManager，要求NM启动任务。启动命令中包含一些信息使得Container可以与Application Master进行通信。

NodeManager为任务设置好运行环境（包括环境变量、jar包、二进制环境等）后，将任务启动命令写到一个脚本中，并通过该运行脚本启动任务（container）。

在Container内执行用户提交的代码，各个Container通过某个RPC协议向ApplicationMaster汇报自己的状态和进度，以让ApplicationMaster随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务。

## 6.6 任务的结束

在应用程序运行过程中，用户可随时通过RPC向ApplicationMaster查询应用程序的当前运行状态，应用程序运行完成后，ApplicationMaster向ResourceManager注销并关闭自己。

# 七 Yarn 资源隔离

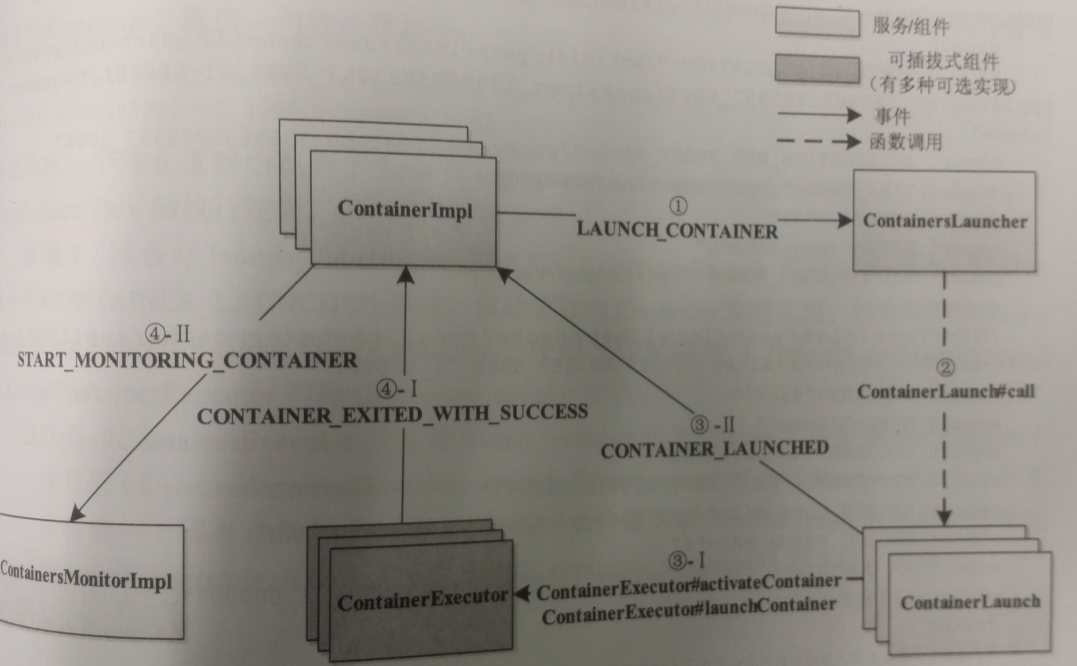
资源调度器是Yarn中最核心的组件之一，且是插拔式的，用户可一个根据接口规范实现自己的调度器。Yarn自带了FIFO、Capacity Scheduler和Fair Scheduler三种常用的资源调度器。

## 7.1 Container构建及资源分配

Container 的启动由ContainerLauncher服务完成的，该服务进一步调用插拔式组件ContainerExcecutor。Yarn 提供两种ContainerExecutor实现，一种是DefaultContainer

Executor，另一种是LinuxContainerExecutor。

Container可以当做NodeManager中用于维护一个Container生命周期的数据结构，其实现为ContainerImpl，该类维护了一个Container状态机，记录了Container可能存在的各个状态以及导致状态间转换的事件，当某个事件发生时，ContainerImpl会根据实际情况进行节点状态转移，同时出发一个行为。过程如下图所示：



Container运行过程

1）ContainerImpl向ContainersLauncher服务发送LAUNCH\_CONTAINER事件，请求启动Container

2）ContainersLauncher收到LAUNCH\_CONTAINER事件后，将为该Container创建一个Callable类型的对象ContainerLaunch，并放到线程池中执行；为Container创建Tokens文件和执行脚本Launch\_container.sh（ContainerLauch.call()中command.addAll(...sh)，通过ShellCommandExecutor.execute执行），并保存到NodeManager私有目录${yarn.nodemana

ger.local-dirs}/nmPrivate中。

3）准备好脚本后，ContainerLaunch首先向ContainerImpl发送一个CONTAINER\_LAUN

CHED事件，以让它启动Container监控机制，然后调用ContainerExecutor#launchContainer启动Container。目前YARN提供DefaultContainerExecutor和LinuxContainerExecutor两种ContainerExecutor实现。

4）ContainerImpl收到CONTAINER\_LAUNCHED事件后，进一步向ContainersMonitor

-Impl服务器发送一个个START\_MONITORING\_CONTAINER事件，启动对该Container的内存资源使用监控，ContainerImpl状态从LOCALIZED促成RUNNIG。

## 7.2 Container Cpu隔离

Yarn使用Cgroup对Cpu资源进行隔离，在上文中，ContainerExecutor#LaunchContainer

方法，实现了Cgroup对Container的资源隔离，其实现是resourceHandler.preExecute(co-

ntainerID,container.getResource)，如果没有配置resource-handler.class，则使用DefaultLC

-EResourcesHandler，方法中没有cpu的隔离。CgroupsLCEResourcesHandler中实现cpu的隔离，在preExecute中调用了CreateCgroup和updateCgroup在Cgroup目录中创建Container对应的cpu目录，并设置其参数，参数为cpu.share，计算公式如下：

int cpuShares = CPU\_DEFAULT\_WEIGHT \* containerResource.getVirtualCores();

creatGroup及updateGroup的核心源码如下所示：

void createCgroup(String controller, String groupName)

throws IOException {

String path = pathForCgroup(controller, groupName);

if (! new File(path).mkdir()) {

throw new IOException("Failed to create cgroup at " + path);

}

updateCgroup源码如下

void updateCgroup(String controller, String groupName, String param,

String value) throws IOException {

String path = pathForCgroup(controller, groupName);

param = controller + "." + param;//省略了部分代码

FileWriter f = new FileWriter(path + "/" + param, false);

f.write(value);

}

其中value为计算处理的cpu.shares值。

当Container执行后，通过CgroupsLCEResourcesHandler#postExecute对Cgroups中生成的Container cpu资源隔离文件进行删除（释放其CPU资源）,postExecute中最终调用:

boolean deleteCgroup(String cgroupPath) {

do {//省略部分代码

deleted = new File(cgroupPath).delete();

} while (!deleted && (clock.getTime() - start) < deleteCgroupTimeout);

}

删除在/cpu/hadoop-yarn/层级下创建的Container层级，完成cpu的释放。

Hadooop CDH 2.3中，通过上面的方式来控制cpu的隔离（设置Container的虚拟核数），通过修改CgroupsLCEResourcesHandler#setupLimits，添加设置cpu.cfs\_quota\_us及cpu.cfs\_period\_us来控制Container运行时占用的Cpu时间，代码如下：

1. public void setupLimits(ContainerId containerId,
2. Resource containerResource) throws IOException {
3. String containerName = containerId.toString();
4. if (isCpuWeightEnabled()) {
5. createCgroup(CONTROLLER\_CPU, containerName);
6. int cpuShares = CPU\_DEFAULT\_WEIGHT \* containerResource.getVirtualCores();
7. // absolute minimum of 10 shares for zero CPU containers
8. cpuShares = Math.max(cpuShares, 10);
9. updateCgroup(CONTROLLER\_CPU, containerName, "shares", String.valueOf(cpuShares));
10. int nmShares =CPU\_DEFAULT\_WEIGHT\*conf.getInt(YarnConfiguration.NM\_VCORES,
11. YarnConfiguration.DEFAULT\_NM\_VCORES)
12. updateCgroup(CONTROLLER\_CPU,containerName,"cfs\_period\_us",String.valueOf(nmShares));
13. updateCgroup(CONTROLLER\_CPU,containerName,"cfs\_quota\_us",String.valueOf(cpuShares));
14. }
15. }

代码修改后，编译成jar包替换原yarn中相应的jar包后，重启Yarn，启动Cpu计算型的作业（计算PI值）：

hadoop jar ./share/hadoop/mapreduce2/hadoop-mapreduce-examples-2.3.0-cdh5.0.0.jar pi -Dmapreduce.job.queuename=root.queueA -Dmapreduce.map.cpu.vcores=3 -Dmapreduce.reduce.cpu.vcores=2 1 10000000

通过上面的命令运行后，ApplicationMaster占用Cpu时间在12.5%（1/8），没有配置AM的vcores参数，因此默认值是1。启动的Map任务占用的cpu时间在37.5%（3/8），Reduce任务大致在25%（2/8）左右。通过top命令，执行的效果图依次如下：



AM Container CPU使用比例



Map Container CPU使用比例



Reduce Container CPU使用比例

结果如下：



## 7.3 Container 内存隔离

应用程序的内存监控是由服务ContainersMonitorImpl实现的，它保存了每个Container进程的pid，这样它内部的Monitoring Thread线程每个一段时间（由yarn.nodemanager

.contain-er.interval-ms指定，默认是3000ms）扫描正在运行的Container进程树，并检查使用的内存量是否超过上限值。

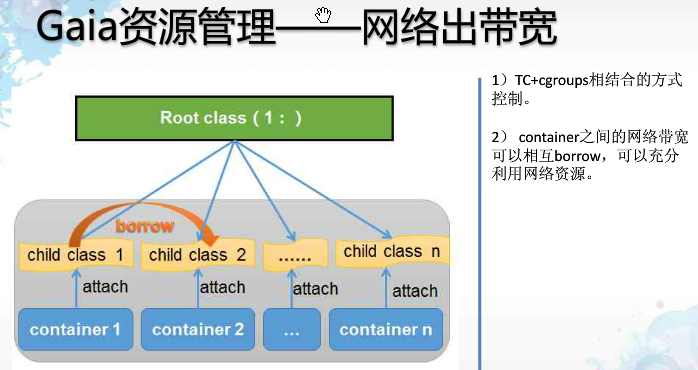
Yarn还提供了一个基于Cgroups的内存隔离方案，该方案实现了一个新的ContainerMo

nitor插件CgroupsContainersMonitor。

## 7.4 其他资源的隔离

当前Yarn仅支持cpu和内存的隔离，在实际运行环境中，磁盘I/O及网络带宽都是比较重要的资源，这两类资源的隔离都可以使用cgroup来实现。

下图是Gaia的网络带宽控制，其采用的是创建根分类及子分类：



# 八.Yarn 资源调度器

## 8.1 Yarn Queue管理

Yarn Queue的信息配置在fair-scheduler.xml中，由yarn.scheduler.fair.allocation.file指定。修改fair-scheduler.xml后，通过命令：

$bin/yarn -rmadmin refreshQueues

更新Yarn中的队列配置。

通过Yarn提交程序时，通过以下命令：

$bin/yarn jar .../hadoop-mapreduce-examplesXXX.jar wordcount /data/helloworld /output/result

通过参数指定队列。

Yarn的队列的资源以Vcore数目为资源管理单位，对其配置最大及最小资源。Application提交到队列中，根据调度策略开始执行后，RM给Application分配Container，对应在/hadoop-yarn的container，Cgroup层级管理，Container属于Hadoop-yarn的子层，如下图所示：

Yarn队列与Cgroup的对应

## 8.2 资源调度器

资源调度器是Hadoop Yarn中最核心的组件之一，是RM的插件式服务组件，负责整个集群资源的管理和分配。从目前来看，多用户资源调度器的设计思路：

1）在一个物理集群上虚拟多个Hadoop集群，这些集群拥有全套独立的Hadoop服务，典型的代表是HOD调度器

2）另一种是扩展Yarn调度器，使之支持多个队列多用户

下面分析第二种设计思路的实现，当前提供FIFO,Yahoo!的Capacity Scheduler和Facebook的Fair Scheduler三类调度器。

### 8.2.1 资源调度器类图



资源调度的核心接口是YarnScheduler，其定义了队列，资源及节点相关的方法，最终实现有CapacityScheduler和FairScheduler，FairScheduler的资源调度更加灵活。

在ResourceManager中有变量ResourceScheduler，根据配置来选择CapacityScheduler或者FairScheduler调度（可插拔）。在这两种实现中，都存储了队列（fair-scheduler.

-xml或者capicity-scheduler.xml定义，也可以通过API动态生成）。QueueManager对队列进行管理，每个队列中存储了队列的资源及提交到该队列的应用信息。在Yarn中使用层级队列管理机制，因此队列分为ParentQueue及LeafQueue两种。队列为每个节点定义了一个类FsSchedulerNode，类中只保存了调度器所相关的信息，包括资源及Containers信息，这样方便Scheduler对节点进行管理，其中RMNodeImpl可以看做是每个节点的状态机。

对于队列的ACL权限管理，通过QueueACLsManager来控制，QueueACL定义两种角色：提交应用及管理队列，在启用ACL后，通过方法checkAccess来检测其权限是否可用。

### 8.2.2 资源调度模型

Yarn采用双层资源调度模型，第一层中，RM中的资源调度器将资源分配给各个Applica

-tionMaster，第二层中ApplicationMaster进一步将资源分配给它内部的各个任务，这部分的调度由应用程序自己决定。Yarn的资源调度是异步的，调度求将资源分配给应用程序后，不会立刻push给对应的ApplicationMaster，而是暂时放到一个缓冲区中，等待AM通过周期性的心跳主动来去。资源分配流程如下所示：



资源调度器资源分配流程

1）NodeManager通过周期性心跳汇报节点信息，心跳信息的汇报通过NodeManager中的NodeStatusUpdateImpl#startStatusUpdater来完成。通过RPC协议获取ResourceTrack

-er，然后调用nodeHeartbeat，将节点信息发送到ResourceManager中

2）ResourceManager为NodeManager返回一个心跳应答，包括需释放的Container的列表信息。RM中的ResourceTrackerService收到NM的心跳后，通过nodeHeartbeat，向NM发送响应信息

3）ResourceManager收到来自NodeManager的信息后，会触发一个NODE\_UPDATE事件

4）ResourceScheduler收到NODE\_UPDATE事件后，会按照一定的策略将该节点上的资源按分配给各应用程序，并将分配结果放到一个内存数据结构中。

5）应用程序的ApplicationMaster向ResourceManager发送周期性心跳，以领取最新分配到的Container。

6）ApplicationMaster收到新分配的container列表后，会将这些Container进一步分配给其他内部到的各个服务

## 8.3 Yarn层级队列管理机制

在Hadoop 0.20x版本及更早的版本中，Hadoop采用了平级队列组织方式，随着Hadoop应用越来越广泛，扁平化的队列组织方式不能满足实际需求，因此出现层级队列组织方式。典型的应用场景，在一个Hadoop集群中，管理员将所有计算资源划分为若干个队列，每个队列对应一个组织，其中有一个“Org1”组织，分到60%的资源，它包含3种类型应用：产品线应用；实验性应用，分属于proj1,proj2,proj3三个项目；其他类型应用。这就引出来层级队列组织方式，具有以下特点：

1）子队列，队列可以嵌套，用户只能将应用程序提交到最底层队列（叶子队列）

2）最少容量及最大容量

另外，Hadoop队列管理机制由用户管理和系统资源管理两个部分。