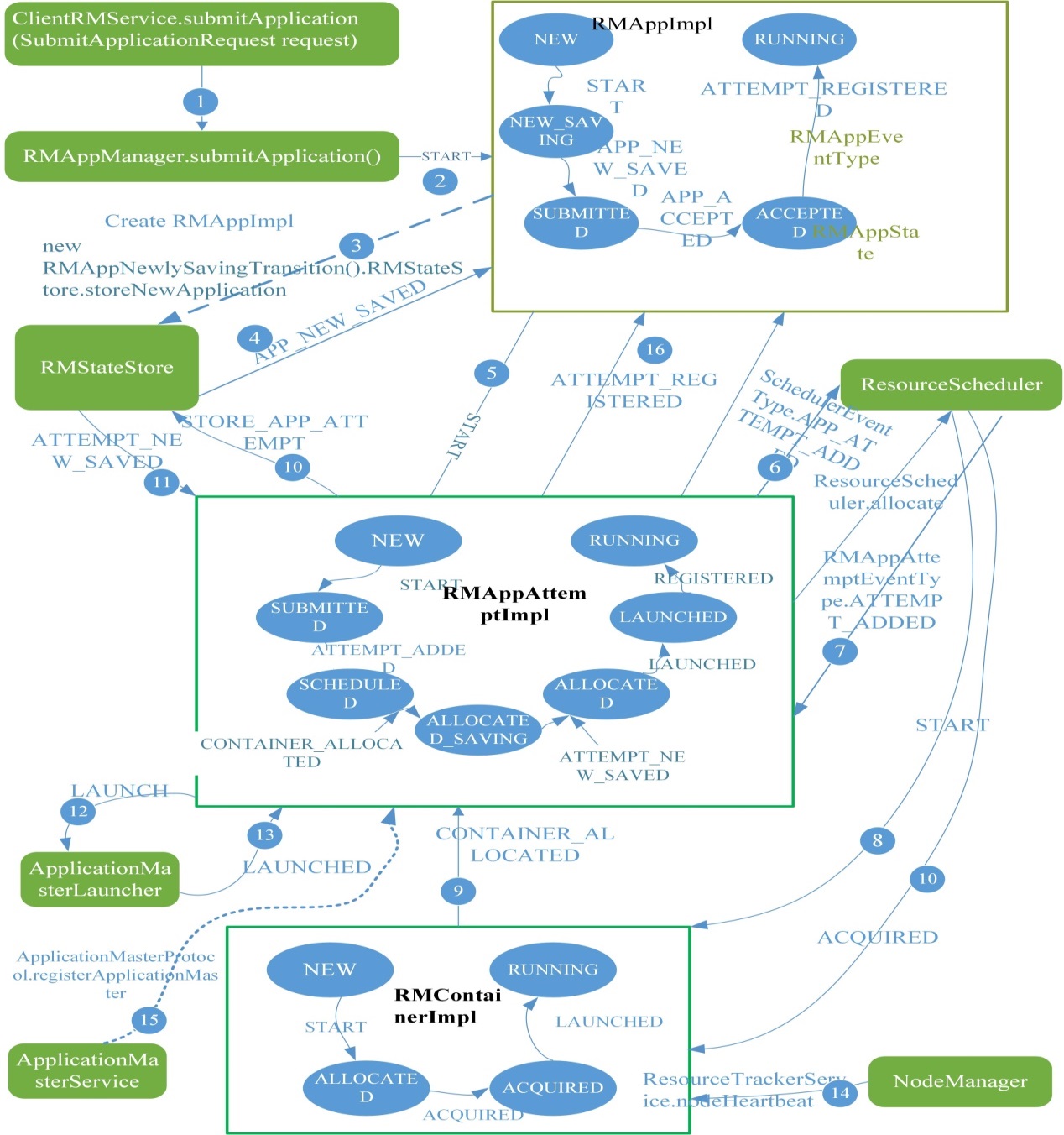
YARN应用提交及启动流程与资源调度隔离分析

# 一 YARN应用提交启动流程分析

YARN应用的提交和执行流程可分为两个步骤：1，启动ApplicationMaster；2，由ApplicationMaster创建应用程序为其申请资源并监控整个运行过程。下面将分别这两个流程进行分析。

**ApplicationMaster启动过程分析：**

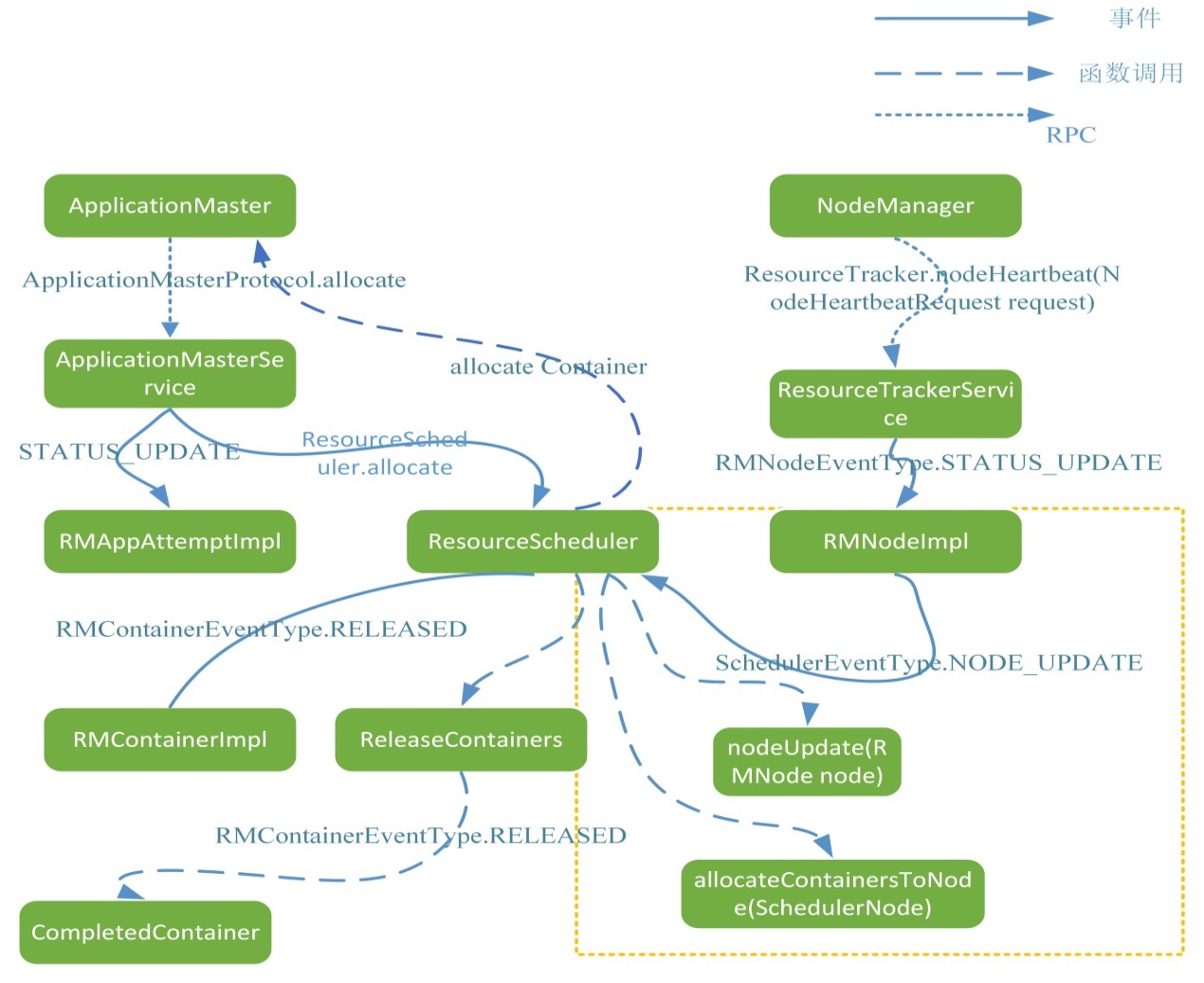


ApplicationMaster的启动过程涉及到：ResourceManager、ClientRMService、RMAppManager、ResourceScheduler、ApplicationMasterLauncher、Application -MasterService、RMNode几个组件和RMAppImpl、RMAppAttemptImpl、AMLauncher、RMContainerImpl 、RMNodeImpl、RMStateStore等几个状态机组件。整个流程从客户端程序调用RPC函数ApplicationClientProtocol. submitApplication (SubmitApplicationRequest request) 开始，其完整过程如图所示，假设中间没有异常流程。

1. ResourceManager 中的ClientRMService实现了ApplicationClientProtocol协议，它处理来自客户端的所有RPC请求，客户端通过SubmitApplication -Request 对象提交应用程序，处理程序从SubmitApplicationRequest 对象中获取一个ApplicationSubmissionContext对象，首先对独立于RM配置的属性进行校验，依赖于RM的属性后续将交由RMAppManager进行验证，之后会继续调用RMAppManager的submitApplication (Application -SubmissionContext submissionContext, long submitTime,String user) ；
2. RMAppManager为该应用程序的其它依赖于RM的属性进行校验，并创建一个RMAppImpl对象以维护它的运行状态，并判断系统状态。如果发生故障重启则向它发送一个RMAppEventType.RECOVER事件，否则发送一个RMAppEventType.START事件；
3. RMAppImpl收到RMAppEventType.START事件后会创建一个RMAppNewlySavingTransition对象，该对象会从RMContext中获取到 RMStateStore对象( RMStateStore是一个可插拔的组件，RM用其存储应用程序的状态，非阻塞线程，一旦执行完会发送RMAppStoredEvent事件通知相应的RMApp。在不启用RM恢复机制的前提下其默认实现为NullRMStateStore，不会进行任何工作，其它实现还有MemoryRMState Store、FileSystemRMStateStore等)，进一步调用RMStateStore.storeNew -Application(RMApp app)，以日志的形式记录RMAppImpl当前信息，至此，RMAppImpl的运行状态由NEW转移到NEW\_SAVING；
4. RMStateStoreAppEvent.STORE\_APP事件会触发操作创建一个StoreApp -Transition对象，该对象进一步会发送RMAppEventType.APP\_NEW\_ SAVED事件，RMAppImpl收到RMAppEventType. APP\_NEW\_SAVED事件会创建一个AddApplicationToSchedulerTransition对象，触发SchedulerEventType. APP\_ADDED事件，此时触发Scheduler初始化工作，根据配置文件初始化相应的资源调度器，创建一个FiXXSchedulerApp对象跟踪和维护该应用程序的运行时信息。至此，RMAppImpl的运行状态由NEW\_SAVING转移到SUBMITTED；然后RMAppImpl 发送RMAppEventType.APP\_ACCEPTED事件，产生StartAppAttemptTransition 对象，调用RMAppImpl. createAndStartNewAttempt().createNewAttempt() 创建RMAppAttempt对象，并产生RMAppAttemptEventType.START事件。至此，RMAppImpl由SUBMITTED转移到ACCEPTED；
5. RMAppAttemptImpl收到RMAppAttemptEventType.START事件后进行一系列工作包括向ApplicationMasterService注册(appAttempt.masterService. registerAppAttempt(appAttempt.applicationAttemptId))，产生ClientToAM TokenSecretManagerInRM Token等工作，产生SchedulerEventType.APP\_ ATTEMPT\_ADDED事件，至此RMAppAttemptState状态由RMAppAttemptState.NEW到RMAppAttemptState.SUBMITTED；
6. ResourceScheduler收到SchedulerEventType.APP\_ATTEMPT\_ADDED事件后，会对该RMAppAttemptImpl进行一些操作包括更新更新调度队列、优先级等，然后产生RMAppAttemptEventType.ATTEMPT\_ADDED事件；
7. RMAppAttemptImpl接收到RMAppAttemptEventType.ATTEMPT\_ADDED会创建一个ScheduleTransition对象，尝试创建一个新的RMAppAttempt对象，此时会先重置容器，因为这些信息会随着请求传递给调度器，而且调度器会在AM容器分配之后减掉相应数量的容器。在创建RMAppAttempt对象时会修改ResourceRequest的容器数量（默认设置为1）、优先级、资源名称等信息。然后调用RMAppAttemptImpl对象内部的YarnScheduler对象的allocate方法产生一个Allocation实例，ApplicationMaster通过它更新或者释放不需要的容器资源。至此，RMAppAttemptImpl由RMAppAttemptState.SUBMITTED转移到RMAppAttemptState.SCHEDULED状态；
8. 之后ResourceScheduler产生RMContainerEventType.START事件，创建ContainerStartedTransition对象，产生RMAppAttemptEventType. CONTAINER\_ALLOCATED事件。
9. RMAppAttemptImpl在收到RMAppAttemptEventType.CONTAINER\_ ALLOCATED事件后会创建AMContainerAllocatedTransition对象，该对象会尝试从Scheduler对象中获取AM Container，然后设置该RMAppAttempt对象的MasterContainer为获取到的第一个容器（id为0）并创建RMContainerImpl。至此RMAppAttempt由RMAppAttemptState. SCHEDULED转移到RMAppAttemptState.ALLOCATED\_SAVING，RMContainerImpl状态由RMContainerState.NEW转移到RMContainerState. ALLOCATED；
10. RMAppAttemptImpl收到RMAppAttemptEventType.CONTAINER\_ ALLOCATED事件后调用ResourceScheduler.allocate获取分配的资源，ResourceScheduler将资源返回之前会向RMContainerImpl发送RMContainerEventType.ACQUIRED事件。RMContainerImpl收到该事件后，会向ContainerAllocationExpirer注册以启动监控。RMAppAttemptImpl收到RMAppAttemptEventType.CONTAINER\_ALLOCATED之后会创建AMContainerAllocatedTransition对象最终产生RMStateStoreEventType. STORE\_APP\_ATTEMPT事件，请求记录日志信息，同时向RMAppImpl发送RMAppEventType.APP\_RUNNING\_ ON\_NODE事件通知该APP，并以日志的形式记录执行信息，RMContainerImpl的状态由RMContainerState.ALLOCATED转移到RMContainerState.ACQUIRED。
11. 日志记录完成后，RMStateStore进一步向RMAppAttemptImpl发送RMAppAttemptEventType.ATTEMPT\_NEW\_SAVED事件；
12. RMAppAttemptImpl收到RMAppAttemptEventType.ATTEMPT\_NEW\_ SAVED事件后，将向ApplicationMasterLauncher发送AMLauncherEvent -Type.LAUNCH事件，至此RMAppAttemptImpl状态从RMAppAttempt -State.ALLOCATED\_SAVING转移到RMAppAttemptState. ALLOCATED。
13. ApplicationMasterLauncher收到AMLauncherEventType.LAUNCH事件后会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，与该对应的NodeManager通信，启动ApplicationMaster，一旦成功启动后将进一步向RMAppAttemptImpl发送RMAppAttempt -EventType.LAUNCHED事件。RMAppAttemptImpl收到RMAppAttempt -EventType.LAUNCHED事件后会向AMLivelinessMonitor注册，以监控运行状态。至此RMAppAttemptImpl状态从RMAppAttemptState. ALLOCATEDG转移到RMAppAttemptState. LAUNCHED。
14. NodeManager通过心跳机制汇报ApplicationMaster所有Container已经成功启动，收到该事件后，ResourceScheduler将发送一个RMContainerEvent -Type.LAUNCHED事件，RMContainerImpl接收到该事件后会从Container -AllocationExpirer监控列表中移除；
15. 以启动的ApplicationMaster通过RPC函数ApplicationMasterProtocol. registerApplicationMaster向ResourceManager注册，ResourceManager中的ApplicationMasterService服务接收到该请求后，将向RMAppAttemptImpl发送RMAppAttemptEventType.REGISTERED事件，而RMAppAttempt -Impl收到该事件后首先保存该ApplicationMaster的基本信息，然后向RMAppImpl发送一个RMAppEventType.ATTEMPT\_ REGISTERED事件。至此RMAppAttemptImpl状态从RMAppAttemptState. LAUNCHED转移到RMAppAttemptState. RUNNNING；
16. RMAppImpl接收到RMAppEventType.ATTEMPT\_REGISTERED事件后将状态由ACCEPTED转换为RUNNING，不做其它操作。

通过以上步骤完成YRAN应用程序提交的第一步：ApplicationMaster启动。ApplicationMaster在NodeManager上成功启动并向ResourceManager注册后需要向ResourceManager请求资源（Container），下面将介绍获取资源的整个过程以及ResourceManager在此过程中内部涉及的主要工作流程。

**资源申请与分配流程分析：**

****

该过程可以看做是以下两个阶段的迭代循环过程：

阶段1 ApplicationMaster通过心跳消息向RM上报资源需求并领取已经分配的资源；

阶段2 NodeManager向RM汇报各个Container的运行状态，如果RM发现该NM上的资源有空闲则进行一次资源再分配，并将分配的资源保存到对应的应用程序数据结构中，等待下次ApplicationMaster发送心跳消息时获取（即第一阶段）。

1. 阶段1

步骤1 ApplicationMaster通过RPC函数ApplicationMasterProtocol.allocate向ResourceManager上报资源需求，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等；

步骤2 ResourceManager中的ApplicationMasterService负责处理来自ApplicationMaster的请求，一旦收到该请求，会授权给该请求并返回当前应用的AMRMTokenIdentifier，进一步检查其是否已注册或者重复注册，之后，会向RMAppAttemptImpl发送一个RMAppAttemptEventType. STATUS\_UPDATE事件，而RMAppAttemptImpl在收到该事件后，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，然后ApplicationMasterService会继续对当前请求进行处理包括设置Resource Requests的标签、资源情况校验、更新Resource Requests、向RMAppAttemptImpl发送新的事件（调用ResourceScheduler.allocate函数）以及通知AM Container更新情况等，最终封装成AllocateResponse；

步骤3 ApplicationMasterService调用ResourceScheduler.allocate函数，将该ApplicationMaster的资源情况汇报给ResourceScheduler；

步骤4 ResourceScheduler首先读取待释放Container列表，依次调用completedContainer，传入 RMContainerEventType.RELEASED事件，杀死标记为待释放的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；

1. 阶段2

步骤1 NodeManager通过RPC函数ResourceTracker.nodeHeartbeat (NodeHeartbeatRequest request)向ResourceManager汇报各个节点的状态（NodeStatus），NodeStatus包含了该节点的信息和当前的状态以及运行在该节点上的所有的Container的状态；

步骤2 ResourceManager中的ResourceTrackerService负责处理来自NodeManager的请求，一旦收到该请求会从请求中取到该请求中的NodeStatus对象，然后对该Node作以下操作：1，检查该Node是否有效；2，检查该Node是否已注册；3，检查此次心跳消息是否为一次新的消息。之后会发送一个RMNodeEventType.STATUS\_UPDATE类型事件到RMNodeImpl。而RMNodeImpl在收到该事件后，首先会更新其LatestNode -HeartBeatResponse为该RMNodeEvent的NodeHeartbeatResponse，进一步读取RMNodeEvent的各项信息更新各Container的状态，封装RMNode -Impl实例并作为参数产生NodeUpdateSchedulerEvent对象，产生Scheduler -EventType.NODE\_ UPDATE类型的事件，然后更新个应用的存活时间；

步骤 3 ResourceScheduler收到事件后会从NodeUpdateSchedulerEvent获取到RMNode对象实例，然后更新该对象，包括：处理新的Lanched Container操作；新的Increased Containers；Completed Containers；如果节点处于decommissioning状态，则发送更新以使总资源等于所使用的资源，在此过程没有可用的资源进行调度；更新节点的资源利用率至此该节点的数据结构已更新，可以被ResourceScheduler重新调度。然后将这些资源通过allocateContainersToNode函数对资源分配给各个应用程序，当然此时分配的资源仅记录到对应的数据结构中，等ApplicationMaster下次通过心跳机制来领取。

至此，完成资源的申请预分配工作。

**ResourceScheduler剖析：**

资源调度器（ResourceScheduler）负责将各个节点上的资源封装成[Container](http://lib.csdn.net/base/docker)，并按照一定的约束条件（按队列分配，每个队列有一定的资源分配上限等）分配给各个application。YARN的资源管理器实际上是一个事件处理器，它需要处理来自Node、RMApp、ContainerAllocationExpirer的6种SchedulerEvent类型的事件，并根据事件的具体含义进行相应的处理。这6种事件含义如下：

（1） NODE\_REMOVED

事件NODE\_REMOVED表示集群中被移除一个计算节点（可能是节点故障或者管理员主动移除），资源调度器收到该事件时需要从可分配资源总量中移除相应的资源量。

（2） NODE\_ADDED

事件NODE\_ADDED表示集群中增加了一个计算节点，资源调度器收到该事件时需要将新增的资源量添加到可分配资源总量中。

（3）APP\_ADDED

事件APP\_ADDED 表示ResourceManager收到一个新的Application。通常而言，资源管理器需要为每个application维护一个独立的[数据结构](http://lib.csdn.net/base/datastructure)，以便于统一管理和资源分配。资源管理器需将该Application添加到相应的数据结构中。

（4）APP\_REMOVED

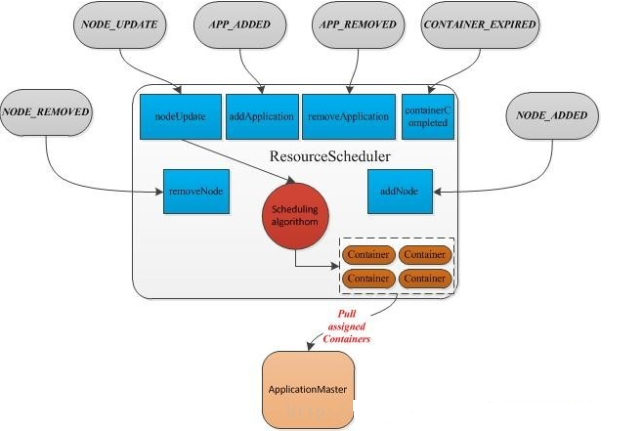
事件APP\_REMOVED表示一个Application运行结束（可能成功或者失败），资源管理器需将该Application从相应的数据结构中清除。

（5） CONTAINER\_EXPIRED

当资源调度器将一个container分配给某个ApplicationMaster后，如果该ApplicationMaster在一定时间间隔内没有使用该container，则资源调度器会对该container进行再分配。

（6）NODE\_UPDATE

NodeManager通过心跳机制向ResourceManager汇报各个container运行情况，会触发一个NODE\_UDDATE事件，由于此时可能有新的container得到释放，因此该事件会触发资源分配，也就是说，该事件是6个事件中最重要的事件，它会触发资源调度器最核心的资源分配机制。



当前YARN支持内存和CPU两种资源类型的管理和分配。当NodeManager启动时，会向ResourceManager注册，而注册信息中会包含该节点可分配的CPU和内存总量，这两个值均可通过配置选项设置，具体如下：

（1）yarn.nodemanager.resource.memory-mb

可分配的物理内存总量，默认是8\*1024MB。

（2）yarn.nodemanager.vmem-pmem-ratio

每单位的物理内存总量对应的虚拟内存量，默认是2.1，表示每使用1MB的物理内存，最多可以使用2.1MB的虚拟内存总量。

（3）yarn.nodemanager.resource.cpu-core（默认是8）

可分配的CPU总个数，默认是8

（4）yarn.nodemanager.vcores-pcores-ratio

为了更细粒度的划分CPU资源，YARN将每个物理CPU划分成若干个虚拟CPU，默认值为2。用户提交应用程序时，可以指定每个任务需要的虚拟CPU个数。在MRAppMaster中，每个Map Task和Reduce Task默认情况下需要的虚拟CPU个数为1，用户可分别通过mapreduce.map.cpu.vcores和mapreduce.reduce. cpu.vcores进行修改（对于内存资源，Map Task和Reduce Task默认情况下需要1024MB，用户可分别通过mapreduce.map.memory.mb和mapreduce.reduce. memory.mb 修改）。

YARN对内存资源和CPU资源采用了不同的资源隔离方案。对于内存资源，为了能够更灵活的控制内存使用量，YARN采用了进程监控的方案控制内存使用，即每个NodeManager会启动一个额外监控线程监控每个container内存资源使用量，一旦发现它超过约定的资源量，则会将其杀死。采用这种机制的另一个原因是Java中创建子进程采用了fork()+exec()的方案，子进程启动瞬间，它使用的内存量与父进程一致，从外面看来，一个进程使用内存量可能瞬间翻倍，然后又降下来，采用线程监控的方法可防止这种情况下导致swap操作。对于CPU资源，则采用了Cgroups进行资源隔离，将在后续章节深入讨论。

在YARN中，用户以队列的形式组织，每个用户可属于一个或多个队列，且只能向这些队列中提交application。每个队列被划分了一定比例的资源。 YARN的资源分配过程是异步的，也就是说，资源调度器将资源分配给一个application后，不会立刻push给对应的ApplicaitonMaster，而是暂时放到一个缓冲区中，等待ApplicationMaster通过周期性的RPC函数主动来取，也就是说，采用了pull-based模型，而不是push-based模型。

**YARN资源隔离**

在YARN中，资源管理由ResourceManager和NodeManager共同完成，其中，ResourceManager中的调度器负责资源的分配，而NodeManager则负责资源的供给和隔离。ResourceManager将某个NodeManager上资源分配给任务（这就是所谓的“资源调度”）后，NodeManager需按照要求为任务提供相应的资源，甚至保证这些资源应具有独占性，为任务运行提供基础的保证，这就是所谓的资源隔离。

**YARN中内存资源的调度和隔离**

基于以上考虑，YARN允许用户配置每个节点上可用的物理内存资源，注意，这里是“可用的”，因为一个节点上的内存会被若干个服务共享，比如一部分给YARN，一部分给HDFS，一部分给Hbase等，YARN配置的只是自己可以使用的，配置参数如下：

（1）yarn.nodemanager.resource.memory-mb

表示该节点上YARN可使用的物理内存总量，默认是8192（MB），注意，如果你的节点内存资源不够8GB，则需要调减小这个值，而YARN不会智能的探测节点的物理内存总量。

（2）yarn.nodemanager.vmem-pmem-ratio

任务每使用1MB物理内存，最多可使用虚拟内存量，默认是2.1。

（3） yarn.nodemanager.pmem-check-enabled

是否启动一个线程检查每个任务正使用的物理内存量，如果任务超出分配值，则直接将其杀掉，默认是true。

（4） yarn.nodemanager.vmem-check-enabled

是否启动一个线程检查每个任务正使用的虚拟内存量，如果任务超出分配值，则直接将其杀掉，默认是true。

（5）yarn.scheduler.minimum-allocation-mb

单个任务可申请的最少物理内存量，默认是1024（MB），如果一个任务申请的物理内存量少于该值，则该对应的值改为这个数。

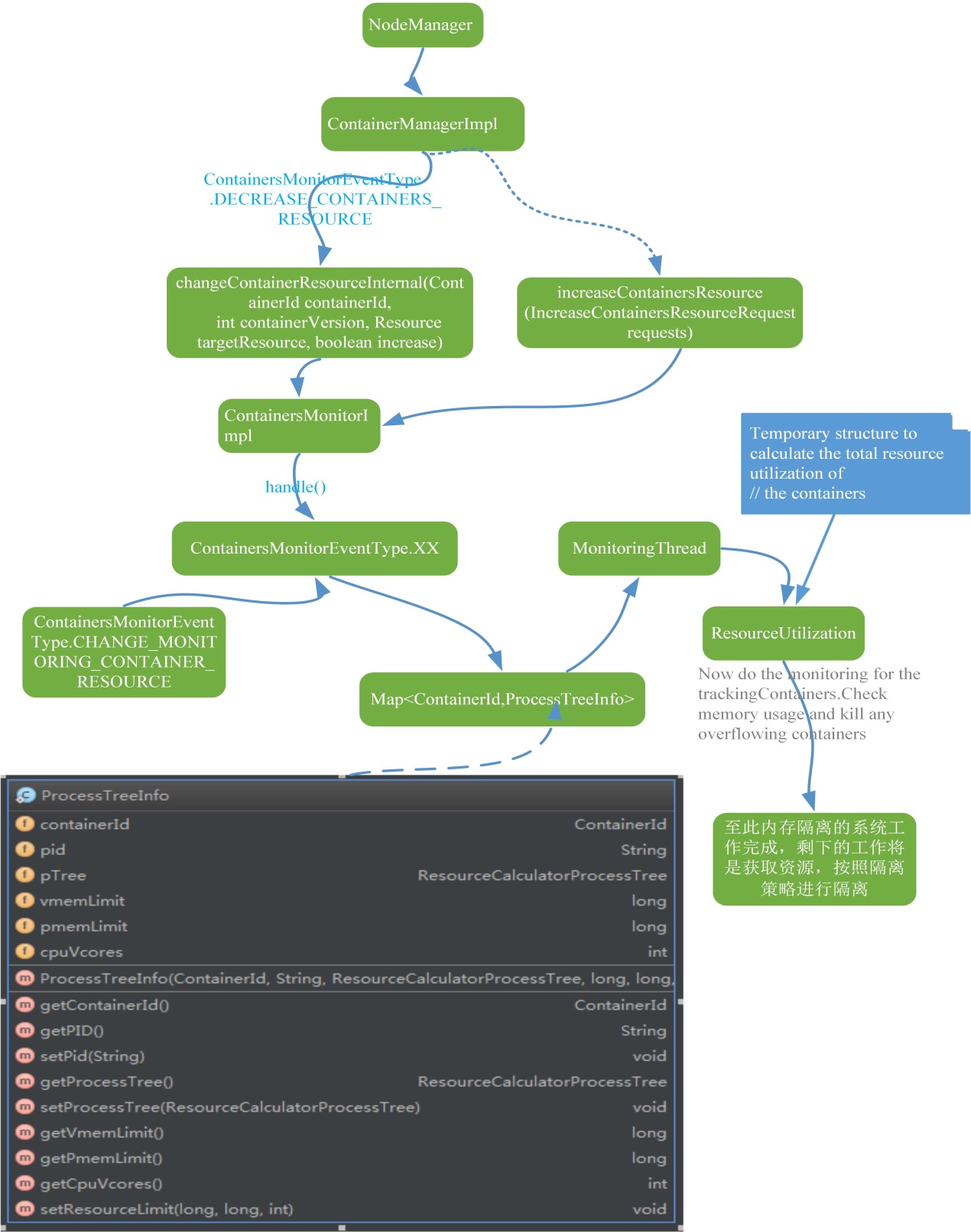
（6）yarn.scheduler.maximum-allocation-mb

单个任务可申请的最多物理内存量，默认是8192（MB）。

默认情况下，YARN采用了线程监控的方法判断任务是否超量使用内存，一旦发现超量，则直接将其杀死。由于Cgroups对内存的控制缺乏灵活性（即任务任何时刻不能超过内存上限，如果超过，则直接将其杀死或者报OOM），而Java进程在创建瞬间内存将翻倍，之后骤降到正常值，这种情况下，采用线程监控的方式更加灵活（当发现进程树内存瞬间翻倍超过设定值时，可认为是正常现象，不会将任务杀死），因此YARN未提供Cgroups内存隔离机制。

**基于线程监控的内存隔离实现**

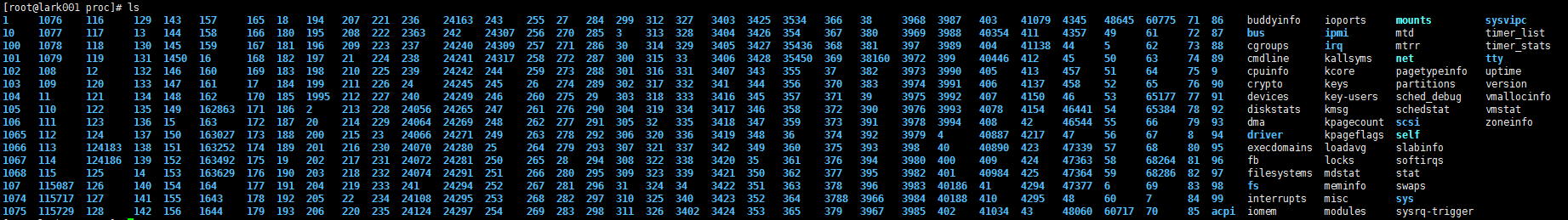
Application的内存监控是由服务ContainersMonitorImpl实现的，它保存了每个Container进程PID，这样它内部的MonitoringThread线程每个一段时间（由参数yarn.nodemanager.container-monitor.interval-ms指定，默认3000ms）扫描所有正在运行的Container进程树，并按照以下步骤检查它们使用的内存量是否超过上限值。ContainersMonitorImpl由NodeManager初始化ContainersManager后由ContainersManager初始化，而后根据触发的ContainersMonitorEventType类型的事件和函数调用完成资源隔离操作的初始化和资源限制对象的生成，此后进入到读取系统进程资源，对资源利用率等进行计算，检查是否超限以及超限处理流程：



**步骤1 构造进程树**

在YARN中每个Container可以认为是一个独立的进程，同时，Container也可能会创建子进程（可能会创建多个子进程，这些子进程可能也会创建子进程），因此Container进程的内存（物理内存、虚拟内存）使用量应该表示为：以Container进程为根的进程树中所有进程的内存（物理内存、虚拟内存）使用总量。

在Linux系统中的 /proc目录下，有大量以整数命名的目录，这些整数是某个正在运行的进程的PID，如下图所示：



而目录/proc/<PID>下面的那些文件分别表示着进程运行时的各方面信息，可参考Linux系统方面的书籍了解。



YARN只关心/proc/<PID>/stat文件。

打开一个/proc/<PID>/stat文件你会发现它仅有一行（多列）文本：

*60717 (rpc.statd) S 1 60717 60717 0 -1 4202816 1262 0 0 0 3 0 0 0 20 0 1 0 412627997 75718656 12987 18446744073709551615 139780868575232 139780868643356 140737412651392 140737412650712 139780856985091 0 0 16846848 18947 18446744071580587289 0 0 17 6 0 0 0 0 0*

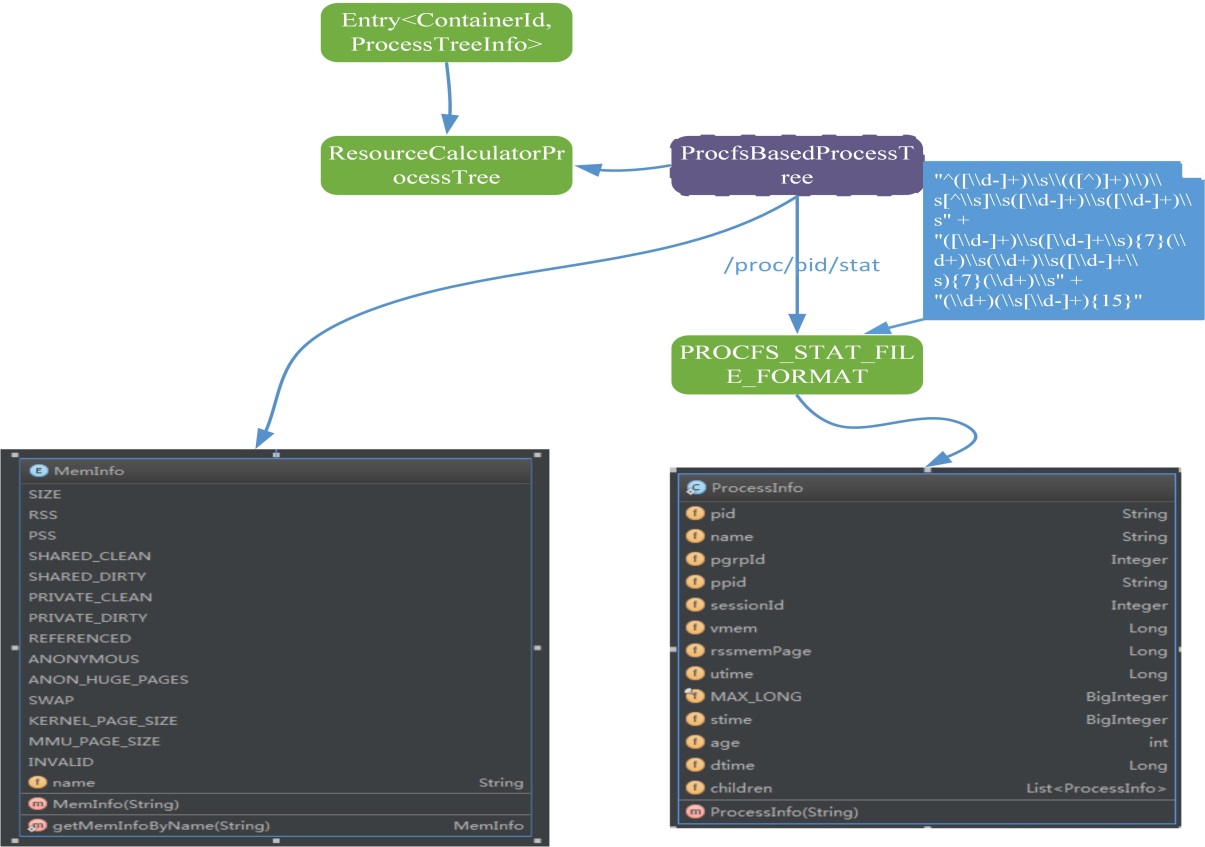
里面记录了该进程相关的信息，可以通过正则表达式*"^([\\d-]+)\\s\\(([^)]+)\\)\\s[^\\s]\\s([\\d-]+)\\s([\\d-]+)\\s+([\\d-]+)\\s([\\d-]+\\s){7}(\\d+)\\s(\\d+)\\s([\\d-]+\\s){7}(\\d+)\\s +(\\d+)(\\s[\\d-]+){15}"*从中抽取进程的运行时信息，包括：进程名称、父进程PID、父进程用户组ID、Session ID在用户态运行的时间（单位：jiffies）、核心态运行的时间（单位：jiffies）、占用虚拟内存大小（单位：page）和占用物理内存大小（单位：page）等。

文件/proc/<PID>/stat中包含的内存大小单位为page，为了获取以字节为单位的内存信息，可通过执行以下Shell脚本获取每个page对应的内存量（单位：B）：

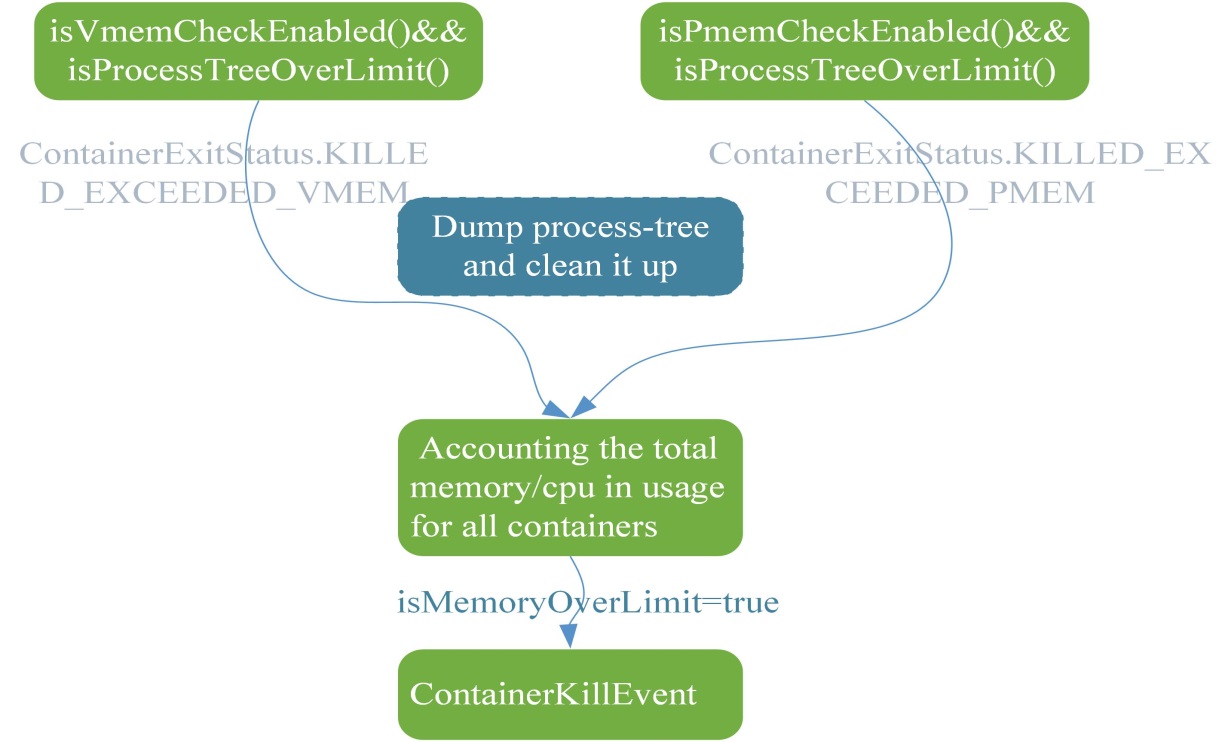
*[root@lark001 60717]# getconf PAGESIZE*

*4096*

通过对以上信息的计算可以得到当前每个运行的Container使用的内存总量。在ContainersMonitorImpl内部维护着每个Container进程的PID，通过遍历/proc下各个进程的stat文件内容（父进程PID、占用虚拟内存大小和占用物理内存大小），YARN构建出每个Container的进程树，从而得出每个进程树的虚拟内存、物理内存使用总量。



**步骤2 判断Container进程树的内存使用量（物理内存、虚拟内存）是否超过上限值**



在获取到每一个Container之后已经可以获得各个Container进程树的内存（物理内存、虚拟内存）使用量，YARN并不会仅凭进程树的内存使用量（物理内存、虚拟内存）是否超过上限值就决定是否“杀死”一个Container，因为“子进程”的内存使用量是有“波动”的(Java中创建子进程采用了fork()+exec()的方案，子进程启动瞬间，它使用的内存量与父进程一致，从外面看来，一个进程使用内存量可能瞬间翻倍，然后又降下来)，为了避免“误杀”的情况出现，Hadoop赋予每个进程“年龄”属性，并规定刚启动进程的年龄是1，MonitoringThread线程每更新一次，各个进程的年龄加一(如上图中ProcessInfo中的age属性)，在此基础上，选择被“杀死”的Container的标准如下：如果一个Contaier对应的进程树中所有进程（年龄大于0）总内存（物理内存或虚拟内存）使用量超过上限值的两倍；或者所有年龄大于1的进程总内存（物理内存或虚拟内存）使用量超过上限值，则认为该Container使用内存超量，则Container发送ContainerEventType.KILL\_CONTAINER事件将其KILL掉。

**核心代码分析：**

**MonitoringThread**

线程监控的核心工作主要是由MonitoringThread（org.apache.hadoop.yarn. server.nodemanager.containermanager.monitor.ContainersMonitorImpl.MonitoringThread）完成的，内部就是一个“while”循环，以指定的时间间隔进行监控其中，时间间隔monitoringInterval由参yarn.nodemanager.container-monitor.interval-ms指定，默认值：3000，单位：ms。

**将新启动的Container加入监控列表以及将已完成的Container移出监控列表**

每次监控开始之前都需要更新监控列表：trackingContainers，将新启动的Container加入监控列表，在2.7.X及以前的版本由containersToBeAdded表示；将已完成的Container移出监控列表，由containersToBeRemoved表示。

*// Add new containers*

*synchronized (containersToBeAdded) {*

*for (Entry<ContainerId, ProcessTreeInfo> entry : containersToBeAdded*

*.entrySet()) {*

*ContainerId containerId = entry.getKey();*

*ProcessTreeInfo processTreeInfo = entry.getValue();*

*LOG.info("Starting resource-monitoring for " + containerId);*

*trackingContainers.put(containerId, processTreeInfo);*

*}*

*containersToBeAdded.clear();*

*}*

*// Remove finished containers*

*synchronized (containersToBeRemoved) {*

*for (ContainerId containerId : containersToBeRemoved) {*

*trackingContainers.remove(containerId);*

*LOG.info("Stopping resource-monitoring for " + containerId);*

*}*

*containersToBeRemoved.clear();*

*}*

containersToBeAdded和containersToBeRemoved都是通过“事件”由org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl.handle负责更新的，如下：

*ContainerId containerId = monitoringEvent.getContainerId();*

*switch (monitoringEvent.getType()) {*

*case START\_MONITORING\_CONTAINER:*

*ContainerStartMonitoringEvent startEvent =*

*(ContainerStartMonitoringEvent) monitoringEvent;*

*synchronized (this.containersToBeAdded) {*

*ProcessTreeInfo processTreeInfo =*

*new ProcessTreeInfo(containerId, null, null,*

*startEvent.getVmemLimit(), startEvent.getPmemLimit());*

*this.containersToBeAdded.put(containerId, processTreeInfo);*

*}*

*break;*

*case STOP\_MONITORING\_CONTAINER:*

*synchronized (this.containersToBeRemoved) {*

*this.containersToBeRemoved.add(containerId);*

*}*

而在2.8.X版本中则完全使用事件模型驱动：

*switch (monitoringEvent.getType()) {*

*case START\_MONITORING\_CONTAINER:*

*ContainerStartMonitoringEvent startEvent =*

*(ContainerStartMonitoringEvent) monitoringEvent;*

*LOG.info("Starting resource-monitoring for " + containerId);*

*updateContainerMetrics(monitoringEvent);*

*trackingContainers.put(containerId,*

*new ProcessTreeInfo(containerId, null, null,*

*startEvent.getVmemLimit(), startEvent.getPmemLimit(),*

*startEvent.getCpuVcores()));*

*break;*

*case STOP\_MONITORING\_CONTAINER:*

*LOG.info("Stopping resource-monitoring for " + containerId);*

*updateContainerMetrics(monitoringEvent);*

*trackingContainers.remove(containerId);*

*break;*

*case CHANGE\_MONITORING\_CONTAINER\_RESOURCE:*

*ChangeMonitoringContainerResourceEvent changeEvent =*

*(ChangeMonitoringContainerResourceEvent) monitoringEvent;*

*ProcessTreeInfo processTreeInfo = trackingContainers.get(containerId);*

*if (processTreeInfo == null) {*

*LOG.warn("Failed to track container "*

*+ containerId.toString()*

*+ ". It may have already completed.");*

*break;*

*}*

对于事件START\_MONITORING\_CONTAINER，它表示有新的Container进程，为其构建一个ProcessTreeInfo实例，用于保存Container的进程树信息，也就是说，这里考虑的不仅仅是Container进程，而是以Container进程为父进程的整个进程树，构造函数参数含义依次如下：

containerId：Container ID；

pid：Container进程的PID；

pTree：Container进程树内存使用量计算器实例，不同的Hadoop运行平台（Windows、Linux）因为统计内存使用量的方式不同，因此需要不同的计算器实例；通过该计算器实例，可以获得当前Container进程树的内存使用量；

vmemLimit：Container进程树可使用的虚拟内存上限值；

pmemLimit：Container进程树可使用的物理内存上限值；

注意：pid、pTree的初始值为Null。

更新监控列表trackingContainers之后，下一步就是对监控列表中的Container进程树的内存使用量进行监控。

**遍历监控列表trackingContainers，逐个处理其中的进程树；**

*for (Entry<ContainerId, ProcessTreeInfo> entry : trackingContainers*

*.entrySet()) {*

*ContainerId containerId = entry.getKey();*

*ProcessTreeInfo ptInfo = entry.getValue();*

*try {*

*String pId = ptInfo.getPID();*

可以看出，监控列表trackingContainers中的每一个进程树元素是由ContainerId和ProcessTreeInfo共同表示的。

下面介绍单独一个进程树的内存监控过程。

初始化进程树信息ProcessTreeInfo；

进程树监控列表trackingContainers是被不断更新的，而新加入监控的Container进程树信息是由ProcessTreeInfo表示的，

其中pid、pTree的初始值为Null，因此监控过程中如果发现进程树信息ProcessTreeInfo的pid、pTree为Null，要对其进行初始化。

（1）获取进程树元素，由containerId和ptInfo表示；

（2）判断如果ptInfo（进程树信息）中的pId（Container进程的PID）为null，则表示需要初始化ptInfo；

（3）获取ProcessTreeInfo pid，将其保存至pId；

Container进程PID（pid）可以通过ContainerId（ptInfo.getContainerId()）从ContainerExecutor（containerExecutor）中获取；如果获取不到相应的PID，可能是因为Container进程尚没有被启动或者ContainerExecutor已将其移除，也意味着此进程树无需监控。

（4）获取ProcessTreeInfo pTree，将其保存至pt；

这里需要介绍一下ResourceCalculatorProcessTree（org.apache.hadoop.yarn.util.ResourceCalculatorProcessTree）的作用。

每一次对ProcessTreeInfo进行监控时，我们都必须获取该进程树内所有进程的运行状态（这里我们仅关心物理、虚拟内存使用情况等），也就是说，我们需要一个“计算器”，能够将进程树内所有进程的运行状态计算出来，ResourceCalculatorProcessTree就是用来充当“计算器”角色的，如下注释所示：

ResourceCalculatorProcessTree是一个抽象类，也就意味着它可以有多种实现，具体选取哪一种实现取决于ResourceCalculatorProcessTree.getResourceCalculatorProcessTree：

其中，processTreeClass由参数yarn.nodemanager.container-monitor.process-tree.class指定，默认值为null。

因为传入的参数clazz值为null，所以我们仅仅关注上图红色箭头所指的逻辑即可。

ProcfsBasedProcessTree和WindowsBasedProcessTree分别对应着ResourceCalculatorProcessTree在Linux平台和Windows平台的实现，通常我们关注ProcfsBasedProcessTree即可，也就是说，Linux平台下pTree的实例类型为ProcfsBasedProcessTree。

（5）将pId、pt更新至ptInfo，初始化过程完成；

根据ResourceCalculatorProcessTree（ProcfsBasedProcessTree）更新进程树的运行状态（这里仅关注物理、虚拟内存），并获取相关的监控信息；

（1）获取当前进程树的ResourceCalculatorProcessTree实例pTree，并更新其内部状态updateProcessTree()，实际就是更新进程树中的进程信息（详细处理逻辑见后）；

（2）获取当前进程树中所有进程的虚拟内存使用总量（currentVmemUsage）、物理内存使用总量（currentPmemUsage）；

（3）获取当前进程树中所有年龄大于1的进程的虚拟内存使用总量（curMemUsageOfAgedProcesses）、物理内存使用总是（curRssMemUsageOfAgedProcesses）；

（4）获取当前进程树的虚拟内存使用总量上限值（vmemLimit）、物理内存使用总量上限值（pmemLimit）；

判断进程树的内存使用量是否超过上限值，虚拟内存与物理内存需要分别处理；

isMemoryOverLimit的值用于表示进程树的内存使用量是否超过上限值，值为true表示超量（虚拟内存或物理内存两者至少有其一超量）；值为false表示未超量（虚拟内存和物理内存两者均未超量）；初始值设置为false。

（1）如果开启虚拟内存监控，则判断进程树虚拟内存使用总量是否超过其上限值；

（2）如果开启物理内存监控，则判断进程树物理内存使用总量是否超过其上限值；

虚拟、物理内存监控选项的开启分别由参数yarn.nodemanager.vmem-check-enabled、yarn.nodemanager.pmem-check-enabled指定，默认值均为true，表示两者均开启监控。

判断虚拟、物理内存使用总量是否超过上限值由isProcessTreeOverLimit()（详细处理逻辑见后）统一处理，两者仅传入的参数值不同，参考上图代码。

如果isMemoryOverLimit值为true，则表示进程树的内存使用量超量（或者虚拟内存、或者物理内存），执行“kill”并从监控列表移除；

至此，进程树内存使用总量监控处理逻辑完成。

ResourceCalculatorProcessTree(ProcfsBasedProcessTree) updateProcessTree

updateProcessTree用于更新当前Container进程的进程树：

（1）获取所有的进程列表；

其中，procfsDir的值为/proc/，numberPattern表示的正则表达式为[1-9][0-9]\*（用于匹配进程PID）。对于Linux系统而言，所以运行着的进程都对应着目录“/proc/”下的一个子目录，子目录名称即为进程PID，子目录中包含着进程的运行时信息。所谓的进程列表，实际就是Linux目录“/proc/”下的这些进程子目录名称。

进程列表processList包含的信息：1、10、100、...。

（2）更新进程树processTree；

因为Container进程树中的进程随时都可能启动或停止，因此每次监控开始之前都需要更新该Container进程的进程树；而且为了方便处理进程的年龄（加一），将该Container进程“旧”的进程树processTree缓存至oldProcs，然后清空processTree（详情见后）。

（3）遍历（1）中进程列表，为每一个进程构建ProcessInfo，并将其保存至allProcessInfo；

ProcessInfo的构建过程由方法constructProcessInfo()完成，处理逻辑很简单：

a.读取“procfsDir/<pid>/stat”（即“/proc/<pid>/stat”）的文件内容，实际内容只有一行；

b.通过正则表达式抽取其中的信息，并更新至pInfo；

可以看出，ProcessInfo保存着一个进程的以下信息：

name：进程名称；

ppid：父进程PID；

pgrpId：父进程所属用户组ID；

session：进程所属会话组ID；

utime：进程用户态占用时间；

stime：进程内核态占用时间；

vsize：进程虚拟内存使用量；

rss：进程物理内存使用量；

遍历构建的过程中，如果发现“me”进程（即当前的Container进程），则将该进程保存至进程树processTree，因为当前的Container进程必须是此Container进程树中的一员；如果没有发现“me”进程，则表示Container进程（树）已经运行结束，无需监控。

（4）维护进程之间的父子关系；

allProcessInfo中保存着所有的进程信息，其中key为PID，value为对应的ProcessInfo，我们通过ProcessInfo的ppid（父进程PID），即可以维护出这些进程之间的父子关系。

对于每一个ProcessInfo（进程）pInfo：

a.根据pInfo ppid找出其父进程的ProcessInfo：parentPInfo；

b.将pInfo加入parentPInfo的子进程列表中(ProcessInfo addChild)；

（5）构建当前Container进程（即（3）中的me）的进程树；

a.将pInfoQueue初始化为me；

b.如果pInfoQueue不为空，执行以下操作：

b1.取出pInfoQueue的头元素pInfo，将其加入进程树processTree（注意重复检测）；

b2.将pInfo的所有子进程加入pInfoQueue；

c.执行b；

上述流程执行完毕之后，processTree中保存着当前Container进程的进程树。

（6）更新当前Container进程的进程树中所有进程的年龄；

处理逻辑很简单：遍历进程树，对于其中的每一个ProcessInfo，如果它是一个“老”进程（即出现在“老”进程树oldInfo中），则将其年龄加一。（注：ProcessInfo age初始值为一）

到此，进程树更新完毕。

我们以虚拟内存为例说明进程树的虚拟内存使用总量是如何计算的，如下：

其实就是根据进程年龄做过滤，然后叠加ProcessInfo中的相关值（虚拟内存：vmem）。

ContainersMonitorImpl.isProcessTreeOverLimit

isProcessTreeOverLimit用于判断内存使用量是否超过上限值，虚拟内存和物理内存共用此方法。

currentMemUsage：进程树中所有进程的虚拟或物理内存使用总量；

curMemUsageOfAgedProcesses：进程树中所有年龄大于1的进程的虚拟或物理内存使用总量；

vmemLimit：进程树虚拟或物理内存使用上限；

满足以下二个条件之一，则认为进程树内存使用超过上限：

（1）currentMemUsage大于vmemLimit的两倍，这样做的目录主要是为了防止误判；

（2）curMemUsageOfAgedProcesses大于vmemLimit（年龄大于1的进程可以认内存使用比较“稳定”）。

**YARN中CPU资源的调度和隔离**

在YARN中，CPU相关配置参数如下：

（1）yarn.nodemanager.resource.cpu-vcores

表示该节点上YARN可使用的虚拟CPU个数，默认是8，注意，目前推荐将该值设值为与物理CPU核数数目相同。如果你的节点CPU核数不够8个，则需要调减小这个值，而YARN不会智能的探测节点的物理CPU总数。

（2） yarn.scheduler.minimum-allocation-vcores

单个任务可申请的最小虚拟CPU个数，默认是1，如果一个任务申请的CPU个数少于该数，则该对应的值改为这个数。

（3）yarn.scheduler.maximum-allocation-vcores

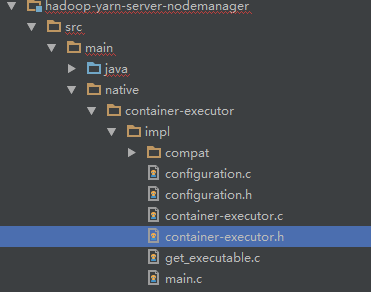
单个任务可申请的最多虚拟CPU个数，默认是32。

默认情况下，NodeManager未启用任何对CPU资源的隔离机制，如果需要启用该机制需使用LinuxContainerExecutor，它能够以应用程序提交者的身份创建文件、运行Container和销毁Container。相比于DefaultContainerExecutor采用NodeManager启动者的身份执行这些操作，LinuxContainerExecutor具有更高的安全性。

LinuxContainerExecutor的核心设计思想是赋予NodeManager启动者以***root***权限，进而使其拥有足够的权限以任意用户身份执行一些操作，从而使得NodeManager执行者可以将Container使用的目录和文件的拥有者修改为应用程序的提交者，并以应用程序提交者的身份运行Container，防止所有Container以NodeManager执行者身份运行进而带来的各种风险。

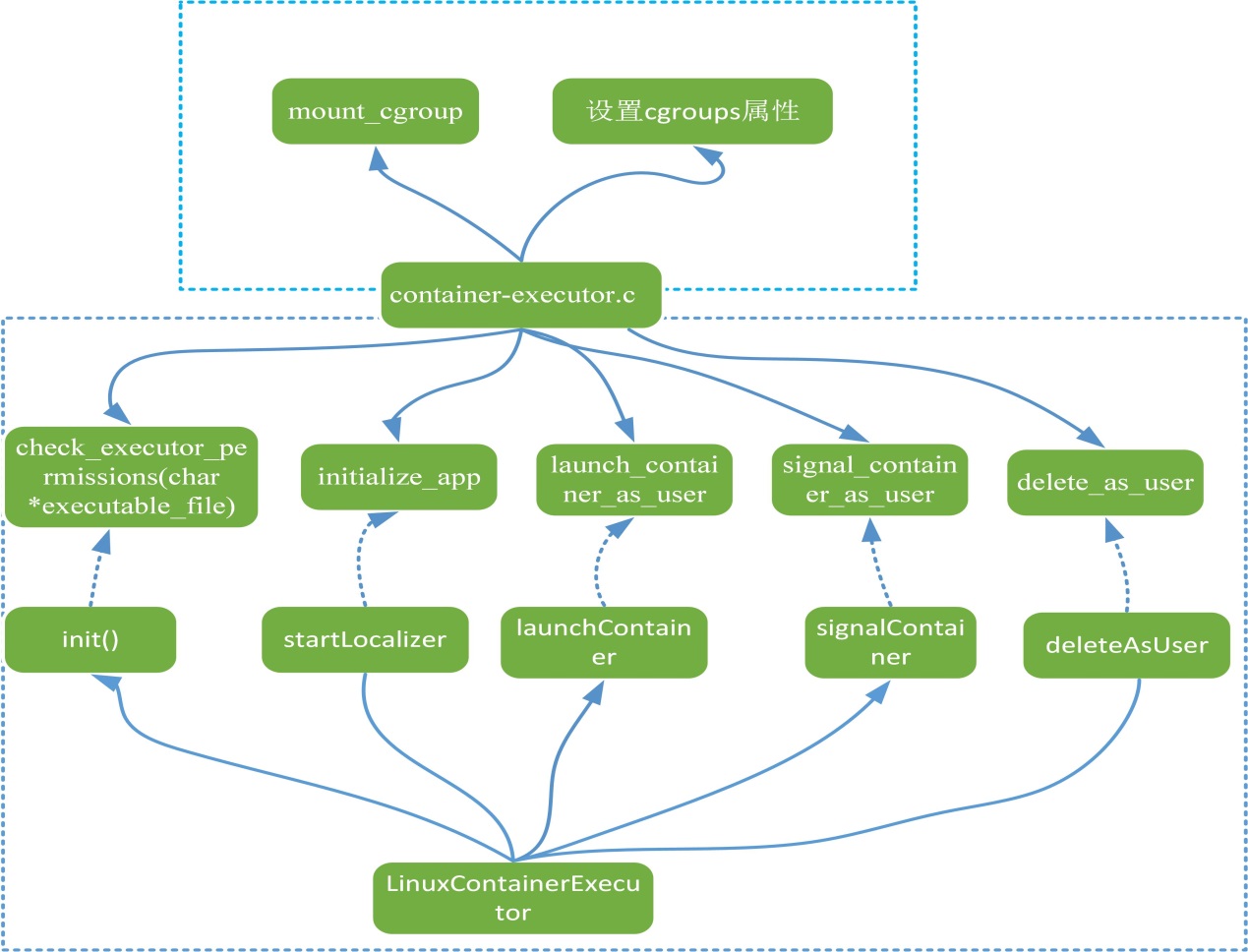
上述机制的实现在YARN的NodeManager采用C语言实现了一个setuid功能的工具container-executor，其位于如下目录：、





该脚本拥有***root***权限，可以完成任意操作，其可执行脚本位于：

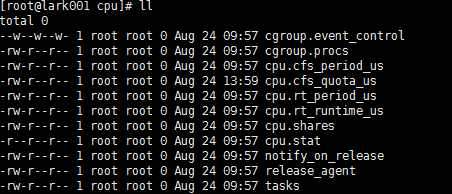
/opt/yarn/hadoop/bin/container-executor。YARN正是通过该脚本创建出Cgroups层级树以及完成Cgroups属性设置等操作。相对应的LinuxContainerExecutor正是通过调用该可执行文件的先关操作来对Container的一些属性进行修改已达到限制Container的非法操作（比如：关闭NodeManager、杀死NodeM等）。其对应关系如下图：



下面将具体分析其实现:

**1操作文件**

首先需要了解的是Cgroups的应用和实现机制，再次不做赘述，既然是对CPU资源的隔离，那么我们只关注CPU文件，其位于：/cgroup/cpu下我们可以看到有以下文件：



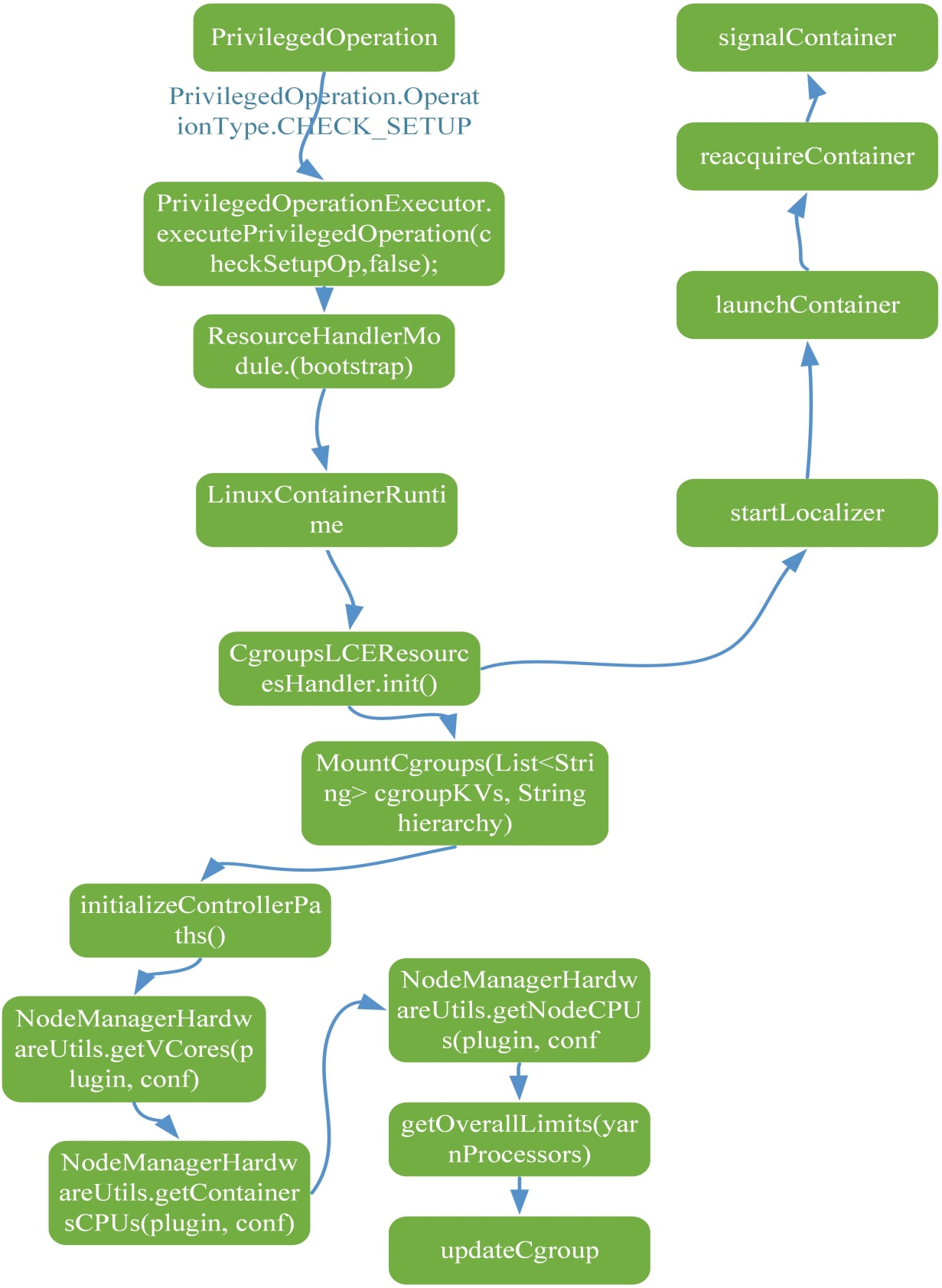
YARN关注的是cpu.cfs\_period\_us和cpu.cfs\_quota\_us文件，

cpu.cfs\_period\_us：设定周期时间，必须与cfs\_quota\_us配合使用。

cpu.cfs\_quota\_us ：设定周期内最多可使用的时间。

这里的配置指task对单个cpu的使用上限，若cfs\_quota\_us是cfs\_period\_us的两倍，就表示在两个核上完全使用。数值范围为1000 - 1000,000（微秒）。YARN正是通过对这两个文件的指的限制达到限制CPU资源的目的。由CgroupsLCEResourcesHandler实现对资源的处理。

2 LCX操作



在介绍LinuxContainerExecutor的核心设计思想时指出：LinuxContainer -Executor通过赋予NodeManager启动者以***root***权限，进而使其拥有足够的权限以任意用户身份执行一些操作，从而使得NodeManager执行者可以将Container使用的目录和文件的拥有者修改为应用程序的提交者，并以应用程序提交者的身份运行Container，防止所有Container以NodeManager执行者身份运行进而带来的各种风险。

因此LCE的第一步操作即是定义特殊权限PrivilegedOperation和其操作者PrivilegedOperationExecutor，PrivilegedOperation定义了LCE支持的特殊权限：

*CHECK\_SETUP("--checksetup"),*

*MOUNT\_CGROUPS("--mount-cgroups"),*

*INITIALIZE\_CONTAINER(""), //no CLI switch supported yet*

*LAUNCH\_CONTAINER(""), //no CLI switch supported yet*

*SIGNAL\_CONTAINER(""), //no CLI switch supported yet*

*DELETE\_AS\_USER(""), //no CLI switch supported yet*

*LAUNCH\_DOCKER\_CONTAINER(""), //no CLI switch supported yet*

*TC\_MODIFY\_STATE("--tc-modify-state"),*

*TC\_READ\_STATE("--tc-read-state"),*

*TC\_READ\_STATS("--tc-read-stats"),*

*ADD\_PID\_TO\_CGROUP(""), //no CLI switch supported yet.*

*RUN\_DOCKER\_CMD("--run-docker");*

PrivilegedOperationExecutor通过定时的操作类型达到特殊控制的目的。之后LCE要进行的操作就是资源的管理，是通过CgroupsLCEResourcesHandler实现，结合PrivilegedOperation实现对如下文件的访问：

MTAB\_FILE = "/proc/mounts";

CONTROLLER\_CPU = "cpu";

CPU\_PERIOD\_US = "cfs\_period\_us";

CPU\_QUOTA\_US = "cfs\_quota\_us";

实现对Cgroup属性访问和修改以及对资源限制的验证并最终执行PrivilegedOperation.OperationType.DELETE\_AS\_USER操作，调用container-executor 可执行脚本删除NodeManager。