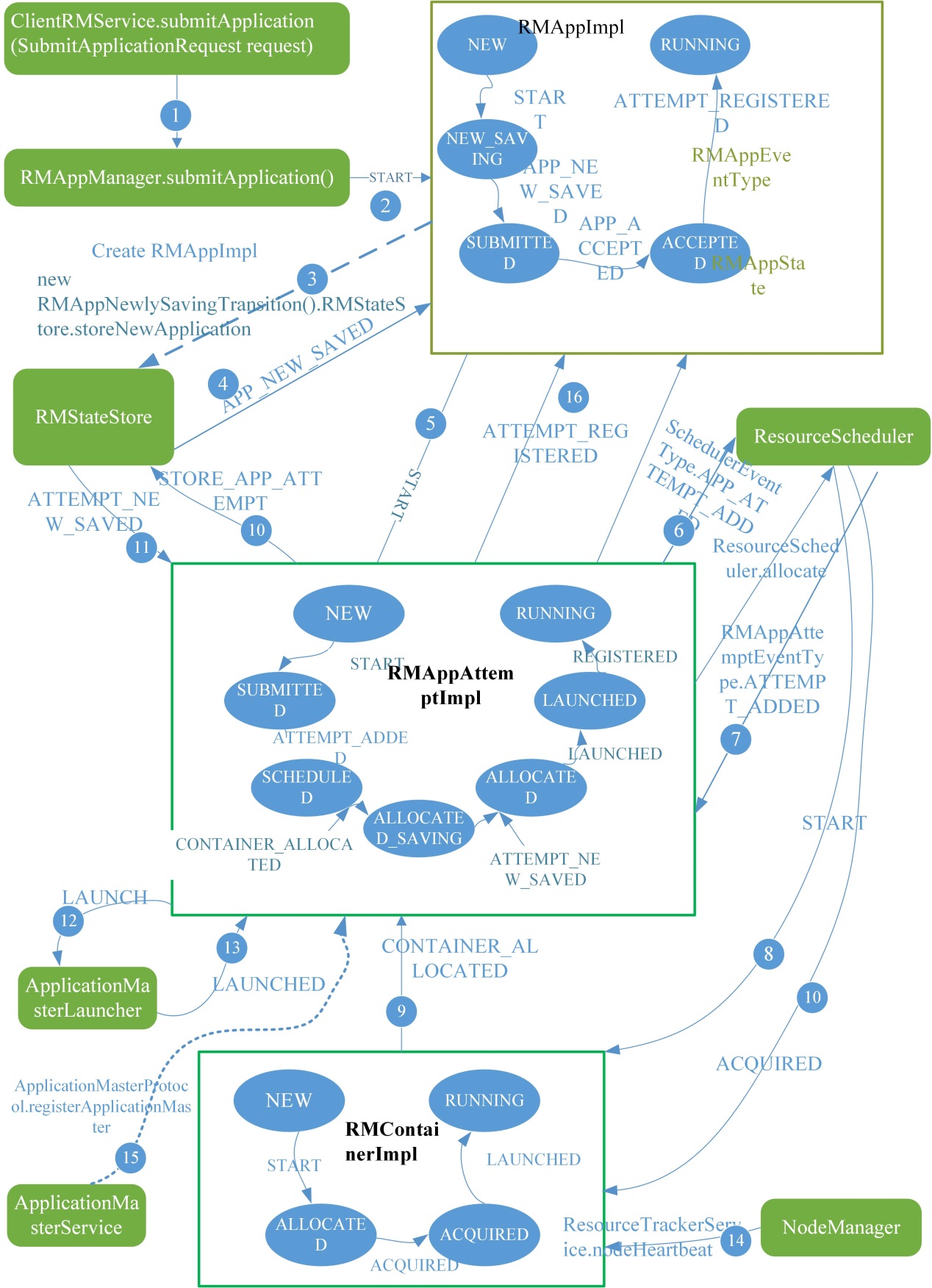
ApplicationMaster启动过程分析：

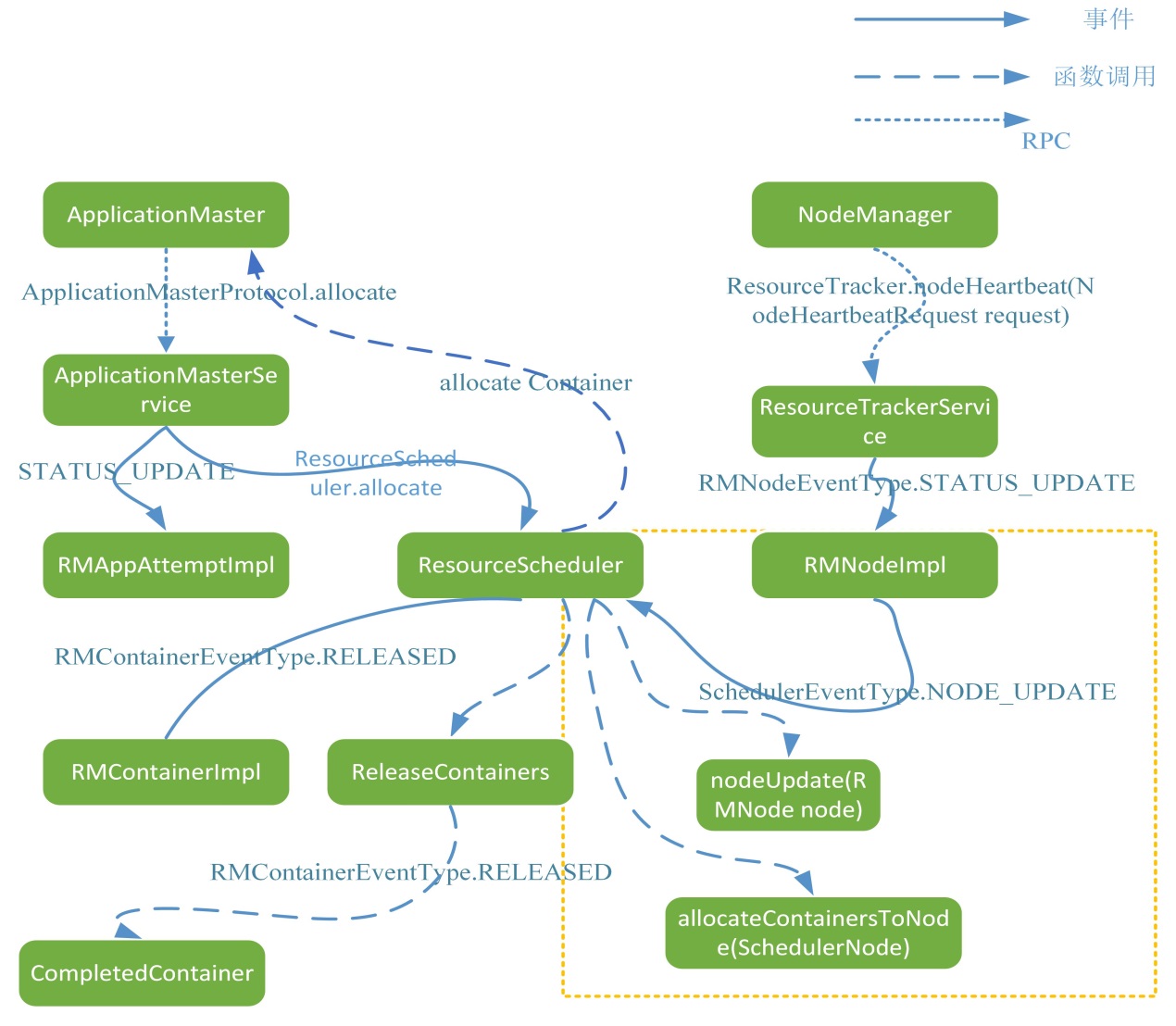


ApplicationMaster的启动过程涉及到：ResourceManager、ClientRMService、RMAppManager、ResourceScheduler、ApplicationMasterLauncher、ApplicationMasterService、RMNode几个组件和RMAppImpl、RMAppAttemptImpl、AMLauncher、RMContainerImpl 、RMNodeImpl、RMStateStore等几个状态机组件。整个流程从客户端程序调用RPC函数ApplicationClientProtocol. submitApplication (SubmitApplicationRequest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

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。客户端通过一个SubmitApplicationRequest 对象提交应用程序，并从SubmitApplicationRequest 对象中获取一个ApplicationSubmissionContext对象。此处会对独立于RM配置的属性进行校验，那些依赖于RM的属性将交由RMAppManager进行验证。之后会继续调用RMAppManager的submitApplication (ApplicationSubmissionContext submissionContext, long submitTime,String user) 作进一步处理；
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MemoryRMStateStore、FileSystemRMStateStore等)，进一步调用RMStateStore.storeNewApplication(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StoreAppTransition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AMLauncherEventType.LAUNCH事件，至此RMAppAttemptImpl状态从RMAppAttemptState.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RMAppAttemptEventType.LAUNCHED事件。RMAppAttemptImpl收到RMAppAttemptEventType.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RMContainerEventType.LAUNCHED事件，RMContainerImpl接收到该事件后会从ContainerAllocationExpirer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RMAppAttemptImpl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。

**申请与分配Container**

****

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" \o "Docker知识库" \t "_blank)，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" \o "算法与数据结构知识库" \t "_blank)，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

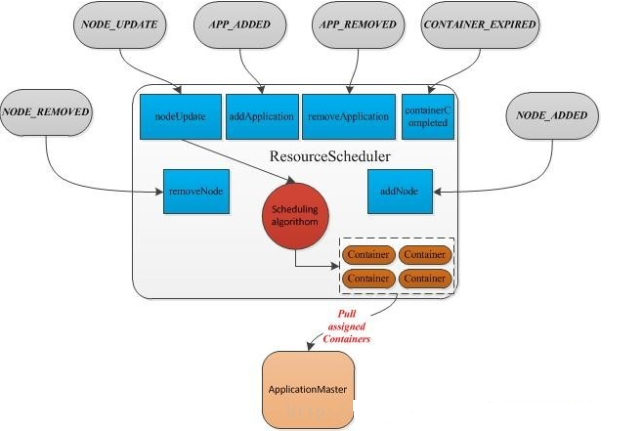
事件APP\_REMOVED表示一个Application运行结束（可能成功或者失败），资源管理器需将该Application从相应的数据结构中清除。

（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。

资源分配模型

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。

【YARN中CPU资源的调度和隔离】

在YARN中，CPU资源的组织方式仍在探索中，目前（2.2.0版本）只是一个初步的，非常粗粒度的实现方式，更细粒度的CPU划分方式已经提出来了，正在完善和实现中。

目前的CPU被划分成虚拟CPU（CPU virtual Core），这里的虚拟CPU是YARN自己引入的概念，初衷是，考虑到不同节点的CPU性能可能不同，每个CPU具有的计算能力也是不一样的，比如某个物理CPU的计算能力可能是另外一个物理CPU的2倍，这时候，你可以通过为第一个物理CPU多配置几个虚拟CPU弥补这种差异。用户提交作业时，可以指定每个任务需要的虚拟CPU个数。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，YARN是不会对CPU资源进行调度的，你需要配置相应的资源调度器让你支持，具体可参考我的这两篇文章：

（1）Hadoop YARN配置参数剖析（4）—Fair Scheduler相关参数

（2）Hadoop YARN配置参数剖析（5）—Capacity Scheduler相关参数

默认情况下，NodeManager不会对CPU资源进行任何隔离，你可以通过启用Cgroups让你支持CPU隔离。

由于CPU资源的独特性，目前这种CPU分配方式仍然是粗粒度的。举个例子，很多任务可能是IO密集型的，消耗的CPU资源非常少，如果此时你为它分配一个CPU，则是一种严重浪费，你完全可以让他与其他几个任务公用一个CPU，也就是说，我们需要支持更粒度的CPU表达方式。

借鉴亚马逊EC2中CPU资源的划分方式，即提出了CPU最小单位为EC2 Compute Unit（ECU），一个ECU代表相当于1.0-1.2 GHz 2007 Opteron or 2007 Xeon处理器的处理能力。YARN提出了CPU最小单位YARN Compute Unit（YCU），目前这个数是一个整数，默认是720，由参数yarn.nodemanager.resource.cpu-ycus-per-core设置，表示一个CPU core具备的计算能力（该feature在2.2.0版本中并不存在，可能增加到2.3.0版本中），这样，用户提交作业时，直接指定需要的YCU即可，比如指定值为360，表示用1/2个CPU core，实际表现为，只使用一个CPU core的1/2计算时间。注意，在操作系统层，CPU资源是按照时间片分配的，你可以说，一个进程使用1/3的CPU时间片，或者1/5的时间片。对于CPU资源划分和调度的探讨，可参考以下几个链接：

https://issues.apache.org/jira/browse/YARN-1089

https://issues.apache.org/jira/browse/YARN-1024