# mapreduce-on-yarn资源管理和申请源码分析


**mapreduce-on-yarn资源管理和申请源码分析**

一 概述

二 YARN 资源管理

2.1 YARN 资源保证机制

2.2 YARN 资源分配算法

2.3 YARN 资源抢占模型

2.4 YARN 资源层级队列管理机制

三 YARN **CapacityScheduler** 配置文件解析

3.1 **CapacityScheduler 配置文件 capacity-scheduler.xml** 解读

四 YARN 容器资源调度器 CapacityScheduler 源码分析

4.1 容器资源调度器 CapacityScheduler 代码结构解析

4.2 CapacityScheduler 的构造方法源码分析

4.3 CapacityScheduler 服务的 serviceInit()

4.3.1 初始化容器调度器(调用其 initScheduler())

4.3.2 资源抢占模式初始化 (调用SchedulingMonitorManager.initialize())

4.3.2.1 创建抢占资源监控 SchedulingMonitor 服务 (调用其 serviceInit())

4.4 CapacityScheduler 服务的 serviceStart()

4.4.1 启动资源抢占服务 调用 SchedulingMonitor.serviceStart()

五 YARN 资源的管理和申请源码分析

5.1 NM 上线注册上报资源和上报容器资源使用情况 (针对已完成的容器释放资源而言 这两种情况都是基于向 RM 发送心跳 NM 上线除外)

5.1.1 NM 注册上报配置资源

5.1.1.1 调用 RM 的 ResourceTrackerService.registerNodeManager() 注册 NM

5.1.1.2 最终调用 CapacityScheduler.handle(NODE_ADDED)

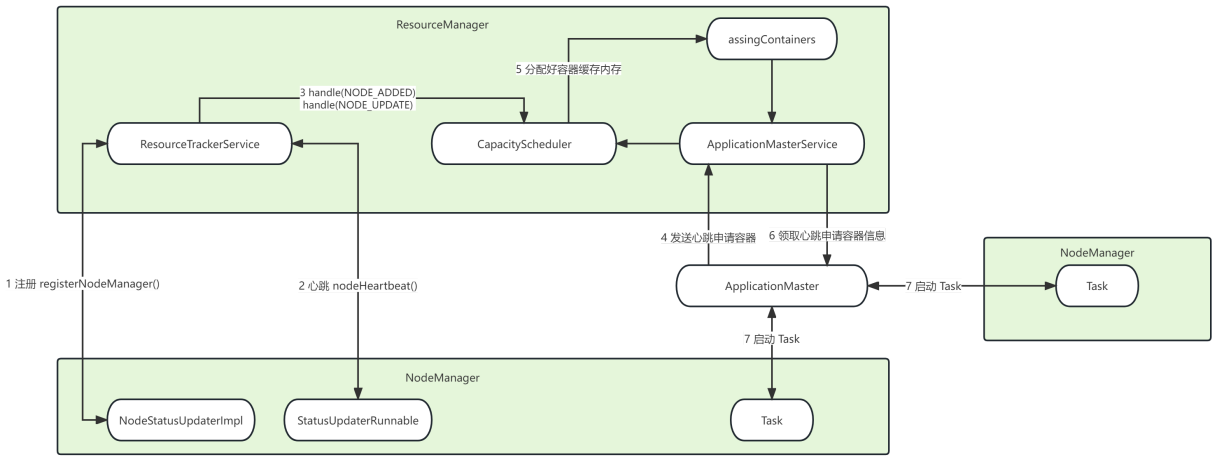
5.1.2 NM 心跳上报容器资源情况

5.1.2.1 调用 RM 的 ResourceTrackerService.nodeHeartbeat() 处理 NM 心跳

5.1.2.2 最终调用 CapacityScheduler.handle(NODE_UPDATE)

5.1.2.3 RM 分配资源给其他 Application 使用(NM 心跳上报已经完成的容器资源被 RM 回收之后)(调用 CapacityScheduler.allocateContainersToNode())

六 AM 申请容器 (调用 RM 的ApplicationMasterService.allocate())


1 NodeManager 启动通过 NodeStatusUpdateImpl 服务向 RM 注册 注册信息携带 NM 的资源信息 (配置资源信息和节点物理资源)
2 NM 通过每隔 1s 向 RM 发送心跳 上报哪些容器已经完成了 RM 需要对这些容器资源回收利用
3 RM 的容器调度器检测到 NM 心跳上报的资源信息进行容器的分配(前提是 AM 申请资源 RM 集群不足导致 AM申请容器请求处于等待状态)
4 容器调度器根据队列、应用任务、应用请求容器依次顺序匹配到最佳的 AM申请容器请求进行分配并缓存在内存数据结构中
5 AM 每隔 1s 发送心跳给 AM 的 ApplicationMasterService 服务申请容器资源 如果AM之前申请的容器被 RM 的容器调度器分配好 则返回
6 AM 通过心跳领用到自己申请的容器信息 则根据容器信息连接 NM 进行部署容器启动任务

# 一 概述

Hadoop 最初是为批处理设计而生，对于资源管理和调度，仅仅支持 FIFO 的调度机制。随着 Hadoop 的发展和流行，单个 Hadoop 集群中的用户量和应用程序类型不断增加，适用于批处理场景的 FIFO 调度机制不能很好地利用集群资源，也不能够满足不同应用程序的服务质量要求，因此需要设计适用于多用户的资源调度器

HOD（Hadoop On Demand）调度器：将物理集群，虚拟成多个 Hadoop 集群

- 优点：完全隔离
- 缺点：运维难度高，负载均衡引起的资源利用率不高，不能实现数据本地性计算特性

YARN 调度器：支持多队列多用户

- 优点：一个集群只有一套资源管理系统，内部运行的每个 Application 都能使用到集群任何节点上的资源，简而言之，将资源划分成多个队列实现资源隔离，但是每个队列都是横跨整个集群的

YARN 调度器将整个 Hadoop 集群逻辑上划分成若干个拥有相对独立资源的子集群，而由于这些子集群实际上是公用大集群中的资源，因此可以共享资源，相对于HOD 而言，提高了资源利用率且降低了运维成本

就 YARN 的整体调度来说，采用**双层资源调度模型**：

- 第一层：ResourceManager 中的资源调度器将资源分配给各个 ApplicationMaster，由 YARN 决定
- 第二层：ApplicationMaster 再进一步将资源分配给它内部的各个任务 Task，由用户应用程序 ApplicationMaster 决定

YARN 是一个统一的资源调度系统，只要满足 YARN 的调度规范的 分布式应用程序，都可以运行在YARN 中，调度规范：**自己定义一个ApplicatoinMaster，问 RM 申请资源，AM 自己来完成 Container 到 Task 的分配**

YARN 采用拉模型实现异步资源分配，资源调度器将资源分配给应用程序之后，暂存于缓冲区中，等待 ApplicationMaster 通过心跳来进行获取

# 二 YARN 资源管理

官方文档指南：https://hadoop.apache.org/docs/r3.1.3/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

在 YARN 中，资源以类似于公司组织架构的方式进行层级队列的资源管理，有利于资源在不同资源间分配和共享，进而提高集群资源利用率。Hadoop-2.x 版本中的YARN 在 MRv1 基础之上提供了三种可用资源调度器，分别是 FIFO（First In First Out）、Yahoo 的 CapacityScheduler（Hadoop 3.x 中的默认调度策略）和 Facebook 的 FairScheduler

YARN 队列管理机制由用户权限管理和系统资源管理两部分组成：

- 用户权限管理：管理员可配置每个叶子队列对应的操作系统用户和用户组，也可以配置每个队列的管理员，他可以杀死该队列中任何应用程序，改变任何应用程序的优先级等（操作系统的用户或者用户组：队列 = n:n）队列管理员
- 系统资源管理：YARN 资源管理和调度均由调度器完成，管理员可在调度器中设置每个队列的资源容量，每个用户可用资源量等信息，而调度器则按照这些资源约束对应用程序进行调度

YARN 中的所有资源通过层级队列的方式来进行管理。每个队列又可以配置多个用户，而且还可以配置管理员

总结一下 YARN 的 CapacityScheduler 的一些核心特点：

- 容量保证：给每个队列，都可以设置一个最低资源使用保证占比，提交到队列内的所有 Application 按照提交时间和优先级来共享使用这些资源
- 灵活性：如果一个队列的资源使用还没有达到上限，则可以暂时借用其他队列富余的资源使用
- 多重租赁：支持多用户共享集群和多应用程序同时运行
- 安全保证：每个队列可以指定使用它的用户。还可以指定队列管理员和系统管理员等
- 资源配置动态更新：YARN 集群管理员可以调整 capacity-scheduler.xml 资源配置，然后动态刷新。动态刷新的时候，不能删除队列

总之，两句话可以完全概括：

- YARN 的资源调度以**层级队列**（队列树）进行资源隔离划分，每个队列可以设置一定比例的**资源使用最低保证**
- 同时，YARN 也可以给每个队列设置**资源使用上限**，队列中的**每个用户也可以设置资源使用上限**，目的就是为了防止资源滥用。当一个队列的资源资源出现富余时，可以暂时借给其他队列使用。设置资源使用上限，也是为了充分发挥资源利用率

## 2.1 YARN 资源保证机制

在分布式资源调度系统中，一般资源的分配保证机制有如下两种常见方案：
1 增量资源分配：当应用程序申请的资源暂时无法保证时，系统会为应用程序预留一个节点上的资源直到累计释放的空闲资源满足应用程序需求
缺点：资源预留会导致资源浪费，降低集群资源利用率

2 一次资源分配：当应用程序申请的资源暂时无法保证时，系统放弃当前资源直到出现一个节点剩余资源一次性满足应用程序需求
缺点：应用程序可能永远等不到满足资源需求的节点出现导致饿死

YARN 采用的是增量资源分配，尽管这种机制会造成浪费，但不会出现饿死现象。正常情况，资源总会在有限时间范围内释放的

## 2.2 YARN 资源分配算法

在 YARN 默认的资源分配算法实现是：DefaultResourceCalculator 资源计算器，只负责调度内存。其实还有另外一个资源计算器：DRF

因为 YARN 支持 CPU 和 内存的资源调度分配，所以 YARN 调度器采用 DRF 这种支持多维资源调度的 **主资源公平调度算法** Dominant Resource Fairness。这种资源算法同时也被 Apache Mesos 采用。该算法扩展了 Max-Min Fairness 最大最小公平算法，非常适合应用于多资源和复杂需求的环境中

关键理解：在 DRF 算法中，将所需份额（资源比例）最大的资源称为主资源，而 DRF 的基本设计思想则是将最大最小公平算法应用于主资源上，进而将多维资源调度问题转化为单资源调度问题

> 假设系统中共有 9 个 CPU 和 18 GB RAM，有两个用户（A 和 B）（或者框架）分别运行了两种任务，需要的资源量分别为 <1CPU,4GB> 和 <3CPU,1GB>。对于用户A，每个任务要消耗总 CPU 的 1/9（份额）和总内存的 2/9，因而A的主资源为内存；对于用户 B，每个任务要消耗总 CPU 的 1/3 和总内存的 1/18，因而 B 的主资源为 CPU

| 调度序列 | A资源份额 | A主资源份额 | B资源份额 | B主资源份额 | CPU用量 | 内存用量 |
|---|---|---|---|---|---|---|
| B | | | <3/9, 1/18> | 3/9 | 3/9 | 1/18 |
| A | <1/9, 4/18> | 4/18 | | | 4/9 | 5/18 |
| A | <2/9, 8/18> | 8/18 | | | 5/9 | 8/18 |
| B | | | <6/9, 2/18> | | 8/9 | 10/18 |
| A | <3/9, 12/18> | | | | 9/9 | 14/18 |

DRF 算法是资源利用率最高的。总之一句话理解：谁是主资源，就按照谁来进行分配

## 2.3 YARN 资源抢占模型

在资源调度器中，每个队列可设置一个最小资源量和最大资源量，其中，**最小资源量是资源紧缺情况下每个队列需保证的资源量**，而**最大资源量则是极端情况下队列**也不能超过的资源使用量

- 最小资源量 X = 最低资源使用保证 = 资源再紧缺，都会保证这个队列至少有这么多的资源可用
- 最大资源量 Y = 最多资源使用上限 = 资源能使用的再多，也不能超过这个值

通常而言，为了提高资源利用率，资源调度器会将负载较轻的队列的资源暂时分配给负载重的队列，仅当负载较轻队列突然收到新提交的应用程序时，调度器才进一步将本属于该队列的资源分配给它。但由于此时资源可能正被其他队列使用，因此调度器必须等待其他队列释放资源后，才能将这些资源“物归原主”，这通常需要一段不确定的等待时间。为了防止应用程序等待时间过长，调度器等待一段时间后若发现资源并未得

到释放，则进行资源抢占

完整的工作机制流程，通过一个案例来表示：

- 队列A 和 队列B 的资源配置为：(30, 40) 和 (70,85)，分别表示最小和最大资源配置
- T1 时刻，队列A 用了 20 资源，队列B 用了 70 资源。可以得到的结论是：队列B的资源用完了。队列A 的资源有剩余。整个系统剩余资源 10
- T2 时刻，队列B 又提交了一个 Application，需要资源 10，系统剩余资源 10，刚好满足要求，此时队列A 使用资源 20，队列B 使用资源 80，系统无资源可用了
- T3 时刻，队列 A 提交了一个 Application，按照配置，队列A 本应该至少还有资源 10 可用的，但是被队列B 给占用了，则队列 A 会从 队列B 中抢占 10 个本该属于队列 A 的资源。当然，为了避免资源浪费，队列 A 首先会等待一段时间，如果这段时间内，还没有释放，则强制回收资源

资源抢占，默认不启用，可以通过 yarn.resourcemanager.scheduler.monitor.enable 参数将其设置为 true 来启用

## 2.4 YARN 资源层级队列管理机制

YARN 最开始采用 平级 队列资源管理方式。新版本的 YARN 采用**层级队列**资源管理方式

在默认情况下，有一个 /root 队列，/root 队列下有一个 default 队列

该队列组织方式具有以下特点：

- 子队列
    1. 队列可以嵌套，每个队列均可以包含子队列
    2. 用户只能将应用程序提交到最底层的队列，即叶子队列
- 最少容量
    1. 每个子队列均有一个“最少容量比”属性，表示可以使用父队列的容量的百分比
    2. 调度器总是优先选择当前资源使用率最低的队列，并为之分配资源
    3. 最少容量不是“总会保证的最低容量”，也就是说，如果一个队列的最少容量为 100，而该队列中所有队列仅使用了 20，那么剩下的 80 可能会分配给其他需要的队列
    4. 最少容量的值为不小于 0 的数，但也不能大于“最大容量”
- 最大容量
    1. 为了防止一个队列超量使用资源，可以为队列设置一个最大容量，这是一个资源使用上限，任何时刻使用的资源总量都不能超过该值
    2. 默认情况下队列的最大容量是无限大，这意味着，当一个队列只分配了 20% 的资源，所有其他队列没有应用程序时，该队列可能使用 100% 的资源，当其他队列有应用程序提交时，再逐步归还

# 三 YARN  CapacityScheduler 配置文件解析

CapacityScheduler 是 Yahoo! 开发的多用户调度器，它以队列为单位划分资源，每个队列可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用。而当一个队列的资源有剩余时，可暂时将剩余资源共享给其他队列

## 3.1  CapacityScheduler 配置文件 capacity-scheduler.xml 解读

具体参考官方指南：https://hadoop.apache.org/docs/r3.1.3/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

# 四 YARN 容器资源调度器 CapacityScheduler 源码分析

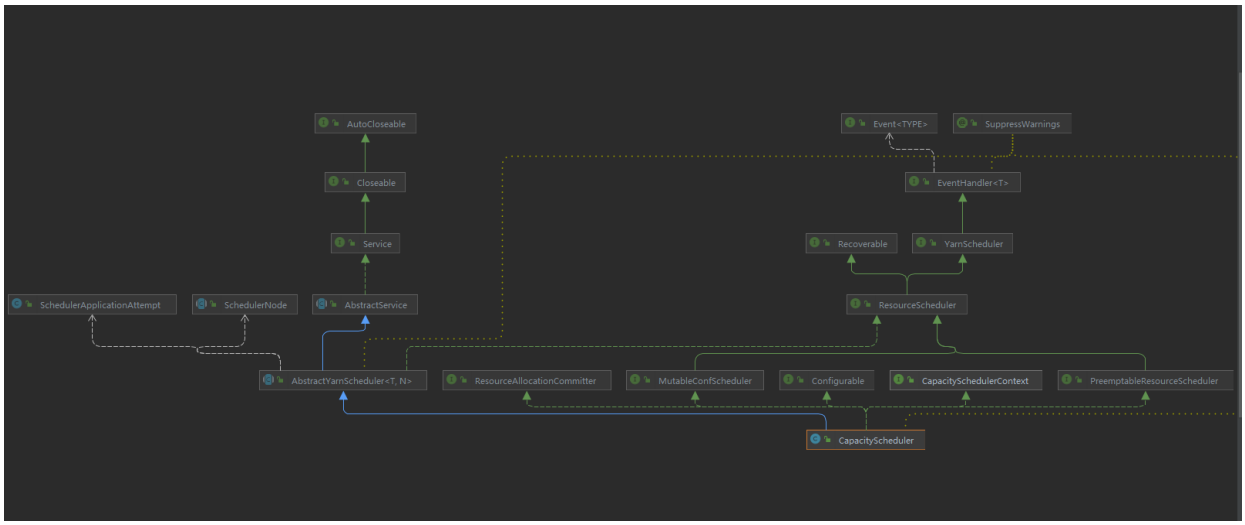
```
//在 ResourceManager 服务启动过程中有一个子服务 RMActiveServices 组合服务，该组合服务在调用其 serviceInit() 时反射创建资源调度器服务(默认 CapacityScheduler 配置 key = yarn.resourcemanager.scheduler.class)

@Override
protected void serviceInit(Configuration configuration) throws Exception {
         ......

         // Initialize the scheduler
        // 反射创建资源调度器(默认 CapacityScheduler 对象)
        scheduler = createScheduler();
        scheduler.setRMContext(rmContext);
        // 添加 CapacityScheduler 服务到 ResourceManager 组合服务
        addIfService(scheduler);
        rmContext.setScheduler(scheduler);

            ......
}
```

## 4.1 容器资源调度器 CapacityScheduler 代码结构解析




可以看到
CapacityScheduler 是一个 Service
    1 构造方法
    2 serviceInit()
    3 serviceStart()
CapacityScheduler 是一个 EventHandler
    4 handle()
CapacityScheduler 是一个 ResourceScheduler
    5 allocate()

## 4.2 CapacityScheduler 的构造方法源码分析

```
@LimitedPrivate("yarn")
@Evolving
@SuppressWarnings("unchecked")
public class CapacityScheduler extends
        AbstractYarnScheduler<FiCaSchedulerApp, FiCaSchedulerNode> implements
        PreemptableResourceScheduler, CapacitySchedulerContext, Configurable,
        ResourceAllocationCommitter, MutableConfScheduler {
   public CapacityScheduler() {
        super(CapacityScheduler.class.getName());
    }
   @Override
    public void setConf(Configuration conf) {
        // 反射创建后调用
        yarnConf = conf;
    }

}
```

## 4.3 CapacityScheduler 服务的 serviceInit()

```
@Override
    public void serviceInit(Configuration conf) throws Exception {
        Configuration configuration = new Configuration(conf);
        // 调用父类
        super.serviceInit(conf);
        // 初始化容器调度器
        initScheduler(configuration);
        // Initialize SchedulingMonitorManager
        // 是否开启资源抢占模式 默认不开启
        schedulingMonitorManager.initialize(rmContext, conf);
    }
```

### 4.3.1 初始化容器调度器(调用其 initScheduler())

```
@VisibleForTesting
    void initScheduler(Configuration configuration) throws
            IOException {
        try {
            writeLock.lock();

            // 默认 file
            String confProviderStr = configuration.get(
                    YarnConfiguration.SCHEDULER_CONFIGURATION_STORE_CLASS,
                    YarnConfiguration.DEFAULT_CONFIGURATION_STORE);

            switch (confProviderStr) {
                case YarnConfiguration.FILE_CONFIGURATION_STORE:
                    // 创建 FileBasedCSConfigurationProvider
                    this.csConfProvider =
                            new FileBasedCSConfigurationProvider(rmContext);
                    break;
                case YarnConfiguration.MEMORY_CONFIGURATION_STORE:
                case YarnConfiguration.LEVELDB_CONFIGURATION_STORE:
                case YarnConfiguration.ZK_CONFIGURATION_STORE:
                    this.csConfProvider = new
MutableCSConfigurationProvider(rmContext);
                    break;
                default:
                    throw new IOException("Invalid configuration store class: " +
                            confProviderStr);
            }
            // 初始化 FileBasedCSConfigurationProvider (啥也不干)
            this.csConfProvider.init(configuration);
            // 加载 capacity-scheduler.xml 配置文件
            this.conf = this.csConfProvider.loadConfiguration(configuration);

            // 校验配置
            validateConf(this.conf);
```

```
            // 解析最小资源分配 默认 <memory:512, vCores:1>
            this.minimumAllocation = super.getMinimumAllocation();

            initMaximumResourceCapability(
                    // 解析最大资源分配 默认 <memory:4096, vCores:4>
                    super.getMaximumAllocation()
            );
            // 创建资源计算器 DefaultResourceCalculator
            this.calculator = this.conf.getResourceCalculator();
            if (this.calculator instanceof DefaultResourceCalculator
                    && ResourceUtils.getNumberOfKnownResourceTypes() > 2) {
                throw new YarnRuntimeException("RM uses DefaultResourceCalculator
which"
                        + " used only memory as resource-type but invalid resource-
types"
                        + " specified " + ResourceUtils.getResourceTypes() + ". Use"
                        + " DominantResourceCalculator instead to make effective use
of"
                        + " these resource-types");
            }
            this.usePortForNodeName = this.conf.getUsePortForNodeName();
            // 创建缓存 Application
            this.applications = new ConcurrentHashMap<>();
            // 节点标签器
            this.labelManager = rmContext.getNodeLabelManager();
            this.appPriorityACLManager = new AppPriorityACLsManager(conf);

            // 创建队列管理器 CapacitySchedulerQueueManager
            this.queueManager = new CapacitySchedulerQueueManager(yarnConf,
                    this.labelManager, this.appPriorityACLManager);
            this.queueManager.setCapacitySchedulerContext(this);

            // 创建并初始化 ActivitiesManager
            this.activitiesManager = new ActivitiesManager(rmContext);
            activitiesManager.init(conf);

            // 初始化队列管理器
            initializeQueues(this.conf);
            this.isLazyPreemptionEnabled = conf.getLazyPreemptionEnabled();

            scheduleAsynchronously = this.conf.getScheduleAynschronously();
            asyncScheduleInterval = this.conf.getLong(ASYNC_SCHEDULER_INTERVAL,
                    DEFAULT_ASYNC_SCHEDULER_INTERVAL);

            this.assignMultipleEnabled = this.conf.getAssignMultipleEnabled();
            this.maxAssignPerHeartbeat = this.conf.getMaxAssignPerHeartbeat();

            // number of threads for async scheduling
            int maxAsyncSchedulingThreads = this.conf.getInt(

 CapacitySchedulerConfiguration.SCHEDULE_ASYNCHRONOUSLY_MAXIMUM_THREAD,
                    1);
```

```
            maxAsyncSchedulingThreads = Math.max(maxAsyncSchedulingThreads, 1);

            if (scheduleAsynchronously) {
                asyncSchedulerThreads = new ArrayList<>();
                for (int i = 0; i < maxAsyncSchedulingThreads; i++) {
                    asyncSchedulerThreads.add(new AsyncScheduleThread(this));
                }
                resourceCommitterService = new ResourceCommitterService(this);
                asyncMaxPendingBacklogs = this.conf.getInt(
                        CapacitySchedulerConfiguration.
                                SCHEDULE_ASYNCHRONOUSLY_MAXIMUM_PENDING_BACKLOGS,
                        CapacitySchedulerConfiguration.

 DEFAULT_SCHEDULE_ASYNCHRONOUSLY_MAXIMUM_PENDING_BACKLOGS);
            }

            // Setup how many containers we can allocate for each round
            offswitchPerHeartbeatLimit = this.conf.getOffSwitchPerHeartbeatLimit();

            LOG.info("Initialized CapacityScheduler with " + "calculator="
                    + getResourceCalculator().getClass() + ", " +
"minimumAllocation=<"
                    + getMinimumResourceCapability() + ">, " + "maximumAllocation=<"
                    + getMaximumResourceCapability() + ">, " +
"asynchronousScheduling="
                    + scheduleAsynchronously + ", " + "asyncScheduleInterval="
                    + asyncScheduleInterval + "ms");
        } finally {
            writeLock.unlock();
        }
    }
```

### 4.3.2 资源抢占模式初始化 (调用SchedulingMonitorManager.initialize())

```
public synchronized void initialize(RMContext rmContext,
      Configuration configuration) throws YarnException {
    this.rmContext = rmContext;
    stopAndRemoveAll();

    // 更新抢占模式调度监控 (本质上 YARN 的抢占模式就是每隔一定时间检查资源不足情况下哪些队列需要
发生资源回收
    // 前提是该队列资源被其他队列占用啦)
    updateSchedulingMonitors(configuration, false);
  }
```

```
private void updateSchedulingMonitors(Configuration conf,
                                      boolean startImmediately) throws
YarnException {
        // 默认不开启资源抢占功能 可以配置 key =
yarn.resourcemanager.scheduler.monitor.enable value = true
        boolean monitorsEnabled = conf.getBoolean(
```

```
                YarnConfiguration.RM_SCHEDULER_ENABLE_MONITORS,
                YarnConfiguration.DEFAULT_RM_SCHEDULER_ENABLE_MONITORS);

        if (!monitorsEnabled) {
            if (!runningSchedulingMonitors.isEmpty()) {
                // If monitors disabled while we have some running monitors, we
should
                // stop them.
                LOG.info("Scheduling Monitor disabled, stopping all services");
                stopAndRemoveAll();
            }

            return;
        }

        // When monitor is enabled, loading policies
        // 当前开启资源抢占功能是 加载抢占策略 默认
        //
org.apache.hadoop.yarn.server.resourcemanager.monitor.capacity.ProportionalCapacityP
reemptionPolicy
        String[] configuredPolicies = conf.getStrings(
                YarnConfiguration.RM_SCHEDULER_MONITOR_POLICIES);
        if (configuredPolicies == null || configuredPolicies.length == 0) {
            return;
        }

        Set<String> configurePoliciesSet = new HashSet<>
(Arrays.asList(configuredPolicies));

        // Add new monitor when needed
        for (String s : configurePoliciesSet) {
            if (!runningSchedulingMonitors.containsKey(s)) {
                Class<?> policyClass;
                try {
                    policyClass = Class.forName(s);
                } catch (ClassNotFoundException e) {
                    String message = "Failed to find class of specified policy=" +
s;
                    LOG.warn(message);
                    throw new YarnException(message);
                }

                if (SchedulingEditPolicy.class.isAssignableFrom(policyClass)) {
                    // 反射创建 ProportionalCapacityPreemptionPolicy
                    SchedulingEditPolicy policyInstance =
                            (SchedulingEditPolicy)
ReflectionUtils.newInstance(policyClass,
                                    null);
                    // 创建抢占资源监控 SchedulingMonitor 服务
                    SchedulingMonitor mon = new SchedulingMonitor(rmContext,
                            policyInstance);
                    // 调用 SchedulingMonitor.serviceInit()
```

```
                    mon.init(conf);
                    // 默认 false
                    if (startImmediately) {
                        // 等到调用 CapacityScheduler.serviceStart() 调用
SchedulingMonitor.serviceStart()
                        mon.start();
                    }
                    runningSchedulingMonitors.put(s, mon);
                } else {
                    String message =
                            "Specified policy=" + s + " is not a
SchedulingEditPolicy class.";
                    LOG.warn(message);
                    throw new YarnException(message);
                }
            }
        }

        // Stop monitor when needed.
        Set<String> disabledPolicies = Sets.difference(
                runningSchedulingMonitors.keySet(), configurePoliciesSet);
        for (String disabledPolicy : disabledPolicies) {
            LOG.info("SchedulingEditPolicy=" + disabledPolicy
                    + " removed, stopping it now ...");
            silentlyStopSchedulingMonitor(disabledPolicy);
            runningSchedulingMonitors.remove(disabledPolicy);
        }
    }
```

#### 4.3.2.1 创建抢占资源监控 SchedulingMonitor 服务 (调用其 serviceInit())

```
public class SchedulingMonitor extends AbstractService {
     public SchedulingMonitor(RMContext rmContext,
                              SchedulingEditPolicy scheduleEditPolicy) {
        super("SchedulingMonitor (" + scheduleEditPolicy.getPolicyName() + ")");
        // ProportionalCapacityPreemptionPolicy
        this.scheduleEditPolicy = scheduleEditPolicy;
        this.rmContext = rmContext;
    }
}
```

```
    public void serviceInit(Configuration conf) throws Exception {
        LOG.info("Initializing SchedulingMonitor=" + getName());
        // 调用 ProportionalCapacityPreemptionPolicy.init()
        scheduleEditPolicy.init(conf, rmContext, rmContext.getScheduler());
        // 默认 3s
        this.monitorInterval = scheduleEditPolicy.getMonitoringInterval();
        super.serviceInit(conf);
    }
```

## 4.4 CapacityScheduler 服务的 serviceStart()

```
    @Override
    public void serviceStart() throws Exception {
        // 启动定时调度线程
        startSchedulerThreads();
        // 启动抢占资源监控
        super.serviceStart();
    }
```

```
private void startSchedulerThreads() {
        try {
            writeLock.lock();
            // 启动
            activitiesManager.start();
            if (scheduleAsynchronously) {
                Preconditions.checkNotNull(asyncSchedulerThreads,
                        "asyncSchedulerThreads is null");
                for (Thread t : asyncSchedulerThreads) {
                    t.start();
                }

                // 启动资源提交线程 调用 ResourceCommitterService.run()
                resourceCommitterService.start();
            }
        } finally {
            writeLock.unlock();
        }
    }
```

```
    @Override
    protected void serviceStart() throws Exception {
        if (updateThread != null) {
            updateThread.start();
        }
        // 启动抢占资源服务 (底层启动线程每隔 3s 检查哪些队列资源出现抢占)
        schedulingMonitorManager.startAll();
        createReleaseCache();
        super.serviceStart();
    }
```

```
    public synchronized void startAll() {
        for (SchedulingMonitor schedulingMonitor : runningSchedulingMonitors
                .values()) {
            // 调用 SchedulingMonitor.serviceStart()
            schedulingMonitor.start();
        }
    }
```

### 4.4.1 启动资源抢占服务 调用 SchedulingMonitor.serviceStart()

```
@Override
    public void serviceStart() throws Exception {
        LOG.info("Starting SchedulingMonitor=" + getName());
        assert !stopped : "starting when already stopped";
        ses = Executors.newSingleThreadScheduledExecutor(new ThreadFactory() {
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r);
                t.setName(getName());
                return t;
            }
        });
        // 往下追
        schedulePreemptionChecker();
        super.serviceStart();
    }
```

```
    private void schedulePreemptionChecker() {
        // 每隔 3s 执行 PolicyInvoker.run()
        handler = ses.scheduleAtFixedRate(new PolicyInvoker(),
                0, monitorInterval, TimeUnit.MILLISECONDS);
    }
```

```
private class PolicyInvoker implements Runnable {
        @Override
        public void run() {
            try {
                if (monitorInterval != scheduleEditPolicy.getMonitoringInterval()) {
                    handler.cancel(true);
                    monitorInterval = scheduleEditPolicy.getMonitoringInterval();
                    schedulePreemptionChecker();
                } else {
                    // 触发检查资源抢占检查
                    invokePolicy();
                }
            } catch (Throwable t) {
                // The preemption monitor does not alter structures nor do
structures
                // persist across invocations. Therefore, log, skip, and retry.
                LOG.error("Exception raised while executing preemption"
                        + " checker, skip this run..., exception=", t);
            }
        }
    }
```

```
@VisibleForTesting
public void invokePolicy() {
    // 调用 ProportionalCapacityPreemptionPolicy.editSchedule()
    scheduleEditPolicy.editSchedule();
}
```

```
/**
 * This class implement a {@link SchedulingEditPolicy} that is designed to be
 * paired with the {@code CapacityScheduler}. At every invocation of {@code
 * editSchedule()} it computes the ideal amount of resources assigned to each
 * queue (for each queue in the hierarchy), and determines whether preemption
 * is needed. Overcapacity is distributed among queues in a weighted fair manner,
 * where the weight is the amount of guaranteed capacity for the queue.
 * Based on this ideal assignment it determines whether preemption is required
 * and select a set of containers from each application that would be killed if
 * the corresponding amount of resources is not freed up by the application.
 *
 * If not in {@code observeOnly} mode, it triggers preemption requests via a
 * {@link ContainerPreemptEvent} that the {@code ResourceManager} will ensure
 * to deliver to the application (or to execute).
 *
 * If the deficit of resources is persistent over a long enough period of time
 * this policy will trigger forced termination of containers (again by generating
 * {@link ContainerPreemptEvent}).
 */
public class ProportionalCapacityPreemptionPolicy
        implements SchedulingEditPolicy, CapacitySchedulerPreemptionContext {
 @Override
    public synchronized void editSchedule() {
        updateConfigIfNeeded();

        long startTs = clock.getTime();

        CSQueue root = scheduler.getRootQueue();
        Resource clusterResources = Resources.clone(scheduler.getClusterResource());
        // 找到哪些候选容器资源抢占之后强制杀掉
        containerBasedPreemptOrKill(root, clusterResources);

        if (LOG.isDebugEnabled()) {
            LOG.debug("Total time used=" + (clock.getTime() - startTs) + " ms.");
        }
    }
}
```

```
/**
     * This method selects and tracks containers to be preemptionCandidates. If a
container
     * is in the target list for more than maxWaitTime it is killed.
     *
     * @param root             the root of the CapacityScheduler queue hierarchy
```

```
     * @param clusterResources the total amount of resources in the cluster
     */
    private void containerBasedPreemptOrKill(CSQueue root,
                                             Resource clusterResources) {
        // Sync killable containers from scheduler when lazy preemption enabled
        if (lazyPreempionEnabled) {
            syncKillableContainersFromScheduler();
        }

        // All partitions to look at
        Set<String> partitions = new HashSet<>();
        partitions.addAll(scheduler.getRMContext()
                .getNodeLabelManager().getClusterNodeLabelNames());
        partitions.add(RMNodeLabelsManager.NO_LABEL);
        this.allPartitions = ImmutableSet.copyOf(partitions);

        // extract a summary of the queues from scheduler
        synchronized (scheduler) {
            queueToPartitions.clear();

            for (String partitionToLookAt : allPartitions) {
                cloneQueues(root, Resources
                                .clone(nlm.getResourceByLabel(partitionToLookAt,
clusterResources)),
                        partitionToLookAt);
            }

            // Update effective priority of queues
        }

        this.leafQueueNames = ImmutableSet.copyOf(getLeafQueueNames(
                getQueueByPartition(CapacitySchedulerConfiguration.ROOT,
                        RMNodeLabelsManager.NO_LABEL)));

        // compute total preemption allowed
        Resource totalPreemptionAllowed = Resources.multiply(clusterResources,
                percentageClusterPreemptionAllowed);

        //clear under served queues for every run
        partitionToUnderServedQueues.clear();

        // based on ideal allocation select containers to be preemptionCandidates
from each
        // queue and each application
        Map<ApplicationAttemptId, Set<RMContainer>> toPreempt =
                new HashMap<>();
        Map<PreemptionCandidatesSelector, Map<ApplicationAttemptId,
                Set<RMContainer>>> toPreemptPerSelector = new HashMap<>();
        ;
        for (PreemptionCandidatesSelector selector :
                candidatesSelectionPolicies) {
            long startTime = 0;
```

```
            if (LOG.isDebugEnabled()) {
                LOG.debug(MessageFormat
                        .format("Trying to use {0} to select preemption candidates",
                                selector.getClass().getName()));
                startTime = clock.getTime();
            }
            // 候选杀掉容器资源 (这些资源出现抢占情况)
            Map<ApplicationAttemptId, Set<RMContainer>> curCandidates =
                    selector.selectCandidates(toPreempt, clusterResources,
                            totalPreemptionAllowed);
            toPreemptPerSelector.putIfAbsent(selector, curCandidates);

            if (LOG.isDebugEnabled()) {
                LOG.debug(MessageFormat
                        .format("{0} uses {1} millisecond to run",
                                selector.getClass().getName(), clock.getTime() -
startTime));
                int totalSelected = 0;
                int curSelected = 0;
                for (Set<RMContainer> set : toPreempt.values()) {
                    totalSelected += set.size();
                }
                for (Set<RMContainer> set : curCandidates.values()) {
                    curSelected += set.size();
                }
                LOG.debug(MessageFormat
                        .format("So far, total {0} containers selected to be
preempted, {1}"
                                        + " containers selected this round\n",
                                totalSelected, curSelected));
            }
        }

        if (LOG.isDebugEnabled()) {
            logToCSV(new ArrayList<>(leafQueueNames));
        }

        // if we are in observeOnly mode return before any action is taken
        if (observeOnly) {
            return;
        }

        // TODO: need consider revert killable containers when no more demandings.
        // Since we could have several selectors to make decisions concurrently.
        // So computed ideal-allocation varies between different selectors.
        //
        // We may need to "score" killable containers and revert the most preferred
        // containers. The bottom line is, we shouldn't preempt a queue which is
already
        // below its guaranteed resource.

        long currentTime = clock.getTime();
```

```
        pcsMap = toPreemptPerSelector;

        // preempt (or kill) the selected containers
        // 杀掉抢占资源的容器
        preemptOrkillSelectedContainerAfterWait(toPreemptPerSelector, currentTime);

        // cleanup staled preemption candidates
        cleanupStaledPreemptionCandidates(currentTime);
    }
```

# 五 YARN 资源的管理和申请源码分析

## 5.1 NM 上线注册上报资源和上报容器资源使用情况 (针对已完成的容器释放资源而言 这两种情况都是基于向 RM 发送心跳 NM 上线除外)

```
// NodeManager 在启动过程中调用其 serviceStart()，调用其子服务
NodeStatusUpdaterImpl.serviceStart() 完成上线注册以及心跳
@Override
    protected void serviceStart() throws Exception {

        // NodeManager is the last service to start, so NodeId is available.
        // 获取 NodeManager 节点 ID
        // 上一个服务 ContainerManagerImpl 组合服务 启动了一个 RPC Server
        // 并绑定通讯协议接口 ContainerManagementProtocol 但是这个 RPC Server
        // 的端口是临时随机的 并且将该 RPC的 IP:随机端口封装成 NodeId 设置到 NodeManager 的
        // 上下文对象 NMContext
        this.nodeId = this.context.getNodeId();
        // Node ID assigned is : hadoop103:40144
        LOG.info("Node ID assigned is : " + this.nodeId);
        // 默认 NodeManager 启动 WebServer 绑定地址 0.0.0.0:8042
        this.httpPort = this.context.getHttpPort();
        this.nodeManagerVersionId = YarnVersionInfo.getVersion();
        try {
            // Registration has to be in start so that ContainerManager can get the
            // perNM tokens needed to authenticate ContainerTokens.
            // 获取 ResourceManager RPC 客户端
            // (ResourceManager 的服务端为 ResourceTrackerService 绑定端口为 8031 )
            this.resourceTracker = getRMClient();

            // 发送 RPC 请求向 ResourceManager 的 ResourceTrackerService 服务注册
NodeManager
            registerWithRM();

            super.serviceStart();

            // 发送 RPC 请求向 ResourceManager 发送心跳
            startStatusUpdater();
        } catch (Exception e) {
            String errorMessage = "Unexpected error starting NodeStatusUpdater";
```

```
            LOG.error(errorMessage, e);
            throw new YarnRuntimeException(e);
        }
    }
```

### 5.1.1 NM 注册上报配置资源

```
 @VisibleForTesting
    protected void registerWithRM()
            throws YarnException, IOException {
        RegisterNodeManagerResponse regNMResponse;
        // 获取 NodeManager 的标签信息
        Set<NodeLabel> nodeLabels =
nodeLabelsHandler.getNodeLabelsForRegistration();

        // Synchronize NM-RM registration with
        // ContainerManagerImpl#increaseContainersResource and
        // ContainerManagerImpl#startContainers to avoid race condition
        // during RM recovery
        synchronized (this.context) {
            // 获取 NodeManager 的所有容器状态 但是 NodeManager 开始启动并什么容器
            // 故一般情况下返回空
            List<NMContainerStatus> containerReports = getNMContainerStatuses();

            // 封装 NodeManager 注册请求
            RegisterNodeManagerRequest request =
                    RegisterNodeManagerRequest.newInstance(
                            nodeId, // NodeManager 节点 ID
                            httpPort, // NodeManager HTTP 服务端口
                            totalResource, // NodeManager 配置资源信息 (内存 CPU)
                            nodeManagerVersionId, // NodeManager 版本 ID
                            containerReports, // 空
                            getRunningApplications(), // 空
                            nodeLabels, // NodeManager 的标签信息
                            physicalResource // NodeManager 的真实物理内存
                    );

            // Registering with RM using containers :[]
            LOG.info("Registering with RM using containers :" + containerReports);

            if (logAggregationEnabled) {
                // pull log aggregation status for application running in this NM
                List<LogAggregationReport> logAggregationReports =
                        context.getNMLogAggregationStatusTracker()
                                .pullCachedLogAggregationReports();
                if (LOG.isDebugEnabled()) {
                    LOG.debug("The cache log aggregation status size:"
                            + logAggregationReports.size());
                }
                if (logAggregationReports != null
                        && !logAggregationReports.isEmpty()) {
```

```
                    request.setLogAggregationReportsForApps(logAggregationReports);
                }
            }

            // 发送 RPC 请求注册 NM 并返回响应数据
            regNMResponse =
                    resourceTracker.registerNodeManager(request);
            // Make sure rmIdentifier is set before we release the lock
            this.rmIdentifier = regNMResponse.getRMIdentifier();
        }

        // if the Resource Manager instructs NM to shutdown.
        if (NodeAction.SHUTDOWN.equals(regNMResponse.getNodeAction())) {
            String message =
                    "Message from ResourceManager: "
                            + regNMResponse.getDiagnosticsMessage();
            throw new YarnRuntimeException(
                    "Received SHUTDOWN signal from Resourcemanager, Registration of
NodeManager failed, "
                            + message);
        }

        // if ResourceManager version is too old then shutdown
        // 默认 minimumResourceManagerVersion = NONE
        if (!minimumResourceManagerVersion.equals("NONE")) {
            if (minimumResourceManagerVersion.equals("EqualToNM")) {
                minimumResourceManagerVersion = nodeManagerVersionId;
            }
            String rmVersion = regNMResponse.getRMVersion();
            if (rmVersion == null) {
                String message = "The Resource Manager's did not return a version. "
                        + "Valid version cannot be checked.";
                throw new YarnRuntimeException("Shutting down the Node Manager. "
                        + message);
            }
            if (VersionUtil.compareVersions(rmVersion,
minimumResourceManagerVersion) < 0) {
                String message = "The Resource Manager's version ("
                        + rmVersion + ") is less than the minimum "
                        + "allowed version " + minimumResourceManagerVersion;
                throw new YarnRuntimeException("Shutting down the Node Manager on RM
"
                        + "version error, " + message);
            }
        }

        // 标识 NM 已经向 RM 注册
        this.registeredWithRM = true;

        // 根据注册返回信息 Token Master & Token Secret 添加到 NM 的上下文对象
        MasterKey masterKey = regNMResponse.getContainerTokenMasterKey();
        // do this now so that its set before we start heartbeating to RM
```

```
        // It is expected that status updater is started by this point and
        // RM gives the shared secret in registration during
        // StatusUpdater#start().
        if (masterKey != null) {
            this.context.getContainerTokenSecretManager().setMasterKey(masterKey);
        }
        masterKey = regNMResponse.getNMTokenMasterKey();
        if (masterKey != null) {
            this.context.getNMTokenSecretManager().setMasterKey(masterKey);
        }

        StringBuilder successfullRegistrationMsg = new StringBuilder();
        successfullRegistrationMsg.append("Registered with ResourceManager as ")
                .append(this.nodeId);

        Resource newResource = regNMResponse.getResource();
        if (newResource != null) {
            updateNMResource(newResource);
            successfullRegistrationMsg.append(" with updated total resource of ")
                    .append(this.totalResource);
        } else {
            successfullRegistrationMsg.append(" with total resource of ")
                    .append(this.totalResource);
        }
        successfullRegistrationMsg.append(nodeLabelsHandler
                .verifyRMRegistrationResponseForNodeLabels(regNMResponse));

        // Registered with ResourceManager as
        // hadoop103:40144 with
        // total resource of <memory:4096, vCores:8>
        LOG.info(successfullRegistrationMsg.toString());
    }
```

#### 5.1.1.1 调用 RM 的 ResourceTrackerService.registerNodeManager() 注册 NM

```
@SuppressWarnings("unchecked")
    @Override
    public RegisterNodeManagerResponse registerNodeManager(
            RegisterNodeManagerRequest request) throws YarnException,
            IOException {
        // 获取 NM 节点信息 (IP:端口)
        NodeId nodeId = request.getNodeId();
        String host = nodeId.getHost();
        int cmPort = nodeId.getPort();
        // 获取 NM HTTP 服务端口
        int httpPort = request.getHttpPort();
        // 获取 NM 配置资源信息 (内存 CPU)
        Resource capability = request.getResource();
        // 获取 NM 版本信息
        String nodeManagerVersion = request.getNMVersion();
        // 获取 NM 真实物理资源 (内存 CPU)
        Resource physicalResource = request.getPhysicalResource();
```

```
        // 构建注册 NM 响应对象
        RegisterNodeManagerResponse response = recordFactory
                .newRecordInstance(RegisterNodeManagerResponse.class);

        // minimumNodeManagerVersion 默认 NONE
        if (!minimumNodeManagerVersion.equals("NONE")) {
            if (minimumNodeManagerVersion.equals("EqualToRM")) {
                minimumNodeManagerVersion = YarnVersionInfo.getVersion();
            }

            if ((nodeManagerVersion == null) ||
                    (VersionUtil.compareVersions(nodeManagerVersion,
minimumNodeManagerVersion)) < 0) {
                String message =
                        "Disallowed NodeManager Version " + nodeManagerVersion
                                + ", is less than the minimum version "
                                + minimumNodeManagerVersion + " sending SHUTDOWN
signal to "
                                + "NodeManager.";
                LOG.info(message);
                response.setDiagnosticsMessage(message);
                response.setNodeAction(NodeAction.SHUTDOWN);
                return response;
            }
        }

        // Check if this node is a 'valid' node
        // 检查注册 NM 是否合法 (白名单与黑名单)
        if (!this.nodesListManager.isValidNode(host) &&
                !isNodeInDecommissioning(nodeId)) {
            String message =
                    "Disallowed NodeManager from  " + host
                            + ", Sending SHUTDOWN signal to the NodeManager.";
            LOG.info(message);
            response.setDiagnosticsMessage(message);
            response.setNodeAction(NodeAction.SHUTDOWN);
            return response;
        }

        // check if node's capacity is load from dynamic-resources.xml
        String nid = nodeId.toString();

        // 默认返回 null
        Resource dynamicLoadCapability = loadNodeResourceFromDRConfiguration(nid);
        if (dynamicLoadCapability != null) {
            if (LOG.isDebugEnabled()) {
                LOG.debug("Resource for node: " + nid + " is adjusted from: " +
                        capability + " to: " + dynamicLoadCapability +
                        " due to settings in dynamic-resources.xml.");
            }
            capability = dynamicLoadCapability;
```

```
            // sync back with new resource.
            response.setResource(capability);
        }

        // Check if this node has minimum allocations
        // 判断 NM 资源是否满足一个容器最小配置资源 (也即注册的 NM 是否满足启动一个容器的最低资源)
        if (capability.getMemorySize() < minAllocMb
                || capability.getVirtualCores() < minAllocVcores) {
            String message = "NodeManager from  " + host
                    + " doesn't satisfy minimum allocations, Sending SHUTDOWN"
                    + " signal to the NodeManager. Node capabilities are " +
capability
                    + "; minimums are " + minAllocMb + "mb and " + minAllocVcores
                    + " vcores";
            LOG.info(message);
            response.setDiagnosticsMessage(message);
            response.setNodeAction(NodeAction.SHUTDOWN);
            return response;
        }

        // 返回容器 Token Secret
        response.setContainerTokenMasterKey(containerTokenSecretManager
                .getCurrentKey());
        // 返回 NM Token Master
        response.setNMTokenMasterKey(nmTokenSecretManager
                .getCurrentKey());

        // 封装 RM 的信息
        RMNode rmNode = new RMNodeImpl(nodeId, rmContext, host, cmPort, httpPort,
                resolve(host), capability, nodeManagerVersion, physicalResource);

        // 将 NM 的 NodeID 跟 RM 信息绑定
        RMNode oldNode = this.rmContext.getRMNodes().putIfAbsent(nodeId, rmNode);
        // 一般情况下为 true
        if (oldNode == null) {
            // 创建 RMNodeStartedEvent 事件
            RMNodeStartedEvent startEvent = new RMNodeStartedEvent(nodeId,
                    request.getNMContainerStatuses(),
                    request.getRunningApplications());
            if (request.getLogAggregationReportsForApps() != null
                    && !request.getLogAggregationReportsForApps().isEmpty()) {
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Found the number of previous cached log aggregation "
                            + "status from nodemanager:" + nodeId + " is :"
                            + request.getLogAggregationReportsForApps().size());
                }
                startEvent.setLogAggregationReportsForApps(request
                        .getLogAggregationReportsForApps());
            }

            // 调用 NodeEventDispatcher.handle() (最终调用 RMNodeImpl.handle())
```

```
            this.rmContext.getDispatcher().getEventHandler().handle(
                    startEvent);
        } else {
            LOG.info("Reconnect from the node at: " + host);
            this.nmLivelinessMonitor.unregister(nodeId);

            if (CollectionUtils.isEmpty(request.getRunningApplications())
                    && rmNode.getState() != NodeState.DECOMMISSIONING
                    && rmNode.getHttpPort() != oldNode.getHttpPort()) {
                // Reconnected node differs, so replace old node and start new node
                switch (rmNode.getState()) {
                    case RUNNING:
                        ClusterMetrics.getMetrics().decrNumActiveNodes();
                        break;
                    case UNHEALTHY:
                        ClusterMetrics.getMetrics().decrNumUnhealthyNMs();
                        break;
                    default:
                        LOG.debug("Unexpected Rmnode state");
                }
                this.rmContext.getDispatcher().getEventHandler()
                        .handle(new NodeRemovedSchedulerEvent(rmNode));

                this.rmContext.getRMNodes().put(nodeId, rmNode);
                this.rmContext.getDispatcher().getEventHandler()
                        .handle(new RMNodeStartedEvent(nodeId, null, null));
            } else {
                // Reset heartbeat ID since node just restarted.
                oldNode.resetLastNodeHeartBeatResponse();

                this.rmContext.getDispatcher().getEventHandler()
                        .handle(new RMNodeReconnectEvent(nodeId, rmNode,
                                request.getRunningApplications(),
                                request.getNMContainerStatuses()));
            }
        }
        // On every node manager register we will be clearing NMToken keys if
        // present for any running application.
        this.nmTokenSecretManager.removeNodeKey(nodeId);
        // NM 加入 RM 监控
        this.nmLivelinessMonitor.register(nodeId);

        // Handle received container status, this should be processed after new
        // RMNode inserted
        if (!rmContext.isWorkPreservingRecoveryEnabled()) {
            if (!request.getNMContainerStatuses().isEmpty()) {
                LOG.info("received container statuses on node manager register :"
                        + request.getNMContainerStatuses());
                for (NMContainerStatus status : request.getNMContainerStatuses()) {
                    handleNMContainerStatus(status, nodeId);
                }
            }
```

```
        }

        // Update node's labels to RM's NodeLabelManager.
        // NM 标签相关处理
        Set<String> nodeLabels = NodeLabelsUtils.convertToStringSet(
                request.getNodeLabels());
        if (isDistributedNodeLabelsConf && nodeLabels != null) {
            try {
                updateNodeLabelsFromNMReport(nodeLabels, nodeId);
                response.setAreNodeLabelsAcceptedByRM(true);
            } catch (IOException ex) {
                // Ensure the exception is captured in the response
                response.setDiagnosticsMessage(ex.getMessage());
                response.setAreNodeLabelsAcceptedByRM(false);
            }
        } else if (isDelegatedCentralizedNodeLabelsConf) {

 this.rmContext.getRMDelegatedNodeLabelsUpdater().updateNodeLabels(nodeId);
        }

        StringBuilder message = new StringBuilder();
        message.append("NodeManager from node ").append(host).append("(cmPort: ")
                .append(cmPort).append(" httpPort: ");
        message.append(httpPort).append(") ")
                .append("registered with capability: ").append(capability);
        message.append(", assigned nodeId ").append(nodeId);
        if (response.getAreNodeLabelsAcceptedByRM()) {
            message.append(", node labels { ").append(
                    StringUtils.join(",", nodeLabels) + " } ");
        }

        // 构建返回 NM 注册信息
        // NodeManager from node
        // hadoop104(cmPort: 39330 httpPort: 8042)
        // registered with capability: <memory:4096, vCores:8>,
        // assigned nodeId hadoop104:39330
        LOG.info(message.toString());
        response.setNodeAction(NodeAction.NORMAL);
        response.setRMIdentifier(ResourceManager.getClusterTimeStamp());
        response.setRMVersion(YarnVersionInfo.getVersion());
        return response;
    }
```

#### 5.1.1.2 最终调用 CapacityScheduler.handle(NODE_ADDED)

```
@Override
    public void handle(SchedulerEvent event) {
        switch (event.getType()) {
            case NODE_ADDED: {
                // NM 节点添加事件
                NodeAddedSchedulerEvent nodeAddedEvent = (NodeAddedSchedulerEvent)
event;
```

```
                // 添加 NM 节点
                addNode(nodeAddedEvent.getAddedRMNode());
                recoverContainersOnNode(nodeAddedEvent.getContainerReports(),
                        nodeAddedEvent.getAddedRMNode());
            }
            break;
            case NODE_REMOVED: {
                NodeRemovedSchedulerEvent nodeRemovedEvent =
(NodeRemovedSchedulerEvent) event;
                removeNode(nodeRemovedEvent.getRemovedRMNode());
            }
            break;
            case NODE_RESOURCE_UPDATE: {
                NodeResourceUpdateSchedulerEvent nodeResourceUpdatedEvent =
                        (NodeResourceUpdateSchedulerEvent) event;
                updateNodeAndQueueResource(nodeResourceUpdatedEvent.getRMNode(),
                        nodeResourceUpdatedEvent.getResourceOption());
            }
            break;
            case NODE_LABELS_UPDATE: {
                NodeLabelsUpdateSchedulerEvent labelUpdateEvent =
                        (NodeLabelsUpdateSchedulerEvent) event;

                updateNodeLabelsAndQueueResource(labelUpdateEvent);
            }
            break;
            case NODE_UPDATE: {
                // NM 更新 也即 NM 发送心跳
                NodeUpdateSchedulerEvent nodeUpdatedEvent =
(NodeUpdateSchedulerEvent) event;
                nodeUpdate(nodeUpdatedEvent.getRMNode());
            }
            break;
            case APP_ADDED: {
                // event = AppAddedSchedulerEvent()
                // eventType = SchedulerEventType.APP_ADDED
                // 容器调度器添加一个任务
                // appAddedEvent =
                AppAddedSchedulerEvent appAddedEvent = (AppAddedSchedulerEvent)
event;
                String queueName =
resolveReservationQueueName(appAddedEvent.getQueue(),
                        appAddedEvent.getApplicationId(),
appAddedEvent.getReservationID(),
                        appAddedEvent.getIsAppRecovering());
                if (queueName != null) {
                    // 不是任务恢复情况
                    if (!appAddedEvent.getIsAppRecovering()) {
                        // 添加任务应用
                        addApplication(
                                appAddedEvent.getApplicationId(),
                                queueName,
```

```
                                appAddedEvent.getUser(),
                                appAddedEvent.getApplicatonPriority(),
                                appAddedEvent.getPlacementContext());
                    } else {
                        addApplicationOnRecovery(appAddedEvent.getApplicationId(),
queueName,
                                appAddedEvent.getUser(),
appAddedEvent.getApplicatonPriority(),
                                appAddedEvent.getPlacementContext());
                    }
                }
            }
            break;
            case APP_REMOVED: {
                AppRemovedSchedulerEvent appRemovedEvent =
(AppRemovedSchedulerEvent) event;
                doneApplication(appRemovedEvent.getApplicationID(),
                        appRemovedEvent.getFinalState());
            }
            break;
            case APP_ATTEMPT_ADDED: {
                // APP_ATTEMPT_ADDED 事件
                // appAttemptAddedEvent = new
AppAttemptAddedSchedulerEvent(SchedulerEventType.APP_ATTEMPT_ADDED)
                AppAttemptAddedSchedulerEvent appAttemptAddedEvent =
                        (AppAttemptAddedSchedulerEvent) event;
                // 添加 ApplicationAttempt
                addApplicationAttempt(
                        appAttemptAddedEvent.getApplicationAttemptId(),
                        appAttemptAddedEvent.getTransferStateFromPreviousAttempt(),
                        appAttemptAddedEvent.getIsAttemptRecovering());
            }
            break;
            case APP_ATTEMPT_REMOVED: {
                AppAttemptRemovedSchedulerEvent appAttemptRemovedEvent =
                        (AppAttemptRemovedSchedulerEvent) event;

 doneApplicationAttempt(appAttemptRemovedEvent.getApplicationAttemptID(),
                        appAttemptRemovedEvent.getFinalAttemptState(),

 appAttemptRemovedEvent.getKeepContainersAcrossAppAttempts());
            }
            break;
            case CONTAINER_EXPIRED: {
                ContainerExpiredSchedulerEvent containerExpiredEvent =
                        (ContainerExpiredSchedulerEvent) event;
                ContainerId containerId = containerExpiredEvent.getContainerId();
                if (containerExpiredEvent.isIncrease()) {
                    rollbackContainerUpdate(containerId);
                } else {
                    completedContainer(getRMContainer(containerId),
                            SchedulerUtils.createAbnormalContainerStatus(
```

```
                                containerId,
                                SchedulerUtils.EXPIRED_CONTAINER),
                        RMContainerEventType.EXPIRE);
                }
            }
            break;
            case RELEASE_CONTAINER: {
                RMContainer container = ((ReleaseContainerEvent)
event).getContainer();
                completedContainer(container,
                        SchedulerUtils.createAbnormalContainerStatus(
                                container.getContainerId(),
                                SchedulerUtils.RELEASED_CONTAINER),
                        RMContainerEventType.RELEASED);
            }
            break;
            case KILL_RESERVED_CONTAINER: {
                ContainerPreemptEvent killReservedContainerEvent =
                        (ContainerPreemptEvent) event;
                RMContainer container = killReservedContainerEvent.getContainer();
                killReservedContainer(container);
            }
            break;
            case MARK_CONTAINER_FOR_PREEMPTION: {
                ContainerPreemptEvent preemptContainerEvent =
                        (ContainerPreemptEvent) event;
                ApplicationAttemptId aid = preemptContainerEvent.getAppId();
                RMContainer containerToBePreempted =
preemptContainerEvent.getContainer();
                markContainerForPreemption(aid, containerToBePreempted);
            }
            break;
            case MARK_CONTAINER_FOR_KILLABLE: {
                ContainerPreemptEvent containerKillableEvent =
(ContainerPreemptEvent) event;
                RMContainer killableContainer =
containerKillableEvent.getContainer();
                markContainerForKillable(killableContainer);
            }
            break;
            case MARK_CONTAINER_FOR_NONKILLABLE: {
                if (isLazyPreemptionEnabled) {
                    ContainerPreemptEvent cancelKillContainerEvent =
                            (ContainerPreemptEvent) event;

 markContainerForNonKillable(cancelKillContainerEvent.getContainer());
                }
            }
            break;
            case MANAGE_QUEUE: {
                QueueManagementChangeEvent queueManagementChangeEvent =
                        (QueueManagementChangeEvent) event;
```

```
                ParentQueue parentQueue =
queueManagementChangeEvent.getParentQueue();
                try {
                    final List<QueueManagementChange> queueManagementChanges =
                            queueManagementChangeEvent.getQueueManagementChanges();
                    ((ManagedParentQueue) parentQueue)

.validateAndApplyQueueManagementChanges(queueManagementChanges);
                } catch (SchedulerDynamicEditException sde) {
                    LOG.error("Queue Management Change event cannot be applied for "
                            + "parent queue : " + parentQueue.getQueueName(), sde);
                } catch (IOException ioe) {
                    LOG.error("Queue Management Change event cannot be applied for "
                            + "parent queue : " + parentQueue.getQueueName(), ioe);
                }
            }
            break;
            default:
                LOG.error("Invalid eventtype " + event.getType() + ". Ignoring!");
        }
    }
```

### 5.1.2 NM 心跳上报容器资源情况

```
// 心跳线程 调用 StatusUpdaterRunnable.run()
private class StatusUpdaterRunnable implements Runnable {
        @Override
        @SuppressWarnings("unchecked")
        public void run() {
            int lastHeartbeatID = 0;
            while (!isStopped) {
                // Send heartbeat
                try {
                    NodeHeartbeatResponse response = null;
                    // 获取 NM 心跳标签
                    Set<NodeLabel> nodeLabelsForHeartbeat =
                            nodeLabelsHandler.getNodeLabelsForHeartbeat();
                    // 获取 NM 节点状态 (也即获取当前 NM 管理启动容器使用资源情况 可能有些容器
运行完成
                    // 那么需要上报给 RM 这些完成容器的资源可以被使用)
                    NodeStatus nodeStatus = getNodeStatus(lastHeartbeatID);
                    // 构建心跳请求对象
                    NodeHeartbeatRequest request =
                            NodeHeartbeatRequest.newInstance(nodeStatus,
                                    NodeStatusUpdaterImpl.this.context

.getContainerTokenSecretManager().getCurrentKey(),
                                    NodeStatusUpdaterImpl.this.context

.getNMTokenSecretManager().getCurrentKey(),
```

```
                                    nodeLabelsForHeartbeat,
                                    NodeStatusUpdaterImpl.this.context
                                            .getRegisteringCollectors());

                    if (logAggregationEnabled) {
                        // pull log aggregation status for application running in
this NM
                        List<LogAggregationReport> logAggregationReports =
                                getLogAggregationReportsForApps(context
                                        .getLogAggregationStatusForApps());
                        if (logAggregationReports != null
                                && !logAggregationReports.isEmpty()) {

 request.setLogAggregationReportsForApps(logAggregationReports);
                        }
                    }

                    // 发送心跳
                    response = resourceTracker.nodeHeartbeat(request);
                    //get next heartbeat interval from response
                    // 获取下一个心跳间隔
                    nextHeartBeatInterval = response.getNextHeartBeatInterval();
                    updateMasterKeys(response);

                    if (!handleShutdownOrResyncCommand(response)) {
                        nodeLabelsHandler.verifyRMHeartbeatResponseForNodeLabels(
                                response);

                        // Explicitly put this method after checking the resync
                        // response. We
                        // don't want to remove the completed containers before
resync
                        // because these completed containers will be reported back
to RM
                        // when NM re-registers with RM.
                        // Only remove the cleanedup containers that are acked
                        removeOrTrackCompletedContainersFromContext(response
                                .getContainersToBeRemovedFromNM());

                        logAggregationReportForAppsTempList.clear();
                        lastHeartbeatID = response.getResponseId();
                        List<ContainerId> containersToCleanup = response
                                .getContainersToCleanup();
                        if (!containersToCleanup.isEmpty()) {
                            dispatcher.getEventHandler().handle(
                                    new
CMgrCompletedContainersEvent(containersToCleanup,
                                            CMgrCompletedContainersEvent.Reason
                                                    .BY_RESOURCEMANAGER));
                        }
                        List<ApplicationId> appsToCleanup =
                                response.getApplicationsToCleanup();
```

```
                        //Only start tracking for keepAlive on FINISH_APP
                        trackAppsForKeepAlive(appsToCleanup);
                        if (!appsToCleanup.isEmpty()) {
                            dispatcher.getEventHandler().handle(
                                    new CMgrCompletedAppsEvent(appsToCleanup,

 CMgrCompletedAppsEvent.Reason.BY_RESOURCEMANAGER));
                        }
                        Map<ApplicationId, ByteBuffer> systemCredentials =
                                response.getSystemCredentialsForApps();
                        if (systemCredentials != null &&
!systemCredentials.isEmpty()) {
                            ((NMContext) context).setSystemCrendentialsForApps(
                                    parseCredentials(systemCredentials));
                            context.getContainerManager().handleCredentialUpdate();
                        }
                        List<org.apache.hadoop.yarn.api.records.Container>
                                containersToUpdate =
response.getContainersToUpdate();
                        if (!containersToUpdate.isEmpty()) {
                            dispatcher.getEventHandler().handle(
                                    new
CMgrUpdateContainersEvent(containersToUpdate));
                        }

                        // SignalContainer request originally comes from end users
via
                        // ClientRMProtocol's SignalContainer. Forward the request
to
                        // ContainerManager which will dispatch the event to
                        // ContainerLauncher.
                        List<SignalContainerRequest> containersToSignal = response
                                .getContainersToSignalList();
                        if (!containersToSignal.isEmpty()) {
                            dispatcher.getEventHandler().handle(
                                    new
CMgrSignalContainersEvent(containersToSignal));
                        }

                        // Update QueuingLimits if ContainerManager supports queuing
                        ContainerQueuingLimit queuingLimit =
                                response.getContainerQueuingLimit();
                        if (queuingLimit != null) {

 context.getContainerManager().updateQueuingLimit(queuingLimit);
                        }
                    }
                    // Handling node resource update case.
                    Resource newResource = response.getResource();
                    if (newResource != null) {
                        updateNMResource(newResource);
                        if (LOG.isDebugEnabled()) {
```

```
                            LOG.debug("Node's resource is updated to " +
                                    newResource.toString());
                        }
                    }
                    if (timelineServiceV2Enabled) {
                        updateTimelineCollectorData(response);
                    }

                } catch (ConnectException e) {
                    //catch and throw the exception if tried MAX wait time to
connect RM
                    dispatcher.getEventHandler().handle(
                            new NodeManagerEvent(NodeManagerEventType.SHUTDOWN));
                    // failed to connect to RM.
                    failedToConnect = true;
                    throw new YarnRuntimeException(e);
                } catch (Exception e) {

                    // TODO Better error handling. Thread can die with the rest of
the
                    // NM still running.
                    LOG.error("Caught exception in status-updater", e);
                } finally {
                    synchronized (heartbeatMonitor) {
                        nextHeartBeatInterval = nextHeartBeatInterval <= 0 ?

 YarnConfiguration.DEFAULT_RM_NM_HEARTBEAT_INTERVAL_MS :
                                    nextHeartBeatInterval;
                        try {
                            heartbeatMonitor.wait(nextHeartBeatInterval);
                        } catch (InterruptedException e) {
                            // Do Nothing
                        }
                    }
                }
            }
        }
```

#### 5.1.2.1 调用 RM 的 ResourceTrackerService.nodeHeartbeat() 处理 NM 心跳

```
 @SuppressWarnings("unchecked")
    @Override
    public NodeHeartbeatResponse nodeHeartbeat(NodeHeartbeatRequest request)
            throws YarnException, IOException {

        NodeStatus remoteNodeStatus = request.getNodeStatus();
        /**
         * Here is the node heartbeat sequence...
         * 1. Check if it's a valid (i.e. not excluded) node
         * 2. Check if it's a registered node
         * 3. Check if it's a 'fresh' heartbeat i.e. not duplicate heartbeat
         * 4. Send healthStatus to RMNode
```

```
         * 5. Update node's labels if distributed Node Labels configuration is
enabled
         */

        NodeId nodeId = remoteNodeStatus.getNodeId();

        // 1. Check if it's a valid (i.e. not excluded) node, if not, see if it is
        // in decommissioning.
        if (!this.nodesListManager.isValidNode(nodeId.getHost())
                && !isNodeInDecommissioning(nodeId)) {
            String message =
                    "Disallowed NodeManager nodeId: " + nodeId + " hostname: "
                            + nodeId.getHost();
            LOG.info(message);
            return YarnServerBuilderUtils.newNodeHeartbeatResponse(
                    NodeAction.SHUTDOWN, message);
        }

        // 2. Check if it's a registered node
        RMNode rmNode = this.rmContext.getRMNodes().get(nodeId);
        if (rmNode == null) {
            /* node does not exist */
            String message = "Node not found resyncing " +
remoteNodeStatus.getNodeId();
            LOG.info(message);
            return
YarnServerBuilderUtils.newNodeHeartbeatResponse(NodeAction.RESYNC,
                    message);
        }

        // Send ping
        this.nmLivelinessMonitor.receivedPing(nodeId);
        this.decommissioningWatcher.update(rmNode, remoteNodeStatus);

        // 3. Check if it's a 'fresh' heartbeat i.e. not duplicate heartbeat
        NodeHeartbeatResponse lastNodeHeartbeatResponse =
rmNode.getLastNodeHeartBeatResponse();
        if (getNextResponseId(
                remoteNodeStatus.getResponseId()) == lastNodeHeartbeatResponse
                .getResponseId()) {
            LOG.info("Received duplicate heartbeat from node "
                    + rmNode.getNodeAddress() + " responseId=" +
remoteNodeStatus.getResponseId());
            return lastNodeHeartbeatResponse;
        } else if (remoteNodeStatus.getResponseId() != lastNodeHeartbeatResponse
                .getResponseId()) {
            String message =
                    "Too far behind rm response id:"
                            + lastNodeHeartbeatResponse.getResponseId() + " nm
response id:"
                            + remoteNodeStatus.getResponseId();
            LOG.info(message);
```

```
            // TODO: Just sending reboot is not enough. Think more.
            this.rmContext.getDispatcher().getEventHandler().handle(
                    new RMNodeEvent(nodeId, RMNodeEventType.REBOOTING));
            return
YarnServerBuilderUtils.newNodeHeartbeatResponse(NodeAction.RESYNC,
                    message);
        }

        // Evaluate whether a DECOMMISSIONING node is ready to be DECOMMISSIONED.
        if (rmNode.getState() == NodeState.DECOMMISSIONING &&
                decommissioningWatcher.checkReadyToBeDecommissioned(
                        rmNode.getNodeID())) {
            String message = "DECOMMISSIONING " + nodeId +
                    " is ready to be decommissioned";
            LOG.info(message);
            this.rmContext.getDispatcher().getEventHandler().handle(
                    new RMNodeEvent(nodeId, RMNodeEventType.DECOMMISSION));
            this.nmLivelinessMonitor.unregister(nodeId);
            return YarnServerBuilderUtils.newNodeHeartbeatResponse(
                    NodeAction.SHUTDOWN, message);
        }

        if (timelineServiceV2Enabled) {
            // Check & update collectors info from request.
            updateAppCollectorsMap(request);
        }

        // Heartbeat response
        NodeHeartbeatResponse nodeHeartBeatResponse =
                YarnServerBuilderUtils.newNodeHeartbeatResponse(

 getNextResponseId(lastNodeHeartbeatResponse.getResponseId()),
                        NodeAction.NORMAL, null, null, null, null,
nextHeartBeatInterval);
        rmNode.setAndUpdateNodeHeartbeatResponse(nodeHeartBeatResponse);

        populateKeys(request, nodeHeartBeatResponse);

        ConcurrentMap<ApplicationId, ByteBuffer> systemCredentials =
                rmContext.getSystemCredentialsForApps();
        if (!systemCredentials.isEmpty()) {
            nodeHeartBeatResponse.setSystemCredentialsForApps(systemCredentials);
        }

        if (timelineServiceV2Enabled) {
            // Return collectors' map that NM needs to know
            setAppCollectorsMapToResponse(rmNode.getRunningApps(),
                    nodeHeartBeatResponse);
        }

        // 4. Send status to RMNode, saving the latest response.
        RMNodeStatusEvent nodeStatusEvent =
```

```
                new RMNodeStatusEvent(nodeId, remoteNodeStatus);
        if (request.getLogAggregationReportsForApps() != null
                && !request.getLogAggregationReportsForApps().isEmpty()) {
            nodeStatusEvent.setLogAggregationReportsForApps(request
                    .getLogAggregationReportsForApps());
        }
        this.rmContext.getDispatcher().getEventHandler().handle(nodeStatusEvent);

        // 5. Update node's labels to RM's NodeLabelManager.
        if (isDistributedNodeLabelsConf && request.getNodeLabels() != null) {
            try {
                updateNodeLabelsFromNMReport(
                        NodeLabelsUtils.convertToStringSet(request.getNodeLabels()),
                        nodeId);
                nodeHeartBeatResponse.setAreNodeLabelsAcceptedByRM(true);
            } catch (IOException ex) {
                //ensure the error message is captured and sent across in response
                nodeHeartBeatResponse.setDiagnosticsMessage(ex.getMessage());
                nodeHeartBeatResponse.setAreNodeLabelsAcceptedByRM(false);
            }
        }

        // 6. check if node's capacity is load from dynamic-resources.xml
        // if so, send updated resource back to NM.
        String nid = nodeId.toString();
        Resource capability = loadNodeResourceFromDRConfiguration(nid);
        // sync back with new resource if not null.
        if (capability != null) {
            nodeHeartBeatResponse.setResource(capability);
        }

        // 7. Send Container Queuing Limits back to the Node. This will be used by
        // the node to truncate the number of Containers queued for execution.
        if (this.rmContext.getNodeManagerQueueLimitCalculator() != null) {
            nodeHeartBeatResponse.setContainerQueuingLimit(
                    this.rmContext.getNodeManagerQueueLimitCalculator()
                            .createContainerQueuingLimit());
        }
        return nodeHeartBeatResponse;
    }
```

#### 5.1.2.2 最终调用 CapacityScheduler.handle(NODE_UPDATE)

```
case NODE_UPDATE: {
                // NM 更新 也即 NM 发送心跳
                NodeUpdateSchedulerEvent nodeUpdatedEvent =
(NodeUpdateSchedulerEvent) event;
                nodeUpdate(nodeUpdatedEvent.getRMNode());
            }
```

```
@Override
```

```
    protected void nodeUpdate(RMNode rmNode) {
        long begin = System.nanoTime();
        try {
            readLock.lock();
            // 更新 NM 心跳时间
            setLastNodeUpdateTime(Time.now());
            // 处理心跳 (核心就是判断 NM 的哪些容器已经完成 RM 知道哪些资源可以使用案例)
            super.nodeUpdate(rmNode);
        } finally {
            readLock.unlock();
        }

        // Try to do scheduling
        if (!scheduleAsynchronously) {
            try {
                writeLock.lock();
                ActivitiesLogger.NODE.startNodeUpdateRecording(activitiesManager,
                        rmNode.getNodeID());

                // reset allocation and reservation stats before we start doing any
                // work
                updateSchedulerHealth(lastNodeUpdateTime, rmNode.getNodeID(),
                        CSAssignment.NULL_ASSIGNMENT);

                allocateContainersToNode(rmNode.getNodeID(), true);
                ActivitiesLogger.NODE.finishNodeUpdateRecording(activitiesManager,
                        rmNode.getNodeID());
            } finally {
                writeLock.unlock();
            }
        }

        long latency = System.nanoTime() - begin;
        CapacitySchedulerMetrics.getMetrics().addNodeUpdate(latency);
    }
```

```
/**
     * Process a heartbeat update from a node.
     *
     * @param nm The RMNode corresponding to the NodeManager
     */
    protected void nodeUpdate(RMNode nm) {
        if (LOG.isDebugEnabled()) {
            LOG.debug("nodeUpdate: " + nm +
                    " cluster capacity: " + getClusterResource());
        }

        // Process new container information
        // NOTICE: it is possible to not find the NodeID as a node can be
        // decommissioned at the same time. Skip updates if node is null.
        SchedulerNode schedulerNode = getNode(nm.getNodeID());
```

```
        // NM 完成容器资源被 RM 回收
        List<ContainerStatus> completedContainers = updateNewContainerInfo(nm,
                schedulerNode);

        // Notify Scheduler Node updated.
        if (schedulerNode != null) {
            schedulerNode.notifyNodeUpdate();
        }

        // Process completed containers
        // 处理 NM 完成的容器
        Resource releasedResources = Resource.newInstance(0, 0);
        int releasedContainers = updateCompletedContainers(completedContainers,
                releasedResources, nm.getNodeID(), schedulerNode);

        // If the node is decommissioning, send an update to have the total
        // resource equal to the used resource, so no available resource to
        // schedule.
        if (nm.getState() == NodeState.DECOMMISSIONING && schedulerNode != null) {
            this.rmContext
                    .getDispatcher()
                    .getEventHandler()
                    .handle(
                            new RMNodeResourceUpdateEvent(nm.getNodeID(),
ResourceOption

.newInstance(schedulerNode.getAllocatedResource(), 0)));
        }

        updateSchedulerHealthInformation(releasedResources, releasedContainers);
        if (schedulerNode != null) {
            updateNodeResourceUtilization(nm, schedulerNode);
        }

        // Now node data structures are up-to-date and ready for scheduling.
        if (LOG.isDebugEnabled()) {
            LOG.debug(
                    "Node being looked for scheduling " + nm + " availableResource:
" +
                            (schedulerNode == null ? "unknown (decommissioned)" :
                                    schedulerNode.getUnallocatedResource()));
        }
    }
```

#### 5.1.2.3 RM 分配资源给其他 Application 使用(NM 心跳上报已经完成的容器资源被 RM 回收之后)(调用 CapacityScheduler.allocateContainersToNode())

```
@Override
    protected void nodeUpdate(RMNode rmNode) {
        long begin = System.nanoTime();
        try {
            readLock.lock();
```

```
            // 更新 NM 心跳时间
            setLastNodeUpdateTime(Time.now());
            // 处理心跳 (核心就是判断 NM 的哪些容器已经完成 RM 知道哪些资源可以使用)
            super.nodeUpdate(rmNode);
        } finally {
            readLock.unlock();
        }

        // Try to do scheduling
        // 默认 false
        if (!scheduleAsynchronously) {
            try {
                writeLock.lock();
                ActivitiesLogger.NODE.startNodeUpdateRecording(activitiesManager,
                        rmNode.getNodeID());

                // reset allocation and reservation stats before we start doing any
                // work
                updateSchedulerHealth(lastNodeUpdateTime, rmNode.getNodeID(),
                        CSAssignment.NULL_ASSIGNMENT);

                // NM 上报完成的容器 RM 回收资源 这些资源可以被其他 Application 应用
                // 如果存在 Application 申请容器 但是 RM 集群没有充足 此处就会分配资源给那些
                // 等待资源的应用 (里面的代码很复杂)
                allocateContainersToNode(rmNode.getNodeID(), true);
                ActivitiesLogger.NODE.finishNodeUpdateRecording(activitiesManager,
                        rmNode.getNodeID());
            } finally {
                writeLock.unlock();
            }
        }

        long latency = System.nanoTime() - begin;
        CapacitySchedulerMetrics.getMetrics().addNodeUpdate(latency);
    }
```

```
// 最终调用叶子节点队列进行分配资源给该队列的容器申请 调用
RegularContainerAllocator.assignContainersOnNode()
/**
 * Allocate normal (new) containers, considers locality/label, etc. Using
 * delayed scheduling mechanism to get better locality allocation.
 */
public class RegularContainerAllocator extends AbstractContainerAllocator {
 private ContainerAllocation assignContainersOnNode(Resource clusterResource,
      FiCaSchedulerNode node, SchedulerRequestKey schedulerKey,
      RMContainer reservedContainer, SchedulingMode schedulingMode,
      ResourceLimits currentResoureLimits) {
    Priority priority = schedulerKey.getPriority();

    ContainerAllocation allocation;
    NodeType requestLocalityType = null;
```

```
// Data-local
// 数据本地性
PendingAsk nodeLocalAsk =
    application.getPendingAsk(schedulerKey, node.getNodeName());
if (nodeLocalAsk.getCount() > 0) {
  requestLocalityType = NodeType.NODE_LOCAL;
  allocation =
      assignNodeLocalContainers(clusterResource, nodeLocalAsk,
          node, schedulerKey, reservedContainer, schedulingMode,
          currentResoureLimits);
  if (Resources.greaterThan(rc, clusterResource,
      allocation.getResourceToBeAllocated(), Resources.none())) {
    allocation.requestLocalityType = requestLocalityType;
    return allocation;
  }
}

// Rack-local
// 机架
PendingAsk rackLocalAsk =
    application.getPendingAsk(schedulerKey, node.getRackName());
if (rackLocalAsk.getCount() > 0) {
  if (!appInfo.canDelayTo(schedulerKey, node.getRackName())) {
    ActivitiesLogger.APP.recordSkippedAppActivityWithoutAllocation(
        activitiesManager, node, application, priority,
        ActivityDiagnosticConstant.SKIP_PRIORITY_BECAUSE_OF_RELAX_LOCALITY);
    return ContainerAllocation.PRIORITY_SKIPPED;
  }

  requestLocalityType = requestLocalityType == null ?
      NodeType.RACK_LOCAL :
      requestLocalityType;

  allocation =
      assignRackLocalContainers(clusterResource, rackLocalAsk,
          node, schedulerKey, reservedContainer, schedulingMode,
          currentResoureLimits);
  if (Resources.greaterThan(rc, clusterResource,
      allocation.getResourceToBeAllocated(), Resources.none())) {
    allocation.requestLocalityType = requestLocalityType;
    return allocation;
  }
}

// Off-switch
// 跨机架
PendingAsk offSwitchAsk =
    application.getPendingAsk(schedulerKey, ResourceRequest.ANY);
if (offSwitchAsk.getCount() > 0) {
  if (!appInfo.canDelayTo(schedulerKey, ResourceRequest.ANY)) {
    ActivitiesLogger.APP.recordSkippedAppActivityWithoutAllocation(
```

```
            activitiesManager, node, application, priority,
            ActivityDiagnosticConstant.SKIP_PRIORITY_BECAUSE_OF_RELAX_LOCALITY);
        return ContainerAllocation.PRIORITY_SKIPPED;
      }

      requestLocalityType = requestLocalityType == null ?
          NodeType.OFF_SWITCH :
          requestLocalityType;

      allocation =
          assignOffSwitchContainers(clusterResource, offSwitchAsk,
              node, schedulerKey, reservedContainer, schedulingMode,
              currentResoureLimits);

      // When a returned allocation is LOCALITY_SKIPPED, since we're in
      // off-switch request now, we will skip this app w.r.t priorities
      if (allocation.getAllocationState() == AllocationState.LOCALITY_SKIPPED) {
        allocation = ContainerAllocation.APP_SKIPPED;
      }
      allocation.requestLocalityType = requestLocalityType;

      return allocation;
    }
    ActivitiesLogger.APP.recordSkippedAppActivityWithoutAllocation(
        activitiesManager, node, application, priority,
        ActivityDiagnosticConstant.PRIORITY_SKIPPED);
    return ContainerAllocation.PRIORITY_SKIPPED;
  }
}
```

# 六 AM 申请容器 (调用 RM 的 ApplicationMasterService.allocate())

```
//调用 AllocatorRunnable.run() 发送心跳给 RM 的 ApplicationMasterService 申请容器
// AM 发送心跳给 RM 申请容器 发送 RPC 请求给 ApplicationMasterService RPC 服务 但是可能容器申请不下来可能是资源不足 那么下次 AM 发送心跳的时候 ApplicationMasterService RPC 服务将会返回之前申请的容器资源给 AM 也即申请资源容器是通过心跳来过渡
```

```
@Override
    public AllocateResponse allocate(AllocateRequest request)
            throws YarnException, IOException {

        AMRMTokenIdentifier amrmTokenIdentifier =
                YarnServerSecurityUtils.authorizeRequest();

        ApplicationAttemptId appAttemptId =
                amrmTokenIdentifier.getApplicationAttemptId();

        this.amLivelinessMonitor.receivedPing(appAttemptId);
```

```
        /* check if its in cache */
        AllocateResponseLock lock = responseMap.get(appAttemptId);
        if (lock == null) {
            String message =
                    "Application attempt " + appAttemptId
                            + " doesn't exist in ApplicationMasterService cache.";
            LOG.error(message);
            throw new ApplicationAttemptNotFoundException(message);
        }
        synchronized (lock) {
            AllocateResponse lastResponse = lock.getAllocateResponse();
            if (!hasApplicationMasterRegistered(appAttemptId)) {
                String message =
                        "AM is not registered for known application attempt: "
                                + appAttemptId
                                + " or RM had restarted after AM registered. "
                                + " AM should re-register.";
                throw new ApplicationMasterNotRegisteredException(message);
            }

            // Normally request.getResponseId() == lastResponse.getResponseId()
            if (getNextResponseId(request.getResponseId()) == lastResponse
                    .getResponseId()) {
                // heartbeat one step old, simply return lastReponse
                return lastResponse;
            } else if (request.getResponseId() != lastResponse.getResponseId()) {
                String message =
                        "Invalid responseId in AllocateRequest from application
attempt: "
                                + appAttemptId + ", expect responseId to be "
                                + lastResponse.getResponseId() + ", but get "
                                + request.getResponseId();
                throw new InvalidApplicationMasterRequestException(message);
            }

            AllocateResponse response =
                    recordFactory.newRecordInstance(AllocateResponse.class);
            // 获取之前申请容器是否已经申请好 如果申请好了塞入 response 作为心跳返回
            this.amsProcessingChain.allocate(
                    amrmTokenIdentifier.getApplicationAttemptId(), request,
response);

            // update AMRMToken if the token is rolled-up
            MasterKeyData nextMasterKey =

 this.rmContext.getAMRMTokenSecretManager().getNextMasterKeyData();

            if (nextMasterKey != null
                    && nextMasterKey.getMasterKey().getKeyId() !=
amrmTokenIdentifier
                    .getKeyId()) {
                RMApp app =
```

```
 this.rmContext.getRMApps().get(appAttemptId.getApplicationId());
                RMAppAttempt appAttempt = app.getRMAppAttempt(appAttemptId);
                RMAppAttemptImpl appAttemptImpl = (RMAppAttemptImpl) appAttempt;
                Token<AMRMTokenIdentifier> amrmToken = appAttempt.getAMRMToken();
                if (nextMasterKey.getMasterKey().getKeyId() !=
                        appAttemptImpl.getAMRMTokenKeyId()) {
                    LOG.info("The AMRMToken has been rolled-over. Send new AMRMToken
back"
                            + " to application: " +
appAttemptId.getApplicationId());
                    amrmToken = rmContext.getAMRMTokenSecretManager()
                            .createAndGetAMRMToken(appAttemptId);
                    appAttemptImpl.setAMRMToken(amrmToken);
                }
                response.setAMRMToken(org.apache.hadoop.yarn.api.records.Token
                        .newInstance(amrmToken.getIdentifier(), amrmToken.getKind()
                                .toString(), amrmToken.getPassword(),
amrmToken.getService()
                                .toString()));
            }

            /*
             * As we are updating the response inside the lock object so we don't
             * need to worry about unregister call occurring in between (which
             * removes the lock object).
             */
            response.setResponseId(getNextResponseId(lastResponse.getResponseId()));
            lock.setAllocateResponse(response);
            return response;
        }
    }
```