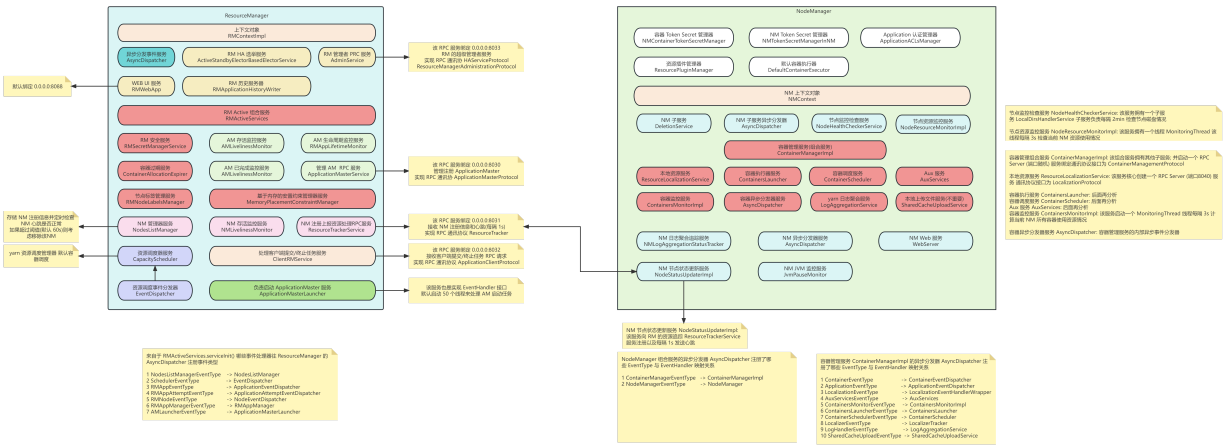
hadoop-yarn集群启动源码分析

hadoop-yarn集群启动源码分析

— ResourceManager 启动源码分析



1.1 源码入口类

```
/**
 * The ResourceManager is the main class that is a set of components.
 * "I am the ResourceManager. All your resources belong to us..."
 */
org.apache.hadoop.yarn.server.resourcemanager.ResourceManager
```

```
public static void main(String[] argv) {
        Thread.setDefaultUncaughtExceptionHandler(new
YarnUncaughtExceptionHandler());
        StringUtils.startupShutdownMessage(ResourceManager.class, argv, LOG);
        try {

            // 创建 YarnConfiguration (与 HDFS 类似)
            Configuration conf = new YarnConfiguration();

            // 解析参数
            GenericOptionsParser hParser = new GenericOptionsParser(conf, argv);
            argv = hParser.getRemainingArgs();
            // If -format-state-store, then delete RMStateStore; else startup
normally
            if (argv.length >= 1) {
                if (argv[0].equals("-format-state-store")) {
```

```
                deleteRMStateStore(conf);
            } else if (argv[0].equals("-remove-application-from-state-store")
                    && argv.length == 2) {
                removeApplication(conf, argv[1]);
            } else {
                printUsage(System.err);
            }
        } else {
            // 创建 ResourceManager
            ResourceManager resourceManager = new ResourceManager();
            ShutdownHookManager.get().addShutdownHook(
                    new CompositeServiceShutdownHook(resourceManager),
                    SHUTDOWN_HOOK_PRIORITY);
            // 初始化 ResourceManager (直接调用其 serviceInit())
            /** 来自于 RMActiveServices.serviceInit() 并执行如下服务的 serviceInit()
             * 1 RMSecretManagerService             (安全服务)
             * 2 ContainerAllocationExpirer         (容器过期服务)
             * 3 AMLivelinessMonitor                (ApplicationMaster 存活监控服
务)
             * 4 AMLivelinessMonitor                (ApplicationMaster 已完成监控服
务)
             * 5 RMAppLifetimeMonitor               (ApplicationMaster 生命周期监控
服务)
             * 6 RMNodeLabelsManager                (ResourceManager 节点标签管理服
务)
             * 7 MemoryPlacementConstraintManager   (基于内存的安置约束管理器服务)
             * 8 NodesListManager                   (NodeManager 管理器服务)
             * 9 CapacityScheduler                  (资源调度器服务)
             * 10 EventDispatcher                   (资源调度事件分发器)
             * 11 NMLivelinessMonitor               (NodeManager 存活监控服务)
             * 12 ResourceTrackerService            (NodeManager 上报资源处理服务)
             * 13 ApplicationMasterService          (管理 ApplicationMaster 服务)
             * 14 ClientRMService                   (客户端[可以提交任务、停止任务等
等]发送 RPC 请求处理服务)
             * 15 ApplicationMasterLauncher         (负责启动 ApplicationMaster 服
务)
             */
            /** 来自于 ResourceManager.serviceInit() 并执行如下服务的 serviceInit()
             * 1 AsyncDispatcher                    (异步中央事件分发器服务)
             * 2 AdminService                       (ResourceManager 管理者服务)
             * 3 ActiveStandbyElectorBasedElectorService (ResourceManager HA 选举
服务)
             * 4 RMApplicationHistoryWriter              (ResourceManager 历史服
务器)
             * 5 CombinedSystemMetricsPublisher          (监控发布服务)
             */
            resourceManager.init(conf);
            // 启动 ResourceManager (直接调用其 serviceStart())
            resourceManager.start();
        }
    } catch (Throwable t) {
        LOG.fatal("Error starting ResourceManager", t);
```

```
            System.exit(-1);
        }
    }
```

1.2 创建 ResourceManager

```
/**
 * The ResourceManager is the main class that is a set of components.
 * "I am the ResourceManager. All your resources belong to us..."
 */
@SuppressWarnings("unchecked")
public class ResourceManager extends CompositeService
        implements Recoverable, ResourceManagerMXBean {
        public ResourceManager() {
                // 往下追
                super("ResourceManager");
     }
}
```

```
/**
 * Composition of services.
 */
@Public
@Evolving
public class CompositeService extends AbstractService {
     public CompositeService(String name) {
        // 往下追
        super(name);
    }
}
```

```
/**
 * This is the base implementation class for services.
 */
@Public
@Evolving
public abstract class AbstractService implements Service {
  /**
     * Construct the service.
     *
     * @param name service name
     */
    public AbstractService(String name) {
        this.name = name;
        // 创建服务状态模型对象 ServiceStateModel
        stateModel = new ServiceStateModel(name);
    }
}
```

1.3 初始化 ResourceManager

```
// 初始化 ResourceManager (直接调用其 serviceInit())
    @Override
    protected void serviceInit(Configuration conf) throws Exception {
        this.conf = conf;
        UserGroupInformation.setConfiguration(conf);
        // 创建 ResourceManager 上下文对象 RMContextImpl
        this.rmContext = new RMContextImpl();
        rmContext.setResourceManager(this);

        // 默认返回 LocalConfigurationProvider (底层会加载 HDFS YARN 默认配置文件)
        this.configurationProvider =
                ConfigurationProviderFactory.getConfigurationProvider(conf);
        this.configurationProvider.init(this.conf);
        rmContext.setConfigurationProvider(configurationProvider);

        // load core-site.xml
        // 加载 core-site.xml
        loadConfigurationXml(YarnConfiguration.CORE_SITE_CONFIGURATION_FILE);

        // Do refreshSuperUserGroupsConfiguration with loaded core-site.xml
        // Or use RM specific configurations to overwrite the common ones first
        // if they exist
        RMServerUtils.processRMProxyUsersConf(conf);
        ProxyUsers.refreshSuperUserGroupsConfiguration(this.conf);

        // load yarn-site.xml
        // 加载 yarn-site.xml
        loadConfigurationXml(YarnConfiguration.YARN_SITE_CONFIGURATION_FILE);

        // 校验配置是否合法
        validateConfigs(this.conf);

        // Set HA configuration should be done before login
        // 判断 ResourceManager 是否启动 HA 模式
        this.rmContext.setHAEnabled(HAUtil.isHAEnabled(this.conf));
        if (this.rmContext.isHAEnabled()) {
            HAUtil.verifyAndSetConfiguration(this.conf);
        }

        // Set UGI and do login
        // If security is enabled, use login user
        // If security is not enabled, use current user
        this.rmLoginUGI = UserGroupInformation.getCurrentUser();
        try {
            doSecureLogin();
        } catch (IOException ie) {
            throw new YarnRuntimeException("Failed to login", ie);
```

```
        }

        // register the handlers for all AlwaysOn services using setupDispatcher().
        // 创建 AsyncDispatcher
        rmDispatcher = setupDispatcher();

        // ResourceManager(本身是一个服务) 是一个组合服务 故将 AsyncDispatcher 服务添加组合
服务
        addIfService(rmDispatcher);

        // ResourceManager 上下文对象引用 AsyncDispatcher
        rmContext.setDispatcher(rmDispatcher);

        // The order of services below should not be changed as services will be
        // started in same order
        // As elector service needs admin service to be initialized and started,
        // first we add admin service then elector service

        // 创建并添加 ResourceManager 管理者服务 AdminService
        adminService = createAdminService();
        addService(adminService);
        rmContext.setRMAdminService(adminService);

        // elector must be added post adminservice
        if (this.rmContext.isHAEnabled()) {
            // If the RM is configured to use an embedded leader elector,
            // initialize the leader elector.
            if (HAUtil.isAutomaticFailoverEnabled(conf)
                    && HAUtil.isAutomaticFailoverEmbedded(conf)) {
                // 如果 ResourceManager 启动 HA 模式
                // 默认返回 ActiveStandbyElectorBasedElectorService
                // 这种 HA 模式选举跟 HDFS 的 NameNode (ZKFC 维护 HA 选举对象) 类似, 故不在
介绍
                EmbeddedElector elector = createEmbeddedElector();
                // 类似 AsyncDispatcher 服务添加到 ResourceManager 组合服务
                addIfService(elector);
                rmContext.setLeaderElectorService(elector);
            }
        }

        rmContext.setYarnConfiguration(conf);

        // 创建并初始化 ResourceManager 是 Active 角色情况下对应的服务 RMActiveServices
(是一个组合服务)
        // RMActiveServices 初始化了很多组件 直接调用 RMActiveServices.serviceInit()
        // RMActiveServices 属于 ResourceManager 内部类
        createAndInitActiveServices(false);
        /** 来自于 RMActiveServices.serviceInit()
         * 哪些事件处理器往 ResourceManager 的 AsyncDispatcher 注册事件类型
         * 1 NodesListManagerEventType -> NodesListManager
         * 2 SchedulerEventType        -> EventDispatcher
         * 3 RMAppEventType            -> ApplicationEventDispatcher
```

```
         * 4 RMAppAttemptEventType     -> ApplicationAttemptEventDispatcher
         * 5 RMNodeEventType           -> NodeEventDispatcher
         * 6 RMAppManagerEventType     -> RMAppManager
         * 7 AMLauncherEventType       -> ApplicationMasterLauncher
         */

        /** 来自于 RMActiveServices.serviceInit() 并执行如下服务的 serviceInit()
         * 1 RMSecretManagerService            (安全服务)
         * 2 ContainerAllocationExpirer        (容器过期服务)
         * 3 AMLivelinessMonitor               (ApplicationMaster 存活监控服务)
         * 4 AMLivelinessMonitor               (ApplicationMaster 已完成监控服务)
         * 5 RMAppLifetimeMonitor              (ApplicationMaster 生命周期监控服务)
         * 6 RMNodeLabelsManager               (ResourceManager 节点标签管理服务)
         * 7 MemoryPlacementConstraintManager  (基于内存的安置约束管理器服务)
         * 8 NodesListManager                  (NodeManager 管理器服务)
         * 9 CapacityScheduler                 (资源调度器服务)
         * 10 EventDispatcher                  (资源调度事件分发器)
         * 11 NMLivelinessMonitor              (NodeManager 存活监控服务)
         * 12 ResourceTrackerService           (NodeManager 上报资源处理服务)
         * 13 ApplicationMasterService         (管理 ApplicationMaster 服务)
         * 14 ClientRMService                  (客户端[可以提交任务、停止任务等等]发送 RPC
请求处理服务)
         * 15 ApplicationMasterLauncher        (负责启动 ApplicationMaster 服务)
         */

        // ResourceMan WEB UI
        webAppAddress = WebAppUtils.getWebAppBindURL(this.conf,
                YarnConfiguration.RM_BIND_HOST,
                WebAppUtils.getRMWebAppURLWithoutScheme(this.conf));

        // 创建并添加 ResourceManager 历史服务器 RMApplicationHistoryWriter
        RMApplicationHistoryWriter rmApplicationHistoryWriter =
                createRMApplicationHistoryWriter();
        addService(rmApplicationHistoryWriter);
        rmContext.setRMApplicationHistoryWriter(rmApplicationHistoryWriter);

        // initialize the RM timeline collector first so that the system metrics
        // publisher can bind to it
        if (YarnConfiguration.timelineServiceV2Enabled(this.conf)) {
            RMTimelineCollectorManager timelineCollectorManager =
                    createRMTimelineCollectorManager();
            addService(timelineCollectorManager);
            rmContext.setRMTimelineCollectorManager(timelineCollectorManager);
        }

        // 系统监控发布服务
        SystemMetricsPublisher systemMetricsPublisher =
                createSystemMetricsPublisher();
        addIfService(systemMetricsPublisher);
        rmContext.setSystemMetricsPublisher(systemMetricsPublisher);

        registerMXBean();
```

```
        /**
         * 1 调用 AsyncDispatcher.serviceInit()
         * 2 调用 AdminService.serviceInit()
         * 3 调用 ActiveStandbyElectorBasedElectorService.serviceInit()
         * 4 调用 RMApplicationHistoryWriter.serviceInit()
         * 5 调用 CombinedSystemMetricsPublisher.serviceInit()
         */
        super.serviceInit(this.conf);
    }
```

1.3.1 创建 ResourceManager 上下文对象 RMContextImpl

```
/**
 * RMContextImpl class holds two services context.
 * <ul>
 * <li>serviceContext : These services called as <b>Always On</b> services.
 * Services that need to run always irrespective of the HA state of the RM.</li>
 * <li>activeServiceCotext : Active services context. Services that need to run
 * only on the Active RM.</li>
 * </ul>
 * <p>
 * <b>Note:</b> If any new service to be added to context, add it to a right
 * context as per above description.
 */
public class RMContextImpl implements RMContext {
    /**
     * Default constructor. To be used in conjunction with setter methods for
     * individual fields.
     */
    public RMContextImpl() {
        // 创建 RMServiceContext
        this.serviceContext = new RMServiceContext();
        // 创建 RMActiveServiceContext
        this.activeServiceContext = new RMActiveServiceContext();
    }
}
```

1.3.2 创建 AsyncDispatcher 服务并注册一个事件处理器

```
/**
     * Register the handlers for alwaysOn services
     */
    private Dispatcher setupDispatcher() {
        // 创建 AsyncDispatcher
        Dispatcher dispatcher = createDispatcher();
        // 往异步事件处理器注册事件类型为 RMFatalEventType, 事件处理函数为
RMFatalEventDispatcher
        dispatcher.register(
                RMFatalEventType.class,
                new ResourceManager.RMFatalEventDispatcher());
        // 返回 AsyncDispatcher
        return dispatcher;
    }
```

```
    protected Dispatcher createDispatcher() {
        // 创建 AsyncDispatcher
        return new AsyncDispatcher("RM Event dispatcher");
    }
```

```
**
 * Dispatches {@link Event}s in a separate thread. Currently only single thread
 * does that. Potentially there could be multiple channels for each event type
 * class and a thread pool can be used to dispatch the events.
 */
@SuppressWarnings("rawtypes")
@Public
@Evolving
public class AsyncDispatcher extends AbstractService implements Dispatcher {
    public AsyncDispatcher(BlockingQueue<Event> eventQueue) {
        // 往下追
        super("Dispatcher");
        // 链表阻塞队列 LinkedBlockingQueue
        this.eventQueue = eventQueue;
        // 枚举事件类型 (Event) 与其事件处理函数 (EventHandler) 关系注册表
        this.eventDispatchers = new HashMap<Class<? extends Enum>, EventHandler>();
    }
}
```

```
/**
     * Construct the service.
     *
     * @param name service name
     */
    public AbstractService(String name) {
        this.name = name;
        // 创建服务状态模型对象 ServiceStateModel
        stateModel = new ServiceStateModel(name);
    }
```

```
// 总结
// 1 异步事件分发驱动器 AsyncDispatcher 是 ResourceManager 组合服务的其中一个服务
// 2 AsyncDispatcher 注册了一个事件类型 RMFatalEventType 对应的事件处理器
RMFatalEventDispatcher
```

1.3.3 创建并添加 ResourceManager 管理者服务 AdminService

```
    protected AdminService createAdminService() {
        // 创建 AdminService
        return new AdminService(this);
    }
```

```
public class AdminService extends CompositeService implements
        HAServiceProtocol, ResourceManagerAdministrationProtocol {
        public AdminService(ResourceManager rm) {
        // 往下追
        super(AdminService.class.getName());
        // ResourceManager
        this.rm = rm;
    }
}
```

```
    public CompositeService(String name) {
        // 往下追
        super(name);
    }
```

```
// 总结
// 1 ResourceManager 管理者服务 AdminService 也是一个组合服务
// 2 管理者服务实现了两个核心 RPC 通讯协议接口：
// 2.1 HAServiceProtocol: HA 选举服务接口
// 2.2 ResourceManagerAdministrationProtocol: 超级管理员服务接口
```

1.3.4 创建并添加HA选举 ActiveStandbyElectorBasedElectorService 服务

```
protected EmbeddedElector createEmbeddedElector() throws IOException {
        EmbeddedElector elector;
        // 默认为 false
        curatorEnabled =
                conf.getBoolean(YarnConfiguration.CURATOR_LEADER_ELECTOR,
                        YarnConfiguration.DEFAULT_CURATOR_LEADER_ELECTOR_ENABLED);
        if (curatorEnabled) {
            this.zkManager = createAndStartZKManager(conf);
            elector = new CuratorBasedElectorService(this);
        } else {
            // 当 ResourceManager 启动 HA 模式
            // 默认创建 ActiveStandbyElectorBasedElectorService 服务来选举
            elector = new ActiveStandbyElectorBasedElectorService(this);
```

```
        }
        return elector;
    }
```

```
/**
 * Leader election implementation that uses {@link ActiveStandbyElector}.
 */
@InterfaceAudience.Private
@InterfaceStability.Unstable
public class ActiveStandbyElectorBasedElectorService extends AbstractService
    implements EmbeddedElector,
    ActiveStandbyElector.ActiveStandbyElectorCallback {
        ActiveStandbyElectorBasedElectorService(ResourceManager rm) {
        // 往下追
        super(ActiveStandbyElectorBasedElectorService.class.getName());
        // ResourceManager
        this.rm = rm;
      }
}
```

```
/**
     * Construct the service.
     *
     * @param name service name
     */
    public AbstractService(String name) {
        this.name = name;
        // 创建服务状态模型对象 ServiceStateModel
        stateModel = new ServiceStateModel(name);
    }
```

1.3.5 创建并初始化 RMActiveServices

```
// 说明
// 1 RMActiveServices 服务是 ResourceManager 为 Active 情况下服务, ResourceManager 在启动的时候就初始化该服务, 否则该服务一般不初始化其很多服务
// 2 无论 ResourceManager 处于 HA 模式还是非 HA 模式,该服务最终还是会启动的(只不过在 HA 模式下, 先把这个服务停止, 等到 HA 选举为 Active 后重新启动), 故这里先介绍
```

```
/**
     * Helper method to create and init {@link #activeServices}. This creates an
     * instance of {@link RMActiveServices} and initializes it.
     *
     * @param fromActive Indicates if the call is from the active state transition
     *                   or the RM initialization.
     */
    protected void createAndInitActiveServices(boolean fromActive) {
        // 创建 RMActiveServices
        activeServices = new RMActiveServices(this);
```

```
        // 默认 false
        activeServices.fromActive = fromActive;
        // 初始化 RMActiveServices 服务 (直接调用其服务的 serviceInit() )
        activeServices.init(conf);
    }
```

1.3.5.1 创建 RMActiveService

```
/**
 * The ResourceManager is the main class that is a set of components.
 * "I am the ResourceManager. All your resources belong to us..."
 */
@SuppressWarnings("unchecked")
public class ResourceManager extends CompositeService
        implements Recoverable, ResourceManagerMXBean {
        RMActiveServices(ResourceManager rm) {
            super("RMActiveServices");
            // ResourceManager
            this.rm = rm;
        }
}
```

1.3.5.2 初始化 RMActiveService

```
// 直接调用其服务的 serviceInit()
@Override
        protected void serviceInit(Configuration configuration) throws Exception {
            // 创建 StandByTransitionRunnable 线程 (Runnable 接口实现类)
            standByTransitionRunnable = new StandByTransitionRunnable();

            // 创建并添加 ResourceManager 安全服务 RMSecretManagerService
            rmSecretManagerService = createRMSecretManagerService();
            addService(rmSecretManagerService);

            // 创建并添加容器过期服务 ContainerAllocationExpirer
            // 比如 ResourceManager 分配了容器给其他计算程序(MR, Spark, Flink) 但是计算程序
迟迟没有
            // 使用 则 ResourceManager 需要回收
            containerAllocationExpirer = new
ContainerAllocationExpirer(rmDispatcher);
            addService(containerAllocationExpirer);
            rmContext.setContainerAllocationExpirer(containerAllocationExpirer);

            // 创建并添加 ApplicationMaster (Spark -> Driver) 存活监控服务
AMLivelinessMonitor
            AMLivelinessMonitor amLivelinessMonitor = createAMLivelinessMonitor();
            addService(amLivelinessMonitor);
            rmContext.setAMLivelinessMonitor(amLivelinessMonitor);

            // 同上 (只不过这个服务是监控 ApplicationMaster 已经完成) AMLivelinessMonitor
            AMLivelinessMonitor amFinishingMonitor = createAMLivelinessMonitor();
```

```
            addService(amFinishingMonitor);
            rmContext.setAMFinishingMonitor(amFinishingMonitor);

            // 创建并添加监控 ApplicationMaster 生命周期服务 RMAppLifetimeMonitor
            RMAppLifetimeMonitor rmAppLifetimeMonitor =
createRMAppLifetimeMonitor();
            addService(rmAppLifetimeMonitor);
            rmContext.setRMAppLifetimeMonitor(rmAppLifetimeMonitor);

            // 创建并添加 ResourceManager 节点标签管理服务 RMNodeLabelsManager
            RMNodeLabelsManager nlm = createNodeLabelManager();
            nlm.setRMContext(rmContext);
            addService(nlm);
            rmContext.setNodeLabelManager(nlm);

            // 创建分配标签管理对象 AllocationTagsManager
            AllocationTagsManager allocationTagsManager =
                    createAllocationTagsManager();
            rmContext.setAllocationTagsManager(allocationTagsManager);

            // 创建并添加 MemoryPlacementConstraintManager 服务 (基于内存的安置约束管理器)
            PlacementConstraintManagerService placementConstraintManager =
                    createPlacementConstraintManager();
            addService(placementConstraintManager);
            rmContext.setPlacementConstraintManager(placementConstraintManager);

            // add resource profiles here because it's used by AbstractYarnScheduler
            // 创建 ResourceProfilesManagerImpl (资源配置文件管理器 通常被资源调度器使用)
            ResourceProfilesManager resourceProfilesManager =
                    createResourceProfileManager();
            // 初始化 ResourceProfilesManagerImpl (默认啥也不干)
            resourceProfilesManager.init(conf);
            rmContext.setResourceProfilesManager(resourceProfilesManager);

            // 创建 ResourceManager 节点标签周期性更新服务 默认没有开启该服务
            RMDelegatedNodeLabelsUpdater delegatedNodeLabelsUpdater =
                    createRMDelegatedNodeLabelsUpdater();
            if (delegatedNodeLabelsUpdater != null) {
                addService(delegatedNodeLabelsUpdater);

 rmContext.setRMDelegatedNodeLabelsUpdater(delegatedNodeLabelsUpdater);
            }

            // 是否启动 ResourceManager 自动恢复 默认为 false
            // 但是一般启动 ResourceManager HA模式 该功能一般在 yarn-site.xml 文件配置为
true
            recoveryEnabled = conf.getBoolean(YarnConfiguration.RECOVERY_ENABLED,
                    YarnConfiguration.DEFAULT_RM_RECOVERY_ENABLED);

            RMStateStore rmStore = null;
            if (recoveryEnabled) {
                // 默认返回 FileSystemRMStateStore (反射创建)
```

```
                rmStore = RMStateStoreFactory.getStore(conf);
                // 默认 true
                boolean isWorkPreservingRecoveryEnabled =
                        conf.getBoolean(

 YarnConfiguration.RM_WORK_PRESERVING_RECOVERY_ENABLED,

 YarnConfiguration.DEFAULT_RM_WORK_PRESERVING_RECOVERY_ENABLED);
                rmContext

.setWorkPreservingRecoveryEnabled(isWorkPreservingRecoveryEnabled);
            } else {
                rmStore = new NullRMStateStore();
            }

            try {
                // 设置 ResourceManager
                rmStore.setResourceManager(rm);
                // 初始化 FileSystemRMStateStore 调用其父类 RMStateStore 的
serviceInit()
                // RMStateStore 底层创建 AsyncDispatcher 并添加一个
                // EventHandler(ResourceManager 简单状态机)
                rmStore.init(conf);
                rmStore.setRMDispatcher(rmDispatcher);
            } catch (Exception e) {
                // the Exception from stateStore.init() needs to be handled for
                // HA and we need to give up master status if we got fenced
                LOG.error("Failed to init state store", e);
                throw e;
            }
            rmContext.setStateStore(rmStore);

            // 是否启动用户组安全功能
            if (UserGroupInformation.isSecurityEnabled()) {
                delegationTokenRenewer = createDelegationTokenRenewer();
                rmContext.setDelegationTokenRenewer(delegationTokenRenewer);
            }

            // Register event handler for NodesListManager
            // 创建 NodeManager 节点管理器服务 NodesListManager
            // NodesListManager 同时也是一个 EventHandler
            nodesListManager = new NodesListManager(rmContext);
            // 往 ResourceManager 的异步事件分发器注册事件处理器 NodesListManager
            rmDispatcher.register(NodesListManagerEventType.class,
nodesListManager);
            addService(nodesListManager);
            rmContext.setNodesListManager(nodesListManager);

            // 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);

            // 创建资源调度事件分发器 EventDispatcher (是一个服务 Service 也是一个事件处理器
EventHandler)
            // EventDispatcher 与 AsyncDispatcher 类似
            schedulerDispatcher = createSchedulerEventDispatcher();
            // 添加服务
            addIfService(schedulerDispatcher);
            // 注册事件处理器
            rmDispatcher.register(SchedulerEventType.class, schedulerDispatcher);

            // Register event handler for RmAppEvents
            // 注册事件处理器 ApplicationEventDispatcher
            rmDispatcher.register(RMAppEventType.class,
                    // 创建 ApplicationEventDispatcher
                    new ApplicationEventDispatcher(rmContext));

            // Register event handler for RmAppAttemptEvents
            // 注册事件处理器 ApplicationAttemptEventDispatcher
            rmDispatcher.register(RMAppAttemptEventType.class,
                    // 创建 ApplicationAttemptEventDispatcher
                    new ApplicationAttemptEventDispatcher(rmContext));

            // Register event handler for RmNodes
            // 注册事件处理器 NodeEventDispatcher
            rmDispatcher.register(RMNodeEventType.class,
                    // 创建 NodeEventDispatcher
                    new NodeEventDispatcher(rmContext));

            // 创建并添加 NodeManager 节点存活监控服务 NMLivelinessMonitor
            nmLivelinessMonitor = createNMLivelinessMonitor();
            addService(nmLivelinessMonitor);

            // 创建并添加 NodeManager 上报资源处理 ResourceTrackerService 服务
            resourceTracker = createResourceTrackerService();
            addService(resourceTracker);
            rmContext.setResourceTrackerService(resourceTracker);

            // ResourceManager 监控相关
            MetricsSystem ms = DefaultMetricsSystem.initialize("ResourceManager");
            if (fromActive) {
                JvmMetrics.reattach(ms, jvmMetrics);
                UserGroupInformation.reattachMetrics();
            } else {
                jvmMetrics = JvmMetrics.initSingleton("ResourceManager", null);
            }
            JvmPauseMonitor pauseMonitor = new JvmPauseMonitor();
            addService(pauseMonitor);
            jvmMetrics.setPauseMonitor(pauseMonitor);
```

```
            // Initialize the Reservation system
            // 是否开启 ResourceManager 保护系统功能 默认为 false
            if (conf.getBoolean(YarnConfiguration.RM_RESERVATION_SYSTEM_ENABLE,
                    YarnConfiguration.DEFAULT_RM_RESERVATION_SYSTEM_ENABLE)) {
                reservationSystem = createReservationSystem();
                if (reservationSystem != null) {
                    reservationSystem.setRMContext(rmContext);
                    addIfService(reservationSystem);
                    rmContext.setReservationSystem(reservationSystem);
                    LOG.info("Initialized Reservation system");
                }
            }

            // 创建管理 ApplicationMaster 服务 ApplicationMasterService
            masterService = createApplicationMasterService();
            // 默认不开启里面功能
            createAndRegisterOpportunisticDispatcher(masterService);
            // 添加 ApplicationMasterService 服务
            addService(masterService);
            rmContext.setApplicationMasterService(masterService);

            // 创建 ApplicationACLsManager (ACL Access Control List)
            applicationACLsManager = new ApplicationACLsManager(conf);
            // 创建 QueueACLsManager
            queueACLsManager = createQueueACLsManager(scheduler, conf);

            // 创建并注册管理 Application 事件处理器 RMAppManager
            rmAppManager = createRMAppManager();
            // Register event handler for RMAppManagerEvents
            rmDispatcher.register(RMAppManagerEventType.class, rmAppManager);

            // 创建并添加 Application 客户端(可以提交任务、停止任务等等)发送 RPC 请求处理服务
ClientRMService
            // ClientRMService 服务实现 RPC 协议 ApplicationClientProtocol
            clientRM = createClientRMService();
            addService(clientRM);
            rmContext.setClientRMService(clientRM);

            // 创建 ApplicationMasterLauncher 服务 同时其也是一个事件处理器
            // 负责启动 ApplicationMaster 服务
            applicationMasterLauncher = createAMLauncher();
            // 注册事件处理器
            rmDispatcher.register(AMLauncherEventType.class,
                    applicationMasterLauncher);
            // 添加服务
            addService(applicationMasterLauncher);

            if (UserGroupInformation.isSecurityEnabled()) {
                addService(delegationTokenRenewer);
                delegationTokenRenewer.setRMContext(rmContext);
            }
```

```
            //  ResourceManager 是否开启联邦功能 默认不开启
            if (HAUtil.isFederationEnabled(conf)) {
                String cId = YarnConfiguration.getClusterId(conf);
                if (cId.isEmpty()) {
                    String errMsg =
                            "Cannot initialize RM as Federation is enabled"
                                    + " but cluster id is not configured.";
                    LOG.error(errMsg);
                    throw new YarnRuntimeException(errMsg);
                }
                federationStateStoreService = createFederationStateStoreService();
                addIfService(federationStateStoreService);
                LOG.info("Initialized Federation membership.");
            }

            // 创建 RMNMInfo (管理 NodeManager 状态列表信息)
            rmnmInfo = new RMNMInfo(rmContext, scheduler);

            if (conf.getBoolean(YarnConfiguration.YARN_API_SERVICES_ENABLE,
                    false)) {
                SystemServiceManager systemServiceManager = createServiceManager();
                addIfService(systemServiceManager);
            }

            /**
             * 哪些事件处理器往 ResourceManager 的 AsyncDispatcher 注册事件类型
             * 1 NodesListManagerEventType -> NodesListManager
             * 2 SchedulerEventType        -> EventDispatcher
             * 3 RMAppEventType            -> ApplicationEventDispatcher
             * 4 RMAppAttemptEventType     -> ApplicationAttemptEventDispatcher
             * 5 RMNodeEventType           -> NodeEventDispatcher
             * 6 RMAppManagerEventType     -> RMAppManager
             * 7 AMLauncherEventType       -> ApplicationMasterLauncher
             */

            // 初始化服务
            /**
             * 1 RMSecretManagerService            (安全服务)
             * 2 ContainerAllocationExpirer        (容器过期服务)
             * 3 AMLivelinessMonitor               (ApplicationMaster 存活监控服务)
             * 4 AMLivelinessMonitor               (ApplicationMaster 已完成监控服务)
             * 5 RMAppLifetimeMonitor              (ApplicationMaster 生命周期监控服务)
             * 6 RMNodeLabelsManager               (ResourceManager 节点标签管理服务)
             * 7 MemoryPlacementConstraintManager  (基于内存的安置约束管理器服务)
             * 8 NodesListManager                  (NodeManager 管理器服务)
             * 9 CapacityScheduler                 (资源调度器服务)
             * 10 EventDispatcher                  (资源调度事件分发器)
             * 11 NMLivelinessMonitor              (NodeManager 存活监控服务)
             * 12 ResourceTrackerService           (NodeManager 上报资源处理服务)
             * 13 ApplicationMasterService         (管理 ApplicationMaster 服务)
             * 14 ClientRMService                  (客户端[可以提交任务、停止任务等等]发送
RPC 请求处理服务)
```

```
             * 15 ApplicationMasterLauncher        (负责启动 ApplicationMaster 服务)
             */
            super.serviceInit(conf);
        }
```

1.3.5.2.1 创建并添加 ResourceManager 安全服务 RMSecretManagerService

```
    protected RMSecretManagerService createRMSecretManagerService() {
        // 创建 RMSecretManagerService
        return new RMSecretManagerService(conf, rmContext);
    }
```

```
public class RMSecretManagerService extends AbstractService {
    /**
     * Construct the service.
     *
     */
    public RMSecretManagerService(Configuration conf, RMContextImpl rmContext) {
        super(RMSecretManagerService.class.getName());
        this.rmContext = rmContext;

        // To initialize correctly, these managers should be created before
        // being called serviceInit().
        nmTokenSecretManager = createNMTokenSecretManager(conf);
        rmContext.setNMTokenSecretManager(nmTokenSecretManager);

        containerTokenSecretManager = createContainerTokenSecretManager(conf);
        rmContext.setContainerTokenSecretManager(containerTokenSecretManager);

        clientToAMSecretManager = createClientToAMTokenSecretManager();
        rmContext.setClientToAMTokenSecretManager(clientToAMSecretManager);

        amRmTokenSecretManager = createAMRMTokenSecretManager(conf, this.rmContext);
        rmContext.setAMRMTokenSecretManager(amRmTokenSecretManager);

        rmDTSecretManager =
                createRMDelegationTokenSecretManager(conf, rmContext);
        rmContext.setRMDelegationTokenSecretManager(rmDTSecretManager);
    }
}
```

1.3.5.2.2 创建并添加容器过期服务 ContainerAllocationExpirer

```
public class ContainerAllocationExpirer extends
        AbstractLivelinessMonitor<AllocationExpirationInfo> {
     public ContainerAllocationExpirer(Dispatcher d) {
            // 往下追
            super(ContainerAllocationExpirer.class.getName());
            // 获取 ResourceManager 的 AsyncDispatcher 的通用事件处理器
GenericEventHandler
            // GenericEventHandler 主要将 Event 添加到阻塞队列 (put)
            this.dispatcher = d.getEventHandler();
        }
}
```

1.3.5.2.3 创建并添加 ApplicationMaster 存活监控服务 AMLivelinessMonitor

```
    protected AMLivelinessMonitor createAMLivelinessMonitor() {
        // 创建 AMLivelinessMonitor
        return new AMLivelinessMonitor(this.rmDispatcher);
    }
```

```
public class AMLivelinessMonitor extends
AbstractLivelinessMonitor<ApplicationAttemptId> {
    public AMLivelinessMonitor(Dispatcher d) {
        // 往下追
        super("AMLivelinessMonitor");
        // 获取 ResourceManager 的 AsyncDispatcher 的通用事件处理器 GenericEventHandler
        // GenericEventHandler 主要将 Event 添加到阻塞队列 (put)
        this.dispatcher = d.getEventHandler();
    }
}
```

1.3.5.2.4 创建并添加 ApplicationMaster已完成监控服务 AMLivelinessMonitor

```
    protected AMLivelinessMonitor createAMLivelinessMonitor() {
        // 创建 AMLivelinessMonitor
        return new AMLivelinessMonitor(this.rmDispatcher);
    }
```

```
public class AMLivelinessMonitor extends
AbstractLivelinessMonitor<ApplicationAttemptId> {
    public AMLivelinessMonitor(Dispatcher d) {
        // 往下追
        super("AMLivelinessMonitor");
        // 获取 ResourceManager 的 AsyncDispatcher 的通用事件处理器 GenericEventHandler
        // GenericEventHandler 主要将 Event 添加到阻塞队列 (put)
        this.dispatcher = d.getEventHandler();
    }
}
```

1.3.5.2.5 创建并添加监控 ApplicationMaster 生命周期服务 RMAppLifetimeMonitor

```
    protected RMAppLifetimeMonitor createRMAppLifetimeMonitor() {
        // 创建 RMAppLifetimeMonitor
        return new RMAppLifetimeMonitor(this.rmContext);
    }
```

```
/**
 * This service will monitor the applications against the lifetime value given.
 * The applications will be killed if it running beyond the given time.
 */
public class RMAppLifetimeMonitor
        extends AbstractLivelinessMonitor<RMAppToMonitor> {
   public RMAppLifetimeMonitor(RMContext rmContext) {
        // 往下追
        super(RMAppLifetimeMonitor.class.getName(), SystemClock.getInstance());
        // ResourceManage 上下文对象
        this.rmContext = rmContext;
    }
}
```

1.3.5.2.6 创建并添加 ResourceManager 节点标签管理服务 RMNodeLabelsManager

```
    protected RMNodeLabelsManager createNodeLabelManager()
            throws InstantiationException, IllegalAccessException {
        // 创建 RMNodeLabelsManager
        return new RMNodeLabelsManager();
    }
```

```
public class RMNodeLabelsManager extends CommonNodeLabelsManager {
}
```

1.3.5.2.7 创建分配标签管理对象 AllocationTagsManager

```
    protected AllocationTagsManager createAllocationTagsManager() {
        // 创建 AllocationTagsManager
        return new AllocationTagsManager(this.rmContext);
    }
```

```
/**
 * In-memory mapping between applications/container-tags and nodes/racks.
 * Required by constrained affinity/anti-affinity and cardinality placement.
 */
@InterfaceAudience.Private
@InterfaceStability.Unstable
public class AllocationTagsManager {
    public AllocationTagsManager(RMContext context) {
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
        readLock = lock.readLock();
```

```
        writeLock = lock.writeLock();
        rmContext = context;
    }
}
```

1.3.5.2.8 创建并添加 MemoryPlacementConstraintManager 服务 (基于内存的安置约束管理器)

```
    createPlacementConstraintManager() {
        // Use the in memory Placement Constraint Manager.
        // 创建 MemoryPlacementConstraintManager
        return new MemoryPlacementConstraintManager();
    }
```

```
/**
 * In memory implementation of the {@link PlacementConstraintManagerService}.
 */
@InterfaceAudience.Private
@InterfaceStability.Unstable
public class MemoryPlacementConstraintManager
        extends PlacementConstraintManagerService {
    public MemoryPlacementConstraintManager() {
        this.globalConstraints = new HashMap<>();
        this.appConstraints = new HashMap<>();
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
        readLock = lock.readLock();
        writeLock = lock.writeLock();
    }
}
```

1.3.5.3.9 创建并初始化 ResourceProfilesManagerImpl (资源配置文件管理器 通常被资源调度器使用)

```
    @VisibleForTesting
    protected ResourceProfilesManager createResourceProfileManager() {
        // 创建 ResourceProfilesManagerImpl
        return new ResourceProfilesManagerImpl();
    }
```

```
/**
 * PBImpl class to handle all proto related implementation for
 * ResourceProfilesManager.
 */
public class ResourceProfilesManagerImpl implements ResourceProfilesManager {
    public ResourceProfilesManagerImpl() {
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
        readLock = lock.readLock();
        writeLock = lock.writeLock();
    }
}
```

```
  public void init(Configuration config) throws IOException {
    conf = config;
    // 加载资源配置文件 (一般情况下啥也不干)
    loadProfiles();
  }
```

1.3.5.3.10 创建并初始化 FileSystemRMStateStore (ResourceManager 简单状态机)

```
// 创建 FileSystemRMStateStore
public class RMStateStoreFactory {
    private static final Log LOG = LogFactory.getLog(RMStateStoreFactory.class);

    public static RMStateStore getStore(Configuration conf) {
        // 默认获取
org.apache.hadoop.yarn.server.resourcemanager.recovery.FileSystemRMStateStore
        Class<? extends RMStateStore> storeClass =
                conf.getClass(YarnConfiguration.RM_STORE,
                        MemoryRMStateStore.class, RMStateStore.class);
        LOG.info("Using RMStateStore implementation - " + storeClass);
        // 反射创建 FileSystemRMStateStore(无参构造函数)并调用其 setConf() 设置
Configuration
        return ReflectionUtils.newInstance(storeClass, conf);
    }
}
```

```
// 初始化 FileSystemRMStateStore 调用其父类 RMStateStore 的 serviceInit()
@Override
    protected void serviceInit(Configuration conf) throws Exception {
        // create async handler
        // 创建 AsyncDispatcher 服务
        dispatcher = new AsyncDispatcher("RM StateStore dispatcher");
        // 初始化 AsyncDispatcher 服务
        dispatcher.init(conf);

        // 注册事件处理函数 ForwardingEventHandler (本质上是一个状态机)
        rmStateStoreEventHandler = new ForwardingEventHandler();
        dispatcher.register(RMStateStoreEventType.class,
                rmStateStoreEventHandler);

        dispatcher.setDrainEventsOnStop();
        // read the base epoch value from conf
        baseEpoch = conf.getLong(YarnConfiguration.RM_EPOCH,
                YarnConfiguration.DEFAULT_RM_EPOCH);

        // 初始化内部 (调用 FileSystemRMStateStore.initInternal())
        initInternal(conf);
    }
```

```
/**
 * A simple class for storing RM state in any storage that implements a basic
```

```
 * FileSystem interface. Does not use directories so that simple key-value
 * stores can be used. The retry policy for the real filesystem client must be
 * configured separately to enable retry of filesystem operations when needed.
 *
 * Changes from 1.1 to 1.2, AMRMTokenSecretManager state has been saved
 * separately. The currentMasterkey and nextMasterkey have been stored.
 * Also, AMRMToken has been removed from ApplicationAttemptState.
 *
 * Changes from 1.2 to 1.3, Addition of ReservationSystem state.
 */
public class FileSystemRMStateStore extends RMStateStore {
      @Override
      public synchronized void initInternal(Configuration conf)
          throws Exception{
        fsWorkingPath = new Path(conf.get(YarnConfiguration.FS_RM_STATE_STORE_URI));
        rootDirPath = new Path(fsWorkingPath, ROOT_DIR_NAME);
        rmDTSecretManagerRoot = new Path(rootDirPath, RM_DT_SECRET_MANAGER_ROOT);
        rmAppRoot = new Path(rootDirPath, RM_APP_ROOT);
        amrmTokenSecretManagerRoot =
            new Path(rootDirPath, AMRMTOKEN_SECRET_MANAGER_ROOT);
        reservationRoot = new Path(rootDirPath, RESERVATION_SYSTEM_ROOT);
        fsNumRetries =
            conf.getInt(YarnConfiguration.FS_RM_STATE_STORE_NUM_RETRIES,
                YarnConfiguration.DEFAULT_FS_RM_STATE_STORE_NUM_RETRIES);
        fsRetryInterval =
            conf.getLong(YarnConfiguration.FS_RM_STATE_STORE_RETRY_INTERVAL_MS,
                    YarnConfiguration.DEFAULT_FS_RM_STATE_STORE_RETRY_INTERVAL_MS);
        intermediateEncryptionEnabled =
            conf.getBoolean(YarnConfiguration.YARN_INTERMEDIATE_DATA_ENCRYPTION,
              YarnConfiguration.DEFAULT_YARN_INTERMEDIATE_DATA_ENCRYPTION);
     }
}
```

1.3.5.3.11 创建 NodeManager 节点管理器服务 NodesListManager 并往 AsyncDispatcher 注册一个事件处理器

```
public class NodesListManager extends CompositeService implements
        EventHandler<NodesListManagerEvent> {
    public NodesListManager(RMContext rmContext) {
        // 往下追
        super(NodesListManager.class.getName());
        // ResourceManager 上下文对象
        this.rmContext = rmContext;
    }
}
```

```
// 往 ResourceManager 的异步事件分发器注册事件处理器 NodesListManager
rmDispatcher.register(NodesListManagerEventType.class, nodesListManager);
```

1.3.5.3.12 创建并添加资源调度器服务(默认 CapacityScheduler 对象)

```
protected ResourceScheduler createScheduler() {
        // 默认调度器为容量调度器
        //
org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler
        String schedulerClassName = conf.get(YarnConfiguration.RM_SCHEDULER,
                YarnConfiguration.DEFAULT_RM_SCHEDULER);
        LOG.info("Using Scheduler: " + schedulerClassName);
        try {
            Class<?> schedulerClazz = Class.forName(schedulerClassName);
            if (ResourceScheduler.class.isAssignableFrom(schedulerClazz)) {
                // 1 反射创建 CapacityScheduler 并调用其无参构造函数
                // 2 调用其 setConf() 设置 Configuration
                return (ResourceScheduler)
ReflectionUtils.newInstance(schedulerClazz,
                        this.conf);
            } else {
                throw new YarnRuntimeException("Class: " + schedulerClassName
                        + " not instance of " +
ResourceScheduler.class.getCanonicalName());
            }
        } catch (ClassNotFoundException e) {
            throw new YarnRuntimeException("Could not instantiate Scheduler: "
                    + schedulerClassName, e);
        }
    }
```

```
@LimitedPrivate("yarn")
@Evolving
@SuppressWarnings("unchecked")
public class CapacityScheduler extends
    AbstractYarnScheduler<FiCaSchedulerApp, FiCaSchedulerNode> implements
    PreemptableResourceScheduler, CapacitySchedulerContext, Configurable,
    ResourceAllocationCommitter, MutableConfScheduler {
      @Override
      public void setConf(Configuration conf) {
          yarnConf = conf;
    }
}
```

1.3.5.3.13 创建并添加资源调度事件分发器 EventDispatcher 并往 AsyncDispatcher 注册一个事件处理器

```
    protected EventHandler<SchedulerEvent> createSchedulerEventDispatcher() {
        // 创建 EventDispatcher
        return new EventDispatcher(this.scheduler, "SchedulerEventDispatcher");
    }
```

```
/**
 * This is a specialized EventHandler to be used by Services that are expected
```

```
 * handle a large number of events efficiently by ensuring that the caller
 * thread is not blocked. Events are immediately stored in a BlockingQueue and
 * a separate dedicated Thread consumes events from the queue and handles
 * appropriately
 * @param <T> Type of Event
 */
public class EventDispatcher<T extends Event> extends
    AbstractService implements EventHandler<T> {
   public EventDispatcher(EventHandler<T> handler, String name) {
    super(name);
    // 针对 ResourceManager 的资源调度器而言 handler = CapacityScheduler
    this.handler = handler;
    // 创建线程 EventProcessor
    this.eventProcessor = new Thread(new EventProcessor());
    this.eventProcessor.setName(getName() + ":Event Processor");
  }
}
```

```
// 注册事件处理器
rmDispatcher.register(SchedulerEventType.class, schedulerDispatcher);
```

1.3.5.3.14 创建 ApplicationEventDispatcher 并往 AsyncDispatcher 注册一个事件处理器

```
// Register event handler for RmAppEvents
            // 注册事件处理器 ApplicationEventDispatcher
            rmDispatcher.register(RMAppEventType.class,
                    // 创建 ApplicationEventDispatcher
                    new ApplicationEventDispatcher(rmContext));
```

```
public static final class ApplicationEventDispatcher implements
            EventHandler<RMAppEvent> {
        public ApplicationEventDispatcher(RMContext rmContext) {
            this.rmContext = rmContext;
        }
}
```

1.3.5.3.15 创建 ApplicationAttemptEventDispatcher 并往 AsyncDispatcher 注册一个事件处理器

```
// Register event handler for RmAppAttemptEvents
            // 注册事件处理器 ApplicationAttemptEventDispatcher
            rmDispatcher.register(RMAppAttemptEventType.class,
                    // 创建 ApplicationAttemptEventDispatcher
                    new ApplicationAttemptEventDispatcher(rmContext));
```

```
public static final class ApplicationAttemptEventDispatcher implements
            EventHandler<RMAppAttemptEvent> {
        public ApplicationAttemptEventDispatcher(RMContext rmContext) {
            this.rmContext = rmContext;
        }
}
```

1.3.5.3.16 创建 NodeEventDispatcher 并往 AsyncDispatcher 注册一个事件处理器

```
// Register event handler for RmNodes
            // 注册事件处理器 NodeEventDispatcher
            rmDispatcher.register(RMNodeEventType.class,
                    // 创建 NodeEventDispatcher
                    new NodeEventDispatcher(rmContext));
```

```
@Private
    public static final class NodeEventDispatcher implements
            EventHandler<RMNodeEvent> {
       public NodeEventDispatcher(RMContext rmContext) {
            this.rmContext = rmContext;
        }
}
```

1.3.5.3.17 创建并添加 NodeManager 节点存活监控服务 NMLivelinessMonitor

```
    private NMLivelinessMonitor createNMLivelinessMonitor() {
        // 创建 NMLivelinessMonitor
        return new NMLivelinessMonitor(this.rmContext
                .getDispatcher());
    }
```

```
public class NMLivelinessMonitor extends AbstractLivelinessMonitor<NodeId> {

    private EventHandler<Event> dispatcher;

    public NMLivelinessMonitor(Dispatcher d) {
        // 往下追
        super("NMLivelinessMonitor");
        // 获取 ResourceManager 的 AsyncDispatcher 的通用事件处理器 GenericEventHandler
        // GenericEventHandler 主要将 Event 添加到阻塞队列 (put)
        this.dispatcher = d.getEventHandler();
    }
}
```

1.3.5.3.18 创建并添加 NodeManager 上报资源处理 ResourceTrackerService 服务

```
    protected ResourceTrackerService createResourceTrackerService() {
        // 创建 ResourceTrackerService
        return new ResourceTrackerService(this.rmContext, this.nodesListManager,
                this.nmLivelinessMonitor,
                this.rmContext.getContainerTokenSecretManager(),
                this.rmContext.getNMTokenSecretManager());
    }
```

```
public class ResourceTrackerService extends AbstractService implements
        ResourceTracker {
    public ResourceTrackerService(RMContext rmContext,
                                  NodesListManager nodesListManager,
                                  NMLivelinessMonitor nmLivelinessMonitor,
                                  RMContainerTokenSecretManager
containerTokenSecretManager,
                                  NMTokenSecretManagerInRM nmTokenSecretManager) {
        super(ResourceTrackerService.class.getName());
        this.rmContext = rmContext;
        this.nodesListManager = nodesListManager;
        this.nmLivelinessMonitor = nmLivelinessMonitor;
        this.containerTokenSecretManager = containerTokenSecretManager;
        this.nmTokenSecretManager = nmTokenSecretManager;
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
        this.readLock = lock.readLock();
        this.writeLock = lock.writeLock();
        this.decommissioningWatcher = new DecommissioningNodesWatcher(rmContext);
    }
}
```

1.3.5.3.19 创建并添加管理 ApplicationMaster 服务 ApplicationMasterService

```
protected ApplicationMasterService createApplicationMasterService() {
        Configuration config = this.rmContext.getYarnConfiguration();
        if (isOpportunisticSchedulingEnabled(conf)) {
            if (YarnConfiguration.isDistSchedulingEnabled(config) &&
                    !YarnConfiguration
                            .isOpportunisticContainerAllocationEnabled(config)) {
                throw new YarnRuntimeException(
                        "Invalid parameters: opportunistic container allocation has
to " +
                                "be enabled when distributed scheduling is
enabled.");
            }
            OpportunisticContainerAllocatorAMService
                    oppContainerAllocatingAMService =
                    new OpportunisticContainerAllocatorAMService(this.rmContext,
                            scheduler);
            this.rmContext.setContainerQueueLimitCalculator(
```

```
 oppContainerAllocatingAMService.getNodeManagerQueueLimitCalculator());
            return oppContainerAllocatingAMService;
        }
        // 一般情况下 创建 ApplicationMasterService
        return new ApplicationMasterService(this.rmContext, scheduler);
    }
```

```
public class ApplicationMasterService extends AbstractService implements
    ApplicationMasterProtocol {
    public ApplicationMasterService(RMContext rmContext,
      YarnScheduler scheduler) {
        // 往下追
        this(ApplicationMasterService.class.getName(), rmContext, scheduler);
  }
}
```

```
  public ApplicationMasterService(String name, RMContext rmContext,
      YarnScheduler scheduler) {
    super(name);
    this.amLivelinessMonitor = rmContext.getAMLivelinessMonitor();
    this.rScheduler = scheduler;
    this.rmContext = rmContext;
    this.amsProcessingChain = new AMSProcessingChain(new DefaultAMSProcessor());
  }
```

1.3.5.3.20 创建 ApplicationACLsManager (ACL Access Control List)

```
@InterfaceAudience.Private
public class ApplicationACLsManager {
  public ApplicationACLsManager(Configuration conf) {
    this.conf = conf;
    this.adminAclsManager = new AdminACLsManager(this.conf);
  }
}
```

1.3.5.3.21 创建 QueueACLsManager

```
    protected QueueACLsManager createQueueACLsManager(ResourceScheduler scheduler,
                                                      Configuration conf) {
        // 创建 QueueACLsManager
        return new QueueACLsManager(scheduler, conf);
    }
```

```
public class QueueACLsManager {
  public QueueACLsManager(ResourceScheduler scheduler, Configuration conf) {
    this.scheduler = scheduler;
    this.isACLsEnable = conf.getBoolean(YarnConfiguration.YARN_ACL_ENABLE,
        YarnConfiguration.DEFAULT_YARN_ACL_ENABLE);
    this.authorizer = YarnAuthorizationProvider.getInstance(conf);
  }
}
```

1.3.5.3.22 创建管理 Application 事件处理器 RMAppManager 并往 AsyncDispatcher 注册一个事件处理器

```
    protected RMAppManager createRMAppManager() {
        // 创建 RMAppManager
        return new RMAppManager(this.rmContext, this.scheduler, this.masterService,
                this.applicationACLsManager, this.conf);
    }
```

```
/**
 * This class manages the list of applications for the resource manager.
 */
public class RMAppManager implements EventHandler<RMAppManagerEvent>,
                                     Recoverable {
      public RMAppManager(RMContext context,
      YarnScheduler scheduler, ApplicationMasterService masterService,
      ApplicationACLsManager applicationACLsManager, Configuration conf) {
        this.rmContext = context;
        this.scheduler = scheduler;
        this.masterService = masterService;
        this.applicationACLsManager = applicationACLsManager;
        this.conf = conf;
        this.maxCompletedAppsInMemory = conf.getInt(
            YarnConfiguration.RM_MAX_COMPLETED_APPLICATIONS,
            YarnConfiguration.DEFAULT_RM_MAX_COMPLETED_APPLICATIONS);
        this.maxCompletedAppsInStateStore =
            conf.getInt(
              YarnConfiguration.RM_STATE_STORE_MAX_COMPLETED_APPLICATIONS,
              this.maxCompletedAppsInMemory);
        if (this.maxCompletedAppsInStateStore > this.maxCompletedAppsInMemory) {
          this.maxCompletedAppsInStateStore = this.maxCompletedAppsInMemory;
        }
        this.authorizer = YarnAuthorizationProvider.getInstance(conf);
        this.timelineServiceV2Enabled = YarnConfiguration.
            timelineServiceV2Enabled(conf);
    }
 }
```

```
// Register event handler for RMAppManagerEvents
rmDispatcher.register(RMAppManagerEventType.class, rmAppManager);
```

1.3.5.3.23 创建并添加 Application 客户端(可以提交任务、停止任务等等)发送 RPC 请求处理服务 ClientRMService

```
    protected ClientRMService createClientRMService() {
        // 创建 ClientRMService
        return new ClientRMService(this.rmContext, scheduler, this.rmAppManager,
                this.applicationACLsManager, this.queueACLsManager,
                this.rmContext.getRMDelegationTokenSecretManager());
    }
```

```
/**
 * The client interface to the Resource Manager. This module handles all the rpc
 * interfaces to the resource manager from the client.
 */
public class ClientRMService extends AbstractService implements
    ApplicationClientProtocol {
    public ClientRMService(RMContext rmContext, YarnScheduler scheduler,
      RMAppManager rmAppManager, ApplicationACLsManager applicationACLsManager,
      QueueACLsManager queueACLsManager,
      RMDelegationTokenSecretManager rmDTSecretManager) {
    // 往下追
    this(rmContext, scheduler, rmAppManager, applicationACLsManager,
        queueACLsManager, rmDTSecretManager, new UTCClock());
  }
}
```

```
public ClientRMService(RMContext rmContext, YarnScheduler scheduler,
      RMAppManager rmAppManager, ApplicationACLsManager applicationACLsManager,
      QueueACLsManager queueACLsManager,
      RMDelegationTokenSecretManager rmDTSecretManager, Clock clock) {
    super(ClientRMService.class.getName());
    this.scheduler = scheduler;
    this.rmContext = rmContext;
    this.rmAppManager = rmAppManager;
    this.applicationsACLsManager = applicationACLsManager;
    this.queueACLsManager = queueACLsManager;
    this.rmDTSecretManager = rmDTSecretManager;
    this.reservationSystem = rmContext.getReservationSystem();
    this.clock = clock;
    this.rValidator = new ReservationInputValidator(clock);
    resourceProfilesManager = rmContext.getResourceProfilesManager();
  }
```

1.3.5.3.24 创建并添加负责启动 ApplicationMaster 服务 ApplicationMasterLauncher 并往 AsyncDispatcher 注册一个事件处理器

```
    protected ApplicationMasterLauncher createAMLauncher() {
        // 创建 ApplicationMasterLauncher
        return new ApplicationMasterLauncher(this.rmContext);
    }
```

```
public class ApplicationMasterLauncher extends AbstractService implements
        EventHandler<AMLauncherEvent> {
     public ApplicationMasterLauncher(RMContext context) {
        super(ApplicationMasterLauncher.class.getName());
        this.context = context;
        // 创建 LauncherThread
        this.launcherHandlingThread = new LauncherThread();
    }
}
```

```
// 注册事件处理器
rmDispatcher.register(AMLauncherEventType.class,applicationMasterLauncher);
```

1.3.5.3.25 创建 RMNMInfo (管理 NodeManager 状态列表信息)

```
/**
 * JMX bean listing statuses of all node managers.
 */
public class RMNMInfo implements RMNMInfoBeans {
    /**
   * Constructor for RMNMInfo registers the bean with JMX.
   *
   * @param rmc resource manager's context object
   * @param sched resource manager's scheduler object
   */
  public RMNMInfo(RMContext rmc, ResourceScheduler sched) {
    this.rmContext = rmc;
    this.scheduler = sched;

    StandardMBean bean;
    try {
      bean = new StandardMBean(this, RMNMInfoBeans.class);
      mbeanObjectName = MBeans.register("ResourceManager", "RMNMInfo", bean);
    } catch (NotCompliantMBeanException e) {
      LOG.warn("Error registering RMNMInfo MBean", e);
    }
    LOG.info("Registered RMNMInfo MBean");
  }
}
```

1.3.5.3.26 总结

```
/**
             * 哪些事件处理器往 ResourceManager 的 AsyncDispatcher 注册事件类型
             * 1 NodesListManagerEventType -> NodesListManager
             * 2 SchedulerEventType        -> EventDispatcher
             * 3 RMAppEventType            -> ApplicationEventDispatcher
             * 4 RMAppAttemptEventType     -> ApplicationAttemptEventDispatcher
             * 5 RMNodeEventType           -> NodeEventDispatcher
             * 6 RMAppManagerEventType     -> RMAppManager
```

```
             * 7 AMLauncherEventType       -> ApplicationMasterLauncher
             */

            // 初始化服务
            /**
             * 1 RMSecretManagerService            (安全服务)
             * 2 ContainerAllocationExpirer        (容器过期服务)
             * 3 AMLivelinessMonitor               (ApplicationMaster 存活监控服务)
             * 4 AMLivelinessMonitor               (ApplicationMaster 已完成监控服务)
             * 5 RMAppLifetimeMonitor              (ApplicationMaster 生命周期监控服务)
             * 6 RMNodeLabelsManager               (ResourceManager 节点标签管理服务)
             * 7 MemoryPlacementConstraintManager  (基于内存的安置约束管理器服务)
             * 8 NodesListManager                  (NodeManager 管理器服务)
             * 9 CapacityScheduler                 (资源调度器服务)
             * 10 EventDispatcher                  (资源调度事件分发器)
             * 11 NMLivelinessMonitor              (NodeManager 存活监控服务)
             * 12 ResourceTrackerService           (NodeManager 上报资源处理服务)
             * 13 ApplicationMasterService         (管理 ApplicationMaster 服务)
             * 14 ClientRMService                  (客户端[可以提交任务、停止任务等等]发送
RPC 请求处理服务)
             * 15 ApplicationMasterLauncher        (负责启动 ApplicationMaster 服务)
             */
```

1.3.5.3 调用 RMActiveService 组合服务的所有子服务 serviceInit()

1.3.5.3.1 安全服务 RMSecretManagerService.serviceInit()

```
    @Override
    public void serviceInit(Configuration conf) throws Exception {
        // 啥也不干
        super.serviceInit(conf);
    }
```

1.3.5.3.2 容器过期监控服务ContainerAllocationExpirer.serviceInit()

```
    public void serviceInit(Configuration conf) throws Exception {
        // 容器过期时间 默认 600 * 1000L 也即 5 分钟
        int expireIntvl = conf.getInt(
                YarnConfiguration.RM_CONTAINER_ALLOC_EXPIRY_INTERVAL_MS,
                YarnConfiguration.DEFAULT_RM_CONTAINER_ALLOC_EXPIRY_INTERVAL_MS);
        setExpireInterval(expireIntvl);
        // 默认 200s 检查
        setMonitorInterval(expireIntvl / 3);
        super.serviceInit(conf);
    }
```

1.3.5.3.3 ApplicationMaster 存活监控服务 AMLivelinessMonitor.serviceInit()

```
    public void serviceInit(Configuration conf) throws Exception {
        super.serviceInit(conf);
        // ApplicationMaster 存活过期时间 默认 5 分钟 (ApplicationMaster 持续上报以便检查不过期)
        int expireIntvl = conf.getInt(YarnConfiguration.RM_AM_EXPIRY_INTERVAL_MS,
                YarnConfiguration.DEFAULT_RM_AM_EXPIRY_INTERVAL_MS);
        setExpireInterval(expireIntvl);
        // 默认每隔 200s 检查
        setMonitorInterval(expireIntvl / 3);
    }
```

1.3.5.3.4 ApplicationMaster 已完成监控服务 AMLivelinessMonitor.serviceInit()

```
    public void serviceInit(Configuration conf) throws Exception {
        super.serviceInit(conf);
        // ApplicationMaster 存活过期时间 默认 5 分钟 (ApplicationMaster 持续上报以便检查不过期)
        int expireIntvl = conf.getInt(YarnConfiguration.RM_AM_EXPIRY_INTERVAL_MS,
                YarnConfiguration.DEFAULT_RM_AM_EXPIRY_INTERVAL_MS);
        setExpireInterval(expireIntvl);
        // 默认每隔 200s 检查
        setMonitorInterval(expireIntvl / 3);
    }
```

1.3.5.3.5 ApplicationMaster 生命周期监控服务 RMAppLifetimeMonitor.serviceInit()

```
@Override
    protected void serviceInit(Configuration conf) throws Exception {
        // 默认监控 ApplicationMaster 周期 3000ms
        long monitorInterval =
                conf.getLong(YarnConfiguration.RM_APPLICATION_MONITOR_INTERVAL_MS,

 YarnConfiguration.DEFAULT_RM_APPLICATION_MONITOR_INTERVAL_MS);
        if (monitorInterval <= 0) {
            monitorInterval =
                    YarnConfiguration.DEFAULT_RM_APPLICATION_MONITOR_INTERVAL_MS;
        }
        setMonitorInterval(monitorInterval);
        setExpireInterval(0); // No need of expire interval for App.
        setResetTimeOnStart(false); // do not reset expire time on restart
        LOG.info("Application lifelime monitor interval set to " + monitorInterval
                + " ms.");
        super.serviceInit(conf);
    }
```

1.3.5.3.6 ResourceManager 节点标签管理服务 RMNodeLabelsManager.serviceInit()

```
    @Override
    protected void serviceInit(Configuration conf) throws Exception {
        // 没啥重要的事
        super.serviceInit(conf);
        authorizer = YarnAuthorizationProvider.getInstance(conf);
    }
```

1.3.5.3.7 基于内存的安置约束管理器服务 MemoryPlacementConstraintManager.serviceInit()

```
    @Override
    protected void serviceInit(Configuration conf) throws Exception {
        // 啥也不干
        super.serviceInit(conf);
    }
```

1.3.5.3.8 NodeManager 管理器服务 NodesListManager.serviceInit()

```
    public NodesListManager(RMContext rmContext) {
        // 往下追
        super(NodesListManager.class.getName());
        // ResourceManager 上下文对象
        this.rmContext = rmContext;
    }
```

1.3.5.3.9 资源调度器服务 CapacityScheduler.serviceInit()

```
@Override
    public void serviceInit(Configuration conf) throws Exception {
        Configuration configuration = new Configuration(conf);
        super.serviceInit(conf);
        // 初始化容器调度器
        initScheduler(configuration);
        // Initialize SchedulingMonitorManager
        schedulingMonitorManager.initialize(rmContext, conf);
    }
```

```
@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();
```

```
        }
    }
```

1.3.5.3.10 资源调度事件分发器 EventDispatcher.serviceInit()

```
// 调用其父类 AbstractService.serviceInit()
/**
     * All initialization code needed by a service.
     * <p>
     * This method will only ever be called once during the lifecycle of
     * a specific service instance.
     * <p>
     * Implementations do not need to be synchronized as the logic
     * in {@link #init(Configuration)} prevents re-entrancy.
     * <p>
     * The base implementation checks to see if the subclass has created
     * a new configuration instance, and if so, updates the base class value
     *
     * @param conf configuration
     * @throws Exception on a failure -these will be caught,
     *                   possibly wrapped, and wil; trigger a service stop
     */
    protected void serviceInit(Configuration conf) throws Exception {
        if (conf != config) {
            LOG.debug("Config has been overridden during init");
            setConfig(conf);
        }
    }
```

1.3.5.3.11 NodeManager 存活监控服务 NMLivelinessMonitor.serviceInit()

```
    public void serviceInit(Configuration conf) throws Exception {
        // 默认 NodeManager 过期时间 600s
        int expireIntvl = conf.getInt(YarnConfiguration.RM_NM_EXPIRY_INTERVAL_MS,
                YarnConfiguration.DEFAULT_RM_NM_EXPIRY_INTERVAL_MS);
        setExpireInterval(expireIntvl);
        setMonitorInterval(expireIntvl / 3);
        super.serviceInit(conf);
    }
```

1.3.5.3.12 NodeManager 注册上报资源处理服务 ResourceTrackerService.serviceInit()

```
@Override
    protected void serviceInit(Configuration conf) throws Exception {
        // 默认 0.0.0.0:8031
        resourceTrackerAddress = conf.getSocketAddr(
                YarnConfiguration.RM_BIND_HOST,
                YarnConfiguration.RM_RESOURCE_TRACKER_ADDRESS,
                YarnConfiguration.DEFAULT_RM_RESOURCE_TRACKER_ADDRESS,
                YarnConfiguration.DEFAULT_RM_RESOURCE_TRACKER_PORT);
```

```
        RackResolver.init(conf);
        // NodeManager 上报心跳间隔 默认 1s
        nextHeartBeatInterval =
                conf.getLong(YarnConfiguration.RM_NM_HEARTBEAT_INTERVAL_MS,
                        YarnConfiguration.DEFAULT_RM_NM_HEARTBEAT_INTERVAL_MS);
        if (nextHeartBeatInterval <= 0) {
            throw new YarnRuntimeException("Invalid Configuration. "
                    + YarnConfiguration.RM_NM_HEARTBEAT_INTERVAL_MS
                    + " should be larger than 0.");
        }

        // 容器申请最小内存 默认 1024 MB
        minAllocMb = conf.getInt(
                YarnConfiguration.RM_SCHEDULER_MINIMUM_ALLOCATION_MB,
                YarnConfiguration.DEFAULT_RM_SCHEDULER_MINIMUM_ALLOCATION_MB);
        // 容器申请最小 CPU 默认 1
        minAllocVcores = conf.getInt(
                YarnConfiguration.RM_SCHEDULER_MINIMUM_ALLOCATION_VCORES,
                YarnConfiguration.DEFAULT_RM_SCHEDULER_MINIMUM_ALLOCATION_VCORES);

        // 默认 NONE
        minimumNodeManagerVersion = conf.get(
                YarnConfiguration.RM_NODEMANAGER_MINIMUM_VERSION,
                YarnConfiguration.DEFAULT_RM_NODEMANAGER_MINIMUM_VERSION);
        timelineServiceV2Enabled = YarnConfiguration.
                timelineServiceV2Enabled(conf);

        // 是否开启节点标签功能 默认不开启
        if (YarnConfiguration.areNodeLabelsEnabled(conf)) {
            isDistributedNodeLabelsConf =
                    YarnConfiguration.isDistributedNodeLabelConfiguration(conf);
            isDelegatedCentralizedNodeLabelsConf =

YarnConfiguration.isDelegatedCentralizedNodeLabelConfiguration(conf);
        }

        // 动态加载资源配置文件 dynamic-resources.xml
        loadDynamicResourceConfiguration(conf);

        // 启动 NodeManager 下线监控线程
        decommissioningWatcher.init(conf);
        super.serviceInit(conf);
    }
```

```
/**
 * DecommissioningNodesWatcher is used by ResourceTrackerService to track
 * DECOMMISSIONING nodes to decide when, after all running containers on
 * the node have completed, will be transitioned into DECOMMISSIONED state
 * (NodeManager will be told to shutdown).
 * Under MR application, a node, after completes all its containers,
 * may still serve it map output data during the duration of the application
```

```
 * for reducers. A fully graceful mechanism would keep such DECOMMISSIONING
 * nodes until all involved applications complete. It could be however
 * undesirable under long-running applications scenario where a bunch
 * of "idle" nodes would stay around for long period of time.
 *
 * DecommissioningNodesWatcher balance such concern with a timeout policy ---
 * a DECOMMISSIONING node will be DECOMMISSIONED no later than
 * DECOMMISSIONING_TIMEOUT regardless of running containers or applications.
 *
 * To be efficient, DecommissioningNodesWatcher skip tracking application
 * containers on a particular node before the node is in DECOMMISSIONING state.
 * It only tracks containers once the node is in DECOMMISSIONING state.
 * DecommissioningNodesWatcher basically is no cost when no node is
 * DECOMMISSIONING. This sacrifices the possibility that the node once
 * host containers of an application that is still running
 * (the affected map tasks will be rescheduled).
 */
public class DecommissioningNodesWatcher {
  public void init(Configuration conf) {
    // 默认 20
    int v = conf.getInt(
        YarnConfiguration.RM_DECOMMISSIONING_NODES_WATCHER_POLL_INTERVAL,
        YarnConfiguration
          .DEFAULT_RM_DECOMMISSIONING_NODES_WATCHER_POLL_INTERVAL);
    // 定时调度执行 PollTimerTask
    pollTimer.schedule(new PollTimerTask(rmContext), 0, (1000L * v));
  }
}
```

1.3.5.3.13 管理 ApplicationMaster 服务 ApplicationMasterService.serviceInit()

```
  @Override
  protected void serviceInit(Configuration conf) throws Exception {
    // 默认 0.0.0.0:8030
    masterServiceAddress = conf.getSocketAddr(
        YarnConfiguration.RM_BIND_HOST,
        YarnConfiguration.RM_SCHEDULER_ADDRESS,
        YarnConfiguration.DEFAULT_RM_SCHEDULER_ADDRESS,
        YarnConfiguration.DEFAULT_RM_SCHEDULER_PORT);

    // 初始化
    initializeProcessingChain(conf);
  }
```

```
private void initializeProcessingChain(Configuration conf) {
    amsProcessingChain.init(rmContext, null);
    addPlacementConstraintHandler(conf);

    List<ApplicationMasterServiceProcessor> processors = getProcessorList(conf);
    if (processors != null) {
      Collections.reverse(processors);
```

```
      for (ApplicationMasterServiceProcessor p : processors) {
        // Ensure only single instance of PlacementProcessor is included
        if (p instanceof AbstractPlacementProcessor) {
          LOG.warn("Found PlacementProcessor=" + p.getClass().getCanonicalName()
              + " defined in "
              + YarnConfiguration.RM_APPLICATION_MASTER_SERVICE_PROCESSORS
              + ", however PlacementProcessor handler should be configured "
              + "by using " + YarnConfiguration.RM_PLACEMENT_CONSTRAINTS_HANDLER
              + ", this processor will be ignored.");
          continue;
        }
        this.amsProcessingChain.addProcessor(p);
      }
    }
  }
```

1.3.5.3.14 客户端[可以提交任务、停止任务等等]发送 RPC 请求处理服务 ClientRMService.serviceInit()

```
  @Override
  protected void serviceInit(Configuration conf) throws Exception {
    // 默认 0.0.0.0:8032
    clientBindAddress = getBindAddress(conf);
    super.serviceInit(conf);
  }
```

1.3.5.3.15 负责启动 ApplicationMaster 服务 ApplicationMasterLauncher.serviceInit()

```
@Override
    protected void serviceInit(Configuration conf) throws Exception {
        // 线程个数 默认 50
        int threadCount = conf.getInt(
                YarnConfiguration.RM_AMLAUNCHER_THREAD_COUNT,
                YarnConfiguration.DEFAULT_RM_AMLAUNCHER_THREAD_COUNT);
        // 自定义线程池
        ThreadFactory tf = new ThreadFactoryBuilder()
                .setNameFormat("ApplicationMasterLauncher #%d")
                .build();
        launcherPool = new ThreadPoolExecutor(threadCount, threadCount, 1,
                TimeUnit.HOURS, new LinkedBlockingQueue<Runnable>());
        launcherPool.setThreadFactory(tf);

        // 创建配置对象 YarnConfiguration
        Configuration newConf = new YarnConfiguration(conf);
        // 默认 10
        newConf.setInt(CommonConfigurationKeysPublic.
                        IPC_CLIENT_CONNECT_MAX_RETRIES_ON_SOCKET_TIMEOUTS_KEY,
                conf.getInt(YarnConfiguration.RM_NODEMANAGER_CONNECT_RETRIES,
                        YarnConfiguration.DEFAULT_RM_NODEMANAGER_CONNECT_RETRIES));
        // 设置配置对象
        setConfig(newConf);
```

```
        super.serviceInit(newConf);
    }
```

1.3.5.4 解析 ResourceManager WEB UI 地址

```
  // ResourceManager WEB UI 默认 0.0.0.0:8088
        webAppAddress = WebAppUtils.getWebAppBindURL(this.conf,
                YarnConfiguration.RM_BIND_HOST,
                WebAppUtils.getRMWebAppURLWithoutScheme(this.conf));
```

1.3.5.5 创建并添加 ResourceManager 历史服务器 RMApplicationHistoryWriter

```
/**
 * <p>
 * {@link ResourceManager} uses this class to write the information of
 * {@link RMApp}, {@link RMAppAttempt} and {@link RMContainer}. These APIs are
 * non-blocking, and just schedule a writing history event. An self-contained
 * dispatcher vector will handle the event in separate threads, and extract the
 * required fields that are going to be persisted. Then, the extracted
 * information will be persisted via the implementation of
 * {@link ApplicationHistoryStore}.
 * </p>
 */
@Private
@Unstable
public class RMApplicationHistoryWriter extends CompositeService {
      public RMApplicationHistoryWriter() {
    // 往下追
    super(RMApplicationHistoryWriter.class.getName());
  }
}
```

1.3.5.6 调用 ResourceManager 组合服务的所有子服务 serviceInit()

1.3.5.6.1 ResourceManager 异步事件分发器 AsyncDispatcher.serviceInit()

```
// 调用其父类 serviceInit()
/**
     * All initialization code needed by a service.
     * <p>
     * This method will only ever be called once during the lifecycle of
     * a specific service instance.
     * <p>
     * Implementations do not need to be synchronized as the logic
     * in {@link #init(Configuration)} prevents re-entrancy.
     * <p>
     * The base implementation checks to see if the subclass has created
     * a new configuration instance, and if so, updates the base class value
     *
     * @param conf configuration
     * @throws Exception on a failure -these will be caught,
```

```
     *                  possibly wrapped, and wil; trigger a service stop
     */
    protected void serviceInit(Configuration conf) throws Exception {
        if (conf != config) {
            LOG.debug("Config has been overridden during init");
            setConfig(conf);
        }
    }
```

1.3.5.6.2 ResourceManager 超级管理者 AdminService.serviceInit()

```
@Override
    public void serviceInit(Configuration conf) throws Exception {
        // 当 ResourceManager 开启 HA 模式 则为 true
        autoFailoverEnabled =
                rm.getRMContext().isHAEnabled()
                        && HAUtil.isAutomaticFailoverEnabled(conf);

        // 默认 0.0.0.0:8033
        masterServiceBindAddress = conf.getSocketAddr(
                YarnConfiguration.RM_BIND_HOST,
                YarnConfiguration.RM_ADMIN_ADDRESS,
                YarnConfiguration.DEFAULT_RM_ADMIN_ADDRESS,
                YarnConfiguration.DEFAULT_RM_ADMIN_PORT);
        daemonUser = UserGroupInformation.getCurrentUser();
        authorizer = YarnAuthorizationProvider.getInstance(conf);
        authorizer.setAdmins(getAdminAclList(conf), daemonUser);
        rmId = conf.get(YarnConfiguration.RM_HA_ID);

        isCentralizedNodeLabelConfiguration =
                YarnConfiguration.isCentralizedNodeLabelConfiguration(conf);

        super.serviceInit(conf);
    }
```

1.3.5.6.3 ResourceManager HA选举服务 ActiveStandbyElectorBasedElectorService.serviceInit()

```
// ResourceManager HA 选举服务跟 HDFS 的 NameNode 选举类似 故不再介绍
@Override
  protected void serviceInit(Configuration conf)
      throws Exception {
    conf = conf instanceof YarnConfiguration
        ? conf
        : new YarnConfiguration(conf);

    // 获取 ZK 地址
    String zkQuorum = conf.get(YarnConfiguration.RM_ZK_ADDRESS);
    if (zkQuorum == null) {
      throw new YarnRuntimeException("Embedded automatic failover " +
          "is enabled, but " + YarnConfiguration.RM_ZK_ADDRESS +
          " is not set");
```

```
    }

    // 获取当前 ResourceManager 节点 的 rmId 比如 rm1、rm2
    String rmId = HAUtil.getRMHAId(conf);
    // 获取集群 ID
    String clusterId = YarnConfiguration.getClusterId(conf);
    localActiveNodeInfo = createActiveNodeInfo(clusterId, rmId);

    // HA 选举 ZK 路径
    String zkBasePath = conf.get(YarnConfiguration.AUTO_FAILOVER_ZK_BASE_PATH,
        YarnConfiguration.DEFAULT_AUTO_FAILOVER_ZK_BASE_PATH);
    String electionZNode = zkBasePath + "/" + clusterId;

    zkSessionTimeout = conf.getLong(YarnConfiguration.RM_ZK_TIMEOUT_MS,
        YarnConfiguration.DEFAULT_RM_ZK_TIMEOUT_MS);

    List<ACL> zkAcls = ZKCuratorManager.getZKAcls(conf);
    List<ZKUtil.ZKAuthInfo> zkAuths = ZKCuratorManager.getZKAuths(conf);

    int maxRetryNum =
        conf.getInt(YarnConfiguration.RM_HA_FC_ELECTOR_ZK_RETRIES_KEY, conf
          .getInt(CommonConfigurationKeys.HA_FC_ELECTOR_ZK_OP_RETRIES_KEY,
            CommonConfigurationKeys.HA_FC_ELECTOR_ZK_OP_RETRIES_DEFAULT));
    // 创建 HA 选举 ActiveStandbyElector
    elector = new ActiveStandbyElector(zkQuorum, (int) zkSessionTimeout,
        electionZNode, zkAcls, zkAuths, this, maxRetryNum, false);

    elector.ensureParentZNode();
    if (!isParentZnodeSafe(clusterId)) {
      notifyFatalError(String.format("invalid data in znode, %s, " +
          "which may require the state store to be reformatted",
          electionZNode));
    }

    super.serviceInit(conf);
  }
```

1.3.5.6.4 ResourceManager 历史服务器 RMApplicationHistoryWriter.serviceInit()

```
@Override
    protected synchronized void serviceInit(Configuration conf) throws Exception {
        // 是否开启历史服务 默认为 false
        historyServiceEnabled =
                conf.getBoolean(YarnConfiguration.APPLICATION_HISTORY_ENABLED,
                        YarnConfiguration.DEFAULT_APPLICATION_HISTORY_ENABLED);
        if (conf.get(YarnConfiguration.APPLICATION_HISTORY_STORE) == null ||
                conf.get(YarnConfiguration.APPLICATION_HISTORY_STORE).length() == 0
||
                conf.get(YarnConfiguration.APPLICATION_HISTORY_STORE).equals(
                        NullApplicationHistoryStore.class.getName())) {
            historyServiceEnabled = false;
        }
```

```
        // Only create the services when the history service is enabled and not
        // using the null store, preventing wasting the system resources.
        // 如果开启了历史服务功能
        if (historyServiceEnabled) {
            writer = createApplicationHistoryStore(conf);
            addIfService(writer);

            dispatcher = createDispatcher(conf);
            dispatcher.register(WritingHistoryEventType.class,
                    new ForwardingEventHandler());
            addIfService(dispatcher);
        }
        super.serviceInit(conf);
    }
```

1.3.5.7 哪些事件处理器往 ResourceManager 的 AsyncDispatcher 注册事件类型

```
 /** 来自于 RMActiveServices.serviceInit()
 * 哪些事件处理器往 ResourceManager 的 AsyncDispatcher 注册事件类型
 * 1 NodesListManagerEventType -> NodesListManager
 * 2 SchedulerEventType        -> EventDispatcher
 * 3 RMAppEventType            -> ApplicationEventDispatcher
 * 4 RMAppAttemptEventType     -> ApplicationAttemptEventDispatcher
 * 5 RMNodeEventType           -> NodeEventDispatcher
 * 6 RMAppManagerEventType     -> RMAppManager
 * 7 AMLauncherEventType       -> ApplicationMasterLauncher
*/
```

1.4 启动 ResourceManager

```
// 直接调用 ResourceManager.serviceStart()
@Override
    protected void serviceStart() throws Exception {
        // 如果 ResourceManager 开启 HA 则先进入 Standby 状态
        if (this.rmContext.isHAEnabled()) {
            // ResourceManager 一启动先进入 Active 状态 故往下是关闭 RMActiveService 所有
的子服务
            // 一旦 ResourceManager 选举成功 则 transitionToActive() 重新启动
RMActiveService 所有的子服务
            // 因为 ResourceManager HA 选举跟 HDFS 的 NameNode HA 选举相同
            // 故不再介绍(默认启动了 RMActiveService 服务)
            /** 只有 ResourceManager 被选举成 Active 时  也即调用 transitionToActive() -
>
             * 来自于 RMActiveServices.serviceStart() 并执行如下服务的 serviceStart()
             * 1 RMSecretManagerService            (安全服务)
             * 2 ContainerAllocationExpirer        (容器过期服务)
             * 3 AMLivelinessMonitor               (ApplicationMaster 存活监控服务)
             * 4 AMLivelinessMonitor               (ApplicationMaster 已完成监控服务)
             * 5 RMAppLifetimeMonitor              (ApplicationMaster 生命周期监控服务)
             * 6 RMNodeLabelsManager               (ResourceManager 节点标签管理服务)
```

```
             * 7 MemoryPlacementConstraintManager  (基于内存的安置约束管理器服务)
             * 8 NodesListManager                  (NodeManager 管理器服务)
             * 9 CapacityScheduler                 (资源调度器服务)
             * 10 EventDispatcher                  (资源调度事件分发器)
             * 11 NMLivelinessMonitor              (NodeManager 存活监控服务)
             * 12 ResourceTrackerService           (NodeManager 上报资源处理服务)
             * 13 ApplicationMasterService         (管理 ApplicationMaster 服务)
             * 14 ClientRMService                  (客户端[可以提交任务、停止任务等等]发送
RPC 请求处理服务)
             * 15 ApplicationMasterLauncher        (负责启动 ApplicationMaster 服务)
             */
            transitionToStandby(false);
        }

        // 启动 RMWebApp 服务
        startWepApp();

        if (getConfig().getBoolean(YarnConfiguration.IS_MINI_YARN_CLUSTER,
                false)) {
            int port = webApp.port();
            WebAppUtils.setRMWebAppPort(conf, port);
        }

        /** 来自于 ResourceManager.serviceStart() 并执行如下服务的 serviceStart()
         * 1 AsyncDispatcher                    (异步中央事件分发器服务)
         * 2 AdminService                       (ResourceManager 管理者服务)
         * 3 ActiveStandbyElectorBasedElectorService (ResourceManager HA 选举服务)
         * 4 RMApplicationHistoryWriter              (ResourceManager 历史服务器)
         * 5 CombinedSystemMetricsPublisher          (监控发布服务)
         */
        super.serviceStart();

        // Non HA case, start after RM services are started.
        // 如果 ResourceManager 不是 HA 模式 直接进入 Active 状态
        if (!this.rmContext.isHAEnabled()) {
            transitionToActive();
        }
    }
```

1.4.1 启动 ResourceManager 组合服务的所有子服务 serviceStart()

1.4.1.1 ResourceManager 异步事件分发器 AsyncDispatcher.serviceStart()

```
@Override
    protected void serviceStart() throws Exception {
        //start all the components
        // 针对 ResourceManager 的 AsyncDispatcher 服务来说 啥也不干
        super.serviceStart();
        // 创建 EventHandler 线程
        eventHandlingThread = new Thread(createThread());
        eventHandlingThread.setName(dispatcherThreadName);
        eventHandlingThread.start();
    }
```

```
Runnable createThread() {
        return new Runnable() {
            @Override
            public void run() {
                while (!stopped && !Thread.currentThread().isInterrupted()) {
                    drained = eventQueue.isEmpty();
                    // blockNewEvents is only set when dispatcher is draining to
stop,
                    // adding this check is to avoid the overhead of acquiring the
lock
                    // and calling notify every time in the normal run of the loop.
                    if (blockNewEvents) {
                        synchronized (waitForDrained) {
                            if (drained) {
                                waitForDrained.notify();
                            }
                        }
                    }
                    Event event;
                    try {
                        // 阻塞拉取 Event
                        event = eventQueue.take();
                    } catch (InterruptedException ie) {
                        if (!stopped) {
                            LOG.warn("AsyncDispatcher thread interrupted", ie);
                        }
                        return;
                    }
                    // 分发 Event 给对应的处理器
                    /**
                     * 针对 ResourceManager 的 AsyncDispatcher 而言
                     * AsyncDispatcher 核心注册了如下 Event 与 EventHandler 关系表
                     * 0 RMFatalEventType          -> RMFatalEventDispatcher
                     * 1 NodesListManagerEventType -> NodesListManager
                     * 2 SchedulerEventType        -> EventDispatcher
                     * 3 RMAppEventType            -> ApplicationEventDispatcher
                     * 4 RMAppAttemptEventType     ->
ApplicationAttemptEventDispatcher
                     * 5 RMNodeEventType           -> NodeEventDispatcher
                     * 6 RMAppManagerEventType     -> RMAppManager
```

```
                    * 7 AMLauncherEventType       -> ApplicationMasterLauncher
                    */
                   dispatch(event);
               }
           }
       };
   }
```

1.4.1.2 ResourceManager 管理者服务 AdminService.serviceStart()

```
@Override
    protected void serviceStart() throws Exception {
        // 启动服务
        startServer();
        super.serviceStart();
    }
```

```
protected void startServer() throws Exception {
        // 获取配置对象
        Configuration conf = getConfig();
        // 反射创建 HadoopYarnProtoRPC(调用其无参构造) RPC 服务(RPC 根据 HDFS 的 RPC 一样)
        YarnRPC rpc = YarnRPC.create(conf);
        // 创建 RPC 服务 默认地址 0.0.0.0:8033 通讯协议为
ResourceManagerAdministrationProtocol
        this.server = (Server) rpc.getServer(
                ResourceManagerAdministrationProtocol.class,
                this,
                // 默认 0.0.0.0:8033
                masterServiceBindAddress,
                conf, null,
                conf.getInt(YarnConfiguration.RM_ADMIN_CLIENT_THREAD_COUNT,
                        YarnConfiguration.DEFAULT_RM_ADMIN_CLIENT_THREAD_COUNT));

        // Enable service authorization?
        // 是否开启服务认证 默认不开启
        if (conf.getBoolean(
                CommonConfigurationKeysPublic.HADOOP_SECURITY_AUTHORIZATION,
                false)) {
            refreshServiceAcls(
                    getConfiguration(conf,
                            YarnConfiguration.HADOOP_POLICY_CONFIGURATION_FILE),
                    RMPolicyProvider.getInstance());
        }

        // ResourceManager 开启 HA 情况下
        if (rm.getRMContext().isHAEnabled()) {
            // 添加 HA 协议接口服务
            RPC.setProtocolEngine(conf, HAServiceProtocolPB.class,
                    ProtobufRpcEngine.class);

            HAServiceProtocolServerSideTranslatorPB haServiceProtocolXlator =
```

```
                    new HAServiceProtocolServerSideTranslatorPB(this);
            BlockingService haPbService =
                    HAServiceProtocolProtos.HAServiceProtocolService
                            .newReflectiveBlockingService(haServiceProtocolXlator);
            server.addProtocol(RPC.RpcKind.RPC_PROTOCOL_BUFFER,
                    HAServiceProtocol.class, haPbService);
        }

        // 启动 RPC 服务
        this.server.start();

        // 0.0.0.0:8033
        conf.updateConnectAddr(YarnConfiguration.RM_BIND_HOST,
                YarnConfiguration.RM_ADMIN_ADDRESS,
                YarnConfiguration.DEFAULT_RM_ADMIN_ADDRESS,
                server.getListenerAddress());
    }
```

1.4.1.2.1 总结 AdminService.serviceStart()

```
// AdminService 作为 ResourceManager 超级管理者服务，主要作用是管理 ResourceManager；其内部启动了一个绑定 0.0.0.0:8033 的 RPC 服务，实现了两个通讯协议接口
// 1. ResourceManagerAdministrationProtocol
// 2. HAServiceProtocol
```

1.4.1.3 ResourceManager HA 选举服务 ActiveStandbyElectorBasedElectorService.serviceStart()

```
    @Override
    protected void serviceStart() throws Exception {
        // 当前 ResourceManager 节点加入 HA 选举
        // 因为 ResourceManager HA 选举跟 HDFS 的 NameNode 选举类似 故不再介绍
        // 当前 ResourceManager HA 选举成为 Active 调用其 transitionToActive()
        // 否则调用其 transitionToStandby()
        elector.joinElection(localActiveNodeInfo);
        super.serviceStart();
    }
```

1.4.1.4 ResourceManager 历史服务器 RMApplicationHistoryWriter.serviceStart()

```
// 调用其父类 serviceStart() 因为 RMApplicationHistoryWriter 是一个组合服务，故还会调用其所有子服务的 serviceStart()，具体查看其 serviceInit() 添加了哪些服务
```

1.4.2 ResourceManager 服务 HA 成为 Active 时启动其 RMActiveService 服务的所有子服务 serviceStart()

1.4.2.1 安全服务 RMSecretManagerService.serviceStart()

```
@Override
    public void serviceStart() throws Exception {
        amRmTokenSecretManager.start();
        containerTokenSecretManager.start();
        nmTokenSecretManager.start();

        try {
            rmDTSecretManager.startThreads();
        } catch (IOException ie) {
            throw new YarnRuntimeException("Failed to start secret manager threads",
ie);
        }
        super.serviceStart();
    }
```

1.4.2.2 容器过期监控服务ContainerAllocationExpirer.serviceStart()

```
// 调用其服务 AbstractLivelinessMonitor.serviceStart()
 @Override
  protected void serviceStart() throws Exception {
    assert !stopped : "starting when already stopped";
    // 重置 Timer
    resetTimer();
    // 创建并启动 PingChecker 线程
    checkerThread = new Thread(new PingChecker());
    checkerThread.setName("Ping Checker");
    checkerThread.start();
    super.serviceStart();
  }
```

1.4.2.3 ApplicationMaster 存活监控服务 AMLivelinessMonitor.serviceStart()

```
// 调用其服务 AbstractLivelinessMonitor.serviceStart()
 @Override
  protected void serviceStart() throws Exception {
    assert !stopped : "starting when already stopped";
    // 重置 Timer
    resetTimer();
    // 创建并启动 PingChecker 线程
    checkerThread = new Thread(new PingChecker());
    checkerThread.setName("Ping Checker");
    checkerThread.start();
    super.serviceStart();
  }
```

1.4.2.4 ApplicationMaster 已完成监控服务 AMLivelinessMonitor.serviceStart()

```
// 调用其服务 AbstractLivelinessMonitor.serviceStart()
 @Override
  protected void serviceStart() throws Exception {
    assert !stopped : "starting when already stopped";
    // 重置 Timer
    resetTimer();
    // 创建并启动 PingChecker 线程
    checkerThread = new Thread(new PingChecker());
    checkerThread.setName("Ping Checker");
    checkerThread.start();
    super.serviceStart();
  }
```

1.4.2.5 ApplicationMaster 生命周期监控服务 RMAppLifetimeMonitor.serviceStart()

```
// 调用其服务 AbstractLivelinessMonitor.serviceStart()
 @Override
  protected void serviceStart() throws Exception {
    assert !stopped : "starting when already stopped";
    // 重置 Timer
    resetTimer();
    // 创建并启动 PingChecker 线程
    checkerThread = new Thread(new PingChecker());
    checkerThread.setName("Ping Checker");
    checkerThread.start();
    super.serviceStart();
  }
```

1.4.2.6 ResourceManager 节点标签管理服务 RMNodeLabelsManager.serviceStart()

```
// 调用其父类 CommonNodeLabelsManager.serviceStart()
@Override
  protected void serviceStart() throws Exception {
    // 判断是否开启节点标签功能 默认不开启
    if (nodeLabelsEnabled) {
      setInitNodeLabelStoreInProgress(true);
      initNodeLabelStore(getConfig());
      setInitNodeLabelStoreInProgress(false);
    }

    // init dispatcher only when service start, because recover will happen in
    // service init, we don't want to trigger any event handling at that time.
    // 初始化 AsyncDispatcher
    initDispatcher(getConfig());

    if (null != dispatcher) {
      // 注册 Event 与 EventHandler 关系表
      dispatcher.register(NodeLabelsStoreEventType.class,
          new ForwardingEventHandler());
```

```
    }

    // 启动 AsyncDispatcher
    startDispatcher();
  }
```

1.4.2.7 基于内存的安置约束管理器服务 MemoryPlacementConstraintManager.serviceStart()

```
// 调用其父类 AbstractService.serviceStart() 啥也不干
```

1.4.2.8 NodeManager 管理器服务 NodesListManager.serviceStart()

```
// 调用其父类 CompositeService.serviceStart() 启动其子服务
// NodesListManager 服务添加了哪些服务查看 serviceInit()
// NodesListManager 存储 NodeManager 注册信息并定时查看 NodeManager 是否上下线
```

1.4.2.9 资源调度器服务 CapacityScheduler.serviceStart()

```
    @Override
    public void serviceStart() throws Exception {
        // 启动定时调度线程 (里面启动很多服务组件 暂时不看 后面提交任务具体看)
        startSchedulerThreads();
        super.serviceStart();
    }
```

1.4.2.10 资源调度事件分发器 EventDispatcher.serviceStart()

```
  @Override
  protected void serviceStart() throws Exception {
    // 启动 EventProcessor 线程
    this.eventProcessor.start();
    super.serviceStart();
  }
```

1.4.2.11 NodeManager 存活监控服务 NMLivelinessMonitor.serviceStart()

```
// 调用其服务 AbstractLivelinessMonitor.serviceStart()
 @Override
  protected void serviceStart() throws Exception {
    assert !stopped : "starting when already stopped";
    // 重置 Timer
    resetTimer();
    // 创建并启动 PingChecker 线程
    checkerThread = new Thread(new PingChecker());
    checkerThread.setName("Ping Checker");
    checkerThread.start();
    super.serviceStart();
  }
```

1.4.2.12 NodeManager 注册上报资源处理服务 ResourceTrackerService.serviceStart()

```
@Override
    protected void serviceStart() throws Exception {
        super.serviceStart();
        // ResourceTrackerServer authenticates NodeManager via Kerberos if
        // security is enabled, so no secretManager.
        // 获取配置对象
        Configuration conf = getConfig();
        // 反射创建 HadoopYarnProtoRPC 对象
        YarnRPC rpc = YarnRPC.create(conf);
        // 创建 RPC Server 默认绑定地址 0.0.0.0:8031
        // 通讯协议接口为 ResourceTracker
        this.server = rpc.getServer(
                ResourceTracker.class, this, resourceTrackerAddress, conf, null,
                // 默认 50 Handler 线程

 conf.getInt(YarnConfiguration.RM_RESOURCE_TRACKER_CLIENT_THREAD_COUNT,

 YarnConfiguration.DEFAULT_RM_RESOURCE_TRACKER_CLIENT_THREAD_COUNT));

        // Enable service authorization?
        // 是否启动服务的认证 默认不开启该功能
        if (conf.getBoolean(
                CommonConfigurationKeysPublic.HADOOP_SECURITY_AUTHORIZATION,
                false)) {
            InputStream inputStream =
                    this.rmContext.getConfigurationProvider()
                            .getConfigurationInputStream(conf,

 YarnConfiguration.HADOOP_POLICY_CONFIGURATION_FILE);
            if (inputStream != null) {
                conf.addResource(inputStream);
            }
            refreshServiceAcls(conf, RMPolicyProvider.getInstance());
        }

        // 启动 RPC 服务
        this.server.start();
        conf.updateConnectAddr(YarnConfiguration.RM_BIND_HOST,
                YarnConfiguration.RM_RESOURCE_TRACKER_ADDRESS,
                YarnConfiguration.DEFAULT_RM_RESOURCE_TRACKER_ADDRESS,
                server.getListenerAddress());
    }
```

1.4.2.13 管理 ApplicationMaster 服务 ApplicationMasterService.serviceStart()

```
@Override
    protected void serviceStart() throws Exception {
        // 获取配置对象
        Configuration conf = getConfig();
```

```
        // 反射创建 HadoopYarnProtoRPC 对象
        YarnRPC rpc = YarnRPC.create(conf);

        Configuration serverConf = conf;
        // If the auth is not-simple, enforce it to be token-based.
        serverConf = new Configuration(conf);
        serverConf.set(
                CommonConfigurationKeysPublic.HADOOP_SECURITY_AUTHENTICATION,
                SaslRpcServer.AuthMethod.TOKEN.toString());

        // 创建 RPC Server 默认绑定地址 0.0.0.0:8030
        // 绑定通讯协议接口为 ApplicationMasterProtocol
        this.server = getServer(rpc, serverConf, masterServiceAddress,
                this.rmContext.getAMRMTokenSecretManager());
        // TODO more exceptions could be added later.
        this.server.addTerseExceptions(
                ApplicationMasterNotRegisteredException.class);

        // Enable service authorization?
        // 是否开启认证服务 默认不开启
        if (conf.getBoolean(
                CommonConfigurationKeysPublic.HADOOP_SECURITY_AUTHORIZATION,
                false)) {
            InputStream inputStream =
                    this.rmContext.getConfigurationProvider()
                            .getConfigurationInputStream(conf,

YarnConfiguration.HADOOP_POLICY_CONFIGURATION_FILE);
            if (inputStream != null) {
                conf.addResource(inputStream);
            }
            refreshServiceAcls(conf, RMPolicyProvider.getInstance());
        }

        // 启动 RPC Server
        this.server.start();
        // 默认 0.0.0.0:8030
        this.masterServiceAddress =
                conf.updateConnectAddr(YarnConfiguration.RM_BIND_HOST,
                        YarnConfiguration.RM_SCHEDULER_ADDRESS,
                        YarnConfiguration.DEFAULT_RM_SCHEDULER_ADDRESS,
                        server.getListenerAddress());
        this.timelineServiceV2Enabled = YarnConfiguration.
                timelineServiceV2Enabled(conf);
        super.serviceStart();
    }
```

1.4.2.14 客户端[可以提交任务、停止任务等等]发送 RPC 请求处理服务 ClientRMService.serviceStart()

```
@Override
    protected void serviceStart() throws Exception {
        // 获取配置对象
        Configuration conf = getConfig();
        // 反射创建 HadoopYarnProtoRPC
        YarnRPC rpc = YarnRPC.create(conf);
        // 创建 RPC Server 默认绑定 0.0.0.0:8032
        // 实现通讯协议接口为 ApplicationClientProtocol
        this.server =
                rpc.getServer(ApplicationClientProtocol.class, this,
                        clientBindAddress,
                        conf, this.rmDTSecretManager,
                        // 默认 Handler 个数 50
                        conf.getInt(YarnConfiguration.RM_CLIENT_THREAD_COUNT,
                                YarnConfiguration.DEFAULT_RM_CLIENT_THREAD_COUNT));

        this.server.addTerseExceptions(ApplicationNotFoundException.class,
                ApplicationAttemptNotFoundException.class,
                ContainerNotFoundException.class,
                YARNFeatureNotEnabledException.class);

        // Enable service authorization?
        // 是否开启认证服务 默认不开启
        if (conf.getBoolean(
                CommonConfigurationKeysPublic.HADOOP_SECURITY_AUTHORIZATION,
                false)) {
            InputStream inputStream =
                    this.rmContext.getConfigurationProvider()
                            .getConfigurationInputStream(conf,

 YarnConfiguration.HADOOP_POLICY_CONFIGURATION_FILE);
            if (inputStream != null) {
                conf.addResource(inputStream);
            }
            refreshServiceAcls(conf, RMPolicyProvider.getInstance());
        }

        this.filterAppsByUser = conf.getBoolean(
                YarnConfiguration.FILTER_ENTITY_LIST_BY_USER,
                YarnConfiguration.DEFAULT_DISPLAY_APPS_FOR_LOGGED_IN_USER);

        // 启动 RPC Server
        this.server.start();
        clientBindAddress = conf.updateConnectAddr(YarnConfiguration.RM_BIND_HOST,
                YarnConfiguration.RM_ADDRESS,
                YarnConfiguration.DEFAULT_RM_ADDRESS,
                server.getListenerAddress());
        this.timelineServiceV2Enabled = YarnConfiguration.
                timelineServiceV2Enabled(conf);
        super.serviceStart();
```

```
    }
```

1.4.2.15 负责启动 ApplicationMaster 服务 ApplicationMasterLauncher.serviceStart()

```
    @Override
    protected void serviceStart() throws Exception {
        // 启动 LauncherThread 线程阻塞拉取需要启动 ApplicationMaster
        launcherHandlingThread.start();
        super.serviceStart();
    }
```

```
private class LauncherThread extends Thread {

        public LauncherThread() {
            super("ApplicationMaster Launcher");
        }

        @Override
        public void run() {
            while (!this.isInterrupted()) {
                Runnable toLaunch;
                try {
                    // 拉取需要启动 ApplicationMaster
                    toLaunch = masterEvents.take();
                    launcherPool.execute(toLaunch);
                } catch (InterruptedException e) {
                    LOG.warn(this.getClass().getName() + " interrupted.
Returning.");
                    return;
                }
            }
        }
    }
```

二 NodeManager 启动源码分析

2.1 源码入口类

```
// org.apache.hadoop.yarn.server.nodemanager.NodeManager.main(args)
public class NodeManager extends CompositeService
        implements EventHandler<NodeManagerEvent>, NodeManagerMXBean {
     public static void main(String[] args) throws IOException {
                Thread.setDefaultUncaughtExceptionHandler(new
YarnUncaughtExceptionHandler());
                StringUtils.startupShutdownMessage(NodeManager.class, args, LOG);

            @SuppressWarnings("resource")
            // 创建 NodeManager 组合服务
            NodeManager nodeManager = new NodeManager();
            // 创建 YarnConfiguration
```

```
            Configuration conf = new YarnConfiguration();
            // 解析入参并传入 YarnConfiguration
            new GenericOptionsParser(conf, args);
            // 初始化并启动 NodeManager
            nodeManager.initAndStartNodeManager(conf, false);
    }
}
```

2.2 创建 NodeManager

```
    public NodeManager() {
        // 往下追
        super(NodeManager.class.getName());
    }
```

2.3 初始化 NodeManager

```
private void initAndStartNodeManager(Configuration conf, boolean hasToReboot) {
        try {
            // Failed to start if we're a Unix based system but we don't have bash.
            // Bash is necessary to launch containers under Unix-based systems.
            if (!Shell.WINDOWS) {
                if (!Shell.checkIsBashSupported()) {
                    String message =
                            "Failing NodeManager start since we're on a "
                                    + "Unix-based system but bash doesn't seem to be
available.";
                    LOG.error(message);
                    throw new YarnRuntimeException(message);
                }
            }

            // Remove the old hook if we are rebooting.
            if (hasToReboot && null != nodeManagerShutdownHook) {

 ShutdownHookManager.get().removeShutdownHook(nodeManagerShutdownHook);
            }

            // 注册钩子程序
            nodeManagerShutdownHook = new CompositeServiceShutdownHook(this);
            ShutdownHookManager.get().addShutdownHook(nodeManagerShutdownHook,
                    SHUTDOWN_HOOK_PRIORITY);

            // System exit should be called only when NodeManager is instantiated
from
            // main() funtion
            this.shouldExitOnShutdownEvent = true;

            // 初始化 NodeManager (直接调用 serviceInit() )
            this.init(conf);
```

```
            // 启动 NodeManager
            this.start();
        } catch (Throwable t) {
            LOG.error("Error starting NodeManager", t);
            System.exit(-1);
        }
    }
```

```
@Override
    protected void serviceInit(Configuration conf) throws Exception {
        UserGroupInformation.setConfiguration(conf);

        // 默认 true
        rmWorkPreservingRestartEnabled = conf.getBoolean(YarnConfiguration
                        .RM_WORK_PRESERVING_RECOVERY_ENABLED,
                YarnConfiguration.DEFAULT_RM_WORK_PRESERVING_RECOVERY_ENABLED);

        try {
            // 如果 NodeManager 开启了持久化恢复 则执行恢复操作(默认不开启该功能)
            initAndStartRecoveryStore(conf);
        } catch (IOException e) {
            String recoveryDirName = conf.get(YarnConfiguration.NM_RECOVERY_DIR);
            throw new
                    YarnRuntimeException("Unable to initialize recovery directory at
"
                    + recoveryDirName, e);
        }

        // 创建容器认证管理 NMContainerTokenSecretManager
        NMContainerTokenSecretManager containerTokenSecretManager =
                new NMContainerTokenSecretManager(conf, nmStore);

        // 创建 NodeManager Token Secret 管理者
        NMTokenSecretManagerInNM nmTokenSecretManager =
                new NMTokenSecretManagerInNM(nmStore);

        // 恢复 Token
        recoverTokens(nmTokenSecretManager, containerTokenSecretManager);

        // 创建 Application 认证管理器 ApplicationACLsManager
        this.aclsManager = new ApplicationACLsManager(conf);

        // 创建周期性检查当前 NodeManager 健康状态信息并保存到本地磁盘
LocalDirsHandlerService 服务
        this.dirsHandler = new LocalDirsHandlerService(metrics);

        // 默认 false
        boolean isDistSchedulingEnabled =
                conf.getBoolean(YarnConfiguration.DIST_SCHEDULING_ENABLED,
                        YarnConfiguration.DEFAULT_DIST_SCHEDULING_ENABLED);
```

```
        // 创建 NodeManager 上下文对象
        this.context = createNMContext(
                containerTokenSecretManager,
                nmTokenSecretManager,
                nmStore, isDistSchedulingEnabled, conf);

        // 创建资源插件管理 ResourcePluginManager
        ResourcePluginManager pluginManager = createResourcePluginManager();
        // 初始化 ResourcePluginManager (默认啥也不干)
        pluginManager.initialize(context);
        ((NMContext) context).setResourcePluginManager(pluginManager);

        // 反射创建容器执行器 默认
org.apache.hadoop.yarn.server.nodemanager.DefaultContainerExecutor
        ContainerExecutor exec = createContainerExecutor(conf);
        try {
            // 初始化容器执行器 (啥也不干)
            exec.init(context);
        } catch (IOException e) {
            throw new YarnRuntimeException("Failed to initialize container
executor", e);
        }
        // 创建并添加 DeletionService 服务
        DeletionService del = createDeletionService(exec);
        addService(del);

        // NodeManager level dispatcher
        // 创建 NodeManager 的异步分发器 AsyncDispatcher
        this.dispatcher = createNMDispatcher();

        // 创建并添加节点监控检查服务 NodeHealthCheckerService
        nodeHealthChecker =
                new NodeHealthCheckerService(
                        // 获取节点监控检查执行脚本 (默认没有配置节点监控检查执行脚本)
                        getNodeHealthScriptRunner(conf),
                        dirsHandler);
        addService(nodeHealthChecker);

        ((NMContext) context).setContainerExecutor(exec);
        ((NMContext) context).setDeletionService(del);

        // 创建节点标签提供商 默认为 null
        nodeLabelsProvider = createNodeLabelsProvider(conf);
        if (null == nodeLabelsProvider) {
            // 创建节点状态更新对象 NodeStatusUpdaterImpl
            nodeStatusUpdater =
                    createNodeStatusUpdater(context, dispatcher, nodeHealthChecker);
        } else {
            addIfService(nodeLabelsProvider);
            nodeStatusUpdater =
                    createNodeStatusUpdater(context, dispatcher, nodeHealthChecker,
```

```
                    nodeLabelsProvider);
        }

        // 创建并添加节点资源监控服务 NodeResourceMonitorImpl
        nodeResourceMonitor = createNodeResourceMonitor();
        addService(nodeResourceMonitor);
        ((NMContext) context).setNodeResourceMonitor(nodeResourceMonitor);

        // 创建并添加容器管理服务 ContainerManagerImpl (组合服务)
        containerManager =
                createContainerManager(context, exec, del, nodeStatusUpdater,
                        this.aclsManager, dirsHandler);
        addService(containerManager);
        ((NMContext) context).setContainerManager(containerManager);

        // 创建并添加 NM 日志聚合追踪服务 NMLogAggregationStatusTracker
        this.nmLogAggregationStatusTracker = createNMLogAggregationStatusTracker(
                context);
        addService(nmLogAggregationStatusTracker);
        ((NMContext) context).setNMLogAggregationStatusTracker(
                this.nmLogAggregationStatusTracker);

        // 创建并添加 NM Web 服务 NMWebApp
        WebServer webServer = createWebServer(context, containerManager
                .getContainersMonitor(), this.aclsManager, dirsHandler);
        addService(webServer);
        ((NMContext) context).setWebServer(webServer);

        ((NMContext) context).setQueueableContainerAllocator(
                new OpportunisticContainerAllocator(
                        context.getContainerTokenSecretManager()));

        // 往 NodeManager 的 AsyncDispatcher 注册 EventType 与 EventHandler 映射表
        dispatcher.register(ContainerManagerEventType.class, containerManager);
        dispatcher.register(NodeManagerEventType.class, this);
        addService(dispatcher);

        // 创建并添加 JVM 监控服务
        pauseMonitor = new JvmPauseMonitor();
        addService(pauseMonitor);
        metrics.getJvmMetrics().setPauseMonitor(pauseMonitor);

        DefaultMetricsSystem.initialize("NodeManager");

        // 默认不开启
        if (YarnConfiguration.timelineServiceV2Enabled(conf)) {
            this.nmCollectorService = createNMCollectorService(context);
            addService(nmCollectorService);
        }

        // StatusUpdater should be added last so that it get started last
        // so that we make sure everything is up before registering with RM.
```

```
        // 添加节点状态更新服务
        addService(nodeStatusUpdater);
        ((NMContext) context).setNodeStatusUpdater(nodeStatusUpdater);
        nmStore.setNodeStatusUpdater(nodeStatusUpdater);

        // Do secure login before calling init for added services.
        try {
            doSecureLogin();
        } catch (IOException e) {
            throw new YarnRuntimeException("Failed NodeManager login", e);
        }

        registerMXBean();

        /**
         * ContainerManagerImpl 容器管理服务是一个组合服务，里面添加了如下子服务
         * 1 ResourceLocalizationService     -> 本地资源服务
         * 2 ContainersLauncher              -> 容器执行器服务
         * 3 ContainerScheduler              -> 容器调度服务
         * 4 AuxServices                     -> Aux 服务
         * 5 ContainersMonitorImpl           -> 容器监控服务
         * 6 AsyncDispatcher                 -> 容器异步分发器服务
         * 7 LogAggregationService           -> yarn 日志聚合服务
         * 8 SharedCacheUploadService        -> 本地上传文件服务
         *
         * 容器管理服务 ContainerManagerImpl 的异步分发器 AsyncDispatcher 注册了
         * 哪些 EventType 与 EventHandler 映射关系
         * 1 ContainerEventType          -> ContainerEventDispatcher
         * 2 ApplicationEventType        -> ApplicationEventDispatcher
         * 3 LocalizationEventType       -> LocalizationEventHandlerWrapper
         * 4 AuxServicesEventType        -> AuxServices
         * 5 ContainersMonitorEventType  -> ContainersMonitorImpl
         * 6 ContainersLauncherEventType -> ContainersLauncher
         * 7 ContainerSchedulerEventType -> ContainerScheduler
         * 8 LogHandlerEventType         -> LogAggregationService
         * 9 SharedCacheUploadEventType  -> SharedCacheUploadService
         */

        /**
         * NodeManager 组合服务添加了哪些子服务 (调用其 serviceInit())
         * 1 DeletionService                   -> 删除容器服务
         * 2 NodeHealthCheckerService          -> 节点监控检查服务
         * 3 NodeResourceMonitorImpl           -> 节点资源监控服务
         * 4 ContainerManagerImpl              -> 容器管理服务
         * 5 NMLogAggregationStatusTracker     -> NM 日志聚合追踪服务
         * 6 NMWebApp                          -> NM Web 服务
         * 7 AsyncDispatcher                   -> NM 异步分发器服务
         * 8 JvmPauseMonitor                   -> NM JVM 监控服务
         * 9 NodeStatusUpdaterImpl             -> NM 节点状态更新服务
         *
         * NodeManager 组合服务的异步分发器 AsyncDispatcher 注册了
         * 哪些 EventType 与 EventHandler 映射关系
```

```
         * 1 ContainerManagerEventType   -> ContainerManagerImpl
         * 2 NodeManagerEventType        -> NodeManager
         */
        super.serviceInit(conf);
        // TODO add local dirs to del
    }
```

2.3.1 创建并添加 NodeManager 的子服务

2.3.1.1 创建 NodeManager 上下文对象 NMContext

```
protected NMContext createNMContext(
            NMContainerTokenSecretManager containerTokenSecretManager,
            NMTokenSecretManagerInNM nmTokenSecretManager,
            NMStateStoreService stateStore, boolean isDistSchedulerEnabled,
            Configuration conf) {
        // 容器状态监听器 默认为空
        List<ContainerStateTransitionListener> listeners =
                conf.getInstances(
                        YarnConfiguration.NM_CONTAINER_STATE_TRANSITION_LISTENERS,
                        ContainerStateTransitionListener.class);

        // 创建 NM 上下文对象 NMContext
        NMContext nmContext = new NMContext(containerTokenSecretManager,
                nmTokenSecretManager, dirsHandler, aclsManager, stateStore,
                isDistSchedulerEnabled, conf);

        nmContext.setNodeManagerMetrics(metrics);

        DefaultContainerStateListener defaultListener =
                new DefaultContainerStateListener();
        nmContext.setContainerStateTransitionListener(defaultListener);
        // 啥也不干
        defaultListener.init(nmContext);

        for (ContainerStateTransitionListener listener : listeners) {
            listener.init(nmContext);
            defaultListener.addListener(listener);
        }

        return nmContext;
    }
```

2.3.1.2 创建资源插件管理 ResourcePluginManager

```
    @VisibleForTesting
    protected ResourcePluginManager createResourcePluginManager() {
        // 创建 ResourcePluginManager
        return new ResourcePluginManager();
    }
```

```
/**
 * Manages {@link ResourcePlugin} configured on this NodeManager.
 */
public class ResourcePluginManager {
}
```

2.3.1.3 反射创建容器执行器 DefaultContainerExecutor

```
/**
 * The {@code DefaultContainerExecuter} class offers generic container
 * execution services. Process execution is handled in a platform-independent
 * way via {@link ProcessBuilder}.
 */
public class DefaultContainerExecutor extends ContainerExecutor {
}
```

2.3.1.4 创建并添加 DeletionService 服务

```
    protected DeletionService createDeletionService(ContainerExecutor exec) {
        // 创建 DeletionService
        return new DeletionService(exec, nmStore);
    }
```

```
public class DeletionService extends AbstractService {
}
```

2.3.1.5 创建并添加 NodeManager 的异步分发器 AsyncDispatcher 服务

```
    /**
     * Unit test friendly.
     */
    protected AsyncDispatcher createNMDispatcher() {
        // 创建 AsyncDispatcher
        return new AsyncDispatcher("NM Event dispatcher");
    }
```

```
/**
 * Dispatches {@link Event}s in a separate thread. Currently only single thread
 * does that. Potentially there could be multiple channels for each event type
 * class and a thread pool can be used to dispatch the events.
 */
@SuppressWarnings("rawtypes")
@Public
@Evolving
public class AsyncDispatcher extends AbstractService implements Dispatcher {
    public AsyncDispatcher(BlockingQueue<Event> eventQueue) {
        // 往下追
        super("Dispatcher");
        // 链表阻塞队列 LinkedBlockingQueue
```

```
        this.eventQueue = eventQueue;
        // 枚举事件类型 (Event) 与其事件处理函数 (EventHandler) 关系注册表
        this.eventDispatchers = new HashMap<Class<? extends Enum>, EventHandler>();
    }
}
```

2.3.1.6 创建并添加节点监控检查服务 NodeHealthCheckerService

```
/**
 * The class which provides functionality of checking the health of the node and
 * reporting back to the service for which the health checker has been asked to
 * report.
 */
public class NodeHealthCheckerService extends CompositeService {
    public NodeHealthCheckerService(NodeHealthScriptRunner scriptRunner,
                                    LocalDirsHandlerService dirHandlerService) {
        super(NodeHealthCheckerService.class.getName());
        // 默认 null
        nodeHealthScriptRunner = scriptRunner;
        dirsHandler = dirHandlerService;
        nodeHealthException = null;
        nodeHealthExceptionReportTime = 0;
    }
}
```

2.3.1.7 创建并添加节点资源监控服务 NodeResourceMonitorImpl

```
    protected NodeResourceMonitor createNodeResourceMonitor() {
        // 创建 NodeResourceMonitorImpl
        return new NodeResourceMonitorImpl(context);
    }
```

```
/**
 * Implementation of the node resource monitor. It periodically tracks the
 * resource utilization of the node and reports it to the NM.
 */
public class NodeResourceMonitorImpl extends AbstractService implements
        NodeResourceMonitor {
 /**
     * Initialize the node resource monitor.
     */
    public NodeResourceMonitorImpl(Context context) {
        super(NodeResourceMonitorImpl.class.getName());
        this.nmContext = context;
        // 创建 MonitoringThread 线程
        this.monitoringThread = new MonitoringThread();
    }
}
```

2.3.1.8 创建并添加容器管理服务 ContainerManagerImpl (组合服务)

```
    protected ContainerManagerImpl createContainerManager(Context context,
                                                           ContainerExecutor exec,
DeletionService del,
                                                           NodeStatusUpdater
nodeStatusUpdater, ApplicationACLsManager aclsManager,
                                                           LocalDirsHandlerService
dirsHandler) {
        // 创建 ContainerManagerImpl
        return new ContainerManagerImpl(context, exec, del, nodeStatusUpdater,
                metrics, dirsHandler);
    }
```

```
public class ContainerManagerImpl extends CompositeService implements
        ContainerManager {
    public ContainerManagerImpl(Context context, ContainerExecutor exec,
                                DeletionService deletionContext, NodeStatusUpdater
nodeStatusUpdater,
                                NodeManagerMetrics metrics, LocalDirsHandlerService
dirsHandler) {
        super(ContainerManagerImpl.class.getName());
        this.context = context;
        this.dirsHandler = dirsHandler;

        // ContainerManager level dispatcher.
        // 创建 AsyncDispatcher
        dispatcher = new AsyncDispatcher("NM ContainerManager dispatcher");
        this.deletionService = deletionContext;
        this.metrics = metrics;

        // 创建并添加本地资源服务 ResourceLocalizationService
        rsrcLocalizationSrvc =
                createResourceLocalizationService(exec, deletionContext, context,
                        metrics);
        addService(rsrcLocalizationSrvc);

        // 创建并添加容器执行器服务 ContainersLauncher
        containersLauncher = createContainersLauncher(context, exec);
        addService(containersLauncher);

        this.nodeStatusUpdater = nodeStatusUpdater;

        // 创建并添加容器调度服务 ContainerScheduler
        this.containerScheduler = createContainerScheduler(context);
        addService(containerScheduler);

        AuxiliaryLocalPathHandler auxiliaryLocalPathHandler =
                new AuxiliaryLocalPathHandlerImpl(dirsHandler);
        // Start configurable services
        // 创建并添加 AuxServices 服务
```

```
        auxiliaryServices = new AuxServices(auxiliaryLocalPathHandler,
                this.context, this.deletionService);
        auxiliaryServices.registerServiceListener(this);
        addService(auxiliaryServices);

        // initialize the metrics publisher if the timeline service v.2 is enabled
        // and the system publisher is enabled
        Configuration conf = context.getConf();
        if (YarnConfiguration.timelineServiceV2Enabled(conf)) {
            if (YarnConfiguration.systemMetricsPublisherEnabled(conf)) {
                LOG.info("YARN system metrics publishing service is enabled");
                nmMetricsPublisher = createNMTimelinePublisher(context);
                context.setNMTimelinePublisher(nmMetricsPublisher);
            }
            this.timelineServiceV2Enabled = true;
        }

        // 创建并添加容器监控服务 ContainersMonitorImpl
        this.containersMonitor = createContainersMonitor(exec);
        addService(this.containersMonitor);

        // 往容器管理者组合服务的 AsyncDispatcher 注册 EventType 与 EventHandler 映射表
        dispatcher.register(ContainerEventType.class,
                new ContainerEventDispatcher());
        dispatcher.register(ApplicationEventType.class,
                // 创建 ApplicationEventDispatcher
                createApplicationEventDispatcher());
        dispatcher.register(LocalizationEventType.class,
                new LocalizationEventHandlerWrapper(rsrcLocalizationSrvc,
                        nmMetricsPublisher));
        dispatcher.register(AuxServicesEventType.class, auxiliaryServices);
        dispatcher.register(ContainersMonitorEventType.class, containersMonitor);
        dispatcher.register(ContainersLauncherEventType.class, containersLauncher);
        dispatcher.register(ContainerSchedulerEventType.class, containerScheduler);

        // 添加 AsyncDispatcher 服务
        addService(dispatcher);

        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
        this.readLock = lock.readLock();
        this.writeLock = lock.writeLock();
    }
}
```

2.3.1.8.1 创建并添加异步分发器 AsyncDispatcher

```
/**
     * Set a name for this dispatcher thread.
     *
     * @param dispatcherName name of the dispatcher thread
     */
    public AsyncDispatcher(String dispatcherName) {
        // 往下追
        this();
        // 如果是 ResourceManager -> RM Event dispatcher
        // 如果是 NodeManager -> NM Event dispatcher
        // 如果是 ContainerManagerImpl -> NM ContainerManager dispatcher
        dispatcherThreadName = dispatcherName;
    }
```

2.3.1.8.2 创建并添加本地资源服务 ResourceLocalizationService

```
    protected ResourceLocalizationService createResourceLocalizationService(
            ContainerExecutor exec, DeletionService deletionContext,
            Context nmContext, NodeManagerMetrics nmMetrics) {
        // 创建 ResourceLocalizationService
        return new ResourceLocalizationService(this.dispatcher, exec,
                deletionContext, dirsHandler, nmContext, nmMetrics);
    }
```

```
public class ResourceLocalizationService extends CompositeService
        implements EventHandler<LocalizationEvent>, LocalizationProtocol {
    public ResourceLocalizationService(Dispatcher dispatcher,
                                       ContainerExecutor exec, DeletionService
delService,
                                       LocalDirsHandlerService dirsHandler, Context
context,
                                       NodeManagerMetrics metrics) {

        super(ResourceLocalizationService.class.getName());
        this.exec = exec;
        this.dispatcher = dispatcher;
        this.delService = delService;
        this.dirsHandler = dirsHandler;

        // 创建定时调度线程池
        this.cacheCleanup = new HadoopScheduledThreadPoolExecutor(1,
                new ThreadFactoryBuilder()
                        .setNameFormat("ResourceLocalizationService Cache Cleanup")
                        .build());
        this.stateStore = context.getNMStateStore();
        this.nmContext = context;
        this.metrics = metrics;
    }
}
```

2.3.1.8.3 创建并添加容器执行器服务 ContainersLauncher

```
protected AbstractContainersLauncher createContainersLauncher(
            Context ctxt, ContainerExecutor exec) {
        Class<? extends AbstractContainersLauncher> containersLauncherClass =
                ctxt.getConf()
                        // 默认
org.apache.hadoop.yarn.server.nodemanager.containermanager.launcher.ContainersLaunch
er
                        .getClass(YarnConfiguration.NM_CONTAINERS_LAUNCHER_CLASS,
                                ContainersLauncher.class,
AbstractContainersLauncher.class);
        AbstractContainersLauncher launcher;
        try {
            // 反射创建 ContainersLauncher 并调用其 setConf() 设置配置对象
            launcher = ReflectionUtils.newInstance(containersLauncherClass,
                    ctxt.getConf());
            // 初始化容器执行器
            launcher.init(ctxt, this.dispatcher, exec, dirsHandler, this);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return launcher;
    }
```

```
/**
 * The launcher for the containers. This service should be started only after
 * the {@link ResourceLocalizationService} is started as it depends on creation
 * of system directories on the local file-system.
 *
 */
public class ContainersLauncher extends AbstractService
    implements AbstractContainersLauncher {
  public ContainersLauncher() {
        super("containers-launcher");
  }
    @Override
  public void init(Context nmContext, Dispatcher nmDispatcher,
      ContainerExecutor containerExec, LocalDirsHandlerService nmDirsHandler,
      ContainerManagerImpl nmContainerManager) {
    this.exec = containerExec;
    this.context = nmContext;
    this.dispatcher = nmDispatcher;
    this.dirsHandler = nmDirsHandler;
    this.containerManager = nmContainerManager;
  }
}
```

2.3.1.8.4 创建并添加容器调度服务 ContainerScheduler

```
    @VisibleForTesting
    protected ContainerScheduler createContainerScheduler(Context cntxt) {
        // Currently, this dispatcher is shared by the ContainerManager,
        // all the containers, the container monitor and all the container.
        // The ContainerScheduler may use its own dispatcher.
        // 创建 ContainerScheduler
        return new ContainerScheduler(cntxt, dispatcher, metrics);
    }
```

```
/**
 * The ContainerScheduler manages a collection of runnable containers. It
 * ensures that a container is launched only if all its launch criteria are
 * met. It also ensures that OPPORTUNISTIC containers are killed to make
 * room for GUARANTEED containers.
 */
public class ContainerScheduler extends AbstractService implements
    EventHandler<ContainerSchedulerEvent> {
 /**
   * Instantiate a Container Scheduler.
   * @param context NodeManager Context.
   * @param dispatcher AsyncDispatcher.
   * @param metrics NodeManagerMetrics.
   */
  public ContainerScheduler(Context context, AsyncDispatcher dispatcher,
      NodeManagerMetrics metrics) {
    // 往下追
    this(context, dispatcher, metrics, context.getConf().getInt(
        YarnConfiguration.NM_OPPORTUNISTIC_CONTAINERS_MAX_QUEUE_LENGTH,
        YarnConfiguration.DEFAULT_NM_OPPORTUNISTIC_CONTAINERS_MAX_QUEUE_LENGTH));
  }
    @VisibleForTesting
  public ContainerScheduler(Context context, AsyncDispatcher dispatcher,
      NodeManagerMetrics metrics, int qLength) {
    super(ContainerScheduler.class.getName());
    this.context = context;
    this.dispatcher = dispatcher;
    this.metrics = metrics;
    // 默认 0
    this.maxOppQueueLength = Math.max(qLength, 0);
    this.utilizationTracker =
        new AllocationBasedResourceUtilizationTracker(this);
    this.opportunisticContainersStatus =
        OpportunisticContainersStatus.newInstance();
  }
}
```

2.3.1.8.5 创建并添加 AuxServices 服务

```
public class AuxServices extends AbstractService
    implements ServiceStateChangeListener, EventHandler<AuxServicesEvent> {
}
```

2.3.1.8.6 创建并添加容器监控服务 ContainersMonitorImpl

```
    protected ContainersMonitor createContainersMonitor(ContainerExecutor exec) {
        // 创建 ContainersMonitorImpl
        return new ContainersMonitorImpl(exec, dispatcher, this.context);
    }
```

```
/**
 * Monitors containers collecting resource usage and preempting the container
 * if it exceeds its limits.
 */
public class ContainersMonitorImpl extends AbstractService implements
    ContainersMonitor {
 public ContainersMonitorImpl(ContainerExecutor exec,
      AsyncDispatcher dispatcher, Context context) {
    super("containers-monitor");

    this.containerExecutor = exec;
    this.eventDispatcher = dispatcher;
    this.context = context;

    // 创建 ContainersMonitorImpl 线程
    this.monitoringThread = new MonitoringThread();

    this.containersUtilization = ResourceUtilization.newInstance(0, 0, 0.0f);
  }
}
```

2.3.1.9 创建并添加 NM 日志聚合追踪服务 NMLogAggregationStatusTracker

```
    private NMLogAggregationStatusTracker createNMLogAggregationStatusTracker(
            Context ctxt) {
        // 创建 NMLogAggregationStatusTracker
        return new NMLogAggregationStatusTracker(ctxt);
    }
```

```
/**
 * {@link NMLogAggregationStatusTracker} is used to cache log aggregation
 * status for finished applications. It will also delete the old cached
 * log aggregation status periodically.
 *
 */
public class NMLogAggregationStatusTracker extends CompositeService {
 public NMLogAggregationStatusTracker(Context context) {
```

```
    super(NMLogAggregationStatusTracker.class.getName());
    this.nmContext = context;
    Configuration conf = context.getConf();
    if (!conf.getBoolean(YarnConfiguration.LOG_AGGREGATION_ENABLED,
        YarnConfiguration.DEFAULT_LOG_AGGREGATION_ENABLED)) {
      disabled = true;
    }
    this.recoveryStatuses = new ConcurrentHashMap<>();
    ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    this.readLocker = lock.readLock();
    this.writeLocker = lock.writeLock();
    this.timer = new Timer();
    long configuredRollingInterval = conf.getLong(
        YarnConfiguration.LOG_AGGREGATION_STATUS_TIME_OUT_MS,
        YarnConfiguration.DEFAULT_LOG_AGGREGATION_STATUS_TIME_OUT_MS);
    if (configuredRollingInterval <= 0) {
      this.rollingInterval = YarnConfiguration
          .DEFAULT_LOG_AGGREGATION_STATUS_TIME_OUT_MS;
      LOG.warn("The configured log-aggregation-status.time-out.ms is "
          + configuredRollingInterval + " which should be larger than 0. "
          + "Using the default value:" + this.rollingInterval + " instead.");
    } else {
      this.rollingInterval = configuredRollingInterval;
    }
    LOG.info("the rolling interval seconds for the NodeManager Cached Log "
        + "aggregation status is " + (rollingInterval/1000));
  }
}
```

2.3.1.10 创建并添加 NM Web 服务 NMWebApp

```
    protected WebServer createWebServer(Context nmContext,
                                        ResourceView resourceView,
ApplicationACLsManager aclsManager,
                                        LocalDirsHandlerService dirsHandler) {
        // 创建 WebServer
        return new WebServer(nmContext, resourceView, aclsManager, dirsHandler);
    }
```

```
public class WebServer extends AbstractService {
   public WebServer(Context nmContext, ResourceView resourceView,
      ApplicationACLsManager aclsManager,
      LocalDirsHandlerService dirsHandler) {
    super(WebServer.class.getName());
    this.nmContext = nmContext;
    // 创建 NMWebApp
    this.nmWebApp = new NMWebApp(resourceView, aclsManager, dirsHandler);
  }
}
```

2.3.1.11 创建并添加 JVM 监控服务 JvmPauseMonitor

```
/**
 * Class which sets up a simple thread which runs in a loop sleeping
 * for a short interval of time. If the sleep takes significantly longer
 * than its target time, it implies that the JVM or host machine has
 * paused processing, which may cause other problems. If such a pause is
 * detected, the thread logs a message.
 */
@InterfaceAudience.Private
public class JvmPauseMonitor extends AbstractService {
   public JvmPauseMonitor() {
    super(JvmPauseMonitor.class.getName());
  }
}
```

2.3.1.12 创建并添加节点状态更新服务 NodeStatusUpdaterImpl

```
    protected NodeStatusUpdater createNodeStatusUpdater(Context context,
                                                        Dispatcher dispatcher,
NodeHealthCheckerService healthChecker) {
        // 创建 NodeStatusUpdaterImpl
        return new NodeStatusUpdaterImpl(context, dispatcher, healthChecker,
                metrics, nodeLabelsProvider);
    }
```

```
public class NodeStatusUpdaterImpl extends AbstractService implements
    NodeStatusUpdater {
    public NodeStatusUpdaterImpl(Context context, Dispatcher dispatcher,
      NodeHealthCheckerService healthChecker, NodeManagerMetrics metrics,
      NodeLabelsProvider nodeLabelsProvider) {
    super(NodeStatusUpdaterImpl.class.getName());
    this.healthChecker = healthChecker;
    this.context = context;
    this.dispatcher = dispatcher;
    this.nodeLabelsProvider = nodeLabelsProvider;
    this.metrics = metrics;
    this.recentlyStoppedContainers = new LinkedHashMap<ContainerId, Long>();
    this.pendingCompletedContainers =
        new HashMap<ContainerId, ContainerStatus>();
    this.logAggregationReportForAppsTempList =
        new ArrayList<LogAggregationReport>();
  }
}
```

2.3.2 启动 NodeManager 组合服务的所有子服务的 serviceInit()

2.3.2.1 删除容器服务 DeletionService.serviceInit()

```
@Override
  protected void serviceInit(Configuration conf) throws Exception {
    ThreadFactory tf = new ThreadFactoryBuilder()
        .setNameFormat("DeletionService #%d")
        .build();
    if (conf != null) {
      // 创建线程池 默认线程个数为 4
      sched = new HadoopScheduledThreadPoolExecutor(
          conf.getInt(YarnConfiguration.NM_DELETE_THREAD_COUNT,
              YarnConfiguration.DEFAULT_NM_DELETE_THREAD_COUNT), tf);
      debugDelay = conf.getInt(YarnConfiguration.DEBUG_NM_DELETE_DELAY_SEC, 0);
    } else {
      sched = new HadoopScheduledThreadPoolExecutor(
          YarnConfiguration.DEFAULT_NM_DELETE_THREAD_COUNT, tf);
    }
    sched.setExecuteExistingDelayedTasksAfterShutdownPolicy(false);
    sched.setKeepAliveTime(60L, SECONDS);
    if (stateStore.canRecover()) {
      recover(stateStore.loadDeletionServiceState());
    }
    super.serviceInit(conf);
  }
```

2.3.2.2 节点监控检查服务 NodeHealthCheckerService.serviceInit()

```
    @Override
    protected void serviceInit(Configuration conf) throws Exception {
        if (nodeHealthScriptRunner != null) {
            addService(nodeHealthScriptRunner);
        }
        // 添加 LocalDirsHandlerService 服务并调用其 serviceInit()
        addService(dirsHandler);
        super.serviceInit(conf);
    }
```

2.3.2.3 节点资源监控服务 NodeResourceMonitorImpl.serviceInit()

```
/**
     * Initialize the service with the proper parameters.
     */
    @Override
    protected void serviceInit(Configuration conf) throws Exception {
        // 默认监控间隔 3000ms
        this.monitoringInterval =
                conf.getLong(YarnConfiguration.NM_RESOURCE_MON_INTERVAL_MS,
                        YarnConfiguration.DEFAULT_NM_RESOURCE_MON_INTERVAL_MS);
```

```
        // 默认 ResourceCalculatorPlugin
        this.resourceCalculatorPlugin =
                ResourceCalculatorPlugin.getNodeResourceMonitorPlugin(conf);

        LOG.info(" Using ResourceCalculatorPlugin : "
                + this.resourceCalculatorPlugin);
    }
```

2.3.2.4 容器管理组合服务 ContainerManagerImpl.serviceInit() （重要服务）

```
@Override
    public void serviceInit(Configuration conf) throws Exception {

        // 创建 yarn 日志聚合服务 LogAggregationService
        logHandler =
                createLogHandler(conf, this.context, this.deletionService);
        addIfService(logHandler);
        dispatcher.register(LogHandlerEventType.class, logHandler);

        // add the shared cache upload service (it will do nothing if the shared
        // cache is disabled)
        // 创建本地上传文件服务 SharedCacheUploadService
        SharedCacheUploadService sharedCacheUploader =
                createSharedCacheUploaderService();
        addService(sharedCacheUploader);
        dispatcher.register(SharedCacheUploadEventType.class, sharedCacheUploader);

        // 默认不开启该功能
        createAMRMProxyService(conf);

        // 250ms + 5000ms + 1000ms
        waitForContainersOnShutdownMillis =
                conf.getLong(YarnConfiguration.NM_SLEEP_DELAY_BEFORE_SIGKILL_MS,
                        YarnConfiguration.DEFAULT_NM_SLEEP_DELAY_BEFORE_SIGKILL_MS)
+
                        conf.getLong(YarnConfiguration.NM_PROCESS_KILL_WAIT_MS,
                                YarnConfiguration.DEFAULT_NM_PROCESS_KILL_WAIT_MS) +
                        SHUTDOWN_CLEANUP_SLOP_MS;

        super.serviceInit(conf);
        recover();
    }
```

2.3.2.4.1 本地资源服务 ResourceLocalizationService.serviceInit()

```
@Override
    public void serviceInit(Configuration conf) throws Exception {
        this.validateConf(conf);
        // 创建本地资源追踪管理 LocalResourcesTrackerImpl
        this.publicRsrc = new LocalResourcesTrackerImpl(null, null, dispatcher,
                true, conf, stateStore, dirsHandler);
```

```
        this.recordFactory = RecordFactoryProvider.getRecordFactory(conf);

        try {
            // 创建本地文件上下文 FileContext
            lfs = getLocalFileContext(conf);
            lfs.setUMask(new FsPermission((short) FsPermission.DEFAULT_UMASK));

            if (!stateStore.canRecover() || stateStore.isNewlyCreated()) {
                cleanUpLocalDirs(lfs, delService);
                cleanupLogDirs(lfs, delService);
                initializeLocalDirs(lfs);
                initializeLogDirs(lfs);
            }
        } catch (Exception e) {
            throw new YarnRuntimeException(
                    "Failed to initialize LocalizationService", e);
        }

        // 反射创建本地磁盘校验器 默认 BasicDiskValidator
        diskValidator = DiskValidatorFactory.getInstance(
                conf.get(YarnConfiguration.DISK_VALIDATOR,
                        YarnConfiguration.DEFAULT_DISK_VALIDATOR));
        LOG.info("Disk Validator: " + YarnConfiguration.DISK_VALIDATOR +
                " is loaded.");
        cacheTargetSize =
                conf.getLong(YarnConfiguration.NM_LOCALIZER_CACHE_TARGET_SIZE_MB,
YarnConfiguration.DEFAULT_NM_LOCALIZER_CACHE_TARGET_SIZE_MB) << 20;
        cacheCleanupPeriod =

 conf.getLong(YarnConfiguration.NM_LOCALIZER_CACHE_CLEANUP_INTERVAL_MS,
YarnConfiguration.DEFAULT_NM_LOCALIZER_CACHE_CLEANUP_INTERVAL_MS);

        // 本地化服务地址 默认 0.0.0.0:8040
        localizationServerAddress = conf.getSocketAddr(
                YarnConfiguration.NM_BIND_HOST,
                YarnConfiguration.NM_LOCALIZER_ADDRESS,
                YarnConfiguration.DEFAULT_NM_LOCALIZER_ADDRESS,
                YarnConfiguration.DEFAULT_NM_LOCALIZER_PORT);

        // 创建并添加 LocalizerTracker 服务
        localizerTracker = createLocalizerTracker(conf);
        addService(localizerTracker);
        dispatcher.register(LocalizerEventType.class, localizerTracker);
        localDirsChangeListener = new DirsChangeListener() {
            @Override
            public void onDirsChanged() {
                checkAndInitializeLocalDirs();
            }
        };
        logDirsChangeListener = new DirsChangeListener() {
            @Override
            public void onDirsChanged() {
```

```
                initializeLogDirs(lfs);
            }
        };
        super.serviceInit(conf);
    }
```

2.3.2.4.2 容器执行器服务 ContainersLauncher.serviceInit()

```
 @Override
  protected void serviceInit(Configuration conf) throws Exception {
    try {
      //TODO Is this required?
      FileContext.getLocalFSFileContext(conf);
    } catch (UnsupportedFileSystemException e) {
      throw new YarnRuntimeException("Failed to start ContainersLauncher", e);
    }
    super.serviceInit(conf);
  }
```

2.3.2.4.3 容器调度服务 ContainerScheduler.serviceInit()

```
@Override
  public void serviceInit(Configuration conf) throws Exception {
    super.serviceInit(conf);
    // 默认 false
    this.usePauseEventForPreemption =
        conf.getBoolean(
            YarnConfiguration.NM_CONTAINER_QUEUING_USE_PAUSE_FOR_PREEMPTION,
            YarnConfiguration.
                DEFAULT_NM_CONTAINER_QUEUING_USE_PAUSE_FOR_PREEMPTION);
  }
```

2.3.2.4.4 Aux 服务 AuxServices.serviceInit()

```
@Override
  public void serviceInit(Configuration conf) throws Exception {
    // 创建文件权限对象 FsPermission
    final FsPermission storeDirPerms = new FsPermission((short)0700);
    Path stateStoreRoot = null;
    FileSystem stateStoreFs = null;
    // 默认 false
    boolean recoveryEnabled = conf.getBoolean(
        YarnConfiguration.NM_RECOVERY_ENABLED,
        YarnConfiguration.DEFAULT_NM_RECOVERY_ENABLED);
    if (recoveryEnabled) {
      stateStoreRoot = new Path(conf.get(YarnConfiguration.NM_RECOVERY_DIR),
          STATE_STORE_ROOT_NAME);
      stateStoreFs = FileSystem.getLocal(conf);
    }

    // 默认返回空
```

```
    Collection<String> auxNames = conf.getStringCollection(
        YarnConfiguration.NM_AUX_SERVICES);
    for (final String sName : auxNames) {
        ......
    }
    super.serviceInit(conf);
  }
```

2.3.2.4.5 容器监控服务 ContainersMonitorImpl.serviceInit()

```
@Override
    protected void serviceInit(Configuration myConf) throws Exception {
        this.conf = myConf;
        // 监控间隔 默认 3000ms
        this.monitoringInterval =
                this.conf.getLong(YarnConfiguration.NM_CONTAINER_MON_INTERVAL_MS,

 this.conf.getLong(YarnConfiguration.NM_RESOURCE_MON_INTERVAL_MS,

 YarnConfiguration.DEFAULT_NM_RESOURCE_MON_INTERVAL_MS));

        // 创建资源计算插件 默认 ResourceCalculatorPlugin
        this.resourceCalculatorPlugin =
                ResourceCalculatorPlugin.getContainersMonitorPlugin(this.conf);
        LOG.info(" Using ResourceCalculatorPlugin : "
                + this.resourceCalculatorPlugin);

        // 创建资源计算处理树 默认 null
        processTreeClass = this.conf.getClass(
                YarnConfiguration.NM_CONTAINER_MON_PROCESS_TREE, null,
                ResourceCalculatorProcessTree.class);
        LOG.info(" Using ResourceCalculatorProcessTree : "
                + this.processTreeClass);

        // 是否启动容器监控 metrics 默认 true
        this.containerMetricsEnabled =
                this.conf.getBoolean(YarnConfiguration.NM_CONTAINER_METRICS_ENABLE,
                        YarnConfiguration.DEFAULT_NM_CONTAINER_METRICS_ENABLE);
        // 容器监控 metrics 周期 默认 -1
        this.containerMetricsPeriodMs =
                this.conf.getLong(YarnConfiguration.NM_CONTAINER_METRICS_PERIOD_MS,
                        YarnConfiguration.DEFAULT_NM_CONTAINER_METRICS_PERIOD_MS);
        // 默认 10s
        this.containerMetricsUnregisterDelayMs = this.conf.getLong(
                YarnConfiguration.NM_CONTAINER_METRICS_UNREGISTER_DELAY_MS,
                YarnConfiguration.DEFAULT_NM_CONTAINER_METRICS_UNREGISTER_DELAY_MS);

        // 获取配置容器内存 默认 8G
        long configuredPMemForContainers =
                NodeManagerHardwareUtils.getContainerMemoryMB(
                        this.resourceCalculatorPlugin, this.conf) * 1024 * 1024L;
        // 获取配置容器 CPU 数 默认 8
```

```
        long configuredVCoresForContainers =
                NodeManagerHardwareUtils.getVCores(this.resourceCalculatorPlugin,
                        this.conf);
        // Setting these irrespective of whether checks are enabled. Required in
        // the UI.
        // ********* Physical memory configuration ******
        this.maxPmemAllottedForContainers = configuredPMemForContainers;
        this.maxVCoresAllottedForContainers = configuredVCoresForContainers;

        // ********* Virtual memory configuration ******
        // 虚拟内存率 默认 2.1f
        vmemRatio = this.conf.getFloat(YarnConfiguration.NM_VMEM_PMEM_RATIO,
                YarnConfiguration.DEFAULT_NM_VMEM_PMEM_RATIO);
        Preconditions.checkArgument(vmemRatio > 0.99f,
                YarnConfiguration.NM_VMEM_PMEM_RATIO + " should be at least 1.0");
        // 默认 8G * 2.1f
        this.maxVmemAllottedForContainers =
                (long) (vmemRatio * configuredPMemForContainers);

        // 是否开启物理内存检查 默认 true
        pmemCheckEnabled = this.conf.getBoolean(
                YarnConfiguration.NM_PMEM_CHECK_ENABLED,
                YarnConfiguration.DEFAULT_NM_PMEM_CHECK_ENABLED);
        // 是否开启虚拟内存检查 默认 true
        vmemCheckEnabled = this.conf.getBoolean(
                YarnConfiguration.NM_VMEM_CHECK_ENABLED,
                YarnConfiguration.DEFAULT_NM_VMEM_CHECK_ENABLED);
        LOG.info("Physical memory check enabled: " + pmemCheckEnabled);
        LOG.info("Virtual memory check enabled: " + vmemCheckEnabled);

        // 默认开启容器监控
        containersMonitorEnabled =
                isContainerMonitorEnabled() && monitoringInterval > 0;
        LOG.info("ContainersMonitor enabled: " + containersMonitorEnabled);

        // 默认 100
        nodeCpuPercentageForYARN =
                NodeManagerHardwareUtils.getNodeCpuPercentage(this.conf);

        // 检查物理内存是否满足配置
        if (pmemCheckEnabled) {
            // Logging if actual pmem cannot be determined.
            long totalPhysicalMemoryOnNM = UNKNOWN_MEMORY_LIMIT;
            if (this.resourceCalculatorPlugin != null) {
                totalPhysicalMemoryOnNM = this.resourceCalculatorPlugin
                        .getPhysicalMemorySize();
                if (totalPhysicalMemoryOnNM <= 0) {
                    LOG.warn("NodeManager's totalPmem could not be calculated. "
                            + "Setting it to " + UNKNOWN_MEMORY_LIMIT);
                    totalPhysicalMemoryOnNM = UNKNOWN_MEMORY_LIMIT;
                }
            }
```

```
            if (totalPhysicalMemoryOnNM != UNKNOWN_MEMORY_LIMIT &&
                    this.maxPmemAllottedForContainers > totalPhysicalMemoryOnNM *
0.80f) {
                LOG.warn("NodeManager configured with "
                        +
TraditionalBinaryPrefix.long2String(maxPmemAllottedForContainers,
                        "", 1)
                        + " physical memory allocated to containers, which is more
than "
                        + "80% of the total physical memory available ("
                        +
TraditionalBinaryPrefix.long2String(totalPhysicalMemoryOnNM, "",
                        1) + "). Thrashing might happen.");
            }
        }
        super.serviceInit(this.conf);
    }
```

2.3.2.4.6 容器异步分发器服务 AsyncDispatcher.serviceInit()

```
// 调用其父类 AbstractService.serviceInit()
```

2.3.2.4.7 yarn 日志聚合服务 LogAggregationService.serviceInit()

```
protected void serviceInit(Configuration conf) throws Exception {
    // 默认 100
    int threadPoolSize = getAggregatorThreadPoolSize(conf);
    this.threadPool = HadoopExecutors.newFixedThreadPool(threadPoolSize,
        new ThreadFactoryBuilder()
            .setNameFormat("LogAggregationService #%d")
            .build());

    // 默认 -1
    rollingMonitorInterval = calculateRollingMonitorInterval(conf);
    LOG.info("rollingMonitorInterval is set as {}. The logs will be " +
        "aggregated every {} seconds", rollingMonitorInterval,
        rollingMonitorInterval);

    super.serviceInit(conf);
  }
```

2.3.2.4.8 总结

```
 /**
         * ContainerManagerImpl 容器管理服务是一个组合服务，里面添加了如下子服务
         * 1 ResourceLocalizationService     -> 本地资源服务
         * 2 ContainersLauncher              -> 容器执行器服务
         * 3 ContainerScheduler              -> 容器调度服务
         * 4 AuxServices                     -> Aux 服务
         * 5 ContainersMonitorImpl           -> 容器监控服务
```

```
         * 6 AsyncDispatcher                  -> 容器异步分发器服务
         * 7 LogAggregationService            -> yarn 日志聚合服务
         * 8 SharedCacheUploadService         -> 本地上传文件服务
         *
         * 容器管理服务 ContainerManagerImpl 的异步分发器 AsyncDispatcher 注册了
         * 哪些 EventType 与 EventHandler 映射关系
         * 1 ContainerEventType           -> ContainerEventDispatcher
         * 2 ApplicationEventType         -> ApplicationEventDispatcher
         * 3 LocalizationEventType        -> LocalizationEventHandlerWrapper
         * 4 AuxServicesEventType         -> AuxServices
         * 5 ContainersMonitorEventType   -> ContainersMonitorImpl
         * 6 ContainersLauncherEventType -> ContainersLauncher
         * 7 ContainerSchedulerEventType -> ContainerScheduler
         * 8 LocalizerEventType           -> LocalizerTracker
         * 9 LogHandlerEventType          -> LogAggregationService
         * 10 SharedCacheUploadEventType  -> SharedCacheUploadService
         */
```

2.3.2.5 NM 日志聚合追踪服务 NMLogAggregationStatusTracker.serviceInit()

```
// 调用其父类 CompositeService.serviceInit()
```

2.3.2.6 NM Web 服务 WebServer.serviceInit()

```
// 调用其父类 AbstractService.serviceInit()
```

2.3.2.7 NM 异步分发器服务 AsyncDispatcher.serviceInit()

```
// 调用其父类 AbstractService.serviceInit()
```

2.3.2.8 NM JVM 监控服务 JvmPauseMonitor.serviceInit()

```
// 暂时忽略
```

2.3.2.9 NM 节点状态更新服务 NodeStatusUpdaterImpl.serviceInit()

```
@Override
    protected void serviceInit(Configuration conf) throws Exception {
        // 获取节点硬件资源
        this.totalResource = NodeManagerHardwareUtils.getNodeResources(conf);
        // 一般在 yarn-site.xml 配置 NodeManager 的内存 比如 4096
        long memoryMb = totalResource.getMemorySize();
        float vMemToPMem =
                conf.getFloat(
                        YarnConfiguration.NM_VMEM_PMEM_RATIO,
                        YarnConfiguration.DEFAULT_NM_VMEM_PMEM_RATIO);
        // 虚拟内存计算 4096 * 2.1 = 8602
        long virtualMemoryMb = (long) Math.ceil(memoryMb * vMemToPMem);
        // 默认虚拟 CPU 8
        int virtualCores = totalResource.getVirtualCores();
```

```
        // Update configured resources via plugins.
        updateConfiguredResourcesViaPlugins(totalResource);

        // <memory:4096, vCores:8> 一般在 yarn-site.xml 配置 NodeManager 的 CPU 和内存
        LOG.info("Nodemanager resources is set to: " + totalResource);

        metrics.addResource(totalResource);

        // Get actual node physical resources
        long physicalMemoryMb = memoryMb;
        int physicalCores = virtualCores;
        // 默认 返回 ResourceCalculatorPlugin
        ResourceCalculatorPlugin rcp =
                ResourceCalculatorPlugin.getNodeResourceMonitorPlugin(conf);
        if (rcp != null) {
            physicalMemoryMb = rcp.getPhysicalMemorySize() / (1024 * 1024);
            physicalCores = rcp.getNumProcessors();
        }
        // 获取 NodeManager 所在节点的真实物理内存和CPU
        this.physicalResource =
                Resource.newInstance(physicalMemoryMb, physicalCores);

        this.tokenKeepAliveEnabled = isTokenKeepAliveEnabled(conf);
        this.tokenRemovalDelayMs =
                conf.getInt(YarnConfiguration.RM_NM_EXPIRY_INTERVAL_MS,
                        YarnConfiguration.DEFAULT_RM_NM_EXPIRY_INTERVAL_MS);

        this.minimumResourceManagerVersion = conf.get(
                YarnConfiguration.NM_RESOURCEMANAGER_MINIMUM_VERSION,
                YarnConfiguration.DEFAULT_NM_RESOURCEMANAGER_MINIMUM_VERSION);

        // 创建 NodeManager 标签处理器
        nodeLabelsHandler = createNMNodeLabelsHandler(nodeLabelsProvider);
        // Default duration to track stopped containers on nodemanager is 10Min.
        // This should not be assigned very large value as it will remember all the
        // containers stopped during that time.
        durationToTrackStoppedContainers =
                conf.getLong(YARN_NODEMANAGER_DURATION_TO_TRACK_STOPPED_CONTAINERS,
                        600000);
        if (durationToTrackStoppedContainers < 0) {
            String message = "Invalid configuration for "
                    + YARN_NODEMANAGER_DURATION_TO_TRACK_STOPPED_CONTAINERS + "
default "
                    + "value is 10Min(600000).";
            LOG.error(message);
            throw new YarnException(message);
        }
        if (LOG.isDebugEnabled()) {
            LOG.debug(YARN_NODEMANAGER_DURATION_TO_TRACK_STOPPED_CONTAINERS + " :"
                    + durationToTrackStoppedContainers);
        }
```

```
        super.serviceInit(conf);
        LOG.info("Initialized nodemanager with :" +
                " physical-memory=" + memoryMb + " virtual-memory=" +
virtualMemoryMb +
                " virtual-cores=" + virtualCores);

        this.logAggregationEnabled =
                conf.getBoolean(YarnConfiguration.LOG_AGGREGATION_ENABLED,
                        YarnConfiguration.DEFAULT_LOG_AGGREGATION_ENABLED);
        this.timelineServiceV2Enabled = YarnConfiguration.
                timelineServiceV2Enabled(conf);
    }
```

2.3.2.10 总结

```
 /**
         * NodeManager 组合服务添加了哪些子服务 (调用其 serviceInit())
         * 1 DeletionService                    -> 删除容器服务
         * 2 NodeHealthCheckerService          -> 节点监控检查服务
         * 3 NodeResourceMonitorImpl           -> 节点资源监控服务
         * 4 ContainerManagerImpl              -> 容器管理服务
         * 5 NMLogAggregationStatusTracker     -> NM 日志聚合追踪服务
         * 6 WebServer                         -> NM Web 服务
         * 7 AsyncDispatcher                   -> NM 异步分发器服务
         * 8 JvmPauseMonitor                   -> NM JVM 监控服务
         * 9 NodeStatusUpdaterImpl             -> NM 节点状态更新服务
         *
         * NodeManager 组合服务的异步分发器 AsyncDispatcher 注册了
         * 哪些 EventType 与 EventHandler 映射关系
         * 1 ContainerManagerEventType    -> ContainerManagerImpl
         * 2 NodeManagerEventType         -> NodeManager
         */
```

2.3.3 启动 NodeManager 组合服务的所有子服务的 serviceStart()

2.3.3.1 删除容器服务 DeletionService.serviceStart()

```
// 调用其父类 AbstractService.serviceStart()
```

2.3.3.2 节点监控检查服务 NodeHealthCheckerService.serviceStart()

```
// 调用其父类 CompositeService.serviceStart() 因其有一个子服务故调用子服务的 serviceStart()
/**
 * The class which provides functionality of checking the health of the local
 * directories of a node. This specifically manages nodemanager-local-dirs and
 * nodemanager-log-dirs by periodically checking their health.
 */
public class LocalDirsHandlerService extends AbstractService {
 /**
   * Method used to start the disk health monitoring, if enabled.
   */
```

```
  @Override
  protected void serviceStart() throws Exception {
    // 默认 true
    if (isDiskHealthCheckerEnabled) {
      dirsHandlerScheduler = new Timer("DiskHealthMonitor-Timer", true);
      dirsHandlerScheduler.scheduleAtFixedRate(monitoringTimerTask,
          diskHealthCheckInterval, diskHealthCheckInterval);
    }
    super.serviceStart();
  }
}
```

2.3.3.3 节点资源监控服务 NodeResourceMonitorImpl.serviceStart()

```
/**
     * Start the thread that does the node resource utilization monitoring.
     */
    @Override
    protected void serviceStart() throws Exception {
        if (this.isEnabled()) {
            // 启动 MonitoringThread 线程
            this.monitoringThread.start();
        }
        super.serviceStart();
    }
```

```
/**
     * Thread that monitors the resource utilization of this node.
     */
    private class MonitoringThread extends Thread {
        /**
         * Initialize the node resource monitoring thread.
         */
        public MonitoringThread() {
            super("Node Resource Monitor");
            this.setDaemon(true);
        }

        /**
         * Periodically monitor the resource utilization of the node.
         */
        @Override
        public void run() {
            while (true) {
                // Get node utilization and save it into the health status
                // 获取 NodeManager 已用物理内存资源(总量物理内存 - 可用物理内存)
                long pmem = resourceCalculatorPlugin.getPhysicalMemorySize() -
                        resourceCalculatorPlugin.getAvailablePhysicalMemorySize();
                // 获取 NodeManager 已用虚拟内存 (总量虚拟内存 - 可用虚拟内存)
                long vmem =
                        resourceCalculatorPlugin.getVirtualMemorySize()
```

```
                                -
resourceCalculatorPlugin.getAvailableVirtualMemorySize();
                // 获取 NodeManager 已用虚拟 CPU
                float vcores = resourceCalculatorPlugin.getNumVCoresUsed();
                nodeUtilization =
                        ResourceUtilization.newInstance(
                                (int) (pmem >> 20), // B -> MB
                                (int) (vmem >> 20), // B -> MB
                                vcores); // Used Virtual Cores

                // Publish the node utilization metrics to node manager
                // metrics system.
                // 发布 NodeManager 资源使用情况
                NodeManagerMetrics nmMetrics = nmContext.getNodeManagerMetrics();
                if (nmMetrics != null) {
                    nmMetrics.setNodeUsedMemGB(nodeUtilization.getPhysicalMemory());
                    nmMetrics.setNodeUsedVMemGB(nodeUtilization.getVirtualMemory());
                    nmMetrics.setNodeCpuUtilization(nodeUtilization.getCPU());
                }

                try {
                    Thread.sleep(monitoringInterval);
                } catch (InterruptedException e) {
                    LOG.warn(NodeResourceMonitorImpl.class.getName()
                            + " is interrupted. Exiting.");
                    break;
                }
            }
        }
    }
```

2.2.3.4 容器管理组合服务 ContainerManagerImpl.serviceStart() （重要服务）

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

        // Enqueue user dirs in deletion context

        Configuration conf = getConfig();
        // 默认 0.0.0.0:0
        final InetSocketAddress initialAddress = conf.getSocketAddr(
                YarnConfiguration.NM_BIND_HOST,
                YarnConfiguration.NM_ADDRESS,
                YarnConfiguration.DEFAULT_NM_ADDRESS,
                YarnConfiguration.DEFAULT_NM_PORT);
        // 默认 true
        boolean usingEphemeralPort = (initialAddress.getPort() == 0);

        if (context.getNMStateStore().canRecover() && usingEphemeralPort) {
            throw new IllegalArgumentException("Cannot support recovery with an "
                    + "ephemeral server port. Check the setting of "
                    + YarnConfiguration.NM_ADDRESS);
```

```
        }
        // If recovering then delay opening the RPC service until the recovery
        // of resources and containers have completed, otherwise requests from
        // clients during recovery can interfere with the recovery process.
        final boolean delayedRpcServerStart =
                context.getNMStateStore().canRecover();

        Configuration serverConf = new Configuration(conf);

        // always enforce it to be token-based.
        serverConf.set(
                CommonConfigurationKeysPublic.HADOOP_SECURITY_AUTHENTICATION,
                SaslRpcServer.AuthMethod.TOKEN.toString());

        // 创建 HadoopYarnProtoRPC
        YarnRPC rpc = YarnRPC.create(conf);

        // 创建容器管理 RPC server 并绑定通讯协议接口 ContainerManagementProtocol
        server =
                rpc.getServer(ContainerManagementProtocol.class,
                        this, initialAddress,
                        serverConf,
                        this.context.getNMTokenSecretManager(),
                        // 默认 handler 个数 20
                        conf.getInt(YarnConfiguration.NM_CONTAINER_MGR_THREAD_COUNT,

YarnConfiguration.DEFAULT_NM_CONTAINER_MGR_THREAD_COUNT));

        // Enable service authorization?
        if (conf.getBoolean(
                CommonConfigurationKeysPublic.HADOOP_SECURITY_AUTHORIZATION,
                false)) {
            refreshServiceAcls(conf, NMPolicyProvider.getInstance());
        }

        String bindHost = conf.get(YarnConfiguration.NM_BIND_HOST);
        String nmAddress = conf.getTrimmed(YarnConfiguration.NM_ADDRESS);
        String hostOverride = null;
        if (bindHost != null && !bindHost.isEmpty()
                && nmAddress != null && !nmAddress.isEmpty()) {
            //a bind-host case with an address, to support overriding the first
            //hostname found when querying for our hostname with the specified
            //address, combine the specified address with the actual port listened
            //on by the server
            hostOverride = nmAddress.split(":")[0];
        }

        // setup node ID
        InetSocketAddress connectAddress;
        if (delayedRpcServerStart) {
            connectAddress = NetUtils.getConnectAddress(initialAddress);
        } else {
```

```
            // 启动 RPC Server
            server.start();
            connectAddress = NetUtils.getConnectAddress(server);
        }
        // 创建节点 ID NodeId
        NodeId nodeId = buildNodeId(connectAddress, hostOverride);
        ((NodeManager.NMContext) context).setNodeId(nodeId);
        this.context.getNMTokenSecretManager().setNodeId(nodeId);
        this.context.getContainerTokenSecretManager().setNodeId(nodeId);

        // start remaining services
        super.serviceStart();

        if (delayedRpcServerStart) {
            waitForRecoveredContainers();
            server.start();

            // check that the node ID is as previously advertised
            connectAddress = NetUtils.getConnectAddress(server);
            NodeId serverNode = buildNodeId(connectAddress, hostOverride);
            if (!serverNode.equals(nodeId)) {
                throw new IOException("Node mismatch after server started, expected
'"
                        + nodeId + "' but found '" + serverNode + "'");
            }
        }

        // IP:随机端口
        LOG.info("ContainerManager started at " + connectAddress);
        // 0.0.0.0
        LOG.info("ContainerManager bound to " + initialAddress);
    }
```

2.3.3.4.1 本地资源服务 ResourceLocalizationService.serviceStart()

```
@Override
    public void serviceStart() throws Exception {
        // 定时执行 CacheCleanup 线程
        cacheCleanup.scheduleWithFixedDelay(new CacheCleanup(dispatcher),
                cacheCleanupPeriod, cacheCleanupPeriod, TimeUnit.MILLISECONDS);
        // 创建 RPC Server 并启动 绑定通讯协议接口 LocalizationProtocol
        server = createServer();
        server.start();

        localizationServerAddress =
                getConfig().updateConnectAddr(YarnConfiguration.NM_BIND_HOST,
                        YarnConfiguration.NM_LOCALIZER_ADDRESS,
                        YarnConfiguration.DEFAULT_NM_LOCALIZER_ADDRESS,
                        server.getListenerAddress());
        // 0.0.0.0:8040
        LOG.info("Localizer started on port " + server.getPort());
        super.serviceStart();
```

```
        dirsHandler.registerLocalDirsChangeListener(localDirsChangeListener);
        dirsHandler.registerLogDirsChangeListener(logDirsChangeListener);
    }
```

2.3.3.4.2 容器执行器服务 ContainersLauncher.serviceStart()

```
// 调用其父类 AbstractService.serviceStart()
```

2.3.3.4.3 容器调度服务 ContainerScheduler.serviceStart()

```
// 调用其父类 AbstractService.serviceStart()
```

2.3.3.4.4 Aux 服务 AuxServices.serviceStart()

```
@Override
  public void serviceStart() throws Exception {
    // TODO fork(?) services running as configured user
    //      monitor for health, shutdown/restart(?) if any should die
    for (Map.Entry<String, AuxiliaryService> entry : serviceMap.entrySet()) {
      AuxiliaryService service = entry.getValue();
      String name = entry.getKey();
      service.start();
      service.registerServiceListener(this);
      ByteBuffer meta = service.getMetaData();
      if(meta != null) {
        serviceMetaData.put(name, meta);
      }
    }
    super.serviceStart();
  }
```

2.3.3.4.5 容器监控服务 ContainersMonitorImpl.serviceStart()

```
@Override
    protected void serviceStart() throws Exception {
        if (containersMonitorEnabled) {
            // 启动容器监控线程 MonitoringThread
            this.monitoringThread.start();
        }
        super.serviceStart();
    }
```

```
private class MonitoringThread extends Thread {
        MonitoringThread() {
            super("Container Monitor");
        }

        @Override
        public void run() {
```

```
            while (!stopped && !Thread.currentThread().isInterrupted()) {
                // Print the processTrees for debugging.
                if (LOG.isDebugEnabled()) {
                    StringBuilder tmp = new StringBuilder("[ ");
                    for (ProcessTreeInfo p : trackingContainers.values()) {
                        tmp.append(p.getPID());
                        tmp.append(" ");
                    }
                    LOG.debug("Current ProcessTree list : "
                            + tmp.substring(0, tmp.length()) + "]");
                }

                // Temporary structure to calculate the total resource utilization
of
                // the containers
                // 临时结构去计算所有容器使用资源情况
                ResourceUtilization trackedContainersUtilization =
                        ResourceUtilization.newInstance(0, 0, 0.0f);

                // Now do the monitoring for the trackingContainers
                // Check memory usage and kill any overflowing containers
                long vmemUsageByAllContainers = 0;
                long pmemByAllContainers = 0;
                long cpuUsagePercentPerCoreByAllContainers = 0;
                // NodeManager 节点的所有容器
                for (Entry<ContainerId, ProcessTreeInfo> entry : trackingContainers
                        .entrySet()) {
                    ContainerId containerId = entry.getKey();
                    ProcessTreeInfo ptInfo = entry.getValue();
                    try {
                        // Initialize uninitialized process trees
                        initializeProcessTrees(entry);

                        String pId = ptInfo.getPID();
                        if (pId == null || !isResourceCalculatorAvailable()) {
                            continue; // processTree cannot be tracked
                        }
                        if (LOG.isDebugEnabled()) {
                            LOG.debug("Constructing ProcessTree for : PID = " + pId
                                    + " ContainerId = " + containerId);
                        }
                        ResourceCalculatorProcessTree pTree =
ptInfo.getProcessTree();
                        pTree.updateProcessTree();    // update process-tree
                        long currentVmemUsage = pTree.getVirtualMemorySize();
                        long currentPmemUsage = pTree.getRssMemorySize();

                        // if machine has 6 cores and 3 are used,
                        // cpuUsagePercentPerCore should be 300%
                        float cpuUsagePercentPerCore = pTree.getCpuUsagePercent();
                        if (cpuUsagePercentPerCore < 0) {
```

```
                            // CPU usage is not available likely because the
container just
                            // started. Let us skip this turn and consider this
container
                            // in the next iteration.
                            LOG.info("Skipping monitoring container " + containerId
                                    + " since CPU usage is not yet available.");
                            continue;
                        }

                        recordUsage(containerId, pId, pTree, ptInfo,
currentVmemUsage,
                                currentPmemUsage, trackedContainersUtilization);

                        checkLimit(containerId, pId, pTree, ptInfo,
                                currentVmemUsage, currentPmemUsage);

                        // Accounting the total memory in usage for all containers
                        vmemUsageByAllContainers += currentVmemUsage;
                        pmemByAllContainers += currentPmemUsage;
                        // Accounting the total cpu usage for all containers
                        cpuUsagePercentPerCoreByAllContainers +=
cpuUsagePercentPerCore;

                        reportResourceUsage(containerId, currentPmemUsage,
                                cpuUsagePercentPerCore);
                    } catch (Exception e) {
                        // Log the exception and proceed to the next container.
                        LOG.warn("Uncaught exception in ContainersMonitorImpl "
                                + "while monitoring resource of {}", containerId,
e);
                    }
                }
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Total Resource Usage stats in NM by all containers :
"
                            + "Virtual Memory= " + vmemUsageByAllContainers
                            + ", Physical Memory= " + pmemByAllContainers
                            + ", Total CPU usage(% per core)= "
                            + cpuUsagePercentPerCoreByAllContainers);
                }

                // Save the aggregated utilization of the containers
                setContainersUtilization(trackedContainersUtilization);

                // Publish the container utilization metrics to node manager
                // metrics system.
                NodeManagerMetrics nmMetrics = context.getNodeManagerMetrics();
                if (nmMetrics != null) {
                    nmMetrics.setContainerUsedMemGB(
                            trackedContainersUtilization.getPhysicalMemory());
                    nmMetrics.setContainerUsedVMemGB(
```

```
                        trackedContainersUtilization.getVirtualMemory());
                nmMetrics.setContainerCpuUtilization(
                        trackedContainersUtilization.getCPU());
            }

            try {
                Thread.sleep(monitoringInterval);
            } catch (InterruptedException e) {
                LOG.warn(ContainersMonitorImpl.class.getName()
                        + " is interrupted. Exiting.");
                break;
            }
        }
    }
```

2.3.3.4.6 容器异步分发器服务 AsyncDispatcher.serviceStart()

```
@Override
    protected void serviceStart() throws Exception {
        //start all the components
        // 针对 ResourceManager 的 AsyncDispatcher 服务来说 啥也不干
        super.serviceStart();
        // 创建 EventHandler 线程
        eventHandlingThread = new Thread(createThread());
        eventHandlingThread.setName(dispatcherThreadName);
        eventHandlingThread.start();
    }
```

2.3.3.4.7 yarn 日志聚合服务 LogAggregationService.serviceStart()

```
  @Override
  protected void serviceStart() throws Exception {
    // NodeId is only available during start, the following cannot be moved
    // anywhere else.
    this.nodeId = this.context.getNodeId();
    super.serviceStart();
  }
```

2.3.3.5 NM 日志聚合追踪服务 NMLogAggregationStatusTracker.serviceStart()

```
@Override
  protected void serviceStart() throws Exception {
    if (disabled) {
      LOG.warn("Log Aggregation is disabled."
          + "So is the LogAggregationStatusTracker.");
    } else {
      this.timer.scheduleAtFixedRate(new LogAggregationStatusRoller(),
          rollingInterval, rollingInterval);
    }
  }
```

2.3.3.6 NM Web 服务 WebServer.serviceStart()

```
@Override
    protected void serviceStart() throws Exception {
        Configuration conf = getConfig();
        // 默认启动 HTTP 并绑定 0.0.0.0:8042
        String bindAddress = WebAppUtils.getWebAppBindURL(conf,
                YarnConfiguration.NM_BIND_HOST,
                WebAppUtils.getNMWebAppURLWithoutScheme(conf));
        // 默认 false
        boolean enableCors = conf
                .getBoolean(YarnConfiguration.NM_WEBAPP_ENABLE_CORS_FILTER,
                        YarnConfiguration.DEFAULT_NM_WEBAPP_ENABLE_CORS_FILTER);
        if (enableCors) {
            getConfig().setBoolean(HttpCrossOriginFilterInitializer.PREFIX
                    + HttpCrossOriginFilterInitializer.ENABLED_SUFFIX, true);
        }

        // Always load pseudo authentication filter to parse "user.name" in an URL
        // to identify a HTTP request's user.
        boolean hasHadoopAuthFilterInitializer = false;
        String filterInitializerConfKey = "hadoop.http.filter.initializers";
        Class<?>[] initializersClasses =
                conf.getClasses(filterInitializerConfKey);
        List<String> targets = new ArrayList<String>();
        if (initializersClasses != null) {
            for (Class<?> initializer : initializersClasses) {
                if (initializer.getName().equals(
                        AuthenticationFilterInitializer.class.getName())) {
                    hasHadoopAuthFilterInitializer = true;
                    break;
                }
                targets.add(initializer.getName());
            }
        }
        if (!hasHadoopAuthFilterInitializer) {
            targets.add(AuthenticationFilterInitializer.class.getName());
            conf.set(filterInitializerConfKey, StringUtils.join(",", targets));
        }
        // 0.0.0.0:8042
        LOG.info("Instantiating NMWebApp at " + bindAddress);
        try {
            this.webApp =
                    WebApps
                            .$for("node", Context.class, this.nmContext, "ws")
                            .at(bindAddress)
                            .with(conf)
                            .withHttpSpnegoPrincipalKey(

 YarnConfiguration.NM_WEBAPP_SPNEGO_USER_NAME_KEY)
                            .withHttpSpnegoKeytabKey(
```

```
YarnConfiguration.NM_WEBAPP_SPNEGO_KEYTAB_FILE_KEY)
                            .withCSRFProtection(YarnConfiguration.NM_CSRF_PREFIX)
                            .withXFSProtection(YarnConfiguration.NM_XFS_PREFIX)
                            .start(this.nmWebApp);
            this.port = this.webApp.httpServer().getConnectorAddress(0).getPort();
        } catch (Exception e) {
            String msg = "NMWebapps failed to start.";
            LOG.error(msg, e);
            throw new YarnRuntimeException(msg, e);
        }
        super.serviceStart();
    }
```

2.3.3.7 NM 异步分发器服务 AsyncDispatcher.serviceStart()

```
@Override
    protected void serviceStart() throws Exception {
        //start all the components
        // 针对 ResourceManager 的 AsyncDispatcher 服务来说 啥也不干
        super.serviceStart();
        // 创建 EventHandler 线程
        eventHandlingThread = new Thread(createThread());
        eventHandlingThread.setName(dispatcherThreadName);
        eventHandlingThread.start();
    }
```

2.3.3.8 NM JVM 监控服务 JvmPauseMonitor.serviceStart()

```
// 暂时忽略
```

2.3.3.9 NM 节点状态更新服务 NodeStatusUpdaterImpl.serviceStart()

```
@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);
        }
    }
```

2.2.3.3.9.1 NM 发送 RPC 请求向 RM 注册

```
@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());
    }
```

```
public class ResourceTrackerService extends AbstractService implements
        ResourceTracker {
 @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;
    }
}
```

```
public class RMNodeImpl implements RMNode, EventHandler<RMNodeEvent> {
public void handle(RMNodeEvent event) {
        LOG.debug("Processing " + event.getNodeId() + " of type " +
event.getType());
        try {
            writeLock.lock();
            NodeState oldState = getState();
            try {
                // 状态机 event.getType()= STARTED (NEW -> STARTED)
                // (最终调用 AddNodeTransition.transition())
                stateMachine.doTransition(event.getType(), event);
            } catch (InvalidStateTransitionException e) {
                LOG.error("Can't handle this event at current state", e);
                LOG.error("Invalid event " + event.getType() +
                        " on Node  " + this.nodeId + " oldState " + oldState);
            }
            if (oldState != getState()) {
                LOG.info(nodeId + " Node Transitioned from " + oldState + " to "
                        + getState());
            }
        } finally {
            writeLock.unlock();
        }
    }
}
```

```
public static class AddNodeTransition implements
            SingleArcTransition<RMNodeImpl, RMNodeEvent> {
 @Override
        public void transition(RMNodeImpl rmNode, RMNodeEvent event) {
            // Inform the scheduler
            RMNodeStartedEvent startEvent = (RMNodeStartedEvent) event;
            List<NMContainerStatus> containers = null;

            // NM 节点的 IP:PORT
            NodeId nodeId = rmNode.nodeId;
            RMNode previousRMNode =
                    rmNode.context.getInactiveRMNodes().remove(nodeId);
            if (previousRMNode != null) {
                rmNode.updateMetricsForRejoinedNode(previousRMNode.getState());
            } else {
                // NM 注册场景 默认 null
                NodeId unknownNodeId =
                        NodesListManager.createUnknownNodeId(nodeId.getHost());
                // NM 注册场景 默认 null
```

```
                previousRMNode =
                        rmNode.context.getInactiveRMNodes().remove(unknownNodeId);
                if (previousRMNode != null) {
                    ClusterMetrics.getMetrics().decrDecommisionedNMs();
                }
                // Increment activeNodes explicitly because this is a new node.
                ClusterMetrics.getMetrics().incrNumActiveNodes();
                // NM 注册场景 默认空
                containers = startEvent.getNMContainerStatuses();
                if (containers != null && !containers.isEmpty()) {
                    for (NMContainerStatus container : containers) {
                        if (container.getContainerState() == ContainerState.RUNNING)
{

 rmNode.launchedContainers.add(container.getContainerId());
                        }
                    }
                }
            }

            // NM 注册场景 默认 null
            if (null != startEvent.getRunningApplications()) {
                for (ApplicationId appId : startEvent.getRunningApplications()) {
                    handleRunningAppOnNode(rmNode, rmNode.context, appId,
rmNode.nodeId);
                }
            }

            // 调用 EventDispatcher.handle(NODE_ADDED)
            // (将该 Event 存储阻塞队列等待 CapacityScheduler.handle(NODE_ADDED) 拉取)
            rmNode.context.getDispatcher().getEventHandler()
                    .handle(
                            // 创建 NodeAddedSchedulerEvent(NODE_ADDED)
                            new NodeAddedSchedulerEvent(rmNode, containers));
            // 调用 NodesListManager.handle(NODE_USABLE)
            rmNode.context.getDispatcher().getEventHandler().handle(
                    // 创建 NodesListManagerEvent(NODE_USABLE)
                    new NodesListManagerEvent(
                            NodesListManagerEventType.NODE_USABLE, rmNode));

            List<LogAggregationReport> logAggregationReportsForApps =
                    startEvent.getLogAggregationReportsForApps();
            if (logAggregationReportsForApps != null
                    && !logAggregationReportsForApps.isEmpty()) {
                rmNode.handleLogAggregationStatus(logAggregationReportsForApps);
            }
        }
    }
}
```

```
public class CapacityScheduler extends
```

```
        AbstractYarnScheduler<FiCaSchedulerApp, FiCaSchedulerNode> implements
        PreemptableResourceScheduler, CapacitySchedulerContext, Configurable,
        ResourceAllocationCommitter, MutableConfScheduler {
         @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());
            }
            ......
        }
}

================================================================================
    private void addNode(RMNode nodeManager) {
        try {
            writeLock.lock();
            // 创建封装 NM 节点对象 FiCaSchedulerNode
            FiCaSchedulerNode schedulerNode = new FiCaSchedulerNode(nodeManager,
                    usePortForNodeName, nodeManager.getNodeLabels());
            // 添加 NM 节点
            nodeTracker.addNode(schedulerNode);

            // update this node to node label manager
            if (labelManager != null) {
                labelManager.activateNode(nodeManager.getNodeID(),
                        schedulerNode.getTotalResource());
            }

            Resource clusterResource = getClusterResource();
            getRootQueue().updateClusterResource(clusterResource,
                    new ResourceLimits(clusterResource));

            // Added node hadoop103:40144 clusterResource: <memory:12288, vCores:24>
            LOG.info(
                    "Added node " + nodeManager.getNodeAddress() + "
clusterResource: "
                            + clusterResource);

            if (scheduleAsynchronously && getNumClusterNodes() == 1) {
                for (AsyncScheduleThread t : asyncSchedulerThreads) {
                    t.beginSchedule();
                }
            }
        } finally {
            writeLock.unlock();
```

```
        }
    }
```

2.3.3.9.2 NM 发送 RPC 请求向 RM 发送心跳

```
    protected void startStatusUpdater() {
        // 创建并启动心跳线程 StatusUpdaterRunnable
        statusUpdaterRunnable = new StatusUpdaterRunnable();
        statusUpdater =
                new Thread(statusUpdaterRunnable, "Node Status Updater");
        statusUpdater.start();
    }
```

```
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
                    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
                        }
                    }
                }
            }
        }
```

```
public class ResourceTrackerService extends AbstractService implements
        ResourceTracker {
  @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;
    }
}
```