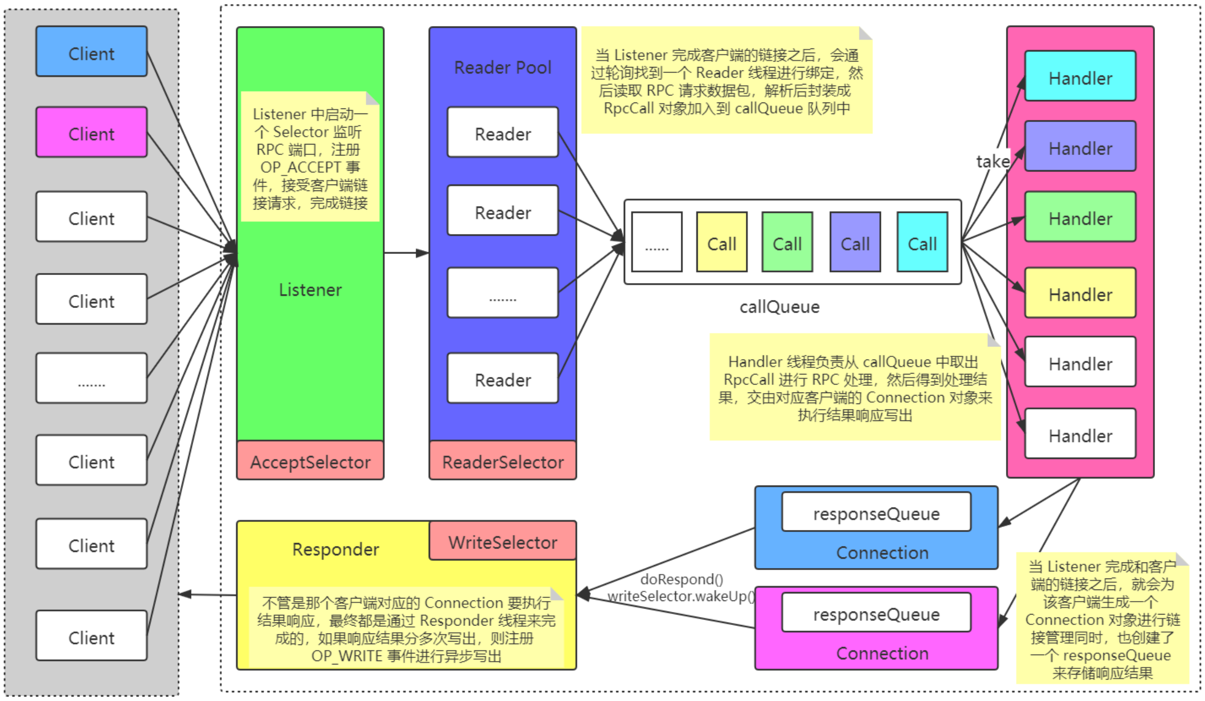
hadoop-yarn核心源码架构分析

一 概述

YARN 的核心设计理念是服务化（Service） 和 事件驱动（Event + EventHandler）。服务化 和 事件驱动软件设计思想的引入，使得 YARN 具有低耦合、高内聚的特点，各个模块只需完成各自功能，而模块之间则采用事件联系起来，系统设计简单且维护方便。这种编程方式具有异步、并发等特点，更加高效，更适合大型分布式系统

二 YARN RPC 网络通讯

YARN RPC 网络通讯架构以及源码分析参考 HDFS 的 RPC



Listener 中启动一个 Selector 监听 RPC 端口，注册 OP_ACCEPT 事件，接受客户端链接请求，完成链接

当 Listener 完成客户端的链接之后，会通过轮询找到一个 Reader 线程进行绑定，然后读取 RPC 请求数据包，解析后封装成 RpcCall 对象加入到 callQueue 队列中

Handler 线程负责从 callQueue 中取出 RpcCall 进行 RPC 处理，然后得到处理结果，交由对应客户端的 Connection 对象来执行结果响应写出

当 Listener 完成和客户端的链接之后，就会为该客户端生成一个 Connection 对象进行链接管理同时，也创建了一个 responseQueue 来存储响应结果

不管是那个客户端对应的 Connection 要执行结果响应，最终都是通过 Responder 线程来完成的，如果响应结果分多次写出，则注册 OP_WRITE 事件进行异步写出

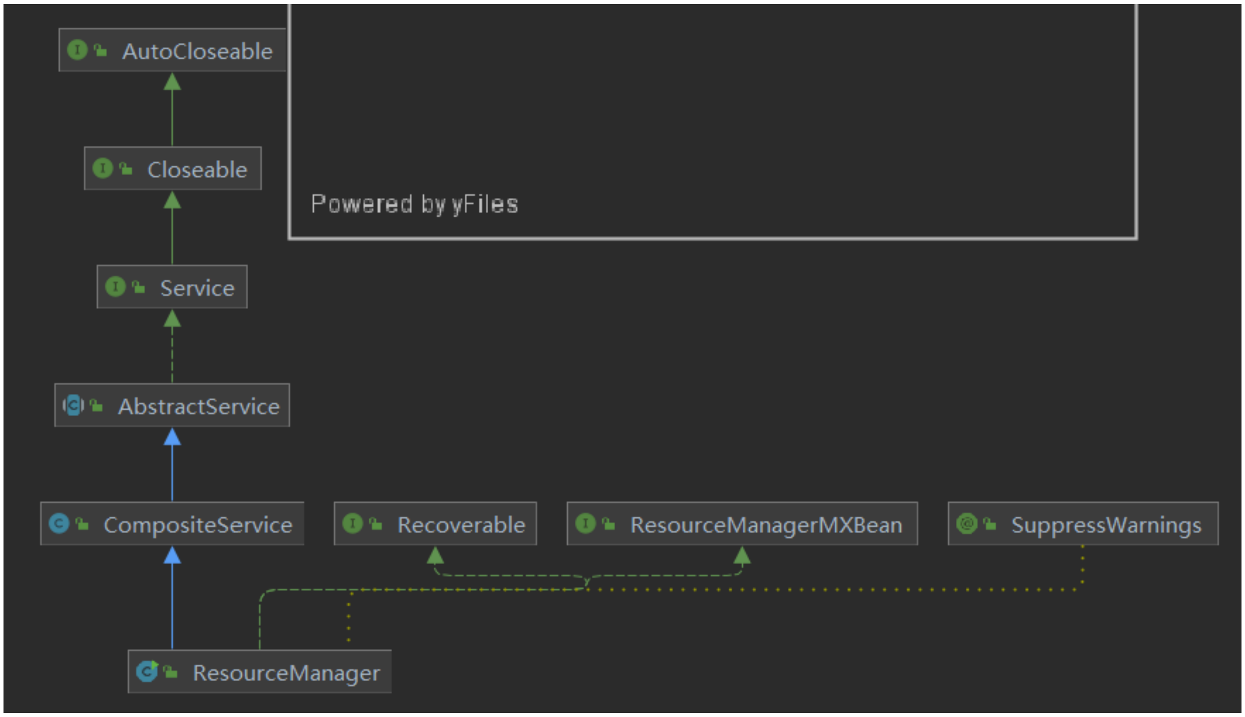
三 YARN Service 服务库

3.1 概述

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

- 将每个被服务化的对象分为 4 个状态：NOTINITED（被创建）、INITED（已初始化）、STARTED（已启动）、STOPPED（已停止）
- 任何服务状态变化都可以触发另外一些动作
- 可通过组合的方式对任意服务进行组合，以便进行统一管理

3.2 源码设计



3.2.1 Service 剖析

```
/**
 * hadoop-yarn 服务接口
 */
public interface Service {

    /**
     * 服务状态枚举
     */
    public enum STATE {

        /**
         * 调用服务构造函数
         */
        NOTINITED(0, "NOTINITED"),

        /**
         * 调用服务 serviceInit()
         */
        INITED(1, "INITED"),

        /**
         * 调用服务的 serviceStart()
         */
        STARTED(2, "STARTED"),

        /**
         * 调用服务的 serviceStop()
         */
        STOPPED(3, "STOPPED");

        private final int value;
        private final String statename;

        private STATE(int value, String name) {
            this.value = value;
            this.statename = name;
        }

        public int getValue() {
            return value;
        }

        @Override
        public String toString() {
            return statename;
        }
    }

    // 服务生命周期调用
    void init(Configuration config);

    void start();

    void stop();

    void close() throws IOException;

    STATE getServiceState();

}
```

3.2.2 AbstractService 剖析

```
/**
 * 抽象的服务对象 (主要提供服务状态的改变功能)
 */
public abstract class AbstractService implements Service {

    private final String name;
    private volatile Configuration config;
    private final ServiceStateModel stateModel;

    public AbstractService(String name) {
        this.name = name;
        stateModel = new ServiceStateModel(name);
    }

    protected void setConfig(Configuration conf) {
        this.config = conf;
```

```java
    }

    @Override
    public void init(Configuration conf) {
        setConfig(conf);
        serviceInit(config);
    }

    protected void serviceInit(Configuration config) {
        setConfig(config);
        enterState(STATE.INITED);
    }

    @Override
    public void start() {
        try {
            serviceStart();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    protected void serviceStart() throws Exception {
        enterState(STATE.STARTED);
    }

    @Override
    public void stop() {
        try {
            serviceStop();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    protected void serviceStop() throws Exception {
        enterState(STATE.STOPPED);
    }

    @Override
    public final void close() throws IOException {
        stop();
    }

    public String getName() {
        return name;
    }

    private STATE enterState(STATE newState) {
        return stateModel.enterState(newState);
    }
    @Override
    public final STATE getServiceState() {
        return stateModel.getState();
    }

}
```

3.2.3 CompositeService 剖析

```
/**
 * 组合服务（本身是一个服务，该服务维护很多其他服务）
 */
public class CompositeService extends AbstractService {

    private final List<Service> serviceList = new ArrayList<Service>();

    public CompositeService(String name) {
        super(name);
    }

    public List<Service> getServices() {
        synchronized (serviceList) {
            return new ArrayList<Service>(serviceList);
        }
    }

    protected void addService(Service service) {
        synchronized (serviceList) {
            serviceList.add(service);
        }
    }

    protected void serviceInit(Configuration conf) {
        List<Service> services = getServices();
        for (Service service : services) {
            service.init(conf);
        }
        super.serviceInit(conf);
    }

    protected void serviceStart() throws Exception {
        List<Service> services = getServices();
        for (Service service : services) {
            service.start();
        }
        super.serviceStart();
    }

    protected void serviceStop() throws Exception {
        List<Service> services = getServices();
        for (Service service : services) {
            service.stop();
        }
        super.serviceStart();
    }

}
```

3.2.4 ResourceManager 剖析

```
/**
 * 组合服务
 */
public class ResourceManager extends CompositeService {

    public ResourceManager(String name) {
        super(name);
    }

    public static void main(String[] args) {

        ResourceManager resourceManager = new ResourceManager("ResourceManager");
        // ResourceManager 服务添加子服务 （可以是组合服务）
        resourceManager.addService(new AService("AService"));
        resourceManager.addService(new BService("BService"));
        resourceManager.addService(new CService("CService"));

        // 初始化服务 （先初始化子服务再启动该服务）
        resourceManager.init(new Configuration());

        // 启动服务 （先启动子服务再启动该服务）
        resourceManager.start();

        // 停止服务 （先停止子服务再启动该服务）
        resourceManager.stop();

    }

}
```

四 YARN AsyncDispatcher事件驱动机制

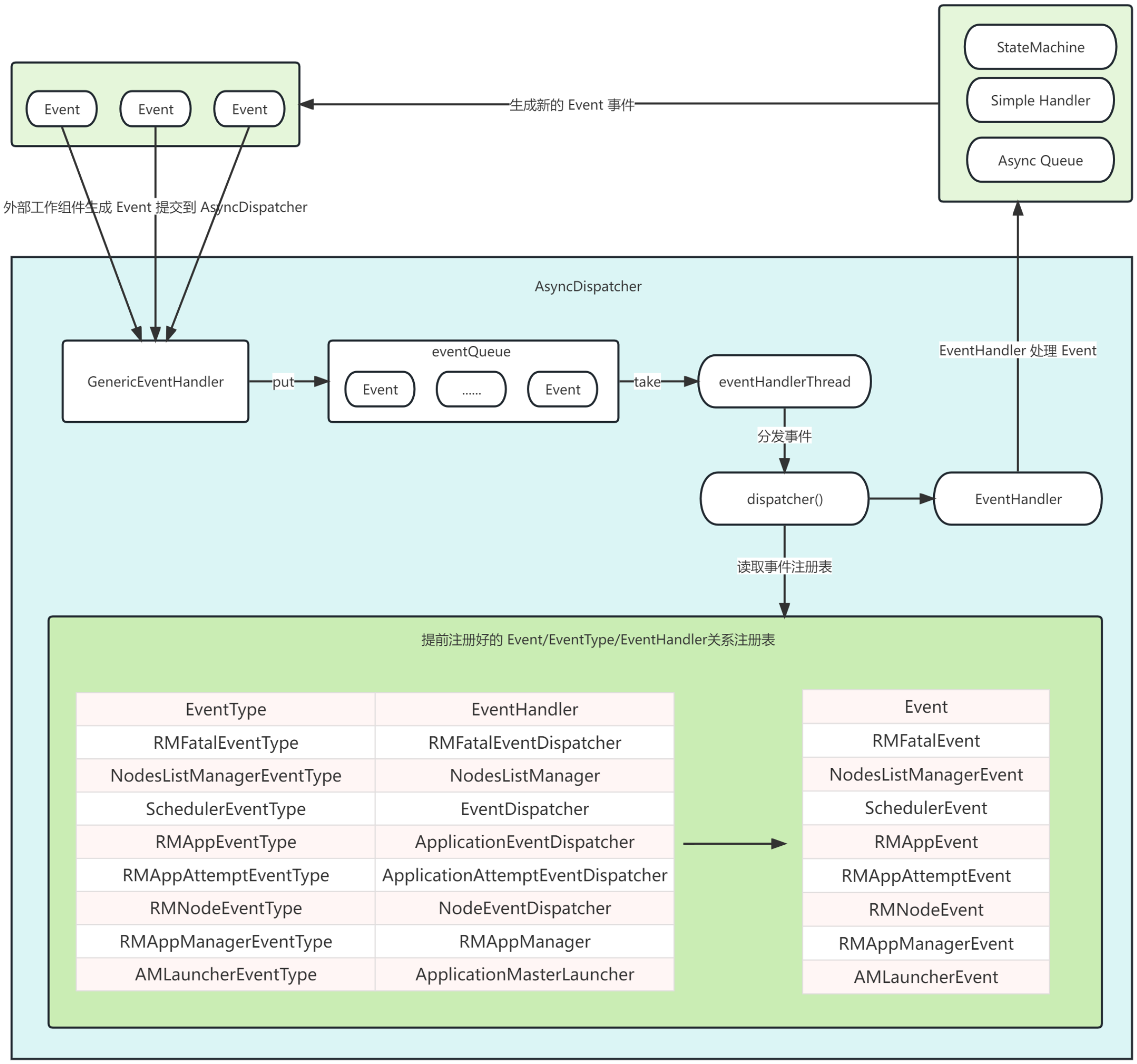
4.1 概述

AsyncDispatcher 中央事件调度器 = 其他的组件提交事件到这个 AsyncDispatcher，内部通过一个队列来缓冲，然后 AsyncDispatcher 的内部还有一个消费线程，获取队列中的 事件，然后做分发到事件和事件处理器的注册表中，去根据事件找到事件处理器，调用事件处理器的 handle 方法来完成事件处理

异步：这个异步组件的内部，必然包含一个 队列 和一个 消费线程

AsyncDispatcher 异步事件驱动模型的中央事件调度器，所以它的内部，注册了很多的<Event + EventHandler> 事件处理二元组

YARN 采用了基于事件驱动的异步通信模型，该模型能够大大增强并发性，从而提高系统整体性能。为了构建该模型，YARN 将各种处理逻辑抽象成事件和对应事件调度器，并将每类事件的处理过程分割成多个步骤，用有限状态机表示。AsycDispatcher 是 YARN 的中央异步调度器/全局事件分发器。在 ResourceManager 中，几乎所有的事件都通过 AsyncDispatcher 进行事件的派发。当一个组件提交一个 事件给 AsyncDispatcher 的时候，AsyncDispatcher 负责给这个事件找到之前注册的 EventHandler 来完成响应的处理。某些特别的 EventHandler 可能是一个 StateMachine



EventType	EventHandler
RMFatalEventType	RMFatalEventDispatcher
NodesListManagerEventType	NodesListManager
SchedulerEventType	EventDispatcher
RMAppEventType	ApplicationEventDispatcher
RMAppAttemptEventType	ApplicationAttemptEventDispatcher
RMNodeEventType	NodeEventDispatcher
RMAppManagerEventType	RMAppManager
AMLauncherEventType	ApplicationMasterLauncher

Event
RMFatalEvent
NodesListManagerEvent
SchedulerEvent
RMAppEvent
RMAppAttemptEvent
RMNodeEvent
RMAppManagerEvent
AMLauncherEvent

整个处理过程大致为：处理请求会作为事件进入系统，由中央异步调度器（AsyncDispatcher）负责传递给相应事件调度器（EventHandler）。该事件调度器可能将该事件转发给另外一个事件调度器，也可能交给一个带有有限状态机的事件处理器，其处理结果也以事件的形式输出给中央异步调度器。而新的事件会再次被中央异步调度器转发给下一个事件调度器，直至处理完成（达到终止条件）。

在 YARN 中，所有核心服务实际上都是一个中央异步调度器，包括 ResourceManager、NodeManager、MRAppMaster 等，它们维护了事先注册的事件与事件处理器，并根据接收的事件类型驱动服务的运行。以 MRAppMaster 为例，它内部包含一个中央异步调度器 AsyncDispatcher，并注册了 TaskAttemptEvent /TaskAttemptImpl、TaskEvent / TaskImpl、JobEvent / JobImpl 等一系列事件/事件处理器，由中央异步调度器统一管理和调度

4.2 源码设计

```
/**
 * 事件分发器
 */
public class AsyncDispatcher {

    // 事件队列
    private final BlockingQueue<Event> eventQueue;

    // 通用 EventHandler
    private final EventHandler<Event> handlerInstance = new GenericEventHandler();

    // 事件和事件处理器关系注册表
    protected final Map<Class<? extends Enum>, EventHandler> eventDispatchers;

    // 队列消费线程
    private final Thread eventHandlingThread;

    // 停止标志
    private volatile boolean stopped = false;

    public AsyncDispatcher() {

        this.eventQueue = new LinkedBlockingQueue<>();
        this.eventDispatchers = new HashMap<>();

        // 启动线程
        eventHandlingThread = new Thread(createThread());
        eventHandlingThread.start();
    }

    Runnable createThread() {
        return () -> {
            while (!stopped) {
                // 获取事件
                Event event;
                try {
                    event = eventQueue.take();
                } catch (InterruptedException ie) {
                    return;
                }
                // 执行分发
                try {
                    dispatch(event);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };
    }
    @SuppressWarnings("unchecked")
    private void dispatch(Event event) throws Exception {
        Class<? extends Enum> type = event.getType().getDeclaringClass();
        // 1 读取注册表，获取 event 对应的 eventHandler
        EventHandler handler = eventDispatchers.get(type);
        if (handler != null) {
            // 2 调用 eventHandler 的 handle 方法来执行事件处理
            handler.handle(event);
        } else {
            throw new Exception("No handler for registered for " + type);
        }
    }

    // Event 和 EventHandler 注册
    @SuppressWarnings("unchecked")
    public void register(Class<? extends Enum> eventType, EventHandler handler) {
        EventHandler<Event> registeredHandler = (EventHandler<Event>)
eventDispatchers.get(eventType);
        if (registeredHandler == null) {
            eventDispatchers.put(eventType, handler);
        } else {
            System.out.println("注册的事件已存在");
        }
    }

    public EventHandler<Event> getEventHandler() {
        return handlerInstance;
    }

    // 通用处理器
    class GenericEventHandler implements EventHandler<Event> {
        @Override
        public void handle(Event event) {
            try {
                eventQueue.put(event);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    // AsyncDispatcher 停止工作
    void stop() {
        this.stopped = true;
    }

    public static void main(String[] args) throws InterruptedException {

        // 1 构建 AsyncDispatcher 事件分发器
        AsyncDispatcher asyncDispatcher = new AsyncDispatcher();
        // 2 注册事件和事件处理器
        SportEventHandler sportEventHandler = new SportEventHandler();
        LearnEventHandler learnEventHandler = new LearnEventHandler();
        asyncDispatcher.register(LearnEventType.class, learnEventHandler);
        asyncDispatcher.register(SportEventType.class, sportEventHandler);

        // 3 休息 3s 等待线程启动成功
        Thread.sleep(3000);

        // 4 开始提交事件
        asyncDispatcher.getEventHandler().handle(new
SportEvent(SportEventType.RUN));
        Thread.sleep(1000);

        asyncDispatcher.getEventHandler().handle(new
SportEvent(SportEventType.BIKE));
        Thread.sleep(1000);

        asyncDispatcher.getEventHandler().handle(new
LearnEvent(LearnEventType.READ));
        Thread.sleep(1000);

        asyncDispatcher.getEventHandler().handle(new
LearnEvent(LearnEventType.LISTEN));
        Thread.sleep(1000);

        asyncDispatcher.getEventHandler().handle(new
LearnEvent(LearnEventType.SPEAK));
        Thread.sleep(1000);

        // 5 测试结束关闭线程（线程处于阻塞关闭不了）
        // asyncDispatcher.stop();
    }
}

// 事件抽象
interface Event<TYPE extends Enum<TYPE>> {

    TYPE getType();

    long getTimestamp();

    String toString();
}

// Event 的抽象实现
abstract class AbstractEvent<TYPE extends Enum<TYPE>> implements Event<TYPE> {
    private final TYPE type;
    private final long timestamp;

    public AbstractEvent(TYPE type) {
        this.type = type;
        timestamp = -1L;
    }

    public AbstractEvent(TYPE type, long timestamp) {
        this.type = type;
        this.timestamp = timestamp;
    }

    @Override
    public long getTimestamp() {
        return timestamp;
    }

    @Override
    public TYPE getType() {
        return type;
    }

    @Override
    public String toString() {
        return "EventType: " + getType();
    }
}

// 事件处理器抽象
interface EventHandler<T extends Event> {
    void handle(T event);
}

// Sport 事件类实现
class SportEvent extends AbstractEvent<SportEventType> {
    public SportEvent(SportEventType type) {
        super(type);
    }
}

// Sport 事件类型
enum SportEventType {
    RUN,
    BIKE,
    SWIM,
    CLIMB;
}

// Sport 事件对应的处理器
class SportEventHandler implements EventHandler<SportEvent> {

    @Override
    public void handle(SportEvent event) {
        switch (event.getType()) {
            case RUN:
                System.out.println("RUN...");
                break;
            case BIKE:
                System.out.println("BIKE...");
                break;
            case SWIM:
                System.out.println("SWIM...");
                break;
            case CLIMB:
                System.out.println("CLIMB...");
                break;
            default:
                break;
        }
    }
}

// Learn 事件实现
class LearnEvent extends AbstractEvent<LearnEventType> {
    public LearnEvent(LearnEventType type) {
        super(type);
    }
}

// Learn 事件类型
enum LearnEventType {
    READ,
    LISTEN,
    WRITE,
    SPEAK
}

// Learn 事件对应的事件处理器
class LearnEventHandler implements EventHandler<LearnEvent> {

    @Override
    public void handle(LearnEvent event) {
        switch (event.getType()) {
            case READ:
                System.out.println("READ...");
                break;
            case LISTEN:
                System.out.println("LISTEN...");
                break;
            case WRITE:
                System.out.println("WRITE...");
                break;
            case SPEAK:
                System.out.println("SPEAK...");
                break;
            default:
                break;
        }
    }
}
```

五 YARN StateMachine 有限状态机

5.1 概述

状态机由一组状态组成，这些状态分为三类：初始状态、中间状态和最终状态。状态机从初始状态开始运行，经过一系列中间状态后，到达最终状态并退出。在一个状态机中，每个状态都可以接收一组特定事件，并根据具体的事件类型转换到另一个状态。当状态机转换到最终状态时，则退出

在 YARN 中，每种状态转换（doTransition 方法去执行状态转换，addTransition 注册状态转换）由一个四元组表示，分别是转换前状态（preState）、转换后状态（postState）、事件（event）和回调函数/转换器（hook = Transition）

连起来解释：此时是 preState，接收到 event 之后，状态机转换表中获取到对应的 Transition 执行转换，得到一个新状态 postState

YARN 定义了三种状态转换方式，具体如下：

- 一个初始状态、一个最终状态、一种事件：经过处理之后，无论如何，进入到一个唯一状态
- 一个初始状态、多个最终状态、一种事件：不同的逻辑处理结果，可能导致进入不同的状态
- 一个初始状态、一个最终状态、多种事件：多个不用的事件，可能触发到多个不同状态的转换

5.2 源码设计

```
/**
 * 人一生可能出现的正常和意外情况下，导致人的状态发生改变
 */
enum ManEventType {
    GROWUP,             // 正常长大
    SICK,               // 生病
    REJUVENATION,       // 返老还童
    INCURABLE_DISEASE   // 绝症
}

/**
 * 人一生的状态
 */
enum ManState {
    BIRTH(1),               // 出生
    BABY(2),                // 婴儿
    CHILD(3),               // 儿童
    YOUNG(4),               // 少年
    YOUTH(5),               // 青年
    ADULT(6),               // 成年
    OLD(7),                 // 老年
    DEATH(8);               // 死亡

    int value;
    ManState(int value) {
        this.value = value;
    }

    static ManState valueOf(int value) throws Exception {
        switch (value) {
            case 1:
                return BIRTH;
            case 2:
                return BABY;
            case 3:
                return CHILD;
            case 4:
                return YOUNG;
            case 5:
                return YOUTH;
            case 6:
                return ADULT;
            case 7:
                return OLD;
            case 8:
                return DEATH;
            default:
                throw new Exception("异常状态值");
        }
    }

    static List<ManState> getStateList() {
        return new ArrayList<ManState>(Arrays.asList(ManState.values()));
    }
}

/**
 * 状态机模拟实现，用来管理一个 人类对象实例的 一生状态
 */
public class ManStateMachine implements EventHandler<ManEvent> {

    // 状态机实体对象
    private final StateMachine<ManState, ManEventType, ManEvent> stateMachine;

    // 状态机工厂实例
    // ManStateMachine, ManState, ManEventType, ManEvent
    private static final StateMachineFactory<
            ManStateMachine, ManState, ManEventType, ManEvent> stateMachineFactory;

    static {
        stateMachineFactory = new StateMachineFactory<>(ManState.BIRTH);
    }

    // addTransition = 注册 状态转移四元组
    // 1 状态机此时的状态，转移之前的状态
    // 2 状态之后得到的新状态
    // 3 发生的事件
    // 4 状态转移器
    // 某个状态机实体对象此时状态是 A，发生了事件 C，然后调用 D 来执行状态转移得到状态 B
    // 状态转移四元组：（A，B，C，D）
    {
        // 长大
        stateMachineFactory.addTransition(ManState.BIRTH, ManState.BABY,
ManEventType.GROWUP, new GrowupTransition());
        stateMachineFactory.addTransition(ManState.BABY, ManState.CHILD,
ManEventType.GROWUP, new GrowupTransition());
        stateMachineFactory.addTransition(ManState.CHILD, ManState.YOUNG,
ManEventType.GROWUP, new GrowupTransition());
        stateMachineFactory.addTransition(ManState.YOUNG, ManState.YOUTH,
ManEventType.GROWUP, new GrowupTransition());
        stateMachineFactory.addTransition(ManState.YOUTH, ManState.ADULT,
ManEventType.GROWUP, new GrowupTransition());
        stateMachineFactory.addTransition(ManState.ADULT, ManState.OLD,
ManEventType.GROWUP, new GrowupTransition());
        stateMachineFactory.addTransition(ManState.OLD, ManState.DEATH,
ManEventType.GROWUP, new GrowupTransition());

        // 返老还童，情况有点多，这里没有补充完整，如果报空指针异常，自己到这里来加情况
        stateMachineFactory.addTransition(ManState.ADULT, ManState.BABY,
ManEventType.REJUVENATION, new RejuvenationTransition());
        stateMachineFactory.addTransition(ManState.YOUTH, ManState.YOUNG,
ManEventType.REJUVENATION, new RejuvenationTransition());
        stateMachineFactory.addTransition(ManState.OLD, ManState.CHILD,
ManEventType.REJUVENATION, new RejuvenationTransition());
        stateMachineFactory.addTransition(ManState.YOUNG, ManState.BABY,
ManEventType.REJUVENATION, new RejuvenationTransition());
        stateMachineFactory.addTransition(ManState.YOUNG, ManState.CHILD,
ManEventType.REJUVENATION, new RejuvenationTransition());
        stateMachineFactory.addTransition(ManState.YOUNG, ManState.BIRTH,
ManEventType.REJUVENATION, new RejuvenationTransition());

        // 生病情况
        stateMachineFactory.addTransition(ManState.BABY, ManState.BABY,
ManEventType.SICK, new SickTransition());
        stateMachineFactory.addTransition(ManState.CHILD, ManState.CHILD,
ManEventType.SICK, new SickTransition());
        stateMachineFactory.addTransition(ManState.YOUNG, ManState.YOUNG,
ManEventType.SICK, new SickTransition());
        stateMachineFactory.addTransition(ManState.YOUTH, ManState.YOUTH,
ManEventType.SICK, new SickTransition());
        stateMachineFactory.addTransition(ManState.ADULT, ManState.ADULT,
ManEventType.SICK, new SickTransition());
        stateMachineFactory.addTransition(ManState.OLD, ManState.OLD,
ManEventType.SICK, new SickTransition());

        // 绝症
        stateMachineFactory.addTransition(ManState.BIRTH, ManState.DEATH,
ManEventType.INCURABLE_DISEASE,
                new IncurableDiseaseTransition()
        );
        stateMachineFactory.addTransition(ManState.BABY, ManState.DEATH,
ManEventType.INCURABLE_DISEASE,
                new IncurableDiseaseTransition()
        );
        stateMachineFactory.addTransition(ManState.CHILD, ManState.DEATH,
ManEventType.INCURABLE_DISEASE,
                new IncurableDiseaseTransition()
        );
        stateMachineFactory.addTransition(ManState.YOUNG, ManState.DEATH,
ManEventType.INCURABLE_DISEASE,
                new IncurableDiseaseTransition()
        );
        stateMachineFactory.addTransition(ManState.YOUTH, ManState.DEATH,
ManEventType.INCURABLE_DISEASE,
                new IncurableDiseaseTransition()
        );
        stateMachineFactory.addTransition(ManState.ADULT, ManState.DEATH,
ManEventType.INCURABLE_DISEASE,
                new IncurableDiseaseTransition()
        );
        stateMachineFactory.addTransition(ManState.OLD, ManState.DEATH,
ManEventType.INCURABLE_DISEASE,
                new IncurableDiseaseTransition()
        );
    }

    @Override
    public void handle(ManEvent event) {
        Class<? extends ManEventType> type = event.getType().getDeclaringClass();
        try {
            doTransition(event.getType(), event);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public ManStateMachine() {
        // 从状态机工厂创建具体的状态机
        this.stateMachine = stateMachineFactory.make(this);
    }

    public ManState doTransition(ManEventType eventType, ManEvent event) throws
Exception {
        // 由具体的状态机进行转换状态
        return stateMachine.doTransition(eventType, event);
    }

    public static void main(String[] args) throws Exception {
        // 创建一个状态机实体对象，内部有一个状态机成员变量
        ManStateMachine manStateMachine = new ManStateMachine();
        ManState manState = null;

        // A = ManState.BIRTH,  C = ManEventType.GROWUP -> B = BABY
        manState = manStateMachine.doTransition(ManEventType.GROWUP, new ManEvent(ManEventType.GROWUP));
        // B = CHILD
        manState = manStateMachine.doTransition(ManEventType.GROWUP, new ManEvent(ManEventType.GROWUP));

        // 改造之后的结果:  上下两句代码等价的，区别在于
        // 第一种方式:  自己做测试方便，直接调用方法
        // 第二种方式:  常规方式，正常方式，YARN 实现方式，当其他组件提交一个事件过来的时候，有可能就是让这个 EventHandler 来处理
        // 只不过这个 EventHandler 同时还是一个 StateMachine
        // manState = manStateMachine.doTransition(ManEventType.GROWUP, new ManEvent(ManEventType.GROWUP));
        // B = YOUNG
        manStateMachine.handle(new ManEvent(ManEventType.GROWUP));

        // B = YOUNG
        manState = manStateMachine.doTransition(ManEventType.SICK, new ManEvent(ManEventType.SICK));

        // A = YOUNG, C = REJUVENATION, B = BABY
        manState = manStateMachine.doTransition(ManEventType.REJUVENATION, new ManEvent(ManEventType.REJUVENATION));
        // A = BABY, C = INCURABLE_DISEASE, B = DEATH
        manState = manStateMachine.doTransition(ManEventType.INCURABLE_DISEASE, new ManEvent(ManEventType.INCURABLE_DISEASE));

        /**
         * 这个地方的测试，不是特别完美!
         *  改造:
         *  1 类似于 YARN , 把 ManStateMachine 做成 EventHandler
         *  2 在 ManStateMachine 的 handle 方法的实现中，调用
         *  manStateMachine.doTransition(ManEventType.GROWUP, new ManEvent(ManEventType.GROWUP));
         *  这种代码，来让状态机来执行响应处理
         *  3 最后的效果就变成了，如果其他的组件提交了事件给异步事件分发器 AsyncDispatcher,
         *  AsyncDispatcher 根据事件类型找到 EventHandler 可能就是找到一个 StateMachine
         */
    }
}

/**
 * 事件处理器抽象
 */
interface EventHandler<T extends Event> {
    void handle(T event);
```

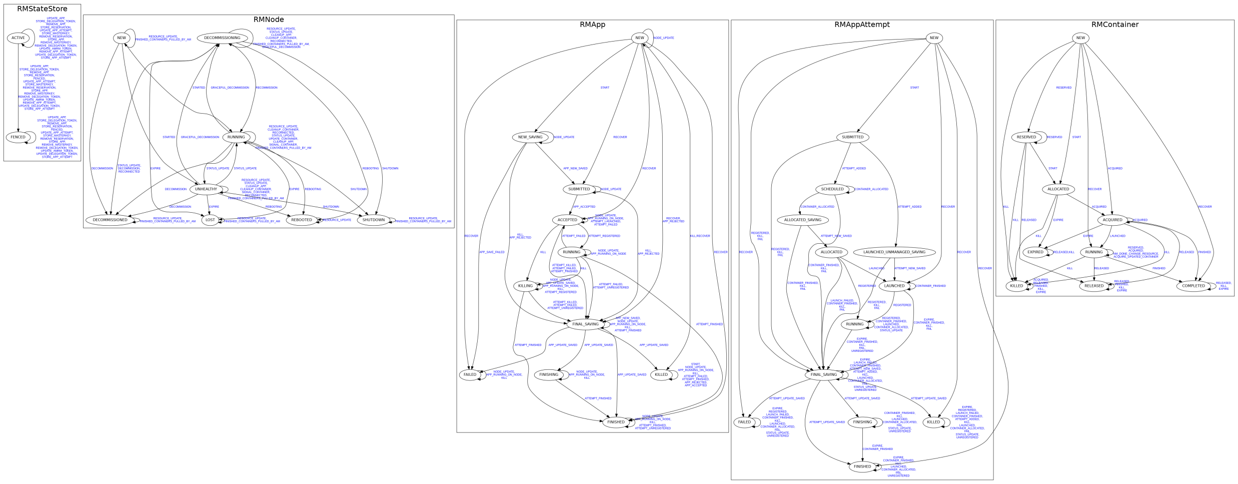
```
}

/**
 * 事件接口
 */
interface Event<TYPE extends Enum<TYPE>> {
    TYPE getType();

    long getTimestamp();

    String toString();
}

/**
 * 事件抽象
 */
abstract class AbstractEvent<TYPE extends Enum<TYPE>> implements Event<TYPE> {
    private final TYPE type;
    private final long timestamp;

    public AbstractEvent(TYPE type) {
        this.type = type;
        timestamp = -1L;
    }

    public AbstractEvent(TYPE type, long timestamp) {
        this.type = type;
        this.timestamp = timestamp;
    }

    @Override
    public long getTimestamp() {
        return timestamp;
    }

    @Override
    public TYPE getType() {
        return type;
    }

    @Override
    public String toString() {
        return "EventType: " + getType();
    }
}

/**
 * 事件实现类
 */
class ManEvent extends AbstractEvent<ManEventType> {
    public ManEvent(ManEventType type) {
        super(type);
```

```java
    }
}

/**
 * 状态机顶级抽象定义
 */
interface StateMachine<STATE extends Enum<STATE>, EVENTTYPE extends Enum<EVENTTYPE>, EVENT> {

    public STATE getCurrentState();

    public STATE doTransition(EVENTTYPE eventType, EVENT event) throws Exception;
}

/**
 * 转换器抽象定义
 */
interface Transition<OPERAND, STATE extends Enum<STATE>, EVENTTYPE extends Enum<EVENTTYPE>, EVENT> {
    STATE doTransition(OPERAND operand, STATE oldState, EVENT event, EVENTTYPE eventType);
}

/**
 * 状态机工厂， 四个泛型了解一下：
 * 1 OPERAND 状态机实体对象
 * 2 STATE extends Enum<STATE>  状态枚举类
 * 3 EVENTTYPE extends Enum<EVENTTYPE> 事件枚举类
 * 4 EVENT 事件
 */
class StateMachineFactory<OPERAND, STATE extends Enum<STATE>, EVENTTYPE extends Enum<EVENTTYPE>, EVENT> {

    // 初始状态
    private STATE defaultInitialState;

    // 状态转换注册表
    private Map<STATE, Map<EVENTTYPE, Transition<OPERAND, STATE, EVENTTYPE, EVENT>>> stateMachineTable;

    // 构造方法
    public StateMachineFactory(STATE defaultInitialState) {
        // 初始状态
        this.defaultInitialState = defaultInitialState;
        // 初始化状态转换表
        /**
         * Map<A, Map<B, C> >
         * A -> 枚举状态(比如 BIRTH)
         * B -> 枚举事件类型(比如 GROWUP)
         * C -> 状态转换动作
         */
        stateMachineTable = new HashMap<>();
    }

    // 添加一个状态转移四元组
    public void addTransition(
            STATE preState,      // 当前状态
            STATE postState,     // 结果状态
            EVENTTYPE eventType, // 事件类型
            Transition<OPERAND, STATE, EVENTTYPE, EVENT> hook // 转换动作 (事件处理)
    ) {
        // 检查该状态的转移表是否存在
        Map<EVENTTYPE, Transition<OPERAND, STATE, EVENTTYPE, EVENT>>
eventTypeTransitionMap =
                stateMachineTable.computeIfAbsent(
                        preState, k -> new HashMap<>());
        eventTypeTransitionMap.put(eventType, hook);
    }

    public StateMachine<STATE, EVENTTYPE, EVENT> make(OPERAND operand) {
        // 封装具体的状态机
        return new InternalStateMachine(operand, defaultInitialState,
stateMachineTable);
    }

    // StateMachine 的唯一实现类
    class InternalStateMachine implements StateMachine<STATE, EVENTTYPE, EVENT> {

        private final OPERAND operand;
        private STATE currentState;
        private Map<STATE, Map<EVENTTYPE, Transition<OPERAND, STATE, EVENTTYPE,
EVENT>>> stateMachineTable;

        InternalStateMachine(OPERAND operand, STATE initialState,
                             Map<STATE, Map<EVENTTYPE, Transition<OPERAND, STATE,
EVENTTYPE, EVENT>>> stateMachineTable) {
            this.operand = operand;
            this.currentState = initialState;
            this.stateMachineTable = stateMachineTable;
        }

        @Override
        public STATE getCurrentState() {
            return currentState;
        }

        @Override
        public STATE doTransition(EVENTTYPE eventType, EVENT event) throws Exception
{
            // 1 首先到注册表中进行查询 找到 Transition
            Map<EVENTTYPE, Transition<OPERAND, STATE, EVENTTYPE, EVENT>>
eventTypeTransitionMap =
                    // 从当前状态获取注册状态转换的所有情况
                    stateMachineTable.get(getCurrentState());
            // 根据事件类型获取对应的状态转换器
            Transition<OPERAND, STATE, EVENTTYPE, EVENT> transition =
eventTypeTransitionMap.get(eventType);

            // 2 调用 Transition 的 doTransition 方法来执行转换
            STATE state = transition.doTransition(operand, getCurrentState(), event,
eventType);

            // 3 返回新的状态
            currentState = state;
            return state;
        }
    }
}

/**
 * 正常长大
 */
class GrowupTransition implements Transition<ManStateMachine, ManState,
ManEventType, ManEvent> {
    @Override
    public ManState doTransition(ManStateMachine manStateMachine, ManState oldState,
ManEvent manEvent, ManEventType eventType) {
        ManState targetManState = null;
        int newValue = oldState.value + 1;
        try {
            targetManState = ManState.valueOf(newValue);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(
                "EventType: " + eventType + ", Old State: " + oldState + ", New
State: " + targetManState + ", Event: " + manEvent);
        return targetManState;
    }
}

/**
 * 生病了，治好
 */
class SickTransition implements Transition<ManStateMachine, ManState, ManEventType,
ManEvent> {
    @Override
    public ManState doTransition(ManStateMachine manStateMachine, ManState oldState,
ManEvent manEvent, ManEventType eventType) {
        System.out.println(
                "EventType: " + eventType + ", Old State: " + oldState + ", New
State: " + oldState + ", Event: " + manEvent);
        return oldState;
    }
}
```

```java
/**
 * 返老还童
 */
class RejuvenationTransition implements Transition<ManStateMachine, ManState,
ManEventType, ManEvent> {
    @Override
    public ManState doTransition(ManStateMachine manStateMachine, ManState oldState,
ManEvent manEvent, ManEventType eventType) {
        // 随机挑一个比当前状态小的，非出生状态
        List<ManState> stateList = ManState.getStateList();
        Random random = new Random();
        int i = random.nextInt(stateList.size());
        ManState targetState = stateList.get(i);
        while (targetState.value >= oldState.value && targetState.value != 1) {
            i = random.nextInt(stateList.size());
            targetState = stateList.get(i);
        }
        System.out.println(
                "EventType: " + eventType + ", Old State: " + oldState + ", New
State: " + targetState + ", Event: " + manEvent);
        return targetState;
    }
}

/**
 * 绝症死亡
 */
class IncurableDiseaseTransition implements Transition<ManStateMachine, ManState,
ManEventType, ManEvent> {
    @Override
    public ManState doTransition(ManStateMachine manStateMachine, ManState oldState,
ManEvent manEvent, ManEventType eventType) {
        System.out.println(
                "EventType: " + eventType + ", Old State: " + oldState + ", New
State: " + ManState.DEATH + ", Event: " + manEvent);
        return ManState.DEATH;
    }
}
```

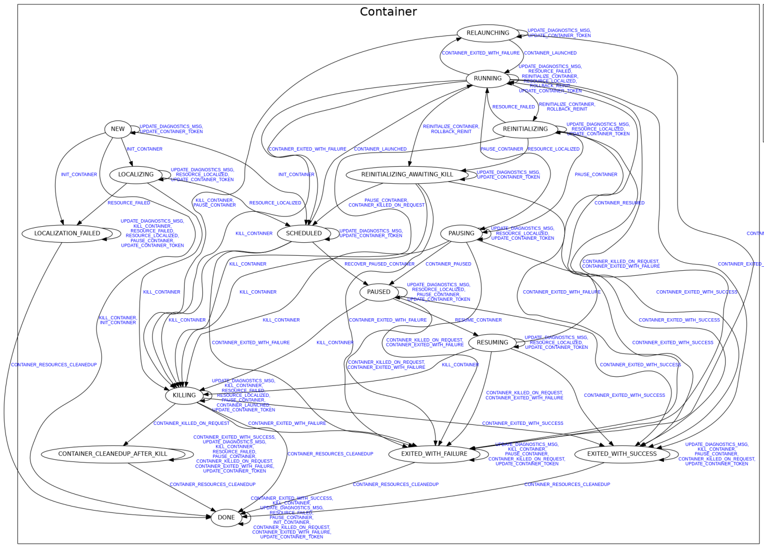
5.3 YARN 角色状态机类别

5.3.1 ResourceManager 状态机



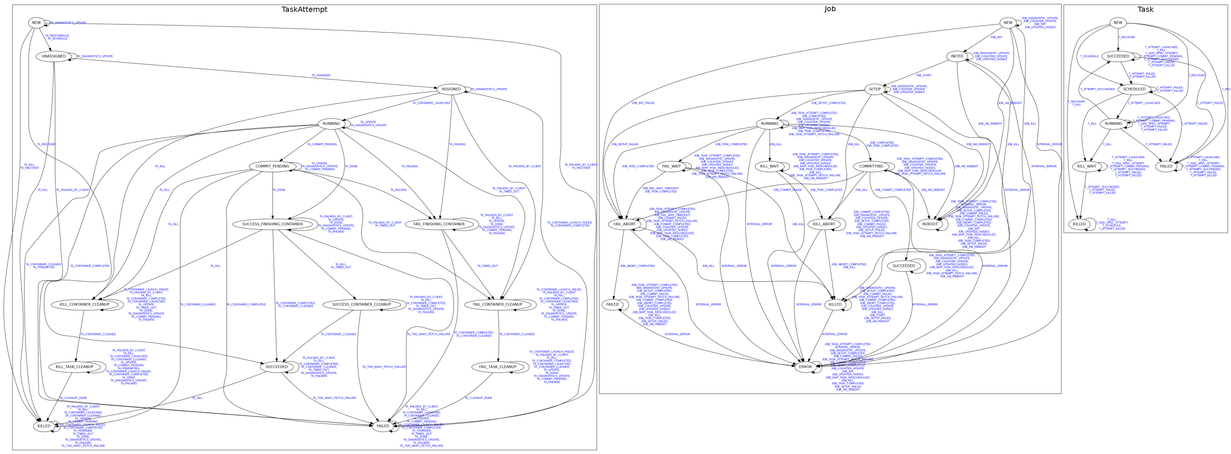
ResourceManager

5.3.2 NodeManager 状态机



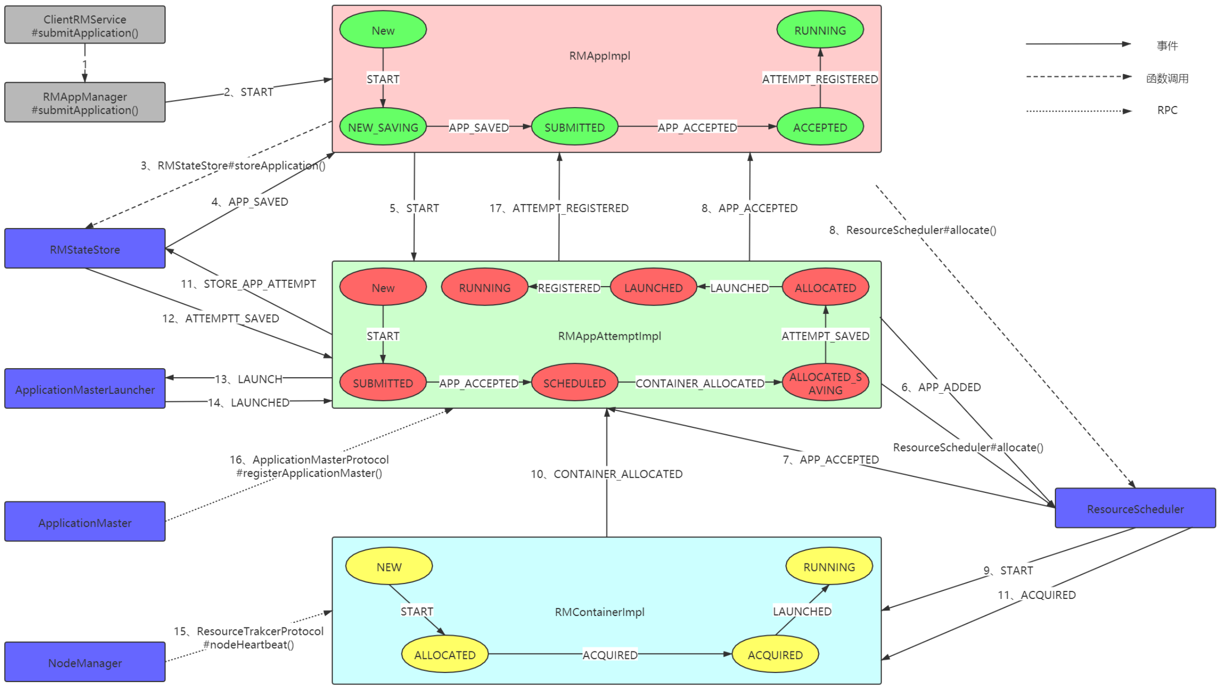
NodeManager

5.3.3 MapReduce 状态机



MapReduce

5.3.4 ApplicationMaster 状态机


ClientRMService
#submitApplication()
1
RMAppManager
#submitApplication()
2、START
RMAppImpl
New
START
NEW_SAVING
APP_SAVED
SUBMITTED
APP_ACCEPTED
ACCEPTED
ATTEMPT_REGISTERED
RUNNING
3、RMStateStore#storeApplication()
4、APP_SAVED
RMStateStore
5、START
17、ATTEMPT_REGISTERED
8、APP_ACCEPTED
8、ResourceScheduler#allocate()
RMAppAttemptImpl
New
RUNNING
REGISTERED
LAUNCHED
LAUNCHED
ALLOCATED
START
ATTEMPT_SAVED
SUBMITTED
APP_ACCEPTED
SCHEDULED
CONTAINER_ALLOCATED
ALLOCATED_S
AVING
11、STORE_APP_ATTEMPT
12、ATTEMPTT_SAVED
ApplicationMasterLauncher
13、LAUNCH
14、LAUNCHED
6、APP_ADDED
ResourceScheduler#allocate()
7、APP_ACCEPTED
ResourceScheduler
16、ApplicationMasterProtocol
#registerApplicationMaster()
ApplicationMaster
10、CONTAINER_ALLOCATED
RMContainerImpl
NEW
START
ALLOCATED
ACQUIRED
ACQUIRED
LAUNCHED
RUNNING
9、START
11、ACQUIRED
15、ResourceTrakcerProtocol
#nodeHeartbeat()
NodeManager
事件
函数调用
RPC