NCR 学习文档

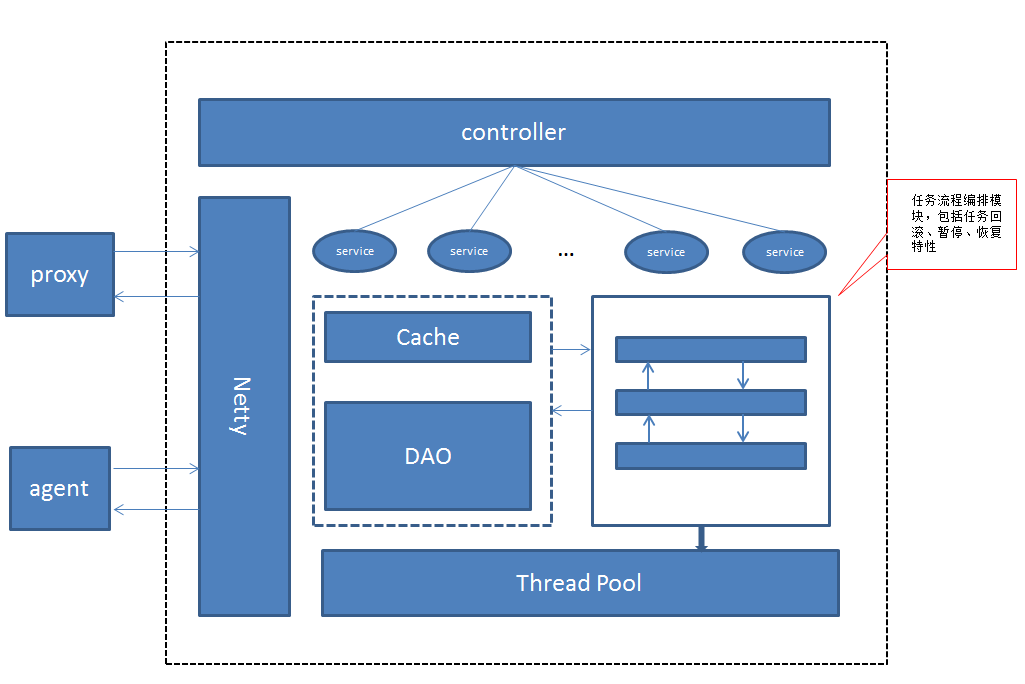
# NCR架构

## 1.1

参见设计文档。

# NCR Manager代码学习

## NCR Manager代码结构



## 结构说明

NCR Manager模块采用spring MVC结构，controller层通过HTTP URL中action参数匹配到各个action处理类，各个action处理类都继承AbstractNCRAction基类，在AbstractNCRAction实现ip地址白名单过滤和用户签名认证。

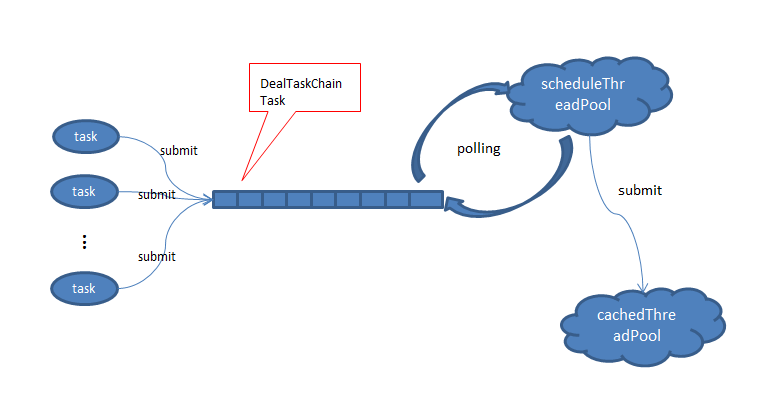
ORM框架采用开源的mybatis，在DAO访问时设计使用ConcurrentHashMap缓存数据。

Agent、proxy通信采用netty异步事件驱动的NIO框架，采用自己设计的二进制通信协议，报头大小固定，报体大小可变。Agent发送心跳消息，manager根据心跳信息更新集群节点状态。Proxy发送三种消息，心跳信息、节点失败、节点恢复消息。

任务流程编排模块提供任务执行、回滚、暂停、恢复等特性。任务处理的各个entry通过双向链表（责任链模式）组成完成的处理流程。每个entry提供执行handle方法和异常回滚undo方法，正常流程执行调用entry的handle方法再调用下一个entry，当发生异常需要回滚调用undo方法，再调用上一个entry的undo方法。强制回滚则从最后一个entry调用undo方法，task会记录当前处理状态，不用担心没有处理的entry会执行回滚动作。

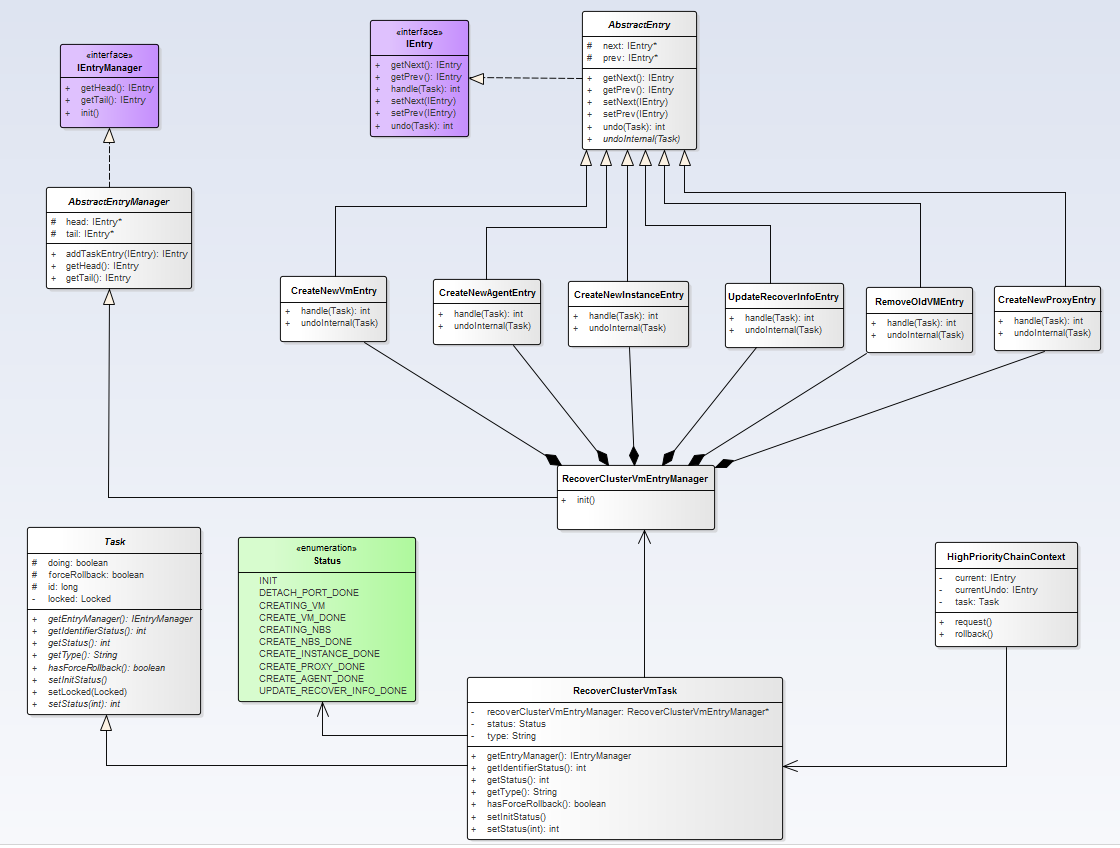
线程池的使用，NCR提供四个线程池，无界线程池cachedThreadPool、定时任务线程池scheduleThreadPool、两个单线程池singleRebuildThreadPool和singleNlbThreadPool。

cachedThreadPool执行任务编排流程（重建集群、恢复集群等任务），scheduleThreadPool执行定时任务如删除过期资源、启动实例检查、发送报警信息等。其中提交任务到cachedThreadPool线程池内的流程编排任务是先提交到DealTaskChainTask，DealTaskChainTask内部阻塞队列，scheduleThreadPool会定时扫描该队列，将队列中的task提交到chacheThreadPool内执行。



# NCR Manager代码分析

## 任务编排流程分析



上图以recoverClusterVM流程为例，总共分成createNewVm、createNewInstance、createNewProxy、createNewAgent、updateRecoverInfo和RemoveOldVm六个步骤。从上图看出这六个步骤实现IEntry接口组成双向链表并由RecoverClusterVmEntryManager类管理。

### 3.1.1 recoverClusterVM流程如何执行？

当recoverClusterVMTask创建时会设置该task处于初始化状态非锁定，并将该task写入数据库表中和加入缓存。然后该task加入DealTaskChainTask的阻塞队列中，由定时任务轮询取该队列中的task，此时会设置该task（setDoing）正在执行。轮询线程池会将该task最终提交到cacheThreadPool线程池中执行。

执行由HighPriorityChainContext中的request方法触发，从task中获取IEntryManager中head节点，开始判断该task处理非锁定状态时会依次调用各个节点的handle方法，同时会更新该task的状态记录处于执行步骤。

当执行完最后一个步骤，又该步骤负责将该任务删除（数据库和缓存都删除）。

### 3.1.2 任务如何暂停？

暂停任务要将任务锁状态设置成Locked.***STOPPED***或*Locked****.LOCKED*** 。每个流程步骤执行前都会先检查该任务的锁状态，只有为Locked.***UNLOCKED***才会继续执行。

### 3.1.3 任务如何恢复？

恢复任务只需将任务锁状态设置成Locked.***UNLOCKED***即可。轮询线程会定时扫描并执行BlockingQueue中的任务，当BlockingQueue为空时会检查系统缓存中的所有任务将恢复的任务重新提交到cacheThreadPool中执行。

### 3.1.2 任务如何回滚？

任务回滚分成两种：

1. 异常回滚

当执行流程中发生异常，调用当前IEntry的undo方法并逐一调用上一步骤的该方法

1. 强制回滚

强制回滚是发生在该任务被暂停（Locked.***LOCKED***）,首先设置该任务的ForceRollback为true,而后将任务再次提交到线程池，由HighPriorityChainContext调用其rollback方法，获取尾节点逆向调用undo方法，并在首部节点执行完删除该任务。

## agent、proxy通信模块分析

组件通信采用netty开源的异步NIO框架实现，netty本身是采用Doug Lea提出的Reactor模式实现，Java高性能的NIO框架都是基于此模式实现。

Netty对消息处理采用拦截链模式，启动时向ChannelPipeline注册handler用于处理接受消息（实现ChannelUpstreamHandler接口）和发送消息（实现ChannelDownstreamHandler接口）。典型应用就对原始字节的编解码处理或消息的日志记录。

Netty采用拦截链模式优点就是屏蔽底层原始字节流消息，上层直接处理业务对象而非原始消息，而NCR Manager中违背这一设计原则，使得上层业务处理和编解码耦合在一起。

如何优化？

1. NettyPacket不包含原始字节流，NettyDecoder解码时将原始字节流组装成完整业务对象再往Handler层抛出。
2. 增加NettyEncoder类对响应的业务对象进行编码，可以继承org.jboss.netty.handler.codec.oneone.OneToOneEncoder类，上层直接写回响应对象Channel.write(respPacket),NettyEncoder类会拦截到downStream拿到respPacket将其编成字节流。

通信协议设计

与agent、proxy通信，NCR自己设计二进制协议，采用固定报头+可变报体，第一个4bytes指定消息大小（包括报头和报体）。

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 类型 | 长度 | 说明 |
| 报文长度 | uint32\_t | 4bytes | 必填，整个报文长度，包括报头、报体 |
| 报文版本 | uint32\_t | 4bytes | 必填 |
| 报文类型 | uint32\_t | 4bytes | 必填 |
| 节点编号 | uint32\_t | 4bytes | 必填 |

如果对高性能有需求为了节省网络流量，可以设计可变报头，报头中有些字段在第一个消息中携带后面都不会变，或者说字段值很少会变化，我们可以保存上次报头内容，在字段变化时更新保存的报头内容，字段不变时报头中就不再携带，默认取上次报头字段内容。NCR中报头不大，网络IO也不是瓶颈，也就没有必要改进。