yarn 学习心得　yarn 上的程序开发

1 概况

YARN 是 Hadoop 系统上的资源统一管理平台，其主要作用是实现集群资源的统一管理和调度。　　YARN 是一个高速发展中的资源管理与调度平台，目前还不是很完善，当前只支持 CPU 和内存的分配。作为资源调度器，　　YARN 支持如下几个资源调度语义：获取指定节点的特定资源量，如　　node1 上 4 个虚拟 CPU 核，1GB 内存 (YARN 上的资源使用容器包装)；

获取指定机架上的特定资源量；

支持资源黑名单 (添加 /删除)；

要求某些应用归还指定的资源，通常用于抢占场景。

YARN 目前不支持的调度语义有　(或者说支持得不是很好)：

获取任意节点上的特定资源量；

获取任意机架上的特定资源量；

获取一组或几组符合特定规则的资源量；

细粒度资源分配，如获取主频大于 2.4G 的 CPU 等；

动态调整资源容器容量 (对长应用比较重要)。

YARN 上的应用按其运行的生命周期长短，可以分为长应用和短应用，短应用通常是分析作业，作业从提交到完成，所耗的时间是有限的， 作业完成后， 其占用的资源就会被释放，归还给 YARN 进行再次分配。 长应用通常是一些服务， 应用启动后除非意外或人为终止，将一直运行下去。长应用通常长期占用集群上的一些资源，且运行期间对资源的需求也时常变化， 因此， 动态调整资源对长应用来说比较重要。 目前，

YARN 对长应用的支持还不是很好，从社区讨论来看，受 hortonworks 的 Hoya 项目推动， YARN 在 2.20 版本后加强了对长应用的支持。

2 应用开发

2.1 概述

YARN 的应用开发主要过程如下： 图 2.1 YARN 应用开发流程

YARN 主要由 ResourceManager 和 NodeManager 组成， ResourceManager 负责资源的管理与分配，NodeManager 则负责具体资源的隔离。 YARN 中，资源使用容器进行封装。 用户在 YARN 上开发应用时， 需要实现如下三个模块： Application Client: 应用客户端用于将应用提交到 YARN 上，使应用运行在 YARN 上，同时，监控应用的运行状态，控制应用的运行；

Application Master: AM　负责整个应用的运行控制，包括向 YARN 注册应用、申请资源、启动容器等，应用的实际工作在容器中进行；

Application Worker:　应用的实际工作，并不是所有的应用都需要编写 worker。NodeManager 启动 AM 发送过来的容器，容器内部封装了该应用 worker 运行所需的资源和启动命令。

实现上述模块，涉及如下 2 个 RPC 协议：

ApplicationClientProtocol: Client-RM 之间的协议，主要用于应用的提交；

ApplicationMasterProtocol: AM-RM 之间的协议， AM 通过该协议向 RM 注册并申请资源；

ContainerManagementProtocol: AM-NM 之间的协议， AM 通过该协议控制 NM 启动容器。

上述协议的定义在 hadoop-yarn-api 工程中。

从业务的角度看，一个应用需要分两部分进行开发，一个是接入 YARN 平台，实现上述 3 个协议，通过 YARN 实现对集群资源的访问和利用；另一个是业务功能的实现，这个与 YARN 本身没有太大关系。下面主要阐述如何将一个应用接入 YARN 平台。

2.2 客户端开发

客户端开发流程如图 2.2 所示：图 2.2 YARN 应用程序客户端开发

从上图可以看出，客户端的主要作用是提交 (部署)应用和监控应用运行两个部分。

2.2.1 提交应用

提交应用涉及 ApplicationClientProtocol 协议中的两个方法：

GetNewApplicationResponse

getNewApplication(GetNewApplicationRequest request)

SubmitApplicationResponse

submitApplication(SubmitApplicationRequest request)

具体步骤如下： 客户端通过 getNewApplication 方法从 RM 上获取应用 ID ；

客户端将应用相关的运行配置封装到 ApplicationSubmissionContext 中，通过 submitApplication 方法将应用提交到 RM 上；

RM 根据 ApplicationSubmissionContext 上封装的内容启动 AM ；

客户端通过 AM 或 RM 获取应用的运行状态， 并控制应用的运行过程。

通过 getNewApplication 可从 RM 上获取全局唯一的应用 ID 和最大可申请的资源量 (内存和虚拟 CPU 核数)，如下所示：

图 2.3 getNewApplication 方法的输入输出

在获取应用程序 ID 后，客户端封装应用相关的配置到 ApplicationSubmissionContext 中，通过 submitApplication 方法提交到 RM 上。图 2.4 submitApplication 方法的输入输出

ApplicationSubmissionContext 主要包括如下几个部分：

applicationId: 通过 getNewApplication 获取的应用 ID ；

applicationName: 应用名称，将显示在 YARN 的 web 界面上；

applicationType: 应用类型，默认为 ” YARN ”；

priority: 应用优先级，数值越小，优先级越高；

queue: 应用所属队列，不同应用可以属于不同的队列，使用不同的调度算法；

unmanagedAM: 布尔类型，表示 AM 是否由客户端启动 (AM 既可以运行在 YARN 平台之上，也可以运行在 YARN 平台之外。运行在 YARN 平台之上的 AM 通过 RM 启动，其运行所需的资源受 YARN 控制)；

cancelTokensWhenComplete: 应用完成后，是否取消安全令牌；

maxAppAttempts: AM 启动失败后，最大的尝试重启次数；

resource: 启动 AM 所需的资源 (虚拟 CPU 数/内存), 虚拟 CPU 核数是一个归一化的值；

amContainerSpec: 启动 AM 容器的上下文，主要包括如下内容：

tokens: AM 所持有的安全令牌；

serviceData: 应用私有的数据，是一个 Map，键为数据名，值为数据的二进制块；

environment: AM 使用的环境变量；

commands: 启动 AM 的命令列表；

applicationACLs ：应程序访问控制列表；

localResource: AM 启动需要的本地资源列表，主要是一些外

部文件、压缩包等。

监控应用运行状态

应用监控涉及 ApplicationClientProtocol 协议中的如下几个方法：

//强制杀死一个应用

KillApplicationResponse

forceKillApplication(KillApplicationRequest request)

//获取应用状态，如进度等

GetApplicationReportResponse

getApplicationReport(GetApplicationReportRequest request)

//获取集群度量

GetClusterMetricsResponse

getClusterMetrics(GetClusterMetricsRequest request)

//获取符合条件的应用的状态　　(列表)

GetApplicationsResponse

getApplications(GetApplicationsRequest request)

//获取集群中各个节点的状态

GetClusterNodesResponse

getClusterNodes(GetClusterNodesRequest request)

//获取 RM 中的队列信息

GetQueueInfoResponse getQueueInfo(GetQueueInfoRequest

request)

//获取当前用户的访问控制信息

GetQueueUserAclsInfoResponse

getQueueUserAcls(GetQueueUserAclsInfoRequest request)

//获取委托令牌，使得容器可以使用这些令牌与服务通信

GetDelegationTokenResponse

getDelegationToken(GetDelegationTokenRequest request)

//更新已存在的委托令牌

RenewDelegationTokenResponse

renewDelegationToken(RenewDelegationTokenRequest

request)

//需要已存在的委托令牌

CancelDelegationTokenResponse

cancelDelegationToken(CancelDelegationTokenRequest

request)

客户端既可以从 RM 上获取应用的信息, 也可以通过 AM 获取。通常为了减少 RM 的压力，使用从 AM 获取应用运行状态的方式。 客户端与 AM 之间的通信使用应用内部的私有协议，与 YARN 无关。

2.3 AM 开发

AM 的主要功能是按照业务需求，从 RM 处申请资源，并利用这些资源完成业务逻辑。因此，AM 既需要与 RM 通信，又需要与 NM 通信。这里涉及两个协议，分别是 AM-RM 协议 (ApplicationMasterProtocol) 和 AM-NM 协议 (ContainerManagementProtocol)，如图 2.5 所示：图 2.5 AM-YARN 接口协议

2.3.1 AM-RM 协议

AM-RM 之间使用 ApplicationMasterProtocol 协议进行通信，该协议提供如下几个方法：

//向 RM 注册 AM

RegisterApplicationMasterResponse registerApplicationMaster(RegisterApplicationMasterRequest request)

//告知 RM，应用已经结束

FinishApplicationMasterResponse

finishApplicationMaster(FinishApplicationMasterRequest

request)

//向 RM 申请/归还资源，维持心跳

AllocateResponse allocate(AllocateRequest request)

客户端向 RM 提交应用后， RM 会根据提交的信息，分配一定的资源来启动 AM ， AM 启动后调用 ApplicationMasterProtocol 协议的 registerApplicationMaster 方法主动向 RM 注册。完成注册后， AM 通过 ApplicationMasterProtocol 协议的 allocate 方法向 RM 申请运行任务的资源，获取资源后，通过 ContainerManagementProtocol 在 NM 上启动资源容器，完成任务。应用完成后， AM 通过 ApplicationMasterProtocol 协议的 finishApplicationMaster 方法向 RM 汇报应用的最终状态，并注销 AM 。主要过程如图 2.6 所示：图 2.6 AM-RM 交互流程

需要注意的是， ApplicationMasterProtocol#allocate() 方法还兼顾维持 AM-RM 心跳的作用，因此，即便应用运行

过程中有一段时间无需申请任何资源，　　AM 都需要周期性的调用相应该方法，以避免触发　　RM 的容错机制。下面具体看一下每一步所传递的信息：

1 AM 向 RM 注册

AM 启动后会主动调用　registerApplicationMaster 方法向 RM 注册，注册信息中包括该　　AM 所在节点和开放的　RPC 服务端口，以及一个应用状态跟踪　　Web 接口 (将在 RM 的 Web 页面上显示)。RM 向 AM 返回一个对象，　里面包含了应用最大可申请的单个容器容量、应用访控制列表和一个用于与客户端通信的安全令牌。图　2.7 registerApplicationMaster 方法输入输出

2 AM 向 RM 申请资源

AM 通过 allocate 方法向 RM 申请或释放资源。 AM 向 RM 发送的信息被封装在　AllocateRequest 里，包括如下内容： responseId: 相应 ID，用于区分重复的响应　；

askList ： AM 向 RM 申请的资源列表，是一个

List<ResourceRequest> 对象，其中 ResourceRequest中一个资源请求的详细参数，包括优先级、容器个数、单个容器容量和分配策略 (是否放宽本地化约束)；

releaseList: AM 主动释放的资源容器列表 ；

resourceBlacklistRequest: 要添加或删除的资源黑名单；

progress:应用的运行进度。图 2.8 AllocateRequest

RM 接受到 AM 的请求后，扫描其上的资源镜像，按照调度算法分配全部或部分申请的资源给 AM ，返回一个 AllocateResponse 对象， 里面内容包括： responseId: 相应 ID ，用于区分重复的响应 ；

numClusterNodes: 集群规模大小；

updatedNodes: 状态被更新过的所有节点列表， 每个节点的状态更新信息被分装在 NodeReport 对象中 ,包括以下内容 ：

nodeId: 节点唯一标识 ；

httpAddress: 节点的 Web 页面地址；

rackName: 节点所在机架名；

numContainers: 节点上当前运行的容器个数；

nodeState: 节点运行状态，是一个枚举类型；

used: 节点上已经使用的资源量；

capability:　节点总的资源量；

healthReport:　节点的健康诊断信息；

lastHealthReportTime:　最新的节点的健康诊断时戳　；

availableResources: 集群的资源净空量；

AMCommand: RM　给 AM 发送的控制命令，　包括重连和关闭；

NMTokens: AM　与 NM 之间的通信令牌　；

allocatedContainers: RM 新分配给　AM 的资源容器列表　,这些资源被封装在资源容器　(Container) 中：

id: 容器 ID，每个容器都具有全局唯一的 ID ；

priority: 优先级；

nodeId: 容器所在节点的 ID ；

nodeHttpAddress: 节点的 Web 页面地址；

containerToken: 容器的安全令牌；

resource: 该容器所持有的资源，包括内存和 CPU。

completedContainersStatuses: 已完成的容器状态列表；

preemptionMessage: 资源抢占信息，包括两部分，强制收回

部分和可自主调配部分　：

strictContract:　强制收回部分，　AM 必须释放的容器列表；

preemptionContract:　可自主调配的部分，该部分包含了两个内容，分别是抢占资源需求和可抢占的资源列表　　,AM 需要从可抢占的资源列表中选出部分资源进行释放，以满足抢占资源需求；

图 2.9 AllocateResponse

3 AM 通知 RM 应用已结束

在应用完成后，　AM 通知 RM 应用结束的消息，　同时向 RM 提供应用的最终状态　(成功 /失败等)、一些失败时的诊断信息和应用跟踪地址，　RM 收到通知后注销相应的　AM，并将注销结果发送给　AM，AM 收到注销成功的消息后，退出进程。 AM 通过调用 ApplicationMasterProtocol#finishApplicationMaster　方法通知

RM，该方法的输入输出如下所示　　:图 2.10

finishApplicationMaster　方法的 I/O

2.3.2 AM-NM　协议

AM 通过 ContainerManagementProtocol 协议与 NM 交互，包括　3 个方面的功能：启动容器、查询容器状态、停止容器，分别对应协议中的三个方法　　：

//启动容器

StartContainersResponse

startContainers(StartContainersRequest request)

//查询容器状态

GetContainerStatusesResponse

getContainerStatuses(GetContainerStatusesRequest request)

//停止容器

StopContainersResponse stopContainers(StopContainersRequest request)、

AM-NM 交互过程如图 2.11 所示 :图 2.11 AM-NM

1 AM 在 NM 上启动容器

AM 通过 ContainerManagementProtocol# startContainers()方法启动一个 NM 上的容器, AM 通过该接口向 NM 提供启动容器的必要配置，包括分配到的资源、安全令牌、启动容器的环境变量和命令等，这些信息都被封装在 StartContainersRequest 中。 NM 收到请求后，会启动相应的容器，并返回启动成功的容器列表和失败的容器列表，同时还返回其上相应的辅助服务元数据。 startContainers 方法的输入输出如图 2.12 所示 :图 2.12 startContainers 的 I/O

2 AM 查询 NM 上的容器运行状态

在应用运行期间， AM 需要实时掌握各个 Container 的运行状态，以便及时响应一些异常，如容器运行失败等。

AM 通过 ContainerManagementProtocol# getContainerStatuses

()方法获取各个容器的运行状态，其输入输出如下图所示　　:图

2.13 getContainerStatuses I/O

3 AM 停止 NM 上的容器

当一个容器运行完成后，分配给它的资源需要被回收。AM 通过 ContainerManagementProtocol#stopContainers()方法停止 NM 上的容器 ,释放相关资源， 然后通过 AM-RM 协议，将释放的资源上报给 RM，RM 完成最终的资源回收。stopContainers 的输入输出如下图所示： 图 2.14 stopContainers I/O

2.4 使用 YARN 编程库开发应用

如 2.3 节所述， YARN 上的应用开发分为平台接入和业务开发两个部分， 其中平台接入就是实现上述三个 RPC 协议。直接实现上述协议的开发难度较高，需要处理很多细节和性能问题，如系统并发等。为此， YARN 提供了一套应用程序编程库来简化应用的开发过程，该编程库是基于事件驱动机制的， 利用了 YARN 内部的服务库、 事件库和状态机库，分为三个部分，与上述三个协议一一对应。

2.4.1 YARN 基础库

1 服务库

YARN 中普遍采用基于服务的对象管理模型，将一些生命周期较长的对应服务化，YARN 提供一套抽象的接口对服务进行了统一描述，该服务具有如下特点 :具有标准状态，所有服务都具有 4 个状态， NOTINITED 、INITED 、STARTED、 STOPPED；

状态驱动，服务状态变化将触发一些动作，使其转变成另一种状态；

服务嵌套，一个服务可以由其他服务组合嵌套而来。

YARN 服务库如下所示 ：

图 2.15 YARN 服务库

2 事件库

YARN 中大量采用了基于事件驱动的并发模型，该模型由事件、异步调度器和事件处理器三个模块组成。处理请求被抽象为事件，放入异步调度器的事件队列中，调度线程从事件队列中取出事件分发给不同的事件处理器，事件处理器处理事件，产生新的事件放入事件队列，如此循环，直到处理完成 (完成事件)。图 2.16 YARN 事件库

3 状态机库

YARN 中使用 {转换前状态、转换后状态、事件、回调函数 } 四元组来表示一个状态变换，一个或多个事件的到来，触发绑定在对象上状态转移函数，使对象的状态发生变化。状态机使得事件处理变得简单可控。图 2.17 状态机库

总的来说， YARN 中的服务由一个或多个含有有

限状态机的事件处理系统组成，　　总体框架如下图所示。　图 2.18　YARN 服务通用模型

2.4.2 YARN 应用客户端库 (CLIENT-RM 编程库)

YARN 的 Client-RM 编程库位于 org.apache.hadoop.yarn.client.YarnClient(Hadoop-yarn-api 项目)，该库实现了通用的 ApplicationClientProtocol 协议，提供了重试机制。 用户利用该库可以快速开发 YARN 应用的客户端程序，而不需要关心 RPC 等底层接口。如图所示 :图 2.19 YarnClient

用户开发自己的应用客户端时，只要设置好 ApplicationSubmissionContext 对象，调用 YarnClient 的相关接口，即可实现应用的提交。

2.4.3 AM-RM 编程库

AM-RM 编程库主要简化了 AM 向 RM 申请资源过程的开发。 YARN 提供了两套 AM-RM 编程库，分别为阻塞

式和非阻塞式模式。如图　2.20 所示。 图 2.20 AM-RM　编程库

其中，　AMRMClient　是阻塞式的，实现了 ApplicationMasterProtocol　协议，用户调用该类的相应接口，可实现 AM 与 RM 的通信。而　AMRMClientAsync　是 AMRMClient　的非阻塞式封装，所有响应通过回调函数的形式返回给用户，用户实现自己的　AM 时，只需要实现 AMRMClientAsync　的 CallbackHandler 即可。如图 2.21 所示 :

图 2.21 AM-RM　编程库

2.4.4 NM 编程库

NM 编程库对　AM 和 RM 与 NM 之间的交互进行了封装，同样有阻塞式和非阻塞式两种封装　(AM 与 NM 和 RM 与 NM 的交互逻辑相似　),如图 2.22 所示。图　2.22 NM 编程库

同样的，　对于异步编程库　NMClientAsync ，用户只需要在自己的　AM 上实现相应的回调函数，就可以控制　NM 上 Container 的启动 /停止和状态监控了。如图　2.23 所示。图 2.23 NM 编程库

2.5 总结

本文介绍了 YARN 平台应用开发的基本流程，总结如下： YARN 平台应用开发主要有两个工作： YARN 平台接入和业务逻辑实现；

YARN 平台应用开发主要需要开发三个组件：客户端、 AM 和 worker ；

YARN 平台接入主要涉及三个协议，分别为 ApplicationClientProtocol 、ApplicationMasterProtocol 和 ContainerManagementProtocol，其中，客户端通过 ApplicationClientProtocol 协议与 RM 交互，提交 (部署)应用并监控应用的运行； AM 通过 ApplicationMasterProtocol 协议维持 AM-RM 心跳，并向 RM 申请 YARN 上的资源； AM 通过 ContainerManagementProtocol 协议控制 NM 启动、停止申请到的容器， 并监控容器的运行状态。 容器是 YARN 对资源的封装，应用的 Worker 在容器中运行，只能使用容器中的

资源，从而实现资源隔离；

YARN 提供了 Client-RM 编程库、 AM-RM 编程库和 NM 编程库，从而简化了 YARN 上的应用开发 (当然还不是很简单)，需要注意的是，该编程库的接口还不是很稳定，以后还有可能发生变化 (hadoop2.0 与 hadoop2.2.0 中 YARN 的这些编程库不兼容)。

总得来说， YARN 是一个资源管理平台， 并不涉及业务逻辑，具体的业务逻辑需要用户自己去实现。 YARN 的核心作用就是分配资源、保证资源隔离。