



首部Hadoop YARN专著，资深Hadoop技术专家根据最新版本撰写，ChinaHadoop和51CTO等专业技术社区联袂推荐！

从应用角度系统讲解YARN的基本库和组件用法、应用程序设计方法、YARN上流行的各种计算框架，以及多个类YARN的开源资源管理系统。

从源代码角度深入分析YARN的设计理念与基本架构、各个组件的实现原理，以及各种计算框架的实现细节。



Hadoop Internals: in-depth study of YARN

Hadoop技术内幕

深入解析YARN架构设计与实现原理

董西成◎著



机械工业出版社
China Machine Press

大数据技术丛书

Hadoop 技术内幕

深入解析 YARN 架构设计与实现原理

董西成 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Hadoop 技术内幕：深入解析 YARN 架构设计与实现原理 / 董西成著. —北京：机械工业出版社，2013.12

(大数据技术丛书)

ISBN 978-7-111-44534-0

I . H… II . 董… III . 数据处理软件 IV . TP274

中国版本图书馆 CIP 数据核字 (2013) 第 252913 号

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书是“Hadoop 技术内幕”系列的第 3 本书，前面两本分别对 Common、HDFS 和 MapReduce 进行了深入分析和讲解，赢得了极好的口碑，Hadoop 领域几乎人手一册，本书则对 YARN 展开了深入的探讨，是首部关于 YARN 的专著。仍然由资深 Hadoop 技术专家董西成执笔，根据最新的 Hadoop 2.0 版本撰写，权威社区 ChinaHadoop 鼎力推荐。

本书从应用角度系统讲解了 YARN 的基本库和组件用法、应用程序设计方法、YARN 上流行的各种计算框架 (MapReduce、Tez、Storm、Spark)，以及多个类 YARN 的开源资源管理系统 (Corona 和 Mesos)；从源代码角度深入分析 YARN 的设计理念与基本架构、各个组件的实现原理，以及各种计算框架的实现细节。

全书共四部分 13 章：第一部分 (第 1~2 章) 主要介绍了如何获取、阅读和调试 Hadoop 的源代码，以及 YARN 的设计思想、基本架构和工作流程；第二部分 (第 3~7 章) 结合源代码详细剖析和讲解了 YARN 的第三方开源库、底层通信库、服务库、事件库的基本使用和实现细节，详细讲解了 YARN 的应用程序设计方法，深入讲解和分析了 ResourceManager、资源调度器、NodeManager 等组件的实现细节；第三篇 (第 8~10 章) 则对离线计算框架 MapReduce、DAG 计算框架 Tez、实时计算框架 Storm 和内存计算框架 Spark 进行了详细的讲解；第四部分 (第 11~13 章) 首先对 Facebook Corona 和 Apache Mesos 进行了深入讲解，然后对 YARN 的发展趋势进行了展望。附录部分收录了 YARN 安装指南、YARN 配置参数以及 Hadoop Shell 命令等非常有用的资料。

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑：孙海亮 罗词亮

印刷

2014 年 1 月第 1 版第 1 次印刷

186mm × 240 mm • 24.75 印张

标准书号：ISBN 978-7-111-44534-0

定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzsj@hzbook.com



前言

为什么要写这本书

在互联网巨头的带动下，开源软件 Hadoop 的应用变得越来越广泛，目前互联网、金融、银行、零售等行业均在使用或者尝试使用 Hadoop。IDC 对未来几年中国的预测中就专门提到了大数据，其认为未来几年，会有越来越多的企业级用户试水大数据平台和应用，而这之中，Hadoop 将成为最耀眼的“明星”。

尽管 Hadoop 整个生态系统是开源的，但由于它包含的软件种类过多，且版本升级过快，大部分公司，尤其是一些中小型公司，难以在有限的时间内快速掌握 Hadoop 蕴含的价值。此外，Hadoop 自身版本的多样化也给很多研发人员带来了很大的学习负担，尽管当前市面上已有很多参考书籍，但遗憾的是，能够深入剖析 Hadoop 内部实现细节的书籍少之又少，而本书则尝试弥补这一缺憾。本书是笔者继《Hadoop 技术内幕：深入解析 MapReduce 架构设计与实现原理》之后的又一本剖析 Hadoop 内幕的书籍。

本书介绍的 YARN（Yet Another Resource Negotiator）系统是 Hadoop 2.0 新增加的一个子项目（与 Common、MapReduce 和 HDFS 三个分支并列），它的引入使得分布式计算系统进入平台化时代，即各种计算框架可以运行在一个集群中，由资源管理系统进行统一管理和调度，它们共享整个集群中的资源进而提高资源利用率。

本书以 Hadoop 2.0 为基础，从基本概念、程序设计和内部实现等方面深入剖析了 Hadoop YARN。本书重点分析了 YARN 的核心实现以及运行在 YARN 上的计算框

架，其中，核心实现包括基础库、编程接口、ResourceManager 实现、资源调度器实现、NodeManager 实现等，而计算框架则包括离线计算框架 MapReduce、DAG 计算框架 Tez、实时计算框架 Storm 和内存计算框架 Spark 等。书中不仅详细介绍了 YARN 各个组件和计算框架的内部实现原理，而且结合源代码进行了深入剖析，使读者可以快速、全面地学习 Hadoop YARN 设计原理和实现细节。

读者对象

（1）Hadoop 二次开发人员

由于在扩展性、容错性和稳定性等方面的诸多优点，Hadoop 已被越来越多公司采用，而为了减少开发成本，大部分公司在 Hadoop 基础上进行二次开发，以打造属于公司内部的 Hadoop 平台。对于这部分 Hadoop 二次开发人员，深入而又全面地了解 Hadoop 的设计与实现细节是修改 Hadoop 内核的前提，而本书可帮助这部分读者快速而又全面地了解 Hadoop 实现细节。

（2）Hadoop 应用开发人员

如果要利用 Hadoop 进行高级应用开发，仅掌握 Hadoop 基本使用方法是远远不够的，必须对 Hadoop 框架的设计原理、架构和运作机制有一定的了解。对这部分读者而言，本书将带领他们全面了解 Hadoop 的设计和实现，加深对 Hadoop 框架的理解，提高开发水平，从而编写出更加高效的应用程序。

（3）Hadoop 运维工程师

对于一名合格的 Hadoop 运维工程师而言，适当地了解 Hadoop 框架的设计原理、架构和运作机制是十分有帮助的，这不仅可以更快地排除各种可能的 Hadoop 故障，也能够让 Hadoop 运维人员与研发人员进行更有效地沟通。通过阅读本书，Hadoop 运维人员可以了解到很多从其他书中无法获取的 Hadoop 实现细节。

（4）开源软件爱好者

Hadoop 是开源软件中的佼佼者，它在实现的过程中吸收了很多开源领域的优秀思想，同时也有很多值得学习的创新。尤为值得一提的是，本书分析 Hadoop 设计和实现的方式也许值得所有开源软件爱好者进行学习和借鉴。通过阅读本书，这部分读者不仅能领略到开源软件的优秀思想，还可以掌握分析开源软件源代码的方法和技巧，从而进一步提高使用开源软件的效率和质量。

如何阅读本书

本书分为四大部分（不包括附录）：

第一部分为基础篇（第 1 ~ 2 章），简单地介绍 Hadoop YARN 的环境搭建和基本设计架构，帮助读者了解一些基础背景知识。

第二部分为 YARN 核心设计篇（第 3 ~ 7 章），着重讲解 YARN 基本库、应用程序设计方法和运行时环境的实现，包括 ResourceManager、NodeManager 和资源调度器等关键组件的内部实现细节。

第三部分为计算框架篇（第 8 ~ 10 章），主要讲解当前比较流行的可运行在 YARN 上的计算框架，包括离线计算框架 MapReduce、DAG 计算框架 Tez、实时计算框架 Storm 和内存计算框架 Spark。

第四部分为高级篇（第 11 ~ 13 章），主要介绍了几个类似于 Hadoop YARN 的开源资源管理系统，包括 Corona、Mesos 等，并总结了资源管理系统的特点及发展趋势。

另外本书最后还添加了几个附录：附录 A 为 YARN 安装指南；附录 B 介绍了常见的 YARN 配置参数；附录 C 介绍了常用的 Hadoop Shell 命令；附录 D 为本书的所有参考资料，包括参考文献、Hadoop jira 和网络资源等。

Hadoop YARN 是 Hadoop 2.0 新引入的系统，对于大部分读者而言，该系统存在很多疑惑与未知之处，而本书正是尝试全方位剖析该系统。为了能够系统化地学习 YARN，推荐读者从第 1 章的基础理论知识开始学习。

勘误和支持

由于笔者的水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。为此，笔者特意创建一个在线支持与应急方案的站点 <http://hadoop123.com>。你可以将书中的错误发布在 Bug 勘误表页面中，同时如果你遇到任何问题，也可以访问 Q&A 页面，我将尽量在线上为读者提供最满意的解答。如果你有更多的宝贵意见，也欢迎发送邮件至邮箱 dongxicheng@yahoo.com，期待能够得到你们的真挚反馈。

致谢

感谢我的导师廖华明副研究员，是她引我进入 Hadoop 世界。

感谢腾讯的蔡斌老师，正是由于他的推荐，才使得两本 Hadoop 书的出版成为可能。

感谢机械工业出版社华章公司的杨福川老师和孙海亮老师在这一年多的时间中始终支持我的写作，他们的鼓励和帮助使我顺利完成了本书。

感谢何鹏、姜冰、郑伟伟、战科宇、周礼、刘晏辰、王群等人给我提供的各种帮助。

最后感谢我的父母，感谢他们的养育之恩，感谢兄长的鼓励和支持，感谢他们时时刻刻给我信心和力量！感谢我的女朋友颢悦对我生活的细心照料与琐事上的宽容。

谨以此书献给我最亲爱的家人，以及众多热爱 Hadoop 的朋友们！

董西成
于北京



目 录

前 言

第一部分 准备篇

第 1 章 环境准备	2
1.1 准备学习环境	2
1.1.1 基础软件下载	2
1.1.2 如何准备 Linux 环境	3
1.2 获取 Hadoop 源代码	5
1.3 搭建 Hadoop 源代码阅读环境	5
1.3.1 创建 Hadoop 工程	5
1.3.2 Hadoop 源代码阅读技巧	8
1.4 Hadoop 源代码组织结构	10
1.5 Hadoop 初体验	12
1.5.1 搭建 Hadoop 环境	12
1.5.2 Hadoop Shell 介绍	15

1.6 编译及调试 Hadoop 源代码	16
1.6.1 编译 Hadoop 源代码	17
1.6.2 调试 Hadoop 源代码	18
1.7 小结	20
第 2 章 YARN 设计理念与基本架构	21
2.1 YARN 产生背景	21
2.1.1 MRv1 的局限性	21
2.1.2 轻量级弹性计算平台	22
2.2 Hadoop 基础知识	23
2.2.1 术语解释	23
2.2.2 Hadoop 版本变迁	25
2.3 YARN 基本设计思想	29
2.3.1 基本框架对比	29
2.3.2 编程模型对比	30
2.4 YARN 基本架构	31
2.4.1 YARN 基本组成结构	32
2.4.2 YARN 通信协议	34
2.5 YARN 工作流程	35
2.6 多角度理解 YARN	36
2.6.1 并行编程	36
2.6.2 资源管理系统	36
2.6.3 云计算	37
2.7 本书涉及内容	38
2.8 小结	38

第二部分 YARN 核心设计篇

第 3 章 YARN 基础库	40
3.1 概述	40
3.2 第三方开源库	41
3.2.1 Protocol Buffers	41

3.2.2	Apache Avro	43
3.3	底层通信库	46
3.3.1	RPC 通信模型	46
3.3.2	Hadoop RPC 的特点概述	48
3.3.3	RPC 总体架构	48
3.3.4	Hadoop RPC 使用方法	49
3.3.5	Hadoop RPC 类详解	51
3.3.6	Hadoop RPC 参数调优	57
3.3.7	YARN RPC 实现	57
3.3.8	YARN RPC 应用实例	61
3.4	服务库与事件库	65
3.4.1	服务库	66
3.4.2	事件库	66
3.4.3	YARN 服务库和事件库的使用方法	68
3.4.4	事件驱动带来的变化	70
3.5	状态机库	72
3.5.1	YARN 状态转换方式	72
3.5.2	状态机类	73
3.5.3	状态机的使用方法	73
3.5.4	状态机可视化	76
3.6	源代码阅读引导	76
3.7	小结	77
3.8	问题讨论	77
第 4 章	YARN 应用程序设计方法	78
4.1	概述	78
4.2	客户端设计	79
4.2.1	客户端编写流程	80
4.2.2	客户端编程库	84
4.3	ApplicationMaster 设计	84
4.3.1	ApplicationMaster 编写流程	84
4.3.2	ApplicationMaster 编程库	92
4.4	YARN 应用程序实例	95

4.4.1	DistributedShell	95
4.4.2	Unmanaged AM	99
4.5	源代码阅读引导	100
4.6	小结	100
4.7	问题讨论	100
第 5 章	ResourceManager 剖析	102
5.1	概述	102
5.1.1	ResourceManager 基本职能	102
5.1.2	ResourceManager 内部架构	103
5.1.3	ResourceManager 事件与事件处理器	106
5.2	用户交互模块	108
5.2.1	ClientRMService	108
5.2.2	AdminService	109
5.3	ApplicationMaster 管理	109
5.4	NodeManager 管理	112
5.5	Application 管理	113
5.6	状态机管理	114
5.6.1	RMApp 状态机	115
5.6.2	RMAppAttempt 状态机	119
5.6.3	RMContainer 状态机	123
5.6.4	RMNode 状态机	127
5.7	几个常见行为分析	129
5.7.1	启动 ApplicationMaster	129
5.7.2	申请与分配 Container	132
5.7.3	杀死 Application	134
5.7.4	Container 超时	135
5.7.5	ApplicationMaster 超时	138
5.7.6	NodeManager 超时	138
5.8	安全管理	139
5.8.1	术语介绍	139
5.8.2	Hadoop 认证机制	139
5.8.3	Hadoop 授权机制	142

5.9 容错机制	144
5.9.1 Hadoop HA 基本框架	145
5.9.2 YARN HA 实现	148
5.10 源代码阅读引导	149
5.11 小结	151
5.12 问题讨论	152
第 6 章 资源调度器	153
6.1 资源调度器背景	153
6.2 HOD 调度器	154
6.2.1 Torque 资源管理器	154
6.2.2 HOD 作业调度	155
6.3 YARN 资源调度器的基本架构	157
6.3.1 基本架构	157
6.3.2 资源表示模型	160
6.3.3 资源调度模型	161
6.3.4 资源抢占模型	164
6.4 YARN 层级队列管理机制	169
6.4.1 层级队列管理机制	169
6.4.2 队列命名规则	171
6.5 Capacity Scheduler	172
6.5.1 Capacity Scheduler 的功能	172
6.5.2 Capacity Scheduler 实现	176
6.6 Fair Scheduler	179
6.6.1 Fair Scheduler 功能介绍	180
6.6.2 Fair Scheduler 实现	182
6.6.3 Fair Scheduler 与 Capacity Scheduler 对比	183
6.7 其他资源调度器介绍	184
6.8 源代码阅读引导	185
6.9 小结	186
6.10 问题讨论	187

第 7 章 NodeManager 剖析	188
7.1 概述	188
7.1.1 NodeManager 基本职能	188
7.1.2 NodeManager 内部架构	190
7.1.3 NodeManager 事件与事件处理器	193
7.2 节点健康状况检测	194
7.2.1 自定义 Shell 脚本	194
7.2.2 检测磁盘损坏数目	196
7.3 分布式缓存机制	196
7.3.1 资源可见性与分类	198
7.3.2 分布式缓存实现	200
7.4 目录结构管理	203
7.4.1 数据目录管理	203
7.4.2 日志目录管理	203
7.5 状态机管理	206
7.5.1 Application 状态机	207
7.5.2 Container 状态机	210
7.5.3 LocalizedResource 状态机	213
7.6 Container 生命周期剖析	214
7.6.1 Container 资源本地化	214
7.6.2 Container 运行	218
7.6.3 Container 资源清理	222
7.7 资源隔离	224
7.7.1 Cgroups 介绍	224
7.7.2 内存资源隔离	228
7.7.3 CPU 资源隔离	230
7.8 源代码阅读引导	234
7.9 小结	235
7.10 问题讨论	236

第三部分 计算框架篇

第 8 章 离线计算框架 MapReduce	238
8.1 概述	238
8.1.1 基本构成	238
8.1.2 事件与事件处理器	240
8.2 MapReduce 客户端	241
8.2.1 ApplicationClientProtocol 协议	242
8.2.2 MRClientProtocol 协议	243
8.3 MRAppMaster 工作流程	243
8.4 MR 作业生命周期及相关状态机	246
8.4.1 MR 作业生命周期	246
8.4.2 Job 状态机	249
8.4.3 Task 状态机	253
8.4.4 TaskAttempt 状态机	255
8.5 资源申请与再分配	259
8.5.1 资源申请	259
8.5.2 资源再分配	262
8.6 Container 启动与释放	263
8.7 推测执行机制	264
8.7.1 算法介绍	265
8.7.2 推测执行相关类	266
8.8 作业恢复	267
8.9 数据处理引擎	269
8.10 历史作业管理器	271
8.11 MRv1 与 MRv2 对比	273
8.11.1 MRv1 On YARN	273
8.11.2 MRv1 与 MRv2 架构比较	274
8.11.3 MRv1 与 MRv2 编程接口兼容性	274
8.12 源代码阅读引导	275
8.13 小结	277
8.14 问题讨论	277

第 9 章 DAG 计算框架 Tez	278
9.1 背景	278
9.2 Tez 数据处理引擎	281
9.2.1 Tez 编程模型	281
9.2.2 Tez 数据处理引擎	282
9.3 DAG Master 实现	284
9.3.1 DAG 编程模型	284
9.3.2 MR 到 DAG 转换	286
9.3.3 DAGAppMaster	288
9.4 优化机制	291
9.4.1 当前 YARN 框架存在的问题	291
9.4.2 Tez 引入的优化技术	292
9.5 Tez 应用场景	292
9.6 与其他系统比较	294
9.7 小结	295
第 10 章 实时 / 内存计算框架 Storm/Spark	296
10.1 Hadoop MapReduce 的短板	296
10.2 实时计算框架 Storm	296
10.2.1 Storm 编程模型	297
10.2.2 Storm 基本架构	302
10.2.3 Storm On YARN	304
10.3 内存计算框架 Spark	307
10.3.1 Spark 编程模型	308
10.3.2 Spark 基本架构	312
10.3.3 Spark On YARN	316
10.3.4 Spark/Storm On YARN 比较	317
10.4 小结	317

第四部分 高级篇

第 11 章 Facebook Corona 剖析	320
11.1 概述	320
11.1.1 Corona 的基本架构	320
11.1.2 Corona 的 RPC 协议与序列化框架	322
11.2 Corona 设计特点	323
11.2.1 推式网络通信模型	323
11.2.2 基于 Hadoop 0.20 版本	324
11.2.3 使用 Thrift	324
11.2.4 深度集成 Fair Scheduler	324
11.3 工作流程介绍	324
11.3.1 作业提交	325
11.3.2 资源申请与任务启动	326
11.4 主要模块介绍	327
11.4.1 ClusterManager	327
11.4.2 CoronaJobTracker	330
11.4.3 CoronaTaskTracker	333
11.5 小结	335
第 12 章 Apache Mesos 剖析	336
12.1 概述	336
12.2 底层网络通信库	337
12.2.1 libprocess 基本架构	338
12.2.2 一个简单示例	338
12.3 Mesos 服务	340
12.3.1 SchedulerProcess	341
12.3.2 Mesos Master	342
12.3.3 Mesos Slave	343
12.3.4 ExecutorProcess	343
12.4 Mesos 工作流程	344
12.4.1 框架注册过程	344

12.4.2	Framework Executor 注册过程	345
12.4.3	资源分配到任务运行过程	345
12.4.4	任务启动过程	347
12.4.5	任务状态更新过程	347
12.5	Mesos 资源分配策略	348
12.5.1	Mesos 资源分配框架	349
12.5.2	Mesos 资源分配算法	349
12.6	Mesos 容错机制	350
12.6.1	Mesos Master 容错	350
12.6.2	Mesos Slave 容错	351
12.7	Mesos 应用实例	352
12.7.1	Hadoop On Mesos	352
12.7.2	Storm On Mesos	353
12.8	Mesos 与 YARN 对比	354
12.9	小结	355
第 13 章	YARN 总结与发展趋势	356
13.1	资源管理系统设计动机	356
13.2	资源管理系统架构演化	357
13.2.1	集中式架构	357
13.2.2	双层调度架构	358
13.2.3	共享状态架构	358
13.3	YARN 发展趋势	359
13.3.1	YARN 自身的完善	359
13.3.2	以 YARN 为核心的生态系统	361
13.3.3	YARN 周边工具的完善	363
13.4	小结	363
附录 A	YARN 安装指南	364
附录 B	YARN 配置参数介绍	367
附录 C	Hadoop Shell 命令介绍	371
附录 D	参考资料	374



第一部分 准备篇

由于 MRv1 (MapReduce version 1) 在扩展性、可靠性、资源利用率和多框架等方面存在明显不足，故 Apache 开始尝试对 MapReduce 进行升级改造，进而诞生了下一代 MapReduce 计算框架 MRv2 (MapReduce version 2)。由于 MRv2 将资源管理模块构建成了一个独立的通用系统 YARN，这使得 MRv2 的核心从单一计算框架 MapReduce 转移为通用资源管理系统 YARN。本书第一部分将介绍学习 MRv2 前的准备工作，并给出 MRv2 和 YARN 的基本概念和架构。

第1章 环境准备

一般而言，在深入研究一个系统的技术细节之前，先要进行一些基本的准备工作，比如准备源代码阅读环境，搭建运行环境并尝试使用该系统等。然而，对于 Hadoop 而言，由于它是一个分布式系统，由多种守护进程组成，具有一定的复杂性，如果想深入学习其设计原理，仅仅进行以上几项准备工作是不够的，还要学习一些调试工具的使用方法，以便对 Hadoop 源代码进行调试、跟踪，边用边学，这样才能事半功倍。

本章编写目的是帮助读者构建一个“高效”的 Hadoop 源代码学习环境，包括 Hadoop 源代码阅读环境、Hadoop 使用环境和 Hadoop 源代码编译调试环境等，主要涉及如下内容：

- 在 Linux 环境下搭建 Hadoop 源代码阅读环境；
- 在 Linux 环境下搭建一个 Hadoop 集群（包括 YARN 和 HDFS 两个系统）；
- Hadoop 的基本使用方法，主要涉及 Hadoop Shell 和 Eclipse 插件两种工具的使用；
- Hadoop 源代码编译和调试方法，其中调试方法包括使用 Eclipse 远程调试和打印调试日志两种。

考虑到大部分用户在单机上学习 Hadoop 源代码，所以本章内容均是基于单机环境的。本章大部分内容较为基础，已经掌握这部分内容的读者可以直接跳过本章。

1.1 准备学习环境

对于大部分公司而言，实验和生产环境中的服务器集群部署的是 Linux 操作系统，考虑到 Linux 在服务器市场中具有统治地位，Hadoop 从一开始便是基于 Linux 操作系统开发的，因而对 Linux 有非常完美的支持。尽管 Hadoop 采用了具有跨平台特性的 Java 作为主要编程语言，但由于它的一些功能实现用到了 Linux 操作系统相关的技术，因而对其他平台支持不够友好，且没有进行过严格测试。换句话说，其他操作系统（如 Windows）仅可作为开发环境^①，不可作为生产环境。对于学习源代码而言，操作系统的选择显得不是非常重要，读者可根据个人爱好自行决定。本节以 64 bit Linux 为例，介绍如何在单机上准备 Hadoop 源代码学习环境。

1.1.1 基础软件下载

前面提到 Hadoop 采用的开发语言主要是 Java，因而搭建 Hadoop 环境所需的最基础软

① 截至本书结稿时，Apache Hadoop SVN 中已经出现了针对 Windows 操作系统的分支，具体见 <http://svn.apache.org/repos/asf/hadoop/common/branches/> 下的 branch-1-win 和 branch-trunk-win，且 Hortonworks 公司发布了 Windows 安装版本，具体见 <http://hortonworks.com/partners/microsoft/>。

件首先应该包括 Java 基础开发包 JDK 和 Java 项目管理工具 Maven，考虑到源代码阅读和调试的便利性，本书采用功能强大的集成开发环境 Eclipse。搭建 Hadoop 阅读环境需要的各种软件以及下载方式如表 1-1 所示。

表 1-1 搭建 Hadoop 阅读环境所需的软件

软件	下载网址	推荐版本
JDK	http://www.oracle.com/technetwork/java/javase/downloads/index.html	1.6 以上
Maven	http://maven.apache.org/download.cgi	3.0.2 以上
Eclipse	http://www.eclipse.org/downloads/	Galileo 以上版本 [⊖]

1.1.2 如何准备 Linux 环境

本节主要介绍如何准备 Linux 下 Hadoop 学习环境。搭建 Linux 学习环境需要安装 JDK 和 Eclipse 等软件。为了方便 1.6 节介绍 Hadoop 源代码编译方法，本节顺便安装 Hadoop 项目管理工具 Maven。本文以 64 bit Ubuntu 为例，介绍安装这些软件的方法，最终安装完成的目录结构为：

```

ROOT
├── home
│   └── dong
│       └── eclipse
├── usr
│   └── lib
│       ├── apache-maven-3.0.5
│       └── jvm
│           └── jdk1.6.0_25

```

1. JDK 安装与配置

一般而言，Ubuntu 系统会自带 JDK，如果没有或者版本不合要求，可按以下步骤进行安装。

步骤 1 安装 JDK。

将下载的 .bin 文件复制到 Linux 的某个目录下，比如 /usr/lib/jvm/，然后在 Shell 中执行以下命令为该文件添加可执行权限：

```
chmod +x /usr/lib/jvm/jdk1.6.0_25.bin
```

然后执行以下命令安装 JDK：

```
sudo /usr/lib/jvm/jdk1.6.0_25.bin
```

之后将会出现安装信息，直至屏幕显示要求按下回车键，此时输入回车键后，会把 JDK 解压到文件夹 jdk1.6.0_25 中。至此，JDK 已安装完毕，下面进行配置。

[⊖] 注意，Indigo 及以上版本与 Hadoop Eclipse 插件可能存在兼容问题。

步骤 2 配置 JDK。

修改 `/etc/profile` 文件，在里面添加以下内容：

```
export JAVA_HOME=/usr/lib/jvm/jdk1.6.0_25
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=$CLASSPATH:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
```

输入以下命令使配置生效：

```
source /etc/profile
```

步骤 3 修改默认 JDK 版本。

Ubuntu 中可能会有默认的 JDK，如 `openjdk`，因而我们需要将自己安装的 JDK 设置为默认 JDK 版本，执行下面的代码：

```
sudo update-alternatives --install /usr/bin/java java /usr/lib/jvm/jdk1.6.0_25/
bin/java 300
sudo update-alternatives --install /usr/bin/javac javac /usr/lib/jvm/jdk1.6.0_25/
bin/javac 300
sudo update-alternatives --install /usr/bin/jar jar /usr/lib/jvm/jdk1.6.0_25/
bin/jar 300
sudo update-alternatives --install /usr/bin/javah javah /usr/lib/jvm/jdk1.6.0_25/
bin/javah 300
sudo update-alternatives --install /usr/bin/javap javap /usr/lib/jvm/jdk1.6.0_25/
bin/javap 300
```

然后执行以下代码选择我们安装的 JDK 版本：

```
sudo update-alternatives --config java
```

步骤 4 验证 JDK 是否安装成功。

重启 Shell 终端，执行 `java -version` 命令，若输出以下内容，则说明安装成功：

```
java version "1.6.0_25"
Java(TM) SE Runtime Environment (build 1.6.0_25-b06)
Java HotSpot(TM) Client VM (build 20.0-b11, mixed mode, sharing)
```

2. 安装、配置 Maven 及 Eclipse

下面介绍 Maven 和 Eclipse 的安装、配置方法。

(1) 安装与配置 Maven

首先解压下载包，比如解压到文件 `/usr/lib/ apache-maven-3.0.5` 目录下，然后修改 `/etc/profile` 文件，在里面添加以下内容：

```
export MAVEN_HOME=/usr/lib/apache-maven-3.0.5
export PATH=$PATH:$MANT_HOME/bin
```

输入以下命令使配置生效：

```
source /etc/profile
```

同 Windows 下的验证方式一样，重启终端，执行 `mvn --version` 命令，若输出以下内容，则说明安装成功：

Apache Maven version 3.0.5 compiled on June 27 2008

(2) 安装 Eclipse

同 Windows 环境下安装方式一样，直接解压即可使用。

1.2 获取 Hadoop 源代码

当前比较流行的 Hadoop 源代码版本有两个：Apache Hadoop 和 Cloudera Distributed Hadoop（简称 CDH）。Apache Hadoop 是由雅虎、Cloudera、Facebook 等公司组成的 Hadoop 社区共同研发的，它属于最原始的开源版本，在该版本基础上，很多公司进行了封装和优化，推出了自己的开源版本，其中，最有名的一个是 Cloudera 公司发布的 CDH 版本。

考虑到 Apache Hadoop 是最原始的版本，且使用最为广泛，因而本书选用了 Apache Hadoop 版本作为分析对象。自从 Apache Hadoop 发布以来，已经陆续推出很多版本（具体见 2.2 节），读者可自行在 Hadoop SVN 地址 <http://svn.apache.org/repos/asf/hadoop/common/branches/> 查看或者下载所有版本，也可以从 Apache 官方主页 <http://hadoop.apache.org/releases.html> 上下载最新版本。

本书介绍的 YARN 属于 Hadoop 2.0 的一个分支（另外两个分支分别是 HDFS 和 MapReduce），Hadoop 2.0 的命名方式一般为 `hadoop-2.x.x`。Apache 官方主页提供了两个压缩包，一个是 Hadoop 源代码（`hadoop-{VERSION}-src.tar.gz`），一个是可直接用于部署的 JAR 包（`hadoop-{VERSION}.tar.gz`），Cloudera 发布的 CDH 版本则将源代码和 JAR 包存放在一起组成一个压缩包（`hadoop-2.0.0-cdh4.x.x.tar.gz`）^①。

本书介绍的 YARN 设计思想适用于所有 Apache Hadoop 2.x 版本，但涉及具体的实现（指源代码级别的实现）时，则以 Apache Hadoop 2.2.0 及更高稳定版本为主，因此，如果你想对比 Hadoop 源代码阅读本书，推荐下载 Apache 2.2.0 或更高版本。

1.3 搭建 Hadoop 源代码阅读环境

1.3.1 创建 Hadoop 工程

本节将介绍如何创建一个 Hadoop 源代码工程以方便阅读源代码。总体上说，目前存在两种 Hadoop 源代码阅读环境搭建方法，分别是构建 Maven 工程和构建 Java 工程。两种方法各有利弊：前者可通过网络自动下载依赖的第三方库，但源代码会被分散到多个工程中进而带来阅读上的不便；后者可将所有源代码组织在一个工程中，但需要自己添加依赖的第三方库，大家可根据自己的喜好选择一种方法。本节将依次介绍这两种方法。

(1) 构建 Maven 工程

通过 Maven 工程搭建 Hadoop 源代码阅读环境的步骤如下：

^① CDH4 下载地址：<http://archive.cloudera.com/cdh4/cdh/4/>。

步骤 1 解压缩 Hadoop 源代码。

将下载到的 Hadoop 源代码压缩包解压到工作目录下，比如 `hadoop-2.0-src.tar.gz`（注意，为了方便，此处直接使用版本号 2.0，实际下载到的源代码版本号并不是这样的，可能是 2.2.0，这样压缩包的名字实际为 `hadoop-2.2.0-src.tar.gz`）。

步骤 2 导入 Maven 工程。

在 Eclipse 中，依次选择“File”→“Import”→“Maven”→“Existing Maven Project”，在弹出的对话框中的“Root Directory”后面，选择 Java 源代码所在的目录。

单击“Next”按钮，在弹出的对话框中选择“Resolve All Later”，并单击“Finish”按钮完成 Maven 项目导入。之后，Eclipse 会自动通过网络从 Maven 库中下载依赖的第三方库（JAR 包等）。注意，你所使用的电脑必须能够联网。

将 Hadoop 2.0 源代码导入 Maven 项目后，会生成 50 个左右的工程，这些都是通过 Maven 构建出来的，每个工程是一个代码模块，且彼此相对独立，可以单独编译。你可以在某个工程下的“`src/main/java`”目录下查看对应的源代码。

（2）构建 Java 工程

通过 Java 工程搭建 Hadoop 源代码阅读环境的步骤如下：

步骤 1 解压缩 Hadoop 源代码。

同“构建 Maven 工程”中的步骤 1 类似。

步骤 2 新建 Java 工程。

打开 Eclipse，进入 Eclipse 可视化界面后，如图 1-1 所示，依次单击“File”→“New”→“Java Project”，并在弹出的对话框中去掉“Use default location”前的勾号，然后选择 Hadoop 安装目录的位置，默认情况下，工程名称与 Hadoop 安装目录名称相同，用户可自行修改。单击“完成”按钮，Hadoop 源代码工程创建完毕。

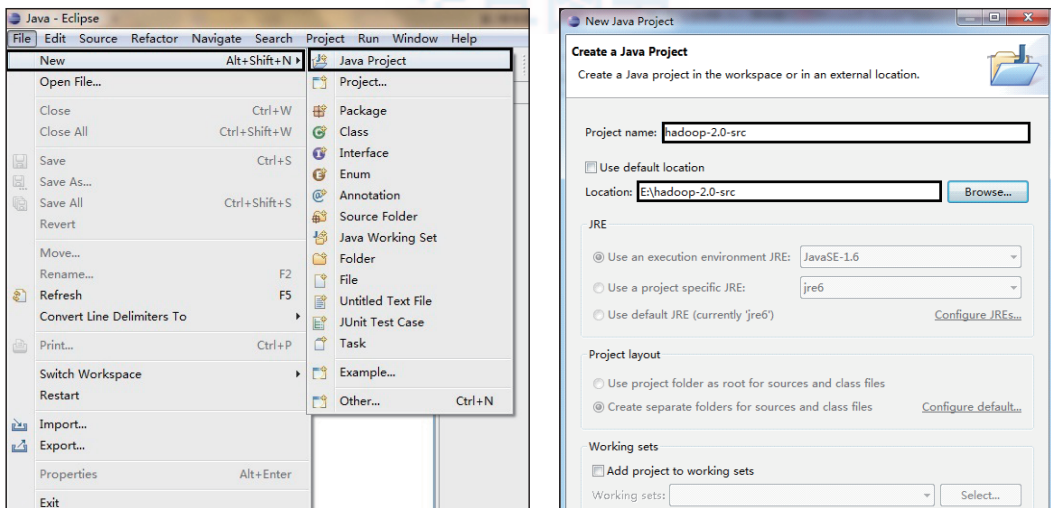


图 1-1 新建 Hadoop 工程

回到 Eclipse 主界面后，打开新建的 Hadoop 工程，可看到整个工程按图 1-2 所示组织代码：按目录组织源代码，且每个目录下以 JAR 包为单位显示各个 Java 文件。

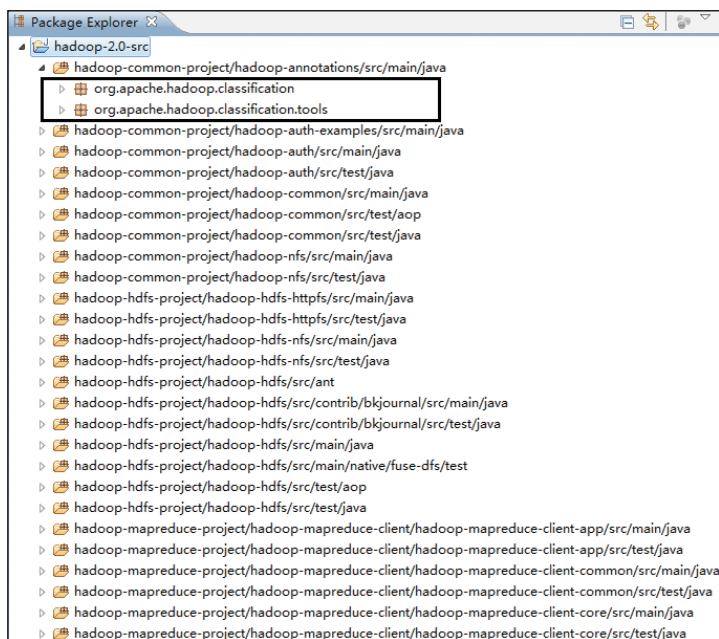


图 1-2 Hadoop 工程展示（部分）源代码方式

除了使用源代码压缩包导入 Eclipse 工程的方法外，读者可也尝试直接从 Hadoop SVN 上导入 Hadoop 源代码。

需要注意的是，通过以上方法导入 Hadoop 2.0 源代码后，很多类或者方法找不到对应的 JAR 包，为了解决这个问题，你需要将第三方 JAR 包导入工程中，如图 1-3 所示，方法如下：解压存放 JAR 包的压缩包，然后右击 Project 名称，在弹出的快捷菜单中选择“Properties”命令，将会弹出一个界面，然后在该界面中依次选择“Java Build Path”→“Libraries”→“Add External JARs...”，将解压目录中 share/hadoop 目录下各个子目录中 lib 文件夹下的 JAR 包导入工程。

前面提到 CDH 版本将源代码和 JAR 包放到了一起，因此，如果使用 CDH 版本，则直接按照上述方法将源代码导入 Eclipse 工程即可。

细心的读者在阅读源代码过程中仍会发现部分类或者函数无法找到，这是因为 Hadoop 2.0 使用了 Protocol Buffers 定义了 RPC 协议，而这些 Protocol Buffers 文件在 Maven 编译源代码时才会生成对应的 Java 类，因此若其他类在源代码中引用这些类则暂时无法找到，解决方法是先编译 Hadoop 2.0 源代码再倒入 Eclipse 工程，具体方法如下（注意，进行以下步骤之前，需先完成 1.6.1 节中的准备工作）。

首先，使用以下命令安装 Eclipse 插件 hadoop-maven-plugins：

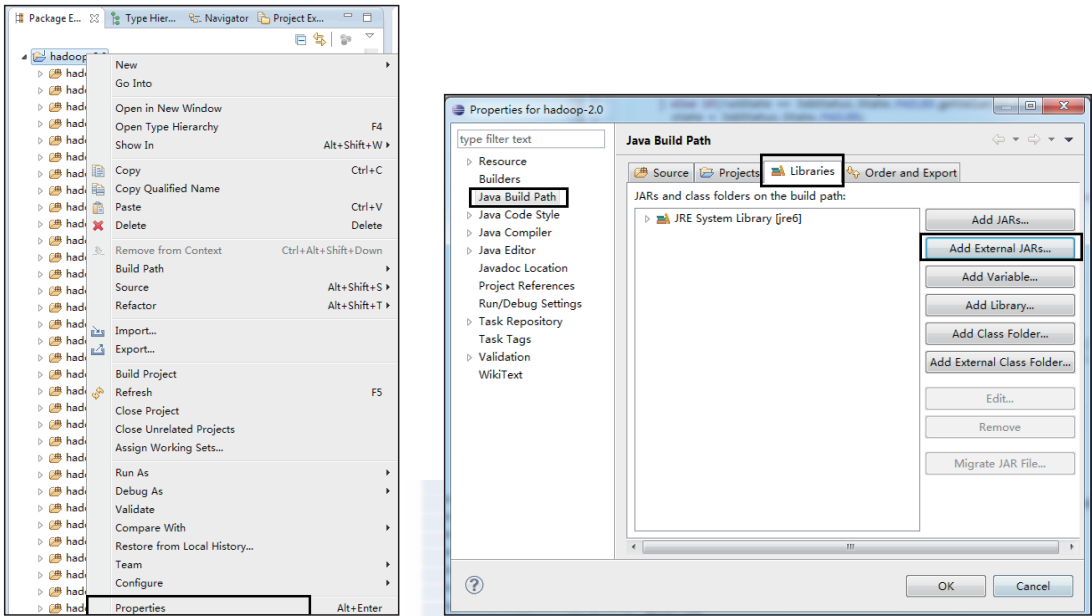


图 1-3 导入依赖的第三方 JAR 包

```
$ cd ${HADOOP_HOME}/hadoop-maven-plugins
$ mvn install
```

然后生成 Eclipse 工程文件：

```
$ cd ${HADOOP_HOME}
$ mvn eclipse:eclipse -DskipTests
```

最后在 Eclipse 中按照以下流程导入源代码：“File”→“Import”→“Existing Projects into Workspace”。

1.3.2 Hadoop 源代码阅读技巧

本节介绍在 Eclipse 下阅读 Hadoop 源代码的一些技巧，比如：如何查看一个基类有哪些派生类，一个方法被其他哪些方法调用等。

（1）查看一个基类或接口的派生类或实现类

在 Eclipse 中，选中某个基类或接口名称，右击，在弹出的快捷菜单中选择“Quick Type Hierarchy”，可在新窗口中看到对应的所有派生类或实现类。

例如，如图 1-4 所示，打开 hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java 目录下 org.apache.hadoop.mapred 包中的 InputFormat.java 文件，查看接口 InputFormat 的所有实现类，结果如图 1-5 所示。

（2）查看函数的调用关系

在 Eclipse 中，选中某个方法名称，右击，在弹出的快捷菜单中选择“Open Call

Hierarchy”，可在窗口“Call Hierarchy”中看到所有调用该方法的函数。

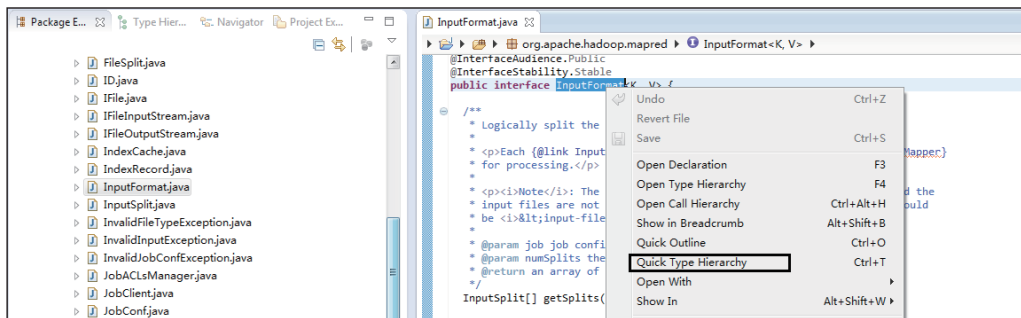


图 1-4 在 Eclipse 中查看 Hadoop 源代码中接口 InputFormat 的所有实现类

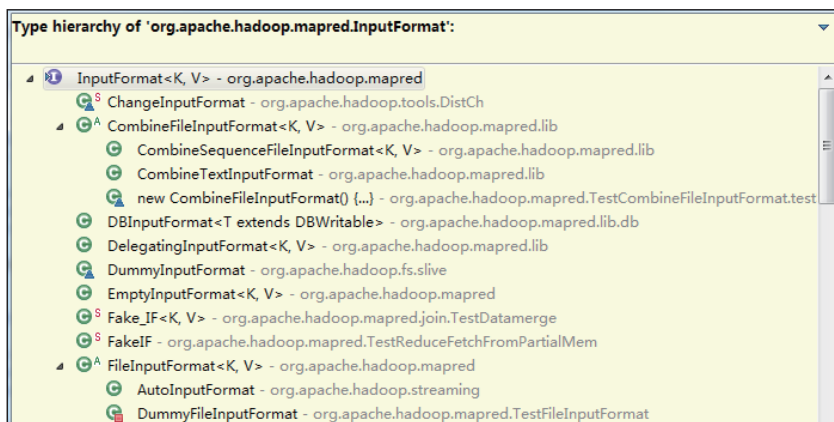


图 1-5 Eclipse 列出接口 InputFormat 的所有实现类

例如，如图 1-6 所示，打开 hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java 目录下 org.apache.hadoop.mapred 包中的 Task.java 文件，查看调用 getJobID 方法的所有函数，结果如图 1-7 所示。

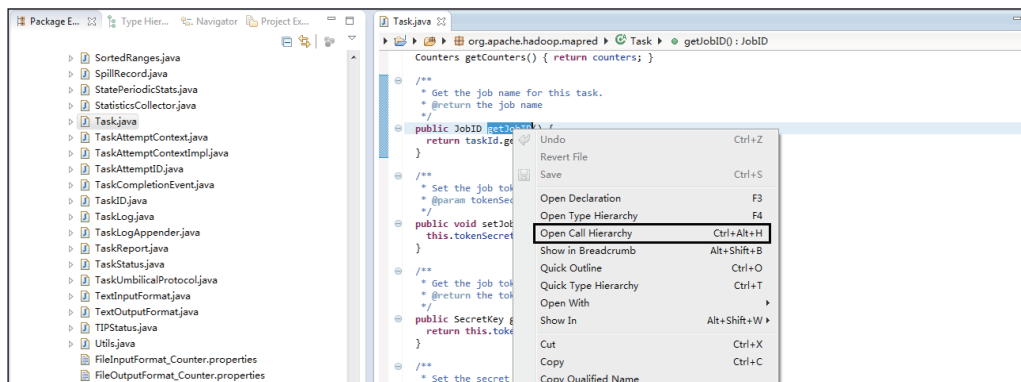


图 1-6 在 Eclipse 中查看 Hadoop 源代码中所有调用 Task.java 中 getJobID 方法的函数

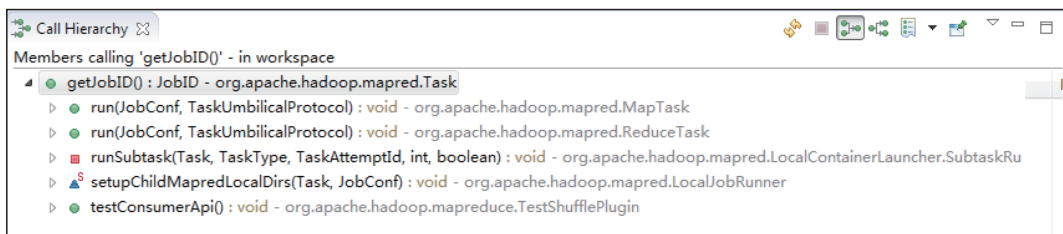


图 1-7 Eclipse 列出所有调用 getJobID 方法的函数

(3) 快速查找类对象的相关信息

与前两个小节类似，选中类对象，右击，在弹出的快捷菜单中选择“Open Declaration”，可跳转到类定义；选择“Quick Outline”，可查看类所有的成员变量和成员方法，具体细节在此不做详细介绍，读者自行尝试。

1.4 Hadoop 源代码组织结构

在 Hadoop 的 JAR 压缩包解压后的目录 `hadoop-{VERSION}` 中包含了 Hadoop 全部的管理脚本和 JAR 包，下面简单对这些文件或目录进行介绍。

- ❑ **bin**：Hadoop 最基本的管理脚本和使用脚本所在目录，这些脚本是 `sbin` 目录下管理脚本的基础实现，用户可以直接使用这些脚本管理和使用 Hadoop。
- ❑ **etc**：Hadoop 配置文件所在的目录，包括 `core-site.xml`、`hdfs-site.xml`、`mapred-site.xml` 等从 Hadoop 1.0 继承而来的配置文件和 `yarn-site.xml` 等 Hadoop 2.0 新增的配置文件。
- ❑ **include**：对外提供的编程库头文件（具体动态库和静态库在 `lib` 目录中），这些头文件均是用 C++ 定义的，通常用于 C++ 语言访问 HDFS 或者编写 MapReduce 程序。
- ❑ **lib**：该目录包含了 Hadoop 对外提供的编程动态库和静态库，与 `include` 目录中的头文件结合使用。
- ❑ **libexec**：各个服务对应的 Shell 配置文件所在目录，可用于配置日志输出目录、启动参数（比如 JVM 参数）等基本信息。
- ❑ **sbin**：Hadoop 管理脚本所在目录，主要包含 HDFS 和 YARN 中各类服务的启动 / 关闭脚本。
- ❑ **share**：Hadoop 各个模块编译后的 JAR 包所在目录。

在 Hadoop 源代码压缩包解压后的目录 `hadoop-{VERSION}-src` 中，可看到如图 1-8 所示的目录结构，其中，比较重要的目录有：`hadoop-common-project`、`hadoop-mapreduce-project`、`hadoop-hdfs-project` 和 `hadoop-yarn-project` 等，下面分别介绍这几个目录的作用。

- ❑ **hadoop-common-project**：Hadoop 基础库所在目录，该目录中包含了其他所有模块可能会用到的基础库，包括 RPC、Metrics、Counter 等。

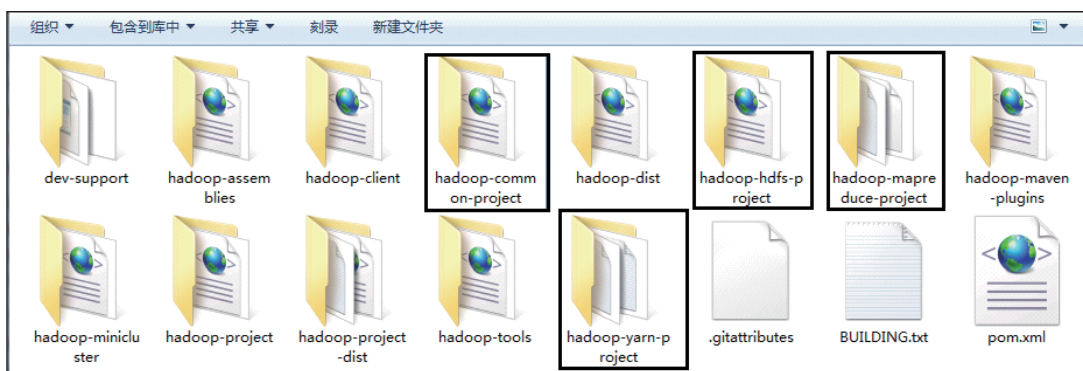


图 1-8 Hadoop 安装目录结构

□ **hadoop-mapreduce-project** : MapReduce 框架的实现, 在 MRv1 中, MapReduce 由编程模型 (map/reduce)、调度系统 (JobTracker 和 TaskTracker) 和数据处理引擎 (MapTask 和 ReduceTask) 等模块组成, 而此处的 MapReduce 则不同于 MRv1 中的实现, 它的资源调度功能由新增的 YARN 完成 (编程模型和数据处理引擎不变), 自身仅包含非常简单的任务分配功能。

□ **hadoop-hdfs-project** : Hadoop 分布式文件系统实现, 不同于 Hadoop 1.0 中单 NameNode 实现, Hadoop 2.0 支持多 NameNode, 同时解决了 NameNode 单点故障问题。

□ **hadoop-yarn-project** : Hadoop 资源管理系统 YARN 实现。这是 Hadoop 2.0 新引入的分支, 该系统能够统一管理系统中的资源, 并按照一定的策略分配给各个应用程序, 本书将重点剖析 YARN 的实现。

本书重点介绍 YARN 的实现原理, 下面就对 Hadoop YARN 源代码组织结构^①进行介绍。YARN 目录组织结构如图 1-9 所示。

总体上看, Hadoop YARN 分为 5 部分: API、Common、Applications、Client 和 Server, 它们的内容具体如下:

□ **YARN API (hadoop-yarn-api 目录)**: 给出了 YARN 内部涉及的 4 个主要 RPC 协议的 Java 声明和 Protocol Buffers 定义, 这 4 个 RPC 协议分别是 ApplicationClientProtocol、ApplicationMasterProtocol、ContainerManagementProtocol 和 ResourceManagerAdministrationProtocol, 本书将在第 2 章对这部分内容进行详细介绍。

□ **YARN Common (hadoop-yarn-common 目录)**: 该部分包含了 YARN 底层库实现, 包括事件库、服务库、状态机库、Web 界面库等, 本书将在第 3 章对这部分内容进行详细介绍。

□ **YARN Applications (hadoop-yarn-applications 目录)**: 该部分包含了两个 Application 编程实例, 分别是 distributedshell 和 Unmanaged AM, 本书将在第 4 章对这部分内容进行详细介绍。

① 不同 Hadoop 版本的源代码组织结构有较大差别, 本书的分析是基于 Hadoop 1.0.0 的。

- ❑ YARN Client (hadoop-yarn-client 目录)：该部分封装了几个与 YARN RPC 协议交互相关的库，方便用户开发应用程序，本书将在第 4 章对这部分内容进行详细介绍。
- ❑ YARN Server (hadoop-yarn-server 目录)：该部分给出了 YARN 的核心实现，包括 ResourceManager、NodeManager、资源管理器等核心组件的实现，本书将在第 5~7 章对这部分内容进行详细介绍。

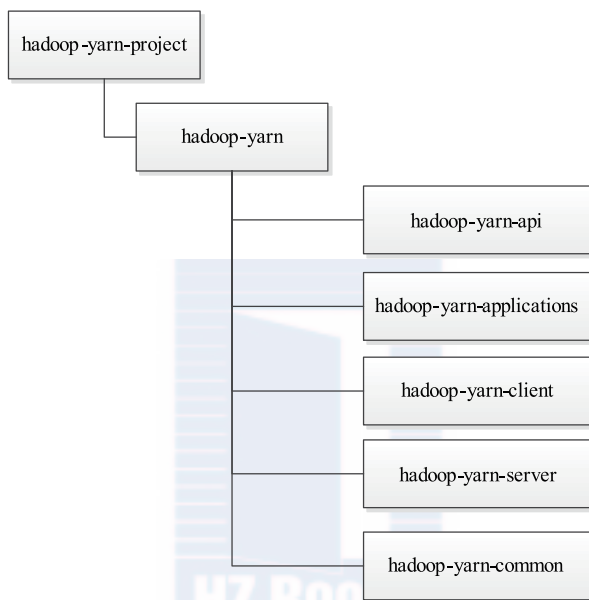


图 1-9 Hadoop YARN 目录组织结构

1.5 Hadoop 初体验

一般而言，我们想要深入学习一个新的系统时，首先要尝试使用该系统，了解系统对外提供的功能，然后再通过某个功能逐步深入其实现细节。本节将介绍如何在伪分布式工作模式^①下使用 Hadoop，包括搭建 Hadoop 环境、访问 HDFS 以及向 YARN 提交应用程序等最基本的操作。本节只是有代表性地介绍 Hadoop 的一些基本使用方法，使读者对 Hadoop 有一个初步认识，并引导读者逐步进行更全面的学习。

1.5.1 搭建 Hadoop 环境

本小节仅介绍单机环境的搭建方法，更加完整的 Hadoop 安装步骤和配置方法可参考

^① 单机环境中，Hadoop 有两种工作模式：本地模式和伪分布式模式。其中，本地模式完全运行在本地，不会加载任何 Hadoop 服务，因而不会涉及 Hadoop 最核心的代码实现，伪分布式即为“单点集群”，在该模式下，所有的守护进程均会运行在单个节点上，因而本节选用该工作模式。

本书最后的附录 A 和附录 B。另外，需要注意的是，由于不同用户拥有的 Linux 环境不尽相同（比如已经安装的软件不同、统一软件的版本不同等），每个人安装 Hadoop 过程中遇到的问题可能不同，此时需要根据具体的日志提示解决问题。本小节仅给出一般情况下，Hadoop 2.0 的安装步骤。

步骤 1 修改 Hadoop 配置文件。

1) 设置环境变量。在 `${HADOOP_HOME}/etc/hadoop/hadoop-env.sh` 中，添加 JAVA 安装目录，命令如下：

```
export JAVA_HOME=/usrb/jvm/java-6-openjdk
```

修改 `conf` 目录下的 `mapred-site.xml`、`core-site.xml`、`yarn-site.xml` 和 `hdfs-site.xml` 四个文件，在 `<configuration>` 与 `</configuration>` 之间添加的内容见下面的介绍。

2) 在 `${HADOOP_HOME}/etc/hadoop/` 下，将 `mapred-site.xml.template` 重命名成 `mapred-site.xml`，并添加以下内容：

```
<property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
</property>
```

【解释】相比于 Hadoop 1.0，用户无须再配置 `mapred.job.tracker`，这是因为 JobTracker 相关实现已变成客户端的一个库（实际上在 Hadoop 2.0 中，JobTracker 已经不存在，它的功能由另外一个称为 MRAppMaster 的组件实现），它可能被随机调度到任何一个 slave 上，也就是它的位置是动态生成的。需要注意的是，在该配置文件中需用 `mapreduce.framework.name` 指定采用的运行时框架的名称，在此指定“yarn”。

3) 在 `${HADOOP_HOME}/etc/hadoop/` 中，修改 `core-site.xml`，为了简单，我们仍采用 Hadoop 1.0 中的 HDFS 工作模式（不配置 HDFS Federation），修改后如下：

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://YARN001:8020</value>
</property>
```

其中，YARN001 表示节点的 IP 或者 host。

4) 在 `${HADOOP_HOME}/etc/hadoop/` 中，修改 `yarn-site.xml`，修改后如下：

```
<property>
  <name>yarn.nodemanager.aux-services</name>
  <value>mapreduce-shuffle</value>
</property>
```

【解释】为了能够运行 MapReduce 程序，需要让各个 NodeManager 在启动时加载 shuffle server，shuffle server 实际上是 Jetty/Netty Server，Reduce Task 通过该 server 从各个 NodeManager 上远程复制 Map Task 产生的中间结果。上面增加的两个配置均用于指定 shuffle server。

5) 修改 `${HADOOP_HOME}/etc/hadoop` 中的 `hdfs-site.xml` 文件:

```
<property>
  <name>dfs.replication</name>
  <value>1</value>
</property>
```

【解释】默认情况下，HDFS 数据块副本数是 3，而在集群规模小于 3 的集群中该参数会导致出现错误，这可通过将 `dfs.replication` 调整为 1 解决。

注意 如果你是在虚拟机中搭建 Hadoop 环境，且虚拟机经常关闭与重启，为了避免每次重新虚拟机后启动 Hadoop 时出现各种问题，建议在 `core-site.xml` 中将 `hadoop.tmp.dir` 属性设置为一个非 `/tmp` 目录，比如 `/data` 或者 `/home/dongxicheng/data`（注意该目录对当前用户需具有读写权限）。

步骤 2 设置免密码登录。

前面提到 Hadoop 启动 / 停止脚本需要通过 SSH 发送命令启动相关守护进程，为了避免每次启动 / 停止 Hadoop 都要输入密码进行验证，需设置免密码登录，步骤如下。

1) 打开命令行终端，输入以下命令：

```
ssh-keygen -t rsa
```

将会在“`~/.ssh/`”目录下生成公钥文件 `id_rsa.pub` 和私钥文件 `id_rsa`。

2) 将公钥文件 `id_rsa.pub` 中的内容复制到相同目录下的 `authorized_keys` 文件中：

```
cd ~/.ssh/
cat id_rsa.pub >> authorized_keys
```

步骤 3 启动 Hadoop。

在 Hadoop 安装目录中，按以下三步操作启动 Hadoop，我们单步启动每一个服务，以便于排查错误，如果某一个服务没有启动成功，可查看对应的日志查看启动失败原因。

1) 格式化 HDFS，命令如下：

```
bin/hadoop namenode -format
```

2) 启动 HDFS。你可以使用以下命令分别启动 NameNode 和 DataNode：

```
sbin/hadoop-daemon.sh start namenode
sbin/hadoop-daemon.sh start datanode
```

如果有多个 DataNode，可使用 `hadoop-daemons.sh` 启动所有 DataNode，具体命令如下：

```
sbin/hadoop-daemons.sh start datanode
```

你也可以使用以下命令一次性启动 NameNode 和所有 DataNode：

```
sbin/ start-dfs.sh
```

3) 启动 YARN。你可以使用以下命令分别启动 ResourceManager 和 NodeManager：

```
sbin/yarn-daemon.sh start resourcemanager
```

```
sbin/yarn-daemon.sh start nodemanager
```

如果有多个 NodeManager，可使用 yarn-daemon.sh 启动所有 NodeManager，具体命令如下：

```
sbin/yarn-daemon.sh start nodemanager
```

你也可以使用以下命令一次性启动 ResourceManager 和所有 NodeManager：

```
sbin/start-yarn.sh
```

通过如下 jps 命令查看是否启动成功：

```
dong@YARN001:/opt/hadoop/hadoop-2.0$ jps
27577 NameNode
30315 ResourceManager
27924 SecondaryNameNode
16803 NodeManager
```

通过以下 URL 可查看 YARN 是否启动成功：

```
http://YARN001:8080/
```

YARN 对外提供的 Web 运行界面如图 1-10 所示。

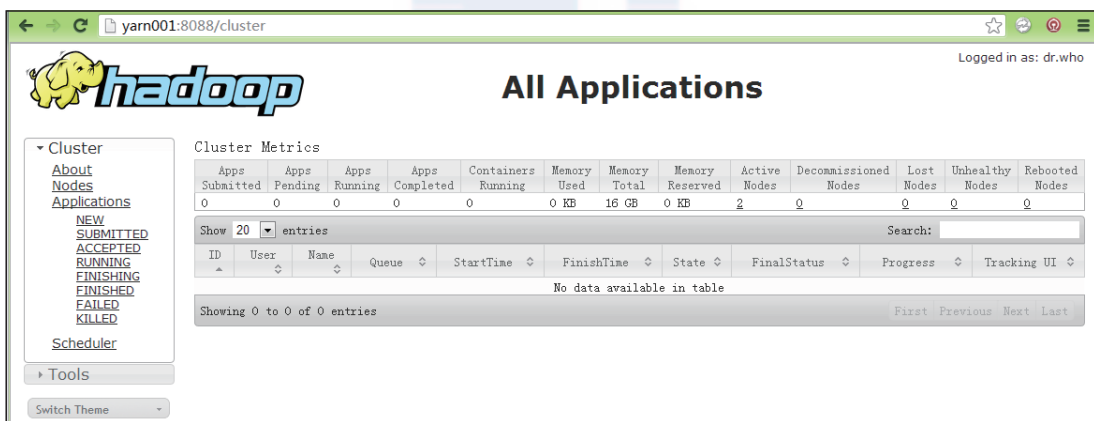


图 1-10 YARN 对外提供的 Web 界面

如果安装过程中出现问题，可通过查看日志发现问题所在。Hadoop 日志存放在 \$HADOOP_HOME/logs 目录下的以“.log”结尾的文件中，比如 yarn-dongxicheng-resourcemanager-yarn001.log 就是 ResourceManager 产生的日志。

经过以上三步操作，Hadoop 成功启动后，接下来可以通过 Hadoop Shell 或者 Eclipse 插件访问 HDFS 和提交 MapReduce 作业。下面两小节分别介绍 Hadoop Shell 和 Eclipse 插件使用方法。

1.5.2 Hadoop Shell 介绍

在 1.4 节我们曾提到，bin 目录下是最基础的集群管理脚本，用户可以通过该脚本完成

各种功能，如 HDFS 文件管理、MapReduce 作业管理等，更加详细的脚本使用说明，可参考附录 C。

作为入门，本节介绍的是 bin 目录下 Hadoop 脚本的使用方法。如果你已经对 Hadoop 1.0 有所了解（比如尝试安装和使用过 Hadoop 1.0），那么可直接使用该脚本，因为该脚本的功能与 Hadoop 1.0 对应的 Hadoop 脚本功能完全一致。

该脚本的使用方法为：

```
hadoop [--config confdir] COMMAND
```

其中，--config 用于设置 Hadoop 配置文件目录。默认目录为 \${HADOOP_HOME}/conf。而 COMMAND 是具体的某个命令，常用的有 HDFS 管理命令 fs、作业管理命令 job 和作业提交命令 jar 等，它们的使用方法如下。

（1）HDFS 管理命令 fs 和作业管理命令 job

它们的用法一样，均为：

```
bin/hadoop command [genericOptions] [commandOptions]
```

其中，command 可以是 fs 或者 job，genericOptions 是一些通用选项，commandOptions 是 fs 或者 job 附加的命令选项，看下面两个例子。

□ 在 HDFS 上创建一个目录 /test，命令如下：

```
bin/hadoop fs -mkdir /test
```

□ 显示所有 Hadoop 上正在运行的作业，命令如下：

```
bin/hadoop job -list
```

（2）作业提交命令 jar

这个命令的用法是：

```
hadoop jar <jar> [mainClass] args..
```

其中，<jar> 表示 JAR 包名，mainClass 表示 main class 名称，可以不必输入而由 jar 命令自动搜索，args 是 main class 输入参数。举例如下：

```
bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar pi 5 10
```

其中 pi 是 hadoop-mapreduce-examples-*.jar 中一个作业名称，该作业主要功能是采用拟蒙特卡罗法估算圆周率 pi (3.1415926...) 的大小，它有两个整型输入参数：Map Task 数目和样本数目。

其他更多命令，读者可自行查阅 Hadoop 官方设计文档。

1.6 编译及调试 Hadoop 源代码

读者在阅读源代码过程中，可能需要修改部分源代码或者使用调试工具以便跟踪某些变量值变化过程，此时要用到 Hadoop 源代码编译和调试方法。本节将介绍 Hadoop 在伪分

布式模式下的编译和调试方法，其中调试方法主要介绍使用 Eclipse 远程调试工具和打印调试日志两种。

Hadoop 天生支持 Linux 而对其他操作系统（如 Windows）很不友好，本书也鼓励读者直接在 Linux 平台下编译和调试 Hadoop 源代码，因此，本节介绍的内容全部在 Linux 环境下。

1.6.1 编译 Hadoop 源代码

在 Linux 环境下编译源代码之前，需进行以下准备工作：

- 确保安装的 Maven 版本在 3.0.2 以上；
- Protocol Buffers 安装版本为 2.5.0；
- 如果要启用 findbugs，则需确认已经安装了 Findbugs；
- 如果要编译 native code，则需确认安装了 CMake 2.6 或者更新版本；
- 第一次编译代码，需确认可以连接互联网（Maven 要从代码库中下载依赖包）。

Maven 编译命令如表 1-2 所示。

表 1-2 Maven 编译命令

命 令	含 义
mvn clean	清理编译结果
mvn compile [-Pnative]	编译源代码
mvn test [-Pnative]	运行测试程序
mvn package	创建 JAR 包
mvn compile findbugs:findbugs	运行 findbugs
mvn compile checkstyle:checkstyle	运行 checkstyle（检查编程规范）
mvn install	将 JAR 包放到 M2 缓存中
mvn deploy	将 JAR 部署到 Maven 仓库中
mvn package [-Pdist][-Pdocs][-Psrc][-Pnative][-Dtar]	构建发布版
mvn versions:set -DnewVersion=NEWVERSION	修改 Hadoop 版本

如果仅编译生成 JAR 包而无须编译 native code、测试用例和生成文档，可在 Hadoop 安装目录下并输入以下命令（推荐使用该命令编译 Hadoop 源代码）：

```
mvn package -Pdist -DskipTests -Dtar
```

如果编译 JAR 包、native code 并生成文档，可使用以下命令：

```
mvn package -Pdist,native,docs -DskipTests -Dtar
```

每个子模块编译后生成的 JAR 包放到了与源代码目录平级的 target 目录中，比如 ResourceManager 的源代码目录是：

```
${YARN_HOME}/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src
```

它对应生成 JAR 包放在了以下目录中：

```
${YARN_HOME}/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/target
```

如果修改了某个模块的代码，可编译后，将对应的 JAR 包覆盖到 `${HADOOP_HOME}/share/hadoop` 目录中对应的 JAR 包上。

如果仅编译 Hadoop 的某一个子模块，需将该模块依赖的 JAR 包作为它的第三方库引入。一种简单的实现方式是在 Hadoop 安装目录下输入以下命令编译所有源代码：

```
mvn install -DskipTests
```

然后进入子模块目录，编译生成对应的 JAR 包。

1.6.2 调试 Hadoop 源代码

本节介绍两种调试 Hadoop 源代码的方法：利用 Eclipse 远程调试工具和打印调试日志。这两种方法均可以调试伪分布式工作模式和完全分布式工作模式下的 Hadoop。本节主要介绍伪分布式工作模式下的 Hadoop 调试方法。

(1) 利用 Eclipse 进行远程调试

下面以调试 ResourceManager 为例，介绍利用 Eclipse 远程调试的基本方法，这可分两步进行。

步骤 1 调试模式下启动 Hadoop。

在 Hadoop 安装目录下运行如下的 Shell 脚本：

```
export YARN_NODEMANAGER_OPTS="-Xdebug -Xrunjdwp:transport=dt_socket,address=8788,server=y,suspend=y"
sbin/start-all.sh
```

运行了脚本后会看到 Shell 命令行终端显示如下信息：

```
Listening for transport dt_socket at address: 8788
```

此时表明 ResourceManager 处于监听状态，直到收到 debug 确认信息。

步骤 2 设置断点。

在前面新建的 Java 工程“hadoop-2.0”中，找到 ResourceManager 相关代码，并在感兴趣的地方设置一些断点。

步骤 3 在 Eclipse 中调试 Hadoop 程序。

在 Eclipse 的菜单栏中，依次选择“Run”→“Debug Configurations”→“Remote Java Applications”，并按照要求填写远程调试器名称（自己定义一个即可），ResourceManager 所在 host 以及监听端口号等信息，并选择 Hadoop 源代码工程，便可进入调试模式。

调试过程中，ResourceManager 输出的信息被存储到日志文件夹下的 `yarn-XXX-resourcemanager-localhost.log` 文件（XXX 为当前用户名）中，可通过以下命令查看调试过程中打印的日志：


```
tail -f logs/yarn-XXX-resourcemanager-localhost.log
```

(2) 打印 Hadoop 调试日志

Hadoop 使用了 Apache log4j^①作为基本日志库，该日志库将日志分别 5 个级别，分别为 DEBUG、INFO、WARN、ERROR 和 FATAL。这 5 个级别是有顺序的，即 $\text{DEBUG} < \text{INFO} < \text{WARN} < \text{ERROR} < \text{FATAL}$ ，分别用来指定日志信息的重要程度。日志输出规则为：只输出级别不低于设定级别的日志信息，比如若级别设定为 INFO，则 INFO、WARN、ERROR 和 FATAL 级别的日志信息都会输出，但级别比 INFO 低的 DEBUG 则不会输出。

在 Hadoop 源代码中，大部分 Java 文件中存在调试日志（DEBUG 级别日志），但默认情况下，日志级别是 INFO，为了查看更详细的运行状态，可采用以下几种方法打开 DEBUG 日志。

方法 1 使用 Hadoop Shell 命令。

可使用 Hadoop 脚本中的 daemonlog 命令查看和修改某个类的日志级别，比如，可通过以下命令查看 NodeManager 类的日志级别：

```
bin/hadoop daemonlog -getlevel ${nodemanager-host}:8042 \
org.apache.hadoop.yarn.server.nodemanager.NodeManager
```

可通过以下命令将 NodeManager 类的日志级别修改为 DEBUG：

```
bin/hadoop daemonlog -setlevel ${nodemanager-host}:8042 \
org.apache.hadoop.yarn.server.nodemanager.NodeManager DEBUG
```

其中，nodemanager-host 为 NodeManager 服务所在的 host，8042 是 NodeManager 的 HTTP 端口号。

方法 2 通过 Web 界面。

用户可以通过 Web 界面查看和修改某个类的日志级别，比如，可通过以下 URL 修改 NodeManager 类的日志级别：

```
http://${nodemanager-host}:8042/logLevel
```

方法 3 修改 log4j.properties 文件。

以上两种方式只能暂时修改日志级别，当 Hadoop 重启后会被重置，如果要永久性改变日志级别，可在目标节点配置目录下的 log4j.properties 文件中添加以下配置选项：

```
log4j.logger.org.apache.hadoop.yarn.server.nodemanager.NodeManager=DEBUG
```

此外，有时为了专门调试某个 Java 文件，需要把该文件的相关日志输出到一个单独文件中，可在 log4j.properties 中添加以下内容：

```
# 定义输出方式为自定义的 TTOUT
log4j.logger.org.apache.hadoop.yarn.server.nodemanager.NodeManager=DEBUG,TTOUT
# 设置 TTOUT 的输出方式为输出到文件
log4j.appender.TTOUT=org.apache.log4j.FileAppender
# 设置文件路径
```

① Apache log4j 网址：<http://logging.apache.org/log4j/index.html>。


```
log4j.appender.TTOUT.File=${hadoop.log.dir}/NodeManager.log
# 设置文件的布局
log4j.appender.TTOUT.layout=org.apache.log4j.PatternLayout
# 设置文件的格式
log4j.appender.TTOUT.layout.ConversionPattern=%d{ISO8601} %p %c: %m%n
```

这些配置选项会把 NodeManager.java 中的 DEBUG 日志写到日志目录下的 NodeManager.log 文件中。

在阅读源代码的过程中,为了跟踪某个变量值的变化,读者可能需要自己添加一些 DEBUG 日志。在 Hadoop 源代码中,大部分类会定义一个日志打印对象,通过该对象可打印各个级别的日志。比如,在 NodeManager 中用以下代码定义对象 LOG:

```
public static final Log LOG = LogFactory.getLog(NodeManager.class);
```

用户可使用 LOG 对象打印调试日志。比如,可在 NodeManager 的 main 函数首行添加以下代码:

```
LOG.debug("Start to launch NodeManager...");
```

然后重新编译 Hadoop 源代码,并将 org.apache.hadoop.yarn.server.nodemanager.NodeManager 的调试级别修改为 DEBUG,重新启动 Hadoop 后便可以看到该调试信息。

1.7 小结

搭建一个高效的源代码学习环境是深入学习 Hadoop 的良好开端,本章主要内容正是帮助读者搭建一个这样的学习环境。在笔者看来,一个高效的 Hadoop 学习环境至少应该包括源代码阅读环境、Hadoop 使用环境和源代码编译调试环境,而本章正是围绕这三个环境的搭建方法组织的。

本章介绍了 Linux 环境下搭建 Hadoop 源代码阅读环境的方法,在此基础上,进一步介绍了 Hadoop 的基本使用方法,主要涉及 Hadoop Shell 和 Eclipse 插件两种工具的使用。最后介绍了 Hadoop 源代码编译和调试方法,其中,调试方法主要介绍了使用 Eclipse 远程调试和打印调试日志两种。

第2章 YARN 设计理念与基本架构

在第1章，我们介绍了 Hadoop 学习环境的搭建方法，这是学习 Hadoop 需要进行的最基本的准备工作。在这一章中，我们将从设计理念和基本架构方面对 Hadoop YARN 进行介绍，这也属于准备工作的一部分。通过本章的介绍将会为下面几章深入剖析 YARN 内部实现奠定基础。

由于 MRv1 在扩展性、可靠性、资源利用率和多框架等方面存在明显不足，Apache 开始尝试对 MapReduce 进行升级改造，于是诞生了更加先进的下一代 MapReduce 计算框架 MRv2。由于 MRv2 将资源管理模块构建成了一个独立的通用系统 YARN，这直接使得 MRv2 的核心从计算框架 MapReduce 转移为资源管理系统 YARN。在本章中，我们将从背景、设计思想和基本架构等方面对 YARN 框架进行介绍。

2.1 YARN 产生背景

2.1.1 MRv1 的局限性

YARN 是在 MRv1 基础上演化而来的，它克服了 MRv1 中的各种局限性。在正式介绍 YARN 之前，我们先要了解 MRv1 的一些局限性，这可概括为以下几个方面：

- **扩展性差**。在 MRv1 中，JobTracker 同时兼备了资源管理和作业控制两个功能，这成为系统的一个最大瓶颈，严重制约了 Hadoop 集群扩展性。
- **可靠性差**。MRv1 采用了 master/slave 结构，其中，master 存在单点故障问题，一旦它出现故障将导致整个集群不可用。
- **资源利用率低**。MRv1 采用了基于槽位的资源分配模型，槽位是一种粗粒度的资源划分单位，通常一个任务不会用完槽位对应的资源，且其他任务也无法使用这些空闲资源。此外，Hadoop 将槽位分为 Map Slot 和 Reduce Slot 两种，且不允许它们之间共享，常常会导致一种槽位资源紧张而另外一种闲置（比如一个作业刚刚提交时，只会运行 Map Task，此时 Reduce Slot 闲置）。
- **无法支持多种计算框架**。随着互联网高速发展，MapReduce 这种基于磁盘的离线计算框架已经不能满足应用要求，从而出现了一些新的计算框架，包括内存计算框架、流式计算框架和迭代式计算框架等，而 MRv1 不能支持多种计算框架并存。

为了克服以上几个缺点，Apache 开始尝试对 Hadoop 进行升级改造，进而诞生了更加先进的下一代 MapReduce 计算框架 MRv2。正是由于 MRv2 将资源管理功能抽象成了一个独立的通用系统 YARN，直接导致下一代 MapReduce 的核心从单一的计算框架 MapReduce

转移为通用的资源管理系统 YARN。为了让读者更进一步理解以 YARN 为核心的软件栈，我们将之与以 MapReduce 为核心的软件栈进行对比，如图 2-1 所示，在以 MapReduce 为核心的软件栈中，资源管理系统 YARN 是可插拔替换的，比如选择 Mesos 替换 YARN，一旦 MapReduce 接口改变，所有的资源管理系统的实现均需要跟着改变；但以 YARN 为核心的软件栈则不同，所有框架都需要实现 YARN 定义的对外接口以运行在 YARN 之上，这意味着 Hadoop 2.0 可以打造一个以 YARN 为核心的生态系统。

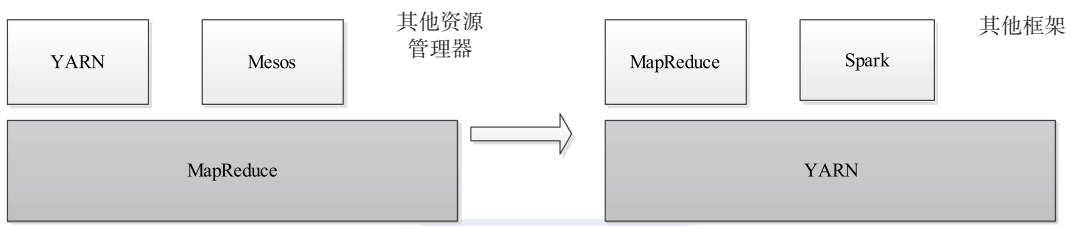


图 2-1 以 MapReduce 为核心和以 YARN 为核心的软件栈对比

2.1.2 轻量级弹性计算平台

随着互联网的高速发展，基于数据密集型应用的计算框架不断出现，从支持离线处理的 MapReduce，到支持在线处理的 Storm，从迭代式计算框架 Spark 到流式处理框架 S4，各种框架诞生于不同的公司或者实验室，它们各有所长，各自解决了某一类应用问题。而在大部分互联网公司中，这几种框架可能同时被采用。比如在搜索引擎公司中，一种可能的技术方案如下：网页建立索引采用 MapReduce 框架，自然语言处理 / 数据挖掘采用 Spark（如网页 PageRank 计算、聚类分类算法等），对性能要求很高的数据挖掘算法用 MPI 等。考虑到资源利用率、运维成本、数据共享等因素，公司一般希望将所有这些框架都部署到一个公共的集群中，让它们共享集群的资源，并对资源进行统一使用，同时采用某种资源隔离方案（如轻量级 cgroups）对各个任务进行隔离，这样便诞生了轻量级弹性计算平台，如图 2-2 所示。YARN 便是弹性计算平台的典型代表。

从上面分析可知，YARN 实际上是一个弹性计算平台，它的目标已经不再局限于支持 MapReduce 一种计算框架，而是朝着对多种框架进行统一管理的发展方向发展。

相比于“一种计算框架一个集群”的模式，共享集群的模式存在多种好处：

- ❑ **资源利用率高。**如图 2-3 所示，如果每个框架一个集群，则往往由于应用程序数量和资源需求的不均衡性，使得在某段时间内，有些计算框架的集群资源紧张，而另外一些集群资源空闲。共享集群模式则通过多种框架共享资源，使得集群中的资源得到更加充分的利用。
- ❑ **运维成本低。**如果采用“一个框架一个集群”的模式，则可能需要多个管理员管理这些集群，进而增加运维成本，而共享模式通常需要少数管理员即可完成多个框架的统一管理。

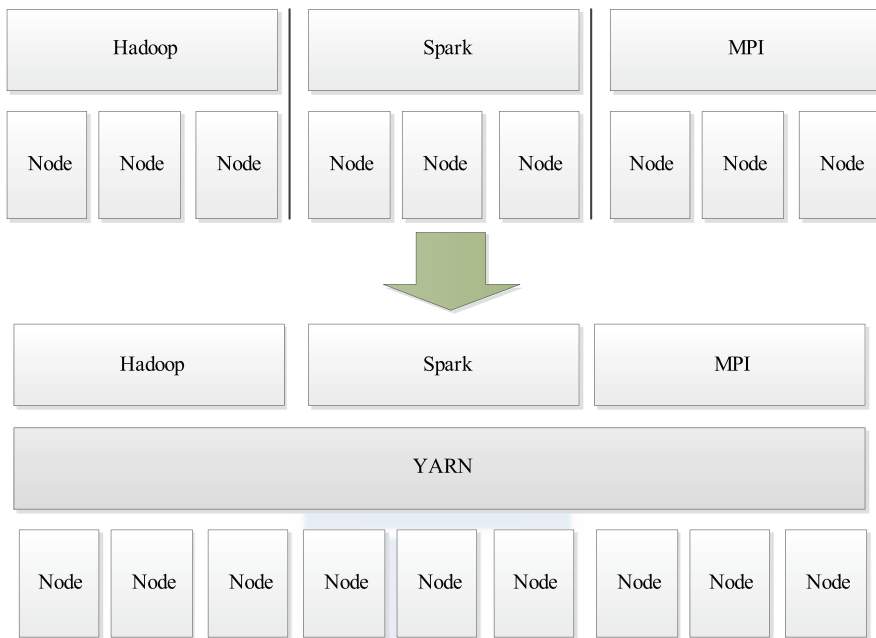


图 2-2 以 YARN 为核心的弹性计算平台的基本架构

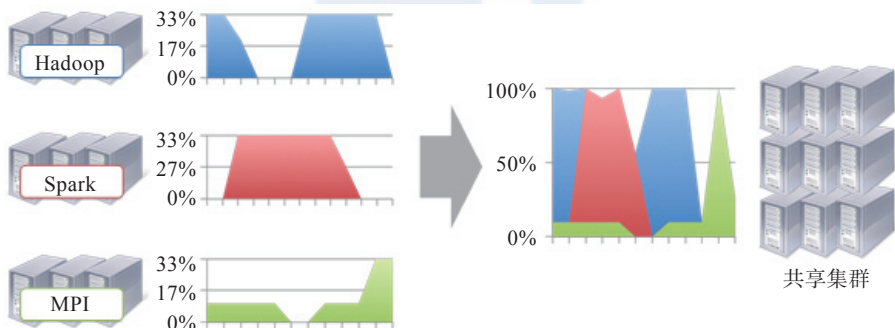


图 2-3 共享集群模式使得资源利用率提高

□ **数据共享。**随着数据量的暴增，跨集群间的数据移动不仅需花费更长的时间，且硬件成本也会大大增加，而共享集群模式可让多种框架共享数据和硬件资源，将大大减小数据移动带来的成本。

2.2 Hadoop 基础知识

2.2.1 术语解释

为了便于本书讲解 Hadoop YARN，本小节对 Hadoop 涉及的术语进行比较全面的介绍。

(1) Hadoop 1.0

Hadoop 1.0 即第一代 Hadoop，由分布式存储系统 HDFS 和分布式计算框架 MapReduce 组成，其中，HDFS 由一个 NameNode 和多个 DataNode 组成，MapReduce 由一个 JobTracker 和多个 TaskTracker 组成，对应 Hadoop 版本为 Apache Hadoop 0.20.x、1.x、0.21.X、0.22.x 和 CDH3^①。

(2) Hadoop 2.0

Hadoop 2.0 即第二代 Hadoop，为克服 Hadoop 1.0 中 HDFS 和 MapReduce 存在的各种问题而提出的。如图 2-4 所示，针对 Hadoop 1.0 中的单 NameNode 制约 HDFS 的扩展性问题，提出了 HDFS Federation，它让多个 NameNode 分管不同的目录进而实现访问隔离和横向扩展，同时它彻底解决了 NameNode 单点故障问题；针对 Hadoop 1.0 中的 MapReduce 在扩展性和多框架支持等方面的不足，它将 JobTracker 中的资源管理和作业控制功能分开，分别由组件 ResourceManager 和 ApplicationMaster 实现，其中，ResourceManager 负责所有应用程序的资源分配，而 ApplicationMaster 仅负责管理一个应用程序，进而诞生了全新的通用资源管理框架 YARN。基于 YARN，用户可以运行各种类型的应用程序（不再像 1.0 那样仅局限于 MapReduce 一类应用），从离线计算的 MapReduce 到在线计算（流式处理）的 Storm 等。Hadoop 2.0 对应 Hadoop 版本为 Apache Hadoop 0.23.x、2.x 和 CDH4^②。

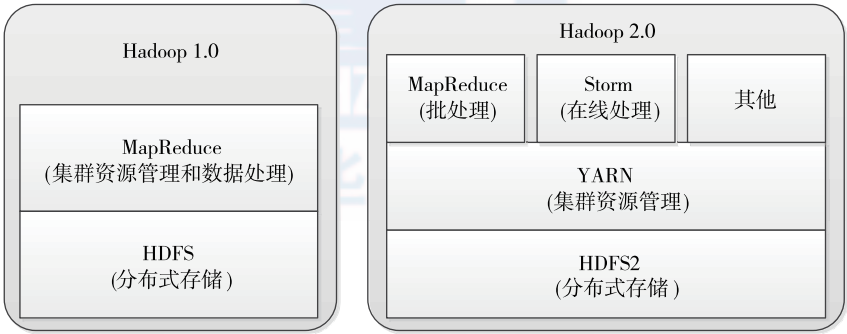


图 2-4 Hadoop 1.0 与 Hadoop 2.0

(3) MapReduce 1.0 或 MRv1

MapReduce 1.0 计算框架主要由三部分组成，分别是编程模型、数据处理引擎和运行时环境。它的基本编程模型是将问题抽象成 Map 和 Reduce 两个阶段，其中 Map 阶段将输入数据解析成 key/value，迭代调用 map() 函数处理后，再以 key/value 的形式输出到本地目录，而 Reduce 阶段则将 key 相同的 value 进行规约处理，并将最终结果写到 HDFS 上；它的数据处理引擎由 MapTask 和 ReduceTask 组成，分别负责 Map 阶段逻辑和 Reduce 阶

① 下载地址为 <http://archive.cloudera.com/cdh/3/>。
② 下载地址为 <http://archive.cloudera.com/cdh4/cdh/4/>。

段逻辑的处理；它的运行时环境由（一个）JobTracker 和（若干个）TaskTracker 两类服务组成，其中，JobTracker 负责资源管理和所有作业的控制，而 TaskTracker 负责接收来自 JobTracker 的命令并执行它。该框架在扩展性、容错性和多框架支持等方面存在不足，这也促使了 MRv2 的产生。

（4）MRv2

MRv2 具有与 MRv1 相同的编程模型和数据处理引擎，唯一不同的是运行时环境。MRv2 是在 MRv1 基础上经加工之后，运行于资源管理框架 YARN 之上的计算框架 MapReduce。它的运行时环境不再由 JobTracker 和 TaskTracker 等服务组成，而是变为通用资源管理系统 YARN 和作业控制进程 ApplicationMaster，其中，YARN 负责资源管理和调度，而 ApplicationMaster 仅负责一个作业的管理。简言之，MRv1 仅是一个独立的离线计算框架，而 MRv2 则是运行于 YARN 之上的 MapReduce。

（5）YARN

YARN 是 Hadoop 2.0 中的资源管理系统，它是一个通用的资源管理模块，可为各类应用程序进行资源管理和调度。YARN 不仅限于 MapReduce 一种框架使用，也可以供其他框架使用，比如 Tez（将在第 9 章介绍）、Spark、Storm（将在第 10 章介绍）等。YARN 类似于几年前的资源管理系统 Mesos^①（将在 12 章介绍）和更早的 Torque^②（将在 6 章介绍）。由于 YARN 的通用性，下一代 MapReduce 的核心已经从简单的支持单一应用的计算框架 MapReduce 转移到通用的资源管理系统 YARN。

（6）HDFS Federation

Hadoop 2.0 中对 HDFS 进行了改进，使 NameNode 可以横向扩展成多个，每个 NameNode 分管一部分目录，进而产生了 HDFS Federation，该机制的引入不仅增强了 HDFS 的扩展性，也使 HDFS 具备了隔离性。

2.2.2 Hadoop 版本变迁

当前 Apache Hadoop 版本非常多，本小节将帮助读者梳理各个版本的特性以及它们之间的联系。在讲解 Hadoop 各版本之前，先要了解 Apache 软件发布方式。对于任何一个 Apache 开源项目，所有的基础特性均被添加到一个称为“trunk”的主代码线（main codeline），当需要开发某个重要的特性时，会专门从主代码线中延伸出一个分支（branch），这被称为一个候选发布版（candidate release），该分支将专注于开发该特性而不再添加其他新的特性，待 bug 修复之后，经过相关人士投票便会对外公开成为发布版（release version），并将该特性合并到主代码线中。需要注意的是，多个分支可能会同时进行研发，这样，版本高的分支可能先于版本低的分支发布。

由于 Apache 以特性为准延伸新的分支，故在介绍 Apache Hadoop 版本之前，先介绍几个独立产生 Apache Hadoop 新版本的重大特性：

① 官方网址：<http://incubator.apache.org/mesos/>。

② 官方网址：<http://www.adaptivecomputing.com/products/open-source/torque/>。

- Append^①：HDFS Append 主要完成追加文件内容的功能，也就是允许用户以 Append 方式修改 HDFS 上的文件。HDFS 最初的一个设计目标是支持 MapReduce 编程模型，而该模型只需要写一次文件，之后仅进行读操作而不会对其修改，即“write-once-read-many”，这就不需要支持文件追加功能。但随着 HDFS 变得流行，一些具有写需求的应用想以 HDFS 作为存储系统，比如，有些应用程序需要往 HDFS 上某个文件中追加日志信息，HBase 需使用 HDFS 具有 Append 功能以防止数据丢失^②等。
- HDFS RAID^③：Hadoop RAID 模块在 HDFS 之上构建了一个新的分布式文件系统 DistributedRaidFileSystem (DRFS)，该系统采用了 Erasure Codes 增强对数据的保护，有了这样的保护，可以采用更低的副本数来保持同样的可用性保障，进而为用户节省大量存储空间。
- Symlink^④：让 HDFS 支持符号链接。符号链接是一种特殊的文件，它以绝对或者相对路径的形式指向另外一个文件或者目录（目标文件），当程序向符号链接中写数据时，相当于直接向目标文件中写数据。
- Security^⑤：Hadoop 的 HDFS 和 MapReduce 均缺乏相应的安全机制，比如在 HDFS 中，用户只要知道某个 block 的 blockID，便可以绕过 NameNode 直接从 DataNode 上读取该 block，用户可以向任意 DataNode 上写 block；在 MapReduce 中，用户可以修改或者杀掉任意其他用户的作业等。为了增强 Hadoop 的安全机制，从 2009 年起，Apache 专门抽出一个团队，从事为 Hadoop 增加基于 Kerberos 和 Deletion Token 的安全认证和授权机制的工作。
- MRv1：正如前面所述，第一代 MapReduce 计算框架由三部分组成：编程模型、数据处理引擎和运行时环境。其中，编程模型由新旧 API 两部分组成；数据处理引擎由 MapTask 和 ReduceTask 组成；运行时环境由 JobTracker 和 TaskTracker 两类服务组成。
- MRv2/YARN^⑥：MRv2 是针对 MRv1 在扩展性和多框架支持等方面的不足而提出来的，它将 MRv1 中的 JobTracker 包含的资源管理和作业控制两部分功能拆分开来，分别将由不同的进程实现。考虑到资源管理模块可以共享给其他框架使用，MRv2 将其做成了一个通用的 YARN 系统，YARN 系统的引入使得计算框架进入了平台化时代。
- NameNode Federation^⑦：针对 Hadoop 1.0 中 NameNode 内存约束限制其扩展性问题

① 0.20-append: <https://issues.apache.org/jira/browse/HDFS-200>。

0.21.0-append: <https://issues.apache.org/jira/browse/HDFS-265>。

② 参考 <http://hbase.apache.org/book/hadoop.html>。

③ 参考 <http://wiki.apache.org/hadoop/HDFS-RAID> 与 <https://issues.apache.org/jira/browse/HDFS-503>。

④ 参考 <https://issues.apache.org/jira/browse/HDFS-245>。

⑤ 参考 <https://issues.apache.org/jira/browse/HADOOP-4487>。

⑥ 参考 <https://issues.apache.org/jira/browse/MAPREDUCE-279>。

⑦ 参考 <https://issues.apache.org/jira/browse/HDFS-1052>。

提出的改进方案，它使 NameNode 可以横向扩展成多个，其中，每个 NameNode 分管一部分目录，这不仅使 HDFS 扩展性得到增强，也使 HDFS 具备了隔离性。

□ NameNode HA^①：大家都知道，HDFS NameNode 存在 NameNode 内存约束限制扩展性和单点故障两个问题，其中，第一个问题通过 NameNode Federation 方案解决，而第二个问题则通过 NameNode 热备方案（NameNode HA）实现。

到 2013 年 8 月为止，Apache Hadoop 已经出现四个大的分支，如图 2-5 所示。

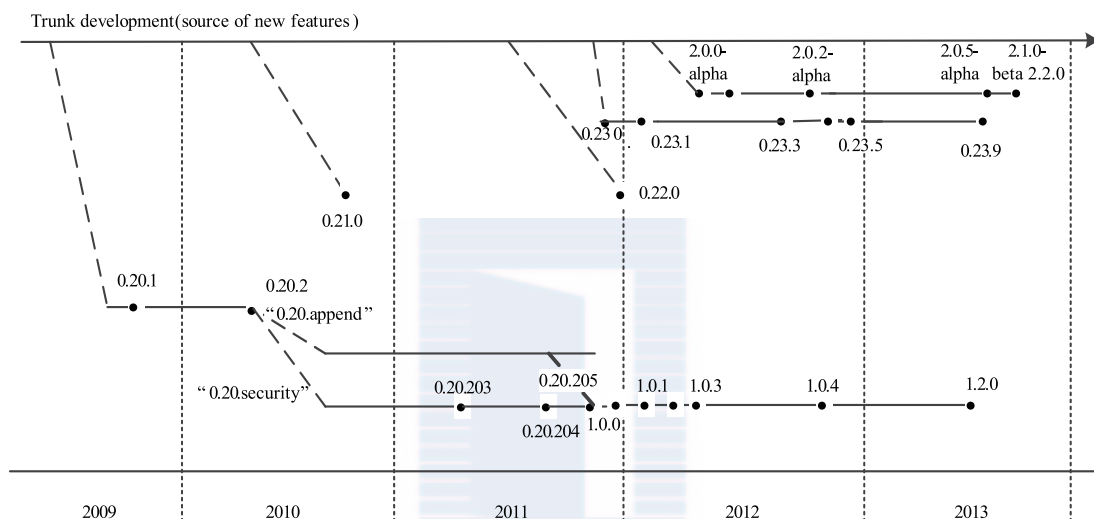


图 2-5 Hadoop 版本变迁图^②

Apache Hadoop 的四大分支构成了三个系列的 Hadoop 版本。

（1）0.20.X 系列

0.20.2 版本发布后，几个重要的特性没有基于 trunk 而是在 0.20.2 基础上继续研发。值得一提的主要有两个特性：Append 与 Security。其中，含 Security 特性的分支以 0.20.203 版本发布，而后续的 0.20.205 版本综合了这两个特性。需要注意的是，之后的 1.0.0 版本仅是 0.20.205 版本的重命名。0.20.X 系列版本是最令用户感到疑惑的，因而它们具有的一些特性，trunk 上没有，反之 trunk 上有一些特性 0.20.X 系列版本却没有。

（2）0.21.0/0.22.x 系列

这一系列版本将整个 Hadoop 项目被分割成三个独立的模块，分别是 Common、HDFS 和 MapReduce。HDFS 和 MapReduce 都对 Common 模块有依赖，但是 MapReduce 对 HDFS 并没有依赖，这样，MapReduce 可以更容易运行在其他的分布式文件系统之上，同时，模块间可以独立开发。具体各个模块的改进如下：

□ Common 模块：最大的新特性是在测试方面添加了 Large-Scale Automated Test

^① 参考 <https://issues.apache.org/jira/browse/HDFS-1623>。

^② 图片修改自 <http://www.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/>。

Framework[⊖]和 fault injection framework[⊖]。

□ **HDFS 模块**：主要增加的新特性包括支持追加操作与建立符号连接、Secondary NameNode 改进（secondary namenode 被剔除，取而代之的是 checkpoint node 同时添加一个 backup node 的角色，作为 NameNode 的冷备）、允许用户自定义 block 放置算法等。

□ **MapReduce 模块**：在作业 API 方面，开始启动新 MapReduce API，但仍然兼容老的 API。

0.22.0 在 0.21.0 基础上修复了一些 bug 并进行了部分优化。

(3) 0.23.X 系列

0.23.X 是为了克服 Hadoop 在扩展性和框架通用性方面的不足而提出来的，它包括基础库 Common、分布式文件系统 HDFS、资源管理框架 YARN 和运行在 YARN 上的 MapReduce 四部分，其中，新增的可对接入的各种计算框架（如 MapReduce、Spark[⊕]等）进行统一管理，该发行版自带 MapReduce 库，而该库集成了迄今为止所有的 MapReduce 新特性。

(4) 2.X 系列

同 0.23.x 系统一样，2.X 系列属于下一代 Hadoop，与 0.23.X 相比，2.X 增加了 NameNode HA 和 Wire-compatibility 等新特性。

表 2-1 总结了 Hadoop 各个发布版的特性以及稳定性。

表 2-1 Hadoop 各个发布版特性以及稳定性

时间	发布版本	特性								是否稳定版本
		Append	RAID	Symlink	Security	MRv1	YARN	NameNode Federation	NameNode HA	
2010 年	0.20.2	×	×	×	×	√	×	×	×	是
	0.21.0	√	√	√	×	√	×	×	√	否
	2.2.x	√	√	×	√	×	√	√	√	是
2011 年	0.20.203	×	×	×	√	√	×	×	×	是 (Yahoo! 在 4500 个节点上部署)
	0.20.205 (1.0.0)	√	×	×	√	√	×	×	×	是
	0.22.0	√	√	√	√ [⊗]	√	×	×	√	否
	0.23.0-alpha	√	√	√	√	×	√	√	×	否

⊖ 参考 <https://issues.apache.org/jira/browse/HADOOP-6332>。

⊖ 参考 <https://issues.apache.org/jira/browse/MAPREDUCE-1084>。

⊕ Spark 是一种内存计算框架，支持迭代式计算，主页是 <http://www.spark-project.org/>。

⊗ 0.22.0 版本中只有 HDFS Security，没有 MapReduce Security。

(续)

时间	发布版本	特性								是否稳定版本
		Append	RAID	Symlink	Security	MRv1	YARN	NameNode Federation	NameNode HA	
2012 年	1.x	√	×	×	√	√	×	×	×	是
	2.x	√	√	√	√	×	√	√	√	否
2013 年	0.23.x-alpha	√	√	√	√	×	√	√	×	否
	2.x-alpha(beta)	√	√	√	√	×	√	√	√	否

本书介绍的 Hadoop YARN 设计思想适用于所有 Apache Hadoop 2.x 版本，但涉及具体的体现（指源代码级别的实现）时，则以 Apache Hadoop 2.2.0 及更高稳定版本为主。

2.3 YARN 基本设计思想

本节我们通过对比两代 MapReduce 的基本框架和编程模型来帮助读者理解 YARN 的基本设计思想。

2.3.1 基本框架对比

在 Hadoop 1.0 中，JobTracker 由资源管理（由 TaskScheduler 模块实现）和作业控制（由 JobTracker 中多个模块共同实现）两部分组成，具体如图 2-6 所示。当前 Hadoop MapReduce 之所以在可扩展性、资源利用率和多框架支持等方面存在不足，正是由于 Hadoop 对 JobTracker 赋予的功能过多而造成负载过重。此外，从设计角度上看，Hadoop 未能够将资源管理相关的功能与应用程序相关的功能分开，造成 Hadoop 难以支持多种计算框架。

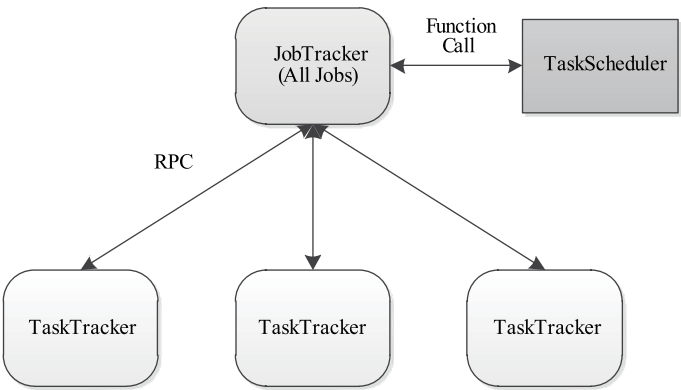


图 2-6 第一代 MapReduce 框架基本架构

下一代 MapReduce 框架的基本设计思想是将 JobTracker 的两个主要功能，即资源管理

和作业控制（包括作业监控、容错等），分拆成两独立的进程，如图 2-7 所示。资源管理进程与具体应用程序无关，它负责整个集群的资源（内存、CPU、磁盘等）管理，而作业控制进程则是直接与应用程序相关的模块，且每个作业控制进程只负责管理一个作业。这样，通过将原有 JobTracker 中与应用程序相关和无关的模块分开，不仅减轻了 JobTracker 负载，也使得 Hadoop 支持更多的计算框架。

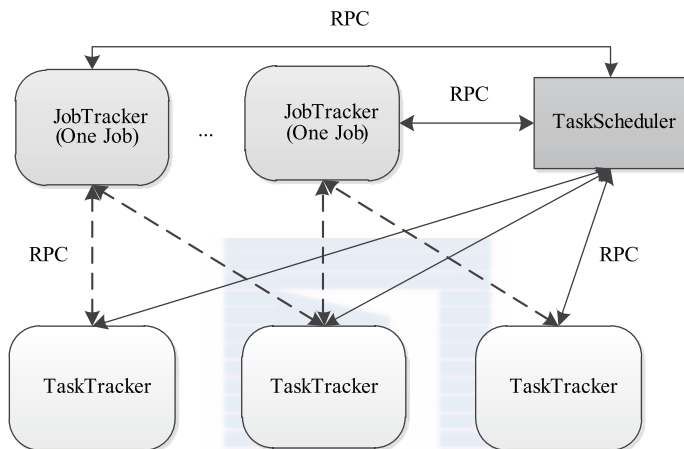


图 2-7 下一代 MapReduce 框架基本架构

从资源管理角度看，下一代 MapReduce 框架实际上衍生出了一个资源统一管理平台 YARN，它使得 Hadoop 不再局限于仅支持 MapReduce 一种计算模型，而是可无限融入多种计算框架，且对这些框架进行统一管理和调度。

2.3.2 编程模型对比

前面提到 MRv1 主要由编程模型（由新旧 API 组成）、数据处理引擎（由 MapTask 和 ReduceTask 组成）和运行时环境（由一个 JobTracker 和若干个 TaskTracker 组成）三部分组成，为了保证编程模型的向后兼容性，MRv2 重用了 MRv1 中的编程模型和数据处理引擎，但运行时环境被完全重写，具体如下。

- **编程模型与数据处理引擎**：MRv2 重用了 MRv1 中的编程模型和数据处理引擎。为了能够让用户应用程序平滑迁移到 Hadoop 2.0 中，MRv2 应尽可能保证编程接口的向后兼容性，但由于 MRv2 本身进行了改进和优化，它在向后兼容性方面存在少量问题。MapReduce 应用程序编程接口有两套，分别是新 API（mapred）和旧 API（mapreduce）^①，MRv2 可做到以下兼容性：采用 MRv1 旧 API 编写的应用程序，可直接使用之前的 JAR 包将程序运行在 MRv2 上；但采用 MRv1 新 API 编写的应用程序

① MapReduce 新旧 API 介绍可参考《Hadoop 技术内幕：深入解析 MapReduce 架构设计与实现原理》一书中的第 3 章。

序则不可以，需要使用 MRv2 编程库重新编译并修改不兼容的参数和返回值，具体将在第 8 章介绍。

□ **运行时环境**：MRv1 的运行时环境主要由两类服务组成，分别是 JobTracker 和 TaskTracker。其中，JobTracker 负责资源和任务的管理与调度，TaskTracker 负责单个节点的资源管理和任务执行。MRv1 将资源管理和应用程序管理两部分混杂在一起，使得它在扩展性、容错性和多框架支持等方面存在明显缺陷。而 MRv2 则通过将资源管理和应用程序管理两部分剥离开，分别由 YARN 和 ApplicationMaster 负责，其中，YARN 专管资源管理和调度，而 ApplicationMaster 则负责与具体应用程序相关的任务切分、任务调度和容错等，具体如图 2-8 所示。

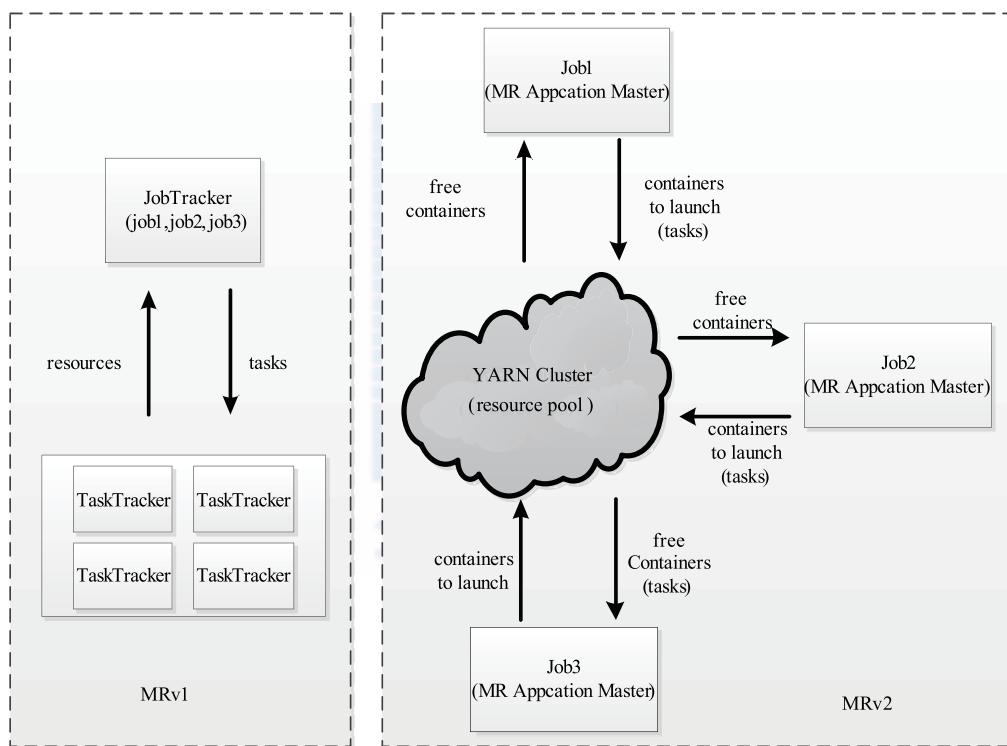


图 2-8 下一代 MapReduce 框架基本架构

2.4 YARN 基本架构

YARN 是 Hadoop 2.0 中的资源管理系统，它的基本设计思想是将 MRv1 中的 JobTracker 拆分成了两个独立的服务：一个全局的资源管理器 ResourceManager 和每个应用程序特有的 ApplicationMaster。其中 ResourceManager 负责整个系统的资源管理和分配，而 ApplicationMaster 负责单个应用程序的管理。

2.4.1 YARN 基本组成结构

YARN 总体上仍然是 Master/Slave 结构，在整个资源管理框架中，ResourceManager 为 Master，NodeManager 为 Slave，ResourceManager 负责对各个 NodeManager 上的资源进行统一管理和调度。当用户提交一个应用程序时，需要提供一个用以跟踪和管理这个程序的 ApplicationMaster，它负责向 ResourceManager 申请资源，并要求 NodeManager 启动可以占用一定资源的任务。由于不同的 ApplicationMaster 被分布到不同的节点上，因此它们之间不会相互影响。在本小节中，我们将对 YARN 的基本组成结构进行介绍。

图 2-9 描述了 YARN 的基本组成结构，YARN 主要由 ResourceManager、NodeManager、ApplicationMaster（图中给出了 MapReduce 和 MPI 两种计算框架的 ApplicationMaster，分别为 MR AppMstr 和 MPI AppMstr）和 Container 等几个组件构成。

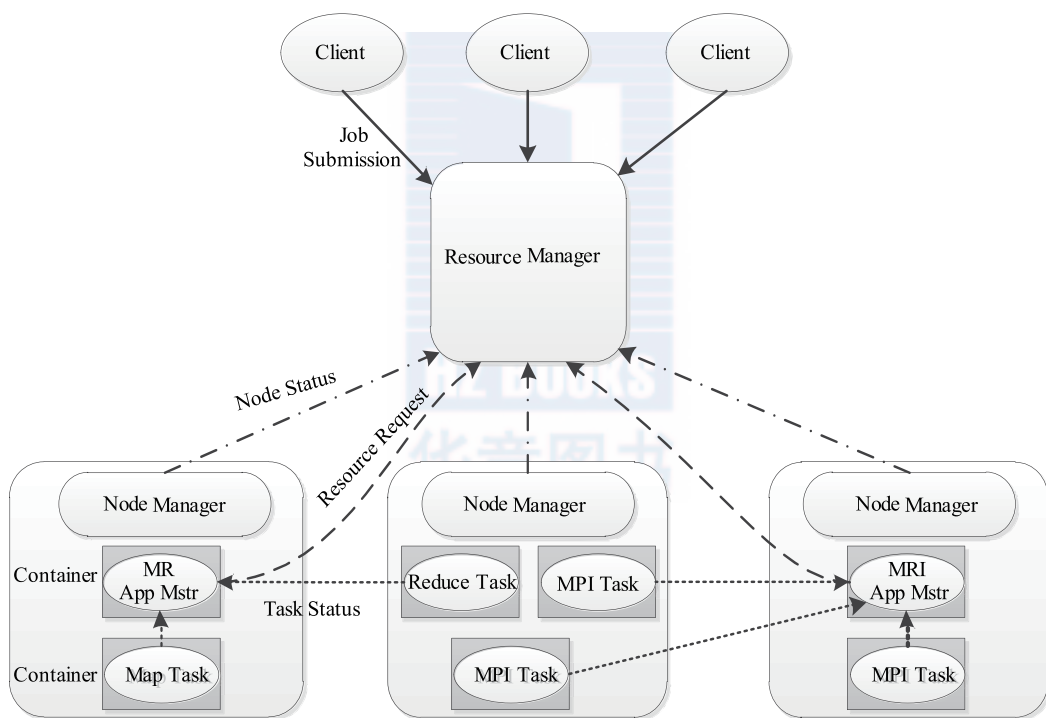


图 2-9 Apache YARN 的基本架构

1. ResourceManager (RM)

RM 是一个全局的资源管理器，负责整个系统的资源管理和分配。它主要由两个组件构成：调度器 (Scheduler) 和应用程序管理器 (Applications Manager, ASM)。

(1) 调度器

调度器根据容量、队列等限制条件（如每个队列分配一定的资源，最多执行一定数量

的作业等), 将系统中的资源分配给各个正在运行的应用程序。需要注意的是, 该调度器是一个“纯调度器”, 它不再从事任何与具体应用程序相关的工作, 比如不负责监控或者跟踪应用的执行状态等, 也不负责重新启动因应用执行失败或者硬件故障而产生的失败任务, 这些均交由应用程序相关的 ApplicationMaster 完成。调度器仅根据各个应用程序的资源需求进行资源分配, 而资源分配单位用一个抽象概念“资源容器”(Resource Container, 简称 Container) 表示, Container 是一个动态资源分配单位, 它将内存、CPU、磁盘、网络等资源封装在一起, 从而限定每个任务使用的资源量。此外, 该调度器是一个可插拔的组件, 用户可根据自己的需要设计新的调度器, YARN 提供了多种直接可用的调度器, 比如 Fair Scheduler 和 Capacity Scheduler 等。

(2) 应用程序管理器

应用程序管理器负责管理整个系统中所有应用程序, 包括应用程序提交、与调度器协商资源以启动 ApplicationMaster、监控 ApplicationMaster 运行状态并在失败时重新启动它等。

2. ApplicationMaster (AM)

用户提交的每个应用程序均包含一个 AM, 主要功能包括:

- 与 RM 调度器协商以获取资源 (用 Container 表示);
- 将得到的任务进一步分配给内部的任务;
- 与 NM 通信以启动 / 停止任务;
- 监控所有任务运行状态, 并在任务运行失败时重新为任务申请资源以重启任务。

当前 YARN 自带了两个 AM 实现, 一个是用于演示 AM 编写方法的实例程序 distributedshell, 它可以申请一定数目的 Container 以并行运行一个 Shell 命令或者 Shell 脚本; 另一个是运行 MapReduce 应用程序的 AM——MRAppMaster, 我们将在第 8 章对其进行介绍。此外, 一些其他的计算框架对应的 AM 正在开发中, 比如 Open MPI、Spark 等^①。

3. NodeManager (NM)

NM 是每个节点上的资源和任务管理器, 一方面, 它会定时地向 RM 汇报本节点上的资源使用情况和各个 Container 的运行状态; 另一方面, 它接收并处理来自 AM 的 Container 启动 / 停止等各种请求。

4. Container

Container 是 YARN 中的资源抽象, 它封装了某个节点上的多维度资源, 如内存、CPU、磁盘、网络等, 当 AM 向 RM 申请资源时, RM 为 AM 返回的资源便是用 Container 表示的。YARN 会为每个任务分配一个 Container, 且该任务只能使用该 Container 中描述的资源。需要注意的是, Container 不同于 MRv1 中的 slot, 它是一个动态资源划分单位, 是根据应用程序的需求动态生成的。截至本书完成时, YARN 仅支持 CPU 和内存两种资源, 且使用了轻量级资源隔离机制 Cgroups 进行资源隔离^②。

^① 参见网址 <http://wiki.apache.org/hadoop/PoweredByYarn>。

^② 参见网址 <https://issues.apache.org/jira/browse/YARN-3>。

2.4.2 YARN 通信协议

RPC 协议是连接各个组件的“大动脉”，了解不同组件之间的 RPC 协议有助于我们更深入地学习 YARN 框架。在 YARN 中，任何两个需相互通信的组件之间仅有一个 RPC 协议，而对于任何一个 RPC 协议，通信双方有一端是 Client，另一端为 Server，且 Client 总是主动连接 Server 的，因此，YARN 实际上采用的是拉式（pull-based）通信模型。如图 2-10 所示，箭头指向的组件是 RPC Server，而箭头尾部的组件是 RPC Client，YARN 主要由以下几个 RPC 协议组成^①：

- ❑ JobClient（作业提交客户端）与 RM 之间的协议——ApplicationClientProtocol：JobClient 通过该 RPC 协议提交应用程序、查询应用程序状态等。
- ❑ Admin（管理员）与 RM 之间的通信协议——ResourceManagerAdministrationProtocol：Admin 通过该 RPC 协议更新系统配置文件，比如节点黑白名单、用户队列权限等。
- ❑ AM 与 RM 之间的协议——ApplicationMasterProtocol：AM 通过该 RPC 协议向 RM 注册和撤销自己，并为各个任务申请资源。
- ❑ AM 与 NM 之间的协议——ContainerManagementProtocol：AM 通过该 RPC 要求 NM 启动或者停止 Container，获取各个 Container 的使用状态等信息。
- ❑ NM 与 RM 之间的协议——ResourceTracker：NM 通过该 RPC 协议向 RM 注册，并定时发送心跳信息汇报当前节点的资源使用情况和 Container 运行情况。

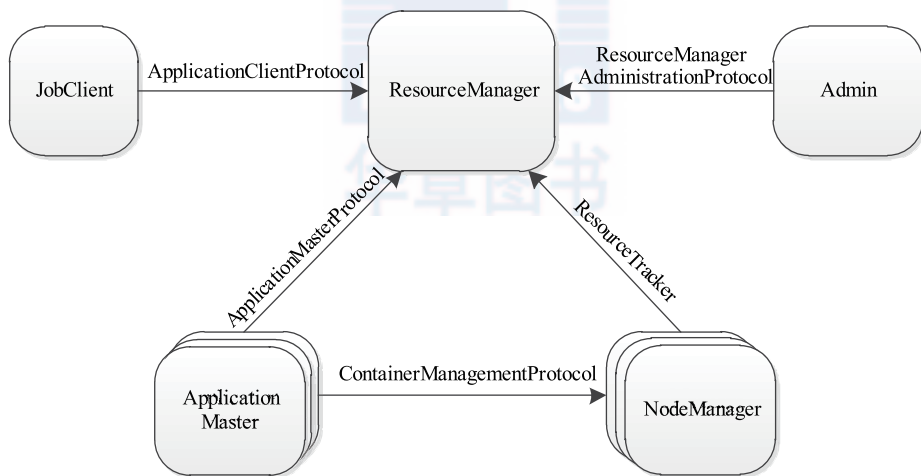


图 2-10 Apache YARN 的 RPC 协议

为了提高 Hadoop 的向后兼容性和不同版本之间的兼容性，YARN 中的序列化框架采用了 Google 开源的 Protocol Buffers。Protocol Buffers 的引入使得 YARN 具有协议向后兼容性，相关内容将在第 3 章介绍。

^① RPC 协议名称在 2.1.0-beta 版本进行了重构，之前的名称分别为：ClientRMProtocol、RMAdminProtocol、AMRMProtocol、ContainerManager、ResourceTracker（该协议名称未变）。

2.5 YARN 工作流程

运行在 YARN 上的应用程序主要分为两类：短应用程序和长应用程序，其中，短应用程序是指一定时间内（可能是秒级、分钟级或小时级，尽管天级别或者更长时间的也存在，但非常少）可运行完成并正常退出的应用程序，比如 MapReduce 作业（将在第 8 章介绍）、Tez DAG 作业（将在第 9 章介绍）等，长应用程序是指不出意外，永不终止运行的应用程序，通常是一些服务，比如 Storm Service（主要包括 Nimbus 和 Supervisor 两类服务），HBase Service（包括 Hmaster 和 RegionServer 两类服务）^①等，而它们本身作为一个框架提供了编程接口供用户使用。尽管这两类应用程序作用不同，一类直接运行数据处理程序，一类用于部署服务（服务之上再运行数据处理程序），但运行在 YARN 上的流程是相同的。

当用户向 YARN 中提交一个应用程序后，YARN 将分两个阶段运行该应用程序：第一个阶段是启动 ApplicationMaster；第二个阶段是由 ApplicationMaster 创建应用程序，为它申请资源，并监控它的整个运行过程，直到运行完成。如图 2-11 所示，YARN 的工作流程分为以下几个步骤：

步骤 1 用户向 YARN 中提交应用程序，其中包括 ApplicationMaster 程序、启动 ApplicationMaster 的命令、用户程序等。

步骤 2 ResourceManager 为该应用程序分配第一个 Container，并与对应的 NodeManager 通信，要求它在这个 Container 中启动应用程序的 ApplicationMaster。

步骤 3 ApplicationMaster 首先向 ResourceManager 注册，这样用户可以直接通过 ResourceManager 查看应用程序的运行状态，然后它将为各个任务申请资源，并监控它的运行状态，直到运行结束，即重复步骤 4~7。

步骤 4 ApplicationMaster 采用轮询的方式通过 RPC 协议向 ResourceManager 申请和领取资源。

步骤 5 一旦 ApplicationMaster 申请到资源后，便与对应的 NodeManager 通信，要求它启动任务。

步骤 6 NodeManager 为任务设置好运行环境（包括环境变量、JAR 包、二进制程序等）后，将任务启动命令写到一个脚本中，并通过运行该脚本启动任务。

步骤 7 各个任务通过某个 RPC 协议向 ApplicationMaster 汇报自己的状态和进度，以让 ApplicationMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务。

在应用程序运行过程中，用户可随时通过 RPC 向 ApplicationMaster 查询应用程序的当前运行状态。

步骤 8 应用程序运行完成后，ApplicationMaster 向 ResourceManager 注销并关闭自己。

① 关于“HBase On YARN”可阅读 <http://hortonworks.com/blog/hoya-hbase-on-yarn-application-architecture/>。

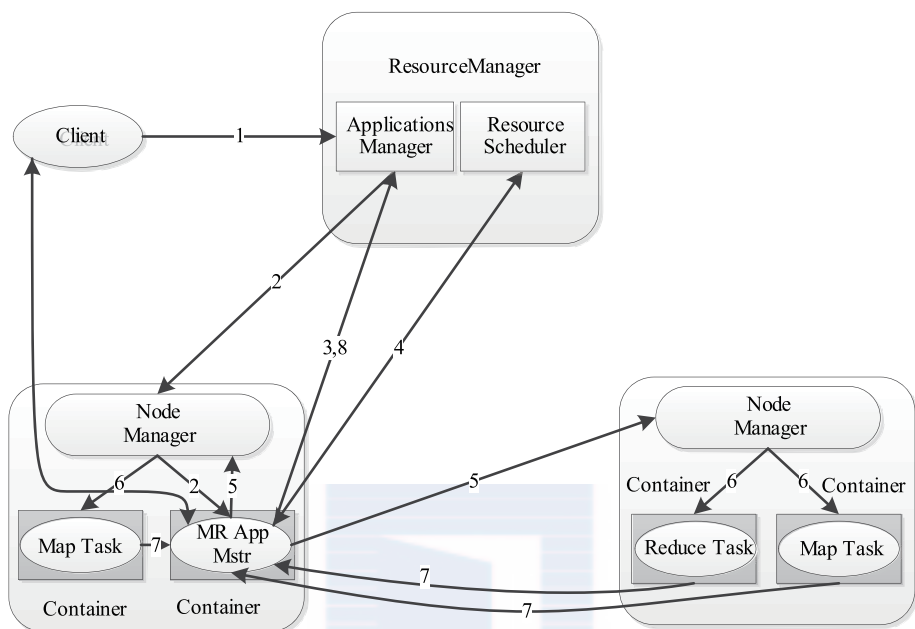


图 2-11 Apache YARN 的工作流程

2.6 多角度理解 YARN

下面我从并行编程、资源管理、云计算等三个角度帮助读者理解 YARN。

2.6.1 并行编程

在单机程序设计中，为了快速处理一个大的数据集，通常采用多线程并行编程，如图 2-12 所示，大体流程如下：先由操作系统启动一个主线程，由它负责数据切分、任务分配、子线程启动和销毁等工作，而各个子线程只负责计算自己的数据，当所有子线程处理完数据后，主线程再退出。类比理解，YARN 上的应用程序运行过程与之非常相近，只不过它是集群上的分布式并行编程。可将 YARN 看做一个云操作系统，它负责为应用程序启动 ApplicationMaster（相当于主线程），然后再由 ApplicationMaster 负责数据切分、任务分配、启动和监控等工作，而由 ApplicationMaster 启动的各个 Task（相当于子线程）仅负责自己的计算任务。当所有任务计算完成后，ApplicationMaster 认为应用程序运行完成，然后退出。

2.6.2 资源管理系统

资源管理系统的主要功能是对集群中各类资源进行抽象，并根据各种应用程序或者服务的要求，按照一定的调度策略，将资源分配给它们使用，同时需采用一定的资源隔离机制防止应用程序或者服务之间因资源抢占而相互干扰。YARN 正是一个资源管理系统，它

的出现弱化了计算框架之争，引入 YARN 这一层后，各种计算框架可各自发挥自己的优势，并由 YARN 进行统一管理，进而运行在一个大集群上。截至本书出版时，各种开源系统都在开发 YARN 版本，包括 MapReduce、Spark、Storm、HBase 等。

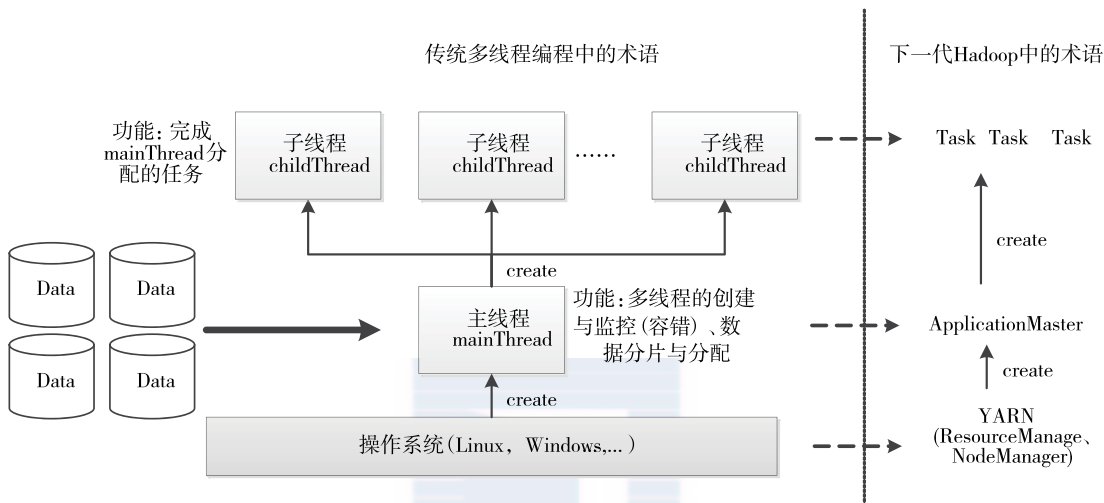


图 2-12 从并行编程角度理解 YARN

2.6.3 云计算

普遍认为，云计算包括以下几个层次的服务：IaaS、PaaS 和 SaaS。这里所谓的层次，是分层体系架构意义上的“层次”。IaaS、PaaS、SaaS 分别实现在基础设施层、软件开放运行平台层、应用软件层。

IaaS(Infrastructure-as-a-Service)：基础设施即服务。消费者通过 Internet 可以从完善的计算机基础设施获得服务。IaaS 通过网络向用户提供计算机（物理机和虚拟机）、存储空间、网络连接、负载均衡和防火墙等基本计算资源；用户在此基础上部署和运行各种软件，包括操作系统和应用程序等。

PaaS(Platform-as-a-Service)：平台即服务。PaaS 是将软件研发的平台作为一种服务，以 SaaS 的模式提交给用户。平台通常包括操作系统、编程语言的运行环境、数据库和 Web 服务器等，用户可以在平台上部署和运行自己的应用。通常而言，用户不能管理和控制底层的基础设施，只能控制自己部署的应用。

SaaS(Software-as-a-Service)：软件即服务。它是一种通过 Internet 提供软件的模式，用户无需购买软件，而是向提供商租用基于 Web 的软件，来管理企业经营活动。云提供商在云端安装和运行应用软件，云用户通过云客户端（比如 Web 浏览器）使用软件。

从云计算分层概念上讲，YARN 可看做 PAAS 层，它能够为不同类型的应用程序提供统一的管理和调度。

2.7 本书涉及内容

本书的主要内容可用图 2-13 表示，主要涉及两部分内容，一个是 YARN 涉及的理念与实现，该部分内容将在第二部分中介绍。另一个是运行于 YARN 之上的比较有名的开源框架，包括 MapReduce、Tez、Storm 和 Spark，其中 MapReduce、Tez 和 Spark 是以短应用程序的形式直接运行在 YARN 之上；而 Storm 则不同，它是以服务的形式运行在 YARN 上，用户编写的 Topology（即为 Storm 应用程序，类似于 MapReduce 作业，将在第 10 章介绍）则运行在 Storm 服务中，该部分内容将在第三部分中介绍。

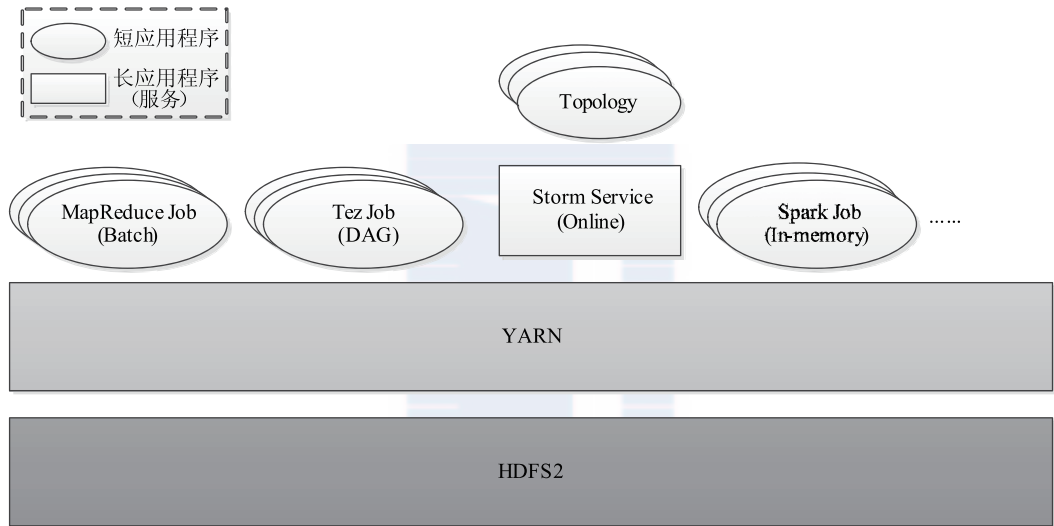


图 2-13 基于 YARN 的动态轻量级弹性计算平台

2.8 小结

本章介绍了 YARN 的设计理念和基本架构，涉及到的内容较多，包括 YARN 产生背景、Hadoop 术语解释和版本变迁、YARN 架构和通信协议等。从编程模型角度看，YARN 与传统并行编程模式非常像，但兼具了分布式和并行两个特点；从资源管理系统角度看，YARN 将扮演为上层计算框架提供计算资源的角色；从云计算角度看，YARN 可看做轻量级的 PAAS 层。

在后面几章中，我们将深入探讨 YARN 内部实现原理，以便让读者进一步深层次理解 YARN。



第二部分

YARN 核心设计篇

YARN 是一个通用资源管理系统，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。本书第二部分将从底层基础库、应用程序编程接口、运行时环境等方面深入剖析 YARN 的内部原理和实现，读者通过对这部分的学习应可编写出运行在 YARN 上的应用程序，且能根据实际项目需求完成对 YARN 的二次开发。

第3章 YARN 基础库

与 MRv1 的实现相比，YARN 的实现要复杂得多。YARN 借用了 MRv1 的一些底层基础库（如 RPC 库），因为引入了很多新的软件设计方式，它的基础库更多，例如直接使用开源序列化框架 Protocol Buffers 和 Apache Avro，及自定义的服务库、事件库和状态机等。

本章介绍的 YARN 基础库是理解后面几章内容的基础，重要性不言而喻。其中有些基础库是开源的，并且被广泛使用，读者可根据自己的情况选择性阅读。

3.1 概述

YARN 基础库是其他一切模块的基础，它的设计直接决定了 YARN 的稳定性和扩展性，概括起来，YARN 的基础库主要有以下几个。

- **Protocol Buffers**：Protocol Buffers 是 Google 开源的序列化库，具有平台无关、高性能、兼容性好等优点。YARN 将 Protocol Buffers 用到了 RPC 通信中，默认情况下，YARN RPC 中所有参数采用 Protocol Buffers 进行序列化 / 反序列化，相比于 MRv1 中基于自定义 Writable 框架的方式，YARN 在向后兼容性、扩展性等方面提高了很多。
- **Apache Avro**：Avro 是 Hadoop 生态系统中的 RPC 框架，具有平台无关、支持动态模式（无需编译）等优点，Avro 的最初设计动机是解决 YARN RPC 兼容性和扩展性差等问题，目前，YARN 采用 Avro 记录 MapReduce 应用程序日志（用于故障后应用程序恢复），今后可能代替 Protocol Buffers 作为 RPC 辅助库（至少会作为一个可选方案）。
- **RPC 库**：YARN 仍采用了 MRv1 中的 RPC 库，但其中采用的默认序列化方法被替换成了 Protocol Buffers。
- **服务库和事件库**：YARN 将所有的对象服务化，以便统一管理（比创建、销毁等），而服务之间则采用事件机制进行通信，不再使用类似 MRv1 中基于函数调用的方式。
- **状态机库**：状态机是一种表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。在 YARN 中，很多对象都是由若干状态组成的，且当有事件发生时，状态之间会发生转移，比如作业、任务、Container 等，而 YARN 正是采用有限状态机描述一些对象的状态以及状态之间的转移。引入状态机模型后，相比 MRv1，YARN 的代码结构更加清晰易懂了。

理解以上这几个库是阅读后面几章的基础，本章将详细介绍这几个库。

3.2 第三方开源库

3.2.1 Protocol Buffers

Protocol Buffers^①是一种轻便高效的结构化数据存储格式，可以用于结构化数据序列化/反序列化。它很适合做数据存储或 RPC 的数据交换格式，常用作通信协议、数据存储等领域的与语言无关、平台无关、可扩展的序列化结构数据格式。目前支持 C++、Java、Python 三种语言。在 Google 内部，几乎所有的 RPC 协议和文件格式都是采用 Protocol Buffers。

相比于常见的 XML 格式，Protocol Buffers 官方网站这样描述它的优点：

- 平台无关、语言无关；
- 高性能，解析速度是 XML 的 20 ~ 100 倍；
- 体积小，文件大小仅是 XML 的 1/10 ~ 1/3；
- 使用简单；
- 兼容性好。

通常编写一个 Protocol Buffers 应用需要以下三步：

- 1) 定义消息格式文件，通常以 proto 作为扩展名；
- 2) 使用 Google 提供的 Protocol Buffers 编译器生成特定语言（目前支持 C++、Java、Python 三类语言）的代码文件；
- 3) 使用 Protocol Buffers 库提供的 API 来编写应用程序。

为了说明 Protocol Buffers 的使用方法，下面给出一个简单的实例。

该实例中首先定义一个消息格式文件 person.proto，描述了通讯录中一个人的基本信息，接着用 Protocol Buffers 提供的方法将一个人的信息写入文件。

步骤 1 定义消息格式文件 person.proto，该文件描述了通讯录中某个人的基本信息，内容如下：

```
package tutorial;                                // 自定义的命名空间
option java_package = "com.example.tutorial";    // 生成文件的包名
option java_outer_classname = "PersonProtos";    // 类名

message Person {                                // 待描述的结构化数据
    required string name = 1;                    // required 表示这个字段不能为空
    required int32 id = 2;                        // 数字“2”表示字段的数字别名
    optional string email = 3;                    // optional 表示该字段可以为空

    message PhoneNumber {                        // 内部 message
        required string number = 1;
        optional int32 type = 2;
```

① 参见网址 <http://code.google.com/p/protobuf/>。

```

    }
    repeated PhoneNumber phone = 4;
}

```

步骤 2 使用 Google 提供的 Protocol Buffers 编译器生成 Java 语言，命令如下：

```
protoc -java_out=. person.proto
```

注意，上面的命令运行时的当前路径是 person.proto 所在目录。

步骤 3 使用 Protocol Buffers 库提供的 API 编写应用程序。该例子创建了一个 Person 对象，先将该对象保存到文件 example.txt 中，之后又从文件中读出并打印出来。

```

public class ProtocolBufferExample {
    static public void main(String[] argv) {
        Person person1 = Person.newBuilder()
            .setName("Dong Xicheng")
            .setEmail("dongxicheng@yahoo.com")
            .setId(11111)
            .addPhone(Person.PhoneNumber.newBuilder()
                .setNumber("15110241024")
                .setType(0))
            .addPhone(Person.PhoneNumber.newBuilder()
                .setNumber("01025689654")
                .setType(1)).build();

        try {
            FileOutputStream output = new FileOutputStream("example.txt");
            person1.writeTo(output);
            output.close();
        } catch (Exception e) {
            System.out.println("Write Error! ");
        }

        try {
            FileInputStream input = new FileInputStream("example.txt");
            Person person2 = Person.parseFrom(input);
            System.out.println("person2:" + person2);
        } catch (Exception e) {
            System.out.println("Read Error!");
        }
    }
}

```

在 YARN 中，所有 RPC 函数的参数均采用 Protocol Buffers 定义的，相比 MRv1 中基于 Writable 序列化的方法，Protocol Buffers 的引入使得 YARN 在向后兼容性和性能方面向前迈进了一大步。

除序列化 / 反序列化之外，Protocol Buffers 也提供了 RPC 函数的定义方法，但并未给出具体实现，这需要用户自行实现^①，而 YARN 则采用了 MRv1 中 Hadoop RPC 库，举例如下：

① 可参考第三方开源实现，网址为 <http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns>。

```

service ContainerManagerService { // 这是 YARN 自带的 ContainerManager 协议的定义
    rpc startContainer(StartContainerRequestProto) returns (StartContainerResponseProto);
    rpc stopContainer(StopContainerRequestProto) returns (StopContainerResponseProto);
    rpc getContainerStatus(GetContainerStatusRequestProto) returns (GetContainerStatusResponseProto);
}

```

在第2章中，介绍了 YARN 中的所有 RPC 协议，而这些协议全是使用 Protocol Buffers 定义的，具体如下：

- ❑ `applicationmaster_protocol.proto`：定义了 AM 与 RM 之间的协议——ApplicationMasterProtocol。
- ❑ `applicationclient_protocol.proto`：定义了 JobClient（作业提交客户端）与 RM 之间的协议——ApplicationClientProtocol。
- ❑ `containermanagement_protocol.proto`：定义了 AM 与 NM 之间的协议——ContainerManagementProtocol。
- ❑ `resourcemanager_administration_protocol.proto`：定义了 Admin（管理员）与 RM 之间的通信协议——ResourceManagerAdministrationProtocol。
- ❑ `yarn_protos.proto`：定义了各个协议 RPC 的参数。
- ❑ `ResourceTracker.proto`：定义了 NM 与 RM 之间的协议——ResourceTracker。

除了以上几个内核中的协议，YARN 还使用 Protocol Buffers 对 MapReduce 中的协议进行了重新定义：

- ❑ `MRClientProtocol.proto`：定义了 JobClient（作业提交客户端）与 MRAppMaster 之间的协议——MRClientProtocol。
- ❑ `mr_protos.proto`：定义了 MRClientProtocol 协议的各个参数。

3.2.2 Apache Avro

Apache Avro^①是 Hadoop 下的一个子项目。它本身既是一个序列化框架，同时也实现了 RPC 的功能。

Avro 官网描述 Avro 的特性和功能如下：

- ❑ 丰富的数据结构类型；
- ❑ 快速可压缩的二进制数据形式；
- ❑ 存储持久数据的文件容器；
- ❑ 提供远程过程调用 RPC；
- ❑ 简单的动态语言结合功能。

相比于 Apache Thrift 和 Google 的 Protocol Buffers，Apache Avro 具有以下特点：

- ❑ **支持动态模式**。Avro 不需要生成代码，这有利于搭建通用的数据处理系统，同时避免了代码入侵。

① 参见网址 <http://avro.apache.org/>。

□ **数据无须加标签。**读取数据前, Avro 能够获取模式定义, 这使得 Avro 在数据编码时只需要保留更少的类型信息, 有利于减少序列化后的数据大小。

□ **无须手工分配的域标识。**Thrift 和 Protocol Buffers 使用一个用户添加的整型域唯一性定义一个字段, 而 Avro 则直接使用域名, 该方法更加直观、更加易扩展。

编写一个 Avro 应用也需如下三步:

- 1) 定义消息格式文件, 通常以 avro 作为扩展名;
- 2) 使用 Avro 编译器生成特定语言的代码文件(可选);
- 3) 使用 Avro 库提供的 API 来编写应用程序。

下面给出一个使用实例。

步骤 1 定义消息格式文件 person.avro, 该文件描述了通讯录中某个人的基本信息, 内容如下:

```
{
  "namespace": "com.example.tutorial",
  "type": "record",
  "name": "Person",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "id", "type": "int" },
    { "name": "email", "type": [ "string", "null" ] },
    { "name": "phone", "type": { "type": "array",
      "items": { "type": "record", "name": "PhoneNumber",
        "fields": [
          { "name": "number", "type": "string" },
          { "name": "type", "type": [ "int", "null" ] }
        ]
      }
    }
  ]
}
```

步骤 2 使用 Avro 编译器生成 Java 语言, 命令如下:

```
java -jar avro-tools-1.7.4.jar compile schema person.avro .
```

注意, 上面的命令运行时的当前路径是 person.avro 所在目录。

步骤 3 使用 Avro 库提供的 API 来编写应用程序。该例子创建一个 Person 对象, 先将该对象保存到文件 example.txt 中, 之后从文件中读出并打印。

```
public class AvroExample {
  static public void main(String[] argv) {
    PhoneNumber phoneNumber1 = PhoneNumber.newBuilder()
      .setNumber("15110241024")
      .setType(0).build();
    PhoneNumber phoneNumber2 = PhoneNumber.newBuilder()
      .setNumber("01025689654")
      .setType(1).build();
```

```

List<PhoneNumber> phoneNumbers = new ArrayList<PhoneNumber>();
phoneNumbers.add(phoneNumber1);
phoneNumbers.add(phoneNumber2);

Person person = Person.newBuilder()
    .setName("Dong Xicheng")
    .setEmail("dongxicheng@yahoo.com")
    .setId(11111)
    .setPhone(phoneNumbers).build();
File file = new File("person.txt");
try {
    DatumWriter<Person> personDatumWriter = new SpecificDatumWriter<Person>(Person.class);
    DataFileWriter<Person> dataFileWriter = new DataFileWriter<Person>(personDatumWriter);
    dataFileWriter.create(person.getSchema(), file);
    dataFileWriter.append(person);
    dataFileWriter.close();
} catch (Exception e) {
    System.out.println("Write Error:" + e);
}
try {
    DatumReader<Person> userDatumReader = new SpecificDatumReader<Person>(Person.class);
    DataFileReader<Person> dataFileReader = new DataFileReader<Person>(file, userDatumReader);
    person = null;
    while (dataFileReader.hasNext()) {
        person = dataFileReader.next(person);
        System.out.println(person);
    }
} catch (Exception e) {
    System.out.println("Read Error:" + e);
}
}
}

```

如果不想编译 person.avro 文件，需要使用另外一套应用程序接口，具体可参考官方文档^①。

Apache Avro 最初是为 Hadoop 量身打造的 RPC 框架，考虑到稳定性^②，YARN 暂时采用 Protocol Buffers 作为序列化库，RPC 仍使用 MRv1 中的 RPC，而 Avro 则作为日志序列化库使用（将在第 8 章介绍）。在 YARN MapReduce 中，所有事件的序列化 / 反序列化均采用 Avro 完成，相关定义在 Events.avpr 文件中，举例如下：

```

{"namespace": "org.apache.hadoop.mapreduce.jobhistory",
 "protocol": "Events",
 "types": [
    ...
    {"type": "record", "name": "JobInfoChange",
     "fields": [

```

① 参见网址 <http://avro.apache.org/docs/current/gettingstartedjava.html>。

② YARN 项目启动时，Apache Avro 尚不成熟，存在各种问题。

```

        {"name": "jobid", "type": "string"},
        {"name": "submitTime", "type": "long"},
        {"name": "launchTime", "type": "long"}
    ],
    },
    {"type": "record", "name": "JobPriorityChange",
     "fields": [
        {"name": "jobid", "type": "string"},
        {"name": "priority", "type": "string"}
    ]
    },
    {"type": "record", "name": "JobStatusChanged",
     "fields": [
        {"name": "jobid", "type": "string"},
        {"name": "jobStatus", "type": "string"}
    ]
    },
    ...
]
}

```

3.3 底层通信库

网络通信模块是分布式系统中最底层的模块，它直接支撑了上层分布式环境下复杂的进程间通信（Inter-Process Communication, IPC）逻辑，是所有分布式系统的基础。远程过程调用（Remote Procedure Call, RPC）是一种常用的分布式网络通信协议，它允许运行于一台计算机的程序调用另一台计算机的子程序，同时将网络的通信细节隐藏起来，使得用户无须额外地为这个交互作用编程。由于 RPC 大大简化了分布式程序开发，因此备受欢迎。

作为一个分布式系统，Hadoop 实现了自己的 RPC 通信协议，它是上层多个分布式系统（如 MapReduce、YARN、HDFS 等）公用的网络通信模块。本节首先从框架设计及实现等方面介绍 Hadoop RPC，接着介绍 RPC 框架在 Hadoop YARN 中的应用。

3.3.1 RPC 通信模型

RPC 是一种通过网络从远程计算机上请求服务，但不需要了解底层网络技术的协议。RPC 协议假定某些传输协议（如 TCP 或 UDP 等）已经存在，并通过这些传输协议为通信程序之间传递访问请求或者应答信息。在 OSI 网络通信模型中，RPC 跨越了传输层和应用层。RPC 使得开发分布式应用程序更加容易^①。

RPC 通常采用客户机 / 服务器模型。请求程序是一个客户机，而服务提供程序则是一个服务器。一个典型的 RPC 框架如图 3-1 所示，主要包括以下几个部分：

□ **通信模块**。两个相互协作的通信模块实现请求 - 应答协议，它们在客户和服务器之

① 参见网址 http://en.wikipedia.org/wiki/Remote_procedure_call。

间传递请求和应答消息，一般不会对数据包进行任何处理。请求-应答协议的实现方式有同步方式和异步方式两种。

如图 3-1 所示，同步模式下客户端程序一直阻塞到服务器端发送的应答请求到达本地；而异步模式不同，客户端将请求发送到服务器端后，不必等待应答返回，可以做其他事情，待服务器端处理完请求后，主动通知客户端。在高并发应用场景中，一般采用异步模式以降低访问延迟和提高带宽利用率。

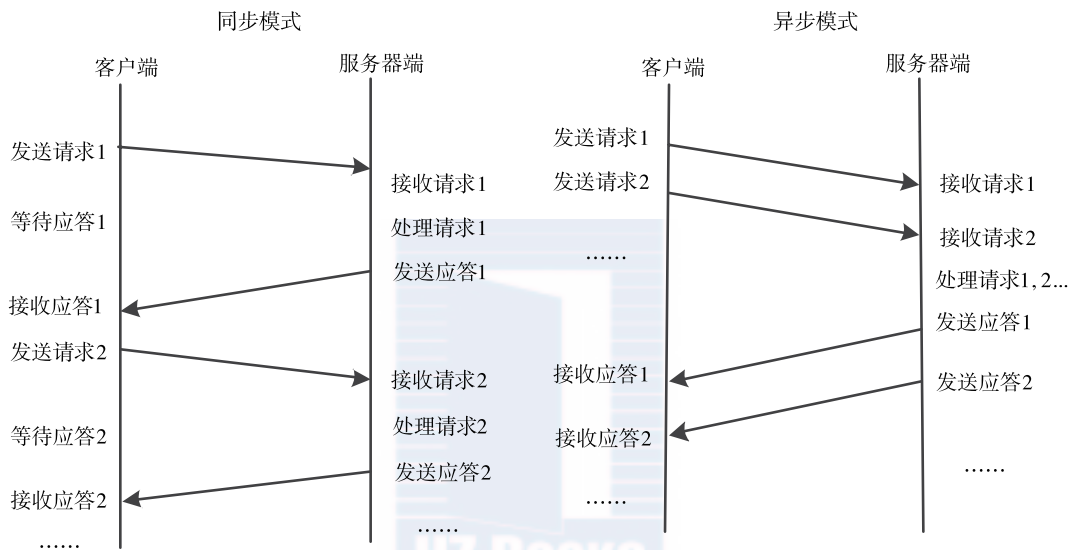


图 3-1 同步模式与异步模式对比

□ **Stub 程序。**客户端和服务端均包含 Stub 程序，可将之看做代理程序。它使得远程函数调用表现得跟本地调用一样，对用户程序完全透明。在客户端，它表现得就像一个本地程序，但不直接执行本地调用，而是将请求信息通过网络模块发送给服务器端。此外，当服务器发送应答后，它会解码对应结果。在服务器端，Stub 程序依次进行解码请求消息中的参数、调用相应的服务过程和编码应答结果的返回值等处理。

□ **调度程序。**调度程序接收来自通信模块的请求消息，并根据其中的标识选择一个 Stub 程序进行处理。通常客户端并发请求量比较大时，会采用线程池提高处理效率。

□ **客户程序 / 服务过程。**请求的发出者和请求的处理者。如果是单机环境，客户程序可直接通过函数调用访问服务过程，但在分布式环境下，需要考虑网络通信，这不得增加通信模块和 Stub 程序（保证函数调用的透明性）。

通常而言，一个 RPC 请求从发送到获取处理结果，所经历的步骤（见图 3-2）下所示。

- 1) 客户程序以本地方式调用系统产生的 Stub 程序；
- 2) 该 Stub 程序将函数调用信息按照网络通信模块的要求封装成消息包，并交给通信

模块发送到远程服务器端。

- 3) 远程服务器端接收此消息后, 将此消息发送给相应的 Stub 程序;
- 4) Stub 程序拆封消息, 形成被调过程要求的形式, 并调用对应函数;
- 5) 被调用函数按照所获参数执行, 并将结果返回给 Stub 程序;
- 6) Stub 程序将此结果封装成消息, 通过网络通信模块逐级地传送给客户程序。

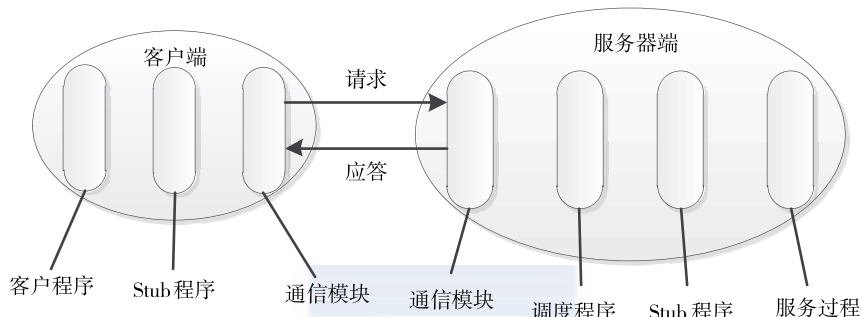


图 3-2 RPC 通用架构

3.3.2 Hadoop RPC 的特点概述

RPC 实际上是分布式计算中 C/S (Client/Server) 模型的一个应用实例, 对于 Hadoop RPC 而言, 它具有以下几个特点。

- **透明性**。这是所有 RPC 框架最根本的特点, 即当用户在一台计算机的程序调用另外一台计算机上的子程序时, 用户自身不应感觉到其间涉及跨机器间的通信, 而是感觉像是在执行一个本地调用。
- **高性能**。Hadoop 各个系统 (如 HDFS、YARN、MapReduce 等) 均采用了 Master/Slave 结构, 其中, Master 实际上是一个 RPC Server, 它负责处理集群中所有 Slave 发送的服务请求, 为了保证 Master 的并发处理能力, RPC Server 应是一个高性能服务器, 能够高效地处理来自多个 Client 的并发 RPC 请求。
- **可控性**。JDK 中已经自带了一个 RPC 框架——RMI (Remote Method Invocation, 远程方法调用), 之所以不直接使用该框架, 主要是考虑到 RPC 是 Hadoop 最底层最核心的模块之一, 保证其轻量级、高性能和可控性显得尤为重要, 而 RMI 重量级过大且用户可控之处太少 (如网络连接、超时和缓冲等均难以定制或者修改)^①。

3.3.3 RPC 总体架构

同其他 RPC 框架一样, Hadoop RPC 主要分为四个部分, 分别是序列化层、函数调用层、网络传输层和服务器端处理框架, 具体实现机制如下:

- **序列化层**。序列化主要作用是将结构化对象转为字节流以便于通过网络进行传输或

^① Doug Cutting 在 Hadoop 最初设计时就是这样描述 Hadoop RPC 设计动机的。

写入持久存储，在 RPC 框架中，它主要用于将用户请求中的参数或者应答转化成字节流以便跨机器传输。前面介绍的 Protocol Buffers 和 Apache Avro 均可用在序列化层，Hadoop 本身也提供了一套序列化框架，一个类只要实现 Writable 接口即可支持对象序列化与反序列化。

- **函数调用层**。函数调用层主要功能是定位要调用的函数并执行该函数，Hadoop RPC 采用了 Java 反射机制与动态代理实现了函数调用。
- **网络传输层**。网络传输层描述了 Client 与 Server 之间消息传输的方式，Hadoop RPC 采用了基于 TCP/IP 的 Socket 机制。
- **服务器端处理框架**。服务器端处理框架可被抽象为网络 I/O 模型，它描述了客户端与服务器端间信息交互方式，它的设计直接决定着服务器端的并发处理能力，常见的网络 I/O 模型有阻塞式 I/O、非阻塞式 I/O、事件驱动 I/O 等，而 Hadoop RPC 采用了基于 Reactor 设计模式的事件驱动 I/O 模型。

Hadoop RPC 总体架构如图 3-3 所示，自下而上可分为两层，第一层是一个基于 Java NIO (New I/O) 实现的客户机-服务器 (C/S) 通信模型。其中，客户端将用户的调用方法及其参数封装成请求包后发送到服务器端。服务器端收到请求包后，经解包、调用函数、打包结果等一系列操作后，将结果返回给客户端。为了增强 Server 端的扩展性和并发处理能力，Hadoop RPC 采用了基于事件驱动的 Reactor 设计模式，在具体实现时，用到了 JDK 提供的各种功能包，主要包括 java.nio (NIO)、java.lang.reflect (反射机制和动态代理)、java.net (网络编程库) 等。第二层是供更上层程序直接调用的 RPC 接口，这些接口底层即为 C/S 通信模型。

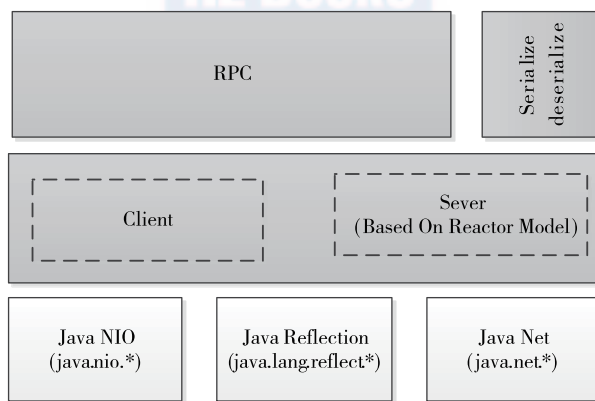


图 3-3 Hadoop RPC 总体架构

3.3.4 Hadoop RPC 使用方法

Hadoop RPC 对外主要提供了两种接口（见类 org.apache.hadoop.ipc.RPC），分别是：

- `public static <T> ProtocolProxy <T> getProxy/waitForProxy(...)`：构造一个客户端代

理对象（该对象实现了某个协议），用于向服务器发送 RPC 请求。

□ `public static Server RPC.Builder (Configuration).build()`：为某个协议（实际上是 Java 接口）实例构造一个服务器对象，用于处理客户端发送的请求。

通常而言，使用 Hadoop RPC 可分为以下 4 个步骤。

1. 定义 RPC 协议

RPC 协议是客户端和服务端之间的通信接口，它定义了服务器端对外提供的服务接口。如下所示，我们定义一个 `ClientProtocol` 通信接口，声明了 `echo()` 和 `add()` 两个方法。需要注意的是，Hadoop 中所有自定义 RPC 接口都需要继承 `VersionedProtocol` 接口，它描述了协议的版本信息。

```
interface ClientProtocol extends org.apache.hadoop.ipc.VersionedProtocol {
    // 版本号，默认情况下，不同版本号的 RPC Client 和 Server 之间不能相互通信
    public static final long versionID = 1L;
    String echo(String value) throws IOException;
    int add(int v1, int v2) throws IOException;
}
```

2. 实现 RPC 协议

Hadoop RPC 协议通常是一个 Java 接口，用户需要实现该接口。对 `ClientProtocol` 接口进行简单的实现如下所示：

```
public static class ClientProtocolImpl implements ClientProtocol {
    // 重载的方法，用于获取自定义的协议版本号，
    public long getProtocolVersion(String protocol, long clientVersion) {
        return ClientProtocol.versionID;
    }
    // 重载的方法，用于获取协议签名
    public ProtocolSignature getProtocolSignature(String protocol, long clientVersion,
        int hashCode) {
        return new ProtocolSignature(ClientProtocol.versionID, null);
    }
    public String echo(String value) throws IOException {
        return value;
    }
    public int add(int v1, int v2) throws IOException {
        return v1 + v2;
    }
}
```

3. 构造并启动 RPC Server

直接使用静态类 `Builder` 构造一个 RPC Server，并调用函数 `start()` 启动该 Server：

```
Server server = new RPC.Builder(conf).setProtocol(ClientProtocol.class)
    .setInstance(new ClientProtocolImpl()).setBindAddress(ADDRESS).setPort(0)
    .setNumHandlers(5).build();
server.start();
```

其中, BindAddress (由函数 setBindAddress 设置) 和 Port (由函数 setPort 设置, 0 表示由系统随机选择一个端口号) 分别表示服务器的 host 和监听端口号, 而 NnumHandlers (由函数 setNumHandlers 设置) 表示服务器端处理请求的线程数目。到此为止, 服务器处理监听状态, 等待客户端请求到达。

4. 构造 RPC Client 并发送 RPC 请求

使用静态方法 getProxy 构造客户端代理对象, 直接通过代理对象调用远程端的方法, 具体如下所示:

```
proxy = (ClientProtocol)RPC.getProxy(
    ClientProtocol.class, ClientProtocol.versionID, addr, conf);
int result = proxy.add(5, 6);
String echoResult = proxy.echo("result");
```

经过以上四步, 我们便利用 Hadoop RPC 搭建了一个非常高效的客户机-服务器网络模型。接下来, 我们将深入到 Hadoop RPC 内部, 剖析它的设计原理及技巧。

3.3.5 Hadoop RPC 类详解

Hadoop RPC 主要由三个大类组成, 即 RPC、Client 和 Server, 分别对应对外编程接口、客户端实现和服务器实现。

1. ipc.RPC 类分析

RPC 类实际上是对底层客户机-服务器网络模型的封装, 以便为程序员提供一套更方便简洁的编程接口。

如图 3-4 所示, RPC 类定义了一系列构建和销毁 RPC 客户端的方法, 构建方法分为 getProxy 和 waitForProxy 两类, 销毁方只有一个, 即为 stopProxy。RPC 服务器的构建则由静态内部类 RPC.Builder, 该类提供了一些列 setXxx 方法 (Xxx 为某个参数名称) 供用户设置一些基本的参数, 比如 RPC 协议、RPC 协议实现对象、服务器绑定地址、端口号等, 一旦设置完成这些参数后, 可通过调用 RPC.Builder.build() 完成一个服务器对象的构建, 之后直接调用 Server.start() 方法便可以启动该服务器。

与 Hadoop 1.x 中的 RPC 仅支持基于 Writable 序列化方式不同, Hadoop 2.x 允许用户使用其他序列化框架, 比如 Protocol Buffers 等, 目前提供了 Writable (WritableRpcEngine) 和 Protocol Buffers (ProtobufRpcEngine) 两种, 默认实现是 Writable 方式, 用户可通过调用 RPC.setProtocolEngine(...) 修改采用的序列化方式。

下面以采用 Writable 序列化为例 (采用 Protocol Buffers 的过程类似), 介绍 Hadoop RPC 的远程过程调用流程。Hadoop RPC 使用了 Java 动态代理完成对远程方法的调用: 用户只需实现 java.lang.reflect.InvocationHandler 接口, 并按照自己需求实现 invoke 方法即可完成动态代理类对象上的方法调用。但对于 Hadoop RPC, 函数调用由客户端发出, 并在服务器端执行并返回, 因此不能像单机程序那样直接在 invoke 方法中本地调用相关函数, 它的做法是, 在 invoke 方法中, 将函数调用信息 (函数名, 函数参数列表等) 打包成可序列

化的 WritableRpcEngine.Invocation 对象，并通过网络发送给服务器端，服务端收到该调用信息后，解析出和函数名，函数参数列表等信息，利用 Java 反射机制完成函数调用，期间涉及到的类关系如下图所示。

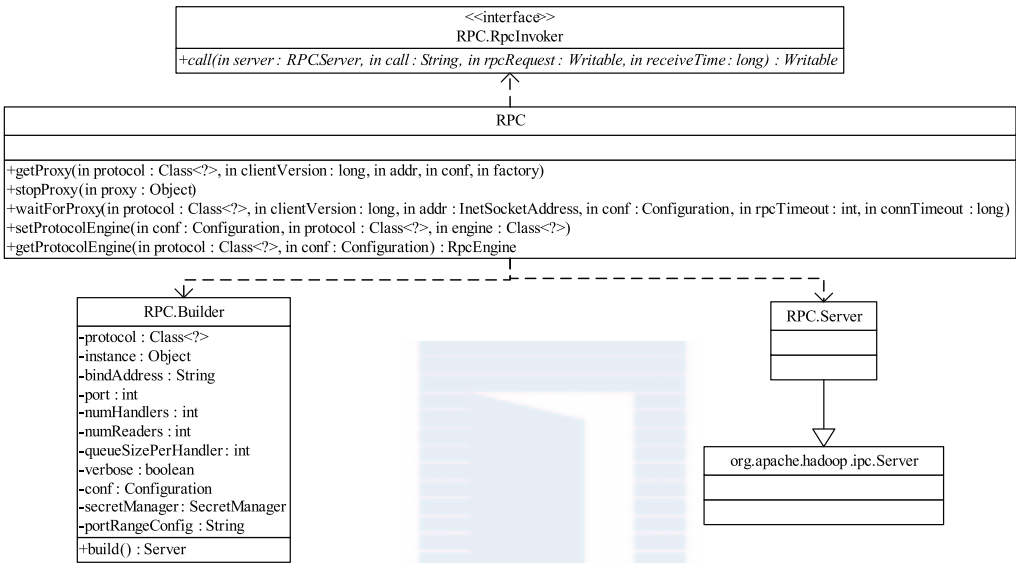


图 3-4 HadoopRPC 的主要类关系图

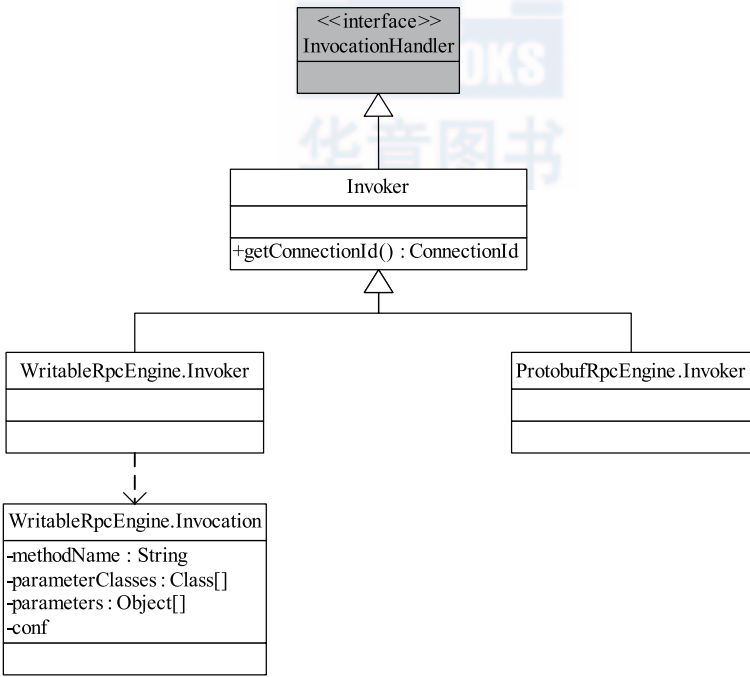


图 3-5 HadoopRPC 中服务器端动态代理实现类图

2. ipc.Client

Client 主要完成的功能是发送远程过程调用信息并接收执行结果。它涉及到的类关系如图 3-6 所示。Client 类对外提供了一类执行远程调用的接口，这些接口的名称一样，仅仅是参数列表不同，比如其中一个的声明如下所示：

```
public Writable call(Writable param, ConnectionIdremoteId)
    throws InterruptedException, IOException;
```

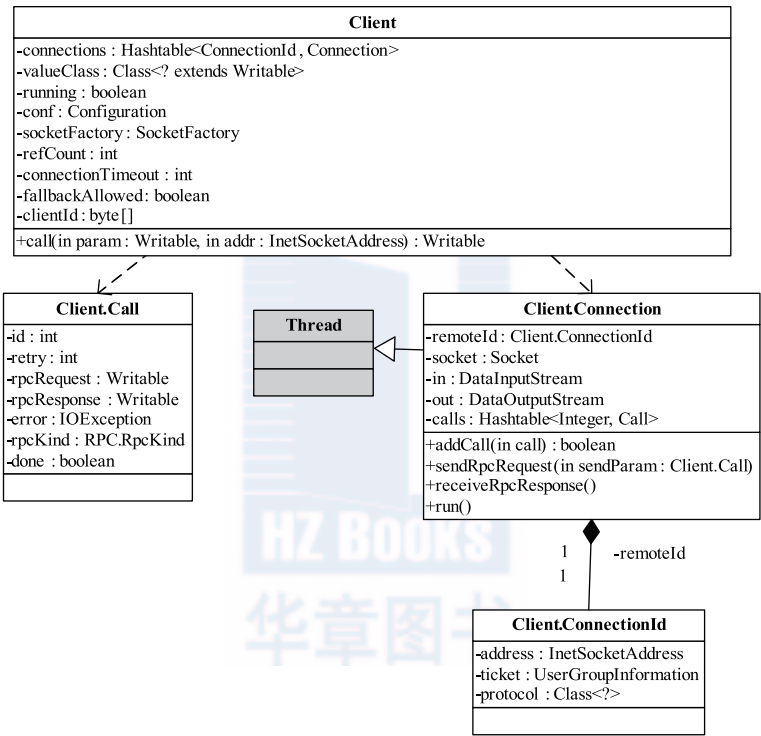


图 3-6 Client 类图

Client 内部有两个重要的内部类，分别是 Call 和 Connection。

□ **Call 类**：封装了一个 RPC 请求，它包含 5 个成员变量，分别是唯一标识 id、函数调用信息 param、函数执行返回值 value、出错或者异常信息 error 和执行完成标识符 done。由于 Hadoop RPC Server 采用异步方式处理客户端请求，这使远程过程调用的发生顺序与结果返回顺序无直接关系，而 Client 端正是通过 id 识别不同的函数调用的。当客户端向服务器端发送请求时，只需填充 id 和 param 两个变量，而剩下的 3 个变量（value、error 和 done）则由服务器端根据函数执行情况填充。

□ **Connection 类**：Client 与每个 Server 之间维护一个通信连接，与该连接相关的基本信息及操作被封装到 Connection 类中，基本信息主要包括通信连接唯一标识

(remoteId)、与 Server 端通信的 Socket (socket)、网络输入数据流 (in)、网络输出数据流 (out)、保存 RPC 请求的哈希表 (calls) 等。操作则包括：

- addCall——将一个 Call 对象添加到哈希表中；
- sendParam——向服务器端发送 RPC 请求；
- receiveResponse ——从服务器端接收已经处理完成的 RPC 请求；
- run——Connection 是一个线程类，它的 run 方法调用了 receiveResponse 方法，会一直等待接收 RPC 返回结果。

当调用 call 函数执行某个远程方法时，Client 端需要进行（如图 3-7 所示）以下 4 个步骤。

- 1) 创建一个 Connection 对象，并将远程方法调用信息封装成 Call 对象，放到 Connection 对象中的哈希表中；
- 2) 调用 Connection 类中的 sendRpcRequest() 方法将当前 Call 对象发送给 Server 端；
- 3) Server 端处理完 RPC 请求后，将结果通过网络返回给 Client 端，Client 端通过 receiveRpcResponse() 函数获取结果；
- 4) Client 检查结果处理状态（成功还是失败），并将对应 Call 对象从哈希表中删除。

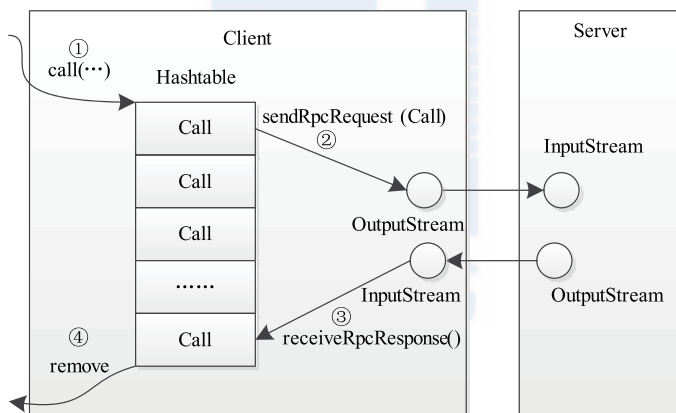


图 3-7 Hadoop RPC Client 处理流程

3. ipc.Server 类分析

Hadoop 采用了 Master/Slave 结构，其中 Master 是整个系统的单点，如 NameNode 或 JobTracker^①，这是制约系统性能和可扩展性的最关键因素之一；而 Master 通过 ipc.Server 接收并处理所有 Slave 发送的请求，这就要求 ipc.Server 将高并发和可扩展性作为设计目标。为此，ipc.Server 采用了很多提高并发处理能力的技术，主要包括线程池、事件驱动和 Reactor 设计模式等，这些技术均采用了 JDK 自带的库实现，这里重点分析它是如何利用 Reactor 设计模式提高整体性能的。

^① HDFS 的单点故障已经在 Hadoop 2.0 中得到了解决，MRv1 中的 JobTracker 的单点故障在 CDH4 中也得到了解决。

Reactor 是并发编程中的一种基于事件驱动的设计模式，它具有以下两个特点：通过派发 / 分离 I/O 操作事件提高系统的并发性能；提供了粗粒度的并发控制，使用单线程实现，避免了复杂的同步处理。典型的 Reactor 实现原理如图 3-8 所示。

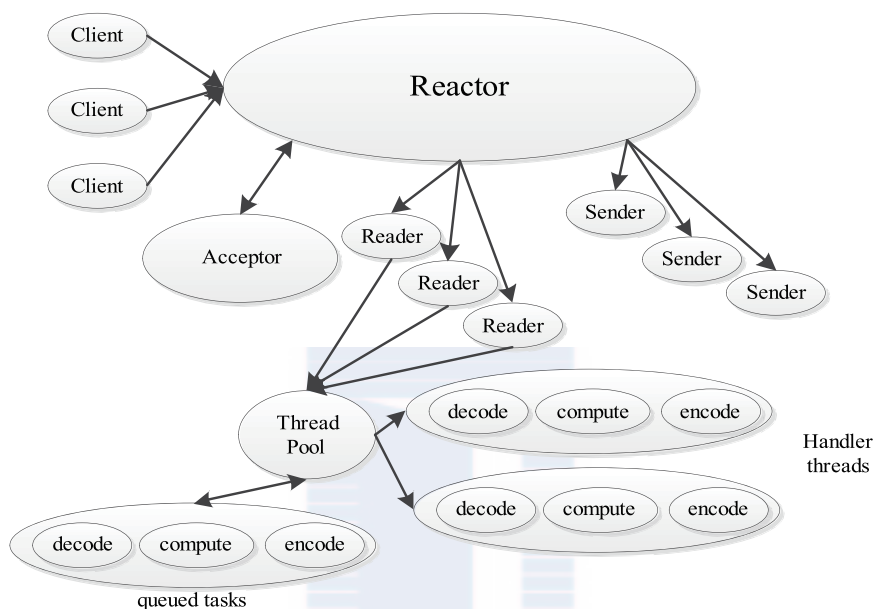


图 3-8 Reactor 模式工作原理

典型的 Reactor 模式中主要包括以下几个角色。

□ **Reactor**：I/O 事件的派发者。

□ **Acceptor**：接受来自 Client 的连接，建立与 Client 对应的 Handler，并向 Reactor 注册此 Handler。

□ **Handler**：与一个 Client 通信的实体，并按一定的过程实现业务的处理。Handler 内部往往会有更进一步的层次划分，用来抽象诸如 read、decode、compute、encode 和 send 等过程。在 Reactor 模式中，业务逻辑被分散的 I/O 事件所打破，所以 Handler 需要有适当的机制在所需的信息还不全（读到一半）的时候保存上下文，并在下一次 I/O 事件到来的时候（另一半可读）能继续上次中断的处理。

□ **Reader/Sender**：为了加速处理速度，Reactor 模式往往构建一个存放数据处理线程的线程池，这样数据读出后，立即扔到线程池中等待后续处理即可。为此，Reactor 模式一般分离 Handler 中的读和写两个过程，分别注册成单独的读事件和写事件，并由对应的 Reader 和 Sender 线程处理。

ipc.Server 实际上实现了一个典型的 Reactor 设计模式，其整体架构与上述完全一致。一旦读者了解典型 Reactor 架构便可很容易地学习 ipc.Server 的设计思路及实现。接下来，我们分析 ipc.Server 的实现细节。

前面提到，ipc.Server 的主要功能是接收来自客户端的 RPC 请求，经过调用相应的函数获取结果后，返回给对应的客户端。为此，ipc.Server 被划分成 3 个阶段：接收请求、处理请求和返回结果，如图 3-9 所示。各阶段实现细节如下。

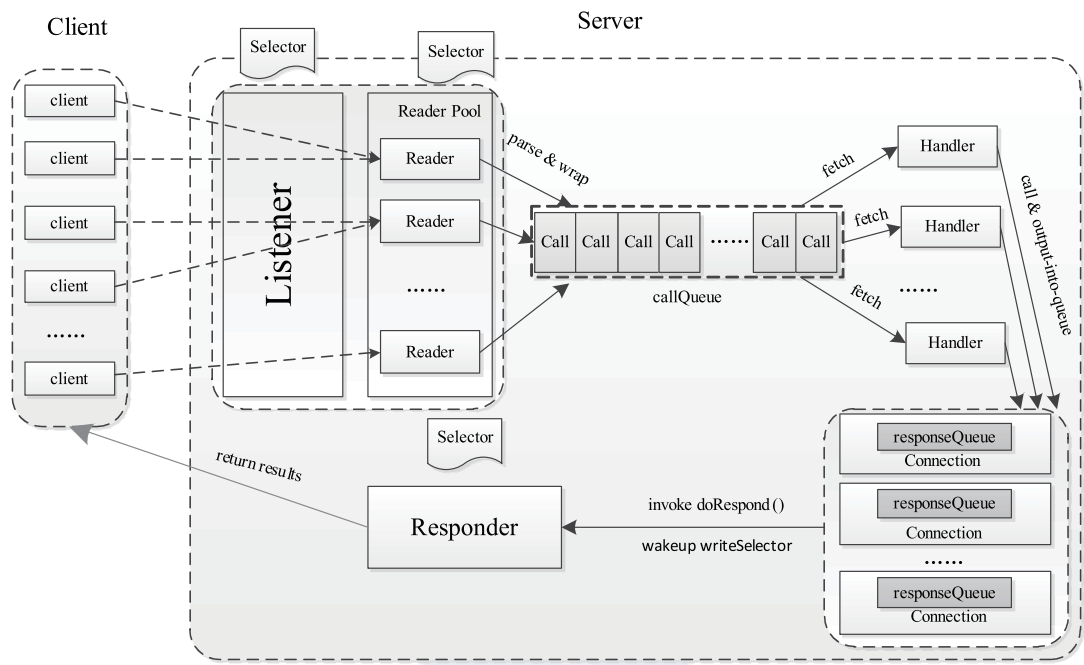


图 3-9 Hadoop RPC Server 处理流程

(1) 接收请求

该阶段主要任务是接收来自各个客户端的 RPC 请求，并将它们封装成固定的格式（Call 类）放到一个共享队列（callQueue）中，以便进行后续处理。该阶段内部又分为建立连接和接收请求两个子阶段，分别由 Listener 和 Reader 两种线程完成。

整个 Server 只有一个 Listener 线程，统一负责监听来自客户端的连接请求，一旦有新的请求到达，它会采用轮询的方式从线程池选择一个 Reader 线程进行处理，而 Reader 线程可同时存在多个，它们分别负责接收一部分客户端连接的 RPC 请求，至于每个 Reader 线程负责哪些客户端连接，完全由 Listener 决定，当前 Listener 只是采用了简单的轮询分配机制。

Listener 和 Reader 线程内部各自包含一个 Selector 对象，分别用于监听 SelectionKey.OP_ACCEPT 和 SelectionKey.OP_READ 事件。对于 Listener 线程，主循环的实现体是监听是否有新的连接请求到达，并采用轮询策略选择一个 Reader 线程处理新连接；对于 Reader 线程，主循环的实现体是监听（它负责的那部分）客户端连接中是否有新的 RPC 请求到达，并将新的 RPC 请求封装成 Call 对象，放到共享队列 callQueue 中。

(2) 处理请求

该阶段主要任务是从共享队列 callQueue 中获取 Call 对象，执行对应的函数调用，并

将结果返回给客户端，这全部由 Handler 线程完成。

Server 端可同时存在多个 Handler 线程，它们并行从共享队列中读取 Call 对象，经执行对应的函数调用后，将尝试着直接将结果返回给对应的客户端。但考虑到某些函数调用返回结果很大或者网络速度过慢，可能难以将结果一次性发送到客户端，此时 Handler 将尝试着将后续发送任务交给 Responder 线程。

（3）返回结果

前面提到，每个 Handler 线程执行完函数调用后，会尝试着将执行结果返回给客户端，但对于特殊情况，比如函数调用返回结果过大或者网络异常情况（网速过慢），会将发送任务交给 Responder 线程。

Server 端仅存在一个 Responder 线程，它的内部包含一个 Selector 对象，用于监听 SelectionKey.OP_WRITE 事件。当 Handler 没能将结果一次性发送到客户端时，会向该 Selector 对象注册 SelectionKey.OP_WRITE 事件，进而由 Responder 线程采用异步方式继续发送未发送完成的结果。

3.3.6 Hadoop RPC 参数调优

Hadoop RPC 对外提供了一些可配置参数，以便于用户根据业务需求和硬件环境对其进行调优。主要的配置参数如下。

- **Reader 线程数目。**由参数 `ipc.server.read.threadpool.size` 配置，默认是 1，也就是说，默认情况下，一个 RPC Server 只包含一个 Reader 线程。
- **每个 Handler 线程对应的最大 Call 数目。**由参数 `ipc.server.handler.queue.size` 指定，默认是 100，也就是说，默认情况下，每个 Handler 线程对应的 Call 队列长度为 100。比如，如果 Handler 数目为 10，则整个 Call 队列（即共享队列 `callQueue`）最大长度为： $100 \times 10 = 1000$ 。
- **Handler 线程数目。**在 Hadoop 中，ResourceManager 和 NameNode 分别是 YARN 和 HDFS 两个子系统上的 RPC Server，其对应的 Handler 数目分别由参数 `yarn.resourcemanager.resource-tracker.client.thread-count` 和 `dfs.namenode.service.handler.count` 指定，默认值分别为 50 和 10，当集群规模较大时，这两个参数值会大大影响系统性能。
- **客户端最大重试次数。**在分布式环境下，因网络故障或者其他原因迫使客户端重试连接是很常见的，但尝试次数过多可能不利于对实时性要求较高的应用。客户端最大重试次数由参数 `ipc.client.connect.max.retries` 指定，默认值为 10，也就是会连续尝试 10 次（每两次之间相隔 1 秒）。

3.3.7 YARN RPC 实现

当前存在非常多的开源 RPC 框架，比较有名的有 Thrift^①、Protocol Buffers 和 Avro。

① 参见网址 <http://thrift.apache.org/>。

同 Hadoop RPC 一样，它们均由两部分组成：对象序列化和远程过程调用（Protocol Buffers 官方仅提供了序列化实现，未提供远程调用相关实现，但三方 RPC 库非常多^①）。相比于 Hadoop RPC，它们有以下几个特点：

- **跨语言特性**。前面提到，RPC 框架实际上是客户机-服务器模型的一个应用实例，对于 Hadoop RPC 而言，由于 Hadoop 采用 Java 语言编写，因而其 RPC 客户端和服务端仅支持 Java 语言；但对于更通用的 RPC 框架，如 Thrift 或者 Protocol Buffers 等，其客户端和服务端可采用任何语言编写，如 Java、C++、Python 等，这给用户编程带来极大方便。
- **引入 IDL**。开源 RPC 框架均提供了一套接口描述语言（Interface Description Language, IDL），它提供一套通用的数据类型，并以这些数据类型来定义更为复杂的数据类型和对外服务接口。一旦用户按照 IDL 定义的语法编写完接口文件后，可根据实际需要生成特定编程语言（如 Java、C++、Python 等）的客户端和服务端代码。
- **协议兼容性**。开源 RPC 框架在设计上均考虑到了协议兼容性问题，即当协议格式发生改变时，比如某个类需要添加或者删除一个成员变量（字段）后，旧版本代码仍然能识别新格式的数据，也就是说，具有向后兼容性。

随着 Hadoop 版本的不断演化，研发人员发现 Hadoop RPC 在跨语言支持和协议兼容性两个方面存在不足，具体表现为：

- 从长远发展看，Hadoop RPC 应允许某些协议的客户端或者服务端采用其他语言实现，比如用户希望直接使用 C/C++ 语言读写 HDFS 中的文件，这就需要有 C/C++ 语言的 HDFS 客户端。
- 当前 Hadoop 版本较多，而不同版本之间不能通信，比如 0.20.2 版本的 JobTracker 不能与 0.21.0 版本中的 TaskTracker 通信，如果用户企图这样做，会抛出 VersionMismatch 异常。

为了解决以上几个问题，Hadoop YARN 将 RPC 中的序列化部分剥离开，以便将现有的开源 RPC 框架集成进来。经过改进之后，Hadoop RPC 的类关系如图 3-10 所示，RPC 类变成了一个工厂，它将具体的 RPC 实现授权给 RpcEngine 实现类，而现有的开源 RPC 只要实现 RpcEngine 接口，便可以集成到 Hadoop RPC 中。在该图中，WritableRpcEngine 是采用 Hadoop 自带的序列化框架实现的 RPC，而 AvroRpcEngine^②和 ProtobufRpcEngine^③分别是开源 RPC（或序列化）框架 Apache Avro 和 Protocol Buffers 对应的 RpcEngine 实现，用户可通过配置参数 `rpc.engine.{protocol}` 以指定协议 {protocol} 采用的序列化方式。需要注意的是，当前实现中，Hadoop RPC 只是采用了这些开源框架的序列化机制，底层的函数调用机制仍采用 Hadoop 自带的。

YARN 提供的对外类是 YarnRPC，用户只需使用该类便可以构建一个基于 Hadoop

① 参见网址：<http://code.google.com/p/protobuf/wiki/ThirdParty/AddOns>。

② AvroRpcEngine 从 Hadoop 0.21.0 版本开始出现。

③ ProtobufRpcEngine 从 Hadoop 2.0-apha 版本开始出现。

RPC 且采用 Protocol Buffers 序列化框架的通信协议。YarnRPC 相关实现类如图 3-11 所示。

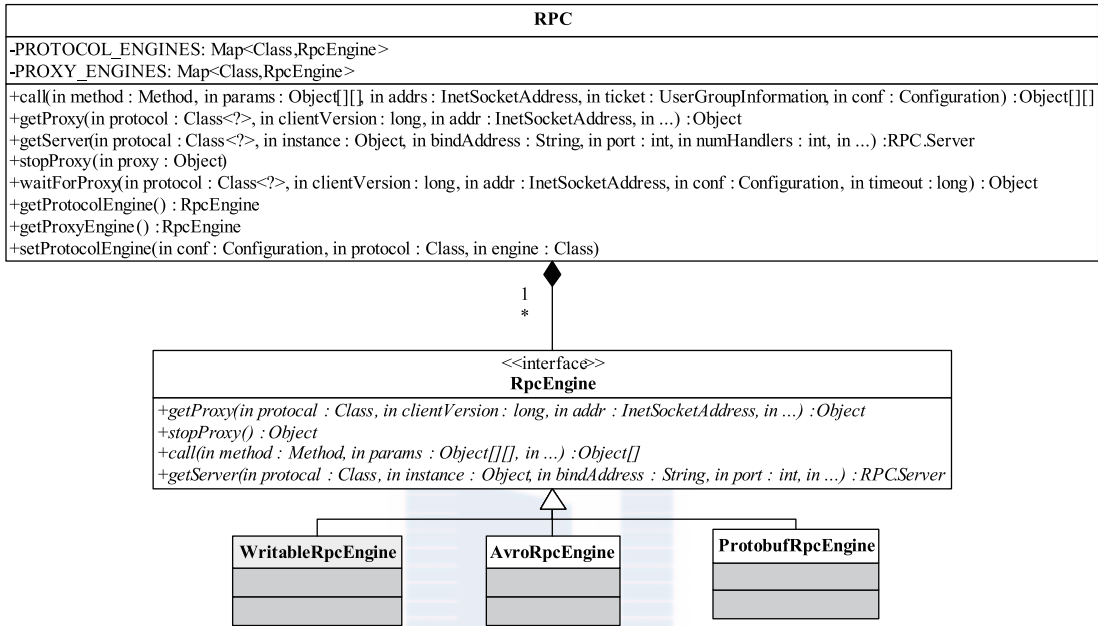


图 3-10 Hadoop RPC 集成多种开源 RPC 框架

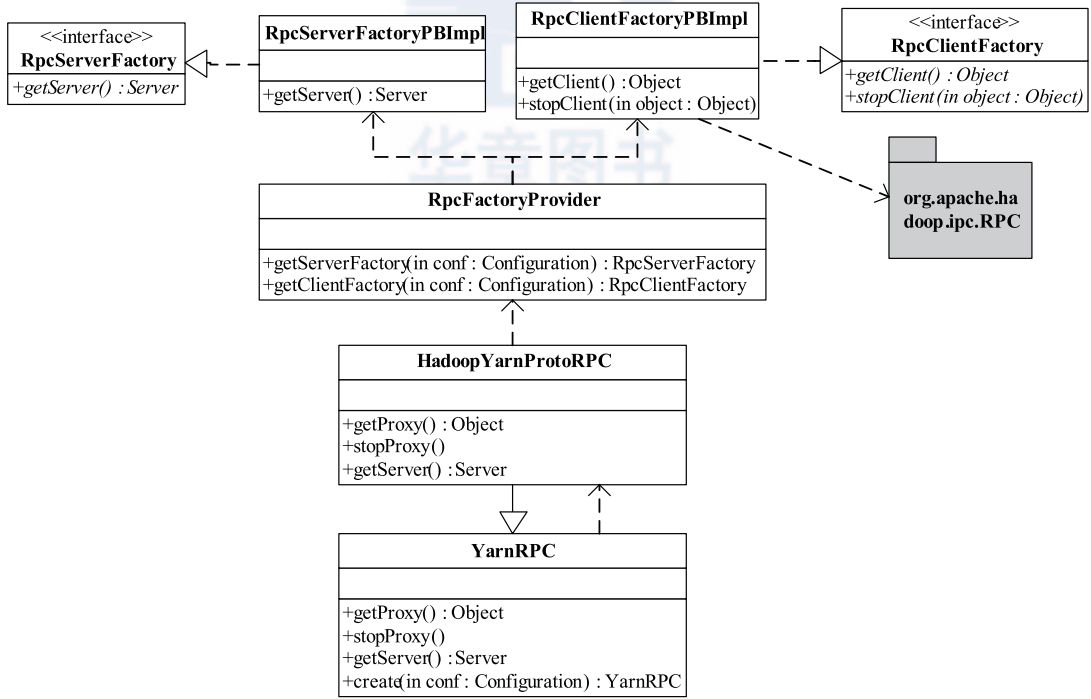


图 3-11 YarnRPC 相关类图

YarnRPC 是一个抽象类，实际的实现由参数 `yarn.ipc.rpc.class` 指定，默认值是 `org.apache.hadoop.yarn.ipc.HadoopYarnProtoRPC`。`HadoopYarnProtoRPC` 通过 RPC 工厂生成器（工厂设计模式）`RpcFactoryProvider` 生成客户端工厂（由参数 `yarn.ipc.client.factory.class` 指定，默认值是 `org.apache.hadoop.yarn.factories.impl.pb.RpcClientFactoryPBImpl`）和服务器工厂（由参数 `yarn.ipc.server.factory.class` 指定，默认值是 `org.apache.hadoop.yarn.factories.impl.pb.RpcServerFactoryPBImpl`），以根据通信协议的 Protocol Buffers 定义生成客户端对象和服务对象。

- **RpcClientFactoryPBImpl**：根据通信协议接口（实际上就是一个 Java interface）及 Protocol Buffers 定义构造 RPC 客户端句柄，但它对通信协议的存放位置和类名命名有一定要求。假设通信协议接口 Xxx 所在 Java 包名为 XxxPackage，则客户端实现代码必须位于 Java 包 XxxPackage.impl.pb.client 中（在接口包名后面增加 “.impl.pb.client”），且实现类名为 PBClientImplXxx（在接口名前面增加前缀 “PBClientImpl”）。
- **RpcServerFactoryPBImpl**：根据通信协议接口（实际上就是一个 Java interface）及 Protocol Buffers 定义构造 RPC 服务器句柄（具体会调用前面节介绍的 RPC.Server 类），但它对通信协议的存放位置和类命名有一定要求。假设通信协议接口 Xxx 所在 Java 包名为 XxxPackage，则客户端实现代码必须位于 Java 包 XxxPackage.impl.pb.server 中（在接口包名后面增加 “.impl.pb.server”），且实现类名为 PBServiceImplXxx（在接口名前面增加前缀 “PBServiceImpl”）。

Hadoop YARN 已将 Protocol Buffers 作为默认的序列化机制^①（而不是 Hadoop 自带的 Writable），这带来的好处主要表现在以下几个方面：

- **继承了 Protocol Buffers 的优势**。Protocol Buffers 已在实践中证明了其高效性、可扩展性、紧凑性和跨语言特性。首先，它允许在保持向后兼容性的前提下修改协议，比如为某个定义好的数据格式添加一个新的字段；其次，它支持多种语言，进而方便用户为某些服务（比如 HDFS 的 NameNode）编写非 Java 客户端^②；此外，实验表明 Protocol Buffers 比 Hadoop 自带的 Writable 在性能方面有很大提升。
- **支持升级回滚**。Hadoop 2.0 已经将 NameNode HA 方案合并进来，在该方案中，NameNode 分为 Active 和 Standby 两种角色，其中，Active NameNode 在当前对外提供服务，而 Standby NameNode 则是能够在 Active NameNode 出现故障时接替它。采用 Protocol Buffers 序列化机制后，管理员能够在不停止 NameNode 对外服务的前提下，通过主备 NameNode 之间的切换，依次对主备 NameNode 进行在线升级（不用考虑版本和协议兼容性问题）。

① 参见网址 <https://issues.apache.org/jira/browse/HADOOP-7347>。

② Hadoop 2.0 中的 RFC 框架是采用 Java 编写的，尚不能像 Thrift 和 Avro 那样支持多语言编程，但引入 Protocol Buffers 序列化框架则使其向前迈进了一步。

3.3.8 YARN RPC 应用实例

为了进一步说明 YARN RPC 的使用方法，本小节给出一个具体的应用实例。

在 YARN 中，ResourceManager 和 NodeManager 之间的通信协议是 ResourceTracker，其中 NodeManager 是该协议的客户端，ResourceManager 是服务端，NodeManager 通过该协议中定义的两个 RPC 函数（registerNodeManager 和 nodeHeartbeat）向 ResourceManager 注册和周期性发送心跳信息。ResourceManager（服务器端）中的相关代码如下：

```
// ResourceTrackerService 实现了 ResourceTracker 通信接口，并启动 RPC Server
public class ResourceTrackerService extends AbstractService implements
    ResourceTracker {
    private Server server;
    ...
    protected void serviceStart() throws Exception {
        super.serviceStart();
        Configuration conf = getConfig();
        YarnRPC rpc = YarnRPC.create(conf); // 使用 YarnRPC 类
        this.server = rpc.getServer(ResourceTracker.class, this, resourceTrackerAddress,
            conf, null, conf.getInt(YarnConfiguration.RM_RESOURCE_TRACKER_CLIENT_THREAD_COUNT,
                YarnConfiguration.DEFAULT_RM_RESOURCE_TRACKER_CLIENT_THREAD_COUNT));
        this.server.start();
    }
    ...
    @Override
    public RegisterNodeManagerResponse registerNodeManager(
        RegisterNodeManagerRequest request) throws YarnException,
        IOException {
        // 具体实现
    }
    @Override
    public NodeHeartbeatResponse nodeHeartbeat(NodeHeartbeatRequest request)
        throws YarnException, IOException {
        // 具体实现
    }
}
```

NodeManager（客户端）中的相关代码如下。

```
// 该函数是从 YARN 源代码中简单修改而来的
protected ResourceTracker getRMClient() throws IOException {
    Configuration conf = getConfig();
    InetSocketAddress rmAddress = getRMAddress(conf, protocol);
    RetryPolicy retryPolicy = createRetryPolicy(conf);
    ResourceTracker proxy = RMProxy.<T>getProxy(conf, ResourceTracker.class, rmAddress);
    LOG.info("Connecting to ResourceManager at " + rmAddress);
    return (ResourceTracker) RetryProxy.create(protocol, proxy, retryPolicy);
}
...
this.resourceTracker = getRMClient();
...
```



```
RegisterNodeManagerResponse regNMResponse = resourceTracker.registerNodeManager(request);
...
response = resourceTracker.nodeHeartbeat(request);
```

为了能够让以上代码正常工作，YARN 按照以下流程实现各种功能。

步骤 1 定义通信协议接口（Java Interface）。定义通信协议接口 ResourceTracker，它包含 registerNodeManager 和 nodeHeartbeat 两个函数，且每个函数包含一个参数和一个返回值，具体如下：

```
public interface ResourceTracker {
    public RegisterNodeManagerResponse registerNodeManager(
        RegisterNodeManagerRequest request) throws YarnException, IOException;
    public NodeHeartbeatResponse nodeHeartbeat(NodeHeartbeatRequest request)
        throws YarnException, IOException;
}
```

步骤 2 为通信协议 ResourceTracker 提供 Protocol Buffers 定义和 Java 实现。前面提到，Protocol Buffers 仅提供了序列化框架，但未提供 RPC 实现，因此 RPC 部分需要由用户自己实现，而 YARN 则让 ResourceTrackerService 类实现了 ResourceTracker 协议，它的 Protocol Buffers 定义（具体见文件 ResourceTracker.proto）如下：

```
option java_package = "org.apache.hadoop.yarn.proto";
option java_outer_classname = "ResourceTracker";
option java_generic_services = true;
option java_generate_equals_and_hash = true;
import "yarn_server_common_service_protos.proto";

service ResourceTrackerService {
    rpc registerNodeManager(RegisterNodeManagerRequestProto) returns (RegisterNodeManagerResponseProto);
    rpc nodeHeartbeat(NodeHeartbeatRequestProto) returns (NodeHeartbeatResponseProto);
}
```

ResourceTracker 的 RPC 函数实现是由 ResourceManager 中的 ResourceTrackerService 完成的。

步骤 3 为 RPC 函数的参数和返回值提供 Protocol Buffers 定义。YARN 需要保证每个 RPC 函数的参数和返回值是采用 Protocol Buffers 定义的，因此 ResourceTracker 协议中 RegisterNodeManagerRequest、RegisterNodeManagerResponse、NodeHeartbeatRequest 和 NodeHeartbeatResponse 四个参数或者返回值需要使用 Protocol Buffers 定义，具体如下（见 yarn_server_common_service_protos.proto 文件）：

```
import "yarn_protos.proto";
import "yarn_server_common_service_protos.proto";

message RegisterNodeManagerRequestProto {
    optional NodeIdProto node_id = 1;
    optional int32 http_port = 3;
    optional ResourceProto resource = 4;
```

```

}

message RegisterNodeManagerResponseProto {
    optional MasterKeyProto container_token_master_key = 1;
    optional MasterKeyProto nm_token_master_key = 2;
    optional NodeActionProto nodeAction = 3;
    optional int64 rm_identifiier = 4;
    optional string diagnostics_message = 5;
}
... // 其他几个参数和返回值的定义

```

步骤4 为RPC函数的参数和返回值提供Java定义和封装。YARN采用了Protocol Buffers作为参数和返回值的序列化框架，且以原生态.proto文件的方式给出了定义，而具体的Java代码生成需在代码编写之后完成。基于以上考虑，为了更容易使用Protocol Buffers生成的（Java语言）参数和返回值定义，YARN RPC为每个RPC函数的参数和返回值提供Java定义和封装，以参数RegisterNodeManagerRequest为例进行说明。

Java接口定义如下（见Java包org.apache.hadoop.yarn.server.api.protocolrecords）：

```

public interface RegisterNodeManagerRequest {
    NodeId getNodeId();
    int getHttpPort();
    Resource getResource();

    void setNodeId(NodeId nodeId);
    void setHttpPort(int port);
    void setResource(Resource resource);
}

```

Java封装如下（见Java包org.apache.hadoop.yarn.server.api.protocolrecords.impl.pb）：

```

public class RegisterNodeManagerRequestPBImpl extends
    ProtoBase<RegisterNodeManagerRequestProto> implements RegisterNodeManagerRequest {
    RegisterNodeManagerRequestProto proto = RegisterNodeManagerRequestProto.
getDefaultInstance();
    RegisterNodeManagerRequestProto.Builder builder = null;
    private NodeId nodeId = null;
    ...
    @Override
    public NodeId getNodeId() {
        RegisterNodeManagerRequestProtoOrBuilder p = viaProto ? proto : builder;
        if (this.nodeId != null) {
            return this.nodeId;
        }
        if (!p.hasNodeId()) {
            return null;
        }
        this.nodeId = convertFromProtoFormat(p.getNodeId());
        return this.nodeId;
    }
}

```

```

@Override
public void setNodeId(NodeId nodeId) {
    maybeInitBuilder();
    if (nodeId == null)
        builder.clearNodeId();
    this.nodeId = nodeId;
}
...
}

```

步骤 5 为通信协议提供客户端和服务端实现。客户端代码放在 `org.apache.hadoop.yarn.server.api.impl.pb.client` 包中，且类名为 `ResourceTrackerPBClientImpl`，实现如下：

```

public class ResourceTrackerPBClientImpl implements ResourceTracker, Closeable {
    private ResourceTrackerPB proxy;
    public ResourceTrackerPBClientImpl(long clientVersion, InetSocketAddress addr,
        Configuration conf) throws IOException {
        RPC.setProtocolEngine(conf, ResourceTrackerPB.class, ProtobufRpcEngine.class);
        proxy = (ResourceTrackerPB)RPC.getProxy(
            ResourceTrackerPB.class, clientVersion, addr, conf);
    }
    @Override
    public RegisterNodeManagerResponse registerNodeManager(
        RegisterNodeManagerRequest request) throws YarnException,
        IOException {
        RegisterNodeManagerRequestProto requestProto = ((RegisterNodeManagerRequestPBImpl) request).getProto();
        try {
            return new RegisterNodeManagerResponsePBImpl(proxy.registerNodeManager(
                null, requestProto));
        } catch (ServiceException e) {
            RPCUtil.unwrapAndThrowException(e);
            return null;
        }
    }
    ...
}

```

服务端代码放在 `org.apache.hadoop.yarn.server.api.impl.pb.server` 包中，且类名为 `ResourceTrackerPBServiceImpl`，实现如下：

```

public class ResourceTrackerPBServiceImpl implements ResourceTrackerPB {
    private ResourceTracker real;
    public ResourceTrackerPBServiceImpl(ResourceTracker impl) {
        this.real = impl;
    }
    @Override
    public RegisterNodeManagerResponseProto registerNodeManager(
        RpcController controller, RegisterNodeManagerRequestProto proto)
        throws ServiceException {
        RegisterNodeManagerRequestPBImpl request = new RegisterNodeManagerRequestPBImpl(proto);
        RegisterNodeManagerResponseProto response = real.registerNodeManager(request, controller);
        return response;
    }
}

```



```
        mpl(proto);
    try {
        RegisterNodeManagerResponse response = real.registerNodeManager(request);
        return ((RegisterNodeManagerResponsePBImp) response).getProto();
    } catch (YarnException e) {
        throw new ServiceException(e);
    } catch (IOException e) {
        throw new ServiceException(e);
    }
}
...
}
```

总结上面几个步骤，为了实现基于 Protocol Buffers 序列化框架的 YARN RPC 通信协议 ResourceTracker，YARN 实现了一系列 Java 接口定义和 Protocol Buffers 封装，具体如图 3-12 所示（以服务器端实现为例）。

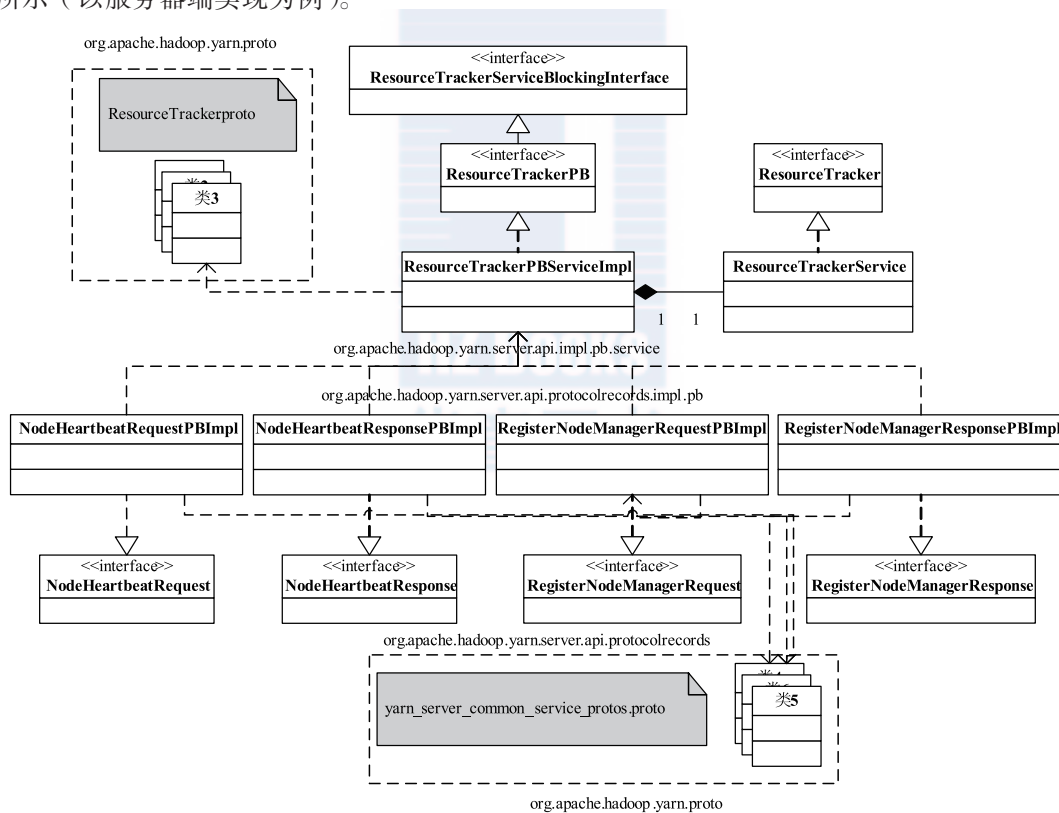


图 3-12 YARN RPC 中的 Protocol Buffers 封装

3.4 服务库与事件库

本节介绍服务库和事件库。

3.4.1 服务库

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。

YARN 中关于服务模型的类图（位于包 org.apache.hadoop.service 中）如图 3-13 所示。在这个图中，我们可以看到，所有的服务对象最终均实现了接口 Service，它定义了最基本的服务初始化、启动、停止等操作，而 AbstractService 类提供了一个最基本的 Service 实现。YARN 中所有对象，如果是非组合服务，直接继承 AbstractService 类即可，否则需继承 CompositeService。比如，对于 ResourceManager 而言，它是一个组合服务，它组合了各种服务对象，包括 ClientRMService、ApplicationMasterLauncher、ApplicationMasterService 等。

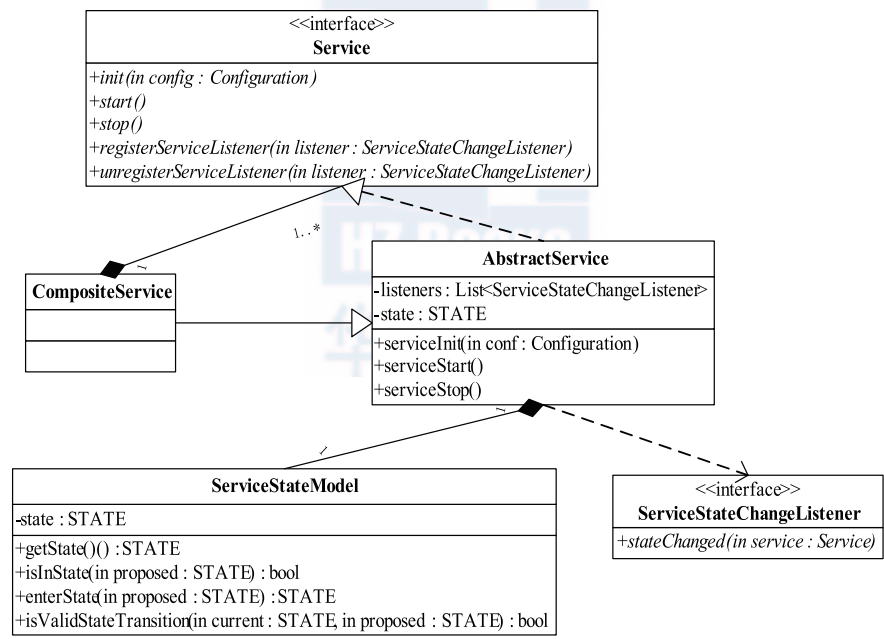


图 3-13 YARN 中服务模型的类图

在 YARN 中，ResourceManager 和 NodeManager 属于组合服务，它们内部包含多个单一服务和组合服务，以实现对内部多种服务的统一管理。

3.4.2 事件库

YARN 采用了基于事件驱动的并发模型，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。YARN 中的事件处理模型可概括为图 3-14 所示。

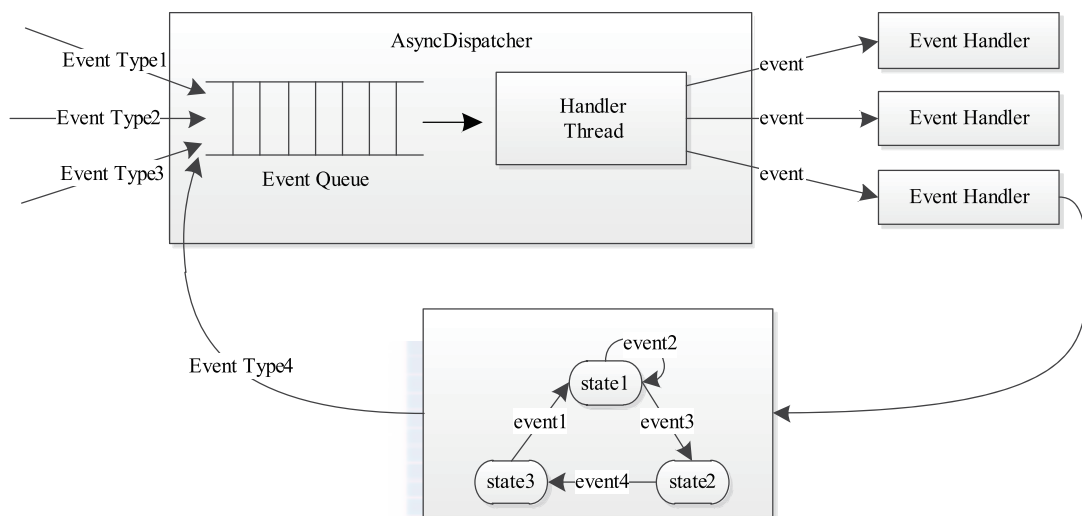


图 3-14 YARN 的事件处理模型

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（Event Handler）。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（MapReduce 应用程序的 ApplicationMaster）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。

YARN 中事件与事件处理器类的关系（位于包 org.apache.hadoop.yarn.event 中）如图 3-15 所示。当使用 YARN 事件库时，通常先要定义一个中央异步调度器 AsyncDispatcher，负责事件的处理与转发，然后根据实际业务需求定义一系列事件 Event 与事件处理器 EventHandler，并注册到中央异步调度器中以实现事件统一管理和调度。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。

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。

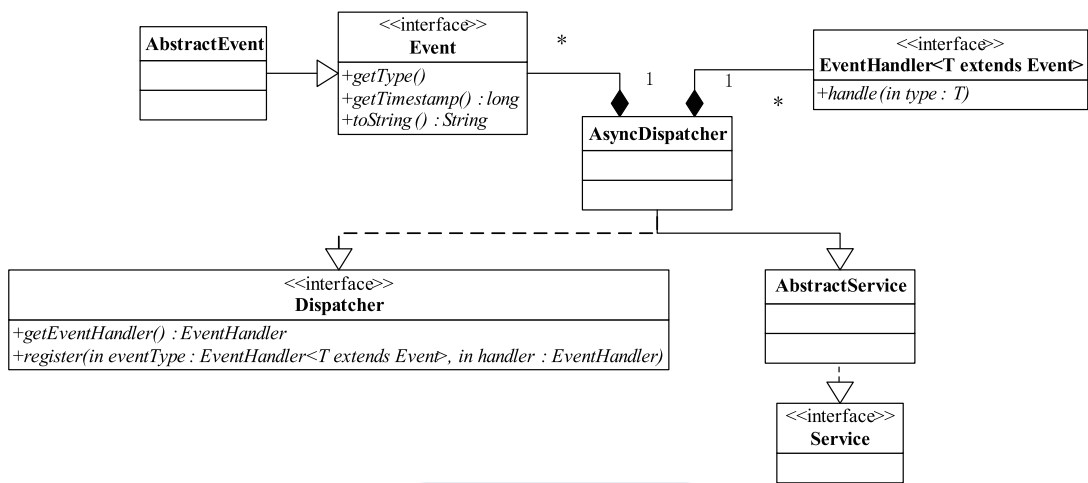


图 3-15 事件与事件处理器

3.4.3 YARN 服务库和事件库的使用方法

为了说明 YARN 服务库和事件库的使用方法，本小节介绍一个简单的实例，该实例可看做 MapReduce ApplicationMaster (MRAppMaster) 的简化版。该例子涉及任务和作业两种对象的事件以及一个中央异步调度器。步骤如下。

1) 定义 Task 事件。

```
public class TaskEvent extends AbstractEvent<TaskEventType> {
    private String taskID; //Task ID
    public TaskEvent(String taskID, TaskEventType type) {
        super(type);
        this.taskID = taskID;
    }

    public String getTaskID() {
        return taskID;
    }
}
```

其中，Task 事件类型定义如下：

```
public enum TaskEventType {
    T_KILL,
    T_SCHEDULE
}
```

2) 定义 Job 事件。

```
public class JobEvent extends AbstractEvent<JobEventType> {
    private String jobID;
    public JobEvent(String jobID, JobEventType type) {
        super(type);
    }
}
```



```

        this.jobID = jobID;
    }

    public String getJobId() {
        return jobID;
    }
}

```

其中，Job 事件类型定义如下：

```

public enum JobEventType {
    JOB_KILL,
    JOB_INIT,
    JOB_START
}

```

3) 事件调度器。

接下来定义一个中央异步调度器，它接收 Job 和 Task 两种类型事件，并交给对应的事件处理器处理，代码如下：

```

@SuppressWarnings("unchecked")
public class SimpleMRAppMaster extends CompositeService {
    private Dispatcher dispatcher; // 中央异步调度器
    private String jobID;
    private int taskNumber;        // 该作业包含的任务数目
    private String[] taskIDs;      // 该作业内部包含的所有任务

    public SimpleMRAppMaster(String name, String jobID, int taskNumber) {
        super(name);
        this.jobID = jobID;
        this.taskNumber = taskNumber;
        taskIDs = new String[taskNumber];
        for(int i = 0; i < taskNumber; i++) {
            taskIDs[i] = new String(jobID + "_task_" + i);
        }
    }

    public void serviceInit(final Configuration conf) throws Exception {
        dispatcher = new AsyncDispatcher(); // 定义一个中央异步调度器
        // 分别注册 Job 和 Task 事件调度器
        dispatcher.register(JobEventType.class, new JobEventDispatcher());
        dispatcher.register(TaskEventType.class, new TaskEventDispatcher());
        addService((Service) dispatcher);
        super.serviceInit(conf);
    }

    public Dispatcher getDispatcher() {
        return dispatcher;
    }

    private class JobEventDispatcher implements EventHandler<JobEvent> {

```

```

@Override
public void handle(JobEvent event) {
    if(event.getType() == JobEventType.JOB_KILL) {
        System.out.println("Receive JOB_KILL event, killing all the tasks");
        for(int i = 0; i < taskNumber; i++) {
            dispatcher.getEventHandler().handle(new TaskEvent(taskIDs[i],
                TaskEventType.T_KILL));
        }
    } else if(event.getType() == JobEventType.JOB_INIT) {
        System.out.println("Receive JOB_INIT event, scheduling tasks");
        for(int i = 0; i < taskNumber; i++) {
            dispatcher.getEventHandler().handle(new TaskEvent(taskIDs[i],
                TaskEventType.T_SCHEDULE));
        }
    }
}
}

private class TaskEventDispatcher implements EventHandler<TaskEvent> {
    @Override
    public void handle(TaskEvent event) {
        if(event.getType() == TaskEventType.T_KILL) {
            System.out.println("Receive T_KILL event of task " + event.getTaskID());
        } else if(event.getType() == TaskEventType.T_SCHEDULE) {
            System.out.println("Receive T_SCHEDULE event of task " + event.getTaskID());
        }
    }
}
}

```

4) 测试程序。

```

@SuppressWarnings("unchecked")
public class SimpleMRAppMasterTest {
    public static void main(String[] args) throws Exception {
        String jobID = "job_20131215_12";
        SimpleMRAppMaster appMaster = new SimpleMRAppMaster("Simple MRAppMaster", jobID, 5);
        YarnConfiguration conf = new YarnConfiguration(new Configuration());
        appMaster.serviceInit(conf);
        appMaster.serviceStart();
        appMaster.getDispatcher().getEventHandler().handle(new JobEvent(jobID,
            JobEventType.JOB_KILL));
        appMaster.getDispatcher().getEventHandler().handle(new JobEvent(jobID,
            JobEventType.JOB_INIT));
    }
}

```

3.4.4 事件驱动带来的变化

在 MRv1 中，对象之间的作用关系是基于函数调用实现的，当一个对象向另外一个对象传递信息时，会直接采用函数调用的方式，且整个过程是串行的。比如，当 TaskTracker 需要执行一个 Task 时，将首先下载 Task 依赖的文件（JAR 包、二进制文件等、字典文件

等)、然后执行 Task。同时在整个过程中会记录一些关键日志,该过程可用图 3-16 描述。在整个过程中,下载依赖文件是阻塞式的,也就是说,前一个任务未完成文件下载之前,后一个新任务将一直处于等待状态,只有在下载完成后,才会启动一个独立进程运行该任务。尽管后来 MRv1 通过启动过独立线程下载文件解决了该问题^①,但这种方式不是在大系统中彻底解决问题之道,必须引入新的编程模型。

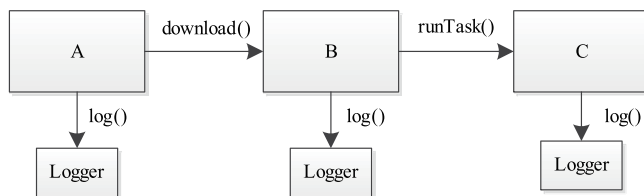


图 3-16 基于函数调用的工作流程

基于函数调用的编程模型是低效的,它隐含着整个过程是串行、同步进行的。相比之下,MRv2 引入的事件驱动编程模型则是一种更加高效的方式。在基于事件驱动的编程模型中,所有对象被抽象成了事件处理器,而事件处理器之间通过事件相互关联。每种事件处理器处理一种类型的事件,同时根据需要触发另外一种事件,该过程如图 3-17 所示,当 A 需要下载文件时,只需向中央异步处理器发送一个事件即可(之后可以继续完成后面的功能而无须等待下载完成),该事件会被传递给对应的事件处理器 B,由 B 完成具体的下载任务。一旦 B 完成下载任务,便可以通过事件通知 A。

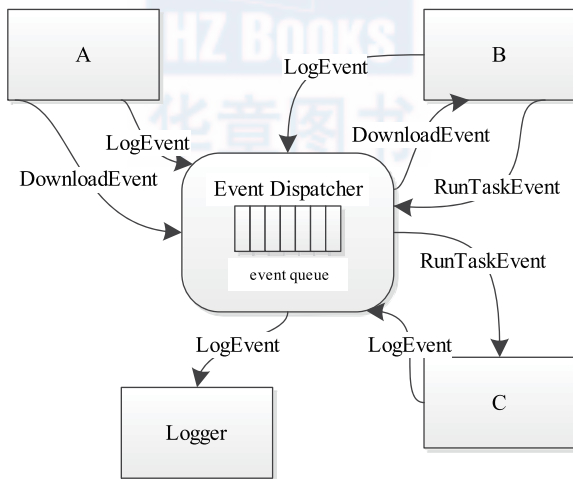


图 3-17 基于事件驱动的工作流程

相比于基于函数调用的编程模型,这种编程方式具有异步、并发等特点,更加高效,因此更适合大型分布式系统。

^① 参见网址 <https://issues.apache.org/jira/browse/MAPREDUCE-2705>。

3.5 状态机库

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。

3.5.1 YARN 状态转换方式

在 YARN 中，每种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（hook）。YARN 定义了三种状态转换方式，具体如下：

1) 一个初始状态、一个最终状态、一种事件（见图 3-18）。该方式表示状态机在 preState 状态下，接收到 Event 事件后，执行函数状态转移函数 Hook，并在执行完成后将当前状态转换为 postState。

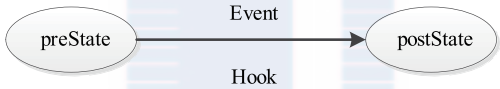


图 3-18 初始状态 : 最终状态 : 事件 = 1 : 1 : 1

2) 一个初始状态、多个最终状态、一种事件（见图 3-19）。该方式表示状态机在 preState 状态下，接收到 Event 事件后，执行函数状态转移函数 Hook，并将当前状态转移为函数 Hook 的返回值所表示的状态。

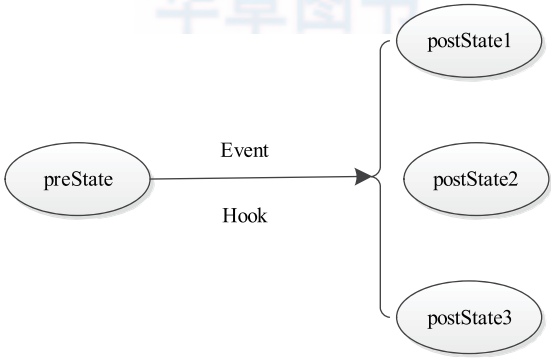


图 3-19 初始状态 : 最终状态 : 事件 = 1 : N : 1

3) 一个初始状态、一个最终状态、多种事件（见图 3-20）。该方式表示状态机在 preState 状态下，接收到 Event1、Event2 和 Event3 中的任何一个事件，将执行函数状态转移函数 Hook，并在执行完成后将当前状态转换为 postState。

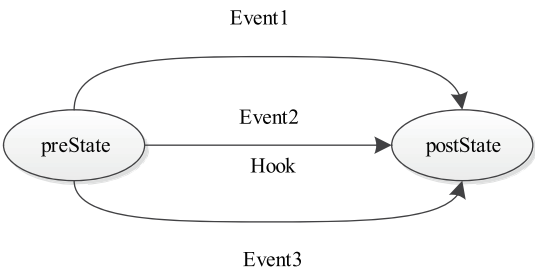


图 3-20 初始状态 : 最终状态 : 事件 = 1 : 1 : N

3.5.2 状态机类

YARN 自己实现了一个非常简单的状态机库（位于包 org.apache.hadoop.yarn.state 中），具体如图 3-21 所示。YARN 对外提供了一个状态机工厂 StateMachineFactory，它提供多种 addTransition 方法供用户添加各种状态转移，一旦状态机添加完毕后，可通过调用 installTopology 完成一个状态机的构建。

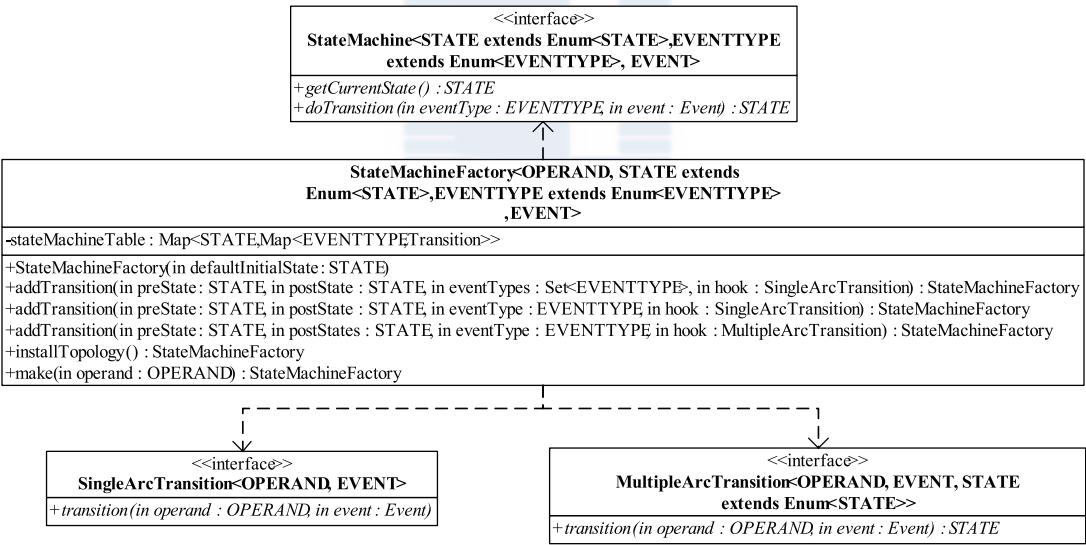


图 3-21 状态机类图

3.5.3 状态机的使用方法

本小节将给出一个状态机应用实例，在该实例中，创建一个作业状态机 JobStateMachine，该状态机维护作业内部的各种状态变化。该状态机同时也是一个事件处理器，当接收到某种事件后，会触发相应的状态转移。该实例中没有给出一个中央异步调度器，可以嵌到 3.4.3 节的实例程序中运行。

1) 定义作业类型。

```
public enum JobEventType {
    JOB_KILL,
    JOB_INIT,
    JOB_START,
    JOB_SETUP_COMPLETED,
    JOB_COMPLETED
}
```

2) 定义作业状态机。

```
@SuppressWarnings({ "rawtypes", "unchecked" })
public class JobStateMachine implements EventHandler<JobEvent>{
    private final String jobID;
    private EventHandler eventHandler;
    private final Lock writeLock;
    private final Lock readLock;

    // 定义状态机
    protected static final
        StateMachineFactory<JobStateMachine, JobStateInternal, JobEventType, JobEvent>
        stateMachineFactory
        = new StateMachineFactory<JobStateMachine, JobStateInternal, JobEventType, JobEvent>
            (JobStateInternal.NEW)

        .addTransition(JobStateInternal.NEW, JobStateInternal.INITED,
            JobEventType.JOB_INIT,
            new InitTransition())
        .addTransition(JobStateInternal.INITED, JobStateInternal.SETUP,
            JobEventType.JOB_START,
            new StartTransition())
        .addTransition(JobStateInternal.SETUP, JobStateInternal.RUNNING,
            JobEventType.JOB_SETUP_COMPLETED,
            new SetupCompletedTransition())
        .addTransition
            (JobStateInternal.RUNNING,
            EnumSet.of(JobStateInternal.KILLED, JobStateInternal.SUCCEEDED),
            JobEventType.JOB_COMPLETED,
            new JobTasksCompletedTransition())
        .installTopology();

    private final StateMachine<JobStateInternal, JobEventType, JobEvent> stateMachine;

    public JobStateMachine(String jobID, EventHandler eventHandler) {
        this.jobID = jobID;
        ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
        this.readLock = readWriteLock.readLock();
        this.writeLock = readWriteLock.writeLock();
        this.eventHandler = eventHandler;
        stateMachine = stateMachineFactory.make(this);
    }
}
```

```

protected StateMachine<JobStateInternal, JobEventType, JobEvent> getStateMachine() {
    return stateMachine;
}

public static class InitTransition
    implements SingleArcTransition<JobStateMachine, JobEvent> {
    @Override
    public void transition(JobStateMachine job, JobEvent event) {
        System.out.println("Receiving event " + event);
        job.eventHandler.handle(new JobEvent(job.getJobId(), JobEventType.JOB_START));
    }
}

public static class StartTransition
    implements SingleArcTransition<JobStateMachine, JobEvent> {
    @Override
    public void transition(JobStateMachine job, JobEvent event) {
        System.out.println("Receiving event " + event);
        job.eventHandler.handle(new JobEvent(job.getJobId(), JobEventType.JOB_SETUP_COMPLETED));
    }
}
...
// 定义类 SetupCompletedTransition 和 JobTasksCompletedTransition

@Override
public void handle(JobEvent event) {
    try {
        writeLock.lock();
        JobStateInternal oldState = getInternalState();
        try {
            getStateMachine().doTransition(event.getType(), event);
        } catch (InvalidStateTransitonException e) {
            System.out.println("Can't handle this event at current state");
        }
        if (oldState != getInternalState()) {
            System.out.println("Job Transitioned from " + oldState + " to "
                + getInternalState());
        }
    }
    finally {
        writeLock.unlock();
    }
}

public JobStateInternal getInternalState() {
    readLock.lock();
    try {
        return getStateMachine().getCurrentState();
    } finally {
        readLock.unlock();
    }
}

```



```

    }
    public enum JobStateInternal { // 作业内部状态
        NEW,
        SETUP,
        INITED,
        RUNNING,
        SUCCEEDED,
        KILLED,
    }
}

```

3.5.4 状态机可视化

YARN 中实现了多个状态机对象，包括 ResourceManager 中的 RMAppImpl、RMAppAttemptImpl、RMContainerImpl 和 RMNodeImpl，NodeManager 中的 ApplicationImpl、ContainerImpl 和 LocalizedResource，MRAppMaster 中的 JobImpl、TaskImpl 和 TaskAttemptImpl 等。为了便于用户查看这些状态机的状态变化以及相关事件，YARN 提供了一个状态机可视化工具[⊖]，具体操作步骤如下。

步骤 1 将状态机转化为 graphviz(.gv) 格式的文件，编译命令如下：

```
mvn compile -Pvisualize
```

经过该步骤后，本地目录中生成了 ResourceManager.gv、NodeManager.gv 和 MapReduce.gv 三个 graphviz 格式的文件（有兴趣可以直接打开查看具体内容）。

步骤 2 使用可视化包 graphviz 中的相关命令生成状态机图，Shell 命令具体如下：

```
dot -Tpng NodeManager.gv > NodeManager.png
```

如果尚未安装 graphviz 包，操作该步骤之前先要安装该包。

3.6 源代码阅读引导

为了帮助读者更好地阅读源代码，笔者特意安排了本节。下面我们分两部分对读者进行指导。

1. Hadoop RPC

Hadoop RPC 内部实现位于源代码目录中 hadoop-common-project/hadoop-common/src/main/java 下的 org.apache.hadoop.ipc 包中，而 YARN 对 RPC 的 Protocol Buffers 封装则位于 hadoop-yarn-project/hadoop-yarn/hadoop-yarn-common/src/main/java 目录下的 org.apache.hadoop.yarn.ipc 包中。建议读者按照以下流程学习这部分源代码：首先尝试使用 Hadoop RPC 编写一个 C/S 服务器，然后在理解 Hadoop RPC 整体架构的基础上，依次阅读客户端部分代码和服务端代码，如果能够描述清楚下面两个流程，则可认为对 Hadoop RPC 有

[⊖] 参见网址 <https://issues.apache.org/jira/browse/MAPREDUCE-2930>。

了较为深入的理解。

- 客户端发送一个请求到接收到请求应答的整个过程是怎样的，依次经过了哪些函数调用和通信过程；
- 多个客户端并发发送请求到服务器端后，服务器端是如何处理的。

2. 服务库、事件库和状态机

了解服务库、事件库和状态机库源代码所在目录可使读者省去许多麻烦。

- 服务库位于源代码目录 `hadoop-common-project/hadoop-common/src/main/java` 下的 `org.apache.hadoop.yarn.service` 包中。
- 事件库和状态机位于源代码目录中 `hadoop-yarn-project/hadoop-yarn/hadoop-yarn-common/src/main/java` 下的 `org.apache.hadoop.yarn.event` 和 `org.apache.hadoop.yarn.state` 两个包中。

建议读者按照以下流程学习这部分源代码：首先弄清楚各个包的对外接口，然后尝试编写几个实例使用这几个库。当然，读者也可以跟着 YARN 源代码学习这几个库，YARN 源代码是这几个库的最好应用实例。

3.7 小结

YARN 基础库是其他一切模块的基础，它的设计直接决定了 YARN 的稳定性和扩展性，YARN 借用了 MRv1 的一些底层基础库，比如 RPC 库等，但因为引入了很多新的软件设计方式，所以它的基础库更多，包括直接使用了开源序列化框架 Protocol Buffers 和 Apache Avro，自定义的服务库、事件库和状态机等。本章介绍了 YARN 最重要的几个基础库。

3.8 问题讨论

问题 1：引入 Protocol Buffers 之后，Hadoop RPC 是否能像 Thrift RPC 或者 Avro RPC 那样具有跨语言特性，比如 Server 采用 Java 编写，Client 采用 C++ 编写？

问题 2：如果 Hadoop RPC 具备跨语言特性，那么相比于 Thrift 或者 Apache Avro，Hadoop RPC 有什么优缺点？如果不具备，该如何让它具备跨语言特性呢？

问题 3：Hadoop RPC 具备跨语言特性后，将带来哪些好处？

第4章 YARN 应用程序设计方法

应用程序 (Application) 是用户编写的处理数据的程序的统称, 它从 YARN 中申请资源以完成自己的计算任务。YARN 自身对应用程序类型没有任何限制, 它可以是处理短类型任务的 MapReduce 作业, 也可以是部署长时间运行的服务的应用程序 (比如将在第 10 章介绍的 Storm On YARN)。应用程序可以向 YARN 申请资源完成各类计算任务。比如 MapReduce 应用程序向 YARN 申请资源, 用于运行 Map Task 和 Reduce Task 两类任务。本章将从高级层面介绍如何设计一个可以运行在 YARN 之上的应用程序。

由于 YARN 应用程序编写比较复杂, 且需对 YARN 本身的架构有一定了解, 因此通常由专业人员开发, 通过回调的形式供其他普通用户使用。比如, 专业人员实现可以直接运行在 YARN 之上的 MapReduce 框架库 (假设打包后为 `yarn-mapreduce.jar`, 主要完成数据切分、资源申请、任务调度与容错、网络通信等功能), 而普通用户只需编写 `map()` 和 `reduce()` 两个函数完成 MapReduce 程序设计 (假设打包后为 `my-app.jar`, 主要完成自己计算所需的逻辑)。这样用户提交应用程序时, YARN 会自动将 `yarn-mapreduce.jar` 和 `my-app.jar` 两个 JAR 包同时提交到 YARN 之上, 以完成一个分布式应用的计算。本章重点介绍的就是专业人员应如何编写一个可直接运行在 YARN 之上的框架 (即前面例子中的 `yarn-mapreduce.jar`)。

4.1 概述

YARN 是一个资源管理系统, 负责集群资源的管理和调度。如果想要将一个新的应用程序运行在 YARN 之上, 通常需要编写两个组件 Client (客户端) 和 ApplicationMaster。这两个组件编写非常复杂, 尤其 ApplicationMaster, 需要考虑 RPC 调用、任务容错等细节。如果大量应用程序可抽象成一种通用框架, 只需实现一个客户端和一个 ApplicationMaster, 然后让所有应用程序重用这两个组件即可。比如 MapReduce 是一种通用的计算框架, YARN 已经为其实现了一个直接可以使用的客户端 (JobClient) 和 ApplicationMaster (MRAppMaster)。

第 2 章中提到, 运行在 YARN 上的应用程序主要分为短应用程序和长应用程序两类, 其中, 短应用程序是短时间内可运行完成的程序, 比如 MapReduce 作业、Tez 作业等; 长应用程序是永不终止运行的服务, 比如 Storm Service、HBase Service 等。尽管这两类应用程序作用不同, 但它们在 YARN 上的工作流程和编写方式是相同的。本节主要介绍如何让一种新的应用程序或者新的框架运行于 YARN 之上。正如前面介绍的, 用户需要编写客户

端和 ApplicationMaster 两个组件完成该功能，其中，客户端负责向 ResourceManager 提交 ApplicationMaster，并查询应用程序运行状态；ApplicationMaster 负责向 ResourceManager 申请资源（以 Container 形式表示），并与 NodeManager 通信以启动各个 Container，此外，ApplicationMaster 还负责监控各个任务运行状态，并在失败时为其重新申请资源。

通常而言，编写一个 YARN Application 会涉及 3 个 RPC 协议，如图 4-1 所示，分别为：

- ❑ ApplicationClientProtocol（用于 Client 与 ResourceManager 之间）。Client 通过该协议可实现将应用程序提交到 ResourceManager 上、查询应用程序的运行状态或者杀死应用程序等功能。
- ❑ ApplicationMasterProtocol（用于 ApplicationMaster 与 ResourceManager 之间）。ApplicationMaster 使用该协议向 ResourceManager 注册、申请资源、获取各个任务运行情况等。
- ❑ ContainerManagementProtocol（用于 ApplicationMaster 与 NodeManager 之间）。ApplicationMaster 使用该协议要求 NodeManager 启动 / 撤销 Container 或者查询 Container 的运行状态。

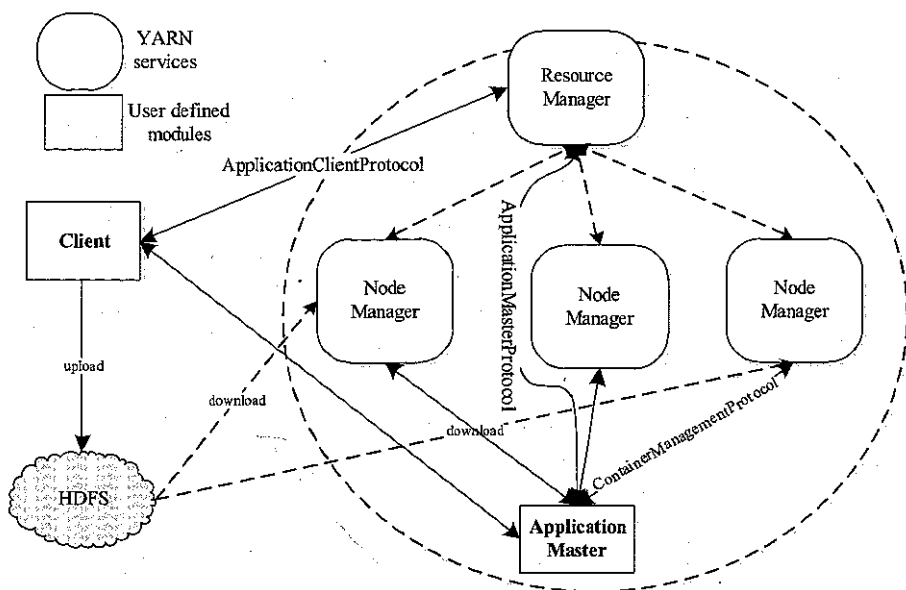


图 4-1 应用程序设计相关的通信协议

我们在前面的章节已经介绍了这 3 个协议，在此不再赘述。

4.2 客户端设计

YARN Application 客户端的主要作用是提供一系列访问接口供用户与 YARN 交互，包括提交 Application、查询 Application 运行状态，修改 Application 属性（如优先级）等。其

中，最重要的访问接口之一是提交 Application 的函数。

4.2.1 客户端编写流程

通常而言，客户端提交一个应用程序需经过以下两个步骤。

步骤 1 Client 通过 RPC 函数 `ApplicationClientProtocol#getNewApplication` 从 `ResourceManager` 中获取唯一的 application ID。

刚开始，客户端应创建一个 `ApplicationClientProtocol` 协议的 RPC Client，并通过该 Client 与 `ResourceManager` 通信：

```
private ApplicationClientProtocol rmClient; //RPC Client;
// rmAddress 为服务器端地址, conf 为配置对象
this.rmClient = (ApplicationClientProtocol) rpc.getProxy(
    ApplicationClientProtocol.class, rmAddress, conf)
```

然后调用 `ApplicationClientProtocol #getNewApplication` 从 `ResourceManager` 上领取唯一的 Application ID，代码如下：

```
GetNewApplicationRequest request =
    Records.newRecord(GetNewApplicationRequest.class);
GetNewApplicationResponse newApp = rmClient.getNewApplication(request);
ApplicationId appId = newApp.getApplicationId();
```

上面的例子中，静态方法是 `Records#newRecord`，常用于构造一个可序列化对象，具体采用的序列化工厂由参数 `yarn.ipc.record.factory.class` 指定，默认是 `org.apache.hadoop.yarn.factories.impl.pb.RecordFactoryPBImpl`，即构造的是 Protocol Buffers 序列化对象。

该函数的返回一个 `GetNewApplicationRequest` 类型的对象，它主要包含两项信息：Application ID 和最大可申请资源量。

步骤 2 Client 通过 RPC 函数 `ApplicationClientProtocol#submitApplication` 将 `ApplicationMaster` 提交到 `ResourceManager` 上。

如图 4-2 所示，客户端将启动 `ApplicationMaster` 所需的所有信息打包到数据结构 `ApplicationSubmissionContext` 中，该数据结构的定义在 Protocol Buffers 文件 `yarn_protos.proto` 中，主要包括以下几个字段（字段名称使用了 Protocol Buffers 文件中定义的名称）。

- `application_id`：Application ID（可通过函数 `ApplicationSubmissionContext#setXXX` 设置，与以下几个字段类似）。
- `application_name`：Application 名称。
- `priority`：Application 优先级。
- `queue`：Application 所属队列。
- `user`：Applications 所属用户名。
- `unmanaged_am`：是否由客户端自己启动 `ApplicationMaster`。
- `cancel_tokens_when_complete`：当应用程序运行完成时，是否取消 Token。通常将该值设为 `true`，除非特殊的应用需求，需要将该应用程序的 Token 共享给其他应用程序。

- `am_container_spec`: 启动 ApplicationMaster 相关的信息, 主要包括下面几项。
- `user`: ApplicationMaster 启动用户 (可通过函数 `ContainerLaunchContext#setXXX` 设置, 与以下几个字段类似)。
 - `resource`: 启动 ApplicationMaster 所需的资源, 当前支持 CPU 和内存两种。
 - `localResources`: ApplicationMaster 运行所需的本地资源, 通常是一些外部文件, 比如字典等。
 - `command`: ApplicationMaster 启动命令 (一般为 Shell 命令)。
 - `environment`: ApplicationMaster 运行时所需的环境变量。

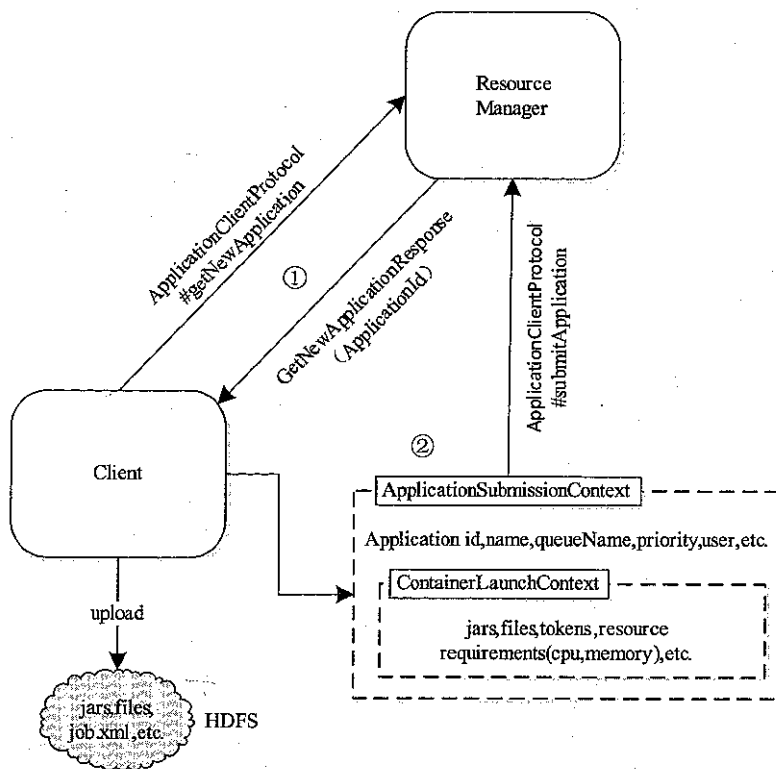


图 4-2 客户端提交应用程序

一段 (简化的) 提交应用程序的实例代码如下:

```

ApplicationSubmissionContext context = Records.newRecord
    (ApplicationSubmissionContext.class);
appContext.setApplicationName(appName); // 设置应用程序名称
... // 设置应用程序其他属性, 比如优先级、队列名称等
ContainerLaunchContext amContainer =
    Records.newRecord(ContainerLaunchContext.class); // 构造一个 AM 启动上下文对象
...// 设置 AM 相关的变量
amContainer.setLocalResources(localResources); // 设置 AM 启动所需的本地资源

```

```

amContainer.setEnvironment(env); // 设置 AM 启动所需的环境变量
appContext.setAMContainerSpec(amContainer);
appContext.setApplicationId(appId); // appId 是上一步获取的 ApplicationId
SubmitApplicationRequest request =
    Records.newRecord(SubmitApplicationRequest.class);
request.setApplicationSubmissionContext(appContext);
rmClient.submitApplication(request); // 将应用程序提交到 ResourceManager 上

```

除了提交 Application 接口外，客户端还需提供以下几种接口的实现，如图 4-3 所示，这些接口的定义在 RPC 协议 ApplicationClientProtocol 中，具体如下：

```

public interface ApplicationClientProtocol {
    // 获取 Application 运行报告，包括用户、队列、运行状态等信息
    public GetApplicationReportResponse getApplicationReport(
        GetApplicationReportRequest request)
        throws YarnRemoteException;
    // 杀死 Application
    public KillApplicationResponse forceKillApplication(
        KillApplicationRequest request)
        throws YarnRemoteException;
    // 获取集群的 metric 信息
    public GetClusterMetricsResponse getClusterMetrics(
        GetClusterMetricsRequest request)
        throws YarnRemoteException;
    // 查看当前系统中所有应用程序信息
    public GetAllApplicationsResponse getAllApplications(
        GetAllApplicationsRequest request)
        throws YarnRemoteException;
    // 查询当前系统中所有节点信息
    public GetClusterNodesResponse getClusterNodes(
        GetClusterNodesRequest request)
        throws YarnRemoteException;
    ...// 其他接口
}

```

以上介绍的 RPC 函数主要用于客户端与 ResourceManager 之间的通信，这一部分对所有类型的应用程序来说都是一致的，故可以做成通用的代码模块。但在实际应用环境中，为了减轻 ResourceManager 的负载，一旦应用程序的 ApplicationMaster 成功启动后，客户端通常直接与 ApplicationMaster 通信，以查询它的运行状态或者控制它的执行流程（比如杀死一个任务等）。这一部分与 ApplicationMaster 的设计相关，因此，不同类型的应用程序是不一样的。也正因如此，用户需要针对不同类型的应用程序开发不同的客户端，如图 4-4 所示。以 MapReduce 的客户端为例，当用户提交一个 MapReduce 应用程序时，需通过 RPC 协议 ApplicationClientProtocol 与 ResourceManager 通信，而一旦 MapReduce 的 ApplicationMaster——MRAppMaster 成功启动后，客户端通过另外一个 RPC 协议——MRClientProtocol 直接与 MRAppMaster 通信，以查询应用程序运行状况和控制应用程序的执行（比如杀死一个 Map Task 等）。

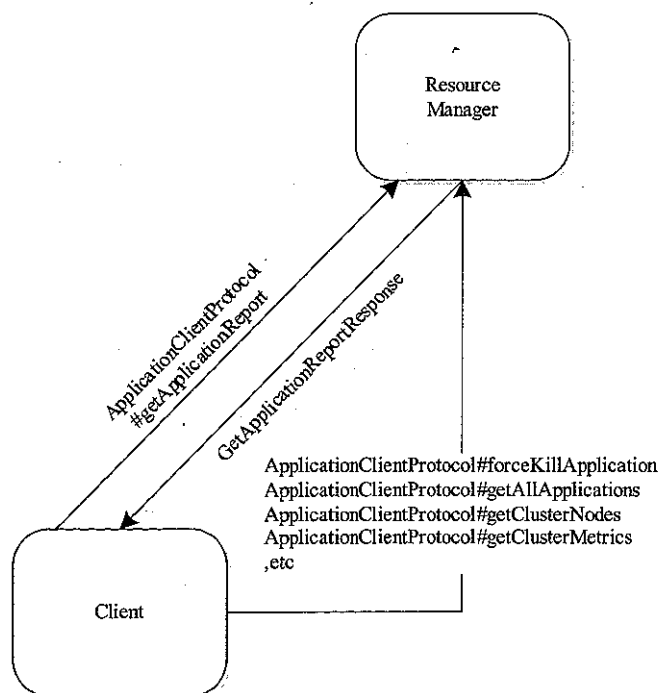


图 4-3 客户端从 RM 上获取应用程序信息

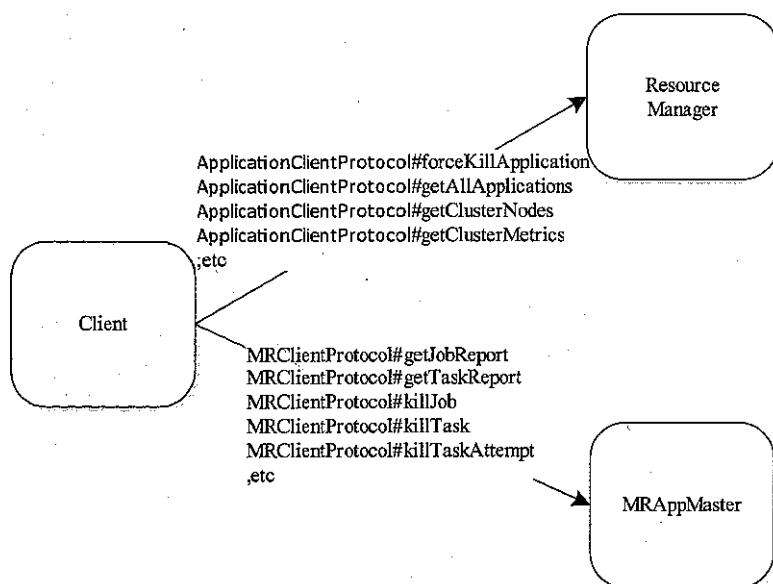


图 4-4 客户端控制应用程序

4.2.2 客户端编程库

前面提到, 不同类型应用程序与 ResourceManager 交互逻辑是类似的, 为了避免简化客户端重复开发, YARN 提供了能与 ResourceManager 交互完成各种操作的编程库 org.apache.hadoop.yarn.client.YarnClient。该库对常用函数进行了封装, 并提供了重试、容错等机制, 用户使用该库可以快速开发一个包含应用程序提交、状态查询和控制等逻辑的 YARN 客户端, 目前 YARN 本身自带的各类客户端均使用该编程库实现。该编程库用法如下 (经过简化):

```
import org.apache.hadoop.yarn.client.api.YarnClient;
import org.apache.hadoop.yarn.client.api.YarnClientApplication;
... // 其他 Java 包
private YarnClient client;
// 构造一个 YarnClient 客户端句柄并初始化
this.client = YarnClient.createYarnClient();
client.init(conf);
// 启动 YarnClient
yarnClient.start();
// 获取一个新的 Application ID
YarnClientApplication app = yarnClient.createApplication();
// 构造 ApplicationSubmissionContext, 用于提交作业
ApplicationSubmissionContext appContext =
app.getApplicationSubmissionContext();
ApplicationId appId = appContext.getApplicationId();
appContext.setApplicationName(appName);
...
yarnClient.submitApplication(appContext); // 将应用程序提交到 ResourceManager 上
```

一个功能完备的 YARN 客户端, 不仅需要与 ResourceManager 交互, 还需要与 ApplicationMaster 交互以查询应用程序内部的信息 (通常 ResourceManager 中没有与某个具体应用程序相关的信息) 或者控制应用程序内部的任务 (比如杀死任务, 同样, ResourceManager 中也不会有具体任务相关的信息), 这一部分需要由应用程序自己设计通信协议。

4.3 ApplicationMaster 设计

ApplicationMaster (AM) 需要与 ResourceManager (RM) 和 NodeManager (NM) 两个服务交互, 通过与 ResourceManager 交互, ApplicationMaster 可获得任务计算所需的资源; 通过与 NodeManager 交互, ApplicationMaster 可启动计算任务 (container), 并监控它直到运行完成。本节将详细介绍 ApplicationMaster 与这两个服务交互逻辑的实现方法。

4.3.1 ApplicationMaster 编写流程

AM 编写流程我们分 AM-RM 和 AM-NM 两部分介绍。

1. AM-RM 编写流程

ApplicationMaster 与 ResourceManager 之间通信涉及三个步骤（分别对应三个 API），如图 4-5 所示，具体如下。

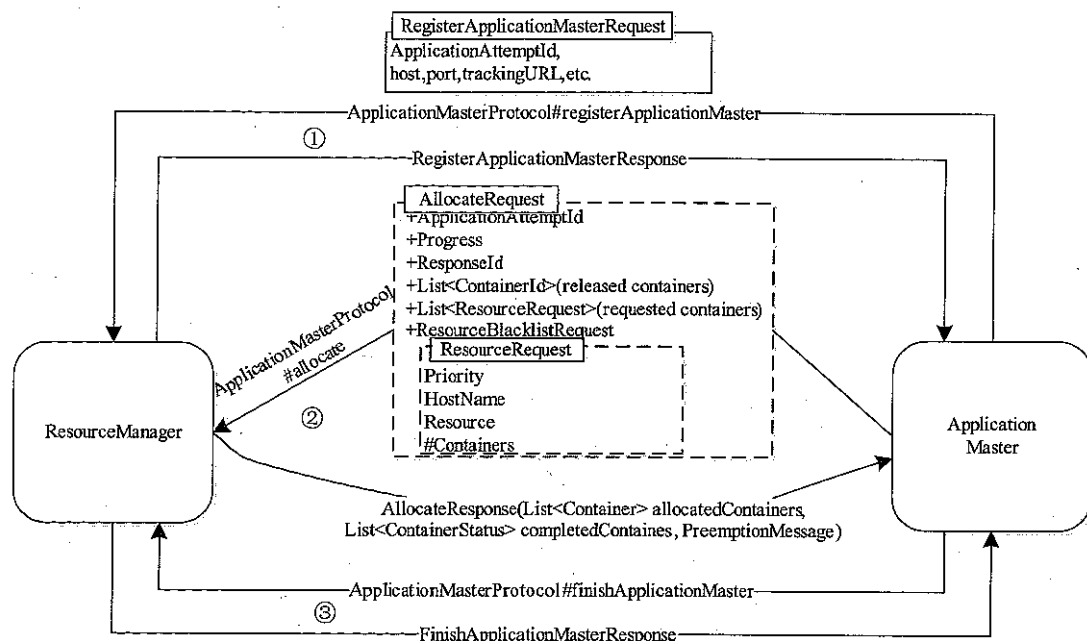


图 4-5 ApplicationMaster 与 ResourceManager 通信流程

步骤 1 ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#registerApplicationMaster 向 ResourceManager 注册。

ApplicationMaster 启动时，首先向 ResourceManager 注册，注册信息封装到 Protocol Buffers 消息 RegisterApplicationMasterRequest 中，主要包括以下字段（字段名称使用 Protocol Buffers 文件中定义的名称）。

- host: ApplicationMaster 本次启动所在的节点 host。用户可通过函数 RegisterApplicationMasterRequest#getHost/ RegisterApplicationMasterRequest#setHost 设置或修改该值。
- rpc_port: ApplicationMaster 本次启动对外的 RPC 端口号。用户可通过函数 RegisterApplicationMasterRequest#getRpcPort/ RegisterApplicationMasterRequest#setRpcPort 设置或修改该值。
- tracking_url: ApplicationMaster 对外提供的追踪 Web URL，客户端可通过该 tracking_url 查询应用程序执行状态。用户可通过函数 RegisterApplicationMasterRequest#getTrackingUrl/ RegisterApplicationMasterRequest#setTrackingUrl 设置或修改该值。

ApplicationMaster 注册成功后，将收到一个 RegisterApplicationMasterResponse 类型的返回值，主要包含以下信息。

- `maximumCapability` : 最大可申请的单个 Container 占用的资源量。用户可通过函数 `RegisterApplicationMasterResponse#getMaximumResourceCapability` 获取该值。
- `client_to_am_token_master_key` : `ClientToAMToken` master key。用户可通过函数 `RegisterApplicationMasterResponse#getClientToAMTokenMasterKey` 获取该值。
- `application_ACLS` : 应用程序访问控制列表。用户可通过函数 `RegisterApplicationMasterResponse#getApplicationACLs` 获取该值。

一段 (简化的) `ApplicationMaster` 向 `ResourceManager` 注册的实例代码如下:

```
// 定义一个 ApplicationMasterProtocol 协议的 RPC Client
ApplicationMasterProtocol rmClient = (ApplicationMasterProtocol) rpc.getProxy(ApplicationMasterProtocol.class, rmAddress, conf);
RegisterApplicationMasterRequest request = recordFactory.newRecordInstance(RegisterApplicationMasterRequest.class);
synchronized (this) {
    request.setApplicationAttemptId(appAttemptId);
}
...// 变量赋值
request.setHost(appHostName); // 设置所在的 host
request.setRpcPort(appHostPort); // 设置对外的 host 端口号
request.setTrackingUrl(appTrackingUrl); // 设置 tracking URL
RegisterApplicationMasterResponse response = rmClient.registerApplicationMaster(request); // 向 ResourceManager 注册
```

一旦 `ApplicationMaster` 注册成功, `ResourceManager` 会为它返回一个 `RegisterApplicationMasterResponse` 类型的返回值, 该对象包含应用程序可申请的最大资源量、应用程序访问控制列表等信息。

步骤 2 `ApplicationMaster` 通过 RPC 函数 `ApplicationMasterProtocol#allocate` 向 `ResourceManager` 申请资源 (以 Container 形式表示)。

`ApplicationMaster` 负责将应用程序需要的资源转化成 `ResourceManager` 能识别的格式, 并通过 RPC 函数 `ApplicationMasterProtocol#allocate` 告诉 `ResourceManager`。该函数只有一个 `AllocateRequest` 类型的参数, 主要包含以下几个字段 (字段名称使用了 Protocol Buffers 文件中定义的名称)。

- `ask` : `ApplicationMaster` 请求的资源列表, 每个资源请求用 `ResourceRequest` 表示, 用户可使用函数 `AllocateRequest#getAskList/AllocateRequest#getAskList` 获取或者设置请求资源列表。 `ResourceRequest` 包含以下字段。
 - `priority`: 资源优先级, 为一个正整数, 值越小, 优先级越高。
 - `resource_name` : 期望资源所在的节点或者机架, 如果是 “*”, 表示任何节点上的资源均可以。
 - `capability`: 所需的资源量, 当前支持 CPU 和内存两种资源。
 - `num_containers`: 需要满足以上条件的资源数目。
 - `relax_locality`: 是否松弛本地性, 即是否在没有满足节点本地性资源时, 自动选择

机架本地性资源或者其他资源，默认值是 true。本地化松弛源于类 MapReduce 应用中的数据本地性优化机制，它可以最大程度地提高任务的数据本地性，提高任务运行效率。

- `release` : ApplicationMaster 释放的 container 列表，当出现任务运行完成、收到的资源无法使用而主动释放资源或者主动放弃分配的 Container 等情况时，ApplicationMaster 将释放 Container。用户可使用函数 `AllocateRequest#getReleaseList/AllocateRequest#setReleaseList` 获取或者设置释放的 Container 列表。
- `response_id` : 本次通信的应答 ID，每次通信，该值会加一。用户可使用函数 `AllocateRequest#getResponseId/AllocateRequest#setResponseId` 获取或者设置 `response_id` 值。
- `progress` : 应用程序执行进度。用户可使用函数 `AllocateRequest#getProgress/AllocateRequest#setProgress` 获取或者设置 `progress` 值。
- `blacklist_request` : 请求加入 / 移除黑名单的节点列表，主要包含以下两个字段。
 - `blacklist_additions` : 请求加入黑名单的节点列表。
 - `blacklist_removals` : 请求移除黑名单的节点列表。

用户可使用函数 `AllocateRequest#getResourceBlacklistRequest/AllocateRequest#setResourceBlacklistRequest` 获取或者设置黑名单列表。通常情况下，当应用程序发现一个节点对自己运行任务不利时（比如该节点上失败的任务数目明显高于其他节点），可申请将其加入自己的黑名单，之后 `ResourceManager` 不再为它分配来自该节点的资源。

注意 即使 ApplicationMaster 不需要任何资源，它仍需周期性调用 `ApplicationMasterProtocol#allocate` 函数以维持与 `ResourceManager` 之间的心跳，否则，如果一定时间内 `ResourceManager` 未收到任何来自 ApplicationMaster 的消息，则系统会认为它已死掉了，会将其从系统中移除或者触发容错机制。除此之外，维持心跳的另外一个功能是周期性询问 `ResourceManager` 是否存在分配给应用程序的资源，如果有，需主动获取。

ApplicationMaster 每次调用 `ApplicationMasterProtocol#allocate` 后，会收到一个 `AllocateResponse` 类型的返回值，该值包含以下字段（字段名称使用了 `Protocol Buffers` 文件中定义的名称）。

- `a_m_command` : ApplicationMaster 需执行的命令，目前主要有两个取值，分别是 `AM_RESYNC` 和 `AM_SHUTDOWN`，分别表示重启和关闭。当 `ResourceManager` 重启或者应用程序信息出现不一致状态时，可能要求 ApplicationMaster 重新启动；当节点处于黑名单中时，`ResourceManager` 则让 ApplicationMaster 关闭。用户可使用函数 `AllocateResponse#getAMCommand` 获取该值。
- `response_id` : 本次通信的应答 ID，每次通信，该值会加一。用户可使用函数 `AllocateResponse#getResponseId` 获取该值。
- `allocated_containers` : 分配给该应用程序的 Container 列表。`ResourceManager` 将每份可用的资源封装成一个 Container，该 Container 中有关于这份资源的详细信息，

通常而言, ApplicationMaster 收到一个 Container 后, 会在这个 Container 中运行一个任务。用户可使用函数 `AllocateResponse#getAllocatedContainers` 获取该值。

❑ `completed_container_statuses` : 运行完成的 Container 状态列表。需要注意的是, 该列表中的 Container 所处的状态可能是运行成功、运行失败和被杀死。用户可使用函数 `AllocateResponse#getCompletedContainersStatuses` 获取该值。

❑ `limit` : 目前集群可用的资源总量。用户可使用函数 `AllocateResponse#getAvailableResources` 获取该值。

❑ `updated_nodes` : 当前集群中所有节点运行状态列表。用户可使用函数 `AllocateResponse#getUpdatedNodes` 获取该值。

❑ `num_cluster_nodes` : 当前集群中可用节点总数。用户可使用函数 `AllocateResponse#getNumClusterNodes` 获取该值。

❑ `preempt` : 资源抢占信息。当 ResourceManager 将要抢占某个应用程序的资源时, 会提前发送一个资源列表让 ApplicationMaster 主动释放这些资源, 如果 ApplicationMaster 在一定时间内未释放这些资源, 则强制进行回收。Preempt 中包含以下两类信息:

○ `strictContract` : 必须释放的 Container 列表, ResourceManager 指定要求一定要释放这些 Container 占用的资源。

○ `contract` : 它包含资源总量和 Container 列表两类信息, ApplicationMaster 可释放这些 Container 占用的资源, 或者释放任意几个占用资源总量达到指定资源量的 Container。

用户可使用函数 `AllocateResponse#getPreemptionMessage` 获取该值。

❑ `nm_tokens` : NodeManager Token。用户可使用函数 `AllocateResponse#getNMTokens` 获取该值。

注意 YARN 采用了覆盖式资源申请方式, 即 ApplicationMaster 每次发出的资源请求, 会覆盖掉之前在同一节点且优先级相同的资源请求^①, 也就是说, 同一个节点上相同优先级的资源请求只能存在一种, 比如可以全是 `<1 vcore 2048 mb>` 的 Container, 否则前面的资源会被后面申请的资源覆盖掉。

一段 (简化的) ApplicationMaster 向 ResourceManager 申请资源的实例代码如下:

```
...// 变量赋值
while(1) { // 维持与 ResourceManager 之间的周期性心跳
    synchronized (this) {
        askList = new ArrayList<ResourceRequest>(ask);
        releaseList = new ArrayList<ContainerId>(release);
        allocateRequest = BuilderUtils.newAllocateRequest(appAttemptId,
```

① 该问题已有出现在 Hadoop Jira 上, 相信在不久的将来会被解决, 具体见 <https://issues.apache.org/jira/browse/YARN-314>。

```

        lastResponseId, progressIndicator,
        askList, releaseList, null}); // 构造一个 AllocateRequest 对象
    }
    // 向 ResourceManager 申请资源, 同时领取新分配的资源
    allocateResponse = rmClient.allocate(allocateRequest);
    // 根据 ResourceManager 的应答信息设计接下来的逻辑 (比如将资源分配任务)
    ...
    Thread.sleep(1000);
}
...

```

步骤3 ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#finishApplicationMaster 告诉 ResourceManager 应用程序执行完毕, 并退出。

当 ApplicationMaster 运行完毕后, 它会调用 ApplicationMasterProtocol#finishApplicationMaster 通知 ResourceManager, 该 RPC 函数的参数类型为 FinishApplicationMasterRequest, 主要包含以下字段。

- diagnostics : 诊断信息。当 ApplicationMaster 运行失败时, 会记录错误原因以便于后期诊断。
- tracking_url: ApplicationMaster 对外提供的追踪 Web URL。
- final_application_status : ApplicationMaster 最终所处状态, 可以是 APP_UNDEFINED (未定义)、APP_SUCCEEDED (运行成功)、APP_FAILED (运行失败)、APP_KILLED (被杀死)。

成功执行该 RPC 函数后, ApplicationMaster 将收到一个 FinishApplicationMasterResponse 类型的返回值, 目前该返回值未包含任何信息。

一段 (简化的) 处理 ApplicationMaster 退出的实例代码如下:

```

FinishApplicationMasterRequest request = recordFactory
    .newRecordInstance(FinishApplicationMasterRequest.class);
request.setAppAttemptId(appAttemptId); // 设置 Application ID
request.setFinishApplicationStatus(appStatus); // 设置最终状态
if(appMessage != null) {
    request.setDiagnostics(appMessage); // 设置诊断信息
}
if(appTrackingUrl != null) {
    request.setTrackingUrl(appTrackingUrl); // 设置 trackingURL
}
rmClient.finishApplicationMaster(request); // 通知 ResourceManager 自己退出

```

ApplicationMaster 将重复步骤2, 不断为应用程序申请资源, 直到资源得到满足或者整个应用程序运行完成。

2. AM-NM 编写流程

ApplicationMaster 与 NodeManager 之间通信涉及三个步骤 (分别对应三个 API), 如图 4-6 所示, 具体如下。

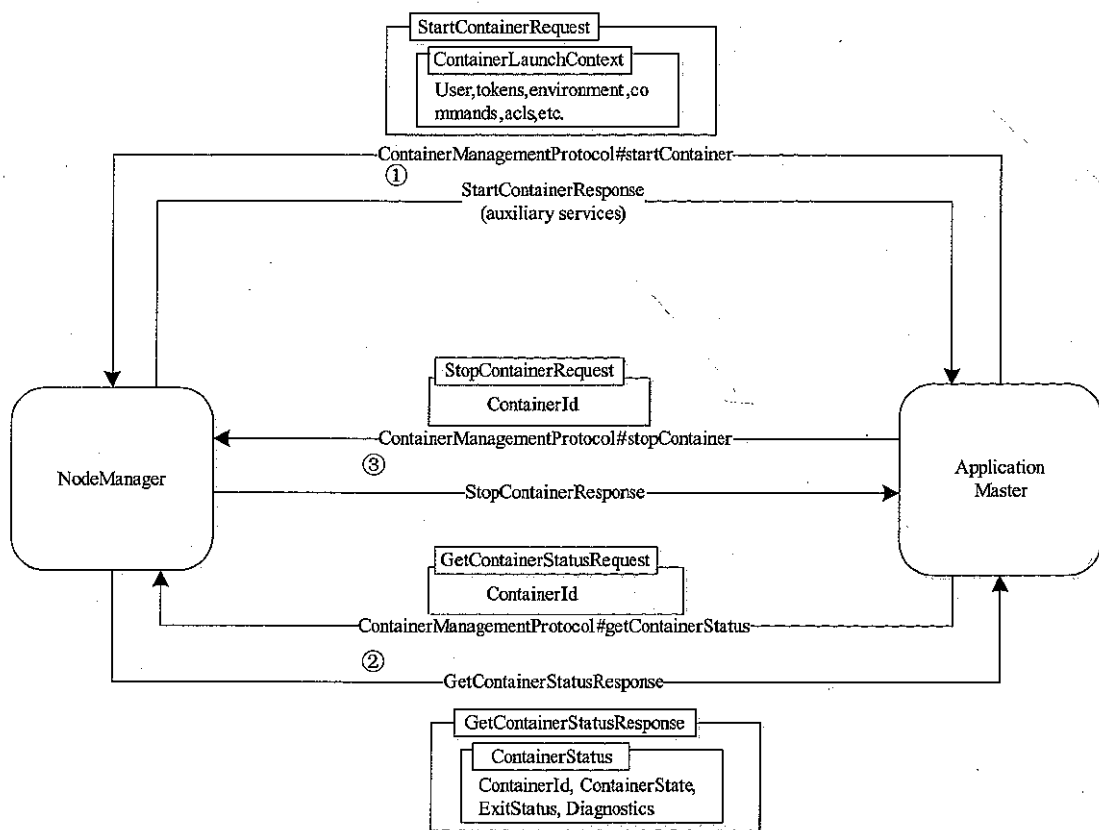


图 4-6 ApplicationMaster 与 NodeManager 通信流程

步骤 1 ApplicationMaster 将申请到的资源二次分配给内部的任务，并通过 RPC 函数 `ContainerManagementProtocol#startContainer` 与对应的 NodeManager 通信以启动 Container（包含任务描述、资源描述等信息），该函数的参数类型为 `StartContainersRequest`，主要包含一个类型为 `StartContainersRequest` 的字段，它自身包含以下两个字段（字段名称使用了 Protocol Buffers 文件中定义的名称）：

- `container_launch_context`：封装了 Container 执行环境，主要包括以下几个字段。
 - `localResources`：Container 执行所需的本地资源，比如字典文件、JAR 包或者可执行文件等，以 `key/value` 格式保存。
 - `tokens`：Container 执行所需的各种 Token。
 - `service_data`：附属服务所需的数据，以 `key/value` 格式保存。
 - `environment`：Container 执行所需的环境变量，以 `key/value` 格式保存。
 - `command`：Container 执行命令，需要是一条 Shell 命令。
 - `application_ACLS`：应用程序访问控制列表，以 `key/value` 格式保存。
- `container_token`：Container 启动时的安全令牌。

ContainerManagementProtocol#startContainer 执行成功后, 会收到一个 StartContainersResponse 类型的返回值, 该值包含以下几个字段:

- services_meta_data: 附属服务返回的元数据信息, 用户可通过函数 StartContainersResponse#getAllServicesMetadata 获取该值。
- succeeded_requests: 成功运行的 Container 列表, 用户可通过函数 StartContainersResponse#getSuccessfullyStartedContainers 获取该值。
- failed_requests: 运行失败的 Container 列表, 用户可通过函数 StartContainersResponse#getFailedRequests 获取该值。

示例代码如下:

```
...
String cmIpPortStr = container.getNodeId().getHost() + ":"
    + container.getNodeId().getPort();
InetSocketAddress cmAddress = NetUtils.createSocketAddr(cmIpPortStr);
LOG.info("Connecting to ContainerManager at " + cmIpPortStr);
this.cm = ((ContainerManagementProtocol) rpc.getProxy(ContainerManagementProtocol.class,
    cmAddress, conf));
ContainerLaunchContext ctx = Records
    .newRecord(ContainerLaunchContext.class);
// 设置 ctx 变量
...
StartContainerRequest startReq = Records
    .newRecord(StartContainerRequest.class);
startReq.setContainerLaunchContext(ctx);
startReq.setContainer(container);
try {
    cm.startContainer(startReq);
} catch (YarnRemoteException e) {
    LOG.info("Start container failed for :" + ", containerId="
        + container.getId());
    e.printStackTrace();
}
```

步骤2 为了实时掌握各个 Container 运行状态, ApplicationMaster 可通过 RPC 函数 ContainerManagementProtocol#getContainerStatus 向 NodeManager 询问 Container 运行状态, 一旦发现某个 Container 运行失败, ApplicationMaster 可尝试重新为对应的任务申请资源。

步骤3 一旦一个 Container 运行完成后, ApplicationMaster 可通过 RPC 函数 ContainerManagementProtocol#stopContainer 释放 Container。注意, YARN 是一个资源管理系统, 它不仅负责分配资源, 还负责回收资源。当一个 Container 运行完成后, 它会主动确认 Container 是否将对应的资源释放了, 也就是说, 任何一个 Container 运行结束后 (此时 Container 可能已经退出且释放资源), ApplicationMaster 必须调用 RPC 函数 ContainerManagementProtocol#stopContainer 释放 Container (确保资源真的得到释放)。

另外, 在应用程序运行过程中, 用户可使用 ApplicationClientProtocol#getApplication

Report 查询应用程序运行状态，也可以使用 ApplicationClientProtocol#forceKillApplication 将应用程序杀死。

4.3.2 ApplicationMaster 编程库

对 AM 编程库的介绍我们同样分 AM-RM 和 AM-NM 两部分进行。

1. AM-RM 编程库

同客户端编写一样，为了简化应用程序开发，ApplicationMaster 与 ResourceManager 交互部分也可以做成一个通用的编程库^①。YARN 提供的编程库涉及的类如图 4-7 所示。

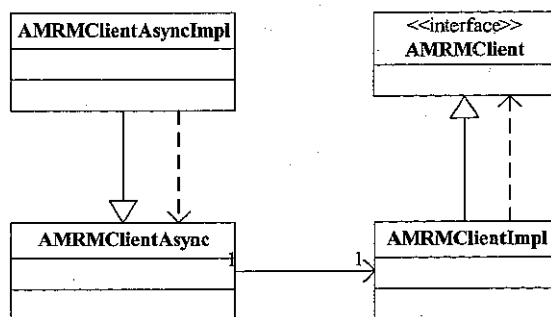


图 4-7 AM-RM 编程库

ApplicationMaster 与 ResourceManager 交互的核心逻辑均由 AMRMClientImpl 和 AMRMClientAsync 实现。其中，AMRMClientImpl 是阻塞式实现，即每个函数执行完成之后才返回；AMRMClientAsync 则是基于 AMRMClientImpl 的非阻塞式实现，ApplicationMaster 触发任何一个操作后，AMRMClientAsync 将之封装成事件放入事件队列后便返回，而事件的处理是由一个专门的线程池完成，这样整个过程变成了异步非阻塞的。当用户想要实现自己的 ApplicationMaster 时，只需实现回调类 AMRMClientAsync.CallbackHandler，该类主要提供了 5 个回调函数，分别是：

```

public interface CallbackHandler {
    /**
     * 被调用时机：ResourceManager 为 ApplicationMaster 返回的心跳应答中包含完成的
     * Container 信息。
     * 注意：如果心跳应答中同时包含完成的 Container 和新分配的 container，则该回调函数将在
     * containersAllocated 之前调用
     */
    public void onContainersCompleted(List<ContainerStatus> statuses);

    /**
     * 被调用时机：ResourceManager 为 ApplicationMaster 返回的心跳应答中包含新分配的
     * Container 信息。
     * 注意：如果心跳应答中同时包含完成的 Container 和新分配的 container，则该回调函数将在
  
```

① 参见网址 <https://issues.apache.org/jira/browse/YARN-103>。

```

    * onContainersCompleted 之后调用
    */
    public void onContainersAllocated(List<Container> containers);

    // 被调用时机: ResourceManager 通知 ApplicationMaster 停止运行
    public void onShutdownRequest();

    // 被调用时机: ResourceManager 管理的节点发生变化 (比如健康节点变得不健康, 节点不可用)
    public void onNodesUpdated(List<NodeReport> updatedNodes);

    // 被调用时机: 任何出现异常的时候
    public void onError(Exception e);
}

```

下面我们举例说明, 假设用户实现了一个 MyCallbackHandler, 代码如下:

```

class MyCallbackHandler implements AMRMClientAsync.CallbackHandler{
    ...
}

```

可以采用如下非阻塞方式使用该 MyCallbackHandler (经简化):

```

import org.apache.hadoop.yarn.client.api.async.AMRMClientAsync;
..... // 其他 Java 包
AMRMClientAsync.CallbackHandler allocListener = new MyCallbackHandler();
// 构造一个 AMRMClientAsync 句柄
asyncClient = AMRMClientAsync.createAMRMClientAsync(1000, allocListener);
asyncClient.init(conf); // 通过传入一个 YarnConfiguration 对象进行初始化
asyncClient.start(); // 启动 asyncClient
// ApplicationMaster 向 ResourceManager 注册
RegisterApplicationMasterResponse response = asyncClient
    .registerApplicationMaster(appMasterHostname, appMasterRpcPort,
        appMasterTrackingUrl);
... // 添加 Container 请求
asyncClient.addContainerRequest(containerRequest);
... // 等待应用程序运行结束
asyncClient.unregisterApplicationMaster(status, appMsg, null);
asyncClient.stop();

```

除了实现以上几个回调函数外, AMRMClientAsync 还提供了以下几个接口供用户编写 ApplicationMaster 使用, 分别是:

```

public RegisterApplicationMasterResponse registerApplicationMaster(
    String appHostName, int appHostPort, String appTrackingUrl)
    throws YarnRemoteException; // 向 ResourceManager 注册 ApplicationMaster
// 通知 ResourceManager 注销 ApplicationMaster
public void unregisterApplicationMaster(FinalApplicationStatus appStatus,
    String appMessage, String appTrackingUrl)
    throws YarnRemoteException;
// 添加资源 (Container) 请求
public void addContainerRequest(AMRMClient.ContainerRequest req);
// 移除资源 (Container) 请求

```

```

public void removeContainerRequest (AMRMClient.ContainerRequest req);
// 请求释放资源
public void releaseAssignedContainer (ContainerId containerId);
...

```

2. AM-NM 编程库

同客户端和 AM-RM 编写一样，为了简化应用程序开发，ApplicationMaster 与 NodeManager 交互部分也可以做成一个通用的编程库^①。需要注意的是，由于 ResourceManager 也要与 NodeManager 交互以启动应用程序的 ApplicationMaster，因此该编程库也可用在 ResourceManager 实现逻辑中，因此该编程库的名称不再是 AMNMClient，而是 NMClient。YARN 提供的 NMClient 编程库涉及的类如图 4-8 所示。

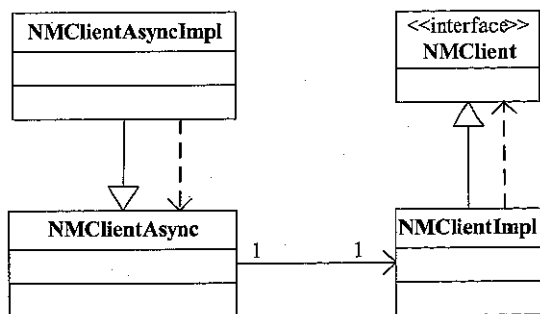


图 4-8 AM-NM 编程库

ApplicationMaster 与 NodeManager 交互的核心逻辑均由 NMClientImpl 和 NMClientAsync 实现，其中，NMClientImpl 是阻塞式实现，即每个函数执行完成之后才返回；NMClientAsync 则是基于 NMClientImpl 的非阻塞式实现，ApplicationMaster 触发任何一个操作后，NMClientAsync 将之封装成事件放入事件队列后便返回，而事件的处理是由一个专门的线程池完成，这样整个过程变成了异步非阻塞的。当用户想要实现自己的 ApplicationMaster 时，只需实现回调类 NMClientAsync.CallbackHandler，该类主要提供了 6 个回调函数，分别是：

```

public static interface CallbackHandler {
    // 当接收到启动 Container 请求时被调用
    void onContainerStarted(ContainerId containerId,
        Map<String, ByteBuffer> allServiceResponse);

    // 当 NodeManager 应答（对之前发送查询状态指令的应答）Container 当前状态时被调用
    void onContainerStatusReceived(ContainerId containerId,
        ContainerStatus containerStatus);

    // 当 NodeManager 应答（对之前发送停止 container 指令的应答）Container 已停止时被调用
    void onContainerStopped(ContainerId containerId);
}

```

① 参见网址 <https://issues.apache.org/jira/browse/YARN-422>。

```

// 当 NodeManager 启动 Container 过程中抛出异常时被调用
void onStartContainerError(ContainerId containerId, Throwable t);

// 当查询 container 运行状态时抛出异常时被调用
void onGetContainerStatusError(ContainerId containerId, Throwable t);

// 当 NodeManager 停止 container 过程中抛出异常时被调用
void onStopContainerError(ContainerId containerId, Throwable t);
}

```

下面举例说明, 假设用户实现了一个 MyCallbackHandler, 代码如下:

```

class MyCallbackHandler implements NMClientAsync.CallbackHandler{
    ...
}

```

可以采用如下非阻塞方式使用该 MyCallbackHandler (经简化):

```

import org.apache.hadoop.yarn.client.api.async.NMClientAsync;
import org.apache.hadoop.yarn.client.api.async.impl.NMClientAsyncImpl;
... // 其他 Java 包
// 构造一个 NMClientAsync 句柄
NMClientAsync asyncClient = new NMClientAsyncImpl(new MyCallbackHandler());
asyncClient.init(conf); // 初始化 asyncClient
asyncClient.start(); // 启动 asyncClient
// 构造 Container 信息
ContainerLaunchContext ctx = Records.newRecord(ContainerLaunchContext.class);
... // 设置 ctx 变量
// 启动 Container
asyncClient.startContainerAsync(container, ctx);
// 获取 Container 状态
asyncClient.getContainerStatusAsync(container.getId(),
    container.getNodeId(), container.getContainerToken());
// 停止 Container
asyncClient.stopContainerAsync(container.getId(), container.getNodeId(),
    container.getContainerToken());
asyncClient.stop();

```

4.4 YARN 应用程序实例

本节介绍两个 YARN 自己带的 Application 示例程序: DistributedShell 和 UnManaged AM。需要注意的是, 尽管这两个示例程序非常简单, 但是它们已经具有了一个应用程序具备的所有功能, 其他任何应用程序均可在这两个程序基础上扩展而来。

4.4.1 DistributedShell

顾名思义, DistributedShell 是一个可以分布式运行 Shell 命令的应用程序, 它可以并行执行用户提供的 Shell 命令或者 Shell 脚本。

DistributedShell 的使用方法如下:

```
bin/hadoop jar \
share/hadoop/yarn/hadoop-yarn-applications-distributedshell-*.jar \
org.apache.hadoop.yarn.applications.distributedshell.Client
[COMMAND_OPTIONS]
```

其中，COMMAND_OPTIONS 的参数及其含义如表 4-1 所示。

表 4-1 COMMAND_OPTIONS 的参数

名 称	含 义
-appname <arg>	应用程序名称，默认值为 DistributedShell
-priority <arg>	应用程序优先级，默认是 0
-queue <arg>	应用程序优先级被提交到的队列的名称
-timeout <arg>	应用程序超时时间（单位：毫秒）
-master_memory <arg>	应用程序 AM 占用的内存（单位：MB）
-jar <arg>	包含 AM 实现的 JAR 包
-shell_command <arg>	运行的 Shell 命令
-shell_script <arg>	运行的 Shell 脚本
-shell_args <arg>	运行的 Shell 脚本的输入参数
-shell_env <arg>	运行 Shell 脚本所需的环境变量，以 key=env_val 的形式定义
-shell_cmd_priority <arg>	Shell 命令所在 Container 的优先级
-container_memory <arg>	运行 Shell 命令或者脚本所需的内存
-num_containers <arg>	Shell 命令并发执行数目
-log_properties <arg>	指定 log4j.properties 文件
-debug	输出调试信息
-help	打印使用说明信息

运行示例如下：

```
bin/hadoop jar\
share/hadoop/yarn/hadoop-yarn-applications-distributedshell-*.jar\
org.apache.hadoop.yarn.applications.distributedshell.Client\
--jar share/hadoop/yarn/hadoop-yarn-applications-distributedshell-*.jar\
--shell_command ls\
--num_containers 10\
--container_memory 350\
--master_memory 350\
--priority 10
```

接下来重点介绍 DistributedShell 的客户端和 ApplicationMaster 实现方法。

(1) 客户端实现

客户端的实现方法与前面描述的步骤完全一致，主要提供了两个函数用于应用程序提交（并监控它的运行过程直到运行完成）和杀死应用程序。在应用程序提交时，Client 首先从参数中获取各个属性值，并构造出 ApplicationSubmissionContext 对象，将应用程序提交到 YARN 上，但需要注意两点。

1) Shell 脚本处理。如果用户指定了一个 Shell 脚本, 则客户端首先将脚本内容写入 HDFS 上的 “/user/\$USER/\$appName/\$appId/ExecShellScript.sh” 文件中, 并将该文件信息写入到以下几个环境变量中。

□ DISTRIBUTEDSHELLSCRIPTLOCATION: Shell 脚本在 HDFS 上的位置。

□ DISTRIBUTEDSHELLSCRIPTTIMESTAMP: Shell 脚本最近修改时间。

□ DISTRIBUTEDSHELLSCRIPTLEN: Shell 脚本长度。

而 ApplicationMaster 则会获取以上三个环境变量信息, 用于验证文件的可用性以及进一步将文件加入到文件分发列表中。

2) 构建 ApplicationMaster 启动命令。ApplicationSubmissionContext 对象中最重要的一个字段是 ApplicationMaster 启动命令, 而 DistributedShell 客户端构造内容如下:

```
/bin/java -Xmx 350m \
org.apache.hadoop.yarn.applications.distributedshell.ApplicationMaster \
--container_memory 350 \
--num_containers 10 \
--priority 10 \
--shell_command ls \
1> <LOG_DIR>/AppMaster.stdout \
2> <LOG_DIR>/AppMaster.stderr
```

ApplicationMaster 将会读取这些参数进一步完成资源申请和 Container 启动等工作。

(2) ApplicationMaster 实现

ApplicationMaster 启动时, 将所需的所有资源一次性发送给 ResourceManager, 相关代码如下:

```
for (int i = 0; i < numTotalContainers; ++i) {
    ContainerRequest containerAsk = setupContainerAskForRM();
    // resourceManager 是一个 AMRMClientAsync 对象
    resourceManager.addContainerRequest(containerAsk);
}

private ContainerRequest setupContainerAskForRM() {
    Priority pri = Records.newRecord(Priority.class); // 设置优先级
    pri.setPriority(requestPriority); // 设置资源量
    Resource capability = Records.newRecord(Resource.class);
    capability.setMemory(containerMemory);
    ContainerRequest request = new ContainerRequest(capability, null, null, pri);
    return request;
}
```

一旦 ApplicationMaster 收到一个 Container 后, 将启动一个独立线程与对应的 NodeManager 通信, 以运行任务; 与此同时, 如果发现某个 Container 被杀死, ApplicationMaster 会为它重新申请资源。DistributedShell ApplicationMaster 中的 ResourceManager 回调类实现如下:

```
private class RMCallbackHandler implements AMRMClientAsync.CallbackHandler {
    public void onContainersCompleted(List<ContainerStatus> completedContainers) {
        for (ContainerStatus containerStatus : completedContainers) {
```



```

// Container 应该处于完成状态, 可能是成功、失败或者被杀死三个状态之一
assert (containerStatus.getState() == ContainerState.COMPLETED);
// increment counters for completed/failed containers
int exitStatus = containerStatus.getExitStatus();
if (0 != exitStatus) { // Container 运行失败
    if (ContainerExitStatus.ABORTED != exitStatus) {
        numCompletedContainers.incrementAndGet();
        numFailedContainers.incrementAndGet();
    } else { // Container 被杀死, 此时需重新为它申请资源
        numAllocatedContainers.decrementAndGet();
        numRequestedContainers.decrementAndGet();
    }
} else { // Container 成功运行完成
    numCompletedContainers.incrementAndGet();
}
}

// 如果 Container 是被杀死的, 需重新为它申请资源
int askCount = numTotalContainers - numRequestedContainers.get();
numRequestedContainers.addAndGet(askCount);
if (askCount > 0) {
    ContainerRequest containerAsk = setupContainerAskForRM(askCount);
    resourceManager.addContainerRequest(containerAsk);
}
// 将进度汇报给 ResourceManager
float progress = (float) numCompletedContainers.get()
    / numTotalContainers;
resourceManager.setProgress(progress);
if (numCompletedContainers.get() == numTotalContainers) {
    done = true;
}
}

public void onContainersAllocated(List<Container> allocatedContainers) {
    numAllocatedContainers.addAndGet(allocatedContainers.size());
    for (Container allocatedContainer : allocatedContainers) {
        LaunchContainerRunnable runnableLaunchContainer = new LaunchContainerRunnable(
            allocatedContainer);
        Thread launchThread = new Thread(runnableLaunchContainer);
        // 每个 Container 由一个独立的线程启动
        launchThreads.add(launchThread);
        launchThread.start();
    }
}

public void onRebootRequest() {}
public void onNodesUpdated(List<NodeReport> updatedNodes) {}
}

```

DistributedShell ApplicationMaster 中的 NodeManager 回调类实现较简单, 有兴趣的读者可自行阅读源代码。

4.4.2 Unmanaged AM

在 YARN 中, 一个 ApplicationMaster 需要占用一个 Container, 该 Container 可能位于任意一个 NodeManager 上, 这给 ApplicationMaster 调试带来很大麻烦。为了解决该问题, YARN 引入了一种新的 ApplicationMaster——Unmanaged AM^①, 这种 AM 运行在客户端, 不再由 ResourceManager 启动和销毁。用户只需在提交应用程序时设置一个参数, YARN 便允许用户将 ApplicationMaster 运行在客户端的一个单独进程中。

Unmanaged AM 工作步骤如下。

- 1) 通过 RPC 函数 ApplicationClientProtocol#getNewApplication 获取一个 Application ID。
- 2) 创建一个 ApplicationSubmissionContext 对象, 填充各个字段, 并通过调用函数 ApplicationClientProtocol.setUnmanagedAM(true) 设置启用 Unmanaged AM 的标志位。
- 3) 通过 RPC 函数 ApplicationClientProtocol#submitApplication 将应用程序提交到 ResourceManager 上, 并监控 Application 运行状态, 直到其状态变为 YarnApplicationState.ACCEPTED。
- 4) 在客户端中的一个独立线程中启动 ApplicationMaster, 然后等待 ApplicationMaster 运行结束和 ResourceManager 报告应用程序运行结束。

YARN 同样给出了一个 Unmanaged AM 的示例程序 UnmanagedAMLauncher, 在该程序中, 用户可指定一个运行在 AM 中的 Shell 命令, 一旦该命令运行结束后, AM 便会退出, 进而整个应用程序退出。

UnmanagedAMLauncher 的使用方法如下:

```
bin/hadoop jar \
share/hadoop/yarn/hadoop-yarn-applications-unmanaged-am-launcher-*.jar \
org.apache.hadoop.yarn.applications.unmanagedamlauncher.UnmanagedAMLauncher
[COMMAND_OPTIONS]
```

其中, COMMAND_OPTIONS 的参数及其含义如表 4-2 所示。

表 4-2 COMMAND_OPTIONS 的参数

名 称	含 义
-appname <arg>	应用程序名称, 默认值为 UnmanagedAM
-classpath	额外需要的 classpath
-priority <arg>	应用程序优先级, 默认是 0
-queue <arg>	应用程序优先级被提交到的队列的名称
-cmd <arg>	启动 unmanaged AM 的命令 (必须的)
-master_memory <arg>	应用程序 AM 占用的内存 (单位: MB)
-help	打印使用说明信息

有兴趣的读者可仿照 UnmanagedAMLauncher 实现一个 Unmanaged AM 应用程序。

① 参见网址 <https://issues.apache.org/jira/browse/YARN-420>。

4.5 源代码阅读引导

为了帮助读者更好地阅读源代码，笔者特意安排了本节。我们建议读者：

- 通信协议：YARN 中主要的通信协议 `ApplicationClientProtocol`、`ApplicationMasterProtocol` 和 `ContainerManagementProtocol` 的 Protocol Buffers 定义描述在 `hadoop-yarn-project/hadoop-yarn/hadoop-yarn-api/src/main/proto` 中，Java 描述在目录 `hadoop-yarn-project/hadoop-yarn/hadoop-yarn-api/src/main/java` 下的 `org.apache.hadoop.yarn.api` 包中。
- 编程库：本章介绍的各个编程库在 `hadoop-yarn-project/hadoop-yarn/hadoop-yarn-client/src/main/java` 目录下，组织结构如图 4-9 所示。

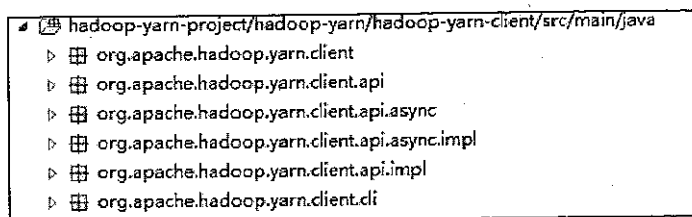


图 4-9 YARN 编程库的组织结构

其中，`org.apache.hadoop.yarn.client.api.*` 是为用户提供的编程库实现，`org.apache.hadoop.yarn.client.cli` 则是客户端通过 Shell 命令行实现，`org.apache.hadoop.yarn.client` 则实现了一个组合各种通信协议的客户端代理。

- YARN 编程实例：本章介绍的两个 Application 编程实例 `DistributedShell` 和 `Unmanaged AM` 相关实现在目录 `hadoop-yarn-project/hadoop-yarn/hadoop-yarn-applications/` 中，读者可仿照这两个非常简单的实例学习 YARN Application 的编写方法。

4.6 小结

本章介绍了 YARN Application 的程序设计方法。当用户想要编写一个可以运行在 YARN 上的应用程序时，通常要实现两个组件，分别是客户端和 `ApplicationMaster`，其中，客户端主要用于提交应用程序和管理应用程序，而 `ApplicationMaster` 则负责实现应用程序的任务切分、调度、监控等功能。

为了便于用户实现应用程序的客户端和 `ApplicationMaster`，YARN 提供了可供用户直接使用的客户端编程库、AM-RM 编程库和 AM-NM 编程库。

4.7 问题讨论

问题 1：改写 `DistributedShell` 程序，使得每个 Container 运行在不同节点上（目前是随机的，可能运行在任意节点上）。

问题 2：改写 DistributedShell 程序，使得某个用户指定的命令可以在集群的每个节点上仅执行一次。

问题 3：扩展 Unmanaged AM 程序，使得 AM 可以向 ResourceManager 申请一定量的资源并在申请到的 Container 中启动用户指定的命令。

问题 4：使用 Thrift RPC 或者 Avro RPC 作为本章涉及的三个主要协议（即 ApplicationClientProtocol、ApplicationMasterProtocol 和 ContainerManagementProtocol）的 RPC（替换现有的基于 Hadoop RPC 的实现），并使用非 Java 语言（比如 C++）实现 DistributedShell。

第 5 章 ResourceManager 剖析

同 MRv1 一样, YARN 也采用了 Master/Slave 结构, 其中, Master 实现为 ResourceManager, 负责整个集群资源的管理与调度; Slave 实现为 NodeManager, 负责单个节点的资源管理与任务启动。本章重点介绍 ResourceManager 的实现。

ResourceManager 是整个 YARN 集群中最重要的组件之一, 它的设计直接决定了系统的可扩展性、可用性和容错性等特点, 它的功能较多, 包括 ApplicationMaster 管理 (启动、停止等)、NodeManager 管理、Application 管理、状态机管理等, 本章将对每个功能点展开详细讨论。

5.1 概述

在 YARN 中, ResourceManager 负责集群中所有资源的统一管理和分配, 它接收来自各个节点 (NodeManager) 的资源汇报信息, 并把这些信息按照一定的策略分配给各个应用程序 (实际上是 ApplicationMaster)。

5.1.1 ResourceManager 基本职能

整体上讲, ResourceManager 需通过两个 RPC 协议与 NodeManager 和 (各个应用程序的) ApplicationMaster 交互, 如图 5-1 所示, 具体如下。

- **ResourceTracker**: NodeManager 通过该 RPC 协议向 ResourceManager 注册、汇报节点健康状况和 Container 运行状态, 并领取 ResourceManager 下达的命令, 这些命令包括重新初始化、清理 Container 等, 在该 RPC 协议中, ResourceManager 扮演 RPC Server 的角色 (由内部组件 ResourceTrackerService 实现), 而 NodeManager 扮演 RPC Client 的角色, 换句话说, NodeManager 与 ResourceManager 之间采用了 “pull 模型” (与 MRv1 类似), NodeManager 总是周期性地主动向 ResourceManager 发起请求, 并通过领取下达给自己的命令。
- **ApplicationMasterProtocol**: 应用程序的 ApplicationMaster 通过该 RPC 协议向 ResourceManager 注册、申请资源和释放资源。在该协议中, ApplicationMaster 扮演 RPC Client 的角色, 而 ResourceManager 扮演 RPC Server 的角色, 换句话说, ResourceManager 与 ApplicationMaster 之间也采用了 “pull 模型”。
- **ApplicationClientProtocol**: 应用程序的客户端通过该 RPC 协议向 ResourceManager 提交应用程序、查询应用程序状态和控制应用程序 (比如杀死应用程序) 等。在

该协议中,应用程序客户端扮演 RPC Client 的角色,而 ResourceManager 扮演 RPC Server 的角色。

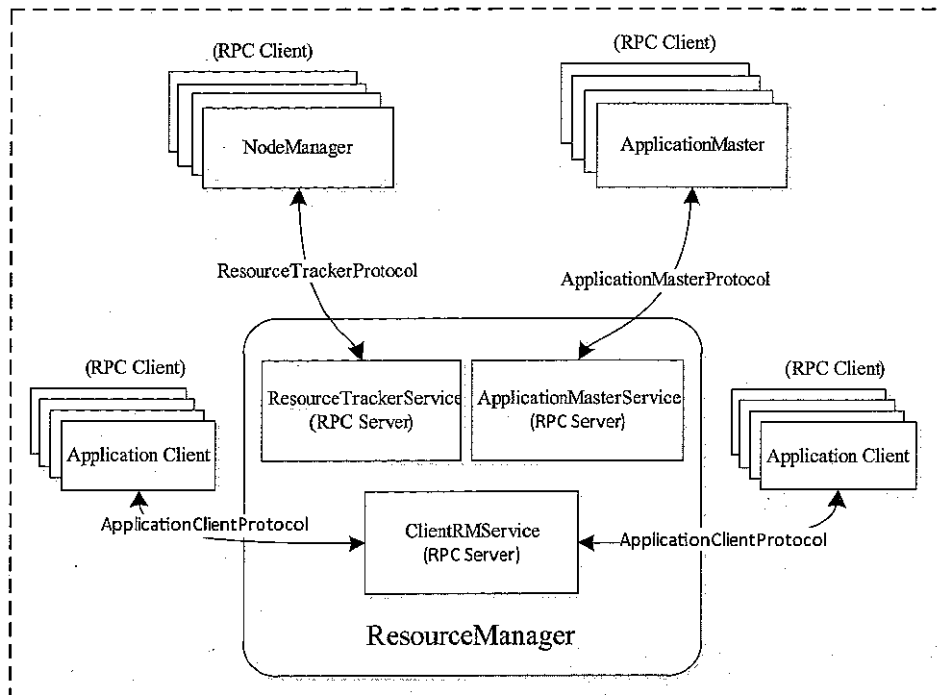


图 5-1 ResourceManager 相关的 RPC 协议

概括起来, ResourceManager 主要完成以下几个功能:

- 与客户端交互, 处理来自客户端的请求;
- 启动和管理 ApplicationMaster, 并在它运行失败时重新启动它;
- 管理 NodeManager, 接收来自 NodeManager 的资源汇报信息, 并向 NodeManager 下达管理指令 (比如杀死 Container 等);
- 资源管理与调度, 接收来自 ApplicationMaster 的资源申请请求, 并为之分配资源。

5.1.2 ResourceManager 内部架构

本小节深入 NodeManager 内部, 介绍它的内部组织结构和主要模块, 如图 5-2 所示, ResourceManager 主要由以下几个部分组成:

- **用户交互模块。** ResourceManager 分别针对普通用户、管理员和 Web 提供了三种对外服务, 具体实现分别对应 ClientRMService、AdminService 和 WebApp。
- **ClientRMService。** ClientRMService 是为普通用户提供的服务, 它处理来自客户端各种 RPC 请求, 比如提交应用程序、终止应用程序、获取应用程序运行状态等。
- **AdminService。** ResourceManager 为管理员提供了一套独立的服务接口, 以防

止大量的普通用户请求使管理员发送的管理命令饿死，管理员可通过这些接口管理集群，比如动态更新节点列表、更新 ACL 列表、更新队列信息等。

○WebApp。为了更加友好地展示集群资源使用情况和应用程序运行状态等信息，YARN 对外提供了一个 Web 界面，这一部分是 YARN 仿照 Haml^①开发的一个轻量级嵌入式 Web 框架^②。

□NM 管理模块。该模块主要涉及以下组件：

○NMLivelinessMonitor。监控 NM 是否活着，如果一个 NodeManager 在一定时间（默认为 10min）内未汇报心跳信息，则认为它死掉了，需将其从集群中移除。

○NodesListManager。维护正常节点和异常节点列表，管理 exclude（类似于黑名单）和 include（类似于白名单）节点列表，这两个列表均是在配置文件中设置的，可以动态加载。

○ResourceTrackerService。处理来自 NodeManager 的请求，主要包括注册和心跳两种请求，其中，注册是 NodeManager 启动时发生的行为，请求包中包含节点 ID、可用的资源上限等信息；而心跳是周期性行为，包含各个 Container 运行状态，运行的 Application 列表、节点健康状况（可通过一个脚本设置）等信息，作为请求的应答，ResourceTrackerService 可为 NodeManager 返回待释放的 Container 列表、Application 列表等信息。

□AM 管理模块。该模块主要涉及以下组件：

○AMLivelinessMonitor。监控 AM 是否活着，如果一个 ApplicationMaster 在一定时间（默认为 10min）内未汇报心跳信息，则认为它死掉了，它上面所有正在运行的 Container 将被置为失败状态，而 AM 本身会被重新分配到另外一个节点上（用户可指定每个 ApplicationMaster 的尝试次数，默认是 2）执行。

○ApplicationMasterLauncher。与某个 NodeManager 通信，要求它为某个应用程序启动 ApplicationMaster。

○ApplicationMasterService (AMS)。处理来自 ApplicationMaster 的请求，主要包括注册和心跳两种请求，其中，注册是 ApplicationMaster 启动时发生的行为，注册请求包中包含 ApplicationMaster 启动节点；对外 RPC 端口号和 tracking URL 等信息；而心跳则是周期性行为，汇报信息包含所需资源描述、待释放的 Container 列表、黑名单列表等，而 AMS 则为之返回新分配的 Container、失败的 Container、待抢占的 Container 列表等信息。

□Application 管理模块。该模块主要涉及以下组件：

○ApplicationACLsManager。管理应用程序访问权限，包含两部分权限：查看权限和修改权限。查看权限主要用于查看应用程序基本信息，而修改权限则主要用于修改应用程序优先级、杀死应用程序等。

① 参见网址 <http://haml.info/>。

② 参见网址 <https://issues.apache.org/jira/browse/MAPREDUCE-2399>。

- RMAppManager。管理应用程序的启动和关闭。
- ContainerAllocationExpirer。当 AM 收到 RM 新分配的一个 Container 后，必须在一定的时间（默认为 10min）内在对应的 NM 上启动该 Container，否则 RM 将强制回收该 Container，而一个已经分配的 Container 是否该被回收则是由 ContainerAllocationExpirer 决定和执行的。
- 状态机管理模块。ResourceManager 使用有限状态机维护有状态对象的生命周期，正如第 3 章所介绍的，状态机的引入使得 YARN 设计架构更加清晰。ResourceManager 共维护了 4 类状态机，分别是 RMAppl、RMApplAttempt、RMContainer 和 RMNode（这几个均是接口，具体实现类为对应接口名加上“Impl”后缀）。
 - RMAppl。RMAppl 维护了一个应用程序（Application）的整个运行周期，包括从启动到运行结束整个过程。由于一个 Application 的生命周期可能会启动多个 Application 运行实例（Application Attempt），因此可认为，RMAppl 维护的是同一个 Application 启动的所有运行实例的生命周期。
 - RMApplAttempt。一个应用程序可能启动多个实例，即一个实例运行失败后，可能再次启动一个重新运行，而每次启动称为一次运行尝试（或者“运行实例”，用“RMApplAttempt”描述，RMApplAttempt 维护了一次运行尝试的整个生命周期。
 - RMContainer。RMContainer 维护了一个 Container 的运行周期，包括从创建到运行结束整个过程。ResourceManager 将资源封装成 Container 发送给应用程序的 ApplicationMaster，而 ApplicationMaster 则会在 Container 描述的运行环境中启动任务，因此，从这个层面上讲，Container 和任务的生命周期是一致的（目前 YARN 尚不支持 Container 重用，一个 Container 用完后会立刻释放，将来可能会增加 Container 重用机制）。
 - RMNode。RMNode 维护了一个 NodeManager 的生命周期，包括启动到运行结束整个过程。
- 安全管理模块。ResourceManage 自带了非常全面的权限管理机制，主要由 ClientToAMSecretManager、ContainerTokenSecretManager、ApplicationTokenSecretManager 等模块完成。
- 资源分配模块。该模块主要涉及一个组件——ResourceScheduler。ResourceScheduler 是资源调度器，它按照一定的约束条件（比如队列容量限制等）将集群中的资源分配给各个应用程序，当前主要考虑内存和 CPU 资源。ResourceScheduler 是一个插拔式模块，YARN 自带了一个批处理资源调度器——FIFO（First In First Out）和两个多用户调度器——Fair Scheduler 和 Capacity Scheduler（默认资源调度器）^①，具体将在第 6 章介绍。

① 在 2.0.2-alpha 版本之前，YARN 默认调度器是 FIFO，但由于 FIFO 在负载均衡和其他方面存在一些问题，可能会影响用户体验，之后被换为 Capacity Scheduler。默认情况下，Capacity Scheduler 是单队列的，且队列内部调度策略与 FIFO 一致。

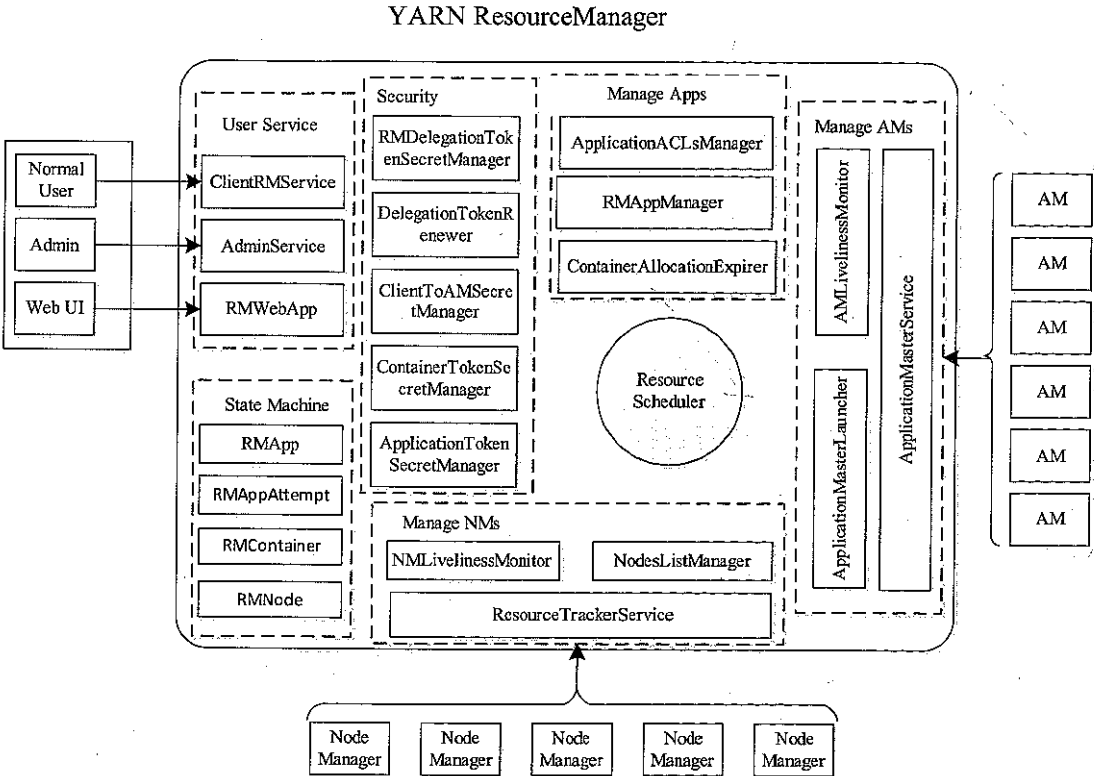


图 5-2 ResourceManager 内部架构图

5.1.3 ResourceManager 事件与事件处理器

前几章提到 YARN 采用了事件驱动的机制，而 ResourceManager 的实现则是一个最好的例证。表 5-1 给出了 ResourceManager 内部各个事件和服务以及它们处理和输出的事件类型。所有服务和组件均是通过中央异步调度器组织在一起的，不同组件之间通过事件交互，从而实现了一个异步并行的高效系统。

表 5-1 NodeManager 内部事件与事件处理器

组件名称	事件处理器 / 服务	处理的事件类型	输出事件类型
ClientRMService	服务	—	RMAppEventType
NMLivelinessMonitor	服务	—	RMNodeEventType
NodesListManager	事件处理器服务	NodesListManagerEvent	—
ResourceTrackerService	服务	—	RMNodeEventType
AMLivelinessMonitor	服务	—	RMAppAttemptEventType
ApplicationMasterLauncher	事件处理器服务	AMLancherEventType	—
RMAppManager	事件处理器	RMAppManagerEventType	RMAppEventType

(续)

组件名称	事件处理器 / 服务	处理的事件类型	输出事件类型
ContainerAllocationExpirer	服务	—	SchedulerEventType
RMAApp (Application EventDispatcher)	事件处理器	RMAAppEventType	RMAAppAttemptEventType RMNodeEventType
RMAAppAttempt (ApplicationAttempt EventDispatcher)	事件处理器	RMAAppAttemptEventType	SchedulerEventType RMAAppEventType
RMContainer	事件处理器	RMContainerEventType	RMAAppAttemptEventType RMNodeEventType
RMNode (NodeEventDispatcher)	事件处理器	RMNodeEventType	SchedulerEventType RMNodeEventType NodesListManagerEventType
ResourceScheduler (Scheduler EventDispatcher)	事件处理器	SchedulerEventType	RMContainerEventType RMAAppAttemptEventType RMNodeEventType

图 5-3 所示从事件驱动角度展示了 ResourceManager 内部各类事件与事件处理器的相互关系。

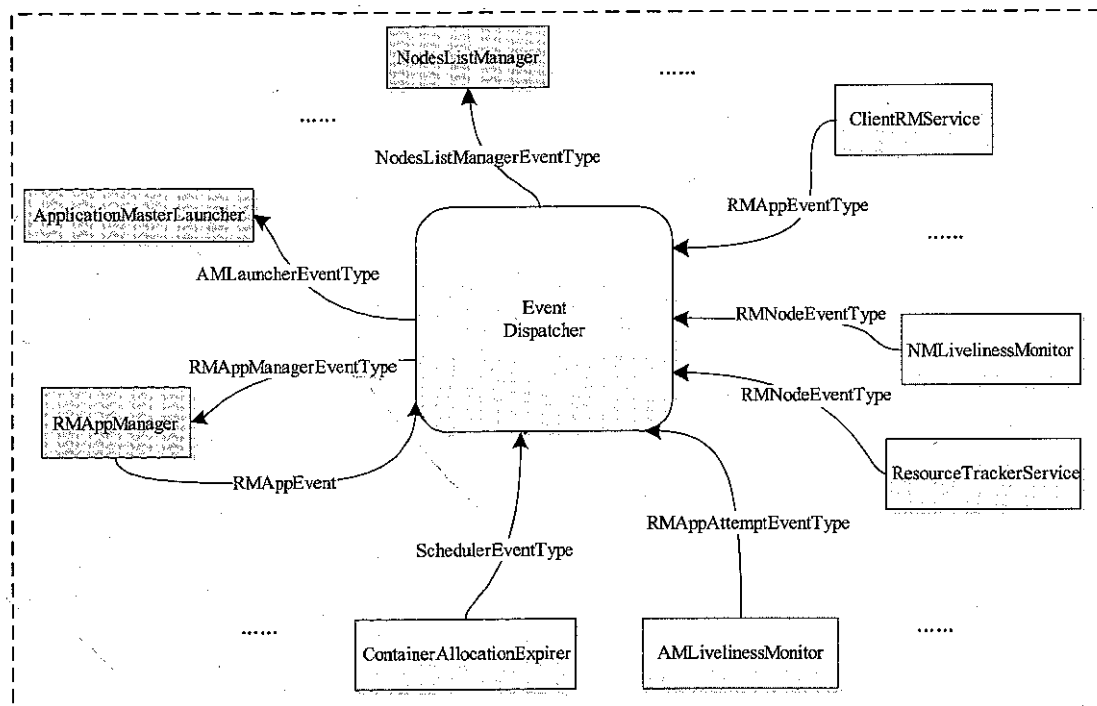


图 5-3 ResourceManager 内部事件与事件处理器交互图

5.2 用户交互模块

在 ResourceManager 中, ClientRMService 和 AdminService 两个服务分别负责处理来自普通用户和管理员的请求, 需要注意的是, 之所以让这两类请求通过两个不同的通信通道发送给 ResourceManager, 是因为要避免普通用户请求过多导致管理员请求被阻塞而迟迟得不到处理。

5.2.1 ClientRMService

ClientRMService 是一个 RPC Server, 为来自客户端的各种 RPC 请求提供服务。从实现角度看, 它是一个实现了 ApplicationClientProtocol 协议的服务, 其定义如下:

```
public class ClientRMService extends AbstractService implements
    ApplicationClientProtocol {
    .....
    protected void serviceStart() throws Exception {
        Configuration conf = getConfig();
        YarnRPC rpc = YarnRPC.create(conf);
        this.server = // 实现了 RPC 协议 ApplicationClientProtocol
            rpc.getServer(ApplicationClientProtocol.class, this,
                clientBindAddress,
                conf, this.rmDTSecretManager,
                conf.getInt(YarnConfiguration.RM_CLIENT_THREAD_COUNT,
                    YarnConfiguration.DEFAULT_RM_CLIENT_THREAD_COUNT));
        ...
    }
    ...
}
```

ClientRMService 类中保留了一个 ResourceManager 上下文对象 RMContext, 通过该对象可获知 ResourceManager 中绝大部分信息, 包括节点列表、队列组织、应用程序列表等, 这样 ClientRMService 可很容易通过查询 RMContext 中的信息为来自客户端的请求做出应答。RMContext 的实现类是 RMContextImpl, 它包含的主要信息如下:

```
public class RMContextImpl implements RMContext {
    private final Dispatcher rmDispatcher; // 中央异步调度器

    private final ConcurrentMap<ApplicationId, RMApp> applications
        = new ConcurrentHashMap<ApplicationId, RMApp>(); // 应用程序列表

    private final ConcurrentMap<NodeId, RMNode> nodes
        = new ConcurrentHashMap<NodeId, RMNode>(); // 节点列表

    private final ConcurrentMap<String, RMNode> inactiveNodes
        = new ConcurrentHashMap<String, RMNode>(); // 非活跃节点列表

    private AMLivelinessMonitor amLivelinessMonitor; // 运行中的 AM 心跳监控
```

```

private AMLivelinessMonitor amFinishingMonitor; // 运行完成的 AM 心跳监控
private RMStateStore stateStore = null; // ResourceManager 状态保存处
// Container 超时监控, 应用程序必须在一定时间内使用分配到的 Container, 否则将被回收
private ContainerAllocationExpirer containerAllocationExpirer;
// 以下是安全相关的组件
private final DelegationTokenRenewer tokenRenewer;
private final AMRMTokenSecretManager appTokenSecretManager;
private final RMContainerTokenSecretManager containerTokenSecretManager;
private final NMTokenSecretManagerInRM nmTokenSecretManager;
private final ClientToAMTokenSecretManagerInRM clientToAMTokenSecretManager;
...
}

```

5.2.2 AdminService

与 ClientRMService 类似, AdminService 也是一个 RPC Server, 但它的服务对象是管理员。在 YARN 中, 管理员列表由属性 yarn.admin.acl 指定 (在 yarn-site.xml 中设置), 默认情况下, 属性值为 “*”, 表示所有用户都是管理员。从实现角度看, 它是一个实现了 ResourceManagerAdministrationProtocol 协议的服务, 其定义如下:

```

public class AdminService extends AbstractService implements ResourceManagerAdministrationProtocol {
    ...
    protected void serviceStart() throws Exception {
        Configuration conf = getConfig();
        YarnRPC rpc = YarnRPC.create(conf);
        this.server = // 实现了 RPC 协议 ResourceManagerAdministrationProtocol
            rpc.getServer(ResourceManagerAdministrationProtocol.class, this, masterServiceAddress,
                conf, null,
                conf.getInt(YarnConfiguration.RM_ADMIN_CLIENT_THREAD_COUNT,
                    YarnConfiguration.DEFAULT_RM_ADMIN_CLIENT_THREAD_COUNT));
    }
    ...
}

```

5.3 ApplicationMaster 管理

ApplicationMaster 管理部分主要由三个服务构成, 分别是 ApplicationMasterLauncher、AMLivelinessMonitor 和 ApplicationMasterService, 它们共同管理应用程序的 ApplicationMaster 的生存周期。在正式介绍这三个组件之前, 先总结一下这三个组件是如何协同管理 ApplicationMaster 生命周期的。由于 ResourceManager 为 ApplicationMaster 申请资源的过程涉及多个状态机之间的转换, 因此该部分将放在后面介绍, 本节主要介绍从 ResourceManager 获得资源启动 ApplicationMaster, 后续涉及的一系列流程具体如图 5-4 所示。

步骤 1 用户向 YARN ResourceManager 提交应用程序, ResourceManager 收到提交请求后, 先向资源调度器申请用以启动 ApplicationMaster 的资源, 待申请到资源后, 再由

ApplicationMasterLauncher 与对应的 NodeManager 通信，从而启动应用程序的 ApplicationMaster。

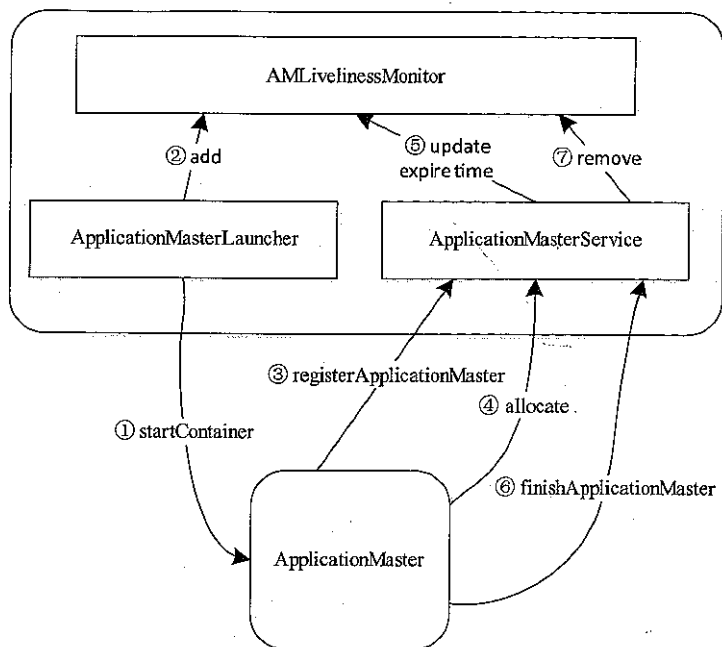


图 5-4 ApplicationMaster 启动流程

步骤 2 ApplicationMaster 启动完成后，ApplicationMasterLauncher 会通过事件的形式，将刚刚启动的 ApplicationMaster 注册到 AMLivelinessMonitor，以启动心跳监控。

步骤 3 ApplicationMaster 启动后，先向 ApplicationMasterService 注册，并将自己所在 host、端口号等信息汇报给它。

步骤 4 ApplicationMaster 运行过程中，周期性地向 ApplicationMasterService 汇报“心跳”信息（“心跳”信息中包含想要申请的资源描述）。

步骤 5 ApplicationMasterService 每次收到 ApplicationMaster 的心跳信息后，将通知 AMLivelinessMonitor 更新该应用程序的最近汇报心跳的时间。

步骤 6 当应用程序运行完成后，ApplicationMaster 向 ApplicationMasterService 发送请求，注销自己。

步骤 7 ApplicationMasterService 收到注销请求后，标注应用程序运行状态为完成，同时通知 AMLivelinessMonitor 移除对它的心跳监控。

接下来依次介绍上述的三个服务。

(1) ApplicationMasterLauncher

ApplicationMasterLauncher 既是一个服务，也是一个事件处理器，它处理 AMLauncherEvent 类型的事件，该类型事件有两种，分别是请求启动一个 ApplicationMaster 的“LAUNCH”事件和请求清理一个 ApplicationMaster 的“CLEANUP”事件。ApplicationMasterLauncher 维

护了一个线程池，从而能够尽快地处理这两种事件。

□ 如果 ApplicationMasterLauncher 收到了“LAUNCH”类型的事件，它会与对应的 NodeManager 通信，要求它启动 ApplicationMaster。整个过程比较简单，首先创建一个 ContainerManagementProtocol 协议的客户端，然后向对应的 NodeManager 发起连接请求，接着将启动 AM 所需的各种信息，包括启动命令、JAR 包、环境变量等信息，封装成一个 StartContainerRequest 对象，然后通过 RPC 函数 ContainerManagementProtocol#startContainer 发送给对应的 NM。

□ 如果 ApplicationMasterLauncher 收到了“CLEANUP”类型的事件，它与对应的 NodeManager 通信，要求它杀死 ApplicationMaster。整个过程与启动 AM 的过程类似。

(2) AMLivelinessMonitor

该服务周期性遍历所有应用程序的 ApplicationMaster，如果一个 ApplicationMaster 在一定时间（可通过参数 yarn.am.liveness-monitor.expiry-interval-ms 配置，默认为 10min）内未汇报心跳信息，则认为它死掉了，它上面所有正在运行的 Container 将被置为运行失败（RM 不会重新执行这些 Container，它只会通过心跳机制告诉对应的 AM，由 AM 决定是否重新执行。如果需要，则 AM 重新向 RM 申请资源）；如果 AM 运行失败，则由 RM 重新为它申请资源，以便能够重新分配到另外一个节点上（用户可在提交应用程序时通过函数 ApplicationSubmissionContext#setMaxAppAttempts 设置 ApplicationMaster 重试次数，如果未设置，则采用全局参数 yarn.resourcemanager.am.max-attempts 设置的值，默认是 2）执行。

(3) ApplicationMasterService

ApplicationMasterService 实现了 RPC 协议 ApplicationMasterProtocol，负责处理来自 ApplicationMaster 的请求，请求主要包括注册、心跳和清理三种，其中，注册是 ApplicationMaster 启动时发生的行为，请求包中包含 AM 所在节点、RPC 端口号和 tracking URL 等信息；心跳是周期性行为，包含请求资源的类型描述、待释放的 Container 列表等，而 AMS 为之返回新分配的 Container、失败的 Container 等信息；清理是应用程序运行结束时发生的行为，ApplicationMaster 向 RM 发送清理应用程序的请求，以回收资源和清理各种内存空间。

ApplicationMasterLauncher 启动 AM 后，AM 做的第一件事是向 RM 注册，这是通过 RPC 函数 ApplicationMasterProtocol#registerApplicationMaster 实现的。

AM 运行过程中，需要周期性地通过 RPC 函数 ApplicationMasterProtocol#allocate 与 RM 通信，这主要有以下三个作用：

- 请求资源；
- 获取新分配的资源；
- 形成周期性心跳，告诉 RM 自己还活着。

AM 运行结束后，需要通过 RPC 函数 ApplicationMasterProtocol#finishApplicationMaster 告诉 RM 自己运行结束，可以回收资源（AM 所占用的 Container）和清理各种数据结果了（比如将该 AM 从 AMLivelinessMonitor 监控列表中删除）。

5.4 NodeManager 管理

NodeManager 管理部分主要由三个服务构成，分别是 NMLivelinessMonitor、NodesListManager 和 ResourceTrackerService，它们共同管理 NodeManager 的生存周期，接下来我们依次介绍这三个服务。

(1) NMLivelinessMonitor

该服务周期性遍历集群中所有 NodeManager，如果一个 NodeManager 在一定时间（可通过参数 `yarn.nm.liveness-monitor.expiry-interval-ms` 配置，默认为 10min）内未汇报心跳信息，则认为它死掉了，它上面所有正在运行的 Container 将被置为运行失败。需要注意的是，RM 不会重新执行这些 Container，它只会通过心跳机制告诉对应的 AM，由 AM 决定是否重新执行。如果需要，则 AM 重新向 RM 申请资源，然后由 AM 与对应的 NodeManager 通信以重新运行失败的 Container。

(2) NodesListManager

NodesListManager 管理 `exclude`（类似于黑名单）和 `include`（类似于白名单）节点列表，这两个列表所在的文件分别可通过 `yarn.resourcemanager.nodes.include-path` 和 `yarn.resourcemanager.nodes.exclude-path` 配置（每个节点的 host 或者 IP 占一行），其中，`exclude` 节点列表可认为是黑名单，它们不允许直接与 RM 通信（直接在 RPC 层抛出异常，导致 NM 退出），而 `include` 节点列表可认为是白名单（通常不将这两个名单直接称为黑名单和白名单，这两个概念被用到了 YARN 其他地方）。默认情况下，这两个列表均为空，表示任何节点均被允许接入 RM。需要注意的是，管理员可通过命令“`bin/yarn rmdadmin-refreshNodes`”动态加载这两个文件。

(3) ResourceTrackerService

ResourceTrackerService 实现了 RPC 协议 ResourceTracker，负责处理来自各个 NodeManager 的请求，请求主要包括注册和心跳两种，其中，注册是 NodeManager 启动时发生的行为，请求包中包含节点 ID、可用的资源上限等信息；而心跳是周期性行为，包含各个 Container 运行状态，运行的 Application 列表、节点健康状况（可通过一个脚本设置，具体阅读第 7 章），而 ResourceTrackerService 则为 NM 返回待释放的 Container 列表、Application 列表等。

NM 启动时，它所做的第一件事是向 RM 注册，这是通过 RPC 函数 `ResourceTracker#registerNodeManager` 实现的，注册信息包括节点可用资源总量、对外开放的 HTTP 端口号等。

注意 一个节点总的可用资源量是在 NodeManager 注册时汇报给 ResourceManager 的，之后整个运行过程中不能动态修改（若修改需要重启 NodeManager），但 YARN-291^①正尝试引入新的 RPC 协议以支持动态修改一个节点上的可用资源。

① 参见网址 <https://issues.apache.org/jira/browse/YARN-291>。

NM 启动后, 它会周期性地通过 RPC 函数 `ResourceTracker#nodeHeartbeat` 汇报心跳, 心跳信息包含各个 Container 运行状态、运行的 Application 列表、节点健康状况等信息, 而 RM 则为之返回需要释放的 Container 列表、Application 列表等。

5.5 Application 管理

在 YARN 中, Application 是指应用程序, 它可能启动多个运行实例, 每个运行实例由一个 `ApplicationMaster` 与一组该 `ApplicationMaster` 启动的任务组成, 它拥有名称、队列名、优先级等属性, 是一个比较宽泛的概念, 可以是一个 MapReduce 作业、一个 DAG 应用程序, 甚至可以是一个 Storm 集群实例。YARN 中 Application 管理涉及应用程序的权限管理、启动与关闭、生命周期管理等, 本节只介绍最基本的管理内容, 比如权限管理、启动与关闭等, 而生命周期管理则放到下一节中介绍。

(1) ApplicationACLsManager

`ApplicationACLsManager` 负责管理应用程序的访问权限, 包含两部分权限: 查看权限和修改权限。其中, 查看权限主要用于查看应用程序基本信息, 比如运行时间、优先级等信息; 而修改权限则主要用于修改应用程序优先级、杀死应用程序等。默认情况下, 任意一个普通用户可以查看所有其他用户的应用程序。用户可以为自己的应用程序设置具有访问权限的用户列表, 具体方法是在客户端使用 `ContainerLaunchContext#newInstance` 构造 `ContainerLaunchContext` 实例时将其作为参数传入。比如, 假设用户 `userX` 编写了一个应用程序, 则下面代码赋予用户 `user1` 和用户组 `group1` 查看权限, 并赋予 `user1` 修改权限:

```
...
Map<ApplicationAccessType, String> acls
    = new HashMap<ApplicationAccessType, String>(2);
acls.put(ApplicationAccessType.VIEW_APP, "user1 group1");
acls.put(ApplicationAccessType.MODIFY_APP, "user1");

// 为 AM Container 构造 ContainerLaunchContext
ContainerLaunchContext amContainer =
    ContainerLaunchContext.newInstance(localResources, environment,
        vargsFinal, null, securityTokens, acls);
...
ApplicationSubmissionContext appContext =
    recordFactory.newRecordInstance(ApplicationSubmissionContext.class);
appContext.setAMContainerSpec(amContainer);
```

通过以上设置之后, 应用程序拥有者 `userX`、集群管理员 (通过参数 `yarn.admin.acl` 设置在 `yarn-site.xml` 文件中) 和 `user1` 具有查看和修改权限, 用户组 `group1` 具有查看权限。

通常而言, 为了便于用户设置该参数, 运行在 YARN 之上的计算框架会预留一些参数供用户提交应用程序时动态设置, 比如 MapReduce 计算框架允许用户通过参数 `mapreduce.job.acl-view-job` 和 `mapreduce.job.acl-modify-job` 为每个应用程序设置查看和修改权限。

(2) RMAppManager

RMAppManager 负责应用程序的启动和关闭。ClientRMService 收到来自客户端的提交应用程序请求后, 将调用函数 RMAppManager#submitApplication 创建一个 RMApp 对象, 它将维护这个应用程序的整个生命周期, 从开始运行到最终结束 (具体将在 5.6 节介绍); 当 RMApp 运行结束后, 将向 RMAppManager 发送一个 RMAppManagerEventType.APP_COMPLETED 事件, 它收到该事件后将调用 RMAppManager#finishApplication 进行收尾工作, 包括:

- 将该应用程序放入已完成应用程序列表中, 以便用户查询历史应用程序运行信息。需要注意的是, 该列表的大小是有限的, 默认是 10000 (管理员可通过参数 yarn.resourcemanager.max-completed-applications 修改), 当已完成应用程序数目超过该值时, 将从内存数据结构中移除 (移除的应用程序可称为“过期的应用程序”), 这样用户只能通过 History Server 获取过期的应用程序信息, History Server 是从磁盘文件中获取这些信息的 (应用程序会将运行日志和基本信息写到磁盘上)。
- 将应用程序从 RMStateStore 中移除。RMStateStore 记录了运行中的应用程序的运行日志, 当集群故障重启后, ResourceManager 可通过这些日志恢复应用程序运行状态, 从而避免全部重新运行, 一旦应用程序运行结束后, 这些日志便失去了意义, 故可以对其进行删除。这属于 ResourceManager 容错机制的范畴 (具体将在 5.9 节介绍)。

(3) ContainerAllocationExpirer

当一个 AM 获得一个 Container 后, YARN 不允许 AM 长时间不对其使用, 因为这会降低整个集群的利用率。当 AM 收到 RM 新分配的一个 Container 后, 必须在一定的时间 (默认为 10min, 管理员可通过参数 yarn.resourcemanager.rm.container-allocation.expiry-interval-ms 修改) 内在对应的 NM 上启动该 Container, 否则 RM 将强制回收该 Container。

5.6 状态机管理

在 YARN 中, 如果一个对象由若干个状态以及触发这些状态发生转移的事件构成, 它将被抽象成一个状态机, 在 YARN ResourceManager 内部, 共有 4 类状态机, 分别是 RMApp、RMAppAttempt、RMContainer 和 RMNode。其中, 前 2 类状态机维护了一个应用程序相关的生命周期, 包括 Application 生命周期、一次运行尝试的生命周期; RMContainer 则维护了分配出去各个资源的使用状态; RMNode 维护了一个 NodeManager (一个节点上可以有多个 NodeManager) 的生命周期。

YARN 中的 Application 生命周期由状态机 RMAppImpl 维护, 每个 Application 可能会尝试运行多次, 每次称为一次“运行尝试” (Application Attempt, 也可称为运行实例), 而每次运行尝试的生命周期则由状态机 RMAppAttemptImpl 维护, 如果一次运行尝试 (实例) 运行失败, RMApp 会创建另外一个运行尝试, 直到某次运行尝试运行成功或

者达到运行尝试上限。对于每次运行尝试, ResourceManager 将为它分配一个 Container, Container 是运行环境的抽象, 内部封装了任务的运行环境和资源等信息, 而一个应用程序的 ApplicationMaster 就运行在这个 Container 中。ApplicationMaster 启动之后, 会不断向 ResourceManager 申请 Container 以运行各类任务。Container 的生命周期由状态机 RMContainerImpl 维护, 整个组织结构可参照图 5-5。

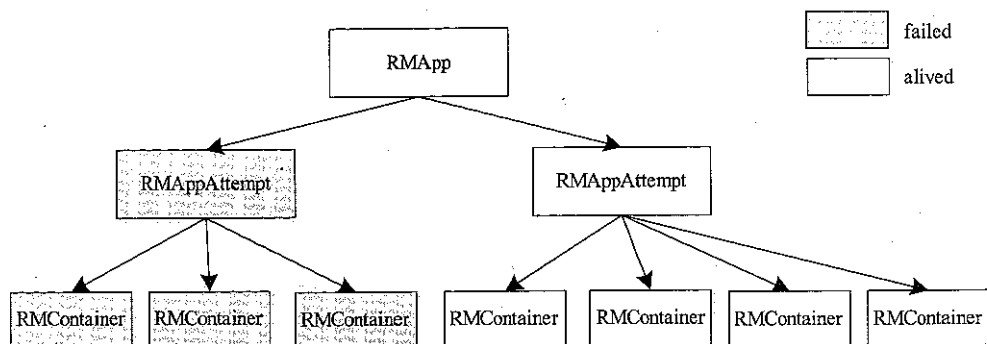


图 5-5 应用程序状态机组织结构

Application Attempt 的生命周期与 ApplicationMaster 的生命周期基本上是一致的：一个 Application 内部所有任务均由 ApplicationMaster 维护和管理, ApplicationMaster 本身需要占用一个 Container, 而这个 Container 由 ResourceManager 为其申请和启动。一旦 ApplicationMaster 成功启动, 它就会与 ResourceManager 通信, 为它内部的任务申请 Container。如果 ApplicationMaster 重新启动, 则意味着一个新的 Application Attempt 被启动, 换句话说, 一个 Application Attempt 的“生死存亡”与 ApplicationMaster 的“命运”紧紧绑定在一起。

除了 Application 相关的 3 类状态机, 还有一类维护 NodeManager 生命周期的状态机 RMNodeImpl, 本节也会进行详细介绍。

5.6.1 RMAppl 状态机

RMAppl 是 ResourceManager 中用于维护一个 Application 生命周期的数据结构, 它的实现是 RMApplImpl 类, 该类维护了一个 Application 状态机, 记录了一个 Application 可能存在的各个状态 (RMApplState) 以及导致状态间转换的事件 (RMApplEvent)。当某个 RMApplEventType 类型的事件发生时, RMApplImpl 会根据实际情况进行状态转移, 同时触发一个行为 (实际是一个回调函数)。除了维护 Application 状态机外, RMApplImpl 还保存了 Application 基本信息 (比如名称、所在队列名称、启动时间等) 和迄今为止所有的运行尝试 (Application Attempt) 信息。

如图 5-6 所示, 在 RM 看来, 每个 Application 有 9 种基本状态 (RMApplState) 和 12 种导致这 9 种状态之间发生转移的事件 (RMApplEventType), RMApplImpl 的作用是等待接

收其他对象发出的 `RMApEvent` 类型的事件，然后根据当前状态和事件类型，将当前状态转移到另外一种状态，同时触发一种行为（实际就是执行一个函数，该函数可能会再次发出某种类型的事件），下面具体进行介绍。

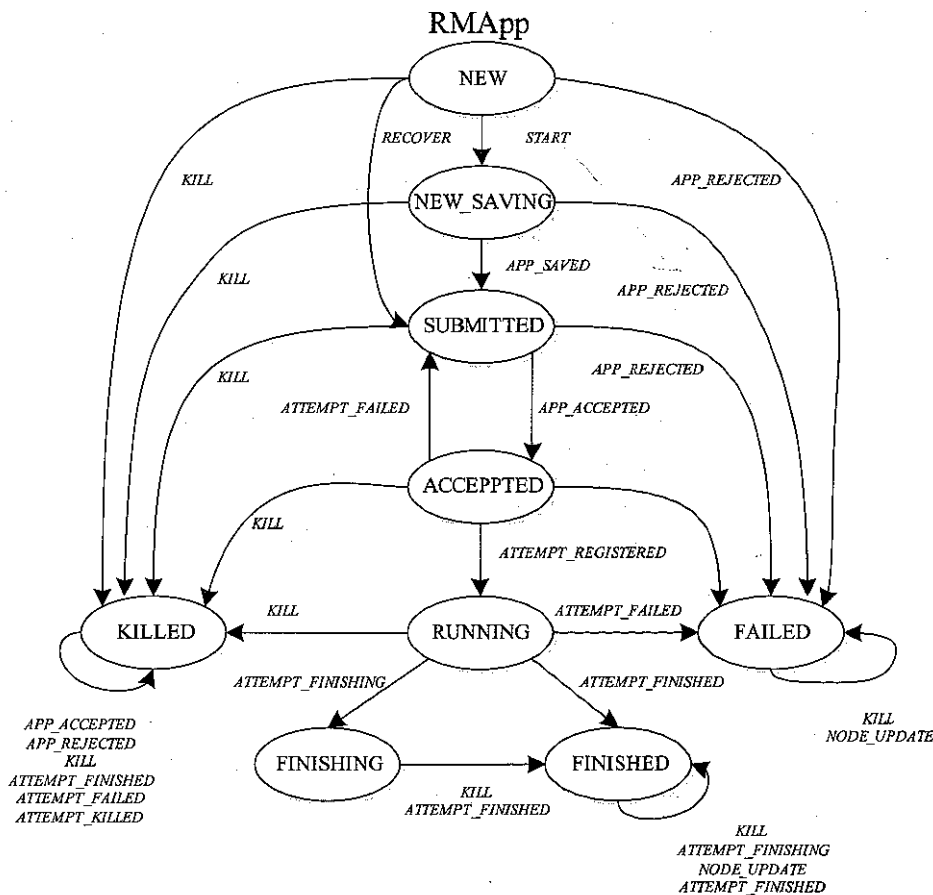


图 5-6 RMAp 状态机

(1) 基本状态

基本状态有以下几种：

- **NEW**：状态机初始状态。每个 Application 对应一个状态机，而每个状态机的初始状态为 NEW。
- **NEW_SAVING**：日志记录应用程序基本信息时所处的状态。ResourceManager 收到用户提交的应用程序后，将为它创建一个 `RMApImpl` 对象以维护它的状态，之后要做的第一件事是记录该应用程序的基本信息，以便故障重启后可以自动恢复运行该应用程序。
- **SUBMITTED**：应用程序已提交状态。客户端通过 RPC 函数 `ApplicationClientProtoc`

ol#submitApplication 向 RM 提交一个 Application，通过合法性验证以及完成日志记录后，RM 会创建了一个 RMApplAttemptImpl 对象，以进行第一次运行尝试，并将 Application（运行）状态置为 SUBMITTED。

- **ACCEPTED**：资源调度器同意接受该应用程序后所处状态。应用程序不仅要在 ClientRMService 中进行合法性检查，也要经资源调度器的合法性检查，比如 Capacity Scheduler 允许管理员配置应用程序提交数目上限，如果超过该上限，则拒绝接受新提交的应用程序。
- **RUNNING**：该应用程序的 ApplicationMaster 已经成功在某个节点上开始运行，这意味着该应用程序的 RMApplAttemptImpl 也已经处于运行状态中。
- **FAILED**：该 Application 的 ApplicationMaster 运行失败时所处的状态。多种原因可能导致 ApplicationMaster 运行失败，比如硬件故障、软件 bug、OOM（Out Of Memory）等。需要注意的是，状态机收到 ATTEMPT_FAILED 事件后不一定会立即转入 FAILED 状态，而是先检查失败次数是否超过用户设置的最大上限（如果用户未设置该值，则采用 ResourceManager 中参数 yarn.resourcemanager.am.max-attempts 设置的全局最大值，默认是 2），如果没有，则再次创建一个 RMApplAttemptImpl 对象，并让状态机重新回到 SUBMITTED 状态，否则才最终会转入 FAILED 状态，这意味着应用程序彻底运行失败。
- **KILLED**：该 Application 被杀死时所处的状态，通常是由于收到来自客户端杀死应用程序命令后，ResourceManager 主动将 ApplicationMaster 杀死。
- **FINISHING**：当 Application Master 通过 RPC 函数 ApplicationMasterProtocol#finishApplicationMaster 通知 RM，自己运行结束将要退出时，Application 将被置为 FINISHING 状态。
- **FINISHED**：NodeManager 通过心跳汇报 ApplicationMaster 所在的 Container 运行结束，RMApplImpl 将被置为 FINISHED 状态，这意味着该应用程序成功运行结束。

(2) 基本事件

基本事件主要包括：

- **STARTED**：客户端调用 RPC 函数 ApplicationClientProtocol#submitApplication 提交应用程序后，会触发 STARTED 事件。
- **RECOVER**：如果管理员开启了应用程序恢复功能（默认不开启，可通过参数 yarn.resourcemanager.recovery.enabled 配置），则 ResourceManager 重启后，会向已提交但尚未开始运行的应用程序发送 RECOVER 事件。
- **KILL**：客户端调用 RPC 函数 ApplicationClientProtocol#forceKillApplication 杀死 Application，此时会触发一个 KILL 事件。
- **APP_REJECTED**：多种情况下会触发 APP_REJECTED 事件，包括客户端通过 RPC 函数 ApplicationClientProtocol#submitApplication 向 RM 提交一个 Application 时，若抛出 IOException 异常，则会触发一个 APP_REJECTED 事件；若资源调度器

认为应用程序非法（比如所在队列不存在或者已达到应用程序数目上限等），则拒绝接受该应用程序，同样最终会触发 APP_REJECTED 事件。

□ APP_ACCEPTED：当资源调度器认为应用程序合法时，将同意接受该应用程序，最终会触发 APP_ACCEPTED 事件。

□ APP_SAVED：用户提交应用程序后，ResourceManager 首先要将应用程序信息保存到磁盘上，以便故障恢复时使用。一旦信息保存完成后，将触发一个 APP_SAVED 事件。

□ ATTEMPT_REGISTERED：应用程序的 ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#registerApplicationMaster 向 RM 注册时，将触发一个 ATTEMPT_REGISTERED 事件。

□ ATTEMPT_FINISHING：当应用程序的 ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#finishApplicationMaster 向 RM 汇报自己运行完成时，会触发一个 ATTEMPT_FINISHING 事件。

□ ATTEMPT_FINISHED：某个 NodeManager 通过心跳汇报 ApplicationMaster 所在的 Container 运行结束时，会触发一个 ATTEMPT_FINISHED 事件。

□ NODE_UPDATE：NodeManager 每次汇报心跳信息时会触发一个 NODE_UPDATE 事件，该事件会被广播给所有先关的应用程序。

□ ATTEMPT_FAILED：应用程序的 ApplicationMaster 运行失败（可能是由于软件 bug、硬件故障等原因导致 ApplicationMaster 自身运行失败，也可能是它内部任务失败数目过多致使 ApplicationMaster 主动退出）时，会触发一个 ATTEMPT_FAILED 事件。需要注意的是，状态机收到 ATTEMPT_FAILED 事件后不一定会立即转入 FAILED 状态，而是先检查失败次数是否超过用户设置的最大上限，如果没有，则再次创建一个 RMAppAttemptImpl 对象，并让状态机重新回到 SUBMITTED 状态，否则才最终会转入 FAILED 状态，这意味着应用程序彻底运行失败。

□ ATTEMPT_KILLED：应用程序的 ApplicationMaster 被杀死时，会触发一个 ATTEMPT_KILLED 事件。

注意 应用程序状态机处于 RUNNING 状态时，可能直接转换为 FINISHED 状态，也可能先转换为 FINISHING 状态，然后再转换为 FINISHED 状态，这主要是由于 ApplicationMaster 汇报自己运行完成和 ApplicationMaster 所在 Container 退出运行这两个事件没有明确的时间先后之分，比如应用程序的 ApplicationMaster 可能没有调用 ApplicationMasterProtocol#finishApplicationMaster 通知 ResourceManager 自己将退出运行，而是直接退出运行；导致状态机进入 FINISHED 状态的唯一事件是 ApplicationMaster 所在 Container 正常退出。

图 5-7 描述了各个事件的来源，这些事件主要来自 ClientRMService 和 RMAppAttemptImpl 两类组件。

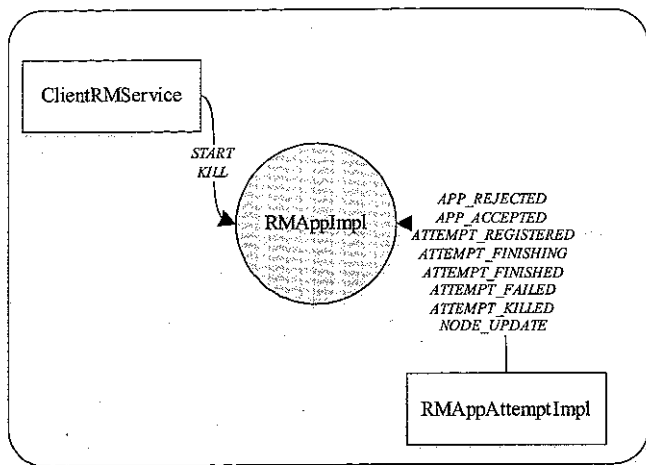


图 5-7 RMAApp 状态机事件来源

5.6.2 RMAAppAttempt 状态机

RMAAppAttempt 是 ResourceManager 中用于维护一个 Application 运行尝试（或者称为“Application Attempt”）的生命周期的数据结构，它的实现是 RMAAppAttemptImpl，该类维护了一个状态机，记录了一个 Application Attempt 可能存在的各个状态以及导致状态间转换的事件。当某个事件发生时，RMAAppAttemptImpl 会根据实际情况进行 Application Attempt 状态转移，同时触发一个行为。除了维护状态机外，RMAAppAttempt 还保存了本次运行尝试的基本信息，包括当前使用的 Container 信息、ApplicationMaster Container 对外 tracking URL 和 RPC 端口号等。由于在一次运行尝试中，最重要的组件是 ApplicationMaster，它的当前状态可代表整个应用程序的当前状态，因此，RMAAppAttemptImpl 本质上是维护的 ApplicationMaster 生命周期。

如图 5-8 所示，在 RM 看来，每个 RMAAppAttemptImpl 有 13 种基本状态（RMAAppAttemptState）和 15 种导致这 13 种状态之间发生转移的事件（RMAAppAttemptEvent），RMAAppAttemptImpl 的作用是等待接收其他对象发出的 RMAAppAttemptEventType 类型的事件，然后根据当前状态和事件类型，将当前状态转移到另外一种状态，同时触发另外一种行为（实际上执行一个函数，该函数可能会再次发出一种其他类型的事件），下面具体进行介绍。

（1）基本状态

基本状态包括：

- ❑ NEW：状态机初始状态，每个 Application Attempt 对应一个状态机，而每个状态机的初始状态为 NEW。
- ❑ SUBMITTED：RMAAppImpl 创建 RMAAppAttempt 之后，将在第一时间向它发送一个 START 事件，为其创建各种 Token 后，将其状态置为 SUBMITTED。

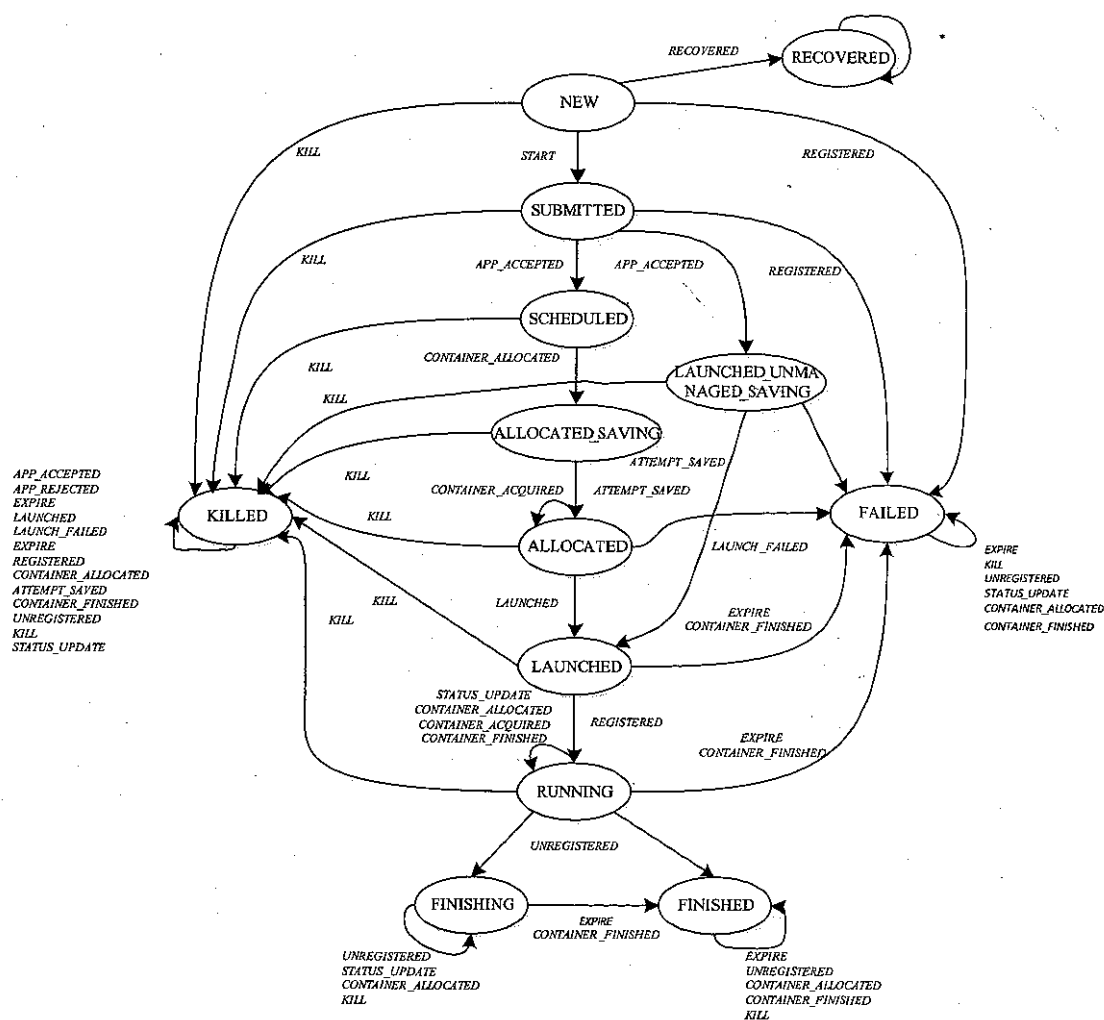


图 5-8 RMApAttempt 状态机

- **SCHEDULED** : RMApAttemptImpl 被创建之后, ResourceManager 会将它添加到 ResourceScheduler 中, 通过 ResourceScheduler 合法性检查后, 状态将被置为 SCHEDULED, 这表明 ResourceScheduler 开始为该 Application 的 ApplicationMaster 分配资源。
- **ALLOCATED_SAVING** : RMApAttemptImpl 接收到 ResourceScheduler 分配的一个 Container 后 (用于启动 ApplicationMaster), 会将该 Container 信息写入到磁盘上以便故障恢复使用。信息保存完成之前, RMApAttemptImpl 将处于 ALLOCATED_SAVING 状态。
- **ALLOCATED** : RMApAttemptImpl 将收到的 Container 信息保存到文件中, 以

便于失败后从磁盘上恢复，信息保存完成之后，RMAppAttemptImpl 状态转换为 ALLOCATED。

- ❑ LAUNCHED：分配到 Container 后，ResourceManager 中的 ApplicationMasterLauncher 与对应的 NodeManager 通信，以启动 ApplicationMaster，此时 RMAppAttemptImpl 将被置为 LAUNCHED 状态。
- ❑ RUNNING：ApplicationMaster 在 NodeManager 上成功启动后，将通过 RPC 函数 ApplicationMasterProtocol#registerApplicationMaster 向 ResourceManager 注册，此时 RMAppAttemptImpl 状态被置为 RUNNING。
- ❑ FAILED：如果 Application 的 ApplicationMaster 运行失败（通过超时机制检测），RMAppAttemptImpl 的状态将转换为 FAILED。
- ❑ KILLED：如果客户端发出杀死应用程序命令，RMAppAttemptImpl 将被置为 KILLED。FINISHING：Application Master 通过 RPC 函数 ApplicationMasterProtocol#finishApplicationMaster 通知 RM，自己将运行结束，此时 RMAppAttemptImpl 被置为 FINISHING 状态。
- ❑ FINISHED：NodeManager 通过心跳汇报 ApplicationMaster 所在的 Container 运行结束，此时 RMAppAttemptImpl 被置为 FINISHED 状态。
- ❑ LAUNCHED_UNMANAGED_SAVING：为了方便对 ApplicationMaster 进行测试和满足特殊情况下对权限的要求，ResourceManager 允许用户直接将 ApplicationMaster 启动在客户端而不是由 ResourceManager 启动，此时仍需对其记录日志以便故障恢复时使用，正在记录日志的 RMAppAttemptImpl 所处的状态是 LAUNCHED_UNMANAGED_SAVING。
- ❑ RECOVERED：如果管理员开启了应用程序恢复功能（默认不开启，可通过参数 yarn.resourcemanager.recovery.enabled 配置），则 ResourceManager 重启后，会从日志中恢复 ApplicationAttemptImpl，恢复完成后所处的状态为 RECOVERED。

（2）基本事件

基本事件包括：

- ❑ START：应用程序的状态机 RMAppImpl 创建运行尝试 RMAppAttemptImpl 后，会第一时间向它发送一个 START 事件，接收到该事件后，RMAppAttemptImpl 会进行一些初始化工作，比如设置启动时间、获取各种安全 Token 等。
- ❑ KILL：当 ClientRMService 接收到来自客户端的杀死应用程序的命令后，它会向该应用程序的 RMAppImpl 对象发送一个 RMAppEventType.KILL 事件，而 RMAppImpl 对象则会进一步向它启动的 RMAppAttemptImpl 对象发送一个 RMAppAttemptEventType.KILL 事件，最终触发一个杀死 ApplicationMaster 所在 RMContainer 的命令，将应用程序杀死。
- ❑ APP_ACCEPTED：资源调度器对 RMAppAttemptImpl 进行各种限制性检查（比如是否超过了应用程序运行数目上限）后，如果同意提交该应用程序，则会

向 `RMAppAttemptImpl` 发送一个 `APP_ACCEPTED` 事件。一旦接收到该事件后, `RMAppAttemptImpl` 将调用 `ApplicationMasterProtocol#allocate` 向资源管理器申请资源以启动 `ApplicationMaster`。

- **CONTAINER_ALLOCATED** : 资源调度器将某个节点上的 `Container` 分配给该应用程序的 `RMAppAttemptImpl` 后, 会创建一个 `RMContainerImpl` 对象, 并向该对象发送一个 `RMContainerEventType.START` 事件。`RMContainerImpl` 收到该事件后, 会进一步向 `RMAppAttemptImpl` 发送一个 `RMAppAttemptEventType.CONTAINER_ALLOCATED` 事件。一旦接收到该事件后, `RMAppAttemptImpl` 将调用 `ApplicationMasterProtocol#allocate` 获取资源管理器分配的资源, 并向 `RMStateStore` 发送一个记录日志事件, 以将资源分配信息写到磁盘上用于故障恢复。
- **ATTEMPT_SAVED** : `RMAppAttemptImpl` 收到分配的 `Container` 后, 将 `<ApplicationAttemptId, Container, AppAttemptTokens>` 等信息存入 `RMStateStore` (`MemoryRMStateStore` 或者 `FileSystemRMStateStore`) 中, 保存成功后, `RMStateStore` 将向 `RMAppAttemptImpl` 发送一个 `ATTEMPT_SAVED` 事件。
- **LAUNCHED** : `ApplicationMasterLauncher` 与 `NodeManager` 通信, 成功启动应用程序的 `ApplicationMaster` 后, 会向 `RMAppAttemptImpl` 发送一个 `LAUNCHED` 事件。收到该事件后, `RMAppAttemptImpl` 会向 `AMLivelinessMonitor` 组件注册, 以开启对 `ApplicationMaster` 的心跳监控。
- **REGISTERED** : `ApplicationMaster` 在 `NodeManager` 启动后, 所要做的第一件事是调用 RPC 函数 `ApplicationMasterProtocol#registerApplicationMaster` 向 `ResourceManager` 注册, `ResourceManager` 中的 `ApplicationMasterService` 处理收到该 RPC 请求后, 会向 `RMAppAttemptImpl` 发送一个 `REGISTERED` 事件。
- **UNREGISTERED** : `ApplicationMaster` 运行完成后, 将调用 RPC 函数 `ApplicationMasterProtocol#finishApplicationMaster` 通知 `ResourceManager` 自己运行结束, `ResourceManager` 中的 `ApplicationMasterService` 处理收到该 RPC 请求后, 会向 `RMAppAttemptImpl` 发送一个 `UNREGISTERED` 事件。需要注意的是, 如果 `ApplicationMaster` 不是由 `ResourceManager` 启动的 (由用户程序启动, 通常启动在客户端), 则该事件会导致 `RMAppAttemptImpl` 直接进入 `FINISHED` 状态, 否则将进入 `FINISHING` 状态, 等待 `ApplicationMaster` 所在的 `Container` 退出后, 再进一步转换为 `FINISHED` 状态。
- **CONTAINER_FINISHED** : `NodeManager` 周期性调用 RPC 函数 `ResourceTracker#nodeHeartbeat` 向 `ResourceManager` 汇报心跳信息 (包括各个 `Container` 状态、完成的 `Container` 列表、节点健康状况等), 当 `ApplicationMaster` 所在 `Container` 退出运行后, `NodeManager` 将它的运行状态汇报给 `ResourceManager`, 这会导致资源调度器触发一个 `RMContainerEventType.FINISHED` 事件, 而 `RMContainerImpl` 收到该事件后, 会进一步向 `RMAppAttemptImpl` 发送一个 `RMAppAttemptEventType`。

CONTAINER_FINISHED 事件。

- ❑ EXPIRE：如果一个 RMApAttemptImpl (ApplicationMaster) 在一定时间内未汇报心跳信息，则 AMLivelinessMonitor 会向它发送一个 EXPIRE 事件，RMApAttemptImpl 收到该事件后，会进一步触发 AMLauncherEventType.CLEANUP 和 RMApAttemptEventType.ATTEMPT_FAILED 两个事件，分别用于清理 ApplicationMaster 和正在使用的 Container。
- ❑ CONTAINER_ACQUIRED：ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#allocate 获取资源管理器最近分配的 Container 后，会依次向这些 Container 发送一个 RMContainerEventType.ACQUIRED 事件，而 RMContainerImpl 收到该事件后，将进一步向 RMApAttemptImpl 发送一个 RMApAttemptEventType.CONTAINER_ACQUIRED 事件，RMApAttemptImpl 收到该事件后，会将资源所在节点保存到一个节点列表中（可用于清理 Container，比如 ResourceManager 杀死应用程序时，可依次向这些节点发送杀死 Container 的命令）。
- ❑ LAUNCH_FAILED：ApplicationMasterLauncher 与 NodeManager 通信启动 ApplicationMaster 的过程中，抛出异常，将触发一个 LAUNCH_FAILED 事件，进而引发后面一系列的资源回收操作。
- ❑ RECOVER：如果管理员启用了故障恢复机制，RMApAttemptImpl 将收到 RMApmpml 发送的 RECOVER 事件，进而尝试从日志中恢复重启前已保存的信息。
- ❑ APP_REJECTED：如果 RMApAttemptImpl 未通过资源调度器的权限检查（比如用户将应用程序提交到没有权限的队列中），将收到 APP_REJECTED 事件，进而引发一系列资源回收操作。
- ❑ STATUS_UPDATE：ApplicationMaster 需周期性地调用 RPC 函数 ApplicationMasterProtocol#allocate 向 ResourceManager 汇报自己的进度、申请资源和获取已经分配的资源，每次调用均会触发一个 STATUS_UPDATE，从而引发 RMApAttemptImpl 保存它的最新执行进度。

图 5-9 描述了各个事件的来源，这些事件来自多个组件和服务，这使得 RMApAttemptImpl 成为一个非常核心的组件。

5.6.3 RMContainer 状态机

在 YARN 中，根据应用程序需求，资源被切分成大小不同的资源块，每份资源基本信息由 Container 描述（具体将在第 6 章介绍），而具体的使用状态追踪则是由 RMContainer 完成的。RMContainer 是 ResourceManager 中用于维护一个 Container 生命周期的数据结构，它的实现是 RMContainerImpl 类，该类维护了一个 Container 状态机，记录了一个 Container 可能存在的各个状态以及导致状态间转换的事件，当某个事件发生时，RMContainerImpl 会根据实际情况进行 Container 状态转移，同时触发一个行为。

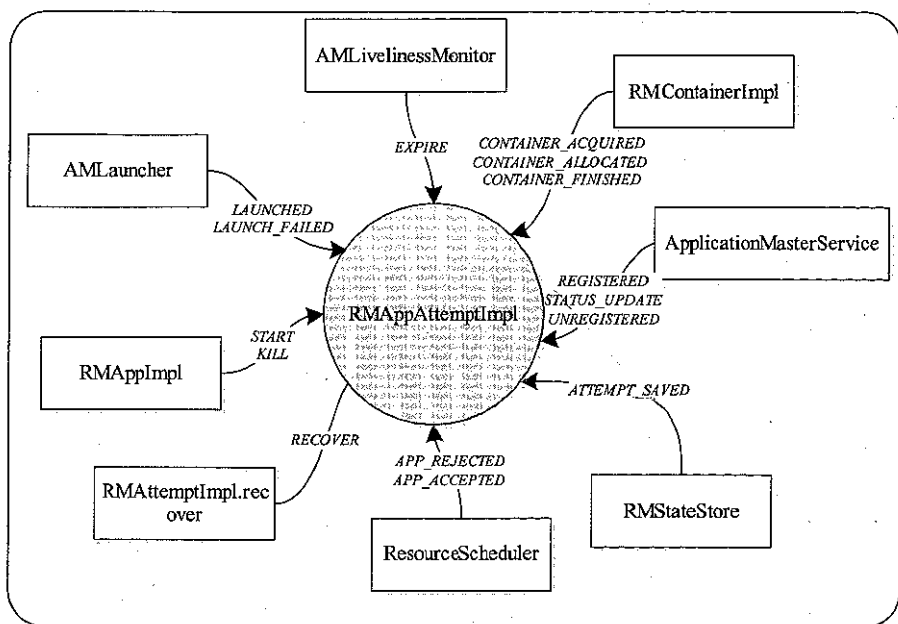


图 5-9 RMApplAttempt 状态机事件来源

如图 5-10 所示，在 RM 看来，每个 Container 有 9 种基本状态 (RMContainerState) 和 8 种导致这 9 种状态之间发生转移的事件 (RMContainerEventType)，RMContainerImpl 的作用是等待接收其他对象发出的 RMContainerEventType 类型的事件，然后根据当前状态和事件类型，将当前状态转移到另外一种状态，同时触发另外一种行为（实际上执行一个函数，该函数可能会再次发出一种其他类型的事件）。下面具体进行介绍。

(1) 基本状态

基本状态包括：

- NEW：状态机初始状态，每个 Container 对应一个状态机，而每个状态机的初始状态均为 NEW。
- RESERVED：当一个节点上的资源不能够满足一个应用程的需求但又不得不分配给它时，YARN 会让该节点为它预留资源，直到累计的空闲资源能够满足应用程序的需求后，才会将之封装成一个 Container 发送给应用程序的 ApplicationMaster。当一个 Container 已被创建，但包含的资源尚不能满足应用程序需求时所处的状态为 RESERVED。
- ALLOCATED：当资源调度器将一个 Container 分配给一个 Application 时，该 Container 处于 ALLOCATED 状态。需要注意的是，此时 Container 处于可使用状态（只在 ResourceManager 内部进行了标记），但是 ApplicationMaster 还未获取该 Container。

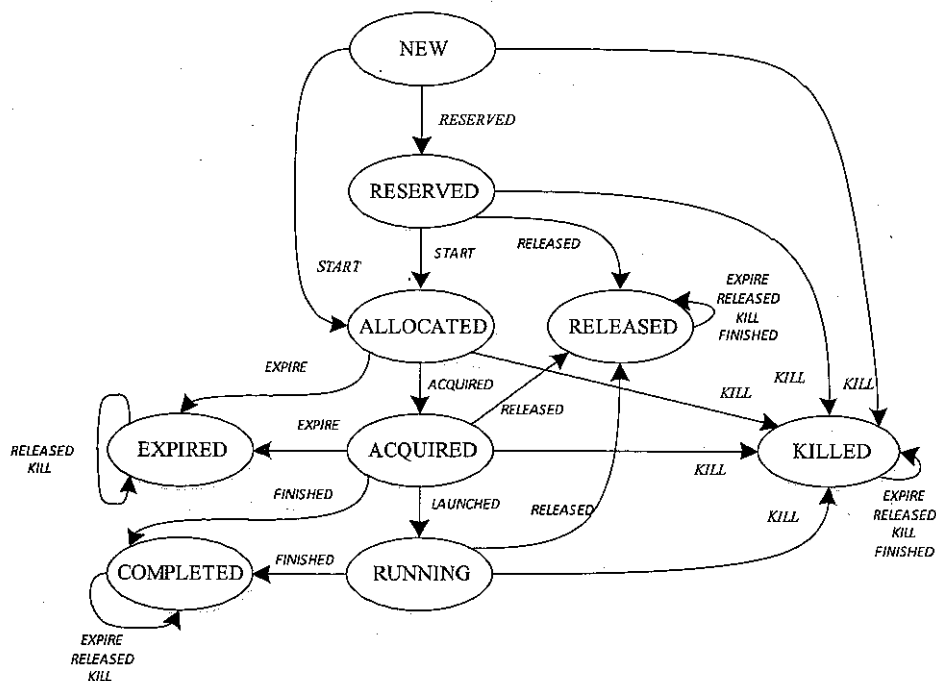


图 5-10 RMContainer 状态机

- **ACQUIRED** : ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#allocate 获取分配给自己的 Container 列表, 此时, 这些 Container 状态将被置为 ACQUIRED。
- **RUNNING** : ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#allocate 拉取分配给自己的 Container 后, 将与对应的 NodeManager 通信以启动这些 Container, 接着 NodeManager 通过心跳机制将这些 Container 状态汇报给 ResourceManager, 最终 ResourceManager 将这些 Container 状态置为 RUNNING。
- **RELEASED** : ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#allocate 向 ResourceManager 发送请求, 要求它释放一些 Container (可能是由于资源过剩或者内部的抢占机制要求释放一些 Container), ResourceManager 收到请求后将这些 Container 状态置为 RELEASED, 同时向 RMNodeImpl 发送 RMNodeEventType.CLEANUP_CONTAINER 事件以清理该 Container。
- **COMPLETED** : NodeManager 通过 RPC 函数 ResourceTracker#nodeHeartbeat 告诉 ResourceManager 已经运行完成的 Container 列表, ResourceManager 收到该信息后, 会将这些 Container 状态置为 COMPLETED。
- **EXPIRED** : ResourceManager 将一个 Container 分配给 ApplicationMaster 后, ApplicationMaster 必须在一定时间内使用该 Container (默认是 10min), 否则 ResourceManager 会强制回收该 Container, 即将 Container 状态置为 EXPIRED, 同时向 NodeManager 发送杀死 Container 的命令。

❑ **KILLED**: 当出现以下几种情况时, 将导致 Container 置为 KILLED 状态。

- 资源调度器为了保证公平性或者更高优先级的应用程序的服务质量, 不得不杀死一些应用程序占用的 Container 以满足另外一些应用程序的要求。
- 某个 NodeManager 在一定时间内未向 ResourceManager 汇报心跳信息, 则 ResourceManager 认为它死掉了, 会将它上面所有正运行的 Container 状态置为 KILLED。
- 用户 (使用 API 或者 Shell 命令) 强制杀死一个 RMAppAttemptImpl 实例时, 会导致它所有的 Container 状态置为 KILLED。

(2) 基本事件

基本事件包括:

- ❑ **START**: 资源调度器将一个 Container 分配给某个应用程序后, 会向 RMContainerImpl 发送一个 START 事件。
- ❑ **RESERVED**: 当资源管理器发现某个节点上的资源不足以满足一个应用程序资源需求时, 资源调度器会创建一个 RMContainerImpl 对象, 同时向它发送一个 RESERVED 事件。
- ❑ **ACQUIRED**: ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#allocate 拉取分配给自己的 Container 时, 资源调度器会向 RMContainerImpl 发送一个 ACQUIRED 事件, RMContainerImpl 收到该事件后, 会将 Container 添加到 ContainerAllocation-Expirer 组件的监控列表中。
- ❑ **LAUNCHED**: ApplicationMaster 通过 RPC 函数 ApplicationMasterProtocol#allocate 拉取分配给自己的 Container 后, 将与对应的 NodeManager 通信以启动这些 Container。之后, NodeManager 通过心跳机制将这些 Container 状态汇报给 ResourceManager, 而 ResourceManager 收到后会为每个 Container 发出一个 LAUNCHED 事件, RMContainerImpl 收到该事件后, 会将 Container 从 ContainerAllocationExpirer 组件的监控列表中移除。
- ❑ **FINISHED**: NodeManager 通过心跳机制告诉 ResourceManager 运行完成的 Container 列表, 而 ResourceManager 收到后会为每个 RMContainerImpl 发出一个 FINISHED 事件。
- ❑ **RELEASED**: 当资源请求过剩或者为了满足一些优先级更高的任务而释放另外一些任务占用的 Container 时, ApplicationMaster 会通过 RPC 函数 ApplicationMasterProtocol#allocate 告诉 ResourceManager 待释放 Container 列表, 而 ResourceManager 会向这些 container 发出一个 RELEASED 事件。
- ❑ **KILL**: 当发生资源抢占、应用程序被杀死或者节点丢失时, 会向相关的 Container 发送一个 KILL 事件以清理它们占用的资源。
- ❑ **EXPIRE**: ResourceManager 将一个 Container 分配给 ApplicationMaster 后, 如果 ApplicationMaster 在一定时间内没有使用该 Container, 则 ResourceManager 会为该 container 发出一个 EXPIRE 事件, 以对其进行回收。

图 5-11 描述了以上各个事件的来源。

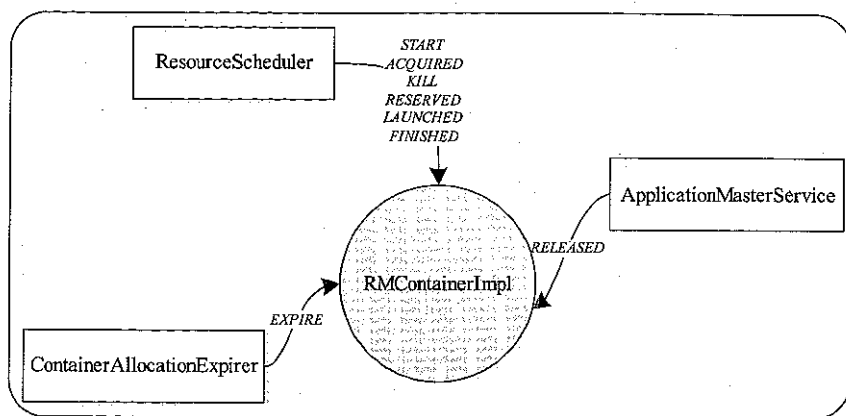


图 5-11 RMContainer 状态机事件来源

5.6.4 RMNode 状态机

RMNode 是 ResourceManager 中用于维护一个节点生命周期的数据结构，它的实现是 RMNodeImpl 类，该类维护了一个节点状态机，记录了节点可能存在的各个状态以及导致状态间转换的事件。当某个事件发生时，RMNodeImpl 会根据实际情况进行节点状态转移，同时触发一个行为。

如图 5-12 所示，在 RM 看来，每个节点有 6 种基本状态 (NodeState) 和 8 种导致这 6 种状态之间发生转移的事件 (RMNodeEventType)，RMNodeImpl 的作用是等待接收其他对象发出的 RMNodeEventType 类型的事件，然后根据当前状态和事件类型，将当前状态转移到另外一种状态，同时触发另外一种行为（实际上执行一个函数，该函数可能会再次发出一种其他类型的事件）。

下面具体进行介绍。

(1) 基本状态

基本状态包括：

- NEW：状态机初始状态，每个 NodeManager 对应一个状态机，而每个状态机的初始状态均为 NEW。
- RUNNING：NodeManager 启动后，会通过 RPC 函数 ResourceTracker#registerNodeManager 向 ResourceManager 注册，此时 NodeManager 会进入 RUNNING 状态。
- DECOMMISSIONED：如果一个节点被加入到 exclude list（黑名单）中，则对应的 NodeManager 将被置为 DECOMMISSIONED 状态，这样，该 NodeManager 将无法与 ResourceManager 通信（直接在 RPC 层抛出异常导致 NodeManager 异常退出）。

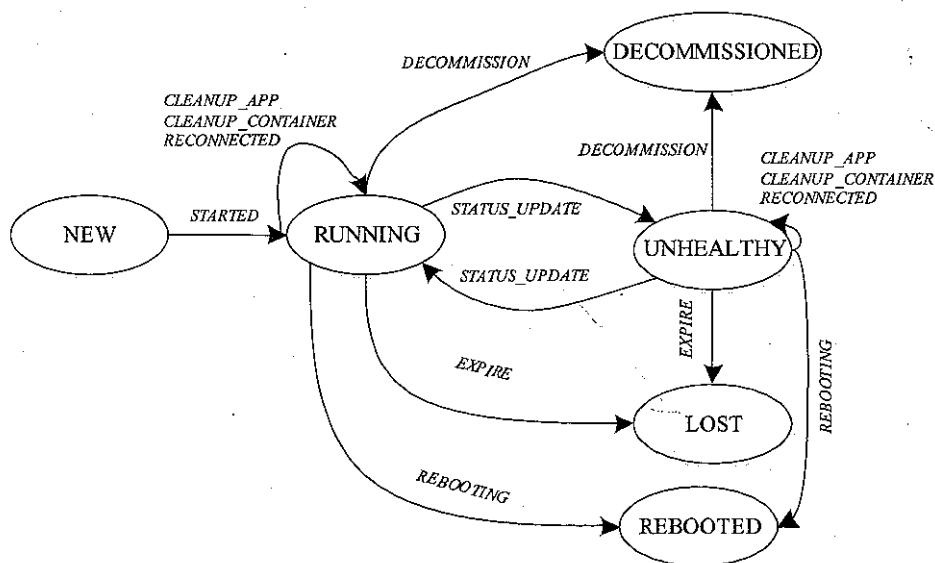


图 5-12 RMNode 状态机

- **UNHEALTHY**：管理员可在每个 NodeManager 上配置一个健康状况监测脚本，NodeManager 中有一个专门线程周期性执行该脚本，以判定 NodeManager 是否处于健康状态。NodeManager 会通过心跳机制将脚本执行结果汇报给 Resource-Manager，如果 ResourceManager 发现它处于不健康状态下，则会将其状态置为 UNHEALTHY，此后 ResourceManager 不会再为该节点分配新的任务，直到它重新变为健康状态。关于健康状况监测脚本的介绍，可阅读第 7 章。
- **LOST**：ResourceManager 中的组件 NMLivelinessMonitor 会跟踪每一个 NodeManager 的心跳信息，如果一个 NodeManager 在一定时间间隔内未汇报心跳信息，则认为它死掉了，RMNodeImpl 会将其置为 LOST 状态，之后它上面所有正运行的 Container 信息将被置为 FAILED。
- **REBOOTED**：如果 ResourceManager 发现 NodeManager 汇报的心跳 ID 与自己保存的不一致，则会将其置为 REBOOTED 状态，从而要求它重新启动以达到同步的目的。

(2) 基本事件

基本事件包括：

- **STARTED**：NodeManager 启动后，会通过 RPC 函数 ResourceTracker#registerNodeManager 向 RM 注册，此时会触发 STARTED 事件。
- **STATUS_UPDATE**：NodeManager 会通过 RPC 函数 ResourceTracker#nodeHeartbeat 周期性向 RM 汇报心跳信息，而每次汇报心跳均会触发一个 STATUS_UPDATE 事件。
- **DECOMMISSION**：当一个节点被加入 exclude list 中后，它上面的 NodeManager 尝

试通过 RPC 函数 ResourceTracker#nodeHeartbeat 与 NodeManager 通信时, 会触发一个 DECOMMISSION 事件。

- ❑ EXPIRE : ResourceManager 中的组件 NMLivelinessMonitor 会跟踪每一个 NodeManager 的心跳信息, 如果一个 NodeManager 在一定时间间隔内未汇报心跳, NMLivelinessMonitor 会触发一个 EXPIRE 事件。
- ❑ REBOOTING : 当 ResourceManager 发现 NodeManager 汇报的心跳 ID 与自己保存的不一致时, 会触发一个 REBOOTING 事件。
- ❑ CLEANUP_APP : 当一个应用程序执行完成时 (可能成功或则失败), 会触发一个 CLEANUP_APP 事件, 以清理应用程序占用的资源。
- ❑ CLEANUP_CONTAINER : 当一个 Container 执行完成时 (可能成功或则失败), 会触发一个 CLEANUP_CONTAINER 事件, 以清理 Container 占用的资源。
- ❑ RECONNECTED : 如果一个已经在 ResourceManager 上注册过的 NodeManager 再次请求注册, ResourceManager 会触发一个 RECONNECTED 事件, 而 RMNodeImpl 收到该事件后将更新 NodeManager 信息为新注册汇报的信息。

图 5-13 描述了以上各个事件的来源。

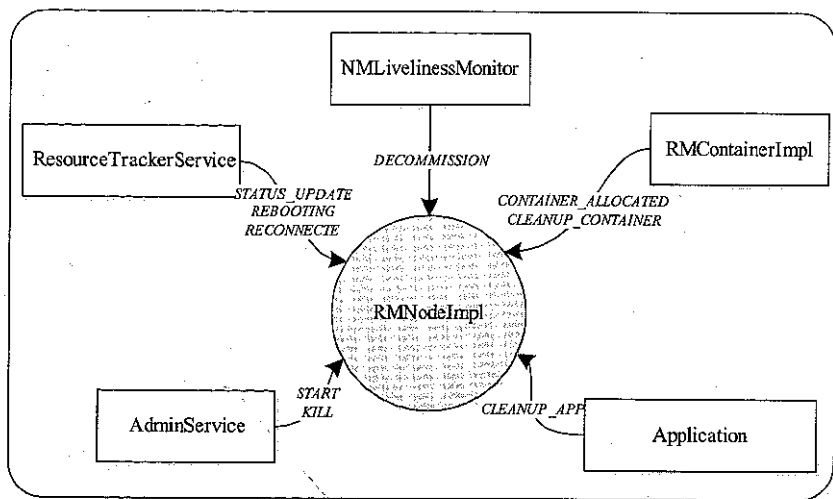


图 5-13 RMNode 状态机事件来源

5.7 几个常见行为分析

5.7.1 启动 ApplicationMaster

本小节介绍从应用程序提交到启动 ApplicationMaster 的整个过程, 期间涉及 Client-RMService、RMAppManager、RMAppImpl、RMAppAttemptImpl、RMNode、ResourceScheduler

等几个主要组件。当客户端调用 RPC 函数 `ApplicationClientProtocol#submitApplication` 后，ResourceManager 端的处理过程（假设整个过程未出现任何异常）如图 5-14 所示。

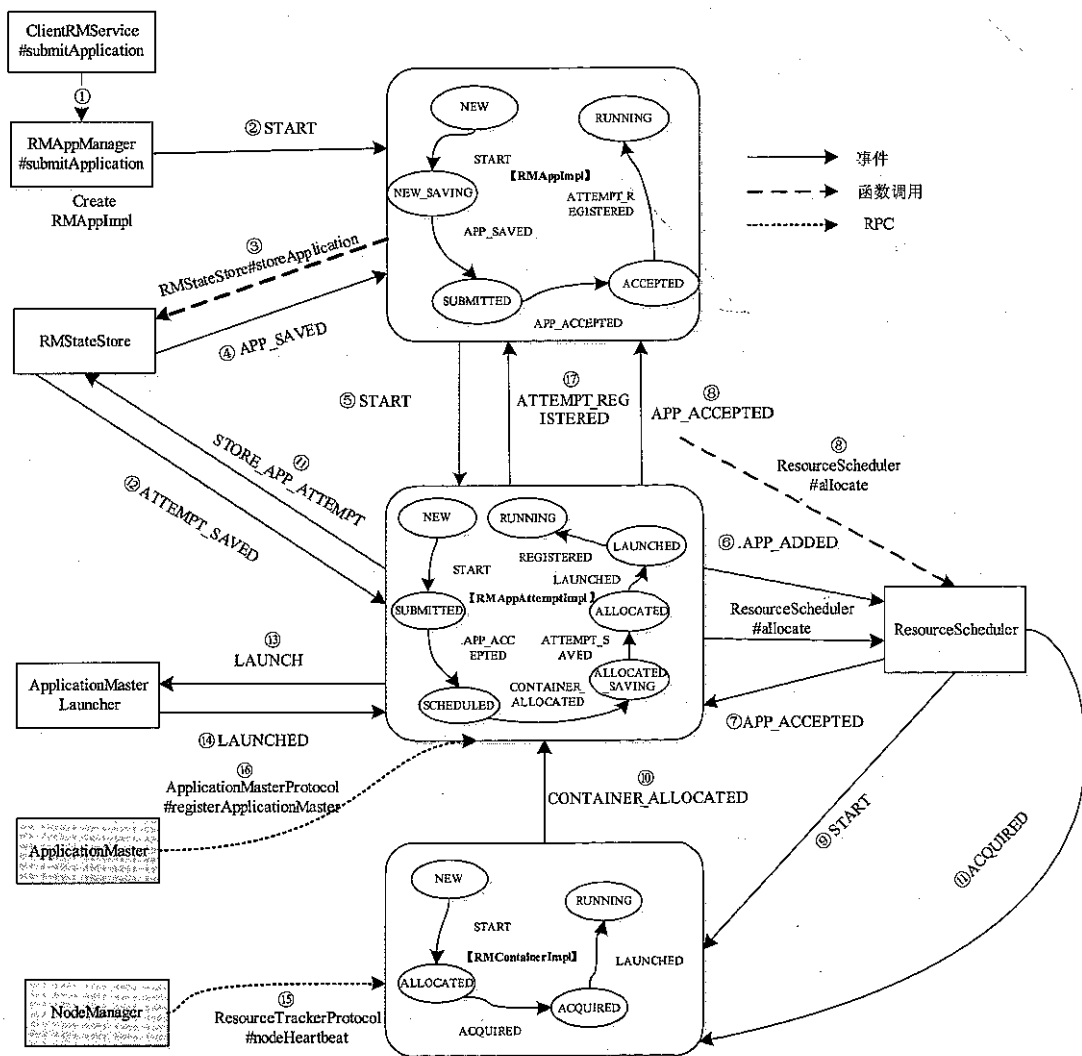


图 5-14 ApplicationMaster 启动流程

具体步骤如下。

步骤 1 ResourceManager 中的 `ClientRMService` 实现了 `ApplicationClientProtocol` 协议，它处理来自客户端的请求，并调用 `RMAppManager#submitApplication` 通知其他相关服务作进一步处理。

步骤 2 `RMAppManager` 为该应用程序创建一个 `RMAppImpl` 对象以维护它的运行状态，并判断系统状态，如果是故障重启状态，则向它发送一个 `RMAppEventType.RECOVER` 事件，否则发送一个 `RMAppEventType.START` 事件。

步骤3 RMApplImpl 收到 RMApplEventType.START 事件后（暂不介绍 RMApplEventType.RECOVER 事件的处理过程），会调用 RMStateStore#storeApplication（RMStateStore 是插拔式组件，在不启用 RM 恢复机制的前提下，默认实现是 NullRMStateStore，它不会进行任何保存工作，其他实现还有 MemoryRMStateStore、FileSystemRMStateStore 等），以日志记录 RMApplImpl 当前信息，至此，RMApplImpl 的运行状态由 NEW 转移为 NEW_SAVING。

步骤4 日志记录完成后，RMStateStore 进一步向 RMApplImpl 发送 RMApplEventType.APP_SAVED 事件。

步骤5 RMApplImpl 收到 RMApplEventType.APP_SAVED 事件后，将创建一个运行实例对象 RMApplAttemptImpl，同时向它发送一个 RMApplAttemptEventType.START 事件，至此，RMApplImpl 的运行状态由 NEW_SAVING 转移为 SUBMITTED。

步骤6 RMApplAttemptImpl 收到 RMApplAttemptEventType.START 事件后，进行一些必要的初始化工作（设置初始事件，各种安全 Token 等），然后向 ResourceScheduler 发送 SchedulerEventtype.APP_ADDED 事件，至此，RMApplAttemptImpl 状态由 NEW 转移为 SUBMITTED。

步骤7 ResourceScheduler 收到 SchedulerEventtype.APP_ADDED 事件后，首先进行一些权限检查（如果通不过这些检查，则拒绝接受应用程序提交），然后将应用程序信息保存到内部的数据结构中，并向 RMApplAttemptImpl 发送 RMApplAttemptEventType.APP_ACCEPTED 事件。

步骤8 RMApplAttemptImpl 收到 RMApplAttemptEventType.APP_ACCEPTED 事件后，首先向 RMApplImpl 发送一个 RMApplEventType.APP_ACCEPTED 事件（RMApplImpl 收到该事件后直接将状态从 SUBMITTED 转移为 ACCEPTED），然后调用 ResourceScheduler#allocate 为应用程序的 ApplicationMaster 申请资源，该资源描述如下：

```
<AM_CONTAINER_PRIORITY, ResourceRequest.ANY, appAttempt.getSubmissionContext().
getResource(), 1 >
```

即一个优先级为 AM_CONTAINER_PRIORITY（值为 0）、可在任意节点（ResourceRequest.ANY）上、资源量为 X（用户提交应用程序时指定）的 Container。

至此，RMApplAttemptImpl 状态由 SUBMITTED 转移为 SCHEDULED。

步骤9 ResourceManager 为应用程序的 ApplicationMaster 分配资源后，创建一个 RMContainerImpl，并向它发送一个 RMContainerEventType.START 事件。

步骤10 RMContainerImpl 收到 RMContainerEventType.START 事件后，直接向 RMApplAttemptImpl 发送一个 RMApplAttemptEventType.CONTAINER_ALLOCATED 事件，至此，RMContainerImpl 状态从 NEW 转移为 ALLOCATED。

步骤11 RMApplAttemptImpl 收到 RMApplAttemptEventType.CONTAINER_ALLOCATED 事件后，调用 ResourceScheduler#allocate 获取分配的资源，ResourceScheduler 将资源返回给它之前，会向 RMContainerImpl 发送一个 RMContainerEventType.ACQUIRED 事件（RMContainerImpl 收到该事件后，会向 ContainerAllocationExpirer 注册以启动监控，之后

向 `RMAppAttemptImpl` 发送 `RMAppAttemptEventType.CONTAINER_ACQUIRED` 事件, 它收到该事件后没有后续处理工作), 而 `RMAppAttemptImpl` 收到资源后, 第一时间向 `RMStateStore` 发送 `MStateStoreEventType.STORE_APP_ATTEMPT` 事件请求记录日志, 至此, `RMAppAttemptImpl` 状态从 `SCHEDULED` 转移为 `ALLOCATED_SAVING`。

步骤 12 日志记录完成后, `RMStateStore` 进一步向 `RMAppAttemptImpl` 发送 `RMAppAttemptEventType.ATTEMPT_SAVED` 事件。

步骤 13 `RMAppAttemptImpl` 收到 `RMAppAttemptEventType.ATTEMPT_SAVED` 事件后, 将向 `ApplicationMasterLauncher` 发送 `AMLancherEventType.LAUNCH` 事件, 至此, `RMAppAttemptImpl` 状态从 `ALLOCATED_SAVING` 转移为 `ALLOCATED`。

步骤 14 `ApplicationMasterLauncher` 收到 `AMLancherEventType.LAUNCH` 事件后, 会将该事件放到事件队列中, 等待 `AMLancher` 线程池中的线程处理该事件。处理方法是, 与对应的 `NodeManager` 通信, 启动 `ApplicationMaster`, 一旦成功启动后, 将进一步向 `RMAppAttemptImpl` 发送 `RMAppAttemptEventType.LAUNCHED` 事件。`RMAppAttemptImpl` 收到 `RMAppAttemptEventType.LAUNCHED` 事件后, 会向 `AMLivelinessMonitor` 注册, 以监控运行状态。至此, `RMAppAttemptImpl` 状态从 `ALLOCATED` 转移为 `LAUNCHED`。

步骤 15 `NodeManager` 通过心跳机制汇报 `ApplicationMaster` 所在 `Container` 已经成功启动, 收到该信息后, `ResourceScheduler` 将发送一个 `RMContainerEventType.LAUNCHED` 事件, `RMContainerImpl` 收到该事件后, 会从 `ContainerAllocationExpirer` 监控列表中移除。

步骤 16 启动的 `ApplicationMaster` 通过 RPC 函数 `ApplicationMasterProtocol#registerApplicationMaster` 向 `ResourceManager` 注册, `ResourceManager` 中的 `ApplicationMasterService` 服务接收到该请求后, 将向 `RMAppAttemptImpl` 发送一个 `RMAppAttemptEventType.REGISTERED` 事件, 而 `RMAppAttemptImpl` 收到该事件后, 首先保存该 `ApplicationMaster` 的基本信息 (比如所在 `host`、启用的 RPC 端口号等), 然后向 `RMAppImpl` 发送一个 `RMAppEventType.ATTEMPT_REGISTERED` 事件。至此, `RMAppAttemptImpl` 状态从 `LAUNCHED` 转移为 `RUNNING`。

步骤 17 `RMAppImpl` 收到 `RMAppEventType.ATTEMPT_REGISTERED` 事件后, 所做的事情仅是将状态从 `ACCEPTED` 转换为 `RUNNING`。

5.7.2 申请与分配 Container

本小节介绍应用程序的 `ApplicationMaster` 在 `NodeManager` 上成功启动并向 `ResourceManager` 注册后, 向 `ResourceManager` 请求资源 (`Container`) 到获取到资源的整个过程中, 及 `ResourceManager` 内部涉及的主要工作流程。整个过程可看做以下两个阶段的迭代循环:

阶段 1 `ApplicationMaster` 汇报资源需求并领取已经分配到的资源;

阶段 2 `NodeManager` 向 `ResourceManager` 汇报各个 `Container` 运行状态, 如果 `ResourceManager` 发现它上面有空闲的资源, 则进行一次资源分配, 并将分配的资源保存到对应的应用程序数据结构中, 等待下次 `ApplicationMaster` 发送心跳信息时获取 (即阶段 1)。

型事件，而 `RMNodeImpl` 收到该事件后，将更新各个 `Container` 的运行状态，并进一步向 `ResourceScheduler` 发送一个 `SchedulerEventType.NODE_UPDATE` 类型事件。

步骤 3 `ResourceScheduler` 收到事件后，如果该节点上有可分配的空闲资源，则会将这些资源分配给各个应用程序，而分配后的资源仅是记录到对应的数据结构中，等待 `ApplicationMaster` 下次通过心跳机制来领取。

5.7.3 杀死 Application

“杀死 Application”行为通常是由客户端发起的，比如用户使用命令“`bin/yarn application -kill XXX`”杀死一个已经提交的应用程序。`ResourceScheduler` 中的服务 `ClientRMService` 负责处理该请求，通过权限检查后（确保该用户有权限杀死该应用程序），它会向该应用程序状态对应的（`RMAppImpl` 类型）状态维护对象发送一个 `RMAppEventType.KILL` 类型的事件，`RMAppImpl` 收到该事件后，根据当前运行状态调用相应的行为函数，这些函数的主要工作是清理该应用程序已经运行完成（但运行失败）或者正在运行的实例（`RMAppAttemptImpl`）。总体上讲，这一行为主要分为两种情况：

情况 1 不存在正在运行的 `RMAppAttemptImpl`。

当应用程序不存在处于运行状态的运行实例 `RMAppAttemptImpl` 时，整个过程比较简单，如图 5-16 所示，`RMAppImpl` 向前几个运行实例（如果有的话，这些实例通常已经运行失败）所在的节点状态维护对象 `RMNodeImpl` 发送 `RMNodeEventType.CLEANUP_APP` 事件，以记录节点曾运行过的应用程序；同时，也会向 `RMApplManager` 发送 `RMApplManagerEventType.APP_COMPLETED` 事件，以标注该应用程序运行状态并移除保存在 `RMStateStore` 中的日志信息。

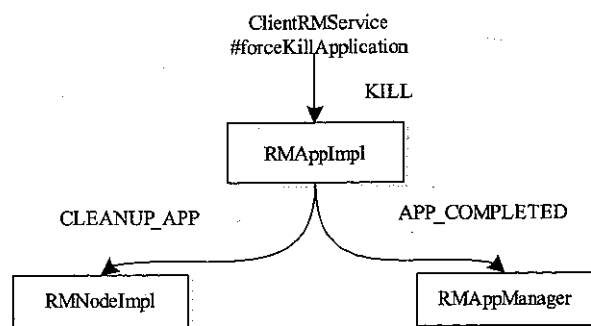


图 5-16 不存在正在运行的 `RMAppAttemptImpl` 的情况

情况 2 存在正在运行的 `RMAppAttemptImpl`。

如图 5-17 所示，当应用程序存在正在运行的 `RMAppAttemptImpl` 时，除了完成情况 1 描述的步骤外，`RMAppImpl` 还要向 `RMAppAttemptImpl` 发送 `RMAppAttemptEventType.KILL`，以回收 `RMAppAttemptImpl` 已经申请和占用的资源。而 `RMAppAttemptImpl` 收到该事件后，将在第一时间向 `RMAppImpl` 发送 `RMAppEventType.ATTEMPT_KILLED` 作为

(对 RMApAttemptEventType.KILL 事件的) 应答, 而真正的资源回收操作则由资源调度器 ResourceScheduler 异步完成的。

- ❑ 回收 ApplicationMaster 占用资源: 向 ApplicationMasterLauncher 发送 AMLauncherEventType.CLEANUP 类型的事件, 以杀死 ApplicationMaster 并回收它占用的资源 (如果 ApplicationMaster 尚未启动则跳过)。
- ❑ 回收 Container 资源: 向各个已经启动的 RMContainerImpl (如果没有则跳过) 发送 RMContainerEventType.KILL 类型的事件, 以杀死已经启动的 Container 并回收它们占用的资源。

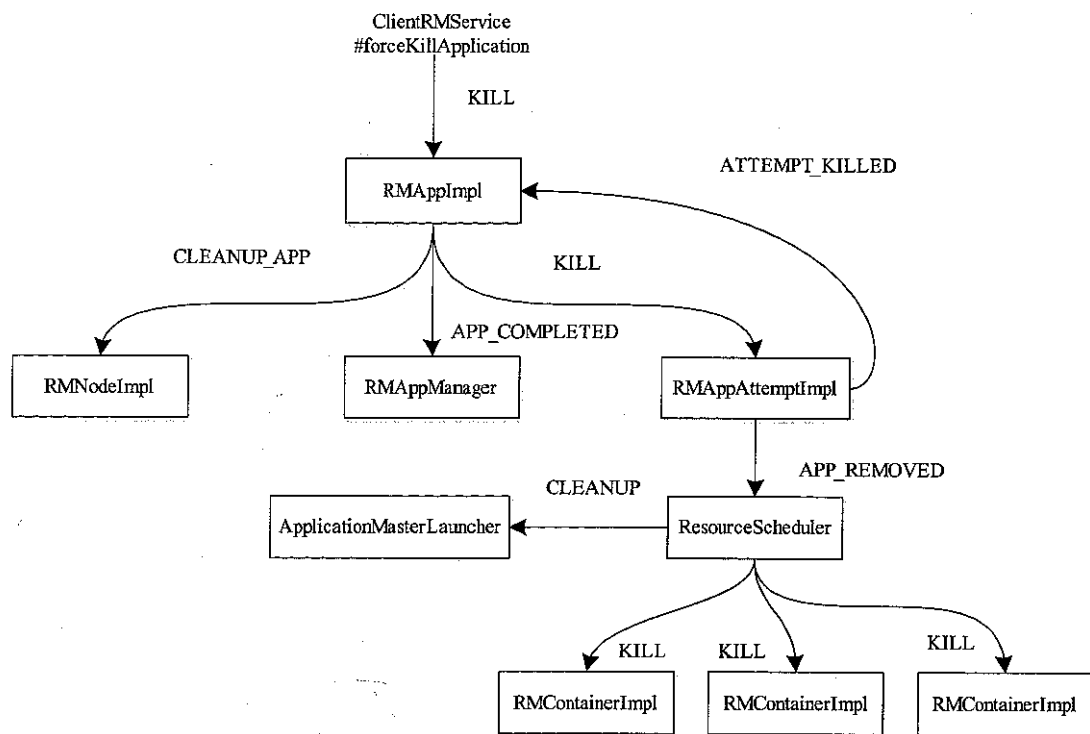


图 5-17 存在正在运行的 RMApAttemptImpl 的情况

5.7.4 Container 超时

在 YARN 中, 存在两种类型的 Container, 分别是用于运行 ApplicationMaster 的 Container (后面简称为 “AM Container”) 和运行普通任务的 Container (后面简称为 “普通 Container”), 第一种 Container 的超时将导致整个 Application 运行失败, 而第二种 Container 超时则会触发一次资源回收。需要注意的是, 第二种 Container 超时导致任务运行失败后, YARN 不会主动将其调度到另外一个节点上运行, 而是将状态告诉应用程序的 ApplicationMaster, 由它决定是否重新申请资源或者重新执行。

(1) AM Container 超时

AM Container 是由 RMApAttemptImpl 根据用户设置的资源需求向 ResourceScheduler 申请的, 一旦申请到满足要求的 Container 后, 它会将之 (RMContainerImpl) 放到 ContainerAllocationExpirer 组件中, 以确保对应的 NodeManager 能够在一定时间内启动它。如果 NodeManager 在一定时间内没有启动 ApplicationMaster (截止 ApplicationMaster 向 ResourceManager 注册), 则会触发一系列资源回收流程, 如图 5-18 所示, 具体如下。

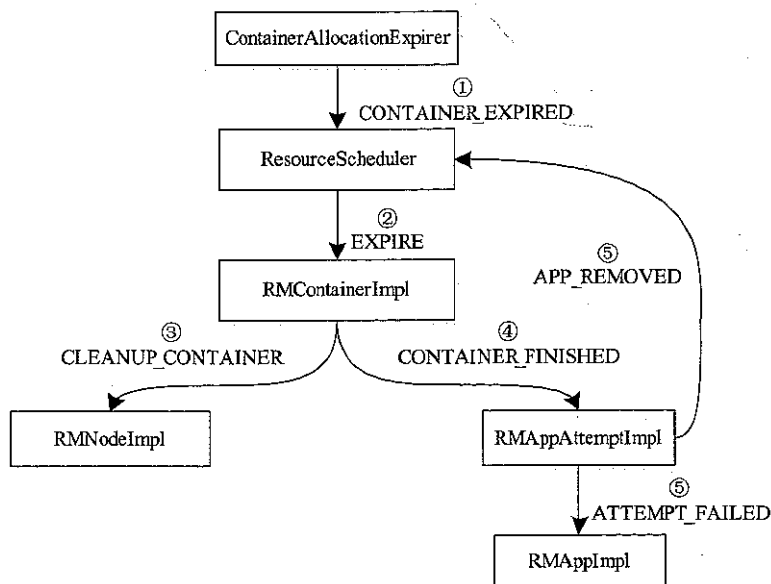


图 5-18 AM Container 超时后资源回收流程

步骤 1 该 NodeManager 没能在一定时间内 (默认是 10 min) 启动应用程序的 ApplicationMaster, 导致 ContainerAllocationExpirer 触发一个 SchedulerEventType.CONTAINER_EXPIRED 类型的事件, 而 ResourceScheduler 收到该事件后, 将进一步向 RMContainerImpl 发送一个 RMContainerEventType.EXPIRE 类型事件。

步骤 2 RMContainerImpl 收到该事件后 (正常情况下, 它处于 RMContainerState.ACQUIRED 状态中), 首先从 ContainerAllocationExpirer 中移除监控, 然后向 AM Container 所在节点的 RMNodeImpl 发送 RMNodeEventType.CLEANUP_CONTAINER 事件, 向 RMApAttemptImpl 发送 RMApAttemptEventType.CONTAINER_FINISHED 事件。

步骤 3 RMNodeImpl 收到 RMNodeEventType.CLEANUP_CONTAINER 事件后, 将之放入待清理 Container 列表中, 等到对应的 NodeManager 汇报心跳时, 将该 Container 返回给它以对其进行清理。

步骤 4 RMApAttemptImpl 收到 RMApAttemptEventType.CONTAINER_FINISHED 事件后 (正常情况下, 它处于 RMApAttemptState.LAUNCHED 状态中), 它将进一步向

RMApplImpl 发送 RMAppEventType.ATTEMPT_FAILED 事件, 向 ResourceScheduler 发送 SchedulerEventType.APP_REMOVED 事件。

步骤 5 RMApplImpl 收到事件后, 如果未超过用户设置的运行次数上限, 将尝试启动一个新的 RMAppAttemptImpl 或者 (否则) 直接宣布该应用程序运行失败; ResourceScheduler 收到事件后, 会清理该应用程序相关信息。

(2) 普通 Container 超时

相对于 AM Container 超时而言, 普通 Container 超时产生的资源回收流程则简单一些。普通 Container 超时发生在普通 Container 被分配给一个 ApplicationMaster 后, 没能够在一定时间内运行起来 (Container 运行成功后, NodeManager 会通过心跳将它的状态汇报给 ResourceManager), 也就是说, ApplicationMaster 没能够在一定时间内使用 Container (原因很多, 比如 ApplicationMaster 得到资源后一直在等待机会使用但未使用, 对应的 NodeManager 收到 Container 后挂掉了)。

如图 5-19 所示, 普通 Container 超时触发的资源回收流程跟 AM Container 的回收流程的前三个步骤是一样的, 不同的是后几个流程: RMAppAttemptImpl 收到 RMAppAttemptEventType.CONTAINER_FINISHED 事件后 (正常情况下, 它处于 RMAppAttemptState.RUNNING 状态中), 将该 Container 保存到已完成列表 (该列表中的 Container 可能处于 COMPLETED、KILLED、RELEASED 或 EXPIRED 四种状态之一) 中, 等到 ApplicationMaster 下次汇报心跳时, 将该列表返回给它, 至于如何处理这些失败的 Container 中的任务, 是重新申请资源运行该任务还是舍弃该任务, 完全由 ApplicationMaster 内部的策略决定。

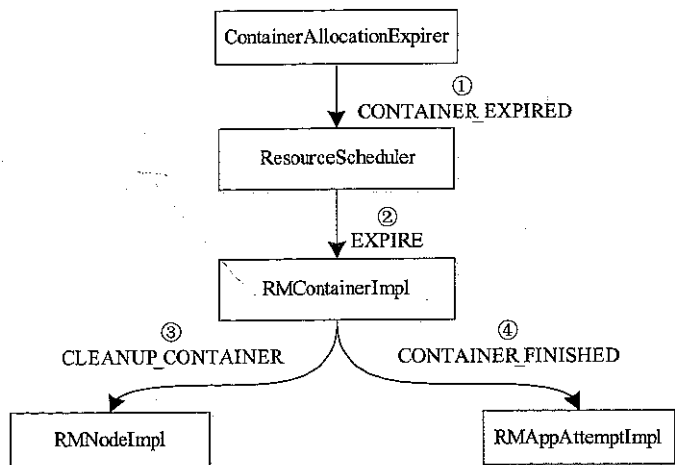


图 5-19 普通 Container 超时后资源回收流程

5.7.5 ApplicationMaster 超时

ApplicationMaster 向 ResourceManager 注册后, 必须周期性通过 RPC 函数 `ApplicationMasterProtocol#allocate` 向 ResourceManager 汇报心跳以表明自己还活着。如果一段时间 (默认是 10 min) 内它未汇报心跳, 则 ResourceManager 宣布它死亡, 进而导致 (如果重试次数未超过用户设置的运行上限值) 应用程序重新运行或者 (超过运行次数上限) 直接退出。ApplicationMaster 超时是由监控组件 `NMLivenessMonitor` 发现并触发的 (`RMAppAttemptEventType.EXPIRE` 事件), 之后整个资源回收过程与 5.7.4 节介绍的 AM Container 超时的过程基本一致, 在此不再详细介绍。

5.7.6 NodeManager 超时

NodeManager 启动后将通过 RPC 函数 `ResourceTracker#registerNodeManager` 向 ResourceManager 注册, 之后它将被加入到 `NMLivenessMonitor` 中进行监控。它必须周期性地通过 RPC 函数 `ResourceTracker#nodeHeartbeat` 向 ResourceManager 汇报心跳以表明自己还活着, 如果一段时间 (默认是 10 min) 内它未汇报心跳, 则 ResourceManager 宣布它死亡, 所以正运行在它上面的 Container 将被回收, 如图 5-20 所示。

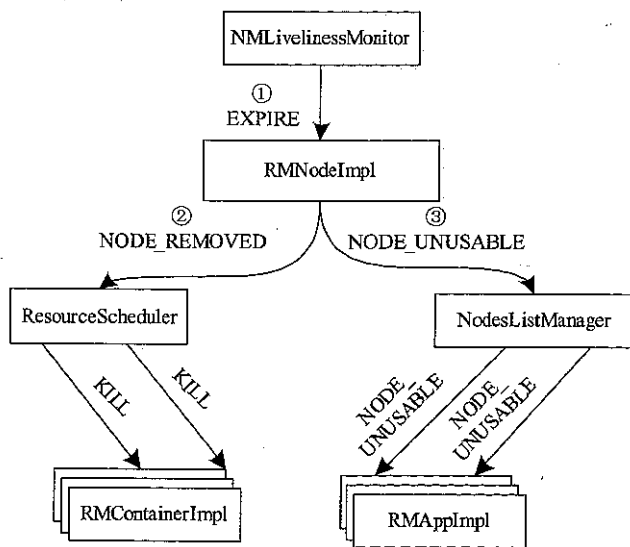


图 5-20 NodeManager 超时后资源回收流程

上述过程具体如下:

步骤 1 `NMLivenessMonitor` 发现 NodeManager 在一段时间内未汇报心跳, 则认为它死掉了, 会触发一个 `RMNodeEventType.EXPIRE` 类型的事件。

步骤 2 `RMNodeImpl` 收到该事件后, 分别向 `ResourceScheduler` 和 `NodesListManager` 发送一个 `SchedulerEventType.NODE_REMOVED` 事件和 `NodesListManagerEventType.NODE_`

UNUSABLE 事件, 同时将节点状态由 RUNNING 转为 LOST。

步骤3 ResourceScheduler 收到 NODE_REMOVED 事件后, 会向运行在死亡节点上的 RMContainerImpl 发送 RMContainerEventType.KILL 事件, 以清理占用的内存空间; NodesListManager 收到 NODE_UNUSABLE 事件后, 会向所有当前正在运行的 RMAppImpl 发送 RMAppNodeUpdateType.NODE_UNUSABLE 事件, 截至本书出版时, RMAppImpl 收到该事件后尚未有任何处理。

5.8 安全管理

由于所有的 Hadoop 集群都部署在有防火墙保护的局域网中且只允许公司内部人员访问, 因此为 Hadoop 添加安全机制的动机并不像传统的安全概念那样是为了防御外部黑客的攻击, 而是为了更好地让多用户在共享 Hadoop 集群环境下安全高效地使用集群资源。

一般而言, 系统安全机制由认证 (authentication) 和授权 (authorization) 两大部分构成。认证就是简单地对一个实体的身份进行判断; 而授权则是向实体授予对数据资源和信息访问权限的决策过程。同 Hadoop 1.0 一样, Hadoop 2.0 中的认证机制采用 Kerberos^①和 Token 两种方案, 而授权则是通过引入访问控制列表 (Access Control List, ACL) 实现的, 接下来分别对它们进行介绍。

5.8.1 术语介绍

本小节先介绍几个与安全管理相关的术语。

- Kerberos: Kerberos 是一种基于第三方服务的认证协议, 其特点是用户只需输入一次身份验证信息就可以凭借此验证获得的票据访问多个服务。Kerberos 认证过程的实现不依赖于主机操作系统的认证, 它不基于主机地址的信任, 也不要求网络上所有主机是物理安全的。Kerberos 是一种非常安全的认证协议。
- Token: Token 是一种基于共享密钥的双方身份认证机制, 它会被加入到当前的 UGI (UserGroupInformation) 对象中, 并以 Credential 对象的形式加入到 JAAS (Java Authentication and Authorization Service, Java 认证和授权服务) Subject 中^②, 当在 UGI.doAS 上下文中执行 RPC 函数时, Subject 信息将被推送到线程上下文中。
- Principal: Hadoop 集群中被认证或授权的主体, 主要包括用户、Hadoop 服务、Container、Application、Localizer、Shuffle Data 等。

5.8.2 Hadoop 认证机制

Hadoop 认证机制的实现同时采用了 Kerberos 和 Token 两种技术, 其中 Kerberos 用于用户与服务和服务与服务之间的认证, 它是一种基于可信任的第三方服务的认证机制, 在

^① 参见网址 <http://web.mit.edu/kerberos/>。

^② 此处涉及 JDK 中提供的安全库, 有兴趣的读者可自行了解。

高并发情况下，效率较低。为了解决该问题，Kerberos 一旦在客户端（可以是用户或者另一个服务）和服务器之间建立一条安全的网络连接后，客户端便可通过该连接从服务端获取一个密钥。由于该密钥仅有客户端和服务端知道，因此，接下来客户端可使用该共享密钥获取服务的认证，即基于授权令牌（Delegation Token）的认证机制。

在 Hadoop 中，Client 与 NameNode 及 Client 与 ResourceManager 之间初次通信均采用了 Kerberos 进行身份认证，之后便换用 Delegation Token 以较小开销。而 DataNode 与 NameNode 及 NodeManager 与 ResourceManager 之间的认证始终采用 Kerberos 机制。默认情况下，Kerberos 认证机制是关闭的，管理员可通过将参数 `hadoop.security.authentication` 设为“kerberos”（默认值为“simple”）启动它。接下来重点分析 Hadoop 中 Token 的工作原理以及实现。

Hadoop 中 Token 的定义在 `org.apache.hadoop.security.token.Token` 中，每类 Token 存在一个唯一 `TokenIdentifier` 标识。Token 主要由表 5-2 列出的几个字段组成。

表 5-2 Token 基本构成

字段名称	字段类型	含 义
identifier	byte[]	TokenIdentifier 中经序列化的 identity 信息
password	byte[]	TokenIdentifier 中经序列化的 password 信息
kind	Text	TokenIdentifier 种类
service	Text	TokenIdentifier 应用到的服务类型
renewer	TokenRenewer	由 ServiceLoader 为这类 TokenIdentifier 加载的插件

当一个客户端或者服务（统称“客户端”）使用令牌向另外一个服务（统称为“服务器端”）获取认证时，经过的步骤如下：

步骤 1 客户端将 `TokenIdentifier` 发送给服务器端。注意，不同类型的 Token，它们的 `TokenIdentifier` 包含的字段是不一样的，但一定可以表示该 Token 的唯一性。

步骤 2 服务端使用 `TokenIdentifier` 和 `masterKey`（`masterKey` 可称为“密钥”，它是客户端经过 Kerberos 验证后获取的，或者客户端向服务端注册时领取，并通过周期性地心跳获取最新的 `masterKey`），重新计算 `TokenAuthenticator` 和 `Token`。其中 `TokenAuthenticator` 计算方法如下：

```
TokenAuthenticator=HMAC-SHA1(masterKey, TokenIdentifier)
```

步骤 3 服务器端检查新的 Token 是否合法。一个 Token 是合法的，当且仅当 Token 在内存中存在，且当前时间仍在有效期内。

步骤 4 如果 Token 是合法的，客户端服务器端分别将 `TokenAuthenticator` 作为密钥、DIGEST-MD5 作为认证协议进行双方认证。

下面重点介绍 YARN 中的各类 Token 及其作用，有兴趣的读者可自行了解 HDFS 中 Token 的种类及其作用。

□ **ResourceManager Delegation Token**。ResourceManager Delegation Token 是

ResourceManager 授权令牌, 持有该令牌的应用程序及其发起的任务可以安全地与 ResourceManager 交互, 比如持有该令牌的 MapReduce 作业可以在 Task 中再次向 ResourceManager 提交一个或者多个作业, 进而形成一个 MapReduce 工作流, Hadoop 生态系统中的工作流引擎 Oozie 正是采用了该策略。该令牌由 ResourceManager 中的组件 RMDelegationTokenSecretManager 管理和维护。

- ❑ YARN Application Token。Application Token 用于保证 ApplicationMaster 与 ResourceManager 之间的通信安全。该 Token 的密钥 (masterKey) 由 ResourceManager 传递给 NodeManager, 并保存到 ApplicationMaster Container 的私有目录下。当 NodeManager 启动 ApplicationMaster 时, 所有的 Token 将被加载到 ApplicationMaster 的 UGI 中 (NodeManager 通过环境变量 HADOOP_TOKEN_FILE_LOCATION 将 Token 所在目录传递给 UGI, 这样 UGI 可以直接从文件中读取 Token 信息, 所有其他 Token 的传递过程也是一样的), 以在与 ResourceManager 通信时进行安全认证。需要注意的是, 该 Token 的生命周期与 ApplicationMaster 实例一致。该 Token 由 ResourceManager 中的 AMRMTOKENSecretManager 管理和维护。
- ❑ YARN NodeManager Token。ApplicationMaster 与 NodeManager 通信时, 需出示 NodeManager Token 以表明 ApplicationMaster 自身的合法性。该 Token 是由 ResourceManager 通过 RPC 函数 ApplicationMasterProtocol#allocate 的应答发送给 ApplicationMaster 的, 它的密钥是各个 NodeManager 向 ResourceManager 注册 (ResourceTracker#registerNodeManager) 和发送心跳信息 (ResourceTracker#nodeHeartbeat) 时领取的。ApplicationMaster 通过 ContainerManagementProtocol 协议与 NodeManager 通信时, 需要出示该 Token。该 Token 由 ResourceManager 中的 NMTOKENSecretManagerInRM 管理和维护。
- ❑ YARN Container Token。ApplicationMaster 与 NodeManager 通信启动 Container 时, 需出示 Container Token 以表明 Container 的合法性。该 Token 是由 ResourceManager 通过 RPC 函数 ApplicationMasterProtocol#allocate 的应答存放到 Container 中发送给 ApplicationMaster 的, 它的密钥是各个 NodeManager 向 ResourceManager 注册和发送心跳信息时领取的。ApplicationMaster 通过 RPC 函数 ContainerManagementProtocol#startContainer 与 NodeManager 通信启动 Container 时, 需要出示相应的 Container Token。该 Token 由 ResourceManager 中的 RMContainerTokenSecretManager 管理和维护。
- ❑ YARN Localizer Token。Localizer Token 用于保证 ContainerLocalizer 与 NodeManager 之间的通信安全。ContainerLocalizer 负责在任务运行之前从 HDFS 上下载各类所需的文件资源, 以构建一个本地执行环境, 在文件下载过程中, ContainerLocalizer 通过 RPC 协议 LocalizationProtocol 不断向 NodeManager 汇报状态信息。文件下载 (也称为“本地化”) 相关内容将在第 7 章介绍。
- ❑ MapReduce Client Token。MapReduce Client Token 用于保证 MapReduce JobClient

与 MapReduce Application Master 之间的通信安全。它由 ResourceManager 在作业提交时创建, 并通过 RPC 函数 `ApplicationClientProtocol#getApplicationReport` 发送给 JobClient。该 Token 由 ResourceManager 中的 `ClientToAMTokenSecretManagerInRM` 管理和维护。

- **MapReduce Job Token**。MapReduce Job Token 用于保证 MapReduce 的各个 Task (包括 Map Task 和 Reduce Task) 与 MapReduce Application Master 之间的通信安全。它由 ApplicationMaster 创建, 通过 RPC 函数 `ContainerManagementProtocol#startContainer` 传递给 NodeManager, 并由 NodeManager 写入 Container 的私有目录中, 以在任务启动时加载到 UGI 中, 从而使得任务可以安全地通过 RPC 协议 `TaskUmbilicalProtocol` 与 ApplicationMaster 通信。
- **MapReduce Shuffle Secret**。MapReduce Shuffle Secret 用于保证运行在各个 NodeManager 上的 ShuffleHandler (内部封装了一个 Netty Server) 与 Reduce Task 之间的通信安全, 即只有同一个作业的 Reduce Task 才允许读取该作业 Map Task 产生的中间结果。该安全机制是借助 Job Token 完成的。

5.8.3 Hadoop 授权机制

Hadoop 的授权机制是通过访问控制列表 (ACL) 实现的, 按照授权实体, 可分为队列访问控制列表、应用程序访问控制列表和服务访问控制列表, 下面分别介绍。

在正式介绍 YARN 授权机制之前, 先要了解 HDFS 的 POSIX 风格的文件访问控制机制, 这与当前 UNIX 的一致, 即将权限授予对象分为用户、同组用户和其他用户, 且可单独为每类对象设置一个文件的读、写和可执行权限。此外, 用户和用户组的关系是插拔式的, 默认情况下共用 UNIX/Linux 下的用户与用户组的对应关系, 这与 YARN 是一致的。

- **队列访问控制列表**：为了方便管理集群中的用户, YARN 将用户 / 用户组分成若干队列, 并可指定每个用户 / 用户组所属的队列。通常而言, 每个队列包含提交应用程序权限和管理应用程序权限 (比如杀死任意应用程序) 两种, 这些是通过资源调度器专属的配置文件设置的, 具体可参考第 6 章。
- **应用程序访问控制列表**：应用程序访问控制机制的设置方法已经在 5.5 节进行了介绍, 主要方法是在客户端为每类 `ApplicationAccessType` (目前只有 `VIEW_APP` 和 `MODIFY_APP` 两种类型) 设置对应的用户列表, 这些信息传递到 ResourceManager 端后, 由它维护和使用。通常而言, 为了用户使用方便, 应用程序可对外提供一些特殊的可直接设置的参数 (而不是通过 API 设置)。以 MapReduce 作业为例, 用户可以通过参数 `mapreduce.job.acl-view-job` 和 `mapreduce.job.acl-modify-job` 为每个作业单独设置查看和修改权限。需要注意的是, 默认情况下, 作业拥有者和超级用户 (可配置) 拥有以上两种权限且不可以修改。
- **服务访问控制列表**：服务访问控制是 Hadoop 提供的最原始的授权机制, 它用于确保只有那些经过授权的客户端才能访问对应的服务。比如可通过为 `Application-`

ClientProtocol 协议设置访问控制列表以指定哪些用户可以向集群中提交应用程序。

服务访问控制是通过控制各个服务之间的通信协议实现的，它通常发生在其他访问控制机制之前，比如文件权限检查、队列权限检查等。

为了启用该功能，管理员需在 core-site.xml 中将参数 `hadoop.security.authorization` 置为 true，并在 `hadoop-policy.xml` 中为各个通信协议指定具有访问权限的用户或者用户组。管理员可为 YARN/MapReduce 中的 8 个协议添加了访问控制列表，如表 5-3 所示。

表 5-3 Hadoop 的各个 ACL 配置

属性名称	含 义
<code>security.resourcetracker.protocol.acl</code>	ACL for ResourceTracker, NodeManager 与 Resource-Manager 之间的通信协议
<code>security.admin.protocol.acl</code>	ACL for ResourceManagerAdministrationProtocol, 客户端（管理员）与 ResourceManager 之间的协议，主要用于更新配置
<code>security.client.resourcemanager.protocol.acl</code>	ACL for ApplicationClientProtocol, 客户端（普通用户）与 ResourceManager 之间的通信协议，可设置哪些用户允许向集群提交作业、查询集群状态等
<code>security.applicationmaster.resourcemanager.protocol.acl</code>	ACL for ApplicationMasterProtocol, Application-Master 与 ResourceManager 之间的通信协议
<code>security.containermanager.protocol.acl</code>	ACL for ContainerManagementProtocol, Application-Master 与 NodeManager 之间通信协议
<code>security.resourcelocalizer.protocol.acl</code>	ACL for ResourceLocalizerProtocol, NodeManager 与 ResourceLocalizer 之间通信协议
<code>security.job.task.protocol.acl</code>	ACL for TaskUmbilicalProtocol, Task 与对应的 Application-Master 之间通信协议
<code>security.job.client.protocol.acl</code>	ACL for MRClientProtocol, 客户端与对应的 Application-Master 之间的通信协议

这 8 个 ACL 的配置方法相同，即每个 ACL 可配置多个用户和用户组，用户之间用“,”分割，用户组之间用“,”分割，而用户和用户组之间用空格分割。注意，如果只有用户组，前面必须保留一个空格，比如：

```
<property>
  <name>security.client.resourcemanager.protocol.acl</name>
  <value>alice,bob group1,group2</value>
</property>
```

上述代码表示用户 `alice` 和 `bob`、用户组 `group1` 和 `group2` 可向 Hadoop 集群中提交作业。又如：

```
<property>
  <name>security.client.resourcemanager.protocol.acl</name>
  <value>*</value>
</property>
```

上述代码表示所有用户和分组均可访问 Hadoop。

注意 默认情况下, 前两个通信协议只对 YARN 服务启动用户开放访问权限, 其他几个对任何用户和分组都开放。

hadoop-policy.xml 文件可使用以下命令动态加载 YARN 相关配置:

```
bin/yarn rmadmin -refreshServiceAcl
```

注意, 只有属性 security.refresh.policy.protocol.acl 指定的用户才可以更新该配置文件。

安全管理是 Hadoop 中最复杂的和最难懂的模块, 涉及 Hadoop 的各个分支和每个分支的各个服务与组件, 为了方便大家详细了解 Hadoop 内部的安全机制实现和各个验证流程, Apache 正在编写一个文档, 具体可参考 HADOOP-9621^①。

5.9 容错机制

在 Hadoop 1.0 中, HDFS 和 MapReduce (MRv1) 均采用了 Master/Slave 结构, 这种结构虽然具有设计非常简单的优点, 但同时也存在 Master 单点故障问题。由于存在单点故障问题的系统不适合在线应用场景, 这使得 Hadoop 在相当长时间内仅用于离线存储和计算。在 Hadoop 2.0 中, HDFS 同样面临着单点故障问题, 但由于每个 MapReduce 作业拥有自己的作业管理组件 (ApplicationMaster), 因此不再存在单点问题, 但新引入的资源管理系统 YARN 也采用了 Master/Slave 结构, 同样出现了单点故障问题。

作为一个分布式系统, YARN 必须具备的一个特点是高容错性, 因此 YARN 需要考虑 ApplicationMaster、NodeManager、Container 和 ResourceManager 等服务或组件的容错性。这些服务或组件的容错机制如下。

- ApplicationMaster 容错: 前面提到, 不同类型的应用程序拥有不同的 ApplicationMaster, 而 ResourceManager 负责监控 ApplicationMaster 的运行状态, 一旦发现它运行失败或者超时, 会为其重新分配资源并启动它。至于启动之后 ApplicationMaster 内部的状态如何恢复需要由自己保证, 比如 MRAppMaster (MapReduce ApplicationMaster) 在作业运行过程中将状态信息动态记录到 HDFS 上, 一旦出现故障重启后, 它能够从 HDFS 读取并恢复之前的运行状态, 以减少重新计算带来的开销。
- NodeManager 容错: 如果 NodeManager 在一定时间内未向 ResourceManager 汇报心跳信息 (可能是网络原因或者自身原因), 则 ResourceManager 认为它已经死掉了, 会将它上面所有正在运行的 Container 状态置为失败, 并告诉对应的 ApplicationMaster (如果 AM Container 运行失败, 则需重新分配资源启动 ApplicationMaster), 以决定如何处理这些 Container 中运行的任务。
- Container 容错: 如果 ApplicationMaster 在一定时间内未启动分配到的 Container, 则 ResourceManager 会将该 Container 状态置为失败并回收它; 如果一个 Container 在运行过程中, 因为外界原因 (比如资源不足、误杀等) 导致运行失败, 则 ResourceManager

^① 参见网址 <https://issues.apache.org/jira/browse/HADOOP-9621>。

会转告给对应的 ApplicationMaster, 由它决定如何处理。

- ResourceManager 容错: ResourceManager 负责整个集群的资源管理和调度, 它的重要性不言而喻, 因此它自身的容错性直接决定了 YARN 的可用性和可靠性。本节也将重点介绍 ResourceManager 的容错机制。

5.9.1 Hadoop HA 基本框架

在 Master/Slave 架构中, 为了解决 Master 的单点故障问题 (也称为高可用问题, 即 HA, High Availability), 通常采用热备方案, 即集群中存在一个对外服务的 Active Master 和若干个处于就绪状态的 Standby Master, 一旦 Active Master 出现故障, 立即采用一定的策略选取某个 Standby Master 转换为 Active Master 以正常对外提供服务。同样, Hadoop 2.0 也正是采用这种方案解决各系统的单点故障问题。

总体上说, Hadoop 2.0 中的 HDFS 和 YARN 均采用了基于共享存储的 HA 解决方案, 即 Active Master 不断将信息写入一个共享存储系统, 而 Standby Master 则不断读取这些信息, 以与 Active Master 的内存信息保持同步。当需要主备切换时, 选中的 Standby Master 需先保证信息完全同步后, 再将自己的角色切换至 Active Master。目前而言, 常用的共享存储系统有以下几个:

- Zookeeper: Zookeeper 是一个针对大型分布式系统的可靠协调系统, 提供的功能包括统一命名服务、配置管理、集群管理、共享锁和队列管理等。需要注意的是, Zookeeper 设计目的并不是数据存储, 但它的确可以安全可靠地存储少量数据以解决分布式环境下多个服务之间的数据共享问题。
- NFS (Network File System): NFS 是一种非常经典的数据共享方式, 它可以透过网络, 让不同的机器和不同的操作系统之间彼此共享文件。
- HDFS: Hadoop 自带的分布式文件系统, 由于它本身存在单点故障问题, 因此 Hadoop 的单点问题不能够通过它解决。
- BookKeeper^①: 由 Zookeeper 项目产生的一个分支项目, 主要用于可靠地记录日志流, 它采用的是分布式多副本解决方案。
- QJM (Quorum Journal Manager): QJM 的基本原理就是用 $2N+1$ 个节点存储数据, 每次有大多数 (大于等于 $N+1$) 节点成功写入数据即认为该次写成功, 并能保证数据高可用。该算法最多容忍 N 台机器挂掉, 如果多于 N 台挂掉, 则这个算法就会失效。

在 Hadoop 2.0 中, YARN HA 采用了基于 Zookeeper 的方案, MRv1 HA (CDH4 发行版中打包了 MRv1 实现, 它采用 HDFS 解决了 JobTracker 单点故障问题^②) 采用了基于 HDFS 的方案, 而 HDFS HA 则提供了基于 NFS、BookKeeper 和 QJM 的三套实现方案。由于引入了共享存储系统, Hadoop 中各个系统实际上是 “Share Nothing But NameNode”。尽管这

① 参见网址 <http://zookeeper.apache.org/bookkeeper/>。

② 由于 HDFS 本身就存在单点故障, 因此, MRv1 基于 HDFS 解决单点故障的假设是, HDFS 的单点故障问题已通过其他方案解决或者它们 (MRv1 与 HDFS) 不会同时出现故障。

几个系统采用的共享存储不同，但它们的 HA 架构是相同的，均分为手动模式和自动模式。其中，手动模式是指由管理员通过命令进行主备切换，这通常用于服务升级；自动模式可降低运维成本，但存在潜在危险。手动模式架构如图 5-21 所示。在该模式中，Master 自身需实现 HADServiceProtocol 协议，或者由一个实现该协议的服务控制，而管理员可通过一个 HADServiceProtocol 协议客户端控制各个 Master 状态。

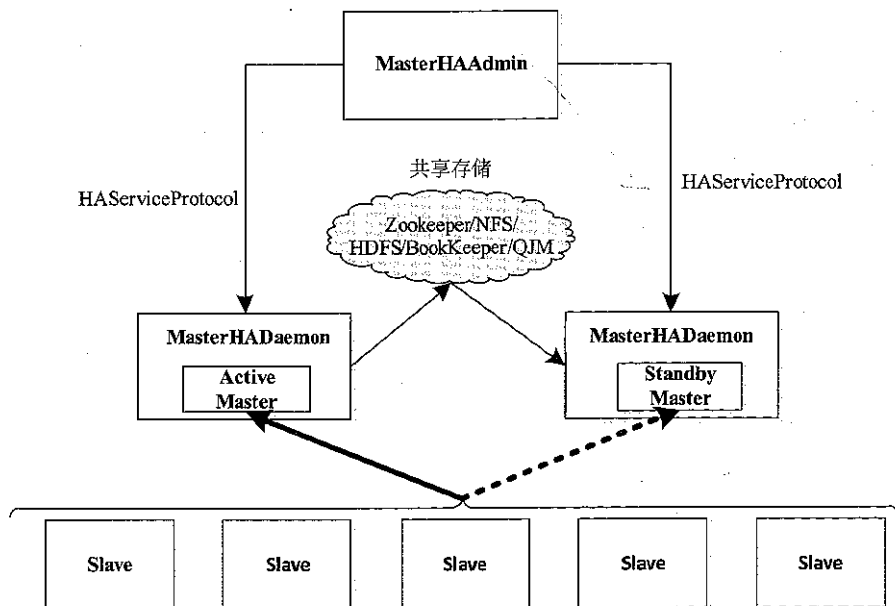


图 5-21 Hadoop HA 手动模式基本框架

自动模式架构如图 5-22 所示。

自动模式主要由以下几个组件构成：

- ❑ MasterHADaemon：与 Master 服务运行在同一个进程中，可接收外部 RPC 命令，以控制 Master 服务的启动和停止。
- ❑ SharedStorage：共享存储系统，Active Master 将信息写入共享存储系统，而 Standby Master 则读取该信息以保持与 Active Master 的同步。
- ❑ ZKFailoverController：基于 Zookeeper 实现的切换控制器，主要由 ActiveStandbyElector 和 HealthMonitor 两个核心组件构成。其中，ActiveStandbyElector 负责与 Zookeeper 集群交互，通过尝试获取全局锁，以判断所管理的 Master 是进入 Active 还是进入 Standby 状态；HealthMonitor 负责监控各个活动 Master 的状态，以根据它们状态进行状态切换。
- ❑ Zookeeper：核心功能是通过维护一把全局锁控制整个集群有且仅有一个 Active Master。当然，如果 SharedStorage 采用了 Zookeeper，则还会记录一些其他状态和运行时信息。

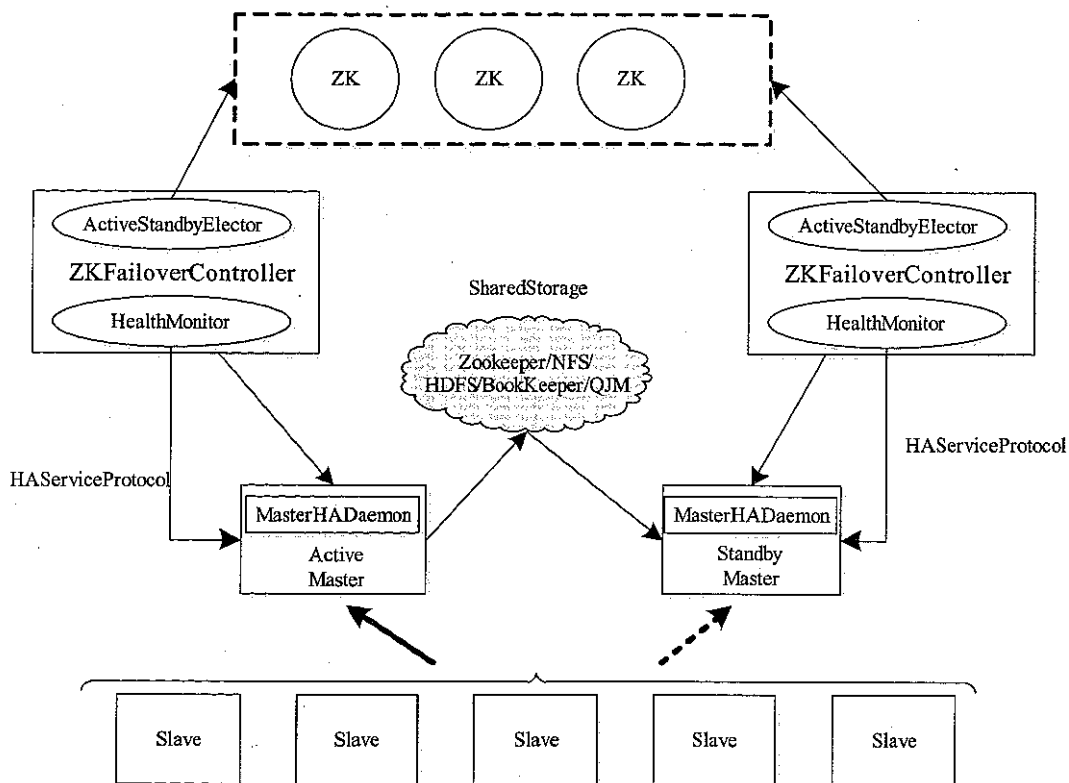


图 5-22 Hadoop HA 自动模式基本框架

要解决 HA 问题需考虑以下几个问题：

（1）脑裂（brain-split）

脑裂是指在主备切换时，由于切换不彻底或其他原因，导致客户端和 Slave 误以为出现两个 Active Master，最终使得整个集群处于混乱状态。通常采用隔离（Fencing）机制解决脑裂问题。解决脑裂问题需从以下三个方面考虑：

- ❑ 共享存储隔离：确保只有一个 Master 往共享存储中写数据。
- ❑ 客户端隔离：确保只有一个 Master 可以响应客户端的请求。
- ❑ Slave 隔离：确保只有一个 Master 可以向 Slave 下发命令。

Hadoop 公共库中对外提供了两种隔离实现，分别是 sshfence 和 shellfence。其中 sshfence 是指通过 SSH 登录目标 Master 节点上，使用命令 fuser 将进程杀死（通过 TCP 端口号定位进程 PID，该方法比 JPS 命令更准确）；shellfence 是指执行一个用户事先定义的 Shell 命令（脚本）完成隔离。

（2）切换对外透明

为了保证整个切换是对外透明的，Hadoop 应保证所有客户端和 Slave 能自动重定向到新的 Active Master 上，这通常是通过若干次尝试连接旧 Master 不成功，再重新尝试链

接新 Master 完成的，整个过程有一定延迟。在新版本的 Hadoop RPC 中，用户可自行设置 RPC 客户端尝试机制、尝试次数和尝试超时时间等参数。

5.9.2 YARN HA 实现

YARN 将共享存储系统抽象成 `RMStateStore`（它是一个 Java 接口），以保存（出故障后）恢复 `ResourceManager` 所必需的信息，包括：

- `Application` 状态信息 `ApplicationState`。内部包含应用程序提交描述信息 `context`（对应类型为 `ApplicationSubmissionContext`）、提交时间 `submitTime`（对应类型为 `long`）、拥有者 `user`（对应类型为 `String`）三个字段。
- `Application` 对应的每个 `ApplicationAttempt` 信息 `ApplicationAttemptState`。内部包含 `attemptId`（对应类型为 `ApplicationAttemptId`）、所在 `Container` 的信息 `masterContainer`（对应类型为 `Container`）、安全 `Token`（对应类型为 `Credentials`）三个字段。
- 安全令牌相关信息 `RMDTSecretManagerState`。内部包含 `delegationTokenState`（授权令牌状态）、`masterKeyState`（master key 状态）、`dtSequenceNumber`（序列号）三个字段。

`ResourceManager` 并不会保存已经分配给每个 `ApplicationMaster` 的资源信息和每个 `NodeManager` 的资源使用信息，这些均可通过相应的心跳汇报机制重构出来。也正因如此，`ResourceManager HA` 的实现是非常轻量级的。

`ResourceManager` 提供了四种 `RMStateStore` 实现（管理员可通过参数 `yarn.resource-manager.store.class` 设置采用的 `RMStateStore` 实现类），分别是：`NullRMStateStore`（不存储任何状态信息，在不启用恢复机制时，它是默认实现的）、`MemoryRMStateStore`（将状态信息存储到内存中，在启用恢复机制时，它是默认实现的）、`FileSystemRMStateStore`（将状态信息存储到 HDFS 中）、`ZKRMStateStore`（将状态信息存储到 Zookeeper 上）^①，其中，前两种仅用于测试，后两种可用于 HA 实现中。但由于 HDFS 本身就存在单点故障问题，因此 YARN HA 最佳实践是采用基于 `ZKRMStateStore` 的共享存储方案。

截至本书结稿时，`ResourceManager` 的容错机制完成度很低，可认为处于“初级阶段”^②。它能够完成的功能是，当 `ResourceManager` 重新启动时，已经运行的应用程序无须重新提交，但那些正在运行的 `Container` 将被杀死。也就是说，当前实现的容错机制带来的好处仅是用户无须重新提交应用程序，而正在运行的 `Container` 将不得不重新运行。

总结起来，目前 `ResourceManager` 的容错机制涉及到的流程如下：

步骤 1 `ResourceManager` 收到客户端发送的提交应用程序请求后，在响应之前，会将应用程序的“application submission context”同步保存下来。这有两点好处：

- 这样可确保一旦客户端成功提交应用程序后，无论何时重启，`ResourceManager` 均有足够的信息将应用程序恢复出来。

① <https://issues.apache.org/jira/browse/YARN-353>。

② <https://issues.apache.org/jira/browse/YARN-128>，<https://issues.apache.org/jira/browse/YARN-149>。

□ 考虑到用户提交应用程序的频率很低，且 ClientRMService 使用了一个独立的 RPC 通道，因此同步记录信息不会产生太大的性能开销。

步骤 2 当一个新的应用程序运行实例被创建 (RMAppAttempt) 时，它的基本信息和之前失败的运行实例的信息将被保存下来。

□ 当 ResourceManager 重启后，它需要获取之前失败的应用程序实例 ID 以产生一个新的运行实例 ID (ID 应是连续的)。

□ 相关信息将被异步记录下来，这主要是考虑到相关事件由中央异步调度器 Async-Dispatcher 处理，而同步记录日志会造成性能问题。

步骤 3 当一个应用程序运行结束后，它保存的信息将被移除。考虑到性能问题，该过程是异步进行的。

步骤 4 ResourceManager 重启后，它首先将之前保存的数据加载到内存中，这些信息将用于构建 RMApp 对象和创建新的运行实例 RMAppAttempt。之后，内部服务将被启动，所有操作照旧进行。

□ 重启的 ResourceManager 收到 NodeManager 汇报的心跳后，发现该 NodeManager 并不存在，则要求它重新启动，这样，它上面正在运行的所有 Container 将被杀死。

□ 重启的 ResourceManager 收到 ApplicationMaster 汇报的心跳后，发现心跳中涉及的运行实例 ID 并不存在，从而会向 ApplicationMaster 发送一个 RemoteException 异常，这将导致 ApplicationMaster 异常退出。

□ NodeManager 重新启动并向 ResourceManager 注册，新的运行实例将被启动，接下来所有操作照旧进行。

随着 YARN HA 方案的完善，最终它可像 HDFS 那样借助 ZKFailoverController 完成自动主备切换，进而使得 YARN 具有更高的可用性，从而促使 YARN 成为集离线应用和在线应用于一体的资源管理系统。有兴趣的读者可追踪 YARN jira 上关于 YARN HA 的两个任务的进展：YARN-149 和 YARN-128。

5.10 源代码阅读引导

ResourceManager 的所有实现在源代码目录下的 `hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java` 目录中，代码结构如图 5-23 所示。

ResourceManager 由多个内部服务组成，这些服务同时也可能是事件处理器，因此，通常可按照以下方法学习这些服务的实现：先单独了解每个服务的内部实现，包括它们要处理的事件类型和每种事件的处理逻辑，然后跳出单个服务，从全局看，不同服务之间是如何通过事件串联在一起的，为了便于理解和总结，可以边阅读代码边画出类似于本章前几节给出的事件转换图。下面依次介绍 ResourceManager 服务涉及的 Java 实现包：

```

hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java
├─ org.apache.hadoop.yarn.server.resourcemanager
├─ org.apache.hadoop.yarn.server.resourcemanager.amlauncher
├─ org.apache.hadoop.yarn.server.resourcemanager.recovery
├─ org.apache.hadoop.yarn.server.resourcemanager.recovery.records
├─ org.apache.hadoop.yarn.server.resourcemanager.recovery.records.impl.pb
├─ org.apache.hadoop.yarn.server.resourcemanager.resource
├─ org.apache.hadoop.yarn.server.resourcemanager.rmapp
├─ org.apache.hadoop.yarn.server.resourcemanager.rmapp.attempt
├─ org.apache.hadoop.yarn.server.resourcemanager.rmapp.attempt.event
├─ org.apache.hadoop.yarn.server.resourcemanager.rmcontainer
├─ org.apache.hadoop.yarn.server.resourcemanager.rmnode
├─ org.apache.hadoop.yarn.server.resourcemanager.scheduler
├─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity
├─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.common.fica
├─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.event
├─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair
├─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.policies
├─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.fifo
├─ org.apache.hadoop.yarn.server.resourcemanager.security
├─ org.apache.hadoop.yarn.server.resourcemanager.security.authorize
├─ org.apache.hadoop.yarn.server.resourcemanager.webapp
├─ org.apache.hadoop.yarn.server.resourcemanager.webapp.dao

```

图 5-23 ResourceManager 源代码组织结构

- `org.apache.hadoop.yarn.server.resourcemanager` : ResourceManager 主类以及部分内部重要服务的实现包, 包括 ResourceManager (组装所有服务, main 函数所在类)、AdminService (实现了 ResourceManagerAdministrationProtocol 协议, 处理来自管理员的 RPC 请求)、ClientRMService (实现了 ApplicationClientProtocol 协议, 处理来自客户端的 RPC 请求)、ResourceTrackerService (实现了 ResourceTracker 协议, 处理来自 NodeManager 的 RPC 请求)、ApplicationMasterService (实现了 ApplicationMasterProtocol 协议, 处理来自 ApplicationMaster 的 RPC 请求)、RMContextImpl (ResourceManager 中的全局上下文信息) 等。
- `org.apache.hadoop.yarn.server.resourcemanager.amlauncher` : 该包负责应用程序 ApplicationMaster 的启动和停止, 主要由 ApplicationMasterLauncher 和 AMLauncher 两个组件构成, 其中 ApplicationMasterLauncher 维护了一个 AMLauncher 线程池, 将收到的 ApplicationMaster 启动和停止等事件交给 AMLauncher 线程处理。
- `org.apache.hadoop.yarn.server.resourcemanager.recovery.*` : 这几个包实现了 ResourceManager 重启恢复相关的功能, 它目前实现了 4 种 RMStateStore : NullRMStateStore (未进行任何处理)、MemoryRMStateStore (将状态信息保存到内存中)、FileSystemRMStateStore (将状态信息保存到文件中, 可用于 ResourceManager 服务升级) 和 ZKRMStateStore^① (将状态信息保存到 Zookeeper 上), 而 YARN 倾向于使

① <https://issues.apache.org/jira/browse/YARN-353>。

用 ZKRMStateStore 解决 ResourceManager HA 问题。

- `org.apache.hadoop.yarn.server.resourcemanager.resource` : 该包描述了目前 YARN 支持的资源类型, 目前仅支持 CPU 和内存两种资源。
- `org.apache.hadoop.yarn.server.resourcemanager.rmapp` : 该包给出了应用程序状态机 `RMAppImpl` 实现以及它处理的 `RMAppEvent` 事件的定义。
- `org.apache.hadoop.yarn.server.resourcemanager.rmapp.attempt.*` : 该包给出了一个应用程序运行实例的状态机 `RMAppAttemptImpl` 实现以及它处理的 `RMAppAttemptEvent` 事件的定义。
- `org.apache.hadoop.yarn.server.resourcemanager.rmcontainer` : 该包给出了 Container 状态机 `RMContainerImpl` 的实现以及它处理的 `RMContainerEvent` 事件的定义。
- `org.apache.hadoop.yarn.server.resourcemanager.rmnode` : 该包给出了节点状态机 `RMNodeImpl` 的实现以及它处理的 `RMNodeEvent` 事件的定义。
- `org.apache.hadoop.yarn.server.resourcemanager.scheduler.*` : 这些包给出了资源调度器的几个常用实现, 包括 FIFO、Fair Scheduler 和 Capacity Scheduler, 具体将在下一章中详细介绍。
- `org.apache.hadoop.yarn.server.resourcemanager.security.*` : 这些包给出了 ResourceManager 中安全相关的几个类, 分别是 `AMRMTokenSecretManager`、`ClientToAMTokenSecretManagerInRM`、`DelegationTokenRenewer`、`NMTokenSecretManagerInRM`、`RMContainerTokenSecretManager` 和 `RMDelegationTokenSecretManager`, 5.8 节对它们已进行了详细介绍。
- `org.apache.hadoop.yarn.server.resourcemanager.webapp` : ResourceManager 对外提供的 Web 展示界面, 通过该界面, 用户可查看已提交应用程序的实时运行信息、集群节点列表及状态和各队列资源使用信息。

5.11 小结

在 YARN 中, ResourceManager 扮演管理者的角色, 它负责整个集群资源的管理与调度。本章详细介绍了 ResourceManager 内部各个功能模块的实现, 包括 ApplicationMaster 管理 (启动、停止等)、NodeManager 管理、Application 管理、状态机管理等。

为了能够让读者了解 ResourceManager 内部工作原理, 本章分析了 ResourceManager 的几个关键行为工作流程, 包括启动 ApplicationMaster、申请与分配 Container、杀死 Application、Container 超时、ApplicationMaster 超时、NodeManager 超时等。

最后, 本章介绍了 ResourceManager 在安全管理和容错方面的设计与实现。

5.12 问题讨论

问题 1：截至本书结稿时，ResourceManager HA 仍然没有实现，那应该如何解决 ResourceManager 的单点故障问题？

问题 2：ResourceManager 中共涉及 4 类状态机，且这些状态机在不断完善中，试着分析你所使用的 Hadoop 版本中的实现与本书中描述的不同之处，并通过电子邮件（dongxicheng@yahoo.com）告诉笔者。

第6章 资源调度器

资源调度器是 Hadoop YARN 中最核心的组件之一，它是 ResourceManager 中的一个插件式服务组件，负责整个集群资源的管理和分配。Hadoop 最初是为批处理作业而设计的，当时（MRv1）仅采用了一个简单的 FIFO 调度机制分配任务。但随着 Hadoop 的普及，单个 Hadoop 集群中的用户量和应用程序种类不断增加，适用于批处理场景的 FIFO 调度机制不能很好地利用集群资源，也不能够满足不同应用程序的服务质量要求，因此，设计适用于多用户的资源调度器势在必行。

从目前看来，主要有两种多用户资源调度器的设计思路：第一种是在一个物理集群上虚拟多个 Hadoop 集群，这些集群各自拥有全套独立的 Hadoop 服务，典型的代表是 HOD（Hadoop On Demand）调度器；另一种是扩展 YARN 调度器，使之支持多个队列多用户，这也是本章的重点。

YARN 资源调度器是直接从 MRv1 基础上修改而来的，它提供了三种可用资源调度器，分别是 FIFO（First In First Out）、Yahoo! 的 Capacity Scheduler 和 Facebook 的 Fair Scheduler，它们的实现原理和实现细节与 MRv1 中对应的三种调度器基本一致。

在 YARN 中，资源调度器是以层级队列方式组织资源的，这种组织方式符合公司的组织架构，有利于资源在不同资源间分配和共享，进而提高集群资源利用率。本章将重点介绍 Capacity Scheduler 和 Fair Scheduler 两种多用户资源调度器的应用场景和设计原理。

6.1 资源调度器背景

Hadoop 最初设计目的是支持大数据批处理作业，如日志挖掘、Web 索引等作业，为此，Hadoop 仅提供了一个非常简单的调度机制：FIFO，即先来先服务，在该调度机制下，所有作业被统一提交到一个队列中，Hadoop 按照提交顺序依次运行这些作业。

但随着 Hadoop 的普及，单个 Hadoop 集群的用户量越来越大，不同用户提交的应用程序往往具有不同的服务质量要求（Quality of Service, QoS），典型的应用有以下几种：

- 批处理作业。这种作业往往耗时较长，对时间完成一般没有严格要求，如数据挖掘、机器学习等方面的应用程序。
- 交互式作业。这种作业期望能及时返回结果，如 SQL 查询（Hive）等。
- 生产性作业。这种作业要求有一定量的资源保证，如统计值计算、垃圾数据分析等。

此外，这些应用程序对硬件资源需求量也是不同的，如过滤、统计类作业一般为 CPU 密集型作业，而数据挖掘、机器学习作业一般为 I/O 密集型作业。因此，简单的 FIFO 调度策略不仅不能满足多样化需求，也不能充分利用硬件资源。

为了克服单队列 FIFO 调度器的不足,多种类型的多用户多队列调度器诞生了。当前主要有两种多用户资源调度器的设计思路:第一种是在一个物理集群上虚拟多个 Hadoop 集群,这些集群各自拥有全套独立的 Hadoop 服务,典型的代表是 HOD 调度器;另一种是扩展 Hadoop 调度器,使之支持多个队列多用户,这种调度器允许管理员按照应用需求对用户或者应用程序分组,并为不同的分组分配不同的资源量,同时通过添加各种约束防止单个用户或者应用程序独占资源,进而能够满足各种 QoS 需求,典型的代表是 Yahoo! 的 Capacity Scheduler 和 Facebook 的 Fair Scheduler。接下来我们将分别介绍这两种多用户资源调度器。

6.2 HOD 调度器

HOD(Hadoop On Demand)调度器^①是一个在共享物理集群上管理若干个 Hadoop 集群的工具,用户可通过 HOD 调度器在一个共享物理集群上快速搭建若干个独立的虚拟 Hadoop 集群,以满足不同的用途,比如运行不同类型的应用程序、运行不同的 Hadoop 版本进行测试等。HOD 调度器可使管理员和用户轻松地快速搭建和使用 Hadoop。HOD 是 Hadoop 1.0 提供的调度器,由于这种调度器本身的局限性,Hadoop 2.0 的发行版中不再包含它。为了让读者理解 HOD 的工作原理和局限性,本节以 Hadoop 1.0 为例介绍 HOD。

HOD 调度器首先使用 Torque 资源管理器^②为一个虚拟 Hadoop 集群分配节点,然后在分配的节点上启动 MapReduce 和 HDFS 中的各个守护进程,并自动为 Hadoop 守护进程和客户端生成合适的配置文件(包括 mapred-site.xml、core-site.xml 和 hdfs-site.xml 等)。接下来我们将分别介绍 Torque 资源管理器和 HOD 调度器的基本工作原理。

6.2.1 Torque 资源管理器

HOD 调度器的工作过程实现中依赖于一个资源管理器来为它分配、回收节点和管理各节点上的作业运行的情况,如监控作业的运行、维护作业的运行状态等。而 HOD 只需在资源管理器所分配的节点上运行 Hadoop 守护进程和 MapReduce 作业即可。当前 HOD 采用的资源管理器是开源的 Torque 资源管理器。

一个 Torque 集群由一个头节点和若干计算节点组成。头节点上运行一个名为 pbs_server 的守护进程,主要用于管理计算节点和监控各个作业的运行状态。每个计算节点上运行一个名为 pbs_mom 的守护进程,用于执行主节点分配的作业。此外,用户可将任何节点作为客户端,用以提交和管理作业。

头节点内部还运行了一个调度器守护进程,该守护进程会与 pbs_server 进行通信,以决定对资源使用和作业分配的本地策略。默认情况下,调度守护进程采用了 FIFO 调度机制,它将所有作业存放到一个队列中,并按照到达时间依次对它们进行调度。需要注意的

^① http://hadoop.apache.org/docs/stable/hod_scheduler.html。

^② <http://www.adaptivecomputing.com/products/open-source/torque/>。

是，Torque 中的调度机制是可插拔的，Torque 还提供许多其他可选的作业调度器。

如图 6-1 所示，用户可通过 qsub 命令向物理集群中提交作业，而 Torque 内部执行流程如下：

步骤 1 当 pbs_server 收到新作业后，会进一步通知调度器。

步骤 2 调度器采用一定的策略为该作业分配节点，并将节点列表与节点对应的作业执行命令返回给 pbs_server。

步骤 3 pbs_server 将作业发送给第一个节点。

步骤 4 第一个节点启动作业，作业开始运行（该作业会通知其他节点执行相应命令）。

步骤 5 作业运行完成或者资源申请到期后，Torque 会回收资源。

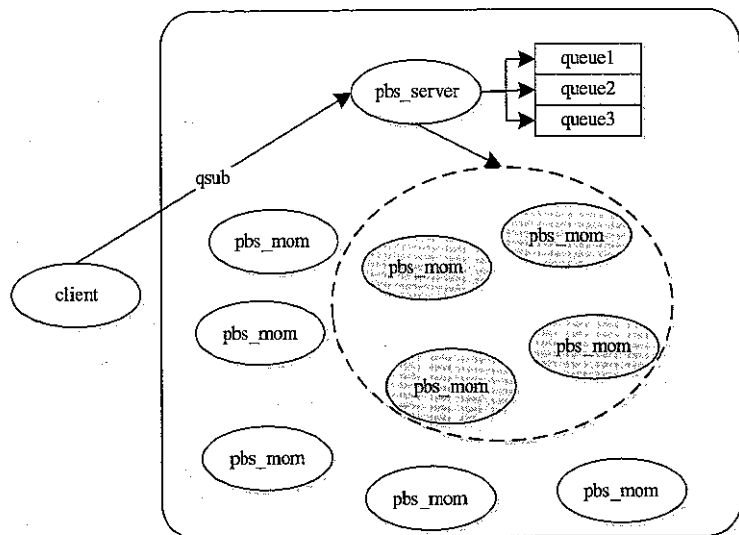


图 6-1 Torque 内部工作原理

6.2.2 HOD 作业调度

理解了 Torque 工作原理后，HOD 调度器工作原理便一目了然：首先利用 Torque 向物理集群申请一个虚拟机群，然后将 Hadoop 守护进程包装成一个 Torque 作业，并在申请的节点上启动，最后用户可直接向启动的 Hadoop 集群中提交作业。通过 HOD 调度器申请集群和运行作业的主要流程如下：

步骤 1 用户向 HOD 调度器申请一个包含一定数目节点的集群，并要求该集群中运行一个 Hadoop 实例。

步骤 2 HOD 客户端利用资源管理器接口 qsub 提交一个被称为 RingMaster 的进程作为 Torque 作业，同时申请一定数目的节点。这个作业被提交到 pbs_server 上。

步骤 3 在各个计算节点上，守护进程 pbs_mom 接受并处理 pbs_server 分配的作业。RingMaster 进程在其中一个计算节点上开始运行。

步骤4 Ringmaster 通过 Torque 的另外一个接口 pbsdsh 在所有分配到的计算节点上运行第二个 HOD 组件 HodRing，即运行于各个计算节点上的分布式任务。

步骤5 HodRing 初始化之后会与 RingMaster 通信以获取 Hadoop 指令，并根据指令启动 Hadoop 服务进程，一旦服务进程开始启动，它们会向 RingMaster 登记，提供关于守护进程的信息。

注意 Hadoop 实例所需的配置文件全部由 HOD 自己生成。HOD 客户端保持和 RingMaster 的通信，可以获取 MapReduce 和 HDFS 守护进程所在的位置。

步骤6 Hadoop 实例启动之后，用户可以向集群中提交 MapReduce 作业。

步骤7 如果一段时间内 Hadoop 集群上没有作业运行，Torque 会回收该虚拟 Hadoop 集群的资源。

管理员将一个物理集群划分成若干个 Hadoop 集群后，用户可将不同类型的应用程序提交到不同 Hadoop 集群上，这样可避免不同用户或者不同应用程序之间争夺资源，从而达到多用户共享集群的目的。

从集群管理和资源利用率两方面看，这种基于完全隔离的集群划分方法存在诸多问题。

- 从集群管理角度看，多个 Hadoop 集群会给运维人员造成管理上的诸多不便。
- 多个 Hadoop 集群会导致集群整体利用率低下，这主要是负载不均衡造成的，比如某个集群非常忙碌时另外一些集群可能空闲，也就是说，多个 Hadoop 集群无法实现资源共享。
- 考虑到虚拟集群回收后数据可能丢失，用户通常将虚拟集群中的数据写到外部的 HDFS 上，如图 6-2 所示。用户通常仅在虚拟集群上安装 MapReduce，至于 HDFS，则使用一个外部全局共享的 HDFS。很明显，这种部署方法会丧失部分数据本地特性。为了解决该问题，一种更好地方法是在整个集群中只保留一个 Hadoop 实例，

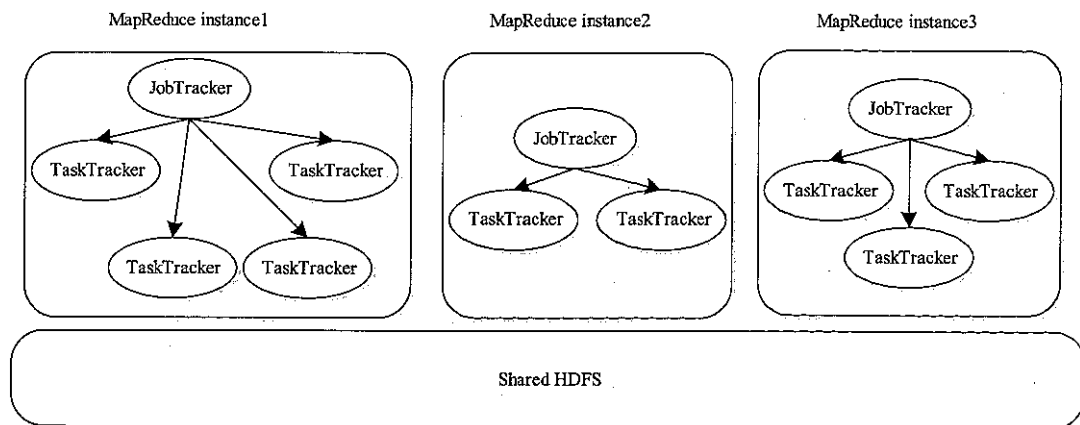


图 6-2 基于外部 HDFS 的多个虚拟 MapReduce 集群

而通过 Hadoop 调度器将整个集群中资源划分给若干个队列，并让这些队列共享所有节点上的资源，当前 Yahoo! 的 Capacity Scheduler 和 Facebook 的 Fair scheduler 正是采用了这种设计思路。

6.3 YARN 资源调度器的基本架构

6.3.1 基本架构

资源调度器是 YARN 中最核心的组件之一，且是插拔式的，它定义了一整套接口规范以便用户可按照需要实现自己的调度器。YARN 自带了 FIFO、Capacity Scheduler 和 Fair Scheduler 三种常用资源调度器，当然，用户可按照接口规范编写一个新的资源调度器，并通过简单的配置使它运行起来。本小节从插拔特性和事件处理器两个角度介绍 YARN 的基本架构。

(1) 作为一个插拔式组件

YARN 中的资源调度器是插拔式的，ResourceManager 在初始化时会根据用户的配置创建一个资源调度器对象，相关代码如下：

```
String schedulerClassName = conf.get(YarnConfiguration.RM_SCHEDULER,
    YarnConfiguration.DEFAULT_RM_SCHEDULER);
try {
    Class<?> schedulerClazz = Class.forName(schedulerClassName);
    if (ResourceScheduler.class.isAssignableFrom(schedulerClazz)) {
        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);
}
```

管理可通过参数 `yarn.resourcemanager.scheduler.class` 设置资源调度器的主类，默认是 `org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler`，即 `CapacityScheduler` ^①。

所有的资源调度器均应该实现接口 `org.apache.hadoop.yarn.server.resourcemanager.scheduler.ResourceScheduler`，该接口的定义如下：

```
public interface ResourceScheduler extends YarnScheduler, Recoverable {
    // 重新初始化 ResourceScheduler，通常在 ResourceManager 初始化时调用
    // (包括主备 ResourceManager 切换)
```

① 在 2.0.2-alpha 版本之前，默认的资源调度器是 FIFO，但该调度器的设计存在缺陷，因此默认调度器被换为 CapacityScheduler，具体可参考 <https://issues.apache.org/jira/browse/YARN-137>。

```
void reinitialize(Configuration conf, RMContext rmContext) throws IOException;
}
```

其中，接口 YarnScheduler 的定义如下：

```
public interface YarnScheduler extends EventHandler<SchedulerEvent> {
    // 获取一个队列的基本信息，queueName 为队列名称，includeChildQueues 表示是否包
    // 含子队列，recursive 表示是否递归返回其子队列的信息
    public QueueInfo getQueueInfo(String queueName, boolean includeChildQueues,
        boolean recursive) throws IOException;

    // 返回当前用户的队列 ACL 权限
    public List<QueueUserACLInfo> getQueueUserAclInfo();

    // 返回调度器最少可分配的资源
    public Resource getMinimumResourceCapability();

    // 返回调度器最多可分配的资源
    public Resource getMaximumResourceCapability();

    // 返回当前集群中可用节点的总数
    public int getNumClusterNodes();

    // ApplicationMaster 和资源调度器之间最主要的 API，ApplicationMaster 通过该 API 更新
    // 资源需求和待释放的 Container 列表，其中 appAttemptId 为应用程序实例 ID，ask 为新
    // 请求资源的描述，release 为待释放 Container 列表，blacklistAdditions 为待加入黑名单
    // 的节点列表，blacklistRemovals 为待移除黑名单的节点列表
    Allocation
    allocate(ApplicationAttemptId appAttemptId,
        List<ResourceRequest> ask,
        List<ContainerId> release,
        List<String> blacklistAdditions,
        List<String> blacklistRemovals);

    // 获取节点资源使用情况报告
    public SchedulerNodeReport getNodeReport(NodeId nodeId);

    // 获取运行实例 appAttemptId 的 SchedulerAppReport 对象
    SchedulerAppReport getSchedulerAppInfo(ApplicationAttemptId appAttemptId);

    // 获取 root 队列的 Metric 信息
    QueueMetrics getRootQueueMetrics();
}
```

接口 Recoverable 的定义如下：

```
public interface Recoverable {
    // ResourceManager 重启后将调用该函数恢复调度器内部的信息
    public void recover(RMState state) throws Exception;
}
```

(2) 作为一个事件处理器

YARN 的资源管理器实际上是一个事件处理器，它需要处理来自外部的 6 种 Scheduler-EventType 类型的事件，并根据事件的具体含义进行相应的处理，具体如图 6-3 所示。这 6 种事件含义如下。

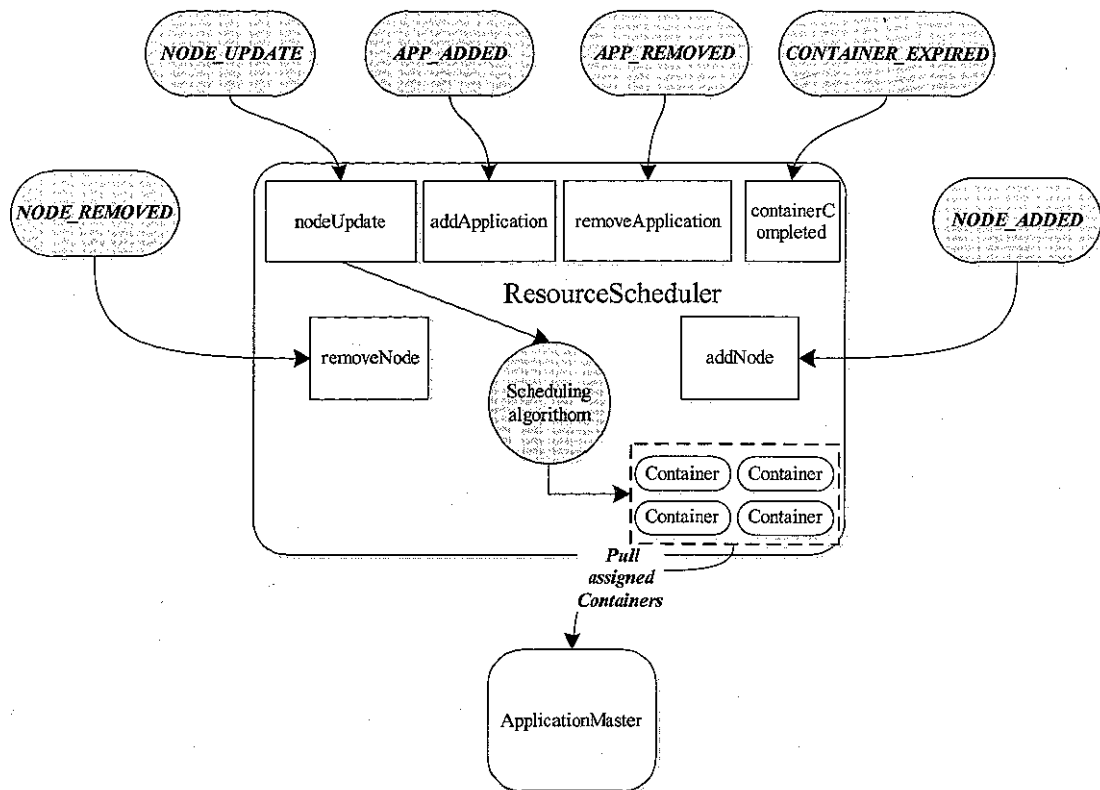


图 6-3 资源调度器基本架构

- **NODE_REMOVED**：表示集群中移除了一个计算节点（可能是节点故障或者管理员主动移除），资源调度器收到该事件时需要从可分配资源总量中移除相应的资源量。
- **NODE_ADDED**：表示集群中增加了一个计算节点，资源调度器收到该事件时需要将新增的资源量添加到可分配资源总量中。
- **APPLICATION_ADDED**：表示 ResourceManager 收到一个新的 Application。通常而言，资源管理器需要为每个 Application 维护一个独立的数据结构，以便于统一管理和资源分配。资源管理器需将该 Application 添加到相应的数据结构中。
- **APPLICATION_REMOVED**：表示一个 Application 运行结束（可能成功或者失败），资源管理器需将该 Application 从相应的数据结构中清除。
- **CONTAINER_EXPIRED**：当资源调度器将一个 Container 分配给某个 ApplicationMaster 后，如果该 ApplicationMaster 在一定时间间隔内没有使用该 Container，则资

源调度器会对该 Container 进行（回收后）再分配。

- **NODE_UPDATE**: ResourceManager 收到 NodeManager 通过心跳机制汇报的信息后，会触发一个 NODE_UPDATE 事件，由于此时可能有新的 Container 得到释放，因此该事件会触发资源分配。也就是说，该事件是 6 个事件中最重要的事件，它会触发资源调度器最核心的资源分配机制。

6.3.2 资源表示模型

当前 YARN 支持内存和 CPU 两种资源类型的管理和分配。同 MRv1 一样，YARN 采用了动态资源分配机制。NodeManager 启动时会向 ResourceManager 注册，注册信息中包含该节点可分配的 CPU 和内存总量，这两个值均可通过配置选项设置，具体如下：

- **yarn.nodemanager.resource.memory-mb**。可分配的物理内存总量，默认是 $8MB \times 1024$ ，即 8GB。
- **yarn.nodemanager.vmem-pmem-ratio**。任务使用单位物理内存量对应最多可使用的虚拟内存量，默认值是 2.1，表示每使用 1MB 的物理内存，最多可以使用 2.1MB 的虚拟内存总量。
- **yarn.nodemanager.resource.cpu-vcores**。可分配的虚拟 CPU 个数，默认是 8。为了更细粒度地划分 CPU 资源和考虑到 CPU 性能异构性，YARN 允许管理员根据实际需要和 CPU 性能将每个物理 CPU 划分成若干个虚拟 CPU，而管理员可为每个节点单独配置可用的虚拟 CPU 个数，且用户提交应用程序时，也可指定每个任务需要的虚拟 CPU 个数。比如 node1 节点上有 8 个 CPU，node2 上有 16 个 CPU，且 node1 CPU 性能是 node2 的 2 倍，那么可为这两个节点配置相同数目的虚拟 CPU 个数，比如均为 32。由于用户设置虚拟 CPU 个数必须是整数，每个任务至少使用 node2 的半个 CPU（不能更少了）。

除了 CPU 和内存两种资源，服务器还有很多其他资源，比如磁盘容量、网络 and 磁盘 I/O 等，YARN 可能在将来支持这些资源的调度。

为了更友好地为应用程序分配资源，YARN 内部包含了一些调度语义，这决定了 YARN 作为一个资源管理系统可给用户带来的服务承诺。当前 YARN 的调度器能够表达的调度语义是有限的，这也决定了它对某些应用是友好的，但对于其他应用则会带来性能问题。当前 YARN 支持的调度语义和不支持的调度语义可总结如下。

（1）支持的调度语义

当前 YARN 支持的调度语义包括：

- 请求某个特定节点上的特定资源量。比如，请求节点 nodeX 上 5 个这样的 Container：虚拟 CPU 个数为 2，内存量为 2GB。
- 请求某个特定机架上的特定资源量。比如，请求机架 rackX 上 3 个这样的 Container：虚拟 CPU 个数为 4，内存量为 3GB。
- 将某些节点加入（或移除）黑名单，不再为自己分配这些节点上的资源。比如，

ApplicationMaster 发现节点 nodeX 和 nodeY 失败的任务数目过多, 可请求将这两个节点加入黑名单, 从而不再收到这两个节点上的资源, 过一段时间后, 可请求将 nodeX 移除黑名单, 从而可再次使用该节点上的资源。

- 请求归还某些资源。比如, ApplicationMaster 已获取的来自节点 nodeX 上的 2 个 Container 暂时不用了, 可将之归还给集群, 这样这些资源可再次分配给其他应用程序。

(2) 不支持的调度语义

当前不支持的调度语义包括:

- 请求任意节点上的特定资源量。比如, 请求任意节点上 5 个这样的 Container: 虚拟 CPU 个数为 3, 内存量为 1GB。
- 请求任意机架上的特定资源量。比如, 请求同一个机架上的 (具体哪个机架并不关心, 但是必须来自同一个机架) 3 个这样的 Container: 虚拟 CPU 个数为 1, 内存量为 6GB。
- 请求一组或几组符合某种特质的资源。比如, 请求来自两个机架上的 4 个 Container, 其中, 一个机架上 2 个这样的 Container: 虚拟 CPU 个数为 2, 内存量为 2GB; 另一个机架上 2 个这样的资源: 虚拟 CPU 个数为 2, 内存量为 3GB。如果目前集群没有这样的资源, 需要从其他应用程序处抢占资源。
- 超细粒度资源。比如 CPU 性能要求、绑定 CPU 等。
- 动态调整 Container 资源, 应允许根据需要动态调整 Container 资源量 (对于长作业尤其有用)。

6.3.3 资源调度模型

本小节我们介绍 YARN 采用的资源调度模型。

(1) 双层资源调度模型

YARN 采用了**双层资源调度模型**: 在第一层中, ResourceManager 中的资源调度器将资源分配给各个 ApplicationMaster; 在第二层中, ApplicationMaster 再进一步将资源分配给它内部的各个任务。本章所介绍的资源调度器主要关注的是第一层的调度问题, 至于第二层的调度策略, 完全由用户应用程序自己决定。

YARN 的资源分配过程是异步的, 也就是说, 资源调度器将资源分配给一个应用程序后, 它不会立刻 push 给对应的 ApplicationMaster, 而是暂时放到一个缓冲区中, 等待 ApplicationMaster 通过周期性的心跳主动来取。也就是说, YARN 采用了 pull-based 通信模型, 而不是 push-based 模型, 这与 MRv1 是一致的。

在 YARN 中, 资源分配过程可概括为以下 7 个步骤:

步骤 1 NodeManager 通过周期性心跳汇报节点信息。

步骤 2 ResourceManager 为 NodeManager 返回一个心跳应答, 包括需释放的 Container 列表等信息。

步骤 3 ResourceManager 收到来自 NodeManager 的信息后，会触发一个 NODE_UPDATE 事件。

步骤 4 ResourceScheduler 收到 NODE_UPDATE 事件后，会按照一定的策略将该节点上的资源（步骤 2 中有释放的资源）分配各应用程序，并将分配结果放到一个内存数据结构中。

步骤 5 应用程序的 ApplicationMaster 向 ResourceManager 发送周期性的心跳，以领取最新分配的 Container。

步骤 6 ResourceManager 收到来自 ApplicationMaster 心跳信息后，为它分配的 container 将以心跳应答的形式返回给 ApplicationMaster。

步骤 7 ApplicationMaster 收到新分配的 container 列表后，会将这些 Container 进一步分配给它内部的各个任务。

资源调度器资源分配示意如图 6-4 所示。

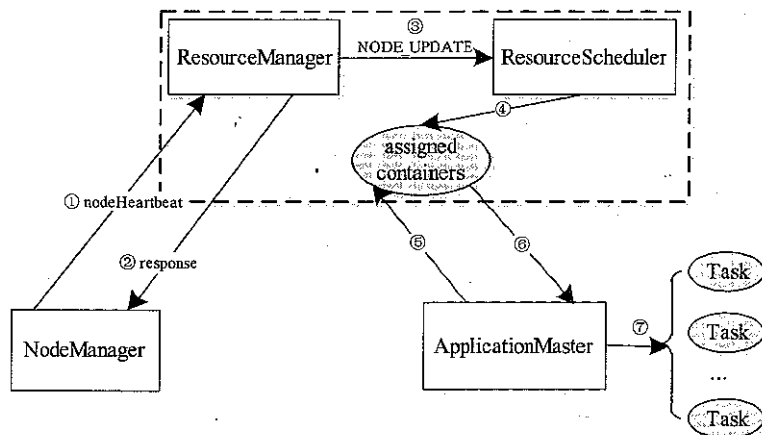


图 6-4 资源调度器资源分配流程

资源调度器关注的是步骤 4 中采用的策略，即如何将节点上空闲的资源分配给各个应用程序，至于步骤 7 中的策略，则完全由用户应用程序自己决定。

（2）资源保证机制

在分布式计算中，资源调度器需选择合适的资源保证这样的机制：当应用程序申请的资源暂时无法保证时，是优先为应用程序预留一个节点上的资源直到累计释放的空闲资源满足应用程序需求（称为“增量资源分配”，即 Incremental placement），还是暂时放弃当前资源直到出现一个节点剩余资源一次性满足应用程序需求（称为“一次性资源分配”，all-or-nothing）。这两种机制均存在优缺点，对于增量资源分配而言，资源预留会导致资源浪费，降低集群资源利用率；而一次性资源分配则会产生饿死现象，即应用程序可能永远等不到满足资源需求的节点出现。

YARN 采用了增量资源分配机制，尽管这种机制会造成浪费，但不会出现饿死现象

(假设应用程序不会永久占用某个资源,它会在一定时间内释放占用的资源)。

(3) 资源分配算法

为了支持多维资源调度, YARN 资源调度器采用了主资源公平调度算法 (Dominant Resource Fairness, DRF)^①, 该算法扩展了最大最小公平 (max-min fairness) 算法^②, 使其能够支持多维资源的调度。由于 DRF 被证明非常适合应用于多资源和复杂需求的环境中, 因此被越来越多的系统采用, 包括 Apache Mesos。

在 DRF 算法中, 将所需份额 (资源比例) 最大的资源称为主资源, 而 DRF 的基本设计思想则是将最大最小公平算法应用于主资源上, 进而将多维资源调度问题转化为单资源调度问题, 即 DRF 总是最大化所有主资源中最小的, 其算法伪代码如下:

```
function void DRFScheduler()
    R ← <r1, ..., rm>; // m 种资源对应的容量
    C ← <c1, ..., cm>; // 已用掉的资源, 初始值为 0
    si (i = 1..n); // 用户 (或者框架) i 的主资源所需份额, 初始化为 0
    Ui ← <ui,1, ..., ui,m> (i = 1..n) // 分配给用户 i 的资源, 初始化为 0
    挑选出主资源所需份额 si 最小的用户 i;
    Di ← { 用户 i 的下一个任务需要的资源量 };
    if C + Di ≤ R then
        // 将资源分配给用户 i
        C ← C + Di; // 更新 C
        Ui ← Ui + Di; // 更新 U
        si = maxj=1..m {ui,j/rj};
    else
        return; // 资源全部用完
    end if
end function
```

下面我们看一个实例。假设系统中共有 9 个 CPU 和 18 GB RAM, 有两个用户 (或者框架) 分别运行了两种任务, 需要的资源量分别为 <1 CPU, 4 GB> 和 <3 CPU, 1 GB>。对于用户 A, 每个任务要消耗总 CPU 的 1/9 (份额) 和总内存的 2/9, 因而 A 的主资源为内存; 对于用户 B, 每个任务要消耗总 CPU 的 1/3 和总内存的 1/18, 因而 B 的主资源为 CPU。DRF 将最大化所有用户的主资源, 具体分配过程如表 6-1 所示。最终, A 获取的资源量为 <3 CPU, 12 GB>, 可运行 3 个任务; 而 B 获取的资源量为 <6 CPU, 2GB>, 可运行 2 个任务。

表 6-1 DRF 算法的调度序列

调度 序列	User A		User B		CPU 使用量	RAM 使用量
	资源份额	主资源份额	资源份额	主资源份额		
User B	<0,0>	0	<3/9,1/18>	1/3	3/9	1/18
User A	<1/9,4/18>	2/9	<3/9,1/18>	1/3	4/9	5/18
User A	<2/9,8/18>	4/9	<3/9,1/18>	1/3	5/9	9/18

① Dominant Resource Fairness: Fair Allocation of Multiple Resources Types. A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, NSDI 2011, March 2011.

② Max-Min Fairness (Wikipedia): http://en.wikipedia.org/wiki/Max-min_fairness.

(续)

调度 序列	User A		User B		CPU 使用量	RAM 使用量
	资源份额	主资源份额	资源份额	主资源份额		
User B	<2/9,8/18>	4/9	<6/9,2/18>	2/3	8/9	10/18
User A	<3/9,12/18>	2/3	<6/9,2/18>	2/3	1	14/18

6.3.4 资源抢占模型

在资源调度器中，每个队列可设置一个最小资源量和最大资源量，其中，最小资源量是资源紧缺情况下每个队列需保证的资源量，而最大资源量则是极端情况下队列也不能超过的资源使用量。资源抢占发生的原因则完全是由于“最小资源量”这一概念。通常而言，为了提高资源利用率，资源调度器（包括 Capacity Scheduler 和 Fair Scheduler）会将负载较轻的队列的资源暂时分配给负载重的队列（即最小资源量并不是硬资源保证，当队列不需要任何资源时，并不会满足它的最小资源量，而是暂时将空闲资源分配给其他需要资源的队列），仅当负载较轻队列突然收到新提交的应用程序时，调度器才进一步将本属于该队列的资源分配给它。但由于此时资源可能正被其他队列使用，因此调度器必须等待其他队列释放资源后，才能将这些资源“物归原主”，这通常需要一段不确定的等待时间。为了防止应用程序等待时间过长，调度器等待一段时间后若发现资源并未得到释放，则进行资源抢占。

下面我们举例说明。如图 6-5 所示，整个集群资源总量为 100（为了简便，没有区分 CPU 或者内存），且被分为三个队列，分别是 QueueA、QueueB 和 QueueC。它们的最小资源量和最大资源量（由管理员配置）分别是 (10,15)、(20,35) 和 (60,65)。某一时刻，它们尚需的资源量和正在使用的资源量分别是 (0,5)、(10,30) 和 (60,65)，即队列 QueueA 负载较轻，部分资源暂时不会使用，它将不会使用的 5 个资源共享给了其他两个队列（QueueB 和 QueueC 分别得到 2 个和 3 个），而队列 QueueB 和 QueueC 除了使用来自队列 QueueA 的资源外，还使用了整个系统共享的 10 个资源（ $100-10-20-60=10$ ）。某一时刻，队列 QueueA 突然增加了一批应用程序，此时共需要 20 个资源，则资源调度器需要从 QueueB 和 QueueC 中抢占 5 个本该属于 QueueA 的资源。需要注意的是，为了避免资源浪费，资源调度器通常会等待一段时间后会强制回收资源，而在这段等待时间内，QueueB 和 QueueC 可能已经释放了本该属于 QueueA 的资源。

仅当启用的调度器实现了 PreemptableResourceScheduler 接口，且参数 `yarn.resourcemanager.scheduler.monitor.enable` 的值被置为 `true`（默认值为 `false`）时，ResourceManager 才启用资源抢占功能。资源抢占是通过第三方策略触发的，这些策略通常被实现成一些插拔式的组件类（实现 SchedulingEditPolicy 接口），并通过参数 `yarn.resourcemanager.scheduler.monitor.policies` 指定（默认情况下，YARN 提供了默认实现 ProportionalCapacityPreemptionPolicy）。ResourceManager 将依次遍历这些策略类，并由监控类 SchedulingMonitor 进一步封装它

们, SchedulingMonitor 将周期性调用策略类中的 editSchedule() 函数, 以决定抢占哪些 Container 的资源。

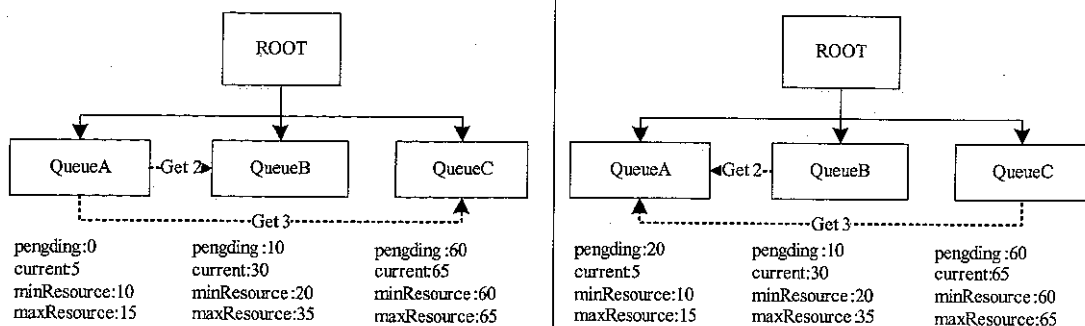


图 6-5 资源抢占发生时机

如图 6-6 所示, 在 YARN 中, 资源抢占整个过程可概括为如下步骤:

步骤 1 SchedulingEditPolicy 探测到需要抢占的资源, 将需要抢占的资源通过事件 DROP_RESERVATION 和 PREEMPT_CONTAINER 发送给 ResourceManager。

步骤 2 ResourceManager 调用 ResourceScheduler 的 dropContainerReservation 和 preemptContainer 函数, 标注待抢占的 Container。

步骤 3 ResourceManager 收到来自 ApplicationMaster 的心跳信息, 并通过心跳应答将待释放的资源总量和待抢占 Container 列表发返回给它。ApplicationMaster 收到该列表后, 可选择如下操作:

- 1) 杀死这些 Container。
- 2) 选择并杀死其他 Container 以凑够总量。
- 3) 不做任务处理, 过段时间可能有 Container 自行释放资源或者由 ResourceManager 杀死 Container。

步骤 4 SchedulingEditPolicy 探测到一段时间内, ApplicationMaster 未自行杀死约定的 Container, 则将这些 Container 封装到 KILL_CONTAINER 事件中发送给 ResourceManager。

步骤 5 ResourceManager 调用 ResourceScheduler 的 killContainer 函数, 而 ResourceScheduler 则标注这些待杀死的 Container。

步骤 6 ResourceManager 收到来自 NodeManager 的心跳信息, 并通过心跳应答将待杀死的 Container 列表返回给它, NodeManager 收到该列表后, 将这些 Container 杀死, 并通过心跳告知 ResourceManager。

步骤 7 ResourceManager 收到来自 ApplicationMaster 的心跳信息, 并通过心跳应答将已经杀死的 Container 列表发送给它 (可能 ApplicationMaster 早已通过内部通信机制知道了这些被杀死的 Container)。

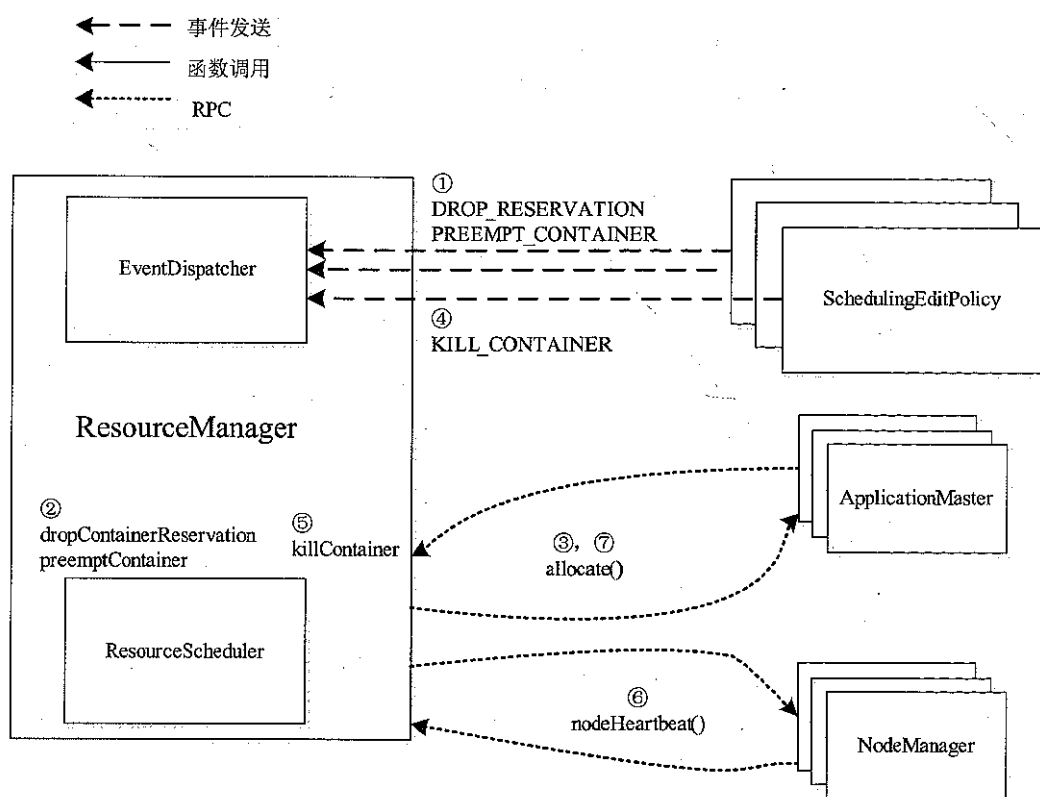


图 6-6 资源抢占流程

关于资源抢占的实现，通常涉及以下几个需认真考虑的问题：

□ 如何决定是否抢占某个队列的资源？

□ 如何使资源抢占代价最小？

接下来详细进行讨论。

问题 1：如何决定是否抢占某个队列的资源？

在 YARN 中，队列是按照树形结构组织的，一个队列当前实际可以使用的资源量 R 取决于最小资源量 A （由管理员配置）、队列资源需求量 B （处于等待或者运行状态的应用程序尚需的资源总量）和同父兄弟队列的空闲资源量 C （多余资源可共享给其他队列），这意味着 R 在不同时间点的取值是不同的，可以按照递归算法求出 $R=F(A, B, C)$ ，这样，如果一个队列当前正在使用资源量 $U>R$ ，则需从该队列中抢占 $(U-R)$ 资源。

接下来以默认实现 ProportionalCapacityPreemptionPolicy 为例，说明如何计算每个队列抢占的资源量。

先来计算当前每个队列真正可用资源量 $ideaAssigned$ 。由于队列是按照树形结构组织的，因此需要采用递归算法求解，算法伪代码如下：

```

// 初始调用时, root.idealAssigned 值为集群资源总量 clusterResource
function void recursivelyComputeIdealAssignment(TempQueue root)
    if root.getChildren() != null &&
        root.getChildren().size() > 0 then
        computeIdealResourceDistribution(root.getChildren(),
            root.idealAssigned);
        for t ← root.getChildren() do
            leafs.addAll(recursivelyComputeIdealAssignment(t));
        end for
    end if
end function

// TempQueue {
//   current/pending: 当前正在使用 / 尚需的资源量
//   idealAssigned: 真正可以使用的资源量, 初始值为 0
//   minResource: 最小资源量
//}

void computeIdealResourceDistribution(List<TempQueue> queues,
    Resource tot_guarant)
    unassigned ← tot_guarant
    while !qAlloc.isEmpty() && unassigned > 0
        wQassigned ← 0
        // 归一化最小资源 normalizedGuarantee=minResource/sum{minResource}
        normalizedGuaranteed(queues)
        for q ← qAlloc do
            wQavail ← unassigned × q.normalizedGuarantee
            accepted ← min(wQavail, q.current + q.pending - q.idealAssigned)
            wQidle ← wQavail - accepted
            q.idealAssigned ← q.idealAssigned + accepted
            wQdone ← wQavail - wQidle
            if wQdone == 0 do
                q.remove()
            end if
            wQassigned ← wQassigned + wQdone
        end for
        unassigned ← unassigned - wQassigned
    end while
end function

```

为了方便读者理解以上算法, 下面给出一个实例。

如图 6-7 所示, 整个集群资源总量为 100 (为了简便, 没有区分 CPU 或者内存), 且被划分为三个队列, 分别是 QueueA、QueueB 和 QueueC, 它们的最小资源量 (由管理员配置) 分别是 24、16 和 40, 在 t_1 时刻, 它们尚需的资源量和正在使用的资源量分别是 (0,20)、(0,20) 和 (80,60), 则按照 computeIdealResourceDistribution 函数计算, 第一轮可得到如表 6-2 所示的结果。

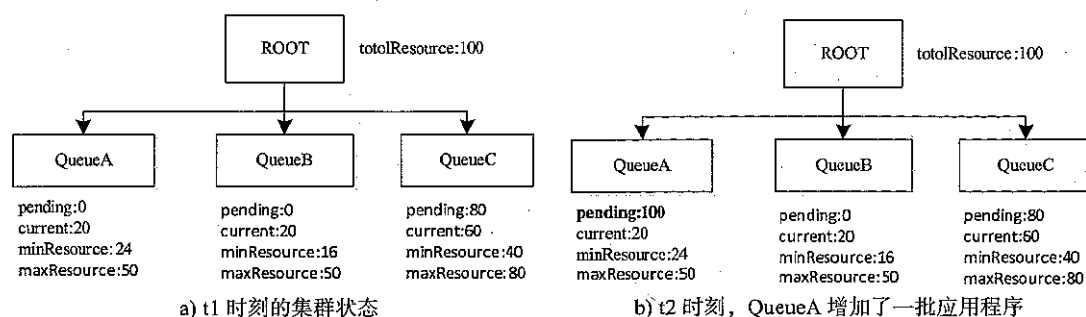


图 6-7 资源抢占资源量计算方法

表 6-2 t1 时刻第一轮计算结果

	normalizedGuarantee	wQavail	accepted	wQidle	idealAssigned	wQdone
QueueA	$24/(24+16+40)=0.3$	$100*0.3=30$	20	10	20	20
QueueB	$16/(24+16+40)=0.2$	$100*0.2=20$	20	0	20	20
QueueC	$40/(24+16+40)=0.5$	$100*0.5=50$	50	0	50	50

经过第一轮计算、wQassigned=90, unassigned=10。

第二轮计算结果如表 6-3 所示。

表 6-3 t1 时刻第二轮计算结果

	normalizedGuarantee	wQavail	accepted	wQidle	idealAssigned	wQdone
QueueA	$24/(24+16+40)=0.3$	$10*0.3=3$	0	3	20	0
QueueB	$16/(24+16+40)=0.2$	$10*0.2=2$	0	2	20	0
QueueC	$40/(24+16+40)=0.5$	$10*0.5=5$	5	0	55	5

经过第二轮计算：wQassigned=95, unassigned=5。由于 QueueA 和 QueueB 的 wQdone 为 0, 不再进入下一轮计算。

第三轮计算结果如表 6-4 所示。

表 6-4 t1 时刻第三轮计算结果

	normalizedGuarantee	wQavail	accepted	wQidle	idealAssigned	wQdone
QueueC	$40/(40)=1.0$	$5*1.0=5$	5	0	60	5

经过这一轮计算：unassigned=0, 所有资源分配完成, 退出计算。这样三个队列真正可以使用的资源为 20、20 和 60, 很显然, 它们当前正在使用的资源均未超过这三个值, 所以不会发生任何资源抢占。

假设 t2 时刻, 用户向 QueueA 提交了一批应用程序, 使得资源需求量由 0 变为 100, 其他状态不变。按照 computeIdealResourceDistribution 函数计算, 第一轮可得到的结果如

表 6-5 所示。

表 6-5 t2 时刻第一轮计算结果

	normalizedGuarantee	wQavail	accepted	wQidle	idealAssigned	wQdone
QueueA	$24/(24+16+40)=0.3$	$100*0.3=30$	30	0	30	30
QueueB	$16/(24+16+40)=0.2$	$100*0.2=20$	20	0	20	20
QueueC	$40/(24+16+40)=0.5$	$100*0.5=50$	50	0	50	50

经过第一轮计算： $wQassigned=100$ ， $unassigned=0$ ，所有资源分配完成，退出计算。由于 QueueC 当前正在使用的资源量为 60，超过了应得的资源量 50，需要从它里面抢占 10 个资源。

问题 2：如何使资源抢占代价最小？

资源抢占是通过杀死正在使用的资源 Container 实现的，由于这些 Container 已经处于运行状态，直接杀死 Container 会导致已经完成的计算白白浪费。为了尽可能避免资源浪费，YARN 优先选择优先级低的 Container 作为资源抢占对象，且不会立刻杀死 Container，而是将释放资源的任务留给应用程序自己：ResourceManager 将待杀死的 Container 列表发送给对应的 ApplicationMaster，以期望它采取一定的机制自行释放这些 Container 占用的资源，比如先进行一些状态保存工作后，再将对应的 Container 杀死，以避免计算浪费，如果一段时间后，ApplicationMaster 尚未主动杀死这些 Container，则 ResourceManager 再强制杀死这些 Container。

本节从调度器基本架构、资源表示模型、资源调度模型和资源抢占模型等方面介绍了 YARN 资源调度器，从这些介绍中可以看出，在 MRv1 和 YARN 中，尽管 Fair Scheduler 和 Capacity Scheduler 均是插拔式的，且实现原理基本一致，但由于 YARN 采用了事件驱动的编程模型、独特的资源调度模型和抢占模型，它的资源调度器设计更加复杂，它要求用户不仅要了解基本的编程接口，还要理解 ResourceManager 与资源调度器之间基于事件的交互逻辑。

6.4 YARN 层级队列管理机制

在学习 Capacity Scheduler 和 Fair scheduler 之前，我们先要了解 Hadoop 的用户和资源管理机制，这是任何 Hadoop 可插拔资源调度器的基础。

6.4.1 层级队列管理机制

在 Hadoop 0.20.x 版本或者更早的版本中，Hadoop 采用了平级队列组织方式。在这种组织方式中，管理员将用户和资源分到若干个扁平队列中，在每个队列中，可指定一个或几个队列管理员管理这些用户和资源，比如杀死任意用户的应用程序、修改任意用户应用程序的优先级等。

随着 Hadoop 应用越来越广泛，扁平化的队列组织方式已不能满足实际需求，从而出现了层级队列组织方式。典型的应用场景如下：在一个 Hadoop 集群中，管理员将所有计算资源划分给了若干个队列，每个队列对应了一个“组织”，其中有一个“Org1”组织，它分到了 60% 的资源，它内部包含 3 中类型的应用：

- 产品线应用；
- 实验性应用，分属于 Proj1、Proj2 和 Proj3 三个不同的项目；
- 其他类型应用。

假设 Org1 管理员想更有效地控制这 60% 资源，比如将大部分资源分配给产品线应用程序的同时，能够让实验性应用程序和其他类型应用程序有最少资源保证。考虑到产品线应用程序提交频率很低，当有产品线应用程序提交时，必须第一时间得到资源，剩下的资源才给其他类型的应用程序，然而，一旦产品线应用程序运行结束，实验性应用程序和其他类型应用程序必须马上获取未使用的资源，一个可能的配置方式如下：

```
grid {
  Org1 min=60% {
    priority min=90% {
      production min=82%
      proj1 min=6% max=10%
      proj2 min=6%
      proj3 min=6%
    }
    miscellaneous min=10%
  }
  Org2 min=40%
}
```

这就引出来层级队列组织方式，该队列组织方式具有以下特点：

□ 子队列。具体包括如下两点：

- 队列可以嵌套，每个队列均可以包含子队列。
- 用户只能将应用程序提交到最底层的队列，即叶子队列。

□ 最少容量。具体包括如下四点：

- 每个子队列均有一个“最少容量比”属性，表示可以使用父队列的容量的百分比。
- 调度器总是优先选择当前资源使用率最低的队列，并为之分配资源。比如同级的两个队列 Q1 和 Q2，它们的最少容量均为 30，而 Q1 已使用 10，Q2 已使用 12，则调度器会优先将资源分配给 Q1。
- 最少容量不是“总会保证的最低容量”，也就是说，如果一个队列的最少容量为 20，而该队列中所有队列仅使用了 5，那么剩下的 15 可能会分配给其他需要的队列。
- 最少容量的值为不小于 0 的数，但也不能大于“最大容量”。

□ 最大容量。具体包括如下两点：

- 为了防止一个队列超量使用资源，可以为队列设置一个最大容量，这是一个资源

使用上限，任何时刻使用的资源总量都不能超过该值；

- 默认情况下队列的最大容量是无限大，这意味着，当一个队列只分配了 20% 的资源，所有其他队列没有应用程序时，该队列可能使用 100% 的资源，当其他队列有应用程序提交时，再逐步归还。

Hadoop 队列管理机制由用户权限管理和系统资源管理两部分组成，下面依次进行介绍。

□**用户权限管理**：Hadoop 的用户管理模块构建在操作系统用户管理之上，增加了“队列”这一用户组织单元，并通过队列建立了操作系统用户和用户组之间的映射关系。管理员可配置每个叶子队列对应的操作系统用户和用户组（需要注意的是，Hadoop 允许一个操作系统用户或者用户组可对应一个或者多个队列），也可以配置每个队列的管理员，他可以杀死该队列中任何应用程序，改变任何应用程序的优先级等（默认情况下每个用户只能管理自己的应用程序）。

□**系统资源管理**：YARN 资源管理和调度均由调度器完成，管理员可在调度器中设置每个队列的资源容量，每个用户可用资源量等信息，而调度器则按照这些资源约束对应用程序进行调度。

通常而言，不同的调度器对资源管理的方式是不同的，我们将在 6.5 节和 6.6 节中分别介绍 Capacity Scheduler 和 Fair Scheduler 两个调度器的工作原理。

6.4.2 队列命名规则

为了防止队列名称冲突和便于识别队列，YARN 采用了自顶向下的路径命名队列，其中，父队列与子队列名称采用“.”拼接。图 6-8 是一个公司内部的组织架构，映射到 YARN 中，所有队列命名如下：

- ROOT
- ROOT.A
- ROOT.A.A1
- ROOT.A.A2
- ROOT.B
- ROOT.B.B1
- ROOT.C
- ROOT.C.C1
- ROOT.C.C
- ROOT.C1.C11
- ROOT.C1.C12

图 6-8 中有两个同名的队列 C，通过引入队列路径后，可以很容易区分出这两个队列，一个是“ROOT.C”，另外一个为“ROOT.C.C”。

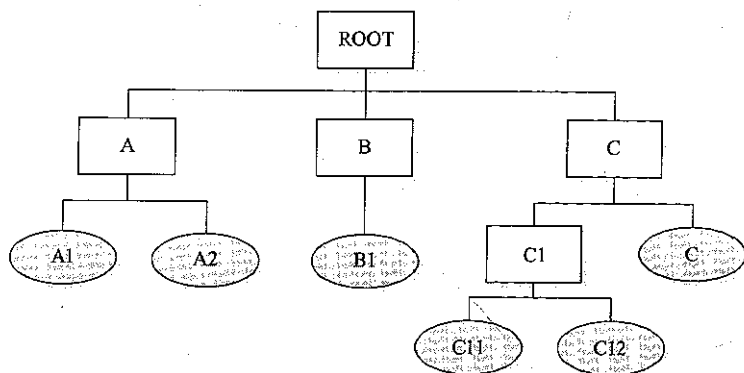


图 6-8 层次队列组织方式

6.5 Capacity Scheduler

Capacity Scheduler^①是 Yahoo! 开发的多用户调度器，它以队列为单位划分资源，每个队列可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用。而当一个队列的资源有剩余时，可暂时将剩余资源共享给其他队列。总之，Capacity Scheduler 主要有以下几个特点：

- 容量保证。管理员可为每个队列设置资源最低保证和资源使用上限，而所有提交到该队列的应用程序共享这些资源。
- 灵活性。如果一个队列中的资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的应用程序提交，则其他队列释放的资源会归还给该队列。相比于 HOD 调度器，这种资源灵活分配的方式可明显提高资源利用率。
- 多重租赁。支持多用户共享集群和多应用程序同时运行。为防止单个应用程序、用户或者队列独占集群中的资源，管理员可为之增加多重约束（比如单个应用程序同时运行的任务数等）。
- 安全保证。每个队列有严格的 ACL 列表规定它的访问用户，每个用户可指定哪些用户允许查看自己应用程序的运行状态或者控制应用程序（比如杀死应用程序）。此外，管理员可指定队列管理员和集群系统管理员。
- 动态更新配置文件。管理员可根据需要动态修改各种配置参数，以实现在线集群管理。

6.5.1 Capacity Scheduler 的功能

Capacity Scheduler 是一个多用户调度器，它设计了多层级别的资源限制条件以更好地让多用户共享一个 Hadoop 集群，比如队列资源限制、用户资源限制、用户应用程序数

① http://hadoop.apache.org/docs/stable/capacity_scheduler.html。

目限制等。为了能够更详尽地了解 Capacity Scheduler 的功能，我们从它的配置文件讲起。Capacity Scheduler 有自己的配置文件，即存放在 conf 目录下的 capacity-scheduler.xml。

在 Capacity Scheduler 的配置文件中，队列 queueX 的参数 Y 的配置名称为 yarn.scheduler.capacity.queueX.Y，为了简单起见，我们记为 Y。每个队列可以配置的参数如下。

(1) 资源分配相关参数

资源分配的相关参数具体如下。

- **capacity**：队列的资源容量（百分比）。当系统非常繁忙时，应保证每个队列的容量得到满足，而如果每个队列应用程序较少，可将剩余资源共享给其他队列。注意，所有队列的容量之和应小于 100。
- **maximum-capacity**：队列的资源使用上限（百分比）。由于存在资源共享，因此一个队列使用的资源量可能超过其容量，而最多使用资源量可通过该参数限制。
- **minimum-user-limit-percent**：每个用户最低资源保障（百分比）。任何时刻，一个队列中每个用户可使用的资源量均有一定的限制。当一个队列中同时运行多个用户的应用程序时，每个用户的使用资源量在一个最小值和最大值之间浮动，其中，最小值取决于正在运行的应用程序数目，而最大值则由 minimum-user-limit-percent 决定。比如，假设 minimum-user-limit-percent 为 25。当两个用户向该队列提交应用程序时，每个用户可使用资源量不能超过 50%，如果三个用户提交应用程序，则每个用户可使用资源量不能超过 33%，如果四个或者更多用户提交应用程序，则每个用户可用资源量不能超过 25%。
- **user-limit-factor**：每个用户最多可使用的资源量（百分比）。比如，假设该值为 30，则任何时刻，每个用户使用的资源量不能超过该队列容量的 30%。

(2) 限制应用程序数目相关参数

限制应用程序数目的相关参数如下：

- **maximum-applications**：集群或者队列中同时处于等待和运行状态的应用程序数目上限，这是一个强限制项，一旦集群中应用程序数目超过该上限，后续提交的应用程序将被拒绝。默认值为 10000。Hadoop 允许从集群和队列两个方面限定该值，其中，集群的总体数目上限可通过参数 yarn.scheduler.capacity.maximum-applications 设置，默认为 1000，而单个队列可通过参数 yarn.scheduler.capacity.<queue-path>.maximum-applications 设置适合自己的值。
- **maximum-am-resource-percent**：集群中用于运行应用程序 ApplicationMaster 的资源比例上限，该参数通常用于限制处于活动状态的应用程序数目。该参数为浮点型数据，默认是 0.1，表示 10%。所有队列的 ApplicationMaster 资源比例上限可通过参数 yarn.scheduler.capacity.maximum-am-resource-percent 设置（可看做默认值），而单个队列可通过参数 yarn.scheduler.capacity.<queue-path>.maximum-am-resource-percent 设置适合自己的值。

(3) 队列访问和权限控制参数

队列访问和权限控制参数如下：

- **state**：队列状态，可以为 STOPPED 或者 RUNNING。如果一个队列处于 STOPPED 状态，用户不可以将应用程序提交到该队列或者它的子队列中。类似的，如果 root 队列处于 STOPPED 状态，则用户不可以向集群中提交应用程序，但正在运行的应用程序仍可以正常运行结束，以便队列可以优雅地退出。
- **acl_submit_applications**：限定哪些用户 / 用户组可向给定队列中提交应用程序。需要注意的是，该属性具有继承性，即如果一个用户可以向某个队列中提交应用程序，则它可以向它的所有子队列中提交应用程序。
- **acl_administer_queue**：为队列指定一个管理员，该管理员可控制该队列的所有应用程序，比如杀死任意一个应用程序等。同样，该属性具有继承性，如果一个用户可以向某个队列中提交应用程序，则它可以向它的所有子队列中提交应用程序。

一个配置文件实例如下：

```
<configuration>
  <property>
    <name>yarn.scheduler.capacity.maximum-applications</name>
    <value>10000</value>
    <description>最多可同时处于等待和运行状态的应用程序数目</description>
  </property>

  <property>
    <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>
    <value>0.1</value>
    <description>集群中可用于运行 Application Master 的资源比例上限，这通常用于限制并发运
行的应用程序数目</description>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>default</value>
    <description>root 队列的所有子队列，该实例中只有一个</description>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.default.capacity</name>
    <value>100</value>
    <description>default 队列的资源容量</description>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.default.user-limit-factor</name>
    <value>1</value>
    <description>
      每个用户可使用的资源限制
    </description>
  </property>
</configuration>
```

```

</property>

<property>
  <name>yarn.scheduler.capacity.root.default.maximum-capacity</name>
  <value>100</value>
  <description>
    default 队列可使用的资源上限
  </description>
</property>

<property>
  <name>yarn.scheduler.capacity.root.default.state</name>
  <value>RUNNING</value>
  <description>
    default 队列的状态，可以是 RUNNING 或者 STOPPED
  </description>
</property>

<property>
  <name>yarn.scheduler.capacity.root.default.acl_submit_applications</name>
  <value>*</value>
  <description>
    限制哪些用户可向 default 队列中提交应用程序
  </description>
</property>

<property>
  <name>yarn.scheduler.capacity.root.default.acl_administer_queue</name>
  <value>*</value>
  <description>
    限制哪些用户可管理 default 队列中的应用程序，“*”表示任意用户
  </description>
</property>

<property>
  <name>yarn.scheduler.capacity.node-locality-delay</name>
  <value>-1</value>
  <description>调度器尝试调度一个 rack-local Container 之前，最多跳过的调度机会，通常而言，该值被设置成集群中机架数目，默认情况下为 -1，表示不启用该功能
  </description>
</property>
</configuration>

```

从上面这些参数可以看出，Capacity Scheduler 将整个系统资源分成若干个队列，且每个队列有较为严格的资源使用限制，包括每个队列的资源容量限制、每个用户的资源量限制等。通过这些限制，Capacity Scheduler 将整个 Hadoop 集群逻辑上划分成若干个拥有相对独立资源的子集群，而由于这些子集群实际上是公用大集群中的资源，因此可以共享资源，相对于 HOD 而言，提高了资源利用率且降低了运维成本。

当管理员需动态修改队列资源配置时，可修改配置文件 `conf/ capacity-scheduler.xml`，

然后运行 “yarn mradmin -refreshQueues”，具体如下：

```
$ vi ${HADOOP_HOME}/conf/capacity-scheduler.xml
$ ${HADOOP_YARN_HOME}/bin/yarn mradmin -refreshQueues
```

注意 当前 Capacity Scheduler 不允许管理员动态减少队列数目，且更新的配置参数值应是合法值，否则会导致配置文件加载失败。

6.5.2 Capacity Scheduler 实现

本小节我们将详细介绍 Capacity Scheduler 的实现过程。

1. 应用程序初始化

应用程序被提交到 ResourceManager 上后，ResourceManager 会向 Capacity Scheduler 发送一个 SchedulerEventType.APP_ADDED 事件，Capacity Scheduler 收到该事件后，将为应用程序创建一个 FiCaSchedulerApp 对象跟踪和维护该应用程序的运行信息，同时将应用程序提交到对应的叶子队列中，叶子队列会对应用程序进行一系列合法性检查。只有通过这些合法性检查，应用程序才算提交成功，这些合法性检查包括以下几个方面：

- ☐ 应用程序所属用户拥有该叶子队列的应用程序提交权限；
- ☐ 队列及其父队列当前处于 RUNNING 状态（递归检查）；
- ☐ 队列当前已提交应用程序数目未达到管理员设定的上限；
- ☐ 应用程序所属用户提交的应用程序数目未超过管理员设定的上限。

2. 资源调度

当 ResourceManager 收到来自 NodeManager 发送的心跳信息后，将向 Capacity Scheduler 发送一个 SchedulerEventType.NODE_UPDATE 事件，Capacity Scheduler 收到该事件后，会依次进行以下操作。

（1）处理心跳信息

NodeManager 发送的心跳信息中有两类信息需资源调度器处理，一类是最新启动的 Container，另一类是运行完成的 Container，具体如下：

- ☐ 对于最新启动的 Container，资源调度器需向 ResourceManager 发送一个 RMContainer-EventType.LAUNCHED，进而将该 Container 从超时监控队列中删除。当资源调度器为 ApplicationMaster 分配一个 Container 后，为了防止 Application-Master 长时间不适用该 Container 造成资源浪费，它会将该 Container 加入一个超时监控队列中。如果一段时间内，该队列中的 Container 仍未被使用，则资源调度器会回收该 Container。
- ☐ 对于运行完成的 Container，资源调度器将回收它使用的资源，以便接下来对这些资源进行再分配。

处理完以上两类信息后，Capacity Scheduler 将节点上的空闲资源分配给应用程序。

(2) 资源分配

前面提到, 用户提交应用程序后, 应用程序对应的 ApplicationMaster 会为它申请资源, 而资源的表示方式是 Container。Container 主要包含 5 类信息, 分别是优先级、期望资源所在节点、资源量、Container 数目和是否松弛本地性 (即是否在没有满足节点本地性资源时, 选择机架本地性资源)。资源调度器收到资源申请后, 将暂时将这些请求存放到一个数据结构中, 以等待空闲资源出现后为其分配合适的资源。Container 请求的描述方式如下:

```
message ResourceRequestProto {
  optional PriorityProto priority = 1; // 优先级
  optional string resource_name = 2; // 期望资源所在的节点或者机架
  optional ResourceProto capability = 3; // 资源量
  optional int32 num_containers = 4; // Container 数目
  optional bool relax_locality = 5 [default = true]; // 是否松弛本地性
}
```

YARN 采用了三级资源分配策略, 如图 6-9 所示, 当一个节点上有空闲资源时, 它会依次选择队列、应用程序和 container (请求) 使用该资源, 接下来依次介绍三级资源分配策略。

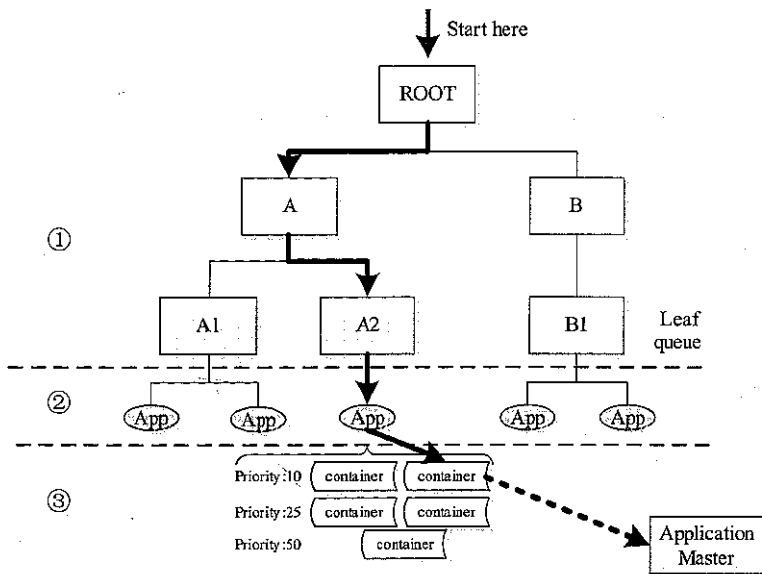


图 6-9 Capacity Scheduler 资源分配流程

步骤 1 选择队列。

YARN 采用了层次结构组织队列, 这将队列结构转换成了树形结构, 这样资源分配过程实际上就是基于优先级的多叉树遍历的过程。在选择队列时, YARN 采用了基于优先级的深度优先遍历方法, 具体如下: 从根队列开始, 按照它的子队列资源使用率 (相当于优先级) 由小到大依次遍历各个子队列。如果子队列为叶子队列, 则依次按照步骤 2 和步骤 3

中方法在队列中选择一个 Container (请求), 否则以该子队列为根队列, 重复以上过程, 直到找到一个合适的 Container (请求) 并退出。

注意 上述“队列资源使用率”计算方法为用已使用的资源量除以队列资源容量 (由管理员配置)。对于非叶子队列, 它的已使用资源量是各个子队列已使用资源量之和。

步骤 2 选择应用程序。

在步骤 1 中选中一个叶子队列后, Capacity Scheduler 按照提交时间对叶子队列中的应用程序进行排序 (实际排序时用的是 Applition ID, 提交时间越早的应用程序, Application ID 越小), 并依次遍历排序后的应用程序, 以找到一个或多个最合适的 Container (请求)。

步骤 3 选择 Container (请求)。

对于同一个应用程序, 它请求的 Container 可能是多样化的, 涉及不同的优先级、节点、资源量和数量。当选中一个应用程序后, Capacity Scheduler 将尝试优先满足优先级高的 Container。对于同一类优先级, 优先选择满足本地性的 Container, 同 MRv1 一样, 它会依次选择 node local、rack local 和 no local 的 Container。

注意 Capacity Scheduler 有两种比较器用以比较两个资源的大小 (比如比较用户当前使用的资源量是否超过了设置的上限资源量), 默认是 DefaultResourceCalculator, 它只考虑内存资源。另外一种 DominantResourceCalculator, 它采用了 DRF 比较算法, 同时考虑内存和 CPU 两种资源。管理员可通过参数 yarn.scheduler.capacity.resource-calculator 设置资源比较器。

综上所述, Capacity Scheduler 资源分配策略与 MRv1 中的基本一致, 均是采用了三级资源分配策略, 即当空闲资源出现时, 依次选择一个队列、应用程序 (作业) 和 Container (任务) 使用该资源。

3. 其他事件处理

除了上述的 SchedulerEventType.APP_ADDED 和 SchedulerEventType.NODE_UPDATE 两种事件, Capacity Scheduler 还会收到来自 ResourceManager 各类组件发送的其他几种 SchedulerEventType 类型的事件, 包括 APP_REMOVED、NODE_ADDED、NODE_REMOVED 和 CONTAINER_EXPIRED。下面分别进行介绍:

- APP_REMOVED: 在多种情况下 Capacity Scheduler 将收到该事件, 包括应用程序正常结束、应用程序被杀死等。Capacity Scheduler 收到该事件后, 首先会向所有未运行完成的 Container 发送一个 RMContainerEventType.KILL 事件, 以释放正在使用的 Container; 然后才会将应用程序相关数据结构从内存中移除。
- NODE_ADDED: 当集群中动态加入一个节点时 (比如管理员动态扩充集群规模或者节点断开连接后又复活等), Capacity Scheduler 将收到该事件。Capacity Scheduler 收到该事件后, 只需在相应数据结构中记录 NodeManager 信息并增加系

统总资源量即可。

- **NODE_REMOVED**: 当集群中动态移除一个节点时 (比如管理员动态移除节点或者节点在一定时间内未汇报心跳而被 ResourceManager 移除集群), Capacity Scheduler 将收到该事件。Capacity Scheduler 收到该事件后, 除了移除 NodeManager 信息并减少系统总资源量外, 还需向所有正运行的 Container 发送一个 RMContainerEventType.KILL 事件, 以清空相关的信息。
- **CONTAINER_EXPIRED**: 当 Capacity Scheduler 将一个 Container 分配给 ApplicationMaster 后, ApplicationMaster 在一定时间内必须使用该 Container, 否则 ResourceManager 将进行强制回收, 此时会触发一个 CONTAINER_EXPIRED 事件。

6.6 Fair Scheduler

Fair Scheduler^①是 Facebook 开发的多用户调度器, 同 Capacity Scheduler 类似, 它以队列为单位划分资源, 每个队列可设定一定比例的资源最低保证和使用上限, 同时, 每个用户也可设定一定的资源使用上限以防止资源滥用; 当一个队列的资源有剩余时, 可暂时将剩余资源共享给其他队列。当然, Fair Scheduler 也存在很多与 Capacity Scheduler 不同之处, 这主要体现在以下几个方面:

- **资源公平共享**。在每个队列中, Fair Scheduler 可选择按照 FIFO、Fair 或 DRF 策略为应用程序分配资源。其中, Fair 策略是一种基于最大最小公平算法^②实现的资源多路复用方式, 默认情况下, 每个队列内部采用该方式分配资源。这意味着, 如果一个队列中有两个应用程序同时运行, 则每个应用程序可得到 1/2 的资源; 如果三个应用程序同时运行, 则每个应用程序可得到 1/3 的资源。
- **支持资源抢占**。当某个队列中有剩余资源时, 调度器会将这些资源共享给其他队列, 而当该队列中有新的应用程序提交时, 调度器要为其回收资源。为了尽可能降低不必要的计算浪费, 调度器采用了先等待再强制回收的策略, 即如果等待一段时间后尚有未归还的资源, 则会进行资源抢占: 从那些超额使用资源的队列中杀死一部分任务, 进而释放资源。
- **负载均衡**。Fair Scheduler 提供了一个基于任务数目的负载均衡机制, 该机制尽可能将系统中的任务均匀分配到各个节点上。此外, 用户也可以根据自己的需要设计负载均衡机制。
- **调度策略配置灵活**。Fair Scheduler 允许管理员为每个队列单独设置调度策略 (当前支持 FIFO、Fair 或 DRF 三种)。
- **提高小应用程序响应时间**。由于采用了最大最小公平算法, 小作业可以快速获取资源并运行完成。

① 参见网址 http://hadoop.apache.org/docs/stable/fair_scheduler.html。

② Max-Min Fairness (Wikipedia): http://en.wikipedia.org/wiki/Max-min_fairness。

6.6.1 Fair Scheduler 功能介绍

同 Capacity Scheduler 类似, Fair Scheduler 也是一个多用户调度器, 它同样添加了多层级别的资源限制条件以更好地让多用户共享一个 Hadoop 集群, 比如队列资源限制、用户应用程序数目限制等。然而, 由于 Fair Scheduler 增加了很多新的特性, 因此它的配置选项更多, 为了能够更详尽地了解 Fair Scheduler 的功能, 我们从它的配置文件讲起。Fair Scheduler 的配置选项包括两部分: 其中一部分在 `yarn-site.xml` 中, 主要用于配置调度器级别的参数; 另外一部分在一个自定义配置文件 (默认是 `fair-scheduler.xml`) 中, 主要用于配置各个队列的资源量、权重等信息。

1. 配置文件 `yarn-site.xml`

配置 `yarn-site.xml` 文件时涉及的参数如下:

- `yarn.scheduler.fair.allocation.file`: 自定义 XML 配置文件所在位置, 该文件主要用于描述各个队列的属性, 比如资源量、权重等, 具体配置格式将在后面介绍。
- `yarn.scheduler.fair.user-as-default-queue`: 当应用程序未指定队列名时, 是否指定用户名作为应用程序所在的队列名。如果设置为 `false` 或者未设置, 则所有未知队列的应用程序将被提交到 `default` 队列中, 默认值为 `true`。
- `yarn.scheduler.fair.preemption`: 是否启用抢占机制, 默认值是 `false`。
- `yarn.scheduler.fair.sizebasedweight`: 在一个队列内部分配资源时, 默认情况下, 采用公平轮询的方法将资源分配给各个应用程序, 而该参数则提供了另外一种资源分配方式, 则按照应用程序资源需求数目分配资源, 需求资源数量越多, 分配的资源越多。默认情况下, 该参数值为 `false`。
- `yarn.scheduler.assignmultiple`: 是否启动批量分配功能。当一个节点出现大量资源时, 可以一次分配完成, 也可以多次分配完成。默认情况下, 该参数值为 `false`。
- `yarn.scheduler.fair.max.assign`: 如果开启批量分配功能, 可指定一次分配的 Container 数目。默认情况下, 该参数值为 `-1`, 表示不限制。
- `yarn.scheduler.fair.locality.threshold.node`: 当应用程序请求某个节点上的资源时, 它可以接受的可跳过的最大资源调度机会。当按照分配策略可将一个节点上的资源分配给某个应用程序时, 如果该节点不是应用程序期望的节点, 可选择跳过该分配机会暂时将资源分配给其他应用程序, 直到满足该应用程序需求的节点资源出现。通常而言, 一次心跳代表一次调度机会, 而该参数则表示跳过调度机会占节点总数的比例, 默认情况下, 该值为 `-1.0`, 表示不跳过任何调度机会。
- `yarn.scheduler.fair.locality.threshold.rack`: 当应用程序请求某个机架上的资源时, 它可以接受的可跳过的最大资源调度机会。
- `yarn.scheduler.increment-allocation-mb`: 内存规整化单位, 默认是 `1024`, 这意味着, 如果一个 Container 请求资源是 `1.5GB`, 则将被调度器规整化为 `ceiling(1.5 GB / 1GB) * 1GB=2GB`。

□ `yarn.scheduler.increment-allocation-vcores` : 虚拟 CPU 规整化单位, 默认是 1, 含义与内存规整化单位类似。

2. 自定义配置文件

Fair Scheduler 允许用户将队列信息专门放到一个配置文件 (默认是 `fair-scheduler.xml`), 对于每个队列, 管理员可配置以下几个选项:

- `minResources` : 最少资源保证量, 设置格式为 “X mb, Y vcores”, 当一个队列的最少资源保证量未满足时, 它将优先于其他同级队列获得资源。对于不同的调度策略 (后面会详细介绍), 最少资源保证量的含义不同, 即对于 Fair 策略, 只考虑内存资源, 即如果一个队列使用的内存资源超过了它的最少资源量, 则认为它已得到了满足; 对于 DRF 策略, 则考虑主资源使用的资源量, 即如果一个队列的主资源量超过它的最少资源量, 则认为它已得到了满足。
- `maxResources` : 最多可以使用的资源量, Fair Scheduler 会保证每个队列使用的资源量不会超过该队列的最多可使用资源量。
- `maxRunningApps` : 最多同时运行的应用程序数目。通过限制该数目, 可以避免多个应用程序同时运行时占用过多的临时资源。比如, 可防止多个 Map Task 同时运行时产生的中间输出结果 “撑爆” 磁盘。
- `minSharePreemptionTimeout` : 最小共享量抢占时间。如果一个资源池在该时间内使用的资源量一直低于最小资源量, 则开始抢占资源。
- `schedulingMode/schedulingPolicy` : 队列采用的调度模式, 可以是 FIFO、Fair 或者 DRF。
- `aclSubmitApps` : 可向队列中提交应用程序的用户列表, 默认情况下为 “*”, 表示任何用户均可以向该队列提交应用程序。需要注意的是, 该属性具有继承性, 即子队列的列表会继承父队列的列表。
- `aclAdministerApps` : 队列的管理员列表。一个队列的管理员可管理队列中的资源和应用程序, 比如可杀死队列中任意应用程序。

管理员也可为单个用户添加 `maxRunningJobs` 属性限制其最多同时运行的应用程序数目。此外, 管理员也可通过以下参数设置以上属性的默认值:

- `userMaxJobsDefault` : 用户的 `maxRunningJobs` 属性的默认值。
- `defaultMinSharePreemptionTimeout` : 队列的 `minSharePreemptionTimeout` 属性的默认值。
- `defaultPoolSchedulingMode` : 队列的 `schedulingMode` 属性的默认值。
- `fairSharePreemptionTimeout` : 公平共享量抢占时间。如果一个资源池在该时间内使用资源量一直低于公平共享量的一半, 则开始抢占资源。

下面通过实例对上述内容进行介绍。假设要为一个 Hadoop 集群设置三个队列 `queueA`、`queueB` 和 `queueC`。其中, `queueB` 和 `queueC` 为 `queueA` 的子队列, 且规定普通用户最多可同时运行 40 个应用程序, 但用户 `userA` 最多可同时运行 400 个应用程序, 那么可在自定义

配置文件中进行如下设置：

```
<allocations>
  <queue name=" queueA" >
    <minResources>100 mb, 100 vcores</minResources>
    <maxResources>150 mb, 150 vcores</maxResources>
    <maxRunningApps>200</maxRunningApps>
    <minSharePreemptionTimeout>300</minSharePreemptionTimeout>
    <weight>1.0</weight>
    <queue name=" queueB" >
      <minResources>30 mb, 30 vcores</minResources>
      <maxResources>50 mb, 50 vcores</maxResources>
    </queue>
    <queue name=" queueC" >
      <minResources>50 mb, 50 vcores</minResources>
      <maxResources>50 mb, 50 vcores</maxResources>
    </queue>
  </queue>

  <user name=" userA" >
    <maxRunningApps>400</maxRunningApps>
  </user>
  <userMaxAppsDefault>40</userMaxAppsDefault>
  <fairSharePreemptionTimeout>6000</fairSharePreemptionTimeout>
</allocations>
```

6.6.2 Fair Scheduler 实现

同 Capacity Scheduler 一样，Fair Scheduler 也需处理 NODE_UPDATE、APP_ADDED、APP_REMOVED、NODE_ADDED、NODE_REMOVED 和 CONTAINER_EXPIRED 等 6 种 SchedulerEventType 类型的事件，其中，后 5 种类型事件的处理逻辑与 Capacity Scheduler 一致，本小节重点介绍 NODE_UPDATE 事件处理机制。

前面提到，用户提交应用程序后，应用程序对应的 ApplicationMaster 会为它申请资源，而资源的表示方式是 Container。Container 主要包含四类信息，分别是优先级、期望资源所在节点、资源量和 Container 数目。资源调度器收到资源申请后，将暂时将这些请求存放到一个数据结构中，以等待空闲资源出现后为其分配合适的资源。

同 Capacity Scheduler 不同的是，Fair Scheduler 提供了更多样化的调度策略，它允许每个队列单独配置调度策略。当前共有三种策略可选，分别是 FIFO、Fair 和 DRF，即先来先服务、公平调度和主资源公平调度。具体含义如下：

- FIFO：按照优先级高低调度，如果优先级相同，则按照提交时间先后顺序调度，如果提交时间也相同，则按照（队列或者应用程序）名称大小（字符串比较）调度。
- Fair：按照内存资源使用量比率调度，即按照 `used_memory/minShare` 大小调度（核心思想是按照该调度算法决定调度顺序，但还需考虑一些边界情况）。
- DRF：按照主资源公平调度算法进行调度，具体已经在 6.3.3 节进行了介绍。

需要注意的是,调度策略在队列间和队列内部可单独设置。对于叶子队列,它设置的调度策略决定了内部的应用程序调度策略;对于非叶子队列,它设置的调度策略决定了各子队列间的调度策略,比如一个叶子队列设置的调度策略是 Fair,则它内部应用程序的调度策略是 Fair,如果一个非叶子队列设置的调度策略是 DRF,则它的子队列的间的调度策略是 DRF。

同 Capacity Scheduler 一样, Fair Scheduler 采用了三级资源分配策略,即当一个节点上有空闲资源时,它会依次选择队列、应用程序和 Container 使用该资源,接下来依次介绍三级资源分配策略。

步骤 1 选择队列。

YARN 采用了层次结构组织队列,但实际存放应用程序的只有叶子队列,其他队列只是一个逻辑概念,用以辅助计算叶子队列的资源量。正是基于以上考虑,选择队列实际上就是根据当前所有队列的资源使用情况查找一个最合适的叶子队列。同 Capacity Scheduler 一样, Fair Scheduler 也采用了深度优先遍历算法:从根队列开始,使用 FIFO、Fair 或者 DRF 策略对它的所有子队列进行排序,然后依次处理每个子队列。对于某个子队列,如果它是叶子队列,则直接返回,否则以该队列为根队列,继续按照以上方法查找叶子队列。

选中一个子队列后, Fair Scheduler 再按照步骤 2 和步骤 3 选择一个或多个最合适的 Container。

步骤 2 选择应用程序。

在步骤 1 中选中一个叶子队列后, Capacity Scheduler 会按照 Fair 策略对叶子队列内部的应用程序进行排序,并依次检查排序后的应用程序,以按照步骤 3 找到一个最合适的 Container。

步骤 3 选择 Container。

对于同一个应用程序,它请求的 Container 可能是多样化的,涉及不同的优先级、节点、资源量和数量。当选中一个应用程序后, Capacity Scheduler 将尝试优先满足优先级高的 Container。对于同一类优先级,优先选择满足本地性的 Container,同 MRv1 一样,它会依次选择 node local、rack local 和 no local 的 Container。

综上所述, YARN 中的 Fair Scheduler 资源分配策略与 MRv1 中的基本一致,均是采用了三级资源分配策略,即当空闲资源出现时,依次选择一个队列、应用程序(作业)和 Container(任务)使用该资源。

6.6.3 Fair Scheduler 与 Capacity Scheduler 对比

随着 Hadoop 版本的演化, Fair Scheduler 和 Capacity Scheduler 的功能越来越完善,包括层级队列组织方式、资源抢占、批量调度等,也正因如此,两个调度器同质化越来越严重。目前看来,两个调度器在应用场景、支持的特性、内部实现等方面非常接近,而由于 Fair Scheduler 支持多种调度策略,因此可以认为 Fair Scheduler 具备了 Capacity Scheduler 具有的所有功能。

表 6-6 从多个方面对比了这两个调度器的异同, 通过这个表读者能更好地理解 Capacity Scheduler 与 Fair Scheduler 的相同点和不同点。

表 6-6 Capacity Scheduler 与 Fair Scheduler 比较

	Capacity Scheduler	Fair Scheduler
目标	提供一种多用户共享 Hadoop 集群的方法, 以提高资源利用率和减小集群管理成本	
设计思想	资源按比例分配给各个队列, 并添加各种严格的限制防止个别用户或者队列独占资源	基于最大最小公平算法将资源分配给各个资源池或者用户
是否支持动态加载配置文件	是	是
是否支持负载均衡	否	是
是否支持资源抢占	是	是
是否支持批量调度	是	是
Container 请求资源粒度	最小资源量的整数倍, 比如 Container 请求量是 1.5GB, 最小资源量是 1GB, 则 Container 请求自动被归一化为 2GB	有专门的内存规整化参数控制, 粒度更小, Container 请求量是 1.5GB, 规整化值为 128MB, 则 Container 请求不变
本地性任务调度优化	基于跳过次数的延迟调度	
队列间资源分配方式	资源使用率低者优先	Fair、FIFO 或 DRF
队列内部资源分配方式	FIFO 或者 DRF	Fair、FIFO 或 DRF

6.7 其他资源调度器介绍

除了前面介绍的调度器外, 本节将介绍几种其他的调度器。

1. 自适应调度器

自适应调度器 (Adaptive Scheduler)^① 是一种以用户期望的运行时间为目标的调度器, 该调度器根据每个作业的不同会被分解成多个任务的事实, 通过已经运行完成的任务的运行时间估算剩余任务的运行时间, 进而使得该调度器能够根据作业的进度和剩余时间动态地为作业分配资源, 以期望作业在规定时间内运行完成。

2. 自学习调度器

自学习调度器 (Learning Scheduler)^② 是一种基于贝叶斯分类算法的资源感知调度器, 与现有的调度器不同, 它更适用于异构 Hadoop 集群。该调度器的创新之处是将贝叶斯分类算法应用到调度器设计中。

该调度器选取了若干个作业特征 (用向量表示) 作为分类属性。分类属性主要有作业

① 参见网址 <https://issues.apache.org/jira/browse/MAPREDUCE-1380>。

② 参见网址 <https://issues.apache.org/jira/browse/MAPREDUCE-1439>。

平均 CPU 利用率、平均网络利用率、平均磁盘 I/O 利用率和平均内存利用率等, 这些属性的值可通过一个离线系统获取。调度器通过用户标注好的一些作业可训练得到一个分类器, 这样当某个 NodeManager 出现剩余资源时, 会通过心跳汇报所在节点的资源使用信息。调度器收到该信息后, 会将所有作业的特征向量作为贝叶斯分类器的输入, 判断出当前哪些作业可在该 NodeManager 上运行 (称为 “good” 作业), 哪些不可以在该 NodeManager 上运行 (称为 “bad” 作业), 最后通过一个效用函数从所有 “good” 作业中选出一个最合适的作业。

3. 动态优先级调度器

在 0.21.x/0.22.x 版本中, Hadoop 引入了一个新的调度器——动态优先级调度器 (Dynamic Priority Scheduler)^②, 该调度器允许用户动态调整自己获取的资源量以满足其服务质量要求。

该调度器试图把 Hadoop 集群看作一个提供商品的买卖市场, 每个消费者有一定的预算购买自己需要的东西, 且消费者需为购买某件商品竞标, 其中出价高的人可获得较多的商品, 反之, 出价少的人获得的商品也少。由于市场中商品价格是不断上下波动的, 因此消费者可结合自己的需要调整自己的价位以买入更多或者更少的产品。对应到 Hadoop 集群中, Slot 是进行买卖的商品, Hadoop 用户是消费者, 每个用户分配有一定的预算, 在任何一个阶段, 可能有多个用户同时向 Hadoop 集群申请资源, 其中出价高的用户获得的资源多, 且申请资源的用户越多, 单个 Slot 的价位也就越高。

动态优先级调度器的核心思想是在一定的预算约束下, 根据用户提供的消费率按比例分配资源。管理员可根据集群资源总量为每个用户分配一定的预算和一个时间单元的长度 (通常为 10s~1min), 而用户可根据自己的需要动态调整自己的消费率, 即每个时间单元内单个 Slot 的价钱。在每个时间单元内, 调度器按照以下步骤计算每个用户获得的资源量:

1) 计算所有用户的消费率之和 p 。

2) 对于每个用户 i , 分配 $s_i/p \times c$, 其中, s_i 为用户 i 的消费率, c 为 Hadoop 集群中 Slot 总数。

3) 对于每个用户 i , 从其预算中扣除 $s_i \times u_i$, 其中 u_i 为用户正在使用的 Slot 数目。

动态优先级调度器也可作为一个元调度器集成到其他调度器 (比如 FIFO, Fair Scheduler 等) 中, 这样每个队列或者用户的可用资源量直接由动态优先级调度器动态计算得到, 而其他调度器只需负责分配资源即可。相比于其他调度器, 动态优先级调度器允许用户根据需要 (比如完成时间) 动态调整资源, 进而可以对作业运行质量进行精细地控制。

6.8 源代码阅读引导

YARN 资源调度器的所有实现在源代码目录下的 `hadoop-yarn-project/hadoop-yarn/`

② Thomas Sandholm and Kevin Lai, Dynamic proportional share scheduling in hadoop. In JSSPP '10: 15th Workshop on Job Scheduling Strategies for Parallel Processing, 2010.

hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java 目录中, 代码结构如图 6-10 所示。

```

└─ org.apache.hadoop.yarn.server.resourcemanager.scheduler
└─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity
└─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.common.fica
└─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.event
└─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair
└─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.policies
└─ org.apache.hadoop.yarn.server.resourcemanager.scheduler.fifo

```

图 6-10 资源调度器源代码组织结构

资源调度器模块定义了一系列接口和规范, 实现一个新的调度器必须实现这些接口并遵循对应的规范。读者在实现一个新的资源调度器之前, 可先尝试了解这些基本知识, 并通过学习 YARN 自带的几个实现了解一个真实调度器的实现方法。下面介绍 YARN 资源调度器相关的 Java 包:

- `org.apache.hadoop.yarn.server.resourcemanager.scheduler`: 资源调度器的接口定义, 涉及 `ResourceScheduler` (所有调度器主类必须实现该接口)、`SchedulerNode` (用于跟踪集群中一个节点资源状态)、`SchedulerApplication` (用于追踪一个应用程序资源分配情况)、`Queue` (描述一个队列的基本信息) 等。
- `org.apache.hadoop.yarn.server.resourcemanager.scheduler.event`: 资源调度器需要处理的事件定义, 这些事件已在 6.3.1 节进行了详细介绍。
- `org.apache.hadoop.yarn.server.resourcemanager.scheduler.common.fica`: FIFO 和 Capacity Scheduler 实现过程中用到的一些公共基础类。
- `org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity`: Capacity Scheduler 相关实现。
- `org.apache.hadoop.yarn.server.resourcemanager.scheduler.fifo`: FIFO 调度器相关实现。
- `org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.*`: Fair Scheduler 相关实现。

6.9 小结

本章介绍了几种常见的多用户作业调度器, 相比于 FIFO 调度器, 多用户调度器能够更好地满足不同应用程序的服务质量要求。

当前主要有两种多用户作业调度器的设计思路: 第一种是在一个物理集群上虚拟多个 Hadoop 集群, 这些集群各自拥有全套独立的 Hadoop 服务, 比如 JobTracker、TaskTracker 等, 典型的代表是 HOD (Hadoop On Demand) 调度器; 另一种是扩展 Hadoop 调度器, 使之支持多个队列多用户, 典型的代表是 Yahoo! 的 Capacity Scheduler 和 Facebook 的 Fair

Scheduler。本章分别对这两种调度器进行了介绍。

HOD 调度器是一个在共享物理集群上管理若干个 Hadoop 集群的工具，它可以帮助用户在一个共享物理集群上快速搭建若干个独立的虚拟 Hadoop 集群。由于该调度器会产生多个独立的小集群，因此会增加集群运维成本和降低资源利用率。

为了克服 HOD 的缺点，Capacity Scheduler 和 Fair Scheduler 出现了。它们通过扩展调度器功能，在不拆分集群的前提下，将集群中的资源和用户分成若干个队列，并为每个队列分配一定量的资源，同时添加各种限制防止用户或者队列独占资源。由于这种方式能够保证只有一个 Hadoop 集群，因此可大大降低运维成本，同时很容易实现资源共享，进而可明显提高资源利用率。

6.10 问题讨论

问题 1：Capacity Scheduler 和 Fair Scheduler 对每个队列的资源使用有最小量和最大量的限制，其中最大量限制是“hard limit”，即队列中的应用程序资源使用总量永不能超过该值；最小量限制是“soft limit”，即如果队列中的应用程序资源使用量达不到该值，调度器也不会为它预留剩下的资源，而是自动共享给其他队列。试着将 Capacity Scheduler 和 Fair Scheduler 的最小资源量限制改为“hard limit”。

问题 2：修改 ResourceManager 和资源调度器相关实现，使其支持以下容错机制；当一个应用程序的 ApplicationMaster 运行失败时，先尝试在原节点重启它，如果重启失败，再将其重新调度到其他节点上。

问题 3：分析 Capacity Scheduler 和 Fair Scheduler 两个调度器采用的调度算法的时间复杂度。

第 7 章 NodeManager 剖析

NodeManager 是运行在单个节点上的代理，它需要与应用程序的 ApplicationMaster 和集群管理者 ResourceManager 交互：从 ApplicationMaster 上接收有关 Container 的命令并执行之（比如启动、停止 Container）；向 ResourceManager 汇报各个 Container 运行状态和节点健康状况，并领取有关 Container 的命令（比如清理 Container）执行之。

本章将以 NodeManager 包含的功能为线索，深入剖析各个模块的实现原理和工作流程。

7.1 概述

NodeManager (NM) 是 YARN 中单个节点上的代理，它管理 Hadoop 集群中单个计算节点，功能包括与 ResourceManager 保持通信、管理 Container 的生命周期、监控每个 Container 的资源使用（内存、CPU 等）情况、追踪节点健康状况、管理日志和不同应用程序用到的附属服务（auxiliary service）。

7.1.1 NodeManager 基本职能

整体上讲，NodeManager 需通过两个 RPC 协议与 ResourceManager 服务和各个应用程序的 ApplicationMaster 交互，如图 7-1 所示。

1. ResourceTrackerProtocol 协议

NodeManager 通过该 RPC 协议向 ResourceManager 注册、汇报节点健康状况和 Container 运行状态，并领取 ResourceManager 下达的命令，包括重新初始化、清理 Container 占用资源等。在该 RPC 协议中，ResourceManager 扮演 RPC Server 的角色，而 NodeManager 扮演 RPC Client 的角色（由内部组件 NodeStatusUpdater 实现），换句话说，NodeManager 与 ResourceManager 之间采用了“pull 模型”（与 MRv1 类似），NodeManager 总是周期性地主动向 ResourceManager 发起请求，并领取下达给自己的命令。ResourceTrackerProtocol 协议主要提供了以下两个 RPC 函数：

(1) registerNodeManager

NodeManager 启动时通过该 RPC 函数向 ResourceManager 注册，注册信息由 RegisterNodeManagerRequest 封装的，包括如下三部分内容。

- httpPort：该 NodeManager 对外提供的 HTTP 端口号，ResourceManager 会在界面上提供一个可直接访问 NodeManager Web 界面的超链接。
- nodeId：该 NodeManager 所在的 host 和对外的 RPC 端口号。

□ **totalResource** : 该 NodeManager 所在节点总的可分配资源, 当前支持 (物理) 内存和虚拟 CPU 两种资源, 管理员可通过参数 `yarn.nodemanager.resource.cpu-vcores` (默认是 8) 和 `yarn.nodemanager.resource.memory-mb` (单位为 MB, 默认是 8192, 还可通过参数 `yarn.nodemanager.vmem-pmem-ratio` 设置物理内存和虚拟内存使用比率, 默认是 2.1, 即每使用 1MB 物理内存, 最多可使用 2.1MB 虚拟内存) 配置。

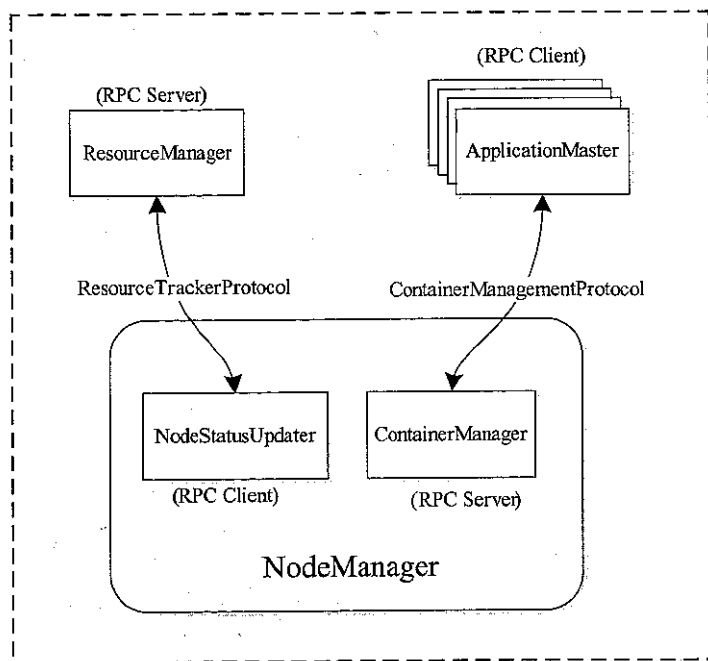


图 7-1 NodeManager 相关的 RPC 协议

ResourceManager 将通过 `registerNodeManager` 函数向 NodeManager 返回一个 `RegisterNodeManagerResponse` 类型的对象, 主要包含以下信息:

- **MasterKey**: 新生成的 Container Token 和 Node Token 的 Master Key。
- **NodeAction**: ResourceManager 向该 NodeManager 返回的下一步操作, 主要包括 NORMAL、RESYNC 和 SHUTDOWN (节点位于黑名单中或者 NodeManager 上报的可用资源低于 ResourceManager 最低要求) 三种, 分别表示正常 (注册成功)、重新同步信息和停止运行。
- **rmIdentifier**: ResourceManager 的标示符 (用于 ResourceManager 重启恢复或者 HA 场景), NodeManager 通过该标识符判断 ApplicationMaster 发送的 Container 来自原始的还是新启动的 ResourceManager。
- **diagnosticsMessage**: NodeManager 注册失败时 (收到 RESYNC 或者 SHUTDOWN 操作指令), 将收到一段诊断信息, 告知具体的失败原因。

(2) nodeHeartbeat

NodeManager 启动后, 定期通过该 RPC 函数向 ResourceManager 汇报 Container 运行信息和节点健康状况, 并领取新的命令, 比如杀死一个 Container。

2. ContainerManagementProtocol 协议

应用程序的 ApplicationMaster 通过该 RPC 协议向 NodeManager 发起针对 Container 的相关操作, 包括启动 Container、杀死 Container、获取 Container 执行状态等。在该协议中, ApplicationMaster 扮演 RPC Client 的角色, 而 NodeManager 扮演 RPC Server 的角色 (由内部组件 ContainerManager 实现), 换句话说, NodeManager 与 ApplicationMaster 之间采用了“push 模型”, ApplicationMaster 可以将 Container 相关操作第一时间告诉 NodeManager, 相比于“pull 模型”, 可大大降低时间延迟。ContainerManagementProtocol 协议主要提供了以下三个 RPC 函数:

- startContainer: ApplicationMaster 通过该 RPC 要求 NodeManager 启动一个 Container。该函数有一个 StartContainerRequest 类型的参数, 封装了 Container 启动所需的本地资源、环境变量、执行命令、Token 等信息。如果 Container 启动成功, 则该函数返回一个 StartContainerResponse 对象。
- stopContainer: ApplicationMaster 通过该 RPC 要求 NodeManager 停止 (杀死) 一个 Container。该函数有一个 StopContainerRequest 类型的参数, 用于指定待杀死的 Container ID。如果 Container 被成功杀死, 则该函数返回一个 StopContainerResponse 对象。
- getContainerStatus: ApplicationMaster 通过该 RPC 获取一个 Container 的运行状态。该函数参数类型为 GetContainerStatusRequest, 封装了目标 Container 的 ID, 返回值为封装了 Container 当前运行状态的类型为 GetContainerStatusResponse 的对象。

7.1.2 NodeManager 内部架构

本节深入 NodeManager 内部, 介绍它的内部组织结构和主要模块, 具体如图 7-2 所示。下面具体介绍图 7-2 中所示的各个模块。

- NodeStatusUpdater: NodeStatusUpdater 是 NodeManager 与 ResourceManager 通信的唯一通道。当 NodeManager 启动时, 该组件负责向 ResourceManager 注册, 并汇报节点上总的可用资源 (该值在运行过程中不再汇报); 之后, 该组件周期性与 ResourceManager 通信, 汇报各个 Container 的状态更新, 包括节点上正运行的 Container、已完成的 Container 等信息, 同时 ResourceManager 会为之返回待清理 Container 列表、待清理应用程序列表、诊断信息、各种 Token 等信息。
- ContainerManager: ContainerManager 是 NodeManager 中最核心组件之一, 它由多个子组件构成, 每个子组件负责一部分功能, 协作共同管理运行在该节点上的所有 Container。各个子组件如下。

NodeManager

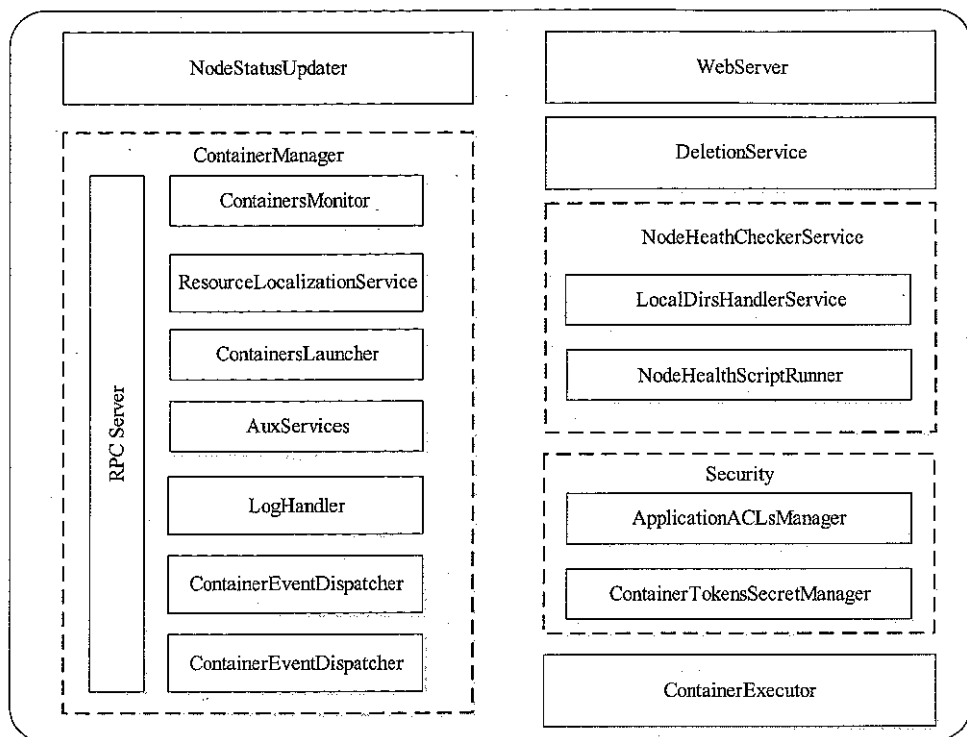


图 7-2 NodeManager 内部架构图

- RPC Server：该 RPC Server 实现了 ContainerManagementProtocol 协议，是 ApplicationMaster 与 NodeManager 通信的唯一通道。ContainerManager 从各个 Application Master 上接收 RPC 请求以启动新的 Container 或者停止正在运行的 Container。需要注意的是，任何 Container 操作均会经 ContainerTokenSecretManager 合法性验证，以防止 ApplicationMaster 伪造启动或者停止 Container 命令。
- ResourceLocalizationService：负责 Container 所需资源的本地化。它能够按照描述从 HDFS 上下载 Container 所需的文件资源，并尽量将它们分摊到各个磁盘上以防止出现访问热点。此外，它会为下载的文件添加访问控制限制，并为之施加合适的磁盘空间使用份额。
- ContainersLauncher：维护了一个线程池以并行完成 Container 相关操作，比如启动或者杀死 Container，其中启动 Container 请求是由 ApplicationMaster 发起的，而杀死 Container 请求则可能来自 ApplicationMaster 或者 ResourceManager。
- AuxServices：NodeManager 允许用户通过配置附属服务的方式扩展自己的功能，这使得每个节点可以定制一些特定框架需要的服务，当然，这些服务是与 NodeManager 上其他服务隔离开的（有自己的安全验证机制）。附属服务需要在

NodeManager 启动之前配置好，并由 NodeManager 统一启动与关闭。典型的应用是 MapReduce 框架中用到的 Shuffle HTTP Server，其是通过封装成一个附属服务（ShuffleHandler）而由各个 NodeManager 启动的。

- ContainersMonitor：ContainersMonitor 负责监控 Container 的资源使用量。为了实现资源隔离和公平共享，ResourceManager 为每个 Container 分配了一定量的资源。而 ContainersMonitor 周期性探测它在运行过程中的资源利用量，一旦发现 Container 超出了它的允许使用份额上限，就向 Container 发送信号将其杀掉，这可以避免资源密集型的 Container 影响同节点上其他正在运行的 Container。在 YARN 中，只有内存资源是通过 ContainersMonitor 监控的方式加以限制的，对于 CPU 资源，则采用了轻量级资源隔离方案 Cgroups。
- LogHandler：一个可插拔组件，用户可通过它控制 Container 日志的保存方式，即是写到本地磁盘上还是将其打包后上传到一个文件系统中。
- ContainerEventDispatcher：Container 事件调度器，负责将 ContainerEvent 类型的事件调度给对应 Container 的状态机 ContainerImpl。
- ApplicationEventDispatcher：Application 事件调度器，负责将 ApplicationEvent 类型的事件调度给对应 Application 的状态机 ApplicationImpl。
- ContainerExecutor：ContainerExecutor 可与底层操作系统交互，安全存放 Container 需要的文件和目录，进而以一种安全的方式启动和清除 Container 对应的进程。目前，YARN 提供了 DefaultContainerExecutor 和 LinuxContainerExecutor 两种实现。其中，DefaultContainerExecutor 是默认实现，未提供任何安全保护措施，它以 NodeManager 启动者的身份启动和停止 Container；而 LinuxContainerExecutor 则以应用程序拥有者的身份启动和停止 Container，因此更加安全，此外，Linux-ContainerExecutor 允许用户通过 Cgroups 对 CPU 资源进行隔离。
- NodeHealthCheckerService：NodeHealthCheckerService 通过周期性地运行一个自定义脚本（由组件 NodeHealthScriptRunner 完成）和向磁盘写文件（由服务 LocalDirsHandlerService 完成）检查节点的健康状况，并将之通过 NodeStatusUpdater 传递给 ResourceManager。一旦 ResourceManager 发现一个节点处于不健康状态，则会将它加入黑名单，此后不再为它分配资源，直到再次转为健康状态。需要注意的是，节点被加入黑名单后，正在运行的 Container 仍会正常运行，不会被杀死。
- DeletionService：NodeManager 将文件删除功能服务化，即提供一个专门的文件删除服务异步删除失效文件，这样可避免同步删除文件带来的性能开销。
- Security：安全模块是 NodeManager 中的一个重要模块，它包含两部分，分别是 ApplicationACLsManager：确保访问 NodeManager 的用户是合法的，ContainerTokenSecretManager：确保用户请求的资源被 ResourceManager 授权过，具体如下：
 - ApplicationACLsManager：NodeManager 需要为所有面向用户的 API 提供安全检

查，如在 Web UI 上只能将 Container 日志显示给授权用户。该组件为每个应用程序维护了一个 ACL 列表，一旦收到类似请求后会利用该列表对其进行验证。

○ ContainerTokenSecretManager：检查收到的各种访问请求的合法性，确保这些请求操作已被 ResourceManager 授权。

□ WebServer：通过 Web 界面向用户展示该节点上所有应用程序运行状态、Container 列表、节点健康状况和 Container 产生的日志等信息。

为了便于读者理解，本章将按照功能点梳理 NodeManager 中的这些模块。

7.1.3 NodeManager 事件与事件处理器

NodeManager 中主要组件之间是通过事件进行交互的，这使得多个组件可异步并发完成各自的功能。总体上讲，NodeManager 中存在两个中央异步调度器，分别位于 NodeManager 和 ContainerManagerImpl 中，它们内部均包含一些事件处理器，这些事件处理器处理的事件类型和事件传递方式如表 7-1 所示。需要注意的是，除了表中这些组件，7.1.2 节中还提到了其他一些组件，而这些组件的功能是直接通过函数调用方式使用的。图 7-3 从动态交互角度展示了事件与事件处理器之间的相互关系。

表 7-1 NodeManager 内部事件与事件处理器

组件名称	事件处理器 / 服务	处理的事件类型	输出事件类型
NodeStatusUpdater	服务	—	ContainerManagerEventType
ContainerManager	事件处理器 服务	ContainerManagerEvent	ApplicationEventType ContainerEventType
ResourceLocalization Service	事件处理器 服务	LocalizationEvent	ApplicationEventType ResourceEventType ContainerEventType
ContainersLauncher	事件处理器 服务	ContainersLauncherEvent	—
AuxServices	事件处理器 服务	AuxServicesEvent	—
ContainersMonitor	事件处理器 服务	ContainersMonitorEvent	ApplicationEventType ContainerEventType
LogHandler	事件处理器 服务	LogHandlerEvent	ApplicationEventType
Container EventDispatcher	事件处理器	ContainerEvent	—
Application EventDispatcher	事件处理器	ApplicationEvent	—
ContainerExecutor	服务	—	ContainerEvent
NodeManager	事件处理器 服务	NodeManagerEvent	ContainerManagerEventType

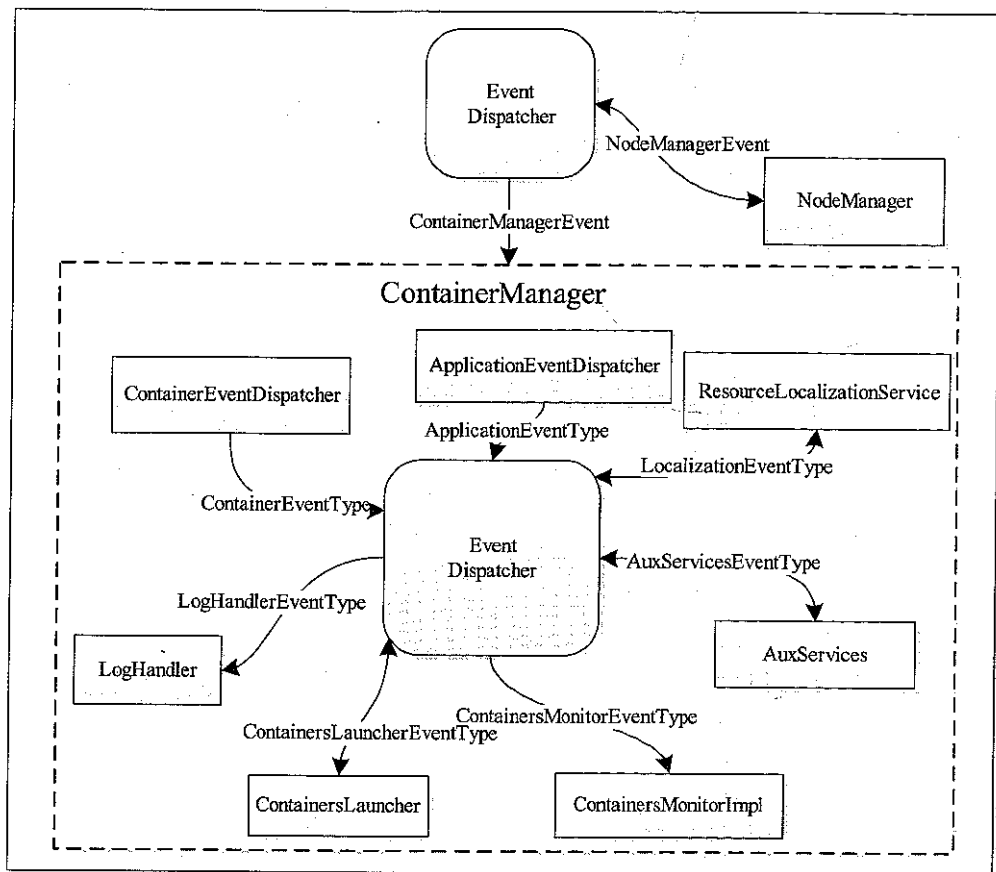


图 7-3 NodeManager 内部事件与事件处理器交互图

7.2 节点健康状况检测

节点健康状况检测是 NodeManager 自带的健康状况诊断机制，通过该机制，NodeManager 可时刻掌握自己的健康状况，并及时汇报给 ResourceManager。而 ResourceManager 则根据每个 NodeManager 的健康状况适当调整分配的任务数目。当 NodeManager 认为自己的健康状况“欠佳”时，可通知 ResourceManager 不再为之分配新任务，待健康状况好转时，再分配任务。该机制不仅可帮助及时发现存在问题的 NodeManager，避免不必要的任务分配，也可以用于动态升级（通过脚本指示 ResourceManager 不再分配任务，等到 NodeManager 上面的任务运行完成后，对它进行升级）。

7.2.1 自定义 Shell 脚本

NodeManager 上有专门一个服务判断所在节点的健康状况，该服务通过两种策略判断

节点健康状况,第一种是通过管理员自定义的 Shell 脚本(NodeManager 上专门有一个周期性任务执行该脚本,一旦该脚本输出以“ERROR”开头的字符串,则认为节点处于不健康状态),另一种是判断磁盘好坏(NodeManager 上专门有一个周期性任务检测磁盘的好坏,如果坏磁盘数目达到一定的比例,则认为节点处于不健康状态)。

NodeHealthScriptRunner 服务主要工作是周期性执行节点健康状况检测脚本。该服务允许管理员配置一个“健康监测脚本”以检查节点健康状况,且管理员可在该脚本中添加任何检查语句作为节点是否健康运行的依据。如果脚本检测到该节点处于不健康状态,它需要在标准输出中打印一条以字符串“ERROR”开头的输出语句。NodeHealthScriptRunner 服务周期性调用健康监测脚本并检查其输出,一旦发现脚本输出是以“ERROR”开头的字符串,则认为节点处于不健康状态,进而将其标注为“unhealthy”并通过心跳告诉 ResourceManager。而 ResourceManager 得知节点状态变为“unhealthy”后,会将其加入黑名单,此后不再为它分配新任务。需要注意的是,只要 NodeManager 服务是活着的,该线程就会一直运行该脚本,一旦发现节点又变为“healthy”,ResourceManager 会立刻将其从黑名单中移除,从而又可以为它分配任务。通过引入该机制,带来了以下几点好处:

- 可作为节点负载的反馈。当前 YARN 仅对 CPU 和内存两种资源进行了隔离,其他资源,比如网络 and 磁盘 IO 等,尚未有任何隔离措施,这使得不同任务之间仍会有干扰。而健康脚本检测的方式可从一定程度上缓解该问题,比如,可让健康检测脚本检查网络、磁盘、文件系统等运行状况,一旦发现特殊情况,比如网络拥塞、磁盘空间不足或者文件系统出现问题,可将健康状况变为“unhealthy”,暂时不接收新的任务,待它们恢复正常后再继续接收新任务。

- 人为暂时维护 NodeManager。如果发现 NodeManager 所在节点出现故障,可通过控制脚本输出暂时让该 NodeManager 停止接收新任务以便进行维护,待维护完成后,修改脚本输出以让 NodeManager 继续接收新任务。

NodeHealthScriptRunner 服务包含四个可配置参数,具体如表 7-2 所示,管理员可根据需要在 yarn-site.xml 文件中配置这些参数。

表 7-2 NodeHealthScriptRunner 服务的配置参数

名 称	含 义
yarn.nodemanager.health-checker.script.path	健康检测脚本所在的绝对路径,服务 NodeHealthScriptRunner 会周期性执行该脚本以判断节点健康状况,如果该值为空,则不会启动该线程
yarn.nodemanager.health-checker.interval-ms	健康监测脚本调用频率(单位:毫秒),默认是 600 000 (10 分钟)
yarn.nodemanager.health-checker.script.timeout-ms	如果健康监测脚本在一定时间(单位:毫秒)内没有响应,则 NodeHealthScriptRunner 服务会将节点标注为“unhealthy”,默认是 1 200 000 (20 分钟)
yarn.nodemanager.health-checker.script.opts	监控脚本的输入参数,如果有多个参数则用逗号隔开

下面给出一个健康监测脚本实例,在这个 Shell 脚本中,当一个节点上的空闲内存

量低于总内存量的 10% 时，将打印以 “ERROR” 开头的字符串，这样该节点将不再向 ResourceManager 请求新任务。

```
#!/bin/bash
MEMORY_RATIO=0.1
freeMem=`grep MemFree /proc/meminfo | awk '{print $2}'`
totalMem=`grep MemTotal /proc/meminfo | awk '{print $2}'`
limitMem=`echo | awk '{print int("'$totalMem'*"$MEMORY_RATIO)}``
if [ $freeMem -lt $limitMem ];then
    echo "ERROR, totalMem=$totalMem, freeMem=$freeMem, limitMem=$limitMem"
else
    echo "OK, totalMem=$totalMem, freeMem=$freeMem, limitMem=$limitMem"
fi
```

7.2.2 检测磁盘损坏数目

除了健康状况检测机制外，YARN 还提供了另外一种判断 NodeManager 是否健康的机制：检测磁盘损坏数目。管理员可通过参数 `yarn.nodemanager.disk-health-checker.enable` 设置是否启用该功能，默认情况下是启用的。该机制是由 LocalDirsHandlerService 服务实现的，它周期性任务检测 NodeManager 本地磁盘的好坏，一旦发现正常磁盘的比例低于一定的比例，则认为节点处于不健康状态，便通过心跳告诉 ResourceManager，从而不再接收到新的任务。

管理员配置 YARN 时，会设置 NodeManager 的本地可用目录列表（由参数 `yarn.nodemanager.local-dirs` 设置，通常用于存储应用程序中间结果，比如 MapReduce 作业中 Map Task 的中间输出结果）和日志存放目录列表（由参数 `yarn.nodemanager.log-dirs` 设置），这些目录的可用性直接决定着 NodeManager 的可用性。因此，NodeManager 作为节点的代理和管理者，应该负责检测这两类目录列表的可用性，并及时将不可用目录剔除掉。LocalDirsHandlerService 服务中专门有一个定时任务周期性检测这些目录的好坏，一旦发现正常磁盘的比例低于 `yarn.nodemanager.disk-health-checker.min-healthy-disks`（默认是 0.25），就认为该节点处于“不健康”的状态，此后 ResourceManager 不再为它分配新任务。

NodeManager 判断一个目录（磁盘）好坏的方法是，如果一个目录具有读、写和执行权限，则认为它是正常的，否则将被加入坏磁盘列表，此后不再使用。

7.3 分布式缓存机制

在 YARN 中，分布式缓存是一种分布式文件分发与缓存机制，类似于 MRv1 中的 DistributedCache，其主要作用是将用户应用程序执行时所需的外部文件资源自动透明地下载并缓存到各个节点上，从而省去了用户手动部署这些文件的麻烦。YARN 分布式缓存工作流程如图 7-4 所示。

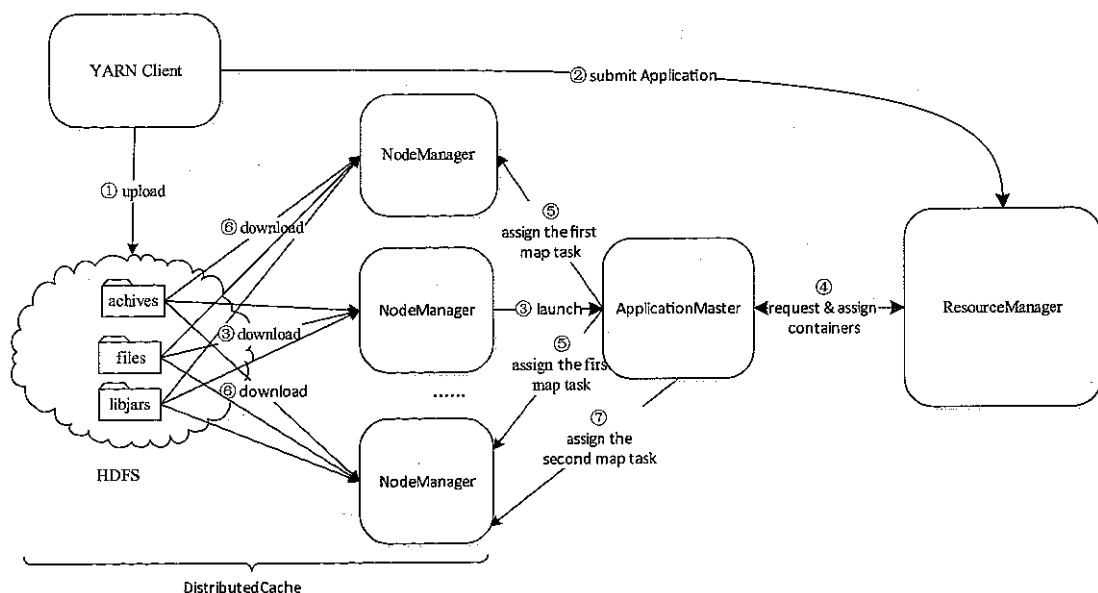


图 7-4 YARN 分布式缓存机制

YARN 分布式缓存工作流程具体如下：

步骤 1 客户端将应用程序所需的文件资源（外部字典、JAR 包、二进制文件等）提交到 HDFS 上。

步骤 2 客户端将应用程序提交到 ResourceManager 上。

步骤 3 ResourceManager 与某个 NodeManager 通信，启动应用程序 ApplicationMaster，NodeManager 收到命令后，首先从 HDFS 下载文件（缓存），然后启动 ApplicationMaster。

步骤 4 ApplicationMaster 与 ResourceManager 通信，以请求和获取计算资源。

步骤 5 ApplicationMaster 收到新分配的计算资源后，与对应的 NodeManager 通信，以启动任务。

步骤 6 如果该应用程序第一次在该节点上启动任务，则 NodeManager 首先从 HDFS 上下载文件缓存到本地，然后启动任务。

步骤 7 NodeManager 后续收到启动任务请求后，如果文件以在本地缓存，则直接运行任务，否则等待文件缓存完成后再启动。

各节点上的缓存文件由对应的 NodeManager 管理和维护。考虑到磁盘空间的有限性，NodeManager 采用了一定的缓存置换算法定期清理失效文件，具体将在 7.3.2 节中介绍。

注意 在 Hadoop 中，分布式缓存并不是将文件缓存到集群中各个节点的内存中，而是将文件缓存到各节点的本地磁盘上，以便执行任务时直接从本地磁盘上读取文件。

需要说明的是，由于分布式缓存可以对文本文件、目录、JAR 包和归档文件等进行下载与缓存。为了规范化，我们统一将文本文件、目录、JAR 包和归档文件等称为“资源”。

上述过程中，步骤 3 和步骤 6 用到了分布式文件缓存机制，接下来重点剖析该机制。

7.3.1 资源可见性与分类

分布式缓存机制是由各个 NodeManager 实现的，主要功能是将应用程序需要的文件资源（一般是只读的）缓存到本地，以方便后续任务运行。资源缓存是用时触发的，也就是说，由第一个用到该资源的任务触发的，后续同类任务无须再次进行缓存，直接使用已经缓存好的即可。

按照可见性（LocalResourceVisibility），如图 7-5 所示，NodeManager 将资源分为三类：

□ PUBLIC：节点上所有用户共享该资源，只要有一个用户的应用程将这些资源缓存到本地，其他所有用户的所有应用程序均可使用它们。

□ PRIVATE：节点上同一用户的所有应用程序共享该资源，一旦该用户的第一个应用程序将之缓存到本地，该用户后续所有的应用程序均可共享该资源。

□ APPLICATION：节点上同一应用程序的所有 Container 共享，其他用户或者通用户的其他程序不可使用该资源。默认情况下，MapReduce 作业的 split 元信息文件 job.splitmetainfo 和属性文件 job.xml 的可见性是 APPLICATION。

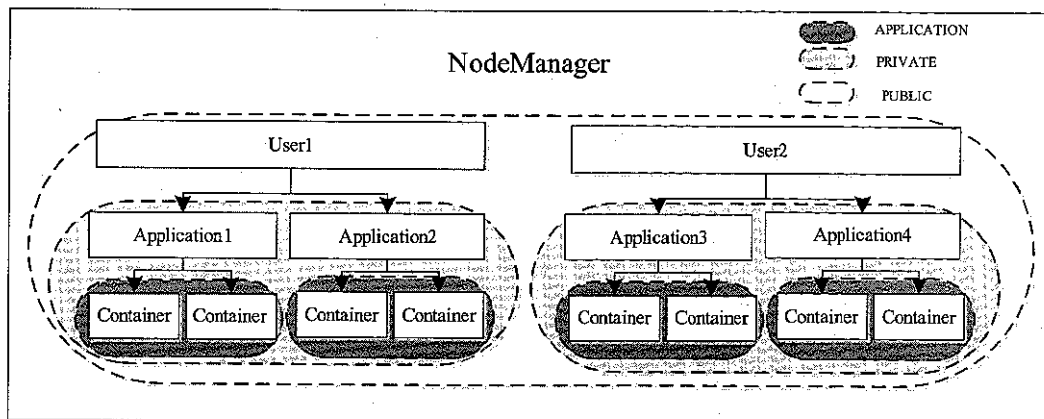


图 7-5 NodeManager 缓存的可见性

与资源可见性相关的 Protocol Buffers 定义如下（在 yarn_protos.proto 中）：

```
enum LocalResourceVisibilityProto {
    PUBLIC = 1;
    PRIVATE = 2;
    APPLICATION = 3;
}
```

按照资源类型（LocalResourceType），NodeManager 将资源分为三类：

□ ARCHIVE：归档文件，当前支持后缀为“jar”、“zip”、“tar.gz”、“tgz”和“tar”的 5 种归档文件，NodeManager 能自动在工作目录中对这 5 类归档文件进行解压缩

(如果是后缀为“.jar”的文件,还可自动将其加到 CLASSPATH 中),方便用户程序使用。

□ FILE: 普通文件, NodeManager 只是简单地将这类文件下载到工作目录中,不做任何处理。

□ PATTERN: 以上两种类型的混合体,有多种类型文件存在,而用户可通过一个正则表达式指定哪些属于 ARCHIVE 文件,需要自动解压缩。

与资源类型相关的 Protocol Buffers 定义如下(在 yarn_protos.proto 中):

```
enum LocalResourceTypeProto {
    ARCHIVE = 1;
    FILE = 2;
    PATTERN = 3;
}
```

应用程序客户端需设置以下几个属性(使用 Protocol Buffers 定义)以定义分布式缓存中的文件。需要注意的是, YARN 是通过比较 resource、timestamp、type 和 pattern 四个字段是否相同来判断两个资源请求是否相同的。如果一个已经被缓存到各个节点上的文件被用户修改了,则下次使用时会自动触发一次缓存更新,以重新从 HDFS 上下载该文件。

```
message LocalResourceProto {
    optional URLProto resource = 1; // 通常是存在 HDFS 上的文件的路径
    optional int64 size = 2; // 文件大小
    optional int64 timestamp = 3; // 最后修改时间
    optional LocalResourceTypeProto type = 4; // 资源类型
    optional LocalResourceVisibilityProto visibility = 5; // 资源可见性
    optional string pattern = 6; // 正则表达式, 当资源可见性为 PATTERN 时有用
}
```

下面举例说明。YARN MapReduce 是采用目录权限方式判断资源可见性的, 如果一个 HDFS 文件的父目录的用户执行权限、组执行权限和其他组执行权限都是打开的(比如可以为“drwxr-xr-x”), 则认为它具有 PUBLIC 可见性, 否则是 PRIVATE 可见性。换句话说, 如果你想将一个文件可见性设置为 PUBLIC, 必须在运行 MapReduce 应用程序之前将它上传到 HDFS 上, 并修改它在所目录的权限。需要注意的是, 每次运行时临时由客户端自动从本地上传到 HDFS 上的文件默认全部是 PRIVATE 权限。同 MRv1 一样, 用户可采用以下 MapReduce 编程接口设置需分布式缓存的文件:

```
// 添加归档文件
void addCacheArchive(URL uri, Configuration conf)
void setCacheArchives(URL[] archives, Configuration conf)
// 添加普通文件
void addCacheFile(URL uri, Configuration conf)
void setCacheFiles(URL[] files, Configuration conf)
// 将三方 JAR 包或者动态库添加到 classpath 中
void addFileToClassPath(Path file, Configuration conf)
// 在任务工作目录下建立文件软连接
void createSymlink(Configuration conf)
```

YARN MapReduce 客户端从 Configuration 中解析出各个属性之后（比如 `mapred.cache.files`、`mapred.cache.archives`、`mapred.job.classpath.files`、`mapred.create.symlink` 等），需将之转换成 Protocol Buffers 对象的定义方式，如图 7-6 所示，具体如下（YARN MapReduce 客户端将作业配置文件 `job.xml`、split 文件可见性设置为 APPLICATION）：

```
private LocalResource createApplicationResource(FileContext fs, Path p,
LocalResourceType type)
    throws IOException {
    LocalResource rsrc = recordFactory.newRecordInstance(LocalResource.class);
    FileStatus rsrcStat = fs.getFileStatus(p);
    rsrc.setResource(ConverterUtils.getYarnUrlFromPath(fs
        .getDefaultFileSystem().resolvePath(rsrcStat.getPath())));
    rsrc.setSize(rsrcStat.getLength());
    rsrc.setTimestamp(rsrcStat.getModificationTime());
    rsrc.setType(type);
    rsrc.setVisibility(LocalResourceVisibility.APPLICATION);
    return rsrc;
}
```

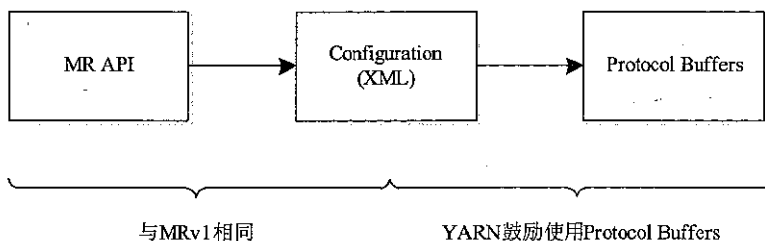


图 7-6 YARN 内部格式转换

注意 在 MRv1 中，与作业相关的所有属性都是用 Configuration 对象表示的，它是一种 key/value 表示方式，用户可以定义任意名称的属性和属性值，它们会被保存到 `job.xml` 文件中，被传递到各个节点上，因此可通过这种方式为作业设置全局变量。然而，由于这种表示方式过于灵活和未进行规范化，从而导致 MRv1 的属性越来越多，且属性名定义非常随意，给 Hadoop 运维和管理带来了很大麻烦。为了克服 Configuration 带来的这些问题，YARN 鼓励用户改用 Protocol Buffers 格式（为了与 MRv1 兼容，YARN 上的 MapReduce 仍采用 Configuration 表示方式），不同服务之间的信息传递最好转换成 Protocol Buffers 对象。

7.3.2 分布式缓存实现

前面提到，按可见性 YARN 将资源分为 3 类，分别是 PUBLIC、PRIVATE 和 APPLICATION，不同可见性的资源开放给用户的权限不同，这是通过设置特殊的目录位置和目录权限实现的，具体如图 7-7 所示。NodeManager 采用轮询分配策略将这三类资源存放在 `yarn.nodemanager.local-dirs` 指定的目录列表中，在每个目录中，资源将按照以下方式存放：

- PUBLIC 资源：存放在 $\${yarn.nodemanager.local-dirs}/filecache/$ 目录下，每个资源将单独存放在以一个随机整数命名的目录中，且目录的访问权限均为 0755。
- PRIVATE 资源：存放在 $\${yarn.nodemanager.local-dirs}/usercache/\${user}/filecache/$ 目录下（其中， $\${user}$ 是应用程序提交者，默认情况下，均为 NodeManager 启动者），每个资源将单独存放在以一个随机整数命名的目录中，且目录的访问权限均为 0710。
- APPLICATION 资源：存放在 $\${yarn.nodemanager.local-dirs}/usercache/\${user}/\${appcache}/\${appid}/filecache/$ 目录下（其中， $\${appid}$ 是应用程序 ID），每个资源将单独存放在以一个随机整数命名的目录中，且目录的访问权限均为 0710；

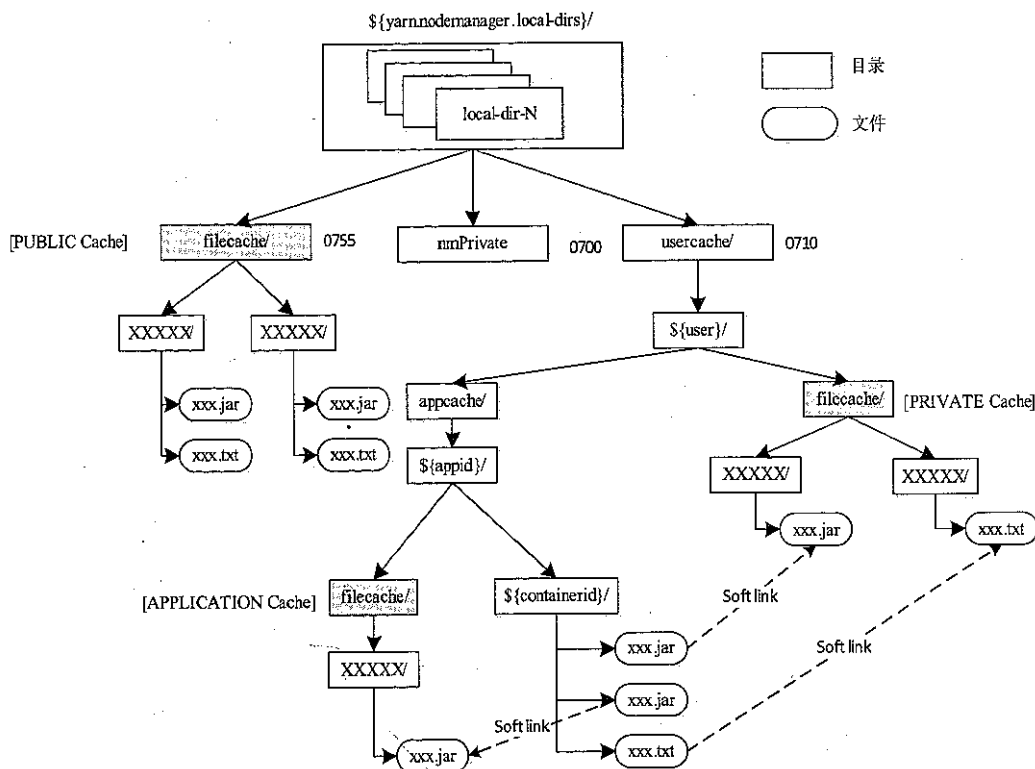


图 7-7 分布式缓存目录结构组织方式

另外，Container 的工作目录位于 $\${yarn.nodemanager.local-dirs}/usercache/\${user}/\${appcache}/\${appid}/\${containerid}/$ 目录下（其中， $\${containerid}$ 是 Container ID），它运行所需的外部资源，比如 JAR 包、字典文件等，处于各个 filecache 目录中，为了避免文件复制带来性能影响，它会建立一个到这些文件的软连接。

不同可见性资源的缓存机制实现不同：对于 PUBLIC 资源，由公共服务 ResourceLocalizationService 中的一个公用线程 PublicLocalizer 下载，它内部维护了一个线程池并行

下载资源；对于 PRIVATE 和 APPLICATION 资源，则由公共服务 ResourceLocalizationService 中一个专门线程 LocalizerRunner（一个 Container 对应一个 LocalizerRunner 线程）下载，同一个 Container 的所有资源是串行下载的^①。考虑到 PRIVATE 和 APPLICATION 资源通常是用户或者应用程序私有的，不允许其他用户看到，因此对应的目录需有严格的权限设置，这是由 ContainerExecutor 组件设置的（目前有 DefaultContainerExecutor 和 LinuxContainerExecutor 两种实现）。

ResourceLocalizationService 内部启动了一个（实现了 LocalizationProtocol 协议的）RPC 服务器，而 ContainerExecutor 则启动了一个资源下载客户端 ContainerLocalizer，它不断地从服务端获取待下载资源信息（ResourceLocalizationService 维护了各种待下载资源列表），然后下载到被设置了特定权限的目录中，具体过程如图 7-8 所示。

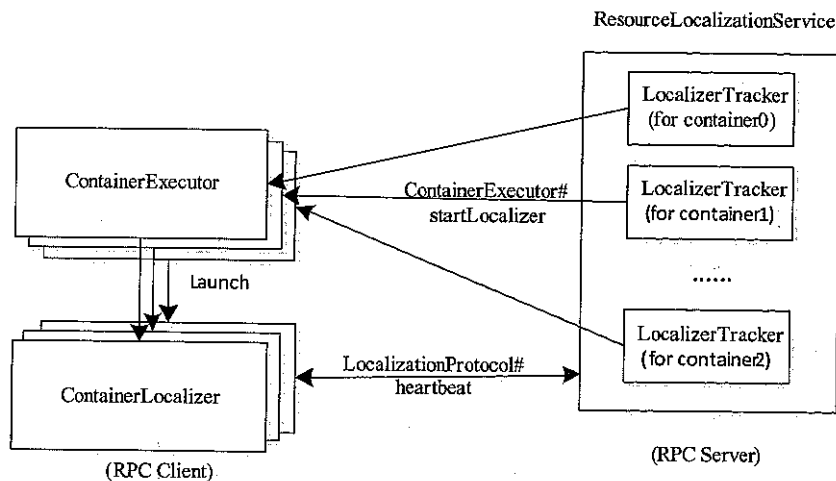


图 7-8 NodeManager 资源下载流程

考虑到分布式缓存完成的主要功能是文件下载，涉及大量的磁盘读写，因此整个过程采用了异步并发模型以加快文件下载速度，同时避免同步模型带来的性能开销。

为了避免缓存的文件过多导致磁盘“撑爆”，NodeManager 会定期清理过期的缓存文件，具体方法如下：每隔一定时间 `yarn.nodemanager.localizer.cache.cleanup.interval-ms`（单位是毫秒，默认值是 $10 \times 60 \times 1000$ ，即 10 分钟）启动一次清理工作，确保每个缓存目录中文件容量小于 `yarn.nodemanager.localizer.cache.target-size-mb`（单位是 MB，默认是 10240，即 10GB），如果超过该值，则采用 LRU（Least Recently Used）算法清除已不再使用的缓存文件，直至文件容量低于设定值。

① 目前是这样实现的，以后可能改为并行下载（已经加入 TODO 列表中）。如果同节点上多个 Container 所需的部分资源是相同的，则这些 Container 会并行下载这些文件。

7.4 目录结构管理

NodeManager 上的 Container 运行任务时通常会将一些临时数据写到本地磁盘上（比如在 MapReduce 计算过程中，Map Task 要将大量中间数据写入本地磁盘，而这些数据不存在备份，一旦丢失后，必须重新计算），且不同 Container 之间往往并行向磁盘写数据，这会因占用大量 IO 资源进而相互干扰。为了避免以上这些问题，尽量提高写数据的可靠性和并发写性能，YARN 允许 NodeManager 配置多个挂在不同磁盘的目录作为中间结果存放目录。对于任意一个应用程序，YARN 会在每个磁盘中创建相同的目录结构，然后采用轮询策略使用这些目录。

NodeManager 上的目录可分为两种：数据目录和日志目录，其中数据目录用于存放执行 Container 所需的数据（比如可执行程序或 JAR 包、配置文件等）和运行过程中产生的临时数据，由参数 `yarn.nodemanager.local-dirs` 指定，而日志目录则用于存放 Container 运行时输出日志，由参数 `yarn.nodemanager.log-dirs` 指定。下面我们分别介绍这两种目录的组织方式。

7.4.1 数据目录管理

假设某个 NodeManager 上通过参数 `yarn.nodemanager.local-dirs` 配置了 N 个目录 `/mnt/disk0/mnt/disk1, ..., /mnt/diskN-1`，且这 N 个目录正好挂在了 N 个不同的磁盘，某一时刻用户提交了一个 ID 为 `appid` 的应用程序，该应用程序需要 K 个 Container，则 NodeManager 为该作业创建的目录结构如图 7-9 所示。NodeManager 在每个磁盘上为该作业创建了相同的目录结构，且采用轮询的调度方式将目录（磁盘）分配给不同的 Container 的不同模块以避免干扰。比如：一个应用程序需要创建工作目录 `work` 和输出结果目录 `output`，为了分摊写负载，NodeManager 可能将 `work` 目录调度到 `/mnt/disk1/` 磁盘上，而将 `output` 目录调度到 `/mnt/disk2` 磁盘上。

考虑到一个应用程序的不同 Container 之间可能存在依赖（比如一个 Reduce Task 要从所有 Map Task 中获取部分输入），为了避免提前清除已经运行完成的 Container 输出的中间数据破坏应用程序的设计逻辑，YARN 统一规定，只有当应用程序运行结束后，才统一清除 Container 产生的中间数据。

7.4.2 日志目录管理

同数据目录一样，YARN 也允许管理员通过参数 `yarn.nodemanager.log-dirs` 配置多个日志目录以分摊负载。需要注意的是，该参数设置的日志目录是 Container 运行日志，而不是 NodeManager 服务产生的日志。NodeManager 服务日志存放在 `$YARN_HOME/logs` 下，命名形式为：`yarn-${user}-nodemanager-${host}.log`，其中 `${user}` 为 NodeManager 服务的启动用户，`${host}` 为 NodeManager 服务所在的节点 `host`。

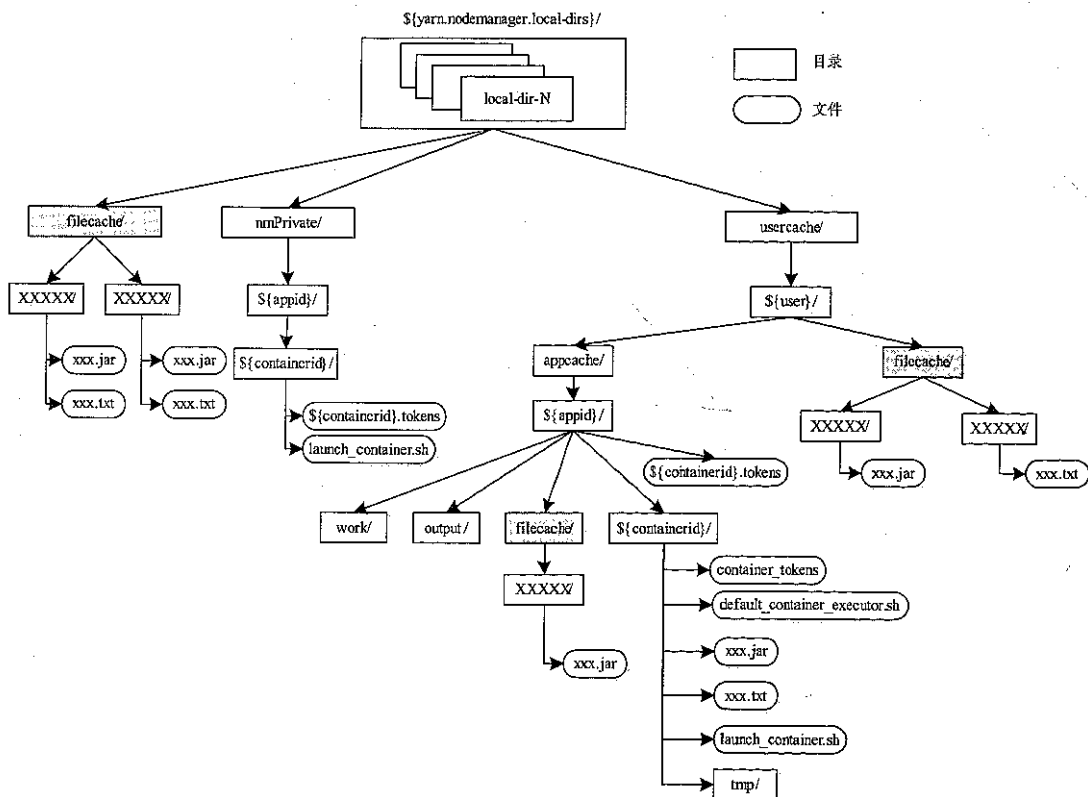


图 7-9 数据目录组织方式

1. 日志目录结构

相比于数据目录结构，日志目录的结构简单很多，如图 7-10 所示，NodeManager 会在所有目录上为同一个应用程序建立相同的目录结构，并采用轮询的调度方式将这些目录分配给不同 Container 使用。每个 Container 将输出三类日志：

- stdout：使用标准输出函数打印的日志，比如 Java 中的 System.out.print 输出的内容。
- stderr：标准错误输出产生的日志信息。
- syslog：使用 log4j 打印的日志信息，这是最常用的打印日志方式，默认情况下，YARN 采用了这种方式打印日志，换句话说，通常情况下，只有这个文件中有内容，其他两个文件为空。

2. 日志清理机制

由于 NodeManager 将所有 Container 的运行日志保存到本地磁盘上，因此，随着时间的积累，日志必将越来越多。为了避免大量 Container 日志“撑爆”磁盘空间，NodeManager 将定期清理日志文件，该功能由组件 LogHandler（当前存在两种实现：NonAggregatingLogHandler 和 LogAggregationService）完成。总起来说，NodeManager 提供

了定期删除（由 NonAggregatingLogHandler 实现）和日志聚集转存（由 LogAggregation-Service 实现）两种日志清理机制，默认情况下，采用的是定期删除机制。

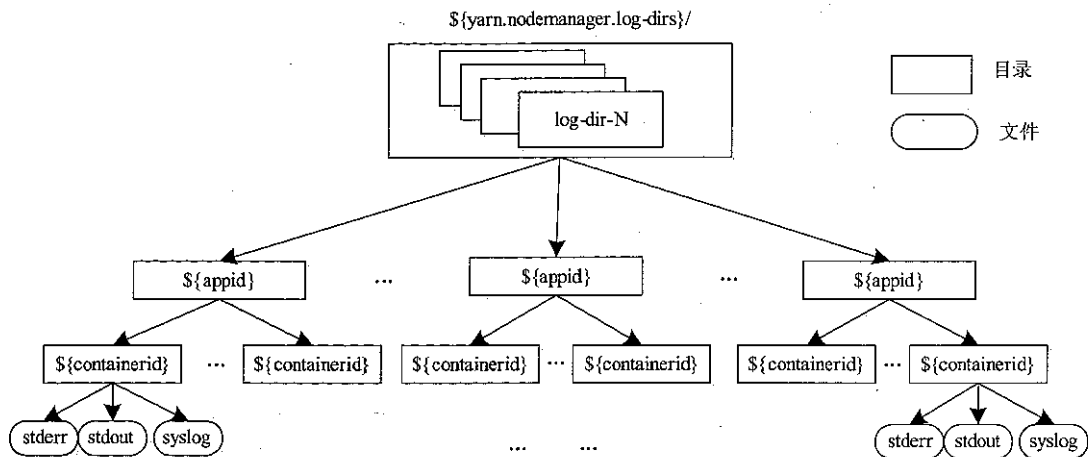


图 7-10 日志目录组织方式

（1）定期删除

NodeManager 允许一个应用程序日志在磁盘上的保留时间为 `yarn.nodemanager.log.retain-seconds`（单位是秒，默认为 $3 \times 60 \times 60$ ，即 3 小时），一旦超过该时间，NodeManager 会将该应用程序所有日志从磁盘上删除。

（2）日志聚集转存

除定期删除外，NodeManager 还提供了另一种日志处理方式——日志聚集转存^①，管理员可通过将配置参数 `yarn.log-aggregation-enable` 置为 `true` 启用该功能。该机制将 HDFS 作为日志聚集仓库，它将应用程序产生的日志上传到 HDFS 上，以便统一管理和维护。该机制由两阶段组成：文件上传和文件生命周期管理。

文件上传

当一个应用程序运行结束时，它产生的所有日志将被统一上传到 HDFS 上的 `${remoteRootLogDir}/${user}/${suffix}/${appid}` 目录中（`${remoteRootLogDir}` 值由参数 `yarn.nodemanager.remote-app-log-dir` 指定，默认是 `/tmp/logs`；`${user}` 为应用程序所有者；`${suffix}` 值由参数 `yarn.nodemanager.remote-app-log-dir-suffix` 指定，默认是 `logs`；`${appid}` 为应用程序 ID），且同一个节点中所有日志保存到该目录中的同一个文件，这些文件以节点 ID 命名。一个示例如图 7-11 所示。

一旦日志全部上传到 HDFS 后，本地磁盘上的日志文件将被删除。此外，为了减少不必要的日志上传，NodeManager 允许用户指定要上传的日志类型。当前支持的日志类型有三种：`ALL_CONTAINERS`（上传所有 Container 日志）、`APPLICATION_MASTER_ONLY`（仅上传 ApplicationMaster 产生的日志）和 `AM_AND_FAILED_CONTAINERS_ONLY`（上传

^① 参见网址 <https://issues.apache.org/jira/browse/MAPREDUCE-3143>。

ApplicationMaster 和运行失败的 Container 产生的日志), 默认情况下采用 ALL_CONTAINERS。

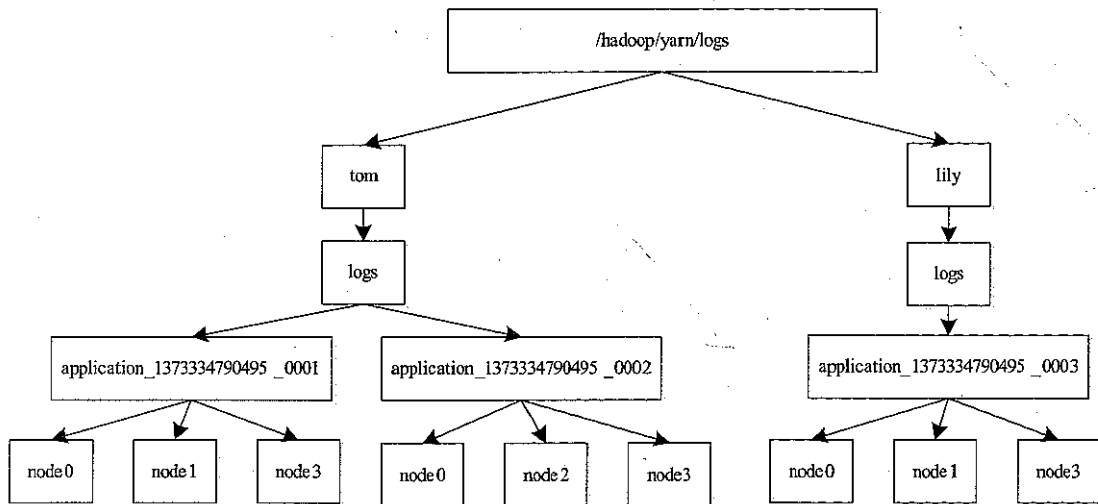


图 7-11 日志目录在 HDFS 上组织方式

文件生命周期管理

转存到 HDFS 上的日志的生命周期不再由 NodeManager 负责, 而是由 JobHistory 服务管理。比如对于 MapReduce 计算框架而言^①, 它专有的 JobHistory 负责定期清理 MapReduce 作业转存到 HDFS 上的日志, 每个日志文件最多存留时间为 yarn.log-aggregation.retain-seconds (单位是秒, 默认为 $3 \times 60 \times 60$, 即 3 小时)。

用户可通过两种方式查看应用程序日志, 一种是通过 NodeManager 的 Web 界面; 另一种是通过 Shell 命令查看。

查看一个应用程序产生的所有日志, 命令如下:

```
bin/yarn logs -applicationId application_1304487270789_0001
```

查看一个 Container 产生的日志, 命令如下:

```
bin/yarn logs -applicationId application_1304487270789_0001 -containerId container_1304487270789_0001_000002 -nodeAddress 127.0.0.1_45454
```

7.5 状态机管理

NodeManager 维护了三类状态机, 分别是 Application、Container 和 LocalizedResource, 它们均直接或者间接参与维护一个应用程序的生命周期。当 NodeManager 收到来自某个应用程序的第一个 Container 启动命令时, 会创建一个 Application 状态机跟踪该应用程序在

^① 社区正在开发通用的 JobHistory 服务 (不仅限于 MapReduce 使用), 这样一个 YARN 集群只需启动一个 JobHistory 服务即可, 具体参考 <https://issues.apache.org/jira/browse/YARN-321>。

该节点上的生命周期，而每个 Container 的运行过程同样由一个状态机维护（见图 7-12）。此外，运行 Application 所需的资源（比如文本文件、JAR 包、归档文件等）需从 HDFS 上下载，每个资源的下载过程均由一个状态机 LocalizedResource 维护和跟踪。本节将详细介绍这三类状态机。

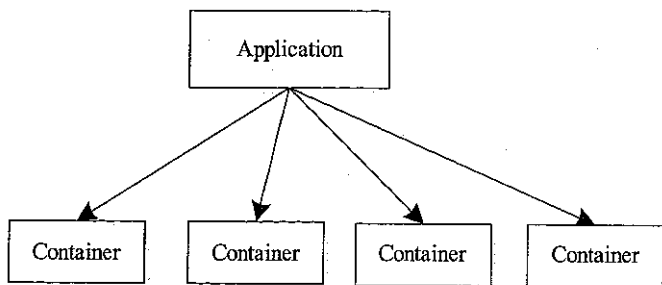


图 7-12 Application 与 Container 关系

7.5.1 Application 状态机

NodeManager 上 Application 维护的信息是 ResourceManager 端 Application 信息的子集，这有利于对一个节点上同一个 Application 的所有 Container 进行统一管理（比如记录每个 Application 运行在该节点的 Container 列表，杀死一个 Application 所有 Container 等）。实际的实现类是 ApplicationImpl，该类维护了一个 Application 状态机，记录了 Application 可能存在的各个状态以及导致状态间转换的事件。当某个事件发生时，ApplicationImpl 会根据实际情况进行节点状态转移，同时触发一个行为。需要注意的是，NodeManager 上 Application 生命周期与 ResourceManager 上 Application 的生命周期是一致的。

如图 7-13 所示，在 NodeManager 看来，每个应用程序有 6 种基本状态（Application-State）和 9 种导致这 6 种状态之间发生转移的事件（ApplicationEventType），ApplicationImpl 的作用是等待接收其他对象发出的 ApplicationEventType 类型的事件，然后根据当前状态和事件类型，将当前状态转移到另外一种状态，同时触发另外一种行为（实际上执行一个函数，该函数可能会再次发出一种其他类型的事件）。

下面进行具体介绍。

（1）基本状态

基本状态包括：

- NEW：状态机初始状态，每个 Application 对应一个状态机，而每个状态机的初始状态均为 NEW。
- INITING：Application 处于初始化状态，即创建日志目录和工作目录，创建 AppLog-Aggregator 对象等。
- FINISHING_CONTAINERS_WAIT：等待回收 Container 占用的资源时所处的状态。当 Application 状态机收到 FINISH_APPLICATION 事件后，会向各个 Container 发

送 KILL 命令以回收它们占用的资源。

- **APPLICATION_RESOURCES_CLEANINGUP** : Application 的所有 Container 占用的资源被收回后, 它将处于 **APPLICATION_RESOURCES_CLEANINGUP** 状态。
- **RUNNING** : Application 初始化 (完成创建日志目录和工作目录, 创建 AppLog-Aggregator 等工作) 完成后, 将进入 **RUNNING** 状态。
- **FINISHED** : Application 将占用的各种文件资源发送给文件删除服务 DeletionService (该服务会异步删除文件, 避免产生性能问题) 后, 进入 **FINISHED** 状态, 表示运行完成。

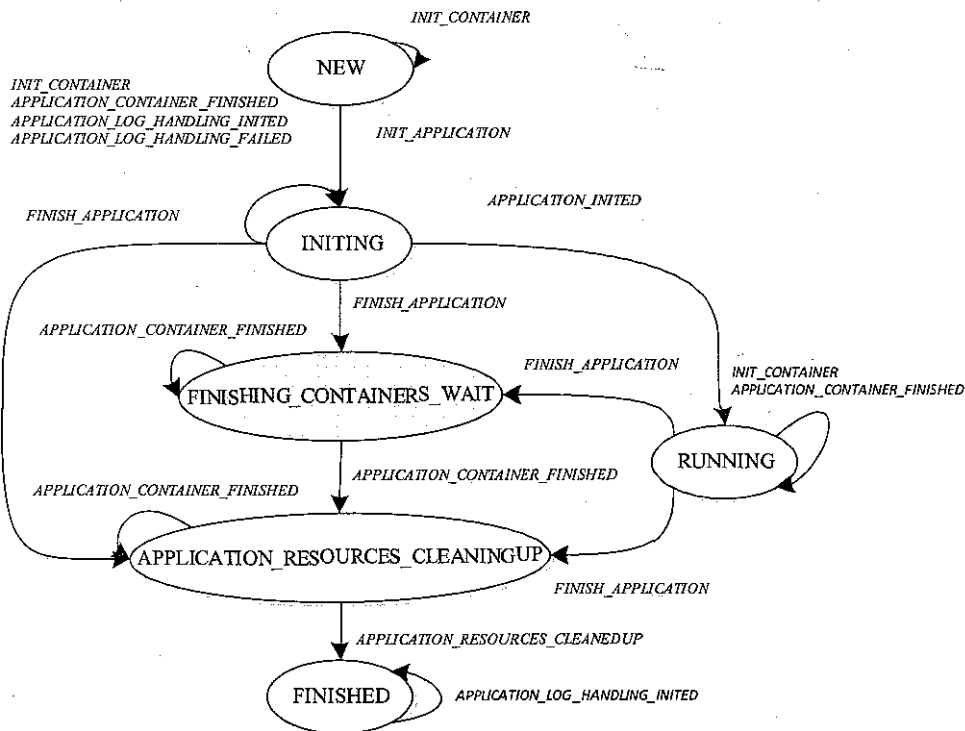


图 7-13 Application 状态机

(2) 基本事件

基本事件包括:

- **INIT_APPLICATION** : NodeManager 收到来自某个 Application 的第一个 Container 后, 会触发一个 **INIT_APPLICATION** 事件, 同时使 Application 状态由初始状态 **NEW** 转换为 **INITING**。
- **APPLICATION_INITED** : Application 初始化完成后将触发一个 **APPLICATION_INITED** 事件。Application 初始化主要工作是初始化各类必需的服务组件 (比如日志记录组件 LogHandler、资源状态追踪组件 LocalResourcesTrackerImpl 等), 供后

续 Container 使用，通常由 Application 的第一个 Container 完成。

- **FINISH_APPLICATION** : NodeManager 收到 ResourceManager 发送的待清理的 Application 列表后，会向这些 Application 发送一个 FINISH_APPLICATION 事件。
- **APPLICATION_CONTAINER_FINISHED** : 该 Application 的一个 Container 运行完成（可能运行失败，也可能运行成功）后将触发一个 APPLICATION_CONTAINER_FINISHED 事件。
- **APPLICATION_RESOURCES_CLEANEDUP** : Application 所有 Container 占用的资源被清理完成（比如占用的临时目录）后将触发一个 APPLICATION_RESOURCES_CLEANEDUP 事件。
- **INIT_CONTAINER** : NodeManager 收到 ApplicationMaster 通过 RPC 函数 ContainerManagementProtocol#startContainer 发送的启动 Container 的请求后，会触发一个 INIT_CONTAINER 事件。
- **APPLICATION_LOG_HANDLING_INITED** : Application 日志记录句柄（供后续各个 Container 记录日志使用）创建成功后，将触发一个 APPLICATION_LOG_HANDLING_INITED 事件，以驱使 Application 进入 RUNNING 状态。
- **APPLICATION_LOG_HANDLING_FINISHED** : Application 运行完成，资源得到回收后，会触发一个 APPLICATION_LOG_HANDLING_FINISHED 事件，以销毁 Application 日志记录句柄。
- **APPLICATION_LOG_HANDLING_FAILED** : Application 日志记录句柄初始化失败后将触发该事件，Application 收到该事件后，将不再记录日志，而是直接进行后续工作。

图 7-14 描述了以上各个事件的来源。

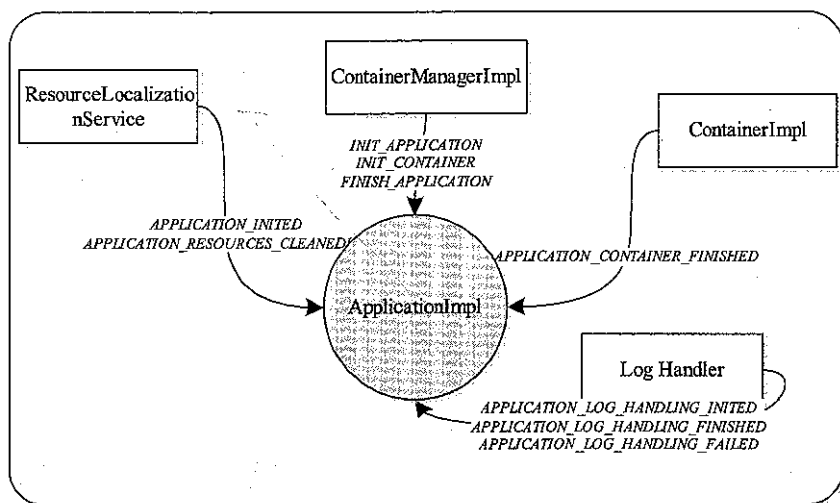


图 7-14 Application 状态机事件来源

7.5.2 Container 状态机

Container 是 NodeManager 中用于维护一个 Container 生命周期的数据结构，它的实现是 ContainerImpl，该类维护了一个 Container 状态机，记录了 Container 可能存在的各个状态以及导致状态间转换的事件，当某个事件发生时，ContainerImpl 会根据实际情况进行节点状态转移，同时触发一个行为。

如图 7-15 所示，在 NM 看来，每个 Container 有 11 种基本状态（ContainerState）和 10 种导致这 11 种状态之间发生转移的事件（ContainerEventType）^①，ContainerImpl 的作用是等待接收其他对象发出的 ContainerEventType 类型的事件，然后根据当前状态和事件类型，将当前状态转移到另外一种状态，同时触发另外一种行为（实际上执行一个函数，该函数可能会再次发出一种其他类型的事件）。

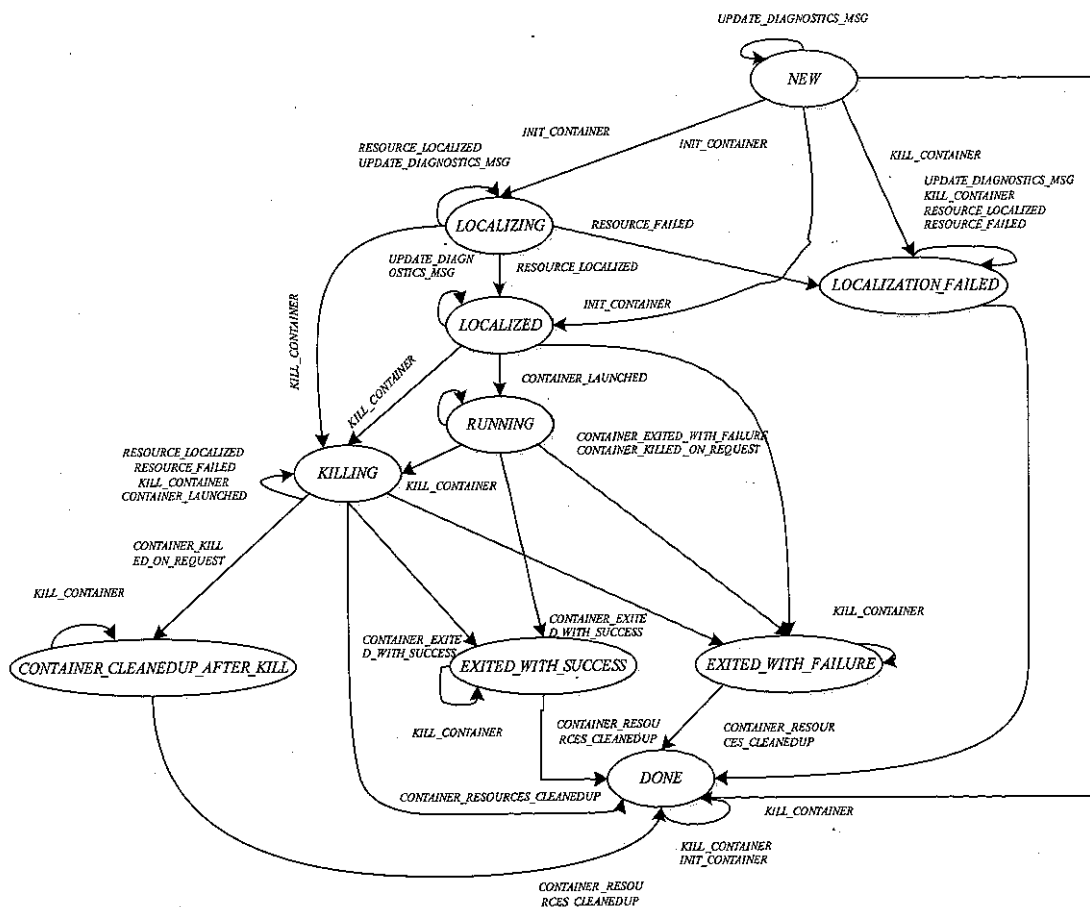


图 7-15 Container 状态机

⊖ 截至本书截稿时，状态 `CONTAINER_RESOURCES_CLEANINGUP` 和事件 `CONTAINER_DONE` 与 `CONTAINER_INITED` 有定义但尚未使用，因此这里不对它们进行介绍。

下面进行具体介绍。

(1) 基本状态

基本状态包括：

- **NEW**：状态机初始状态，每个 Container 对应一个状态机，而每个状态机的初始状态均为 NEW。
- **LOCALIZING**：Container 运行之前，需从 HDFS 上下载依赖的文件资源，Container 正在下载文件时所处的状态称为 LOCALIZING。
- **LOCALIZED**：运行 Container 所需的文件资源已经全部下载（缓存）到本地后，将进入该状态。
- **LOCALIZATION_FAILED**：由于文件损坏、磁盘损坏等原因，Container 下载资源失败（这将导致依赖该资源的所有 Container 运行失败），此时 Container 所处的状态为 LOCALIZATION_FAILED。
- **RUNNING**：ContainerLaunch 组件为 Container 创建工作目录和构造执行脚本，并通知 ContainerExecutor 执行该脚本，使得 Container 进入 RUNNING 状态。
- **EXITED_WITH_SUCCESS**：ContainerExecutor 启动 Container 执行脚本后，阻塞直到脚本正常退出执行，此时 Container 将处于 EXITED_WITH_SUCCESS 状态。
- **DONE**：Container 正常退出执行后，首先需清理它占用的各种临时文件，一旦清理完成后，Container 状态将转移为完成状态 DONE。
- **KILLING**：Container 正在被杀死时所处的状态，通常是由于内存超量使用被监控线程杀死，或者 ResourceManager 和 ApplicationMaster 主动杀死 Container。
- **EXITED_WITH_FAILURE**：Container 在执行过程中异常退出后所处的状态，通常是由于 Container 内部原因导致，比如程序 bug、硬件故障等。
- **CONTAINER_CLEANEDUP_AFTER_KILL**：Container 被杀死后所处的状态，处于该状态时，Container 已经异常退出（执行 Shell 命令返回错误编号为 137 或 143）。

(2) 基本事件

基本事件包括：

- **INIT_CONTAINER**：NodeManager 收到来自 ApplicationMaster 的启动 Container 的请求，则会创建一个 Container 对象，并触发一个 INIT_CONTAINER 事件，使 Container 状态由初始状态 NEW 转换为 LOCALIZING。
- **RESOURCE_LOCALIZED**：Container 成功从 HDFS 下载一种资源到本地（缓存），会触发一个 RESOURCE_LOCALIZED 事件。注意，一个 Container 可能需要下载多种资源，因此，该事件可能使 Container 维持在 LOCALIZING 状态或者进入新状态 LOCALIZED（所有资源均下载完成）。
- **CONTAINER_LAUNCHED**：ContainerLaunch 调用函数 ContainerExecutor#launch Container 成功启动后，会触发一个 CONTAINER_LAUNCHED 事件，使得 Container 从 LOCALIZED 状态转换为 CONTAINER_LAUNCHED 状态。需要注意的，由于函

数 `ContainerExecutor#launchContainer` 是阻塞的，所以它要等到 Container 退出执行后才会退出，因此，该事件将在该函数调用之后发出。

- **CONTAINER_EXITED_WITH_SUCCESS**：Container 正常退出（执行 Container 实际上是执行一个 Shell 脚本，正常结束运行后会返回 0），会触发一个 **CONTAINER_EXITED_WITH_SUCCESS** 事件。
- **CONTAINER_RESOURCES_CLEANEDUP**：NodeManager 清理完成 Container 使用的各种临时目录（主要是删除分布式缓存中的临时数据），此时会触发一个 **CONTAINER_RESOURCES_CLEANEDUP** 事件，使得 Container 从 **EXITED_WITH_SUCCESS** 状态转换为 **DONE** 状态。
- **RESOURCE_FAILED**：Container 本地化过程中抛出异常，会触发一个 **RESOURCE_FAILED** 事件，导致 Container 失败。
- **KILL_CONTAINER**：在多种场景下会触发产生 **KILL_CONTAINER** 事件，包括 ResourceManager 要求 NodeManager 杀死一个 Container；Container 使用的内存量超过约定值，被监控线程杀死；ApplicationMaster 要求 NodeManager 杀死一个 Container（通过 RPC 函数 `ContainerManagementProtocol#stopContainer`）。
- **CONTAINER_EXITED_WITH_FAILURE**：Container 异常退出（运行过程中抛出 Throwable 异常）时，会触发一个 **CONTAINER_EXITED_WITH_FAILURE** 事件。
- **CONTAINER_KILLED_ON_REQUEST**：Container 运行过程中被强制杀死或者终止（返回码为 137 或者 143）时，会触发一个 **CONTAINER_KILLED_ON_REQUEST** 事件。
- **UPDATE_DIAGNOSTICS_MSG**：ContainerExecutor 执行 `launchContainer` 函数启动 Container 过程中抛出 `IOException` 异常，此时会通过 **UPDATE_DIAGNOSTICS_MSG** 事件告诉 Container 状态机。

图 7-16 描述了以上各个事件的来源。

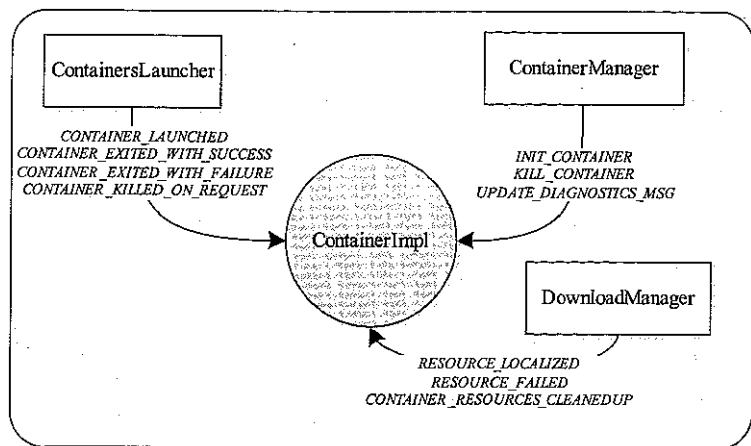


图 7-16 Container 状态机的事件来源

7.5.3 LocalizedResource 状态机

LocalizedResource 是 NodeManager 中用于维护一种“资源”（指文本文件、JAR 包、归档文件等外部文件资源）生命周期的数据结构，它维护了一个状态机，记录了“资源”可能存在的各个状态以及导致状态间转换的事件。当某个事件发生时，LocalizedResource 会根据实际情况进行节点状态转移，同时触发一个行为。

如图 7-17 所示，在 NM 看来，每个资源有 4 种基本状态（ResourceState）和 4 种导致这 4 种状态之间发生转移的事件（ResourceEventType），LocalizedResource 的作用是等待接收其他对象发出的 ResourceEventType 类型的事件，然后根据当前状态和事件类型，将当前状态转移到另外一种状态，同时触发另外一种行为（实际上执行一个函数，该函数可能会再次发出一种其他类型的事件）。

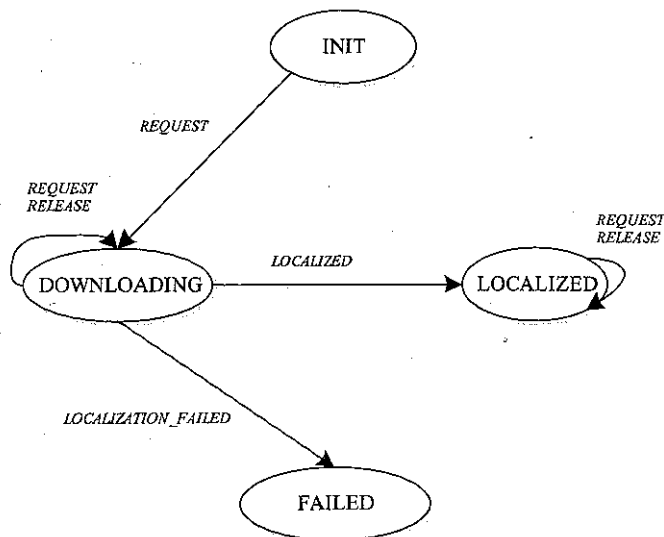


图 7-17 LocalizedResource 状态机

下面具体进行介绍：

(1) 基本状态

基本状态包括：

- INIT：状态机初始状态，每种资源对应一个状态机，而每个状态机的初始状态均为 INIT。
- DOWNLOADING：资源处于下载状态，即调用 `FileSystem#copyToLocalFile` 将文件下载保存到临时文件 `XXX.tmp` 中，下载完成后重命名成最终文件名 `XXX`。
- LOCALIZED：资源下载完成时所处的状态。
- FAILED：资源下载失败时（资源下载过程中抛出 `ExecutionException` 异常）所处的状态。

(2) 基本事件

基本事件包括：

- **REQUEST**：当需要为 Container 下载某种资源时（比如 JAR 包或者字典文件，这些文件一般位于 HDFS 上，需要在 Container 执行前下载到本地），会发出一个 REQUEST 事件。
- **LOCALIZED**：一种资源下载成功后，ResourceLocalizationService 会触发一个 LOCALIZED 事件。
- **RELEASE**：当 Container 执行完成（可能成功或者失败）后，会触发一个 RELEASE 事件，以清理该资源（通常是删除文件或者目录）。
- **LOCALIZATION_FAILED**：如果资源下载过程中抛出 ExecutionException 异常，则 ResourceLocalizationService 会触发一个 LOCALIZATION_FAILED 事件。

以上 4 类事件全部来自资源下载服务 ResourceLocalizationService。

7.6 Container 生命周期剖析

Container 启动命令是由各个 ApplicationMaster 通过 RPC 函数 ContainerManagementProtocol#startContainer 向 NodeManager 发起的，NodeManager 中的 ContainerManager 组件（组件实现为 ContainerManagerImpl）负责接收并处理该请求。Container 启动过程主要经历三个阶段：资源本地化、启动并运行 Container 和资源清理。

- **资源本地化**主要指前面提到的分布式缓存机制完成的工作，功能包括初始化各种服务组件、创建工作目录、从 HDFS 下载运行所需的各种资源（比如文本文件、JAR 包、可执行文件）等。资源本地化主要由两部分组成，分别是应用程序初始化和 Container 本地化。其中，应用程序初始化的主要工作是初始化各类必需的服务组件（比如日志记录组件 LogHandler、资源状态追踪组件 LocalResourcesTrackerImpl 等），供后续 Container 使用，通常由 Application 的第一个 Container 完成；Container 本地化则是创建工作目录，从 HDFS 下载各类文件资源。
- **Container 启动**是由 ContainersLauncher 服务完成的，该服务将进一步调用插拔式组件 ContainerExecutor，YARN 提供了两种 ContainerExecutor 实现，一种是 DefaultContainerExecutor，另一种是 LinuxContainerExecutor。
- **资源清理**则是资源本地化的逆过程，它负责清理各种资源，它们均由 ResourceLocalizationService 服务完成的。

本节将依次介绍这三个阶段。

7.6.1 Container 资源本地化

Container 资源本地化是指准备 Container 运行所需的环境，包括创建 Container 工作目录、从 HDFS 下载运行 Container 所需的各种资源（JAR 包、可执行文件等）等，这是通过

YARN 提供的分布式缓存机制实现的, 主要内部机理已在 7.3 节进行了详细分析, 本节重点介绍整个工作流程。

前面提到, YARN 将资源分为 PUBLIC、PRIVATE 和 APPLICATION 三类, 不同级别的资源对不同用户和应用程序的访问权限不同, 这也直接导致资源的本地化方式不同。尽管它们的本地化由 ResourceLocalizationService 服务完成, 但内部由不同线程负责下载, 即所有应用程序的 PUBLIC 资源由 ResourceLocalizationService 内部专门的线程 PublicLocalizer 下载完成, 该线程内部维护了一个线程池以加快资源下载速度, 而每个应用程序的每种 PRIVATE 或 APPLICATION 资源分别由一个专门的线程 LocalizerRunner 负责下载。该线程将启动一个资源下载客户端 ContainerLocalizer, 该客户端通过 RPC 协议 LocalizationProtocol 访问 ResourceLocalizationService 服务, 以获取待下载的资源。

7.3 节同样提到: “考虑到分布式缓存完成的主要功能是文件下载, 涉及大量的磁盘读写, 因此整个过程采用了异步并发模型加快文件下载速度, 以避免同步模型带来的性能开销。” 接下来详细介绍 Container 资源本地化的整个流程, 以方便读者进一步理解基于事件驱动的异步并发机制。为了更清楚地讲解整个流程, 本节分两种情况讲解: 一种是该 Container 是 ApplicationMaster 发送到该节点的第一个 Container; 另一种则不是第一个 Container。

情况 1: 来自 ApplicationMaster 的第一个 Container

来自 ApplicationMaster 的第一个 Container 在本地化之前, 还要负责应用程序的初始化 (比如创建 ApplicationImpl 和 LogHandler 对象), 整个过程如图 7-18 所示, 其中步骤 1~6 属于应用程序初始化工作, 后面几个步骤完成 Container 本地化工作, 具体如下。

步骤 1 ContainerManagerImpl 收到来自 ApplicationMaster 的启动 Container 的请求后, 首先为该 Container 创建生命周期管理对象 ContainerImpl, 然后判断该 Container 是否是迄今为止来自该应用程序的第一个。如果是, 则为该应用程序创建一个 ApplicationImpl 对象, 并向该对象发送一个 INIT_APPLICATION 事件, 同时再向它发送一个 INIT_CONTAINER 事件。

步骤 2 ApplicationImpl 收到 INIT_APPLICATION 事件后, 设置 ACL 属性, 并进一步向 LogHandler (目前有两种实现, 分别是 LogAggregationService 和 NonAggregating-LogHandler, 具体已在 7.4.2 节进行了介绍) 发送一个 APPLICATION_STARTED 事件, 之后, 状态由 NEW 转换为 INITING。ApplicationImpl 收到 INIT_CONTAINER 事件后, 将该 Container 加入它的 Container 维护列表中, 不再有后续操作 (此时 Application 处于 INITING 状态)。

步骤 3 LogHandler 收到 APPLICATION_STARTED 事件后, 将创建应用程序日志目录、设置目录权限等, 具体跟采用的插拔式 LogHandler 实现有关, 但不管是哪种实现, 处理完相应的操作后, 都要向 ApplicationImpl 发送一个 APPLICATION_LOG_HANDLING_INITED 事件。

步骤 4 ApplicationImpl 收到 APPLICATION_LOG_HANDLING_INITED 事件后, 直接向

源,并保存到 ContainerImpl 中特定的数据结构中,之后向 ResourceLocalizationService 发送 INIT_CONTAINER_RESOURCES 事件(该事件中封装了该 Container 需要下载的所有资源)。至此,ContainerImpl 运行状态由 NEW 转换为 LOCALIZING。

步骤 8 ResourceLocalizationService 收到 INIT_CONTAINER_RESOURCES 事件后,依次将该 Container 所需的每个资源单独封装成一个 REQUEST 事件,发送给对应的资源状态追踪器 LocalResourcesTrackerImpl(它是在步骤 5 中创建的,不同可见性的资源具有不同的状态追踪器)。

步骤 9 LocalResourcesTrackerImpl 收到 REQUEST 事件后,将为对应的资源创建一个状态机对象 LocalizedResource 以跟踪资源的生命周期,并将 REQUEST 事件进一步传送给该 LocalizedResource。需要注意的是,此处会判断是否已经为该资源创建了 LocalizedResource 对象(比如 MapReduce 中所有 MapTask 需要的资源均相同,创建一个即可)以避免重复创建,两种资源是否相同的判断依据是资源位置、资源类型、最近修改时间和 PATTERN 四个属性是否相等(具体参见 7.3.1 节)。

步骤 10 LocalizedResource 收到 REQUEST 事件后,检查资源当前所处状态执行不同的操作,如果是 LOCALIZED,则表示资源已经下载完成,直接向 ContainerImpl 发送一个 RESOURCE_LOCALIZED 事件,不再执行下面的步骤;如果是 INIT 或者 DOWNLOADING 状态^①,则将待下载资源信息通过 REQUEST_RESOURCE_LOCALIZATION 事件发送给资源下载服务 ResourceLocalizationService,之后 LocalizedResource 状态由 NEW 转换为 DOWNLOADING;

步骤 11 ResourceLocalizationService 收到 REQUEST_RESOURCE_LOCALIZATION 事件后,如果是 PUBLIC 资源,则统一交给线程 PublicLocalizer 处理,否则检查是否已经为该 Container 创建了 LocalizerRunner 线程,如果没有,则创建一个,否则直接添加到该线程的下载队列中。该线程将进一步调用函数 ContainerExecutor#startLocalizer 下载资源,在该函数中,它创建了一个资源下载客户端 ContainerLocalizer,该客户端通过协议 LocalizationProtocol 与 ResourceLocalizationService 通信,以顺序获取待下载资源位置并下载之(目前仅支持资源阻塞式串行下载)。待一个资源下载完成后,PublicLocalizer 或者 LocalizerRunner 会向 LocalizedResource 发送一个 LOCALIZED 事件。

步骤 12 LocalizedResource 收到 LOCALIZED 事件后,将进一步向 ContainerImpl 发送一个 RESOURCE_LOCALIZED 事件,之后状态由 DOWNLOADING 转换为 LOCALIZED,至此,一类资源下载完毕。

ContainerImpl 收到 RESOURCE_LOCALIZED 事件后,会检查所依赖的资源是否全部下载完毕,如果是,则向 ContainersLauncher 服务发送一个 LAUNCH_CONTAINER 事件,以运行 Container,同时 ContainerImpl 状态由 LOCALIZING 转换为 LOCALIZED。

① 逻辑上讲,一个处于 DOWNLOADING 状态的资源已经在等待下载,无须重复向 ResourceLocalizationService 发送资源下载请求,尽管目前实现仍会重复发送,但由于资源下载是串行的,所以同一个资源不会重复被下载。

总之，资源本地化过程可概括为：在 NodeManager 上，同一个应用程序的所有 ContainerImpl 异步并发向资源下载服务 ResourceLocalizationService 发送待下载的资源。而 ResourceLocalizationService 下载完一类资源后，将通知依赖该资源的所有 Container。一旦一个 Container 依赖的资源已经全部下载完成，则该 Container 进入运行阶段。

情况 2：来自 ApplicationMaster 的非第一个 Container

相比于第一个 Container，非第一个 Container 的资源本地化过程则简单得多，它不必进行 Application 初始化工作，整个过程如图 7-19 所示。图 7-19 中的步骤 2~8 相当于情况 1 中的步骤 7~12，在此不再具体介绍。需要注意的是，如果非第一个 Container 到达 NodeManager 后，前面的 Container 仍在下载文件，且部分未下载文件也是该 Container 所需要的（比如可见性为 PRIVATE 或者 APPLICATION 的文件），则该 Container 会尝试下载前面 Container 还未开始下载的文件，以加快文件下载速度。

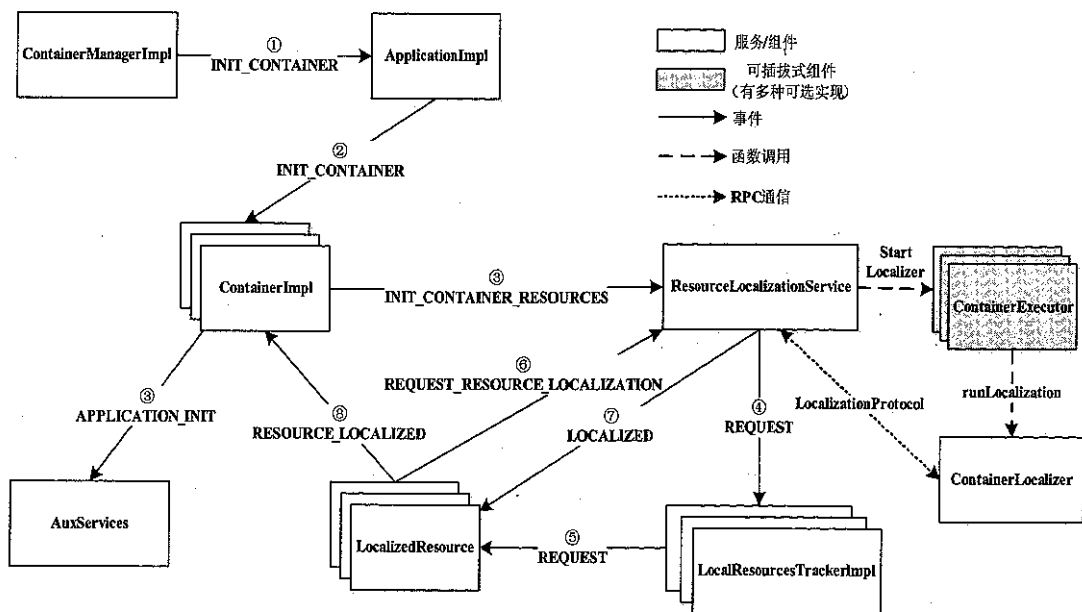


图 7-19 来自 ApplicationMaster 的非第一个 Container 本地化过程

7.6.2 Container 运行

Container 运行是由 ContainersLauncher 服务实现的，主要过程可概括为：将待运行 Container 所需的环境变量和运行命令写到 Shell 脚本 launch_container.sh 中，并将启动该脚本的命令写入 default_container_executor.sh 中，然后通过运行该脚本启动 Container。之所以要将 Container 运行命令写到脚本中并通过运行脚本执行它，主要是直接执行命令可能让一些特殊符号发生转义。一个 Container 运行流程如图 7-20 所示。

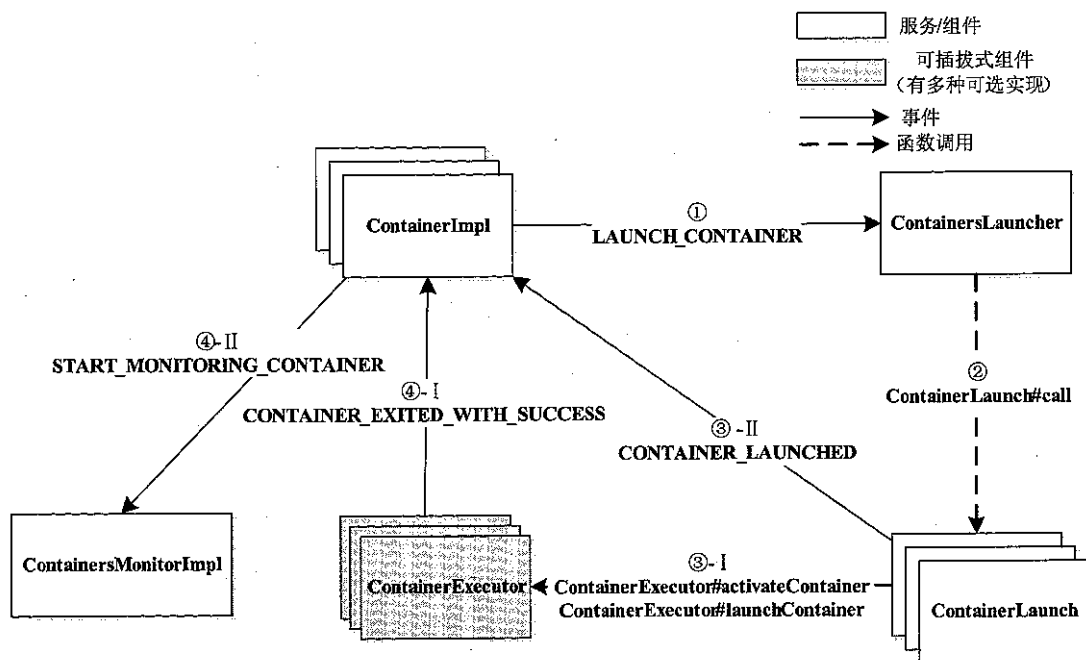


图 7-20 Container 运行过程（步骤 3 和步骤 4 中若干操作是并行执行的）

主要步骤如下。

步骤 1 ContainerImpl 向 ContainersLauncher 服务发送 LAUNCH_CONTAINER 事件，请求启动 Container。

步骤 2 ContainersLauncher 收到 LAUNCH_CONTAINER 事件后，将为该 Container 创建一个 Callable 类型的对象 ContainerLaunch，并放到线程池中执行：为 Container 创建 Tokens 文件和执行脚本 launch_container.sh，并将它们保存到 NodeManager 私有目录 \${yarn.nodemanager.local-dirs}/nmPrivate 中。其中，launch_container.sh 包含执行 Container 所需的环境变量、运行命令等全部信息。以 MapReduce 框架为例，脚本内容如下：

```
#!/bin/bash

export
YARN_LOCAL_DIRS="/mnt/disk/yarn/local/usercache/dongxicheng/appcache/
application_1359695803957_0006"
export NM_HTTP_PORT="8042"
export HADOOP_COMMON_HOME="/opt/hadoop/hadoop-2.0"
export JAVA_HOME="/usr/lib/jvm/java-6-openjdk"
export NM_HOST="yarn002"
export
CLASSPATH="$PWD:$HADOOP_CONF_DIR:$HADOOP_COMMON_HOME/share/hadoop/
common/*:$HADOOP_COMMON_HOME/share/hadoop/common/lib/*:$HADOOP_HDFS_HOME/share/
hadoop/hdfs/*:$HADOOP_HDFS_HOME/share/hadoop/hdfs/lib/*:$YARN_HOME/share/hadoop/
yarn/*:$YARN_HOME/share/hadoop/yarn/lib/*:$YARN_HOME/share/hadoop/mapreduce/*:$YARN_
```

```

HOME/share/hadoop/mapreduce/lib/*:job.jar/:job.jar/classes/:job.jar/lib/*:$PWD/*"
export
HADOOP_TOKEN_FILE_LOCATION="/mnt/disk/yarn/local/usercache/dongxicheng/appcache/
application_1359695803957_0006/container_1359695803957_0006_01_000001/container_
tokens"
export APPLICATION_WEB_PROXY_BASE="/proxy/application_1359695803957_0006"
export YARN_HOME="/opt/hadoop/hadoop-2.0"
export JVM_PID="$ $"
export USER="dongxicheng"
export HADOOP_HDFS_HOME="/opt/hadoop/hadoop-2.0"
export
PWD="/mnt/disk/yarn/local/usercache/dongxicheng/appcache/application_
1359695803957_0006/container_1359695803957_0006_01_000001"
export NM_PORT="41070"
export HOME="/home/"
export LOGNAME="dongxicheng"
export APP_SUBMIT_TIME_ENV="1359700313597"
export HADOOP_CONF_DIR="/opt/hadoop/hadoop-2.0/etc/hadoop"
export MALLOC_ARENA_MAX="4"
export AM_CONTAINER_ID="container_1359695803957_0006_01_000001"
mkdir -p jobSubmitDir
ln -sf
"/mnt/disk/yarn/local/usercache/dongxicheng/appcache/application_
1359695803957_0006/filecache/-3801428777082625206/appTokens" "jobSubmitDir/appTokens"
mkdir -p jobSubmitDir
ln -sf
"/mnt/disk/yarn/local/usercache/dongxicheng/appcache/application_
1359695803957_0006/filecache/9160676915897562791/job.split" "jobSubmitDir/job.split"
mkdir -p jobSubmitDir
ln -sf
"/mnt/disk/yarn/local/usercache/dongxicheng/appcache/application_
1359695803957_0006/filecache/1006580749354161193/job.splitmetainfo"
"jobSubmitDir/job.splitmetainfo"
ln -sf
"/mnt/disk/yarn/local/usercache/dongxicheng/appcache/application_
1359695803957_0006/filecache/-471834009513672189/job.xml" "job.xml"
ln -sf "/mnt/disk/yarn/local/usercache/dongxicheng/appcache/application_
1359695803957_0006/filecache/-4649686916775863221/job.jar" "job.jar"
exec /bin/bash -c "$JAVA_HOME/bin/java
-Dlog4j.configuration=container-log4j.properties -Dyarn.app.mapreduce.container.log.
dir=/mnt/disk/yarn/log/application_1359695803957_0006/container_1359695803957_0006_01_000001
-Dyarn.app.mapreduce.container.log.filesize=0 -Dhadoop.root.logger=INFO,CLA -Xmx1024m
org.apache.hadoop.mapreduce.v2.app.MRAppMaster
1>/mnt/disk/yarn/log/application_1359695803957_0006/container_1359695803957_0006_01_00
0001/stdout
2>/mnt/disk/yarn/log/application_1359695803957_0006/container_1359695803957_0006_01_00
0001/stderr "

```

该脚本最重要的命令是最后一句：

```
exec /bin/bash -c "... org.apache.hadoop.mapreduce.v2.app.MRAppMaster ..."
```

这句话表明这是一个执行 MapReduce ApplicationMaster 的 Container，如果是执行普通任务的 Container，则这一句的内容如下：

```
exec /bin/bash -c "... org.apache.hadoop.mapred.YarnChild 100.88.88.88 47733
attempt_1359695803957_0006_r_000002_0 51 ..."
```

即最终在一个独立进程中执行了 Java 类 org.apache.hadoop.mapred.YarnChild，该类封装了 MapReduce Task 的执行逻辑。

步骤3 准备好 launch_container.sh 脚本后，ContainerLaunch 首先向 ContainerImpl 发送一个 CONTAINER_LAUNCHED 事件，以让它启动 Container 监控机制，然后调用 ContainerExecutor#launchContainer 启动 Container。目前 YARN 提供了 DefaultContainerExecutor 和 LinuxContainerExecutor 两种 ContainerExecutor 实现：

- DefaultContainerExecutor：它将 launch_container.sh 和 Tokens 文件复制到 Container 工作目录（\${yam.nodemanager.local-dirs}/usercache/\${user}/appcache/\$appid/\$containerid）中，并重新构造一个脚本 default_container_executor.sh，最终以 NodeManager 启动者的身份运行 launch_container.sh 脚本。一个 default_container_executor.sh 脚本的示例如下：

```
#!/bin/bash

echo $$ >
/mnt/disk/yarn/local/nmPrivate/container_1359695803957_0006_01_000001.pid.tmp
/bin/mv -f
/mnt/disk/yarn/local/nmPrivate/container_1359695803957_0006_01_000001.pid.tmp /
mnt/disk/yarn/local/nmPrivate/container_1359695803957_0006_01_000001.pid
exec setsid /bin/bash
"/mnt/disk/yarn/local/usercache/dongxicheng/appcache/application_1359695803957_0006/co
ntainer_1359695803957_0006_01_000001/launch_container.sh"
```

- LinuxContainerExecutor：与 DefaultContainerExecutor 的逻辑类似，但不同之处是它以应用程序所属用户身份运行该脚本，这是通过调用一个采用 C 语言实现的 setuid 可执行程序 container-executor 完成的。LinuxContainerExecutor 是 Hadoop 引入安全机制后加入的，后面将要提到的 CPU 资源隔离机制也是在该 ContainerExecutor 中实现的。

步骤4 ContainerImpl 收到 CONTAINER_LAUNCHED 事件后，进一步向 ContainersMonitorImpl 服务发送一个 START_MONITORING_CONTAINER 事件，以启动对该 Container 的内存资源使用量监控，至此 ContainerImpl 状态从 LOCALIZED 转换为 RUNNING；同时，由于 ContainerExecutor#launchContainer 函数是阻塞式的，因此，只有当脚本执行完成后才退出，这使得 ContainerLauncher 可在第一时间知道 Container 完成时间，之后向 ContainerImpl 发送一个 CONTAINER_EXITED_WITH_SUCCESS 事件，此时 ContainerImpl 状态由 RUNNING 转换为 EXITED_WITH_SUCCESS。至此，一个 Container 运行完成，接下来将进入该 Container 的资源清理阶段。

7.6.3 Container 资源清理

Container 资源清理是指 Container 运行完成之后（可能成功或者失败），NodeManager 需回收它占用的资源，这些资源主要是 Container 运行时使用的临时文件，它们的来源主要是 ResourceLocalizationService 和 ContainerExecutor 两个服务/组件，其中，ResourceLocalizationService 将数据 HDFS 文件下载到本地，ContainerExecutor 为 Container 创建私有工作目录，并保存一些临时文件（比如 Container 进程 pid 文件）。因此，Container 资源清理过程主要是通知这两个组件删除临时目录，过程如图 7-21 所示。

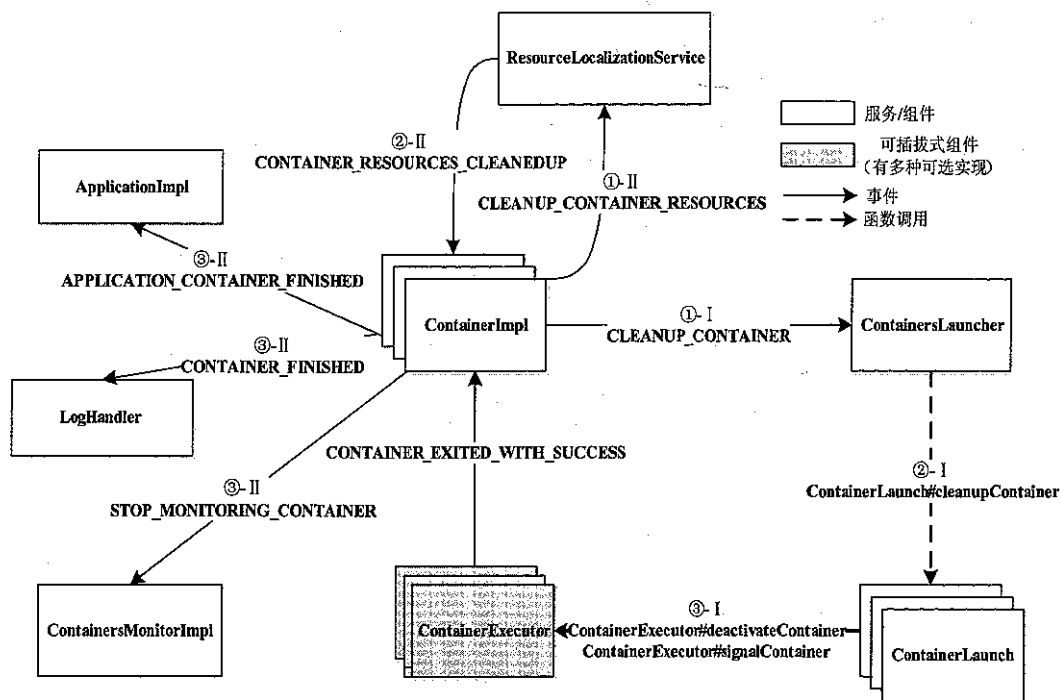


图 7-21 Container 资源清理过程（每个步骤均包含若干个并行操作）

Container 资源清理的主要步骤如下：

步骤 1 ContainerImpl 收到 CONTAINER_EXITED_WITH_SUCCESS 事件后，将依次向 ContainersLauncher 和 ResourceLocalizationService 发送 CLEANUP_CONTAINER 和 CLEANUP_CONTAINER_RESOURCES 事件。

步骤 2 ContainersLauncher 收到 CLEANUP_CONTAINER 事件后，将 Container 从正运行 Container 列表中移除，并调用函数 ContainerLaunch#cleanupContainer 清理 Container 占用的临时目录；ResourceLocalizationService 收到 CLEANUP_CONTAINER_RESOURCES 事件后，将依次清理用户工作目录和 NodeManager 私有目录下的 Container 目录，即 `${yarn.nodemanager.local-dirs}/usercache/${user}/appcache/${appid}/${containerid}` 和 `${yarn.nodemanager.local-dirs}/nmPrivate/${appid}/${containerid}` 目录，这两个目录均存放

了 Tokens 文件和 Shell 运行脚本（注意，用于存放临时数据的目录 `${yarn.nodemanager.local-dirs}/usercache/${user}/appcache/${appid}/output` 并不会删除）。之后，向 `ContainerImpl` 发送 `CONTAINER_RESOURCES_CLEANEDUP` 事件。

步骤3 `ContainerLaunch` 检查 `Container` 的进程 `pid` 文件是否存在，如果存在则说明 `Container` 进程尚未退出，`ContainerLaunch` 将调用 `ContainerExecutor#signalContainer` 依次向该进程发送 `SIGTERM` 和 `SIGKILL` 以强制回收资源防止资源泄露，之后删除进程 `pid` 文件；`ContainerImpl` 收到 `CONTAINER_RESOURCES_CLEANEDUP` 事件后，设置一些 Metric 信息，然后依次向 `ApplicationImpl`、`ContainerMonitorImpl` 和 `LogHandler` 发送 `APPLICATION_CONTAINER_FINISHED`、`STOP_MONITORING_CONTAINER` 和 `CONTAINER_FINISHED` 三个事件。这三个组件收到事件后，将该 `Container` 移除维护的 `Container` 列表，并移除对该 `Container` 的资源使用量监控和记录 `Container` 完成日志。

注意 由于应用程序可能涉及多种类型的计算任务，且这些计算任务之间有依赖关系，因此，`NodeManager` 不能在 `Container` 运行完成之后立刻清理它占用的所有资源，尤其是产生的中间数据，而只有当所有 `Container` 运行完成之后，才能够全部清空这些资源。由于每个 `NodeManager` 上只负责处理一个应用程序的部分任务（见图 7-22），因此它无法知道一个应用程序何时完成，该信息只有控制着全部信息的 `ResourceManager` 知道，因此当一个应用程序运行结束时，需要由它广播给各个 `NodeManager`，再进一步由 `NodeManager` 清理应用程序占用的所有资源，包括产生的中间数据。

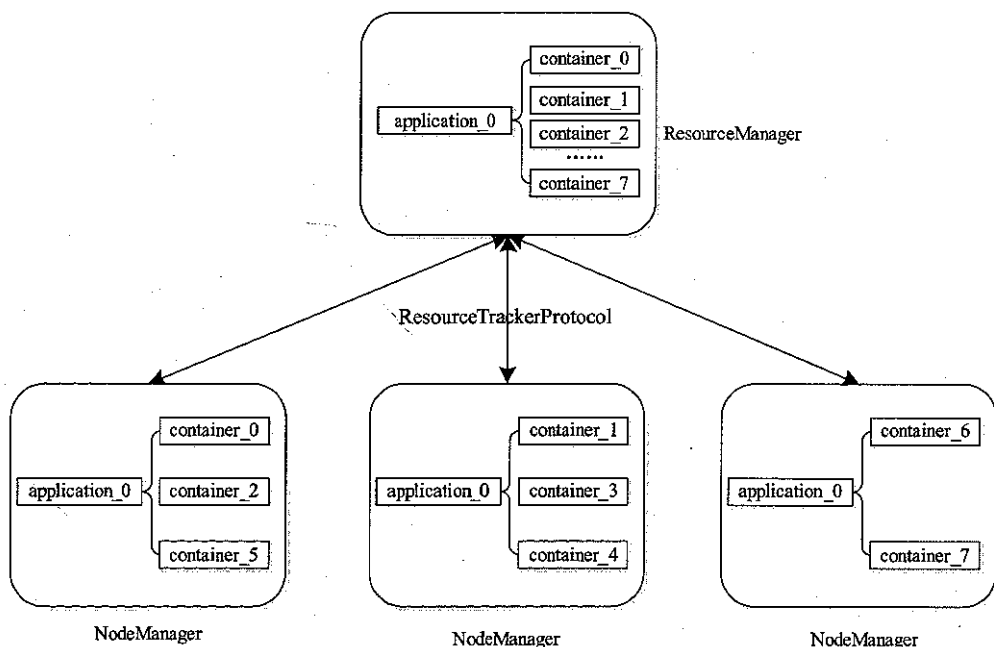


图 7-22 Container 分布

7.7 资源隔离

资源隔离是指为不同任务提供可独立使用的计算资源以避免它们相互干扰。当前存在很多资源隔离技术，比如硬件虚拟化、虚拟机、Cgroups^①、Linux Container^②等。

YARN 对内存资源和 CPU 资源采用了不同的资源隔离方案。对于内存资源，它是一种限制性资源，它的量的大小直接决定的应用程序的死活。为了能够更灵活地控制内存使用量，YARN 提供了两种可选方案：线程监控方案和基于轻量级资源隔离技术 Cgroups 的方案^③。默认情况下，YARN 采用了进程监控的方案控制内存使用，即每个 NodeManager 会启动一个额外监控线程监控每个 Container 内存资源使用量，一旦发现它超过约定的资源量，则将其杀死。采用这种机制的另一个原因是 Java 中创建子进程采用了“fork()+exec()”的方案，子进程启动瞬间，它使用的内存量与父进程一致。从外面看来，一个进程使用的内存量可能瞬间翻倍，然后又降下来，采用线程监控的方法可防止这种情况下导致 swap 操作。另一种可选的方案则基于轻量级资源隔离技术 Cgroups，Cgroups 是 Linux 内核提供的弹性资源隔离机制，可以严格限制内存使用上限，一旦进程使用资源量超过事先定义的上限值，则可将其杀死^④（截至本书结稿时，该方案尚未合并到 YARN 内核中）。对于 CPU 资源，它是一种弹性资源，它的量的大小不会直接影响应用程序的死活，因此采用了 Cgroups。具体可参考 YARN-2^⑤。

7.7.1 Cgroups 介绍

Cgroups(Control groups)^⑥是 Linux 内核提供了一种可以限制、记录、隔离进程组所使用的物理资源（如 CPU、内存、IO 等）的机制，最初由 Google 的工程师提出，后来被整合进 Linux 内核。

Cgroups 最初的目标是为资源管理提供一个统一的框架，既整合现有的 cpuset 等子系统，也为未来开发新的子系统提供接口，以使得 Cgroups 适用于多种应用场景，从单个进程的资源控制到实现操作系统层次的虚拟化的应用场景均支持。总结起来，Cgroups 提供了以下功能：

- 限制进程组使用的资源量。比如，内存子系统可以为进程组设定一个内存使用上限，一旦进程组使用的内存达到限额再申请内存，就会出发 OOM。
- 进程组的优先级控制。比如，可以使用 CPU 子系统为某个进程组分配特定 cpu share（下面将会介绍，Hadoop YARN 正是使用了该功能）。
- 对进程组使用的资源量进行记账。比如，可以记录某个进程组使用的 CPU 时间，以

① 参见网址 <http://en.wikipedia.org/wiki/Cgroups>。

② 参见网址 <http://lxc.sourceforge.net/>。

③ 参见网址 <https://issues.apache.org/jira/browse/YARN-3>。

④ 参考网址 <https://issues.apache.org/jira/browse/YARN-3> 中的补丁 MAPREDUCE-4334-v2.patch。

⑤ 参见网址 <https://issues.apache.org/jira/browse/YARN-2>。

⑥ 参见网址 <https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>。

便对不同进程组拥有者计费。

□ 进程组控制。比如，可将某个进程组挂起和恢复。

1. 相关术语

为了便于大家理解 Cgroups，下面给出一些术语：

□ 任务。在 Cgroups 中，一个任务实际上就是系统的一个进程。

□ 控制组。控制组就是一组按照某种标准划分的进程。Cgroups 中的资源控制是以控制组为单位实现。一个进程可以加入到某个控制组，也可以动态地从一个控制组迁移到另一个控制组。Cgroups 能够以控制组为单位为同一个进程组的进程分配资源，同时为该控制组加以资源限制与隔离。

□ 层级。控制组可以组织成层级的形式（树状结构，称为控制组树）。控制组树上的子节点控制组是父节点控制组的孩子，继承父控制组特定的属性。

□ 子系统。一个子系统就是一个资源控制器，比如 CPU 子系统就是控制 CPU 时间分配的一个控制器。子系统必须附加到一个层级上才能起作用，一个子系统附加到某个层级以后，这个层级上的所有控制族群都受到这个子系统的控制。当前 Cgroups 中包含的子系统主要有以下几个：

○ blkio：这个子系统为块设备设定输入/输出限制，比如物理设备（磁盘、固态硬盘、USB 等）。

○ CPU：这个子系统使用调度程序控制任务对 CPU 的访问。（YARN 使用了该子系统）

○ cpuacct：这个子系统自动生成任务所使用的 CPU 报告。

○ cpuset：这个子系统为任务分配独立 CPU（通常为多核系统）和内存节点。

○ devices：这个子系统可允许或者拒绝任务访问设备。

○ freezer：这个子系统挂起或者恢复 Cgroups 中的任务。

○ Memory：这个子系统设定任务使用的内存限制，并自动生成由那些任务使用的内存资源报告。

○ net_cls：这个子系统使用等级识别符标记网络数据包，可允许 Linux 流量控制程序识别 Cgroups 中任务生成的数据包。

○ ns：名称空间子系统。

YARN 使用了其中的 CPU 和 Memory 子系统。CPU 子系统用于控制 Cgroups 中所有进程可以使用的 CPU 时间片。用户可根据需要修改 Cgroups 的目录下的 `cpu.shares` 文件配置进程的 CPU 使用份额，而 CPU 子系统能够根据写入的整数值控制该进程获得的时间片。例如，在两个 Cgroups 中都将 `cpu.shares` 设定为 1 的任务将获得相同的 CPU 使用时间。然而，`cpu.shares` 设定为 2 的任务可使用的 CPU 时间是 `cpu.shares` 设定为 1 的任务可使用 CPU 时间的两倍。Memory 子系统可用于限定一个进程的内存使用上限，一旦超过该限制，将认为它 OOM，会将其杀死。

CPU 子系统是通过 Linux CFS (Completely Fair Scheduler)^①调度器实现的,在“完全理想的多任务处理器”下,CFS 能够使得每个进程都能同时获得 CPU 的执行时间。当系统中有两个进程时,CPU 的计算时间被分成两份,每个进程获得 50%。然而在实际的硬件上,当一个进程占用 CPU 时,其他进程就必须等待。所以 CFS 将惩罚当前进程,使其他进程能够在下次调度时尽可能取代当前进程,最终实现所有进程的公平调度。

2. Cgroups 安装与使用^②

安装 Cgroups 之前,请确定 Linux 内核版本为 2.6.24 或以上,较旧的 Linux 内核并没有引入 Cgroups 技术。

如果是 Redhat 系统,则使用下面的命令安装:

```
yum install libcgroup
```

如果是 Ubuntu 系统,则使用下面的命令安装:

```
sudo apt-get install cgroup-bin
```

Cgroups 服务的启动和停止命令如下:

```
service cgconfig start|stop
```

Cgroups 启动时,会读取配置文件 /etc/cgconfig.conf 的内容,根据其内容创建和挂载指定的 Cgroups 子系统。/etc/cgconfig.conf 是 Cgroups 配置工具 libcgroup 用来进行 Cgroups 组定义,参数设定以及挂载点定义的配置文件,主要由 mount 和 group 两个 section 构成。

mount section 的语法格式如下:

```
mount {
    <controller> = <path>;
    ...
}
```

其中,<controller> 是 Cgroups 子系统的名称;<path> 为该子系统的挂载点。举例如下:

```
mount {
    cpu = /cgroup/cpu_test;
}
```

上面的定义相当于如下 Shell 指令 (YARN 使用了 Shell 命令进行安装):

```
mkdir /cgroup/cpu
mount -t cgroup -o cpu cpu /cgroup/cpu
```

group section 的语法格式如下:

```
group <name> {
    [<permissions>]
    <controller> {
        <param name> = <param value>;
        ...
    }
}
```

^① <https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt>。

^② 部分内容参考自《红帽企业版 Linux 6 资源管理指南》。

```

}
...
}

```

其中, <name> 是指定 Cgroups 的名称; <permissions> 为可选项, 表示指定 Cgroups 对应的挂载点文件系统的权限和管理员权限; <controller> 是子系统的名称; <param name> 和 <param value> 表示子系统的属性及其属性值。<permissions> 的语法如下:

```

perm {
    task {
        uid = <task user>;
        gid = <task group>;
    }
    admin {
        uid = <admin name>;
        gid = <admin group>;
    }
}

```

对 group section 举例如下:

mount { # 定义需要创建的 cgroup 子系统及其挂载点, 这里创建 CPU 与 cpuacct (统计) 两个 cgroup 子系统

```

cpu = /cgroups/cpu;
cpuacct = /cgroups/cpu;
}

```

group hadoop-yarn { # 定义 hadoop-yarn 组

perm { # 定义这个组的权限

```

    task {
        uid = root;
        gid = webmaster;
    }
    admin {
        uid = root;
        gid = root;
    }
}

```

cpu { ## 定义 CPU 子系统的属性及其值, 即属于此组的任务的权重为 500

```

    cpu.shares = 500;
}

```

上面配置文件的定义相当于执行了如下 Shell 命令:

```

mkdir /cgroups/cpu
mount -t cgroup -o cpu,cpuacct cpu /cgroups/cpu
mkdir /cgroups/cpu/hadoop-yarn
chown root:root /cgroups/cpu/hadoop-yarn/*
chown root:webmaster /cgroups/cpu/hadoop-yarn/tasks
echo 500 > /cgroups/cpu/hadoop-yarn/cpu.shares

```

注意 修改 /etc/cgconfig.conf 之后, 需要重启 cgconfig 服务才能使配置生效。

如果 Cgroups 子系统安装成功，可在 `/proc/mounts` 文件中查看到类似下面的结果：

```
# grep '^cgroup' /proc/mounts
cgroup /cgroup/cpu cgroup rw,relatime,cpu 0 0
cgroup /cgroup/cpuacct cgroup rw,relatime,cpuacct 0 0
cgroup /cgroup/memory cgroup rw,relatime,memory 0 0
```

总之，Cgroups 存在两种安装方式：借助配置文件和使用 Shell 命令。在 YARN 中，管理员可事先安装好 Memory 和 CPU 两个子系统供 NodeManager 启动 Container 时直接使用，否则管理员需要进行一系列配置，以指导 NodeManager 自动进行安装，具体将在 7.7.2 和 7.7.3 节中介绍。

Cgroups 配置完成之后，接下来就是使用它。常见的 Cgroups 使用场景有如下两种。

(1) 在 Cgroups 中启动一个进程

如果想要在 Cgroups 中启动一个新的进程，可以使用 `cgexec` 命令，其语法为：

```
cgexec -g subsystems:path_to_cgroup [--sticky] command arguments
```

其中 `subsystems` 可以是一个子系统或者多个以逗号“,”隔开的子系统；`path_to_cgroup` 是 Cgroups 层级路径；`command` 为待启动进程的执行命令；`arguments` 为命令参数；此外，如果添加了“-sticky”参数，则该进程的所有子进程也被放在同一个 Cgroups 中，否则子进程按照普通的新进程处理。

例如：

```
cgexec -g cpu:hadoop-yarn ./container-executor XXX
```

(2) 将已有进程加入 Cgroups

如果想要将一个已有进程动态加入到 Cgroups 中，可以使用 `cgclassify` 命令，其用法为：

```
cgclassify -g subsystems:path_to_cgroup pidlist
```

其中 `pidlist` 为待启动进程的 ID，如果有多个进程 ID，则它们之间以空格隔开。例如：

```
cgclassify -g cpu,memory:hadoop-yarn 1674
```

除了使用 `cgclassify` 命令之外，也可以直接使用 `echo` 将进程加入 Cgroups。使用方法为

```
echo pid > /cgroup_path/tasks
```

对于前面的例子，使用 `echo` 填充的方法可写成如下形式：

```
echo 1674 > /cgroup/cpu/hadoop-yarn/tasks
echo 1674 > /cgroup/memory/hadoop-yarn/tasks
```

Hadoop YARN 使用了 `echo` 填充的方法。

7.7.2 内存资源隔离

在 MRv1 中，各个任务被运行在独立的 Java 虚拟机中以达到资源隔离的目的。然而，考虑到用户应用程序可能会创建其他子进程，如 Hadoop Pipes（或者 Hadoop Streaming）

编写的 MapReduce 应用程序中每个任务至少由 Java 进程和 C++ 进程两个进程组成，这难以通过创建单独的虚拟机达到资源隔离的效果。为了弥补这个不足，在 Linux 环境（其他操作系统暂不支持）中，MRv1 在各个 TaskTracker 上启动一个线程以监控各个任务内存使用情况，一旦发现某个任务使用内存过量，则直接将其杀死。而 YARN 则采纳了 MRv1 这种基于线程监控的资源控制方式，这样做的主要出发点与 MRv1 类似：这种方式更加灵活，且能够防止内存骤增骤降导致内存不足而死掉。

1. 相关配置参数

由于不同的应用程序对内存的需求不同，因此，Hadoop 提供了各种配置参数帮助用户和管理员合理地使用内存资源。这些参数可分为两类：一类是用户配置参数，比如用户根据自己应用程序特点设定的每个任务需要的最大内存量；另一类是管理员配置参数，比如每个节点的最大可用内存。

(1) 用户配置参数

用户可配置参数是由各个应用程序的 ApplicationMaster 提供的，不同应用程序需根据具体情况定义自己的参数，以 MapReduce ApplicationMaster 为例，其可定义的参数为 `mapred.job.{map|reduce}.memory.mb`，表示作业的 Map Task 或 Reduce Task 需使用的内存量（单位：MB）。

(2) 管理员配置参数

管理员配置参数主要包括：

- ❑ `yarn.nodemanager.pmem-check-enabled`：是否启动物理内存量监控，默认是 `true`。
- ❑ `yarn.nodemanager.vmem-check-enabled`：是否启动虚拟内存量监控，默认是 `true`。
- ❑ `yarn.nodemanager.vmem-pmem-ratio`：虚拟内存与物理内存使用比例，默认是 2.1，表示每使用 1MB 物理内存，最多可使用 2.1MB 虚拟内存。
- ❑ `yarn.nodemanager.resource.memory-mb`：该节点上最多可用物理内存（单位：MB），默认是 8×1024 ，即 8GB。

2. 基于线程监控的内存隔离方案

应用程序内存监控是由服务 `ContainersMonitorImpl` 实现的，它保存了每个 Container 进程的 pid，这样它内部的 `MonitoringThread` 线程每隔一段时间（由参数 `yarn.nodemanager.container-monitor.interval-ms` 指定，默认是 3000 ms）扫描所有正在运行的 Container 进程树，并按照以下步骤检查它们使用的内存量是否超过上限值。

步骤 1 构造进程树。

在 `/proc` 目录下，有大量以整数命名的目录，这些整数是某个正在运行的进程的进程号，而目录下面则是该进程运行时的一些信息。为了更全面地监控一个 Container 的内存使用量，NodeManager 通过读取 `/proc/<pid>/stat` 文件构造出以该 Container 进程为根的进程树，这样通过监控该进程树使用的内存总量可严格限制一个任务使用的内存量。

步骤 2 判断单个任务内存使用量是否超过最大值内存量。

NodeManager 使用的正则表示式为:

```
] + \\s) {7} ([0-9]+) \\s{([0-9]+)} \\s{([0-9-]+) \\s) {7} ([0-9]+) \\s{([0-9]+)} (\\s[0-9-]+) {15}
```

息，NodeManager 通过执行以下 Shell 命令获取每个 page 对应的内存量（单位：B）：

```
getconf PAGESIZE
```

KILL CONTAINER 事件将其杀死。

因此，如果你想使用 Cgroups 隔离方案，需要修改 YARN 内核代码。

YARN 采用了 Cgroups 对 CPU 资源进行隔离。

ContainerExecutor, 并按照以下步骤进行操作。

① 参见网址 <https://issues.apache.org/jira/browse/YARN-3>。

步骤1 编译生成可执行文件。编译生成 LinuxContainerContainer 可执行文件（采用 C 语言编写，实际上是一个 setuid 可执行程序）：

```
$ mvn package -Dcontainer-executor.conf.dir=/etc/hadoop/
```

其中，Dcontainer-executor.conf.dir 指向的路径应保存了该 setuid 可执行程序的配置文件（见步骤3），而可执行文件可安装在目录 \$HADOOP_YARN_HOME/bin 下。

步骤2 修改可执行文件权限。修改可执行文件权限为 6050（--Sr-s---），并修改它的拥有者为 root 用户、所在用户组为 hadoop（或者其他用户组均可）。注意，hadoop 用户组中只能有启动 NodeManager 的那个用户和 root 两个用户，不能有其他应用程序用户，否则会破坏安全机制。

步骤3 修改配置文件。修改 yarn-site.xml 和 container-executor.cfg 两个配置文件。

首先我们介绍 yarn-site.xml 文件的修改，具体如下。

□ 添加 LinuxContainerExecutor 相关的配置选项，具体如下：

```
<property>
  <description> 配置 ContainerExecutor 实现 </description>
  <name>yarn.nodemanager.container-executor.class</name>
  <value>org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor</value>
</property>
<property>
  <description> LCE 资源处理器 </description>
  <name>yarn.nodemanager.linux-container-executor.resources-handler.class</name>
  <value>org.apache.hadoop.yarn.server.nodemanager.util.CgroupsLCEResourcesHandler
    </value>
</property>
<property>
  <description>Cgroups 层次结构在何处存放 YARN 进程（不能包含空格），如果将 yarn.nodemanager.
linux-container-executor.cgroups.mount 设置为 false，Cgroups 层次必须已经存在且是由 NodeManager
用户写入的，否则 NodeManager 将启动失败。该参数只有在 LCE 资源处理器为 CgroupsLCEResources Handler 时
生效 </description>
  <name>yarn.nodemanager.linux-container-executor.cgroups.hierarchy</name>
  <value>/hadoop-yarn</value>
</property>
<property>
  <description> 当 Cgroups 不存在时，LCE 是否尝试安装 Cgroups。该参数只有在 LCE 资源处理器为
CgroupsLCEResourcesHandler 时生效 </description>
  <name>yarn.nodemanager.linux-container-executor.cgroups.mount</name>
  <value>true</value>
</property>
<property>
  <description> 如果 Cgroups 不存在时，LCE 尝试将其安装的路径。该参数只有在 LCE 资源处理器为
CgroupsLCEResourcesHandler，且 yarn.nodemanager.linux-container-executor.cgroups.mount
设置为 true 时生效 </description>
  <name>yarn.nodemanager.linux-container-executor.cgroups.mount-path</name>
  <value>/cgroup</value>
</property>
</property>
```

```

<description>LCE 可执行文件所属的用户组 </description>
<name>yarn.nodemanager.linux-container-executor.group</name>
<value>hadoop</value>
</property>

```

□ 配置虚拟 CPU 数目，具体如下：

```

<property>
  <name>yarn.nodemanager.resource.cpu-vcores</name>
  <value>32</value>
</property>

```

YARN 引入了“虚拟 CPU”这一术语，它是由物理 CPU 映射产生的，比如一个物理 CPU 代表 4 个虚拟 CPU，若一台机器可用 CPU 个数为 8，则该值可配成 32。YARN 不让管理员和用户配置可用物理 CPU 个数，而是直接配置虚拟 CPU 个数。虚拟 CPU 的引入，带来了很多好处，包括：允许用户更细粒度的设置 CPU 资源量，比如你想让自己的一个任务在最差情况下使用一个 CPU 的 50%，可在提交应用程序时设置 CPU 虚拟个数为 2（假设物理 CPU 和虚拟 CPU 映射关系为 1:4）；从一定程度上解决了 CPU 异构问题，可以根据物理 CPU 的性能高低为它们设置不同的虚拟 CPU 个数。

下面我们看看 container-executor.cfg 文件的修改，具体如下：

```

yarn.nodemanager.linux-container-executor.group=hadoop
min.user.id=1000

```

注意 Cgroups 只能够保证应用程序的 CPU 使用量下限，但不能限制使用上限。举例说明，一个节点上有 10 个可用虚拟 CPU，如果只运行了一个应用程序（设置的虚拟 CPU 需求数目为 4），则它最多能够使用全部 CPU；如果运行了两个应用程序（设置的虚拟 CPU 需求数目均为 4），则每个应用程序最多可使用全部 CPU，最少可使用一半的 CPU 资源；如果运行了三个应用程序（其中两个使用虚拟 CPU 个数为 4，一个是 2），则它们最多均可以使用全部 CPU，最少可使用 CPU 资源比例分别为： $4/(4+4+2) \times 10=40\%$ 、 $4/(4+4+2) \times 10=40\%$ 、 $2/(4+4+2) \times 10=20\%$ 。

2. CPU 资源隔离实现

默认情况下，NodeManager 未启用任何 CPU 资源隔离机制，如果想启用该机制，需使用 LinuxContainerExecutor，它能够以应用程序提交者的身份创建文件、运行 Container 和销毁 Container。相比于 DefaultContainerExecutor 采用 NodeManager 启动者的身份执行这些操作，LinuxContainerExecutor 的这种方式要安全得多。

LinuxContainerExecutor 的核心设计思想是，赋予 NodeManager 启动者以 root 权限，进而使它拥有足够的权限以任意用户身份执行一些操作，从而使得 NodeManager 执行者可以将 Container 使用的目录和文件的拥有者修改为应用程序提交者，并以应用程序提交者的身份运行 Container，防止所有 Container 以 NodeManager 执行者身份运行进而带来的各种安全风险。比如防止用户在 Container 中执行一些只有 NodeManager 用户有权限执行的命

令（杀死其他应用程序的命令、关闭或者杀死 NodeManager 进程等）。

为了实现上述机制，NodeManager 采用 C 语言实现了一个具有 setuid 功能的工具——container-executor，它拥有 root 权限，可以完成任意操作，比如创建 Cgroups 层级树、设置 Cgroups 属性等。LinuxContainerExecutor 通过调用这个可执行文件可以修改 Container 的一些属性以限制 Container 的非法操作（比如关闭 NodeManager、杀死 NodeManager 等）。

LinuxContainerExecutor 继承自基类 ContainerExecutor，主要实现了以下几种方法：

```
public abstract class ContainerExecutor implements Configurable {
    // 初始化，包括检查配置文件设置，目录权限设置等
    public abstract void init() throws IOException;
    // 为应用程序的 Container 准备运行环境，包括创建目录、准备资源等
    public abstract void startLocalizer(Path nmPrivateContainerTokens,
        InetAddress nmAddr, String user, String appId, String locId,
        List<String> localDirs, List<String> logDirs)
        throws IOException, InterruptedException;
    // 启动 Container，需要注意的是，该函数是阻塞式的，等到 Container 运行结束后才退出
    public abstract int launchContainer(Container container,
        Path nmPrivateContainerScriptPath, Path nmPrivateTokensPath,
        String user, String appId, Path containerWorkDir, List<String> localDirs,
        List<String> logDirs) throws IOException;
    // 向可执行程序 container-executor 发送信号，当前仅会发送 TERM 和 KILL 两种信号
    public abstract boolean signalContainer(String user, String pid,
        Signal signal)
        throws IOException;
    // 以用户 user 身份清理 Container 使用的临时目录，通常在 Container 执行完成后调用
    public abstract void deleteAsUser(String user, Path subDir, Path... basedirs)
        throws IOException, InterruptedException;
}
```

container-executor 工具包含了一系列与上面几个方法对应的 C 语言函数，这些函数将被以上几个方法以 Shell 命令的形式调用执行，以完成实际的处理工作。这些函数如下：

```
// 被 LinuxContainerExecutor#init 调用
int check_executor_permissions(char *executable_file);
// 被 LinuxContainerExecutor#startLocalizer 调用
int initialize_app(const char *user, const char *app_id,
    const char *credentials, char* const* local_dirs,
    char* const* log_dirs, char* const* args);
// 被 LinuxContainerExecutor#launchContainer 调用
int launch_container_as_user(const char * user, const char *app_id,
    const char *container_id, const char *work_dir,
    const char *script_name, const char *cred_file,
    const char *pid_file, char* const* local_dirs,
    char* const* log_dirs, const char *resources_key,
    char* const* resources_value);
// 被 LinuxContainerExecutor#signalContainer 调用
int signal_container_as_user(const char *user, int pid, int sig);
// 被 LinuxContainerExecutor#deleteAsUser 调用
int delete_as_user(const char *user,
```



```
const char *dir_to_be_deleted,
char* const* baseDirs);
```

LinuxContainerExecutor 允许用户事先设置好 Cgroups 层级，这样它启动 Container 后可直接通过将 CPU 份额和进程 ID 写入 cgroups 路径的方式实现 CPU 资源隔离。当然，用户可以不进行任何设置，但需要按照前面的介绍进行配置以让 LinuxContainerExecutor 自动完成这些工作。

7.8 源代码阅读引导

NodeManager 的所有实现在源代码目录下的 `hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-nodemanager/src/main/java` 目录中，代码结构如图 7-23 所示。

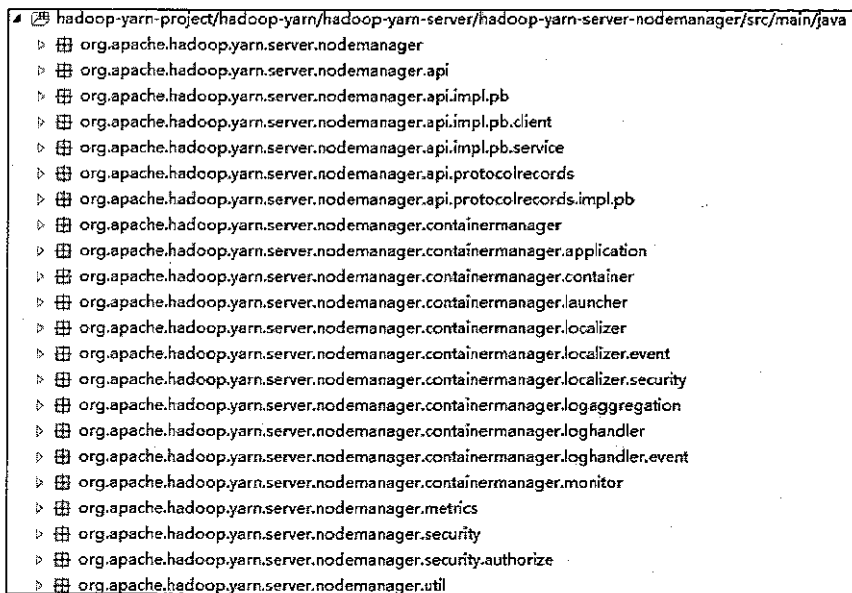


图 7-23 NodeManager 源代码组织结构

NodeManager 由多个内部服务组成，这些服务同时也可能是事件处理器，因此，通常可按照以下方法学习这些服务的实现：先单独了解每个服务的内部实现，如果是服务，则了解它的主要功能及实现；如果是事件处理器，则弄清楚要处理的事件类型和每种事件的处理逻辑，然后跳出单个服务，从全局看，不同服务之间是如何通过事件或者函数调用串联在一起的，为了便于理解和总结，可以边阅读代码边画出类似于本章前几节给出的事件转换图。下面依次介绍 NodeManager 服务涉及的 Java 实现包：

- `org.apache.hadoop.yarn.server.nodemanager`：NodeManager 主类以及部分内部重要服务的实现包，包括 NodeManager（组装所有服务，main 函数所在类）；ContainerExecutor，

是 Container 执行器, 目前有 DefaultContainerExecutor 和 Linux-ContainerExecutor 两种实现, 已在 7.6 节和 7.7 节进行了介绍; DeletionService, 是文件删除服务, 采用异步方式删除文件或目录, 已在 7.1.2 节进行了介绍; LocalDirsHandlerService, 是磁盘可用性检测服务, 已在 7.2.2 节进行了介绍; NodeHealthCheckerService, 通过执行用户自定义脚本判断节点健康情况, 已在 7.2.1 节进行了介绍; NodeStatusUpdaterImpl, 是 NodeManager 与 ResourceManager 交互的服务, 已在 7.1.2 节进行了介绍。

- org.apache.hadoop.yarn.server.nodemanager.api.*: 该包定义了 RPC 协议 Localization-Protocol 的相关 API, 该协议已在 7.6 节介绍。
- org.apache.hadoop.yarn.server.nodemanager.containermanager: 该包实现了 NodeManager 中最重要的服务组件 ContainerManager 及各个子组件的实现, 包括 AuxServices、ContainerLocalizationImpl 等。
- org.apache.hadoop.yarn.server.nodemanager.containermanager.application: 该包实现了状态机 ApplicationImpl, 已在 7.5.1 节进行了介绍。
- org.apache.hadoop.yarn.server.nodemanager.containermanager.container: 该包实现了状态机 ContainerImpl, 已在 7.5.2 节进行了介绍。
- org.apache.hadoop.yarn.server.nodemanager.containermanager.launcher: 该包实现了 Container 启动服务 ContainersLauncher, 已在 7.6 节进行了介绍。
- org.apache.hadoop.yarn.server.nodemanager.containermanager.localizer.*: 该包实现了 Container 资源本地化(下载)服务 ResourceLocalizationService 及其他辅助模块, 已在 7.6 节进行了介绍。
- org.apache.hadoop.yarn.server.nodemanager.containermanager.logaggregation: 该包实现了基于聚集转存的日志清理服务 LogAggregationService, 已在 7.4.2 节进行了介绍。
- org.apache.hadoop.yarn.server.nodemanager.containermanager.loghandler.*: 该包实现了基于定期删除的日志清理服务 NonAggregatingLogHandler, 已在 7.4.2 节进行了介绍。
- org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor: 该包实现了 Container 资源使用量监控服务 ContainersMonitorImpl, 已在 7.7.2 节进行了介绍。
- org.apache.hadoop.yarn.server.nodemanager.security: 该包是安全机制相关的实现。

7.9 小结

本章以功能点为线索逐步剖析了 NdeManager 内部模块和 workflows, 包括节点健康状况监测、分布式缓存机制、目录结构管理、状态机管理、Container 生命周期、资源隔离机制等各个方面。

需要注意的是, 作为资源管理系统 YARN 的一个重要服务, NodeManager 管理的是

Container，而不是任务，一个 Container 中可能运行着各种任务，但是对 NodeManager 而言是透明的，它只负责 Container 相关的操作，比如管理 Container 的生命周期，即启动 Container、监控 Container 和清理 Container 等，而这正是本章的核心所在。

7.10 问题讨论

问题 1：修改代码，使得 NodeManager 可通过 CGroups 对内存进行资源隔离，管理员可配置 CPU 是采用 Cgroups 隔离还是现有的基于线程监控的方案。

问题 2：在 NodeManager 中实现以下功能：根据应用程序优先级设置每个 Container 可使用的磁盘 IO 比例（可使用 Cgroups）。

问题 3：NodeManager 负责对单节点上的磁盘资源分配，那么它是如何将磁盘路径传递给 Container 的呢？能否为一个 Container 分配多个可用磁盘路径？

问题 4：在 NodeManager 中实现以下功能：将端口号作为一种资源，由 NodeManager 负责统一管理和分配，并增加端口号申请、分配、查看等函数接口，如果 Container 内部任务需要使用某个端口号，则统一向 NodeManager 申请。



第三部分

计算框架篇

作为一个通用资源管理系统，YARN 能让各类计算框架运行在一个集群中，并为它们提供统一的资源管理和调度。将一个计算框架运行在 YARN 上，可为它带来易于运维、计算资源弹性可伸缩和数据可共享等众多好处。本章将介绍几个常见的计算框架的编程模型和基本架构，包括离线计算框架 MapReduce、DAG 计算框架 Tez、实时计算框架 Storm 和内存计算框架 Spark 等，并重点分析如何将它们运行在 YARN 上。

第 8 章 离线计算框架 MapReduce

第 2 章已经介绍了 MRv1 存在的众多不足, 包括单点故障、扩展性差等, 而正是这些问题催生了资源统一管理平台 YARN。YARN 的诞生, 使得 MapReduce 运行时环境更加完善。

我们知道, MRv1 主要由编程模型 (MapReduce API)、资源管理与作业控制模块 (由 JobTracker 和 TaskTracker 组成) 和数据处理引擎 (由 MapTask 和 ReduceTask 组成) 三部分组成。而 YARN 出现后, 资源管理模块则交由 YARN 实现, 这样为了让 MapReduce 框架运行在 YARN 上, 仅需实现一个 ApplicationMaster 组件完成作业控制模块功能即可, 其他部分, 包括编程模型和数据处理引擎等, 可直接采用 MRv1 原有的实现。本章将详细介绍 MapReduce On YARN (即 MRv2) 的基本架构、模块组成以及各模块的实现。

8.1 概述

《Hadoop 技术内幕: 深入解析 MapReduce 架构设计与实现原理》一书中已经详细介绍了 MRv1 的设计架构和实现原理, 本节不再赘述, 而是重点介绍 MapReduce On YARN。MapReduce On YARN 与 MRv1 在编程模型和数据处理引擎方面的实现是一样的, 唯一的不同是运行时环境。不同于 MRv1 中由 JobTracker 和 TaskTracker 构成的运行时环境, MapReduce On YARN 的运行时环境由 YARN 与 ApplicationMaster 构成, 这种新颖的运行时环境使得 MapReduce 可以与其他计算框架运行在一个集群中, 从而达到共享集群资源、提高资源利用率的目的。随着 YARN 的成熟与完善, MRv1 的独立运行模式将被 MapReduce On YARN 取代。

MRAppMaster 是 MapReduce 的 ApplicationMaster 实现, 它使得 MapReduce 应用程序可以直接运行于 YARN 之上。在 YARN 中, MRAppMaster 负责管理 MapReduce 作业的生命周期, 包括作业管理、资源申请与再分配、Container 启动与释放、作业恢复等。本节将介绍 MRAppMaster 基本构成。

8.1.1 基本构成

如图 8-1 所示, MRAppMaster 主要由以下几种组件 / 服务构成。

□ ContainerAllocator。与 ResourceManager 通信, 为 MapReduce 作业申请资源。作业的每个任务资源需求可描述为 5 元组 <Priority, hostname, capability, containers, relax_locality>, 分别表示作业优先级、期望资源所在的 host、资源量 (当前支持内存和 CPU 两种资源)、Container 数目、是否松弛本地性。ContainerAllocator 周期性

通过 RPC 与 ResourceManager 通信, 而 ResourceManager 则通过心跳应答的方式为之返回已经分配的 Container 列表、完成的 Container 列表等信息。

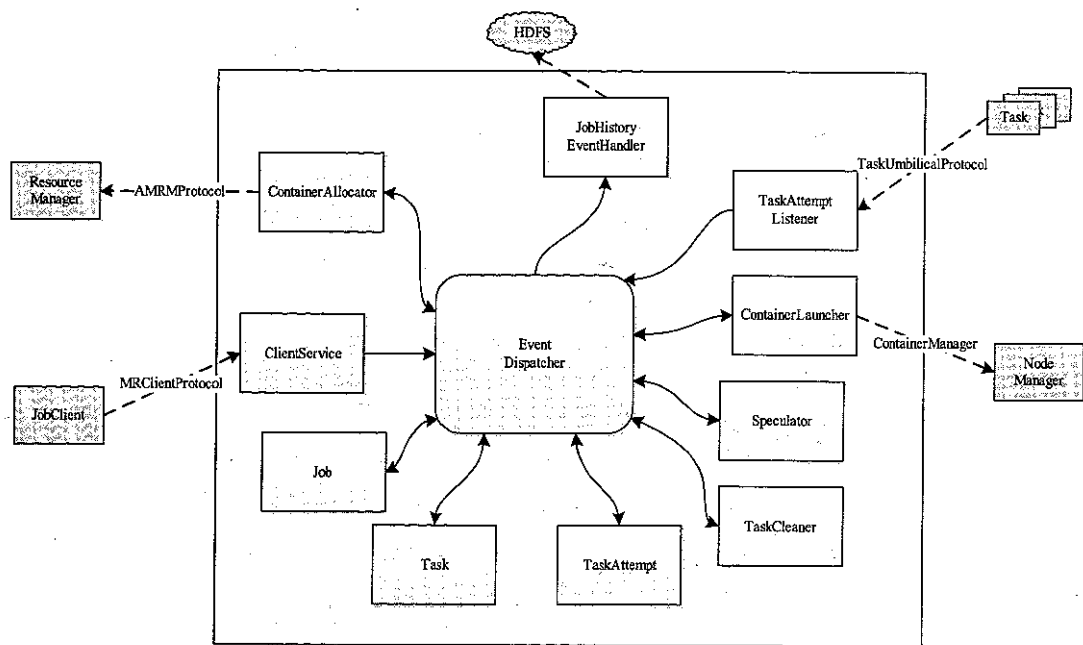


图 8-1 MRAppMaster 架构图

- ❑ **ClientService**。ClientService 是一个接口, 由 MRClientService 实现。MRClientService 实现了 MRClientProtocol 协议, 客户端可通过该协议获取作业的执行状态 (不必通过 ResourceManager) 和控制作业 (比如杀死作业、改变作业优先级等)。
- ❑ **Job**。Job 表示一个 MapReduce 作业, 与 MRv1 的 JobInProgress 功能一样, 负责监控作业的运行状态。它维护了一个作业状态机, 以实现异步执行各种作业相关的操作。
- ❑ **Task**。Task 表示一个 MapReduce 作业中的某个任务, 与 MRv1 中的 TaskInProgress 功能类似, 负责监控一个任务的运行状态。它维护了一个任务状态机, 以实现异步执行各种任务相关的操作。
- ❑ **TaskAttempt**。TaskAttempt 表示一个任务运行实例, 它的执行逻辑与 MRv1 中的 MapTask 和 ReduceTask 运行实例完全一致, 实际上, 它直接使用了 MRv1 的数据处理引擎, 但经过了一些优化, 我们将在 8.9 节详细介绍。正是由于它与 MRv1 的数据处理引擎一样, 它对外提供的编程接口也与 MRv1 完全一致, 这意味着 MRv1 的应用程序可直接运行在 YARN 之上。
- ❑ **TaskCleaner**。TaskCleaner 负责清理失败任务或者被杀死任务使用的目录和产生的临时结果 (可统称为垃圾数据), 它维护了一个线程池和一个共享队列, 异步删除任

务产生的垃圾数据。

- ❑ **Speculator**。Speculator 完成推测执行功能。当同一个作业的某个任务运行速度明显慢于其他任务时，Speculator 会为该任务启动一个备份任务，让它与原任务同时处理同一份数据，谁先计算完成则将谁的结果作为最终结果，并将另一个任务杀掉。该机制可有效防止那些“拖后腿”任务拖慢整个作业的执行进度。
- ❑ **ContainerLauncher**。ContainerLauncher 负责与 NodeManager 通信，以启动一个 Container。当 ResourceManager 为作业分配资源后，ContainerLauncher 会将任务执行相关信息填充到 Container 中，包括任务运行所需资源、任务运行命令、任务运行环境、任务依赖的外部文件等，然后与对应的 NodeManager 通信，要求它启动 Container。
- ❑ **TaskAttemptListener**。TaskAttemptListener 负责管理各个任务的心跳信息，如果一个任务一段时间内未汇报心跳，则认为它死掉了，会将其从系统中移除。同 MRv1 中的 TaskTracker 类似，它实现了 TaskUmbilicalProtocol 协议，任务会通过该协议汇报心跳，并询问是否能够提交最终结果。
- ❑ **JobHistoryEventHandler**。JobHistoryEventHandler 负责对作业的各个事件记录日志，比如作业创建、作业开始运行、一个任务开始运行等，这些日志会被写到 HDFS 的某个目录下，这对于作业恢复非常有用。当 MRAppMaster 出现故障时，YARN 会将其重新调度到另外一个节点上。为了避免重新计算，MRAppMaster 首先会从 HDFS 上读取上次运行产生的日志，以恢复已经运行完成的任务，进而能够只运行尚未运行完成的任务。

8.1.2 事件与事件处理器

前面提到，YARN 使用了基于事件驱动的异步编程模型，它通过事件将各个组件联系起来，并由一个中央异步调度器统一将各种事件分配给对应的事件处理器。在 MRAppMaster 中，每种组件是一个事件处理器，当 MRAppMaster 启动时，它们会以服务的形式注册到 MRAppMaster 的中央异步调度器上，并告诉调度器它们处理的事件类型。这样当出现某一种事件时，MRAppMaster 会查询<事件，事件处理器>表，并将该事件分配给对应的事件处理器。表 8-1 给出了 MRAppMaster 相关的事件与事件处理器。

表 8-1 MRAppMaster 相关事件与事件处理器

事件类型	事件处理器
ContainerAllocator.EventType	ContainerAllocator
JobEventType	Job（实现类为 JobImpl）
TaskEvent	Task（实现类为 TaskImpl）
TaskAttemptEvent	TaskAttempt（实现类为 TaskAttemptImpl）
CommitterEventType	CommitterEventHandler

(续)

事件类型	事件处理器
Speculator.EventType	Speculator
ContainerLauncher.EventType	ContainerLauncher
org.apache.hadoop.mapreduce.jobhistory.EventType	JobHistoryEventHandler

8.2 MapReduce 客户端

MapReduce 客户端是 MapReduce 用户与 YARN (和 MRAppMaster) 进行通信的唯一途径, 通过该客户端, 用户可以向 YARN 提交作业, 获取作业的运行状态和控制作业 (比如杀死作业、杀死任务等)。MapReduce 客户端涉及两个 RPC 通信协议:

- **ApplicationClientProtocol**。在 YARN 中, ResourceManager 实现了 ApplicationClientProtocol 协议, 任何客户端需使用该协议完成提交作业、杀死作业、改变作业优先级等操作。
- **MRClientProtocol**。当作业的 ApplicationMaster 成功启动后, 它会启动 MRClientService 服务, 该服务实现了 MRClientProtocol 协议, 从而允许客户端直接通过该协议与 ApplicationMaster 通信以控制作业和查询作业运行状态, 以减轻 ResourceManager 负载。

为了与 MRv1 的客户端兼容, MRAppMaster 混合使用了继承和组合的设计模式。通过使用继承设计模式, MRAppMaster 可保证它的对外接口与 MRv1 的一致, 通过使用组合设计模式, MapReduce 客户端使用 RPC 通信协议 ApplicationClientProtocol 和 MRClientProtocol 分别与 ResourceManager 和 ApplicationMaster 通信, 以完成作业提交和后续的作业运行状态查询和作业控制。

下面对 MapReduce 客户端的实现方法做更细节的介绍。如图 8-2 所示, 当用户通过 JobClient 提交作业时, 客户端会通过 Java 标准库中的 ServiceLoader 动态加载所有的 ClientProtocolProvider 实现, 默认情况下, YARN 提供了两种实现: LocalClientProtocolProvider 和 YamClientProtocolProvider。如果用户在配置文件中将选项 mapreduce.framework.name 置为 “yam”, 则客户端会采用 YamClientProtocolProvider, 该类会创建一个 YARNRunner 对象作为真正的客户端。YARNRunner 实现了 MRv1 中的 ClientProtocol 接口, 并将 Application-ClientProtocol 协议的 RPC 客户端句柄作为它的内部成员。这样, 用户的作业实际是通过 YARNRunner 类的 submitJob 函数提交的, 在该函数内部, 它会进一步调用 ApplicationClientProtocol 的 submitApplication 函数, 最终将作业提交到 ResourceManager 上。当 ApplicationMaster 成功启动后, 客户端可以通过 ClientServiceDelegate 直接与 ApplicationMaster 交互 (而不必与 ResourceManager 交互) 以查询作业运行状态和控制作业 (比如杀死任务)。

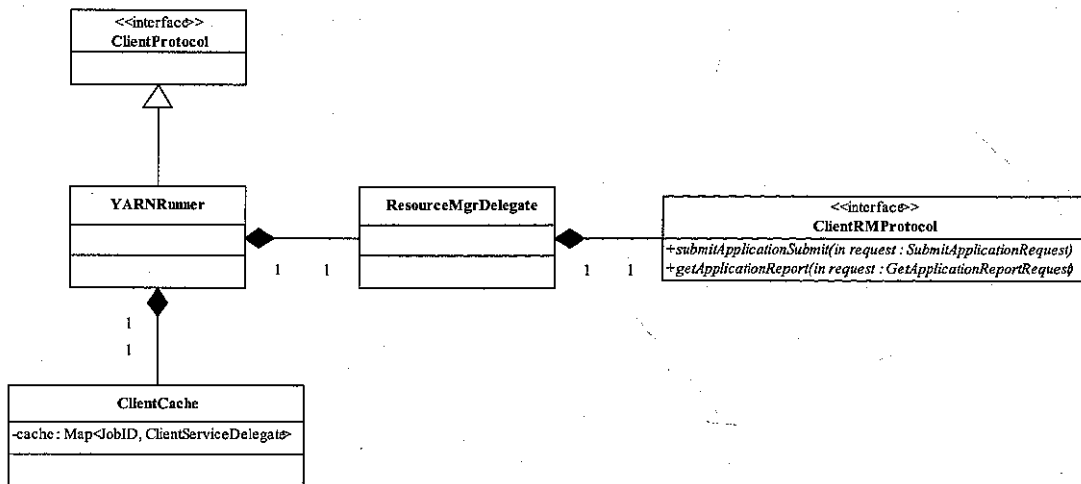


图 8-2 MapReduce 客户端实现

前面几章已经提到，在 YARN 中，应用程序（作业）的运行过程包括两个步骤：启动 Application-Master 和运行应用程序（作业）内部的各类任务，其中，ApplicationMaster 是由 ResourceManager 直接与 NodeManager 通信而启动的，在它启动起来之前，客户端只能与 ResourceManager 交互以查询作业相关信息。一旦作业的 ApplicationMaster 成功启动，客户端可直接与它交互以查询作业信息和控制作业。接下来，我们分别介绍这两个通信协议。

8.2.1 ApplicationClientProtocol 协议

MapReduce 客户端通过该协议与 ResourceManager 通信，以提交应用程序和查询集群信息。ResourceManager 用 Application 表示用户提交的作业，并提供了以下接口供用户使用：

```

public interface ApplicationClientProtocol {
    public GetNewApplicationResponse getNewApplication( // 获取一个 Application ID
        GetNewApplicationRequest request) throws YarnRemoteException;
    public SubmitApplicationResponse submitApplication( // 提交一个 Application
        SubmitApplicationRequest request) throws YarnRemoteException;
    public KillApplicationResponse forceKillApplication( // 杀死一个 Application
        KillApplicationRequest request) throws YarnRemoteException;
    public GetApplicationReportResponse getApplicationReport( // 获取 Application 运行报告
        GetApplicationReportRequest request) throws YarnRemoteException;
    public GetClusterMetricsResponse getClusterMetrics( // 获取集群所有 Metric
        GetClusterMetricsRequest request) throws YarnRemoteException;
    public GetAllApplicationsResponse getAllApplications( // 列出所有 Application
        GetAllApplicationsRequest request) throws YarnRemoteException;
    public GetClusterNodesResponse getClusterNodes( // 获取集群中所有节点
        GetClusterNodesRequest request) throws YarnRemoteException;
}

```

```

public GetQueueUserAclsInfoResponse getQueueUserAcls( // 获取用户访问控制权限
    GetQueueUserAclsInfoRequest request) throws YarnRemoteException;
...
}

```

8.2.2 MRClientProtocol 协议

MRAppMaster 实现了 MRClientProtocol 协议为客户端提供服务, 该协议提供了作业和任务的查询和控制接口, 主要如下:

```

public interface MRClientProtocol {
    ...
    public GetJobReportResponse getJobReport(GetJobReportRequest request) throws
        YarnRemoteException; // 获取作业运行报告
    public GetTaskReportResponse getTaskReport(GetTaskReportRequest request)
        throws YarnRemoteException; // 获取所有任务运行报告
    public GetTaskAttemptReportResponse getTaskAttemptReport(GetTaskAttemptReportR
        equest request) throws YarnRemoteException; // 获取所有任务实例的运行报告
    public GetCountersResponse getCounters(GetCountersRequest request) throws
        YarnRemoteException; // 获取所有 Counter
    public GetTaskAttemptCompletionEventsResponse getTaskAttemptCompletionEvents(G
        etTaskAttemptCompletionEventsRequest request) throws YarnRemoteException; //
        获取所有运行完成的任务
    public GetTaskReportsResponse getTaskReports(GetTaskReportsRequest request)
        throws YarnRemoteException; // 获取所有任务运行报告
    public GetDiagnosticsResponse getDiagnostics(GetDiagnosticsRequest request)
        throws YarnRemoteException; // 获取作业诊断信息
    public KillJobResponse killJob(KillJobRequest request) throws
        YarnRemoteException; // 杀死一个作业
    public KillTaskResponse killTask(KillTaskRequest request) throws
        YarnRemoteException; // 杀死一个任务
    public KillTaskAttemptResponse killTaskAttempt(KillTaskAttemptRequest request)
        throws YarnRemoteException; // 杀死一个任务实例
    public FailTaskAttemptResponse failTaskAttempt(FailTaskAttemptRequest request)
        throws YarnRemoteException; // 让一个任务实例运行失败
    ...
}

```

由于客户端的函数中的所有参数对象仍采用了 MRv1 中基于 Writable 的序列化和反序列化机制, 为了与 YARN 中基于 Protocol Buffers 的机制兼容, YARN 提供了一系列方便它们两者相互转换的接口, 有兴趣的读者可自行了解。

8.3 MRAppMaster 工作流程

按照作业大小不同, MRAppMaster 提供了三种作业运行模式: 本地模式 (通常用于作业调试, 同 MRv1 一样, 不再赘述)、Uber 模式^①和 Non-Uber 模式。对于小作业, 为

^① 参见网址 <https://issues.apache.org/jira/browse/MAPREDUCE-2405>。

了降低其延迟,可采用 Uber 模式,在该模式下,所有 Map Task 和 Reduce Task 在同一个 Container (MRAppMaster 所在 Container) 中顺次执行;对于大作业,则采用 Non-Uber 模式,在该模式下,MRAppMaster 先为 Map Task 申请资源,当 Map Task 运行完成数目达到一定比例后再为 Reduce Task 申请资源。

(1) Uber 运行模式

为了降低小作业延时,YARN 专门对小作业运行方式进行了优化。对于小作业而言,MRAppMaster 无须再为每个任务分别申请资源,而是让其重用同一个 Container,并按照先 Map Task 后 Reduce Task 的运行方式串行执行每个任务。在 YARN 中,如果一个 MapReduce 作业同时满足以下条件,则认为是小作业,可运行在 Uber 模式下:

- ☐ Map Task 数目不超过 `mapreduce.job.ubertask.maxmaps` (默认是 9)。
- ☐ Reduce Task 数目不超过 `mapreduce.job.ubertask.maxmaps` (默认是 1)。
- ☐ 输入文件大小不超过 `mapreduce.job.ubertask.maxbytes` (默认是一个 Block 大小)。
- ☐ Map Task 和 Reduce Task 需要的资源量不超过 MRAppMaster 可使用的资源量。

另外,由于链式作业会并发执行不同资源需求的 Map Task 和 Reduce Task,因此不允许运行在 Uber 模式下。

(2) Non-Uber 运行模式

在大数据环境下,Uber 运行模式通常只能覆盖到一小部分作业,而对于其他大多数作业,仍将运行在 Non-Uber 模式下。在 Non-Uber 模式下,MRAppMaster 将一个作业的 Map Task 和 Reduce Task 分为四种状态:

- ☐ `pending`: 刚启动但尚未向 ResourceManager 发送资源请求。
- ☐ `scheduled`: 已经向 ResourceManager 发送资源请求但尚未分配到资源。
- ☐ `assigned`: 已经分配到了资源且正在运行。
- ☐ `completed`: 已经运行完成。

对于 Map Task 而言,它的生命周期为 `scheduled` → `assigned` → `completed`;而对于 Reduce Task 而言,它的生命周期为 `pending` → `scheduled` → `assigned` → `completed`。由于 Reduce Task 的执行依赖于 Map Task 的输出结果,因此,为避免 Reduce Task 过早启动造成资源利用率低下,MRAppMaster 让刚启动的 Reduce Task 处于 `pending` 状态,以便能够根据 Map Task 运行情况决定是否对其进行调度。在 YARN 之上运行 MapReduce 作业需要解决两个关键问题:如何确定 Reduce Task 启动时机以及如何完成 Shuffle 功能。

由于 YARN 中不再有 Map Slot 和 Reduce Slot 的概念,且 RedouceManager 也不知道 Map Task 与 Reduce Task 之间存在依赖关系,因此,MRAppMaster 自己需设计资源申请策略以防止因 Reduce Task 过早启动造成资源利用率低下和 Map Task 因分配不到资源而“饿死”。MRAppMaster 在 MRv1 原有策略 (Map Task 完成数目达到一定比例后才允许启动 Reduce Task) 基础上添加了更为严格的资源控制策略和抢占策略。总结起来,Reduce Task 启动时机由以下三个参数控制:

- ☐ `mapreduce.job.reduce.slowstart.completedmaps`: 当 Map Task 完成的比例达到该值后

才会为 Reduce Task 申请资源，默认是 0.05。

□ `yarn.app.mapreduce.am.job.reduce.rampup.limit`：在 Map Task 完成前，最多启动的 Reduce Task 比例，默认为 0.5。

□ `yarn.app.mapreduce.am.job.reduce.preemption.limit`：当 Map Task 需要资源但暂时无法获取资源（比如 Reduce Task 运行过程中，部分 Map Task 因结果丢失需重算）时，为了保证至少一个 Map Task 可以得到资源，最多可以抢占的 Reduce Task 比例，默认为 0.5。

按照 MapReduce 的执行逻辑，Shuffle HTTP Server 应该分布到各个节点上，以便能够支持各个 Reduce Task 远程复制数据。然而，由于 Shuffle 是 MapReduce 框架中特有的一个处理流程，从设计上讲，不应该将它直接嵌到 YARN 的某个组件（比如 NodeManager）中。

前面提到，YARN 采用了服务模型管理各个对象，且多个服务可以通过组合的方式交由一个核心服务进行统一管理。在 YARN 中，NodeManager 作为一种组合服务模式，允许动态加载应用程序临时需要的附属服务，利用这一特性，YARN 将 Shuffle HTTP Server 组装成了一种服务，以便让各个 NodeManager 启动时加载它。

当用户向 YARN 中提交一个 MapReduce 应用程序后，YARN 将分两个阶段运行该应用程序：第一个阶段是由 ResourceManager 启动 MRAppMaster；第二个阶段是由 MRAppMaster 创建应用程序，为它申请资源，并监控它的整个运行过程，直到运行成功。如图 8-3 所示，YARN 的工作流程分为以下几个步骤：

步骤 1 用户向 YARN 中提交应用程序，其中包括 MRAppMaster 程序、启动 MRAppMaster 的命令、用户程序等。

步骤 2 ResourceManager 为该应用程序分配第一个 Container，并与对应的 NodeManager 通信，要求它在这个 Container 中启动应用程序的 MRAppMaster。

步骤 3 MRAppMaster 启动后，首先向 ResourceManager 注册，这样用户可以直接通过 ResourceManager 查看应用程序的运行状态，之后，它将为内部任务申请资源，并监控它们的运行状态，直到运行结束，即重复步骤 4~7。

步骤 4 MRAppMaster 采用轮询的方式通过 RPC 协议向 ResourceManager 申请和领取资源。

步骤 5 一旦 MRAppMaster 申请到资源后，则与对应的 NodeManager 通信，要求它启动任务。

步骤 6 NodeManager 为任务设置好运行环境（包括环境变量、JAR 包、二进制程序等）后，将任务启动命令写到一个脚本中，并通过运行该脚本启动任务。

步骤 7 各个任务通过 RPC 协议向 MRAppMaster 汇报自己的状态和进度，以让 MRAppMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务。

注意 在应用程序运行过程中，用户可随时通过 RPC 向 MRAppMaster 查询应用程序的当前运行状态。

步骤 8 应用程序运行完成后，MRAppMaster 向 ResourceManager 注销并关闭自己。

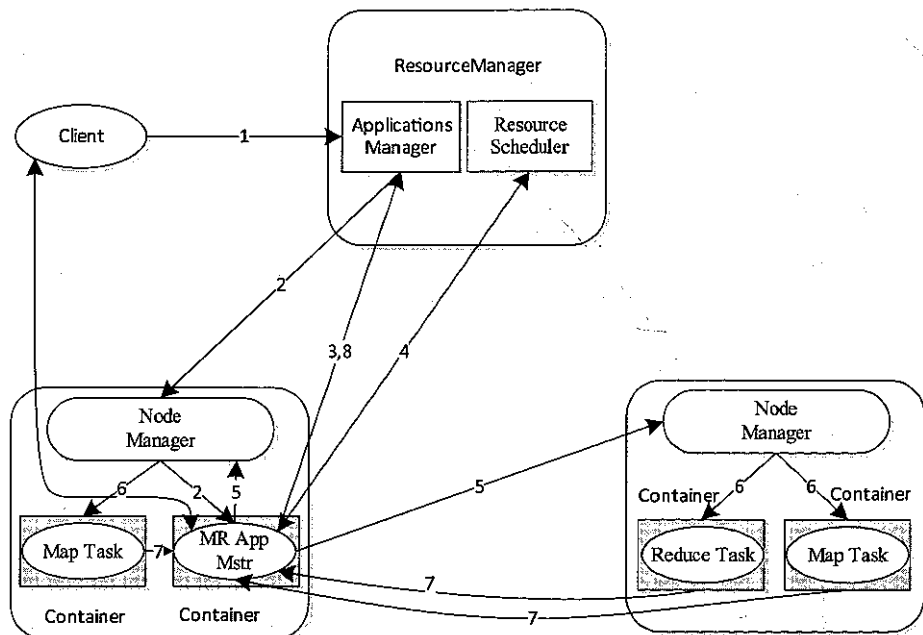


图 8-3 Apache YARN 的工作流程

8.4 MR 作业生命周期及相关状态机

本节将分析一个 MapReduce 作业从开始运行到运行结束所经历的整个过程，期间涉及的各种事件和状态变化。

8.4.1 MR 作业生命周期

在正式介绍作业 (Job) 生命周期之前，先要了解 MRAppMaster 中作业表示方式。如图 8-4 所示，与 MRv1 一样，MRAppMaster 根据 InputFormat 组件的具体实现（通常是根数据量切分数据），将作业分解成若干个 Map Task 和 Reduce Task，其中每个 Map Task 处理一片 InputSplit 数据，而每个 Reduce Task 则进一步处理 Map Task 产生的中间结果。每个 Map/Reduce Task 只是一个具体计算任务的描述，真正的任务计算工作则是由运行实例 TaskAttempt 完成的，每个 Map/Reduce Task 可能顺次启动多个运行实例，比如第一个运行实例失败了，则另起一个实例重新计算，直到这一份数据处理完成或者达到尝试次数上限；也可能同时启动多个运行实例，让它们竞争同时处理一片数据。在 MRAppMaster 中，上述 Job、Task 和 TaskAttempt 的生命周期均由一个有限状态机描述，其中 TaskAttempt 是实际进行任务计算的组件，其他两个只负责监控和管理。正是由于 MRAppMaster 中的

TaskAttempt 重用了 MRv1 中的代码, MRv1 中的 MapReduce 应用程序可直接运行在 YARN 上。

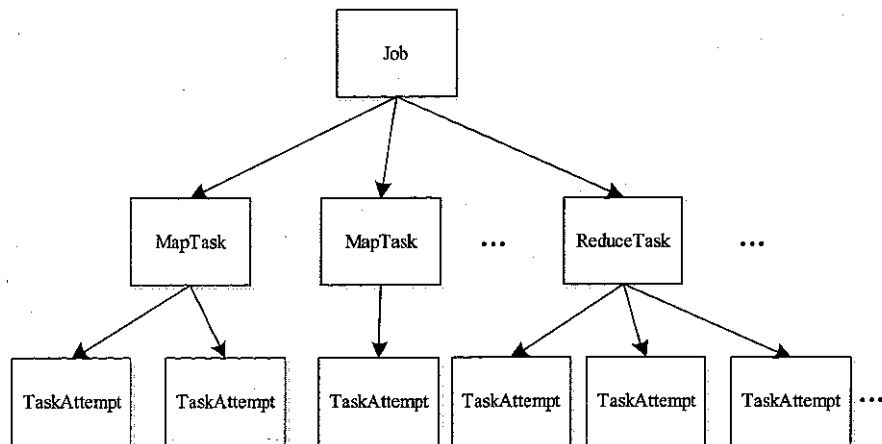


图 8-4 作业表示方式

作业的创建入口在 MRAppMaster 类中, 如下所示:

```
public class MRAppMaster extends CompositeService {
    public void start() {
        ...
        job = createJob(getConfig()); // 创建 Job
        JobEvent initJobEvent = new JobEvent(job.getID(), JobEventType.JOB_INIT);
        jobEventDispatcher.handle(initJobEvent); // 发送 JOB_INIT, 创建 MapTask、ReduceTask
        startJobs(); // 启动作业, 这是后续一切动作的触发之源
        ...
    }
    protected Job createJob(Configuration conf) {
        Job newJob =
            new JobImpl(jobId, appAttemptID, conf, dispatcher.getEventHandler(),
                taskAttemptListener, jobTokenSecretManager, fsTokens, clock,
                completedTasksFromPreviousRun, metrics, committer, newApiCommitter,
                currentUser.getUserName(), appSubmitTime, amInfos, context);
        ((RunningAppContext) context).jobs.put(newJob.getID(), newJob);
        dispatcher.register(JobFinishEvent.Type.class,
            createJobFinishEventHandler());
        return newJob;
    }
}
```

之后, MapReduce 作业将依次经历作业 / 任务初始化和作业启动两个阶段。

(1) 作业 / 任务初始化

JobImpl 首先接收到 JOB_INIT 事件, 然后触发调用函数 InitTransition(), 进而导致作业状态从 NEW 变为 INITED, InitTransition 函数主要工作是创建 Map Task 和 Reduce Task, 代码如下:

```

public static class InitTransition
    implements MultipleArcTransition<JobImpl, JobEvent, JobState> {
    ...
    createMapTasks(job, inputLength, taskSplitMetaInfo);
    createReduceTasks(job);
    ...
}

```

其中，createMapTasks 函数实现如下：

```

private void createMapTasks(JobImpl job, long inputLength,
    TaskSplitMetaInfo[] splits) {
    for (int i=0; i < job.numMapTasks; ++i) {
        TaskImpl task =
            new MapTaskImpl(job.jobId, i,
                job.eventHandler,
                job.remoteJobConfFile,
                job.conf, splits[i],
                job.taskAttemptListener,
                job.committer, job.jobToken, job.fsTokens,
                job.clock, job.completedTasksFromPreviousRun,
                job.applicationAttemptId.getAttemptId(),
                job.metrics, job.appContext);
        job.addTask(task);
    }
}

```

(2) 作业启动

启动作业的代码如下：

```

public class MRAppMaster extends CompositeService {
    protected void startJobs() {
        JobEvent startJobEvent = new JobEvent(job.getID(), JobEventType.JOB_START);
        dispatcher.getEventHandler().handle(startJobEvent);
    }
}

```

JobImpl 接收到 JOB_START 事件后，将执行函数 StartTransition()，进而触发 Map Task 和 Reduce Task 的调度执行，同时使得作业状态从 INITED 变为 RUNNING，具体如下：

```

public static class StartTransition
    implements SingleArcTransition<JobImpl, JobEvent> {
    public void transition(JobImpl job, JobEvent event) {
        job.scheduleTasks(job.mapTasks);
        job.scheduleTasks(job.reduceTasks);
    }
}

```

之后，每个 Map Task 和 Reduce Task 负责各自的状态变化，ContainerAllocator 模块会首先为 Map Task 申请资源，然后是 Reduce Task，一旦一个 Task 获取到了资源，就会创建

一个运行实例 Task Attempt。如果该实例运行成功,则 Task 运行成功,否则,Task 还会启动下一个运行实例 Task Attempt,直到一个 Task Attempt 运行成功或者达到尝试次数上限。当所有 Task 运行成功后,Job 运行成功。一个运行成功的作业/任务所经历的状态变化(不包含失败或者被杀死情况)如图 8-5 所示。

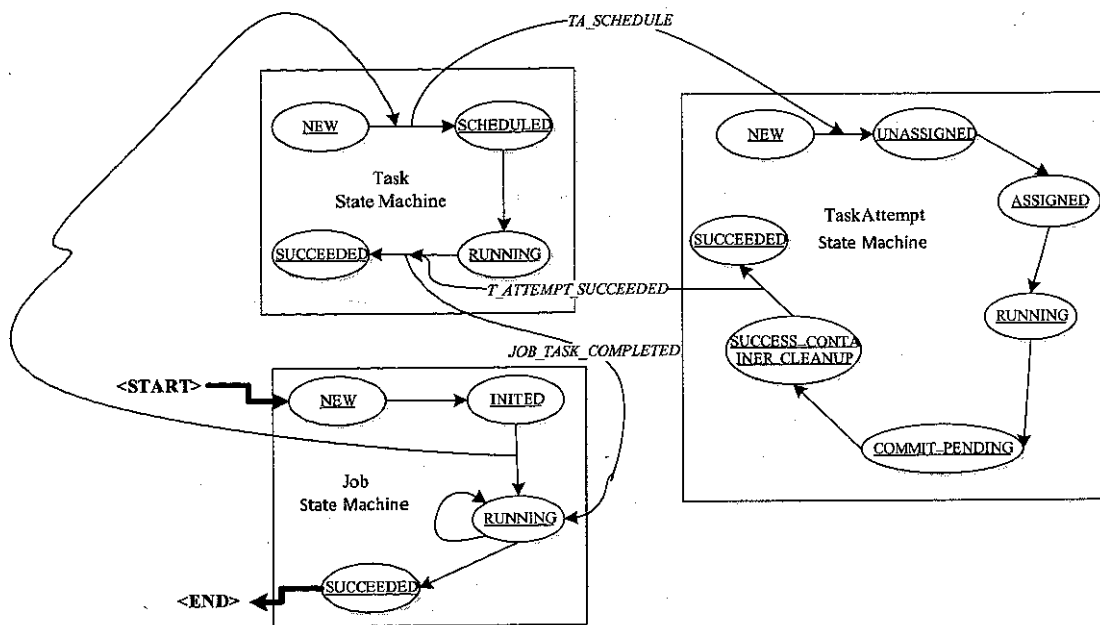


图 8-5 作业/任务状态转移图

为了更深入的介绍作业的生命周期,我们将详细介绍 Job、Task 和 TaskAttempt 的状态机。

8.4.2 Job 状态机

Job 状态机维护了一个 MapReduce 应用程序的生命周期,即从提交到运行结束整个过程。一个 Job 由多个 Map Task 和 Reduce Task 构成,而 Job 状态机则负责管理这些任务。Job 状态机由类 JobImpl 实现,主要包括 14 种状态和 19 种导致状态发生变化的事件,如图 8-6 所示。本节将详细剖析 Job 状态机的实现。

(1) Job 状态

Job 状态包括:

- NEW: 作业初始状态。当一个作业被创建时,初始状态被置为 NEW。
- INITED: 作业经初始化后的状态。该状态意味着,作业中的 Map Task 和 Reduce Task 被创建(但尚未开始运行)。
- SETUP: 作业启动状态。该状态意味着,作业的运行时间被设置,等待任务开始被

调度。

- **RUNNING**：作业运行状态。当作业位于该状态时，MRAppMaster 将为作业中的 Map Task 和 Reduce Task 申请资源，并将申请到的资源二次分配给各个任务，并与 NodeManager 通信以启动任务，直到全部任务成功运行完成。
- **COMMITTING**：作业等待提交最终结果时所处的状态。作业运行过程中产生的结果将被存放到一个临时目录中，仅当所有任务运行成功后，才会移动到最终目录下（即提交最终结果）。
- **SUCCEEDED**：作业运行成功。当作业的最终结果被成功移动到最终目录下后，作业运行成功。

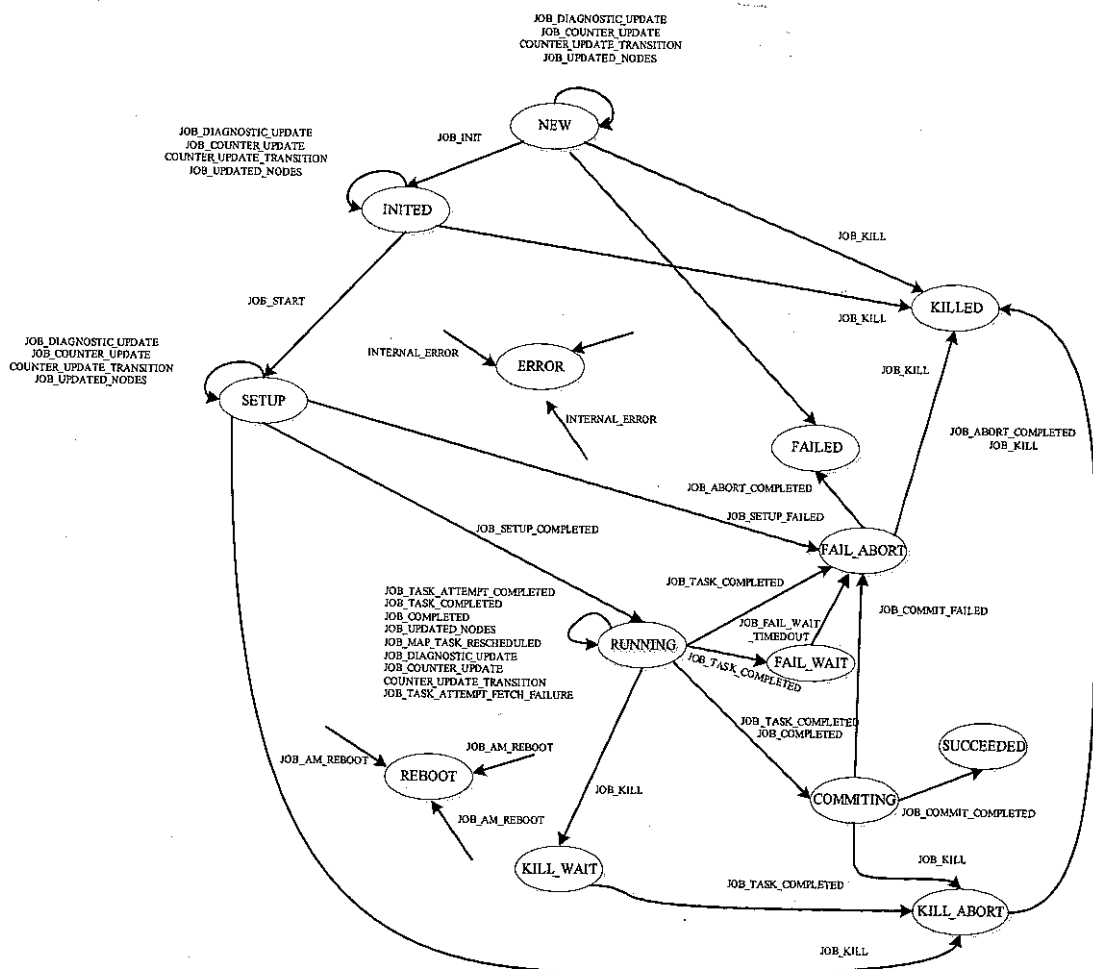


图 8-6 Job 状态机

上面 6 种状态是任何一个成功运行完成的 MapReduce 作业必须依次经历的状态，而下

面 8 种状态则是未运行成功的作业可能所处的状态。

- ❑ **ERROR**: 作业运行过程中出错后所处的状态。作业位于任何状态下时, 均有可能因为执行异常出错导致作业运行失败。
- ❑ **FAIL_WAIT**: 等待正在运行的任务被杀死时所处状态。当失败的 Map Task 或者 Reduce Task 数目超过了一定的比例, 将导致作业运行失败, 此时作业将设定一个时间片以期望在该时间段内杀死所有正在运行的任务。
- ❑ **FAIL_ABORT**: 作业运行失败将被注销时所处的状态。当作业处于 SETUP、RUNNING 或者 COMMITING 三种状态之一时, 可能进入 FAIL_ABORT 状态, 具体原因如下:
 - **SETUP → FAIL_ABORT**: 作业启动时抛出异常 (比如磁盘损坏导致文件创建失败等)。
 - **RUNNING → FAIL_ABORT**: 失败的 Map Task 或者 Reduce Task 数目超过了一定的比例, 导致作业运行失败, 其中最多失败的 Map Task 比例由参数 `mapreduce.map.failures.maxpercent` 指定, 最多失败的 Reduce Task 比例由参数 `mapreduce.reduce.failures.maxpercent`, 默认参数值均为 0。
 - **COMMITING → FAIL_ABORT**: 作业提交最终结果失败。
- ❑ **FAILED**: 作业运行失败所处的状态。当作业处于 SETUP 和 FAIL_ABORT 两种状态之一时, 可能进入 FAILED 状态, 具体原因如下:
 - **SETUP → FAILED**: 作业初始化失败, 比如上传文件失败等。
 - **FAIL_ABORT → FAILED**: 作业被成功注销 (成功执行函数 `OutputCommitter#abortJob`) 后, 将转为 FAILED 状态。
- ❑ **KILL_WAIT**: 作业等待被杀死时所处的状态。通常而言, 一个处于 RUNNING 状态的作业收到来自客户端的杀死作业请求后, 将被置为 KILL_WAIT 状态, 此后, 作业首先杀死所有正在运行的任务, 然后自行退出。
- ❑ **KILL_ABORT**: 作业被注销时所处的状态。位于该状态的作业, 它内部的所有任务要么成功运行完成, 要么被杀死。当作业处于 SETUP、COMMITING 或者 KILL_WAIT 三种状态之一时, 可能进入 KILL_ABORT 状态, 具体原因如下:
 - **SETUP/COMMITING → KILL_ABORT**: 处于 SETUP 或者 COMMITING 状态的作业收到来自客户端的杀死作业请求。
 - **KILL_WAIT → KILL_ABORT**: 一个处于 RUNNING 状态的作业收到客户端的杀死作业请求后, 将被置为 KILL_WAIT 状态, 此后作业将尝试杀死所有正在运行的任务, 待任务被杀死后, 该作业进入 KILL_ABORT 状态。
- ❑ **KILLED**: 作业被杀死后所处的状态。
- ❑ **REBOOT**: 作业重启所处状态。当 ResourceManager 重启时, 本来维护的作业信息可能丢失, 而如果此时 MRAppMaster 与它通信, ResourceManager 会通知它重新启动。

(2) Job 状态转移事件

Job 状态转移事件包括:

- **JOB_DIAGNOSTIC_UPDATE**: 记录作业诊断信息。作业执行过程中收到杀死作业请求或者出现错误时, 会记录作业的诊断信息, 以表明作业是如何退出的。需要注意的是, 该事件仅用于记录作业的诊断信息, 并不会改变作业当前的状态。
- **JOB_COUNTER_UPDATE**: 作业计数器更新。作业执行过程中, 系统计数器和用户自定义计数器均会定期更新, 每当有计数器更新时, 会触发一个 **JOB_COUNTER_UPDATE** 事件。同 **JOB_DIAGNOSTIC_UPDATE** 事件一样, 该事件不会改变作业的当前状态。
- **JOB_INIT**: 作业初始化事件。当 MRAppMaster 初始化时, 会发送一个 **JOB_INIT** 事件通知 JobImpl 初始化 MapReduce 作业。
- **JOB_START**: 启动作业事件。当作业初始化完成后, MRAppMaster 将向 JobImpl 发送 **JOB_START** 事件以启动 MapReduce 作业。
- **JOB_KILL**: 杀死作业时触发的事件。一般而言, 两种情况将触发该事件。用户请求杀死作业和框架主动杀死作业, 对于后者, 通常是由于作业请求的 Container 资源量超过了系统最大可分配的资源量。
- **INTERNAL_ERROR**: 作业执行过程中遇到错误时触发的事件。作业执行过程中可能遇到各种错误, 比如不可识别的事件类型、抛出 `IOException` 等。需要注意的是, 该事件将直接导致作业运行失败。
- **JOB_UPDATED_NODES**: 节点状态更新。作业执行过程中, 可从 ResourceManager 端获取节点状态变化列表, 比如哪些节点不可用、哪些节点恢复正常等, 而 MRAppMaster 可根据节点状态变化调整任务运行状态, 每次获取新的节点列表时将触发一个 **JOB_UPDATED_NODES** 事件。
- **JOB_SETUP_COMPLETED**: 作业启动成功时触发的事件。作业启动的主要任务是调用 `OutputCommitter#setupJob` 函数进行一些初始化工作, 只要该函数执行成功, 作业就会启动成功。
- **JOB_SETUP_FAILED**: 作业启动失败时触发的事件。与 **JOB_SETUP_COMPLETED** 事件相反, 如果作业启动时执行 `OutputCommitter#setupJob` 函数失败 (抛出 `Exception`), 则作业启动失败。
- **JOB_TASK_ATTEMPT_COMPLETED**: 作业的一个任务实例运行完成时触发的事件。当作业的任何任务实例运行完成时, 无论失败还是成功, 均会触发一个 **JOB_TASK_ATTEMPT_COMPLETED** 事件。
- **JOB_TASK_COMPLETED**: 作业的一个任务运行成功时触发的事件。当作业的任何任务成功运行完成时, 会触发一个 **JOB_TASK_COMPLETED** 事件, 该事件可能导致作业运行完成。
- **JOB_COMPLETED**: 作业运行完成时触发的事件。如果一个作业没有任何任务,

则将直接触发一个 JOB_COMPLETED 事件, 进而让作业运行结束。

- JOB_MAP_TASK_RESCHEDULED: 任务重新调度时触发的事件。当一个 Map Task 实例运行失败或者被杀死时, 将触发一个 JOB_MAP_TASK_RESCHEDULED 事件, 以重新调度该任务。
- JOB_TASK_ATTEMPT_FETCH_FAILURE: Reduce Task 远程复制数据失败时触发的事件。如果一个 Reduce Task 远程复制某个 Map Task 中间结果失败, 则会触发该事件。
- JOB_COMMIT_COMPLETED: 作业成功提交最终结果时触发的事件。作业的所有任务成功运行完成后, 将调用 OutputCommitter#commitJob 函数提交最终结果, 即将结果从临时目录移动到最终目录下。
- JOB_COMMIT_FAILED: 作业提交最终结果失败时触发的事件。与 JOB_COMMIT_COMPLETED 事件相反, 如果作业提交最终结果时执行 OutputCommitter#commitJob 函数失败 (抛出 Exception), 则作业提交失败。
- JOB_ABORT_COMPLETED: 作业注销成功时触发的事件。作业注销时, 会执行 OutputCommitter#abortJob 函数, 只要该函数执行完成, 则会触发 JOB_ABORT_COMPLETED 事件。
- JOB_FAIL_WAIT_TIMEDOUT: 当失败的 Map Task 或者 Reduce Task 数目超过了一定的比例时, 将导致作业运行失败, 此时作业将设定一个时间片以期望在该时间段内杀死所有正在运行的任务, 一旦设定的时间片到达后, 将会触发一个 JOB_FAIL_WAIT_TIMEDOUT 事件, 进而强制退出作业。

8.4.3 Task 状态机

Task 状态机维护了一个任务的生命周期, 即从创建到运行结束整个过程。一个任务可能存在多次运行尝试, 每次运行尝试被称为一个“运行实例”, Task 状态机则负责管理这些运行实例。Task 状态机由 TaskImpl 类实现, 其主要包括 7 种状态和 9 种导致状态发生变化的事件, 如图 8-7 所示。本节将详细剖析 Task 状态机的实现。

(1) Task 状态

Task 状态包括:

- NEW: 任务初始状态。当一个任务被创建时, 状态被置为 NEW。
- SCHEDULED: 任务开始被调度时所处的状态。该状态意味着, MRAppMaster 开始为任务 (向 ResourceManager) 申请资源, 但任务尚未获取到资源。
- RUNNING: 任务的一个实例开始运行。该状态意味着, MRAppMaster 已经为该任务申请到资源 (Container), 并与对应的 NodeManager 通信成功启动了 Container。需要注意的, 在某一时刻, 一个任务可能有多个运行实例, 且可能存在运行失败的运行实例。
- SUCCEEDED: 任务的一个实例运行成功。该状态意味着, 该任务成功运行并完成。

需要注意的，只要任务的一个实例运行成功，则意味着该任务运行成功。

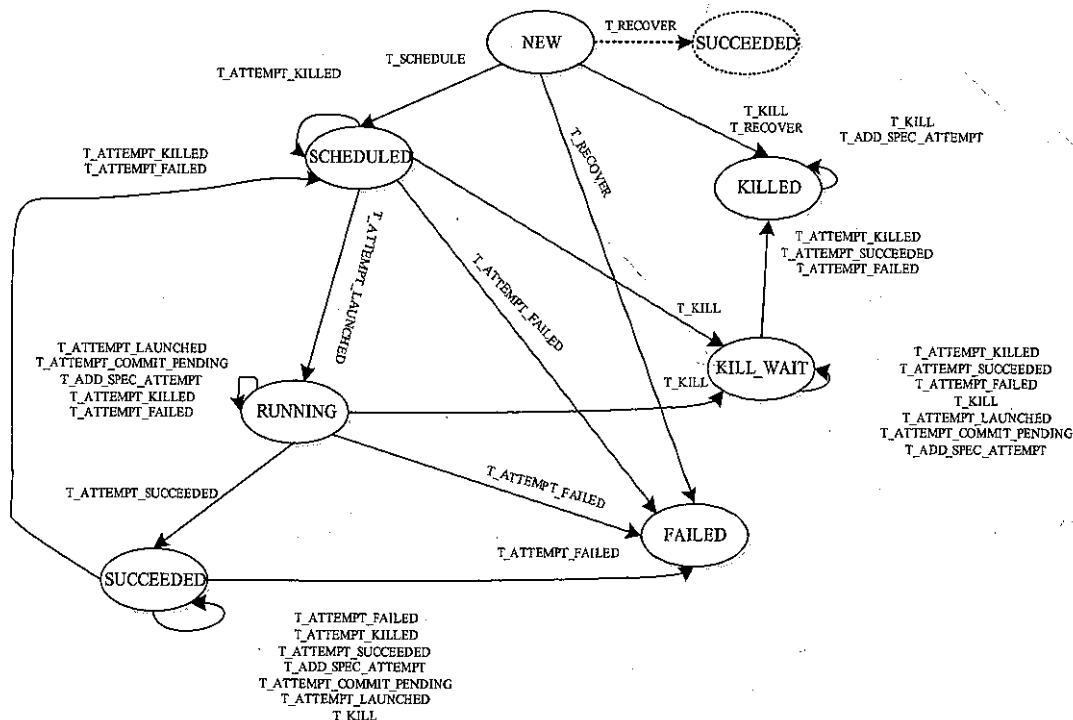


图 8-7 Task 状态机包括的状态和事件

- **KILL_WAIT**：任务等待被杀死时所处的状态。通常而言，一个处于 **RUNNING** 状态的任务收到来自客户端的杀死任务请求后，将被置为 **KILL_WAIT** 状态。此后，任务首先杀死正在运行的任务实例，然后退出。
- **KILLED**：任务被杀死所处的状态。当一个任务的所有运行实例被杀死后，才认为该任务被杀死。
- **FAILED**：任务运行失败所处的状态。每个任务的运行实例数目有一定上限，一旦超过该上限，才认为该任务运行失败，其中，Map Task 运行实例数目上限由参数 `mapreduce.map.maxattempts` 指定，默认值为 4，Reduce Task 运行实例数目上限由参数 `mapreduce.reduce.maxattempts` 指定，默认值为 4。需要注意的是，一个任务运行失败并不一定会导致整个作业运行失败，这同样取决于作业的错误容忍率（默认是 0）。

(2) Task 事件

Task 事件包括：

- **T_SCHEDULE**：任务开始调度时触发的事件。作业启动完成后（作业收到 `JOB_SETUP_COMPLETED` 事件），开始调度 Map Task 和 Reduce Task，这同时意味着作

业进入 RUNNING 状态。

- **T_RECOVER**：任务恢复时触发的事件。当 ResourceManager 重启时，将从日志中恢复之前作业的运行状态，仅当一个任务处于 RUNNING、KILLED、FAILED 或者 SUCCEEDED 状态之一时，才会进行恢复，其他状态的任务需重新运行。
- **T_ATTEMPT_LAUNCHED**：任务的一个实例成功启动时触发的事件。当任务获取到资源并成功启动后，会触发一个 T_ATTEMPT_LAUNCHED。
- **T_ATTEMPT_COMMIT_PENDING**：任务的一个实例运行完成后提交最终结果时触发的事件。在 MapReduce 中，一个任务可能有多个实例在运行，它们分别将结果写到一个独立的文件中，最先完成的实例才能够成功提交结果。
- **T_ATTEMPT_SUCCEEDED**：任务的一个实例运行成功时触发的事件。
- **T_ATTEMPT_KILLED**：任务的一个运行实例被杀死时触发的事件。需要注意的是，运行实例可以被无限次杀死（但是运行失败次数是有上限的），运行实例被杀死不会直接导致任务被杀死。
- **T_ATTEMPT_FAILED**：任务的一个运行实例运行失败时触发的事件。需要注意的是，一个运行实例运行失败并不一定会导致任务运行失败，这取决于任务的最大运行尝试次数。只要运行失败的实例数目未达到上限，则任务将一直处于 RUNNING 状态。
- **T_ADD_SPEC_ATTEMPT**：为一个正在运行的实例启动一个备份任务。当一个实例运行速度慢于其他实例时，MRAppMaster 会为该任务启动一个备份任务，让这两个实例同时处理一份数据，谁先处理完成，则采用谁的计算结果。
- **T_KILL**：杀死任务时触发的事件。一般而言，两种情况触发该事件，分别是用户请求杀死任务和框架主动杀死任务，对于后者，通常是由于作业被杀死或者备份任务被杀死造成的。

注意 处于 SUCCEEDED 状态的 Map Task 收到 T_ATTEMPT_FAILED 或者 T_ATTEMPT_KILLED 事件后，仍可能转换为 SCHEDULED 状态重新开始运行，这通常是由于 Map Task 的中间计算结果损坏或者 Map Task 所在节点宕掉导致的，这种情况下需要重新计算相关 Map Task 产生的结果。

8.4.4 TaskAttempt 状态机

TaskAttempt 状态机维护了一个任务运行实例的生命周期，即从创建到运行结束整个过程。它由 TaskAttemptImpl 类实现，主要包括 13 种状态和 17 种导致状态发生变化的事件，如图 8-8 所示。本节将详细剖析 TaskAttempt 状态机的实现。

在 YARN 中，任务实例是运行在 Container 中的，因此，Container 状态变化往往伴随任务实例的状态变化，比如任务实例运行完成后，会清理 Container 占用的空间，而 Container 空间的清理实际上就是任务实例空间的清理。目前每个任务存在两种类型的实例：

Avataar.VIRGIN 和 Avataar.SPECULATIVE, 分别表示原始任务和备份任务（通过推测执行机制启动的）。

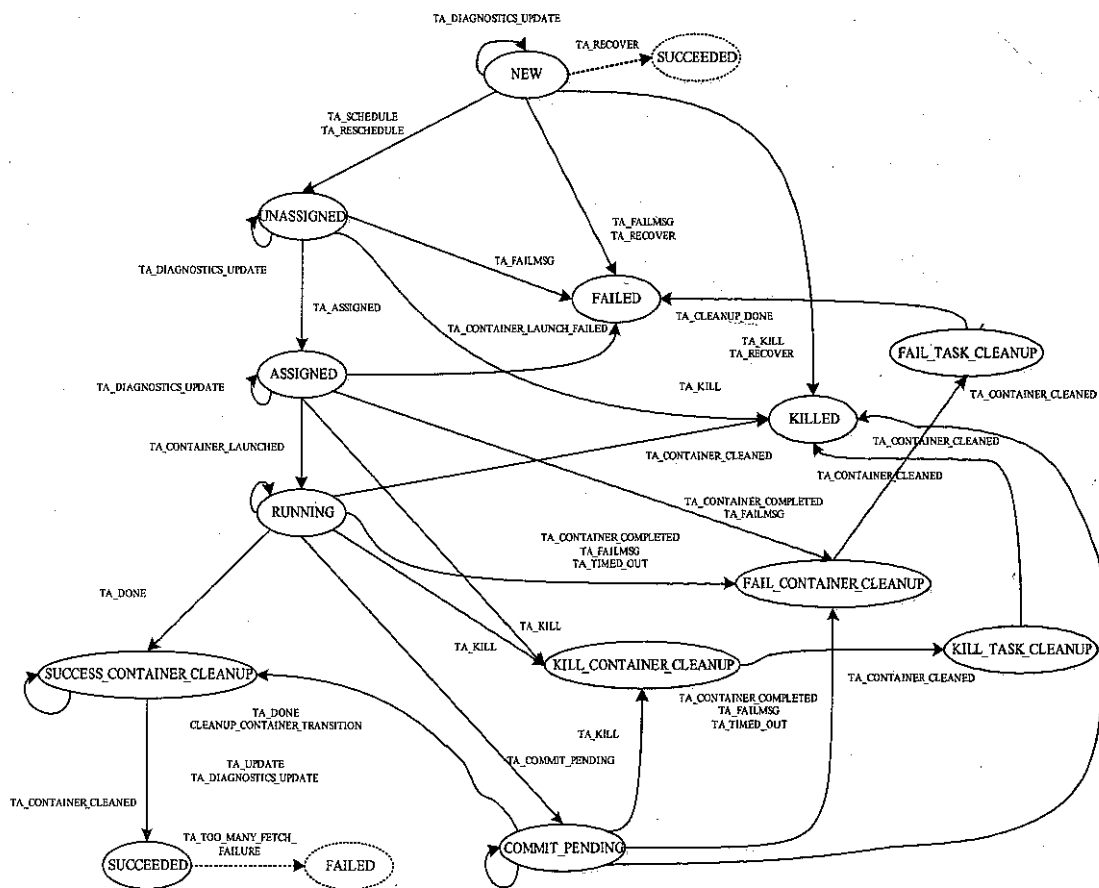


图 8-8 TaskAttempt 状态机包括的状态和事件

(1) TaskAttempt 状态

TaskAttempt 状态包括:

- **NEW**: 任务实例初始状态。当一个任务实例被创建时, 状态被置为 NEW。
- **UNASSIGNED**: 等待分配资源所处的状态。当一个任务实例被创建后, 它会发出一个资源申请请求, 等待 MRAppMaster 为它申请和分配资源。
- **ASSIGNED**: 任务实例获取到一个 Container。
- **RUNNING**: 任务实例成功启动。MRAppMaster 将资源分配给某个任务后, 会与对应的 NodeManager 通信, 以启动 Container。只要 Container 启动成功, 则任务实例启动成功。
- **COMMIT_PENDING**: 任务实例等待提交最终结果。任务实例运行完成后, 需向

MRAppMaster 请求提交最终结果,一旦提交成功后,该任务的其他实例将被杀死。

- SUCCESS_CONTAINER_CLEANUP: Container 运行完成,等待清理空间。
- SUCCEEDED: 成功清理完成 Container 空间。任务实例运行完成后,需清理它使用的 Container 占用的空间,只有该空间清理完成后,才认为任务实例运行完成。
- FAIL_CONTAINER_CLEANUP: 清理运行失败 Container 时所处的状态。当一个 Container 运行失败后,需清理它占用的资源,比如临时文件、临时数据等。
- FAIL_TASK_CLEANUP: 清理完成运行失败的 Container 时所处的状态。
- FAILED: 任务实例运行失败后所处的状态。
- KILL_CONTAINER_CLEANUP: 清理被杀死的 Container 时所处的状态。当一个 Container 被杀死后,需清理它占用的资源,比如临时文件、临时数据等。
- KILL_TASK_CLEANUP: 清理完成被杀死的 Container 时所处的状态。
- KILLED: 任务实例被杀死后所处的状态。

注意 当任务实例处于 RUNNING 状态时,可能直接转换为 SUCCESS_CONTAINER_CLEANUP 状态,也可能先转换为 COMMIT_PENDING 状态,然后再转换为 SUCCESS_CONTAINER_CLEANUP 状态,这主要是由于 MRAppMaster 可从两条途径获知 Container 和 Task 的运行完成状态:一个是通过 ResourceManager (MRAppMaster 与 ResourceManager 中维持一个心跳信息),另外一个直接通过 Task Attempt (每个 Task Attempt 与 MRAppMaster 之间有专用的 RPC 协议)。这两条途径是独立的,没有先后顺序之分。如果 MRAppMaster 首先从 ResourceManager 上获知 Container 运行完成(实际上就是任务实例运行完成)信息,则任务实例直接从 RUNNING 转换为 SUCCESS_CONTAINER_CLEANUP 状态,如果首先从 TaskAttempt 中获知任务运行完成信息。则将首先转换为 COMMIT_PENDING 状态,然后再转换为 SUCCESS_CONTAINER_CLEANUP 状态。

(2) TaskAttempt 事件

TaskAttempt 事件包括:

- TA_SCHEDULE: 一个任务实例开始调度。作业启动完成后(作业收到 JOB_SETUP_COMPLETED 事件),开始调度 Map Task 和 Reduce Task,同时每个 Task 会创建一个任务实例。
- TA_RESCHEDULE: 为任务重新调度一个实例。一个任务只要有曾经失败过的实例,则此后它所有的实例被调度时,均会触发一个 TA_RESCHEDULE 事件。
- TA_RECOVER: 任务实例恢复时触发的事件。当 ResourceManager 重启时,将从日志中恢复之前作业的运行状态,截至本书截稿时,仅当一个任务实例处于 KILLED、FAILED 或者 SUCCEEDED 状态之一时,才会进行恢复,其他状态(比如 RUNNING)的任务实例需重新运行。
- TA_KILL: 杀死一个任务实例。在很多情况下,任务实例可能收到 TA_KILL 事件,比如客户端发送杀死任务实例请求、ResourceManager 主动向 MRAppMaster 发送杀

死任务请求等。

- **TA_FAILMSG**：任务实例运行失败。多种情况下会触发该事件，包括本地文件读写错误、Shuffle 阶段错误、任务在一定时间内未汇报进度、内存使用量超过期望值或者其他运行过程中出现的错误等。
- **TA_DIAGNOSTICS_UPDATE**：更新任务诊断信息。通常而言，当发生异常时，会触发该事件，包括任务实例运行失败或被杀死、Container 启动失败等。需要注意的是，该事件不会改变运行实例的当前状态。
- **TA_ASSIGNED**：任务实例获取到资源。MRAppMaster 收到 ResourceManager 分配的资源后，进一步分配给各个任务，如果一个任务实例获得资源，则会触发一个 TA_ASSIGNED 事件。
- **TA_CONTAINER_LAUNCHED**：任务实例对应的 Container 成功启动。MRAppMaster 将资源分配给某个任务后，会与对应的 NodeManager 通信，以启动 Container。
- **TA_CONTAINER_LAUNCH_FAILED**：任务实例对应的 Container 启动失败。
- **TA_CONTAINER_COMPLETED**：Container 运行结束。当一个 Container 运行结束后，MRAppMaster 可直接从 ResourceManager 上获知。
- **TA_UPDATE**：任务实例运行更新状态。各个任务运行实例需定期向 MRAppMaster 汇报进度和状态，每次汇报都会触发一个 TA_UPDATE 事件。
- **TA_DONE**：任务实例运行完成。一个任务可能对应多个正在运行的实例，但只有最先完成的任务实例才最终提交成功，其他实例将被杀死。
- **TA_COMMIT_PENDING**：任务实例请求提交最终结果。
- **TA_TIMED_OUT**：任务实例超时。任何一个正在运行的任务实例，必须每隔一段时间向 MRAppMaster 汇报进度和状态，否则 MRAppMaster 认为该任务实例处于僵死状态，会将它杀死。
- **TA_CONTAINER_CLEANED**：清理完成 Container 占用的空间。一个任务实例运行完成后，MRAppMaster 会调用 ContainerManager#stopContainer 函数，以清理 Container 占用的空间。
- **TA_CLEANUP_DONE**：清理任务实例完成空间。当任务实例运行失败或者被杀死时，需清理它占用的磁盘空间和产生结果，这是通过调用函数 OutputCommitter#abortTask 完成的。
- **TA_TOO_MANY_FETCH_FAILURE**：Reduce Task 远程复制 Map Task 输出结果失败。当 Reduce Task 远程复制一个已经运行完成的 Map Task 输出数据时，可能因为磁盘或者网络等原因，导致数据损坏或者数据丢失，这时会触发一个 TA_TOO_MANY_FETCH_FAILURE 事件，从而触发 MRAppMaster 重新调度执行该 Map Task。

8.5 资源申请与再分配

ContainerAllocator 是 MRAppMaster 中负责资源申请和分配的模块。用户提交的作业被分解成 Map Task 和 Reduce Task 后, 这些 Task 所需的资源统一由 ContainerAllocator 模块负责从 ResourceManager 中申请, 而一旦 ContainerAllocator 得到资源后, 需采用一定的策略进一步分配给作业的各个任务。

在 YARN 中, 作业的资源需求可描述为 5 元组: `<priority, hostname, capability, containers, relax_locality>`, 分别表示作业优先级、期望资源所在的 host、资源量 (当前支持内存与 CPU 两种资源)、Container 数目、是否松弛本地性, 比如:

```
<10, "node1", "memory:1G,CPU:1", 3, true> // 优先级是一个正整数, 优先级值越小, 优先级越高
<10, "node2", "memory:2G,CPU:1", 1, false> // 1 个必须来自 node2 上大小为 2GB 内存、1 个 CPU 的 Container (不能来自 node2 所在的机架或者其他节点)
<2, "*", "memory:1G,CPU:1", 20, false> // * 表示这样的资源可来自任意一个节点, 即不考虑数据本地性
```

ContainerAllocator 周期性通过心跳与 ResourceManager 通信, 以获取已分配的 Container 列表、完成的 Container 列表、最近更新的节点列表等信息, 而 ContainerAllocator 根据这些信息完成相应的操作。

8.5.1 资源申请

本小节我们对资源申请的相关知识。下面首先介绍资源申请的过程。

1. 资源申请过程分析

当用户提交作业之后, MRAppMaster 会为之初始化, 并创建一系列 Map Task 和 Reduce Task, 由于 Reduce Task 依赖于 Map task 之间结果, 所以 Reduce Task 会延后调度。在 ContainerAllocator 中, 当 Map Task 数目完成一定比例 (由 `mapreduce.job.reduce.slowstart.completedmaps` 指定, 默认是 0.05, 即 5%) 且 Reduce Task 可允许占用的资源 (Reduce Task 可占用资源比由 `yarn.app.mapreduce.am.job.reduce.rampup.limit` 指定) 能够折合成整数个任务时, 才会调度 Reduce Task。

考虑到 Map Task 和 Reduce Task 之间的依赖关系, 因此, 它们之间的状态转移也是不一样的, 对于 Map Task 而言, 会依次转移到以下几个任务集合中:

```
scheduled->assigned->completed
```

对于 Reduce Task 而言, 则按照以下流程进行:

```
pending->scheduled->assigned->completed
```

其中, pending 表示等待 ContainerAllocator 发送资源请求的任务集合; scheduled 表示已经将资源请求发送给 RM, 但还没有收到分配的资源的任务集合; assigned 是已经收到 RM 分配的资源的任务集合; completed 表示已运行完成的任务集合。

Reduce Task 之所以会多出一个 pending 状态, 主要是为了根据 Map Task 情况调整

Reduce Task 状态（在 pending 和 scheduled 中相互转移）。进一步说，这主要是为了防止 Map Task 饿死，因为在 YARN 中不再有 Map Slot 和 Reduce Slot 的概念（这两个概念从一定程度上减少了作业饿死的可能性），只有内存、CPU 等真实的资源，需要由 ApplicationMaster 控制资源申请的顺序，以防止可能产生的作业饿死。

此外，ContainerAllocator 将所有任务划分成三类，分别是 Failed Map Task、Map Task 和 Reduce Task，并分别赋予它们优先级 5、20 和 10，也就是说，当三种任务同时有资源需求时，会优先分配给 Failed Map Task，然后是 Reduce Task，最后是 Map Task。

总结起来，ContainerAllocator 工作流程如下：

步骤 1 将 Map Task 的资源需求发送给 RM。

步骤 2 如果达到了 Reduce Task 调度条件，则开始为 Reduce Task 申请资源。

步骤 3 如果为某个 Task 申请到了资源，则取消其他重复资源的申请。由于在 HDFS 中，任何一份数据通常有三备份，而对于一个任务而言，考虑到 rack 和 any 级别的本地性，它可能会对应 7 个资源请求，分别是：

```
<20, "node1", "memory:1G", 1, true>
<20, "node2", "memory:1G", 1, true>
<20, "node3", "memory:1G", 1, true>
<20, "rack1", "memory:1G", 1, true>
<20, "rack2", "memory:1G", 1, true>
<20, "rack3", "memory:1G", 1, true>
<20, "*", "memory:1G", 1, true>
```

一旦该任务获取了以上任意一种资源，都会取消其他 6 个的资源申请。

在作业运行过程中，会出现资源重新申请和资源取消的行为，具体如下：

❑ 如果任务运行失败，则会重新为该任务申请资源。

❑ 如果一个任务运行速度过慢，则会为其额外申请资源以启动备份任务（如果启动了推测执行功能）。

❑ 如果一个节点失败的任务数目过多，则会撤销对该节点的所有资源的申请请求。

2. 相关类介绍

ContainerAllocator 实际上是一接口，它只定义了三个事件：CONTAINER_REQ、CONTAINER_DEALLOCATE 和 CONTAINER_FAILED，分别表示请求 Container、释放 Container 和 Container 运行失败。

ContainerAllocator 的实现是 RMContainerAllocator，它只接收和处理 ContainerAllocator 接口中定义的三种事件，它的运行是这三种事件驱动的。

ContainerAllocator 实现类的关系如图 8-9 所示。

RMContainerAllocator 中最核心的框架是维护了一个心跳信息，在 RMCommunicator 类中的实现如下：

```
while (!stopped.get() && !Thread.currentThread().isInterrupted()) {
    try {
```

```

Thread.sleep(rmPollInterval);
try {
    heartbeat();
} catch (YarnException e) {
    LOG.error("Error communicating with RM: " + e.getMessage(), e);
    return;
} catch (Exception e) {
    LOG.error("ERROR IN CONTACTING RM. ", e);
    continue;
}
lastHeartbeatTime = context.getClock().getTime();
executeHeartbeatCallbacks();
} catch (InterruptedException e) {
    LOG.warn("Allocated thread interrupted. Returning.");
    return;
}
}

```

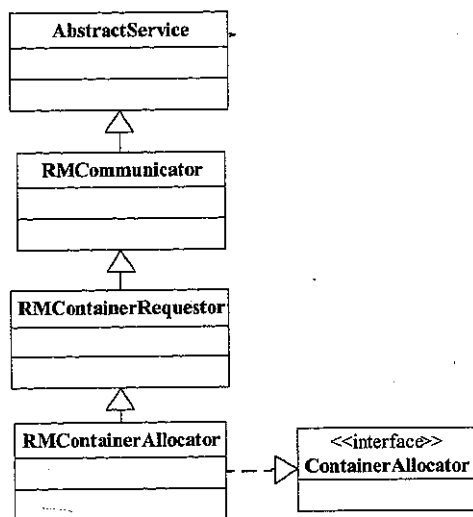


图 8-9 ContainerAllocator 实现类关系图

其中，heartbeat() 函数的定义（在 RMContainerAllocator 类中）如下：

```

protected synchronized void heartbeat() throws Exception {
    scheduleStats.updateAndLogIfChanged("Before Scheduling: ");
    List<Container> allocatedContainers = getResources();
    if (allocatedContainers.size() > 0) {
        scheduledRequests.assign(allocatedContainers);
    }
    ...
}

```

其中，getResources() 函数用于向 RM 发送心跳信息，并处理心跳应答。需要注意的是，大部分情况下，心跳信息中并不包含新的资源请求信息，即空的心跳信息，这种心跳有以

下几个作用：

- 周期性发送心跳，告诉 RM 自己还活着。
- 周期性询问 RM，以获取新分配的资源 and 各个 Container 运行状况。

assign() 函数作用是将收到的 Container 分配给某个任务，如果这个 Container 无法分配下去（比如内存空间不够），则在下次心跳中通知 RM 释放该 Container，如果 Container 可以分下去，则会释放对应任务的其他资源请求，同时会向 TaskAttempt 发送一个 TA_ASSIGNED 事件，以通知 ContainerLauncher 启动 Container。

除了新分配和已经运行完成的 Container 列表外，ContainerAllocator 还会从 RM 中获取节点更新列表，这个列表给出了最近发生变化的节点，比如新增节点、不可用节点等，当前 ContainerAllocator 仅处理了不可用节点，即一旦发现节点不可用，ContainerAllocator 会将该节点上正运行的任务的状态置为被杀死状态，并重新为这些任务申请资源。

8.5.2 资源再分配

一旦 MRAppMaster 收到新分配的 Container 后，会将这些 Container 进一步分配给各个任务，Container 分配过程如下：

步骤 1 判断收到的 Container 包含的资源是否满足要求，如果不满足，则通过下次心跳通知 ResourceManager 释放该 Container。

步骤 2 判断收到的 Container 所在节点是否被加入黑名单中，如果是，则寻找一个与该 Container 匹配的任务，并重新为该任务申请资源，同时通过下次心跳通知 ResourceManager 释放该 Container。

注意 当作业在一个节点上失败的任务实例数目超过一定上限（通过参数 `mapreduce.job.maxtaskfailures.per.tracker` 配置，默认值为 3，管理员可通过参数 `yarn.app.mapreduce.am.job.node-blacklisting.enable` 配置是否启用该功能，默认情况下为 true，表示启动该功能）后，该节点将被加入到黑名单中；为了防止大量节点被加入黑名单导致作业无法运行完成或者运行效率过低，YARN 允许管理员通过参数 `yarn.app.mapreduce.am.job.node-blacklisting.ignore-threshold-node-percent` 设置最多被加入黑名单的节点比例，默认是 33，即最多 33% 的集群节点可被加入黑名单。

步骤 3 根据 Container 的优先级，将它分配给对应类型的任务。如果优先级为 `PRIORITY_FAST_FAIL_MAP`（对应数值为 5），则分配给失败的 Map Task；如果优先级为 `PRIORITY_REDUCE`，则分配给 Reduce Task；否则，分配给正常的 Map Task。对于前两种情况，ContainerAllocator 直接从对应队列中取出第一个任务即可，对于最后一种情况，则依次尝试从 `node-local`（输入数据与 Container 在同一个节点）、`rack-local`（输入数据与 Container 在同一个机架）和 `no-local`（输入数据与 Container 不在同一个机架）几个任务列表中查找 Map Task。

8.6 Container 启动与释放

ContainerLauncher 负责与各个 NodeManager 通信, 以启动或者释放 Container。在 YARN 中, 运行 Task 所需的全部信息被封装到 Container 中, 包括所需资源、依赖的外部文件、JAR 包、运行时环境变量、运行命令等。ContainerLauncher 通过 RPC 协议与 ContainerManager 通信, 以控制 Container 的启动与释放, 进而控制任务的执行 (比如启动任务、杀死任务等), ContainerManager 协议定义了三个 RPC 接口, 具体如下:

```
StartContainerResponse startContainer(StartContainerRequest request)
    throws YarnRemoteException; // 启动一个 container
StopContainerResponse stopContainer(StopContainerRequest request)
    throws YarnRemoteException; // 停止一个 container
GetContainerStatusResponse getContainerStatus(
    GetContainerStatusRequest request) throws YarnRemoteException; // 获取一个
container 运行情况
```

ContainerLauncher 是一个 Java 接口, 它定义了两种事件:

- CONTAINER_REMOTE_LAUNCH。启动一个 Container。当 ContainerAllocator 为某个任务申请到资源后, 会将运行该任务相关的所有信息封装到 Container 中, 并要求对应的节点启动该 Container。需要注意的是, Container 中运行的任务对应的数据处理引擎与 MRv1 中完全一致, 仍为 MapTask 和 ReduceTask, 正因如此, MRv1 中的程序与 YARN 中的 MapReduce 程序完全兼容。
- CONTAINER_REMOTE_CLEANUP。停止 / 杀死一个 Container。存在多种可能触发该事件的行为, 常见的有:
 - 推测执行时一个任务运行完成, 需杀死另一个同输入数据的任务;
 - 用户发送一个杀死任务请求;
 - 任意一个任务运行结束时, YARN 会触发一个杀死任务的命令, 以释放对应 Container 占用的资源。

尤其需要注意的是第三种情况, YARN 作为资源管理系统, 应确保任何一个任务运行结束后资源得到释放, 否则会造成资源泄露。而实现这一要求的可行方法是, 任何一个任务结束后, 不管它对应的资源是否得到释放, YARN 均会主动显式检查和回收资源 (container)。

ContainerLauncher 接口由 ContainerLauncherImpl 类实现, 它是一个服务, 接收和处理来自事件调度器发送过来的 CONTAINER_REMOTE_LAUNCH 和 CONTAINER_REMOTE_CLEANUP 两种事件, 它采用了线程池方式并行处理这两种事件。

对于 CONTAINER_REMOTE_LAUNCH 事件, 它会调用 Container.launch() 函数与对应的 NodeManager 通信, 以启动 Container (可以同时启动多个 Container), 代码如下:

```
proxy = getCMPProxy(containerMgrAddress, containerID); // 构造一个 RPC client
```

```

ContainerLaunchContext containerLaunchContext =
    event.getContainer();
StartContainerRequest startRequest =
    StartContainerRequest.newInstance(containerLaunchContext,
        event.getContainerToken());
List<StartContainerRequest> list = new ArrayList<StartContainerRequest>();
list.add(startRequest);
StartContainersRequest requestList =
    StartContainersRequest.newInstance(list);
// 调用 RPC 函数, 获取返回值
StartContainersResponse response =
    proxy.getContainerManagementProtocol().startContainers(requestList);

```

启动的 Container 中封装的任务对应的数据处理引擎仍为 MRv1 中的 MapTask 和 ReduceTask, 但 YARN 对其进行了一些优化, 具体将在 8.9 节中介绍。

对于 CONTAINER_REMOTE_CLEANUP 事件, 它会调用 Container. kill() 函数与对应的 NodeManager 通信, 以杀死 Container 释放资源 (可以同时杀死多个 Container), 代码如下:

```

proxy = getCMPProxy(this.containerMgrAddress, this.containerID);
// kill the remote container if already launched
List<ContainerID> ids = new ArrayList<ContainerID>();
ids.add(this.containerID);
StopContainersRequest request = StopContainersRequest.newInstance(ids);
StopContainersResponse response =
    proxy.getContainerManagementProtocol().stopContainers(request);

```

总之, ContainerLauncherImpl 是一个非常简单的服务, 其最核心的代码组织方式是“队列+线程池”, 以处理事件调度器发送过来的 CONTAINER_REMOTE_LAUNCH 和 CONTAINER_REMOTE_CLEANUP 两种事件。

8.7 推测执行机制

在分布式集群环境下, 因软件 Bug、负载不均衡或者资源分布不均等原因, 造成同一个作业的多个任务之间运行速度不一致, 有的任务运行速度明显慢于其他任务 (比如某个时刻, 一个作业的某个任务进度只有 10%, 而其他所有 Task 已经运行完毕), 则这些任务将拖慢作业的整体执行进度。为了避免这种情况发生, 应用推测执行 (Speculative Execution) 机制, Hadoop 会为该任务启动一个备份任务, 让该备份任务与原始任务同时处理一份数据, 谁先运行完, 则将谁的结果作为最终结果。

8.7.1 算法介绍

MRv2 实现的推测执行算法与 MRv1 中的不同^①，MRv2 实现的推测算法重点关注新启动的备份任务实例是否有潜力比当前正在运行的任务实例完成得更早。如果通过一定的算法推测某一时刻启动备份任务实例，该备份任务实例肯定会比当前任务实例完成得晚，那么启动该备份任务实例只会浪费更多的资源；然而，如果推测备份任务实例比当前任务实例完成得早，则启动备份任务实例会加快数据处理，且备份任务实例完成得越早，启动备份任务实例的价值越大。

假设某一时刻，任务 T 的当前运行实例执行进度为 progress，则可通过一定的算法推测出该任务实例的最终完成时刻 $\text{estimatedEndTime}_1$ ，从另一方面，如果此刻为该任务启动一个备份任务实例，则可推断出它可能的完成的时刻 $\text{estimatedEndTime}_2$ ，于是可得出以下几个公式：

$$\begin{aligned}\text{estimatedEndTime}_1 &= \text{estimatedRunTime} + \text{taskAttemptStartTime} \\ \text{estimatedRunTime} &= (\text{currentTimestamp} - \text{taskAttemptStartTime}) / \text{progress} \\ \text{estimatedEndTime}_2 &= \text{currentTimestamp} + \text{averageRunTime}\end{aligned}$$

其中，currentTimestamp 为当前时刻；taskAttemptStartTime 为该任务实例启动时刻；averageRunTime 为已经成功运行完成的任务的平均运行时间。

很明显，如果 $\text{estimatedEndTime}_2$ 大于 $\text{estimatedEndTime}_1$ ，则没必要启动备份任务实例，因为即使启动了，它的完成时刻也会大于当前正在运行任务实例的完成时刻，只有当 $\text{estimatedEndTime}_2$ 小于 $\text{estimatedEndTime}_1$ 时，才有必要启动备份任务实例。MRAppMaster 总是选择 $\{\text{estimatedEndTime}_1 - \text{estimatedEndTime}_2\}$ 差值最大的任务，并为之启动一个备份任务实例，且启动备份任务实例之前需检查是否满足以下条件：

- 每个任务最多只能有一个备份任务实例；
- 已经完成的任务数目比例不小于 MINIMUM_COMPLETE_PROPORTION_TO_SPECULATE (0.05，即 5%)，只有这样才能有足够的历史信息估算 $\text{estimated-EndTime}_2$ 。

为了防止大量任务同时启动备份任务造成资源浪费，默认是实现中，每个作业同时启动的备份任务数目有一定的上限，该数目是以下三个数值的最大值：

- MINIMUM_ALLOWED_SPECULATIVE_TASKS (常量 10)
- PROPORTION_TOTAL_TASKS_SPECULATABLE (常量 0.01) * totalTaskNumber
- PROPORTION_RUNNING_TASKS_SPECULATABLE (常量 0.1) * numberRunningTasks

其中，totalTaskNumber 为 Map Task 或者 Reduce Task 总数，numberRunningTasks 为当前正在运行的 Map Task 或者 Reduce Task 数目。

为了便于读者理解，再次提炼一下以上推测执行算法的核心思想：某一时刻，判断一个任务是否拖后腿或者是否是值得为其启动备份任务时，采用的方法为，先假设为其

① MRv1 中推测执行机制算法可参考《Hadoop 技术内幕：深入解析 MapReduce 架构设计与实现原理》一书中的 6.6 节。

启动一个备份任务，则可估算备份任务的完成时间 $\text{estimatedEndTime}_2$ ；同样，如果按照此刻该任务的计算速度，可估算该任务最有可能的完成时间 $\text{estimatedEndTime}_1$ ，这样 $\text{estimatedEndTime}_1$ 与 $\text{estimatedEndTime}_2$ 之差越大，表明为该任务启动备份任务的价值越大，即倾向于为这样的任务启动备份任务。这种算法的最大优点是可最大化备份任务的有效率，其中有效率是指有效备份任务数与所有备份任务数的比值，有效备份任务是指完成时间早于原始任务完成时间的备份任务（即带来实际收益的备份任务）。备份任务的有效率越高，推测执行算法越优秀，带来的收益也就越大。

推测执行机制实际上采用了经典的算法优化方法：以空间换时间，它同时启动多个相同任务处理相同的数据，并让这些任务竞争以缩短数据处理时间，显然这种方法需要占用更多的计算资源，在集群资源紧缺的情况下，应合理使用该机制，争取在多用少量资源情况下，减少大作业的计算时间。

8.7.2 推测执行相关类

在 MRv2 中，想要实现一个推测执行算法必须实现 `Speculator` 接口，`Speculator` 同时也是一个事件处理器，负责处理 `SpeculatorEvent` 类型的事件，目前主要有四种该类型的事件，分别如下：

- `ATTEMPT_STATUS_UPDATE`：任务实例状态更新时触发的事件，`Speculator` 可通过该事件获取任务实例最新的运行状态和进度。
- `ATTEMPT_START`：一个新的任务实例启动时触发的事件，`Speculator` 收到该事件后将启动对该实例的监控。
- `TASK_CONTAINER_NEED_UPDATE`：Container 数量变化时将触发的事件。
- `JOB_CREATE`：作业被创建时触发的事件，`Speculator` 收到该事件后将做一些初始化工作。

在 `MRAppMaster` 中，用户可单独为 Map Task 和 Reduce Task 设置是否启用推测执行机制，这分别由参数 `mapreduce.map.speculative` 和 `mapreduce.reduce.speculative` 设置，默认情况下这两个参数值均为 `true`，表示启用推测执行机制。用户在实际应用中可选择性开启或者关闭某一类任务的推测执行机制，比如应用程序的 Reduce Task 负载严重不均衡，可关闭 Reduce Task 推测执行机制。

推测执行实现类是通过配置参数 `yarn.app.mapreduce.am.job.speculator.class` 指定的，默认为 `DefaultSpeculator`，它实现了 8.7.1 节描述的算法。`DefaultSpeculator` 每隔一段时间会扫描一次所有正在运行的任务，如果一个任务可以启动备份任务，则会向 Task 发出一个 `T_ADD_SPEC_ATTEMPT` 事件，以启动另外一个任务实例（备份任务）。

如图 8-10 所示，`DefaultSpeculator` 依赖于一个执行时间估算器，默认采用了 `LegacyTaskRuntimeEstimator`，此外 MRv2 还提供了另外一个实现——`ExponentiallySmoothedTaskRuntimeEstimator`，该实现采用了平滑算法对结果进行平滑处理。

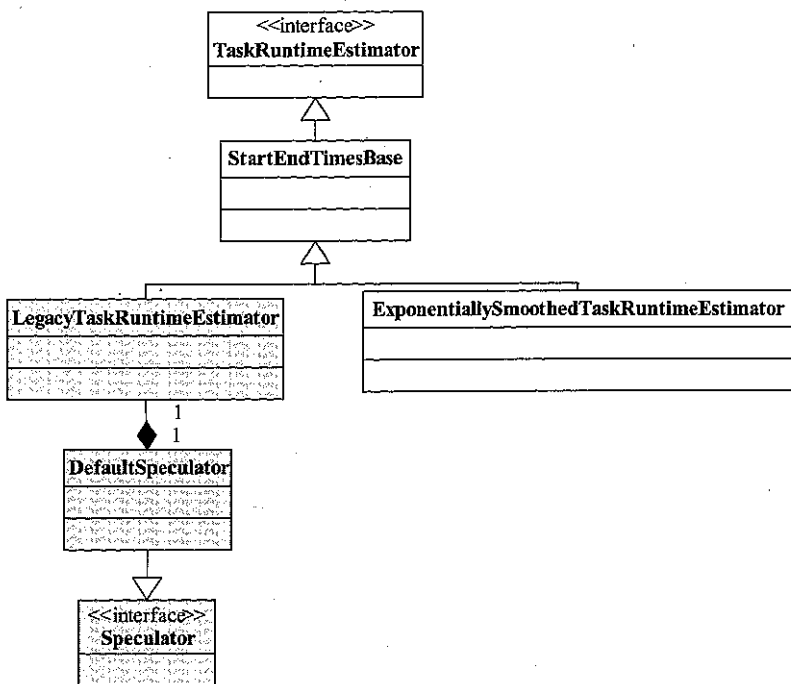


图 8-10 推测执行实现类关系图

8.8 作业恢复

当一个作业的 MRAppMaster 运行失败时, ResourceManager 会将其迁移到另外一个节点上执行。为了避免作业重新计算带来的资源浪费, MRAppMaster 在运行过程中会(在 HDFS 上)记录一些运行时日志以便重启时恢复之前的作业运行状态。

从作业恢复粒度角度看,当前存在三种不同级别的恢复机制,级别由低到高依次是作业级别、任务级别和记录级别,其中级别越低实现越简单,但造成的资源浪费越严重。当前 MRAppMaster 采用了任务级别的恢复机制^①,即以任务为基本单位进行恢复,这种机制是基于事务型日志完成作业恢复的,它只关注两种任务:运行完成的任务和未运行完成的任务。作业执行过程中,MRAppMaster 会以日志的形式将作业以及任务状态记录下来,一旦 MRAppMaster 重启,则可从日志中恢复作业的运行状态,其中已经运行完成的任务无须再运行,而未开始运行或者运行中的任务需重新运行。

作业恢复采用了 Redo 日志的方式,即作业运行过程中,记录作业或者任务运行日志,当 MRAppMaster 重启恢复作业时,重做这些日志,以重构作业或者任务的内存信息,进而让作业沿着之前的断点继续开始执行。

① 参考网址 <https://issues.apache.org/jira/browse/HADOOP-3245>。

MRAppMaster 以事件的形式记录作业和任务运行过程，它记录的事件类型及其含义如表 8-2 所示。

表 8-2 作业恢复相关的事件

事件 (HISTORYEVENT)	事件类型 (AVRO EVENTTYPE)	含 义
AMStartedEvent	AM_STARTED	启动 MRAppMaster
JobSubmittedEvent	JOB_SUBMITTED	作业提交
JobInitiatedEvent	JOB_INITED	作业初始化
JobInfoChangeEvent	JOB_INFO_CHANGED	作业信息改变
JobFinishedEvent	JOB_FINISHED	作业运行完成
JobPriorityChangeEvent	JOB_PRIORITY_CHANGED	作业优先级改变
JobStatusChangedEvent	JOB_STATUS_CHANGED	作业状态改变
JobUnsuccessfulCompletionEvent	JOB_FAILED JOB_KILLED	作业未成功运行完成（失败或者被杀死）
TaskStartedEvent	TASK_STARTED	任务启动
TaskFinishedEvent	TASK_FINISHED	任务运行完成
TaskFailedEvent	TASK_FAILED	任务运行失败
TaskUpdatedEvent	TASK_UPDATED	任务信息更新
TaskAttemptStartedEvent	MAP_ATTEMPT_STARTED REDUCE_ATTEMPT_STARTED	任务实例启动
MapAttemptFinishedEvent	MAP_ATTEMPT_FINISHED	Map 任务实例运行完成
TaskAttemptUnsuccessfulCompletionEvent	MAP_ATTEMPT_FAILED MAP_ATTEMPT_KILLED REDUCE_ATTEMPT_FAILED REDUCE_ATTEMPT_KILLED	任务实例未成功运行完成
ReduceAttemptFinishedEvent	REDUCE_ATTEMPT_FINISHED	Reduce 任务实例运行完成
NormalizedResourceEvent	NORMALIZED_RESOURCE	资源归一化

MRAppMaster 采用了开源数据序列化工具 Apache Avro^①记录这些事件（定义参考 Events.avpr）。Avro（读音类似于 [ævrə]）是 Hadoop 的一个子项目，由 Hadoop 的创始人 Doug Cutting 牵头开发。Avro 是一个数据序列化系统，通常用于支持大批量数据交换和跨语言 RPC 的应用。它的主要特点有：支持二进制序列化方式，可以便捷、快速地处理大量数据；支持动态模式，无须生成代码便可以方便地处理 Avro 格式数据；支持跨语言 RPC 等。Avro 最初设计目的是替换现有 Hadoop RPC 框架，进而使得 Hadoop RPC 具有支持跨语言、向后兼容性等优点，然而，由于 Hadoop 2.0 设计之初 Avro 稳定性不够，因此尚未将之用在 Hadoop RPC 中。目前 Avro 在 Hadoop 2.0 中仅得到了非常有限的应用，而本节介绍的 MRAppMaster 作业恢复便是一种典型应用。

当作业参数 yarn.app.mapreduce.am.job.recovery.enable 被置为 true 时（默认情况下就

① <http://avro.apache.org/>.

是 true)，MRAppMaster 从第二次运行开始，将尝试读取前一次记录的日志，通过日志回放的方式恢复作业运行前的状态。整个过程如下。

步骤 1 MRAppMaster 初始化时，解析前一次记录的事件日志，并将已经运行完成的任务存放到 completedTasksFromPreviousRun（类型为 Map<TaskId, TaskInfo>）中；

步骤 2 MRAppMaster 构建新的 JobImpl 对象，并将 completedTasksFromPreviousRun 传递给该对象的构造函数；

步骤 3 当 JobImpl 经过初始化后，开始调度内部的 TaskImpl，如果任务处于 completed-TasksFromPreviousRun 中，则向 TaskImpl 发行一个 T_RECOVER 事件，以恢复该任务之前的运行状态和基本信息，否则按照正常逻辑，向 TaskImpl 发送一个 T_SCHEDULE 事件；

步骤 4 TaskImpl 收到 T_RECOVER 事件后，通过上次运行信息恢复状态，即根据日志信息将状态直接转移到最终状态 FAILED、KILLED 或者 SUCCEED。

当前 MRAppMaster 的作业恢复机制仅能做到恢复上次已经运行完成的任务，对于正在运行的任务，则在前一次 MRAppMaster 运行实例退出时由 ResourceManager 强制将其杀死并回收资源。很明显，目前实现会造成大量的计算浪费，而更为高效的算法应该避免那些正在运行的任务重新运行。当然这需要 ResourceManager 本身支持，即当一个 MRAppMaster 失败退出时不会强制回收它正在使用的资源，而是将之划归到下次启动的 MRAppMaster 中，这会为 ResourceManager 引入一定的复杂性，因此社区尚未考虑这一优化机制。

8.9 数据处理引擎

MRAppMaster 仍采用了 MRv1 中的数据处理引擎，分别由数据处理引擎 MapTask 和 ReduceTask 完成 Map 任务和 Reduce 任务的处理^①。但相比于 MRv1，MRAppMaster 对这两个引擎进行了优化，这些优化主要体现在 Shuffle 阶段，具体如下。

1. Map 端——用 Netty 代替 Jetty

MRv1 版本中，TaskTracker 采用了 Jetty 服务器处理来自各个 Reduce Task 的数据读取请求。由于 Jetty 采用了非常简单的网络模型，因此性能比较低。在 Hadoop 2.0 中，MRAppMaster 改用 Netty——另一种开源的客户/服务器端编程框架，由于它内部采用了 Java NIO 技术，故其相比 Jetty 更加高效。Netty 社区也比 Jetty 的更加活跃，且稳定性更好。

2. Reduce 端——批拷贝

MRv1 版本中，在 Shuffle 过程中，Reduce Task 会为每个数据分片建立一个专门的 HTTP 连接（One-connection-per-map），即使多个分片同时出现在一个 TaskTracker 上也是如此。为了提高数据复制效率，Hadoop 2.0 尝试采用批拷贝技术：不再为每个 Map Task 建

① Map Task 和 Reduce Task 内部实现可参考书籍《Hadoop 技术内幕：深入解析 MapReduce 架构设计与实现原理》中的第 8 章。

立一个 HTTP 连接, 而是为同一个 TaskTracker 上的多个 Map Task 建立一个 HTTP 连接, 进而能够一次读取多个数据分片, 具体如图 8-11 所示。

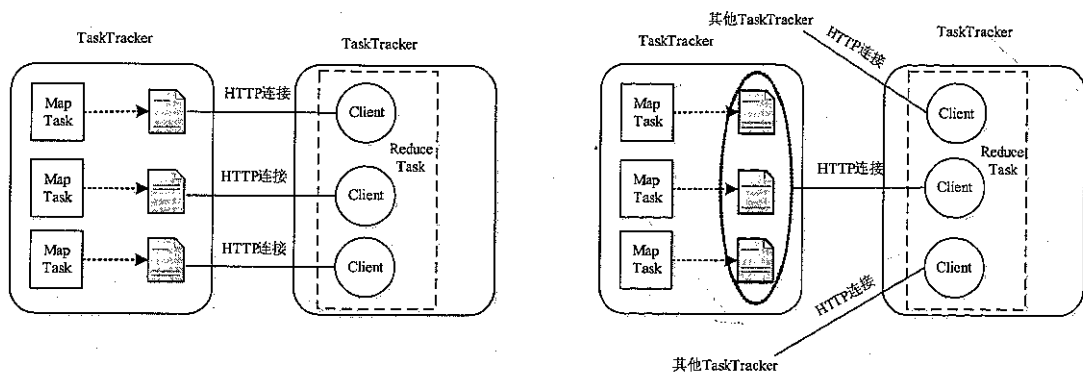


图 8-11 One-connection-per-map 与批拷贝模型对比

3. Reduce 端——Shuffle 和排序插件化

下面我们介绍 Shuffle 和排序四种插件化。

(1) Shuffle 插件化

MapReduce 模型的最大瓶颈在 Shuffle 阶段, MRv1 中 Shuffle 实现不尽理想, 而 Hadoop 2.0 提出了更为合理高效的实现: 将 Shuffle 代码独立出来, 成为一个可插拔模块, 用户可根据需要实现自己的 Shuffle, 比如可使用 RDMA 代替 HTTP 用作数据传输协议^①。

定制化一个 Shuffle 插件需要一个运行在 NodeManager 上的 `org.apache.hadoop.yarn.server.nodemanager.containermanager.AuxServices.AuxiliaryService` 实现 (AuxiliaryService 是一个抽象类) 和一个运行在 Reduce Task 中的 `org.apache.hadoop.mapred.ShuffleConsumerPlugin` 实现 (ShuffleConsumerPlugin 是一个接口), 它们的默认值分别是:

- `org.apache.hadoop.mapred.ShuffleHandler`
- `org.apache.hadoop.mapreduce.task.reduce.Shuffle`

(2) 排序插件化

MapReduce 固有的实现中采用了基于排序的数据聚集算法, 且排序统一采用了自己实现的快速排序算法。这种算法会对所有数据进行全排序, 这对于很多应用而言是低效的, 因此 MRv2 将排序插件化, 用户可自定义一个排序算法实现诸如基于 Hash 的聚集算法或者 Limit-N 算法。

定制化一个排序插件需要一个运行在 Map Task 中的 `org.apache.hadoop.mapred.MapOutputCollector` 实现 (MapOutputCollector 是一个接口) 和一个运行在 Reduce Task 中的 `org.apache.hadoop.mapred.ShuffleConsumerPlugin` 实现 (ShuffleConsumerPlugin 是一个接口), 它们的默认值分别是:

① 参考网址 <https://issues.apache.org/jira/browse/MAPREDUCE-4049>。

□ org.apache.hadoop.mapred.MapTask\$MapOutputBuffer

□ org.apache.hadoop.mapreduce.task.reduce.Shuffle

除了 Shuffle HTTP Server 运行在 NodeManager 上之外, 其他插件运行在任务中, 这意味着这些插件可以为每个作业单独设置, 总之插件化配置方法如下。

作业配置 (每个作业可单独配置) 的属性如表 8-3 所示。

表 8-3 作业配置属性

属 性	默 认 值	解 释
mapreduce.job.reduce.shuffle.consumer.plugin.class	org.apache.hadoop.mapreduce.task.reduce.Shuffle	ShuffleConsumerPlugin 实现
mapreduce.job.map.output.collector.class	org.apache.hadoop.mapred.MapTask\$MapOutputBuffer	MapOutputCollector 实现

注: 用户可将这两个属性值放到客户端配置文件 mapred-site.xml 中, 以作为所有作业的默认值。

NodeManager 配置 (添加到所有节点的 yarn-site.xml 文件中) 的属性如表 8-4 所示。

表 8-4 作业配置属性

属 性	默 认 值	解 释
yarn.nodemanager.aux-services	...,mapreduce-shuffle [⊖]	附属服务名称
yarn.nodemanager.aux-services.mapreduce-shuffle.class	org.apache.hadoop.mapred.ShuffleHandler	附属服务实现类

注意 如果用户自己实现了一个附属服务, 且命名为 dongxicheng, 则该名称需要添加到属性 yarn.nodemanager.aux-services 中, 且它的实现类必须通过属性 yarn.nodemanager.aux-services.dongxicheng.class 指定。

8.10 历史作业管理器

为了方便用户查看 MapReduce 历史作业信息, MRAppMaster 提供了一个 JobHistory-Server 服务, 该服务主要由四个子服务组成 (见图 8-12), 具体如下:

□ HistoryClientService。为 Web 界面展示历史作业信息提供后端实现, 它通过调用 JobHistory 中的相关 API 获取作业信息, 比如提交时间、提交用户、Counter 值等, 并将这些信息发送到前端。

□ HSAdminService。管理员可使用 “bin/mapred hsadmin” 命令动态更新 JobHistory-Server 访问和管理权限信息。用户输入该命令后, 后端的 HSAdminService 服务将执

[⊖] YARN 允许用户配置多个附属服务, 每个附属服务名称之间用 “,” 分割, 而 “...” 表示其他已经配置的附属服务, 如果没有, 则删除即可。

行该命令，当前该命令可动态更新以下几种信息：

- **管理员列表。**历史作业查看器的管理员列表（具有启动和停止 JobHistoryServer 服务、更新权限信息等权限）是通过 core-site.xml 中的 mapreduce.jobhistory.admin.acl 属性（默认是“*”，表示任何人都是管理员）设置的。
- **超级用户组列表。**超级用户组信息是通过 core-site.xml 中的 hadoop.proxyuser.groups 和 hadoop.proxyuser.hosts 属性设置的，超级用户组是 Hadoop 引入 Kerberos 安全机制后，为方便上层应用（比如 HBase、Oozie 等）访问 Hadoop 内核服务而新增加的配置项。
- **用户与用户组映射关系。**用户与用户组映射关系是通过 core-site.xml 中的 hadoop.security.group.mapping 属性设置的，它需要是一个实现了 GroupMappingServiceProvider 接口的类，默认实现是 ShellBasedUnixGroupsMapping，即采用 Linux 操作系统的用户和用户组映射关系。
- **AggregatedLogDeletionService。**该服务周期性扫描所有历史作业日志目录，如果一个日志目录存放时间超过 yarn.nodemanager.log.retain-seconds（单位是秒，默认为 $3 \times 60 \times 60$ ，即 3 小时），则直接将其从 HDFS 上删除。
- **JobHistory。**该服务从 HDFS 上读取 MapReduce 历史作业日志，并将其解析成格式化信息，供前端查看。MRAppMaster 将执行完成的作业信息保存到 mapreduce.jobhistory.intermediate-done-dir 指定的目录中，如果用户未设置该属性，则保存到 $\{\text{yarn.app.mapreduce.am.staging-dir}\}/\text{history/done\_intermediate}$ 目录中，其中每个作业保存作业配置文件（后缀为 .xml）和作业执行日志（后缀为 .jhist）两个文件，通过解析作业执行日志，JobHistory 能获取作业执行时的所有信息，包括作业提交时间、作业 Counter、每个任务提交时间和运行时间、每个任务的 Counter 等。

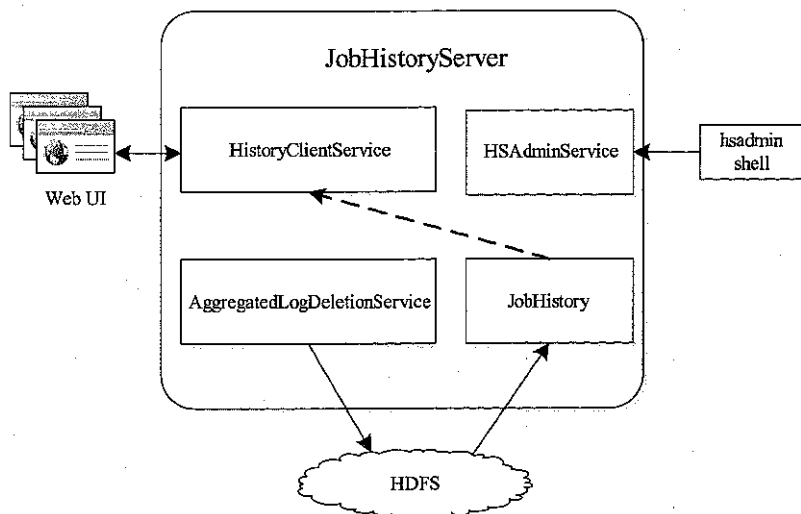


图 8-12 MapReduce 历史作业管理器

8.11 MRv1 与 MRv2 对比

8.11.1 MRv1 On YARN

为了让读者进一步理解本章介绍的 MapReduce On YARN (即 MRv2), 本小节介绍另外一种 MapReduce 运行方式, 即将 MRv1 直接运行在 YARN 上。

MRv1 由 JobTracker 和 TaskTracker 两类服务组成, 为了能够让 MRv1 运行在 YARN 上, 需将 JobTracker 和 TaskTracker 作为 Task 运行在 YARN 中的 Container 中, 一种可行的架构如图 8-13 所示。JobTracker 运行在 MRv1 ApplicationMaster 中, 而 TaskTracker 则运行在该 ApplicationMaster 从 Resource Manager 上申请的各个 Container 中。

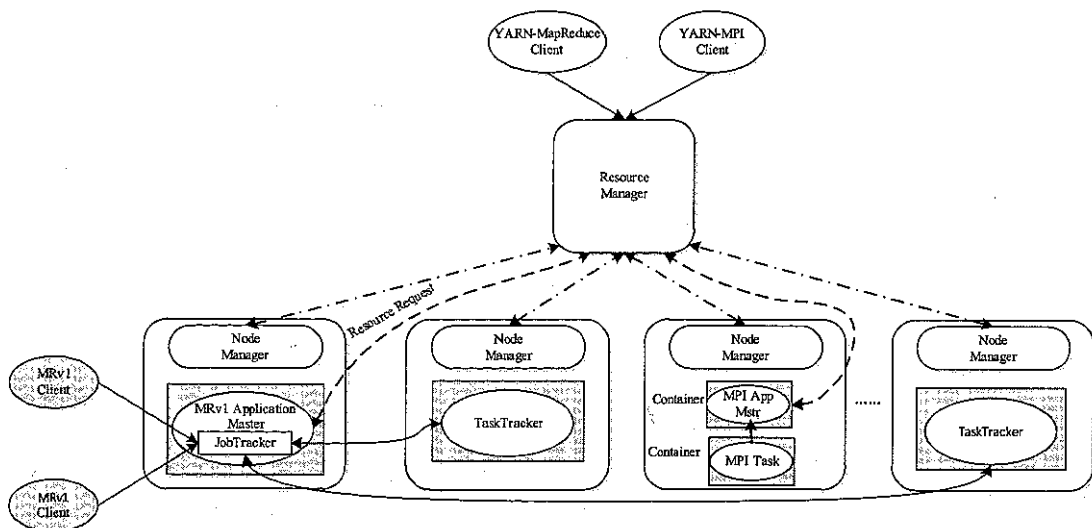


图 8-13 MRv1 On YARN

这种部署方案与传统的 MRv1 部署方式相比, 具有以下优点:

- 计算能力可伸缩。MRv1 ApplicationMaster 可根据当前系统 MapReduce 作业负载调整 TaskTracker 数量以达到计算能力动态伸缩的目的, 即当 MapReduce 作业负载较轻时, 可减少 TaskTracker 数量从而将资源暂时迁移给其他类型的应用程序, 而当 MapReduce 负载较重时, 可通过增加 TaskTracker 数量扩充计算能力。
- 共享存储。通过将 MRv1 运行在 YARN 之上, 既可重用本公司内部 MRv1 版本 (可能有自己的一些优化和改进), 也可以与其他计算框架运行在同一集群中, 共享一个大的底层分布式存储系统, 避免数据在集群间拷贝带来的网络开销和存储开销。

这种方案与 MRv2 相比, 存在以下缺点:

- MR 作业之间相互干扰。由于不同 MR 作业运行在同一个集群中, 共用一个 JobTracker, 因此它们之间会相互干扰, 且一旦 JobTracker 出现故障, 所有作业将无法正常运行。
- 数据本地性难以保证。由于 TaskTracker 只运行在集群中的若干个节点上, 而数据则

可能位于 YARN 的所有节点上，因此 MR 作业的本地性不会太高。

8.11.2 MRv1 与 MRv2 架构比较

正如前面介绍的，MRv1 和 MRv2 在应用程序编程接口和数据处理引擎两方面是相同的，不同之处仅在于运行时环境。MRv2 采用了新的资源管理系统 YARN 和作业管理组件 MRAppMaster 作为运行时环境和资源分配器，使得 MapReduce 框架在扩展性、可用性等方面向前迈进了一大步。表 8-5 从编程接口、运行时环境和数据处理引擎等方面对比了 MRv1 与 MRv2。

表 8-5 MRv1 与 MRv2 比较

	MRv1	MRv2
应用程序编程接口	新旧 API	新旧 API
运行时环境	由 JobTracker 和 TaskTracker 构成	YARN(由 ResourceManager 和 NodeManager 构成) 和 MRAppMaster
数据处理引擎	MapTask/ReduceTask	MapTask/ReduceTask

8.11.3 MRv1 与 MRv2 编程接口兼容性

总体上讲，MRv2 由于重用了 MRv1 的数据处理引擎和应用程序编程接口，因此大部分情况下可以兼容 MRv1 原有的应用程序，但由于 MRv2 对 MRv1 部分代码进行了改进和重构，这导致一些新 API 从一定程度上破坏了向后兼容性。本节将从 JAR 包兼容性、编程接口兼容性等方面详细讨论 MRv2 与 MRv1 在编程接口方面的兼容性问题^①。

(1) JAR 包兼容性

MRv1 中存在两套编程接口，分别是旧 API (mapred) 和新 API (mapreduce)^②，考虑到旧 API 目前仍然是主流的编程接口，所以 MRv2 仅兼容旧 API 编写的 MapReduce 应用程序。但需要注意，当运行 Hadoop 自带的示例程序时，由于 YARN 自动将 `hadoop-mapreduce-examples-2.x.x.jar` 放到了环境变量中，默认情况下仍是运行该包中的程序。为了能够尝试运行 MRv1 中的程序，你需要将 `hadoop-mapreduce-examples-2.x.x.jar` 从环境变量中移除或者将环境变量 `HADOOP_USER_CLASSPATH_FIRST` 设为 `true`，并设置 `HADOOP_CLASSPATH=...:hadoop-examples-1.x.x.jar`。

(2) 编程接口兼容性

在编程接口方面，MRv2 能够完全兼容 MRv1 的旧 API，但由于新 API 的个别函数的参数列表和返回值经过了调整，因此不能够完全兼容 MRv1 的新 API，但用户很容易通过修改个别编程接口的使用方法将程序编译成功。表 8-6 给出了 MRv2 中对编程接口的修改。

① MapReduce 应用程序编程接口的讨论见 <https://issues.apache.org/jira/browse/MAPREDUCE-5108>。

② MapReduce 新旧 API 介绍可参考《Hadoop 技术内幕：深入解析 MapReduce 架构设计与实现原理》一书中的第 3 章。

表 8-6 MRv2 与 MR1 兼容性比较

编程接口 / 常量名称	不兼容问题
org.apache.hadoop.util.ProgramDriver#drive	返回值由 void 变为 int
org.apache.hadoop.mapred.jobcontrol.Job#getMapredJobID	返回值由 String 变为 JobID
org.apache.hadoop.mapred.TaskReport#getTaskId	返回值由 String 变为 TaskID
org.apache.hadoop.mapred.ClusterStatus# UNINITIALIZED_MEMORY_VALUE	数据类型由 long 变为 int
org.apache.hadoop.mapreduce.filecache.DistributedCache# getArchiveTimestamps	返回值由 long[] 变为 String[]
org.apache.hadoop.mapreduce.filecache.DistributedCache# getFileTimestamps	返回值由 long[] 变为 String[]
org.apache.hadoop.mapreduce.Job#failTask	返回值由 void 变为 boolean
org.apache.hadoop.mapreduce.Job#killTask	返回值由 void 变为 boolean
org.apache.hadoop.mapreduce.Job# getTaskCompletionEvents	返回值由 o.a.h.mapred.TaskCompletionEvent[] 变为 o.a.h.mapreduce.TaskCompletionEvent[]

(3) 不再支持的工具

由于 JobTracker/TaskTracker 服务在 MRv2 中已不再存在, 因此类似于 mradmin 这样的作用在 JobTracker 上的命令行工具也不再需要, 而由于 YARN 自身管理的需要, 一个类似于 mradmin 的命令——rmadmin 出现了, 它的功能与 mradmin 非常类似, 但直接作用于 ResourceManager 服务上。

由于 MRv2 支持旧 API 编写的 MapReduce 应用程序, 因此上层应用, 比如 Hive 和 Pig, 仍能够直接运行在 MRv2 上, 但由于每个应用程序需首先启动一个 Application-Master, 这使得同一 SQL 在 MRv2 中运行延时要大于 MRv1 环境下运行延时, 这通常有两种优化机制:

- 将 ApplicationMaster 做成一个服务, 供所有 Hive 和 Pig 作业重用;
 - 引入新的数据处理引擎, 使得每个 SQL 转化后的作业共用一个 ApplicationMaster。
- 以上两点均在设计中, 具体将在第 9 章介绍。

8.12 源代码阅读引导

在 YARN 中, MapReduce 相关代码目录组织结构如图 8-14 所示。

每个目录存放的代码及代码功能如下:

- hadoop-mapreduce-client-app: 存放了 MRAppMaster 核心实现代码, 本章介绍的绝大部分内容相关的代码均在该目录下。
- hadoop-mapreduce-client-common: MRAppMaster 用到的公共库, 包括 application ID 与 job ID 之间的映射、Protocol Buffers 消息的封装等。
- hadoop-mapreduce-client-core: 编程模型与编程接口的定义, 数据处理引擎 MapTask 和 ReduceTask 实现等, 该目录下存放的代码实际上是 MRv1 经 JobTracker

和 TaskTracker 去除后的代码，有兴趣的读者可阅读书籍《Hadoop 技术内幕：深入解析 MapReduce 架构设计与实现原理》了解实现细节。

□ `hadoop-mapreduce-client-hs`：实现了一个历史作业信息查看服务，客户端可通过该服务查看作业产生的 JobHistory 日志信息，具体已经在 8.10 节介绍了。

□ `hadoop-mapreduce-client-hs-plugins`：存放了能够根据应用程序 ID 生成应用程序 tracking URL 的插件，目前实现了一个专为 MapReduce 生成 tracking URL 的插件 `MapReduceTrackingUriPlugin`。

□ `hadoop-mapreduce-client-jobclient`：存放客户端实现代码，具体已在 8.2 节进行了介绍。

□ `hadoop-mapreduce-client-shuffle`：Shuffle 阶段用到的基于 Netty 实现的 HTTP Server 实现，该 Server 将作为附属服务运行在各个 NodeManager 上。

下面重点介绍目录 `hadoop-mapreduce-client-app` 中涉及的几个主要 Java 包，具体如图 8-14 所示。

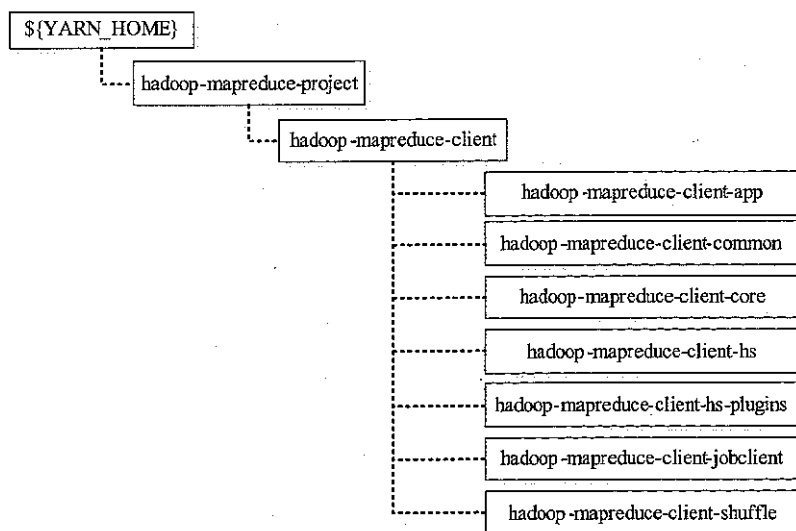


图 8-14 目录 `hadoop-mapreduce-client-app` 涉及的 Java 包

□ `org.apache.hadoop.mapreduce.jobhistory`：生成和处理 JobHistory 日志相关的实现，具体已经在 8.10 节进行了介绍。

□ `org.apache.hadoop.mapreduce.v2.app`：MRAppMaster 实现，具体已经在 8.1 节进行了介绍。

□ `org.apache.hadoop.mapreduce.v2.app.client`：客户端实现，具体已经在 8.2 节进行了介绍。

□ `org.apache.hadoop.mapreduce.v2.app.job`：Job、Task 和 TaskAttempt 的接口定义。

□ `org.apache.hadoop.mapreduce.v2.app.job.event`：Job、Task 和 TaskAttempt 状态

机涉及的事件类型定义，具体已经在 8.4 节进行了介绍。

- `org.apache.hadoop.mapreduce.v2.app.job.impl`: `Job`、`Task` 和 `TaskAttempt` 的实现，具体已经在 8.4 节进行了介绍。
- `org.apache.hadoop.mapreduce.v2.app.launcher`: `ContainerLauncher` 的定义与实现，具体已经在 8.6 节进行了介绍。
- `org.apache.hadoop.mapreduce.v2.app.local`: `local` 模式实现，本章未介绍，有兴趣的读者可自行了解。
- `org.apache.hadoop.mapreduce.v2.app.rm`: `ContainerAllocator` 的接口定义与实现，具体已经在 8.5 节进行了介绍。
- `org.apache.hadoop.mapreduce.v2.app.speculate`: 推测执行机制的实现，具体已经在 8.7 节进行了介绍。
- `org.apache.hadoop.mapreduce.v2.app.webapp`: 与 Web 界面信息展示相关的实现。

8.13 小结

MRv1 主要由编程模型、资源管理与作业控制模块、数据处理引擎三部分组成。而 YARN 出现后，资源管理模块则交由 YARN 实现，这样为了让 MapReduce 框架运行在 YARN 上，仅需实现一个 `ApplicationMaster` 组件完成作业控制模块功能，其他部分，包括编程模型和数据处理引擎，可直接采用 MRv1 的实现。本章详细介绍了 MapReduce On YARN 的基本架构、模块组成以及各模块实现。

8.14 问题讨论

问题 1: YARN 和 MRv1 都需要配置 `mapred-site.xml`，两者有什么不用，是否兼容？

问题 2: 利用 8.9 节介绍的 MapReduce 排序插件化特点，编写一个排序插件实现 Limit-N 算法，当用户提交作业时可指定一个参数启用该插件（默认排序算法不变）。

问题 3: 同问题 2，编写一个排序插件实现基于 Hash 的聚集算法，当用户提交作业时指定一个参数启用该插件（默认排序算法不变）。

问题 4: MRv1 和 MRv2（运行在 YARN 上的 MapReduce）在基本架构和编程模型方面有什么异同？

第9章 DAG 计算框架 Tez

在前面，我们介绍了运行在 YARN 上的 MapReduce 计算模型，该模型将计算过程抽象成 Map 和 Reduce 两个阶段，并通过 Shuffle 机制将两个阶段连接起来。但在一些应用场景中，为了套用 MapReduce 模型解决问题，不得不将问题分解成若干个有依赖关系的子问题，每个子问题对应一个 MapReduce 作业，最终所有这些作业形成一个有向图（Directed Acyclic Graph, DAG）。在该 DAG 中，由于每个节点是一个 MapReduce 作业，因此它们均会从 HDFS 上读一次数据和写一次数据（默认写三份），即使中间节点的产生数据仅是临时数据。很显然，这种表达作业依赖关系的方式低效的问题，进而会产生大量不必要的磁盘和网络 IO 的浪费。

为了更高效地运行存在依赖关系的作业（比如 Pig 和 Hive 产生的 MapReduce 作业），减少磁盘和网络 IO 的浪费，Hortonworks 开发了 DAG 计算框架 Tez^①。Tez 是从 MapReduce 计算框架演化而来的通用 DAG 计算框架，可作为 MapReduceR/Pig/Hive 等系统的底层数据处理引擎，它天生融入 Hadoop 2.0 中的资源管理平台 YARN，且由 Hadoop 2.0 核心人员精心打造，这势必会使其成为计算框架中的后起之秀。本章将重点介绍 Tez 的编程模型和设计架构^②。

9.1 背景

在实际大数据处理场景中，很多问题需转化成 DAG 模型解决，典型的有两类：

（1）用户编写的应用程序

很多场景下，用户编写的多个 MapReduce 应用程序之间存在依赖关系或者为了使用 MapReduce 解决一个问题，不得不将问题转化成一系列存在依赖关系的 MapReduce 作业，而为了表达这些作业的依赖关系，用户通常借助于像 Oozie^③或者 Cascading^④这样的流式作业管理工具。

下面举例说明。在搜索引擎领域中，常常需要统计最近最热门的 K 个查询词，这就是典型的“Top K”问题，也就是从海量查询中统计出现频率最高的前 K 个。如果采用 MapReduce 模型解决该问题，则可分解成两个 MapReduce 作业，分别完成统计词频和找出词频最高的前 K 个查询词的功能，如图 9-1 所示。这两个作业存在依赖关系，第二个作业

① 参见网址 <http://tez.incubator.apache.org/>。

② 本章核心内容已经发表在《程序员》2003 年 8 月刊云计算版块，标题为《Tez：运行在 YARN 上的 DAG 计算框架》。

③ 参见网址 <http://oozie.apache.org/>。

④ 参见网址 <http://www.cascading.org/>。

需要依赖前一个作业的输出结果。第一个作业是典型的 WordCount 问题。对于第二个作业，首先 Map 函数输出前 K 个频率最高的词，然后 Reduce 函数汇总 Map 任务的计算结果，并输出频率最高的前 K 个查询词。

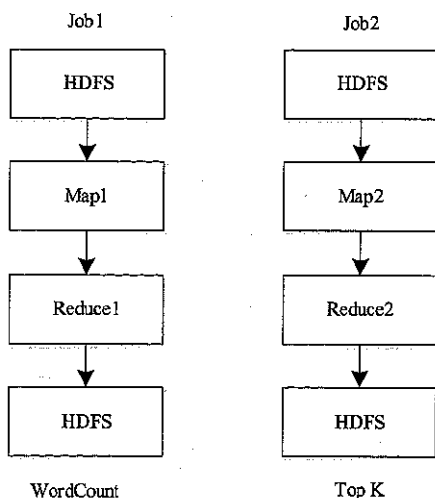


图 9-1 使用 MR 解决“Top K”问题

为了采用 MapReduce 计算模型解决“Top K”问题，我们不得不将整个计算过程分解成两个 MapReduce 作业，而是实际上，如果有一种支持 MAP → REDUCE → REDUCE 的计算框架，则会更加灵活高效。

使用 Tez 解决“Top K”问题的工作流程如图 9-2 所示。

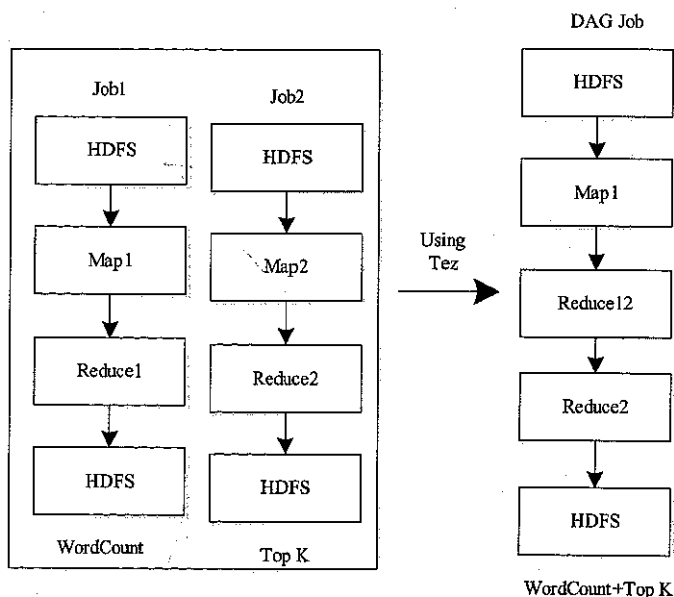


图 9-2 使用 Tez 解决“Top K”问题

(2) 类似 Pig 和 Hive 的系统

Pig 和 Hive 是构建在 MapReduce 之上的系统，它们允许用户采用更易于编写的结构化语言或者脚本语言进行大数据处理，而用户编写的结构化语言或者脚本语言往往会转化成多个存在依赖关系的 MapReduce 作业，这种运行方式非常的低效。

下面举例说明。以下 Hive 语句将被转化成了四个有依赖关系的 MR 作业，它们的运行过程如图 9-3 所示。

```
SELECT a.state, COUNT(*), AVERAGE(c.price)
FROM a
JOIN b ON(a.id = b.id)
JOIN c ON(a.itemId = c.itemId)
GROUP BY a.state
```

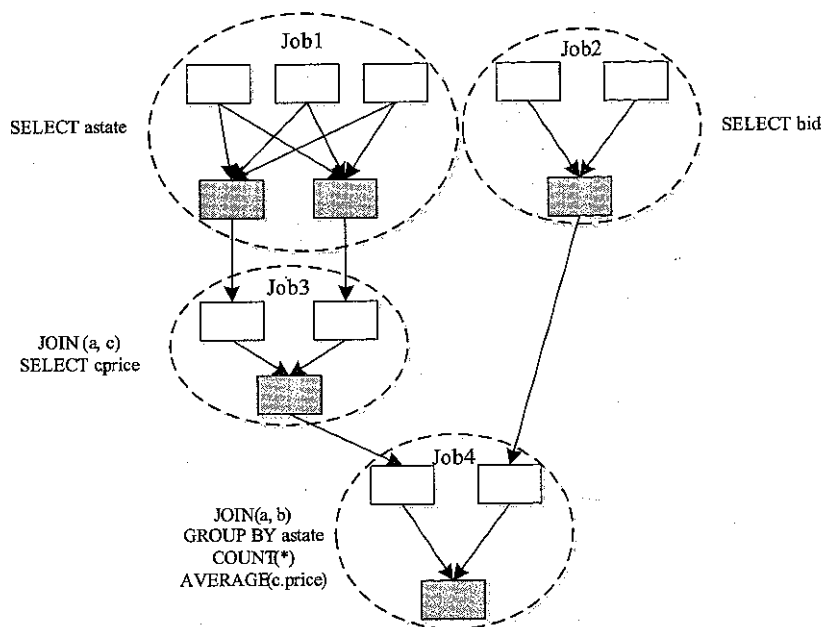


图 9-3 一个 Hive 作业的 DAG 图

以上运行 DAG 作业的方式是低效的，根本原因是作业之间的数据不是直接流动的，而是借助 HDFS 作为共享数据存储系统，即一个作业将处理后产生的数据写入 HDFS，另一个依赖于该作业的作业需再从 HDFS 上重新读取数据进行处理。很明显，更高效的方式是第一个作业直接将产生的数据传输给依赖它的作业，因为这样可大大减少磁盘和网络 IO 使用。

为了帮助用户开发更有效的运行存在依赖关系的应用程序，Hortonworks 开源了 DAG 计算框架 Tez。它直接源于 MapReduce 框架，核心思想是将 Map 和 Reduce 两个操作进一步拆分，即 Map 被拆分成 Input、Processor、Sort、Merge 和 Output，Reduce 被拆分成

Input、Shuffle、Sort、Merge、Processor 和 Output 等，这样，这些分解后的元操作可以灵活组合，产生新的操作。这些操作经过一些控制程序组装后，可形成一个大的 DAG 作业。总结起来，Tez 有以下特点：

- 运行在 YARN 之上，充分利用 YARN 的资源管理和容错等功能；
- 提供了丰富的数据流（dataflow）API；
- 扩展性良好的“Input-Processor-Output”运行时模型；
- 动态生成物理数据流关系。

接下来重点介绍一下第三个特点，Apache 当前有顶级项目 Oozie 用于 DAG 作业设计，但 Oozie 是比较高层（作业层面）的，它只是提供了一种多类型作业（比如 MR 程序、Hive、Pig 等）依赖关系表达方式，并按照这种依赖关系提交这些作业。而 Tez 则不同，它在更底层提供了 DAG 编程接口，用户编写程序时直接采用这些接口进行程序设计，这种更底层的编程方式会带来更高的效率。这个例子用 Tez 解决后，工作流程如 9-4 所示。

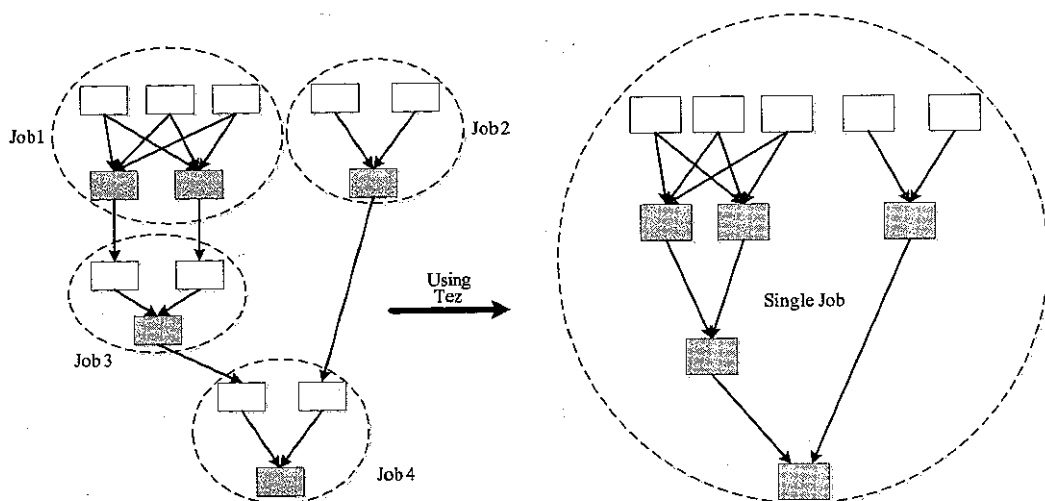


图 9-4 使用 Tez 优化 Hive DAG

通过上面的例子可以看出，Tez 可以将多个有依赖的作业转换为一个作业（这样只需写一次 HDFS，且中间节点较少），从而大大提升了 DAG 作业的性能。

9.2 Tez 数据处理引擎

9.2.1 Tez 编程模型

Tez 提供了 6 种应用程序执行接口，均位于 `org.apache.tez.engine.api.*` 中，分别是：

- Input：对输入数据源的抽象，类似于 MapReduce 模型中的 InputFormat，它解析输入数据格式，并输出一个个 key/value。
- Output：对输出数据源的抽象，类似于 MapReduce 模型中的 OutputFormat，它按照

一定的格式将用户程序产生的 key/value 写入文件系统。

- Partitioner: 对数据进行分片, 类似于 MapReduce 模型中的 Partitioner。
- Processor: 对计算单元的抽象, 它从 Input 中获取数据, 经过用户定义的计算逻辑处理后, 通过 Output 输出到文件系统中。
- Task: 对任务的抽象, 每个 Task 由 Input、Output 和 Processor 三个组件构成。
- Master: 管理各个 Task 的依赖关系, 并按照依赖关系执行它们。

9.2.2 Tez 数据处理引擎

Tez 数据处理引擎是以上 6 种可编程组件的驱动器, 它实现了一些常见的算法、常见的组件, 以提高用户编程效率。Tez 数据处理引擎的基础是两种组件——Sort (排序) 和 Shuffle (混洗), 这两个组件直接来自 MRv1 中的数据处理引擎 MapTask 和 ReduceTask。

(1) Sort 组件实现

Tez 数据处理引擎提供了通用的排序实现, 用户只需调用 `DefaultSorter#write(Object key, Object value)` 函数将所有待排序 key/value 写入排序器, 此外, Tez 允许用户定义一个 Partitioner 对所有 key/value 进行分组, 这样, 排序器会以组为单位组织数据, 不同组的内部数据按照 key 排序。排序过程如图 9-5 所示, 可分为以下几个步骤:

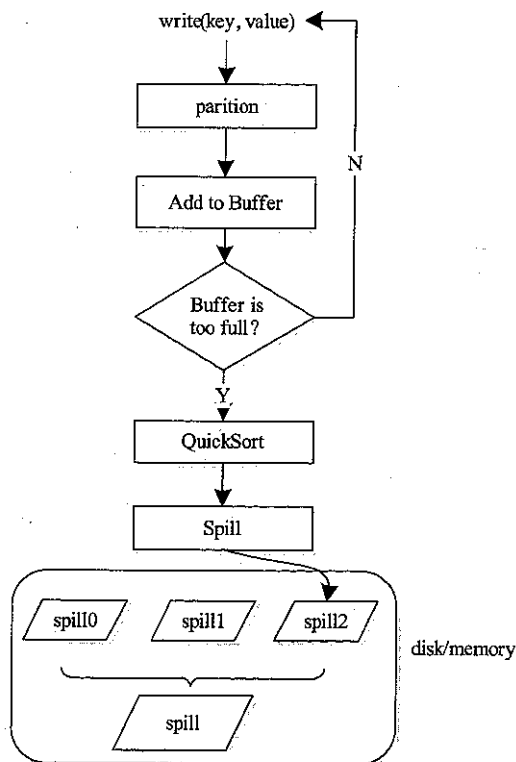


图 9-5 Sort 组件实现

步骤 1 用户将数据写入排序器后, 排序器调用 `partitioner` 组件确定数据所在分组号;

步骤 2 排序器将分组号、`key` 和 `value` 值写入一个环形缓冲区;

步骤 3 一旦该缓冲区使用率达到一定阈值, 则对缓冲区数据进行排序 (默认使用快速排序), 排序方法是先按照分组号排序, 再按照 `key` 排序, 这样相同分组的数据被聚集在一起, 且分组内部数据有序;

步骤 4 将排好序的有序数据被写入文件中;

步骤 5 重复执行步骤 1~4, 直到所有数据处理完成, 此时会生成多个有序文件, 而排序器最终会将所有文件合并成一个大的有序文件。

在 MRv1 中, Map Task 和 Reduce Task 均采用了外部排序, 尤其是 Map Task 输出的中间结果经排序后必须写到本地磁盘上, 用户无法控制。而在 Tez 中, 用户可选择将排序结果写入本地磁盘或者内存中, 当然, 如果将数据写入内存中时, 需确保有足够的内存可用。

(2) Shuffle 组件的实现

Shuffle 组件的整体计算流程如图 9-6 所示, 共分为 3 个阶段, 分别是:

- **Shuffle 阶段**。又称 Copy 阶段, Reduce Task 从各个 Map Task 上远程复制一片数据, 并针对某一片数据, 如果其大小超过一定阈值, 则写到磁盘上, 否则直接放到内存中。
- **Merge 阶段**。在远程复制数据的同时, Reduce Task 启动了两个后台线程对内存和磁盘上的文件进行合并, 以防止内存使用过多或磁盘上文件过多。
- **Sort 阶段**。按照 MapReduce 语义, 用户编写的 `reduce()` 函数输入的是按 `key` 进行聚集的一组数据。为了将 `key` 相同的数据聚在一起, Hadoop 采用了基于排序的策略, 由于各个 Map Task 已经实现对自己的处理结果进行了局部排序, 因此 Reduce Task 只需对所有数据进行一次归并排序即可。

在 Shuffle 中, Shuffle 阶段和 Merge 阶段是并行进行的, 当远程复制数据量达到一定阈值后, 便会触发相应的合并线程对数据进行合并。

如图 9-6 所示, 总体上看, Shuffle&Merge 阶段可进一步划分为三个子阶段:

- **准备运行完成的任务列表**。EventFetcher 线程周期性通过 RPC 从 Master 上获取已完成任务列表, 并保存到映射表 `mapLocations` (保存了节点 `host` 与已完成任务列表的映射关系) 中。
- **远程复制数据**。Shuffle 组件同时启动多个 Fetcher 线程, 这些线程从 `scheduled-Copies` 列表中获取待获取任务的数据输出位置, 并通过 HTTP GET 远程复制数据。为防止出现网络热点, Shuffle 组件会为每个 Fetcher 随机分配一个待复制任务输出数据位置。对于复制的数据分片, 如果大小超过一定阈值, 则存放到磁盘上, 否则直接放到内存中。
- **合并内存文件和磁盘文件**。为了防止内存或者磁盘上文件数据过多, Shuffle 组件启动了 `InMemoryMerger` 和 `OnDiskMerger` 两个线程分别对内存和磁盘上文件进行合并。

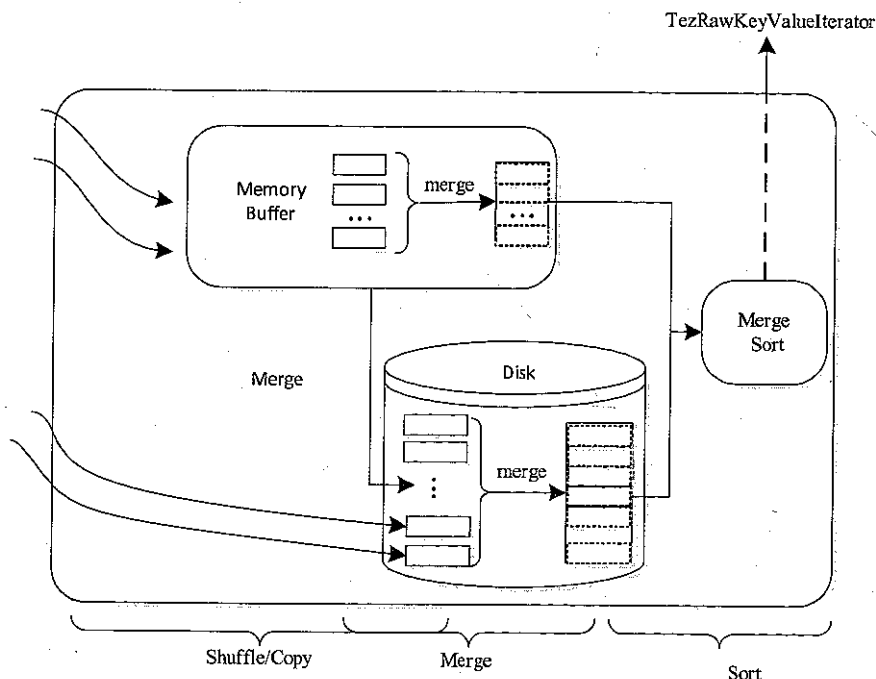


图 9-6 Shuffle 组件实现

(3) 编程组件实现

为了用户使用方便, Tez 数据处理引擎还提供了多种 Input、Output、Task 和 Sort 的实现, 具体如下:

- **Input 实现:** LocalMergedInput (文件本地合并后作为输入)、ShuffledMergedInput (远程复制数据且合并后作为输入);
- **Output 实现:** InMemorySortedOutput (内存排序后输出)、LocalOnFileSortedOutput (本地磁盘排序后输出)、OnFileSortedOutput (磁盘排序后输出);
- **Task 实现:** RunTimeTask (对 Input、Output 和 Processor 的封装);
- **Sort 实现:** DefaultSorter (对数据进行外部排序)、InMemoryShuffleSorter (远程复制数据并对数据进行内存排序)。

9.3 DAG Master 实现

9.3.1 DAG 编程模型

Tez 通过有向图模型组织用户编写的各类 Task, 这些 Task 按照依赖关系可形成一个 DAG, 具体如图 9-7 所示, 一个 DAG 由若干个顶点 (Vertex) 和连接这些顶点的边 (Edge) 组成, 每个顶点对应一段作用在一个数据集上的处理逻辑, 可同时启动多个任务,

而边则定义了两个顶点之间的数据通信方式。如图 9-7 所示，MapReduce 作业是一个非常简单的 DAG，Map 和 Reduce 相当于两个顶点，它们均可以同时启动多个任务处理数据，而 Map 和 Reduce 之间的 Shuffle 通信模型相当于连接两个顶点的边。

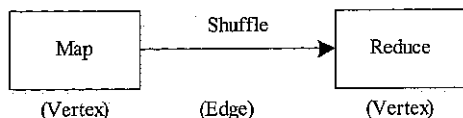


图 9-7 MR DAG 图

DAG 编程接口位于 `org.apache.tez.dag.api.*` 中，Vertex 定义如下：

```
public class Vertex {
    private final String vertexName; // 顶点名称
    private final String processorName; // 该顶点的 Processor 名称

    private final int parallelism; // 该顶点的并发度（同时运行任务数目）
    private VertexLocationHint taskLocationsHint; // 本地性信息
    private Resource taskResource; // 每个任务资源量
    private Map<String, LocalResource> taskLocalResources; // 本地资源
    private Map<String, String> taskEnvironment; // 任务运行所需环境变量

    // 以下是入顶点、出顶点、入边和出边集合
    private final List<Vertex> inputVertices = new ArrayList<Vertex>();
    private final List<Vertex> outputVertices = new ArrayList<Vertex>();
    private final List<String> inputEdgeIds = new ArrayList<String>();
    private final List<String> outputEdgeIds = new ArrayList<String>();
    ...
}
```

Edge 定义如下：

```
public class Edge {
    private final Vertex inputVertex; // 入顶点
    private final Vertex outputVertex; // 出顶点
    private final EdgeProperty edgeProperty; // 边属性
    ...
}

public class EdgeProperty {
    ...
    ConnectionPattern connectionPattern;
    SourceType sourceType;
    String inputClass;
    String outputClass;
    ...
}
```

Tez 中的 DAG 表示方式采用了典型的图数据结构，具体如图 9-8 所示。

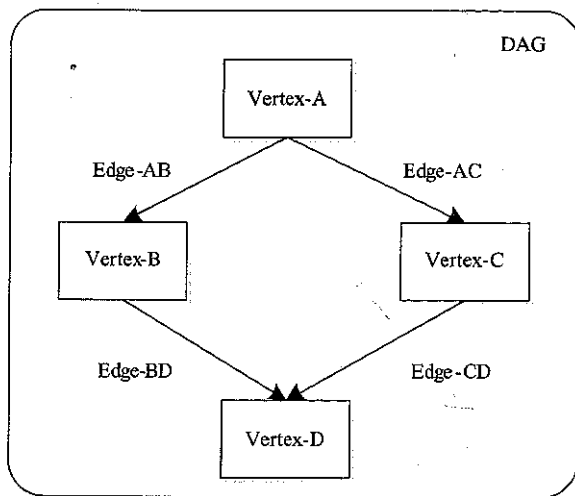


图 9-8 Tez 中 DAG 表示方式

同 MRv1 一样，一个作业是由描述作业的配置文件和运行作业所需的文件资源（JAR 包、可执行程序等）组成的，在 Tez 中，作业描述信息由类 DAGConfiguration（继承自 MRv1 中 org.apache.hadoop.conf.Configuration）保存，利用 Tez 编写应用程序实际上是构建一个 DAGConfiguration 的过程。DAGConfiguration 提供了很多 setXXX 方法：

```

public void setEdgeProperties(List<Edge> edges)
public void setNumVertexTasks(String vertexName, int numTasks)
public void setVertexTaskMemory(String vertexName, int memory)
public void setVertexTaskCores(String vertexName, int cores)
public void setVertices(List<Vertex> vertices)
public void setInputVertices(String vertexName, List<Vertex> inputVertices)
public void setOutputVertices(String vertexName, List<Vertex> outputVertices)
public void setOutputEdgeIds(String vertexName, List<String> edgeIds)

```

9.3.2 MR 到 DAG 转换

由于 Tez 是从 MRv2 演化而来的，故它重用了 MRv2 中大量的代码，这使得它遗传了 MRv2 所有优点，同时可与它兼容，用户可直接将一个 MapReduce 程序运行在 Tez 上。除此之外，Tez 还提供了一个 MR 到 DAG 的转换工具，通过该工具，用户很容易将多个有依赖关系的 MapReduce 作业合并成一个 DAG 作业，这将大大减少磁盘 IO 次数，从而提高程序运行效率。

下面举例说明。假设数据集 EmployeeTable 存放了某个公司内部员工信息，为了简化问题，员工信息有两个，分别是 ID（员工 ID、用于唯一标示一个员工）和 DeptName（所在部门名称），存放格式如下：

```

00010 IT
00011 sale

```

```
00100 IT
00200 market
...
```

接下来要通过编写 MapReduce 程序的方法实现类似以下 SQL 的功能：

```
Select DeptName, COUNT(*) as cnt FROM EmployeeTable GROUP BY DeptName ORDER BY cnt;
```

为实现该功能，需编写两个 MapReduce 程序，分别如下：

- ❑ **CountEmployeeByDept**。按照部门名称 (DeptName) 统计人数，整个计算逻辑类似于经典的 MapReduce 实现实例 Wordcount，即在 Mapper 中，直接输出 <deptName, 1> 对，在 Reducer 中，按照 DeptName 累加人数。
- ❑ **OrderEmployeeByCount**。按照部门人数排序，实际上是利用了 MapReduce 的内嵌的分布式排序机制，为保证数据全局有序，需将该作业的 Reduce Task 个数设置为 1。该作业的实现如下：在 Mapper 中，颠倒 DeptName 与 COUNT (部门人数)，即输出 <COUNT, DeptName>，在 Reducer 中，再次颠倒 DeptName 与 COUNT，输出 <DeptName, COUNT>。

在该实现中，两个 MapReduce 作业均要先从 HDFS 上读取数据，并再次写到 HDFS 上。从整个解决方案的逻辑看，第二个作业依赖于第一个作业的输出，而第一个作业产生的结果实际上仅是中间临时结果，将这样的数据写入 HDFS 是没有必要的 (HDFS 默认写三份数据)。通常而言，为了方便表达这种作业间依赖关系，会借助于类似于 Oozie、Cascading 这样的数据流系统。

而 Tez 的引入正是为了提高存在依赖关作业的运行效率，使用 Tez 改写这两个 MapReduce 程序，只需编写一个 Tez DAG 作业 GroupByOrderEmployee，该作业的计算逻辑是 Map-Reduce-Reduce，具体如下：

- ❑ **GroupByOrderEmployee.ScanMapper**。与 CountEmployeeByDept.Mapper 一样；
- ❑ **GroupByOrderEmployee.CountReducer**。相当于 CountEmployeeByDept.Reducer 与 OrderEmployeeByCount.Mapper 的结合，它按照 DeptName 累加人数，并输出 <COUNT, DeptName> 对；
- ❑ **GroupByOrderEmployee.OrderReducer**。与 OrderEmployeeByCount.Reducer 一样。

上面 ScanMapper、CountReducer 和 OrderReducer 的编写方法与 MRv1 中编写 Mapper 和 Reducer 的方法完全一样 (两者兼容)，但在配置属性的时候要添加以下几行代码：

```
Configuration conf = new Configuration();
conf.setInt(MRJobConfig.MRR_INTERMEDIATE_STAGES, 1);

// 设置中间 Reducer
conf.setClass(MultiStageMRConfigUtil.getPropertyNameForIntermediateStage(1,
    "mapreduce.job.reduce.class"), CountReducer.class, Reducer.class);
// 设置 Reducer 输出 key 类型
conf.setClass(MultiStageMRConfigUtil.getPropertyNameForIntermediateStage(1,
    "mapreduce.map.output.key.class"), IntWritable.class, Object.class);
```

```
// 设置 Reducer 输出 value 类型
conf.setClass(MultiStageMRConfigUtil.getPropertyNameForIntermediateStage(1,
    "mapreduce.map.output.value.class"), Text.class, Object.class);
conf.setInt(MultiStageMRConfigUtil.getPropertyNameForIntermediateStage(1,
    "mapreduce.job.reduces"), 2);
// 其他设置与 MRv1 完全一样
...
```

Tez 提供的 MR 到 DAG 转换工具只适合“MAP-REDUCE+”类型的作业，也就是说，在传统的 Map 和 Reduce 两阶段之间穿插了若干个 Reduce 阶段。典型的应用如下：有 N 个 Job，即 Job1, Job2, ..., JobN，其中 JobK 的 Map 阶段为 MapK，Reduce 阶段为 ReduceK，且这些作业之间存在顺序依赖关系，即 Job(K+1) 依赖于 JobK，则通过 Tez 转换工具，可将该作业转换成一个 DAG 作业，即 Map1' (对应 Map1)、Reduce1' (Reduce1 与 Map2 组合而成)，..., ReduceK' (ReduceK 与 Map(K+1) 组合而成)，ReduceN' (对应 ReduceN)。

如果将若干个存在顺序依赖关系的 MapReduce 作业合并成一个 DAG 作业，用户所要做的是重新设计中间的几个 Reduce 阶段，即融合相邻两个作业的 Reduce 和 Map 阶段，具体如图 9-9 和 9-10 所示。

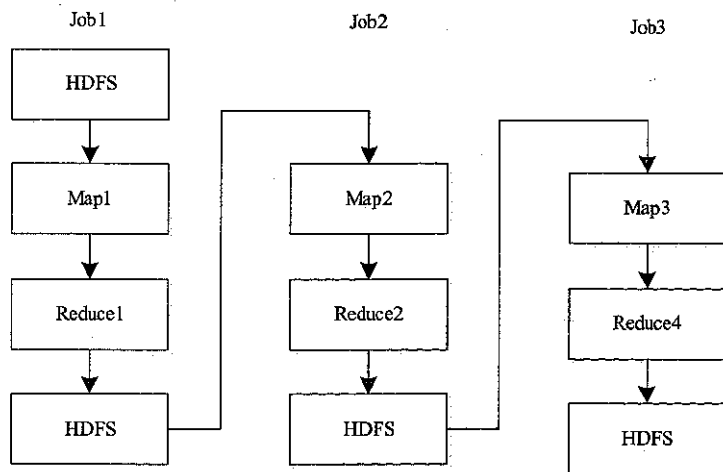


图 9-9 顺序依赖关系的 MR 作业

9.3.3 DAGAppMaster

为了能够让 Tez 运行在 YARN 之上，需开发一个 YARN 客户端和一个 ApplicationMaster，本小节重点介绍 Tez 的 YARN ApplicationMaster——DAGAppMaster。

DAGAppMaster 直接源自 MRAppMaster，重用了它的大部分实现机制和代码，总结起来，它主要完成以下几个功能：

- 数据切分和作业分解。DAGAppMaster 负责对输入数据集切分，并创建一系列任务处理这些数据。

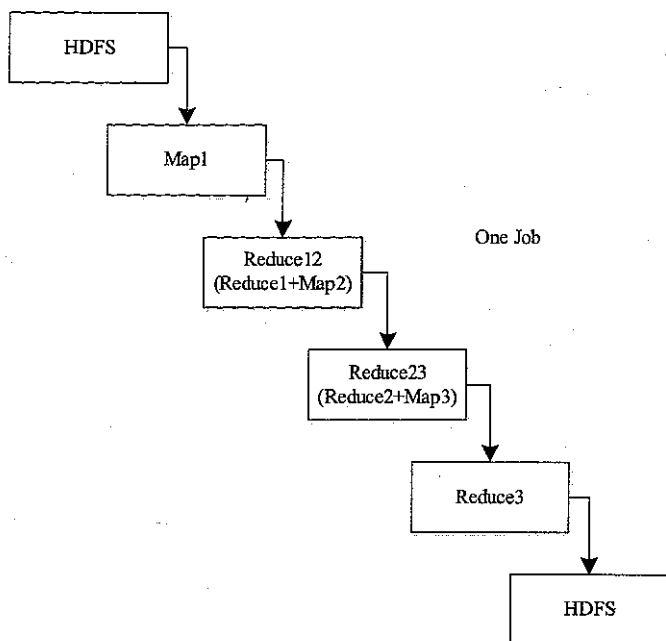


图 9-10 若干个 MR 作业合并成一个 DAG 作业

□ **任务调度。**将作业分解成一系列存在依赖关系的任务，并按照依赖关系调度和执行它们。

□ **与 ResourceManager 通信，为 DAG 作业申请资源。**

□ **与 NodeManager 通信，启动 DAG 作业中的任务。**

□ **监控 DAG 作业的运行过程，确保它快速运行结束。**当一个任务运行失败时，会重新为它申请资源并启动它；当一个任务运行较慢时，会为其启动一个备份任务。

每个 DAGAppMaster 负责管理一个 DAG 作业，前面已经讲到，一个 DAG 由若干个顶点（Vertex）和连接这些顶点的边（Edge）组成，每个顶点对应一段作用在一个数据集上的处理逻辑。每个顶点可配置一定数目的并发度，并发度实际上就是任务个数。DAGAppMaster 优先为那些不依赖任何顶点的顶点任务申请资源，一旦一个顶点中所有任务运行完成，则认为该顶点运行结束，而将该顶点从 DAG 中移除，再寻找新的不依赖任何顶点的顶点，并为它们申请资源，这样持续下去，直到所有顶点运行完成，这样整个 DAG 作业便运行完成了。

如图 9-11 所示，对于 DAG 中的一个顶点（Vertex），它由一定数目的任务（Task）组成，这些任务分别处理输入数据集中的一份数据，一旦所有任务运行完成，则意味着该顶点运行完成；对于任何一个任务，它可能对应多个运行实例 TaskAttempt，比如任务刚开始运行时，会创建一个运行实例，如果该实例运行失败，则会再启动一个实例重新运行；如果该实例运行速度过慢，则会为其再启动一个相同实例同时处理一份数据，这些机制均借鉴了 MRv1 中的设计。

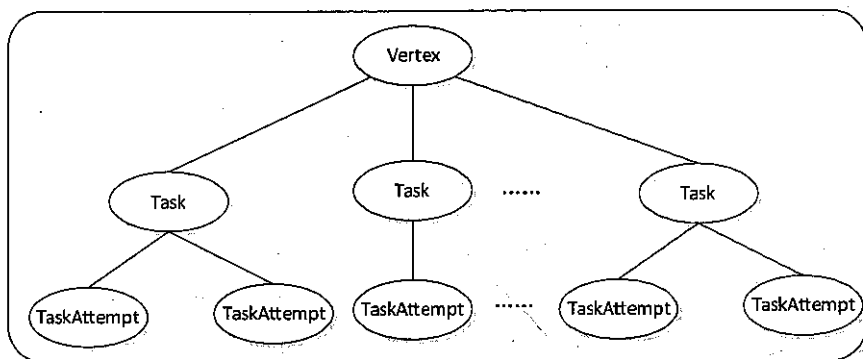


图 9-11 Tez 内部顶点运行方式

当用户向 YARN 中提交一个 DAG 应用程序后，YARN 将分两个阶段运行该应用程序：第一个阶段是启动 DAGAppMaster；第二个阶段是由 DAGAppMaster 创建应用程序，为它申请资源，并监控它的整个运行过程，直到运行成功。如图 9-12 所示，YARN 的工作流程分为以下几个步骤：

步骤 1 用户向 YARN 中提交应用程序，其中包括 DAGAppMaster 程序、启动 DAGAppMaster 的命令、用户程序等。

步骤 2 ResourceManager 为该应用程序分配第一个 Container，并与对应的 NodeManager 通信，要求它在这个 Container 中启动应用程序的 DAGAppMaster。

步骤 3 DAGAppMaster 首先向 ResourceManager 注册，这样用户可以直接通过 ResourceManager 查看应用程序的运行状态，然后它将为各个任务申请资源，并监控它的运行状态，直到运行结束，即重复步骤 4~7。

步骤 4 DAGAppMaster 采用轮询的方式通过 RPC 协议向 ResourceManager 申请和领取资源。

步骤 5 一旦 DAGAppMaster 申请到资源后，则与对应的 NodeManager 通信，要求它启动任务。

步骤 6 NodeManager 为任务设置好运行环境（包括环境变量、JAR 包、二进制程序等）后，将任务启动命令写到一个脚本中，并通过运行该脚本启动任务。

步骤 7 各个任务通过某个 RPC 协议向 DAGAppMaster 汇报自己的状态和进度，以让 DAGAppMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务。

在应用程序运行过程中，用户可随时通过 RPC 向 DAGAppMaster 查询应用程序的当前运行状态。

步骤 8 应用程序运行完成后，DAGAppMaster 向 ResourceManager 注销，并关闭自己。

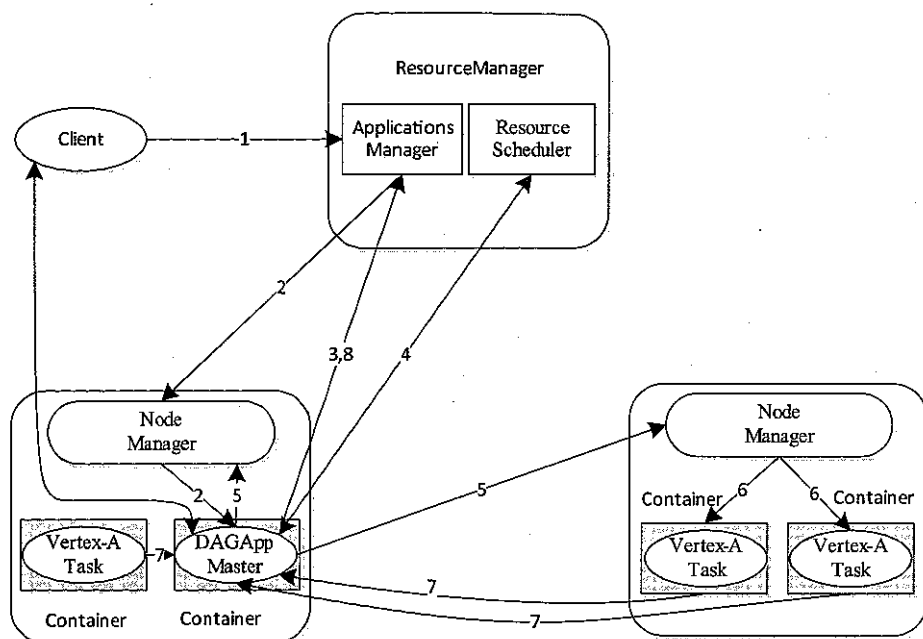


图 9-12 Apache YARN 的工作流程

9.4 优化机制

当前 YARN 框架还存在一些问题，而为了解决这些问题或减小它们带来的影响，Tez 引入了一些优化技术。本节将重点介绍这些技术。

9.4.1 当前 YARN 框架存在的问题

当前 YARN 框架存在如下一些问题。

- 每个作业启用一个 ApplicationMaster。MRv1 中所有应用程序共用一个追踪器 JobTracker，当 JobTracker 出现故障时，整个系统将不可用，且所有应用程序将运行失败。与 MRv1 不同，YARN 中每个应用程序启用一个独立的应用程序追踪器 ApplicationMaster，解决了 MRv1 中单点故障和扩展瓶颈问题。但这种方式将引入一个新的问题：应用程序延迟较大，即每个应用程序首先要申请资源启动一个 ApplicationMaster 后，才可以启动任务，也就是说，与 MRv1 中的应用程序运行过程相比，YARN 应用程序将耗费更多的计算资源和产生更长的运行延迟，这不利于运行小作业和 DAG 作业。尤其是 DAG 作业（如 Hive SQL 和 Pig 产生的 DAG 作业），将需要更多的计算资源。
- 资源无法重用。在 MRv1 中，用户可为自己的作业设置是否启用 JVM 重用功能。如果启用该功能，则同一个 JVM 可运行多个任务，从而降低作业延迟提高作业效率。

在 YARN MRAppMaster 中, MRAppMaster 总是为未运行的任务申请新的资源, 也就是说, 任务运行完成后便会释放对应的资源, 并为接下来运行的任务重新申请资源, 而不会向 MRv1 那样重用资源 (JVM)。

9.4.2 Tez 引入的优化技术

为了克服当前 YARN 存在的问题, Apache Tez 提出了一系列优化技术, 其中值得一提的是 ApplicationMaster 缓冲池、预先启动 Container、Container 重用三项优化技术。

- ApplicationMaster 缓冲池: 在 Apache Tez 中, 用户并不是直接将作业提交到 ResourceManager 上, 而是提交到一个称为 AMPoolServer 的服务上。该服务启动后, 会预启动若干个 ApplicationMaster, 形成一个 ApplicationMaster 缓冲池, 这样, 当用户提交作业时, 直接将作业提交到某个已经启动的 ApplicationMaster 上即可。这样做的好处是, 避免了每个作业用时动态启动一个独立的 ApplicationMaster。在 Apache Tez 中, 管理员可采用参数 `tez.ampool.am-pool-size` 和 `tez.ampool.max.am-pool-size` 配置 ApplicationMaster 缓冲区中最小 ApplicationMaster 个数和最大 ApplicationMaster 个数。
- 预先启动 Container: ApplicationMaster 缓冲池中的每个 ApplicationMaster 启动时可以预先启动若干个 Container, 以提高作业运行效率。
- Container 重用: 一个任务运行完成后, ApplicationMaster 不会马上注销它使用的 Container, 而是将它重新分配给其他未运行的任务, 从而达到资源重用的目的。

9.5 Tez 应用场景

本节介绍几种 Tez 的典型应用场景。

(1) 直接编写应用程序

同使用 MapReduce 计算框架编程应用程序一样, 用户也可以直接使用 Tez 编写 DAG 类型的应用程序。此外, 前面也曾讲到, Tez 还提供了一个 MR 到 DAG 的转换工具, 通过使用该工具, 用户很容易将多个有依赖关系的 MapReduce 作业合并成一个 DAG 作业, 这将大大减少磁盘 IO 次数, 从而提高程序运行效率。

(2) 优化 Pig、Hive 等引擎

Hortonworks 正在尝试将 Tez 应用到 Hive 引擎中^① (下一代 Hive 引擎被称为 Stinger), 从而依靠 Tez 数据处理引擎的更灵活的表达方式对 Hive 带来性能的提升。Hortonworks 官方博客的测试结果表明, Stinger 性能比 Hive 提升几十倍, 从内部实现看, 导致性能提升的因素很多, 包括高效的数据存储格式、更加智能的查询计划优化器等, 但从 Tez 引擎角度看, 它为 Stinger 带来的好处如下:

- 避免查询语句转换成过多的 MapReduce 作业后产生大量不必要的网络和磁盘 I/O。

① 参考 <http://hortonworks.com/blog/apache-hive-0-11-stinger-phase-1-delivered/>。

当一个查询语句中存在多个 join、sort 或者 group by 操作时, Hive 会将该语句转化成多个存在依赖关系的 MapReduce 作业, 且每个作业均需将输出结果写到 HDFS 上, 再由下一个作业读取作进一步处理。而引入 Tez 后, 整个查询语句将被转化成一个 MapReduce 作业, 整个计算过程只需写一次 HDFS, 从而大大减少磁盘和网络 I/O 的浪费。下面给出一个 Hive SQL 示例予以说明:

```
SELECT a.state, COUNT(*)
FROM a JOIN b ON(a.id = b.id)
GROUP BY a.state
```

该 SQL 包含一个两表连接操作和一个分组操作, 因此, Hive 会将它转化成两个 MapReduce 作业: 第一个作业进行两表连接, 并将结果写到 HDFS 上; 第二个作业从 HDFS 上读取第一个作业的输出结果, 进行分组后, 将数据再次写到 HDFS 上。而使用 Tez 后, 该 SQL 只被转化成一个 DAG 作业, 可减少中间写 HDFS 和读 HDFS 的 I/O 开销, 具体如图 9-13 所示。

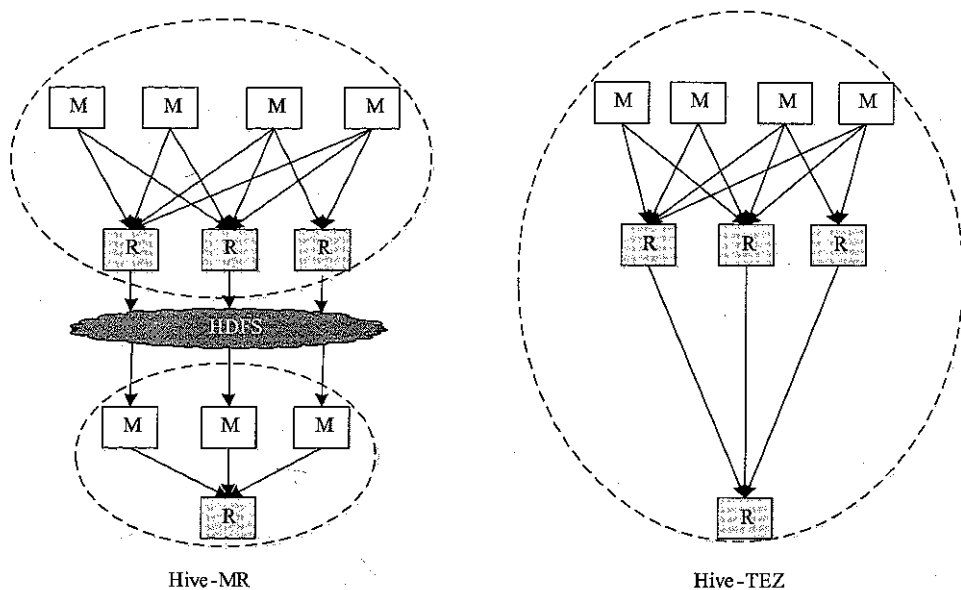


图 9-13 Hive 使用 MapReduce 和 Tez 作为数据处理引擎效果对比

□ **更加智能的任务处理引擎。**在 MapReduce 模型中, 为了提高容错性, Map Task 产生的结果必须写一次磁盘, 而 Tez 是可选的, 可根据任务产生的数据量决定将其存放在内存中还是写磁盘, 这可大大减少中间临时数据的读写 I/O 次数, 能显著提高实时任务和交互式任务的执行效率, 具体如图 9-14 所示。

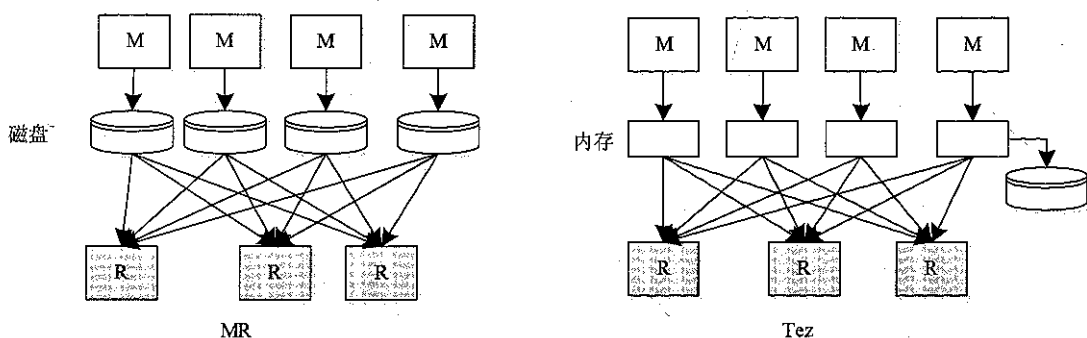


图 9-14 MapReduce 和 Tez 对中间临时数据存储对比

9.6 与其他系统比较

为了帮助读者更好地理解 Tez，本节我们再将与其他一些系统进行比较。

（1）与类 Oozie 系统比较

Oozie 是一个工作流调度系统（其他类似系统还有 Cascading、Azkaban 等），它能够按照用户事先定义好的作业依赖关系调度作业，并提供了灵活的作业级别的依赖关系表达方式和容错机制。但与 Tez 这种 DAG 执行引擎不同，Oozie 只是一种作业依赖关系表达和调度框架，它在逻辑上并没有将有依赖关系的作业合并成一个作业来优化 I/O 读写，换句话说，Oozie 只是为了方便用户组织和表达存在依赖关系的 MapReduce 作业而设计的，并不能用于优化 DAG 作业的执行效率。

（2）与 MapReduce 比较

MapReduce 只是一种简单的数据处理模型，它将数据处理过程简化为 Map 和 Reduce 两个阶段，这限制了 MapReduce 的计算表达能力。而 Tez 不同，它可以包含任意多个数据处理阶段，并提供了一种更加灵活高效的编程模型，可以应用于 MapReduce 难以高效表达的计算场景，典型的场景是数据挖掘和自然语言处理，这两种场景下经常用到的 DAG 计算。Tez 可作为 MapReduce 之下的数据处理引擎，即用户仍采用现有的 MapReduce API 编写程序，但使用 Tez 将之组装成一个 DAG 作业。

（3）与 Spark 比较

Tez 和 Spark 均为了克服 MapReduce 在内存计算、DAG 计算方面性能低下而提出的，它们存在非常多的共性，主要如下：

- 提供了多种算子（比如 Map、Shuffle 等）供用户使用；
- 通过减少不必要的磁盘和网络 I/O 浪费提高 DAG 作业的性能；
- 可运行在 YARN 上；
- 能够充分利用内存提高小作业执行效率。

当然，它们也存在很多不同，主要有：

- **数据集重用。** Spark 引入了“弹性分布式数据集”(RDD)这一数据表示模型,这使得一些频繁访问的数据集可在不同任务间重用和分享,这一点 Tez 暂时无法做到。
- **支持的算子种类。** Spark 提供的算子种类要远远多于 Tez,比如在数据传输模型方面,Spark 支持多对多(Shuffle)、一对多和多对一等算子,而 Tez 目前仅支持多对多(Shuffle)一种,但 Tez 正在尝试丰富它的算子库,相信在不久的将来,两者在这方面就不会存在明显差距了。
- **与 MapReduce 兼容性。** Tez 是直接从 Hadoop MapReduce 计算框架演化而来的,不仅继承了它的良好扩展性、容错性等优点,而且与 MapReduce 编程接口完全兼容,用户可以直接将自己的 MapReduce 程序运行在 Tez 上,也可以很容易地将已有的 MapReduce DAG 作业转换成 Tez 作业。Spark 则是完全重新开发的计算模型,与现有的 MapReduce 编程接口无法兼容。
- **实践验证。** Tez 是直接在 Hadoop MapReduce 源代码基础上修改而来的,充分利用了 MapReduce 自身强大的扩展性和容错性等优点,且这些特性已经在实践中得到验证。但 Spark 的扩展性和容错性有待验证。
- **开发语言。** Spark 采用了 Scala 作为开发语言,充分利用了 Scala 的优点,这使得 Spark 代码非常精简,但新语言的引入增加了学习成本;Tez 则不同,它仍采用 Java 作为开发语言,与 Hadoop 整个生态系统一脉相承。

9.7 小结

Tez 目前还处于开发阶段,尚未发布稳定版,从源代码可看出,Tez 很多地方仍保留了 Hadoop MapReduce 原有的注释,且代码中存在大量 TODO 标识尚未实现,但从它的 jira^①系统上可以看出,它的更新速度非常快,可以预想,一旦 Tez 成熟,用户可以很容易使用 Tez 编写 DAG 应用程序,同时,经过 Tez 优化的 Hive(即 Stinger)的运算速度也会更快,加上 Hive 正在开发的 cost-based 查询优化器,Hive 将有能力在实时计算、交互式计算和离线计算等领域发挥更大更强的作用。此外,Tez 和 Spark、Stinger 和 Shark(运行在 Spark 之上的 SQL 引擎)将直接存在竞争关系,这将为用户带来更多选择。

① 参见网址 <https://issues.apache.org/jira/browse/TEZ>。

第 10 章 实时 / 内存计算框架 Storm/Spark

前面提到, Hadoop MapReduce 适合离线批处理场景, 而不擅长实时计算和近实时计算相关的应用场景。为了克服 MapReduce 这方面的不足, Twitter 开源了实时计算框架 Storm, 伯克利开发了基于内存的 MapReduce 实现 Spark, 这两种计算框架可用于实时计算和交互式计算领域, 与 Hadoop MapReduce 的离线批处理应用场景互补。

本章将介绍 Storm 和 Spark 两种计算框架的编程模型和基本架构, 并重点分析如何将这两个计算框架运行在 YARN 上。

10.1 Hadoop MapReduce 的短板

前面提到, Hadoop MapReduce 是一个离线计算框架, 具有良好的容错性、扩展性, 且易于编程, 但它的以下特点决定它仅适合离线批处理应用场景:

- 启动时间长。一个 MapReduce 作业由启动任务、Map 任务、Reduce 任务和清理任务四种组成, 其中, 启动任务和清理任务分别在作业开始和结束时运行, 分别完成作业初始化和清理工作, 这导致最简单的 MapReduce 作业 (Map 和 Reduce 任务不做任何事情) 也会运行几秒到十几秒。此外, 任务是运行在 JVM 中的, 而 JVM 总是用时才启动的, 这会消耗掉一定的启动时间。
- 调度开销大。Hadoop 调度器负责为 MapReduce 作业分配资源, 当一个作业包含的任务数目很大时, Hadoop 将花费较长时间将所有任务调度到各个计算节点上, 此外, 当资源不充足时, 作业需排队等待资源。
- 中间数据写磁盘。为了保证 MapReduce 作业的容错性, Map Task 产生的中间结果总是要写到本地磁盘上, 这给小作业带来较大延时。

随着互联网的发展, 新的需求产生了, 这些需求期望数据能够得到实时处理, 而这正是 Hadoop MapReduce 的短板。

10.2 实时计算框架 Storm

本章所涉及的实时计算, 是指被处理的数据像流水一样不断流入系统, 而系统需要针对每条数据进行实时处理和计算, 并永不停止 (直到用户显式杀死进程)。考虑到数据源中数据流动的特点, 实时计算也被称为“流式计算”。

从数据源特征角度看, 本节要介绍的 Storm 计算框架与 MapReduce 的明显不同是, Storm 的数据源是动态的, 即收到一条便处理一条, 而 MapReduce 的数据源是静态的, 即

数据被处理前整个数据集已经确定，且计算过程中不能被修改。实时计算框架能够解决很多实际应用问题，比如广告推荐、用户行为日志实时分析等。

Storm 官方网站^①上谈到，在出现 Storm 之前，用户可以通过手动维护一个由消息队列和消息处理器组成的实时处理网络进行实时计算，消息处理器从消息队列取出一个消息进行处理，更新数据库，并发送新消息给其他队列。但不幸的是，这种方式有以下几个缺点：

- **缺乏自动化。**你需要花费大量的时间用于服务部署和运维方面，将消耗掉大量的运维和维护成本。
- **缺乏健壮性。**你需要开发很多额外的辅助程序保证所有的消息处理器和消息队列正常运行，比如一个服务宕掉了，如何进行容错。
- **伸缩性差。**当系统遭遇瓶颈时，你将面对一下难题：如何对系统进行扩展和对数据分流，以消除系统瓶颈。这通常需要大量人工介入才能完成，而对于一个高速发展的系统，这无疑是不可接受的。

Storm 定义了一批实时计算的原语。如同 MapReduce 大大简化了并行批量数据处理一样，Storm 的这些原语大大简化了并行实时数据处理。Storm 的一些关键特性如下：

- **适用场景广泛。**Storm 可以用来处理消息和更新数据库（消息流处理），例如，对一个数据集进行持续的查询并返回给客户端（持续计算）；对一个耗资源的查询做实时并行化的处理（分布式远程过程调用）。总之，Storm 提供的这些基础原语可以满足大量的场景。
- **良好的伸缩性。**Storm 的可伸缩性可以让它每秒可以处理的消息量达到很高。用户通过简单地增加机器和重新设置这个任务的并行度来扩展一个实时计算任务。
- **保证无数据丢失。**对于一个良好的实时系统而言，应该提供机制保证所有的数据被成功处理。Storm 能够保证每一条消息都会被处理，这一点与另一个系统——Yahoo！开源的实时系统 S4 有明显差别^②。
- **异常健壮。**不同于 Hadoop MapReduce 的难以管理，Storm 集群非常容易管理。易于管理是 Storm 的设计目标之一。
- **良好的容错性。**如果在计算过程中出了一些错误，Storm 会重新分配这个出问题的任务。Storm 会保证一个处理逻辑永远运行（除非显式杀掉这个处理逻辑）。
- **支持多语言编程。**健壮性和伸缩性不应该局限于一个平台。Storm 的拓扑和消息处理组件可以用任意语言来定义，这一点使得任何人都可以使用 Storm。

10.2.1 Storm 编程模型

想要学习 Storm 的编程模型，先要了解它定义的专业术语。Storm 的术语包括 Tuple、Stream、Spout、Bolt、Task、Stream Grouping 和 Topology 等，具体如图 10-1 所示。

下面依次对上述这几个术语进行介绍。

① 参见网址 <https://github.com/nathanmarz/storm/wiki>。

② 参见网址 <http://incubator.apache.org/s4/>。

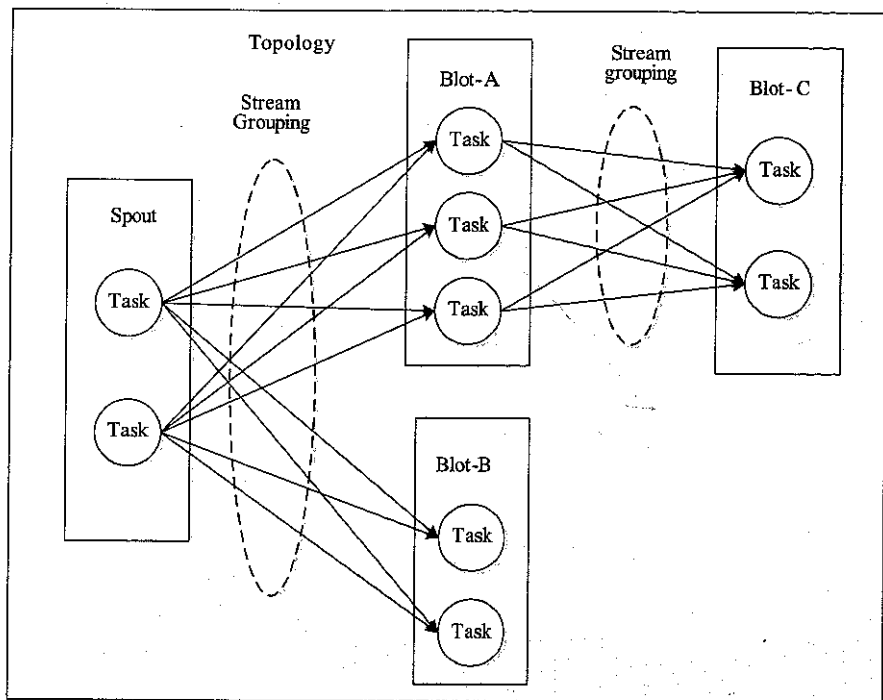


图 10-1 Storm 中的主要术语

- 数据表示模型 Tuple：Tuple 是 Storm 采用的数据表示模型，所有的数据都以 Tuple 的形式在各个组件之间流动。Tuple 是一组字段列表，每个字段由一个字段名和字段值组成，每个 Tuple 类似于数据库中的一行记录。在默认的情况下，Tuple 的字段类型可以是 integer、long、short、byte、string、double、float、boolean 和 byte array，当然，你也可以通过实现序列化器自定义类型。
- 消息流 Stream：Storm 将每个待处理或者新产生的 Tuple 封装成“消息”，而一个消息流（Stream）则是一个没有边界的 Tuple 序列，而这些 Tuple 会以一种分布式的方式被并行地创建和处理。
- 拓扑结构 Topology：一个 Storm 应用程序的处理逻辑被封装到 Topology 对象中，Topology 相当于 Hadoop 里面的一个 MapReduce Job，只不过一个 MapReduce Job 最终总是会结束的，而一个 Storm 的 Topology 会一直运行，除非显式地杀死它。一个 Topology 是由若干 Spout 和 Bolt 组成的图状结构，而链接这些 Spout 和 Bolt 的是 Stream grouping。
- 消息源 Spout：消息源是数据的产生者，它从一个外部源（比如消息队列）中读取数据并向 Topology 中发出消息 Tuple。消息源 Spout 可以是可靠的也可以是不可靠的。对于一个可靠的消息源，当一个 Tuple 未被成功处理时，它将重新发射这个 Tuple；而不可靠的消息源 Spout 则不同，它一旦发出一个 Tuple 便不再发送第二次。

- 消息处理者 Bolt：Storm 中消息处理逻辑被封装在 Bolt 中，Bolt 可以有多个，它们各自分工，完成不同功能，比如扫描、过滤、聚合、连接等。不同 Bolt 之间通常存在数据依赖关系，它们类似于“流水线”的形式合作完成数据处理，整个过程可抽象成一个“有向图”。
- 任务 Task：Spout 和 Bolt 可看作 Storm 的组件，每种组件可以被无限地并行化，每个并行化后的处理单元被称为一个“Task”，用户可根据需要设置每类 Spout 或者 Bolt 的并发任务数目，且该数目可以动态调整。
- 消息分发策略 Stream Grouping：Stream Grouping 用于定义一个 Stream 如何将消息分配给 Bolt 上面的多个 Task。这类似于 Map Task 与 Reduce Task 之间的数据划分方法，MapReduce 只提供了一种基于 Shuffle 的数据分配策略，而 Storm 则提供了 7 种。
 - Shuffle Grouping: 随机分组，随机派发 Stream 中的 Tuple，并保证每个 Bolt 接收到的 Tuple 数目相同。
 - Fields Grouping：按字段分组，比如按 id 字段分组，具有相同 id 值的 Tuple 会被分到相同的 Bolt 上，而具有不同 id 值的 Tuple 则会被分配到不同的 Bolt。
 - All Grouping: 广播发送，所有的 Bolts 会收到每一个 Tuple。
 - Global Grouping: 全局分组，每个 Tuple 被分配到 Storm 中的一个 Bolt 的某个 Task 上。再具体一点就是分配给 id 值最小的那个 Task。
 - Non Grouping: 不分组，即 Stream 不关心到底谁会收到它的 Tuple。目前这种分组和 Shuffle Grouping 效果一致，有一点不同的是 Storm 会把这个 Bolt 放到它的订阅者同一个线程里面去执行。
 - Direct Grouping: 直接分组，这是一种比较特别的分组方法，用这种分组意味着消息的发送者指定由消息接收者的哪个 Task 处理这个消息。只有被声明为 Direct Stream 的消息流可以声明这种分组方法。
 - Local or shuffle Grouping：如果目标 Bolt 有一个或者多个 Task 在同个工作进程中，Tuple 将会被随机发送给这些 Task，否则，和普通的 Shuffle Grouping 行为一致。

如果你学过 Hadoop MapReduce，那么应该知道接触到的第一个程序是 wordcount，这里我们同样从 wordcount 程序开始学习 Storm 编程模型。

在 Storm wordcount 例子中^①，构造了一个数据源，它采用随机算法生成一系列字符串，经过分词后，相同的单词被分配给同一个任务，并由该任务完成字符串计数功能，工作流程如图 10-2 所示，具体涉及的 Spout/Blot 如下：

- RandomSentenceSpout: 消息源 Spout 实现，它不断地从一组字符串中随机选择一个，发送给第一个 Blot；
- SplitSentence：第一个 Blot 实现，它接收来自 Spout 的字符串，并对其分词，采用

① 该例子来自 <https://github.com/nathanmarz/storm-starter>。

Fields Grouping 依次将每个单词发送给下一个 Blot (确保同一个单词发送给同一个 Task);

□ WordCount: 第二个 Blot 实现, 它接收来自 SplitSentence 的单词, 并对其进行计数。

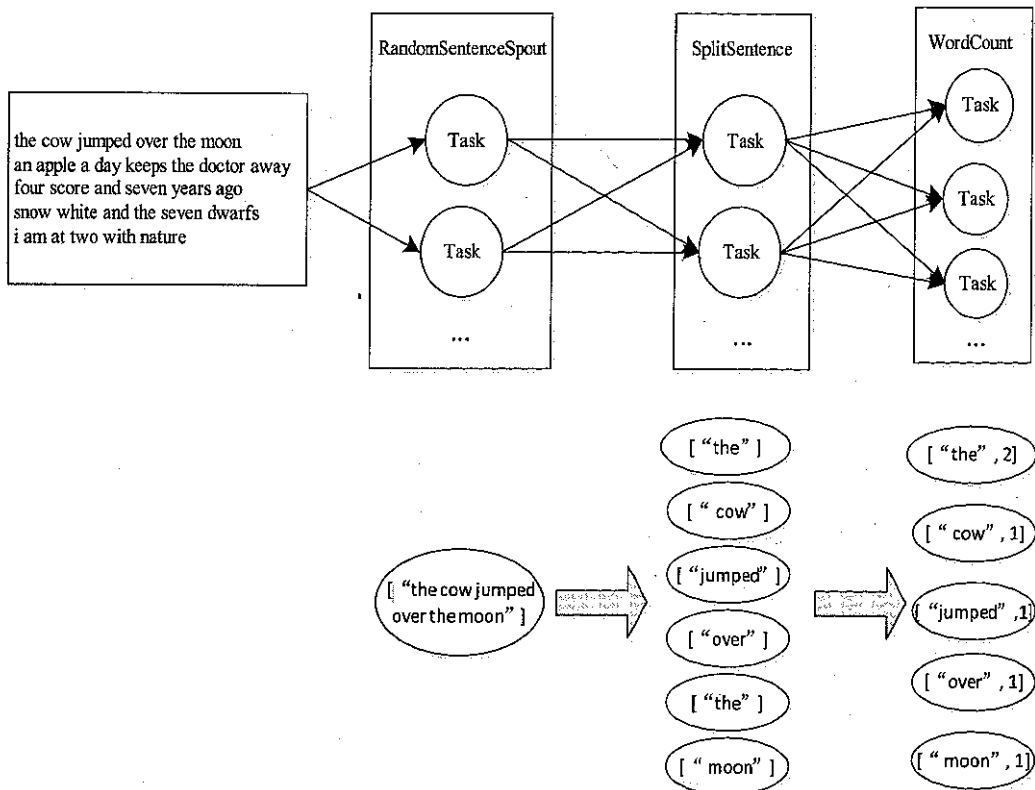


图 10-2 wordcount 工作流程

wordcount 的具体实现如下:

□ RandomSentenceSpout 实现, 代码如下:

```
// 用户实现的 Spout 要继承基类 BaseRichSpout
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector
collector) {
        _collector = collector;
        _rand = new Random(); // 定义一个随机产生器
    }

    @Override
```

```

public void nextTuple() {
    Utils.sleep(100);
    String[] sentences = new String[] {
        "the cow jumped over the moon",
        "an apple a day keeps the doctor away",
        "four score and seven years ago",
        "snow white and the seven dwarfs",
        "i am at two with nature"}; // 待选的 5 个字符串
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
}
...
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // 将输出字符串所在字段的名称定义为 "word"
    declarer.declare(new Fields("word"));
}
}

```

□ SplitSentence 实现，具体代码如下：

```

public static class SplitSentence extends ShellBolt implements IRichBolt {
    public SplitSentence() {
        super("python", "splitsentence.py"); // 采用 Python 脚本实现分词
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // 将输出字符串所在字段的名称定义为 "word"
        declarer.declare(new Fields("word"));
    }
    .....
}

```

其中，splitsentence.py 实现如下：

```

import storm
class SplitSentenceBolt(storm.BasicBolt): // 继承基类 storm.BasicBolt
    def process(self, tup):
        words = tup.values[0].split(" ") // 对句子分词
        for word in words:
            storm.emit([word]) // 依次输出每个单词

```

```
SplitSentenceBolt().run()
```

□ WordCount 实现，具体代码如下：

```

public static class WordCount extends BaseBasicBolt {
    // 使用 HashMap 保存单词与词频的映射关系
    Map<String, Integer> counts = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {

```

```

        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if(count==null) count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // 输出字段名称分别是 "word" 和 "count"
        declarer.declare(new Fields("word", "count"));
    }
}

```

□ 定义 Topology, 具体代码如下:

```

TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5); // 消息源任务数为 5
/*SplitSentence 并发任务数为 8, 且规定 RandomSentenceSpout 和 SplitSentence 之间用 Shuffle
grouping 策略划分消息 (均匀分配消息)*/
builder.setBolt("split", new SplitSentence(), 8)
    .shuffleGrouping("spout");
/*SplitSentence 并发任务数为 12, 并规定 SplitSentence 和 WordCount 之间使用 Fields
grouping 策略划分消息 (保证同一个单词最终交给同一个 task 处理)*/
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("split", new Fields("word"));

```

□ 将 Topology 提交到集群中, 具体代码如下:

```

public static void main(String[] args) throws Exception {
    ...
    Config conf = new Config();
    if(args!=null && args.length > 0) {
        conf.setNumWorkers(3); // 设置 worker 数目, worker 介绍见下一节
        StormSubmitter.submitTopology(args[0], conf,
            builder.createTopology()); // 提交到集群中
    }
}

```

10.2.2 Storm 基本架构

从整体架构上看, Storm 仍采用了 Master/Slave 架构, 但是 Master 与 Slave 不直接通信, 而是通过 Zookeeper 间接通信 (见图 10-3), 这使得 Storm 在容错性方面表现异常优秀^①。

Storm 由两种节点组成: (一个) 控制节点和 (多个) 工作节点。控制节点上面运行一个名为 Nimbus 的服务, 它的作用类似于 Hadoop MapReduce (MRv1) 中的 JobTracker, Nimbus 负责在集群里面分发代码、分配计算任务给机器并监控这些计算机的状态; 每一个工作节点上面运行一个名为 Supervisor 的服务, Supervisor 作用是根据需要启动 / 关闭工作

^① 参见网址 <https://github.com/nathanmarz/storm/wiki/Tutorial>。

进程 (Worker)。

每个 Supervisor 可同时启动一定数目的 Worker，每个 Worker 是一个进程，执行一个 Topology 的一个子集；一个运行的 Topology 由运行在很多机器上的多个 Worker 进程组成。

每个 Worker 内部可以启动多个用于运行同种组件 (Spout 或 Blot) 的 Executor，每个 Executor 是一个线程，可以运行某个组件的多个任务。

Nimbus 和 Supervisor 之间的所有协调工作都是通过 Zookeeper 集群完成的。Nimbus 进程和 Supervisor 进程都是快速失败和无状态的，它们所有的状态要么在 Zookeeper 里面，要么在本地磁盘上。这也就意味着可以用 kill -9 来杀死 Nimbus 和 Supervisor 进程，然后再重启它们，就好像什么都没有发生过。这个设计使得 Storm 异常稳定。

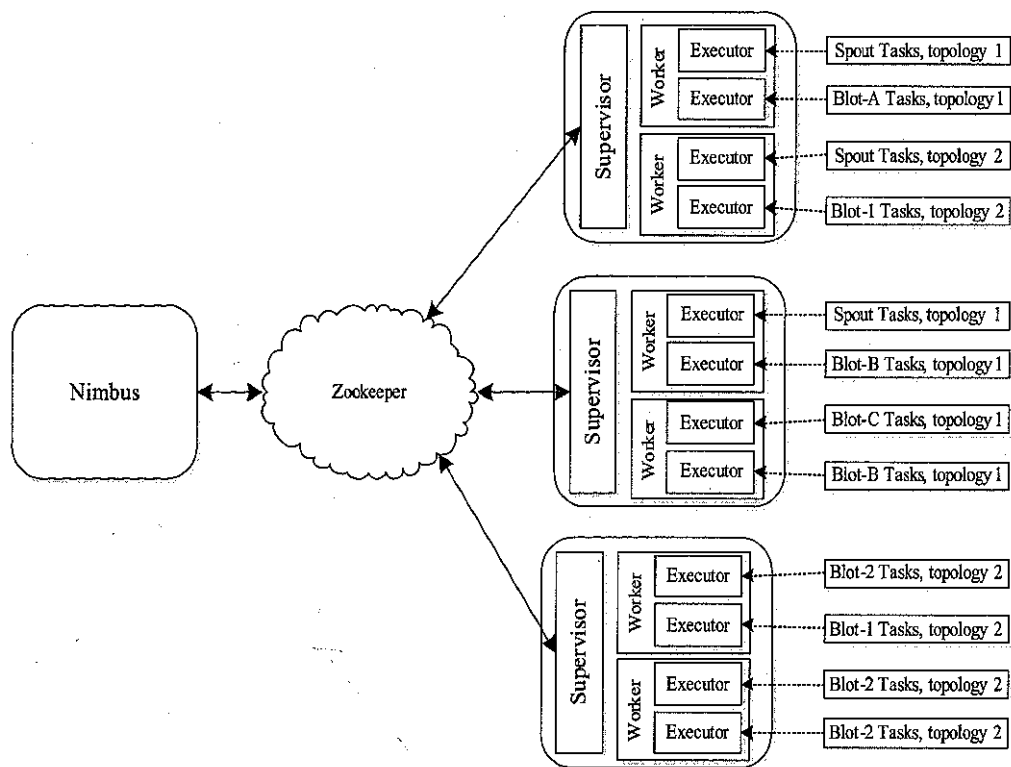


图 10-3 Storm 基本架构

为了进一步加深对 Storm 基本架构的理解，表 10-1 将之与 MRv1 主要组件进行了类比。

表 10-1 MRv1 与 Storm 对比

	Hadoop MapReduce (MRv1)	Storm
系统服务	JobTracker	Nimbus
	TaskTracker	Supervisor
	Child	Worker

(续)

	Hadoop MapReduce (MRv1)	Storm
应用程序名称	Job	Topology
编程模型	Map/Reduce	Spout/Blot
	Shuffle	Stream grouping

10.2.3 Storm On YARN

本小节我们具体介绍 Storm On YARN。

(1) Storm On YARN 带来的好处

相比于将 Storm 部署到一个独立的集群中, Storm On YARN 带来的好处很多,主要有以下几个:

- 弹性计算资源。将 Storm 运行到 YARN 上后, Storm 可与其他应用程序(比如 MapReduce 批处理应用程序)共享整个集群中的资源,这样当 Storm 负载骤增时,可动态为它增加计算资源;而当负载减小时,可释放部分资源,从而将这些资源暂时分配给负载更重的批处理应用程序。
- 共享底层存储。Storm 可与运行在 YARN 上的其他框架共享底层的一个 HDFS 存储系统,可避免多个集群带来的维护成本,同时避免数据跨集群复制带来的网络开销和时间延迟。
- 支持多版本。可同时多个 Storm 版本运行于 YARN 上,避免一个版本一个集群带来的维护成本。

(2) Storm On YARN 架构

第 4 章已经介绍了如何在 YARN 上开发一个应用程序,通常而言,需要开发两个组件,分别是客户端和 ApplicationMaster,其中客户端的主要作用是将应用程序提交到 YARN 上,并与 YARN 和 ApplicationMaster 交互,完成用户发送的一些指令;而 ApplicationMaster 则负责向 YARN 申请资源,并与 NodeManager 通信,以启动任务。

不修改 Storm 任何源代码即可让 Storm 运行在 YARN 上,最简单的实现方法是将 Storm 的各个服务组件(包括 Nimbus 和 Supervisor)作为单独的任务运行在 YARN 上,而 Zookeeper 则作为一个公共的服务运行在 YARN 集群之外的几个节点上。总之,为了让 Storm 运行在 YARN 上,可经过以下几个步骤(见图 10-4):

步骤 1 通过 YARN-Storm Client 将 Storm Application 提交到 YARN 上;

步骤 2 ResourceManager 为 YARN-Storm ApplicationMaster 申请资源,并将之运行在一个节点上;

步骤 3 YARN-Storm ApplicationMaster 在自己内部启动 Nimbus 和 Web UI 服务;

步骤 4 YARN-Storm ApplicationMaster 按照用户配置向 ResourceManager 申请资源,并在申请到的 Container 中启动 Supervisor 服务;

步骤 5 与向普通 Storm 集群提交 Topology 一样，用户向运行在 YARN 之上的 Storm 集群提交 Topology。

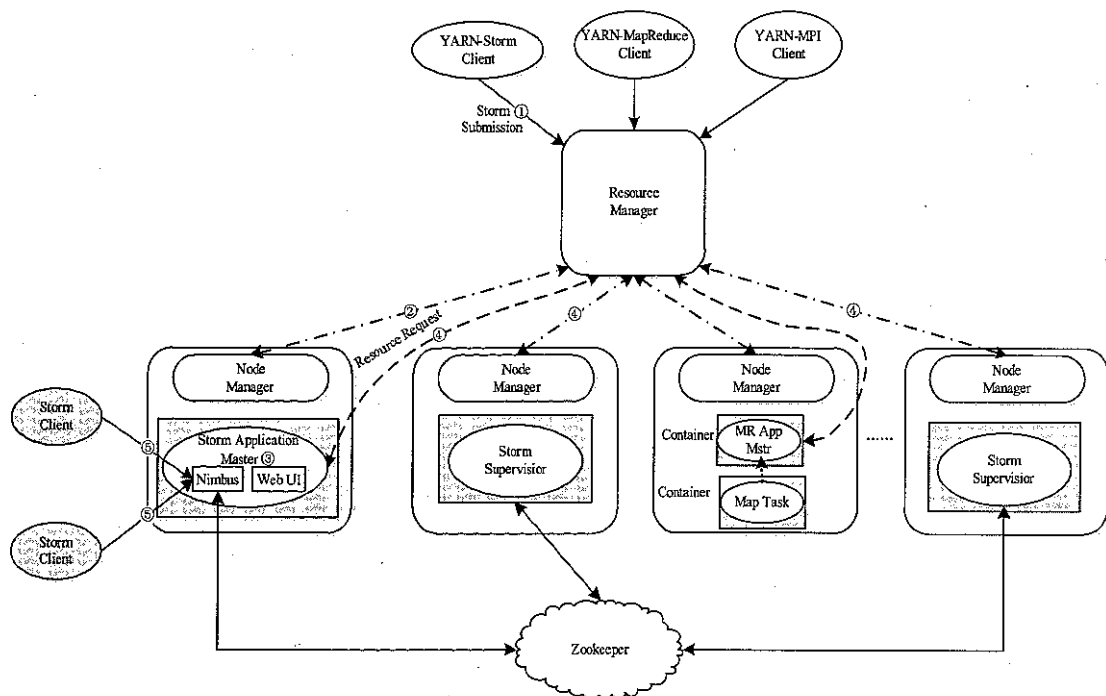


图 10-4 Storm On YARN

本章的编写目的并不是给出一个精确的“Storm On YARN”实现方案，而是尝试着给出一个可行的设计框架，并在一些设计细节处讨论多种可行方案，进而引导读者根据自己业务特点设计符合实际需求的方案。下面就几个设计细节进行讨论。

- YARN-Storm Client 包含的功能：YARN-Storm Client 应包含其他计算框架的 YARN Client 所具备的通用功能，比如提交应用程序（构造 Storm 集群）、杀死应用程序（撤销 Storm 集群）等，此外为了能够通过 YARN 构造一个弹性 Storm 计算框架，还应支持增加 / 减少 Supervisor 服务个数、增加 / 减少 Supervisor 服务所占用的内存等。
- YARN-Storm ApplicationMaster 包含的功能：YARN-Storm ApplicationMaster 是最核心的部件之一，它除了能为其他 ApplicationMaster 申请资源、与 NodeManager 通信启动任务之外，还需要完成 YARN-Storm Client 期望的特有功能，比如支持增加 / 减少 Supervisor 服务个数。当然，为了让 Storm 变得更加智能，ApplicationMaster 可探测 Topology 数目和对资源的需求，进而自动增加 / 减少 Supervisor 服务个数。
- Nimbus 服务的启动位置：上面给出的方案中，Nimbus 直接启动在 ApplicationMaster，这样做有一个好处是，ApplicationMaster 与 ResourceManager 之间维持了心跳信息，一旦 ApplicationMaster 失败后，ResourceManager 将自动重启它（用户可以为 YARN 中的每

个 ApplicationMaster 设置最多重启次数), 而 Nimbus 是无状态的, 因此 ApplicationMaster 重启后, Storm Nimbus 可再次变得可用。考虑到 ApplicationMaster 维护了 Storm 集群的资源分配, 因此 ApplicationMaster 需日志记录各个 Supervisor 的位置及占用的资源 (另一种比较简单的方案是从 Zookeeper 获取, 当然, 前提是 Zookeeper 上已经保存了这些信息), 以便失效重启后重构这些信息。

- Supervisor 服务所占资源量: Supervisor 服务占用的资源量直接影响到它的隔离性。这通常有两种可选方案, 一种是每个 Supervisor 占用某个节点上的全部资源, 这意味着 Supervisor 是独占机器资源的, 这种情况下, Storm 集群受其他应用程序的影响最小; 另一种是每个 Supervisor 只占用某个节点上的部分资源, 这意味着 Supervisor 服务与其他应用程序的任务混搭运行在同一个节点上, 这可以提高节点整体资源利用率, 但会对 Storm 上的业务造成干扰^①。另外, 需要强调的是, 这两种方案都仅仅实现了服务级别的隔离, 而没有像 MapReduce 和 Tez 那样, 实现更细粒度的作业 (应用程序) 级别的隔离。

当前比较有名的 “Storm On YARN” 实现是由 Yahoo! 开源的^②, 它基本实现了上述描述的功能, 下面具体进行说明:

(1) YARN-Storm Client

YARN-Storm Client 提供了一系列 Shell 命令供用户控制 YARN 上的 Storm 服务, 比如构建一个 Storm 集群命令如下:

```
storm-yarn launch <storm-yarn-config>
```

其中, <storm-yarn-config> 是 Storm 配置信息, 包括启动的 Supervisor 个数、Storm ApplicationMaster 占用的内存等。

启动 Storm 之后, 用户可通过以下命令控制 Storm:

```
storm-yarn [command] -appId [appId] -output [file] [-supervisors [n]]'
```

其中, Command 为具体命令 (见表 10-2), 参数 “-appId” 为启动的 Storm 的应用程序 ID, “-supervisors” 为需增加的 Supervisor 服务个数, 该参数只对命令 “addSupervisors” 有效。

表 10-2 Storm On YARN 主要参数汇总

Shell 命令	含 义	说 明
setStormConfig	重置 Storm 集群配置	整个集群将重新启动
getStormConfig	获取当前 Storm 集群配置	所有配置属性将以 JSON 格式返回
addSupervisors	增加 Supervisor 个数	ApplicationMaster 向 ResourceManager 申请资源以启动新增的 Supervisor

① 注意, 尽管 YARN 采用 Cgroups 对各个任务进行了隔离, 但当前只做到了 CPU 和内存两种资源的隔离, 其他资源, 比如网络和磁盘, 尚不能实现隔离。

② 参见网址 <https://github.com/yahoo/storm-yarn>。

(续)

Shell 命令	含 义	说 明
startNimbus/stopNimbus	启动 / 停止 Nimbus 服务	Nimbus 服务运行在 ApplicationMaster 中
startUI/stopUI	启动 / 停止 Web UI 服务	Web UI 服务运行在 ApplicationMaster 中
startSupervisors/ stopSupervisors	启动 / 停止所有 Supervisor 服务	Supervisor 服务运行在 ApplicationMaster 向 ResourceManager 申请的 Container 中
shutdown	销毁整个 Storm 集群	所有服务将被杀死, 资源将归还给 Resource-Manager

结合使用 startNimbus/stopNimbus、startUI/stopUI 和 startSupervisors/ stopSupervisors 等命令, 可完成对 Storm 集群的升级。

(2) YARN-Storm ApplicationMaster

Storm ApplicationMaster 初始化时, 将在同一个 Container 中启动 Storm Nimbus 和 Storm Web UI 两个服务, 然后根据待启动的 Supervisor 数目向 ResourceManager 申请资源。在目前实现中, ApplicationMaster 将请求一个节点上所有资源然后启动 Supervisor 服务, 也就是说, 当前 Supervisor 将独占节点而不会与其他服务共享节点资源, 这种情况下可避免其他服务对 Storm 集群的干扰。

除了运行 Storm Nimbus 和 Web UI 外, Storm ApplicationMaster 还会启动一个 Thrift Server 以处理来自 YARN-Storm Client 端的各种请求, 在此不再赘述。

10.3 内存计算框架 Spark

Spark^⑨发源于美国加州大学伯克利分校 AMPLab 实验室的集群计算平台, 它克服了 MapReduce 在迭代式计算和交互式计算方面的不足, 通过引入 RDD (Resilient Distributed Datasets) 数据表示模型, 能够很好地解决 MapReduce 不易解决的问题。相比于 MapReduce, Spark 能够充分利用内存资源提高计算效率, 因此本书将 Spark 归为“内存计算框架”。

现有的计算框架, 比如 MapReduce, 每次均需从 HDFS 上读取数据进行处理, 且不同计算之间只能通过 HDFS 重用数据, 这使得在以下几种应用场景下计算效率极其缓慢:

□ **迭代计算**。在数据挖掘、机器学习、自然语言处理等领域, 有很多算法需要迭代多轮才能得到最终结果, 比如 PageRank, K-means 聚类、逻辑回归等算法。这些算法的典型特点是, 相邻两轮计算要共享一部分数据。如果使用 MapReduce, 则只能将 HDFS 作为可行的存储系统。

□ **交互式计算**。在数据仓库应用中, 用户通过大量的查询语句从一个或多个数据集上获取结果, 通常而言, 这些查询语句中的部分计算是相同的, 即作用在同一个数据集上的相同运算。当使用 Hive 或者 Pig 这样的交互式系统时, 由于它们每次都会重

⑨ <http://spark-project.org/>。

复计算，但无法重用计算结果，因此效率极其低下。

为了解决以上几种应用场景中的问题，Spark 引入了弹性分布式数据集（RDD），它是一个有容错机制、可以被并行操作的数据集合，能够被缓存到内存中，供其他计算使用，而不必像 MapReduce 那样每次都从 HDFS 上重新加载数据。Spark 提供了两种针对 RDD 的操作：

- **转换（transformation）**。根据已有的 RDD 数据集创建一个新的 RDD 数据集。
- **动作（action）**。在 RDD 数据集上运行计算后，返回一个值或者将结果写入外部存储系统。

本节将详细介绍 Spark 的编程模型、基本架构和 Spark 如何运行在 YARN 上。

10.3.1 Spark 编程模型

一个 Spark 用户程序主要包含一个驱动程序，该驱动程序执行用户的 main 函数并将各种算子分布到集群中并行运行。Spark 中最主要的抽象是将数据集抽象成 RDD，这是一种分布到各个节点上的数据集合，且能够被并行处理。RDD 是通过 HDFS 上的文件或者其他 Scala 数据集创建的，并可以通过一定的逻辑转换成另外一种 RDD。此外，RDD 可被缓存到内存中，进而被多个并行执行的任务重用。最后，RDD 是能够容错的，当一个节点宕掉后，丢失的 RDD 可被重构。

Spark 另外一个抽象是可被并行算子共用的共享变量。默认情况下，Spark 会将一个算子转化为一组任务，并调度到各个节点上来执行。它会为每个任务拷贝一个算子中的变量，有些情况下，这些任务需要共享一些变量，或者任务与驱动程序之间共享一些变量，而 Spark 共享变量正是为此设计的。Spark 支持两种共享变量：

- **广播变量**。可以缓存到各个节点的内存中的变量，通常为只读，类似于 Hadoop 中 DistributedCache 的数据。
- **累加器**。只能用来做加法，例如计数或求和，类似于 Hadoop 中的 Counter。

注意，Spark 之所以为轻量级，一个重要原因是充分利用了 Scala 语言简洁强大的特点，因此学习 Spark 之前，应该对 Scala 语言有一个基础的理解。

1. 初始化 Spark

当需要导入一些 Spark 的类和隐式转换时，需将下面几行代码加入到程序的顶部：

```
import spark.SparkContext
import SparkContext._
```

写 Spark 程序需要做的第一件事情是创建一个 SparkContext 对象，它封装了 Spark 程序的执行环境，这个通常是通过下面的构造器来实现的：

```
new SparkContext(master, jobName, [sparkHome], [jars])
```

master 参数是一个字符串，指定了需连接的 YARN 集群或者 Mesos 集群，也可以用特殊的字符串“local”来指明用 local 模式运行；jobName 是作业的名称；sparkHome 是

Spark 在集群机器上的安装路径（所有节点必须全部一致）；jars 是本地节点上包含了作业代码和依赖的 Jars 文件列表。Spark 会把它们部署到所有的集群节点上。

2. 分布式数据集

Spark 中最核心的概念是弹性分布式数据集（RDD），一个有容错机制，可以被并行操作的集合。目前主要有两种类型的 RDD：并行集合，接收一个已经存在的 Scala 集合，并在它上面运行各种并发计算；Hadoop 数据集，类似于 MapReduce 模式，可在一个数据片的每条记录上，运行各种函数，但比 MapReduce 更加灵活。这两种集合继承自相同的父类，可以通过一系列相同的算子进行操作。

（1）并行集合

并行集合是通过调用 SparkContext 的 parallelize 方法，在一个已经存在的 Scala 集合上创建而来。集合的对象将会被复制成多份以创建一个分布式数据集，进而实现并行处理。下面通过 Spark 解释器的例子，展示如何从一个数组创建一个并行集合：

```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)
scala> val distData = sc.parallelize(data)
distData: spark.RDD[Int] = spark.ParallelCollection@10d13e3e
```

在上面例子中，一旦被创建，并行集合 distData 可以被并行操作。例如，可以调用 distData.reduce(_+_) 将数组的所有元素累加起来。

另外，需要说明的是，创建并行集合的一个重要参数是 slice 的数目，即将数据集切分为几份（类似于 MapReduce 中的 InputSplit）。在集群模式中，Spark 将在每份 slice 启动一个 Task 进行处理。一般来说，Spark 会尝试根据集群的状况，来自动设定 slice 的数目。当然，也允许通过 parallelize 方法的第二个参数（如 sc.parallelize(data, 10)）手动设置 slice 的数目。

（2）Hadoop 数据集

为了更好地与 Hadoop 结合，Spark 允许从任何存储在 HDFS 文件系统或者 Hadoop 支持的其他文件系统（包括本地文件、HBase 等）上的文件创建数据集。目前，Spark 可以支持文本文件、SequenceFile 及其他任何 Hadoop 输入格式（实现 InputFormat）。

对于文本文件，RDD 可以通过 SparkContext 的 textFile 方法创建，该方法接受文件的 URI 地址，举例如下：

```
scala> val distFile = sc.textFile("data.txt")
distFile: spark.RDD[String] = spark.HadoopRDD@1d4cee08
```

一旦被 RDD 创建，RDD 数据集 distFile 就可以进行并行操作了。例如，可以使用如下 Map 和 Reduce 操作将所有行数的长度相加：

```
distFile.map(_._size).reduce(_+_ )
```

当然，textFile 方法也接受可选的第二参数来控制文件的分片数目。默认情况下，Spark 为每个 HDFS 数据块创建一个分片（默认大小为 64MB），但用户可以通过传入一个更大的

值来指定更多的分片。

对于 SequenceFile, RDD 可通过 SparkContext 的 sequenceFile[K, V] 方法创建 (K 和 V 分别是文件中的 key 和 values 类型, 它们必须是 Hadoop 的 Writable 的子类, 例如 IntWritable 和 Text)。

最后, 对于其他类型的 Hadoop 输入格式, 可以使用 SparkContext.hadoopRDD 方法, 它可以接收任意类型的 JobConf 和输入格式、Key 类型和 Value 类型。总之, 用户可按照对 Hadoop 作业一样的方法设置 Spark 输入源。

3. 分布式数据集操作

总起来说, 分布式数据集支持两种操作:

□ 转换: 根据现有的 RDD 数据集创建一个新的 RDD 数据集。

□ 动作: 在 RDD 数据集上运行计算后, 将计算结果传给驱动程序或者写入存储系统。

默认情况下, 每次 RDD 转换是通过重新计算得到结果的, 然而, 为了便于 RDD 数据集重复利用, 用户可使用 persist 或 cache 方法将 RDD 数据集缓存到内存或者磁盘中。当然, 用户可灵活选择将 RDD 数据集缓存到内存中、磁盘上或其他节点上。缓存是 Spark 中构造迭代算法的关键工具, 甚至可以在解释器中交互使用。

Spark 当前支持的部分转换和动作分别如表 10-3 和表 10-4 所示。

表 10-3 Spark 转换操作 (部分)

转 换	解 释
map(func)	数据集每条元素经过 func 函数转换后形成一个新的分布式数据集
filter(func)	数据集中让 func 函数返回值为 true 的元素形成一个新的分布式数据集
flatMap(func)	类似于 Map, 但是每个输入元素可能会被映射为 0 到多个输出元素
groupByKey([numTasks])	按照 key 进行分组, 即在一个由 (K,V) 对组成的数据集上调用, 返回一个 (K, Seq[V]) 对的数据集。其中, numTasks 为并行任务数目
reduceByKey(func, [numTasks])	类似于 MapReduce 中的 Reduce 阶段, 将数据按照 key 分组后, 调用 func 函数处理。其中, numTasks 为并行任务数目
join(otherDataset, [numTasks])	根据 Key 连接两个数据集, 即将类型为 (K,V) 和 (K,W) 类型的数据集合并成一个 (K,(V,W)) 类型数据集
sortByKey([ascendingOrder])	按照 key 对数据集进行排序, 其中参数 ascendingOrder 决定升序还是降序

表 10-4 Spark 动作 (部分)

动 作	解 释
reduce(func)	通过函数 func 对数据集的所有元素进行规约操作。func 函数接受两个参数并返回一个值。需要注意的是, 这个函数必须具有可交换性和关联性, 以确保可以被正确地并发执行
collect()	在驱动程序中以数组的形式返回数据集的所有元素。这通常在 filter 或者其他操作后调用, 返回一个足够小的数据集
count()	返回数据集的元素个数

(续)

动 作	解 释
countByKey()	只能用于 <K,V> 类型的 RDD 数据集,它能够按照 key 分组,并统计每个组的元素目,即针对每个 key 返回一个 (K, Int) 对
saveAsTextFile(path)	将数据集的元素以 textFile 的形式保存到本地文件系统、HDFS 或者任何其他 Hadoop 支持的文件系统
saveAsSequenceFile(path)	将数据集的元素以 Sequencefile 的格式保存到指定的目录下、本地系统、HDFS 或者任何其他 Hadoop 支持的文件系统

4. RDD 缓存

用户可使用 persist 或 cache 方法将任意 RDD 数据集缓存到内存或者磁盘中。RDD 是能够容错的,如果一个 RDD 分片丢失了,Spark 能通过最初构建它的转换算法重构它。此外,用户可使用不同的缓存级别对 RDD 进行缓存。比如,可将数据集缓存到磁盘上,或者以 Java 序列化对象的形式缓存到内存中,甚至复制到多个节点上。用户可通过 spark.storage.StorageLevel 类型的参数设置 RDD 缓存级别,Spark 目前支持的 RDD 缓存级别如表 10-5 所示。

表 10-5 RDD 缓存级别

缓存级别	解 释
MEMORY_ONLY	将 RDD 数据集以 Java 对象的形式保存到 JVM 内存中,如果有些分片太大不能够保存到内存中,则将不再缓存而是每次用时重算。该级别是默认缓存级别
MEMORY_AND_DISK	将 RDD 数据集以 Java 对象的形式保存到 JVM 内存中,如果有些分片太大不能够保存到内存中,则将其保存到磁盘上,并在下次使用时重新从磁盘上读取
MEMORY_ONLY_SER	将 RDD 数据集以序列化后的 Java 对象形式保存到 JVM 内存中(一个分片一个大字节数组),这种方式更加高效,尤其是采用高效的序列化器,比如 kryo 序列化器 ^①
MEMORY_AND_DISK_SER	与 MEMORY_ONLY_SER 类似,但当分片太大不能够保存到内存时,会将其保存到磁盘上
DISK_ONLY	将 RDD 数据集保存到磁盘上
MEMORY_ONLY_2, MEMORY_AND_DISK_2 等	与 DISK_ONLY 类似,但将每个分片复制到两个节点上

在实际应用时,具体采用哪种缓存级别,完全由用户自己决定。通常而言,如果内存够用,则尽量采用 MEMORY_ONLY,否则尝试使用 MEMORY_ONLY_SER。不要将数据集写到磁盘上,除非该数据集重算代价十分昂贵。另外,如果你想让某个 RDD 数据集具有很强的容错性,可采用多备份缓存级别。

^① 参见网址 <http://code.google.com/p/kryo/>。

5. 共享变量

通常而言，当一个函数被传递给 Spark 操作后，该函数将在集群节点上并行运行，且函数中用到的所有变量都在各节点上分别复制了一份，以供函数使用而不会互相影响。这些变量会被复制到每一台机器上，而在远程机器上，在对变量的所有更新都不会被传播回驱动程序中。然而，为了用户使用方便，Spark 提供两种有限的共享变量：广播变量和累加器。

(1) 广播变量

广播变量允许用户保留一个只读的变量并将之缓存到每一台机器上（并非每个任务单独保存一份副本）。广播变量是从变量 *v* 创建的，通过调用 `SparkContext.broadcast(v)` 方法实现变量广播。这个广播变量是一个 *v* 的分装器，它只可通过调用 `value` 方法获得。如下代码展示了如何应用广播变量：

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: spark.Broadcast[Array[Int]] =
  spark.Broadcast(b5c40191-a864-4c7d-b9bf-d87e1a4e787c)
scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

广播变量被创建后，能在集群运行的任何函数中调用，从而避免 *v* 值重复传递到这些节点上。另外，对象 *v* 是只读的，不能在广播后修改，从而保证所有节点的变量收到的都是一模一样的。

(2) 累加器

累加器是只支持加法操作的变量，可以高效地并行化。用户可以用它实现计数器和变量求和。Spark 原生支持 `Int` 和 `Double` 类型的计数器，用户可根据需要添加新的类型。

用户可以通过调用 `SparkContext.accumulator(v)` 方法来创建计数器（*v* 是初始值），而运行在集群上的任务可以使用“加法”增加该值。然而，运算过程中，这些任务不能读取计数器的值，只有驱动程序，可获取累加器的值。

如下的解释器展示了如何利用累加器将一个数组里面的所有元素相加：

```
scala> val accum = sc.accumulator(0)
accum: spark.Accumulator[Int] = 0
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value
res2: Int = 10
```

10.3.2 Spark 基本架构

与 Storm 不同，Spark 中每个应用程序维护了自己的一套运行时环境，该运行时环境在应用程序开始运行时构建，在运行结束时销毁。相比于 Storm 这种所有应用程序公用一套运行时环境的方式，它能够极大地缓解应用程序之间相互影响，Spark 工作原理如图 10-5 所示。

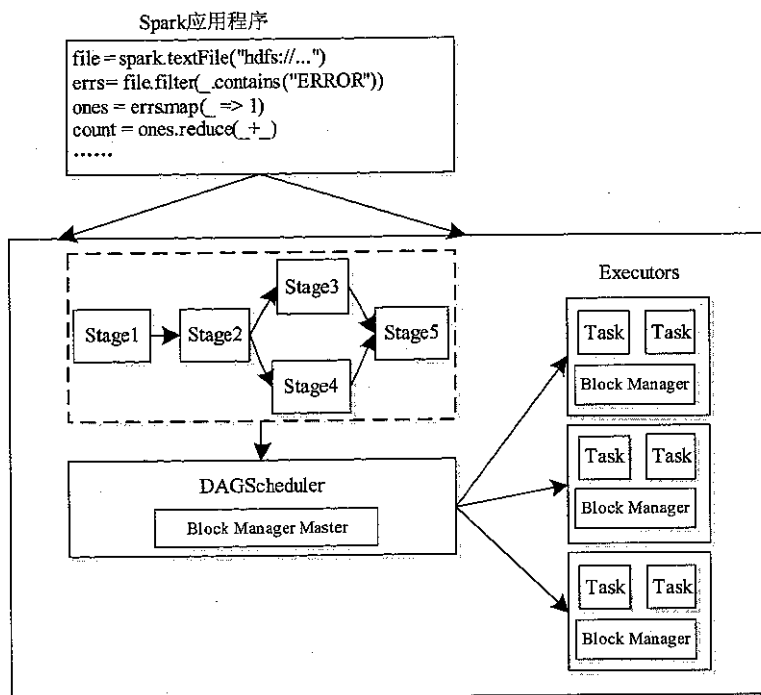


图 10-5 Spark 工作原理图

总体来说，一个 Spark 运行时环境由四个阶段构成：

- 阶段 1：构建应用程序运行时环境。
- 阶段 2：将应用程序转换成 DAG 图。
- 阶段 3：按照依赖关系调度执行 DAG 图。
- 阶段 4：销毁应用程序运行时环境。

接下来详细介绍这四个阶段。

(1) 阶段 1：构建应用程序运行环境

不管是何种计算框架，均要包含两部分功能：资源管理和应用程序管理，其中，资源管理模块是通用的，而应用程序管理模块因计算框架的不同而差别很大。正因如此，随着计算框架类型的不断增多，类似 Mesos 和 YARN 这种通用资源管理系统便出现了。这种系统可使计算框架的开发工作大大减少，用户只需专注于应用程序管理模块的开发即可，而 Spark 正是基于以上思想设计的。

为了运行一个应用程序，Spark 首先根据应用程序资源需求构建一个运行时环境，这是通过与 YARN 或者 Mesos 这种资源管理系统交互来完成的。通常而言，存在两种运行时环境构建方式：

- 粗粒度构建方式：应用程序被提交到 YARN 或者 Mesos 集群上后，它在正式运行任务之前，将根据应用程序资源需求一次性将这些资源凑齐，之后使用这些资源运行

任务，整个运行过程中不再申请新资源。

- 细粒度构建：应用程序被提交到 YARN 或者 Mesos 集群上后，动态向 YARN 或者 Mesos 申请资源，只要得到的资源足以运行一个任务，便开始运行该任务，而不必等到所有资源全部到位。

对于 Spark On Mesos，支持粗粒度构建和细粒度构建两种方式；对于 Spark On YARN，目前仅支持粗粒度构建方式。不管何种方式，除了启动任务相关的组件外，每个节点还需启动一个 RDD 缓存管理服务 BlockManager，该服务采用了分布式 master/slaves 架构，其中，主控节点上启动 master 服务—BlockManagerMaster，它掌握着所有 RDD 缓存位置，而从节点则启动了供客户端存取 RDD 的 slave 服务。

(2) 阶段 2：将应用程序转换成 DAG 图

下面详解阶段 2 的工作过程。RDD 的数据结构里很重要的一个域是对父 RDD 的依赖。如图 10-6 所示，依赖有两类——窄（Narrow）依赖和宽（Wide）依赖。

窄依赖指父 RDD 的每一个分区最多被一个子 RDD 的分区所用，表现为一个父 RDD 的分区对应于一个子 RDD 的分区和两个父 RDD 的分区对应于一个子 RDD 的分区。图 10-6 中，map/filter 和 union 属于第一类，对输入进行协同划分（co-partitioned）的 join 属于第二类。

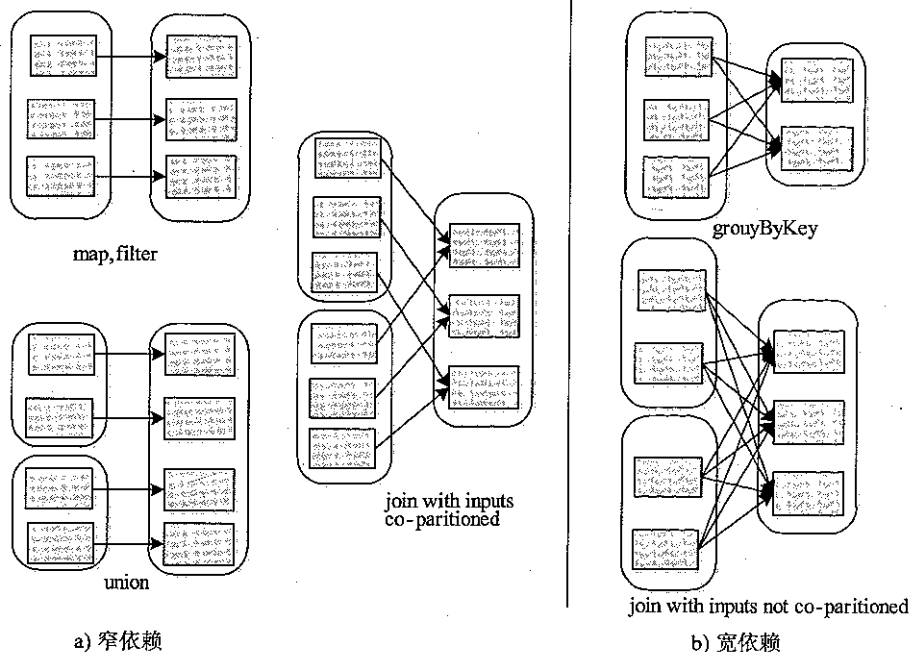


图 10-6 窄依赖与宽依赖

宽依赖指子 RDD 的分区依赖于父 RDD 的所有分区，图 10-6 中的 groupByKey 和未经

协同划分的 join 属于宽依赖。

应用程序运行过程中, Spark 按 RDD 依赖关系将应用程序划分成若干个 Stage, 每个 Stage 启动一定数目的任务进行并行处理。Spark 采用了贪心算法划分阶段, 即如果子 RDD 的分区到父 RDD 的分区是窄依赖, 就可以实施经典的 fusion 优化, 把对应操作划分到一个阶段。如果连续的变换算子序列都是窄依赖, 就可以把很多个操作并到一个阶段, 直到遇到一个宽依赖。这不但减少了大量的全局 barrier, 而且无须物化很多中间结果 RDD, 这将极大地提升性能。Spark 把这个称为流水线 (pipeline) 优化。图 10-7 给出了一个应用程序转化为 DAG 阶段后的结果 (其中黑框表示已经缓存在内存中的 RDD 数据集), 该程序最终被转化为 3 个阶段, 其中每个阶段将启动多个任务并行处理。

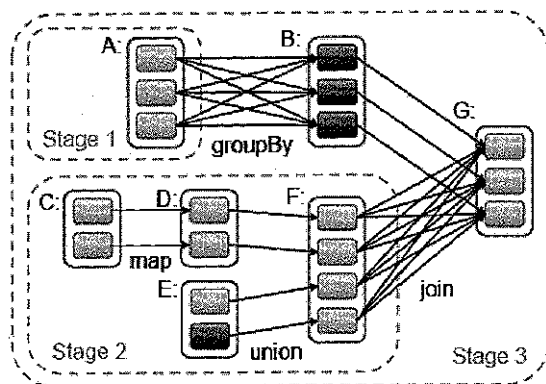


图 10-7 Stage 划分

(3) 阶段 3: 按照依赖关系调度执行 DAG 图

在该阶段中, DAGScheduler 将按照依赖关系调度执行每个阶段: 优先选择那些不依赖任何阶段的阶段, 待这些阶段执行完成后, 再调度那些依赖的阶段已经运行完成的阶段, 依此进行, 这样一直调度下去, 直到所有阶段运行完成。在处理某个阶段时, Spark 将为之启动一定数目的任务并行执行, 为了提高任务的执行效率, Spark 借鉴 MapReduce 中的若干优化机制:

- **数据本地性。**数据本地性是指对任务进行调度时 (即任务选择合适的节点), 优先选择数据所在节点, 其次是数据所在机架上的其他节点, 最后考虑其他机架上的节点。考虑到输入数据量较少时数据本地性会变差, Spark 借鉴了 MapReduce 中采用了 Delay Scheduling 机制, 即当不存在满足本地性的资源时, 暂时将资源分配给其他任务, 直到出现满足本地性的资源或者达到设定的最大时间延迟。
- **推测执行。**当检测到同类任务中存在明显拖后腿的任务时, Spark 借鉴了 MapReduce 中的推测执行机制, 即尝试为这些拖后腿的任务启动备份任务, 并将最先完成的任务的计算结果作为最终结果。

(4) 阶段 4: 销毁应用程序运行时环境

阶段4与阶段1互为逆过程，阶段4释放应用程序占用的资源，归还给YARN或者Mesos集群，以便供其他应用程序使用。

10.3.3 Spark On YARN

与Storm On YARN中所有应用程序共用一套运行环境不同，Spark On YARN更加细粒度，每个Spark应用程序拥有一套运行环境，开始运行时创建，运行结束时销毁。第4章已经介绍了如何在YARN上开发一个应用程序，通常而言，需要开发两个组件，分别是客户端和ApplicationMaster，其中，客户端的主要作用是将应用程序提交到YARN上，并与YARN和ApplicationMaster交互，完成用户发送的一些指令；而ApplicationMaster则负责向YARN申请资源，并与NodeManager通信，以启动任务。

为了能够将Spark运行在YARN上，Spark开发了一套客户端和ApplicationMaster，其中，客户端负责将应用程序提交到YARN上，而ApplicationMaster负责启动ClusterScheduler，为执行任务申请资源、监控任务运行状态并提供容错机制等。

总结起来，为了让Spark运行在YARN上，可经过以下几个步骤（见图10-8）：

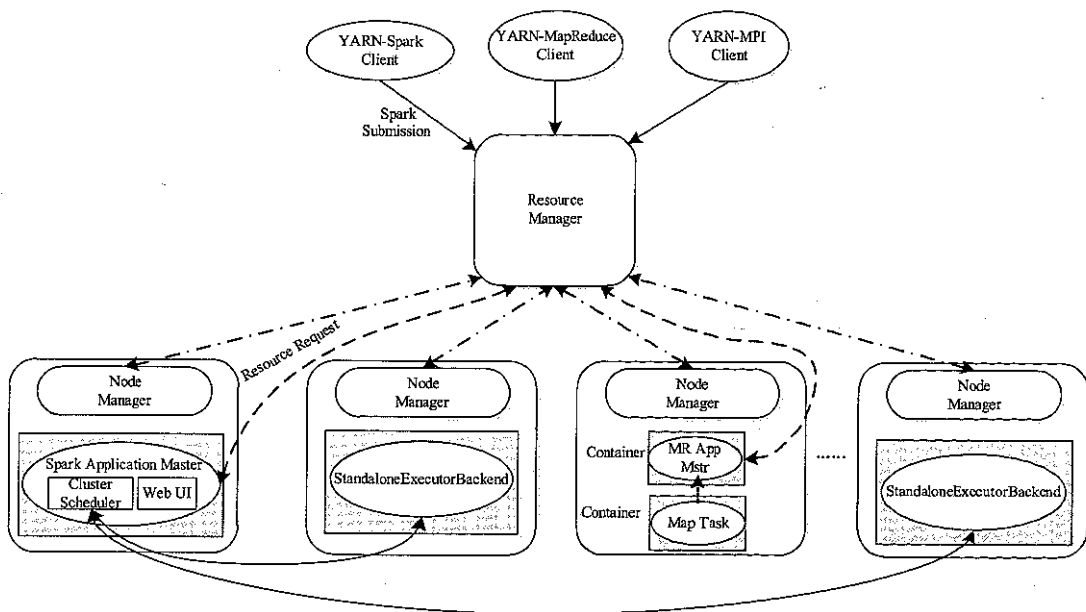


图 10-8 Spark On YARN

步骤1 通过YARN-Spark Client将Spark Application提交到YARN上；

步骤2 ResourceManager为YARN-Spark ApplicationMaster申请资源，并将之运行在一个节点上；

步骤3 YARN-Spark ApplicationMaster在自己内部启动ClusterScheduler，生成DAG图，启动Web UI服务等；

步骤 4 YARN-Spark ApplicationMaster 按照用户配置向 ResourceManager 申请资源,并在申请到的 Container 中启动 StandaloneExecutorBackend 服务, StandaloneExecutorBackend 通过 akka 向 ClusterScheduler 注册,以等待领取任务;

步骤 5 ClusterScheduler 向 StandaloneExecutorBackend 分配新任务, StandaloneExecutorBackend 收到任务后执行它,当所有任务运行完成后, ApplicationMaster 归还所有资源,并退出。

当前 Spark On YARN 的资源分配仍是粗粒度的即根据指定的 StandaloneExecutorBackend 资源量决定每个 Container 大小,而 ClusterScheduler 则根据指定的 CPU core 个数决定每个 StandaloneExecutorBackend 可同时运行的任务数。

10.3.4 Spark/Storm On YARN 比较

Spark On YARN 和 Storm On YARN 体现了 YARN 的两种不同使用方式,从资源分配粒度上讲,前者可以仿照 Spark On Mesos 实现细粒度资源分配机制^①,这样,它运行的是应用程序,一个应用程序申请自己需要的资源,用完后立刻归还给 YARN,这有利于应用程序间的资源共享;后者则是一种粗粒度的资源分配,它运行的是 Storm 服务,即将 Storm 自动化部署到 YARN 上,使 Storm 集群变成一个弹性实时计算平台(计算资源可根据需要动态伸缩),由于服务启动时通常会将所需要的资源全部占下,以便用户提交应用程序时直接使用,因此是一种粗粒度的资源分配,不利于资源共享。从运行的应用程序类型看,前者运行的是短作业,运行时间通常是分钟或者小时级别的;而后者则是长作业,实际上是服务,通常永远不会终止,除非管理员主动杀死它或者服务故障。

10.4 小结

本章介绍了 Storm 和 Spark 两种计算框架,其中,Storm 用于解决实时计算问题,即数据源源不断地流入系统而被实时分析和处理,常见的应用场景是广告推荐系统、实时日志分析等;Spark 可用于迭代式计算和交互式计算等场景,它基于 RDD 的模型允许用户对访问频繁的数据集进行缓存以减少计算开销。

目前,这两种计算框架均可运行在 YARN 上。随着这两种计算框架的加入,YARN 上的计算框架体系日趋完善,它们拥有不同的应用场景,通过 YARN 将它们运行在一个集群中,可解决多版本运维烦琐、底层数据无法共享、计算资源难以动态扩充等问题。

^① Spark On Mesos 有两种运行模式,分别是“细粒度模式”和“粗粒度模式”,具体参考 <http://spark.incubator.apache.org/docs/latest/running-on-mesos.html>。与 Spark On Mesos 类似,Spark on YARN 也可以同时实现这两种模式,但截至本书出版时,Spark On YARN 仅实现了“粗粒度模式”,另一种模式正在实现中。



第四部分 高级篇

随着计算框架种类的增多和分布式集群管理成本的增加，必然会产生类似于 YARN 的资源管理系统，它们能提供通用的资源管理和调度功能，从而允许将所有应用程序运行在一个集群中，并对它们进行统一管理。目前除了 YARN 之外，还有一些其他类似的开源系统，包括 Facebook Corona 和 Apache Mesos，本书最后一个部分将详细介绍这两个系统并与 YARN 进行对比，总结这类开源系统的特点和发展趋势。

第 11 章 Facebook Corona 剖析

Corona^①是 Facebook 于 2012 年 11 月开源的下一代 MapReduce 框架，它的设计目标与 YARN 类似，在 Facebook 团队的博文^②中如此描述 Corona 的设计目标：

- 更好的可扩展性和集群资源利用率。
- 更小的短作业运行延时。
- 支持在线版本更新。
- 基于真实资源需求进行任务调度。

与 YARN 最大的不同是，Corona 暂时未考虑对多计算框架的支持，也就是说，Corona 的目标限定在 MapReduce 一种计算框架中（尽管在源代码中将 Corona 和 MapReduce 代码分开存放到不同 JAR 包中，但它们还是耦合在一起的）。

在接下来我们将重点介绍 Corona 的基本框架和工作原理。

11.1 概述

同 YARN 设计思想一样，Corona 也采用了 Master/Slave 结构，它们的基本思想也是一致的，即将 JobTracker 拆分成了两个独立的服务：全局的资源管理器 ClusterManager 和每个应用程序特有的 CoronaJobTracker，其中 ClusterManager 负责整个系统的资源管理和分配，而 CoronaJobTracker 负责单个应用程序的管理。

11.1.1 Corona 的基本架构

Corona 基本架构如图 11-1 所示。

Corona 主要由以下几个组件构成：

(1) ClusterManager (CM)

ClusterManager 类似于 YARN 中的 ResourceManager，负责系统资源管理和调度。ClusterManager 掌握着各个节点的资源使用情况，并将资源分配给各个应用程序（采用的调度器为 Fair Scheduler）。此外，考虑到在新架构中，ClusterManager 本身出故障的可能性非常低，它的高可用性通常仅用于在线升级，因此，Corona 在 ClusterManager 高可用性方面设计得非常简单：仅支持人工触发命令将 ClusterManager 状态信息保存到一个镜像文件中，并在升级结束后将其从该文件中恢复。

① 源代码见 <https://github.com/facebook/hadoop-20/tree/master/src/contrib/corona>。

② Under the Hood: Scheduling MapReduce jobs more efficiently with Corona。

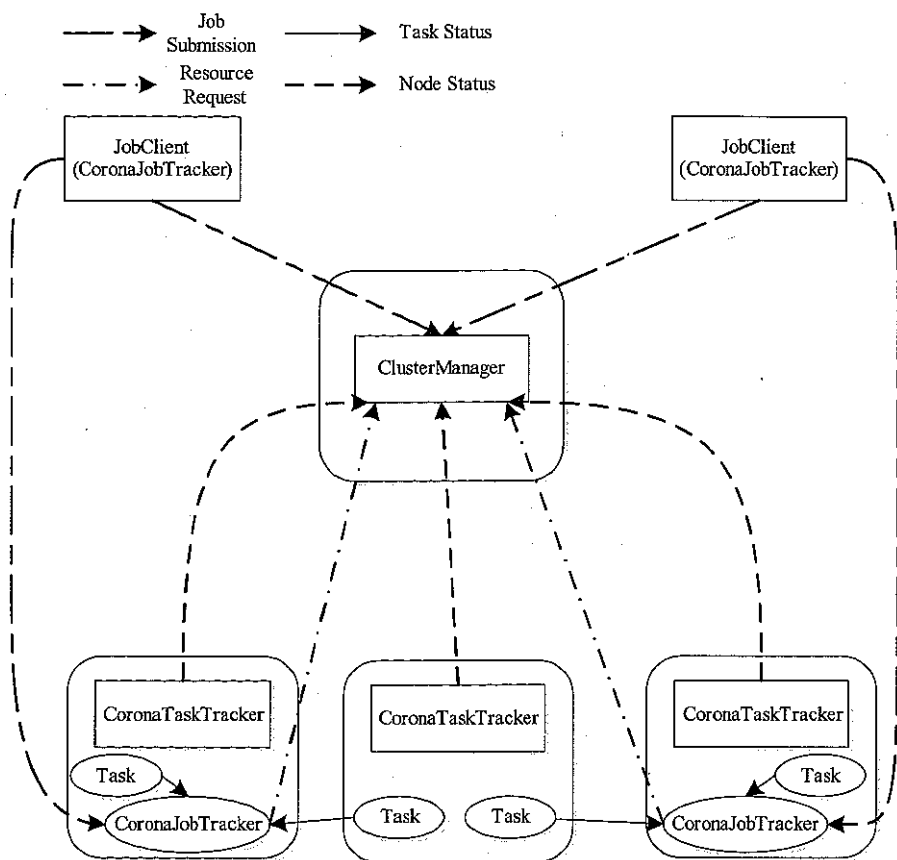


图 11-1 Corona 基本架构

(2) CoronaJobTracker (CJT)

CoronaJobTracker 类似于 YARN 中的 Application Master，用于 MapReduce 应用程序的监控和容错。在 Corona 中，CoronaJobTracker 可以运行在如下三个模式下：

- **同进程模式**：运行在客户端中，该模式能大大降低小作业运行延时。
- **转发模式**：将来自客户端的请求转发给位于远程模式下的 CoronaJobTracker。
- **远程模式**：运行在某个 CoronaTaskTracker 上，它最重要的任务是为作业申请资源和监控作业的运行过程直到它运行结束。

与 MRv1 中的 JobTracker 管理所有作业不同，每个 CoronaJob Tracker 只负责管理一个作业。

(3) CoronaTaskTracker (CTT)

CoronaTaskTracker 功能类似于 YARN 中的 NodeManager，它的实现重用了 MRv1 中 TaskTracker 的很多代码（实际上，CoronaTaskTracker 类继承了 MRv1 的 TaskTracker 类）：一方面，它通过心跳将节点资源使用情况汇报给 ClusterManager；另一方面，它与

CoronaJobTracker 通信, 以获取待运行的任务和汇报正运行任务的状态。

(4) Proxy JobTracker

Proxy JobTracker 是一种离线查看作业运行信息的工具。当一个作业正在运行时, 用户可通过 CoronaJobTracker 提供的 Web 服务查看作业的当前运行情况, 而一旦作业运行完成或者节点处于离线状态时, 用户可通过 Proxy JobTracker 查看作业的详细运行信息, 比如作业的各个任务执行情况、作业 Metrics、作业 Counter 等。

11.1.2 Corona 的 RPC 协议与序列化框架

Facebook Corona 是基于 Hadoop 0.20 版本实现的, 所以 Hadoop 0.20 中的大部分 RPC 协议在 Corona 中仍被采用, 包括客户端与 JobTracker 通信协议 JobSubmissionProtocol、TaskTracker 与 JobTracker 的通信协议 InterTrackerProtocol 和 Task 与 TaskTracker 之间的通信协议 TaskUmbilicalProtocol^①。这几个协议在《Hadoop 技术内幕: 深入解析 MapReduce 架构设计与实现原理》一书中有详细介绍, 有兴趣的读者可阅读了解详细实现。除了在 MRv1 中已存在的这些协议, Corona 又增加了一些其他协议, 如图 11-2 所示, 主要包括:

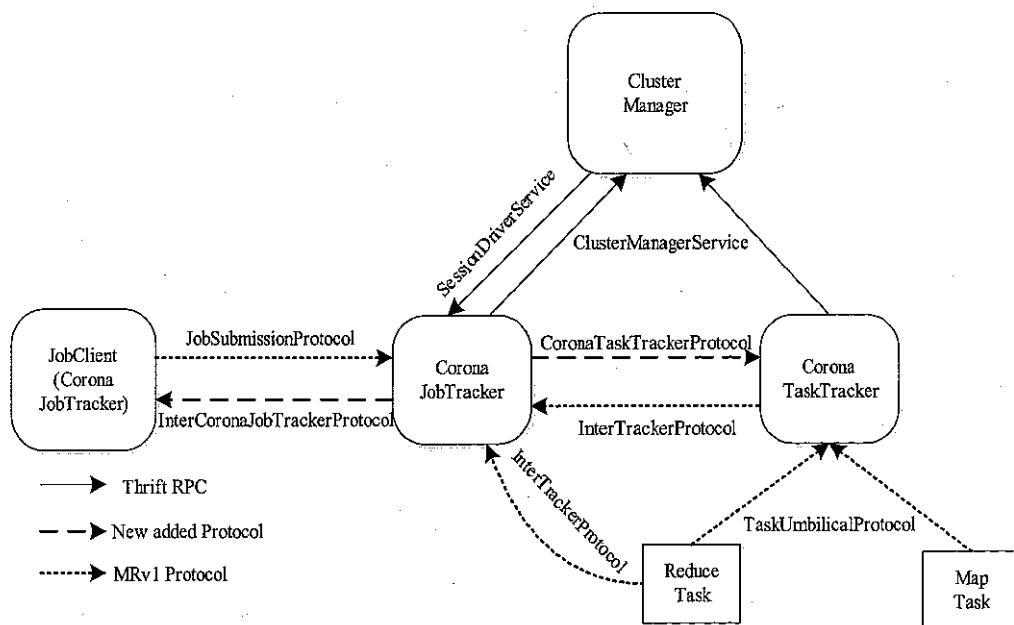


图 11-2 Corona 中的 RPC 协议

- ClusterManagerService：它是采用 Thrift RPC 实现的协议，主要用于 CoronaJobTracker、CoronaTaskTracker 与 ClusterManager 通信，它定义了 CoronaJobTracker 申请和释放资源、CoronaTaskTracker 汇报心跳等 RPC 接口。

① 在 Corona 中, Reduce Task 可以直接与 CoronaJobTracker 通信获取已经运行完成的 Map Task 列表, 而不必通过 CoronaTaskTracker 间接获取。

- SessionDriverService: 该协议也是采用 Thrift RPC 实现的, ClusterManager 通过该协议主动将信息 (如新分配的资源、待释放的资源等) 推送给 CoronaJobTracker (而不是让 CoronaJobTracker 轮询获取这些信息) 以降低延迟。
- CoronaTaskTrackerProtocol: 该协议是采用 Hadoop 自带的 RPC 框架实现的, CoronaJobTracker 可通过该协议主动将命令 (如启动新任务、杀死任务等命令) 推送给 CoronaTaskTracker 以降低延迟 (而不是像 MRv1 那样, TaskTracker 通过心跳周期性拉取 JobTracker 的命令)。
- InterCoronaJobTrackerProtocol: 该协议是采用 Hadoop 自带的 RPC 框架实现的, 远程模式下的 CoronaJobTracker 刚启动时, 将通过该协议向客户端 (父 CoronaJobTracker) 汇报它所在节点的 host 和端口号, 此后它还会周期性向客户端汇报心跳, 以探测父 CoronaJobTracker 是否活着, 一旦发现父 CoronaJobTracker 失败, 则直接退出。

11.2 Corona 设计特点

11.2.1 推式网络通信模型

拉式通信模型如图 11-3 所示, 服务 Service-B 周期性与服务 Service-A 通信, 以拉取新的信息, 然而, Service-A 却无法主动与 Service-B 通信, 只能通过应答的形式将信息发送给对方。MRv1 和 YARN 采用了拉式通信模型, 比如, 在 YARN 中, ApplicationMaster 需通过周期性地心跳从 ResourceManager 上拉取新分配的资源, 而 ResourceManager 无法主动将新分配的资源推送给对应的 ApplicationMaster。这种拉式通信模型优点很明显, 即简单且易实现, 但缺点也很明显, 即时间延迟较大。

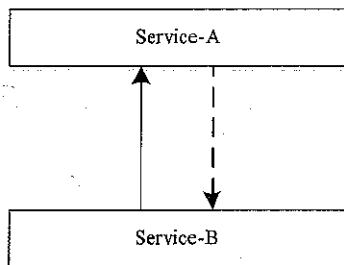


图 11-3 拉式通信模型

为了克服拉式通信模型的这个缺点, Corona 在原有拉式模型基础上又增加了推式模型, 具体如图 11-4 所示。服务 Service-B 不仅能够主动将信息发送给 Service-A, Service-A 也能够主动将信息发送给 Service-B。比如, 在 Corona 中, CoronaJobTracker 可以主动通过 RPC 函数向 ClusterManager 申请资源, ClusterManager 也可以主动将新分配的资源推送给对应的 CoronaJobTracker。

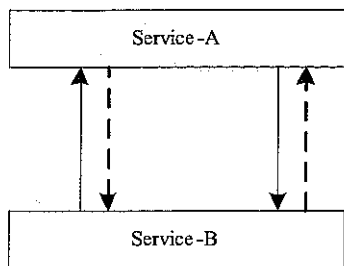


图 11-4 推式通信模型

11.2.2 基于 Hadoop 0.20 版本

Corona 是在 Hadoop 0.20 基础上开发的，它重用了 Hadoop 0.20 的大部分代码，包括 JobTracker、TaskTracker、Task 等，这不同于 YARN，YARN 是一个完全重新设计的系统，它的所有代码是从头开始编写的。

11.2.3 使用 Thrift

Corona 采用了 Facebook 开源的 RPC 框架 Thrift，Thrift 的引入使得开发速度加快且协议之间具有向后兼容性（同 YARN 引入 Protocol Buffers 带来的好处类似）。

11.2.4 深度集成 Fair Scheduler

Fair Scheduler 是 Facebook 开源的 Hadoop 多用户调度器，是 Facebook 内部一直采用的 Hadoop 调度器，而 Corona 则深度集成了该调度器。换句话说，Corona 暂时不支持插拔式调度器，仅支持 Fair Scheduler。

11.3 工作流程介绍

本节介绍 Facebook Corona 的工作流程。同 YARN 类似，当用户提交一个作业后，Facebook Corona 分两个阶段运行该作业。

第一个阶段是作业提交，实际上就是启动 CoronaJobTracker 的过程。根据作业大小不同，Corona JobTracker 采用不同的启动模式。如果一个作业的 Map Task 数目小于一定阈值，CoronaJobTracker 将直接启动在客户端中，这被称为本地启动模式，如图 11-5 所示。很显然，该模式可大大降低小作业运行延迟。另外一种远程启动模式，在该模式下，客户端首先为 CoronaJobTracker（向 Cluster Manager）申请资源，然后在某个 CoronaTaskTracker 上启动 CoronaJobTracker，这种模式将带来一定的运行延迟，本节将重点介绍该启动模式。

第二个阶段是资源申请与任务启动。在该阶段中，CoronaJobTracker 不断向 Cluster-Manager 申请资源，要求 CoronaTaskTracker 启动任务，并监控任务的运行过程，直到所有

任务运行完成。

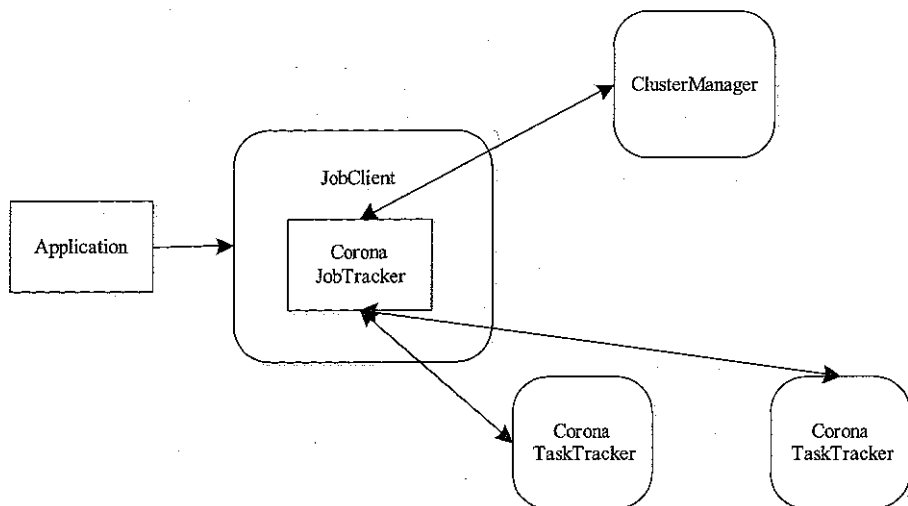


图 11-5 Corona 中的本地启动模式

接下来，我们重点介绍大作业（Map Task 数目大于一定阈值）的提交和运行过程。

11.3.1 作业提交

作业提交过程实际上就是启动 CoronaJobTracker 的过程。为了与 MRv1 兼容，Facebook Corona 仍由 JobClient 提交作业，但里面的代码已修改过。JobClient 可选择是将作业提交到 MRv1 的 JobTracker 上还是 Corona 中。如果提交到 JobTracker 上，则作业运行在 MRv1 中，否则，JobClient 内部会创建一个 CoronaJobTracker 对象，然后由 CoronaJobTracker（运行在转发模式下）负责提交作业。之后的过程如图 11-6 所示。

CoronaJobTracker 的启动大致可分为以下几个步骤：

步骤 1 CoronaJobTracker 内部创建代理对象 RemoteJTPProxy，由它与 ClusterManager 和 CoronaTaskTracker 通信，为 CoronaJobTracker（运行在远程模式下）申请资源并启动它；

步骤 2 RemoteJTPProxy 收到启动 CoronaJobTracker 的请求后，首先需向 ClusterManager 申请资源；

步骤 3 ClusterManager 中的资源调度器为其分配对应的资源量，并返回给 RemoteJTPProxy；

步骤 4 RemoteJTPProxy 根据收到的资源描述信息（包括资源所在节点，资源量等信息），与对应的 CoronaTaskTracker 通信，要求它启动 CoronaJobTracker；

步骤 5 CoronaTaskTracker 成功启动 CoronaJobTracker 后，告诉 RemoteJTPProxy，然后再由 RemoteJTPProxy 告诉客户端，客户端得知 CoronaJobTracker 启动成功后，向 RemoteJTPProxy 提交作业；

步骤 6 RemoteJTProxy 进一步将作业提交到刚刚启动的 CoronaJobTracker 上。至此，一个作业正式提交成功。

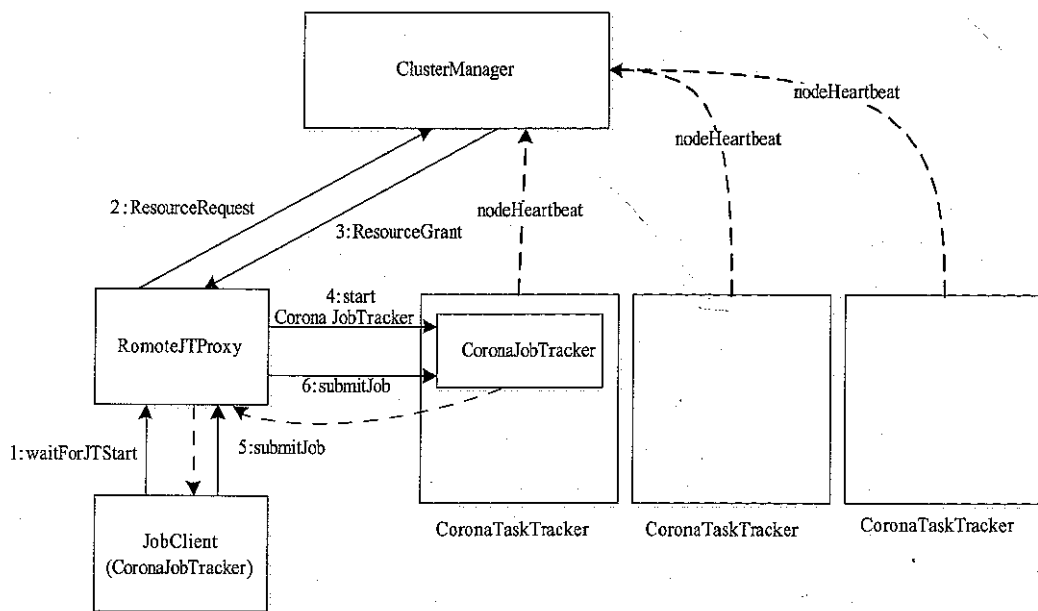


图 11-6 CoronaJobTracker 启动过程

需要注意的是，整个过程涉及两个 CoronaJobTracker：第一个运行在转发模式下，负责提交作业和转发客户端的各种请求；第二个运行在远程模式下，负责申请资源和监控作业运行状态。

11.3.2 资源申请与任务启动

CoronaJobTracker 负责为作业申请资源，并与 CoronaTaskTracker 通信，要求它运行 Task，总之，CoronaJobTracker 资源申请与任务启动过程如图 11-7 所示。

CoronaJobTracker 资源申请与任务启动的主要步骤如下：

步骤 1 CoronaJobTracker 向 ClusterManager 发送资源请求；

步骤 2 当某个 CoronaTaskTracker 出现空闲资源后，ClusterManager 根据调度策略决定将该资源分配给哪些作业，并将资源描述发送给对应的 CoronaJobTracker；

步骤 3 CoronaJobTracker 收到新分配的资源后，与对应的 CoronaTaskTracker 通信，要求它启动任务。

CoronaJobTracker 会重复以上三个步骤，直到所有任务运行完成，此时，CoronaJobTracker 通知 ClusterManager 释放所占用的资源，并退出。至此，一个作业运行完成。

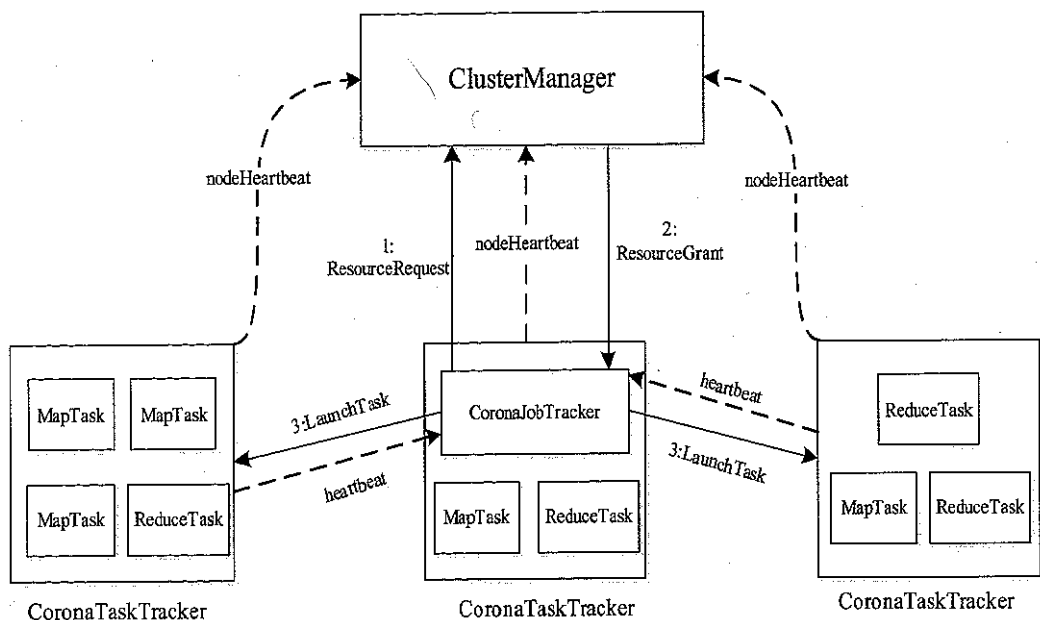


图 11-7 CoronaJobTracker 资源申请与任务启动过程

11.4 主要模块介绍

11.4.1 ClusterManager

在 Corona 中, **ClusterManager** 负责整个集群的资源管理, 包括: ①维护各个节点的资源使用情况, ②将各个节点中的资源按照一定的约束分配 (比如每个 pool 使用的资源不能超过其上线, 任务分配时应考虑负载均衡等) 给各个应用程序。

需要注意的是, **ClusterManager** 是一个纯粹的资源管理器, 它不再 (向 MRv1 中的 **JobTracker** 那样) 负责作业监控相关工作, 如监控各个任务的运行状态, 任务失败时重新启动等, 这由 **CoronaJobTracker** 完成。

ClusterManager 实际上由两部分组成: 节点资源管理器和资源分配模型, 其中, 节点资源管理器维护各个节点的资源变化, 而资源分配模型由 (经修改的) MRv1 中的 **Fair Scheduler** 实现, 但需要注意的是, **ClusterManager** 深度集成了 **Fair Scheduler**, 也就是说, 调度器模块不再可插拔, 它跟 **ClusterManager** 紧紧耦合在一起。

ClusterManager 需要与 **CoronaJobTracker** 和 **CoronaTaskTracker** 通信, 这些都是通过 thrift RPC 实现的, 具体涉及的 RPC 协议, 我已在前几篇文章中进行了介绍, 在此不赘述。为了提高效率, **ClusterManager** 采用了非阻塞异步编程模型, 具体将在下一节中介绍。

ClusterManager 架构如图 11-8 所示:

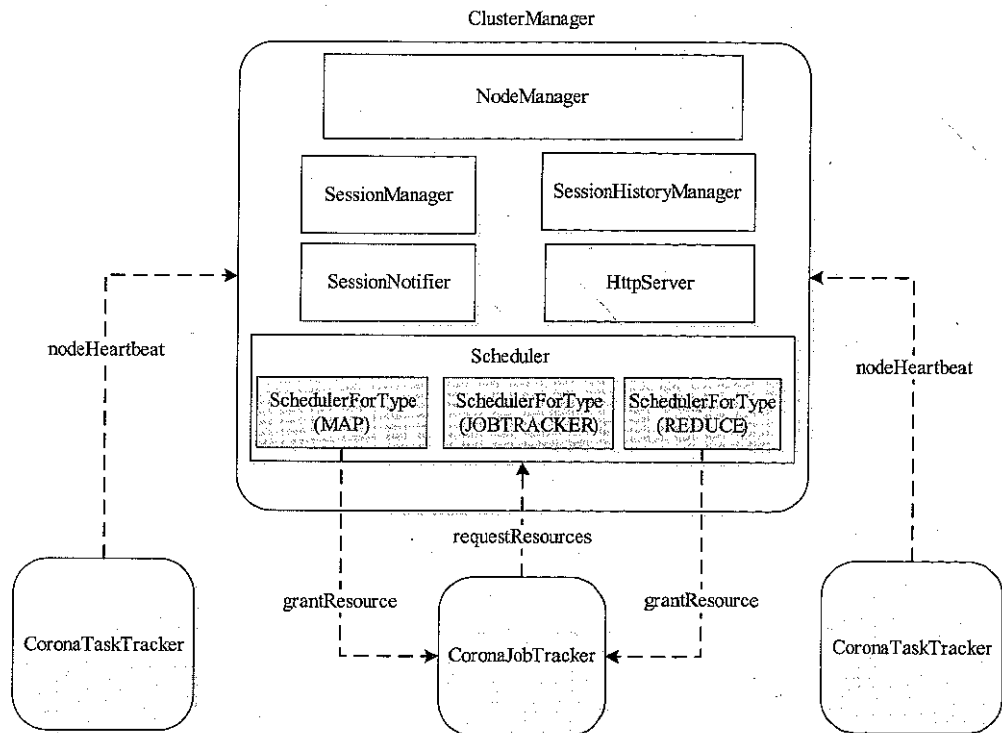


图 11-8 ClusterManager 基本架构

从内部实现角度看，ClusterManager 实现了 ClusterManagerService.Iface 接口，这意味着 ClusterManager 是一个 Thrift Server，为此它需要实现 ClusterManager.thrift 中定义的所有关于 ClusterManagerService 服务的接口，这些接口将被 CoronaJobTracker、CoronaTaskTracker、JobClient 等组件调用。除此之外，ClusterManager 内部还有以下几个组件。

- **NodeManager**：负责管理各个节点上的资源使用情况，当前主要考虑内存、磁盘和 CPU 三种资源。CoronaTaskTracker 通过 thrift RPC 汇报资源使用信息后，ClusterManager 将交由 NodeManager 管理。
- **SessionManager**：负责管理系统中所有的 Session。在 Corona 中，每个作业的生命周期对应一个 Session。当一个作业启动时，会创建一个对应的 Session；当作业运行结束时，会销毁对应的 Session。
- **SessionNotifier**：Session 事件通知器。SessionNotifier 内部包含多个线程，这些线程接收外部的 Session 事件，并采用异步非阻塞的方式通知各个 Session。
- **SessionHistoryManager**：Session 日志管理。当前的实现非常简单，仅提供了获取 Session 日志路径的接口。
- **HttpServer**：对外提供 Web 查询服务的 HTTP Server。它内嵌一个 Jetty Server，允许用户通过 Web 界面查询 ClusterManager 中的一些实时信息。

□ SchedulerForType：资源分配线程，每种资源（Corona 中有三类资源：MAP、REDUCE 和 JOBTRACKER，分别用于启动 Map Task、Reduce Task 和 CoronaJobTracker）一个。当出现空闲资源时，它从当前系统中选择出最合适的作业，并将资源分配它。

前面提到，Corona 已将 Fair Scheduler 深度集成到了 ClusterManager 中，相比于 MRv1 中的 Fair Scheduler，它增加了 group 的概念，即不再只有平级 pool 的概念，而是引入了更高一层的 pool 组织方式——group，管理员可将整个集群资源划分成若干个 group，并可进一步将一个 group 划分成若干个 pool，具体如图 11-9 所示。

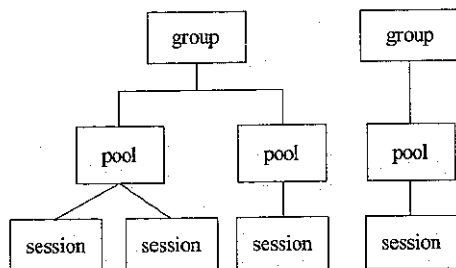


图 11-9 ClusterManager 基本架构

一个 corona.xml 配置实例如下：

```

<?xml version="1.0"?>
<configuration>
  <defaultSchedulingMode>FAIR</defaultSchedulingMode>
  <nodeLocalityWaitMAP>0</nodeLocalityWaitMAP>
  <rackLocalityWaitMAP>5000</rackLocalityWaitMAP>
  <preemptedTaskMaxRunningTime>60000</preemptedTaskMaxRunningTime>
  <shareStarvingRatio>0.9</shareStarvingRatio>
  <starvingTimeForShare>60000</starvingTimeForShare>
  <starvingTimeForMinimum>30000</starvingTimeForMinimum>
  <grantsPerIteration>5000</grantsPerIteration>
  <group name="group_a">
    <minMAP>200</minMAP>
    <minREDUCE>100</minREDUCE>
    <maxMAP>200</maxMAP>
    <maxREDUCE>200</maxREDUCE>
    <pool name="pool_sla">
      <minMAP>100</minMAP>
      <minREDUCE>100</minREDUCE>
      <maxMAP>200</maxMAP>
      <maxREDUCE>200</maxREDUCE>
      <weight>2.0</weight>
      <schedulingMode>FIFO</schedulingMode>
    </pool>
    <pool name="pool_nonsla">
    </pool>
  </group>

```



```

<group name="group_b">
  <maxMAP>200</maxMAP>
  <maxREDUCE>200</maxREDUCE>
  <weight>3.0</weight>
</group>
</configuration>

```

该配置文件中三个参数需要解释：

- **grantsPerIteration**：该参数表示一次性分配的 task 数目上限，默认是 5000，如果你的集群足够小，那么，Corona 可以一次性将集群中所有任务分配到各个节点上。MRv1 中的资源分配是以 TaskTracker 为单位进行的，也就是说，只有当 TaskTracker 汇报心跳请求新任务时，调度器才会为该节点分配任务。而 Corona 则是采用了异步模型批量分配任务，它将汇报心跳和分配任务分开，分别由不同的协议实现，当分配完一批任务后，它会通知一些线程，由这些线程进一步通知各个 CoronaJobTracker，然后再由 CoronaJobTracker 与各个 CoronaTaskTracker 交互，进而启动任务。
- **schedulingMode**：即 pool 内部采用的调度模式。Corona 提供 6 种调度模式，分别是 FAIR (Fair Scheduling)、FIFO (考虑优先级和达到时间)、DEADLINE (基于作业 deadline，用户提交作业时可通过参数“mapred.job.deadline”为作业设置一个时间限制，调度器会优先调度 deadline 最小的作业)、FAIR_PREEMPT (-1*FAIR)、FIFO_PREEMPT (-1*FIFO) 和 DEADLINE_PREEMPT (-1*DEADLINE)。
- **minMAP/minREDUCE/maxMAP/maxREDUCE**：group 或者 pool 中至少保证的资源数目（需要注意，由于 group 包含 pool，因此，每个 group 的参数应不小于其内部各个 pool 对应参数之和）和最多可用的资源数目（注意，是数目，不是百分比）。说到这里，很多读者可能不理解了，前面提到 Corona 是基于真实资源量进行调度的，在此应该配置内存、CPU 等这种资源的使用限制才说的过去，为什么只配置一个数目呢？我个人觉得，这是 Corona 与 MRv1 杂交的结果，Corona 是在 MRv1 基础上实现的，它的资源分配模型掺有 MRv1 的特点，它并不能说是一个纯的，像 YARN 那样基于真实资源量的调度模型。阅读其代码可发现，在 pool 或者 group 内部，一旦分配了一个新任务，则对应的资源使用量会加 1，也就是说，这四个参数实际上是配置的 Task 数目限制！

当 SchedulerForType 为某个作业分配资源后，并不会立即通知对应的 CoronaJobTracker，而是随机交给线程池中的一个 SessionNotifierThread 线程，并由它通过 thrift RPC 通知对应的 CoronaJobTracker。

11.4.2 CoronaJobTracker

CoronaJobTracker 实际上是一个单 Job 版本的 JobTracker^②，它只负责管理一个 Job 的生命周期（不再负责资源管理和调度相关工作），包括该 Job 的创建、并行化、任务

② 可阅读《Hadoop 技术内幕：深入解析 MapReduce 架构设计与实现原理》一书中的第 6 章了解更多细节。

失败时重启、任务运行慢时为其额外启动一个备份任务等。相比于 JobTracker, 由于 CoronaJobTracker 不再负责资源管理 (由 ClusterManager 负责), 故它更加轻量级, 维护的信息更少, 且它会被随机调度到某个 CoronaTaskTracker 上执行。因此, 它不存在 JobTracker 所具有的那些缺点, 比如维护信息过多影响系统扩展性, 一旦失败则整个系统不可用等。

CoronaJobTracker 是在 MRv1 的 JobTracker 基础上修改而来的, 阅读代码可发现, 它仍保持 JobTracker 的基本代码框架, 但代码更加简洁, 毕竟它只需要维护一个作业的运行时信息。此外, 作业的信息维护是由 CoronaJobInProgress 完成的, 它是在 MRv1 的 JobInProgress 类基本上修改而来的, 它的所有代码与 Facebook 版本的 MRv1 中的 JobInProgress 基本一致, 只是对作业的任务获取函数进行了修改 (即函数 ObtainNewXXX()), 改后的函数只需要返回某个 TaskInProgress (这个类未经过修改, 直接重用) 的 Task Attempt 即可。

CoronaJobTracker 最大的修改是 RPC 部分。Corona 将 MRv1 中单向通信模型改为了双向通信模型, 以便实现 push-based 模型以减小作业延时。为此, 需要为每对通信双方额外添加一个通信协议, 进而使每个服务既是 RPC Client, 又是 RPC Server。CoronaJobTracker 与 ClusterManager 之间采用了 Thrift 通信协议, 具体定义在 src\contrib\corona\interface\ClusterManager.thrift 中:

- SessionDriverService: ClusterManager 通过该协议为 CoronaJobTracker 分配资源、回收资源和处理死节点。
- ClusterManagerService: CoronaJobTracker 通过该协议向 ClusterManager 申请资源、释放资源等。

CoronaJobTracker 与 CoronaTaskTracker 之间仍采用原始的 Hadoop RPC 实现, 具体如下:

- InterTrackerProtocol: 与 MRv1 一样, CoronaTaskTracker 通过该协议向 CoronaJobTracker 汇报各个任务的运行状态。
- CoronaTaskTrackerProtocol: CoronaJobTracker 通过该协议向 CoronaTaskTracker 下达各种命令, 比如启动新任务, 杀死任务等。ClusterManager 通过该协议让 CoronaTaskTracker 启动某个 CoronaJobTracker。

CoronaJobTracker 主要组成模块/线程如图 11-10 所示, 下面分别对这几个模块进行介绍。

下面我们对图 11-10 所示的架构进行简单介绍。

- ResourceUpdater: CoronaJobTracker 中的一个独立线程, 每隔 1s 重复更新 Speculative 任务的数目、处理死节点上的任务 (重新为之申请资源)、向 ClusterManager 汇报最新的资源需求和需要释放的资源这三个动作一次。
- interTrackerServer: 实现 InterTrackerProtocol 协议的 RPC Server, 它接收来自 CoronaTaskTracker 的心跳信息, 并根据需要更新任务的运行状态。
- ExpireTasks: CoronaJobTracker 中的一个独立线程, 它包含跟踪新分配的任务是否

成功启动（如果一定时间内未成功启动则会重新分配资源以再次启动它）和跟踪每个任务的运行状态（如果一个任务在一定时间内未汇报进度，则认为它处于僵死状态，需将其杀死）两个功能。

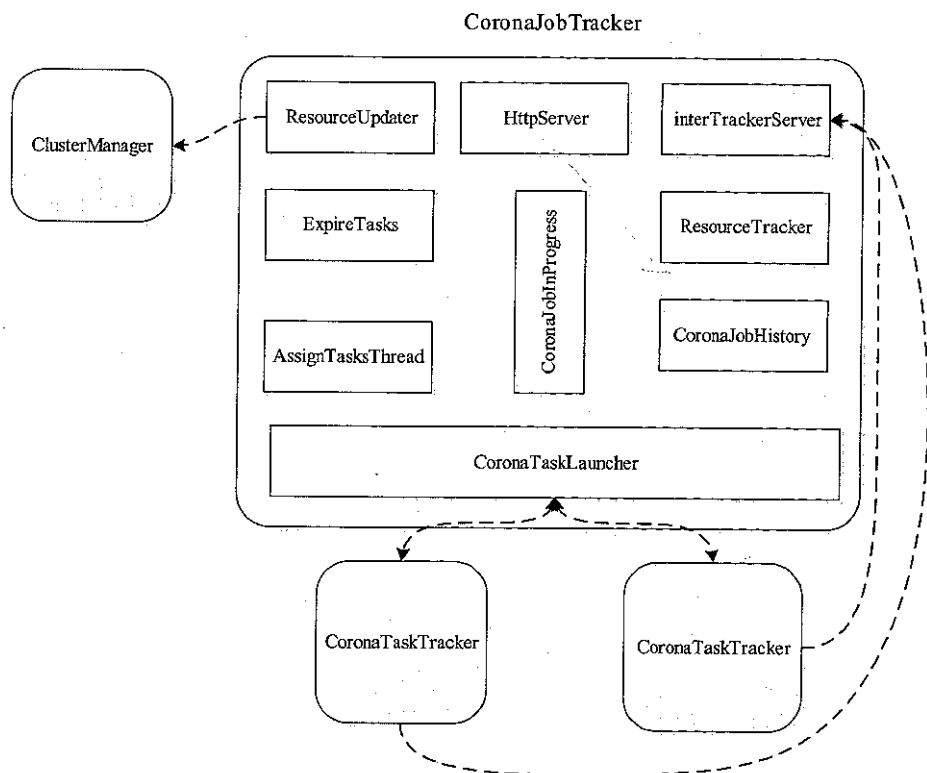


图 11-10 CoronaJobTracker 基本架构

□ ResourceTracker：调整作业所需的资源。在作业启动时，将 Map 和 Reduce 任务所需的资源转化为标准形式，并在任务失败时为其重新计算资源，此外，它还会调整 Speculative 任务所需的资源。在 Corona 中，资源标准表示形式如下：

```
< id, hosts, specs, type, excludeHosts>
```

其中，hosts 为期望资源所在的节点（数据本地性）；specs 为资源描述，当前会传递 4 种资源，分别是内存、网络、CPU 和磁盘，但实际上仅使用了内存、磁盘和 CPU 三种资源；type 为资源类型，当前有 3 种类型，分别是 MAP（用于运行 Map Task）、REDUCE（用于运行 Reduce Task）和 JOBTRACKER（用于运行 CoronaJobTracker）；excludeHosts 为 hosts 黑名单，即分配的资源不能来自这些节点，可能是因为作业在该节点上失败过或者其他原因。默认情况下，每个 Map Task 所需的资源配置为 <1 CPU, 10MBps network, 1024MB Memory, 10GB Disk>，每个 Reduce Task 所需的资源配置为 <1 CPU, 50MBps network, 1024MB Memory, 10GB Disk>，每个 Corona JobTracker 所需的资源配置与 Reduce Task 相同。

- ❑ **AssignTasksThread**：将申请到的资源分配给各个任务，并与对应的 **CoronaTaskTracker** 通信，要求它启动这些新任务。该线程实际上会将启动任务操作授权给另外一个线程 **CoronaTaskLauncher**，让它与 **CoronaTaskTracker** 通信，异步启动 Task。
- ❑ **CoronaJobHistory**：同 MRv1 中的 **JobHistory** 作用一样，即对作业的各种关键性事件（如启动作业，启动任务，任务完成等）做日志，并在需要的时候解析这些日志。
- ❑ **HttpServer**：为用户提供 Web 作业查询功能，通过该 HTTP Server，用户可通过 Web 界面查看作业实时运行信息，包括作业运行状态、各个任务运行状态以及 Counter 值等。
- ❑ **CoronaJobInProgress**：管理作业的运行周期，包括作业的状态、各个任务的状态、进度等信息。与 MRv1 中的 **JobInProgress** 功能基本一致，不同之处是一个 **CoronaJobInProgress** 仅管理一个作业。
- ❑ **CoronaTaskLauncher**：与 **CoronaTaskTracker** 通信，以要求它启动新任务。为了加快任务启动速度，同时防止单个有问题的 **CoronaTaskTracker** 阻塞后面新任务的启动，**CoronaTaskLauncher** 将同时启动多个发送线程与 **CoronaTaskTracker** 通信。

11.4.3 CoronaTaskTracker

CoronaTaskTracker 类似于 MRv1 中的 **TaskTracker**^①，是每个节点上的代理服务，负责向 **ClusterManager** 和 **CoronaJobTracker** 汇报心跳信息，接收来自 **CoronaTaskTracker** 的命令（如 launch task、Kill task 等），并进行处理。其中，向 **ClusterManager** 汇报的心跳信息包括：节点总的资源量（包括磁盘、内存、CPU 和网络四种资源）和空闲资源量；向 **CoronaJobTracker** 汇报的心跳信息（包括各个任务的运行状态）。

CoronaTaskTracker 是在原有的 **TaskTracker** 基础上实现的，它继承了 MRv1 中的 **TaskTracker**，以便重用它里面的大部分函数，包括各种 Action 的处理函数等。

CoronaTaskTracker 仍然通过 **InterTrackerProtocol** 协议向 **CoronaJobTracker** 发送任务运行信息，但是，**CoronaJobTracker** 不再通过心跳应答的（被动）方式为之发送各种 Action，而是改为通过另外一个 RPC 协议主动发送以缩短响应时间。此外，由于每个应用程序对应一个 **CoronaJobTracker**，因此，每个 **CoronaTaskTracker** 需要同时往多个 **CoronaJobTracker** 上发送心跳信息。

为了支持 **CoronaJobTracker** 主动向 **CoronaTaskTracker** 发送 Action，**CoronaTaskTracker** 实现了一个新的接口协议 **CoronaTaskTrackerProtocol**，它包含两个 RPC 函数：

```
public void submitActions(TaskTrackerAction[] actions); // 处理各种 Action
public void startCoronaJobTracker(Task jobTask, CoronaSessionInfo info);
// 启动 CoronaJobTracker
```

此外，**CoronaTaskTracker** 周期性地向 **ClusterManager** 汇报自己的资源信息，包括总资源量和空闲资源量等。

① 可阅读《Hadoop 技术内幕：深入解析 MapReduce 架构设计与实现原理》一书中的第 7 章了解更多细节。

如果你已经非常了解 MRv1 中的 TaskTracker 架构, 那么经过前面的介绍, 大家应该已经大体知道了 CoronaTaskTracker 的架构, 如图 11-11 所示。

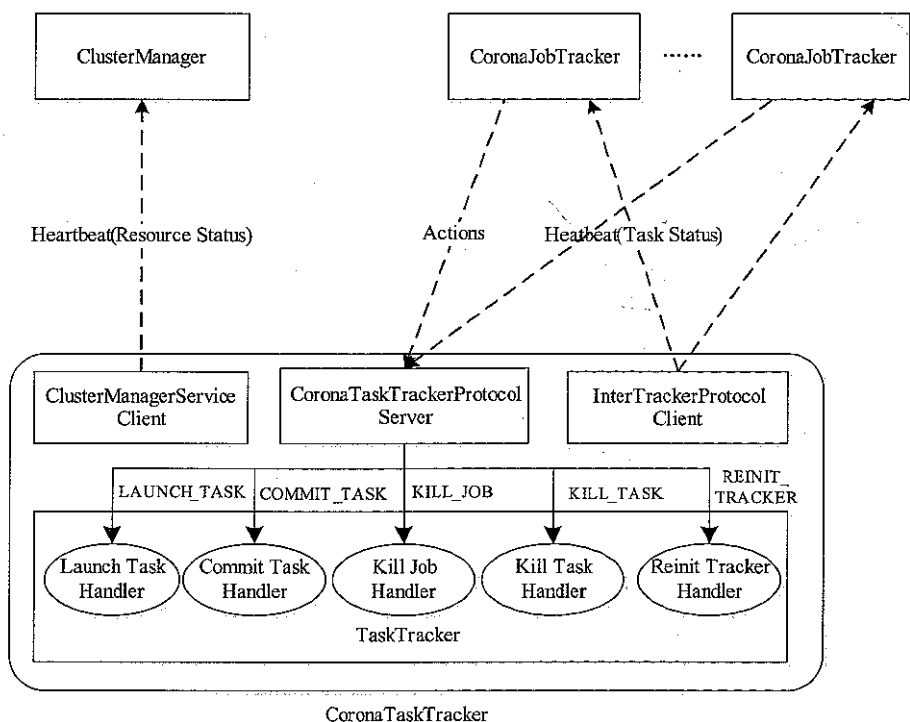


图 11-11 CoronaTaskTracker 基本架构

CoronaTaskTracker 有三个对外通道, 分别对应 RPC 协议 InterTrackerProtocol、CoronaTaskTrackerProtocol 和 ClusterManagerService, 下面依次进行介绍。

- **InterTrackerProtocol 协议**: CoronaTaskTracker 通过该协议向各个 CoronaJobTracker 汇报与其对应的任务的运行状态, 但需要注意的是, 与 MRv1 中的 TaskTracker 不同, CoronaJobTracker 不会通过该协议向对应的 TaskTracker 发送各种命令。
- **CoronaTaskTrackerProtocol 协议**: CoronaJobTracker 通过该协议向 CoronaTaskTracker 发送各种命令, 同 MRv1 一样, 包括 (或者称为 Action) LAUNCH_TASK、COMMIT_TASK、KILL_JOB、KILL_TASK、REINIT_TRACKER 等命令。
- **ClusterManagerService 协议 (采用 Thrift 实现)**: CoronaTaskTracker 通过该协议向 ClusterManager 汇报本节点上的资源情况, 包括总的资源量和剩余资源量。当前会汇报四种资源, 包括 CPU、磁盘、网络带宽和内存。需要注意的是, 截至本书截稿时, 网络带宽被设为常量 100, 而调度机制仅考虑了内存资源。

除了以上三个 RPC 协议, CoronaTaskTracker 最核心的部分是对各种 Action 的处理函数, 它本身未对这些进行修改, 完全重用了 MRv1 中 TaskTracker 的实现。

11.5 小结

本章主要分析了 Facebook 的资源管理系统 Corona，包括它的基本框架和工作原理。

Corona 主要由 ClusterManager、CoronaJobTracker 和 CoronaTaskTracker 三部分组成，其中 ClusterManager 负责整个系统的资源管理和分配，CoronaJobTracker 则负责单个应用程序的管理，CoronaTaskTracker 负责单个节点的任务监控和资源汇报。Corona 混合使用了 Hadoop RPC 和 Thrift RPC 进行设计。

Corona 的主要设计特点包括采用了推式网络通信模型、基于 Hadoop 0.20 版本进行开发、采用了乐观锁机制、使用 Thrift 和深度集成 Fair Scheduler 等。

除了 Corona 的基本框架和设计特点，本章还介绍了 Corona 的工作流程和主要模块的设计细节。

第 12 章 Apache Mesos 剖析

Mesos[Ⓐ]是诞生于加州大学伯克利分校（UC Berkeley）的一个研究项目，它的设计动机是解决编程模型和计算框架在多样化环境下，不同框架间的资源隔离和共享问题。尽管它的直接设计动机与 MRv1 无关，但它的架构和实现策略与 YARN 或者 Corona 类似，因此，本章将对其进行介绍。当前有一些公司正在使用 Mesos 管理集群资源，比如国外的 Twitter、国内的豆瓣[Ⓑ]等。

12.1 概述

如图 12-1 所示，Apache Mesos 由以下四个组件组成，接下来我们将详细对这些组件进行介绍。

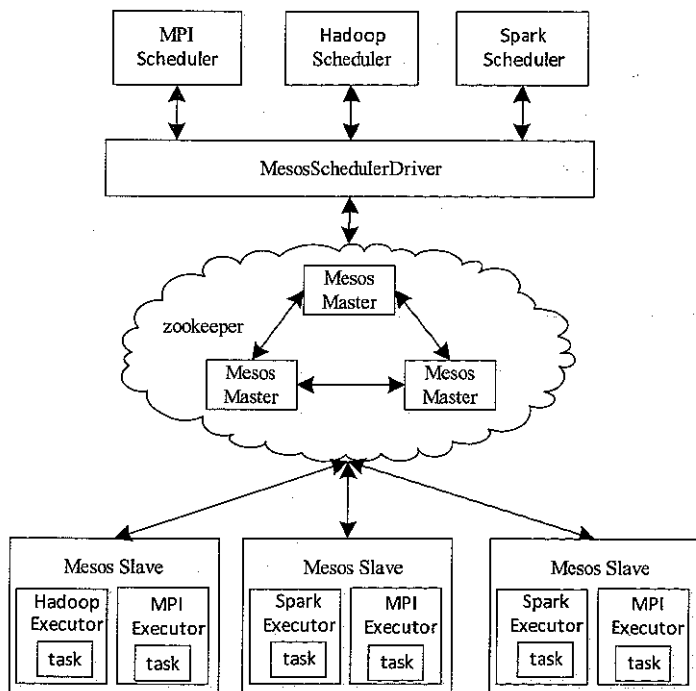


图 12-1 Mesos 的基本架构

Ⓐ Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R. Katz, S. Shenker and I. Stoica, NSDI 2011, March 2011.

Ⓑ <https://github.com/douban/dpark/>.

- Mesos Master : Mesos Master 是整个系统的核心, 负责管理整个系统中所有资源和接入 Mesos 的各种框架 (Framework), 并将 Mesos Slave 上的资源按照某种策略分配给框架。为了防止 Mesos Master 出现故障后导致集群不可用, Mesos 允许用户配置多个 Mesos Master, 并通过 Zookeeper 进行管理, 当主 Mesos Master 出现故障后, Zookeeper 可马上为之选择一个新的主 Mesos Master。
- Mesos Slave : Mesos Slave 负责接收并执行来自 Mesos Master 的命令, 并定时将任务执行状态汇报给 Mesos Master。Mesos Slave 将节点上资源使用情况发送给 Mesos Master, 由 Mesos Master 中的 Allocator 模块决定将资源分配给哪个 Framework。需要注意的是, 当前 Mesos 仅考虑了 CPU 和内存两种资源。为了避免任务之间相互干扰, 同 YARN 一样, Mesos Slave 采用了轻量级资源隔离机制 Cgroups。
- Framework Scheduler : Framework 是指外部的框架, 如 MPI、MapReduce、Spark 等, 这些框架可通过注册的方式接入 Mesos, 以便 Mesos 进行统一管理和资源分配。Mesos 要求接入的框架必须有一个调度器模块 Framework Scheduler, 该调度器负责框架内部的任务调度。一个 Framework 在 Mesos 上工作流程为: 首先通过自己的调度器向 Mesos 注册, 并获取 Mesos 分配给自己的资源; 然后再由自己的调度器将这些资源分配给框架中的任务。也就是说, 同 YARN 一样, Mesos 系统采用了双层调度框架: 第一层, 由 Mesos 将资源分配给框架; 第二层, 框架自己的调度器将资源分配给内部的各个任务。当前 Mesos 支持三种语言编写的调度器, 分别是 C++、Java 和 Python, 为了向各种调度器提供统一的接入方式, Mesos 内部采用 C++ 实现了一个 MesosSchedulerDriver (调度器驱动器), Framework 的调度器可调用该 Driver 中的接口与 Mesos Master 交互, 完成一系列功能 (如注册、资源分配等)。
- Framework Executor: Framework Executor 主要用于启动框架内部的任务。由于框架的不同, 启动任务的接口或者方式也不同, 当一个新的框架要接入 Mesos 时, 需要编写一个对应的 Executor, 告诉 Mesos 如何启动该框架中的任务。为了给各种框架提供统一的执行器编写方式, Mesos 内部采用 C++ 实现了一个 MesosExecutorDriver (执行器驱动器), Framework 可通过该驱动器的相关接口告诉 Mesos 启动任务的方法。

12.2 底层网络通信库

libprocess^①是一套基于 Socket 实现的通信协议库, 它采用了 Protocol Buffers 序列化框架, 通过结合两者实现了一套很高效的基于消息传递的底层网络通信库, 而 Mesos 底层通信协议正是采用了该库。

① <https://github.com/3rdparty/libprocess>.

12.2.1 libprocess 基本架构

libprocess 采用了基于消息传递的网络通信模型，每一个服务（进程）内部实际上运行了一个 Socket Server，而不同服务之间通过消息（事件）进行通信。在每个服务内部，注册了很多类型的消息，而每种消息对应的一个处理器，一旦它收到某种类型的消息，就会调用相应的处理器进行处理。在处理过程中，可能会产生另外一种消息发送给另一个服务。整个过程如图 12-2 所示。

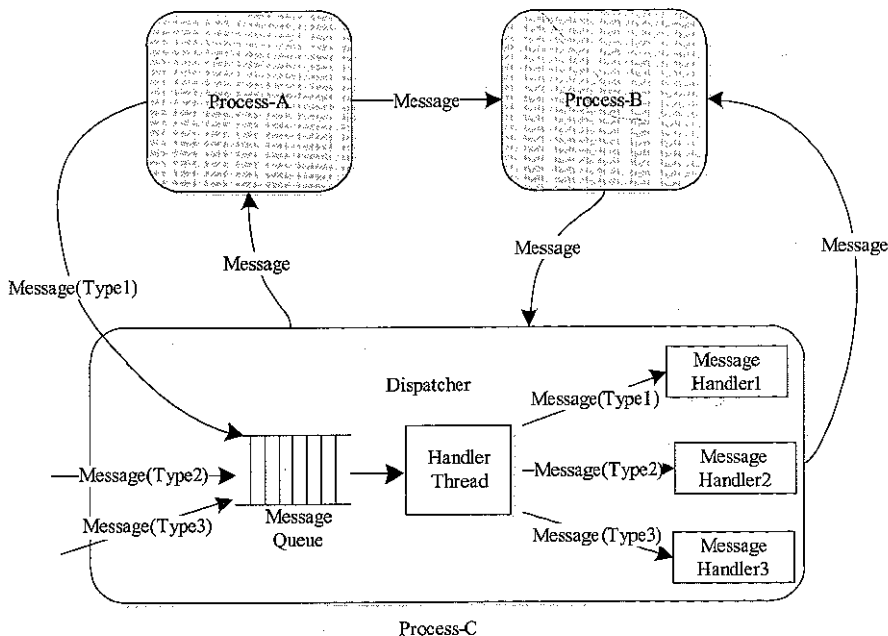


图 12-2 libprocess 中的消息传递机制

12.2.2 一个简单示例

假设有两个服务 Master 和 Slave，Slave 周期性向 Master 汇报自己的进度，而 Master 则不定期地向 Slave 下达任务，采用 libprocess 实现如下。

(1) Master 类设计

步骤 1 继承 ProtobufProcess 类。让 Master 类继承 libprocess 中的 ProtobufProcess 类，该类实际上维护了一个 Socket Server，该 Server 可以处理实现注册好的各种 Protocol Buffers 定义的 Message。

```
class Master : public ProtobufProcess<Master> {
//...
}
```

步骤 2 注册消息处理器。在初始化函数 initialize() 中（使用 install 函数，代码见下

面)注册一个 ReportProgressMessage 类型(需使用 Protocol Buffers 定义)的消息处理器,该消息处理器对应的函数是 Master::reportProgress,该函数带有一个参数 tasks。这样,Master 内部的 Socket Server 会监听来自外部的各种消息包,一旦发现 ReportProgressMessage 类型的消息包,则会调用函数 reportProgress 进行处理。注意,reportProgress 函数中的参数必须正好对应 ReportProgressMessage 消息定义。

```
void initialize() { // 继承 ProtobufProcess 类后,需实现该函数
    install<ReportProgressMessage>{ // 使用 install 函数进行注册
        &Master::reportProgress,
        &LaunchTasksMessage::tasks);
    }
```

步骤 3 定义 ReportProgressMessage 消息。使用 Protocol Buffers 定义 ReportProgressMessage 消息如下:

```
//master.proto
message ReportProgressMessage {
    repeated Task tasks = 3;
}

message Task {
    required string name = 1;
    required TaskID task_id = 2;
    required float progress = 3;
}

message TaskID {
    required string value = 1;
}
```

步骤 4 编写消息处理器 reportProgress。代码如下:

```
void reportProgress(const vector<Task>& tasks) {
    for (int i = 0; i < tasks.size(); i++) {
        //update tasks[i] information;
    }
    ...
}
```

步骤 5 编写 main 函数启动 Master。代码如下:

```
int main(int argc, char** argv)
{
    process::initialize("master"); // 初始化一个名为 Master 的进程
    Master* master = new Master();
    process::spawn(master); // 启动 Master,实际上是一个 Socket Server
    process::wait(master->self());
    delete master;
    return 0;
}
```

Master 主要代码如下:

```
class Master : public ProtobufProcess<Master> // 步骤 1
{
    Master(): ProcessBase("master") {}
    void initialize() {
        install<ReportProgressMessage>( // 步骤 2
            &Master::reportProgress,
            &LaunchTasksMessage::tasks);
    }

    void reportProgress(const vector<Task>& tasks) { // 步骤 4
        for (int i = 0; i < tasks.size(); i++) {
            //update tasks[i] information;
        }
    }
}
```

(2) Slave 类设计

Slave 设计与 Master 类似, 具体如下:

```
class Slave : public ProtobufProcess<Slave>
{
    Slave(): ProcessBase("slave") {}
    void initialize() {
        install<LaunchTasksMessage>(
            &Master::launchTasks,
            &LaunchTasksMessage::id,
            &LaunchTasksMessage::tasks);
    }

    void launchTasks(const int id,
                     const vector<TaskInfo>& tasks) {
        for (int i = 0; i < tasks.size(); i++) {
            //launch tasks[i];
        }
    }
}
```

12.3 Mesos 服务

Mesos 中包含四类主要的服务 (实际上是一个 Socket Server), 分别是 Mesos Master、Mesos Slave、SchedulerProcess 和 ExecutorProcess, 它们之间通过 Protocol Buffers 消息进行通信, 每种服务内部注册了若干种 Protocol Buffers 消息及其对应的消息处理器, 一旦收到某种消息, 则会调用相应的消息处理器进行处理。

除了以上四种服务之外, Mesos 还对外提供了三种可编程组件, 分别是 Allocator、Framework Scheduler 和 Framework Executor, 编写这几个组件必须按照要求实现了几个接

口, 这些接口将分别被 Mesos Master、SchedulerProcess 和 ExecutorProcess 中的事件处理器调用。Mesos 主要服务组件如图 12-3 所示。

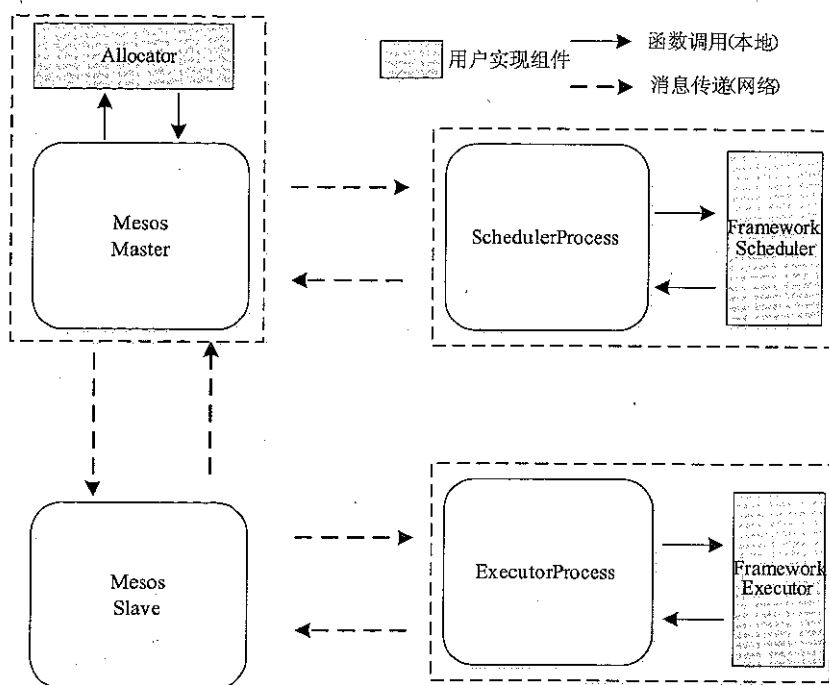


图 12-3 Mesos 主要服务组件

12.3.1 SchedulerProcess

SchedulerProcess 是 Framework Scheduler 与 Mesos Master 之间的通信代理服务, 它将来自 Mesos Master 的 Protocol Buffers 消息传达给 Framework Scheduler, 同时将 Framework Scheduler 的请求传达给 Mesos Master。表 12-1 给出了 SchedulerProcess 内部处理的各种 Protocol Buffers 消息, 以及对应的处理逻辑。

表 12-1 SchedulerProcess 基本信息

Protocol Buffers 消息类型	消息来源	调用 Framework Scheduler 中的函数	解 释
NewMasterDetectedMessage	MasterDetector	—	检测到可用的 Mesos Master, SchedulerProcess 会向 Mesos Master 发送 RegisterFrameworkMessage 消息
NoMasterDetectedMessage	MasterDetector	disconnected	未检测到可用的 Mesos Master
FrameworkRegisteredMessage	Mesos Master	registered	Framework 向 Mesos Master 注册成功
FrameworkReregisteredMessage	Mesos Master	reregistered	Framework 丢失, 恢复正常后重新注册成功

(续)

Protocol Buffers 消息类型	消息来源	调用 Framework Scheduler 中的函数	解 释
ResourceOffersMessage	Mesos Master	resourceOffers	为 Framework 分配资源
RescindResourceOfferMessage	Mesos Master	offerRescinded	杀死任务
StatusUpdateMessage	Mesos Master	statusUpdate	任务状态更新
LostSlaveMessage	Mesos Master	slaveLost	Mesos Slave 宕掉 (超时)
- (未处理, 见 Master:: exited-Executor)	Mesos Master	executorLost	Executor 宕掉
ExecutorToFrameworkMessage	Mesos Master	frameworkMessage	Executor 向 Framework 发送的消息 (通过 Mesos Master 中转)
FrameworkErrorMessage	Mesos Master	error	Framework 出错退出

12.3.2 Mesos Master

Mesos Master 负责整个集群的资源管理和调度, 它处理的消息可能来自 MasterDetector、Mesos Slave 和 Framework Scheduler 等组件。表 12-2 给出了 Mesos Master 内部处理的各种 Protocol Buffers 消息, 以及对应的处理逻辑。

表 12-2 Mesos Master 基本信息

Protocol Buffers 消息类型	消息来源	消息处理函数	解 释
SubmitSchedulerRequest	mesos/main.cpp	submitScheduler	(尚未使用)
NewMasterDetectedMessage	MasterDetector	newMasterDetected	检测到新的 Mesos Master
NoMasterDetectedMessage	MasterDetector	noMasterDetected	未检测到新的 Mesos Master
RegisterFrameworkMessage	SchedulerProcess	registerFramework	Framework 相关操作, 包括注册、重新注册、退出和失效 (暂时不接受新消息)
ReregisterFrameworkMessage	SchedulerProcess	reregisterFramework	
UnregisterFrameworkMessage	SchedulerProcess	unregisterFramework	
DeactivateFrameworkMessage	SchedulerProcess	deactivateFramework	
ResourceRequestMessage	SchedulerProcess	resourceRequest	Framework 向 Master 请求资源
LaunchTasksMessage	SchedulerProcess	launchTasks	请求启动任务
ReviveOffersMessage	SchedulerProcess	reviveOffers	请求清除 Framework 之前设置的资源过滤器
KillTaskMessage	SchedulerProcess	killTask	请求杀死任务
FrameworkToExecutorMessage	SchedulerProcess	schedulerMessage	Framework 向 Executor 发送消息
RegisterSlaveMessage	Mesos Slave	registerSlave	Mesos Slave 相关操作, 包括注册、重新注册和退出
ReregisterSlaveMessage	Mesos Slave	reregisterSlave	
UnregisterSlaveMessage	Mesos Slave	unregisterSlave	
StatusUpdateMessage	Mesos Slave	statusUpdate	任务状态更新
ExecutorToFrameworkMessage	Mesos Slave	executorMessage	来自某个 Executor 发送给 Framework 的消息 (需转发给对应的 Framework)
ExitedExecutorMessage	Mesos Slave	exitedExecutor	Executor 退出

12.3.3 Mesos Slave

Mesos Slave 负责单个节点上的资源和任务管理，它处理的消息可能来自 MasterDetector、Mesos Master 和 Framework Executor 等组件。表 12-3 给出了 Mesos Slave 内部处理的各种 Protocol Buffers 消息，以及对应的处理逻辑。

表 12-3 Mesos Slave 基本信息

Protocol Buffers 消息类型	消息来源	消息处理函数	解 释
NewMasterDetectedMessage	MasterDetector	newMasterDetected	检测 到 新 的 Mesos Master
NoMasterDetectedMessage	MasterDetector	noMasterDetected	未 检 测 到 新 的 Mesos Master
SlaveRegisteredMessage	Mesos Master	registered	Mesos Slave 向 Mesos Master 注册
SlaveReregisteredMessage	Mesos Master	reregistered	Mesos Slave 丢失，恢复正常后重新注册
RunTaskMessage	Mesos Master	runTask	运行任务
KillTaskMessage	Mesos Master	killTask	杀死任务
ShutdownFrameworkMessage	Mesos Master	shutdownFramework	关闭 Framework
FrameworkToExecutorMessage	Mesos Master	schedulerMessage	Framework 向 Executor 发送消息
StatusUpdateAcknowledgement Message	SchedulerProcess	statusUpdateAcknowledgement	确认收到任务状态更新消息
UpdateFrameworkMessage	Mesos Master	updateFramework	更新 Framework 信息
RegisterExecutorMessage	ExecutorProcess	registerExecutor	Executor 注册
StatusUpdateMessage	ExecutorProcess	statusUpdate	状态更新
ExecutorToFrameworkMessage	ExecutorProcess	executorMessage	该 Mesos Slave 上的某个 Executor 向 Framework 发送消息

12.3.4 ExecutorProcess

ExecutorProcess 是 Framework Executor 与 Mesos Slave 之间的通信代理服务，它将来自 Mesos Slave 的 Protocol Buffers 消息传达给 Framework Executor，同时将 Framework Executor 的请求传达给 Mesos Slave。表 12-4 给出了 ExecutorProcess 内部处理的各种 Protocol Buffers 消息，以及对应的处理逻辑。

表 12-4 ExecutorProcess 基本信息

Protocol Buffers 消息类型	消息来源	调用 Framework Executor 中的函数	解 释
ExecutorRegisteredMessage	Mesos Slave	registered	Executor 注册成功
RunTaskMessage	Mesos Slave	launchTask	运行某个任务

(续)

Protocol Buffers 消息类型	消息来源	调用 Framework Executor 中的函数	解 释
KillTaskMessage	Mesos Slave	killTask	杀死任务
FrameworkToExecutorMessage	Mesos Slave	frameworkMessage	Framework 向 Executor 发送的消息
ShutdownExecutorMessage	Mesos Slave	shutdown	Executor 关闭

12.4 Mesos 工作流程

为了让读者更深入地了解 Mesos 内部各种行为的逻辑, 本节选取了非常重要的几种行为进行分析, 包括框架注册、Framework Executor 注册、资源分配、任务启动、任务状态更新等。

12.4.1 框架注册过程

一个外部框架的是通过一个 Framework Scheduler 接入 Mesos 中的, 接入过程被称为注册, 如图 12-4 所示。

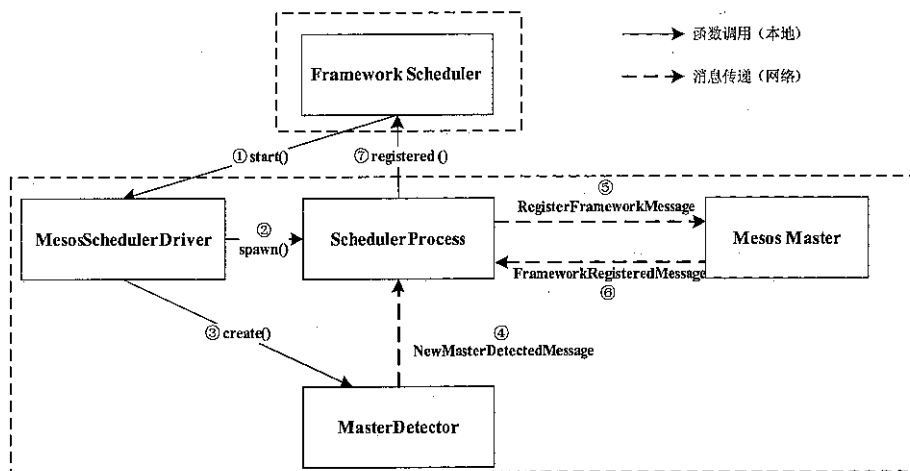


图 12-4 框架注册过程

框架注册的具体步骤如下:

- 步骤 1 用户编写的 Framework 启动时, 创建一个 MesosSchedulerDriver 对象;
- 步骤 2 初始化 MesosSchedulerDriver, 启动一个 SchedulerProcess 服务;
- 步骤 3 MesosSchedulerDriver 创建一个 MasterDetector 对象;
- 步骤 4 MasterDetector 对象探测到正常的 Mesos Master, 向 SchedulerProcess 服务发送一个 NewMasterDetectedMessage 消息;
- 步骤 5 SchedulerProcess 向 Mesos Master 发送一个 RegisterFrameworkMessage 消息,

进而向 Mesos Master 注册该 Framework;

步骤 6 Mesos Master 向 SchedulerProcess 返回 FrameworkRegisteredMessage 消息, 确认 Framework 注册成功;

步骤 7 SchedulerProcess 调用 Scheduler.registered(), 通知 Framework Scheduler 注册成功。

12.4.2 Framework Executor 注册过程

对于任意一个 Mesos Slave, 当它收到来自某个 Framework 的第一个任务时, 在启动该任务之前, 将首先启动 Framework Executor, 这样接下来所有任务均会由该 Executor 启动。Framework Executor 启动过程实际上就是向 Mesos Slave 注册的过程, 如图 12-5 所示。

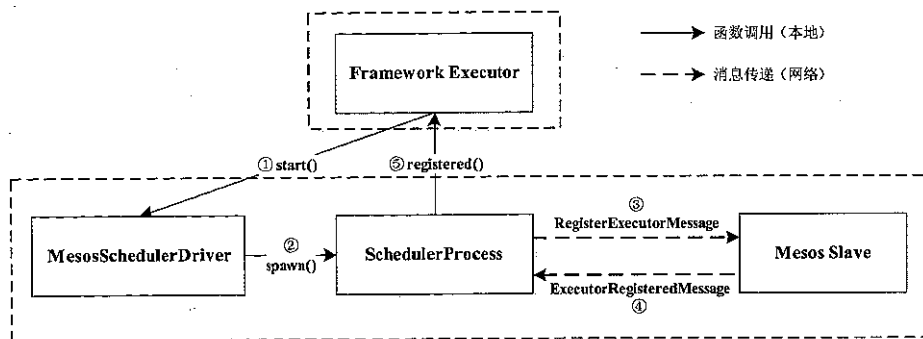


图 12-5 Framework Executor 注册过程

Framework Executor 注册的主要步骤如下:

步骤 1 用户编写的 Framework Executor 启动时, 创建一个 MesosExecutorDriver 对象;

步骤 2 初始化 MesosExecutorDriver, 启动一个 ExecutorProcess 服务;

步骤 3 ExecutorProcess 向 Mesos Slave 发送一个 RegisterExecutorMessage 消息, 从而向 Mesos Slave 注册 Framework Executor;

步骤 4 Mesos Slave 向 ExecutorProcess 返回 ExecutorRegisteredMessage 消息, 确认 Framework Executor 注册成功;

步骤 5 ExecutorProcess 调用 Executor.registered(), 通知 Framework Executor 注册成功。

12.4.3 资源分配到任务运行过程

Mesos Master 最重要的功能之一是资源分配, 它的资源分配功能是通过插拔式组件 Allocator 实现的, 如图 12-6 所示。

在 Mesos 中, 从资源分配到任务运行所经历的步骤有如下几个:

步骤 1 当出现以下几种事件中的一种时, 会触发资源分配行为: 新框架注册、框架注销、增加节点、出现空闲资源等。

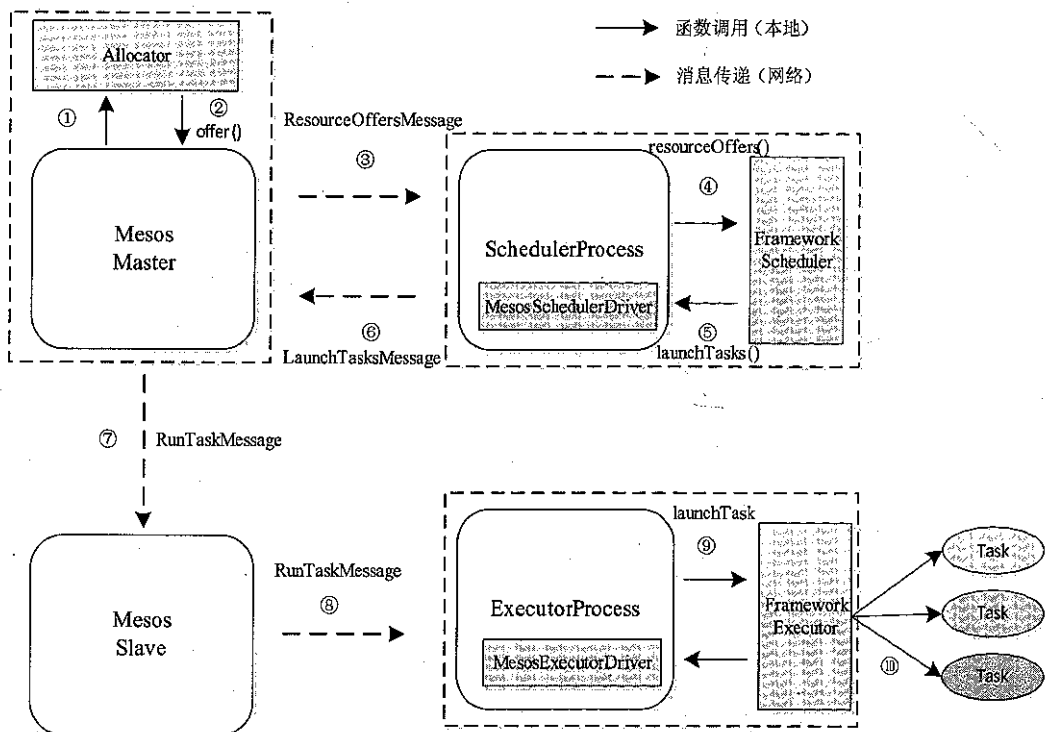


图 12-6 资源分配到任务运行过程

步骤 2 Mesos Master 中的 Allocator 模块为某个框架分配资源，并将资源封装到 ResourceOffersMessage (Protocol Buffers Message) 中，通过网络传输给 SchedulerProcess。

步骤 3 SchedulerProcess 调用用户编写的 Scheduler 中的 resourceOffers 函数 (部分版本可能有变动)，告之有新资源可用。

步骤 4 用户的 Scheduler 调用 MesosSchedulerDriver 中的 launchTasks() 函数，告之将要启动的任务。

步骤 5 SchedulerProcess 将待启动的任务封装到 LaunchTasksMessage (Protocol Buffers Message) 中，通过网络传输给 Mesos Master。

步骤 6 Mesos Master 将待启动的任务封装成 RunTaskMessage 发送给各个 Mesos Slave。

步骤 7 Mesos Slave 收到 RunTaskMessage 消息后，将之进一步发送给对应的 ExecutorProcess。

步骤 8 ExecutorProcess 收到消息后，进行资源本地化，并准备任务运行环境，最终调用用户编写的 Executor 中的 launchTask 启动任务 (如果 Executor 尚未启动，则先要启动 Executor)。

12.4.4 任务启动过程

Mesos Slave 收到来自 Mesos Master 的启动任务命令后，将先检查该 Framework 的 Framework Executor 是否已经启动，如果尚未启动，则首先启动它，然后再启动任务，否则，直接向 ExecutorProcess 发送任务启动命令以启动任务，整个过程如图 12-7 所示。

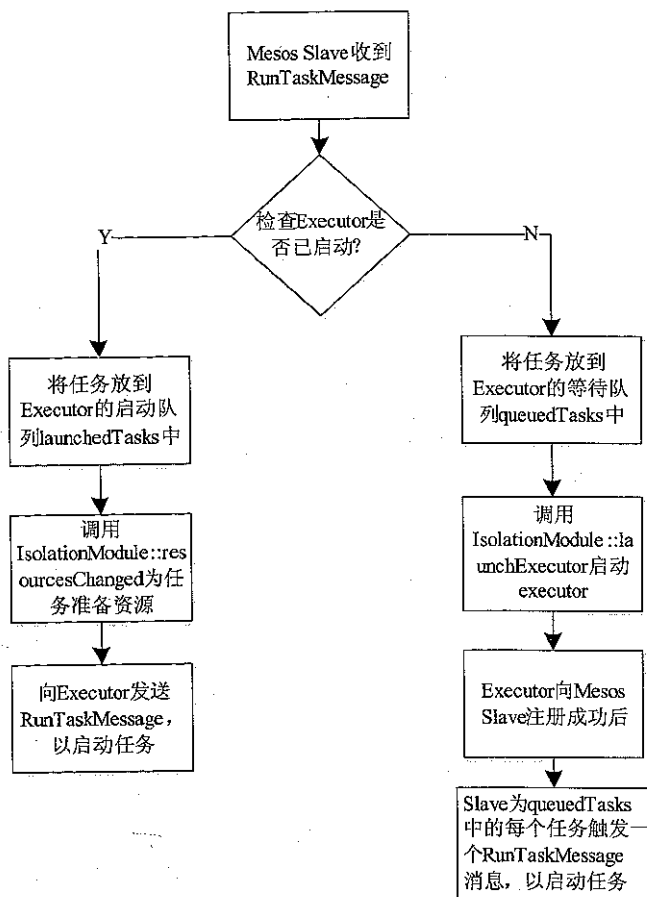


图 12-7 任务启动过程

为了防止任务之间的干扰，同 YARN 一样，Mesos 采用 Cgroups 进行资源隔离，在此不再赘述。

12.4.5 任务状态更新过程

为了让 Framework Scheduler 掌握各个任务的运行状态，Framework Executor 应定期汇报自己所管理的那些任务的运行状态。如图 12-8 所示，在 Mesos 中，Framework Executor 更新任务状态的过程共分为以下几个步骤：

步骤 1 用户编写的 Framework Executor 调用 `MesosExecutorDriver.sendStatusUpdate()`

函数汇报任务运行状态；

步骤 2 ExecutorProcess 向 Mesos Slave 发送一个 StatusUpdateMessage 消息；

步骤 3 经过简单的合法性检查后，Mesos Slave 进一步向 Mesos Master 发送一个 StatusUpdateMessage 消息；

步骤 4 经过简单的合法性检查后，Mesos Master 进一步向对应的 SchedulerProcess 发送一个 StatusUpdateMessage 消息；

步骤 5 SchedulerProcess 调用用户编写的 Scheduler.statusUpdate()，完成任务状态更新；

步骤 6 SchedulerProcess 向对应的 Mesos Slave 发送确认消息 StatusUpdateAcknowledgementMessage，至此完成一次任务状态更新。

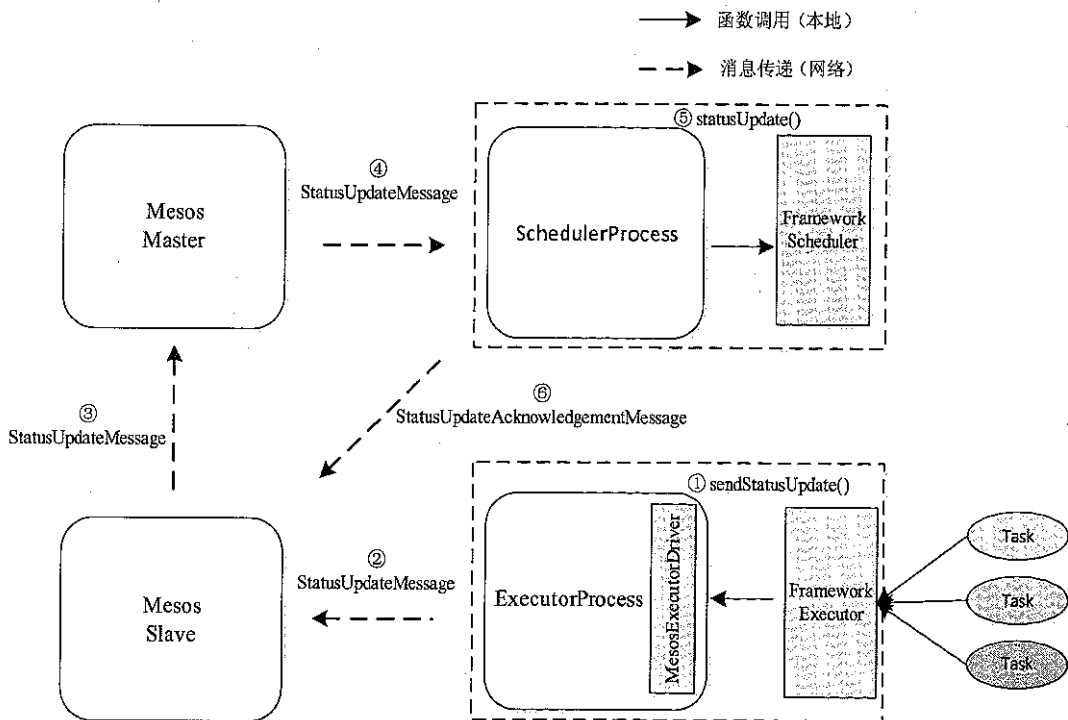


图 12-8 任务状态更新过程

以上整个过程中，如果 Mesos Slave 在一定时间间隔内未收到确认消息，则会重新发送任务状态。

12.5 Mesos 资源分配策略

本节将重点介绍与 Mesos 资源分配策略相关的知识。

12.5.1 Mesos 资源分配框架

Mesos 中最核心的问题是如何构建一个兼具良好扩展性和性能的调度模型以支持各种计算框架。由于不同框架可能有不同的调度需求（这往往跟它的编程模型、通信模型、任务依赖关系和数据放置策略等因素相关），因此，为 Mesos 设计一个好的调度模型是一个极具挑战性的工作。

一种可能的解决方案是构建一个具有丰富表达能力的中央调度器，该调度器接收来自不同框架的详细需求描述，比如资源需求、任务调度顺序和组织关系等，然后为这些任务构建一个全局的调度序列。但是，在真实系统中，由于每种计算框架具有不同的调度需求，且有些框架的调度需求非常复杂，因此，提供一个具有丰富表达能力的 API 以捕获所有框架的需求是不太可能的，也就是说，该方案过于理想化，在真实系统中很难实现。

Mesos 提供了一种简化的方案：将资源调度的控制授权给各个框架。Mesos 负责按照一些简单的策略（如 FIFO、Fair 等）将资源分配给各个框架，而框架内部调度器则根据一些个性化的需求将得到的资源进一步推送给各个作业。考虑到 Mesos 缺少对各个框架的实际资源需求的了解，为保证框架能高效地获取到自己需要的资源，它提供了三个机制：

（1）资源拒绝

如果 Mesos 为某个框架分配的资源不符合它的要求，则框架可以拒绝接受该资源直到出现满足自己需求的资源。该机制使得框架在复杂的资源约束条件下，还能够保证 Mesos 设计简单和具有良好的扩展性。

（2）资源过滤

每次发生资源调度时，Mesos Master 均需要与 Framework Scheduler 进行通信，如果有些框架总是拒绝某些节点上的资源，那么由于额外的通信开销会使得调度性能变得低效。为避免不必要的通信，Mesos 提供了资源过滤机制，允许框架只接收来自“剩余资源量大于 L 的 Mesos Slave”或者“位于特定列表中的 Mesos Slave”上的资源。

（3）资源回收

如果某个框架在一定的时间内没有为分配的资源返回对应的任务，则 Mesos 将回收为其分配的资源，并将这些资源重新分配给其他框架。

12.5.2 Mesos 资源分配算法

同 YARN 和 Corona 类似，Mesos 也是采用了基于真实资源需求量的调度模型。为了支持多维资源调度，Mesos 采用了主资源公平调度（Dominant Resource Fairness, DRF）算法^①，该算法扩展了最大最小公平（max-min fairness）算法^②，使其能够支持多维资源的调度。由于 DRF 被证明非常适合应用于多资源和复杂需求的环境中，因此被越来越多的系统

① Dominant Resource Fairness: Fair Allocation of Multiple Resources Types. A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, NSDI 2011, March 2011.

② Max-Min Fairness (Wikipedia): http://en.wikipedia.org/wiki/Max-min_fairness.

采用, 包括 Apache YARN^①。该算法已经在 6.3.3 节进行了详细介绍, 在此不再赘述。

12.6 Mesos 容错机制

Mesos 作为一个资源管理系统, 应当力求做到两点: 容错和动态升级。首先是容错, 当核心服务组件(比如 Master 和 Slave)自身出现故障时, 不应影响用户应用程序的运行; 其次是动态升级, 当核心服务组件进行升级时, 所有正在运行的应用程序不应受影响。

Mesos 采用的方案有两种: 基于 Zookeeper 的方案和状态快照。其中, Master 采用了基于 Zookeeper 的容错方案, 而 Slave 则采用了快照的方式(即 Slave 通过 checkpoint 将关键信息存放到本地文件; 升级时通过加载 checkpoint 信息恢复自身状态, 并给关联组件发送相关的消息)。

12.6.1 Mesos Master 容错

当主 Master 出现故障时, Zookeeper 会通过选举算法重新选出一个主 Master, 并通知各个 Slave 和 Scheduler 重新向新的主 Master 注册, 以恢复状态信息, 如图 12-9 所示, 具体如下:

- Slave 发送 ReregisterSlaveMessage, Master 重构所存的 Slave/Executor/Task。
 - Slave 信息: 通过重新注册方式可以完整恢复。
 - Executor 信息: 当前仅汇报正在运行的与 Framework 对应的 Executor(已经完成退出的 Framework 信息丢失)。
 - Task 信息: 当前仅汇报正在运行及尚未运行的 Task(已经完成的 Task 信息丢失)。
- Framework Scheduler 发送 ReregisterFrameworkMessage, Master 重构 Framework 信息。Framework 的 register_time 等信息会丢失。

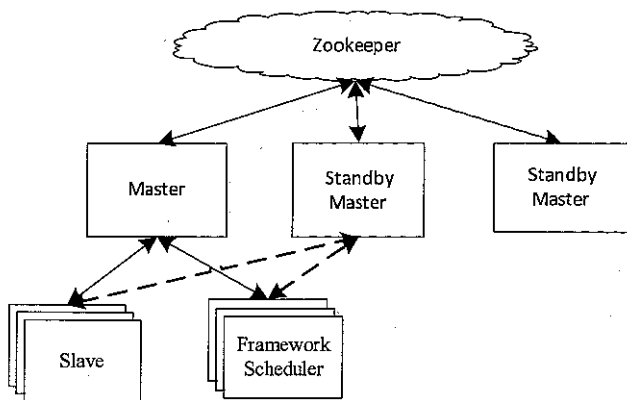


图 12-9 Master 出现故障时触发容错机制

① 参见网址 <https://issues.apache.org/jira/browse/YARN-2>。

12.6.2 Mesos Slave 容错

本小节我们主要介绍 Mesos Slave 容错相关知识。

(1) 超时检测

Slave 与 Master 之间维持了心跳信息，如果一个 Slave 在一段时间内未向 Master 汇报心跳，则 Master 认为该 Slave 已断开连接，从而将它上面所有运行任务的状态置为失败，并通知对应的 Framework Scheduler。

而对于 Slave 的升级，可以忽略 Master 的策略，只要 Slave 做好 checkpoint，之后将信息重新注册到 Master 即可。

(2) 快照

Mesos Slave 按照如图 12-10 所示的层次目录结构将所有信息记录下来，当重启时，再从这些文件中读取信息并恢复状态。当用户使用“-checkpoint”启动 Slave 服务时，它会将 Task、Executor、Framework 和 Slave 自身的信息按照图 12-10 所属关系组织成层次目录结构，并适时保存到磁盘上，当 Slave 突然挂掉或者重启升级时，Slave 不会杀死正在运行的 Executor 或者 Task，而仅是自己退出；当用户使用“-recovery”参数重新启动 Slave 服务时，Slave 会再次从磁盘读取这些信息到内存中以恢复之前的内存状态，并通知 Executor 重新向 Slave 注册，这样 Slave 状态可以得到完全恢复。

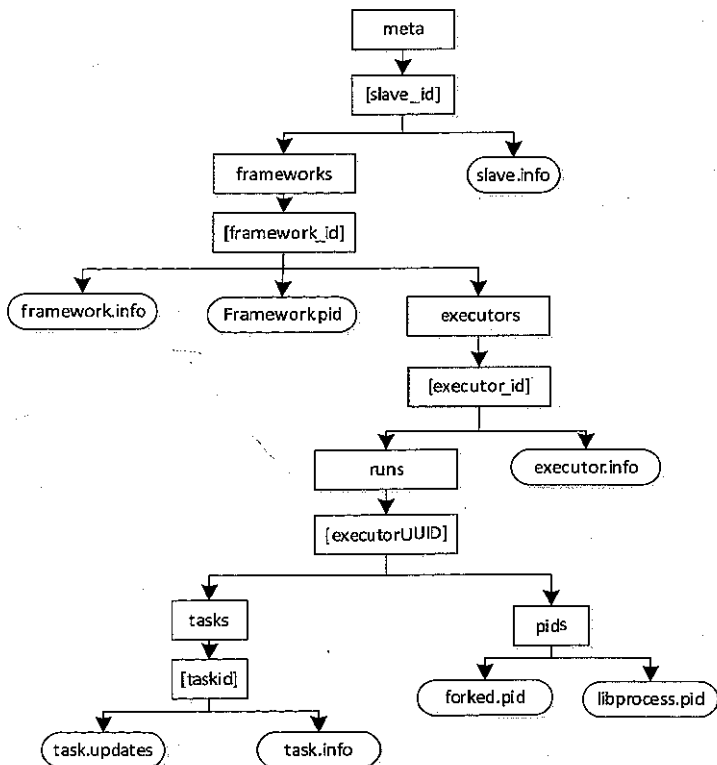


图 12-10 Mesos Slave 快照目录结构组织方式

12.7 Mesos 应用实例

当前有多个框架可直接运行在 Mesos 上, 包括 Hadoop、Storm、Spark、MPI 等, 本节以 Hadoop 和 Storm 两个系统为例介绍如何将一个系统运行在 Mesos 上。

12.7.1 Hadoop On Mesos

由于 Hadoop MapReduce 的 JobTracker 和 TaskTracker 与 Mesos 模型中的 Framework Scheduler 和 Framework Executor 在设计上是完全对应的, 因此, 只须修改少量代码便可让 MapReduce 运行于 Mesos 之上。

为了让 Mesos 支持 Hadoop MapReduce, Mesos 专门为 MapReduce 设计了 scheduler 和 executor 两个组件, 它们的作用如下。

(1) scheduler

Mesos 利用了 Hadoop 调度器可插拔的特性, 重新实现了一个可接入到 Mesos 的调度器 MesosScheduler, 以实现框架注册、资源分配等功能。

(2) executor

Mesos 为 Hadoop 实现了一个 executor——FrameworkExecutor, 通过该 executor, Mesos 可控制 TaskTracker 的启动或停止, 比如, 如果一定时间内没有任务运行, Mesos 会关闭某些节点上的 TaskTracker。

为了使 Hadoop 能运行于 Mesos 之上, JobTracker 启动时通过调度器向 Mesos 注册, 这之后, 资源分配过程如图 12-11 所示, 大致分为以下几个步骤:

步骤 1 Mesos Slave 向 Mesos Master 汇报自己的资源使用情况, 当前支持的资源类型包括 CPU 和内存两种。

步骤 2 如果 Mesos Slave 上有空闲资源, Mesos Master 会按照一定的策略将资源分配给当前向其注册的框架。如果分配给 Hadoop, 则告诉其调度器, 而调度器会记录已经分配到的资源, 并返回一系列 Mesos Task 列表, 其中每个 Mesos Task 会对应一个 Hadoop 任务 (Map Task 或 Reduce Task)。

步骤 3 Mesos Master 将 Mesos Task 列表发送给对应的 Mesos Slave。

步骤 4 Mesos Slave 收到一个 Mesos Task 后, 检查它是否是收到的第一个来自 Hadoop 框架的任务, 如果是, 则通过 executor 为其启动 TaskTracker。

步骤 5 同正常的 Hadoop 集群一样, TaskTracker 向 JobTracker 汇报心跳, 以领取新的任务。

步骤 6 JobTracker 收到来自 TaskTracker 的心跳后, 通过调度器为其分配任务, 需要注意的是, 此时调度器分配多少任务并不取决于该 TaskTracker 上有多少空闲的 Slot, 而是取决于 Mesos 为其分配的资源量 (即在步骤 2 中分配的资源量)。

步骤 7 JobTracker 将新分配的任务返回给对应的 TaskTracker。

步骤 8 TaskTracker 收到新任务后, 启动这些任务。需要注意的是, 这些任务执行过

程中的状态更新不仅会告诉 TaskTracker，也会通过 executor 汇报给 Mesos 内部的服务。

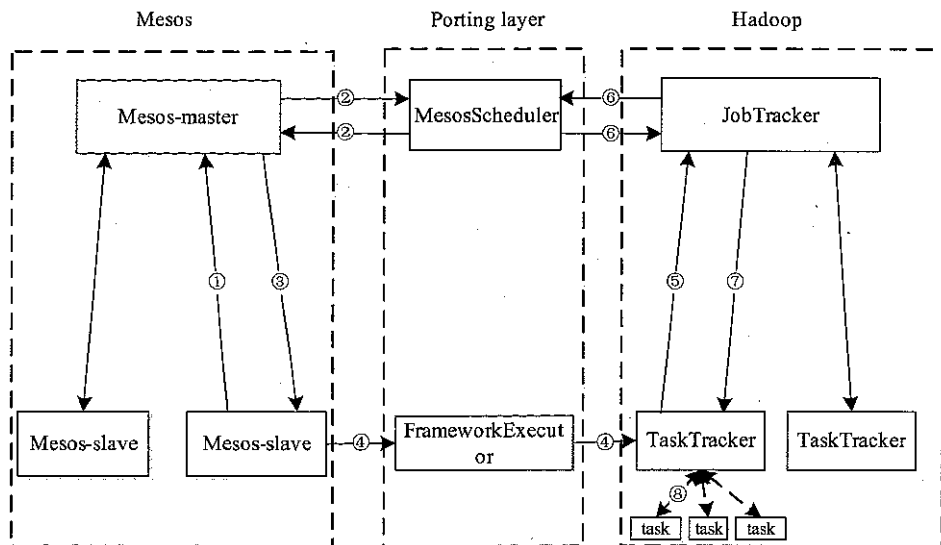


图 12-11 Hadoop On Mesos

12.7.2 Storm On Mesos

同 Hadoop On Mesos 一样，为了让 Storm 运行到 Mesos 之上，Mesos 设计了 MesosNimbus 和 MesosSupervisor 两个组件，它们的作用如下。

(1) MesosNimbus

MesosNimbus 实现了 ISupervisor 接口，并由函数 `backtype.storm.daemon.supervisor.launch` 启动，它内部实现了一个 Framework Scheduler——NimbusScheduler，而 MesosNimbus 则通过该组件与 Mesos Master 交互，以实现框架注册、资源分配、任务启动等功能。

(2) MesosSupervisor

MesosSupervisor 实现了 ISupervisor 接口，并由函数 `backtype.storm.daemon.supervisor.launch` 启动，它内部实现了一个 Framework Executor——StormExecutor，通过该组件，Storm 可控制每个 Supervisor 占用的资源和同时运行的任务数目。

为了使 Storm 能运行于 Mesos 之上，MesosNimbus 启动时向 Mesos 注册，这之后，资源分配过程如图 12-12 所示。

资源分配大致分为以下几个步骤：

步骤 1 Mesos Slave 向 Mesos Master 汇报自己的资源使用情况，当前支持的资源类型包括 CPU 和内存两种。

步骤 2 如果 Mesos Slave 上有空闲资源，Mesos Master 会按照一定的策略将资源分配给当前向其注册的框架。如果分配给 Storm，则告诉 MesosNimbus 中的 NimbusScheduler，

它将 MesosSupervisor 启动命令封装成一个任务。

步骤 3 Mesos Master 将任务发送给对应的 Mesos Slave。

步骤 4 Mesos Slave 收到一个任务后，启动该任务，实际上就是启动 MesosSupervisor。

通过以上步骤，一个 Storm 集群就运行起来了，之后的整个逻辑流程和使用方式跟独立搭建的 Storm 集群的完全一致。

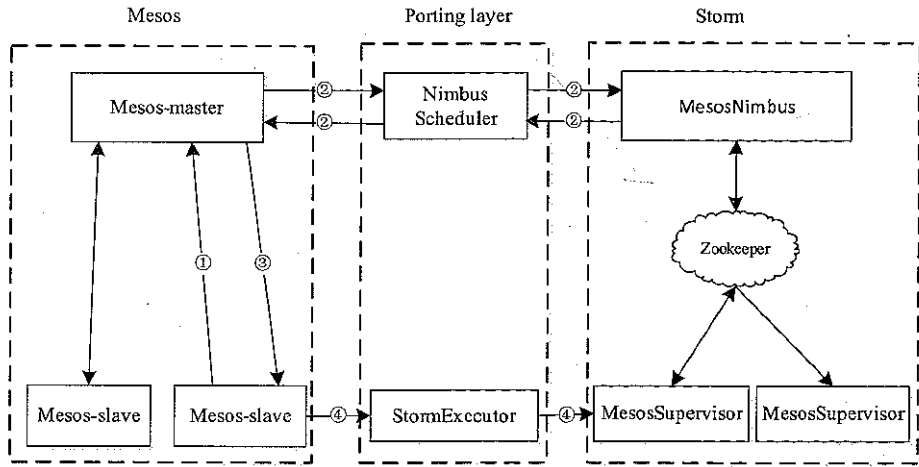


图 12-12 Storm On Mesos

12.8 Mesos 与 YARN 对比

本节将重点分析 Mesos 与 YARN 的异同。

(1) 基本架构对比

尽管 YARN 和 Mesos 诞生于不同的公司或者研究机构，但它们的架构却大同小异，表 12-5 给出了它们内部基本组件的对应关系（同时给出了与 Corona 的对比）。

表 12-5 YARN、Corona 和 Mesos 基本组件对应关系

YARN	Corona	Mesos	功 能
Resource Manager	Cluster Manager	Mesos Master	负责整个集群的资源管理和调度
Node Manager	Corona TaskTracker	Mesos Slave	负责单个节点的资源管理（资源隔离、资源使用汇报等）、任务管理（启动、杀死等）和任务运行进度汇报等
		Framework- executor	
Application Master	Corona JobTracker	Framework-scheduler	负责单个应用程序的管理和资源的二次调度（将资源分配给内部的各个任务）

(2) 其他方面对比

表 12-6 所示从设计目标、容错性、在线升级、调度模型、调度算法、资源隔离等方面对比了 Mesos 和 YARN。

表 12-6 Mesos 与 YARN 对比

	Mesos	YARN
设计目标	提供一个通用的资源管理平台供上层框架使用，使用户可专注于应用程序逻辑相关的实现而无需关注资源分配和调度	
主服务容错性	均使用 Zookeeper 实现多 Master	
在线升级	Master 和 Slave 均可在线升级	Master 可在线升级，但 Slave 不可以
调度模型	均采用双层调度模型，且为细粒度资源调度模型，即将 CPU、内存等资源直接分配给应用程序，目前均只支持内存和 CPU 两种资源	
调度算法	基于 DRF 进行多类别资源调度	提供了多种调度算法，包括 Capacity Scheduler 和 Fair Scheduler 等，同时也实现了基于 DRF 的调度算法
资源隔离	均采用 CGroups 进行资源隔离	
支持的框架	目前均支持 MapReduce、Storm、Spark 等框架，并有更多框架在添加中	
社区活跃度	一般	非常活跃

12.9 小结

本章介绍了 Mesos，一个诞生于 UC Berkeley 的资源管理系统，它解决编程模型和计算框架在多样化环境下，不同框架间的资源隔离和共享问题。尽管它的直接设计动机与 MRv1 无关，但它的架构和实现策略与 YARN 非常类似，目前，有多个框架可直接运行在 Mesos 之上，包括 Storm、MapReduce、Spark、MPI 等。

本章从基本架构、工作流程、调度策略、容错机制等方面介绍了 Mesos，还在最后通过两个实例——Hadoop On Mesos 和 Storm On Mesos，介绍了如何将现有框架运行在 Mesos 之上。

第 13 章 YARN 总结与发展趋势

本章作为本书的最后一章，将总结本书的主要内容并尝试给出 YARN 可能的发展趋势，本章内容仅用于讨论，读者可各抒己见，在讨论中加深对 YARN 的理解。

13.1 资源管理系统设计动机

尽管目前 YARN 的设计框架比较适合运行类似于 MapReduce 这样的短作业，但它最终的定位是通用的资源管理系统。除了 YARN 之外，目前市面上存在很多其他资源管理系统，典型代表有 Google 的 Borg、Twitter 的 Mesos、腾讯的 Torca[⊖]和 Facebook 的 Corona，它们是大数据时代的必然产物。概括起来，这类系统的动机是解决以下两类问题：

- **提高集群资源利用率。**在大数据时代，为了存储和处理海量数据，需要规模较大的服务器集群或者数据中心。一般说来，这些集群上运行着数量众多、类型纷杂的应用程序和服务，比如离线作业、流式作业、迭代式作业、Crawler Server、Web Server 等。传统的做法是，每种类型的作业或者服务对应一个单独的集群，以避免相互干扰。这样，集群被分割成数量众多的小集群：Hadoop 集群、HBase 集群、Storm 集群，Web server 集群等，然而，由于不同类型的作业 / 服务需要的资源量不同，因此，这些小集群的利用率通常很不均衡，有的集群满负荷、资源紧张，而另外一些则长时间闲置、资源利用率极低。而由于这些集群之间资源无法共享，因此造就了不同时间段不同集群资源利用率不同。为了提高资源整体利用率，一种解决方案是将这些小集群合并成一个大集群，让它们共享这个大集群的资源，并由一个资源统一调度系统进行资源管理和分配，这就诞生了类似于 YARN 的系统。从集群共享角度看，这类系统实际上将公司的所有硬件资源抽象成一台大型计算机，供所有用户使用。
- **服务自动化部署。**一旦将所有计算资源抽象成一个“大型计算机”后，就会产生一个问题：公司的各种服务如何进行部署？同样，Borg、YARN、Mesos、Torca、Corona 一类系统需要具备服务自动化部署的功能。因此，从服务部署的角度看，这类系统实际上是服务统一管理系统，这类系统提供服务资源申请、服务自动化部署、服务容错等功能。

⊖ 参见网址 <http://djt.qq.com/bbs/thread-29998-1-1.html>。

13.2 资源管理系统架构演化

Google 在论文《Omega: flexible, scalable schedulers for large compute clusters》^①中介绍了 Google 经历的三代资源调度器的架构, 如图 13-1 所示, 分别是集中式调度器架构(类似于 Hadoop JobTracker, 但是支持多种类型作业调度)、双层调度器架构(类似于 Mesos 和 YARN)和共享状态架构(Omega), 并分别讨论了这几个架构的优缺点, 重点剖析了最新资源管理系统 Omega 的相关实现。本节将解读 Google 的这篇论文, 并结合开源界相关系统的演化, 阐述资源管理系统架构的演化过程。

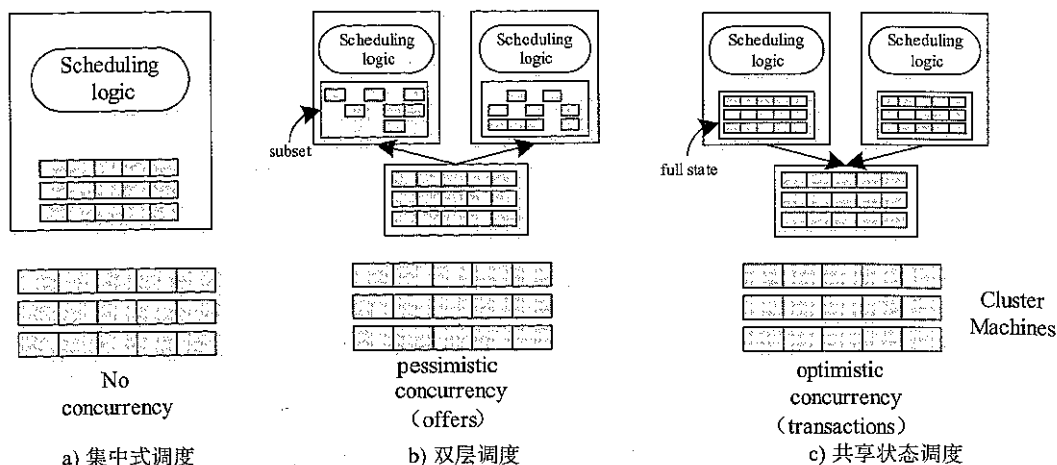


图 13-1 资源管理系统架构演化

13.2.1 集中式架构

集中式调度器 (Monolithic Scheduler) 的特点是, 资源的调度和应用程序的管理功能全部放到一个进程中完成, 开源界典型的代表是 MRv1 JobTracker 的实现。这种设计方式的缺点很明显, 扩展性差: 首先, 集群规模受限; 其次, 新的调度策略难以融入现有代码中, 比如之前仅支持 MapReduce 作业, 现在要支持流式作业, 而将流式作业的调度策略嵌入到中央式调度器中是一项很难的工作。

上述的 Omega 论文中提到了一种对集中式调度器的优化方案: 将每种调度策略放到单独一个路径 (模块) 中, 不同的作业由不同的调度策略进行调度。这种方案在作业量和集群规模都比较小时, 能大大缩短作业相应时间, 但由于所有调度策略仍在一个集中式的组件中, 整个系统扩展性没有变得更好。

① SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In Proc. EuroSys (2013).

13.2.2 双层调度架构

为了解决集中式调度器的不足，双层调度器（Two-level scheduler）是一种很容易被想到的解决之道，它可看做一种分而治之的机制或者是策略下放机制：双层调度器仍保留一个经简化的集中式资源调度器，但具体任务相关的调度策略则下放到各个应用程序调度器完成。这种调度器的典型代表是 Mesos 和 YARN。上述的 Omega 论文重点介绍了 Mesos，Mesos 是 twitter 开源的资源管理系统。正如第 12 章的介绍，Mesos 调度器由两部分组成，分别是资源调度器和框架（应用程序）调度器，其中，资源调度器负责将集群中的资源分配给各个框架（应用程序），而框架（应用程序）调度器负责将资源进一步分配给内部各个任务，用户很容易将一种框架或者系统接入 Mesos。当前很多框架已经成功接入 Mesos 中，包括 Hadoop、MPI、Spark 等。

双层调度器的特点是：各个框架调度器并不知道整个集群资源使用情况，只是被动地接收资源；资源调度器仅将可用的资源推送给各个框架，而由框架自己选择是使用还是拒绝这些资源；一旦框架接收到新资源，再进一步将资源分配给其内部的任务，进而实现双层调度。

然而，双层调度器的缺点也是非常明显的：

- 各个框架无法知道整个集群的实时资源使用情况。很多框架不需要知道整个集群的实时资源使用情况就可以运行得很顺畅，但是对于其他一些应用，为之提供实时资源使用情况可以挖掘潜在的优化空间。比如，当集群非常繁忙时，一个服务在一个节点上运行失败了，此时是应该选择换一个节点重新运行它呢，还是在这个节点上再次尝试运行？通常而言，换一个节点可能会更有利（排除硬件故障和节点环境问题导致失败），但是，如果此时集群非常繁忙，所有节点只剩下不足 5GB 的内存，而这个服务需要 10GB 内存，那么换一个节点可能意味着长时间等待资源释放，且这个等待时间是无法确定的。
- 采用悲观锁，并发粒度小。在数据库领域，悲观锁与乐观锁争论一直不休。悲观锁通常采用锁机制控制并发，这会大大降低性能，而乐观锁则采用多版本并发控制，典型代表是 MySQL innodb，这种机制通过多版本方式控制并发，可大大提升性能。在 YARN 或 Mesos 中，在任意一个时刻，Mesos 资源调度器只会将某些资源推送给一个框架，等到该框架返回资源使用情况后，才能将资源再次推动给其他框架。因此，YARN 或 Mesos 资源调度器中实际上有一个全局锁，这大大限制了系统并发性。

13.2.3 共享状态架构

为了克服双层调度器的以上两个缺点，Google 开发了下一代资源管理系统 Omega。Omega 是一种基于共享状态的调度器（Shared State Scheduler），该调度器将双层调度器中的集中式资源调度模块简化成了一些持久化的共享数据（状态）和针对这些数据的验证代码，而这里的“共享数据”实际上就是整个集群的实时资源使用信息。一旦引入共享数据

后，共享数据的并发访问方式就成为了该系统设计的核心，而 Omega 则采用了传统数据库中基于多版本的并发访问控制方式（也称为乐观锁），这大大提升了 Omega 的并发性。

由于 Omega 不再有集中式的调度模块，因此，不能像 Mesos 或者 YARN 那样，在一个统一模块中完成以下功能：对整个集群中的所有资源分组，限制每类应用程序的资源使用量，限制每个用户的资源使用量等，这些全部由各个应用程序调度器自我管理和控制。根据论文所述，Omega 只是将优先级这一限制放到了共享数据的验证代码中，即当同时由多个应用程序申请同一份资源时，优先级最高的那个应用程序将获得该资源，其他资源限制全部下放到各个子调度器。

引入多版本并发控制后，限制该机制性能的一个因素是资源访问冲突的次数，冲突次数越多，系统性能下降得越快，而 Google 通过实际负载测试证明，这种方式的冲突次数是完全可以接受的。

上述的 Omega 论文中谈到，Omega 是从 Google 现有系统上演化而来的。也就是从类似于 YARN 或者 Mesos 的系统改造而来的。经过前面的介绍，有心的读者应该会发现，可以很容易将 YARN 或者 Mesos 改造成一个类 Omega 的系统，有兴趣的读者可以尝试一下这项工作。

13.3 YARN 发展趋势

本章尝试从 YARN 自身完善、YARN 生态系统和 YARN 周边工具的完善等几个方面分析 YARN 可能的发展趋势，供大家参考和讨论。

13.3.1 YARN 自身的完善

本小节我们先从 YARN 自身来分析一下其发展趋势。

(1) 调度框架

当前 YARN 支持内存和 CPU 两种资源类型的管理和分配。除了 CPU 和内存两种资源，服务器还有很多其他资源，比如磁盘容量、网络 and 磁盘 IO 等，YARN 可能在将来支持这些资源的调度和隔离。

为了更友好地为应用程序分配资源，YARN 内部包含了一些调度语义，这决定了 YARN 作为一个资源管理系统可给用户带来的服务承诺。当前 YARN 的调度器能够表达的语义是有限的，这也决定了它对某些应用是友好的，但对于其他应用来说会带来性能问题。当前 YARN 支持的语义和不支持的语义可总结如下。

□ 支持的调度语义：

- 请求某个特定节点上的特定资源量。比如，请求节点 nodeX 上 5 个这样的 Container：
虚拟 CPU 个数为 2，内存量为 2GB。
- 请求某个特定机架上的特定资源量。比如，请求机架 rackX 上 3 个这样的 Container：
虚拟 CPU 个数为 4，内存量为 3GB。

- 将某些节点加入（或移除）黑名单，不再为自己分配这些节点上的资源。比如，ApplicationMaster 发现节点 nodeX 和 nodeY 失败的任务数目过多，可请求将这两个节点加入黑名单，从而不再收到这两个节点上的资源，过一段时间后，可请求将 nodeX 移除黑名单，从而可再次使用该节点上的资源。
- 请求归还某些资源。比如，ApplicationMaster 已获取的来自节点 nodeX 上的 2 个 Container 暂时不用了，可将之归还给集群，这样这些资源可再次分配给其他应用程序。
- 不支持的调度分配语义^①：
 - 请求任意节点上的特定资源量。比如，请求任意节点上 5 个这样的 Container：虚拟 CPU 个数为 3，内存量为 1GB。
 - 请求任意机架上的特定资源量。比如，请求同一个机架上的（具体并不关心是哪个机架，但是必须来自同一个机架）3 个这样的 Container：虚拟 CPU 个数为 1，内存量为 6GB。
 - 请求一组或几组符合某种特质的资源。比如，请求来自两个机架上的 4 个 Container，其中一个机架上 2 个这样的 Container：虚拟 CPU 个数为 2，内存量为 2GB，另一个机架上 2 个这样的资源：虚拟 CPU 个数为 2，内存量为 3GB。如果目前集群没有这样的资源，需要从其他应用程序那里抢占资源。
 - 超细粒度资源。比如 CPU 性能要求、绑定 CPU 等。
 - 动态调整 Container 资源，应允许根据需要动态调整 Container 资源量（对于长作业尤其有用）。

（2）在线升级

YARN 作为一个通用资源管理平台，不仅可以运行短作业（通常运行几分钟或者几个小时，比如 MapReduce 作业），还可以运行长作业（通常永不停止运行，比如 HBase Server 或者 Web Server）等。对于长作业而言，由于它们需对外不间断提供服务，因此，当 YARN 自身升级时，不应该干扰这类长作业，比如 NodeManager 重启时，正运行在它上面的任务（尤其是长服务）不应受到干扰，这一点，同类系统 Mesos 已经可以做到（具体参考第 12 章）。

（3）容错机制

将短作业和长作业同时运行在 YARN 上后，不同类型的应用程序对容错性需求通常不一样，对于长作业而言，由于它们通常拥有自己的客户端，需为它们不间断提供服务。因此，当这些作业出现故障时，应尝试本地重启它们（防止客户端无法找到服务位置），如果重启失败，再尝试将其调度到其他节点上，而不应向 MapReduce 作业那样，一旦运行失败则换另外一个节点重新运行。

^① 很多特性已经提出来并已在解决中，具体参考 <https://issues.apache.org/jira/browse/YARN-896>。

13.3.2 以 YARN 为核心的生态系统

YARN 是未来的一个趋势，YARN 本身已经变成了一个云操作系统，很多计算框架或者应用程序不再基于传统的操作系统开发（比如 Linux），而是基于 YARN 这个云操作系统。这意味着，很多新的计算框架或者应用程序脱离了 YARN 将不再能够单独运行，典型的代表是 DAG 计算框架 Tez 和 Spark（Spark 也可以运行在 Mesos 上）。

当前很多计算框架都在往 YARN 上迁移，即使不迁移也要支持在 YARN 上运行。本小节将对这部分内容进行简单的梳理。

- MapReduce: MapReduce 是一个非常经典的离线计算框架，在 MRv1 中，MapReduce 应用程序需运行在由 JobTracker 和 TaskTracker 组成的运行时环境中，而在 YARN 中，不再有 JobTracker 和 TaskTracker 这样的服务组件，取而代之的是一个 ApplicationMaster 组件，它只负责应用程序相关的管理，比如任务切分和调度、任务监控和容错等，而资源相关的调度和管理交给 YARN 完成，具体在第 8 章已经进行了介绍。
- Tez: Hortonworks 开源的 DAG 计算框架，是在 MapReduce 基础上扩展而来的，重用了 MapReduce 大量代码，仅支持运行在 YARN 上，不可单独运行，具体在第 9 章已经进行了介绍。
- Storm: 实时计算框架，运行时环境由 Nimbus 和 supervisor 等组件组成，为了让 Storm 运行在 YARN 上，Yahoo! 开源了一个简单的版本 Storm On YARN，具体在第 10 章已经进行了介绍。
- Spark: Spark 是一个基于内存实现的 MapReduce 计算框架，某些应用场景下更加高效，它目前已经支持运行在 YARN 上，具体在第 10 章已经进行了介绍。
- HBase: 构建在 HDFS 之上的实时存取系统，运行时环境由 HMaster 和 RegionServer 等组件构成。Hortonworks 公司开源的 Hoya 项目正常时将 HBase 自动化部署到 YARN 上^①。
- Giraph: 开源图算法库，最开始的版本是基于 MRv1 实现的，随着 Hadoop 2.0 的成熟，正尝试将所有图算法运行在 YARN 之上（不再基于 MapReduce）^②。
- OpenMPI: 非常经典的高性能并行编程接口，目前正尝试将其运行在 YARN 上^③。

最终，YARN 之上可以运行各种应用类型的框架，包括离线计算框架 MapReduce、实时计算框架 Storm、DAG 计算框架 Tez 等，真正实现一个集群的多用途，这样的集群或者系统我们通常称为轻量级弹性计算平台。说它轻量级，是因为 YARN 采用了 Cgroups 轻量级隔离方案；说它弹性，是因为 YARN 能根据各种计算框架、应用的负载或者需求调整它们各自占用的资源，实现集群资源共享、资源弹性收缩。在不久的将来，普遍采用的部署方案应该如图 13-2 所示。

① 参见网址 <https://github.com/hortonworks/hoya>。

② 参见网址 <https://issues.apache.org/jira/browse/GIRAPH-13>。

③ 参见网址 <https://issues.apache.org/jira/browse/MAPREDUCE-2911>。

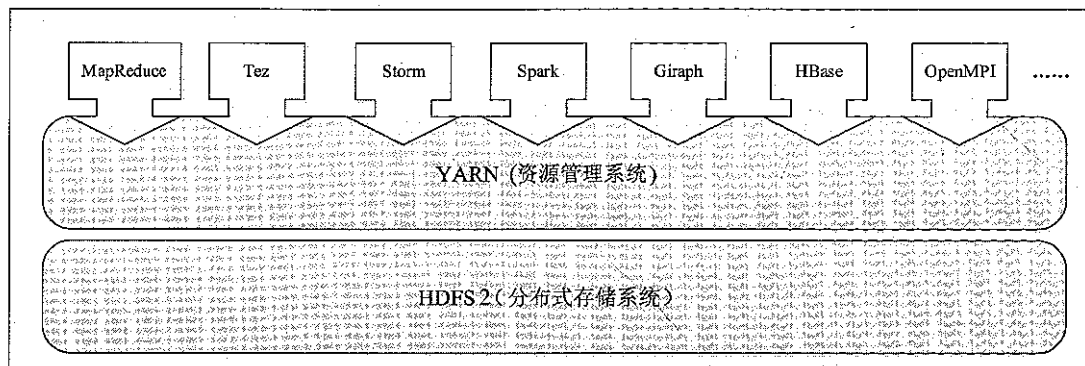


图 13-2 以 YARN 为核心的生态系统

当然，随着 YARN 朝着资源管理系统方向更好地发展，最终 Web Server、MySQL Server 等服务，均可以部署到 YARN 之上，这样 YARN 将变为一个服务统一部署和管理的平台，最终形成一个以 YARN 为核心的生态系统。如图 13-3 所示，未来整个集群或者数据中心只需部署三个基础系统，分别是 HDFS2（Hadoop 2.0 中的 HDFS 实现，增加了 NameNode HA 和 Name Node Federation 两个重要功能）、YARN 和 Zookeeper，其他系统，比如 HBase、Storm、MySQL 等，均可根据需要自动部署运行到 YARN 上。

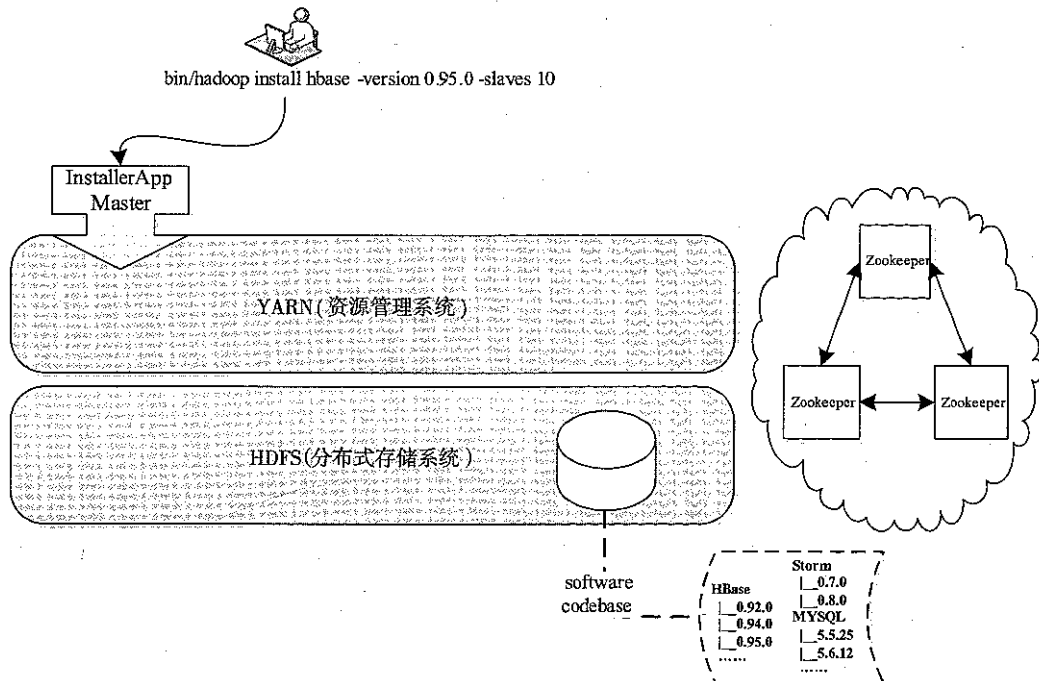


图 13-3 未来可能的 Hadoop 生态系统基础软件形态

然而，从实际应用看来，由于不同应用或者服务对调度策略的要求不同，这将使得

YARN 资源调度器变得非常复杂,难以兼顾所有需求。为了解决该问题,YARN 将来可能出现多种定制化版本应对不同类型的调度问题,比如现有 YARN 用来运行短作业,而定制化的 YARN 用来运行各类服务等。

13.3.3 YARN 周边工具的完善

随着 YARN 的不断发展和完善,各种类型的应用程序,包括类似 MapReduce 的短作业、类似 Web Service 的长作业等,均可直接部署和运行到 YARN 上。当前 YARN 对外提供的接口均是底层接口,这给用户编写和调试应用程序带来了很大的麻烦,比如无法聚集分散在各个节点上的应用程序日志、应用程序生命周期难以管理、缺乏第三方工具将一个现有的系统运行在 YARN 上等。为了解决以上关于 YARN 上应用程序的创建、管理、调试等方面的问题,一些开源软件诞生了,典型代表是 Continuity 公司的 Weave^①和 Cloudera 公司的 Kitten^②。

接下来以 Weave 为例,说明 YARN 尚需进一步完善的周边工具。Weave 针对 YARN 之上应用程序编写、调试和管理等方面的不完善,进行了以下改进:

- 一套经简化的,用于定义、运行和管理应用程序的编程接口。
- 一个通用的 ApplicationMaster 实现,以便直接部署简单的服务。
- 应用程序的日志和 Metric 信息自动聚集呈现。
- Discovery 服务。该组件对长作业尤其有用,比如将一个服务部署到 YARN 上,服务运行起来后,可向 Zookeeper 注册,这样当服务失败被重新调度运行到另外节点上后,Weave 可将新的位置通知给各个客户端。

相比于 Weave, Kitten 提供了一套 Shell 配置工具提交应用程序,这对于部署一些简单的服务或者应用程序非常有用。

13.4 小结

本章从资源管理系统设计动机、架构演化和发展趋势几个方面对 YARN 进行了总结和展望。当前类似于 YARN 的资源管理系统较多,它们主要设计动机是提高资源利用率和服务自动化部署。从架构发展上看,它们将会经历集中式架构、双层调度架构和共享状态架构的变迁几个阶段,对于 YARN 而言,它未来将会在完善自我的同时,逐步进化成以自己为核心的生态系统。

① 参见网址 <http://continuity.github.io/weave/>。

② 参见网址 <https://github.com/cloudera/kitten>。

附录 A YARN 安装指南

本文档将介绍非安全模式下 YARN 的安装方法，为了让读者能够在安装过程中进一步认识 YARN 的组织结构，这里统一按照原生态 JAR 包方法进行安装。

Hadoop YARN 安装主要分为两步：配置和启动。其中配置比较复杂，涉及服务启动环境设置（修改 `hadoop-env.sh` 和 `yarn-env.sh`）、各种配置文件设置（修改 `core-site.xml`、`hdfs-site.xml`、`yarn-site.xml` 和 `mapred-site.xml`）等，而启动则相对简单，只要执行 Hadoop 提供的脚本即可。

1. 服务启动环境变量设置

服务启动环境是指 Hadoop 服务启动时的环境变量，比如 JDK 路径、服务进程占用的 JVM 堆大小、服务进程 JVM 参数（比如垃圾回收机制等）等，通常在 `${HADOOP_HOME}/etc/hadoop` 下的配置文件 `hadoop-env.sh` 和 `yarn-env.sh` 中设置环境变量。关键环境变量如下：

- `JAVA_HOME` (`hadoop-env.sh`)：JDK 安装路径，默认情况下直接读取 Linux 环境环境变量 `${JAVA_HOME}`。
- 服务进程 JVM 相关环境变量：具体如表 A-1 所示。

表 A-1 服务进程 JVM 相关环境变量

服务进程	环境变量	配置文件
NameNode	<code>HADOOP_NAMENODE_OPTS</code>	<code>hadoop-env.sh</code>
DataNode	<code>HADOOP_DATANODE_OPTS</code>	<code>hadoop-env.sh</code>
Secondary NameNode	<code>HADOOP_SECONDARYNAMENODE_OPTS</code>	<code>hadoop-env.sh</code>
ResourceManager	<code>YARN_RESOURCEMANAGER_OPTS</code>	<code>yarn-env.sh</code>
NodeManager	<code>YARN_NODEMANAGER_OPTS</code>	<code>yarn-env.sh</code>
WebAppProxy	<code>YARN_PROXYSERVER_OPTS</code>	<code>yarn-env.sh</code>
Map Reduce Job History Server	<code>HADOOP_JOB_HISTORYSERVER_OPTS</code>	<code>hadoop-env.sh</code>

比如，为 NameNode 启用 `parallelGC` 垃圾回收机制，则可这样配置：

```
export HADOOP_NAMENODE_OPTS="-XX:+UseParallelGC ${HADOOP_NAMENODE_OPTS}"
```

- 服务进程输出日志位置相关环境变量：具体如下。

- YARN 相关服务：`YARN_LOG_DIR`，默认是 `$YARN_HOME/logs`。

- 其他服务：`HADOOP_LOG_DIR`，默认是 `$HADOOP_HOME/logs`。

- 服务进程 JVM 堆环境变量：可通过 `HADOOP_HEAPSIZE` 和 `YARN_HEAPSIZE` 设

置服务进程 JVM 堆大小，比如设置成 1000，表示 JVM 空间上限为 1000MB。也可以通过表 A-2 所示参数定制化每个服务进程堆大小。

表 A-2 服务进程 JVM 堆环境变量

服务进程	环境变量	配置文件
ResourceManager	YARN_RESOURCEMANAGER_HEAPSIZE	yarn-env.sh
NodeManager	YARN_NODEMANAGER_HEAPSIZE	yarn-env.sh
WebAppProxy	YARN_PROXYSERVER_HEAPSIZE	yarn-env.sh
Map Reduce Job History Server	HADOOP_JOB_HISTORYSERVER_HEAPSIZE	yarn-env.sh

2. 服务配置文件

本节介绍最精简的 YARN 集群配置参数（见表 A-2 ~ 表 A-6），更细粒度的参数配置将在附录 B 中介绍。

表 A-3 core-site.xml

参数名称	默认值	说明
fs.defaultFS	—	NameNode 的 RUI，形式如：hdfs://host:port/

表 A-4 hdfs-site.xml

参数名称	默认值	说明
dfs.namenode.name.dir	—	存放 NameNode 命名空间和日志信息的目录，如果有多个，用“,”分割
dfs.datanode.data.dir	—	DataNode 用于存放数据块的目录，如果有多个，用“,”分割

表 A-5 yarn-site.xml

参数名称	默认值	说明
yarn.resourcemanager.address	0.0.0.0:8032	ResourceManager 对外的 IPC 地址
yarn.nodemanager.aux-services	—	附属服务名称，如果使用 MapReduce，需将之配置为 mapreduce-shuffle [Ⓔ]

表 A-6 mapred-site.xml

参数名称	默认值	说明
mapreduce.framework.name	local	MapReduce 的运行环境，需将之设置为 yarn

对于 master/slaves，通常而言，Hadoop 集群中有一个节点运行 NameNode，一个节点运行 ResourceManager，其他节点（slave）则同时运行 DataNode 和 NodeManager。

Slave 节点列表在 slaves 文件中设置，每个 slave 的 host 或者 IP 占一行。

[Ⓔ] Hadoop 2.2.0 版本之前配置名称为“mapreduce.shuffle”，具体参考 <https://issues.apache.org/jira/browse/YARN-1229>。

3. 启动 / 关闭服务

在 NameNode 所在节点上进行以下操作：

□ 格式化 NameNode: `bin/hdfs namenode -format`。

□ 启动 HDFS: `sbin/start-dfs.sh`。

□ 关闭 HDFS: `sbin/stop-dfs.sh`。

在 ResourceManager 所在节点上进行以下操作：

□ 启动 YARN 所有服务: `sbin/start-yarn.sh`。

□ 关闭 YARN 所有服务: `sbin/stop-yarn.sh`。

在任意节点上，启动 MapReduce History Server：`sbin/mr-jobhistory-daemon.sh start historyserver`。

附录 B YARN 配置参数介绍

相比于 MRv1, YARN 的配置参数命名则规范得多, 它的参数采用层次命名方式; 即前缀统一为 “yarn.”, 接下来是服务名称, 可以是 “resourcemanager”、“nodemanager”、“scheduler”、“admin” 等, 比如 “yarn.resourcemanager.address” 表示 ResourceManager 对外 IPC 地址。采用这样的命名方式, 给出一个参数名称, 用户即可很容易通过分析获知它的可能含义。

1. YARN 配置参数介绍

通信地址类参数如表 B-1 所示。

表 B-1 通信地址类参数

参数名称	默认值	说 明
yarn.resourcemanager.address	0.0.0.0:8032	RM 对外的 IPC 地址
yarn.resourcemanager.scheduler.address	0.0.0.0:8030	调度器对外的 IPC 地址
yarn.resourcemanager.resource-tracker.address	0.0.0.0:8031	ResourceTracker 对外的 IPC 地址
yarn.resourcemanager.admin.address	0.0.0.0:8033	RM Admin 对外的 IPC 地址
yarn.resourcemanager.webapp.address	0.0.0.0:8080	RM 的 Web 访问地址
yarn.nodemanager.aux-services	—	附属服务名称, 如果使用 MapReduce, 需将之配置为 mapreduce-shuffle

目录类参数如表 B-2 所示。

表 B-2 目录类参数

参数名称	默认值	说 明
yarn.nodemanager.local-dirs	\${hadoop.tmp.dir}/nm-local-dir	NM 存放临时文件的本地目录
yarn.nodemanager.log-dirs	\${yarn.log.dir}/userlogs	NM 存放 Container 日志的本地目录
yarn.nodemanager.log-retain-seconds	10800	Container 日志在 NodeManager 上的保存时间 (未启用日志聚集功能下有效)
yarn.nodemanager.remote-app-log-dir	/tmp/logs	Container 日志在 HDFS 存放位置 (启用日志聚集功能下时有效)
yarn.nodemanager.remote-app-log-dir-suffix	logs	日志目录后缀 (启用日志聚集功能时有效)

资源类参数如表 B-3 所示。

安全类参数如表 B-4 所示。

表 B-3 资源类参数

参数名称	默认值	说 明
yarn.nodemanager.resource.cpu-vcores	8	NM 上可用虚拟 CPU 数目
yarn.nodemanager.resource.memory-mb	8192	NM 上可用内存量, 单位是 MB
yarn.nodemanager.vmem-pmem-ratio	2.1	每使用 1MB 物理内存, 最多可使用虚拟内存量
yarn.scheduler.maximum-allocation-mb	8192	单个 Container 可申请的最多内存资源
yarn.scheduler.minimum-allocation-mb	1024	单个 Container 可申请的最少内存资源
yarn.scheduler.maximum-allocation-vcores	32	单个 Container 可申请的最多虚拟 CPU 个数
yarn.scheduler.minimum-allocation-vcores	1	单个 Container 可申请的最少虚拟 CPU 个数

表 B-4 安全类参数

参数名称	默认值	说 明
yarn.acl.enable	true	是否启用 ACL
yarn.admin.acl	*	集群管理员列表
arn.resourcemanager.nodes.include-path	—	节点白名单
yarn.resourcemanager.nodes.exclude-path	—	节点黑名单

节点健康状况监测如表 B-5 所示。

表 B-5 节点健康状况监测

参数名称	默认值	说 明
yarn.nodemanager.health-checker.script.path	—	健康检测脚本所在的绝对路径, 服务 NodeHealthScriptRunner 会周期性执行该脚本以判断节点健康状况, 如果该值为空, 则不会启动该线程
yarn.nodemanager.health-checker.interval-ms	600 000	健康监测脚本调用频率 (单位: 毫秒)
yarn.nodemanager.health-checker.script.timeout-ms	1 200 000	如果健康监测脚本在一定时间 (单位: 毫秒) 内没有响应, 则 NodeHealth-ScriptRunner 服务会将节点标注为 “unhealthy”
yarn.nodemanager.health-checker.script.opts	—	监控脚本的输入参数, 如果有多个参数则用逗号隔开
yarn.nodemanager.disk-health-checker.enable	true	是否开启磁盘健康监测功能
yarn.nodemanager.disk-health-checker.min-healthy-disks	0.25	最低健康磁盘比率, 若低于该比率, NodeManager 将不会收到新任务

其他参数如表 B-6 所示。

2. MapReduce 配置参数介绍

MapReduce 参数在配置文件 `mapred-site.xml` 中设置, 由于 MapReduce 库已经变为 YARN 客户端程序, 因此该配置文件只在客户端有效。

MapReduce 客户端设置如表 B-7 所示。

表 B-6 其他参数

参数名称	默认值	说 明
yarn.resourcemanager.scheduler.class	org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler	调度器类
yarn.log-aggregation-enable	false	是否启用日志聚集功能
yarn.resourcemanager.resource-tracker.client.thread-count	50	ResourceManager 启动的处理 NodeManager RPC 请求的 Handler 数目
yarn.resourcemanager.scheduler.client.thread-count	50	ResourceManager 启动的处理 ApplicationMaster RPC 请求的 Handler 数目
yarn.resourcemanager.nodemanager.heartbeat-interval-ms	1000	NodeManager 心跳间隔 (单位: 毫秒)

表 B-7 MapReduce 客户端设置

参数名称	默认值	说 明
mapreduce.framework.name	local	MapReduce 的运行环境, 需将之设置为 yarn
mapreduce.job.reduce.shuffle.consumer.plugin.class	org.apache.hadoop.mapreduce.task.reduce.Shuffle	Shuffle 实现, 必须实现接口 org.apache.hadoop.mapred.ShuffleConsumerPlugin

MapReduce Job History Server (可以是任意节点) 如表 B-8 所示。

表 B-8 MapReduce Job History Server

参数名称	默认值	说 明
mapreduce.jobhistory.address	0.0.0.0:10020	MapReduce Job History Server 的 IPC 地址
mapreduce.jobhistory.webapp.address	0.0.0.0:19888	MapReduce Job History Server 的 Web 地址

用户提交作业时可根据个性化设置 (也可写入 mapred-site.xml, 将这些参数值设置为默认值), 具体如表 B-9 所示。

表 B-9 个性化设置

参数名称	默认值	说 明
mapreduce.job.name	—	作业名称
mapreduce.job.priority	NORMAL	作业优先级
yarn.app.mapreduce.am.resource.mb	1536	MR ApplicationMaster 占用的内存量
yarn.app.mapreduce.am.resource.cpu-vcores	1	MR ApplicationMaster 占用的虚拟 CPU 个数
mapreduce.am.max-attempts	2	MR ApplicationMaster 最大失败尝试次数
mapreduce.map.memory.mb	1024	每个 Map Task 需要的内存量
mapreduce.map.cpu.vcores	1	每个 Map Task 需要的虚拟 CPU 个数
mapreduce.map.maxattempts	4	Map Task 最大失败尝试次数

(续)

参数名称	默认值	说 明
mapreduce.reduce.memory.mb	1024	每个 Reduce Task 需要的内存量
mapreduce.reduce.cpu.vcores	1	每个 Reduce Task 需要的虚拟 CPU 个数
mapreduce.reduce.maxattempts	4	Reduce Task 最大失败尝试次数
mapreduce.map.speculative	false	是否对 Map Task 启用推测执行机制
mapreduce.reduce.speculative	false	是否对 Reduce Task 启用推测执行机制
mapreduce.job.queue.name	default	作业提交到的队列
mapreduce.task.io.sort.mb	100	任务内部排序缓冲区大小
mapreduce.map.sort.spill.percent	0.8	Map 阶段溢写文件的阈值（排序缓冲区大小的百分比）
mapreduce.reduce.shuffle.parallelcopies	5	Reduce Task 启动的并发复制数据的线程数目

注意，MRv2 重命名了 MRv1 中的所有配置参数，但兼容 MRv1 中的旧参数，只不过会打印一条警告日志提示用户参数过期。MapReduce 新旧参数对照表可参考 Java 类 `org.apache.hadoop.mapreduce.util.ConfigUtil`，举例如表 B-10 所示。

表 B-10 MapReduce 新旧参数对照表（部分）

过期参数名	新参数名
mapred.job.name	mapreduce.job.name
mapred.job.priority	mapreduce.job.priority
mapred.job.queue.name	mapreduce.job.queue.name
mapred.map.tasks.speculative.execution	mapreduce.map.speculative
mapred.reduce.tasks.speculative.execution	mapreduce.reduce.speculative
io.sort.factor	mapreduce.task.io.sort.factor
io.sort.mb	mapreduce.task.io.sort.mb

附录 C Hadoop Shell 命令介绍

在 Hadoop 1.0 中，所有模块混合在一起，比如所有管理功能由 `hadoop` 脚本负责，所有 JAR 包存放在同一个目录下等，而 Hadoop 2.0 则改变了这种方式，它将不同模块或者分支完全独立出来。在 Hadoop 2.0 部署目录中，`bin` 目录下有四个常见 Shell 脚本：`hadoop`、`hdfs`、`yarn` 和 `mapred`，它们分别负责 `common`、`HDFS`、`YARN` 和 `MapReduce` 四个模块的管理工作，用户可通过输入“`script --help`”查看该脚本的用法。比如要查看 `yarn` 脚本的用法，可输入“`bin/yarn --help`”。下面分别介绍这几个脚本的用法。

(1) `hadoop` 脚本

该脚本能够向后兼容 Hadoop 1.0 中同名脚本的所有功能，但如果使用 `HDFS` 和 `MapReduce` 相关的命令，则会提示命令已过期，目前，`hadoop` 脚本友好支持（不提示“已过期”）的命令可以通过如下命令查看：

```
dongxicheng@yarn001:/opt/hadoop/hadoop-2.0$ bin/hadoop --help
```

用法：

```
hadoop [--config confdir] COMMAND
```

其中 `COMMAND` 可以为：

- `fs`：运行一个通用的 filesystem 客户端。
- `version`：打印 Hadoop 版本信息。
- `jar <jar>`：运行一个 JAR 包文件。
- `checknative [-a|-h]`：检查本地 Hadoop 库和压缩库的可用性。
- `distcp <srcurl> <desturl>`：递归复制文件或者目录。
- `archive -archiveName NAME -p <parent path> <src>* <dest>`：创建一个 archive 文件。
- `classpath`：打印获取 Hadoop JAR 包和其他库文件所需的 `CLASSPATH` 路径。
- `daemonlog`：获取或者设置服务 LOG 级别。
- `CLASSNAME`：运行一个名字为 `CLASSNAME` 的类。

读者可依次尝试运行这些命令。有的命令进一步拥有自己的参数，当用户输入命令并按下回车后，`hadoop` 脚本会进一步给出使用方法的提示。比如读者可使用以下命令查看 `fs` 命令支持的所有参数：

```
bin/hadoop fs
```

读者也可以通过以下命令查看 Hadoop 版本信息：

```
bin/hadoop version
```

为了向后兼容性，以下这些两类命令也是支持的，但会提示已过期：

□ HDFS 命令主要有：namenode、secondarynamenode、datanode、dfs、dfsadmin、fsck、balancer、fetchdt、oiv、dfsgroups。

□ MapReduce 命令主要有：pipes、job、queue、mrgroups、mradmin、jobtracker、tasktracker
Hadoop 鼓励用户在后面介绍的 hdfs 和 mapred 两个脚本中使用这些命令。

(2) hdfs 脚本

hdfs 脚本可同时供普通用户和管理员使用，它包了少量普通用户可以直接使用的命令，比如 dfs、groups，而大部分命令只有管理员（可通过参数设置管理员列表）才有权限使用。相比于 Hadoop 1.0 中的 HDFS，Hadoop 2.0 中的 HDFS 更加强大，它增加了 Federation、HA 等特性，这意味着 Hadoop 2.0 中的 hdfs 脚本提供的功能更多。

用户可通过以下命令查看 hdfs 的用法：

```
dongxicheng@yarn001:/opt/hadoop/hadoop-2.0$ bin/hdfs --help
```

用法：

```
hdfs [--config confdir] COMMAND
```

其中 COMMAND 可以为：

□ dfs：运行一个 Hadoop 支持的文件系统中的命令（比如创建一个目录）。

□ namenode -format：格式化 DFS 文件系统。

□ secondarynamenode：运行 DFS secondary namenode。

□ namenode：运行 DFS namenode。

□ journalnode：运行 DFS journalnode。

□ zkfc：运行 ZK Failover Controller daemon（Namenode HA 新加入的）。

□ datanode：运行一个 DFS datanode。

□ dfsadmin：运行一个 DFS 管理员客户端。

□ haadmin：运行一个 DFS HA 管理员客户端。

□ fsck：运行一个 DFS 文件系统检测工具。

□ balancer：集群数据均衡工具（数据重分布以达到数据均衡的目的）。

□ jmxget：从 NameNode 或者 DataNode 上获取的 JMX 值。

□ oiv：一个可以离线查看 fsiamge 信息的工具。

□ oev：一个可以离线查看 edits 信息的工具。

□ fetchdt：从 NameNode 上获取一个授权令牌。

□ getconf：从配置文件中获取配置属性值。

□ groups：查看某个用户所属的用户组。

(3) mapred 脚本

在 Hadoop 2.0 中，MapReduce 应用程序相关的所有库已经变成了客户端程序，这也意味着与 MapReduce 相关的任务操作都可由普通用户控制，而 mapred 脚本正是为普通用户管理 MapReduce 作业而提供的。用户可通过以下命令查看 hdfs 用法：

```
dongxicheng@yarn001:/opt/hadoop/hadoop-2.0$ bin/mapred --help
```

用法:

```
mapred [--config confdir] COMMAND
```

其中 COMMAND 可以为:

- pipes: 运行一个 Pipes 作业。
- job: 管理 MapReduce 作业, 比如列出所有作业列表、杀死一个作业等。
- queue: 获取队列信息。
- classpath: 打印运行 MapReduce 子命令所需的 classpath。
- groups: 获取用户所属用户组。
- historyserver: 以独立服务的方式运行 job history server。
- distcp <srcurl> <desturl>: 递归复制文件或者目录。
- archive -archiveName NAME -p <parent path> <src>* <dest>: 创建 archive 文件。

以上大部分命令均拥有自己的参数, 当用户输入该命令后, Hadoop 将进一步给出提示, 比如输入 “bin/mapred pipes” 可提示 pipes 命令的用法。

(4) yarn 脚本

同 hdfs 脚本一样, yarn 脚本可同时供普通用户和管理员使用, 它包了少量普通用户可以直接使用的命令, 比如 jar、logs 等, 而大部分命令只有管理员有权限使用。

用户可通过以下命令查看 yarn 用法:

```
dongxicheng@yarn001:/opt/hadoop/hadoop-2.0$ bin/yarn --help
```

用法:

```
yarn [--config confdir] COMMAND
```

其中 COMMAND 可以为:

- resourcemanager: 运行 ResourceManager。
- nodemanager: 运行一个 NodeManager。
- rmadmin: 管理员工具 (动态更新信息)。
- version: 打印版本信息。
- jar <jar>: 运行 JAR 文件。
- logs: 获取 container 日志。
- classpath: 打印获取 Hadoop JAR 包和其他库文件所需的 CLASSPATH 路径。
- daemonlog: 获取或者设置服务 LOG 级别。
- CLASSNAME: 运行一个名字为 CLASSNAME 的类。

除了 bin 目录下的这些脚本, sbin 目录下也有一些管理脚本, 这些脚本是对 bin 目录下脚本的进一步封装, 通常只有管理员才有权限执行这些管理脚本, 有兴趣的读者可以自行尝试使用这些脚本。

附录 D 参考资料

第 1 章

- [1] Hadoop SVN 地址: <http://svn.apache.org/repos/asf/hadoop/common/branches/>.
- [2] Apache 官方主页: <http://hadoop.apache.org/releases.html>.

第 2 章

- [1] CDH3 下载地址为: <http://archive.cloudera.com/cdh/3/>.
- [2] CDH4 下载地址为: <http://archive.cloudera.com/cdh4/cdh/4/>.
- [3] Mesos 官方网址: <http://incubator.apache.org/mesos/>.
- [4] Torque 官方网址: <http://www.adaptivecomputing.com/products/open-source/torque/>.
- [5] HDFS-200: In HDFS, sync() not yet guarantees data available to the new readers.
- [6] HDFS-265: Revisit append.
- [7] HDFS-RAID: <http://wiki.apache.org/hadoop/HDFS-RAID>.
- [8] HDFS-503: Implement erasure coding as a layer on HDFS.
- [9] HDFS-245: Create symbolic links in HDFS.
- [10] HADOOP-4487: Security features for Hadoop.
- [11] MAPREDUCE-279: Map-Reduce 2.0.
- [12] HDFS-1052: HDFS scalability with multiple namenodes.
- [13] HDFS-1623: High Availability Framework for HDFS NN.
- [14] <http://www.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/>.
- [15] HADOOP-6332: Large-scale Automated Test Framework.
- [16] MAPREDUCE-1084 : Implementing aspects development and fault injection framework for MapReduce.
- [17] Spark 官方首页: <http://www.spark-project.org/>.
- [18] YARN-3: Add support for CPU isolation/monitoring of containers.
- [19] <http://wiki.apache.org/hadoop/PoweredByYarn>.

第 3 章

- [1] Thrift 主页: <http://thrift.apache.org/>.
- [2] Protocol Buffers 主页: <http://code.google.com/p/protobuf/>.
- [3] Avro 主页: <http://avro.apache.org/>.

- [4] HADOOP-7347: IPC Wire Compatibility.
- [5] MAPREDUCE-2930: Generate state graph from the State Machine Definition.

第 4 章

- [1] YARN-103: Add a yarn AM - RM client module.
- [2] YARN-422: Add NM client library.
- [3] YARN-314 : Schedulers should allow resource requests of different sizes at the same priority and location.

第 5 章

- [1] Haml 框架主页: <http://haml.info/>.
- [2] MAPREDUCE-2399: The embedded web framework for MAPREDUCE-279.
- [3] YARN-128: RM Restart.
- [4] YARN-149: ResourceManager (RM) High-Availability (HA).
- [5] YARN-353: Add Zookeeper-based store implementation for RMStateStore.
- [6] YARN-291: Dynamic resource Configuration.
- [7] HADOOP-9621: Document/analyze current security model.

第 6 章

- [1] http://hadoop.apache.org/docs/stable/hod_scheduler.html.
- [2] <http://www.adaptivecomputing.com/products/open-source/torque/>.
- [3] YARN-137: Change the default scheduler to the CapacityScheduler.
- [4] Dominant Resource Fairness: Fair Allocation of Multiple Resources Types. A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, NSDI 2011, March 2011.
- [5] Max-Min Fairness (Wikipedia): http://en.wikipedia.org/wiki/Max-min_fairness.
- [6] http://hadoop.apache.org/docs/stable/capacity_scheduler.html.
- [7] http://hadoop.apache.org/docs/stable/fair_scheduler.html.
- [8] MAPREDUCE-1380: Adaptive Scheduler.
- [9] MAPREDUCE-1439: Learning Scheduler.
- [10] Thomas Sandholm and Kevin Lai. Dynamic proportional share scheduling in hadoop. In JSSPP '10: 15th Workshop on Job Scheduling Strategies for Parallel Processing, 2010.

第 7 章

- [1] MAPREDUCE-3143: Complete aggregation of user-logs spit out by containers onto DFS.
- [2] YARN-321: Generic application history service.
- [3] <http://en.wikipedia.org/wiki/Cgroups>.

- [4] <http://lxc.sourceforge.net/>.
- [5] YARN-2: Enhance CS to schedule accounting for both memory and cpu cores.
- [6] <https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>.
- [7] <https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt>.
- [8] 《红帽企业版 Linux 6 资源管理指南》.

第 8 章

- [1] MAPREDUCE-2405 : MR-279: Implement uber-AppMaster (in-cluster LocalJobRunner for MRv2).
- [2] HADOOP-3245: Provide ability to persist running jobs.(extend HADOOP-1876).
- [3] MAPREDUCE-4049: plugin for generic shuffle service.
- [4] MAPREDUCE-5108 : Changes needed for Binary Compatibility for MR applications via YARN.

第 9 章

- [1] Oozie 主页: <http://oozie.apache.org/>.
- [2] Cascading 主页: <http://www.cascading.org/>.
- [3] <http://hortonworks.com/blog/apache-hive-0-11-stinger-phase-1-delivered/>.
- [4] Azkaban 主页: <http://data.linkedin.com/opensource/azkaban>.
- [5] <https://issues.apache.org/jira/browse/TEZ>.

第 10 章

- [1] Storm Wiki: <https://github.com/nathanmarz/storm/wiki>.
- [2] Yahoo!s4: <http://incubator.apache.org/s4/>.
- [3] Storm 实例: <https://github.com/nathanmarz/storm-starter>.
- [4] Storm On YARN: <https://github.com/yahoo/storm-yarn>.
- [5] <http://spark-project.org/>.
- [6] kryo 序列化器: <http://code.google.com/p/kryo/>.
- [7] storm 简介: <http://www.searchtb.com/2012/09/introduction-to-storm.html>.
- [8] 徐明明的博客: <http://xumingming.sinaapp.com/category/storm/>.

第 11 章

- [1] Corona 源代码: <https://github.com/facebook/hadoop-20/tree/master/src/contrib/corona>.
- [2] Under the Hood: Scheduling MapReduce jobs more efficiently with Corona.

第12章

- [1] Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R. Katz, S. Shenker and I. Stoica, NSDI 2011, March 2011.
- [2] Dpark: <https://github.com/douban/dpark/>.
- [3] Dominant Resource Fairness: Fair Allocation of Multiple Resources Types. A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, NSDI 2011, March 2011.
- [4] Libprocess 代码: <https://github.com/3rdparty/libprocess>.

第13章

- [1] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In Proc. EuroSys (2013).
- [2] <https://issues.apache.org/jira/browse/GIRAPH-13>.
- [3] MAPREDUCE-2911: Hamster: Hadoop And Mpi on the same cluSTER.
- [4] Weave 主页: <http://continuity.github.io/weave/>.
- [5] Kitten 主页: <https://github.com/cloudera/kitten>.