HZ BOOKS
华章科技

多位Spark的贡献者和专家联袂推荐，详细剖析Spark内核各个模块并辅以相应源码解析的著作。以源码为基础，全面分析Spark内核各个模块的设计思想和实现原理，深入理解其内部运作机制乃至实现细节。帮助Spark领域的从业人员全面掌握Spark核心技术，进而在应用开发中做到游刃有余和性能调优时做到有的放矢。

大数据技术丛书

Spark Internals
Design and Implement Principle of Spark Core

Spark技术内幕

深入解析Spark内核架构设计与实现原理

张安站◎著

机械工业出版社
China Machine Press

大数据技术丛书

Spark Internals
Design and Implement Principle of Spark Core

Spark技术内幕

深入解析Spark内核架构设计与实现原理

张安站◎著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Spark 技术内幕：深入解析 Spark 内核架构设计与实现原理 / 张安站著 . —北京：机械工业出版社，2015.8
（大数据技术丛书）

ISBN 978-7-111-50964-6

I. S… II. 张… III. 数据处理软件 IV. TP274

中国版本图书馆 CIP 数据核字（2015）第 170549 号

Spark 是不断壮大的大数据分析解决方案家族中备受关注的新成员。它不仅为分布式数据集的处理提供了一个有效框架，而且以高效的方式处理分布式数据集。它支持实时处理、流处理和批处理，提供了统一的解决方案，因此极具竞争力。本书以源码为基础，深入分析 Spark 内核的设计理念和架构实现，系统讲解各个核心模块的实现，为性能调优、二次开发和系统运维提供理论支持，为更好地使用 Spark Streaming、MLlib、Spark SQL 和 GraphX 等奠定基础。

Spark 技术内幕

深入解析 Spark 内核架构设计与实现原理

出版发行：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：李 艺
责任校对：殷 虹
印 刷：北京瑞德印刷有限公司
版 次：2015 年 9 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16
印 张：13.5
书 号：ISBN 978-7-111-50964-6
定 价：49.00 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）88379426 88361066
购书热线：（010）68326294 88379649 68995259
投稿热线：（010）88379604
读者信箱：hzit@hzbook.com

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

Forword 序

Apache Spark项目的高速发展超出了很多人的预期。在2009年到2013年，Spark还是UC Berkeley大学AMPLab的一个研究项目，因为其架构设计得简洁和高效，逐渐吸引了工业界和学术界的广泛关注。我还记得2013年2月份在Santa Clara召开的Strata Conference上，虽然是长达一整天的Spark技术培训，大厅里还是人满为患，大家都在认真学习这种新的计算框架，并被其高速的性能所折服。尽管在随后的一年半时间内，主流的Hadoop厂商并没有接受这个新框架：Cloudera在忙着开发自己的Impala引擎，Hortonworks经过评估后认为可以改造Map/Reduce来实现类似Spark的DAG机制，也就是后来的Tez，而MapR还在纠结是否要全力投入Drill项目。但在2014年的夏天，第二次Spark Summit召开时，已经在Spark上积累大量开发者和用户，从互联网到传统行业，甚至是生物神经学家都用Spark来分析脑活动的数据。在这次会议上，大部分的Hadoop厂商以及应用开发商开始接受Spark，并宣布支持Spark作为Hadoop上的另一个计算引擎。自此以后，Spark的被接受程度飞速提高。到2014年10月份，几乎所有的大数据厂商都宣布支持Spark，Spark作者们创办的DataBricks公司也宣布认证了50多个以Spark为基础的应用系统。而到了2015年，大家在谈论的是Spark即将全面替代Hadoop中的Map/Reduce。

星环科技从2013年创业的第一天，就开始改造Spark引擎来开发批处理和交互式分析引擎。今天在星环的全系列产品中，已经几乎看不到Map/Reduce计算框架。星环科技已经证明了在所有Map/Reduce擅长的领域，Spark计算引擎都可以更高效地执行，性能可以提升数倍到数十倍，并且可以7x24稳定运行。这也从侧面证明了Spark引擎的潜力。

本书详细剖析了 Spark 核心引擎的源代码及其工作原理，内容翔实准确，也是我目前看到的一本比较全面解析 Spark Core 的不可多得的好书。特别是有志于 Spark 内核开发的研发人员，仔细阅读本书并研读代码，将起到事半功倍的效果。

孙元浩

星环科技创始人兼 CTO

2015 年 8 月上海

Preface 前　言

诞生于2005年的Hadoop解决了大数据的存储和计算问题，已经成为大数据处理的事实标准。但是，随着数据规模的爆炸式增长和计算场景的丰富细化，使得Hadoop越来越难以满足用户的需求。针对不同的计算场景，开源社区和各大互联网公司也推出了各种大数据分析的平台，旨在满足特定应用场景下的计算需求。但是，众多的平台使得用户不得不为平台开发类似的策略，这增加了运维开发成本。

2009年诞生于AMPLab的Spark，它的设计目标就是为大数据的多种计算场景提供一个通用的计算引擎，同时解决大数据处理的4V难题，即Volume（海量）、Velocity（快速）、Variety（多样）、Value（价值）。正如Spark的核心作者之一的Ion Stoica所说，“The goal is to build a new generation of data analytics software，to be used across academia and industry。”Hadoop之父Doug Cutting也说过，MapReduce引擎将被Spark替代（Use of MapReduce engine for Big Data projects will decline，replaced by Apache Spark）。可以说，Spark自诞生之日起就得到了广泛的关注，也是近年来开源社区最活跃的项目之一。Spark的1.X版本的每次发布，都包含了数百位贡献者的上千次提交。最新的版本是发布于2015年6月11日的1.4.0，是迄今为止Spark最大的一次版本发布，涵盖了210位开发者的贡献。

Spark得到了众多大数据公司的支持，这些公司包括Hortonworks、IBM、Intel、Cloudera、MapR、Pivotal和星环科技；Spark也被百度、阿里、腾讯、京东、携程、优酷土豆等互联网公司应用到多种不同的计算场景中，并且在实际的生产环境中获得了很多收益。当前百度的Spark已应用于凤巢、大搜索、直达号、百度大数据等业务；

阿里利用 GraphX 构建了大规模的图计算和图挖掘系统，实现了很多生产系统的推荐算法；腾讯 Spark 集群达到 8000 台的规模，是当前已知的世界上最大的 Spark 集群。

但是，当前并没有一本系统介绍 Spark 内核实现原理的书，而 Spark 内核是 Spark SQL、Spark Streaming、MLlib、GraphX 等多个模块的基础，这些模块最终的计算执行都是由内核模块完成的。为了在应用开发中做到游刃有余，在性能调优时做到有的放矢，需要了解内核模块的实现原理。笔者从 Spark 发布版本 0.8.1 时开始关注 Spark，并深入学习内核模块的架构实现原理。Spark 在 1.0 发布后，内核模块趋于稳定，虽然内核模块依旧会有不断地改进和完善，但是整体的设计思想和实现方法是不会变的，因此笔者决定为 Spark 社区的用户和关注者写一本书，详细介绍 Spark 内核模块的实现原理。最终，笔者基于 Spark 1.2.0 版本完成了本书。

写作是一件严肃的事情，同样是一份苦差事，尤其是在工作比较忙的时候。本书在半年前就完成了基本的框架，但是随后又对本书进行了多次修改和完善。笔者认为，对一本架构分析的书，一个最基本的要求就是基于源码如实描述系统的实现，能做到这点就是一本及格的书；如果能做到分析这个架构的好坏，指出架构改进的方案，那么就是一本质量比较好的书；如果能高屋建瓴地进行再次抽象，指出类似架构不同实现的优劣，抽象出一些理论，那么这就是一本质量上乘，可以当作教科书的书。我深知自己的能力水平，希望这本书最起码是一本及格的书，即能基于源码如实描述系统的实现，对那些希望深入学习 Spark 架构实现的同仁有所帮助。

目标读者

本书适合大数据领域的架构师、运维人员，尤其是 Spark 领域的从业人员阅读，也适合作为研究生和高年级的本科生大数据领域分布式架构具体实现原理的参考资料。

内容概述

第 1 章介绍了 Spark 的技术背景和特点，给出了架构的整体概述，并简单介绍了 Spark 的生态圈。

第 2 章介绍了 Spark 源码如何获取和学习环境如何搭建。

第 3 章是 RDD 的详细介绍，介绍了 RDD 的定义和 Spark 对于 DAG 的实现，最后通过 RDD 计算的详细介绍，讲解了 Spark 对于计算的实现原理。

第 4 章详细介绍任务调度的实现，包括如何通过 DAG 来生成计算任务，最后通过“Word Count”来加深对这个实现过程的理解。

第 5 章介绍了 Spark 的运行模式，尤其是 Standalone 模式。Standalone 是 Spark 自身实现的资源管理和调度的模块，这里会详细介绍它的实现原理。

第 6 章是 Executor 模块的详细讲解。Executor 是最终执行计算任务的单元，这章将详细介绍 Executor 的实现原理，包括 Executor 的分配、Task 在 Executor 的详细执行过程。

第 7 章详细介绍了 Spark 对于 Shuffle 的实现原理，包括基于 Hash 和基于排序的实现。除了详细阐述基于 Hash 和排序的 Shuffle 写和 Shuffle 读之外，还介绍了 Shuffle Pluggable 框架，为需要实现特定 Shuffle 逻辑的读者介绍其实现原理。

第 8 章详细介绍了 Spark 的 Storage 模块，在详细介绍了模块的架构后详细解析了不同存储级别的实现细节。

第 9 章介绍了 Spark 在百度、腾讯和阿里等国内互联网领域的应用现状。

致谢

本书在写作的过程中，得到了很多朋友、同仁的帮助和支持，在此表示衷心感谢！

感谢七牛云的技术总监陈超先生、蘑菇街的资深架构师刘旭晖先生、百度公司的高级架构师柴华先生、Databricks 软件工程师连城先生在百忙之中为本书审稿，并提出了很多宝贵的修改意见。尤其感谢星环科技的创始人兼 CTO 孙元浩先生对本书的完成给予了很多的支持，还在百忙之中为本书作序。感谢华为诺亚方舟实验室的董振华博士，在 Spark 上做机器学习方面给了我很多指导。

感谢百度上海研发中心网页搜索部的同事们。在一年多的工作中，笔者得到了很多同事的指导、支持和帮助，尤其感谢曲晶莹、吴永巍、汪韬、葛俊、刘桐仁、段雪涛、周波涛、马鑫云、李战平、杨大毛、朱晓阳、赵鹏程等。

感谢机械工业出版社的姚蕾编辑，她不但积极策划和推动本书的出版，还容忍我蜗牛般的写作速度；感谢她在这一年多的时间中给予的理解与支持。感谢机械工业出版社的李艺编辑为本书做了非常辛苦和专业的编辑工作。

还要感谢我的家人一直以来对我的支持和宽容。感谢父亲、母亲和三个姐姐，你们是我漫漫求学路的最强大支柱和后盾；感谢我的妻子王珊珊，不但在家庭方面付出很多，也为本书的顺利出版做出了很重要的贡献；感谢我的女儿，你的微笑是爸爸消除疲惫的良药。

联系方式

由于本书的写作都是在业余时间完成，加上笔者自身水平有限，错误在所难免，敬请读者谅解，如果有任何问题，可以通过下列方式与 Spark 的关注者和使用者进行交流沟通：

Spark 架构实现原理交流 QQ 群：473842835

Spark 用户使用交流 QQ 群：473853269

也可以直接与笔者联系：

邮箱：anzhsoft@gmail.com

新浪微博：@anzhsoft

个人博客：http://blog.csdn.net/anzhsoft

Contents 目　录

序

前言

第 1 章　Spark 简介 ····· 1

1.1　Spark 的技术背景 ····· 1

1.2　Spark 的优点 ····· 2

1.3　Spark 架构综述 ····· 4

1.4　Spark 核心组件概述 ····· 5

1.4.1　Spark Streaming ····· 5

1.4.2　MLlib ····· 6

1.4.3　Spark SQL ····· 7

1.4.4　GraphX ····· 8

1.5　Spark 的整体代码结构规模 ····· 8

第 2 章　Spark 学习环境的搭建 ····· 9

2.1　源码的获取与编译 ····· 9

2.1.1　源码获取 ····· 9

2.1.2　源码编译 ····· 10

2.2　构建 Spark 的源码阅读环境 ····· 11

2.3　小结 ····· 15

第 3 章　RDD 实现详解……16

3.1　概述……16

3.2　什么是 RDD……17

3.2.1　RDD 的创建……19

3.2.2　RDD 的转换……20

3.2.3　RDD 的动作……22

3.2.4　RDD 的缓存……23

3.2.5　RDD 的检查点……24

3.3　RDD 的转换和 DAG 的生成……25

3.3.1　RDD 的依赖关系……26

3.3.2　DAG 的生成……30

3.3.3　Word Count 的 RDD 转换和 DAG 划分的逻辑视图……30

3.4　RDD 的计算……33

3.4.1　Task 简介……33

3.4.2　Task 的执行起点……33

3.4.3　缓存的处理……35

3.4.4　checkpoint 的处理……37

3.4.5　RDD 的计算逻辑……39

3.5　RDD 的容错机制……39

3.6　小结……40

第 4 章　Scheduler 模块详解……41

4.1　模块概述……41

4.1.1　整体架构……41

4.1.2　Scheduler 的实现概述……43

4.2　DAGScheduler 实现详解……45

4.2.1　DAGScheduler 的创建……46

4.2.2　Job 的提交……48

4.2.3　Stage 的划分……49

4.2.4　任务的生成……54

4.3 任务调度实现详解……57
4.3.1 TaskScheduler 的创建……57
4.3.2 Task 的提交概述……58
4.3.3 任务调度具体实现……61
4.3.4 Task 运算结果的处理……65
4.4 Word Count 调度计算过程详解……72
4.5 小结……74

第 5 章 Deploy 模块详解……76

5.1 Spark 运行模式概述……76
5.1.1 local……77
5.1.2 Mesos……78
5.1.3 YARN……82
5.2 模块整体架构……86
5.3 消息传递机制详解……87
5.3.1 Master 和 Worker……87
5.3.2 Master 和 Client……89
5.3.3 Client 和 Executor……91
5.4 集群的启动……92
5.4.1 Master 的启动……92
5.4.2 Worker 的启动……96
5.5 集群容错处理……98
5.5.1 Master 异常退出……98
5.5.2 Worker 异常退出……99
5.5.3 Executor 异常退出……101
5.6 Master HA 实现详解……102
5.6.1 Master 启动的选举和数据恢复策略……103
5.6.2 集群启动参数的配置……105
5.6.3 Curator Framework 简介……106
5.6.4 ZooKeeperLeaderElectionAgent 的实现……109
5.7 小结……110

第 6 章　Executor 模块详解 ····· 112

6.1　Standalone 模式的 Executor 分配详解 ····· 113

6.1.1　SchedulerBackend 创建 AppClient ····· 114

6.1.2　AppClient 向 Master 注册 Application ····· 116

6.1.3　Master 根据 AppClient 的提交选择 Worker ····· 119

6.1.4　Worker 根据 Master 的资源分配结果创建 Executor ····· 121

6.2　Task 的执行 ····· 122

6.2.1　依赖环境的创建和分发 ····· 123

6.2.2　任务执行 ····· 125

6.2.3　任务结果的处理 ····· 128

6.2.4　Driver 端的处理 ····· 130

6.3　参数设置 ····· 131

6.3.1　spark.executor.memory ····· 131

6.3.2　日志相关 ····· 132

6.3.3　spark.executor.heartbeatInterval ····· 132

6.4　小结 ····· 133

第 7 章　Shuffle 模块详解 ····· 134

7.1　Hash Based Shuffle Write ····· 135

7.1.1　Basic Shuffle Writer 实现解析 ····· 136

7.1.2　存在的问题 ····· 138

7.1.3　Shuffle Consolidate Writer ····· 139

7.1.4　小结 ····· 140

7.2　Shuffle Pluggable 框架 ····· 141

7.2.1　org.apache.spark.shuffle.ShuffleManager ····· 141

7.2.2　org.apache.spark.shuffle.ShuffleWriter ····· 143

7.2.3　org.apache.spark.shuffle.ShuffleBlockManager ····· 143

7.2.4　org.apache.spark.shuffle.ShuffleReader ····· 144

7.2.5　如何开发自己的 Shuffle 机制 ····· 144

7.3　Sort Based Write ····· 144

7.4 Shuffle Map Task 运算结果的处理 148
7.4.1 Executor 端的处理 148
7.4.2 Driver 端的处理 150
7.5 Shuffle Read 152
7.5.1 整体流程 152
7.5.2 数据读取策略的划分 155
7.5.3 本地读取 156
7.5.4 远程读取 158
7.6 性能调优 160
7.6.1 spark.shuffle.manager 160
7.6.2 spark.shuffle.spill 162
7.6.3 spark.shuffle.memoryFraction 和 spark.shuffle.safetyFraction 162
7.6.4 spark.shuffle.sort.bypassMergeThreshold 163
7.6.5 spark.shuffle.blockTransferService 163
7.6.6 spark.shuffle.consolidateFiles 163
7.6.7 spark.shuffle.compress 和 spark.shuffle.spill.compress 164
7.6.8 spark.reducer.maxMbInFlight 165
7.7 小结 165

第 8 章 Storage 模块详解 167

8.1 模块整体架构 167
8.1.1 整体架构 167
8.1.2 源码组织结构 170
8.1.3 Master 和 Slave 的消息传递详解 173
8.2 存储实现详解 181
8.2.1 存储级别 181
8.2.2 模块类图 184
8.2.3 org.apache.spark.storage.DiskStore 实现详解 186
8.2.4 org.apache.spark.storage.MemoryStore 实现详解 188
8.2.5 org.apache.spark.storage.TachyonStore 实现详解 189
8.2.6 Block 存储的实现 190

8.3 性能调优……194
8.3.1 spark.local.dir……194
8.3.2 spark.executor.memory……194
8.3.3 spark.storage.memoryFraction……194
8.3.4 spark.streaming.blockInterval……195
8.4 小结……195

第 9 章 企业应用概述……197

9.1 Spark 在百度……197
9.1.1 现状……197
9.1.2 百度开放云 BMR 的 Spark……198
9.1.3 在 Spark 中使用 Tachyon……199
9.2 Spark 在阿里……200
9.3 Spark 在腾讯……200
9.4 小结……201

第 1 章 Chapter 1

Spark 简介

Apache Spark 是一种快速、通用、可扩展的大数据分析引擎。它是不断壮大的大数据分析解决方案家族中备受关注的明星成员，为分布式数据集的处理提供了一个有效框架，并以高效的方式处理分布式数据集。Spark 集批处理、实时流处理、交互式查询与图计算于一体，避免了多种运算场景下需要部署不同集群带来的资源浪费。Spark 在过去的一年中获得了极大关注，并得到广泛应用，Spark 社区也成为大数据领域和 Apache 软件基金会最活跃的项目之一，其活跃度甚至远超曾经只能望其项背的 Hadoop。

1.1 Spark 的技术背景

无论是工业界还是学术界，都已经广泛使用高级集群编程模型来处理日益增长的数据，如 MapReduce 和 Dryad。这些系统将分布式编程简化为自动提供位置感知性调度、容错以及负载均衡，使得大量用户能够在商用集群上分析超大数据集。大多数现有的集群计算系统都是基于非循环的数据流模型。即从稳定的物理存储（如分布式文件系统）中加载记录，记录被传入由一组确定性操作构成的 DAG（Directed Acyclic Graph，有向无环图），然后写回稳定存储。DAG 数据流图能够在运行时自动实现任务

调度和故障恢复。

尽管非循环数据流是一种很强大的抽象方法，但仍然有些应用无法使用这种方式描述。这类应用包括：①机器学习和图应用中常用的迭代算法（每一步对数据执行相似的函数）；②交互式数据挖掘工具（用户反复查询一个数据子集）。基于数据流的框架并不明确支持工作集，所以需要将数据输出到磁盘，然后在每次查询时重新加载，这会带来较大的开销。针对上述问题，Spark 实现了一种分布式的内存抽象，称为**弹性分布式数据集**（Resilient Distributed Dataset，RDD）。它支持基于工作集的应用，同时具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

RDD 提供了一种高度受限的共享内存模型，即 RDD 是只读记录分区的集合，只能通过在其他 RDD 执行确定的转换操作（如 map、join 和 groupBy）而创建，然而这些限制使得实现容错的开销很低。与分布式共享内存系统需要付出高昂代价的检查点和回滚机制不同，RDD 通过 Lineage 来重建丢失的分区：一个 RDD 中包含了如何从其他 RDD 衍生所必需的相关信息，从而不需要检查点操作就可以重建丢失的数据分区。尽管 RDD 不是一个通用的共享内存抽象，但它具备了良好的描述能力、可伸缩性和可靠性，能够广泛适用于数据并行类应用。

1.2 Spark 的优点

Spark 凭借什么在众多的大数据分析处理平台中脱颖而出呢？这得益于 Spark 的几个主要优点，具体如下所示。

第一个优点是速度。与 Hadoop 的 MapReduce 相比，Spark 基于内存的运算要快 100 倍以上（如图 1-1 所示）；而基于硬盘的运算也要快 10 倍以上。Spark 实现了高效的 DAG 执行引擎，可以通过基于内存来高效处理数据流。

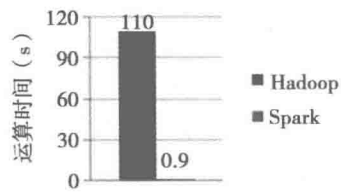


图 1-1 逻辑回归在 Hadoop、Spark 的运算速度对比

第二个优点就是易用。Spark 支持 Java、Python 和 Scala 的 API，还支持超过 80 种高级算法，使用户可以快速构建不同的应用。而且 Spark 支持交互式的 Python 和 Scala 的 shell，这意味着可以非常方便地在这些 shell 中使用 Spark 集群来验证解决问题的方法，而不是像以前一样，需要打包、上传集群、验证等。这对于原型开发非常重要。

第三个优点是通用性。Spark 提供了统一的解决方案。Spark 可以用于批处理、交互式查询（通过 Spark SQL）、实时流处理（通过 Spark Streaming）、机器学习（通过 Spark MLlib）和图计算（通过 Spark GraphX），如图 1-2 所示。这些不同类型的处理都可以在同一个应用中无缝使用。Spark 统一的解决方案非常具有吸引力，毕竟任何公司都想用统一的平台去处理遇到的问题，减少开发和维护的人力成本和部署平台的物力成本。当然还有，作为统一的解决方案，Spark 并没有以牺牲性能为代价。相反，在性能方面，Spark 具有很大的优势。

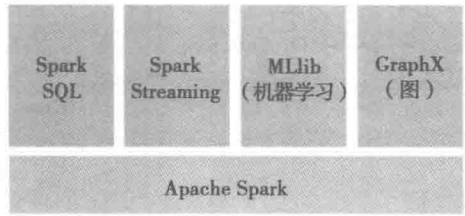


图 1-2　Spark 支持丰富的应用计算场景

第四个优点就是可融合性。Spark 可以非常方便地与其他的开源产品进行融合。比如，Spark 可以使用 Hadoop 的 YARN 和 Apache Mesos 作为它的资源管理和调度器，并且可以处理所有 Hadoop 支持的数据，包括 HDFS、HBase 和 Cassandra 等。这对于已经部署 Hadoop 集群的用户特别重要，因为不需要做任何数据迁移就可以使用 Spark 的强大处理能力。Spark 也可以不依赖于第三方的资源管理和调度器，它实现了 Standalone 作为其内置的资源管理和调度框架，这样进一步降低了 Spark 的使用门槛，使得所有人都可以非常容易地部署和使用 Spark。此外，Spark 还提供了在 EC2 上部署 Standalone 的 Spark 集群的工具。

1.3 Spark 架构综述

Spark 的整体架构如图 1-3 所示。其中，Driver 是用户编写的数据处理逻辑，这个逻辑中包含用户创建的 SparkContext。SparkContext 是用户逻辑与 Spark 集群主要的交互接口，它会和 Cluster Manager 交互，包括向它申请计算资源等。Cluster Manager 负责集群的资源管理和调度，现在支持 Standalone、Apache Mesos 和 Hadoop 的 YARN。Worker Node 是集群中可以执行计算任务的节点。Executor 是在一个 Worker Node 上为某应用启动的一个进程，该进程负责运行任务，并且负责将数据存在内存或者磁盘上。Task 是被送到某个 Executor 上的计算单元。每个应用都有各自独立的 Executor，计算最终在计算节点的 Executor 中执行。

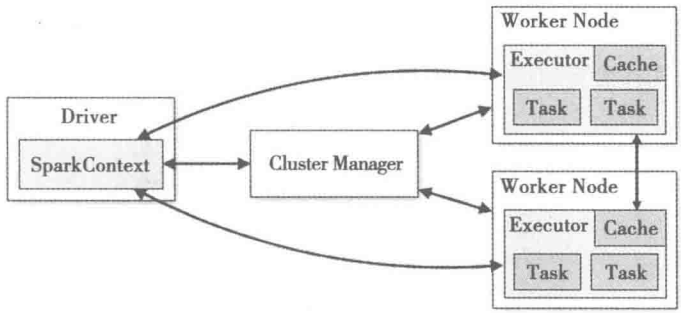


图 1-3 整体架构图

用户程序从最开始的提交到最终的计算执行，需要经历以下几个阶段：

1）用户程序创建 SparkContext 时，新创建的 SparkContext 实例会连接到 Cluster Manager。Cluster Manager 会根据用户提交时设置的 CPU 和内存等信息为本次提交分配计算资源，启动 Executor。

2）Driver 会将用户程序划分为不同的执行阶段，每个执行阶段由一组完全相同的 Task 组成，这些 Task 分别作用于待处理数据的不同分区。在阶段划分完成和 Task 创建后，Driver 会向 Executor 发送 Task。

3）Executor 在接收到 Task 后，会下载 Task 的运行时依赖，在准备好 Task 的执

行环境后，会开始执行 Task，并且将 Task 的运行状态汇报给 Driver。

4）Driver 会根据收到的 Task 的运行状态来处理不同的状态更新。Task 分为两种：一种是 Shuffle Map Task，它实现数据的重新洗牌，洗牌的结果保存到 Executor 所在节点的文件系统中；另外一种是 Result Task，它负责生成结果数据。

5）Driver 会不断地调用 Task，将 Task 发送到 Executor 执行，在所有的 Task 都正确执行或者超过执行次数的限制仍然没有执行成功时停止。

1.4　Spark 核心组件概述

图 1-2 简单介绍了 Spark 支持的丰富的应用计算场景，这是通过在 Spark Core 上构建多个组件来完成的。比如 Spark Streaming 实现了实时流处理；GraphX 实现了图计算；MLlib 实现了很多机器学习算法；Spark SQL 实现了基于 Spark 的交互式查询。

1.4.1　Spark Streaming

Spark Streaming 基于 Spark Core 实现了可扩展、高吞吐和容错的实时数据流处理。现在支持的数据源有 Kafka、Flume、Twitter、ZeroMQ、Kinesis、HDFS、S3 和 TCP socket。处理后的结果可以存储到 HDFS、Database 或者 Dashboard 中，如图 1-4 所示。

图 1-4　Spark Streaming 支持的数据源

Spark Streaming 是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是 Spark，也就是把 Spark Streaming 的输入数据按照批处理尺寸（如 1 秒）分成一段一段的数据（Stream），每一段数据都转换成 Spark 中的 RDD，然后将 Spark Streaming

中对 DStream 的转换操作变为针对 Spark 中对 RDD 的转换操作，将 RDD 经过操作变成中间结果保存在内存中。整个流式计算可以根据业务的需求对中间的结果进行叠加，或者存储到外部设备。上述过程如图 1-5 所示。

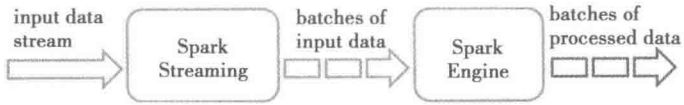


图 1-5 Spark Streaming 的实现原理

Spark Streaming 提供了一套高效、可容错的准实时大规模流式处理框架，它能和批处理及即时查询放在同一个软件栈中，降低学习成本。对于熟悉 Spark 编程的用户，可以用较低的成本学习 Spark Streaming 编程。

1.4.2 MLlib

MLlib 是 Spark 对常用的机器学习算法的实现库，同时含有相关的测试和数据生成器，包括分类、回归、聚类、协同过滤、降维（dimensionality reduction）以及底层基本的优化元素。

在 Spark 1.2.0 中，MLlib 最大的改进是引入了称为 spark.ml 的机器学习工具包，支持了流水线的学习模式，即多个算法可以用不同参数以流水线的形式运行。在工业界的机器学习应用部署过程中，流水线的工作模式是很常见的。新的 ML 工具包使用 Spark 的 SchemaRDD 来表示机器学习的数据集合，提供了 Spark SQL 直接访问的接口。此外，在机器学习的算法方面，增加了两种基于树的方法，即随机森林和梯度增强树。

现在，MLlib 实现了许多常用的算法，与分类和回归相关的算法包括 SVM、逻辑回归、线性回归、朴素贝叶斯分类、决策树等；协同过滤实现了交替最小二乘法（Alternating Least Square，ALS）；聚类实现了 K-means、高斯混合（Gaussian mixture）、Power Iteration Clustering（PIC）、Latent Dirichlet Allocation（LDA）和 Streaming 版本的 K-means；降维实现了 Singular Value Decomposition（SVD）和 Principal Component Analysis（PCA）；频繁模式挖掘（frequent pattern mining）实现

了 FP-growth。

1.4.3　Spark SQL

自从 Spark 1.0 版本的 Spark SQL 问世以来，它最常见的用途之一就是作为一个从 Spark 平台获取数据的渠道。早期用户比较喜爱 Spark SQL 提供的从现有 Apache Hive 表以及流行的 Parquet 列式存储格式中读取数据的支持。之后，Spark SQL 还增加了对其他格式的支持，比如说 JSON。到了 Spark 1.2 版本，Spark 的原生资源与更多的输入源进行整合集成，这些新的整合将随着纳入新的 Spark SQL 数据源 API 而成为可能。Spark SQL 支持的数据源如图 1-6 所示。

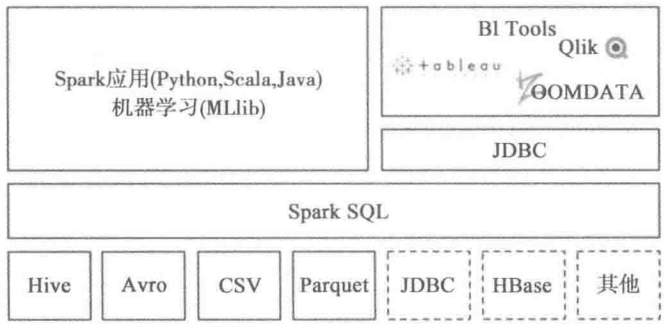


图 1-6　Spark SQL 支持的数据源

数据源 API 通过 Spark SQL 提供了访问结构化数据的可插拔机制。这使数据源有了简便的途径进行数据转换并加入到 Spark 平台中。由 API 提供的密集优化器集合意味着过滤和列修剪在很多情况下都会被运用于数据源。这些综合的优化极大地减少了需要处理的数据量，因此能够显著提高 Spark 的工作效率。

数据源 API 的另一个优点就是不管数据的来源如何，用户都能够通过 Spark 支持的所有语言来操作这些数据。例如，那些用 Scala 实现的数据源，pySpark 用户不需要其他的库开发人员做任何额外的工作就可以直接使用。此外，Spark SQL 可以使用单一接口访问不同数据源的数据。总之，Spark 1.2 提供的这些功能进一步统一了大数据分析的解决方案。

1.4.4 GraphX

Spark GraphX 是 Spark 提供的关于图和图并行计算的 API，它集 ETL、试探性分析和迭代式的图计算于一体，并且在不失灵活性、易用性和容错性的前提下获得了很好的性能。现在 GraphX 已经提供了很多的算法，新的算法也在不断加入，而且很多的算法都是由 Spark 的用户贡献的。

1.5 Spark 的整体代码结构规模

Spark Core 是 Spark 的核心，它体现了 Spark 的核心设计思想。Spark SQL、MLlib、GraphX 和 Spark Streaming 都是基于 Spark Core 的实现和衍生。在 Spark 1.2.0 版本中，Spark Core 包含了近 6.5 万行 Scala 源码。其中，不同模块之间的代码统计见图 1-7。仅代码规模上，看起来好像远没有想象中的那么复杂；但是如果没有借助 Scala 语言的强大表现力，比如使用 C++ 或者 Java 的话，代码规模可能要大得多。还可以从另外一个角度来评估当前的代码规模，Spark 的广泛应用导致新特性的不断加入，使得 Spark 单纯代码的增长速度不断加快。

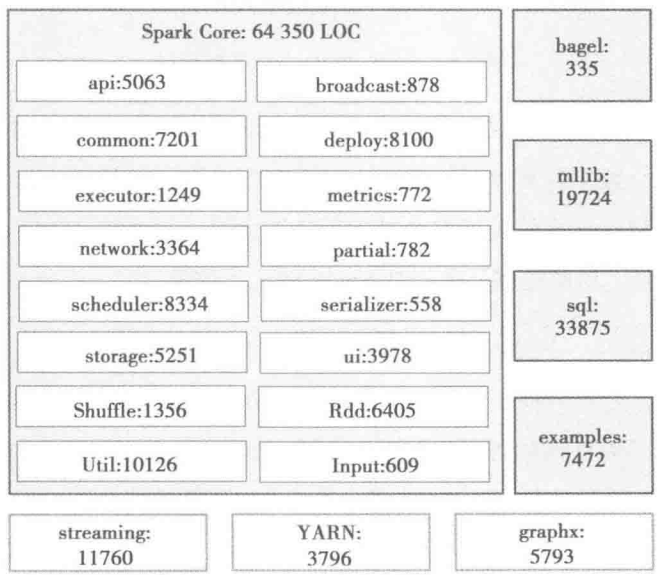


图 1-7 Spark 的整体代码结构图

第 2 章 Chapter 2

Spark 学习环境的搭建

本章主要介绍源码阅读环境的搭建，包括 Spark 源码的获取和编译，安装和部署以及简单的设置和使用。另外，通过订阅 Spark 的邮件组 user@spark.apache.org 可以查看大家使用时遇到的问题和这些问题的解决方法；如果希望贡献社区，可以订阅邮件组 dev@spark.apache.org 来了解社区关于开发的相关讨论。遇到问题或者希望更进一步地了解开发状态，可以访问 Spark JIRA 的主页 https://issues.apache.org/jira/browse/SPARK，了解 Spark 对于某个问题的持续改进，帮助更好地理解架构的演进。

2.1 源码的获取与编译

2.1.1 源码获取

可以到 Spark 官网的 http://spark.apache.org/downloads.html 页面下载最新的 Spark 源码。

Spark 源码在 Github 上的主页地址是 https://github.com/apache/spark，读者也可以通过 git 命令获取最新的源码：git clone https://github.com/apache/spark.git。由于服务

器在美国，因此下载的速度会比较慢，如果中途遇到失败，可以重复执行上述命令直至成功。

2.1.2 源码编译

首先要安装 JDK 6+ 和 Scala 2.10.* 或者 2.11.* 版本。截至本书截稿时，Kafka 库和 JDBC 并不支持 Scala 2.11 的编译，因此请使用 Scala 2.10。

1. maven 编译

Spark 支持多个版本的 Hadoop，包括社区版和商业发行版，通过指定 Hadoop 的版本，可以使 Spark 使用不同版本的 HDFS、YARN。现在可用的编译方式如表 2-1 所示。

表 2-1 可用编译方式

Hadoop 版本	maven 编译命令
Apache Hadoop 1.2.1	mvn -Dhadoop.version=1.2.1 -DskipTests clean package
Cloudera CDH 4.2.0 with MapReduce v1	mvn -Dhadoop.version=2.0.0mr1-cdh-4.2.0 -DskipTestsclean-package
Apache Hadoop 0.23.x	mvn -Phadoop-0.23 -Dhadoop.version=0.23.7 -DskipTests clean package
Apache Hadoop 2.2.X	mvn -Pyarn -Phadoop-2.2 -Dhadoop.version=2.2.0 -DskipTests clean package
Apache Hadoop 2.3.X	mvn -Pyarn -Phadoop-2.3 -Dhadoop.version=2.3.0 -DskipTests clean package
Apache Hadoop 2.4.X or 2.5.X	mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=VERSION -DskipTests clean package
不同版本的 HDFS 和 YARN	mvn -Pyarn -Phadoop-2.3 -Dhadoop.version=2.3.0 -Dyarn.version=2.2.0 -DskipTests clean package

如果在编译过程中遇到如下错误：

```
[ERROR] PermGen space -> [Help 1]
[ERROR] Java heap space -> [Help 1]
```

这个错误是由于内存不足造成的。可以通过设置环境变量 MAVEN_OPTS 为 maven

分配更多的内存来解决：

```
export MAVEN_OPTS="-Xmx4g -XX:MaxPermSize=1g
   -XX:ReservedCodeCacheSize=1g"
```

2. sbt 编译

尽管 maven 是 Spark 官网推荐的编译方式，但是 sbt 的编译速度更胜一筹。因此对于 Spark 的开发者来说，sbt 编译可能是更好的选择。由于 sbt 编译也是基于 maven 的 POM 文件，因此 sbt 的编译参数与 maven 的编译参数是一致的。比如可以使用下面的命令进行编译：

```
build/sbt -Pyarn -Phadoop-2.3 assembly
```

2.2　构建 Spark 的源码阅读环境

查看 Spark 源码最好的 IDE 就是 IntelliJ IDEA，这个也是大部分 Spark 开发者都在使用的。IDEA 分两个版本，一个是免费的社区版，一个是收费的商业发行版。不过 Apache 的贡献者可以免费获得商业发行版的使用权。

当然使用 Eclipse 也是可以的，读者可以访问 http://scala-ide.org/ 下载和安装 Eclipse，在这里就不再赘述。不过笔者认为使用 Eclipse 阅读 Spark 源码的体验远不如使用 IntelliJ IDEA，推荐大家都去尝试，即使您可能对 Eclipse 非常熟悉。

IDEA 的社区版和商业发行版都可以从 https://www.jetbrains.com/idea/download/ 获得。IDEA 有三个版本：Windows 版、MAC OS 版和 Linux 版。下载后直接安装即可。

为了阅读和构建 Spark 的开发环境，需要为 IDEA 安装 Scala 插件。首先单击图 2-1 中红框内的“Configure”。接着单击图 2-2 中的“Plugins”。然后单击左下角的“Install JetBrains plugin”（图 2-3）。在新的界面图 2-4 中输入“Scala”回车，就可以看到 Scala 插件了，直接单击右侧的“Install plugin”就可以安装了。

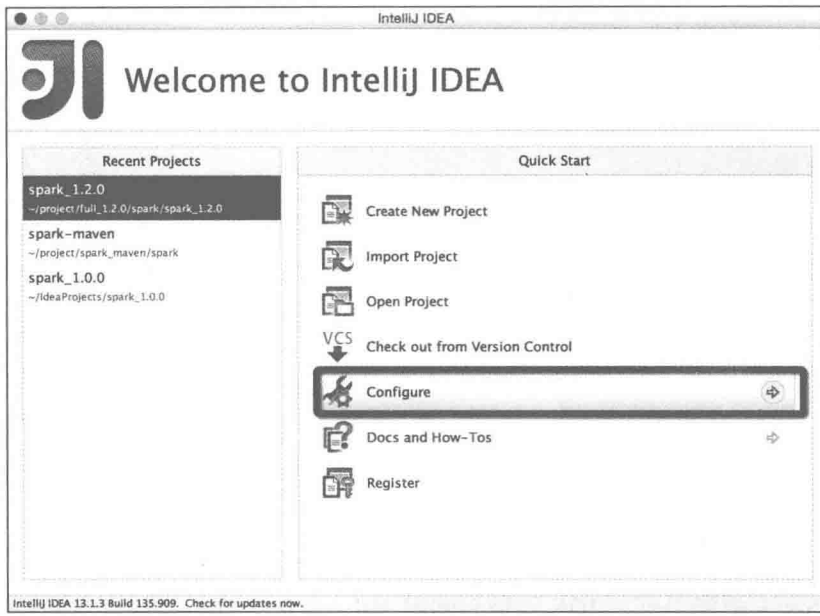


图 2-1 单击“Configure”以安装 Scala 插件

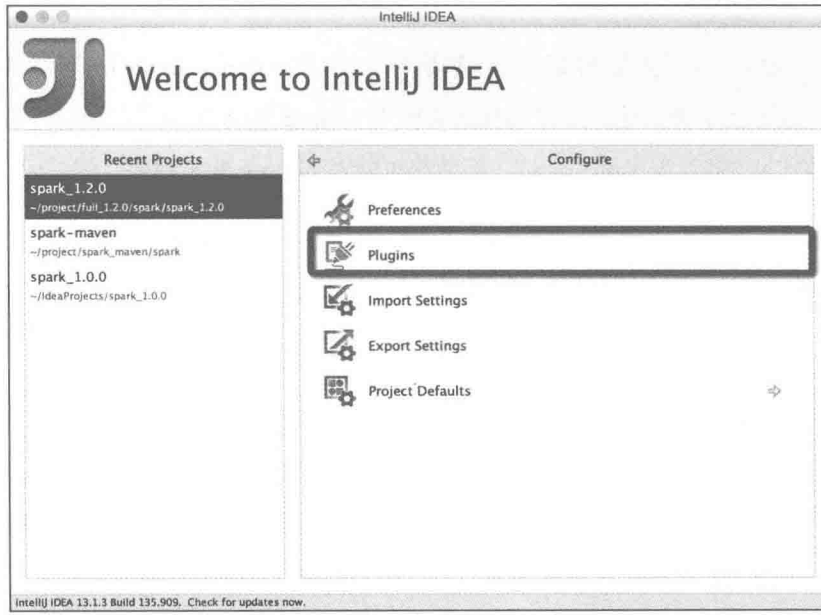


图 2-2 选择“Plugins”

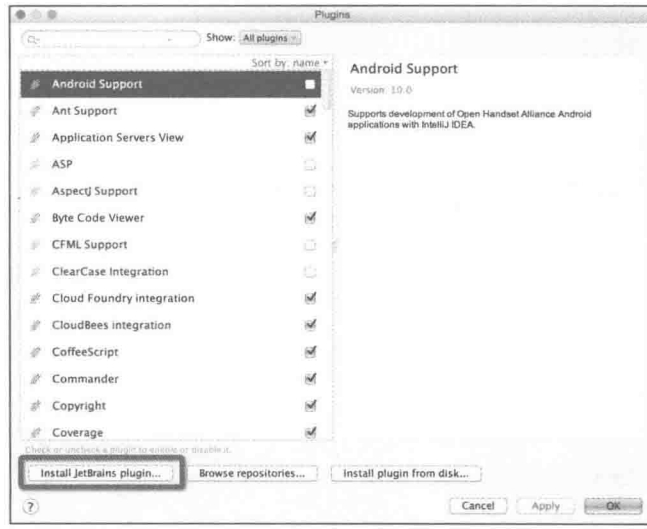


图 2-3　单击“Install JetBrains plugin”

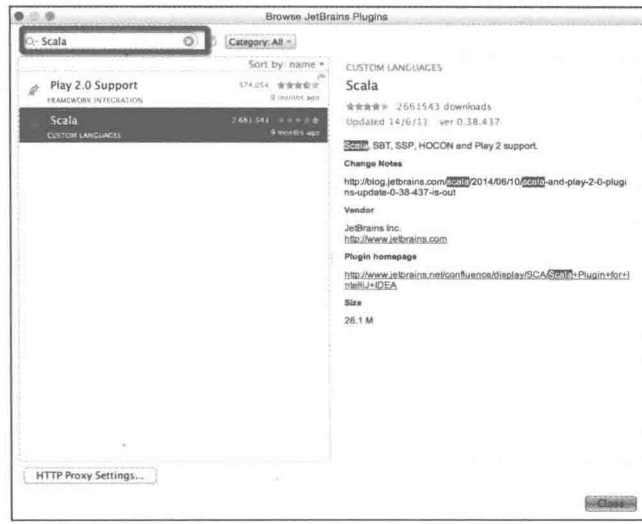


图 2-4　搜索并安装 Scala

然后要为 Spark 生成 IDEA 所需要的工程文件，在源代码的根目录执行下面的命令：

```
./sbt/sbt?gen-idea
```

命令成功执行后，SBT 的工程文件就生成了。接下来直接导入工程即可，如图 2-5 所示。

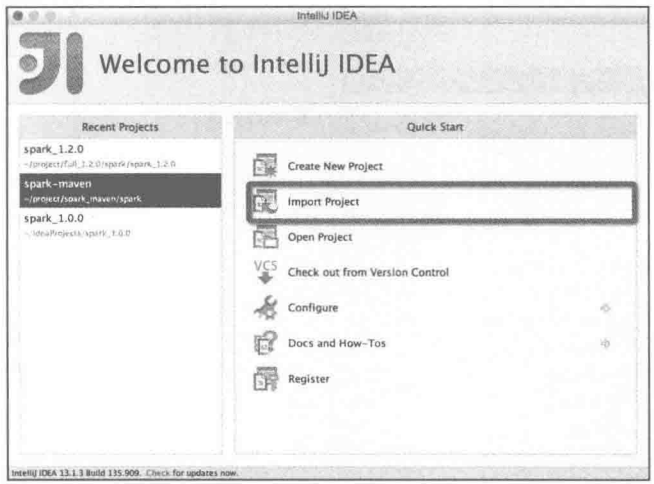


图 2-5 导入工程

在选择了工程所在的根目录后，选择导入“SBT project”，如图 2-6 所示。

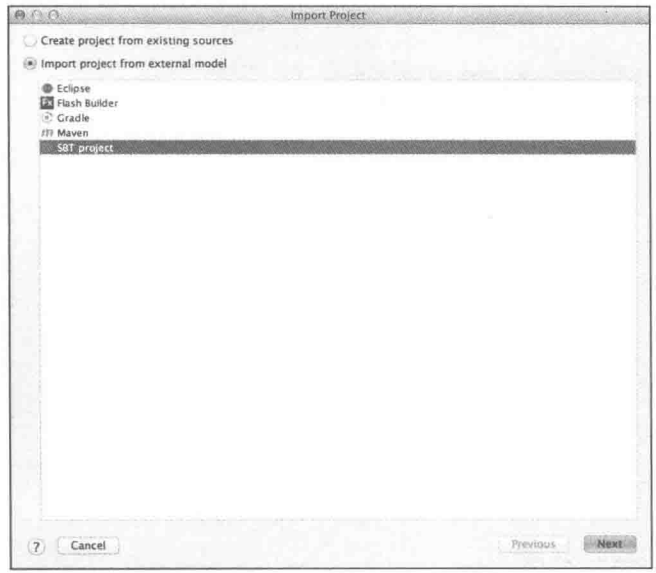


图 2-6 选择导入“SBT project”

选择了图 2-7 所示的复选框后，单击“Finish”即可完成工程的建立。

完成导入后就可以开始阅读源码了。

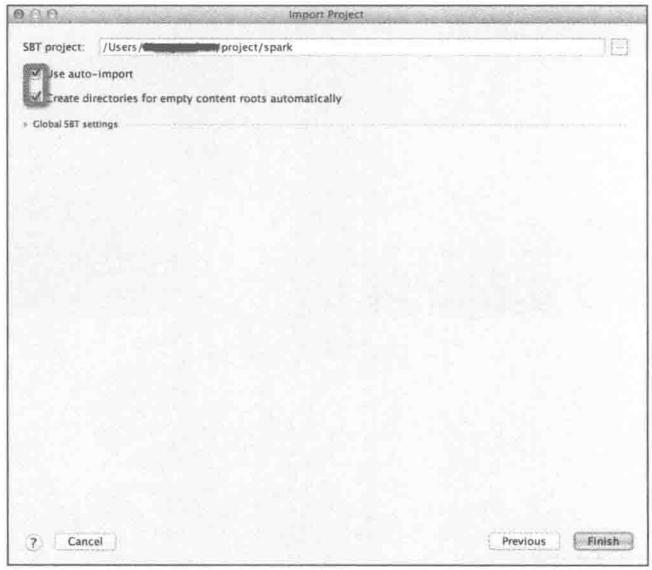
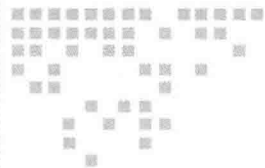


图 2-7　完成导入

2.3　小结

本章主要介绍了 Spark 学习环境的搭建，包括源码的获取和编译，Spark 源码阅读环境的建立等。接下来，将逐步介绍 Spark Core 的各个模块的实现原理和设计思想。



Chapter 3 第 3 章

RDD 实现详解

RDD 是 Spark 最基本也是最根本的数据抽象，它具备像 MapReduce 等数据流模型的容错性，并且允许开发人员在大型集群上执行基于内存的计算。现有的数据流系统对两种应用的处理并不高效：一是迭代式算法，这在图应用和机器学习领域很常见；二是交互式数据挖掘工具。这两种情况下，将数据保存在内存中能够极大地提高性能。为了有效地实现容错，RDD 提供了一种高度受限的共享内存，即 RDD 是只读的，并且只能通过其他 RDD 上的批量操作来创建。尽管如此，RDD 仍然足以表示很多类型的计算，包括 MapReduce 和专用的迭代编程模型（如 Pregel）等。Spark 实现的 RDD 在迭代计算方面比 Hadoop 快 20 多倍，同时还可以在 5 ～ 7 秒内交互式地查询 1TB 数据集。

3.1 概述

Spark 的目标是为基于工作集的应用（即多个并行操作重用中间结果的应用）提供抽象，同时保持 MapReduce 及其相关模型的优势特性，即自动容错、位置感知性调度和可伸缩性。RDD 比数据流模型更易于编程，同时基于工作集的计算也具有良好的描述能力。

在这些特性中，最难实现的是容错性。一般来说，分布式数据集的容错性有两种

方式：数据检查点和记录数据的更新。我们面向的是大规模数据分析，数据检查点操作成本很高：需要通过数据中心的网络连接在机器之间复制庞大的数据集，而网络带宽往往比内存带宽低得多，同时还需要消耗更多的存储资源（在内存中复制数据可以减少需要缓存的数据量，而存储到磁盘则会降低应用程序速度）。所以，我们选择记录更新的方式。但是，如果更新太多，记录更新成本也不低。因此，RDD 只支持粗粒度转换，即在大量记录上执行的单个操作。将创建 RDD 的一系列转换记录下来（即 Lineage），以便恢复丢失的分区。

虽然只支持粗粒度转换限制了编程模型，但 RDD 仍然可以很好地适用于很多应用，特别是支持数据并行的批量分析应用，包括数据挖掘、机器学习、图算法等，因为这些程序通常都会在很多记录上执行相同的操作。RDD 不太适合那些异步更新共享状态的应用，例如并行 Web 网络爬虫。因此，Spark 的目标是为大多数分析型应用提供有效的编程模型，而其他类型的应用则交给专门的系统。

3.2　什么是 RDD

什么是 RDD？ RDD 是只读的、分区记录的集合。RDD 只能基于在稳定物理存储中的数据集和其他已有的 RDD 上执行确定性操作来创建。这些确定性操作称为转换，如 map、filter、groupBy、join。RDD 不需要物化。RDD 含有如何从其他 RDD 衍生（即计算）出本 RDD 的相关信息（即 Lineage），因此在 RDD 部分分区数据丢失的时候可以从物理存储的数据计算出相应的 RDD 分区。

RDD 支持基于工作集的应用，同时具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

每个 RDD 有 5 个主要的属性：

1）一组分片（Partition），即数据集的基本组成单位。对于 RDD 来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建 RDD 时指定 RDD 的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分

配到的 CPU Core 的数目。图 3-1 描述了分区存储的计算模型，每个分配的存储是由 BlockManager 实现的。每个分区都会被逻辑映射成 BlockManager 的一个 Block，而这个 Block 会被一个 Task 负责计算。

2）一个计算每个分区的函数。Spark 中 RDD 的计算是以分片为单位的，每个 RDD 都会实现 compute 函数以达到这个目的。compute 函数会对迭代器进行复合，不需要保存每次计算的结果。详情请参阅 3.4.5 节。

3）RDD 之间的依赖关系。RDD 的每次转换都会生成一个新的 RDD，所以 RDD 之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark 可以通过这个依赖关系重新计算丢失的分区数据，而不是对 RDD 的所有分区进行重新计算。

4）一个 Partitioner，即 RDD 的分片函数。当前 Spark 中实现了两种类型的分片函数，一个是基于哈希的 HashPartitioner，另外一个是基于范围的 RangePartitioner。只有对于 key-value 的 RDD，才会有 Partitioner，非 key-value 的 RDD 的 Parititioner 的值是 None。Partitioner 函数不但决定了 RDD 本身的分片数量，也决定了 parent RDD Shuffle 输出时的分片数量。

5）一个列表，存储存取每个 Partition 的优先位置（preferred location）。对于一个 HDFS 文件来说，这个列表保存的就是每个 Partition 所在的块的位置。按照“移动数据不如移动计算”的理念，Spark 在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

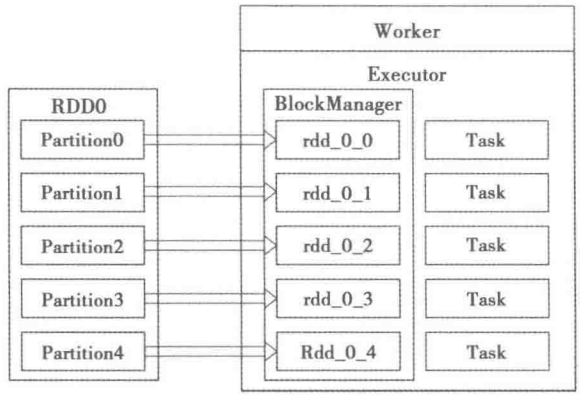


图 3-1　RDD Partition 的存储和计算模型

3.2.1　RDD 的创建

可以通过两种方式创建 RDD：

1）由一个已经存在的 Scala 集合创建。

2）由外部存储系统的数据集创建，包括本地的文件系统，还有所有 Hadoop 支持的数据集，比如 HDFS、Cassandra、HBase、Amazon S3 等。

RDD 创建后，就可以在 RDD 上进行数据处理。RDD 支持两种操作：转换（transformation），即从现有的数据集创建一个新的数据集；动作（action），即在数据集上进行计算后，返回一个值给 Driver 程序。例如，map 就是一种转换，它将数据集每一个元素都传递给函数，并返回一个新的分布式数据集表示结果。另一方面，reduce 是一种动作，通过一些函数将所有元素叠加起来，并将最终结果返回给 Driver（还有一个并行的 reduceByKey，能返回一个分布式数据集）。

图 3-2 描述了从外部数据源创建 RDD，经过多次转换，通过一个动作操作将结果写回外部存储系统的逻辑运行图。整个过程的计算都是在 Worker 中的 Executor 中运行。

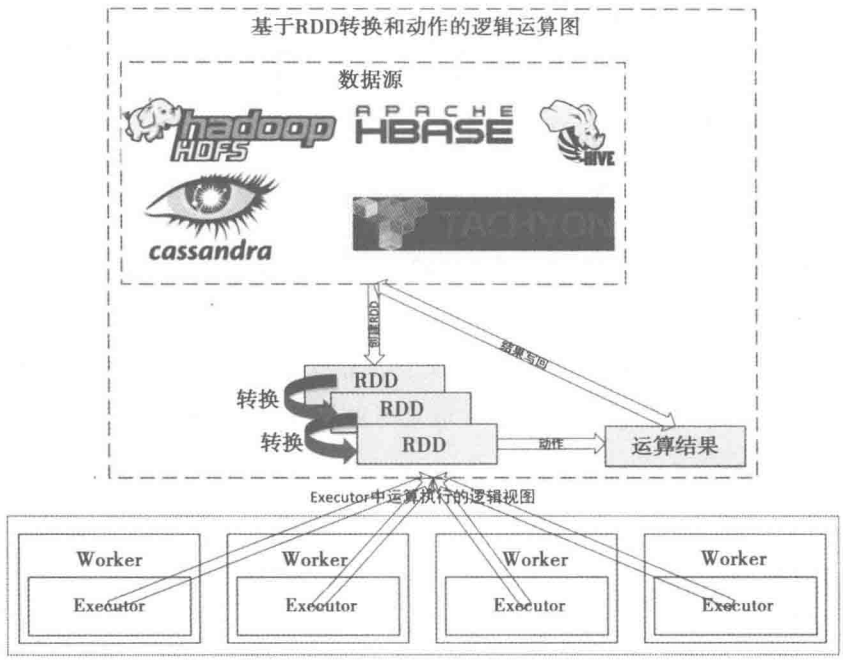


图 3-2　RDD 创建、转换和动作的逻辑计算图

3.2.2 RDD 的转换

RDD 中的所有转换都是惰性的，也就是说，它们并不会直接计算结果。相反的，它们只是记住这些应用到基础数据集（例如一个文件）上的转换动作。只有当发生一个要求返回结果给 Driver 的动作时，这些转换才会真正运行。这个设计让 Spark 更加有效率地运行。例如我们可以实现：通过 map 创建的一个新数据集，并在 reduce 中使用，最终只返回 reduce 的结果给 Driver，而不是整个大的新数据集。图 3-3 描述了 RDD 在进行 groupByRey 时的内部 RDD 转换的实现逻辑图。图 3-4 描述了 reduceByKey 的实现逻辑图。

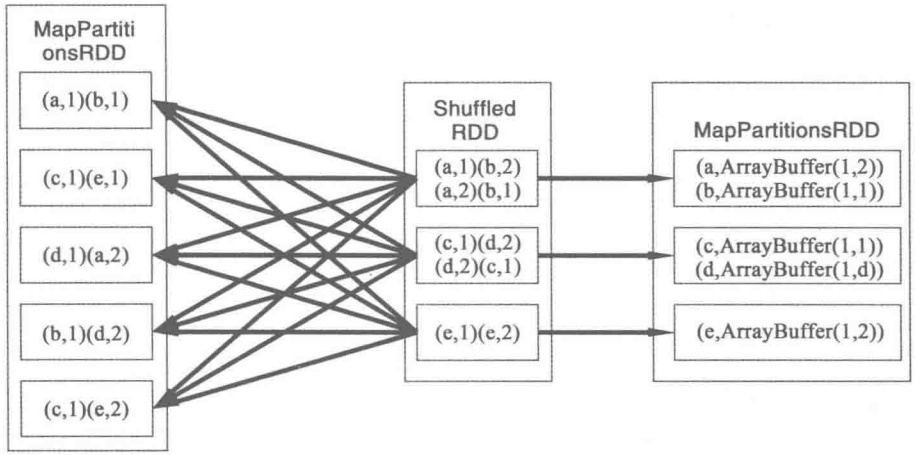


图 3-3 RDD groupByKey 的逻辑转换图

在 groupByKey 的操作中，会在 MapPartitionsRDD 做一次 Shuffle，图 3-3 中设置的分片数量是 3，因此 ShuffledRDD 会有 3 个分片，ShuffledRDD 实际上仅仅是从上游的任务中读取 Shuffle 的结果，因此图的箭头是指向上游的 MapPartitionsRDD 的。关于 Shuffle 的实现实际上要比图中展示得复杂得多，具体的实现细节可以参阅第 7 章。

reduceByKey 和 groupByKey 的实现差不多，它在 Shuffle 完成之后，需要做一次 reduce。

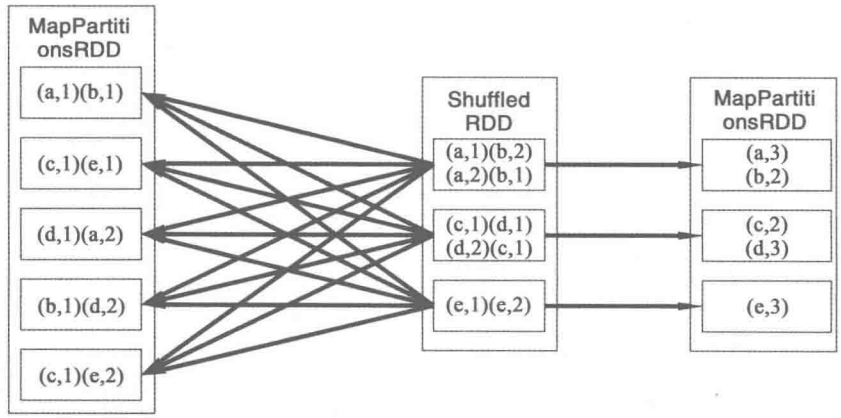


图 3-4　RDD reduceByKey 的逻辑转换图

默认情况下，每一个转换过的 RDD 都会在它执行一个动作时被重新计算。不过也可以使用 persist（或者 cache）方法，在内存中持久化一个 RDD。在这种情况下，Spark 将会在集群中保存相关元素，下次查询这个 RDD 时能更快访问它。也支持在磁盘上持久化数据集，或在集群间复制数据集，这些选项将在下一节进行描述。

RDD 支持的转换如表 3-1 所示。

表 3-1　RDD 支持的转换

转　换	含　义
map(func)	返回一个新的分布式数据集，该数据集由每一个输入元素经过 func 函数转换后组成
filter(func)	返回一个新的数据集，该数据集由经过 func 函数计算后返回值为 true 的输入元素组成
flatMap(func)	类似于 map，但是每一个输入元素可以被映射为 0 或多个输出元素（因此 func 应该返回一个序列，而不是单一元素）
mapPartitions(func)	类似于 map，但独立地在 RDD 的每一个分片上运行，因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 Iterator[T] => Iterator[U]
mapPartitionsWithSplit(func)	类似于 mapPartitions，但 func 带有一个整数参数表示分片的索引值。因此在类型为 T 的 RDD 上运行时，func 的函数类型必须是 (Int, Iterator[T]) => Iterator[U]
sample(withReplacement,fraction,seed)	根据 fraction 指定的比例对数据进行采样，可以选择是否用随机数进行替换，seed 用于指定随机数生成器种子

（续）

转　换	含　义
union(otherDataset)	返回一个新的数据集，新数据集是由源数据集和参数数据集联合而成
distinct([numTasks]))	返回一个包含源数据集中所有不重复元素的新数据集
groupByKey([numTasks])	在一个（K,V）对的数据集上调用，返回一个（K, Seq[V]) 对的数据集 **注意**　默认情况下，只有 8 个并行任务来做操作，但是可以传入一个可选的 numTasks 参数来改变它
reduceByKey(func, [numTasks])	在一个（K, V) 对的数据集上调用，返回一个（K, V）对的数据集，使用指定的 reduce 函数，将相同 key 的值聚合到一起。与 groupByKey 类似，reduce 任务的个数是可以通过第二个可选参数来设置的
sortByKey([ascending], [numTasks])	在一个（K, V) 对的数据集上调用，K 必须实现 Ordered 接口，返回一个按照 key 进行排序的（K, V）对数据集。升序或降序由 ascending 布尔参数决定
join(otherDataset, [numTasks])	在类型为（K,V) 和（K,W) 类型的数据集上调用，返回一个相同 key 对应的所有元素对在一起的 (K, (V, W)) 数据集
cogroup(otherDataset, [numTasks])	在类型为（K,V) 和（K,W) 的数据集上调用，返回一个 (K, Seq[V], Seq[W]) 元组的数据集。这个操作也可以称为 groupwith
cartesian(otherDataset)	笛卡儿积，在类型为 T 和 U 类型的数据集上调用，返回一个 (T, U) 对数据集（两两的元素对）

3.2.3　RDD 的动作

RDD 支持的动作如表 3-2 所示。

表 3-2　RDD 支持的动作

动　作	含　义
reduce(func)	通过函数 func（接受两个参数，返回一个参数）聚集数据集中的所有元素。这个功能必须可交换且可关联的，从而可以正确地并行执行
collect()	在驱动程序中，以数组的形式返回数据集的所有元素。通常在使用 filter 或者其他操作返回一个足够小的数据子集后再使用会比较有用
count()	返回数据集的元素的个数
first()	返回数据集的第一个元素（类似于 take（1））
take(n)	返回一个由数据集的前 n 个元素组成的数组 **注意**　这个操作目前并非并行执行，而是由驱动程序计算所有的元素

（续）

动　作	含　义
takeSample(withReplacement, num,seed)	返回一个数组，该数组由从数据集中随机采样的 num 个元素组成，可以选择是否用随机数替换不足的部分，seed 用于指定随机数生成器种子
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。对于每个元素，Spark 将会调用 toString 方法，将它转换为文件中的文本行
saveAsSequenceFile(path)	将数据集的元素以 Hadoop sequencefile 的格式保存到指定的目录下，可以是本地系统、HDFS 或者任何其他 Hadoop 支持的文件系统。这个只限于由 key-value 对组成，并实现了 Hadoop 的 Writable 接口，或者可以隐式地转换为 Writable 的 RDD（Spark 包括了基本类型的转换，例如 Int、Double、String 等）
countByKey()	对 (K,V) 类型的 RDD 有效，返回一个 (K, Int) 对的 map，表示每一个 key 对应的元素个数
foreach(func)	在数据集的每一个元素上，运行函数 func 进行更新。通常用于边缘效果，例如更新一个累加器，或者和外部存储系统进行交互，例如 HBase

3.2.4　RDD 的缓存

Spark 速度非常快的原因之一，就是在不同操作中在内存中持久化（或缓存）一个数据集。当持久化一个 RDD 后，每一个节点都将把计算的分片结果保存在内存中，并在对此数据集（或者衍生出的数据集）进行的其他动作（action）中重用。这使得后续的动作变得更加迅速（通常快 10 倍）。RDD 相关的持久化和缓存，是 Spark 最重要的特征之一。可以说，缓存是 Spark 构建迭代式算法和快速交互式查询的关键。

通过 persist() 或 cache() 方法可以标记一个要被持久化的 RDD，一旦首次被触发，该 RDD 将会被保留在计算节点的内存中并重用。实际上，cache() 是使用 persist() 的快捷方法，它们的实现如下：

```
/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)
/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def cache(): this.type = persist()
```

图 3-5 中，假设首先进行了 RDD0 → RDD1 → RDD2 的计算作业，那么计算结束时，RDD1 就已经缓存在系统中了。在进行 RDD0 → RDD1 → RDD3 的计算作

业时，由于 RDD1 已经缓存在系统中，因此 RDD0 → RDD1 的转换不会重复进行，计算作业只须进行 RDD1 → RDD3 的计算就可以了，因此计算速度可以得到很大提升。

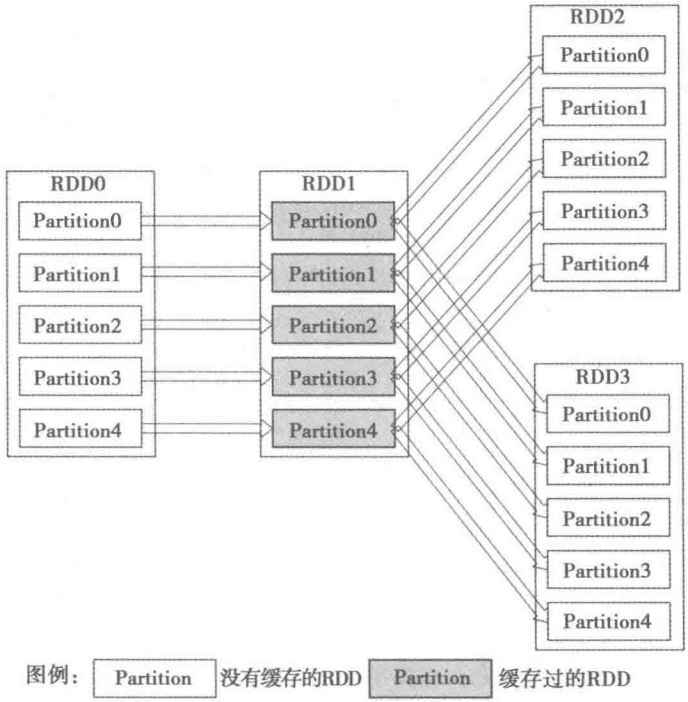


图 3-5 RDD 缓存过的 Partition 可以加快下一次的计算速度

缓存有可能丢失，或者存储于内存的数据由于内存不足而被删除。RDD 的缓存的容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于 RDD 的一系列的转换，丢失的数据会被重算。RDD 的各个 Partition 是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部 Partition。

3.2.5 RDD 的检查点

RDD 的缓存能够在第一次计算完成后，将计算结果保存到内存、本地文件系统或者 Tachyon 中。通过缓存，Spark 避免了 RDD 上的重复计算，能够极大地提升计算速度。但是，如果缓存丢失了，则需要重新计算。如果计算特别复杂或者计算耗时特别

多，那么缓存丢失对于整个 Job 的影响是不容忽视的。为了避免缓存丢失重新计算带来的开销，Spark 又引入了检查点（checkpoint）机制。

缓存是在计算结束后，直接将计算结果通过用户定义的存储级别（存储级别定义了缓存存储的介质，现在支持内存、本地文件系统和 Tachyon）写入不同的介质。而检查点不同，它是在计算完成后，重新建立一个 Job 来计算。为了避免重复计算，推荐先将 RDD 缓存，这样就能保证检查点的操作可以快速完成。

用户可以通过调用 org.apache.spark.rdd.RDD#checkpoint() 来指定 RDD 需要检查点机制。

3.3　RDD 的转换和 DAG 的生成

Spark 会根据用户提交的计算逻辑中的 RDD 的转换和动作来生成 RDD 之间的依赖关系，同时这个计算链也就生成了逻辑上的 DAG。接下来以“ Word Count”为例，详细描述这个 DAG 生成的实现过程。

Spark Scala 版本的 Word Count 程序如下：

```
1:  val file = spark.textFile("hdfs://...")
2:  val counts = file.flatMap(line => line.split(" "))
3:          .map(word => (word, 1))
4:          .reduceByKey(_ + _)
5:  counts.saveAsTextFile("hdfs://...")
```

file 和 counts 都是 RDD，其中 file 是从 HDFS 上读取文件并创建了 RDD，而 counts 是在 file 的基础上通过 flatMap、map 和 reduceByKey 这三个 RDD 转换生成的。最后，counts 调用了动作 saveAsTextFile，用户的计算逻辑就从这里开始提交的集群进行计算。那么上面这 5 行代码的具体实现是什么呢？

1）行 1：spark 是 org.apache.spark.SparkContext 的实例，它是用户程序和 Spark 的交互接口。spark 会负责连接到集群管理者，并根据用户设置或者系统默认设置来申请计算资源，完成 RDD 的创建等。

spark.textFile("hdfs://...") 就完成了一个 org.apache.spark.rdd.HadoopRDD 的创建，并

且完成了一次 RDD 的转换：通过 map 转换到一个 org.apache.spark.rdd.MapPartitions-RDD。也就是说，file 实际上是一个 MapPartitionsRDD，它保存了文件的所有行的数据内容。

2）行 2：将 file 中的所有行的内容，以空格分隔为单词的列表，然后将这个按照行构成的单词列表合并为一个列表。最后，以每个单词为元素的列表被保存到 MapPartitionsRDD。

3）行 3：将第 2 步生成的 MapPartitionsRDD 再次经过 map 将每个单词 word 转为 (word, 1) 的元组。这些元组最终被放到一个 MapPartitionsRDD 中。

4）行 4：首先会生成一个 MapPartitionsRDD，起到 map 端 combiner 的作用；然后会生成一个 ShuffledRDD，它从上一个 RDD 的输出读取数据，作为 reducer 的开始；最后，还会生成一个 MapPartitionsRDD，起到 reducer 端 reduce 的作用。

5）行 5：首先会生成一个 MapPartitionsRDD，这个 RDD 会通过调用 org.apache.spark.rdd.PairRDDFunctions#saveAsHadoopDataset 向 HDFS 输出 RDD 的数据内容。最后，调用 org.apache.spark.SparkContext#runJob 向集群提交这个计算任务。

RDD 之间的关系可以从两个维度来理解：一个是 RDD 是从哪些 RDD 转换而来，也就是 RDD 的 parent RDD(s) 是什么；还有就是依赖于 parent RDD(s) 的哪些 Partition(s)。这个关系，就是 RDD 之间的依赖，org.apache.spark.Dependency。根据依赖于 parent RDD(s) 的 Partitions 的不同情况，Spark 将这种依赖分为两种，一种是宽依赖，一种是窄依赖。

3.3.1 RDD 的依赖关系

RDD 和它依赖的 parent RDD(s) 的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）。

1）窄依赖指的是每一个 parent RDD 的 Partition 最多被子 RDD 的一个 Partition 使用，如图 3-6 所示。

2）宽依赖指的是多个子 RDD 的 Partition 会依赖同一个 parent RDD 的 Partition，如图 3-7 所示。

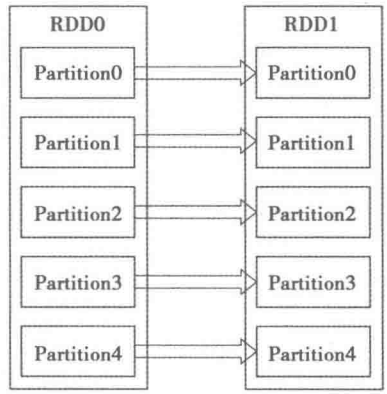


图 3-6　RDD 的窄依赖

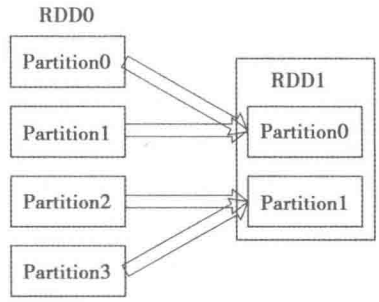


图 3-7　RDD 的宽依赖

接下来可以从不同类型的转换来进一步理解 RDD 的窄依赖和宽依赖的区别，如图 3-8 所示。

对于 map 和 filter 形式的转换来说，它们只是将 Partition 的数据根据转换的规则进行转化，并不涉及其他的处理，可以简单地认为只是将数据从一个形式转换到另一个形式。对于 union，只是将多个 RDD 合并成一个，parent RDD 的 Partition(s) 不会有任何的变化，可以认为只是把 parent RDD 的 Partition(s) 简单进行复制与合并。对于 join，如果每个 Partition 仅仅和已知的、特定的 Partition 进行 join，那么这个依赖关系也是窄依赖。对于这种有规则的数据的 join，并不会引入昂贵的 Shuffle。对于窄依赖，由于 RDD 每个 Partition 依赖固定数量的 parent RDD(s) 的 Partition(s)，因此可以通过一个计算任务来处理这些 Partition，并且这些 Partition 相互独立，这些计算任务也就可以并行执行了。

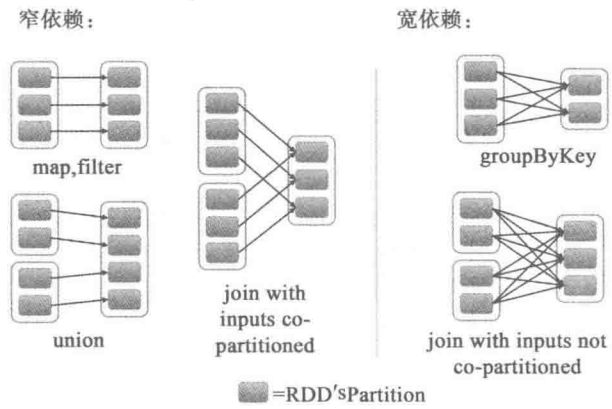


图 3-8 RDD 的窄依赖和宽依赖

对于 groupByKey，子 RDD 的所有 Partition(s) 会依赖于 parent RDD 的所有 Partition(s)，子 RDD 的 Partition 是 parent RDD 的所有 Partition Shuffle 的结果，因此这两个 RDD 是不能通过一个计算任务来完成的。同样，对于需要 parent RDD 的所有 Partition 进行 join 的转换，也是需要 Shuffle，这类 join 的依赖就是宽依赖而不是前面提到的窄依赖了。

所有的依赖都要实现 trait Dependency[T]：

```
abstract class Dependency[T] extends Serializable {
    def rdd: RDD[T]
}
```

其中 rdd 就是依赖的 parent RDD。

对于窄依赖的实现是：

```
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency[T] {
    //返回子RDD的partitionId依赖的所有的parent RDD的Partition(s)
    def getParents(partitionId: Int): Seq[Int]
    override def rdd: RDD[T] = _rdd
}
```

现在有两种窄依赖的具体实现，一种是一对一的依赖，即 OneToOneDependency：

```
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd) {
    override def getParents(partitionId: Int) = List(partitionId)
}
```

通过 getParents 的实现不难看出，RDD 仅仅依赖于 parent RDD 相同 ID 的 Partition。

还有一个是范围的依赖，即 RangeDependency，它仅仅被 org.apache.spark.rdd.UnionRDD 使用。UnionRDD 是把多个 RDD 合成一个 RDD，这些 RDD 是被拼接而成，即每个 parent RDD 的 Partition 的相对顺序不会变，只不过每个 parent RDD 在 UnionRDD 中的 Partition 的起始位置不同。因此它的 getPartents 如下：

```
override def getParents(partitionId: Int) = {
    if(partitionId >= outStart && partitionId < outStart + length) {
       List(partitionId - outStart + inStart)
    } else {
       Nil
    }
}
```

其中，inStart 是 parent RDD 中 Partition 的起始位置，outStart 是在 UnionRDD 中的起始位置，length 就是 parent RDD 中 Partition 的数量。

宽依赖的实现只有一种：ShuffleDependency。子 RDD 依赖于 parent RDD 的所有 Partition，因此需要 Shuffle 过程：

```
class ShuffleDependency[K, V, C](
    @transient _rdd: RDD[_ <: Product2[K, V]],
    val partitioner: Partitioner,
    val serializer: Option[Serializer] = None,
    val keyOrdering: Option[Ordering[K]] = None,
    val aggregator: Option[Aggregator[K, V, C]] = None,
    val mapSideCombine: Boolean = false)
extends Dependency[Product2[K, V]] {

override def rdd = _rdd.asInstanceOf[RDD[Product2[K, V]]]
//获取新的shuffleId
val shuffleId: Int = _rdd.context.newShuffleId()
//向ShuffleManager注册Shuffle的信息
val shuffleHandle: ShuffleHandle =
_rdd.context.env.shuffleManager.registerShuffle(
    shuffleId, _rdd.partitions.size, this)

    _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
}
```

宽依赖支持两种 Shuffle Manager，即 org.apache.spark.shuffle.hash.HashShuffleManager

（基于 Hash 的 Shuffle 机制）和 org.apache.spark.shuffle.sort.SortShuffleManager（基于排序的 Shuffle 机制）。

3.3.2 DAG 的生成

原始的 RDD(s) 通过一系列转换就形成了 DAG。RDD 之间的依赖关系，包含了 RDD 由哪些 Parent RDD(s) 转换而来和它依赖 parent RDD(s) 的哪些 Partitions，是 DAG 的重要属性。借助这些依赖关系，DAG 可以认为这些 RDD 之间形成了 Lineage（血统）。借助 Lineage，能保证一个 RDD 被计算前，它所依赖的 parent RDD 都已经完成了计算；同时也实现了 RDD 的容错性，即如果一个 RDD 的部分或者全部的计算结果丢失了，那么就需要重新计算这部分丢失的数据。

那么 Spark 是如何根据 DAG 来生成计算任务呢？首先，根据依赖关系的不同将 DAG 划分为不同的阶段（Stage）。对于窄依赖，由于 Partition 依赖关系的确定性，Partition 的转换处理就可以在同一个线程里完成，窄依赖被 Spark 划分到同一个执行阶段；对于宽依赖，由于 Shuffle 的存在，只能在 parent RDD(s) Shuffle 处理完成后，才能开始接下来的计算，因此宽依赖就是 Spark 划分 Stage 的依据，即 Spark 根据宽依赖将 DAG 划分为不同的 Stage。在一个 Stage 内部，每个 Partition 都会被分配一个计算任务（Task），这些 Task 是可以并行执行的。Stage 之间根据依赖关系变成了一个大粒度的 DAG，这个 DAG 的执行顺序也是从前向后的。也就是说，Stage 只有在它没有 parent Stage 或者 parent Stage 都已经执行完成后，才可以执行。这个过程可以查询第 4 章。

3.3.3 Word Count 的 RDD 转换和 DAG 划分的逻辑视图

上文分析了在 Word Count 的 RDD 转换时，Spark 生成了不同的 RDD。这些 RDD 有的和用户逻辑直接显式对应，比如 map 操作会生成一个 org.apache.spark.rdd.MapPartitionsRDD；而有的 RDD 则是和 Spark 的实现原理相关，是 Spark 隐式生成的，比如 org.apache.spark.rdd.ShuffledRDD，这个过程对于用户来说是透明的，用户只需要关心 RDD 的转换和动作即可。

RDD 在创建子 RDD 的时候，会通过 Dependency 来定义它们之间的关系。通过

Dependency，子 RDD 也可以获得它的 parent RDD 和 parent RDD 的 Partition。

RDD 转换的细节如图 3-9 所示。

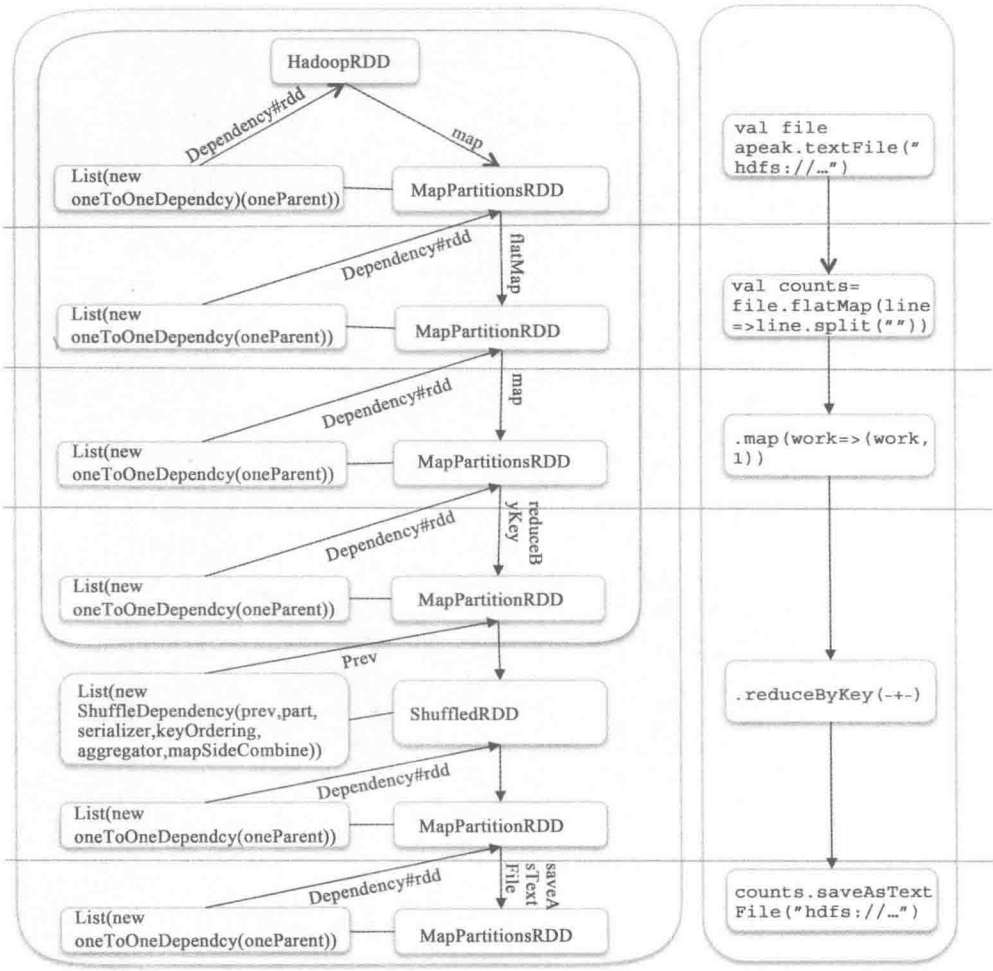


图 3-9 “Word Count”的 RDD 转换

通过图 3-9，可以清晰地看到 Spark 对于用户提交的 Application 所做的处理。用户定义的 RDD 被系统显式和隐式地转换成多个 RDD 以及这些 RDD 之间的依赖，这些依赖构建了这些 RDD 的处理顺序及相互关系。关于这些 RDD 的转换时如何在计算节点上运行的，请参阅第 4 章。

为了对图 3-9 有更加直观的理解，图 3-10 以一个有五个分片的输入文件为例，详细描述了“ Word Count”的逻辑执行过程。之所以称为逻辑执行过程，是因为具体的计算过程可能会有网络的交互，有频繁地将处理中间数据写入磁盘等过程。

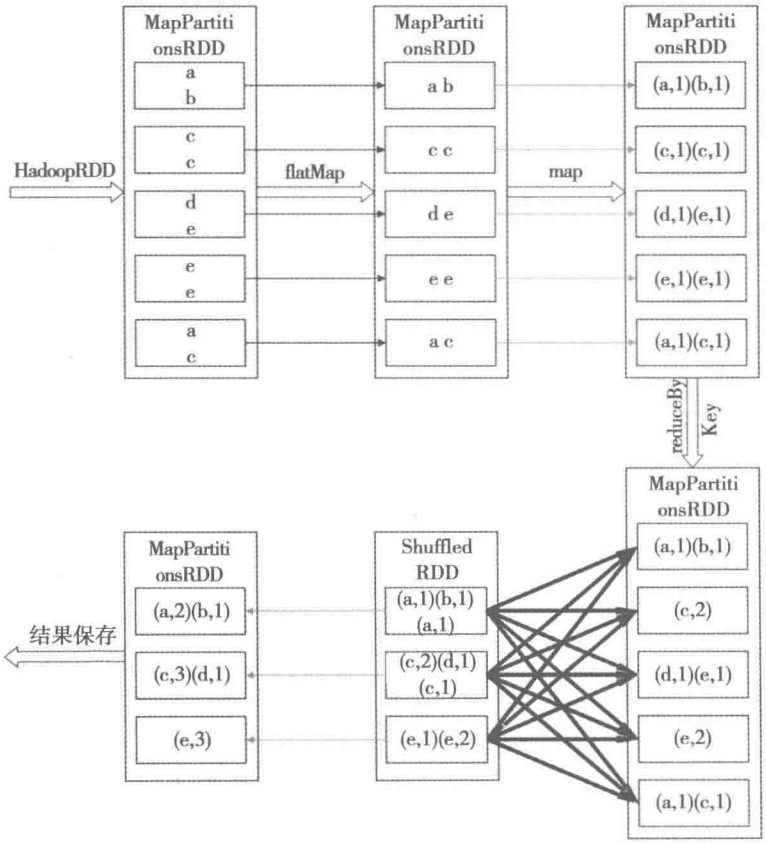


图 3-10 “Word Count” RDD 的逻辑转换关系图

需要强调的一点是在转换操作 reduceByKey 时会触发一个 Shuffle（洗牌）的过程。在 Shuffle 开始之前，有一个本地聚合的过程，比如第三个分片的（e,1）（e,1）聚合成了（e,2）。Shuffle 的结果是为下游的 Task 生成了三个分片，这三个分片就构成了 ShuffledRDD。之后在做了一个聚合之后，就生成了结果的 RDD。关于 Shuffle 的具体实现过程，可以参阅第 7 章。

3.4　RDD 的计算

3.4.1　Task 简介

原始的 RDD 经过一系列转换后，会在最后一个 RDD 上触发一个动作，这个动作会生成一个 Job。在 Job 被划分为一批计算任务（Task）后，这批 Task 会被提交到集群上的计算节点去计算。计算节点执行计算逻辑的部分称为 Executor。Executor 在准备好 Task 的运行时环境后，会通过调用 org.apache.spark.scheduler.Task#run 来执行计算。Spark 的 Task 分为两种：

1）org.apache.spark.scheduler.ShuffleMapTask

2）org.apache.spark.scheduler.ResultTask

简单来说，DAG 的最后一个阶段会为每个结果的 Partition 生成一个 ResultTask，其余所有的阶段都会生成 ShuffleMapTask。生成的 Task 会被发送到已经启动的 Executor 上，由 Executor 来完成计算任务的执行，执行过程的实现在 org.apache.spark.executor.Executor.TaskRunner#run。第 6 章会介绍这一部分的实现原理和设计思想。

3.4.2　Task 的执行起点

org.apache.spark.scheduler.Task#run 会调用 ShuffleMapTask 或者 ResultTask 的 runTask；runTask 会调用 RDD 的 org.apache.spark.rdd.RDD#iterator。计算由此开始。

```
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
   if(storageLevel != StorageLevel.NONE) {
       //如果存储级别不是NONE，那么先检查是否有缓存；没有缓存则要进行计算
       SparkEnv.get.cacheManager.getOrCompute(this, split, context,storageLevel)
   } else {
       //如果有checkpoint，那么直接读取结果；否则直接进行计算
       computeOrReadCheckpoint(split, context)
   }
}
```

其中，SparkEnv 中包含了一个运行时节点所需要的所有的环境信息。cache-

Manager 是 org.apache.spark.CacheManager，它负责调用 BlockManager 来管理 RDD 的缓存，如果当前 RDD 原来计算过并且把结果缓存起来，那么接下来的运算都可以通过 BlockManager 来直接读取缓存后返回。SparkEnv 除了 cacheManager，还包括以下重要的成员变量：

1）akka.actor.ActorSystem：运行在该节点的 Actor System，其中运行在 Driver 上的名字是 sparkDriver；运行在 Executor 上的是 sparkExecutor。

2）org.apache.spark.serializer.Serializer：序列化和发序列化的工具。

3）org.apache.spark.MapOutputTracker：保存 Shuffle Map Task 输出的位置信息。其中在 Driver 上的 Tracer 是 org.apache.spark.MapOutputTrackerMaster；而在 Executor 上的 Tracker 是 org.apache.spark.MapOutputTrackerWorker，它会从 org.apache.spark.MapOutputTrackerMaster 获取信息。

4）org.apache.spark.shuffle.ShuffleManager：Shuffle 的管理者，其中 Driver 端会注册 Shuffle 的信息，而 Executor 端会上报和获取 Shuffle 的信息。现阶段内置支持 Hash Based Shuffle 和 Sort Based Shuffle，具体实现细节请参阅第 7 章。

5）org.apache.spark.broadcast.BroadcastManager：广播变量的管理者。

6）org.apache.spark.network.BlockTransferService：Executor 读取 Shuffle 数据的 Client。当前支持 netty 和 nio，可以通过 spark.shuffle.blockTransferService 来设置。具体详情可以参阅第 7 章。

7）org.apache.spark.storage.BlockManager：提供了 Storage 模块与其他模块的交互接口，管理 Storage 模块。

8）org.apache.spark.SecurityManager：Spark 对于认证授权的实现。

9）org.apache.spark.HttpFileServer：可以提供 HTTP 服务的 Server。当前主要用于 Executor 端下载依赖。

10）org.apache.spark.metrics.MetricsSystem：用于搜集统计信息。

11）org.apache.spark.shuffle.ShuffleMemoryManager：管理 Shuffle 过程中使用的内存。ExternalAppendOnlyMap 和 ExternalSorter 都会从 ShuffleMemoryManager 中申请内存，在数据 spill 到 Disk 后会释放内存。当然了，当 Task 退出时这个内存也会被回收。为了使得每个 thread 都会比较公平地获取内存资源，避免一个 thread 申请了大

量内存后造成其他的 thread 需要频繁地进行 spill 操作，它采取的内存分配策略是：对于 N 个 thread，每个 thread 可以至少申请 1/（2*N）的内存，但是至多申请 1/N。这个 N 是动态变化的，感兴趣的读者可以查阅这个类的具体实现。

在用户创建 org.apache.spark.SparkContext 时会创建 org.apache.spark.SparkEnv。

3.4.3　缓存的处理

如果存储级别不是 NONE，那么先检查是否有缓存；没有缓存则要进行计算。什么是存储级别？从用户的角度来看就是缓存保存到不同的存储位置，比如内存、硬盘、Tachyon；还有缓存的数据是否需要序列化等。详细的存储级别的介绍可以参阅第 8 章。

cacheManager 对 Storage 模块进行了封装，使得 RDD 可以更加简单地从 Storage 模块读取或者写入数据。RDD 的每个 Partition 对应 Storage 模块的一个 Block，只不过 Block 是 Partition 经过处理后的数据。在系统实现的层面上，可以认为 Partition 和 Block 是一一对应的。cacheManager 会通过 getOrCompute 来判断当前的 RDD 是否需要进行计算。

首先，cacheManager 会通过 RDD 的 ID 和当前计算的 Partition 的 ID 向 Storage 模块的 BlockManager 发起查询请求，如果能够获得 Block 的信息，会直接返回 Block 的信息。否则，代表该 RDD 是需要计算的。这个 RDD 以前可能计算过并且被存储到了内存中，但是后来由于内存紧张，这部分内存被清理了。在计算结束后，计算结果会根据用户定义的存储级别，写入 BlockManager 中。这样，下次就可以不经过计算而直接读取该 RDD 的计算结果了。核心实现逻辑如下：

```
def getOrCompute[T](
        rdd: RDD[T],
        partition: Partition,
        context: TaskContext,
        storageLevel: StorageLevel): Iterator[T] = {
    //获取RDD的BlockId
    val key = RDDBlockId(rdd.id, partition.index)
    logDebug(s"Looking for partition $key")
    blockManager.get(key) match { //向BlockManager查询是否有缓存
```

```
        case Some(blockResult) => //缓存命中
            //更新统计信息，将缓存作为结果返回
            context.taskMetrics.inputMetrics = Some(blockResult.inputMetrics)
            new InterruptibleIterator(context, blockResult.data.asInstanceOf[Iterator[T]])

        case None => //没有缓存命中，需要计算
            // 判断当前是否有线程在处理当前的Partition，如果有那么等待它结束后，直接从Block
            // Manager中读取处理结果如果没有线程在计算，那么storedValues就是None，否则
            // 就是计算的结果
            val storedValues = acquireLockForPartition[T](key)
            if (storedValues.isDefined) { // 已经被其他线程处理了，直接返回结果
                return new InterruptibleIterator[T](context, storedValues.get)
            }

            // 需要计算
            try {
            // 如果被checkpoint过，那么读取checkpoint的数据；否则调用rdd的compute()开始
            // 计算
            val computedValues = rdd.computeOrReadCheckpoint(partition,context)

            // Task是在Driver端执行的话就不需要缓存结果，这个主要是为了first() 或者take()
            // 这种仅仅有一个执行阶段的任务的快速执行。这类任务由于没有Shuffle阶段，直接运行
            // 在Driver端可能会更省时间
            if (context.isRunningLocally) {
                return computedValues
            }
            // 将计算结果写入到BlockManager
            val updatedBlocks = new ArrayBuffer[(BlockId, BlockStatus)]
            val cachedValues =
                putInBlockManager(key, computedValues, storageLevel, updatedBlocks)
        // 更新任务的统计信息
        val metrics = context.taskMetrics
        val lastUpdatedBlocks = metrics.updatedBlocks.getOrElse(
                                Seq[(BlockId, BlockStatus)]())
        metrics.updatedBlocks = Some(lastUpdatedBlocks ++ updatedBlocks.toSeq)
        new InterruptibleIterator(context, cachedValues)

    } finally {
        loading.synchronized {
        loading.remove(key)
        // 如果有其他的线程在等待该Partition的处理结果，那么通知它们计算已经完成，结果已
```

```
            // 经存到BlockManager中（注意前面那类不会写入BlockManager的本地任务）
            // loading.notifyAll()
                }
            }
        }
    }
```

3.4.4　checkpoint 的处理

在缓存没有命中的情况下，首先会判断是否保存了 RDD 的 checkpoint，如果有，则读取 checkpoint。为了理解 checkpoint 的 RDD 是如何读取计算结果的，需要先看一下 checkpoint 的数据是如何写入的。

首先在 Job 结束后，会判断是否需要 checkpoint。如果需要，就调用 org.apache.spark.rdd.RDDCheckpointData#doCheckpoint。doCheckpoint 首先为数据创建一个目录；然后启动一个新的 Job 来计算，并且将计算结果写入新创建的目录；接着创建一个 org.apache.spark.rdd.CheckpointRDD；最后，原始 RDD 的所有依赖被清除，这就意味着 RDD 的转换的计算链（compute chain）等信息都被清除。这个处理逻辑中，数据写入的实现在 org.apache.spark.rdd.CheckpointRDD$#writeToFile。简要的核心逻辑如下：

```
// 创建一个保存checkpoint数据的目录
     val path = new Path(rdd.context.checkpointDir.get, "rdd-" + rdd.id)
     val fs = path.getFileSystem(rdd.context.hadoopConfiguration)
     if (!fs.mkdirs(path)) {
         throw new SparkException("Failed to create checkpoint path " + path)
     }

     // 创建广播变量
     val broadcastedConf = rdd.context.broadcast(
         new SerializableWritable(rdd.context.hadoopConfiguration))
    //开始一个新的Job进行计算，计算结果存入路径path中
    rdd.context.runJob(rdd, CheckpointRDD.writeToFile[T](path.toString, broadcastedConf) _)
    //根据结果的路径path来创建CheckpointRDD
    val newRDD = new CheckpointRDD[T](rdd.context, path.toString)
    //保存结果，清除原始RDD的依赖、Partition信息等
    RDDCheckpointData.synchronized {
       cpFile = Some(path.toString)
       cpRDD = Some(newRDD) // RDDCheckpointData对应的CheckpointRDD
```

```
        rdd.markCheckpointed(newRDD)        // 清除原始RDD的依赖，Partition
        cpState = Checkpointed              //标记checkpoint的状态为完成
    }
```

至此，RDD 的 checkpoint 完成，其中 checkpoint 的数据可以通过 checkpointRDD 的 readFromFile 读取。但是，上述逻辑在清除了 RDD 的依赖后，并没有和 checkpointRDD 建立联系，那么 Spark 是如何确定一个 RDD 是否被 checkpoint 了，而且正确读取 checkpoint 的数据呢？

答案就在 org.apache.spark.rdd.RDD#dependencies 的实现，它会首先判断当前的 RDD 是否已经 Checkpoint 过，如果有，那么 RDD 的依赖就变成了对应的 CheckpointRDD：

```
privatedefcheckpointRDD: Option[RDD[T]]=checkpointData.flatMap(_.checkpointRDD)
final def dependencies: Seq[Dependency[_]] = {
   checkpointRDD.map(r => List(new OneToOneDependency(r))).getOrElse {
       if (dependencies_ == null) { //没有checkpoint
           dependencies_ = getDependencies
       }
       dependencies_
   }
}
```

理解了 Checkpoint 的实现过程，接下来看一下 computeOrReadCheckpoint 的实现。前面提到了，它一共在两个地方被调用，org.apache.spark.rdd.RDD#iterator 和 org.apache.spark.CacheManager#getOrCompute。它实现的逻辑比较简单，首先检查当前 RDD 是否被 Checkpoint 过，如果有，读取 Checkpoint 的数据；否则开始计算。实现如下：

```
private[spark] def computeOrReadCheckpoint(split: Partition, context: TaskContext)
   : Iterator[T] =
{
    if (isCheckpointed) firstParent[T].iterator(split, context) else compute(split, context)
}
```

firstParent[T].iterator(split, context) 会调用对应 CheckpointRDD 的 iterator，最终调用到它的 compute：

```
override def compute(split: Partition, context: TaskContext): Iterator[T] = {
    val file=new Path(checkpointPath, CheckpointRDD.splitIdToFile(split.index))
    CheckpointRDD.readFromFile(file, broadcastedConf, context) //读取Checkpoint的数据
}
```

3.4.5　RDD 的计算逻辑

RDD 的计算逻辑在 org.apache.spark.rdd.RDD#compute 中实现。每个特定的 RDD 都会实现 compute。比如前面提到的 CheckpointRDD 的 compute 就是直接读取 checkpoint 数据。HadoopRDD 就是读取指定 Partition 的数据。MapPartitionsRDD 就是将用户的转换逻辑作用到指定的 Partition 上。

3.5　RDD 的容错机制

RDD 实现了基于 Lineage 的容错机制。RDD 的转换关系，构成了 compute chain，可以把这个 compute chain 认为是 RDD 之间演化的 Lineage。在部分计算结果丢失时，只需要根据这个 Lineage 重算即可。

图 3-11 中，假如 RDD2 所在的计算作业先计算的话，那么计算完成后 RDD1 的结果就会被缓存起来。缓存起来的结果会被后续的计算使用。图中的示意是说 RDD1 的 Partition2 缓存丢失。如果现在计算 RDD3 所在的作业，那么它所依赖的 Partition0、1、3 和 4 的缓存都是可以使用的，无须再次计算。但是 Partition2 由于缓存丢失，需要从头开始计算，Spark 会从 RDD0 的 Partition2 开始，重新开始计算。

内部实现上，DAG 被 Spark 划分为不同的 Stage，Stage 之间的依赖关系可以认为就是 Lineage。关于 DAG 的划分可以参阅第 4 章。

提到 Lineage 的容错机制，不得不提 Tachyon。Tachyon 包含两个维度的容错，一个是 Tachyon 集群的元数据的容错，它采用了类似于 HDFS 的 Name Node 的元数据容错机制，即将元数据保存到一个 Image 文件，并且保存了元数据变化的编辑日志（EditLog）。另外一个是 Tachyon 保存的数据的容错机制，这个机制类似于 RDD 的 Lineage，Tachyon 会保留生成文件数据的 Lineage，在数据丢失时会通过这个 Lineage 来恢复数据。如果是 Spark 的数据，那么在数据丢失时 Tachyon 会启动 Spark 的 Job 来重算这部分内容。如果是 Hadoop 产生的数据，那么重新启动相应的 Map Reduce Job 就可以。现在 Tachyon 的容错机制的实现还处于开发阶段，并不推荐将这个机制应用于生产环境。不过，这并不影响 Spark 使用 Tachyon。如果 Spark

保存到 Tachyon 的部分数据丢失，那么 Spark 会根据自有的容错机制来重算这部分数据。

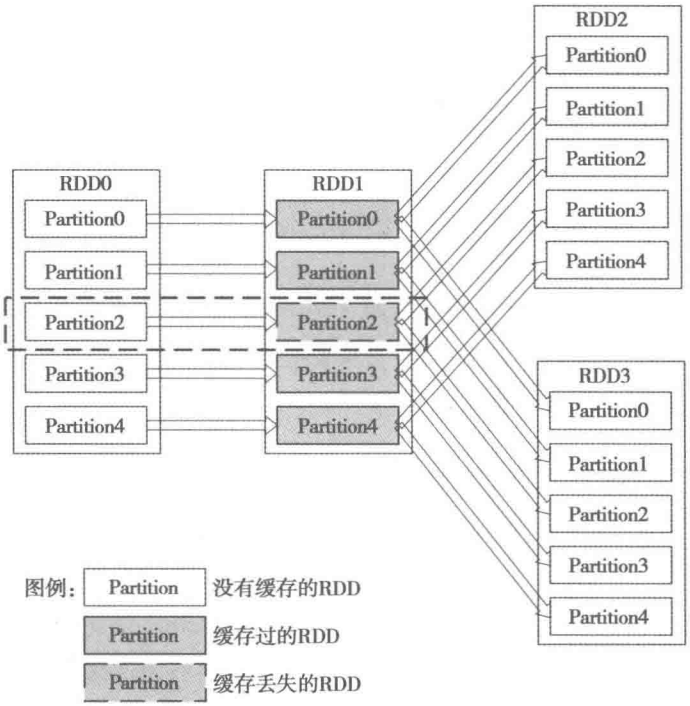
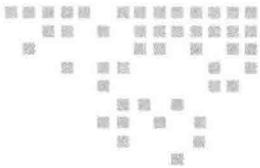


图 3-11 RDD 的部分缓存丢失的逻辑图

3.6 小结

RDD 是 Spark 最基本，也是最根本的数据抽象。RDD 是只读的、分区记录的集合。RDD 只能基于在稳定物理存储中的数据集和其他已有的 RDD 上执行确定性操作来创建。这些确定性操作称为转换，如 map、filter、groupBy、join。RDD 支持丰富的转换操作，极大地简化了用户应用的编写。

RDD 不需要物化。RDD 含有如何从其他 RDD 衍生（即计算）出本 RDD 的相关信息（即 Lineage），据此在 RDD 部分分区数据丢失时可以通过物理存储的数据计算出相应的 RDD 分区。



第 4 章 Chapter 4

Scheduler 模块详解

Scheduler（任务调度）模块作为 Spark Core 的核心模块之一，充分地体现了与 MapReduce 完全不同的设计思想。Spark 对于 DAG（Directed Acyclic Graph，有向无环图）的实现及不同执行阶段的划分和任务的提交执行，充分体现了其设计的优雅和高效。本章主要介绍任务调度，即组成应用的多个 Job 之间如何分配计算资源。

4.1 模块概述

4.1.1 整体架构

任务调度模块主要包含两大部分，即 DAGScheduler 和 TaskScheduler，它们负责将用户提交的计算任务按照 DAG 划分为不同的阶段并且将不同阶段的计算任务提交到集群进行最终的计算。整个过程可以使用图 4-1 表示。

RDD Objects 可以理解为用户实际代码中创建的 RDD，这些代码逻辑上组成了一个 DAG，支持复杂拓扑。Spark 的易用性就体现在这部分，它提供了基于 RDD 的多种转换和动作，使 Spark 的用户可以在基本不增加用户学习成本的前提下使用比较复杂的拓扑来实现策略。用户实际编程的时候可以认为它处理的数据都可以存到内存中，

而无须关心最终在集群中运行的任务是否整个数据可以装载到内存中或者说究竟需要多少节点参与运算（如果对运算速度或者性能有要求，需要设置一些合适的参数，请具体参阅第 6.7.8 章的性能调优部分）。

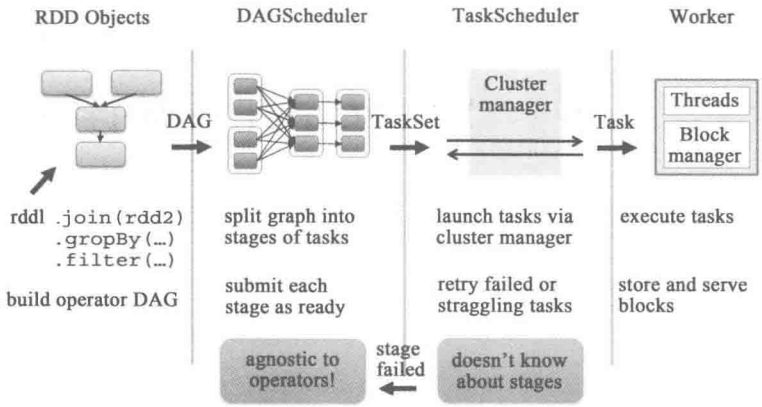


图 4-1 任务调度逻辑视图

DAGScheduler 主要负责分析用户提交的应用，并根据计算任务的依赖关系建立 DAG，然后将 DAG 划分为不同的 Stage（阶段），其中每个 Stage 由可以并发执行的一组 Task 构成，这些 Task 的执行逻辑完全相同，只是作用于不同的数据。而且 DAG 在不同的资源管理框架（即部署方式，包括 Standalone、Mesos、YARN、Local、EC2 等）下的实现是相同的。

在 DAGScheduler 将这组 Task 划分完成后，会将这组 Task 提交到 TaskScheduler。TaskScheduler 通过 Cluster Manager 在集群中的某个 Worker 的 Executor 上启动任务。在 Executor 中运行的任务，如果缓存中没有计算结果，那么就需要开始计算，同时，计算的结果会回传到 Driver 或者保存在本地。在不同的资源管理框架下，TaskScheduler 的实现方式是有差别的，但是最重要的实现是 org.apache.spark.scheduler.TaskSchedulerImpl。对于 Local、Standalone 和 Mesos 来说，它们的 TaskScheduler 就是 TaskSchedulerImpl；对于 YARN Cluster 和 YARN Client 的 TaskScheduler 的实现也是继承自 TaskSchedulerImpl。

4.1.2　Scheduler 的实现概述

任务调度模块涉及的最重要的三个类是：

1）org.apache.spark.scheduler.DAGScheduler

2）org.apache.spark.scheduler.SchedulerBackend

3）org.apache.spark.scheduler.TaskScheduler

其中 1 就是前面提到的 DAGScheduler 的实现。

org.apache.spark.scheduler.SchedulerBackend 是一个 trait，作用是分配当前可用的资源，具体就是向当前等待分配计算资源的 Task 分配计算资源（即 Executor），并且在分配的 Executor 上启动 Task，完成计算的调度过程。它使用 reviveOffers 完成上述的任务调度。reviveOffers 可以说是它最重要的实现。org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend 是 SchedulerBackend 的一个实现，同时 YARN、Standalone 和 Mesos 都是基于它加入了自身特有的逻辑。

org.apache.spark.scheduler.TaskScheduler 也是一个 trait，它的作用是为创建它的 SparkContext 调度任务，即从 DAGScheduler 接收不同 Stage 的任务，并且向集群提交这些任务，并为执行特别慢的任务启动备份任务。TaskScheduler 是以后实现多种任务调度器的基础，不过当前的 org.apache.spark.scheduler.TaskSchedulerImpl 是唯一实现。TaskSchedulerImpl 会在以下几种场景下调用 org.apache.spark.scheduler.SchedulerBackend#reviveOffers：

1）有新任务提交时。

2）有任务执行失败时。

3）计算节点（即 Executor）不可用时。

4）某些任务执行过慢而需要为它重新分配资源时。

图 4-2 描述了 Standalone、YARN 和 Local 部署方式的 TaskScheduler 和 SchedulerBackend 的类图。

每个 SchedulerBackend 都会对应一个的唯一 TaskScheduler，而它们都被 Spark-

Context 创建和持有。

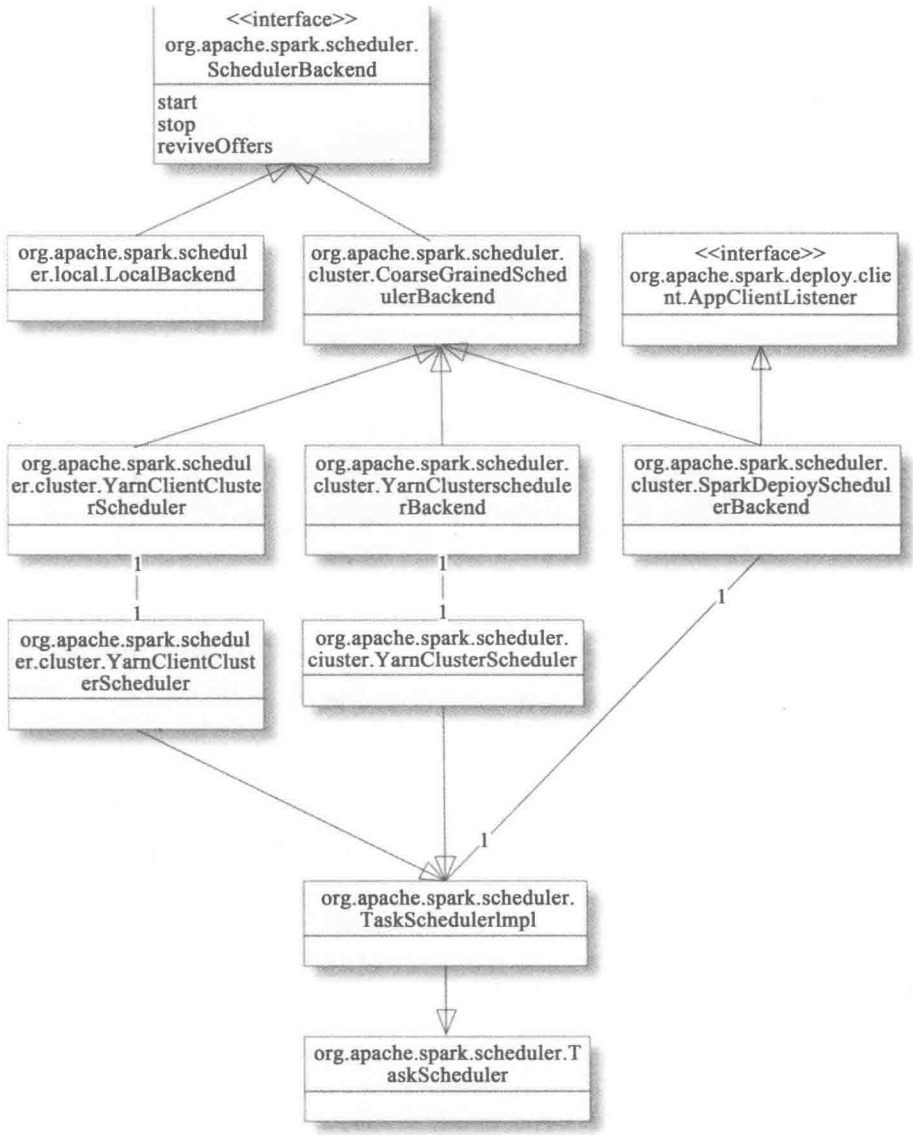


图 4-2 模块类图

类图中没有包含 Mesos 的实现，实际上，Mesos 的 TaskScheduler 也是 TaskScheduler-Impl，而 Mesos 有两种 SchedulerBackend，一个就是“粗放型”（coarsegrained）的，即

org.apache.spark.scheduler.cluster.mesos.CoarseMesos-SchedulerBackend，它继承自 org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend，并加入了 Mesos 调度自身特有的逻辑；另外一个就是“细粒度”（fine-grained）的，即 org.apache.spark.scheduler.cluster.mesos.Mesos-SchedulerBackend，它将每个 Spark 的 Task 映射到一个 Mesos 的 Task，从而将这个任务的执行与调度的控制权交给 Mesos。

图 4-3 是任务调度的逻辑图，本章接下来将会详细讨论实现原理和设计思想。

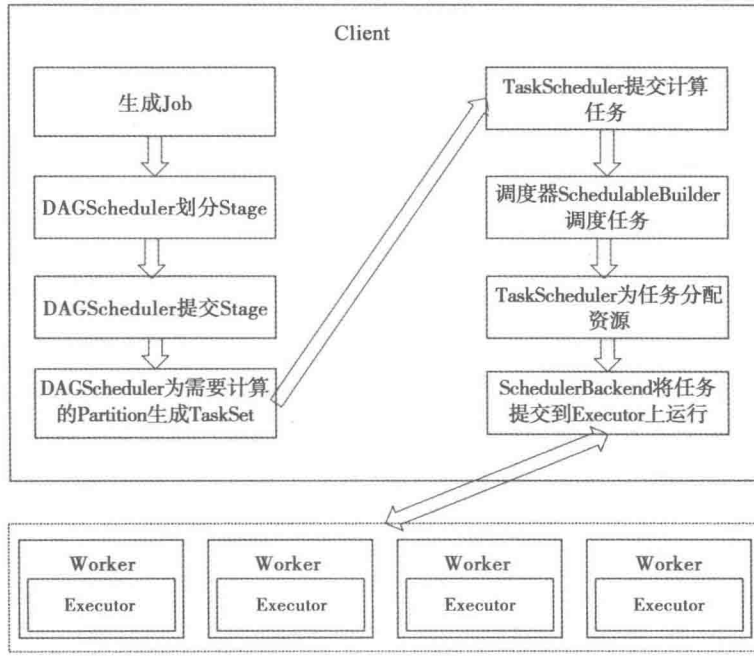


图 4-3　任务调度的逻辑图

4.2　DAGScheduler 实现详解

前面提到过，DAGScheduler 主要负责将用户的应用的 DAG 划分为不同的 Stage（阶段），其中每个 Stage 由可以并发执行的一组 Task 构成，这些 Task 的执行逻辑完全相同，只是作用于不同的数据。而且 DAG 在不同的资源管理框架（即部署方式，包括 Standalone，Mesos，YARN，Local，EC2 等）下的实现是相同的。

以 RDD 的 Action count 为例，解释一下 Spark 的内部实现原理。首先，RDD 的 count 实际上是调用 SparkContext 的 org.apache.spark.SparkContext#runJob 来进行提交 Job 的：

```
def count(): Long = sc.runJob(this, Utils.getIteratorSize _).sum
```

SparkContext 实现了很多 runJob，这些函数的区别就是调用参数的不同，不过这些 runJob 最后都会调用 DAGScheduler 的 runJob：

```
dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, allowLocal,
    resultHandler, localProperties.get)
```

而 DAGScheduler 的 runJob 会开始对用户提交的 Job 进行处理，包括 Stage 的划分、Task 的生成等。接下来，将从 DAGScheduler 的创建，用户提交 Job 的处理，到结果的生成来详细讲解这个过程。

4.2.1 DAGScheduler 的创建

TaskScheduler 和 DAGScheduler 都是在 SparkContext 创建的时候创建的。其中，Taskcheduler 是通过 org.apache.spark.SparkContext#createTaskScheduler 创建的。而 DAGScheduler 是直接调用的它的构造函数创建。只不过，DAGScheduler 保存了 TaskScheduler 的引用，因此需要在 TaskScheduler 创建之后进行创建。

在 SparkContext 中创建 DAGScheduler：

```
dagScheduler = new DAGScheduler(this)
```

这个构造函数的实现是：

```
def this(sc: SparkContext) = this(sc, sc.taskScheduler)
```

继续看 this(sc, sc.taskScheduler) 的实现：

```
def this(sc: SparkContext, taskScheduler: TaskScheduler) = {
    this(
        sc,
        taskScheduler,
        sc.listenerBus,
        sc.env.mapOutputTracker.asInstanceOf[MapOutputTrackerMaster],
        sc.env.blockManager.master,
        sc.env)
}
```

this(sc, sc.taskScheduler) 通过调用下面的构造函数完成 DAGScheduler 的创建：

```
private[spark]
class DAGScheduler(
    private[scheduler] val sc: SparkContext,
    private[scheduler] val taskScheduler: TaskScheduler,
    listenerBus: LiveListenerBus,
    mapOutputTracker: MapOutputTrackerMaster,
    blockManagerMaster: BlockManagerMaster,
    env: SparkEnv,
    clock: Clock = SystemClock)
extends Logging {
}
```

其中，MapOutputTrackerMaster 是运行在 Driver 端管理 Shuffle Map Task 的输出的，下游的 Task 可以通过 MapOutputTrackerMaster 来获取 Shuffle 输出的位置信息，具体的实现详情请参阅第 7 章。BlockManagerMaster 也是运行在 Driver 端的，它管理整个 Job 的 Block 的信息，具体的实现详情请参阅第 8 章。

DAGScheduler 除了初始化用于保存集群状态信息的数据结构，还会创建一个 Actor-（org.apache.spark.scheduler.DAGSchedulerEventProcessActor，变量的名称为 eventProcessActor），这个 Actor 的主要职责就是调用 DAGScheduler 的相应方法来处理 DAGScheduler 发送给它的各种消息。这个 Actor 的创建比较特殊，需要重点梳理一下。

由于要保证 eventProcessActor 在 DAGScheduler 创建完成之后就需要处于可用的状态，因此 Spark 引入了 org.apache.spark.scheduler.DAGSchedulerActorSupervisor，由它来完成 eventProcessActor 的创建。

DAGSchedulerActorSupervisor 主要利用了 Actor 的监督策略，即它创建的子 Actor 在出现错误时是如何进行错误处理；如果它的子 Actor（就是 eventProcessActor）出现错误，则取消 DAGScheduler 的所有 Job，停止 SparkContext，最终退出。DAGSchedulerActorSupervisor 只处理一种消息，即创建一个 Actor：

```
def receive = {
    case p: Props => sender ! context.actorOf(p)
    case _ => logWarning("received unknown message in DAGSchedulerActorSupervisor")
```

```
}
DAGSchedulerEventProcessActor的创建过程:
implicit val timeout = Timeout(30 seconds)
val initEventActorReply =
    dagSchedulerActorSupervisor ? Props(new DAGSchedulerEventProcessActor(this))
eventProcessActor = Await.result(initEventActorReply, timeout.duration).
    asInstanceOf[ActorRef]
```

说明一下，向 Actor 发送消息！和？的区别是：！发送消息后立即返回；而？发送消息后会返回一个 Future 对象。因此 eventProcessActor 是一个可用的 Actor，到这里 DAGScheduler 已经可以接受外界的消息，开始提供服务了。

4.2.2 Job 的提交

前面提到过，用户提交的 Job 最终会调用 DAGScheduler 的 runJob，它又会调用 submitJob。以 RDD 的动作 count 为例，调用过程如下：

```
1) org.apache.spark.rdd.RDD#count
2) org.apache.spark.SparkContext#runJob
3) org.apache.spark.scheduler.DAGScheduler#runJob
4) org.apache.spark.scheduler.DAGScheduler#submitJob
5) org.apache.spark.scheduler.DAGSchedulerEventProcessActor#receive
   (JobSubmitted)
6) org.apache.spark.scheduler.DAGScheduler#handleJobSubmitted
```

调用栈 3 主要的实现就是调用 submitJob 来处理：

```
val waiter = submitJob(rdd, func, partitions, callSite, allowLocal, resultHandler,properties)
waiter.awaitResult() match {
    case JobSucceeded => {
        logInfo("Job %d finished: %s, took %f s".format
            (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
    }
    case JobFailed(exception: Exception) =>
        logInfo("Job %d failed: %s, took %f s".format
            (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
        throw exception
}
```

而 submitJob 首先会为这个 Job 生成一个 Job ID，并且生成一个 JobWaiter 的实例来监听 Job 的执行情况：

```
val waiter = new JobWaiter(this, jobId, partitions.size, resultHandler)
```

JobWaiter 会监听 Job 的执行状态，而 Job 是由多个 Task 组成的，因此只有 Job 的所有 Task 都成功完成，Job 才标记为成功；任意一个 Task 失败都会标记该 Job 失败，这是 DAGScheduler 通过调用 org.apache.spark.scheduler.JobWaiter#jobFailed 实现的。

对于近似估计的 Job，DAGScheduler 会调用 runApproximateJob，除了 JobWaiter 换成了 org.apache.spark.partial.ApproximateActionListener，其余的逻辑是相同的。

最后，DAGScheduler 会向 eventProcessActor 提交该 Job：

```
eventProcessActor ! JobSubmitted(
   jobId, rdd, func2, partitions.toArray, allowLocal, callSite, waiter, properties)
```

eventProcessActor 最终会调用 org.apache.spark.scheduler.DAGScheduler#handleJobSubmitted 来处理这次提交的 Job。

4.2.3　Stage 的划分

org.apache.spark.scheduler.DAGScheduler#handleJobSubmitted 首先会根据 RDD 创建 finalStage。finalStage，顾名思义，就是最后的那个 Stage。然后创建 ActiveJob 后提交计算任务。在讨论这个过程的实现之前，先看一下究竟什么是 Stage。

用户提交的计算任务是一个由 RDD 构成的 DAG，如果 RDD 在转换的时候需要做 Shuffle，那么这个 Shuffle 的过程就将这个 DAG 分为了不同的阶段（即 Stage）。由于 Shuffle 的存在，不同的 Stage 是不能并行计算的，因为后面 Stage 的计算需要前面 Stage 的 Shuffle 的结果。而一个 Stage 由一组完全独立的计算任务（即 Task）组成，每个 Task 的运算逻辑完全相同，只不过每个 Task 都会处理其所对应的 Partition。其中，Partition 的数量和 Task 的数量是一致的，即一个 Partition 会被该 Stage 的一个 Task 处理。

1. 划分依据

那么 RDD 在哪种转换的时候需要 Shuffle 呢？这个取决于该 RDD 和它所依赖的 RDD(s) 的关系。RDD 和它依赖的 parent RDD(s) 的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）。

对于窄依赖，由于 RDD 每个 Partition 依赖固定数量的 parent RDD 的 Partition，因此可以通过一个 Task 来处理这些 Partition，而且这些 Partition 相互独立，所以这些 Task 可以并行执行。

对于宽依赖，由于需要 Shuffle，因此只有所有的 parent RDD 的 Partition Shuffle 完成，新的 Partition 才会形成，这样接下来的 Task 才可以继续处理。因此，宽依赖可以认为是 DAG 的分界线，或者说 Spark 根据宽依赖将 Job 划分为不同的阶段（Stage）。

关于窄依赖和宽依赖的定义，请查看 3.3.1 小节。

2. 划分过程

理解了窄依赖和宽依赖，下面以图 4-4 为例讲解 Stage 的具体划分过程。

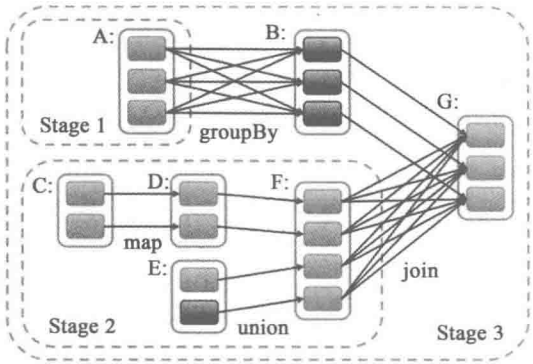


图 4-4　RDD 划分示意

Stage 的划分是从最后一个 RDD 开始的，也就是触发 Action 的那个 RDD。图 4-4 中就是 RDD G。首先，RDD G 会从 SparkContext 的 runJob 开始，通过以下的调用栈来开始 Stage 的划分：

```
1）org.apache.spark.SparkContext#runJob
2）org.apache.spark.scheduler.DAGScheduler#runJob
3）org.apache.spark.scheduler.DAGScheduler#submitJob
4）org.apache.spark.scheduler.DAGSchedulerEventProcessActor#receive
   （JobSubmitted）
5）org.apache.spark.scheduler.DAGScheduler#handleJobSubmitted
```

handleJobSubmitted 会开始 Stage 的划分。划分从 RDD G 的依赖开始，RDD G

依赖于两个 RDD，一个是 RDD B，一个是 RDD F。其中先处理 RDD B 还是 RDD F 是随机的。这里首先处理 RDD B，由于和 RDD B 的依赖是窄依赖，因此 RDD G 和 RDD B 可以划分到一个 Stage（即 Stage 3）。再处理 RDD F，由于这个依赖是宽依赖，RDD F 和 RDD G 被划分到不同的 Stage（即 Stage 2 和 Stage 3），其中 RDD F 所在的 Stage 2 是 RDD G 所在的 Stage 3 的 parent Stage。

接下来处理 RDD B 的依赖，由于它依赖的 RDD A 是宽依赖，因此它们属于不同的 Stage（即 Stage 1 和 Stage 3）。这样，RDD B 和 RDD G 同属于 Stage 3，而这个 Stage 直接的 parent Stage 有两个，就是 RDD A 和 RDD F 分别属于的两个 Stage（即 Stage 1 和 Stage 2）。接下来的划分采用相同的逻辑，递归进行。

最后，这个 DAG 被划分为三个 Stage，即 RDD A 所在的 Stage 1，RDD C、RDD D、RDD E、RDD F 所在的 Stage 2 和 RDD B 及 RDD G 所在的 Stage 3。其中 Stage 1 和 Stage 2 是相互独立的，可以并发执行；Stage 3 依赖于 Stage 1 和 Stage 2，只有 Stage 1 和 Stage 2 完成计算，它才可以开始计算。

那么 Stage 是如何计算的？以 Stage 1 为例，由于 RDD A 有三个 Partition，因此它会生成三个 org.apache.spark.scheduler.ShuffleMapTask，这些 Task 会将结果写入到三个 Partition，实现细节可以参阅第 7 章。Stage 2 同样也是由 ShuffleMapTask 组成。不过 Stage 3 是由三个 org.apache.spark.scheduler.ResultTask 构成的。

3. 实现细节

handleJobSubmitted 通过调用 org.apache.spark.scheduler.DAGScheduler#newStage 来创建 finalStage，即上图中的 Stage 3：

```
finalStage = newStage(finalRDD, partitions.size, None, jobId, callSite)
```

newStage 首先会获取当前 Stage 的 parent Stage，然后创建当前的 Stage：

```
val parentStages = getParentStages(rdd, jobId)
val id = nextStageId.getAndIncrement()
val stage = new Stage(id, rdd, numTasks, shuffleDep, parentStages, jobId,
   callSite)
```

通过调用 getParentStages，图 4-4 中的 Stage 1 和 Stage 2 就创建出来了。然后根

据它们来创建 Stage 3。

getParentStages 是划分 Stage 的核心实现。它负责生成当前 RDD 所在的 Stage 的 parent Stage。它的实现是每遇到一个 ShuffleDependency（宽依赖的一种），就会生成一个 parent Stage。下面是源码实现：

```
private def getParentStages(rdd: RDD[_], jobId: Int): List[Stage] = {
    val parents = new HashSet[Stage]      //存储parent stage
    val visited = new HashSet[RDD[_]]     //存储已经被访问到得RDD
    //存储需要被处理的RDD，Stack中的RDD都需要被处理
    val waitingForVisit = new Stack[RDD[_]]
    def visit(r: RDD[_]) {                //广度优先遍历rdd生成的依赖树
        if (!visited(r)) {
        visited += r
        for (dep <- r.dependencies) {     //逐个处理当前RDD依赖的parent RDD
            dep match {
            case shufDep: ShuffleDependency[_, _, _] =>
            // 在依赖是ShuffleDependency时需要生成新的stage
            parents += getShuffleMapStage(shufDep, jobId)
       case _ =>
           //不是ShuffleDependency，那么就属于同一个Stage
           waitingForVisit.push(dep.rdd)
       }
    }
  }
}
// 以输入的rdd作为第一个需要处理的RDD。然后从该rdd开始，顺序处理其parent rdd
   waitingForVisit.push(rdd)
   while (!waitingForVisit.isEmpty) { //只要stack不为空，则一直处理。
        // 每次visit如果遇到了ShuffleDependency，那么就会形成一个Stage，否则这些RDD属于
           同一个Stage
        visit(waitingForVisit.pop())
    }
    parents.toList
}
```

getShuffleMapStage 就是获取 ShuffleDependency 所依赖的 Stage，如果没有则创建新的 Stage：

```
private def getShuffleMapStage(shuffleDep: ShuffleDependency[_, _, _], jobId: Int):
    Stage = {
shuffleToMapStage.get(shuffleDep.shuffleId) match {//根据shuffleId查找Stage是否存在
```

```
        case Some(stage) => stage //如果已经创建，则直接返回
        case None =>
          // 看是否该Stage的Parent Stage已经生成，如果没有，则生成它们
          registerShuffleDependencies(shuffleDep, jobId)
          // 生成当前rdd所在的Stage
          val stage =
              newOrUsedStage(
              shuffleDep.rdd, shuffleDep.rdd.partitions.size, shuffleDep, jobId,
              shuffleDep.rdd.creationSite)
           shuffleToMapStage(shuffleDep.shuffleId) = stage

           stage
        }
    }
```

上述代码中有两个调用比较重要：一个是 registerShuffleDependencies，负责确认该 Stage 的 parent Stage 是否已经生成，如果没有则生成它们；另外一个是 newOrUsedStage，它也是生成一个 Stage，不过如果这个 Stage 已存在，那么将恢复这个 Stage 的结果，从而避免了重复计算。registerShuffleDependencies 的实现如下：

```
private def registerShuffleDependencies(shuffleDep: ShuffleDependency[_, _, _],
                                        jobId: Int) = {
    //首先获取没有生成Stage的ShuffleDependency
    val parentsWithNoMapStage = getAncestorShuffleDependencies(shuffleDep.rdd)
    while (!parentsWithNoMapStage.isEmpty) {
        val currentShufDep = parentsWithNoMapStage.pop()
        //根据ShuffleDependency来生成Stage
        val stage =
            newOrUsedStage(
                currentShufDep.rdd, currentShufDep.rdd.partitions.size,
                    currentShufDep, jobId,
                currentShufDep.rdd.creationSite)
        shuffleToMapStage(currentShufDep.shuffleId) = stage
     }
}
```

getAncestorShuffleDependencies 的实现和 getParentStages 差不过，只不过它遇到 ShuffleDependency 时首先会判断 Stage 是否已经存在，不存在则把这个依赖作为返回值的一个元素，由调用者来完成 Stage 的创建。

newOrUsedStage 的实现如下：

```
val stage = newStage(rdd, numTasks, Some(shuffleDep), jobId, callSite)
if (mapOutputTracker.containsShuffle(shuffleDep.shuffleId)) {
    //Stage已经被计算过，从mapOutputTracer中获取计算结果
    val serLocs = mapOutputTracker.getSerializedMapOutputStatuses(shuffle
       Dep.shuffleId)
    val locs = MapOutputTracker.deserializeMapStatuses(serLocs)
    for (i <- 0 until locs.size) {
        //计算结果复制到stage中
        stage.outputLocs(i) = Option(locs(i)).toList   // locs(i) will be null if missing
   }
   //保存stage可用结果的数量；对于不可用的部分，会被重新计算
   stage.numAvailableOutputs = locs.count(_ != null)
} else {
    //向mapOutputTracer中注册该Stage
    mapOutputTracker.registerShuffle(shuffleDep.shuffleId, rdd.partitions.size)
}
```

ShuffleMapTask 的计算结果都会传给 Driver 端的 mapOutputTracker，其他的 Task 可以通过查询它来获取这些结果。mapOutputTracker.registerShuffle(shuffleDep.shuffleId, rdd.partitions.size) 实现了存储这些元数据的占位，而 ShuffleMapTask 的结果通过调用 registerMapOutputs 来保存计算结果。这个结果并不是真实的数据，而是这些数据所在的位置、大小等元数据信息，这样下游的 Task 就可以通过这些元数据信息获取其需要处理的数据。这个实现详情可以参阅第 7 章。

4.2.4 任务的生成

现在让我们回到 org.apache.spark.scheduler.DAGScheduler#handleJobSubmitted，生成 finalStage 后就会为该 Job 生成一个 org.apache.spark.scheduler.ActiveJob，并准备计算这个 finalStage：

```
val job = new ActiveJob(jobId, finalStage, func, partitions, callSite, listener,
    properties)
```

这个 Job 如果满足以下所有条件，那么它将以本地模式运行：

1）spark.localExecution.enabled 设置为 true

2）用户程序显式指定可以本地运行

3）finalStage 没有 parent Stage

4）仅有一个 Partition

其中，3 和 4 主要为了保证任务可以快速执行；如果有多个 Stage 或者多个 Partition 的话，本地运行可能会因为本机计算资源的局限性而影响任务的计算速度。

除了本地运行的，handleJobSubmitted 会调用 submitStage 来提交这个 Stage，如果它的某些 parent Stage 没有提交，那么递归提交那些未提交（或者叫未计算）的 parent Stage。只有所有的 parent Stage 都计算完成后，才能提交它：

```
private def submitStage(stage: Stage) {
    val jobId = activeJobForStage(stage)
    if (jobId.isDefined) {
        logDebug("submitStage(" + stage + ")")
        // 如果当前stage不再等待其parent stage的返回，不是正在运行，且没有提示失败，那么
        // 就尝试提交它
        if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage))
{
            val missing = getMissingParentStages(stage).sortBy(_.id)
            logDebug("missing: " + missing)
            if (missing == Nil) {
               // 如果所有的parent stage都已经完成，那么提交该stage所包含的task
               logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no
                   missing parents")
               submitMissingTasks(stage, jobId.get)
            } else {
                for (parent <- missing) {
                    // 有parent stage未完成，则递归提交它
                    submitStage(parent)
                }
                waitingStages += stage
            }
    }
    } else { //无效的stage，直接停止它
        abortStage(stage, "No active job for stage " + stage.id)
    }
}
```

前面图 4-4 划分的 Stage 的提交逻辑如图 4-5 所示。三个 Stage 中，Stage1 和 Stage2 是可以并行的，因此在有计算资源的情况下会首先被提交，并且在这两个 Stage 都计算完成的情况下，提交 Stage 3。最后的计算都是在计算节点的 Executor 中执行的。

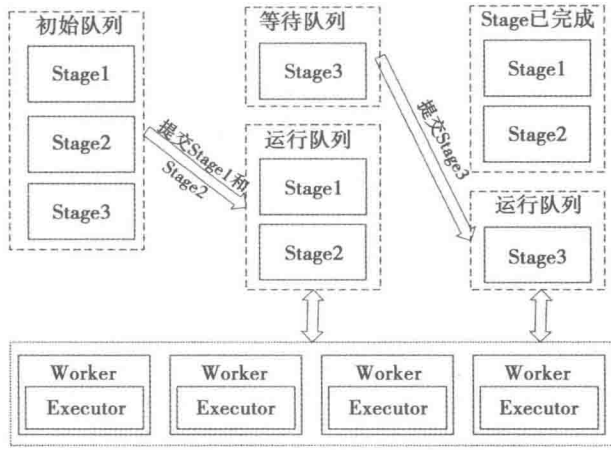


图 4-5 Stage 提交顺序图

org.apache.spark.scheduler.DAGScheduler#submitMissingTasks 会完成 DAGScheduler 最后的工作，向 TaskScheduler 提交 Task。首先，它先取得需要计算的 Partition，对于最后的 Stage，它对应的 Task 是 ResultTask，因此判断该 Partition 的 ResultTask 是否已经结束，如果结束那么就无需计算；对于其他的 Stage，它们对应的 Task 都是 ShuffleMapTask，因此只需要判断 Stage 是否有缓存的结果即可（结果都会缓存到 org.apache.spark.scheduler.Stage#outputLocs 中）。

在判断出哪些 Partition 需要计算后，就会为每个 Partition 生成 Task，然后这些 Task 会被封装到 org.apache.spark.scheduler.TaskSet#TaskSet，然后提交给 TaskScheduler。从逻辑上，Stage 包含了 TaskSet，图 4-5 的划分结果的逻辑视图就转换成了图 4-6。

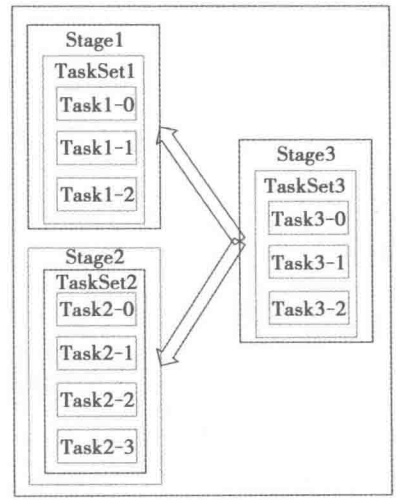


图 4-6 Task 被保存到 TaskSet 后的逻辑视图

TaskSet 保存了 Stage 包含的一组完全相同的 Task，每个 Task 的处理逻辑完全相同，不同的是处理数据，每个 Task 负责处理一个 Partition。对于一个 Task 来说，它从数据源获得逻辑，然后按照拓扑顺序，顺序执行。

TaskSet 是一个数据结构，存储了这一组 Task：

```
private[spark] class TaskSet(
    val tasks: Array[Task[_]],
    val stageId: Int,
    val attempt: Int,
    val priority: Int,
    val properties: Properties) {
    val id: String = stageId + "." + attempt
override def toString: String = "TaskSet " + id
}
```

至此，DAGScheduler 就完成了它的使命，然后它会等待 TaskScheduler 最终向集群提交这些 Task，监听这些 Task 的状态。

4.3　任务调度实现详解

每个 TaskScheduler 都对应一个 SchedulerBackend。其中，TaskScheduler 负责 Application 的不同 Job 之间的调度，在 Task 执行失败时启动重试机制，并且为执行速度慢的 Task 启动备份的任务。而 SchedulerBackend 负责与 Cluster Manager 交互，取得该 Application 分配到的资源，并且将这些资源传给 TaskScheduler，由 TaskScheduler 为 Task 最终分配计算资源。

4.3.1　TaskScheduler 的创建

TaskScheduler 和 DAGScheduler 都是在 SparkContext 创建的时候创建的。其中，TaskScheduler 是通过 org.apache.spark.SparkContext#createTaskScheduler 创建的。

org.apache.spark.SparkContext#createTaskScheduler 会根据传入的 Master 的 URL 的规则判断集群的部署方式（或者说是资源管理方式），比如是 Standalone、Mesos、YARN 或者是 Local 等。根据不同的部署方式，生成不同的 TaskScheduler 和 SchedulerBackend。org.apache.spark.scheduler.SchedulerBackend 是一个 trait，作用是分配当前可用的资源，具体就是向当前等待分配计算资源的 Task 分配计算资源（即 Executor），并且在分配的 Executor 上启动 Task，完成计算的调度过程。它使用 reviveOffers 完成上述的任务调度。reviveOffers 可以说是它最重要的实现。

以 Standalone 模式为例，创建过程如下：

```
val SPARK_REGEX = """spark://(.*)""".r
master match {
    case SPARK_REGEX(sparkUrl) =>
        val scheduler = new TaskSchedulerImpl(sc)
        val masterUrls = sparkUrl.split(",").map("spark://" + _)
        val backend = new SparkDeploySchedulerBackend(scheduler, sc, masterUrls)
        scheduler.initialize(backend)
        (backend, scheduler)
```

如果需要开启任务的推测执行，那么需要将 spark.speculation 设置为 true，在 TaskScheduler 创建完成后，会周期性地调用 org.apache.spark.scheduler.TaskSetManager 的 checkSpeculatableTasks 来检查是否有这种 Task：

```
sc.env.actorSystem.scheduler.schedule(SPECULATION_INTERVAL milliseconds,
    SPECULATION_INTERVAL milliseconds) {
    Utils.tryOrExit { checkSpeculatableTasks() }
}
```

周期性的间隔是通过 spark.speculation.interval 设置的，单位是毫秒；默认是 100 毫秒。

4.3.2 Task 的提交概述

DAGScheduler 完成了对 Stage 的划分后，会按照顺序逐个将 Stage 通过下面的调用提交到 TaskScheduler：

```
org.apache.spark.scheduler.TaskSchedulerImpl#submitTasks
```

由 submitTasks 开始 Task 级别的资源调度。最终，这些 Task 会被分配 Executor，运行在 Worker 上的 Executor 完成任务的最终执行。这个过程详细的调用堆栈如下：

```
1)org.apache.spark.scheduler.TaskSchedulerImpl#submitTasks
2)org.apache.spark.scheduler.SchedulableBuilder#addTaskSetManager
3)org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend#reviveOffers
4)org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend.
  DriverActor#makeOffers
5)org.apache.spark.scheduler.TaskSchedulerImpl#resourceOffers
6)org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend.
  DriverActor#launchTasks
7)org.apache.spark.executor.CoarseGrainedExecutorBackend.
```

```
  receiveWithLogging#launchTask
8)org.apache.spark.executor.Executor#launchTask
```

这只是一个简化了的调用栈。每个调用可能还会调用很多其他的任务，接下来会详解每个调用的具体实现。这个简化的调用栈只是为了说明任务是如何从 TaskScheduler 开始，最终提交的到 Executor 上去的。其中，调用栈 1 ～ 6 都是在 Driver 端的；7 ～ 8 就到了 Executor 上。8 最终完成了任务的执行。关于 7 和 8 任务执行的详情，可以参阅第 6 章。本节主要讨论在 Driver 端的调度处理的实现。

在调用栈 1 中，主要是将保存这组任务的 TaskSet 加入到一个 TaskSetManager 中：

```
val manager = new TaskSetManager(this, taskSet, maxTaskFailures)
```

TaskSetManager 会根据数据的就近原则（locality aware）为 Task 分配计算资源，监控 Task 的执行状态并采取必要的措施，比如失败重试，慢任务的推测性执行。

调用栈 2 对应的源码是：

```
schedulableBuilder.addTaskSetManager(manager, manager.taskSet.properties)
```

schedulableBuilder 是 Application 级别的调度器，现在支持两种调度策略，FIFO（Fist In First Out，先进先出）和 FAIR（公平调度）。调度策略可以通过 spark.scheduler.mode 设置，默认是 FIFO。schedulableBuilder 会确定 TaskSetManager 的调度顺序，然后由 TaskSetManager 根据就近原则来确定 Task 运行在哪个 Executor 上。

调用栈 3 和 4 并没有特别复杂的逻辑。

调用栈 5 就是响应 CoarseGrainedSchedulerBackend 的资源调度请求，为每个 Task 具体分配资源。它的输入是一个 Executor 的列表，输出是 org.apache.spark.scheduler.TaskDescription 的二维数组。TaskDescription 包含了 Task ID、Executor ID 和 Task 执行环境的依赖信息等。省略了非关键代码的具体实现如下：

```
def resourceOffers(offers: Seq[WorkerOffer]): Seq[Seq[TaskDescription]] =
synchronized {
    // Mark each slave as alive and remember its hostname
    // Also track if new executor is added
    var newExecAvail = false
    for(o <- offers) {
```

```
    if (!executorsByHost.contains(o.host)) {
       //有新的Executor加入
       executorsByHost(o.host) = new HashSet[String]()
       executorAdded(o.executorId, o.host)
       newExecAvail = true
    }
}
// 为了避免将Task集中分配到某些机器，随机打散它们
val shuffledOffers = Random.shuffle(offers)
// 存储分配好资源的task
val tasks = shuffledOffers.map(o => new ArrayBuffer[TaskDescription](o.cores))
val availableCpus = shuffledOffers.map(o => o.cores).toArray
// 非常重要，获取按照调度策略排序好的TaskSetManager
val sortedTaskSets = rootPool.getSortedTaskSetQueue
for (taskSet <- sortedTaskSets) {
    if (newExecAvail) {
        taskSet.executorAdded() //重新计算该TaskSetManager的就近原则
    }
}

// 为从rootPool里获取的TaskSetManager列表分配资源。分配的原则是就近原则，优先分配顺
// 序PROCESS_LOCAL、NODE_LOCAL、NO_PREF、RACK_LOCAL, ANY
 var launchedTask = false
 for (taskSet <- sortedTaskSets; maxLocality <- taskSet.myLocalityLevels) {
      do {
         launchedTask = false
         for (i <- 0 until shuffledOffers.size) {//顺序遍历当前存在的Executor
      val execId = shuffledOffers(i).executorId //Executor的ID
      val host = shuffledOffers(i).host //Executor的host名字
      if (availableCpus(i) >= CPUS_PER_TASK) { //该Executor可以被分配任
       // 务核心实现，通过调用TaskSetManager来为Executor分配Task
          for (task <- taskSet.resourceOffer(execId, host, maxLocality)) {
          tasks(i) += task
          val tid = task.taskId
          taskIdToTaskSetId(tid) = taskSet.taskSet.id
          taskIdToExecutorId(tid) = execId
          executorsByHost(host) += execId
          availableCpus(i) -= CPUS_PER_TASK
          assert(availableCpus(i) >= 0)
          launchedTask = true
        }
      }
    }
```

```
    } while (launchedTask)
  }

  if (tasks.size > 0) {
      hasLaunchedTask = true
  }
  return tasks
}
```

调用栈 6 就是将上个调用得到的 tasks 发送到 Executor：

```
executorData.executorActor!LaunchTask(newSerializableBuffer(serializedTask))
```

Executor 在接到这个消息后，就会开始执行 Task。这个部分的实现详情请参阅第 6 章。

4.3.3　任务调度具体实现

在 org.apache.spark.scheduler.TaskSchedulerImpl#submitTasks 中的关键调用是：

```
schedulableBuilder.addTaskSetManager(manager, manager.taskSet.properties)
```

调度方式如果是 FIFO，schedulableBuilder 的实现是 org.apache.spark.scheduler.FIFOSchedulableBuilder；调度方式如果是 FAIR，schedulableBuilder 的实现就是 org.apache.spark.scheduler.FairSchedulableBuilder。它们的创建方式如下：

```
rootPool = new Pool("", schedulingMode, 0, 0)
schedulableBuilder = {
    schedulingMode match {
        case SchedulingMode.FIFO => //调度方式是FIFO
          new FIFOSchedulableBuilder(rootPool)
        case SchedulingMode.FAIR => //调度方式是FAIR
            new FairSchedulableBuilder(rootPool, conf)
    }
}
schedulableBuilder.buildPools()
```

其中，org.apache.spark.scheduler.Pool 包含了一组可以调度的实体。对于 FIFO 来说，rootPool 包含了一组 TaskSetManager；而对于 FAIR 来说，rootPool 包含了一组 Pool，这些 Pool 构成了一棵调度树，其中这棵树的叶子节点就是 TaskSetManager。schedulableBuilder.buildPools() 因调度方式而异，如果是 FIFO，它的实现是空的，因为 rootPool 并没有包含 Pool，而是直接包含 TaskSetManager：submitTasks 直接将

TaskSetManager 添加到 rootPool 即可。

1. FIFO 调度

采用 FIFO 任务调度的顺序是如何决定的？首先要保证 Job ID 较小的先被调度，如果是同一个 Job，那么 Stage ID 小的先被调度（同一个 Job，可能多个 Stage 可以并行执行，比如 Stage 划分过程中图 4-4 中的 Stage 1 和 Stage 2）。FIFO 的调度逻辑如图 4-7 所示。

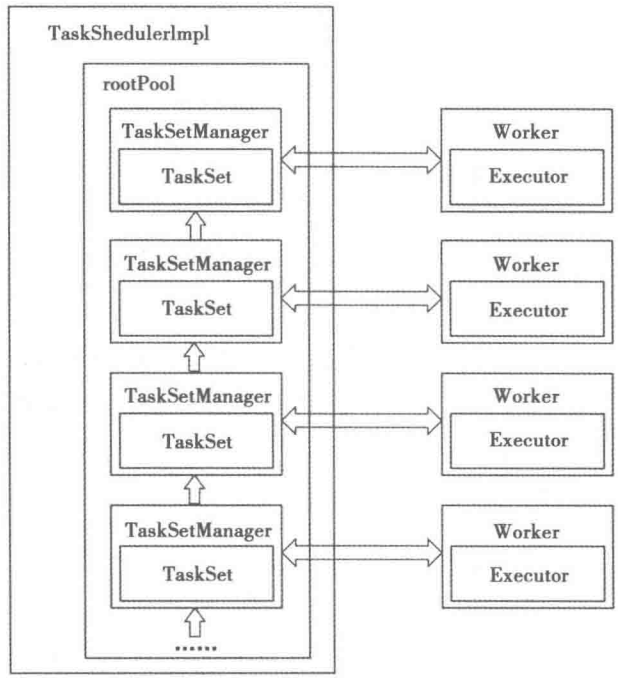


图 4-7 FIFO 调度逻辑图

它的调度算法如下：

```
private[spark] class FIFOSchedulingAlgorithm extends SchedulingAlgorithm {
    override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
        val priority1 = s1.priority // 这个priority实际上是Job ID
        val priority2 = s2.priority //同上
        var res = math.signum(priority1 - priority2) //首先比较Job ID
        if (res == 0) {
            //如果Job ID相同，那么比较Stage ID
```

```
            val stageId1 = s1.stageId
            val stageId2 = s2.stageId
            res = math.signum(stageId1 - stageId2)
        }
        if (res < 0) {
            true
        } else {
            false
        }
    }
}
```

2. FAIR 调度

对于 FAIR，它需要在 rootPool 的基础上根据配置文件来构建这课调度树。一个有效的配置文件如下：

```
<?xml version="1.0"?>
<allocations>
    <pool name="FIFO_Pool">
        <schedulingMode>FIFO</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
</pool>
<pool name="FARI_POOL">
    <schedulingMode>FAIR</schedulingMode>
    <weight>3</weight>
    <minShare>4</minShare>
</pool>
</allocations>
```

核心逻辑实现如下：

```
override def buildPools() {
    var is: Option[InputStream] = None
   try {
       is = Option { // 以spark.scheduler.allocation.file设置文件名字创建
            //  FileInputStream
            schedulerAllocFile.map { f =>
                new FileInputStream(f)
            }.getOrElse {
            // 若spark.scheduler.allocation.file没有设置，那么直接以fairscheduler.
               xml创建FileInputStream
Utils.getSparkClassLoader.getResourceAsStream(DEFAULT_SCHEDULER_FILE)
            }
```

```
        }
        //以is对应的文件内容建立Pool
        is.foreach { i => buildFairSchedulerPool(i) }
    } finally {
        is.foreach(_.close())
    }
    // 创建名字为"default"的Pool
    buildDefaultPool()
}
```

对于 FAIR，首先是挂到 rootPool 下面的 pool 先确定调度顺序，然后在每个 pool 内部使用相同的算法来确定 TaskSetManager 的调度顺序，调度逻辑如图 4-8 所示。算法详细实现如下：

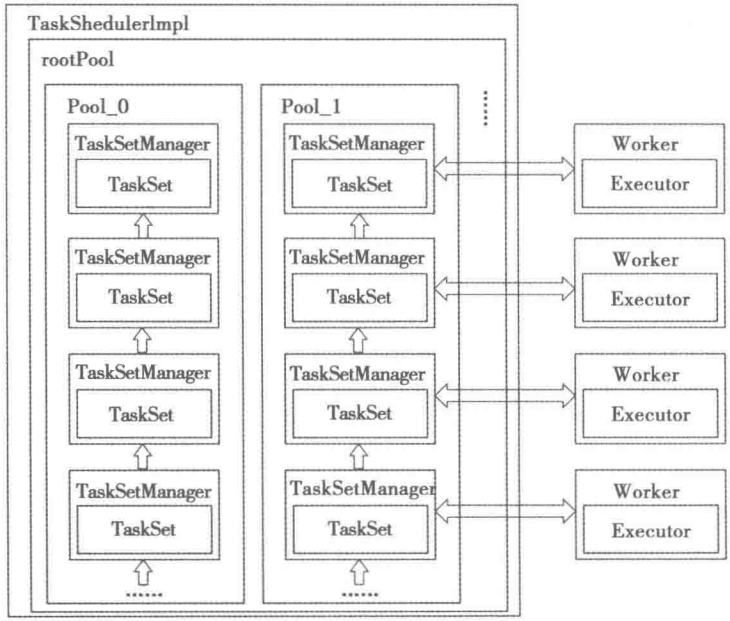


图 4-8 FAIR 调度逻辑图

```
private[spark] class FairSchedulingAlgorithm extends SchedulingAlgorithm {
    override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
        val minShare1 = s1.minShare
        val minShare2 = s2.minShare
        val runningTasks1 = s1.runningTasks
        val runningTasks2 = s2.runningTasks
```

```
            val s1Needy = runningTasks1 < minShare1
            val s2Needy = runningTasks2 < minShare2
            val minShareRatio1 = runningTasks1.toDouble / math.max(minShare1, 1.0).
                toDouble
            val minShareRatio2 = runningTasks2.toDouble / math.max(minShare2, 1.0).
                toDouble
            val taskToWeightRatio1 = runningTasks1.toDouble / s1.weight.toDouble
            val taskToWeightRatio2 = runningTasks2.toDouble / s2.weight.toDouble
            var compare:Int = 0

            if (s1Needy && !s2Needy) {
                return true
            } else if (!s1Needy && s2Needy) {
                return false
            } else if (s1Needy && s2Needy) {
                compare = minShareRatio1.compareTo(minShareRatio2)
            } else {
                compare = taskToWeightRatio1.compareTo(taskToWeightRatio2)
            }
            if (compare < 0) {
                true
            } else if (compare > 0) {
                false
            } else {
                s1.name < s2.name
            }
        }
    }
```

确定了调度顺序，TaskScheduler 就可以按照这个顺序将 Task 发送到 Executor 进行执行了。

4.3.4　Task 运算结果的处理

1. Driver 收到 Executor 的任务执行结果

Task 在 Executor 执行完成时，会通过向 Driver 发送 StatusUpdate 的消息来通知 Driver 任务的状态更新为 TaskState.FINISHED。

Driver 首先会将任务的状态更新通知 TaskScheduler，然后会在这个 Executor

上重新分配新的计算任务。TaskScheduler 的实现在 org.apache.spark.scheduler.TaskSchedulerImpl#statusUpdate：

```
taskIdToTaskSetId.get(tid) match {
    case Some(taskSetId) =>
    if (TaskState.isFinished(state)) {
       //如果Task的状态是FINISHED或者FAILED或者KILLED或者LOST
       //该Task就认为执行结束，那么清理本地的数据结构
           taskIdToTaskSetId.remove(tid)
           taskIdToExecutorId.remove(tid)
       }
       activeTaskSets.get(taskSetId).foreach { taskSet =>
       if (state == TaskState.FINISHED) { //任务成功完成
           //TaskSetManager标记该任务已经结束，注意这里不一定是成功结束的
           taskSet.removeRunningTask(tid)
           //处理任务的计算结果
           taskResultGetter.enqueueSuccessfulTask(taskSet, tid, serializedData)
       } else if (Set(TaskState.FAILED,
           TaskState.KILLED, TaskState.LOST).contains(state)) {
           //TaskSetManager标记该任务已经结束，注意这里不一定是成功结束的
           taskSet.removeRunningTask(tid)
           //处理任务失败的情况
           taskResultGetter.enqueueFailedTask(taskSet, tid, state, serializedData)
       }
    }
}
```

一个 Task 的状态只有是 TaskState.FINISHED 才标记它成功执行；其余的状态包括 TaskState.FAILED、TaskState.KILLED 和 TaskState.LOST 都是执行失败。

其中，处理每次结果都是由一个 Daemon 线程池负责，默认这个线程池由 4 个线程组成，可以通过 spark.resultGetter.threads 设置。

Executor 在将结果回传到 Driver 时，会根据结果的大小使用不同的策略：

1）如果结果大于 1GB，那么直接丢弃这个结果。这个是 Spark1.2 中新加的策略。可以通过 spark.driver.maxResultSize 来进行设置。

2）对于“较大”的结果，将其以 tid 为 key 存入 org.apache.spark.storage.BlockManager；如果结果不大，则直接回传给 Driver。那么如何判定这个阈值呢？

这里的回传是直接通过 AKKA 的消息传递机制。因此这个大小首先不能超过这

个机制设置的消息的最大值。这个最大值是通过 spark.akka.frameSize 设置的，单位是 MByte，默认值是 10MB。除此之外，还有 200KB 的预留空间。因此这个阈值就是 conf.getInt(""spark.akka.frameSize",10) * 1024 *1024 – 200*1024。

3）其他的直接通过 AKKA 回传到 Driver。

2. 处理任务成功执行的机制

理解了 Executor 回传的结果，就能理解 Driver 端对结果的处理了，处理成功完成 Task 的具体实现如下：

```
def enqueueSuccessfulTask(
    taskSetManager: TaskSetManager, tid: Long, serializedData: ByteBuffer) {
    //通过线程池来执行结果获取
    getTaskResultExecutor.execute(new Runnable {
        override def run(): Unit = Utils.logUncaughtExceptions {
            try {
                val (result, size) = serializer.get().deserialize[TaskResult[_]]
                   (serializedData) match {
                case directResult: DirectTaskResult[_] => //结果是计算结果
                    //确定大小符合要求
                if (!taskSetManager.canFetchMoreResults(serializedData.limit())) {
                   return
                }
                (directResult, serializedData.limit())
             case IndirectTaskResult(blockId, size) => //需要向远程的Worker网
                //络获取结果
                //确定大小符合要求
                if (!taskSetManager.canFetchMoreResults(size)) {
                // 从远程的Worker删除该结果
                sparkEnv.blockManager.master.removeBlock(blockId)
                return
             }
              logDebug("Fetching indirect task result for TID %s".format(tid))
              scheduler.handleTaskGettingResult(taskSetManager, tid)
              //从远程的BlockManager获取计算结果
              valserializedTaskResult= sparkEnv.blockManager.getRemoteBytes(blockId)
              if (!serializedTaskResult.isDefined) {
                //如果在Executor的任务执行完成和Driver端取结果之间，Executor所在机
                //器出现故障或者其他错误，会导致获取结果失败
                 scheduler.handleFailedTask(
                     taskSetManager, tid, TaskState.FINISHED, TaskResultLost)
                 return
```

```
                }
            //反序列化结果
        val deserializedResult=serializer.get().deserialize[DirectTaskResult[_]](
                serializedTaskResult.get)
                    //将远程的结果删除
                    sparkEnv.blockManager.master.removeBlock(blockId)
                    (deserializedResult, size)
                }

                result.metrics.resultSize = size
                scheduler.handleSuccessfulTask(taskSetManager, tid, result)
            } catch {
            }
        }
    })
}
```

scheduler.handleSuccessfulTask(taskSetManager, tid, result) 负责处理获取到的计算结果，这个处理过程的调用栈如下：

```
1) org.apache.spark.scheduler.TaskSchedulerImpl#handleSuccessfulTask
2) org.apache.spark.scheduler.TaskSetManager#handleSuccessfulTask
   如果TaskSetManager的所有Task都已经成功完成，那么从rootPool中删除它。
3) org.apache.spark.scheduler.DAGScheduler#taskEnded
4) org.apache.spark.scheduler.DAGScheduler#eventProcessActor
5) org.apache.spark.scheduler.DAGScheduler#handleTaskCompletion
```

核心的处理在调用栈 5，它完成了计算结果的处理。对于 ResultTask，通过调用 org.apache.spark.scheduler.JobWaiter 来告知调用者任务已经结束，核心逻辑如下：

```
task match {
    case rt: ResultTask[_, _] =>
        stage.resultOfJob match {
            case Some(job) =>
                if (!job.finished(rt.outputId)) {
                    job.finished(rt.outputId) = true
                    job.numFinished += 1
                    // If the whole job has finished, remove it
                if (job.numFinished == job.numPartitions) {
                    //所有Task都已经结束，将该Stage标记为结束
                    markStageAsFinished(stage)
                    cleanupStateForJobAndIndependentStages(job)
                        listenerBus.post(SparkListenerJobEnd(job.jobId,
```

```
                    JobSucceeded))
            }

            //对于ResultTask，会运行用户自定义的结果处理函数，因此可能会抛出异常
            try {
                job.listener.taskSucceeded(rt.outputId, event.result)
            } catch {
                case e: Exception =>
                    //直接标记为失败
                    job.listener.jobFailed(new SparkDriverExecutionException(e))
            }
        }
    case None =>
      //应该有任务的推测执行，因此一个Task可能会运行多次
        logInfo("Ignoring result from " + rt + " because its job has finished")
}
```

job.listener.taskSucceeded(rt.outputId, event.result) 实现了对结果的进一步处理，这个处理逻辑是由用户自定义的，然后如果所有的 Task 都返回了，那么需要通知 JobWaiter 任务已经能够结束：

```
override def taskSucceeded(index: Int, result: Any): Unit = synchronized {
    //调用用户逻辑处理结果
    resultHandler(index, result.asInstanceOf[T])
    finishedTasks += 1
    if (finishedTasks == totalTasks) { 该Job结束了
        _jobFinished = true
        jobResult = JobSucceeded
        //会通知org.apache.spark.scheduler.JobWaiter#awaitResult任务结束
        this.notifyAll()
    }
}
```

对于 ShuffleMapTask，首先需要将结果保存到 Stage：

```
val status = event.result.asInstanceOf[MapStatus]
stage.addOutputLoc(smt.partitionId, status)
```

如果该 Stage 的所有 Task 都结束了，那么需要将整体结果注册到 org.apache.spark.MapOutputTrackerMaster；这样下一个 Stage 的 Task 就可以通过它来获取 Shuffle 结果的元数据信息，进而从 Shuffle 数据所在的节点获取数据了。

```
mapOutputTracker.registerMapOutputs(
    stage.shuffleDep.get.shuffleId,
    stage.outputLocs.map(list => if (list.isEmpty) null else list.head).toArray,
    changeEpoch = true)
```

如果在所有任务都返回的情况下，Stage 还有部分数据是空的，则说明部分 Task 执行失败，需要重新提交这个 Stage，重新计算丢失的部分；如果该 Stage 成功完成，那么就将可以提交的 Stage 都提交了（即 Stage 没有 parent Stage 或者 parent Stage 都已经计算完成了）：

```
if (stage.outputLocs.exists(_ == Nil)) {
    // 部分Task失败，需要重新提交
    submitStage(stage)
} else {
    val newlyRunnable = new ArrayBuffer[Stage]
    for (stage <- waitingStages if getMissingParentStages(stage) == Nil) {
        newlyRunnable += stage //可以提交的Stage
    }
    waitingStages --= newlyRunnable
    runningStages ++= newlyRunnable
    for {
        stage <- newlyRunnable.sortBy(_.id)
        jobId <- activeJobForStage(stage)
    } {
    //提交Stage的Task
    submitMissingTasks(stage, jobId)
    }
}
```

至此，一个 Stage 就执行完成了。如果 Stage 是最后一个 Stage，即对应 ResultTask 的 Stage，那么说明该 Job 执行完成了。

3. 处理任务执行失败的容错机制

对于失败的 Task，org.apache.spark.scheduler.TaskSchedulerImpl#statusUpdate 会调用 org.apache.spark.scheduler.TaskResultGetter#enqueueFailedTask 来处理。这个和处理成功的 Task 一样共用一个线程池来执行处理逻辑。相对处理成功的，这个逻辑要简单的多：

```
def enqueueFailedTask(taskSetManager: TaskSetManager, tid: Long, taskState:
    TaskState,
```

```
        serializedData: ByteBuffer) {
        var reason : TaskEndReason = UnknownReason
        getTaskResultExecutor.execute(new Runnable {
            override def run(): Unit = Utils.logUncaughtExceptions {
                try {
                    if (serializedData != null && serializedData.limit() > 0) {
                        reason = serializer.get().deserialize[TaskEndReason](
                        serializedData, Utils.getSparkClassLoader)
                    }
                    } catch {
                       //处理异常
                    }
                    scheduler.handleFailedTask(taskSetManager, tid, taskState, reason)
                }
            })
    }
```

核心实现都在 scheduler.handleFailedTask(taskSetManager, tid, taskState, reason)，它首先会调用 TaskSetManager 来处理任务失败的情况，如果任务的失败次数没有超过阈值（阈值可以通过 spark.task.maxFailures 设置，默认值是 4），那么会重新提交任务。TaskSetManager 逻辑如下：

```
taskSetManager.handleFailedTask(tid, taskState, reason)
   if (!taskSetManager.isZombie && taskState != TaskState.KILLED) {
       // TaskSetManager已经将失败的任务放入待运行的队列等待下一次的调度因此这里开始新
       // 一轮的调度，以便失败的Task有机会获得资源
       backend.reviveOffers()
}
```

TaskSetManager 首先会根据失败的原因来采取不同的动作，比如如果是因为 Task 的结果发序列化失败，那么说明任务的执行是有问题的，这种任务即使重试也不会成功，因此这个 TaskSetManager 会直接失败。如果这些任务需要重试，那么它会重新将这些任务标记为等待调度。核心的实现是：

```
//调用DAGScheduler级别的容错
sched.dagScheduler.taskEnded(tasks(index), reason, null, null, info, taskMetrics)
//标记为等待调度
addPendingTask(index)
if (!isZombie && state != TaskState.KILLED) {
    numFailures(index) += 1
```

```
        //如果失败次数已经超过阈值，那么标记该TaskSetManager为失败
        //阈值可以通过spark.task.maxFailures设置，默认值是4
        if (numFailures(index) >= maxTaskFailures) {
            abort("Task %d in stage %s failed %d times, most recent failure: %s\n
               Driver stacktrace:"
               .format(index, taskSet.id, maxTaskFailures, failureReason))
            return
        }
    }
```

其中，DAGScheduler 级别的容错处理也在 org.apache.spark.scheduler.DAGScheduler#handleTaskCompletion 中。感兴趣的读者可以自己研究其实现。

4.4 Word Count 调度计算过程详解

在 3.3.3 节中提到了“Word Count”中 RDD 转换的逻辑视图，但是由于当时没有提到 Stage 划分和任务调度，因此对这个逻辑视图是如何分阶段转化为不同的任务没有涉及。而前面几个小节将用户提交的 Stage 的划分，任务的顺序提交和在 Executor 端的执行，最终计算结果的处理和任务执行异常时的容错机制都做了详细解析。因此，图 4-9 中在图 3-7 的基础上，增加了 RDD 是如何通过依赖关系查找所依赖的 Parent RDD 的 Partition 的，读者可以对照图 4-9 和 4.2.3.3 节的详解，以便理解这个逻辑图背后的实现原理。

对于前面提到的“Word Count”（图 3-8），会生成两类 Task，一个是 Shuffle 之前的 Task，即 ShuffleMapTask，由于初始的 RDD 有 5 个 Paritition，因此会有 5 个 ShuffleMapTask 生成，如图 4-10 所示；Shuffle 之后的 Task 是需要输出结果的 Task，因此它是 ResultTask，由于设定了 Shuffle 的输出分片数量是 3 个，因此这里会生成 3 个 ResultTask，如图 4-11 所示。

生成的 Task 会被发送到已经启动的 Executor 上，由 Executor 来完成计算任务的执行，执行过程的实现在 org.apache.spark.executor.Executor.TaskRunner#run。后面在第 6 章中会介绍这一部分的实现原理和设计思想。

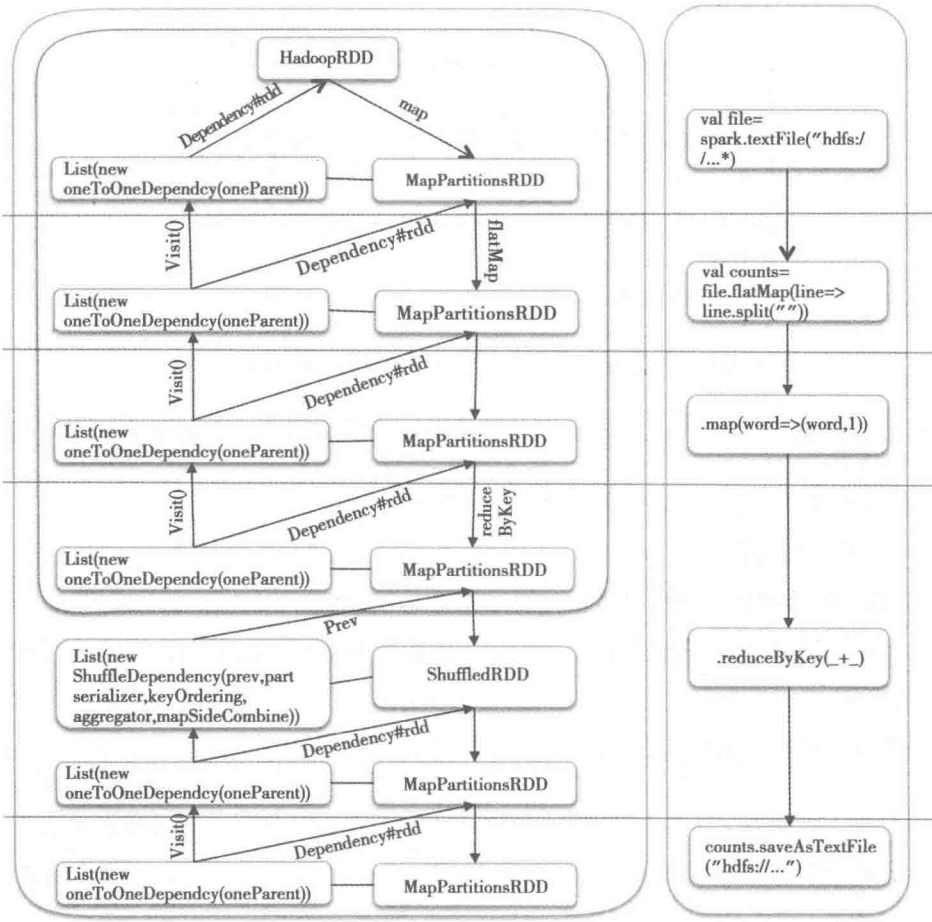


图 4-9　“Word Count”的 Stage 划分和运算逻辑图

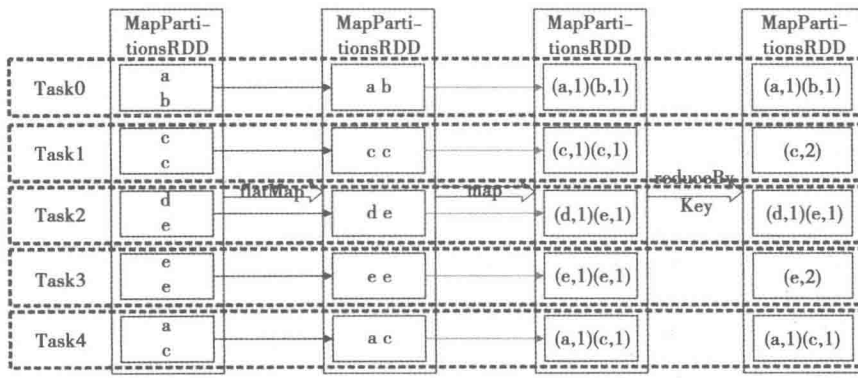


图 4-10　“Word Count”中的 ShuffleMapTask 的逻辑运行图

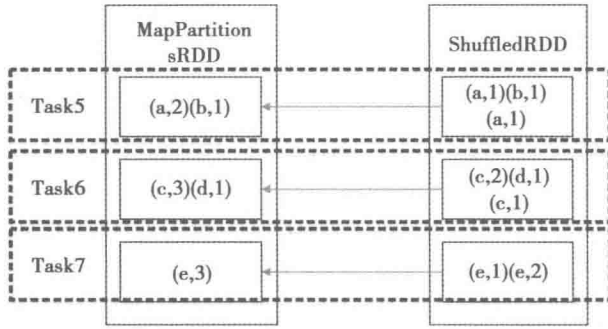


图 4-11 “Word Count”中的 ResultTask 的逻辑运行图

但是，Spark 对于这个过程的实现却不像看上去那么简单，它要解决以下问题：

1）如何将用户逻辑转换成可以并发执行的任务？

2）如何定义和实现 Shuffle？

3）如何传递数据，如何将数据传递到正确的任务上？

4）如何尽量避免数据在集群上的移动（即数据本地性，Data Locality），任务最好在数据所在的节点上执行？

5）如何为用户分配计算资源？

对于第一个问题，本章已经给出了答案；对于第二个和第三个问题，由第 7 章回答；对于第四个问题，是在任务调度的时候实现的；最后一个问题，为用户分配计算资源，对于 Standalone 的方式来说，实际上在创建 SparkContext 的时候，Master 就会开始为用户提交的计算分配计算资源。在 DAGScheduler 将计算任务划分完成后，TaskScheduler 就会在已经分配好的计算资源上启动计算任务。

4.5 小结

要理解什么是 Stage，首先要搞明白什么是 Task。Task 是在集群上运行的基本单位。一个 Task 负责处理 RDD 的一个 Partition。RDD 的多个 Patition 会分别由不同的 Task 去处理。当然，这些 Task 的处理逻辑是完全一致的。这一组 Task 就组成了一个 Stage。有两种 Task：

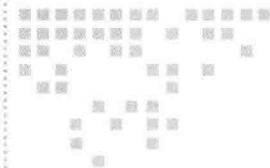
```
1）org.apache.spark.scheduler.ShuffleMapTask
2）org.apache.spark.scheduler.ResultTask
```

ShuffleMapTask 根据 Task 的 partitioner 将计算结果放到不同的 bucket 中。而 ResultTask 将计算结果发送回 Driver Application。一个 Job 包含了多个 Stage，而 Stage 是由一组完全相同的 Task 组成的。最后的 Stage 包含了一组 ResultTask。

在用户触发了一个动作后，比如 count、collect，SparkContext 会通过 runJob 的函数开始进行任务提交。最后会通过 DAG 的事件处理器传递到 DAGScheduler 本身的 handleJobSubmitted，它首先会划分 Stage，提交 Stage，然后提交 Task。至此，Task 就开始在集群上运行了。

一个 Stage 的开始就是从外部存储或者 shuffle 结果中读取数据；一个 Stage 的结束就是由于发生 shuffle 或者生成结果时。

在 DAGScheduler 将用户提交的应用划分为不同的 Stage 后，TaskScheduler 模块负责为 Stage 的 Task 分配计算资源，这个计算资源的分配实际上是 Cluster Manager 职责。接下来的章节将以 Standalone 模式为例，分析 Spark 是如何为应用分配计算的。



Chapter 5 第 5 章

Deploy 模块详解

Spark 的 Cluster Manager 有以下几种部署模式：Standalone，Mesos，YARN，EC2，Local。

Spark 的 Cluster Manager 最开始的时候仅支持 Mesos，后来为了方便 Hadoop 的用户加入了对 YARN 的支持，即使用 YARN 作为资源的管理器和应用内的任务调度器。而 Standalone 模式的出现，更好地扩展了 Spark 的应用场景。本章将着重讲解 Standalone 部署方式的详细实现，即 Deploy 模块的详细实现。

5.1 Spark 运行模式概述

在 4.3.1 节介绍 TaskScheduler 的创建时，提到过不同的运行模式实际上实现了不同的 SchedulerBackend 和 TaskScheduler：

1）org.apache.spark.scheduler.SchedulerBackend

2）org.apache.spark.scheduler.TaskScheduler

在 SparkContext 的创建过程中，会通过传入的 Master URL 的值来确定不同的运行模式，并且创建不同的 SchedulerBackend 和 TaskScheduler，这个创建过程的实现在

org.apache.spark.SparkContext#createTaskScheduler。

5.1.1　local

Master URL 如果使用以下方式，那么就是以本地方式启动 Spark：

1）local

使用一个工作线程来运行计算任务，不会重新计算失败的计算任务。

2）local[N]/local[*]

对于 local[N]，使用 N 个工作线程；对于 local[*]，工作线程的数量取决于本机的 CPU Core 的数目，保证逻辑上一个工作线程可以使用一个 CPU Core。和 local 一样，不会重新计算失败的计算任务。

3）local[threads, maxFailures]

threads 设置了工作线程的数目；maxFailures 则设置了计算任务最大的失败重试次数。

4）local-cluster[numSlaves, coresPerSlave, memoryPerSlave]

伪分布式模式，本机会运行 Master 和 Worker。其中 numSlaves 设置了 Worker 的数目；coresPerSlave 设置了 Worker 所能使用的 CPU Core 的数目；memoryPerSlave 设置了每个 Worker 所能使用的内存数。

对于前 3 种方式，内部实现是相同的，区别就是启动的工作线程数和计算失败时重试的次数不一样。对于 local 模式来说，SchedulerBackend 的实现是 org.apache.spark.scheduler.local.LocalBackend。TaskScheduler 的实现是 org.apache.spark.scheduler.TaskSchedulerImpl。local 模式的系统架构图如图 5-1 所示。有一点要注意的是，LocalBackend 也持有一个 Actor：org.apache.spark.scheduler.local.LocalActor，并通过这个 Actor 和 Executor 进行通信。这样 Executor 的实现就可以被各个运行模式复用。

对于第 4 种伪分布式模式，实际上是在本地机器模拟了一个分布式环境，除了

Master 和 Worker 都运行在本机外，与 Standalone 模式并无区别。整体架构如图 5-2 所示。SchedulerBackend 的实现是 org.apache.spark.scheduler.cluster.SparkDeploy- SchedulerBackend。TaskScheduler 的实现也是 org.apache.spark.scheduler.Task-SchedulerImpl。需要强调的是，在 Worker 接到 Master 的 LaunchExecutor 消息后，会创建一个实例 org.apache.spark.deploy.worker.ExecutorRunner，由 Executor-Runner 来启动一个新的进程。这个进程的实现是 org.apache.spark.executor.Coarse-GrainedExecutorBackend。因此在图 5-2 中使用了虚线框来表示这是一个进程，而不是在 Worker 进程内。还有一点要强调的是，local-cluster 模式可以启动多个 Worker，这一点并没有在图中标记出来。

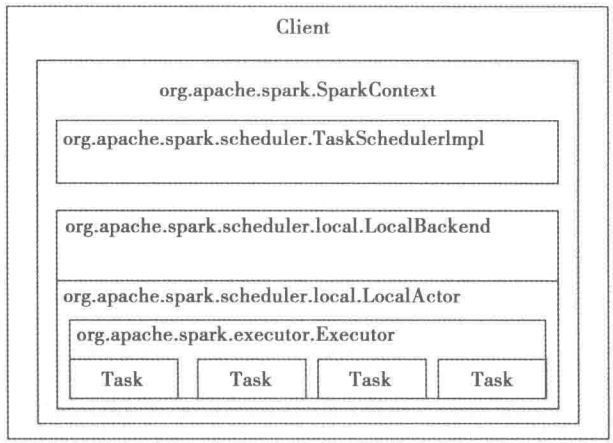


图 5-1 local 模式

5.1.2 Mesos

Apache Mesos（http://mesos.apache.org）采用了 Master/Slave 的架构，主要由四部分组成：Mesos Master、Mesos Slave、Framework（也称为 Mesos Application）和 Executor。并且 Mesos 通过 ZooKeeper 实现了 Master 的高可用性。整体架构如图 5-3 所示。

Mesos Master 是整个系统的核心。它通过 frameworks_manager 管理接入的各个 framework，通过 slaves_manager 管理所有的 Slave，并将 Slave 上的资源按照某种分配策略分配给 framework。

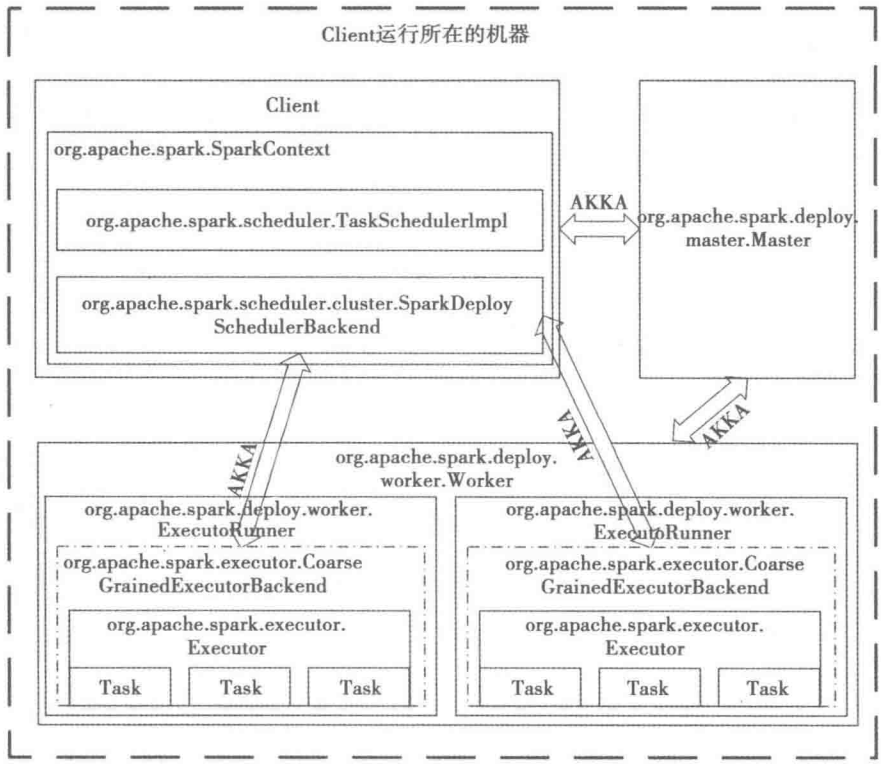


图 5-2　local Cluster 模式

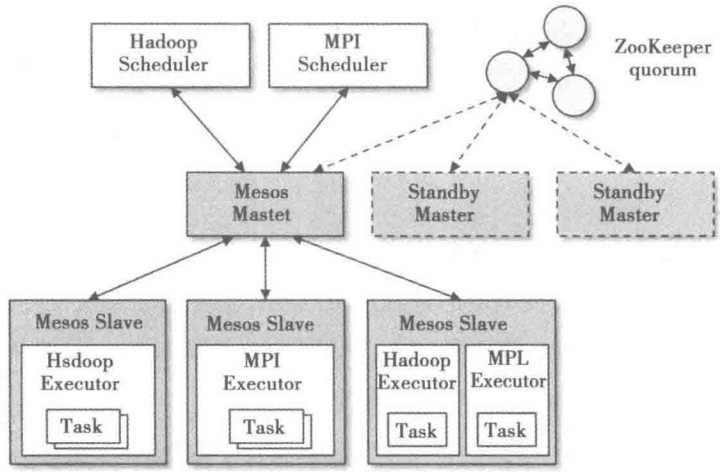


图 5-3　Mesos 整体架构

Mesos Slave 需要将自己的资源情况汇报给 Master（当前主要的资源主要有 CPU 和 Memory 两种），负责接收并执行来自 Mesos Master 的命令，并且为运行在本节点的 Task 分配资源并且管理这些 Task。当前 Mesos 也支持很多容器技术，可以很方便地做到各个计算任务的资源隔离。

Framework 指的是外部的计算框架，这些计算框架通过注册接入 Mesos 后，由 Master 进行统一管理和资源分配。要接入的外部框架必须实现一个调度模块，该调度模块负责框架内部的任务调度，即外部框架在从 Mesos 获得了资源后，需要由这个调度模块分配给本框架中的计算任务。换句话说，Mesos 系统采用了双层调度框架：

1）由 Mesos 将资源分配给外部框架。

2）外部框架的调度模块将资源分配给外部框架的内部计算任务。

Executor 主要用于启动外部框架内部的计算任务。由于不同的框架中启动任务的接口或者方式都不相同，因此 Mesos 规定外部框架需要实现自己的 Executor。这样 Mesos 就知道如何去启动不同框架下的计算任务。

Mesos 的资源调度可以分为两类，即粗粒度调度和细粒度调度。

粗粒度的调度方式是每个 Executor 获得资源后就长期持有，直到应用程序退出才会释放资源。这种调度方式的优点就是减少了资源调度的时间开销，缺点是由于其分配的资源被长期占有，在应用程序大部分的计算任务都已经完成的情况下，会造成资源的浪费。尤其是有些计算任务出现长尾时，这个资源浪费的情况可能会变得不可接受。

而细粒度的资源调度是根据任务的实际需要动态申请的，任务完成后就会将资源还给系统，所以避免了粗粒度调度的资源浪费问题。但是由于每次任务的调度都要从系统动态申请资源，调度的时间开销较大。特别是对于那些运行时间很短但是计算任务数量又很多的应用程序来说，性能会受到较大影响。

Spark 可以通过选项 spark.mesos.coarse 来设置是采用粗粒度的调度模式还是细粒度的调度模式。

1. 粗粒度资源调度

Client 在启动时会通过 org.apache.spark.scheduler.cluster.mesos.CoarseMesosSchedulerBackend 向 Mesos Master 获取资源。CoarseMesosSchedulerBackend 与 Standalone 模式一样，也是继承了 org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend，它还额外地实现了 Mesos 的资源调度接口，用于向 Mesos 资源调度框架注册和接受来自 Mesos 的资源分配，此外，它在得到分配的资源后会通过 Mesos 框架远程启动 org.apache.spark.executor.CoarseGrainedExecutorBackend。Executor 的实现也和其他的模式保持一致，都是 org.apache.spark.executor.Executor。系统架构如图 5-4 所示。

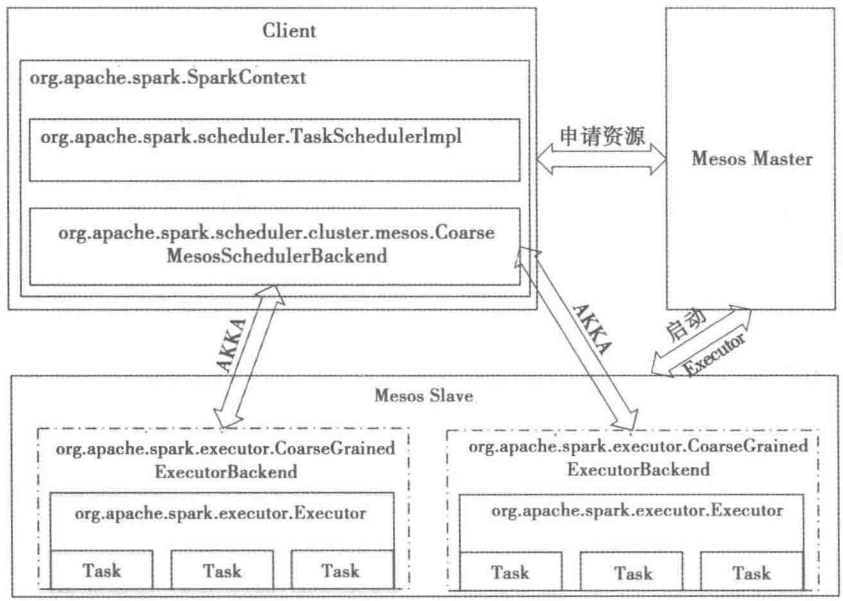


图 5-4　Mesos 粗粒度模式

粗粒度模式是在 Client 启动时向 Mesos Master 申请全部的资源，并且在所有的计算任务结束后，再向 Mesos 归还资源。这一点并没有在图中标记出来。

2. 细粒度调度模式

细粒度的调度模式的 SchedulerBackend 的实现是 org.apache.spark.scheduler.cluster.mesos.MesosSchedulerBackend，它和粗粒度模式一样，也实现了 Mesos 的资源调度接口，向 Mesos 资源调度框架注册和接受来自 Mesos 的资源分配。区别就是它会

按照任务来向 Mesos Master 申请计算资源，并且在任务结束时归还资源。系统架构如图 5-5 所示。

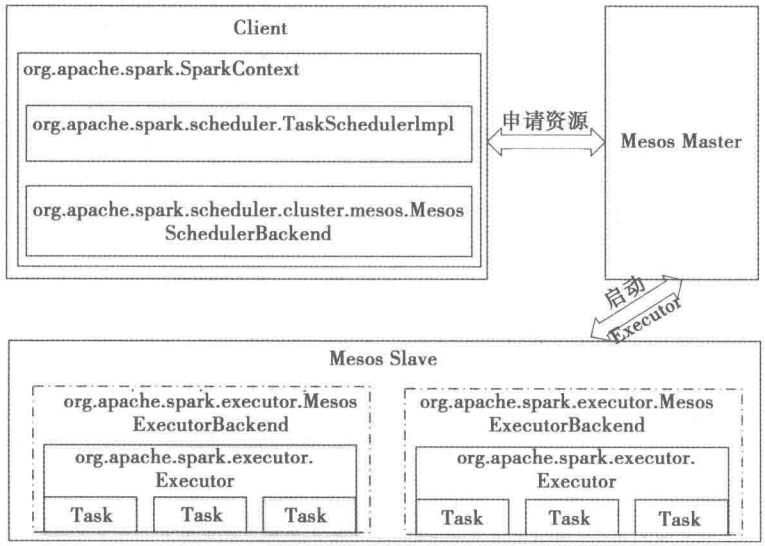


图 5-5 Mesos 细粒度模式

5.1.3 YARN

与第一版 Hadoop 中经典的 MapReduce 引擎相比，YARN 在可伸缩性、效率和灵活性上提供了明显的优势。YARN 的整体架构如图 5-6 所示。

ResourceManager（即资源管理器）全局管理所有应用程序计算资源的分配。它和每一台机器的 NodeManager（即节点管理服务器）能够管理应用在那台机器上的进程并能对计算进行组织。

每一个应用的 ApplicationMaster 则负责相应的调度和协调。ApplicationMaster 是一个详细的框架库，它结合从 ResourceManager 获得的资源和 NodeManager 协同工作来运行和监控任务。每一个应用的 ApplicationMaster 的职责有：向调度器索要适当的资源容器，运行任务，跟踪应用程序的状态和监控它们的进程，处理任务的失败原因。

ResourceManager 支持分层级的应用队列，这些队列享有集群一定比例的资源。

从某种意义上讲它就是一个纯粹的调度器，它在执行过程中不对应用进行监控和状态跟踪。同样，它也不能重启因应用失败或者硬件错误而运行失败的任务。

ResourceManager 是基于应用程序对资源的需求进行调度的；每一个应用程序需要不同类型的资源因此就需要不同的容器。资源包括：内存、CPU、磁盘、网络等。可以看出，这同现 Mapreduce 固定类型的资源使用模型有显著区别，它给集群的使用带来负面的影响。资源管理器提供一个调度策略的插件，它负责将集群资源分配给多个队列和应用程序。调度插件可以基于现有的能力调度和公平调度模型。

NodeManager 是每一台机器框架的代理，是执行应用程序的容器，监控应用程序的资源使用情况（CPU、内存、硬盘、网络）并且向调度器汇报。

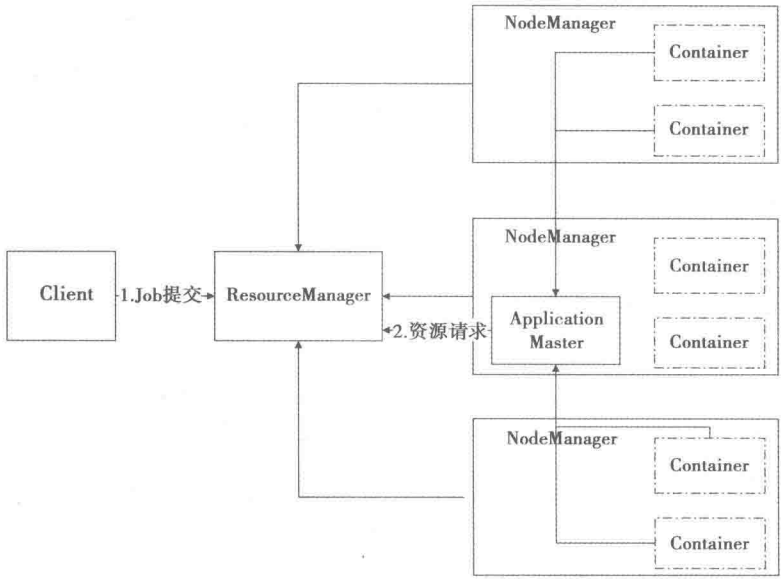


图 5-6　YARN 整体架构

在 Client 提交了 Job 后，Application Master 会向 ResourceManager 请求资源，在获得了资源后，Application Master 会在 NodeManager 上启动 Container，运行计算任务，并且和 Container 保持联系，监控任务的运行状态等。

1. YARN Cluster 模式

YARN Cluster 模式，就是通过 Hadoop YARN 来调度 Spark Application 所需要的

资源。整体架构图如图 5-7 所示。

用户提交的 Application 会通过 YARN Client 提交到 YARN 的主节点 ResourceManager 上，ResourceManager 会在一个工作节点上启动 ApplicationMaster（实现是 org.apache.spark.deploy.yarn.ApplicationMaster）。启动了 ApplicationMaster 后，才算是完成了 Spark Application 的提交。

ApplicationMaster 在将自己注册成为一个 YARN ApplicationMaster 后，才会开始执行用户提交的 Application。YARN Cluster 模式的 TaskScheduler 的实现是 org.apache.spark.scheduler.cluster.YarnClusterScheduler。YarnClusterScheduler 继承自 org.apache.spark.scheduler.TaskSchedulerImpl，额外实现的逻辑是确定 ApplicationMaster 初始化完成。ApplicationMaster 会通过 YARN ResourceManager 和 NodeManager 的接口在集群中启动若干个容器，用于启动 org.apache.spark.executor.CoarseGrainedExecutorBackend，之后它会启动 org.apache.spark.executor.Executor。

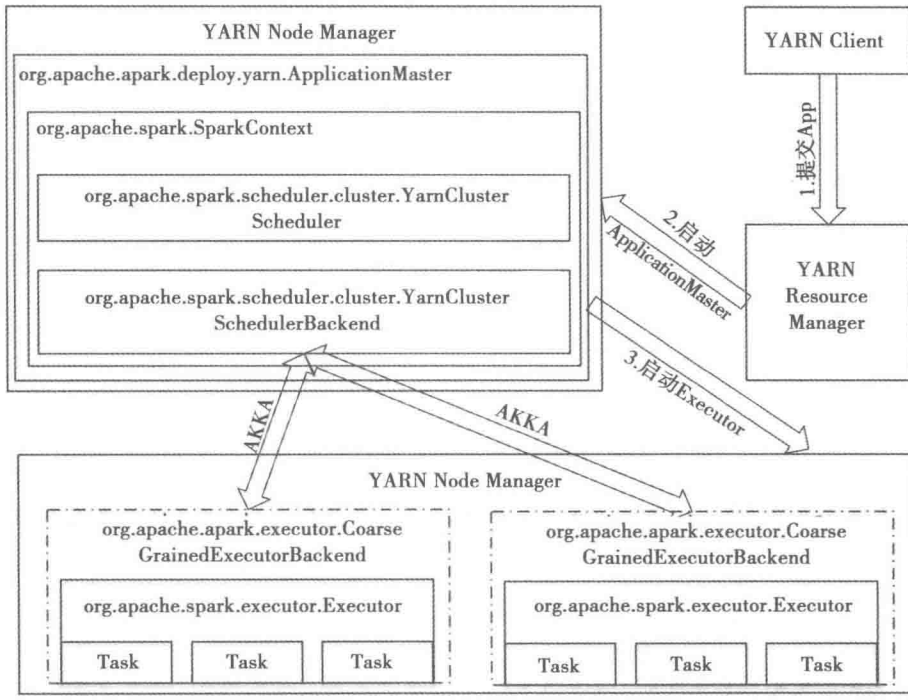


图 5-7　YARN Cluster 模式

2. YARN Client 模式

该模式和 YARN Cluster 模式的区别在于，用户提交的 Application 的 SparkContext 是在本机上运行，适合 Application 本身需要在本地进行交互的场景；而 YARN Cluster 中，所有的计算都是在 YARN 的节点上运行的。系统的整体架构图如图 5-8 所示。

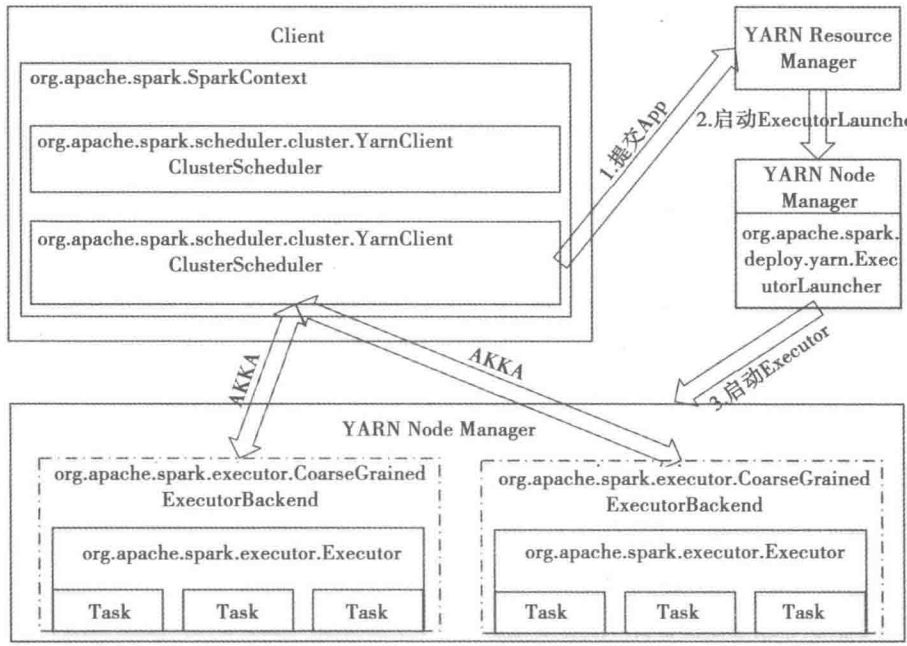


图 5-8　YARN Client 模式

SparkContext 在初始化的过程中会创建 org.apache.spark.scheduler.cluster.YarnClientClusterScheduler 和 org.apache.spark.scheduler.cluster.YarnClientSchedulerBackend。YarnClientSchedulerBackend 继承自 org.apache.spark.scheduler.cluster.YarnSchedulerBackend，而它继承自 org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend，可以和 ExecutorBackend 通过 AKKA 进行通信。

YarnClientSchedulerBackend 会启动一个 org.apache.spark.deploy.yarn.Client，用于向 ResourceManager 提交 Application，并且由 ApplicationMaster 来负责在多个节点上启动 Executor。

5.2 模块整体架构

Deploy 模块采用的也是典型的 Master/Slave 架构，其中 Master 负责整个集群的资源调度和 Application 的管理。Slave（即 Worker）接收 Master 的资源分配调度命令后启动 Executor，由 Executor 完成最终的计算任务。而 Client 则负责 Application 的创建和向 Master 注册 Application，并且负责接收来自 Executor 的状态更新和计算结果等。其整体架构如图 5-9 所示。

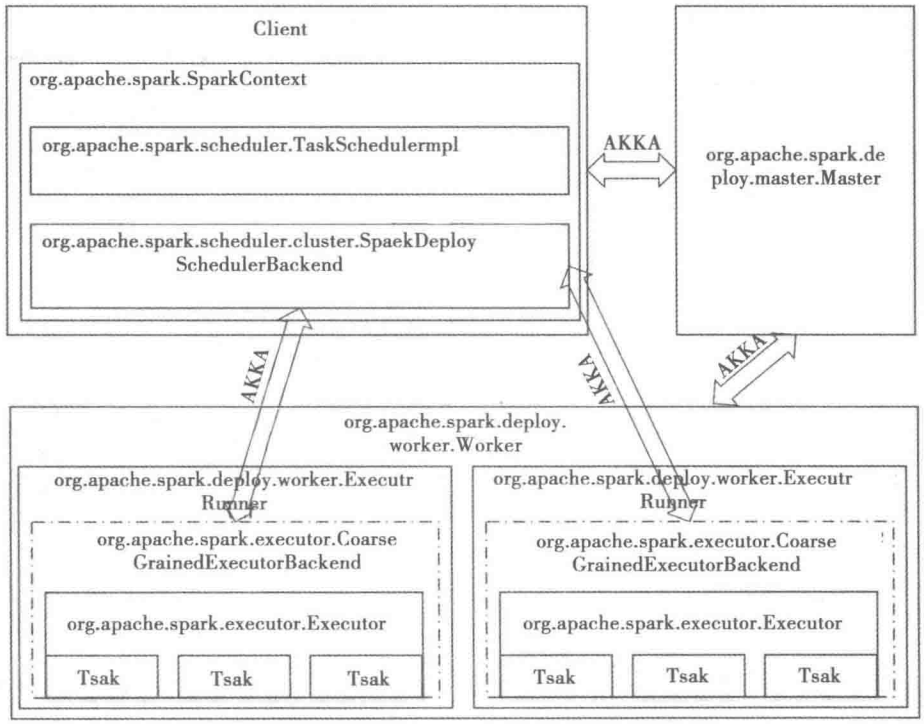


图 5-9 Standalone 模式的整体架构

Deploy 模块主要包含 3 个子模块：Master、Worker、Client，它们之间的通信通过 AKKA 完成。对于 Master 和 Worker，它们本身就是一个 Actor，因此可以直接通过 AKKA 实现通信。Client 虽然本身不是一个 Actor，这三者的主要职责如下：

1）Master：接收 Worker 的注册并管理所有的 Worker，接收 Client 提交的 Application，FIFO 调度等待的 Application 并向 Worker 提交。

2）Worker：向 Master 注册自己，根据 Master 发送的 Application 配置进程环境，并启动 StandaloneExecutorBackend。

3）Client：向 Master 注册并监控 Application。当用户创建 SparkContext 时会实例化 SparkDeploySchedulerBackend，而实例化 SparkDeploySchedulerBackend 的同时会启动 Client，通过向 Client 传递启动参数和 Application 有关信息，Client 向 Master 发送请求注册 Application 并且在计算节点上启动 StandaloneExecutorBackend。

5.3　消息传递机制详解

Deploy 模块之间主要通过 AKKA 通信，因此，通过对各个子模块之间消息通信协议的梳理，可以分析每个子模块的功能职责。

5.3.1　Master 和 Worker

Master 作为整个集群的管理者，需要 Worker 通过注册、汇报状态来维护整个集群的运行状态，并且通过这些状态来决定资源调度策略等。图 5-10 是 Worker、Master 和 Driver 的注册逻辑图。

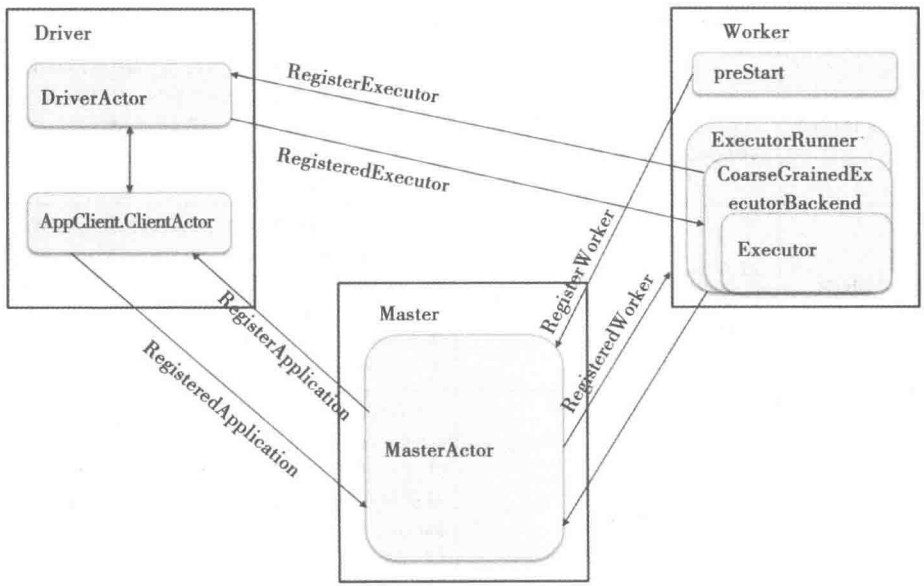


图 5-10　Worker、Master 和 Driver 的注册逻辑图

Worker 向 Master 发送的消息主要包含三类：

1）注册：Worker 启动时需要向 Master 注册，注册时需要汇报自身的信息。

2）状态汇报：汇报 Executor 和 Driver 的运行状态；在 Master 故障恢复时，需要汇报 Worker 上当前运行的 Executor 和 Driver 的信息。

3）报活心跳：Worker 每隔指定周期会向 Master 发送报活的心跳。

Worker 向 Master 发送的消息如表 5-1 所示。

表 5-1 Worker 向 Master 发送的消息

消息名称	参数	作用
RegisterWorker	id: String, host: String, port: Int, cores: Int, memory: Int, webUiPort: Int, publicAddress: String	Worker 启动后，会向 Master 发送消息进行注册。Master 完成注册后会回复 RegisteredWorker。Worker 完成注册后就可以接受来自 Master 的调度请求
ExecutorStateChanged	appId: String, execId: Int, state: ExecutorState, message: Option[String], exitStatus: Option[Int]	当 Executor 的状态有更新时，向 Master 汇报。Worker 接到 LaunchExecutor 的命令后，会向 Master 汇报其启动 Executor 的状态。然后 Executor 在被杀死或者退出时，ExecutorRunner 会向 Worker 汇报该情况而 Worker 又会将这个状态汇报到 Master
DriverStateChanged	driverId: String, state: DriverState, exception: Option[Exception]	在 Driver 的状态变化时，org.apache.spark.deploy.worker.DriverRunner 会向 Worker 发送状态更新的消息，Worker 会将这个消息转到 Master。在 Master 接到停止 Driver 的消息后，也会向自身发送 Driver 状态更新的消息
WorkerSchedulerStateResponse	id: String, executors: List[Executor-Description], driverIds: Seq[String]	在 Master 故障恢复时，会向注册过的 Worker 发送 MasterChanged(masterUrl: String, master-WebUiUrl: String) 的消息，Worker 会通过新的 Master URL 来创建 Master Actor 的 Reference，然后通过向 Master 回复 WorkerSchedulerStateResponse 消息来汇报 Worker 上运行的 Executor 和 Driver 的信息
Heartbeat	workerId: String	Worker 向 Master 的报活心跳，间隔可以通过 spark.worker.timeout 设置，注意这个值的四分之一才是心跳间隔。默认值是 15 秒。Worker 会记录本身资源的使用情况，包括已经使用的 CPU Core 数、memory 数等，但是这个统计只是为了 Web UI 的展现。Master 会记录 Worker 的资源使用情况，无需 Worker 自身汇报。Worker 与 Master 之间的心跳的目的仅仅是为了报活，不会携带其他的信息

Master 向 Worker 发送的消息除了响应 Worker 的注册外，还有一些控制命令，包括让 Worker 重新注册、让 Worker 启动 Executor 或者 Driver、停止 Executor 和 Driver 等。消息列表如表 5-2 所示。

表 5-2　Master 向 Worker 发送的消息

消息名称	参　数	作　用
RegisteredWorker	masterUrl: String, masterWebUiUrl: String	Worker 注册成功后的返回消息，Worker 收到该消息则表明注册成功，根据返回的 masterUrl 构建 Master Actor 的 Reference
RegisterWorkerFailed	message: String	Worker 注册失败的返回消息，Worker 在接到该消息后会直接退出
ReconnectWorker	masterUrl: String	注册过的 Worker 在心跳超时后重新发送心跳，那么 Master 会发送该消息告知该 Worker 需要重新注册。Worker 收到该消息后会开始向 Master 注册
KillExecutor	masterUrl: String, appId: String, execId: Int	在 Application 完成的时候，Master 会告知 Worker 删除指定的 Executor
LaunchExecutor	masterUrl: String, appId: String, execId: Int, appDesc: ApplicationDescription, cores: Int, memory: Int	Master#schedule 会为当前等待分配资源的 Driver 和 Application 分配资源。Master 会根据调度策略为 Application 分配资源并且会向 Worker 发送启动 Executor 的命令
LaunchDriver	driverId: String, driverDesc: DriverDescription	类似于 LaunchExecutor，只不过调度的对象变成了 Driver
KillDriver	driverId: String	通知 Driver 所在的 Worker 杀死该 Driver

5.3.2　Master 和 Client

这里的 Client 分为 Driver Client 和 AppClient。Driver Client 向 Master 发送的消息如表 5-3 所示。

表 5-3 Driver Client 向 Master 发送的消息

消息名称	参数	作用
RequestSubmitDriver	driverDescription: DriverDescription	向 Master 提交 Driver
RequestKillDriver	driverId: String	向 Master 发送 KillDriver 的请求
RequestDriverStatus	driverId: String	获取 Driver 的当前运行状态

而 Master 使用表 5-4 中列举的消息响应 Driver Client 的请求。

表 5-4 Master 响应 Driver Client 请求的消息

消息名称	参数	作用
SubmitDriverResponse	success: Boolean, driverId: Option[String], message: String	Master 向 Driver 返回注册是否成功。如果当前的 Master 是 Active 的，那么注册就会成功
KillDriverResponse	driverId: String, success: Boolean, message: String	Master 在接到 Driver Client 的停止 Driver 的请求后，会判断 Driver 是否已经在运行，如果还在等待调度的阶段，那么直接将它从等待列表中删除；如果已经在运行状态，那么通过消息 KillDriver 通知 Worker 停止该 Driver
DriverStatusResponse	found: Boolean, state: Option[DriverState], workerId: Option[String], workerHostPort: Option[String], exception: Option[Exception]	回复 Driver 查询当前运行状态的请求。其中参数 found 代表该 Driver 是否存在，state 代表了 Driver 的状态信息

AppClient 本身并不是一个 Actor，但是它持有一个 Actor：org.apache.spark.deploy.client.AppClient.ClientActor。Master 与 AppClient 的交互实际上是 Master 与 ClientActor 的交互。消息列表如表 5-5 所示。

表 5-5 AppClient 向 Master 发送的消息

消息名称	参数	作用
RegisterApplication	appDescription:ApplicationDescription	AppClient 向 Master 注册 Application 时发送该消息。其中，参数 appDescription 包含了 Application 的所有信息，比如最多可使用的 CPU core 的数量，每个节点最多可以使用的内存，还有拉起 ExecutorBackend 的方式等

（续）

消息名称	参　数	作　用
MasterChangeAcknowledged	appId: String	Master 在故障恢复后，会通过发送消息 MasterChanged 通知 AppClient 和 Worker。而它们接到消息后，会更改它们保存的 Master 的信息并且通过此消息回复

Master 向 AppClient 发送的消息如表 5-6 所示。

表 5-6　Master 向 AppClient 发送的消息

消息名称	参　数	作　用
RegisteredApplication	appId: String, masterUrl: String	回复 Application 的注册结果
ExecutorAdded	id: Int, workerId: String, hostPort: String, cores: Int,memory: Int	在 Worker 启动了 Executor 后，会通过这个消息告知 AppClient
ExecutorUpdated	id: Int, state: ExecutorState, message: Option[String], exitStatus: Option[Int]	将 Executor 的状态更新通知 AppClient
ApplicationRemoved	message: String	在 Application 无论成功完成或者失败的情况下，都会通过这个消息通知 AppClient；AppClient 会停止它
MasterChanged	masterUrl: String, masterWebUiUrl: String	在 Master 故障恢复后，它会通过该消息通知 AppClient 和 Worker 当前 Maser 已经更改

5.3.3　Client 和 Executor

org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend.DriverActor 负责，而 Executor 实际上对应于 org.apache.spark.executor.CoarseGrainedExecutorBackend。

Driver 向 Executor 发送的消息分为两类：

1）启动 Task，停止 Task。

2）响应 Executor 注册的请求，回复成功还是失败。

Driver 向 Executor 发送的消息的详细说明如表 5-7 所示。

表 5-7 Driver 向 Executor 发送的消息

消息名称	参数	作用
LaunchTask	data: SerializableBuffer	Driver 通知 Executor 启动一个 Task，data 保存了序列化的 Task 的信息，Executor 通过这个信息来启动 Task
KillTask	taskId: Long, executor: String, interruptThread: Boolean	停止 Task
RegisteredExecutor	无	回复 ExecutorBackend 注册功能，ExecutorBackend 接到后会创建 org.apache.spark.executor.Executor
RegisterExecutorFailed	message: String	注册失败，ExecutorBackend 接到该响应后会退出

Executor 向 Driver 发送的消息如表 5-8 所示。

表 5-8 Executor 向 Driver 发送的消息

消息名称	参数	作用
RegisterExecutor	executorId: String, hostPort: String, cores: Int	注册 Executor。SchedulerBackend 在确认 executorId 未被注册后，会回复注册成功的消息，然后开始新一轮的调度
StatusUpdate	executorId: String, taskId: Long, state: TaskState, data: SerializableBuffer	汇报 Executor 中运行的 Task 的状态

5.4 集群的启动

在 Spark 集群部署完成后，启动整个集群时会发生什么？本节就 Master 的启动和 Worker 的启动来分析架构实现和设计思想。

5.4.1 Master 的启动

Master 的实现是 org.apache.spark.deploy.master.Master。它是一个 Actor，因此 Worker 和 AppClient 可以直接和它通过 AKKA 进行通信。一个集群可以部署多个 Master，以达到高可用性的目的，因此它还实现了 org.apache.spark.deploy.master.

LeaderElectable 以在多个 Master 中选举出一个 Leader。LeaderElectable 是一个 trait：

```
@DeveloperApi
trait LeaderElectable {
    def electedLeader()
    def revokedLeadership()
}
```

从定义的接口可以清晰地看出它的作用。用户自定义的选举方式的 Master 也需要实现它。

除了选举机制，还要注意的是 Master 元数据持久化的方式。Master 保存了整个集群的元数据，包含 Worker、Application 和 Driver Client。Spark 的 Standalone 模式支持以下几种方式的元数据持久化方式和选举机制：

1. ZooKeeper

实现了基于 ZooKeeper 的选举机制，元数据信息会持久化到 ZooKeeper 中。因此在 Master 故障后，ZooKeeper 会在备份的 Master 中选举出新的 Master，新的 Master 在启动后会从 ZooKeeper 中获取元数据信息并且恢复这些数据。

2. FILESYSTEM

集群的元数据信息会保存在本地文件系统。而 Master 启动后则会立即成为 Active 的 Master。如果不考虑机器本身的故障和在设置了 Master 进程退出后能自动重启的前提下，这种方式也是可以接受的。

3. CUSTOM

这个是用户自定义的。如果需要自定义的机制，那么需要实现 org.apache.spark.deploy.master.StandaloneRecoveryModeFactory，并且将实现的类的名字配置到 spark.deploy.recoveryMode.factory。主要实现的是两个接口：

1）def createPersistenceEngine(): PersistenceEngine　该接口实现了持久化数据和恢复数据，这些数据包括 Worker、Application 和 Driver Client。

2）def createLeaderElectionAgent(master: LeaderElectable): LeaderElectionAgent 该接口实现了选举机制，即从几个 Standby 的 Master 中选举出一个 Master 作为集群的

管理者。

4. NONE

不会持久化集群的元数据，Master 在启动后会立即接管集群的管理工作。

可以通过 spark.deploy.recoveryMode 进行设置。如果不设置，那么就默认是第 4 种方式 NONE，即没有备份的 Master，集群所有历史的元数据信息在 Master 重启后都会丢失。

在 Master 的进程启动后，ZooKeeper 方式的选举机制会根据自身策略来选举出 Leader；对于 FILESYSTEM 和 NONE 方式，进程启动后会立即成为 Leader。这通过调用 org.apache.spark.deploy.master.Master#electedLeader 来实现：

```
override def electedLeader() {
   self ! ElectedLeader
}
```

而被选举为 Leader 的 Master，会首先读取集群的元数据信息，如果有读到的数据，那么 Master 的状态就会变为 RecoveryState.RECOVERING，然后开始恢复数据和通知 Worker、AppClient 和 Driver Client，Master 已经更改，恢复结束后 Master 的状态会变成 RecoveryState.ALIVE。对于没有读取任何数据的 Master，状态会立即变成 RecoveryState.ALIVE。Master 只有在状态是 RecoveryState.ALIVE 时才可以对外服务，包括接受 Worker、Application 和 Driver Client 的注册和状态更新等。主要的代码逻辑如下。

```
case ElectedLeader => { //处理当前的Master被选举为Leader的消息
     //读取集群当前运行的Application、Driver Client和Worker
     val (storedApps, storedDrivers, storedWorkers) = persistenceEngine.readPersistedData()
     //如果所有的元数据都是空的，那么就不需要恢复
     state = if (storedApps.isEmpty && storedDrivers.isEmpty && storedWorkers.isEmpty) {
          RecoveryState.ALIVE
     } else {
          RecoveryState.RECOVERING
     }
     logInfo("I have been elected leader! New state: " + state)
     if (state == RecoveryState.RECOVERING) {
          //开始恢复集群之前的状态，恢复实际上就是通知Application和Worker，
```

```
            //Master已经更改
            beginRecovery(storedApps, storedDrivers, storedWorkers)
            //在WORKER_TIMEOUT毫秒后，尝试将恢复标记为完成
            recoveryCompletionTask =
            context.system.scheduler.scheduleOnce(WORKER_TIMEOUT millis, self,
                CompleteRecovery)
        }
    }
```

对于已经有 Application 运行的集群来说，Master 故障恢复的时候，是需要将 Application、Worker 和 Driver Client 的元数据恢复的。恢复数据通过调用 beginRecovery 实现。对于 Application，Master 会将它的状态设置为 UNKNOWN 后通知 AppClient：

```
app.state = ApplicationState.UNKNOWN
app.driver ! MasterChanged(masterUrl, masterWebUiUrl)
```

对于 Worker，Master 会以类似的方式处理：

```
worker.state = WorkerState.UNKNOWN
worker.actor ! MasterChanged(masterUrl, masterWebUiUrl)
```

Master 在接到 AppClient 和 Worker 的响应后，会将它们的状态从 UNKNOWN 设置为正常的状态。

对于 Driver Client，只有在分配给的 Worker 出现问题时才会重新调度这个 Driver Client。因此这里的恢复只是将它加到 Master 在内存中维护的 Driver Client 列表中。

由于这里的恢复只是以异步的方式通知 AppClient 和 Worker，那么什么时候才会结束呢？首先，Master 在接到 AppClient 的响应（消息 MasterChangeAcknowledged）和 Worker 的响应（消息 WorkerSchedulerStateResponse）后，会查看是不是所有的 Application 和 Worker 的状态都不是 UNKNOWN 了。如果确认都不是，则代表恢复已经完成。若有部分 AppClient 或者 Worker 确实出了问题，长时间没有响应呢？Master 还有第二个机制，即设置一个超时时间，如果超时之后仍有 AppClient 或者 Worker 未响应，那么 Master 还是会认为恢复已经结束。默认超时时间为 60 秒。对于规模不同的集群，这个超时时间可以通过 spark.worker.timeout 来设置，单位是秒。

Master 在判定恢复已经结束时会调用 completeRecovery()，由它来完成恢复的最终处理：

```
def completeRecovery() {
    // 由于存在多个时机调用，因此需要加锁以确保只能执行一次
    synchronized {
        if (state != RecoveryState.RECOVERING) { return }
        state = RecoveryState.COMPLETING_RECOVERY
    }

    //将所有未响应的Worker和Application删除
    workers.filter(_.state==WorkerState.UNKNOWN).foreach(removeWorker)
    apps.filter(_.state==ApplicationState.UNKNOWN).foreach(finishApplication)

   //对于未分配Worker的Driver Client（有可能Worker已经死掉），
   //确定是否需要重新启动
   drivers.filter(_.worker.isEmpty).foreach { d =>
       logWarning(s"Driver ${d.id} was not found after master recovery")
       if (d.desc.supervise) {//需要重新启动Driver Client
           logWarning(s"Re-launching ${d.id}")
           relaunchDriver(d)
       } else {//将没有设置重启的Driver Client删除
           removeDriver(d.id, DriverState.ERROR, None)
           logWarning(s"Did not re-launch ${d.id} because it was not supervised")
       }
    }
    //设置Master的状态为ALIVE，此后Master开始正常工作
    state = RecoveryState.ALIVE
    schedule() //开始新一轮的资源调度
    logInfo("Recovery complete - resuming operations!")
}
```

5.4.2 Worker 的启动

Worker 的启动相对于 Master 的启动要简单很多。Worker 启动只会做一件事情，就是向 Master 注册。

在接到 Worker 的注册请求后，如果 Master 是 Active 的并且 Worker 没有注册过，那么 Master 会回复 Worker 消息 RegisterWorker，表示 Worker 注册成功；若注册失败，Master 会回复消息 RegisterWorkerFailed，Worker 接到该消息后直接退出（由于

Worker 会重复多次发送请求，因此退出前需要判断是否注册成功了，如果没有注册成功才会退出；如果已经注册成功了，那么忽略这个消息）。

Worker 在向 Master 注册的时候有重试机制，即在指定时间如果收不到 Master 的响应，那么 Worker 会重新发送注册请求。目前重试的次数至多为 16 次。为了避免所有的 Worker 都在同一个时刻向 Master 发送注册请求，每次重试的时间间隔是随机的。而且前 6 次的重试间隔在 5 ～ 15 秒，而后 10 次的重试间隔在 30 ～ 90 秒。

在 Worker 刚启动时，会调用 org.apache.spark.deploy.worker.Worker#registerWithMaster 来进行注册。第一次调用 registerWithMaster 的时候除了向所有的 Master 发出注册请求，还要启动一个定时器，在超时的时候进行重新注册。

```
registered = false                          //标记未注册，为true则代表注册成功
    tryRegisterAllMasters()                 //向所有的Master发出注册请求
    connectionAttemptCount = 0              //记录重试次数
    registrationRetryTimer = Some {         //开启定时器，超时时间在5～15秒

context.system.scheduler.schedule(INITIAL_REGISTRATION_RETRY_INTERVAL,
            INITIAL_REGISTRATION_RETRY_INTERVAL, self, ReregisterWithMaster)
  }
```

tryRegisterAllMasters() 会向所有的 Master 发送注册请求：

```
for (masterUrl <- masterUrls) {
    val actor = context.actorSelection(Master.toAkkaUrl(masterUrl))
    actor ! RegisterWorker(workerId, host, port, cores, memory, webUi.boundPort,
                           publicAddress)
}
```

而定时器在超时后会向自己发送消息 ReregisterWithMaster，这个消息由 reregisterWithMaster() 处理。如果在调用它的时候已经注册成功了，那么它会取消定时器后返回；如果还未注册成功并且重试次数未到阈值，那么就会开始新一轮的注册。如果重试次数达到 6 次，那么需要重置定时器，以便重试间隔增加到 30 ～ 90 秒。

如果超过阈值还未注册成功，那么 Worker 会直接退出。

注册成功后，Worker 就可以对外服务了，即可以接收 Master 的指令等。

5.5 集群容错处理

容错（fault tolerance）指的是在一个系统的部分模块出现错误的情况下还能持续地对外提供服务；如果出现了服务质量的下降，这个下降也是和出错的严重性成正比的。对于没有容错的系统，即使一个微小的错误也可能会导致整个服务停止。

对于一个集群来说，机器故障、网络故障等都被视为常态，尤其是当集群达到一定规模后，可能每天都会有物理故障导致某台机器不能提供服务。因此对于分布式系统来说，应对这种场景的容错也是设计目标之一。接下来将从 Master、Worker 和 Executor 的异常退出出发，讨论 Spark 是如何处理的。

提到容错，不得不提一下容灾（或者称为灾难恢复，disaster recovery）。容灾技术是通过在异地建立和维护一个备份系统，利用地理上的分散性来保证数据对于灾难性事件的抵抗能力。容灾系统在实现中可以分为两个层次：数据容灾和应用容灾。数据容灾指建立一个异地的数据系统，作为本地关键应用数据的一个备份。应用容灾是在数据容灾的基础上，在异地建立一套完整的和本地生产系统相同的备份应用系统（可以是互为备份）。在灾难情况下，由远程系统迅速接管业务运行。

容错和容灾都是为了实现系统的高可用性，容错是在系统的部分模块出现问题时的错误恢复机制；容灾则是在整个系统的层面，通过使用数据和应用的镜像，来实现服务的高可用性的。

5.5.1 Master 异常退出

如果 Master 异常退出，此时新的计算任务就无法进行提交了。虽然老的计算任务可以继续运行，但是由于状态更新等都中断了，很多功能也同时会受到影响。比如计算任务完成后的资源回收，这个回收指令是 Master 发送给 Worker 的。因此，Master 的异常退出，是一个非常严重的错误。

前面提到过集群可以部署多个 Master，借助 ZooKeeper 的 Leader 选举机制选出一个 Master 作为集群的管理者，其他的都作为备份。因此，在这种情况下 Master 的异常退出，ZooKeeper 会在备份的 Master 中选择一个充当集群的管理者。这个被新选

出来的 Master 会首先从 ZooKeeper 中读取集群的元数据（包括 Worker、Driver Client 和 Application 的信息）进行数据恢复，然后告知 Worker 和 AppClient，Master 已经更改的消息。在收到所有的 Worker 和 AppClient 的响应或者超时后，Master 就会变成 ACTIVE 的状态，并开始对外提供服务。因此，对于生产环境的系统，推荐使用这种方式，具体过程示意图如图 5-11 所示。

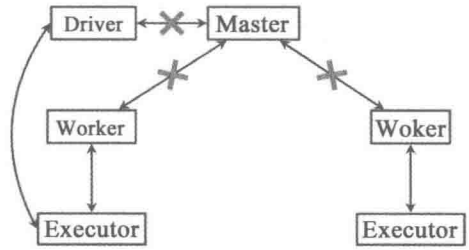


图 5-11　Master 的异常退出

对于使用 FILESYSTEM 方式的 Master，推荐通过 supervisor 拉起 Master 进程。supervisor 是用 Python 开发的一套通用的进程管理程序，能将一个普通的命令行进程变为后台 daemon，并监控进程状态，异常退出时能自动重启。因此，如果不是由于机器故障导致的 Master 异常退出，那么 Master 在被 supervisor 重新拉起后，经过数据恢复和通知 Worker 和 AppClient 后，就可以正常对外服务了。但是不推荐生产环境使用这种方式，因为从一个长远的角度看，机器出故障的概率非常高，因此如果出现这种问题，整个集群可能会停服很久。

对于使用 NONE 方式的 Master，如果也是通过 supervisor 拉起的话，那么和 FILESYSTEM 方式的唯一区别就是集群的元数据信息全部丢失。

对于用户自定义的方式，就需要根据用户的逻辑和实现来决定。

5.5.2　Worker 异常退出

对于一个集群来说，Worker 的异常退出发生概率非常高。Worker 退出时，集群是如何进行容错处理的呢？

1）Worker 在退出前，会将所有运行在它上面的 Executor 和 Driver Client 删除。

```
executors.values.foreach(_.kill())          //杀死所有的Executor
drivers.values.foreach(_.kill())            //杀死所有的Driver Client
shuffleService.stop()                       //停止Shuffle服务
```

2）Worker 需要周期性地向 Master 发送心跳消息。这个周期就是 spark.worker.timeout（默认值 60 秒）设置的 1/4。由于 Worker 的异常退出，使得它的心跳会超时，Master 认为该 Worker 已经异常退出，那么 Master 会将该 Worker 上运行的所有 Executor 的状态标记为丢失（ExecutorState.LOST），然后将这个状态更新通过消息 ExecutorUpdated 通知 AppClient；对于该 Worker 上运行的 Driver Client，如果它设置了需要重启（即设置了 supervise），那么需要重新调度来重新启动这个 Driver Client，否则直接将它删除，并且将状态设置为 DriverState.ERROR。上述逻辑的实现在类 org.apache.spark.deploy.master.Master 的成员 removeWorker 中，感兴趣的读者可以通过代码来加深理解。

3）AppClient 接到 Master 的 StatusUpdate 消息后会将状态更新汇报到 org.apache.spark.scheduler.cluster.SparkDeploySchedulerBackend，而它会根据消息内容来判断是否是 Executor 异常退出。接下来的调用栈：

a）org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend.DriverActor#removeExecutor
b）org.apache.spark.scheduler.TaskSchedulerImpl#executorLost

然后重新进行调度，Task 会被分配新的 Executor，完成最终的计算。

这个过程的示意图如图 5-12 所示。

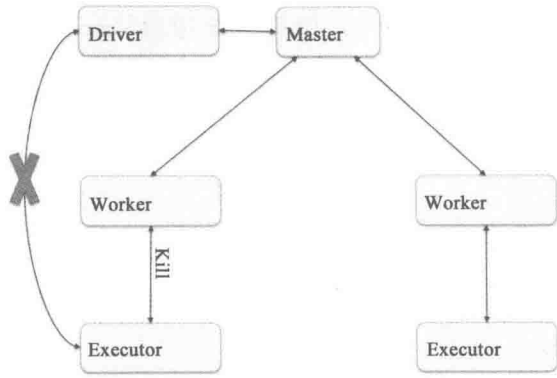


图 5-12　Worker 的异常退出

Worker 在退出的时候，会通过 org.apache.spark.deploy.worker.ExecutorRunner 杀死 Executor。ExecutorRunner 实际上会将以进程的方式启动 org.apache.spark.executor.CoarseGrained-ExecutorBackend。这个启动过程的实现在 fetchAndRunExecutor。因此，Executor-Runner 杀死 Executor 实际上就是杀死 CoarseGrainedExecutorBackend 的过程。

5.5.3　Executor 异常退出

Executor 模块负责运行 Task 计算任务，并将计算结果回传到 Driver。Spark 支持多种资源调度框架，这些资源框架在为计算任务分配了资源后，最后都会使用 Executor 模块完成最终的计算。

在讲解 Executor 异常退出的容错机制前，需要先了解 Executor 的启动机制。Worker 接收到 Master 的 LaunchExecutor 命令后，会创建 org.apache.spark.deploy.worker.Executor-Runner。

org.apache.spark.deploy.worker.ExecutorRunner#fetchAndRunExecutor 会负责 Driver 通过 AKKA 通信。它在启动后，会首先向 Driver 注册 Executor：

```
driver = context.actorSelection(driverUrl)
driver ! RegisterExecutor(executorId, hostPort, cores)
```

Driver 是 org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend::DriverActor 的 Reference。Driver 在接到 RegisterExecutor 后，会将 Executor 的信息保存在本地，并且调用 CoarseGrainedSchedulerBackend.DriverActor#makeOffers 实现在 Executor 上启动 Task。回到 CoarseGrainedExecutorBackend，它在接到 Driver 的回应 RegisteredExecutor 后，会创建一个 org.apache.spark.executor.Executor。至此，Executor 创建完毕。Coarse-Grained-Executor-Backend 唯一对应一个 Executor，它负责与 Driver 的通信，汇报 Executor 的状态。而 Executor 负责 Task 最终的计算任务。

ExecutorRunner 在启动了 CoarseGrainedExecutorBackend 后，会监控 Executor 的状态。如果 Executor 退出，它将通过 ExecutorStateChanged 汇报给其所在的 Worker，

Worker 会将这个消息转发到 Master。由于 Executor 是异常退出，Master 将会为该 Application 分配新的 Executor。如果失败次数超过 10 次，那么就会将这个 Application 标记为失败，具体过程如图 5-13 所示。

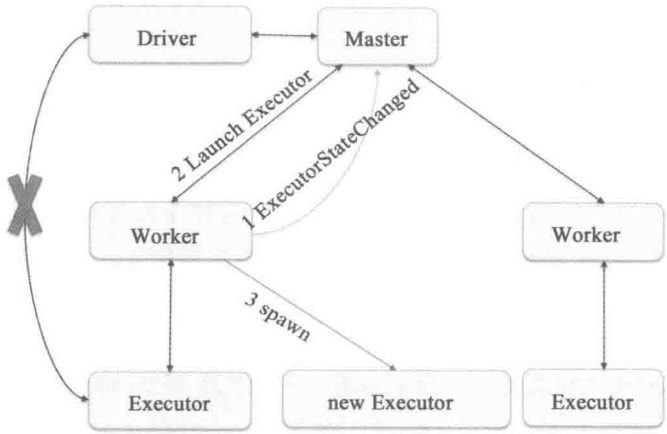


图 5-13 Executor 的异常退出

5.6 Master HA 实现详解

Standalone 是一个采用 Master/Slave 的典型架构，所以 Master 会出现单点故障（Single Point of Failure，SPOF）问题。在前面介绍 Master 的启动时，已经介绍过 Spark 可以选用 ZooKeeper 来实现高可用性（High Availability，HA）。本节将详细讲解这个过程的实现。

众所周知，ZooKeeper 提供了一个 Leader 选举机制，利用这个机制可以保证虽然集群存在多个 Master 但是只有一个是 Active 的。当 Active 的 Master 出现故障时，另外的一个 Standby Master 会被选举出来。由于集群的信息，包括 Worker、Driver Client 和 Application 的信息都已经持久化到 ZooKeeper 中，因此在切换的过程中只会影响新 Job 的提交，对于正在进行的 Job 没有任何的影响。 加入 ZooKeeper 的集群整体架构如图 5-14 所示。

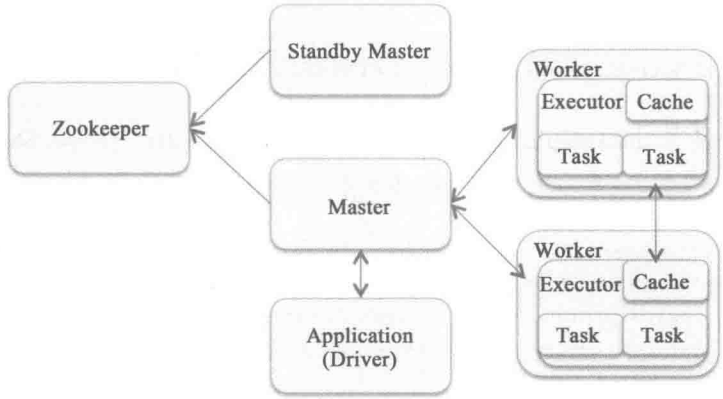


图 5-14　基于 ZooKeeper 的整体架构图

5.6.1　Master 启动的选举和数据恢复策略

除了集群搭建后的第一次启动，Master 每次启动都会恢复集群当前的运行状态。这些状态包括当前正在运行的 Application、Driver Client 和 Worker。5.3.1 节提到过，当前 Standalone 模式支持四种策略：

1）ZOOKEEPER

实现了基于 ZooKeeper 的选举机制，元数据信息会持久化到 ZooKeeper 中。因此在 Master 故障后，ZooKeeper 会在 Standby 的 Master 中选举出新的 Master，新的 Master 在启动后会从 ZooKeeper 中获取元数据信息并且恢复这些数据。

2）FILESYSTEM

集群的元数据信息会保存在本地文件系统。而 Master 启动后会则立即成为了 Active 的 Master。如果不考虑机器本身的故障和在设置了 Master 进程退出后能自动重启的前提下，这种方式也是可以接受的。

3）CUSTOM

这个是用户自定义的。如果需要自定义的机制，那么需要实现 org.apache.spark.deploy.master.StandaloneRecoveryModeFactory，并且将实现的类的名字配置到 spark.deploy.recoveryMode.factory。

4）NONE

不会持久化集群的元数据，Master 在启动后会立即接管集群的管理工作。

这些策略是在 spark.deploy.recoveryMode 设置的，Master 会在 preStart() 中根据这个设置来选择不同的选举机制和元数据持久化 / 恢复机制：

```
val (persistenceEngine_, leaderElectionAgent_) = RECOVERY_MODE match {
     case "ZOOKEEPER" => //使用ZooKeeper的方式
          logInfo("Persisting recovery state to ZooKeeper")
          val zkFactory =new ZooKeeperRecoveryModeFactory(conf, Serialization-Extension
            (context.system))
(zkFactory.createPersistenceEngine(), zkFactory.createLeaderElectionAgent
          (this))
          case "FILESYSTEM" => //使用文件系统的方式
          val fsFactory =new FileSystemRecoveryModeFactory(conf, Serialization-
              Extension(context.system))
(fsFactory.createPersistenceEngine(), fsFactory.createLeaderElectionAgent(this))
    case "CUSTOM" => //用户自定义
        val clazz = Class.forName(conf.get("spark.deploy.recoveryMode.factory"))
        val factory = clazz.getConstructor(conf.getClass, Serialization.getClass)
            .newInstance(conf, SerializationExtension(context.system))
            .asInstanceOf[StandaloneRecoveryModeFactory]
        (factory.createPersistenceEngine(), factory.createLeaderElectionAgent(t
            his))
        case _ => //无
            (new BlackHolePersistenceEngine(), new MonarchyLeaderAgent(this))
}
persistenceEngine = persistenceEngine_
leaderElectionAgent = leaderElectionAgent_
```

其中 persistenceEngine 通过 persist(name: String, obj: Object) 实现了元数据持久化，通过 readPersistedData() 来恢复元数据。leaderElectionAgent 实现了 Leader 的选举，对于 FILESYSTEM 和 NONE 来说，它的实现是 MonarchyLeaderAgent，在创建的时候就会把创建它的 Master 设置为 Leader：

```
private[spark] class MonarchyLeaderAgent(val masterActor: LeaderElectable)
    extends LeaderElectionAgent {
    masterActor.electedLeader() //直接将传入的Master设置为Leader
}
```

而对于 NONE 来说，它不会持久化集群的任何数据，而是通过一个空的

persistenceEngine 来实现的。这种实现的好处是可以做到对外接口的统一。具体实现是 org.apache.spark.deploy.master.BlackHolePersistenceEngine。

将所有的策略抽象为持久化和选举两部分，从而实现了一个 Pluggable（可插拔、可扩展的）框架。因此第三方可以很容易地实现自定义的策略。

Spark 使用的并不是 ZooKeeper 的原生 API，而是 Apache Curator。Curator 在 ZooKeeper 上做了一层很友好的封装。

5.6.2　集群启动参数的配置

为了使用 ZooKeeper，需要设置相关的参数，可以通过在 spark-env.sh 中设置：

```
spark.deploy.recoveryMode=ZOOKEEPER
spark.deploy.zookeeper.url=zk_server_1:2181,zk_server_2:2181,zk_server_3:2181
spark.deploy.zookeeper.dir=/spark
```

或者通过以下方式设置：

```
export SPARK_DAEMON_JAVA_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER "
export SPARK_DAEMON_JAVA_OPTS="${SPARK_DAEMON_JAVA_OPTS} -Dspark.deploy.
zookeeper.url=zk_server1:2181,zk_server_2:2181,zk_server_3:2181"
```

各个参数的意义如表 5-9 所示。

表 5-9　启动参数

参　数	默 认 值	含　义
spark.deploy.recoveryMode	NONE	恢复模式（即 Master 启动后集群元数据的恢复模式），当前支持 4 种： 1）ZooKeeper 2）FILESYSTEM 3）CUSTOM 4）NONE
spark.deploy.zookeeper.url	无	ZooKeeper 的 Server 地址
spark.deploy.zookeeper.dir	/spark	ZooKeeper 保存集群元数据信息的根目录，保存的元数据信息包括 Worker，Driver Client 和 Application

5.6.3 Curator Framework 简介

Standalone Master 使用到的 Curator 主要是 org.apache.curator.framework.CuratorFramework 和 org.apache.curator.framework.recipes.leader.{LeaderLatchListener, LeaderLatch}。在详细介绍具体实现前，本节会主要介绍 Curator 的基础知识，其中涵盖 Spark 实现用到的部分。

Curator Framework 极大地简化了 ZooKeeper 的使用，它提供了 high-level 的 API，并且基于 ZooKeeper 添加了很多特性，包括：

1）自动连接管理：连接到 ZooKeeper 的 Client 有可能会连接中断，Curator 处理了这种情况，对于 Client 来说自动重连是透明的。

2）简洁的 API：简化了原生态的 ZooKeeper 的方法，事件等；提供了一个简单易用的接口。

3）Recipe 的实现（更多介绍请查阅 http://curator.apache.org/curator-recipes/index.html）：

a）Leader 选举

b）共享锁

c）缓存和监控

d）分布式队列

e）分布式优先队列

Curator Framework 通过 CuratorFrameworkFactory 来创建线程安全的 ZooKeeper 的实例，即通过 org.apache.curator.framework.CuratorFrameworkFactory#newClient 来创建 ZooKeeper 的实例，并可以传入不同的参数对实例进行控制。获取实例后，必须通过 start() 来启动这个实例，在结束时，需要调用 close()。Spark 在 org.apache.spark.deploy.master.SparkCuratorUtil 中对它进行了进一步的封装，使得调用更加简单：

```
def newClient(conf: SparkConf): CuratorFramework = {
    val ZK_URL = conf.get("spark.deploy.zookeeper.url") //获取ZooKeeper的地址
    val zk = CuratorFrameworkFactory.newClient(ZK_URL,
```

```
        ZK_SESSION_TIMEOUT_MILLIS, ZK_CONNECTION_TIMEOUT_MILLIS,
    new ExponentialBackoffRetry(RETRY_WAIT_MILLIS,
        MAX_RECONNECT_ATTEMPTS))
    zk.start()
    zk
}
```

对于传入的 SparkConf，它只需要 ZooKeeper 的地址，即 spark.deploy.zookeeper.url。其他的参数都是在 SparkCuratorUtil 硬编码的：

```
val ZK_CONNECTION_TIMEOUT_MILLIS = 15000
val ZK_SESSION_TIMEOUT_MILLIS = 60000
val RETRY_WAIT_MILLIS = 5000
val MAX_RECONNECT_ATTEMPTS = 3
```

调用者只需要传入 SparkConf 就可以创建 ZooKeeper 的实例，进一步简化了 Curator 的使用。创建了 ZooKeeper 的实例后，就可以使用该实例进行读写操作了。下面假设创建的实例为 zk，来简单介绍一下这些接口的使用方式。

1）创建 ZooKeeper 的实例：

```
val zk: CuratorFramework = SparkCuratorUtil.newClient(conf)
```

2）创建一个路径 path：

```
zk.create().creatingParentsIfNeeded().forPath(path)
```

3）获取一个路径为 path 的节点的数据：

```
val fileData = zk.getData().forPath(path)
```

4）将一个序列化后的数据 serialized 写入 path：

```
zk.create().withMode(CreateMode.PERSISTENT).forPath(path, serialized)
```

5）删除一条数据（路径为 path）：

```
zk.delete().forPath(path)
```

6）删除一个路径 path：

```
if (zk.checkExists().forPath(path) != null) {
    for (child <- zk.getChildren.forPath(path)) {
        zk.delete().forPath(path + "/" + child) //删除子节点
    }
```

```
        zk.delete().forPath(path)
    }
```

除了 ZooKeeper 作为一个元数据的存储引擎外，还需要关注它的选举机制的使用方式。主要需要关注两个 Recipe：org.apache.curator.framework.recipes.leader.{LeaderLatchListener, LeaderLatch}，它们都是由 Java 实现的。

首先看一下 LeaderlatchListener，它在 LeaderLatch 状态变化的时候被通知：

1）在该节点被选为 Leader 的时候，会调用接口 isLeader()。

2）在节点被剥夺 Leader 的时候，会调用接口 notLeader()。

```
public interface LeaderLatchListener
{
   public void isLeader();
   public void notLeader();
}
```

但是由于通知是异步的，在接口被调用的时候，这个状态可能又会发生变化，因此需要重复确认 LeaderLatch 的 hasLeadership() 是否的确是 true/false。这一点将在接下来 Spark 的实现中得到体现。

LeaderLatch 负责在众多连接到 ZooKeeper 集群的竞争者中选择一个 Leader。Leader 的选择机制可以由 ZooKeeper 的具体实现决定，LeaderLatch 对它进行了很好的封装。这里只需要了解 LeaderLatch 的用法即可。在创建了 LeaderLatch 的实例后，需要通过 addListener“注册”LeaderLatchListener，从而在状态改变时通知监听者。

```
public void addListener(LeaderLatchListener listener)
{
    listeners.addListener(listener);
}
```

最后，LeaderLatch 需要调用 start() 来开始竞争以便成为 Leader。

5.6.4　ZooKeeperLeaderElectionAgent 的实现

借助 Curator，Spark 实现 Master 的 HA 就变得非常简单，ZooKeeperLeaderElectionAgent 实现了接口 LeaderLatchListener，在 isLeader() 确认所属的 Master 被选为 Leader 后，向 Master 发送消息 ElectedLeader，Master 会在恢复了元数据后将自己的状态改为 ALIVE。当 noLeader() 被调用时，它会向 Master 发送消息 RevokedLeadership，此时，Master 直接退出。

接下来看一下具体的实现。在 org.apache.spark.deploy.master.ZooKeeperLeaderElectionAgent 创建的时候，会创建 ZooKeeper 的实例，同时创建 LeaderLatch：

```
zk = SparkCuratorUtil.newClient(conf)
leaderLatch = new LeaderLatch(zk, WORKING_DIR)
leaderLatch.addListener(this) //它实现了LeaderLatchListener
leaderLatch.start()
```

在调用 leaderLatch.start() 后，就启动了 Leader 的竞争与选举。就如前面分析的，主要逻辑在 isLeader 和 notLeader 中，即状态变化时 leaderLatch 会调用 Listener 的这两个接口。注意，接口实现的时候，需要再次确认状态是否再次改变了：

```
override def isLeader() {
    synchronized {
    //有可能状态已经再次改变，即Leader已经再次变化，因此需要再次确认
    if (!leaderLatch.hasLeadership) {
        return
    }
    //已经被选举为Leader
    logInfo("We have gained leadership")
    updateLeadershipStatus(true)
    }
}

override def notLeader() {
    synchronized {
        //有可能状态已经再次改变，即Leader已经再次变化，因此需要再次确认
        if (leaderLatch.hasLeadership) {
            return
        }
        //被剥夺Leader
        logInfo("We have lost leadership")
```

```
            updateLeadershipStatus(false)
        }
    }
```

updateLeadershipStatus 的逻辑很简单，就是向 Master 发送消息。

```
def updateLeadershipStatus(isLeader: Boolean) {
    if (isLeader && status == LeadershipStatus.NOT_LEADER) {
        status = LeadershipStatus.LEADER
        masterActor.electedLeader() //Master已经被选举为Leader
    } else if (!isLeader && status == LeadershipStatus.LEADER) {
        status = LeadershipStatus.NOT_LEADER
        masterActor.revokedLeadership() //Master已经被剥夺Leader
    }
}
```

为了解决 Standalone 模式下 Master 的单点故障问题，Spark 采用了 ZooKeeper 提供的选举机制和作为集群元数据的存储引擎。Spark 并没有采用 ZooKeeper 原生的 Java API，而是采用了 Curator，一个对 ZooKeeper 进行了封装的框架。采用 Curator 后，Spark 不用管理与 ZooKeeper 的连接，数据读和写的接口更加简单，这些对于 Spark 来说都是透明的。也就是说，Spark 仅仅使用了 100 行代码，就实现了 Master 的高可用性。

5.7 小结

通过对 Deploy 模块的分析来看，该模块相对来说比较简单，也没有特别复杂的逻辑结构。Standalone 并不是为了替代更谈不上完善 Mesos/YARN，正如前面所说的，Deploy 模块是为了降低使用门槛，使更多的用户能够使用 Spark 而实现的一种方案。

当然现阶段看来还略显简陋，比如 Application 的调度方式 (FIFO) 是否会造成小应用长时间等待大应用的结束，是否有更好的调度策略；资源的衡量标准是否可以更多、更合理，而不单单是 CPU 数量，因为现实场景中有的应用是本地 IO 密集型，有的是网络 IO 密集型，这样就算 CPU 资源有富余，调度新的 Application 也不一定会很有意义。

总的来说作为一种简单替代方式，Deploy 模块对于推广 Spark 还是有积极意义的。

在为 Application 分配了计算资源后，Scheduler 模块的 TaskScheduler 会根据不同的调度策略为 Task 分配 Application 得到计算资源 Executor，并且将 Task 发送到 Executor。接下来的章节会分析 Executor 的详细实现。

Chapter 6 第 6 章

Executor 模块详解

Executor 模块负责运行 Task 计算任务，并将计算结果回传到 Driver。Spark 支持多种资源调度框架，这些资源框架在为计算任务分配资源后，最后都会使用 Executor 模块完成最终的计算。

图 6-1 是计算的一个顶层逻辑关系图。每个 Spark 的 Application 都是从 SparkContext 开始的，它通过 Cluster Manager 和 Worker 上的 Executor 建立联系，由每个 Executor 完成 Application 的部分计算任务。不同的 Cluster Master，即资源调度框架的实现模式会有区别，但是任务的划分和调度都是由运行 SparkContext 端的 Driver 完成的，资源调度框架在为 Application 分配资源后，将 Task 分配到计算的物理单元 Executor 去处理。本章将着重讲解 Executor 模块的实现。

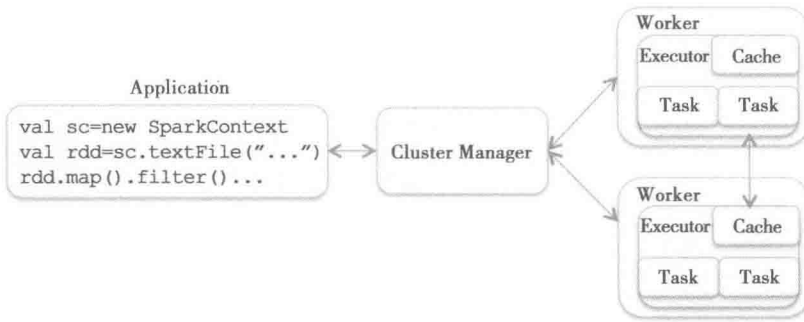


图 6-1　计算的顶层逻辑关系

6.1　Standalone 模式的 Executor 分配详解

在 Standalone 模式下集群启动时，Worker 会向 Master 注册，使得 Master 可以感知进而管理整个集群；Master 通过借助 Zookeeper，可以简单实现高可用性；而应用方通过 SparkContext 这个与集群的交互接口，在创建 SparkContext 时就完成了 Application 的注册，由 Master 为其分配 Executor；在应用方创建了 RDD 并且在这个 RDD 上进行了很多的转换后，触发 Action，通过 DAGScheduler 将 DAG 划分为不同的 Stage，并将 Stage 转换为 TaskSet 交给 TaskSchedulerImpl；再由 TaskSchedulerImpl 通过 SparkDeploySchedulerBackend 的 reviveOffers，最终向 ExecutorBackend 发送 LaunchTask 的消息；ExecutorBackend 接收到消息后，启动 Task，开始在集群中启动计算。具体过程如图 6-2 所示。

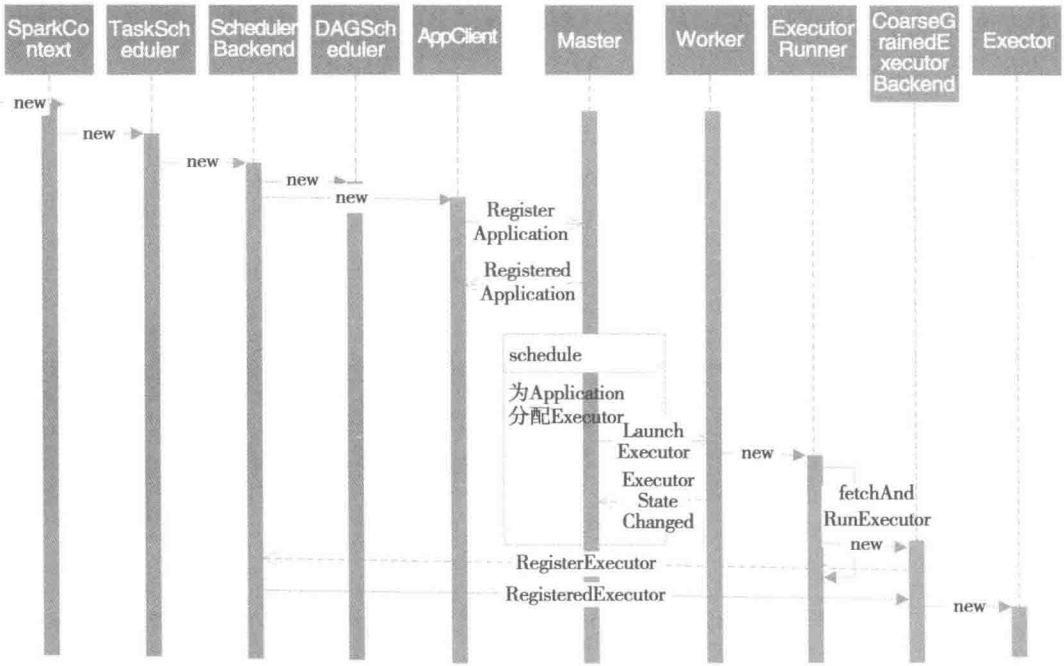


图 6-2　Standalone 模式下 Executor 的分配序列图

SparkContext 是用户应用和 Spark 集群的交换的主要接口，用户应用一般首先要创建它。如果你使用 SparkShell，你不必显式地创建它，系统会自动创建一个名

为 sc 的 SparkContext 的实例。创建 SparkContext 的实例，主要工作是设置一些参数，比如 Executor 使用到的内存的大小。如果系统的配置文件已经设置，那么就直接读取该配置；否则读取环境变量。如果都没有设置，那么取默认值为 512M。当然，这个数值还是很保守的，特别是在内存已经不是那么昂贵的今天。除了加载这些集群的参数，它完成了 TaskScheduler 和 DAGScheduler 的创建。关于 TaskScheduler 和 DAGScheduler 可以参阅 4.3 节。

6.1.1 SchedulerBackend 创建 AppClient

SparkDeploySchedulerBackend 是 Standalone 模式的 SchedulerBackend。在 org.apache.spark.scheduler.cluster.SparkDeploySchedulerBackend#start 中会创建 AppClient，而 AppClient 可以向 Standalone 的 Master 注册 Application，然后 Master 会通过 Application 的信息为它分配 Worker，包括每个 Worker 上使用 CPU core 的数目等。AppClient 向 Master 注册 Application 时需要发送以下消息：

```
RegisterApplication(appDescription: ApplicationDescription)
```

ApplicationDescription 包含一个 Application 的所有信息，包括：

```
private[spark] class ApplicationDescription(
val name: String, //Application的名字，可以通过spark.app.name设置
val maxCores: Option[Int],
// 最多可以使用CPU core的数量，可以通过spark.cores.max设置。每个Executor可以占用的内存
// 数，可以通过spark.executor.memory、SPARK_EXECUTOR_MEMORY或者SPARK_MEM设置，默
// 认值是512MB
val memoryPerSlave: Int,
// Worker Node拉起的ExecutorBackend进程的Command。在Worker接到 Master LaunchExecutor
// 后会通过ExecutorRunner启动这个Command。Command包含了启动一个Java进程所需要的信息，
// 包括启动的ClassName、所需参数、环境信息等
val command: Command,
var appUiUrl: String,
// Application 的web ui的hostname:port如果spark.eventLog.enabled（默认为false）
// 指定为true的话，eventLogFile就设置为spark.eventLog.dir定义的目录
val eventLogFile: Option[String] = None)
```

其中，Command 中比较重要的是 Class Name 和启动参数，在 Standalone 模式中，Class Name 就是 org.apache.spark.executor.CoarseGrainedExecutorBackend；而启动参数是一个列表：

```
val args = Seq(driverUrl, "{{EXECUTOR_ID}}", "{{HOSTNAME}}", "{{CORES}}",
   "{{APP_ID}}", "{{WORKER_URL}}")
```

除了 driverUrl，这些参数值都是未知的，比如 Executor ID，只有在分配了 Executor 时这个值才可用。因此在通过 Command 启动时，需要将这些参数替换成真实分配的值：

```
def substituteVariables(argument: String): String = argument match {
    case "{{WORKER_URL}}" => workerUrl
    case "{{EXECUTOR_ID}}" => execId.toString
    case "{{HOSTNAME}}" => host
    case "{{CORES}}" => cores.toString
    case "{{APP_ID}}" => appId
    case other => other
}
```

driverUrl 是 Driver 的 URL，Executor 可以通过这个 URL 创建 Driver Actor 的 Reference，然后通过这个 Reference 和 Driver 进行通信。这个 Actor 的实现是 org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend.DriverActor。

SparkDeploySchedulerBackend 还实现了 org.apache.spark.deploy.client.AppClientListener。

AppClientListener 是一个 trait，主要为了确保 AppClient 在某种情况下能及时通知 SparkDeploySchedulerBackend 某些状态更新。在以下情况中，AppClient 会回调相关函数以通知 SparkDeploySchedulerBackend：

1）向 Master 成功注册 Application，即成功连接到集群。

2）断开连接，如果当前 SparkDeploySchedulerBackend::stop == false，那么可能原来的 Master 有故障了，待新的 Master 准备就绪后，会重新恢复原来的连接。

3）Application 由于不可恢复的错误停止了，这个时候需要重新提交出错的 TaskSet。

4）添加一个 Executor，在这里仅仅实现了打印 log，并没有额外的逻辑。

5）删除一个 Executor，可能有两个原因：一个是 Executor 退出了，这里可以得到 Executor 的退出码；另一个是由于 Worker 的退出导致运行在它之上的 Executor 退出。这两种情况需要不同的逻辑来处理。

SparkDeploySchedulerBackend 创建 AppClient 时，会将自身作为一个参数传递给 AppClient，这样 AppClient 在接到 Master 某些状态更新后，就能调用它来实现状态更新的通知。一些实现细节描述如下。

org.apache.spark.scheduler.cluster.SparkDeploySchedulerBackend#start 中 创 建 AppClient：

```
client = new AppClient(sc.env.actorSystem, masters, appDesc, this, conf)
```

注意第 4 个参数 this。AppClient 将它作为自己的数据成员 listener：

```
private[spark] class AppClient(
actorSystem: ActorSystem,
masterUrls: Array[String],
appDescription: ApplicationDescription,
listener: AppClientListener,
conf: SparkConf)
```

在 AppClient 接到 Master 的某些响应后，会调用 listener 的相应接口，以 Application 被 Master 删除的消息（ApplicationRemoved）为例，通过调用 listener.dead(reason) 就可以通知 SparkDeploySchedulerBackend 了。

总之，SparkDeploySchedulerBackend 准备好 Application 相关的信息后，创建 AppClient；AppClient 通过调用 SparkDeploySchedulerBackend 的某些接口来通知 Driver 某些状态更新；通过 ApplicationDescription 中携带的 Driver Actor 的 URL，Executor 可以通过 AKKA 和 Driver 进行通信。

6.1.2 AppClient 向 Master 注册 Application

AppClient 是 Application 和 Master 交互的接口。它包含一个实现为 org.apache.spark.deploy.client.AppClient.ClientActor 的成员变量 actor。它负责了所有与 Master 的交互。Master 与 actor 主要的消息如下：

1）RegisteredApplication(appId_, masterUrl) => // 注：来自 Master 的成功注册 Application 的消息。

2）ApplicationRemoved(message) => // 注：来自 Master 的删除 Application 的消息。

Application 执行成功或者失败最终都会被删除。

3）ExecutorAdded(id: Int, workerId: String, hostPort: String, cores: Int, memory: Int) => // 注：来自 Master 的消息。

4）ExecutorUpdated(id, state, message, exitStatus) => ?// 注：　来　自 Master 的 Executor 状态更新的消息，如果 Executor 是完成的状态，那么回调 SchedulerBackend 的 executorRemoved 的函数。

5）MasterChanged(masterUrl, masterWebUiUrl) =>??// 注：来自新竞选成功的 Master。Master 可以选择 ZK 实现 HA，并且使用 ZK 来持久化集群的元数据信息。因此在 Master 变成 leader 后，会恢复持久化的 Application、Driver 和 Worker 的信息。

6）StopAppClient => // 注：来自 AppClient::stop()。

注册过程如图 6-3 所示。

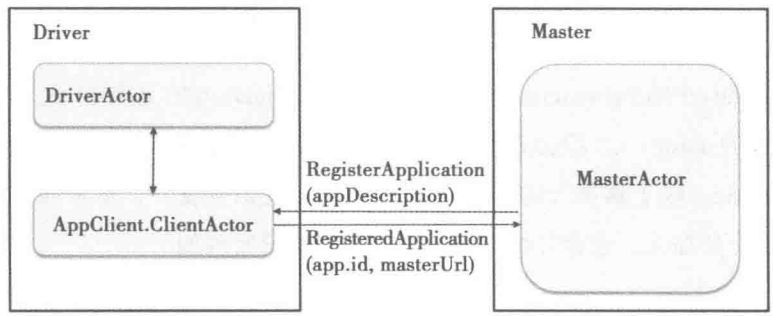


图 6-3　Application 的注册

在 actor 的 preStart 里，actor 就会向 Master 发起注册 Application 的请求：

```
def tryRegisterAllMasters() {
    for (masterUrl <- masterUrls) {
        logInfo("Connecting to master " + masterUrl + "...")
        val actor = context.actorSelection(Master.toAkkaUrl(masterUrl))
        actor ! RegisterApplication(appDescription)
    }
}
```

如果超过 20s 没有接收到注册成功的消息，那么会重新注册；如果重试超过 3 次仍未成功，那么本次注册就失败了。实际上，除非连接不上 Master，否则注册都会成

功的。

Master 端在接到注册的请求后，首先会将 Application 放到自己维护的数据结构中：

1）apps += app // HashSet[ApplicationInfo]，保存了 Master 上所有的 Application。

2）idToApp(app.id) = app // HashMap[String, ApplicationInfo], app.id 是在 Master 端分配的，格式是“app-currentdate-nextAppNumber”，其中 nextAppNumber 是 Master 启动以来注册的 Application 的总数 -1，取四位数。

3）actorToApp(app.driver) = app // HashMap[ActorRef, ApplicationInfo]，app.driver 就是 org.apache.spark.deploy.client.AppClient.ClientActor。

4）addressToApp(appAddress) = app // HashMap[Address, ApplicationInfo]，appAddress = app.driver.path.address。

5）waitingApps += app // ArrayBuffer[ApplicationInfo]，等待被调度的 Application。

在调用 Master 的 PersistenceEngine 持久化 Application 的元数据后，会将结果返回 AppClient 的 actor；之后会开始新一轮的资源调度 org.apache.spark.deploy.master.Master#schedule。除了新加入的 Application 会开始资源调度外，资源的变动本身，比如新的 Worker 的加入，也会开始资源调度。核心代码实现如下：

```
case RegisterApplication(description) => {
    if (state == RecoveryState.STANDBY) {
        // Standby的Master会忽略注册消息，如果AppClient发送请求到Standby的Master，
        // 会触发超时机制（默认是20s），超时会重试
    } else {
        logInfo("Registering app " + description.name)
        val app = createApplication(description, sender)
        // app是ApplicationInfo(now, newApplicationId(date), desc, date,
        // driver, defaultCores)，driver就是AppClient的actor

        // 保存到Master维护的成员变量中
    /* apps += app;
       idToApp(app.id) = app
       actorToApp(app.driver) = app
       addressToApp(appAddress) = app
       waitingApps += app */
```

```
registerApplication(app)
logInfo("Registered app " + description.name + " with ID " + app.id)
   persistenceEngine.addApplication(app)  //持久化app的元数据信息，可以选择持
       //久化到ZooKeeper、本地文件系统，或者不持久化
sender ! RegisteredApplication(app.id, masterUrl)
   schedule() //为处于待分配资源的Application分配资源。在每次有新的Application
              //加入或者新的资源加入时都会调用schedule进行调度
  }
}
```

6.1.3　Master 根据 AppClient 的提交选择 Worker

org.apache.spark.deploy.master.Master#schedule 为处于待分配资源的 Application 分配资源。在每次有新的 Application 加入或者新的资源加入时都会调用 schedule 进行调度。为 Application 分配资源选择 Worker（Executor），现在有两种策略：

1）尽量打散，即将一个 Application 尽可能多地分配到不同的节点。可以通过设置 spark.deploy.spreadOut 来实现。默认值为 true。

2）尽量集中，即一个 Application 尽量分配到尽可能少的节点。CPU 密集型而内存占用比较少的 Application 适合使用这种策略。

对于同一个 Application，它在一个 Worker 上只能拥有一个 Executor；这并不代表一个物理节点只能部署一个 Executor：可以通过在一个物理节点上部署多个 Worker 来完成。

```
if (spreadOutApps) {//尽量打散负载，如有可能，每个executor分配一个core
    for (app <- waitingApps if app.coresLeft > 0) {
        // 可用的Worker的标准：State是ALIVE，其上并没有该Application的Executor，
        // 可用内存满足要求。在可用的Worker中，优先选择可用core数多的
        val usableWorkers = workers.toArray.filter(_.state == WorkerState.ALIVE)
        .filter(canUse(app, _)).sortBy(_.coresFree).reverse
        val numUsable = usableWorkers.length
        // 记录与usableWorkers对应的Worker上可以使用多少core
        val assigned = new Array[Int](numUsable) // Number of cores to give on each node
        // 待分配的core数，如果当前可用的core满足Application需求，那么全部分配；否则将
        // 剩余的core分配给该Application
        var toAssign = math.min(app.coresLeft, usableWorkers.map(_.
           coresFree).sum)
        var pos = 0
        while (toAssign > 0) {
```

```
                if (usableWorkers(pos).coresFree - assigned(pos) > 0) {
                // 注意usableWorkers是按照剩余core数倒序排列的，因此拥有最多可用core的
                // Worker先被使用
                    toAssign -= 1
                    assigned(pos) += 1
                    }
                    pos = (pos + 1) % numUsable
                }
                // 已经确定了分配策略，根据确定的在某个Worker上分配的core数来在Worker上启
                // 动Executor
                for (pos <- 0 until numUsable) {
                    if (assigned(pos) > 0) {
                        val exec = app.addExecutor(usableWorkers(pos), assigned(pos))
                        //在Worker上启动Executor
                        launchExecutor(usableWorkers(pos), exec)
                        app.state = ApplicationState.RUNNING
                    }
                }
            }
        } else {
            // 尽可能多地利用Worker的core，将Executor的数量减少到最低，比如对于计算密集型
            // 但是内存使用较少的Application
            for (worker <- workers if worker.coresFree > 0 && worker.state ==
                 WorkerState.ALIVE)      {
                for (app <- waitingApps if app.coresLeft > 0) {
                // 可用的Worker的标准：State是ALIVE，其上并没有该Application的Executor，
                // 可用内存满足要求
                if (canUse(app, worker)) {
                    val coresToUse = math.min(worker.coresFree, app.coresLeft)
                if (coresToUse > 0) {
                    val exec = app.addExecutor(worker, coresToUse)
                    //在Worker上启动Executor
                    launchExecutor(worker, exec)
                    app.state = ApplicationState.RUNNING
                }
            }
        }
    }
}
```

在选择了 Worker 和确定了 Worker 上 Executor 需要的 CPU Core 数后，Master 会调用 ?launchExecutor(worker: WorkerInfo, exec: ExecutorInfo) 向 Worker 发送请求，向 AppClient 的 actor 发送 Executor 已经添加的消息。同时会更新 Master 保存的 Worker

的信息，包括增加 Executor、减少可用的 CPU Core 数和 memory 数（注意，Worker 上的资源信息并不是 Worker 主动上报到 Master 的，而是 Master 主动维护的，因为所有的资源分配都是由 Master 来完成的）。Master 不会等到真正在 Worker 上成功启动 Executor 后再更新 Worker 的信息。如果 Worker 启动 Executor 失败，那么它会发送 FAILED 的消息给 Master，Master 收到该消息时再次更新 Worker 的信息即可，这种实现简化了逻辑。

```
def launchExecutor(worker: WorkerInfo, exec: ExecutorInfo) {

    logInfo("Launching executor " + exec.fullId + " on worker " + worker.id)
    // 更新Worker的信息，可用core数和memory数减去本次分配的Executor占用的，因此Worker
    // 的信息无需Worker主动汇报
    worker.addExecutor(exec)
    // 向Worker发送启动Executor的请求
    worker.actor ! LaunchExecutor(masterUrl,exec.application.id, exec.id,
        exec.application.desc, exec.cores, exec.memory)
    // 向AppClient的actor发送Executor已经添加的消息
    exec.application.driver ! ExecutorAdded(exec.id, worker.id, worker.
        hostPort, exec.cores, exec.memory)
}
```

现在的分配方式比较简单，并没有考虑节点的当前总体负载。虽然这样的方式可以使节点上 Executor 的分配比较均匀，而且单纯静态地从 Executor 分配到得 CPU core 数和 memory 数来看，负载是比较均衡的。但是从实际情况来看，可能有的 Executor 的 CPU 资源消耗比较大；而其他的 Executor 可能网络负载比较高，因此会导致集群负载不均衡和资源的倾斜。这个需要从生产环境的数据得到反馈来进一步修正和细化分配策略，以达到更好的资源利用率。

6.1.4　Worker 根据 Master 的资源分配结果创建 Executor

Worker 接收到来自 Master 的 LaunchExecutor 的消息后，会创建 org.apache.spark.deploy.worker.ExecutorRunner。

ExecutorRunner 会将在 org.apache.spark.scheduler.cluster.SparkDeploySchedulerBackend 中准备好的 org.apache.spark.deploy.ApplicationDescription 以进程的形式启动。当时以下几个参数还是未知的：

```
val args = Seq(driverUrl, "{{EXECUTOR_ID}}", "{{HOSTNAME}}", "{{CORES}}",
   "{{WORKER_URL}}")。
```

ExecutorRunner 需要将它们替换成已经分配好的实际值：

```
def substituteVariables(argument: String): String = argument match {
    case "{{WORKER_URL}}" => workerUrl
    case "{{EXECUTOR_ID}}" => execId.toString
    case "{{HOSTNAME}}" => host
    case "{{CORES}}" => cores.toString
    case "{{APP_ID}}" => appId
    case other => other
}
```

通过 command.arguments.map(substituteArguments) 可以将其替换为真实值。

org.apache.spark.deploy.worker.ExecutorRunner#fetchAndRunExecutor 会负责启动 CoarseGrainedExecutorBackend。CoarseGrainedExecutorBackend 会负责通过 AKKA 和 Driver 通信。它在启动后，会首先向 Driver 注册 Executor：

```
driver = context.actorSelection(driverUrl)
driver ! RegisterExecutor(executorId, hostPort, cores)
```

Driver 是一个 Actor 的 Reference，实现是 org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend::DriverActor 的 Reference。Driver 在接到 RegisterExecutor 消息后，会将 Executor 的信息保存在本地，并且调用 CoarseGrainedSchedulerBackend.DriverActor#makeOffers 实现在 Executor 上启动 Task。具体的实现细节请参阅 4.3 节。

回到 CoarseGrainedExecutorBackend，它在接到 Driver 的回应 RegisteredExecutor 后，会创建一个 org.apache.spark.executor.Executor。至此，Executor 创建完毕。Executor 在 Mesos、YARN 和 Standalone 模式中都是相同的，不同的只是资源的分配管理方式。

6.2 Task 的执行

org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend.DriverActor#launchTasks 会将分配到 Executor 的 Task 进行分配：

```
val executorData = executorDataMap(task.executorId)
executorData.freeCores -= scheduler.CPUS_PER_TASK
executorData.executorActor ! LaunchTask(new SerializableBuffer(serializedTask))
```

org.apache.spark.executor.CoarseGrainedExecutorBackend 收到 LaunchTask 的消息后，会调用 Executor 的 launchTask 启动 Task：

```
case LaunchTask(data) =>
    if (executor == null) {
        logError("Received LaunchTask command but executor was null")
        System.exit(1)
    } else {
        val ser = env.closureSerializer.newInstance()
        val taskDesc = ser.deserialize[TaskDescription](data.value)
        logInfo("Got assigned task " + taskDesc.taskId)
        executor.launchTask(this, taskDesc.taskId, taskDesc.name, taskDesc.
            serializedTask)
}
```

Executor 会为这个 Task 生成一个 TaskRunner（继承自 Runnable）；TaskRunner 的 run 中会执行 Task。TaskRunner 最终会被放到一个 ThreadPool 中去执行。

```
val tr = new TaskRunner(context, taskId, taskName, serializedTask)
runningTasks.put(taskId, tr)
threadPool.execute(tr)
```

接下来将介绍 org.apache.spark.executor.Executor.TaskRunner#run 的实现。

6.2.1　依赖环境的创建和分发

在 Driver 端封装 Task 时，会将 Task 依赖的文件封装到 Task 中：

```
val out = new ByteArrayOutputStream(4096)
val dataOut = new DataOutputStream(out)

// Write currentFiles
dataOut.writeInt(currentFiles.size)
for ((name, timestamp) <- currentFiles) {
    dataOut.writeUTF(name)
    dataOut.writeLong(timestamp)
}

// Write currentJars
```

```
dataOut.writeInt(currentJars.size)
for ((name, timestamp) <- currentJars) {
    dataOut.writeUTF(name)
    dataOut.writeLong(timestamp)
}

// Write the task itself and finish
dataOut.flush()
val taskBytes = serializer.serialize(task).array()
out.write(taskBytes)
ByteBuffer.wrap(out.toByteArray)
```

在 Executor 端，恢复 Task 的依赖和 Task 本身：

```
val in = new ByteBufferInputStream(serializedTask)

val dataIn = new DataInputStream(in)

// Read task's files
val taskFiles = new HashMap[String, Long]()
val numFiles = dataIn.readInt()
for (i <- 0 until numFiles) {
    taskFiles(dataIn.readUTF()) = dataIn.readLong()
}

// Read task's JARs
val taskJars = new HashMap[String, Long]()
val numJars = dataIn.readInt()
for (i <- 0 until numJars) {
    taskJars(dataIn.readUTF()) = dataIn.readLong()
}
// Create a sub-buffer for the rest of the data, which is the serialized Task object
val subBuffer = serializedTask.slice()   // ByteBufferInputStream will have
   read just up to task
(taskFiles, taskJars, subBuffer)
```

从序列化的 Task 中获取依赖的文件和依赖的位置信息后，org.apache.spark.executor.Executor 会调用 updateDependencies 下载这些依赖：

```
Utils.fetchFile(name, new File(SparkFiles.getRootDirectory), conf,env.
    securityManager, hadoopConf, timestamp, useCache = !isLocal)
```

依赖下载完成后，Executor 就具备执行 Task 的能力了，接下来，Executor 会通过 TaskRunner 执行任务。

6.2.2　任务执行

在准备好了 Task 的运行时环境后，Task 就会执行：

```
// 开始执行任务，并且记录任务的耗时情况
taskStart = System.currentTimeMillis()
val value = task.run(taskId.toInt) //执行Task，计算结果存入value
val taskFinish = System.currentTimeMillis()
// 如果任务已经被杀掉，那么直接失败返回
if (task.killed) {
    throw new TaskKilledException
}
```

task.run(taskId.toInt) 就是执行最终的计算，它首先为本次计算创建一个 TaskContext。TaskContext 会保留该 Task 的上下文，而且这个 TaskContext 是一个 ThreadLocal 变量：

```
private static ThreadLocal<TaskContext> taskContext =
    new ThreadLocal<TaskContext>();
```

通过 setTaskContext 可以设置这个 Context 为特定 Thread 的上下文：

```
static void setTaskContext(TaskContext tc) {
    taskContext.set(tc);
}
```

而通过 get() 可以获取这个上下文：

```
public static TaskContext get() {
    return taskContext.get();
}
```

比如在 org.apache.spark.rdd.PairRDDFunctions#combineByKey 中会通过它来获取当前 Task 的上下文。

除此之外，TaskContext 还有一个重要的设置是用户可以设置 Task 结束时的回调函数，Task 会在结束时调用这个回调函数来完成最终的处理。目前推荐使用下面两种方式来完成这个回调函数的注册：

```
public abstract TaskContext addTaskCompletionListener(TaskCompletionListener
    listener);
public abstract TaskContext addTaskCompletionListener(final Function1
```

```
    <TaskContext, Unit> f);
```

而之前的方式已经被废弃（deprecated）了，不再推荐使用：

```
public abstract void addOnCompleteCallback(final Function0<Unit> f);
```

org.apache.spark.TaskContextImpl 就是 TaskContext 的一个具体实现。addOnCompleteCallback 其中的一个实现如下：

```
override def addTaskCompletionListener(f: TaskContext => Unit): this.type = {
    onCompleteCallbacks += new TaskCompletionListener {
    override def onTaskCompletion(context: TaskContext): Unit = f(context)
    }
    this
}
```

在 Task 执行结束时，会通过调用 org.apache.spark.TaskContextImpl#markTaskCompleted 来调用回调函数：

```
completed = true //标记Task完成为true
val errorMsgs = new ArrayBuffer[String](2) //记录错误信息
// Process complete callbacks in the reverse order of registration
onCompleteCallbacks.reverse.foreach { listener =>
    try {
        listener.onTaskCompletion(this) //执行回调函数
    } catch {
        case e: Throwable =>
                errorMsgs += e.getMessage //发生异常，记录错误信息
                logError("Error in TaskCompletionListener", e)
        }
    }
    if (errorMsgs.nonEmpty) {//如果错误信息不为空，那么抛出异常
        throw new TaskCompletionListenerException(errorMsgs)
}
```

它的核心实现如下：

```
// 为本次的任务生成TaskContext
context = new TaskContextImpl(stageId, partitionId, attemptId, runningLocally =
    false)
//设置上下文信息，实际上会调用org.apache.spark.TaskContext#setTaskContext
    TaskContextHelper.setTaskContext(context)
context.taskMetrics.hostname = Utils.localHostName() //更新metrics信息
```

```
//当前线程，在被打断的时候可以通过它来停止该线程
    taskThread = Thread.currentThread()
    if (_killed) {//如果当前Task被杀死，那么需要退出Task的执行
        kill(interruptThread = false)
    }
    try {
        runTask(context) //执行本次Task
} finally {
    //任务结束，执行任务结束时的回调函数
        context.markTaskCompleted()
        TaskContextHelper.unset()
}
```

runTask(context) 在不同的 Task 会有不同的实现。现在有两种 Task：

1）org.apache.spark.scheduler.ResultTask

对于最后一个 Stage，会根据生成结果的 Partition 来生成与 Partition 数量相同的 ResultTask。然后，ResultTask 会将计算的结果汇报到 Driver 端。具体实现如下：

```
//获取用于反序列化的实例
val ser = SparkEnv.get.closureSerializer.newInstance()
//获取rdd和作用于rdd结果的函数
val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)](
    ByteBuffer.wrap(taskBinary.value),
    Thread.currentThread.getContextClassLoader)
//Task的metrics信息
metrics = Some(context.taskMetrics)
//调用rdd.iterator执行rdd上的计算。详细可以参阅第3章。
func(context, rdd.iterator(partition, context))
```

2）org.apache.spark.scheduler.ShuffleMapTask

对于非最后的 Stage，会根据每个 Stage 的 Partition 数量来生成 ShuffleMapTask。ShuffleMapTask 会根据下游 Task 的 Partition 数量和 Shuffle 的策略来生成一系列文件。它的计算过程如下：

a）val manager = SparkEnv.get.shuffleManager

从 SparkEnv 中获得 Shuffle Manager，现在 Spark 支持 Hash 和 Sort Based Shuffle，是内置的功能；此外 Spark 还支持额外的 Shuffle Service。用户可以通过实现几个类就

可以使用自定义的 Shuffle Service。

b）writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)

从 Shuffle Manager 里取得 Writer，对于 Hash Based Shuffle 来说，writer 就是 org.apache.spark.shuffle.hash.HashShuffleWriter；对于 Sort Based Shuffle 来说，writer 就是 org.apache.spark.shuffle.sort.SortShuffleWriter。

c）writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_<:Product2[Any, Any]]])

调用 rdd 开始计算，并且将计算结果通过 writer 写入文件系统。

d）return writer.stop(success = true).get

关闭 writer，并将计算结果返回。详情请参阅第 7 章。

6.2.3 任务结果的处理

在 Executor 运行 Task 时，得到的计算结果会存入 org.apache.spark.scheduler.DirectTaskResult。

```
// value中保存Task的计算结果
val value = task.run(taskId.toInt)
val resultSer = env.serializer.newInstance()              //获得序列化工具
val valueBytes = resultSer.serialize(value)               //序列化结果
//首先将结果直接放入org.apache.spark.scheduler.DirectTaskResult
val directResult = new DirectTaskResult(valueBytes, accumUpdates, task.
metrics.orNull)
val ser = env.closureSerializer.newInstance()
val serializedDirectResult = ser.serialize(directResult) //序列化结果
val resultSize = serializedDirectResult.limit            //序列化结果的大小
```

但是这个 serializedDirectResult 并不是直接回传到 Driver。在将结果回传到 Driver 时，会根据结果的大小使用不同的策略：

1）如果结果大于 1GB，那么直接丢弃这个结果。这个是 Spark 1.2 中新加入的策略。可以通过 spark.driver.maxResultSize 来进行设置。

2）对于“较大”的结果，将其以 taskId 为 key 存入 org.apache.spark.storage.

BlockManager；如果结果不大，则直接回传给 Driver。那么如何判定这个阈值呢？这里的回传是直接通过 AKKA 的消息传递机制。因此结果首先不能超过该机制设置的消息的最大值。这个最大值是通过 spark.akka.frameSize 设置的，单位是 MByte，默认值是 10MB。除此之外，还有 200KB 的预留空间。因此这个阈值就是 conf.getInt("spark.akka.frameSize", 10) * 1024 *1024–200*1024。

3）其他情况则直接通过 AKKA 回传到 Driver。

实现源码解析如下：

```
val serializedResult = {
    if (maxResultSize > 0 && resultSize > maxResultSize) {
    // 如果结果的大小大于1GB，那么直接丢弃，可以在spark.driver.maxResultSize设置
    ser.serialize(new IndirectTaskResult[Any](TaskResultBlockId(taskId),
        resultSize))
    } else if (resultSize >= akkaFrameSize - AkkaUtils.reservedSizeBytes) {
    // 如果不能通过Akka的消息传递，那么放入BlockManager等待调用者以网络的形式来获取。
    // Akka的消息的默认大小可以通过spark.akka.frameSize来设置，默认值是10MB
        val blockId = TaskResultBlockId(taskId)
        env.blockManager.putBytes(
            blockId, serializedDirectResult, StorageLevel.MEMORY_AND_DISK_SER)
        ser.serialize(new IndirectTaskResult[Any](blockId, resultSize))
    } else {
        //结果可以直接回传到Driver
        serializedDirectResult
    }
}
// 通过Akka向Driver汇报本次Task的已经完成
execBackend.statusUpdate(taskId, TaskState.FINISHED, serializedResult)
```

而 execBackend 是 org.apache.spark.executor.ExecutorBackend 的一个实例，它实际上是 Executor 与 Driver 通信的接口：

```
private[spark] trait ExecutorBackend {
    def statusUpdate(taskId: Long, state: TaskState, data: ByteBuffer)
}
```

TaskRunner 会将 Task 执行的状态汇报给 Driver（org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend.DriverActor）。而 Driver 会转给 org.apache.spark.scheduler.TaskSchedulerImpl#statusUpdate。

6.2.4 Driver 端的处理

TaskRunner 将 Task 的执行状态汇报给 Driver 后，Driver 会转给 org.apache.spark.scheduler.TaskSchedulerImpl#statusUpdate。在这里，不同的状态有不同的处理：

如果类型是 TaskState.FINISHED，那么调用 org.apache.spark.scheduler.TaskResultGetter#enqueueSuccessfulTask 进行处理。

如果类型是 TaskState.FAILED 或者 TaskState.KILLED 或者 TaskState.LOST，调用 org.apache.spark.scheduler.TaskResultGetter#enqueueFailedTask 进行处理。对于 TaskState.LOST，还需要将其所在的 Executor 标记为 FAILED，并且根据更新后的 Executor 重新调度。

enqueueSuccessfulTask 的逻辑也比较简单，即如果是 IndirectTaskResult，那么需要通过 blockId 来获取结果：sparkEnv.blockManager.getRemoteBytes(blockId)；如果是 DirectTaskResult，那么结果就无需远程获取了。然后调用以下调用栈进行处理。核心逻辑是第 5 个调用栈。

```
1）org.apache.spark.scheduler.TaskSchedulerImpl#handleSuccessfulTask
2）org.apache.spark.scheduler.TaskSetManager#handleSuccessfulTask
3）org.apache.spark.scheduler.DAGScheduler#taskEnded
4）org.apache.spark.scheduler.DAGScheduler#eventProcessActor
5）org.apache.spark.scheduler.DAGScheduler#handleTaskCompletion
```

如果 Task 是 ShuffleMapTask，那么它需要将结果通过某种机制告诉下游的 Stage，以便其可以作为下游 Stage 的输入。这个机制是怎么实现的？

实际上，对于 ShuffleMapTask 来说，其结果实际上是 org.apache.spark.scheduler.MapStatus；序列化后存入 DirectTaskResult 或者 IndirectTaskResult 中。而 DAGScheduler#handleTaskCompletion 通过下面的方式来获取这个结果：

```
val status =event.result.asInstanceOf[MapStatus]
```

通过将这个 status 注册到 org.apache.spark.MapOutputTrackerMaster，就完成了结果处理的过程：

```
mapOutputTracker.registerMapOutputs(
```

```
stage.shuffleDep.get.shuffleId,
stage.outputLocs.map(list =>
     if (list.isEmpty) null else list.head).toArray,
changeEpoch = true)
```

而 registerMapOutputs 的处理也很简单，以 shuffleId 为 key 将 MapStatus 的列表存入带有时间戳的 HashMap：TimeStampedHashMap[Int, Array[MapStatus]]()。如果设置了 cleanup 的函数，那么这个 HashMap 会将超过一定时间（TTL，Time to Live）的数据清理掉。

6.3 参数设置

6.3.1 spark.executor.memory

它配置了 Executor 可以最多使用的内存大小，是通过设置 Executor 的 JVM 的 Heap 尺寸来实现的。由于一个集群的内存资源总归是有限的，而且这些内存会被很多的应用共享，因此，设置一个合理的内存值是非常必要的。如果设置的过大，那么可能会导致部分任务由于分配不到资源而等待，延长整个应用的执行时间；如果设置过小，那么就会产生频繁的垃圾回收和读写外部磁盘，同样影响性能。

Executor 的内存是被其内部所有的任务共享，而每个 Executor 上可以支持的任务的数量取决于 Executor 所持有的 CPU Core 的数量。因此为了评估一个 Executor 占用多少内存是合适的，需要了解每个任务的数据规模的大小和计算过程中所需要的临时内存空间的大小。实际上，要比较精确地计算出一个任务所需要的内存空间还是非常困难的，首先，因为数据本身加载到内存中，由于有管理这些内存的额外内存开销，可能需要的真实内存是数据大小的数倍；其次，任务计算过程中所需要的临时内存空间的大小会因为算法的不同而不同，这是比较难评估的。如果需要比较准确的评估数据集的大小的话，可以将 RDD cache 在内存中，从 BlockManager 的日志中可以看到每个 Cache 分区的大小（实际上，这个大小也是一个估计值）。

如果内存比较紧张，就需要合理规划每个分区任务的数据规模，例如采用更多的分区，用增加任务数量（进而需要更多的批次来运算所有的任务）的方式来减小每个

任务所需处理的数据大小。

6.3.2 日志相关

如果 spark.eventLog.enabled 设置为 true（默认为 false），那么需要设置日志写入的目录 spark.eventLog.dir，这样，日志就可以保存到本地，方便调试和问题追踪。但是随着时间推移，节点的日志必须需要一个清除机制，否则日志很容易写满磁盘。通过表 6-1 的设置，可以设置日志清理的策略。

表 6-1 日志相关的配置选项

配 置 项	默 认 值	说 明
spark.executor.logs.rolling.strategy	none	设置 Executor 日志回滚的策略。默认这个功能是关闭的。现在支持两种形式：基于时间（设置为“time”）的回滚和基于文件大小（设置为“size”）的回滚。对于基于时间的回滚，可以通过 spark.executor.logs.rolling.time.interval 来设置回滚的时间间隔。对于基于文件大小的回滚，可以通过设置 spark.executor.logs.rolling.size.maxBytes 来设置回滚文件的大小
spark.executor.logs.rolling.time.interval	daily	设置 Executor 日志回滚的时间间隔。有效地设置包括： daily：按照天级回滚。 hourly：按照小时级回滚。 minutely：按照分钟级回滚。 有效的数字：单位是秒，按照设置的秒数进行回滚。 建议同时设置 spark.executor.logs.rolling.maxRetainedFiles 以清除老的文件
spark.executor.logs.rolling.size.maxBytes	none	设置日志文件的最大值（单位是 Byte）。超过此数值时，日志文件将会被分割成两个文件。建议同时设置 spark.executor.logs.rolling.maxRetainedFiles 以清除老的文件
spark.executor.logs.rolling.maxRetainedFiles	none	设置系统保留的最大日志文件的数量。多于设置数量的日志文件按照生成时间（最早生成的）会被清除

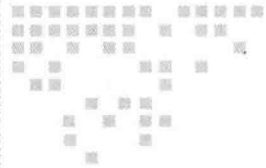
6.3.3 spark.executor.heartbeatInterval

Executor 和 Driver 之间心跳的间隔，单位是毫秒。心跳主要是 Executor 向 Driver 汇报运行状态和 Executor 上报 Task 的统计信息（metric）。这个数值一般无须专门设置。

6.4　小结

Executor 模块负责运行 Task 计算任务，并将计算结果回传到 Driver。Spark 支持多种资源调度框架，这些资源框架在为计算任务分配了资源后，最后都会使用 Executor 模块完成最终的计算。

本章以 Standalone 模式为例，解析用户提交的应用是如何分配到 Executor 资源的，接着详细介绍了 Task 在 Executor 执行的实现细节。



Chapter 7 第 7 章

Shuffle 模块详解

Shuffle，无疑是性能调优的一个重点，本章将从源码实现的角度，深入解析 Spark Shuffle 的实现细节；并且根据 Shuffle 实现迭代的历史，了解 Spark 的 Shuffle 如何解决遇到的问题，这个对于互联网工程实践也具有很重要的指导意义：一个可用、可上线与完美的方案是有区别的，与其自己在线下不断“优化”，不如将一个能够满足当前需要的实现先上线，并且根据需求或者应用场景的变化不断进行迭代、改进。这样，一个系统或者功能就会慢慢地发展起来。

Shuffle，翻译成中文就是洗牌。之所以需要 Shuffle，还是因为具有某种共同特征的一类数据需要最终汇聚（aggregate）到一个计算节点上进行计算。这些数据分布在各个存储节点上并且由不同节点的计算单元处理。以最简单的 Word Count 为例，其中数据保存在 Node1、Node2 和 Node3；经过处理后，这些数据最终会汇聚到 Nodea、Nodeb 处理，如图 7-1 所示。

这个数据重新打乱然后汇聚到不同节点的过程就是 Shuffle。但是实际上，Shuffle 过程可能会非常复杂：

1）数据量会很大，比如单位为 TB 或 PB 的数据分散到几百甚至数千、数万台机器上。

2）为了将这个数据汇聚到正确的节点，需要将这些数据放入正确的 Partition，因为数据大小已经大于节点的内存，因此这个过程中可能会发生多次硬盘续写。

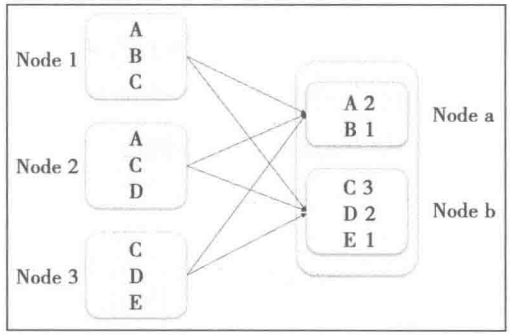


图 7-1　Word Count 的 Shuffle 示意图

3）为了节省带宽，这个数据可能需要压缩，如何在压缩率和压缩解压时间中间做一个比较好的选择？

4）数据需要通过网络传输，因此数据的序列化和发序列化也变得相对复杂。

一般来说，每个 Task 处理的数据可以完全载入内存（如果不能，可以减小每个 Partition 的大小），因此 Task 可以做到在内存中计算。除非非常复杂的计算逻辑，否则为了容错而持久化中间的数据是没有太大收益的，毕竟中间某个过程出错了可以从头开始计算。但是对于 Shuffle 来说，如果不持久化这个中间结果，一旦数据丢失，就需要重新计算依赖的全部 RDD，因此有必要持久化这个中间结果。

接下来会深入分析每个 Shuffle Map Task 结束时，数据是如何持久化（即 Shuffle Write）以使得下游的 Task 可以获取到其需要处理的数据的（即 Shuffle Read）。最后展望 Shuffle 接下来可能会有的改进。需要注意的是在 Spark 0.8 以后，Shuffle Write 会将数据持久化到硬盘，虽然之后 Shuffle Write 不断进行演进优化，但是数据落地到本地文件系统的实现并没有改变。

7.1　Hash Based Shuffle Write

在 Spark 1.0 以前，Spark 只支持 Hash Based Shuffle。因为在很多运算场景中并

不需要排序，因此多余的排序只能使性能变差，比如 Hadoop 的 Map Reduce 就是这么实现的，也就是 Reducer 拿到的数据都是已经排好序的。实际上 Spark 的实现很简单：每个 Shuffle Map Task 根据 key 的哈希值，计算出每个 key 需要写入的 Partition 然后将数据单独写入一个文件，这个 Partition 实际上就对应了下游的一个 Shuffle Map Task 或者 Result Task。因此下游的 Task 在计算时会通过网络（如果该 Task 与上游的 Shuffle Map Task 运行在同一个节点上，那么此时就是一个本地的硬盘读写）读取这个文件并进行计算。

7.1.1 Basic Shuffle Writer 实现解析

在 Executor 上执行 Shuffle Map Task 时，最终会调用 org.apache.spark.scheduler.ShuffleMapTask 的 runTask。核心逻辑：

```
val manager = SparkEnv.get.shuffleManager
writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)
writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_<:Produ
    ct2[Any, Any]]])
return writer.stop(success = true).get
```

这个逻辑比较简单，总结如下：

1）从 SparkEnv 中获得 shuffleManager，就如前面提到的，Spark 除了支持 Hash 和 Sort Based Shuffle 外，还支持 external 的 Shuffle Service。用户可以通过实现几个类就可以使用自定义的 Shuffle。

2）从 manager 里取得 Writer，在这里获得的是 org.apache.spark.shuffle.hash.HashShuffleWriter。

3）调用 rdd 开始运算，运算结果通过 Writer 进行持久化，逻辑在 org.apache.spark.shuffle.hash.HashShuffleWriter#write。开始时通过 org.apache.spark.ShuffleDependency 是否定义了 org.apache.spark.Aggregator 来确定是否需要做 Map 端的聚合。然后将原始结果或者聚合后的结果通过 org.apache.spark.shuffle.FileShuffleBlockManager#forMapTask 的方法写入。写入完成后，会将元数据信息写入 org.apache.spark.scheduler.MapStatus。然后下游的 Task 可以通过这个 MapStatus 取得需要处理的数据。

下面我们来看一下步骤 3 的具体实现。首先 ShuffledRDD 会调用 org.apache.spark.shuffle.hash.HashShuffleWriter#write 进行写操作：

```
val iter = if (dep.aggregator.isDefined) {
    if (dep.mapSideCombine) { //如果需要做聚合，那么将聚合records
        dep.aggregator.get.combineValuesByKey(records, context)
    } else {//此处省略一些非关键代码
        records
    }

    for (elem <- iter) {
    val bucketId = dep.partitioner.getPartition(elem._1)//获得该element需要写
       入的partition
    // 实际调用org.apache.spark.shuffle.FileShuffleBlockManager.forMapTask进行写入
    shuffle.writers(bucketId).write(elem)
}
```

而 org.apache.spark.shuffle.FileShuffleBlockManager#forMapTask #writers 的定义：

```
    val writers: Array[BlockObjectWriter] = if (consolidateShuffleFiles) {
        //此处省略与本部分无关代码
    } else {
        Array.tabulate[BlockObjectWriter](numBuckets) { bucketId =>
            val blockId = ShuffleBlockId(shuffleId, mapId, bucketId)
            val blockFile = blockManager.diskBlockManager.getFile(blockId)
            //如果blockFile已经存在，那么删除它并打印日志。代码已经忽略
            blockManager.getDiskWriter(blockId, blockFile, serializer,
                bufferSize, writeMetrics)
    }
}
```

writers 通过 org.apache.spark.ShuffleDependency#partitioner#numPartitions 来获得下游 Partition 的数量；而通过前文的分析我们知道，这个 Partition 的数量实际与下游 Task 的数量相对应：从运行时角度来看，每个 Partition 对应一个 Task，从而形成流水线来高效并发地处理数据。

下游的每个 Partition 都会对应于一个文件，文件名字的格式就是 "shuffle_" + shuffleId + "_" + mapId + "_" + reduceId。其中 reduceId 和 Partition 的数量对应，比如有 1000 个 Partition，那么这个 reduceId 就是 0 ~ 999。blockManager.getDiskWriter 为每个文件创建一个 org.apache.spark.storage.DiskBlockObjectWriter，DiskBlockObjectWriter 可以直接向一个文件写入数据，如果文件已经存在，那么会以追加的方

式写入。由于 blockFile 已经存在的话会被删除，因此这里只会创建一个新的文件。

可以通过图 7-2 来加深上述过程的理解。

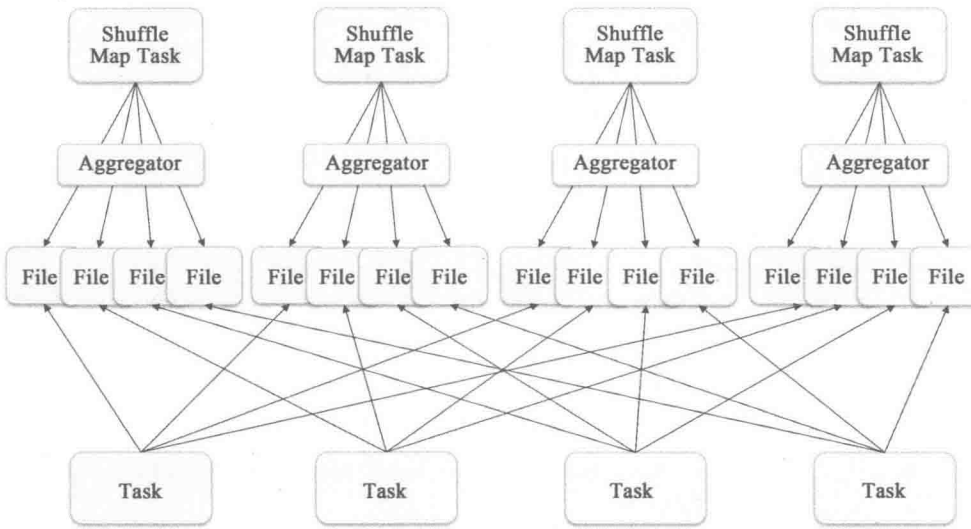


图 7-2 基本的 Hash Based Shuffle Write

7.1.2 存在的问题

由于每个 Shuffle Map Task 需要为每个下游的 Task 创建一个单独的文件，因此文件的数量就是 number(shuffle_map_task) * number(following_task)。如果 Shuffle Map Task 是 1000，下游的 Task 是 500，那么理论上会产生 500 000 个文件（对于 size 为 0 的文件 Spark 有特殊的处理）。生产环境中 Task 的数量实际上会更多，因此这个简单的实现会带来以下问题：

1）每个节点可能会同时打开多个文件，每次打开文件都会占用一定内存。假设每个 Write Handler 的默认需要 100KB 的内存，那么同时打开这些文件需要 50GB 的内存，对于一个集群来说，还是有一定的压力的。尤其是如果 Shuffle Map Task 和下游的 Task 同时增大 10 倍，那么整体的内存就增长到 5TB。

2）从整体的角度来看，打开多个文件对于系统来说意味着随机读，尤其是每个文

件比较小但是数量非常多的情况。而现在机械硬盘在随机读方面的性能特别差，非常容易成为性能的瓶颈。如果集群依赖的是固态硬盘，也许情况会改善很多，但是随机写的性能肯定不如顺序写的。

7.1.3　Shuffle Consolidate Writer

为了解决 Shuffle 过程产生文件过多的问题，在 Spark 0.8.1 中加入了 Shuffle Consolidate Files 机制。它的主要目标是减少 Shuffle 过程产生的文件。若使用这个功能，则需要将 spark.shuffle.consolidateFiles 设置为 true。在 Spark 1.2.0 中，这个功能还是没有成为默认选项。官方给出的说法是这个功能还欠稳定。接下来介绍这个机制的实现原理。图 7-3 描述了 Shuffle Consolidate Files 的实现细节。

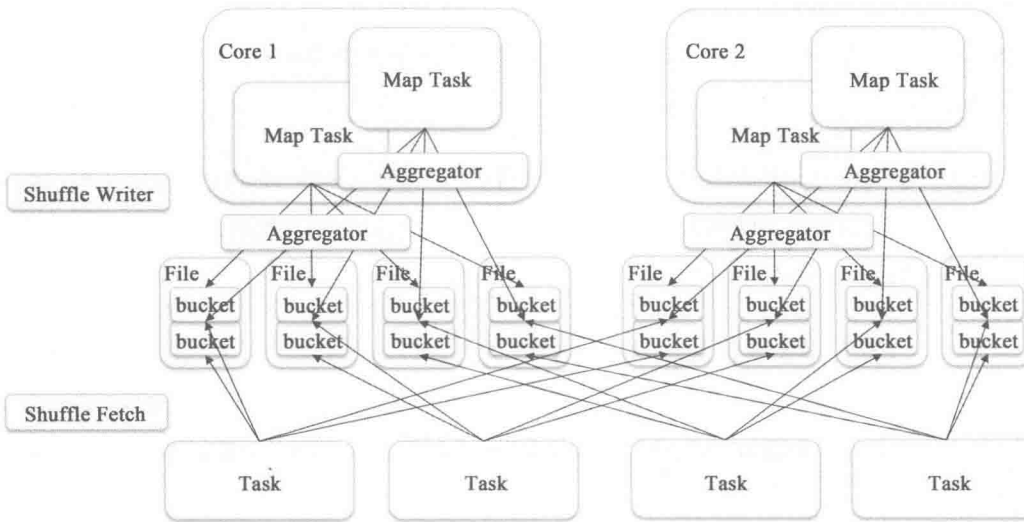


图 7-3　Shuffle Consolidate Files

对于运行在同一个 Core 的 Shuffle Map Task，第一个 Shuffle Map Task 会创建一个文件；之后的就会将数据追加到这个文件上而不是新建一个文件。因此文件数量就从 number(shuffle_map_task)*number(following_task) 变成了 number(cores)*number(following_task)。当然，如果每个 Core 都只运行一个 Shuffle Map Task，那么就和原来的机制一样了。但是 Shuffle Map Task 明显多于 Core 数量或者说每个 Core 都会运

行多个 Shuffle Map Task，所以这个实现能够显著减少文件的数量。

而实现的细节就是不同的 org.apache.spark.shuffle.FileShuffleBlockManager#forMapTask #writers 的实现：

```
val writers: Array[BlockObjectWriter] = if (consolidateShuffleFiles) {
    fileGroup = getUnusedFileGroup() //获得没有使用的FileGroup
    Array.tabulate[BlockObjectWriter](numBuckets) { bucketId =>
        val blockId = ShuffleBlockId(shuffleId, mapId, bucketId)
        blockManager.getDiskWriter(blockId, fileGroup(bucketId), serializer,
            bufferSize,
        writeMetrics)
    }
} else { // Basic Shuffle Writer的实现
```

org.apache.spark.shuffle.FileShuffleBlockManager.ShuffleFileGroup 可以理解成一个文件组，这个文件组的每个文件都对应一个 Partition 或者下游的 Task。因此对第一个 Shuffle Map Task 来说，它创建了一个文件；而接下来的 Shuffle Map Task 都是以追加的方式写这个文件。

blockManager.getDiskWriter 为每个文件创建一个 org.apache.spark.storage.DiskBlockObjectWriter，DiskBlockObjectWriter 可以直接向一个文件写入数据，如果文件已经存在那么会以追加的方式写入。

但是下游的 Task 如何区分文件不同的部分呢？在同一个 Core 上运行 Shuffle Map Task 相当于写了这个文件的不同的部分。答案就在 org.apache.spark.shuffle.FileShuffleBlockManager.ShuffleFileGroup#getFileSegmentFor。这将在 7.5 节详细解析。

7.1.4 小结

Shuffle Consolidate 的机制虽然在某些场景下缓解了 Shuffle 过程产生文件过多的问题，但是还没有彻底解决 7.1.2 节中提出的两个问题，即同时打开多个文件造成 Writer Handler 内存使用过大的问题和产生多个文件造成的随机读从而影响性能的问题。实际上，通过上面的分析，要解决 Basic Shuffle Write 的问题还是比较困难。毕竟对于一个已经在很多生产环境中使用的平台，推倒重来要冒很大的风险；其次对于一

个通用的平台，任何的性能测试都只是针对一部分场景，谁也不能保证每个“优化”、“改进”都会给用户带来正收益等。

Spark 采用了一个很聪明的做法，即建立另外一套 Shuffle 机制，让用户自己选择。Spark 1.0 建立了 Shuffle Pluggable 的框架，通过实现一个接口来很容易地实现第三方的 external Shuffle Service。而 Spark 1.1 实现了 Sort Based Shuffle；而 Spark1.2 里，Sort Based Shuffle 取代 Hash Based Shuffle 成为 Shuffle 的默认选项。

接下来会解析 Shuffle Pluggable 框架和 Sort Based Shuffle 的实现原理。

7.2　Shuffle Pluggable 框架

首先介绍一下需要实现的接口。框架的类图如图 7-4 所示。如果需要实现新的 Shuffle 机制，那么需要实现这些接口。

7.2.1　org.apache.spark.shuffle.ShuffleManager

Driver 和每个 Executor 都会持有一个 ShuffleManager，这个 ShuffleManager 可以通过配置项 spark.shuffle.manager 指定，并且由 SparkEnv 创建。Driver 中的 ShuffleManager 负责注册 Shuffle 的元数据，比如 shuffleId、Map Task 的数量等。Executor 中的 ShuffleManager 则负责读和写 Shuffle 的数据。

需要实现的函数及其功能说明如下：

1）由 Driver 注册元数据信息

```
def registerShuffle[K, V, C](
    shuffleId: Int,
    numMaps: Int,
    dependency: ShuffleDependency[K, V, C]): ShuffleHandle
```

一般如果没有特殊的需求，可以使用下面的实现，实际上 Hash Based Shuffle 和 Sort Based Shuffle 都是这么实现的。

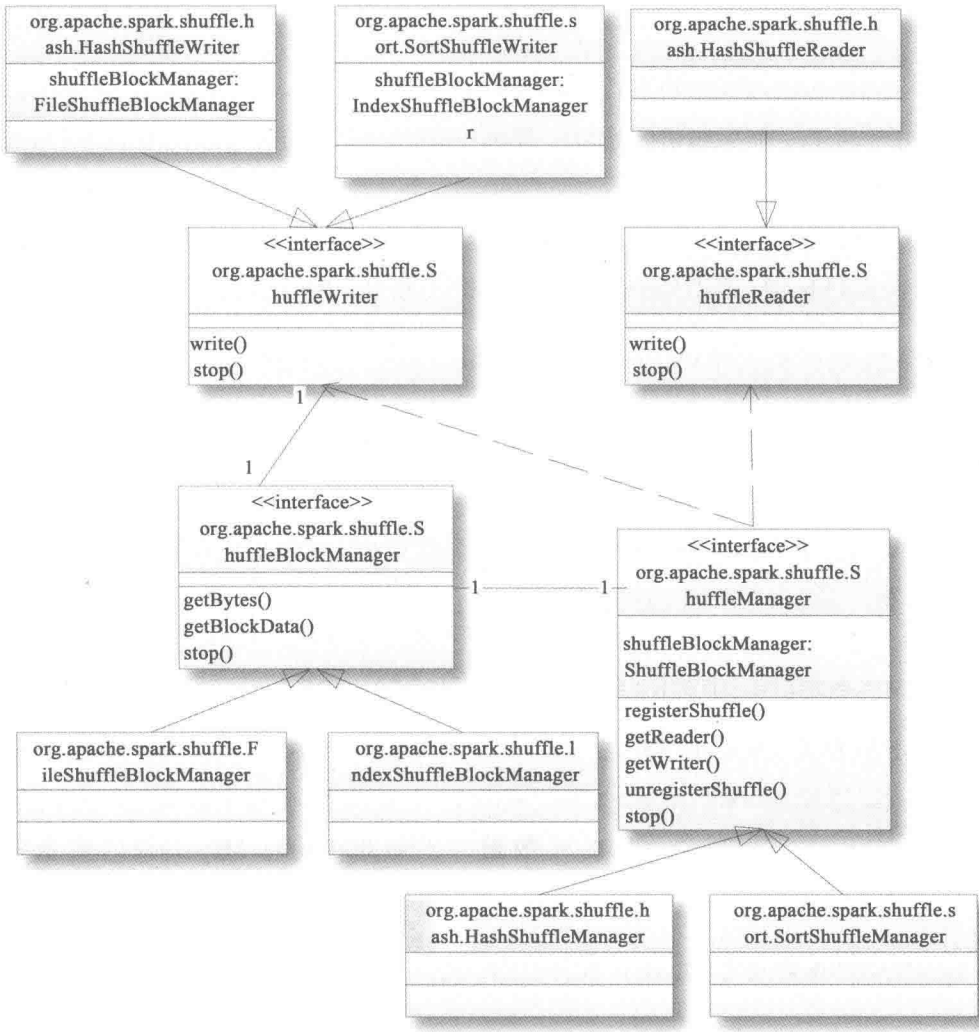


图 7-4 Shuffle Pluggable 框架类图

```
override def registerShuffle[K, V, C](
    shuffleId: Int,
    numMaps: Int,
    dependency: ShuffleDependency[K, V, C]): ShuffleHandle = {
        new BaseShuffleHandle(shuffleId, numMaps, dependency)
}
```

2）获得 Shuffle Writer，根据 Shuffle Map Task 的 ID 为其创建 Shuffle Writer。

```
def getWriter[K, V](handle: ShuffleHandle, mapId: Int, context: TaskContext):
   ShuffleWriter[K, V]
```

3）获得 Shuffle Reader，根据 shuffleId 和 Partition 的 ID 为其创建 Shuffle Reader。

```
def getReader[K, C](
    handle: ShuffleHandle,
    startPartition: Int,
    endPartition: Int,
context: TaskContext): ShuffleReader[K, C]
```

4）为数据成员 shuffleBlockManager 赋值，以保存实际的 ShuffleBlockManager。

5）def unregisterShuffle(shuffleId: Int): Boolean，删除本地的 Shuffle 的元数据。

6）def stop(): Unit，停止 Shuffle Manager。

每个接口的具体实现的例子，可以参照 org.apache.spark.shuffle.sort.SortShuffleManager 和 org.apache.spark.shuffle.hash.HashShuffleManager。

7.2.2　org.apache.spark.shuffle.ShuffleWriter

Shuffle Map Task 通过 ShuffleWriter 将 Shuffle 数据写入本地。这个 Writer 主要通过 ShuffleBlockManager 来写入数据，因此它的功能是比较轻量级的。

1）def write(records: Iterator[_<: Product2[K, V]]): Unit，写入所有的数据。需要注意的是如果需要在 Map 端做聚合，那么写入前需要将 records 做聚合。

2）def stop(success: Boolean): Option[MapStatus]，写入完成后提交本次写入。

对于 Hash Based Shuffle，请查看 org.apache.spark.shuffle.hash.HashShuffleWriter；对于 Sort Based Shuffle，请查看 org.apache.spark.shuffle.sort.SortShuffleWriter。

7.2.3　org.apache.spark.shuffle.ShuffleBlockManager

主要使用从本地读取 Shuffle 数据的功能。这些接口都是通过 org.apache.spark.storage.BlockManager 调用的。

1）def getBytes(blockId: ShuffleBlockId): Option[ByteBuffer]，一般通过调用下一个接口实现，只不过将 ManagedBuffer 转换成了 ByteBuffer。

2）def getBlockData(blockId: ShuffleBlockId): ManagedBuffer，核心读取逻辑。因为不同的实现，文件的组织方式可能是不一样的，比如 Hash Based Shuffle 从本地读取文件都是通过这个接口实现的，比如 Sort Based Shuffle 需要先通过读取 Index 索引文件获得每个 Partition 的起始位置后，才能读取真正的数据文件。

3）def stop(): Unit，停止该 Manager。

对于 Hash Based Shuffle，请查看 org.apache.spark.shuffle.FileShuffleBlockManager；对于 Sort Based Shuffle，请查看 org.apache.spark.shuffle.IndexShuffleBlockManager。

7.2.4 org.apache.spark.shuffle.ShuffleReader

ShuffleReader 实现了下游 Task 如何读取上游 ShuffleMapTask 的 Shuffle 输出的逻辑。这个逻辑比较复杂，简单来说就是通过 org.apache.spark.MapOutputTracker 获得数据的位置信息，如果数据在本地则调用 org.apache.spark.storage.BlockManager 的 getBlockData 读取本地数据（实际上 getBlockData 最终会调用 org.apache.spark.shuffle.ShuffleBlockManager 的 getBlockData）。具体的 Shuffle Read 的逻辑请查看 7.5 节。

7.2.5 如何开发自己的 Shuffle 机制

通过 Hash Based Shuffle 和 Sort Based Shuffle 的源码，可以得出使用 Spark Pluggable 框架开发一个第三方的 Shuffle Service 是比较容易的；这个容易是指功能实现方面。但是这个实现必须要考虑超大规模数据场景下的性能问题以及资源消耗问题。

7.3 Sort Based Write

在 Spark 1.2.0 中，Spark Core 的一个重要的升级就是将默认的 Hash Based Shuffle 换成了 Sort Based Shuffle，即 spark.shuffle.manager 从 Hash 换成了 Sort，对应的实现类分别是 org.apache.spark.shuffle.hash.HashShuffleManager 和 org.apache.spark.shuffle.sort.SortShuffleManager。

这个方式的选择是在 org.apache.spark.SparkEnv 完成的：

```
// Let the user specify short names for shuffle managers
val shortShuffleMgrNames = Map(
    "hash" -> "org.apache.spark.shuffle.hash.HashShuffleManager",
    "sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager")
val shuffleMgrName = conf.get("spark.shuffle.manager", "sort") //获得Shuffle
    // Manager的类型，默认为sort
val shuffleMgrClass = shortShuffleMgrNames.getOrElse(shuffleMgrName.
   toLowerCase, shuffleMgrName)
val shuffleManager = instantiateClass[ShuffleManager](shuffleMgrClass)
```

那么 Sort Based Shuffle“取代”Hash Based Shuffle 作为默认选项的原因是什么？

正如前面提到的，Hash Based Shuffle 的每个 Mapper 都需要为每个 Reducer 写一个文件，供 Reducer 读取，即需要产生 M*R 个数量的文件，如果 Mapper 和 Reducer 的数量比较大，产生的文件数会非常多。Hash Based Shuffle 设计的目标之一就是避免不需要的排序（Hadoop Map Reduce 被人诟病的地方，很多不需要 Sort 的地方的 Sort 导致了不必要的开销）。但是它在处理超大规模数据集的时候，产生了大量的 Disk IO 和内存的消耗，这无疑很影响性能。Hash Based Shuffle 也在不断的优化中，正如前面讲到的 Spark 0.8.1 引入的 File Consolidation 在一定程度上解决了这个问题。为了更好地解决这个问题，Spark 1.1 引入了 Sort Based Shuffle。首先，每个 Shuffle Map Task 不会为每个 Reducer 生成一个单独的文件；相反，它会将所有的结果写到一个文件里，同时会生成一个 Index 文件，Reducer 可以通过这个 Index 文件取得它需要处理的数据。避免产生大量文件的直接收益就是节省了内存的使用和顺序 Disk IO 带来的低延时。节省内存的使用可以减少 GC 的风险和频率。而减少文件的数量可以避免同时写多个文件给系统带来的压力。

实现详解

Shuffle Map Task 会按照 key 相对应的 Partition ID 进行 Sort，其中属于同一个 Partition 的 key 不会 Sort。因为对于不需要 Sort 的操作来说，这个 Sort 是负收益的；要知道之前 Spark 刚开始使用 Hash Based 的 Shuffle 而不是 Sort Based 就是为了避免 Hadoop Map Reduce 对于所有计算都会 Sort 的性能损耗。对于那些需要 Sort 的运算，比如 sortByKey，这个 Sort 在 Spark 1.2.0 里还是由 Reducer 完成的。整个过程

如图 7-5 所示。

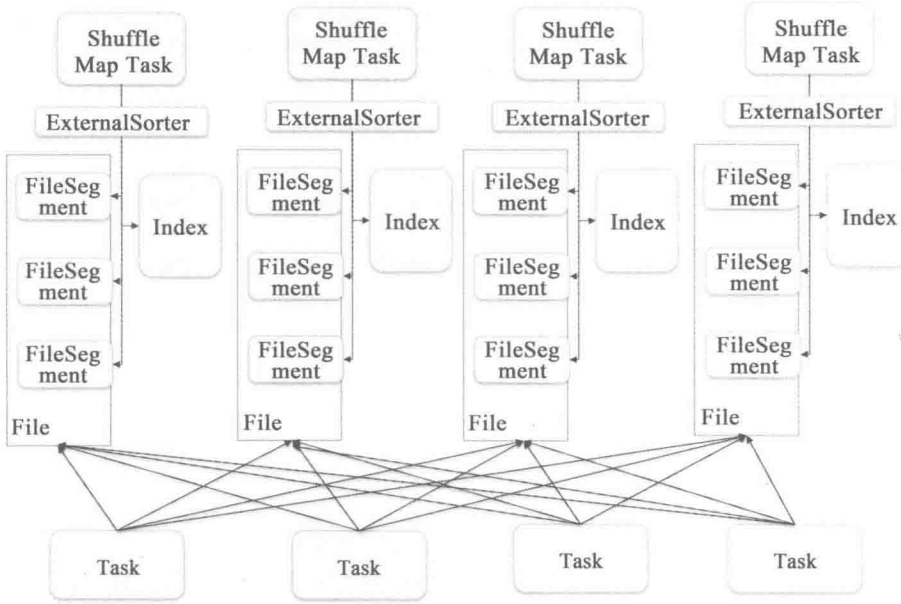


图 7-5 Sort Based Shuffle

如果这个过程内存不够用了，那么这些已经排序的内容会被写入到外部存储。然后在结束的时候将这些不同的文件进行归并排序。

为了便于下游的 Task 获取到其需要的 Partition，这里会生成一个 Index 文件，去记录不同的 Partition 的位置信息。当然，org.apache.spark.storage.BlockManager 也实现了这种新的寻址方式。

核心实现的逻辑都在类 org.apache.spark.shuffle.sort.SortShuffleWriter 和它依赖的类中。下面简要分析它的实现：

1）对于每个 Partition，创建一个 scala.Array 存储它所包含的 key/value 对。每个待处理的 key/value 对都会插入相应的 scala.Array。

2）如果 scala.Array 的大小超过阈值，那么需要将这个内存的数据写入到外部存储。这个文件的开始部分会记录这个 Partition 的 ID 及这个文件保存了多少个数据条目等信息。

3）最后需要将所有写入到外部存储的文件进行归并排序。同时打开的文件不能过多，过多会消耗大量的内存，增加内存溢出（Out of Memory，OOM）或者垃圾回收的风险；也不能过少，过少就会影响性能，增大计算的延时。一般推荐每次同时打开 10 ～ 100 个文件。

4）在生成最后的数据文件时，需要同时生成 Index 索引文件。正如前面提到的，这个索引文件将记录不同 Partition 的起始位置。

当然，你可能还有这样的疑问，Hash Based Shuffle 说白了就是根据 key 需要写入的 org.apache.spark.HashPartitioner，为每个 Reducer 写入单独的 Partition。只不过对于同一个 Core 启动的 Shuffle Map Task 来说，如果选择 spark.shuffle.consolidateFiles，第二个 Shuffle Map Task 会把结果追加到上一个文件中去。那么 Sort 的逻辑完全可以整合到 Hash Based Shuffle 中去，为什么又要重新实现一种 Shuffle Writer 呢？我认为有以下几点原因：

1）Shuffle 机制是所有类似计算模块的核心机制之一，要进行大规模的优化的风险非常高；比如一个看似简单的 Consolidation 机制在 Spark 0.8.1 就引入了，但是 Spark 1.2.0 还是没有将其作为默认选项。

2）Hash Based Shuffle 如果修改为 Sort 的逻辑，所谓的改进可能会影响原来已经稳定的 Spark 应用。比如一个应用在使用 Hash Based Shuffle 时性能是完全符合预期的，那么迁移到 Spark 1.2.0 后，只需要将配置文件修改一下就可以完成这个无缝的迁移。

3）作为一个通用的计算平台，你的测试永远包含不了所有的场景。所以，还是留给用户去选择吧。

4）Sort 机制还处于不断完善的阶段。因此，期待 Sort 在以后的版本中更加完善吧。

接下来从 org.apache.spark.shuffle.sort.SortShuffleManager 开始，介绍 Sort based shuffle 的具体实现。回忆一下在 org.apache.spark.rdd.ShuffledRDD 的计算，它会通过 SparkEnv 的 shuffleManger 取得 Reader：

```
SparkEnv.get.shuffleManager.getReader(dep.shuffleHandle, split.index, split.
    index + 1, context).read().asInstanceOf[Iterator[(K, C)]]
```

如果是 Sort Based Shuffle 的话，那么这个 Shuffle Manager 就是 org.apache.spark.shuffle.sort.SortShuffleManager。其 Reader 实际上就是 Hash Based Shuffle 的 Reader。

7.4 Shuffle Map Task 运算结果的处理

Shuffle Map Task 运算结果的处理分为两部分，一个是在 Executor 端直接处理 Task 结果的；另一个是 Driver 端在接到 Task 运行结束的消息时对 Shuffle Write 的结果进行处理，从而在调度下游的 Task 时，使其可以得到需要的数据。

7.4.1 Executor 端的处理

在解析 Basic Shuffle Writer 时，我们知道 Shuffle Map Task 在 Executor 上运行时，最终会调用 org.apache.spark.scheduler.ShuffleMapTask 的 runTask：

```
override def runTask(context: TaskContext): MapStatus = {
    // 反序列化广播变量taskBinary得到RDD
    val ser = SparkEnv.get.closureSerializer.newInstance()
    val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_, _, _])](
        ByteBuffer.wrap(taskBinary.value), Thread.currentThread.
            getContextClassLoader)
    //省略一些非核心代码
    val manager = SparkEnv.get.shuffleManager //获得Shuffle Manager
    //获得Shuffle Writer
    writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)
    // 首先调用rdd .iterator，如果该RDD已经缓存过或者生成检查点了，那么直接读取结果，否
    // 则开始计算，计算的结果将调用Shuffle Writer写入本地文件系统
    writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <:
        Product2[Any, Any]]])
    // 返回数据的元数据信息，包括location和size
    return writer.stop(success = true).get
```

那么，这个结果最终是如何处理的呢？特别是下游的 Task 是如何获取这些 Shuffle 的数据呢？还要从 Task 是如何开始执行讲起。在 Worker 上接收 Task 执行命令的是 org.apache.spark.executor.CoarseGrainedExecutorBackend。它在接收到 LaunchTask 的命令后，通过在 Driver 创建 SparkContext 时已经创建的 org.apache.spark.executor.Executor 的实例的 launchTask，启动 Task：

```
def launchTask(
    context: ExecutorBackend, taskId: Long, taskName: String,serializedTask:
        ByteBuffer) {
    val tr = new TaskRunner(context, taskId, taskName, serializedTask)
    runningTasks.put(taskId, tr)
    threadPool.execute(tr) // 开始在Executor中运行
}
```

最终 Task 的执行是在 org.apache.spark.executor.Executor.TaskRunner#run。

在 Executor 运行 Task 时，得到的计算结果会存入 org.apache.spark.scheduler.DirectTaskResult。

```
//开始执行Task，最终得到的是org.apache.spark.scheduler.ShuffleMapTask#runTask
//返回的org.apache.spark.scheduler.MapStatus
val value = task.run(taskId.toInt)
val resultSer = env.serializer.newInstance()              //获得序列化工具
val valueBytes = resultSer.serialize(value)               //序列化结果
//首先将结果直接放入org.apache.spark.scheduler.DirectTaskResult
val directResult = new DirectTaskResult(valueBytes, accumUpdates, task.
metrics.orNull)
val ser = env.closureSerializer.newInstance()
val serializedDirectResult = ser.serialize(directResult)//序列化结果
val resultSize = serializedDirectResult.limit             //序列化结果的大小
```

在将结果回传到 Driver 时，会根据结果的大小使用不同的策略（本部分内容在第 6 章的第 2 节也有涉及，但是如果缺失会影响不阅读第 6 章的读者，因此这里加以保留，已经阅读过第 6 章的读者可以跳过）：

1）如果结果大于 1GB，那么直接丢弃这个结果。这个是 Spark 1.2 中新加入的策略。可以通过 spark.driver.maxResultSize 来进行设置。

2）对于“较大”的结果，将其以 taskId 为 key 存入 org.apache.spark.storage.BlockManager；如果结果不大，则直接回传给 Driver。那么如何判定这个阈值呢？这里的回传是直接通过 AKKA 的消息传递机制。因此结果首先不能超过该机制设置的消息的最大值。这个最大值是通过 spark.akka.frameSize 设置的，单位是 MByte，默认值是 10MB。除此之外，还有 200KB 的预留空间。因此这个阈值就是 conf.getInt("spark.akka.frameSize", 10)*1024*024 － 200*1024。

3）其他情况则直接通过 AKKA 回传到 Driver。

实现源码解析如下：

```
val serializedResult = {
    if (maxResultSize > 0 && resultSize > maxResultSize) {
    // 如果结果的大小大于1GB，那么直接丢弃，可以在spark.driver.maxResultSize设置
    ser.serialize(new IndirectTaskResult[Any](TaskResultBlockId(taskId),resu
        ltSize))
    } else if (resultSize >= akkaFrameSize - AkkaUtils.reservedSizeBytes) {
    // 如果不能通过AKKA的消息传递，那么放入BlockManager等待调用者以网络的形式来获取。
    // AKKA的消息的默认大小可以通过spark.akka.frameSize来设置，默认值是10MB
    val blockId = TaskResultBlockId(taskId)
    env.blockManager.putBytes(
    blockId, serializedDirectResult, StorageLevel.MEMORY_AND_DISK_SER)
    ser.serialize(new IndirectTaskResult[Any](blockId, resultSize))
    } else {
        //结果可以直接回传到Driver
        serializedDirectResult
    }
}
// 通过AKKA向Driver汇报本次Task的已经完成
execBackend.statusUpdate(taskId, TaskState.FINISHED, serializedResult)
```

而 execBackend 是 org.apache.spark.executor.ExecutorBackend 的一个实例，它实际上是 Executor 与 Driver 通信的接口：

```
private[spark] trait ExecutorBackend {
  def statusUpdate(taskId: Long, state: TaskState, data: ByteBuffer)
}
```

TaskRunner 会将 Task 执行的状态汇报给 Driver（org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend.DriverActor）。而 Driver 会转给 org.apache.spark.scheduler.TaskSchedulerImpl#statusUpdate。

7.4.2 Driver 端的处理

TaskRunner 将 Task 的执行状态汇报给 Driver 后，Driver 会转给 org.apache.spark.scheduler.TaskSchedulerImpl#statusUpdate。在这里，不同的状态有不同的处理（本部分内容在 6.2.4 节也有涉及，但为了不影响不阅读第 6 章的读者，因此这里的内容加以保留，已经阅读过第 6 章的读者可以跳过）：

如果类型是 TaskState.FINISHED，那么调用 org.apache.spark.scheduler.TaskResultGetter#enqueueSuccessfulTask 进行处理。

如果类型是 TaskState.FAILED 或者 TaskState.KILLED 或者 TaskState.LOST，调用 org.apache.spark.scheduler.TaskResultGetter#enqueueFailedTask 进行处理。对于 TaskState.LOST，还需要将其所在的 Executor 标记为 failed, 并且根据更新后的 Executor 重新调度。

enqueueSuccessfulTask 的逻辑也比较简单，即如果是 IndirectTaskResult，那么需要通过 blockId 来获取结果：sparkEnv.blockManager.getRemoteBytes(blockId)；如果是 DirectTaskResult，那么结果就无需远程获取了。然后调用以下调用栈进行处理。核心逻辑是第 5 个调用栈。

1）org.apache.spark.scheduler.TaskSchedulerImpl#handleSuccessfulTask

2）org.apache.spark.scheduler.TaskSetManager#handleSuccessfulTask

3）org.apache.spark.scheduler.DAGScheduler#taskEnded

4）org.apache.spark.scheduler.DAGScheduler#eventProcessActor

5）org.apache.spark.scheduler.DAGScheduler#handleTaskCompletion

如果 task 是 ShuffleMapTask，那么它需要将结果通过某种机制告诉下游的 Stage，以便于其可以作为下游 Stage 的输入。这个机制是怎么实现的？

实际上，对于 ShuffleMapTask 来说，其结果实际上是 org.apache.spark.scheduler.MapStatus；其序列化后存入了 DirectTaskResult 或者 IndirectTaskResult 中。而 DAGScheduler#handleTaskCompletion 通过下面的方式来获取这个结果：

```
val status =event.result.asInstanceOf[MapStatus]
```

通过将这个 status 注册到 org.apache.spark.MapOutputTrackerMaster，就完成了结果处理的过程：

```
mapOutputTracker.registerMapOutputs(
        stage.shuffleDep.get.shuffleId,
        stage.outputLocs.map(list => if (list.isEmpty) null else list.head).toArray,
        changeEpoch = true)
```

而 registerMapOutputs 的处理也很简单，以 shuffleID 为 key 将 MapStatus 的列表存入带有时间戳的 HashMap：TimeStampedHashMap[Int, Array[MapStatus]]()。如果设置了 cleanup 的函数，那么这个 HashMap 会将超过一定时间（TTL，Time to Live）的数据清理掉。

7.5 Shuffle Read

回忆一下，每个 Stage 的上边界，要么需要从外部存储读取数据，要么需要读取上一个 Stage 的输出；而下边界，要么是需要写入本地文件系统（需要 Shuffle），以供 child Stage 读取，要么是最后一个 Stage，需要输出结果。这里的 Stage，在运行时就是可以以流水线的方式运行的一组 Task，除了最后一个 Stage 对应的是 ResultTask，其余的 Stage 对应的都是 Shuffle Map Task。

而除了需要从外部存储读取数据和 RDD 已经做过 cache 或者 checkpoint 的 Task，一般 Task 都是从 ShuffledRDD 的 Shuffle Read 开始的。本节将详细讲解 Shuffle Read 的过程。

7.5.1 整体流程

Shuffle Read 的整体架构如图 7-6 所示。org.apache.spark.rdd.ShuffledRDD# compute 开始，通过调用 org.apache.spark.shuffle.ShuffleManager 的 getReader 方法，获取到 org.apache.spark.shuffle.ShuffleReader，然后调用其 read() 方法进行读取。在 Spark 1.2.0 中，不管是 Hash Based Shuffle 或者是 Sort Based Shuffle，内置的 Shuffle Reader 都是 org.apache.spark.shuffle.hash.HashShuffleReader。核心实现：

```
override def read(): Iterator[Product2[K, C]] = {
    val ser = Serializer.getSerializer(dep.serializer)
    // 获取结果
    val iter = BlockStoreShuffleFetcher.fetch(handle.shuffleId, start-
       Partition,context,ser)
    // 处理结果
```

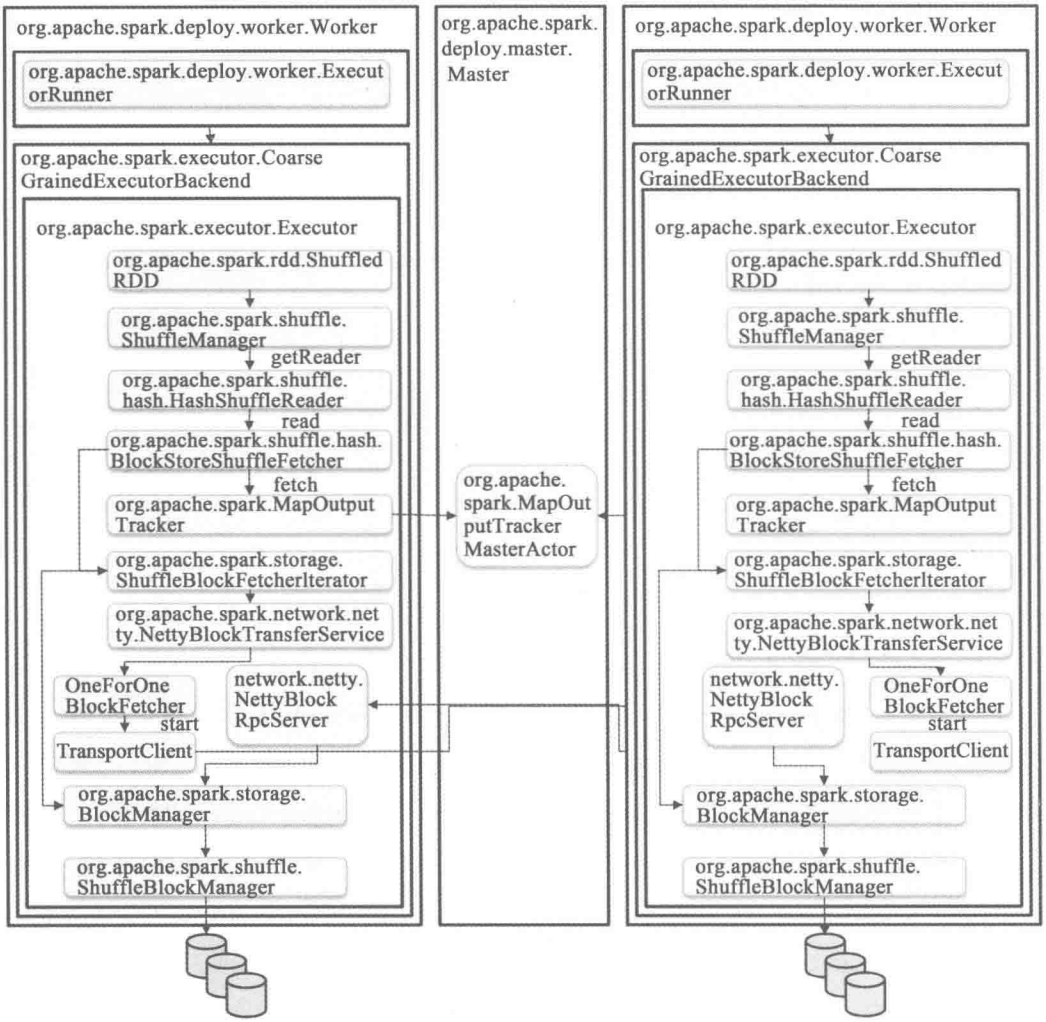


图 7-6　Shuffle Read 的实现

```
val aggregatedIter: Iterator[Product2[K, C]] = if (dep.aggregator.
   isDefined) {//需要聚合
if (dep.mapSideCombine) {//需要mapSide的聚合
    new InterruptibleIterator(context, dep.aggregator.get.combineCombiners-
       ByKey(iter, context))
} else {//只需要Reducer端的聚合
    new InterruptibleIterator(context, dep.aggregator.get.combineValues-
       ByKey(iter, context))
```

```
    }
    } else { // 无需聚合操作
        iter.asInstanceOf[Iterator[Product2[K, C]]].map(pair => (pair._1, pair._2))
    }

    // Sort the output if there is a sort ordering defined.
    dep.keyOrdering match {//判断是否需要排序
        case Some(keyOrd: Ordering[K]) => //对于需要排序的情况
        // 使用ExternalSorter进行排序，注意如果spark.shuffle.spill是false，那么数
        // 据是不会写入到硬盘的
        val sorter = new ExternalSorter[K, C, C](ordering = Some(keyOrd),
           serializer = Some(ser))
        sorter.insertAll(aggregatedIter)
        context.taskMetrics.memoryBytesSpilled += sorter.memoryBytesSpilled
        context.taskMetrics.diskBytesSpilled += sorter.diskBytesSpilled
        sorter.iterator
    case None => //无需排序
        aggregatedIter
    }
}
```

org.apache.spark.shuffle.hash.BlockStoreShuffleFetcher#fetch 会获得数据，它首先会通过 org.apache.spark.MapOutputTracker#getServerStatuses 来获得数据的 meta 信息，这个过程有可能需要向 org.apache.spark.MapOutputTrackerMasterActor 发送读请求，这个读请求是在 org.apache.spark.MapOutputTracker#askTracker 发出的。在获得了数据的 meta 信息后，它会将这些数据存入 Seq[(BlockManagerId, Seq[(BlockId, Long)])] 中，然后调用 org.apache.spark.storage.ShuffleBlockFetcherIterator 最终发起请求。ShuffleBlockFetcherIterator 根据数据的本地性原则进行数据获取。如果数据在本地，那么会调用 org.apache.spark.storage.BlockManager#getBlockData 进行本地数据块的读取。而对于 shuffle 类型的数据，会调用 ShuffleManager 的 ShuffleBlockManager 的 getBlockData。

如果数据在其他的 Executor 上，若用户使用的 spark.shuffle.blockTransferService 是 netty，那么就会通过 org.apache.spark.network.netty.NettyBlockTransferService#fetchBlocks 获取；如果使用的是 nio，那么就会通过 org.apache.spark.network.nio.NioBlockTransferService#fetchBlocks 获取。（注：netty 和 nio 是两种远程读取方式，详细内容见 7.5.4 节。）

7.5.2　数据读取策略的划分

org.apache.spark.storage.ShuffleBlockFetcherIterator 会通过 splitLocalRemoteBlocks 划分数据的读取策略：如果数据在本地，那么可以直接从 BlockManager 中获取；如果需要从其他的节点上获取，则需要通过网络。由于 Shuffle 的数据量可能会很大，因此这里的网络读取分为以下几种策略：

1）每次最多启动 5 个线程到最多 5 个节点上读取数据。

2）每次请求的数据大小不会超过 spark.reducer.maxMbInFlight(默认值为 48MB) 的五分之一。这样做的原因有以下几点：

1）避免占用目标机器过多带宽，在千兆网卡为主流的今天，带宽还是比较重要的。如果机器使用的是万兆网卡，那么可以通过设置 spark.reducer.maxMbInFlight 来充分利用带宽。

2）请求数据可以平行化，这样请求数据的时间可以大大减少。请求数据的总时间就是请求中耗时最长的。这样可以缓解一个节点出现网络拥塞时的影响。

主要实现过程如下：

```
private[this] def splitLocalRemoteBlocks(): ArrayBuffer[FetchRequest] = {

    val targetRequestSize = math.max(maxBytesInFlight / 5, 1L)

    val remoteRequests = new ArrayBuffer[FetchRequest]
    for ((address, blockInfos) <- blocksByAddress) {
        if (address.executorId == blockManager.blockManagerId.executorId) {
            // Block在本地，需要过滤大小为0的Block
            localBlocks ++= blockInfos.filter(_._2 != 0).map(_._1)
            numBlocksToFetch += localBlocks.size
        } else { //需要远程获取的Block
            val iterator = blockInfos.iterator
            var curRequestSize = 0L
            var curBlocks = new ArrayBuffer[(BlockId, Long)]
            while (iterator.hasNext) {
                // blockId 是org.apache.spark.storage.ShuffleBlockId，格式：
                // "shuffle_" + shuffleId + "_" + mapId + "_" + reduceId
                val (blockId, size) = iterator.next()
                // 跳过空的Block
                if (size > 0) {
```

```
                    curBlocks += ((blockId, size))
                    remoteBlocks += blockId
                    numBlocksToFetch += 1
                    curRequestSize += size
                }

                if (curRequestSize >= targetRequestSize) {
                    // 当前总的Size已经可以批量放入一次Request中
                    remoteRequests += new FetchRequest(address, curBlocks)
                    curBlocks = new ArrayBuffer[(BlockId, Long)]
                    curRequestSize = 0
                }
            }
            // 剩余的请求组成一次Request
            if (curBlocks.nonEmpty) {
                remoteRequests += new FetchRequest(address, curBlocks)
            }
        }
    }
    remoteRequests
}
```

7.5.3 本地读取

fetchLocalBlocks() 负责本地 Block 的获取。在 splitLocalRemoteBlocks 中，已经将本地的 Block 列表存入了 localBlocks：private[this] val localBlocks = new ArrayBuffer[BlockId]()。具体过程如下：

```
val iter = localBlocks.iterator
while (iter.hasNext) {
    val blockId = iter.next()
    try {
        val buf = blockManager.getBlockData(blockId)
        shuffleMetrics.localBlocksFetched += 1
        buf.retain()
        results.put(new SuccessFetchResult(blockId, 0, buf))
    } catch {
    }
}
```

而 blockManager.getBlockData(blockId) 的实现是：

```
override def getBlockData(blockId: BlockId): ManagedBuffer = {
```

```
    if (blockId.isShuffle) {
shuffleManager.shuffleBlockManager.getBlockData(blockId.asInstance-
    Of[ShuffleBlockId])
    }
```

这就调用了 ShuffleBlockManager 的 getBlockData 方法。在 Shuffle Pluggable 框架中我们介绍了实现一个 Shuffle Service 之一就是要实现 ShuffleBlockManager。

以 Hash Based Shuffle 为例，它的 ShuffleBlockManager 是 org.apache.spark.shuffle.FileShuffleBlockManager。FileShuffleBlockManager 有两种情况，一种是 consolidateFile 的，这种需要根据 Map ID 和 Reduce ID 首先获得 FileGroup 的一个文件，然后根据在文件中的 offset 和 size 来获取需要的数据；如果是没有 consolidateFile，那么根据 Shuffle Block ID 直接读取整个文件就可以。

```
override def getBlockData(blockId: ShuffleBlockId): ManagedBuffer = {
    if (consolidateShuffleFiles) {
        val shuffleState = shuffleStates(blockId.shuffleId)
        val iter = shuffleState.allFileGroups.iterator
    while (iter.hasNext) {
        // 根据Map ID和Reduce ID获取File Segment的信息
        val segmentOpt = iter.next.getFileSegmentFor(blockId.mapId, blockId.
           reduceId)
        if (segmentOpt.isDefined) {
            val segment = segmentOpt.get
            // 根据File Segment的信息，从FileGroup中找到相应的File和Block在文件中
            // 的offset和size
            return new FileSegmentManagedBuffer(
                transportConf, segment.file, segment.offset, segment.length)
            }
        }
        throw new IllegalStateException("Failed to find shuffle block: " + blockId)
    } else {
        val file = blockManager.diskBlockManager.getFile(blockId) //直接获取文
            // 件句柄
        new FileSegmentManagedBuffer(transportConf, file, 0, file.length)
    }
}
```

对于 Sort Based Shuffle，它需要通过索引文件来获得数据块在数据文件中的具体位置信息，从而读取这个数据。

具体实现在 org.apache.spark.shuffle.IndexShuffleBlockManager#getBlockData 中。

```
override def getBlockData(blockId: ShuffleBlockId): ManagedBuffer = {
    // 根据ShuffleID和MapID从org.apache.spark.storage.DiskBlockManager获取索引文件
    val indexFile = getIndexFile(blockId.shuffleId, blockId.mapId)
    val in = new DataInputStream(new FileInputStream(indexFile))
    try {
        ByteStreams.skipFully(in, blockId.reduceId * 8)  //跳到本次Block的数据区
        val offset = in.readLong()                       //数据文件中的开始位置
        val nextOffset = in.readLong()                   //数据文件中的结束位置
        new FileSegmentManagedBuffer(
        transportConf,
        getDataFile(blockId.shuffleId, blockId.mapId),
        offset,
        nextOffset - offset)
    } finally {
        in.close()
    }
}
```

7.5.4 远程读取

现在支持两种远程读取的方式，一种是 netty，一种是 nio，可以通过 spark.shuffle.blockTransferService 来进行设置。

org.apache.spark.storage.ShuffleBlockFetcherIterator#sendRequest 会向远程的节点发起读取 Block 的请求：

```
shuffleClient.fetchBlocks(address.host,address.port,address.executorId,
    blockIds.toArray,
    new BlockFetchingListener {
        override def onBlockFetchSuccess(blockId: String,buf: Managed-
            Buffer): Unit = {
        //请求成功，省略非关键代码
        buf.retain()
        results.put(new SuccessFetchResult(BlockId(blockId), sizeMap
            (blockId), buf))
        }
        override def onBlockFetchFailure(blockId: String, e: Throwable):
            Unit = {
```

```
                results.put(new FailureFetchResult(BlockId(blockId), e))
        }
    }
```

shuffleClient 实际上在默认情况下（即 spark.shuffle.service.enabled 为 false）就是 blockTransferService：

```
private[spark] val shuffleClient = if (externalShuffleServiceEnabled) {
    val transConf = SparkTransportConf.fromSparkConf(conf, numUsableCores)
    new ExternalShuffleClient(transConf, securityManager, securityManager.
       isAuthenticationEnabled())
} else {
    blockTransferService
}

    blockTransferService是在SparkEnv里创建的:
    val blockTransferService =
        conf.get("spark.shuffle.blockTransferService", "netty").toLowerCase
            match {
        case "netty" =>
            new NettyBlockTransferService(conf, securityManager,
               numUsableCores)
        case "nio" =>
            new NioBlockTransferService(conf, securityManager)
}
```

接下来回头看一下 org.apache.spark.network.netty.NettyBlockTransferService 的 fetchBlocks 的实现：它会调用 org.apache.spark.network.shuffle.OneForOneBlockFetcher，OneForOneBlockFetcher 持有 org.apache.spark.network.client.TransportClient，它就是最终发送请求的 Handler。TransportClient 的请求会被 org.apache.spark.network.netty.NettyBlockRpcServer 接收并处理，通过上述的网络调用，请求最终会传到远程节点的 BlockManager：由 org.apache.spark.storage.BlockManager# getBlockData 处理这个读取 Block 的请求。具体实现过程如图 7-7 所示。接下来的调用栈就和本地读取一致了。

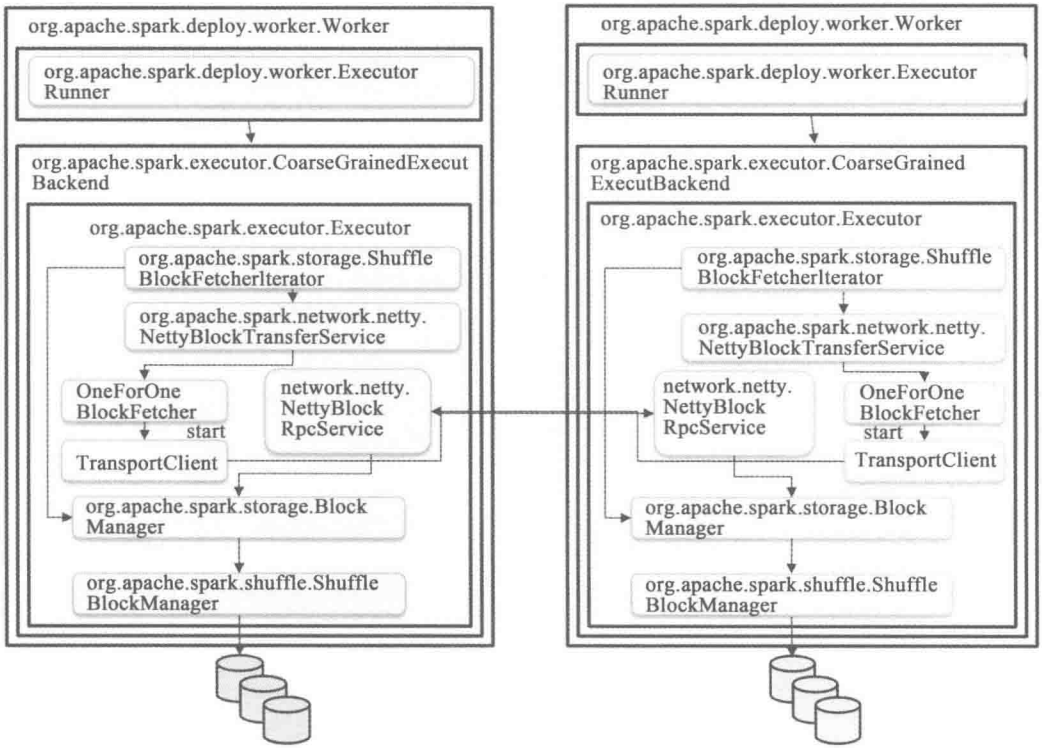


图 7-7 Shuffle Read 的详细实现

7.6 性能调优

通过上面的架构和源码实现的分析，不难得出 Shuffle 是 Spark Core 比较复杂的模块的结论。它也是非常影响性能的操作之一。因此，在这里整理了会影响 Shuffle 性能的各项配置。尽管在前文已经解释过它大部分配置项的含义，但是由于这些参数的确非常重要，这里算是做一个详细的总结。

7.6.1 spark.shuffle.manager

Spark 1.2.0 官方版本支持两种方式的 Shuffle，即 Hash Based Shuffle 和 Sort Based Shuffle。其中在 Spark 1.0 之前仅支持 Hash Based Shuffle。Spark 1.1 引入了 Sort Based Shuffle。Spark 1.2 的默认 Shuffle 机制从 Hash 变成了 Sort。如果需要 Hash

Based Shuffle，只需将 spark.shuffle.manager 设置成“hash”即可。

如果对性能有比较苛刻的要求，那么就要理解这两种不同的 Shuffle 机制的原理，结合具体的应用场景进行选择。

Hash Based Shuffle，就是根据 Hash 的结果，将各个 Reducer Partition 的数据写到单独的文件中去，写数据时不会有排序的操作。如果 Reducer 的 Partition 比较多，会产生大量的磁盘文件。这会带来两个问题：

1）同时打开的文件比较多，那么大量的文件句柄和写操作分配的临时内存会非常大，会对内存的使用和 GC 带来很多压力。尤其是在 Spark 的 YARN 模式下 Executor 分配的内存普遍比较小的时候，这个问题会更严重。

2）从整体来看，这些文件带来大量的随机读，读性能可能会遇到瓶颈。更加细节可以参见 7.1 节和 7.6.6 节。

Sort Based Shuffle 会根据实际情况对数据采用不同的方式进行 Sort。这个排序可能仅仅是按照 Reducer 的 Partition 进行排序，保证同一个 Shuffle Map Task 对应的不同的 Reducer Partition 的数据都可以写到同一个数据文件，通过一个 offset 来标记不同 Reducer Partition 的分界。因此一个 Shuffle Map Task 仅仅会生成一个数据文件（还有一个 Index 索引文件），从而避免了 Hash Based Shuffle 文件数量过多的问题。

选择 Hash 还是 Sort，取决于内存、排序和文件操作等因素的综合影响。

对于不需要进行排序且 Shuffle 产生的文件数量不是特别多时，Hash Based Shuffle 可能是更好的选择；因为 Sort Based Shuffle 会按照 Reducer 的 Partition 进行排序。

而 Sort Based Shuffle 的优势就在于可扩展性，它的出现实际上很大程度上是解决 Hash Based Shuffle 的可扩展性的问题。由于 Sort Based Shuffle 还在不断地演进中，因此它的性能会得到不断改善。

对于选择哪种 Shuffle，如果性能要求苛刻，最好还是通过实际测试后再做决定。不过选择默认的 Sort，可以满足大部分的场景需要。

7.6.2 spark.shuffle.spill

这个参数的默认值是 true，用于指定 Shuffle 过程中如果内存中的数据超过阈值（参考 spark.shuffle.memoryFraction 的设置）时是否需要将部分数据临时写入外部存储。如果设置为 false，那么这个过程就会一直使用内存，会有内存溢出的风险。因此只有在确定内存足够使用时，才可以将这个选项设置为 false。

Hash Based Shuffle 的 Shuffle Write 过程中使用的 org.apache.spark.util.collection.AppendOnlyMap 就是全内存的方式，而 org.apache.spark.util.collection.ExternalAppendOnlyMap 对 org.apache.spark.util.collection.AppendOnlyMap 有了进一步的封装，在内存使用超过阈值时会将它写入到外部存储，在最后的时候会对这些临时文件进行合并。

而 Sort Based Shuffle Write 使用到的 org.apache.spark.util.collection.ExternalSorter 也会有类似的写入。

对于 Shuffle Read，如果需要做聚合，也可能在聚合的过程中将数据写入的外部存储。

7.6.3 spark.shuffle.memoryFraction 和 spark.shuffle.safetyFraction

在启用 spark.shuffle.spill 的情况下，spark.shuffle.memoryFraction 决定了当 Shuffle 过程中使用的内存达到总内存多少比例的时候开始 spill。在 Spark 1.2.0 里，这个值是 0.2。通过这个参数可以设置 Shuffle 过程占用内存的大小，它直接影响了写入到外部存储的频率和垃圾回收的频率。

如果写入到外部存储的频率太高，那么可以适当地增加 spark.shuffle.memoryFraction 来增加 Shuffle 过程的可用内存数，进而减少写入到外部存储的频率。当然为了避免内存溢出，可能就需要减少 RDD cache 所用的内存，即需要减少 spark.storage.memoryFraction 的值；但是减少 RDD cache 所用的内存有可能会带来其他影响，因此需要综合考量。

在 Shuffle 过程中，Shuffle 占用的内存数是估计出来的，并不是每次新增的数据项都会计算一次占用的内存大小，这样做是为了降低时间开销。但是估计也会有误差，因此存在实际使用的内存数比估算值要大的情况，因此参数 spark.shuffle.safetyFraction 作为一个保险系数降低实际 Shuffle 过程所需要的内存值，可以降低实际内存超出用户配置值的风险。

7.6.4　spark.shuffle.sort.bypassMergeThreshold

这个配置的默认值是 200，用于设置在 Reducer 的 Partition 数目少于多少的时候，Sort Based Shuffle 内部不使用归并排序的方式处理数据，而是直接将每个 Partition 写入单独的文件。这个方式和 Hash Based 的方式类似，区别就是在最后这些文件还是会合并成一个单独的文件，并通过一个 Index 索引文件来标记不同 Partition 的位置信息。从 Reducer 来看，数据文件和索引文件的格式和内部是否做过归并排序是完全相同的。

这个可以看作 Sort Based Shuffle 在 Shuffle 量比较小的时候对于 Hash Based Shuffle 的一种折中。当然了它和 Hash Based Shuffle 一样，也存在同时打开文件过多导致内存占用增加的问题。因此如果 GC 比较严重或者内存比较紧张，可以适当降低这个值。

7.6.5　spark.shuffle.blockTransferService

在 Spark 1.2.0 中这个配置的默认值是 netty，而在之前的版本中是 nio。它主要是用于在各个 Executor 之间传输 Shuffle 数据。netty 的实现更加简洁，但实际上用户不用太关心这个选项。除非有特殊需求，否则采用默认配置即可。

7.6.6　spark.shuffle.consolidateFiles

这个配置的默认值是 false。主要是为了解决在 Hash Based Shuffle 过程中产生过多文件的问题。如果配置选项为 true，那么对于同一个 Core 上运行的 Shuffle Map Task 不会产生一个新的 Shuffle 文件而是重用原来的。但是每个 Shuffle Map Task 还是需要产生下游 Task 数量的文件，因此它并没有减少同时打开文件的数量。如果需要了

解更多细节，可以阅读 7.1 节。

但是 consolidateFiles 的机制在 Spark 0.8.1 就引入了，到 Spark 1.2.0 还是没有稳定下来。从源码实现的角度看，实现源码是非常简单的，但是由于涉及本地文件系统等限制，这个策略可能会带来各种各样的问题。由于它并没有减少同时打开文件的数量，因此不能减少由文件句柄带来的内存消耗。如果 Shuffle 的文件数量非常大，那么是否打开这个选项最好还是通过实际测试后再决定。

7.6.7 spark.shuffle.compress 和 spark.shuffle.spill.compress

这两个参数的默认配置都是 true。spark.shuffle.compress 和 spark.shuffle.spill.compress 都是用来设置 Shuffle 过程中是否对 Shuffle 数据进行压缩。其中，前者针对最终写入本地文件系统的输出文件；后者针对在处理过程需要写入到外部存储的中间数据，即针对最终的 shuffle 输出文件。

1. 设置 spark.shuffle.compress

如果下游的 Task 通过网络获取上游 Shuffle Map Task 的结果的网络 IO 成为瓶颈，那么就需要考虑将它设置为 true：通过压缩数据来减少网络 IO。由于上游 Shuffle Map Task 和下游的 Task 现阶段是不会并行处理的，即上游 Shuffle Map Task 处理完成后，下游的 Task 才会开始执行。那么需要压缩的时间消耗就是 Shuffle Map Task 压缩数据的时间 + 网络传输的时间 + 下游 Task 解压的时间；而不需要压缩的时间消耗仅仅是网络传输的时间。因此需要评估压缩解压时间带来的时间消耗和因为数据压缩带来的时间节省。如果网络成为瓶颈，比如集群普遍使用的是千兆网络，那么将这个选项设置为 true 可能更合理；如果计算是 CPU 密集型的，那么将这个选项设置为 false 可能更好。

2. 设置 spark.shuffle.spill.compress

如果设置为 true，代表处理的中间结果在 spill 到本地硬盘时都会进行压缩，在将中间结果取回进行 merge 的时候，要进行解压。因此要综合考虑 CPU 由于引入压缩、

解压的消耗时间和 Disk IO 因为压缩带来的节省时间的比较。在 Disk IO 成为瓶颈的场景下，设置为 true 可能比较合适；如果本地硬盘是 SSD，那么设置为 false 可能比较合适。

7.6.8　spark.reducer.maxMbInFlight

这个参数用于限制一个 Reducer Task 向其他的 Executor 请求 Shuffle 数据时所占用的最大内存数，尤其是如果网卡是千兆和千兆以下的网卡时。默认值是 48MB。设置这个值需要综合考虑网卡带宽和内存。

7.7　小结

在 Spark 0.6 和 0.7 时，Shuffle 的结果都需要先存储到内存中（有可能要写入磁盘），因此在大数据量的情况下，发生 GC 和 OOM 的概率非常大。因此在 Spark 0.8 的时候，Shuffle 的每个结果都会直接写入磁盘，并且为下游的每个 Task 都生成一个单独的文件。这样解决了 Shuffle 的结果都需要存入内存的问题，但是又引入了另外一个问题：生成的小文件过多，尤其在每个文件的数据量不大而文件特别多的时候，大量的随机读会严重影响性能。Spark 0.8.1 为了解决 0.8 中引入的问题，引入了 File Consolidation 机制，这在一定程度上解决了这个问题。由此可见，Hash Based Shuffle 在可扩展性方面的确有局限性。而 Spark1.0 中引入的 Shuffle Pluggable 框架，为加入新的 Shuffle 机制和引入第三方的 Shuffle 机制奠定了基础。在 Spark1.1 的时候，引入了 Sort Based Shuffle；并且在 Spark1.2.0 时，Sort Based Shuffle 成为 Shuffle 的默认选项。但是，随着内存成本的不断下降和容量的不断上升，Spark Core 会在未来重新将 Shuffle 过程在内存中进行吗？我认为这个不太可能也没太大必要，如果用户对于性能有比较苛刻的要求而 Shuffle 的过程的确是性能优化的重点时，可以尝试以下实现方式：

1）Worker 的节点采用固态硬盘。

2）Woker 的 Shuffle 结果保存到 RAM Disk 上。

3）根据自己的应用场景，实现自己的 Shuffle 机制。

我们知道只有 Parent Stage 的所有 Task 都完成了，下游的 Reducer Task 才能开始执行。这种实现逻辑比较简单，但是会影响性能。如果 Reducer Task 可以提早启动而不是等待所有的上游 Shuffle Map Task 都执行完成，是不是性能会更好，集群的利用率会更高？更进一步，现在都是采用 Reducer Task 从其他的节点上拉取数据（pull data），如果采用推送数据（push data）到 Reducer Task 呢？

第 8 章 Chapter 8

Storage 模块详解

Storage 模块负责管理 Spark 计算过程中产生的数据，包括基于 Disk 的和基于 Memory 的。用户在实际编程中，面对的是 RDD，可以将 RDD 的数据通过调用 org.apache.spark.rdd.RDD#cache 将数据持久化；持久化的动作都是由 Storage 模块完成的，包括 Shuffle 过程中的数据，也都是由 Storage 模块管理的。可以说，RDD 实现用户的逻辑，而 Storage 管理用户的数据。在 Driver 端和 Executor 端，都会有 Storage 模块，那么它们功能的共同点和不同点是什么？本章将讲解 Storage 模块的实现。

8.1 模块整体架构

8.1.1 整体架构

Storage 模块采用的是 Master/Slave 的架构。Master 负责整个 Application 的 Block 的元数据信息的管理和维护；而 Slave 需要将 Block 的更新等状态上报到 Master，同时接收 Master 的命令，比如删除一个 RDD、Shuffle 相关的数据或者是广播变量。而 Master 与 Slave 之间通过 AKKA 消息传递机制进行通信。

在 SparkContext 创建时，它会创建 Driver 端的 SparkEnv，而 SparkEnv 会创建 BlockManager，BlockManager 创建的时候会持有一个 BlockManagerMaster。BlockManagerMaster 会把请求转发给 BlockManagerMasterActor 来完成元数据的管理和维护。

而在 Executor 端，也存在一个 BlockManager，它也会持有一个 BlockManagerMaster，只不过 BlockManagerMaster 会持有一个 Driver 端 BlockManagerMasterActor 的 Reference，因此 Executor 端的 BlockManager 就能通过这个 Actor 的 Reference 将 Block 的信息上报给 Master。

BlockManager 本身还持有一个 BlockManagerSlaveActor，而这个 Slave 的 Actor 还会被上报到 Master。Master 会持有这个 Slave Actor 的 Reference，并通过这个 Reference 向 Salve 发送一些命令，比如删除 Slave 上的 RDD、Shuffle 相关的数据或者是广播变量。上面说的这些关系可以通过图 8-1 来表示。

图 8-1 并没有指出 Master 和 Slave 之间的心跳是如何维持的，即 Master 如何知道 Slave 是否已经下线呢？实际上，Master 和 Slave 之间并没有专门的心跳，而是通过 Driver 和 Executor 之间的心跳来间接完成的。

Master 持有整个 Application 的 Block 的元数据信息，包括 Block 所在的位置，Block 所占的存储空间大小（含三种类别：内存、Disk 和 Tachyon）。这些信息都保存在 org.apache.spark.storage.BlockManagerMasterActor 的三个数据结构中：

1）private val blockManagerInfo = new mutable.HashMap[BlockManagerId, BlockManagerInfo]：保存了 BlockManagerId 到 BlockManagerInfo 的映射。BlockManagerInfo 保存了 Slave 节点的内存使用情况、Slave 上的 Block 的状态、Slave 上的 BlockManagerSlaveActor 的 Reference（Master 通过这个 Reference 可以向 Slave 发送命令和查询请求）。

2）private val blockManagerIdByExecutor = new mutable.HashMap[String, BlockManagerId]：保存了 Executor ID 到 BlockManagerId 的映射。这样 Master 就可以通过 Executor ID 快速查找到 BlockManagerId。

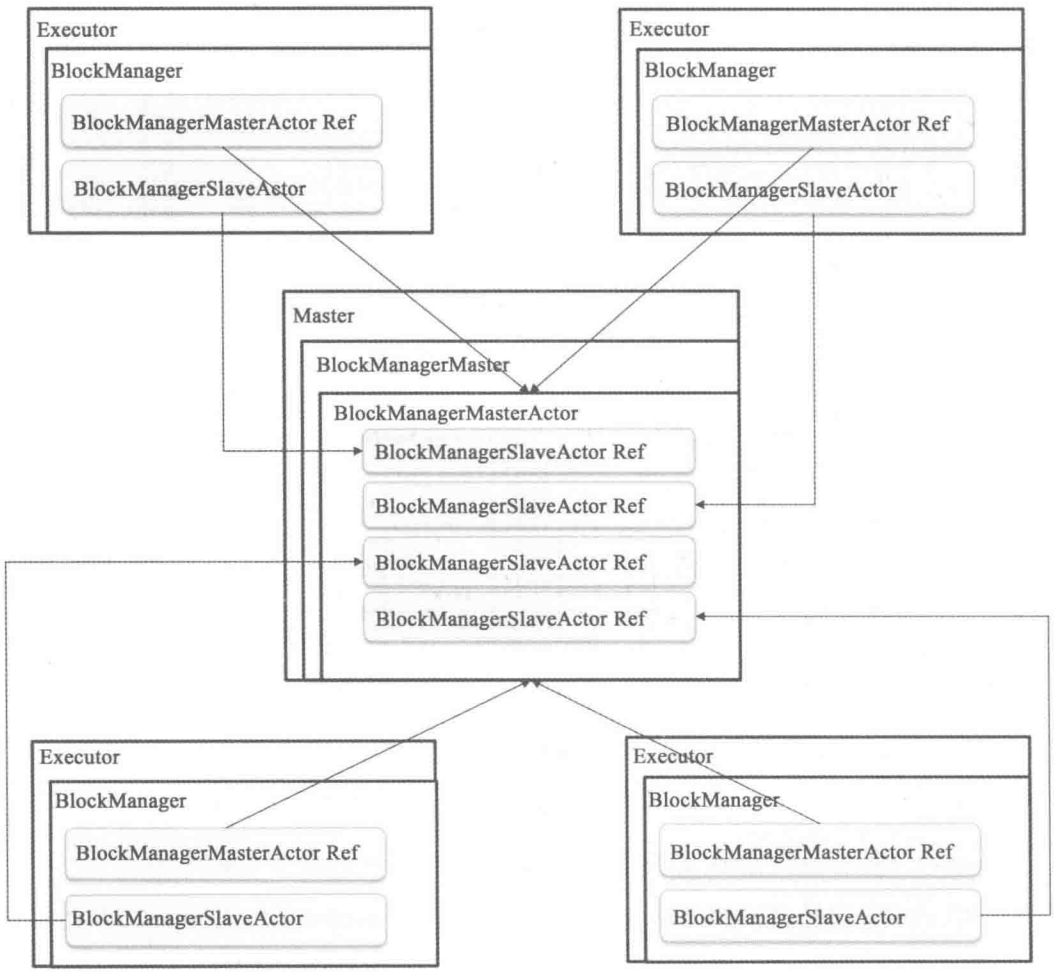


图 8-1　Storage 模块架构图

3）private val blockLocations = new JHashMap[BlockId, mutable.HashSet [BlockManagerId]]：保存 Block 是在哪些 BlockManager 上的 Hash Map；由于 Block 可能在多个 Slave 上都有备份，因此注意 Value 是一个 mutable.HashSet。通过查询 blockLocations 就可以找到某个 Block 所在的物理位置了。

那么 BlockManagerMasterActor 是如何维护和管理这些数据结构，进而实现管理整个 Application 的 Block 呢？实现过程如图 8-2 所示。

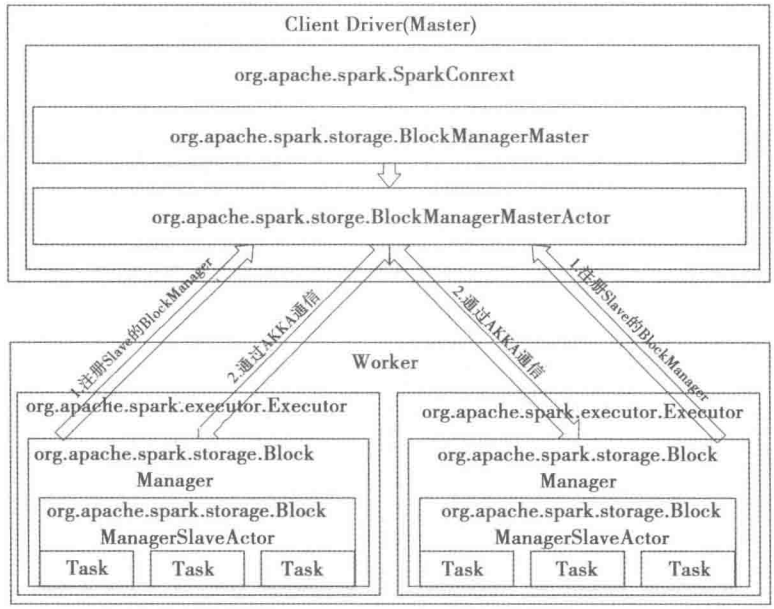


图 8-2 BlockManager 的注册实现逻辑图

BlockManger 在创建后，需要向 BlockManagerMasterActor 发送消息 RegisterBlockManager 进行注册，具体实现代码在 org.apache.spark.storage.BlockManager#initialize：

```
master.registerBlockManager(blockManagerId, maxMemory, slaveActor)
```

注册的信息包括 blockManagerId（标识了 Slave 的 Executor ID、Hostname 和 Port），节点的最大可用内存数，BlockManagerSlaveActor。BlockManagerMasterActor 接到注册请求后，会将 Slave 的信息保存到 Master 端：

```
blockManagerIdByExecutor(id.executorId) = id
blockManagerInfo(id) = new BlockManagerInfo(
    id, System.currentTimeMillis(), maxMemSize, slaveActor)
```

8.1.2 源码组织结构

org.apache.spark.storage.BlockManager 是 Storage 模块与其他模块交互最主要的类，它提供了读和写 Block 的接口。这里的 Block，实际上就对应了 RDD 中提到的 Partition，每一个 Partition 都会对应一个 Block。每个 Block 由唯一的 Block ID（org.

apache.spark.storage.RDDBlockId）标识，格式是 "rdd_" + rddId + "_" + partitionId，如图 8-3 所示。

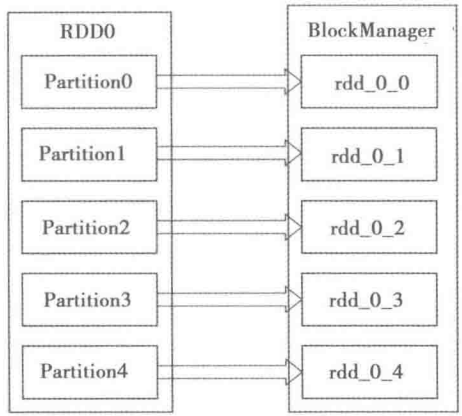


图 8-3　RDD 的 Partition 和 Block 的逻辑关系图

BlockManager 会运行在 Driver 和每个 Executor 上。而运行在 Driver 上的 BlockManger 负责整个 Application 的 Block 的管理工作；运行在 Executor 上的 BlockManger 负责管理该 Executor 上的 Block，并且向 Driver 的 BlockManager 汇报 Block 的信息和接收来自它的命令。

模块的类图如图 8-4 所示。各个主要类的功能说明如下：

1）org.apache.spark.storage.BlockManager：提供了 Storage 模块与其他模块的交互接口，管理 Storage 模块。

2）org.apache.spark.storage.BlockManagerMaster：Block 管理的接口类，主要通过调用 org.apache.spark.storage.BlockManagerMasterActor 来完成。

3）org.apache.spark.storage.BlockManagerMasterActor：在 Driver 节点上的 Actor，负责 track 所有 Slave 节点的 Block 的信息。

4）org.apache.spark.storage.BlockManagerSlaveActor：运行在所有的节点上，接收来自 org.apache.spark.storage.BlockManagerMasterActor 的命令，比如删除某个 RDD 的数据、删除某个 Block、删除某个 Shuffle 数据、返回某些 Block 的状态等。

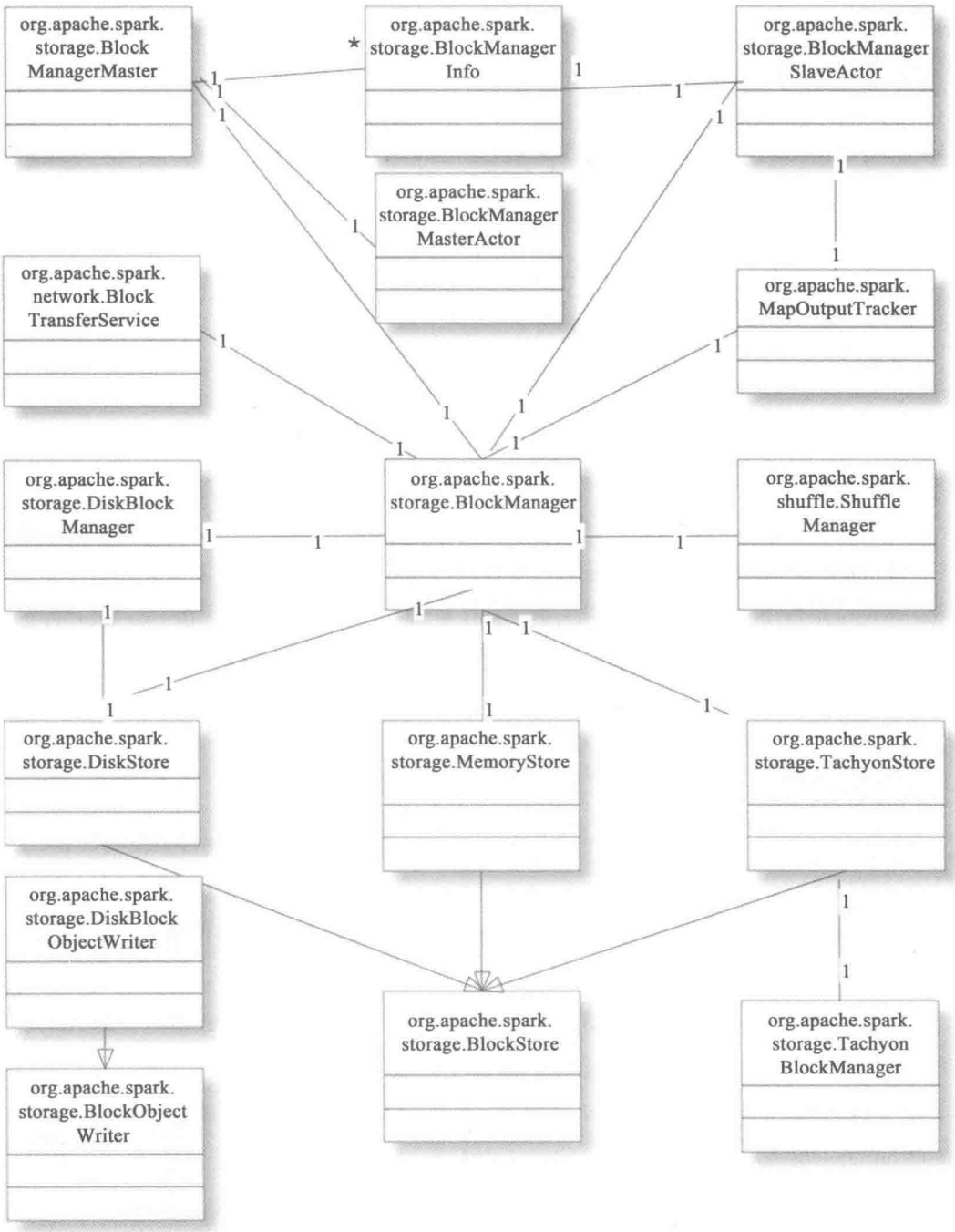


图 8-4 Storage 模块类图

5）org.apache.spark.storage.BlockObjectWriter：一个抽象类，可以将任何的 JVM Object 写入外部存储系统。注意，它不支持并发的写操作。

6）org.apache.spark.storage.DiskBlockObjectWriter：支持直接写入一个文件到 Disk，并且还支持文件的追加。实际上它是 org.apache.spark.storage.BlockObjectWriter 的一个实现。下面的类在需要写入数据到 Disk 时，就是通过它来完成的：

a）org.apache.spark.util.collection.ExternalSorter

b）org.apache.spark.shuffle.FileShuffleBlockManager

7）org.apache.spark.storage.DiskBlockManager：管理和维护了逻辑上的 Block 和存储在 Disk 上的物理的 Block 的映射。一般来说，一个逻辑的 Block 会根据它的 BlockId 生成的名字映射到一个物理上的文件。这些物理文件会被 hash 到由 spark.local.dir（或者通过 SPARK_LOCAL_DIRS）设置的目录中。

8）org.apache.spark.storage.BlockStore：存储 Block 的抽象类。现在它的实现有：

a）org.apache.spark.storage.DiskStore

b）org.apache.spark.storage.MemoryStore

c）org.apache.spark.storage.TachyonStore

9）org.apache.spark.storage.DiskStore：实现了存储 Block 到 Disk 上。其中写 Disk 是通过 org.apache.spark.storage.DiskBlockObjectWriter 实现的。

10）org.apache.spark.storage.MemoryStore：实现了存储 Block 到内存中。

11）org.apache.spark.storage.TachyonStore：实现了存储 Block 到 Tachyon 上。

12）org.apache.spark.storage.TachyonBlockManager：管理和维护逻辑上的 Block 和 Tachyon 文件系统上的文件之间的映射。这点和 org.apache.spark.storage.DiskBlockManager 功能类似。

8.1.3　Master 和 Slave 的消息传递详解

通过对消息传递协议的梳理，可以更加清楚地理解 Storage 模块在 Driver 和 Executor 之间的控制信息是如何传递的；还可以了解 Storage 模块管理 Block 的框架和脉络。

这里的 Master 指的是 org.apache.spark.storage.BlockManagerMasterActor，它运行在 Driver 端，通过保存 Slave 的 Actor Reference 向 Slave 发送消息；Slave 指的是 org.apache.spark.storage.BlockManagerSlaveActor，它运行在 Executor 端，每个 Executor 都有一个，主要的功能是接收来自 Master 的命令，做一些清理工作并且响应 Master 获取 Block 的状态的请求，同时 Executor 都会保存有 org.apache.spark.storage.BlockManagerMasterActor 的 Actor Reference，Executor 通过这个 Reference 和 Master 进行通信。

1. Master 到 Slave 消息详解

Slave 有任何的 Block 的状态更新，都会主动通过 Master Actor Reference 上报到 Master。而从 Master 到 Slave 的消息主要是由 Master 向 Slave 发送的控制信息或者获取状态的请求。控制信息主要指删除 Block、RDD 相关的 Block、广播变量相关的数据和 Shuffle 相关的数据。获取状态的请求主要是获取 Block 的状态信息和匹配的 Block ID 等信息。

那么 Master 端是如何向 Slave 发送消息呢?

在 Slave 向 Master 发起注册 BlockManager 的请求中，每个 BlockManager 的信息都保存在 org.apache.spark.storage.BlockManagerInfo 中，因此通过 BlockManagerInfo 就可以得到 Slave Actor 的 Reference，进而向它发送消息。下面通过删除 RDD 的例子，来详细讲解这一个过程。在 RDD 调用了 org.apache.spark.rdd.RDD#unpersist，详细的调用栈如下：

1）org.apache.spark.SparkContext#unpersistRDD

相关代码：

```
sc.unpersistRDD(id, blocking)
```

2）org.apache.spark.storage.BlockManagerMaster#removeRdd

相关代码：

```
env.blockManager.master.removeRdd(rddId, blocking)
```

3）org.apache.spark.storage.BlockManagerMaster#askDriverWithReply

相关代码：

```
val future = askDriverWithReply[Future[Seq[Int]]](RemoveRdd(rddId))
```

4）org.apache.spark.util.AkkaUtils$#askWithReply(java.lang.Object, akka.actor.ActorRef, int, int, scala.concurrent.duration.FiniteDuration)

相关代码：

```
AkkaUtils.askWithReply(message, driverActor, AKKA_RETRY_ATTEMPTS, AKKA_
RETRY_INTERVAL_MS, timeout)
```

5）org.apache.spark.storage.BlockManagerMasterActor#receiveWithLogging

相关代码：

```
case RemoveRdd(rddId) =>sender ! removeRdd(rddId) )
```

6）org.apache.spark.storage.BlockManagerMasterActor#removeRdd：

这个比较复杂，首先需要删除这个 RDD 相关的元数据信息，然后删除 Slave 上的 RDD 的信息。

首先删除 Master 保存的 RDD 相关的元数据信息。

```
val blocks = blockLocations.keys.flatMap(_.asRDDId).filter(_.rddId == rddId)
blocks.foreach { blockId =>val bms: mutable.HashSet[BlockManagerId] = block-
    Locations.get(blockId)
bms.foreach(bm => blockManagerInfo.get(bm).foreach(_.removeBlock(blockId)))
    blockLocations.remove(blockId)
}
```

其次删除 Slave 上的 RDD 的信息。

```
val removeMsg = RemoveRdd(rddId)
Future.sequence(
    blockManagerInfo.values.map { bm =>
        bm.slaveActor.ask(removeMsg)(akkaTimeout).mapTo[Int]
    }.toSeq
```

```
)
```

在 Slave 接到 RemoveRdd 的消息后，会调用 BlockManager 删除 RDD：

```
case RemoveRdd(rddId) =>
    doAsync[Int]("removing RDD " + rddId, sender) {
        blockManager.removeRdd(rddId)
}
```

BlockManager 端的实现：

```
def removeRdd(rddId: Int): Int = {
    logInfo(s"Removing RDD $rddId")
    val blocksToRemove = blockInfo.keys.flatMap(_.asRDDId).filter(_.rddId == rddId)
    blocksToRemove.foreach { blockId => removeBlock(blockId, tellMaster =
        false) }
    blocksToRemove.size
}
```

下面 6 个消息都是从 Master Actor 传到 Slave Actor 的简要说明。如果对实现感兴趣，可以深入阅读一下相关的实现，这里不再赘述。

1）RemoveBlock(blockId: BlockId)：根据 blockId 删除该 Executor 上的 Block，实际上通过调用 org.apache.spark.storage.BlockManager#removeBlock 删除。

2）RemoveRdd(rddId: Int)：根据 rddId 删除该 Executor 上 RDD 所关联的所有 Block，实际上通过调用 org.apache.spark.storage.BlockManager#removeRdd 删除。

3）RemoveShuffle(shuffleId: Int)：根据 shuffleId 删除该 Executor 上所有和该 Shuffle 相关的 Block，需要通过两步实现：

a）org.apache.spark.MapOutputTracker#unregisterShuffle

b）org.apache.spark.shuffle.ShuffleManager#unregisterShuffle

4）RemoveBroadcast(broadcastId: Long, removeFromDriver: Boolean = true)：根据 broadcastId 删除该 Executor 和该广播变量相关的所有 Block，通过 org.apache.spark.storage.BlockManager#removeBroadcast 实现。

5）GetBlockStatus(blockId: BlockId, _)：根据 blockId 向 Master 返回该 Block 的 Status。这个一般都是测试使用的，注意这个操作非常耗时。实现方式：org.apache.spark.storage.BlockManager#getStatus。

6）GetMatchingBlockIds(filter, _)：根据 filter 向 Master 返回符合 filter 的所有

BlockId。通过 org.apache.spark.storage.BlockManager#getMatchingBlockIds 实现。这个和第 5 个消息一样，也非常耗时，一般都是在测试中使用的。

2. Slave 到 Master 消息详解

首先强调一点，Slave Actor 并不会主动向 Master Actor 发送消息。这里的 Slave 到 Master 的消息，主要还是由 Slave 节点上的 BlockManager 发出的。下面将通过分析 Slave 如何向 Master 汇报某个 Block 的状态更新，来详细阐述实现细节。

BlockManager 通过调用 org.apache.spark.storage.BlockManager#reportBlockStatus 来汇报一个 Block 的状态。那么什么情况下会汇报 Block 的状态呢？当然是 Block 的状态改变的时候。比如通过 org.apache.spark.storage.BlockManager#dropFromMemory 将某个 Block 从内存中移出；比如通过 org.apache.spark.storage.BlockManager#removeBlock 删除一个 Block 并且需要通知 Master 时；比如通过 org.apache.spark.storage.BlockManager#dropOldBlocks 移除一个 Block 时；比如通过 org.apache.spark.storage.BlockManager#doPut 写入一个新的 Block 时。org.apache.spark.storage.BlockManager#reportBlockStatus 之后的调用栈如下：

1）org.apache.spark.storage.BlockManager#tryToReportBlockStatus

相关代码：

```
val needReregister = !tryToReportBlockStatus(blockId, info, status,
   droppedMemorySize)
```

2）org.apache.spark.storage.BlockManagerMaster#updateBlockInfo

相关代码：

```
master.updateBlockInfo( blockManagerId, blockId, storageLevel, inMemSize,
    onDiskSize, inTachyonSize)
```

3）org.apache.spark.storage.BlockManagerMaster#askDriverWithReply

相关代码：

```
val res = askDriverWithReply[Boolean](UpdateBlockInfo(blockManagerId,
```

```
    blockId, storageLevel, memSize, diskSize, tachyonSize))
```

4）org.apache.spark.util.AkkaUtils$#askWithReply(java.lang.Object, akka.actor.ActorRef, int, int, scala.concurrent.duration.FiniteDuration)

相关代码：

```
AkkaUtils.askWithReply(message, driverActor, AKKA_RETRY_ATTEMPTS, AKKA_
    RETRY_INTERVAL_MS, timeout)
```

5）org.apache.spark.storage.BlockManagerMasterActor#receiveWithLogging

相关代码：

```
case UpdateBlockInfo(
blockManagerId, blockId, storageLevel, deserializedSize, size, tachyonSize) =>
sender ! updateBlockInfo(
blockManagerId, blockId, storageLevel, deserializedSize, size, tachyonSize)
```

6）org.apache.spark.storage.BlockManagerMasterActor#updateBlockInfo

相关实现：这个调用会进行 Master 端控制信息的更新。主要更新两个数据结构：

a）blockManagerInfo 中是 blockManagerId 为 key，BlockManagerInfo 为 value 的 HashMap。

b）blockLocations 中 是 blockId 为 key，mutable.HashSet[BlockManagerId] 为 value 的 HashMap。

更新 blockManagerInfo：

```
blockManagerInfo(blockManagerId).updateBlockInfo(
blockId, storageLevel, memSize, diskSize, tachyonSize)
```

更新 blockLocations：

```
var locations: mutable.HashSet[BlockManagerId] = null
if (blockLocations.containsKey(blockId)) {
    //该Block是有信息更新或者有多个备份
    locations = blockLocations.get(blockId)
} else {
    //新加入的Block
```

```
        locations = new mutable.HashSet[BlockManagerId]
        blockLocations.put(blockId, locations)
    }

    if (storageLevel.isValid) {
        locations.add(blockManagerId) //为该Block加入新的位置
    } else {
        locations.remove(blockManagerId) //删除无效的Block的位置
    }
    //删除在Slave上已经不存在的Block
    if (locations.size == 0) {
        blockLocations.remove(blockId)
    }
```

一般来说，Slave 向 Master 汇报信息获取查询信息，调用栈都是类似的：

1）org.apache.spark.storage.BlockManager

2）org.apache.spark.storage.BlockManagerMaster

3）org.apache.spark.storage.BlockManagerMasterActor

其中 2 ～ 3 实际上是需要通过网络的，即通过 BlockManagerMaster 持有的 Master Actor Reference 向 Master Actor 发送消息。

下面 14 个消息都是从 Master Actor 传到 Slave Actor 的简要说明。如果对实现感兴趣，可以深入阅读一下相关的实现，这里不再赘述。

1）RegisterBlockManager(blockManagerId: BlockManagerId, maxMemSize: Long, sender: ActorRef)：　由 org.apache.spark.storage.BlockManagerMaster 向 org.apache.spark.storage.BlockManagerMasterActor 发起的注册。通过注册，Master Actor 会保存该 BlockManger 所包含的 Block 等信息。

2）UpdateBlockInfo(var blockManagerId: BlockManagerId,var blockId: BlockId,var storageLevel: StorageLevel,var memSize: Long,var diskSize: Long,var tachyonSize: Long)：向 Master 汇报 Block 的信息，Master 会记录这些信息并且供 Slave 查询。

3）GetLocations(blockId: BlockId)：获得某个 Block 所在的位置信息，返回由 org.apache.spark.storage.BlockManagerId 组成的列表（包含 Executor ID，Executor 所在的 hostname 和开放的 port 号）。注意：Block 可能在多个节点上都有备份。

4）GetLocationsMultipleBlockIds(blockIds：Array[BlockId])：与 GetLocations (blockId: BlockId) 基本相同，区别是它可以一次获取多个 Block 的位置信息。

5）GetPeers(blockManagerId: BlockManagerId)：获得其他的 BlockManager。这个在做 Block 的副本时会用到。BlockManager 通过 org.apache.spark.storage.BlockManager#doPut 去执行写 Block 的操作。在写 Block 的时候如果需要写副本，那么需要调用 org.apache.spark.storage.BlockManager#replicate。在复制的时候会获取其他的 BlockManager 的位置信息，相对于它自身，这些 BlockManager 就是它的副本了。

6）GetActorSystemHostPortForExecutor(executorId: String)：根据 executorId 获取 Executor 的 Thread Dump，是由 org.apache.spark.SparkContext#getExecutorThreadDump 发起的调用，它在获得了 Executor 的 hostname 和 port 后，会通过 AKKA 向 Executor 发送请求信息。实际上获取 Thead Dump 最开始是由 org.apache.spark.ui.exec.Executor-ThreadDumpPage 发起的。

7）RemoveExecutor(execId: String)：删除 Master 上保存的 execId 对应的 Executor 上的 BlockManager 的信息。

8）StopBlockManagerMaster：在停止 org.apache.spark.storage.BlockManagerMaster 时调用。它会停止 Master 的 Actor。实际上这是 Spark 的 Driver 和 Executor 退出的时候发起的停止动作。

9）GetMemoryStatus：获取所有 Executor 的内存使用的状态，包括使用的最大的内存大小、剩余的内存大小。

10）GetStorageStatus：返回每个 Executor 的 Storage 的状态，包括每个 Executor 最大可用的内存数和 Block 的信息。

11）GetBlockStatus(blockId: BlockId, askSlaves: Boolean = true)：根据 blockId 获取 Block 的 Status。如果 askSlaves 为 true，那么需要到所有的 Slave 上查询结果，因此可能会非常耗时。而且在某种情况下可能某些 Block 的状态并没有上报。推荐还是作为一个测试项使用。

12）GetMatchingBlockIds(filter: BlockId => Boolean, askSlaves: Boolean = true)：和 GetBlockStatus 类似，只不过这个是根据 filter 获取。

13）BlockManagerHeartbeat(blockManagerId: BlockManagerId)：前面提到过 Slave 和 Master 的心跳的实现。实际上，Slave 的 BlockManager 并不会主动发起心跳，而是通过 Executor 和 Driver 之间的心跳来实现的。Driver 在创建 SparkContext 的时候就会初始化 org.apache.spark.HeartbeatReceiver，这是一个 Actor，而 Executor 启动时会创建这个 Actor 的 Reference。然后 Executor 就通过 Actor 之间的通信来维持这个心跳。Driver 端在收到心跳后，会调用 org.apache.spark.scheduler.TaskScheduler#executorHeartbeatReceived，而 TaskScheduler 会调用 DAGScheduler 向 org.apache.spark.storage.BlockManagerMasterActor 发送心跳。

14）ExpireDeadHosts：删除心跳超时的 BlockManager。心跳检测的线程是在 Master Actor 的 preStart() 里启动的，默认的超时时间是 60 秒。可以通过 spark.storage.blockManagerTimeoutIntervalMs 来设置，注意时间单位是毫秒。

8.2　存储实现详解

8.2.1　存储级别

存储级别，对用户来说是 RDD 相关的持久化和缓存。这实际上也 Spark 最重要的特征之一。在每个节点都将 RDD 的 Partition 的数据保存在内存中时，后续的计算将会变得非常快（通常会快 10 倍以上）。可以说，缓存是 Spark 构建迭代式算法和快速交互式查询的关键。

用户可以直接调用 persist() 或者 cache() 来标记一个 RDD 需要持久化。实际上，cache() 是使用默认存储级别的快捷方法：

```
/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)

/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def cache(): this.type = persist()
```

前面也介绍过，只有触发了一个 Action 后，计算才会提交到集群开始真正的运算。因此 RDD 只有经过一次 Action 后，才能将 RDD 缓存到内存中以供以后的计算使

用。这个缓存也有容错机制，如果某个缓存丢失了，那么会通过原来的计算过程进行重算。

如果用户有特殊需求，可以调用 def persist(newLevel: StorageLevel) 来设置需要的存储级别。不同的存储级别，可以选择持久化数据到 Disk、Memory、Tachyon；还可以选择数据是否需要序列化从而节省空间；甚至可以将数据远程复制到其他节点。

1. 存储级别的定义

RDD 的 Partition 和 Storage 模块的 Block 是一一对应的关系。表 8-1 列出了 Storage 模块支持的存储级别。为了和官网保持一致，在介绍存储级别相关的含义时，都使用了 Partition 而不是 Block。

表 8-1 Storage 模块支持的存储级别

存储级别	意义
NONE	不会保存任何数据
DISK_ONLY	直接将 RDD 的 Partition 保存在该节点的 Disk 上
MEMORY_ONLY	将 RDD 的 Partition 对应的原生的 Java Object 保存在 JVM 中。如果 RDD 太大导致它的部分 Partition 不能存储在内存中，那么这些 Partition 将不会被缓存，并且在需要的时候被重新计算。这是默认的级别
MEMORY_ONLY_SER	将 RDD 的 Partition 序列化后的对象（每一个 Partition 占用一个字节数组）存储在 JVM 中。通常来说，这将比直接保存原始对象的空间利用率更高，尤其当在使用 fast serializer（快速序列化）时。但在读取时由于需要反序列化会比较占用 CPU
MEMORY_AND_DISK	将 RDD 的 Partition 反序列化后的对象存储在 JVM 中。如果 RDD 太大导致它的部分 Partition 不能存储在内存中，超出的 partition 将被保存在 Disk 上，并且在需要时读取
MEMORY_AND_DISK_SER	与 MEMORY_ONLY_SER 类似，但是会把超出内存的 Partition 存储在 Disk 上而不是在每次需要的时候重新计算
DISK_ONLY_2 MEMORY_ONLY_2 MEMORY_ONLY_SER_2 MEMORY_AND_DISK_2 MEMORY_AND_DISK_SER_2	同上述的存储级别。不同就是在其他的节点上会保存一个相同的备份。从集群的角度看，一共有两个备份

（续）

存储级别	意　义
OFF_HEAP(experimental)	将 RDD 的 Partition 序列化后存储在 Tachyon 中。相比 MEMORY_ONLY_SER，OFF_HEAP 有几个优势： 1）减少了 GC 带来的性能损耗 2）使得 Executor 内存使用更加轻量级 3）在集群的角度共享一个内存池，非常有利于对内存有超大需求的 Application；而且使得在过个 Executor 中间共享内存数据成为可能

读者可以通过 http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence 来获取最新的关于存储级别的说明。

实际上，上述的存储级别是通过 org.apache.spark.storage.StorageLevel 定义的。

```
object StorageLevel {
    val NONE = new StorageLevel(false, false, false, false)
    val DISK_ONLY = new StorageLevel(true, false, false, false)
    val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
    val MEMORY_ONLY = new StorageLevel(false, true, false, true)
    val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
    val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
    val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
    val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
    val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
    val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
    val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
    val OFF_HEAP = new StorageLevel(false, false, true, false)
```

各个参数的含义可以参见：

```
class StorageLevel private(
    private var _useDisk: Boolean,
    private var _useMemory: Boolean,
    private var _useOffHeap: Boolean,
    private var _deserialized: Boolean,
    private var _replication: Int = 1)
```

2. 选择合适的存储级别

Spark 不同的存储级别是内存使用和 CPU 效率的折中。Spark 官网建议按照以下步骤来选择合适的存储级别：

1）如果你的 RDD 可以和默认的存储级别有很好的契合，那么就无需任何特殊的设定了。默认的存储级别是 CPU 最高效的选项，也是运算能够最快完成的选项。

2）如果不行，那么需要减少内存的使用，可以使用 MEMORY_ONLY_SER。这时候需要选择一个合适的序列化方案，可以参考 http://spark.apache.org/docs/latest/tuning.html#data-serialization 。需要在空间效率和反序列化时所需要的 CPU 中做一个合适的选择。

3）尽量不要落到硬盘上，除非是计算逻辑非常复杂，或者是需要从一个超大规模的数据集过滤出一小部分数据。否则重新计算一个 Partition 的速度可能和从硬盘读差不多（考虑到出错的概率和写硬盘的开销，因此采用失败重算要比读硬盘持久化的数据要好）。

4）如果你需要故障的快速恢复能力（比如使用 Spark 来处理 Web 的请求），那么可以考虑使用存储级别的多副本机制。实际上所有的存储级别都提供了 Partition 数据丢失时的重算机制，只不过有备份的话可以让 Application 直接使用副本而无须等待重新计算丢失的 Partition 数据。

5）如果集群有大量的内存或者有很多的运行任务，则选择 OFF_HEAP。现在处于试验阶段的 OFF_HEAP 有以下的优势：

a）它使得多个 Executor 可以共享一个内存池。

b）它显著地减少了 GC 的开销。

c）缓存在内存中的数据即使是产生它的 Executor 异常退出了也不会丢失。

8.2.2 模块类图

org.apache.spark.storage.BlockStore：存储 Block 的抽象类。它的实现有：

1）org.apache.spark.storage.DiskStore

2）org.apache.spark.storage.MemoryStore

3）org.apache.spark.storage.TachyonStore

该模块的类图如图 8-5 所示。

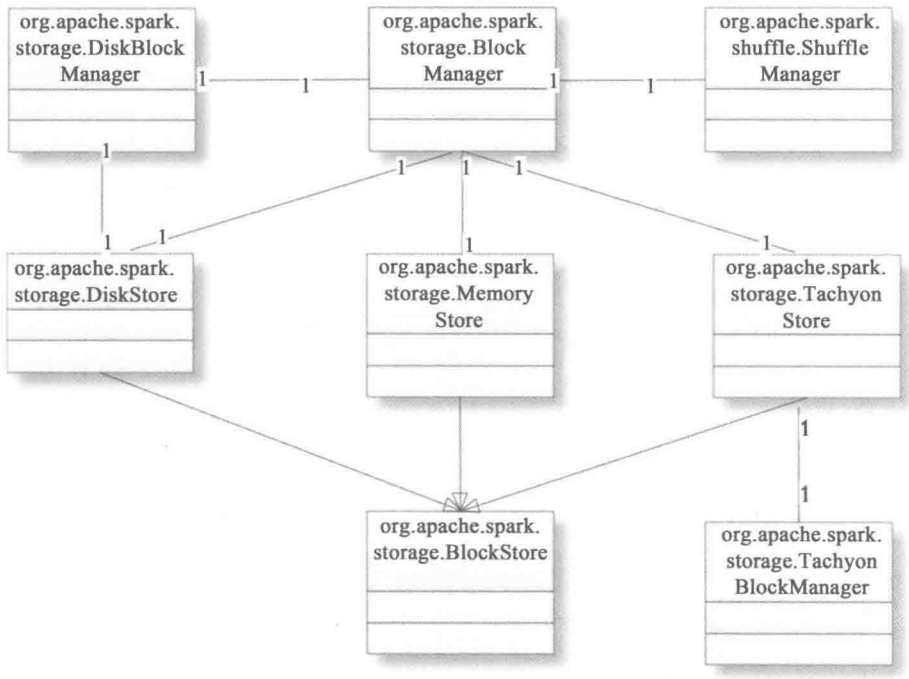


图 8-5　存储实现类图

首先通过 org.apache.spark.storage.BlockStore 来看一下实现类需要的接口名字及接口的功能。

```
private[spark] abstract class BlockStore(val blockManager: BlockManager)
    extends Logging {
    //根据StorageLevel将blockId标识的Block的内容bytes写入系统
    def putBytes(blockId: BlockId, bytes: ByteBuffer, level: StorageLevel):
       PutResult
    //将values写入系统。如果returnValues = true，需要将结果写入PutResult
    def putIterator(
        blockId: BlockId,
        values: Iterator[Any],
        level: StorageLevel,
        returnValues: Boolean): PutResult
    //同上，只不过由Iterator编程Array
    def putArray(
        blockId: BlockId,
        values: Array[Any],
        level: StorageLevel,
```

```
        returnValues: Boolean): PutResult
    // 获得Block的大小
    def getSize(blockId: BlockId): Long
    //获得Block的数据，返回类型ByteBuffer
    def getBytes(blockId: BlockId): Option[ByteBuffer]
    //获得Block的数据，返回类型Iterator[Any]
    def getValues(blockId: BlockId): Option[Iterator[Any]]
    //删除Block，成功返回true，否则false
    def remove(blockId: BlockId): Boolean
    //查询是否包含某个Block
    def contains(blockId: BlockId): Boolean
    //退出时清理回收资源
    def clear() { }
}
```

8.2.3 org.apache.spark.storage.DiskStore 实现详解

DiskStore 通过 org.apache.spark.storage.DiskBlockManager 来管理文件。前面介绍过，DiskBlockManager 管理和维护了逻辑上的 Block 和存储在 Disk 上物理的 Block 的映射。一般来说，一个逻辑的 Block 会根据它的 blockId 生成的名字映射到一个物理上的文件。这些物理文件会被 hash 到由 spark.local.dir（或者 SPARK_LOCAL_DIRS）设置的目录中。

一般来说，DiskStore 会通过 blockId 从 DiskBlockManager 获取一个文件句柄，然后通过这个文件句柄来读写文件：

```
val file = diskManager.getFile(blockId) //获取文件句柄
val channel = new FileOutputStream(file).getChannel
while (bytes.remaining > 0) {
    channel.write(bytes)
}
channel.close()
```

那么 DiskBlockManager 是如何管理这些文件的呢？

spark.local.dir（或者 SPARK_LOCAL_DIRS）可以设置多个目录。DiskBlockManager 会为 Executor 在每个目录下创建一个子目录，子目录的命名方式是“spark-local- yyyyMMddHHmmss-xxxx”，其中，最后的 xxxx 是一个随机数，这个实现的逻

辑在 org.apache.spark.storage.DiskBlockManager#createLocalDirs 中。而每个“spark-local- yyyyMMddHHmmss-xxxx”目录下，会根据需要生成个数至多为 spark.diskStore.subDirectories（默认值为 64）的子目录，子目录以数字命名（从 00 到文件个数）。图 8-6 表示了用户在 spark.local.dir 中设置了 3 个目录，每个目录下有 64 个子目录的目录布局。

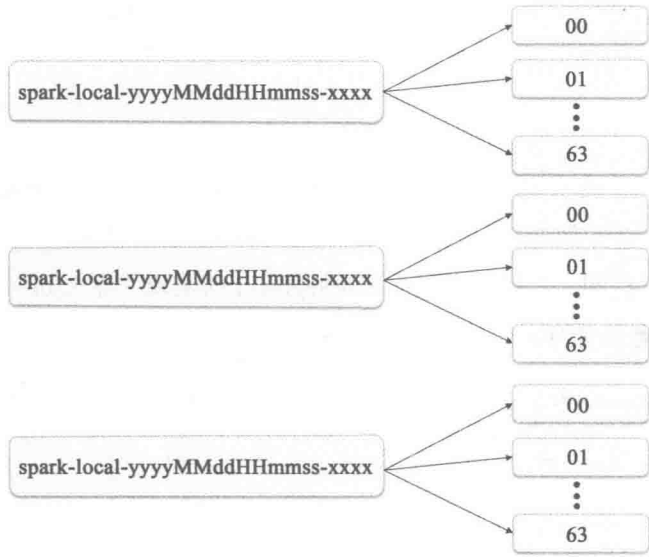


图 8-6　Block 存储的目录布局

那么如何确定某个 Block 需要保存在哪个子目录呢？根据文件名的 hash 值取得其应该存放的一级目录（即“spark-local- yyyyMMddHHmmss-xxxx”），然后根据该 hash 值取得其应该存放的二级目录（即一级目录的子目录）。主要的实现如下：

```
val hash = Utils.nonNegativeHash(filename)
val dirId = hash % localDirs.length
val subDirId = (hash / localDirs.length) % subDirsPerLocalDir
```

需要注意的是二级目录不一定存在，因此在返回前需要确认，如果不存在则需要创建：

```
var subDir = subDirs(dirId)(subDirId)
if (subDir == null) {
    subDir = subDirs(dirId).synchronized {
```

```
            val old = subDirs(dirId)(subDirId)
            if (old != null) {
                old
            } else {
                val newDir = new File(localDirs(dirId), "%02x".format(subDirId))
                if (!newDir.exists() && !newDir.mkdir()) {
                    throw new IOException(s"Failed to create local dir in $newDir.")
                }
                subDirs(dirId)(subDirId) = newDir
            newDir
            }
        }
    }
```

再看 DiskStore 的实现就比较容易理解了。回忆一下在 Hash Based Shuffle 中，为了避免引入大量文件造成的内存压力和随机 Disk IO 的性能问题，引入了 Shuffle Consolidate Files 机制。这里重要的是 FileSegment 的实现。那么如何读这个文件呢？答案就在这个函数的实现：

```
def getBytes(segment: FileSegment): Option[ByteBuffer] = {
    // 根据文件句柄，开始的offset和要读取的长度来读取文件
    getBytes(segment.file, segment.offset, segment.length)
}
```

8.2.4 org.apache.spark.storage.MemoryStore 实现详解

MemoryStore 实际上是使用一个 Hash Map 来保存 Block 的数据：

```
private val entries = new LinkedHashMap[BlockId, MemoryEntry](32, 0.75f, true)
```

entries 保存了 Block 的数据，因此 Block 的读取和写入都是围绕 entries 展开的。除了在内存紧张的情况下，新的缓存的加入可能需要将部分老的缓存清除的逻辑外，这部分的实现还是很简单的。

所有对外提供的写入的接口，最终都是通过调用 org.apache.spark.storage.MemoryStore 的私有函数 tryToPut 来完成的。tryToPut 会检查当前的内存是否可以容下当前的请求，如果可以，则放入这个 Hash Map，期间可能会淘汰老的缓存；如果内存超过了当前的最大内存，那么就会返回调用者，如果存储级别还有 Disk，那么会将当前的数据通过 DiskStore 写入 Disk，否则缓存将不会被持久化，下次用的话还需要

重新计算。

如何确定当前的内存足够呢？这个功能是在 org.apache.spark.storage.MemoryStore 的 ensureFreeSpace 实现的。

```
if (space > maxMemory) { //无法载入内存
    return ResultWithDroppedBlocks(success = false, droppedBlocks)
}
val actualFreeMemory = freeMemory - currentUnrollMemory
if (actualFreeMemory < space) {
    //如果可用内存不足以存储当期的Block，那么淘汰部分Block
    val rddToAdd = getRddId(blockIdToAdd)
    val selectedBlocks = new ArrayBuffer[BlockId]
    var selectedMemory = 0L

    // 保证这个过程中entries不会改变，否则同时遍历entrie时会导致异常
    entries.synchronized {
        val iterator = entries.entrySet().iterator()
        while (actualFreeMemory + selectedMemory < space && iterator.hasNext) {
            val pair = iterator.next()
            val blockId = pair.getKey
            if (rddToAdd.isEmpty || rddToAdd != getRddId(blockId)) {
                selectedBlocks += blockId //这个Block将会被移出内存
                selectedMemory += pair.getValue.size
            }
        }
    }
    if (actualFreeMemory + selectedMemory >= space) {
        for (blockId <- selectedBlocks) {
            //根据entries装入data，此处忽略一些代码
            val droppedBlockStatus = blockManager.dropFromMemory(blockId, data)
            droppedBlockStatus.foreach { status => droppedBlocks += ((blockId,
                status)) }
        }
    }
```

8.2.5　org.apache.spark.storage.TachyonStore 实现详解

TachyonStore 实现了 Spark 将缓存的数据放到 Tachyon 中。这个实现可以看作是实现了一个 Tachyon 客户端，通过这个客户端 Spark 可以读写 Tachyon 的数据。而 org.apache.spark.storage.TachyonStore 作为 BlockStore 的一个实现，并没

有复杂的逻辑，关于 Tachyon 文件的读取和写入都是通过 org.apache.spark.storage.TachyonBlockManager 来完成的。

TachyonStore 如何读取一个 Block：

```
override def getBytes(blockId: BlockId): Option[ByteBuffer] = {
    val file = tachyonManager.getFile(blockId)
    if (file == null || file.getLocationHosts.size == 0) {
        return None
    }
    val is = file.getInStream(ReadType.CACHE)
    try {
        val size = file.length
        val bs = new Array[Byte](size.asInstanceOf[Int])
        ByteStreams.readFully(is, bs)
        Some(ByteBuffer.wrap(bs))
    } catch {//省略了异常处理的代码
    } finally {
        is.close()
    }
}
```

与读一个 Block 类似，也是从 TachyonBlockManager 获取 Block 的文件句柄，然后就可以写入了。其中，它生成的目录布局和 org.apache.spark.storage.DiskBlockManager 有比较相似的风格，它为每个 Executor 生成不同的 root dir:

```
val storeDir = conf.get("spark.tachyonStore.baseDir", "/tmp_spark_tachyon")
val appFolderName = conf.get("spark.tachyonStore.folderName")
val tachyonStorePath = s"$storeDir/$appFolderName/${this.executorId}"
```

不过目录还是要加上 spark-tachyon- yyyyMMddHHmmss-xxxx，然后在这个目录下面还是会有 spark.tachyonStore.subDirectories（默认值也是 64）个数的子目录。

8.2.6 Block 存储的实现

上面分析了 DiskStore、MemoryStore 和 TachyonStore 的详细实现，那么 Block 是如何存储的呢？实际上，org.apache.spark.storage.BlockManager 对外屏蔽了存储实现的方式，而是通过用户传入存储级别来确定最终的持久化方案。接下来，将通过对 RDD 部分计算过程的分析来阐明这个实现过程。

如果用户设置的存储级别不是 StorageLevel.NONE，那么在计算的时候，会首先通过 org.apache.spark.CacheManager 来判断结果是否已经缓存，如果是，则直接读取缓存，否则开始计算，并且将计算的结果根据存储级别进行缓存。相关代码：

```
SparkEnv.get.cacheManager.getOrCompute(this, split, context, storageLevel)
```

如果 org.apache.spark.CacheManager 在缓存中没有找到计算需要的结果，那么它需要 RDD 开始计算：

```
val computedValues = rdd.computeOrReadCheckpoint(partition, context)
```

而计算的结果会通过调用 org.apache.spark.storage.BlockManager#putArray 写入缓存：

```
blockManager.putArray(key, arr, level, tellMaster = true, effectiveStorageLevel)
```

tellMaster = true 表示这个更新需要上报到 Master；而 effectiveStorageLevel 表示了存储级别。而 org.apache.spark.storage.BlockManager#putArray 会调用 org.apache.spark.storage.BlockManager#doPut。doPut 是一个 BlockManager 的私有接口，供自身调用，它会根据存储级别决定缓存存储的方式，比如是 Memory、Disk 还是 Tachyon；还有是否需要远程复制到其他的节点进行备份等。实际上 BlockManager 对外开放了三个接口来做数据的 put：

1）org.apache.spark.storage.BlockManager#putIterator

2）org.apache.spark.storage.BlockManager#putArray

3）org.apache.spark.storage.BlockManager#putBytes

这些接口都会调用 doPut 来完成最终的处理。对于需要远程复制的，不同的调用将会有不同的处理：

1）如果调用来自 org.apache.spark.storage.BlockManager#putBytes，那么在刚开始的时候就会启动复制的动作，因此这时候数据可以直接使用，这个由 doPut 传入数据的类型是否是 ByteBufferValues 来确定：

```
val replicationFuture = data match {
        case b: ByteBufferValues if putLevel.replication > 1 =>
    // Duplicate doesn't copy the bytes, but just creates a wrapper
```

```
        val bufferView = b.buffer.duplicate()
        Future { replicate(blockId, bufferView, putLevel) }
        case _ => null
    }
```

2）如果来自其他的调用，因为复制需要获取数据，而这个数据有可能还需要序列化，因此这里的做法是调用 BlockStore 去缓存数据，并且对于需要远程复制的请求，告知 BlockStore 调用者需要缓存数据，这样就可以通过这个数据进行远程复制了。以 org.apache.spark.storage.BlockStore#putIterator 为例，它的参数是 (blockId: BlockId, values: Iterator[Any], level: StorageLevel, returnValues: Boolean)，最后一个参数就是表明调用者需要缓存的数据，这个数据通过返回值 org.apache.spark.storage.PutResult 返回。

然后 BlockManager 会根据用户选择的存储级别来选择不同的 BlockStore，并且确定是否需要结果。returnValues 就是标记是否需要 BlockStore 返回结果。

```
val (returnValues, blockStore: BlockStore) = {
    if (putLevel.useMemory) {
        // Put it in memory first, even if it also has useDisk set to true;
        // We will drop it to disk later if the memory store can't hold it.
        (true, memoryStore)
    } else if (putLevel.useOffHeap) {
        // Use tachyon for off-heap storage
        (false, tachyonStore)
    } else if (putLevel.useDisk) {
        // Don't get back the bytes from put unless we replicate them
        (putLevel.replication > 1, diskStore)
    } else {
        assert(putLevel == StorageLevel.NONE)
        throw new BlockException(
            blockId, s"Attempted to put block $blockId without specifying
                storage level!")
    }
}
```

对于同时使用 Memory 和 Disk 的存储级别，Spark 并不一定会写 Disk，只有当 Memory 不够用时才会写 Disk。接下来会使用刚才通过存储级别获取的 BlockStore 开始真正的写操作：

```
val result = data match {
    case IteratorValues(iterator) =>
        blockStore.putIterator(blockId, iterator, putLevel, returnValues)
    case ArrayValues(array) =>
        blockStore.putArray(blockId, array, putLevel, returnValues)
    case ByteBufferValues(bytes) =>
        bytes.rewind()
        blockStore.putBytes(blockId, bytes, putLevel)
}
```

然后将状态汇报到 Master：

```
if (tellMaster) {
    reportBlockStatus(blockId, putBlockInfo, putBlockStatus)
}
```

前面介绍过，这个 reportBlockStatus 会调用 org.apache.spark.storage.BlockManagerMaster，而请求会通过网络转发到 org.apache.spark.storage.BlockManagerMasterActor，完成状态的汇报。

对于 ByteBufferValues 类型的数据，数据在被 BlockStore 处理前就已经开始复制到其他的节点，那么对于其他的类型（ArrayValues 和 IteratorValues）呢？

```
if (putLevel.replication > 1) {
    data match {
        case ByteBufferValues(bytes) =>
            if (replicationFuture != null) {
                Await.ready(replicationFuture, Duration.Inf)
            }
        case _ =>
            val remoteStartTime = System.currentTimeMillis
            // Serialize the block if not already done
            if (bytesAfterPut == null) {
                if (valuesAfterPut == null) {
                    throw new SparkException(
                        "Underlying put returned neither an Iterator nor
                            bytes! This shouldn't happen.")
                }
                bytesAfterPut = dataSerialize(blockId, valuesAfterPut)
            }
            replicate(blockId, bytesAfterPut, putLevel)
            logDebug("Put block %s remotely took %s"
```

```
                .format(blockId, Utils.getUsedTimeMs(remoteStartTime)))
    }
```

8.3 性能调优

8.3.1 spark.local.dir

这个目录用于写中间数据，如 RDD Cache、Shuffle 时存储数据的位置，在 8.2.3 节已经详细介绍了它的使用方式，那么，设置该目录时需要注意什么呢？

首先，最基本的是我们可以配置多个路径（用逗号分隔）到多个磁盘上增加整体 IO 带宽。

其次，在 8.2.3 小节中也提到过，一个逻辑的 Block 会根据它的 blockId 生成的名字映射到一个物理上的文件。这些物理文件会被 hash 到由 spark.local.dir（或者 SPARK_LOCAL_DIRS）设置的目录中。因此，如果存储设备的读写速度不一样，那么可以在较快的存储设备上配置更多的目录来增加它被使用的比例，从而更好地利用快速存储设备。在 Spark 能够感知具体的存储设备类型前，这个变通方法可以取得一个不错的效果。

此外，还需要注意的是，在 Spark 1.0 以后，SPARK_LOCAL_DIRS (Standalone, Mesos) 或者 LOCAL_DIRS (YARN) 参数会覆盖这个配置。比如 YARN 模式的时候，Spark Executor 的本地路径依赖于 YARN 的配置，而不取决于这个参数。

8.3.2 spark.executor.memory

本节与 6.3.1 节内容一致，这里不再赘述。

8.3.3 spark.storage.memoryFraction

如前文所说，spark.executor.memory 决定了每个 Executor 可用内存的大小，那么 spark.storage.memoryFraction 则决定了在这部分内存中有多少可以用于 Memory Store 管理 RDD Cache 数据，多少内存用来满足任务运行时各种其他内存空间的需要。关于

Memory Store 的实现原理可参阅 8.2.4 节。

spark.executor.memory 默认值为 0.6，官方文档建议这个比值不要超过 JVM Old Gen 区域的比值。这也很容易理解，因为 RDD Cache 数据通常都是长期驻留内存的，也就是说最终会被转移到 Old Gen 区域（如果该 RDD 还没有被删除的话），如果这部分数据允许的尺寸太大，势必把 Old Gen 区域占满，造成频繁的全量的垃圾回收。

如何调整这个比值，取决于你的应用对数据的使用模式和数据的规模，粗略地来说，如果频繁发生全量的垃圾回收，可以考虑降低这个比值，这样 RDD Cache 可用的内存空间减少（剩下的部分 Cache 数据就需要通过 Disk Store 写到磁盘上了），虽然会带来一定的性能损失，但是腾出更多的内存空间用于执行任务，减少全量的垃圾回收发生的次数，反而可能改善程序运行的整体性能。

8.3.4　spark.streaming.blockInterval

这个参数用来设置 Spark Streaming 里 Stream Receiver 生成 Block 的时间间隔，默认为 200ms。具体表现为具体的 Receiver 所接收的数据，以相同的时间间隔，同期性地从 Buffer 中生成一个 StreamBlock 放进队列，等待进一步被存储到 BlockManager 中供后续计算过程使用。从理论上来说，为了保证每个 StreamingBatch 间隔里的数据是均匀的，这个时间间隔应该能被 Batch 的间隔时间长度整除。总体来说，如果内存大小够用，Streaming 的数据来得及处理，这个时间间隔的影响不大，当然，如果数据存储级别是 Memory+Ser，即做了序列化处理，那么时间间隔的大小会影响序列化后数据块的大小，对于 Java 的垃圾回收的行为会有一些影响。

此外，spark.streaming.blockQueueSize 决定了在 StreamBlock 被存储到 BlockMananger 之前，队列中最多可以容纳多少个 StreamBlock。默认为 10，因为这个队列轮询的时间间隔是 100ms，所以如果 CPU 不是特别繁忙的话，基本上应该没有问题。

8.4　小结

Storage 模块负责管理 Spark 计算过程中产生的数据，包括基于 Disk 和基于

Memory 的数据。用户在实际编程中，面对的是 RDD，可以将 RDD 的数据通过调用 org.apache.spark.rdd.RDD#cache 将数据持久化；持久化的动作都是由 Storage 模块完成的。包括 Shuffle 过程中的数据，也都是由 Storage 模块管理的。可以说，RDD 实现了用户的逻辑，而 Storage 管理了用户的数据。

本章首先给出了 Storage 模块的整体架构，通过对 Master 端（即 Driver 端）和 Slave 端（即 Executor 端）的消息梳理，分析其在 Driver 端和 Executor 端的不同实现，其中着重分析了 DiskStore、MemoryStore 和 TachyonStore 的实现。最后通过对用户可以配置参数的详细介绍，解析了这些参数的实际作用，为用户在实际应用环境中的性能调优提供理论支撑。

第 9 章 Chapter 9

企业应用概述

9.1 Spark 在百度

百度构建了国内规模最大的 Spark 集群之一：实际生产环境，最大单集群规模 1300 台（包含数万核心和上百 TB 内存），公司内部同时还运行着大量的小型 Spark 集群。百度分布式计算团队从 2011 年开始持续关注 Spark，并于 2014 年将 Spark 正式引入百度分布式计算生态系统中，在国内率先面向开发者及企业用户推出了支持 Spark 并兼容开源接口的大数据处理产品 BMR。

9.1.1 现状

当前百度的 Spark 集群平均每天提交应用量多达数百，已应用于凤巢、大搜索、直达号、百度大数据等业务。Spark 的三个优点使之在百度得到快速发展：

1）快速高效。首先，Spark 使用了线程池模式，任务调度效率很高；其次，Spark 可以最大限度地利用内存，多轮迭代任务执行效率高。

2）API 友好易用。这主要基于两个方面：第一，Spark 支持多门编程语言，可以满足不同语言背景的人使用要求；第二，Spark 的表达能力非常丰富，并且封装了大量

常用操作。

3）组件丰富。Spark 生态圈当下已比较完善，在官方组件涵盖 SQL、图计算、机器学习和实时计算的同时，还有很多第三方开发的优秀组件，足以应对日常的数据处理需求。

9.1.2 百度开放云 BMR 的 Spark

BMR 的全称是 Baidu MapReduce，但是这个名称已经不能完全表示出这个平台：BMR 是百度开放云的数据分析服务产品。BMR 是基于百度多年大数据处理分析经验，面向企业和开发者提供按需部署的 Hadoop & Spark 集群计算服务，让客户具备海量数据分析和挖掘能力，从而提升业务竞争力。

如图 9-1 所示，BMR 基于 BCC（百度云服务器），建立在 HDFS 和 BOS（百度对象存储）分布式存储之上，其处理引擎包含了 MapReduce 和 Spark，同时还使用了 HBase 数据库。在此之上，系统集成了 Pig、Hive、SQL、Streaming、GraphX、MLLib 等专有服务。在系统的最上层，BMR 提供了一个基于 Web 的控制台，以及一个 API 形式的 SDK。Scheduler 在 BMR 中起到了管理作用，使用它可以编写比较复杂的作业流。

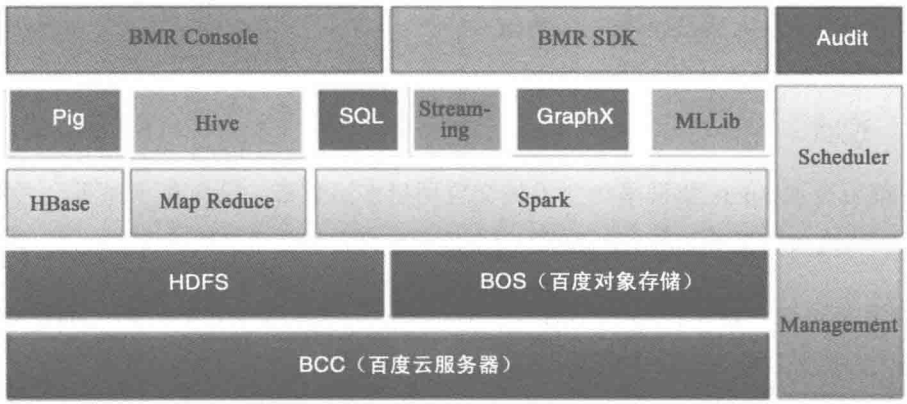


图 9-1 BMR 整体架构

类似于一般的云服务，BMR 中的 Spark 同样随用随起，集群空闲即销毁，帮助用户节省预算。此外，集群创建可以在 3 ～ 5 分钟内完成，包含了完整的 Spark+HDFS+YARN 堆栈。同时，BMR 也提供 Long Running 模式，并有多种套餐可选。

在安全上，用户拥有虚拟的独立网络，在同一用户全部集群可互联的同时，BMR 用户间网络被完全隔离。同时，BMR 还支持动态扩容，节点规模可弹性伸缩。除此之外，在实现 Spark 全组件支持的同时，BMR 可无缝对接百度对象存储（BOS）服务，借力百度多年的存储研发经验，保证数据存储的高可靠性。

9.1.3　在 Spark 中使用 Tachyon

在 Spark 使用过程中，用户经常困扰于 3 个问题：首先，两个 Spark 实例通过存储系统来共享数据，这个过程中对磁盘的操作会显著降低性能；其次，因为 Spark 崩溃所造成的数据丢失；最后，垃圾回收机制，如果两个 Spark 实例需要同样的数据，那么这个数据会被缓存两次，从而造成很大的内存压力，并降低性能。而在百度，计算集群和存储集群往往不在同一个地理位置的数据中心，在大数据分析时，远程数据读取将带来非常高的延时，特别是 ad-hoc 查询。因此，将 Tachyon 作为一个传输缓存层，百度通常会将之部署在计算集群上。首次查询时，数据会从远程存储取出，而在以后的查询中，数据就会从本地的 Tacnyon 上读取，从而大幅改善了延时问题，如图 9-2 所示。

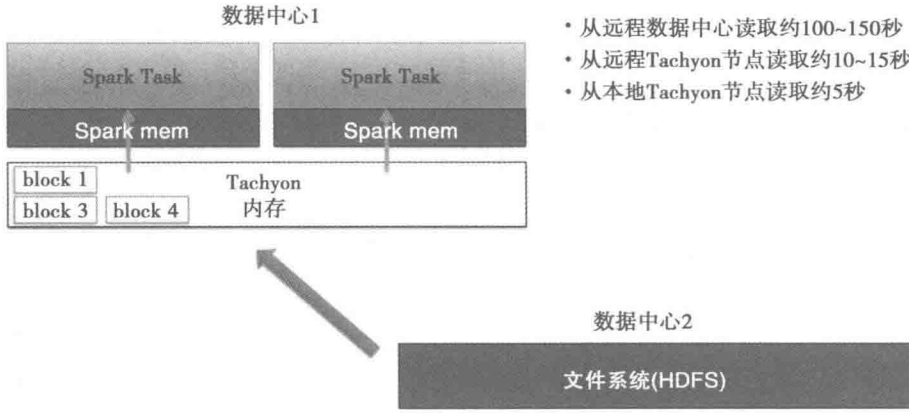


图 9-2　Tachyon 的整体架构

在百度，Tachyon 的部署还处于初始阶段，大约部署了 50 台机器，主要服务于 ad-hoc 查询。但是相信随着 Spark 在百度的快速发展，Tachyon 也一定会得到更加广泛的应用。

9.2 Spark 在阿里

阿里也是国内最早使用 Spark 的公司之一，同时也是最早在 Spark 中使用了 YARN 的公司之一。

值得一提的是，淘宝网络数据挖掘和计算团队负责人明风先生也是国内著名的 Spark 方面的专家。明风和他的团队针对淘宝的大数据和应用场景，在 MLlib、GraphX 和 Streaming 三大块进行了广泛的模型训练和生产应用，并且取得了很好的效果。

尤其是他们利用 GraphX 构建了大规模的图计算和图挖掘系统，实现了很多生产环境中的推荐算法，包括（但不局限于）以下的计算场景：基于度分布的中枢节点发现、基于最大连通图的社区发现、基于三角形计数的关系衡量、基于随机游走的用户属性传播等。可以说，淘宝技术团队在利用 Spark 来解决多次迭代的机器学习算法、高计算复杂度的算法方面，在国内居于领先的位置。

此外，阿里积极拥抱并回馈开源社区，对 Spark 社区的各个 Feature 和 PR 选择性地进行跟进和贡献。同时，阿里的内部版本和社区版本也保持同步性和一致性。阿里也在积极打造 Spark 周边的生产环境，包括 MLStudio 调度平台，使得 Spark 在阿里巴巴的应用更具推广性，可以满足大部分算法工程师和数据科学家的需求。

9.3 Spark 在腾讯

腾讯 Spark 集群已经达到 8000 台的规模，是当前已知的最大的 Spark 集群。每天运行了超过 10 000 个作业，作业类型包括 ETL、SparkSQL、机器学习、图计算和流式计算。

腾讯的广点通是最早使用 Spark 的应用之一。腾讯大数据精准推荐借助 Spark 快速迭代的优势，围绕“数据＋算法＋系统”这套技术方案，实现了在“数据实时采集、算法实时训练、系统实时预测”的全流程实时并行高维算法，最终成功应用于广点通 pCTR 投放系统上，支持每天上百亿的请求量。基于日志数据的快速查询系统业务构建于 Spark 之上的 Shark，利用其快速查询以及内存表等优势，承担了日志数据的实时查询工作。在性能方面，普遍比 Hive 高 2 ～ 10 倍，如果使用内存表的功能，性能将会比 Hive 高百倍。

此外，腾讯使用千台规模的 Spark 集群来对千亿量级的节点对进行相似度计算，通过实验对比，性能是 MapReduce 的 6 倍以上，是 GraphX 的 2 倍以上。相似度计算在信息检索、数据挖掘等领域有着广泛的应用，是目前推荐引擎中的重要组成部分。随着互联网用户数目和内容的爆炸性增长，对大规模数据进行相似度计算的需求变得日益增强。在传统的 MapReduce 框架下进行相似度计算会引入大量的网络开销，导致性能低下。腾讯借助 Spark 对内存计算的支持以及图划分的思想，大大降低了网络数据传输量；并通过在系统层次对 Spark 的改进优化，使其可以稳定地扩展至上千台规模。

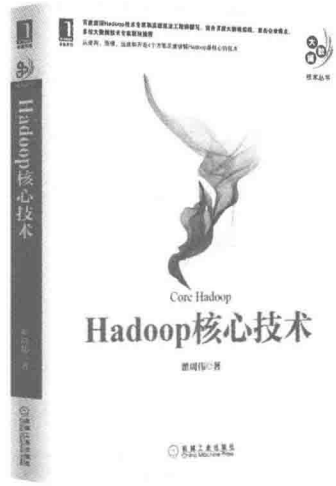
9.4　小结

Spark 在国内的应用越来越广泛，除了上面提到的百度、阿里和腾讯，现在京东、优酷土豆、携程等互联网公司也在使用 Spark。除了 Spark 的使用，华为、星环科技等公司也在做基于 Spark 的商业解决方案，尤其是星环科技，基于 Spark 做了很多二次开发。除此之外，英特尔上海有超过 10 个人的团队在做 Spark 的开发，是 Spark 非常重要的贡献者。

推荐阅读

Spark大数据处理：技术、应用与性能优化

作者：高彦杰 ISBN：978-7-111-48386-1 定价：59.00元



Hadoop核心技术

作者：翟周伟 ISBN：978-7-111-49468-3 定价：69.00元

大规模分布式系统架构与设计实战

作者：彭渊 ISBN：978-7-111-45503-5 定价：59.00元

大规模分布式存储系统：原理解析与架构实战

作者：杨传辉 ISBN：978-7-111-43052-0 定价：59.00元

在目前的大数据领域，Apache Spark无疑最受关注。越来越多的企业选择Spark作为大数据处理的核心框架。因此，除了熟悉Spark的编程模型之外，也非常有必要更加深入地理解Spark内核。本书详细剖析了Spark中的各个核心模块，并辅以相应的源码解析，使读者能更清晰地掌握Spark的来龙去脉。推荐Spark爱好者尤其是对Spark内核有兴趣的朋友阅读。

—— **陈超**（@CrazyJvm），七牛云技术总监，Spark布道者

在过去的两三年中，Spark凭借其简单灵活的编程接口以及高效的调度、计算和存储模型毫无争议地成为大数据领域中一个不可或缺的重要组成部分。随着Spark核心代码的日趋成熟，它在SQL / 实时计算 / 机器学习 / 图计算等应用领域也迸发出强大的生命力。但是万变不离其宗，只有掌握了Spark核心模块的流程原理，在应用开发过程中才能做到游刃有余。本书作者通过详尽的代码分析，详细阐述了Spark核心模块各个功能组件的设计思想和工作原理，相信能成为读者学习Spark过程中的良师益友。

—— **刘旭晖**，蘑菇街资深架构师，Spark贡献者

Apache Spark已经成为大数据领域炙手可热的未来之星，对于那些想要在生产环境中实施Spark技术方案的相关技术人员，深入理解Spark基本原理、运作机制和工程细节是必不可少的技能要求。本书对Apache Spark底层核心组件进行了细致讲解，相信可以帮助读者更好地钻研分析Apache Spark应用的核心技术，进而在更高层面把控和解决生产环境中的实际问题。

—— **柴华**，百度高级架构师，Spark布道者

Apache Spark已经大大简化了一体化大数据流水线的搭建，然而大数据领域固有的内在复杂性使得其研发、运维人员不仅需要掌握各基础组件的常规用法，还必须深入理解其内部运作机制乃至实现细节，这样才能对系统进行细致的调优、诊断和修正。本书针对Apache Spark的核心组件进行了细致剖析，相信可以令读者在钻研的路上事半功倍。

—— **连城**，DataBricks公司软件工程师

本书详细剖析了Spark核心引擎的源代码及其工作原理，内容翔实准确，也是我目前看到的一本比较全面解析Spark Core的不可多得的好书。特别是有志于Spark内核开发的研发人员，仔细阅读本书并研读代码，将起到事半功倍的效果。

—— **孙元浩**，星环科技创始人兼CTO

上架指导：计算机/大数据分析与处理
ISBN 978-7-111-50964-6
9 787111 509646 >
定价：49.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

[General Information]
书名=SPARK技术内幕　深入解析SPARK内核架构设计与实现原理
作者=张安站著
页数=202
SS号=13843983
DX号=
出版日期=2015.09
出版社=机械工业出版社