根据最新技术版本，系统、全面、详细讲解Spark的各项功能使用、原理机制、技术细节、应用方法、性能优化，以及BDAS生态系统的相关技术

华章科技

大数据技术丛书

Data Processing with Spark
Technology, Application and Performance Optimization

Spark大数据处理

技术、应用与性能优化

高彦杰◎著

机械工业出版社
China Machine Press

大数据技术丛书

Spark大数据处理：技术、应用与性能优化

高彦杰　著

ISBN：978-7-111-48386-1

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

前言

Spark是发源于美国加州大学伯克利分校AMPLab的大数据分析平台，它立足于内存计算，从多迭代批量处理出发，兼顾数据仓库、流处理和图计算等多种计算范式，是大数据系统领域的全栈计算平台。Spark当下已成为Apache基金会的顶级开源项目，拥有庞大的社区支持，技术也逐渐走向成熟。

为什么要写这本书

大数据还在如火如荼地发展着，突然之间，Spark就火了。还记得最开始接触Spark技术时资料匮乏，只有官方文档和源码可以作为研究学习的资料。写一本Spark系统方面的技术书籍，是我持续了很久的一个想法。由于学习和工作较为紧张，最初只是通过几篇笔记在博客中分享自己学习Spark过程的点滴，但是随着时间的推移，笔记不断增多，最终还是打算将笔记整理成书，也算是一个总结和分享。

在国外Yahoo！、Intel、Amazon、Cloudera等公司率先应用并推广Spark技术，在国内淘宝、腾讯、网易、星环等公司敢为人先，并乐于分享。在随后的发展中，IBM、MapR、Hortonworks、微策略等公司纷纷将Spark融进现有解决方案，并加入Spark阵营。Spark在工业界的应用也呈星火燎原之势。

随着Spark技术在国内的大范围落地、Spark中国峰会的召开，及各地meetup的火爆举行，开源软件Spark也因此水涨船高。随着大数据相关技术和产业的逐渐成熟，公司生产环境往往需要同时进行多种类型的大数据分析作业：批处理、各种机器学习、流式计算、图计算、SQL查询等。在Spark出现前，要在一个平台内同时完成以上数种大数据分析任务，就不得不与多套独立的系统打交道，这需要系统间进行代价较大的数据转储，但是这无疑会增加运维负担。

在1年之前，关注Spark的人和公司不多，由于它包含的软件种类多，版本升级较快，技术较为新颖，初学者难以在有限的时间内快速掌握Spark蕴含的价值。同时国内缺少一本实践与理论相结合的Spark书籍，很多Spark初学者和开发人员只能参考网络上零星的Spark技术相关博客，自己一点一滴地阅读源码和文档，缓慢地学习Spark。本书也正是为了解决上面的问题而编写的。

本书从一个系统化的视角，秉承大道至简的主导思想，介绍Spark中最值得关注的内容，讲解Spark部署、开发实战，并结合Spark的运行机制及拓展，帮读者开启Spark技术之旅。

本书特色

本书是国内首本系统讲解Spark编程实战的书籍，涵盖Spark技术的方方面面。

1）对Spark的架构、运行机制、系统环境搭建、测试和调优进行深入讲解，以期让读者知其所以然。讲述Spark最核心的技术内容，以激发读者的联想，进而衍化至繁。

2）实战部分不但给出编程示例，还给出可拓展的应用场景。

3）剖析BDAS生态系统的主要组件的原理和应用，让读者充分了解Spark生态系统。

本书的理论和实战安排得当，突破传统讲解方式，使读者读而不厌。

本书中一些讲解实操部署和示例的章节，比较适合作为运维和开发人员工作时手边的书；运行机制深入分析方面的章节，比较适合架构师和Spark研究人员，可帮他们拓展解决问题的思路。

读者对象

·Spark初学者

·Spark二次开发人员

·Spark应用开发人员

·Spark运维工程师

·开源软件爱好者

·其他对大数据技术感兴趣的人员

如何阅读本书

本书分为两大部分，共计9章内容。

第1章　从Spark概念出发，介绍了Spark的来龙去脉，阐述Spark生态系统全貌。

第2章　详细介绍了Spark在Linux集群和Windows上如何进行部署和安装。

第3章　详细介绍了Spark的计算模型，RDD的概念与原理，RDD上的函数算子的原理和使用，广播和累加变量。

第4章　详细介绍了Spark应用执行机制、Spark调度与任务分配、Spark I/O机制、Spark通信模块、容错机制、Shuffle机制，并对Spark机制原理进行了深入剖析。

第5章　从实际出发，详细介绍了如何在Intellij、Eclipse中配置开发环境，如何使用SBT构建项目，如何使用SparkShell进行交互式分析、远程调试和编译Spark源码，以及如何构建Spark源码阅读环境。

第6章　由浅入深，详细介绍了Spark的编程案例，通过WordCount、Top K到倾斜连接

等，以帮助读者快速掌握开发Spark程序的技巧。

第7章　展开介绍了主流的大数据Benchmark的原理，并对比了Benchmark优劣势，指导Spark系统性能测试和性能问题诊断。

第8章　围绕Spark生态系统，介绍了Spark之上的SQL on Spark、Spark Streaming、GraphX、MLlib的原理和应用。

第9章　详细介绍了如何对Spark进行性能调优，以及调优方法的原理。

如果您是一位有着一定经验的资深开发人员，能够理解Spark的相关基础知识和使用技巧，那么可以直接阅读4章、7章、8章、9章。如果你是一名初学者，请一定从第1章的基础知识开始学起。

勘误和支持

由于笔者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果您有更多的宝贵意见，欢迎访问我的个人Github上的Spark大数据处理专版：https://github.com/YanjieGao/SparkInAction，您可以将书中的错误提交PR或者进行评论，我会尽量在线上为读者提供最满意的解答。您也可以通过微博@高彦杰gyj、微信公共号@Spark大数据、博客http://blog.csdn.net/gaoyanjie55或者邮箱gaoyanjie55@163.com联系到我，期待能够得到读者朋友们的真挚反馈，在技术之路上互勉共进。

致谢

感谢中国人民大学的杜小勇老师、何军老师、陈跃国老师，是老师们将我带进大数据技术领域，教授我专业知识与学习方法，并在每一次迷茫时给予我鼓励与支持。

感谢微软亚洲研究院的闫莺老师和其他老师及同事，在实习工作中给予我的帮助和指导。

感谢IBM中国研究院的陈冠诚老师和其他老师及同事，在实习工作中给予我的帮助和指导。

感谢连城、明风、Daoyuan Wang等大牛以及Michael Armbrust、Reynold Xin、Sean Owen等多位社区大牛，在开发和技术学习中对我的点拨和指导，以及社区的各位技术专家们的博客文章。本书多处引用了他们的观点和思想。

感谢机械工业出版社华章公司的首席策划杨福川和编辑高婧雅，在近半年的时间中始终支持我的写作，给我鼓励和帮助，引导我顺利完成全部书稿。

特别致谢

最后，我要特别感谢我的女友蒋丹彤对我的支持，我为写作这本书，牺牲了很多陪伴你的时间。同时也要感谢你花了很大的精力帮助我进行书稿校对。正因为有了你的付出与支持，我才能坚持写下去。

感谢我的父母、姐姐，有了你们的帮助和支持，我才有时间和精力去完成全部写作。

谨以此书献给我最亲爱的家人，以及众多热爱大数据技术的朋友们！

第1章　Spark简介

本章主要介绍Spark大数据计算框架、架构、计算模型和数据管理策略及Spark在工业界的应用。围绕Spark的BDAS项目及其子项目进行了简要介绍。目前，Spark生态系统已经发展成为一个包含多个子项目的集合，其中包含SparkSQL、Spark Streaming、GraphX、MLlib等子项目，本章只进行简要介绍，后续章节再详细阐述。

1.1 Spark是什么

Spark是基于内存计算的大数据并行计算框架。Spark基于内存计算，提高了在大数据环境下数据处理的实时性，同时保证了高容错性和高可伸缩性，允许用户将Spark部署在大量廉价硬件之上，形成集群。

Spark于2009年诞生于加州大学伯克利分校AMPLab。目前，已经成为Apache软件基金会旗下的顶级开源项目。下面是Spark的发展历程。

1.Spark的历史与发展

·2009年：Spark诞生于AMPLab。

·2010年：开源。

·2013年6月：Apache孵化器项目。

·2014年2月：Apache顶级项目。

·2014年2月：大数据公司Cloudera宣称加大Spark框架的投入来取代MapReduce。

·2014年4月：大数据公司MapR投入Spark阵营，Apache Mahout放弃MapReduce，将使用Spark作为计算引擎。

·2014年5月：Pivotal Hadoop集成Spark全栈。

·2014年5月30日：Spark 1.0.0发布。

·2014年6月：Spark 2014峰会在旧金山召开。

·2014年7月：Hive on Spark项目启动。

目前AMPLab和Databricks负责整个项目的开发维护，很多公司，如Yahoo！、Intel等参与到Spark的开发中，同时很多开源爱好者积极参与Spark的更新与维护。

AMPLab开发以Spark为核心的BDAS时提出的目标是：one stack to rule them all，也就是说在一套软件栈内完成各种大数据分析任务。相对于MapReduce上的批量计算、迭代型计算以及基于Hive的SQL查询，Spark可以带来上百倍的性能提升。目前Spark的生态系统日趋完善，Spark SQL的发布、Hive on Spark项目的启动以及大量大数据公司对Spark全栈的支持，让Spark的数据分析范式更加丰富。

2.Spark之于Hadoop

更准确地说，Spark是一个计算框架，而Hadoop中包含计算框架MapReduce和分布式文件系统HDFS，Hadoop更广泛地说还包括在其生态系统上的其他系统，如Hbase、Hive等。

Spark是MapReduce的替代方案，而且兼容HDFS、Hive等分布式存储层，可融入Hadoop的生态系统，以弥补缺失MapReduce的不足。

Spark相比Hadoop MapReduce的优势[1]如下。

（1）中间结果输出

基于MapReduce的计算引擎通常会将中间结果输出到磁盘上，进行存储和容错。出于任务管道承接的考虑，当一些查询翻译到MapReduce任务时，往往会产生多个Stage，而这些串联的Stage又依赖于底层文件系统（如HDFS）来存储每一个Stage的输出结果。

Spark将执行模型抽象为通用的有向无环图执行计划（DAG），这可以将多Stage的任务串联或者并行执行，而无须将Stage中间结果输出到HDFS中。类似的引擎包括Dryad、Tez。

（2）数据格式和内存布局

由于MapReduce Schema on Read处理方式会引起较大的处理开销。Spark抽象出分布式内存存储结构弹性分布式数据集RDD，进行数据的存储。RDD能支持粗粒度写操作，但对于读取操作，RDD可以精确到每条记录，这使得RDD可以用来作为分布式索引。Spark的特性是能够控制数据在不同节点上的分区，用户可以自定义分区策略，如Hash分区等。Shark和Spark SQL在Spark的基础之上实现了列存储和列存储压缩。

（3）执行策略

MapReduce在数据Shuffle之前花费了大量的时间来排序，Spark则可减轻上述问题带来的开销。因为Spark任务在Shuffle中不是所有情景都需要排序，所以支持基于Hash的分布式聚合，调度中采用更为通用的任务执行计划图（DAG），每一轮次的输出结果在内存缓存。

（4）任务调度的开销

传统的MapReduce系统，如Hadoop，是为了运行长达数小时的批量作业而设计的，在某些极端情况下，提交一个任务的延迟非常高。

Spark采用了事件驱动的类库AKKA来启动任务，通过线程池复用线程来避免进程或线程启动和切换开销。

3.Spark能带来什么

Spark的一站式解决方案有很多的优势，具体如下。

（1）打造全栈多计算范式的高效数据流水线

Spark支持复杂查询。在简单的“map”及“reduce”操作之外，Spark还支持SQL查询、流式计算、机器学习和图算法。同时，用户可以在同一个工作流中无缝搭配这些计算范式。

（2）轻量级快速处理

Spark 1.0核心代码只有4万行。这是由于Scala语言的简洁和丰富的表达力，以及Spark充分利用和集成Hadoop等其他第三方组件，同时着眼于大数据处理，数据处理速度是至关重要的，Spark通过将中间结果缓存在内存减少磁盘I/O来达到性能的提升。

（3）易于使用，Spark支持多语言

Spark支持通过Scala、Java及Python编写程序，这允许开发者在自己熟悉的语言环境下进行工作。它自带了80多个算子，同时允许在Shell中进行交互式计算。用户可以利用Spark像书写单机程序一样书写分布式程序，轻松利用Spark搭建大数据内存计算平台并充分利用内存计算，实现海量数据的实时处理。

（4）与HDFS等存储层兼容

Spark可以独立运行，除了可以运行在当下的YARN等集群管理系统之外，它还可以读取已有的任何Hadoop数据。这是个非常大的优势，它可以运行在任何Hadoop数据源上，如Hive、HBase、HDFS等。这个特性让用户可以轻易迁移已有的持久化层数据。

（5）社区活跃度高

Spark起源于2009年，当下已有超过50个机构、260个工程师贡献过代码。开源系统的发展不应只看一时之快，更重要的是支持一个活跃的社区和强大的生态系统。

同时我们也应该看到Spark并不是完美的，RDD模型适合的是粗粒度的全局数据并行计算。不适合细粒度的、需要异步更新的计算。对于一些计算需求，如果要针对特定工作负载达到最优性能，还是需要使用一些其他的大数据系统。例如，图计算领域的GraphLab在特定计算负载性能上优于GraphX，流计算中的Storm在实时性要求很高的场合要比Spark Streaming更胜一筹。

随着Spark发展势头日趋迅猛，它已被广泛应用于Yahoo！、Twitter、阿里巴巴、百度、网易、英特尔等各大公司的生产环境中。

[1] 参见论文：Reynold Shi Xin，Joshua Rosen，Matei Zaharia，Michael Franklin，Scott Shenker，Ion Stoica Shark:SQL and Rich Analytics at Scale。

1.2 Spark生态系统BDAS

目前，Spark已经发展成为包含众多子项目的大数据计算平台。伯克利将Spark的整个生态系统称为伯克利数据分析栈（BDAS）。其核心框架是Spark，同时BDAS涵盖支持结构化数据SQL查询与分析的查询引擎Spark SQL和Shark，提供机器学习功能的系统MLbase及底层的分布式机器学习库MLlib、并行图计算框架GraphX、流计算框架Spark Streaming、采样近似计算查询引擎BlinkDB、内存分布式文件系统Tachyon、资源管理框架Mesos等子项目。这些子项目在Spark上层提供了更高层、更丰富的计算范式。

图1-1为BDAS的项目结构图。

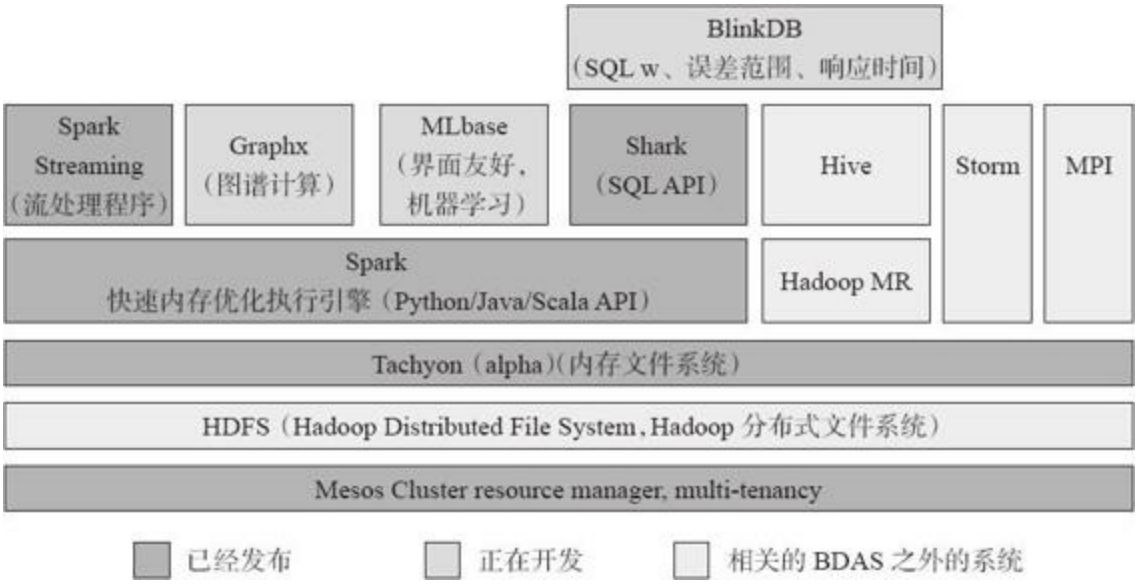

图1-1　伯克利数据分析栈（BDAS）项目结构图

下面对BDAS的各个子项目进行更详细的介绍。

（1）Spark

Spark是整个BDAS的核心组件，是一个大数据分布式编程框架，不仅实现了MapReduce的算子map函数和reduce函数及计算模型，还提供更为丰富的算子，如filter、join、groupByKey等。Spark将分布式数据抽象为弹性分布式数据集（RDD），实现了应用任务调度、RPC、序列化和压缩，并为运行在其上的上层组件提供API。其底层采用Scala这种函数

式语言书写而成，并且所提供的API深度借鉴Scala函数式的编程思想，提供与Scala类似的编程接口。

图1-2为Spark的处理流程（主要对象为RDD）。

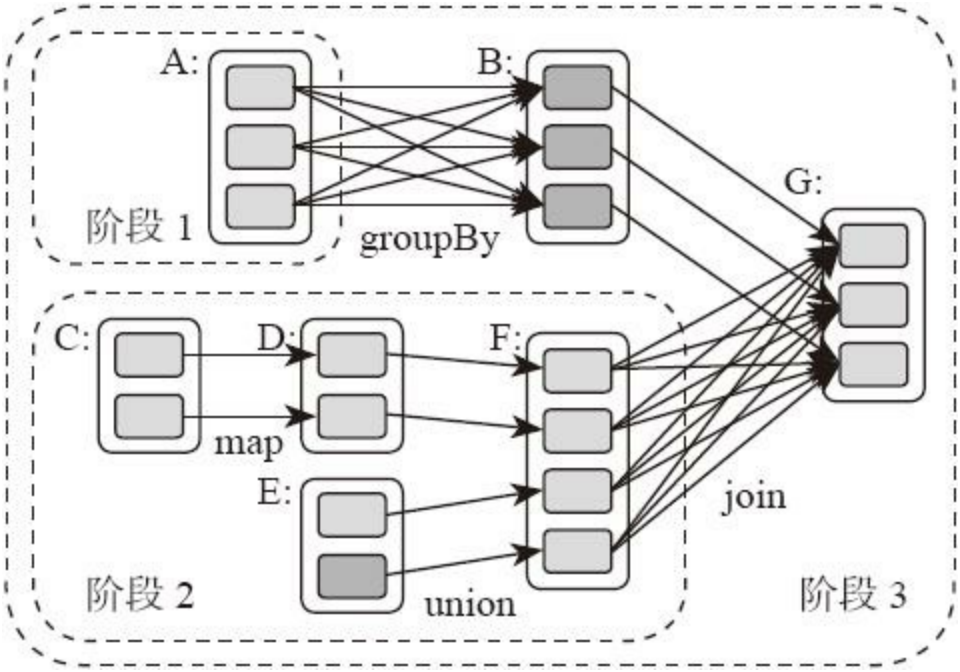

图1-2　Spark的任务处理流程图

Spark将数据在分布式环境下分区，然后将作业转化为有向无环图（DAG），并分阶段进行DAG的调度和任务的分布式并行处理。

（2）Shark

Shark是构建在Spark和Hive基础之上的数据仓库。目前，Shark已经完成学术使命，终止开发，但其架构和原理仍具有借鉴意义。它提供了能够查询Hive中所存储数据的一套SQL接口，兼容现有的Hive QL语法。这样，熟悉Hive QL或者SQL的用户可以基于Shark进行快速的Ad-Hoc、Reporting等类型的SQL查询。Shark底层复用Hive的解析器、优化器以及元数据存储和序列化接口。Shark会将Hive QL编译转化为一组Spark任务，进行分布式运算。

（3）Spark SQL

Spark SQL提供在大数据上的SQL查询功能，类似于Shark在整个生态系统的角色，它们

可以统称为SQL on Spark。之前，Shark的查询编译和优化器依赖于Hive，使得Shark不得不维护一套Hive分支，而Spark SQL使用Catalyst做查询解析和优化器，并在底层使用Spark作为执行引擎实现SQL的Operator。用户可以在Spark上直接书写SQL，相当于为Spark扩充了一套SQL算子，这无疑更加丰富了Spark的算子和功能，同时Spark SQL不断兼容不同的持久化存储（如HDFS、Hive等），为其发展奠定广阔的空间。

（4）Spark Streaming

Spark Streaming通过将流数据按指定时间片累积为RDD，然后将每个RDD进行批处理，进而实现大规模的流数据处理。其吞吐量能够超越现有主流流处理框架Storm，并提供丰富的API用于流数据计算。

（5）GraphX

GraphX基于BSP模型，在Spark之上封装类似Pregel的接口，进行大规模同步全局的图计算，尤其是当用户进行多轮迭代时，基于Spark内存计算的优势尤为明显。

（6）Tachyon

Tachyon是一个分布式内存文件系统，可以理解为内存中的HDFS。为了提供更高的性能，将数据存储剥离Java Heap。用户可以基于Tachyon实现RDD或者文件的跨应用共享，并提供高容错机制，保证数据的可靠性。

（7）Mesos

Mesos是一个资源管理框架[1]，提供类似于YARN的功能。用户可以在其中插件式地运行Spark、MapReduce、Tez等计算框架的任务。Mesos会对资源和任务进行隔离，并实现高效的资源任务调度。

（8）BlinkDB

BlinkDB是一个用于在海量数据上进行交互式SQL的近似查询引擎。它允许用户通过在查询准确性和查询响应时间之间做出权衡，完成近似查询。其数据的精度被控制在允许的误差范围内。为了达到这个目标，BlinkDB的核心思想是：通过一个自适应优化框架，随着时间的推移，从原始数据建立并维护一组多维样本；通过一个动态样本选择策略，选择一个适当大小的示例，然后基于查询的准确性和响应时间满足用户查询需求。

[1] Spark自带的资源管理框架是Standalone。

1.3 Spark架构

从上文介绍可以看出，Spark是整个BDAS的核心。生态系统中的各个组件通过Spark来实现对分布式并行任务处理的程序支持。

1.Spark的代码结构

图1-3展示了Spark-1.0的代码结构和代码量（不包含Test和Sample代码），读者可以通过代码架构对Spark的整体组件有一个初步了解，正是这些代码模块构成了Spark架构中的各个组件，同时读者可以通过代码模块的脉络阅读与剖析源码，这对于了解Spark的架构和实现细节都是很有帮助的。

下面对图1-3中的各模块进行简要介绍。

scheduler：文件夹中含有负责整体的Spark应用、任务调度的代码。

broadcast：含有Broadcast（广播变量）的实现代码，API中是Java和Python API的实现。

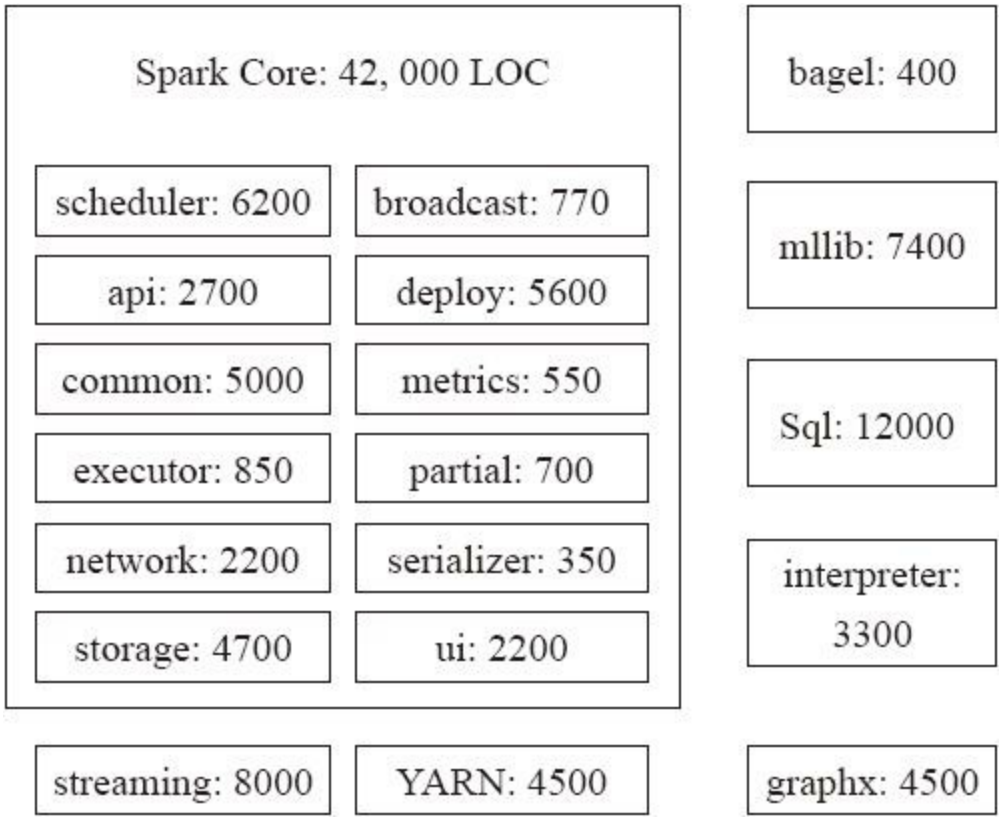

图1-3　Spark代码结构和代码量

deploy：含有Spark部署与启动运行的代码。

common：不是一个文件夹，而是代表Spark通用的类和逻辑实现，有5000行代码。

metrics：是运行时状态监控逻辑代码，Executor中含有Worker节点负责计算的逻辑代码。

partial：含有近似评估代码。

network：含有集群通信模块代码。

serializer：含有序列化模块的代码。

storage：含有存储模块的代码。

ui：含有监控界面的代码逻辑。其他的代码模块分别是对Spark生态系统中其他组件的实现。

streaming：是Spark Streaming的实现代码。

YARN：是Spark on YARN的部分实现代码。

graphx：含有GraphX实现代码。

interpreter：代码交互式Shell的代码量为3300行。

mllib：代表MLlib算法实现的代码量。

sql代表Spark SQL的代码量。

2.Spark的架构

Spark架构采用了分布式计算中的Master-Slave模型。Master是对应集群中的含有Master进程的节点，Slave是集群中含有Worker进程的节点。Master作为整个集群的控制器，负责整个集群的正常运行；Worker相当于是计算节点，接收主节点命令与进行状态汇报；Executor负责任务的执行；Client作为用户的客户端负责提交应用，Driver负责控制一个应用的执行，如图1-4所示。

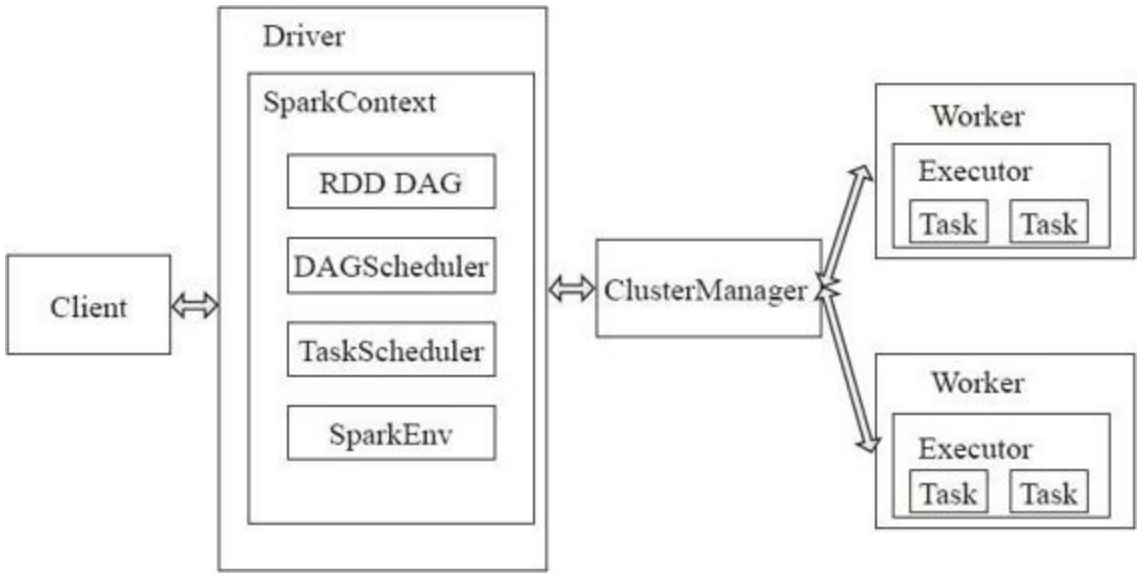

图1-4　Spark架构图

Spark集群部署后，需要在主节点和从节点分别启动Master进程和Worker进程，对整个

集群进行控制。在一个Spark应用的执行过程中，Driver和Worker是两个重要角色。Driver程序是应用逻辑执行的起点，负责作业的调度，即Task任务的分发，而多个Worker用来管理计算节点和创建Executor并行处理任务。在执行阶段，Driver会将Task和Task所依赖的file和jar序列化后传递给对应的Worker机器，同时Executor对相应数据分区的任务进行处理。

下面详细介绍Spark的架构中的基本组件。

·ClusterManager：在Standalone模式中即为Master（主节点），控制整个集群，监控Worker。在YARN模式中为资源管理器。

·Worker：从节点，负责控制计算节点，启动Executor或Driver。在YARN模式中为NodeManager，负责计算节点的控制。

·Driver：运行Application的main（）函数并创建SparkContext。

·Executor：执行器，在worker node上执行任务的组件、用于启动线程池运行任务。每个Application拥有独立的一组Executors。

·SparkContext：整个应用的上下文，控制应用的生命周期。

·RDD：Spark的基本计算单元，一组RDD可形成执行的有向无环图RDD Graph。

·DAG Scheduler：根据作业（Job）构建基于Stage的DAG，并提交Stage给TaskScheduler。

·TaskScheduler：将任务（Task）分发给Executor执行。

·SparkEnv：线程级别的上下文，存储运行时的重要组件的引用。

SparkEnv内创建并包含如下一些重要组件的引用。

·MapOutPutTracker：负责Shuffle元信息的存储。

·BroadcastManager：负责广播变量的控制与元信息的存储。

·BlockManager：负责存储管理、创建和查找块。

·MetricsSystem：监控运行时性能指标信息。

·SparkConf：负责存储配置信息。

Spark的整体流程为：Client提交应用，Master找到一个Worker启动Driver，Driver向Master或者资源管理器申请资源，之后将应用转化为RDD Graph，再由DAGScheduler将RDD Graph转化为Stage的有向无环图提交给TaskScheduler，由TaskScheduler提交任务给Executor执行。在任务执行的过程中，其他组件协同工作，确保整个应用顺利执行。

3.Spark运行逻辑

如图1-5所示，在Spark应用中，整个执行流程在逻辑上会形成有向无环图（DAG）。Action算子触发之后，将所有累积的算子形成一个有向无环图，然后由调度器调度该图上的任务进行运算。Spark的调度方式与MapReduce有所不同。Spark根据RDD之间不同的依赖关系切分形成不同的阶段（Stage），一个阶段包含一系列函数执行流水线。图中的A、B、C、D、E、F分别代表不同的RDD，RDD内的方框代表分区。数据从HDFS输入Spark，形成RDD A和RDD C，RDD C上执行map操作，转换为RDD D，RDD B和RDD E执行join操作，转换为F，而在B和E连接转化为F的过程中又会执行Shuffle，最后RDD F通过函数saveAsSequenceFile输出并保存到HDFS中。

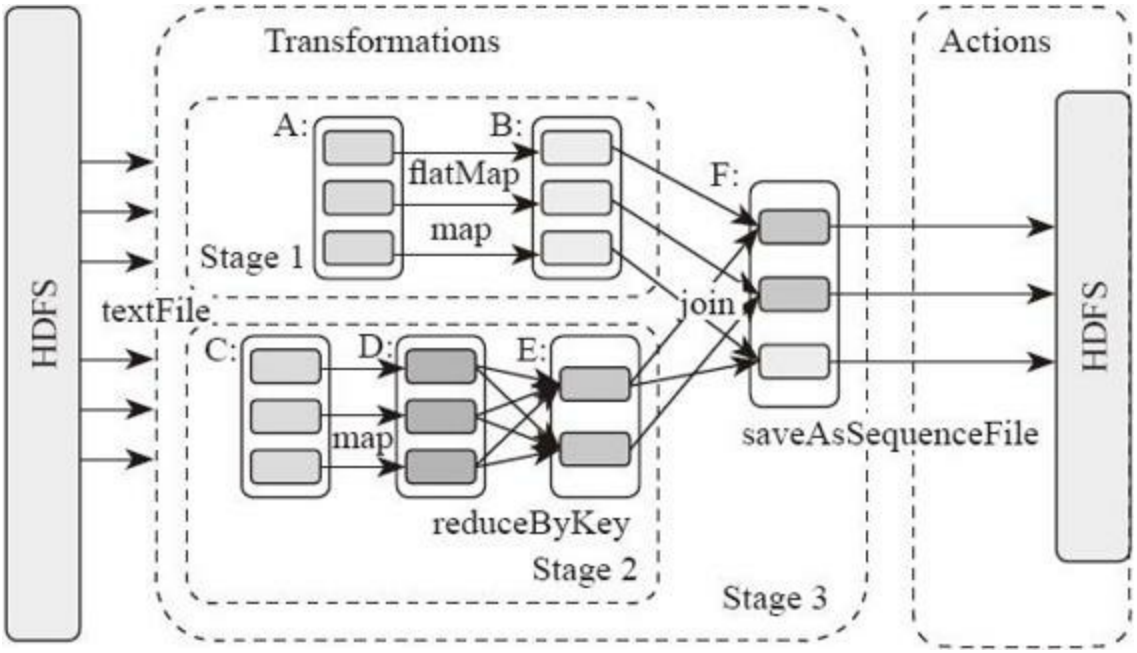

图1-5　Spark执行有向无环图

1.4 Spark分布式架构与单机多核架构的异同

我们通常所说的分布式系统主要指的是分布式软件系统，它是在通信网络互连的多处理机的架构上执行任务的软件系统，包括分布式操作系统、分布式程序设计语言、分布式文件系统和分布式数据库系统等。Spark是分布式软件系统中的分布式计算框架，基于Spark可以编写分布式计算程序和软件。为了整体宏观把握和理解分布式系统，可以将一个集群视为一台计算机。分布式计算框架的最终目的是方便用户编程，最后达到像原来编写单机程序一样编写分布式程序。但是分布式编程与编写单机程序还是存在不同点的。由于分布式架构和单机的架构有所不同，存在内存和磁盘的共享问题，这也是我们在书写和优化程序的过程中需要注意的地方。分布式架构与单机架构的对比如图1-6所示。

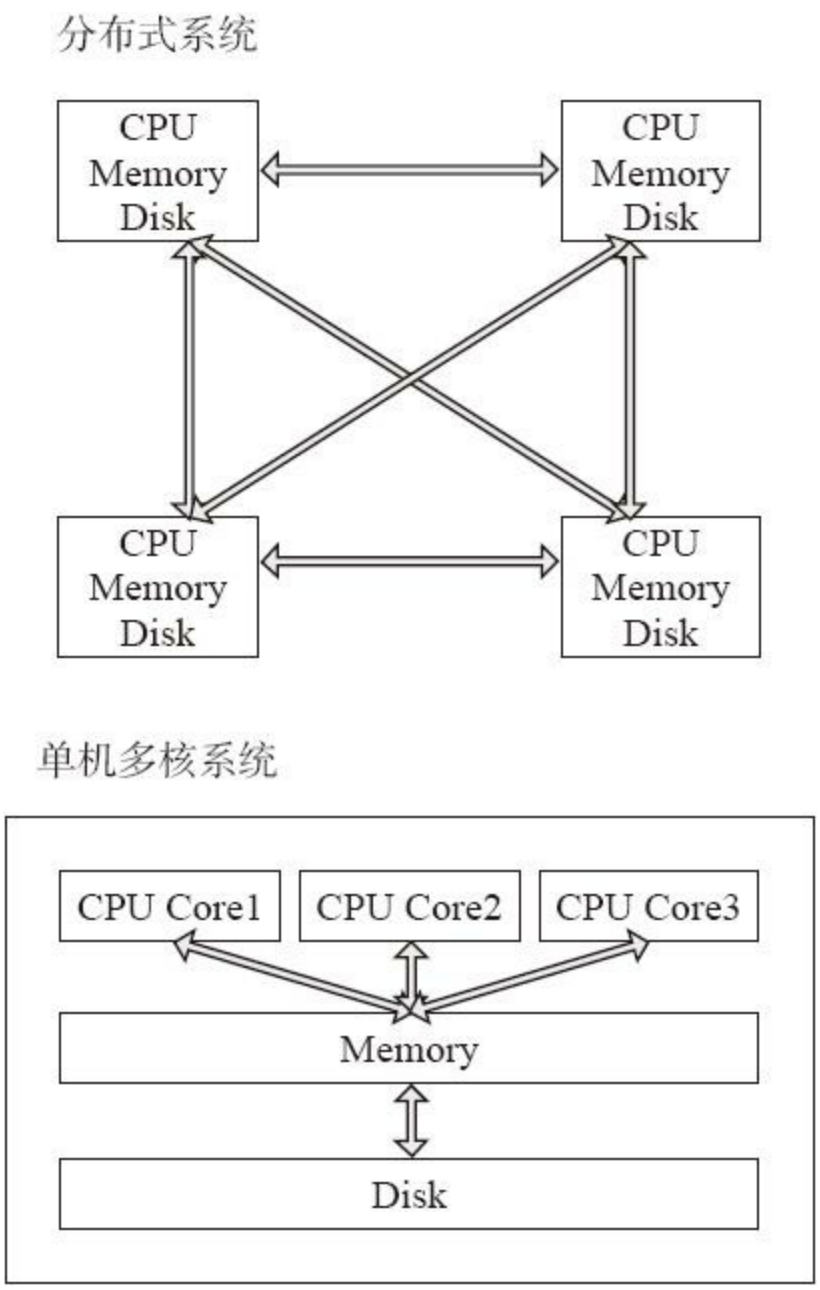

图1-6　分布式体系结构与单机体系结构的对比

1）在单机多核环境下，多CPU共享内存和磁盘。当系统所需的计算和存储资源不够，需要扩展CPU和存储时，单机多核系统显得力不从心。

2）大规模分布式并行处理系统是由许多松耦合的处理单元组成的，要注意的是，这里指的是处理单元而非处理器。每个单元内的CPU都有自己私有的资源，如总线、内存、硬盘等。这种结构最大的特点在于不共享资源。在不共享资源（Share Nothing）的分布式架构下，节点可以实现无限扩展，即计算能力和存储的扩展性可以成倍增长。

在分布式运算下，数据尽量本地运算，减少网络I/O开销。由于大规模分布式系统要在

不同处理单元之间传送信息，在网络传输少时，系统可以充分发挥资源的优势，达到高效率。也就是说，如果操作相互之间没有什么关系，处理单元之间需要进行的通信比较少，则采用分布式系统更好。因此，分布式系统在决策支持（DSS）和数据挖掘（Data Mining）方面具有优势。

Spark正是基于大规模分布式并行架构开发，因此能够按需进行计算能力与存储能力的扩展，在应对大数据挑战时显得游刃有余，同时保证容错性，让用户放心地进行大数据分析。

1.5 Spark的企业级应用

随着企业数据量的增长，对大数据的处理和分析已经成为企业的迫切需求。Spark作为Hadoop的替代者，引起学术界和工业界的普遍兴趣，大量应用在工业界落地，许多科研院校开始了对Spark的研究。

在学术界，Spark得到各院校的关注。Spark源自学术界，最初是由加州大学伯克利分校的AMPLab设计开发。国内的中科院、中国人民大学、南京大学、华东师范大学等也开始对Spark展开相关研究。涉及Benchmark、SQL、并行算法、性能优化、高可用性等多个方面。

在工业界，Spark已经在互联网领域得到广泛应用。互联网用户群体庞大，需要存储大数据并进行数据分析，Spark能够支持多范式的数据分析，解决了大数据分析中迫在眉睫的问题。例如，国外Cloudera、MapR等大数据厂商全面支持Spark，微策略等老牌BI厂商也和Databricks达成合作关系，Yahoo！使用Spark进行日志分析并积极回馈社区，Amazon在云端使用Spark进行分析。国内同样得到很多公司的青睐，淘宝构建Spark on Yarn进行用户交易数据分析，使用GraphX进行图谱分析。网易用Spark和Shark对海量数据进行报表和查询。腾讯使用Spark进行精准广告推荐。

下面将选取代表性的Spark应用案例进行分析，以便于读者了解Spark在工业界的应用状况。

1.5.1 Spark在Amazon中的应用

亚马逊云计算服务AWS（Amazon Web Services）提供IaaS和PaaS服务。Heroku、Netflix等众多知名公司都将自己的服务托管其上。AWS以Web服务的形式向企业提供IT基础设施服务，现在通常称为云计算。云计算的主要优势是能够根据业务发展扩展的较低可变成本替代前期资本基础设施费用。利用云，企业无须提前数周或数月来计划和采购服务器及其他IT基础设施，即可在几分钟内即时运行成百上千台服务器，并更快达成结果。

1.亚马逊AWS云服务的内容

目前亚马逊在EMR中提供了弹性Spark服务，用户可以按需动态分配Spark集群计算节点，随着数据规模的增长，扩展自己的Spark数据分析集群，同时在云端的Spark集群可以无缝集成亚马逊云端的其他组件，一起构建数据分析流水线。

亚马逊云计算服务AWS提供的服务包括：亚马逊弹性计算云（Amazon EC2）、亚马逊简单存储服务（Amazon S3）、亚马逊弹性MapReduce（Amazon EMR）、亚马逊简单数据库（Amazon SimpleDB）、亚马逊简单队列服务（Amazon Simple Queue Service）、Amazon DynamoDB以及Amazon CloudFront等。基于以上的组件，亚马逊开始提供EMR上的弹性Spark服务。用户可以像之前使用EMR一样在亚马逊动态申请计算节点，可随着数据量和计算需求来动态扩展计算资源，将计算能力水平扩展，按需进行大数据分析。亚马逊提供的云服务中已经支持使用Spark集群进行大数据分析。数据可以存储在S3或者Hadoop存储层，通过Spark将数据加载进计算集群进行复杂的数据分析。

亚马逊AWS架构如图1-7所示。

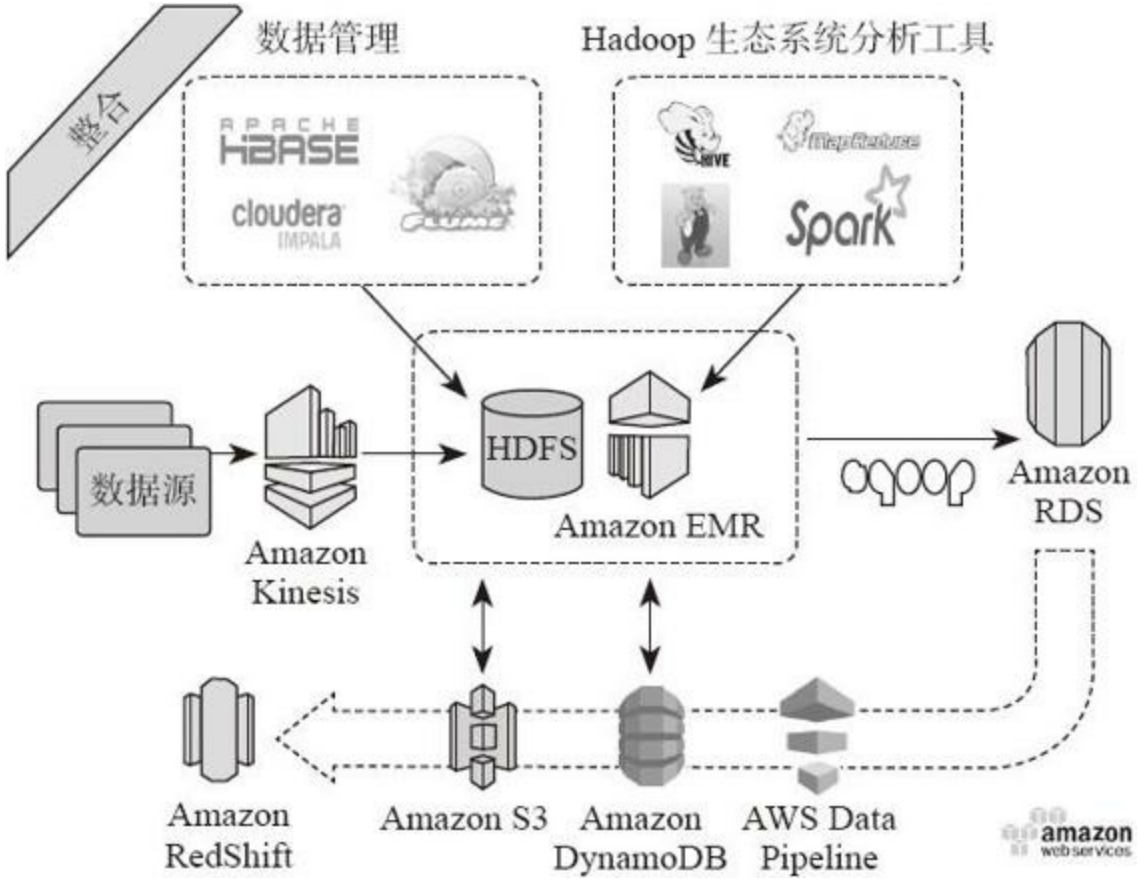

图1-7　亚马逊AWS架构

2.亚马逊的EMR中提供的3种主要组件

·Master Node：主节点，负责整体的集群调度与元数据存储。

·Core Node：Hadoop节点，负责数据的持久化存储，可以动态扩展资源，如更多的CPU Core、更大的内存、更大的HDFS存储空间。为了防止HDFS损坏，不能移除Core Node。

·Task Node：Spark计算节点，负责执行数据分析任务，不提供HDFS，只负责提供计算资源（CPU和内存），可以动态扩展资源，可以增加和移除Task Node。

3.使用Spark on Amazon EMR的优势

·构建速度快：可以在几分钟内构建小规模或者大规模Spark集群，以进行数据分析。

·运维成本低：EMR负责整个集群的管理与控制，EMR也会负责失效节点的恢复。

·云生态系统数据处理组件丰富：Spark集群可以很好地与Amazon云服务上的其他组件无缝集成，利用其他组件构建数据分析管道。例如，Spark可以和EC2 Spot Market、Amazon Redshift、Amazon Data pipeline、Amazon CloudWatch等组合使用。

·方便调试：Spark集群的日志可以直接存储到Amazon S3中，方便用户进行日志分析。

综合以上优势，用户可以真正按需弹性使用与分配计算资源，实现节省计算成本、减轻运维压力，在几分钟内构建自己的大数据分析平台。

4.Spark on Amazon EMR架构解析

通过图1-8可以看到整个Spark on Amazon EMR的集群架构。下面以图1-8为例，分析用户如何在应用场景使用服务。

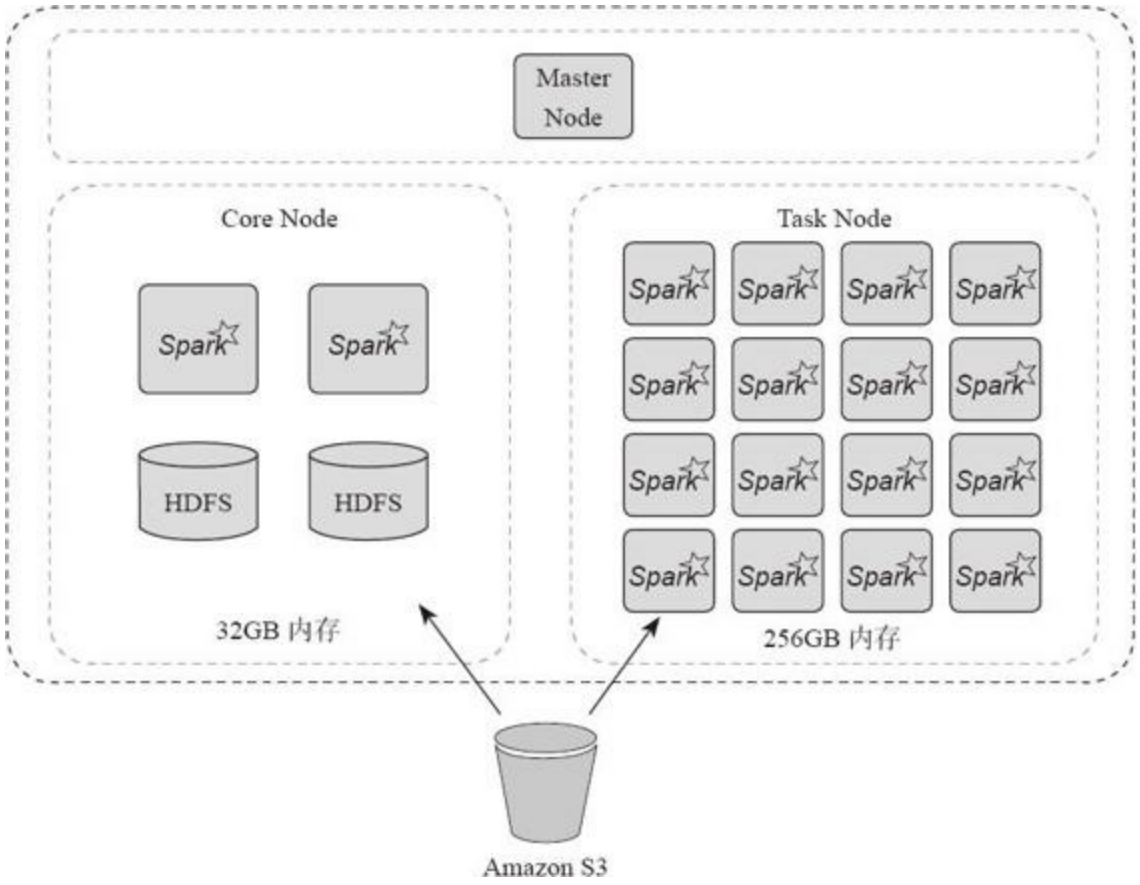

图1-8　Amazon Spark on EMR

构建集群，首先创建一个Master Node作为集群的主节点。之后创建两个Core Node存储数据，两个Core Node总共有32GB的内存。但是这些内存是不够Spark进行内存计算的。

接下来动态申请16个Task Node，总共256GB内存作为计算节点，进行Spark的数据分析。

当用户开始分析数据时，Spark RDD的输入既可以来自Core Node中的HDFS，也可以来自Amazon S3，还可以通过输入数据创建RDD。用户在RDD上进行各种计算范式的数据分析，最终可以将分析结果输出到Core Node的HDFS中，也可以输出到Amazon S3中。

5.应用案例：构建1000个节点的Spark集群

读者可以通过下面的步骤，在Amazon EMR上构建自己的1000个节点的Spark数据分析平台。

1）启动1000个节点的集群，这个过程将会花费10~20分钟。

```
./elas2c-mapreduce    --create     -alive
--name     "Spark/Shark      Cluster"               \
--bootstrap-ac2on
s3：//elasBcmapreduce/samples/spark/0.8.1/install-spark-shark.sh
--bootstrap-name "Spark/Shark"
--instance-type   m1.xlarge
--instance-count 1000
```

2）如果希望继续动态增加计算资源，可以输入下面命令增加Task Node。

```
--add-instance-group TASK
--instance-count  INSTANCE_COUNT
--instance-type  INSTANCE_TYPE
```

执行完步骤1）或者1）、2）后，集群将会处于图1-9所示的等待状态。

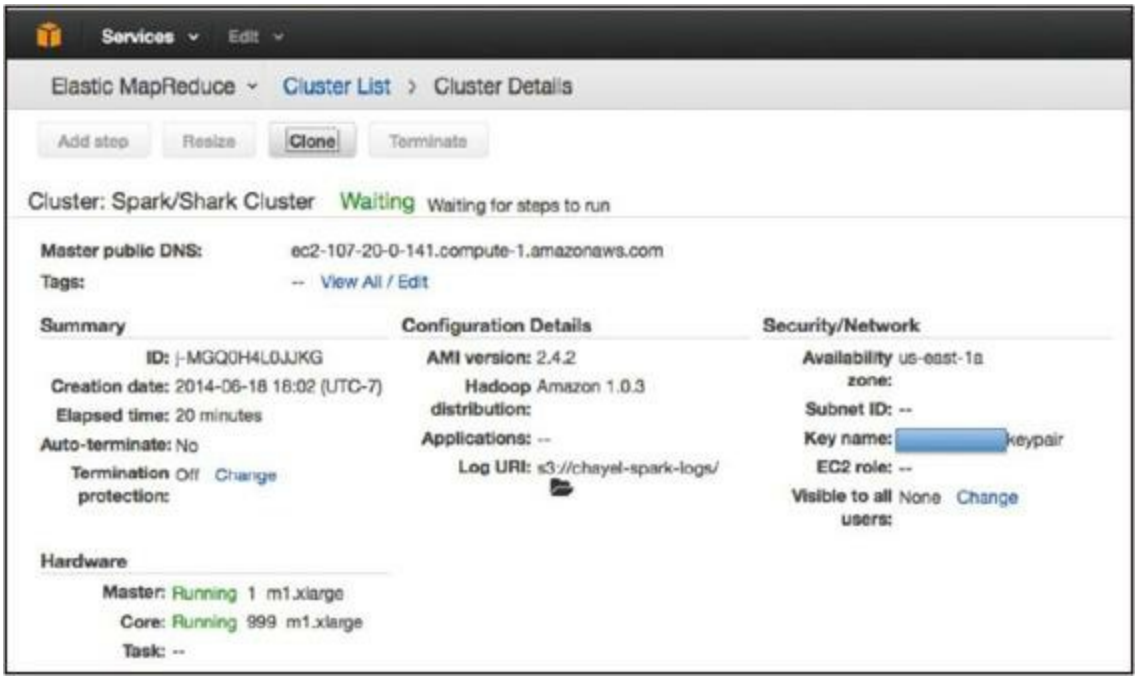

图1-9　集群细节监控界面

进入管理界面http://localhost:9091可以查看集群资源使用状况；进入http://localhost:8080可以观察Spark集群的状况。Lynx界面如图1-10所示。

3）加载数据集。

示例数据集使用Wiki文章数据，总量为4.5TB，有1万亿左右记录。Wiki文章数据存储在S3中，下载地址为s3：//bigdata-spark-demo/wikistats/。

下面创建wikistats表，将数据加载进表：

```
create external table wikistats
(
projectcode string,
pagename string,
pageviews int,
pagesize int
)
ROW FORMAT
DELIMITED FIELDS
TERMINTED BY"
LOCATION 's3n：//bigdata-spark-demo/wikistats/';
ALTER TABLE wikistats add partition(dt='2007-12')location 's3n：//bigdata-spark-demo//wikistats/2007/2007-12';
......
```

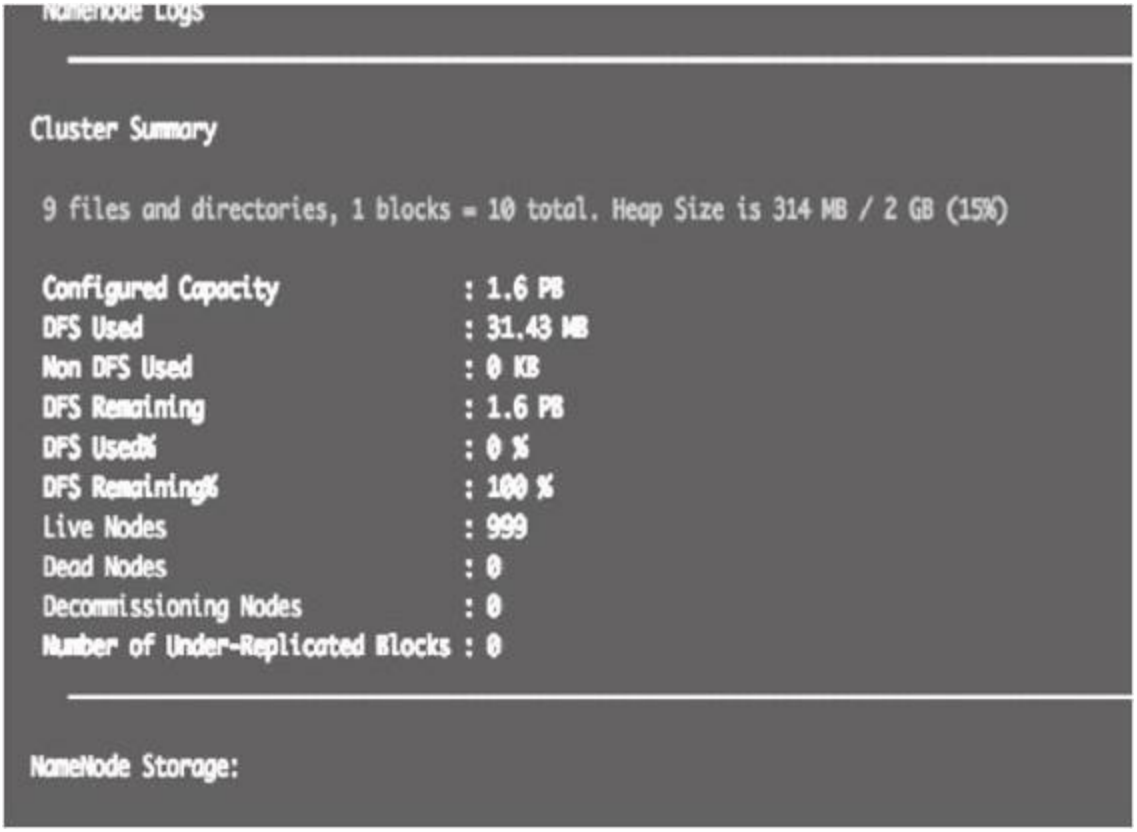

图1-10　Lynx界面

4）分析数据。

使用Shark获取2014年2月的Top 10页面。用户可以在Shark输入下面的SQL语句进行分析。

```
Select pagename，sum（pageviews） c from  wikistats_cached where dt='2014-01'
group by pagename order by c desc limit 10；
```

这个语句大致花费26s，扫描了250GB的数据。

云计算带来资源的按需分配，用户可以采用云端的虚机作为大数据分析平台的底层基础设施，在上端构建Spark集群，进行大数据分析。随着处理数据量的增加，按需扩展分析节点，增加集群的数据分析能力。

1.5.2 Spark在Yahoo！的应用

在Spark技术的研究与应用方面，Yahoo！始终处于领先地位，它将Spark应用于公司的各种产品之中。移动App、网站、广告服务、图片服务等服务的后端实时处理框架均采用了Spark+Shark的架构。

在2013年，Yahoo！拥有72656600个页面，有上百万的商品类别，上千个商品和用户特征，超过800万用户，每天需要处理海量数据。

通过图1-11可以看到Yahoo！使用Spark进行数据分析的整体架构。

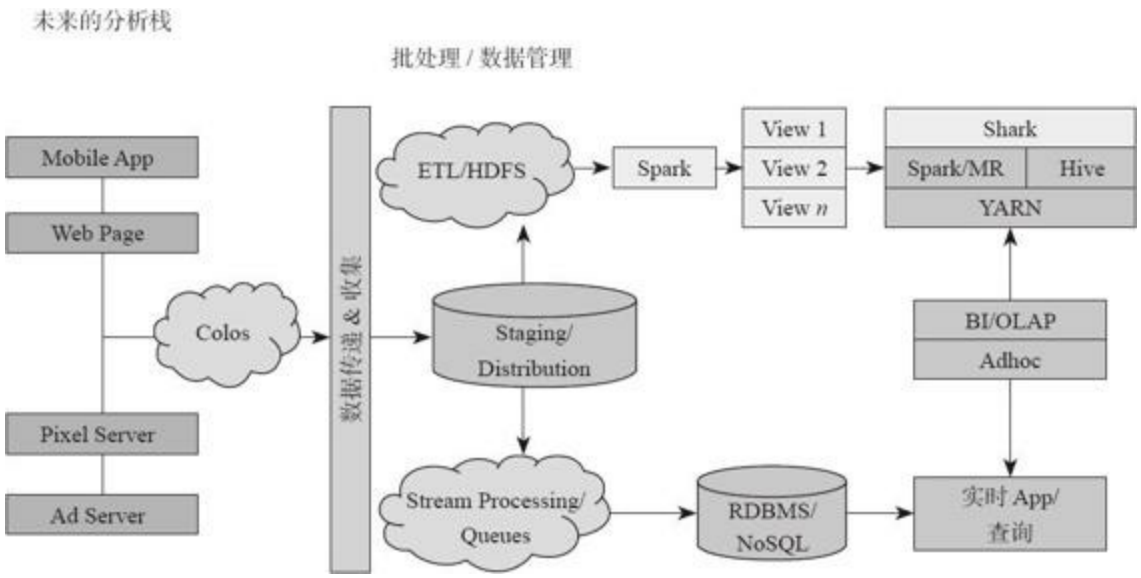

图1-11　Yahoo！大数据分析栈

大数据分析平台架构解析如下。

整个数据分析栈构建在YARN之上，这是为了让Hadoop和Spark的任务共存。主要包含两个主要模块：

1）离线处理模块：使用MapReduce和Spark+Shark混合架构。由于MapReduce适合进行ETL处理，还保留Hadoop进行数据清洗和转换。数据在ETL之后加载进HDFS/HCat/Hive数据仓库存储，之后可以通过Spark、Shark进行OLAP数据分析。

2）实时处理模块：使用Spark Streaming+Spark+Shark架构进行处理。实时流数据源源不断经过Spark Steaming初步处理和分析之后，将数据追加进关系数据库或者NoSQL数据库。之后，结合历史数据，使用Spark进行实时数据分析。

之所以选择Spark，Yahoo！基于以下几点进行考虑。

1）进行交互式SQL分析的应用需求。

2）RAM和SSD价格不断下降，数据分析实时性的需求越来越多，大数据急需一个内存计算框架进行处理。

3）程序员熟悉Scala开发，接受Spark学习曲线不陡峭。

4）Spark的社区活跃度高，开源系统的Bug能够更快地解决。

5）传统Hadoop生态系统的分析组件在进行复杂数据分析和保证实时性方面表现得力不从心。Spark的全栈支持多范式数据分析能够应对多种多样的数据分析需求。

6）可以无缝将Spark集成进现有的Hadoop处理架构。

Yahoo！的Spark集群在2013年已经达到9.2TB持久存储、192GB RAM、112节点（每节点为SATA 1×500GB（7200转的硬盘））、400GB SSD（1×400GB SATA 300MB/s）的集群规模。

1.5.3 Spark在西班牙电信的应用

西班牙电信（Telefónica，S.A.）是西班牙的一家电信公司。这是全球第五大固网和移动通信运营商。

Telefónica成立于1924年。在1997年电信市场自由化之前，Telefónica是西班牙唯一的电信运营商，至今仍占据主要的市场份额（2004年超过75%）。

西班牙电信的数据与日俱增，随着数据的增长，网络安全成为一个不可忽视的问题而凸显。DDoS攻击、SQL注入攻击、网站置换、账号盗用等网络犯罪频繁发生。如何通过大数据分析，预防网络犯罪与正确检测诊断成为迫在眉睫的问题。

传统的应对方案是，采用中心化的数据存储，收集事件、日志和警告信息，对数据分析预警，并对用户行为进行审计。但是随着犯罪多样化与数据分析技术越来越复杂，架构已经演变为中心架构服务化，并提供早期预警、离线报告、趋势预测、决策支持和可视化的大数据网络安全分析预警策略。

西班牙电信采用Stratio公司提供的含有Spark的数据分析解决方案构建自身的网络安全数据分析栈，将使用的大数据系统缩减了一半，平台复杂性降低，同时处理性能成倍提升。

整体架构如图1-12所示。

在架构图中，最顶层通过Kafka不断收集事件、日志、预警等多数据源的信息，形成流数据，完成数据集成的功能。接下来Kafka将处理好的数据传输给Storm，Storm将数据混合与预处理。最后将数据存储进Cassandra、Mongo和HDFS进行持久化存储，使用Spark进行数据分析与预警。

在数据收集阶段：数据源是多样化的，可能来自DNS日志、用户访问IP、社交媒体数据、政府公共数据源等。Kafka到数据源拉取不同数据维度数据。

在数据预处理阶段：通过Storm进行数据预处理与规范化。在这个阶段为了能够实时预警，采用比Spark Streaming实时性更高的Storm进行处理。

在数据批处理阶段：数据经过预处理阶段之后将存储到Cassandra中持久化。开发人员通过Cassandra进行一些简单的查询和数据报表分析。对于复杂的数据分析，需要使用Spark来完成。Spark+Cassandra的架构结合了两个系统的优势。Cassandra的二级索引能够加速查询处理。

Spark对机器学习和图计算等复杂数据分析应对自如，二者组合能够应对常见和复杂的数据分析负载。

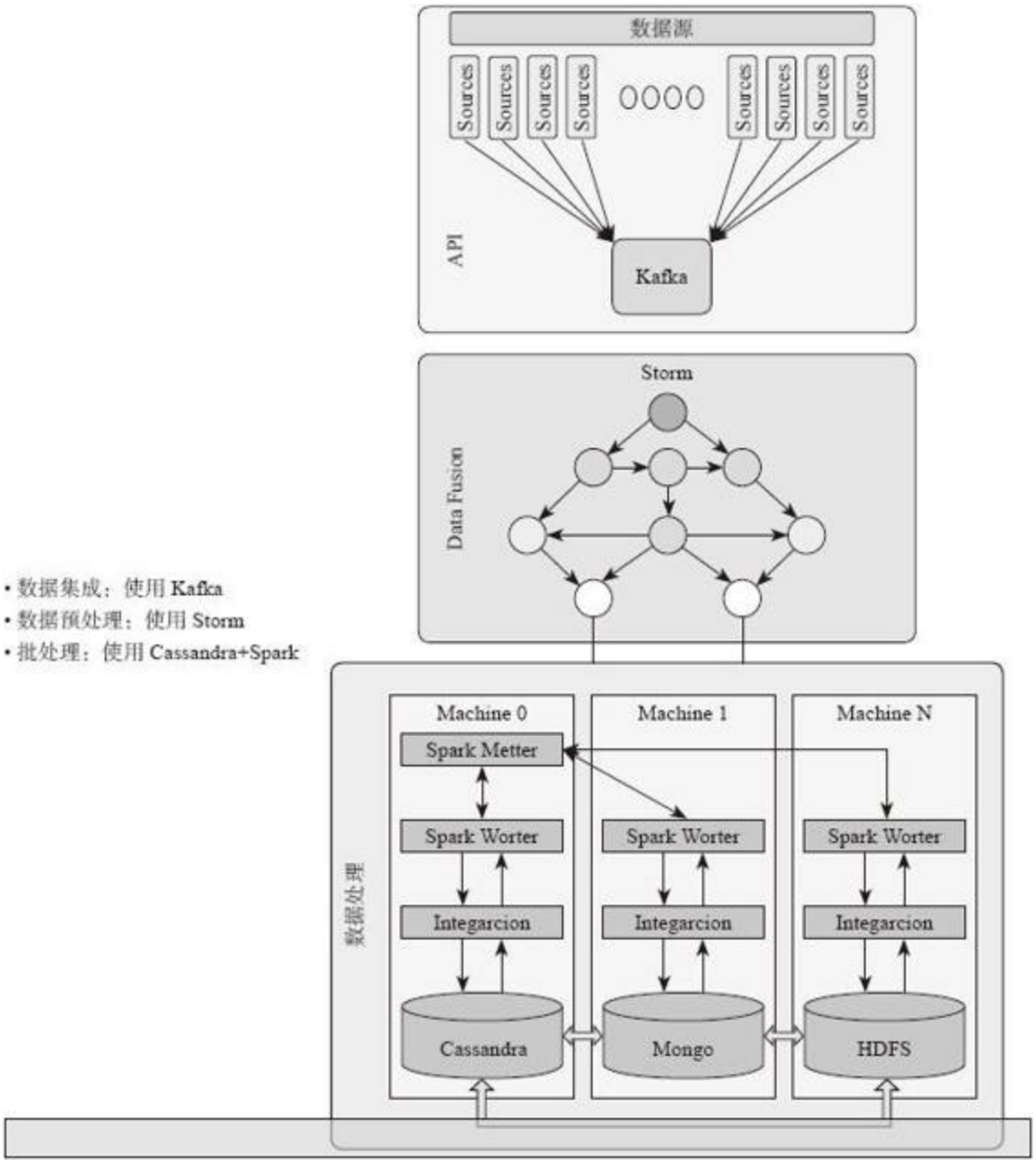

图1-12　西班牙电信数据分析平台

1.5.4 Spark在淘宝的应用

数据挖掘算法有时候需要迭代，每次迭代时间非常长，这是淘宝选择一个更高性能计算框架Spark的原因。Spark编程范式更加简洁也是一大原因。另外，GraphX提供图计算的能力也是很重要的。

1.Spark on YARN架构

Spark的计算调度方式从Mesos到Standalone，即自建Spark计算集群。虽然Standalone方式性能与稳定性都得到了提升，但自建集群资源少，需要从云梯集群复制数据，不能满足数据挖掘与计算团队业务需求[1]。而Spark on YARN能让Spark计算模型在云梯YARN集群上运行，直接读取云梯上的数据，并充分享受云梯YARN集群丰富的计算资源。图1-13为Spark on YARN的架构。

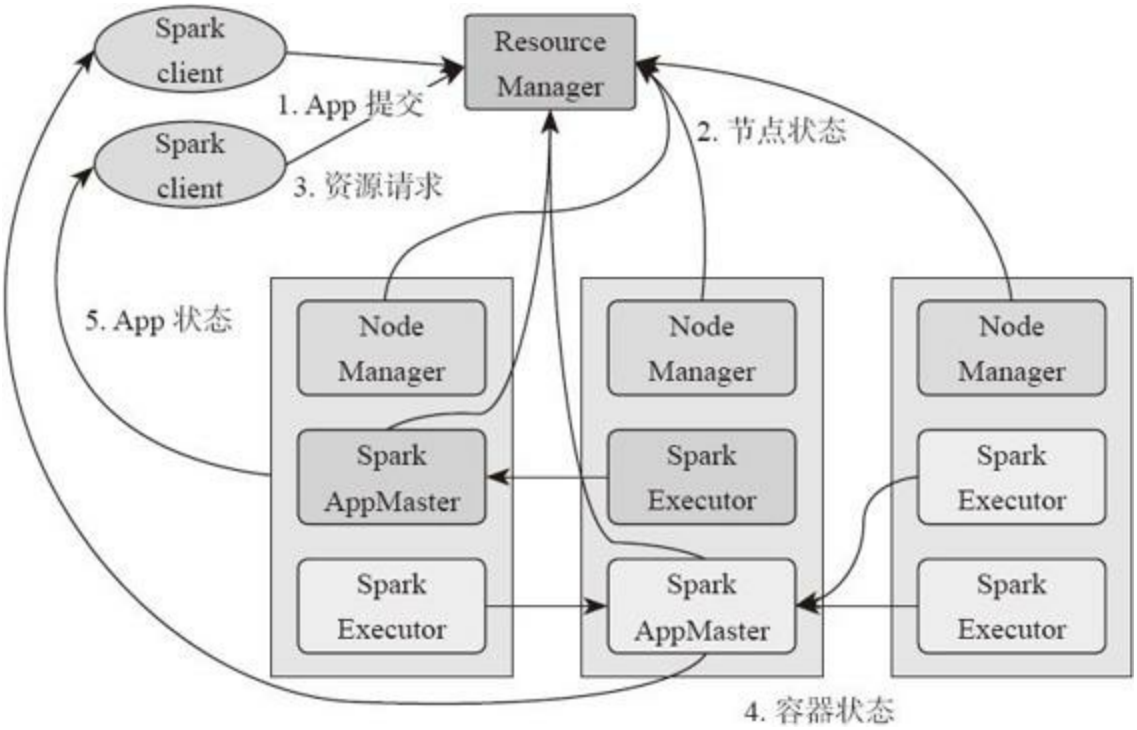

图1-13 Spark on YARN架构

Spark on YARN架构解析如下。

基于YARN的Spark作业首先由客户端生成作业信息，提交给

ResourceManager，ResourceManager在某一NodeManager汇报时把AppMaster分配给NodeManager，NodeManager启动SparkAppMaster，SparkAppMaster启动后初始化作业，然后向ResourceManager申请资源，申请到相应资源后，SparkAppMaster通过RPC让NodeManager启动相应的SparkExecutor，SparkExecutor向SparkAppMaster汇报并完成相应的任务。此外，SparkClient会通过AppMaster获取作业运行状态。目前，淘宝数据挖掘与计算团队通过Spark on YARN已实现MLR、PageRank和JMeans算法，其中MLR已作为生产作业运行。

2.协作系统

1）Spark Streaming：淘宝在云梯构建基于Spark Streaming的实时流处理框架。Spark Streaming适合处理历史数据和实时数据混合的应用需求，能够显著提高流数据处理的吞吐量。其对交易数据、用户浏览数据等流数据进行处理和分析，能够更加精准、快速地发现问题和进行预测。

2）GraphX[2]：淘宝将交易记录中的物品和人组成大规模图。使用GraphX对这个大图进行处理（上亿个节点，几十亿条边）。GraphX能够和现有的Spark平台无缝集成，减少多平台的开发代价。

本节主要介绍了Spark在工业界的应用。Spark起源于学术界，发展于工业界，现在已经成为大数据分析不可或缺的计算框架。通过Amazon提供Spark云服务，可以看到Big Data on Cloud已经兴起。Yahoo！很早就开始使用Spark，将Spark用于自己的广告平台、商品交易数据分析和推荐系统等数据分析领域。同时Yahoo！也积极回馈社区，与社区形成良好的互动。Stratio公司为西班牙电信提供基于Spark+Cassandra+Storm架构的数据分析解决方案，实现流数据实时处理与离线数据分析兼顾，通过它们的案例可以看到多系统混合提供多数据计算范式分析平台是未来的一个趋势。最后介绍国内淘宝公司的Spark应用案例，淘宝是国内较早使用Spark的公司，通过Spark进行大规模机器学、图计算以及流数据分析，并积极参与社区，与社区形成良好互动，并乐于分享技术经验。希望读者通过企业案例能够全面

了解Spark的广泛应用和适用场景。

[1] 参见沈洪的《深入剖析阿里巴巴云梯YARN集群》，《程序员》，2013.12。

[2] 参见文章：黄明，吴炜.快刀初试：Spark GraphX在淘宝的实践.程序员，2014.8。

1.6 本章小结

本章首先介绍了Spark分布式计算平台和BDAS。BDAS的核心框架Spark为用户提供了系统底层细节透明、编程接口简洁的分布式计算平台。Spark具有计算速度快、实时性高、容错性好等突出特点。基于Spark的应用已经逐步落地，尤其是在互联网领域，如淘宝、腾讯、网易等公司的发展已经成熟。同时电信、银行等传统行也开始逐步试水Spark并取得了较好效果。本章也对Spark的基本情况、架构、运行逻辑等进行了介绍。最后介绍了Spark在工业界的应用，读者可以看到Spark的蓬勃发展以及在大数据分析平台中所处的位置及重要性。

读者通过本章可以初步认识和理解Spark，更为底层的细节将在后续章节详细阐述。

相信读者已经想搭建自己的Spark集群环境一探究竟了，接下来将介绍Spark的安装与配置。

第2章　Spark集群的安装与部署

Spark的安装简便，用户可以在官网上下载到最新的软件包，网址为http://spark.apache.org/。

Spark最早是为了在Linux平台上使用而开发的，在生产环境中也是部署在Linux平台上，但是Spark在UNIX、Windwos和Mac OS X系统上也运行良好。不过，在Windows上运行Spark稍显复杂，必须先安装Cygwin以模拟Linux环境，才能安装Spark。

由于Spark主要使用HDFS充当持久化层，所以完整地使用Spark需要预先安装Hadoop。下面介绍Spark集群的安装和部署。

2.1 Spark的安装与部署

Spark在生产环境中，主要部署在安装有Linux系统的集群中。在Linux系统中安装Spark需要预先安装JDK、Scala等所需的依赖。由于Spark是计算框架，所以需要预先在集群内有搭建好存储数据的持久化层，如HDFS、Hive、Cassandra等。最后用户就可以通过启动脚本运行应用了。

2.1.1 在Linux集群上安装与配置Spark

下面介绍如何在Linux集群上安装与配置Spark。

1.安装JDK

安装JDK大致分为下面4个步骤。

1）用户可以在Oracle JDK的官网下载相应版本的JDK，本例以JDK 1.6为例，官网地址为http://www.oracle.com/technetwork/java/javase/downloads/index.html。

2）下载后，在解压出的JDK的目录下执行bin文件。

```
./jdk-6u38-ea-bin-b04-linux-amd64-31_oct_2012.bin
```

3）配置环境变量，在/etc/profile增加以下代码。

```
JAVA_HOME=/home/chengxu/jdk1.6.0_38
PATH=$JAVA_HOME/bin：$PATH
CLASSPATH=.：$JAVA_HOME/jre/lib/rt.jar：$JAVA_HOME/jre/lib/dt.jar：$JAVA_HOME/jre/lib/tools.jar
export JAVA_HOME PATH CLASSPATH
```

4）使profile文件更新生效。

```
./etc/profile
```

2.安装Scala

Scala官网提供各个版本的Scala，用户需要根据Spark官方规定的Scala版本进行下载和安装。Scala官网地址为http://www.scala-lang.org/。

以Scala-2.10为例进行介绍。

1）下载scala-2.10.4.tgz。

2）在目录下解压：

```
tar -xzvf scala-2.10.4.tgz
```

3）配置环境变量，在/etc/profile中添加下面的内容。

```
export SCALA_HOME=/home/chengxu/scala-2.10.4/scala-2.10.4
export PATH=${SCALA_HOME}/bin：$PATH
```

4）使profile文件更新生效。

```
./etc/profile
```

3.配置SSH免密码登录

在集群管理和配置中有很多工具可以使用。例如，可以采用pssh等Linux工具在集群中分发与复制文件，用户也可以自己书写Shell、Python的脚本分发包。

Spark的Master节点向Worker节点发命令需要通过ssh进行发送，用户不希望Master每发送一次命令就输入一次密码，因此需要实现Master无密码登录到所有Worker。

Master作为客户端，要实现无密码公钥认证，连接到服务端Worker。需要在Master上生成一个密钥对，包括一个公钥和一个私钥，然后将公钥复制到Worker上。当Master通过ssh连接Woker时，Worker就会生成一个随机数并用Master的公钥对随机数进行加密，发送给Worker。Master收到加密数之后再用私钥进行解密，并将解密数回传给Worker，Worker确认解密数无误之后，允许Master进行连接。这就是一个公钥认证过程，其间不需要用户手工输入密码，主要过程是将Master节点公钥复制到Worker节点上。

下面介绍如何配置Master与Worker之间的SSH免密码登录。

1）在Master节点上，执行以下命令。

```
ssh-keygen-trsa
```

2）打印日志执行以下命令。

```
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa)：
/*回车，设置默认路径*/
Enter passphrase (empty for no passphrase)：
/*回车，设置空密码*/
Enter same passphrase again：
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
```

如果是root用户，则在/root/.ssh/目录下生成一个私钥id_rsa和一个公钥id_rsa.pub。

把Master上的id_rsa.pub文件追加到Worker的authorized_keys内，以172.20.14.144（Worker）节点为例。

3）复制Master的id_rsa.pub文件。

```
scp id_rsa.pub root@172.20.14.144：/home
/*可使用pssh对全部节点分发*/
```

4）登录172.20.14.144（Worker节点），执行以下命令。

```
cat /home/id_rsa.pub >> /root/.ssh/authorized_keys
/*可使用pssh对全部节点分发*/
```

其他的Worker执行同样的操作。

注意：配置完毕，如果Master仍然不能访问Worker，可以修改Worker的authorized_keys文件的权限，命令为chmod 600 authorized_keys。

4.安装Hadoop

下面讲解Hadoop的安装过程和步骤。

（1）下载hadoop-2.2.0

1）选取一个Hadoop镜像网址，下载Hadoop（官网地址为http://hadoop.apache.org/）。

```
$ wgethttp：//www.trieuvan.com/apache/hadoop/common/
hadoop-2.2.0/hadoop-2.2.0.tar.gz
```

2）解压tar包。

```
$ sudo tar-vxzf hadoop-2.2.0.tar.gz -C /usr/local
$ cd /usr/local
$ sudo mv hadoop-2.2.0 hadoop
$ sudo chown -R hduser：hadoop hadoop
```

（2）配置Hadoop环境变量

1）编辑profile文件。

```
vi /etc/profile
```

2）在profile文件中增加以下内容。

```
export JAVA_HOME=/usr/lib/jvm/jdk/
export HADOOP_INSTALL=/usr/local/hadoop
export PATH=$PATH：$HADOOP_INSTALL/bin
export PATH=$PATH：$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export YARN_HOME=$HADOOP_INSTALL
```

通过如上配置就可以让系统找到JDK和Hadoop的安装路径。

（3）编辑配置文件

1）进入Hadoop所在目录/usr/local/hadoop/etc/hadoop。

2）配置hadoop-env.sh文件。

```
export JAVA_HOME=/usr/lib/jvm/jdk/
```

3）配置core-site.xml文件。

```
<configuration>
/*这里的值指的是默认的HDFS路径*/
<property>
<name>fs.defaultFS</name>
<value>hdfs：//Master：9000</value>
</property>
/*缓冲区大小：io.file.buffer.size默认是4KB*/
<property>
<name>io.file.buffer.size</name>
<value>131072</value>
</property>
/*临时文件夹路径*/
<property>
<name>hadoop.tmp.dir</name>
<value>file：/home//tmp</value>
<description>Abase for other
temporary directories.       </description>
</property>
<property>
<name>hadoop.proxyuser.hduser.hosts</name>
<value>*</value>
</property>
<property>
<name>hadoop.proxyuser.hduser.groups</name>
<value>*</value>
</property>
</configuration>
```

4）配置yarn-site.xml文件。

```
<configuration>
<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
<property>
<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
/*resourcemanager的地址*/
<property>
<name>yarn.resourcemanager.address</name>
<value>Master：8032</value>
</property>
/*调度器的端口*/
<property>
<name>yarn.resourcemanager.scheduler.address</name>
<value> Master1：8030</value>
</property>
/*resource-tracker端口*/
<property>
<name>yarn.resourcemanager.resource-tracker.address</name>
<value> Master：8031</value>
</property>
/*resourcemanager管理器端口*/
<property>
<name>yarn.resourcemanager.admin.address</name>
<value> Master：8033</value>
</property>
/* ResourceManager 的 Web 端口，监控 job 的资源调度*/
<property>
<name>yarn.resourcemanager.webapp.address</name>
<value> Master：8088</value>
</property>
```

```
</configuration>
```

5）配置mapred-site.xml文件，加入如下内容。

```
<configuration>
/*hadoop对map-reduce运行框架一共提供了3种实现，在mapred-site.xml中通过"mapreduce.framework.name"这个属性来设置为"classic"、"yarn"或者"local"*/
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
/*MapReduce JobHistory Server地址*/
<property>
<name>mapreduce.jobhistory.address</name>
<value>Master：10020</value>
</property>
/*MapReduce JobHistory Server Web UI地址*/
<property>
<name>mapreduce.jobhistory.webapp.address</name>
<value>Master：19888</value>
</property>
</configuration>
```

（4）创建namenode和datanode目录，并配置其相应路径

1）创建namenode和datanode目录，执行以下命令。

```
$ mkdir /hdfs/namenode
$ mkdir /hdfs/datanode
```

2）执行命令后，再次回到目录/usr/local/hadoop/etc/hadoop，配置hdfs-site.xml文件，在文件中添加如下内容。

```
<configuration>
/*配置主节点名和端口号*/
<property>
<name>dfs.namenode.secondary.http-address</name>
<value>Master：9001</value>
</property>
/*配置从节点名和端口号*/
<property>
<name>dfs.namenode.name.dir</name>
<value>file：/hdfs/namenode</value>
</property>
/*配置datanode的数据存储目录*/
<property>
<name>dfs.datanode.data.dir</name>
<value>file：/hdfs/datanode</value>
</property>
/*配置副本数*/
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
/*将dfs.webhdfs.enabled属性设置为true，否则就不能使用webhdfs的LISTSTATUS、LISTFILESTATUS等需要列出文件、文件夹状态的命
```

```
令，因为这些信息都是由namenode保存的*/
<property>
<name>dfs.webhdfs.enabled</name>
<value>true</value>
</property>
</configuration>
```

（5）配置Master和Slave文件

1）Master文件负责配置主节点的主机名。例如，主节点名为Master，则需要在Master文件添加以下内容。

```
Master /*Master为主节点主机名*/
```

2）配置Slaves文件添加从节点主机名，这样主节点就可以通过配置文件找到从节点，和从节点进行通信。例如，以Slave1~Slave5为从节点的主机名，就需要在Slaves文件中添加如下信息。

```
/Slave*为从节点主机名*/
Slave1
Slave2
Slave3
Slave4
Slave5
```

（6）将Hadoop的所有文件通过pssh分发到各个节点

执行如下命令。

```
./pssh -h hosts.txt -r /hadoop  /
```

（7）格式化Namenode（在Hadoop根目录下）

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

（8）启动Hadoop

```
./sbin/start-all.sh
```

（9）查看是否配置和启动成功

如果在x86机器上运行，则通过jps命令，查看相应的JVM进程

```
2584 DataNode
2971 ResourceManager
3462 Jps
3179 NodeManager
2369 NameNode
2841 SecondaryNameNode
```

注意，由于在IBM JVM中没有jps命令，所以需要用户按照下面命令逐个查看。

```
ps-aux|grep *DataNode*      /*查看DataNode进程*/
```

5.安装Spark

进入官网下载对应Hadoop版本的Spark程序包（见图2-1），官网地址为http://spark.apache.org/downloads.html。

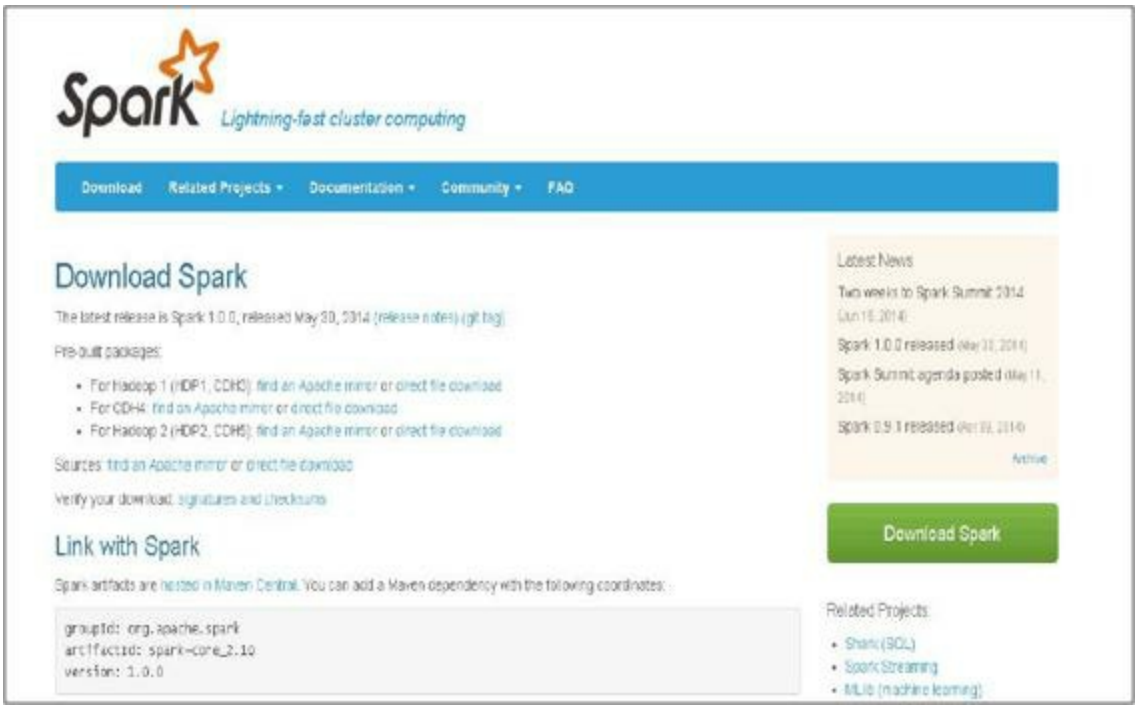

图2-1　Spark下载官网

截止到笔者进行本书写作之时，Spark已经更新到1.0版本。

以Spark1.0版本为例，介绍Spark的安装。

1）下载spark-1.0.0-bin-hadoop2.tgz。

2）解压tar-xzvf spark-1.0.0-bin-hadoop2.tgz。

3）配置conf/spark-env.sh文件

①用户可以配置基本的参数，其他更复杂的参数请见官网的配置（Configuration）页面，Spark配置（Configuration）地址为：http://spark.apache.org/docs/latest/configuration.html。

②编辑conf/spark-env.sh文件，加入下面的配置参数。

```
export SCALA_HOME=/path/to/scala-2.10.4
export SPARK_WORKER_MEMORY=7g
export SPARK_MASTER_IP=172.16.0.140
export MASTER=spark：//172.16.0.140：7077
```

参数SPARK_WORKER_MEMORY决定在每一个Worker节点上可用的最大内存，增加这个数值可以在内存中缓存更多数据，但是一定要给Slave的操作系统和其他服务预留足够的内存。

需要配置SPARK_MASTER_IP和MASTER，否则会造成Slave无法注册主机错误。

4）配置slaves文件。

编辑conf/slaves文件，以5个Worker节点为例，将节点的主机名加入slaves文件中。

```
Slave1
Slave2
Slave3
Slave4
Slave5
```

6.启动集群

（1）Spark启动与关闭

1）在Spark根目录启动Spark。

```
./sbin/start-all.sh
```

2）关闭Spark。

```
./sbin/stop-all.sh
```

（2）Hadoop的启动与关闭

1）在Hadoop根目录启动Hadoop。

```
./sbin/start-all.sh
```

2）关闭Hadoop。

```
./sbin/stop-all.sh
```

（3）检测是否安装成功

1）正常状态下的Master节点如下。

```
-bash-4.1# jps
23526 Jps
2127 Master
7396 NameNode
7594 SecondaryNameNode
7681 ResourceManager
```

2）利用ssh登录Worker节点。

```
-bash-4.1# ssh slave2
-bash-4.1# jps
1405 Worker
1053 DataNode
22455 Jps
31935 NodeManager
```

至此，在Linux集群上安装与配置Spark集群的步骤告一段落。

2.1.2　在Windows上安装与配置Spark

本节介绍在Windows系统上安装Spark的过程。在Windows环境下需要安装Cygwin模拟Linux的命令行环境来安装Spark。

（1）安装JDK

相对于Linux、Windows的JDK安装更加自动化，用户可以下载安装Oracle JDK或者OpenJDK。只安装JRE是不够的，用户应该下载整个JDK。

安装过程十分简单，运行二进制可执行文件即可，程序会自动配置环境变量。

（2）安装Cygwin

Cygwin[1]是在Windows平台下模拟Linux环境的一个非常有用的工具，只有通过它才可以在Windows环境下安装Hadoop和Spark。具体安装步骤如下。

1）运行安装程序，选择install from internet。

2）选择网络最好的下载源进行下载。

3）进入Select Packages界面（见图2-2），然后进入Net，选择openssl及openssh。因为之后还是会用到ssh无密钥登录的。

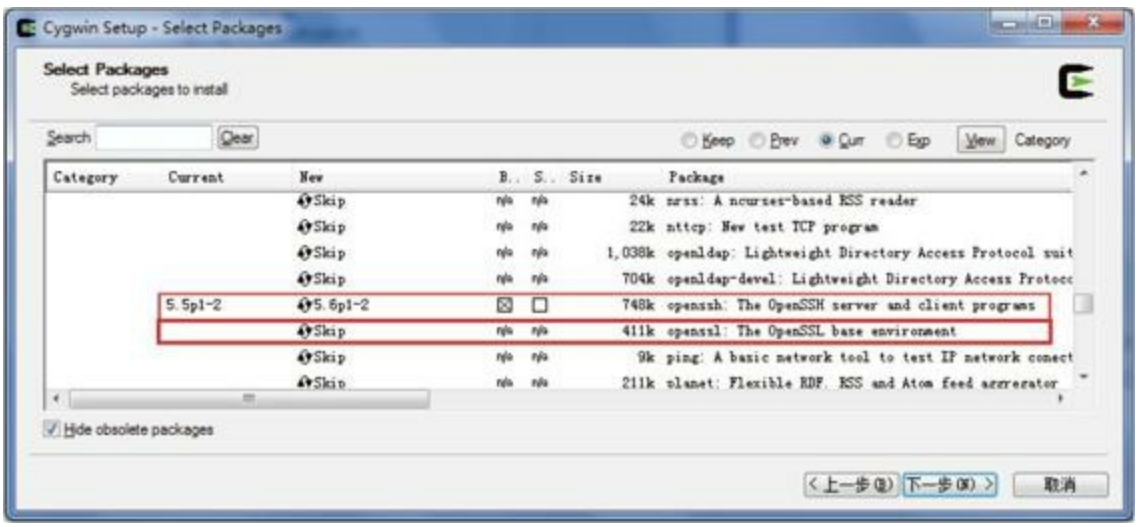

图2-2　Cygwin安装选择界面

另外应该安装“Editors Category”下面的“vim”。这样就可以在Cygwin上方便地修改配置文件。

最后需要配置环境变量，依次选择“我的电脑”→“属性”→“高级系统设置”→“环境变量”命令，更新环境变量中的path设置，在其后添加Cygwin的bin目录和Cygwin的usr\bin两个目录。

（3）安装sshd并配置免密码登录

1）双击桌面上的Cygwin图标，启动Cygwin，执行ssh-host-config-y命令，出现如图2-3所示的界面。

图2-3　Cygwin安装sshd选择界面

2）执行后，提示输入密码，否则会退出该配置，此时输入密码和确认密码，按回车键。最后出现Host configuration finished.Have fun！表示安装成功。

3）输入net start sshd，启动服务。或者在系统的服务中找到并启动Cygwin sshd服务。

注意，如果是Windows 8操作系统，启动Cygwin时，需要以管理员身份运行（右击图

标，选择以管理员身份运行），否则会因为权限问题，提示“发生系统错误5”。

（4）配置SSH免密码登录

1）执行ssh-keygen命令生成密钥文件，如图2-4所示。

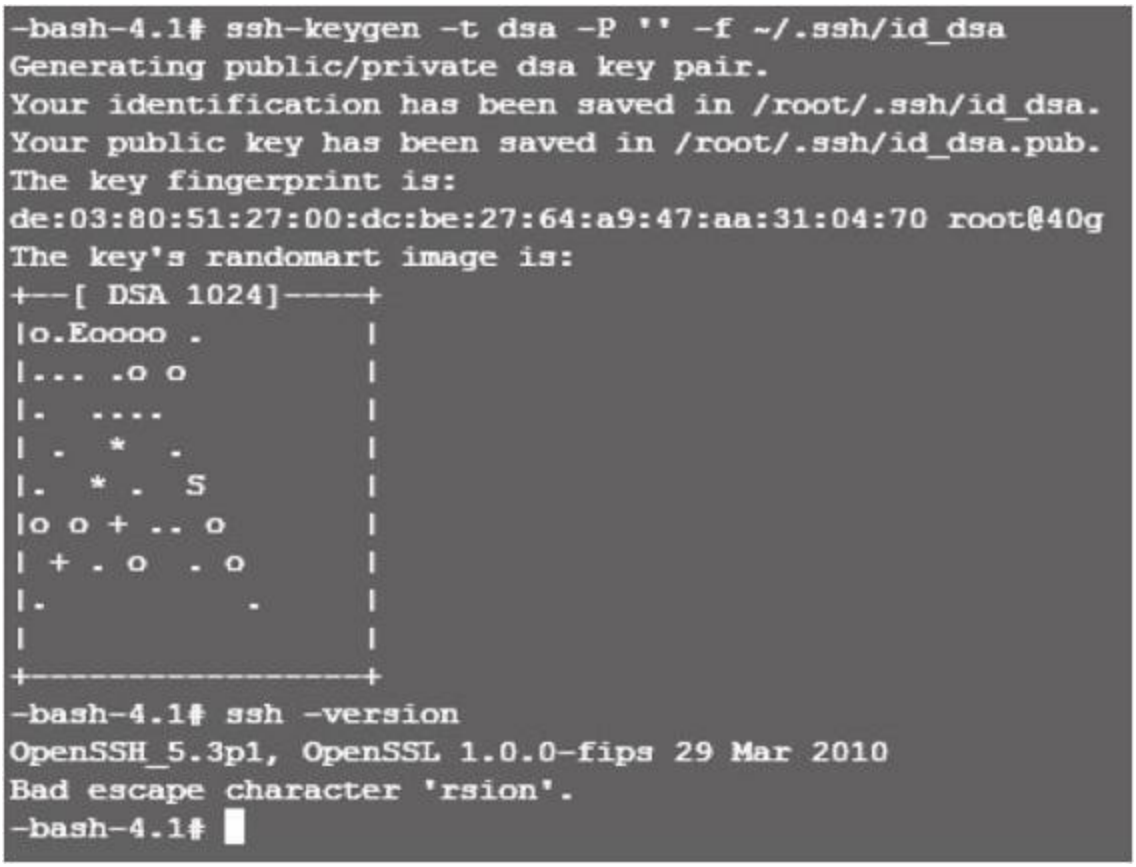

图2-4　Cygwin ssh生成密钥

2）执行此命令后，在你的Cygwin\home\用户名路径下面会生成.ssh文件夹，可以通过命令ls-a/home/用户名查看，通过ssh-version命令查看版本。

3）执行完ssh-keygen命令后，再执行下面命令，生成authorized_keys文件。

```
cd ~/.ssh/
cp id_dsa.pub authorized_keys
```

这样就配置好了sshd服务。

（5）配置Hadoop

修改和配置相关文件与Linux的配置一致，读者可以参照上文Linux中的配置方式，这里不再赘述。

（6）配置Spark

修改和配置相关文件与Linux的配置一致，读者可以参照上文Linux中的配置方式，这里不再赘述。

（7）运行Spark

1）Spark的启动与关闭

①在Spark根目录启动Spark。

```
./sbin/start-all.sh
```

②关闭Spark。

```
./sbin/stop-all.sh
```

2）Hadoop的启动与关闭

①在Hadoop根目录启动Hadoop。

```
./sbin/start-all.sh
```

②关闭Hadoop。

```
./sbin/stop-all.sh
```

3）检测是否安装成功

正常状态下会出现如下内容。

```
-bash-4.1# jps
23526 Jps
2127 Master
7396 NameNode
7594 SecondaryNameNode
7681 ResourceManager
1053 DataNode
31935 NodeManager
1405 Worker
```

如缺少进程请到logs文件夹下查看相应日志，针对具体问题进行解决。

[1] 可以通过官网进行下载：http://www.cygwin.com/。

2.2 Spark集群初试

假设已经按照上述步骤配置完成Spark集群，可以通过两种方式运行Spark中的样例。下面以Spark项目中的SparkPi为例，可以用以下方式执行样例。

1）以./run-example的方式执行

用户可以按照下面的命令执行Spark样例。

```
./bin/run-example  org.apache.spark.examples.SparkPi
```

2）以./Spark Shell的方式执行

Spark自带交互式的Shell程序，方便用户进行交互式编程。下面进入Spark Shell的交互式界面。

```
./bin/spark-shell
```

用户可以将下面的例子复制进Spark Shell中执行。

```
importscala.math.random
importorg.apache.spark._
objectSparkPi {
def main(args: Array[String]) {
  val slices = 2
  val n = 100000 * slices
  val count = sc.parallelize(1 to n, slices).map { i =>
  val x = random * 2 - 1
  val y = random * 2 - 1
  if (x*x + y*y < 1) 1 else 0
      }.reduce(_ + _)
  println("Pi is roughly " + 4.0 * count / n)
  }
}
```

按回车键执行上述命令。

注意，Spark Shell中已经默认将SparkContext类初始化为对象sc。用户代码如果需要用到，则直接应用sc即可，否则用户自己再初始化，就会出现端口占用问题，相当于启动两个

上下文。

3）通过Web UI查看集群状态

浏览器输入http://masterIP:8080，也可以观察到集群的整个状态是否正常，如图2-5所示。集群会显示与图2-5类似的画面。masterIP配置为用户的Spark集群的主节点IP。

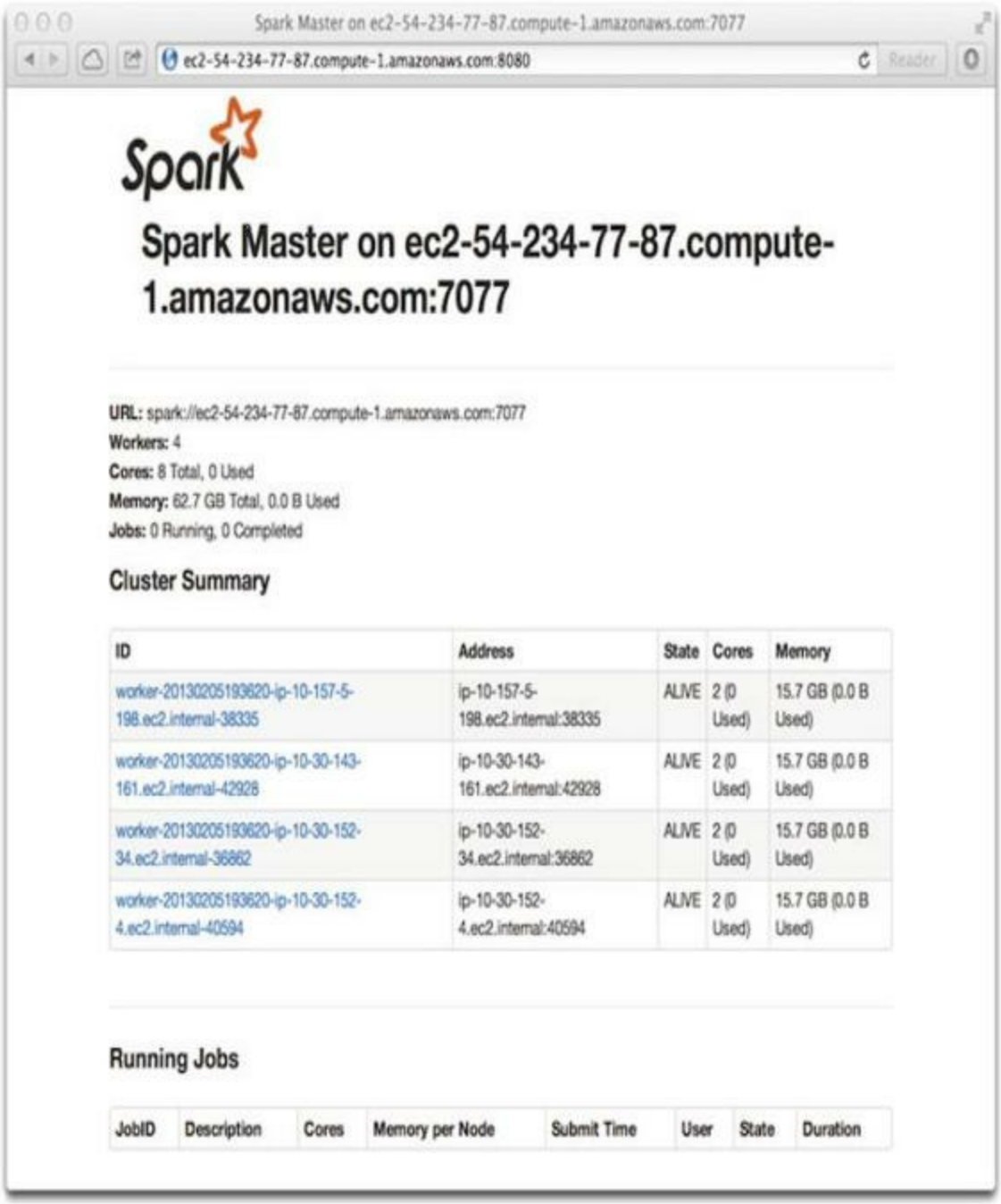

图2-5　Spark Web UI

2.3 本章小结

本章主要介绍了如何在Linux和Windows环境下安装部署Spark集群。

由于Spark主要使用HDFS充当持久化层，所以完整地使用Spark需要预先安装Hadoop。通过本章介绍，读者就可以开启Spark的实战之旅了。

下一章将介绍Spark的计算模型，Spark将分布式的内存数据抽象为弹性分布式数据集（RDD），并在其上实现了丰富的算子，从而对RDD进行计算，最后将算子序列转化为有向无环图进行执行和调度。

第3章　Spark计算模型

创新都是站在巨人的肩膀上产生的，在大数据领域也不例外。微软的Dryad使用DAG执行模式、子任务自由组合的范型。该范型虽稍显复杂，但较为灵活。Pig也针对大关系表的处理提出了很多有创意的处理方式，如flatten、cogroup。经典虽难以突破，但作为后继者的Spark借鉴经典范式并进行创新。经过实践检验，Spark的编程范型在处理大数据时显得简单有效。<Key，Value>的数据处理与传输模式也大获全胜。

Spark站在巨人的肩膀上，依靠Scala强有力的函数式编程、Actor通信模式、闭包、容器、泛型，借助统一资源分配调度框架Mesos，融合了MapReduce和Dryad，最后产生了一个简洁、直观、灵活、高效的大数据分布式处理框架。

与Hadoop不同，Spark一开始就瞄准性能，将数据（包括部分中间数据）放在内存，在内存中计算。用户将重复利用的数据缓存到内存，提高下次的计算效率，因此Spark尤其适合迭代型和交互型任务。Spark需要大量的内存，但性能可随着机器数目呈多线性增长。本章将介绍Spark的计算模型。

3.1 Spark程序模型

下面通过一个经典的示例程序来初步了解Spark的计算模型，过程如下。

1）SparkContext中的textFile函数从HDFS[1]读取日志文件，输出变量file[2]。

```
val file=sc.textFile("hdfs://xxx")
```

2）RDD中的filter函数过滤带“ERROR”的行，输出errors（errors也是一个RDD）。

```
val errors=file.filter(line=>line.contains("ERROR"))
```

3）RDD的count函数返回“ERROR”的行数：errors.count（ ）。

RDD操作起来与Scala集合类型没有太大差别，这就是Spark追求的目标：像编写单机程序一样编写分布式程序，但它们的数据和运行模型有很大的不同，用户需要具备更强的系统把控能力和分布式系统知识。

从RDD的转换和存储角度看这个过程，如图3-1所示。

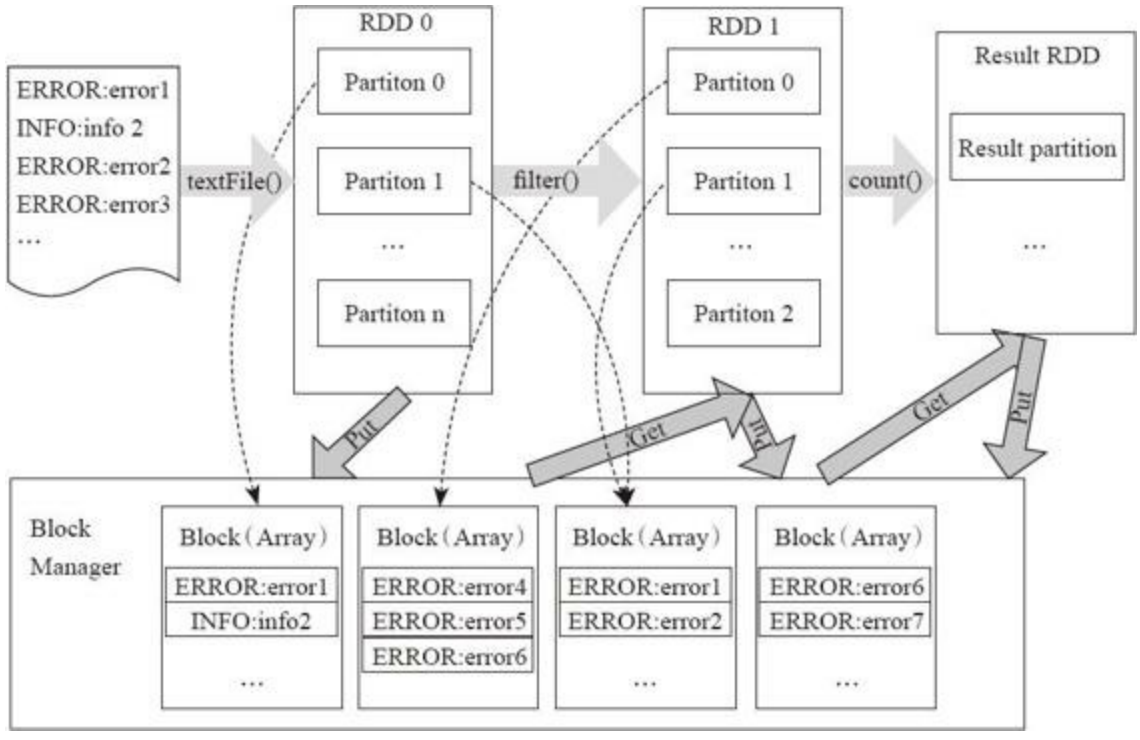

图3-1　Spark程序模型

在图3-1中，用户程序对RDD通过多个函数进行操作，将RDD进行转换。Block-Manager管理RDD的物理分区，每个Block就是节点上对应的一个数据块，可以存储在内存或者磁盘。而RDD中的partition是一个逻辑数据块，对应相应的物理块Block。本质上一个RDD在代码中相当于是数据的一个元数据结构，存储着数据分区及其逻辑结构映射关系，存储着RDD之前的依赖转换关系。

[1] 也可以是本地文件或者其他的持久化层，如Hive等。

[2] file是一个RDD，数据项是文件中的每行数据。

3.2 弹性分布式数据集

本节简单介绍RDD，并介绍RDD与分布式共享内存的异同。

3.2.1 RDD简介

在集群背后，有一个非常重要的分布式数据架构，即弹性分布式数据集（resilient distributed dataset，RDD），它是逻辑集中的实体，在集群中的多台机器上进行了数据分区。通过对多台机器上不同RDD分区的控制，就能够减少机器之间的数据重排（data shuffling）。Spark提供了"partitionBy"运算符，能够通过集群中多台机器之间对原始RDD进行数据再分配来创建一个新的RDD。RDD是Spark的核心数据结构，通过RDD的依赖关系形成Spark的调度顺序。通过对RDD的操作形成整个Spark程序。

（1）RDD的两种创建方式

1）从Hadoop文件系统（或与Hadoop兼容的其他持久化存储系统，如Hive、Cassandra、Hbase）输入（如HDFS）创建。

2）从父RDD转换得到新的RDD。

（2）RDD的两种操作算子

对于RDD可以有两种计算操作算子：Transformation（变换）与Action（行动）。

1）Transformation（变换）。

Transformation操作是延迟计算的，也就是说从一个RDD转换生成另一个RDD的转换操作不是马上执行，需要等到有Actions操作时，才真正触发运算。

2）Action（行动）

Action算子会触发Spark提交作业（Job），并将数据输出到Spark系统。

（3）RDD的重要内部属性

1）分区列表。

2）计算每个分片的函数。

3）对父RDD的依赖列表。

4）对Key-Value对数据类型RDD的分区器，控制分区策略和分区数。

5）每个数据分区的地址列表（如HDFS上的数据块的地址）。

3.2.2 RDD与分布式共享内存的异同

RDD是一种分布式的内存抽象，表3-1列出了RDD与分布式共享内存（Distributed Shared Memory，DSM）的对比。在DSM系统[1]中，应用可以向全局地址空间的任意位置进行读写操作。DSM是一种通用的内存数据抽象，但这种通用性同时也使其在商用集群上实现有效的容错性和一致性更加困难。

RDD与DSM主要区别在于[2]，不仅可以通过批量转换创建（即"写"）RDD，还可以对任意内存位置读写。RDD限制应用执行批量写操作，这样有利于实现有效的容错。特别是，由于RDD可以使用Lineage（血统）来恢复分区，基本没有检查点开销。失效时只需要重新计算丢失的那些RDD分区，就可以在不同节点上并行执行，而不需要回滚（Roll Back）整个程序。

表3-1 RDD与DSM的对比

对比项目	RDD	DSM
读	批量或细粒度读操作	细粒度读操作
写	批量转换操作	细粒度转换操作
一致性	不重要（RDD是不可更改的）	取决于应用程序或运行时
容错性	细粒度，低开销使用lineage（血统）	需要检查点操作和程序回滚
落后任务的处理	任务备份，重新调度执行	很难处理
任务安排	基于数据存放的位置自动实现	取决于应用程序

通过备份任务的复制，RDD还可以处理落后任务（即运行很慢的节点），这点与MapReduce类似，DSM则难以实现备份任务，因为任务及其副本均需读写同一个内存位置的数据。

与DSM相比，RDD模型有两个优势。第一，对于RDD中的批量操作，运行时将根据数据存放的位置来调度任务，从而提高性能。第二，对于扫描类型操作，如果内存不足以缓存整个RDD，就进行部分缓存，将内存容纳不下的分区存储到磁盘上。

另外，RDD支持粗粒度和细粒度的读操作。RDD上的很多函数操作（如count和collect等）都是批量读操作，即扫描整个数据集，可以将任务分配到距离数据最近的节点上。同

时，RDD也支持细粒度操作，即在哈希或范围分区的RDD上执行关键字查找。

后续将算子从两个维度结合在3.3节对RDD算子进行详细介绍。

1）Transformations（变换）和Action（行动）算子维度。

2）在Transformations算子中再将数据类型维度细分为：Value数据类型和Key-Value对数据类型的Transformations算子。Value型数据的算子封装在RDD类中可以直接使用，Key-Value对数据类型的算子封装于PairRDDFunctions类中，用户需要引入import org.apache.spark.SparkContext._才能够使用。进行这样的细分是由于不同的数据类型处理思想不太一样，同时有些算子是不同的。

[1] 注意，这里的DSM，不仅指传统的共享内存系统，还包括那些通过分布式哈希表或分布式文件系统进行数据共享的系统，如Piccolo。

[2] 参见论文：Resilient Distributed Datasets:A Fault-Tolerant Abstraction for In-Memory Cluster Computing。

3.2.3 Spark的数据存储

Spark数据存储的核心是弹性分布式数据集（RDD）。RDD可以被抽象地理解为一个大的数组（Array），但是这个数组是分布在集群上的。逻辑上RDD的每个分区叫一个Partition。

在Spark的执行过程中，RDD经历一个个的Transfomation算子之后，最后通过Action算子进行触发操作。逻辑上每经历一次变换，就会将RDD转换为一个新的RDD，RDD之间通过Lineage产生依赖关系，这个关系在容错中有很重要的作用。变换的输入和输出都是RDD。RDD会被划分成很多的分区分布到集群的多个节点中。分区是个逻辑概念，变换前后的新旧分区在物理上可能是同一块内存存储。这是很重要的优化，以防止函数式数据不变性（immutable）导致的内存需求无限扩张。有些RDD是计算的中间结果，其分区并不一定有相应的内存或磁盘数据与之对应，如果要迭代使用数据，可以调cache（）函数缓存数据。

图3-2为RDD的数据存储模型。

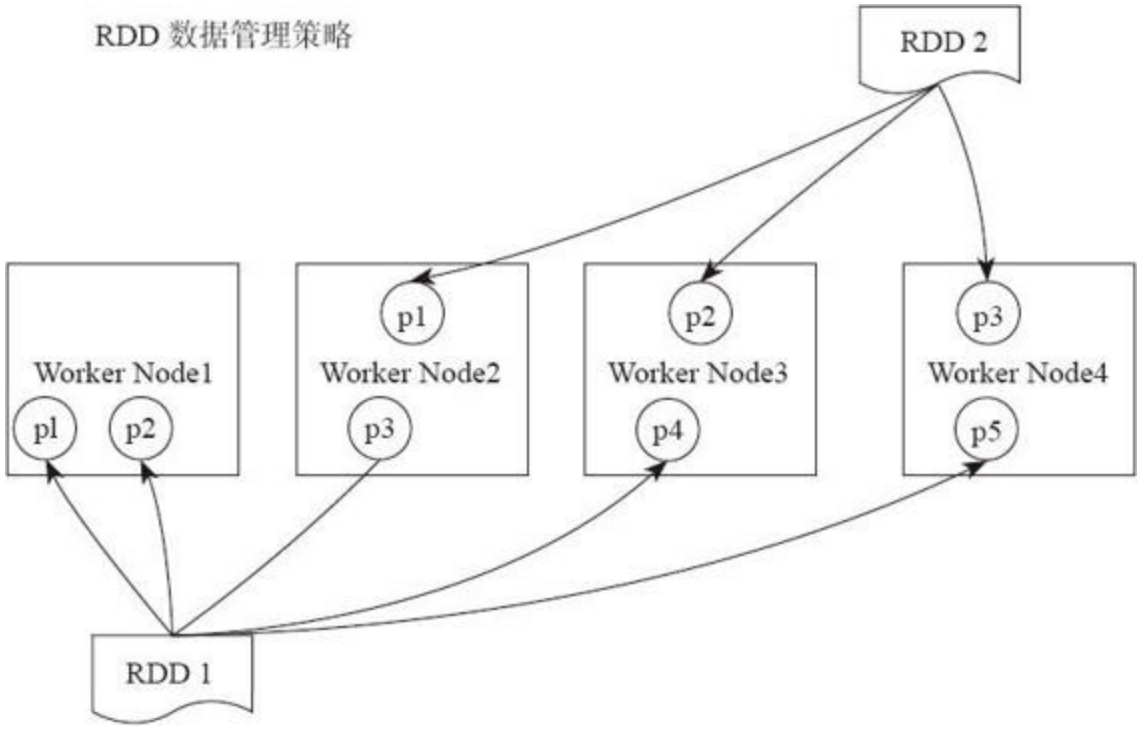

图3-2　RDD数据管理模型

图3-2中的RDD_1含有5个分区（p1、p2、p3、p4、p5），分别存储在4个节点（Node1、node2、Node3、Node4）中。RDD_2含有3个分区（p1、p2、p3），分布在3个节点（Node1、Node2、Node3）中。

在物理上，RDD对象实质上是一个元数据结构，存储着Block、Node等的映射关系，以及其他的元数据信息。一个RDD就是一组分区，在物理数据存储上，RDD的每个分区对应的就是一个Block，Block可以存储在内存，当内存不够时可以存储到磁盘上。

每个Block中存储着RDD所有数据项的一个子集，暴露给用户的可以是一个Block的迭代器（例如，用户可以通过mapPartitions获得分区迭代器进行操作），也可以就是一个数据项（例如，通过map函数对每个数据项并行计算）。本书会在后面章节具体介绍数据管理的底层实现细节。

如果是从HDFS等外部存储作为输入数据源，数据按照HDFS中的数据分布策略进行数据分区，HDFS中的一个Block对应Spark的一个分区。同时Spark支持重分区，数据通过Spark默认的或者用户自定义的分区器决定数据块分布在哪些节点。例如，支持Hash分区（按照数据项的Key值取Hash值，Hash值相同的元素放入同一个分区之内）和Range分区（将属于同一数据范围的数据放入同一分区）等分区策略。

下面具体介绍这些算子的功能。

3.3 Spark算子分类及功能

本节将主要介绍Spark算子的作用，以及算子的分类。

1.Saprk算子的作用

图3-3描述了Spark的输入、运行转换、输出。在运行转换中通过算子对RDD进行转换。算子是RDD中定义的函数，可以对RDD中的数据进行转换和操作。

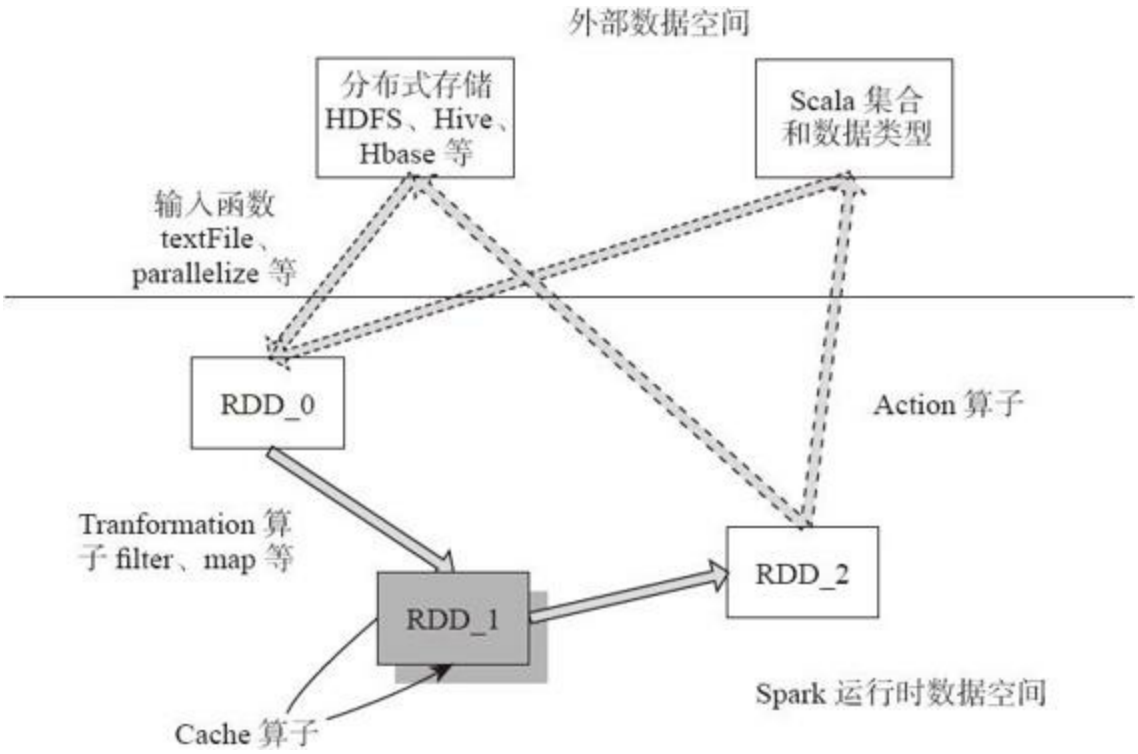

图3-3　Spark算子和数据空间

1）输入：在Spark程序运行中，数据从外部数据空间（如分布式存储：textFile读取HDFS等，parallelize方法输入Scala集合或数据）输入Spark，数据进入Spark运行时数据空间，转化为Spark中的数据块，通过BlockManager进行管理。

2）运行：在Spark数据输入形成RDD后便可以通过变换算子，如fliter等，对数据进行操作并将RDD转化为新的RDD，通过Action算子，触发Spark提交作业。如果数据需要复用，可以通过Cache算子，将数据缓存到内存。

3）输出：程序运行结束数据会输出Spark运行时空间，存储到分布式存储中（如saveAsTextFile输出到HDFS），或Scala数据或集合中（collect输出到Scala集合，count返回Scala int型数据）。

Spark的核心数据模型是RDD，但RDD是个抽象类，具体由各子类实现，如MappedRDD、ShuffledRDD等子类。Spark将常用的大数据操作都转化成为RDD的子类。

2.算子的分类

大致可以分为三大类算子。

1）Value数据类型的Transformation算子，这种变换并不触发提交作业，针对处理的数据项是Value型的数据。

2）Key-Value数据类型的Transfromation算子，这种变换并不触发提交作业，针对处理的数据项是Key-Value型的数据对。

3）Action算子，这类算子会触发SparkContext提交Job作业。

下面分别对这3类算子进行详细介绍。

3.3.1 Value型Transformation算子

处理数据类型为Value型的Transformation算子可以根据RDD变换算子的输入分区与输出分区关系分为以下几种类型。

1）输入分区与输出分区一对一型。

2）输入分区与输出分区多对一型。

3）输入分区与输出分区多对多型。

4）输出分区为输入分区子集型。

5）还有一种特殊的输入与输出分区一对一的算子类型：Cache型。Cache算子对RDD分区进行缓存。

1.输入分区与输出分区一对一型

（1）map

将原来RDD的每个数据项通过map中的用户自定义函数f映射转变为一个新的元素。源码中的map算子相当于初始化一个RDD，新RDD叫作MappedRDD（this，sc.clean（f））。

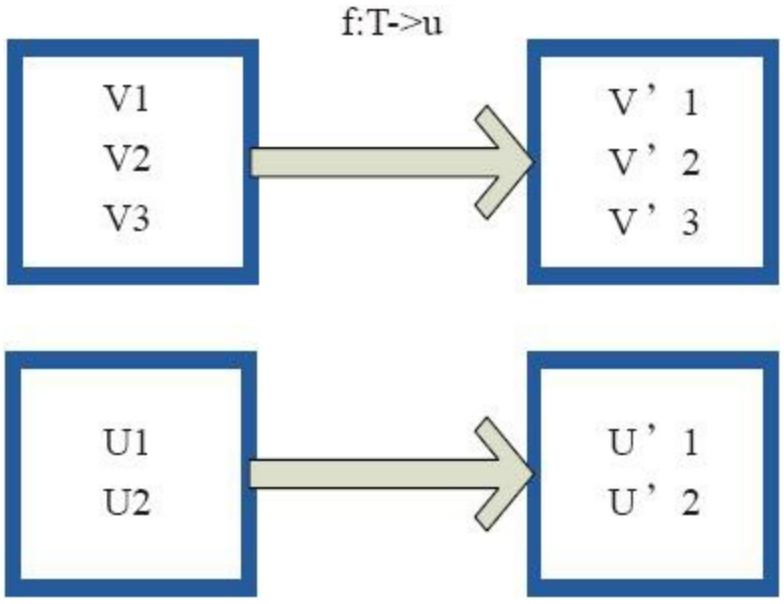

图3-4　map算子对RDD转换

图3-4中的每个方框表示一个RDD分区，左侧的分区经过用户自定义函数f：T->U映射为右侧的新的RDD分区。但是实际只有等到Action算子触发后，这个f函数才会和其他函数在一个Stage中对数据进行运算。V1输入f转换输出V'1。

（2）flatMap

将原来RDD中的每个元素通过函数f转换为新的元素，并将生成的RDD的每个集合中的元素合并为一个集合。内部创建FlatMappedRDD（this，sc.clean（f））。

图3-5中小方框表示RDD的一个分区，对分区进行flatMap函数操作，flatMap中传入的函数为f:T->U，T和U可以是任意的数据类型。将分区中的数据通过用户自定义函数f转换为新的数据。外部大方框可以认为是一个RDD分区，小方框代表一个集合。V1、V2、V3在一个集合作为RDD的一个数据项，转换为V'1、V'2、V'3后，将结合拆散，形成为RDD中的数据项。

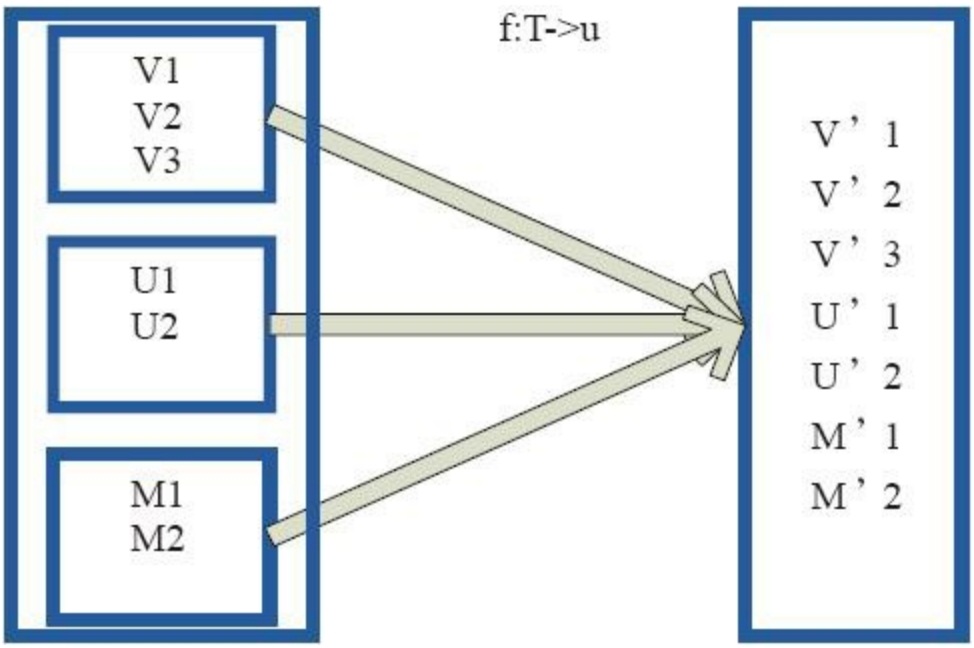

图3-5　flapMap算子对RDD转换

（3）mapPartitions

mapPartitions函数获取到每个分区的迭代器，在函数中通过这个分区整体的迭代器对整个分区的元素进行操作。内部实现是生成MapPartitionsRDD。图3-6中的方框代表一个RDD分区。

图3-6中，用户通过函数f（iter）=>iter.filter（_>=3）对分区中的所有数据进行过滤，>=3的数据保留。一个方块代表一个RDD分区，含有1、2、3的分区过滤只剩下元素3。

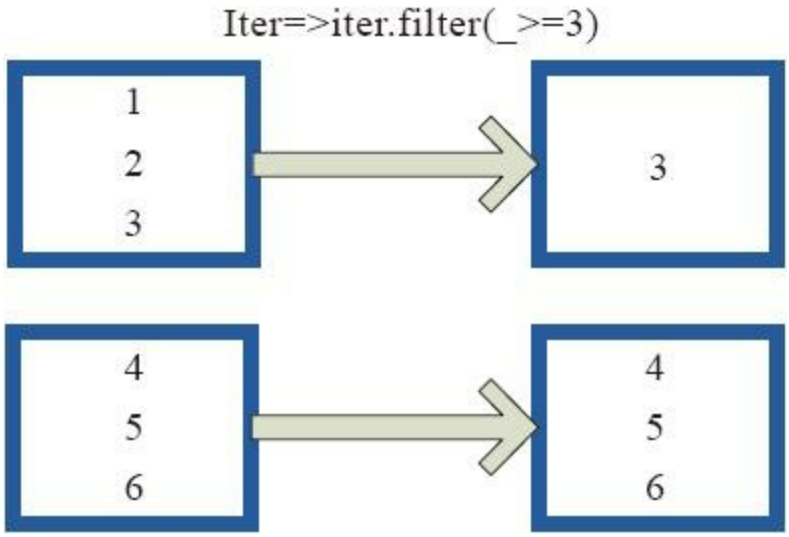

图3-6　mapPartitions算子对RDD转换

（4）glom

glom函数将每个分区形成一个数组，内部实现是返回的GlommedRDD。图3-7中的每个方框代表一个RDD分区。

图3-7中的方框代表一个分区。该图表示含有V1、V2、V3的分区通过函数glom形成一个数组Array[（V1），（V2），（V3）]。

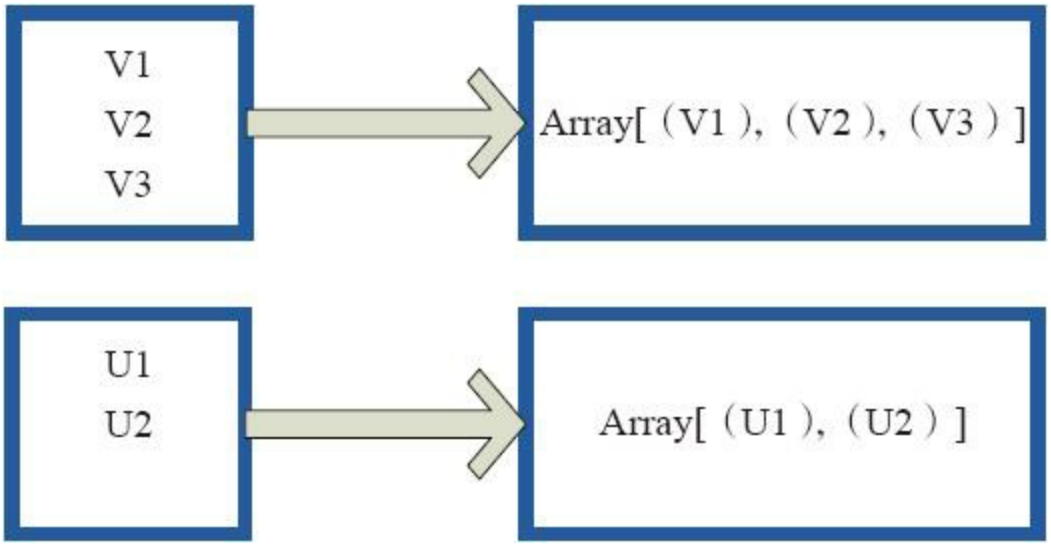

图3-7　glom算子对RDD转换

2.输入分区与输出分区多对一型

（1）union

使用union函数时需要保证两个RDD元素的数据类型相同，返回的RDD数据类型和被合并的RDD元素数据类型相同，并不进行去重操作，保存所有元素。如果想去重，可以使用distinct（）。++符号相当于uion函数操作。

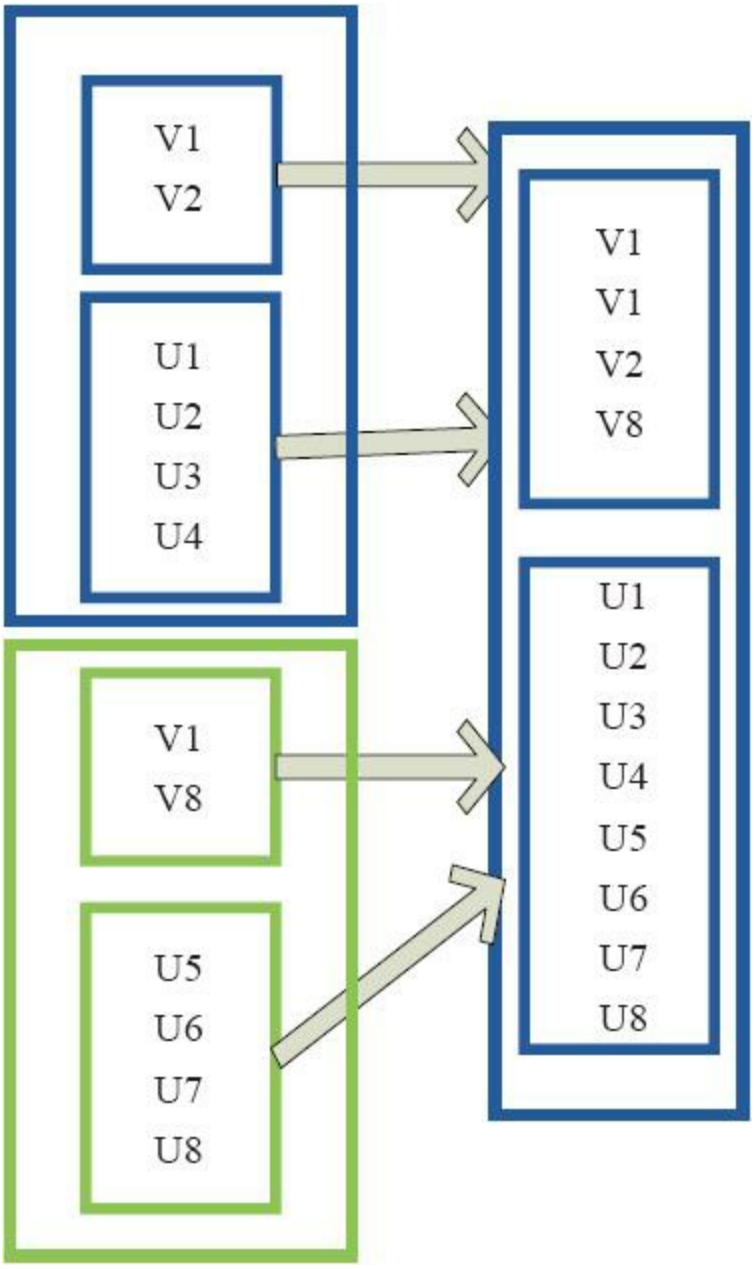

图3-8　union算子对RDD转换

图3-8中左侧的大方框代表两个RDD，大方框内的小方框代表RDD的分区。右侧大方框代表合并后的RDD，大方框内的小方框代表分区。含有V1，V2...U4的RDD和含有V1，V8...U8的RDD合并所有元素形成一个RDD。V1、V1、V2、V8形成一个分区，其他元素同理进行合并。

（2）cartesian

对两个RDD内的所有元素进行笛卡尔积操作。操作后，内部实现返回CartesianRDD。图3-9中左侧的大方框代表两个RDD，大方框内的小方框代表RDD的分区。右侧大方框代表

合并后的RDD，大方框内的小方框代表分区。

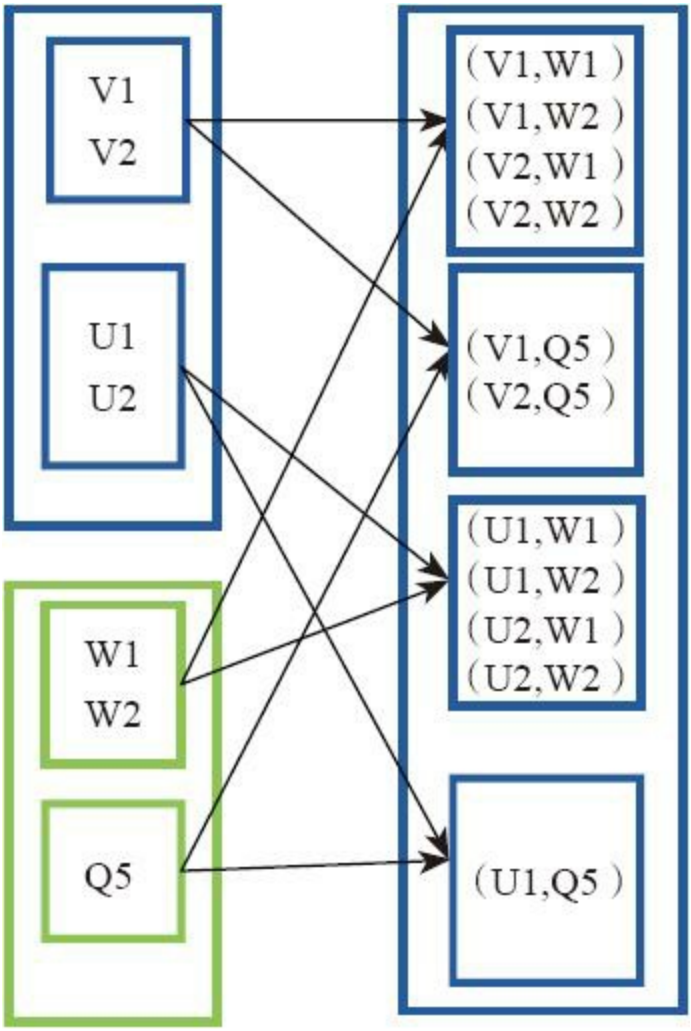

图3-9　cartesian算子对RDD转换

图3-9中的大方框代表RDD，大方框中的小方框代表RDD分区。例如，V1和另一个RDD中的W1、W2、Q5进行笛卡尔积运算形成（V1，W1）、（V1，W2）、（V1，Q5）。

3.输入分区与输出分区多对多型

groupBy：将元素通过函数生成相应的Key，数据就转化为Key-Value格式，之后将Key相同的元素分为一组。

函数实现如下。

①sc.clean（ ）函数将用户函数预处理：

```
val cleanF = sc.clean(f)
```

②对数据map进行函数操作，最后再对groupByKey进行分组操作。

```
this.map(t => (cleanF(t), t)).groupByKey(p)
```

其中，p中确定了分区个数和分区函数，也就决定了并行化的程度。图3-10中的方框代表RDD分区。

图3-10中的方框代表一个RDD分区，相同key的元素合并到一个组。例如，V1，V2合并为一个Key-Value对，其中key为“V”，Value为“V1，V2”，形成V，Seq（V1，V2）。

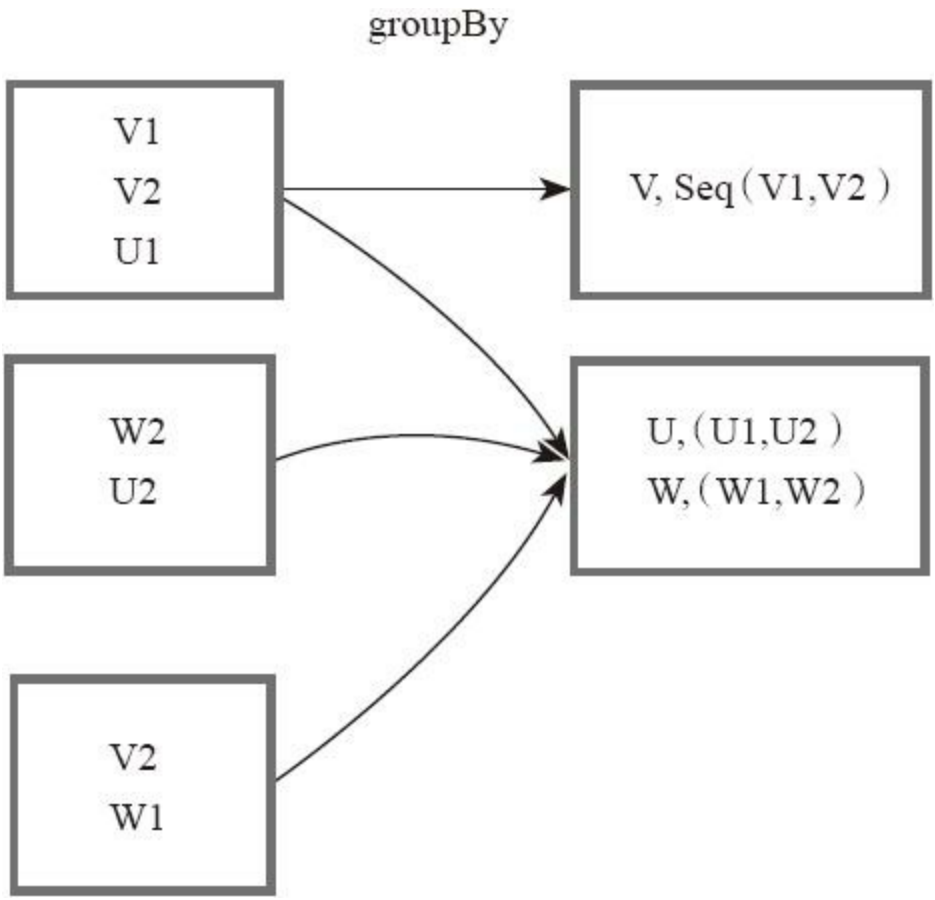

图3-10　groupBy算子对RDD转换

4.输出分区为输入分区子集型

（1）filter

filter的功能是对元素进行过滤，对每个元素应用f函数，返回值为true的元素在RDD中保

留，返回为false的将过滤掉。内部实现相当于生成FilteredRDD（this，sc.clean（f））。

下面代码为函数的本质实现。

```
def filter(f:T=>Boolean):RDD[T]=new FilteredRDD(this,sc.clean(f))
```

图3-11中的每个方框代表一个RDD分区。T可以是任意的类型。通过用户自定义的过滤函数f，对每个数据项进行操作，将满足条件，返回结果为true的数据项保留。例如，过滤掉V2、V3保留了V1，将区分命名为V1'。

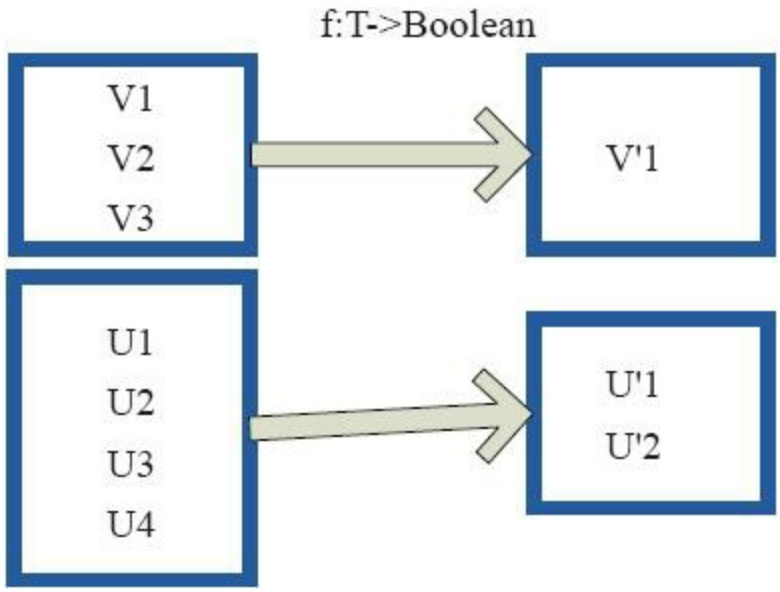

图3-11　filter算子对RDD转换

（2）distinct

distinct将RDD中的元素进行去重操作。图3-12中的方框代表RDD分区。

图3-12中的每个方框代表一个分区，通过distinct函数，将数据去重。例如，重复数据V1、V1去重后只保留一份V1。

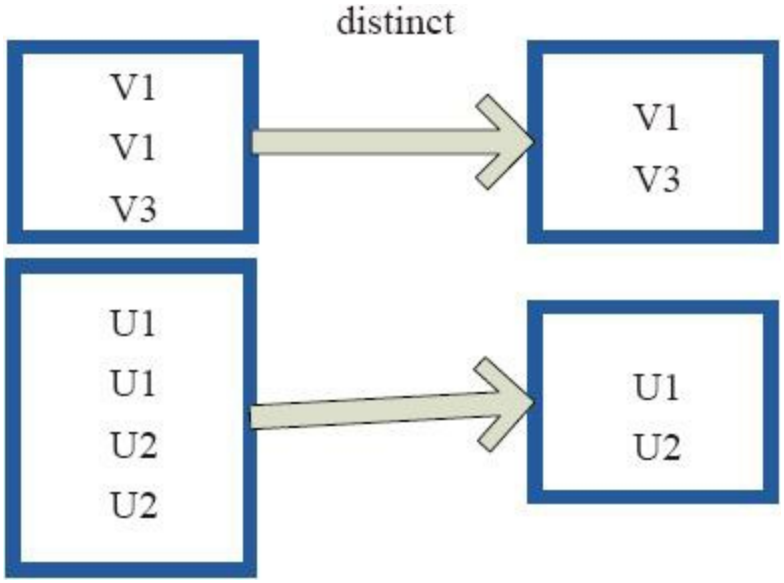

图3-12　distinct算子对RDD转换

（3）subtract

subtract相当于进行集合的差操作，RDD 1去除RDD 1和RDD 2交集中的所有元素。

图3-13中左侧的大方框代表两个RDD，大方框内的小方框代表RDD的分区。右侧大方框代表合并后的RDD，大方框内的小方框代表分区。V1在两个RDD中均有，根据差集运算规则，新RDD不保留，V2在第一个RDD有，第二个RDD没有，则在新RDD元素中包含V2。

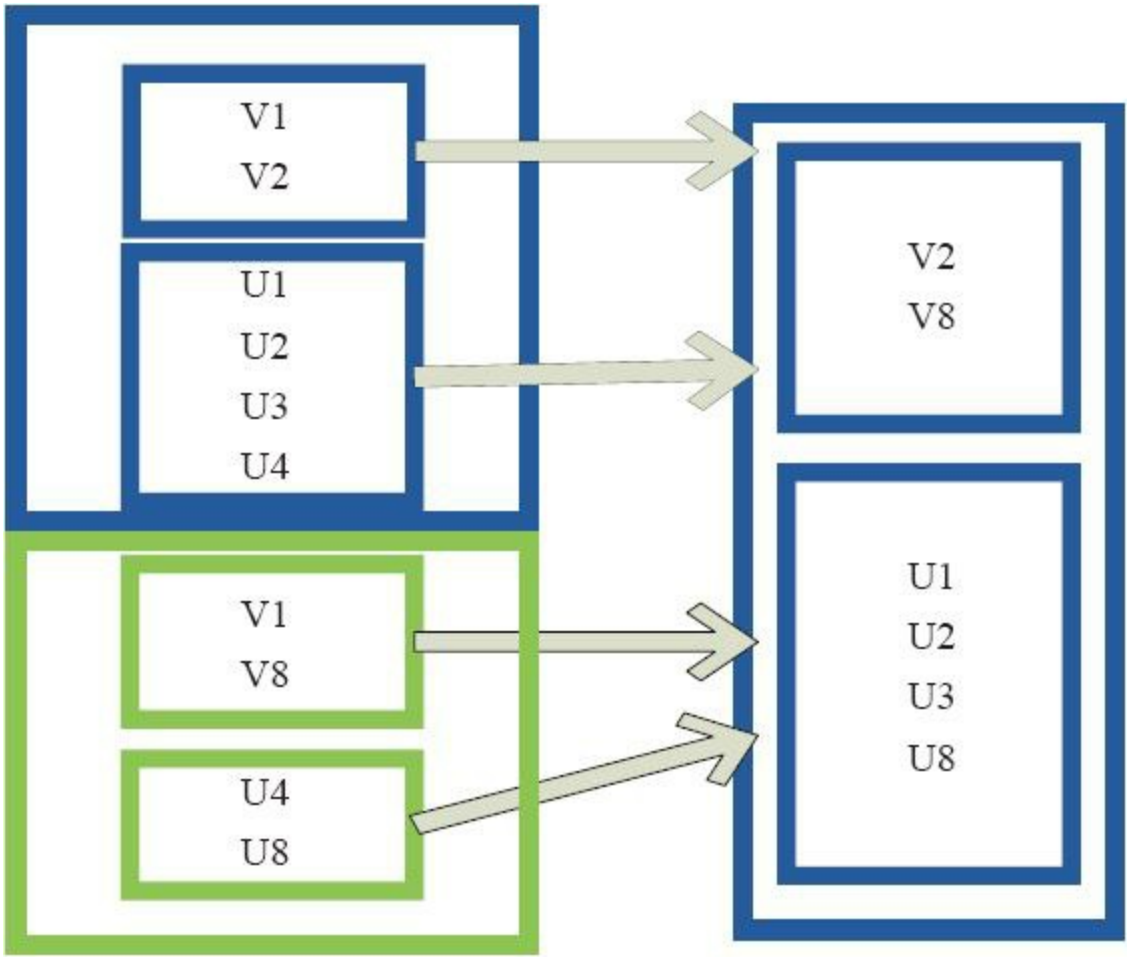

图3-13　subtract算子对RDD转换

（4）sample

sample将RDD这个集合内的元素进行采样，获取所有元素的子集。用户可以设定是否有放回的抽样、百分比、随机种子，进而决定采样方式。

内部实现是生成SampledRDD（withReplacement，fraction，seed）。

函数参数设置如下。

·withReplacement=true，表示有放回的抽样；

·withReplacement=false，表示无放回的抽样。

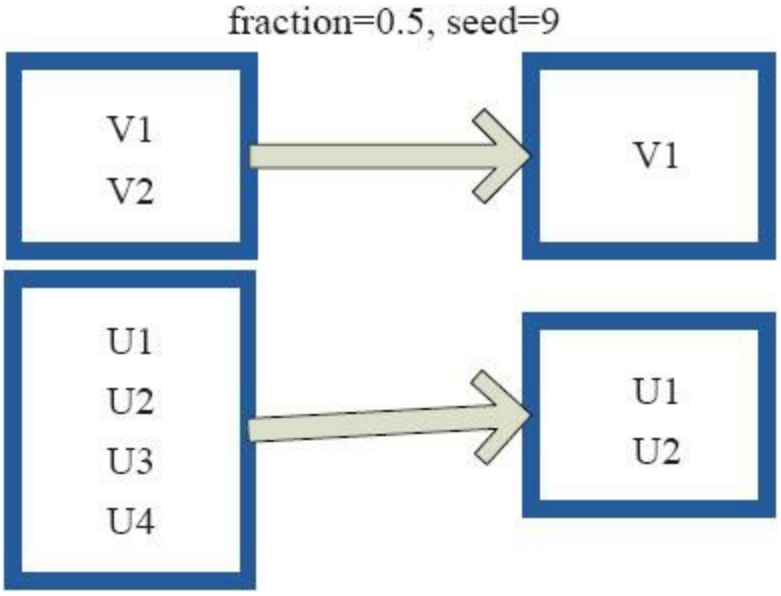

图3-14　sample算子对RDD转换

图3-14中的每个方框是一个RDD分区。通过sample函数，采样50%的数据。V1、V2、U1、U2、U3、U4采样出数据V1和U1、U2，形成新的RDD。

（5）takeSample

takeSample（）函数和上面的sample函数是一个原理，但是不使用相对比例采样，而是按设定的采样个数进行采样，同时返回结果不再是RDD，而是相当于对采样后的数据进行Collect（），返回结果的集合为单机的数组。

图3-15中左侧的方框代表分布式的各个节点上的分区，右侧方框代表单机上返回的结果数组。通过takeSample对数据采样，设置为采样一份数据，返回结果为V1。

5.Cache型

（1）cache

cache将RDD元素从磁盘缓存到内存，相当于persist（MEMORY_ONLY）函数的功能。图3-14中的方框代表RDD分区。

图3-16中的每个方框代表一个RDD分区，左侧相当于数据分区都存储在磁盘，通过

cache算子将数据缓存在内存。

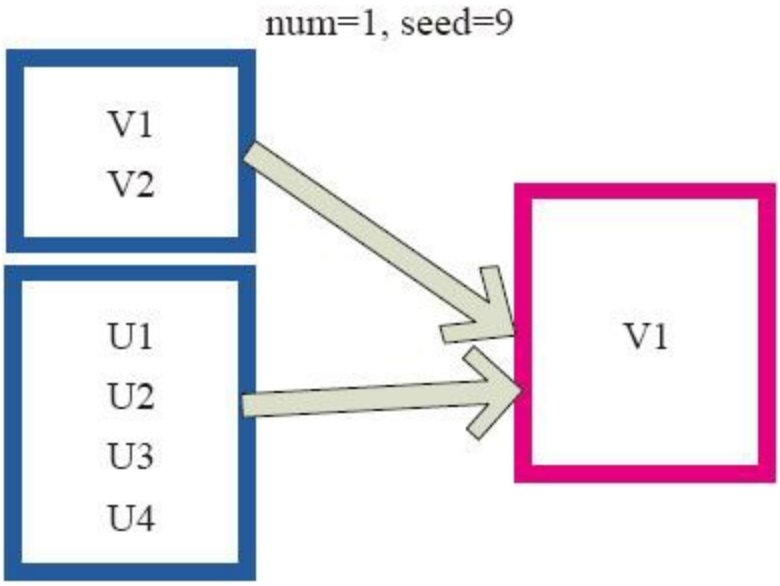

图3-15　takeSample算子对RDD转换

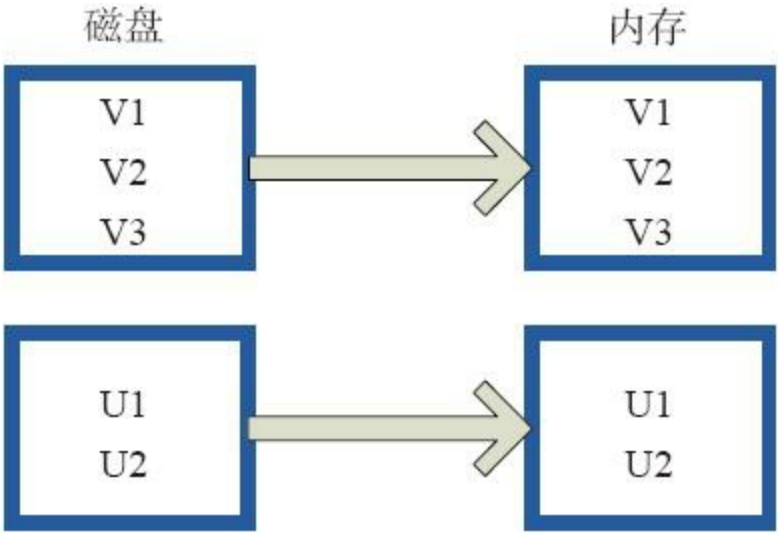

图3-16　cache算子对RDD转换

（2）persist

persist函数对RDD进行缓存操作。数据缓存在哪里由StorageLevel枚举类型确定。有以下几种类型的组合（见图3-15），DISK代表磁盘，MEMORY代表内存，SER代表数据是否进行序列化存储。

下面为函数定义，StorageLevel是枚举类型，代表存储模式，用户可以通过图3-17按需选择。

```
persist ( newLevel : StorageLevel )
```

图3-17中列出persist函数可以缓存的模式。例如，MEMORY_AND_DISK_SER代表数据可以存储在内存和磁盘，并且以序列化的方式存储。其他同理。

```
val DISK_ONLY: StorageLevel
val DISK_ONLY_2: StorageLevel
val MEMORY_AND_DISK: StorageLevel
val MEMORY_AND_DISK_2: StorageLevel
val MEMORY_AND_DISK_SER: StorageLevel
val MEMORY_AND_DISK_SER_2: StorageLevel
val MEMORY_ONLY: StorageLevel
val MEMORY_ONLY_2: StorageLevel
val MEMORY_ONLY_SER: StorageLevel
val MEMORY_ONLY_SER_2: StorageLevel
val NONE: StorageLevel
val OFF_HEAP: StorageLevel
```

图3-17　persist算子对RDD转换

图3-18中的方框代表RDD分区。disk代表存储在磁盘，mem代表存储在内存。数据最初全部存储在磁盘，通过persist（MEMORY_AND_DISK）将数据缓存到内存，但是有的分区无法容纳在内存，例如：图3-18中将含有V1，V2，V3的RDD存储到磁盘，将含有U1，U2的RDD仍旧存储在内存。

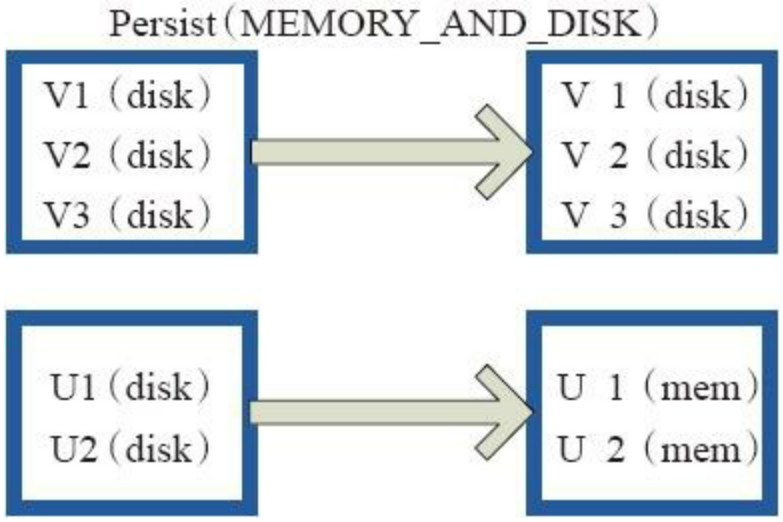

图3-18　Persist算子对RDD转换

3.3.2 Key-Value型Transformation算子

Transformation处理的数据为Key-Value形式的算子，大致可以分为3种类型：输入分区与输出分区一对一、聚集、连接操作。

1.输入分区与输出分区一对一

mapValues：针对（Key，Value）型数据中的Value进行Map操作，而不对Key进行处理。

图3-19中的方框代表RDD分区。a=>a+2代表只对（V1，1）数据中的1进行加2操作，返回结果为3。

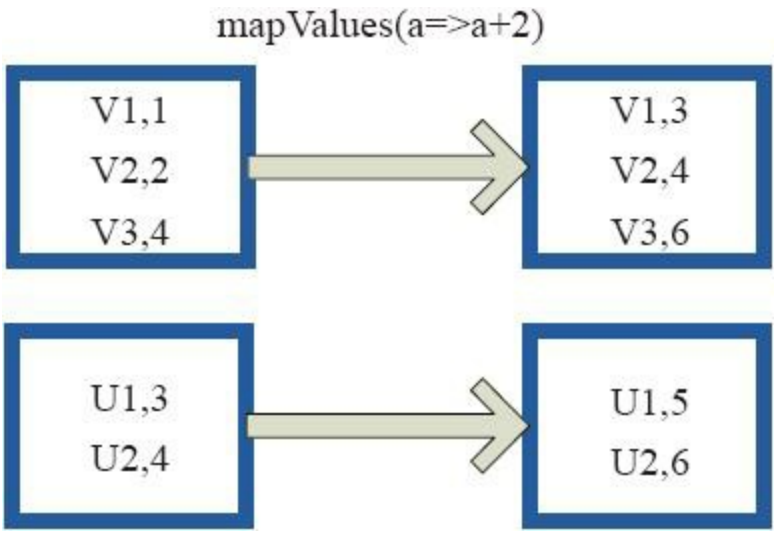

图3-19　mapValues算子RDD对转换

2.对单个RDD或两个RDD聚集

（1）单个RDD聚集

1）combineByKey。

定义combineByKey算子的代码如下。

```
combineByKey[C](createCombiner: (V) => C,
mergeValue: (C, V) => C,
mergeCombiners: (C, C) => C,
partitioner: Partitioner
mapSideCombine: Boolean = true,
serializer: Serializer =null): RDD[(K, C)]
```

说明：

·createCombiner：V=>C，在C不存在的情况下，如通过V创建seq C。

·mergeValue：（C，V）=>C，当C已经存在的情况下，需要merge，如把item V加到seq C中，或者叠加。

·mergeCombiners：（C，C）=>C，合并两个C。

·partitioner：Partitioner（分区器），Shuffle时需要通过Partitioner的分区策略进行分区。

·mapSideCombine：Boolean=true，为了减小传输量，很多combine可以在map端先做。例如，叠加可以先在一个partition中把所有相同的Key的Value叠加，再shuffle。

·serializerClass：String=null，传输需要序列化，用户可以自定义序列化类。

例如，相当于将元素为（Int，Int）的RDD转变为了（Int，Seq[Int]）类型元素的RDD。

图3-20中的方框代表RDD分区。通过combineByKey，将（V1，2）、（V1，1）数据合并为（V1，Seq（2，1））。

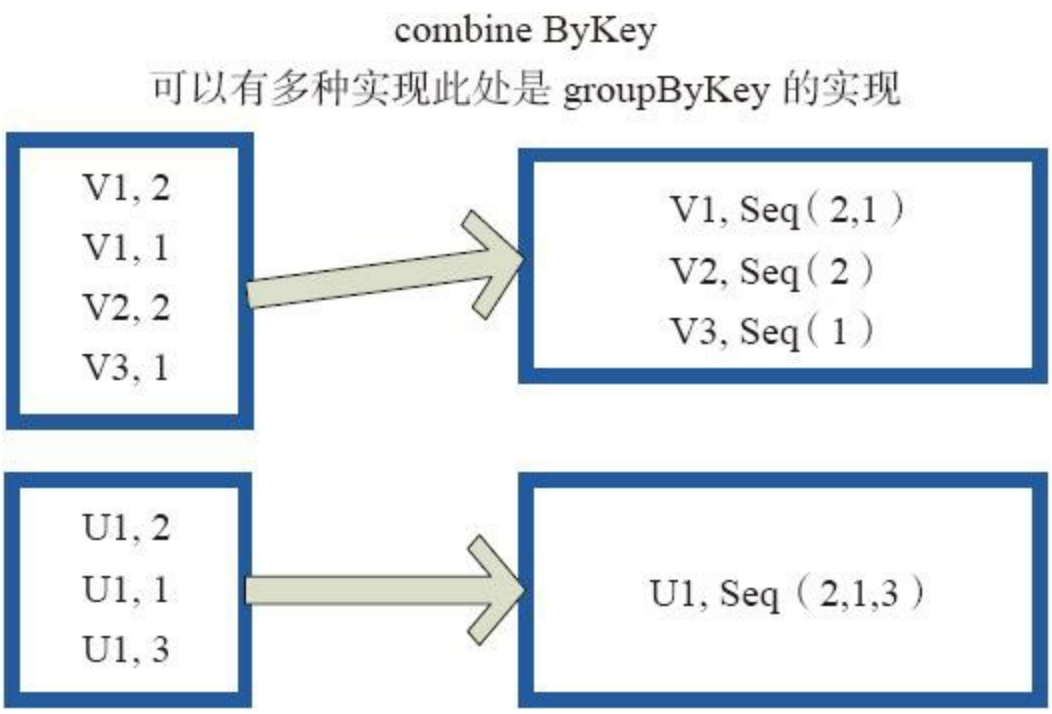

图3-20　comBineByKey算子对RDD转换

2）reduceByKey。

reduceByKey是更简单的一种情况，只是两个值合并成一个值，所以createCombiner很简单，就是直接返回v，而mergeValue和mergeCombiners的逻辑相同，没有区别。

函数实现代码如下。

```
def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)] = {
  combineByKey[V]((v: V) => v, func, func, partitioner)
}
```

图3-21中的方框代表RDD分区。通过用户自定义函数（A，B）=>（A+B），将相同Key的数据（V1，2）、（V1，1）的value相加，结果为（V1，3）。

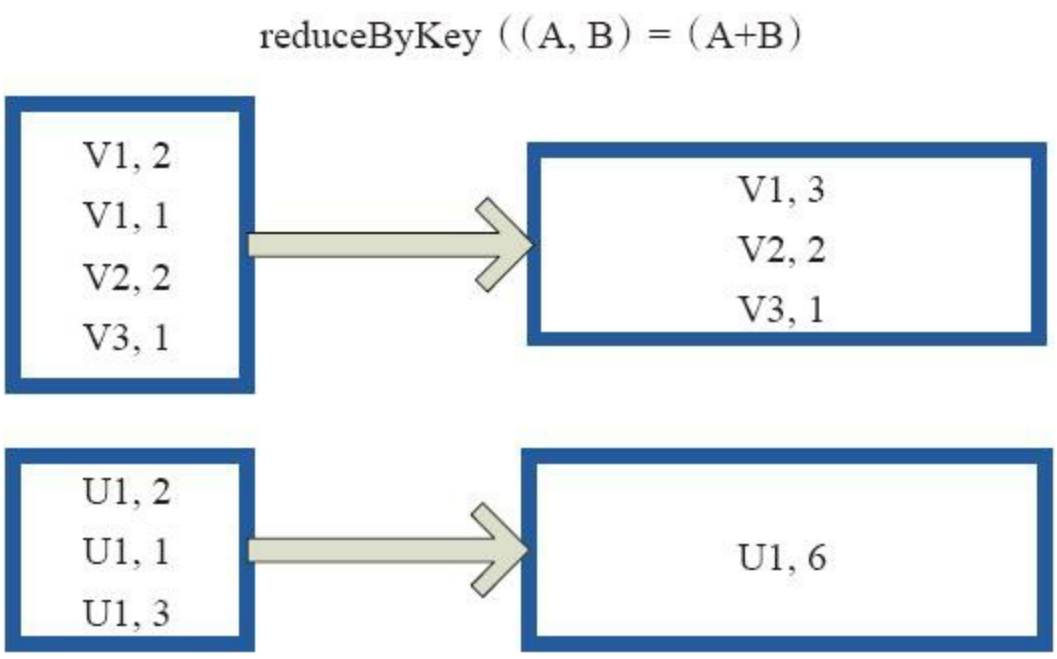

图3-21　reduceByKey算子对RDD转换

3）partitionBy。

partitionBy函数对RDD进行分区操作。

函数定义如下。

```
partitionBy(partitioner:Partitioner)
```

如果原有RDD的分区器和现有分区器（partitioner）一致，则不重分区，如果不一致，则相当于根据分区器生成一个新的ShuffledRDD。

图3-22中的方框代表RDD分区。通过新的分区策略将原来在不同分区的V1、V2数据都合并到了一个分区。

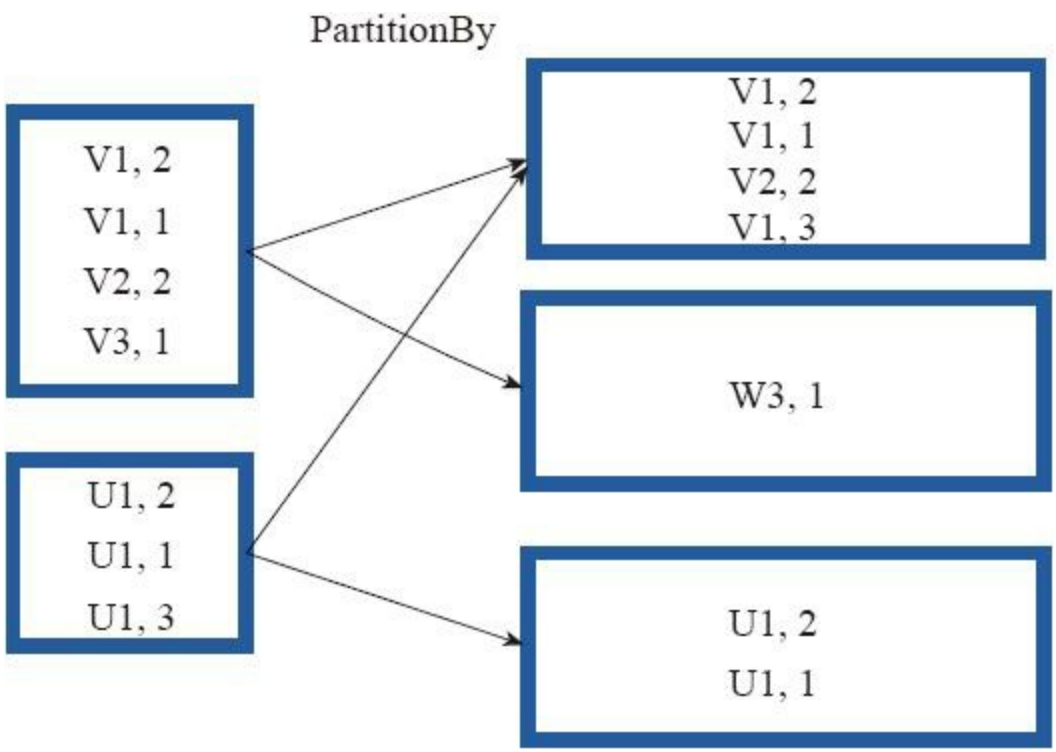

图3-22　partitionBy算子对RDD转换

（2）对两个RDD进行聚集

cogroup函数将两个RDD进行协同划分，cogroup函数的定义如下。

```
cogroup[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Iterable[V], Iterable[W]))]
```

对在两个RDD中的Key-Value类型的元素，每个RDD相同Key的元素分别聚合为一个集合，并且返回两个RDD中对应Key的元素集合的迭代器。

```
(K, (Iterable[V], Iterable[W]))
```

其中，Key和Value，Value是两个RDD下相同Key的两个数据集合的迭代器所构成的元组。

图3-23中的大方框代表RDD，大方框内的小方框代表RDD中的分区。将RDD1中的数据（U1，1）、（U1，2）和RDD2中的数据（U1，2）合并为（U1，（（1，2），（2））），。

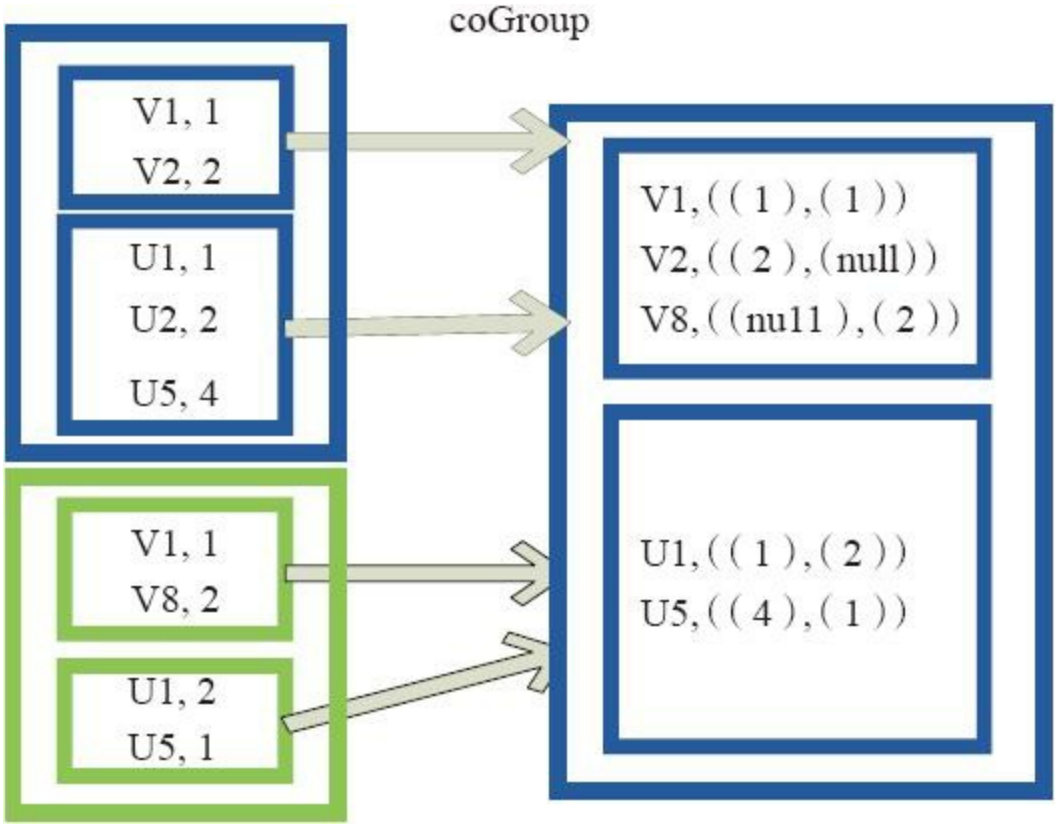

图3-23　Cogroup算子对RDD转换

3.连接

（1）join

join对两个需要连接的RDD进行cogroup函数操作，cogroup原理请见上文。cogroup操作之后形成的新RDD，对每个key下的元素进行笛卡尔积操作，返回的结果再展平，对应Key下的所有元组形成一个集合，最后返回RDD[（K，（V，W））]

下面代码为join的函数实现，本质是通过cogroup算子先进行协同划分，再通过flatMapValues将合并的数据打散。

```
this.cogroup(other, partitioner).flatMapValues { case (vs, ws) =>
      for (v <- vs; w <- ws) yield (v, w) }
```

图3-24是对两个RDD的join操作示意图。大方框代表RDD，小方框代表RDD中的分区。函数对拥有相同Key的元素（例如V1）为Key，以做连接后的数据结果为（V1，（1，1））和（V1，（1，2））。

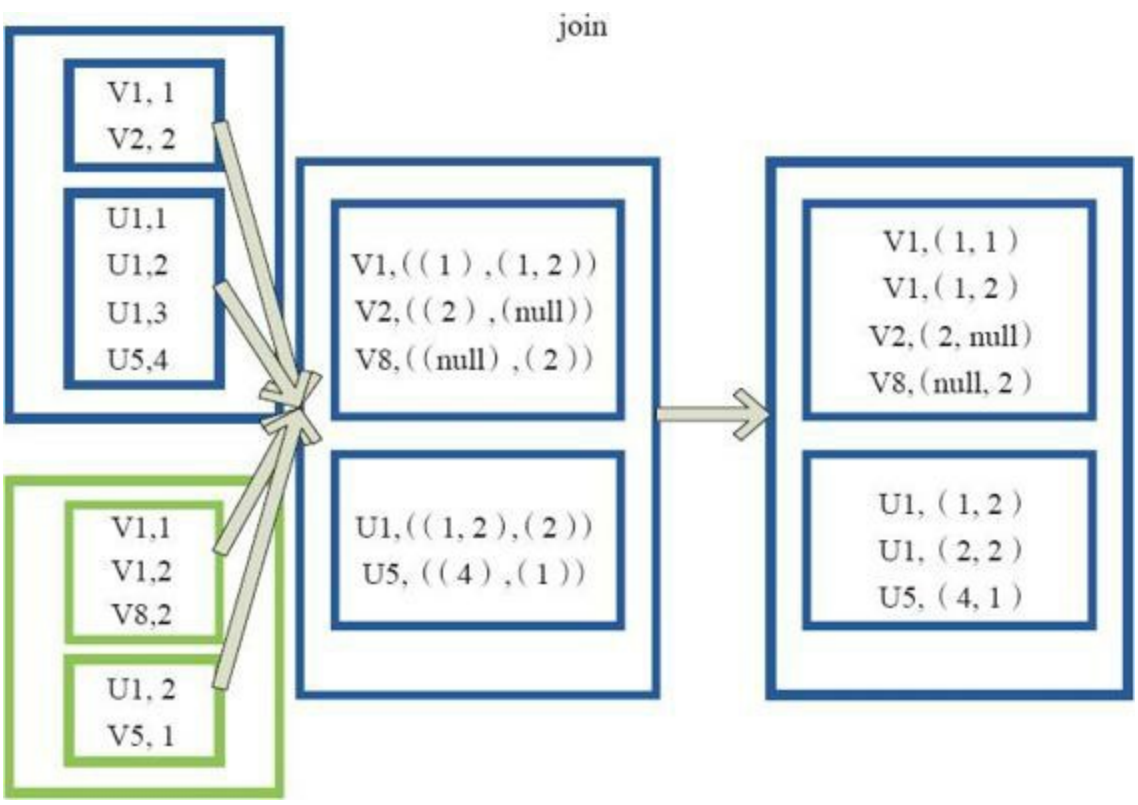

图3-24　join算子对RDD转换

（2）leftOutJoin和rightOutJoin

LeftOutJoin（左外连接）和RightOutJoin（右外连接）相当于在join的基础上先判断一侧的RDD元素是否为空，如果为空，则填充为空。如果不为空，则将数据进行连接运算，并返回结果。

下面代码是leftOutJoin的实现。

```
if (ws.isEmpty) {
       vs.map(v => (v, None))
     } else {
       for (v <- vs; w <- ws) yield (v, Some(w))
     }
```

3.3.3 Actions算子

本质上在Actions算子中通过SparkContext执行提交作业的runJob操作，触发了RDD DAG的执行。

例如，Actions算子collect函数的代码如下，感兴趣的读者可以顺着这个入口进行源码剖析。

```
/*返回这个RDD的所有数据，结果以数组形式存储*/
  def collect( ) : Array[T] = {
/*提交Job*/
    val results = sc.runJob( this , ( iter : Iterator[T] ) => iter.toArray )
    Array.concat( results : _* )
  }
```

下面根据Action算子的输出空间将Action算子进行分类：无输出、HDFS、Scala集合和数据类型。

1.无输出

（1）foreach

对RDD中的每个元素都应用f函数操作，不返回RDD和Array，而是返回Uint。

图3-25表示foreach算子通过用户自定义函数对每个数据项进行操作。本例中自定义函数为println（ ），控制台打印所有数据项。

2.HDFS

（1）saveAsTextFile

函数将数据输出，存储到HDFS的指定目录。

下面为函数的内部实现。

```
this.map(x => (NullWritable.get(), new Text(x.toString)))
.saveAsHadoopFile[TextOutputFormat[NullWritable, Text]](path)
```

将RDD中的每个元素映射转变为（Null，x.toString），然后再将其写入HDFS。

图3-26中左侧的方框代表RDD分区，右侧方框代表HDFS的Block。通过函数将RDD的每个分区存储为HDFS中的一个Block。

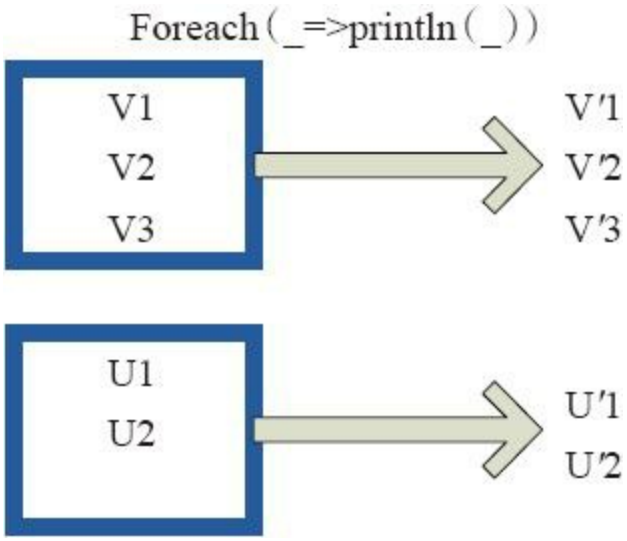

图3-25　foreach算子对RDD转换

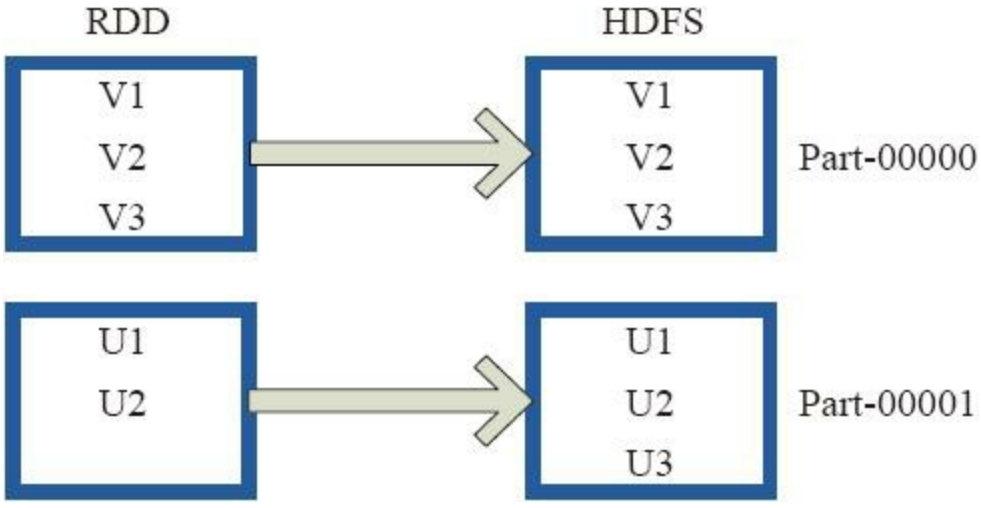

图3-26　saveAsHadoopFile算子对RDD转换

（2）saveAsObjectFile

saveAsObjectFile将分区中的每10个元素组成一个Array，然后将这个Array序列化，映射为（Null，BytesWritable（Y））的元素，写入HDFS为SequenceFile的格式。

下面代码为函数内部实现。

```
map(x=>(NullWritable.get(),new BytesWritable(Utils.serialize(x))))
```

图3-27中的左侧方框代表RDD分区，右侧方框代表HDFS的Block。通过函数将RDD的每个分区存储为HDFS上的一个Block。

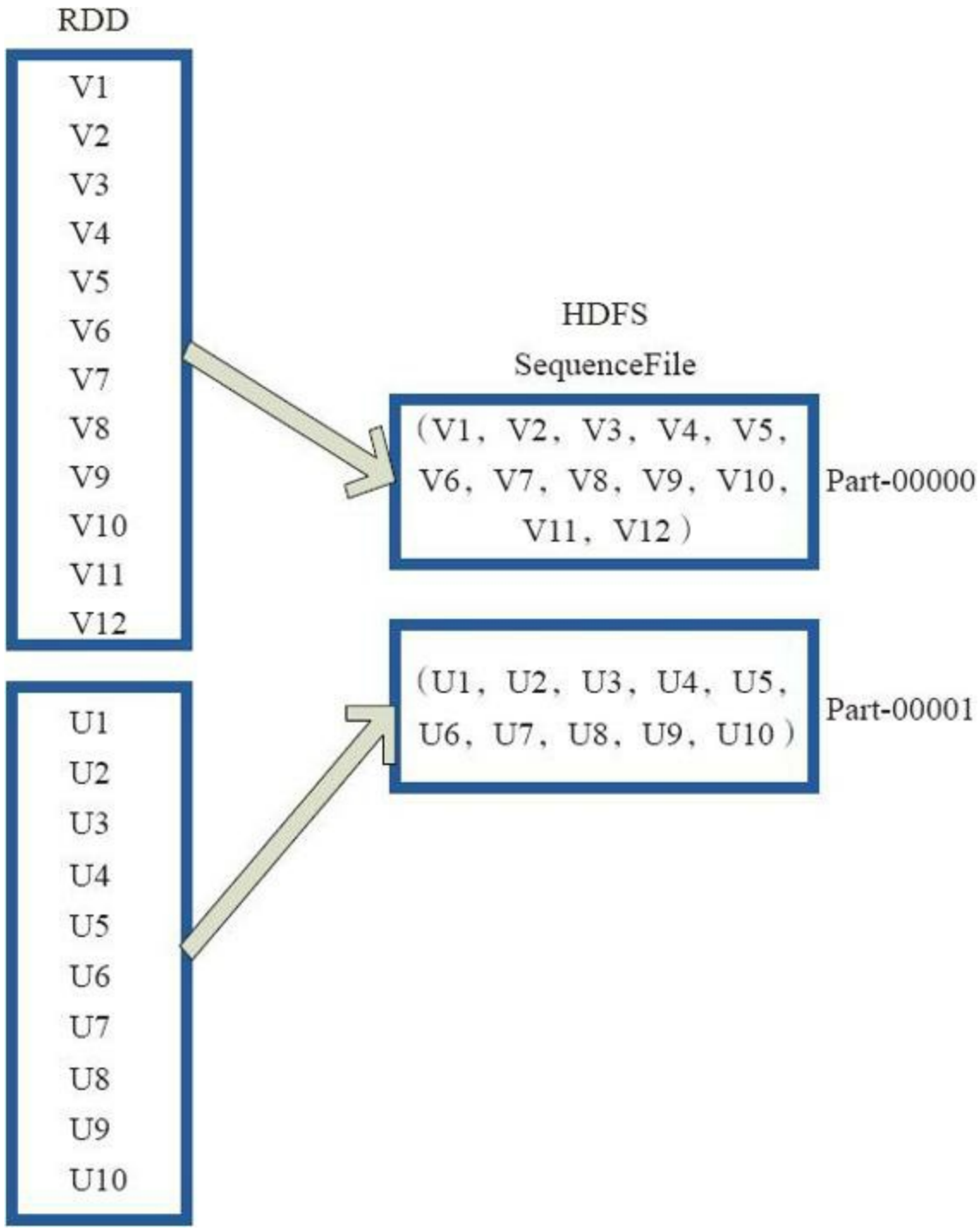

图3-27　saveAsObjectFile算子对RDD转换

3.Scala集合和数据类型

（1）collect

collect相当于toArray，toArray已经过时不推荐使用，collect将分布式的RDD返回为一个单机的scala Array数组。在这个数组上运用scala的函数式操作。

图3-28中的左侧方框代表RDD分区，右侧方框代表单机内存中的数组。通过函数操作，将结果返回到Driver程序所在的节点，以数组形式存储。

（2）collectAsMap

collectAsMap对（K，V）型的RDD数据返回一个单机HashMap。对于重复K的RDD元素，后面的元素覆盖前面的元素。

图3-29中的左侧方框代表RDD分区，右侧方框代表单机数组。数据通过collectAsMap函数返回给Driver程序计算结果，结果以HashMap形式存储。

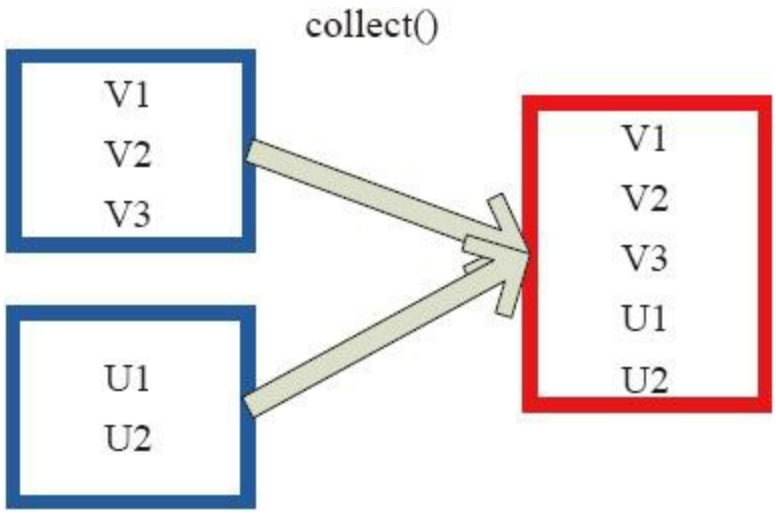

图3-28　Collect算子对RDD转换

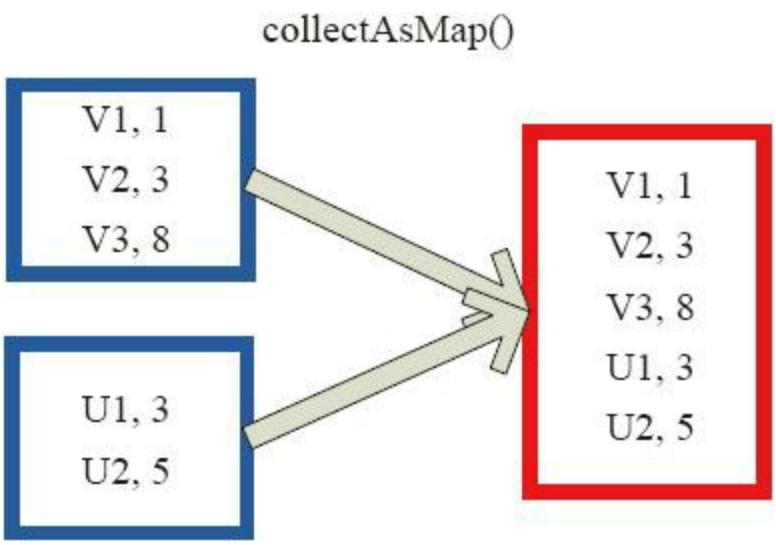

图3-29　collectAsMap算子对RDD转换

（3）reduceByKeyLocally

实现的是先reduce再collectAsMap的功能，先对RDD的整体进行reduce操作，然后再收集所有结果返回为一个HashMap。

（4）lookup

下面代码为lookup的声明。

```
lookup(key:K):Seq[V]
```

Lookup函数对（Key，Value）型的RDD操作，返回指定Key对应的元素形成的Seq。这个函数处理优化的部分在于，如果这个RDD包含分区器，则只会对应处理K所在的分区，然后返回由（K，V）形成的Seq。如果RDD不包含分区器，则需要对全RDD元素进行暴力扫描处理，搜索指定K对应的元素。

图3-30中的左侧方框代表RDD分区，右侧方框代表Seq，最后结果返回到Driver所在节点的应用中。

（5）count

count返回整个RDD的元素个数。内部函数实现如下。

```
Def count ( ) : Long=sc.runJob ( this , Utils.getIteratorSize_ ) .sum
```

在图3-31中，返回数据的个数为5。一个方块代表一个RDD分区。

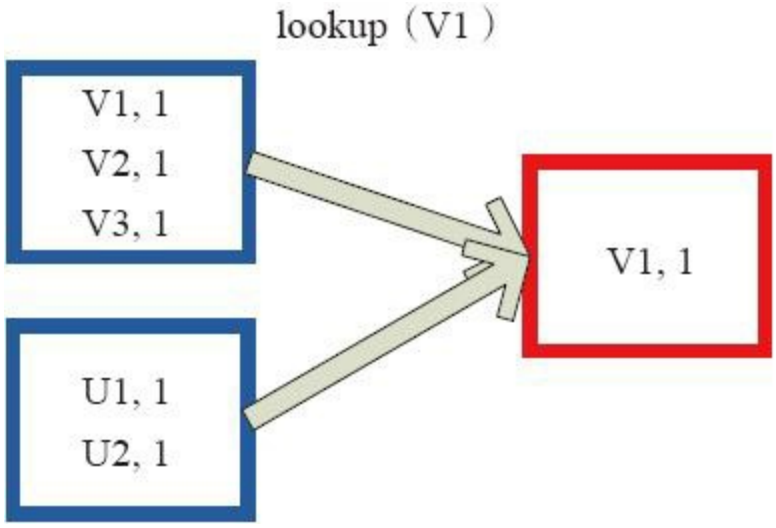

图3-30　lookup对RDD转换

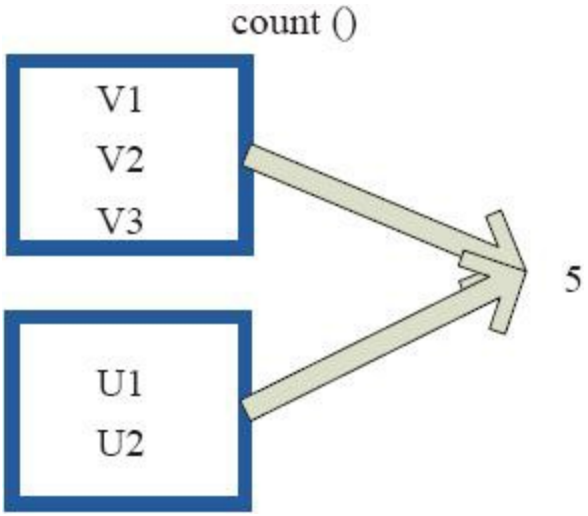

图3-31　count对RDD转换

（6）top

top可返回最大的k个元素。函数定义如下。

```
top ( num : Int ) ( implicit ord : Ordering[T] ) : Array[T]
```

相近函数说明如下。

·top返回最大的k个元素。

·take返回最小的k个元素。

·takeOrdered返回最小的k个元素，并且在返回的数组中保持元素的顺序。

·first相当于top（1）返回整个RDD中的前k个元素，可以定义排序的方式Ordering[T]。返回的是一个含前k个元素的数组。

（7）reduce

reduce函数相当于对RDD中的元素进行reduceLeft函数的操作。函数实现如下。

```
Some(iter.reduceLeft(cleanF))
```

reduceLeft先对两个元素<K，V>进行reduce函数操作，然后将结果和迭代器取出的下一个元素<k，V>进行reduce函数操作，直到迭代器遍历完所有元素，得到最后结果。

在RDD中，先对每个分区中的所有元素<K，V>的集合分别进行reduceLeft。每个分区形成的结果相当于一个元素<K，V>，再对这个结果集合进行reduceleft操作。

例如：用户自定义函数如下。

```
f:(A,B)=>(A._1+"@"+B._1,A._2+B._2)
```

图3-32中的方框代表一个RDD分区，通过用户自定函数f将数据进行reduce运算。示例最后的返回结果为V1@[1]V2U！@U2@U3@U4，12。

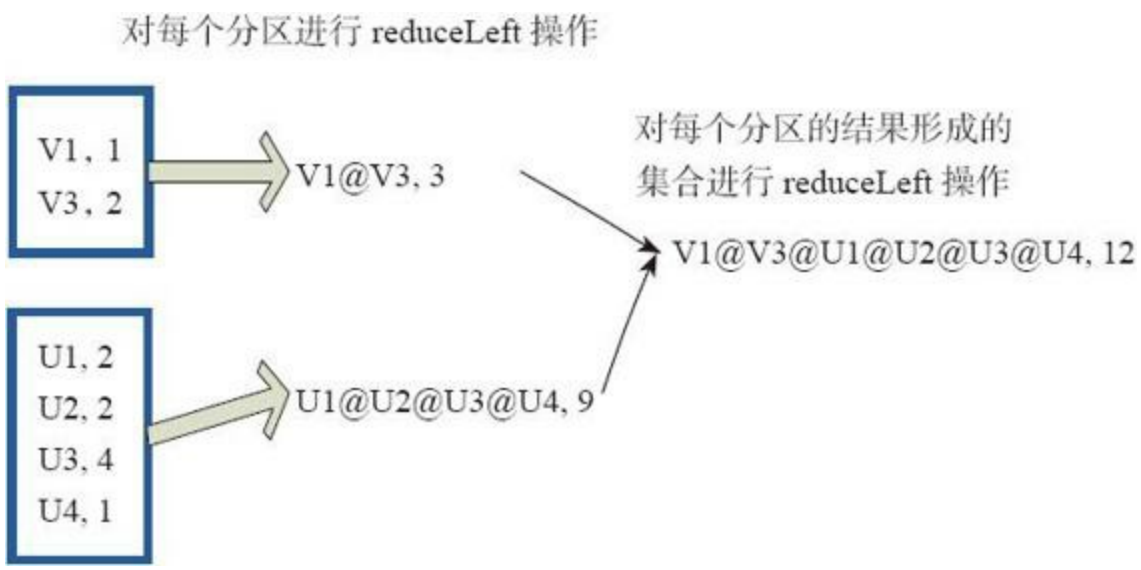

图3-32　reduce算子对RDD转换

（8）fold

fold和reduce的原理相同，但是与reduce不同，相当于每个reduce时，迭代器取的第一个元素是zeroValue。

图3-33中通过下面的用户自定义函数进行fold运算，图中的一个方框代表一个RDD分区。读者可以参照（7）reduce函数理解。

```
fold(("V0@",2))(  (A,B)=>(A._1+"@"+B._1,A._2+B._2))
```

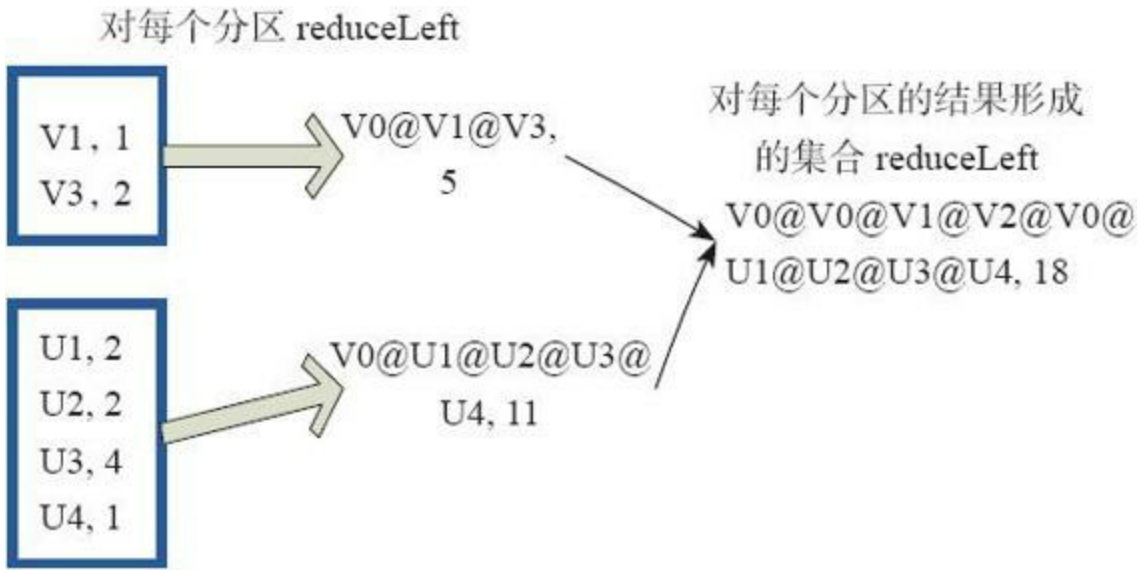

图3-33　fold算子对RDD转换

（9）aggregate

aggregate先对每个分区的所有元素进行aggregate操作，再对分区的结果进行fold操作。

aggreagate与fold和reduce的不同之处在于，aggregate相当于采用归并的方式进行数据聚集，这种聚集是并行化的。而在fold和reduce函数的运算过程中，每个分区中需要进行串行处理，每个分区串行计算完结果，结果再按之前的方式进行聚集，并返回最终聚集结果。

函数的定义如下。

```
aggregate[B](z: B)(seqop: (B,A) => B,combop: (B,B) => B): B
```

图3-34通过用户自定义函数对RDD 进行aggregate的聚集操作，图中的每个方框代表一个RDD分区。

```
rdd.aggregate("V0@",2)((A,B)=>(A._1+"@"+B._1,A._2+B._2)),
(A,B)=>(A._1+"@"+B_1,A._@+B_.2))
```

最后，介绍两个计算模型中的两个特殊变量。

广播（broadcast）变量：其广泛用于广播Map Side Join中的小表，以及广播大变量等场景。这些数据集合在单节点内存能够容纳，不需要像RDD那样在节点之间打散存储。Spark运行时把广播变量数据发到各个节点，并保存下来，后续计算可以复用。相比Hadoop的distributed cache，广播的内容可以跨作业共享。Broadcast的底层实现采用了BT机制。有兴趣的读者可以参考论文[2]。

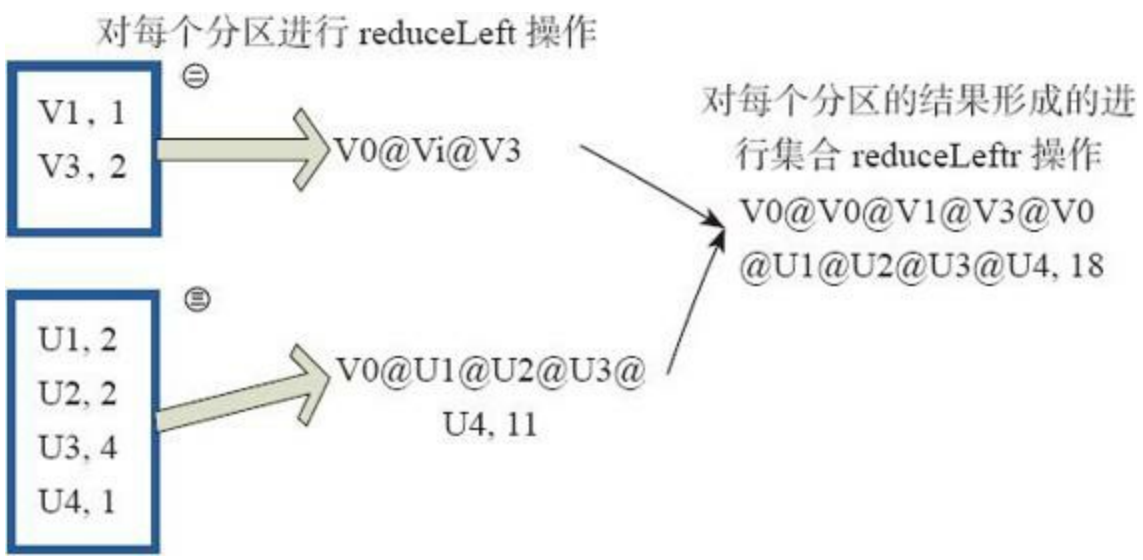

图3-34　aggregate算子对RDD转换

②代表V。

③代表U。

accumulator变量：允许做全局累加操作，如accumulator变量广泛使用在应用中记录当前的运行指标的情景。

[1] @代表数据的分隔符，可替换为其他分隔符。

[2] 参见：Mosharaf Chowdhury，Performance and Scalability of Broadcast in Spark。

3.4 本章小结

本章主要介绍了Spark的计算模型，Spark将应用程序整体翻译为一个有向无环图进行调度和执行。相比MapReduce，Spark提供了更加优化和复杂的执行流。

读者还可以深入了解Spark的运行机制与Spark算子，这样能更加直观地了解API的使用。Spark提供了更加丰富的函数式算子，这样就为Spark上层组件的开发奠定了坚实的基础。

通过阅读本章，读者可以对Spark计算模型进行更为宏观的把握。相信读者还想对Spark内部执行机制进行更深入的了解，下面章节就对Spark的内核进行更深入的剖析。

第4章　Spark工作机制详解

通过前一章的介绍，读者对Spark的计算模型有了全面的把握，本章将深入介绍Spark的内部运行机制。Spark的主要模块包括调度与任务分配、I/O模块、通信控制模块、容错模块以及Shuffle模块。Spark按照应用、作业、Stage和Task几个层次分别进行调度，采用了经典的FIFO和FAIR等调度算法。在Spark的I/O中，将数据以块为单位进行管理，需要处理的块可以存储在本机内存、磁盘或者集群中的其他机器中。集群中的通信对于命令和状态的传递极为重要，Spark通过AKKA框架进行集群消息通信。分布式系统中的容错十分重要，Spark通过Lineage（血统）和Checkpoint机制进行容错性保证。最后介绍Spark中的Shuffle机制，虽然Spark也借鉴了MapReduce模型，但其对Shuffle机制进行了创新与优化。下面开始Spark内部运行机制的探索。

4.1 Spark应用执行机制

下面介绍Spark的应用执行机制。

4.1.1 Spark执行机制总览

Spark应用提交后经历了一系列的转换，最后成为Task在每个节点上执行。Spark应用转换（见图4-1）：RDD的Action算子触发Job的提交，提交到Spark中的Job生成RDD DAG，由DAGScheduler转化为Stage DAG，每个Stage中产生相应的Task集合，TaskScheduler将任务分发到Executor执行。每个任务对应相应的一个数据块，使用用户定义的函数处理数据块。

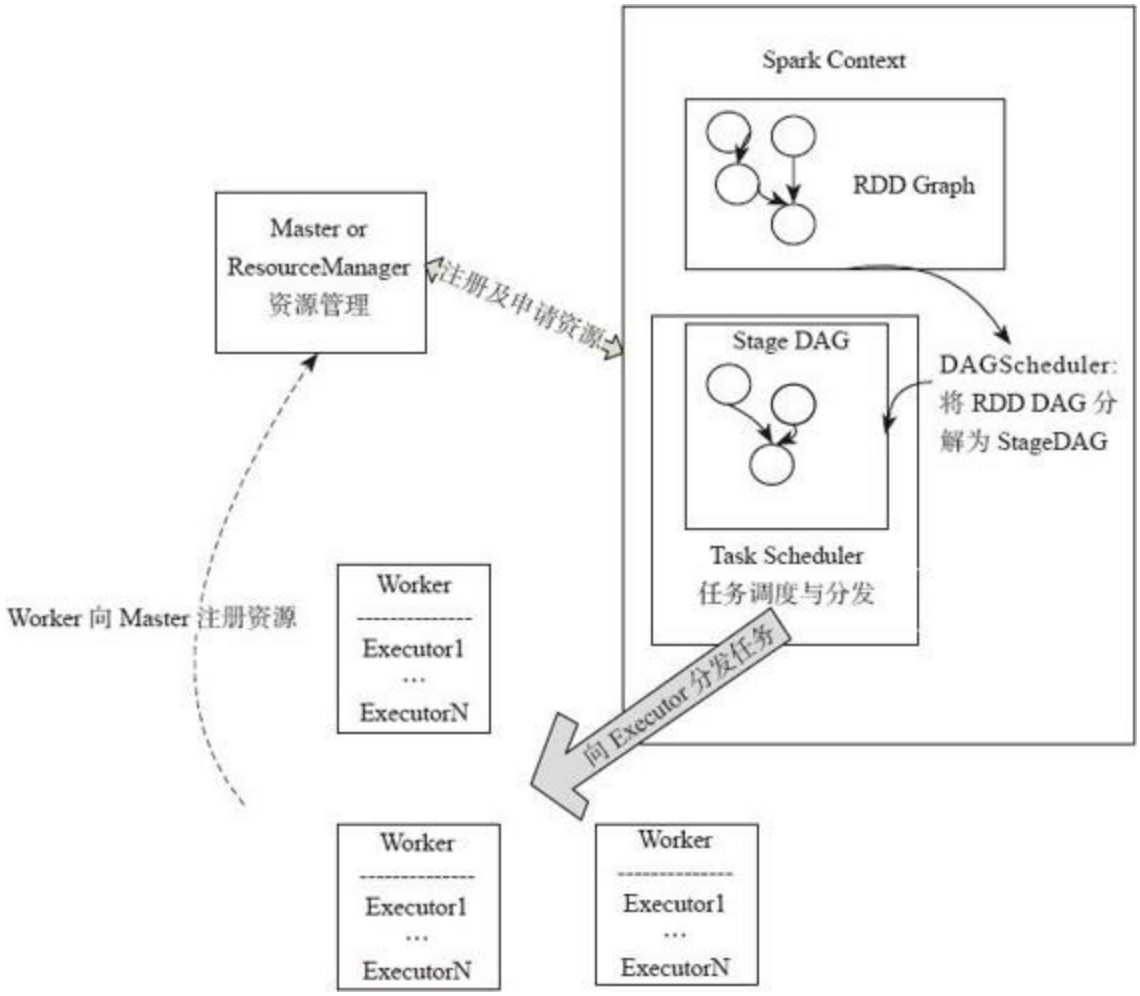

图4-1　Spark应用转换流程

Spark执行的底层实现原理，如图4-2所示。在Spark的底层实现中，通过RDD进行数据的管理，RDD中有一组分布在不同节点的数据块，当Spark的应用在对这个RDD进行操作时，调度器将包含操作的任务分发到指定的机器上执行，在计算节点通过多线程的方式执行任务。一个操作执行完毕，RDD便转换为另一个RDD，这样，用户的操作依次执行。Spark

为了系统的内存不至于快速用完，使用延迟执行的方式执行，即只有操作累计到Action（行动），算子才会触发整个操作序列的执行，中间结果不会单独再重新分配内存，而是在同一个数据块上进行流水线操作。

在集群的程序实现上，有一个重要的分布式数据结构，即弹性分布式数据集（Resilient Distributed Dataset，RDD）。Spark实现了分布式计算和任务处理，并实现了任务的分发、跟踪、执行等工作，最终聚合结果，完成Spark应用的计算。

对RDD的块管理通过BlockManger完成，BlockManager将数据抽象为数据块，在内存或者磁盘进行存储，如果数据不在本节点，则还可以通过远端节点复制到本机进行计算。

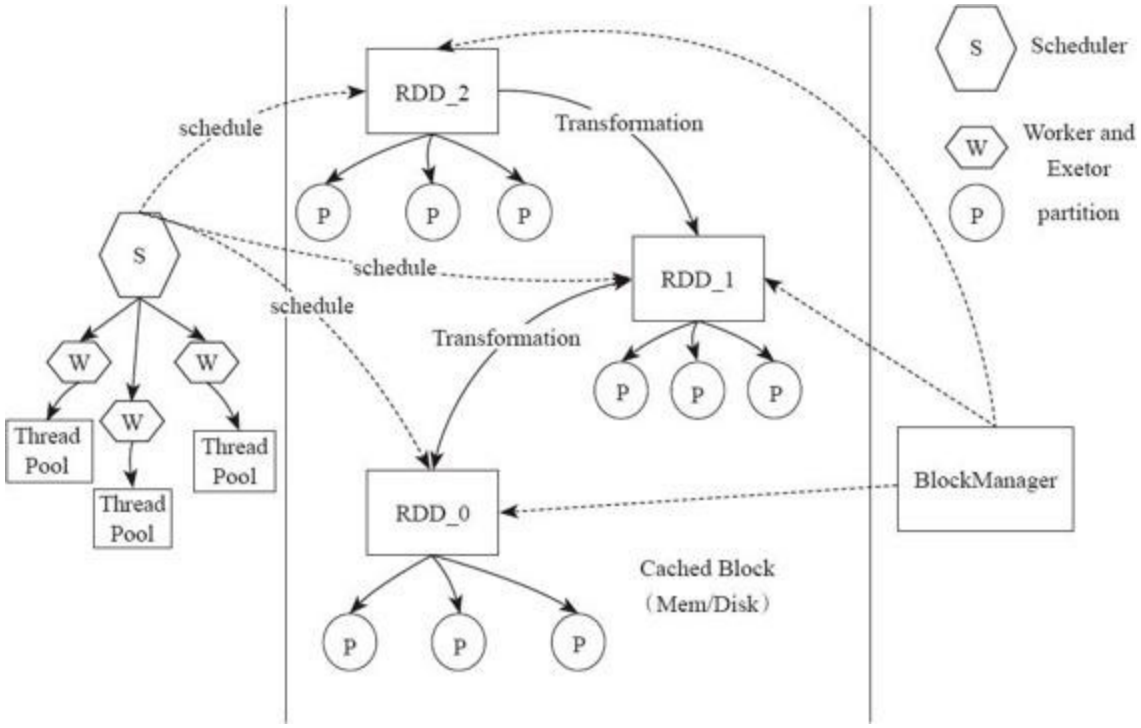

图4-2　Spark执行底层实现

在计算节点的执行器Executor中会创建线程池，这个执行器将需要执行的任务通过线程池并发执行。

4.1.2 Spark应用的概念

Spark应用（Application）是用户提交的应用程序。执行模式有Local、Standalone、YARN、Mesos。根据Spark Application的Driver Program是否在集群中运行，Spark应用的运行方式又可以分为Cluster模式和Client模式。图4-3为Application包含的组件。

应用的基本组件如下。

·Application：用户自定义的Spark程序，用户提交后，Spark为App分配资源，将程序转换并执行。

·Driver Program：运行Application的main（）函数并创建SparkContext。

·RDD Graph：RDD是Spark的核心结构，可以通过一系列算子进行操作（主要有Transformation和Action操作）。当RDD遇到Action算子时，将之前的所有算子形成一个有向无环图（DAG），也就是图中的RDD Graph。再在Spark中转化为Job，提交到集群执行。一个App中可以包含多个Job。

·Job：一个RDD Graph触发的作业，往往由Spark Action算子触发，在SparkContext中通过runJob方法向Spark提交Job。

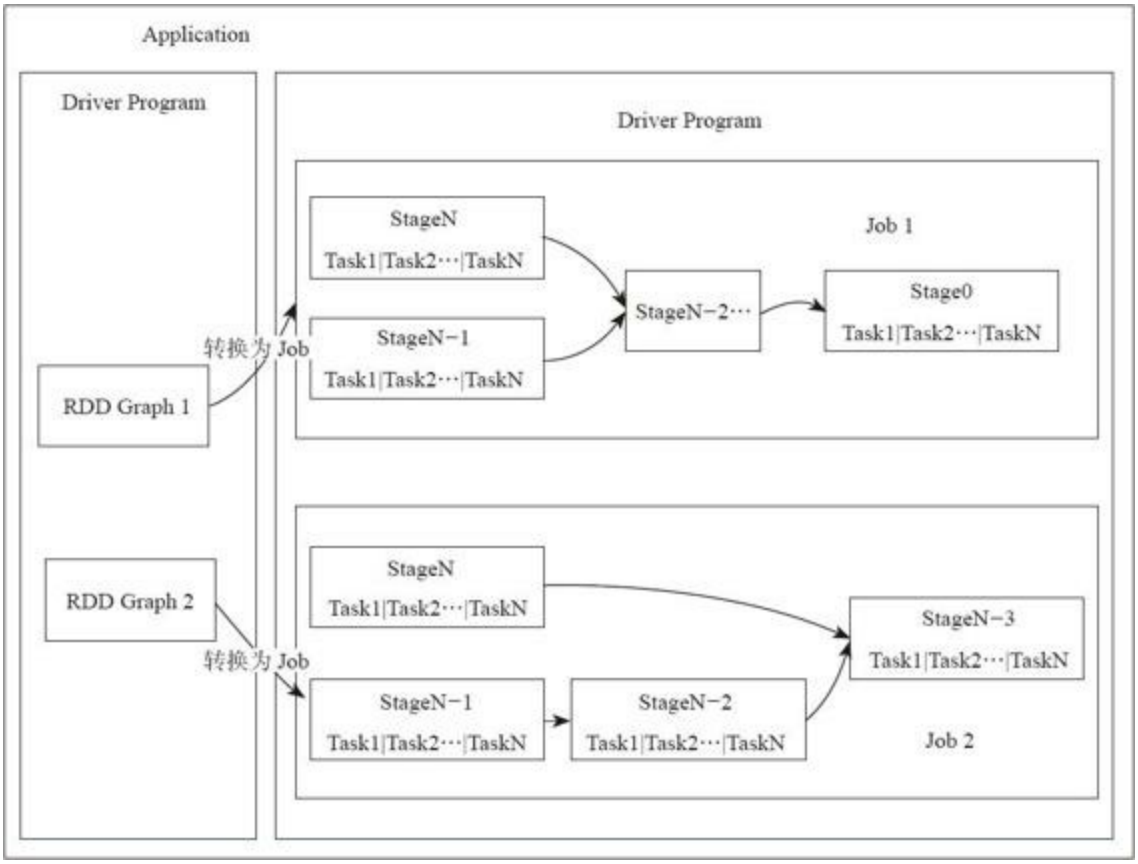

图4-3　Spark Application基本组件

·Stage：每个Job会根据RDD的宽依赖关系被切分很多Stage，每个Stage中包含一组相同的Task，这一组Task也叫TaskSet。

·Task：一个分区对应一个Task，Task执行RDD中对应Stage中包含的算子。Task被封装好后放入Executor的线程池中执行。

4.1.3 应用提交与执行方式

应用的提交包含以下两种方式。

·Driver进程运行在客户端，对应用进行管理监控。

·主节点指定某个Worker节点启动Driver，负责整个应用的监控。

Driver进程是应用的主控进程，负责应用的解析、切分Stage并调度Task到Executor执行，包含DAGScheduler等重要对象。下面具体介绍这两种方式的原理。

1.Driver在客户端运行

例如，执行Spark自带的样例程序：./bin/run-example org.apache.spark.examples.SparkTC spark：//UserHostIP：port。

应用执行和控制如图4-4所示。

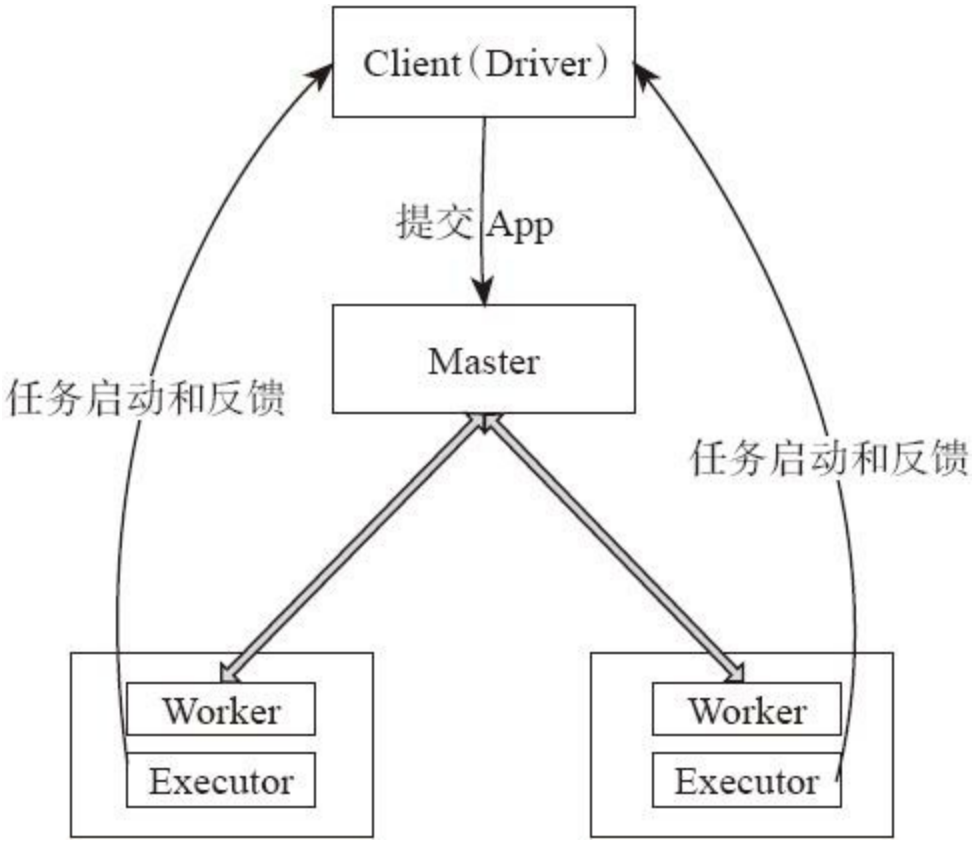

图4-4 Spark Driver位于Client

作业执行流程描述如下。

用户启动客户端，之后客户端运行用户程序，启动Driver进程。在Driver中启动或实例化DAGScheduler等组件。客户端的Driver向Master注册。

Worker向Master注册，Master命令Worker启动Exeuctor。Worker通过创建ExecutorRunner线程，在ExecutorRunner线程内部启动ExecutorBackend进程。

ExecutorBackend启动后，向客户端Driver进程内的SchedulerBackend注册，这样Driver进程就能找到计算资源。Driver的DAGScheduler解析应用中的RDD DAG并生成相应的Stage，每个Stage包含的TaskSet通过TaskScheduler分配给Executor。在Executor内部启动线程池并行化执行Task。

2.Driver在Worker运行

如果Driver在Worker启动执行需要通过org.apache.spark.deploy.Client类执行应用，命令如下。

```
./bin/spark-class org.apache.spark.deploy.Client launch spark：//UserHostIP：port
file：//your_jar org.apache.spark.examples.SparkTC spark：//UserHostIP：port
```

应用提交与执行机制如图4-5所示。

应用执行流程如下。

1）用户启动客户端，客户端提交应用程序给Master。

2）Master调度应用，针对每个应用分发给指定的一个Worker启动Driver，即Scheduler-Backend。Worker接收到Master命令后创建DriverRunner线程，在DriverRunner线程内创建SchedulerBackend进程。Driver充当整个作业的主控进程。Master会指定其他Worker启动Exeuctor，即ExecutorBackend进程，提供计算资源。流程和上面很相似，Worker创建

ExecutorRunner线程，ExecutorRunner会启动ExecutorBackend进程。

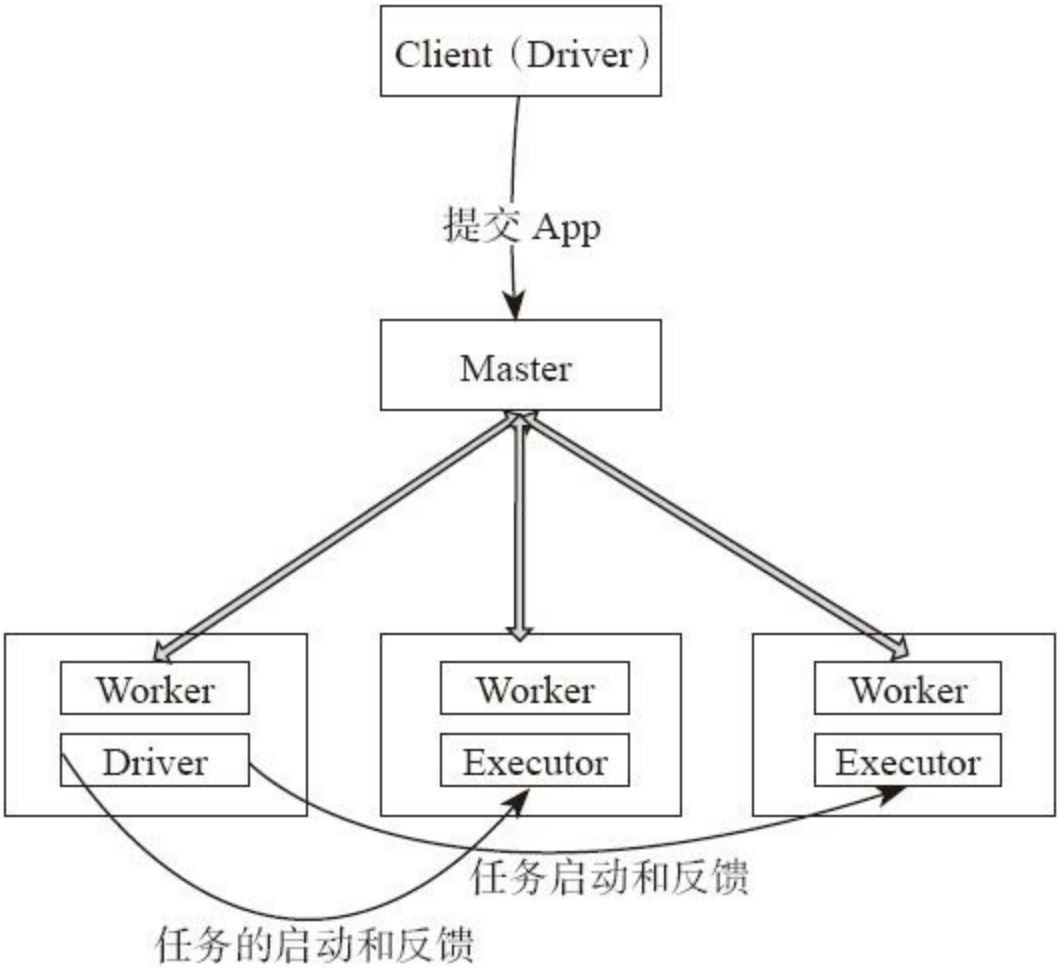

图4-5　Spark Driver位于Worker节点的应用提交与执行机制

3）ExecutorBackend启动后，向Driver的SchedulerBackend注册，这样Driver获取了计算资源就可以调度和将任务分发到计算节点执行。SchedulerBackend进程中包含DAGScheduler，它会根据RDD的DAG切分Stage，生成TaskSet，并调度和分发Task到Executor。对于每个Stage的TaskSet，都会被存放到TaskScheduler中。TaskScheduler将任务分发到Executor，执行多线程并行任务。

4.2 Spark调度与任务分配模块

系统设计很重要的一环便是资源调度。设计者将资源进行不同粒度的抽象建模，然后将资源统一放入调度器，通过一定的算法进行调度，最终达到高吞吐量或者低访问延迟的目的。Spark的调度器设计精良，扩展性极好，为它的后续发展奠定了很好的基础。

Spark有多种运行模式，如Local模式、Standalone模式、YARN模式、Mesos模式。在集群环境下，为了减少复杂性，抓住系统主要脉络进行理解。本节主要介绍Standalone模式中的名词，其他运行模式中各角色实现的功能基本一致，只不过是在特定资源管理器下使用略为不同的名称和调度机制。

在Standalone模式下，集群启动之后，使用jps命令在主节点会看到Master进程，在从节点会看到Worker进程。其中，Master负责接收客户端提交的作业，管理Worker。提供了Web UI呈现集群运行时状态信息，方便用户诊断性能问题。

在Spark的应用提交之后，Spark调度应用。系统设计的一个核心就是调度。从Spark整体上看，调度可以分为4个级别，Application调度、Job调度、Stage的调度、Task的调度与分发。上节已经介绍了这4个概念和概念之间的对应关系。下面对这4个层级调度进行介绍。

4.2.1 Spark应用程序之间的调度

通过前面的介绍，读者了解到每个应用拥有对应的SparkContext.SparkContext维持整个应用的上下文信息，提供一些核心方法，如runJob可以提交Job。然后，通过主节点的分配获得独立的一组Executor JVM进程执行任务。Executor空间内的不同应用之间是不共享的，一个Executor在一个时间段内只能分配给一个应用使用。如果多用户需要共享集群资源，依据集群管理者的配置，用户可以通过不同的配置选项来分配管理资源。

对集群管理者来说简单的配置方式就是静态配置资源分配规则。例如，在不同的运行模式下，用户可以通过配置文件中进行集群调度的配置。配置每个应用可以使用的最大资源总量、调度的优先级等。

1.调度配置

下面根据不同集群的运行模式配置调度[1]。

（1）Standalone

默认情况下，用户向以Standalone模式运行的Spark集群提交的应用使用FIFO（先进先出）的顺序进行调度。每个应用会独占所有可用节点的资源。用户可以通过配置参数spark.cores.max决定一个应用可以在整个集群申请的CPU core数。注意，这个参数不是控制单节点可用多少核。如果用户没有配置这个参数，则在Standalone模式下，默认每个应用可以分配由参数spark.deploy.defaultCores决定的可用核数。

（2）Mesos

如果用户在Mesos上使用Spark，并且想要静态地配置资源的分配策略，则可以通过配置参数spark.mesos.coarse为true，将Mesos配置为粗粒度调度模式。然后配置参数spark.cores.max来限制应用可以使用的CPU core的最大限额。同时用户应该对参数

spark.executor.memory进行配置，进而限制每个Executor的内存使用量。Mesos中还可以配置动态共享CPU core的执行模式，用户只需要使用mesos://URL而不配置spark.mesos.coarse参数为true，就能以这种方式执行，使Mesos运行在细粒度调度模型下。在这种模式下，每个Spark应用程序还是会拥有独立和固定的内存分配，但是当应用占用的一些机器上不再运行任务，机器处于空闲状态时，其他机器可以使用这些机器上空闲的CPU core来执行任务，相当于复用空闲的CPU提升了资源利用率。这种模式在集群上再运行大量不活跃的应用情景下十分有用，如大量不同用户发起请求的场景。

（3）YARN

当Spark运行在YARN平台上时，用户可以在YARN的客户端通过配置--num-executors选项控制为这个应用分配多少个Executor，然后通过配置--executor-memory及--executor-cores来控制应用被分到的每个Executor的内存大小和Executor所占用的CPU核数。这样便可以限制用户提交的应用不会过多的占用资源，让不同用户能够共享整个集群资源，提升YARN吞吐量。

注意：以上3种运行模式都不提供跨应用的共享内存。如果用户想共享内存数据，Spark官网推荐用户开发一个单机服务，这个服务可以接收多个对同一个RDD的查询请求，并返回结果，类似的Shark JDBC Server就是这样工作的，Spark SQL在新版本中也会提供这样的功能。目前版本，Spark SQL暂时使用的Shark Server2 Github地址为https://github.com/amplab/shark/tree/sparkSql。

Shark的Github地址为：https://github.com/amplab/shark。

2.FIFO的调度代码

最后，读者可以参考下面源码，了解在Standalone模式中，集群是如何完成应用FIFO的调度的。Spark的应用接收提交和调度的代码在Master.scala文件中，在schedule（）方法中实现调度。

```
private def schedule() {
……
  for (worker <- shuffledWorkers if worker.state == WorkerState.ALIVE) {
    for (driver <- List(waitingDrivers: _*)) {
      if (worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.desc.cores) {
        launchDriver(worker, driver)
        waitingDrivers -= driver
      }
    }
  }
```

从源码中可以看到，Master先统计可用资源，然后在waitingDrivers的队列中通过FIFO方式为App分配资源和指定Worker启动Driver执行应用。

[1] 参见：http://spark.apache.org/docs/latest/job-scheduling.html#scheduling-across-applications。

4.2.2 Spark应用程序内Job的调度

在Spark应用程序内部，用户通过不同线程提交的Job可以并行运行，这里所说的Job就是Spark Action（如count、collect等）算子触发的整个RDD DAG为一个Job，在实现上，算子中的本质是调用SparkContext中的runJob提交了Job。例如，通过count的源码看这个过程。

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

其中，sc就是SparkContext对象，调用runJob方法提交Job。

Spark的调度器是完全线程安全的，并且支持一个应用处理多请求的用例（如多用户进行查询）。

（1）FIFO模式

在默认情况下，Spark的调度器以FIFO（先进先出）方式调度Job的执行，如图4-6所示。每个Job被切分为多个Stage。第一个Job优先获取所有可用的资源，接下来第二个Job再获取剩余资源。以此类推，如果第一个Job并没有占用所有的资源，则第二个Job还可以继续获取剩余资源，这样多个Job可以并行运行。如果第一个Job很大，占用所有资源，则第二个Job就需要等待第一个任务执行完，释放空余资源，再申请和分配Job。

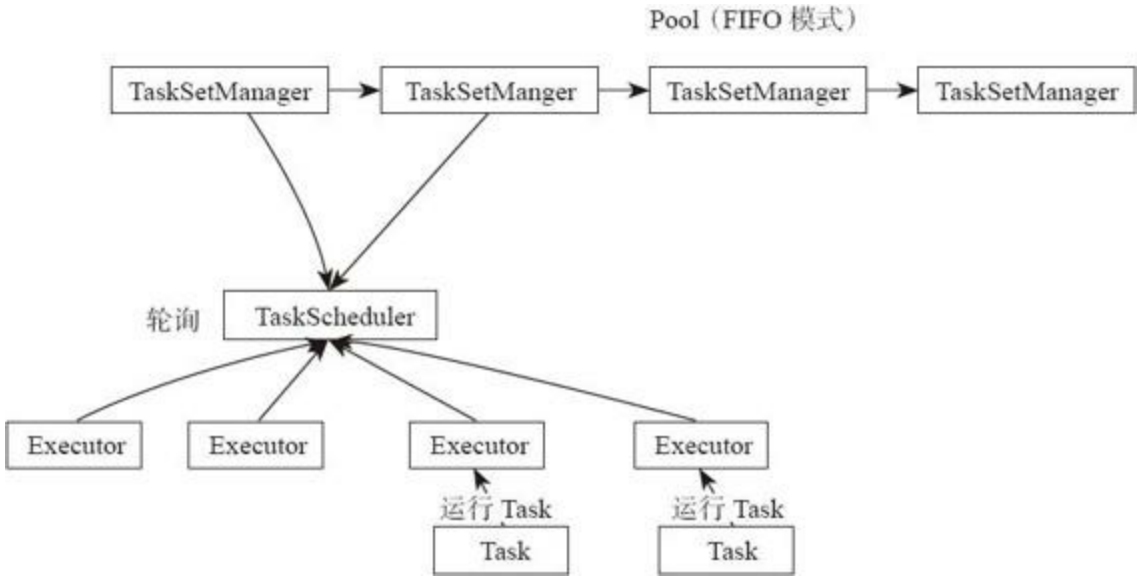

图4-6　FIFO模式调度示意图

读者可以通过图示大致了解FIFO模式，下面通过源码更加深入地剖析FIFO模式。

```
private[spark] class FIFOSchedulingAlgorithm extends
 SchedulingAlgorithm {
  override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
/*执行优先级*/
      val priority1 = s1.priority
      val priority2 = s2.priority
      var res = math.signum(priority1 - priority2)
if (res == 0) {
        val stageId1 = s1.stageId
        val stageId2 = s2.stageId
/*signum是符号函数。功能是如果参数为 0，则返回 0；如果参数大于 0，则返回 1.0；如果参数小于 0，则返回-1.0*/
      res = math.signum(stageId1 - stageId2)
    }
    if (res < 0) {
       true
    } else {
       false
    }
  }
}
```

在算法执行中，先看优先级，TaskSet的优先级是JobID，因为先提交的JobID小，所以就会被更优先地调度，这里相当于进行了两层排序，先看是否是同一个Job的Taskset，不同Job之间的TaskSet先排序。

最后执行的stageId最小为0，最先应该执行的stageId最大。但是这里的调度机制是优先调度Stageid小的。在DAGScheduler中控制Stage是否被提交到队列中，如果还有父母Stage未执行完，则该stage的Taskset不会提交到调度池中，这就保证了虽然最先做的stage的id大，但是排序完，由于后面的还没提交到调度池中，所以会先执行。由此可见，stage

的TaskSet调度逻辑主要在DAGScheduler中，而Job调度由FIFO或者FAIR算法调度。

Job调度的FIFO或FAIR方式是通过Pool类实现的。在下面代码为Pool类的实现、代码通过taskSetSchedulingAlgorithm选择使用FIFO还是FAIR进行Job调度。

```
private[spark] class Pool(
……
  var taskSetSchedulingAlgorithm: SchedulingAlgorithm = {
    schedulingMode match {
      case SchedulingMode.FAIR =>
        new FairSchedulingAlgorithm()
      case SchedulingMode.FIFO =>
        new FIFOSchedulingAlgorithm()
    }
  }
……
```

/*这里是使用调度算法的地方，实际上通过调度算法进行了Job的调度和Job内的TaskSetManager的两级调度。获取优先级最高的可调度资源执行*/

```
    override def getSortedTaskSetQueue: ArrayBuffer[TaskSetManager] = {
         var sortedTaskSetQueue = new ArrayBuffer[TaskSetManager]
    val sortedSchedulableQueue =
    /*使用比较器进行排序*/
schedulableQueue.toSeq.sortWith(taskSetSchedulingAlgorithm.comparator)
         for (schedulable <- sortedSchedulableQueue) {
       sortedTaskSetQueue ++= schedulable.getSortedTaskSetQueue
     }
     sortedTaskSetQueue
  }
```

这里是使用调度算法的地方，实际上通过调度算法进行了Job的调度和Job内的TaskSetManager的两级调度。获取优先级最高的可调度资源执行。这里使用了设计模式中的策略模式，使用FIFO充当TaskSetManager的比较器。

（2）FAIR模式

从Spark 0.8版本开始，可以通过配置FAIR共享调度模式调度Job，如图4-7所示。在FAIR共享模式调度下，Spark在多Job之间以轮询（round robin）方式为任务分配资源，所有的任务拥有大致相当的优先级来共享集群的资源。这就意味着当一个长任务正在执行时，短任务仍可以分配到资源，提交并执行，并且获得不错的响应时间。这样就不用像以前一样需

要等待长任务执行完才可以。这种调度模式很适合多用户的场景。用户可以通过配置spark.scheduler.mode方式来让应用以FAIR模式调度。FAIR调度器同样支持将Job分组加入调度池中调度，用户可以同时针对不同优先级对每个调度池配置不同的调度权重。这种方式允许更重要的Job配置在高优先级池中优先调度。这种方式借鉴了Hadoop的FAIR调度模型，如图4-7所示。

如果读者对FAIR调度模式的源码感兴趣，可以参照FairSchedulingAlgorithm.scala源码了解，限于篇幅先不在这里介绍。

在默认情况下，每个调度池拥有相同的优先级来共享整个集群的资源，同样default pool中的每个Job也拥有同样优先级进行资源共享，但是在用户创建的每个资源池中，Job是通过FIFO方式进行调度的。例如，如果每个用户都创建了一个调度池，这就意味着每个用户的调度池将会获得同样的优先级来共享整个集群，但是每个用户的调度池内部的请求是按照先进先出的方式调度的，后到的请求不能比先到的请求更早获得资源。

在没有外部干预的情况下，新提交的任务放入default pool中进行调度。如果用户也可以自定义调度池，通过在SparkContext中配置参数spark.scheduler.pool创建调度池。

```
/*假设sc是 SparkContext变量*/
sc.setLocalProperty("spark.scheduler.pool", "pool6")
```

这样配置了这个参数的线程每次提交的任务都是放入这个池中进行调度（如这个线程调用RDD.collect或者RDD.count等action算子）。这种调度池的配置可以很方便地让同一个用户在一个线程中运行多个Job。如果用户不想再使用这个调度池，可以通过调用SparkContext的方法来终止这个调度池的使用：

```
sc.setLocalProperty("spark.scheduler.pool6", null)
```

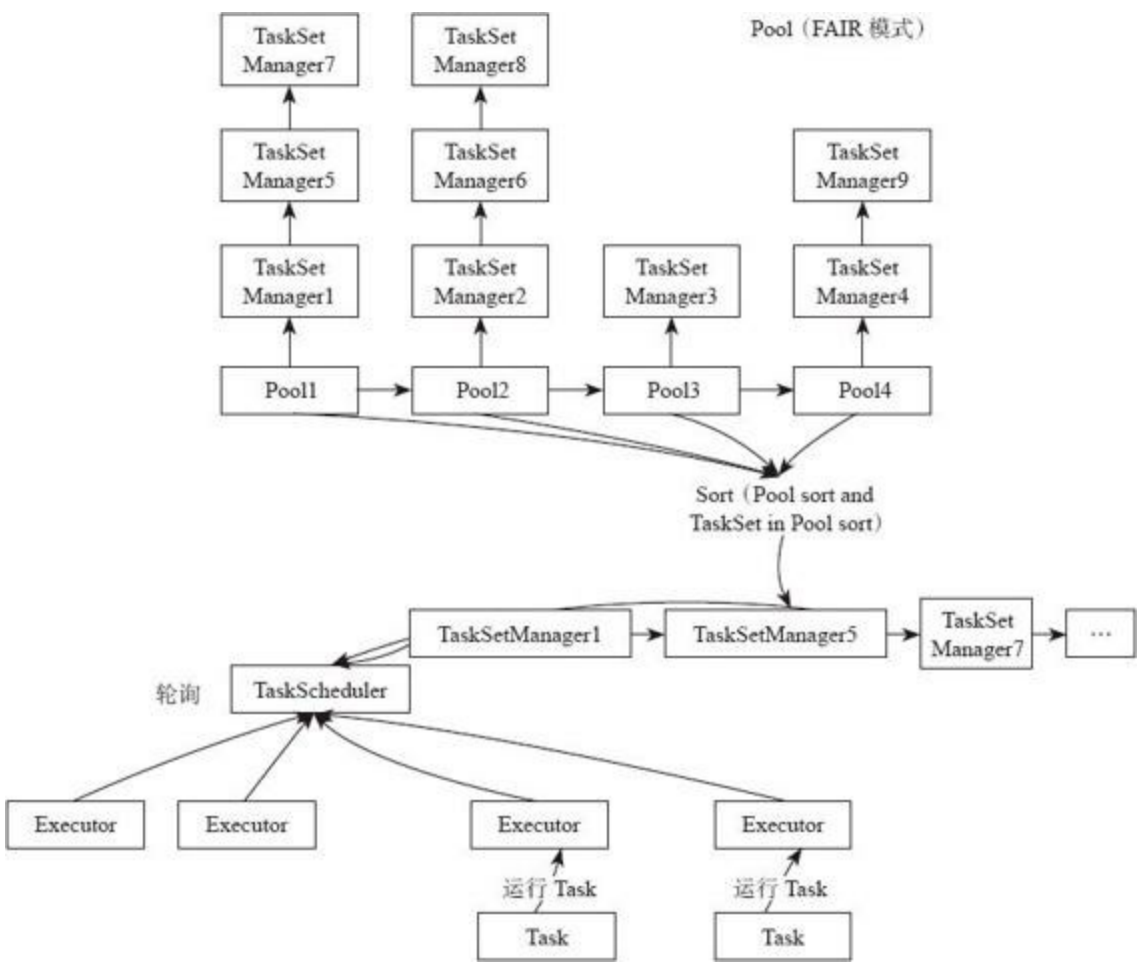

图4-7　FAIR调度模型

（3）配置调度池

用户可以通过配置文件自定义调度池的属性。每个调度池支持下面3个配置参数。

1）调度模式（schedulingMode）：用户可以选择FIFO或者FAIR方式进行调度。

2）权重（Weight）：这个参数控制在整个集群资源的分配上，这个调度池相对其他调度池优先级的高低。例如，如果用户配置一个指定的调度池权重为3，那么这个调度池将会获得相对于权重为1的调度池3倍的资源。

3）minShare：配置minShare参数（这个参数代表多少个CPU核），这个参数决定整体调度的调度池能给待调度的调度池分配多少资源就可以满足调度池的资源需求，剩余的资源还可以继续分配给其他调度池。

用户可以通过conf/fairscheduler.xml文件配置调度池的属性，同时需要在程序的

SparkConf对象中配置属性。

```
conf.set("spark.scheduler.allocation.file", "/path/to/file")
```

读者可以参考下面官方文档的配置例子进行配置，配置文件的格式为XML。

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

读者可以参考conf/fairscheduler.xml.template这个模板文件，文件中提供了更加全面的配置介绍。

4.2.3 Stage和TaskSetManager调度方式

下面介绍Stage和TaskSetManager的调度方式。

1.Stage的生成

Stage的调度是由DAGScheduler完成的。由RDD的有向无环图DAG切分出了Stage的有向无环图DAG。Stage的DAG通过最后执行的Stage为根进行广度优先遍历，遍历到最开始执行的Stage执行，如果提交的Stage仍有未完成的父母Stage，则Stage需要等待其父Stage执行完才能执行。同时DAGScheduler中还维持了几个重要的Key-Value集合结构，用来记录Stage的状态，这样能够避免过早执行和重复提交Stage。waitingStages中记录仍有未执行的父母Stage，防止过早执行。runningStages中保存正在执行的Stage，防止重复执行。failedStages中保存执行失败的Stage，需要重新执行，这里的设计是出于容错的考虑。

读者可以通过下面DAGScheduler中的源码进一步理解这个广度优先遍历过程。

```
/*waitingStages存储等待执行的Stage集合*/
   private[scheduler] val waitingStages = new HashSet[Stage]
/*runningStages存储正在执行的stage集合*/
    private[scheduler] val runningStages = new HashSet[Stage]*/
/*failedStages存储执行失败的Stage集合*/
    private[scheduler] val failedStages = new HashSet[Stage]*/
    private def submitStage(stage: Stage) {
    val jobId = activeJobForStage(stage)
    if (jobId.isDefined) {
        logDebug("submitStage(" + stage + ")")
        if(!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {
           val missing = getMissingParentStages(stage).sortBy(_.id)
           logDebug("missing: " + missing)
           if (missing == Nil) {
               logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing parents")
          /*遇到可以执行的Stage，提交Stage中的任务*/
          submitMissingTasks(stage, jobId.get)
          runningStages += stage
        } else {
          /*继续遍历需要执行的Stage*/
          for (parent <- missing) {
            submitStage(parent)
          }
          waitingStages += stage
        }
      }
    } else {
      abortStage(stage, "No active job for stage " + stage.id)
    }
  }
/*在DAGScheduler中通过pendingTasks存储每个Stage等待执行的Task集合*/
  private[scheduler] val pendingTasks = new HashMap[Stage, HashSet[Task[_]]]
/*在submitMissingTasks方法中提交任务*/
```

```
  private def submitMissingTasks(stage: Stage, jobId: Int) {
    logDebug("submitMissingTasks(" + stage + ")")
    /* 获取等待执行的任务，并在pendingTask结构中记录它们*/
    val myPending = pendingTasks.getOrElseUpdate(stage, new HashSet)
    myPending.clear()
    var tasks = ArrayBuffer[Task[_]]()
    if (stage.isShuffleMap) {
       for (p <- 0 until stage.numPartitions if stage.outputLocs(p) == Nil) {
         val locs = getPreferredLocs(stage.rdd, p)
         tasks += new ShuffleMapTask(stage.id, stage.rdd, stage.shuffleDep.get, p, locs)
       }
    } else {
      /* 这是一个最后阶段，计算出作业缺失的分区*/
      val job = resultStageToJob(stage)
      for (id <- 0 until job.numPartitions if !job.finished(id)) {
        val partition = job.partitions(id)
        val locs = getPreferredLocs(stage.rdd, partition)
        tasks += new ResultTask(stage.id, stage.rdd, job.func, partition, locs, id)
      }
     }
……
/*TaskScheudler提交任务*/
  taskScheduler.submitTasks(
new TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.jobId, properties))
……
}
```

在TaskScheduler中将每个Stage中对应的任务进行提交分和调度。注意：一个应用对应一个TaskScheduler，也就是这个应用中所有Action触发的Job中的TaskSetManager都是由这个TaskScheduler调度的。

2.TaskSetManager的调度

结合上面介绍的Job的调度和Stage的调度方式，可以知道，每个Stage对应的一个TaskSetManager通过Stage回溯到最源头缺失的Stage提交到调度池pool中，在调度池中，这些TaskSetMananger又会根据Job ID排序，先提交的Job的TaskSetManager优先调度，然后一个Job内的TaskSetManager ID小的先调度，并且如果有未执行完的父母Stage的TaskSetManager，则是不会提交到调度池中。

4.2.4 Task调度

通过分析下面的源码，读者可以了解Spark的Task的调度方式。

1.提交任务

在DAGScheduler中提交任务时，分配任务执行节点。

```
private def submitMissingTasks(stage: Stage, jobId: Int) {
   logDebug("submitMissingTasks(" + stage + ")")
   val myPending = pendingTasks.getOrElseUpdate(stage, new HashSet)
   myPending.clear()
var tasks = ArrayBuffer[Task[_]]()
/*判断是否为Shuffle map stage，如果是，则这个stage输出的结果会经过Shuffle阶段作为下一个stge的输入，如果是Result Stage，则stage的结果输出到Spark空间（如count()，save()）*/
   if (stage.isShuffleMap) {
      for (p <- 0 until stage.numPartitions if stage.outputLocs(p) == Nil) {
         val locs = getPreferredLocs(stage.rdd, p)
/*初始化ShuffleMapTask*/
        tasks += new ShuffleMapTask(stage.id, stage.rdd, stage.shuffleDep.get, p, locs)
      }
     } else {
      val job = resultStageToJob(stage)
      for (id <- 0 until job.numPartitions if !job.finished(id)) {
        val partition = job.partitions(id)
        val locs = getPreferredLocs(stage.rdd, partition)
/*初始化ResultTask*/
        tasks += new ResultTask(stage.id, stage.rdd, job.func, partition, locs, id)
      }
   }
   val properties = if (jobIdToActiveJob.contains(jobId)) {
     jobIdToActiveJob(stage.jobId).properties
   } else {
     // this stage will be assigned to "default" pool
     null
   }
/*通过此方法获取任务最佳的执行节点*/
private[spark]
  def getPreferredLocs(rdd: RDD[_], partition: Int): Seq[TaskLocation] = synchronized {
……
}
```

2.分配任务执行节点

1）如果是调用过cache()方法的RDD，数据已经缓存在内存，则读取内存缓存中分区的数据。

```
val cached = getCacheLocs(rdd)(partition)
if (!cached.isEmpty) {
    return cached
}
```

2）如果直接能获取到执行地点，则返回执行地点作为任务的执行地点，通常DAG中最源头的RDD或者每个Stage中最开始的RDD会有执行地点的信息。例如，HadoopRDD从HDFS读出的分区就是最好的执行地点。这里涉及Hadoop分区的数据本地性问题，感兴趣的读者可以查阅Hadoop的资料了解。

```
/*@deprecated("Replaced by PartitionwiseSampledRDD", "1.0.0")
private[spark] class SampledRDD[T: ClassTag](
  override def getPreferredLocations(split: Partition): Seq[String] =
    firstParent[T].preferredLocations(split.asInstanceOf[SampledRDDPartit//ion].prev)*/
    val rddPrefs = rdd.preferredLocations(rdd.partitions(partition)).toList
    if (!rddPrefs.isEmpty) {
       return rddPrefs.map(host => TaskLocation(host))
}
```

3）如果不是上面两种情况，将遍历RDD获取第一个窄依赖的父亲RDD对应分区的执行地点。

```
rdd.dependencies.foreach {
  case n: NarrowDependency[_] =>
    for (inPart <- n.getParents(partition)) {
```

获取到子RDD分区的父母分区的集合，再继续深度优先遍历，不断获取到这个分区的父母分区的第一个分区，直到没有Narrow Dependency。可以通过图4-8看到RDD2的p0分区位置就是RDD0中p0分区的位置。

```
    val locs = getPreferredLocs(n.rdd, inPart)
    if (locs != Nil) {
       return locs
    }
}
    case _ =>
  }
  Nil
}
```

如果是Shuffle Dependency，由于在Stage之间需要进行Shuffle，而分区无法确定，所以无法获取分区的存储位置。这表示如果一个Stage的父母Stage还没执行完，则子Stage中的Task不能够获得执行位置。

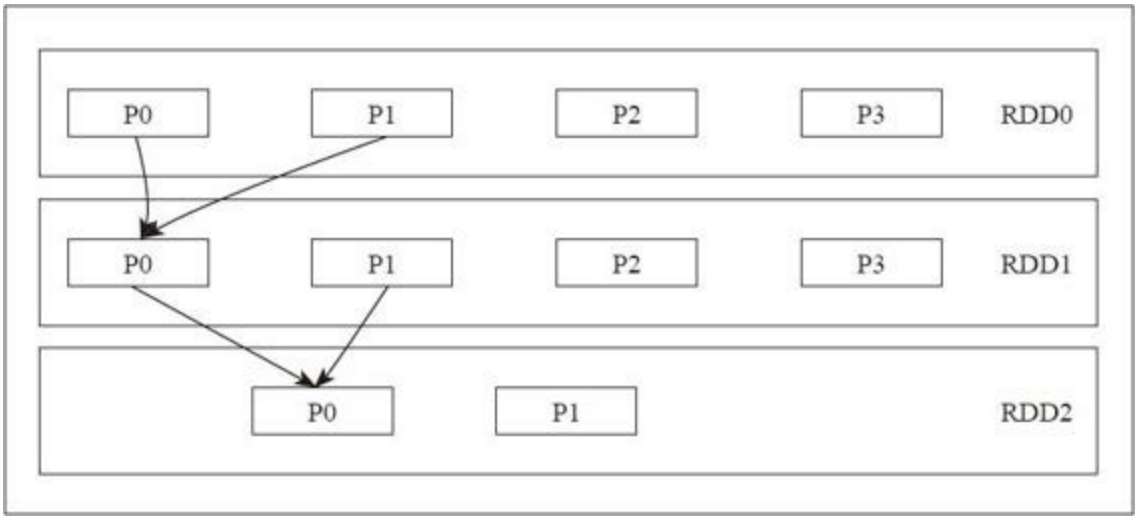

图4-8　分区获取prefer位置

整体的Task分发由TaskSchedulerImpl来实现，但是Task的调度（本质上是Task在哪个分区执行）逻辑由TaskSetManager完成。这个类监控整个任务的生命周期，当任务失败时（如执行时间超过一定的阈值），重新调度，也会通过delay scheduling进行基于位置感知（locality-aware）的任务调度。TaskSchedulerImpl类有几个主要接口：接口resourceOffer，作用为判断任务集合是否需要在一个节点上运行。接口statusUpdate，其主要作用为更新任务状态。

任务的locality由以下两种方式确定。

1）RDD DAG源头有HDFS等类型的分布式存储，它们内置的数据本地性决定（RDD中配置preferred location确定）数据存储位置和分区的选取。

2）每个其他非源头Stage由于都要进行Shuffle，所以地址以在resourceoffer中进行round robin来确定，初始提交Stage时，将prefer的位置设置为Nil。但在Stage调度过程中，内部是通过Narrow dep的祖先Stage确定最佳执行位置的。这样相当于每个RDD的分区都有prefer执行位置。

4.3 Spark I/O机制

Spark的I/O由传统的I/O演化而来，但又有所不同。

·单机计算机系统中，数据集中化，结构化数据、半结构化数据、非结构化数据都只存储在一个主机中，而Spark中的数据分区是分散在多个计算机系统中的。

·传统计算机数据量小。Spark需要处理TB、PB级别的数据。

这样会产生一些问题，Spark进行I/O不仅要考虑本地主机的I/O开销，还要考虑数据在不同主机之间的传输开销。同时Spark对数据的寻址方式也要改变，以应对大数据的挑战。

4.3.1 序列化

序列化是将对象转换为字节流，本质上可以理解为将链表存储的非连续空间的数据存储转化为连续空间存储的数组中。这样就可以将数据进行流式传输或者块存储。相反，反序列化就是将字节流转化为对象。

序列化主要有以下两个目的。

·进程间通信：不同节点之间进行数据传输。

·数据持久化存储到磁盘：本地节点将对象写入磁盘。

Spark通过集中方式实现进程通信，包括Actor的消息模式、Java NIO和netty的OIO。

在Spark中，序列化拥有重要地位。无论是内存或者磁盘中的RDD含有的对象存储，还是节点间的传输数据，都需要执行序列化的过程。序列化与反序列化的速度、序列化后的数据大小等都影响数据传输的速度，以致影响集群的计算效率。Spark可以使用Java的序列化库，也可以使用Kyro序列化库。Kyro具有紧凑、快速、轻量的优点，允许自定义序列化方法，且扩展性很好。下面用户可以通过表4-1参数进行序列化配置。

表4-1 序列化参数简介

序列化方式	参数	说明
spark.closure.serializer	org.apache.spark.serializer.JavaSerializer	用于闭包的序列化器

（续）

序列化方式	参数	说明
spark.serializer.objectStreamReset	10000	使用 JavaSerializer 序列化器，序列化器会缓冲对象，以防止写入冗余数据，通过 Reset 参数设定垃圾回收这些缓存对象的阈值。如果不使用缓存对象，则将值设定为 <0。默认设定为 10000，表示允许达到 10000 对象再进行回收
spark.kryo.referenceTracking	true	当使用 kyro 序列化器时，追踪对相同对象的引用，这样能够对引用多次的对象只存储一份，减少空间占用
spark.kryoserializer.buffer.mb	2	Kyro 序列化器中的缓冲区大小，其大小应该大于允许创建的对象的最大空间占用

其他详细介绍会在性能调优章节介绍。

4.3.2 压缩

当大片连续区域进行数据存储并且存储区域中数据重复性高的状况下，数据适合进行压缩。数组或者对象序列化后的数据块可以考虑压缩。所以序列化后的数据可以压缩，使数据紧缩，减少空间开销。

1.Spark对压缩方式的选择

压缩采用了两种算法：Snappy和LZF，底层分别采用了两个第三方库实现，同时可以自定义其他压缩库对Spark进行扩展。Snappy提供了更高的压缩速度，LZF提供了更高的压缩比，用户可以根据具体需求选择压缩方式。

压缩格式及解编码器如下。

·LZF：org.apache.spark.io.LZFCompressionCodec。

·Snappy：org.apache.spark.io.SnappyCompressionCodec。

压缩算法的对比，如图4-9所示。

（1）Ning-Compress

Ning-compress是一个对数据进行LZF格式压缩和解压缩的库，这个库是Tatu Saloranta（tatu.saloranta@iki.fi）书写的。用户可以在Github地址：https://github.com/ning/compress下载，进行学习和研究。

（2）snappy-java

Snappy算法的前身是Zippy，被Google用于MapReduce、BigTable等许多内部项目。snappy-java由谷歌开发，是以C++开发的Snappy压缩解压缩库的Java分支。Github地址为：https://github.com/xerial/snappy-java。

Snappy的目标是在合理的压缩量情况下，提供高压缩速度的库。因此Snappy的压缩比和LZF差不多，并不是很高。根据数据集的不同，压缩比能达到20%~100%。有兴趣的读者可以看一个压缩算法Benchmark，它对基于JVM运行语言的压缩库进行对比。这个Benchmark对snappy-java和其他压缩工具LZO-java/LZF/QuickLZ/Gzip/Bzip2进行了比较。地址为Github：https://github.com/ning/jvm-compressor-benchmark/wiki。这个Benchmark是由Tatu Saloranta@cotowncoder开发的。

Snappy通常在达到相当压缩的情况下，要比同类的LZO、LZF、FastLZ和QuickLZ等快速的压缩算法快。它对纯文本的压缩比大概是1.5~1.7x，对HTML网页是2~4x，对图片等二进制数据基本没有压缩，为1x。

Snappy分别对64位和32位处理器进行了优化，不论是32位处理，还是64位处理器，都能达到很高的效率。据官方介绍，Snappy经过PB级别的大数据的考验，稳定性方面没有问题，Google的map reduce、rpc等很多框架都用到了Snappy压缩算法。

压缩是在时间和空间上的一种权衡。更长的压缩和解压缩时间会节省更多的空间。而空间占用少意味着可以缓存更多的数据，节省I/O时间和网络传输时间。不同的压缩算法是在不同情境的一种权衡，而且对不同数据类型文件进行压缩又会产生差异。可以参考图4-9，对不同算法的使用进行权衡。

Library	Level	Compressed size	Compression Mbyte/s	Decompression Mbyte/s
QuickLZ C 1.5.0	1	47.9%	308	358
QuickLZ C 1.5.0	2	42.3%	131	309
QuickLZ C 1.5.0	3	40.0%	31	516
QuickLZ C# 1.5.0	1	47.9%	133	132
QuickLZ Java 1.5.0	1	47.9%	127	95
LZF 3.1	UF	54.9%	204	396
LZF 3.1	VF	51.9%	193	384
FastLZ 0.1.0	1	53.0%	173	442
FastLZ 0.1.0	2	50.7%	167	406
LZO 1X 2.02	1	48.3%	169	434
zlib 1.22	1	37.6%	55	234
Core i7 920 benchmark. Details				

图4-9　压缩方式对比

2.在Spark程序中使用压缩

用户可以通过下面两种方式配置压缩。

（1）在Spark-env.sh文件中配置

用户可以在启动前配置文件spark-env.sh设定压缩配置的参数。

```
export SPARK_JAVA_OPTS="-Dspark.broadcast.compress"
```

（2）在应用程序中配置

sc是SparkContext对象，conf是SparkConf对象。

```
val conf=sc.getConf
```

1）获取压缩的配置。

```
conf.getBoolean( "spark.broadcast.compress" , true )
```

2）压缩的配置。

```
conf.set("spark.broadcast.compress",true)
```

其他参数如表4-2所示。

表4-2　压缩配置的其他参数

参数	取值	说明
spark.broadcast.compress	true	参数决定broadcast变量是否进行压缩。通常情况下压缩它是一个好的选择
spark.rdd.compress	false	设置此参数决定是否压缩一个已经序列化的RDD，用户可以在创建RDD时，通过StorageLevel.MEMORY_ONLY_SER设定是否序列化。这样虽然耗费一些压缩时间，但是可以节省大量的内存空间
spark.io.compression.codec	org.apache.spark.io.LZFCompressionCodec	用户通过这个参数决定是采用LZF，还是Snappy压缩算法。LZF压缩率较高，Snappy的压缩时间短，用户可以根据需求，进行具体权衡
spark.io.compression.snappy.block.size	32768	用户通过这个参数设置Snappy压缩算法的块大小

在分布式计算中，序列化和压缩是两个重要的手段。Spark通过序列化将链式分布的数据转化为连续分布的数据，这样就能够进行分布式的进程间数据通信，或者在内存进行数据压缩等操作，提升Spark的应用性能。通过压缩，能够减少数据的内存占用，以及IO和网络数据传输开销。

4.3.3 Spark块管理

RDD在逻辑上是按照Partition分块的，可以将RDD看成是一个分区作为数据项的分布式数组。这也是Spark在极力做到的一点，让编写分布式程序像编写单机程序一样简单。而物理上存储RDD是以Block为单位的，一个Partition对应一个Block，用Partition的ID通过元数据的映射到物理上的Block，而这个物理上的Block可以存储在内存，也可以存储在某个节点的Spark的硬盘临时目录，等等。

整体的I/O管理分为以下两个层次。

1）通信层：I/O模块也是采用Master-Slave结构来实现通信层的架构，Master和Slave之间传输控制信息、状态信息。

2）存储层：Spark的块数据需要存储到内存或者磁盘，有可能还需传输到远端机器，这些是由存储层完成的。

通过图4-10，可以大致了解整个Spark存储（Store）模块。下面从以下几个方面介绍存储模块。

1.实体和类

可以从以下几个维度理解整个存储系统。

（1）管理和接口

BlockManager：当其他模块要和storage模块进行交互时，storage模块提供了统一的操作类BlockManager，外部类与storage模块打交道都需要调用BlockManager相应接口来实现。

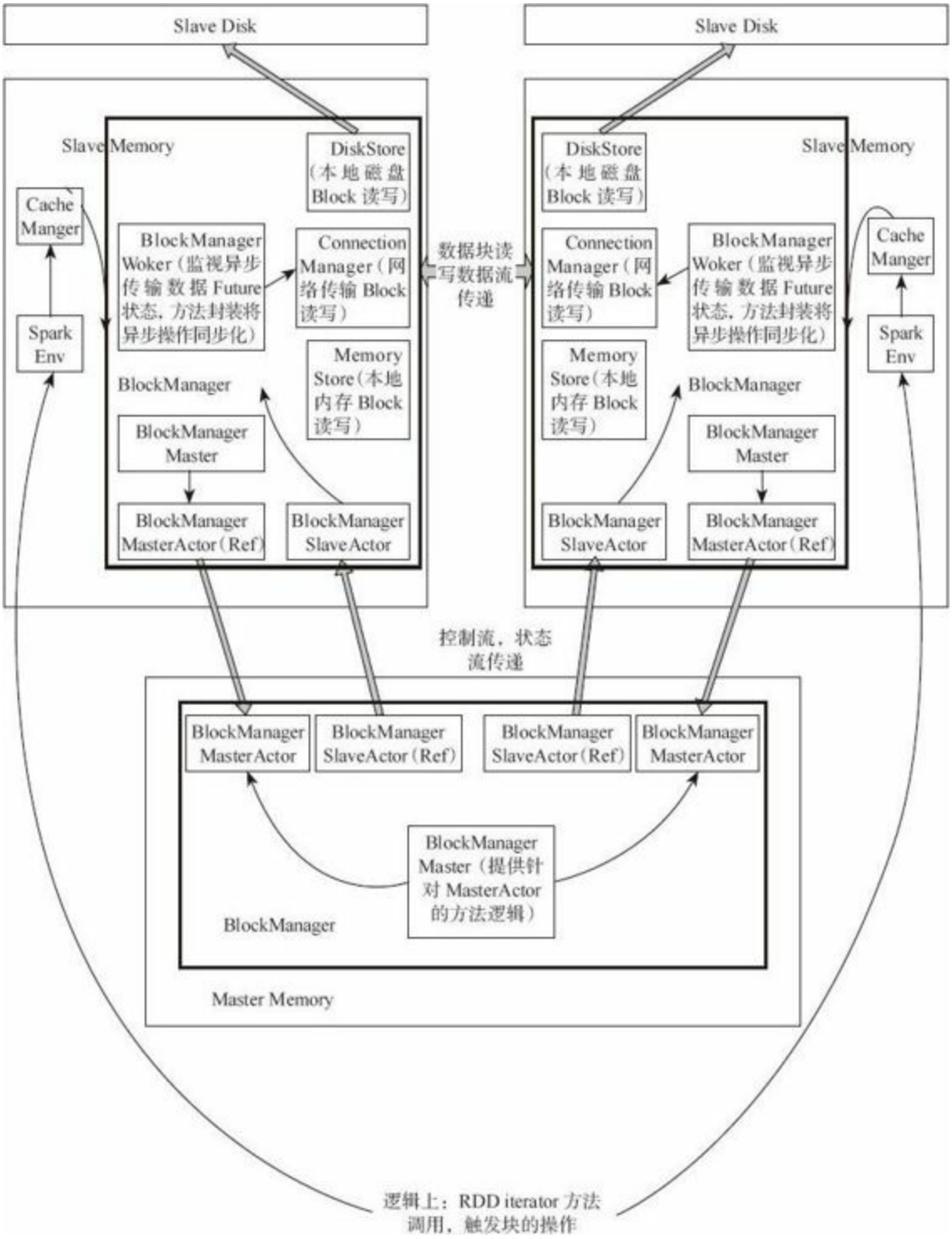

图4-10　Spark存储模块全景

（2）通信层

·BlockManagerMasterActor：从主节点创建，从节点通过这个Actor的引用向主节点传递消息和状态。

·BlockManagerSlaveActor：在从节点创建，主节点通过这个Actor的引用向从节点传递命令，控制从节点的块读写。

·BlockManagerMaster：对Actor通信进行管理。

（3）数据读写层

·DiskStore：提供Block在磁盘上以文件形式读写的功能。

·MemoryStore：提供Block在内存中的Block读写功能。

·ConnectionManager：提供本地机器和远端节点进行网络传输Block的功能。

·BlockManagerWorker：对远端数据的异步传输进行管理。

2.BlockManager中的通信

主节点和从节点之间通过Actor传送消息来传递命令和状态。

各个类在Master和Slave上所扮演的角色如图4-11所示。

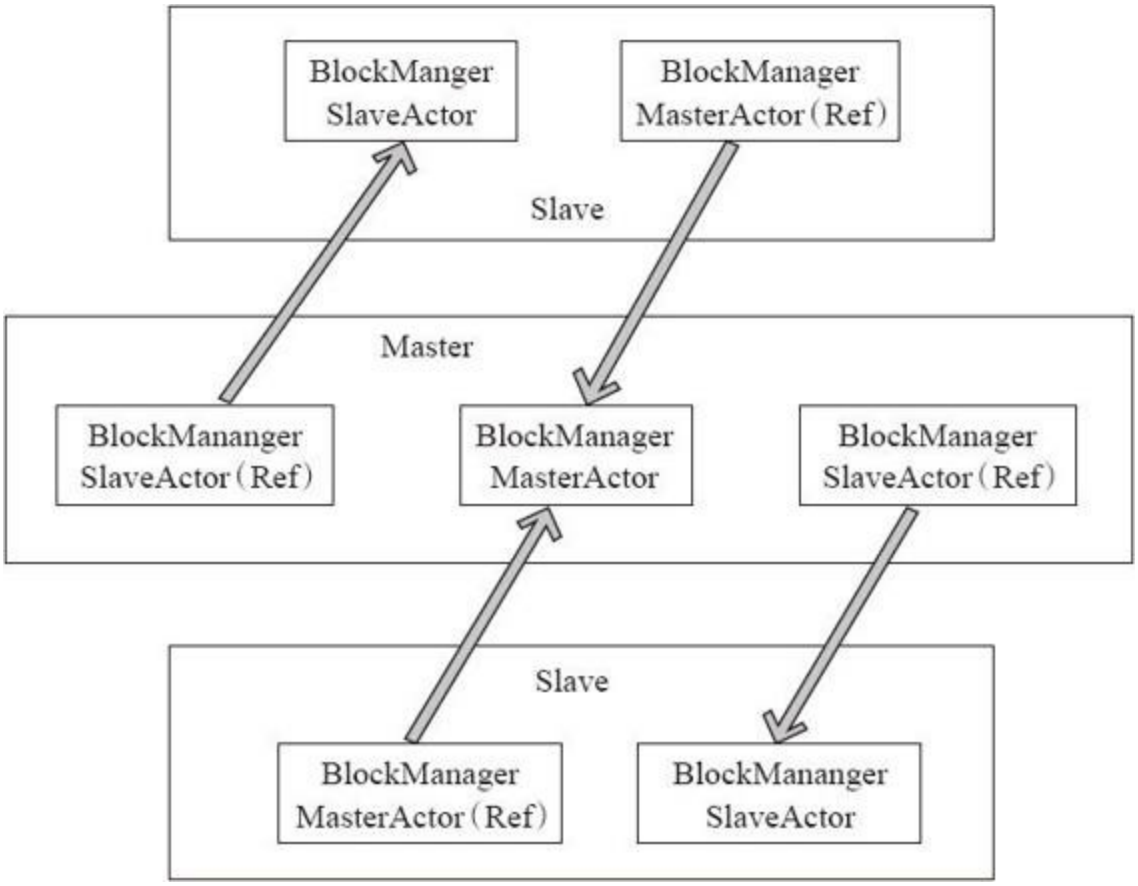

图4-11　Spark存储模块通信

整体的数据存储通信仍相当于Master-Slave模型，节点之间传递消息和状态，Master节点负责总体控制，Slave节点接收命令、汇报状态。（补充介绍：Actor和ref是AKKA中两个不同的Actor引用。）

BlockManager的创建对于Master和Slave来说有所不同。

（1）Master端

BlockManagerMaster对象拥有BlockManagerMasterActor的actor引用以及所有BlockManagerSlaveActor的ref引用。

（2）Slave端

对于Slave，BlockManagerMaster对象拥有BlockManagerMasterActor对象的ref的引用和自身BlockManagerSlaveActor的actor的引用。BlockManagerMasterActor在ref和Actor之间通信，BlockManagerSlaveActor在ref和Actor之间通信。

BlockManager在内部封装BlockManagerMaster，并通过BlockManagerMaster进行通信。Spark在各节点创建各自的BlockManager，通过BlockManager对storage模块进行操作。BlockManager对象在SparkEnv中创建，SparkEnv相当于线程的上线下文变量，在SparkEnv中也会创建很多的管理组件。例如，connectionManager、broadcastManager、cacheManager等的创建过程如下。

```
private[spark] def create(
 conf: SparkConf,
 executorId: String,
 hostname: String,
 port: Int,
 isDriver: Boolean,
 isLocal: Boolean,
 listenerBus: LiveListenerBus = null): SparkEnv = {
……
mapOutputTracker.trackerActor = registerOrLookup(
 "MapOutputTracker",
 new MapOutputTrackerMasterActor(mapOutputTracker.asInstanceOf[MapOutputTrackerMaster], conf))
val blockManagerMaster = new BlockManagerMaster(registerOrLookup(
  "BlockManagerMaster", new BlockManagerMasterActor(isLocal, conf, listenerBus)), conf)
/*创建blockManager*/
```

```
val blockManager = new BlockManager(executorId, actorSystem, blockManagerMaster,
  serializer, conf, securityManager, mapOutputTracker)
val connectionManager = blockManager.connectionManager
val broadcastManager = new BroadcastManager(isDriver, conf, securityManager)
val cacheManager = new CacheManager(blockManager)
val httpFileServer =
  if (isDriver) {
     val server = new HttpFileServer(securityManager)
     server.initialize()
     conf.set("spark.fileserver.uri",  server.serverUri)
     server
  } else {
     null
  }
val metricsSystem = if (isDriver) {
  MetricsSystem.createMetricsSystem("driver", conf, securityManager)
} else {
  MetricsSystem.createMetricsSystem("executor", conf, securityManager)
}
metricsSystem.start()
val sparkFilesDir: String = if (isDriver) {
  Utils.createTempDir().getAbsolutePath
} else {
  "."
}
val shuffleManager = instantiateClass[ShuffleManager](
  "spark.shuffle.manager", "org.apache.spark.shuffle.hash.HashShuffleManager")
if (conf.contains("spark.cache.class")) {
   logWarning("The spark.cache.class property is no longer being used! Specify storage " + "levels using the
RDD.persist() method instead.")
}
new SparkEnv(
  executorId,
  actorSystem,
  serializer,
  closureSerializer,
  cacheManager,
  mapOutputTracker,
  shuffleManager,
  broadcastManager,
  blockManager,
  connectionManager,
  securityManager,
  httpFileServer,
  sparkFilesDir,
  metricsSystem,
  conf)
}
```

通信层中涉及许多控制消息和状态消息的通信以及消息处理，感兴趣的读者可以参照源码。

3.读写流程

（1）数据写入

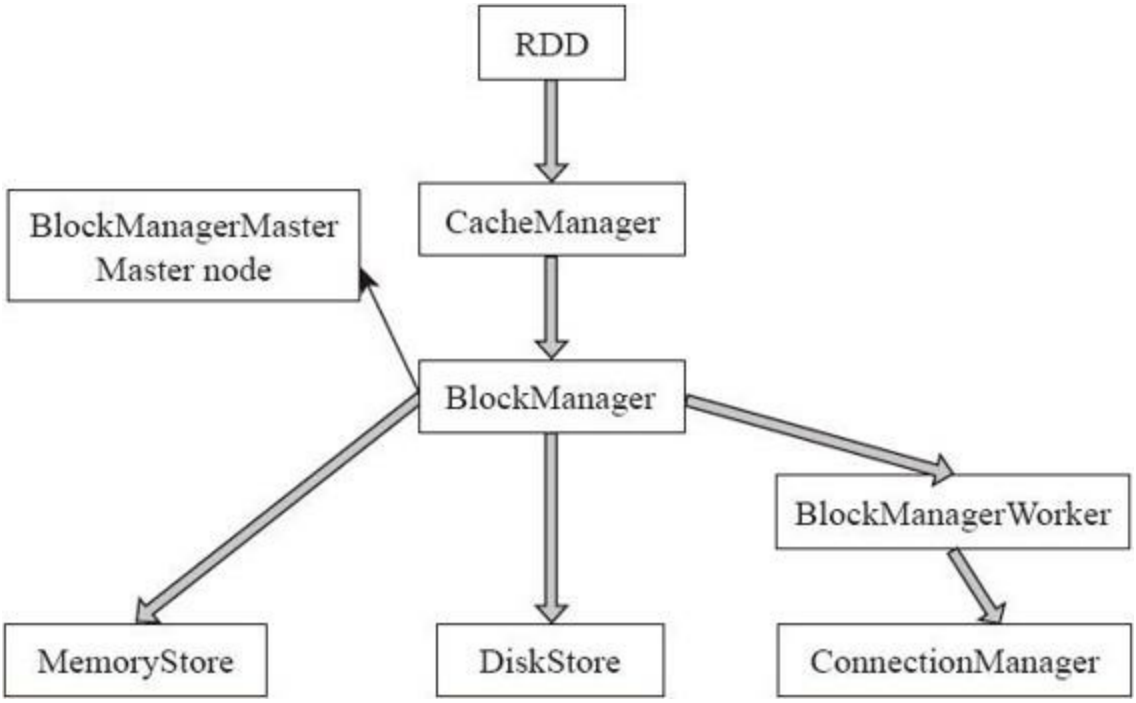

图4-12　Spark数据读写

数据写入的简要流程，读取流程和写入流程类似。数据写入流程主要分为以下几个步骤。

1）RDD调用compute（ ）方法进行指定分区的写入。

2）CacheManager中调用BlockManater判断数据是否已经写入，如果未写则写入。

3）BlockManager中数据与其他节点同步。

4）BlockManager根据存储级别写入指定的存储层。

5）BlockManager向主节点汇报存储状态。

详细步骤如下。

1）入口在RDD类中通过compute方法调用iterator方法进行某个分区Partition的读写，Partition是逻辑概念，在物理上是一个Block。其具体实现如下：

```
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
    if (storageLevel != StorageLevel.NONE) {
        SparkEnv.get.cacheManager.getOrCompute(this, split, context, storageLevel)
```

```
      } else {
         computeOrReadCheckpoint(split, context)
       }
   }
```

2）在CacheManager类中，getOrCompute方法通过调用BlockManager的put接口来写入数据。

我们可以看到，在这里有个判断逻辑，它先从内存cache读取是否有块可以读取，如果没有，则需要进行RDD的计算，通过触发RDD的执行和块的计算来加载数据。其具体实现如下：

```
def getOrCompute[T](
     rdd: RDD[T],
     partition: Partition,
     context: TaskContext,
     storageLevel: StorageLevel): Iterator[T] = {
  val key = RDDBlockId(rdd.id, partition.index)
  logDebug(s"Looking for partition $key")
  blockManager.get(key) match {
  ……
    case None =>
      ……
      /*如果BlockManager中还没有数据，则将数据写入BlockManager中*/
        val cachedValues = putInBlockManager(key, computedValues, storageLevel,
updatedBlocks)
        context.taskMetrics.updatedBlocks = Some(updatedBlocks)
        new InterruptibleIterator(context, cachedValues)
    ……
 }
 private def putInBlockManager[T](
    ……
      updatedBlocks ++= blockManager.put(key, values, storageLevel, tellMaster = true)
    ……
}
/*在BlockManager中，调用put方法*/
def put(
       blockId: BlockId,
       values: Iterator[Any],
       level: StorageLevel,
       tellMaster: Boolean): Seq[(BlockId, BlockStatus)] = {
       require(values != null, "Values is null")
       doPut(blockId, IteratorValues(values), level, tellMaster)
  }
/*调用doPut方法*/
private def doPut(
        blockId: BlockId,
        data: BlockValues,
        level: StorageLevel,
        tellMaster: Boolean = true): Seq[(BlockId, BlockStatus)] = {
……
```

3）将写入的数据与其他Woker进行同步。其具体实现如下：

```
val replicationFuture = data match {
        case b: ByteBufferValues if level.replication > 1 =>
```

```
            副本并不复制二进制数据，只是创建封装器
            val bufferView = b.buffer.duplicate( )
            Future { replicate(blockId, bufferView, level) }
          case _ => null
      }
……
var marked = false
       try {
```

4）根据用户设置的StorageLevel来判断数据写入哪个存储层。其具体实现如下：

```
val (returnValues, blockStore: BlockStore) = {
 if (level.useMemory) {
   /*优先写入内存，即设置useDisk为真，如果内存不能存储完，再写入磁盘*/
   (true, memoryStore)
 } else if (level.useOffHeap) {
   /*写入tachyon，这样存储于Java Heap之外的内存空间*/
   (false, tachyonStore)
 } else if (level.useDisk) {
   them
   /*写入磁盘*/
   (level.replication > 1, diskStore)
 } else {
   assert(level == StorageLevel.NONE)
   throw new BlockException(
        blockId, s"Attempted to put block $blockId without specifying storage
        level!")
      }
    }
  /*依据适用的Store类写入相应的存储，这里体现面向接口的编程*/
    val result = data match {
      case IteratorValues(iterator) =>
        blockStore.putValues(blockId, iterator, level, returnValues)
      case ArrayBufferValues(array) =>
        blockStore.putValues(blockId, array, level, returnValues)
      case ByteBufferValues(bytes) =>
        bytes.rewind( )
        blockStore.putBytes(blockId, bytes, level)
     }
……
```

5）通知BlockManagerMaster有新数据写入，在BlockManagerMaster中保存元数据。代码实现如下：

```
    reportBlockStatus(blockId, putBlockInfo, putBlockStatus)
  ……
}
```

（2）数据读取

在RDD类中，通过compute方法调用iterator读写某个分区（Partition），作为数据读取的入口。分区是逻辑概念，在物理上是一个数据块（block）。

```
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
      if (storageLevel != StorageLevel.NONE) {
         SparkEnv.get.cacheManager.getOrCompute(this, split, context, storageLevel)
      } else {
       computeOrReadCheckpoint(split, context)
    }
  }
/*在CacheManager的getOrCompute方法中*/
def getOrCompute[T](rdd: RDD[T], split: Partition, context: TaskContext,
       storageLevel: StorageLevel): Iterator[T] = {
  ……
   /*本质调用BlockManager的get方法获取数据*/
     blockManager.get(key) match {
       case Some(values) =>
         // Partition is already materialized, so just return its values
         new InterruptibleIterator(context, values.asInstanceOf[Iterator[T]])
       case None =>
……
}
```

（3）读取逻辑

通过下面BlockManager读取代码进入读取逻辑。

```
private[spark] class BlockManager(
     executorId: String,
     actorSystem: ActorSystem,
     val master: BlockManagerMaster,
     val defaultSerializer: Serializer,
     maxMemory: Long,
     val conf: SparkConf,
     securityManager: SecurityManager,
     mapOutputTracker: MapOutputTracker)
  extends Logging {
……
 def get(blockId: BlockId): Option[Iterator[Any]] = {
     /*如果需要读取的数据块在本地，则返回本地的数据块*/
     val local = getLocal(blockId)
     ……
     /*如果需要的数据块不在本地，则远端获取（Fetch）数据块*/
     val remote = getRemote(blockId)
     ……
     /*如果远端也没有，则数据块不存在*/
     None
  }
……
}
```

1）本地读取。

在本地同步读取数据块，首先看能否在内存读取数据块，如果不能读取，则看能否从Tachyon读取数据块，如果仍不能读取，则看能否从磁盘读取数据块。

```
private def doGetLocal(blockId: BlockId, asValues: Boolean): Option[Any] = {
     val info = blockInfo.get(blockId).orNull
     if (info != null) {
           /*同步读取数据块*/
        info.synchronized {
```

```
        ……
            /*在内存读取数据块*/
            if (level.useMemory) {
                logDebug("Getting block " + blockId + " from memory")
                val result = if (asValues) {
                    memoryStore.getValues(blockId)
                } else {
                    memoryStore.getBytes(blockId)
                }
              ……
              }
             /*在Tachyon读取数据块*/
             if (level.useOffHeap) {
                logDebug("Getting block " + blockId + " from tachyon")
                if (tachyonStore.contains(blockId)) {
                   tachyonStore.getBytes(blockId) match {
                     case Some(bytes) => {
                       if (!asValues) {
                          return Some(bytes)
                       } else {
                         return Some(dataDeserialize(blockId, bytes))
                       }
                     }
                     case None =>
                       logDebug("Block " + blockId + " not found in tachyon")
                   }
                }
              }
              /*在磁盘读取数据块*/
              if (level.useDisk) {
                 logDebug("Getting block " + blockId + " from disk")
                 val bytes: ByteBuffer = diskStore.getBytes(blockId) match {
                case Some(bytes) => bytes
                case None =>
                  throw new Exception("Block " + blockId + " not found on disk, though
                  it should be")
              }
            ……
        }
```

2）远程读取。

远程获取调用路径，然后getRemote调用doGetRemote，通过BlockManagerWorker.syncGetBlock从远程获取数据。

```
    private def doGetRemote(blockId: BlockId, asValues: Boolean): Option[Any] = {
        ……
        /*获取远端数据块，返回data数据块*/
            val data = BlockManagerWorker.syncGetBlock(
              GetBlock(blockId), ConnectionManagerId(loc.host, loc.port))
        ……
    }
```

在BlockManagerWorker中调用syncGetBlock获取远端数据块，这里使用了Future模型。Future本身是一种被广泛运用的并发设计模式，可在很大程度上简化需要数据流同步的并发应用开发。这里以java.util.concurrent.Future为例，简单介绍Future的具体工作方式。Future对象本身可以看做是一个显式的引用，一个对异步处理结果的引用。由于其异步性

质，在创建之初，它所引用的对象可能还并不可用（如尚在运算中、网络传输中或等待中）。这时，得到Future的程序流程如果并不急于使用Future所引用的对象，就可以做其他需要做的工作，当流程进行到需要Future背后引用的对象时，可能有以下两种情况。

第一种情况：希望能看到这个对象可用，并完成一些相关的后续流程。如果实在不可用，则也可以进入其他分支流程。

第二种情况：没有这个结果，则无法执行下去（可以设置超时进行时间限制）。对于前一种情况，可以通过调用Future.isDone（）判断引用的对象是否就绪，并采取不同的处理；后一种情况则只需调用get（）或get（long timeout，TimeUnit unit），通过同步阻塞方式等待对象就绪。实际运行期是阻塞，还是立即返回，取决于get（）的调用时机和对象就绪的先后。Future模式可以在连续流程中满足数据驱动的并发需求，这样既获得了并发执行的性能提升，又不失连续流程的简洁优雅。

下面通过SyncGetBlock方法了解获取数据块的方式。

```
def syncGetBlock(msg: GetBlock, toConnManagerId: ConnectionManagerId): ByteBuffer = {
  ……
/*返回responseMessage对象相当于是个Future*/
    val responseMessage = connectionManager.sendMessageReliablySync(
        toConnManagerId, blockMessageArray.toBufferMessage)
  ……
  }
 def sendMessageReliablySync(connectionManagerId: ConnectionManagerId,
      message: Message): Option[Message] = {
    Await.result(sendMessageReliably(connectionManagerId, message), Duration.Inf)
  }
```

在ConnectionManager中，通过sendMessage方法获取远端数据，通过Future异步计算模型获取远端读取结果状态。

```
def sendMessageReliably(connectionManagerId: ConnectionManagerId, message: Message)
      : Future[Option[Message]] = {
    val promise = Promise[Option[Message]]
    val status = new MessageStatus(
      message, connectionManagerId, s => promise.success(s.ackMessage))
    messageStatuses.synchronized {
      messageStatuses += ((message.id, status))
    }
    sendMessage(connectionManagerId, message)
    promise.future
```

```
    }
```

在sendMessage方法中：

```
private def sendMessage(connectionManagerId: ConnectionManagerId, message: Message) {
    ……
     /*需要和远端身份验证建立连接*/
        connection.getAuthenticated().synchronized {
     /*发送消息*/
    connection.send(message)
    wakeupSelector()
  }
```

4.数据块读写管理

数据块的读写，如果在本地内存存在所需数据块，则先从本地内存读取，如果不存在，则看本地的磁盘是否有数据，如果仍不存在，再看网络中其他节点上是否有数据，即数据有3个类别的读写来源。

（1）MemoryStore内存块读写

通过源码可以看到进行块读写是线程间同步的。通过entries.synchronized控制多线程并发读写，防止出现异常。

PutBlock对象用来确保只有一个线程写入数据块。这样确保数据读写且线程安全的。示例代码如下：

```
private val putLock = new Object()
```

内存Block块管理是通过链表来实现的，如图4-13所示。

```
private val entries = new LinkedHashMap[BlockId, MemoryEntry](32, 0.75f, true)
```

图4-13　MemoryStore数据存储格式

MemoryStroe内存快读写示例代码如下所示。通过getValues等方法为入口进行数据块的同步读，通过trytoPut等方法为入口进行数据块的同步写。

```
/*读取内存数据块*/
override def getValues(blockId: BlockId): Option[Iterator[Any]] = {
    /*同步读取数据*/
    val entry = entries.synchronized {
      entries.get(blockId)
    }
    if (entry == null) {
       None
    } else if (entry.deserialized) {
      Some(entry.value.asInstanceOf[ArrayBuffer[Any]].iterator)
    } else {
      val buffer = entry.value.asInstanceOf[ByteBuffer].duplicate() /* 实际并不复制数据*/
      Some(blockManager.dataDeserialize(blockId, buffer))
    }
  }
  /*内存写入数据块*/
 private def tryToPut(
       blockId: BlockId,
       value: Any,
       size: Long,
       deserialized: Boolean): ResultWithDroppedBlocks = {
    ……
    /*同步进行数据写*/
    putLock.synchronized {
    ……
    /*如果有足够内存空间*/
    if (enoughFreeSpace) {
         val entry = new MemoryEntry(value, size, deserialized)
    /*互斥写入entries容器*/
        entries.synchronized {
          entries.put(blockId, entry)
          currentMemory += size
        }
        val valuesOrBytes = if (deserialized) "values" else "bytes"
        logInfo("Block %s stored as %s in memory (estimated size %s, free %s)".format(
           blockId, valuesOrBytes, Utils.bytesToString(size), Utils.bytesToString
(freeMemory)))
        putSuccess = true
      } else {
    ……
}
```

（2）DiskStore磁盘块写入

在DiskStore中，一个Block对应一个文件。在diskManager中，存储blockId和一个文件路径映射。数据块的读写入相当于读写文件流。

写入二进制数据的具体实现代码如下所示。

```
override def putBytes(blockId: BlockId, _bytes: ByteBuffer, level: StorageLevel): PutResult = {
    val bytes = _bytes.duplicate()
   ……
    val file = diskManager.getFile(blockId)
    /*获取这个块对应的文件输出流*/
```

```
  val channel = new FileOutputStream(file).getChannel
  while (bytes.remaining > 0) {
  /*将数据块写入文件*/
      channel.write(bytes)
  }
  ……
}
```

4.4 Spark通信模块

Spark的Cluster Manager可以有Local、Standalone、Mesos、YARN等部署模式。为了研究Spark的通信机制，本节介绍Standalone模式，其他模式可以对照此模式进行类比，感兴趣的读者可以进一步通过源码了解。

下面介绍分布式通信的几种方式。

（1）RPC（Remote Produce Call）

RPC是远程过程调用协议，基于C/S模型调用。过程大致可以理解为本地分布式对象向本机发请求，不用自己编写底层通信本机。通过网络向服务器发送请求，服务器对象接收参数后，进行处理，再把处理后的结果发送回客户端。

（2）RMI（Remote Method Invocation）

RMI和RPC一样都是调用远程的方法，可以把RMI看做是用Java语言实现了RPC协议。RPC不支持对象通信，支持对象传输，这也是RMI相比于RPC的优越之处。

（3）JMS（Java Remote Service）

JMS是在各个Java类之间传递消息的标准。其支持P2P和pub/stub两种消息模型，即点对点和发布订阅两种模型。其优点在于：支持异步通信、消息生产者和消费者耦合度低。应用程序通过读写出入队列的消息（针对应用程序的数据）来通信，而无须专用连接来连接它们。

（4）EJB（Enterprise Java Bean）

EJB是Java EE中的一个规范，该规范描述了分布式应用程序需要解决的事务处理，安全、日志、分布式等问题，与此同时，Sun公司也实现了自己定义的这一个标准，相当于自

己颁布一个标准。EJB是一个特殊的Java类，按照Java服务器接口定义，放在容器里可以帮助该类管理事务、分布式、安全等，只有大型分布式系统才会用到EJB，一般小型程序不会用到。

（5）Web Service

Web Service是网络间跨语言、跨平台的分布式系统间的通信标准。传输XML、JSON等格式的数据，应用范围较广。

4.4.1 通信框架AKKA

Spark在模块间通信使用的是AKKA框架。AKKA基于Scala开发，用于编写Actor应用。Actor模型在并发编程中是比较常见的一种模型。很多开发语言都提供了原生的Actor模型（Erlang、Scala）。Actors是一些包含状态和行为的对象。它们通过显式传递消息来进行通信，这些消息会被发送到它们的收件箱中（消息队列）。从某种意义上来说，Actor是面向对象编程中最严格的实现形式。它们之间可以通过消息来通信。一个Actor收到其他Actor的信息后，可以根据需要做出各种响应。通过Scala的强大模式匹配功能可以让用户自定义多样化的消息。Actor建立一个消息队列，每次收到消息后，放入队列，而它每次也从队列中取出消息体来处理。通常情况下，这个过程是循环的。让Actor可以时刻接收处理发送来的消息。

注意：一个ActorSystem是一个重量级的结构。它会分配N个线程。所以对于每一个应用来说，仅创建一个ActorSystem即可。

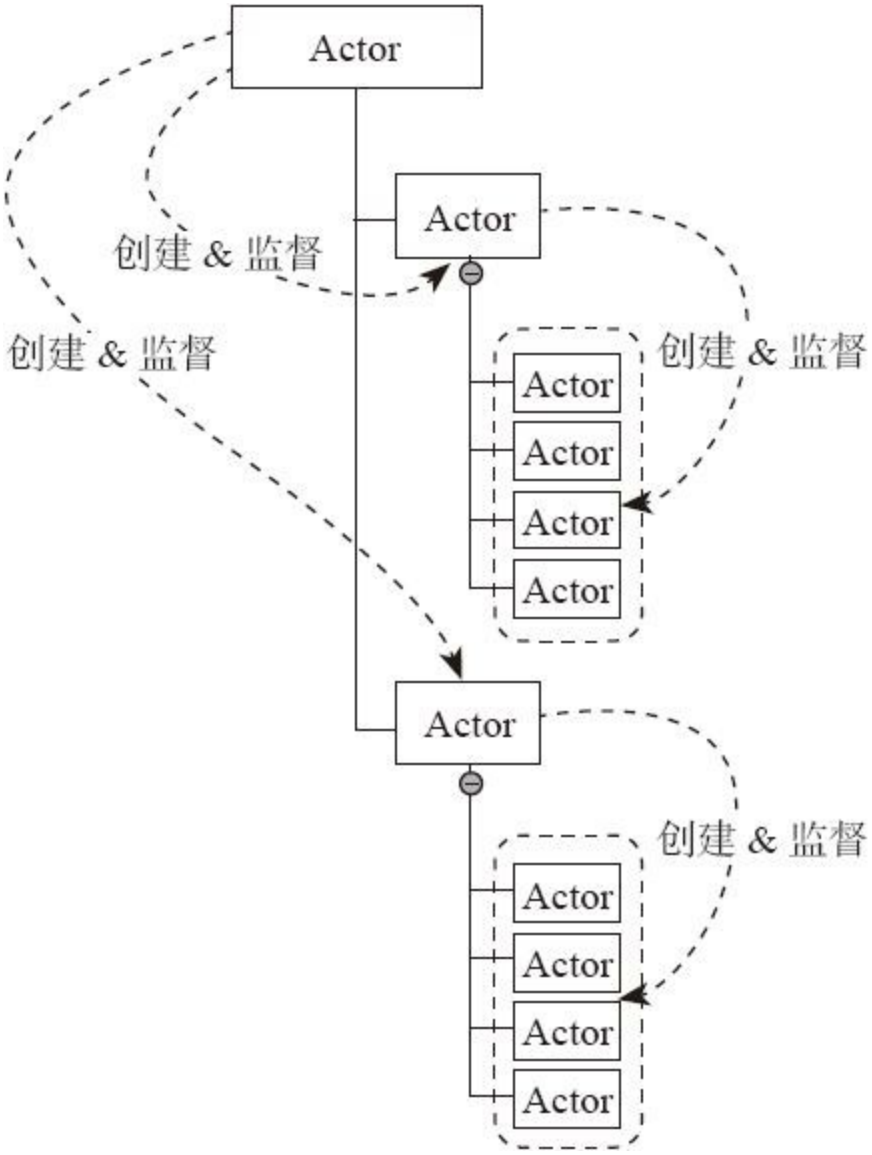

图4-14　actor模型

AKKA Actor树形结构Actors以树形结构组织起来。一个Actor可能会把自己的任务划分成更多更小的、利于管理的子任务。为了达到这个目的，它会开启自己的子Actor，并负责监督这些子Actor。关于监督的具体细节就不在这里讨论了。我们只需知道一点，就是每一个Actor都会有一个监督者，即创建这些Actor的Actor。

AKKA的优势和特性如下。

1）并行和分布式：AKKA在设计时采用了异步通信和分布式架构。

2）可靠性：在本地/远程都有监控和恢复机制。

3）高性能：在单机环境中每秒可发送50000000个消息。1GB内存中可创建和保持

2500000个Actor对象。

4）去中心：区别于Master-Slave模式，采取无中心节点的架构。

5）可扩展性：可以在分布式环境下进行Scale out，线性扩充计算能力。

可以看到AKKA具有强大的并发处理能力，在国内，豌豆荚对AKKA集群做了很有深度的研究和实践，感兴趣的读者可以进一步了解。

Spark中并没有充分挖掘AKKA强大的并行计算能力，而是将其作为分布式系统中的RPC框架。很多组件封装为Actor，进行控制和状态通信。

Spark中的Client、Master和Worker都是一个Actor。

例如，Master通过worker.actor！LaunchDriver（driver.id，driver.desc）向Worker节点发送启动Driver命令消息，在Worker节点中通过receive的方式响应命令消息。

```
override def receive = {
……
 case LaunchDriver(driverId, driverDesc) => {
……
    }
……
```

综上所述，通过AKKA简洁地实现了Spark模块间通信。

4.4.2 Client、Master和Worker间的通信

在Standalone模式下，存在以下角色。

·Client：提交作业。

·Master：接收作业，启动Driver和Executor，管理Worker。

·Worker：管理节点资源，启动Driver和Executor。

1.模块间的主要消息

这里结合图4-15列出了各个模块之间传递的主要消息及其作用：

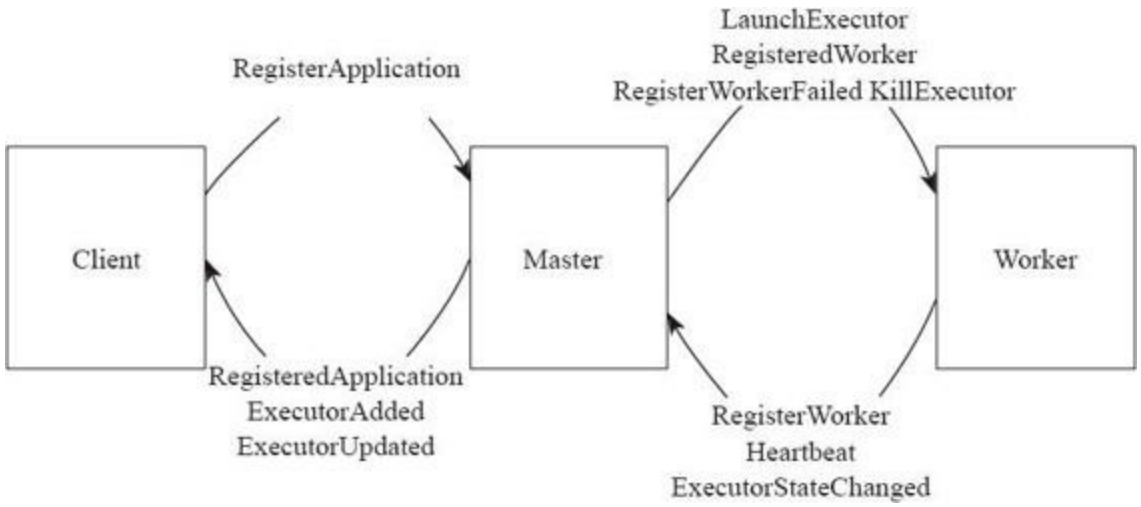

图4-15 Spark通信模型

（1）Client to Master

RegisterApplication：注册应用。

（2）Master to Client

·RegisteredApplication：注册应用后，回复给Client。

·ExecutorAdded：通知Client Worker已经启动了Executor，当向Worker发送Launch-

Executor时，通知Client Actor。

·ExecutorUpdated：通知Client Executor状态已更新。

（3）Master to Worker

·LaunchExecutor：启动Executor。

·RegisteredWorker：Worker注册的回复。

·RegisterWorkerFailed：注册Worker失败的回复。

·KillExecutor：停止Executor进程。

（4）Worker to Master

·RegisterWorker：注册Worker。

·Heartbeat：周期性地Master发送心跳信息。

·ExecutorStateChanged：通知Master，Executor状态更新。

2.主要的通信逻辑

Actor之间，消息发送端通过“！”符号发送消息，接收端通过receive方法中的case模式匹配接收和处理消息。下面通过源码介绍Client、Master、Worker这3个Actor的主要通信接收逻辑。

（1）Client Actor通信代码逻辑

Client Actor通信代码逻辑如下所示。

```
private class ClientActor(driverArgs: ClientArguments, conf: SparkConf) extends Actor with Logging {
……
override def preStart() = {
```

```
    masterActor = context.actorSelection(Master.toAkkaUrl(driverArgs.master))
   ……
    driverArgs.cmd match {
      case "launch" =>
     ……
  /*在这段代码向Master 的Actor提交Driver*/
       masterActor ! RequestSubmitDriver(driverDescription)
  ……
    }
  }
……
override def receive = {
   /*接收Master命令在Worker创建Driver成功与否的消息*/
    case SubmitDriverResponse(success, driverId, message) =>
      println(message)
      if (success) pollAndReportStatus(driverId.get) else System.exit(-1)
   /*接收终止Driver成功与否的通知*/
    case KillDriverResponse(driverId, success, message) =>
      println(message)
      if (success) pollAndReportStatus(driverId) else System.exit(-1)
    ……
  }
}
```

（2）Master Actor通信代码逻辑

Master Actor通信代码逻辑如下所示。

```
private[spark] class Master(
     host: String,
     port: Int,
     webUiPort: Int,
     val securityMgr: SecurityManager)
extends Actor with Logging {
……
override def receive = {
   /*选举为Master，并判断该Master的State为RecoveryState.RECOVERING，恢复beginRecovery*/
    case ElectedLeader => {
     ……
    }
   /*完成恢复*/
    case CompleteRecovery => completeRecovery()
    ……
/*注册Worker*/
    case RegisterWorker(id, workerHost, workerPort, cores, memory, workerUiPort,
    publicAddress) =>
    {
      ……
      /*如果Master的状态为RecoveryState.STANDBY，则不对Worker进行注册*/
      if (state == RecoveryState.STANDBY) {
      } else if (idToWorker.contains(id)) {
      /*该Worker已经注册，通知Worker不能重复注册*/
        sender ! RegisterWorkerFailed("Duplicate worker ID")
      } else {
        val worker = new WorkerInfo(id, workerHost, workerPort, cores, memory,
          sender, workerUiPort, publicAddress)
        if (registerWorker(worker)) {
          persistenceEngine.addWorker(worker)
       /*Worker注册成功，通知Worker*/
          sender ! RegisteredWorker(masterUrl, masterWebUiUrl)
          schedule()
        } else {
        ……
      /*Worker注册失败，通知Worker*/
          sender ! RegisterWorkerFailed("Attempted to re-register worker at same
              address: "+ workerAddress)
        }
```

```
    }
  }
  case RequestSubmitDriver(description) => {
  /*如果Master的状态为ALIVE，则提交Driver，否则通知Client无法提交*/
    if (state != RecoveryState.ALIVE) {
      val msg = s"Can only accept driver submissions in ALIVE state. Current state: $state."
      sender ! SubmitDriverResponse(false, None, msg)
    } else {
      ……
     /*提交Driver*/
      sender ! SubmitDriverResponse(true, Some(driver.id),
        s"Driver successfully submitted as ${driver.id}")
    }
  }
  case RequestKillDriver(driverId) => {
    if (state != RecoveryState.ALIVE) {
    /*如果Master状态不是ALIVE，则通知请求者无法终止Driver*/
       val msg = s"Can only kill drivers in ALIVE state. Current state: $state."
      sender ! KillDriverResponse(driverId, success = false, msg)
    } else {
      logInfo("Asked to kill driver " + driverId)
      val driver = drivers.find(_.id == driverId)
      driver match {
        case Some(d) =>
          if (waitingDrivers.contains(d)) {
          /*如果请求终止的Driver在等待队列，则从队列中删除Driver并更新Driver状态为KILLED*/
            waitingDrivers -= d
            self ! DriverStateChanged(driverId, DriverState.KILLED, None)
          } else {
           /*通知所有Worker，查看Worker上是否运行着需要被终止运行的Driver进程，如果存
           在则终止相应进程*/
           d.worker.foreach { w =>
              w.actor ! KillDriver(driverId)
            }
          }
          val msg = s"Kill request for $driverId submitted"
          logInfo(msg)
          sender ! KillDriverResponse(driverId, success = true, msg)
        case None =>
          // 通知请求者，请求被终止运行的Driver已经被终止运行或者不存在
         ……
      }
    }
  }
  case RequestDriverStatus(driverId) => {
/*请求查找指定Driver的状态，如果找到，则返回相应的状态*/
      ……
  }
  case RegisterApplication(description) => {
    if (state == RecoveryState.STANDBY) {
    } else {
     /*如果Master的状态不为STANDBY，则创建并注册Application，并通知请求者*/
      logInfo("Registering app " + description.name)
      val app = createApplication(description, sender)
      registerApplication(app)
      logInfo("Registered app " + description.name + " with ID " + app.id)
      persistenceEngine.addApplication(app)
      sender ! RegisteredApplication(app.id, masterUrl)
      schedule()
    }
  }
  case ExecutorStateChanged(appId, execId, state, message, exitStatus) => {
   /*通过元数据映射，获取到Executor，然后通知使用这个Executor的Driver更新Executor状
          态。如果执行完，则移除Executor，如果异常退出，则移除Application*/
   ……
  }
  case DriverStateChanged(driverId, state, exception) => {
    /*当Driver的state为ERROR | FINISHED | KILLED | FAILED时，删除这个Driver*/
    }
  }
  case Heartbeat(workerId) => {
    idToWorker.get(workerId) match {
      case Some(workerInfo) =>
    /*更新Worker的最近心跳时间为最新时间*/
```

```
            workerInfo.lastHeartbeat = System.currentTimeMillis()
          ……
        }
      }
      case MasterChangeAcknowledged(appId) => {
      ……
      /*将指定的App状态置为WAITING，为下一步切换Master做准备*/
            app.state = ApplicationState.WAITING
      ……
      }
      case WorkerSchedulerStateResponse(workerId, executors, driverIds) => {
       /*如果找到指定的Worker，则将Worker的状态置为ALIVE，并且查找对应App状态为idDefined的Executors，将这些executors都加入
app中，然后保存这些Exectutor信息到Worker中，并将DriverIds中的Driver加入这个Worker中*/
       ……
      }
      case DisassociatedEvent(_, address, _) => {
        /*Worker或者Application发送请求，删除请求的Worker*/
        addressToWorker.get(address).foreach(removeWorker)
        /*删除请求的应用*/
        addressToApp.get(address).foreach(finishApplication)
        /*如果满足条件，则终止恢复*/
        if (state == Recov· eryState.RECOVERING && canCompleteRecovery) { completeRecovery() }
      }
      case RequestMasterState => {
        /*向请求者返回Master状态*/
        sender ! MasterStateResponse(host, port, workers.toArray, apps.toArray, completedApps.toArray,
          drivers.toArray, completedDrivers.toArray, state)
      }
      case CheckForWorkerTimeOut => {
        /*检查并删除超时Worker*/
      }
      case RequestWebUIPort => {
        /*向请求者返回Web UI的端口号*/
      }
    }
  ……
}
```

3.Worker Actor的消息处理逻辑

Worker Actor的消息处理逻辑，将通过下面的代码分析进行介绍。

```
private[spark] class Worker(
  host: String,
  port: Int,
  webUiPort: Int,
  cores: Int,
  memory: Int,
  masterUrls: Array[String],
  actorSystemName: String,
  actorName: String,
  workDirPath: String = null,
  val conf: SparkConf,
  val securityMgr: SecurityManager)
  extends Actor with Logging {
  import context.dispatcher
……
override def receive = {
    case RegisteredWorker(masterUrl, masterWebUiUrl) =>
      /*Worker收到Master传回的注册成功消息，然后Worker配置对应的Master*/
      ……
    case SendHeartbeat =>
      /*收到主节点消息后，向主节点发送心跳，证明本Worker存活*/
      masterLock.synchronized {
        if (connected) { master ! Heartbeat(workerId) }
      }
    case WorkDirCleanup =>
```

```
    /*启动一个独立的线程去清理旧应用的目录和文件*/
    val cleanupFuture = concurrent.future {
         logInfo("Cleaning up oldest application directories in " + workDir + " ...")
         Utils.findOldFiles(workDir, APP_DATA_RETENTION_SECS)
          .foreach(Utils.deleteRecursively)
      }
      ……
    case MasterChanged(masterUrl, masterWebUiUrl) =>
      /*当选举出新的Master时，Worker更新Master节点URL等信息*/
      logInfo("Master has changed, new master is at " + masterUrl)
      changeMaster(masterUrl, masterWebUiUrl)
     ……
/*从Driver节点接收心跳消息*/
    case Heartbeat =>
     ……
/*在主节点注册Worker失败*/
    case RegisterWorkerFailed(message) =>
    /*启动Executor*/
    case LaunchExecutor(masterUrl, appId, execId, appDesc, cores_, memory_) =>
      /*启动Executor进程*/
      if (masterUrl != activeMasterUrl) {
         logWarning("Invalid Master (" + masterUrl + ") attempted to launch executor.")
      } else {
        try {
          logInfo("Asked to launch executor %s/%d for %s".format(appId, execId, appDesc.name))
          val manager = new ExecutorRunner(appId, execId, appDesc, cores_, memory_,
            self, workerId, host,
            appDesc.sparkHome.map(userSparkHome => new File(userSparkHome)).getOrElse(sparkHome),
            workDir, akkaUrl, conf, ExecutorState.RUNNING)
          /*对元数据进行更新*/
          executors(appId + "/" + execId) = manager
          manager.start()
          coresUsed += cores_
          memoryUsed += memory_
          masterLock.synchronized {
            master ! ExecutorStateChanged(appId, execId, manager.state, None, None)
          }
        } catch {
          case e: Exception => {
            logError("Failed to launch executor %s/%d for %s".format(appId, execId, appDesc.name))
            if (executors.contains(appId + "/" + execId)) {
                executors(appId + "/" + execId).kill()
                executors -= appId + "/" + execId
            }
            masterLock.synchronized {
              master ! ExecutorStateChanged(appId, execId, ExecutorState.FAILED,
                  None, None)
            }
          }
        }
      }
    /*Executor状态更新*/
    case ExecutorStateChanged(appId, execId, state, message, exitStatus) =>
      masterLock.synchronized {
        /*同步通知Master节点，Executor状态进行了更新*/
        master ! ExecutorStateChanged(appId, execId, state, message, exitStatus)
      }
      val fullId = appId + "/" + execId
      if (ExecutorState.isFinished(state)) {
          executors.get(fullId) match {
          /*如果Executor完成工作，则释放资源*/
          case Some(executor) =>
            logInfo("Executor " + fullId + " finished with state " + state +
              message.map(" message " + _).getOrElse("") +
              exitStatus.map(" exitStatus " + _).getOrElse(""))
            executors -= fullId
            finishedExecutors(fullId) = executor
            coresUsed -= executor.cores
            memoryUsed -= executor.memory
         ……
        }
      }
   /*终止Executor*/
    case KillExecutor(masterUrl, appId, execId) =>
```

```
      /*终止本Worker节点上运行的Executor*/
      ……
        executors.get(fullId) match {
          case Some(executor) =>
            logInfo("Asked to kill executor " + fullId)
            executor.kill()
  ……
        }
      }
    /*启动Driver*/
    case LaunchDriver(driverId, driverDesc) => {
      /*接收Master节点命令，启动Driver*/
      logInfo(s"Asked to launch driver $driverId")
      val driver = new DriverRunner(driverId, workDir, sparkHome, driverDesc, self, akkaUrl)
      drivers(driverId) = driver
      driver.start()
      coresUsed += driverDesc.cores
      memoryUsed += driverDesc.mem
    }
    case KillDriver(driverId) => {
       /*终止本Worker节点正在运行的Driver*/
      ……
    }
    /*Driver状态更新*/
    case DriverStateChanged(driverId, state, exception) => {
      /*如果Worker上运行的Apllication的Driver状态为错误异常或者正常结束等，则打印相应的
状态日志，通知主节点并移除Driver*/
      state match {
        case DriverState.ERROR =>
          logWarning(s"Driver $driverId failed with unrecoverable exception:
                    ${exception.get}")
        case DriverState.FAILED =>
          logWarning(s"Driver $driverId exited with failure")
        case DriverState.FINISHED =>
          logInfo(s"Driver $driverId exited successfully")
        case DriverState.KILLED =>
          logInfo(s"Driver $driverId was killed by user")
        case _ =>
          logDebug(s"Driver $driverId changed state to $state")
      }
      masterLock.synchronized {
        master ! DriverStateChanged(driverId, state, exception)
      }
      val driver = drivers.remove(driverId).get
      finishedDrivers(driverId) = driver
      memoryUsed -= driver.driverDesc.mem
      coresUsed -= driver.driverDesc.cores
    }
    case x: DisassociatedEvent if x.remoteAddress == masterAddress =>
      /*与主节点失去连接，更新connected状态为false，等待Master重新连接*/
      logInfo(s"$x Disassociated !")
      masterDisconnected()
    case RequestWorkerState => {
      /*向请求者返回本Worker的目前状态信息*/
      sender ! WorkerStateResponse(host, port, workerId, executors.values.toList,
        finishedExecutors.values.toList, drivers.values.toList,
        finishedDrivers.values.toList, activeMasterUrl, cores, memory,
        coresUsed, memoryUsed, activeMasterWebUiUrl)
    }
  }
……
}
```

4.5 容错机制

在众多特性中，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。面向大规模数据分析，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。因此，Spark选择记录更新的方式。但是，如果更新粒度太细太多，那么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Lineage（即血统）记录下来，以便恢复丢失的分区。Lineage本质上很类似于数据库中的重做日志（Redo Log），只不过这个重做日志粒度很大，是对全局数据做同样的重做进而恢复数据。

4.5.1 Lineage机制

最后，为了说明模型的容错性，图4-16给出了3个算子的血统（lineage）关系图。在lines RDD上执行filter操作，得到errors，然后filter、map后得到新的RDD（filter、map和collect都是Spark中对RDD的函数操作）。Spark调度器以流水线的方式执行后三个转换，向拥有errors分区缓存的节点发送一组任务。此外，如果某个errors分区丢失，则Spark只在相应的lines分区上执行filter操作来重建该errors分区。

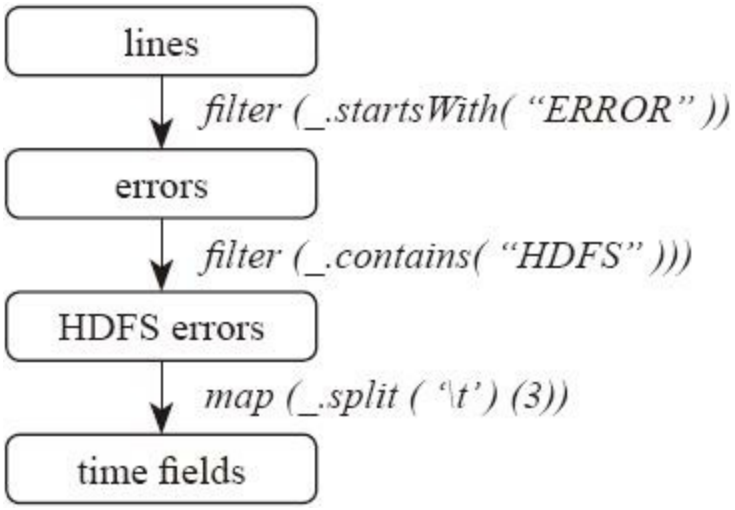

图4-16 RDD Lineage

1.Lineage简介

相比其他系统的细颗粒度的内存数据更新级别的备份或者LOG机制，RDD的Lineage记录的是粗颗粒度的特定数据Transformation操作（如filter、map、join等）行为。当这个RDD的部分分区数据丢失时，它可以通过Lineage获取足够的信息来重新运算和恢复丢失的数据分区。因为这种粗颗粒的数据模型，限制了Spark的运用场合，所以Spark并不适用于所有高性能要求的场景，但同时相比细颗粒度的数据模型，也带来了性能的提升。

2.两种依赖

RDD在Lineage依赖方面分为两种：Narrow Dependencies与Shuffle Dependencies，用来解决数据容错的高效性。Narrow Dependencies是指父RDD的每一个分区最多被一个子

RDD的分区所用，表现为一个父RDD的分区对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区，也就是说一个父RDD的一个分区不可能对应一个子RDD的多个分区。Shuffle Dependencies是指子RDD的分区依赖于父RDD的多个分区或所有分区，即存在一个父RDD的一个分区对应一个子RDD的多个分区。

本质理解：根据父RDD分区是对应1个还是多个子RDD分区来区分Narrow Dependency（父分区对应一个子分区）和Shuffle Dependency（父分区对应多个子分区）。如果对应多个，则当容错重算分区时，因为父分区数据只有一部分是需要重算子分区的，其余数据重算就造成了冗余计算。

·Narrow Dependency：1个父RDD分区对应1个子RDD分区，这其中又分两种情况：1个子RDD分区对应1个父RDD分区（如map、filter等算子），1个子RDD分区对应N个父RDD分区（如co-paritioned（协同划分）过的Join）。

·Shuffle Dependency：1个父RDD分区对应多个子RDD分区，这其中又分两种情况：1个父RDD对应所有子RDD分区（未经协同划分的Join）或者1个父RDD对应非全部的多个RDD分区（如groupByKey）。

对于Shuffle Dependencies，Stage计算的输入和输出在不同的节点上，对于输入节点完好，而输出节点死机的情况，通过重新计算恢复数据这种情况下，这种方法容错是有效的，否则无效，因为无法重试，需要向上追溯其祖先看是否可以重试（这就是lineage，血统的意思），Narrow Dependencies对于数据的重算开销要远小于Wide Dependencies的数据重算开销。

Narrow Dependency和Shuffle Dependency的概念主要用在两个地方：一个是容错中相当于Redo日志的功能；另一个是在调度中构建DAG作为不同Stage的划分点。

3.容错原理

在容错机制中，如果一个节点死机了，而且运算Narrow Dependency，则只要把丢失的父RDD分区重算即可，不依赖于其他节点。而Shuffle Dependency需要父RDD的所有分区都存在，重算就很昂贵了。可以这样理解开销的经济与否：在Narrow Dependency中，在子RDD的分区丢失、重算父RDD分区时，父RDD相应分区的所有数据都是子RDD分区的数据，并不存在冗余计算。在Shuffle Dependency情况下，丢失一个子RDD分区重算的每个父RDD的每个分区的所有数据并不是都给丢失的子RDD分区用的，会有一部分数据相当于对应的是未丢失的子RDD分区中需要的数据，这样就会产生冗余计算开销，这也是Shuffle Dependency开销更大的原因。因此如果使用Checkpoint算子来做检查点，不仅要考虑Lineage是否足够长，也要考虑是否有宽依赖，对Shuffle Dependency加Checkpoint是最物有所值的。下面结合图4-17进行分析。

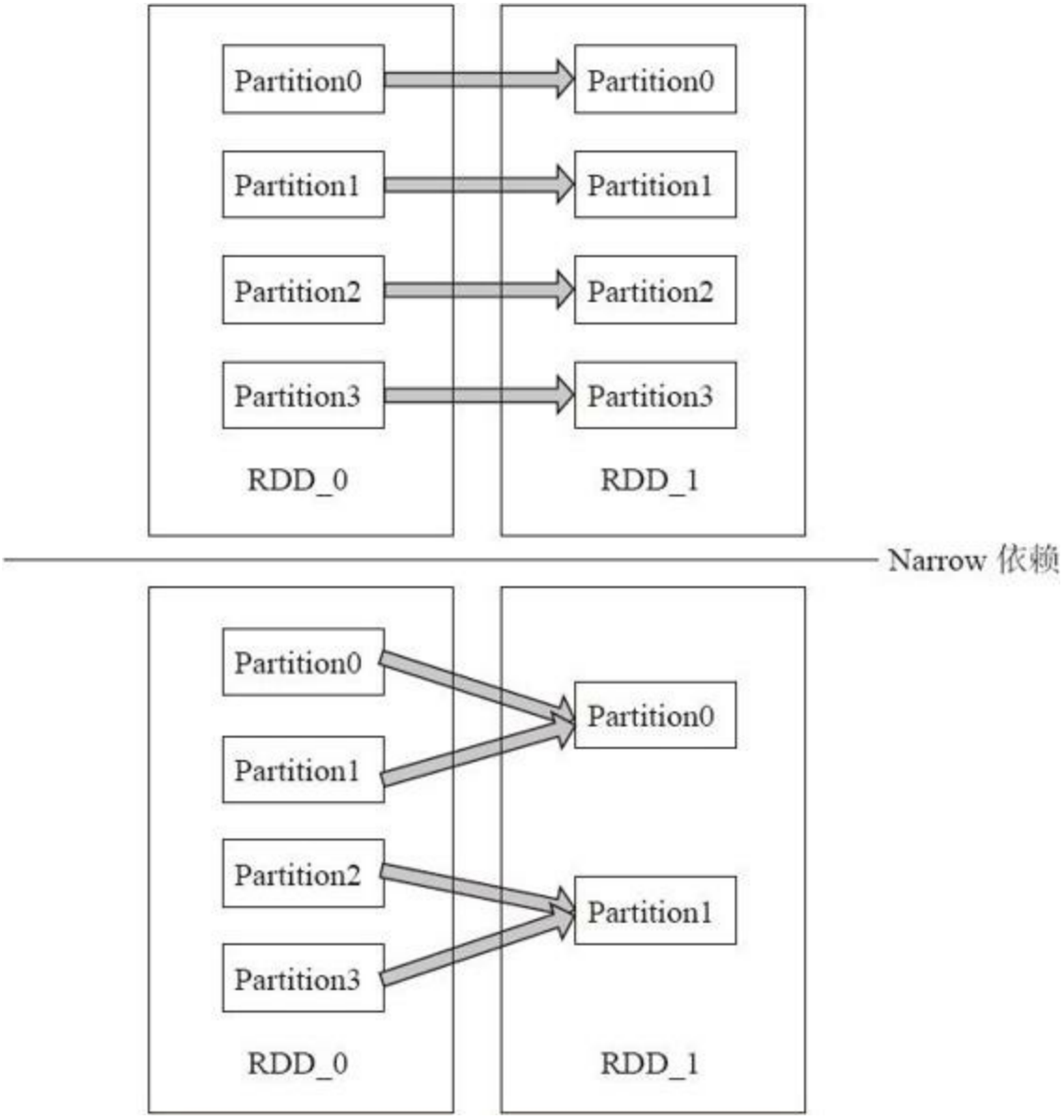

图4-17　Narrow依赖

以图4-17上端的图为例，如果RDD_1中的Partition3出错丢失，则Spark会回溯到Partition3的父分区RDD_0的Partition3，对RDD_0的Partition3重算算子，得到RDD_1的Partition3。其他分区丢失也是同理重算进行容错恢复。

以图4-18下端的图为例，其中RDD_1中的Partition3丢失出错，由于其父分区是RDD_0的所有分区，所以需要回溯到RDD_0，重算RDD_0的所有分区，然后将RDD_1的Partition3需要的数据聚集合并为RDD_1的Partition3。在这个过程中，由于RDD_0中不是RDD_1中Partition3需要的数据也全部进行了重算，所以产生了大量冗余数据重算的开销。

通过代码介绍容错的具体调用。

下面通过CacheManager类的getsOrCompute方法作用入口，进一步分析容错机制。

```
def getOrCompute[T](
      rdd: RDD[T],
      partition: Partition,
      context: TaskContext,
      storageLevel: StorageLevel): Iterator[T] = {
……
 case None =>
        val storedValues = acquireLockForPartition[T](key)
        if (storedValues.isDefined) {
           return new InterruptibleIterator[T](context, storedValues.get)
        }
        try {
           logInfo(s"Partition $key not found, computing it")
/*如果所需的分区丢失了，在BlockManager无法找到，这个分区就会重新计算（注意数据首次加载也相当于无法找到，需要重新计算）*/
val computedValues = rdd.computeOrReadCheckpoint(partition, context)
```

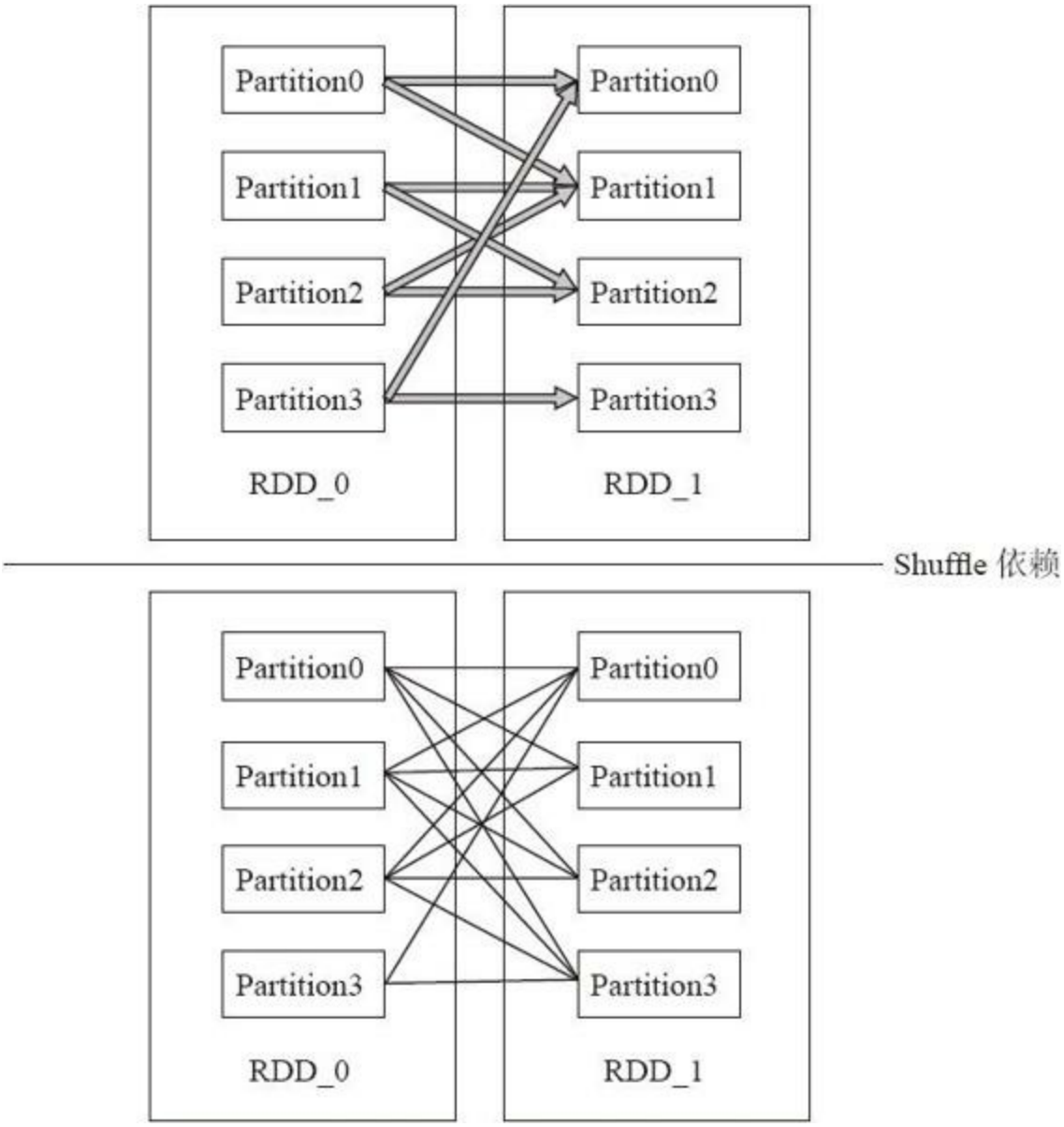

图4-18　Shuffle依赖

下面通过RDD的computeOrReadCheckpoint方法的代码进一步分析容错机制。

```
private[spark] def computeOrReadCheckpoint(split: Partition, context: TaskContext): Iterator[T] =
{
/*这里相当于对这个分区回溯到父节点或者祖先节点，然后一路计算回来得到这个分区，相当于只需要计算这个分区的依赖，因为是获取这个分区，而不是计算所有分区*/
  if (isCheckpointed) firstParent[T].iterator(split, context) else compute(split, context)
}
```

可以通过图4-19来理解重做Lineage的过程，虚线方框表示逻辑分区，相当于计算完成后就不存在了，已经转化为RDD_2中分区的数据。实线方框表示分区的重新计算过程就是由于RDD_2的分区丢失了，程序用到Partition0分区，找不到，就反向回溯Lineage到RDD_0的分区Partition0和Partition1，然后对其进行重新计算，计算结果为RDD_1的

Partition0，RDD_1再重新计算Partition0为RDD_2中的Partition0。这时就不需要其他分区参与计算了。

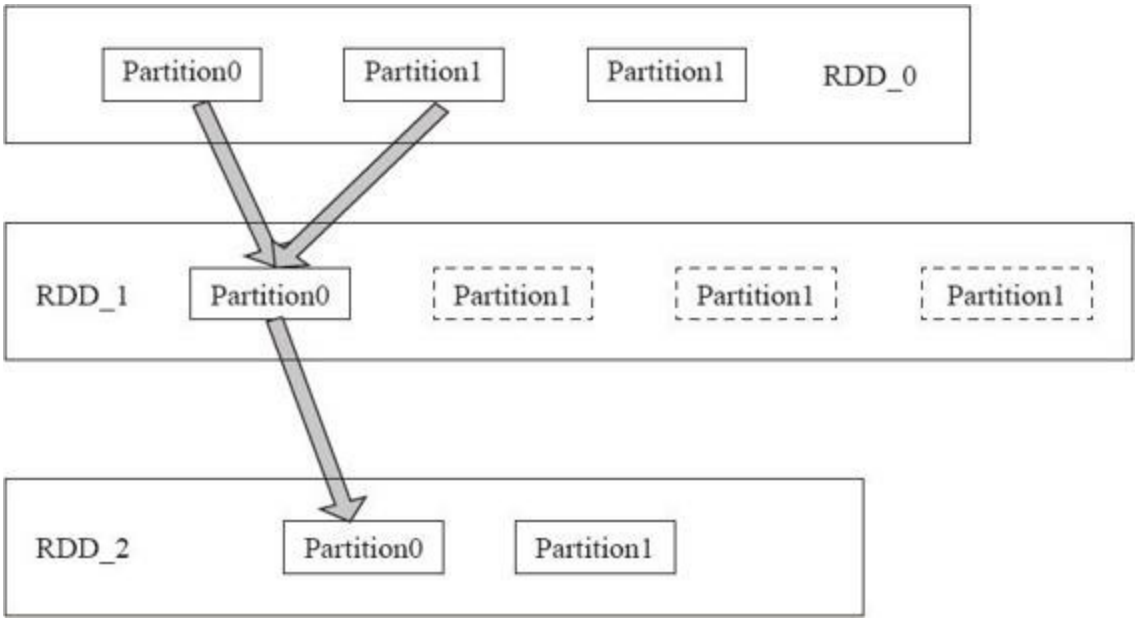

图4-19　回溯Lineage

4.5.2 Checkpoint机制

通过上述分析可以看出在以下两种情况下，RDD需要加检查点。

1）DAG中的Lineage过长，如果重算，则开销太大（如在PageRank中）。

2）在Shuffle Dependency上做Checkpoint（检查点）获得的收益更大。

由于RDD是只读的，所以Spark的RDD计算中一致性不是主要关心的内容，内存相对容易管理，这也是设计者很有远见的地方，这样减少了框架的复杂性，提升了性能和可扩展性，为以后上层框架的丰富奠定了强有力的基础。

在RDD计算中，通过检查点机制进行容错，传统做检查点有两种方式：通过冗余数据和日志记录更新操作。在RDD中的doCheckPoint方法相当于通过冗余数据来缓存数据，而之前介绍的血统就是通过相当粗粒度的记录更新操作来实现容错的。

在Spark中，通过RDD中的checkpoint（）方法来做检查点。

```
def checkpoint( ) : Unit
```

可以通过SparkContext.setCheckPointDir（）设置检查点数据的存储路径，进而将数据存储备份，然后Spark删除所有已经做检查点的RDD的祖先RDD依赖。这个操作需要在所有需要对这个RDD所做的操作完成之后再做，因为数据会写入持久化存储造成I/O开销。官方建议，做检查点的RDD最好是在内存中已经缓存的RDD，否则保存这个RDD在持久化的文件中需要重新计算，产生I/O开销。

下面通过源码来了解检查点的机制。

检查点（本质是通过将RDD写入Disk做检查点）是为了通过lineage做容错的辅助，lineage过长会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果之后有

节点出现问题而丢失分区，从做检查点的RDD开始重做Lineage，就会减少开销。

在RDD中通过doCheckpoint ( ) 方法作为检查点的入口方法。

```
    private[spark] def doCheckpoint ( ) {  ……
checkpointData.get.doCheckpoint ( )       } else
{         dependencies.foreach ( _.rdd.doCheckpoint ( ) )       }     ……   }
```

在RDDCheckpointData中，通过doCheckpoint ( ) 方法做检查点。

```
def doCheckpoint ( ) {   ……
```

RDD通过同步方式做检查点，具体使用Synchronized保证方法的同步和线程安全。代码实现如下。

```
/*path检查点RDD的输出文件路径*/
CheckpointData.synchronized {    ……
val path = new Path ( rdd.context.checkpointDir.get , "rdd-" + rdd.id )
val fs = path.getFileSystem ( new Configuration ( ) )
if ( ! fs.mkdirs ( path ) ) {
throw new SparkException ( "Failed to create checkpoint path " + path )     }
/*在SparkContext提交作业，将检查点RDD写入之前设置的路径中*/
       rdd.context.runJob ( rdd , CheckpointRDD.writeToFile ( path.toString ) _ )
     val newRDD = new CheckpointRDD[T] ( rdd.context , path.toString )   ……
  }
/*在CheckPointRDD中调用 writeToFile方法将RDD写入HDFS*/
def writeToFile[T] ( path : String , blockSize : Int = -1 ) ( ctx :
 TaskContext , iterator : Iterator[T] ) {
val env = SparkEnv.get
val outputDir = new Path ( path )
/*本质相当于在Hadoop 的分布式文件系统将RDD数据写进HDFS*/
val fs = outputDir.getFileSystem ( env.hadoop.newConfiguration ( ) )     val finalOutputName =
splitIdToFile ( ctx.splitId )
val finalOutputPath = new Path ( outputDir , finalOutputName )
val tempOutputPath = new Path ( outputDir , "." + finalOutputName + "-attempt-" + ctx.attemptId ) ……
/*根据数据量不同设置，不同的缓冲区大小*/
val bufferSize = System.getProperty ( "spark.buffer.size" , "65536" ).toInt
val fileOutputStream = if ( blockSize < 0 ) {
 fs.create ( tempOutputPath , false , bufferSize )
} else {       // This is mainly for testing purpose
       fs.create ( tempOutputPath , false , bufferSize ,
fs.getDefaultReplication , blockSize )     }
   /*创建序列化器*/
       val serializer = env.serializer.newInstance ( )
val serializeStream = serializer.serializeStream ( fileOutputStream )
/*此处为写入操作，关键是在iterator上相当于将iteraor迭代器的对象序列化写到HDFS中*/
    serializeStream.writeAll ( iterator )     serializeStream.close ( )     ……   }
SerializationStream写入操作
trait SerializationStream {
def writeObject[T] ( t : T ) : SerializationStream
def flush ( ) : Unit
def close ( ) : Unit
def writeAll[T] ( iter : Iterator[T] ) : SerializationStream = {
while ( iter.hasNext ) {       writeObject ( iter.next ( ) )     }     this   } }
/*如果配置了kyro序列化器进行写入，则调用下面的writeObject方法将数据序列化后写入HDFS*/
private[spark] class KryoSerializationStream ( kryo : Kryo , outStream : OutputStream ) extends
SerializationStream
```

```
{   val output = new KryoOutput(outStream)
……
def writeObject[T](t: T): SerializationStream = {
kryo.writeClassAndObject(output, t)
this   }
def flush() { output.flush() }
def close() { output.close() }
……
}
```

4.6 Shuffle机制

Shuffle的本义是洗牌、混洗，即把一组有一定规则的数据打散重新组合转换成一组无规则随机数据分区。Spark中的Shuffle更像是洗牌的逆过程，把一组无规则的数据尽量转换成一组具有一定规则的数据，Spark中的Shuffle和MapReduce中的Shuffle思想相同，在实现细节和优化方式上不同，因此掌握Hadoop的Shuffle原理的用户很容易将原有知识迁移过来。

为什么Spark计算模型需要Shuffle过程？我们都知道，Spark计算模型是在分布式的环境下计算的，这就不可能在单进程空间中容纳所有的计算数据来进行计算，这样数据就按照Key进行分区，分配成一块一块的小分区，打散分布在集群的各个进程的内存空间中，并不是所有计算算子都满足于按照一种方式分区进行计算。例如，当需要对数据进行排序存储时，就有了重新按照一定的规则对数据重新分区的必要，Shuffle就是包裹在各种需要重分区的算子之下的一个对数据进行重新组合的过程。在逻辑上还可以这样理解[1]：由于重新分区需要知道分区规则，而分区规则按照数据的Key通过映射函数（Hash或者Range等）进行划分，由数据确定出Key的过程就是Map过程，同时Map过程也可以做数据处理，例如，在Join算法中有一个很经典的算法叫Map Side Join，就是确定数据该放到哪个分区的逻辑定义阶段。Shuffle将数据进行收集分配到指定Reduce分区，Reduce阶段根据函数对相应的分区做Reduce所需的函数处理。

下面结合源码和图4-20从物理实现上看Spark的Shuffle是怎样实现的，将Shuffle分为两个阶段：Shuffle Write和Shuffle Fetch阶段（Shuffle Fetch中包含聚集Aggregate），在Spark中，整个Job转化为一个有向无环图（DAG）来执行，从图4-21中可以看出在整个DAG中是在每个Stage的承接阶段做Shuffle过程。

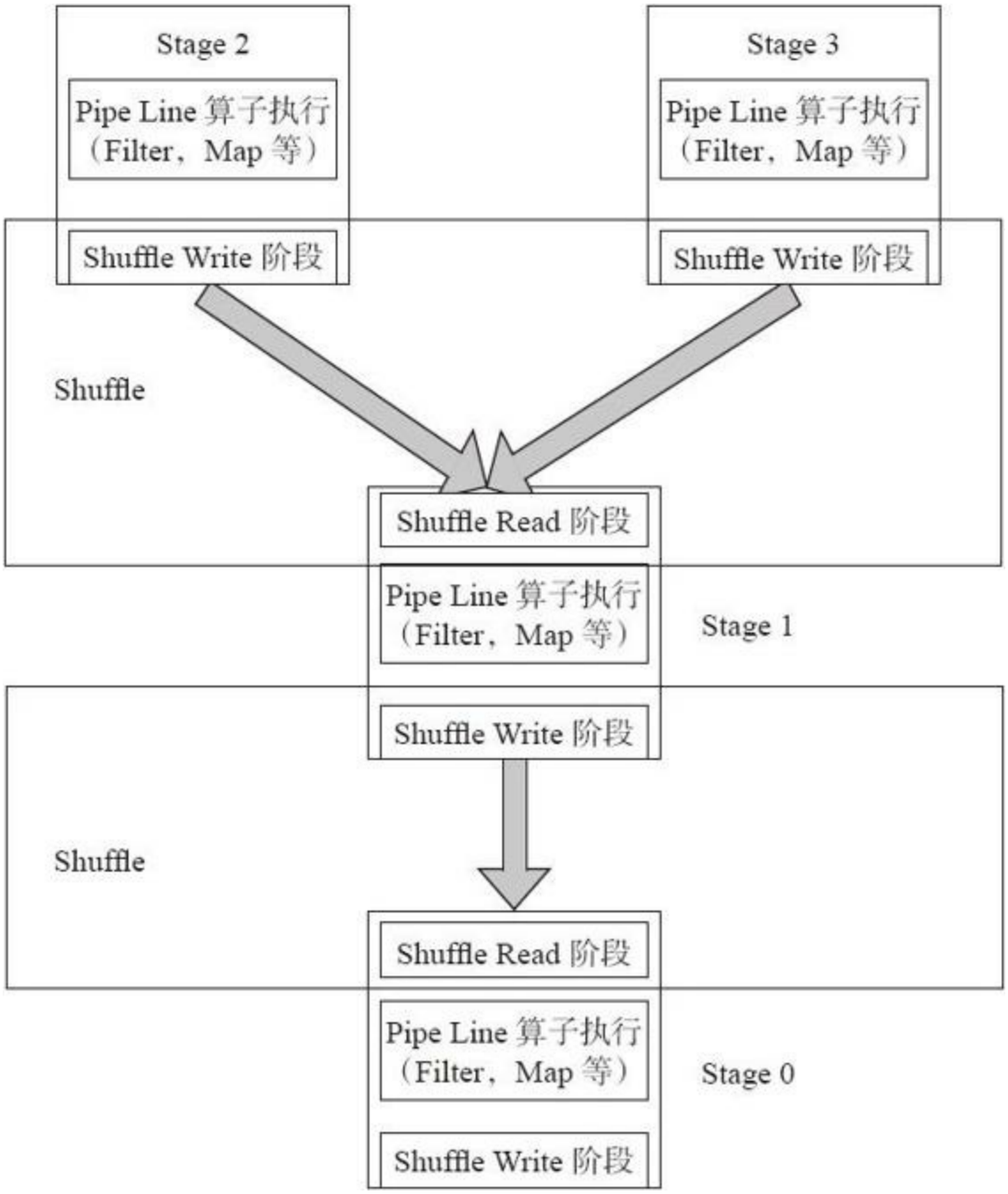

图4-20　Shuffle阶段图

图4-20中，整个Job分为Stage0~Stage3，4个Stage。

首先从最上端的Stage2、Stage3执行，每个Stage对每个分区执行变换（transformation）的流水线式的函数操作，执行到每个Stage最后阶段进行Shuffle Write，将数据重新根据下一个Stage分区数分成相应的Bucket，并将Bucket最后写入磁盘。这个过程就是Shuffle Write阶段。

执行完Stage2、Stage3之后，Stage1去存储有Shuffle数据节点的磁盘Fetch需要的数

据，将数据Fetch到本地后进行用户定义的聚集函数操作。这个阶段叫Shuffle Fetch，Shuffle Fetch包含聚集阶段。这样一轮一轮的Stage之间就完成了Shuffle操作。

下面我们更细粒度地将Shuffle的阶段进行拆分，以更深入剖析和了解。

1.Shuffle Write

由于Spark的每个Stage中是通过执行任务来进行运算的，而Spark中只分为两种任务，ShuffleMapTask和ResultTask。其中ResultTask就是最底层的Stage，也是整个任务执行的最后阶段将数据输出到Spark执行空间Stage，除了这个阶段执行ResultTask，其余阶段都执行ShuffleMapTask。因此主要的Shuffle Write逻辑存在这种任务的代码中。

由于Shuffle属于大数据优化的一个很重要的阶段，所以这里的代码优化会比较频繁，下面基于Spark 1.0的代码进行介绍，后续发展变化请读者参考相应版本。

（1）Shuffle Write流程

ShuffleWrite的入口是通过ShuffleMapTask中的runTask方法进入的，也是整个Shuffle Write的控制骨架。

```
override def runTask(context: TaskContext): MapStatus = {
 ……
  writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)
/*此处相当于使用ShuffleWriter将相应的分区进行Shuffle Write*/
  writer.write(rdd.iterator(split, context).asInstanceOf[Iterator[_ <: Product2[Any, Any]]])
  return writer.stop(success = true).get
  ……
  }
```

ShuffleWriter是个抽象的特征（Trait），下面看下它的具体实现。例如，我们看看HashShuffleWriter中是怎样实现的，HashShuffleWriter的主要功能其实就是判断是否需要做MapSideCombine或者做普通的Shuffle，并且提供Shuffle Write各个流程的函数。

```
  override def write(records: Iterator[_ <: Product2[K, V]]): Unit = {
private val shuffle = shuffleBlockManager.forMapTask(dep.shuffleId, mapId, numOutputSplits, ser)
/*这里判断是否进行MapSideCombine，也就是判断是否做Map端聚集合并，如果合并能够在Map端做，将会很大程度减少网络传输的数据量，减少开销*/
    val iter = if (dep.aggregator.isDefined) {
```

```
      if (dep.mapSideCombine) {
         dep.aggregator.get.combineValuesByKey(records, context)
      } else {
        records
      }
     ……
    for (elem <- iter) {
      val bucketId = dep.partitioner.getPartition(elem._1)
/*这里调用ShuffleWriterGroup的writers获取数据写入器，将数据写入bucket*/
      shuffle.writers(bucketId).write(elem)
    }
  }
```

下面进入ShuffleBlockManager，来分析最终要做的Shuffle Write逻辑。从这段代码中可以看出Spark支持两种类型的Shuffle：Shuffle和优化的Consolidate Shuffle。

```
 val writers: Array[BlockObjectWriter] = if (consolidateShuffleFiles) {
          fileGroup = getUnusedFileGroup()
          Array.tabulate[BlockObjectWriter](numBuckets) { bucketId =>
          val blockId = ShuffleBlockId(shuffleId, mapId, bucketId)
          /*两种Shuffle的区别其实是在对Bucket的处理是否写入FileGroup中
            FileGroup就是一个文件数组，存储文件的引用。在内存中维持这些FileGroup的引用*/
blockManager.getDiskWriter(blockId, fileGroup(bucketId), serializer, bufferSize)
        }
      } else {
        Array.tabulate[BlockObjectWriter](numBuckets) { bucketId =>
          val blockId = ShuffleBlockId(shuffleId, mapId, bucketId)
/*此处逻辑是获取相应的块，由于每次都是第一次获取，所以会创建新文件，这里每次都会产生新的文件*/
          val blockFile = blockManager.diskBlockManager.getFile(blockId)
          ……
          blockManager.getDiskWriter(blockId, blockFile, serializer, bufferSize)
        }
      }
```

其中，图4-21为Shuffle FileGroup的结构。

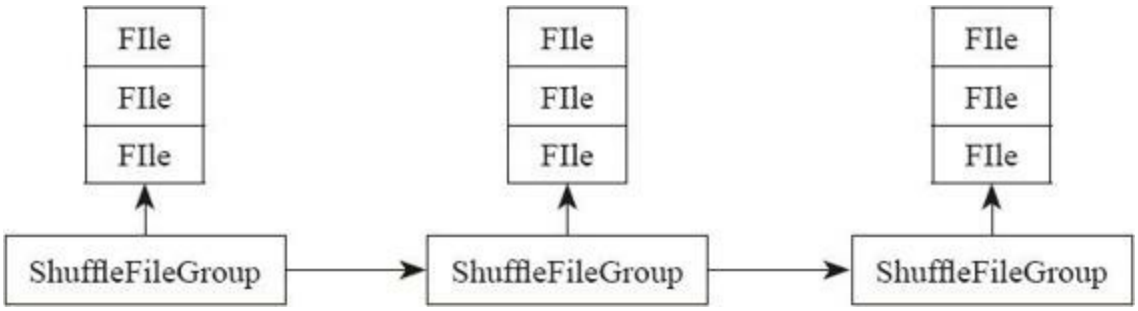

图4-21　Shuffle FileGroup结构

Shuffle做Shuffle Write的细节如图4-22所示。注意：这里的数据是直接写入缓冲中，而未经过排序。

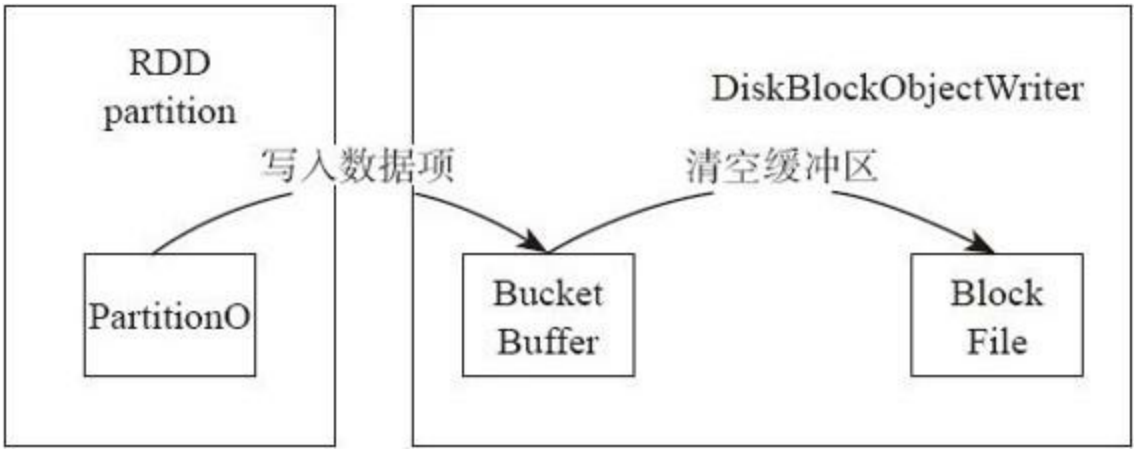

图4-22　Shuffle数据写入磁盘

最终在HashShuffleWriter，将内存的Bucket写到磁盘，存储为文件，并将Shuffle的各个Bucket及映射信息返回给主节点。

（2）Shuffle和Consolidate Shuffle对比

下面从图4-23和图4-24中，更加直观地对比Shuffle和Consolidate Shuffle的整体流程区别。

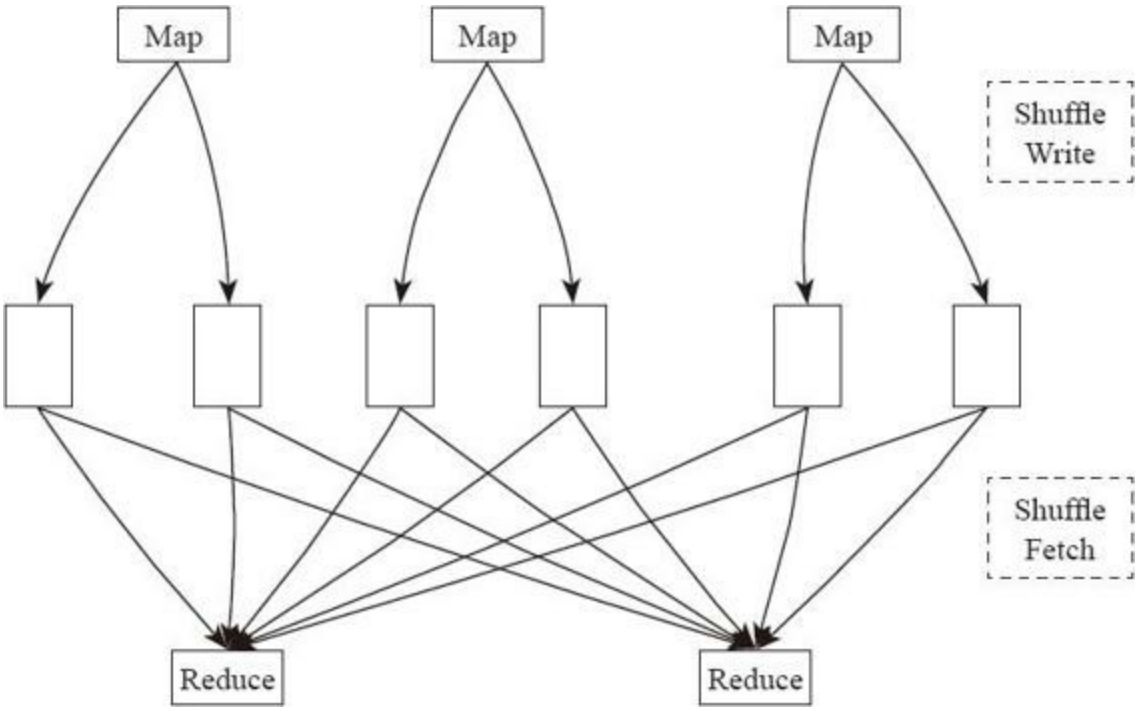

图4-23　Shuffle流程

图4-23中是进行Shuffle的整体流程，假定该Shuffle中有3个Mapper和2个Reducer，这样

会产生3×2=6个Bucket，也就是会产生6个Shuffle文件。因此，产生的Shuffle文件个数为M×R，M是Map任务个数，R是Reduce任务数。

图4-24是Consolidation Shuffle的流程图。其中每一个Bucket并非对应一个文件，而是对应文件中的一个segment，同时Consolidation Shuffle所产生的Shuffle文件数量与Spark Core的个数也具有相关性。在上面的图例中，Job的4个Mapper分为两批运行，在第一批2个Mapper运行时，申请4个Bucket，产生4个Shuffle文件；在第二批Mapper运行时，由于只有一个Mapper，申请的4个bucket并不会再产生4个新的文件，而是追加写到之前的其中两个文件后面，这样一共只有4个shuffle文件，而在文件内部这有6个不同的segment。因此，从理论上讲Shuffle Consolidation所产生的shuffle文件数量为C×R，其中C是Spark集群的Core Number，R是Reducer的个数。

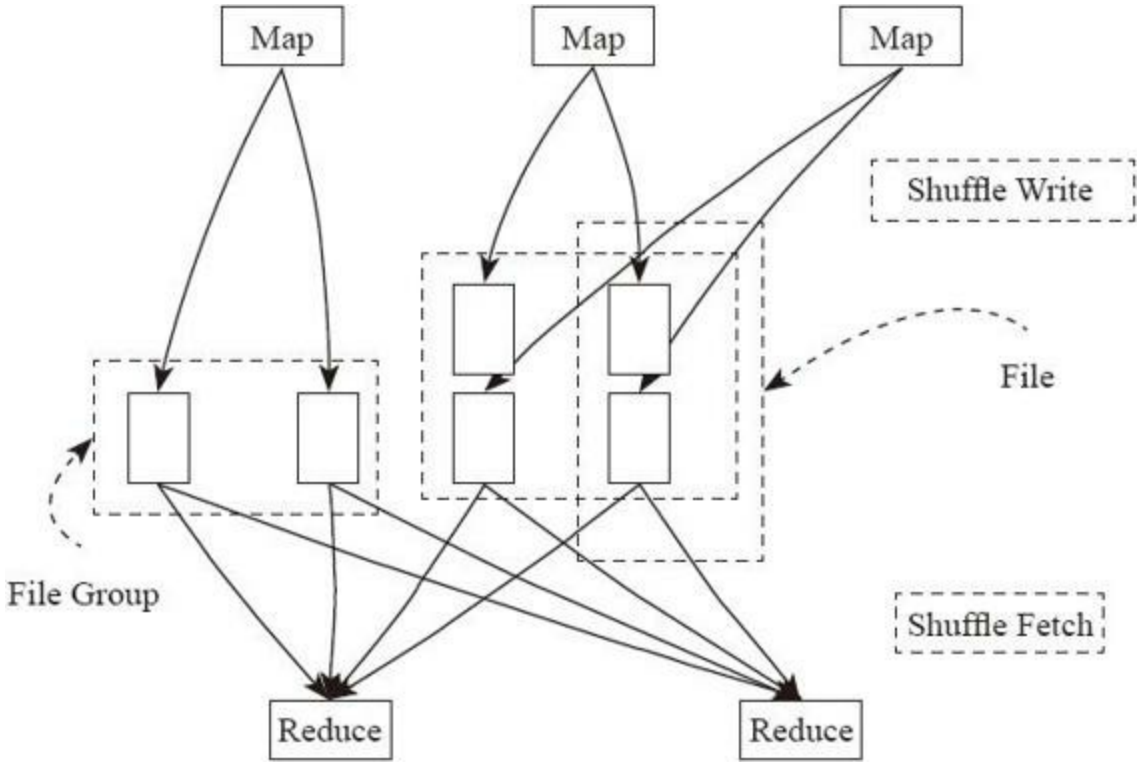

图4-24　Consolidation Shuffle流程

这里的特殊情况是当M=C时，Consolidation Shuffle所产生的文件数和之前的实现相同。

Consolidation Shuffle显著减少了shuffle文件的数量，解决了文件数量过多的问题，但

是Writer Handler的Buffer开销过大依然没有减少，若要减少Writer Handler的Buffer开销，只能减少Reducer的数量，但是这又会引入新的问题。

2.Shuffle Fetch

Shuffle write阶段写到各个节点的数据，Reducer端的节点通过拉取数据进而获取需要的数据，在Spark中这个叫Fetch。这就需要Shuffle Fetcher将所需的数据拉过来。这里的fetch包括本地和远端，因为shuffle数据有可能一部分存储在本地。Spark使用两套框架实现Shuffle Fetcher：NIO通过Socket连接去fetch数据；OIO通过Netty去Fetch数据，分别对应的类是BasicBlockFetcherIterator和NettyBlockFetcherIterator。

Spark的团队最终还是想用一个NIO的通信层来解决问题，但是经过性能测试，在一些特定情况下，如集群CPU核数很多地进行大规模Shuffle时，NIO性能表现不如OIO，所以Spark开发团队目前选择让二者共存。

图4-25以reduceByKey为例介绍这个算子对应的Shuffle Fetch阶段。这个Job分为两个Stage，在Stage1和Stage0之间做Shuffle Fetch的操作。HadoopRDD的每个B代表HDFS的一个分区，读入后通过映射转化为MapPartitionsRDD，做完Shuffle Write之后，Shuffle数据按照Bucket存储磁盘。Stage0的每个Task通过元数据知道数据存储在哪个节点，到该节点Fetch需要的指定Key的数据。在Stage0将Fetch到的数据形成分区，所有分区形成ShuffledRDD。通过聚集函数将ShuffledRDD每个分区中的每条数据存储到AppandOnlyMap（其本质可以理解为一个哈希表）中，在这个过程中执行用户定义的聚集函数，做聚集操作。最后将形成的结果形成分区，所有分区形成MapPartitionsRDD。

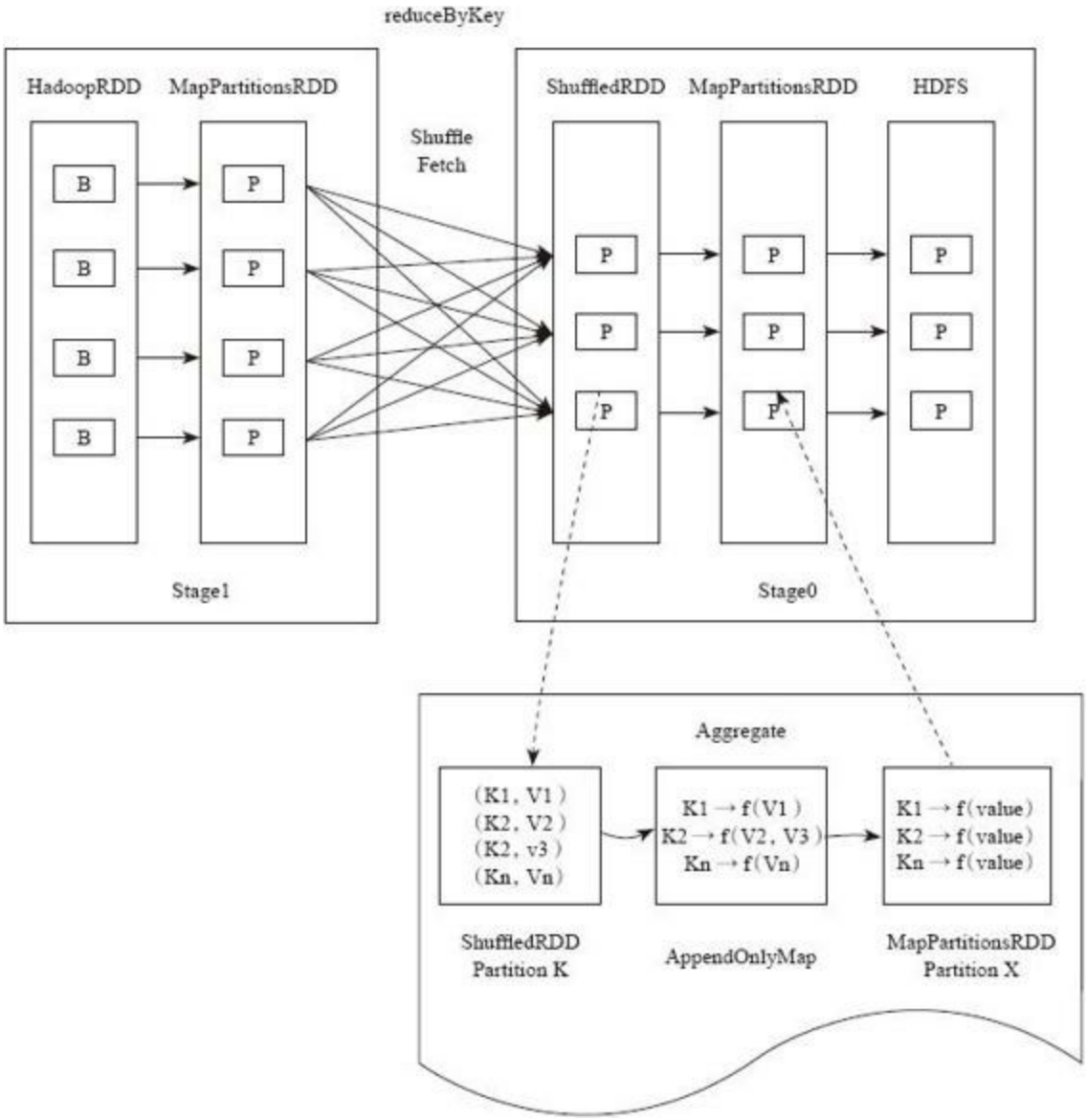

图4-25　reduceByKey的Shuffle Fetch流程

Shuffle Fetch和聚集Aggregate的操作过程是边Fetch数据边处理，而不是一次性Fetch完再处理。通过Aggregate的数据结构，AppandOnlyMap（一个Spark封装的哈希表）。Shuffle Fetch得到一条Key-Value对，直接将其放进AppandOnlyMap中。如果该HashMap已经存在相应的Key，那么直接处理用户自定义聚集函数，合并聚集数据。

3.Shuffle Aggregator

接下来介绍Aggregator（聚集）。我们都知道在Hadoop MapReduce的Shuffle过程中，Shuffle Fetch过来的数据会进行归并排序（merge sort），使得相同Key下的不同Value按序归并到一起供Reducer使用，但是Spark认为并不是所有的情况下Aggregator都需要排序，强制的排序只会增加不必要的开销。

下面介绍Spark的聚集是怎样实现的。

Spark的聚集方式分为两种：不需要外排和需要外排的。不需要外排的聚集，在内存中的AppendOnlyMap中对数据进行聚集，而需要外排的聚集，先在内存做聚集，当内存数据达到阈值时，将数据排序后写入磁盘，由于磁盘的每部分数据只是整体的部分数据，最后再将磁盘数据全部进行合并和聚集。实现上，分别采用了不同自定义容器收集聚集。Aggregator采用封装好的数据容器存储Key-Value，本质上是一个哈希表来存储。

图4-26是AppendOnlyMap不需要外排的聚集。容器本质上可以理解为一个HashMap。当要增加数据时，首先对关键字进行哈希运算查找存放位置，如果存放位置已经被占用，则通过探测方法来找下一个空闲位置。图4-26中如果插入Key1-Value3，则冲突两次，需要再哈希两次，找到新位置插入数据。

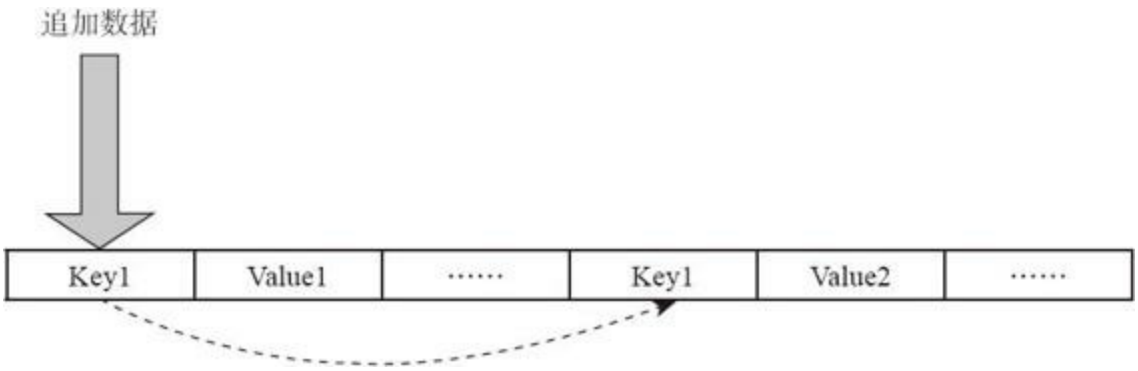

图4-26　Aggregator底层存储结构AppendOnlyMap

当进行迭代AppendOnlyMap中的元素时，从前到后扫描输出。

如果Array的利用率达到70%，就扩张一倍，并对所有Key进行再哈希后，重新排列每个Key的位置。

当用户计算count时，它会更新shuffle fetch到的每一个Key-Value对数据，插入Map中（若在Map中没有查找到，则插入其中；若查找到，则更新value值）。数据来一个处理一个，减少了不必要的排序开销。但同时需要注意，Reducer的内存必须足以存放这个分区的

所有Key和count值，因此需要Worker节点保证提供足够内存。

需要外排的聚集的原因是，如果是Reduce型的操作，则数据不断被计算合并，数据量不会暴增。考虑一下如果是groupByKey这样的操作，Reducer需要得到Key对应的所有Value。Spark需要将Key-Value全部存放在Hashmap中，并将Value合并成一个数组。为了能够存放所有数据，必须确保每一个分区足够小，内存能够容纳这个分区。因此官方建议涉及这类操作时，尽量增加分区数量，也就是增加Mapper和Reducer的数量。

增加Mapper和Reducer的数量可以减小分区的大小，使得内存可以容纳这个分区。Bucket的数量由Mapper和Reducer的数量决定，Task越多，Bucket增加得越多，由此带来Writer所需的Buffer缓存也会更多。增加Task数量，又会带来缓冲开销更大的问题，正是这个原因，Spark提供了外排方案。下面通过源码剖析内排和外排两种方式的选择逻辑。

下面代码为Aggregator类，其封装相应的聚集函数逻辑：

```
@DeveloperApi
case class Aggregator[K, V, C] (
     createCombiner: V => C,
     mergeValue: (C, V) => C,
     mergeCombiners: (C, C) => C) {
/*此处决定内存容量不足时是否采用外排的方式，而这又是通过这个参数来确定的*/
  private val externalSorting =
 SparkEnv.get.conf.getBoolean("spark.shuffle.spill", true)
……
  def combineValuesByKey(iter: Iterator[_ <: Product2[K, V]],
         context: TaskContext): Iterator[(K, C)] = {
    if (!externalSorting) {
/*此处使用了内部优化的数据结构combiners存储了combiner的集合，每个combiner代表一个Key和对应Key的元素Seq*/
      val combiners = new AppendOnlyMap[K,C]
      var kv: Product2[K, V] = null
      val update = (hadValue: Boolean, oldValue: C) => {
/*看处理的是否是第一个元素，如果是，则需要创建集合结构，如果不是第一个，则插入原来创建的结构中*/
        if (hadValue) mergeValue(oldValue, kv._2) else createCombiner(kv._2)
      }
      while (iter.hasNext) {
        kv = iter.next()
/*如果不采用外排，则调用AppendOnlyMap的聚集数据结构进行存储，有兴趣可以看具体的实现*/
        combiners.changeValue(kv._1, update)
      }
      combiners.iterator
    } else {
      val combiners = new ExternalAppendOnlyMap[K, V, C](createCombiner, mergeValue, mergeCombiners)
      while (iter.hasNext) {
        val (k, v) = iter.next()
/*如果采用外排，则使用ExternalAppendOnlyMap这个Spark定义的数据结构存储聚集数据*/
       combiners.insert(k, v)
      }
    }
  }
```

[1] 注意：在Spark中没有明确限定是Map过程，还是Reduce过程，由于整个逻辑过程在Hadoop已经成为事实公认的标准，可以迁移原有知识理解。

4.7 本章小结

本章介绍了Spark的内部运行机制。主要介绍了Spark的执行机制和调度机制，包括调度与任务分配机制、I/O机制、通信机制、容错机制和Shuffle机制。Spark在执行过程中由Driver控制应用生命周期。调度中，Spark采用了经典的FIFO和FAIR等调度算法对内部的资源实现不同级别的调度。在Spark的I/O中，将数据抽象以块为单位进行管理，RDD中的一个分区就是需要处理的一个块。集群中的通信对于命令和状态的传递极为重要，Spark通过AKKA框架进行集群消息通信。Spark通过Lineage和Checkpoint机制进行容错性保证，Lineage进行重算操作，Checkpoint进行数据冗余备份。最后介绍了Spark中的Shuffle机制，Spark也借鉴了MapReduce模型，但是其Shuffle机制进行了创新与优化。通过阅读本章，读者可以深入了解Spark的内部原理，这对上层应用开发与性能调优是十分重要的。

介绍完Spark内部的执行执行机制，相信读者已经跃跃欲试，希望开发自己的Spark程序，下面章节将引导读者配置Spark开发环境，然后介绍Spark的编程实战。

第5章　Spark开发环境配置及流程

通过前面的介绍，相信读者已经对Spark的内部机制有了一定的了解，本章将介绍如何在Spark中开发应用程序，以及如何进行程序的编译和调试。在编写Spark应用程序之前，需要安装和配置开发环境，一般可以选择Intellij或Eclipse进行开发和调试，使用SBT编译项目。

5.1 Spark应用开发环境配置

Spark的开发可以通过Intellij或者Eclipse IDE进行，在环境配置的开始阶段，还需要安装相应的Scala插件。

5.1.1 使用Intellij开发Spark程序

下面介绍如何使用Intellij IDEA构建Spark开发环境和源码阅读环境。由于Intellij对Scala的支持更好，所以目前Spark开发团队使用Intellij作为开发环境。

1.配置开发环境

（1）安装JDK

用户可以自行安装JDK6、JDK7。官网地址为：http://www.oracle.com/technetwork/java/javase/downloads/index.html

下载后，如果在Windows下直接运行安装程序，则自动配置环境变量，安装成功后，在CMD的命令行下输入Java，如有Java版本的日志信息提示，则证明安装成功。

如果在Linux下安装，下载JDK包解压缩后，还需要配置环境变量。

在/etc/profile文件中，配置环境变量这样程序就能找到JDK的安装路经：

```
export JAVA_HOME=/usr/java/jdk1.6.0_27
export JAVA_BIN=/usr/java/jdk1.6.0_27/bin
export PATH=$PATH：$JAVA_HOME/bin
export CLASSPATH=.：$JAVA_HOME/lib/dt.jar：$JAVA_HOME/lib/tools.jar
export JAVA_HOME JAVA_BIN PATH CLASSPATH
```

（2）安装Scala

Spark对Scala的版本有约束，用户可以在Spark的官方下载界面看到相应的Scala版本号。下载指定的Scala包，官网地址为：http://www.scala-lang.org/download/。

（3）安装Intellij IDEA

用户可以下载安装最新版本的Intellij，官网地址为：http://www.jetbrains.com/idea/download/。

目前Intellij最新的版本中已经可以支持新建sbt工程，安装Scala插件可以很好地支持Scala开发。

（4）在Intellij中安装Scala插件

在Intellij菜单中选择“Configure”→“Plugins”→“Browse repositories”命令，在弹出的界面中输入“Scala”搜索插件（见图5-1），然后点击相应安装按钮进行安装，重启Intellij使配置生效。

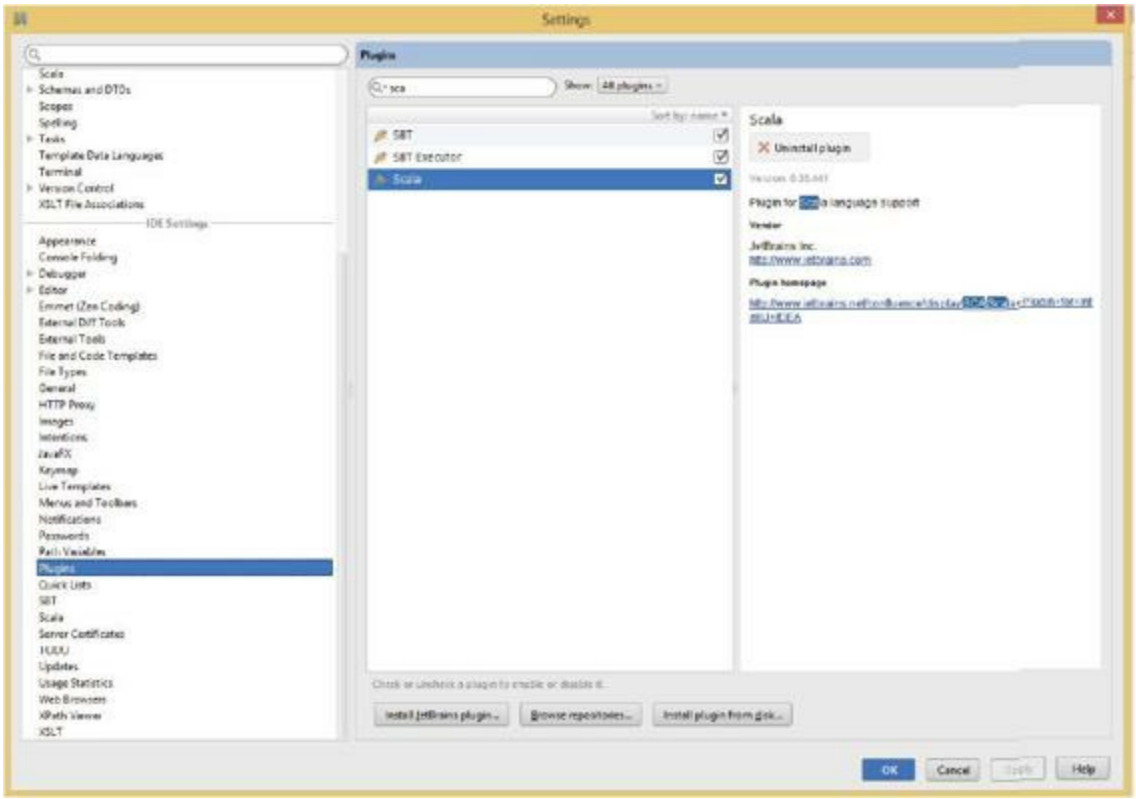

图5-1　输入“Scala”搜索插件

2.配置Spark应用开发环境

1）在Intellij IDEA中创建Scala Project，名称为SparkTest。

2）选择菜单中的“File”→“project structure”→“Libraries”，然后选择“+”，导入spark-assembly_2.10-1.0.0-incubating-hadoop2.2.0.jar。

只需导入上述Jar包即可，该包可以通过sbt/sbt assembly命令生成，这个命令相当于将

Spark的所有依赖包和Spark源码打包为一个整体。

在assembly/target/scala-2.10.4/目录下生成spark-assembly-1.0.0-incubating-hadoop2.2.0.jar。

3）如果IDE无法识别Scala库，则需要以同样方式将Scala库的jar包导入，之后可以开始开发Scala程序，如图5-2所示。本例将Spark默认的示例程序SparkPi复制进文件。

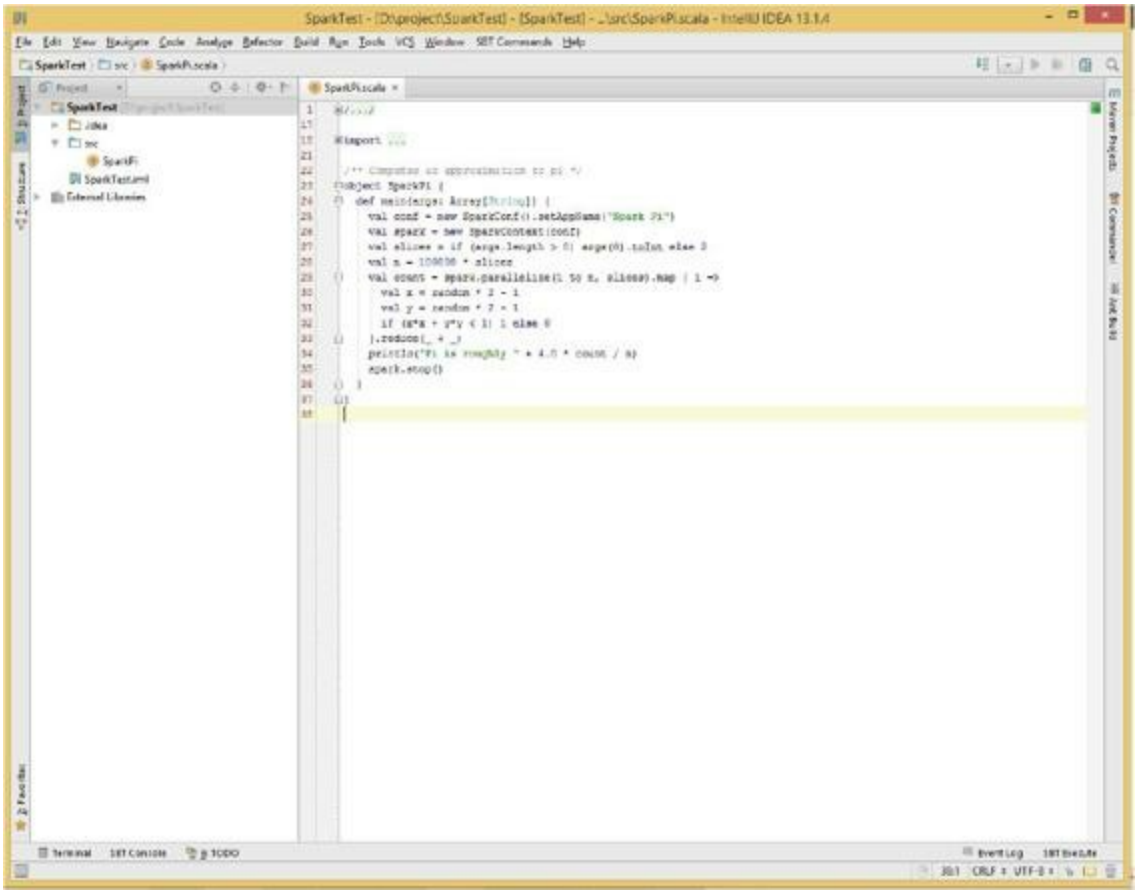

图5-2　编写程序

3.运行Spark程序

（1）本地运行

编写完Scala程序后，可以直接在Intellij中以本地（local）模式运行（见图5-3），方法如下。

注意，设置Program arguments中参数为local。

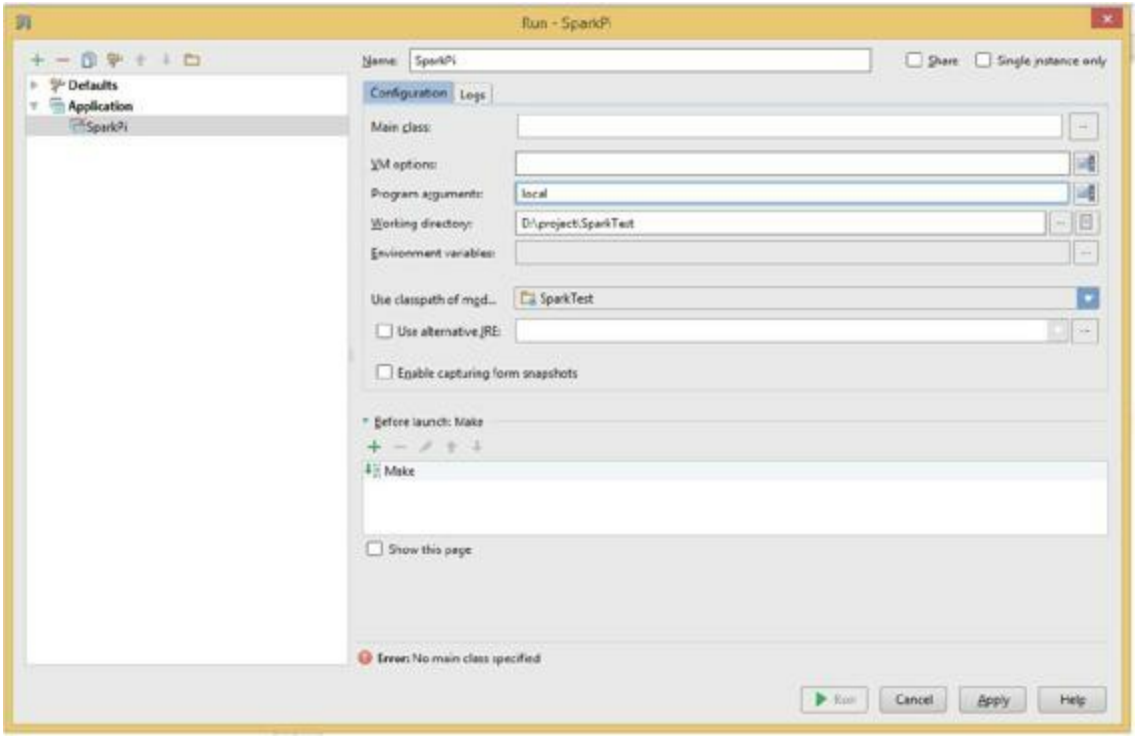

图5-3　以local模式运行

在Intellij中点击Run/Debug Configuration按钮，在其下拉列表选择Edit Configurations选项。在Run输入选择界面中，如图5-3所示，在输入框Program arguments中输入main函数的输入参数local，即为本地单机执行Spark应用。然后右键选择需要运行的类，点击Run运行Spark应用程序。

（2）在集群上运行Spark应用Jar包

如果想把程序打成Jar包，通过命令行的形式在Spark集群中运行，可以按照以下步骤操作。

1）选择“File”→“Project Structure”命令，然后选择“Artifact”，单击“+”按钮，选择“Jar”→“From Modules with dependencies”，如图5-4所示。

选择Main函数，在弹出的对话框中选择输出Jar位置，并单击“OK”按钮。

在图5-4中点击From mudules with dependencies后将会出现如图5-5所示的输入框，在其中的输入框中选择需要执行的Main函数。

在图5-5所示的界面中单击OK按钮后，在图5-6所示的对话框中通过OutPut layout中的“+”选择依赖的Jar包。

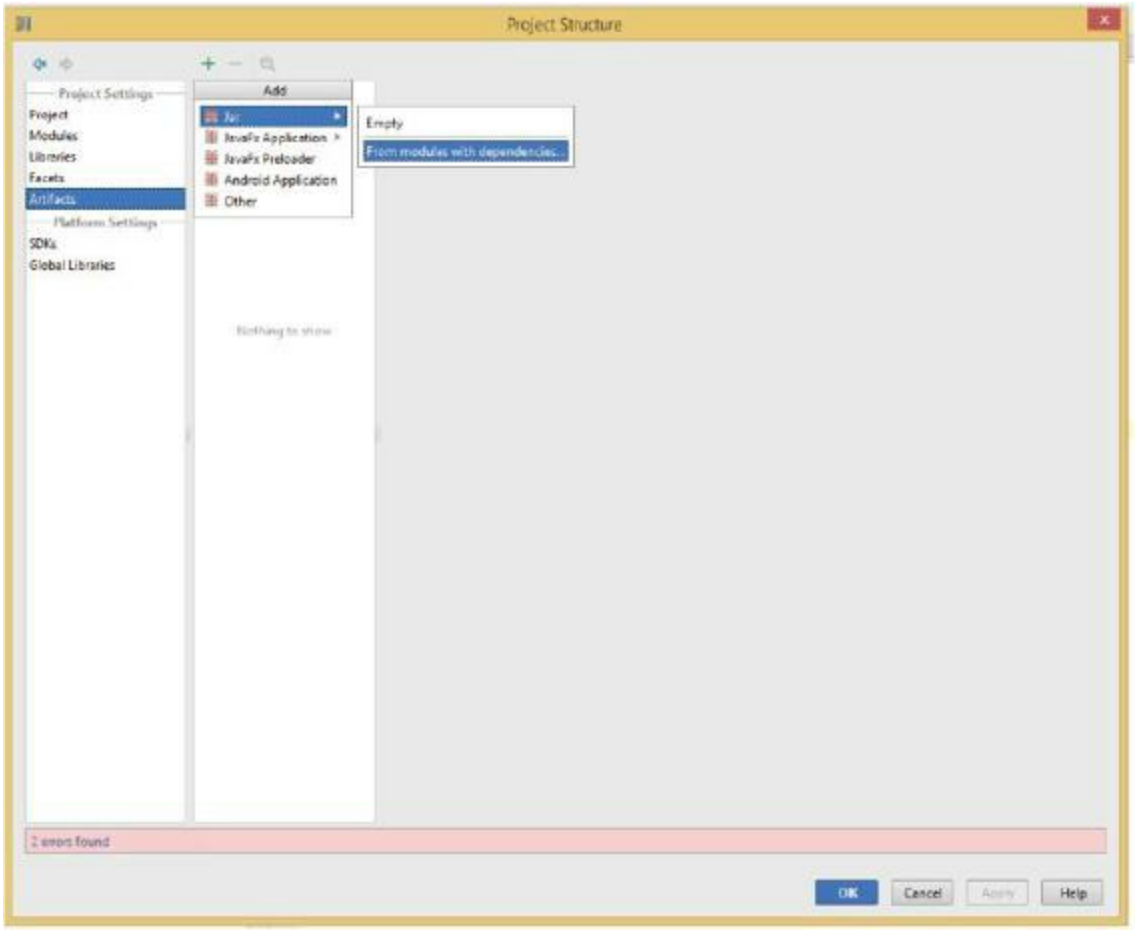

图5-4　生成Jar包第一步

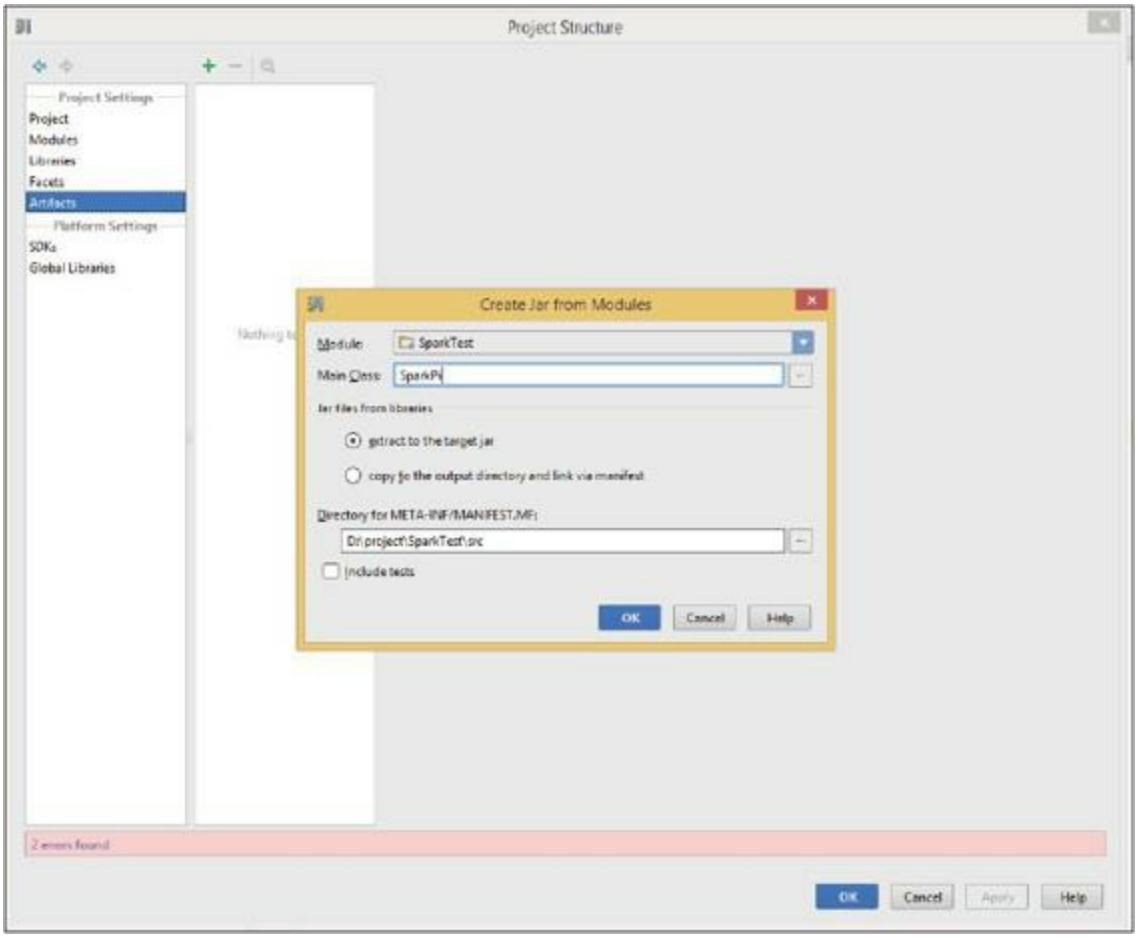

图5-5　生成Jar包第二步

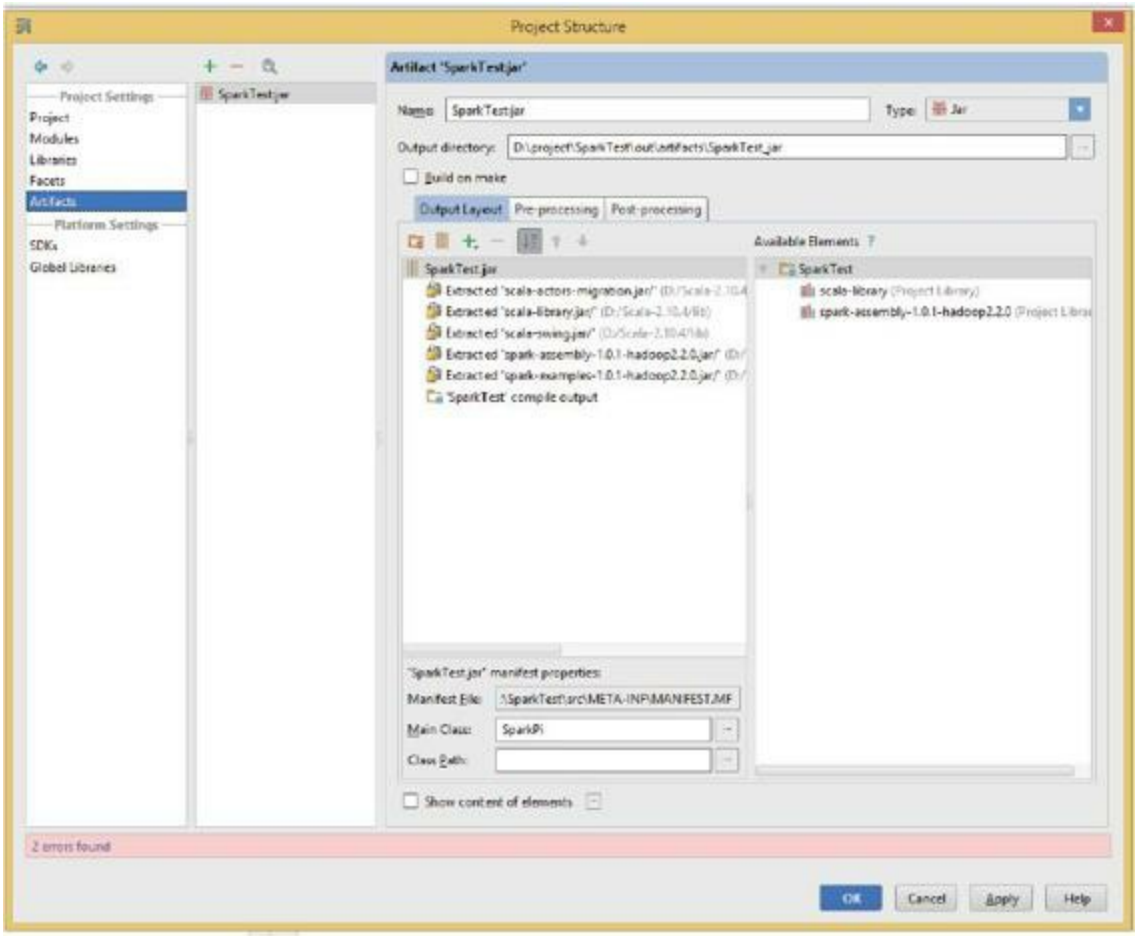

图5-6　生成Jar包第三步

2）在主菜单中选择“Build”→“Build Artifact”命令，编译生成Jar包。

3）在集群的主节点，通过下面命令执行生成的Jar包SparkTest.jar。

```
java -jar SparkTest.jar
```

5.1.2 使用Eclipse开发Spark程序

下面介绍如何使用Eclipse配置和开发Spark的环境，用户可以在Windows或者Linux环境下使用Eclipse进行开发。

1.环境配置

与Intellij配置环境一样，需要用户下载安装JDK和Scala。前文已详细介绍，这里不再赘述。

1）下载Eclipse Scala IDE插件，官网地址为http://scala-ide.org/download/sdk.html，可在官网中自行下载安装。

2）下载Eclipse[1]，官网地址为http://www.eclipse.org/downloads/。

2.安装Scala插件

1）将Eclipse Scala IDE插件中的features和plugins两个目录下的所有文件复制到Eclipse解压后对应的根目录中。重启Eclipse，单击Eclipse右上角方框按钮，如图5-7所示，在弹出的Open Perspective对话框中查看是否有“Scala”一项，如果有则直接单击打开。

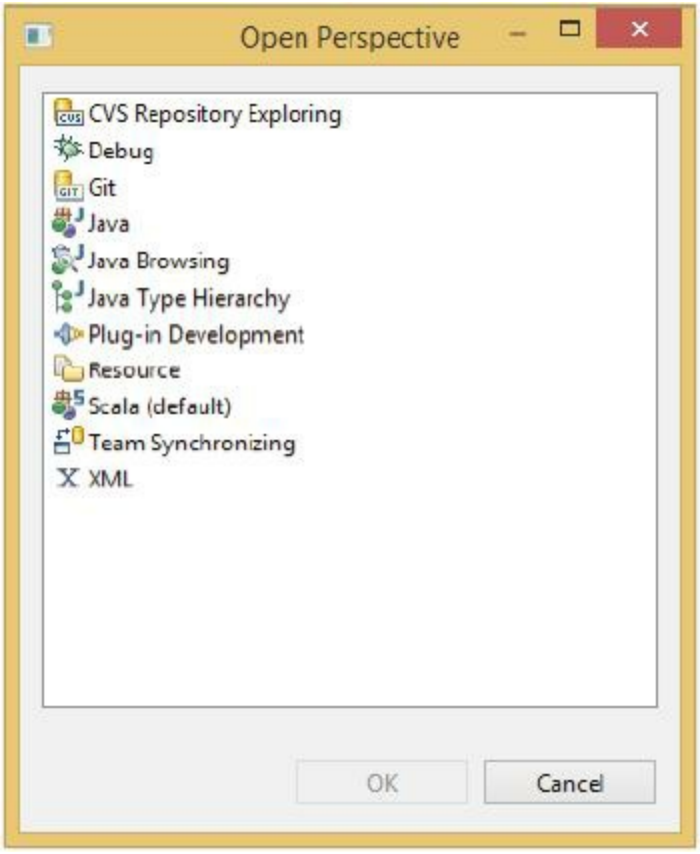

图5-7　安装插件

2）在Eclipse中，选择Help按钮，然后点击Install New Software命令，在打开的输入框里填入http://download.scala-ide.org/sdk/e38/scala29/stable/site，并按回车键，可看到以下内容（见图5-8中加框突出部分），选择Scala IDE for Eclipse和Scala IDE for Eclipse development support两项进行安装即可，如图5-8所示。

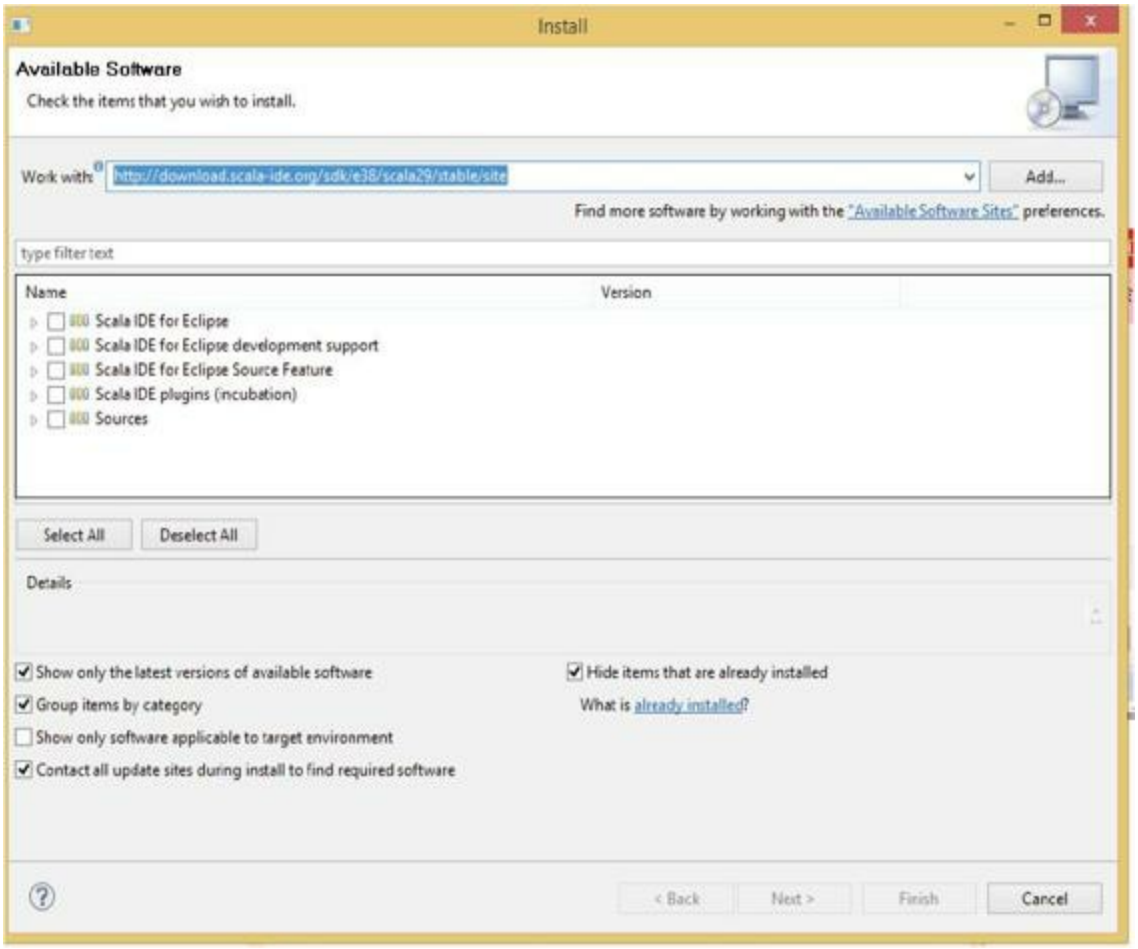

图5-8　设置安装选项

3）直接下载Scala IDE，可以在官网http://scala-ide.org/下载。现在的ScalaIDE中默认自带了Eclipse，用户可以直接使用。

3.开发Spark程序

1）在安装好Scala插件的Eclipse中，选择File→New→Other命令，在弹出的New窗口中选择Scala Wizard→Scala Project命令，创建Scala项目，如图5-9所示。

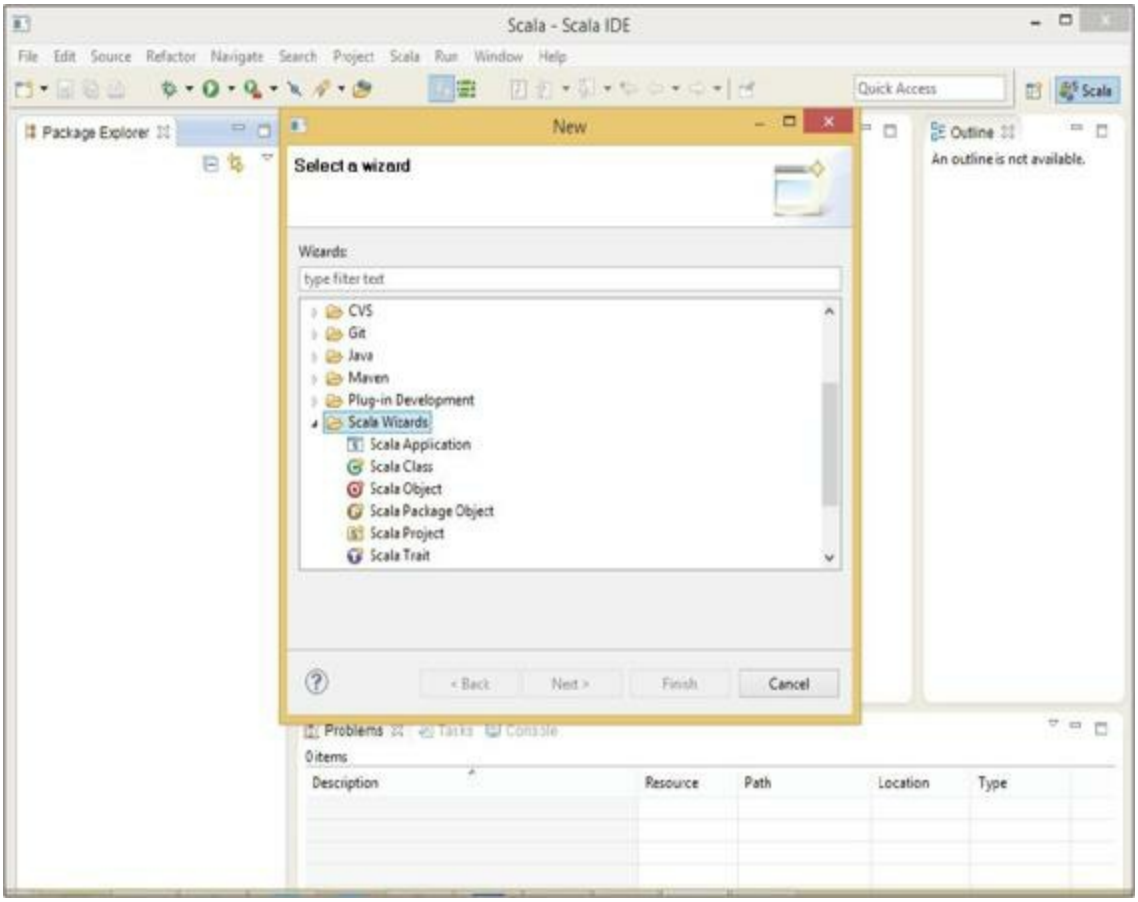

图5-9　创建Scala项目

2）右击新建工程，在快捷菜单中选择Properties命令，在弹出的窗口（见图5-10）中依次选择Java Build Path→Libraties→Add External JARs即可，导入assembly/target/scala-2.9.3/目录下的spark-assembly-1.0.0-incubating-hadoop2.2.0.jar（这个包可以通过sbt/sbt assembly生成，也可以在预编译版本的Spark中找到）。

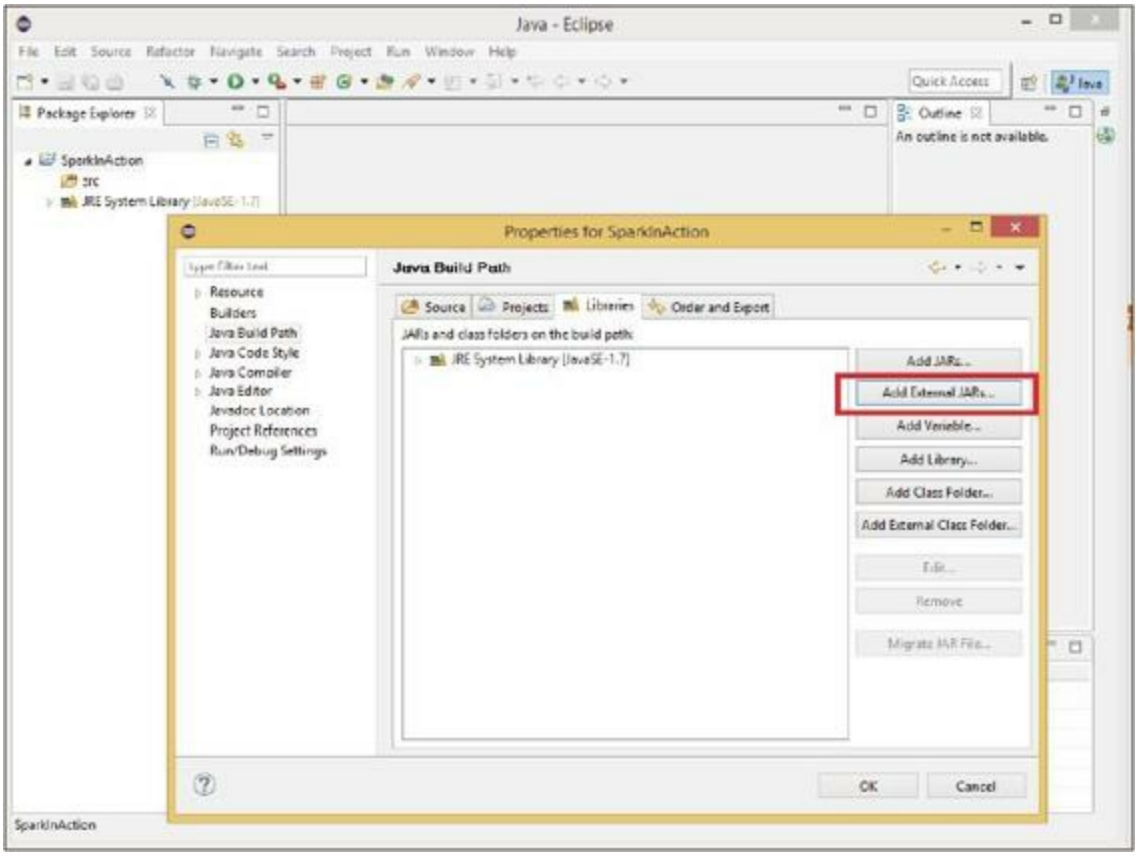

图5-10　增加外部Jar

3）在工程中创建一个Scala对象（Object），命名为WordCount，在Name后的输入框填入WordCount，如图5-11所示。

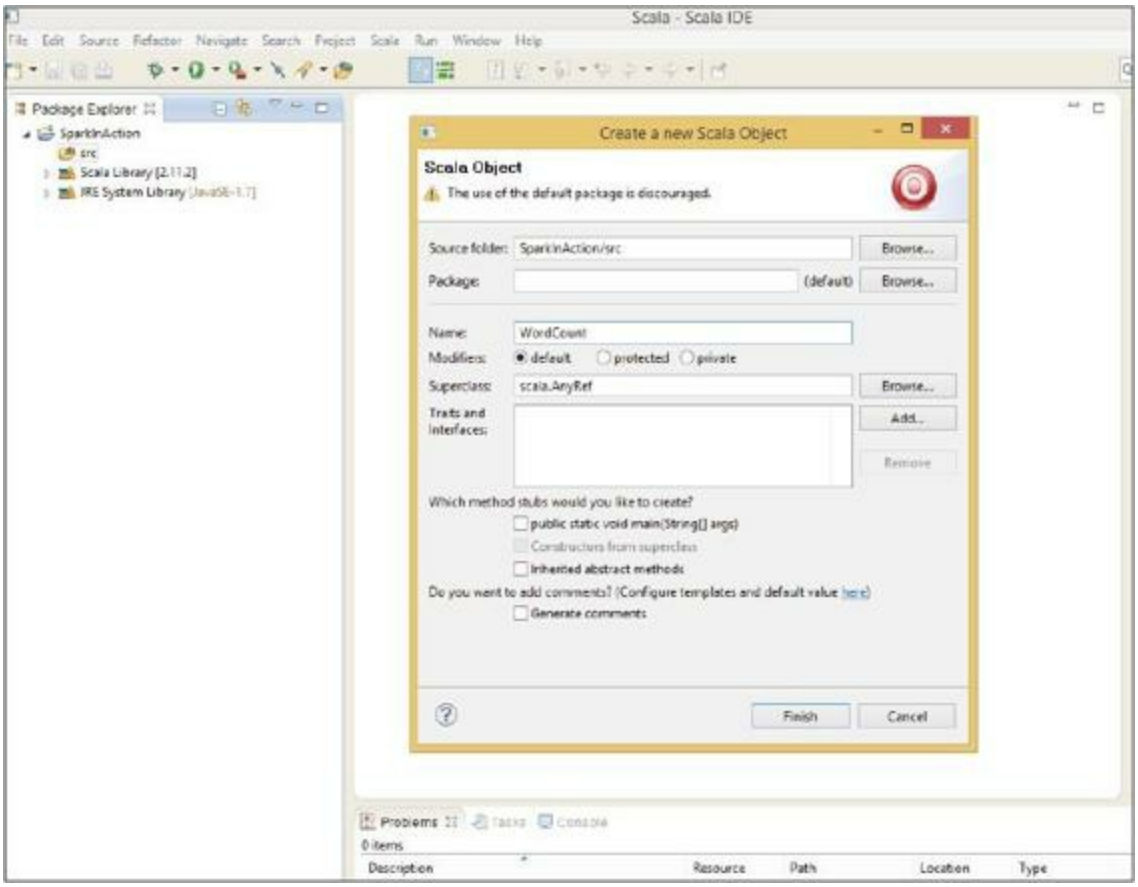

图5-11　创建Scala类

WordCount是一个测试程序，统计输入的词频，读者可以参考第6章来了解。

在SparkTest工程中，右击WordCount.scala，在弹出的快捷菜单中选择Export命令，然后在弹出的窗口中选择Java→JAR File命令，将文件命名为WordCount。最后生成WordCound.jar的可执行Jar包。

或者直接在SparkContext中将第一个参数配置为local，然后直接在Eclipse点击run按钮，本地运行程序。

 提示　Java语言开发Spark程序。

将Spark开发程序包spark-assembly-1.0.0-incubating-hadoop2.2.0.jar作为第三方依赖库。由于Scala也是运行在JVM之上，并且可以和Java合编程，所以可以按原有方式开发Java程序并调用Spark中的API。

[1] 下载的Eclipse版本一定要与Eclipse Scala IDE插件版本一致。

5.1.3　使用SBT构建Spark程序

用户也可以直接使用SBT构建Spark应用。在这个应用中，以统计包含“Hello”字符的行数为案例。

（1）构建开发环境

1）下载并解压Spark 1.0.0程序包或者通过git clone https://github.com/apache/spark命令将项目克隆到Spark根目录。

2）运行sbt/sbt assembly构建项目。

3）为了使用SBT成功构建Spark，预先安装SBT。

（2）开发应用程序

在构建Spark以后，就可以开始开发应用了。

1）创建目录mkdir HelloWorld。

2）创建一个.sbt文件，在目录HelloWorld中创建simple.sbt文件。

3）在.sbt文件中加入如下配置项配置应用名，版本和依赖等信息。

```
name ：= "HelloWorld Project"
version ：= "1.0"
scalaVersion ：= "2.10.3"
libraryDependencies += "org.apache.spark" %% "spark-core" % "0.9.1"
resolvers += "Akka Repository" at "http：//repo.akka.io/releases/
```

（3）创建代码文件

```
HelloWorld/src/main/scala/HelloWorld.scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object HelloWorld {
 def main（args： Array[String]） {
```

```
 val logFile = "src/data/sample.txt"
 val sc = new SparkContext("local", "Simple App", "/path/to/spark-1.0.0-incubating",
 List("target/scala-2.10/simple-project_2.10-1.0.jar"))
 val logData = sc.textFile(logFile, 2).cache()
 val numHello = logData.filter(line => line.contains("Hello")).count()
 println("Lines with the: %s".format(numHello))
 }
}
```

（4）运行程序

1）程序创建后，返回到HelloWorld根目录。

2）运行sbt package进行构建与打包。

3）运行sbt run，系统执行构建好的程序。

5.1.4 使用Spark Shell开发运行Spark程序

因为运行Spark Shell时，会默认创建一个SparkContext，命名为sc，所以不需要在Spark Shell创建新的SparkContext。在运行Spark Shell之前，可以设定参数MASTER指定Spark应用提交MASTER指向的相应集群或者本地模式执行。可以通过参数ADD_JARS将JARS添加到classpath中。

如果希望spakr-shell在本地通过4核的CPU运行，需要以如下方式启动。

```
$MASTER=local[4] ./spark-shell
```

这里的4是指启动4个工作线程。

如果要添加JARS，可以用如下方法实现：

```
$MASTER=local[4]  ADD_JARS=code.jar ./spark-shell
```

在Spark Shell中，输入下面代码，读取dir文件，以输出文件中有多少数据项。

```
scala>val text=sc.textFile("dir")
scala>text.count
```

按回车键，即可运行Shell中的程序。

5.2 远程调试Spark程序

本地调试Spark程序和传统的调试单机的Java程序基本一致，读者可以参照原来的方式调试，关于单机调试本书暂不介绍。对于远程调试服务器上的Spark代码，首先确保在服务器和本地的Spark版本一致。需按前文介绍的方法预先安装好JDK和git。

1.编译Spark

在服务器端和本地计算机下载Spark项目。

通过下面命令克隆一份Spark源码。

```
git clone https：//github.com/apache/spark
```

然后针对指定的Hadoop版本进行编译。配置代码如下：

```
SPARK_HADOOP_VERSION=2.3.0 sbt/sbt assembly
```

2.在服务器端的配置

1）根据相应的Spark配置指定版本的Hadoop并启动Hadoop。

2）对编译好的Spark进行配置，在conf/spark-env.sh文件中进行如下配置。下面代码配置了Spark调试所需的Java参数。

```
export SPARK_JAVA_OPTS="  -agentlib：jdwp=transport=dt_socket，server=y，suspend=y，address=9999"
```

其中suspend=y表示设置为需要挂起的模式。这样，当启动Spark的作业时，程序会自动挂起，等待本地的IDE附加（attach）到被调试的应用程序上。address后接的是开放等待连接的端口号。

3.启动Spark集群和应用程序

1）启动Spark集群。

```
./sbin/start-all.sh
```

2）启动需要调试的程序，以Spark中自带的HdfsWordCount为例。

```
MASTER=spark：//10.10.1.168：7077 ./bin/run-example org.apache.spark.examples.streaming.HdfsWordCount
hdfs：//localhost：9000/test/test.txt
```

执行后程序挂起，并等待本地的Intellij进行连接，如图5-12所示。

图5-12　远程调试

4.配置本地IDE

配置并连接服务器端挂起的程序。

在Intellij中点击run→edit configuration，在弹出的Run/Debug Configuration界面中选择remote，在默认配置中将端口号Port设置为9999，将主机IP地址的Host改为服务器的地址10.10.1.168，同时用选择Debugger mode为Attach（附加）方式，如图5-13所示。

选择附加方式后，在程序中设置断点即可进行调试。

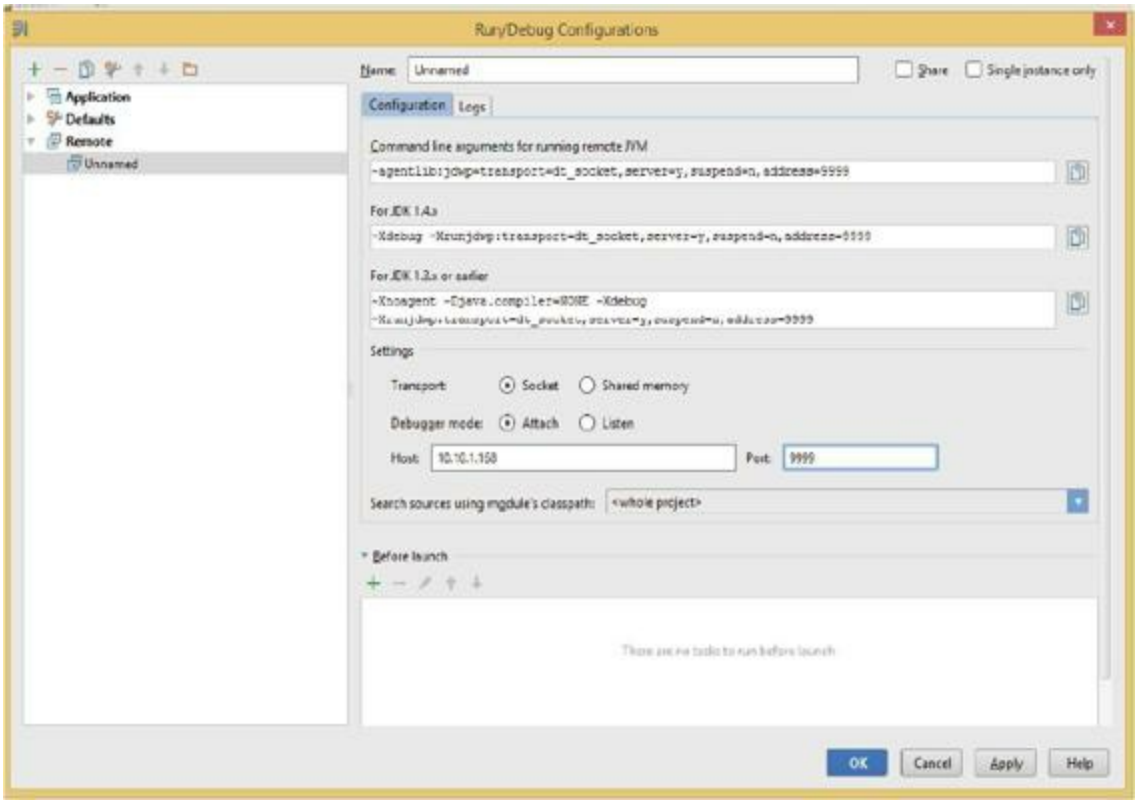
Run/Debug Configurations
Application
Defaults
Remote
Unnamed
Name: Unnamed
Share
Single instance only
Configuration
Logs
Command line arguments for running remote JVM
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=9999
For JDK 1.4.x
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=9999
For JDK 1.2.x or earlier
-Xnoagent -Djava.compiler=NONE -Xdebug
Settings
Transport: Socket
Shared memory
Debugger mode: Attach
Listen
Host: 10.10.1.158
Port: 9999
Search sources using module's classpath:
<whole project>
Before launch
There are no tasks to run before launch
OK
Cancel
Apply
Help

图5-13　远程调试设置

5.3 Spark编译

用户可以通过Spark的默认构建工具SBT编译和打包源码。当用户需要对源码进行二次开发时，需要对源码进行增量编译。通过下面的方式可以实现编译和增量编译。

1.克隆Spark源码

命令如下：

```
git clone https：//github.com/apache/spark
```

这样从Github将Spark源码下载到本地，建立本地的仓库，如图5-14所示。

```
[                  source]$ git clone https://github.com/apache/spark
Initialized empty Git repository in /home/hucheng/yanjie/source/spark/.git/
remote: Counting objects: 119763, done.
remote: Compressing objects: 100% (50/50), done.
remote: Total 119763 (delta 16), reused 64 (delta 16)
Receiving objects: 100% (119763/119763), 73.20 MiB | 3.20 MiB/s, done.
Resolving deltas: 100% (54923/54923), done.
```

图5-14　git clone Spark库

2.编译Spark

在Spark项目的根目录内执行编译和打包命令，命令如下：

```
sbt/sbt assembly
```

如图5-15所示，执行过程中会解析依赖和下载需要的依赖Jar包。执行完成后，将所有Jar包打包为一个Jar包，即可运行Spark集群和示例了。

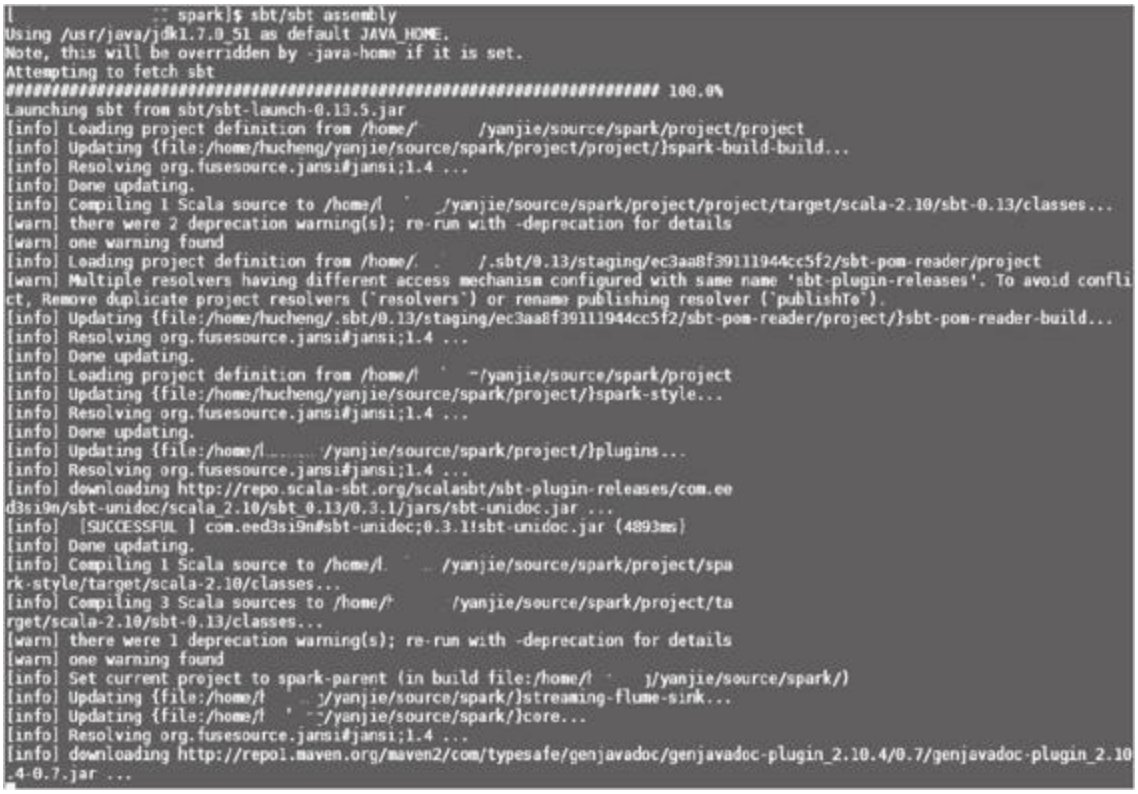

图5-15　编译Spark源码

3.增量编译

在有些情况下，用户需要修改源码，修改之后如果每次都重新下载Jar包或者重新编译一遍全部源码，就会很浪费时间，用户可以通过下面的增量编译方法，只编译改变的源码。

1）编译打包一个assembly的Jar包。命令如下。

```
$ sbt/sbt clean assembly
```

2）这时的Spark程序已经可以运行。用户可以进入Spark Shell执行程序。命令如下：

```
$ ./bin/spark-shell
```

3）配置export SPARK_PREPEND_CLASSES参数为true，开启增量编译模式。命令如下：

```
$ export SPARK_PREPEND_CLASSES=true
```

4）继续使用Spark Shell中的程序。通过下面命令进入Spark Shell。

```
$ ./bin/spark-shell
```

5）这时可以对代码进行修改和二次开发。

```
/*
开发……
*/
```

6）编译Spark源码。

```
$ sbt/sbt compile
/*
开发……
*/
$ sbt/sbt compile
```

7）解除增量编译模式。

```
$ unset SPARK_PREPEND_CLASSES
```

8）返回正常使用Spark Shell的情景。通过下面命令进入Spark Shell

```
$ ./bin/spark-shell # Back to normal, using Spark classes from the assembly jar
```

9）如果不想每次都开启一个新的sbt会话，就可以在compile命令前加上~。命令如下：

```
$ sbt/sbt ~compile
```

4.查看Spark源码依赖图

使用SBT查看依赖图，需要运行下面的命令，结果如图5-16所示。

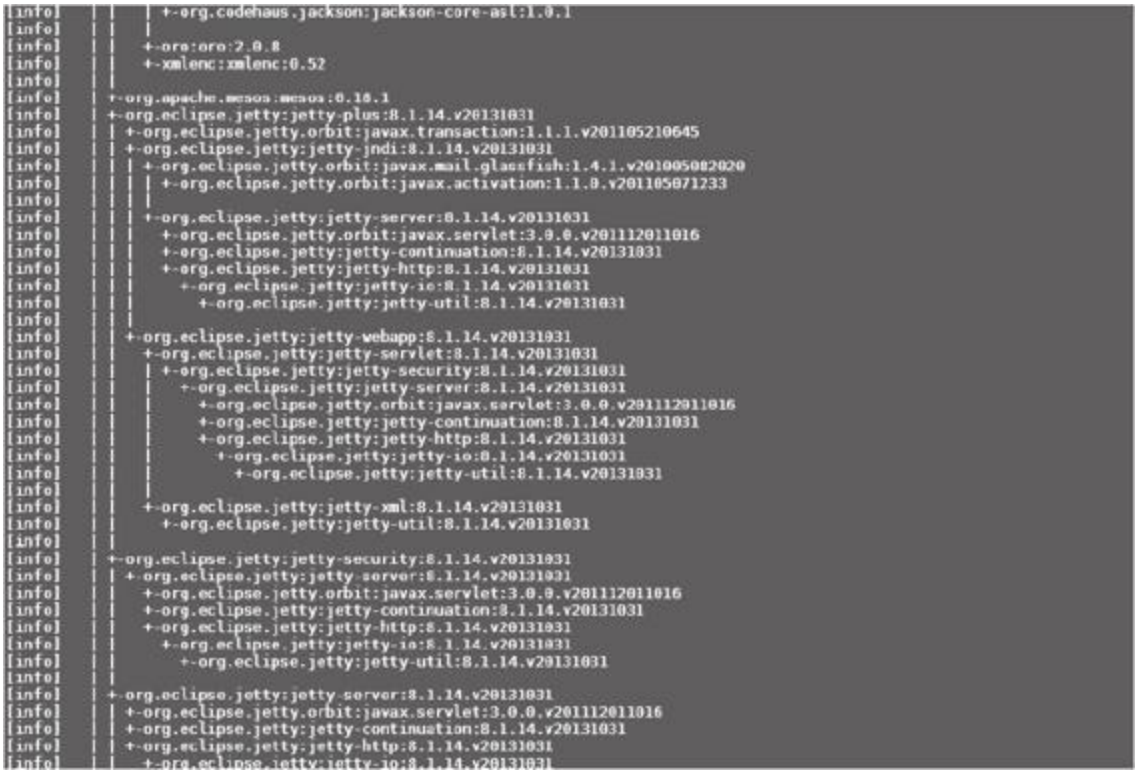

图5-16　查看依赖

```
$ # sbt
$ sbt/sbt dependency-tree
```

使用Maven查看依赖图，需要运行下面的命令。

```
$ # Maven
$ mvn -DskipTests install
$ mvn dependency：tree
```

5.4 配置Spark源码阅读环境

由于Spark使用SBT作为项目管理构建工具，SBT的配置文件中配置了依赖的jar包网络路径，在编译或者生成指定类型项目时，需要从网络下载jar包，需要预先安装Git。在Linux操作系统或者Windows操作系统中（可以下载Git Shell，在Git Shell中进行命令行操作）通过sbt/sbt gen-idea命令，生成Intellij项目文件，然后在Intellij IDEA中直接通过“Open Project”选项打开项目。

1）克隆Spark源码。命令如下：

```
git clone https：//github.com/apache/spark
```

2）生成项目。

3）在所需的软件安装好后，在Spark源代码根目录下输入以下代码。

```
sbt/sbt gen-idea
```

这样SBT会自动下载依赖包和编译源文件编译，并生成Intellij所需的项目文件。

如果在Eclipse下，则需要运行如下代码：

```
sbt/sbt gen-eclipse
```

这样SBT会自动下载依赖包和编译源文件编译，并生成Eclipse所需的项目文件。用户就可以在IDE中查看源码了。

5.5 本章小结

本章主要介绍了Spark应用程序的开发流程以及如何编译和调试Spark程序。用户可以选用能够很好支持Scala项目的Intellij IDE。如果之前经常使用Eclipse开发Java程序，也可以在Eclipse中安装Scala IDE插件，开发与调试Spark程序。由于Spark项目基于SBT构建，用户可以创建SBT项目，开发应用。在应用的开发过程中，需要进行调试诊断问题。在本章最后部分介绍的远程调试方法可以很好地帮助用户调试Spark程序。

通过本章的介绍，读者可以搭建Spark开发环境，下面将通过Spark编程实战进入Spark程序的开发之旅。

第6章　Spark编程实战

前面章节已经介绍了很多关于Spark的基础知识、运行机制，Spark的集群安装与配置方法，Spark内核是由Scala语言开发的，当然用户也可以使用Java和Python进行开发。本章将从题目、程序和应用出发，介绍如何使用Spark进行程序开发，以及如何编写Spark程序。

6.1 WordCount

WordCount是大数据领域的经典范例，如同程序设计中的Hello World一样，是一个入门程序。本节主要从并行处理的角度出发，介绍设计Spark程序的过程。

1.实例描述

输入：

```
Hello World Bye World
Hello Hadoop Bye Hadoop
Bye Hadoop Hello Hadoop
```

输出：

```
<Bye , 3>
<Hadoop , 4>
<Hello , 3>
<World , 2>
```

2.设计思路

在map阶段会将数据映射为：

```
<Hello , 1>
<World , 1>
<Bye , 1>
<World , 1>
<Hello , 1>
<Hadoop , 1>
<Bye , 1>
<Hadoop , 1>
<Bye , 1>
<Hadoop , 1>
<Hello , 1>
<Hadoop , 1>
```

在reduceByKey阶段会将相同key的数据合并，并将合并结果相加。

```
<Bye , 1 , 1 , 1>
<Hadoop , 1 , 1 , 1 , 1>
<Hello , 1 , 1 , 1>
<World , 1 , 1>
```

3.代码示例

WordCount的主要功能是统计输入中所有单词出现的总次数，编写步骤如下。

（1）初始化

创建一个SparkContext对象，该对象有4个参数：Spark master位置、应用程序名称、Spark安装目录和Jar存放位置。

需要引入下面两个文件。

```
import org.apache.spark._
import SparkContext._
val sc = new SparkContext(args(0), "WordCount",
   System.getenv("SPARK_HOME"),
   Seq(System.getenv("SPARK_TEST_JAR")))
```

（2）加载输入数据

从HDFS上读取文本数据，可以使用SparkContext中的textFile函数将输入文件转换为一个RDD，该函数采用Hadoop中的TextInputFormat解析输入数据。

```
val textRDD = sc.textFile(args(1))
```

textFile中的每个Hadoop Block相当于一个RDD分区。

（3）词频统计

对于WordCount而言，首先需要从输入数据中的每行字符串中解析出单词，然后分而治之，将相同单词放到一个组中，统计每个组中每个单词出现的频率。

```
val result = textRDD.flatMap{
      case(key, value)  => value.toString().split("\\s+");
}.map(word => (word, 1)).reduceByKey(_ + _)
```

其中，flatMap函数每条记录转换，转换后，如果每个记录是一个集合；则将集合中的元素变为RDD中的记录；map函数将一条记录映射为另一条记录；reduceByKey函数将key相同的关键字的数据聚合到一起进行函数运算。

（4）存储结果

可以使用SparkContext中的saveAsTextFile函数将数据集保存到HDFS目录下。

```
result.saveAsSequenceFile(args(2))
```

4.应用场景

WordCount的模型可以在很多场景中使用，如统计过去一年中访客的浏览量、最近一段时间相同查询的数量和海量文本中的词频等。

6.2 Top K

Top K算法有两步，一是统计词频，二是找出词频最高的前K个词。

1.实例描述

假设取Top 1，则有如下输入和输出。

输入：

```
Hello World Bye World
Hello Hadoop Bye Hadoop
Bye Hadoop Hello Hadoop
```

输出：

```
词Hadoop 词频4
```

2.设计思路

首先统计WordCount的词频，将数据转化为（词，词频）的数据对，第二个阶段采用分治的思想，求出RDD每个分区的Top K，最后将每个分区的Top K结果合并以产生新的集合，在集合中统计出Top K的结果。每个分区由于存储在单机的，所以可以采用单机求Top K的方式。本例采用堆的方式。也可以直接维护一个含K个元素的数组，感兴趣的读者可以参考其他资料了解堆的实现。

3.代码示例

Top K算法示例代码如下：

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object TopK {
  def main(args:Array[String]) {
    /*执行WordCount，统计出最高频的词*/
    val spark = new SparkContext("local", "TopK",
```

```
      System.getenv("SPARK_HOME"), SparkContext.jarOfClass(this.getClass))
    val count = spark.textFile("data").flatMap(line =>
      line.split(" ")).map(word =>
      (word, 1)).reduceByKey(_ + _)
    /*统计RDD每个分区内的Top K查询*/
    val topk = count.mapPartitions(iter => {
      while(iter.hasNext) {
        putToHeap(iter.next())
      }
      getHeap().iterator
    }
    ).collect()
    /*将每个分区内统计出的TopK查询合并为一个新的集合，统计出TopK查询*/
    val iter = topk.iterator
    while(iter.hasNext) {
      putToHeap(iter.next())
    }
    val outiter=getHeap().iterator
    /*输出TopK的值*/
    println("Topk 值 :")
    while(outiter.hasNext) {
     println("\n 词频:"+outiter.next()._1+"  词:"+outiter.next()._2)
    }
    spark.stop()
  }
}
def putToHeap(iter : (String, Int)) {
  /*数据加入含k个元素的堆中*/
……
}
def getHeap() : Array[(String, Int)] = {
  /*获取含k个元素的堆中的元素*/
  val a=new Array[(String, Int)]()
……
}
```

4.应用场景

Top K的示例模型可以应用在求过去一段时间消费次数最多的消费者、访问最频繁的IP地址和最近、更新、最频繁的微博等应用场景。

6.3 中位数

海量数据中通常有统计集合中位数的计算需求，读者可以通过以下示例了解Spark求中位数的方式。

1.实例描述

若有很大一组数据，数据的个数是N，在分布式数据存储情况下，找到这N个数的中位数。

数据输入是以下整型数据。

```
1、2、3、4、5、6、8、9、11、12、34
```

输出为：

```
6
```

2.设计思路

海量数据求中位数有很多解决方案。假设海量数据已经预先排序本例的解决方案为：将整个数据空间划分为K个桶。第一轮，在mapPartition阶段先将每个分区内的数据划分为K个桶，统计桶中的数据量，然后通过reduceByKey聚集整个RDD每个桶中的数据量。第二轮，根据桶统计的结果和总的数据量，可以判读数据落在哪个桶里，以及中位数的偏移量（offset）。针对这个桶的数据进行排序或者采用Top K的方式，获取到偏移为offset的数据。

3.代码示例

```
import org.apache.spark.{SparkContext， SparkConf}
import org.apache.spark.SparkContext.rddToPairRDDFunctions
 */
object Median {
```

```
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Spark Pi")
    val spark = new SparkContext(conf)
    val data = spark.textFile("data")
    /*将数据逻辑划分为10个桶，这里用户可以自行设置桶数量，统计每个桶中落入的数据量*/
    val mappeddata =data.map(num => {
      (num/1000 , num)
    })
    val count = mappeddata.reduceByKey((a , b) => {
      a+b
    }).collect()
    /*根据总的数据量，逐次根据桶序号由低到高依次累加，判断中位数落在哪个
    桶中，并获取到中位数在桶中的偏移量*/
    val sum_count=count.map(data => {
       data._2
    }).sum
    var temp = 0
    var index = 0
    var mid = sum_count/2
    for( i <- 0 to 10) {
        temp=temp+count(i)
        if(temp >= mid) {
        index=i
        break
      }
    }
    /*中位数在桶中的偏移量*/
    val offset = temp - mid
    /*获取到中位数所在桶中的偏移量为offset的数，也就是中位数*/
    val result = mappeddata.filter(num => num._1 == index ).takeOrdered(offset)
    println("Median is " + result(offset))
    spark.stop()
  }
}
```

4.应用场景

统计海量数据时，经常需要预估中位数，由中位数大致了解某列数据，做机器学习和数据挖掘的很多公式中也需要用到中位数。

6.4 倒排索引

倒排索引（inverted index）源于实际应用中需要根据属性的值来查找记录。在索引表中，每一项均包含一个属性值和一个具有该属性值的各记录的地址。由于记录的位置由属性值确定，而不是由记录确定，因而称为倒排索引。将带有倒排索引的文件称为倒排索引文件，简称倒排文件（inverted file）。其基本结构如图6-1所示。

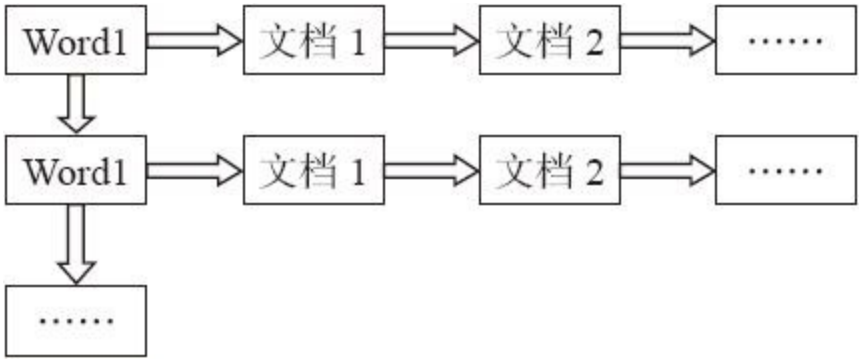

图6-1 倒排索引基本结构

搜索引擎的关键步骤是建立倒排索引。相当于为互联网上几千亿页网页做了一个索引，与书籍目录相似，用户想看与哪一个主题相关的章节，直接根据目录即可找到相关的页面。

1.实例描述

输入为一批文档集合，合并为一个HDFS文件，以分隔符分隔。

```
Id1 The Spark
……
Id3 The Spark
……
```

输出如下（单词，文档ID合并字符串）。

```
Spark id1 id2
Hadoop id3 id4
The id1 id3 id6
```

2.设计思路

首先进行预处理和分词，转换数据项为（文档ID，文档词集合）的RDD，然后将数据映射为（词，文档ID）的RDD，去重，最后在reduceByKey阶段聚合每个词的文档ID。

3.代码示例

倒排索引的示例代码如下所示：

```
import org.apache.spark.SparkContext
import scala.collection.mutable._
object InvertedIndex {
  def main(args : Array[String]) {
    val spark = new SparkContext("local", "TopK",
      System.getenv("SPARK_HOME"), SparkContext.jarOfClass(this.getClass))
    /*读取数据，数据格式为一个大的HDFS文件中用\n分隔不同的文件，用\t分隔文件ID和文件内容，用" "分隔文件内的词汇*/
    val words = spark.textFile("dir").map(file => file.split("\t")).map(item => {
      (item(0), item(1))
    }).flatMap(file => {
      /*将数据项转换为LinkedList[(词， 文档id)]的数据项，并通过flatmap将RDD内的数据项转换为（词， 文档ID）*/
      val list = new LinkedList[(String, String)]
      val words = file._2.split(" ").iterator
      while(words.hasNext) {
        list+words.next()
      }
      list
    }).distinct()
    /*将（词，文档ID）的数据进行聚集，相同词对应的文档ID统计到一起，形成
    （词， "文档ID1，文档ID2 ，文档ID3……"），形成简单的倒排索引*/
    words.map(word => {
      (word._2, word._1)
    }).reduce((a, b) => {
      a+"\t"+b
    }).saveAsTextFile("index")
  }
}
```

4.应用场景

搜索引擎及垂直搜索引擎中需要构建倒排索引，文本分析中有的场景也需要构建倒排索引。

6.5 CountOnce

假设HDFS只存储一个标号为ID的Block，每份数据保存2个备份，这样就有2个机器存储了相同的数据。其中ID是小于10亿的整数。若有一个数据块丢失，则需要找到哪个是丢失的数据块。

在某个时间，如果得到一个数据块ID的列表，能否快速地找到这个表中仅出现一次的ID？即快速找出出现故障的数据块的ID。

问题阐述：已知一个数组，数组中只有一个数据是出现一遍的，其他数据都是出现两遍，将出现一次的数据找出。

1.实例描述

输入为Block ID。

```
1、2、2、3、3、1、5、7、11……
```

输出为：

```
100
```

2.设计思路

利用异或运算将列表中的所有ID异或，之后得到的值即为所求ID。先将每个分区的数据异或，然后将结果进行异或运算。

3.代码示例

CountOnce的代码示例如下：

```
import org.apache.spark.SparkContext
```

```
import org.apache.spark.SparkContext._
 */
object NumOnce {
  def computeOneNum(args: Array[String]) {
    val spark = new SparkContext("local[1]", "NumOnce",
      System.getenv("SPARK_HOME"), SparkContext.jarOfClass(this.getClass))
    val data = spark.textFile("data")
   /*每个分区分别对数据进行异或运算，最后在reduceByKey阶段，将各分区异或运算的结果再做异或运算合并。偶数次出现的数字，异或运算之后为0，奇数次出现的数字，异或后为数字本身*/
    val result = data.mapPartitions(iter => {
        var temp = iter.next()
        while(iter.hasNext) {
          temp = temp^iter.next()
        }
        Seq((1, temp)).iterator
      }).reduceByKey(_^_).collect()
    println("num appear once is :"+result(0))
  }
}
```

4.应用场景

数据块损坏检索。例如，每个数据块有两个副本，有一个数据块损坏，需要找到那个数据块。

6.6 倾斜连接

并行计算中，我们总希望分配的每一个任务（task）都能以相似的粒度来切分，且完成时间相差不大。但是由于集群中的硬件和应用的类型不同、切分的数据大小不一，总会导致部分任务极大地拖慢了整个任务的完成时间。硬件不同暂且不论，下面举例说明不同应用类型的情况，如Page Rank或者Data Mining中的一些计算，它的每条记录消耗的成本不太一样，这里只讨论关于关系型运算的Join连接的数据倾斜状况。

数据倾斜原因如下。

1）业务数据本身的特性。

2）Key分布不均匀。

3）建表时考虑不周。

4）某些SQL语句本身就有数据倾斜。

数据倾斜表现如下。

任务进度长时间维持，查看任务监控页面，由于其处理的数据量与其他任务差异过大，会发现只有少量（1个或几个）任务未完成。

1.实例描述

输入：

表A（数据倾斜），表B

输出：

表C（A，B连接后的表）

2.设计思路

数据倾斜有很多解决方案，本例简要介绍一种实现方式。假设表A和表B连接，表A数据倾斜，只有一个Key倾斜。首先对A进行采样，统计出最倾斜的Key。将A表分隔为A1只有倾斜Key，A2不包含倾斜Key，然后分别与B连接。

3.代码示例

倾斜连接的代码示例如下：

```
import org.apache.spark.SparkContext
import  org.apache.spark.SparkContext._
 */
object SkewJoin {
  def main(args : Array[String]) {
    val skewedTable = left.execute()
    val spark = new SparkContext("local", "TopK",
      System.getenv("SPARK_HOME"), SparkContext.jarOfClass(this.getClass))
    /*存在数据倾斜的数据表*/
    val skewTable = spark.textFile("skewTable")
    /*与skewTable连接的表*/
    val Table =  spark.textFile("Table")
    /*对数据倾斜的表进行采样，假设只有一个key倾斜最严重，获取倾斜最大的key*/
    val sample = skewTable.sample(false, 0.3, 9).groupByKey().collect()
   val maxrowKey = sample.map(rows => (rows._2.size, rows._1)).maxBy(rows =>
 rows._1)._2
    /*将倾斜的表拆分为两个RDD，一个为只含有倾斜key的表，一个为不含有倾斜key的表*/
    /*分别与原表连接*/
    val maxKeySkewedTable = skewTable.filter(row => {
      buildSideKeyGenerator(row) == maxrowKey
    })
    val mainSkewedTable = skewTable.filter(row => {
      !(buildSideKeyGenerator(row) == maxrowKey)
    })
    /*分别与原表连接*/
    val maxKeyJoinedRdd = maxKeySkewedTable.join(Table)
    val mainJoinedRdd = mainSkewedTable.join(Table)
    /*将结果合并*/
    sc.union(maxKeyJoinedRdd, mainJoinedRdd)
  }
}
```

4.应用场景

在大数据分析平台中，经常遇到数据倾斜问题，读者可以参照相应的思路处理数据倾斜的处理。SQL on Hadoop系统中也需要处理数据倾斜问题。

6.7 股票趋势预测

本例将介绍如何使用Spark构建实时数据分析应用[1]，以分析股票价格趋势。

本例假设已预先连接了Spark Streaming。读者可以阅读介绍BDAS的章节预先了解相关概念。

第一步，需要获取数据流，本例使用JSON/WebSocket格式呈现6种实时市场金融信息。

第二步，需要知道如何使用获取到的数据流，本例不涉及专业的金融知识，但可以在这个应用中通过价格改变规律，预测价格趋势。

1.实例描述

本例通过使用scalawebsocket库（Github网址为https://github.com/pbuda/scalawebsocket）访问WebSocket。scalawebsocket库只支持Scala 2.10。获取网上的金融数据流。

输入为：股票名和相应价格。

```
-------------------------------------------
Time: 1375194945000 ms
-------------------------------------------
Croda International PLC - 24.82 - 24.82
ASOS PLC - 47.485 - 47.485
Arian Silver Corp - 0.0435 - 0.0435
Medicx Fund Ltd - 0.7975 - 0.7975
Supergroup PLC - 10.73 - 10.73
Diageo PLC - 20.07 - 20.075
Barclays PLC - 2.891 - 2.8925
QinetiQ Group PLC - 1.874 - 1.874
CSR PLC - 5.7 - 5.7
United Utilities Group PLC - 7.23 - 7.23
```

输出为：处于增长趋势的股票名称。

```
-------------------------------------------
Positive Trending (Time: 1375269240035 ms)
```

```
-------------------------------------------
Real estate
Telecommunication
Graphics , publishing & printing media
Environmental services & recycling
Agriculture & fishery
```

2.设计思路

通过Spark Streaming的时间窗口，增加新数据，减少旧数据。本例中的reduce函数用于对所有价格改变求和（有正向的改变和负向的改变）。之后希望看到正向的价格改变数量是否大于负向的价格改变数量，这里通过改变正向数据将计数器加1，改变负向的数据将计数器减1进行统计，从而统计出股票的趋势。

3.代码示例

通过本书的BDAS章节，假设读者已经对Spark和Spark Streaming有了初步了解，下面将介绍整个应用的设计与开发。

（1）接收流数据

为了在Spark中处理流数据，需要创建一个StreamingContext对象（Spark Streaming中的上下文对象），作为流处理的上下文。之后注册一个输入流（InputDStream），它会初始化一个接收器（Receiver）对象（Spark默认提供了许多类型的接收器，如Twitter、Akka Actor、ZeroMQ等）。由于默认没有网页套接字（WebSocket）的实现，所以本例将自定义这个类，获取网页流数据。

本例通过使用scalawebsocket库（Github网址为https://github.com/pbuda/scalawebsocket）访问WebSocket。scalawebsocket库只支持Scala 2.10。

读者可以选用支持Scala 2.10的Spark版本。

通过下面的代码实现一个简单的trait，进而使用WebSocket（它产生所有可用的股票序

列）。

```
import scalawebsocket.WebSocket
trait PriceWebSocketClient {
  import Listings._
  def createSocket(handleMessage: String => Unit) = {
    websocket = WebSocket().open("ws://localhost:8080/1.0/marketDataWs").onTextMessage(m => {
      handleMessage(m)
    })
    subscriptions.foreach(listing => websocket.sendText("{\"subscribe\":{" + listing + "}}"))
  }
  var websocket: WebSocket = _
}
classPriceEchoextendsPriceWebSocketClient{
createSocket(println)
}
```

为了能够让Spark正确挂接到WebSocket，并不断接收消息，可以通过实现一个接收器（Receiver）达到这个目的。由于接收的数据符合通用的网络协议，所以通过继承NetworkReceiver类实现接收器。用户需要创建一个块生成器（block generator），并将接收到的消息附加到块生成器中。

```
classPriceReceiverextendsNetworkReceiver[String]withPriceWebSocketClient{
lazy val blockGenerator = new BlockGenerator(StorageLevel.MEMORY_ONLY_SER)
protected override def onStart() {
blockGenerator.start
createSocket(m => blockGenerator += m)
}
protected override def onStop() {
blockGenerator.stop
websocket.shutdown
}
}
```

到目前为止，获取的流数据是以JSON格式存储的文本字符串，通过抽取数据中重要的部分进而用其创建case类，这样数据处理将变得更容易。创建一个PriceUpdate case类。

```
import scala.util.parsing.json.JSON
import scala.collection.JavaConversions
import java.util.TreeMap
case class PriceUpdate(id: String, price: Double, lastPrice: Double)
object PriceUpdate{
val lastPrices = JavaConversions.asMap(new TreeMap[String,Double])
def apply(text: String): PriceUpdate = {
val(id, price) = getIdAndPriceFromJSON(text)
val lastPrice: Double = lastPrices.getOrElse(id, price)
lastPrices.put(id, price)
PriceUpdate(id, price, lastPrice)
}
  /*此方法解析与处理JSON数据格式，暂不赘述*/
def getIdAndPriceFromJSON(text: String) = // snip - simple JSON processing
}
/*这时，还不能找到金融序列属性，不能获取之前的价格信息。
```

```
同时，需要更新接收器类为下面的情况，解析输入数据*/
import spark.streaming.dstream.NetworkReceiver
import spark.storage.StorageLevel
class PriceReceiver extends NetworkReceiver[PriceUpdate]withPriceWebSocketClient{
lazy val blockGenerator = new BlockGenerator( StorageLevel.MEMORY_ONLY_SER )
protected override def onStart( ) {
blockGenerator.start
createSocket( m => {
val priceUpdate = PriceUpdate( m )
blockGenerator += priceUpdate
} )
}
protected override def onStop( ) {
blockGenerator.stop
websocket.shutdown
}
}
```

还需要实现一个输入流（InputDStream），这个输入流需要实现getReceiver方法，当外部调用这个方法时，返回一个初始化好的价格接收器。

```
object streamextendsNetworkInputDStream[PriceUpdate]( ssc ) {
override def getReceiver( ) : NetworkReceiver[PriceUpdate] = {
newPriceReceiver( )
}
}
```

将之前的程序加入Spark Streaming的主干程序中。

```
/*创建Spark Streaming上下文*/
val ssc = new StreamingContext( "local", "datastream", Seconds( 15 ), "C:/software/spark-1.0.0",
List( "target/scala-2.10.3=/spark-data-stream_2.10.3-1.0.jar" ) )
 /* 创建并注册输入流*/
 ssc.registerInputStream( stream )
 /*启动流数据处理引擎*/
 ssc.start( )
```

上面这段代码初始化了流数据处理的上下文，并配置了应用。

local表示在本地执行，datastream是应用的名称，Seconds（15）定义批处理的时间片，"C:/software/spark-1.0.0"定义Spark的路径，List（"target/scala-2.10.3=/spark-data-stream_2.10.3-1.0.jar"）定义需要的Jar包。

项目结构要为SBT的项目格式，之后在根目录运行SBT package run即可运行，这样将会编译，打包程序生成target/scala-2.10.3=/spark-data-stream_2.10.3-1.0.jar，然后Spark应用使用Jar中的类。

下面代码为到目前为止应用的完整代码。

```
override def main(args: Array[String]) {
import Listings._
val ssc = new StreamingContext("local", "datastream", Seconds(15), "C:/software/spark-0.7.3",
List("target/scala-2.9.3/spark-data-stream_2.9.3-1.0.jar"))
object stream extends NetworkInputDStream[PriceUpdate](ssc) {
override de fgetReceiver(): NetworkReceiver[PriceUpdate] = {
newPriceReceiver()
}
}
ssc.registerInputStream(stream)
stream.map(pu => listingNames(pu.id) + " - " + pu.lastPrice + " - " + pu.price).print()
ssc.start()
}
```

控制台将会产生以下输出。

```
-------------------------------------------
Time: 1375194945000 ms
-------------------------------------------
Croda International PLC - 24.82 - 24.82
ASOS PLC - 47.485 - 47.485
Arian Silver Corp - 0.0435 - 0.0435
Medicx Fund Ltd - 0.7975 - 0.7975
Supergroup PLC - 10.73 - 10.73
Diageo PLC - 20.07 - 20.075
Barclays PLC - 2.891 - 2.8925
QinetiQ Group PLC - 1.874 - 1.874
CSR PLC - 5.7 - 5.7
United Utilities Group PLC - 7.23 - 7.23
……
```

（2）处理流数据

通过上文的初始化和数据接收，已经可以源源不断地获取数据了。下面介绍如何处理和分析数据。

下面程序可将数据转化为类股，改变价格和频度的序列。第一次处理时，将每个数据项转化为（类股，价格改变，1）的元组。通过下面的代码完成这个过程。

```
val sectorPriceChanges = stream.map(pu => (listingSectors(pu.id), (pu.price - pu.lastPrice, 1)))
```

现在就可以使用reduceByKeyAndWindow函数了，这个函数允许用户使用滑动窗口处理数据，时间窗口内的数据将会使用reduce函数处理，使用Key-Value对中的Key作为reduce的关键字，这里将使用一个reduce函数和反向reduce函数。

这样每次在时间窗口内迭代时，Spark都对新数据进行reduce处理，需要丢弃的旧数据不再使用reduce处理。

Spark需要做的就是撤销之前最左侧旧数据对整个reduce数据结果的改变，增加右侧新的reduce数据对整个reduce数据结果产生新的改变。

需要写一个reduce和inverse reduce函数。在本例中，reduce函数用于对所有价格改变求和（有正向的改变和负向的改变）。为了看到正向的价格改变数量大于负向的价格改变数量，这里可以通过改变正向数据，将计数器加1，改变负向的数据，将计数器减1达到这个效果。代码如下。

```
val reduce = (reduced: (Double, Int), pair: (Double, Int)) => {
if(pair._1 > 0) (reduced._1 + pair._1, reduced._2 + pair._2)
else(reduced._1 + pair._1, reduced._2 - pair._2)
}
val invReduce = (reduced: (Double, Int), pair: (Double, Int)) => {
if(pair._1 > 0) (reduced._1 + pair._1, reduced._2 - pair._2)
else(reduced._1 + pair._1, reduced._2 + pair._2)
}
val windowedPriceChanges = sectorPriceChanges.reduceByKeyAndWindow(reduce, invReduce, Seconds(5*60),
Seconds(15))
```

现在通过上文介绍的函数，已经可以构建一个reduce流处理应用，这个应用能够感知价格波动和趋势。由于只希望呈现出正向波动最剧烈的一些类股。可以过滤流数据，保留下正向波动的类股，然后将数据元组Key-Value的Key改变为可以排序，统计出波动最大的类股的属性。本例假设正向波动剧烈与否的权重为价格改变大小乘以价格改变计数器值，将Value中的两个值组合计算出的结果作为新的Key。最后，将数据按照新的Key打印出最大的5个类股。

```
import scala.collection.immutable.List
import spark.SparkContext._
import spark.streaming._
import spark.streaming.StreamingContext._
import spark.streaming.dstream._
object DataStreamextendsApp{
val reportHeader = """-------------------------------------------------
Positive Trending
================
""".stripMargin
override def main(args: Array[String]) {
import Listings._
```

```
import System._
val ssc = new StreamingContext("local", "datastream", Seconds(15), "C:/software/spark-0.7.3",
List("target/scala-2.9.3/spark-data-stream_2.9.3-1.0.jar"))
object stream extends NetworkInputDStream[PriceUpdate](ssc) {
override def getReceiver(): NetworkReceiver[PriceUpdate] = {
newPriceReceiver()
}
}
ssc.checkpoint("spark")
ssc.registerInputStream(stream)
val reduce = (reduced: (Double, Int), pair: (Double, Int)) => {
if(pair._1 > 0) (reduced._1 + pair._1, reduced._2 + pair._2)
else(reduced._1 + pair._1, reduced._2 - pair._2)
}
val invReduce = (reduced: (Double, Int), pair: (Double, Int)) => {
if(pair._1 > 0) (reduced._1 + pair._1, reduced._2 - pair._2)
else(reduced._1 + pair._1, reduced._2 + pair._2)
}
val sectorPriceChanges = stream.map(pu => (listingSectors(pu.id), (pu.price - pu.lastPrice, 1)))
val windowedPriceChanges = sectorPriceChanges.reduceByKeyAndWindow(reduce, invReduce, Seconds(5*60),
Seconds(15))
val positivePriceChanges = windowedPriceChanges.filter{case (_, (_, count)) => count > 0}
val priceChangesToSector = positivePriceChanges.map{case(sector, (value, count)) => (value * count,
sector)}
val sortedSectors = priceChangesToSector.transform(rdd => rdd.sortByKey(false)).map(_._2)
sortedSectors.foreach(rdd => {
println("""|-------------------------------------------
|Positive Trending (Time: %d ms)
|-------------------------------------------
|""".stripMargin.format(currentTimeMillis + rdd.take(5).map(sectorCodes(_)).mkString("\n")))
})
ssc.start()
}
}
```

运行上述示例，将会打印出下面的日志，这样就构建出了预测类股趋势的Spark应用。

```
-------------------------------------------
Positive Trending (Time: 1375269240035 ms)
-------------------------------------------
Real estate
Telecommunication
Graphics, publishing & printing media
Environmental services & recycling
Agriculture & fishery
-------------------------------------------
Positive Trending (Time: 1375269255035 ms)
-------------------------------------------
Real estate
Graphics, publishing & printing media
Environmental services & recycling
Agriculture & fishery
Electrical appliances & components
-------------------------------------------
Positive Trending (Time: 1375269270034 ms)
-------------------------------------------
Environmental services & recycling
Agriculture & fishery
Electrical appliances & components
Vehicles
Precious metals & precious stones
```

4.应用场景

读者可以通过这个示例，开发自己的流数据分析应用。数据源可以是爬虫抓取的数据，也可以是消息中间件输出的数据等待。

Spark是整个Spark生态系统的底层核心引擎，单一的Spark框架并不能完成所有计算范式任务。如果有更复杂的数据分析需求，就需要借助Spark的上层组件。例如，为了分析大规模图数据，需要借助GraphX构建内存的图存储结构，然后通过BSP模型迭代算法。为了进行机器学习，需要借助MLlib底层实现的SGD等优化算法，进行搜索和优化。分析流数据需要借助Spark Streaming的流处理框架，将流数据转换为RDD，输入与分析流数据。如果进行SQL查询或者交互式分析，就需要借助Spark SQL这个查询引擎，将SQL翻译为Spark Job。相应的示例用户可以参考BDAS章节和示例进行学习。

[1] 本节参考文章：James Phillpotts，Real-time Data Analysis Using Spark，29 Jul 2013。

6.8 本章小结

通过本章的介绍，相信读者已经可以独立编写Spark用例了。Spark使用Scala书写，不熟悉的读者可以预先学习Scala语法，这样编写Spark程序才会游刃有余。

WordCount是大数据程序的入门程序，程序虽然简单，但其中的程序并行化思想很值得借鉴。连接示例，让读者可以进一步了解如何进行数据统计，Top K，倒排索引，查找中位数、倾斜连接。最后介绍的股票趋势预测应用较为复杂，但是通过整个架构可以体会Spark是如何处理实际问题的。

读者对Spark编程有了一定的基础之后，需要使用Benchmark对应用进行基准测试，进而调整算法。需要进行系统选型时，也需要使用Benchmark进行性能评测。下面将对大数据领域的Benchmark进行全景介绍，读者可以通过大数据Benchmark进行Spark系统或应用的基准测试。

第7章　Benchmark使用详解

Benchmark作为一种评价方式，在整个计算机领域有着长期的应用。在维基百科上的解释是"As computer architecture dvanced，it became more difficult to compare the performance of various computer systems simply by looking at their specifications.Therefore，tests were developed that allowed comparison of different architectures."Benchmark在计算机领域应用最成功的就是性能测试，主要测试负载的执行时间、传输速度、吞吐量、资源占用率等。

大数据领域Benchmark标准尚未统一，生产环境和科研实验室迫切需要大数据Benchmark进行基准测试，对大数据分析系统选型和系统二次开发进行指导。Spark刚刚兴起，针对Spark进行开发的Benchmark还不多，用户可以根据自己的需求进行Benchmark选型，并可以借鉴其他Benchmark的数据生成器生成数据集，开发相应的典型工作负载完成对Spark的基准测试，诊断系统问题，更好地进行应用开发和系统二次开发。

7.1 Benchmark简介

Benchmark性能基准测试本质上就是生成模拟数据或真实数据，在系统上运行典型负载（Workload），进而暴露系统瓶颈和性能优势，最终完成系统评测。

Benchmark的种类很多，有些偏重于硬件，有些偏重于软件，还有些是对整个系统进行综合度量和评价。

我们为什么要使用Benchmark呢？其实Benchmark对软硬件系统生产厂商和用户都是很有价值的。生产厂商利用Benchmark在产品的开发过程中诊断性能瓶颈，随时进行调优和优化。产品发布之后，也能够根据评测结果进行很有说服力的宣传，说明产品基于哪个Benchmark性能，达到什么样的标准。同时有些厂商没有真实环境的数据，需要利用Benchmark来生产数据和负载，模拟生产环境的运行状况。对用户来说就是可以根据Benchmark的结果对比不同厂商的不同产品，更加方便和透明地选择系统。

同时，Benchmark对一个领域的技术发展很有积极意义，例如，在数据库领域，TPC的Bench已经成为开发数据库的主流Benchmark。开发者在开发的过程中，利用Benchmark生成结构化的数据，同时利用查询生成器生成指定数据库的SQL方言。这样在开发的数据库或者改进的算法上用SQL负载（Workload）进行测试，就能够更加精准地了解性能瓶颈，对系统进行调优。

选择Benchmark需要有明确的目的。当用于产品发布时，就应该用这个领域主流和权威的Benchmark进行评估。同时有些Benchmark只适用于某些领域。例如，TPC针对数据仓库开发了TPC-DS Benchmark，而针对其他的数据库又有TPC-H等系列的Benchmark供用户使用。

在现在的Bigdata领域，Benchmark标准尚在制定，但是有一些Benchmark得到部分应用。

下面介绍现有的几款大数据的Benchmark，用户可以根据自己的情况选用，对Spark或者Spark上的系统进行性能评测。

7.1.1 Intel Hibench与Berkeley BigDataBench

首先来介绍Hibench。

1.Hibench

Hibench[1]是由Intel开发的一个针对Hadoop的基准测试工具。它包含有一组Hadoop工作负载的集合，有人工模拟实验环境的工作负载，也有一部分是生产环境的Hadoop应用程序。Hibench是广泛应用的Hadoop Benchmark。因为Hibench针对Hadoop，如果想使用其作为Spark的Benchmark，则需要自己针对数据集开发一些Workload算法。Hibench是开源的，用户可以到Github库中下载：https://github.com/intel-hadoop/HiBench。

Hibench包含的负载如表7-1所示。

表7-1 Hibench所包含的负载

工作负载	所属领域	类型	实现
Sort	基本 Benchmark	I/O 密集型，CPU 利用率中等	内部实现
WordCount	基本 Benchmark	CPU 密集型	内部实现

（续）

工作负载	所属领域	类型	实现
Tera Sort	基本 Benchmark	Map 和 Shuffle 阶段 CPU 密集，I/O 中等，Reduce 阶段 I/O 密集，CPU 中等	内部实现
Nutch Indexing	搜索引擎	Map 阶段 CPU 密集，Reduce 阶段 I/O 密集，CPU 中等	基于 Nutch
Page Rank	搜索引擎	CPU 密集型	基于 SmartFRog
Bayesis Classisfication	机器学习	I/O 密集型	基于 Mahout
K-means	机器学习	中心点计算阶段，CPU 密集型 聚类阶段，I/O 密集型	基于 Mahout
DFSIO	HDFS I/O	I/O 密集型	自己实现

下面介绍在Hibench的基础上衍生出的一款新的Big Data Benchmark。

2.Berkeley BigDataBench

Berkeley BigDataBench[2]是随着Spark、Shark的推出，由AMPLab开发的一套大数据基准测试工具。由于其一部分是基于Hibench的数据集和数据生成器，所以将二者放在一起介

绍，同时它有一部分数据集是Common Crawl上采样的文档数据集，这点和Hibench不同。其目前主要针对SQL on Hadoop产品进行基准测试。现在还很简陋，不排除以后还会增加对Spark和其他Spark生态系统组件的支持。感兴趣的读者可以到官方主页（https://amplab.cs.berkeley.edu/benchmark/）了解更多内容。

现在支持Documents、Ranking和UserVisits 3个数据集，这3个数据集的模式，如表7-2所示。

表7-2　3个数据集的模式

Documents	Rankings	UserVisits
非结构化 HTML 文档	网址列表和网址相应的 PageRank 评分	每个页面的访问日志
非结构化文档	表属性： pageURL VARCHAR(300) pageRank INT avgDuration INT	表属性： sourceIP VARCHAR(116) destURL VARCHAR(100) visitDate DATE adRevenue FLOAT userAgent VARCHAR(256) countryCode CHAR(3) languageCode CHAR(6) searchWord VARCHAR(32) duration INT

为了便于广大用户理解工作负载，BigDataBench选用了SQL作为测试的工作负载，而没有选择机器学习、流计算和图计算等工作负载。

下面简要介绍Berkely BigData Benchmark工作负载。

（1）Scan Query

```
SELECT pageURL, pageRank FROM rankings WHERE pageRank > X
```

这个查询的目的是对关系表进行选择和投影操作。

（2）Aggregation Query

```
SELECT SUBSTR(sourceIP, 1, X), SUM(adRevenue) FROM uservisits GROUP BY SUBSTR(sourceIP, 1, X)
```

这个查询的目的是先对关系表分组，然后使用字符串解析的函数对每个元组进行解析，最后进行一个高基数的聚集函数操作。

（3）Join Query

```
SELECT sourceIP, totalRevenue, avgPageRank
FROM
  (SELECT sourceIP,
        AVG(pageRank) as avgPageRank,
        SUM(adRevenue) as totalRevenue
    FROM Rankings AS R, UserVisits AS UV
    WHERE R.pageURL = UV.destURL
      AND UV.visitDate BETWEEN Date(`1980-01-01') AND Date(`X')
    GROUP BY UV.sourceIP)
  ORDER BY totalRevenue DESC LIMIT 1
```

这个查询使用大小表连接，然后对结果进行排序。因为很多SQL on Hadoop产品都是基于Map Reduce计算模型，所以这里涉及一个经典的优化方式是Map Side Join，可以避免Shuffle阶段的网络开销。

（4）External Script Query

```
CREATE TABLE url_counts_partial AS
 SELECT TRANSFORM (line)
   USING "python /root/url_count.py" as (sourcePage, destPage, cnt)
 FROM documents;
CREATE TABLE url_counts_total AS
 SELECT SUM(cnt) AS totalCount, destPage
 FROM url_counts_partial
 GROUP BY destPage;
```

在这个查询中，调用一个会抽取和聚集URL信息的Python外部函数，然后分组聚集整个URL的数量。

[1] 感兴趣的读者可参见：http://baidutech.blog.51cto.com/4114344/743496，HCE Benchmark.51CTO博客，2011-02-11。还可参见论文：Shengsheng Huang, Jie Huang, Yan Liu and Jinquan Dai，HiBench: A Representative and Comprehensive Hadoop Benchmark Suite。

[2] 参见：https://amplab.cs.berkeley.edu/benchmark/，参考自amplab对Benchmark的相

关论述。

7.1.2 Hadoop GridMix

作为Hadoop自带的Benchmark，Gridmix同样不支持Spark，用户要使用Spark，仍需自己实现Workload算法。作为Hadoop自带的测试工具，使用方便、负载经典，所以应用广泛。

Gridmix的使用用例不能代表所有的Hadoop使用场景。Gridmix的用例中，没有包括较为复杂的计算，也没有明显的CPU密集型的用例。而现实应用中，存在很多I/O密集型的应用，同时CPU密集型的应用也大量存在，如机器学习算法、构建倒排索引等。因此，Gridmix的WorkLoad负载并不能完全展现大数据工作负载的全貌。[1]表7-3为Gridmix负载的介绍。

表7-3 Gridmix所包含的负载

工作负载	说明
javasort	数据排序，使用的是 Java 的原生接口
pipesort	数据排序，使用 pipe 接口对数据进行排序
streamsort	数据排序，使用 streaming 接口对数据进行排序
webdatascan	根据设定的一定采样率对原始数据进行一轮采样，使用 GenericMRLoader 采样器
webdatasort	设定采样率为 100%，对原始数据进行三轮迭代的采样。同样，使用的采样器是 GenericMRLoader
monsterQuery	设定一定的采样率对数据采样，进行三轮迭代，使用的采样器是 GenericMRLoader
maxent	设定一定采样率进行八轮迭代，使用的采样器是 GenericMRLoader

[1] 参见：http://baidutech.blog.51cto.com/4114344/743496，HCE Benchmark.51CTO博客，2011-02-11。

7.1.3 Bigbench、BigDataBenchmark与TPC-DS

1.Bigbench

BigBench[1]是由Teradata、多伦多大学、InfoSizing、Oracle开发的一款大数据Benchmark。其设计思想和复用扩展的方式很具有研究价值，如果读者感兴趣，可以参阅论文：Bigbench：Towards an industry standard benchmark for big data analytics。

Bigbench可以作为Benchmark研究者的一个范例。Bigbench基于零售业的产品销售场景，用于评测的系统是大规模并行关系数据库和MapReduce类型的执行引擎。

BigBench的数据模型具有以下3种类型。

·结构化的数据：利用TPC-DS生成，筛选了TPC-DS的星型模型数据，并从中挑选部分典型关系表。

·半结构化的数据：利用网页浏览日志并通过PDGF工具生成，这些日志由零售业的客户的浏览页面产生。这些日志格式很像Apache服务器产生的日志格式，并和TPC-DS的数据模式融合，其数据规模也可以随着配置规模因子弹性调整。

·非结构化的数据：数据基于真实数据作为输入，进行采样，并利用字典通过一种使用马尔科夫链的算法生成的文本数据。同样，非结构化数据与结构化和半结构化数据融合。数据规模也可以根据配置的因子弹性动态增长。

整个数据生成器的数据模型如图7-1所示。

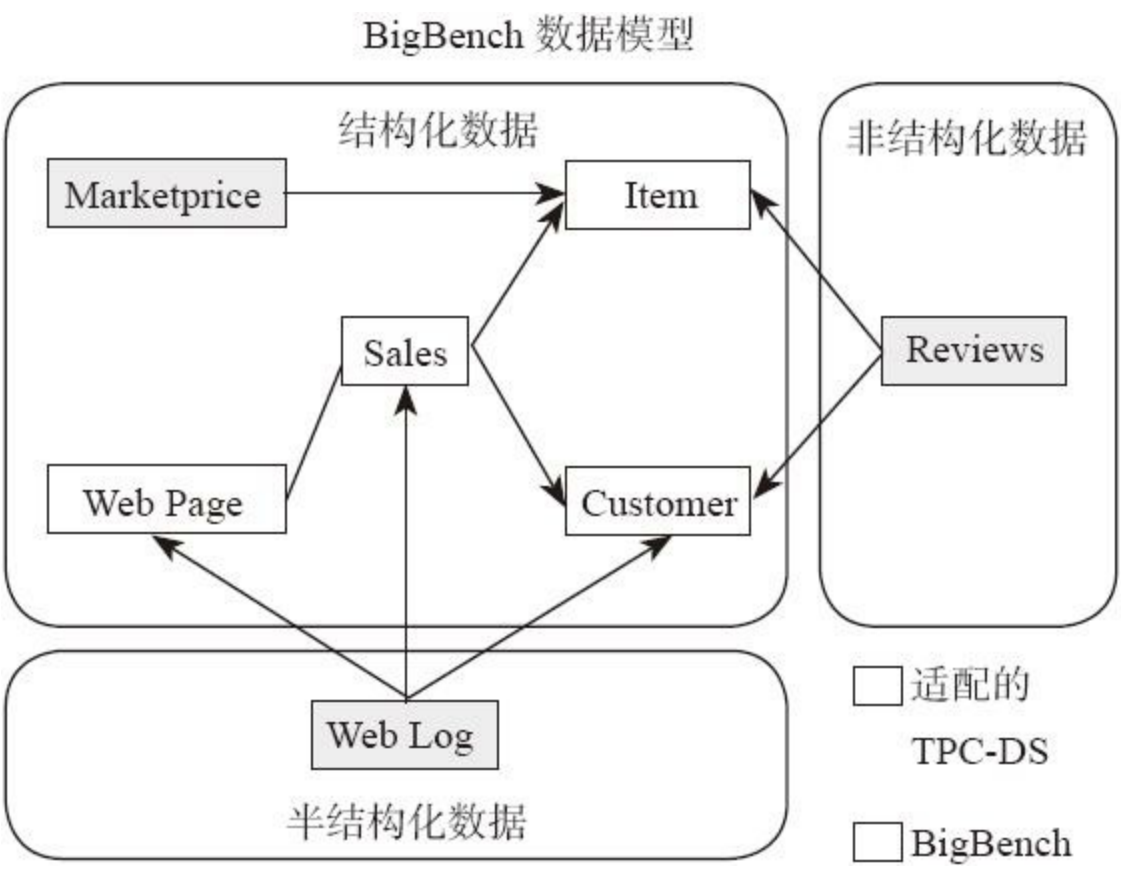

图7-1　BigBench数据模型

BigBench的工作负载主要是针对零售业的大数据分析，同时覆盖了机器学习等算法。现在的工作负载还不是很多，但已经有30个Query。这个Benchmark工具还在扩展。

可以从以下几个维度来理解这个Benchmark工具现在和未来将要扩展的工作负载类型。

1）从业务维度：针对市场、销售、运营、供应链、报表5方面的负载。

2）从数据源的维度：针对结构化数据、半结构化数据、非结构化数据。

3）数据处理的方式和类型的维度：声明型语言、结构化语言或者二者的混合。

4）从分析技术的维度：统计分析、数据挖掘分析和采样报告3个维度。

2.BigDataBenchmark

BigDataBenchmark[2]是由中科院计算所开发的一款开源的大数据Benchmark。BigDataBenchmark集成了19个大数据Benchmark，并从应用情景维度、运营和算法维度、

数据类型维度、数据源维度以及软件栈和应用类型维度综合考虑，开发出这款Benchmark用来公正地对比和评判大数据系统和架构。BigDataBenchmark包含多样的数据输入类型。感兴趣的读者可以参见论文：BigDataBench：a Big Data Benchmark Suite from Internet Services，访问官方主页：http://prof.ict.ac.cn/BigDataBench/publications/。

BigDataBenchmark的设计基于典型的大数据负载。由于在生产环境中，大数据应用的主导领域是搜索引擎、社交网络、电商。这三大领域占据了整个互联网80%的页面。BigDataBenchmark围绕这三大方向选取和开发相应领域的典型负载。

数据生成器也是基于这3个领域开发和设计的。针对搜索引擎领域产生文本数据，针对社交网络产生图数据，针对电商产生结构化关系表数据。

同时针对三大领域生成不同计算延迟的负载。在线（online）需要短的延迟；离线（offline）需要进行复杂数据计算分析；实时（Real-Time）需要交互式分析的负载。

3.TPC-DS

SQL on Hadoop产品的本质就是数据仓库系统，其作用是在大规模分布式的环境下分析和查询离线数据。TPC-DS（注：参见：TPC Benchmark^{TM} DS (TPC-DS):The New Decision Support Benchmark Standard。）广泛用于SQL on Hadoop的产品评测。但是，目前TPC-DS基准已经很难模拟越来越复杂的决策支持系统的业务需求。

TPC-H的数据模型满足数据库模式设计的第三范式，数据模型不是现在决策支持系统主流的星型模型或者雪花型模型，业务类型也不能很好地体现物化视图和索引等OLAP型查询引擎的优势，而且其数据表不能表达数据倾斜问题，限制了索引的过度使用。为了应对这个挑战，TPC组织推出了TPC-DS，现在也正在制作和推出大数据领域的Benchmark，但还未发布。所以现在很多厂商和科研院所采用TPC-DS暂时作为SQL on Hadoop的大数据测试的Benchmark。

传统数据仓库使用的主流Benchmark就是TPC-DS。我们可以使用TPC-DS最高生成100TB的数据，能够满足大数据量的要求。数据模式采用雪花型模式：24个平均含有18列的数据库表，同时提供了99个典型Query供用户使用。这些Query类型丰富，如Ad-hoc Query、Reporting Query等，满足用户多方面的要求。

笔者在2013年参与学校DB-IIR实验室的SQL on Hadoop结构化大数据测试，采用TPC-DS作为Benchmark测试大数据分析系统，这个测试结果在中国大数据大会2013上发布，并在VL DB 2014的workshop上进行了公布。

当时采用的查询负载有以下几种。

（1）单表查询

```
--qA5o--
select ss_store_sk as store_sk, ss_sold_date_sk as date_sk
ss_ext_sales_price as sales_price, ss_net_profit as profit
from store_sales
where ss_ext_sales_price>20
order by profit
limit 100;
--qA9--
select count(*) from store_sales
where ss_quantity between 1 and 20
limit 100;
```

（2）Ad hoc查询：--qB65g--（2表连接）

```
select ss_store_sk,
ss_item_sk,
sum(ss_sales_price) as revenue
from store_sales
join date_dim on(store_sales.ss_sold_date_sk =date_dim.d_date_sk)
where d_month_seq between 1176 and 1176+11
group by ss_store_sk, ss_item_sk
limit 100;
```

（3）星型查询：--qD27go--（5表连接）

```
select i_item_id, s_state, avg(ss_quantity) agg1, avg(ss_list_price) agg2,
avg(ss_coupon_amt) agg3, avg(ss_sales_price) agg4
from store_sales ss
join customer_demographics cd on(ss.ss_cdemo_sk = cd.cd_demo_sk)
join date_dim dd on(ss.ss_sold_date_sk = dd.d_date_sk)
join store s on(ss.ss_store_sk = s.s_store_sk)
```

```
join item i on(ss.ss_item_sk = i.i_item_sk)
where
cd_gender = 'M' and
cd_marital_status = 'S' and
cd_education_status = 'College' and
d_year = 2002 and s_state='TN'
group by i_item_id, s_state
order by i_item_id ,s_state
limit 100 ;
```

（4）复杂查询：--qD6gho--（5表连接）

```
select a.ca_state state, count(*) cnt
from customer_address a
join customer c on(a.customer_address.ca_address_sk =
c.c_current_addr_sk)
join store_sales s on(c.c_customer_sk = s.ss_customer_sk)
join date_dim d on(s.ss_sold_date_sk = d.d_date_sk)
join item i on(s.ss_item_sk = i.i_item_sk)
group by a.ca_state
having count(*) >= 10
order by cnt
limit 100;
```

---

[1] 参见论文：BigBench:Towards an Industry Standard Benchmark for Big Data Analytics。

[2] 参见：BigDataBench,A Big Data Benchmark Suite,ICT,Chinese Academy of Sciences。

7.1.4 其他Benchmark

在大数据领域还有一些针对特定负载的大数据Benchmark，读者感兴趣可以深入研究。下面介绍几个典型的Benchmark。

1）Malstone：针对数据密集型计算和分析的工作负载的Benchmark工具。它基于大规模并行计算，也具有云计算的属性。

2）Cloud Harmony：使用黑盒方式度量云服务提供商的性能。它基于大规模并行计算，并且面向硬件架构，评测复杂数据的大数据运算。

3）YCSB：度量和对比云数据库的框架。基于大规模并行计算，面向大数据和云计算。

4）SWIM：一个针对MapReduce的统计工作负载。基于MapReduce面向大数据的复杂数据集的分析测试。

5）LinkBench：针对图数据库的Bechmark，在Facebook数据库工程团队，通过分析Facebook的数据库工作负载（workload）并开发了这款称为LinkBench的数据库性能测试工具。LinkBench已经开源并发布到了Github。

6）DFSIO：是一个分布式文件系统的Benchmark，针对Hadoop测试HDFS的读写性能。

7）Hive performance Benchmark（Pavlo）：这是由Palvo最早提出的测试工具。这个Hive性能测试工具用于比较Hadoop和并行分析型数据库。它拥有5个工作负载，第一个是Grep（源于MapReduce的论文），其他4个典型的查询设计为代表传统的结构化分析工作负载，包括选择、聚集、连接、用户自定义函数的工作负载。Berkeley Big Data Bench就是借鉴Pavlo的Benchmark思想而进一步开发和实现的。

7.2 Benchmark的组成

Benchmark的核心由3部分组成：数据集、工作负载、度量指标。理解了这3个部分，就可以宏观了解Benchmark。下面从以下几点介绍Benchmark。

7.2.1 数据集

数据类型分为结构化数据、半结构化数据和非结构化数据。由于大数据环境下的数据类型复杂，负载多样，所以大数据Benchmark需要生成3种类型的数据和对应负载。

1）结构化数据：传统的关系数据模型、行数据，存储于数据库，可用二维表结构表示。

典型场景为互联网电商交易数据、企业ERP系统、财务系统、医疗HIS数据库、政务信息化系统、其他核心数据库等。结构规整，处理方案较为成熟。使用关系数据库进行存储和处理。

2）半结构化数据：类似XML、HTML之类，自描述，数据结构和内容混杂在一起。

典型应用场景为邮件系统、Web搜索引擎存储、教学资源库、档案系统，等等。可以考虑使用Hbase等典型的Key-Value存储系统存储。在互联网公司中存在大量的半结构化数据。

3）非结构化数据：各种文档、图片、视频/音频等。

典型应用场景为视频网站、图片相册、医疗影像系统、教育视频点播、交通视频监控、文件服务器（PDM/FTP）等具体应用。可以考虑使用HDFS等文件系统存储。在互联网公司同样存在大规模的非结构化数据。

7.2.2 工作负载

1.工作负载的维度

对工作负载的理解和设计可以分为以下几个维度。

（1）密集型计算类型

①CPU密集型计算。

②I/O密集型计算。

③网络密集型计算。

（2）计算范式

①SQL。

②批处理。

③流计算。

④图计算。

⑤机器学习。

（3）计算延迟

①在线计算。

②离线计算。

③实时计算。

（4）应用领域

①搜索引擎。

②社交网络。

③电子商务。

④地理位置服务。

⑤媒体，游戏。

由于互联网领域数据庞大，用户量大，成为大数据问题产生的天然土壤。大数据Benchmark的很多工作负载都是根据互联网领域的典型应用场景产生的。从图7-5中可以看到，BAT三巨头分别规划了自己的互联网布局，涉及电子商务、媒体游戏、社交媒体、搜索门户以及基于地理位置服务等多个领域。每个巨头旗下都有数家小公司与其有着紧密的联系，正是互联网应用中产生了大量的典型大数据工作负载。

由于Spark兴起的时间较Hadoop晚很多，其相应的Benchmark也不如Hadoop可用的多。但是我们也看到，Spark兼容和利用Hadoop存储的数据，这就使用户可以利用以往Hadoop的Benchmark的数据生成器生成数据，使用Hadoop存储数据，然后根据特定的负载重写Spark的工作负载。因为Spark的编程表现力要远远超过Hadoop，所以Hadoop的工作负载完全能用Spark重写，而且Benchmark的负载目的只是突出在特定的计算密集型计算下暴露系统性能瓶颈，一般逻辑简单，所以改写工作量并不大。

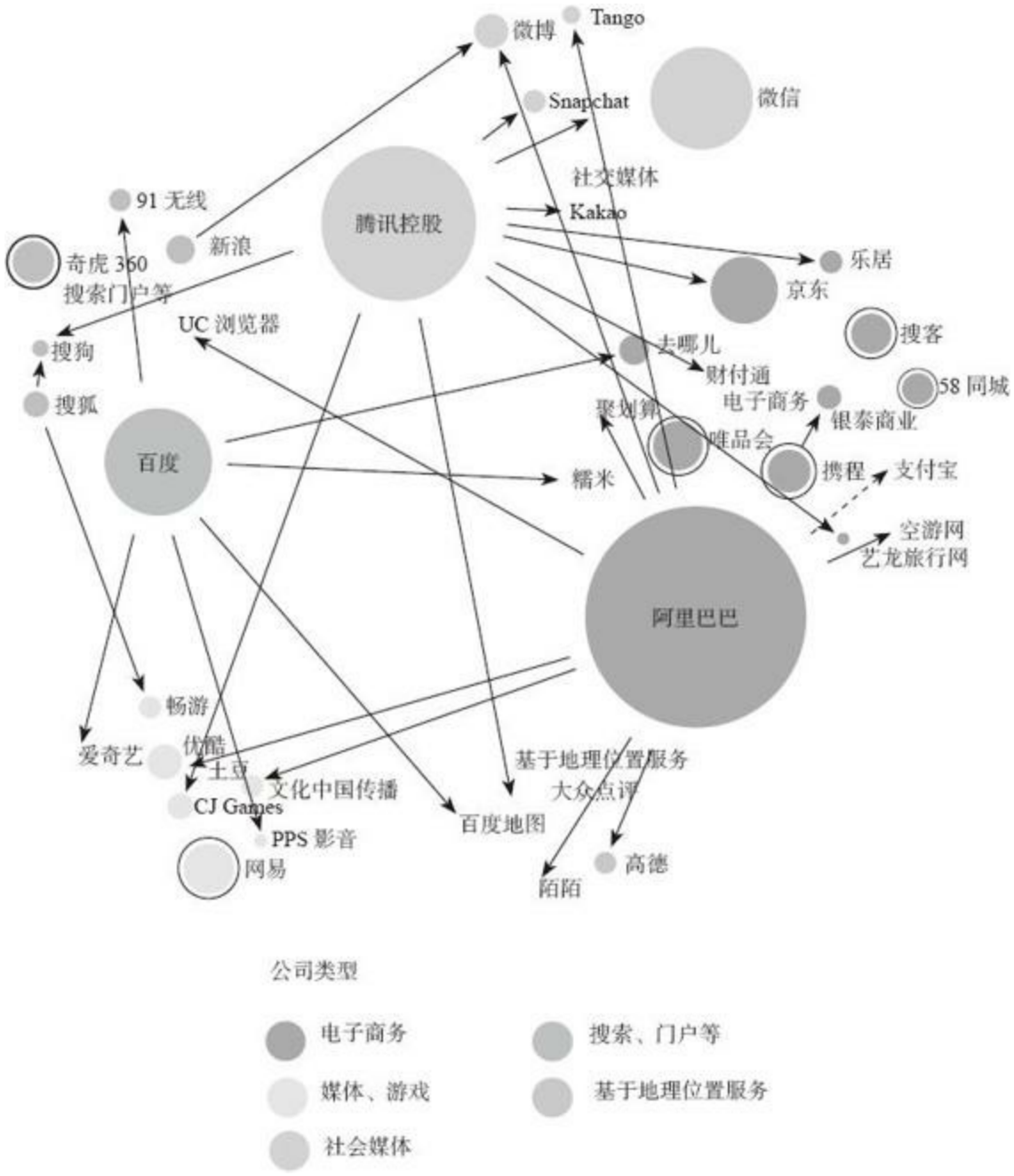

图7-2　互联网公司业务类型

2.典型的工作负载

下面从计算范式角度介绍典型的大数据工作负载。

（1）基本负载

1）Word Count。

WordCount是CPU密集型的操作负载，WordCount已经在前面有所介绍，在此不再详述。

2）Sort。

排序算法是I/O密集型的负载。

排序算法的实现如下。

```
object Sort {
 def main(args: Array[String]): Unit = {
  val host = "Spark://127.0.0.1:7077"  /*指定Spark的主机地址*/
  var splits = 1    /*读者可以自行设定这里的分区个数*/
  val spark = new SparkContext(host, "Sort",
   SPARK_HOME, List(JARS))
  val filename = "SortText"
  val save_file = "SortSavedFile"
  val lines = spark.textFile(filename, splits)
  val mapData = lines.map(line => {
   (line, 1)
  })    /*这里进行映射是为了使用sortByKey的算子，因为sortByKey只能处理key-value pair类型的数据*/
  val result = mapData.sortByKey().map{line => line._1}
  result.saveAsTextFile(save_file)
 }
}
```

3）Tera Sort

在运行的过程中，map映射和Shuffle阶段是CPU密集型的（CPU intensive），I/O程度中等，在reduce阶段是I/O密集型的（I/O intensive），CPU计算中等。

算法实现思想：当把传统的串行排序算法设计成并行的排序算法时，通常会想到分而治之的策略。排序并行化的一般做法是：把要排序的数据划成M个数据块（可以用Hash的方法做到），然后每个map task对一个数据块进行局部排序，之后，一个reduce task对所有数据进行全排序。这种设计思路可以保证在map阶段并行度很高，但在reduce阶段完全没有并行。为了提高reduce阶段的并行度，TeraSort作业对以上算法进行改进：在map阶段，每个map task都会将数据划分成R个数据块（R为reduce task个数），其中第i（i>0）个数据块的所有数据都会比第i+1个数据块中的数据大；在reduce阶段，第i个reduce task处理（进行排序）所有map task的第i块，这样第i个reduce task产生的结果均会比第i+1个大，最后将1~R个reduce task的排序结果顺序输出，即为最终的排序结果。

（2）机器学习

下面以K-Means聚类算法为例介绍机器学习计算范式

在计算中心点时，K-Means是CPU密集型的计算。在聚类时，K-Means进行I/O密集型运算。

K-Means算法是最为经典的基于划分的聚类方法，是十大经典数据挖掘算法之一。K-Means算法的基本思想是：以空间中k个点为中心进行聚类，对最靠近它们的对象归类。通过迭代的方法，逐次更新各聚类中心点的值，直至得到最好的聚类结果。

假设要把样本集分为c个类别，算法描述如下。

1）适当选择k个类的初始中心。

2）在第n次迭代中，对于任意一个样本，求其到k各中心的距离，将该样本归到距离最短的中心所在的类。

3）利用均值等方法更新该类的中心值。

4）对于所有的k个聚类中心，如果利用2）、3）的迭代法更新后，值保持不变。或者达到指定的迭代次数，则迭代结束，否则继续迭代。

该算法的最大优势在于简洁和快速。算法的关键在于初始中心的选择和距离公式。

（3）图计算

下面以PageRank图计算算法为例介绍图计算的计算范式。PageRank广泛用于搜索引擎，对网页图谱进行分析。

1）算法介绍。

PageRank用于衡量特定网页相对于搜索引擎索引中其他网页而言的重要程度，是Google的专有算法。20世纪90年代后期由Larry Page和Sergey Brin开发。PageRank将链接价值概念作为排名因素。

2）算法思想。

一个页面的PageRank由所有链向它的页面（链入页面）的重要性经过递归算法得到。一个页面的权重由所有链向它的页面的重要性决定，到一个页面的超链接相当于为该页投一票。一个有较多链入的页面会有较高的权重，反之，一个页面链入页面越少，权重越低。简而言之，从许多的权重高网页链接过来的网页，必定还是权重高的网页。

PageRank计算基于以下两个基本假设。

·质量假设：指向页面A的入链质量不同，质量高的页面会通过链接向其他页面传递更多的权重。因此，质量越高的页面指向页面A，则页面A也越重要。

·数量假设：若一个页面节点接收到的其他网页指向的入链数量越多，则该页面越重要。

3）算法原理。

初始阶段：网页通过链接关系构建有向图，每个页面设置相同的PageRank值，通过若干轮的迭代计算，得到每个页面最终获得的PageRank值。在每轮迭代中，网页当前的PageRank值不断更新。

迭代：在更新页面PageRank得分的每轮计算中，各页面将其当前的PageRank值平均分配到本页面包含的出链上，每个链接即获得了相应的权值。而每个页面将所有指向本页面的入链所传入的权值进行求和，即可得到新的PageRank得分。且每个页面都获得了更新后的PageRank值时，一轮PageRank计算完成。

（4）计算公式

$$PR(p_i)=\frac{1-d}{n}+d\sum_{pj\in M(i)}\frac{PR(p_j)}{L(j)}$$

PR（pi）：pi页面的PageRank值。

N：所有页面的数量。

pi：不同的网页p1、p2、p3。

M（i）：pi链入网页的集合。

L（j）：pj链出网页的数量。

d：阻尼系数，任意时刻，用户到达某页面后并继续向后浏览的概率。

（1-d=0.15）：表示用户停止点击，随机跳到新URL的概率。

取值范围：0＜d≤1，Google设为0.85。

通过链接关系，就构造出了“转移矩阵”。

（5）SQL

结构化查询语言（structured query language，SQL）是一种数据库查询和程序设计语言，用于存取数据以及查询、更新和管理关系数据库系统。SQL可以大致分为以下几个类型：席查询（Ad-hoc query）、报表查询（Reporting query）、迭代查询（Iterative Query）、星型查询（Star query）等，感兴趣的用户可以查看TPC-DS的介绍进行了解。

7.2.3 度量指标

性能调优的两大利器就是Benchmark和Profile工具，读者可以结合Spark性能调优章节，通过Benchmark和Profile工具，及相应的调优方法对Spark性能调优。Benchmark用压力测试挖掘整个系统的性能状况，而Profile工具最大限度地呈现系统的运行时状态和性能指标，方便用户诊断性能问题和进行调优。

用户在实战中可以采用一些原生的Profile工具，通过以下几个方面对系统性能指标进行度量。

1.工具使用

1）在架构层面：perf、nmon等工具和命令。

2）在JVM层面：btrace、Jconsole、JVisualVM、JMap，JStack等工具和命令。

3）在Spark层面：web ui、console log，也可以通过修改Spark源码打印日志进行性能监控。

2.度量指标

（1）从架构角度进行度量

·浮点型操作密度。

·整数型操作密度。

·指令中断。

·cache命中率（L1 miss、L2 miss、L3 miss）。

·TLB命中。

（2）从Spark系统执行时间和吞吐的角度度量

·Job作业执行时间。

·Job吞吐量。

·Stage执行时间。

·Stage吞吐量。

·Task执行时间。

·Task吞吐量。

（3）从Spark系统资源利用率的角度度量

·CPU在指定时间段的利用率。

·内存在指定时间段的利用率。

·磁盘在指定时间段的利用率。

·网络带宽在指定时间段的利用率。

（4）从扩展性的角度度量

·数据量扩展。

·集群节点数扩展（scale out）。

·单机性能扩展（scale up）。

7.3 Benchmark的使用

下面介绍3个典型的Benchmark的使用，即Hibench、BigDataBench和TPC-DS。

7.3.1 使用Hibench

下面介绍Hibench[1]的使用方法。

1.前期准备

（1）设置HiBench-2.2

下载或者签出HiBench-2.2 benchmark suite，官方网址为https://github.com/intel-hadoop/HiBench/zipball/HiBench-2.2。

（2）设置Hadoop

在运行其他工作负载之前，请确认已经正确安装了Hadoop，所有的工作负载已经在Cloudera Distribution of Hadoop 3 update 4（cdh3u4）and Hadoop version 1.0.3版本的Hadoop上测试通过。

（3）设置Hive[2]

如果需要测试hivebench，则确认实验环境已经安装了Hive，或者使用benchmark中已经打包的Hive 0.9。

（4）针对所有的工作负载配置参数

需要在使用前在bin/hibench-config.sh中配置一些全局变量。

·HADOOP_HOME：Hadoop的安装路径。

·HADOOP_CONF_DIR：Hadoop的配置文件目录，默认为$HADOOP_HOME/conf目录下。

·COMPRESS_GLOBAL：设置是否压缩输入输出数据，0表示不压缩，1表示压缩。

·COMPRESS_CODEC_GLOBAL：设置默认的输入输出压缩方式。

（5）针对每个工作负载参数配置

如果工作负载的目录下游conf/configure.sh文件，则可以通过修改conf/configure.sh来配置每个工作负载，所有数据规模以及和这个工作负载相关的参数都在这个目录配置。

同步所有节点的时间，这在dfsioe是必须做的，其他可做可不做。

2.运行

（1）一起运行多个工作负载

在配置文件conf/benchmarks.lst中定义了当运行/run-all.sh时需要运行的工作负载。文件中的每一行都是一个指定的工作负载.可以用#符号来注释掉不需要运行的负载。

（2）单独运行各个工作负载

也可以单独运行各个工作负载。通常情况下，在每个工作负载的目录下都有3个独立的shell文件，这3个文件的功能如下。

·conf/configure.sh：这个配置文件包含数据规模和测试的运行参数。

·bin/prepare*.sh：生成测试数据或者将输入数据复制到HDFS中。

·bin/run*.sh：执行工作负载。

用户可以按照下面的顺序执行工作负载。

1）配置Benchmark。

如果需要更高级的测试需求，则通过修改配置文件configure.sh来配置参数。

2）准备数据。

通过运行Shell文件bin/prepare.sh生成和准备数据（bin/prepare-read.sh这个文件针对dfsioe）。

3）运行Benchmark。

```
bin/run*.sh
```

[1] https://github.com/intel-hadoop/HiBench。

[2] 这个只在运行Hive的Benchmark时，才需要安装。

7.3.2 使用TPC-DS

下面介绍TPC-DS（注：参见：http://www.tpc.org/tpcds/，TPC Benchmark™DS (TPC-DS):The New Decision Support Benchmark Standard。）的使用方式。

（1）下载最新的包

用户可以到TPC主页进行下载：http://www.tpc.org。

（2）Make生成可执行文件[1]

1）将Makefile.suite文件复制为Makefile。

2）编辑修改Makefile，找到含有“OS=”的行。

3）阅读注释并增加指定的OS，如“OS=LINUX”。

4）执行make。

（3）Windows操作系统编译TPC-DS，生成可执行文件

1）安装Microsoft Visual Studio 2005。

2）双击打开整个解决方案dbgen2.sln（可能会看到报错“project file grammar.vcproj”，可以忽略这个错误）。

3）在项目列表中，右击dbgen2（是数据生成器），然后选择“build”（或者从顶层菜单单击Build→Build Solution）。

4）重复步骤3构建查询生成器qgen2。

5）针对X64 and IA64 on X86平台进行交叉编译，安装Microsoft Visual Studio

2005“Team Suite”SKU，然后选择from Build→Configuration Manager命令，并重复执行步骤3和步骤4，进而修改目标的运行平台。

（4）生成数据

1）运行“dbgen2–h”以获取帮助信息。注意：许多高级选项并不是必须的。

2）在目录/tmp下生成个100GB数据。

```
dbgen2 -scale 100 -dir /tmp
```

常用的数据规模参数可以有100GB、300GB、1TB、3TB、10TB、30TB和100TB。

3）可以通过配置参数“-delimiter‘<c>’”选项修改文件分隔。

4）当数据规模巨大时，可以通过并行方式生成。例如，生成100GB的数据，通过4路并行在Linux/Unix上运行。

```
dbgen2 -scale 100 -dir /tmp -parallel 4 -child 1 &
dbgen2 -scale 100 -dir /tmp -parallel 4 -child 2 &
dbgen2 -scale 100 -dir /tmp -parallel 4 -child 3 &
dbgen2 -scale 100 -dir /tmp -parallel 4 -child 4 &
```

（5）加载数据

在Shark中需要先建表，然后加载数据，加载数据的方式和Hive是相近的。加载数据时候，需要注意使用的分隔符。

注意由于TPC-DS的数据类型在Shark中并不完全适用，所以可以修改数据类型为以下格式。

```
drop table customer_address;
create table customer_address
(
    ca_address_sk bigint ,
    ca_address_id string ,
    ca_street_number string ,
    ca_street_name string ,
```

```
    ca_street_type string ,
    ca_suite_number string ,
    ca_city string ,
    ca_county string ,
    ca_state string ,
    ca_zip string ,
    ca_country string ,
    ca_gmt_offset double ,
    ca_location_type string
)
row format delimited fields terminated by '|' lines terminated by '\n' stored as textfile ;
```

在Shell中执行如下命令加载数据，或者这条命令在SQL中让Shark执行。

```
LOAD DATA INPATH 'hdfs：//hive01：9000/3t/customer.dat' INTO TABLE customer_address；
```

（6）通过模板生成查询

query_templates文件夹下有99个查询模板，用户可以到其中查询模板并生成指定的查询。

由于不同厂家的SQL并不是全部遵循ANSI标准。例如“LIMIT”和“BEGIN/COMMIT”，qgen2需要指定方言“dialect”。现在支持几类模板：db2.tpl、netezza.tpl、oracle.tpl、sqlserver.tpl。下面是生成oracle方言，针对100GB数据规模，使用query99模板查询的例子。

```
qgen2 -query99.tpl -directory query_templates
-dialect oracle -scale 100
```

（7）运行查询

查询的运行依赖于当时正在运行的大数据系统。

Shark执行查询的命令和方式如下。

1）在Shark的Shell中执行，将query的SQL语句复制到Shell执行。

```
$ ./bin/shark # Start CLI for interactive session
```

2）在命令行后追加SQL语句执行查询。

```
$ ./bin/shark -e "SELECT * FROM foo"  # Run a specific query and exit
```

3）在命令行后追加文件执行查询。

```
$ ./bin/shark -i queries.hql # Run queries from a file
```

关于其他的高级选项，感兴趣的用户可以通过查看文档了解。

[1] 针对AIX、LINUX、HPUX、NCR和Solaris操作系统。

7.3.3 使用BigDataBench

BigDataBench[1]针对不同的负载，用户可以使用下面的方式生成数据和使用工作负载。

在Spark的Shell中已经默认设置了数据生成的命令和配置，感兴趣的读者或者有特定数据规模需求的读者可以修改对应的Shell调整负载。

1.离线分析（offline analytics）

Spark版本的工作负载包含3个：sort、grep、wordcount。

（1）sort

用户在运行之前需要将Spark复制到各个节点。

1）准备工作。

在官网下载指定的包后，解压压缩包。

```
 BigDataBench_Sprak_V3.0.tar.gz
tar -xzf BigDataBench_Sprak_V3.0.tar.gz
```

2）打开目录。

```
cd BigDataBench_Sprak_V3.0.tar.gz /MicroBenchmarks/
```

3）生成数据。

```
sh genData_MicroBenchmarks.sh
sh sort-transfer.sh
```

4）运行。

注意：如果运行的负载是排序，则需要提前运行sh文件sort-transfer进行转换。

第一步：sh genData_MicroBenchmarks.sh。

第二步：sh sort-transfer.sh。

第三步：运行。

```
./run-bigdatabench cn.ac.ict.bigdatabench.Sort <master> <data_file> <save_file> [<slices>]
```

参数介绍：

·<master>：Spark服务器的URL，例如：spark://172.16.1.39：7077。

·<data_file>：输入数据的HDFS路径，例如：/test/data.txt。

·<save_file>：保存结果的HDFS路径。

·[<slices>]：可选参数（worker数量）。

（2）执行grep负载

```
./run-bigdatabench cn.ac.ict.bigdatabench.Grep <master> <data_file> <keyword> <save_file>
[<slices>]
```

参数介绍：

·<master>：Spark服务器的URL，例如：spark://172.16.1.39：7077。

·<data_file>：输入数据的HDFS路径，例如：/test/data.txt。

·<keyword>：过滤文本用的关键词。

·<save_file>：保存结果用的HDFS路径。

·[<slices>]：可选参数，Worker数量。

（3）执行wordcount负载

```
./run-bigdatabench cn.ac.ict.bigdatabench.WordCount <master> <data_file> <save_file>
[<slices>]
```

参数介绍：

·<master>：Spark服务器的URL，例如：spark://172.16.1.39：7077。

·<data_file>：输入数据的HDFS路径，例如：/test/data.txt。

·<save_file>：保存结果的HDFS路径。

·[<slices>]：可选参数，worker数量。

2.分析型的工作负载

（1）PageRank

PageRank的程序和数据源自Hibench。

```
Spark-version
```

1）前期准备。

下载并解压文件：BigDataBench_Sprak_V3.0.tar.gz。

```
tar xzf BigDataBench_Sprak_V3.0tar.gz
```

2）打开目录。

```
cd BigDataBench_Sprak_V3.0.tar.gz /SearchEngine/ Pagerank
```

3）生成数据。

```
sh genData_PageRank.sh
```

4）运行。

```
./run-bigdatabenchorg.apache.spark.examples.PageRank
<master> <file> <number_of_iterations> <save_path> [<slices>]
```

参数介绍：

·<master>：Spark服务器的URL，例如：spark://172.16.1.39：7077。

·<file>：输入数据的HDFS路径，例如：/test/data.txt。

·<number_of_iterations>：运行算法的迭代次数。

·<save_path>：保存结果的路径。

·[<slices>]：可选参数，worker数量。

（2）Kmeans

Kmeans的工作负载来源于Mahout。

1）准备工作。

下载并解压包：BigDataBench_Sprak_V3.0.tar.gz。

```
tar -xzf BigDataBench_Sprak_V3.0.tar.gz
```

2）打开根目录。

```
cd BigDataBench_Sprak_V3.0.tar.gz /SNS
```

3）生成数据。

```
sh genData_ Kmeans.sh
```

4）运行。

```
./run-bigdatabench org.apache.spark.mllib.clustering.KMeans <master> <input_file> <k>
<max_iterations> [<runs>]
```

参数介绍：

·<master>：Spark服务器的URL，例如：spark://172.16.1.39：7077。

·<input_file>：输入数据的HDFS路径，例如：/test/data.txt。

·[<k>]：数据中心的数量。

·<max_iterations>：运行算法的迭代次数。

·[<runs>]：通过上述命令即可进行kMears基准测试。

（3）Connected Components

Connected Components程序源自PEGASUS。

1）准备工作。

①下载并解压包：BigDataBench_Sprak_V3.0.tar.gz。

```
tar xzf BigDataBench_Sprak_V3.0.tar.gz
```

②打开根目录。

```
cd BigDataBench_Sprak_V3.0.tar.gz/ SNS/connect/
```

③生成数据。

```
sh genData_ connectedComponents.sh
```

2）运行。

```
./run-bigdatabench cn.ac.ict.bigdatabench.ConnectedComponent <master> <data_file> [<slices>]
```

参数介绍：

·<master>：Spark服务器的URL，例如：spark://172.16.1.39：7077。

·<data_file>：输入数据的HDFS路径，例如：/test/data.txt。

·[<slices>]：可选参数，worker数量。

通过上述命令即可运行Connected Components负载。

（4）Naive Bayes

Naive Bayes（朴素贝叶斯）算法也是源自Mahout。通过下面命令进行数据生成与测试。

1）下载并解压包：BigDataBench_Sprak_V2.2.tar.gz。

```
tar -xzf BigDataBench_Sprak_V2.2.tar.gz。
```

2）打开根目录。

```
cd BigDataBench_Sprak_V2.2.tar.gz / E-commerce
```

3）生成数据。

```
sh genData_ naivebayes.sh
```

4）运行。

```
sh run_naivebayes.sh
```

[1] 参见：http://prof.ict.ac.cn/BigDataBench/，A Big Data Benchmark Suite,ICT,Chinese Academy of Sciences。

7.4 本章小结

本章主要介绍了大数据Benchmark，包括Benchmark的原理和常用Benchmark的使用。

Benchmark标准尚未形成统一，但一些Benchmark已经崭露头角。用户可以根据系统需求有针对性地选用。Benchmark包含三大组件，读者通过了解三大组件可以理解Benchmark的原理和作用。

最后本章介绍了Hibench、BigDataBench、TPC-DS这三个广泛使用的Benchmark的使用方法，读者可以采用需要的Benchmark进行实践。

相信通过之前几章的介绍，读者已经对Spark有了一定程度的了解。Spark发展得如火如荼发展的一个重要原因就是生态系统的完善，下面通过介绍BDAS的主要组件，使读者全面了解Spark生态系统。

第8章　BDAS简介

随着Spark中国峰会的举行，Spark工业界应用的大范围落地，Spark生态系统在国内发展势头强劲。前段时间Spark也正式升级为Apache顶级项目，证明Spark得到了更加广泛的认可。AMPLab的Spark团队创立了大数据公司Databricks，提供Spark的产品化支持，为后续Spark的产品化和落地提供了更加强有力的保障。

提到Spark就不得不说伯克利大学AMPLab开发的BDAS（Berkeley Data Analytics Stack）数据分析的软件栈。其中用内存分布式大数据计算引擎Spark替代原有的MapReduce，上层通过Spark SQL/Shark替代Hive等数据仓库，Spark Streaming替换Storm等流式计算框架，GraphX替换Graph Lab等大规模图计算框架，MLlib替换Mahout等机器学习框架等，其整体框架基于内存计算解决了原来Hadoop的性能瓶颈问题。他们提出One Framework to Rule Them All的理念，用户可以利用Spark一站式构建自己的数据分析流水线。

8.1 SQL on Spark[1]

AMPLab将大数据分析负载分为三大类型：批量数据处理、交互式查询、实时流处理，而其中很重要的一环便是交互式查询。大数据分析栈中需要满足用户ad-hoc、reporting、iterative等类型的查询需求，需要提供SQL接口来兼容原有数据库用户的使用习惯，同时需要SQL能够重组关系模式。完成这些重要的SQL任务的便是Spark SQL和Shark这两个开源分布式大数据查询引擎，它们可以理解为轻量级Hive SQL在Spark上的实现，业界将该类技术统称为SQL on Hadoop。

在刚刚结束的Spark Summit 2014上，Databricks宣布不再支持Shark的开发，全力以赴开发Shark的下一代技术Spark SQL，同时Hive社区也启动了Hive on Spark项目，将Spark作为Hive除MapReduce和Tez之外的新执行引擎。根据伯克利的Big Data Benchmark测试对比数据，Shark的In Memory性能可以达到Hive的100倍，即使是On Disk，也能达到10倍的性能提升，是Hive强有力的替代解决方案。而作为Shark进化版本的Spark SQL，在AMPLab最新的测试中，性能已经超过Shark。在本文中，统称Spark SQL、Shark和Hive on Spark为SQL on Spark。虽然Shark不再开发，但其架构和优化仍有借鉴意义，因此也会在文章中有所介绍。图8-1展示了Spark SQL和Hive on Spark是新的发展方向。

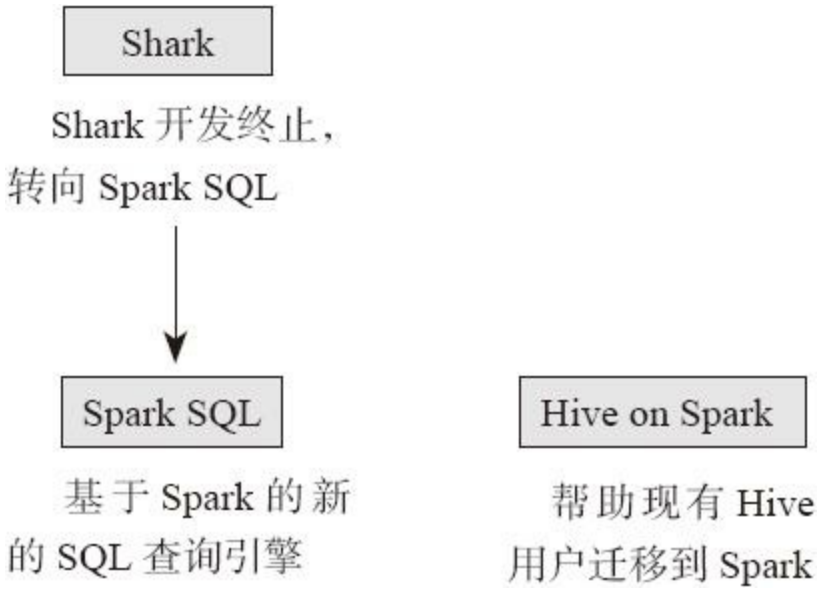

图8-1　Spark SQL和Hive on Spark是新的发展方向

[1] 参考文章：高彦杰，陈冠诚所写的《Spark SQL：基于内存的大数据分析引擎》，《程序员》，2014.8。

8.1.1 使用Spark SQL的原因

由于Shark底层依赖于Hive，这个架构的优势是对传统Hive用户可以将Shark无缝集成进现有系统运行查询负载。但是我们也看到一些问题：随着版本的升级，查询优化器依赖于Hive，不方便添加新的优化策略，需要学习另一套系统和进行二次开发，学习成本很高。另一方面，MapReduce是进程级并行。例如，Hive在不同的进程空间会使用一些静态变量，当在同一进程空间并行执行多线程时，多线程同时写同名称的静态变量会产生一致性问题，因此Shark需要使用另外一套独立维护的Hive源码分支。为了解决这个问题，AMPLab和Databricks利用Catalyst开发了Spark SQL。在Spark 1.0版本中已经发布Spark SQL。

Spark的Full Stack解决方案为用户提供了多样的数据分析框架，机器学习、图计算、流计算如火如荼地发展和流行吸引了大批的学习者，那么为什么我们今天还是要重视在大数据环境下使用SQL呢？笔者认为主要有以下几点原因。

1）易用性与用户惯性。在过去的很多年中，有大批程序员的工作是围绕DB+应用的架构来做的，因为SQL的易用性提升了应用的开发效率。程序员已经习惯了采用业务逻辑代码调用SQL的模式去写程序，惯性的力量是强大的，如果还能用原有的方式解决现有的大数据问题，何乐而不为呢？提供SQL和JDBC的支持会让传统用户像以前一样书写程序，大大减少迁移成本。

2）生态系统的力量。很多系统软件性能好，但是未取得成功便没落了，很大程度上是由于生态系统问题。传统SQL在JDBC、ODBC、SQL的各种标准下形成了一整套成熟的生态系统，很多应用组件和工具可以迁移使用，像一些可视化的工具、数据分析工具等，原有企业的IT工具可以无缝过渡。

3）数据解耦。Spark SQL正在扩展支持多种持久化层，用户可以使用原有的持久化层存储数据，但是也可以体验和迁移到Spark SQL提供的数据分析环境下分析Big Data。

8.1.2 Spark SQL架构分析

Spark SQL与传统DBMS的查询优化器+执行器的架构较为类似，只不过其执行器是在分布式环境中实现，并采用Spark作为执行引擎。Spark SQL的查询优化是Catalyst，其基于Scala语言开发，可以灵活利用Scala原生的语言特性方便地扩展功能，奠定了Spark SQL的发展空间。Catalyst将SQL翻译成最终的执行计划，并在这个过程中进行查询优化。这里和传统不太一样的地方就在于，SQL经过查询优化器最终转换为可执行的查询计划，传统DB就可以执行这个查询计划了。而Spark SQL最后执行还是会在Spark内将执行计划转换为Spark的有向无环图DAG再执行。

1.Catalyst架构及执行流程分析

Catalyst的整体架构如图8-2所示。

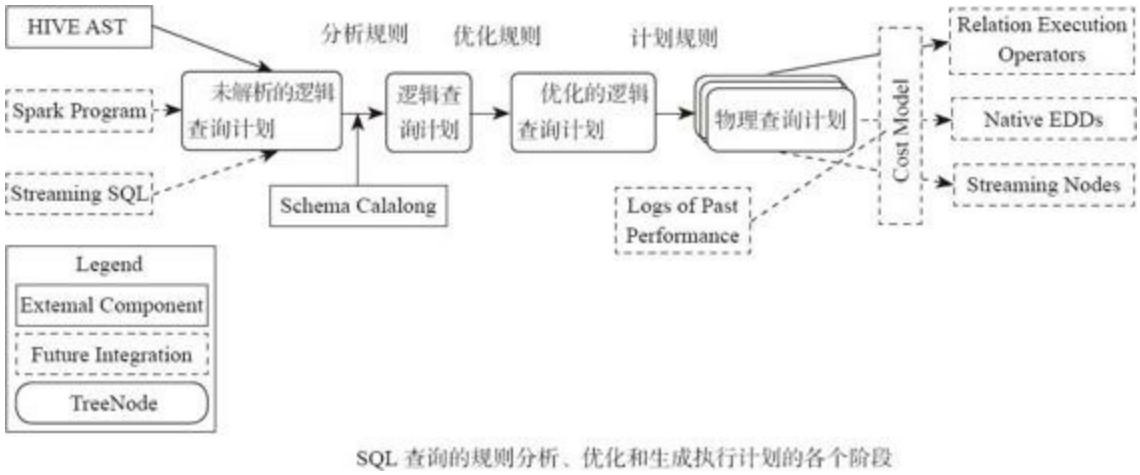

图8-2　Spark SQL查询引擎Catalyst的架构

从图8-2中可以看到整个Catalyst是Spark SQL的调度核心，遵循传统数据库的查询解析步骤，对SQL进行解析，转换为逻辑查询计划和物理查询计划，最终转换为Spark的DAG执行。Catalyst的执行流程如图8-3所示。

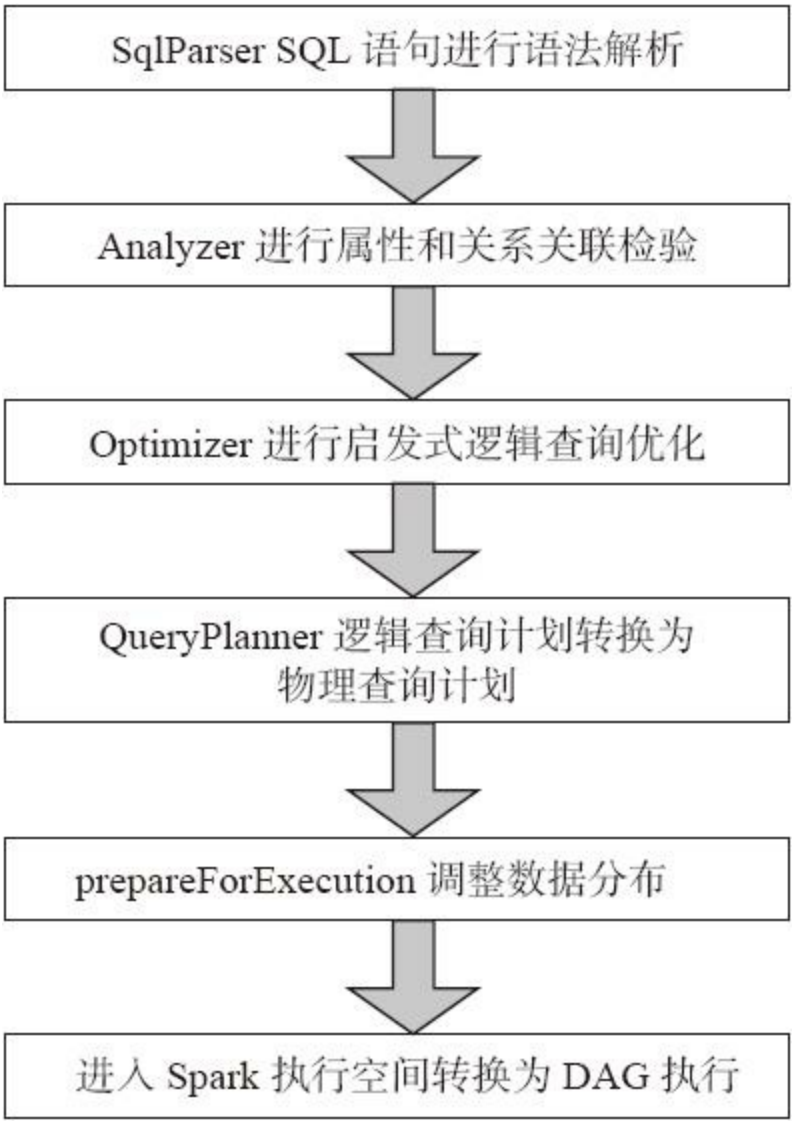

图8-3　Catalyst的执行流程

SqlParser将SQL语句转换为逻辑查询计划，Analyzer对逻辑查询计划进行属性和关系关联检验，之后Optimizer通过逻辑查询优化将逻辑查询计划转换为优化的逻辑查询计划，QueryPlanner将优化的逻辑查询计划转换为物理查询计划，prepareForExecution调整数据分布，最后将物理查询计划转换为执行计划进入Spark执行任务。

2.Spark SQL优化策略

查询优化是传统数据库中最为重要的一环，这项技术在传统数据库中已经很成熟。除了查询优化，Spark SQL在存储上也进行了优化，下面介绍Spark SQL的一些优化策略。

（1）内存列式存储与内存缓存表

Spark SQL可以通过cacheTable将数据存储转换为列式存储，同时将数据加载到内存缓存。cacheTable相当于在分布式集群的内存物化视图，将数据缓存，这样迭代的或者交互式的查询不用再从HDFS读数据，直接从内存读取数据大大减少了I/O开销。列式存储的优势在于Spark SQL只需要读出用户需要的列，而不需要像行存储那样每次都将所有列读出，从而大大减少内存缓存数据量，更高效地利用内存数据缓存，同时减少网络传输和I/O开销。数据按照列式存储，由于是数据类型相同的数据连续存储，所以能够利用序列化和压缩减少内存空间的占用。

（2）列存储压缩

为了减少内存和硬盘空间占用，Spark SQL采用了一些压缩策略对内存列存储数据进行压缩。Spark SQL的压缩方式要比Shark丰富很多，如它支持PassThrough、RunLengthEncoding、DictionaryEncoding、BooleanBitSet、IntDelta、LongDelta等多种压缩方式，这样能够大幅度减少内存空间占用、网络传输和I/O开销。

（3）逻辑查询优化

SparkSQL在逻辑查询优化（见图8-4）上支持列剪枝、谓词下压、属性合并等逻辑查询优化方法。列剪枝为了减少读取不必要的属性列、减少数据传输和计算开销，在查询优化器进行转换的过程中会优化列剪枝。

下面介绍一个逻辑优化的例子。

```
SELECT Class FROM (SELECT ID,Name,Class  FROM STUDENT ) S WHERE S.ID=1
```

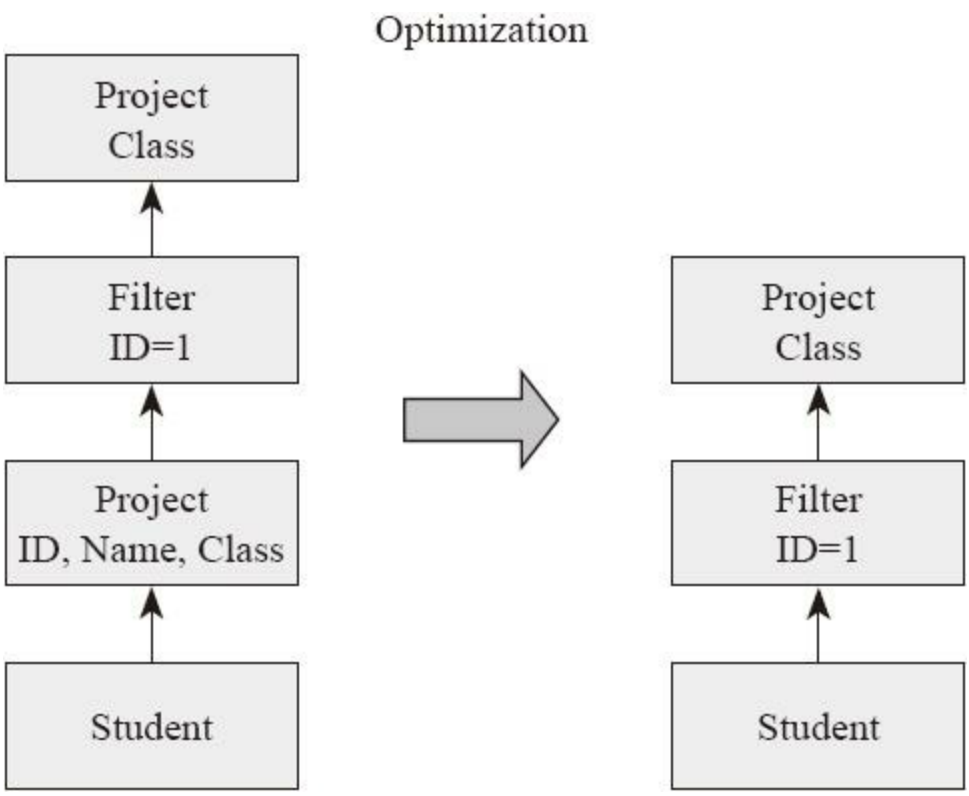

图8-4　逻辑查询优化

Catalyst将原有查询通过谓词下压，将选择操作ID=1优先执行，这样过滤大部分数据，通过属性合并将最后的投影只做一次，最终保留Class属性列。

（4）Join优化

Spark SQL深度借鉴传统数据库的查询优化技术的精髓，同时在分布式环境下调整和创新特定的优化策略。现在Spark SQL对Join进行了优化，支持多种连接算法，现在的连接算法已经比Shark丰富，而且很多原来Shark的元素也逐步迁移过来，如BroadcastHashJoin、BroadcastNestedLoopJoin、HashJoin、LeftSemiJoin，等等。

下面介绍其中的一个Join算法。

BroadcastHashJoin将小表转化为广播变量进行广播，这样避免Shuffle开销，最后在分区内做Hash连接。这里使用的就是Hive中Map Side Join的思想，同时使用DBMS中的Hash连接算法做连接。

随着Spark SQL的发展，未来会有更多的查询优化策略加入进来，同时后续Spark SQL会

支持像Shark Server一样的服务端和JDBC接口，兼容更多的持久化层，如NoSQL、传统的DBMS等。一个强有力的结构化大数据查询引擎正在崛起。

3.如何使用Spark SQL

下面给出使用Spark SQL的示例。

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
/* 在这里引入sqlContext下的所有方法，就可以直接用SQL方法查询*/
import sqlContext._
case class Person(name: String, age: Int)
/* 下面的people是含有case类型数据的RDD，默认由Scala的implicit机制将RDD转换为SchemaRDD，SchemaRDD是SparkSQL中的核心RDD*/
val people = sc.textFile("examples/src/main/resources/people.txt").map(_.split(",")).map(p =>
Person(p(0), p(1).trim.toInt))
/* 在内存的元数据中注册表信息，这样一个Spark SQL表就创建完成了*/
people.registerAsTable("people")
/* SQL语句就会触发上面分析的Spark SQL的执行过程
val teenagers = sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
/* 最后生成的teenagers也是一个RDD*/
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

下面将介绍Shark，虽然Shark已经完成学术使命终止开发，但是其中的架构和优化策略还是有借鉴意义的。

8.1.3 Shark简介

下面介绍Shark的架构，如图8-5所示。在整体架构中，Shark复用了Hive Metastore、Hive SerDe，以及查询解析器和优化器，但是用Spark重写了Hive的执行Operator，并实现了基于内存的优化策略。最初Shark为了学术使命，复用Hive的查询优化器，虽然缩短了开发周期，但是这样不得不维护一个单独的Hive分支用来支持Shark，随着系统复杂性的提升，优化策略的不断扩充，维持Hive的查询优化器已经代价太大，最终Databricks宣布终止Shark开发。

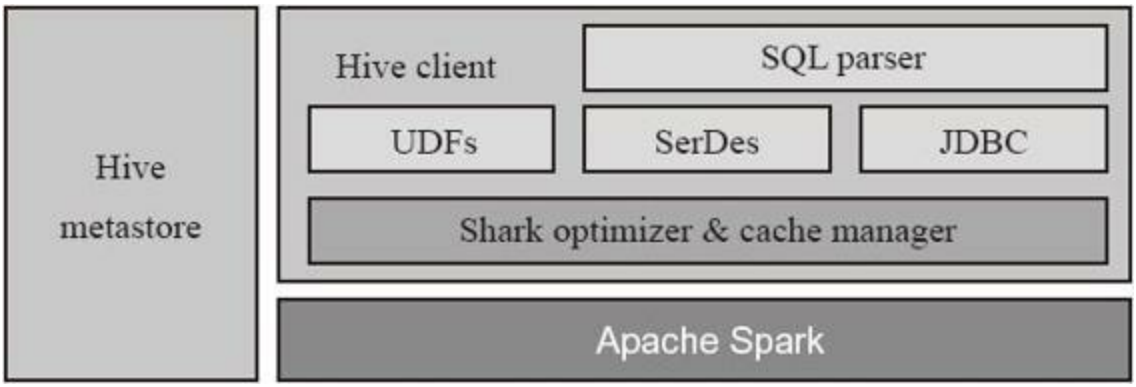

图8-5 Shark架构

1.执行流程

Shark读取用户的查询表达式，运用Hive的解析器和查询优化器形成查询树进行语法解析和逻辑物理优化，最终形成等待执行的执行计划。执行器遍历执行计划树到叶子节点的Operator执行，执行后再回溯到父母节点继续执行，直到完全执行完整个查询计划。Operator中不再用Hadoop的MapReduce进行分布式计算，而是用Spark重写Operator进行分布式计算。

2.容错性

Spark记录了RDD的Lineage，即RDD的依赖关系，功能类似于传统数据库中的redo日志的功能。当有分区丢失或者出错时，Spark可以从源头的基础数据重做运算恢复分区数据。

这也是和Impala进行对比的一个优势，Impala如果任务失败，则需要整体重做全部任务。

3.多数据计算范式混合

Shark和其他SQL on Hadoop产品对比的一大优势还源于其可以和其他多种计算范式混合计算。使用Shark通过SQL建立内存表，既可以通过MLlib进行Machine Learning的运算，又可以用GraphX进行大规模图计算，等等，使用户方便地进行一站式数据流水线计算，而不需要有一个持久化层，如HDFS暂存中间数据。无疑会大幅度减少性能开销，同时提升开发效率和复杂度，更减少了不同系统间兼容的代价。

下面看一个SQL和机器学习结合的例子。

进入Spark Shell进行交互式查询，方便用户迅速实现想法。

```
$./bin/shark-shell
/* 通过SparkContext的SQL2rdd方法运行SQL查询，从Shark中读出表并在内存建立RDD */
scala> val youngUsers = sc.SQL2rdd("SELECT * FROM users WHERE age < 20")
/* 对内存youngUsers的RDD进行map，将数据提取要进行Machine Learning的feature*/
scala> val featureMatrix = youngUsers.map(extractFeatures(_))
/* 调用MLlib中的kmeans进行用户数据聚类*/
scala> kmeans(featureMatrix)
```

我们看到通过短短几行代码就实现了SQL和机器学习的运算需求。

4.性能对比

如图8-6所示，从最新的伯克利Big Data Benchmark上的一个例子可以看到：这个查询的执行首先解析每个元组的应用字符串，然后进行一次高基数的聚集函数运算。由于UserVistits表有些列没有用到，Redshift的列存储只读需要的数据体现了优势。同时Shark在内存也是基于列存储，从两者的对比看来，Shark的性能瓶颈在于字符串解析。再看Impala，由于Impala是从操作系统的cache读数据，它就需要读和解压缩整个行，造成和Shark相比有一定的劣势，但是Impala相比Shark应用了更加高效的编译后的执行代码，比Shark有一定的优势，这两个因素造成Impala和Shark达到差不多的处理内存表的吞吐量。对大的结果集来说，Impala会由于物化输出表造成更高的延迟。

```
Aggregation Query
SELECT SUBSTR (sourceIP, 1, X), SUM (adRevenue) FROM uservisits GROUP BY SUBSTR (sourceIP, 1, X)
```

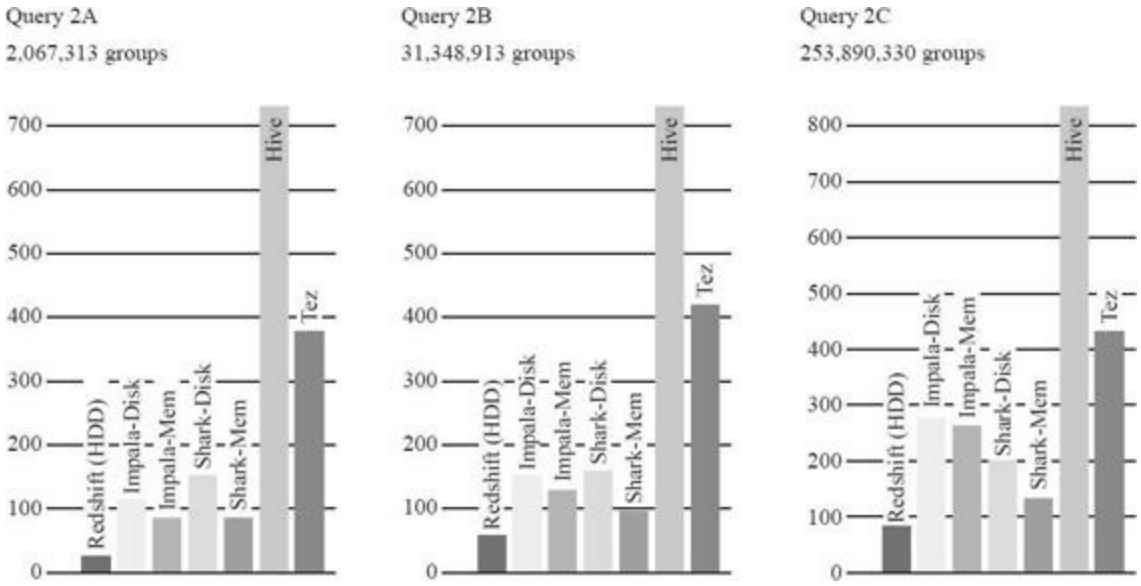

图8-6　Shark等SQL on Hadoop测试对比

综合看来，Spark SQL和Shark相比其他SQL on Hadoop产品存在以下几点优势。

1）依托内存计算框架Spark，利用内存计算大幅度提升性能。

2）支持Spark Shell进行交互式查询，使用户想法可以快速实现。

3）依托Spark生态系统，可以方便地构建全栈数据解决方案。

8.1.4 Hive on Spark

随着Hive on Spark的立项，未来的Hive会支持MapReduce、Tez和Spark三大执行引擎。相比Shark和Spark SQL，Hive on Spark会全面支持现有Hive，也就意味着原来使用Hive的用户可以无缝地过渡，数据不需要迁移，原来针对业务逻辑写的Hive QL脚本不需要重写，只是换了个更加快速的执行引擎。语法上支持全部的Hive QL语法和扩展特性，同时会集成Hive的权限管理、运行监控、审计和其他基于Hive的管理工具。基于Hive生态系统的组件可以过渡到Spark执行引擎中使用。由于都是基于Spark作为执行引擎，上层做的优化难说谁比谁有多大的性能优势，生态系统的力量将是最重要的决定因素。

Hive on Spark设计方向及潜在问题如下。

1）数据表以RDD方式存储。

2）Shuffle和Join：由于Spark的Shuffle不进行分组排序，所以Hive的Join基于MapReduce的Shuffle来做MapSideJoin和ReduceSideJoin，Spark社区已经开始发起改变或者提供相应的Shuffle API，同时原有的Hive Join算法会迁移过来。

3）线程安全问题：Spark执行任务和分区是在一个JVM空间执行多线程，而传统Hive的Map端操作树或者Reduce端操作树将任务的每个线程分在不同的JVM，因为Hive的操作中有静态变量，这样就会产生并发和线程安全问题。这里也是需要重新设计的地方。

4）Java API：Hive on Spark需要社区提供Job监控和RDD扩展的API，这样就能够和原有组件融合。原有组件可以更容易地使用Spark。

感兴趣的读者可以在JIRA上了解这个项目[1]。

[1] https://issues.apache.org/jira/browse/HIVE-7292。

8.1.5 未来展望

Spark SQL提供了对RDD的SQL支持，同时支持其他数据源，如Parquet文件和Hive表。统一这些强大的数据存储模型能够让用户更加方便地分析复杂的数据。统一的Spark数据平台能够让用户选择需要的工具去处理数据，而不需要再构建另一套系统。未来Databricks会继续在Spark SQL生成自定义字节码加速解析表达式，支持更多数据源，如Avro、Hbase以及更丰富的其他语言API。

Databricks和AMPLab会继续投资Spark SQL，希望使其成为结构化数据分析的标准。Shark已经完成学术使命退出历史舞台，Hive on Spark刚刚发起。Spark SQL、Shark、Hive on Spark扮演了Spark生态系统中SQL on Hadoop这个重要的角色，为Spark生态系统完备性提供强有力的支持。同时看到随着Spark生态系统的发展及壮大，三者也从中受益，用户通过全栈的数据分析栈开源节流，会越来越接纳和采用Spark的全栈式解决方案，这样用户也会越来越多地采用SQl on Spark作为自身的OLAP解决方案。这一切迹象表明，未来SQL on Spark的应用和发展会很有想象空间。

8.2 Spark Streaming

Spark Streaming是一个批处理的流式计算框架。它的核心执行引擎是Spark，适合处理实时数据与历史数据混合处理的场景，并保证容错性。下面将详细介绍Spark Streaming。

8.2.1 Spark Streaming简介

Spark Streaming是构建在Spark上的实时流计算框架，扩展了Spark流式大数据处理能力。Spark Streaming将数据流以时间片为单位分割形成RDD，使用RDD操作处理每一块数据，每块数据（也就是RDD）都会生成一个Spark Job进行处理，最终以批处理的方式处理每个时间片的数据，如图8-7所示。

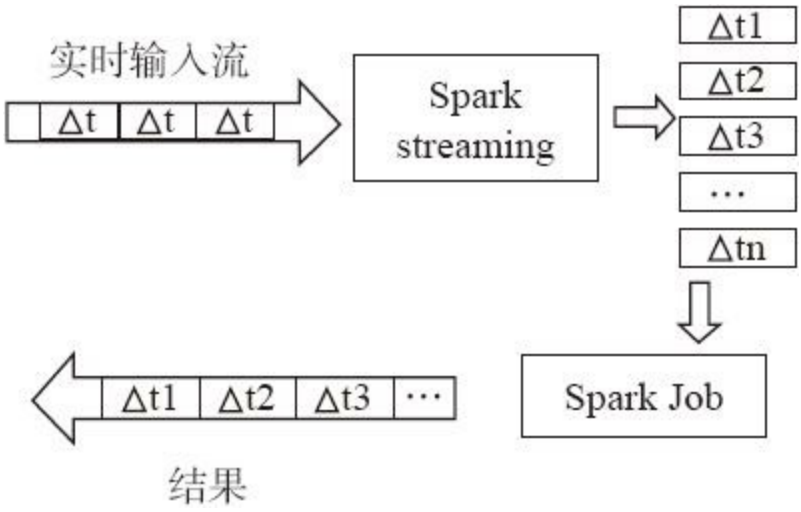

图8-7 Spark Streaming生成Job

Spark Streaming编程接口和Spark很相似。在Spark中，通过在RDD上进行Transformation（如map、filter等）和Action（如count、collect等）算子运算。在Spark Streaming中通过在DStream（表示数据流的RDD序列）上进行算子运算。图8-8为Spark Streaming转化过程。

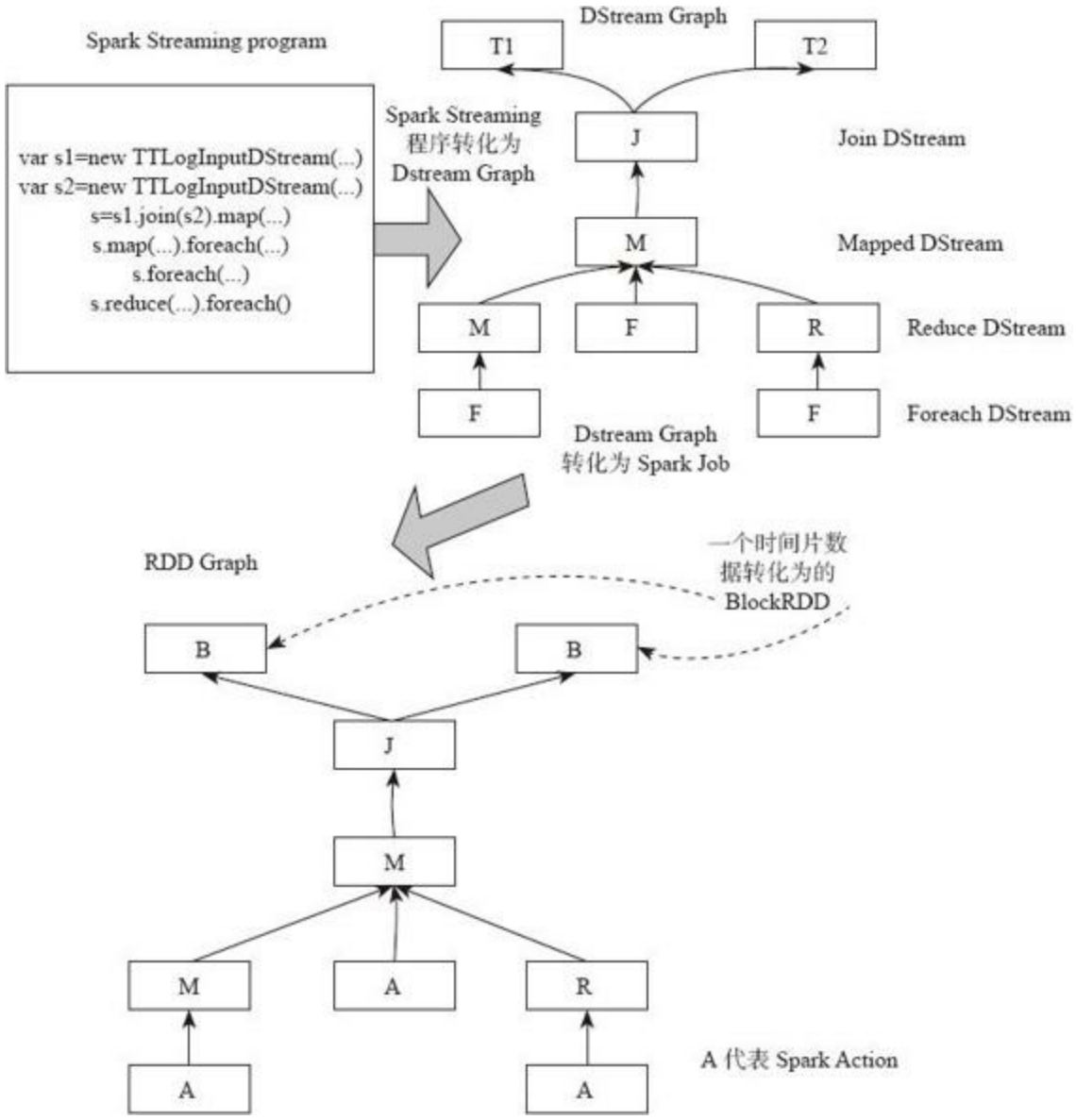

图8-8　Spark Streaming转化过程

图8-8中的Spark Streaming将程序中对DStream的操作转换为DStream有向无环图（DAG）。对于每个时间片，DStream DAG会产生一个RDD DAG。在RDD中通过Action算子触发一个Job，Spark Streaming将Job提交给JobManager。JobManager将Job插入维护的Job队列，JobManager将队列中的Job逐个提交给Spark DAGScheduler，Spark调度Job，并将Task分发到各节点的Executor上执行。

（1）优势及特点

1）多范式数据分析管道：能和Spark生态系统其他组件融合，实现交互查询和机器学习等多范式组合处理。

2）扩展性：可以运行在100个节点以上的集群，延迟可以控制在秒级。

3）容错性：使用Spark的Lineage及内存维护两份数据进行备份达到容错。RDD通过Lineage记录下之前的操作，如果某节点在运行时出现故障，则可以通过冗余备份数据在其他节点重新计算得到。

对于Spark Streaming来说，其RDD的Lineage关系如图8-9所示，图中的每个长椭圆形表示一个RDD，椭圆中的每个圆形代表一个RDD中的一个分区（partition），图中每一列的多个RDD表示一个DStream（图中有3个DStream），t = 1和t = 2代表不同分片下的不同RDD DAG。可以看到图中的每一个RDD都是通过Lineage相连接形成了DAG，由于Spark Streaming输入数据可以来自于磁盘，如HDFS（通常由三份副本），也可以来自于网络，Spark Streaming会将网络输入的数据中的每一个数据流复制两份到其他的机器。这些数据都能通过冗余数据及Lineage的重算机制保证容错性。所以RDD中的任意Partition出错，都可以并行地在其他机器上将缺失的Partition重算出来。

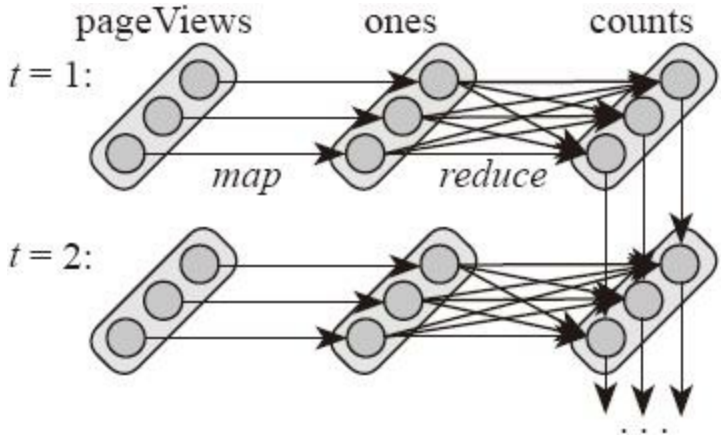

图8-9　Spark Streaming容错性

4）吞吐量大：将数据转换为RDD，基于批处理的方式，提升数据处理吞吐量。图8-10是Berkeley利用WordCount和Grep两个用例所做的测试。

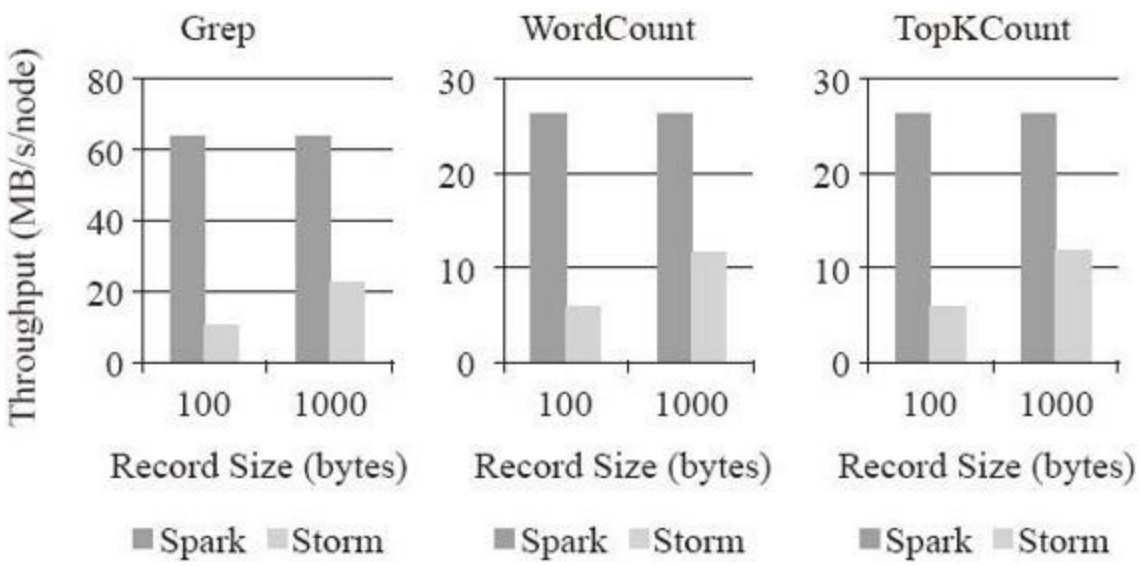

图8-10　Spark Streaming与Storm吞吐量的比较

5）实时性：Spark Streaming也是一个实时计算框架，Spark Streaming能够满足除对实时性要求非常高（如高频实时交易）之外的所有流式准实时计算场景。目前选取Spark Streaming最小的Batch Size在0.5~2s（对比：Storm目前最小的延迟是100ms左右）。

（2）适用场景

Spark Streaming适合需要历史数据和实时数据结合进行分析的应用场景，对于实时性要求不是特别高的场景也能够胜任。

8.2.2 Spark Streaming架构

Spark Streaming的整体架构如图8-11所示。

组件介绍如下。

·Network Input Tracker：通过接收器接收流数据，并将流数据映射为输入DStream。

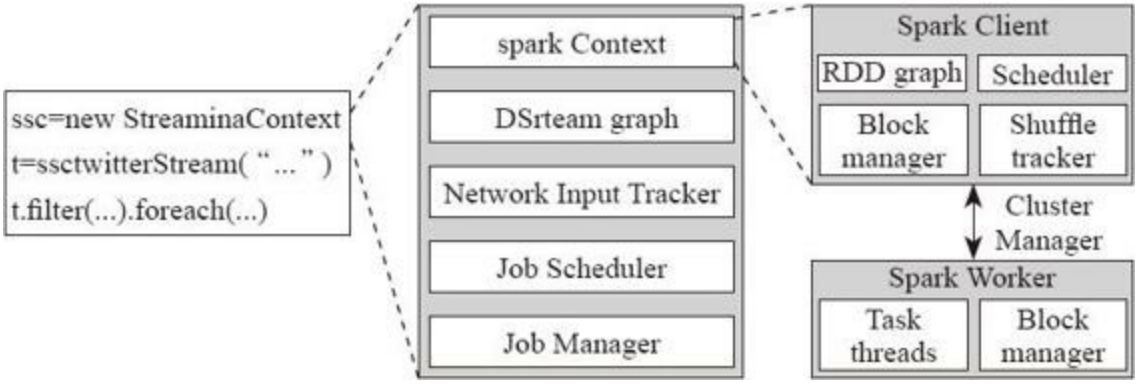

图8-11　Spark Streaming架构图

·Job Scheduler：周期性地查询DStream图，通过输入的流数据生成Spark Job，将Spark Job提交给Job Manager执行。

·JobManager：维护一个Job队列，将队列中的Job提交到Spark执行。

通过图8-11可以看到Job Scheduler负责作业调度，Taskscheduler负责分发具体的任务，Block tracker进行块管理。在从节点，如果是通过网络输入的流数据，则将数据存储两份进行容错。Input receiver源源不断地接收输入流，Task execution负责执行主节点分发的任务，Block manager负责块管理。Spark Streaming的整体架构和Spark很相近，很多思想是可以迁移理解的。

8.2.3 Spark Streaming原理剖析

下面将通过一个example示例的源码呈现Spark Streaming的底层机制。示例及源码基于Spark 1.0版本，后续的发布版中可能会有更新。

1.初始化与集群上分布接收器

图8-12所示为Spark Streaming执行模型从中可看到数据接收及组件间的通信。

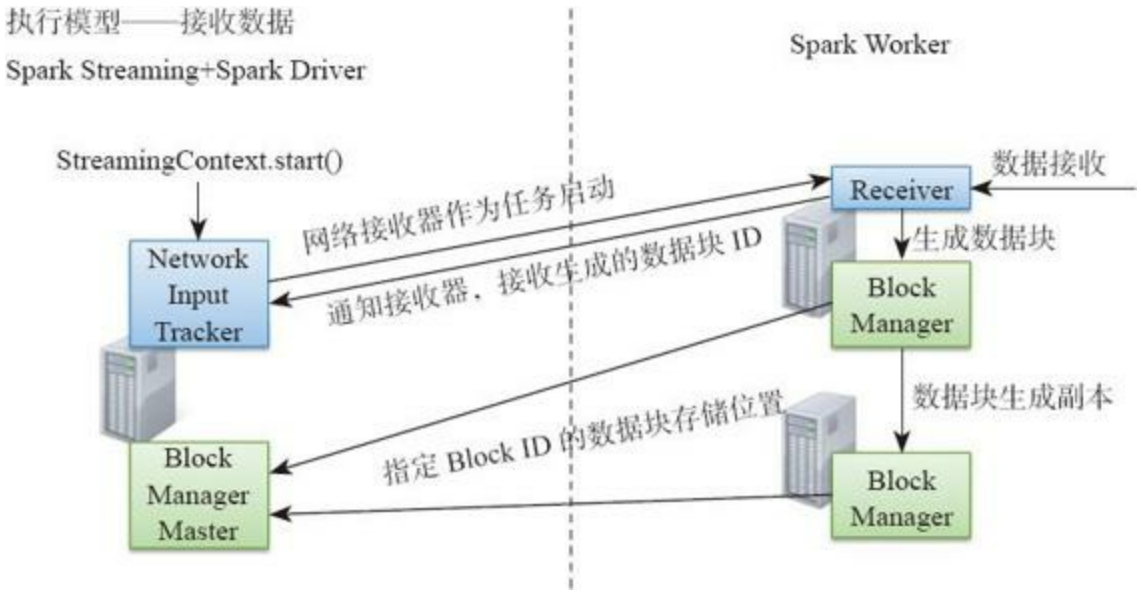

图8-12　Spark Streaming执行模型

初始化的过程主要可以概括为以下两点。

1）调度器的初始化。

2）将输入流的接收器转化为RDD在集群打散，然后启动接收器集合中的每个接收器。

下面通过具体的代码更深入地理解这个过程。

（1）NetworkWordCount示例

本例以NetworkWordCount作为研究Spark Streaming的入口程序。

```
object NetworkWordCount {
  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage: NetworkWordCount <hostname> <port>")
      System.exit(1)
    }
    StreamingExamples.setStreamingLogLevels()
val sparkConf = new SparkConf().setAppName("NetworkWordCount")
/*创建StreamingContext对象，形成整个程序的上下文*/
val ssc = new StreamingContext(sparkConf, Seconds(1))
/*通过socketTextStream接收源源不断地socket文本流*/
    val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

（2）进入scoketTextStream

```
def socketTextStream(
     hostname: String,
     port: Int,
     storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): 
ReceiverInputDStream[String] = {
/*内部实际调用的socketStream方法 */
    socketStream[String](hostname, port, SocketReceiver.bytesToLines, storageLevel)
  }
   /*进入socketStream方法 */
  def socketStream[T: ClassTag](
      hostname: String,
      port: Int,
      converter: (InputStream) => Iterator[T],
      storageLevel: StorageLevel
  ): ReceiverInputDStream[T] = {
/*此处初始化SocketInputDStream对象 */
    new SocketInputDStream[T](this, hostname, port, converter, storageLevel)
  }
```

（3）初始化SocketInputDStream

在之前的Spark Streaming介绍中，读者已经了解到整个Spark Streaming的调度灵魂就是DStream的DAG，可以将这个DStream DAG类比Spark中的RDD DAG，而DStream类比RDD，DStream可以理解为包含各个时间段的一个RDD集合。SocketInputDStream就是一个DStream。

```
private[streaming]
class SocketInputDStream[T: ClassTag](
   @transient ssc_ : StreamingContext,
   host: String,
   port: Int,
   bytesToObjects: InputStream => Iterator[T],
   storageLevel: StorageLevel
  ) extends ReceiverInputDStream[T](ssc_) {
```

```
  def getReceiver( ) : Receiver[T] = {
   new SocketReceiver( host , port , bytesToObjects , storageLevel )
  }
}
```

（4）触发StreamingContext中的Start（ ）方法

上面的步骤基本完成了Spark Streaming的初始化工作。类似于Spark机制，Spark Streaming也是延迟（Lazy）触发的，只有调用了start（ ）方法，才真正地执行了。

```
private[streaming] val scheduler = new JobScheduler( this )
  /*StreamingContext中维持着一个调度器*/
  def start( ) : Unit = synchronized {
   ……
/*启动调度器*/
    scheduler.start( )
  ……
  }
```

（5）JobScheduler.start（ ）启动调度器

在start方法中初始化了很多重要的组件。

```
def start( ) : Unit = synchronized {
   ……
/*初始化事件处理Actor，当有消息传递给Actor时，调用processEvent进行事件处理*/
     eventActor = ssc.env.actorSystem.actorOf( Props( new Actor {
       def receive = {
         case event : JobSchedulerEvent => processEvent( event )
       }
 } ) , "JobScheduler" )
 /*启动监听总线*/
 listenerBus.start( )
    receiverTracker = new ReceiverTracker( ssc )
 /*启动接收器的监听器receiverTracker*/
    receiverTracker.start( )
 /*启动job生成器*/
    jobGenerator.start( )
   ……
  }
```

（6）ReceiverTracker类

/*进入ReceiverTracker查看*/

```
private[streaming]
class ReceiverTracker( ssc : StreamingContext ) extends Logging {
 val receiverInputStreams = ssc.graph.getReceiverInputStreams( )
  def start( ) = synchronized {
……
```

```
val receiverExecutor = new ReceiverLauncher( )
  ……
if ( !receiverInputStreams.isEmpty ) {
/*初始化ReceiverTrackerActor */
      actor = ssc.env.actorSystem.actorOf( Props( new ReceiverTrackerActor ) ,
        "ReceiverTracker" )
   /*启动ReceiverLauncher( ) 实例，( 7 ) 中进行介绍*/
      receiverExecutor.start( )
  ……
    }
  }
/*读者可以先参考ReceiverTrackerActor的代码查看实现注册Receiver和注册Block元数据信息的功能。 */
  private class ReceiverTrackerActor extends Actor {
def receive = {
/*接收注册receiver的消息，每个receiver就是一个输入流接收器，Receiver分布在Worker节点，一个Receiver接收一个输入流，一个
Spark Streaming集群可以有多个输入流 */
      case RegisterReceiver( streamId , typ , host , receiverActor ) =>
        registerReceiver( streamId , typ , host , receiverActor , sender )
        sender ! true
      case AddBlock( receivedBlockInfo ) =>
        addBlocks( receivedBlockInfo )
      ……
    }
  }
```

（7）receivelauncher类，在集群上分布式启动接收器

```
class ReceiverLauncher {
   ……
    @transient val thread  = new Thread( ) {
      override def run( ) {
      ……
/*启动ReceiverTrackerActor已经注册的Receiver*/
        startReceivers( )
      ……
   }
```

下面进入startReceivers方法，方法中将Receiver集合转变为RDD，从而在集群上打散，分布式分布。如图8-13所示，一个集群可以分布式地在不同的Worker节点接收输入数据流。

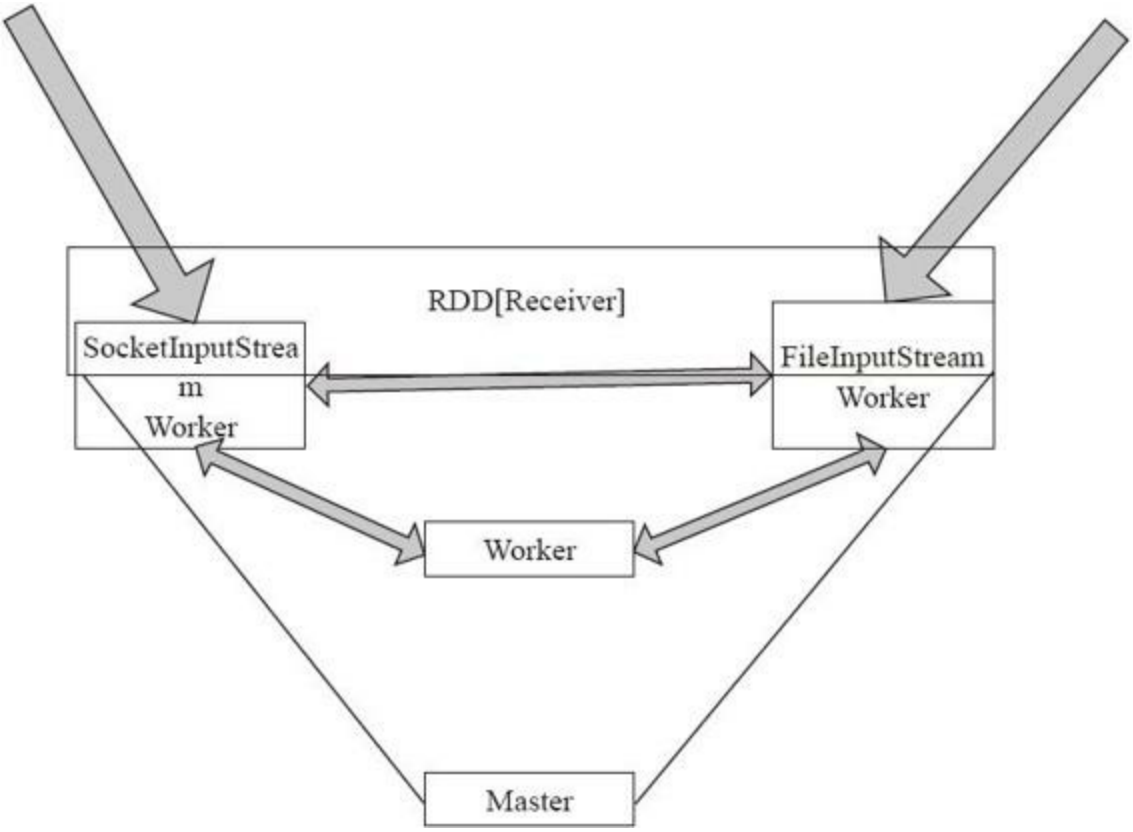

图8-13　Spark Streaming接收器

```
private def startReceivers( ) {
/*获取之前配置的接收器 */
     val receivers = receiverInputStreams.map( nis => {
        val rcvr = nis.getReceiver( )
        rcvr.setReceiverId( nis.id )
        rcvr
      } )
      ……
      /* 创建并行的在不同Worker节点分布的receiver集合 */
      val tempRDD =
        if ( hasLocationPreferences ) {
         val receiversWithPreferences = receivers.map( r => ( r , Seq( r.preferredLocation.get ) ) )
         ssc.sc.makeRDD[Receiver[_]]( receiversWithPreferences )
        } else {
/*在这里创造RDD相当于进入SparkContext.makeRDD，此经典之处在于将receivers集合作为一个RDD [Receiver]进行分区。即使只有一个输入流，按照分布式分区方式，也是将输入分布在Worker端，而不在Master*/
        ssc.sc.makeRDD( receivers , receivers.size )
/*调用Sparkcontext中的makeRDD方法，本质是调用将数据分布式化的方法parallelize*/
/* def makeRDD[T： ClassTag]( seq： Seq[T] , numSlices： Int = defaultParallelism )： //RDD[T] = {
parallelize( seq , numSlices ) */
/*在RDD[Receiver[_]]每个分区的每个Receiver 上都同时启动，这样其实Spark Streaming可以构建大量的分布式输入流 */
      val startReceiver = ( iterator： Iterator[Receiver[_]] ) => {
        if ( !iterator.hasNext ) {
          throw new SparkException(
            "Could not start receiver as object not found." )
        }
        val receiver = iterator.next( )
/*此处的supervisorImpl是一个监督者的角色，在下面的内容中将会剖析这个对象的作用 */
       val executor = new ReceiverSupervisorImpl( receiver , SparkEnv.get )
       executor.start( )
       executor.awaitTermination( )
     }
 /
/*将receivers的集合打散，然后启动它们 */
 ……
      ssc.sparkContext.runJob( tempRDD , startReceiver )
 ……
    }
```

2.数据接收与转化

在“1.初始化与集群上分布接收器”中介绍了，receiver集合转换为RDD在集群上分布式地接收数据流。那么每个receiver是怎样接收并处理数据流的呢？Spark Streaming数据接收与转化的示意图如图8-14所示。

图8-14的主要流程如下。

1）数据缓冲：在Receiver的receive函数中接收流数据，将接收到的数据源源不断地放入BlockGenerator.currentBuffer。

2）缓冲数据转化为数据块：在BlockGenerator中有一个定时器（recurring timer），将当前缓冲区中的数据以用户定义的时间间隔封装为一个数据块Block，放入BlockGenerator的blocksForPush队列中。

3）数据块转化为Spark数据块：在BlockGenerator中有一个BlockPushingThread线程，不断地将blocksForPush队列中的块传递给Blockmanager，让BlockManager将数据存储为块，读者可以在本书的Spark IO章节了解Spark的底层存储机制。BlockManager负责Spark中的块管理。

4）元数据存储：在pushArrayBuffer方法中还会将已经由BlockManager存储的元数据信息（如Block的ID号）传递给ReceiverTracker，ReceiverTracker将存储的blockId放到对应StreamId的队列中。

上面过程中涉及最多的类就是BlockGenerator，在数据转化的过程中，其扮演着不可或缺的角色。

```
private[streaming] class BlockGenerator(
    listener: BlockGeneratorListener,
    receiverId: Int,
    conf: SparkConf
  ) extends Logging
```

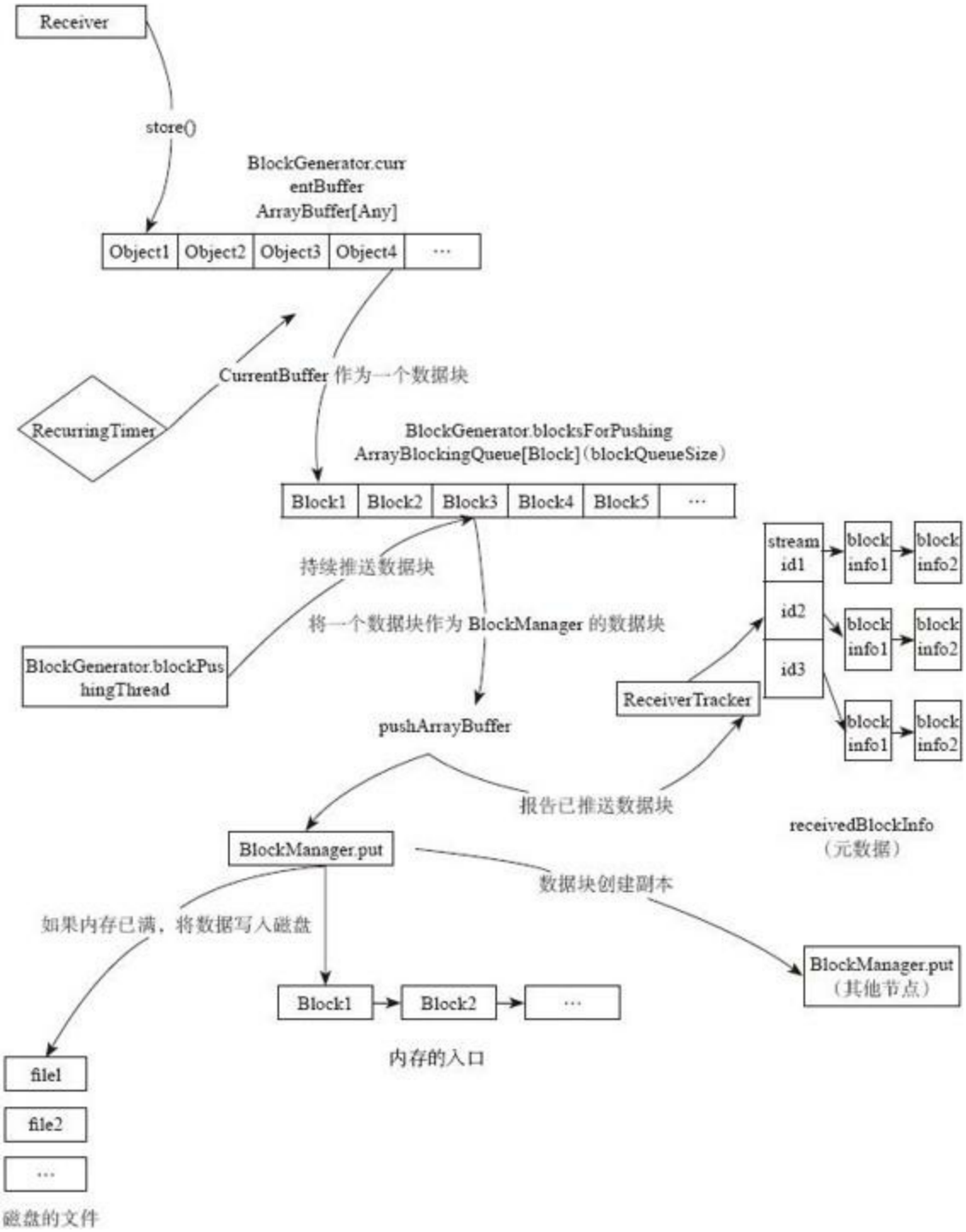

图8-14　Spark Streaming数据接收与转化

感兴趣的读者可以参照图8-14中的类和方法更加具体地了解机制。由于篇幅所限，这个数据生成过程的代码不再具体剖析。

3.生成RDD与提交Spark Job

Spark Streaming根据时间段，将数据切分为RDD，然后触发RDD的Action提交Job，Job被提交到JobManager中的Job Queue中，由JobScheduler调度，Job Scheduler将Job提交到

Spark的Job调度器，然后将Job转换为大量的任务分发给Spark集群执行。

如图8-15所示，Jobgenerator中通过下面的方法生成Job调度和执行。

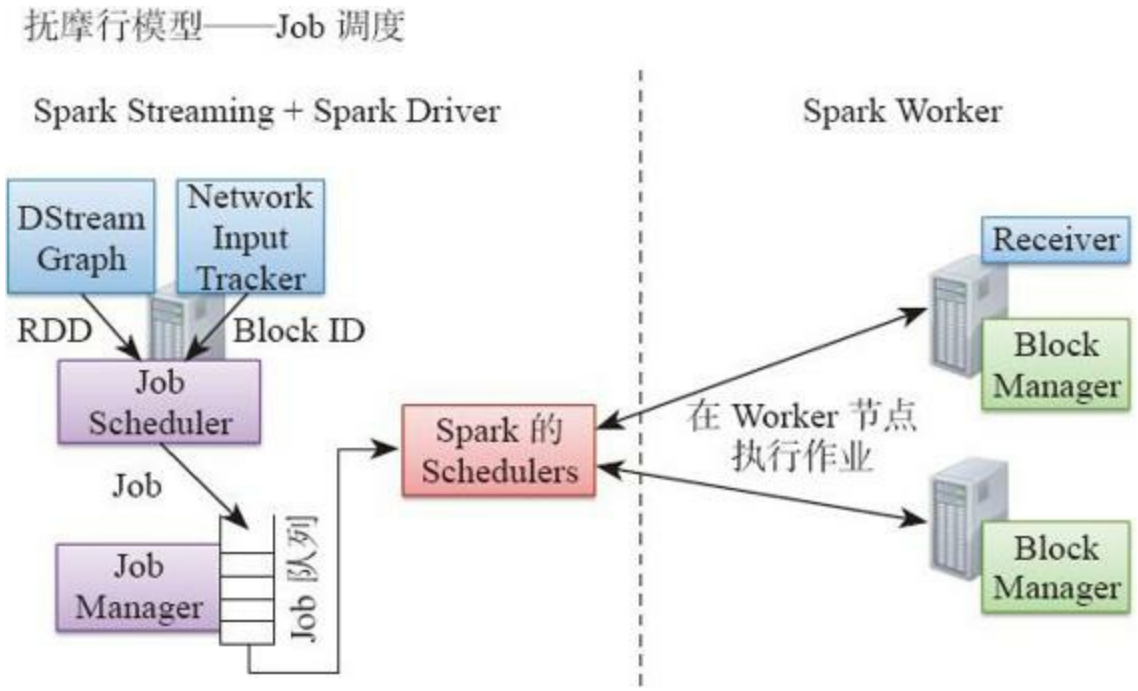

图8-15　Spark Streaming调度模型

从下面的代码中可以看出，Jobs是从outputStream中生成的，然后触发反向回溯执行整个DStream DAG，类似于RDD的机制。

```
private def generateJobs(time: Time) {
    SparkEnv.set(ssc.env)
    Try(graph.generateJobs(time)) match {
      case Success(jobs) =>
        /*获取输入数据块的信息*/
        val receivedBlockInfo = graph.getReceiverInputStreams.map { stream =>
         ……
        }.toMap
        jobScheduler.submitJobSet(JobSet(time, jobs, receivedBlockInfo))
      case Failure(e) =>
       jobScheduler.reportError("Error generating jobs for time " + time, e)
    }
    eventActor ! DoCheckpoint(time)
  }
/*下面进入JobScheduler的submitJobSet方法一探究竟，JobScheduler是整个Spark Streaming调度的核心组件*/
 def submitJobSet(jobSet: JobSet) {
    ……
      jobSets.put(jobSet.time, jobSet)
      jobSet.jobs.foreach(job => jobExecutor.execute(new JobHandler(job)))
 …
  }
/*进入Graph生成job的方法，graph的本质是DStreamGraph类生成的对象 */
    final private[streaming] class DStreamGraph extends Serializable with Logging {
    def generateJobs(time: Time): Seq[Job] = {
     ……
    private val inputStreams = new ArrayBuffer[InputDStream[_]]()
     private val outputStreams = new ArrayBuffer[DStream[_]]()
      ……
```

```
      val jobs = this.synchronized {
        outputStreams.flatMap(outputStream => outputStream.generateJob(time))
    …
  }
/*outputStreams中的对象是DStream，下面进入DStream的generateJob一探究竟*/
private[streaming] def generateJob(time: Time): Option[Job] = {
   getOrCompute(time) match {
    case Some(rdd) => {
      val jobFunc = () => {
       val emptyFunc = { (iterator: Iterator[T]) => {} }
/*此处相当于针对每个时间段生成的一个RDD，会调用SparkContext的方法runJob提交Spark的一个Job*/
       context.sparkContext.runJob(rdd, emptyFunc)
      }
      Some(new Job(time, jobFunc))
    }
    case None => None
  }
 }
/*在DStream算是父类，一些具体的DStream，如SocketInputStream等的类的父类。可以通过SocketInputDStream查看如何通过上面的
getOrCompute生成RDD*/
  private[streaming] def getOrCompute(time: Time): Option[RDD[T]] = {
   generatedRDDs.get(time) match {
     …
     case None => {
       if (isTimeValid(time)) {
/* Dstream是个父类，Dstream的子类可以完成不同算子运算，这样的继承关系意味着Action类型的Dstream会触发compute函数运算，并反向
回溯到顶层的Dstream类运行compute函数计算。这样每隔一段时间，生成的RDD反向计算一次。计算模式类似于RDD的DAG */
          compute(time) match {
            ……
             generatedRDDs.put(time, newRDD)
           ……
  }
在SocketInputDStream的compute方法中生成对应时间片的RDD。
  override def compute(validTime: Time): Option[RDD[T]] = {
    if (validTime >= graph.startTime) {
      val blockInfo = ssc.scheduler.receiverTracker.getReceivedBlockInfo(id)
      receivedBlockInfo(validTime) = blockInfo
      val blockIds = blockInfo.map(_.blockId.asInstanceOf[BlockId])
      Some(new BlockRDD[T](ssc.sc, blockIds))
    } else {
      Some(new BlockRDD[T](ssc.sc, Array[BlockId]()))
    }
  }
```

Dstream是个父类，其子类可以完成不同算子运算，这样的继承关系意味着Action类型的Dstream会触发compute函数运算，并反向回溯到顶层的Dstream类运行compute函数计算，这样每隔一段时间，生成的新RDD反向计算。计算模式类似于RDD的DAG。

8.2.4 Spark Streaming调优

Spark Streaming调优方式和Spark调优方式很相近，可以互相借鉴。

1.运行时间调优

并行度优化。确保任务使用整个集群的资源，防止数据倾斜。

减少数据序列化、反序列化以及减少Task提交和分发开销。用户可以通过配置使用Kyro使序列化更优化。当批处理窗口时间间隔非常小（例如小于500ms）时，提交和分发任务的延迟变得很大，此时应适当调大批处理窗口。通常情况下，使用Standalone模式和Coarse-grained Mesos模式会比使用Fine-Grained Mesos模式延迟更小。

设置合理的批处理窗口。Job是流水线执行，要防止流水线阻塞，就需要设置合理的批处理窗口。

2.空间占用调优

定时清理不用的数据。用户通过配置spark.cleaner.ttl时长来及时清理超时的无用数据及元数据。

GC（JVM垃圾回收）调优。具体方法可参考Spark性能调优的章节。

控制批处理量。Spark Streaming一个批处理窗口内接收到的所有数据均在Spark可用内存区域中存放。确保当前节点Spark的可用内存能够容纳这个batch窗口内的所有数据。

8.2.5 Spark Streaming实例

在互联网应用中，流数据处理是一种常用的应用模式，需要在不同粒度上对不同数据进行统计，保证实时性的同时，又需要涉及聚合（aggregation）、去重（distinct）、连接（join）等较为复杂的统计需求[1]。如果使用MapReduce框架，虽然可以容易地实现较为复杂的统计需求，但实时性却无法得到保证；反之，若是采用Storm这样的流式框架，实时性虽可以得到保证，但需求的实现复杂度也大大提高了。Spark Streaming在实时性与复杂统计需求之间的权衡中找到了一个平衡点，能够满足大多数用户的流计算需求。

Spark Streming是Spark的一个组成部分，提供高扩展性、容错的流处理功能。下面的例子基于Standalone的Spark程序，接收和处理Twitter的真实采样推特流。在这个例子中，用户可以选择使用Scala或者Java书写程序。

1.设置Setup

首先介绍基本的配置Spark Streaming程序的方法，然后介绍如何进行Twitter流身份验证令牌的配置。

（1）系统设置

读者需要在官网：https://github.com/amplab/training/tree/ampcamp4/streaming，预先下载示例程序的模板。在用户的集群，假设下面介绍的模板和程序已经在目录/root/streaming/下配置，用户将会在目录下发现下面的数据项。

1）twitter.txt：包含Twitter证书细节的文件。

2）目录介绍

①Scala用户：

·scala/sbt：包含SBT工具的目录。

·scala/build.sbt：SBT项目文件。

·scala/Tutorial.scala：主程序，需要用户编辑、编译和运行。

·scala/TutorialHelper.scala：包含一些帮助函数的Scala文件。

②Java用户：

·java/sbt：包含SBT工具的目录。

·java/build.sbt：SBT项目文件。

·java/Tutorial.java：Java主程序，需要用户编辑、编译和运行。

·java/TutorialHeler.java：包含一些帮助函数的Java文件。

·java/ScalaHelper.java：包含一些帮助函数的Scala文件。

用户需要编辑、编译和运行的主文件是Tutorial.scala或者Tutorial.java。注意：需要在模板文件中更改sparkUrl。

```
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.streaming._
import org.apache.spark.streaming.twitter._
import org.apache.spark.streaming.StreamingContext._
import TutorialHelper._
object Tutorial {
  def main(args: Array[String]) {
    /* Spark的目录*/
    val sparkHome = "/root/spark"
  /* Spark集群的Master 节点链接*/
  val sparkUrl = "local[4]"
    /* 应用所需的Jar包地址*/
    val jarFile = "target/scala-2.10/tutorial_2.10-0.1-SNAPSHOT.jar"
  /* 为了检查点而配置的HDFS目录 */
    val checkpointDir = TutorialHelper.getHdfsUrl() + "/checkpoint/"
    /* 使用 twitter.txt 配置twitter证书 */
    TutorialHelper.configureTwitterCredentials()
    /* 在此处书写用户代码 */
  }
}
```

为了方便用户，例子中增加了一些帮助函数来配置需要的参数。

getSparkUrl（）是一个帮助函数用于在/root/spark-ec2/cluster-url下获取Spark集群的URL。

configureTwitterCredential（）是一个帮助函数。使用文件file/root/streaming/twitter.txt配置Twitter证书。这个配置将会在下面介绍。

（2）Twitter证书设置

由于所有的例子都是基于Twitter采样tweet流。首先需要有一个Twitter账号用来配置OAuth证书。为了达到这个目的，用户需要使用Twitter账号来设置一个消费者的key+secret对和访问token+secret对。请读者按照下面的步骤通过Twitter账号设置这些临时访问关键字。

1）打开链接https://dev.twitter.com/apps。用户页面罗列了基于Twitter的应用、应用的消费者关键字和访问令牌。如果没有创建任何应用，则这个页面是空的。在这个教程中，用户可以创建一个临时的应用。单击蓝色的“Create a new application”按钮，出现一个新应用的页面，如图8-16所示。需要填入一些信息：因为应用的名字（Name）必须是全局唯一的，所以可以使用Twitter的用户名作为前缀。描述（Description）字段可以随意设置。网址（Website）字段可以任意，但是需要确保是一个有http://前缀的全格式的URL。单击Developer Rules of the Road下面的“Yes，I agree”，最后单击“Create your Twitter application”按钮。

2）创建应用程序后，将会看到一个和图8-17相似的确认页面。用户可以获取consumer key和consumer secret。为了生成访问token和secret，需要单击页面底部的蓝色“Create my access token”按钮。注意：页面顶部会出现小的绿色确认信息，说明令牌已经生成。

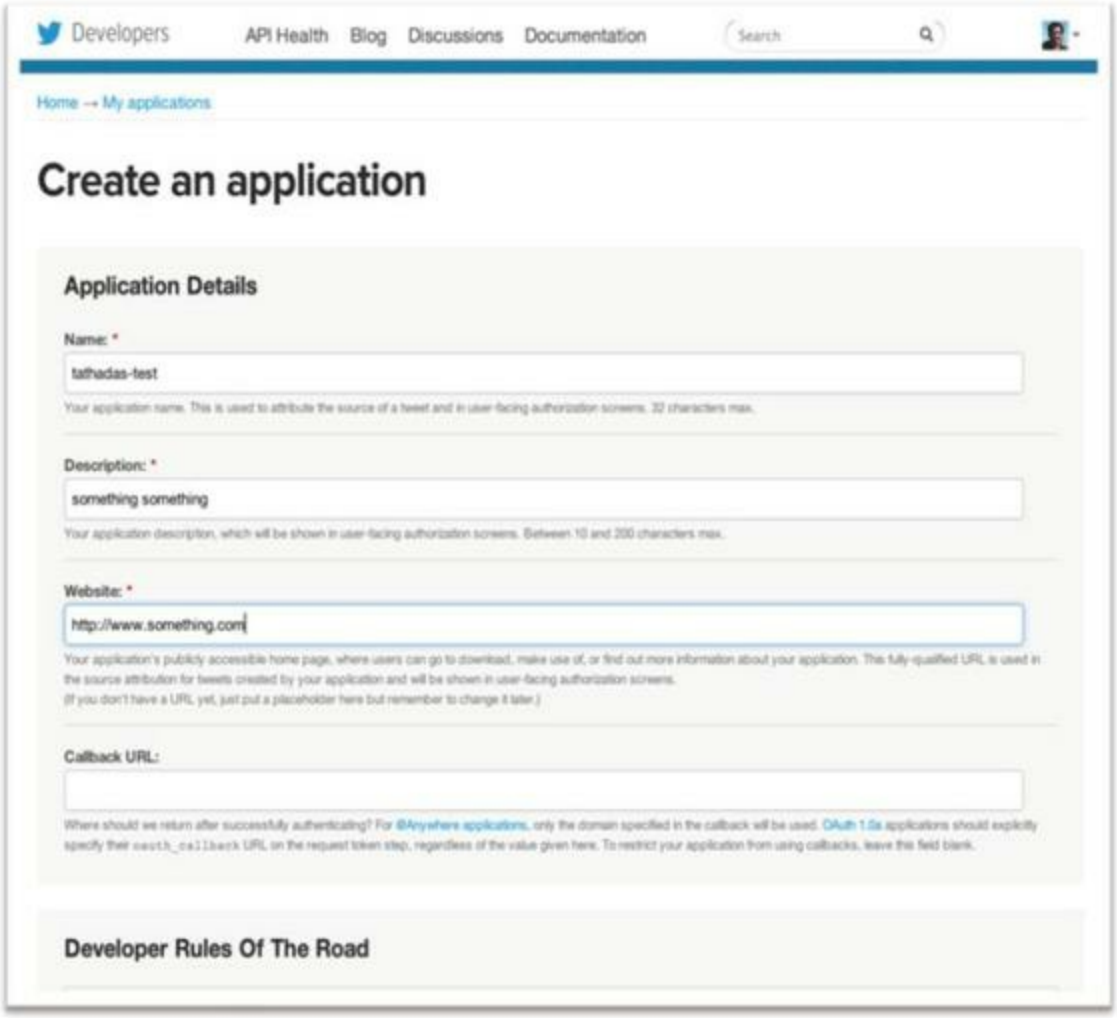

图8-16　填写应用信息

3）为了获取证书需要的所有key和secret，在页面的菜单顶部单击OAuth Tool，将会看到如图8-18所示的页面。

4）更新twitter.txt配置文件。

```
cd /root/streaming/
vim twitter.txt
```

用户会看到下面的模板。

```
consumerKey =
consumerSecret =
accessToken =
accessTokenSecret =
```

请用户复制之网页上相应参数值到这个配置文件对应位置，复制后，会出现类似下面的

配置信息。

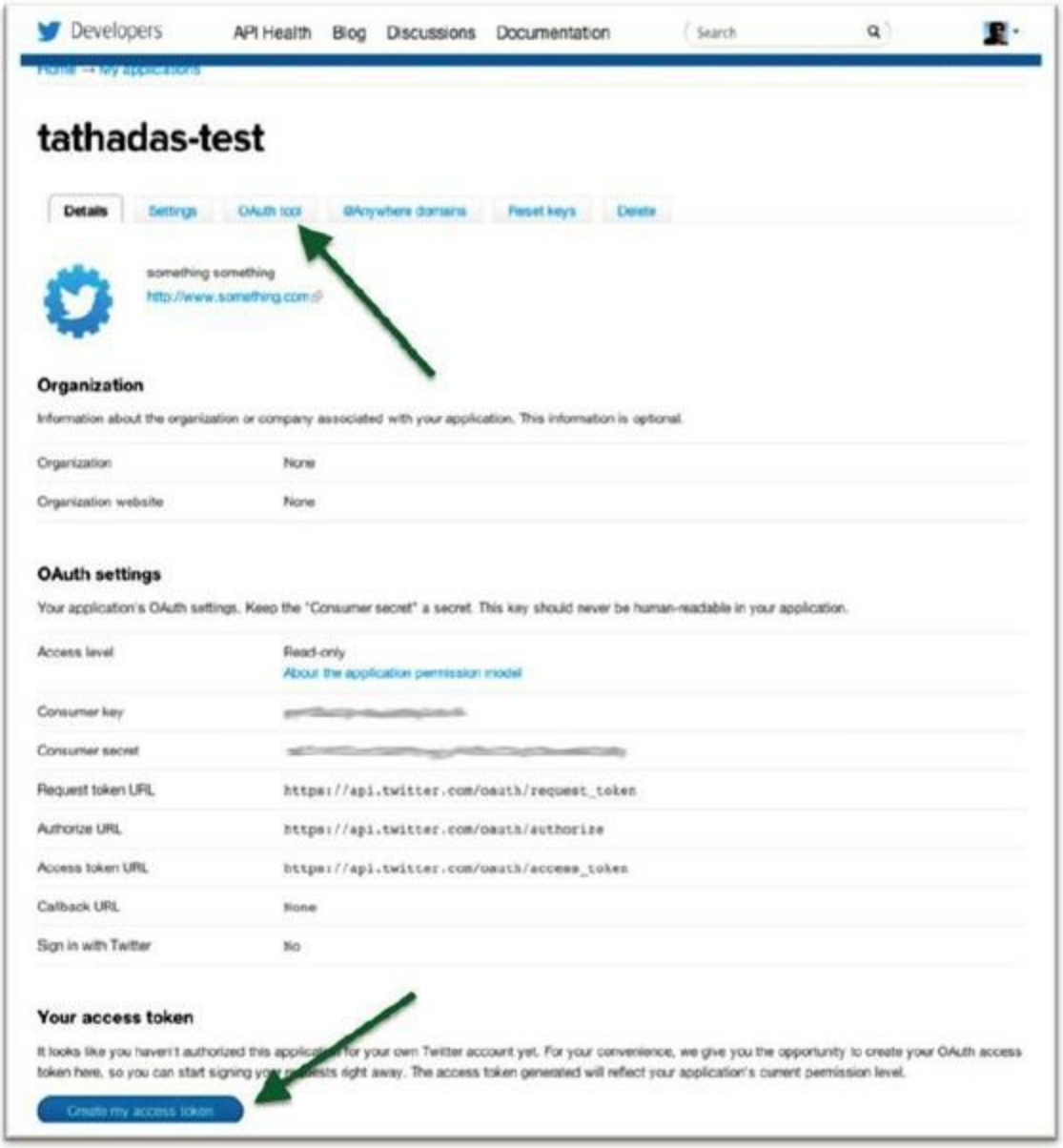

图8-17　确认页面

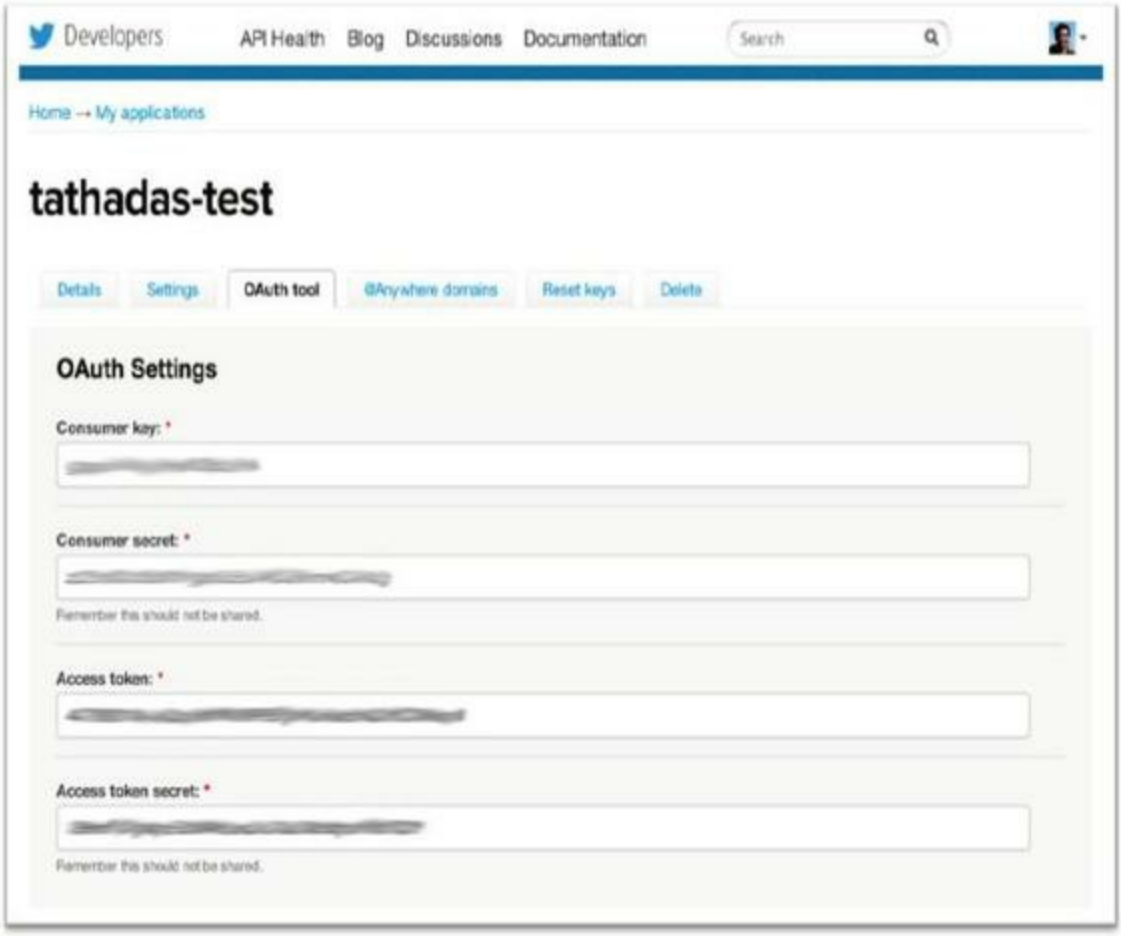

图8-18　OAuth Tool页面

```
consumerKey = z25xt02zcaadf12 ...
consumerSecret = gqc9uAkjla13 ...
accessToken = 8mitfTqDrgAzasd ...
accessTokenSecret = 479920148 ...
```

确认无误后，保存文件，就可以开发Spark Streaming程序了。

5）如果做完练习不再使用这个Twitter应用，可以到官网的页面单击Delete按钮，将应用删除，如图8-19所示。

图8-19　删除应用页面

2.书写第一个Spark Streaming程序

下面介绍一个简单的Spark Streaming应用程序，它会每秒将接收到的推文打印出来。

1）打开并编辑Tutorial.scala文件。

```
cd /root/streaming/scala/
vim Tutorial.scala
```

2）创建StreamingContext对象。这个对象是Spark Streaming程序的入口。

```
val ssc = new StreamingContext(sparkUrl, "Tutorial", Seconds(1), sparkHome, Seq(jarFile))
```

在本例中，创建了一个StreamingContext对象，并传入Spark集群的URL（sparkUrl）、流数据的批处理（batch）持续时间（Seconds（1））、Spark的根目录（sparkHome）程序运行需要的jar包jarFile）和应用程序名（Tutorial）。

```
val tweets = TwitterUtils.createStream(ssc, None)
```

3）使用StreamingContext对象创建tweet数据流。

tweets对象是一个DStream对象，是一个源源不断的RDD流，RDD中的数据项就是twitter4j.Status对象。用户可以通过下面的语句打印出现在的数据流一探究竟。

```
val statuses = tweets.map(status => status.getText())
    statuses.print()
```

类似本书前几章提到的RDD变换（transformation），tweets上的map算子作用在tweets对象上又创建了一个新的Dstream，叫作status。print函数打印DStream中每个RDD的前10条数据。

如果需要容错，可以调用Checkpoint方法，输入参数为HDFS的文件路径，将数据冗余存储在HDFS中。

```
ssc.checkpoint(checkpointDir)
```

4）通过下面两个方法触发整个程序的运行。

```
ssc.start()
ssc.awaitTermination()
```

注意：上面的两个参数应该在用户做完所有操作之后再触发。

5）编辑好后保存Tutorial.scala文件，在根目录运行下面的命令。

```
sbt/sbt package run
```

这个命令将会自动编译Tutorial类，并在/root/streaming/[language]/target/scala-2.10/.目录下创建jar包。最后运行这个程序，如果运行成功，将会在控制台看到类似下面的日志信息。

```
-------------------------------------------
Time: 1359886325000 ms
-------------------------------------------
RT @__PiscesBabyyy: You Dont Wanna Hurt Me But Your Constantly Doing It
@Shu_Inukai ??????????????????????????????????????
@Condormoda Us vaig descobrir a la @080_bcn_fashion. Molt bona desfilada. Salutacions des de #Manresa
RT @dragon_itou: ?RT???????3000????????????????????????????????????10???????
??????????????????2?3????9???? #???? http://t.co/PwyA5dsI ? h ...
Sini aku antar ke RSJ ya "@NiieSiiRenii: Memang (?? ?`? )"@RiskiMaris: Stresss"@NiieSiiRenii: Sukasuka aku donk:p"@RiskiMaris: Makanya jgn"
@brennn_star lol I would love to come back, you seem pretty cool! I just dont know if I could ever do graveyard again :( It KILLs me
????????????????????????????????????????????????????????????????????????????????????????????????ww
??????????
When the first boats left the rock with the artificers employed on.
@tgs_nth ??????????????????????????????
……
-------------------------------------------
Time: 1359886326000 ms
-------------------------------------------
??????????
??????????
@amatuki007 ????????????????????????????????
???????????????????
RT @BrunoMars: Wooh!
Lo malo es qe no tiene toallitas
Sayang beb RT @enjaaangg Piye ya perasaanmu nyg aku :o
Baz? ?eyler yar??ma ya da reklam konusu olmamal? d???ncesini yenemiyorum.
```

```
?????????????MTV???????the HIATUS??
@anisyifaa haha. Cukupla merepek sikit2 ：3
@RemyBot ?????????
……
```

---

[1] 示例参考http://ampcamp.berkeley.edu/big-data-mini-course/realtime-processing-with-spark-streaming.html。

8.3 GraphX

Graphx是Spark中的一个重要子项目，它利用Spark为计算引擎，实现了大规模图计算的功能，并提供了类似Pregel的编程接口。GraphX的出现，使Spark生态系统更加完善和丰富，同时其与Spark生态系统其他组件很好的融合，以及强大的图数据处理能力，使其在工业界得到了广泛的应用。本章主要介绍GraphX的架构、原理和使用方式。

8.3.1 GraphX简介

GraphX是常用图算法在Spark上的并行化实现，提供了丰富的API接口。图算法是很多复杂机器学习算法的基础，在单机环境下有很多应用案例。在大数据环境下，图的规模大到一定程度后，单机很难解决大规模的图计算，需要将算法并行化，在分布式集群上进行大规模图处理。目前，比较成熟的方案有GraphX和GraphLab等大规模图计算框架。图8-20为GraphX发展史简图。

图8-20 GraphX发展史

GraphX的特点是离线计算、批量处理、基于同步的整体同步并行计算模型（即BSP计算模型），这样的优势在于可以提升数据处理的吞吐量和规模，但是会造成速度上稍逊一筹。目前大规模图处理框架还有基于MPI模型的异步图计算模型GraphLab和同样基于BSP模型的Giraph等。

现在和GraphX可以组合使用的分布式图数据库是Neo4J和Titan。Neo4j一个高性能的、非关系的、具有完全事务特性的、鲁棒的图数据库。Titan是一个分布式的图形数据库，特别为存储和处理大规模图形而优化。二者均可作为GraphX的持久化层，存储大规模图数据。

8.3.2 GraphX的使用

类似Spark在RDD上提供了一组基本操作符（如map、filter、reduce），GraphX同样也有针对Graph的基本操作符，用户可以在这些操作符传入自定义函数和通过修改图的节点属性或结构生成新的图。

GraphX提供了丰富的针对图数据的操作符。Graph类中定义了核心的、优化过的操作符。一些更加方便的由底层核心操作符组合而成的上层操作符在GraphOps中定义。正是通过Scala语言的implicit关键字，GraphOps中定义的操作符可以作为Graph中的成员。这样做的目的是未来GrpahX会支持不同类型的图，而每种类型图的呈现必须实现核心的操作符和复用大部分GrpahOps中实现的操作符。

下面将操作符分为几个类别进行介绍。

1.属性操作符

属性操作符如表8-1所示。

表8-1 属性操作符

属性操作符	说明
mapVertices[VD2](map: (VertexId, VD) => VD2): Graph [VD2, ED]	使用 map 函数对图中所有顶点进行转换操作
mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]	使用 map 函数对图中所有边属性进行转换操作
mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]	可以对边中的顶点属性或者边属性进行 map 函数转换操作

2.结构操作符

结构操作符如表8-2所示。

表8-2 结构操作符

结构操作符	说明
reverse: Graph[VD, ED]	反转图中所有边的方向
subgraph(epred:EdgeTriplet[VD,ED]=>Boolean,vpred:(VertexId,VD)=>Boolean):Graph[VD,ED]	获取图中顶点和边满足函数条件的子图
mask[VD2,ED2](other:Graph[VD2,ED2]): Graph[VD,ED]	将本图中所有包含在 other 图中的顶点和边保留，顶点和边的属性不变
groupEdges(merge:(ED,ED)=>ED):Graph[VD,ED]	将两个顶点间的多条边合并为一条边

3.图信息属性

图信息属性如表8-3所示。

表8-3　图信息属性

图信息属性	说明
val numEdges: Long	图中边的数量
val numVertices: Long	图中顶点的数量
val inDegrees: VertexRDD[Int] val outDegrees: VertexRDD[Int] val degrees: VertexRDD[Int]	inDegrees：图中入度数量 outDegrees：图中出度数量 degrees：图中度的总数量

4.邻接聚集操作符与Join操作符

邻接聚集操作符与Join操作符如表8-4所示。

表8-4　邻接聚集操作符与Join操作符

邻接聚集操作符与 Join 操作符	说明
mapReduceTriplets[A](map:EdgeTriplet[VD,ED]=>Iterator[(VertexId,A)],reduce:(A, A) => A): VertexRDD[A]	函数的作用是对每个顶点进行聚集操作 函数中的 map 函数将三元组数据映射为目的节点为 Key 的 Key-Value 对，reduce 函数将同一个顶点的数据汇聚形成入度数，作为新图的顶点属性 val rawGraph: Graph[(),()] = Graph.textFile("twittergraph") val inDeg: RDD[(VertexId, Int)] = mapReduceTriplets[Int](et => Iterator((et.dst.id, 1)), _ + _)
joinVertices[U](table: RDD[(VertexId,U)])(map: (VertexId, VD, U) => VD):Graph[VD, ED]	将图中顶点和另一个 RDD 连接（其 VertexId 为 key），并将连接后的数据项进行 map 函数运算
outerJoinVertices[U,VD2](table: RDD [(VertexId,U)])(map: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]	类似上面的 joinVertices，但是输入的 table RDD 应该保证有对应 graph 的所有 VertexId，如果没有，则 map 函数的输入为 None

5.缓存操作符

缓存操作符如表8-5所示。

表8-5　缓存操作符

缓存操作符	说明
def cache(): Graph[VD, ED]	缓存图中的顶点和边
Def persist(newLevel: StorageLevel=StorageLevel.MEMORY_ONLY): Graph [VD, ED]	用户可以指定存储级别来缓存图的顶点和边
def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]	不再缓存顶点，但保留边数据

6.Pregel API

Pregel基于BSP模型，提供了3个重要的需要用户书写的函数。通过官方PageRank算法进一步理解这3个函数的使用。

```
def run[VD: ClassTag, ED: ClassTag](
    graph: Graph[VD, ED], numIter: Int, resetProb: Double = 0.15): Graph[Double, Double] =
  {
val pagerankGraph: Graph[Double, Double] = graph
  /* 将节点的度与图中节点关联*/
  .outerJoinVertices(graph.outDegrees) {
    (vid, vdata, deg) => deg.getOrElse(0)
  }
  /* 根据度设置边的权重*/
  .mapTriplets(e => 1.0 / e.srcAttr)
  /* 设置节点属性为PageRank算法初始值*/
  .mapVertices( (id, attr) => 1.0)
def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double =
  resetProb + (1.0 - resetProb) * msgSum
def sendMessage(id: VertexId, edge: EdgeTriplet[Double, Double]): Iterator [(VertexId, Double)] =
 Iterator( (edge.dstId, edge.srcAttr * edge.attr) )
def messageCombiner(a: Double, b: Double): Double = a + b
val initialMessage = 0.0
/* 以固定迭代次数执行Pregel*/
Pregel(pagerankGraph, initialMessage, numIter)(
  vertexProgram, sendMessage, messageCombiner)
}
```

这3个函数按顺序执行完一次是一个迭代轮次，numIter决定需要执行多少伦次完成迭代。

但初始由于vprog函数是没有输入的，所以还需要用户输入initialMessage作为第一轮的初始化数据。

下面通过表8-6介绍Pregel API。

表8-6　Pregel API

Pregel API	说明
vprog: (VertexId, VD, A) => VD	对应例子中： def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double = resetProb + (1.0 - resetProb) * msgSum 如果上一轮次的 mergeMsg 已经计算完成（如果是第一次，则是 initialMessage 的结果），则通过函数输入可以获取上一轮次对这个顶点的聚合结果 msgSum，然后利用这个值对顶点赋予新的属性（resetProb + (1.0 - resetProb) * msgSum）reseProb 是随机游走概率值
sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)]	对应例子中： def sendMessage(id: VertexId, edge: EdgeTriplet[Double, Double]) ： Iterator[(VertexId, Double)] = Iterator((edge.dstId, edge.srcAttr * edge.attr)) 函数中需要用户将数据封装为（目的顶点 id，value）的数据对，这样 GraphX 会将 pair 数据分发到目的顶点 ID 所在机器

（续）

Pregel API	说明
mergeMsg: (A, A) => A	对应例子中：def messageCombiner(a: Double, b: Double): Double = a + b 如果上一步的 pair 已经到达顶点所在的机器，则将顶点收到的其他顶点传给自己的所有数据（形式为：（顶点 id，value）），通过用户定义的 mergeMsg 函数聚合运算

8.3.3 GraphX架构

1.整体架构

GraphX的整体架构可以分为以下3部分，如图8-21所示。

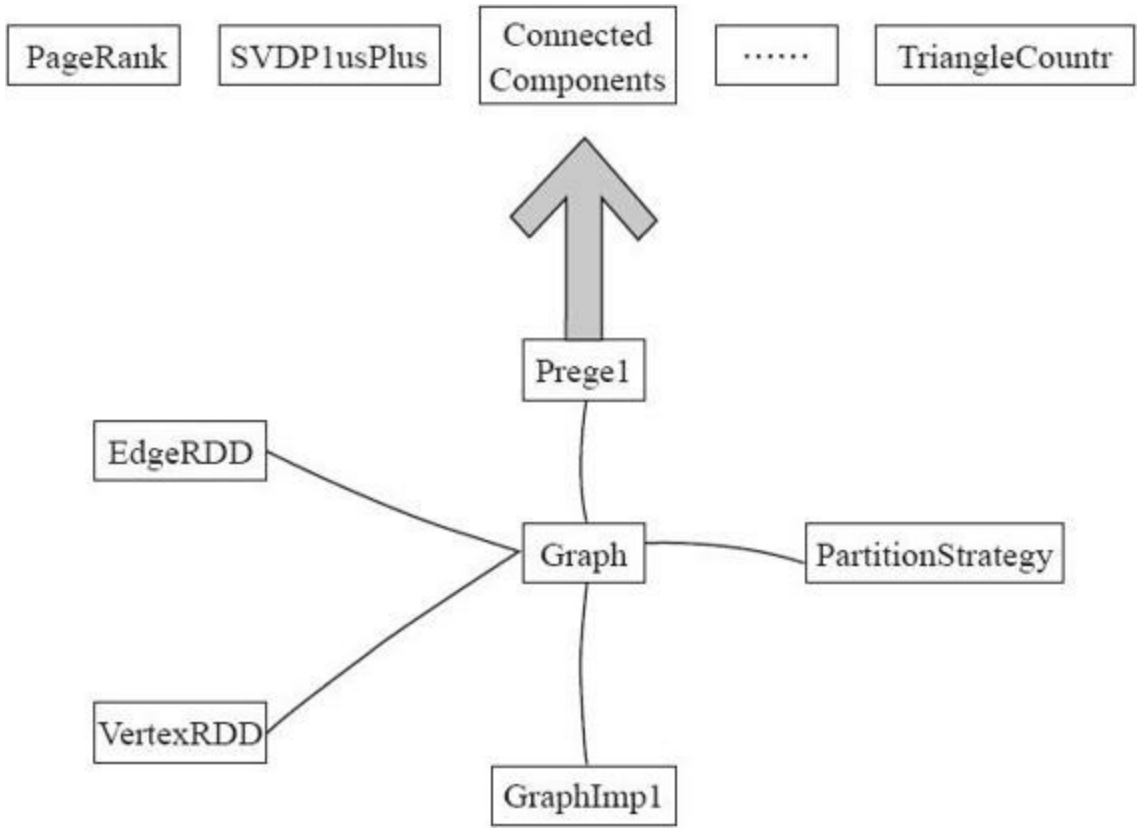

图8-21 GraphX架构

1）存储和原语层：Graph类是图计算的核心类，内部含有VertexRDD、EdgeRDD和RDD[EdgeTriplet]引用。GraphImpl是Graph类的子类，实现了图操作。

2）接口层：在底层RDD的基础之上实现了Pregel模型、BSP模式的计算接口。

3）算法层：基于Pregel接口实现了常用的图算法。包括：PageRank、SVDPlusPlus、TriangleCount、ConnectedComponents、StronglyConnectedConponents等算法。

2.存储结构

在正式的工业级应用中，图的规模极大，上百万个节点经常出现。为了提高处理速度和

数据量，希望能够将图以分布式的方式来存储、处理图数据。图的分布式存储大致有两种方式：边分割（edge cut）和点分割（vertex cut），如图8-22所示。最早期的图计算的框架中，使用的是边分割存储方式，而GraphX的设计者考虑到真实世界中的大规模图典型地是边多于点的图，所以采用点分割方式存储。点分割能够减少网络传输和存储开销。底层实现是将边放到各个节点存储，而在数据交换时，将点在各个机器之间广播进行传输。对边进行分区和存储的算法主要基于PartitionStrategy中封装的分区方法。其中的几种分区方法分别是对不同应用情景的权衡，用户可以根据具体的需求，在程序中指定边的分区方式。例如：

```
val g = Graph(vertices, partitionBy(
edges, PartitionStrategy.EdgePartition2D))
```

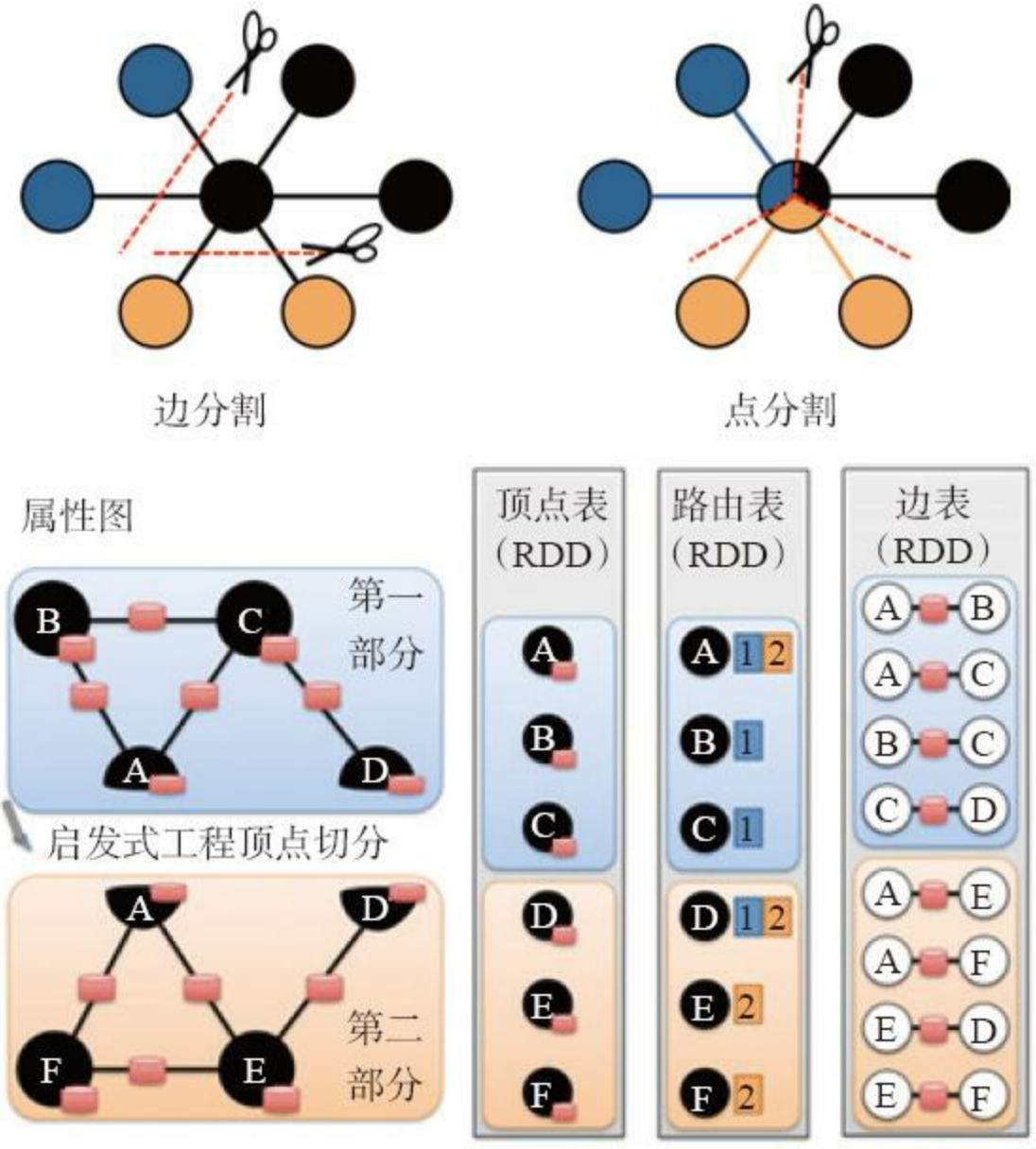

图8-22　GraphX存储模型

一旦边已经在集群上分区和存储，大规模并行图计算的关键挑战就变成了如何将点的属性连接到边。GraphX的处理方式是在集群上移动传播点的属性数据。由于不是每个分区都需要所有点的属性（因为每个分区只是一部分边），GraphX内部维持一个路由表（routing table），这样当需要广播点到需要这个点的边的所在分区时，就可以通过路由表映射，将需要的点属性传输到指定的边分区。

点分割的好处是在边的存储上没有冗余数据，而且对于某个点与其邻居的交互操作，只要满足交换律和结合律即可。例如，求顶点的邻接顶点权重的和，可以在不同的节点进行并行运算，最后汇总每个节点的运行结果，网络开销较小。代价是每个顶点属性可能要冗余存储多份，需要更新点数据时，要有数据同步开销。

3.使用技巧

采样观察可以通过不同的采样比例，先从小数据量进行计算，观察效果，调整参数，再逐步增加数据量进行大规模的运算。可以通过RDD的sample方法采样，通过Web UI观察集群的资源消耗。

1）内存释放：保留旧图对象的引用，并尽快释放不使用的图的顶点属性，节省空间占用。通过方法unPersistVertices释放顶点。

2）GC调优，请参考性能调优章节的内容。

3）调试：在各个时间点可以通过graph.vertices.count（）进行调试，观测图现有状态，进行问题诊断和调优。

8.3.4 运行实例

上文介绍了GraphX的组件和API。下面通过Berkeley的示例构建一个真实的图数据分析流水线。[1]示例中将会使用Wikipedia的连接数据，使用Spark的操作符清洗数据和抽取结构，使用GraphX操作符分析图结构，最后检验和评价图分析的结果。上面的操作可以通过Spark Shell执行。

GraphX为了达到最佳的性能表现需要使用Kyro序列化器。用户可以通过Spark的Web UI确认使用了哪种序列化器。在浏览器中输入链接（见图8-23）：http://<MASTER_URL>：4040/environment/，检查spark.serializer属性：

在默认情况下，Spark没有使用Kyro序列化器。在实战中，用户可以首先退出Spark Shell。

编译配置文件/root/spark/conf/spark-env.sh，在文件中加入下面的内容。

```
SPARK_JAVA_OPTS+='
 -Dspark.serializer=org.apache.spark.serializer.KryoSerializer
 -Dspark.kryo.registrator=org.apache.spark.graphx.GraphKryoRegistrator 'export SPARK_JAVA_OPTS
```

也可以使用下面的命令在命令行配置文件。

```
echo -e "SPARK_JAVA_OPTS+=' -Dspark.serializer=org.apache.spark.serializer.KryoSerializer -
Dspark.kryo.registrator=org.apache.spark.graphx.GraphKryoRegistrator ' \nexport SPARK_JAVA_OPTS" >>
/root/spark/conf/spark-env.sh
```

如果用户在ec2的环境下，则运行下面的命令将配置文件更新到集群中的所有机器中。

```
/root/spark-ec2/copy-dir.sh /root/spark/conf
```

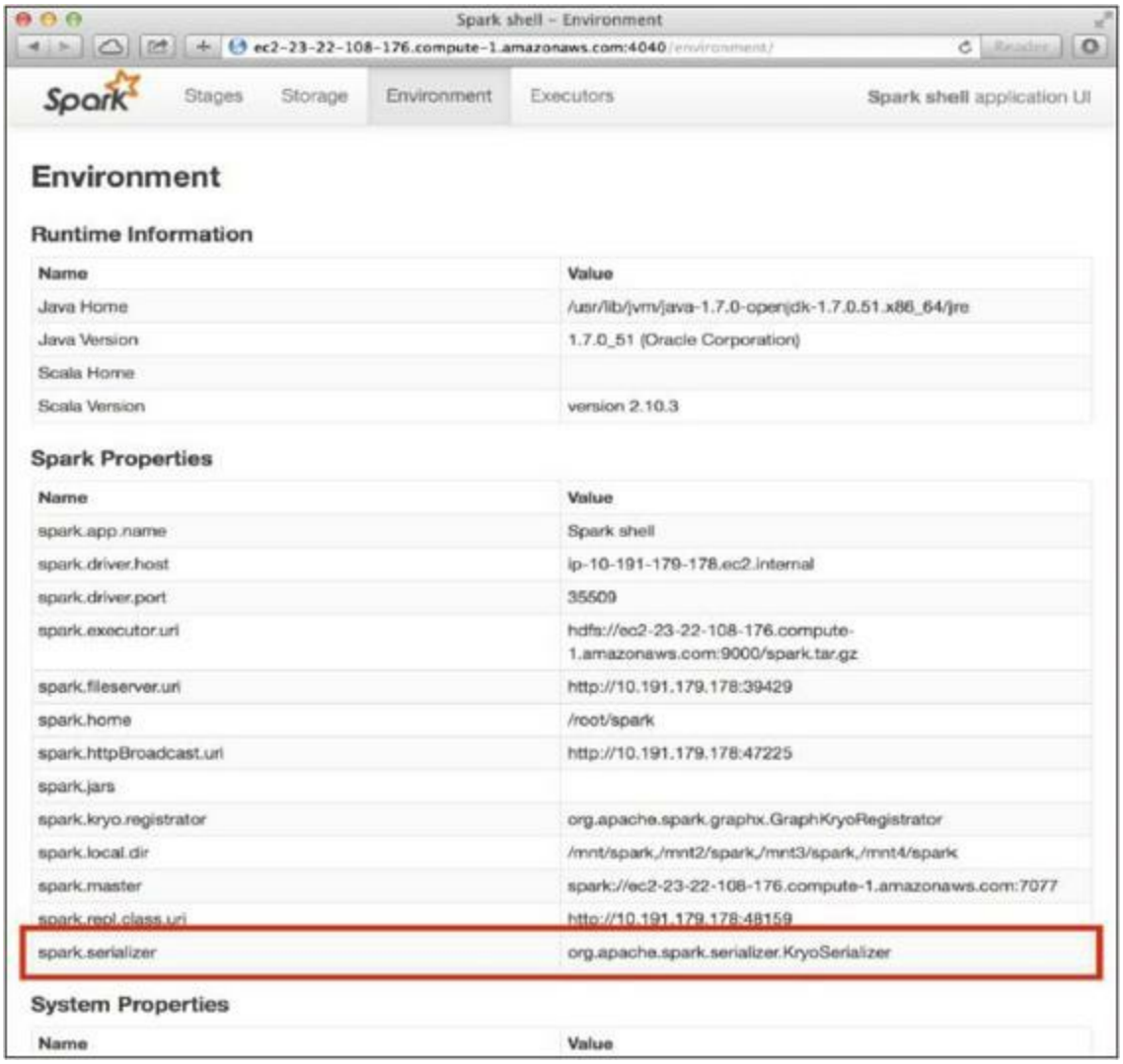

图8-23　Spark配置监测UI

如果用户是在其他环境下，则可以使用pssh等集群分发工具将/conf文件同步到所有机器中。例如，在pssh环境下：

```
./pscp -h hosts.txt -r   /root/spark/conf    /root/spark/conf
```

最终通过下面的命令重启整个集群。

```
/root/spark/sbin/stop-all.sh
sleep 3/root/spark/sbin/start-all.sh
```

在启动Spark Shell之后，查看http://<MASTER_URL>:4040/environment/中的spark.serializer属性。

如果配置成功，则显示已经配置为org.apache.spark.serializer.KryoSerializer.

下面开始应用开发流程。

（1）开始

启动Spark Shell。

```
/root/spark/bin/spark-shell
```

下面的命令都是在spark-shell中输入的，引入需要的标准包。

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```

（2）加载Wikipedia数据

加载原生数据进入Spark。假定读者已经将Wikipedia的数据加载进HDFS。下面将HDFS的数据加载进RDD。

将数据直接加载至内存，这样可以减少重复的磁盘IO开销。调用coalesce函数将分区紧缩至20个分区，以减少过量的通信开销。

```
val wiki： RDD[String] = sc.textFile("/wiki_links/part*").coalesce(20)
```

查看第一个数据项：用户可以通过RDD中的first方法呈现第一篇文章。

```
wiki.first
/* res0： String = AccessibleComputing [[Computer accessibility]]*/
```

（3）数据清洗

清洗数据和抽取图结构。在这个例子中，将会抽取连接图，也可以在其他数据集迁移示例（如文档中的关键词图、贡献者图等）。在已经采样的数据中，可以观察到一些数据中蕴

含的结构。每一行的第一个词是文章的名称，其余字符串包含这个文章中的链接。

使用已经观察到的结构进行数据清理。

```
/*定义Article类 */
case class Article(val title: String, val body: String)
/*根据分隔符\t分隔文章 */
val articles = wiki.map(_.split('\t')).
  /* 过滤字符串，保留字符串长度大于1，且第二个字符串中包含REDIRECT关键字的元组 */
  filter(line => (line.length > 1 && !(line(1) contains "REDIRECT"))).
  /*将结果存储到对象中，便于后续访问*/
  map(line => new Article(line(0).trim, line(1).trim)).cache
  程序可以通过count函数查看清洗后文章的剩余数量。
articles.count
```

（4）创建一个顶点RDD

此时，数据已经清洗，可以创建顶点RDD。由于之前的目的是从数据中抽取链接图，所以顶点的一个自然属性就是文章的标题。之后仍需要将文章标题哈希映射为顶点ID。

```
//将文章标题转换为ID的哈希函数
def pageHash(title: String): VertexId = {
  title.toLowerCase.replace(" ", "").hashCode.toLong
}
// 数据项为ID和文章标题的顶点RDD
/* val vertices: RDD[(VertexId, String)] = /* implement */*/
/*创建顶点RDD*/
val vertices = articles.map(a => (pageHash(a.title), a.title)).cache
/*通过 Action算子count强制触发RDD的执行*/
vertices.count
```

（5）创建边RDD

数据清洗的下一步是抽取边，进而构建连接图的结构。MeidaWiki语法中是以格式“[[链接到的文章]].”存储数据，用户可以通过正则表达式抽取[[]]中的内容。通过上面的方法抽取所有的文章中包含的链接，返回包含在文章中的所有链接。

读者可以将下面的代码拷贝到Spark shell中执行。

```
val pattern = "\\[\\[.+?\\]\\]".r
val edges: RDD[Edge[Double]] = articles.flatMap { a =>
  val srcVid = pageHash(a.title)
  pattern.findAllIn(a.body).map { link =>
    val dstVid = pageHash(link.replace("[[", "").replace("]]", ""))
    Edge(srcVid, dstVid, 1.0)
  }
```

```
}
```

这段代码抽取每个页面的所有导出链接，然后给边RDD的每个边分配一个统一的权重。

（6）创建图

之前的准备工作完成后，就可以开始创建图了。之前的准备工作是使用Spark的核心代码以关系表的视角创建数据。使用之前的顶点RDD、边RDD和默认的顶点属性创建图。默认的顶点属性是为了初始化那些现在还未在顶点RDD中的顶点。但是Wikipedia数据中经常有链接指向不存在的页面。在本例中将会使用一个空的文章标题字符串作为默认的顶点属性代表一个损坏的链接。注意：之所以进行这样的考虑，是因为在真实的数据中可能有脏数据。

```
val graph = Graph(vertices, edges, "").subgraph(vpred = {(v, d) => d.nonEmpty}).cache
```

通过Action算子count强制计算图的一些属性（这大约会耗费2分钟）。

```
graph.vertices.count
```

在第一次创建图时，GraphX针对所有图中的顶点创建索引，发现并重新分配丢失的顶点。由于创建了索引，所以会更快地计算图中的三元组。

```
graph.triplets.count
```

（7）在Wikipedia数据集上运行PageRank算法

到此为止，用户可以进行一些实际的图分析操作了。在这个例子中，将会运行PageRank计算图中最重要的页面。

首先需要进行一些初始化工作。

```
val prGraph=graph.staticPageRank(5).cache
```

Graph.staticPageRank方法返回的顶点属性是每个页面PageRank值的图。这里读者可能会怀疑，新的图属性包含的是PageRank值，不再包含原始的顶点属性文章标题了。其实之前的包含文章标题顶点属性的prGraph仍然存在，可以将两个图Join，这样就会返回一个包含两类信息的一个新图，两个图的所有顶点属性将以元组的形式存储作为一个新顶点属性。之后在这个新的顶点序列中还可以进行更深入的基于表的操作，如找到最重要的10个顶点（也就是PageRank值最高的10个顶点），并打印出它们的标题。将这些操作融合在一起就实现了在WiKi文章图中找到最重要的10个文章标题的功能。

```
val titleAndPrGraph = graph.outerJoinVertices(prGraph.vertices)
{  (v, title, rank) => (rank.getOrElse(0.0), title)
}
titleAndPrGraph.vertices.top(10) {
  Ordering.by((entry: (VertexId, (Double, String))) => entry._2._1)
}.foreach(t => println(t._2._2 + ": " + t._2._1))
```

最后，可以通过下面的例子，在提到“Berkeley”的文章所构成的图中，找到最重要的页面。

```
val berkeleyGraph = graph.subgraph(vpred = (v, t) => t.toLowerCase contains "berkeley")
val prBerkeley = berkeleyGraph.staticPageRank(5).cache
berkeleyGraph.outerJoinVertices(prBerkeley.vertices) {
  (v, title, r) => (r.getOrElse(0.0), title)
}.vertices.top(10) {
  Ordering.by((entry: (VertexId, (Double, String))) => entry._2._1)
}.foreach(t => println(t._2._2 + ": " + t._2._1))
```

读者可以以上面的方式在GraphX平台进行更加深入的数据分析。

[1] 示例参考：http://ampcamp.berkeley.edu/big-data-mini-course/graph-analytics-with-graphx.html

8.4 MLlib

MLlib是构建在Spark上的分布式机器学习库，充分利用了Spark的内存计算和适合迭代型计算的优势，使性能大幅度提升，同时Spark算子丰富的表现力，让大规模机器学习的算法开发不再复杂。在正式介绍ML1ib之前，先介绍一个拓展知识——数据挖掘组件化思想。

知识拓展：数据挖掘组件化思想

针对不同的数据挖掘任务，研究人员设计了各种各样的数据挖掘算法。与此同时，每年仍有大量新的数据挖掘算法产生。对于数据挖掘初学者来说，弄清算法之间的联系和区别很困难。可以通过数据挖掘组件化思想（Hand et al，2001）来拆分和理解数据挖掘算法，理清其中的脉络。问题可拆解，意味着问题组件解耦，每个组件都可以重新设计组合，从而衍生出多种多样的新算法。该思想认为，许多数据挖掘算法由5个"标准组件"构成，即模型模式、模数据挖掘任务、数据挖掘任务、评分函数、搜索和优化方法、数据管理策略。每一种组件都含有一些通用的方法或者模型。例如，模型模式中包括马尔科夫链、贝叶斯网络、神经网络等。理解了每一个组件的基本原理，再来理解多个组件组合起来的算法会更加容易，而且不同算法之间进行比较能更加容易地看出异同。下面介绍这5个组件。

1.模型或者模式结构

训练数据相当于数据挖掘算法的输入，而模型（model）或者模式（pattern）结构相当于数据挖掘算法的输出，如决策树模型、频繁序列模式。

模型是对数据集合全局整体的描述，模式是对数据集合局部集合的描述。模型与模式也是联系的。例如，聚类算法用于异常点检测的原理就是异常点监测的局部模式只有和全局的正常模型比较，才能暴露出异常。模型和模式都含有参数，例如，$Y=aX^2+bX+c$参数是a、b、c，模式$X>d$、$Y<e$的概率为p的参数为d、e、p。把参数不确定的模式（如d、e、p取不同的值）叫模式的结构。当参数确定，就说明这个模型或者模式已经拟合。结构的参数不确

定，是一般形式，拟合的模型模式是具有了特定参数值的特殊形式。

2.数据挖掘任务

根据对全局集合的描述状况可以将数据挖掘任务分为模式挖掘、回归分类、描述建模。

（1）模式挖掘

模式是对某个数据集中部分结构的描述，可以是一个子集合、一个子序列、一个子树和一个子图。例如，交易数据中的啤酒、尿布就是频繁项集，是一个子集合。

（2）回归分类

回归分类根据现有数据建立模型，然后应用这个模型对未来数据进行预测。在预测模型中，一个变量表达为其他变量的函数。因此，可以把预测建模的过程看做是学习一种映射或函数Y=f（X）。相当于在整个数据集合看，数据集合分为现有数据和未来待处理数据，回归分类是对现有数据集合的描述，然后估计近似认为整个数据集的描述，待处理数据会根据历史数据得出模型进行处理。

（3）描述建模

描述建模要获取对现有数据集合的全局结构的描述。

3.评分函数

当确定了模型或者模式的结构之后，下一步就是需要确定结构中的参数，将结构拟合到数据。由于模型或者模式的结构是函数的一般形式，参数的空间很大，可选择的参数非常多，所以需要一个评分函数作为评价指标，确定哪个参数组合才能达到最优的结果。常用的评分函数有误差平方和、准确率召回率和ROC等。

4.搜索和优化方法

确定了评分函数，也就有了衡量数据和模型或者模式拟合程度的标准。搜索和优化方法就是为了确定模型模式结构的参数值。如果模型模式结构没有确定，则需要使用搜索方法（如贪婪法、深度优先遍历等），从模型模式的集合中发现最优的模型模式结构，还需要确定最优的结构参数。对于固定的模型模式结构，搜索就在参数空间进行，使用的是优化方法（如爬山法、期望最大化法等）。

5.数据管理策略

通常在大数据环境下，数据不能被单机容纳，需要分布式存储。这就需要设计好数据管理策略优化提升算法性能。针对大数据可以采用、近似、压缩、索引等技术提升机器学习算法效率。

组件化思想十分重要。将算法分解，揭示了算法的本质。这样学习相应机器学习算法，就不再是无穷无尽的算法，机器学习人员应该面对新应用，根据需求决定选取哪个组件中的哪个方法来组合出一个新的算法，而不是考虑选用哪个现成的算法。

8.4.1 MLlib简介

MLlib是一些常用的机器学习算法和库在Spark平台上的实现，是AMPLab的在研机器学习项目MLBase的底层组件。MLBase是一个机器学习平台，MLI是一个接口层，提供很多结构，MLlib是底层算法实现层。三者关系如图8-24所示。

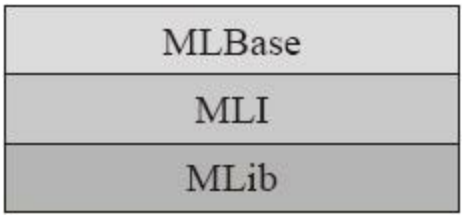

图8-24 MLbase

MLlib在Spark 1.0中包含分类与回归、聚类、协同过滤、数据降维组件以及底层的优化库。

通过图8-25，读者可以对MLlib的整体组件和依赖库有宏观的把握。

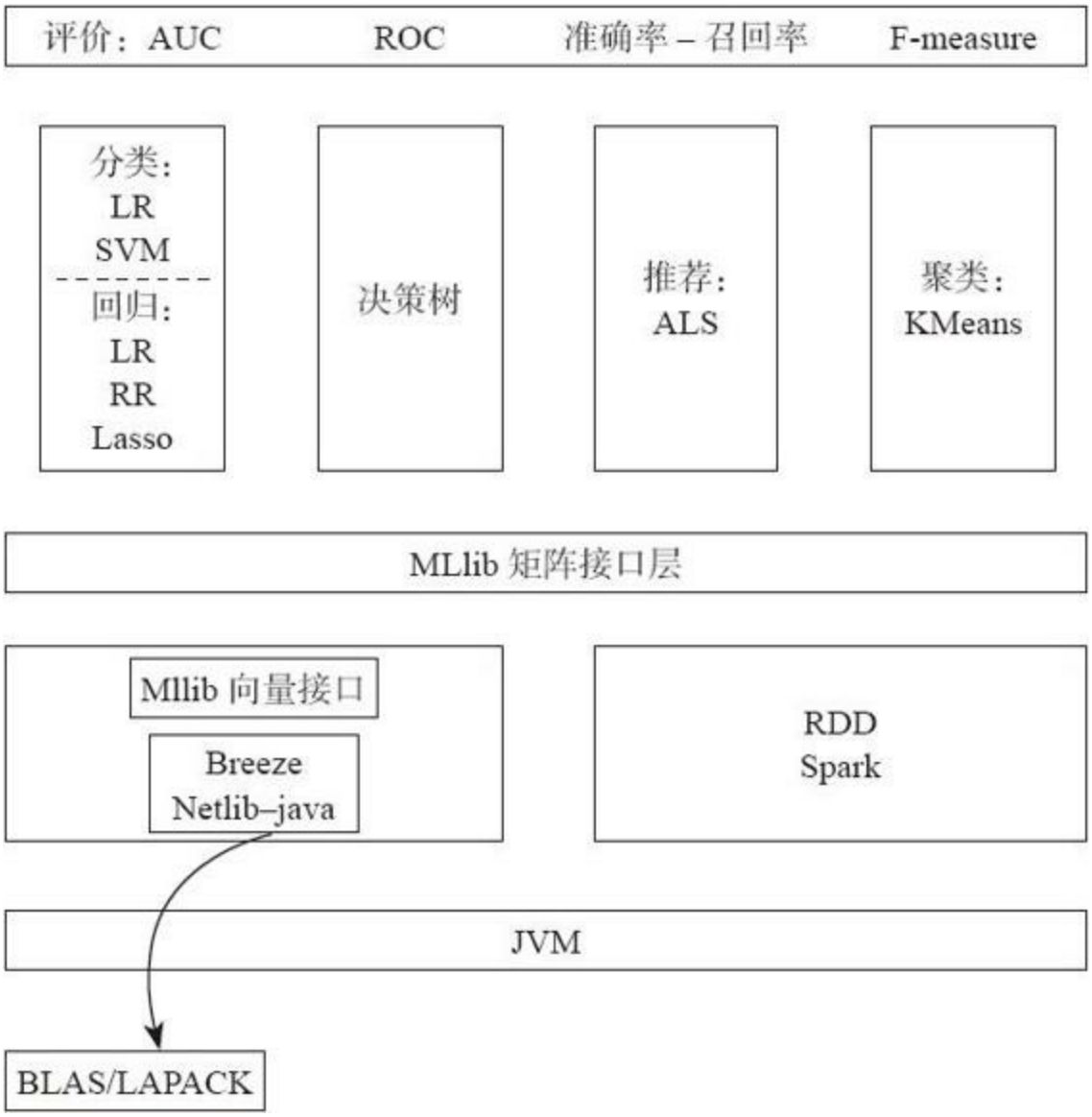

图8-25　MLlib组件

下面简要介绍图8-25中，读者可能不太熟悉的底层组件。

BLAS/LAPACK层：LAPACK是用Fortran编写的算法库，顾名思义，Linear Algebra PACKage是为了解决通用的线性代数问题的。另外必须要提的算法包是BLAS（Basic Linear Algebra Subprograms），其实LAPACK底层使用了BLAS库。不少计算机厂商都提供了针对不同处理器进行了优化的BLAS/LAPACK算法包。

Netlib-java（官网地址为https://github.com/fommil/netlib-java/）是一个对底层BLAS、LAPACK封装的Java接口层。

Breeze（官网地址为：https://github.com/scalanlp/breeze）是一个Scala编写的数值处

理库，提供向量、矩阵运算等API。

库依赖：MLlib底层使用了Scala编写的线性代数库Breeze，Breeze底层依赖netlib-java库。netlib-java底层依赖原生的Fortran routines。因此，当用户使用时，需要在节点上预先安装gfortran runtime library（下载地址为https://github.com/mikiobraun/jblas/wiki/Missing-Libraries）。由于许可证（license）问题，官方的MLlib依赖集中没有引入netlib-java原生库的依赖。如果运行时环境没有可用原生库，就会看到警告信息。如果程序中需要使用netlib-java的库，就需要在项目中引入com.github.fommil.netlib：all：1.1.2的依赖或者参照指南（网址为https://github.com/fommil/netlib-java/blob/master/README.md#machine-optimised-system-libraries）build用户自己的项目。如果需要使用python接口，则需要1.4或者更高版本的NumPy。（注意：MLlib源码中注释有Experimental/DeveloperApi的API在未来的发布版本中可能会调整和改变，官方会在不同版本发布时提供迁移指南。）

8.4.2 MLlib的数据存储

MLlib支持存储在本地的向量和矩阵，也提供分布式的矩阵（底层实现是一个或多个RDD）。[1]在目前发布版本的实现中，本地的向量和矩阵数据模型提供公共服务接口，基础的线性代数操作是基于Breeze和jblas库的。在MLlib监督学习中的一个训练样例叫做"标记向量"（labeled point）。

1.本地向量或矩阵

（1）本地向量

一个本地向量内的数据类型为double，并且数组序号是从0开始（0-based indices）的整数类型。本地向量存储在单机中。MLlib支持两种类型的本地向量：稠密向量和稀疏向量。稠密向量的底层实现是一个double型的数组存储向量每个元素的值，一个稀疏向量的底层实现是两个并行的数组，一个数组存储向量的序号，一个存储向量元素值。例如，向量（1.0，0.0，3.0）在稠密向量的存储是[1.0，0.0，3.0]，而在稀疏矩阵中的存储是（3，[0，2]，[1.0，3.0]），如图8-26所示。这里的3代表向量的维数。图8-26所示为两种向量存储模式。

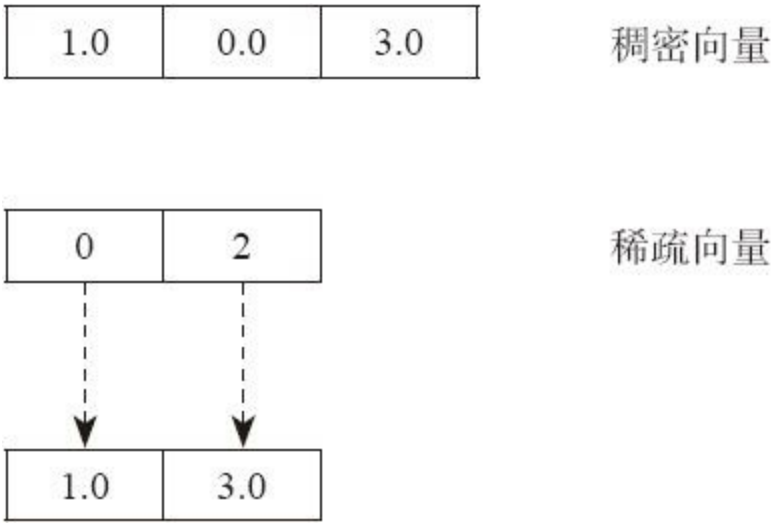

图8-26　稠密向量与稀疏向量存储

本地向量的基本类是Vector类，官方提供了Vector类的两种实现：稠密向量（dense vector）和稀疏向量（sparse vector）。官方推荐使用Vectors类中提供的工厂模式的方法创建本地向量。

下面看官方的向量创建例子。

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}
// 创建一个密集向量 (1.0, 0.0, 3.0)
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
/* 创建一个稀疏向量 (1.0, 0.0, 3.0) by specifying its indices and values corresponding to nonzero entries */
val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))
/* 创建一个密集向量 (1.0, 0.0, 3.0) by specifying its nonzero entries */
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

注意：由于Scala默认情况下引入了import scala.collection.immutable.Vector库，所以需要引入import org.apache.spark.mllib.linalg.Vector库区显式使用MLlib内的向量。

（2）标记向量

一个标记向量（labeled point）是一个本地向量，可以是稠密向量，也可以是稀疏向量，并和一个标记（label）相关联。在MLlib中，标记在监督学习的算法（如分类和回归算法）中使用。在标记向量中使用一个Double类型数据存储标记，即可在回归和分类中都可以使用标记向量。对于二元分类来说，label可以为0（代表negative）或者1（代表positive）。对于多元分类问题，标记可以使用从0开始的序号：0，1，2……

一个标记向量可以使用case类LabeledPoint来存储和呈现。

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
/* 创建一个有正标记和稠密特征向量的标记点 */
val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))
/* 创建一个有负标记和稀疏特征向量的标记点 */
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0)))
```

（3）稀疏数据

常见的情况是使用稀疏数据训练模型。下面介绍稀疏数据格式。MLlib支持读取LIBSVM

格式（一种文本格式）的训练数据，这种数据默认被LIBSVM和LIBLINEAR使用。文件的每一行代表一个被标记的稀疏特征向量，它的格式请参考：

```
label index1:value1 index2:value2 ...
```

默认序号索引是从1开始并且是升序的，加载之后，特征的序号被转换为从0开始。

可以使用MLUtils.loadLibSVMFile方法读取存储为LIBSVM格式的训练数据。

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.rdd.RDD
val examples: RDD[LabeledPoint] = MLUtils.loadLibSVMFile(sc, "mllib/data/sample_libsvm_data.txt")
```

（4）本地矩阵

一个本地矩阵也存储double型的数据，使用整型的行号和列号。MLlib支持稠密矩阵，稠密矩阵的值存储在一个单独的double数组中，以列序为主序存储。例如，下面的矩阵。

```
(1.0 2.0)
(3.0 4.0)
(5.0 6.0)
```

底层存储是以一维数组中存储数据（如[1.0，3.0，5.0，2.0，4.0，6.0]）和存储矩阵的行列大小，如（3，2）。在Spark 1.0只提供稠密矩阵，官方将在下一个版本提供稀疏矩阵的实现。本地矩阵的基本类是Matrix，官方目前提供了一种实现：DenseMatrix。官方推荐使用Matrices类中的工厂方法来创建本地矩阵。

```
import org.apache.spark.mllib.linalg.{Matrix, Matrices}
/* 创建一个密集矩阵 ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))*/
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
```

2.分布式矩阵

一个分布式的矩阵存储的是double类型的数据，底层采用一个或者多个RDD存储，行列

序号采用long型存储。存储分布式大数据量矩阵的关键问题是选取正确的数据格式。将一个分布式矩阵转换为另一种不同的格式需要一个全局的Shuffle操作。在分布式环境下，Shuffle开销很大。官方已经实现了3种类型的分布式矩阵，将会在未来提供更多类型矩阵的实现。分布式矩阵的基本类型是RowMatrix。

一个RowMatrix是一个面向行的分布式矩阵。例如，一个特征向量集合的底层实现是一个RDD，RDD的每个元素是一个本地特征向量。在这种情况下，对RowMatrix来说是假设用户的特征矩阵维数不高。一个IndexedRowMatrix和RowMatrix相似，但是会存储行的序号，行序号可以确定行以及有助于进行连接操作。一个CoordinateMatrix是一个分布式矩阵以coordinate list（COO）格式存储，底层也是以存储它的数据项的一个RDD实现的。

注意：因为已经缓存了整个矩阵数据大小的空间，分布式矩阵底层实现的RDD必须是确定的。如果用非确定性的RDD，就很容易出错。

（1）行矩阵

一个行矩阵是面向行存储的分布式矩阵，底层是一个以本地行向量为数据项的一个RDD。由于每个行是一个本地向量，以long型为行序号，所以向量的维数会受数据类型范围限制，但在实际情况下，向量维数会小于这个范围。

一个RowMatrix可以从RDD[Vector]实例创建，然后可以计算行列来统计数据。

```
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val rows： RDD[Vector] = ... // an RDD of local vectors
/* 通过 RDD[Vector]创建一个行矩阵*/
val mat： RowMatrix = new RowMatrix(rows)
/* 获取矩阵大小*/
val m = mat.numRows()
val n = mat.numCols()
```

（2）行索引矩阵

一个行索引矩阵（indexed Row matrix）的底层实现是一个带行索引的RDD，这个RDD

每行是一个长整型的索引和本地向量。一个行索引矩阵可以通过RDD[IndexedRow]的RDD创建，一个indexed row是一个元组（Long，Vector）的封装，Long代表索引，Vector代表本地向量。一个indexed row matrix通过消除行索引可以转换为row matrix。

（3）坐标矩阵

坐标矩阵（coordinate matrix）是一个分布式矩阵，其底层实现也是使用一个RDD存储它的数据项。每个数据项是一个元组（i：Long，j：Long，value：Double），其中，i是行序号索引，j是列序号索引，value是数据项的值。一个坐标矩阵适用于矩阵的行列维度都很大，但矩阵数据很稀疏的情况。

一个坐标矩阵可以从RDD[MatrixEntry]实例创建，matrix entry是一个（Long，Long，Double）的封装。一个coordinate matrix可以通过调用方法toIndexedRowMatrix被转换成含有稀疏行的index row matrix。在Spark MLlib 1.0中，官方不提供针对coordinate matrix的其他计算方法。

[1] 参见Spark官网：https://spark.apache.org/docs/latest/mllib-basics.html。

8.4.3 数据转换为向量（向量空间模型VSM）

因为机器学习的本质主要是进行矩阵运算，所以在特定的应用领域，需要用户建模，并将领域数据转化为向量或矩阵形式。下面介绍文本处理中的一个常用模型：向量空间模型（VSM）。

向量空间模型将文档映射为一个特征向量V（d）=（t1，ω1（d）；…；tn，ωn（d）），其中ti（i=1，2，…，n）为一列互不雷同的词条项，ωi（d）为ti在d中的权值，一般被定义为ti在d中出现频率tfi（d）的函数，即ωi（d）=ψ（tfi（d））。

在信息检索中，TF-IDF函数ψ-tfi（d）×$^{\log \frac{(N)}{ni}}$是常用的词条权值计算方法，其中N为所有文档的数目，ni为含有词条ti的文档数目。TF-IDF公式有很多变种，以下是一个常用的TF-IDF公式。

$$\omega_i(d)=\frac{tf_i(d)\log(\frac{N}{n_j}+0.1)}{\sqrt{\sum_{n-1}^{n}(tf_i(d))^2\times\log^2\left(\frac{N}{n_j}+0.1\right)}}$$

根据TF-IDF公式，文档集中包含某一词条的文档越多，说明该文档区分文档类别属性的能力越低，其权值越小；另一方面，某一文档中某一词条出现的频率越高，说明该文档区分文档内容属性的能力越强，其权值越大。

可以用其对应的向量之间的夹角余弦来表示两文档之间的相似度，即文档di、dj的相似度可以表示为：

$$sim(d_i,d_j)=\cos\theta=\frac{\sum_{t=1}^{n}\omega_k(d_d)\times\omega_k(d_j)}{\sqrt{\left(\sum_{k=1}^{n}\omega_k^2(d_d)\right)\left(\sum_{k=1}^{n}\omega_k^2(d_j)\right)}}$$

在查询过程中，先将查询条件Q进行向量化，主要依据以下布尔模型。

当ti在查询条件Q中时，将对应的第i坐标置为1，否则置为0，即

$$q_i=\begin{cases}1 & t_i\in Q\\ 0 & t_i\notin Q\end{cases}$$

文档d与查询Q的相似度为：

$$sim(Q,d)=\frac{\sum_{d=1}^{n}\omega_i(d)\times q_i}{\sqrt{\left(\sum_{d=1}^{n}\omega_d^2(d)\right)\left(\sum_{d=1}^{n}q_d^2\right)}}$$

根据文档之间的相似度，结合机器学习的神经网络算法，K-近邻算法和贝叶斯分类算法等一些算法，可将文档集划分为一些小的文档子集。

在查询过程中，可以计算出每个文档与查询的相似度，进而根据相似度的大小，对查询的结果进行排序。

向量空间模型可以实现文档的自动分类，并对查询结果的相似度排序，这能够有效提高检索效率。其缺点是相似度的计算量大，当有新文档加入时，必须重新计算词的权值。

8.4.4 MLlib中的聚类和分类

聚类和分类是机器学习中两个常用的算法，聚类将数据分开为不同的集合，分类预测新数据类别，下面介绍这两类算法。

1.什么是聚类和分类

（1）什么是聚类分析

聚类（Clustering）是指将数据对象分组成为多个类或者簇（Cluster），它的目标是：在同一个簇中的对象之间具有较高的相似度，而不同簇中对象的差别较大。其实，聚类在人们日常生活中是一种常见的行为，即所谓的"物以类聚，人以群分"，其核心思想在于分组，人们不断地改进聚类模式来学习如何区分各个事物和人。

（2）什么是分类分析

数据仓库、数据库或者其他信息库中有许多可以为商业、科研等活动提供决策的知识。分类与预测即是其中的两种数据分析形式，可以用来抽取能够构建分类模型描述重要数据集合或预测未来数据趋势。分类方法（classification）用于预测数据对象的离散类别（categorical Label）；预测方法（prediction）用于预测数据对象的连续取值。

·分类流程：新样本→特征选取→分类→评价。

·训练流程：训练集→特征选取→训练→分类器。

最初，机器学习的分类应用大多都是在这些方法及基于内存基础上所构造的算法。目前，数据挖掘方法都要求具有处理大规模数据集合能力，同时具有可扩展能力。

2.MLlib中的聚类和分类

MLlib目前已经实现了K-Means聚类算法、朴素贝叶斯和决策树分类算法。这里主要介绍

广泛使用的K-Means聚类算法和贝叶斯分类算法。

（1）K-Means算法

1）K-Means算法简介。

K-Means聚类算法能轻松地对聚类问题建模。K-Means聚类算法容易理解，并且能在分布式的环境下并行运行。学习K-Means聚类算法，能更容易地理解聚类算法的优缺点，以及其他算法对于特定数据的高效性。

K-Means聚类算法中的K是聚类的数目，在算法中会强制要求用户输入。如果将新闻聚类成诸如政治、经济、文化等大类，可以选择10~20的数字作为K。因为这种顶级类别的数量是很小的。如果要对这些新闻详细分类，选择50~100的数字也没有问题。K-Means聚类算法主要分为3步。第一步是为待聚类的点寻找聚类中心；第二步是计算每个点聚类中心的距离，将每个点聚类到离该点最近的聚类中；第三步是计算聚类中所有点的坐标平均值，并将这个平均值作为新的聚类中心点。反复执行第二步，直到聚类中心不再进行大范围的移动，或者聚类次数达到要求为止。

2）k-Means示例。

表8-7中的例子有7名选手，每名选手有两个类别的比分：A类比分和B类比分。

表8-7　A类和B类比分

Subject	A	B	Subject	A	B
1	1.0	1.0	5	3.5	5.0
2	1.5	2.0	6	4.5	5.0
3	3.0	4.0	7	3.5	4.5
4	5.0	7.0			

这些数据将会聚为两个簇。随机选取1号和4号选手作为簇的中心。1号和4号选手信息如表8-8所示。

表8-8　1号和4号选手信息

组名 \ 属性	Individual	Mean Vector (centroid)
Group 1	1	(1.0, 1.0)
Group 2	4	(5.0, 7.0)

第一步将1号和4号选手分别作为两个簇的中心点，下面每一步将选取点和两个簇中心计算欧几里得距离，和哪个中心距离小就放到哪个簇中。表8-9所示为第一聚类。

表8-9　第一步聚类

步骤 \ 簇名	Cluster 1		Cluster 2	
Step	Individual	Mean Vector (centroid)	Individual	Mean Vector (centroid)
1	1	(1.0, 1.0)	4	(5.0, 7.0)
2	1, 2	(1.2, 1.5)	4	(5.0, 7.0)
3	1, 2, 3	(1.8, 2.3)	4	(5.0, 7.0)
4	1, 2, 3	(1.8, 2.3)	4, 5	(4.2, 6.0)
5	1, 2, 3	(1.8, 2.3)	4, 5, 6	(4.3, 5.7)
6	1, 2, 3	(1.8, 2.3)	4, 5, 6, 7	(4.1, 5.4)

第一轮聚类的结果产生了，如表8-10所示。

表8-10　第一轮结果

簇名 \ 属性	Individual	Mean Vector (centroid)
Cluster 1	1, 2, 3	(1.8, 2.3)
Cluster 2	4, 5, 6, 7	(4.1, 5.4)

第二轮将使用（1.8，2.3）和（4.1，5.4）作为新的簇中心，重复上面的过程，直到迭代次数达到用户设定的次数为止。最后一轮迭代分出的两个簇就是最后的聚类结果。

3）MLlib的KMeans源码解析。

MLlib中的KMeans初始的类簇中心点的选取有两种方法，一种是随机，一种是采用KMeans||（KMeans++的一个变种）。算法的停止条件是迭代次数达到设置的次数，或者在某一次迭代后，所有run的KMeans算法都收敛。

①类簇中心初始化。

MLlib中KMeans初始化方法是对每个运行的KMeans都随机选择K个点作为初始类簇。代码实现如下。

```
private def initRandom(data: RDD[Array[Double]]): Array[ClusterCenters] = {
    // Sample all the cluster centers in one pass to avoid repeated scans
    val sample = data.takeSample(true, runs * k, new Random().nextInt()).toSeq
    Array.tabulate(runs)(r => sample.slice(r * k, (r + 1) * k).toArray)
  }
```

②计算属于某个类簇的点。

在每一次迭代中，首先计算属于各个类簇的点，然后更新各个类簇的中心。

```
/* KMeans算法的并行实现通过Spark 的mapPartitions函数获取到分区的迭代器。可以在每个分区内计算该分区内的点属于哪个类簇，之后对于每个运行算法中的每个类簇，计算属于该类簇的点数以及累加和 */
      val totalContribs = data.mapPartitions { points =>
        val runs = activeCenters.length
        val k = activeCenters(0).length
        val dims = activeCenters(0)(0).length
        val sums = Array.fill(runs, k)(new DoubleMatrix(dims))
        val counts = Array.fill(runs, k)(0L)
        for (point <- points; (centers, runIndex) <- activeCenters.zipWithIndex) {
          /*找到距离该点最近的类簇中心点 */
          val (bestCenter, cost) = KMeans.findClosest(centers, point)
          /*统计该运行算法开销， 用于在之后选取开销最小的运行的算法 */
          costAccums(runIndex) += cost
          /*将该点加到最近的类簇的统计总和中，方便之后计算该类簇的新中心点 */
          sums(runIndex)(bestCenter).addi(new DoubleMatrix(point))
         /*将距离该点最近的类簇的点数加1，sum.divi(count)就是类簇的新中心 */
          counts(runIndex)(bestCenter) += 1
        }
        val contribs = for (i <- 0 until runs; j <- 0 until k) yield
{          ((i, j), (sums(i)(j), counts(i)(j)))
        }
         contribs.iterator
/*对于每个运行算法的每个类簇，计算属于该类簇的点，并对点个数求和 */
}.reduceByKey(mergeContribs).collectAsMap()
/* mergeContribs是一个负责合并的函数。 */
 def mergeContribs(p1: WeightedPoint, p2: WeightedPoint): WeightedPoint = {
        (p1._1.addi(p2._1), p1._2 + p2._2)
}
```

③更新类簇的中心点。

```
for ((run, i) <- activeRuns.zipWithIndex) {
       var changed = false
       for (j <- 0 until k) {
         val (sum, count) = totalContribs((i, j))
         if (count != 0) {
           /*计算类簇的新中心点 */
           val newCenter = sum.divi(count).data
           if (MLUtils.squaredDistance(newCenter, centers(run)(j)) > epsilon *
epsilon) {
           /*此处代码和算法的停止条件有关 */
             changed = true
          }
```

```
            centers(run)(j) = newCenter
          }
        }
      /* 如果在某个run的KMeans算法的某轮次迭代中，K个类簇的中心点变化都不超过指定阈值， 则认为该KMeans算法收敛 */
      if (!changed) {
       active(run) = false
       logInfo("Run " + run + " finished in " + (iteration + 1) + " iterations")
      }
      costs(run) = costAccums(i).value
    }
```

④算法停止条件。

算法的停止条件是迭代次数达到设置的次数，或者所有运行的KMeans算法都收敛。

```
while (iteration < maxIterations && !activeRuns.isEmpty)
```

（2）朴素贝叶斯分类

朴素贝叶斯分类算法是贝叶斯分类算法的多个变种之一。朴素是指假设各属性之间是相互独立的。研究发现，在大多数情况下，朴素贝叶斯分类算法（Naive Bayes Classifier）在性能上与决策树（decision tree）、神经网络（netural network）相当。贝叶斯分类算法在大数据集的应用中具有方法简便、准确率高和速度快的优点。但事实上，贝叶斯分类也有其缺点。由于贝叶斯定理假设一个属性值对给定类的影响独立于其他的属性值，因此假设在实际情况经常是不成立的，其分类准确率可能会下降。

朴素贝叶斯分类算法是一种监督学习算法，使用朴素贝叶斯分类算法对文本进行分类主要有两种模型，即多项式模型（multinomial model）和伯努利模型（Bernoulli model）。MLlib使用广泛应用的多项式模型。下面将以实例简单介绍使用多项式模型的朴素贝叶斯分类算法。

在多项式模型中，设某文档d=（t1，t2，…，tk），tk是该文档中出现过的单词，允许重复。

先验概率P（c）=类c下单词总数/整个训练样本的单词总数

类条件概率P（tk|c）=（类c下单词tk在各个文档中出现的次数之和+1）/（类c下单词总数+|V|）

V是训练样本的单词表（即抽取单词，单词出现多次，只算一个），|V|表示训练样本包含多少种单词。P（tk|c）可以看做是单词tk在证明d属于类c上提供了多大的证据，P（c）则可以认为是类别c在整体上占多大比例（有多大可能性）。

给定如下一组分好类的文本训练数据如表8-11所示。

表8-11　文本训练数据

文档 id	文档内容	类别是否为河北
1	河北　北京　河北	yes
2	河北　河北　上海	yes
3	河北　广东	yes
4	吉林　香港　河北	no

给定一个新样本（河北河北河北吉林香港），对其进行分类。该文本用属性向量表示为d=（河北，河北，河北，吉林，香港），类别集合为Y={yes，no}。

类yes下总共有8个单词，类no下总共有3个单词，训练样本单词总数为11，因此P（yes）=8/11，P（no）=3/11。类条件概率计算如下。

```
P(河北 | yes)=(5+1)/(8+6)=6/14=3/7
P(河北 | yes)=P(吉林 | yes)=(0+1)/(8+6)=1/14
P(河北 | no)=(1+1)/(3+6)=2/9
P(香港 | no)=P(吉林| no)=(1+1)/(3+6)=2/9
```

分母中的8是指yes类别下textc的长度，即训练样本的单词总数，6是指训练样本有河北、北京、上海、广东、吉林、香港共6个单词，3是指no类下共有3个单词。所有项中分子中的1和分母中的6代表正则化参数。

有了以上类条件概率，计算后验概率如下。

```
P(yes | d)=(3/7)3×1/14×1/14×8/11=108/184877≈0.00058417
P(no | d)= (2/9)3×2/9×2/9×3/11=32/216513≈0.00014780
```

比较大小，即可知道这个文档属于类别“河北”。

8.4.5 算法应用实例

MLlib是一些常用机器学习算法在Spark上的实现，Spark的设计初衷是支持一些迭代的大数据算法。下面通过一个例子使用Mllib支持向量机进行分类，并将程序打包执行，读者可以通过示例开启MLlib之旅。

1.程序代码

使用支持向量机进行分类的代码如下。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.classification.SVMWithSGD
import org.apache.spark.mllib.regression.LabeledPoint
Val SL = new Sparkcontext("Local","SVM","/root",Sep "svm.jar"))
/* 加载和解析样例数据文件 */
val data = sc.textFile("mllib/data/sample_svm_data.txt")
val parsedData = data.map { line =>
  val parts = line.split(' ')
/*将数据转化为标记向量 */
  LabeledPoint(parts(0).toDouble, parts.tail.map(x => x.toDouble).toArray)
}
/* 配置迭代次数为20 */
val numIterations = 20
/* 使用SVMWithSGD类，训练模型。其中使用的优化方法是随机梯度下降(SGD) */
val model = SVMWithSGD.train(parsedData, numIterations)
/* 使用训练好的模型进行分类预测 */
val labelAndPreds = parsedData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
/*统计分类错误并打印 */
val trainErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / parsedData.count
println("trainError = " + trainErr)
```

与开发Spark应用程序一样，可以在spark-shell中交互式地运行MLlib，也可以把代码打包成一个jar提交到Spark集群中执行。

2.在Spark Shell中运行

在Spark根目录执行bin/spark-shell命令。然后将示例代码复制进Shell。

3.打包为Jar运行

1）新建项目，项目中需要创建一个SVM_TEST类，将使用支持向量机进行分类的代码

复制到main函数中，然后配置SparkContext。

2）示例项目下需要有一个sbt配置文件simple.sbt，输入下面的配置项。

```
name ：= "SVM_TEST"
version ：= "1.0"
scalaVersion ：= "2.10.3"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.0.0-incubating"
#由于本例使用MLlib底层库，所以需要加入MLlib库依赖
libraryDependencies += "org.apache.spark" %% "spark-mllib" % "1.0.0-incubating"
resolvers += "Akka Repository" at "http：//repo.akka.io/releases/"
```

3）把上述文件组织成一定的目录结构。

```
./simple.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SVM_TEST.scala
```

4）在示例项目根目录下执行以下操作。

①打包项目。

```
sbt package
```

②执行项目。

```
sbt run
```

③在控制台得到以下结果。

```
trainError = 0.40372670807453415
14/08/26 18：26：22 INFO network.ConnectionManager： Selector thread was interrupted！
[success] Total time： 4 s， completed Oct 1， 2014 8：26：22 PM
```

8.4.6　利用MLlib进行电影推荐

下面介绍MLib进行个性化的电影推荐应用。通过Berkely的这个典型案例，用户可以更加深入地理解MLlib以及学会如何构建自己的MLlib应用。[1]本例中使用MovieLenss收集的72000名用户在1万部影片上的1千万个评分数据集。这里假定这个数据集已经预加载进集群的HDFS文件夹/movielens/large下。为了快速测试代码，可以先使用文件夹/movielens/medium中的小数据集进行测试，这个数据集包含6000名用户在4000部影片上的1百万个评分数据。

1.数据集

本例使用MovieLens数据集中的两个文件："ratings.dat."和"movies.dat"。所有的评分数据按照下面的格式存储"ratings.dat"中。

```
UserID：：MovieID：：Rating：：Timestamp
```

在"movies.dat"中以下面的格式存储电影信息。

```
MovieID：：Title：：Genres
```

2.协同过滤

协同过滤是推荐系统普遍使用的方法。这些技术的本质目的是填充user-item关联矩阵中的缺失数据项。MLlib 1.0支持基于模型的协同过滤，这时通过一个隐含因子的小集合来预测缺失的数据项。MLlib通过实现交替最小二乘法（ALS）去求出这些隐含因子（latent factors）。图8-25为ALS算法的解析图。ALS算法为了最小化损失函数f，通过交替固定Users或者Movies向量，对损失函数求导，最终求出逼近Ratings矩阵的结果，使用这个结果进行电影推荐，如图8-27所示。

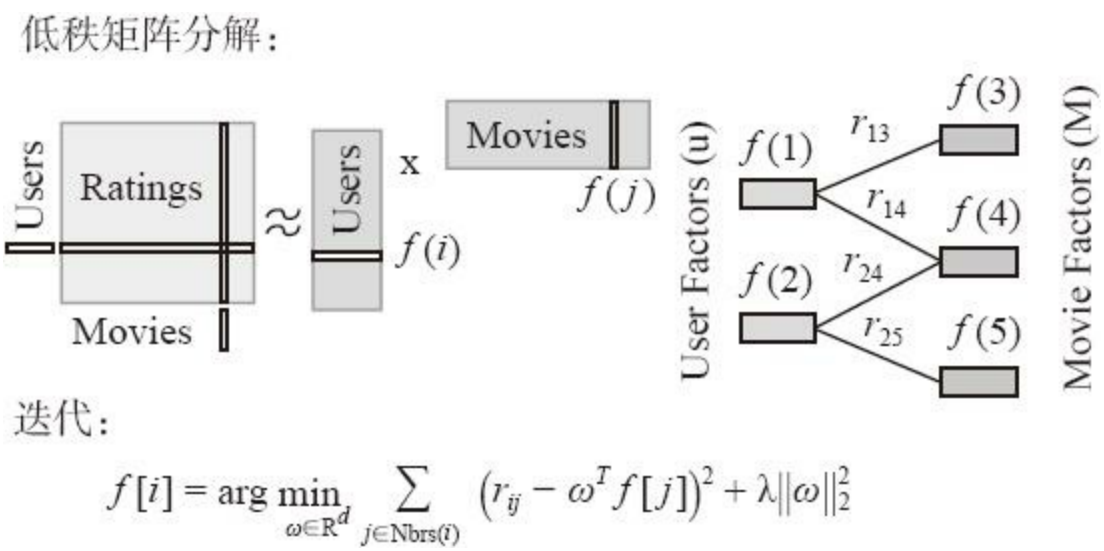

图8-27　ALS算法

3.配置

针对本例使用一个standalone项目模板。假设在用户的环境中，已经配置好所需路径和文件（实例下载地址为https://github.com/amplab/training/tree/ampcamp4/machine-learning/scala），这些已经在/root/machine-learning/scala/中设置，读者将会在目录下找到以下选项。

·sbt：包含SBT工具的目录。

·build.sbt：SBT项目文件。

·MovieLensALS.scala：用户需要编译和运行的主要Scala主程序。

·solution：包含solution代码的目录。

用户需要编辑、编译和运行的主要文件是MovieLensALS.scala，可以将下面的代码模板拷贝到文件中。

```
import java.util.Random
import org.apache.log4j.Logger
import org.apache.log4j.Level
import scala.io.Source
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
```

```
import org.apache.spark.rdd._
import org.apache.spark.mllib.recommendation.{ALS , Rating , MatrixFactorizationModel}
object MovieLensALS {
  def main ( args : Array[String] ) {
    Logger.getLogger ( "org.apache.spark" ) .setLevel ( Level.WARN )
    Logger.getLogger ( "org.eclipse.jetty.server" ) .setLevel ( Level.OFF )
    if ( args.length != 1 ) {
      println ( "Usage : sbt/sbt package \"run movieLensHomeDir\"" )
      exit ( 1 )
    }
    /* 配置环境 */
    val jarFile = "target/scala-2.10/movielens-als_2.10-0.0.jar"
    val sparkHome = "/root/spark"
    val master = Source.fromFile ( "/root/spark-ec2/cluster-url" ) .mkString.trim
    val masterHostname = Source.fromFile ( "/root/spark-ec2/masters" ) .mkString.trim
    val conf = new SparkConf ( )
      .setMaster ( master )
      .setSparkHome ( sparkHome )
      .setAppName ( "MovieLensALS" )
      .set ( "spark.executor.memory" , "8g" )
      .setJars ( Seq ( jarFile ) )
    val sc = new SparkContext ( conf )
    /* 加载评分和电影标题 */
    val movieLensHomeDir = "hdfs : //" + masterHostname + " : 9000" + args ( 0 )
    val ratings = sc.textFile ( movieLensHomeDir + "/ratings.dat" ) .map { line =>
      val fields = line.split ( " : : " )
      /* 格式为 : ( timestamp % 10 , Rating ( userId , movieId , rating ) ) */
      ( fields ( 3 ) .toLong % 10 , Rating ( fields ( 0 ) .toInt , fields ( 1 ) .toInt , fields ( 2 ) .toDouble ) )
    }
    val movies = sc.textFile ( movieLensHomeDir + "/movies.dat" ) .map { line =>
      val fields = line.split ( " : : " )
      /* 格式为 : ( movieId , movieName ) */
      ( fields ( 0 ) .toInt , fields ( 1 ) )
    }.collect.toMap
    sc.stop ( ) ;
  }
  /** 计算RMSE ( Root Mean Squared Error ) */
  def computeRmse ( model : MatrixFactorizationModel , data : RDD[Rating] , n : Long ) = {
    ……
  }
  def elicitateRatings ( movies : Seq[ ( Int , String ) ] ) = {
    ……
  }
}
```

首先通过文本编辑器打开MovieLensALS文件。

```
cd /root/machine-learning/scala
```

vim MovieLensALS.scala#如果不使用vim，还可以使用emacs或者nano进行文本编辑。

可以选用常用的文本编辑器打开文件，将示例拷贝进文件中。

对于任何的Spark计算任务来说，第一步都需要创建SparkConf对象。然后通过它创建SparkContext对象。对于Scala或者Java程序来说，需要配置Spark集群的URL、Spark主目录和用户程序需要的JAR文件进行初始化。对于Python程序来说，只需要配置Spark cluster

URL一个参数即可。最后还需要配置一个应用名称，以便在Spark Web UI中确认程序。

①可以参照下面的初始化代码。

```
val conf = new SparkConf( )
  .setMaster( master )
  .setSparkHome( sparkHome )
  .setAppName( "MovieLensALS" )
  .set( "spark.executor.memory" , "8g" )
  .setJars( Seq( jarFile ) )
val sc = new SparkContext( conf )
```

②使用SparkContext对象读取评分文件。评分文件以“：：”作为分隔符。下面的代码解析评分文件的每行创建以（Int，Rating）对为数据项的一个RDD。这里保存时间戳的最后一个数字作为一个随机关键字。Ratingclass是一个对元组（user：Int，product：Int，rating：Double）的包装类，它在MLlib的包org.apache.spark.mllib.recommendation中定义。

```
val movieLensHomeDir = "hdfs://" + masterHostname + ":9000" + args( 0 )
   val ratings = sc.textFile( movieLensHomeDir + "/ratings.dat" ).map { line =>
    val fields = line.split( "::" )
     /* 格式为： ( timestamp % 10 , Rating( userId , movieId , rating ) ) */
     ( fields( 3 ).toLong % 10 , Rating( fields( 0 ).toInt , fields( 1 ).toInt , fields( 2 ).
     toDouble ) )
  }
```

③通过读取movie id和title将其转化电影ID到title的映射。

```
val movies = sc.textFile( movieLensHomeDir + "/movies.dat" ).map { line =>
      val fields = line.split( "::" )
      /* 格式为： ( movieId , movieName ) */
      ( fields( 0 ).toInt , fields( 1 ) )
   }.collect.toMap
```

至此，用户可以统计一些评分数据。

```
val numRatings = ratings.count
   val numUsers = ratings.map( _._2.user ).distinct.count
   val numMovies = ratings.map( _._2.product ).distinct.count
   println( "Got " + numRatings + " ratings from "
     + numUsers + " users on " + numMovies + " movies." )
```

4.运行程序

用户应该知道如何运行例子。保存之前编辑的MovieLensALS文件，然后运行下面的命令。

```
cd /root/machine-learning/scala
```

如果用户需要运行大数据集，则将参数medium转换为large，进而在大数据集上运行例子：

```
sbt/sbt package "run /movielens/medium" / *其中/movielens/medium是主程序main函数的输入参数 */
```

这个命令将会编译MovieLensALS类，然后在/root/machine-learning/scala/target/scala-2.10/目录下创建一个JAR文件，最后运行这个程序，在用户的控制台上显示下面的输出。

```
Got 1000209 ratings from 6040 users on 3706 movies.
```

5.启发评级

为了向用户推荐，需要通过用户评价的一些电影了解用户的兴趣。需要统计每个电影接收到的评分，然后根据评分数排序。最后获取评分最高的50部电影，采样出一个小集合进行启发评级。

```
val mostRatedMovieIds = ratings.map( _._2.product )
/* 抽取 movie id*/
.countByValue
/* 计算每个movie的评分*/
.toSeq
/* 将数据转换为Seq格式*/
.sortBy( - _._2 )
/* 通过评分数排序*/
.take( 50 )
/* 获得评分数最多的50部movie*/
.map( _._1 )
/* 获取它们的ID*/
val random = new Random( 0 )
val selectedMovies = mostRatedMovieIds.filter( x => random.nextDouble( ) < 0.2 )
.map( x => ( x , movies( x ) ) )
.toSeq
```

每个被挑选到的电影都需要评分（评分为0~5的整数，如果没有看过，则填0）。方法eclicitateRatings返回用户的评分，用户会分配到一个特殊的用户ID 0。这个评分通过sc.parallelize.转换为一个RDD[Rating]实例。

```
val myRatings = elicitateRatings(selectedMovies)
val myRatingsRDD = sc.parallelize(myRatings)
```

运行以上程序，将会看到和下面类似的提示。

```
Please rate the following movie (1-5 (best), or 0 if not seen):
Raiders of the Lost Ark (1981):
```

6.切分训练数据

使用MLlib中的ALS算法将会用RDD[Rating]实例作为输入来训练一个模型。ALS算法有一些训练参数。例如，矩阵因子的排名和正则化器的实例。为了确定一个好的训练参数，基于时间戳的最后一位将数据分成3个没有交集的子集，称为训练集、测试集和评价集，并将它们缓存。接下来会通过训练集训练不同的模型，通过RMSE（root mean squared error）方法和评价集选取最好的集合，通过测试集评价最好的模型，并将用户的评分加入测试集中，进而对用户推荐。在这个过程中，由于需要多次访问这些数据，所以会把训练集、评价集和测试集通过persist方法放到内存。

```
val numPartitions = 20
   val training = ratings.filter(x => x._1 < 6)
                         .values
                         .union(myRatingsRDD)
                         .repartition(numPartitions)
                         .persist
    val validation = ratings.filter(x => x._1 >= 6 && x._1 < 8)
                            .values
                            .repartition(numPartitions)
                            .persist
    val test = ratings.filter(x => x._1 >= 8).values.persist
    val numTraining = training.count
    val numValidation = validation.count
    val numTest = test.count
    println("Training: " + numTraining + ", validation: " + numValidation + ", test: " + numTest)
```

在进行切分之后，用户将会看到下面的日志信息。

```
Training: 602251, validation: 198919, test: 199049.
```

7.通过ALS算法进行模型训练

下面使用ALS.train方法来训练一组模型，然后从中评价和选择出最好的模型。ALS所有训练算法中最重要的参数是rank、lambda（正则化常数）和迭代次数iterations。使用的ALS算法中的train方法以下面的方式给出。

```
object ALS {
  def train(ratings: RDD[Rating], rank: Int, iterations: Int, lambda: Double)
    : MatrixFactorizationModel = {
      ……
  }
}
```

在理想情况下，用户希望能够尝试所有的参数组合来发现最好的状况。但是由于时间的约束，本例只会测试8种组合：两种不同的rank（8和12）、两种不同的lambdas（1.0和10.0）以及两种不同的iterations（10和20）。MLlib中提供方法computeRmse在每个模型的评价集上计算RMSE。RMSE值最小的评价集将被最后选择，RMSE值作为评价指标。

```
val ranks = List(8, 12)
    val lambdas = List(0.1, 10.0)
    val numIters = List(10, 20)
    var bestModel: Option[MatrixFactorizationModel] = None
    var bestValidationRmse = Double.MaxValue
    var bestRank = 0
    var bestLambda = -1.0
    var bestNumIter = -1
    for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
      val model = ALS.train(training, rank, numIter, lambda)
      val validationRmse = computeRmse(model, validation, numValidation)
      println("RMSE (validation) = " + validationRmse + " for the model trained
      with rank = "
        + rank + ", lambda = " + lambda + ", and numIter = " + numIter + ".")
      if (validationRmse < bestValidationRmse) {
        bestModel = Some(model)
        bestValidationRmse = validationRmse
        bestRank = rank
        bestLambda = lambda
        bestNumIter = numIter
      }
    }
    val testRmse = computeRmse(bestModel.get, test, numTest)
    println("The best model was trained with rank = " + bestRank + " and lambda = " + bestLambda
      + ", and numIter = " + bestNumIter + ", and its RMSE on the test set is "
      + testRmse + ".")
```

程序运行成功后，用户将在控制台看到下面的信息。

```
The best model was trained using rank 8 and lambda 10.0, and its RMSE on test is 0.8808492431998702.
```

8.电影推荐

最后可以看到通过训练出的模型推荐给用户哪些电影。推荐是通过生成用户没有评分的电影的（0，movieId）对，然后调用predict方法获取预测。

```
class MatrixFactorizationModel {
  def predict(userProducts: RDD[(Int, Int)]): RDD[Rating] = {
      ……
  }
}
```

获取所有的预测之后，用户可以列出top 50的推荐电影。

```
val myRatedMovieIds = myRatings.map(_.product).toSet
    val candidates = sc.parallelize(movies.keys.filter(!myRatedMovieIds.contains(_)).toSeq)
    val recommendations = bestModel.get
                                   .predict(candidates.map((0, _)))
                                   .collect
                                   .sortBy(-_.rating)
                                   .take(50)
    var i = 1
    println("Movies recommended for you:")
    recommendations.foreach { r =>
      println("%2d".format(i) + ": " + movies(r.product))
      i += 1
    }
```

用户会得到类似下面的输出。

```
Movies recommended for you:
 1: Silence of the Lambs, The (1991)
 2: Saving Private Ryan (1998)
 3: Godfather, The (1972)
 4: Star Wars: Episode IV - A New Hope (1977)
 5: Braveheart (1995)
 6: Schindler's List (1993)
 7: Shawshank Redemption, The (1994)
 8: Star Wars: Episode V - The Empire Strikes Back (1980)
 9: Pulp Fiction (1994)
10: Alien (1979)
……
```

由于数据集较旧，显示的基本都是老电影。

[1] 示例参考：http://ampcamp.berkeley.edu/big-data-mini-course/movie-

recommendation-with-mllib.html

8.5 本章小结

本章主要介绍了BDAS中广泛应用的几个数据分析组件。SQL on Spark提供在Spark上的SQL查询功能，让用户可以基于内存计算和SQL进行大数据分析。通过Spark Streaming，用户可以构建实时流处理应用，高吞吐量，以及适合历史和实时数据混合分析的特性，使Spark Streaming在流数据处理框架中突出重围。GraphX充当Spark生态系统中图计算的角色，其简洁的API使图处理算法的书写更加便捷。最后介绍了MLlib，Spark上的机器学习库。它充分利用Spark内存计算和适合迭代的特性，使分布式系统与并行机器学习算法完美结合。相信随着Spark生态系统的日臻完善，这些组件还会长足发展。

最后一章将介绍Spark的性能调优，在实战中如何让Spark运行得更快，更节省资源，是系统开发者追求的目标。

第9章　Spark性能调优

本章主要介绍如何对Spark进行性能调优。当程序能够运行起来时，开发人员开始关注这个程序能否节省空间占用、能否运行得更快。这些性能需求就需要开发者通过重构代码或者调整集群的配置参数进行优化。下面介绍一些Spark可以优化的场景和技巧。

9.1 配置参数

在Spark应用程序的开发中，需要根据具体的算法和应用场景选择调优方法，没有万能的解决方案，每一种调优方法对一些方面的性能可以优化，对另一些方面的性能可能会产生损耗，下面先介绍对性能调优产生影响的主要参数。

1.如何进行参数配置性能调优

熟悉Hadoop开发的用户对配置项应该不陌生。根据不同问题，调整不同的配置项参数是比较基本的调优方案。配置项可以在脚本文件中添加，也可以在代码中添加。

（1）通过配置文件spark-env.sh添加

可以参照spark-env.sh.template模板文件中的格式进行配置，参见下面的格式。

```
export JAVA_HOME=/usr/local/jdk
export SCALA_HOME=/usr/local/scala
export SPARK_MASTER_IP=127.0.0.1
export SPARK_MASTER_WEBUI_PORT=8088
export SPARK_WORKER_WEBUI_PORT=8099
export SPARK_WORKER_CORES=4
export SPARK_WORKER_MEMORY=8g
```

（2）在程序中通过SparkConf对象添加

如果是在代码中添加，则需要在SparkContext定义之前修改配置项的修改。例如：

```
val conf = new SparkConf( ) .setMaster( "local" ) .setAppName( "My
application" ) .set( "spark.executor.memory" , "1g" )
val sc = new SparkContext( conf )
```

（3）在程序中通过System.setProperty添加

如果是在代码中添加，则需要在SparkContext定义之前修改配置项。例如：

```
System.setProperty( "spark.executor.memory" , "14g" )
System.setProperty( "spark.worker.memory" , "16g" )
val conf = new SparkConf( ).setAppName( "Simple Application" )
```

```
val sc = new SparkContext(conf)
```

Spark官网（http://spark.apache.org/docs/latest/configuration.html）推荐了很多配置参数，读者可以参阅。

2.如何观察性能问题

可以通过下面的方式监测性能。

1）Web UI。

2）Driver程序控制台日志。

3）logs文件夹下日志。

4）work文件夹下日志。

5）Profiler工具。

例如，一些JVM的Profiler工具，如Yourkit、Jconsole或者JMap、JStack等命令。更全面的可以通过集群的Profiler工具，如Ganglia、Ambaria等。

9.2 调优技巧

一个应用程序可以完成基本功能其实还不够，还有一些更加细节和有实际意义的问题需要考虑，尤其是性能优化问题，但以往的经验教训告诉我们，过早的性能优化是万恶之源，性能优化应该随着程序的开发、调试以及作业的运行观察性能瓶颈，进而进行性能调优。

性能方面的提高概括来说主要包括时间性能提升和空间性能提升，而这两个方面又是一个权衡和矛盾的地方，需要根据应用的具体需求运行环境适当调节，进而在正确完成功能的基础之上，使执行的时间尽可能的短，占用的空间尽量小。

当处理大规模数据时，调优是必须面对的问题，Spark是内存计算，内存问题就变得尤为重要。下面介绍的调优方法并不能涵盖Spark的全部，更多细节的调优可以到Spark的社区进行提问和查看，上面会有很多和你遇到同样问题的解决方案，以及很多的高手乐于帮你解答。

下面从以下几个方面来介绍Spark的性能调优。

9.2.1 调度与分区优化

下面从几个方面讲解调度与分区优化问题。

1.小分区合并问题

在用户使用Spark的过程中，常常会使用filter算子进行数据过滤。而频繁的过滤或者过滤掉的数据量过大就会产生问题，造成大量小分区的产生（每个分区数据量小）。由于Spark是每个数据分区都会分配一个任务执行，如果任务过多，则每个任务处理的数据量很小，会造成线程切换开销大，很多任务等待执行，并行度不高的问题，是很不经济的。例如：

```
val rdd2 = rdd1.filter(line=>lines.contains("error"
  )).filter(line=>line.contains(info).collect();
```

解决方式：可以采用RDD中重分区的函数进行数据紧缩，减少分区数，将小分区合并变为大分区。

通过coalesce函数来减少分区，具体如下。

```
def coalesce(numPartitions: Int, shuffle: Boolean = false)(implicit
  ord: Ordering[T] = null): RDD[T]
```

这个函数会返回一个含有numPartitions数量个分区的新RDD，即将整个RDD重分区。

以下几个情景请大家注意，当分区由10000重分区到100时，由于前后两个阶段的分区是窄依赖的，所以不会产生Shuffle的操作。

但是如果分区数量急剧减少，如极端状况从10000重分区为一个分区时，就会造成一个问题：数据会分布到一个节点上进行计算，完全无法开掘集群并行计算的能力。为了规避这个问题，可以设置shuffle=true。请看源码：

```
new CoalescedRDD(new ShuffledRDD[Int, T, T, (Int,T)]
      (mapPartitionsWithIndex(distributePartition),
    new HashPartitioner(numPartitions)),
      numPartitions).values
```

由于Shuffle可以分隔Stage，这就保证了上一阶段Stage中的上游任务仍是10000个分区在并行计算。如果不加Shuffle，则两个上下游的任务合并为一个Stage计算，这个Stage便会在1个分区状况下进行并行计算。

同时还会遇到另一个需求，即当前的每个分区数据量过大，需要将分区数量增加，以利用并行计算能力，这就需要把Shuffle设置为true，然后执行coalesce函数，将分区数增大，在这个过程中，默认使用Hash分区器将数据进行重分区。

```
def repartition(numPartitions: Int)(implicit ord: Ordering[T]
    = null): RDD[T]
```

rePartition方法会返回一个含有numPartitions个分区的新RDD。

repartition的源码如下。

```
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null):
   RDD[T] = {
     coalesce(numPartitions, shuffle = true)
  }
```

reparition本质上就是调用的coalesce方法。因此如果用户不想进行Shuffle，就需用coalese配置重分区，为了方便起见，可以直接用repartition进行重分区。

2.倾斜问题

倾斜（skew）问题是分布式大数据计算中的重要问题，很多优化研究工作都围绕该问题展开。倾斜有数据倾斜和任务倾斜两种情况，数据倾斜导致的结果即为任务倾斜，在个别分区上，任务执行时间过长。当少量任务处理的数据量和其他任务差异过大时，任务进度长时间维持在99%（或100%），此时，任务监控页面中有少量（1个或几个）reduce子任务

未完成。单一reduce的记录数与平均记录数差异过大，最长时长远大于平均时长，常可能达到3倍甚至更多。

（1）数据倾斜

产生数据倾斜的原因大致有以下几种。

1）key的数据分布不均匀（一般是分区key取得不好或者分区函数设计得不好）。

2）业务数据本身就会产生数据倾斜（像TPC-DS为了模拟真实环境负载特意用有倾斜的数据进行测试）。

3）结构化数据表设计问题。

4）某些SQL语句会产生数据倾斜。

（2）任务倾斜

产生任务倾斜的原因较为隐蔽，一般就是那台机器的正在执行的Executor执行时间过长，因为服务器架构，或JVM，也可能是来自线程池的问题，等等。

解决方式：可以通过考虑在其他并行处理方式中间加入聚集运算，以减少倾斜数据量。

数据倾斜一般可以通过在业务上将极度不均匀的数据剔除解决。这里其实还有Skew Join的一种处理方式，将数据分两个阶段处理，倾斜的key数据作为数据源处理，剩下的key的数据再做同样的处理。二者分开做同样的处理。

（3）任务执行速度倾斜

产生原因可能是数据倾斜，也可能是执行任务的机器在架构，OS、JVM各节点配置不同或其他原因。

解决方式：设置spark.speculation=true把那些执行时间过长的节点去掉，重新调度分

配任务，这个方式和Hadoop MapReduce的speculation是相通的。同时可以配置多长时间来推测执行，spark.speculation.interval用来设置执行间隔进行配置。在源码中默认是配置的100，示例如下。

```
val SPECULATION_INTERVAL = conf.getLong("spark.speculation.interval", 100)
```

（4）解决方案

1）增大任务数，减少每个分区数据量：增大任务数，也就是扩大分区量，同时减少单个分区的数据量。

2）对特殊key处理：空值映射为特定Key，然后分发到不同节点，对空值不做处理。

3）广播。

①小数据量表直接广播。

②数据量较大的表可以考虑切分为多个小表，多阶段进行Map Side Join。

4）聚集操作可以Map端聚集部分结果，然后Reduce端合并，减少Reduce端压力。

5）拆分RDD：将倾斜数据与原数据分离，分两个Job进行计算。

3.并行度

在分布式计算的环境下，如果不能正确配置并行度，就不能够充分利用集群的并行计算能力，浪费计算资源。Spark会根据文件的大小，默认配置Map阶段任务数量，也就是分区数量（也可以通过SparkContext.textFile等方法进行配置）。而Reduce的阶段任务数量配置可以有两种方式，下面分别进行介绍。

第一种方式：写函数的过程中通过函数的第二个参数进行配置。

```
def reduceByKey(func: (V, V) => V, numPartitions: Int):
    RDD[(K, V)]          = {
     reduceByKey(new HashPartitioner(numPartitions), func)
    }
```

第二种方式：通过配置spark.default.parallelism来进行配置。它们的本质原理一致，均是控制Shuffle过程的默认任务数量。

下面介绍通过配置spark.default.parallelism来配置默认任务数量（如groupByKey、reduceByKey等操作符需要用到此参数配置任务数），这里的数量选择也是权衡的过程，需要在具体生产环境中调整，Spark官方推荐选择每个CPU Core分配2~3个任务，即cpu core num*2（或3）数量的并行度。

如果并行度太高，任务数太多，就会产生大量的任务启动和切换开销。

如果并行度太低，任务数太小，就会无法发挥集群的并行计算能力，任务执行过慢，同时可能会造成内存combine数据过多占用内存，而出现内存溢出（out of memory）的异常。

下面通过源码介绍这个参数是怎样发挥作用的。可以通过分区器的代码看到，分区器函数式决定分区数量和分区方式，因为Spark的任务数量由分区个数决定，一个分区对应一个任务。

```
object Partitioner {
  def defaultPartitioner(rdd: RDD[_], others: RDD[_]*): Partitioner = {
    val bySize = (Seq(rdd) ++ others).sortBy(_.partitions.size).reverse
    for (r <- bySize if r.partitioner.isDefined) {
      return r.partitioner.get
    }
    if (rdd.context.conf.contains("spark.default.parallelism"))
{
      new HashPartitioner(rdd.context.defaultParallelism)
    } else {
      new HashPartitioner(bySize.head.partitions.size)
    }
  }
}
```

从RDD的代码中可以看到，默认的分区函数设置了groupBy的分区数量。

```
def groupBy[K](f: T => K)(implicit kt: ClassTag[K]):
    RDD[(K, Iterable[T])]    =
      groupBy[K](f, defaultPartitioner(this))
```

reduceByKey中也是通过默认分区器设置分区数量。

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)] = {
    reduceByKey(defaultPartitioner(self), func)
  }
```

4.DAG调度执行优化

1）同一个Stage中尽量容纳更多的算子，以减少Shuffle的发生。

由于Stage中的算子是按照流水线方式执行的，所以更多的Transformation放在一起执行能够减少Shuffle的开销和任务启动和切换的开销。

2）复用已经cache过的数据。可以使用cache和persist函数将数据缓存在内存，其实用户可以按单机的方式理解，存储仍然是多级存储，数据存储在访问快的存储设备中，提高快速存储命中率会提升整个应用程序的性能。

9.2.2 内存存储优化

下面将从以下几个方面讲解内存存储的优化。[1]

1.JVM调优

内存调优过程的大方向上有三个方向是值得考虑的。

1）应用程序中对象所占用的内存空间。

2）访问这些内存对象的代价。

3）垃圾回收的开销。

通常状况下，Java的对象访问速度是很快的，但是相对于对象中存储的原始数据，Java对象整体会耗费2~5倍的内存空间。

（1）内存耗损原因

内存耗损是由以下几个原因造成的，熟悉JVM的用户可能会比较熟悉其中的原因。

1）不同的Java对象都会有一个对象头（object header），这个对象头大约为16byte，包含指向这个对象的类的指针等信息，对一些只有少量数据的对象，这是极为不经济的。例如，只有一个Int属性的对象，这个头的信息所占空间会大于对象的数据空间。

2）Java中的字符串（String）占用40byte空间。String的内存是将真正字符串的信息存储在一个char数组中，并且还会存储其他的信息，如字符串长度，同时如果采用UTF-16编码，一个字符就占用2byte的空间。综合以上，一个10字符的字符串会占用超过60byte的内存空间。

3）常用的一些集合类，如LinkedList等是采用链式数据结构存储的，对底层的每个数据

项进行了包装，这个对象不只存储数据，还会存储指向其他数据项的指针，这些指针也会产生数据空间的占用和开销。

4）集合类中的基本数据类型常常采用一些装箱的对象存储，如java.lang.Ingeger。装箱与拆箱的机制在很多程序设计语言中都有，Java中装箱意味着将这些基本数据类型包装为对象存储在内存的Java堆中，而拆箱意味着将堆中对象转换为栈中存储的数据。

（2）计算内存的消耗

计算数据在集群内存占用的空间的大小的最好方法是创建一个RDD，读取这些数据，将数据加载到cache，在驱动程序的控制台查看SparkContext的日志。这些日志信息会显示每个分区占用多少空间（当内存空间不够时，将数据写到磁盘上），然后用户可以根据分区的总数大致估计出整个RDD占用的空间。例如，下面的日志信息。

```
INFO BlockManagerMasterActor： Added rdd_0_1 in memory on mbk.local：50311 （size： 717.5 KB， free： 332.3 MB）
```

这表示RDD0的partition1消耗了717.5KB内存空间。

（3）调整数据结构

减少内存消耗的第一步就是减少一些除原始数据以外的Java特有信息的消耗，如链式结构中的指针消耗、包装数据产生的元数据消耗等。

1）在设计和选用数据结构时能用数组类型和基本数据类型最好，尽量减少一些链式的Java集合或者Scala集合类型的使用。可以采用fastutil这个第三方库，其中有很多对基本数据类型的集合，能够基本覆盖大部分的Java标准库集合和数据类型。官网地址为http://fastutil.di.unimi.it/。

2）减少对象嵌套。例如，使用大量数据量小、个数多的对象和内含指针的集合数据结构，这样会产生大量的指针和对象头元数据的开销。《编程之美》中提出的"程序简单就是

美”的思想在这里也能够体现，不是数据结构设计多复杂，这个程序就多好，而是能解决问题，采用的数据结构又很简单，代码量小，开销小，这样才是最见功力的。

3）考虑使用数字的ID或者枚举对象，而不是使用字符串作为key键的数据类型。从前面也看到，字符串的元数据和本身的字符编码问题产生的空间占用过大。

4）当内存小于32GB时，官方推荐配置JVM参数-XX：+UseCompressedOops，进而将指针由8byte压缩为4byte。OOP的全称是ordinary object pointer，即普通对象指针。在64位HotSpot中，OOP使用32位指针，默认64位指针会比32位指针使用的内存多1.5倍，启用CompressOops后，会压缩的对象如下。

①每个Class的属性指针（静态成员变量）。

②每个对象的属性指针。

③普通对象数组每个元素的指针。

但是，指向PermGen的Class对象指针、本地变量、堆栈元素、入参、返回值、NULL指针不会被压缩。可以通过配置文件spark-env.sh配置这个参数，从而在Spark中启用JVM指针压缩。

（4）序列化存储RDD

如果通过上面的优化方式进行优化，对象存储空间仍然很大，一个更加简便的减少内存消耗的方法是以序列化的格式来存储这些对象。在程序中可以通过设置StorageLevels这个枚举类型来配置RDD的数据存储方式，官网的API文档中提供了更为丰富的RDD存储方式，有兴趣的读者可以自行学习参考。例如，当配置RDD为MEMORY_ONLY_SER存储方式时，Spark将这个RDD的每个分区存储为一个大的byte数组。当然这也是一个权衡的过程，这样的存储会带来数据访问变慢的问题，这是由于每次访问数据还需要经过反序列化的过程。用户如果希望在内存中缓存数据，则官方推荐使用Kyro的序列化库进行序列化，因为

Kyro相比于Java的标准序列化库序列化后的对象占用空间更小，性能更好。

（5）JVM垃圾回收（GC）调优

当Spark程序产生大数据量的RDD时，JVM的垃圾回收就会成为一个问题。当JVM需要替换和回收旧对象所占空间来为新对象提供存储空间时，根据JVM垃圾回收算法，JVM将遍历所有Java对象，然后找到不再使用的对象进而回收。这里其实开销最大的因素是程序中使用了大量的对象，所以设计数据结构时应该尽量使用创建更少的对象的数据结构，如尽量采用数组Array，而少用链表的LinkedList，从而减少垃圾回收开销。更好的一个方式是将数据缓存为序列化的形式，这些将在序列化的优化方法中详细介绍，这样只有一个对象，即一个byte数组作为一个RDD的分区存储。当遇到GC（垃圾回收）问题时，首先考虑用序列化的方式尝试解决。

当Spark任务的工作内存空间和RDD的缓存数据空间产生干扰时，垃圾回收同样会成为一个问题，可以通过控制分给RDD的缓存来缓解这个问题。

1）度量GC的影响。

GC调优的第一步是统计GC的频率和GC的时间开销。可以设置spark-env.sh中的SPARK_JAVA_OPTS参数，添加选项-verbose：gc-XX：+PrintGCDetails-XX：+PrintGCTime-Stamps。当用户下一次的Spark任务运行时，将会看到worker的日志信息中出现打印GC的时间等信息，需要注意的是，这些信息都Worker节点显示，而不在驱动程序的控制台显示。

2）缓存大小调优。

对GC来说，一个重要的配置参数就是内存给RDD用于缓存的空间大小。默认情况下，Spark用配置好的Executor 60%的内存（spark.executor.memory）缓存RDD。这就意味着40%的剩余内存空间可以让Task在执行过程中缓存新创建的对象。在有些情况下，用户的任务变慢，而且JVM频繁地进行垃圾回收或者出现内存溢出（out of memory异常），

这时可以调整这个百分比参数为50%。这个百分比参数可以通过配置spark-env.sh中的变量spark.storage.memoryFraction=0.5进行配置。同时结合序列化的缓存存储对象减少内存空间占用，将会更加有效地缓解垃圾回收问题。下面介绍一些高级GC调优技术。图9-1为Java堆中的各代内存分布。

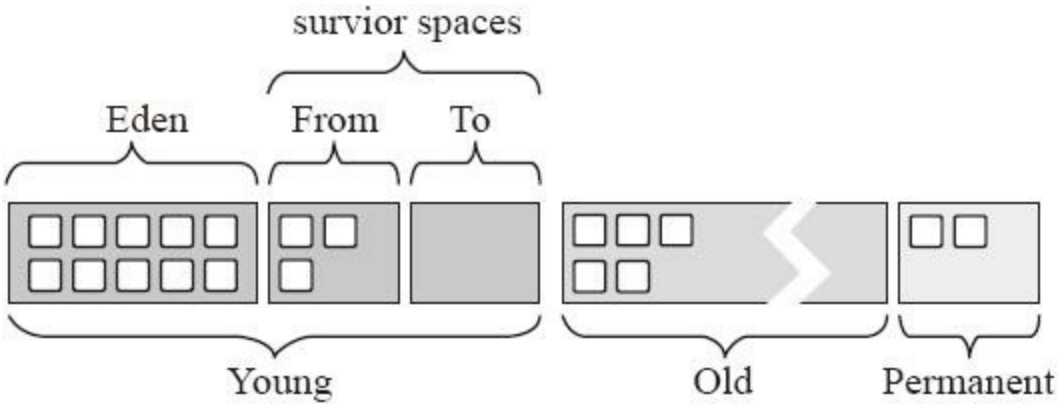

图9-1　JVM内存分布

①Young（年轻代）。

年轻代分为3个区：一个Eden区和两个Survivor区（Survivor Space）。大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当一个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当这个Survivor区也满时，从第一个Survivor区复制过来的且还存活的对象，将被复制Tenured（老年）区。需要注意的是，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden区复制过来的对象和从前一个Survivor区复制过来的对象，而复制到年老区的只有从第一个Survivor区过来的对象。而且，Survivor区总有一个是空的。大多数情况下Java程序新建的对象都是从新生代分配内存。

不同的GC方式会以不同的方式按此值来划分Eden区和Survivor区的大小，有的GC还会根据运行情况动态调整这3个区的大小。

②Tenured（年老代）。

年老代存放从年轻代存活的对象。一般来说，年老代存放的都是生命期较长的对象。

③Perm（持久代）。

持久代用于存放静态文件、Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，这时需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小可通过-XX：MaxPermSize=设置。

持久代对应内存模型中的方法区，存放了加载类的信息（名称、修饰符等）、类中的静态变量、类中定义为final类型的常量、类中的field信息、类中的方法信息，开发人员通过反射机制访问该区域。在sun jdk中，该区默认的最小值为16MB，最大值为64MB，可以通过-XX：PermSize和-XX：MaxPermSize来指定最小值和最大值。

3）全局GC调优。

Spark中全局的GC调优要确保只有存活时间长的RDD存储在老年代（Old generation）区域，这样保证年轻代（Young）有足够的空间存储存活时间短的对象。这有助于减少Spark任务执行时需要给数据分配的空间，用户可以通过下面的方法观察和解决full GC问题。

可以通过观察日志信息查看是否存在过多过频繁的GC。如果full GC在任务执行完成之前被触发多次，就表示对正在执行的任务没有足够的内存空间分配。

如果从打印的GC日志来看，老年代将要满了，就应该减少缓存数据的内存使用量，可以通过配置spark.storage.memoryFraction属性进行配置，缓存更少的对象还是比减慢内存执行时间更加经济。下面具体讲解spark.storage.memoryFraction属性。

spark.storage.memoryFraction控制用于Spark缓存的Java堆空间，默认值为0.67，即2/3的Java堆空间用于Spark的缓存。如果任务的计算过程中需要用到较多的内存，而RDD所需内存较少，就可以调低这个值，以减少计算过程中因为内存不足而产生的GC过程。在调

优过程中发现，GC过多是导致任务运行时间较长的一个常见原因。如果任务运行较慢，想确定是否是GC太多导致的，可以在spark-env.sh中设置JAVA_OPTS参数，以打印GC的相关信息，设置如下。

```
JAVA_OPTS=" -verbose：gc -XX：+PrintGCDetails -XX：+PrintGCTimeStamps"
```

这样如果有GC发生，就可以在master和work的日志上看到。

下面通过源码看看这个参数是怎样发挥作用的。

在BlockManager中对memoryFraction

```
private def getMaxMemory(conf： SparkConf)： Long = {
   val memoryFraction = conf.getDouble("spark.storage.memoryFraction", 0.6)
   (Runtime.getRuntime.maxMemory * memoryFraction).toLong
  }
```

如果看到GC日志中有很多minor GC信息，而非major GC信息，分配更多的内存给Eden区将会很有帮助。可以设定估计出的每个任务执行需要的内存为Eden区内存大小。如果已经设置Eden区内存大小为E，就可以通过JVM配置参数-Xmn=4/3*E设置年轻代大小，多出的空间分配给Survivor区域使用。

例如，如果任务是从HDFS读取数据，内存空间的占用可以通过从HDFS读取的数据块大小和数量估计。需要注意的是，一般情况下，压缩的数据压缩之后通常为原来数据块大小的2~3倍。因此如果一个JVM中要执行3~4个任务，同时HDFS的数据块大小是64MB，就可以估计需要的Eden代大小是4×3×64MB大小的空间。

最后监控修改了配置参数之后，Spark应用的GC频率和时间开销，进一步调优。

官方给出的GC调优建议是，GC调优依赖于两个关键因素：应用程序和集群能够提供的可用内存大小。在Oracle的官网（http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html）上有更多高级的GC调优方法。不论怎样减少GC的频度，都可以明显

减少开销。

2.OOM问题优化

相信有一定Spark或者Hadoop开发经验的用户或多或少都遇到过OutOfMemoryError内存溢出问题。

通过之前的介绍，读者已经对JVM的内存管理有了大致的了解，JVM管理大致分为这几个区域：permanent generation space（持久代区域）、heap space（堆区域）、Java stacks（Java栈）。

持久代区域主要存放类和元数据信息，Class第一次加载时被放入PermGen space区域，Class需要存储的内容主要包括方法和静态属性。

堆区域用来存放对象，对象需要存储的内容主要是非静态属性，包括年轻代和年老代。每次用new创建一个对象实例后，对象实例存储在堆区域中，这部分空间也由JVM的垃圾回收机制管理。

Java栈与大多数编程语言，包括汇编语言的栈功能相似，主要存储基本类型变量以及方法的输入输出参数。Java程序的每个线程中都有一个独立的堆栈，然后值类型会存储在栈上。

容易发生内存溢出问题的内存空间包括permanent generation space（持久代空间）和heap space（堆空间），笔者常遇到的情景就是heap space的问题。

发生内存溢出问题的原因是Java虚拟机创建的对象太多，在进行垃圾回收时，虚拟机分配到的堆内存空间已经用满了，与heap space有关。解决这类问题有两种思路，一种是减少App的内存占用消耗，另一种是增大内存资源的供给，具体的做法如下。

1）检查程序，看是否有死循环或不必要重复创建大量对象的地方。找到原因后，修改程序和算法。有很多Java profile工具可以使用，官方推荐的是YourKit其他还有JvisualVM、

Jcohsole等工具可以使用。

2）按照之前内存调优中总结的能够减少对象在内存数据存储空间的方法开发程序开发和配置参数。

3）增加Java虚拟机中Xms（初始堆大小）和Xmx（最大堆大小）参数的大小，如set JAVA_OPTS=-Xms256m-Xmx1024m。

引起这个问题的原因还很可能是Shuffle类操作符在任务执行过程中在内存建立的Hash表过大。在这种情况下，可以通过增加任务数，即分区数来提升并行性度，减小每个任务的输入数据，减少内存占用来解决。

3.磁盘临时目录空间优化

配置参数spark.local.dir能够配置Spark在磁盘的临时目录，默认是/tmp目录。在Spark进行Shuffle的过程中，中间结果会写入Spark在磁盘的临时目录中，或者当内存不能够完全存储RDD时，内存放不下的数据会写到配置的磁盘临时目录中。

这个临时目录设置过小会造成No space left on device异常。也可以配置多个盘块spark.local.dir=/mn1/spark，/mnt2/spar，/mnt3/spark来扩展Spark的磁盘临时目录，让更多的数据可以写到磁盘，加快I/O速度。

[1] 参考自http://spark.apache.org/docs/latest/tuning.html#memory-tuning ，Spark官网。

9.2.3 网络传输优化

1.大任务分发优化

在任务的分发过程中会序列化任务的元数据信息，以及任务需要的jar和文件。任务的分发是通过AKKA库中的Actor模型之间的消息传送的。因为Spark采用了Scala的函数式风格，传递函数的变量引用采用闭包方式传递，所以当需要传输的数据通过Task进行分发时，会拖慢整体的执行速度。配置参数spark.akka.frameSize（默认buffer的大小为10MB）可以缓解过大的任务造成AKKA缓冲区溢出的问题，但是这个方式并不能解决本质的问题。下面具体讲解配置参数spark.akka.frameSize。

spark.akka.frameSize控制Spark框架内使用的AKKA框架中，Actor通信消息的最大容量（如任务（Task）的输出结果），因为整个Spark集群的消息传递都是通过Actor进行的，默认为10MB。当处理大规模数据时，任务的输出可能会大于这个值，需要根据实际数据设置一个更高的值。如果是这个值不够大而产生的错误，则可以从Worker节点的日志中排查。通常Worker上的任务失败后，主节点Master的运行日志上提示“Lost TID：”，可通过查看失败的Worker日志文件$SPARK_HOME/work/目录下面的日志文件中记录的任务的Serialized size of result是否超过10MB来确定通信数据超过AKKA的Buffer异常。

2.Broadcast在调优场景的使用

Spark的Broadcast（广播）变量对数据传输进行优化，通过Broadcast变量将用到的大数据量数据进行广播发送，可以提升整体速度。Broadcast主要用于共享Spark在计算过程中各个task都会用到的只读变量，Broadcast变量只会在每台计算机器上保存一份，而不会每个task都传递一份，这样就大大节省了空间，节省空间的同时意味着传输时间的减少，效率也高。在Spark的HadoopRDD实现中，就采用Broadcast进行Hadoop JobConf的传输。官方文档的说法是，当task大于20KB时，可以考虑使用Broadcast进行优化，还可以在控制台日志看到任务是多大，进而决定是否优化。还需要注意，每次迭代所传输的Broadcast变量都

会保存在从节点Worker的内存中，直至内存不够用，Spark才会把旧的Broadcast变量释放掉，不能提前进行释放。BroadCast变量有一些应用场景，如MapSideJoin中的小表进行广播、机器学习中需要共享的矩阵的广播等。

用户可以调用SparkContext中的方法生成广播变量。

```
def broadcast[T](value: T)(implicit arg0: ClassTag[T]): Broadcast[T]
```

3.Collect结果过大优化

在开发程序的过程中，会常常用到Collect操作符。Collect函数的实现如下。

```
def collect(): Array[T] = {
   val results = sc.runJob(this, (iter: Iterator[T]) => iter.toArray)
    Array.concat(results: _*)
  }
```

函数通过SparkContext将每个分区执行变为数组，返回主节点后，将所有分区的数组合并成一个数组。这时，如果进行Collect的数据过大，就会产生问题，大量的从节点将数据写回同一个节点，拖慢整体运行时间，或者可能造成内存溢出的问题。

解决方式：当收集的最终结果数据过大时，可以将数据存储在分布式的HDFS或其他分布式持久化层上。将数据分布式地存储，可以减小单机数据的I/O开销和单机内存存储压力。或者当数据不太大，但会超出AKKA传输的Buffer大小时，需要增加AKKA Actor的buffer，可以通过配置参数spark.akka.frameSize（默认大小为10MB）进行调整。

9.2.4 序列化与压缩

前面章节详细介绍了Spark的I/O机制，下面介绍I/O中的主要调优方向。

1.通过序列化优化

序列化的本质作用是将链式存储的对象数据，转化为连续空间的字节数组存储的数据。这样的存储方式就会产生以下几个好处。

1）对象可以以数据流方式进行进程间传输（包含网络传输），同样可以以连续空间方式存储到文件或者其他持久化层中。

2）连续空间的存储意味着可以进行压缩。这样减少数据存储空间和传输时间。

3）减少了对象本身的元数据信息和基本数据类型的元数据信息的开销。

4）对象数减少也会减少GC的开销和压力。

综上所述，数据进行序列化还是很有价值的。

Spark中提供了两个序列化库和两种序列化方式：使用Java标准序列化库进行序列化的方式和使用Kyro库进行序列化的方式。Java标准序列化库兼容性好，但体积大、速度慢，Kyro库兼容性略差，但是体积小、速度快。所以在能使用Kyro的情况下，还是推荐使用Kyro进行序列化。

可以通过spark.serializer="org.apache.spark.serializer.KryoSerializer"来配置是否使用Kyro进行序列化，这个配置参数决定了Shuffle进行网络传输和当内存无法容纳RDD将分区写入磁盘时，使用的序列化器的类型。

在分布式应用中，序列化处于举足轻重的地位。那些需要大量时间进行序列化的数据格式和占据过大空间的对象会拖慢整个应用。通常情况下，序列化是Spark调优的第一步。

Spark为了权衡兼容性和性能提供了两种序列化库。

（1）Java标准序列化库

在默认情况下，Spark使用ObjectOutputStream框架进行对象的序列化。可以通过实现java.io.Serializable接口，使对象可以被序列化，也可以扩展java.io.Externalizable进而控制序列化的性能。Java标准序列化库很灵活，并且兼容性好，但是通常情况下，速度较慢，而且导致序列化后数据量较大。

（2）Kyro序列化库

Spark也可以使用Kyro序列化库来更加快速地序列化对象。Kyro相对于Java序列化库能够更加快速和紧凑地进行序列化（通常有10倍的性能优势），但是Kyro并不能支持所有可序列化的类型，如果对程序有较高的性能优化要求，就需要自定义注册类。官方推荐对于网络传输密集型（network-intensive）计算，采用Kyro序列化性能更好。

Spark自动引入了对许多常用的Scala核心类的Kyro的序列化支持，这些类均是在Spark使用的Twitter chill库支持的类。

（3）序列化示例

下面通过一个例子演示如何自定义一个Kryo的可序列化的类。

创建一个公共类，这个类要扩展org.apache.spark.serializer.KryoRegistrator，然后配置spark.kryo.registrator指向它，代码如下。

```
import com.esotericsoftware.kryo.Kryo
import org.apache.spark.serializer.KryoRegistrator
class MyRegistrator extends KryoRegistrator {
  override def registerClasses( kryo： Kryo ) {
    kryo.register( classOf[MyClass1] )
    kryo.register( classOf[MyClass2] )
  }
}
val conf = new SparkConf( ).setMaster( ... ).setAppName( ... )
conf.set( "spark.serializer", "org.apache.spark.serializer.KryoSerializer" )
conf.set( "spark.kryo.registrator", "mypackage.MyRegistrator" )
```

```
val sc = new SparkContext(conf)
```

Kyro序列化库的官方文档上描述了更加高级的一些注册方式，如增加自定义序列化代码，用户在使用时可以参考相应文档中的样例，文档网址是https://code.google.com/p/kryo/。

如果对象占用空间很大，需要增加Kryo的缓冲区容量，就需要增加配置项spark.kryoserializer.buffer.mb的数值，默认是2MB，但参数值应该足够大，以便容纳最大的序列化后对象的传输。

如果用户不注册自定义的类，Kyro仍可以运行，但是它会针对每个对象存储一次整个类名，这样会造成很大的空间浪费。

2.通过压缩方式优化

在Spark中对RDD或者Broadcast数据进行压缩，是提高数据吞吐量和性能的一种手段。压缩数据，可以大量减少磁盘的存储空间，同时压缩后的文件在磁盘间传输和I/O以及网络传输的通信开销也会减小；当然压缩和解压缩也会带来额外的CPU开销，但可以节省更多的I/O和使用更少的内存开销。

在Spark应用中，有很大一部分作业是I/O密集型的。数据压缩对I/O密集型的作业带来性能的大大提升，但是如果用户的jobs作业是CPU密集型的，那么再压缩就会降低性能，这就要判断作业的类型，权衡是否要压缩数据。

压缩数据，可以最大限度地减少文件所需的磁盘空间和网络I/O的开销，但压缩和解压缩数据总会增加CPU的开销，故最好对那些I/O密集型的作业使用数据压缩——这样的作业会有富余的CPU资源，或者对那些磁盘空间不富裕的系统。

Spark目前支持LZF和Snappy两种解压缩方式。Snappy提供了更高的压缩速度，LZF提供了更高的压缩比，用户可以根据具体的需求选择压缩方式。具体的介绍可以参见第2章。可

以通过表9-1的配置参数配置压缩。

表9-1　压缩配置

参数	参数值	说明
spark.broadcast.compress	true	设置这次参数决定 broadcast 变量是否进行压缩。通常情况下压缩它是一个好的选择
spark.rdd.compress	false	设置此参数决定是否压缩一个已经序列化的 RDD，可以在创建 RDD 时，通过 StorageLevel.MEMORY_ONLY_SER 设定是否序列化。这样虽然耗费一些压缩时间，但是可以节省大量的内存空间
spark.io.compression.codec	org.apache.spark.io.LZFCompressionCodec	通过这个参数决定是采用 LZF，还是 Snappy 压缩算法。LZF 压缩率较高，Snappy 的压缩时间较短，用户可以根据需求具体权衡
spark.io.compression.snappy.block.size	32768	通过这个参数设置 Snappy 压缩算法的块大小

9.2.5 其他优化方法

除了之前介绍的性能调优方法，还有一些其他方法可供使用。

1.批处理

有些程序可能会调用外部资源，如数据库连接等，这些连接通过JDBC或者ODBC与外部数据源进行交互。用户可能会在编写程序时忽略掉一个问题。例如，将所有数据写入数据库，如果是一条一条地写：

```
rdd.map{line=>con=getConnection;
con.write(line.toString);
con.close}
```

因为整个RDD的数据项很大，整个集群会在短时间内产生高并发写入数据库的操作，对数据库压力很大，将产生很大的写入开销。

这里，可以将单条记录写转化为数据库的批量写，每个分区的数据写一次，这样可以利用数据库的批量写优化减少开销和减轻数据库压力。

```
rdd.mapPartitions(lines => conn.getDBConn;
for(item <- lines)
write(item.toString);
conn.close)
```

同理，对于其他类型的需要和外部资源进行交互的操作，也是应该采用这种处理方式。

2.reduce和reduceByKey的优化

reduce是Action操作，reduceByKey是Transformation操作。

reduce的源码如下。

```
def reduce(f: (T, T) => T): T = {
   val cleanF = sc.clean(f)
```

```
    val reducePartition: Iterator[T] => Option[T] = iter => {
      if (iter.hasNext) {
        Some(iter.reduceLeft(cleanF))
      } else {
        None
      }
    }
    var jobResult: Option[T] = None
    val mergeResult = (index: Int, taskResult: Option[T]) => {
      if (taskResult.isDefined) {
        jobResult = jobResult match {
          case Some(value) => Some(f(value, taskResult.get))
          case None => taskResult
        }
      }
    }
    sc.runJob(this, reducePartition, mergeResult)
    jobResult.getOrElse(throw new UnsupportedOperationException("empty collection"))
  }
```

在reduce函数中会触发sc.runJob，提交任务，reduce是一个Action操作符。

reduceByKey的源码如下。

```
  def reduceByKey(partitioner: Partitioner, func: (V, V) => V):
RDD[(K, V)] = {
    combineByKey[V]((v: V) => v, func, func, partitioner)
  }
```

由代码可知，reduceByeKey并没有触发runJob，而是调用了combineByKey，该函数调用聚集器聚集数据。

reduce是一种聚合函数，可以把各个任务的执行结果汇集到一个节点，还可以指定自定义的函数传入reduce执行。Spark也对reduce的实现进行了优化，可以把同一个任务内的结果先在本地Worker节点执行聚合函数，再把结果传给Driver执行聚合。但最终数据还是要汇总到主节点，而且reduce会把接收到的数据保存到内存中，直到所有任务都完成为止。因此，当任务很多，任务的结果数据又比较大时Driver容易造成性能瓶颈，这样就应该考虑尽量避免reduce的使用，而将数据转化为Key-Value对，并使用reduceByKey实现逻辑，使计算变为分布式计算。

reduceByKey也是聚合操作，是根据key聚合对应的value。同样的，在每一个mapper把数据发送给reducer前，会在Map端本地先合并（类似于MapReduce中的Combiner）。与reduce不同的是，reduceByKey不是把数据汇集到Driver节点，是分布式进行的，因此不会

存在reduce那样的性能瓶颈。

3.Shuffle操作符的内存使用

在有些情况下，应用将会遇到OutOfMemory的错误，其中并不是因为内存大小不能够容纳RDD，而是因为执行任务中使用的数据集合太大（如groupByKey）。Spark的Shuffle操作符（sortByKey、groupByKey、reduceByKey、join等都可以算是Shuffle操作符，因为这些操作会引发Shuffle）在执行分组操作的过程中，会在每个任务执行过程中，在内存创建Hash表来对数据进行分组，而这个Hash表在很多情况下通常变得很大。最简单的一种解决方案就是增加并行度，即增加任务数量和分区数量。这样每轮次每个Executor执行的任务数是固定的，每个任务接收的输入数据变少会减少Hash表的大小，占用的内存就会减少，从而避免内存溢出OOM的发生。

Spark通过多任务复用Worker的JVM，每个节点所有任务的执行是在同一个JVM上的线程池中执行的，这样就减少了线程的启动开销，可以高效地支持单个任务200ms的执行时间。通过这个机制，可以安全地将任务数量的配置扩展到超过集群的整体的CPU core数，而不会出现问题。

9.3 本章小结

本章主要介绍了Spark程序的性能调优。在应用开发中首先应该是能够让程序运行，第二步才是在静态代码或者运行程序中诊断性能瓶颈，查找造成性能问题的代码或配置项，然后通过性能调优的原则指导Spark的调优，优化改进代码和配置项。过早的优化是万恶之源，在不恰当的时间进行优化会增加程序复杂性以及延缓开发周期。同时我们也看到大数据系统软件栈多，集群环境复杂，需要考虑更多的因素进行性能调优，这是挑战，同时也是机遇。