



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



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



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

Spark 大数据处理: 技术、应用与性能优化 / 高彦杰著. — 北京: 机械工业出版社, 2014.11

(大数据技术丛书)

ISBN 978-7-111-48386-1

I. S… II. 高… III. 数据处理软件 IV. TP274

中国版本图书馆 CIP 数据核字 (2014) 第 246890 号



## Spark 大数据处理: 技术、应用与性能优化

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

责任编辑: 高婧雅

责任校对: 殷虹

印 刷:

版 次: 2014 年 11 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 16.75

书 号: ISBN 978-7-111-48386-1

定 价: 59.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

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個想法。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](mailto:gaoyanjie55@163.com) 联系我，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！



## Contents 目 录

前 言	2.3 本章小结	35
第1章 Spark简介	第3章 Spark计算模型	36
1.1 Spark 是什么	3.1 Spark 程序模型	36
1.2 Spark 生态系统 BDAS	3.2 弹性分布式数据集	37
1.3 Spark 架构	3.2.1 RDD 简介	38
1.4 Spark 分布式架构与单机多核架构的异同	3.2.2 RDD 与分布式共享内存的异同	38
1.5 Spark 的企业级应用	3.2.3 Spark 的数据存储	39
1.5.1 Spark 在 Amazon 中的应用	3.3 Spark 算子分类及功能	41
1.5.2 Spark 在 Yahoo! 的应用	3.3.1 Value 型 Transformation 算子	42
1.5.3 Spark 在西班牙电信的应用	3.3.2 Key-Value 型 Transformation 算子	49
1.5.4 Spark 在淘宝的应用	3.3.3 Actions 算子	53
1.6 本章小结	3.4 本章小结	59
第2章 Spark集群的安装与部署	第4章 Spark工作机制详解	60
2.1 Spark 的安装与部署	4.1 Spark 应用执行机制	60
2.1.1 在 Linux 集群上安装与配置 Spark	4.1.1 Spark 执行机制总览	60
2.1.2 在 Windows 上安装与配置 Spark	4.1.2 Spark 应用的概念	62
2.2 Spark 集群初试	4.1.3 应用提交与执行方式	63



4.2 Spark 调度与任务分配模块 ····· 65	5.5 本章小结 ····· 135
4.2.1 Spark 应用程序之间的调度 ··· 66	
4.2.2 Spark 应用程序内 Job 的 调度 ····· 67	
4.2.3 Stage 和 TaskSetManager 调度方式 ····· 72	
4.2.4 Task 调度 ····· 74	
4.3 Spark I/O 机制 ····· 77	
4.3.1 序列化 ····· 77	
4.3.2 压缩 ····· 78	
4.3.3 Spark 块管理 ····· 80	
4.4 Spark 通信模块 ····· 93	
4.4.1 通信框架 AKKA ····· 94	
4.4.2 Client、Master 和 Worker 间的通信 ····· 95	
4.5 容错机制 ····· 104	
4.5.1 Lineage 机制 ····· 104	
4.5.2 Checkpoint 机制 ····· 108	
4.6 Shuffle 机制 ····· 110	
4.7 本章小结 ····· 119	
<b>第5章 Spark开发环境配置及流程 ··· 120</b>	
5.1 Spark 应用开发环境配置 ····· 120	
5.1.1 使用 IntelliJ 开发 Spark 程序 ····· 120	
5.1.2 使用 Eclipse 开发 Spark 程序 ····· 125	
5.1.3 使用 SBT 构建 Spark 程序 ··· 129	
5.1.4 使用 Spark Shell 开发运行 Spark 程序 ····· 130	
5.2 远程调试 Spark 程序 ····· 130	
5.3 Spark 编译 ····· 132	
5.4 配置 Spark 源码阅读环境 ····· 135	
<b>第6章 Spark编程实战 ····· 136</b>	
6.1 WordCount ····· 136	
6.2 Top K ····· 138	
6.3 中位数 ····· 140	
6.4 倒排索引 ····· 141	
6.5 CountOnce ····· 143	
6.6 倾斜连接 ····· 144	
6.7 股票趋势预测 ····· 146	
6.8 本章小结 ····· 153	
<b>第7章 Benchmark使用详解 ····· 154</b>	
7.1 Benchmark 简介 ····· 154	
7.1.1 Intel Hibench 与 Berkeley BigDataBench ····· 155	
7.1.2 Hadoop GridMix ····· 157	
7.1.3 Bigbench、BigDataBenchmark 与 TPC-DS ····· 158	
7.1.4 其他 Benchmark ····· 161	
7.2 Benchmark 的组成 ····· 162	
7.2.1 数据集 ····· 162	
7.2.2 工作负载 ····· 163	
7.2.3 度量指标 ····· 167	
7.3 Benchmark 的使用 ····· 168	
7.3.1 使用 Hibench ····· 168	
7.3.2 使用 TPC-DS ····· 170	
7.3.3 使用 BigDataBench ····· 172	
7.4 本章小结 ····· 176	
<b>第8章 BDAS简介 ····· 177</b>	
8.1 SQL on Spark ····· 177	
8.1.1 使用 Spark SQL 的原因 ····· 178	

8.1.2	Spark SQL 架构分析 ·····	179
8.1.3	Shark 简介 ·····	182
8.1.4	Hive on Spark ·····	184
8.1.5	未来展望 ·····	185
8.2	Spark Streaming ·····	185
8.2.1	Spark Streaming 简介 ·····	186
8.2.2	Spark Streaming 架构 ·····	188
8.2.3	Spark Streaming 原理剖析 ···	189
8.2.4	Spark Streaming 调优 ·····	198
8.2.5	Spark Streaming 实例 ·····	198
8.3	GraphX ·····	205
8.3.1	GraphX 简介 ·····	205
8.3.2	GraphX 的使用 ·····	206
8.3.3	GraphX 架构 ·····	209
8.3.4	运行实例 ·····	211
8.4	Mllib ·····	215
8.4.1	Mllib 简介 ·····	217
8.4.2	Mllib 的数据存储 ·····	219
8.4.3	数据转换为向量(向量空间模型 VSM)·.....	222
8.4.4	Mllib 中的聚类和分类 ·····	223
8.4.5	算法应用实例 ·····	228
8.4.6	利用 Mllib 进行电影推荐 ···	230
8.5	本章小结 ·····	237
第9章 Spark性能调优 ·····		238
9.1	配置参数 ·····	238
9.2	调优技巧 ·····	239
9.2.1	调度与分区优化 ·····	240
9.2.2	内存存储优化 ·····	243
9.2.3	网络传输优化 ·····	249
9.2.4	序列化与压缩 ·····	251
9.2.5	其他优化方法 ·····	253
9.3	本章小结 ·····	255

# 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<sup>⊖</sup>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

---

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

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。

1.2 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。

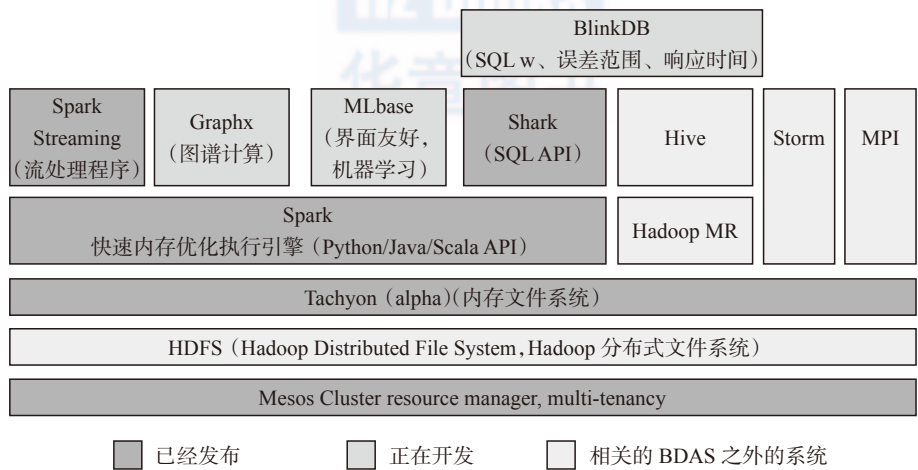


图 1-1 伯克利数据分析栈（BDAS）项目结构图

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）。

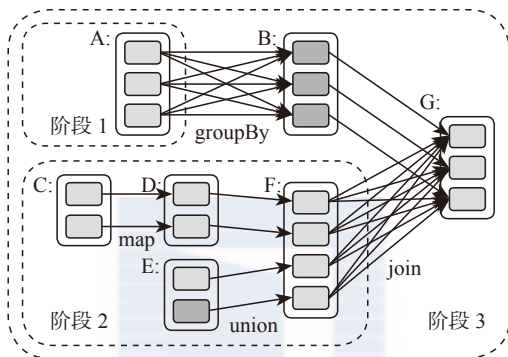


图 1-2 Spark 的任务处理流程图

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<sup>⊖</sup>，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。

## 1.3 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。

---

<sup>⊖</sup> Spark 自带的资源管理框架是 Standalone。



**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。

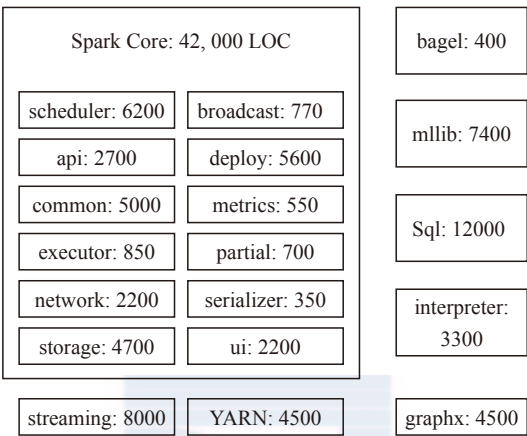


图 1-3 Spark 代码结构和代码量

**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。

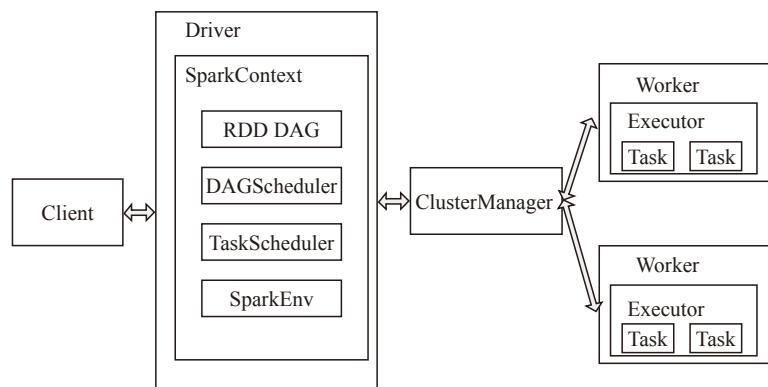


图 1-4 Spark 架构图

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。

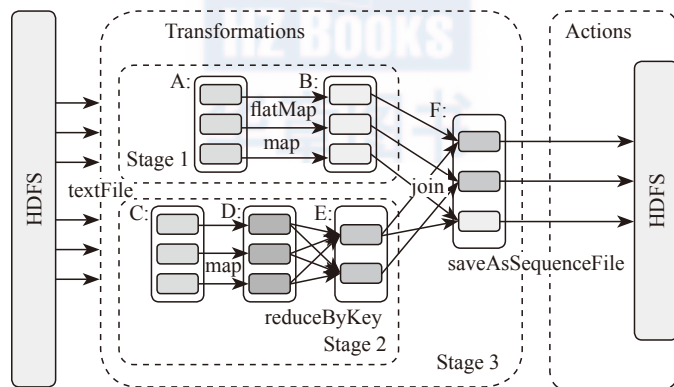


图 1-5 Spark 执行有向无环图

## 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。

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。

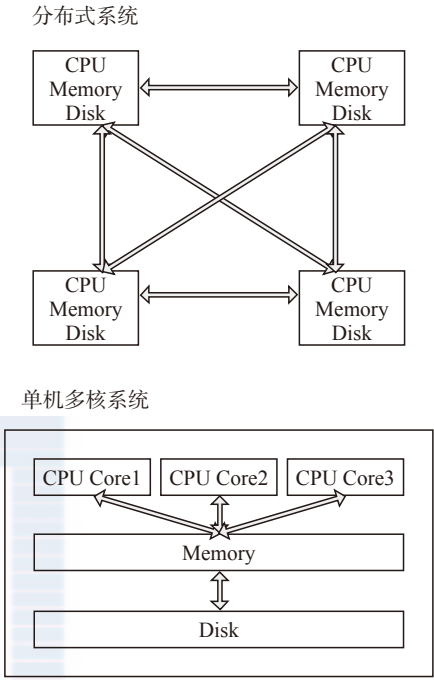


图 1-6 分布式体系结构与单机体系结构的对比

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。

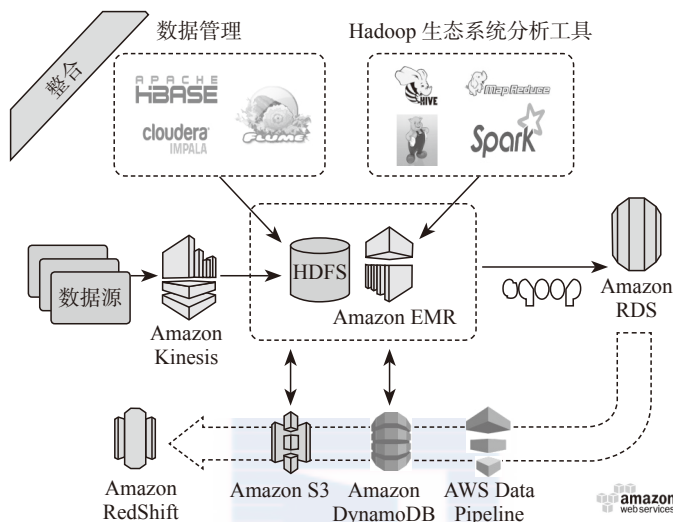


图 1-7 亚马逊 AWS 架构

## 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。

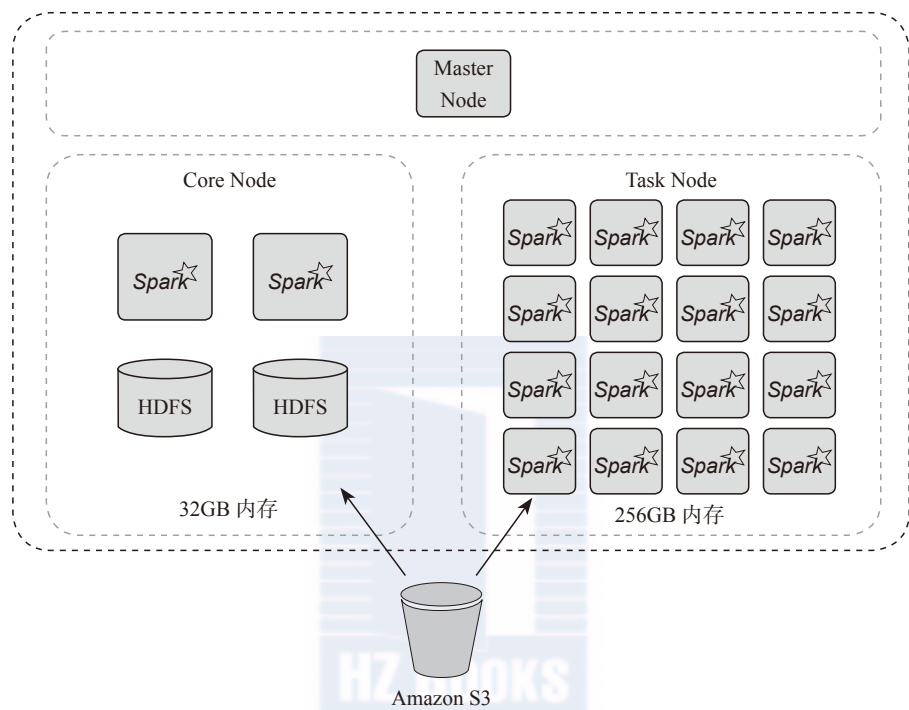


图 1-8 Amazon Spark on EMR

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。

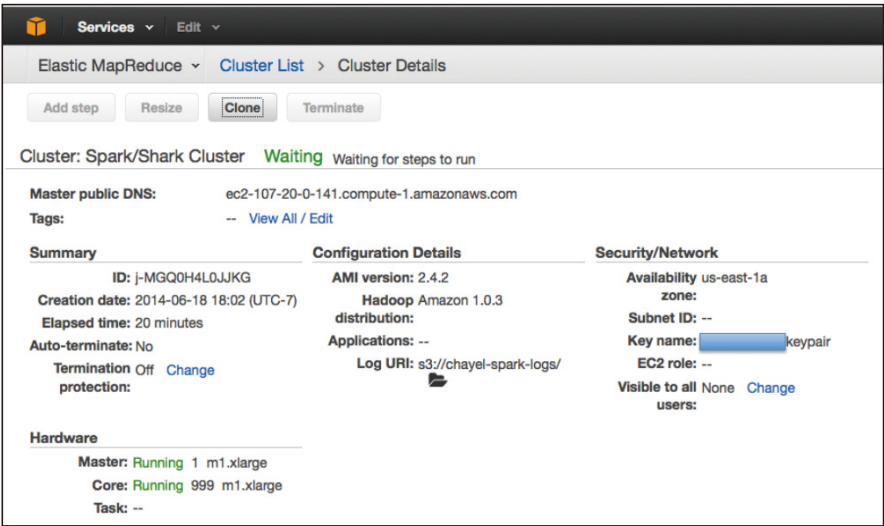


图 1-9 集群细节监控界面

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
TERMINATED BY"
LOCATION 's3n://bigdata-spark-demo/wikistats/';

ALTER TABLE wikistats add partition(dt='2007-12')location 's3n://bigdata-spark-
demo//wikistats/2007/2007-12';
.....

```

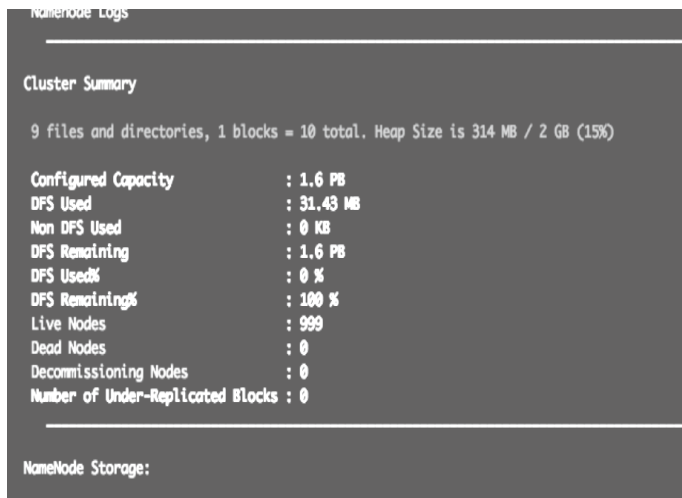


图 1-10 Lynx 界面

#### 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 72 656 600 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。

未来的分析栈

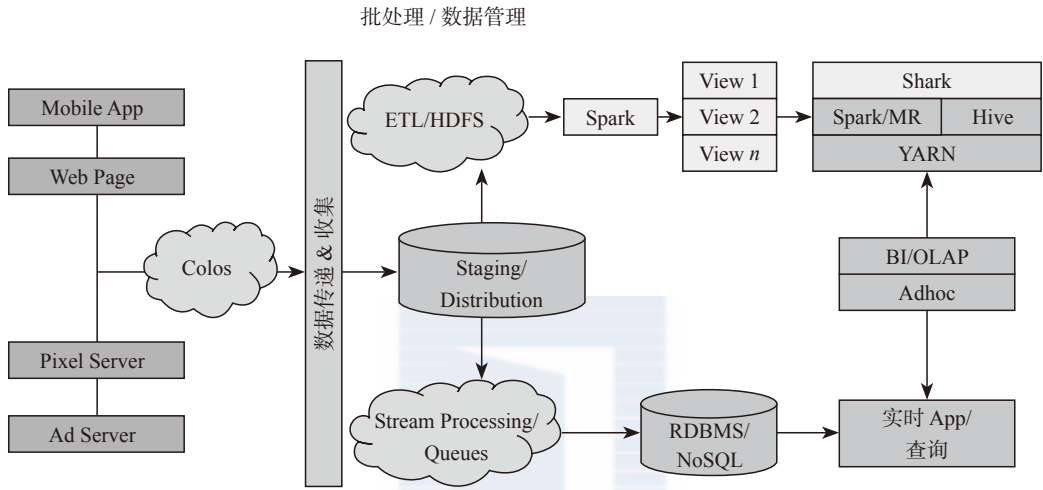


图 1-11 Yahoo! 大数据分析栈

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。

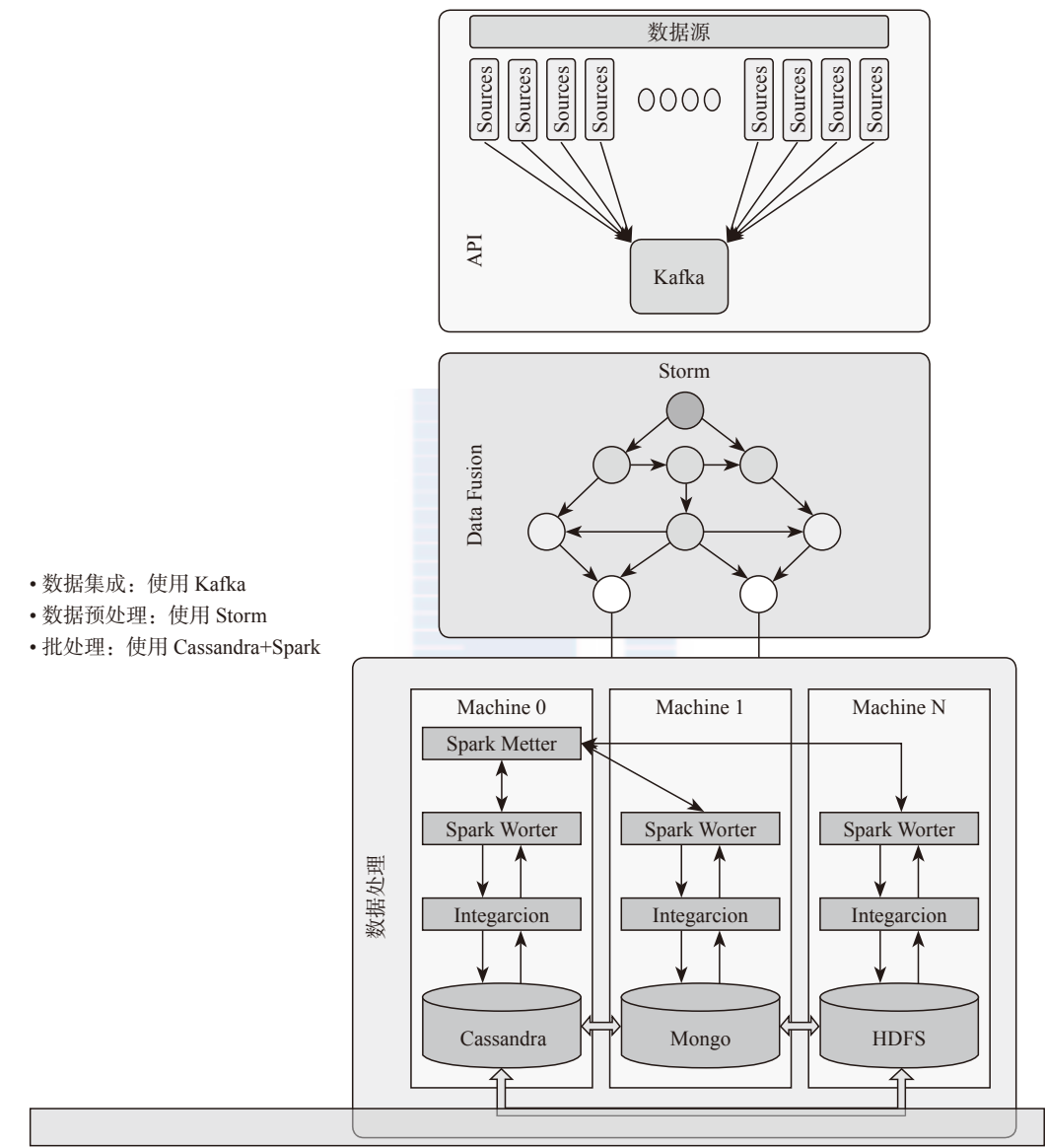


图 1-12 西班牙电信数据分析平台

### 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<sup>①</sup>。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。

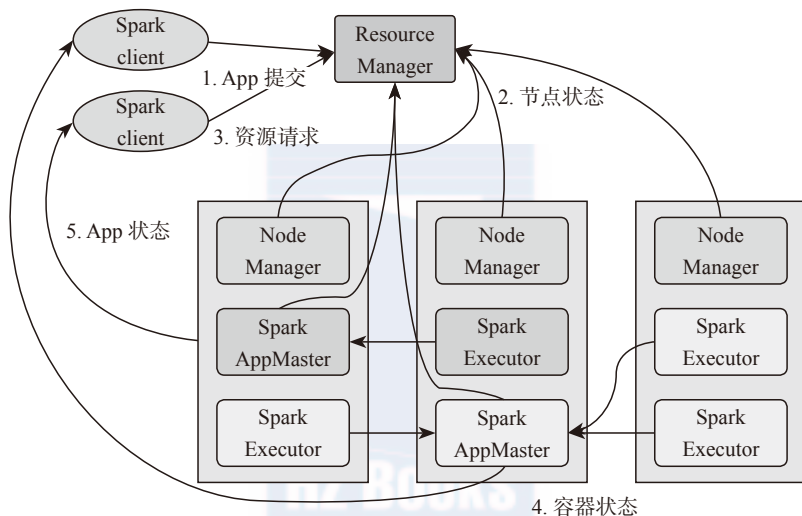


图 1-13 Spark on YARN 架构

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

<sup>①</sup> 参见沈洪的《深入剖析阿里巴巴云梯 YARN 集群》，《程序员》，2013.12。

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<sup>⊖</sup>：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 进行大规模机器学习、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。

## 1.6 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、银行等传统行业也开始逐步试水 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。

---

<sup>⊖</sup> 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。

## Spark 集群的安装与部署

Spark 的安装简便，用户可以在官网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、Windows 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 -xzf 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。

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 Worker 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/>)。

```
$ wget http://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、LIST-
```

FILESTATUS 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>。

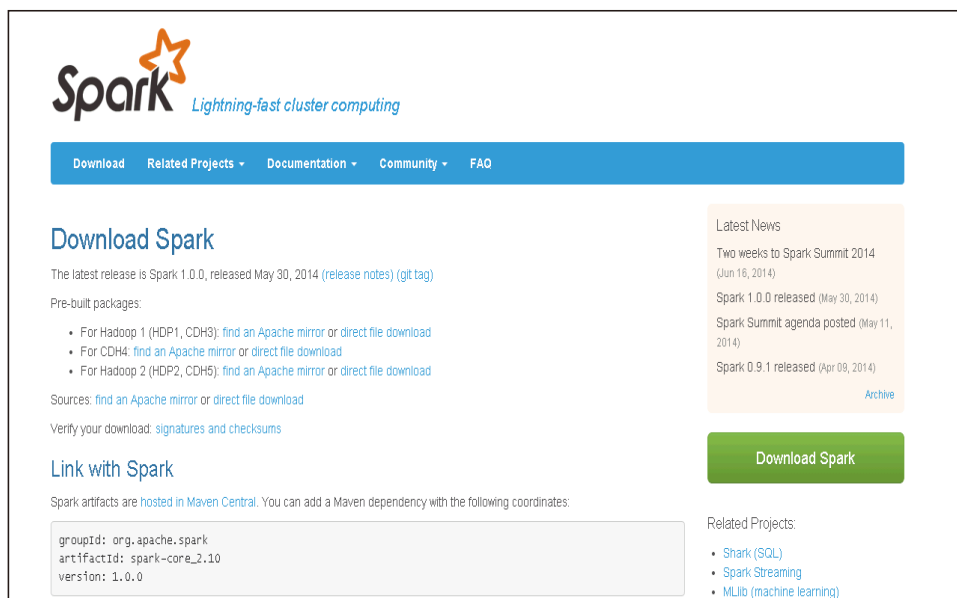


图 2-1 Spark 下载官网

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<sup>①</sup>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。

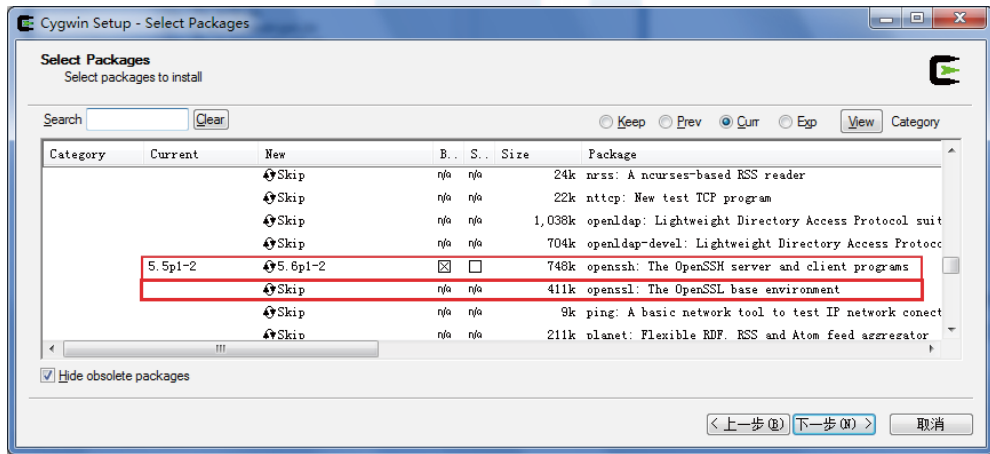


图 2-2 Cygwin 安装选择界面

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

<sup>①</sup> 可以通过官网进行下载：<http://www.cygwin.com/>。



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。

```
$ ssh-host-config -y
*** Query: Overwrite existing /etc/ssh_config file? (yes/no) yes
*** Info: Creating default /etc/ssh_config file
*** Query: Overwrite existing /etc/sshd_config file? (yes/no) yes
*** Info: Creating default /etc/sshd_config file
*** Info: Privilege separation is set to yes by default since OpenSSH 3.3.
*** Info: However, this requires a non-privileged account called 'sshd'.
*** Info: For more info on privilege separation read /usr/share/doc/openssh/README.privsep.
*** Query: Should privilege separation be used? (yes/no) yes
*** Info: Note that creating a new user requires that the current account have
*** Info: Administrator privileges. Should this script attempt to create a
*** Query: new local account 'sshd'? (yes/no) yes
no*** Info: Updating /etc/sshd_config file

*** Query: Do you want to install sshd as a service?
*** Query: (Say "no" if it is already installed as a service) (yes/no) yes
*** Query: Enter the value of CYGWIN for the daemon: []
*** Info: On Windows Server 2003, Windows Vista, and above, the
*** Info: SYSTEM account cannot setuid to other users -- a capability
*** Info: sshd requires. You need to have or to create a privileged
*** Info: account. This script will help you do so.

*** Info: You appear to be running Windows XP 64bit, Windows 2003 Server,
*** Info: or later. On these systems, it's not possible to use the LocalSystem
*** Info: account for services that can change the user id without an
*** Info: explicit password (such as passwordless logins [e.g. public key
*** Info: authentication] via sshd).

*** Info: If you want to enable that functionality, it's required to create
*** Info: a new account with special privileges (unless a similar account
*** Info: already exists). This account is then used to run these special
*** Info: servers.

*** Info: Note that creating a new user requires that the current account
*** Info: have Administrator privileges itself.

*** Info: No privileged account could be found.

*** Info: This script plans to use 'cyg_server'.
*** Info: 'cyg_server' will only be used by registered services.
*** Query: Create new privileged user account 'cyg_server'? (yes/no) yes
*** Info: Please enter a password for new user cyg_server. Please be sure
*** Info: that this password matches the password rules given on your system.
*** Info: Entering no password will exit the configuration.
*** Query: Please enter the password:
*** Query: Reenter:
*** Query: Please enter the password:
*** Query: Reenter:

*** Info: User 'cyg_server' has been created with password 'liu314725'.
*** Info: If you change the password, please remember also to change the
*** Info: password for the installed services which use (or will soon use)
*** Info: the 'cyg_server' account.

*** Info: Also keep in mind that the user 'cyg_server' needs read permissions
*** Info: on all users' relevant files for the services running as 'cyg_server'.
*** Info: In particular, for the sshd server all users' .ssh/authorized_keys
*** Info: files must have appropriate permissions to allow public key
*** Info: authentication. (Re-)running ssh-user-config for each user will set
*** Info: these permissions correctly. [Similar restrictions apply, for
*** Info: instance, for .rhosts files if the rshd server is running, etc].

*** Info: The sshd service has been installed under the 'cyg_server'
*** Info: account. To start the service now, call 'net start sshd' or
*** Info: 'cygrunsrv -S sshd'. Otherwise, it will start automatically
*** Info: after the next reboot.

*** Info: Host configuration finished. Have fun!

$ net start sshd
CYGWIN sshd 服务正在启动。
CYGWIN sshd 服务已经启动成功。
```

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。

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

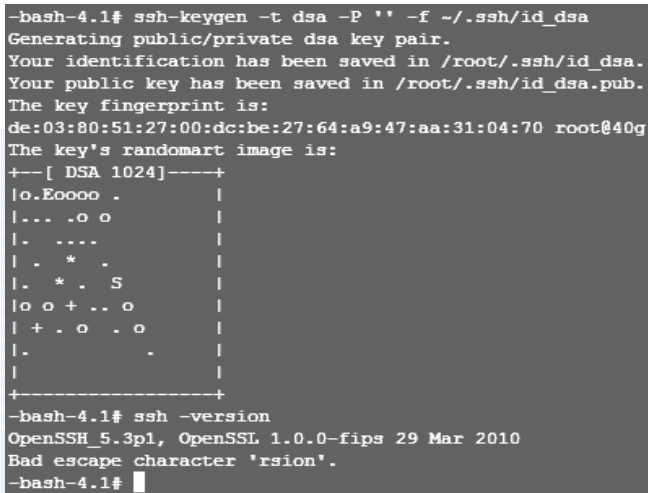
①在 Spark 根目录启动 Spark。

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

②关闭 Spark。

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

2) Hadoop 的启动与关闭



```
-bash-4.1# ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
Generating public/private dsa key pair.
Your identification has been saved in /root/.ssh/id_dsa.
Your public key has been saved in /root/.ssh/id_dsa.pub.
The key fingerprint is:
de:03:80:51:27:00:dc:be:27:64:a9:47:aa:31:04:70 root@40g
The key's randomart image is:
+--[ DSA 1024]-----+
|o.Eoooo .           |
|... .o o            |
|. ....             |
| - * .              |
|. * . S              |
|o o + .. o          |
| + . o . o          |
|.                   |
|                   |
+-----+
-bash-4.1# ssh -version
OpenSSH_5.3p1, OpenSSL 1.0.0-fips 29 Mar 2010
Bad escape character 'rsion'.
-bash-4.1#
```

图 2-4 Cygwin ssh 生成密钥

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。

## 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。

```
import scala.math.random
import org.apache.spark._
object SparkPi {
  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。

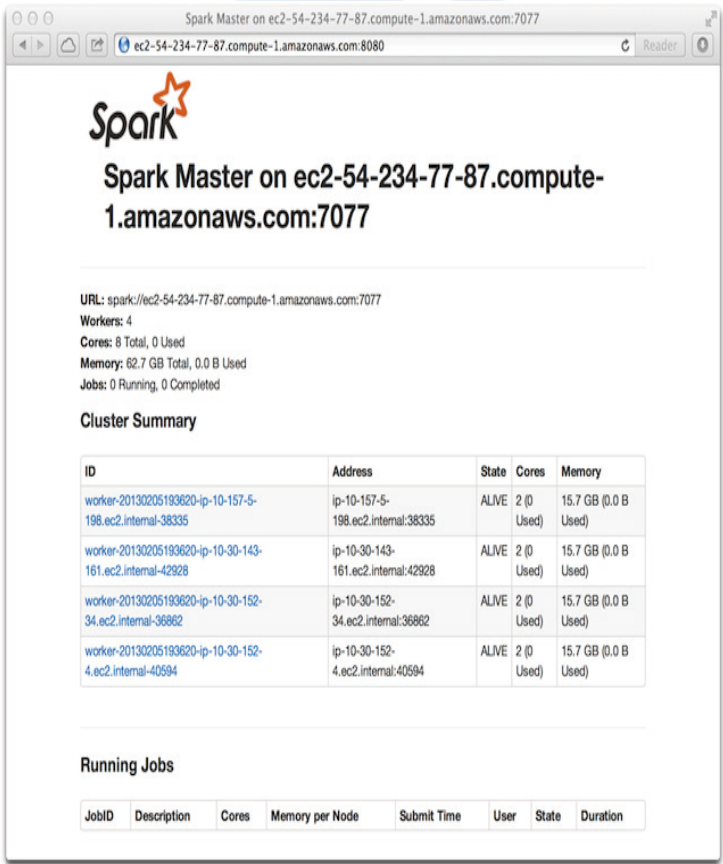


图 2-5 Spark Web UI

## 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。



## 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<sup>⊖</sup> 读取日志文件，输出变量 `file`<sup>⊕</sup>。

---

⊖ 也可以是本地文件或者其他的持久化层，如 Hive 等。

⊕ `file` 是一个 RDD，数据项是文件中的每行数据。

```
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。

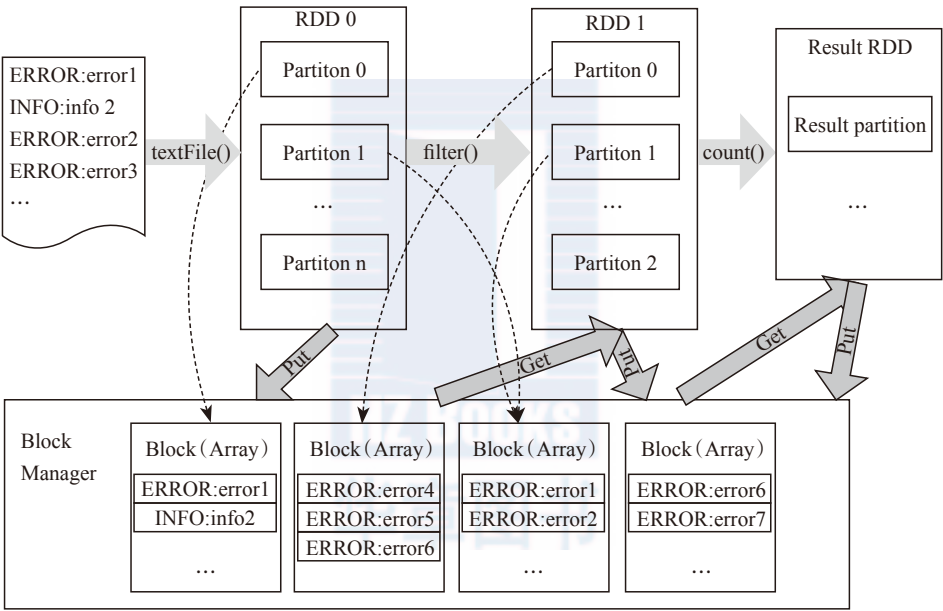


图 3-1 Spark 程序模型

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。

### 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 分区的管理，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<sup>①</sup>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。

---

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。



RDD 与 DSM 主要区别在于<sup>⊖</sup>，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。

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

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

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 Transformation 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。

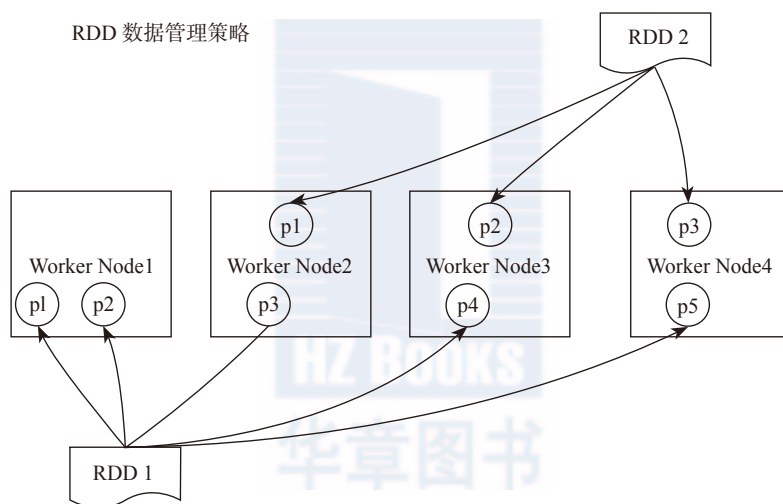


图 3-2 RDD 数据管理模型

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。

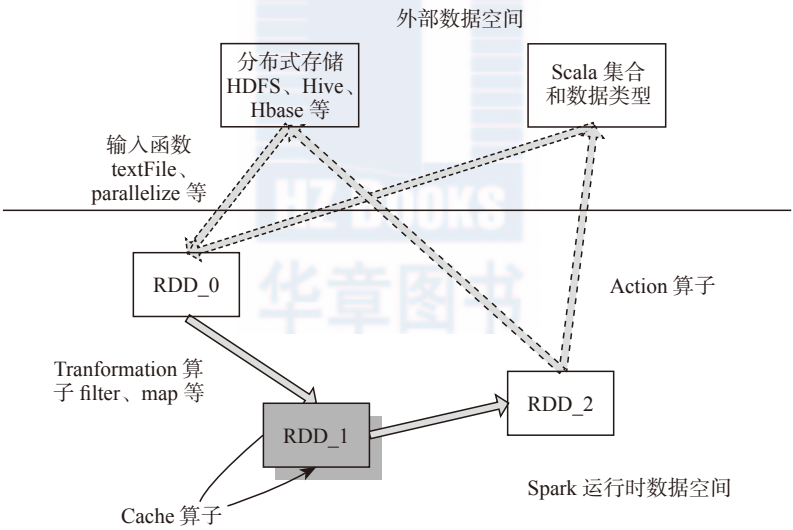


图 3-3 Spark 算子和数据空间

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 Transformation 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 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  $f:T \rightarrow U$  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

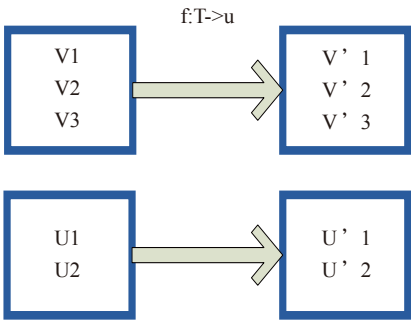


图 3-4 map 算子对 RDD 转换

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  $f:T \rightarrow U$ ，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，形成成为 RDD 中的数据项。

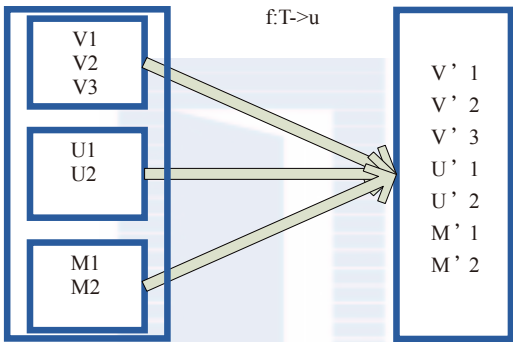


图 3-5 flatMap 算子对 RDD 转换

(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  $f(iter) \Rightarrow iter.filter(\_ \geq 3)$  对分区中的所有数据进行过滤， $\geq 3$  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。

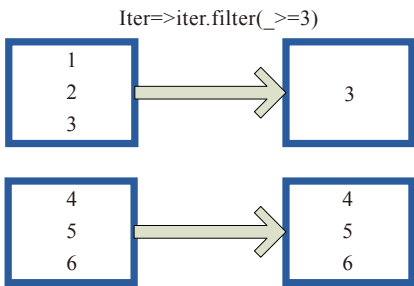


图 3-6 mapPartitions 算子对 RDD 转换

#### (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)]`。

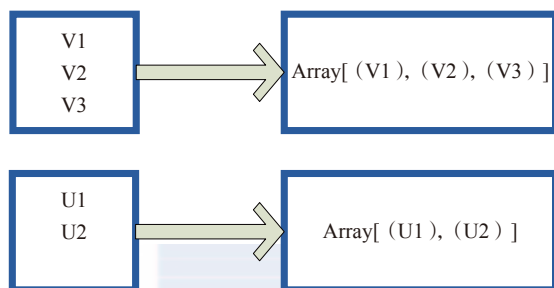


图 3-7 glom 算子对 RDD 转换

## 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 union 函数操作。

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。

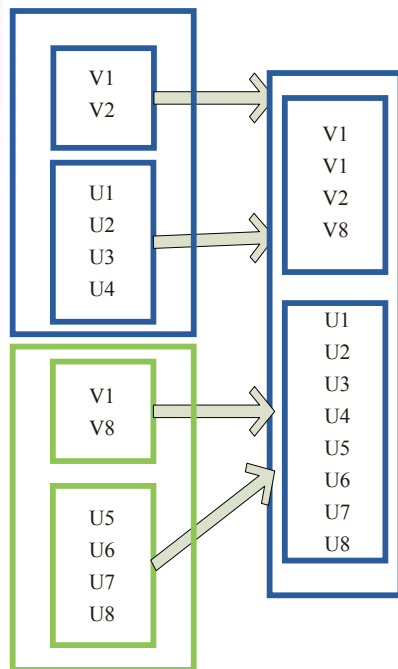


图 3-8 union 算子对 RDD 转换

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)、(V1,W2)、(V2,W2)。

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。

① sc.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。

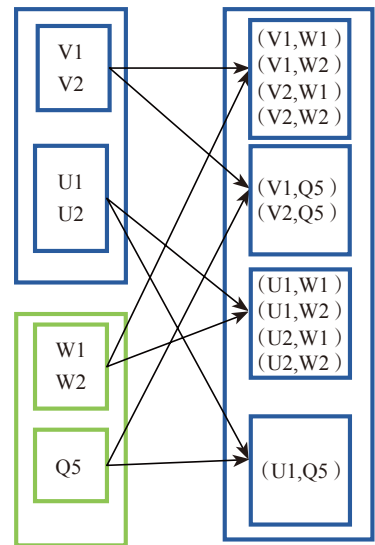


图 3-9 cartesian 算子对 RDD 转换

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)。

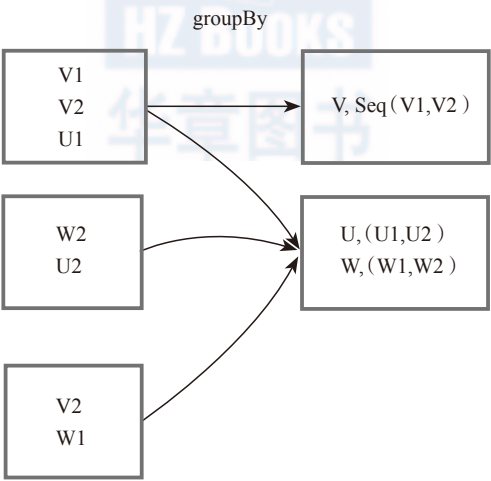


图 3-10 groupBy 算子对 RDD 转换

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'。

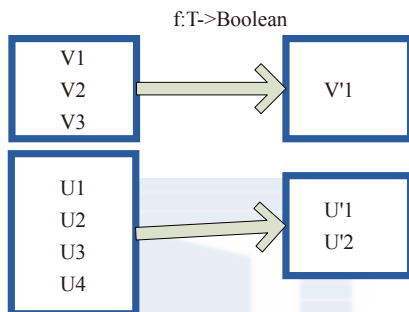


图 3-11 filter 算子对 RDD 转换

## (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。

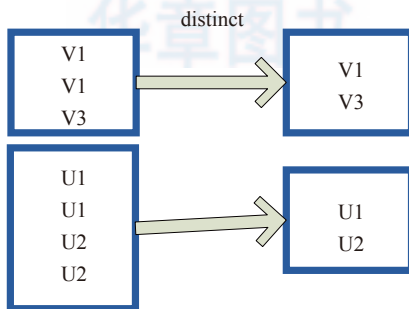


图 3-12 distinct 算子对 RDD 转换

## (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。

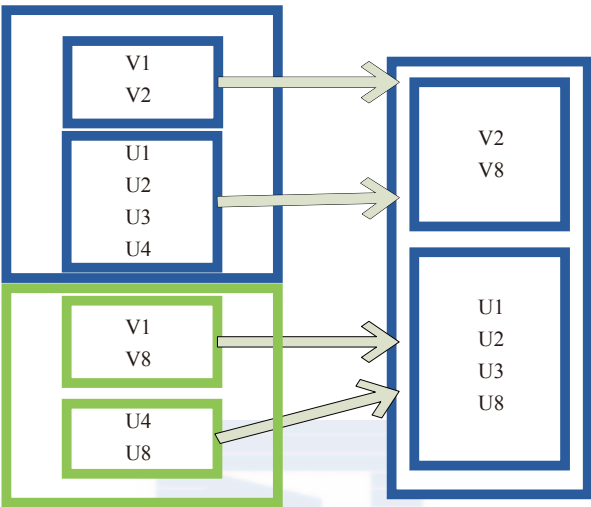


图 3-13 subtract 算子对 RDD 转换

(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 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。

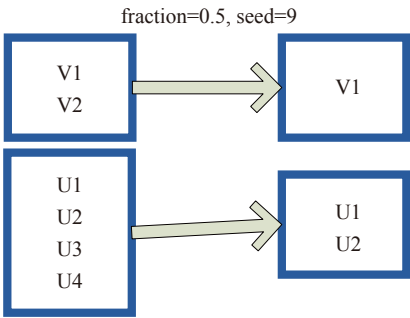


图 3-14 sample 算子对 RDD 转换

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。

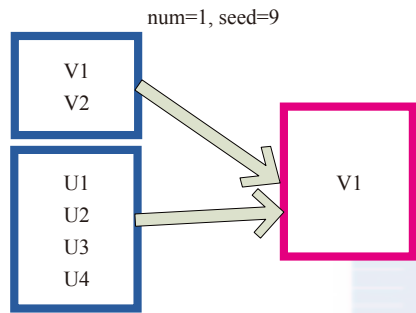


图 3-15 takeSample 算子对 RDD 转换

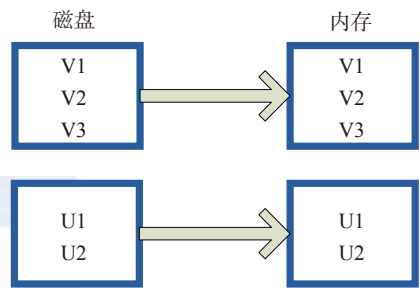


图 3-16 cache 算子对 RDD 转换

(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。

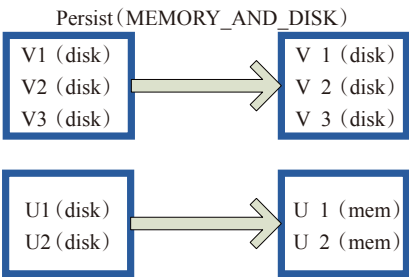


图 3-18 Persist 算子对 RDD 转换

### 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。

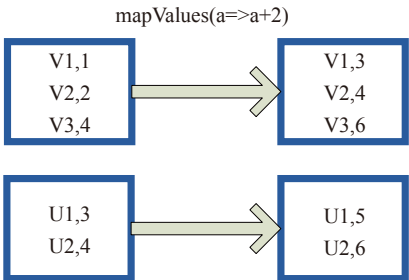


图 3-19 mapValues 算子 RDD 对转换

#### 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))`。

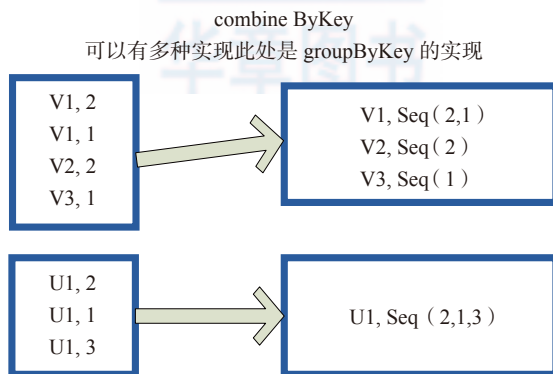


图 3-20 `comBineByKey` 算子对 RDD 转换

## 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  $(A, B) \Rightarrow (A + B)$ ，将相同 Key 的数据  $(V1, 2)$ 、 $(V1, 1)$  的 value 相加，结果为  $(V1, 3)$ 。

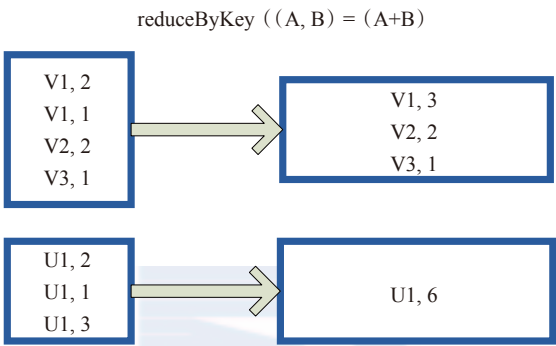


图 3-21 reduceByKey 算子对 RDD 转换

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。

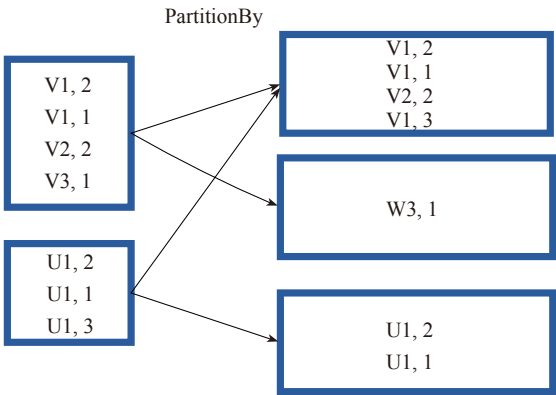


图 3-22 partitionBy 算子对 RDD 转换

## (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)))。

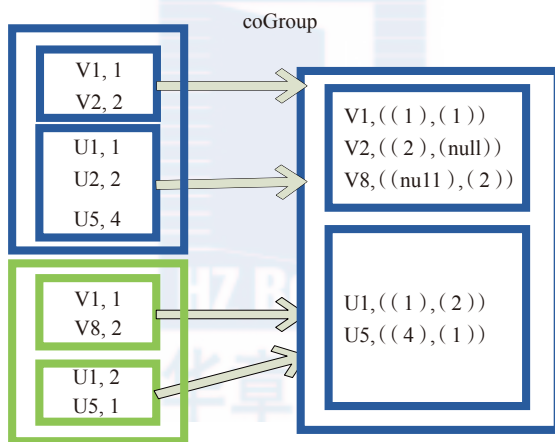


图 3-23 Cogroup 算子对 RDD 转换

### 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))。

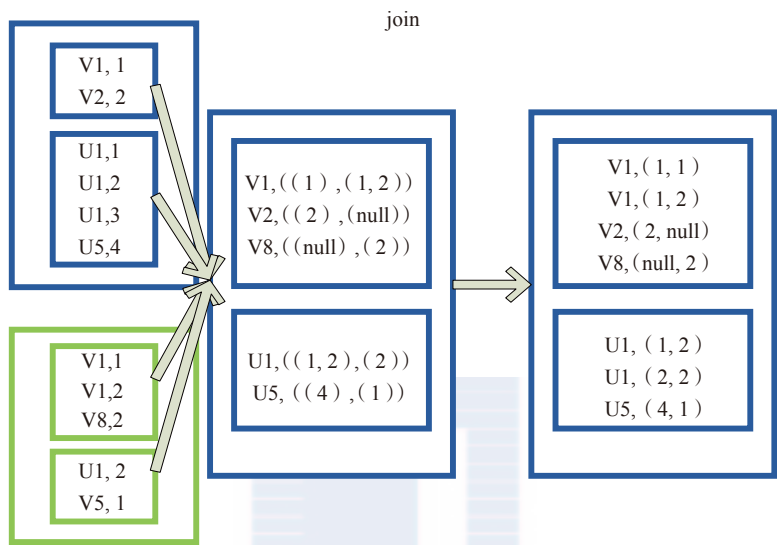


图 3-24 join 算子对 RDD 转换

(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 Unit。

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。

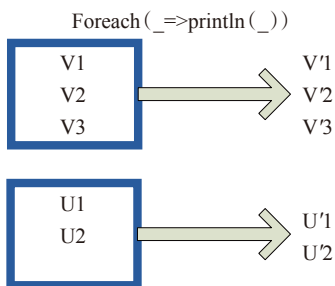


图 3-25 foreach 算子对 RDD 转换

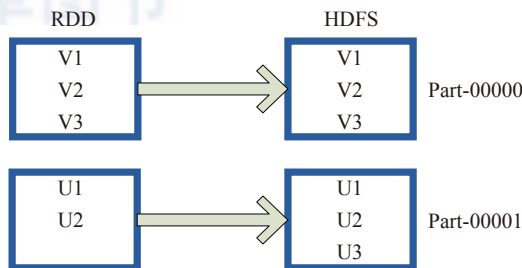


图 3-26 saveAsHadoopFile 算子对 RDD 转换

### (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。

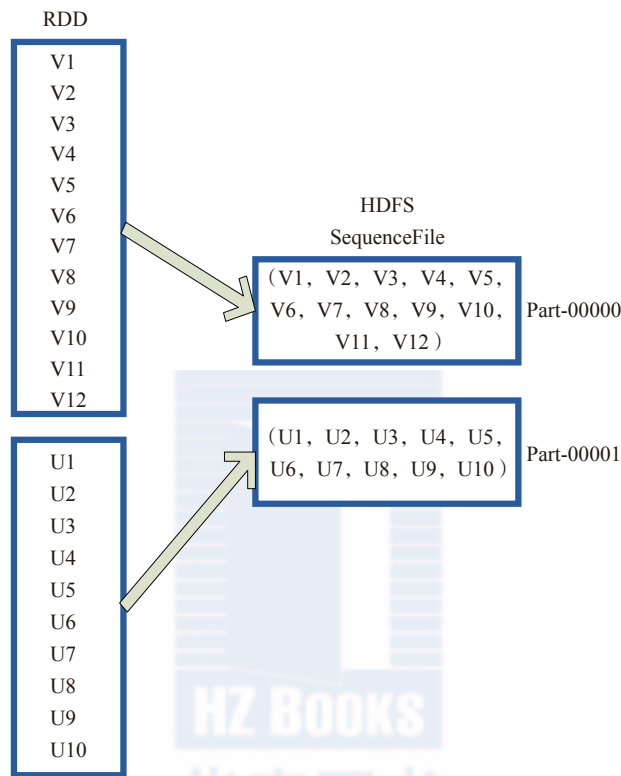


图 3-27 `saveAsObjectFile` 算子对 RDD 转换

### 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。

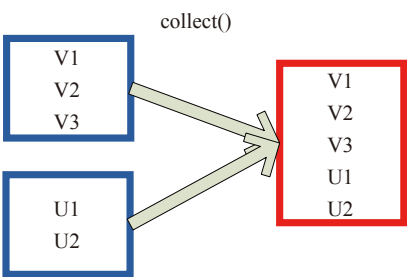


图 3-28 Collect 算子对 RDD 转换

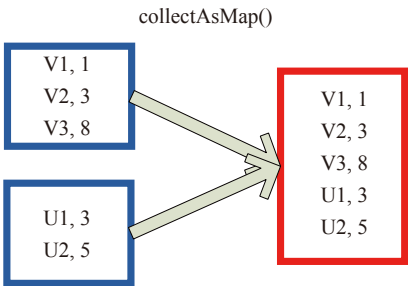


图 3-29 collectAsMap 算子对 RDD 转换

(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,Utills.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。

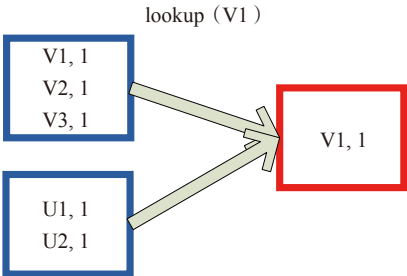


图 3-30 lookup 对 RDD 转换

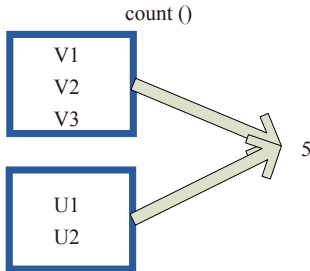


图 3-31 count 对 RDD 转换

(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@V3@U1@U2@U3@U4,12。

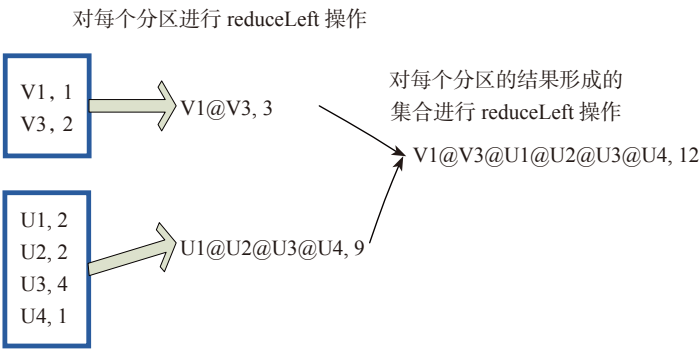


图 3-32 reduce 算子对 RDD 转换

⊖ @ 代表数据的分隔符，可替换为其他分隔符。

## (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) )
```

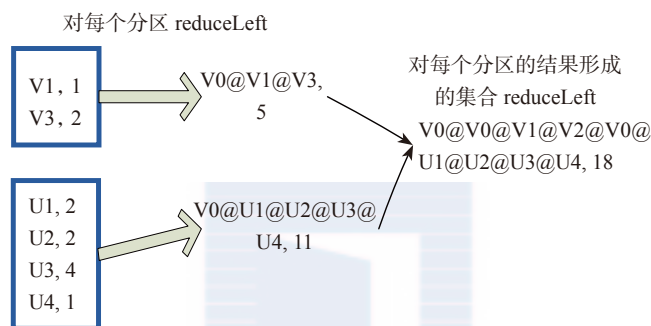


图 3-33 fold 算子对 RDD 转换

## (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。

aggregate 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._2+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<sup>⊖</sup>。

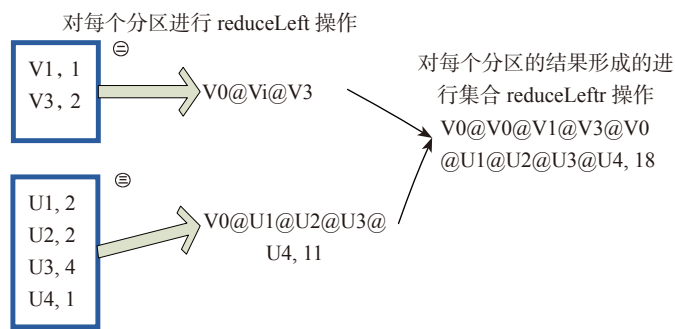


图 3-34 aggregate 算子对 RDD 转换

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。

### 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。

⊖ 参见：Mosharaf Chowdhury, Performance and Scalability of Broadcast in Spark。  
⊖ 代表 V。  
⊖ 代表 U。

作为一个基于内存计算的大数据并行计算框架，Spark不仅很好地解决了数据的实时处理问题，而且保证了高容错性和高可伸缩性。具体来讲，它有如下优势：

- 打造全栈多计算范式的高效数据流水线
- 轻量级快速处理
- 易于使用，支持多语言
- 与HDFS等存储层兼容
- 社区活跃度高

.....

Spark已经在全球范围内广泛使用，无论是Intel、Yahoo!、Twitter、阿里巴巴、百度、腾讯等国际互联网巨头，还有一些尚处于成长期的小公司，都在使用Spark。本书作者结合自己在微软和IBM实践Spark的经历和经验，编写了这本书。站着初学者的角度，不仅系统、全面地讲解了Spark的各项功能及其使用方法，而且较深入地探讨了Spark的工作机制、运行原理以及BDAS生态系统中的其他技术，同时还有一些可供操作的案例，能让没有经验的读者迅速掌握Spark。更为重要的是，本书还对Spark的性能优化进行了探讨。

投稿热线：(010) 88379604  
客服热线：(010) 88378991 88361066  
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com  
网上购书：www.china-pub.com  
数字阅读：www.hzmedia.com.cn

