Broadview®
www.broadview.com.cn

大数据丛书

大数据处理之道

何金池　编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

大数据丛书

大数据
处理之道

何金池　编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书覆盖了当前大数据处理领域的热门技术，包括Hadoop、Spark、Storm、Dremel、Drill等，详细分析了各种技术的应用场景和优缺点；同时阐述了大数据下的日志分析系统，重点讲解了ELK日志处理方案；最后分析了大数据处理技术的发展趋势。

本书采用幽默的表述风格，使读者容易理解、轻松掌握；重点从各种技术的起源、设计思想、架构等方面阐述，以帮助读者从根源上悟出大数据处理之道。

本书适合大数据开发、大数据测试人员，以及其他软件开发或者管理人员和计算爱好者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

大数据处理之道 / 何金池编著．—北京：电子工业出版社，2016.9
（大数据丛书）
ISBN 978-7-121-28723-7
Ⅰ．①大…　Ⅱ．①何…　Ⅲ．①数据处理　Ⅳ．①TP274
中国版本图书馆CIP数据核字（2016）第204120号

责任编辑：刘　皎
特约编辑：赵树刚
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱　　邮编：100036
开　　本：720×1000　1/16　印张：17.75　字数：341千字
版　　次：2016年9月第1版
印　　次：2016年9月第1次印刷
定　　价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

前言

近年来，“大数据”已然成为IT界如火如荼的词，与“云计算”并驾齐驱，成为带动IT行业发展的两列高速火车。尤其是在物联网快速发展的时代，数据已经被称为新的资源，是支撑物联网发展的基石。

那么，如何把“死”的数据变成真正有效的“资源”，成为近年来IT界人士共同思考的问题。一时间，各种大数据处理技术如井喷一般涌现。Hadoop、Spark、Storm、Dremel、Drill等大数据解决方案争先恐后地展现出来。需要说明的是，这里所有的方案并不是一种技术，而是数种甚至数十种技术的组合。就拿Hadoop来说，Hadoop只是“领头羊”，关键成员还有MapReduce、HDFS、Hive、HBase、Pig、ZooKeeper等，大有“八仙过海，各显神通”的气势和场面。

本书首先横向总结性地阐述了各种大数据处理技术，重点从缘起缘落、设计思想、架构原理等角度剖析了各种技术，分析了各种技术的优缺点和适用场景。本书并不涉及软件的安装等，因为如何安装和使用，在网络上搜索即可，着实没有必要浪费读者的时间和金钱。在这一部分，第1篇为Hadoop军营；第2篇为Spark星火燎原；第3篇讲述了其他大数据处理技术，如Storm、Dremel、Drill等。

其次阐述了大数据下的日志分析技术。在大数据时代，日志分析方案呈现出遍地开花的景象。如果将大数据处理系统比作一个可能得病的人，那么日志分析就是负责看病的医生，要想让大数据处理系统健康、平稳地运行，日志分析和监控非常重要。这一部分重点阐述了日志分析技术中如日中天的方案ELK。

最后展望了大数据处理技术的发展趋势。大数据处理技术发展迅猛，数据量越来越大，技术的革新在所难免。

作为大数据研发人员，只有时刻学习新技术，方能立于技术前沿。

由于时间仓促，书中难免出现不足之处，恳请读者指正。本书编写过程中得到了团队其他成员的支持，贡献力量的有张帅、王占伟、李峰、欧立奇等，在此衷心感谢朋友和家人的鼎力支持。

闲言少叙，直接上干货吧！亲爱的读者朋友，请吧……

目录

0 “疯狂”的大数据 1

0.1 大数据时代 1
0.2 数据就是“金库” 3
0.3 让大数据“活”起来 4

第1篇 Hadoop军营

1 Hadoop一石激起千层浪 7

1.1 Hadoop诞生——不仅仅是玩具 7
1.2 Hadoop发展——各路英雄集结 8
1.3 Hadoop和它的小伙伴们 10
1.4 Hadoop应用场景 12
1.5 小结 13

2 MapReduce奠定基石 14

2.1 MapReduce设计思想 14
2.2 MapReduce运行机制 19

2.2.1 MapReduce 的组成 …… 19
2.2.2 MapReduce 作业运行流程 …… 20
2.2.3 JobTracker 解剖 …… 26
2.2.4 TaskTracker 解剖 …… 34
2.2.5 失败场景分析 …… 42
2.3 MapReduce 实例分析 …… 43
2.3.1 运行 WordCount 程序 …… 44
2.3.2 WordCount 源码分析 …… 45
2.4 小结 …… 48

3 分布式文件系统 49

3.1 群雄并起的 DFS …… 49
3.2 HDFS 文件系统 …… 51
3.2.1 HDFS 设计与架构 …… 52
3.2.2 HDFS 操作与 API …… 56
3.2.3 HDFS 的优点及适用场景 …… 60
3.2.4 HDFS 的缺点及改进策略 …… 61
3.3 小结 …… 62

4 Hadoop 体系的“四剑客” 63

4.1 数据仓库工具 Hive …… 63
4.1.1 Hive 缘起何处 …… 63
4.1.2 Hive 和数据库的区别 …… 65
4.1.3 Hive 设计思想与架构 …… 66
4.1.4 适用场景 …… 74
4.2 大数据仓库 HBase …… 74

4.2.1　HBase 因何而生 74
4.2.2　HBase 的设计思想和架构 77
4.2.3　HBase 优化技巧 84
4.2.4　HBase 和 Hive 的区别 86
4.3　Pig 编程语言 87
4.3.1　Pig 的缘由 87
4.3.2　Pig 的基本架构 88
4.3.3　Pig 与 Hive 的对比 90
4.3.4　Pig 的执行模式 90
4.3.5　Pig Latin 语言及其应用 91
4.4　协管员 ZooKeeper 96
4.4.1　ZooKeeper 是什么 96
4.4.2　ZooKeeper 的作用 97
4.4.3　ZooKeeper 的架构 98
4.4.4　ZooKeeper 的数据模型 100
4.4.5　ZooKeeper 的常用接口及操作 102
4.4.6　ZooKeeper 的应用场景分析 105
4.5　小结 108
5　Hadoop 资源管理与调度　110
5.1　Hadoop 调度机制 110
5.1.1　FIFO 111
5.1.2　计算能力调度器 111
5.1.3　公平调度器 113
5.2　Hadoop YARN 资源调度 114

5.2.1 YARN 产生的背景......114
5.2.2 Hadoop YARN 的架构......116
5.2.3 YARN 的运作流程......118
5.3 Apache Mesos 资源调度......120
5.3.1 Apache Mesos 的起因......120
5.3.2 Apache Mesos 的架构......121
5.3.3 基于 Mesos 的 Hadoop......123
5.4 Mesos 与 YARN 对比......127
5.5 小结......128
6 Hadoop 集群管理之道 129
6.1 Hadoop 集群管理与维护......129
6.1.1 Hadoop 集群管理......129
6.1.2 Hadoop 集群维护......131
6.2 Hadoop 集群调优......132
6.2.1 Linux 文件系统调优......132
6.2.2 Hadoop 通用参数调整......133
6.2.3 HDFS 相关配置......133
6.2.4 MapReduce 相关配置......134
6.2.5 Map 任务相关配置......136
6.2.6 HBase 搭建重要的 HDFS 参数......137
6.3 Hadoop 集群监控......137
6.3.1 Apache Ambari 监控......137
6.3.2 Ganglia 监控 Hadoop......138
6.4 小结......138

第 2 篇 Spark 星火燎原

7 Spark 宝刀出鞘 141

7.1 Spark 的历史渊源 141
7.1.1 Spark 的诞生 141
7.1.2 Spark 的发展 142
7.2 Spark 和 Hadoop MapReduce 对比 143
7.3 Spark 的适用场景 145
7.4 Spark 的硬件配置 146
7.5 Spark 架构 147
7.5.1 Spark 生态架构 147
7.5.2 Spark 运行架构 149
7.6 小结 151

8 Spark 核心 RDD 153

8.1 RDD 简介 153
8.1.1 什么是 RDD 153
8.1.2 为什么需要 RDD 154
8.1.3 RDD 本体的设计 154
8.1.4 RDD 与分布式共享内存 155
8.2 RDD 的存储级别 155
8.3 RDD 依赖与容错 157
8.3.1 RDD 依赖关系 157
8.3.2 RDD 容错机制 160
8.4 RDD 操作与接口 161

8.4.1 RDD Transformation 操作与接口 162
8.4.2 RDD Action 操作与接口 164
8.5 RDD 编程示例 165
8.6 小结 166

9 Spark 运行模式和流程 167

9.1 Spark 运行模式 167
9.1.1 Spark 的运行模式列表 167
9.1.2 Local 模式 168
9.1.3 Standalone 模式 169
9.1.4 Spark on Mesos 模式 171
9.1.5 Spark on YARN 模式 173
9.1.6 Spark on EGO 模式 175
9.2 Spark 作业流程 177
9.2.1 YARN-Client 模式的作业流程 178
9.2.2 YARN-Cluster 模式的作业流程 179
9.3 小结 181

10 Shark 和 Spark SQL 183

10.1 从 Shark 到 Spark SQL 183
10.1.1 Shark 的撤退是进攻 183
10.1.2 Spark SQL 接力 185
10.1.3 Spark SQL 与普通 SQL 的区别 186
10.2 Spark SQL 应用架构 187
10.3 Spark SQL 之 DataFrame 188
10.3.1 什么是 DataFrame 188

10.3.2　DataFrame 的创建......188

10.3.3　DataFrame 的使用......190

10.4　Spark SQL 运行过程分析......190

10.5　小结......192

11　Spark Streaming 流数据处理新贵　193

11.1　Spark Streaming 是什么......193

11.2　Spark Streaming 的架构......194

11.3　Spark Streaming 的操作......195

11.3.1　Spark Streaming 的 Transformation 操作......196

11.3.2　Spark Streaming 的 Window 操作......197

11.3.3　Spark Streaming 的 Output 操作......198

11.4　Spark Streaming 性能调优......198

11.5　小结......200

12　Spark GraphX 图计算系统　201

12.1　图计算系统......201

12.1.1　图存储模式......202

12.1.2　图计算模式......203

12.2　Spark GraphX 的框架......206

12.3　Spark GraphX 的存储模式......207

12.4　Spark GraphX 的图运算符......208

12.5　小结......211

13　Spark Cluster 管理　212

13.1　Spark Cluster 部署......212

13.2　Spark Cluster 管理与监控......213

13.2.1 内存优化机制 213
13.2.2 Spark 日志系统 213
13.3 Spark 高可用性 215
13.4 小结 216

第 3 篇 其他大数据处理技术

14 专为流数据而生的 Storm 218
14.1 Storm 起因 218
14.2 Storm 的架构与组件 220
14.3 Storm 的设计思想 222
14.4 Storm 与 Spark 的区别 224
14.5 Storm 的适用场景 225
14.6 Storm 的应用 226
14.7 小结 227
15 Dremel 和 Drill 228
15.1 Dremel 和 Drill 的历史背景 228
15.2 Dremel 的原理与应用 230
15.3 Drill 的架构与流程 232
15.4 Dremel 和 Drill 的适用场景与应用 234
15.5 小结 234

第 4 篇 大数据下的日志分析系统

16 日志分析解决方案 236
16.1 百花齐放的日志处理技术 236

16.2 日志处理方案 ELK......238
16.2.1 ELK 的三大金刚......238
16.2.2 ELK 的架构......240
16.2.3 ELK 的组网形式......242
16.3 Logstash 日志收集解析......245
16.3.1 Input Plugins 及应用示例......246
16.3.2 Filter Plugins 及应用示例......248
16.3.3 Output Plugins 及应用示例......249
16.4 ElasticSearch 存储与搜索......250
16.4.1 ElasticSearch 的主要概念......251
16.4.2 ElasticSearch Rest API......252
16.5 Kibana 展示......253
16.6 小结......255
17 ELK 集群部署与应用 256
17.1 ELK 集群部署与优化......256
17.1.1 ELK HA 集群部署......256
17.1.2 ElasticSearch 优化......257
17.2 如何开发自己的插件......259
17.3 ELK 在大数据运维系统中的应用......261
17.4 ELK 实战应用......262
17.4.1 ELK 监控 Spark 集群......262
17.4.2 ELK 监控系统资源状态......263
17.4.3 ELK 辅助日志管理和故障排查......263
17.5 小结......264

第5篇　数据分析技术前景展望

18　大数据处理的思考与展望　266

18.1　大数据时代的思考　266
18.2　大数据处理技术的发展趋势　267
18.3　小结　270

第 2 篇

Spark 星火燎原

7 Spark宝刀出鞘

7.1 Spark的历史渊源

7.1.1 Spark的诞生

任何一个强大事物的出现都有一个精彩有趣的历史背景，Spark也不例外。Spark最初的思想火花来自加利福尼亚大学伯克利分校（UC Berkeley）的一支研究团队。你可能会说，“哇，外国人在大数据处理方面着实很有创新力”。但是仅有创新力是不够的，必须要有创新的动机，才能造就今天Spark全球开花的局面。

说到Spark的创新动机，这里有个很有趣的故事。话说Netflix（一家在线影片租赁公司）为了提高服务质量，更好地推荐适合用户口味的电影，搞了一个Netflix大奖赛，这个大奖赛从2006年10月份开始。Netflix公开了50万用户针对2万部电影的大约1亿个匿名影片评级，数据集仅包含影片名称、评价星级和评级日期，没有任何文本评价的内容。大奖赛的目的在于预测Netflix的客户分别喜欢什么影片，要求把预测的准确度提高10%以上，以此来为用户推荐最适合的影片。最主要的是大奖赛的奖金为100万美元。

可谓重赏之下必有勇夫！一时间竟在IT界掀起一层不小的波浪，引来无数程序员跃跃欲试，如何处理这“1亿个匿名影片评级”的大数据，摆在了众多参赛者

的面前。UC Berkeley 研究团队中有位名叫 Lester Mackey 的博士生，偶然的机会看到了这个悬赏，也打算试上一把。Lester 当时在 AMPLab 的大数据实验室进行博士研究，有着很好的资源和大数据处理能力。看到这个需求，Lester 马上想到了当时业界已经很流行的 Hadoop MapReduce。研究了一段时间，Lester 发现将很多精力用到了 MapReduce 的编程模型和低效的执行模式上，而不是花费在如何提高效率上。这样下去，等到把预测的准确度提高 10%，100 万美元奖金就与自己失之交臂了。痛彻思悟数天依然无果，便去找实验室的另一位专门研究分布式系统的人 Matei Zaharia 请教。说起 Matei，当时在业界已经小有名气，他是 Hadoop 的重要贡献者之一。二人一拍即合，共同分析了 Hadoop MapReduce 在此应用上的乏力之处，吸其精华，改其不足，定制化地输出了区区几百行代码，满足了 Lester 可以高效率地分布式建模的愿望。这就是 Spark 最初的版本。

看到这里，读者可能会想，拿着这么高大上的“宝剑”去参加 Netflix 武林大会，必赢无疑吧？但是我要遗憾地告诉你，Lester 所在的团队并没有抱得大奖归。虽然提高效率上和另一支团队的方案相差无几，并列第一，可是 Lester 所在的团队晚提交了 20 分钟，终和大奖擦肩而过。不过相比今天 Spark 给大数据处理方面带来的贡献，区区百万美元，实乃冰山一角。追溯此二人的发展，此后都成了学术界的杰出人物，Lester 成了斯坦福大学的教授，Matei 则成了麻省理工学院的教授，二人联合创办了 DatatBricks 公司，此公司正在主导并致力于 Spark 的高速发展。

7.1.2 Spark 的发展

2009 年，Spark 诞生于加利福尼亚大学伯克利分校的 AMPLab，最初属于伯克利分校的研究性项目。它于 2010 年正式开源，并于 2013 年捐赠给 Apache 来管理源码和后续发展，并于 2014 年成为 Apache 基金会的顶级项目。整个过程不到 5 年，发展速度如此之快，着实让人大为惊诧。如今已经逐渐形成了 Spark 生态圈，其对大数据的支持，从内存计算和流处理，到交互式查询，再到图计算和机器学习等，都在快速发展。如前文所述，在大数据处理方面，Hadoop MapReduce 已如日中天，是一种通用的大数据处理系统。还有诸如 Pregel、Storm、Griaph 等特殊化的大数据处理系统。Spark 作为后起之秀，不断地挑战 MapReduce，当然也会面临新产品的挑战。

关于 Spark 的发展历程，官方网站公布了每个 Release 版本新增的特性，这里

无须逐一展示。因为在很多时候人们更加关心的是 Spark 目前能做什么，以及不能做什么，而不是从什么时候可以做的。

Spark 以其 RDD 模型的强大表现能力，不断完善自己的功能，逐渐形成了一套自己的生态系统，其中主要包括 Spark 内存中的批处理、Shark 交互式查询、Spark Streaming 流式计算三大部分。此外还有 GraphX 和 MLBase 提供的常用图计算和机器学习算法。Spark 开源生态系统得到了大幅增长，已成为大数据领域最活跃的开源项目之一，当前主要贡献的公司和组织有 IBM、Hortonworks、Cloudera、MapR、Pivotal、华为、阿里巴巴、百度等。

7.2 Spark 和 Hadoop MapReduce 对比

一提到大数据处理，相信很多人第一时间想到的是 Hadoop MapReduce。没错，Hadoop MapReduce 为大数据处理技术奠定了基础。近年来，随着 Spark 的发展，越来越多的声音提到了 Spark。在业界有两种说法：一是 Spark 将代替 Hadoop MapReduce，成为未来大数据处理发展的方向；二是 Spark 将会和 Hadoop 结合，形成更大的生态圈。其实 Spark 和 Hadoop MapReduce 的重点应用场合有所不同。相对于 Hadoop MapReduce 来说，Spark 有点“青出于蓝”的感觉，Spark 是在 Hadoop MapReduce 模型上发展起来的，在它的身上我们能明显看到 MapReduce 的影子，所有的 Spark 并非从头创新，而是站在了巨人“MapReduce”的肩膀上。千秋功罪，留于日后评说，我们暂且搁下争议，来看看相比 Hadoop MapReduce，Spark 都有哪些优势。

1. 计算速度快

大数据处理首先追求的是速度。Spark 到底有多快？用官方的话说，“Spark 允许 Hadoop 集群中的应用程序在内存中以 100 倍的速度运行，即使在磁盘上运行也能快 10 倍”。可能有的读者看到这里会大为感叹，的确如此，在有迭代计算的领域，Spark 的计算速度远远超过 MapReduce，并且迭代次数越多，Spark 的优势越明显。这是因为 Spark 很好地利用了目前服务器内存越来越大这一优点，通过减少磁盘 I/O 来达到性能提升。它们将中间处理数据全部放到了内存中，仅在必要时才批量存入硬盘中。或许读者会问：如果应用程序特别大，内存能放下多少 GB？答曰：什么？ GB？目前 IBM 服务器内存已经扩展至几 TB 了。

2. 应用灵活，上手容易

知道 AMPLab 的 Lester 为什么放弃 MapReduce 吗？因为他需要把很多精力放到 Map 和 Reduce 的编程模型上，极为不便。Spark 在简单的 Map 及 Reduce 操作之外，还支持 SQL 查询、流式查询及复杂查询，比如开箱即用的机器学习算法。同时，用户可以在同一个工作流中无缝地搭配这些能力，应用十分灵活。

Spark 核心部分的代码为 63 个 Scala 文件，非常的轻量级。并且允许 Java、Scala、Python 开发者在自己熟悉的语言环境下进行工作，通过建立在 Java、Scala、Python、SQL（应对交互式查询）的标准 API 以方便各行各业使用，同时还包括大量开箱即用的机器学习库。它自带 80 多个高等级操作符，允许在 Shell 中进行交互式查询。即使是新手，也能轻松上手应用。

3. 兼容竞争对手

Spark 可以独立运行，除了可以运行在当下的 YARN 集群管理外，还可以读取已有的任何 Hadoop 数据。它可以运行在任何 Hadoop 数据源上，比如 HBase、HDFS 等。有了这个特性，让那些想从 Hadoop 应用迁移到 Spark 上的用户方便了很多。Spark 有兼容竞争对手的胸襟，何愁大事不成？

4. 实时处理性能非凡

MapReduce 更加适合处理离线数据（当然，在 YARN 之后，Hadoop 也可以借助其他工具进行流式计算）。Spark 很好地支持实时的流计算，依赖 Spark Streaming 对数据进行实时处理。Spark Streaming 具备功能强大的 API，允许用户快速开发流应用程序。而且不像其他的流解决方案，比如 Storm，Spark Streaming 无须额外的代码和配置，就可以做大量的恢复和交付工作。

5. 社区贡献力量巨大

从 Spark 的版本演化来看，足以说明这个平台旺盛的生命力及社区的活跃度。尤其自 2013 年以来，Spark 一度进入高速发展期，代码库提交与社区活跃度都有显著增长。以活跃度论，Spark 在所有的 Apache 基金会开源项目中位列前三，相较于其他大数据平台或框架而言，Spark 的代码库最为活跃。

Spark 非常重视社区活动，组织也极为规范，会定期或不定期地举行与 Spark 相关的会议。会议分为两种：一种是 Spark Summit，影响力极大，可谓全球 Spark

顶尖技术人员的峰会，目前已于 2013—2015 年在 San Francisco 连续召开了三届 Summit 大会；另一种是 Spark 社区不定期地在全球各地召开的小型 Meetup 活动。Spark Meetup 也会在我国的一些大城市定期召开，比如北京、深圳、西安等地，读者可以关注当地的微信公众号进行参与。

7.3 Spark 的适用场景

从大数据处理需求来看，大数据的业务大概可以分为以下三类：

（1）复杂的批量数据处理，通常的时间跨度在数十分钟到数小时之间。

（2）基于历史数据的交互式查询，通常的时间跨度在数十秒到数分钟之间。

（3）基于实时数据流的数据处理，通常的时间跨度在数百毫秒到数秒之间。

目前已有很多相对成熟的开源和商业软件来处理以上三种情景：第一种业务，可以利用 MapReduce 来进行批量数据处理；第二种业务，可以用 Impala 来进行交互式查询；对于第三种流式数据处理，可以想到专业的流数据处理工具 Storm。但是这里有一个很重要的问题：对于大多数互联网公司来说，一般会同时遇到以上三种情景，如果采用不同的处理技术来面对这三种情景，那么这三种情景的输入 / 输出数据无法无缝共享，它们之间可能需要进行格式转换，并且每个开源软件都需要一支开发和维护团队，从而提高了成本。另外一个不便之处就是，在同一个集群中对各个系统协调资源分配比较困难。

那么，有没有一种软件可以同时处理以上三种情景呢？ Spark 就可以，或者说有这样的潜力。Spark 同时支持复杂的批处理、互操作和流计算，而且兼容支持 HDFS 和 Amazon S3 等分布式文件系统，可以部署在 YARN 和 Mesos 等流行的集群资源管理器上。

从 Spark 的设计理念（基于内存的迭代计算框架）出发，其最适合有迭代运算的或者需要多次操作特定数据集的应用场合。并且迭代次数越多，读取的数据量越大，Spark 的应用效果就越明显。因此，对于机器学习之类的“迭代式”应用，Spark 可谓拿手好戏，要比 Hadoop MapReduce 快数十倍。另外，Spark Streaming 因为内存存储中间数据的特性，处理速度非常快，也可以应用于需要实时处理大数据的场合。

当然，Spark 也有不适用的场合。对于那种异步细粒度更新状态的应用，例如 Web 服务的存储或增量的 Web 爬虫和索引，也就是对于那种增量修改的应用模型不适合。Spark 也不适合做超级大的数据量的处理，这里所说的“超级大”是相对于这个集群的内存容量而言的，因为 Spark 要将数据存储在内存中。一般来说，10TB 以上（单次分析）的数据就可以算是“超级大”的数据了。

一般来说，对于中小企业的数据中心而言，在单次计算的数据量不大的情况下，Spark 都是很好的选择。另外，Spark 也不适合应用于混合的云计算平台，因为混合的云计算平台的网络传输是很大的问题，即便有专属的宽带在云端 Cluster 和本地 Cluster 之间传输数据，相比内存读取速度来说，依然不抵。

7.4 Spark 的硬件配置

一般来讲，用户如果用 Spark Cluster 的形式来处理大数据，那么集群硬件配置自然较高。Spark 好比一匹宝马，我们要让马儿跑得快，就需要给马儿吃好料。

1. 内存的配置

虽然 Spark 是基于内存的迭代计算框架，但从官方资料来看，Spark 对内存的要求并不高，8GB 即可，当然这和 Spark Cluster 要处理的数据量的大小有关。一般情况下，内存容量越大越好，但这里要注意的是，如果内存太大，则可能需要特殊的配置，因为 JVM 对超大内存（256GB 以上）的管理存在问题。考虑到系统运行需要耗费内存，并且需要为 Application 的运行预留一些 Buffer（缓冲区），所以一般情况下为 Spark 应用分配 75% 的内存空间。如果需要处理超大规模的数据，则可以设置数据集的存储级别（后续讨论），以此来保证内存的高效、实用。

2. CPU 的配置

如果内存足够大，那么制约运算速度的可能就是 CPU 核数了。由于 Spark 实现的是线程之间的最小共享，可以支持一台机器扩展至数十个 CPU 核。对于目前服务器级别的机器来说，CPU 的配置一般在 16 核以上，完全可以满足 Spark 的运行要求。

3. 网络的配置

这里要特别说明的是，我们不建议将 Spark Cluster 部署到“混合型”的大数

据处理平台上，尽管这种“混合型”的大数据处理平台在本地很多云端都有专属的传输通道，但依然很难满足 Spark“低延迟”的需求。建议将 Spark Cluster 部署在 10GB 及以上网络带宽的局域网中，或者网络中拥有专用的大数据传输设备。

4. 存储的配置

从目前的硬件发展来看，存储硬盘的价格越来越低廉，性能越来越高效，借助一些大数据的存储方案，存储已不再是大数据处理的瓶颈所在。虽然 Spark 能够在内存中执行大量的计算，但它仍然需要本地硬盘作为部分数据的存储。Spark 官方推荐为每个节点配置 4 ～ 8 块磁盘，并且不需要配置为 RAID。另外，建议 Spark Cluster 存储本地化，可以通过配置 spark.local.dir 来指定磁盘列表。

7.5 Spark 架构

7.5.1 Spark 生态架构

如前所述，由于 Hadoop 缺少对迭代场景的支持和硬盘存储引起的延迟，让后起之秀 Spark 几乎在过去的一两年间“红透半边天”，Spark 和它的小伙伴们渐趋成熟稳定，配合也越来越默契，已经形成了 Spark Cluster 生态圈。好了，Spark 和他的小伙伴们在后台等待已久，现在通过图 7-1，让我们请 Spark 和它的小伙伴们集体亮相。

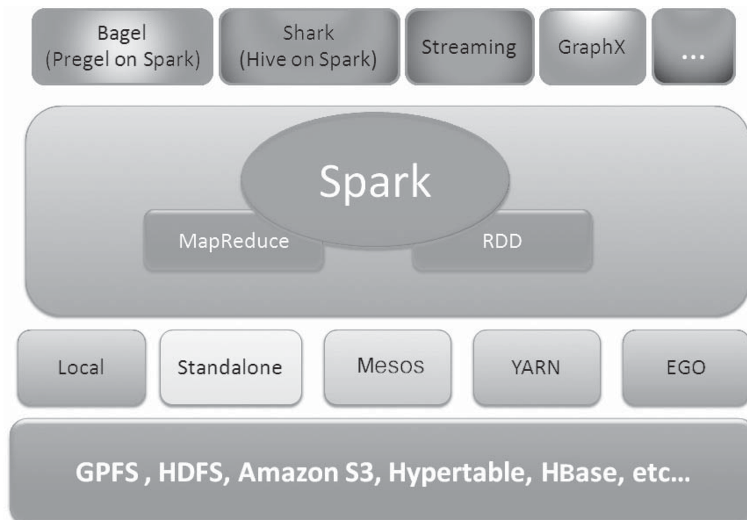

图 7-1 Spark 生态架构

读者可能会发现，有几个成员看着眼熟，比如 Hive、YARN、HDFS 等。是的，它们已经在 Hadoop 阵营中发展壮大，现在也为 Spark 的成长添砖加瓦，贡献自己的力量。我们先对 Spark 家族的主要成员进行简单介绍，后续章节我们再详细阐述。

从整个架构来看，最底层是存储类产品，比如之前详细介绍过的 HDFS、亚马逊的云存储服务 Amazon S3，Spark Cluster 可以通过各种 RESTful API 以编程方式实现与 S3 服务的交互。另外，耳熟能详的 Google 的 Bigtable 是一种高性能的、可伸缩的数据库技术，这里的 Hypertable 和 Bigtable 模型类似，具有高度可伸缩性，并且开源。可伸缩性是目前大数据处理平台越来越看重的技术之一。HBase 在之前详细介绍过，这里不再阐述。Spark 所用的存储类技术可谓遍地开花，不同的云平台提供不同的技术支持，后文我们还会专门讨论 IBM GPFS 技术。

有关 Spark 的运行模式多种多样，依托的资源分配技术也灵活多变。主要的运行模式有 Local、Standalone、Spark on Mesos、Spark on YARN、Spark on EGO 等。

RDD 可以说是 Spark 的“心脏”，也是最主要的创新点。其中文名称为“弹性分布式数据集”，它是一种只读的、分区记录的集合，每次对 RDD 数据集的操作结果都可以存放到内存中，下一个操作可以直接从内存中输入，省去了 MapReduce 大量的磁盘 I/O 操作。

Bagel（Pregel on Spark）：Pregel 是 Google 鼎鼎有名的图计算框架，在此不做过多介绍。将 Pregel 的思想在 Spark 上实现，便产生了 Bagel。这是一个非常有用的小项目。为什么说它小呢？因为 Bagel 最开始仅仅有 200 行代码，但却实现了 Pregel 的功能。除了代码作者能力超群外，还体现出 Spark 超强的融合能力。

Spark SQL：Spark SQL 其实是 Shark 的发展，而 Shark 基本上就是在 Spark 框架的基础上提供与 Hive 一样的 HiveQL 命令接口。随着 Spark 的发展，由于 Shark 对于 Hive 的依赖太强，便停止了 Shark 的开发，由 Spark SQL 来继承 Shark 的功能。Spark SQL 是一个即席查询系统，可以通过 SQL 表达式、HiveQL 或者 Scala DSL 在 Spark 上执行查询。Spark SQL 汲取了 Shark 的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive 兼容性等，重新开发了 Spark SQL 代码。由于摆脱了对 Hive 的依赖，Spark SQL 在数据兼容、性能优化、组件扩展等方面都得到了极大的提升，真可谓“退一步，海阔天空”。

Spark Streaming：构建在 Spark 上处理流数据的框架，其基本原理是将 Stream 数据分成小的时间片段（秒级），以类似 batch 批量处理的方式来处理这一小部分数据。Spark Streaming 构建在 Spark 上，一方面是因为 Spark 的低延迟执行引擎（100ms+）可以用于实时计算；另一方面，相比基于 Record 的其他处理框架（如 Storm），RDD 数据集更容易做高效的容错处理。此外，小批量处理的方式使得它可以同时兼容批量和实时数据处理的逻辑与算法，方便了一些需要历史数据和实时数据联合分析的特定应用场合。

7.5.2 Spark 运行架构

1. Spark 运行的基本术语

我们首先需要搞清楚 Spark 中的几个概念和术语。可参看图 7-2 理解。

- Application：应用，其实就是用户需要完成的应用程序。一般来说，这部分代码需要用户根据自己的需求来完成。这部分代码主要包括两部分：Driver 功能部分和 Executor。那么问题来了:什么是 Driver？什么又是 Executor 呢?
- Driver：顾名思义，驱动者，为运行 Application 准备环境，驱动并监控 Application 运行。Application 中的 main() 函数在运行的时候便创建了一个 SparkContext（通常用 SparkConext 来代表 Driver）。SparkContext 初始化后，向资源管理器（可以是 Standalone、Mesos 或 YARN）注册并申请资源，进行任务的分配和监控等。当各个任务全部运行完成后，Driver 会将 SparkContext 关闭。
- Worker：当 SparkContext 申请到资源后，就会确定此应用程序在 Cluster 的哪些节点中运行。运行 Application 的这些节点就叫 Worker。在并行计算中，一个 Application 中会有多个 Worker 同时工作。
- Executor：顾名思义，执行者，其实就是运行在 Worker 上的一个进程，负责此 Worker 上的 Task 的运行及数据存储（内存或者磁盘）。举例来说，在 Spark on YARN 模式下，其进程名称为 CoarseGrainedExecutorBackend，和 Hadoop MapReduce 中的 YARN Child 类似。CoarseGrainedExecutorBackend 进程负责将 Task 包装成 TaskRunner，使其运行。
- Job：作业。多个并行计算的 Task 的集合，并且包含多个 RDD 及作用于相应 RDD 上的各种操作。

- Stage：阶段。每个 Job 会被拆分为很多组 Task，每组 Task 被称为 Stage，所以 TaskSet 是 Stage 的表现形式。
- Task：任务。它是在 Worker 上执行的最小单元。

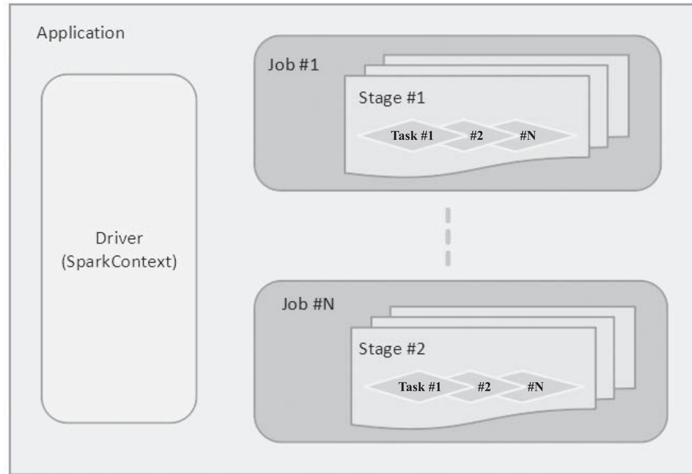

图 7-2　Spark 应用运行要素

2. Spark 运行流程

下面介绍一下 Spark Application 从开始运行到结束的整个过程，以便读者对 Spark 运行架构有一个整体的把握。将运行过程模型化，如图 7-3 所示（图中省略了诸多细节）。

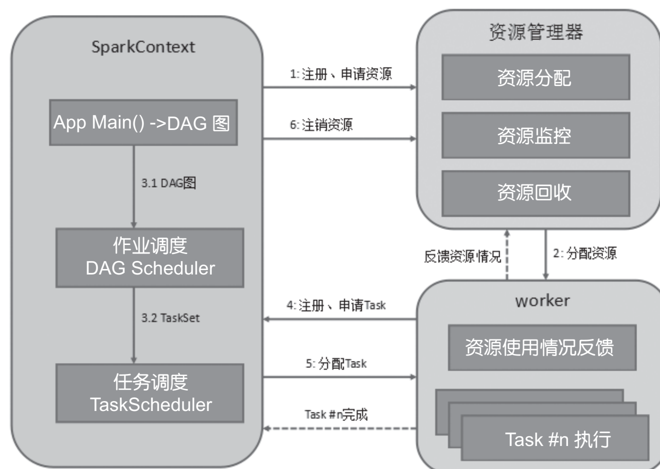

图 7-3　Spark 运行流程图

Spark 具体的运行流程如下：

（1）构建 Spark Application 的运行环境。创建 SparkContext 后，SparkContext 向资源管理器注册并申请资源。这里所说的资源管理器有 Standalone、Messos、YARN 等。事实上，Spark 和资源管理器关系不大，主要是能够获取 Executor 进程，并能保持相互通信。在 SparkContext 初始化过程中，Spark 分别创建作业调度模块 DAGScheduler 和任务调度模块 TaskSchedule（此例为在 Standalone 模式下，在 YARN-Client 模式下任务调度模块为 YarnClientClusterScheduler，在 YARN-Cluster 模式下任务调度模块为 YarnClusterScheduler）。

（2）资源管理器根据预先设定的算法，在资源池里分配合适的 Executor 运行资源。在运行过程中，Executor 运行情况将随着心跳发送到资源管理器上。考虑到 Spark Application 运行过程中 SparkContext 和 Executor 之间有大量的信息交换，提交 SparkContext 的 Client 应该靠近 Worker 节点，以方便信息传输。

（3）SparkContext 构建 DAG 图（Directed Acyclic Graph，有向无环图），作业调度模块 DAGScheduler 将 DAG 图分解成 Stage。DAGScheduler 决定了运行 Task 的理想位置，并把这些信息连同 Task 本身传递给下层的 TaskScheduler。

（4）Executor 向 SparkContext 申请 Task，告诉 SparkContext，“我准备好了，可以干活了，请给我分配任务吧”。

（5）TaskScheduler 维护所有的 TaskSet，当 Driver 收到 Executor 心跳的时候，TaskScheduler 会根据其资源剩余情况分配相应的 Task 到 Executor 运行，同时 SparkContext 将应用程序代码发送给 Worker，随后 Task 便开始在 Worker 上运行。在此期间，TaskScheduler 还维护着所有 Task 的运行状态，重试失败的 Task。如果 Task 失败是因为 Shuffle 数据丢失而引起的，则 DAGScheduler 需要重新提交运行之前的 Stage；如果 Shuffle 数据没有丢失，则交由 TaskScheduler 处理。

（6）当 Task 运行结束后，反馈给 SparkContext，并释放资源。

7.6 小结

本章回顾了 Spark 的诞生和发展过程。Spark 犹如它的名字一样，自诞生以

来，便以星火燎原之势，迅速发展成大数据处理技术中的佼佼者。站在 Hadoop MapReduce 这个“巨人”的肩膀上，扬长避短，使用 RDD 来提升性能，满足交互式分析的需求，同时提升迭代算法的性能，这使得 Spark 非常适合迭代计算场景。总之，Spark 依然是大数据处理中的一个标杆。

8

Spark 核心 RDD

8.1　RDD 简介

8.1.1　什么是 RDD

之前我们已经多次提到 RDD，RDD 是 Spark 的核心成员，是 Spark 的思想精华所在。那么，RDD 到底是什么？它为何如此重要呢？

RDD 的全称为 Resilient Distributed Datasets，也有的资料缩写为 RDDs，它的中文名称是“弹性分布式数据集”。从它的名字中我们可以看到以下三点：

首先，RDD 是一个“数据集”。这个很容易理解。不过值得一提的是，RDD 是只读的、可分区的数据集。

其次，“分布式”的，也就是说，数据可以分布在 Cluster 中的多台机器上进行并行计算。有读者会问，这不就是“分布式共享内存”吗？其实它们还是有区别的，稍后我们会详细分析二者的不同。

最后，“弹性”的。有读者可能在想：“所谓弹性，可大可小，可伸可缩。我见过弹簧是弹性的，数据怎么也有弹性了呢？”在了解 RDD 之初，我们不妨这样理解这个“弹性”：在计算过程中，Spark 会根据内存的大小和使用情况来和磁盘进行数据交换，当内存不足时，它会将特定的部分数据写入磁盘；当数据块

被大量使用时，会将此数据块加载到内存。所以，相对于内存存储来说，数据是“弹性”的。

RDD 本质上是一个只读的分区记录集合，一个 RDD 可以包含多个分区，每个分区就是一个数据集片段。RDD 拥有很多操作，包括生成、转换和执行。

8.1.2 为什么需要 RDD

Hadoop MapReduce 虽然为大数据处理奠定了基础，并且具有自动容错、平衡负载和可扩展等优点，但是其最大的缺点是采用非循环式的数据流模型，使得在迭代计算时要进行反复的磁盘 I/O 操作，导致计算速度下降，使得 MapReduce 的性能大打折扣。

有的读者会想到，大量的磁盘 I/O 操作导致 MapReduce 性能下降，那把计算所需的数据块存储到内存中，问题岂不是解决了？对！这就是 Spark 解决问题的总体思路。

Spark RDD 可以 cache（缓存）到内存中，每次对 RDD 数据集的操作结果都可以存放到内存中，下一个操作可以直接从内存中输入，省去了 MapReduce 大量的磁盘 I/O 操作。这对于迭代运算比较常见的机器学习算法、交互式数据挖掘来说，效率提升比较大。这也是 Spark 存在的价值和发展迅速的理由。

8.1.3 RDD 本体的设计

读过《西游记》的人一定不会忘记唐僧的自我介绍：“贫僧自东土大唐而来，前往西天拜佛求经……”这大概是世界上最简明扼要的自我介绍了，他明确地说了“我是谁（贫僧）”、“从哪里来”、“到哪里去”、“要干什么”（下文我们改为“怎么来”）。我们让 Spark RDD 按照唐僧的思路来做个自我介绍。

从设计思路上看，每个 RDD 都要包含以下几部分。

（1）“我是谁”：数据集，包含一组数据分片（Partition，即数据集的基本组成单位）。

（2）“从哪里来”：关于 Lineage（血统）的信息，源码中的 Dependencies。我们可以简单地理解为它的“父母”是谁。这里需要说明的是，不一定每个 RDD 都有父 RDD。RDD 的生成有两种方式：一是父 RDD 通过 map 之类的函数转换而来；

二是从文件系统（比如 HDFS）输入创建，它是没有父 RDD 的，它的计算函数只是读取文件的每一行并作为一个元素返回给 RDD。

（3）“怎么来”：计算每个分片的函数（该 RDD 分片如何从父 RDD 分片中计算得到）。我们可以形象地理解为父 RDD 如何“制造”此 RDD。

（4）“到哪里去”：每个数据分片的预定义地址列表。包括存储每个 Partition 的优先位置（源码中的 preferredLocations），以及对于 key-value 的 RDD 的 Partitioner，或者对于一个 HDFS 文件来说，存储每个 Partition 所在的块的位置。此项为可选项。

8.1.4 RDD 与分布式共享内存

分布式共享内存（Distributed Shared Memory，DSM）是一种通用的内存数据抽象，使不同的机器无须共享物理内存就可以访问数据，进程访问 DSM 中的数据就如同访问本机的内存一样。

分布式共享内存和 RDD 相似的地方在于都将数据分布式地存储在内存中。二者的主要区别在于以下几点：

（1）RDD 可以批量操作（读操作或者转换创建），很多函数操作（如 count、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；而 DSM 只能通过细粒度的读操作和写操作。

（2）从容错机制来看，RDD 更容易实现而且快速（后面详细讨论）；而 DSM 则需要程序回滚（RollBack）。

（3）从一致性保护角度来看，RDD 无须此类保护机制，原因在于 RDD 是不可变的；而 DSM 需要保证在不同机器上执行的进程可以观察到他人对 DSM 的修改。

（4）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 / 写同一内存位置的数据。

8.2 RDD 的存储级别

每个 RDD 可以使用不同的存储级别，可以把这个数据集存储在硬盘上，或者

以序列化的 Java 对象的形式存储在内存中，或者在多个节点上进行备份。RDD 根据“是否使用内存”、“是否使用硬盘”、“序列化”、“备份数”4 个参考项组合提供了 11 种存储级别（org.apche.spark.storage.StorageLevel），如表 8-1 所示。

表 8-1　RDD 存储级别列表

序　号	存储级别	参数	说明
1	NONE	StorageLevel(false, false, false)	
2	DISK_ONLY	StorageLevel(true, false, false)	以反序列化 Java 对象的形式存储在磁盘上
3	DISK_ONLY_2	StorageLevel(true, false, false, 2)	级别同上，但备份 2 份
4	MEMORY_ONLY	StorageLevel(false, true, true)	以反序列化 Java 对象的形式存储在内存，如果 RDD 太大不能完全放在内存，则多余的一些划分块不会被存储，会在需要时现生成
5	MEMORY_ONLY_2	StorageLevel(false, true, true, 2)	级别同上，但备份 2 份
6	MEMORY_ONLY_SER	StorageLevel(false, true, false)	以序列化的 Java 对象的形式存储在内存，这样比反序列化 Java 对象节省空间，但是耗费 CPU
7	MEMORY_ONLY_SER_2	StorageLevel(false, true, false, 2)	级别同上，但备份 2 份
8	MEMORY_AND_DISK	StorageLevel(true, true, true)	以反序列化 Java 对象的形式存储在内存，如果 RDD 太大不能完全放在内存，则把多余的划分块存储在硬盘上
9	MEMORY_AND_DISK_2	StorageLevel(true, true, true, 2)	级别同上，但备份 2 份
10	MEMORY_AND_DISK_SER	StorageLevel(true, true, false)	顾名思义，以序列化 Java 对象的形式存储在内存，多余的块存储在硬盘上
11	MEMORY_AND_DISK_SER_2	StorageLevel(true, true, false, 2)	级别同上，但备份 2 份

通过向 persist() 方法传递 org.apche.spark.storage.StorageLevel 来选择存储级别。调用的函数 persist() 如下：

```
private def persist(newLevel: StorageLevel, allowOverride: Boolean):
this.type = {
  if (storageLevel != StorageLevel.NONE && newLevel != storageLevel &&
!allowOverride) {
```

```
    throw new UnsupportedOperationException(
      "Cannot change storage level of an RDD after it was already assigned
a level")
   }
   if (storageLevel == StorageLevel.NONE) {
    sc.cleaner.foreach(_.registerRDDForCleanup(this))
    sc.persistRDD(this)
   }
   storageLevel = newLevel
   this
  }
```

默认的存储类别为 MEMORY_ONLY。

```
def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)
```

在存储类别为 MEMORY_ONLY 的情况下，CPU 的效率最高，从速度上来说也最有利于 RDD 的操作。一般来讲，内存使用率越大，CPU 效率越高。在选择存储类别时，应该兼顾二者。在非特殊情况下，非计算数据集的代价很高或者生成该数据集会过滤掉很多数据，否则不建议把数据存储在硬盘上，因为重新计算一个划分块比从硬盘中读取要快很多。当需要快速容错时，则使用带备份的存储级别。

RDD 的 cache() 方法类似于 MEMORY_ONLY，其实调用了 persist() 方法。

```
def cache(): this.type = persist()
```

8.3 RDD 依赖与容错

8.3.1 RDD 依赖关系

RDD 的容错能力很强，相对来说开销不大。在将 RDD 容错之前，我们需要明确 RDD 的依赖关系。通过依赖关系，我们可以看出 RDD 的容错能力是天生的，因为 RDD 容错很大程度上是从它和父 RDD 的关系中恢复的。

RDD 的依赖关系主要包括两种：窄依赖（Narrow Dependency）和宽依赖（Wide Dependency）。

窄依赖是指父 RDD 的每个分区最多被一个子 RDD 的分区所用。注意这里

的主体不是 RDD，而是 RDD 对应的 Partition（分区），而且这种依赖关系是不需要 Shuffle 的。在这类依赖中，可以根据 getParents() 方法获取某个 Partition 的父 Partitions。

```
@DeveloperApi
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency[T] {
 /**
  *为子分区获得相关的父分区
  * @参数 partitionId 代表子RDD的分区
  * 返回子RDD依赖的所有父RDD的分区
  */
 def getParents(partitionId: Int): Seq[Int]
 override def rdd: RDD[T] = _rdd
}
```

窄依赖有两种表现形式。

表现形式 1：表现为一个父 RDD 的分区对应一个子 RDD 的分区（OneToOneDependency），一个“上线”（父 RDD 的分区）对应一个“下线”（子 RDD 的分区）。例如，图 8-1 中的 map/filter 和 union 属于第一类。

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

表现形式 2：表现为两个父 RDD 的分区对应一个子 RDD 的分区（RangeDependency），两个“上线”对应一个“下线”。例如，图 8-1 中对输入进行协同划分（co-partitioned）的 Join 属于第二类。

```
@DeveloperApi
class RangeDependency[T](rdd: RDD[T], inStart: Int, outStart: Int, length: Int)
 extends NarrowDependency[T](rdd) {
 override def getParents(partitionId: Int): List[Int] = {
  if (partitionId >= outStart && partitionId < outStart + length) {
   List(partitionId - outStart + inStart)
  } else {
   Nil
  }
 }
}
```

窄依赖示例如图 8-1 所示。

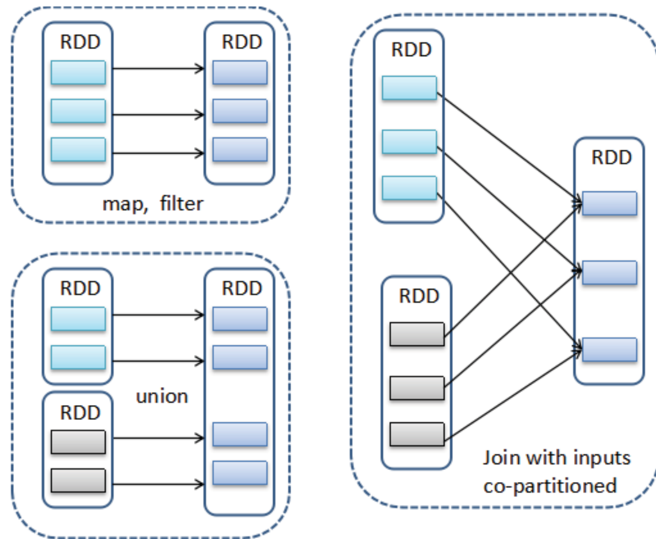

图 8-1 窄依赖示例

宽依赖是指子 RDD 的分区依赖父 RDD 的所有分区，这是因为 Shuffle 类操作，如图 8-2 中的 groupByKey 和未经协同划分的 Join。由于需要对父 Partition 进行划分，故需要用到 Shuffle，而 Shuffle 一般采用键值对的形式。宽依赖示例如图 8-2 所示。

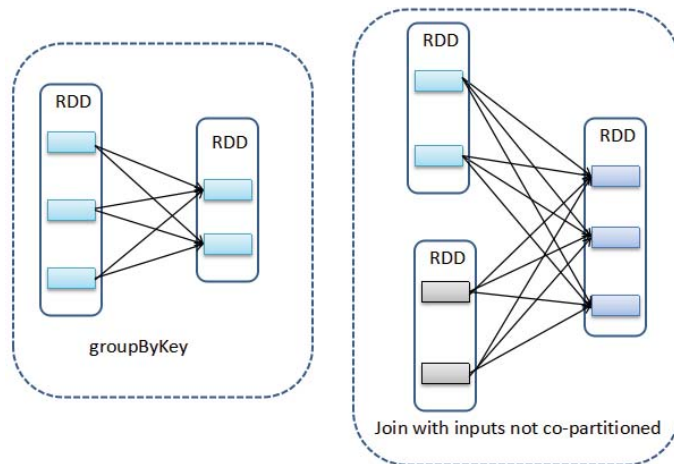

图 8-2 宽依赖示例

这里为每个 Shuffle 分配了一个全局唯一的变量 ShuffleId。为了进行 Shuffle，需要指定如何进行 Shuffle，这对应于参数 Partitioner；由于 Shuffle 是需要网络传输的，故需要进行序列化 Serializer。

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

 override def rdd: RDD[Product2[K, V]] = _rdd.asInstanceOf[RDD[Product2[K, V]]]

 val shuffleId: Int = _rdd.context.newShuffleId()

 val shuffleHandle: ShuffleHandle = _rdd.context.env.shuffleManager.
registerShuffle(shuffleId, _rdd.partitions.size, this)

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

8.3.2 RDD 容错机制

基于以上 RDD 的依赖关系，RDD 设计了两种容错机制：Lineage 机制和 Checkpoint 机制。在容错过程中，这两种机制是相互配合来发挥作用的。

传统的容错方法就是冗余复制存储，冗余系数越大，容错能力越强。一块数据出错后，从另一份冗余存储复制过来。但是这样的存储有一个很大的问题：浪费存储空间，且成本太高。Spark 将计算中间结果存储到内存中，如果采用传统的容错方法，必将是内存的一大挑战。

回忆 8.3.1 节的 RDD 依赖关系，不难发现，RDD 有着天生的容错机制。首先，它自身是一个只读的数据集；其次，只需要记住构建它的操作图。因此，当执行任务的 Worker 失败时，完全可以根据父 RDD，按照之前计算的操作图获得之前执行的转换函数（比如 Map 变换）进行重新计算。由于无须采用冗余复制的方式支持容错，有效地降低了跨网络的数据传输成本，在容错速度和容错可靠度方面都有很大的优势。这种方法被称为“Lineage 机制”，只需要记录父 RDD 和它的操作图。

不过，在某些场景下，Spark 也需要利用 Checkpoint 的方式来支持容错。为什么呢？在窄依赖中，错误的子 RDD 可以直接通过计算父 RDD 的某个数据分

区来获得。但是宽依赖则要等到父 RDD 所有涉及的数据块都计算完成，并且父 RDD 的计算结果进行 Hash 计算并传递到对应节点上后才能计算子 RDD。比如 groupByKey 变换，子 RDD 中的数据块会依赖于多个父 RDD 中的数据分区，因为一个 key 可能错在父 RDD 的任何一个数据分区中。那么对于有宽依赖的并且长 Lineage 链的 RDD 来说，这种恢复非常耗时，适当地设置 Checkpoint 是很有必要的。另外，在 Spark Streaming 中，针对数据进行 update 操作，或者调用 Streaming 提供的 window 操作时，需要恢复执行过程的中间状态。此时，需要通过 Spark 提供的 Checkpoint 机制，以支持操作能够从 Checkpoint 中得到恢复。

如果看重容错的话，则可以使用带备份的存储类别（如 DISK_ONLY_2、MEMORY_ONLY_SER_2、MEMORY_AND_DISK_2）。虽然所有的存储级别都可以通过重新计算丢失的数据来达到容错，但是备份可以使得系统无须重新计算就能继续在 RDD 上执行 Task，效率显著提高。

8.4 RDD 操作与接口

RDD 操作主要有两大类：转换（Transformation）操作和行动（Action）操作。那么，什么是转换操作和行动操作？二者之间有什么关系呢？

我们举生活中做菜的例子来说说。做菜主要有两个环节：备菜（包括洗菜、切菜）和炒菜。在备菜阶段，只是使菜在形态上做出了一些改变，但本质是没有变化的，还是“生”菜。这个阶段的操作类似于 RDD 的 Transformation 操作。Transformation 是从新的数据集创建一个新的数据集。比如 Map 就是一种转换，它将数据集的每一个元素都传递给函数，并返回一个新的分布数据集表示结果。

而炒菜动作类似于 RDD 的 Action 操作。此阶段菜在本质上发生了变化，不再是生菜，而是美味的食品。在 RDD 经过 Action 操作之后，不再是数据集，而是一个值，并将其返回给驱动程序。比如 Reduce 是一种动作，通过一些函数将所有元素叠加起来，并将最终结果返回给 Driver 程序（不过也有个例，并行的 reduceByKey 能返回一个分布式数据集）。

需要特别说明的是，RDD 的 Transformation 操作是有“惰性”的，即不立即执行 Transformation 操作，需要 Action 操作来触发。好比我们不会无缘无故地去备菜，只有当炒菜这个动作来触发的时候，才会执行。RDD 的这种设计可以节省

很多无用的行为，可以更加高效地运行。驱动程序最终关注的是 Action 操作之后的结果（值），而不是中间数据集。例如，通过 map() 函数转换而来的新数据集（中间 RDD）将在 Count() 中使用。这个过程最终只返回 Count() 的结果给驱动程序，而不是整个过程数据集。

8.4.1 RDD Transformation 操作与接口

以下列举出源码中常用的 Transformation 操作，并对每个常用操作的功能进行简要说明。

- map 操作。

```
def map[U: ClassTag](f: T => U): RDD[U]
```

map 操作将 RDD[T] 中的每个元素进行函数运算、映射，生成一个新的 RDD[U]。

- filter 操作。

```
def filter(f: T => Boolean): RDD[T]
```

filter 操作将返回一个 RDD，包含了所有满足过滤条件的元素。

- distinct 操作。

```
def distinct(): RDD[T]
```

distinct 操作返回 RDD 中所有相异的元素，排除相同的元素。经过 distinct 操作之后，每个数据将只会出现一次。

- flatMap 操作。

```
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U]
```

flatMap 操作是将 RDD[T] 中的每个元素进行一对多转换，一个 T 可以映射成 0 到多个 U，两个 RDD 元素通常不等。

- sample 操作。

```
def sample(withReplacement: Boolean, fraction: Double, seed: Long =
Utils.random.nextLong): RDD[T]
```

sample 操作从数据中抽取一定比例的数据子集。

- union 操作。

```
def union(other: RDD[T]): RDD[T]
```

union 操作将两个 RDD[T] 合并成一个 RDD[T]。如果有相同的元素，并不会被排除，而是会多次显示（在实际操作中，可以使用 distinct() 函数将相同的元素排异）。

- coalesce 操作。

```
def coalesce(numPartitions: Int, shuffle: Boolean = false)(implicit
ord: Ordering[T] = null)
```

coalesce 操作对 RDD 的分区函数进行重新划分。

- repartition 操作。

```
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]
```

repartition 操作是 coalesce 接口中 shuffle 参数为 true 时的实现。

- mapPartitions 操作。

```
def mapPartitions[U: ClassTag](f: Iterator[T] => Iterator[U],
preservesPartitioning: Boolean = false): RDD[U]
```

mapPartitions 操作是 Partition 级的转换，将每个 Partition 经过转换之后形成新的 RDD。

- groupByKey 操作。

```
def groupByKey(): RDD[(K, Iterable[V])]
```

groupByKey 操作将 RDD[(K,V)] 中所有的 (K,V) 键值对按 K 进行分组，每组一个元素，形成新的 RDD[(K, Iterable[V])]。

- reduceByKey 操作。

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

在一个 (RDD[(K,V)] 中所有 (K,V) 键值对的数据集上使用，返回一个 (K,V) 键值对的 RDD，key 相同的值都被使用指定的 reduce() 函数聚合到一起。和 groupByKey 类似，任务的个数是可以通过可选参数来配置的。

- join 操作。

```
def join[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))]
```

将 RDD[(K,V)] 与 RDD[(K,W)] 执行 join 操作，形成新的 RDD[(K,(V,W))]，最终形成的 RDD 中只有两个 RDD 中共有的 K。

- mapValues 操作。

```
def mapValues[U](f: V => U): RDD[(K, U)]
```

仅对 RDD[(K,V)] 中的 V 进行转换，转换后和原先的 K 一起形成新的 (K,U) 键值对，通常和 groupByKey 一起使用。

- partitionBy 操作。

```
def partitionBy(partitioner: Partitioner): RDD[(K, V)]
```

partitionBy 操作将 RDD[(K,V)] 按照 K 进行重新分区。

8.4.2 RDD Action 操作与接口

- reduce 操作。

```
def reduce(f: (T, T) => T): T
```

reduce 操作通过函数 f 对 RDD[T] 中的元素进行两两合并，最终合并成一个值。这个函数必须是关联性的，确保可以被正确地并发执行。

- collect 操作。

```
def collect(): Array[T]
```

collect 操作以数组的形式返回数据集中的所有元素。

- count 操作。

```
def count(): Long
```

count 操作返回数据集的元素个数。

- take 操作。

```
def take(num: Int): Array[T]
```

take 操作返回一个包含数据集中前 *n* 个元素的数组。此操作目前并非在多个节点上并行执行，而是由 Driver 程序所在的机器单机计算所有的元素。

- first 操作。

```
def first(): T
```

first 操作返回数据集的第一个元素（类似于 take (1)）。

- saveAsTextFile 操作。

```
def saveAsTextFile(path: String): Unit
```

saveAsTextFile 操作将 RDD 的元素以 TextFile 的形式保存到分布式文件系统

（比如 HDFS 或者其他文件系统）。Spark 会调用 toString() 方法来把每个元素存成文本文件的一行。此方法最大的好处就是解决了通过 collect 到 Driver 再保存到磁盘的问题。

- saveAsSequenceFile 操作。

```
def saveAsSequenceFile[K, V, C <: CompressionCodec](
      pyRDD: JavaRDD[Array[Byte]],
      batchSerialized: Boolean,
      path: String,
      compressionCodecClass: String): Unit
```

将 RDD 的元素以 SequenceFile 的形式保存到分布式文件系统的指定目录（比如 HDFS 或者其他文件系统）。

- foreach(func) 操作。

```
def foreach(f: T => Unit): Unit
```

foreach(func) 操作对数据集中的每个元素逐个运行预定义的函数，比如用于更新一个累加器的变量。

8.5 RDD 编程示例

下面通过一个示例来说明 Transformation 与 Actions 操作在 Spark 中的使用。图 8-3 所示为将数据从文件系统中读入到处理完毕的 RDD 示例图（注：为了便于理解，省略了其他环节）。

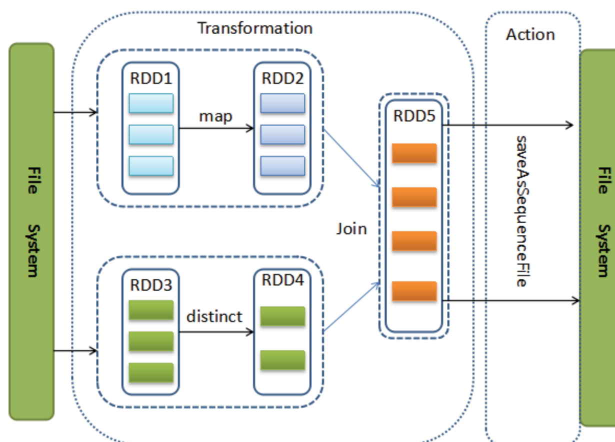

图 8-3 RDD 编程示例

```
val sourceData = new SparkContext(master, "Example", System.getenv("SPARK_
HOME"), Seq(System.getenv("SPARK_EXAMPLE_JAR")))

val rdd1 = sourceData.textFile(hdfs://path/to/data1)
val rdd2 = sourceData.textFile(hdfs://path/to/data2)

val rdd3 = rdd1.map(line => (line.substring(10), 1))
val rdd4 = rdd2.distinct()
val rdd5 = rdd4.join(rdd_5)

rdd5.saveAsSequenceFile(hdfs://save/as/path)
```

8.6 小结

RDD 作为 Spark 的核心，充分体现了 Spark 的主要设计思想，很大程度上解决了 Hadoop MapReduce 的不足之处。本章讲述了 RDD 的本体设计，阐述了 RDD 与传统的分布式共享内存技术的区别，介绍了 RDD 的永久性存储技术。本章重点讲述了 RDD 的依赖关系，以及从这种依赖关系中衍生出的容错机制。RDD 大体上有两种操作，即转换操作和行动操作，并列举了常用的操作和接口。总之，要想掌握 Spark 的精华，就必须先理解 RDD。