前言

欢迎阅读《Spark权威指南》！ 我们很高兴出版这本书，这是目前为止关于Apache Spark最全面的学习资源，本书特别关注Spark 2.0中引入的新一代Spark API。

Apache Spark是目前最流行的大规模数据处理系统之一，提供支持多种编程语言的API，并且具有大量内置库和第三方库的支持。在2009年，Spark项目是在加州大学伯克利分校开始的一个研究项目，然后自2013年起发布到Apache开源社区上。该项目已经存在了很多年，开源社区持续在Spark上构建了更强大的API和高级库，所以关于这个项目还有很多需要写的东西。我们决定写这本书有两个主要原因：首先，我们希望提供关于Apache Spark的最全面的书籍，其中包含了大量基本用例以及易于运行的示例；其次，我们重点介绍了Apache Spark 2.0中定义的高级"结构化”API，即DataFrame、Dataset、Spark SQL和Structured Streaming（结构化流处理），而以前关于Spark的书籍并不一定包含这些新API。我们希望本书可以帮助你使用最新的Spark API编写Apache Spark应用程序。

在这里，我们会告诉你一些背景知识，并解释本书适合的读者，以及我们是如何组织本书内容的。我们还要感谢帮助我们编辑和审阅的很多人，如果没有他们，是不可能完成本书的。

关于作者

本书的两位作者都参与了Apache Spark项目很长时间，所以我们很高兴能够为你带来这本书。

Bill Chambers于2014年开始在几个研究项目中使用Spark。目前，Bill是Databricks的产品经理，帮助用户编写各种类型的Apache Spark应用程序。Bill还定期发布关于Spark的博客，并在有关该主题的会议和小型聚会上发表演讲。Bill拥有加州大学伯克利分校信息管理与系统硕士学位。

Matei Zaharia于2009年在加州大学伯克利分校攻读博士学位期间创建了Spark项目。Matei与伯克利分校的其他研究人员和几名外校合作者一起设计了Spark 的核心API，并发展Spark社区，持续参与了诸如结构化API和结构化流处理等新功能的扩展工作。2013年，Matei和加州大学伯克利分校Spark团队的其他成员共同创立了Databricks公司，进一步发展开源项目并提供相关的商业化服务。现在，Matei一直担任Databricks的首席技术官，同时还是斯坦福大学计算机系的助理教授，从事大规模系统和人工智能的研究。Matei于2013年获得加州大学伯克利分校计算机系的博士学位。

本书的读者对象

本书适合希望使用Apache Spark的数据科学家和数据工程师阅读。这两个角色的需求稍有不同，但实际上，大多数应用程序开发的过程都涵盖了这两个角色，所以本书对两者都有用。具体而言，数据科学家的工作更侧重于交互式查询数据以回答问题并建立统计模型，而数据工程师的工作重点在于编写易于维护、易于复用的应用程序，他们要么在实践中使用数据科学家的模型，要么只是准备用于进一步分析的数据（例如，构建数据摄取管道）。但是，在使用Spark方面，我们经常会看到这两类角色的分工并不明确。例如，数据科学家需要能够在没有太多麻烦的情况下开发应用程序，数据工程师需要使用交互式分析来了解和检查其数据，进而构建和维护数据处理管道。

虽然我们试图提供数据科学家和工程师需要的所有内容，但本书中有些内容我们没有深入阐述。首先，本书不包含对Apache Spark中的某些分析技术的深入介绍，例如机器学习。我们假设你已经掌握了机器学习的基本知识，所以本书仅向你展示如何在Spark中使用已有的库来调用这些技术。许多关于机器学习的专业书籍详细地介绍了这些技术，因此如果你想深入了解这些领域，我们建议你首先阅读这些专业书籍。其次，本书更侧重于应用程序开发而不是操作和管理（例如，如何管理有数十个用户的Apache Spark集群）。尽管如此，我们试图在本书的第五部分和第六部分中介绍有关监控、调试和配置的全面资料，以帮助工程师高效地运行其应用程序，并解决日常维护遇到的问题。最后，本书并没有详细介绍Spark中例如RDD和Dstream这些较旧和较低级别的API，而是更多地基于较新的更高级别的结构化API来介绍大多数概念。因

此，如果你需要维护旧的基于RDD或Dstream的应用程序，本书可能不是最合适的，但是如果你要编写新应用程序，本书绝对是非常好的选择。

本书约定

本书使用了以下排版约定：

斜体（Italic）
: 表示新术语、URL、电子邮件地址、文件名、文件扩展名。

等宽字体（`Constant width`）
: 表示程序片段，以及段落内的程序代码，例如变量名、函数名、数据库、数据类型、环境变量、语句、关键字等。

等宽粗体（**`Constant width bold`**）
: 表示需要用户逐字输入的命令或其他文本。

等宽斜体（*`Constant width italic`*）
: 表示需要用户提供的值或根据上下文确定的值替换的文本。

表示提示或建议。

表示一般性的说明。

表示警告或提醒。

使用代码示例

本书中的代码都可以在真实数据集上运行，我们使用Databricks Notebook编写了整本书，并在GitHub（*https://github.com/databricks/Spark-The-Definitive-Guide*）上发布了

相关数据集和相关资料，你可以随时运行和编辑所有本书中涉及的代码，或者将其复制到你自己的应用程序代码中，我们尽可能使用真实数据来说明在构建大规模数据处理应用时遇到的挑战。最后，我们还在本书的GitHub存储库中包含了几个较大的独立应用程序，这些应用程序并不依赖于本书内容。

当Spark功能更新的时候，GitHub存储库也将相应地更新，所以请一定要记得从GitHub上下载新的资料。

本书是为了帮助你完成工作，一般来说，你可以在程序和文档中使用本书中涉及的示例代码。除非你要复制大部分代码，否则你无需联系我们获得许可。例如，使用本书中几个示例代码块的内容不需要许可。销售或分发来自O'Reilly书籍的示例CD光盘需要许可，引用本书内容并引用示例代码来回答问题并不需要许可，将本书中的大量示例代码整合到产品文档中需要获得许可。

我们感激你在引用本书内容时做引用说明，但我们不强制。引用说明通常包括书名，作者，出版社和ISBN信息。例如：“Spark: The Definitive Guide by Bill Chambers and Matei Zaharia (O'Reilly). Copyright 2018 Databricks, Inc.,978-1-491-91221-8”。

如果你不确定是否超出了示例代码的合理使用范围或许可范围，请随时通过与我们联系*permissions@oreilly.com*。

O'Reilly Safari

Safari（之前叫Safari Books Online）是一个会员制的培训与参考平台，供企业、政府、教师及个人使用。

会员可以访问上千种图书、培训视频、学习计划、互动教程及精选内容，这些内容涵盖 250 家出版商，其中包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 及 Course Technology 等。

更多信息，请访问：*http://oreilly.com/safari*。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

对于本书的评论和技术性问题，请发送电子邮件到：*bookquestions@oreilly.com*。

如果要发表评论或询问技术问题，请发邮件给 *bookquestions@oreilly.com*。

其他书籍、课程、会议及新闻，请访问网站：http://www.oreilly.com。

我们的Facebook是：*http://facebook.com/oreilly*。

我们的Twitter是：https://twitter.com/oreillymedia。

我们的YouTube是：http://www.youtube.com/oreillymedia。

致谢

我们要感谢很多人。

首先，我们要感谢我们的老板Databricks为支持我们写作本书而分配了工作时间。如果没有公司的支持，我们不可能完成本书。我们特别感谢Ali Ghodsi、Ion Stoica和Patrick Wendell的支持。

此外，很多人阅读了本书或者个别章节的草稿，并提供了宝贵的反馈意见，他们是最好的审稿人。

这些审稿人按姓氏的字母顺序排列如下：

- Lynn Armstrong
- Mikio Braun
- Jules Damji
- Denny Lee

- Alex Thomas

除了正式的审稿人之外，还有许多其他的Spark用户、贡献者和提交者阅读了某些章节或帮助制订了修改计划。这些帮助本书写作的人按姓氏字母顺序排列如下：

- Sameer Agarwal
- Bagrat Amirbekian
- Michael Armbrust
- Joseph Bradley
- Tathagata Das
- Hossein Falaki
- Wenchen Fan
- Sue Ann Hong
- Yin Huai
- Tim Hunter
- Xiao Li
- Cheng Lian
- Xiangrui Meng
- Kris Mok
- Josh Rosen
- Srinath Shankar
- Takuya Ueshin
- Herman van Hovell
- Reynold Xin
- Philip Yang
- Burak Yavuz
- Shixiong Zhu

最后，我们要感谢朋友、家人和亲人。没有他们的支持、耐心和鼓励，我们不可能完成这本书。

第 I 部分

大数据与Spark概述

第1章

Spark是什么?

Apache Spark是一个在集群上运行的统一计算引擎以及一组并行数据处理软件库。Spark是目前最流行的开源大数据处理引擎，是所有对大数据感兴趣的开发人员和数据科学家的标配工具。Spark支持多种常用的编程语言（Python，Java，Scala和R），提供支持SQL、流处理、机器学习等多种任务的软件库，它既可以在笔记本计算机上运行，也可以在数千台的服务器组成的集群上运行。这使得它成为一个既适合初学者的简单系统，也适合处理大数据的大规模系统，甚至扩展到惊人的超大规模系统。

图1-1 展示了Spark向终端用户提供的所有组件和软件库。

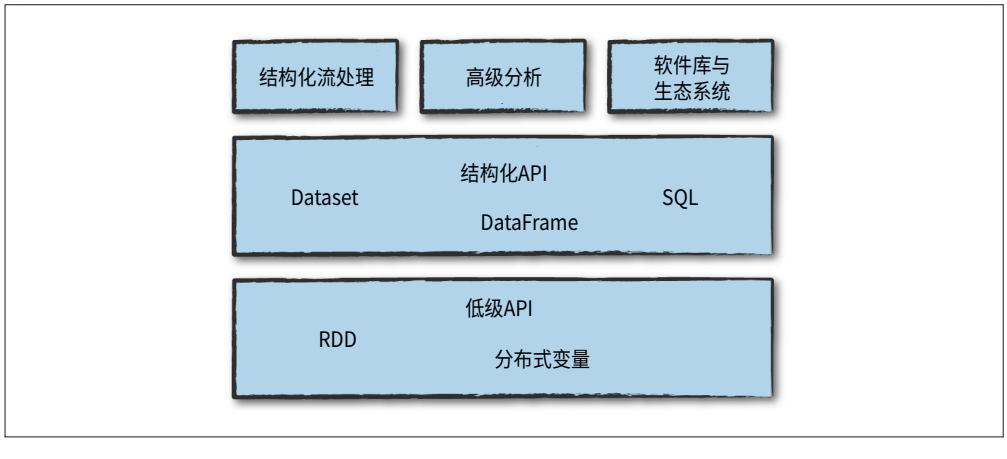


图1-1：Spark工具包

图1-1中所示的多个组件大致对应于本书的不同部分，我们想让你了解Spark的各个方面，并让你知道Spark确实是由许多不同的组件所组成。

鉴于你正在阅读这本书，你可能已经对Apache Spark的功能有所了解并知道它可以做什么。尽管如此，在本章中，我们将简要介绍Spark背后的重要哲学，以及它出现的背景（为什么每个人都热衷于并行数据处理）及其历史。我们还将概述运行Spark的前几个步骤。

Apache Spark的设计哲学

Apache Spark作为统一计算引擎和大数据处理软件库的集成，我们来细分一下它的关键组件：

统一平台

Spark的主要目标是为编写大数据应用程序提供统一的平台，统一是什么意思呢？ Spark通过统一计算引擎和利用一套统一的API，支持广泛的数据分析任务，从简单的数据加载，到SQL查询，再到机器学习和流式计算。这一目标背后的驱动原因是，真实世界的数据分析任务结合了许多不同的处理类型和软件库，不论是Jupyter notebook这种交互式分析工具，还是用于生产应用的传统软件开发。

Spark的统一API使得这些任务更易编写且更加高效。首先，Spark提供了一致的、可组合的API，你可以使用这些API来构建应用程序，或使用代码片段亦或是从现有的库来构建应用程序，它还允许你在上面编写自己的数据分析库。但是，可组合的API还不够，利用Spark的API，用户还可以组合不同库和函数来优化用户程序，从而实现高性能。例如，如果你使用SQL查询语句来加载数据，然后使用Spark的ML库评估其上的机器学习模型，则引擎可以将这些步骤合并为一次数据扫描。通用API设计和高性能执行的设计，使Spark成为开发交互式程序和生产应用程序的强大平台。

Spark在定义统一平台方面与软件其他领域实现统一平台的思路相同。例如，数据科学家在进行建模时受益于统一的软件库支持（例如Python或R），Web开发人员可从统一框架（如Node.js或Django）中受益。在Spark之前，没有任何开源系统试图为并行数据处理提供这种统一引擎，这意味着用户必须拼凑多套API和多个系统来开发应用程序。因此，Spark迅速成为这种发展的标准，随着时间的推移，Spark不断扩展其内置API以涵盖更多应用领域，同时该项目的开发人员也在不断完善统一引擎。值得一提的是，本书的重点之一是在Spark 2.0中定义的

“结构化API”（DataFrame，Dataset和SQL），以帮助进一步优化用户应用程序。

计算引擎

在Spark致力于统一平台的同时，它也专注于计算引擎，Spark从存储系统加载数据并对其执行计算，加载结束时不负责永久存储。你可以将多重持久化存储系统与Spark结合使用，包括云存储系统（如Azure存储和Amazon S3）、分布式文件系统（如Apache Hadoop）、键值存储系统（如Apache Cassandra），以及消息队列系统（如Apache Kafka）。但是，Spark本身既不负责持久化数据，也不偏向于使用某一特定的存储系统，主要原因是大多数数据已经存在于混合存储系统中，而移动这些数据的费用非常高，因此Spark专注于对数据执行计算，而不考虑数据存储于何处。Spark努力使这些存储系统让用户使用起来大致相似，这样应用程序无需担心数据存储对数据处理的影响。

Spark对计算的关注使其不同于早期的大数据软件平台，例如Apache Hadoop。Hadoop包括一个存储系统（HDFS，专为使用普通服务器集群进行低成本存储而设计）和计算系统（MapReduce），它们紧密集成在一起。但是，这种设计导致无法运行独立于HDFS的MapReduce系统。更重要的是，这种选择也让用户编写访问其他存储的应用程序更加困难。尽管Spark在Hadoop存储上运行良好，但现在它也广泛用于除Hadoop之外的存储系统，例如公有云（可以单独租赁云存储资源）或流处理应用程序。

配套的软件库

Spark的最后一个组件是它的软件库，这与Spark的设计理念一脉相承，即构建一个统一的引擎，为通用数据分析任务提供统一的API。Spark不仅随计算引擎提供标准库，同时也支持一系列由开源社区发布为第三方包的外部库。今天，Spark的标准库实际上已经成为了一系列开源项目的集成：Spark核心引擎本身自第一次发布以来几乎没有什么变化，但是配套的软件库已经越来越强大，提供越来越多的功能。Spark包括SQL和处理结构化数据的库（Spark SQL）、机器学习库（MLlib）、流处理库（Spark Streaming和较新的结构化流式处理），以及图分析（GraphX）的库。除了这些库之外，还有数百种开源外部库，从用于各种存储系统的连接器到机器学习算法。spark-packages.org（*https://spark-packages.org/*）上提供了一个外部库的索引。

背景：大数据问题

为什么我们迫切需要用于数据分析的新计算引擎和编程模型？与许多计算领域的新趋势一样，这是由于计算机应用和硬件技术背后的商业形势变化所导致的。

在计算机发展的很长一段时间里，计算机每年都会因处理器速度的提高而变得更快：每年新处理器的运行速度可能比上一年更快。因此，在不对其代码进行任何更改的情况下，应用程序每年都会变得更快。这种趋势的持续建立并形成了大型应用程序的生态系统，而其中大部分应用程序只被设计为在单个处理器上运行。随着时间的推移，这些应用程序也促使了处理器速度的提高，并要求支持更多的计算量和处理更大的数据量。

不幸的是，这种硬件发展趋势在2005年左右停止了：由于硬盘散热的限制，硬件开发人员停止了制造更快单个处理器的思路，并转向为增加更多的相同计算能力的并行CPU计算核心。这种变化意味着需要修改应用程序以支持并行计算，以便更快运行，这促使了Apache Spark等新编程模型的出现。

最重要的是，虽然在2005年处理器性能的增速减慢了，但是数据存储和数据采集技术的发展并没有减慢。存储1TB数据的成本每14个月继续下降两次，对于各种规模的企业来说，这意味着存储大量数据的成本非常便宜。此外，许多数据采集技术（传感器、摄像头、公共数据集等）不断发展，成本持续降低或者分辨率持续提高。例如，摄像头技术分辨率在不断提高，每像素的成本年年都在下降，1200万像素的摄像头成本仅为3~4美元；这使得收集大量视频数据的成本非常便宜，不论是来自人们实际生活中的拍摄视频或是自动传感器。此外，摄像头本身还是其他数据采集设备（如望远镜和基因测序仪）的关键传感器，这也促使了这些技术的成本下降。

最终的结果是我们的世界发展成了一个采集数据非常便宜的世界，但处理它需要大规模的并行计算，通常是在群集上进行。而且，在这个新的世界里，过去50年开发的软件无法自动扩展为并行程序，数据处理领域的传统编程模型也无法被自动的并行化，因此我们需要一种新的编程模型。Apache Spark在这种新形势下应运而生。

Spark的历史

Apache Spark是2009年在加州大学伯克利分校进行Spark研究项目，该校AMPLab实验室的Matei Zaharia，Mosharaf Chowdhury，Michael Franklin，Scott Shenker和Ion Stoica于次年发表了题为“Spark：Cluster Computing with Working Sets”（*https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf*）的论文。当时，

Hadoop MapReduce是运行在计算机集群上的主要并行计算引擎，是第一个在数千个节点群集上进行并行数据处理的开源系统。AMPLab曾与多位早期的MapReduce用户合作，以了解这种新编程模型的优缺点，因此能够针对多个实际中的问题，开始设计更通用的计算平台。此外，Zaharia还与加利福尼亚大学伯克利分校的Hadoop用户合作，了解他们对平台的需求，特别是使用迭代算法进行大规模机器学习的团队，他们需要对数据进行多次迭代处理。

在与MapReduce用户的交流中，有两件事很确定。首先，集群计算具有巨大的潜力：在使用MapReduce的每个组织机构中，可以使用现有数据构建全新的应用程序，并且许多研究组在尝试了入门示例后开始使用该系统。其次，MapReduce引擎构建大型应用程序既具有挑战性又低效。例如，典型的机器学习算法可能需要对数据进行10次或20次迭代处理，而在MapReduce中，每次迭代都必须通过一个MapReduce作业来完成，必须在分布式集群上重新读取全部数据并单独启动一次作业。

为了解决这个问题，Spark团队首先设计了一个基于函数式编程的API，可以简洁地表达多计算步骤的应用程序。然后，该团队通过一个新的引擎实现了这个API，该引擎可以跨多个计算步骤执行高效的内存数据共享。该团队还开始与伯克利分校和校外用户一起测试该系统。

Spark的第一个版本仅支持批处理应用程序，但很快又发现了另一类重要的应用：交互式数据处理和即席（ad-hoc）查询。通过简单地将Scala解释器插入Spark，该项目可以提供一个高可用的交互式系统，用于在数百台机器上运行查询。基于这个想法，AMPLab很快的开发了Shark系统，这是一个可以在Spark上运行SQL查询并满足分析师与数据科学家的交互式应用的引擎。Shark于2011年首次发布。

在这些初始版本发布之后，我们很快就清楚地认识到，Spark最强大的功能将来自于新的软件库，因此该项目开始遵循当今的“标准库”方法。特别是，AMPLab中不同的研究组开始开发MLlib、Spark Streaming和GraphX。他们还确保这些API具有高度的互操作性，使人们首次可以在同一个引擎中编写多种端到端的大数据应用程序。

2013年，这个项目已经得到了广泛的应用，来自加州大学伯克利分校以外的30多个组织有超过100个贡献者参与了该项目，AMPLab作为该项目的一个长期的、非商业机构为Apache软件基金会贡献了Spark。早期的AMPLab团队还成立了一家名为Databricks的公司来加强该项目，并加入其他为Spark贡献力量的公司和组织。从那时起，Apache Spark社区在2014年发布了Spark 1.0，在2016年发布了Spark 2.0，并且继续定期发布，为项目带来新功能。

最后，Spark可组合API的核心思想也在不断完善。早期版本的Spark（1.0版之前）很大程度上只定义了API的功能操作（即Java对象集合的map和reduce等并行操作）。从1.0版开始，该项目添加了Spark SQL，这是一种用于处理结构化数据（即具有固定数据格式且与Java的内存表示无关的数据表）的新API。利用对输入数据格式和用户代码的了解，Spark SQL进一步优化了相关软件库和API的性能。之后，该项目增加了大量针对结构化数据的新API，包括DataFrame、机器学习管道和结构化流处理（一个高级别的自动优化的流处理API）。在本书中，我们将用大量的篇幅来解释这些大多已经可被用于生产应用的下一代API。

Spark的现状和未来

Spark已经存在了很多年，其仍然很流行并且不断有新的应用案例出现，Spark生态系统中的许多新项目扩大了Spark可能的应用范围。例如，2016年推出了一种新的高级流处理引擎，即结构化流处理。这项技术帮助解决了很多公司的大数据处理挑战，例如，Uber和Netflix等技术公司使用Spark的结构化流处理技术和机器学习工具，包括美国航空航天局、欧洲核子研究中心、麻省理工学院布罗德研究所和哈佛大学研究所等机构将Spark应用于科学数据分析。

在可预见的将来，鉴于该项目仍处于快速发展期，Spark仍将是企业进行大数据分析的基石，任何需要解决大数据问题的数据科学家或工程师们可能都需要在他们的机器上安装Spark，当然最好这本书也被一起放到他们的书架上！

运行Spark

本书包含大量与Spark相关的代码，你应该一边学习一边尝试运行这些代码。大多数情况下，你需要运行交互式代码，以便对其进行试验。在开始使用本书的代码之前，让我们了解一下可用的编程语言。

你可以使用Python，Java，Scala，R或SQL等语言来运行Spark。Spark本身是用Scala编写的，并且运行在Java虚拟机（JVM）上，因此为了在笔记本计算机或集群上运行Spark，你需要安装Java。如果你想使用Python API，你还需要安装Python（2.7或更高版本）。如果你想使用R语言，你需要在你的机器上安装R语言。

我们推荐两种方法来开始使用Spark：在你的笔记本电脑上下载并安装Apache Spark，或者在Databricks Community Edition（一种用于学习Spark的免费云环境，包括本书中的代码）上运行基于Web的版本。我们接下来分别介绍这两个选项。

本地下载Spark

如果你想在本地下载并运行Spark，第一步是确保你的机器上安装了Java，如果你想使用Python，还需要安装Python。接下来，访问该项目的官方下载页面（*http://spark.apache.org/downloads.html*），选择“Pre-built for Hadoop 2.7 and later”的包类型，然后单击“直接下载”。这将下载一个压缩的TAR文件或tarball，然后你需要解压缩。本书的大部分内容都是使用Spark 2.2编写的，因此应该下载2.2或更高版本。

下载Hadoop集群的Spark

Spark可以在没有任何分布式存储系统（如Apache Hadoop）支持的情况下在本地运行。但是，如果你希望将笔记本计算机上的Spark版本连接到Hadoop集群，请确保你下载支持该Hadoop版本的Spark版本，该版本可以在*http://spark.apache.org/downloads.html*下载。我们将在后面的章节中讨论Spark如何在集群和HDFS上运行，但是现在我们建议你只需在笔记本计算机上运行Spark即可。

在Spark 2.2中，开发人员还增加了通过pip install pyspark来安装支持Python语言的Spark版本的选项。因为是在本书编写时出现的功能，所以我们无法包含相关指示说明。

从源代码构建Spark

你也可以从源代码构建和配置Spark，但是我们在本书中将不做介绍。你可以在Apache下载页面上选择源代码包以获取源代码，并按照README文件中的说明进行构建。

下载完Spark之后，你需要打开命令行终端，然后解压缩包。我们以安装Spark 2.2作为示例，下面是一个代码片断，你可以在任何UNIX风格的命令行终端上运行，解压Spark源码文件并移动到目录中：

```
cd ~/Downloads
tar -xf spark-2.2.0-bin-hadoop2.7.tgz
cd spark-2.2.0-bin-hadoop2.7.tgz
```

请注意，Spark在项目中含有大量的目录和文件，但不要被吓倒，只有在阅读源代码时你才需要了解哪些目录是相关的。下一章将介绍若干重要的目录，这些目录里含有运行不同Spark功能的交互式控制台程序。

启动Spark交互式控制台

Spark支持几种不同的编程语言的交互式控制台，本书的大部分内容都是用Python、Scala和SQL编写的，因此，我们推荐利用交互式控制台来学习Spark。

启动Python控制台

你需要安装Python 2或Python 3才能启动Python控制台。在Spark的主目录下运行以下代码：

```
./bin/pyspark
```

完成之后，输入“spark”并按Enter键，你将看到打印的SparkSession对象，第2章将详细介绍。

启动Scala控制台

要启动Scala控制台，你需要运行以下命令：

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

完成之后，输入“spark”并按Enter键，就像在Python中一样，你会看到SparkSession对象，第2章将详细介绍。

启动SQL控制台

本书的部分内容将涉及大量的Spark SQL，你可能想要启动SQL控制台，在我们接触这本书中的相关主题之后，我们将介绍更多细节。

```
./bin/spark-sql
```

在云平台上运行Spark

如果你想有个更简单的交互式体验来学习Spark，你可能更喜欢使用Databricks Community Edition。如前所述，Databricks是由伯克利团队创立的公司，创立了Spark，并提供免费的基于云服务的Community Edition作为学习环境。 Databricks Community Edition包含了本书的所有数据和代码示例，你可以快速运行。要使用Databricks Edition，请按照*https://github.com/databricks/Spark-The-Definitive-Guide*中的操作说明，你将通过Web界面使用Scala，Python，SQL或R来运行Spark程序，也可以将得到的处理结果可视化。

本书中使用的数据

我们将在本书中使用一些数据集作为示例，如果想在本地运行代码，你可以从*https://github.com/databricks/Spark-The-Definitive-Guide*上下载它们。你需要首先下载数据，然后将其放在一个文件夹中，并运行本书中的代码片段。

第2章

Spark浅析

关于Apache Spark的历史我们介绍完了，现在是开始学习如何使用它的时候了！本章将对Spark的所有功能大致地介绍一下，其中我们将介绍集群计算的核心体系结构和Spark应用程序，并结合DataFrame和SQL介绍Spark结构化API，我们还将介绍Spark的核心术语和概念。首先我们将从Spark的基本背景信息开始。

Spark的基本架构

一般来说，当你想到一台“计算机”时，你会想到你家中或办公室的办公桌上的一台机器，这台机器可以用来看电影或使用电子表格软件，但是许多用户可能会在某个时候体验到，有些工作任务是你的计算机不能胜任的。数据处理就是这样一个特别有挑战性的任务，单台机器没有足够强大的计算能力和计算资源来执行处理这些大量的数据（或者用户没有足够耐心等待计算结束）。一个集群或一组计算机将许多机器的资源集中在一起，使我们能够像使用单台计算机一样使用这些资源。但是如果一群机器没有协调机制，那么这些机器并不能产生强大的计算能力，你需要一个软件框架来协调它们之间的工作。Spark就是这样一种软件框架，它管理和协调跨多台计算机的计算任务。

Spark用来执行计算任务的若干台机器由像Spark的集群管理器、YARN或Mesos这样的集群管理器管理，然后我们提交Spark应用程序给这些集群管理器，它们将计算资源分配给应用程序，以便完成我们的工作。

Spark应用程序

Spark应用程序由一个驱动器进程和一组执行器进程组成。驱动进程运行main()函数，位于集群中的一个节点上，它负责三件事：维护Spark应用程序的相关信息；回应用户的程序或输入；分析任务并分发给若干执行器进行处理。驱动器是必须的，它是Spark应用程序的核心，它在应用程序执行的整个生命周期中维护着所有相关信息。

执行器负责执行驱动器分配给它的实际计算工作，这意味着每个执行器只负责两件事：执行由驱动器分配给它的代码，并将该执行器的计算状态报告给运行驱动器的节点。

图2-1演示了集群管理器如何控制物理机器并为Spark应用程序分配资源，这个集群管理器可以是三个核心集群管理器之中的任意一个：Spark的独立集群管理器、YARN或Mesos。这意味着可以同时在群集上运行多个Spark应用程序，我们将在第四部分更详细地讨论集群管理器。

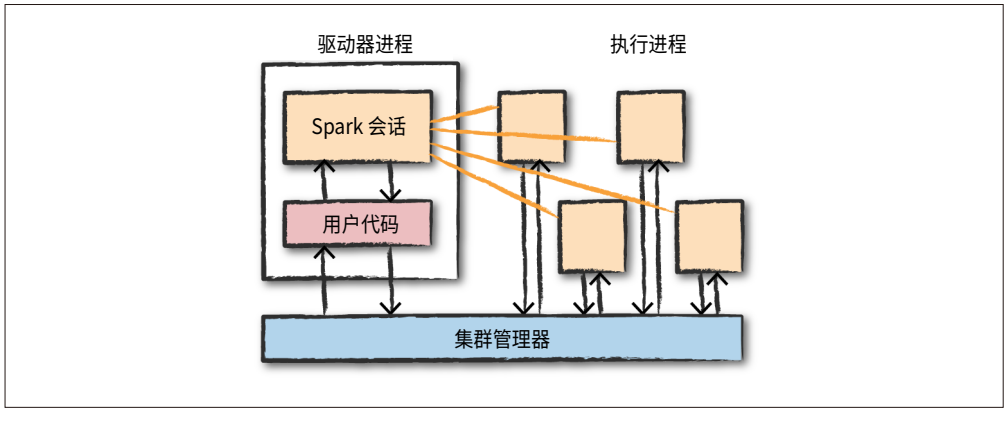


图2-1：Spark架构

在图2-1中我们可以看到左边的驱动器和右边的四个执行器。在该图中，我们忽略了群集节点的概念，用户可以配置指定每个节点上运行多少个执行器。

除了集群模式外，Spark还具有本地运行模式。驱动器和执行器只是简单的进程，这意味着它们可以位于同一台机器或位于不同的机器上。在本地模式下，驱动器和执行器在个人计算机上运行（作为线程）而不是在集群上。我们以本地模式为运行环境编写本书，因此你应该能够在单台机器上运行所有内容。

以下是关于Spark应用程序需要了解的几点：

- Spark使用一个集群管理器来跟踪可用的资源。
- 驱动器进程负责执行驱动器的命令来完成给定的任务。

执行器大部分时候都运行Spark代码，但是驱动器可以通过多种不同语言调用Spark API 来“驱动”。我们来看看下一节的内容。

Spark API的多语言支持

Spark API的多语言支持允许使用多种编程语言运行Spark代码。大多数情况下，Spark在每种语言中都提供了一些核心“概念”，这些概念被转换成在集群上可运行的Spark代码。如果仅使用结构化API，则所有语言都应该具有相似的性能特征。以下是一个简短的概要：

Scala

Spark主要用Scala编写，它也是Spark的“默认”语言。本书大部分地方都将使用Scala代码示例。

Java

尽管Spark是用Scala编写的，但Spark的开发者们在设计时一直很小心，以确保你可以用Java编写Spark代码。本书主要使用Scala，但在某些地方也会提供Java示例。

Python

Python几乎支持所有Scala支持的结构。如果存在相关的Python API，本书会同时提供Python代码与Scala代码。

SQL

Spark支持ANSI SQL 2003标准的一个子集，这使习惯使用SQL的数据分析人员和非程序员可以轻松利用Spark的大数据处理能力，本书会在相关地方提供SQL代码示例。

R

Spark有两个常用的R库：一个作为Spark核心的一部分（SparkR），另一个是R语言开源社区维护的包（sparklyr）。我们将在第32章介绍这两个软件库。

图2-2展示了Spark及其支持这些语言关系的一个简单例子。

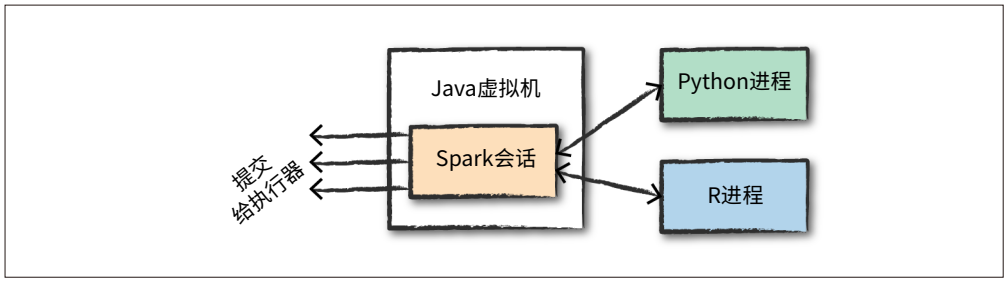


图2-2：SparkSession与支持Spark API编程语言的关系

每种语言的API都维护着我们前面描述的这个核心的概念，SparkSession对象可供用户使用，它也是运行Spark代码的入口点。基于Python或R使用Spark时，不要写显式的JVM指令，相反，你可以编写Python和R代码来调用SparkSession，Spark将它们转换为可以在JVM上运行的代码。

Spark的API

虽然你可以使用各种语言启动Spark任务，但是以这些语言提供的内容需要详细介绍。 Spark有两套基本的API：低级“非结构化”API和更高级别的结构化API。两者我们在本书中都会进行介绍，但更多会关注更高级别的结构化API。

启动Spark

到目前为止，我们已经介绍了Spark应用程序的基本概念，但是这些概念都很抽象，当我们真正开始编写Spark应用程序时，需要一种将用户命令和数据发送给Spark的方法，我们通过创建一个SparkSession来实现。

启动Spark的本地模式，就像第1章介绍的那样，运行`./bin/spark-shell`打开Scala控制台来启动一个交互式会话，你也可以使用`./bin/pyspark`启动Python控制台，这就启动了一个交互式Spark应用程序。还有一个向Spark提交独立应用程序的命令`park-submit`，你可以将预编译的应用程序提交给Spark，第3章将详细介绍如何使用这个命令。

当以这种交互模式启动Spark时，你相当于隐式地创建了一个SparkSession来管理Spark应用程序。当你不是通过交互模式而是通过独立应用程序启动Spark时，你必须在应用程序代码中显式地创建SparkSession对象。

SparkSession

正如本章开头所介绍的，你可以通过名为SparkSession的驱动器来控制Spark应用程序，你需要创建一个SparkSession实例来在群集中执行用户定义的操作，每一个Spark应用程序都需要一个SparkSession与之对应。在Scala和Python中，当你启动控制台时，这个SparkSession就被实例化为一个名为spark的对象。让我们继续看看Scala和Python中如何使用SparkSession：

```
spark
```

在Scala，你会看到如下内容：

```
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@...
```

在Python中你会看到如下的内容：

```
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>
```

现在我们执行一个简单任务，创建一组固定范围的数字，就像电子表格的一个命名列一样：

```
// in Scala
val myRange = spark.range(1000).toDF("number")
# in Python
myRange = spark.range(1000).toDF("number")
```

刚刚已经运行了你的第一个Spark代码！我们创建了一个DataFrame，其中一列包含1000行，值为0~999。这些数字即是一个分布式集合，在集群上运行此命令时，这个集合的每一部分都会被分配到不同的执行器上。这个集合就是一个Spark DataFrame。

DataFrame

DataFrame是最常见的结构化API，简单来说它是包含行和列的数据表。说明这些列和列类型的一些规则被称为模式（schema）。你可以将DataFrame想象为具有多个命名列的电子表格。但是我们熟悉的电子表格和Spark DataFrame是不同的，图2-3展示了它们之间的根本区别：电子表格位于一台计算机上，而Spark DataFrame可以跨越数千台计算机。将数据放在多台计算机上的原因显而易见：因为数据量太大而不适合在一台计算机存储，或者是因为在一台计算机上执行计算所需的时间太长。

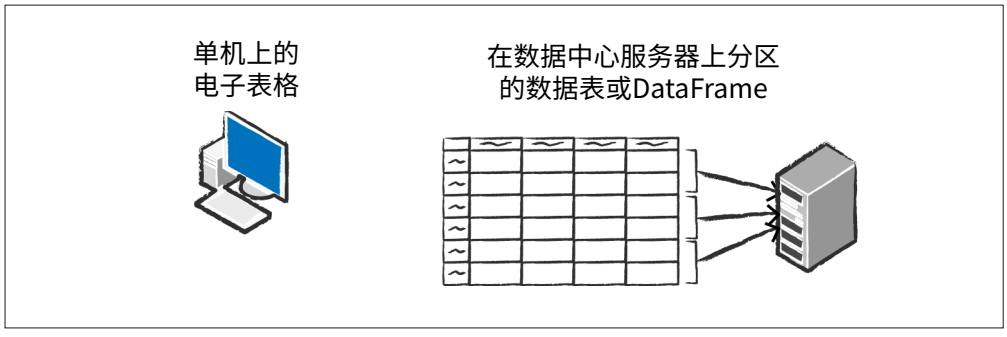


图2-3：分布式数据分析 vs 单机数据分析

DataFrame概念并不是Spark独有的，R和Python都有相似的概念。但是，Python DataFrame和R DataFrame存在于一台机器上（有些例外情况），而不是多台机器上，这就要求你只能使用给定机器上计算资源来操作DataFrame。尽管如此，由于Spark具有适用于Python和R的语言接口，因此可以非常容易地将Pandas（Python） DataFrame转换为Spark DataFrame或将R DataFrame转换为Spark DataFrame。

Spark有几个核心抽象：Dataset，DataFrame，SQL表和弹性分布式数据集（Resilient Distributed Datasets，RDD）。这些不同的抽象都表示分布式数据集合，其中最简单和最有效的是DataFrame，它支持所有语言。我们在第二部分的最后将介绍Dataset，并在第三部分介绍RDD。

数据分区

为了让多个执行器并行地工作，Spark将数据分解成多个数据块，每个数据块叫做一个分区。分区是位于集群中的一台物理机上的多行数据的集合，DataFrame的分区也说明了在执行过程中，数据在集群中的物理分布。如果只有一个分区，即使拥有数千个执行器，Spark也只有一个执行器在处理数据。类似地，如果有多个分区，但只有一个执行器，那么Spark仍然只有一个执行器在处理数据，就是因为只有一个计算资源单位。

值得注意的是，当使用DataFrame时，（大部分时候）你不需要手动操作分区，只需指定数据的高级转换操作，然后Spark决定此工作如何在集群上执行。较低级别的API（通过RDD接口）也是存在的，我们将在第三部分详细介绍。

转换操作

Spark的核心数据结构在计算过程中是保持不变的，这意味着它们在创建之后无法更改。这听起来很奇怪：如果不能改变它，应该如何使用它呢？要“更改”DataFrame，你需要告诉Spark如何修改它以执行你想要的操作，这个过程被称为转换。下面执行一个简单的转换来查找当前DataFrame中的所有偶数：

```
// in Scala
val divisBy2 = myRange.where("number % 2 = 0")

# in Python
divisBy2 = myRange.where("number % 2 = 0")
```

注意这些转换并没有实际输出，这是因为我们仅指定了一个抽象转换。在我们调用一个动作操作（我们将在本章后面详细介绍）之前，Spark不会真的执行转换操作。转换操作是使用Spark表达业务逻辑的核心，有两类转换操作：第一类是指定窄依赖关系的转换操作，第二类是指定宽依赖关系的转换操作。

具有窄依赖关系（narrow dependency）的转换操作（我们称之为窄转换）是每个输入分区仅决定一个输出分区的转换。在前面的代码片段中，where语句指定了一个窄依赖关系，其中一个分区最多只会对一个输出分区有影响，如图2-4所示。

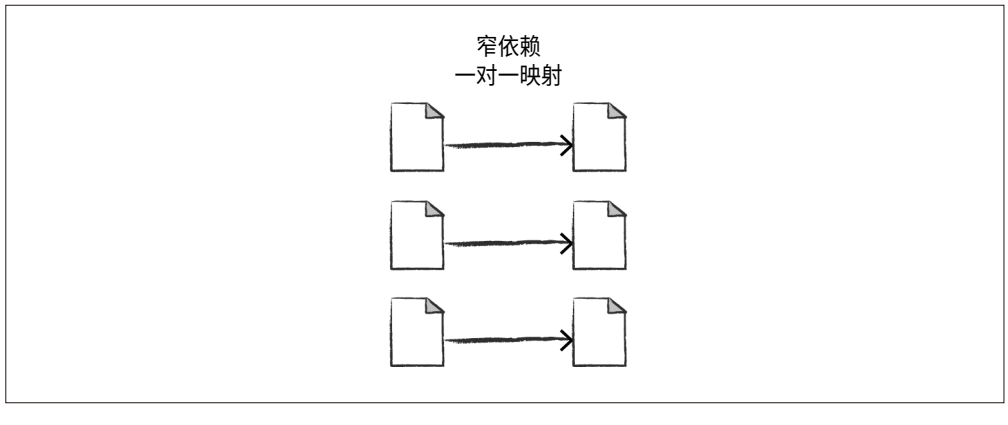


图2-4：窄依赖

具有宽依赖关系（wide dependency）的转换（或宽转换）是每个输入分区决定了多个输出分区。这种宽依赖关系的转换经常被称为洗牌（shuffle）操作，它会在整个集群中执行互相交换分区数据的功能。 如果是窄转换，Spark将自动执行流水线处理，这意味着如果我们在DataFrame上指定了多个过滤操作，它们将全部在内存中执行。

而属于宽转换的shuffle操作不是这样，当我们执行shuffle操作时，Spark将结果写入磁盘。图2-5中展示了宽转换操作。

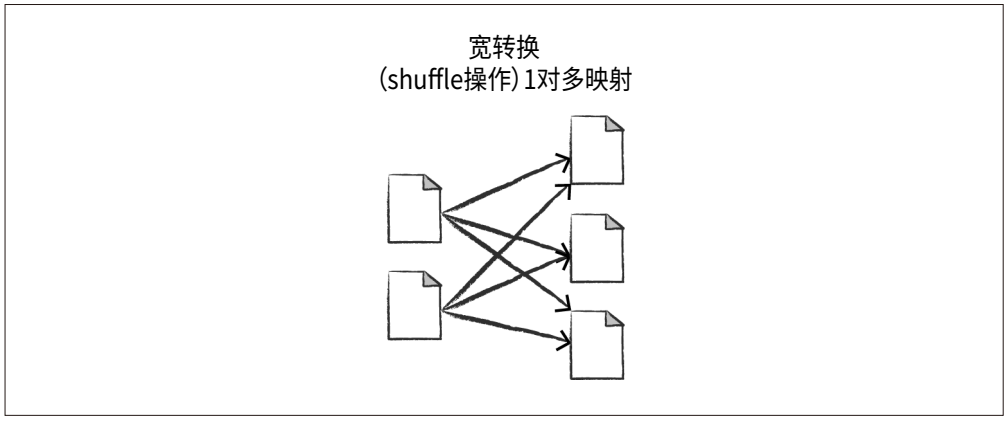


图2-5：宽依赖

你可能会看到很多关于优化shuffle操作的讨论，因为它非常重要。但现在你只需要了解存在这两种转换形式就行，而它们两者简单的区别在于执行不同的数据操作方式，这引出了下面这个称为惰性评估的主题。

惰性评估

惰性评估（lazy evaluation）的意思就是等到绝对需要时才执行计算。在Spark中，当用户表达一些对数据的操作时，不是立即修改数据，而是建立一个作用到原始数据的转换计划。Spark首先会将这个计划编译为可以在集群中高效运行的流水线式的物理执行计划，然后等待，直到最后时刻才开始执行代码。这会带来很多好处，因为Spark可以优化整个从输入端到输出端的数据流。一个很好的例子就是DataFrame的谓词下推（predicate pushdown），假设我们构建一个含有多个转换操作的Spark作业，并在最后指定了一个过滤操作，假设这个过滤操作只需要数据源（输入数据）中的某一行数据，则最有效的方法是在最开始仅访问我们需要的单个记录，Spark会通过自动下推这个过滤操作来优化整个物理执行计划。

动作操作

转换操作使我们能够建立逻辑转换计划。为了触发计算，我们需要运行一个动作操作（action）。一个动作指示Spark在一系列转换操作后计算一个结果。最简单的动作操作是count，它计算一个DataFrame中的记录总数：

```
divisBy2.count()
```

上面代码输出应该是500。当然，count并不是唯一的动作，有三类动作：

- 在控制台中查看数据的动作。
- 在某个语言中将数据汇集为原生对象的动作。
- 写入输出数据源的动作。

下面介绍一个指定动作操作的例子，我们启动一个Spark作业，首先执行过滤转换（一个窄转换），然后执行一个聚合操作（一个宽转换），再在每个分区上执行计数count操作，然后通过collect操作将所有分区的结果汇集到一起，生成某个语言的一个原生对象。你可以通过Spark UI看到所有这些操作，Spark UI是一个包含在Spark软件包中的工具，你可以使用它监视Spark集群上运行的Spark作业。

Spark用户接口

你可以通过Spark的Web UI监控一个作业的进度，Spark UI占用驱动器节点的4040端口。如果在本地模式下运行，你可以通过*http://localhost:4040*访问Spark Web UI。Spark UI上显示了Spark作业的运行状态、执行环境和群集状态等信息，这些信息非常有用，可用于性能调优和代码调试。图2-6展示了一个Spark作业状态的UI示例，其中显示了Spark作业包含了两个运行阶段的九个任务执行情况。

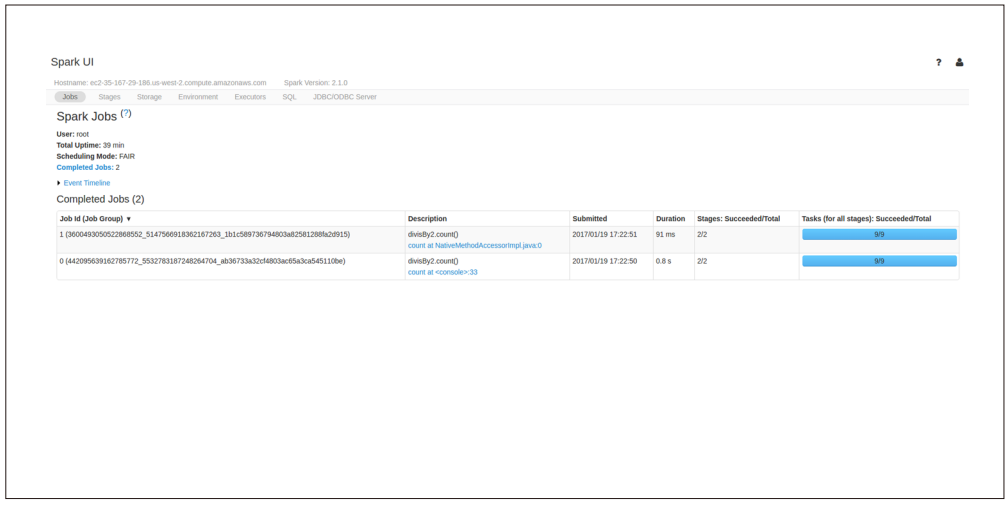


图2-6：Spark UI示例

本章不会详细介绍Spark作业执行和Spark UI，而将在第18章介绍。在这里你只需了解，一个Spark作业包含一系列转换操作并由一个动作操作触发，并可以通过Spark UI监视该作业。

一个完整的例子

在前面的例子中，我们创建了一个包含某数字范围的DataFrame，其实处理的并不是典型的大数据。在本节中，我们将通过一个更实际的例子来强化之前所学的内容，并解释每一步的操作要点。我们将使用Spark分析某国交通局统计的一些航班数据。

在CSV文件夹内有许多文件，当然也有一些包含其他文件格式的文件夹，我们将在第9章讨论，现在我们首先关注CSV文件。

CSV是一种半结构化的数据格式，每个CSV文件包含许多行，每一行对应着以后我们建立的DataFrame中的一行：

```
$ head /data/flight-data/csv/2015-summary.csv

DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count
United States,Romania,15
United States,Croatia,1
United States,Ireland,344
```

Spark可以从大量数据源中读取数据或写入数据，为了读取这些数据，需要用到与我们创建的SparkSession所关联的DataFrameReader，还需要指定文件格式及设置其他选项。在这个例子中，我们将执行一种被称作模式推理的操作，即让Spark猜测DataFrame的模式，我们还将指定文件的第一行为文件头，这些也可以通过设置选项来指定。

为了获取模式信息，Spark会从文件中读取一些数据，然后根据Spark支持的类型尝试解析读取这些行中的数据类型。当然你也可以在读取数据时选择严格指定模式（我们建议在实际生产应用中严格指定模式）：

```
// in Scala
val flightData2015 = spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
.csv("/data/flight-data/csv/2015-summary.csv")

# in Python
flightData2015 = spark\
  .read\
  .option("inferSchema", "true")\
```

```
.option("header", "true")\
.csv("/data/flight-data/csv/2015-summary.csv")
```

这些DataFrame（在Scala和Python）中的每一个都有一些列，但是行数没有指定。行数未指定的原因是因为读取数据是一种转换操作，所以也是一种惰性操作。Spark只了解了几行数据后，就试图猜测每列应该是什么类型。图2-7展示了CSV文件被读到一个DataFrame里后，又被转换为一个本地数组或行列表的过程。

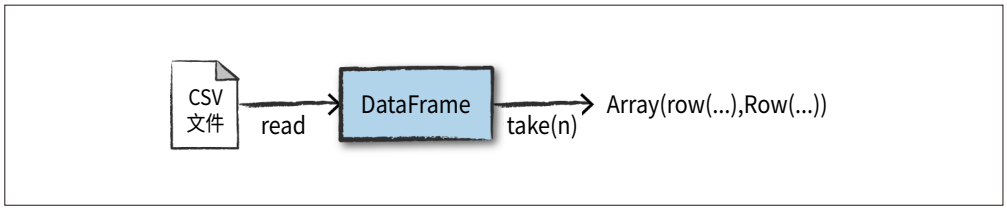


图2-7：CSV文件被读到一个DataFrame里后，又被转换为一个本地数组或行列表

如果我们在DataFrame上执行take操作，我们将看到下面的结果：

```
flightData2015.take(3)

Array([United States,Romania,15], [United States,Croatia...
```

我们还可以指定更多的转换！让我们根据count列的值（这是一个整数类型）排序，图2-8展示了这个过程。

请注意，sort操作不会修改DataFrame，因为sort是一个转换，它通过转换之前的DataFrame来返回新的DataFrame。我们来看看当在结果DataFrame上执行take操作时发生了什么（见图2-8）。

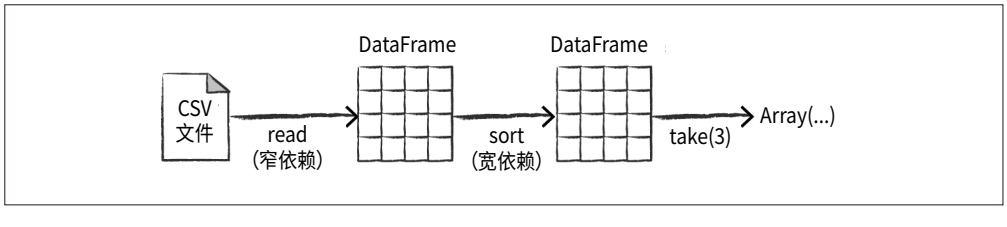


图2-8：DataFrame的读取、排序和收集

当我们调用sort时，什么也不会发生，因为这只是一个转换操作。但是可以通过调用explain函数观察到Spark正在创建一个执行计划，并且可以看到这个计划将会怎样在

集群上执行，调用某个DataFrame的explain操作会显示DataFrame的来源（即Spark是如何执行查询操作的）：

```
flightData2015.sort("count").explain()
== Physical Plan ==
*Sort [count#195 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#195 ASC NULLS FIRST, 200)
   +- *FileScan csv [DEST_COUNTRY_NAME#193,ORIGIN_COUNTRY_NAME#194,count#195] ...
```

恭喜，你刚刚阅读了你的第一个解释计划！解释计划有点神秘，但通过一些实践你就会熟悉它。你可以采用从上到下的方式阅读解释计划，上面是最终结果，下面是数据源。在这种情况下，如果查看每行的第一个关键字，你将看到排序、交换和FileScan。这是因为排序其实是一个宽转换，行需要相互比较和交换。不必过分担心如何理解关于解释计划的所有内容，在使用Spark时，解释计划可以作为调试和帮助理解的有效工具。

现在，我们需要指定一个动作来触发这个计划的执行。在此之前，我们首先完成一个配置。默认情况下，shuffle操作会输出200个shuffle分区，我们将此值设置为5以减少shuffle输出分区的数量：

```
spark.conf.set("spark.sql.shuffle.partitions", "5")

flightData2015.sort("count").take(2)

... Array([United States,Singapore,1], [Moldova,United States,1])
```

图2-9展示了这一操作。请注意，除了逻辑转换外，这里还给出了物理分区的数量。

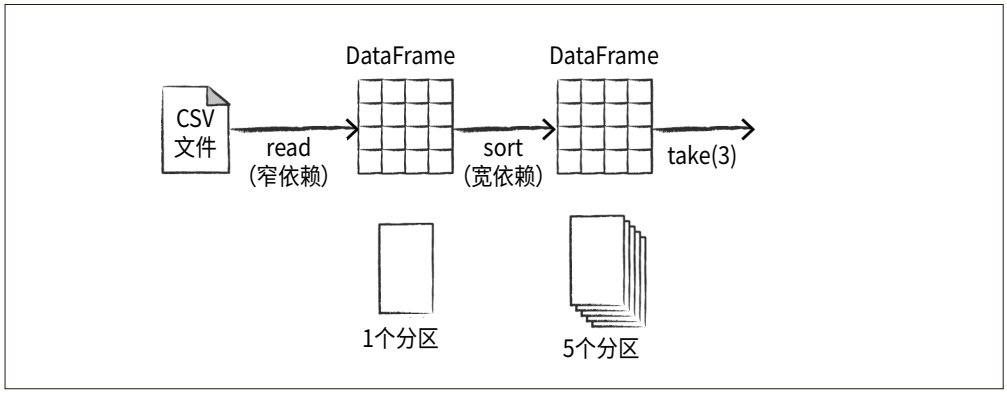


图2-9：DataFrame逻辑操作和物理操作的过程

我们构建的转换逻辑计划定义了DataFrame的血统，这样在任何给定的时间点，Spark都知道如何通过对输入数据执行之前的操作来重新计算任何分区，这就是Spark编程模型的核心——函数式编程，当数据转换保持不变时，相同的输入始终导致相同的输出。

我们不能操纵物理数据，但是我们可以配置参数来指定物理执行的特性。比如我们刚刚设置的shuffle分区参数，程序最终输出了5个分区，因为我们指定了shuffle分区参数为5。你可以更改此设置以改变Spark作业的物理执行特性，尝试使用不同的值，并自己查看分区的数量，你将会看到不同设置下会需要不同的时间。别忘了，你可以通过在4040端口上的Spark UI来监控作业进度，并查看作业的物理和逻辑执行的情况。

DataFrame和SQL

在前面的例子中介绍了一个简单的转换工作，现在让我们通过一个更复杂的转换继续介绍如何使用DataFrame和SQL。不管使用什么语言，Spark以完全相同的方式执行转换操作，你可以使用SQL或DataFrame（基于R，Python，Scala或Java语言）表达业务逻辑，并且在实际运行代码之前，Spark会将该逻辑编译到底层执行计划（可以在解释计划中看到）。使用Spark SQL，你可以将任何DataFrame注册为数据表或视图（临时表），并使用纯SQL对它进行查询。编写SQL查询或编写DataFrame代码并不会造成性能差异，它们都会被“编译”成相同的底层执行计划。

你可以使用一个简单的方法将任何DataFrame放入数据表或视图中：

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

现在我们可以在SQL中查询数据，我们将使用`spark.sql`函数（别忘了`spark`是我们的SparkSession变量），它可方便地返回新的DataFrame。尽管这在逻辑上会有种绕圈的感觉，即对DataFrame的SQL查询返回另一个DataFrame，但它实际上非常有用。这使得你可以在任何给定的时间点以最方便的方式指定转换操作，而不会牺牲效率！为了理解这种情况，我们来看看下面两个解释计划：

```
// in Scala
val sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

val dataFrameWay = flightData2015
  .groupBy('DEST_COUNTRY_NAME)
  .count()
```

```
sqlWay.explain
dataFrameWay.explain

# in Python
sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

dataFrameWay = flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .count()

sqlWay.explain()
dataFrameWay.explain()

== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
   +- *HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[partial_count(1)])
      +- *FileScan csv [DEST_COUNTRY_NAME#182] ...
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
   +- *HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[partial_count(1)])
      +- *FileScan csv [DEST_COUNTRY_NAME#182] ...
```

请注意，这些计划编译后是完全相同的基本执行计划！

让我们从数据中提取一些有趣的统计结果。有一点要说明的是，Spark中的DataFrame（和SQL）已经有了大量的可用操作，有数百种函数可供你使用和导入，以帮助你更快地解决大数据问题。我们将使用max函数来统计往返任何特定位置的航班最大数量，这要扫描DataFrame中相关列中的每个值，并检查它是否大于先前看到的值。这是一个转换，因为我们不断过滤，最后仅得到一行。下面我们来看看如何编写Spark程序：

```
// in Scala
import org.apache.spark.sql.functions.max

flightData2015.select(max("count")).take(1)
# in Python
from pyspark.sql.functions import max

flightData2015.select(max("count")).take(1)
```

这是一个简单的例子，结果为370002。我们来执行一些更复杂的操作，在数据中找到前五个目标国家，这是我们的第一个多转换查询，所以我们将逐步介绍。让我们从一个相当简单的SQL聚合开始：

```
// in Scala
val maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")
maxSql.show()

# in Python
maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")

maxSql.show()

+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

现在让我们看看DataFrame语法，它的语义和SQL相似但实现略有不同。但是，正如我们所提到的，两者的物理执行计划是相同的。接下来，让我们运行查询并观察现象。

```
// in Scala
import org.apache.spark.sql.functions.desc

flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
  .withColumnRenamed("sum(count)", "destination_total")
  .sort(desc("destination_total"))
  .limit(5)
  .show()

# in Python
from pyspark.sql.functions import desc

flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .sum("count")\
  .withColumnRenamed("sum(count)", "destination_total")\
```

```
  .sort(desc("destination_total"))\
  .limit(5)\
  .show()

+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

从输入数据开始我们需要七个步骤，可以在DataFrame的解释计划中看到这一点。图2-10展zhan示我们在“代码”中执行的一系列步骤。由于Spark会针对物理执行计划做一系列优化，所以真正的执行计划（调用`explain`函数返回的执行计划）将不同于图2-10所示的执行计划。这个执行计划是一个有向无环图（directed acyclic graph，DAG）的转换，每个转换产生一个新的不可变的DataFrame，我们可以在这个DataFrame上调用一个动作来产生一个结果。

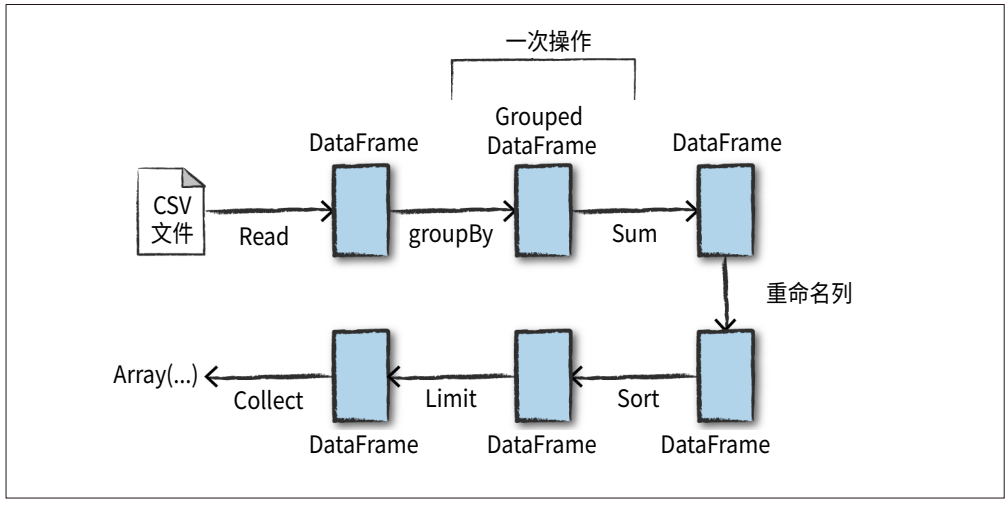


图2-10：DataFrame转换的完整流程

第一步读取数据。我们之前定义了DataFrame，但是Spark实际上并没有真正读取它，直到在DataFrame上调用动作操作后才会真正读取它。

第二步分组。当调用`groupBy`时，我们最终得到了一个`RelationalGroupedDataset`对象，它是一个DataFrame对象，具有指定的分组，但需要用户指定聚合操作，然后才

能进一步查询。我们按键（或键集合）分组，然后再对每个键对应分组进行聚合操作。

第三步指定聚合操作。我们使用sum聚合操作，这需要输入一个列表达式，或者简单的一个列名称。sum方法调用的结果是产生一个新的DataFrame，它有一个新的表结构，并知道每个列的类型。再次强调，到这里还是没有执行计算，这只是我们表达的另一种转换操作，而Spark能够通过这些转换操作跟踪我们的类型信息。

第四步简单的重命名。我们使用带有两个参数的`withColumnRenamed`方法，即原始列名称和新列名称。当然，这还不会执行计算，也只是一种转换！

第五步对数据进行排序。如果我们要所有行按照`destination_total`列的大小排序，获得该DataFrame中`destination_total`值较大的一些行作为结果。

你可能注意到我们必须导入一个函数来执行此操作，即`desc`函数。可能还注意到，`desc`函数不是返回一个字符串，而是一个`Column`。通常来说，许多DataFrame方法将接受字符串（作为列名称）、Column类型或表达式，列和表达式实际上是完全相同的东西。

倒数第二步，我们指定了一个限制。这只是说明我们只想返回最终DataFrame中的前五个值，而不是所有数据。

最后一步是我们要执行的动作！现在我们实际上才开始收集DataFrame的结果，Spark将返回一个我们所使用语言的数组或列表。为了更好的理解这些内容，我们来看看前面查询的解释计划：

```
// in Scala
flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
  .withColumnRenamed("sum(count)", "destination_total")
  .sort(desc("destination_total"))
  .limit(5)
  .explain()

# in Python
flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .sum("count")\
  .withColumnRenamed("sum(count)", "destination_total")\
  .sort(desc("destination_total"))\
  .limit(5)\
  .explain()
== Physical Plan ==
```

```
TakeOrderedAndProject(limit=5, orderBy=[destination_total#16194L DESC], outpu...
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[sum(count#7325L)])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#7323, 5)
      +- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[partial_sum...
         +- InMemoryTableScan [DEST_COUNTRY_NAME#7323, count#7325L]
               +- InMemoryRelation [DEST_COUNTRY_NAME#7323, ORIGIN_COUNTRY_NA...
                     +- *Scan csv [DEST_COUNTRY_NAME#7578,ORIGIN_COUNTRY_NAME...
```

虽然这个解释计划与我们确切的“概念计划”不符，但所有步骤都在那里，你可以看到`limit`语句以及`orderBy`（在第一行），还可以看到我们的聚合操作是如何在`partial_sum`调用中的两个阶段发生的，这是因为数字的sum操作是可交换的，并且Spark可以在每个分区单独执行sum操作。当然，我们可以看到如何在DataFrame中读取数据。

当然，我们并不总是需要收集数据。我们也可以将它写出到Spark支持的任何数据源。例如，假设我们想要将信息存储在像PostgreSQL这样的数据库中，或者将它们写入到另一个文件中。

小结

本章介绍了Apache Spark的基础知识。我们讨论了转换和动作，以及Spark如何惰性执行转换操作的DAG图来优化DataFrame上的物理执行计划。我们还讨论了如何将数据组织到分区中，并为处理更复杂的转换设定多个阶段。在第3章中我们将介绍庞大的Spark生态系统，并了解Spark中提供的包括流数据处理和机器学习等一些更高级的概念和工具。

第3章

Spark工具集介绍

第2章介绍了Spark的核心概念，包括Spark结构化API的转换和动作。这些简单的概念是学习Apache Spark生态系统中众多工具库（见图3-1）的基础，Spark除了提供这些低级API和结构化API，还包括一系列标准库来提供额外的功能。

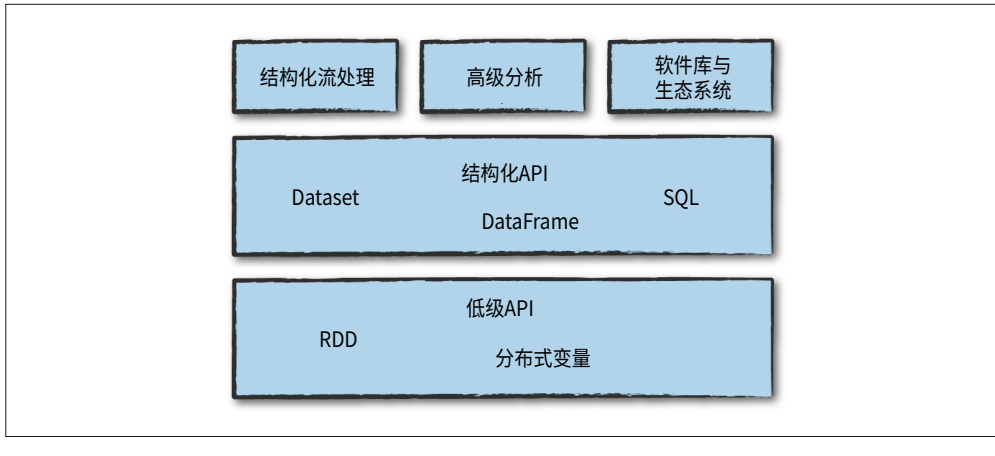


图3-1：Spark工具集

Spark库支持各种不同的任务，包括图分析、机器学习、流处理，以及提供与其他计算系统和存储系统的集成能力等。本章介绍了Spark所提供的大部分功能，包括一些我们尚未涉及的API以及一些主要的库。关于每个部分你都可以在本书之后的内容中找到更详细的介绍，本章的目的是给你提供一个关于Spark基本功能的概览。

本章包括以下内容：

- 使用spark-submit运行应用程序。
- Dataset：类型安全的结构化数据API。
- 结构化流处理。
- 机器学习和高级分析。
- 弹性分布式数据集（RDD）：Spark的低级API。
- SparkR。
- 第三方软件包生态系统。

学习完本章后，你可以跳到本书的相应部分找到你所关心的特定主题的问题的答案。

运行生产应用程序

Spark简化了开发和构建大数据处理程序的过程，Spark还可以通过内置的命令行工具`spark-submit`轻松地将测试级别的交互式程序转化为生产级别的应用程序。`spark-submit`将你的应用程序代码发送到一个集群并在那里执行，应用程序将一直运行，直到它（完成任务后）正确退出或遇到错误。你的程序可以在集群管理器的支持下进行，包括Standalone，Mesos和YARN等。

`spark-submit`提供了若干控制选项，你可以指定应用程序需要的资源，以及应用程序的运行方式和运行参数等。

你可以使用Spark支持的任何语言编写应用程序，然后提交它执行。最简单的例子是在本地机器上运行应用程序，我们下面将通过运行Spark附带的Scala示例应用程序，在你下载Spark的目录中运行以下命令：

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local \
  ./examples/jars/spark-examples_2.11-2.2.0.jar 10
```

该示例应用程序计算pi的值以达到某个精度。在这里，告诉`spark-submit`我们想在本地机器上运行，想运行哪个类和哪个JAR，以及一些命令行参数。

我们还可以使用以下命令运行该应用程序的Python版本：

```
./bin/spark-submit \
  --master local \
  ./examples/src/main/python/pi.py 10
```

通过更改spark-submit的master参数，我们还可以将相同的应用程序提交到一个集群上，该集群需要运行Spark独立集群管理器、Mesos或YARN。

使用spark-submit可以很方便地运行本书中的许多例子。在本章的其余部分中，还将介绍我们尚未见到的一些API示例。

Dataset：类型安全的结构化API

我们将要描述的一个API是Spark结构化API的类型安全的版本，叫做Dataset，用于在Java和Scala中编写静态类型的代码。Dataset API在Python和R中不可用，因为这些语言是动态类型的。

回想一下，我们在前一章中看到的DataFrame是一个分布式的类型为Row的对象集合，它可以存储多种类型的表格数据。Dataset API让用户可以用Java / Scala类定义DataFrame中的每条记录，并将其作为类型对象的集合来操作，类似于Java ArrayList或Scala Seq。Dataset中可用的API是类型安全的，这意味着Dataset中的对象不会被视为与初始定义的类不相同的另一个类。这使得Dataset在编写大型应用程序时尤其有效，这样的话多个软件工程师可以通过协商好的接口进行交互。

Dataset类通过内部包含的对象类型进行参数化，如Java中的Dataset <T>和Scala中的Dataset [T] Dataset [Person]将仅包含Person类的对象。从类型需要Spark 2.0开始，受支持的类型遵循Java中的JavaBean模式，或是Scala中的case类。之所以这些类型需要受到限制，是因为Spark要能够自动分析类型T，并为Dataset中的表格数据创建适当的模式。

Dataset的一个好处是，在有需要或只要你想要的时候就可以使用它们。例如，在下面的例子中，我们将定义wan自己的数据类型，并通过某种map函数和filter函数来操作它。在完成我们的操作之后，Spark可以自动将其重新转换为DataFrame，并且可以使用Spark包含的数百个函数进一步处理它。这样可以很容易地降到较低的级别，在必要时执行类型安全的编码，并且也可以升级到更高级的SQL，以进行更快速的分析。下面是一个小例子，展示了如何使用类型安全函数和DataFrame类SQL表达式来快速编写业务逻辑：

```
// in Scala
case class Flight(DEST_COUNTRY_NAME: String,
                  ORIGIN_COUNTRY_NAME: String,
                  count: BigInt)
val flightsDF = spark.read
  .parquet("/data/flight-data/parquet/2010-summary.parquet/")
val flights = flightsDF.as[Flight]
```

最后一个好处是，当在Dataset上调用collect或take时，它将会收集Dataset中合适类型的对象，而不是DataFrame的Row对象。这样可以很容易地保证类型安全，并以分布式和本地方式安全地执行操作，而无需更改代码：

```
// in Scala
flights
  .filter(flight_row => flight_row.ORIGIN_COUNTRY_NAME != "Canada")
  .map(flight_row => flight_row)
  .take(5)

flights
  .take(5)
  .filter(flight_row => flight_row.ORIGIN_COUNTRY_NAME != "Canada")
  .map(fr => Flight(fr.DEST_COUNTRY_NAME, fr.ORIGIN_COUNTRY_NAME, fr.count + 5))
```

我们将在第11章深入介绍Dataset。

结构化流处理

结构化流处理是用于数据流处理的高级API，它在Spark 2.2版本后可用。你可以像在批处理模式下一样使用Spark的结构化API执行结构化流处理，并以流式方式运行它们，使用结构化流处理可以减少延迟并允许增量处理。最重要的是，它可以让你快速地从流式系统中提取有价值的信息，而且几乎不需要更改代码。你可以按照传统批处理作业的模式进行设计，然后将其转换为流式作业，即增量处理数据，这样就使得流处理任务变得异常简单。

让我们通过一个简单的例子来说明如何使用Spark结构化流处理。为此，我们将使用一个销售数据集（*https://github.com/databricks/Spark-The-Definitive-Guide/tree/master/data/retail-data*），这个数据集有日期和时间信息供我们使用能够使用。我们将使用按天分组的文件，每一个文件夹中包含一天的数据。

我们用另外一个进程来模拟持续产生的数据，由于这是一组零售数据，假设这些数据是由零售商持续生成的，并由我们的结构化流式处理作业进行处理。

我们展示一个数据样本，以便你可以了解一下大概的数据格式：

```
InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country
536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,2010-12-01 08:26:00,2.55,17...
536365,71053,WHITE METAL LANTERN,6,2010-12-01 08:26:00,3.39,17850.0,United Kin...
536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,2010-12-01 08:26:00,2.75,17850...
```

我们首先将按照静态数据集的处理方法来进行分析，并创建一个DataFrame来执行此操作。 我们还将从这个静态数据集创建一个schema模式（我们还将在第五部分介绍其他一些流处理的模式推理方法）：

```
// in Scala
val staticDataFrame = spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/data/retail-data/by-day/*.csv")

staticDataFrame.createOrReplaceTempView("retail_data")
val staticSchema = staticDataFrame.schema

# in Python
staticDataFrame = spark.read.format("csv")\
  .option("header", "true")\
  .option("inferSchema", "true")\
  .load("/data/retail-data/by-day/*.csv")

staticDataFrame.createOrReplaceTempView("retail_data")
staticSchema = staticDataFrame.schema
```

因为我们处理的是时间序列数据，所以在此之前我们需要强调如何对数据进行分组和聚合操作。在这个例子中，我们将查看特定客户（由CustomerId识别）进行大量采购的时间。例如，我们添加一个列用于统计总费用，并查看客户花费最多的那个日期。

窗口函数包含每天的所有数据，它只是我们数据中时间序列栏的一个窗口，这是一个用于处理日期和时间戳的有用工具，因为我们可以通过时间间隔指定我们的需求，而Spark将会把所有数据集合起来交给我们：

```
// in Scala
import org.apache.spark.sql.functions.{window, column, desc, col}
staticDataFrame
  .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost",
    "InvoiceDate")
  .groupBy(
    col("CustomerId"), window(col("InvoiceDate"), "1 day"))
  .sum("total_cost")
  .show(5)

# in Python
from pyspark.sql.functions import window, column, desc, col
staticDataFrame\
```

```
  .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost",
    "InvoiceDate")\
  .groupBy(
    col("CustomerId"), window(col("InvoiceDate"), "1 day"))\
  .sum("total_cost")\
  .show(5)
```

值得一提的是你也可以像SQL代码那样运行它，就像我们在前一章中看到的那样。

以下是你将看到的输出：

```
+----------+--------------------+------------------+
|CustomerId|              window|   sum(total_cost)|
+----------+--------------------+------------------+
|   17450.0|[2011-09-20 00:00...|          71601.44|
...
|      null|[2011-12-08 00:00...|31975.590000000007|
+----------+--------------------+------------------+
```

null表示某些事务没有customerId标签。

这是静态的DataFrame版本，如果你熟悉语法，这里很容易理解。

如果你以本地模式运行它，应该将shuffle分区数设置为更适合本地模式的数量，此配置指定在shuffle后应创建的分区数量。在默认情况下，这个值是200，但是因为这台机器上没有足够的执行进程，所以应该把分区数量减少到5。我们在第2章中做了同样的配置，所以如果你不记得为什么这很重要，请翻阅回顾一下。

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
```

现在已经了解看到了它的工作原理，让我们来看看它的流处理代码，你会注意到代码的实际更改很少，最大的变化是我们使用readStream而不是read，另外你会注意到maxFilesPerTrigger选项，它只是指定我们应该一次读入的文件数量。这是为了让我们的演示程序更像是“流处理”，而在实际生产场景中，这可能会被省略。

```
val streamingDataFrame = spark.readStream
    .schema(staticSchema)
    .option("maxFilesPerTrigger", 1)
    .format("csv")
    .option("header", "true")
    .load("/data/retail-data/by-day/*.csv")

# in Python
streamingDataFrame = spark.readStream\
    .schema(staticSchema)\
    .option("maxFilesPerTrigger", 1)\
```

```
    .format("csv")\
    .option("header", "true")\
    .load("/data/retail-data/by-day/*.csv")
```

现在我们可以看到我们的DataFrame是否代表流数据：

```
streamingDataFrame.isStreaming // 返回 true
```

我们对流数据执行与之前对静态DataFrame一样的业务逻辑（按时间窗口统计花费）。我们随后对此过程进行总结：

```
// in Scala
val purchaseByCustomerPerHour = streamingDataFrame
  .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost",
    "InvoiceDate")
  .groupBy(
    $"CustomerId", window($"InvoiceDate", "1 day"))
  .sum("total_cost")

# in Python
purchaseByCustomerPerHour = streamingDataFrame\
  .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost",
    "InvoiceDate")\
  .groupBy(
    col("CustomerId"), window(col("InvoiceDate"), "1 day"))\
  .sum("total_cost")
```

这仍然是一个惰性操作，所以我们需要调用一个对流数据的动作来执行这个流处理。

流数据动作与静态数据动作有点不同，因为我们首先要将流数据缓存到某个地方，而不是像对静态数据那样直接调用count函数（对流数据没有任何意义）。流数据将被缓存到一个内存上的数据表里，当每次被触发器触发后更新这个内存缓存。在这个例子中，因为我们之前设置的maxFilesPerTrigger选项每次读完一个文件后都会被触发，Spark将基于新读入的文件更新内存数据表的内容，这样的话，聚合操作可以始终维护着历史数据中的最大值。

```
// in Scala
purchaseByCustomerPerHour.writeStream
    .format("memory") // memory代表将表存入内存
    .queryName("customer_purchases") // 存入内存的表的名称
    .outputMode("complete") // complete表示保存表中所有记录
    .start()

# in Python
purchaseByCustomerPerHour.writeStream\
```

```
    .format("memory")\
    .queryName("customer_purchases")\
    .outputMode("complete")\
.start()
```

当启动数据流后，我们可以运行查询来调试我们的结果，查看我们的结果是否已经被写入结果的接收器：

```
// in Scala
spark.sql("""
  SELECT *
  FROM customer_purchases
  ORDER BY `sum(total_cost)` DESC
  """)
  .show(5)

# in Python
spark.sql("""
  SELECT *
  FROM customer_purchases
  ORDER BY `sum(total_cost)` DESC
  """)\
  .show(5)
```

你会注意到，输出表格的内容会随着读入更多的数据而发生实时变化。在处理完每个文件后，结果可能会根据数据发生改变，也可能不会。当然，因为我们要根据客户购买能力对客户进行分组，所以我们希望随着时间的推移，客户的最大购买量会增加（并且会持续一段时间）。 另外，你也可以将结果输出到控制台：

```
purchaseByCustomerPerHour.writeStream
    .format("console")
    .queryName("customer_purchases_2")
    .outputMode("complete")
    .start()
```

你不应该在实际生产中使用这两种流处理方法，但它们确实可以方便地演示结构化流处理的强大功能。注意这个时间窗口是基于事件时间的，而不是Spark处理数据的时间，在新的结构化流处理解决这个问题以前，这是Spark流处理的缺点之一。我们将在第五部分深入探讨结构化流处理。

机器学习和高级数据分析

Spark的另一个优势是它使用称为MLlib的机器学习算法内置库支持大规模机器学习。Mllib支持对数据进行预处理、整理、模型训练和大规模预测，甚至可以使用MLlib中训练的模型在结构化流处理中对流数据进行预测。Spark提供了一个复杂的机器学习

API，用于执行各种机器学习任务，从分类到回归，从聚类到深度学习。为了说明这个功能，我们将使用称为k-means的标准聚类算法对数据执行一些基本的聚类操作。

k-Means是什么？

k-means是一种聚类算法。首先从数据中随机选出k个初始聚类中心，最接近某个中心的那些点被分配到一个聚类里，并根据分配到该聚类的点计算它们的中心，这个中心被称为centroid。然后，将最接近该centroid的点标记为属于该centroid的点，并根据分配到某个centroid的点群计算新的中心用来更新centroid。 重复这个过程来进行有限次的迭代，或者直到收敛（中心点停止变化）。

Spark准备了许多内置的预处理方法，下面将演示这些预处理方法，这些预处理方法将原始数据转换为合适的数据格式，它将在之后用于实际训练模型中，并进一步进行预测：

```
staticDataFrame.printSchema()

root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: timestamp (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)
```

MLlib中的机器学习算法要求将数据表示为数值形式，而我们当前的数据由多种不同类型表示，包括时间戳、整数和字符串等，因此我们需要将这些数据转换为数值。在这个例子中，我们将使用几个DataFrame转换来处理我们的日期数据：

```
// in Scala
import org.apache.spark.sql.functions.date_format
val preppedDataFrame = staticDataFrame
  .na.fill(0)
  .withColumn("day_of_week", date_format($"InvoiceDate", "EEEE"))
  .coalesce(5)

# in Python
from pyspark.sql.functions import date_format, col
preppedDataFrame = staticDataFrame\
  .na.fill(0)\
  .withColumn("day_of_week", date_format(col("InvoiceDate"), "EEEE"))\
  .coalesce(5)
```

我们也需要将数据分成训练和测试集。在该示例中，我们手动将某个购买日期之前的数据作为训练集，之后的数据为测试集。我们也可以使用MLlib的转换API通过训练验证分割或交叉验证来创建训练和测试集（这些主题将在第六部分中详细介绍）：

```
// in Scala
val trainDataFrame = preppedDataFrame
  .where("InvoiceDate < '2011-07-01'")
val testDataFrame = preppedDataFrame
  .where("InvoiceDate >= '2011-07-01'")

# in Python
trainDataFrame = preppedDataFrame\
  .where("InvoiceDate < '2011-07-01'")
testDataFrame = preppedDataFrame\
  .where("InvoiceDate >= '2011-07-01'")
```

现在我们已经准备好了数据，再把它分成一个训练集和一个测试集。由于这是一组时间序列数据，因此我们在数据集中选择一个一个的日期作为分割，虽然这可能不是训练集和测试集的最佳分割，但对于当前的这个例子来说，这种分割足够好了。我们会看到我们的数据集被大致分为两部分：

```
trainDataFrame.count()
testDataFrame.count()
```

请注意，这些转换是DataFrame转换，我们将在第二部分进行更详细的介绍。 Spark的Mllib也提供了一些转换，我们可以用它们自动化一些常用的转换。例如StringIndexer就是这样一种MLib提供的转换：

```
// in Scala
import org.apache.spark.ml.feature.StringIndexer
val indexer = new StringIndexer()
  .setInputCol("day_of_week")
  .setOutputCol("day_of_week_index")

# in Python
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer()\
  .setInputCol("day_of_week")\
.setOutputCol("day_of_week_index")
```

这将使每周的星期几转换成相应的数值，例如，Spark可以将星期六表示为6，将星期一表示为1。但是，通过此编号方案，我们隐含地指出星期六大于星期一（因为由纯数值表示），但是这显然是不正确的。为了解决这个问题，我们需要使用一个OneHotEncoder来将每个值编码为其原来对应的列，这些布尔变量标识了该数值是否为与星期几相关的日子：

```
// in Scala
import org.apache.spark.ml.feature.OneHotEncoder
val encoder = new OneHotEncoder()
  .setInputCol("day_of_week_index")
  .setOutputCol("day_of_week_encoded")

# in Python
from pyspark.ml.feature import OneHotEncoder
encoder = OneHotEncoder()\
  .setInputCol("day_of_week_index")\
  .setOutputCol("day_of_week_encoded")
```

其中每一个都会产生一组列，我们将“组合”它们成一个向量。Spark中的机器学习算法输入都为vector类型，即一组数值：

```
// in Scala
import org.apache.spark.ml.feature.VectorAssembler

val vectorAssembler = new VectorAssembler()
  .setInputCols(Array("UnitPrice", "Quantity", "day_of_week_encoded"))
  .setOutputCol("features")

# in Python
from pyspark.ml.feature import VectorAssembler

vectorAssembler = VectorAssembler()\
  .setInputCols(["UnitPrice", "Quantity", "day_of_week_encoded"])\
  .setOutputCol("features")
```

在这里，我们有三个关键特征：价格、数量和星期几。接下来，我们把这些操作设置为流水线处理模式，这样一来，就可以通过完全相同的流程对未来新产生的数据进行转换。

```
// in Scala
import org.apache.spark.ml.Pipeline

val transformationPipeline = new Pipeline()
  .setStages(Array(indexer, encoder, vectorAssembler))

# in Python
from pyspark.ml import Pipeline

transformationPipeline = Pipeline()\
  .setStages([indexer, encoder, vectorAssembler])
```

训练的准备过程需要两步，首先需要为我们的数据设置合适的转换操作，我们将在第六部分深入介绍这些内容，我们的StringIndexer需要知道有多少非重复值，这样才能对应每个字符串一个数值，另外编码操作很容易，但Spark必须查看要索引的列中存在的所有不同值，这样才可以在稍后存储这些值：

```
// in Scala
val fittedPipeline = transformationPipeline.fit(trainDataFrame)

# in Python
fittedPipeline = transformationPipeline.fit(trainDataFrame)
```

在配置好了训练数据后，我们下一步是采用流水线处理模型完成整个数据的预处理过程，以持续的和可重复的模式来转换我们的所有数据：

```
// in Scala
val transformedTraining = fittedPipeline.transform(trainDataFrame)

# in Python
transformedTraining = fittedPipeline.transform(trainDataFrame)
```

在这里值得一提的是，我们可以将模型训练过程也加入到流水线处理过程中，而我们不这样做是为了缓存整个训练数据。这样我们可以对模型训练过程中的超参数进行调整，避免持续重复训练过程中的转换操作。对于缓存过程，我们会在第四部分对缓存优化进行更详细的讨论，它将中间转换数据集的副本立即放入内存，这样我们可以用较低的代价反复访问数据，这远远比重新运行整个流水线处理得到训练数据集节省开销。如果你很好奇这会造成多大的性能区别，可以尝试对两种情况进行分别训练，你会看到性能的明显差别：

```
transformedTraining.cache()
```

我们现在有了一套训练数据集，是时候训练这个模型了。首先，将导入我们想要使用的相关模型包，并使用和实例化它：

```
// in Scala
import org.apache.spark.ml.clustering.KMeans
val kmeans = new KMeans()
  .setK(20)
  .setSeed(1L)

# in Python
from pyspark.ml.clustering import KMeans
kmeans = KMeans()\
  .setK(20)\
  .setSeed(1L)
```

在Spark中，训练机器学习模型是一个具有两阶段的过程。首先，我们初始化一个未经训练的模型，然后进行训练。在MLlib的DataFrame API中，每种算法都有两种类型，对于未经训练的算法版本，它们遵循“XX Algorithm”的命名方式，对于训练后的算法版本，我们使用“XXX AlgorithmModel”的命名方式。在我们的例子中，就是未训练的“KMeans”和训练完的“KMeansModel”。

MLlib的DataFrame API中的估计器与我们之前看到的像StringIndexer这样的预处理转换操作使用大致相同的接口，它使得整个流水线处理过程（包括模型训练）变得简单。在这里，我们希望一步一步地解释，所以在这个例子中我们选择不把模型训练包含到流水线处理过程中。

```
// in Scala
val kmModel = kmeans.fit(transformedTraining)

# in Python
kmModel = kmeans.fit(transformedTraining)
```

在我们训练完这个模型之后，我们可以根据训练集上的一些评价指标来评估开销。处理这个数据集带来的开销实际上相当高，这可能是由于我们的预处理和数据扩展部分没有做好，我们在将在第25章深入讨论这点：

```
kmModel.computeCost(transformedTraining)

// in Scala
val transformedTest = fittedPipeline.transform(testDataFrame)

# in Python
transformedTest = fittedPipeline.transform(testDataFrame)

kmModel.computeCost(transformedTest)
```

当然，我们可以继续改进这个模型，执行更多的预处理过程，以及执行超参数调整，以确保我们获得一个更好的模型，我们将在第六部分对这个主题进行深入的讨论。

低级API

Spark包含了很多低级原语，以支持通过弹性分布式数据集（RDD）对任意的Java和Python对象进行操作，事实上，Spark中的所有对象都建立在RDD之上。正如我们将在第4章讨论的，DataFrame操作都是基于RDD之上的，这些高级操作被编译到较低级的RDD上执行，以方便和实现极其高效的分布式执行。有些时候你可能会使用RDD，特别是在读取或操作原始数据时，但大多数情况下你应该坚持使用高级的结构化API。RDD比DataFrame更低级，因为它向终端用户暴露物理执行特性（如分区）。

你可能使用RDD来并行化已经存储在驱动器机器内存中的原始数据。例如，让我们并行化一些简单的数字并创建一个DataFrame，我们可以将RDD转换为DataFrame，以便与其他DataFrame一起使用它。

```
// in Scala
spark.sparkContext.parallelize(Seq(1, 2, 3)).toDF()
```

```
# in Python
from pyspark.sql import Row

spark.sparkContext.parallelize([Row(1), Row(2), Row(3)]).toDF()
```

RDD可以在Scala和Python中使用，但是它们并不完全等价，这与DataFrame API（执行特性相同）有所不同，这是由于RDD某些底层实现细节导致的区别。我们将在第四部分介绍一些低级别的API，包括RDD。作为终端用户，除非你维护的是较老的Spark代码，否则你不需要使用RDD来执行任务。Spark最新版本基本上没有RDD的实例，所以除了处理一些非常原始的未处理和非结构化数据之外，你应该使用结构化API而不是RDD。

SparkR

SparkR是一个在Spark上运行的R语言工具，它具有与Spark其他支持语言相同的设计准原则。要使用SparkR，只需将SparkR库导入到你的环境中并运行代码。它与Python API非常相似，只是它遵循R的语法而不是Python的。在大多数情况下，SparkR支持Python支持的所有功能：

```
# in R
library(SparkR)
sparkDF <- read.df("/data/flight-data/csv/2015-summary.csv",
         source = "csv", header="true", inferSchema = "true")
take(sparkDF, 5)

# in R
collect(orderBy(sparkDF, "count"), 20)
```

R用户还可以使用其他的R语言库，如magrittr中的管道运算符，使Spark转换更符合R语言风格，这样将易于使用其他库，如调用ggplot库可以进行更复杂的绘图：

```
# in R
library(magrittr)
sparkDF %>%
  orderBy(desc(sparkDF$count)) %>%
  groupBy("ORIGIN_COUNTRY_NAME") %>%
  count() %>%
  limit(10) %>%
  collect()
```

我们将不会像在Python中那样包含R代码示例，因为本书中适用于Python的每个概念几乎都适用于SparkR，唯一的区别是语法。我们将在第Ⅶ部分介绍SparkR和sparklyr。

Spark的生态系统和工具包

Spark最棒的部分之一就是开源社区维护的工具包和支持它的生态系统，其中一些工具甚至会在成熟并广泛使用后直接进入Spark的核心项目。在撰写本文时，工具包列表相当长，数量超过了300个，而且更多的工具包将被添加进来。你可以在*https://spark-packages.org/*找到Spark Packages的索引，所有用户都可以将自己开发的工具包发布到此代码库中，还可以在网上（如GitHub）找到各种其他项目和工具包。

小结

我们在本章展示了如何将Spark应用到业务与技术工作中的多种方法，Spark简单强大的编程模型使其可以轻松应用于各种问题的处理，而由数百名开发人员创建的大量的工具包帮助Spark处理大量的业务问题与挑战。随着生态系统和社区的增长，越来越多的工具包可能会持续出现，我们期待看到这个社区的发展！

本书的其余部分将深入介绍图3-1中的模块。

你可以按照你喜欢的方式阅读本书的其余部分，我们发现，大多数人在听到某一术语后会跳跃到对应区域阅读，又或者，他们会根据自己在Spark应用中遇到的实际问题来选择对应的章节阅读。

第 II 部分

结构化API——DataFrame、SQL和Dataset

第4章

结构化API概述

本书的这一部分将深入探讨Spark的结构化API。结构化API是处理各种数据类型的工具，可处理非结构化的日志文件、半结构化的CSV文件以及高度结构化的Parquet文件。结构化API指以下三种核心分布式集合类型的API：

- Dataset类型。
- DataFrame类型。
- SQL表和视图。

虽然这三种类型将在本书的不同部分介绍，但大多数结构化API均适用于批处理和流处理，这意味着使用结构化API编写代码时，几乎不费吹灰之力就可以从批处理程序转换为流处理程序（反之亦然）。流处理将在第五部分进行详细介绍。

结构化API是在编写大部分数据处理程序时会用到的基础抽象概念。到目前为止，本书一直根据Spark所包含的内容循序渐进地讲述，而该部分则会进行更深层次地探索。在本章中，首先介绍需要理解的一些基本概念：类型化和非类型化的API，以及它们的差异；核心术语；Spark如何将结构化API数据流实际应用到集群上并执行它。然后，我们将提供更具体的相关任务信息来处理某些类型的数据或数据源。

在进行下一步之前，先回顾一下在第一部分中介绍的基本概念和定义。Spark是一个分布式编程模型，用户可以在其中指定转换操作。多次转换操作后建立起指令的有向无环图。指令图的执行过程作为一个作业由一个动作操作触发，在执行过程中一个作业被分解为多个阶段和任务在集群上执行。转换操作和动作操作操纵的逻辑结构是DataFrame和Dataset，执行一次转换操作会都会创建

一个新的DataFrame或Dataset，而动作操作则会触发计算，或者将DataFrame和Dataset转换成本地语言类型。

DataFrame类型和Dataset类型

第一部分中对DataFrame类型进行了讨论。Spark支持两种结构化集合类型：DataFrame和Dataset。接下来先了解它们各自的定义，再介绍它们之间的细微差别。

DataFrame和Dataset是具有行和列的类似于（分布式）数据表的集合类型。所有列的行数相同（可以使用null来指定缺省值），并且某一列的类型必须在所有行中保持一致。Spark中的DataFrame和Dataset代表不可变的数据集合，可以通过它指定对特定位置数据的操作，该操作将以惰性评估方式执行。当对DataFrame执行动作操作时，将触发Spark执行具体转换操作并返回结果，这些代表了如何操纵行和列来计算出用户期望结果的执行计划。

表和视图与DataFrame基本相同，所以我们通常在DataFrame上执行SQL操作，而不是用DataFrame专用的执行代码。在第10章中将专门介绍Spark SQL的相关内容。

为了使这些定义更具体，需要先讨论一下Schema数据模式，Schema数据模式定义了该分布式集合中存储的数据类型。

Schema

Schema定义了DataFrame的列名和类型，可以手动定义或者从数据源读取模式（通常定义为模式读取）。Schema数据模式需要指定数据类型，这意味着你需要指定在什么地方放置什么类型的数据。

结构化Spark类型概述

Spark实际上有它自己的编程语言，Spark内部使用一个名为Catalyst的引擎，在计划制订和执行作业的过程中使用Catalyst来维护它自己的类型信息，这样就会带来很大的优化空间，这些优化可以显著提高性能。Spark类型直接映射到不同语言API，并且针对Scala、Java、Python、SQL和R语言，都有一个对应的API查找表。即使通过Python或R语言来使用Spark结构化API，大多数情况下也是操作Spark类型而非Python类型。

例如，以下代码不会在Scala或Python中执行加法，而实际上完全是在Spark中执行加法：

```
// in Scala
val df = spark.range(500).toDF("number")
df.select(df.col("number") + 10)

# in Python
df = spark.range(500).toDF("number")
df.select(df["number"] + 10)
```

这样执行加法操作是因为Spark会将用输入语言编写的表达式转换为代表相同功能的Spark内部Catalyst表示。然后，它将根据该内部表示进行操作。在讨论为什么会出现这种情况之前，我们先来讨论Dataset类型。

DataFrame与 Dataset的比较

实质上，结构化API包含两类API，即非类型化的DataFrame和类型化的Dataset。说DataFrame是无类型的可能不太准确，因为它们其实是有类型的，只是Spark完全负责维护它们的类型，仅在运行时检查这些类型是否与schema中指定的类型一致。与之相对应的，Dataset在编译时就会检查类型是否符合规范。Dataset仅适用于基于Java虚拟机（JVM）的语言（比如Scala和Java），并通过case类或Java beans指定类型。

因此在大多数情况下，你会使用DataFrame。在Scala版本的Spark中，DataFrame就是一些Row类型的Dataset的集合。“Row”类型是Spark用于支持内存计算而优化的数据格式。这种格式有利于高效计算，因为它避免使用会带来昂贵垃圾回收开销和对象实例化开销的JVM类型，而是基于自己的内部格式运行，所以并不会产生这种开销。Python版本和R语言版本的Spark并不支持Dataset，所有东西都是DataFrame，这样我们就可以使用这种优化的数据格式进行计算处理。

内部Catalyst格式在很多Spark演示中都有详细介绍。鉴于本书面向的是广大普通读者，因此我们不会涉及到它的具体实现。如果你感兴趣的话，可以参考Databricks公司的Josh Rosen 和Herman van Hovell的一些很棒的报告，里面介绍了他们开发Spark的Catalyst引擎方面的工作。

充分理解DataFrame、Spark类型和Schema模式是需要一些时间的，需要强调的是，当使用DataFrame时，你就会大大受益于这种优化过的Spark内部格式。这种格式对所有Spark支持的语言API都具有相同的效率增益。如果你对性能有严格要求，请阅读第11章以了解更多信息。

接下来，介绍一些相对来说更熟悉的概念：列和行。

列

列表示一个简单类型（例如,整数或字符串），或者一个复杂类型(例如，数组或map映射)，或者空值null。Spark记录所有这些类型的信息并提供多种转换方法。第5章将对列类型进行更广泛的讨论，简单来说，你可以将Spark列想象为一个数据表的列即可。

行

一行对应一个数据记录。正如我们在下面对DataFrame调用collect方法时所看到的，DataFrame中的每条记录都必须是Row类型。我们可以通过SQL手动创建、或者从弹性分布式数据集（RDD）提取、或从数据源手动创建这些行。下面的示例使用范围函数range()来创建一个row对象：

```
// in Scala
spark.range(2).toDF().collect()

# in Python
spark.range(2).collect()
```

执行这两行会各自产生一个Row对象的数组。

Spark类型

之前提到过Spark有大量的内部类型表示。在接下来的内容中提供了一个方便查阅的参考表，它标示了在你使用语言中的某数据类型和与之相对应的Spark类型。

首先我们讨论一下如何实例化一个列为特定类型，或如何声明一个列为特定类型。

通过以下方法可以使用正确的Scala类型：

```
import org.apache.spark.sql.types._ val
b = ByteType
```

利用如下的工厂方法来使用正确的Java类型：

```
import org.apache.spark.sql.types.DataTypes;
ByteType x = DataTypes.ByteType;
```

有时Python类型有一定的要求，在表4-1中列出；Scala和Java中也是如此，分别在表4-2 和表4-3中列出。用如下方法使用正确的Python类型：

```
from pyspark.sql.types import *
b = ByteType()
```

以下几个表提供了Spark对应于每种语言规范的详细类型信息。

表4-1：Python类型参考表

数据类型	Python的值类型	获取或者创建数据类型的API
ByteType	int或long。注意：数字在运行时转换为1字节的带符号整数。 确保数字在-128~127的范围内	ByteType()
ShortType	int或long。注意：数字在运行时将转换为2字节带符号的整数。 确保数字在-32768到32767的范围内	ShortType()
IntegerType	int或long。注意：Python对“整数”有一个宽松的定义。如果使用IntegerType()，那么太大的数字将被Spark SQL拒绝。在这种情况下，最好使用LongType()	IntegerType()
LongType	long。注意：数字在运行时将转换为8字节有符号整数。确保数字在-9223372036854775808~9223372036854775807范围内。否则，请将数据类型转换为decimal.Decimal，并使用DecimalType	IntegerType()
FloatType	float型。注意：在运行时，数字将被转换为4字节的单精度浮点数	FloatType()
DoubleType	float型	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	Bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	List，tuple或array	ArrayType（elementType，[containsNull]）。注意：containsNull的默认值为True

表4-1：Python类型参考表（续）

数据类型	Python的值类型	获取或者创建数据类型的API
MapType	字典	MapType（keyType，valueType，[valueContainsNull]）。注意：valueContainsNull的默认值为True
StructType	列表或元组	StructType（fields）。注意：fields是一个包含多个StructFiled的list，并且任意两个StructField不能同名
StructField	该字段对应的Python数据类型（例如，int是IntegerType的StructField）	StructField（name，dataType，[nullable]）。注意：nullable指定该field是否可以为空值，默认值为True

表4-2：Scala类型参考表

数据类型	Scala的值类型	获取或创建数据类型的API
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Int	IntegerType
LongType	Long	LongType
FloatType	Float	FloatType
DoubleType	Double	DoubleType
DecimalType	java.math.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array[Type]	BinaryType
BooleanType	Boolean	BooleanType
TimestampType	java.sql.TimeStamp	TimestampType
DataType	Java.sql.Date	DateType
ArrayType	scala.collection.Seq	ArrayType(elementType, [containsNull])。注意：containsNull的默认值为true

表4-2：Scala类型参考表（续）

数据类型	Scala的值类型	获取或创建数据类型的API
MapType	scala.collection.Map	MapType（keyType，valueType，[valueContainsNull]）。注意：valueContainsNull的默认值为true
StructType	org.apache.spark.sql.Row	StructType（fields）。注意：fields是一个包含多个StructField的Array，并且任意两个StructField不能同名
StructField	该字段对应的Scala数据类型（例如，int是IntegerType的StructField）	StructField（name，dataType，[nullable]）。注意：nullable的默认值为true

表4-3：Java类型参考表

数据类型	Java中的值类型	获取或创建数据类型的API
ByteType	byte或Byte	DataTypes.ByteType
ShortType	short或Short	DataTypes.ShortType
IntegerType	int或Integer	DataTypes.IntegerType
LongType	long或Long	DataTypes.LongType
FloatType	float或Float	DataTypes.FloatType
DoubleType	double或Double	DataTypes.DoubleType
DecimalType	java.math.BigDecimal	DataTypes.createDecimalTyp()DataTypes.createDecimalType（precision，scale）
StringType	String	DataTypes.StringType
BinaryType	byte[]	DataTypes.BinaryType
BooleanType	boolean或Boolean	DataTypes.BooleanType
TimestampType	java.sql.Timestamp	DataTypes.TimestampType
DateType	java.sql.Date	DataTypes.DateType
ArrayType	java.util.List	DataTypes.createArrayType(elementType)。注意：containsNull的值将为true。DataTypes.createArrayType(elementType，containsNull)
MapType	java.util.Map	DataTypes.createMapType（keyType，valueType）。注意：valueContainsNull的值将为true。DataTypes.createMapType（keyType，valueType，valueContainsNull）
StructType	org.apache.spark.sql.Row	DataTypes.createStructType（fields）。注意：fields是一个包含多个StructField的Array，并且任意两个StructField不能同名
StructField	该字段对应的Scala数据类型（例如，int是IntegerType的StructField）	DataTypes.createStructField（name, dataType, nullable）

值得注意的是，随着时间的推移，类型可能会随着Spark SQL版本的更新而不断变化，因此针对未来可能更新的情况应该参考Spark的官方文档。当然，所有这些类型都很有用，但是你几乎不会用到纯静态的DataFrame对象，你将会一直操作和转换它们。因此，接下来介绍结构化API中的执行过程。

结构化API执行概述

本节将演示用户代码是如何在集群上执行的，这对理解编写代码过程和在集群上执行代码的过程都很有帮助，也有助于后期的调试。因此接下来针对一个结构化API查询任务，我们逐步分析从用户代码到执行代码的过程。步骤如下：

- 编写DataFrame / Dataset / SQL代码。
- 如果代码能有效执行，Spark将其转换为一个逻辑执行计划（Logical Plan）。
- Spark将此逻辑执行计划转化为一个物理执行计划（Physical Plan），检查可行的优化策略，并在此过程中检查优化。
- Spark在集群上执行该物理执行计划（RDD操作）。

我们编写的代码通过控制台提交给Spark，或者以一个Spark作业的形式提交。然后代码将交由Catalyst优化器决定如何执行，并指定一个执行计划。最后代码被运行，得到的结果返回给用户。图4-1展示了整个过程。

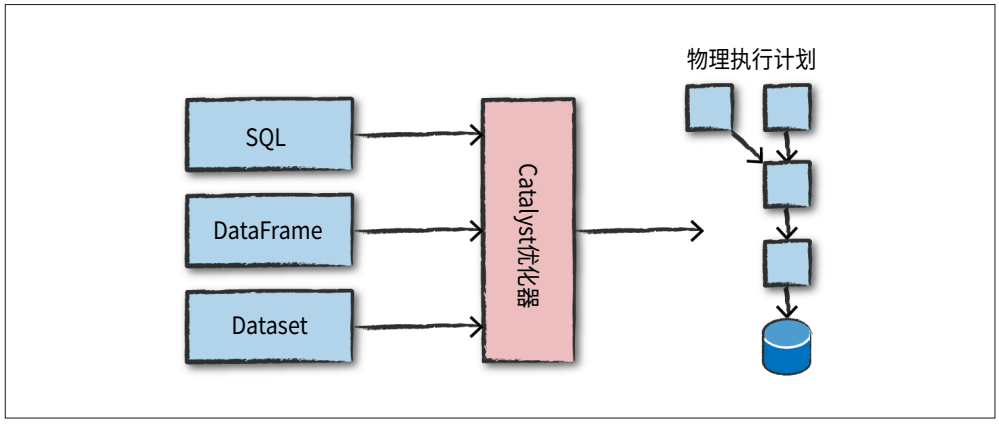


图4-1：Catalyst优化器

逻辑计划

执行的第一阶段旨在获取用户代码并将其转换为逻辑计划。图4-2展示了该过程。

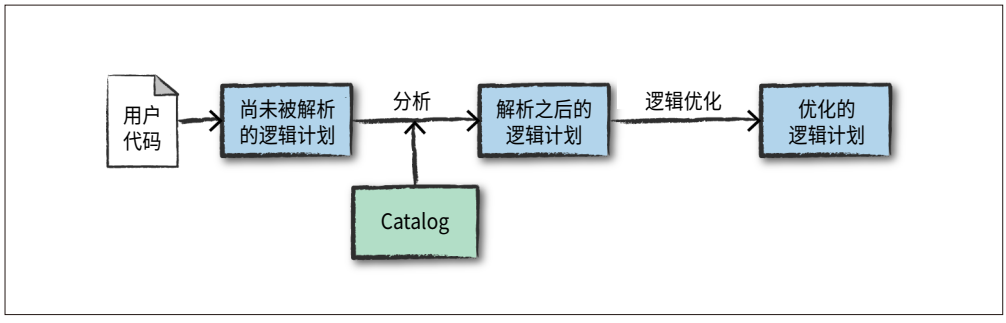


图4-2：结构化API的逻辑计划

这个逻辑计划仅代表一组抽象转换，并不涉及执行器或驱动器，它只是将用户的表达式集合转换为最优的版本。它通过将用户代码转换为未解析的逻辑计划来实现这一点。这个计划没有解析，因为虽然你的代码可能是有效的，但它引用的表或列可能存在也可能不存在。Spark使用catalog（所有表和DataFrame信息的存储库）在分析器中解析列和表格。如果目录中不存在所需的表或列名称，分析器可能会拒绝该未解析的逻辑计划。如果分析器可以解析它，结果将通过Catalyst优化器，Catalyst优化器尝试通过下推谓词或选择操作来优化逻辑计划。用户也可以扩展Catalyst优化器来支持自己的特定领域优化策略。

物理计划

在成功创建优化的逻辑计划后，Spark开始执行物理计划流程。物理计划（通常称为Spark计划）通过生成不同的物理执行策略，并通过代价模型进行比较分析，从而指定如何在集群上执行逻辑计划，具体流程如图4-3所示。例如执行一个连接操作就会涉及代价比较，它通过分析数据表的物理属性（表的大小或分区的大小），对不同的物理执行策略进行代价比较，选择合适的物理执行计划。

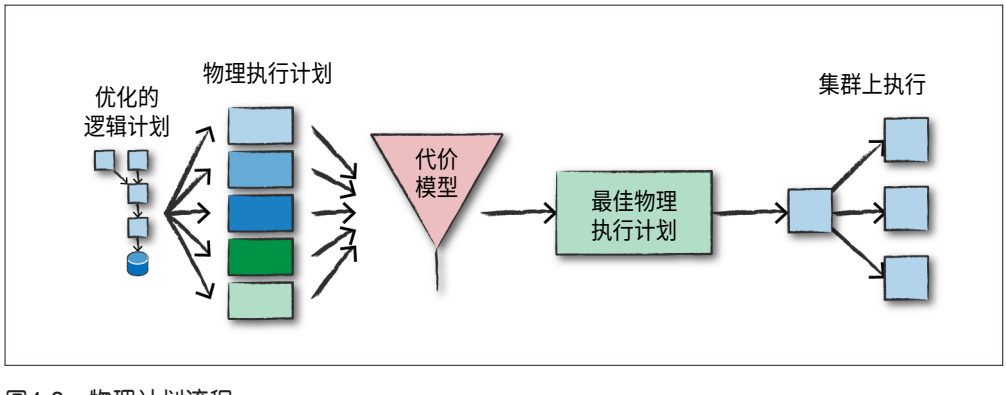


图4-3：物理计划流程

物理规划产生一系列的RDD和转换操作。这就是Spark被称为编译器的原因，因为它将对DataFrame、Dataset和SQL中的查询来作为你编译一系列RDD的转换操作。

执行

在选择一个物理计划时，Spark将所有代码运行在Spark的底层编程接口RDD上（第Ⅲ部分将会介绍）。Spark在运行时执行进一步优化，生成可以在执行期间优化任务或阶段的本地Java字节码，最终将结果返回给用户。

小结

在本章中，我们介绍了Spark结构化API以及Spark如何将你的代码转换为在集群上物理执行的代码。接下来的章节中，将介绍一些核心概念以及如何使用结构化API的关键功能。

第5章

基本的结构化操作

在第4章，我们已经介绍了关于结构化API的核心抽象。本章将重点介绍DataFrame的操作，而聚合操作（aggregation）、窗口函数（window function），以及连接操作（join）将在后面的章节中逐一介绍。

DataFrame由记录（record）组成，record是Row类型（与一个table中行相似）。一条record由多列组成（类似于一个电子表格中的列），列表示可以在该Dataset中每个单独的记录上执行的计算表达式。模式定义了DataFrame列的名以及列的数据类型。DataFrame的分区定义了DataFrame以及Dataset在集群上的物理分布，而划分模式定义了partition的分配方式，你可以自定义分区的方式，也可以采取随机分配的方式。

下面为一个DataFrame创建示例：

```
// in Scala
val df = spark.read.format("json")
.load("/data/flight-data/json/2015-summary.json")

# in Python
df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
```

上面已经介绍了DataFrame有很多列，而模式定义了这些列的名字和数据类型。可以用下面的方法查询DataFrame的模式：

```
df.printSchema()
```

数据模式包含很多信息，接下来我们详细探讨Schema 。

模式

模式定义DataFrame的列名以及列的数据类型，它可以由数据源来定义模式（称为读时模式，schema-on-read），也可以由我们自己来显式地定义。

实际应用场景决定了定义Schema的方式。当应用于即席分析时，使用读时模式即可（尽管在处理如CSV和JSON等纯文本文件时速度较慢）。但是，这也可能导致数据精度损失问题，例如在文件中读取时，将long型错误地解析为整数。当使用Spark进行实现生产级别ETL（Extract提取、Transform转换操作、Load加载）的时候，最好采取显式定义Schema的方式，尤其是在处理诸如CSV和JSON之类的无类型数据源时更是如此，这是因为模式推断方法会根据读入数据类型而变化。

让我们从第4章中看到的一个简单文件开始，利用行分隔的JSON半结构化性质来定义这个结构。这是来自某国交通局的航班统计数据：

```
// in Scala
spark.read.format("json").load("/data/flight-data/json/2015-summary.json").schema
```

Scala返回以下内容：

```
org.apache.spark.sql.types.StructType = ...
StructType(StructField(DEST_COUNTRY_NAME,StringType,true),
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true))

# in Python
spark.read.format("json").load("/data/flight-data/json/2015-summary.json").schema
```

Python中返回以下内容：

```
org.apache.spark.sql.types.StructType = ...
StructType(StructField(DEST_COUNTRY_NAME,StringType,true),
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true))
```

一个模式是由许多字段构成的StructType。这些字段即为StructField，具有名称、类型、布尔标志（该标志指定该列是否可以包含缺失值或空值），并且用户可指定与该列关联的元数据（metadata）。元数据存储着有关此列的信息（Spark在其机器学习库中使用此功能）。

模式还包含其他的StructType（Spark的复杂类型），在第6章中讨论复杂类型的使用

时将看到这一点。如果（在运行时）数据的类型与定义的schema模式不匹配，Spark将抛出一个错误。以下示例演示了如何为一个DataFrame创建并指定模式。

```
// in Scala
import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}
import org.apache.spark.sql.types.Metadata

val myManualSchema = StructType(Array(
  StructField("DEST_COUNTRY_NAME", StringType, true),
  StructField("ORIGIN_COUNTRY_NAME", StringType, true),
  StructField("count", LongType, false,
    Metadata.fromJson("{\"hello\":\"world\"}"))
))

val df = spark.read.format("json").schema(myManualSchema)
  .load("/data/flight-data/json/2015-summary.json")
```

以下是如何在Python中执行相同的操作：

```
# in Python
from pyspark.sql.types import StructField, StructType, StringType, LongType

myManualSchema = StructType([
  StructField("DEST_COUNTRY_NAME", StringType(), True),
  StructField("ORIGIN_COUNTRY_NAME", StringType(), True),
  StructField("count", LongType(), False, metadata={"hello":"world"})
])
df = spark.read.format("json").schema(myManualSchema)\
  .load("/data/flight-data/json/2015-summary.json")
```

正如第4章所讨论的，因为Spark会维护它自己的类型信息，所以我们不能简单地通过每种语言的类型来设置类型。接下来讨论模式定义的列。

列和表达式

Spark中的列与电子表格、R dataframe或pandas DataFrame中的列类似， 可以对DataFrame中的列进行选择、转换操作和删除，并将这些操作表示为表达式。

对于Spark而言，列是逻辑结构，它只是表示根据表达式为每个记录计算出的值。这意味着要为一列创建一个真值，我们需要有一行，而要有一行则需要有一个DataFrame。你不能在DataFrame的范围外操作一个列，必须对DataFrame使用Spark的转换操作来修改列的内容。

列

有很多不同的方法来构造和引用列，两个最简单的方法是通过col函数或column函数。使用这两个函数，需要传入列名：

```
// in Scala
import org.apache.spark.sql.functions.{col, column}
col("someColumnName")
column("someColumnName")

# in Python
from pyspark.sql.functions import col, column
col("someColumnName")
column("someColumnName")
```

本书统一使用col函数。如前所述，DataFrame可能不包含某列，所以该列要将列名与catalog中维护的列名相比较之后才会确定该列是否会被解析。如第4章中所述，列和数据表的解析在分析器阶段发生。

> 刚提到了引用列的两种不同方法，而Scala中有一些特有的语言支持，可以使用更多简短的方式来引用列。下面的表达式执行相同的功能，即创建列，但并不能改善性能：
>
> ```
> // in Scala
> $"myColumn"
> 'myColumn
> ```
>
> 符号$将字符串指定为表达式，而符号（'）指定一个symbol，是Scala引用标识符的特殊结构。它们都执行相同的功能，通过列名引用列的简写方式。 阅读别人的Spark代码时，你可能会看到上述提到的各种不同的引用方式，你可以选择使用这些引用方法，选择那些你和你的同事认为好用的和维护性好的方法。

显式列引用

如果你需要引用某DataFrame的某一列，则可以在这个DataFrame上使用col方法。当执行连接操作时，如果两个连接的DataFrame存在一个同名列，该方法会非常有用，这在第8章将详细介绍。显式引用列的另一个好处就是Spark不用自己解析该列（在分析阶段）：

```
df.col("count")
```

表达式

我们之前介绍了列就是表达式，但什么是表达式？表达式（expression）是对一个

DataFrame中某一个记录的一个或多个值的一组转换操作。把它想象成一个函数，它将一个或多个列名作为输入，解析它们，然后针对数据集中的每条记录应用表达式来得到一个单值。这个“单值”实际上可以是一个复杂的类型，如Map或Array。我们将在第6章对复杂类型进行介绍。

在最简单的情况下，通过expr函数创建的表达式，仅仅是一个DataFrame列的引用，也就是说，expr("someCol")等同于col("someCol")。

列作为表达式

列提供了表达式功能的一个子集。如果你使用col()，并想对该列执行转换操作，则必须对该列的引用执行这些转换操作。当使用表达式时，expr函数实际上可以将字符串解析成转换操作和列引用，也可以在之后将其传递到下一步的转换操作。我们来看一些例子。

expr("someCol - 5")与执行col("someCol") -5，甚至是expr("someCol") - 5，都是相同的转换操作。Spark将它们编译为表示操作顺序的逻辑树。这可能容易混淆，但请记住下面两点：

- 列只是表达式。
- 列与对这些列的转换操作被编译后生成的逻辑计划，与解析后的表达式的逻辑计划是一样的。

以一个例子来说明这一点：

```
(((col("someCol") + 5) * 200) - 6) < col("otherCol")
```

图5-1概括描述了逻辑树的结构。

图5-1：逻辑树

逻辑树是一种有向无环图，该图等同于以下代码：

```
// in Scala
import org.apache.spark.sql.functions.expr
expr("(((someCol + 5) * 200) - 6) < otherCol")

# in Python
from pyspark.sql.functions import expr
expr("(((someCol + 5) * 200) - 6) < otherCol")
```

需要强调的是，上面的表达式是有效SQL代码，就像SELECT表达式。这是因为SQL与DataFrame代码在执行之前会编译成相同的底层逻辑树。这意味着SQL表达式和DataFrame代码的性能是一样的，这在第4章讨论过。

访问DataFrame的列

有时，可以使用printSchema查询DataFrame的列，但是你要想在程序中访问列，可以使用属性columns查询DataFrame的所有列：

```
spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
  .columns
```

记录和行

在Spark中，DataFrame的每一行都是一个记录，而记录是Row类型的对象。Spark使用列表达式操纵Row类型对象。Row对象内部其实是字节数组，但是Spark没有提供访问这些数组的接口，因此我们只能使用列表达式去操纵。

当使用DataFrame时，向驱动器请求行的命令总是返回一个或多个Row类型的行数据。

在本章中我们会交替地使用“行”和“记录”，它们代表同一个意思，多数情况下会使用后者。Row表示Row类型的对象。

我们可以通过在DataFrame上调用first()来查看一行：

```
df.first()
```

创建行

你可以基于已知的每列数值去手动实例化一个Row对象来创建行。需要注意的是，只有DataFrame具有模式，行对象本身没有模式，这意味着，如果你手动创建Row对象，则必须按照该行所附属的DataFrame的列顺序来初始化Row对象（我们将在讨论创建DataFrame时看到这一点）：

```
// in Scala
import org.apache.spark.sql.Row
val myRow = Row("Hello", null, 1, false)

# in Python
from pyspark.sql import Row
```

```
myRow = Row("Hello", None, 1, False)
```

访问行的数据也同样简单：你只需指定你想要的位置。使用Scala或Java时，你必须使用辅助方法或显式地指定值类型。而使用Python或者R时，该值将被自动转化为正确的类型：

```
// in Scala
myRow(0) // 任意类型
myRow(0).asInstanceOf[String] // 字符串
myRow.getString(0) // 字符串
myRow.getInt(2) // 整型

# in Python
myRow[0]
myRow[2]
```

你还可以使用Dataset的API通过相应的Java虚拟机（JVM）对象中显式返回一组Data数据。这在第11章中会介绍。

DataFrame转换操作

现在我们已经了解了DataFrame的核心部分，接下来将继续介绍如何处理DataFrame。处理DataFrame对象时通常涉及如下几个基本目标，这些可以被归纳为如下几个核心操作，如图5-2描述：

- 添加行或列。
- 删除行或列。
- 将一行转换操作为一列（或者将一列转换操作为一行）。
- 根据列中的值更改行的顺序。

幸运的是，我们可以把所有的这些操作转化成简单的转换操作，最常见的是，取一列，然后逐行更改，最后返回结果。

创建DataFrame

如前所述，我们可以从原始数据源中创建DataFrame，这在第9章会详细讨论。我们先来创建一个DataFrame示例（为了支持本章后面的示例，我们也将其注册为临时视图，以便可以用SQL访问并展示一些SQL中的基本转换操作）：

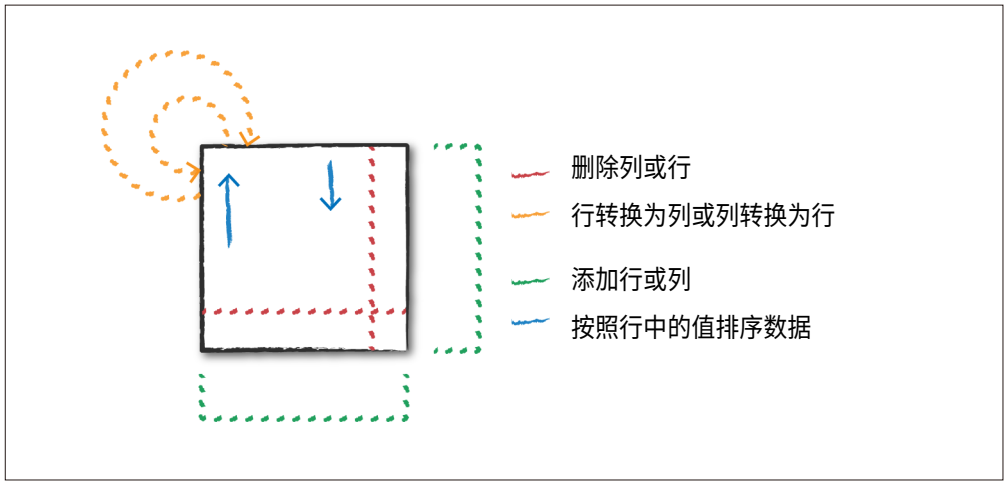


图5-2：不同种类的转换操作

```
// in Scala
val df = spark.read.format("json")
  .load("/data/flight-data/json/2015-summary.json")
df.createOrReplaceTempView("dfTable")

# in Python
df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
df.createOrReplaceTempView("dfTable")
```

我们也可以通过获取一组行并将它们转换操作为一个DataFrame来即时创建DataFrame。

```
// in Scala
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}

val myManualSchema = new StructType(Array(
  new StructField("some", StringType, true),
  new StructField("col", StringType, true),
  new StructField("names", LongType, false)))
val myRows = Seq(Row("Hello", null, 1L))
val myRDD = spark.sparkContext.parallelize(myRows)
val myDf = spark.createDataFrame(myRDD, myManualSchema)
myDf.show()
```

在Scala中，还能还可以利用Spark的隐式方法（使用implicit关键字），对Seq类型执行toDF函数来实现，由于对于null类型的支持并不稳定，所以并不推荐在实际生产中使用。

```
// in Scala
val myDF = Seq(("Hello", 2, 1L)).toDF("col1", "col2", "col3")

# in Python
from pyspark.sql import Row
from pyspark.sql.types import StructField, StructType, StringType, LongType
myManualSchema = StructType([
  StructField("some", StringType(), True),
  StructField("col", StringType(), True),
  StructField("names", LongType(), False)
])
myRow = Row("Hello", None, 1)
myDf = spark.createDataFrame([myRow], myManualSchema)
myDf.show()
```

输出如下：

```
+-----+----+-----+
| some| col|names|
+-----+----+-----+
|Hello|null|    1|
+-----+----+-----+
```

现在你已经知道如何创建DataFrame了，接下来看看DataFrame类型支持的最有用的方法：处理列或表达式时的select方法，以及处理字符串表达式时的selectExpr方法。当然有的转换操作不是针对列的操作方法，因此org.apache.spark.sql.functions包中包含一组函数方法用来提供额外支持。

借助这三个工具，你应该能够解决在DataFrame中可能会遇到的绝大多数转换操作的问题。

select函数和selectExpr函数

Select函数和selectExpr函数支持在DataFrame上执行类似数据表的SQL查询：

```
-- in SQL
SELECT * FROM dataFrameTable
SELECT columnName FROM dataFrameTable
SELECT columnName * 10, otherColumn, someOtherCol as c FROM dataFrameTable
```

简单来说，你可以使用select和selectExpr来操作DataFrame中的列。我们将通过DataFrame的一些示例来介绍各种不同的写法。最简单的方式就是使用select方法，待处理的列名字符串作为参数传递：

```
// in Scala
df.select("DEST_COUNTRY_NAME").show(2)
```

```
# in Python
df.select("DEST_COUNTRY_NAME").show(2)

-- in SQL
SELECT DEST_COUNTRY_NAME FROM dfTable LIMIT 2
```

输出如下：

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|    United States|
|    United States|
+-----------------+
```

你可以使用相同格式的查询来选择多个列，只需在select方法调用中添加更多的列名字符串即可：

```
// in Scala
df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)

# in Python
df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)

-- in SQL
SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME FROM dfTable LIMIT 2
```

输出如下：

```
+-----------------+-------------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|
+-----------------+-------------------+
|    United States|            Romania|
|    United States|            Croatia|
+-----------------+-------------------+
```

正如本章前面的“列和表达式”所述，你可以通过多种不同的方式引用列，而且这些方式可以等价互换：

```
// in Scala
import org.apache.spark.sql.functions.{expr, col, column}
df.select(
    df.col("DEST_COUNTRY_NAME"),
    col("DEST_COUNTRY_NAME"),
    column("DEST_COUNTRY_NAME"),
    'DEST_COUNTRY_NAME,
    $"DEST_COUNTRY_NAME",
    expr("DEST_COUNTRY_NAME"))
  .show(2)
```

```
# in Python
from pyspark.sql.functions import expr, col, column
df.select(
    expr("DEST_COUNTRY_NAME"),
    col("DEST_COUNTRY_NAME"),
    column("DEST_COUNTRY_NAME"))\
  .show(2)
```

一个常见的错误是混淆Column对象和字符串。例如，以下代码将导致编译错误：

```
df.select(col("DEST_COUNTRY_NAME"), "DEST_COUNTRY_NAME")
```

expr是我们目前使用到的最灵活的引用方式。它能够引用一列，也可以引用对列进行操纵的字符串表达式。为了说明这一点，我们先更改列名，然后使用AS关键字和列上的alias方法将名字重新改回去。

```
// in Scala
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)

# in Python
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)

-- in SQL
SELECT DEST_COUNTRY_NAME as destination FROM dfTable LIMIT 2
```

上面的操作将列名改为“destination”，你可以进一步操作：

```
// in Scala
df.select(expr("DEST_COUNTRY_NAME as destination").alias("DEST_COUNTRY_NAME"))
  .show(2)

# in Python
df.select(expr("DEST_COUNTRY_NAME as destination").alias("DEST_COUNTRY_NAME"))\
  .show(2)
```

上述操作将列名更改回原来的名称。

因为select后跟着一系列expr是非常常见的写法，所以Spark有一个有效地描述此操作序列的接口：selectExpr，它可能是最常用的接口：

```
// in Scala
df.selectExpr("DEST_COUNTRY_NAME as newColumnName", "DEST_COUNTRY_NAME").show(2)

# in Python
df.selectExpr("DEST_COUNTRY_NAME as newColumnName", "DEST_COUNTRY_NAME").show(2)
```

这是Spark最强大的地方，我们可以利用selectExpr构建复杂表达式来创建DataFrame。实际上，我们可以添加任何不包含聚合操作的有效SQL语句，并且只要

列可以解析，它就是有效的！下面是一个简单的例子，在DataFrame中增加一个新列withinCountry，该列描述了destination和origin是否相同：

```
// in Scala
df.selectExpr(
    "*", // 包含所有原始表中的列
    "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry")
  .show(2)

# in Python
df.selectExpr(
  "*", # all original columns
  "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry")\
  .show(2)

-- in SQL
SELECT *, (DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry
FROM dfTable
LIMIT 2
```

输出如下：

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|            Romania|   15|        false|
|    United States|            Croatia|    1|        false|
+-----------------+-------------------+-----+-------------+
```

使用select语句，我们还可以利用系统预定义好的聚合函数来指定在整个DataFrame上的聚合操作，代码示例如下所示：

```
// in Scala
df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))").show(2)

# in Python
df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))").show(2)

-- in SQL
SELECT avg(count), count(distinct(DEST_COUNTRY_NAME)) FROM dfTable LIMIT 2
```

输出如下：

```
+-----------+---------------------------------+
| avg(count)|count(DISTINCT DEST_COUNTRY_NAME)|
+-----------+---------------------------------+
|1770.765625|                              132|
+-----------+---------------------------------+
```

转换操作成Spark类型（字面量）

有时候需要给Spark传递显式的值，它们只是一个值而非新列。这可能是一个常量值，或接下来需要比较的值。我们的方式是通过字面量（literal）传递，简单来说，就是将给定的编程语言的字面上的值转换操作为Spark可以理解的值。字面量就是表达式，你可以用操作表达式的方式来使用它们：

```
// in Scala
import org.apache.spark.sql.functions.lit
df.select(expr("*"), lit(1).as("One")).show(2)

# in Python
from pyspark.sql.functions import lit
df.select(expr("*"), lit(1).alias("One")).show(2)
```

在SQL中，字面量只是特定的值：

```
-- in SQL
SELECT *, 1 as One FROM dfTable LIMIT 2
```

输出：

```
+-----------------+-------------------+-----+---+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|One|
+-----------------+-------------------+-----+---+
|    United States|            Romania|   15|  1|
|    United States|            Croatia|    1|  1|
+-----------------+-------------------+-----+---+
```

当你需要比较一个值是否大于一个常量或者程序创建的变量时，可以使用上面的方法。

添加列

使用`WithColumn`可以为DataFrame增加新列，这种方式更为正式一些。例如，添加一个仅包含数字1的列：

```
// in Scala
df.withColumn("numberOne", lit(1)).show(2)

# in Python
df.withColumn("numberOne", lit(1)).show(2)

-- in SQL
SELECT *, 1 as numberOne FROM dfTable LIMIT 2
```

输出如下：

```
+-----------------+-------------------+-----+---------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|numberOne|
+-----------------+-------------------+-----+---------+
|    United States|            Romania|   15|        1|
|    United States|            Croatia|    1|        1|
+-----------------+-------------------+-----+---------+
```

让我们做一些更有趣的事，来接触一下实际的表达式。在下一个示例中，当出发国家与目的地国家相同时，我们将为其设置一个布尔标志：

```
// in Scala
df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME"))
  .show(2)

# in Python
df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME"))\
  .show(2)
```

请注意，withColumn函数有两个参数：列名和为给定行赋值的表达式。我们也可以用WithColumn对某一列重命名。SQL语法与之前介绍的一样，所以这个例子省略SQL代码：

```
df.withColumn("Destination", expr("DEST_COUNTRY_NAME")).columns
```

结果是：

```
... DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count, Destination
```

重命名列

我们不仅可以使用WithColumn对列重命名，还可以使用WithColumnRenamed方法实现对列重命名。WithColumnRenamed中，第一个参数是要被修改的列的名，第二个参数是新的列名。

```
// in Scala
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns

# in Python
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns

... dest, ORIGIN_COUNTRY_NAME, count
```

保留字与关键字

你可能会遇到列名中包含空格或者连字符等保留字符，要处理这些保留字符意味着要适当地对列名进行转义。在Spark中，我们通过使用反引号（`）字符来实现。

withColumn是你刚学会的一个允许使用保留字来创建列的方法。接下来介绍两个示例：在第一个示例中，我们不需要转义字符，但在第二个示例中，我们则会用到。

```
// in Scala
import org.apache.spark.sql.functions.expr

val dfWithLongColName = df.withColumn(
  "This Long Column-Name",
  expr("ORIGIN_COUNTRY_NAME"))

# in Python
dfWithLongColName = df.withColumn(
    "This Long Column-Name",
    expr("ORIGIN_COUNTRY_NAME"))
```

这里不需要转义字符，因为withColumn的第一个参数只是新列名的字符串。但在下面这个例子中，我们需要使用反引号，因为我们在表达式中引用了一个列：

```
// in Scala
dfWithLongColName.selectExpr(
    "`This Long Column-Name`",
    "`This Long Column-Name` as `new col`")
  .show(2)

# in Python
dfWithLongColName.selectExpr(
    "`This Long Column-Name`",
    "`This Long Column-Name` as `new col`")\
  .show(2)

dfWithLongColName.createOrReplaceTempView("dfTableLong")

-- in SQL
SELECT `This Long Column-Name`, `This Long Column-Name` as `new col`
FROM dfTableLong LIMIT 2
```

如果我们显式地使用字符串来引用列，则可以引用带有保留字符的类（而不用转义他们），这个字符串会被解释成字面值，而不是表达式。我们只需要转义使用保留字符或者关键字的表达式。下面两个例子会产生相同的DataFrame：

```
// in Scala
dfWithLongColName.select(col("This Long Column-Name")).columns

# in Python
dfWithLongColName.select(expr("`This Long Column-Name`")).columns
```

区分大小写

Spark默认是不区分大小写的，但可以通过如下配置使Spark区分大小写：

```
-- in SQL
set spark.sql.caseSensitive true
```

删除列

现在我们已经创建了列，接下来看看如何从DataFrame中删除列。你可能已经注意到，我们可以通过select实现。但是也可以使用drop方法来删除列：

```
df.drop("ORIGIN_COUNTRY_NAME").columns
```

可以通过传入多个列作为参数来实现删除多个列：

```
dfWithLongColName.drop("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME")
```

更改列的类型（强制类型转换）

有时，我们可能需要将某一列类型转换操作为另一种类型。例如有一组StringType类型集合，但是它们本应该是整型，我们可以通过更改列的类型来转换数据类型。例如，下面我们将count列从integer整型转换操作成Long长整型：

```
df.withColumn("count2", col("count").cast("long"))
```

```
-- in SQL
SELECT *, cast(count as long) AS count2 FROM dfTable
```

过滤行

为了过滤行，只要创建一个表达式来判断该表达式是true还是false，然后过滤使表达式为false的行。在DataFrame上实现过滤操作最常见的方法是创建一个字符串表达式，或者通过列操作来构建表达式。有两种实现过滤的方式，分别是where和filter，它们可以执行相同的操作，接受相同参数类型。在本书中我们一直使用where，因为这更像SQL语法，但filter也是有效的。

在Scala或Java中使用Dataset API时，filter的输入也可以是能够作用到Dataset每条记录上的任意函数。想了解更多信息，请参阅第11章 。

以下过滤器是等效的，结果在Scala和Python中是一样的：

```
df.filter(col("count") < 2).show(2)
df.where("count < 2").show(2)
```

```
-- in SQL
SELECT * FROM dfTable WHERE count < 2 LIMIT 2
```

输出如下：

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Croatia|    1|
|    United States|          Singapore|    1|
+-----------------+-------------------+-----+
```

我们可能本能地想把多个过滤条件放到一个表达式中，尽管这种方式可行，但是并不总有效。因为Spark会同时执行所有过滤操作，不管过滤条件的先后顺序，因此当你想指定多个AND过滤操作时，只要按照先后顺序以链式的方式把这些过滤条件串联起来，然后让Spark执行剩下的工作：

```
// in Scala
df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") =!= "Croatia")
  .show(2)

# in Python
df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") != "Croatia")\
  .show(2)

-- in SQL
SELECT * FROM dfTable WHERE count < 2 AND ORIGIN_COUNTRY_NAME != "Croatia"
LIMIT 2
```

输出如下：

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|          Singapore|    1|
|          Moldova|      United States|    1|
+-----------------+-------------------+-----+
```

获得去重后的行

一个常见的应用场景是去除DataFrame中重复的行，这可以去除一列或者多列中重复的值。我们实现去重的方式是使用DataFrame的distinct方法，它能够对DataFrame中的行进行去重操作。例如，从数据集中找出所有不同的出发国家-目的地国家组合或者所有不同的出发国家，这是一个转换操作，它将返回去重后的（不包含重复行的）DataFrame：

```
// in Scala
df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").distinct().count()

# in Python
df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").distinct().count()

-- in SQL
SELECT COUNT(DISTINCT(ORIGIN_COUNTRY_NAME, DEST_COUNTRY_NAME)) FROM dfTable
```

结果为256。

```
// in Scala
df.select("ORIGIN_COUNTRY_NAME").distinct().count()

# in Python
df.select("ORIGIN_COUNTRY_NAME").distinct().count()

-- in SQL
SELECT COUNT(DISTINCT ORIGIN_COUNTRY_NAME) FROM dfTable
```

结果为125。

随机抽样

有时可能想从DataFrame中随机抽取一些记录，你可以使用DataFrame的`sample`方法来实现此操作，它按一定比例从DataFrame中随机抽取一部分行，也可以通过`withReplacement`参数指定是否放回抽样， true为有放回的抽样（可以有重复样本），false为无放回的抽样（无重复样本）：

```
val seed = 5
val withReplacement = false
val fraction = 0.5
df.sample(withReplacement, fraction, seed).count()

# in Python
seed = 5
withReplacement = False
fraction = 0.5
df.sample(withReplacement, fraction, seed).count()
```

输出为126。

随机分割

当需要将原始DataFrame随机分割成多个分片时，可以使用随机分割。这通常是在机器学习算法中，用于分割数据集来创建训练集、验证集和测试集。在下一个示例中，

我们要设置分割比例（随机分割函数的参数）来将我们的DataFrame分割成两个不同的DataFrame。由于随机分割是一种随机方法，所以我们还需要指定一个随机种子（只需在代码中用特定数字来替换seed）。需要注意的是，如果一个DataFrame的分割比例的和不为1，则比例参数会被自动归一化：

```
// in Scala
val dataFrames = df.randomSplit(Array(0.25, 0.75), seed)
dataFrames(0).count() > dataFrames(1).count() // 结果为False

# in Python
dataFrames = df.randomSplit([0.25, 0.75], seed)
dataFrames[0].count() > dataFrames[1].count() # 结果为False
```

连接和追加行（联合操作）

正如在上一节看到的，DataFrame是不可变的，这意味着用户不能向DataFrame追加行。如果想要向DataFrame追加行，你必须将原始的DataFrame与新的DataFrame联合起来，即union操作，也就是拼接两个DataFrame。若想联合两个DataFrame，你必须确保它们具有相同的模式和列数，否则联合操作将会失败。

目前，联合操作是基于位置而不是基于数据模式schema执行，也就是说它并不会自动根据列名匹配对齐后再进行联合，所以两个联合的DataFrame需要具有完全相同的模式和列数。

```
// in Scala
import org.apache.spark.sql.Row
val schema = df.schema
val newRows = Seq(
  Row("New Country", "Other Country", 5L),
  Row("New Country 2", "Other Country 3", 1L)
)
val parallelizedRows = spark.sparkContext.parallelize(newRows)
val newDF = spark.createDataFrame(parallelizedRows, schema)
df.union(newDF)
  .where("count = 1")
  .where($"ORIGIN_COUNTRY_NAME" =!= "United States")
  .show() // get all of them and we'll see our new rows at the end
```

在Scala中，你需要使用=!= 运算符，这是因为它不仅能比较字符串，也能够比较表达式：

```
# in Python
from pyspark.sql import Row
schema = df.schema
```

```
newRows = [
  Row("New Country", "Other Country", 5L),
  Row("New Country 2", "Other Country 3", 1L)
]
parallelizedRows = spark.sparkContext.parallelize(newRows)
newDF = spark.createDataFrame(parallelizedRows, schema)
# in Python
df.union(newDF)\
  .where("count = 1")\
  .where(col("ORIGIN_COUNTRY_NAME") != "United States")\
  .show()
```

输出如下：

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Croatia|    1|
...
|    United States|            Namibia|    1|
|    New Country 2|    Other Country 3|    1|
+-----------------+-------------------+-----+
```

当DataFrame追加了记录后，需要对产生的新DataFrame进行引用。一个常见的方式是将这个新DataFrame变成视图（View）或者注册成一个数据表，以便在代码中使用。

行排序

当对DataFrame中的值进行排序时，我们通常是想要获得DataFrame里的一些最大值或者最小值。`sort`和`orderBy`方法是相互等价的操作，执行的方式也一样。它们均接收列表达式和字符串，以及多个列。默认设置是按升序排序：

```
// in Scala
df.sort("count").show(5)
df.orderBy("count", "DEST_COUNTRY_NAME").show(5)
df.orderBy(col("count"), col("DEST_COUNTRY_NAME")).show(5)

# in Python
df.sort("count").show(5)
df.orderBy("count", "DEST_COUNTRY_NAME").show(5)
df.orderBy(col("count"), col("DEST_COUNTRY_NAME")).show(5)
```

若要更明确地指定升序或是降序，则需使用`asc`函数和`desc`函数：

```
// in Scala
import org.apache.spark.sql.functions.{desc, asc}
df.orderBy(expr("count desc")).show(2)
df.orderBy(desc("count"), asc("DEST_COUNTRY_NAME")).show(2)
```

```
# in Python
from pyspark.sql.functions import desc, asc
df.orderBy(expr("count desc")).show(2)
df.orderBy(col("count").desc(), col("DEST_COUNTRY_NAME").asc()).show(2)

-- in SQL
SELECT * FROM dfTable ORDER BY count DESC, DEST_COUNTRY_NAME ASC LIMIT 2
```

一个高级技巧是你可以指定空值在排序列表中的位置，使用asc_nulls_first指示空值安排在升序排列的前面，使用desc_nulls_first指示空值安排在降序排列的前面，使用asc_nulls_last指示空值安排在升序排列的后面，使用desc_nulls_last指示空值安排在降序排列的后面。

出于性能优化的目的，最好是在进行别的转换之前，先对每个分区进行内部排序。可以使用sortWithinPartitions方法实现这一操作：

```
// in Scala
spark.read.format("json").load("/data/flight-data/json/*-summary.json")
  .sortWithinPartitions("count")
# in Python
spark.read.format("json").load("/data/flight-data/json/*-summary.json")\
  .sortWithinPartitions("count")
```

我们将在第III部分讨论性能调优时，更详细地介绍这个问题。

limit方法

通常，你可能想限制从DataFrame中提取的内容。例如，你可能只需要某个DataFrame的前十条记录。你可以使用limit方法实现这一点：

```
// in Scala
df.limit(5).show()

# in Python
df.limit(5).show()

-- in SQL
SELECT * FROM dfTable LIMIT 6

// in Scala
df.orderBy(expr("count desc")).limit(6).show()

# in Python
df.orderBy(expr("count desc")).limit(6).show()

-- in SQL
SELECT * FROM dfTable ORDER BY count desc LIMIT 6
```

重划分和合并

另一个重要的优化是根据一些经常过滤的列对数据进行分区，控制跨群集数据的物理布局，包括分区方案和分区数。

不管是否有必要，重新分区都会导致数据的全面洗牌。如果将来的分区数大于当前的分区数，或者当你想要基于某一组特定列来进行分区时，通常只能重新分区：

```
// in Scala
df.rdd.getNumPartitions // 1

# in Python
df.rdd.getNumPartitions() # 1

// in Scala
df.repartition(5)

# in Python
df.repartition(5)
```

如果你知道你经常按某一列执行过滤操作，则根据该列进行重新分区是很有必要的：

```
// in Scala
df.repartition(col("DEST_COUNTRY_NAME"))

# in Python
df.repartition(col("DEST_COUNTRY_NAME"))
```

你还可以指定你想要的分区数量：

```
// in Scala
df.repartition(5, col("DEST_COUNTRY_NAME"))

# in Python
df.repartition(5, col("DEST_COUNTRY_NAME"))
```

而另一方面，合并操作（coalesce）不会导致数据的全面洗牌，但会尝试合并分区。下面的示例代码将基于目的地国家名的列将数据重新划分成5个分区，然后再合并它们（没有导致数据全面洗牌）：

```
// in Scala
df.repartition(5, col("DEST_COUNTRY_NAME")).coalesce(2)

# in Python
df.repartition(5, col("DEST_COUNTRY_NAME")).coalesce(2)
```

驱动器获取行

正如前几章所讨论的，Spark的驱动器维护着集群状态，有时候你需要让驱动器收集一些数据到本地，这样你可以在本地机器上处理它们。

到目前为止，我们并没有明确定义这个操作。但我们使用了几种不同的方法来实现完全相同的效果。下面的代码示例使用collect函数从整个DataFrame中获取所有数据，使用take函数选择前N行，并使用show函数打印一些行。

```
// in Scala
val collectDF = df.limit(10)
collectDF.take(5) // 获取整数行
collectDF.show() // 更友好的打印
collectDF.show(5, false)
collectDF.collect()

# in Python
collectDF = df.limit(10)
collectDF.take(5) # 获取整数行
collectDF.show() # 更友好的打印
collectDF.show(5, False)
collectDF.collect()
```

为了遍历整个数据集，还有一种让驱动器获取行的方法，即toLocalIterator函数。toLocalIterator函数式一个迭代器，将每个分区的数据返回给驱动器。这个函数允许你以串行的方式一个一个分区地迭代整个数据集：

```
collectDF.toLocalIterator()
```

将数据集合传递给驱动器的代价很高。当数据集很大时调用collect函数，可能会导致驱动器崩溃。如果使用toLocalIterator，并且分区很大，则容易使驱动器节点崩溃并丢失应用程序的状态，代价也是巨大的。因此我们可以一个一个分区进行操作，而不是并行运行。

小结

本章介绍了DataFrame的基本操作，讲解了一些有助于你学会使用Spark DataFrame的基本概念和工具。第6章将更为详细地介绍处理DataFrame的多种方法。

第6章

处理不同的数据类型

第5章介绍了关于DataFrame的基本概念和抽象。本章将介绍表达式的构建，这是Spark结构化操作的基础。还将介绍对不同数据类型的处理，主要包括以下内容：

- 布尔型。
- 数字型。
- 字符串型。
- 日期和时间戳类型。
- 空值处理。
- 复杂类型。
- 用户自定义的函数。

在哪里查找API

在正式介绍本章内容之前，我们先介绍用户查找转换操作的方法。Spark是一个不断成长的项目，所有关于Spark的书籍（包括本书）只能够记录一段时间的技术内容。因此，你应该知道如何查询数据操作的函数。这也是编写本书的目的之一。以下是几个主要查询的方法：

`DataFrame(Dataset)`方法

这有一个小窍门，因为DataFrame本质上就是一个`Row`类型的`Dataset`，所以你最终查看的就是`Dataset`的方法，通过该链接可以找到（*http://bit.ly/2rKkALY*）。

Dataset的子模块（如DataFrameStatFunctions（*http://bit.ly/2DPYhJC*）和DataFrameNaFunctions（*http://bit.ly/2DPAqd3*））有更多的解决具体问题的方法。例如，DataFrameStatFunctions包含许多统计相关的函数，而DataFrameNaFunctions包含处理空值相关的函数。

Column类型的方法

第5章中介绍了大部分Column类型的方法，其中包含各种与列相关的通用方法，例如alias或contains。可以在这里找到Column的方法API的链接（*http://bit.ly/2FloFbr*）。

org.apache.spark.sql.functions包含针对一系列不同数据类型的函数方法。因为这些函数方法经常被用到，所以这个包经常被整个导入代码中。可以在这里找到SQL和DataFrame 的函数方法（*http://bit.ly/2DPAycx*）。

这么多的函数方法会让我们感到茫然无措，但是不必担心，因为大部分函数可以在SQL和解析系统中找到。这些工具都是将一种数据格式或结构的数据行转换为另一种数据格式或结构的数据行，这可能会导致行数的增减。首先，让我们来读取数据以便后续分析使用：

```
// in Scala
val df = spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/data/retail-data/by-day/2010-12-01.csv")
df.printSchema()
df.createOrReplaceTempView("dfTable")

# in Python
df = spark.read.format("csv")\
  .option("header", "true")\
  .option("inferSchema", "true")\
  .load("/data/retail-data/by-day/2010-12-01.csv")
df.printSchema()
df.createOrReplaceTempView("dfTable")
```

以下是结果和一小部分数据示例：

```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: timestamp (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)
```

```
+---------+---------+--------------------+--------+-------------------+----...
|InvoiceNo|StockCode|         Description|Quantity|        InvoiceDate|Unit...
+---------+---------+--------------------+--------+-------------------+----...
|   536365|   85123A|WHITE HANGING HEA...|       6|2010-12-01 08:26:00|    ...
|   536365|    71053| WHITE METAL LANTERN|       6|2010-12-01 08:26:00|    ...
...
|   536367|    21755|LOVE BUILDING BLO...|       3|2010-12-01 08:34:00|    ...
|   536367|    21777|RECIPE BOX WITH M...|       4|2010-12-01 08:34:00|    ...
+---------+---------+--------------------+--------+-------------------+----...
```

转换成Spark类型

在本章中要做的一件事是将原始类型转换成Spark类型。我们要使用`lit`函数来实现这一点，它是我们介绍的第一个函数，该函数将把其他语言的类型转换为与其相对应的Spark表示。下面我们将几种不同类型的Scala和Python值转换为对应的Spark类型数据：

```
// in Scala
import org.apache.spark.sql.functions.lit
df.select(lit(5), lit("five"), lit(5.0))

# in Python
from pyspark.sql.functions import lit
df.select(lit(5), lit("five"), lit(5.0))
```

SQL中不需要等效函数，所以我们可以直接使用这些值：

```
-- in SQL
SELECT 5, "five", 5.0
```

处理布尔类型

布尔类型在数据分析中至关重要，因为它是所有过滤操作的基础。布尔语句由四个要素组成：*and*、*or*、*true*和 *false*。我们基于这些简单要素来构建返回值为true或false的逻辑语句，这些语句经常被作为过滤条件，如果某一行数据满足条件则执行操作，否则将被过滤掉。

我们基于零售数据来说明处理布尔类型的方法，可以在其中指定等于、小于或大于：

```
// in Scala
import org.apache.spark.sql.functions.col
df.where(col("InvoiceNo").equalTo(536365))
  .select("InvoiceNo", "Description")
  .show(5, false)
```

Scala有一些关于==和===用法的特殊语义。在Spark中，如果想通过相等条件来进行过滤，应该使用===（等于）或者=!=（不等于）符号，还可以使用not函数和equalTo方法来实现。

```
// in Scala
import org.apache.spark.sql.functions.col
df.where(col("InvoiceNo") === 536365)
  .select("InvoiceNo", "Description")
  .show(5, false)
```

Python使用更传统的运算符：

```
# in Python
from pyspark.sql.functions import col
df.where(col("InvoiceNo") != 536365)\
  .select("InvoiceNo", "Description")\
  .show(5, False)
```

```
+---------+-----------------------------+
|InvoiceNo|Description                  |
+---------+-----------------------------+
|536366   |HAND WARMER UNION JACK       |
...
|536367   |POPPY'S PLAYHOUSE KITCHEN    |
+---------+-----------------------------+
```

另一种方法是使用字符串形式的谓词表达式（该方法可能是最简洁的方法），Python或Scala支持这种方法。注意，这里使用了另一种表达“不等于”的方法。

```
df.where("InvoiceNo = 536365")
  .show(5, false)

df.where("InvoiceNo <> 536365")
  .show(5, false)
```

我们之前提到可以使用and或者or将多个Boolean表达式连接起来。但是在Spark中，最好是以链式连接的方式组合起来，形成顺序执行的过滤器。

这样做的原因是因为即使Boolean语句是顺序表达的（一个接着一个），Spark也会将所有这些过滤器合并为一条语句，并同时执行这些过滤器，创建and语句。尽管说你可以在语句中显式地使用and，但如果将它们串起来就更容易理解和阅读。or语句需要在同一个语句中指定：

```
// in Scala
val priceFilter = col("UnitPrice") > 600
val descripFilter = col("Description").contains("POSTAGE")
```

```
df.where(col("StockCode").isin("DOT")).where(priceFilter.or(descripFilter))
  .show()

# in Python
from pyspark.sql.functions import instr
priceFilter = col("UnitPrice") > 600
descripFilter = instr(df.Description, "POSTAGE") >= 1
df.where(df.StockCode.isin("DOT")).where(priceFilter | descripFilter).show()

-- in SQL
SELECT * FROM dfTable WHERE StockCode in ("DOT") AND(UnitPrice > 600 OR
instr(Description, "POSTAGE") >= 1)

+---------+---------+--------------+--------+-------------------+---------+...
|InvoiceNo|StockCode|   Description|Quantity|        InvoiceDate|UnitPrice|...
+---------+---------+--------------+--------+-------------------+---------+...
|   536544|      DOT|DOTCOM POSTAGE|       1|2010-12-01 14:32:00|   569.77|...
|   536592|      DOT|DOTCOM POSTAGE|       1|2010-12-01 17:06:00|   607.49|...
+---------+---------+--------------+--------+-------------------+---------+...
```

过滤器不一定非要使用Boolean表达式，要过滤DataFrame，也可以删掉指定一个Boolean布尔类型的列：

```
// in Scala
val DOTCodeFilter = col("StockCode") === "DOT"
val priceFilter = col("UnitPrice") > 600
val descripFilter = col("Description").contains("POSTAGE")
df.withColumn("isExpensive", DOTCodeFilter.and(priceFilter.or(descripFilter)))
  .where("isExpensive")
  .select("unitPrice", "isExpensive").show(5)

# in Python
from pyspark.sql.functions import instr
DOTCodeFilter = col("StockCode") == "DOT"
priceFilter = col("UnitPrice") > 600
descripFilter = instr(col("Description"), "POSTAGE") >= 1
df.withColumn("isExpensive", DOTCodeFilter & (priceFilter | descripFilter))\
  .where("isExpensive")\
  .select("unitPrice", "isExpensive").show(5)

-- in SQL
SELECT UnitPrice, (StockCode = 'DOT' AND
  (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1)) as isExpensive
FROM dfTable
WHERE (StockCode = 'DOT' AND
       (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1))
```

注意我们并没有将过滤器设置为一条语句，使用一个列名无需其他工作就可以实现。

如果你擅长SQL，就会对上面的语句很熟悉。的确，上面所有这些都可以表示为一个where子句。实际上，将过滤器表示为SQL语句比使用编程式的DataFrame接口更简单，同时Spark SQL实现这点并不会造成性能下降。例如，以下两条语句是等价的：

```
// in Scala
import org.apache.spark.sql.functions.{expr, not, col}
df.withColumn("isExpensive", not(col("UnitPrice").leq(250)))
  .filter("isExpensive")
  .select("Description", "UnitPrice").show(5)
df.withColumn("isExpensive", expr("NOT UnitPrice <= 250"))
  .filter("isExpensive")
  .select("Description", "UnitPrice").show(5)
```

下面是Python中使用SQL where子句的代码：

```
# in Python
from pyspark.sql.functions import expr
df.withColumn("isExpensive", expr("NOT UnitPrice <= 250"))\
  .where("isExpensive")\
  .select("Description", "UnitPrice").show(5)
```

创建布尔表达式时，如果要处理空值数据则会出现问题。如果数据存在空值，则需要以不同的方式处理了。下面这条语句可以保证执行空值安全的等价测试：

```
df.where(col("Description").eqNullSafe("hello")).show()
```

IS [NOT] DISTINCT FROM语句虽然目前在Spark 2.2中尚不可用，但Spark 2.3版本将支持它。

处理数值类型

在处理大数据时，过滤之后要执行的第二个最常见的任务是计数。在大多数情况下，我们只需要简单地表达计算方法，并且确保计算表达式对于数值型数据是正确可行的。

举个例子，假设我们发现错误地记录了零售数据集中的数量，而真实数量其实等于（当前数量*单位价格）2 + 5。这需要我们写一个数值计算函数——pow函数，来对指定列进行幂运算：

```
// in Scala
import org.apache.spark.sql.functions.{expr, pow}
val fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
df.select(expr("CustomerId"), fabricatedQuantity.alias("realQuantity")).show(2)

# in Python
from pyspark.sql.functions import expr, pow
fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
df.select(expr("CustomerId"), fabricatedQuantity.alias("realQuantity")).show(2)
```

```
+----------+------------------+
|CustomerId|      realQuantity|
+----------+------------------+
|   17850.0|239.08999999999997|
|   17850.0|          418.7156|
+----------+------------------+
```

注意，我们可以对两列数据进行乘法操作，这是因为它们都是数值类型，当然，也能进行加法和减法操作。实际上，也可以使用SQL表达式来实现所有这些操作：

```
// in Scala
df.selectExpr(
  "CustomerId",
  "(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity").show(2)

# in Python
df.selectExpr(
  "CustomerId",
  "(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity").show(2)

-- in SQL
SELECT customerId, (POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity
FROM dfTable
```

另一个常见的数值型操作是“四舍五入”操作。如果希望四舍五入为一个整数，通常将数值转换为整型即可。但是，Spark中还有更具体的函数来执行某个级别精度的转换。在下面的例子中，我们将四舍五入至小数点后一位：

```
// in Scala
import org.apache.spark.sql.functions.{round, bround}
df.select(round(col("UnitPrice"), 1).alias("rounded"), col("UnitPrice")).show(5)
```

默认情况下，如果恰好位于两个数字之间，则round函数会向上取整，也可以通过bround函数进行向下取整。

```
// in Scala
import org.apache.spark.sql.functions.lit
df.select(round(lit("2.5")), bround(lit("2.5"))).show(2)

# in Python
from pyspark.sql.functions import lit, round, bround

df.select(round(lit("2.5")), bround(lit("2.5"))).show(2)

-- in SQL
SELECT round(2.5), bround(2.5)

+-------------+--------------+
|round(2.5, 0)|bround(2.5, 0)|
+-------------+--------------+
```

```
|          3.0|           2.0|
|          3.0|           2.0|
+-------------+--------------+
```

另一个数值型操作就是计算两列的相关性。例如，我们可以通过两列的Pearson相关系数来查看是否东西越便宜买的就越多。我们可以通过函数以及DataFrame统计方法实现此操作：

```
// in Scala
import org.apache.spark.sql.functions.{corr}
df.stat.corr("Quantity", "UnitPrice")
df.select(corr("Quantity", "UnitPrice")).show()

# in Python
from pyspark.sql.functions import corr
df.stat.corr("Quantity", "UnitPrice")
df.select(corr("Quantity", "UnitPrice")).show()

-- in SQL
SELECT corr(Quantity, UnitPrice) FROM dfTable

+-------------------------+
|corr(Quantity, UnitPrice)|
+-------------------------+
|     -0.04112314436835551|
+-------------------------+
```

另一个常见操作是计算一列或一组列的汇总统计信息，可以用`describe`方法来实现。它会计算所有数值型列的计数、均值、标准差、最小值和最大值。将来schema可能有所变化，所以主要是在命令行中使用它来查看：

```
// in Scala
df.describe().show()

# in Python
df.describe().show()

+-------+------------------+------------------+------------------+
|summary|          Quantity|         UnitPrice|        CustomerID|
+-------+------------------+------------------+------------------+
|  count|              3108|              3108|              1968|
|   mean| 8.627413127413128| 4.151946589446603|15661.388719512195|
| stddev|26.371821677029203|15.638659854603892|1854.4496996893627|
|    min|               -24|               0.0|           12431.0|
|    max|               600|            607.49|           18229.0|
+-------+------------------+------------------+------------------+
```

如果需要这些精确的数字，也可以通过导入函数并在所需列上应用来实现聚合操作：

```
// in Scala
import org.apache.spark.sql.functions.{count, mean, stddev_pop, min, max}
```

```
# in Python
from pyspark.sql.functions import count, mean, stddev_pop, min, max
```

StatFunctions包中封装了许多可供使用的统计函数（如下面代码所示，可以使用stat来访问）。这些是适用于各种计算的DataFrame方法，比如，可以使用approxQuantile方法来计算数据的精确分位数或近似分位数：

```
// in Scala
val colName = "UnitPrice"
val quantileProbs = Array(0.5)
val relError = 0.05
df.stat.approxQuantile("UnitPrice", quantileProbs, relError) // 2.51

# in Python
colName = "UnitPrice"
quantileProbs = [0.5]
relError = 0.05
df.stat.approxQuantile("UnitPrice", quantileProbs, relError) # 2.51
```

也可以使用它来查看交叉列表或频繁项对（注意：该输出会很大，所以这里省略）：

```
// in Scala
df.stat.crosstab("StockCode", "Quantity").show()

# in Python
df.stat.crosstab("StockCode", "Quantity").show()

// in Scala
df.stat.freqItems(Seq("StockCode", "Quantity")).show()

# in Python
df.stat.freqItems(["StockCode", "Quantity"]).show()
```

最后一点，我们还可以使用monotonically_increasing_id函数为每行添加一个唯一的ID。它会从0开始，为每行生成一个唯一值：

```
// in Scala
import org.apache.spark.sql.functions.monotonically_increasing_i
df.select(monotonically_increasing_id()).show(2)

# in Python
from pyspark.sql.functions import monotonically_increasing_id
df.select(monotonically_increasing_id()).show(2)
```

每个发布版本都会新增一些函数方法，请查阅文档获取更多的方法。例如，有一些随机数生成方法（例如rand()和randn()），可以使用它们生成随机数，但是这些随机数方法并不是完全随机的，具有一定的确定性（你可以在Spark邮件列表中找到关于该问题的讨论）。还有一些更高级的操作，例如在本章开头提到的stat包中的布隆过滤器和数据略图算法。请务必查看API文档以获取更多信息和函数。

处理字符串类型

字符串操作几乎在每个数据流中都有，你可能正在操作执行正则表达式提取或替换的日志文件，或者检查其中是否包含简单的字符串，或者将所有字符串都变成大写或小写。

我们从字符串大小写转换开始，因为它最简单直接的，`initcap`函数会将给定字符串中空格分隔的每个单词首字母大写。

```
// in Scala
import org.apache.spark.sql.functions.{initcap}
df.select(initcap(col("Description"))).show(2, false)

# in Python
from pyspark.sql.functions import initcap
df.select(initcap(col("Description"))).show()

-- in SQL
SELECT initcap(Description) FROM dfTable

+----------------------------------+
|initcap(Description)              |
+----------------------------------+
|White Hanging Heart T-light Holder|
|White Metal Lantern               |
+----------------------------------+
```

正如刚提到的，你还可以将字符串转为大写或小写：

```
// in Scala
import org.apache.spark.sql.functions.{lower, upper}
df.select(col("Description"),
  lower(col("Description")),
  upper(lower(col("Description")))).show(2)

# in Python
from pyspark.sql.functions import lower, upper
df.select(col("Description"),
    lower(col("Description")),
upper(lower(col("Description")))).show(2)

-- in SQL
SELECT Description, lower(Description), Upper(lower(Description)) FROM dfTable

+--------------------+--------------------+-------------------------+
|         Description|  lower(Description)|upper(lower(Description))|
+--------------------+--------------------+-------------------------+
|WHITE HANGING HEA...|white hanging hea...|     WHITE HANGING HEA...|
| WHITE METAL LANTERN| white metal lantern|      WHITE METAL LANTERN|
+--------------------+--------------------+-------------------------+
```

另一个简单的任务是删除字符串周围的空格或者在其周围添加空格，可以使用lpad、ltrim、rpad 及rtrim、trim来实现。

```
// in Scala
import org.apache.spark.sql.functions.{lit, ltrim, rtrim, rpad, lpad, trim}
df.select(
    ltrim(lit("    HELLO    ")).as("ltrim"),
    rtrim(lit("    HELLO    ")).as("rtrim"),
    trim(lit("    HELLO    ")).as("trim"),
    lpad(lit("HELLO"), 3, " ").as("lp"),
rpad(lit("HELLO"), 10, " ").as("rp")).show(2)

# in Python
from pyspark.sql.functions import lit, ltrim, rtrim, rpad, lpad, trim
df.select(
    ltrim(lit("    HELLO    ")).alias("ltrim"),
    rtrim(lit("    HELLO    ")).alias("rtrim"),
    trim(lit("    HELLO    ")).alias("trim"),
    lpad(lit("HELLO"), 3, " ").alias("lp"),
rpad(lit("HELLO"), 10, " ").alias("rp")).show(2)

-- in SQL
SELECT
  ltrim('   HELLLOOOO  '),
  rtrim('    HELLLOOOO  '),
  trim('    HELLLOOOO  '),
  lpad('HELLOOOO  ', 3, ' '),
  rpad('HELLOOOO  ', 10, ' ')
FROM dfTable

+---------+---------+-----+---+----------+
|    ltrim|    rtrim| trim| lp|        rp|
+---------+---------+-----+---+----------+
|HELLO    |    HELLO|HELLO| HE|HELLO     |
|HELLO    |    HELLO|HELLO| HE|HELLO     |
+---------+---------+-----+---+----------+
```

注意，如果lpad或rpad方法输入的数值参数小于字符串长度，它将从字符串的右侧删除字符。

正则表达式

最常见的任务之一是在一个字符串中搜索子串，替换被选中的字符串等。在很多编程语言中，使用正则表达式实现这些功能。正则表达式使得用户可以指定一组规则，从字符串中提取子串或者替换子串。

Spark充分利用了Java正则表达式的强大功能，但Java正则表达式与其他编程语言中的略有差别，因此实际应用之前需要检查。想要执行正则表达式操作，你需要用到

Spark中两个关键函数：regexp_extract和regexp_replace，这两个函数分别用于提取值和替换值。

接下来说明如何使用regexp_replace函数来替换掉Description列中的颜色名：

```
// in Scala
import org.apache.spark.sql.functions.regexp_replace
val simpleColors = Seq("black", "white", "red", "green", "blue")
val regexString = simpleColors.map(_.toUpperCase).mkString("|")
// "|"在正则表达式中是"或"的意思
df.select(
  regexp_replace(col("Description"), regexString, "COLOR").alias("color_clean"),
  col("Description")).show(2)

# in Python
from pyspark.sql.functions import regexp_replace
regex_string = "BLACK|WHITE|RED|GREEN|BLUE"
df.select(
  regexp_replace(col("Description"), regex_string, "COLOR").alias("color_clean"),
  col("Description")).show(2)

-- in SQL
SELECT
  regexp_replace(Description, 'BLACK|WHITE|RED|GREEN|BLUE', 'COLOR') as
  color_clean, Description
FROM dfTable

+--------------------+--------------------+
|         color_clean|         Description|
+--------------------+--------------------+
|COLOR HANGING HEA...|WHITE HANGING HEA...|
| COLOR METAL LANTERN| WHITE METAL LANTERN|
+--------------------+--------------------+
```

另一个任务是用其他字符替换给定的字符。构建正则表达式来实现该操作可能会有些冗长，所以Spark还提供了translate函数来实现该替换操作。这是在字符级上完成的操作，并将用给定字符串中替换掉所有出现的某字符串：

```
// in Scala
import org.apache.spark.sql.functions.translate
df.select(translate(col("Description"), "LEET", "1337"), col("Description"))
  .show(2)

# in Python
from pyspark.sql.functions import translate
df.select(translate(col("Description"), "LEET", "1337"),col("Description"))\
  .show(2)

-- in SQL
SELECT translate(Description, 'LEET', '1337'), Description FROM dfTable

+---------------------------------+--------------------+
|translate(Description, LEET, 1337)|         Description|
```

```
+---------------------------------+-------------------+
|              WHI73 HANGING H3A...|WHITE HANGING HEA...|
|               WHI73 M37A1 1AN73RN| WHITE METAL LANTERN|
+---------------------------------+-------------------+
```

我们也可以执行其他类似的任务，比如取出第一个被提到的颜色：

```
// in Scala
import org.apache.spark.sql.functions.regexp_extract
val regexString = simpleColors.map(_.toUpperCase).mkString("(", "|", ")")
//  "|"是正则表达式中的"或"的意思
df.select(
     regexp_extract(col("Description"), regexString, 1).alias("color_clean"),
     col("Description")).show(2)

# in Python
from pyspark.sql.functions import regexp_extract
extract_str = "(BLACK|WHITE|RED|GREEN|BLUE)"
df.select(
     regexp_extract(col("Description"), extract_str, 1).alias("color_clean"),
     col("Description")).show(2)

-- in SQL
SELECT regexp_extract(Description, '(BLACK|WHITE|RED|GREEN|BLUE)', 1),
  Description
FROM dfTable

+-------------+--------------------+
|  color_clean|         Description|
+-------------+--------------------+
|        WHITE|WHITE HANGING HEA...|
|        WHITE| WHITE METAL LANTERN|
+-------------+--------------------+
```

有时，我们并不是要提取字符串，而是只想检查它们是否存在。此时可以在每列上用contains方法来实现这个操作。该方法将返回一个布尔值，它表示指定的值是否在该列的字符串中：

```
// in Scala
val containsBlack = col("Description").contains("BLACK")
val containsWhite = col("DESCRIPTION").contains("WHITE")
df.withColumn("hasSimpleColor", containsBlack.or(containsWhite))
  .where("hasSimpleColor")
  .select("Description").show(3, false)
```

在Python和SQL中，可以使用instr函数：

```
# in Python
from pyspark.sql.functions import instr
containsBlack = instr(col("Description"), "BLACK") >= 1
containsWhite = instr(col("Description"), "WHITE") >= 1
df.withColumn("hasSimpleColor", containsBlack | containsWhite)\
```

```
  .where("hasSimpleColor")\
  .select("Description").show(3, False)

-- in SQL
SELECT Description FROM dfTable
WHERE instr(Description, 'BLACK') >= 1 OR instr(Description, 'WHITE') >= 1

+---------------------------------+
|Description                      |
+---------------------------------+
|WHITE HANGING HEART T-LIGHT HOLDER|
|WHITE METAL LANTERN              |
|RED WOOLLY HOTTIE WHITE HEART.   |
+---------------------------------+
```

仅仅两个值时看起来很简单，但当有很多值时，它会变得更复杂。

我们利用Spark可以接收不定数量参数的能力以更严谨的方式来解决这个问题。当将一列值转换为一组参数并将它们传递到函数中时， 我们使用了var args，这可以有效地解析任意长度的数组，并将它作为参数传递给函数。结合select方法，就可以动态地创建任意数量的列:

```
// in Scala
val simpleColors = Seq("black", "white", "red", "green", "blue")
val selectedColumns = simpleColors.map(color => {
   col("Description").contains(color.toUpperCase).alias(s"is_$color")
}):+expr("*") // could also append this value
df.select(selectedColumns:_*).where(col("is_white").or(col("is_red")))
  .select("Description").show(3, false)

+---------------------------------+
|Description                      |
+---------------------------------+
|WHITE HANGING HEART T-LIGHT HOLDER|
|WHITE METAL LANTERN              |
|RED WOOLLY HOTTIE WHITE HEART.   |
+---------------------------------+
```

用Python来实现就很容易。在这种情况下，将使用locate函数，它返回整数位置 (从1开始)，然后将返回值转换为布尔值之后再使用它：

```
# in Python
from pyspark.sql.functions import expr, locate
simpleColors = ["black", "white", "red", "green", "blue"]
def color_locator(column, color_string):
  return locate(color_string.upper(), column)\
          .cast("boolean")\
          .alias("is_" + c)
selectedColumns = [color_locator(df.Description, c) for c in simpleColors]
selectedColumns.append(expr("*")) # has to a be Column type
```

```
df.select(*selectedColumns).where(expr("is_white OR is_red"))\
  .select("Description").show(3, False)
```

这个简单的特性通常可以帮助你以一种易于理解和扩展的方式以编程方式生成列或Boolean过滤器。此外，我们还可以把它扩展到计算给定输入值的最小公分母，或者一个数字是否是素数。

处理日期和时间戳类型

在程序语言和数据库领域中，处理日期和时间一直都是难题。必须确保失去和时间格式正确无误。Spark通过显式地关注两种时间相关的信息来极致地简化了这个问题，分别是专门针对日历日期的date，以及包括日期和时间信息的timestamp。正如从当前数据集中所看到的，Spark将尽最大努力正确识别列数据类型，例如，当设置`inferSchema`为true的时候，Spark可以自动推理出日期和时间戳数据类型。可以看出， 这对当前的数据集非常有效，因为不需为其提供具体细节Spark就能识别并读取日期数据格式。

处理日期和时间戳类型与处理字符串类型密切相关， 因为我们经常将时间戳或日期存储为字符串， 并在运行时将它们转换为日期类型。虽然在使用数据库和结构化数据时，可能并不需要字符串类型的转换，但在处理文本和CSV文件时很常见。接下来就会进行相关的实验。

遗憾的是，在处理日期和时间戳时有很多警告，尤其当涉及时区处理时。在版本2.1 及其之前的版本中，如果时区没有被显式指定，Spark则根据计算机的时区来解析。如果有需要，可以通过在SQL配置中设置`spark.conf.sessionLocalTimeZone`来设置会话本地时区，这应该根据Java TimeZone格式进行设置。

```
df.printSchema()

root
|-- InvoiceNo: string (nullable = true)
|-- StockCode: string (nullable = true)
|-- Description: string (nullable = true)
|-- Quantity: integer (nullable = true)
|-- InvoiceDate: timestamp (nullable = true)
|-- UnitPrice: double (nullable = true)
|-- CustomerID: double (nullable = true)
|-- Country: string (nullable = true)
```

尽管Spark会尽最大努力解析日期或时间，然而，有时对于奇怪格式的日期和时间也

束手无策。了解你需要应用的转换的关键，是确保你确切地知道在每个给定步骤中数据具有何种类型和格式。另一个常见的难题是Spark的TimestampType类只支持二级精度，这意味着如果要处理毫秒或微秒，可能需要将数据作为long类型操作才能解决该问题。在强制转换为TimestampType时，任何更高的精度都被删除。

Spark对在某一时刻使用何种日期类型有些特殊，在解析或转换数据的时候要确保格式没有问题。到目前为止，Spark仍在使用Java日期和时间戳，因此要确保符合这些标准。我们从基础开始，获取当前日期和当前时间戳:

```
// in Scala
import org.apache.spark.sql.functions.{current_date, current_timestamp}
val dateDF = spark.range(10)
  .withColumn("today", current_date())
  .withColumn("now", current_timestamp())
dateDF.createOrReplaceTempView("dateTable")

# in Python
from pyspark.sql.functions import current_date, current_timestamp
dateDF = spark.range(10)\
  .withColumn("today", current_date())\
  .withColumn("now", current_timestamp())
dateDF.createOrReplaceTempView("dateTable")
dateDF.printSchema()

root
 |-- id: long (nullable = false)
 |-- today: date (nullable = false)
 |-- now: timestamp (nullable = false)
```

现在有一个简单的DataFrame可以使用，我们从今天起增加和减去5天。这些函数读取一列， 然后将添加或减去的天数作为参数:

```
// in Scala
import org.apache.spark.sql.functions.{date_add, date_sub}
dateDF.select(date_sub(col("today"), 5), date_add(col("today"), 5)).show(1)

# in Python
from pyspark.sql.functions import date_add, date_sub
dateDF.select(date_sub(col("today"), 5), date_add(col("today"), 5)).show(1)

-- in SQL
SELECT date_sub(today, 5), date_add(today, 5) FROM dateTable

+------------------+------------------+
|date_sub(today, 5)|date_add(today, 5)|
+------------------+------------------+
|        2017-06-12|        2017-06-22|
+------------------+------------------+
```

另一项常见任务是查看两个日期之间的间隔时间。可以使用datediff函数来完成，该

函数将返回两个日期之间的天数。大多数情况下，我们只关心天数。由于每个月的天数不同， 还有一个months_between函数，它可以给出两个日期之间相隔的月数:

```
// in Scala
import org.apache.spark.sql.functions.{datediff, months_between, to_date}
dateDF.withColumn("week_ago", date_sub(col("today"), 7))
  .select(datediff(col("week_ago"), col("today"))).show(1)
dateDF.select(
    to_date(lit("2016-01-01")).alias("start"),
    to_date(lit("2017-05-22")).alias("end"))
  .select(months_between(col("start"), col("end"))).show(1)

# in Python
from pyspark.sql.functions import datediff, months_between, to_date
dateDF.withColumn("week_ago", date_sub(col("today"), 7))\
  .select(datediff(col("week_ago"), col("today"))).show(1)

dateDF.select(
    to_date(lit("2016-01-01")).alias("start"),
    to_date(lit("2017-05-22")).alias("end"))\
  .select(months_between(col("start"), col("end"))).show(1)

-- in SQL
SELECT to_date('2016-01-01'), months_between('2016-01-01', '2017-01-01'),
datediff('2016-01-01', '2017-01-01')
FROM dateTable

+-------------------------+
|datediff(week_ago, today)|
+-------------------------+
|                       -7|
+-------------------------+

+--------------------------+
|months_between(start, end)|
+--------------------------+
|              -16.67741935|
+--------------------------+
```

请注意， 我们引入了一个新函数——to_date函数。该函数以指定的格式将字符串转换为日期数据。如果使用这个函数，则要在Java SimpleDateFormat中指定我们想要的格式，这一步非常重要：

```
// in Scala
import org.apache.spark.sql.functions.{to_date, lit}
spark.range(5).withColumn("date", lit("2017-01-01"))
  .select(to_date(col("date"))).show(1)

# in Python
from pyspark.sql.functions import to_date, lit
spark.range(5).withColumn("date", lit("2017-01-01"))\
  .select(to_date(col("date"))).show(1)
```

如果Spark无法解析日期，它不会抛出错误，而只是返回null。在大规模流水线操作时，如果你想获取某种格式的日期数据，再将其转化成另一种格式，这可能就带来麻烦。为了解释这一点，来看看从“年-月-日”切换到到“年-日-月”的日期格式。Spark将无法解析此日期，并默认返回null：

```
dateDF.select(to_date(lit("2016-20-12")), to_date(lit("2017-12-11"))).show(1)

+------------------+------------------+
|to_date(2016-20-12)|to_date(2017-12-11)|
+------------------+------------------+
|              null|        2017-12-11|
+------------------+------------------+
```

我们发现这是一个比较棘手的bug，因为某些日期可能与正确的格式相匹配， 而某些日期则可能不匹配。在上面的示例中，第二个日子显示为12月11日，而不是正确日期11月12日。Spark不会抛出错误，因为它无法知道是日期格式用错了，还是这行数据本身就是不正确的。

接下来我们逐步修复这个流水线操作，并提出一个可靠的方法来完全避免这些问题。第一步，记住我们需要根据Java SimpleDateFormat 标准指定日期格式。

我们将使用两个函数来解决此问题：to_date和to_timestamp。前者可选择一种日期格式，而后者则强制要求使用一种日期格式：

```
// in Scala
import org.apache.spark.sql.functions.to_date
val dateFormat = "yyyy-dd-MM"
val cleanDateDF = spark.range(1).select(
    to_date(lit("2017-12-11"), dateFormat).alias("date"),
    to_date(lit("2017-20-12"), dateFormat).alias("date2"))
cleanDateDF.createOrReplaceTempView("dateTable2")

# in Python
from pyspark.sql.functions import to_date
dateFormat = "yyyy-dd-MM"
cleanDateDF = spark.range(1).select(
    to_date(lit("2017-12-11"), dateFormat).alias("date"),
    to_date(lit("2017-20-12" ), dateFormat).alias("date2" ))
cleanDateDF.createOrReplaceTempView("dateTable2" )

-- in SQL
SELECT to_date(date, 'yyyy-dd-MM'), to_date(date2, 'yyyy-dd-MM'), to_date(date)
FROM dateTable2

+----------+----------+
|      date|     date2|
+----------+----------+
|2017-11-12|2017-12-20|
```

```
+----------+----------+
```

现在举一个to_timestamp的例子，它要求指定一种格式:

```
// in Scala
import org.apache.spark.sql.functions.to_timestamp
cleanDateDF.select(to_timestamp(col("date"), dateFormat)).show()

# in Python
from pyspark.sql.functions import to_timestamp
cleanDateDF.select(to_timestamp(col("date"), dateFormat)).show()

-- in SQL
SELECT to_timestamp(date, 'yyyy-dd-MM'), to_timestamp(date2, 'yyyy-dd-MM')
FROM dateTable2

+----------------------------------+
|to_timestamp(`date`, 'yyyy-dd-MM')|
+----------------------------------+
|               2017-11-12 00:00:00|
+----------------------------------+
```

日期和时间戳之间的转换在所有语言中都很简单，在 SQL 中， 可以按以下方式实现此操作:

```
-- in SQL
SELECT cast(to_date("2017-01-01", "yyyy-dd-MM") as timestamp)
```

在以正确的格式和类型获取了日期或时间戳之后，它们之间的比较实际上很简单，只需要确保使用同一种日期/时间戳类型格式， 或者根据yyyy-MM-dd这种正确格式来指定字符串:

```
cleanDateDF.filter(col("date2") > lit("2017-12-12")).show()
```

另一个要说明的是，也可以将它设置为一个字符串，这样Spark将把它解析为文本:

```
cleanDateDF.filter(col("date2") > "'2017-12-12'").show()
```

使用隐式类型转换可能会在无意中造成“搬起石头砸自己的脚”，尤其是在处理null值或不同时区或格式的日期时。我们建议使用显式的类型转换，不要采取隐式转换的方法。

处理数据中的空值

在实际应用中，建议始终使用null来表示 DataFrame中缺少或空的数据。相较于使用空字符串或其他值来说，使用null值更有利于Spark进行优化。基于DataFrame，处理null值主要的方式是使用.na子包，还有一些用于执行操作并显式指定Spark应如何处理null值的函数。想了解更多信息， 请参阅第5章（在介绍排序的地方）， 还可以参阅本章前面的“处理布尔类型”。

空值处理是所有编程语言中具有挑战性的部分，当然Spark也不例外。在我们看来，显式处理空值比隐式处理好一些。例如，在本书的这一部分中， 我们看到了如何将列定义为允许null类型。然而，这带来一个问题。当我们声明列没有空值时，这并不是实际意义上的强制无空值，当你定义一个数据模式，其中所有的列被声明为不具有null值类型时，Spark不会强制拒绝空值插入。可为空值的设置只是为了帮助Spark SQL处理该列时进行性能优化，如果在不该有空值的列中有空值，则可能会得到不正确的结果或很奇怪的表达式，这给调试带来很大困难。

对于null值，可以执行以下两项操作: 显式删除null值，也可以用某实值来代替空值。我们来实验一下。

合并

Spark通过使用coalesce函数，实现从一组列中选择第一个非空值。在下面例子中，因为没有null值， 所以它只是返回第一列:

```
// in Scala
import org.apache.spark.sql.functions.coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()

# in Python
from pyspark.sql.functions import coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

ifnull、nullIf、nvl 和 nvl2

还有一些其他SQL函数可用于实现类似的操作。ifnull的功能是，如果第一个值为空， 则允许选择第二个值， 并将其默认为第一个。或者可以使用nullif， 如果两个值相等， 则返回null， 否则返回第二个值。nvl的功能是，如果第一个值为null，则返回第二个值，否则返回第一个。最后，nvl2的功能是，如果第一个不为null，返回第二个值；否则，它将返回最后一个指定值（下面示例中的else_value）：

```
-- in SQL
SELECT
```

```
  ifnull(null, 'return_value'),
  nullif('value', 'value'),
  nvl(null, 'return_value'),
  nvl2('not_null', 'return_value', "else_value")
FROM dfTable LIMIT 1

+------------+----+------------+------------+
|           a|   b|           c|           d|
+------------+----+------------+------------+
|return_value|null|return_value|return_value|
+------------+----+------------+------------+
```

当然， 我们也可以在 DataFrame上的select表达式中使用它们。

drop

drop是最简单的函数， 它用于删除包含null的行，默认删除包含null值的行：

```
df.na.drop()
df.na.drop("any")
```

在SQL中，必须逐列进行：

```
-- in SQL
SELECT * FROM dfTable WHERE Description IS NOT NULL
```

若指定“any”作为参数，当存在一个值是null时，就删除改行；若指定“all”为参数，只有当所有的值为null或者NaN时才能删除该行：

```
df.na.drop("all")
```

我们也可以通过指定某几列，来对这些列进行删除空值操作：

```
// in Scala
df.na.drop("all", Seq("StockCode", "InvoiceNo"))

# in Python
df.na.drop("all", subset=["StockCode", "InvoiceNo"])
```

fill

fill函数可以用一组值填充一列或多列，它可以通过指定一个映射（即一个特定值和一组列）来完成此操作。

例如，要替换某字符串类型列中的所有null值为某一字符串，可以指定以下内容：

```
df.na.fill("All Null values become this string")
```

对于Integer类型的列，可以使用df.na.fill(5:Integer)来实现；对于Doubles类型的列，则使用df.na.fill(5:Double)。想要指定多列，需传入一个列名的数组，如同前面的例子中所示:

```
// in Scala
df.na.fill(5, Seq("StockCode", "InvoiceNo"))

# in Python
df.na.fill("all", subset=["StockCode", "InvoiceNo"])
```

我们还可以使用Scala的Map映射来实现，其中主键是列名，而值是我们想用来替换null 值：

```
// in Scala
val fillColValues = Map("StockCode" -> 5, "Description" -> "No Value")
df.na.fill(fillColValues)

# in Python
fill_cols_vals = {"StockCode": 5, "Description" : "No Value"}
df.na.fill(fill_cols_vals)
```

replace

除了像使用drop和fill函数来替换 null 值之外，还有不只针对空值的灵活操作。可能最常见的用例是根据当前值替换掉某列中的所有值，唯一的要求是替换值与原始值的类型相同:

```
// in Scala
df.na.replace("Description", Map("" -> "UNKNOWN"))

# in Python
df.na.replace([""], ["UNKNOWN"], "Description")
```

排序

正如在第5章中讨论的那样，可以使用asc_nulls_first，desc_nulls_first，asc_nulls_last或desc_nulls_last来指定希望 null 值出现在有序 DataFrame中的位置。

处理复杂类型

复杂类型有助于针对想解决的问题以更有意义的方式去组织和构造数据，接下来介绍三种复杂类型: 结构体、数组和map映射。

结构体

可以把结构体视为 DataFrame中的 DataFrame，下面这个例子会帮你更清晰的理解这一点。我们通过在查询中用圆括号括起一组列来创建一个结构体:

```
df.selectExpr("(Description, InvoiceNo) as complex", "*")

df.selectExpr("struct(Description, InvoiceNo) as complex", "*")

// in Scala
import org.apache.spark.sql.functions.struct
val complexDF = df.select(struct("Description", "InvoiceNo").alias("complex"))
complexDF.createOrReplaceTempView("complexDF")

# in Python
from pyspark.sql.functions import struct
complexDF = df.select(struct("Description", "InvoiceNo").alias("complex"))
complexDF.createOrReplaceTempView("complexDF")
```

现在有一个包含complex列的DataFrame，我们可以像查询另一个 DataFrame一样查询它，唯一的区别是， 使用“.”来访问或列方法getField来实现：

```
complexDF.select("complex.Description")
complexDF.select(col("complex").getField("Description"))
```

还可以使用 * 来查询结构体中的所有值，这将调出顶层 DataFrame的所有列：

```
complexDF.select("complex.*")

-- in SQL
SELECT complex.* FROM complexDF
```

数组

定义数组之前，先来看一个用例。使用当前数据， 目标是读取Description列中的每个单词并将其转换成DataFrame中的一行。

第一个操作是将Description列转换为一个复杂类型，即数组。

split

使用split函数并指定分隔符来执行此操作:

```
// in Scala
import org.apache.spark.sql.functions.split
df.select(split(col("Description"), " ")).show(2)

# in Python
```

```
from pyspark.sql.functions import split
df.select(split(col("Description"), " ")).show(2)

-- in SQL
SELECT split(Description, ' ') FROM dfTable

+---------------------+
|split(Description,  )|
+---------------------+
| [WHITE, HANGING, ...|
| [WHITE, METAL, LA...|
+---------------------+
```

这个功能非常有用，因为Spark允许我们将这种复杂类型作为另一列来操作，还可以使用类似 Python 语法来查询数组的值：

```
// in Scala
df.select(split(col("Description"), " ").alias("array_col"))
  .selectExpr("array_col[0]").show(2)

# in Python
df.select(split(col("Description"), " ").alias("array_col"))\
  .selectExpr("array_col[0]").show(2)

-- in SQL
SELECT split(Description, ' ')[0] FROM dfTable
```

结果如下:

```
+------------+
|array_col[0]|
+------------+
|       WHITE|
|       WHITE|
+------------+
```

数组长度

可以通过查询数组的大小来确定数组的长度:

```
// in Scala
import org.apache.spark.sql.functions.size
df.select(size(split(col("Description"),  " "))).show(2) // 打印5和3

# in Python
from pyspark.sql.functions import size
df.select(size(split(col("Description"),  " "))).show(2) # 打印5和3
array_contains
```

还可以查询此数组是否包含某个值：

```
// in Scala
import org.apache.spark.sql.functions.array_contains
df.select(array_contains(split(col("Description"), " "), "WHITE")).show(2)

# in Python
from pyspark.sql.functions import array_contains
df.select(array_contains(split(col("Description"), " "), "WHITE")).show(2)

-- in SQL
SELECT array_contains(split(Description, ' '), 'WHITE') FROM dfTable
```

结果如下：

```
+-----------------------------------------+
|array_contains(split(Description,  ), WHITE)|
+-----------------------------------------+
|                                     true|
|                                     true|
+-----------------------------------------+
```

但以上这些并没有解决我们的问题。若想将复杂类型转换为一系列行（数组中的每个值为一行）， 则需要使用explode函数。

explode

explode函数的输入参数为一个包含数组的列，并为该数组中的每个值创建一行（每行重复该值）。图6-1展示了这个流程。

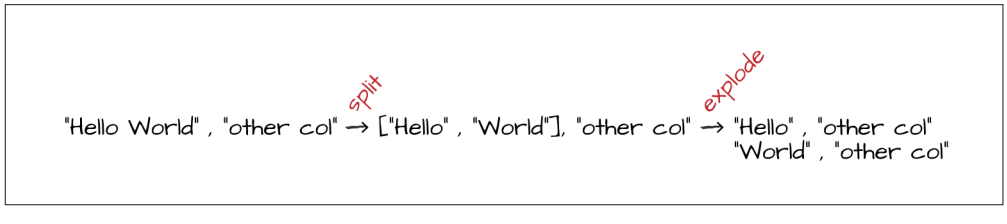


图6-1：展开一列文本

```
// in Scala
import org.apache.spark.sql.functions.{split, explode}

df.withColumn("splitted", split(col("Description"), " "))
  .withColumn("exploded", explode(col("splitted")))
  .select("Description", "InvoiceNo", "exploded").show(2)

# in Python
from pyspark.sql.functions import split, explode

df.withColumn("splitted", split(col("Description"), " "))\
  .withColumn("exploded", explode(col("splitted")))\
```

```
  .select("Description", "InvoiceNo", "exploded").show(2)

-- in SQL
SELECT Description, InvoiceNo, exploded
FROM (SELECT *, split(Description, " ") as splitted FROM dfTable)
LATERAL VIEW explode(splitted) as exploded
```

结果如下：

```
+--------------------+---------+--------+
|         Description|InvoiceNo|exploded|
+--------------------+---------+--------+
|WHITE HANGING HEA...|   536365|   WHITE|
|WHITE HANGING HEA...|   536365| HANGING|
+--------------------+---------+--------+
```

map

map映射是通过map函数构建两列内容的键值对映射形式。然后，便可以像在数组中一样去选择它们:

```
// in Scala
import org.apache.spark.sql.functions.map
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map")).show(2)

# in Python
from pyspark.sql.functions import create_map
df.select(create_map(col("Description"), col("InvoiceNo")).alias("complex_map"))\
  .show(2)

-- in SQL
SELECT map(Description, InvoiceNo) as complex_map FROM dfTable
WHERE Description IS NOT NULL
```

结果如下：

```
+--------------------+
|         complex_map|
+--------------------+
|Map(WHITE HANGING...|
|Map(WHITE METAL L...|
+--------------------+
```

可以使用正确的键值（key）对它们进行查询。若键值（key）不存在则返回 null：

```
// in Scala
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))
  .selectExpr("complex_map['WHITE METAL LANTERN']").show(2)

# in Python
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))\
```

```
  .selectExpr("complex_map['WHITE METAL LANTERN']").show(2)
```

结果如下：

```
+--------------------------------+
|complex_map[WHITE METAL LANTERN]|
+--------------------------------+
|                            null|
|                          536365|
+--------------------------------+
```

还可以展开map类型，将其转换成列：

```
// in Scala
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))
  .selectExpr("explode(complex_map)").show(2)

# in Python
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))\
  .selectExpr("explode(complex_map)").show(2)
```

结果如下：

```
+--------------------+------+
|                 key| value|
+--------------------+------+
|WHITE HANGING HEA...|536365|
| WHITE METAL LANTERN|536365|
+--------------------+------+
```

处理JSON类型

Spark对处理 JSON 数据有一些独特的支持，比如可以在Spark中直接操作JSON字符串，并解析JSON或提取JSON对象。我们首先创建一个JSON类型的列：

```
// in Scala
val jsonDF = spark.range(1).selectExpr("""
  '{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}' as jsonString""")

# in Python
jsonDF = spark.range(1).selectExpr("""
  '{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}' as jsonString""")
```

无论是字典还是数组，均可以使用get_json_object直接查询JSON对象。如果此查询的JSON对象仅有一层嵌套， 则可使用json_tuple：

```
// in Scala
import org.apache.spark.sql.functions.{get_json_object, json_tuple}
jsonDF.select(
```

```
    get_json_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]") as "column",
json_tuple(col("jsonString"), "myJSONKey")).show(2)

# in Python
from pyspark.sql.functions import get_json_object, json_tuple

jsonDF.select(
    get_json_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]") as "column",
    json_tuple(col("jsonString"), "myJSONKey")).show(2)
```

以下是 SQL 中的等价表示:

```
jsonDF.selectExpr(
  "json_tuple(jsonString, '$.myJSONKey.myJSONValue[1]') as column").show(2)
```

结果如下表所示:

```
+------+--------------------+
|column|                  c0|
+------+--------------------+
|     2|{"myJSONValue":[1...|
+------+--------------------+
```

还可以使用to_json函数将 StructType 转换为 JSON 字符串：

```
// in Scala
import org.apache.spark.sql.functions.to_json
df.selectExpr("(InvoiceNo, Description) as myStruct")
  .select(to_json(col("myStruct")))

# in Python
from pyspark.sql.functions import to_json
df.selectExpr("(InvoiceNo, Description) as myStruct")\
  .select(to_json(col("myStruct")))
```

这个函数也可以接受参数的映射作为输入，这些映射与JSON的数据源相同。还可以使用from_json函数将JSON数据解析出来。这需要你指定一个模式，也可以指定其他的映射：

```
// in Scala
import org.apache.spark.sql.functions.from_json
import org.apache.spark.sql.types._
val parseSchema = new StructType(Array(
  new StructField("InvoiceNo",StringType,true),
  new StructField("Description",StringType,true)))
df.selectExpr("(InvoiceNo, Description) as myStruct")
  .select(to_json(col("myStruct")).alias("newJSON"))
  .select(from_json(col("newJSON"), parseSchema), col("newJSON")).show(2)

# in Python
from pyspark.sql.functions import from_json
```

```
from pyspark.sql.types import *
parseSchema = StructType((
  StructField("InvoiceNo",StringType(),True),
  StructField("Description",StringType(),True)))
df.selectExpr("(InvoiceNo, Description) as myStruct")\
  .select(to_json(col("myStruct")).alias("newJSON"))\
  .select(from_json(col("newJSON"), parseSchema), col("newJSON")).show(2)
```

结果如下:

```
+----------------------+--------------------+
|jsontostructs(newJSON)|             newJSON|
+----------------------+--------------------+
|  [536365,WHITE HAN...|{"InvoiceNo":"536...|
|  [536365,WHITE MET...|{"InvoiceNo":"536...|
+----------------------+--------------------+
```

用户自定义函数

Spark最强大的功能之一就是自定义函数，用户自定义函数（UDF）让用户可以使用Python 或 Scala 编写自己的自定义转换操作，甚至可以使用外部库。UDF可以将一个或多个列作为输入，同时也可以返回一个或多个列。Spark UDF非常强大，因为它允许使用多种不同的编程语言编写，而不需要使用一些难懂的格式或限定某些领域特定语言来编写。这些函数只是描述了（一个接一个地）处理数据记录的方法。默认情况下，这些函数被注册为SparkSession或者Context的临时函数。

虽然可以使用Scala、Python 或 Java编写 UDF， 但要注意性能方面的问题。为了说明这一点，我们将详细介绍在创建UDF，在其被发送到Spark，以及在使用该UDF执行代码时，具体都发生了什么。

第一步是设计一个实际的函数，我们用一个简单的例子来说明。我们来编写一个power3函数，该函数接收一个数字并返回它的三次幂：

```
// in Scala
val udfExampleDF = spark.range(5).toDF("num")
def power3(number:Double):Double = number * number * number
power3(2.0)

# in Python
udfExampleDF = spark.range(5).toDF("num")
def power3(double_value):
  return double_value ** 3
power3(2.0)
```

从这个例子中可以看到函数按预期的那样执行，我们提供一个输入，并且得到了预期

的结果。但是目前这个UDF函数的输入要求很高：它必须是特定类型（数值类型），并且不能为null（参见本章前面的“在数据中使用null”）。

现在我们已经创建了这些函数并对它们进行了测试，接下来需要在Spark上注册UDF以便在所有的工作机器上使用它们。Spark将在驱动进程上序列化该函数，并将它通过网络传递到所有执行进程。无论你使用何种语言都是这个过程。

当你使用该函数时，基本上有两种不同的情况发生。如果该函数是用Scala或Java编写的， 则可以在Java虚拟机(JVM)中使用它。这意味着不能使用spark为内置函数提供的代码生成功能，或导致性能的一些下降。在第19章的性能优化部分中介绍了这一点。

如果函数是用 Python 编写的， 则会出现一些截然不同的情况。Spark在worker上启动一个Python 进程， 将所有数据序列化为 Python 可解释的格式（请记住， 数据之前在JVM 中），在 Python 进程中对该数据逐行执行函数，最终将对每行的操作结果返回给JVM 和Spark。图 6-2 提供了该过程的概述。

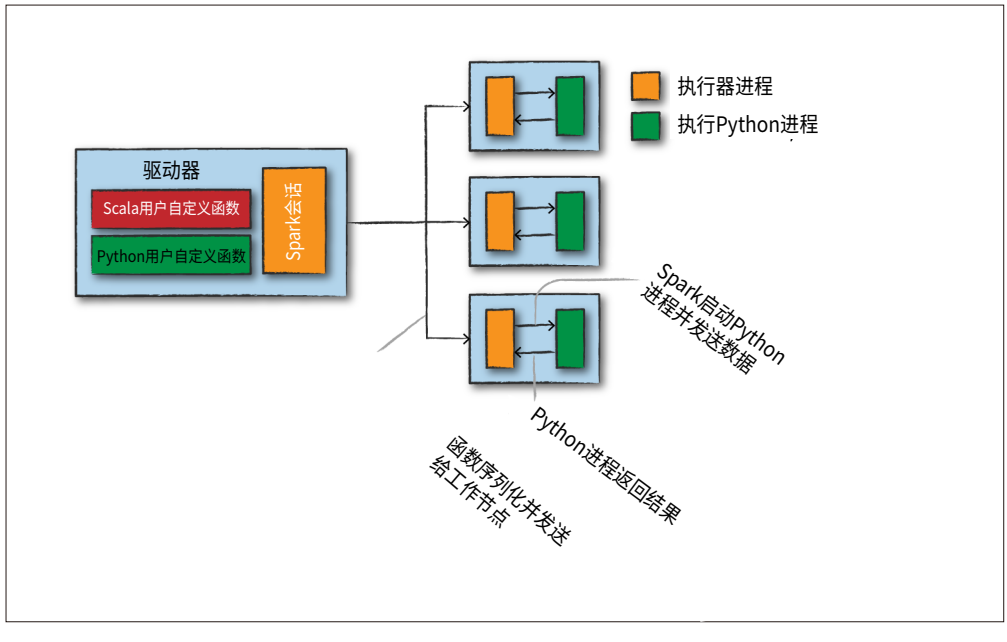


图6-2：基于Python的UDF在Spark中的执行过程

启动此 Python 进程代价很高， 但主要代价是将数据序列化为Python可理解的格式的这个过程。造成代价高的原因有两个: 一个是计算昂贵， 另一个是数据进入 Python 后Spark无法管理worker的内存。这意味着， 如果某个worker因资

> 源受限而失败 (因为 JVM 和 Python 都在同一台计算机上争夺内存)， 则可能会导致该worker出现故障。所以建议使用 Scala 或 Java编写UDF，不仅编写程序的时间少，还能提高性能。当然仍然可以使用Python编写函数。

你对这个过程有所了解了， 现在来实现一个例子吧。首先，需要注册该函数， 以便其可用作 DataFrame 函数:

```
// in Scala
import org.apache.spark.sql.functions.udf
val power3udf = udf(power3(_:Double):Double)
```

可以像使用其他 DataFrame 函数一样使用UDF：

```
// in Scala
udfExampleDF.select(power3udf(col("num"))).show()
```

这同样适用于 Python。首先，注册UDF：

```
# in Python
from pyspark.sql.functions import udf
power3udf = udf(power3)
```

然后，可以在我们的 DataFrame 代码中使用UDF：

```
# in Python
from pyspark.sql.functions import col
udfExampleDF.select(power3udf(col("num"))).show(2)
```

```
+-----------+
|power3(num)|
+-----------+
|          0|
|          1|
+-----------+
```

此时， 我们只能将它用作 DataFrame 函数。也就是说， 我们不能在字符串表达式中使用它。但是，也可以将此UDF 注册为Spark SQL 函数。这种做法很有用，因为它使得能在SQL语言中以及跨语言环境下使用此函数。

接下来在 Scala 中注册该函数：

```
// in Scala
spark.udf.register("power3",  power3(_:Double):Double)
udfExampleDF.selectExpr("power3(num)").show(2)
```

因为这个函数是在Spark SQL注册的，并且任何Spark SQL 函数或表达式都可以在处理DataFrame时使用，所以我们也可以转而使用Python来调用我们之前用Scala编写的

UDF，但我们并不是将它作为DataFrame 函数使用，而是将其作为 SQL 表达式使用：

```
# in Python
udfExampleDF.selectExpr("power3(num)").show(2)
# 在Scala代码中，它已经注册过了
```

我们也可以将Python 函数注册为SQL 函数， 这样就也可以在任何语言中使用它。

为了确保我们的函数正常工作，还要做的一件事是指定返回类型。正如在本节的开头看到的， Spark管理它自己的类型信息， 它不完全与 Python 的类型相一致。因此，最好的做法是在定义函数时定义该函数的返回值类型。请注意， 指定返回值类型不是必需的，但建议指定返回值类型。

如果指定的返回值类型与函数返回的实际类型不匹配， 则Spark不会抛出错误， 但会返回null以表明失败。如果将以下函数中的返回值类型改为Double型， 则可以看到返回null的情况：

```
# in Python
from pyspark.sql.types import IntegerType, DoubleType
spark.udf.register("power3py", power3, DoubleType())

# in Python
udfExampleDF.selectExpr("power3py(num)").show(2)
# registered via Python
```

这是由于整型的表示范围造成的，当在 Python 中操作整数时， Python 不会将它们转换为浮点型（对应于Spark的double类型），因此我们会看到返回null。可以确保Python 函数返回浮点型而非整型来解决这个问题。

在注册后，我们可以在SQL中使用任一个UDF函数：

```
-- in SQL
SELECT power3(12), power3py(12) -- doesn't work because of return type
```

当你想从UDF 中选择返回一个值时， 应该在Python中返回None，并在Scala中返回一个Option类型：

```
## Hive 用户自定义函数
```

最后，还可以使用Hive语法来创建UDF/UDAF。为了实现这一点， 首先必须在创建SparkSession 时启用Hive支持（通过SparkSession.builder().enableHiveSupport()来启用）。然后， 你可以在SQL中注册UDF。这仅支持预编译的Scala和Java包， 因此你需要将它们指定为依赖项:

```
-- in SQL
CREATE TEMPORARY FUNCTION myFunc AS 'com.organization.hive.udf.FunctionName'
```

此外，还能通过删除TEMPORARY将其注册为Hive Metastore中的永久函数。

小结

本章展示了将Spark SQL扩展到你自己应用的简单方法，这并不需要你使用一些深奥的、特定领域的语言，只要用简单的UDF函数就可以，同时这些函数即使不用Spark也能轻松地进行测试和维护！这是一个非常强大的工具，可以使用它来指定复杂的业务逻辑，这些逻辑既可以在本地计算机上运行几行程序，也可以在100个节点集群上处理TB级的数据！

第7章

聚合操作

聚合操作将数据整合到一起，是大数据分析中很常见的基本操作。在聚合操作中，需要指定键或分组方式，以及指定如何转换一列或多列数据的聚合函数。当给定多个输入值时，聚合函数给每个分组计算出一个结果。Spark的聚合功能很完善，支持多种不同的用例。一般情况下，用户使用聚合操作对数据分组后的各组内的数值型数据进行汇总，这个汇总运算可能是求和、累乘、或者是简单的计数。另外，Spark可以将任何类型的值聚合成为array数组、list列表、或map映射，如本章后面的“聚合到复杂类型”所示。

除了处理任意类型的值之外，Spark还允许创建以下分组类型：

- 最简单的分组是通过在select语句中执行聚合来汇总整个DataFrame。
- “group by”指定一个或多个key也可以指定一个或多个聚合函数，来对包含value的列执行转换操作。
- “window”指定一个或多个key也可以指定一个或多个聚合函数，来对包含value的列执行转换操作。但是，输入到该函数的行与当前行有某种联系。
- “grouping set”可用于在多个不同级别进行聚合。grouping set是SQL中的一个保留字，而在DataFrame中需要使用rollup和cube。
- “rollup”指定一个或多个key，也可以指定一个或多个聚合函数，来对包含value的列执行转换操作，并会针对指定的多个key进行分级分组汇总。
- “cube”指定一个或多个key，也可以指定一个或多个聚合函数，来对包含value的列执行转换操作，并会针对指定的多个key进行全组合分组汇总。

每个分组操作都会返回RelationalGroupedDataset，基于它来进行聚合操作。

要考虑的一件重要的事是返回结果的精确度。在进行大数据计算的时候，获得一个精确的结果开销会很大，但是计算出一个近似结果相对要容易得多。本书提到的一些近似函数，通常都会提高Spark作业执行速度和效率，特别是对交互式和ad hoc进行分析。

下面的代码首先读取零售业的采购数据，然后对数据进行重划分以减少分区数量（因为我们事先知道仅有少量数据存储在大量小文件里），最后将这些数据缓存起来以便后续的快速访问：

```
// in Scala
val df = spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/data/retail-data/all/*.csv")
  .coalesce(5)
df.cache()
df.createOrReplaceTempView("dfTable")

# in Python
df = spark.read.format("csv")\
  .option("header", "true")\
  .option("inferSchema", "true")\
  .load("/data/retail-data/all/*.csv")\
  .coalesce(5)
df.cache()
df.createOrReplaceTempView("dfTable")
```

下面是数据样本，以便参考：

```
+---------+---------+--------------------+--------+--------------+---------+-----
|InvoiceNo|StockCode|         Description|Quantity|   InvoiceDate|UnitPrice|Cu...
+---------+---------+--------------------+--------+--------------+---------+-----
|   536365|   85123A|WHITE HANGING...    |       6|12/1/2010 8:26|     2.55|  ...
|   536365|    71053|WHITE METAL...      |       6|12/1/2010 8:26|     3.39|  ...
...
|   536367|    21755|LOVE BUILDING BLO...|       3|12/1/2010 8:34|     5.95|  ...
|   536367|    21777|RECIPE BOX WITH M...|       4|12/1/2010 8:34|     7.95|  ...
+---------+---------+--------------------+--------+--------------+---------+-----
```

如上所述，基本的聚合操作将作用于整个DataFrame。最简单的例子是count方法：

```
df.count () == 541909
```

如果你看过前面的内容，你会知道count操作与转换操作不一样，它是动作操作，会立即返回计算结果。可以使用count来获得数据集的总体大小，但它还有一个作用是

可以缓存整个DataFrame到内存里，就像在本例中所做的那样。

用count这种方法实现缓存数据有点奇怪，主要是因为在本例中count立即执行，而不像转换操作那样惰性执行。在下一节中，我们还会看见如何把count作为惰性函数来使用。

聚合函数

除了DataFrame的某些操作或通过.stat访问的方法（如第6章介绍的），所有聚合操作都是以函数形式出现。大多数聚合函数可以在org.apache.spark.sql.functions（*http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$*）包中找到。

从Scala和Python中导入的函数与SQL中的函数有点不一样。每个版本中都有所改变，所以不可能提供一个全面明确的聚合函数列表。本节中将介绍最常见的一些函数。

count

第一个聚合函数是count，但在此示例中，count聚合操作是一个transformation转换操作而不是一个动作操作。在这种情况下，我们可以执行以下两项操作之一：第一个是对指定的列进行计数，第二个是使用count(*)或count(1)对所有列进行计数，如下面例子所示：

```
// in Scala
import org.apache.spark.sql.functions.count
df.select(count("StockCode")).show() // 541909

# in Python
from pyspark.sql.functions import count
df.select(count("StockCode")).show() # 541909

-- in SQL
SELECT COUNT(*) FROM dfTable
```

关于对null值进行计数有一些注意的地方。例如，当执行count(*)时，Spark会对null值进行计数，而当对某指定列计数时，则不会对null值进行计数。

countDistinct

有时，数据的总量不重要，而获得唯一（unique）组的数量才是我们需要的。要获得

唯一组数量，可以使用countDistinct函数，而这个函数仅在统计针对某列的计数时才有意义：

```
// in Scala
import org.apache.spark.sql.functions.countDistinct
df.select(countDistinct("StockCode")).show() // 4070

# in Python
from pyspark.sql.functions import countDistinct
df.select(countDistinct("StockCode")).show() # 4070

-- in SQL
SELECT COUNT(DISTINCT *) FROM DFTABLE
```

approx_count_distinct

通常，在处理大数据集的时候，精确的统计计数并不那么重要，某种精度的近似值也是可以接受的，此时可以使用approx_count_distinct函数：

```
// in Scala
import org.apache.spark.sql.functions.approx_count_distinct
df.select(approx_count_distinct("StockCode", 0.1)).show() // 3364

# in Python
from pyspark.sql.functions import approx_count_distinct
df.select(approx_count_distinct("StockCode", 0.1)).show() # 3364

-- in SQL
SELECT approx_count_distinct(StockCode, 0.1) FROM DFTABLE
```

注意，approx_count_distinct带了另一个参数，该参数指定可容忍的最大误差。本例中我们指定了一个相当大的误差率，因此得到的答案与正确值差距很大，但执行速度更快，比countDistinct函数执行耗时更少。档处理更大的数据集的时候，这种提升会更加明显。

first和last

很明显，这两个函数可以得到DataFrame的第一个值和最后一个值，它是基于DataFrame中行的顺序而不是DataFrame中值的顺序：

```
// in Scala
import org.apache.spark.sql.functions.{first, last}
df.select(first("StockCode"), last("StockCode")).show()

# in Python
from pyspark.sql.functions import first, last
df.select(first("StockCode"), last("StockCode")).show()
```

```
-- in SQL
SELECT first(StockCode), last(StockCode) FROM dfTable

+-----------------------+----------------------+
|first(StockCode, false)|last(StockCode, false)|
+-----------------------+----------------------+
|                 85123A|                 22138|
+-----------------------+----------------------+
```

min和max

若要从 DataFrame 中提取最小值和最大数值，请使用min和max函数：

```
// in Scala
import org.apache.spark.sql.functions.{min, max}
df.select(min("Quantity"), max("Quantity")).show()

# in Python
from pyspark.sql.functions import min, max
df.select(min("Quantity"), max("Quantity")).show()

-- in SQL
SELECT min(Quantity), max(Quantity) FROM dfTable

+-------------+-------------+
|min(Quantity)|max(Quantity)|
+-------------+-------------+
|       -80995|        80995|
+-------------+-------------+
```

sum

另一个简单任务是使用sum函数累加一行中的所有值：

```
// in Scala
import org.apache.spark.sql.functions.sum
df.select(sum("Quantity")).show() // 5176450

# in Python
from pyspark.sql.functions import sum
df.select(sum("Quantity")).show() # 5176450

-- in SQL
SELECT sum(Quantity) FROM dfTable
sumDistinct
```

除了计算总和外，还可以使用sumDistinct函数来对一组去重（distinct）值进行求和：

```
// in Scala
import org.apache.spark.sql.functions.sumDistinct
```

```
df.select(sumDistinct("Quantity")).show() // 29310

# in Python
from pyspark.sql.functions import sumDistinct
df.select(sumDistinct("Quantity")).show() # 29310

-- in SQL
SELECT SUM(Quantity) FROM dfTable -- 29310
avg
```

尽管可以通过总和除以总数计算出平均值，Spark还提供了一种通过avg或mean函数获取平均值的方法。在这个例子中，我们使用alias（别名），以便以后更方便地使用这些值：

```
// in Scala
import org.apache.spark.sql.functions.{sum, count, avg, expr}

df.select(
    count("Quantity").alias("total_transactions"),
    sum("Quantity").alias("total_purchases"),
    avg("Quantity").alias("avg_purchases"),
    expr("mean(Quantity)").alias("mean_purchases"))
  .selectExpr(
    "total_purchases/total_transactions",
    "avg_purchases",
    "mean_purchases").show()

# in Python
from pyspark.sql.functions import sum, count, avg, expr

df.select(
    count("Quantity").alias("total_transactions"),
    sum("Quantity").alias("total_purchases"),
    avg("Quantity").alias("avg_purchases"),
    expr("mean(Quantity)").alias("mean_purchases"))\
  .selectExpr(
    "total_purchases/total_transactions",
    "avg_purchases",
    "mean_purchases").show()
```

```
+--------------------------------------+----------------+----------------+
|(total_purchases / total_transactions)|   avg_purchases|  mean_purchases|
+--------------------------------------+----------------+----------------+
|                      9.55224954743324|9.55224954743324|9.55224954743324|
+--------------------------------------+----------------+----------------+
```

你还可以计算所有去重（distinct）值的平均值。实际上，大多数聚合函数都支持对去重值进行聚合计算。

方差和标准差

当计算均值的时候，自然会想到计算方差和标准差，这些都是与均值一样描述数据分布性质的主要指标。方差是各数据样本与均值之间差的平方的平均值，标准差是方差的平方根。可以在Spark中使用相应的函数来计算这些值。然而值得注意的是，Spark既支持统计样本标准差，也支持统计总体标准差，它们两个在统计学上是完全不同的概念，一定要区分它们。如果使用variance函数和stddev函数，默认是计算样本标准差或样本方差的。

你还可以显式指定这些值或引用总体标准偏差或方差：

```
// in Scala
import org.apache.spark.sql.functions.{var_pop, stddev_pop}
import org.apache.spark.sql.functions.{var_samp, stddev_samp}
df.select(var_pop("Quantity"), var_samp("Quantity"),
  stddev_pop("Quantity"), stddev_samp("Quantity")).show()

# in Python
from pyspark.sql.functions import var_pop, stddev_pop
from pyspark.sql.functions import var_samp, stddev_samp
df.select(var_pop("Quantity"), var_samp("Quantity"),
  stddev_pop("Quantity"), stddev_samp("Quantity")).show()

-- in SQL
SELECT var_pop(Quantity), var_samp(Quantity),
  stddev_pop(Quantity), stddev_samp(Quantity)
FROM dfTable
```

```
+------------------+------------------+--------------------+-------------------+
| var_pop(Quantity)|var_samp(Quantity)|stddev_pop(Quantity)|stddev_samp(Quan...|
+------------------+------------------+--------------------+-------------------+
|47559.303646609056|47559.391409298754|  218.08095663447796|   218.081157850...|
+------------------+------------------+--------------------+-------------------+
```

skewness和kurtosis

偏度系数（skewness）和峰度系数（kurtosis）都是对数据集中的极端数据点的衡量指标。偏度系数衡量数据相对于平均值的不对称程度，而峰度系数衡量数据分布形态陡缓程度。在将数据建模为随机变量的概率分布时，它们都很重要。虽然在这里我们不会深入讨论它们背后的数学原理，但你可以很容易地在网上搜索相关定义。使用以下函数计算偏度和峰度：

```
import org.apache.spark.sql.functions.{skewness, kurtosis}
df.select(skewness("Quantity"), kurtosis("Quantity")).show()

# in Python
from pyspark.sql.functions import skewness, kurtosis
```

```
df.select(skewness("Quantity"), kurtosis("Quantity")).show()

-- in SQL
SELECT skewness(Quantity), kurtosis(Quantity) FROM dfTable

+------------------+------------------+
| skewness(Quantity)|kurtosis(Quantity)|
+------------------+------------------+
|-0.2640755761052562|119768.05495536952|
+------------------+------------------+
```

协方差和相关性

我们讨论了单列聚合，不过有的函数是去比较两个不同列的值之间的相互关系。其中两个函数就是cov和corr，它们分别用于计算协方差和相关性。相关性采用Pearson相关系数来衡量，范围是-1~+1。协方差的范围由数据中的输入决定。

跟var函数一样，协方差又分为样本协方差和总体协方差，因此在使用的时候需要指定，这一点很重要。相关性没有这个概念，因此没有总体或样本的相关性之分。以下是它们的使用方式：

```
// in Scala
import org.apache.spark.sql.functions.{corr, covar_pop, covar_samp}
df.select(corr("InvoiceNo", "Quantity"), covar_samp("InvoiceNo", "Quantity"),
    covar_pop("InvoiceNo", "Quantity")).show()

# in Python
from pyspark.sql.functions import corr, covar_pop, covar_samp
df.select(corr("InvoiceNo", "Quantity"), covar_samp("InvoiceNo", "Quantity"),
    covar_pop("InvoiceNo", "Quantity")).show()

-- in SQL
SELECT corr(InvoiceNo, Quantity), covar_samp(InvoiceNo, Quantity),
  covar_pop(InvoiceNo, Quantity)
FROM dfTable

+-------------------------+-----------------------------+---------------------+
|corr(InvoiceNo, Quantity)|covar_samp(InvoiceNo, Quantity)|covar_pop(InvoiceN...|
+-------------------------+-----------------------------+---------------------+
|     4.912186085635685E-4|             1052.7280543902734|            1052.7...|
+-------------------------+-----------------------------+---------------------+
```

聚合输出复杂类型

在Spark中，不仅可以在数值型上执行聚合操作，还能在复杂类型上执行聚合操作。例如，可以收集某列上的值到一个list列表里，或者将unique唯一值收集到一个set集合里。

用户可以在流水线处理的后续操作中再访问该集合，或者将整个集合传递给用户自定义函数（UDF）：

```
// in Scala
import org.apache.spark.sql.functions.{collect_set, collect_list}
df.agg(collect_set("Country"), collect_list("Country")).show()

# in Python
from pyspark.sql.functions import collect_set, collect_list
df.agg(collect_set("Country"), collect_list("Country")).show()

-- in SQL
SELECT collect_set(Country), collect_set(Country) FROM dfTable

+--------------------+---------------------+
|collect_set(Country)|collect_list(Country)|
+--------------------+---------------------+
|[Portugal, Italy,...| [United Kingdom, ...|
+--------------------+---------------------+
```

分组

到目前为止，我们只在DataFrame级别上进行聚合操作。更常见的任务是根据分组数据进行计算，典型应用是处理类别数据，根据某一列中的数据进行分组，然后基于分组情况来对其他列的数据进行计算。

实践是最好的解释方法，直接来执行一些分组操作。首先我们执行计数操作。我们将按每个唯一的invoice编号进行分组并获取该invoice上的项目数。请注意，这将返回另一个DataFrame并会延迟执行。

分两个阶段进行分组：首先指定要对其进行分组的一列或多列，然后指定一个或多个聚合操作。第一步返回一个RelationalGroupedDataset，第二步返回DataFrame。

如上所述，可以指定任意数量的列进行分组：

```
df.groupBy("InvoiceNo", "CustomerId").count().show()

-- in SQL
SELECT count(*) FROM dfTable GROUP BY InvoiceNo, CustomerId

+---------+----------+-----+
|InvoiceNo|CustomerId|count|
+---------+----------+-----+
|   536846|     14573|   76|
...
|  C544318|     12989|    1|
+---------+----------+-----+
```

使用表达式分组

如前所述，计数有点特殊，因为它是作为一种方法存在的。为此，通常我们更喜欢使用count函数。我们不是将该函数作为表达式传递到select语句中，而是在agg中指定它。这使得仅需指定一些聚合操作，即可传入任意表达式。甚至可以在转换某列之后给它取别名，以便在之后的数据流处理中使用：

```
// in Scala
import org.apache.spark.sql.functions.count

df.groupBy("InvoiceNo").agg(
  count("Quantity").alias("quan"),
  expr("count(Quantity)")).show()

# in Python
from pyspark.sql.functions import count

df.groupBy("InvoiceNo").agg(
    count("Quantity").alias("quan"),
    expr("count(Quantity)")).show()

+---------+----+---------------+
|InvoiceNo|quan|count(Quantity)|
+---------+----+---------------+
|   536596|   6|              6|
...
|  C542604|   8|              8|
+---------+----+---------------+
```

使用Map进行分组

有时将转换操作指定为一系列Map会更方便，其中键为列，值为要执行的字符串形式的聚合函数。如果以inline方式指定也可以重用多个列名：

```
// in Scala
df.groupBy("InvoiceNo").agg("Quantity"->"avg", "Quantity"->"stddev_pop").show()

# in Python
df.groupBy("InvoiceNo").agg(expr("avg(Quantity)"),expr("stddev_pop(Quantity)"))\
  .show()

-- in SQL
SELECT avg(Quantity), stddev_pop(Quantity), InvoiceNo FROM dfTable
GROUP BY InvoiceNo

+---------+------------------+--------------------+
|InvoiceNo|     avg(Quantity)|stddev_pop(Quantity)|
+---------+------------------+--------------------+
|   536596|               1.5|  1.1180339887498947|
...
```

```
|  C542604|              -8.0|  15.173990905493518|
+---------+-------------------+--------------------+
```

window函数

还可以使用window函数来执行某些特殊的聚合操作，具体就是在指定数据“窗口”上执行聚合操作，并使用对当前数据的引用来定义它，此窗口指定将哪些行传递给此函数。这么说有些抽象，它有点类似一个标准的group-by，所以我们来稍微对它们进行区分。

在用group-by处理数据分组时，每一行只能进入一个分组。窗口函数基于称为框（frame）的一组行，计算表的每一输入行的返回值，每一行可以属于一个或多个框。常见用例就是查看某些值的滚动平均值，其中每一行代表一天，那么每行属于7个不同的框。我们稍后会定义框，Spark支持三种窗口函数：排名函数、解析函数和聚合函数。

图7-1说明某一行如何分配到多个框中。

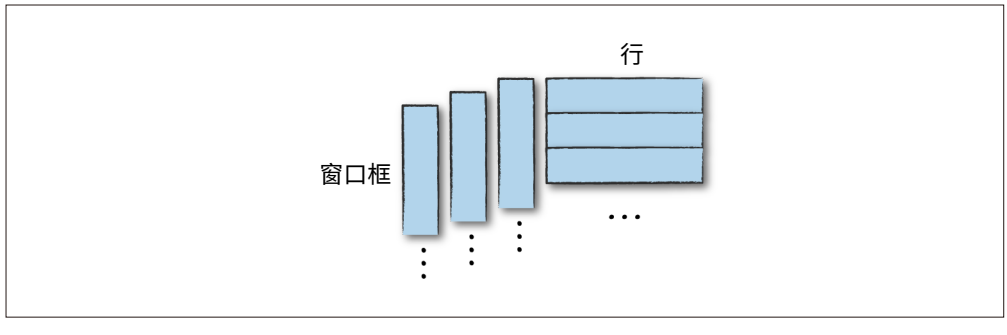


图7-1：窗口函数的可视化示例

为了更好演示，我们将添加一个date列，该列将发票日期转换为仅包含日期信息（不包括时间信息）的列：

```
// in Scala
import org.apache.spark.sql.functions.{col, to_date}
val dfWithDate = df.withColumn("date", to_date(col("InvoiceDate"),
  "MM/d/yyyy H:mm"))
dfWithDate.createOrReplaceTempView("dfWithDate")

# in Python
from pyspark.sql.functions import col, to_date
dfWithDate = df.withColumn("date", to_date(col("InvoiceDate"), "MM/d/yyyy H:mm"))
dfWithDate.createOrReplaceTempView("dfWithDate")
```

配置窗口函数的第一步是创建一个窗口规范。请注意，partition by与我们目前为止所接触的分组概念无关，它只是描述如何进行分组的一个类似概念。排序语句指定了在一个分区内如何对数据排序，最后的rowsBetween语句指定了frame配置。在本例中，设置了当前输入行之前的所有行都包含在这个frame里：

```
// in Scala
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions.col
val windowSpec = Window
  .partitionBy("CustomerId", "date")
  .orderBy(col("Quantity").desc)
  .rowsBetween(Window.unboundedPreceding, Window.currentRow)

# in Python
from pyspark.sql.window import Window
from pyspark.sql.functions import desc
windowSpec = Window\
  .partitionBy("CustomerId", "date")\
  .orderBy(desc("Quantity"))\
  .rowsBetween(Window.unboundedPreceding, Window.currentRow)
```

现在，我们使用聚合函数来了解有关每个特定客户的更多信息。一个例子是计算一个客户有史以来的最大购买数量，为了获得该结果，我们可以使用之前介绍的聚合函数，并将某一列名或表达式作为输入参数。此外，我们还指明了使用某个具体的窗口规范，它定义了此函数将应用于哪些frame：

```
import org.apache.spark.sql.functions.max
val maxPurchaseQuantity = max(col("Quantity")).over(windowSpec)

# in Python
from pyspark.sql.functions import max
maxPurchaseQuantity = max(col("Quantity")).over(windowSpec)
```

这将返回一列（或表达式）。现在，我们可以在 DataFrame的select语句中使用它。不过，在这样做之前，我们先创建购买数量排名。我们使用dense_rank函数来确定每个用户在哪天购买数量最多。使用dense_rank而不是rank，是为了避免在有等值（在我们例子中是重复行）的情况下避免排序结果不连续：

```
// in Scala
import org.apache.spark.sql.functions.{dense_rank, rank}
val purchaseDenseRank = dense_rank().over(windowSpec)
val purchaseRank = rank().over(windowSpec)

# in Python
from pyspark.sql.functions import dense_rank, rank
purchaseDenseRank = dense_rank().over(windowSpec)
purchaseRank = rank().over(windowSpec)
```

这还会返回可在select语句中使用的列。我们可以执行select来查看计算出的窗口值：

```
// in Scala
import org.apache.spark.sql.functions.col

dfWithDate.where("CustomerId IS NOT NULL").orderBy("CustomerId")
  .select(
    col("CustomerId"),
    col("date"),
    col("Quantity"),
    purchaseRank.alias("quantityRank"),
    purchaseDenseRank.alias("quantityDenseRank"),
    maxPurchaseQuantity.alias("maxPurchaseQuantity")).show()

# in Python
from pyspark.sql.functions import col

dfWithDate.where("CustomerId IS NOT NULL").orderBy("CustomerId")\
  .select(
    col("CustomerId"),
    col("date"),
    col("Quantity"),
    purchaseRank.alias("quantityRank"),
    purchaseDenseRank.alias("quantityDenseRank"),
    maxPurchaseQuantity.alias("maxPurchaseQuantity")).show()

-- in SQL
SELECT CustomerId, date, Quantity,
  rank(Quantity) OVER (PARTITION BY CustomerId, date
                       ORDER BY Quantity DESC NULLS LAST
                       ROWS BETWEEN
                         UNBOUNDED PRECEDING AND
                         CURRENT ROW) as rank,

  dense_rank(Quantity) OVER (PARTITION BY CustomerId, date
                             ORDER BY Quantity DESC NULLS LAST
                             ROWS BETWEEN
                               UNBOUNDED PRECEDING AND
                               CURRENT ROW) as dRank,

  max(Quantity) OVER (PARTITION BY CustomerId, date
                      ORDER BY Quantity DESC NULLS LAST
                      ROWS BETWEEN
                        UNBOUNDED PRECEDING AND
                        CURRENT ROW) as maxPurchase
FROM dfWithDate WHERE CustomerId IS NOT NULL ORDER BY CustomerId

+----------+----------+--------+------------+----------------+---------------+
|CustomerId|      date|Quantity|quantityRank|quantityDenseRank|maxP...Quantity|
+----------+----------+--------+------------+----------------+---------------+
|     12346|2011-01-18|   74215|           1|               1|          74215|
|     12346|2011-01-18|  -74215|           2|               2|          74215|
|     12347|2010-12-07|      36|           1|               1|             36|
```

```
|     12347|2010-12-07|      30|           2|                2|             36|
...
|     12347|2010-12-07|      12|           4|                4|             36|
|     12347|2010-12-07|       6|          17|                5|             36|
|     12347|2010-12-07|       6|          17|                5|             36|
+----------+----------+--------+------------+-----------------+---------------+
```

分组集

本章介绍到这里，我们已经介绍了简单的group-by表达式，配合group-by表达式可以对一组列上的值进行聚合操作。但是，在某些情况下我们需要更完善的功能，比如跨多个组的聚合操作。这能通过分组集（grouping set）来实现。分组集是用于将多组聚合操作组合在一起的底层工具，使得能够在group-by语句中创建任意的聚合操作。

让我们通过一个例子来更好的理解它。我们希望获得所有用户的各种股票的数量。为此，我们使用以下 SQL 表达式：

```
// in Scala
val dfNoNull = dfWithDate.drop()
dfNoNull.createOrReplaceTempView("dfNoNull")

# in Python
dfNoNull = dfWithDate.drop()
dfNoNull.createOrReplaceTempView("dfNoNull")

-- in SQL
SELECT CustomerId, stockCode, sum(Quantity) FROM dfNoNull
GROUP BY customerId, stockCode
ORDER BY CustomerId DESC, stockCode DESC

+----------+---------+-------------+
|CustomerId|stockCode|sum(Quantity)|
+----------+---------+-------------+
|     18287|    85173|           48|
|     18287|   85040A|           48|
|     18287|   85039B|          120|
...
|     18287|    23269|           36|
+----------+---------+-------------+
```

可以使用grouping set实现完全相同的操作：

```
-- in SQL
SELECT CustomerId, stockCode, sum(Quantity) FROM dfNoNull
GROUP BY customerId, stockCode GROUPING SETS((customerId, stockCode))
ORDER BY CustomerId DESC, stockCode DESC

+----------+---------+-------------+
|CustomerId|stockCode|sum(Quantity)|
+----------+---------+-------------+
```

```
|     18287|    85173|           48|
|     18287|   85040A|           48|
|     18287|   85039B|          120|
...
|     18287|    23269|           36|
+----------+---------+-------------+
```

分组集取决于聚合级别的 null 值。如果不过滤空值，则会得到不正确的结果。cube、rollup和分组集都是这样。

上面这个任务很简单，但是如果还想要统计股票总数，而不区分客户和股票，该怎么实现？使用传统的group-by语句是不可能实现的，但是使用分组集将会很简单。用户仅需要在分组集中指定他所希望执行聚合操作的级别。其实分组集就是实现了将各种分组统计得到的结果union在一起。

```
-- in SQL
SELECT CustomerId, stockCode, sum(Quantity) FROM dfNoNull
GROUP BY customerId, stockCode GROUPING SETS((customerId, stockCode),())
ORDER BY CustomerId DESC, stockCode DESC

+----------+---------+-------------+
|customerId|stockCode|sum(Quantity)|
+----------+---------+-------------+
|     18287|    85173|           48|
|     18287|   85040A|           48|
|     18287|   85039B|          120|
...
|     18287|    23269|           36|
+----------+---------+-------------+
```

`GROUPING SETS`操作仅在SQL中可用。若想在DataFrame中执行相同的操作，使用`rollup`和`cube`操作可以得到完全相同的结果。接下来看看如何使用这两个操作。

rollup

当我们设置分组的key为多个列时，Spark会分析这些列，并根据各列中存在的实际数值，确定列值组合作为分组的key。而rollup分组聚合是一种多维聚合操作，可以执行多种不同group-by风格的计算。

我们接下来根据时间（`Date`列）和地点（`Country`列）来创建一个rollup分组，并且创建一个新的DataFrame，它将包括所有日期交易的总股票数、每个日期交易的所有股票数，以及在每个日期中每个国家产生的股票交易数：

```
val rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))
  .selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")
  .orderBy("Date")
rolledUpDF.show()

# in Python
rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))\
  .selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")\
  .orderBy("Date")
rolledUpDF.show()
```

```
+----------+--------------+--------------+
|      Date|       Country|total_quantity|
+----------+--------------+--------------+
|      null|          null|       5176450|
|2010-12-01|United Kingdom|         23949|
|2010-12-01|       Germany|           117|
|2010-12-01|        France|           449|
...
|2010-12-03|        France|           239|
|2010-12-03|         Italy|           164|
|2010-12-03|       Belgium|           528|
+----------+--------------+--------------+
```

每列的null值表示不区分该列的总数（比如Country列为null值表示该日期所有地点的总数），而如果在两列中都是null值则表示所有日期和地点的总数：

```
rolledUpDF.where("Country IS NULL").show()

rolledUpDF.where("Date IS NULL").show()
```

```
+----+-------+--------------+
|Date|Country|total_quantity|
+----+-------+--------------+
|null|   null|       5176450|
+----+-------+--------------+
```

cube

cube分组聚合则更进一步，它不同于rollup的分级聚合，而是对所有参与的列值进行所有维度的全组合聚合。也就是说，它不仅基于任一日期对各地点进行汇总聚合，也会基于任一地点对各日期进行汇总聚合。cube分组聚合可以计算如下的分组聚合统计：

- 在所有日期和所有国家发生的交易总数。
- 在每个日期发生于所有国家的交易总数。
- 在每个日期发生于每个国家的交易总数。

- 在所有日期发生于每个国家的交易总数。

方法调用非常相似，但不是调用rollup，而是调用cube：

```
// in Scala
dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))
  .select("Date", "Country", "sum(Quantity)").orderBy("Date").show()

# in Python
from pyspark.sql.functions import sum

dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))\
  .select("Date", "Country", "sum(Quantity)").orderBy("Date").show()
```

```
+----+--------------------+-------------+
|Date|             Country|sum(Quantity)|
+----+--------------------+-------------+
|null|               Japan|        25218|
|null|            Portugal|        16180|
|null|         Unspecified|         3300|
|null|                null|      5176450|
|null|           Australia|        83653|
...
|null|              Norway|        19247|
|null|           Hong Kong|         4769|
|null|               Spain|        26824|
|null|      Czech Republic|          592|
+----+--------------------+-------------+
```

这是一个快速简单获得我们数据表的几乎所有汇总信息的好方法，这个汇总信息可以为以后的数据处理继续使用。

对元数据进行分组

有时，当使用cube和rollup时，希望能够查询聚合级别，以便可以轻松地找到自己想要的信息。可以使用grouping_id来完成此操作，这会在返回结果集中多增加一列。以下示例中的查询将返回四个不同的分组级别ID：

表7-1：分组ID的目的

分组 ID	描述
3	代表最高级别的聚合结果，返回所有customerId和stockCode的股票数量
2	代表针对每个股票的聚合结果，返回每个stockCode的股票数量
1	代表针对每个客户的聚合结果，返回每个customerId的股票数量
0	This will give us the total quantity for individual customerId and stockCode combinations

这有点抽象，下面的实例更好理解：

```
// in Scala
import org.apache.spark.sql.functions.{grouping_id, sum, expr}

dfNoNull.cube("customerId", "stockCode").agg(grouping_id(), sum("Quantity"))
.orderBy(expr("grouping_id()").desc)
.show()

+----------+---------+-------------+-------------+
|customerId|stockCode|grouping_id()|sum(Quantity)|
+----------+---------+-------------+-------------+
|      null|     null|            3|      5176450|
|      null|    23217|            2|         1309|
|      null|   90059E|            2|           19|
...
+----------+---------+-------------+-------------+
```

透视转换

透视转换可以根据某列中的不同行创建多个列。例如，在当前数据中，我们有一个Country列，通过一个透视转换，我们可以对每个Country执行聚合操作，并且以易于查看的方式显示他们：

```
// in Scala
val pivoted = dfWithDate.groupBy("date").pivot("Country").sum()

# in Python
pivoted = dfWithDate.groupBy("date").pivot("Country").sum()
```

在使用了透视转换后，现在DataFrame会为每一个Country和数值型列组合产生一个新列，以及之前的date列。例如，对于USA，就有USA_sum（Quantity），USA_sum（UnitPrice），USA_sum（CustomerID）这些列，对应于我们数据集中的每个数值型列（为USA和每个数值型列构建新列是因为一些聚合操作会作用于这些数值上）。

以下是这个数据的一个示例查询和结果：

```
pivoted.where("date > '2011-12-05'").select("date" ,"`USA_sum(Quantity)`").show()

+----------+-----------------+
|      date|USA_sum(Quantity)|
+----------+-----------------+
|2011-12-06|             null|
|2011-12-09|             null|
|2011-12-08|             -196|
|2011-12-07|             null|
+----------+-----------------+
```

所有列都可以用作透视分组，但是这往往取决于你想做什么样的数据分析。如果某列值的选择性足够低，你就可以依据这列的值构建透视分组，这有利于用户弄清楚数据集的数据模式，帮助用户了解他应该执行什么查询。

用户自定义的聚合函数

用户定义的聚合函数（UDAF）是用户根据自定义公式或业务逻辑定义自己的聚合函数的一种方法。可以使用UDAF来计算输入数据组（与单行相对）的自定义计算。Spark维护单个AggregationBuffer，它用于存储每组输入数据的中间结果。

若要创建UDAF，必须继承UserDefinedAggregateFunction基类并实现以下方法：

- inputSchema用于指定输入参数，输入参数类型为StructType。
- bufferSchema用于指定UDAF中间结果，中间结果类型为StructType。
- dataType用于指定返回结果，返回结果的类型为DataType。
- deterministic是一个布尔值，它指定此UDAF对于某个输入是否会返回相同的结果。
- initialize初始化聚合缓冲区的初始值。
- update描述应如何根据给定行更新内部缓冲区。
- merge描述应如何合并两个聚合缓冲区。
- evaluate将生成聚合最终结果。

下面的例子实现了一个BoolAnd，它将返回（给定列）所有的行是否为 true；如果不是，则返回 false：

```
// in Scala
import org.apache.spark.sql.expressions.MutableAggregationBuffer
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
class BoolAnd extends UserDefinedAggregateFunction {
  def inputSchema: org.apache.spark.sql.types.StructType =
    StructType(StructField("value", BooleanType) :: Nil)
  def bufferSchema: StructType = StructType(
    StructField("result", BooleanType) :: Nil
  )
  def dataType: DataType = BooleanType
  def deterministic: Boolean = true
  def initialize(buffer: MutableAggregationBuffer): Unit = {
```

```
    buffer(0) = true
  }
  def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
    buffer(0) = buffer.getAs[Boolean](0) && input.getAs[Boolean](0)
  }
  def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
    buffer1(0) = buffer1.getAs[Boolean](0) && buffer2.getAs[Boolean](0)
  }
  def evaluate(buffer: Row): Any = {
    buffer(0)
  }
}
```

现在，我们简单地实例化我们的类，也可以将其注册为一个函数：

```
// in Scala
val ba = new BoolAnd
spark.udf.register("booland", ba)
import org.apache.spark.sql.functions._
spark.range(1)
  .selectExpr("explode(array(TRUE, TRUE, TRUE)) as t")
  .selectExpr("explode(array(TRUE, FALSE, TRUE)) as f", "t")
  .select(ba(col("t")), expr("booland(f)"))
  .show()

+----------+----------+
|booland(t)|booland(f)|
+----------+----------+
|      true|     false|
+----------+----------+
```

UDAF目前仅在Scala或Java中可用。但是，在Spark2.3 中，还可以通过注册该函数来调用Scala或Java的UDF和UDAF，就像我们在第 6章的 UDF 部分中所展示的一样。有关更多信息，请参见SPARK-19439（*https://issues.apache.org/jira/browse/SPARK-19439*）。

小结

本章介绍了可以在Spark中执行的不同聚合操作，了解了简单分组到window函数，以及rollup和cube。第8章将介绍如何执行连接操作以将不同的数据源整合在一起。

第8章

连接操作

虽然第7章介绍的应用于单个数据集的聚合操作对实际工作很有用处，但更多时候Spark应用程序将处理多个不同的数据集。因此，连接操作（join）是几乎所有Spark作业的必不可少的部分，Spark处理不同数据的能力意味着可以利用公司内各种数据源。本章不仅介绍Spark中的连接操作以及使用方法，还包括Spark在集群中执行连接操作的具体过程，了解这些知识可以帮助你避免耗尽内存并解决其他一些难题。

连接表达式

join操作比较左侧和右侧数据集的一个或多个键，并评估连接表达式的结果，以此来确定Spark是否将左侧数据集的一行和右侧数据集的一行组合起来。最常见的连接表达式即`equi-join`，它用于比较左侧数据集一行和右侧数据集一行中的指定键是否匹配，相等则组合左侧和右侧数据集的对应行，对于键值不匹配的行则会丢弃。除了`equi-join`之外，Spark还提供很多复杂的连接策略，甚至还能使用复杂类型并在执行连接时执行诸如检查数组中是否存在键的操作。

连接类型

连接表达式决定两行是否应该连接，而连接类型决定了如何连接。Spark中提供了各种不同的连接类型：

- inner join，内部连接（保留左、右数据集内某个键都存在的行）。
- outer join，外部连接（保留左侧或右侧数据集中具有某个键的行）。

- left outer join，左外部连接（保留左侧数据集中具有某个键的行）。
- right outer join，右外部连接（保留右侧数据集中具有某个键的行）。
- left semi join，左半连接（如果某键在右侧数据行中出现，则保留且仅保留左侧数据行）。
- left anti join，左反连接 （如果某键在右侧数据行中没出现，则保留且仅保留左侧数据行）。
- natural join，自然连接（通过隐式匹配两个的数据集之间具有相同名称的列来执行连接）。
- cross join（笛卡尔连接Cartesian join），交叉连接（将左侧数据集中的每一行与右侧数据集中的每一行匹配）。

如果接触过关系型数据库或者Excel表格，那么就容易理解连接不同数据集的概念。我们继续展示每个连接类型的示例。这些例子可以帮助你理解如何使用不同的连接类型来解决自己的问题。为此，我们来创建一些可以在我们例子中使用的简单数据集：

```
// in Scala
val person = Seq(
    (0, "Bill Chambers", 0, Seq(100)),
    (1, "Matei Zaharia", 1, Seq(500, 250, 100)),
    (2, "Michael Armbrust", 1, Seq(250, 100)))
  .toDF("id", "name", "graduate_program", "spark_status")
val graduateProgram = Seq(
    (0, "Masters", "School of Information", "UC Berkeley"),
    (2, "Masters", "EECS", "UC Berkeley"),
    (1, "Ph.D.", "EECS", "UC Berkeley"))
  .toDF("id", "degree", "department", "school")
val sparkStatus = Seq(
    (500, "Vice President"),
    (250, "PMC Member"),
    (100, "Contributor"))
  .toDF("id", "status")

# in Python
person = spark.createDataFrame([
    (0, "Bill Chambers", 0, [100]),
    (1, "Matei Zaharia", 1, [500, 250, 100]),
    (2, "Michael Armbrust", 1, [250, 100])])\
  .toDF("id", "name", "graduate_program", "spark_status")
graduateProgram = spark.createDataFrame([
    (0, "Masters", "School of Information", "UC Berkeley"),
    (2, "Masters", "EECS", "UC Berkeley"),
    (1, "Ph.D.", "EECS", "UC Berkeley")])\
  .toDF("id", "degree", "department", "school")
sparkStatus = spark.createDataFrame([
    (500, "Vice President"),
```

```
    (250, "PMC Member"),
    (100, "Contributor")])\
  .toDF("id", "status")
```

接下来将它们注册为表格，以便在整个章节中使用它们：

```
person.createOrReplaceTempView("person")
graduateProgram.createOrReplaceTempView("graduateProgram")
sparkStatus.createOrReplaceTempView("sparkStatus")
```

内连接

内连接（inner join）判断来自两个DataFrame或表中两行的指定键是否相等，如果相等则将这两行连接在一起并返回（匹配两个DataFrame中指定键相等的任意两行，并将其连接后返回）：

```
// in Scala
val joinExpression = person.col("graduate_program") === graduateProgram.col("id")

# in Python
joinExpression = person["graduate_program"] == graduateProgram['id']
```

若果来自两个DataFrame或者表中指定的键不相等，则会返回一个空的DataFrame。例如下面的表达式会导致生成的DataFrame为空：

```
// in Scala
val wrongJoinExpression = person.col("name") === graduateProgram.col("school")

# in Python
wrongJoinExpression = person["name"] == graduateProgram["school"]
```

内连接是默认的连接操作，因此我们只需指定左侧DataFrame并在JOIN表达式中连接右侧即可：

```
person.join(graduateProgram, joinExpression).show()

-- in SQL
SELECT * FROM person JOIN graduateProgram
  ON person.graduate_program = graduateProgram.id
```

```
+---+----------------+----------------+---------------+---+-------+----------+---
| id|            name|graduate_program|   spark_status| id| degree|department|...
+---+----------------+----------------+---------------+---+-------+----------+---
|  0|   Bill Chambers|               0|          [100]|  0|Masters| School...|...
|  1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|      EECS|...
|  2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|      EECS|...
+---+----------------+----------------+---------------+---+-------+----------+---
```

我们还可以通过传入第三个参数来显式指定连接类型JoinType：

```
// in Scala
var joinType = "inner"

# in Python
joinType = "inner"

person.join(graduateProgram, joinExpression, joinType).show()

-- in SQL
SELECT * FROM person INNER JOIN graduateProgram
  ON person.graduate_program = graduateProgram.id
```

```
+---+----------------+----------------+---------------+---+-------+--------------
| id|            name|graduate_program|   spark_status| id| degree| department...
+---+----------------+----------------+---------------+---+-------+--------------
|  0|   Bill Chambers|               0|          [100]|  0|Masters|     School...
|  1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|       EECS...
|  2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|       EECS...
+---+----------------+----------------+---------------+---+-------+--------------
```

外连接

外连接（outer join）是指两个DataFrame或表中两行的指定键是否相等，如果相等，将这两行连接在一起并返回；如果不相等，将左侧或右侧 DataFrame 中没有匹配的行的各列用null替换，再与左侧或右侧 DataFrame 中已有的行连接在一起返回：

```
joinType = "outer"

person.join(graduateProgram, joinExpression, joinType).show()

-- in SQL
SELECT * FROM person FULL OUTER JOIN graduateProgram
  ON graduate_program = graduateProgram.id
```

```
+----+----------------+----------------+---------------+---+-------+-------------
|  id|            name|graduate_program|   spark_status| id| degree| departmen...
+----+----------------+----------------+---------------+---+-------+-------------
|   1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|       EEC...
|   2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|       EEC...
|null|            null|            null|           null|  2|Masters|       EEC...
|   0|   Bill Chambers|               0|          [100]|  0|Masters|    School...
+----+----------------+----------------+---------------+---+-------+-------------
```

左外连接

左外连接（left outer join）检查两个 DataFrame或表中的键，并包括左侧DataFrame

中的所有行以及右侧 DataFrame中与左侧DataFrame有匹配项的行。如果在右侧DataFrame 中没有对应的行，则Spark将插入null：

```
joinType = "left_outer"

graduateProgram.join(person, joinExpression, joinType).show()

-- in SQL
SELECT * FROM graduateProgram LEFT OUTER JOIN person
  ON person.graduate_program = graduateProgram.id
```

```
+---+-------+----------+-----------+----+----------------+----------------+---
| id| degree|department|     school|  id|            name|graduate_program|...
+---+-------+----------+-----------+----+----------------+----------------+---
|  0|Masters| School...|UC Berkeley|   0|   Bill Chambers|               0|...
|  2|Masters|      EECS|UC Berkeley|null|            null|            null|...
|  1|  Ph.D.|      EECS|UC Berkeley|   2|Michael Armbrust|               1|...
|  1|  Ph.D.|      EECS|UC Berkeley|   1|   Matei Zaharia|               1|...
+---+-------+----------+-----------+----+----------------+----------------+---
```

右外连接

右外连接（right outer join）评估两个 DataFrame或表中的键，并包括来自右侧DataFrame 中的所有行以及左侧DataFrame 中与右侧DataFrame 有匹配项的行。如果在左侧DataFrame中没有对应的行，则Spark将插入null：

```
joinType = "right_outer"

person.join(graduateProgram, joinExpression, joinType).show()

-- in SQL
SELECT * FROM person RIGHT OUTER JOIN graduateProgram
  ON person.graduate_program = graduateProgram.id
```

```
+----+----------------+----------------+---------------+---+-------+------------+
|  id|            name|graduate_program|   spark_status| id| degree|  department|
+----+----------------+----------------+---------------+---+-------+------------+
|   0|   Bill Chambers|               0|          [100]|  0|Masters|School of...|
|null|            null|            null|           null|  2|Masters|        EECS|
|   2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|        EECS|
|   1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|        EECS|
+----+----------------+----------------+---------------+---+-------+------------+
```

左半连接

半连接（semi join）与其他连接操作有点区别，它实际上并不包括右侧DataFrame 中的任何值，它只是查看左侧DataFrame的值是否存在于右侧DataFrame里，如果存在则在连接结果中保留，即使左侧DataFrame中存在重复键，对应行也将保留在结果中。

可以将左半连接（left semi join）看作 DataFrame 上的过滤器，而不是常规连接函数：

```
joinType = "left_semi"

graduateProgram.join(person, joinExpression, joinType).show()

+---+-------+--------------------+-----------+
| id| degree|          department|     school|
+---+-------+--------------------+-----------+
|  0|Masters|School of Informa...|UC Berkeley|
|  1|  Ph.D.|                EECS|UC Berkeley|
+---+-------+--------------------+-----------+

// in Scala
val gradProgram2 = graduateProgram.union(Seq(
    (0, "Masters", "Duplicated Row", "Duplicated School")).toDF())

gradProgram2.createOrReplaceTempView("gradProgram2")

# in Python
gradProgram2 = graduateProgram.union(spark.createDataFrame([
    (0, "Masters", "Duplicated Row", "Duplicated School")]))

gradProgram2.createOrReplaceTempView("gradProgram2")

gradProgram2.join(person, joinExpression, joinType).show()

-- in SQL
SELECT * FROM gradProgram2 LEFT SEMI JOIN person
  ON gradProgram2.id = person.graduate_program

+---+-------+--------------------+-----------------+
| id| degree|          department|           school|
+---+-------+--------------------+-----------------+
|  0|Masters|School of Informa...|      UC Berkeley|
|  1|  Ph.D.|                EECS|      UC Berkeley|
|  0|Masters|      Duplicated Row|Duplicated School|
+---+-------+--------------------+-----------------+
```

左反连接

左反连接（left anti join）与左半连接（left semi join）相反。与左半连接类似的是，它实际上并不包含右侧DataFrame 中的任何值，它只是查看该值是否存在于右侧DataFrame中。但是，左反连接并不保留第二个 DataFrame 中存在的值，而是只保留在第二个 DataFrame 中没有相应键的值。可以将反连接视为一个NOT IN SQL类型的过滤器：

```
joinType = "left_anti"
graduateProgram.join(person, joinExpression, joinType).show()
```

```
-- in SQL
SELECT * FROM graduateProgram LEFT ANTI JOIN person
  ON graduateProgram.id = person.graduate_program

+---+-------+----------+-----------+
| id| degree|department|     school|
+---+-------+----------+-----------+
|  2|Masters|      EECS|UC Berkeley|
+---+-------+----------+-----------+
```

自然连接

自然连接（natural join）不必指定连接条件，它对你执行连接操作中可能要基于的列进行隐式猜测（往往是根据相同的列名），找出匹配列并返回连接结果。目前自然连接支持左连接、右连接和外连接。

隐示的操作始终是危险的!以下查询将给出不正确的结果，因为两个 DataFrame/表都存在一个列名为id的列，但它代表数据集中的不同内容。应该始终谨慎地使用自然连接。

```
-- in SQL
SELECT * FROM graduateProgram NATURAL JOIN person
```

交叉连接（笛卡尔连接）

最后一个连接是交叉连接（cross join）或笛卡尔积（cartesian product）。简单来说，交叉连接是不指定谓词的内部连接。交叉连接会将左侧DataFrame的每一行与右侧DataFrame的每一行进行连接。这将导致结果DataFrame 中包含的行数非常多。如果每个DataFrame中有1000行，则它们的交叉连接结果就有100万(1000x1000) 行。因此，你一定要清楚是否确实需要交叉连接：

```
joinType = "cross"
graduateProgram.join(person, joinExpression, joinType).show()

-- in SQL
SELECT * FROM graduateProgram CROSS JOIN person
  ON graduateProgram.id = person.graduate_program

+---+-------+----------+-----------+---+---------------+----------------+-------
| id| degree|department|     school| id|           name|graduate_program|spar...
+---+-------+----------+-----------+---+---------------+----------------+-------
|  0|Masters| School...|UC Berkeley|  0|   Bill Chambers|               0|    ...
|  1|  Ph.D.|      EECS|UC Berkeley|  2|Michael Armbrust|               1|  [2...
|  1|  Ph.D.|      EECS|UC Berkeley|  1|   Matei Zaharia|               1|[500...
+---+-------+----------+-----------+---+---------------+----------------+-------
```

如果真的想进行交叉连接，则可以显式调用它：

```
person.crossJoin(graduateProgram).show()

-- in SQL
SELECT * FROM graduateProgram CROSS JOIN person
```

```
+---+----------------+----------------+---------------+---+-------+-------------+
| id|            name|graduate_program|   spark_status| id| degree|   departm...|
+---+----------------+----------------+---------------+---+-------+-------------+
|  0|   Bill Chambers|               0|          [100]|  0|Masters|    School...|
...
|  1|   Matei Zaharia|               1|[500, 250, 100]|  0|Masters|    School...|
...
|  2|Michael Armbrust|               1|     [250, 100]|  0|Masters|    School...|
...
+---+----------------+----------------+---------------+---+-------+-------------+
```

只有在真的百分之百需要交叉连接的时候才使用它。在Spark中定义交叉连接时，需要清楚这种做法很危险！ 高级用户可以将会话级别配置`spark.sql.crossJoin.enable`设置为 true，以便允许不带警告的进行交叉连接，而且可以避免Spark为你执行另一个替换连接。

连接操作常见问题与解决方案

执行连接操作时常会遇到一些常见问题，接下来的部分将提供对这些常见问题的解决方法，然后从宏观角度解释下Spark是如何执行连接操作的。我们将在本书的后面部分介绍一些优化方法。

对复杂类型的连接操作

尽管这看起来像是一个挑战，但实际上并不是，任何返回Boolean值的表达式都是有效的连接表达式：

```
import org.apache.spark.sql.functions.expr

person.withColumnRenamed("id", "personId")
  .join(sparkStatus, expr("array_contains(spark_status, id)")).show()

# in Python
from pyspark.sql.functions import expr

person.withColumnRenamed("id", "personId")\
  .join(sparkStatus, expr("array_contains(spark_status, id)")).show()

-- in SQL
```

```
SELECT * FROM
  (select id as personId, name, graduate_program, spark_status FROM person)
  INNER JOIN sparkStatus ON array_contains(spark_status, id)
```

```
+--------+---------------+---------------+--------------+---+-------------+
|personId|           name|graduate_program|  spark_status| id|       status|
+--------+---------------+---------------+--------------+---+-------------+
|       0|  Bill Chambers|              0|         [100]|100|  Contributor|
|       1|  Matei Zaharia|              1|[500, 250, 100]|500|Vice President|
|       1|  Matei Zaharia|              1|[500, 250, 100]|250|   PMC Member|
|       1|  Matei Zaharia|              1|[500, 250, 100]|100|  Contributor|
|       2|Michael Armbrust|              1|    [250, 100]|250|   PMC Member|
|       2|Michael Armbrust|              1|    [250, 100]|100|  Contributor|
+--------+---------------+---------------+--------------+---+-------------+
```

处理重复列名

连接操作中棘手的问题是在生成的 DataFrame 中处理重复的列名。DataFrame 中的每一列在Spark的SQL引擎Catalyst中都有唯一的ID。它仅在内部可见，不能直接引用。当DataFrame的列的名字相同时，指定引用一个列会出现问题。

在以下两种情况下可能会发生问题：

- 指定的连接表达式没有将执行连接操作的两个同名列的其中一个key删除。
- 连接操作的两个DataFrame中的非连接列同名。

接下来创建一个问题数据集，以便可以用来说明这些问题：

```
val gradProgramDupe = graduateProgram.withColumnRenamed("id", "graduate_program")

val joinExpr = gradProgramDupe.col("graduate_program") === person.col(
  "graduate_program")
```

注意现在有两个graduate_program列，我们将基于该列做连接操作：

```
person.join(gradProgramDupe, joinExpr).show()
```

当我们引用其中一个列时，就会出现问题：

```
person.join(gradProgramDupe, joinExpr).select("graduate_program").show()
```

这个代码将返回一个错误。在这个特定的例子中，Spark会产生如下消息：

```
org.apache.spark.sql.AnalysisException: Reference 'graduate_program' is
ambiguous, could be: graduate_program#40, graduate_program#1079.;
```

方法1： 采用不同的连接表达式

当有两个同名的键时，最简单的解决方法是将连接表达式从布尔表达式更改为字符串或序列。这会在连接过程中自动删除其中一个列：

```
person.join(gradProgramDupe,"graduate_program").select("graduate_program").show()
```

方法2： 连接后删除列

另一种方法是在连接后删除有冲突的列。在执行此操作时，我们需要通过原始源DataFrame引用该列，如果连接使用相同的键名，或者源 DataFrame具有同名的列，则可以执行此操作：

```
person.join(gradProgramDupe, joinExpr).drop(person.col( “graduate_program"))
  .select( “graduate_program").show()

val joinExpr = person.col( “graduate_program") === graduateProgram.col( “id")
person.join(graduateProgram, joinExpr).drop(graduateProgram.col( “id")).show()
```

如果显式指定列名，则Spark不需要解析该列而可以直接提取对应列。注意对该列的引用是通过.col方法而不是通过column函数的，这使我们可以通过其特定的ID隐式指定该列。

方法3： 在连接前重命名列

如果在连接之前重命名其中一个列，则可以完全避免此问题：

```
val gradProgram3 = graduateProgram.withColumnRenamed("id", "grad_id")
val joinExpr = person.col("graduate_program") === gradProgram3.col("grad_id")
person.join(gradProgram3, joinExpr).show()
```

Spark如何执行连接

要了解Spark如何执行连接操作，需要了解两个起作用的核心模块：点对点通信模式（node-to-node communication strategy）和逐点计算模式（per node computation strategy）。这些看起来好像与业务问题没关系。然而，理解Spark执行连接操作的过程将会决定Spark作业是否会较快完成，还是会执行失败。

通信策略

在连接过程中，Spark以两种不同的方式处理集群通信问题。它要么执行导致all-to-all通信的shuffle join，要么就采用broadcast join。这里我们不会介绍太多细节，随着时间的推移，其中一些内部优化可能会随着基于成本的优化器的新改进和通信策略的改

进而发生改变。出于这个原因，我们将专注于在宏观层面的例子，以帮助准确理解某些更常见情况，并正确使用低级功能来加速你的作业。

简而言之，假设Spark中仅存在大表或小表，虽然这种方式具有局限性 (如果你有一个"中等大小的表"那事情就不一样了)，但这种二分状态有助于理解概念。

大表与大表连接

当用一个大表连接另一个大表时，最终就是个shuffle join，如图 8-1所示。

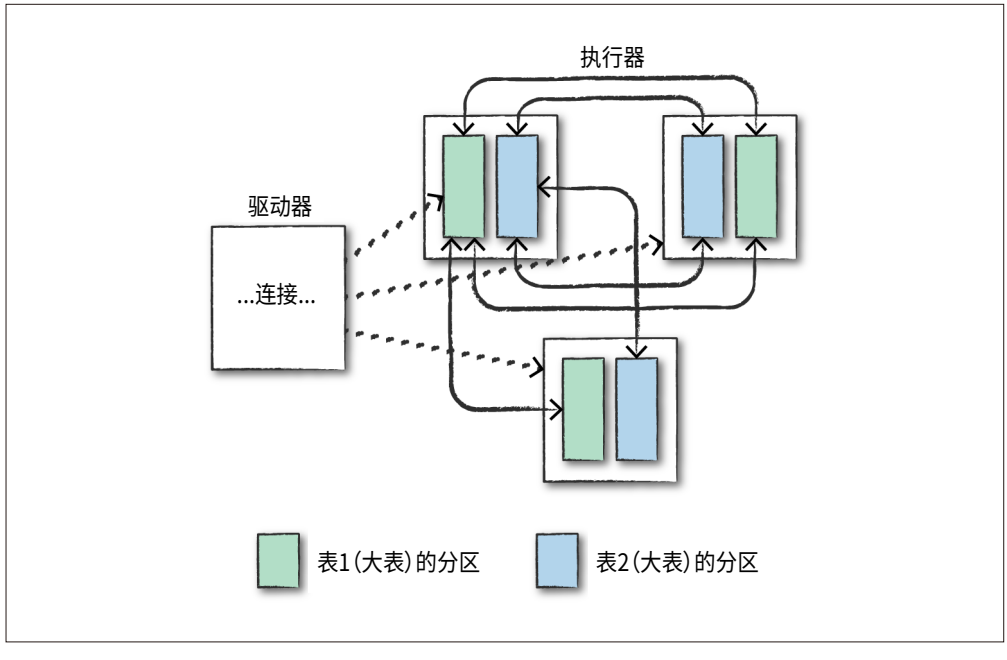


图8-1：连接两个大表

执行shuffle join则每个节点都与所有其他节点进行通信，并根据哪个节点具有（你正在用于连接的）某个键或某一组键来共享数据。由于网络会因通信量而阻塞，所以这种方式很耗时，特别是如果数据没有合理分区的情况下。

此连接描述了两个大数据表连接的过程，例如一家每天从物联网接收数十亿条消息的公司，并且需要检测日常变化，这需要基于`deviceId`列、`messageType`列、`date`列和（`data-1`）天的列上进行连接操作。

图8-1中，DataFrame 1 和 DataFrame 2 都是数据量很大的DataFrame。这意味着如果

没有合理进行数据分区，所有工作节点（以及潜在的每个分区）在整个连接过程中都需要相互通信。

大表与小表连接

当表的大小足够小以便能够放入单个节点内存中且还有空闲空间的时候，我们就可以优化连接。尽管我们仍然可以使用上面那种大表-大表的shuffle join通信策略，但使用broadcast join通常更高效。也就是说，我们可以把数据量较小的DataFrame复制到集群中的所有工作节点上。虽然听起来很耗时，但这样做会避免在整个连接过程中执行all-to-all的通信，只需在开始时执行一次，然后让每个工作节点独立执行作业，而无需等待其他工作节点，也无需与其他工作节点通信，如图 8-2所示。

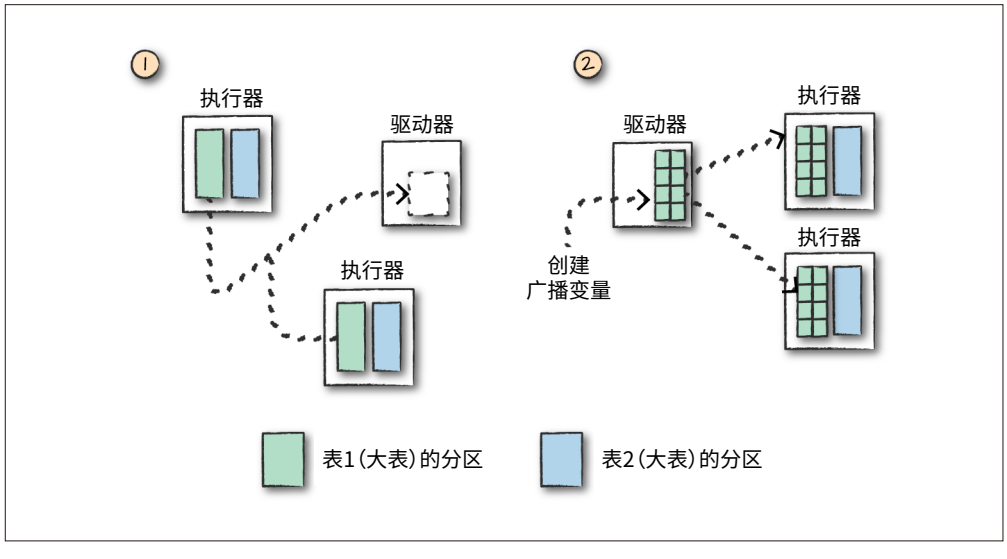


图8-2：广播连接

这种连接通信模式在开始时会有一次大的通信，就像大表之间连接时一样。但是，在第一次通信后，节点之间将不再有其他的通信。这意味着连接操作将在每个节点上独立执行，而最大限度的利用CPU资源专注于执行连接操作。对于我们当前的数据集，通过查看Spark解释方案可以看到Spark自动将其设置为基于广播通信模式的连接操作：

```
val joinExpr = person.col( “graduate_program") === graduateProgram.col( “id")

person.join(graduateProgram, joinExpr).explain()

== Physical Plan ==
*BroadcastHashJoin [graduate_program#40], [id#5....
```

```
:- LocalTableScan [id#38, name#39, graduate_progr...
+- BroadcastExchange HashedRelationBroadcastMode(....
   +- LocalTableScan [id#56, degree#57, departmen....
```

通过DataFrame API还可以显式地告知优化器，使用broadcast函数作用于较小DataFrame上并执行广播通信模式的连接操作。在下面这个例子中，我们将看到相同的物理执行计划，然而并不总是如此：

```
import org.apache.spark.sql.functions.broadcast
val joinExpr = person.col("graduate_program") === graduateProgram.col("id")
person.join(broadcast(graduateProgram), joinExpr).explain()
```

SQL 接口支持显式指定连接操作的物理通信模式，但这些并不是强制执行的，所以优化程序可能会选择忽略它们。可以使用特殊的注释语法来设置一个提示，MAPJOIN，BROADCAST和BROAD CASTJOIN都是指示同样广播连接：

```
-- in SQL
SELECT /*+ MAPJOIN(graduateProgram) */ * FROM person JOIN graduateProgram
  ON person.graduate_program = graduateProgram.id
```

这也不是毫无开销的，如果你试图广播太大的表，可能导致驱动器节点崩溃（因为广播大表的代价是昂贵的），这很可能是Spark未来优化计划的一部分。

小表与小表连接

当执行小表的连接时，通常最好让Spark决定如何连接它们。如果你注意到奇怪的行为，你可以随时强制执行广播连接。

小结

在本章中，我们讨论了连接，它可能是最常见的数据操作之一。有一件事没有提到但值得考虑：如果在连接之前正确划分数据，则可以更高效的执行连接操作，因为即使选择的是shuffle join，如果来自两个不同DataFrame的数据已经位于同一台机器，Spark可以避免Shuffle操作。可以试着对数据进行预先分区，看看在执行连接时速度会不会提升。在第9章中，我们将讨论Spark的数据源API。两个执行连接操作的数据集在不同顺序时，也会影响性能，因为连接操作的在前面或后面的数据集往往起到过滤器的作用，过滤器的大小会决定网络传输的通信开销，这是提高你Spark作业性能的一个很简单的方法。

下一章将不再介绍用户操作，而是介绍使用结构化API进行读取和写入数据的操作。

第9章

数据源

本章正式介绍可以在Spark中使用的各种其他数据源（data source）以及来自社区构建的各种其他数据源。Spark包含六大"核心" 数据源以及由社区生成的许多外部数据源，提供对各种数据源的读写能力支持，以及为社区创造贡献无疑是Spark最擅长做的。以下是Spark的核心数据源：

- CSV。
- JSON。
- Parquet。
- ORC。
- JDBC/ODBC连接。
- 纯文本文件。

如前所述，Spark有许多由社区创建的数据源。这里只是一小部分样本：

- Cassandra。
- HBase。
- MongoDB。
- AWS Redshift。
- XML。
- 其他数据源。

本章的目标是让你学习后能够实现读写Spark核心数据源，并在涉及到第三方数据源时知道需要查找什么。为了实现该目标，我们将重点介绍需要知道和理解的核心概念。

数据源 API 的结构

在介绍对特定格式的读写操作之前，先来看看数据源API（Data Source API）的整体组织结构。

Read API 结构

读取数据的核心结构如下：

```
DataFrameReader.format(...).option("key", "value").schema(...).load()
```

我们将使用此格式来读取所有数据源。format是可选的，默认情况下Spark将使用Parquet格式，option使你能配置键值对（key-value）来参数化读取数据的方式。最后，如果数据源包含某种schema或你想使用模式推理（schema inference），则可以选择指定schema。每种格式都有一些必选项，下面介绍具体格式时候会给出更多细节。

Spark社区中有很多简写符号，数据源读取 API 也不例外。我们会介绍这些简写符号并在整本书中保证前后一致。

数据读取基础

Spark数据读取使用`DataFrameReader`，通过`SparkSession`的`read`属性得到：

```
Spark.read
```

有了DataFrame reader之后，我们需要指定几个值：

- format。
- schema。
- read 模式。
- 一系列option选项。

format，option和schema都会返回一个DataFrameReader，它可以进行进一步的转换，并且都是可选的（那些仅有唯一可选项的就只能选择唯一可选项了）。每个数据源都有一组特定的选项，用于设置如何将数据读入Spark（下面将很快会介绍这些选项）。至少，你需要为 DataFrameReader提供 一个读取路径。

下面是一个整体结构的例子：

```
spark.read.format( “csv")
  .option( “mode", “FAILFAST")
  .option( “inferSchema", “true")
  .option( “path", “path/to/file(s)")
  .schema(someSchema)
  .load()
```

有很多种配置参数的方法，比如可以构造一个map（作为参数）传入进行配置。现在，我们继续使用刚介绍的简单明了的方式。

读取模式

从外部源读取数据很容易会遇到错误格式的数据，尤其是在处理半结构化数据时。读取模式指定当Spark遇到错误格式的记录时应采取什么操作，表 9-1 中列出了读取模式的选项。

表9-1：Spark的读取模式

读取模式	说明
permissive	当遇到错误格式的记录时，将所有字段设置为null并将所有错误格式的记录放在名为_corrupt_record字符串列中
dropMalformed	删除包含错误格式记录的行
failFast	遇到错误格式的记录后立即返回失败

默认是permissive。

Write API 结构

写数据的核心结构如下：

```
DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save()
```

我们将使用此格式向所有数据源写入数据。format是可选的，默认情况下Spark将使用arquet 格式，option仍用于配置写出数据的方法，PartitionBy，bucketBy和sortBy仅适用基于文件的数据源，你可以使用这些方法来控制写出目标文件的具体结构。

写数据基础

写数据与读取数据非常相似，不同的是，需要用到的是DataFrameWriter而不是DataFrameReader了。因为总是需要将数据写入一些给定数据源中，所以我们通过每个DataFrame的write属性来获取DataFrameWriter：

```
// in Scala
dataFrame.write
```

有了DataFrameWriter之后，我们需要指定三个值：format、一系列option选项和save模式，并且必须至少提供一条写入路径（来指定目标地址）。我们将介绍一些option可选项，请注意不同数据源的option可选项未必相同。

```
// in Scala
dataframe.write.format( “csv")
  .option( “mode", “OVERWRITE")
  .option( “dateFormat", “yyyy-MM-dd")
  .option( “path", “path/to/file(s)")
  .save()
```

保存模式

保存模式指明如果Spark在指定目标路径发现有其他数据占用时应采取什么操作。表9-2列出保存模式的选项。

表9-2：Spark的保存模式

保存模式	描述
append	将输出文件追加到目标路径已存在的文件上或目录的文件列表
overwrite	将完全覆盖目标路径中已存在的任何数据
errorIfExists	如果目标路径已存在数据或文件，则抛出错误并返回写入操作失败
ignore	如果目标路径已存在数据或文件，则不执行任何操作

默认值为errorIfExists，也就是说如果目标路径已有数据存在数据则Spark立即写入失败。

我们已经介绍了很多使用数据源时需要了解的核心概念，现在详细介绍一下Spark的各种本地数据源。

CSV 文件

CSV意即逗号分隔值（comma-separated values），这是一种常见的文本文件格式，

其中每行表示一条记录，用逗号分隔记录中的每个字段。虽然CSV文件看起来结构良好，但实际上它会遇到各种各样的问题，是最难处理的文件格式之一，这是因为实际应用场景中遇到的数据内容或数据结构并不会那么规范。因此，CSV读取程序包含大量选项，通过这些选项可以帮助你解决像忽略特定字符等的这种问题，比如当一列的内容也以逗号分隔时，需要识别出该逗号是列中的内容，还是列间分隔符。

CSV 选项

表 9-3列出 CSV 读取程序中的可选项。

表9-3：CSV 数据源选项

read/write	Key	取值范围	默认值	说明
Both	sep	任意单个字符串字符	,	用作每个字段和值的分隔符的单个字符
Both	header	true，false	false	一个布尔标记符，用于声明文件中的第一行是否为列的名称
Both	escape	任意字符串字符	\	用于转义的字符
Both	inferSchema	true，false	false	指定在读取文件时Spark是否自动推断列类型
Both	ignoreLeadingWhiteSpace	true，false	false	声明是否应跳过读取值中的前导空格
Both	ignoreTrailingWhiteSpace	true，false	false	声明是否应跳过读取值的尾部空格
Both	nullValue	任意字符串字符	""	声明在文件中什么字符表示null值
Both	nanValue	任意字符串字符	NaN	声明什么字符表示 CSV 文件中的 NaN 或缺失字符
Both	positiveInf	任意字符串或字符	Inf	声明什么字符表示正无穷大

表9-3：CSV 数据源选项（续）

read/write	Key	取值范围	默认值	说明
Both	negativeInf	任何字符串或字符	-Inf	声明什么字符表示负无穷大
Both	Compression 或codec	None，uncompressed, bzip2, deflate, gzip, lz4, or snappy	none	声明Spark应该使用什么压缩编解码器来读取或写入文件
Both	dateFormat	任何符合 Java 的 SimpleDataFormat 的字符串或字符	yyyy-MM-dd	日期类型的日期格式
Both	timestampFormat	任何符合 Java 的 SimpleDataFormat 的字符串或字符	MMdd 'T' HH: mm ss.SSSZZ	时间戳类型时间戳格式
Read	maxColumns	任意整数	20480	声明文件中的最大列数
Read	maxCharsPerColumn	任意整数	1000000	声明列中的最大字符数
Read	escapeQuotes	true，false	true	声明Spark是否应该转义在行中找到的引号
Read	maxMalformedLogPerPartition	任意整数	10	设置Spark将为每个分区记录错误格式的行的最大数目，超出此数目的格式错误的记录将被忽略
Write	quoteAll	true，false	false	指定是否将所有值括在引号中，而不是仅转义具有引号字符的值
Read	multiline	true，false	false	此选项用于读取多行CSV文件，其中CSV文件中的每个逻辑行可能跨越文件本身中的多行

读CSV文件

与读取其他格式一样，要读取CSV文件必须首先为该特定格式创建一个DataFrameReader。这里我们将格式指定为CSV：

```
spark.read.format("csv")
```

然后，我们可以选择指定schema和modes选项。我们设置一些选项，其中有些是本书开头介绍过的，另一些还没有介绍。我们将对CSV文件进行设置：header为true、mode为FAILFAST，以及inferSchema为true：

```
// in Scala
spark.read.format( “csv")
  .option( “header", “true")
  .option( “mode", “FAILFAST")
  .option( “inferSchema", “true")
  .load( “some/path/to/file.csv")
```

如上所述，我们可以使用该模式来指定对错误格式数据的容错数量。例如，可以使用这些modes以及在第5章中创建的schema以确保文件符合我们所期望的数据：

```
// in Scala
import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}
val myManualSchema = new StructType(Array(
  new StructField( “DEST_COUNTRY_NAME", StringType, true),
  new StructField( “ORIGIN_COUNTRY_NAME", StringType, true),
  new StructField( “count", LongType, false)
))
spark.read.format( “csv")
  .option( “header", “true")
  .option( “mode", “FAILFAST")
  .schema(myManualSchema)
  .load( “/data/flight-data/csv/2010-summary.csv")
  .show(5)
```

当我们不希望数据以某种格式出现时，事情就变得棘手了，但这种情况总会出现。例如，我们采取当前的schema，但是将所有列类型更改为LongType，这与实际的schema并不匹配，但此时Spark并不会报错。只有当Spark实际读取数据时，问题才会暴露出来。一旦我们开始执行Spark作业，由于数据不符合指定的schema，它会（在执行作业之后）立即失败：

```
// in Scala
val myManualSchema = new StructType(Array(
                     new StructField( “DEST_COUNTRY_NAME", LongType, true),
                     new StructField( “ORIGIN_COUNTRY_NAME", LongType, true),
                     new StructField( “count", LongType, false) ))
```

```
spark.read.format( “csv")
  .option( “header", “true")
  .option( “mode", “FAILFAST")
  .schema(myManualSchema)
  .load( “/data/flight-data/csv/2010-summary.csv")
  .take(5)
```

通常，Spark只会在作业执行而不是DataFrame定义时发生失败，例如，即使我们指向一个不存在的文件也是这样。这是由于惰性评估造成的，在第2章中我们学过这个概念 。

写CSV文件

同读取数据一样，写CSV 文件时，也有多种选项 (在表 9-3中列出) 用于写入数据。写入选项表是读取选项表的子集，因为很多选项在写入数据时并不适用 (例如maxColumns和inferSchema)。下面是一个示例：

```
// in Scala
val csvFile = spark.read.format( “csv")
  .option( “header", “true").option( “mode", “FAILFAST").schema(myManualSchema)
  .load( “/data/flight-data/csv/2010-summary.csv")

# in Python
csvFile = spark.read.format( “csv")\
  .option( “header", “true")\
  .option( “mode", “FAILFAST")\
  .option( “inferSchema", “true")\
  .load( “/data/flight-data/csv/2010-summary.csv")
```

例如，我们可以很容易地读取CSV文件内容并写入TSV文件：

```
// in Scala
csvFile.write.format( “csv").mode( “overwrite").option( “sep", “\t")
  .save( “/tmp/my-tsv-file.tsv")

# in Python
csvFile.write.format( “csv").mode( “overwrite").option( “sep", “\t")\
  .save( “/tmp/my-tsv-file.tsv")
```

当列出目标目录时，你可以看到my-tsv-file实际上是一个包含大量文件的文件夹：

```
$ ls /tmp/my-tsv-file.tsv/

/tmp/my-tsv-file.tsv/part-00000-35cf9453-1943-4a8c-9c82-9f6ea9742b29.csv
```

这实际上对应着写出时DataFrame的数据分片。如果在写出之前对数据进行重新分片，最终会得到不同的文件数量。在本章末尾将详细讨论该问题。

JSON 文件

有JavaScript背景知识的人可能很熟悉JavaScript Object Notation（简称JSON）。在处理JSON数据之前需要了解的是，在Spark中，我们提及的JSON文件指的是换行符分隔的JSON，每行必须包含一个单独的、独立的有效JSON对象，这与包含大的JSON对象或数组的文件是有区别的。

换行符分隔JSON对象还是一个对象可以跨越多行，这个可以由`multiLine`选项控制，当`multiLine`为`true`时，则可以将整个文件作为一个`json`对象读取，并且Spark将其解析为DataFrame。换行符分隔的JSON实际上是一种更稳定的格式，因为它可以在文件末尾追加新记录（而不是必须读入整个文件然后再写出），我们也建议使用换行符分隔的格式。换行符分隔的JSON格式流行的另一个关键原因是JSON对象具有结构化信息，并且（基于JSON的）JavaScript也支持基本类型，这使得它更易用，Spark可以代表我们完成很多对结构化数据的操作。你会发现，由于JSON结构化对象封装的原因，导致JSON文件选项比CSV的要少得多。

JSON 选项

表 9-4列出 JSON 对象可用的选项以及说明。

表9-4：JSON数据源选项

read/write	Key	取值范围	默认值	说明
Both	Compression或codec	None, uncompressed, bzip2, deflate, gzip, lz4, 或snappy	none	声明Spark应该使用什么压缩编解码器来读取或写入文件
Both	dateFormat	任何符合 Java SimpleDataFormat格式的字符串或字符	yyyy-MM-dd	为日期类型的列声明日期格式
Both	timestampFormat	任何符合 Java SimpleDataFormat格式的字符串或字符	yyyy-MM-dd'T'HH:mm:ss.SSSZZ	为时间戳类型的列声明时间戳格式
Read	primitiveAsString	true，false	false	将所有原始值推断为字符串类型

表9-4：JSON数据源选项（续）

read/write	Key	取值范围	默认值	说明
Read	allowComments	true，false	false	忽略JSON记录中的Java / C ++样式注释
Read	allowUnquotedFieldNames	true，false	false	允许不带引号 的JSON 字段名
Read	allowSingleQuotes	true，false	true	除双引号外，还允许使用单引号
Read	allowNumericLeadingZeros	true，false	false	允许数字中存在前导零 (例如，00012)
Read	allowBackslashEscAPIngAny	true，false	false	允许反斜杠机制接受所有字符
Read	columnNameOfCorruptRecord、	Any string	spark.sql.column & NameOfCorruptRecord	允许重命名permissive 模式下添加的新字段，会覆盖重写
Read	multiLine	true，false	false	允许读取非换行符分隔的JSON文件

读取换行符分隔的 JSON 文件仅仅是指定的格式（format）和选项有所不同：

```
spark.read.format("json")
```

读JSON文件

我们看一个读取 JSON 文件的例子，并比较其中的选项：

```
// in Scala
spark.read.format( “json").option( “mode", “FAILFAST").schema(myManualSchema)
  .load( “/data/flight-data/json/2010-summary.json").show(5)

# in Python
spark.read.format( “json").option( “mode", “FAILFAST")\
  .option( “inferSchema", “true")\
  .load( “/data/flight-data/json/2010-summary.json").show(5)
```

写JSON 文件

写JSON 文件和读JSON文件一样简单，而且无论要写出的数据是什么格式。因此，我

们可以重用以前创建的 CSV DataFrame，将其里面内容写入到一个JSON 文件里。这也遵循我们之前指定的规则：每个数据分片作为一个文件写出，而整个DataFrame将输出到一个文件夹。文件中每行仍然代表一个JSON对象：

```
// in Scala
csvFile.write.format( “json").mode( “overwrite").save( “/tmp/my-json-file.json")

# in Python
csvFile.write.format( “json").mode( “overwrite").save( “/tmp/my-json-file.json")
$ ls /tmp/my-json-file.json/

/tmp/my-json-file.json/part-00000-tid-543....json
```

Parquet文件

Parquet是一种开源的面向列的数据存储格式，它提供了各种存储优化，尤其适合数据分析。Parquet提供列压缩从而可以节省空间，而且它支持按列读取而非整个文件地读取。作为一种文件格式，Parquet与Apache Spark配合得很好，而且实际上也是Spark的默认文件格式。我们建议将数据写到Parquet以便长期存储，因为从Parquet文件读取始终比从JSON文件或CSV文件效率更高。Parquet的另一个优点是它支持复杂类型，也就是说如果列是一个数组（CSV文件无法存储数组列）、map映射或struct结构体，仍可以正常读取和写入，不会出现任何问题。以下代码指定Parquet为文件读取格式：

```
spark.read.format("parquet")
```

读Parquet文件

Parquet的可选项很少，因为它在存储数据时执行本身的schema，因此，你只需要指定一下Parquet格式。如果我们对DataFrame的模式（schema）有严格的要求，则可以设置`schema`。一般来说，在读取的时候使用默认的schema，所以就不需要再设置了，这类似于CSV文件的`infer Schema`。然而由于schema内置于文件中，不需要推断，所以Parquet文件格式更强大。

以下是一些从parquet中读取数据的简单例子：

```
spark.read.format( “parquet")

// in Scala
spark.read.format( “parquet")
  .load( “/data/flight-data/parquet/2010-summary.parquet").show(5)
```

```
# in Python
spark.read.format( “parquet")\
  .load( “/data/flight-data/parquet/2010-summary.parquet").show(5)
```

Parquet可选项

由于Parquet含有明确定义且与Spark概念密切一致的规范，所以它只有很少的可选项，实际上只有两个，表9-5列出了可选项。

虽然只有两个选项，如果你使用的是不兼容的Parquet文件，仍然会遇到问题。当使用不同版本的Spark（尤其是旧版本时）写入Parquet文件时要小心，因为这可能会导致让人头疼的问题。

表9-5：Parquet数据源选项

read/write	Key	取值范围	默认值	说明
write	compression或codec	None, uncompressed, bzip2, deflate, gzip, lz4,或 snappy	None	声明Spark应该使用什么压缩编解码器来读取或写入文件
read	merge Schema	true，false	配置值spark.sql.par quet.mergeSchema	增量地添加列到同一表/文件夹中的Parquet文件里，此选项用于启用或禁用此功能

写Parquet文件

写Parquet文件和读取它一样简单，只需指定文件的位置即可。这里应用相同的分片规则：

```
// in Scala
csvFile.write.format( “parquet").mode( “overwrite")
  .save( “/tmp/my-parquet-file.parquet")

# in Python
csvFile.write.format( “parquet").mode( “overwrite")\
  .save( “/tmp/my-parquet-file.parquet")
```

ORC文件

ORC是为Hadoop作业而设计的自描述、类型感知的列存储文件格式。它针对大型流

式数据读取进行优化，但集成了对快速查找所需行的相关支持。实际上，读取ORC文件数据时没有可选项，这是因为Spark非常了解该文件格式。一个问题常被问到：ORC和Parquet有什么区别？在大多数情况下，它们非常相似，本质区别是，Parquet针对Spark进行了优化，而ORC则是针对Hive进行了优化。

读ORC文件

以下是读ORC文件的示例：

```
// in Scala
spark.read.format("orc").load("/data/flight-data/orc/2010-summary.orc").show(5)

# in Python
spark.read.format("orc").load("/data/flight-data/orc/2010-summary.orc").show(5)
```

写Orc文件

写ORC文件的方法，与之前看到的方式相同，先指定格式然后保存文件：

```
// in Scala
csvFile.write.format("orc").mode("overwrite").save("/tmp/my-json-file.orc")

# in Python
csvFile.write.format("orc").mode("overwrite").save("/tmp/my-json-file.orc")
```

SQL数据库

因为很多系统的标准语言都采用SQL，所以SQL数据源是很强大的连接器，只要支持SQL就可以和许多系统兼容。例如，你可以连接到MySQL数据库、PostgreSQL数据库或Oracle数据库，还可以连接到SQLite，我们的例子就是连接SQLite。数据库并不是一些数据文件，而是一个系统，你有许多连接数据库的方式可供选择。你需要考虑诸如身份验证和连通性等问题（即需要确定Spark集群网络是否容易连接到数据库系统所在的网络上）。

介绍数据库配置并不是本书主题，我们提供了一个在SQLite上运行的简单参考示例来帮助你理解数据库相关的基础。选择SQLite介绍是因为这样可以跳过许多细节介绍，因为SQLite可以在本地计算机以最简配置工作，但在分布式环境中不行。如果想在分布式环境中运行这些示例，则需要连接其他数据库

SQLite 入门

SQLite 是目前使用最多的数据库引擎，它功能强大、速度快且易于理解，这是因为SQLite数据库只是一个文件。我们已经将SQLite源代码文件放在本书的官方代码库（*https://github.com/databricks/Spark-The-Definitive-Guide/tree/master/data/flight-data/jdbc*）中，所以你可以很容易地启动并运行，只需将该文件下载到你的本地计算机上，即可对该文件进行读写操作。我们使用的是SQLite，但这里的所有代码都适用于传统的关系数据库，如 MySQL。主要区别在于连接到数据库时所需要指定的连接选项略有不同，当我们使用SQLite 时，并没有用户或密码的概念。

尽管SQLite是一个很好的参考示例，但它可能并不适合作为生产中应用的数据库。此外，由于SQLite需要在写入时锁定整个数据库，所以SQLite在分布式环境下不一定能够正常工作。我们在这里提供的示例在MySQL或PostgreSQL中也能正常运行。

读写这些数据库需要两步：在Spark类路径中为指定的数据库包含Java Database Connectivity（JDBC）驱动，并为连接驱动器提供合适的JAR包。例如，为了能够读写PostgreSQL数据库，可能需要运行如下代码：

```
./bin/spark-shell \
--driver-class-path postgresql-9.4.1207.jar \
--jars postgresql-9.4.1207.jar
```

同其他数据源一样，在读取和写入 SQL 数据库时，有许多选项可用。我们当前示例中只展示了部分选项的使用方式，而在表9-6列出了使用JDBC数据库时可以设置的所有选项。

表9-6：JDBC 数据源选项

属性名称	说明
Url	表示要连接的JDBC URL，可以在URL中指定特定源的连接属性；例如，*jdbc:postgresql:///localhost/test?user=fred&password=secret*
dbtable	表示要读取的JDBC表， 请注意，可以使用SQL查询的FROM子句中任何有效内容。 例如，你可以在圆括号中使用子查询，而不是全表查询
driver	用于连接到此URL的JDBC驱动器的类名

表9-6：JDBC 数据源选项（续）

属性名称	说明
partitionColumn，lowerBound，upperBound	如果指定了这些选项中的任何一个，则必须设置其他所有选项。另外，还必须指定numPartitions。 这些属性描述了如何在从多个worker并行读取时对表格进行划分。partitionColumn是要分区的列，必须是整数类型。 请注意，lowerBound和upperBound仅用于确定分区跨度，而不用于过滤表中的行。 因此，表中的所有行都将被划分并返回。 该选项仅适用于读操作
numPartitions	在读取和写入数据表时，数据表可用于并行的最大分区数，这也决定了并发 JDBC 连接的最大数目。如果要写入的分区数超过此限制，则通过在写入前调用coalesce(numPartitions)来将分区数降到符合此限制
fetchsize	表示JDBC每次读取多少条记录。这个设置与JDB驱动器的性能有关系，JDBC驱动器默认该值为低获取行数（例如，具有10行的Oracle）。该选项仅适用于读操作
batchsize	表示JDBC批处理大小，用于指定每次写入多少条记录。 这个选项与JDBC 驱动器性能有关系，该选项仅适用于写操作， 默认值为1000
isolationLevel	表示数据库的事务隔离级别（适用于当前连接）。它可以取值为NONE、READ_COMMITTED，READ_UNCOMMITTED，REPEATABLE_READ或 SERIALIZABLE，分别对应于 JDBC的Connection对象定义的标准事务隔离级别。默认值为READ_UNCOMMITTED，此选项仅适用于写操作。更多信息，请参阅java.sql.Connection文档
truncate	这是一个与JDBC写入相关的选项。 当spark要执行覆盖表操作时，即启用SaveMode.Overwrite，Spark将截取现有表，而不是删除之后再重新创建它。 这样可以提高效率，并防止表元数据（例如索引）被删除。但是，在某些情况下，例如新数据具有不同的schema时，它并不起作用。默认值为false，该选项仅适用于写操作
createTableOptions	这是一个与JDBC写入相关的选项。用于在创建表时设置特定数据库的表和分区选项，例如， CREATE TABLE t (name string) ENGINE=InnoDB。该选项仅适用于写操作
createTableColumnTypes	表示创建表时使用的数据库列数据类型，而不使用默认值。 应该使用与CREATE TABLE列语法相同的格式（例如，“name CHAR（64），comments VARCHAR（1024）"）来指定数据类型信息，指定的类型应是有效的Spark SQL数据类型。 该选项仅适用于写操作

从SQL数据库中读取数据

从SQL数据库读取文件和之前看到的其他数据源没有什么不同，与其他数据源一样，我们先指定格式（format）和选项，然后加载数据：

```
// in Scala
val driver =  "org.sqlite.JDBC"
val path =  "/data/flight-data/jdbc/my-sqlite.db"
val url = s"jdbc:sqlite:/${path}"
val tablename =  "flight_info"

# in Python
driver =  "org.sqlite.JDBC"
path =  "/data/flight-data/jdbc/my-sqlite.db"
url =  "jdbc:sqlite:" + path
tablename =  "flight_info"
```

定义连接属性后，可以测试与数据库的连接以确保其正常工作。这是一个很好的故障排除方法，可以确认你的数据库（至少）对Spark驱动器可用。这对SQLite来说意义不大，因为它只是计算机上的一个文件（随时保证可用），但如果你使用的是MySQL之类的数据库，则可以使用以下命令测试连接：

```
import java.sql.DriverManager
val connection = DriverManager.getConnection(url)
connection.isClosed()
connection.close()
```

如果连接成功，你就能继续执行下一步，接下来我们从SQL表中读取DataFrame：

```
// in Scala
val dbDataFrame = spark.read.format( "jdbc").option( "url", url)
  .option( "dbtable", tablename).option( "driver",  driver).load()

# in Python
dbDataFrame = spark.read.format( "jdbc").option( "url", url)\
  .option( "dbtable", tablename).option( "driver",  driver).load()
```

SQLite 需要的配置非常简单（比如，没有用户概念），而其他数据库（比如PostgreSQL）则需要配置更多的参数。现在使用PostgreSQL来执行与之前相同的读取操作：

```
// in Scala
val pgDF = spark.read
  .format( "jdbc")
  .option( "driver",  "org.postgresql.Driver")
  .option( "url",  "jdbc:postgresql://database_server")
  .option( "dbtable",  "schema.tablename")
  .option( "user",  "username").option( "password","my-secret-password").load()
```

```
# in Python
pgDF = spark.read.format( “jdbc")\
  .option( “driver", “org.postgresql.Driver")\
  .option( “url", “jdbc:postgresql://database_server")\
  .option( “dbtable", “schema.tablename")\
  .option( “user", “username").option( “password", “my-secret-password").load()
```

在此创建的DataFrame和以前的并没什么区别：你对它执行查询、转换和连接，这都没有问题。你还会注意到这已经有一个schema模式，这是因为Spark从表本身收集这些数据，并将其类型映射为对应的Spark数据类型。我们来获取去重后的国家字段，以验证是否可以按预期对它执行查询：

```
dbDataFrame.select( “DEST_COUNTRY_NAME").distinct().show(5)
```

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|         Anguilla|
|           Russia|
|         Paraguay|
|          Senegal|
|           Sweden|
+-----------------+
```

太棒了，这表明我们可以查询数据库了！在继续下一个主题之前，有一些有细节问题需要介绍。

查询下推

首先，在创建DataFrame之前，Spark会尽力过滤数据库中的数据。例如，在之前的示例查询中，可以从查询计划中看到它从表中只选择相关的列名：

```
dbDataFrame.select("DEST_COUNTRY_NAME").distinct().explain
```

```
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#8108], functions=[])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#8108, 200)
    +- *HashAggregate(keys=[DEST_COUNTRY_NAME#8108], functions=[])
        +- *Scan JDBCRelation(flight_info) [numPartitions=1] ...
```

在某些查询中，Spark实际上可以做得更好。例如，如果我们在DataFrame上指定一个filter（过滤器），Spark就会将过滤器函数下推到数据库端，我们可以在解释计划（explain plan）中看到PushedFilters的操作。

```
// in Scala
dbDataFrame.filter("DEST_COUNTRY_NAME in ('Anguilla', 'Sweden')").explain
```

```
# in Python
dbDataFrame.filter("DEST_COUNTRY_NAME in ('Anguilla', 'Sweden')").explain()

== Physical Plan ==
*Scan JDBCRel... PushedFilters: [*In(DEST_COUNTRY_NAME, [Anguilla,Sweden])],
...
```

Spark不能把它的所有函数转换为你所使用的SQL数据库中的函数，因此有时要用SQL表达整个查询并将结果作为DataFrame返回。这可能看起来有点复杂，但实际上很简单。只需要以一种特殊的方式指定SQL查询而非指定表名，必须将查询语句包含在圆括号内，然后对其重命名。下面例子将其重命名为同一表名flight_info：

```
// in Scala
val pushdownQuery = """(SELECT DISTINCT(DEST_COUNTRY_NAME) FROM flight_info)
  AS flight_info"""
val dbDataFrame = spark.read.format("jdbc")
  .option("url", url).option("dbtable", pushdownQuery).option("driver",  driver)
  .load()

# in Python
pushdownQuery = """(SELECT DISTINCT(DEST_COUNTRY_NAME) FROM flight_info)
  AS flight_info"""
dbDataFrame = spark.read.format("jdbc")\
  .option("url", url).option("dbtable", pushdownQuery).option("driver",  driver)\
  .load()
```

当查询此表时，实际上查询的是那个SQL语句的查询结果。查看解释计划可以看到这一点，Spark甚至不知道表的实际schema，只知道之前查询产生的结果：

```
dbDataFrame.explain()

== Physical Plan ==
*Scan JDBCRelation(
(SELECT DISTINCT(DEST_COUNTRY_NAME)
  FROM flight_info) as flight_info
) [numPartitions=1] [DEST_COUNTRY_NAME#788] ReadSchema: ...
```

并行读数据库

整本书中，我们介绍了数据划分及其在数据处理中的重要性。Spark有一个底层算法，可以将多个文件放入一个数据分片，或者反过来将一个文件划分到多个数据分片，这取决于文件大小以及文件类型和压缩格式是否允许划分。SQL数据库中也存在与文件一样的分片灵活性，但是你必须手动配置它。正如前面介绍的选项配置，你可以通过指定最大分区数量来限制并行读写的最大数量：

```
// in Scala
val dbDataFrame = spark.read.format("jdbc")
  .option("url", url).option("dbtable", tablename).option("driver", driver)
```

```
  .option( “numPartitions", 10).load()

# in Python
dbDataFrame = spark.read.format( “jdbc")\
  .option( “url", url).option( “dbtable", tablename).option( “driver",  driver)\
  .option( “numPartitions", 10).load()
```

在这种情况下，由于数据不够多，所以仍然作为一个分区。但是，此配置可帮助你确保在读取和写入数据时不会导致数据库过载：

```
dbDataFrame.select("DEST_COUNTRY_NAME").distinct().show()
```

你可以通过在连接中显式地将谓词下推到SQL数据库中执行，这有利于通过指定谓词来控制分区数据的物理存放位置。这个功能很不错，来看一个简单的示例。假设我们仅需要来自两个国家的数据：安圭拉和瑞典，可以对它们进行过滤，并将过滤查询操作下推到数据库，但是在Spark中还可以进一步指定需要过滤国家的数据放入同一个分区。下面通过在创建数据源时指定谓词列表来实现此操作：

```
// in Scala
val props = new java.util.Properties
props.setProperty( “driver", “org.sqlite.JDBC")
val predicates = Array(
   “DEST_COUNTRY_NAME = 'Sweden' OR ORIGIN_COUNTRY_NAME = 'Sweden'",
   “DEST_COUNTRY_NAME = 'Anguilla' OR ORIGIN_COUNTRY_NAME = 'Anguilla'")
spark.read.jdbc(url, tablename, predicates, props).show()
spark.read.jdbc(url, tablename, predicates, props).rdd.getNumPartitions // 2

# in Python
props = { “driver":"org.sqlite.JDBC"}
predicates = [
   “DEST_COUNTRY_NAME = 'Sweden' OR ORIGIN_COUNTRY_NAME = 'Sweden'",
   “DEST_COUNTRY_NAME = 'Anguilla' OR ORIGIN_COUNTRY_NAME = 'Anguilla'"]
spark.read.jdbc(url, tablename, predicates=predicates, properties=props).show()
spark.read.jdbc(url,tablename,predicates=predicates,properties=props)\
  .rdd.getNumPartitions() # 2

+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|           Sweden|      United States|   65|
|    United States|             Sweden|   73|
|         Anguilla|      United States|   21|
|    United States|           Anguilla|   20|
+-----------------+-------------------+-----+
```

如果指定的谓词集合不相交，则会出现大量重复行。以下是一组会产生重复行的谓词集示例：

```
// in Scala
val props = new java.util.Properties
```

```
props.setProperty( “driver”, “org.sqlite.JDBC")
val predicates = Array(
   “DEST_COUNTRY_NAME != 'Sweden' OR ORIGIN_COUNTRY_NAME != 'Sweden'",
   “DEST_COUNTRY_NAME != 'Anguilla' OR ORIGIN_COUNTRY_NAME != 'Anguilla'")
spark.read.jdbc(url, tablename, predicates, props).count() // 510

# in Python
props = { “driver":"org.sqlite.JDBC"}
predicates = [
   “DEST_COUNTRY_NAME != 'Sweden' OR ORIGIN_COUNTRY_NAME != 'Sweden'",
   “DEST_COUNTRY_NAME != 'Anguilla' OR ORIGIN_COUNTRY_NAME != 'Anguilla'"]
spark.read.jdbc(url, tablename, predicates=predicates, properties=props).count()
```

基于滑动窗口的分区

现在介绍如何基于谓词进行分区，在下面示例中，我们将基于数值型的count列进行分区。在这里，我们为第一个分区和最后一个分区分别制定一个最小值和一个最大值，超出该范围的数据将存放到第一个分区或最后一个分区；接下来，我们指定分区总数（这是为了并行操作的）。然后，Spark会并行查询数据库，并返回numPartitions个分区。我们只需修改count列数值的上界和下界，即可将数据相应地存放到各个分区中。这个例子没有像上一个示例一样进行过滤操作：

```
// in Scala
val colName = "count"
val lowerBound = 0L
val upperBound = 348113L // 这是数据集最大行数
val numPartitions = 10

# in Python
colName = "count"
lowerBound = 0L
upperBound = 348113L # 这是数据集最大行数
numPartitions = 10
```

上面代码会根据count列数值从小到大均匀划分10个间隔区间的数据，之后每个区间数据被分配到一个分区：

```
// in Scala
spark.read.jdbc(url,tablename,colName,lowerBound,upperBound,numPartitions,props)
  .count() // 255

# in Python
spark.read.jdbc(url, tablename, column=colName, properties=props,
                lowerBound=lowerBound, upperBound=upperBound,
                numPartitions=numPartitions).count() # 255
```

写入SQL数据库

写入 SQL 数据库和之前一样简单，只需指定 URI 并指定写入模式来写入数据即可。

在下面的示例中，我们指定写入模式为overwrite，因此会覆盖整个表。我们将使用之前定义的 CSV DataFrame 来实现此操作：

```
// in Scala
val newPath = "jdbc:sqlite://tmp/my-sqlite.db"
csvFile.write.mode("overwrite").jdbc(newPath, tablename, props)

# in Python
newPath = "jdbc:sqlite://tmp/my-sqlite.db"
csvFile.write.jdbc(newPath, tablename, mode="overwrite", properties=props)
```

查看结果：

```
// in Scala
spark.read.jdbc(newPath, tablename, props).count() // 255

# in Python
spark.read.jdbc(newPath, tablename, properties=props).count() # 255
```

当然，可以很容易地在该表后面追加新表：

```
// in Scala
csvFile.write.mode("append").jdbc(newPath, tablename, props)

# in Python
csvFile.write.jdbc(newPath, tablename, mode="append", properties=props)
```

注意看计数增加了：

```
// in Scala
spark.read.jdbc(newPath, tablename, props).count() // 765

# in Python
spark.read.jdbc(newPath, tablename, properties=props).count() # 765
```

文本文件

Spark还支持读取纯文本文件，文件中的每一行将被解析为DataFrame 中的一条记录，然后根据你的要求进行转换。假设你需要将某些 Apache 日志文件解析为结构化的格式，或是想解析一些纯文本以进行自然语言处理，这些都需要操作文本文件。由于文本文件能够充分利用原生类型（native type）的灵活性，因此它很适合作为Dataset API的输入。

读文本文件

读文本文件非常简单：只需指定类型为`textFile`即可。With textFile, partitioned

directory names are ignored. To read and write text files according to partitions, you should use text, which respects partitioning on reading and writing：

```
spark.read.textFile( “/data/flight-data/csv/2010-summary.csv")
  .selectExpr( “split(value, ',') as rows").show()

+--------------------+
|                rows|
+--------------------+
|[DEST_COUNTRY_NAM...|
|[United States, R...|
...
|[United States, A...|
|[Saint Vincent an...|
|[Italy, United St...|
+--------------------+
```

写文本文件

当写文本文件时，需确保仅有一个字符串类型的列写出；否则，写操作将失败：

```
csvFile.select("DEST_COUNTRY_NAME").write.text("/tmp/simple-text-file.txt")
```

如果在执行写操作时同时执行某些数据分片操作（接下来将介绍数据分片），则可以写入更多的列。但是这些列将在要写入的文件夹中显示为目录，而不是每个文件中存在多列：

```
// in Scala
csvFile.limit(10).select( “DEST_COUNTRY_NAME", “count")
  .write.partitionBy( “count").text( “/tmp/five-csv-files2.csv")

# in Python

csvFile.limit(10).select( “DEST_COUNTRY_NAME", “count")\
  .write.partitionBy( “count").text( “/tmp/five-csv-files2py.csv")
```

高级I/O概念

我们可以通过在写入之前控制数据分片来控制写入文件的并行度，还可以通过控制数据分桶（bucketing）和数据划分（partitioning）来控制特定的数据布局方式。

可分割的文件类型和压缩

某些文件格式是“可分割的”，因此Spark可以只获取该文件中的满足查询条件的某一个部分，无需读取整个文件，从而提高读取效率。此外，假设你使用的是 Hadoop

分布式文件系统（HDFS），则如果该文件包含多个文件块，分割文件则可进一步优化提高性能。与此同时需要进行压缩管理，并非所有的压缩格式都是可分割的。存储数据的方式对Spark作业稳定运行至关重要，我们推荐采用gzip压缩格式的Parquet文件格式。

并行读数据

多个执行器不能同时读取同一文件，但可以同时读取不同的文件。通常，这意味着当你从包含多个文件的文件夹中读取时，每个文件都将被视为DataFrame的一个分片，并由执行器并行读取，多余的文件会进入读取队列等候。

并行写数据

写数据涉及的文件数量取决于DataFrame的分区数。默认情况是每个数据分片都会有一定的数据写入，这意味着虽然我们指定的是一个“文件”，但实际上它是由一个文件夹中的多个文件组成，每个文件对应着一个数据分片。

以下是代码示例：

```
csvFile.repartition(5).write.format("csv").save("/tmp/multiple.csv")
```

它会生成包含五个文件的文件夹，调用ls命令就可以查看到：

```
ls /tmp/multiple.csv

/tmp/multiple.csv/part-00000-767df509-ec97-4740-8e15-4e173d365a8b.csv
/tmp/multiple.csv/part-00001-767df509-ec97-4740-8e15-4e173d365a8b.csv
/tmp/multiple.csv/part-00002-767df509-ec97-4740-8e15-4e173d365a8b.csv
/tmp/multiple.csv/part-00003-767df509-ec97-4740-8e15-4e173d365a8b.csv
/tmp/multiple.csv/part-00004-767df509-ec97-4740-8e15-4e173d365a8b.csv
```

数据划分

数据划分工具支持你在写入数据时控制存储什么数据以及存储数据的位置。将文件写出时，你可以将列编码为文件夹，这使得你在之后读取时可跳过大量数据，只读入与问题相关的列数据而不必扫描整个数据集。所有基于文件的数据源都支持这些：

```
// in Scala
csvFile.limit(10).write.mode(“overwrite").partitionBy(“DEST_COUNTRY_NAME")
  .save(“/tmp/partitioned-files.parquet")

# in Python
csvFile.limit(10).write.mode(“overwrite").partitionBy(“DEST_COUNTRY_NAME")\
  .save(“/tmp/partitioned-files.parquet")
```

写操作完成后，Parquet“文件”中就会有一个文件夹列表：

```
$ ls /tmp/partitioned-files.parquet

...
DEST_COUNTRY_NAME=Costa Rica/
DEST_COUNTRY_NAME=Egypt/
DEST_COUNTRY_NAME=Equatorial Guinea/
DEST_COUNTRY_NAME=Senegal/
DEST_COUNTRY_NAME=United States/
```

其中每一个都将包含Parquet文件，这些文件包含文件夹名称中谓词为 true的数据：

```
$ ls /tmp/partitioned-files.parquet/DEST_COUNTRY_NAME=Senegal/

part-00000-tid.....parquet
```

读取程序对某表执行操作之前经常执行过滤操作，这时数据划分就是最简单的优化。例如，基于日期来划分数据最常见，因为通常我们只想查看前一周的数据（而不是扫描所有日期数据），这个优化可以极大提升读取程序的速度。

数据分桶

数据分桶是另一种文件组织方法，你可以使用该方法控制写入每个文件的数据。具有相同桶 ID （哈希分桶的ID）的数据将放置到一个物理分区中，这样就可以避免在稍后读取数据时进行shuffle（洗牌）。根据你之后希望如何使用该数据来对数据进行预分区，就可以避免连接或聚合操作时执行代价很大的shuffle操作。

与其根据某列进行数据划分，不如考虑对数据进行分桶，因为某列如果存在很多不同的值，就可能写出一大堆目录。这将创建一定数量的文件，数据也可以按照要求组织起来放置到这些“桶”中：

```
val numberBuckets = 10
val columnToBucketBy =  “count"

csvFile.write.format( “parquet").mode( “overwrite")
  .bucketBy(numberBuckets, columnToBucketBy).saveAsTable( “bucketedFiles")
$ ls /user/hive/warehouse/bucketedfiles/

part-00000-tid-1020575097626332666-8....parquet
part-00000-tid-1020575097626332666-8....parquet
part-00000-tid-1020575097626332666-8....parquet
...
```

数据分桶仅支持Spark管理的表。有关数据分桶和数据划分的更多信息，请参阅2017年Spark Summit的演讲（*https://spark-summit.org/2017/events/why-you-should-care-*

about-data-layout-in-the-filesystem/）。

写入复杂类型

正如我们第 6章中所介绍的，Spark具有多种不同的内部类型。尽管Spark可以使用所有这些类型，但并不是每种数据文件格式都支持这些内部类型。例如，CSV文件不支持复杂类型，而Parquet和ORC文件则支持复杂类型。

管理文件大小

管理文件大小对数据写入不那么重要，但对之后的读取很重要。当你写入大量的小文件时，由于管理所有的这些小文件而产生很大的元数据开销。许多文件系统（如HDFS）都不能很好地处理大量的小文件，而Spark特别不适合处理小文件。你可能听说过“小文件问题”，反之亦然，你也不希望文件太大，因为当你只需要其中几行时，必须读取整个数据块就会使效率低下。

Spark 2.2中引入了一种更自动控制文件大小的新方法。之前介绍了输出文件数量，与写入时数据分片数量以及选取的划分列有关。现在，则可以利用另一个工具来限制输出文件大小，从而可以选出最优的文件大小。可以使用maxRecordsPerFile选项来指定每个文件的最大记录数，这使得你可以通过控制写入每个文件的记录数来控制文件大小。例如，如果你将写程序（writer）的选项设置为df.write.option(“maxRecordsPerFile”，5000），Spark将确保每个文件最多包含5000条记录。

小结

在本章中，我们介绍了在Spark中读和写数据时可用的各种选项，这几乎涵盖了Spark用户所需了解的所有内容。有很多方法可以用于实现自定义的数据源，但由于API正在不断演化发展（为了更好地支持结构化流式处理），所以我们省略了如何实现该操作的介绍。如果你有兴趣了解如何实现自定义数据源，则可以参考Cassandra Connector（*https://github.com/datastax/spark-cassandra-connector*）。

在第10章中，我们将介绍Spark SQL，以及如何与结构化API进行互操作。

第10章

Spark SQL

Spark SQL 可以说是Spark中最重要和最强大的工具之一。本章介绍Spark SQL 中的一些核心概念，但本章不会重新定义ANSI-SQL规范或枚举每种类型的 SQL 表达式。如果你阅读本书的其他任何部分，就会发现在出现DataFrame 代码的地方就有 SQL 代码，以使代码示例能够更容易地相互参照，附录和参考资料部分也提供了其他例子。

简而言之，使用Spark SQL，你可以对存储到数据库中的视图或表进行SQL查询，还可以使用系统函数或用户定义函数来分析查询计划以优化其工作负载。这直接集成到DataFrame和Dataset API中。正如前几章中看到的那样，你可以用SQL和DataFrame表示数据操作，它们都会编译成相同的低级代码。

什么是 SQL？

结构化查询语言（Structured Query Language，SQL）是一种表示数据关系操作的特定领域语言。SQL广泛应用在关系数据库中，许多“NoSQL”数据库也支持类SQL语言以使其更便于使用。SQL 无处不在，即使有些技术专家预言它会落伍，但它依然是许多企业赖以生存的数据工具。Spark实现了 ANSI SQL：2003（*https://en.wikipedia.org/wiki/SQL:2003*）标准的子集，此 SQL 标准是在大多数 SQL 数据库中都支持的，这种支持意味着Spark能够运行各种流行的TPC-DS基准测试（*http://www.tpc.org/default.asp*）。

大数据和SQL：Apache Hive

在Spark流行之前，Hive是支持SQL的主流大数据处理工具。Hive最初是由 Facebook

开发，曾经是支持大数据SQL操作的一个非常流行的工具。它在许多方面帮助将Hadoop 推广到不同的行业，因为分析师可以运行 SQL 查询命令来实现他们的操作。尽管Spark最初是作为一个基于弹性分布式数据集（RDDs）的通用处理引擎开发的，但现在大量用户都在使用Spark SQL。

大数据和SQL：Spark SQL

Spark 2.0 发布了一个支持Hive操作的超集，并提供了一个能够同时支持ANSI-SQL和HiveQL的原生SQL解析器。Spark SQL和DataFrame的互操作性，使得Spark SQL成为各大公司强有力的工具。2016年末，发布Hive的Facebook公司宣布已经开始运行Spark工作负载，并获得巨大的好处。用Facebook博客作者的话来说：

> 我们使用Spark以取代Hive流水线，这将数以百计的Hive作业转化成一个Spark作业。通过一系列对性能和可靠性的改进，我们将Spark拓展到实际生产中的实体排序任务中……与旧的基于Hive的流水线相比，基于Spark的流水线有了显著的性能改进 (4.5~6倍的CPU、3~4倍的资源预留和大约 5倍的延迟)，并且它已经在实际应用中运行了几个月。

Spark SQL在以下关键方面具有强大的能力：SQL 分析人员可通过Thrift Server或者Spark的SQL接口利用Spark的计算能力，数据工程师或者科学家可以在任何数据流中使用Spark SQL。Spark SQL这个统一的API功能强大，允许使用SQL提取数据，并将数据转化成DataFrame进行处理，可以把数据交由Spark MLlib的大型机器学习算法处理，还可以将数据写到另一个数据源中。

Spark SQL 的目的是作为一个在线分析处理（OLAP）数据库，而不是在线事务处理（OLTP）数据库。这意味着Spark SQL现在还不适合执行对低延迟要求极高的查询，但是未来，Spark SQL将会支持这一点。

Spark与Hive的关系

Spark SQL与Hive的联系很紧密，因为Spark SQL可以与Hive metastores连接。Hive metastore维护了Hive跨会话数据表的信息，使用Spark SQL可以连接到Hive metastore访问表的元数据，这可以在访问信息的时候减少文件列表操作带来的开销。对于从传统Hadoop环境转而使用Spark环境运行工作负载的用户来说，这很受欢迎。

Hive metastore

要连接到Hive metastore，你需要设置几个属性。首先，你需要设置 Metastore 版本(`spark.SQL.hive.metastore.version`)，它对应于你要访问的Hive metastore。默认情况下的版本号为1.2.1。如果要更改HiveMetastoreClient的初始化方式，则还需要设置`spark.SQL.hive.metastore.jars`。Spark使用默认版本，但也可以通过设置Java 虚拟机 (JVM) 来指定Maven repositories或classpath。此外，你可能需要提供适当的类前缀，以便与存储Hive metastore 的不同数据库进行通信。你要将这些设置为“Spark”和“Hive”共享的共享前缀(`spark.SQL.hive.metastore.sharedPrefixes`)。

如果要连接到自己的 metastore, 则要查询该文档以了解相关的更新和信息。

如何运行Spark SQL 查询

Spark提供了以下几个接口来执行 SQL 查询。

Spark SQL CLI

Spark SQL CLI 是一个方便的工具, 你可以在本地模式的命令行中实现基本的Spark SQL 查询。请注意, Spark SQL CLI 无法与Thrift JDBC 服务器通信。要启动Spark SQL CLI, 请在Spark目录中运行以下内容：

```
./bin/spark-sql
```

通过将*hive-site.xml*，*core-site.xml*，*hdfs-site.xml*文件放入到*conf/*中来配置Hive。可以运行`./bin/Spark-SQL -help`，来查看所有的可选选项的完整列表。

Spark的可编程SQL接口

除了启用服务器之外，还可以通过任何Spark支持语言的API执行SQL。可以通过`SparkSession`对象上的`sql`方法来实现，这将返回一个DataFrame。例如，在 Python 或 Scala 中，我们可以运行以下内容：

```
spark.sql("SELECT 1 + 1").show()
```

命令`spark.sql ("SELECT 1 + 1")`返回一个 DataFrame，可以被后续处理，就像其他的转换操作一样, 它不会被立即执行而是惰性执行。这是一个非常强大的接口, 因为有一些转换操作通过SQL 代码表达要比通过DataFrame表达要简单得多。

通过将多行字符串传入sql函数中，可以很简单地表示多行查询。例如，你可以在Python 或 Scala 中执行以下代码：

```
spark.sql( """SELECT user_id, department, first_name FROM professors
  WHERE department IN
    (SELECT name FROM department WHERE created_date >= '2016-01-01')""")
```

更强大的是，你可以根据需要在SQL和DataFrame 之间实现完全的互操作。例如，你可以创建一个 DataFrame，使用 SQL 操作它，然后再次将其作为 DataFrame 进行操作。这是一个强大的抽象，你可能会发现自己经常这么用：

```
// in Scala
spark.read.json("/data/flight-data/json/2015-summary.json")
  .createOrReplaceTempView("some_sql_view") //  DataFrame转换为SQL

spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count)
FROM some_sql_view GROUP BY DEST_COUNTRY_NAME
""")
  .where("DEST_COUNTRY_NAME like 'S%'").where("`sum(count)` > 10")
  .count() //SQL转换为DataFrame
# in Python
spark.read.json("/data/flight-data/json/2015-summary.json")\
  .createOrReplaceTempView("some_sql_view") #  DataFrame转换为SQL

spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count)
FROM some_sql_view GROUP BY DEST_COUNTRY_NAME
""")\
  .where("DEST_COUNTRY_NAME like 'S%'").where("`sum(count)` > 10")\
  .count() # SQL转换为DataFrame
```

SparkSQL Thrift JDBC/ODBC服务器

Spark提供了一个 Java 数据库连接 (JDBC) 接口，通过它你或远程程序可以连接到Spark驱动器，以便执行Spark SQL 查询，比如一个业务分析人员将商业智能软件 (如Tableau) 连接到Spark。此处实现的Thrift JDBC/ ODBC 服务器对应于Hive 1.2.1 中的HiveServer2，可以使用带有Spark或Hive 1.2.1 的beeline脚本来测试 JDBC 服务器。

要启动 JDBC/ODBC 服务器，请在Spark目录下运行以下内容：

```
./sbin/start-thriftserver.sh
```

此脚本支持全部的bin/Spark-submit命令行选项。要查看配置此Thrift服务器的所有可用选项，请运行./sbin/start-thriftserver.sh --help。默认情况下，服务器监听localhost:10000。可以通过更改环境变量或系统属性来更新该监听地址和端口。

对于环境变量配置，请使用以下方法：

```
export HIVE_SERVER2_THRIFT_PORT=<listening-port>
export HIVE_SERVER2_THRIFT_BIND_HOST=<listening-host>
./sbin/start-thriftserver.sh \
  --master <master-uri> \
  ...
```

对于系统属性，可以参考下面：

```
./sbin/start-thriftserver.sh \
  --hiveconf hive.server2.thrift.port=<listening-port> \
  --hiveconf hive.server2.thrift.bind.host=<listening-host> \
  --master <master-uri>
  ...
```

然后，可以通过运行以下命令来测试此连接：

```
./bin/beeline
beeline> !connect jdbc:hive2://localhost:10000
```

beeline将询问你的用户名和密码。在非安全模式下，只需在你的计算机上输入用户名和一个空白密码即可。对于安全模式，请按照beeline文档中给出的说明进行操作。

Catalog

Spark SQL 中最高级别的抽象是Catalog。Catalog是一个抽象，用于存储用户数据中的元数据以及其他有用的东西，如数据库，数据表，函数和视图。它在`org.apache.spark.SQL.catalog.Catalog`包中，它包含许多有用的函数，用于执行诸如列举表、数据库和函数之类的操作，我们将很快介绍到这些内容。对于用户来说它具有自解释性，因此我们将省略代码示例，它实际上只是Spark SQL的另一个编程接口。本章仅介绍能够执行的 SQL，因此如果你使用的该编程接口，请记住你需要将所有内容放在`spark.sql`函数中以执行相关代码。

数据表

要使用Spark SQL来执行任何操作之前，首先需要定义数据表。数据表在逻辑上等同于DataFrame，因为它们都是承载数据的数据结构。我们可以执行表连接操作，执行数据表过滤操作，在数据表上执行聚合操作等各种在前几章中接触过的不同操作。数据表和DataFrame的核心区别在于：DataFrame是在编程语言范围内定义的，而数据表是在数据库中定义的。在创建表时（假定你从未更改过数据库），这个数据表将属于默认数据库。在本章后面，我们将主要在数据库环境下讨论。

需要注意的一个重要问题是，在Spark 2.X 中，数据表始终是实际包含数据的，没有类似视图的临时表概念，只有视图不包含数据。这一点很重要，因为如果你要删除一个表，那么可能会导致丢失数据。

Spark托管表

托管表（managed table）和非托管表（unmanaged table）是很重要的概念。表存储两类重要的信息，表中的数据以及关于表的数据即元数据，Spark既可以管理一组文件的元数据也可以管理实际数据。当定义磁盘上的若干文件为一个数据表时，这个就是非托管表；在 DataFrame 上使用`saveAsTable`函数来创建一个数据表时，就是创建了一个托管表，Spark将跟踪托管表的所有相关信息。

在DataFrame上使用`saveAsTable`函数将读取你的表并将其写到一个新的位置（以Spark格式），你可以看到这也体现在新的解释计划中。在解释计划中，你还会注意到这将写入到默认的Hive仓库位置。可以通过配置`spark.SQL.warehouse.dir`为创建 SparkSession 时所选择的目录。默认情况下，Spark将此设置为`/user/hive/warehouse`。

Spark也有数据库，我们将在本章后面讨论。需要提前说明的一点是，你可以在某个其他数据库系统中执行查询命令*`show tables IN databaseName`*，来查看该数据库中的表，其中databaseName表示要查询的数据库的名称。

如果在新的集群或本地模式下运行，则不会返回结果。

创建表

你可以从多种数据源创建表。Spark支持在SQL中重用整个Data Source API，这意味着你不需要首先定义一个表再加载数据，Spark允许你从某数据源直接创建表，从文件中读取数据时，你甚至可以指定各种复杂的选项。

例如，下面是一个简单的表创建方法来查看我们在前几章中使用的航班数据：

```
CREATE TABLE flights (
  DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count LONG)
USING JSON OPTIONS (path '/data/flight-data/json/2015-summary.json')
```

USING和STORED AS

前几个例子中的USING语法规范都具有重要意义。如果未指定格式，则Spark将默认为Hive SerDe配置，但是Hive SerDes 比Spark的本机序列化要慢得多。Hive用户还可以使用STORED AS语法来指定这是一个Hive表。

还可以向表中的某些列添加注释，这样可以帮助其他开发人员了解表中的数据：

```
CREATE TABLE flights_csv (
  DEST_COUNTRY_NAME STRING,
  ORIGIN_COUNTRY_NAME STRING COMMENT  “remember, the US will be most prevalent",
  count LONG)
USING csv OPTIONS (header true, path '/data/flight-data/csv/2015-summary.csv')
```

你也可以从查询结果创建表：

```
CREATE TABLE flights_from_select USING parquet AS SELECT * FROM flights
```

另外，只有表不存在时，才能创建该表：

在此示例中，我们正在创建一个与Hive兼容的表，因为我们没有通过USING显式地指定格式。我们还可以执行以下操作：

```
CREATE TABLE IF NOT EXISTS flights_from_select
  AS SELECT * FROM flights
```

最后，就像第9章中指出的那样，你可以通过写出已分区的数据集来控制数据布局：

```
CREATE TABLE partitioned_flights USING parquet PARTITIONED BY (DEST_COUNTRY_NAME)
AS SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count FROM flights LIMIT 5
```

这些表可以在整个Spark会话中使用，而临时表不存在Spark中，所以必须创建临时的视图，视图将在本章介绍。

创建外部表

正如本章开头提到的，Hive是首批出现的面向大数据的SQL 系统，而Spark SQL 与Hive SQL (HiveQL) 完全兼容。你可能会遇到的一种情况是，将旧的Hive语句端口移植到Spark SQL中。幸运的是，你可以在大多数情况下直接将Hive语句复制并粘贴到Spark SQL 中。例如，在下面的示例中，我们创建一个非托管表，Spark将管理表的元数据，但是数据文件不是由Spark管理。你可以使用CREATE EXTERNAL TABLE语句来创建此表。

你可以通过运行以下命令查看已定义的文件：

```
CREATE EXTERNAL TABLE hive_flights (
  DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count LONG)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION '/data/flight-data-hive/'
```

还可以从 select 子句创建外部表：

```
CREATE EXTERNAL TABLE hive_flights_2
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/flight-data-hive/' AS SELECT * FROM flights
```

插入表

插入操作遵循标准 SQL 语法：

```
INSERT INTO flights_from_select
  SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count FROM flights LIMIT 20
```

如果想要只写入某个分区，可以选择提供分区方案。请注意，写操作也将遵循分区模式（可能导致上述查询运行相当缓慢），它将其他文件只添加到最后的分区中：

```
INSERT INTO partitioned_flights
  PARTITION (DEST_COUNTRY_NAME="UNITED STATES")
  SELECT count, ORIGIN_COUNTRY_NAME FROM flights
  WHERE DEST_COUNTRY_NAME='UNITED STATES' LIMIT 12
```

描述表的元数据

之前我们看到，你可以在创建表时添加注释。通过描述数据表的元数据来显示相关注释：

```
DESCRIBE TABLE flights_csv
```

你还可以使用以下方法查看数据的分区方案 (请注意，这仅适用于已分区的表)：

```
SHOW PARTITIONS partitioned_flights
```

刷新表元数据

维护表的元数据确保从最新的数据集读取数据。有两个命令可刷新表的元数据。REFRESH TABLE用来刷新与表关联的所有缓存项（实质上是文件）。如果之前缓存了该表，则在下次扫描时会惰性缓存它：

```
REFRESH table partitioned_flights
```

另一个相关命令为REPAIR TABLE，它刷新该表在catalog中维护的分区。此命令重点是收集新的分区信息。例如，可以手动写出新分区，并相应地修复表：

```
MSCK REPAIR TABLE partitioned_flights
```

删除表

不能删除表： 只能“drop”它们，可以使用DROP关键字。如果要drop托管表（例如flights_csv），则数据和表定义都将被删除：

```
DROP TABLE flights_csv;
```

drop操作会删除表中的信息，因此在执行此操作时需要非常小心。

如果尝试drop不存在的表，则会报错。若drop已存在的表，请使用DROP TABLE IF EXISTS。

```
DROP TABLE IF EXISTS flights_csv;
```

这将删除表中的数据，因此在执行此操作时应格外小心。

删除非托管表

如果要删除非托管表 (例如，hive_flights)，则不会删除数据，但你将无法再按表名引用此数据。

缓存表

就像 DataFrame 一样，你可以缓存和不缓存表。只需使用以下语法指定要缓存的表：

```
CACHE TABLE flights
```

或者不缓存它们：

```
UNCACHE TABLE FLIGHTS
```

视图

在创建了一个表后，就可以定义视图了。定义视图即指定基于现有表的一组转换操作，基本上只是保存查询计划，这可以方便地组织或重用查询逻辑。Spark有几种不同的视图概念，视图可以是全局的、针对某个数据库的或针对每个会话的。

创建视图

对于终端用户，视图显示为表，但是数据并没有重写到新位置，它们只是在查询时对源数据执行转换操作。可能是filter，select，或者可能是更大的GROUP BY或者ROLLUP。例如，在下面的示例中，我们创建一个目的地为United States的视图，以便仅查看那些特定目的地航班：

```
CREATE VIEW just_usa_view AS
  SELECT * FROM flights WHERE dest_country_name = 'United States'
```

与表类似，你可以创建仅在当前会话期间可用，且未注册到数据库的临时视图：

```
CREATE TEMP VIEW just_usa_view_temp AS
  SELECT * FROM flights WHERE dest_country_name = 'United States'
```

或者也可以是全局临时视图，全局 temp 视图与具体database无关，在整个Spark应用程序中都可查看，但在会话结束时会删除它们：

```
CREATE GLOBAL TEMP VIEW just_usa_global_view_temp AS
  SELECT * FROM flights WHERE dest_country_name = 'United States'

SHOW TABLES
```

你还可以使用下面示例中显式的关键字指定你是否要覆盖视图（如果已存在），可以覆盖临时视图和常规视图：

```
CREATE OR REPLACE TEMP VIEW just_usa_view_temp AS
  SELECT * FROM flights WHERE dest_country_name = 'United States'
```

现在，你可以像查询数据表一样查询此视图：

```
SELECT * FROM just_usa_view_temp
```

视图实际上是一种转换，Spark只会在查询时执行它。 这意味着它只会在你实际查询表之后（而不是更早）应用该过滤器。 实际上，视图等同于从现有DataFrame创建新的DataFrame。

事实上，你可以通过比较Spark DataFrame 和Spark SQL 生成的查询计划来看到这一点。在 DataFrame 中，需要用如下语句：

```
val flights = spark.read.format( “json")
  .load( “/data/flight-data/json/2015-summary.json")
val just_usa_df = flights.where( “dest_country_name = 'United States'")
just_usa_df.selectExpr( “*").explain
```

在SQL中，需要使用如下语句（从我们的视图中查询）：

```
EXPLAIN SELECT * FROM just_usa_view
```

或等同于：

```
EXPLAIN SELECT * FROM flights WHERE dest_country_name = 'United States'
```

鉴于此，你应该在DataFrame或SQL上编写逻辑，这对你来说是最舒适和最易维护的。

删除视图

可以按照删除表的方式删除视图，你只需指定要删除的内容是视图而不是表。删除视图和删除表之间的主要区别是，在视图中不删除基础数据，只删除视图定义本身：

```
DROP VIEW IF EXISTS just_usa_view;
```

数据库

数据库是组织数据表的工具。如果没有一个提前定义好的数据库，Spark将使用默认的数据库。在Spark中执行的SQL语句（包括DataFrame命令）都在数据库的上下文中执行。这意味着，如果更改数据库，那么用户定义的表都将保留在先前的数据库中，并且需要以不同的方式进行查询。

这可能让人很迷惑，尤其是当你正在和同事共享连接或者上下文的时候，所以确保恰当设定了数据库。

可以使用下面命令查看所有存在的数据库：

```
SHOW DATABASES
```

创建数据库

创建数据库使用CREATE DATABASE关键字：

```
CREATE DATABASE some_db
```

选择数据库

如果你想要选择特定的数据库以执行查询，请使用USE关键字后跟数据库名称：

```
USE some_db
```

选择数据库之后，所有的查询都会将表名解析为该数据库中的表名。原本正常的查询可能会失败或产生不正确的结果，这很可能是因为你位于其他数据库下：

```
SHOW tables

SELECT * FROM flights -- fails with table/view not found
```

你也可以使用前缀来标识数据库进行查询：

```
SELECT * FROM default.flights
```

你可以通过运行以下命令来查看当前正在使用的数据库：

```
SELECT current_database()
```

也可以切换回默认数据库：

```
USE default;
```

删除数据库

删除数据库同样很容易，只需使用DROP DATABASE关键字：

```
DROP DATABASE IF EXISTS some_db;
```

选择语句

Spark SQL查询支持以下 ANSI SQL要求（此处列出了SELECT表达式）：

```
SELECT [ALL|DISTINCT] named_expression[, named_expression, ...]
    FROM relation[, relation, ...]
    [lateral_view[, lateral_view, ...]]
    [WHERE boolean_expression]
    [aggregation [HAVING boolean_expression]]
```

```
    [ORDER BY sort_expressions]
    [CLUSTER BY expressions]
    [DISTRIBUTE BY expressions]
    [SORT BY sort_expressions]
    [WINDOW named_window[, WINDOW named_window, ...]]
    [LIMIT num_rows]

named_expression:
    : expression [AS alias]

relation:
    | join_relation
    | (table_name|query|relation) [sample] [AS alias]
    : VALUES (expressions)[, (expressions), ...]
          [AS (column_name[, column_name, ...])]

expressions:
    : expression[, expression, ...]

sort_expressions:
    : expression [ASC|DESC][, expression [ASC|DESC], ...]
```

case…when…语句

有时候可能需要在 SQL 查询中依据某条件来替换值，你可以通过使用case...when...then...end类型语句来实现，这实质上相当于程序中的if语句：

```
SELECT
  CASE WHEN DEST_COUNTRY_NAME = 'UNITED STATES' THEN 1
       WHEN DEST_COUNTRY_NAME = 'Egypt' THEN 0
       ELSE -1 END
FROM partitioned_flights
```

高级主题

我们已经定义了数据的存储位置以及如何组织它们的方法，那下面我们介绍如何查询数据。通过SQL 语句可以完成对数据的操作、定义或控制，最常见的情况是数据操作,这也是本书的重点。

复杂类型

标准 SQL中不支持复杂类型，但是支持复杂类型可以提供非常强大的功能，了解如何在 SQL 中恰当地操作它们是非常有必要的。Spark SQL中支持了三种复杂类型：结构体（struct），列表（list）和映射(map)。

结构体

结构体类似映射，它们提供了一种在Spark中创建或查询嵌套数据的方法。 要创建一个结构体，只需要在括号中包含一组列（或表达式）：

```
CREATE VIEW IF NOT EXISTS nested_data AS
  SELECT (DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME) as country, count FROM flights
```

你可以查询此数据的形式：

```
SELECT * FROM nested_data
```

你也可以查询结构中的每一列：

```
SELECT country.DEST_COUNTRY_NAME, count FROM nested_data
```

还可以使用结构体的名字选择结构体所有的子值以及所有的子列。尽管这些并不是真正的子列，但是这种方式提供了更简单的操作方式，我们可以把他们当做列，完成我们想要的操作：

```
SELECT country.*, count FROM nested_data
```

列表

如果你熟悉编程语言中的列表，那么你对Spark SQL中的列表也不会陌生。在Spark SQL中有两种创建列表的方式：collect_list创建一个包含值的列表；collect_set创建一个不含有重复值的列表。这两种函数都是聚合函数，因此只能够在聚合操作中指定：

```
SELECT DEST_COUNTRY_NAME as new_name, collect_list(count) as flight_counts,
  collect_set(ORIGIN_COUNTRY_NAME) as origin_set
FROM flights GROUP BY DEST_COUNTRY_NAME
```

但是，你也可以通过设定值方法来创建数组，如下所示：

```
SELECT DEST_COUNTRY_NAME, ARRAY(1, 2, 3) FROM flights
```

还可以使用类似 Python 的数组查询语法，按位置查询列表：

```
SELECT DEST_COUNTRY_NAME as new_name, collect_list(count)[0] FROM flights GROUP BY
DEST_COUNTRY_NAME
```

还可以执行诸如将数组转换回行的操作，你可以使用explode函数来执行此任务。为了演示，我们创建一个包含聚合结果的新视图：

```
CREATE OR REPLACE TEMP VIEW flights_agg AS
  SELECT DEST_COUNTRY_NAME, collect_list(count) as collected_counts
  FROM flights GROUP BY DEST_COUNTRY_NAME
```

现在，我们将复杂类型数组中的每个值作为结果中的一行。DEST_COUNTRY_NAME将被重复复制到数组中的每个值，执行与原始collect相反的操作，返回一个DataFrame：

```
SELECT explode(collected_counts), DEST_COUNTRY_NAME FROM flights_agg
```

函数

除了复杂类型外，Spark SQL 还提供了多种高级函数，大部分函数可以在 DataFrame 函数参考中找到，当然也应该知道在 SQL 中找到这些函数的方法。若要查看Spark SQL 中的函数列表，可以使用SHOW FUNCTIONS语句：

```
SHOW FUNCTIONS
```

你还可以指定查询系统函数（即Spark内置函数）和用户函数：

```
SHOW SYSTEM FUNCTIONS
```

用户函数是由你或者与你共享Spark环境的用户定义的函数，也是我们前面所介绍的用户自定义函数（本章后面讲介绍如何创建用户函数）：

```
SHOW USER FUNCTIONS
```

在SHOW命令中，可以通过传递带有通配符 (*) 的字符串来实现过滤选择。在这里，我们可以看到以 "s" 开头的所有函数：

```
SHOW FUNCTIONS "s*";
```

也可以包括LIKE关键字：

```
SHOW FUNCTIONS LIKE "collect*";
```

虽然列出函数一定有用，但通常你可能希望了解有关特定函数本身的更多信息。为此，请使用DESCRIBE关键字，它返回特定函数的文档。

用户定义的函数

正如我们在第3章和第4章中看到的，Spark允许用户定义自己的函数并以分布式方式使用它们。你可以像以前一样定义函数，用某种语言编写函数，然后注册该函数：

```
def power3(number: Double): Double = number * number * number
spark.udf.register("power3", power3(_: Double): Double)
```

```
SELECT count, power3(count) FROM flights
```

你还可以通过Hive CREATE TEMPORARY FUNCTION注册函数。

子查询

可以在其他查询中指定子查询，这使得你可以在 SQL 中指定一些复杂的逻辑。在Spark中有两个基本子查询：相关子查询（Correlated Subquery）使用来自查询外的一些信息。不相关子查询（Uncorrelated Subquery）不包括外部的信息。每个查询都可以返回单个值（标量查询Scalar Subquery）或多个值。Spark还包括对谓词子查询（Predicate Subquery）的支持，它允许基于值进行筛选。

不相关谓词子查询

例如，让我们来看看谓词子查询。在此示例中，由两个不相关查询组成。第一个查询只是获取流量最大的前五个国家目的地：

```
SELECT dest_country_name FROM flights
GROUP BY dest_country_name ORDER BY sum(count) DESC LIMIT 5
```

以下结果：

```
+-----------------+
|dest_country_name|
+-----------------+
|    United States|
|           Canada|
|           Mexico|
|   United Kingdom|
|            Japan|
+-----------------+
```

现在，我们将此子查询where子句的过滤器中，并检查出发地国家是否存在于这个列表中：

```
SELECT * FROM flights
WHERE origin_country_name IN (SELECT dest_country_name FROM flights
      GROUP BY dest_country_name ORDER BY sum(count) DESC LIMIT 5)
```

此查询是不相关的，因为它不包含来自查询外部的信息，这是一个你可以自行运行的查询。

相关谓词子查询

相关谓词子查询允许你在内部查询中使用外部作用域的信息。例如，如果你想查看是

否有航班从目的地国家飞回出发地国家，你可以检查是否有将你的目的地国家作为起飞国家，并且将你的起飞地作为目的地国家的航班：

```
SELECT * FROM flights f1
WHERE EXISTS (SELECT 1 FROM flights f2
            WHERE f1.dest_country_name = f2.origin_country_name)
AND EXISTS (SELECT 1 FROM flights f2
            WHERE f2.dest_country_name = f1.origin_country_name)
```

EXISTS仅检查子查询是否有某值，如果有值，则返回真。可以通过将NOT运算符置于它前面来实现相反的检查。这将等同于寻找一个到达目的地的航班，但你将无法返回！

不相关标量查询

使用不相关的标量查询scalar query，可以引入一些以前可能没有的补充信息。例如，如果希望将最大值包含在整个计数数据集中作为其自己的列，则可以执行以下操作：

```
SELECT *, (SELECT max(count) FROM flights) AS maximum FROM flights
```

其他功能

Spark SQL 中有一些无法归到之前章节里的功能，我们都在这一节中介绍。在执行优化或调试 SQL 代码时，这些功能都可能会用到。

配置

我们在表 10-1中列出了几个Spark SQL 应用程序配置。你可以在应用程序初始化或应用程序执行过程中设置它们（就像我们在本书中看到的Shuffle分区）。

表10-1：Spark SQL 配置

Property Name	Default	Meaning
spark.sql.inMemoryColumnarStorage.compressed	true	如果设置为 true，则Spark SQL 会根据数据的统计信息自动为每一列选择压缩编解码器
spark.sql.inMemoryColumnarStorage.batchSize	10000	控制柱状缓存的批处理大小。较大的批处理可以提高内存利用率和压缩能力，但在缓存数据时有OutOfMemoryErrors (OOMs)风险
spark.sql.files.maxPartitionBytes	134217728 (128 MB)	单个分区中的最大字节数

表10-1：Spark SQL 配置（续）

Property Name	Default	Meaning
spark.sql.files.openCostIn Bytes	4194304 (4 MB)	The estimated cost to open a file, measured by the number of bytes that could be scanned in the same time. This is used when putting multiple files into a partition. It is better to overestimate; that way the partitions with small files will be faster than partitions with bigger files (which is scheduled first).
spark.sql.broadcastTimeout	300	广播连接中广播等待时间的超时秒数 (以秒为单位)
spark.sql.autoBroadcastJoin Threshold	10485760 (10 MB)	配置在执行连接时将广播给所有工作节点的表的最大大小 (以字节为单位)。可以通过将此值设置为-1 来禁用广播。请注意, 当前的统计信息仅支持配置单元 Metastore 表。命令分析表计算 STA TISTICS noscan已运行
spark.sql.shuffle.partitions	200	配置在为连接或聚合shuffle数据时要使用的分区数

在 SQL 中设置配置值

我们将在第15章讨论配置，值得一提的是如何从SQL中配置，下面是如何设置shuffle分区的方法：

```
SET spark.sql.shuffle.partitions=20
```

小结

从本章中可以清楚地看出，Spark SQL和DataFrame密切相关, 你应该可以在DataFrame上使用本书中几乎所有的示例，有时候需要一些微小的语法调整。本章介绍了更多与Spark SQL相关的细节，第11 章将重点介绍一个新概念：允许类型安全地进行结构化转换的Dataset。

第11章

Dataset

Dataset是结构化 API 的基本类型。我们已经使用了 DataFrame，它是Row类型的Dataset，在Spark各种语言中都得到了支持。Dataset具有严格的 Java 虚拟机 (JVM) 语言特性，仅与 Scala 和 Java 一起使用。你可以定义Dataset中每一行所包含的对象。在Scala 中就是一个case类对象，它实质上定义了一种模式schema，而在Java中就是Java Bean。经验丰富的用户经常将Dataset称为Spark中的“类型化的API”（Dataset在编译时检查类型，DataFrame在运行时检查类型）。有关更多信息，请参考第4章。

在第4章中， 我们讨论了Spark的类型，例如StringType，BigIntType，StructType等。这些特定于Spark的类型在Spark支持的语言中都有对应的类型，如String，Integer和Double。使用 DataFrame API 时，不需要创建字符串或整数Spark就可以通过操作Row对象来处理数据。如果使用 Scala 或 Java，则所有“DataFrame”实际上都是Row类型的Dataset。为了有效地支持特定领域的对象，需要一个称为“编码器”（Encoder）的特殊概念，编码器将特定领域类型 T 映射为Spark的内部类型。

例如，给定一个类Person具有两个字段，名称(string)和年龄(int)，编码器保证Spark在运行时生成代码以序列化Person对象为二进制结构。使用 DataFrame 或“标准的”结构化 API 时，此二进制结构就是Row类型。当我们要创建自己的特定领域对象时，可以在 Scala 中指定一个case类，或者在 Java 中通过JavaBean，这样我们可以通过Spark以分布式方式操作此对象(不是Row类型的对象)。

当使用Dataset API 时，将Spark Row格式的每一行转换为指定的特定领域类型的对象(case类或 Java 类)。此转换会减慢操作速度，但可以提供更大的灵活性。

你将看到性能的下降，但这与Python 中的用户定义函数 (UDF) 之类的概念有着巨大的不同，因为性能成本并不像切换编程语言那样极端，用户最好能了解这点。

何时使用Dataset

你可能会想，如果在使用Dataset时损失性能，那为什么我们还要使用它们呢？有下面几个主要原因：

- 当你要执行的操作无法使用DataFrame操作表示时。
- 如果需要类型安全，并且愿意牺牲一定性能来实现它。

接下来我们更详细地探讨一下。有些操作不能使用我们在前面章节中看到的结构化 API 来表示，虽然这不是很常见，但你可能有业务逻辑想用特定的函数而非 SQL 或DataFrame来实现，这就要用到Dataset了。此外，因为Dataset API 是类型安全的，对于其类型无效的操作 (例如，两个字符串类型相减) 将在编译时出错，而不是在运行时失败，如果正确性和防御性代码是你更需要考虑的事情，那么牺牲一些性能或许是好的选择。这不能保证不接受格式错误的数据，但可以让你更方便地处理它。

使用Dataset的另一种情况是，在单节点作业和Spark作业之间重用对行的各种转换代码。如果你熟悉 Scala，可能会注意到Spark的 API 包含了Scala Sequence类型，它们以分布式方式运行。事实上，这是Scala 的发明者Martin Odersky在2015年Spark欧洲峰会说的。因此，使用Dataset的一个优点是，如果你将所有数据和转换定义为 case 类，那么在分布式和单机作业中使用它们没什么区别。此外，当你在本地磁盘存储DataFrame时，它们一定是正确的类和类型，这使进一步的操作更容易。

最流行的应用场景可能是先用DataFrame和再用Dataset的情况，这可以手动在性能和类型安全之间进行权衡。这在有些情况时是很有用的，比如当基于DataFrame执行的提取、转换和加载 (ETL) 转换作业之后，想将数据送入驱动器并使用单机库操作时，或者是当需要在Spark SQL 中执行过滤和进一步操作之前，进行每行分析的预处理转换操作的时候。

创建Dataset

创建Dataset有些是手动操作，要求你提前知道和定义数据schema。

Java：编码器

Java 编码器相当简单，只需指定类，然后在需要DataFrame（即Dataset <Row>类型）的时候对该类进行编码：

```
import org.apache.spark.sql.Encoders;

public class Flight implements Serializable{
  String DEST_COUNTRY_NAME;
  String ORIGIN_COUNTRY_NAME;
  Long DEST_COUNTRY_NAME;
}

Dataset<Flight> flights = spark.read
  .parquet( "/data/flight-data/parquet/2010-summary.parquet/")
  .as(Encoders.bean(Flight.class));
```

Scala：Case类

在 Scala 中创建Dataset，要定义 Scala的case类，case类是具有以下特征的类：

- 不可变。
- 通过模式匹配可分解，来获取类属性。
- 允许基于结构的比较而不是基于引用的比较。
- 易于使用和操作。

这些特点对于数据分析很有用，因为case类很容易解析。而最重要的特性应该是case类的不变性，可以通过内容结构而不是引用进行比较。下面是 Scala 文档对它的描述（*http://docs.scala-lang.org/tutorials/tour/case-classes.html*）：

- 不变性使你无需跟踪对象变化和时间和位置。
- “按值比较”允许直接比较实例的值，就像它们是基本类型值（primitive value）一样，这样就避免了混淆类实例基于值比较或是基于引用比较所带来的不确定性。
- 模式匹配简化了分支逻辑，从而会引起更少的 bug 和带来可读性更好的代码。

Spark使用Scala，所以当然也会有这些优点。

创建Dataset，首先为我们的数据集定义一个case class：

```
case class Flight(DEST_COUNTRY_NAME: String,
                  ORIGIN_COUNTRY_NAME: String, count: BigInt)
```

现在，我们定义了case class，它表示数据中的单个记录。更简单地说，它是我们现在有一个包含Flight类型的Dataset，只有模式没有方法。当读数据时，我们会得到一个DataFrame，我们使用as方法将其强制转换为指定的行类型：

```
val flightsDF = spark.read
  .parquet( "/data/flight-data/parquet/2010-summary.parquet/")
val flights = flightsDF.as[Flight]
```

动作操作

Dataset的作用很大，但重要的是要理解，不管是使用Dataset或者DataFrame，像collect，take和count这样的操作的应用：

```
flights.show(2)
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|    1|
|    United States|            Ireland|  264|
+-----------------+-------------------+-----+
```

你还会注意到，当我们实际去访问这些case class时，不需要执行任何类型强制转化，只需指定case class的属性名并返回，返回包括预期值和预期类型：

```
flights.first.DEST_COUNTRY_NAME // 结果是United States
```

转换操作

Dataset上的转换操作与DataFrame上的转换操作相同。你在本章中看到的任何转换操作都适用于Dataset，我们也建议你查看有关聚合操作或连接操作的特定章节。除了那些转换操作之外，Dataset还允许我们指定比 DataFrame转换更复杂的和强类型的转换操作，因为我们可以操作底层的Java 虚拟机 (JVM) 类型。为了说明这个底层对象操作，我们在刚创建的Dataset上执行filter过滤操作。

过滤

看一个简单的例子，创建一个接受Flight为参数的简单函数，它返回一个判断出发地和目的地是否相同的Boolean值。这不是一个 UDF (至少与 Spark SQL 定义 UDF 的方式不同)，而是一个通用函数。

在下面的示例中你将注意到，我们创建一个函数来定义此过滤器（filter）。这与我们迄今为止在书中所做的事情有很大的不同。通过指定函数，我们强制Spark在Dataset中的每一行上验证此函数，这可能非常耗资源。对于简单过滤器，总是首选编写 SQL 表达式，这将大大降低过滤数据的成本，同时仍允许你稍后将其作为Dataset进行操作：

```
def originIsDestination(flight_row: Flight): Boolean = {
  return flight_row.ORIGIN_COUNTRY_NAME == flight_row.DEST_COUNTRY_NAME
}
```

现在，我们可以将此函数传递到过滤器方法中，指定每行应验证此函数是否返回true，并相应地过滤Dataset：

```
flights.filter(flight_row => originIsDestination(flight_row)).first()
```

结果是：

```
Flight = Flight(United States,United States,348113)
```

正如我们前面看到的，这个函数不需要在Spark代码中执行，可以在使用Spark之前在本地机器的数据上测试用。

例如，这个Dataset足够小，我们可以让驱动器处理这些数据 (一个Flight数组)，执行完全相同的过滤操作：

```
flights.collect().filter(flight_row => originIsDestination(flight_row))
```

结果是：

```
Array[Flight] = Array(Flight(United States,United States,348113))
```

我们可以看到，得到了与之前完全相同的答案。

映射

过滤是一个简单的转换操作，但有时需要将一个值映射到另一个值。前面示例函数的输入参数是一个flight，并返回一个布尔值，但是有时候我们可能实际上需要执行一些更复杂的操作，比如提取一个数值，比较一组值，或者类似的操作。

最简单的示例从Dataset的每行中提取一个值，这实际上是在Dataset上执行像select这样的操作。该示例中我们提取目的地：

```
val destinations = flights.map(f => f.DEST_COUNTRY_NAME)
```

注意，我们返回String类型的Dataset，这是因为Spark已经知道这个结果应该返回的 JVM 类型，如果由于某种原因返回结果无效的话，我们也可以从编译时检查中受益。

我们可以在驱动器上collect结果并返回字符串数组：

```
val localDestinations = destinations.take(5)
```

你可能觉得这些不必要，我们可以在 DataFrame 上实现大部分操作。但我们建议你这样做，因为你从中会获得许多好处，例如代码生成，通过用户定义函数可能无法做到，但是可以用更复杂的逐行操作实现。

连接

如前所述，连接操作的用法与DataFrame一样。但是，Dataset还提供了更复杂的方法，即joinWith方法。joinWith大致等同于一个co-group(RDD术语) ，将两个Dataset整合成一个，每一列都表示一个Dataset，并可以相应地进行操作。当你需要在连接操作中维护更多信息，或执行一些更复杂的操作 (如高级映射或筛选) 时，这可能很有用。

让我们创建一个假的flight元数据（metadata）的Dataset来演示joinWith：

```
case class FlightMetadata(count: BigInt, randomData: BigInt)

val flightsMeta = spark.range(500).map(x => (x, scala.util.Random.nextLong))
  .withColumnRenamed("_1", "count").withColumnRenamed("_2", "randomData")
  .as[FlightMetadata]
val flights2 = flights
  .joinWith(flightsMeta, flights.col("count") === flightsMeta.col("count"))
```

注意，我们输出包含一个键值对的Dataset，其中每一行表示一个Flight和Flight元数据（Metadata）。当然，我们可以查询这些Dataset或具有复杂类型的 DataFrame：

```
flights2.selectExpr("_1.DEST_COUNTRY_NAME")
```

也可以像以前一样在驱动器上获取它们：

```
flights2.take(2)

Array[(Flight, FlightMetadata)] = Array((Flight(United States,Romania,1),...
```

当然，“常规的”连接就会很好地运行，尽管在这种情况下你会注意到我们最终会得到一个 DataFrame (从而丢失了 JVM 类型信息)。

```
val flights2 = flights.join(flightsMeta, Seq("count"))
```

我们还可以把该DataFrame转换回Dataset。还需要注意的是，连接 DataFrame 和Dataset也完全没有问题，下面代码会得到相同的结果：

```
val flights2 = flights.join(flightsMeta.toDF(), Seq("count"))
```

分组和聚合

分组（grouping）和聚合（aggregation）基本上和我们在之前的聚合章节中了解的聚合操作差不多，因此groupBy rollup和cube仍可用，但它们返回 DataFrame而不是Dataset(丢失类型信息)：

```
flights.groupBy("DEST_COUNTRY_NAME").count()
```

这通常不会造成太大的损失，但如果你想保持类型信息，也有其他方法可以执行分组和聚合。一个很好的示例是groupByKey方法，它可以按Dataset中的特定键进行分组，并返回具有类型信息的Dataset。但是，此方法接受一个函数而非特定的列名。这使你可以指定更复杂的分组功能，这个函数更类似于以下内容：

```
flights.groupByKey(x => x.DEST_COUNTRY_NAME).count()
```

虽然这提供了灵活性，但需要权衡，因为现在我们引入了 JVM 类型以及无法被Spark优化的函数。这意味着会产生性能差异，当我们检查解释计划时我们可以看到这一点。下面你会看到我们正在高效地追加一个新列到 DataFrame (分组函数的结果)，然后在该列上执行分组：

```
flights.groupByKey(x => x.DEST_COUNTRY_NAME).count().explain
== Physical Plan ==
*HashAggregate(keys=[value#1396], functions=[count(1)])
+- Exchange hashpartitioning(value#1396, 200)
   +- *HashAggregate(keys=[value#1396], functions=[partial_count(1)])
      +- *Project [value#1396]
         +- AppendColumns <function1>, newInstance(class ...
         [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, ...
            +- *FileScan parquet [D...
```

在Dataset上根据某key执行分组后，我们就可以在Key Value Dataset上根据我们定义的函数操作数据，该函数将分组作为原始对象操作：

```
def grpSum(countryName:String, values: Iterator[Flight]) = {
  values.dropWhile(_.count < 5).map(x => (countryName, x))
}
flights.groupByKey(x => x.DEST_COUNTRY_NAME).flatMapGroups(grpSum).show(5)
```

```
+--------+--------------------+
|      _1|                  _2|
+--------+--------------------+
|Anguilla|[Anguilla,United ...|
|Paraguay|[Paraguay,United ...|
|  Russia|[Russia,United St...|
| Senegal|[Senegal,United S...|
|  Sweden|[Sweden,United St...|
+--------+--------------------+
def grpSum2(f:Flight):Integer = {
  1
}
flights.groupByKey(x => x.DEST_COUNTRY_NAME).mapValues(grpSum2).count().take(5)
```

我们甚至可以创建新的操作并定义如何执行reduceGroups聚合：

```
def sum2(left:Flight, right:Flight) = {
  Flight(left.DEST_COUNTRY_NAME, null, left.count + right.count)
}
flights.groupByKey(x => x.DEST_COUNTRY_NAME).reduceGroups((l, r) => sum2(l, r))
  .take(5)
```

很容易看出这是一个比扫描后立即聚合（直接调用groupBy）更耗时的过程，况且它得到是相同的结果：

```
flights.groupBy( “DEST_COUNTRY_NAME").count().explain
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#1308], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#1308, 200)
   +- *HashAggregate(keys=[DEST_COUNTRY_NAME#1308], functions=[partial_count(1)])
      +- *FileScan parquet [DEST_COUNTRY_NAME#1308] Batched: tru...
```

所以，仅在用户定义分组聚合时使用Dataset才有意义，这可能在大数据流水线处理的开始或结束的位置。

小结

本章中，我们介绍了Dataset的基本知识，并提供了一些示例。虽然简短，但本章实际上介绍了基本上所有你需要知道的有关Dataset的内容和使用方法。可以将它视为高级结构化 API 和低级 RDD API 之间的产物，低级RDD将是第12章的主题。

第III部分

低级API

第12章

弹性分布式数据集

本书第二部分中介绍了Spark的结构化API，在几乎所有场景下用户都应该尽量使用这些结构化API。但当这类高级API无法解决遇到的业务或工程问题的时候，就需要使用Spark的低级API，特别是弹性分布式数据集（RDD）、SparkContext和分布式共享变量（例如累加器和广播变量）。本部分将介绍这些低级API和它们的使用方法。

如果你是初学Spark，不适合从该部分开始阅读。建议从第二部分结构化API开始阅读，这样有利于更快地写出Spark应用程序。

什么是低级API？

有两种低级API：一种用于处理分布式数据（RDD），另一种用于分发和处理分布式共享变量（广播变量和累加器）。

何时使用低级 API?

下列三种场景，通常需使用到低级API：

- 当在高级API中找不到所需的功能时，例如要对集群中数据的物理放置进行非常严格的控制时。
- 当需要维护一些使用RDD编写的遗留代码库时。

- 当需要执行一些自定义共享变量操作时（关于共享变量，将在第14章做详细介绍）。

在这些场景中，需要使用到这些低级工具，另外，由于Spark所有工作负载都将编译成这些基本原语，因此学习低级API也有助于我们更好的理解 这些工具。当调用一个DataFrame的转换操作时，实际上等价于一组RDD的转换操作。当你开始调试越来越复杂的工作负载，这种对等价转换操作的理解可以帮你更容易完成任务。

即使你是一个希望充分利用Spark所有功能（包括高级和低级功能）的高级开发人员，仍建议重点关注结构化API。但有时可能会使用到一些低级API中来完成任务，比如，你可能需要借助这些低级API去使用一些遗留代码，或者实现一些自定义分区程序，或在数据流水线的执行过程中更新和跟踪变量的值。为了防止在使用时弄巧成拙，这些工具提供了更细粒度的控制。

如何使用低级API？

SparkContext是低级API函数库的入口，可以通过SparkSession来获取SparkContext，`SparkSession`是用于在Spark集群上执行计算的工具。第15章中将进一步介绍SparkSession，现在仅需知道能通过下面的调用方式访问SparkContext即可：

```
Spark.sparkContext
```

关于RDD

RDD是Spark 1.X系列中主要的API，在2.X系列中仍然可以使用它，但是已不常用。正如在本书前面提到，无论是DataFrame还是Dataset，运行的所有Spark代码都将编译成一个RDD。本书第18章中介绍的Spark UI中还将涉及RDD的job执行。因此，至少需要对RDD是什么以及如何使用RDD有一个基本的了解。

简单来说，RDD是一个只读不可变的且已分块的记录集合，并可以被并行处理。RDD与 DataFrame不同，DataFrame中每个记录即是一个结构化的数据行，各字段已知且schema已知，而 RDD中的记录仅仅是程序员选择的Java、Scala 或 Python 对象。

正因为RDD中每个记录仅仅是一个Java或 Python 对象，因此能完全控制RDD，即能以任何格式在这些对象中存储任何内容。这使你具有很大的控制权，同时也带来一些潜在问题。比如，值之间的每个操作和交互都必须手动定义，也就是说，无论实现什么任务，都必须从底层开发。另外，因为Spark不像对结构化API那样清楚地理解记录的内部结构，所以往往需要用户自己写优化代码。比如，Spark的结构化API会自动以

优化后的二进制压缩格式存储数据，而在使用低级API时，为了实现同样的空间效率和性能，你就需要在对象内部实现这种压缩格式，以及针对该格式进行计算的所有低级操作。同样，像重排过滤和聚合等这类SparkSQL中自动化的优化操作，也需要你手动实现。因此，强烈建议尽可能使用Spark结构化API。

RDD API与Dataset类似，这点在本书的前面章节中就能看出来，所不同的是RDD不在结构化数据引擎上进行存储或处理。然而，在 RDD和Dataset之间来回转换的代价很小，因此可以同时使用两种API来取长补短。在本书的这一部分，将展示如何实现这一点。

RDD类型

看一下Spark的 API 文档，就会发现其中有很多 RDD 子类，大部分是 DataFrame API 优化物理执行计划时用到的内部表示。但是，作为用户，一般只会创建两种类型的RDD：“通用”型RDD或提供附加函数的key-value RDD，例如基于key的聚合函数。这两种类型的RDD是最常使用的，两者都是表示对象的集合，但是key-value RDD支持特殊操作并支持按key的自定义数据分片。两者都只是表示对象的集合，但key-value RDD具有特殊的操作以及按key自定义分区的概念。

现在来正式定义RDD。每个RDD具有以下五个主要内部属性：

- 数据分片（Partition）列表。
- 作用在每个数据分片的计算函数。
- 描述与其他RDD的依赖关系列表。
- (可选)为key-value RDD配置的Partitioner(分片方法，如hash分片）。
- (可选)优先位置列表，根据数据的本地特性，指定了每个Partition分片的处理位置偏好（例如，对于一个HDFS文件来说，这个列表就是每个文件块所在的节点）。

使用RDD的一个主要原因是你想定制Partitioner，如果正确地使用你自定义的Partitioner，则能在性能和稳定性上得到有效提升。在第13章中介绍键值对RDD时，将深入介绍Partitioner。

这些属性决定了Spark的所有调度和执行用户程序的能力，不同RDD都各自实现了上述的每个属性，并允许你定义新的数据源。

RDD遵循与前面章节完全相同的Spark编程范例。它支持两种类型（算子）操作：惰性执行的转换操作和立即执行的动作操作，都是以分布式的方式来处理数据。它们的工作方式与DataFrame和Dataset上的转换操作和动作操作相同，但是RDD中没有“行”的概念，RDD的单个记录只是原始的Java/Scala/Python对象，因此你必须手动处理这些数据，而不能使用结构化API中的函数库。

RDD API支持Python，Scala和Java。对于Scala和Java而言，其性能基本是相同的，主要的性能开销花费在处理原始对象上。但对于Python来说，使用RDD会极大地影响性能，运行Python RDD等同于逐行运行用户定义的Python函数（UDF）。正如在第6章中看到的一样，我们需要将数据序列化到Python进程中，使用Python进行操作，然后再将其序列化到Java虚拟机（JVM）中，这会导致Python RDD操作产生极大开销。尽管过去有人在实际生产中运行这种代码，但我们建议在Python中使用结构化API，除非有些情况必须使用RDD。

何时使用RDD？

一般来说，除非有非常非常明确的理由，否则不要手动创建RDD。它们是很低级的API，虽然它提供了大量的功能，但同时缺少结构化API中可用的许多优化。在绝大多数情况下，DataFrame比RDD更高效、更稳定并且具有更强的表达能力。

当你需要对数据的物理分布进行细粒度控制（自定义数据分区）时，可能才需要使用RDD。

Dataset和使用Case Class转换的RDD

在网上有一个有趣的问题：使用Case Class转换的RDD和Dataset的区别是什么？不同之处在于，虽然它们可以执行同样的功能，但是Dataset可以利用结构化API提供的丰富功能和优化，无需选择是在JVM类型还是Spark类型上进行操作。你可以采用其中最简单或最灵活的方式，这样就有了两全其美之法。

创建RDD

前面已经介绍了一些RDD的关键属性，接下来通过实际应用来更好地理解如何使用它们。

在DataFrame，Dataset和RDD之间进行交互操作

获取RDD的最简单方法之一是从现有的DataFrame或Dataset转换，将它们转换成RDD很简单：只要调用这些数据类型的rdd方法即可。如果从Dataset[T]转换成RDD，会返回合适的本机类型T（这仅适用于Scala和Java）：

```
// in Scala: 将 Dataset[Long] 类型转换为 RDD[Long]类型
spark.range(500).rdd
```

因为Python中只有DataFrame，并没有Dataset，所以会得到一个Row类型的RDD：

```
# in Python
spark.range(10).rdd
```

要对这些数据进行操作，则需将此Row类型的对象转换为正确的数据类型或从中提取出值（如下面例子所示）。下面就是一个Row类型的RDD了：

```
// in Scala
spark.range(10).toDF().rdd.map(rowObject => rowObject.getLong(0))

# in Python
spark.range(10).toDF("id").rdd.map(lambda row: row[0])
```

你可以使用相同的方法从RDD中创建DataFrame或Dataset，操作方法就是调用RDD的toDF方法：

```
// in Scala
spark.range(10).rdd.toDF()

# in Python
spark.range(10).rdd.toDF()
```

上述命令创建了一个Row类型的RDD，这个Row类型是Spark在结构化API中用于表示数据的内部Catalyst格式。在合适的用例中，此功能可以实现结构化API和低级API之间的跳转（在第13章将介绍这一点）。

RDD API与第11章中的Dataset API非常相似，RDD是Dataset的低级表示，但是它们并没有像结构化API那样方便的函数和接口。

从本地集合中创建RDD

要从集合中创建RDD，需要调用SparkContext（在SparkSession中）中的parallelize方法，该方法会将位于单个节点的数据集合转换成一个并行集合。在创建该并行集合时，还可以显式指定该并行集合的分片数量。下面的示例中创建了两个数据分片：

```
// in Scala
val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"
  .split(" ")
val words = spark.sparkContext.parallelize(myCollection, 2)

# in Python
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
  .split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
```

另一个功能是，可以命名此RDD以方便将其展示在Spark UI上：

```
// in Scala
words.setName("myWords")
words.name // 结果是myWords

# in Python
words.setName("myWords")
words.name() # 结果是myWords
```

从数据源创建

尽管从数据源或文本文件中都能创建RDD，但通常最好使用数据源API来创建。RDD中没有像DataFrame中“Data Source API”这样的概念，使用RDD中主要是用来定义它们的依赖结构和数据分片方案。第9章中介绍到的Data Source API是读取数据最好的方式，也就是说，你也可以使用sparkContext将数据读取为RDD。以下示例实现逐行读取一个文本文件：

```
spark.sparkContext.textFile("/some/path/withTextFiles")
```

这将创建一个RDD，RDD中的每个记录都代表该文本文件中的一行。或者可以按照下面方法把每个文本文件读取成一条记录。这里用例中的每个文件，可能里面包含了一个大型JSON对象或文档：

```
spark.sparkContext.wholeTextFiles("/some/path/withTextFiles")
```

在该RDD中，文件的名称是第一个对象，文本文件的内容则是第二个字符串对象。

操作RDD

操作RDD的方式与操作DataFrame的方式非常相似，如前所述，核心区别在于RDD操作的是原始Java或Scala对象而非Spark类型，以及还缺乏一些可帮助你简化计算的“辅助”方法或函数。因此，你必须自己定义每个过滤器filter、映射函数map functions、聚合aggregation，以及其他想被作为函数使用的任何操作。

为了演示一些数据操作，我们将使用之前创建的简单的RDD(words)来详细说明细节。

转换操作

大多数情况下，转换操作与结构化API中的转换操作在功能上相同。就像使用DataFrame和Dataset一样，需要在RDD上指定转换操作来创建一个新的RDD，这也将导致新的RDD依赖于原有的RDD。

distinct

在RDD上调用distinct方法用于删除RDD中的重复项：

```
words.distinct().count()
```

上述命令输出结果是10。

filter

过滤器filter等同于创建一个类似SQL中的where子句。你可以通过RDD中的记录来查看谓词函数匹配，该匹配的函数被用来作为过滤函数时，只需要返回一个布尔类型值，而其输入应该是用户给定的行。如下示例中，我们将过滤RDD中数据，仅保留以字母“S”开头的单词：

```
// in Scala
def startsWithS(individual:String) = {
  individual.startsWith("S")
}

# in Python
def startsWithS(individual):
  return individual.startswith("S")
```

定义了该函数后，下面开始过滤数据。如果你阅读了第11章，会觉得下面操作很熟悉，其中仅使用了一个在RDD中按记录操作的函数。下面这个函数定义为逐个处理RDD中的每条记录：

```
// in Scala
words.filter(word => startsWithS(word)).collect()
```

这行代码将返回Spark和Simple（以“S”开头的单词），可以看到，与Dataset API一

样，它返回的是本地数据类型。这是因为我们从不将数据强制转换为Row类型，也不在收集数据后做数据转换操作。

map

第11章中介绍过map操作。指定一个函数，将给定数据集中的记录一条一条地输入该函数处理以得到你期望的结果。在下面例子中，我们将当前单词映射为一个三元组记录，包括该原单词，该单词的首字母，以及该单词是否以“S”开头的布尔值。

请注意，在下面例子中，我们使用相关的lambda语法以内联方式定义了我们的函数：

```
// in Scala
val words2 = words.map(word => (word, word(0), word.startsWith("S")))

# in Python
words2 = words.map(lambda word: (word, word[0], word.startswith("S")))
```

你可以继续根据words2结果中的布尔值（即第三项）来进行过滤：

```
// in Scala
words2.filter(record => record._3).take(5)

# in Python
words2.filter(lambda record: record[2]).take(5)
```

这将返回由“Spark”，“S”和“True”组成的三元组，以及由“Simple”，“S”和“True”组成的三元组。

flatMap

flatMap是对上面的map函数的一个简单扩展。在映射操作中，有时每个当前行会映射为多行，例如，将一组单词由flatMap映射为一组字符。由于每个单词有多个字符，因此需要使用flatMap来展开它。flatMap映射要求输出是一个可展开的迭代器：

```
// in Scala
words.flatMap(word => word.toSeq).take(5)

# in Python
words.flatMap(lambda word: list(word)).take(5)
```

这将输出S， P， A， R，K。

排序

要对RDD进行排序，必须使用sortBy方法，同其他RDD操作一样，可以通过一个指定

函数从RDD对象中读取值，然后对该值进行排序。例如，以下示例按字长从长到短排序：

```
// in Scala
words.sortBy(word => word.length() * -1).take(2)

# in Python
words.sortBy(lambda word: len(word) * -1).take(2)
```

随机分割

通过randomSplit方法可以将一个RDD随机切分成若干个RDD，这些RDD组成一个RDD的Array返回。randomSplit方法有两个参数：一个是包含权重的Array（randomSplit方法根据weight权重值进行切分，权重越高划分得到的元素较多的几率就越大。数组的长度即为划分成RDD的数量，注意，weight数组内weight权重值的加和应为1），第二个是随机数种子（seed，可选参数）：

```
// in Scala
val fiftyFiftySplit = words.randomSplit(Array[Double](0.5, 0.5))

# in Python
fiftyFiftySplit = words.randomSplit([0.5, 0.5])
```

返回一个可以被操作的RDD数组。

动作操作

就像在DataFrame和Dataset中的动作操作一样，KDD的动作操作用来触发具体的转换操作。动作操作或者将数据收集到驱动器，或者将其写到外部数据源。

reduce

reduce方法指定一个函数将RDD中的任何类型的值“规约”为一个值。举个例子：给定一组数字，可以用reduce方法计算它们的和。reduce方法要指定一个函数来接收两个参数，然后对它们求和返回一个新值，新值再作为其中一个参数继续传递给该函数，另一个参数为下一个数字，以此重复多次，直到最后一个元素，最终返回一个值，即它们的和。如果有函数式编程方面的经验，应该比较熟悉这个概念：

```
// in Scala
spark.sparkContext.parallelize(1 to 20).reduce(_ + _) // 210

# in Python
spark.sparkContext.parallelize(range(1, 21)).reduce(lambda x, y: x + y) # 210
```

你也可以用它来获取单词集中最长的单词（上面刚刚介绍的例子），关键在于要定义正确的函数：

```
// in Scala
def wordLengthReducer(leftWord:String, rightWord:String): String = {
  if (leftWord.length > rightWord.length)
    return leftWord
  else
    return rightWord
}

words.reduce(wordLengthReducer)

# in Python
def wordLengthReducer(leftWord, rightWord):
  if len(leftWord) > len(rightWord):
    return leftWord
  else:
    return rightWord

words.reduce(wordLengthReducer)
```

上面的reducer是一个很好的例子，因为它会输出两者中的一个。在数据分片上的reduce操作是不确定的，对于单词“definitive”或“processing”（长度均为10）都可能成为“left”word。这意味着你的结果不是确定的，有时会得到一种结果，有时又会得到另一种结果。

count

count方法用于获取数据集中的元素个数，比如，可以使用它计算RDD中的行数：

```
words.count()
```

countApprox

它是count方法的一个近似方法，用于返回大概的计数结果，但它必须在指定时间内执行完成（当超过指定时间时，返回一个不准确的近似结果）。

confidence（置信度）表示近似结果的误差区间包括真实值的概率，也就是说，如果以0.9的置信度重复调用countApprox方法，则表示在90%的情况下的近似计数结果中为真实的计数值。置信度取值必须在[0，1]范围内，否则会抛出异常：

```
val confidence = 0.95
val timeoutMilliseconds = 400
words.countApprox(timeoutMilliseconds, confidence)
```

countApproxDistinct

countApproxDistinct函数用来计算去重后的值的大约个数。它有两种实现，都是基于streamlib实现的这篇文章的方法“HyperLogLog in Practice Algorithmic Engineering of a State-of-the-Art Cardinality Estimation Algorithm”。

在第一种函数实现中，传入函数的参数是相似精度。值与值之间的相似度达到该参数指定值，则将其看作是一样的值。相似精度值越小，代表值与值之间越相似，计数结果可能越大。该值要求必须大于0.000017：

```
words.countApproxDistinct(0.05)
```

另一种函数实现中，则可以基于两个参数来指定相似精度：一个是用于“常规”数据，另一个是用于稀疏数据。

这两个参数分别是p和sp，其中p表示准确率（precision），sp表示稀疏准确率（sparse precision）。相似精度约为1.054/sqrt（2^p）。设置非零值sp（sp>p）能减少内存消耗，并在处理低选择性数据时提高准确性。p和sp均为整数：

```
words.countApproxDistinct(4, 10)
```

countByValue

此方法用于对给定RDD中的值的个数进行计数，它需要将结果集加载到驱动器的内存中来实现计数。因为整个结果map映射集合都会加载到驱动器的内存中，所以只有当结果映射集预计很小时才使用此方法。因此，只有在总行数较少或不同Key数量较少的情况下，才适合使用此方法：

```
words.countByValue()
```

countByValueApprox

它和countByValue函数的功能是一样的，只是它返回的是一个近似值。该函数必须在指定的timeout（第一个参数）时间内返回结果（如果超过了timeout时间时，则会返回一个未完成的结果）。

置信度表示返回结果集在误差范围内包含真实值的概率，也就是说，如果以0.9的置信度重复调用countApprox函数，则表示期望有90%的概率在结果中包含真实的计数值。置信度取值必须在[0，1]范围内，否则将抛出异常：

```
words.countByValueApprox（1000，0.95）
```

first

first方法返回数据集中的第一个值：

```
words.first()
```

max和min

max和min方法则分别返回最大值和最小值：

```
spark.sparkContext.parallelize(1 to 20).max()
spark.sparkContext.parallelize(1 to 20).min()
```

take

take和它的派生方法的功能是从RDD中读取一定数量的值。具体执行流程是：首先扫描一个数据分区，然后根据该分区的实际返回结果的数量来预估还需要再读取多少个分区。

这个函数有很多变体，例如takeOrdered，takeSample和top函数。takeSample函数用于从RDD中获取指定大小的随机样本，并可以指定withReplacement（采样过程是否允许替换）、返回样本数量和随机数种子这3个参数。按默认排序返回前几个值时，top函数与takeOrdered函数返回值的排序顺序相反：

```
words.take(5)
words.takeOrdered(5)
words.top(5)
val withReplacement = true
val numberToTake = 6
val randomSeed = 100L
words.takeSample(withReplacement, numberToTake, randomSeed)
```

保存文件

保存文件指将RDD写入纯文本文件。对于RDD，不能以常规的方式读取它并“保存”到数据源中。而是必须遍历分区才能将每个分区的内容保存到某个外部数据库，这是一种低级方法，它展现了在更高级的API中所执行的基础操作。Spark会把RDD中每个分区都读取出来，并写到指定位置中。

saveAsTextFile

要将RDD保存到文本文件中，只需指定文件路径和压缩编码器（可选参数）即可：

```
words.saveAsTextFile("file:/tmp/bookTitle")
```

要设置一个压缩编码器，则必须从Hadoop中导入适当的编码器。在org.apache.hadoop.io.compress库中可找到这些编解码器：

```
// in Scala
import org.apache.hadoop.io.compress.BZip2Codec
words.saveAsTextFile("file:/tmp/bookTitleCompressed", classOf [BZip2Codec])
```

SequenceFiles

Spark最初是由Hadoop生态系统发展而来的，因此它与各种Hadoop工具紧密集成。sequenceFile是一种由二进制键值对（key-value）组成的文件，它也是Hadoop MapReduce作业中常用的输入/输出格式。

Spark中可以使用saveAsObjectFile方法或者显式写出键值对的方式将RDD写入sequenceFile格式文件中，在第13章中会详细介绍：

```
words.saveAsObjectFile("/tmp/my/sequenceFilePath")
```

Hadoop文件

你可以将RDD保存为多种不同的Hadoop文件格式，并可指定类、输出格式、Hadoop配置和压缩方式[想了解这些格式的具体信息，请阅读《Hadoop：The Definitive Guide》（O'Reilly，2015）]这本书，除非你在深入研究Hadoop生态系统或在优化一些传统的mapReduce作业，否则这些格式基本无关紧要。

缓存

缓存RDD的原理与缓存DataFrame和Dataset相同，你可以缓存或持久化RDD。 默认情况下，仅对内存中的数据进行缓存和持久化。可以使用本章前面提到的setName函数来命名它：

```
words.cache()
```

可以通过org.spache.spark.storage.StorageLevel来指定单例对象的任意存储级别，存储级别包括仅内存（memory only）、仅磁盘（disk only）、堆外内存（off heap）。

接着来查询一下存储级别（在第20章中介绍持久性时，将详细说明存储级别）：

```
// in Scala
words.getStorageLevel

# in Python
words.getStorageLevel()
```

检查点

DataFrame API中没有检查点（checkpointing）这个概念。检查点是将RDD保存到磁盘上的操作，以便将来对此RDD的引用能直接访问磁盘上的那些中间结果，而不需要从其源头重新计算RDD。它与缓存类似，只是它不存储在内存中，只存储在磁盘上。这在执行迭代计算时很有用，与缓存的用例类似：

```
spark.sparkContext.setCheckpointDir("/some/path/for/checkpointing")
words.checkpoint()
```

当引用此RDD时，它将从检查点直接获得而非从源数据重新计算而来，这是个很有用的优化。

通过pipe方法调用系统命令操作RDD

pipe方法可能是Spark最有趣的方法之一，通过pipe方法，可以利用管道技术调用外部进程来生成RDD。将每个数据分区交给指定的外部进程来计算得到结果RDD，每个输入分区的所有元素被当做另一个外部进程的标准输入，输入元素由换行符分隔。最终结果由该外部进程的标准输出生成，标准输出的每一行产生输出分区的一个元素。空分区也会调用一个外部进程。

print操作可以通过提供的两个函数来进行适配。

举一个简单的例子：将每个分区传递给wc命令。每个元素将作为一个输入行，因此如果执行行计数，将得到每个分区的总行数：

```
words.pipe("wc -l").collect()
```

在上面例子中，结果是每个分区有五行。

mapPartitions

前面的命令已表明Spark在实际执行代码时是基于每个分区运行的。你可能已经注意到了，RDD上的map函数实际上是基于MapPartitionsRDD来实现的，map只是mapPartitions基于行操作的一个别名，mapPartitions函数每次处理一个数据分区

（可以通过迭代器来遍历该数据分区）。实际上在集群中我们也是每次处理一个分区，而不是具体的一行。举个简单的例子：为数据中的每个分区创建值“1”，以下表达式中计算总和将得出我们拥有的分区数量：

```
// in Scala
words.mapPartitions(part => Iterator[Int](1)).sum() // 2

# in Python
words.mapPartitions(lambda part: [1]).sum() # 2
```

这意味着我们能按照每个分区进行操作，处理单元为整个分区。在RDD的整个子数据集上执行某些操作很有用，你可以将属于某类的值收集到一个分区中，或分组到一个分区上，然后对整个分组进行操作。例如你可以通过一些自定义的机器学习算法来管理这个数据集，并基于该公司的部分数据集训练一个独立的模型。一个Facebook工程师曾对pipe操作的实现制作了一个有趣的演示，并在Spark Summit East 2017上展示了类似的用例。

与mapPartitions函数类似的函数还有mapPartitionsWithIndex函数。mapPartitionsWithIndex接收一个指定函数作为参数，该指定函数接收两个参数，一个为分区的索引和另一个为遍历分区内所有项的迭代器。分区索引是RDD中的分区号，它标识数据集中每条记录的位置（便于利用此信息调试），你可以使用mapPartitionsWithIndex来测试map函数是否正确运行：

```
// in Scala
def indexedFunc(partitionIndex:Int, withinPartIterator: Iterator[String]) = {
  withinPartIterator.toList.map(
    value => s"Partition: $partitionIndex => $value").iterator
}
words.mapPartitionsWithIndex(indexedFunc).collect()

# in Python
def indexedFunc(partitionIndex, withinPartIterator):
  return ["partition: {} => {}".format(partitionIndex,
    x) for x in withinPartIterator]
words.mapPartitionsWithIndex(indexedFunc).collect()
```

foreachPartition

mapPartitions函数需要返回值才能正常执行，但foreachPartition函数不需要。foreachPartition函数仅用于迭代所有的数据分区，与mapPartitions的不同在于它没有返回值，这使得它非常适合像写入数据库这样的操作（不需要返回计算结果）。实际上，许多数据源连接器也是基于此函数实现的。如果需要创建自己的文本文件源，可以基于一个随机ID指定输出到临时目录中来实现：

```
words.foreachPartition { iter =>
  import java.io._
  import scala.util.Random
  val randomFileName = new Random().nextInt()
  val pw = new PrintWriter(new File(s"/tmp/random-file-${randomFileName}.txt"))
  while (iter.hasNext) {
      pw.write(iter.next())
  }
  pw.close()
}
```

执行完上述代码后，浏览*/tmp*目录，便会发现这两个文件。

glom

glom是一个有趣的函数，它用于将数据集中的每个分区都转换为数组。当需要将数据收集到驱动器并想为每个分区创建一个数组时，就很适合用glom函数实现。但是，这可能会导致严重的稳定性问题，因为当有很大的分区或大量分区时，该函数很容易导致驱动器崩溃。

下面例子中，可以看到其中有两个分区，每个词落入其中一个分区：

```
// in Scala
spark.sparkContext.parallelize(Seq("Hello", "World"), 2).glom().collect()
// 结果是Array(Array(Hello), Array(World))

# in Python
spark.sparkContext.parallelize(["Hello", "World"], 2).glom().collect()
# 结果是[['Hello'], ['World']]
```

小结

本章介绍了RDD API的基础知识，包括单个RDD的操作。第13章将涉及更高级的RDD概念，例如连接操作和键值RDD。

第13章

高级RDD

第12章讲解了基本的简单 RDD 操作，包括如何创建 RDD以及为什么使用RDD。此外，我们还讨论了映射、过滤、约减，以及如何创建函数来转换RDD 数据。本章介绍高级 RDD 操作，重点介绍键值RDD，这是操作数据的一种强大抽象形式。我们还涉及到一些更高级的主题，如自定义分区，这是你可能最想要使用 RDD 的原因。使用自定义分区函数，你可以精确控制数据在集群上的分布，并相应地操作单个分区。在开始介绍之前，首先总结一下将讨论的关键主题：

- 聚合和key-value RDD。
- 自定义分区。
- RDD 连接。

这一系列的API 是Spark项目创建之初使用的API，网上有很多相关的例子，所以很容易找到如何实现这些操作的样例。

我们使用上一章中使用的数据集：

```
// in Scala
val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"
  .split(" ")
val words = spark.sparkContext.parallelize(myCollection, 2)

# in Python
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
```

```
  .split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
```

Key-Value基础 (Key-Value RDD)

基于RDD的许多方法要求数据是key-value格式，这种方法都有形如<some-operation>ByKey的API名称，只要在方法名称中看到 ByKey，就意味着只能以 PairRDD 类型执行此操作。最简单的方法就是将当前的 RDD 映射到基本的key-value结构，也就是说在 RDD 的每个记录中都有两个值：

```
// in Scala
words.map(word => (word.toLowerCase, 1))

# in Python
words.map(lambda word: (word.lower(), 1))
```

keyBy

前面的示例演示了创建key的简单方法，但是也可以使用keyBy函数，它是根据当前value创建key的函数。本例中，将单词中的第一个字母作为key，然后Spark将该单词记录保存为RDD 的value：

```
// in Scala
val keyword = words.keyBy(word => word.toLowerCase.toSeq(0).toString)

# in Python
keyword = words.keyBy(lambda word: word.lower()[0])
```

对值进行映射

在有一组键值对之后，你可以开始对它们进行操作。如果我们有一个元组，Spark将假设第一个元素是key，第二个是value。在这种格式中，你可以显式选择映射value(并忽略key)。当然，可以手动执行此操作，但当你知道只是要修改value时，这可以帮助防止错误：

```
// in Scala
keyword.mapValues(word => word.toUpperCase).collect()

# in Python
keyword.mapValues(lambda word: word.upper()).collect()
```

下面是 Python 中的输出：

```
[('s', 'SPARK'),
```

```
('t', 'THE'),
('d', 'DEFINITIVE'),
('g', 'GUIDE'),
(':', ':'),
('b', 'BIG'),
('d', 'DATA'),
('p', 'PROCESSING'),
('m', 'MADE'),
('s', 'SIMPLE')]
```

(在Scala 中的结果相同，本处省略。)

正如第12章所介绍的，你可以在行（row）上进行flatMap操作来扩展行数，使每行表示一个字符。在下面的示例中，我们将省略输出，它只是输出每个字符，因为我们将单词转换成了字符数组：

```
// in Scala
keyword.flatMapValues(word => word.toUpperCase).collect()

# in Python
keyword.flatMapValues(lambda word:  word.upper()).collect()
```

提取key和value

当我们的数据是键值对这种格式时，我们还可以使用以下方法提取特定的key或value：

```
// in Scala
keyword.keys.collect()
keyword.values.collect()

# in Python
keyword.keys().collect()
keyword.values().collect()
```

lookup

在RDD 上很常见的任务就是查找某个key对应的value。请注意，并不强制规定每一个输入都只有一个键，所以如果当我们查找“s”时，我们将获得与该key相关的两个value，即“Spark”和“Simple”：

```
keyword.lookup("s")
```

sampleByKey

有两种方法可以通过一组key来采样 RDD，这可以是近似的方法也可以是精确的方

法。这两种操作都可以使用或不使用替换策略，以及根据给定键值对数据集部分采样。这是通过对RDD的一次遍历来简单随机采样，采样数量大约是所有key-value对数量的math.ceil(numItems * samplingRate)这么多：

```
// in Scala
val distinctChars = words.flatMap(word => word.toLowerCase.toSeq).distinct
  .collect()
import scala.util.Random
val sampleMap = distinctChars.map(c => (c, new Random().nextDouble())).toMap
words.map(word => (word.toLowerCase.toSeq(0), word))
  .sampleByKey(true, sampleMap, 6L)
  .collect()

# in Python
import random
distinctChars = words.flatMap(lambda word: list(word.lower())).distinct()\
  .collect()
sampleMap = dict(map(lambda c: (c, random.random()), distinctChars))
words.map(lambda word: (word.lower()[0], word))\
  .sampleByKey(True, sampleMap, 6).collect()
```

下面使用sampleByKeyExact的方法不同于sampleByKey，因为它需要额外遍历 RDD，以99.99%的置信度构造大小等于所有key-value对数量的math.ceil(numItems * samplingRate)这么多的样本集合。若设置不替换，则要一次额外遍历 RDD以保证样本集大小，若设置替换取样，需要另外两次额外遍历：

```
// in Scala
words.map(word => (word.toLowerCase.toSeq(0), word))
  .sampleByKeyExact(true, sampleMap, 6L).collect()
```

聚合操作

你可以在纯 RDD或 PairRDD上执行聚合操作，具体取决于所使用的方法。下面使用数据集来演示一下：

```
// in Scala
val chars = words.flatMap(word => word.toLowerCase.toSeq)
val KVcharacters = chars.map(letter => (letter, 1))
def maxFunc(left:Int, right:Int) = math.max(left, right)
def addFunc(left:Int, right:Int) = left + right
val nums = sc.parallelize(1 to 30, 5)

# in Python
chars = words.flatMap(lambda word: word.lower())
KVcharacters = chars.map(lambda letter: (letter, 1))
def maxFunc(left, right):
  return max(left, right)
def addFunc(left, right):
```

```
  return left + right
nums = sc.parallelize(range(1,31), 5)
```

明白了这些内容之后就可以执行类似countByKey的操作，它对每个key对应的项进行计数。

countByKey

可以计算每个key对应的数据项的数量，并将结果写到本地Map中。你还可以近似地执行此操作，在Scala或Java中指定超时时间和置信度：

```
// in Scala
val timeout = 1000L //毫秒
val confidence = 0.95
KVcharacters.countByKey()
KVcharacters.countByKeyApprox(timeout, confidence)

# in Python
KVcharacters.countByKey()
```

了解聚合操作的实现

有几种方法可以创建key-value PairRDD，但是实现方法对任务的稳定性非常重要。我们比较两个基本的实现方法：groupBy和reduce。我们仅介绍groupByKey和reduceByKey的实现，groupBy和reduceBy的实现思路类似。

groupByKey

阅读了API 文档后，你可能会觉得groupByKey配合使用map操作是汇总每个key数量的最佳方法：

```
// in Scala
KVcharacters.groupByKey().map(row => (row._1, row._2.reduce(addFunc))).collect()

# in Python
KVcharacters.groupByKey().map(lambda row: (row[0], reduce(addFunc, row[1])))\
  .collect()
#注意这是Python 2版本的代码。使用Python 3的话，reduce方法需要由functools库导入
```

但是，在大多数情况下，这是错误的方法。根本问题是每个执行器在执行函数之前必须在内存中保存一个key对应的所有value。这会有什么问题么？如果有严重的key负载倾斜现象，则某些分组可能由于key对应着太多的value而导致超载问题，进而出现OutOfMemoryErrors错误。当前的小数据集显然不会出现这种问题，但它可能会在处理大规模数据时爆发严重问题。这不一定会发生，但它可能会发生。

groupByKey在某些情况下是可以的。如果每个key的value数量都差不多，并且知道它们能够被执行器的内存容纳那就可以了。对于其他情况，有一种首选的方法，就是使用reduceByKey。

reduceByKey

因为我们是执行一个简单的计数，一个更稳妥的方法是同样执行flatMap，然后执行map将每个字母实例映射为数字1，然后执行reduceByKey配以求和函数以将结果存储到数组。这种方法更加稳定，因为reduce发生在每个分组，并且不需要将所有内容放在内存中。此外，此操作不会导致shuffle过程，在执行最后的reduce之前所有的任务都在每个工作节点单独执行。这大大提高了执行速度以及该操作的稳定性：

```
KVcharacters.reduceByKey(addFunc).collect()
```

以下是返回结果：

```
Array((d,4), (p,3), (t,3), (b,1), (h,1), (n,2),
...
(a,4), (i,7), (k,1), (u,1), (o,1), (g,3), (m,2), (c,1))
```

reduceByKey方法返回一个组(对应一个key)的 RDD 和不保证有序的元素序列。因此，当我们的任务操作满足结合律时，这种方法是完全可行的，而如果元素的顺序很重要时则就不适合了。

其他聚合方法

还有许多高级聚合方法，使用它们主要取决于具体工作负载，而我们发现在当今Spark作业中，用户极少遇到这种工作负载 (或需要执行这种操作) 。因为使用结构化API 执行更简单的聚合时，很少会使用这些非常低级的工具。这些函数允许你在极大程度上非常具体地、非常低级地控制在集群上执行某种聚合操作。

aggregate

有一个函数叫aggregate，此函数需要一个 null 值和一个起始值，并需要你指定两个不同的函数，第一个函数执行分区内聚合，第二个执行分区间聚合。起始值在两个聚合级别都使用：

```
// in Scala
nums.aggregate(0)(maxFunc, addFunc)

# in Python
nums.aggregate(0, maxFunc, addFunc)
```

aggregate确实有一些性能问题，因为它在驱动器上执行最终聚合。如果执行器的结果太大，则会导致驱动器出现OutOfMemoryError错误并最终让程序崩溃。还有另一个方法treeAggregate，它基于不同的实现方法可以得到与aggregate相同的结果。它基本上是以 "下推"方式完成一些子聚合(创建从执行器到执行器传输聚合结果的树)，最后再执行最终聚合。多层级的形式确保驱动器在聚合过程中不会耗尽内存，这些基于树的实现通常会提高某些操作的稳定性：

```
// in Scala
val depth = 3
nums.treeAggregate(0)(maxFunc, addFunc, depth)

# in Python
depth = 3
nums.treeAggregate(0, maxFunc, addFunc, depth)
```

aggregateByKey

此函数与aggregate基本相同，但是基于key聚合而非基于分区聚合。起始值和函数的属性配置也都相同：

```
// in Scala
KVcharacters.aggregateByKey(0)(addFunc, maxFunc).collect()

# in Python
KVcharacters.aggregateByKey(0, addFunc, maxFunc).collect()
```

combineByKey

不但可以指定聚合函数，还可以指定一个合并函数。该合并函数针对某个key进行操作，并根据某个函数对value合并，然后合并各个合并器的输出结果并得出最终结果。我们还可以按照自定义输出分区程序指定输出分区的数量：

```
// in Scala
val valToCombiner = (value:Int) => List(value)
val mergeValuesFunc = (vals:List[Int], valToAppend:Int) => valToAppend :: vals
val mergeCombinerFunc = (vals1:List[Int], vals2:List[Int]) => vals1 ::: vals2
// 现在将这些定义为函数变量
val outputPartitions = 6
KVcharacters
  .combineByKey(
    valToCombiner,
    mergeValuesFunc,
    mergeCombinerFunc,
    outputPartitions)
  .collect()

# in Python
def valToCombiner(value):
```

```
  return [value]
def mergeValuesFunc(vals, valToAppend):
  vals.append(valToAppend)
  return vals
def mergeCombinerFunc(vals1, vals2):
  return vals1 + vals2
outputPartitions = 6
KVcharacters\
  .combineByKey(
    valToCombiner,
    mergeValuesFunc,
    mergeCombinerFunc,
    outputPartitions)\
  .collect()
```

foldByKey

foldByKey使用满足结合律函数和中性的“零值”合并每个key的value，支持多次累积到结果，并且不能更改结果 (例如，0 为加法，或1为乘法)：

```
// in Scala
KVcharacters.foldByKey(0)(addFunc).collect()

# in Python
KVcharacters.foldByKey(0, addFunc).collect()
```

CoGroups

CoGroups在Scala中允许将三个key-value RDD一起分组，在Python中允许将两个 key-value RDD 一起分组。它基于key连接value，这实际上等效于基于组的RDD连接操作。执行此操作时，还可以指定多个输出分区或自定义分区函数，以精确控制此数据在整个集群上的分布情况 (我们将在本章稍后部分讨论分区函数)：

```
// in Scala
import scala.util.Random
val distinctChars = words.flatMap(word => word.toLowerCase.toSeq).distinct
val charRDD = distinctChars.map(c => (c, new Random().nextDouble()))
val charRDD2 = distinctChars.map(c => (c, new Random().nextDouble()))
val charRDD3 = distinctChars.map(c => (c, new Random().nextDouble()))
charRDD.cogroup(charRDD2, charRDD3).take(5)
# in Python
import random
distinctChars = words.flatMap(lambda word: word.lower()).distinct()
charRDD = distinctChars.map(lambda c: (c, random.random()))
charRDD2 = distinctChars.map(lambda c: (c, random.random()))
charRDD.cogroup(charRDD2).take(5)
```

结果是一组key-value对，key在一边，其所有的value在另一边。

连接操作

RDD的连接与结构化API中的连接有很多相同之处，它们都遵循相同的基本格式，包括执行连接操作的两个RDD，以及输出分区数或自定义分区函数。我们将在本章后面部分讨论分区函数。

内连接

下面给出内连接的示例代码。请注意，我们如何设置输出分区数：

```
// in Scala
val keyedChars = distinctChars.map(c => (c, new Random().nextDouble()))
val outputPartitions = 10
KVcharacters.join(keyedChars).count()
KVcharacters.join(keyedChars, outputPartitions).count()

# in Python
keyedChars = distinctChars.map(lambda c: (c, random.random()))
outputPartitions = 10
KVcharacters.join(keyedChars).count()
KVcharacters.join(keyedChars, outputPartitions).count()
```

本书不会提供其他连接类型的示例，但它们都遵循相同的基本格式。可以从第8章中了解以下连接类型：

- 全外连接（`fullOuterJoin`）。
- 左外连接（`leftOuterJoin`）。
- 右外连接（`rightOuterJoin`）。
- 笛卡尔连接(cartesian，再次说明该连接操作非常危险!它不接受连接key，并且可能会有大量的输出)。

zip

`zip`其实并不是一个连接操作，但它将两个 RDD组合在一起，因此我们暂将它归类为连接操作。`zip`把两个 RDD的元素对应的匹配在一起，要求两个RDD的元素个数相同，同时也要求两个RDD的分区数也相同，结果会生成一个`PairRDD`：

```
// in Scala
val numRange = sc.parallelize(0 to 9, 2)
words.zip(numRange).collect()

# in Python
numRange = sc.parallelize(range(10), 2)
```

```
words.zip(numRange).collect()
```

下面是结果，即一个匹配后的key-value数组：

```
[('Spark', 0),
 ('The', 1),
 ('Definitive', 2),
 ('Guide', 3),
 (':', 4),
 ('Big', 5),
 ('Data', 6),
 ('Processing', 7),
 ('Made', 8),
 ('Simple', 9)]
```

控制分区

使用 RDD，你可以控制数据在整个集群上的物理分布，其中一些方法与结构化 API 中的基本相同，但是最关键的区别(在结构化 API 中不支持的)在于，它可以指定一个数据分区函数(自定义Partitioner，将随后介绍)。

coalesce

coalesce有效地折叠（collapse）同一工作节点上的分区，以便在重新分区时避免数据洗牌（shuffle）。例如，存储words变量的RDD 当前有两个分区，可以使用coalesce将其折叠为一个分区，从而避免了数据shuffle：

```
// in Scala
words.coalesce(1).getNumPartitions // 1

# in Python
words.coalesce(1).getNumPartitions() # 1
```

repartition

Repartition操作将对数据进行重新分区，跨节点的分区会执行shuffle操作。对于map和filter操作，增加分区可以提高并行度：

```
words.repartition(10) // 10个分区
```

repartitionAndSortWithinPartitions

此操作将对数据重新分区，并指定每个输出分区的顺序。此操作的文档很完善，此处

就不举例说明了。使用repartitionAndSortWithinPartition，分区和key比较函数可由用户定义。

自定义分区

自定义分区是使用RDD的主要原因之一，而结构化 API不支持自定义数据分区，RDD包含影响任务能否成功运行的低级实现细节。自定义分区的典型示例是PageRank实现，你需要控制集群上数据的分布并避免shuffle操作，而在我们的shopping数据集中，可能需要我们根据客户 ID 对数据进行分区 (我们随后就会看到这个例子)。

简而言之，自定义分区的唯一目标是将数据均匀地分布到整个集群中，以避免诸如数据倾斜之类的问题。

如果要使用自定义分区，则应从结构化 API定义的数据降级为RDD，应用自定义分区程序，然后再将RDD转换回 DataFrame 或Dataset。只有真正需要时，才会使用RDD自定义分区，这样就可以利用两方面的优势。

要执行自定义分区，你需要实现Partitioner的子类。只有当你很了解特定领域知识时，你才需要这样做。如果你只是想对一个值或一组值 (列) 进行分区，那么用DataFrame API实现就可以了。

让我们看一个示例：

```
// in Scala
val df = spark.read.option("header", "true").option("inferSchema", "true")
  .csv("/data/retail-data/all/")
val rdd = df.coalesce(10).rdd

# in Python
df = spark.read.option("header", "true").option("inferSchema", "true")\
  .csv("/data/retail-data/all/")
rdd = df.coalesce(10).rdd
df.printSchema()
```

Spark有两个内置的分区器，你可以在 RDD API 中调用，它们是用于离散值划分的HashPartitioner（基于哈希的分区）以及RangePartitioner（根据数值范围分区），这两个分区器分别针对离散值和连续值。Spark的结构化 API 已经包含了它们，也可以在 RDD中使用它们：

```
// in Scala
import org.apache.spark.HashPartitioner
rdd.map(r => r(6)).take(5).foreach(println)
val keyedRDD = rdd.keyBy(row => row(6).asInstanceOf[Int].toDouble)
```

```
keyedRDD.partitionBy(new HashPartitioner(10)).take(10)
```

虽然哈希分区和范围分区程序都很有用，但它们是最基本的分区方法。有时，因为数据量很大并且存在严重的数据倾斜（由于某些key对应的value项比其他key对应的value项多很多导致的数据倾斜），你将需要实现一些非常底层的分区方法。你希望尽可能多地拆分这些key以提高并行性，并在执行过程中防止OutOfMemoryError错误发生。

一个典型情况是，（当且仅当某个key有特定形式时）由于某个key对应的value太多，需要把这个key拆分成很多key。例如，数据集中可能对某两个客户的数据处理总是会使分析过程崩溃，我们需要对这两个客户数据进行细分，就是说比其他客户ID更细粒度地分解它们。由于这两个key倾斜的情况很严重，所以需要特别处理，而其他的key可以被集中到大组中。这虽然是一个极端的例子，但你可能会在数据中看到类似的情况：

```
// in Scala
import org.apache.spark.Partitioner
class DomainPartitioner extends Partitioner {
 def numPartitions = 3
 def getPartition(key: Any): Int = {
   val customerId = key.asInstanceOf[Double].toInt
   if (customerId == 17850.0 || customerId == 12583.0) {
     return 0
   } else {
     return new java.util.Random().nextInt(2) + 1
   }
 }
}

keyedRDD
  .partitionBy(new DomainPartitioner).map(_._1).glom().map(_.toSet.toSeq.length)
  .take(5)
```

运行代码后，你将看到每个分区中的结果数量，而第二个分区和第三个分区的数量会有所不同，因为后两个分区是随机分布的 (我们将在 Python 中完成同样的操作)，但划分原则是一致的：

```
# in Python
def partitionFunc(key):
  import random
  if key == 17850 or key == 12583:
    return 0
  else:
    return random.randint(1,2)

keyedRDD = rdd.keyBy(lambda row: row[6])
keyedRDD\
```

```
  .partitionBy(3, partitionFunc)\
  .map(lambda x: x[0])\
  .glom()\
  .map(lambda x: len(set(x)))\
  .take(5)
```

自定义key分发的逻辑仅在 RDD 级别适用。当然，这是一个简单的示例，但它展示了以任意逻辑在集群中部署数据的能力。

自定义序列化

最后一个主题是*Kryo* 序列化问题，任何你希望并行处理（或函数操作）的对象都必须是可序列化的：

```
// in Scala
class SomeClass extends Serializable {
  var someValue = 0
  def setSomeValue(i:Int) = {
    someValue = i
    this
  }
}

sc.parallelize(1 to 10).map(num => new SomeClass().setSomeValue(num))
```

默认序列化的方式可能很慢，Spark可以使用 Kryo 库 (第2版) 更快地序列化对象。Kryo 序列化的速度比 Java 序列化更快，压缩也更紧凑 (通常是 10倍)，但并不是支持所有序列化类型的，并且要求你先注册在程序中使用的类。

你可以借助于SparkConf使用 Kryo 初始化你的任务，并设置“`spark.serializer`”为“`org.apache.spark.serializer.KryoSerializer`”(我们在本书的下一部分中讨论这一点)。此配置用于在工作节点之间数据传输时或将 RDD写到磁盘上时，Spark所采用序列化工具。Spark没有选择Kryo做为默认序列化工具的原因是它要求自定义注册，但我们建议在网络传输量大的应用程序中尝试使用它。自Spark 2.0.0 后，我们在对简单类型、简单类型数组或字符串类型的 RDD进行shuffle操作时，已经默认采用 Kryo 序列化。

Spark为Twitter chill库中AllScalaRegistrar涵盖的许多常用核心 Scala 类自动使用了 Kryo 序列化。

若要借助于Kryo注册自定义类，请使用`registerKryoClasses`方法：

```
// in Scala
val conf = new SparkConf().setMaster(...).setAppName(...)
```

```
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

小结

在本章中，我们讨论了有关 RDD的许多更高级的主题。特别需要注意的部分是自定义分区，它允许你以特定的函数来划分数据。在第14 章中，我们将讨论另一个Spark的低级工具：分布式变量。

第14章

分布式共享变量

除了弹性分布式数据集RDD外，Spark的第二种低级 API 是“分布式共享变量”。它包括两种类型：广播变量（broadcast variable）和累加器（accumulator）。这些变量可以在用户定义函数中使用（例如，在RDD 或 DataFrame 上的 map 函数中），在集群上运行时具有特殊性质。具体地说，累加器将所有任务中的数据累加到一个共享结果中（例如，实现一个计数器，以便可以查看有多少输入记录无法解析）。广播变量允许你在所有工作节点上保存一个共享值，当在Spark各种操作中重用它时，就不需要将其重新在机器间传输。本章讨论了这些分布式共享变量类型及其使用方法。

广播变量

通过广播变量可以在集群上有效地共享（只读的）不变量，而不需要将其封装到函数中去。在驱动节点上使用变量的一般方法是简单地在函数闭包（function closure）中引用它(例如，在map操作中)，但这种方式效率很低，尤其是对于大数据变量来说 (如数据库表或机器学习模型)。原因在于，当在闭包（closure）中使用变量时，必须在工作节点上执行多次反序列化 (每个任务一次)。此外，如果你在多个Spark操作和作业中使用相同的变量，它将被重复发送到工作节点上的每一个作业中，而不是只发送一次。

这就是引入广播变量的原因所在。广播变量是共享的、不可修改的变量，它们缓存在集群中的每个节点上，而不是在每个任务中都反复序列化。一个典型的应用场景是把一个大型查找表作为广播变量传递，它适合于执行器（executor）的内存并在函数中使用，如图 14-1所示。

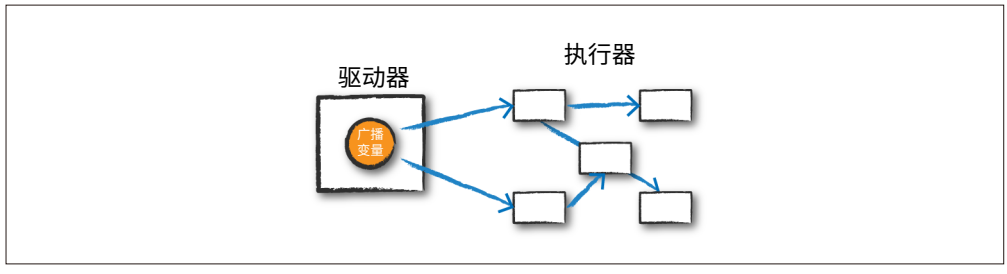


图14-1：广播变量

例如，假设你有一个包含词或值的列表：

```
// in Scala
val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"
  .split(" ")
val words = spark.sparkContext.parallelize(myCollection, 2)

# in Python
my_collection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
  .split(" ")
words = spark.sparkContext.parallelize(my_collection, 2)
```

你希望用其他信息补充你的单词列表，它的大小可能是KB、MB甚至是GB级别的，如果我们从 SQL 的角度考虑它，这是一个右连接：

```
// in Scala
val supplementalData = Map("Spark" -> 1000, "Definitive" -> 200,
                           "Big" -> -300, "Simple" -> 100)

# in Python
supplementalData = {"Spark":1000, "Definitive":200,
                    "Big":-300, "Simple":100}
```

我们可以通过Spark广播它并使用`suppBroadcast`来引用它。当触发动作操作时，该值是不可变的，并且惰性复制到集群中的所有节点上：

```
// in Scala
val suppBroadcast = spark.sparkContext.broadcast(supplementalData)

# in Python
suppBroadcast = spark.sparkContext.broadcast(supplementalData)
```

我们通过value方法引用此广播变量的值。该方法在序列化函数中是可访问的，无需对数据进行序列化。这可以节省大量的序列化和反序列化的成本，因为Spark在集群中通过广播更高效地传输数据：

```
// in Scala
```

```
suppBroadcast.value

# in Python
suppBroadcast.value
```

现在我们可以用这个值转换RDD。在这个例子中，我们将根据在广播map变量中可能拥有的value来创建一个key-value对。如果没有value，我们将简单地用0替代它：

```
// in Scala
words.map(word => (word, suppBroadcast.value.getOrElse(word, 0)))
  .sortBy(wordPair => wordPair._2)
  .collect()

# in Python
words.map(lambda word: (word, suppBroadcast.value.get(word, 0)))\
  .sortBy(lambda wordPair: wordPair[1])\
  .collect()
```

在Python中会返回以下值，在Scala中也会返回是具有相同值的数组类型：

```
[('Big', -300),
 ('The', 0),
...
 ('Definitive', 200),
 ('Spark', 1000)]
```

共享变量和函数闭包中传递变量的区别在于，我们以更有效的方式完成了这一操作 。当然，这取决于数据的大小和执行器的数量，对于小型集群的小数据 (KB级别的)可能不是。这个例子中的小型字典数据集可能不需要太大的成本，但如果你要用一个数据量更大的值，为每个任务都序列化数据的成本可能是相当大的。

有一点值得注意的是，既可以在RDD中使用广播变量，也可以是在UDF或Dataset中使用广播变量，都将获得相同的结果。

累加器

Spark第二种类型的共享变量是累加器(见图14-2)，它用于将转换操作更新的值以高效和容错的方式传输到驱动节点。

累加器提供一个累加用的变量，Spark集群可以以按行方式对其进行安全更新，你可以用它来进行调试(例如，跟踪每个分区中某个变量的值) 或创建低级聚合。累加器仅支持由满足交换律和结合律的操作进行累加的变量，因此对累加器的操作可以被高效并行，你可以使用累加器实现计数器 (如 MapReduce) 或求和操作。Spark提供对数字类型累加器的原生支持，程序员可以自行添加对新类型的支持。

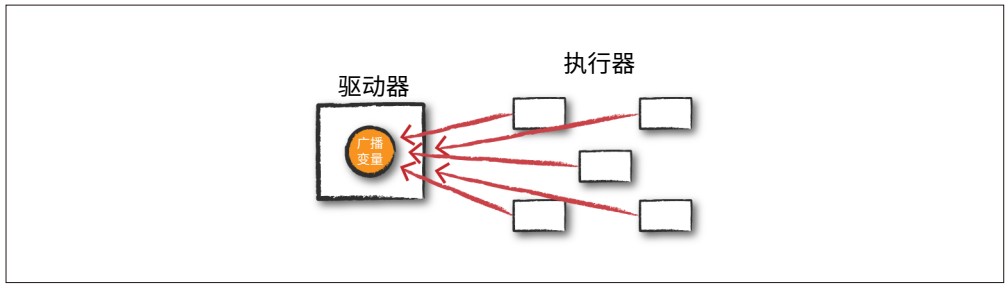


图14-2：加器变量

对于在仅发生在动作操作内执行的累加器更新，Spark保证每个任务对累加器的更新将只发生一次，重新启动的任务不会再次更新该值。但是在转换操作中，如果任务或作业阶段重新执行，则应注意每个任务的累加更新可能发生多次。

累加器也遵循Spark的惰性评估机制。如果 RDD 的某个操作要更新累加器，则它的值只会在实际计算 RDD 时更新 (例如，在对该 RDD 或依赖于它的 RDD 调用动作操作时)。因此，在像 map这样的惰性转换中，不保证累加器更新会被立即执行。

累加器可以是命名和未命名的。命名累加器将在Spark用户界面上显示它们的运行结果，而未命名的累加器则不会显示出来。

基本示例

我们在前面所创建的Flight数据集上执行自定义聚合来进行实验。在此示例中，我们将使用Dataset API 而不是 RDD API，但非常相似：

```
// in Scala
case class Flight(DEST_COUNTRY_NAME: String,
                  ORIGIN_COUNTRY_NAME: String, count: BigInt)
val flights = spark.read
  .parquet("/data/flight-data/parquet/2010-summary.parquet")
  .as[Flight]

# in Python
flights = spark.read\
  .parquet("/data/flight-data/parquet/2010-summary.parquet")
```

现在，我们创建一个累加器，它将计算往返中国的航班数量。尽管我们可以在 SQL 中以很简单的方式来执行此操作，但许多事情可能不会如此简单。累加器提供了一种以编程方式实现的计数器。下面演示如何创建未命名的累加器：

```
// in Scala
import org.apache.spark.util.LongAccumulator
```

```
val accUnnamed = new LongAccumulator
val acc = spark.sparkContext.register(accUnnamed)

# in Python
accChina = spark.sparkContext.accumulator(0)
```

我们的应用场景更适合于命名的累加器。有两种方法可以做到这一点：一个简短的方法和一个稍繁杂的方法。最简单的是使用SparkContext，或者我们可以实例化累加器然后使用名称对其注册：

```
// in Scala
val accChina = new LongAccumulator
val accChina2 = spark.sparkContext.longAccumulator("China")
spark.sparkContext.register(accChina, "China")
```

我们可以传递给LongAccumulator函数一个字符串值作为累加器的名称，或者将该字符串作为第二个参数传递到register函数中。命名累加器将显示在Spark用户界面（Spark UI）中，而未命名的累加器不会。

下一步是定义递增累加器的逻辑，这个过程很简单：

```
// in Scala
def accChinaFunc(flight_row: Flight) = {
  val destination = flight_row.DEST_COUNTRY_NAME
  val origin = flight_row.ORIGIN_COUNTRY_NAME
  if (destination == "China") {
    accChina.add(flight_row.count.toLong)
  }
  if (origin == "China") {
    accChina.add(flight_row.count.toLong)
  }
}

# in Python
def accChinaFunc(flight_row):
  destination = flight_row["DEST_COUNTRY_NAME"]
  origin = flight_row["ORIGIN_COUNTRY_NAME"]
  if destination == "China":
    accChina.add(flight_row["count"])
  if origin == "China":
    accChina.add(flight_row["count"])
```

现在，通过foreach方法遍历航班数据集中的每一行。这是因为foreach是一个动作操作，而Spark保证操作仅发生在动作操作内部。

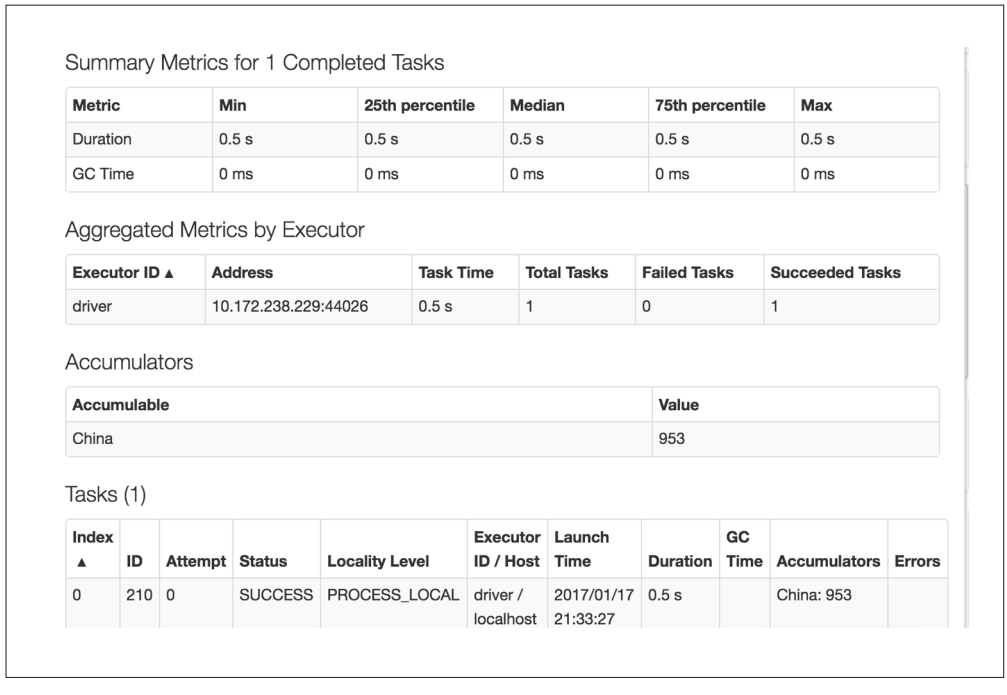
foreach方法为输入DataFrame中的每一行都运行一次我们定义的函数(假设我们没有过滤它)，逐行运行函数，相应地递增累加器：

```
// in Scala
```

```
flights.foreach(flight_row => accChinaFunc(flight_row))

# in Python
flights.foreach(lambda flight_row: accChinaFunc(flight_row))
```

这将很快完成，但是如果你切换到Spark UI，则可以在每个执行器上看到相关的值，这可以在以编程方式查询之前，如图 14-3所示。



Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.5 s	0.5 s	0.5 s	0.5 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks
driver	10.172.238.229:44026	0.5 s	1	0	1

Accumulators

Accumulable	Value
China	953

Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	210	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/01/17 21:33:27	0.5 s		China: 953	

图14-3：Spark用户界面上某执行器节点的信息

当然，我们也可以通过编程来访问它。为此，我们需要使用value属性：

```
// in Scala
accChina.value // 953

# in Python
accChina.value # 953
```

自定义累加器

尽管Spark确实提供了一些累加器类型，但有时你可能需要构建自己的自定义累加器。为此，你需要创建AccumulatorV2类的子类，实现几种抽象方法，如下面的示例所示。在此示例中，我们将只把偶数加到累加器里，虽然这再简单不过了，但它说明创建了自定义累加器的简单性：

```
// in Scala
import scala.collection.mutable.ArrayBuffer
import org.apache.spark.util.AccumulatorV2

val arr = ArrayBuffer[BigInt]()

class EvenAccumulator extends AccumulatorV2[BigInt, BigInt] {
  private var num:BigInt = 0
  def reset(): Unit = {
    this.num = 0
  }
  def add(intValue: BigInt): Unit = {
    if (intValue % 2 == 0) {
        this.num += intValue
    }
  }
  def merge(other: AccumulatorV2[BigInt,BigInt]): Unit = {
    this.num += other.value
  }
  def value():BigInt = {
    this.num
  }
  def copy(): AccumulatorV2[BigInt,BigInt] = {
    new EvenAccumulator
  }
  def isZero():Boolean = {
    this.num == 0
  }
}
val acc = new EvenAccumulator
val newAcc = sc.register(acc, "evenAcc")
// in Scala
acc.value // 0
flights.foreach(flight_row => acc.add(flight_row.count))
acc.value // 31390
```

如果你主要是 Python 用户，则还可以通过定义AccumulatorParam的子类来创建自定义累加器，形式和上面例子类似。

小结

本章中，我们介绍了分布式变量，在优化或调试中它非常有用。在第15章中，我们介绍Spark如何在集群上运行，以便更好地理解分布式变量在什么时候会对你有所帮助。

第Ⅳ部分

生产与应用

第15章

Spark如何在集群上运行

在此之前，我们主要介绍了Spark的编程接口，我们也讨论了结构化API如何进行逻辑运算，如何将其分解为逻辑计划，如何进一步将其转换为对弹性分布式数据集（RDD）的操作和实际在集群上执行的物理计划。本章重点介绍Spark在集群上执行代码的全过程，我们的介绍不依赖于实现，既与你使用的集群管理器无关，也与你运行的代码无关，所有的Spark代码都以相同的方式运行。

本章涵盖以下几个关键主题：

- Spark应用程序的体系结构和组件。
- Spark应用程序（内部和外部）的生命周期。
- 重要的底层执行属性，例如流水线处理。
- 运行一个Spark应用程序都需要什么？

我们下面首先介绍Spark应用程序的体系结构。

Spark应用程序的体系结构

在第2章，我们讨论了Spark应用程序的一些高级模块，我们现在来回顾一下：

Spark驱动器

Spark驱动器是控制你应用程序的进程。它负责控制整个Spark应用程序的执行并且维护着Spark集群的状态，即执行器的任务和状态，它必须与集群管理器交互

才能获得物理资源并启动执行器。简而言之，它只是一个物理机器上的一个进程，负责维护群集上运行的应用程序的状态。

Spark执行器

Spark执行器也是一个进程，它负责执行由Spark驱动器分配的任务。执行器的核心功能是：完成驱动器分配的任务，运行它们，并报告其状态（成功或失败）和执行结果。每个Spark应用程序都有自己的执行器进程。

集群管理器

Spark驱动器和执行器并不是孤立存在的，集群管理器会将他们联系起来，集群管理器负责维护一组运行Spark应用程序的机器。集群管理器也拥有自己的“驱动器”（即master）和worker的抽象，核心区别在于集群管理器管理的是物理机器，而不是进程。 图15-1展示了一个基本的集群配置，图左侧的机器是集群管理器的驱动节点（即master），圆圈表示在每个物理节点上运行的进程，它们负责管理每个物理节点。因为目前还没有运行Spark应用程序，所以这些进程只是集群管理器的进程，并不是运行Spark应用程序的驱动器和执行器。

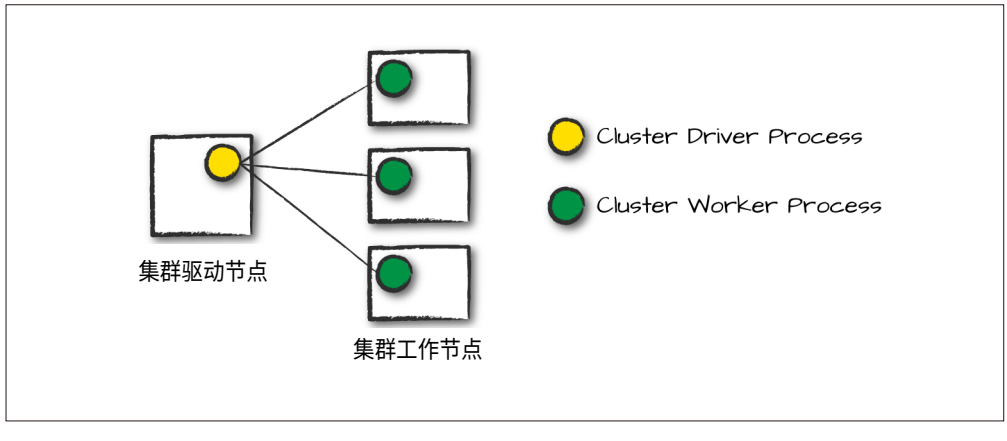


图15-1：集群的驱动节点master和工作节点worker（还没有Spark应用程序在集群上运行）

当实际运行Spark应用程序时，我们会从集群管理器那里请求资源来运行它。根据应用程序的配置，我们可能获得一个运行Spark驱动器的机器资源，或者可能获得的是我们Spark执行器的计算资源。在Spark应用程序执行过程中，集群管理器将负责管理和运行执行应用程序的底层机器。

Spark目前支持三个集群管理器：一个简单的内置独立集群管理器，Apache Mesos和Hadoop YARN。但是，支持的集群管理器可能会越来越多，因此请务必查看集群管理

器的文档以了解最新的更新，这会让你有机会使用你最喜欢的集群管理器。

现在我们已经介绍了应用程序的基本组件，让我们来看看在运行应用程序时需要做的第一个选择：选择执行模式。

执行模式

当在运行应用程序之前，通过选择执行模式你将能够确定计算资源的物理位置。你有三种模式可供选择：

- 集群模式。
- 客户端模式。
- 本地模式。

我们将使用图15-1作为示例来详细介绍其中的每一个细节。在下面的章节中，实线框矩形表示Spark驱动器，而虚线框句型则表示执行器。

集群模式

集群模式可能是运行Spark应用程序的最常见方式。在集群模式下，用户将预编译的JAR包，Python脚本或R语言脚本提交给集群管理器。除执行器进程外，集群管理器还会在集群内的某个工作节点上启动驱动器进程，这意味着集群管理器负责维护所有与Spark应用程序相关的进程。在图15-2展示的情况中，集群管理器将Spark驱动器放置在一个工作节点上，并将Spark执行器放在其他工作节点上。

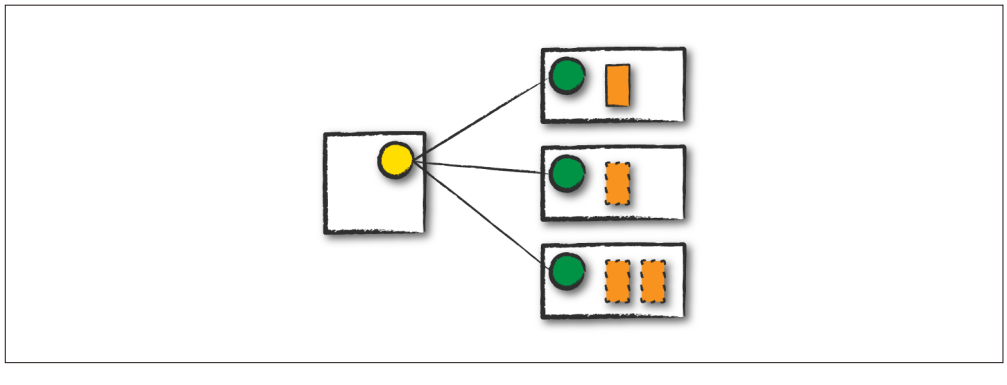


图15-2：Spark的集群模式

客户端模式

客户端模式与集群模式几乎相同，只是Spark驱动器保留在提交应用程序的客户端机器上。这意味着客户端机器负责维护Spark驱动器进程，并且集群管理器维护执行器进程。在图15-3中，我们使用一台集群外的机器上运行Spark应用程序，这些机器通常被称为网关机器（gateway machines）或边缘节点（edge nodes）。在图15-3中，你可以看到驱动器正在集群外部的计算机上运行，但工作节点位于集群中的 计算机上。

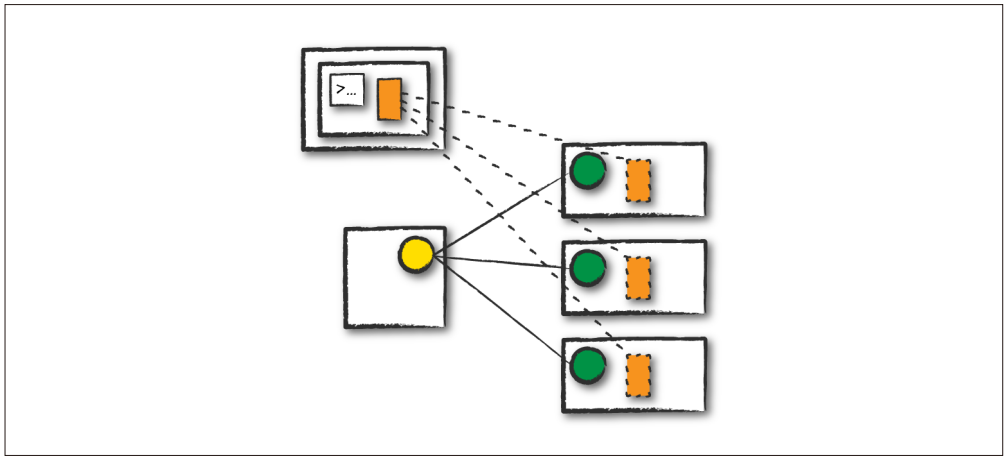


图15-3：Spark的客户端模式

本地模式

本地模式与前两种模式有很大不同：它在一台机器上运行整个Spark应用程序。它通过单机上的线程实现并行性。在本地模式上运行Spark是学习Spark的常用方法，也是测试应用程序或进行迭代本地开发的常用方法，但是我们不建议使用本地模式运行生产级别的应用程序。

Spark应用程序的生命周期（Spark外部）

我们已经介绍了Spark应用程序的相关概念和名词，现在将从实际Spark代码的“外部”讨论Spark应用程序的整个生命周期，我们将通过`spark-submit`（请见第3章）运行一个应用程序示例。我们假设一个集群已经运行了四个节点，包括一个驱动节点（不是Spark驱动器，而是集群管理器的驱动节点）和三个工作节点。本节使用之前介绍的词汇从初始化到程序退出一步一步地介绍Spark应用程序的生命周期。

本章仍然使用之前约定的记号来介绍，另外，我们现在用直线来代表网络通信的线路，较深的箭头表示Spark或Spark相关进程的通信，而虚线表示其他的普通通信（如用于集群管理的通信）。

客户请求

第一步是提交一个应用程序，这是一个编译好的JAR包或者库。此时，你正在本地计算机上执行代码，并且将向集群管理器驱动节点发出请求（图15-4所示红线）。在这里，我们仅会为Spark驱动器进程显式地请求资源，假设集群管理器接受请求并将驱动器放置到集群中的一个物理节点上（右侧黄线），之后提交原始作业的客户端进程退出（左侧黄线），应用程序开始在群集上运行。

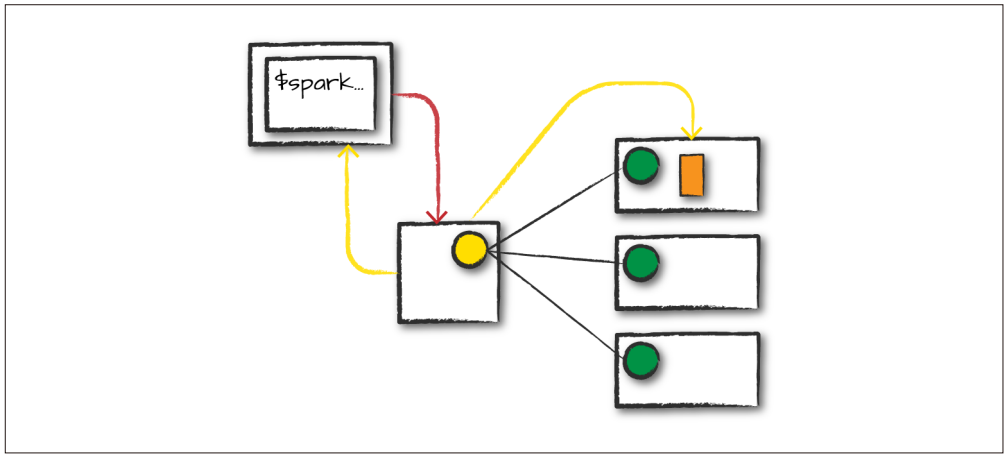


图15-4：为驱动器请求资源

为此，你需要在终端中运行以下命令：

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode cluster \
  --conf <key>=<value> \
  ···  # other options
  <application-jar> \
  [application-arguments]
```

启动

现在驱动器进程已经被放到集群上了，它开始执行用户代码（见图15-5），此代码必须包含一个初始化Spark集群（如驱动器和若干执行器）的SparkSession，SparkSession随后将与集群管理器驱动节点通信（红线），要求它在集群上启动Spark执行器，集群管理器随后在集群工作节点上启动执行器（黄线），执行器的数量及其相关配置由用户通过最开始spark-submit调用中的命令行参数设置。

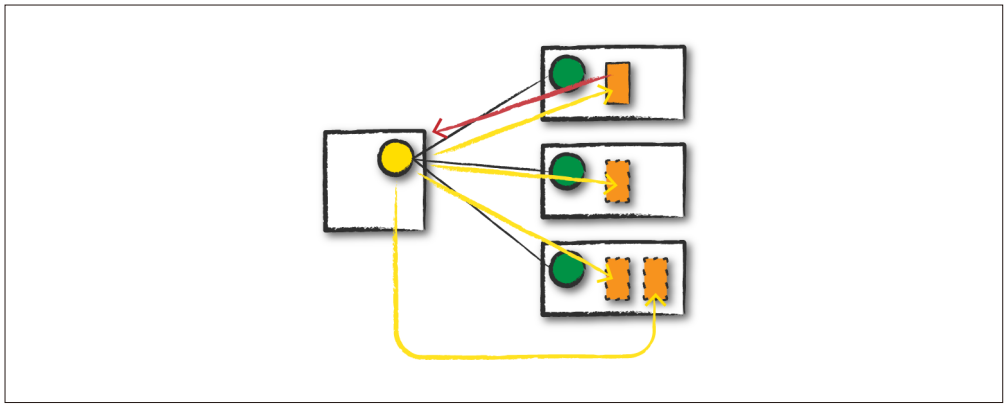


图15-5：启动Spark应用程序

假如一切顺利的话，集群管理器就会启动Spark执行器，并将程序执行位置等相关信息发送给Spark驱动器。在所有程序都正确关联之后，正如你期待的那样，我们就成功构建了一个“Spark集群”。

执行

现在我们有了一个“Spark集群”，Spark就可以开始顺利地执行代码了，如图15-6所示。集群的驱动节点和工作节点相互通信、执行代码、和移动数据，驱动节点将任务安排到每个工作节点上，每个工作节点回应给驱动节点这些任务的执行状态，也可能回复启动成功或启动失败等（我们将会在后面章节详细介绍这些细节）。

完成

Spark应用程序完成后，Spark驱动器会以成功或失败的状态退出（见图15-7），然后集群管理器会为该Spark驱动器关闭Spark集群中的Spark执行器。此时，你可以向集群管理器询问来获知Spark应用程序是成功退出还是失败退出。

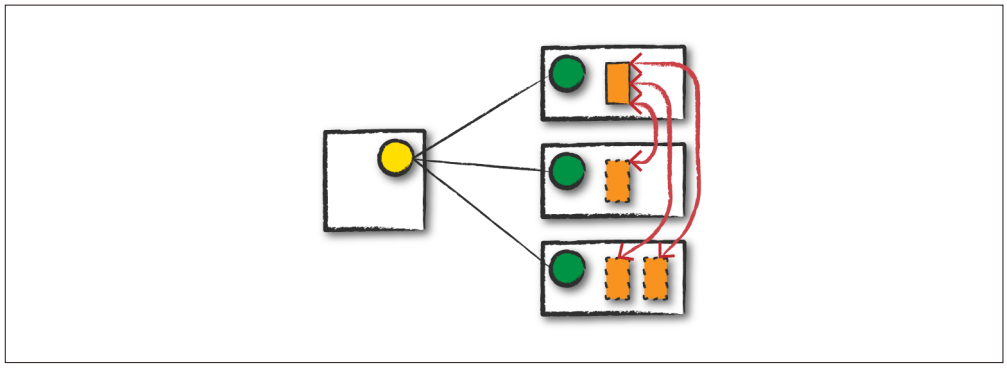


图15-6：应用程序的执行

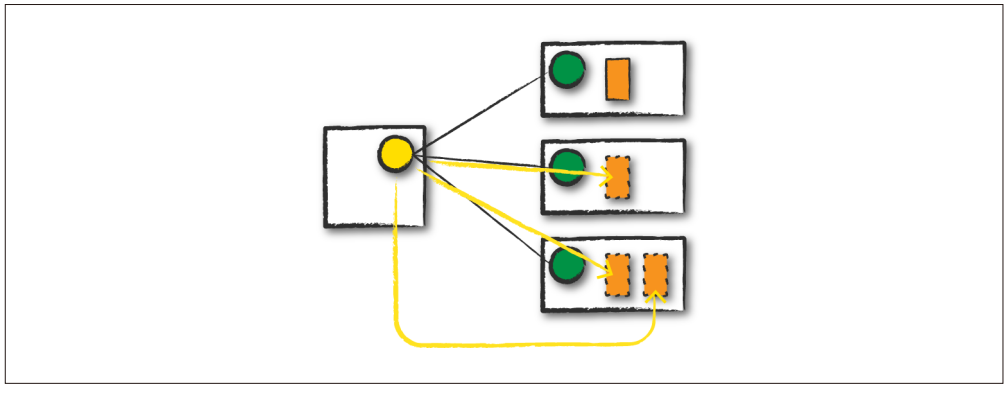


图15-7：关闭应用程序

Spark应用程序的生命周期（Spark内部）

我们在用户代码之外（支持Spark运行的基础架构）介绍了Spark应用程序的生命周期，但是了解Spark应用程序在运行时的内部执行流程更为重要，这也就是“用户代码”（你编写的定义Spark应用程序的实际代码）。每个应用程序由一个或多个Spark作业组成，应用程序内的一系列Spark作业是串行执行的（除非你使用多线程并行启动多个作业）。

SparkSession

任何Spark应用程序的第一步都是创建一个SparkSession。在交互模式中，通常已经为你预先创建了，但在应用程序中你必须自己创建。

一些老旧的代码可能会使用new SparkContext这种方法创建，但是应该尽量避免使用

这种方法，而是推荐使用SparkSession的构建器方法，该方法可以更稳定地实例化Spark和SQL Context，并确保没有多线程切换导致的上下文冲突，因为可能有多个库试图在相同的Spark应用程序中创建会话：

```
// 采用Scala语言创建SparkSession
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().appName("Databricks Spark Example")
  .config("spark.sql.warehouse.dir", "/user/hive/warehouse")
  .getOrCreate()

# 采用Python语言创建SparkSession
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local").appName("Word Count")\
    .config("spark.some.config.option", "some-value")\
    .getOrCreate()
```

在创建SparkSession后，你就应该可以运行你的Spark代码了。通过SparkSession，你可以相应地访问所有低级的和老旧的Spark功能和配置。请注意，SparkSession类是在Spark 2.X版本后才支持的，你可能会发现较旧的代码会直接为结构化API创建SparkContext和SQLContext。

SparkContext

SparkSession中的SparkContext对象代表与Spark集群的连接，可以通过它与一些Spark的低级API（如RDD）进行通信，在较早的示例和文档中，它通常以变量名sc存储。通过SparkContext，你可以创建RDD、累加器和广播变量，并且可以在集群上运行代码。

大多数情况下，你不需要显式初始化SparkContext，你应该可以通过SparkSession来访问它。如果你想要，一般来说你可以通过getOrCreate方法来创建它：

```
// in Scala
import org.apache.spark.SparkContext
val sc = SparkContext.getOrCreate()
```

SparkSession，SQLContext和HiveContext

在早期的Spark版本中，SQLContext和HiveContext提供了使用DataFrame和Spark SQL的功能，并且通常在示例、文档和遗留代码中以变量名sqlContext存储。曾经在Spark 1.X版本中有两个有效的上下文类，SparkContext和SQLContext，它们各自负责不同的功能。前者侧重于更精细地控制Spark的核心抽象，而后者侧重于Spark SQL等更高级别的工具。在Spark 2.X版本中，社区将这两个API组合成了我们今天所使用的SparkSession。但是，这两个API仍然存在，你可以通过SparkSession访问它们。需要注意的是，你应该永远不需要使用SQLContext，并尽量避免使用SparkContext。

初始化SparkSession之后就可以执行一些代码了。正如我们在前面的章节中所介绍的，所有Spark代码都编译为RDD。因此，在下一节中，我们将使用一些逻辑指令（一个DataFrame作业）并一步一步地介绍执行过程。

逻辑指令

正如你在本书开头所看到的，Spark代码基本上由转换和动作组成，无论是SQL查询、低级的RDD操作，还是机器学习算法，如何构建这些都取决于你。了解如何使用DataFrame之类的声明式指令并将其转换为物理执行计划，是理解Spark如何在群集上运行的重要一步。在本节中，请务必在新环境（一个新的Spark shell）中执行操作，以跟踪作业、阶段和任务编号。

逻辑指令到物理执行

我们在第Ⅱ部分中简单介绍了这一点，但值得重申一下，这样你可以更好地理解Spark如何实际在集群上运行你的代码。我们将逐行阅读更多的代码，解释背后的执行过程，以便你更好地理解Spark应用程序。在后面的章节中，当我们讨论监控时，我们将通过Spark UI更详细地跟踪Spark作业。而在这个例子中，我们将用相对简单的方法，使用一个简单的DataFrame执行三步操作：①重新分区；②执行逐个值的操作；③执行聚合操作并收集最终结果。

这段代码是用Python编写并运行在Spark 2.2上的（因为使用Scala你将得到相同的结果，所以我们省略了Scala的代码）。作业数量可能不会发生剧烈变化，但可能对底层优化产生作用并改变物理执行策略。

```
# in Python
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")

step4.collect() # 2500000000000
```

当你运行这段代码时，我们可以看到你的动作触发了一个完整的Spark作业。让我们来看一下解释计划以理解实际的物理执行计划。我们可以在Spark UI中的SQL选项卡上（在我们实际运行查询之后）查看这些信息：

```
step4.explain()
```

```
== Physical Plan ==
*HashAggregate(keys=[], functions=[sum(id#15L)])
+- Exchange SinglePartition
   +- *HashAggregate(keys=[], functions=[partial_sum(id#15L)])
      +- *Project [id#15L]
         +- *SortMergeJoin [id#15L], [id#10L], Inner
            :- *Sort [id#15L ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(id#15L, 200)
            :     +- *Project [(id#7L * 5) AS id#15L]
            :        +- Exchange RoundRobinPartitioning(5)
            :           +- *Range (2, 100000000, step=2, splits=8)
            +- *Sort [id#10L ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(id#10L, 200)
                  +- Exchange RoundRobinPartitioning(6)
                     +- *Range (2, 100000000, step=4, splits=8)
```

当你调用`collect`（或任何动作）时，你将执行Spark作业，它们由阶段和任务组成。如果你正在本地计算机上运行以查看Spark UI，请在浏览器上访问localhost：4040。随着我们深入细节层次，我们将继续关注“作业”标签，然后介绍阶段和任务。

Spark作业

一般来说，一个动作应该触发一个Spark作业，调用动作总是会返回结果，每个作业被分解成一系列阶段，其数量取决于需要进行多少次shuffle操作。

这项工作分为以下几个阶段和任务：

- 第一阶段，有8个任务。
- 第二阶段，有8个任务。
- 第三阶段，有6任务。
- 第四阶段，有5个任务。
- 第五阶段，有200个任务。
- 第六阶段，有1个任务。

希望你至少对我们如何得到这些数字感到好奇，这样更有利于我们花更多的时间来帮助你更好地理解执行过程。

阶段

Spark中的阶段（stage）代表可以一起执行的任务组，用以在多台机器上执行相同的操作。一般来说，Spark会尝试将尽可能多的工作（即作业内部尽可能多的转换操

作）加入同一个阶段，但引擎在shuffle操作之后启动新的阶段。一次shuffle操作意味着一次对数据的物理重分区，例如对DataFrame进行排序，或对从文件中加载的数据按key进行分组（这要求将具有相同key的记录发送到同一节点），这种重分区需要跨执行器的协调来移动数据。Spark在每次shuffle之后开始一个新阶段，并按照顺序执行各阶段以计算最终结果。

在我们前面看到的作业中，前两个阶段是执行的`range`操作，它将创建你的DataFrame。默认情况下，当你使用range创建DataFrame时，它有8个分区。下一步是重新分区，通过对数据的shuffle操作来改变分区的数量，这些DataFrame被shuffle成6个分区和5个分区，对应于阶段3和阶段4中的任务数量。

阶段3和阶段4在每个DataFrame上执行，并且阶段的末尾表示连接操作（需要shuffle过程）。这时，我们有了200个任务，这是因为Spark SQL的配置，`spark.sql.shuffle.partitions`的默认值是200，这意味着在执行过程中执行一个shuffle操作时，默认输出200个shuffle分区。你可以通过改变这个值来改变输出分区的数量。

我们将在第19章中详细介绍分区的数量，因为它是一个非常重要的参数。它应该根据集群中的核的数量设置，以确保高效执行。以下是设置它的方法：

```
spark.conf.set("spark.sql.shuffle.partitions", 50)
```

对于分区数量的设置，一个经验法则是分区数量应该大于集群上执行器的数量，这可能取决于工作负载相关的多个因素。如果你在本地计算机上运行代码，则应该将分区数量设置得较低，因为你的本地计算机不太可能并行执行这些任务。对于可能有更多执行核心可以使用的集群来说，就应该设置的更多。无论分区数量设置成多少，整个阶段都是并行执行的，系统可以分别对这些分区并行执行聚合操作，将这些局部结果发送到一个汇总节点，然后再在这些局部结果上执行最终的聚合操作获得最终结果，再把该结果返回给驱动器。在本书的这一部分的介绍中，我们会多次看到分区数量这个配置。

任务

Spark中的阶段由若干任务（task）组成，每个任务都对应于一组数据和一组将在单个执行器上运行的转换操作。如果数据集中只有一个大分区，我们将只有1个任务；如果有1000个小分区，我们将有1000个可以并行执行的任务。任务是应用于每个数据单元（分区）的计算单位，将数据划分为更多分区意味着可以并行执行更多分区。虽然

可以通过增加分区数量来增加并行性，但这不是万能的，只是可以通过这一点来做一些简单的优化。

执行细节

Spark中任务和阶段的一些重要执行细节也值得关注。第一个执行细节是Spark会自动的以流水线的方式一并完成连续的阶段和任务，例如map操作接着另一个map操作。另外一个执行细节是，对于所有的shuffle操作，Spark会将数据写入持久化存储（例如磁盘），并可以在多个作业中重复使用它。我们将依次讨论这些概念，当你通过Spark UI检查应用程序时也会看到这些细节。

流水线执行

使Spark成为一个著名的“内存计算工具”的很重要一点就是，与之前的工具不同（例如MapReduce），Spark在将数据写入内存或磁盘之前执行尽可能多的操作。Spark执行的关键优化之一是流水线，它在RDD级别或其以下级别上执行。通过流水线技术，一系列有数据依赖关系的操作，如果不需要任何跨节点的数据移动，就可以将这一系列操作合并为一个单独的任务阶段。例如，如果你编写一个基于RDD的程序，首先执行map操作，然后是filter操作，然后接着另一个map操作，这些操作不需要在节点间移动数据，所以就可以将它们合并为一个单一阶段的任务，即读取每个输入记录，然后经由第一个map操作，再执行filter操作，然后如果需要的话再执行map操作。通过流水线优化的计算要比每步完成后将中间结果写入内存或磁盘要快得多。对于执行select，filter和select序列操作的DataFrame或SQL计算，也会同样地执行流水线操作。

实际上，当你编写应用程序时，流水线优化对你来说是透明的，Spark引擎会自动的完成这项工作。但是如果你通过Spark UI或其日志文件检查你的应用程序，你将看到Spark系统将多个RDD或DataFrame操作通过流水线优化合并为一个执行阶段。

shuffle数据持久化

你会偶尔观察到的第二个属性是shuffle数据的持久化。当Spark需要运行某些需要跨节点移动数据的操作时，例如按键约减操作（即reduce-by-key操作，其中每个键对应的输入数据需要先从多个节点获取并合并在一起），处理引擎不再执行流水线操作，而是执行跨网络的shuffle操作。在Spark执行shuffle操作时，总是首先让前一阶段的“源”任务（发送数据的那些任务）将要发送的数据写入到本地磁盘的shuffle文

件上，然后下一阶段执行按键分组和约减的任务将从每个shuffle文件中获取相应的记录并执行某些计算任务（例如，获取并处理特定键范围的数据）。将shuffle文件持久化到磁盘上允许Spark稍晚些执行reduce阶段的某些任务（例如，如果没有足够多的执行器同时执行分配任务，由于数据已经持久化到磁盘上，便可以稍晚些执行某些任务），另外在错误发生时，也允许计算引擎仅重新执行reduce任务而不必重新启动所有的输入任务。

shuffle操作数据持久化有一个附带作用，在已经执行了shuffle操作的数据上运行新的作业并不会重新运行shuffle操作的“源”一侧的任务（即生产shuffle数据的任务）。由于shuffle文件早已写入磁盘，因此Spark知道可以直接使用这些已经生成好的shuffle文件来运行作业的后一阶段，而不需要重做之前的阶段。在Spark UI和日志中，你将看到标记为“skipped”的预shuffle阶段。这种自动优化可以节省在同一数据上运行多个作业所花费的时间，当然，如果想要获得更好的性能，你也可以使用DataFrame或RDD的cache方法自己设置缓存，这样你可以精确控制哪些数据需要保存，并且控制保存到哪里。通过对聚合后的数据运行一些Spark动作操作，并在Spark UI中监视这些执行过程，你将很快熟悉这种执行机制。

小结

在本章中，我们讨论了Spark应用程序在如何在集群上执行的，包括集群如何实际地执行我们的代码，以及在此过程中Spark应用程序中到底发生了什么。此时，你应该很了解Spark应用程序内部和外部发生的事情了，对Spark运行机制的理解将有助于应用程序的调试，第16章将讨论编写Spark应用程序以及此过程中的一些注意事项。

第16章

开发Spark应用程序

在第15章，我们介绍了Spark如何在集群上运行你的代码，在这一章我们将展示如何开发独立的Spark应用程序并将其部署到集群上。我们将使用一个简单的模板，该模板分享了一些关于如何构建应用程序的简单提示，包括设置构建工具和单元测试。本书的代码库提供了该模板，但是这个模板并不是真的必要，因为从零开始编写应用程序并不困难，但它确实有帮助。接下来让我们开始编写我们的第一个应用程序。

编写Spark应用程序

Spark应用程序包含两个部分：一个是Spark集群，另一个是你的代码。在本例中，集群被设置为本地模式，应用程序是预定义的，接下来我们来看看基于每个语言的应用程序。

一个简单的基于Scala的应用程序

Scala可以说是Spark的“母语”，因为Spark系统本身就是Scala编写的，使用Scala自然而然地成为编写应用程序的好方法，这样的话编写Spark应用程序和编写Scala应用程序没有什么不同。

Scala可能看起来很吓人，这取决于你的背景知识，即使是为了更好地理解Spark，Scala也是值得学习的。另外，你不需要学习语言的所有方方面面，仅从基础知识开始，你将会发现使用Scala编写应用程序非常容易，学习使用Scala也会让你接触更多知识。稍加练习，通过Spark的代码库进行代码级跟踪也不困难。

你可以使用sbt或Apache Maven来构建应用程序，这是两个基于Java虚拟机（JVM）的程序构建工具。与任何程序构建工具一样，它们有各自的使用特点。使用sbt构建可能是更简单些，你可以在sbt官方网站上下载sbt，安装并了解sbt。你也可以从Maven的官方网站上下载Maven，然后安装Maven。

使用sbt构建Scala应用程序，我们需要修改*build.sbt*文件来配置软件包依赖信息。在*build.sbt*文件里面，包括以下几个关键的信息需要设置：

- 项目元数据（包名称，包版本信息等）。
- 在哪里解决依赖关系。
- 构建库所需要的包依赖关系。

还有许多其他的选项可以指定，但是本书不做详细介绍（你可以从官方网站sbt文档中找到关于这方面的信息）。也有一些介绍sbt技术的书籍，如果遇到较深入的问题你也可以参考。以下是Scala的*built.sbt*文件示例（以及我们在模板中包含的样例）。注意我们必须指定Scala版本以及Spark版本：

```
name := "example"
organization := "com.databricks"
version := "0.1-SNAPSHOT"
scalaVersion := "2.11.8"

// Spark相关信息
val sparkVersion = "2.2.0"

// 包含Spark软件包
resolvers += "bintray-spark-packages" at
  "https://dl.bintray.com/spark-packages/maven/"

resolvers += "Typesafe Simple Repository" at
  "http://repo.typesafe.com/typesafe/simple/maven-releases/"

resolvers += "MavenRepository" at
  "https://mvnrepository.com/"

libraryDependencies ++= Seq(
  // spark内核
  "org.apache.spark" %% "spark-core" % sparkVersion,
  "org.apache.spark" %% "spark-sql" % sparkVersion,
// 这里忽略文件其余部分
)
```

现在已经定义了构建文件，可以将代码添加到我们的项目中，我们将使用标准Scala的项目结构，这可以在sbt参考手册中找到（Maven项目具有相同的目录结构）：

```
src/
```

```
main/
  resources/
     <files to include in main jar here>
  scala/
     <main Scala sources>
  java/
     <main Java sources>
test/
  resources
     <files to include in test jar here>
  scala/
     <test Scala sources>
  java/
     <test Java sources>
```

我们把源代码放在Scala和Java目录中，在源代码文件中加入如下内容，包括初始化SparkSession，运行应用程序，然后退出：

```
object DataFrameExample extends Serializable {
  def main(args: Array[String]) = {

    val pathToDataFolder = args(0)

    // 创建 SparkSession
    // 显式配置
    val spark = SparkSession.builder().appName("Spark Example")
      .config("spark.sql.warehouse.dir", "/user/hive/warehouse")
      .getOrCreate()

    // 注册用户自定义函数
    spark.udf.register("myUDF", someUDF(_:String):String)
    val df = spark.read.json(pathToDataFolder + "data.json")
    val manipulated = df.groupBy(expr("myUDF(group)")).sum().collect()
     .foreach(x => println(x))

  }
}
```

注意我们需要定义一个main类，当我们使用spark-submit命令行将它提交给集群时，它才可以执行。

现在我们已经建立了项目并添加了一些代码，现在我们来构建它。我们可以使用sbt assemble命令来构建一个包含所有依赖项的“uber-jar”或“fat-jar”。对于某些部署来说，这可能是简单的方式，但是在某些其他情况下可能造成混乱（尤其是需要解决依赖冲突）。更轻量级的方法是运行sbt package，它将把所有依赖关系收集到目标文件夹中，但不会将它们全部打包到一个大的JAR中。

运行应用程序

目标文件夹包含JAR文件，我们可以将这个JAR包作为spark-submmit的参数以在集群上执行应用程序。在构建完Scala包之后，你可以使用下面的代码在本地机器上提交作业（该代码片段使用了$SPARK_HOME变量，你可以将$SPARK_HOME替换为你下载的Spark的文件夹位置）：

```
$SPARK_HOME/bin/spark-submit \
   --class com.databricks.example.DataFrameExample \
   --master local \
   target/scala-2.11/example_2.11-0.1-SNAPSHOT.jar "hello"
```

编写Python应用程序

编写PySpark应用程序与普通的Python应用程序没有区别，与编写命令行应用程序非常相似。Spark没有构建的概念，只有Python脚本，所以要运行应用程序，只需在集群上执行脚本即可。

为了便于代码重用，通常将多个Python文件打包成包含Spark代码的egg文件或ZIP文件。 为了包含这些文件，可以通过spark-submit的--py-files参数来添加要与应用程序一起分发的*.py*，*.zip*或*.egg*文件。

在运行代码的时候，你需要在Python中创建一个“Scala/Java main class”，指定一个特定的脚本作为构建SparkSession的可执行脚本，这也是spark-submit命令所需要的一个主要参数：

```
# in Python
from __future__ import print_function
if __name__ == '__main__':
    from pyspark.sql import SparkSession
    spark = SparkSession.builder \
        .master("local") \
        .appName("Word Count") \
        .config("spark.some.config.option", "some-value") \
        .getOrCreate()

    print(spark.range(5000).where("id > 500").selectExpr("sum(id)").collect())
```

通过这种方法你将得到一个SparkSession，可以通过它提交你的应用程序。 最好是在运行时再实例化SparkSession，而不是在每个Python类中都实例化它。

在Python中开发时，建议使用pip指定PySpark作为依赖项。你可以运行命令pip install pyspark来首先安装它，然后像使用其他Python包的方式来使用它，随后可以使用各种编辑器的自动补全功能。这是Spark 2.2版本中的全新功能，还有些不完善，

因此可能再需要1~2个版本后才能完全适合实际生产。但Python在Spark社区非常流行，它肯定会成为Spark未来的基石。

运行应用程序

在编写代码之后，就可以提交它并在集群上执行了（我们执行与项目模板中相同的代码）。你需要调用spark-submit执行你的应用程序：

```
$SPARK_HOME/bin/spark-submit --master local pyspark_template/main.py
```

编写Java应用程序

编写Java Spark应用程序就像编写Scala应用程序一样，关键区别就是如何指定依赖关系。

这个例子假设你使用Maven来指定依赖关系，在这种情况下，你将使用以下格式。在Maven中，你必须添加Spark Packages存储库，以便从这些位置获取依赖关系：

```
<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
        <version>2.1.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.1.0</version>
    </dependency>
    <dependency>
        <groupId>graphframes</groupId>
        <artifactId>graphframes</artifactId>
        <version>0.4.0-spark2.1-s_2.11</version>
    </dependency>
</dependencies>
<repositories>
    <!-- list of other repositories -->
    <repository>
        <id>SparkPackagesRepo</id>
        <url>http://dl.bintray.com/spark-packages/maven</url>
    </repository>
</repositories>
```

自然地，你需要遵循与Scala项目版本相同的目录结构（因为它们都符合Maven规范），然后，我们只需遵循相关的Java示例来实际构建和执行代码。现在我们创建一个简单的例子来指定一个main类（本章后面会给出更多相关内容）：

```
import org.apache.spark.sql.SparkSession;
```

```
public class SimpleExample {
    public static void main(String[] args) {
        SparkSession spark = SparkSession
                .builder()
                .getOrCreate();
        spark.range(1, 2000).count();
    }
}
```

然后通过使用mvn package命令来打包（你需要安装Maven才能执行此命令）。

运行应用程序

这个操作和运行Scala应用程序（或者Python应用程序）完全相同，只需执行spark-submit命令：

```
$SPARK_HOME/bin/spark-submit \
   --class com.databricks.example.SimpleExample \
   --master local \
   target/spark-example-0.1-SNAPSHOT.jar "hello"
```

测试Spark应用程序

你现在知道编写和运行Spark应用程序的过程了，接下来讨论一个不那么令人兴奋但仍然非常重要的问题：测试。测试Spark应用程序需要你在编写应用程序时应牢记的几个关键原则和策略。

战略原则

测试数据流通过程和测试Spark应用程序与实际编写它们一样重要，因为你希望确保它们能够适应未来数据、逻辑和输出的变化。在本节中，我们将首先讨论在典型Spark应用程序中应该测试的内容，然后讨论如何编写代码以进行简单的测试。

对输入数据的适应性

适应不同类型的输入数据，对于编写数据处理程序非常重要。因为业务需求的变化，输入数据也可能会变化。因此，你的Spark应用程序和数据处理管道应该对输入数据的某种程度变化具有适应性，或者能够确保以优雅和灵活的方式处理这些因变化导致的异常。大多数情况下，这就意味着要编写处理不同输入边界情况的测试用例，并确保能够在发现重大错误时做出反应。

对业务逻辑的适应性和演变

业务逻辑可能会随输入数据的变化而变化。更重要的是，你要确保从原始数据中推断

出的内容正是你想要推断的内容。这意味着你需要使用真实数据进行各种逻辑测试，以确保得到的确实是你想要的结果。有一点需要注意的是，你不需要编写一大堆测试Spark功能的“Spark单元测试”，而是需要测试你的业务逻辑，并确保你设置的复杂业务流程实际上正在做应该做的事情。

对输出模式和原子性的适应性

假设你的应用程序已经准备好了应对各种输入数据，并且你的业务逻辑也已经经过了充分测试，你现在需要确保你的输出模式符合你的期望，你将需要正确地解析输出模式。通常情况下，数据不会简单地转储到某个位置，也不会再次读取，而是可能会输出给其他Spark处理程序。出于这个原因，你需要确保下游消费者了解输出数据的“状态”，比如输出数据更新的频率以及输出数据是否“完整”（例如是否有迟到数据），以及不会在最后一刻对数据进行修正。

所有上述问题都是你在构建数据管道时应该考虑的原则（无论你是否使用Spark），这些原则对于成功构建你所期望的系统非常重要。

战术指导

尽管战略思维很重要，但我们还是要详细介绍下你可以使用的一些具体策略，以使你的应用程序易于测试。最好的方法是通过采用适当的单元测试来验证你的业务逻辑是否正确，并确保你能够适应不断变化的输入数据，或者是通过结构化的方法使输入模式的演化在未来会更易处理。如何做到这一点在很大程度上取决于开发人员，因为它会根据业务领域和专业知识的不同而有所不同。

管理SparkSessions

使用JUnit或ScalaTest等单元测试框架测试Spark代码相对容易，因为Spark的本地模式只是创建一个本地模式的SparkSession并加载到测试框架中运行。但是，为了使这项工作顺利进行，在管理代码中的SparkSessions时，应该尽可能地尝试执行依赖注入，也就是说，只需将SparkSession初始化一次，然后在运行时将其传递给相关的函数和类，以便在测试期间可以轻易替换，这使得在单元测试中使用SparkSession哑元对象测试每个单独的函数变得更加容易。

使用哪种Spark API？

Spark提供了多种API选择，包括SQL、DataFrame和Dataset，其中每一种对应用程序的可维护性和可测试性都会产生不同的影响。诚恳地说，选择何种API取决于开发团

队和他们的需求：一些团队和项目需要限制比较少的的SQL和DataFrame API来提高开发速度，而另一些团队和项目则希望使用类型安全的Dataset API或RDD。

通常，无论使用哪种API，我们都建议记录和测试每个函数的输入和输出类型。类型安全的API会自动为你执行强制类型转换，以便于其他代码在其上构建。如果你的团队倾向于使用DataFrame或SQL，那么花一些时间来记录和测试每个函数返回的内容以及它接受的输入类型，以避免以后出现意外，就如同任何动态类型的编程语言一样。尽管较低级别的RDD API也是静态类型的，但我们建议只有在你需要低级特性（如Dataset API中没有数据分块的低级特性）时才使用它，其实真正需要低级特性的情况并不是很常见。Dataset API允许更多的性能优化，并且未来可能会提供更多的性能优化技术。

类似的考虑因素也包括应用程序所使用的编程语言，选择语言这个问题对于每个开发团队来说都没有统一的正确答案。根据你的具体需求，每种语言都会有不同的好处。我们通常推荐使用Scala和Java等静态类型语言来处理大型应用程序，或者来处理那些你希望能够完全控制性能优化的应用程序。但在其他情况下，Python和R可能会更好，例如你需要使用这些语言的其他库。对于测试来说，你编写的Spark代码需要在各种语言的标准单元测试框架中都通过测试。

连接到单元测试框架

为了对代码进行单元测试，我们建议使用标准测试框架（例如JUnit或ScalaTest），并设置你的测试框架来为每次测试创建和清理SparkSession，不同的测试框架提供了不同的机制来执行此操作，例如“之前”（before）和“之后”（after）方法。我们在本章的应用程序模板中包含了一些单元测试代码示例。

连接到数据源

应该尽量避免测试代码连接到实际生产数据源，以便开发人员隔离测试代码，这样可以在数据源发生改变时仍然能够正确地运行代码。实现此目的的一个简单方法是让所有业务逻辑功能的输入都为DataFrame或Dataset，而不是直接连接到数据源。毕竟，无论数据源是什么，后续的代码都会以相同的方式工作。如果你在Spark中使用结构化API，另一种方法是使用不同名称的表，你可以简单地将一些虚拟数据集(例如，从小型文本文件或内存对象加载的数据集)注册为不同的表名，然后让测试代码处理这些数据表。

开发过程

Spark应用程序的开发流程与其他应用程序的开发流程类似。首先，你可能需要一个临时的开发环境，例如一个交互式Notebook环境等，然后在构建关键组件和算法时，需要将它们迁移到一个更加持久稳定的地方，如在版本控制器上维护的库和软件包。我们推荐使用Notebook环境（本书作者就是使用Notebook环境编写本书），因为可以简单地执行实验。还有一些工具，例如Databricks，它们允许你在Notebook上运行生产应用程序。

在本地机器上运行时，使用`spark-shell`及其支持各种语言的版本可能是开发应用程序的最佳选择。通常情况下，命令行（shell）用于交互式应用程序，而`spark-submit`用于Spark群集上运行的生产应用程序。正如我们在本书开头介绍的那样，你可以使用shell以交互式方式运行Spark，包括PySpark，Spark SQL和SparkR都是基于交互式的运行方式。在Spark目录下的bin文件夹中，你会找到启动这些shell的可执行器，比如`spark-shell`（用于Scala），`spark-sql`，`pyspark`和`sparkR`。

完成了应用程序开发，并在创建了程序包或执行脚本后，如果要提交到集群上运行，那么使用`spark-submit`命令是最合适的。

启动应用程序

运行Spark应用程序的最常见方式是通过`spark-submit`。在本章之前，我们就向你展示了如何运行`spark-submit`，你需设置你的启动选项、应用程序JAR包文件或执行脚本、以及相关参数：

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar-or-script> \
  [application-arguments]
```

当你使用`spark-submit`提交Spark作业时，你可以指定是以客户端模式还是以集群模式运行。 但是你应该更倾向于以集群模式运行（或在集群上以客户端模式运行），这样可以利用更多的计算资源，减少运行时间。

提交应用程序时，可以提交*.py*文件或者*.jar*包，然后通过`--py-files`选项指定将后缀名为`.zip`，`.egg`和`.py`的文件添加到搜索路径中。

作为参考，表16-1列出所有可用的spark-submit选项，其中包括针对特定集群管理器的选项，你也可以运行spark-submit -help命令列举所有这些选项。

表16-1：Spark-submit命令选项

Parameter	Description
--master MASTER_URL	指定master节点URL，例如spark：//host：port，mesos：//host：port，yarn，or local
--deploy-mode DEPLOY_MODE	配置是在本地以客户端模式（“client”）还是在一台集群中节点上以集群模式(“cluster”)运行应用程序 (默认使用客户端模式)
--class CLASS_NAME	配置应用程序的入口类 (main函数所在的类，适合 Java / Scala 应用)
--name NAME	配置应用程序的名字
--jars JARS	配置驱动器或者执行器路径上包括的本地jar包，用逗号隔开
--packages	配置驱动器或者执行器路径上包括的Maven依赖包，用逗号隔开。将会首先搜索本地Maven版本库（repo），然后搜索Maven Central及远程版本库（远程repo通过--repositories选项指定）。依赖软件包的格式是groupId：artifactId：version
--exclude-packages	为了避免依赖冲突，配置排除在--packages选项中指定的依赖包，通过逗号隔开，格式是：artifactId
--repositories	配置除了通过--packages指定的，其他的Maven远程依赖库，通过逗号隔开
--py-files PY_FILES	配置Python应用程序需要的.zip、.egg或者.py文件（即放在PYTHONPATH路径上的文件），用逗号隔开
--files FILES	配置在每个执行器工作目录路径下的文件，用逗号隔开
--conf PROP=VALUE	配置Spark属性
--properties-file FILE	配置需要从哪个文件加载额外的属性，默认是conf/spark-defaults.conf
--driver-memory MEM	配置驱动器的内存大小（例如，1000MB，2GB）（默认：1024MB）
--driver-python-options	配置驱动器的Java参数
--driver-library-path	配置驱动器的library path
--driver-class-path	配置驱动器的classpath。注意，通过—jars添加的JAR包已经自动包含在classpath里了

表16-1：Spark-submit命令选项（续）

Parameter	Description
--executor-memory MEM	配置执行器的内存大小（例如，1000MB，2GB）（默认：1024MB）
--proxy-user NAME	配置提交应用程序时的代理用户，在配置了--principal/--keytab选项时，这个配置不生效
--help，-h	显示帮助信息并退出
--verbose，-v	打印额外的debug信息
--version	打印Spark版本号

还有一些特定于部署的配置（请参阅表16-2）。

表16-2：部署相关配置

Cluster Managers	Modes	Conf	Description
Standalone	Cluster	--driver-cores NUM	驱动器的核心数量（默认：1）
Standalone/Mesos	Cluster	--supervise	失败后重新启动驱动器
Standalone/Mesos	Cluster	--kill SUBMISSION_ID	杀掉指定驱动器进程
Standalone/Mesos	Cluster	--status SUBMISSION_ID	获取指定驱动器的状态
Standalone/Mesos	Either	--total-executor-cores NUM	所有执行器的总核心数
Standalone/YARN	Either	--executor-cores NUM1	每个执行器的核心数(默认：YARN 模式下为1，或在standalone模式下worker节点的所有可用核心数)
YARN	Either	--driver-cores NUM	集群模式下的驱动器的核心数（默认：1）
YARN	Either	queue QUEUE_NAME	提交到YARN的队列名（默认：“default”）
YARN	Either	--num-executors NUM	启动的执行器（默认：2）。如果配置了动态分配，那么初始的执行器数量最少为NUM
YARN	Either	--archives ARCHIVES	需要提取到每个执行器工作目录下的archive，用逗号分隔
YARN	Either	--principal PRINCIPAL	当运行安全HDFS时，登录到KDC用到的原则

表16-2：部署相关配置（续）

Cluster Managers	Modes	Conf	Description
YARN	Either	--keytab KEYTAB	包含上面指定principal 的keytab的完整路径，keytab将会通过Distributed Cache被复制到执行应用程序的master节点上，为定期更新登录口令使用

应用程序启动示例

前面已经介绍了一些以本地模式运行的应用程序示例，如何使用上面列出的这些选项配置也非常重要，Spark的examples目录中还包含其他几个示例应用程序，如果你对某些参数配置有疑惑，可以在本地机器上使用SparkPi类作为测试类来尝试一下各参数选项的作用：

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  replace/with/path/to/examples.jar \
  1000
```

下面的代码片段也同样适用于Python，你可以在Spark目录下运行，提交Python应用程序（全部在一个脚本中）给standalone集群管理器，也可以配置与上例中相同的执行进程数量限制等参数：

```
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

你也可以将其更改为以本地模式运行，即将master设置为local或local[*]（在计算机的所有核心上运行），还需要将/path/to/examples.jar对应地更改为你正在运行的Scala 和Spark的对应版本。

配置应用程序

Spark涉及许多配置选项，其中一些我们已经在第15章介绍了，本节将介绍更多细节。除非你特别了解某些配置，否则大多数内容只需要粗略了解。大多数配置选项分为以下几类：

- 应用程序属性。
- 运行时环境。
- shuffle行为。
- Spark UI。
- 压缩和序列化。
- 内存管理。
- 执行行为。
- 网络。
- 调度。
- 动态分配。
- 安全。
- 加密。
- Spark SQL。
- Spark流处理。
- SparkR。

你可以通过三种手段配置Spark系统：

- 通过Spark属性可以控制大多数应用程序参数，可以通过SparkConf对象进行设置。
- Java属性配置。
- 硬编码的配置文件。

在Spark根目录下的*/conf*目录中包含了一些属性配置模板，你可以将这些属性设置为应用程序的硬编码变量，或者在运行时指定它们。你可以使用环境变量配置每个节点的属性，例如可以通过`conf / spark-env.sh`脚本来配置每个节点的IP地址。另外，你可以通过`log4j.properties`配置日志记录属性。

SparkConf

SparkConf管理着我们所有的应用配置，首先需要使用import语句并创建一个SparkConf对象，如以下示例所示。在创建SparkConf对象之后，SparkConf对象将不可以改写：

```
// in Scala
import org.apache.spark.SparkConf
val conf = new SparkConf().setMaster("local[2]").setAppName("DefinitiveGuide")
  .set("some.conf", "to.some.value")

# in Python
from pyspark import SparkConf
conf = SparkConf().setMaster("local[2]").setAppName("DefinitiveGuide")\
  .set("some.conf", "to.some.value")
```

你可以使用SparkConf来配置每个Spark应用程序的Spark属性，这些Spark属性包括如何控制Spark应用程序的运行方式以及如何配置集群。下面的示例将本地集群配置为具有两个线程，并指定了Spark UI中显示的应用程序名称。

你也可以在运行时通过命令行参数配置Spark应用程序，这主要是在使用Spark Shell时会用到，启动Spark Shell将自动为你创建一个基本的Spark应用程序; 例如：

```
./bin/spark-submit --name "DefinitiveGuide" --master local[4] ...
```

值得注意的是，在设置时间属性时，应该使用以下格式：

- 25ms（毫秒）。
- 5s（秒）。
- 10m或10min（分钟）。
- 3h（小时）。
- 5d（天）。
- 1y（年）。

应用程序属性

应用程序属性可以通过spark-submit命令行配置或从代码创建Spark应用程序时配置。他们定义了基本的应用程序元数据以及一些执行特性。表16-3列出了当前Spark版本支持的应用程序属性。

表16-3：应用程序属性

Property name	Default	Meaning
spark.app.name	(none)	应用程序名称，将会出现在用户界面和日志里
spark.driver.cores	1	驱动器进程使用的核心数（仅在集群模式下有效）
spark.driver.maxResultSize	1g	每个Spark动作（例如collect）所允许产生的（包括来自所有分区的）中间结果大小限制，最小设置为1MB，0代表无限制。如果产生的中间结果超过了这个限制，作业将会被强制终止。如果这个限制设置太大可能会导致 OutOfMemoryErrors错误(这也与spark.driver.memory的设置和JVM中Java对象的内存开销有关)，设置一个合理的限制会避免出现OutOfMemoryErrors错误
spark.driver.memory	1g	驱动器进程所允许使用的内存大小（例如，1g，2g）。请注意：在客户端模式下，这个参数不能再你的应用程序中通过SparkConf设置，因为执行驱动器进程的JVM在这个时候已经启动了，所以必须通过 命令行选项--driver-memory或者在配置文件中设置该值
spark.executor.memory	1g	执行器进程所允许使用的内存大小 (例如，2g，8g)
spark.extraListeners	(none)	指定SparkListener的实现类，通过逗号分隔。在初始化SparkContext对象时，这些实现类将会被实例化和被注册，如果SparkListener实现类的某个构造器函数需要一个SparkConf参数，该构造器会被默认调用，否则会调用空参数的构造器，如果找不到以上构造器函数，SparkContext对象的创建将会失败并抛出异常
spark.logConf	FALSE	当SparkContext启动后，在日志中记录有效的SparkConf信息，并用INFO标识
spark.master	(none)	设置集群管理器的URL，请参考符合规范的URL格式
spark.submit.deployMode	(none)	Spark 驱动器的部署模式，即客户端模式（client）或者集群模式（cluster），客户端模式代表在本地执行，而集群模式代表在集群中的某个节点远程执行
spark.log.callerContext	(none)	当运行Yarn/HDFS时，设置应用程序信息写入Yarn的RM日志中或HDFS的监听日志中。应用程序信息长度取决于Hadoop参数hadoop。caller.context.max.size，而且应该精确设置且小于50字符

表16-3：应用程序属性（续）

Property name	Default	Meaning
spark.driver.supervise	FALSE	如果设置为真，驱动进程将会在遇到一个意外退出后重新启动，仅在Spark的standalone模式和Mesos的集群配置模式下有效

你可以访问应用程序的Web用户界面（驱动节点的4040端口）来检查配置是否正确，这些配置显示在“Environment”选项卡下，只有那些通过`spark-defaults.conf`，SparkConf或命令行明确指定的配置参数才会出现在这里，其他属性会默认使用默认配置。

运行时属性

尽管不是很常见，有时你可能还需要配置应用程序的运行时属性，由于篇幅有限，本书将省略该部分的详细介绍，请参阅Spark官方文档中关于运行时环境方面的表格，你可以配置驱动和执行器需要的额外classpath和Python路径、Python节点配置、以及其他日志相关的属性。

执行属性

执行属性是你最需要了解的配置属性，因为可以通过配置执行属性对应用程序进行细粒度的控制，由于篇幅有限，本书将省略该部分的详细介绍，请参阅Spark官方文档中有关执行行为的相关表格。最常见的配置是`spark.executor.cores`（用于控制使用的核心数）和`spark.files.maxPartitionBytes`（读取输入文件时的最大分块大小）。

配置内存管理

有时你可能需要手动配置内存选项以尝试优化应用程序，终端用户大多数情况下不需要手动进行内存管理，因为它们涉及很多旧概念，并且在Spark 2.X版本中内存的细粒度控制并不推荐使用，而是推荐使用新支持的自动内存管理。由于篇幅有限，本书将省略该部分的详细介绍，请参阅Spark官方文档中有关内存管理的相关表格。

配置Shuffle行为

Shuffle是Spark作业执行中的瓶颈，因为它的通信开销很大。因此，有许多用于控制shuffle行为的低级配置。由于篇幅有限，本书将省略该部分的详细介绍，请参阅Spark官方文档中有关shuffle行为的相关表格。

环境变量

你可以通过环境变量来配置Spark的某些参数，这些环境变量的默认位置是在Spark根目录下的*conf/spark-env.sh*脚本中读取的（或Windows上的*conf/spark-env.cmd*）。在Standalone模式和Mesos模式下，该文件可以提供特定主机的信息，例如主机名。此外，本地运行Spark应用程序或提交脚本时，它也会被读取和加载。

请注意，默认情况下在安装Spark后并不存在*conf/spark-env.sh*文件，需要复制conf/*spark-env.sh.template*来创建它，并且请赋予该文件具有可执行权限。

可以在*spark-env.sh*中设置以下变量：

`JAVA_HOME`
: 安装Java的位置（如果它不在默认路径上）。

`PYSPARK_PYTHON`
: 在驱动和工作节点上PySpark使用的Python（如果可用，默认为Python2.7，否则为Python）。如果设置了属性`spark.pyspark.python`，则优先使用。

`PYSPARK_DRIVER_PYTHON`
: 在驱动节点上PySpark使用的Python（默认为`PYSPARK_PYTHON`）。如果设置了属性`spark.pyspark.driver.python`，则优先使用。

`SPARKR_DRIVER_R`
: SparkR shell使用的R（默认为R）。如果设置了属性`spark.r.shell.command`，则优先使用。

`SPARK_LOCAL_IP`
: 节点绑定的IP地址。

`SPARK_PUBLIC_DNS`
: 用于与其他节点通信的主机名。

除了上面列出的这些环境变量外，还有用于设置Spark的standalone集群的选项，例如每台计算机上使用的内核数量和最大内存。由于*spark-env.sh*是一个shell脚本，你可以通过编程方式设置，例如，你可以查找特定网络接口的IP并依此来设置`SPARK_LOCAL_IP`。

在集群模式下在YARN上运行Spark时，需要使用*conf/spark-defaults.conf*文件中的`spark.yarn.appMasterEnv.[EnvironmentVariableName]`属性来设置环境变量。在*spark-env.sh*中设置的环境变量不会反映在集群模式下的YARN应用程序master进程中。有关更多信息，请参阅YARN相关的Spark属性。

应用程序之内的作业调度

在给定的Spark应用程序中，如果多个并行作业是从不同的线程提交的，它们可以同时运行。Spark作业的意思是一个Spark动作以及执行该动作的所需要启动的任务。Spark的调度是完全线程安全的，使应用程序支持多个请求（例如，来自多个用户的查询请求）。

默认情况下，Spark遵循FIFO（先进先出）方式调度作业。如果位于队列头的作业不需要使用整个集群资源，后面的作业可以立即开始，但如果队列头的作业工作量很大，位于队列后部的作业可能会被延迟启动。

也可以配置作业之间公平共享集群资源。Spark以轮询（round-robin）方式调度各作业并分配任务，以便所有作业获得大致相等的集群资源份额。这意味着当一个大作业正在运行并占用集群资源时，新提交的小作业可以立即获得计算资源并被启动，无需等待大作业的结束再开始，所以可以获得较短的响应时间。此模式最适合多用户情况。

要启用公平调度，需要在配置`SparkContext`时将`spark.scheduler.mode`属性设置为`FAIR`。

公平调度程序还支持将作业分组到作业池中，并为每个池设置不同的调度策略或优先级。可以为更重要的作业创建高优先级作业池，也可以将每个用户的工作组合在一起配置一个作业池，并为每个用户分配相同的调度资源，而不管他们的并发作业有多少。这种方法类似Hadoop的Fair Scheduler。

默认情况下，新提交的作业会进入默认池，也可以通过`SparkContext`设置作业要提交到的作业池，即设置`spark.scheduler.pool`属性。这可以按如下方法完成，假定sc是你的`SparkContext`：

```
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

设置此本地属性后，此线程提交的所有作业将使用此作业池名称。该设置是按线程进行配置的，以便让代表同一用户的线程可以配置多个作业。如果你想清除该线程关联的作业池，请将该属性设置为null。

小结

本章涵盖了很多关于Spark应用程序的内容，涉及了如何用Spark支持的所有语言编写、测试、运行和配置它们。在第17章，我们讨论运行Spark应用程序时的程序部署选项和集群管理选项。

第17章

部署Spark

本章探讨Spark的基础架构，了解这些有助于你部署和运行Spark应用程序：

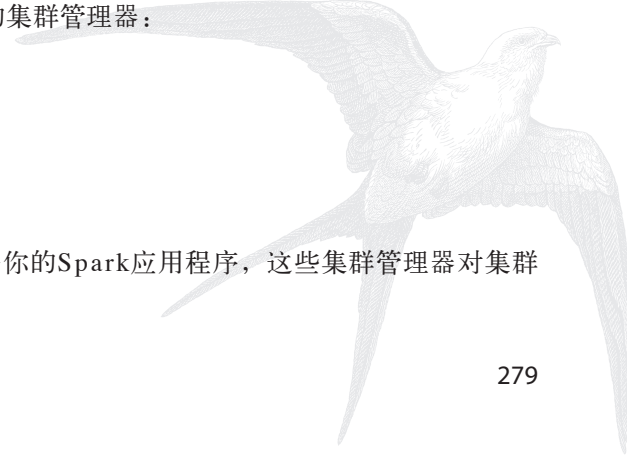
- 集群部署的选择。
- Spark的不同集群管理者。
- 部署注意事项以及配置部署。

在大多数情况下，Spark在各种集群管理器下的工作模式类似。然而，定制集群管理器就需要了解每个集群管理系统的工作原理，最困难的部分是选择集群管理器（或选择托管服务）。尽管我们很乐意介绍关于如何使用不同群集管理器配置集群的所有细节，但本书不可能涵盖各种环境下的各种情况。因此，本章的目标不是详细讨论每个集群管理器，而是主要介绍它们的根本区别，另外Spark网站上已有大量资料可供参考。值得注意的是，对于“哪个是最容易的集群管理器”这个问题并没有简单的答案，这取决于实际应用环境、用户经验和可用资源。Spark官方文档提供了大量有关示例介绍部署Spark的详细信息。

在撰写本文时，Spark有三个官方支持的集群管理器：

- Standalone模式。
- Hadoop YARN。
- Apache Mesos。

这些集群管理器维护一组机器用于部署你的Spark应用程序，这些集群管理器对集群



的管理模式各不相同，所以你需要权衡利弊。不过尽管如此，它们都以相同的方式运行Spark应用程序（如第16章所述）。首先我们从第一个问题开始：在哪里部署集群。

在哪里部署Spark集群

关于在哪里部署Spark集群，有两个选择：部署在本地群集或公共云中。这个选择很重要，所以我们要详细讨论一下。

本地部署集群

在某些时候应该将Spark部署到本地集群，特别是对于那些有自己数据中心的某些大企业或组织，他们就应该将Spark部署到本地数据中心。当然，这种方法也是有利有弊。本地集群允许你完全控制所用硬件设备，可以针对特定的工作负载来优化性能。但是，它也带来了一些问题，特别是在应付数据分析作业方面。首先，如果采用本地部署，集群的大小往往是固定的，而数据分析作业的资源需求往往是有弹性的。如果你的集群太小，那么很难执行偶尔需要的任务量异常繁重的分析查询或训练工作，而如果集群规模很大，将会有资源闲置和资源浪费情况发生。其次，本地集群需要相应地维护自己的存储系统，例如HDFS或某些具有扩展性的键值存储，你也需要设置一套灾备方案，比如根据需要设置备份机制和容错机制。

如果你打算在本地部署，解决资源利用问题的最佳方法就是使用集群管理器，集群管理器支持在集群上运行多个Spark应用程序并在它们之间动态地重分配资源，甚至支持非Spark应用程序共享集群资源。Spark的所有集群管理器都支持多个并发Spark应用程序，但YARN和Mesos对动态资源分配和非Spark程序共享有更好的支持。处理资源共享问题可能是使用Spark本地部署与云部署最大的区别，在公有云中，可以为每个应用程序构建合适的Spark集群大小，并在应用程序的生命周期内为其动态提供所需的集群资源。

关于存储，也有几种不同的选择，若要深入理解所有的利害并了解所有操作细节，你需要一本介绍存储的专业书籍。Spark所使用的最常见的存储系统是分布式文件系统，如Hadoop的HDFS以及Apache Cassandra等键值存储。像Apache Kafka这样的流式消息系统也经常用于缓存摄取的数据。所有这些存储系统都不同程度地支持存储管理、备份和地理备份等功能，这些功能有时内置于系统中，有时需要借助于第三方工具。在选择存储之前，建议评估其与Spark连接器的性能并考虑管理工具的易用性。

在公有云上的Spark

虽然早期的大数据系统是为本地集群部署而设计的，但云计算平台是目前部署Spark的主流平台。当处理大数据时，在公有云上部署具有几个优势。首先，可以弹性地申请和释放计算资源，可能你偶尔有一个特别消耗资源的需求，需要数百个节点运行数百小时来完成一个超大作业，使用云计算平台你就无需为这偶尔的计算需求而采购数百个节点，你仅需要按需付费。即使为了满足日常的普通需求，你也可以为应用程序选择不同的集群规模和集群节点硬件配置，以获得最优的性价比，例如，你可以申请带有图形处理器（GPU）的节点用于深入学习工作。其次，公有云支持低成本的跨地域备份存储，可以更便捷地管理这些大数据。

许多公司希望将他们的应用程序部署到公有云上，并像在本地集群上运行引用程序一样在公有云上运行它们，所有的主流云服务提供商（亚马逊的Amazon Web Services [AWS]，微软的Azure，谷歌的Cloud Platform [GCP]和IBM的Bluemix）都为它们的客户提供配置好的Hadoop集群，提供HDFS存储和Apache Spark计算框架。然而，这实际上并不是运行Spark的最好方法，因为如果绑定计算框架和存储，或者使用固定的计算集群规模和文件系统规模，你将无法充分利用云计算的弹性优势。所以建议使用与计算集群分离的存储系统（如Amazon S3，Azure Blob Storage，Google Cloud Storage），并为每个Spark作业动态绑定存储系统。通过分离计算和存储，你就可以仅在需要时为计算资源付费，并且更容易扩展它们，并可以混合使用不同的硬件类型。所以，在云中运行Spark并不仅仅意味着将本地安装迁移到云虚拟机上，你可以自然而然的运行与存储系统解耦的Spark来充分利用云计算的弹性优势，降低成本，其丰富的管理工具可帮助你摆脱在本地部署中管理集群的困扰。

有几家公司提供基于云的Spark服务，所有Apache Spark安装可以直接连接到云存储。由加州大学伯克利分校的Spark团队创办的Databricks公司是一家专门提供基于云的Spark应用集群构建的服务提供商。Databricks提供了无需安装Hadoop的简单方式来运行Spark作业，也提供了许多在云中更有效地运行Spark的功能，例如集群自动扩展、集群自动终止、针对云存储的优化连接器、以及适用于Notebook协作开发环境和单机运行作业。Databricks公司还提供免费的社区版用于学习Spark，你可以在一个小型群集上运行Notebook协作环境，并与其他人分享使用。有趣的是，整本书的示例都是基于免费的Databricks社区版编写的，我们发现集成Spark的Notebook环境、实时协作和集群管理这些因素，是编写和测试示例的最简单方法。

如果你在云中运行Spark，本章中的大部分内容可能并不是特别相关，因为你可以为每个作业创建一个单独的和短期的Spark集群，在这种情况下，Standalone集群管理器

可能是最容易使用的。但是，如果你想创建一个多应用共享的长期运行Spark集群，或者你需要自行在虚拟机上安装Spark，则你仍可能需要仔细阅读本章内容。

集群管理器

除非你使用高级托管服务，否则你需要为Spark选择集群管理器。Spark支持三种集群管理器：Standalone集群，Hadoop YARN和Mesos。我们接下来回顾一下这些内容。

Standalone集群管理器

Spark的Standaone集群管理器是专门为Apache Spark工作负载构建的轻量级平台。Standaone集群管理器允许你在同一个物理集群上运行多个Spark应用程序，它还提供了简单的操作界面，也可以扩展到大规模Spark工作负载。Standaone模式的主要缺点是它的功能相对其他的集群管理器来说比较有限，特别是集群只能运行Spark作业。不过，如果你只想快速让Spark集群运行，并且你没有使用YARN或Mesos的经验，那么Standaone模式是最合适的。

启动Standalone集群

启动Standaone集群首先需要操控集群包含的物理节点，也就是要启动它们，确保它们可以通过网络互相通信，并在这些机器上安装正确版本的Spark软件。之后，有两种方法可以启动集群：手动或使用内置的启动脚本。

我们首先手动启动一个集群。第一步是使用以下命令在某台机器上启动主进程：

```
$SPARK_HOME/sbin/start-master.sh
```

当我们运行这个命令时，集群管理器主进程将在该机器上启动，然后会在命令行打印出一个URI，即`spark:// HOST:PORT`。你可以在应用程序初始化时将此URI用作SparkSession的主参数，你还可以在master节点的Web用户界面上找到此URI，默认情况下的Web用户界面地址是*http:// master-ip-address:8080*。另外，可以登录到每台计算节点并使用该URI运行以下脚本来启动worker节点，注意master节点必须在网络上可访问，并且master节点的指定端口也必须打开：

```
$SPARK_HOME/sbin/start-slave.sh <master-spark-URI>
```

只要你在另一台工作节点上运行它，就启动了一个worker进程，这样你就构建了一个包括一个master节点和一个worker节点的Spark集群。这个创建过程是手动的，幸运的是，有脚本可以自动化这个过程。

集群启动脚本

你能配置可以自动启动Standalone集群的启动脚本。为此，你需要在Spark目录中创建一个名为*conf/slaves*的文件，该文件需要包含启动worker进程的所有计算机主机名，每行一个。如果这个文件不存在，那么集群将会以本地模式启动。实际启动集群时，master节点将通过Secure Shell（SSH）访问每个worker节点。默认情况下，SSH并行运行，并要求你配置无密码（使用私钥）访问环境。如果你没有无密码设置，则需要设置环境变量SPARK_SSH_FOREGROUND来顺序地为每个worker节点提供访问密码。

设置此文件后，可以使用以下shell脚本启动或停止集群，这些脚本基于Hadoop的部署脚本，并且可在$SPARK_HOME / sbin中找到：

`$ SPARK_HOME/sbin/start-master.sh`
: 在执行脚本的机器上启动master实例。

`$ SPARK_HOME / sbin/ start-slaves.sh`
: 在*conf/slaves*文件中指定的每台机器上启动一个slave实例。

`$ SPARK_HOME / sbin/ start-slave.sh`
: 在执行脚本的机器上启动一个slave实例。

`$ SPARK_HOME / sbin/ start-all.sh`
: 如前所述，按照配置文件，在指定的机器上启动一个master实例和在指定的多台机器上启动多个slave实例。

`$ SPARK_HOME/sbin/stop-master.sh`
: 停止通过*bin/start-master.sh*脚本启动的master实例。

`$ SPARK_HOME/sbin/stop-slaves.sh`
: 停止*conf/slaves*文件中指定的机器上的slave实例。

`$ SPARK_HOME/sbin/ stop-all.sh`
: 如前所述，停止配置文件指定机器上的master实例和slave实例。

Standalone集群配置

Standalone集群具有许多调节应用程序的配置，这些配置涵盖关于终止应用程序的旧文件处置、工作节点的计算核心和内存资源配置等方面，这些是通过环境变量或应用程序属性来进行控制的。由于篇幅有限，本书不会介绍所有的配置，请参阅Spark文档中关于Standonle环境变量的相关总结表格。

提交应用程序

创建集群后，你可以在代码中通过指定master节点的URI来将应用程序提交给集群，你也可以使用spark-submit命令在master节点或其他机器上提交执行应用程序申请。我们在第16章的“启动应用程序”介绍了一些Standalone模式的命令行参数。

YARN集群管理器

Hadoop YARN是支持作业调度和集群资源管理的框架。Spark经常被误认为是“Hadoop生态系统”的一部分，但事实上，Spark与Hadoop几乎没有关系，虽然Spark本身支持Hadoop YARN集群管理器，但它并不需要Hadoop的支持。

通过在spark-submit命令行参数中将master节点指定为YARN，你可以在Hadoop YARN上运行Spark作业。就像使用Standalone模式一样，你可以根据需求调整集群配置。由于Hadoop YARN是大量不同执行框架的通用调度器，因此配置的参数数量自然也多于Spark的Standalone模式。

本书并不讨论配置一个YARN集群，但是关于该主题的一些精彩的书籍可以帮助你有更深的了解，或者使用托管服务可以简化该配置过程。

提交应用程序

将应用程序提交给YARN时，与其他部署的核心区别在于，--master需要指定的是yarn而不是master节点的IP（在Standalone模式中需要master节点IP）。Spark将使用环境变量HADOOP_CONF_DIR或YARN_CONF_DIR来查找YARN配置文件，在将这些环境变量设置为Hadoop的安装目录后，就可以像我们在第16章介绍的那样运行spark-submit来提交应用程序。

有两种部署模式可用于在YARN上启动Spark。正如前面的章节所讨论的，集群模式将spark驱动器作为由YARN集群管理的进程，客户端在创建应用程序后退出。在客户端模式下，驱动器将运行在客户端进程中，因此YARN只负责将执行器的资源授予应用程序，而不是维护master节点。另外值得注意的是，在集群模式下，Spark不一定在你正在执行spark-submit命令的同一台机器上运行。因此，库和外部jar必须手动配置或通过--jars命令行参数配置。

有几个特定于YARN的属性可以使用spark-submit来设置，这些包括优先级队列控制以及诸如keytabs之类的安全性配置。我们之前在第16章的“启动应用程序”介绍了这些内容。

在YARN应用程序上配置Spark

将Spark部署为YARN应用程序需要你了解Spark应用程序的各种不同配置及其含义。本节将介绍基本一些运行Spark应用程序配置的最佳实践经验。

Hadoop配置

如果你想使用Spark从HDFS进行读写，则需要在Spark的classpath中包含两个Hadoop配置文件：*hdfs-site.xml*（配置HDFS客户端）和*core-site.xml*（设置默认的文件系统名称）。这些配置文件的位置因Hadoop版本而异，但常见的位置在*/etc/hadoop/conf*中。一些工具也可以即时创建这些配置文件，因此了解托管服务如何部署这些配置也很重要。

为了让这些文件对Spark可见，请将*$SPARK_HOME/spark-env.sh*中的`HADOOP_CONF_DIR`设置为包含配置Hadoop文件的位置，或者在你启动应用程序时将其设置为环境变量。

YARN的应用程序属性

有很多与Hadoop相关的配置，也有很多与Spark不太相关的配置，它们是为了运行或保护YARN而同时会涉及Spark的运行方式。由于篇幅有限，我们无法在此处介绍详细配置。请参阅Spark文档中有关YARN配置的相关表格。

Mesos集群管理器

Apache Mesos是另一个可以运行Spark的集群系统。Mesos项目其实也是由许多Spark的原创作者开发的，其中包括本书的作者之一。按照Mesos项目自己的话说：

> Apache Mesos将CPU、内存、存储和其他计算资源从（物理的或虚拟的）机器中抽象出来，使容错机制和弹性分布式系统能够轻松构建并有效运行。

在大多数情况下，Mesos是一个数据中心规模集群管理器，它不仅管理像Spark这样的短期应用程序，还管理诸如Web应用程序等长时间运行的应用程序。Mesos是一个重量级的集群管理器，只有当你的组织机构已经大规模部署了Mesos时才建议选择此集群管理器，但不可否认它仍然是一个优秀的集群管理器。

Mesos是一个庞大的架构，我们需要太多的内容来具体介绍如何部署和维护Mesos集群，关于这个主题有很多很好的书籍可以参考，包括Dipa Dubhashi和Akhil Das的《Mastering Mesos》（O'Reilly，2016）。本节的目标是总结在Mesos上运行Spark应用程序时需要考虑的一些注意事项。

例如，关于Mesos上Spark的最常见的讨论是使用细粒度模式还是粗粒度模式，在以前的老版本中Mesos支持各种不同的模式（细粒度和粗粒度），但现在它仅支持粗粒度调度（细粒度已被弃用）。粗粒度模式意味着每个Spark执行作业作为单个Mesos任务运行，Spark执行器的大小根据以下应用程序属性确定：

- `spark.executor.memory`
- `spark.executor.cores`
- `spark.cores.max/spark.executor.cores`

提交应用程序

将应用程序提交给Mesos集群与提交到其他集群管理器的操作类似。在大多数情况下，你应该在使用Mesos的集群模式，客户端模式需要你进行额外的配置，特别是在集群周围分配资源方面。

例如，在客户端模式下，驱动器需要*spark-env.sh*中的额外配置信息才能使用Mesos。

在*spark-env.sh*中设置一下一些环境变量：

```
export MESOS_NATIVE_JAVA_LIBRARY=<path to libmesos.so>
```

此路径通常是*<prefix>/lib/libmesos.so*，默认情况下前缀为*/usr/local*。在Mac OS X上，该库的名称为*libmesos.dylib*而不是*libmesos.so*：

```
export SPARK_EXECUTOR_URI=<URL of spark-2.2.0.tar.gz uploaded above>
```

最后，将Spark应用程序属性`spark.executor.uri`设置为`<URL of spark-2.2.0.tar.gz>`。现在，当在集群启动Spark应用程序时，在创建`SparkContex`时将一个master的URL提交，`mesos://` URL，并将该属性设置为`SparkConf`变量中的参数或对`SparkSession`的初始化时用到：

```
// in Scala
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder
  .master("mesos://HOST:5050")
  .appName("my app")
  .config("spark.executor.uri", "<path to spark-2.2.0.tar.gz uploaded above>")
  .getOrCreate()
```

提交集群模式的应用程序相当简单，并遵循与之前相同的`spark-submit`结构。我们在第16章的“启动应用程序”中涵盖了这些内容。

配置Mesos

就像任何其他集群管理器一样，我们可以通过多种方式配置在Mesos上运行的Spark应用程序。由于篇幅有限，我们无法在此处介绍所有配置。请参阅Spark文档中关于Mesos配置的相关表格。

安全部署配置

Spark还提供了一些保证应用程序安全性的低级功能，尤其是在不可信的环境中，大部分设置作用在Spark外部。这些配置主要是网络相关的，以帮助Spark以更安全的方式运行，这包括身份验证、网络加密、以及设置TLS和SSL等配置。由于篇幅有限，本书不在此处介绍所有配置信息，请参阅Spark文档中安全配置主题的相关表格。

集群网络配置

就像shuffle操作一样，网络相关的配置对Spark运行也很重要。当你需要对Spark集群进行自定义网络部署配置时，例如在特定节点之间使用代理，网络配置也很重要。虽然网络配置并不是决定Spark性能最重要的配置，但网络相关配置常会出现在部署方案中。由于篇幅有限，本书不在此处介绍所有详细配置信息，请参阅Spark文档中有关网络配置的相关表格。

应用程序调度

Spark提供若干工具用于在应用程序之间的计算资源调度。首先回想一下，每个Spark应用程序都会启动一组执行器进程，集群管理器要对这些进程进行调度。其次，在一个Spark应用程序中，如果有多个作业是由不同的线程提交的，则可能会并发地运行多个作业（即Spark动作），Spark包含一个公平调度器来调度每个应用程序中的计算资源。我们在前一章中已经介绍了这个主题。

如果多个用户需要共享集群并运行不同的Spark应用程序，则根据群集管理器的不同，可以使用不同的选项来管理和分配计算资源。最简单的调度方式是资源静态划分，通过这种方法，每个应用程序被分配到一定量的资源，并在整个程序运行期间一直占用这些资源。使用`spark-submit`命令可以设置一些选项来控制特定应用程序的资源分配。请参阅第16章以获取更多信息。另外，可以打开动态分配（随后介绍），这样可以根据当前未执行任务的数量进行动态扩展和缩减计算资源。另外，如果你希望用户能够以细粒度的方式共享内存和计算资源，则可以在单个Spark应用程序内进行线程调度来并行处理多个请求。

动态分配

如果在同一个群集上同时运行多个Spark应用程序，Spark提供了一种可以根据工作负载动态调整应用程序占用的资源的机制。也就是说，在应用程序不再使用资源时将资源返回给集群，并在有资源需要时再次请求使用。如果多个应用程序共享Spark集群中的资源，资源的动态分配就尤为重要。

此功能在默认情况下处于禁用状态，但是在粗粒度集群管理器上都支持，即Standalone模式，YARN模式和Mesos粗粒度模式。使用此功能有两个要求：首先，你的应用程序必须将`spark.dynamicAllocation.enabled`属性设置为`true`；其次，你要在每个工作节点上设置外部Shuffle服务，并在应用程序中将`spark.shuffle.service.enabled`属性设置为`true`。外部shuffle服务的目的是为了允许终止执行进程而不删除由它们输出的shuffle文件。每个集群管理器对设置动态分配功能的方法都不同，这在配置作业调度部分进行了描述。由于篇幅有限，本书不详细介绍动态分配的配置参数，请参阅动态分配配置的相关表格。

其他注意事项

在部署Spark应用程序时还要考虑其他几个方面，这些可能会影响你对集群管理器的选择及其具体设置的选择，这些是你比较不同部署选项时应该考虑的事情。

其中一个重要的考虑因素是你打算运行的应用程序的数量和类型。例如，YARN非常适合基于HDFS的应用程序，但不常用于其他许多应用程序。此外，YARN不太适合在云计算环境下运行，因为它需要从HDFS上获取信息，计算和存储在很大程度上是耦合在一起的，这意味着当需要扩展存储和计算中的某一种资源时，你都需要同时扩展存储和计算。Mesos支持更广泛的应用程序类型，但它仍然需要预先配置机器，并且在某种意义上来说需要更大规模的投入，但是只为运行Spark应用程序而部署Mesos集群是没多大意义的。Spark的Standalone模式是最轻量级的集群管理器，并且相对易于理解和使用。但是如果后期你需要更多其他的分布式计算框架，使用YARN或Mesos则更适合。

另一个难题是管理不同Spark版本的应用程序。如果你想运行基于不同Spark版本的多个应用程序，那么这些应用程序常常是混淆在一起的。除非使用管理良好的应用程序，否则你需要花费很多时间来管理这些Spark应用程序的配置脚本，或者直接放弃这种能力而只允许用户使用同一Spark版本的应用程序。

无论选择哪种集群管理器，都需要考虑如何设置日志记录、如何存储日志以供将来参

考，并允许终端用户基于日志调试其应用程序。YARN或Mesos提供了良好的日志功能，如果使用Standalone模式，则需要进行一些配置。

另外一件你可能需要考虑的是维护一个元数据存储，以维护关于数据集的元数据，例如table catalog。在使用Spark SQL创建和维护数据表的时候，可能就涉及创建一个元数据存储。维护一个Apache Hive的元数据存储也有益于提高数据集的使用效率，特别是对多个应用程序共享同一数据集的情况，但是Apache Hive元数据存储并不在本书的介绍范围之内。

根据你的工作量，可能要考虑使用Spark的外部shuffle服务。Spark会在某节点的本地磁盘上存储shuffle块（shuffle输出）。外部shuffle服务存储这些shuffle块，以便它们可供所有进程使用，也就是说，可以终止任意执行进程，而其他进程仍然可以访问该被终止进程产生的shuffle输出。

最后，需要配置一些基本的监控方案以帮助用户调试集群运行的Spark作业，第18章将具体介绍这些细节。

小结

本章介绍部署Spark时的配置选项。尽管大多数普通用户并不会涉及部署高效的Spark，但是如果你需要部署Spark时，本章就值得一读。还有其他一些可以控制更低层次行为的配置，本章并没有介绍，你可以通过学习Spark文档或Spark源代码来了解这些内容。第18章将介绍监测Spark应用程序时的一些配置选项。

第18章

监控与调试

本章将介绍关于Spark应用程序的监控和调试。我们将通过一个示例查询来熟悉Spark UI，以帮助你了解在整个作业执行生命周期内如何跟踪自己提交的Spark作业，我们也将从示例中了解如何调试作业，以及如何定位错误产生的位置。

监控级别

在发生错误的时候，需要监控Spark作业的执行情况以了解问题所在，关注一下哪些部分我们必须监控的，这是非常必要的。下面回顾我们可以监控的组件，并概述一些监控选项（见图18-1）。

Spark应用程序和作业

无论是为了调试还是更好地理解应用程序在集群上的执行过程，通过Spark UI和Spark日志是最方便获取监控报告的方式，这些报告包括Spark应用程序的运行状态信息，例如RDD转换和查询计划的执行信息等。本章将详细讨论如何使用这些Spark监视工具。

JVM

Spark在Java虚拟机（JVM）上运行执行器。因此，下一个监视层次是监控虚拟机（VM）以更好地理解代码的运行方式。JVM提供一些监视工具，如用于跟踪堆栈的jstack，用于创建堆转储（heap-dumps）的*jmap*，用于报告时序统计信息的*jstat*，以及用于可视化JVM属性的*jconsole*，这些工具对于那些熟悉JVM内部机理的人员非常有用。你也可以使用像*jvisualvm*这样的工具来帮助分析Spark作

业。其中一些JVM监视信息已经在Spark UI中提供了，但对于更低层次的调试，上述工具可派上用场。

操作系统/主机

JVM运行在主机操作系统（OS）上，监视这些机器的运行状态也很重要，这包括监控诸如CPU、网络、I/O等。这些信息通常在集群级监控方案中也会报告，但是你可以使用更专业的工具来获得更详细的监视信息，这些工具包括*dstat*、*iostat*和*iotop*。

集群

当然，你也可以监视运行Spark应用程序的集群，这可能是YARN，Mesos或Standalone集群。集群监控方案很重要，如果集群不正常工作，你需要很快知道。一些流行的集群级监控工具包括Ganglia和Prometheus。

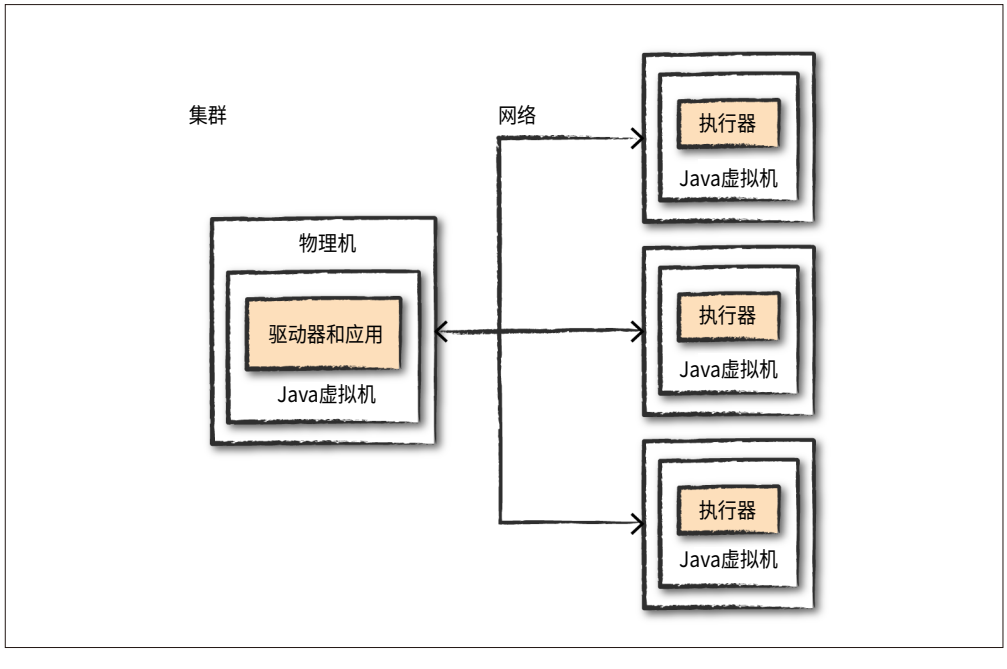


图18-1：可监控的Spark应用程序组件

要监视什么

在简要介绍了监控级别之后，我们来讨论如何监控和调试我们的Spark应用程序。你需要监控的主要有两个方面：运行应用程序的进程信息（CPU使用率，内存使用率等）以及查询执行过程（作业和任务）。

驱动器和执行器进程

当监控一个Spark应用程序时，最需要注意的是驱动器进程，应用程序的所有状态都会在驱动器进程上有所反映，你需要确保它正确而稳定的运行。如果你只能监控一台机器或一台JVM，那首选就是驱动器节点。当然，了解执行器的状态对于监控Spark作业也非常重要，Spark提供一个基于Dropwizard Metrics Library的可配置指标监视系统，它的配置文件一般在*$SPARK_HOME/conf/metrics.properties*中指定，可以通过更改`spark.metrics.conf`配置属性来自定义配置文件位置，这些监控指标可以输出到包括Ganglia等多种不同的监控系统。

查询，作业，阶段和任务

尽管监视驱动器和执行器进程很重要，但有时你还需要对特定查询级别的进程进行调试。Spark提供了到查询级别、作业级别、阶段级别和任务级别的监控能力（我们在第15章已经介绍了这部分内容）。通过这些信息，你可以确切地知道在某时间在集群上正在执行什么工作。对于性能调优或程序调试来说，应该从这里开始找问题。

在知道我们想要监视什么之后，让我们看看两种最常见的监视方式：通过Spark日志和Spark UI。

Spark日志

获取最详细Spark监视信息的方法之一就是通过日志文件。Spark日志中记录的反常事件，或在Spark应用程序中有意添加的输出，都可以帮助你发现导致作业执行失败的原因。如果你使用本书提供的应用程序模板，我们在模板中设置的日志记录框架可以和Spark系统本身的日志配合得很好，以便于更好发现程序错误。有一个问题是Python无法直接与Spark的基于Java的日志记录库集成，但是使用Python的logging模块或者甚至简单的打印语句，仍然会将结果打印到报告错误的输出流，所以也很容易查找。

要更改Spark的日志级别，只需运行以下命令：

```
spark.sparkContext.setLogLevel("INFO")
```

这将允许你阅读日志，如果你使用我们的应用程序模板，则也可以记录你需要记录的相关信息，从而便于你检查自己的应用程序和Spark。在运行本地模式时，日志将会被打印到标准错误输出流，如果在集群上运行Spark时，日志会由群集管理器保存到

文件。请参阅集群管理器的文档，了解如何设置日志的输出位置，通常它们可通过集群管理器的Web UI获得。

通过搜索日志并不一定总会找到需要的答案，但它可以帮助你查明问题，或在应用程序中添加新的日志记录语句以便于更好地理解问题所在。你可能也会收集日志以为将来所用。例如，如果你的应用程序崩溃，而你无法访问或调试崩溃的程序，通过查询程序的日志可能会帮助你分析崩溃原因。你还可以将程序运行的日志发送到其他机器上进行保存，避免机器的异常崩溃或关闭（特别是在云中运行应用程序时）。

Spark UI

Spark UI提供了一种可视化的的方式在Spark和JVM级别来监视运行中的应用程序以及Spark工作负载的性能指标。每个运行的SparkContext都将启动一个Web UI，默认情况下在端口4040，它将列出应用程序的有用信息。例如，在本地模式下运行Spark时，通过访问*http://localhost:4040*即可在本地计算机上查看Web UI。如果你运行多个应用程序，它们将各自启动一个Web UI，并累加端口号（4041，4042，...），集群管理器还会从它自己的用户界面链接到每个应用程序的Web UI。

图18-2展示了Spark UI中所有的可用选项卡。

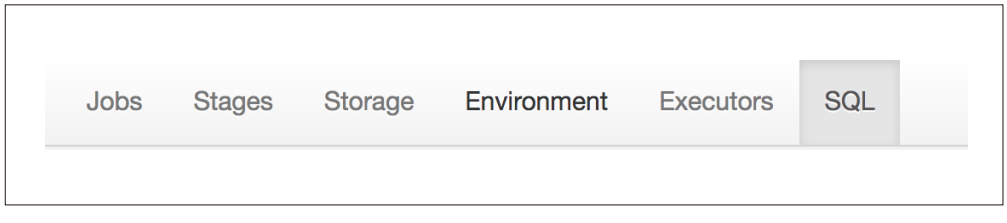


图18-2：Spark UI选项卡

这些标签涵盖了我们希望监控的每件事情，其对应的内容都很容易猜到：

- Jobs选项卡对应Spark作业。
- Stages选项卡对应各个阶段（及其相关任务）。
- Storage选项卡包含当前在Spark应用程序中缓存的信息和数据。
- Environment选项卡包含有关Spark应用程序的配置等相关信息。
- SQL选项卡对应我们提交的结构化API查询（包括SQL和DataFrame）。
- Executors选项卡提供应用程序的每个执行器的详细信息。

让我们来通过下面的例子来监控一个给定查询的状态信息。打开一个新的Spark shell，运行下面的代码，我们将通过Spark UI跟踪它的执行情况：

```
# in Python
spark.read\
  .option("header", "true")\
  .csv("/data/retail-data/all/online-retail-dataset.csv")\
  .repartition(2)\
  .selectExpr("instr(Description, 'GLASS') >= 1 as is_glass")\
  .groupBy("is_glass")\
  .count()\
  .collect()
```

输出结果为三行不同的值。上面代码启动了一个SQL查询，所以我们选择SQL选项卡，之后应该看到类似图18-3所示的信息。

最先看到的是关于此查询的汇总统计信息：

```
Submitted Time: 2017/04/08 16:24:41
Duration: 2 s
Succeeded Jobs: 2
```

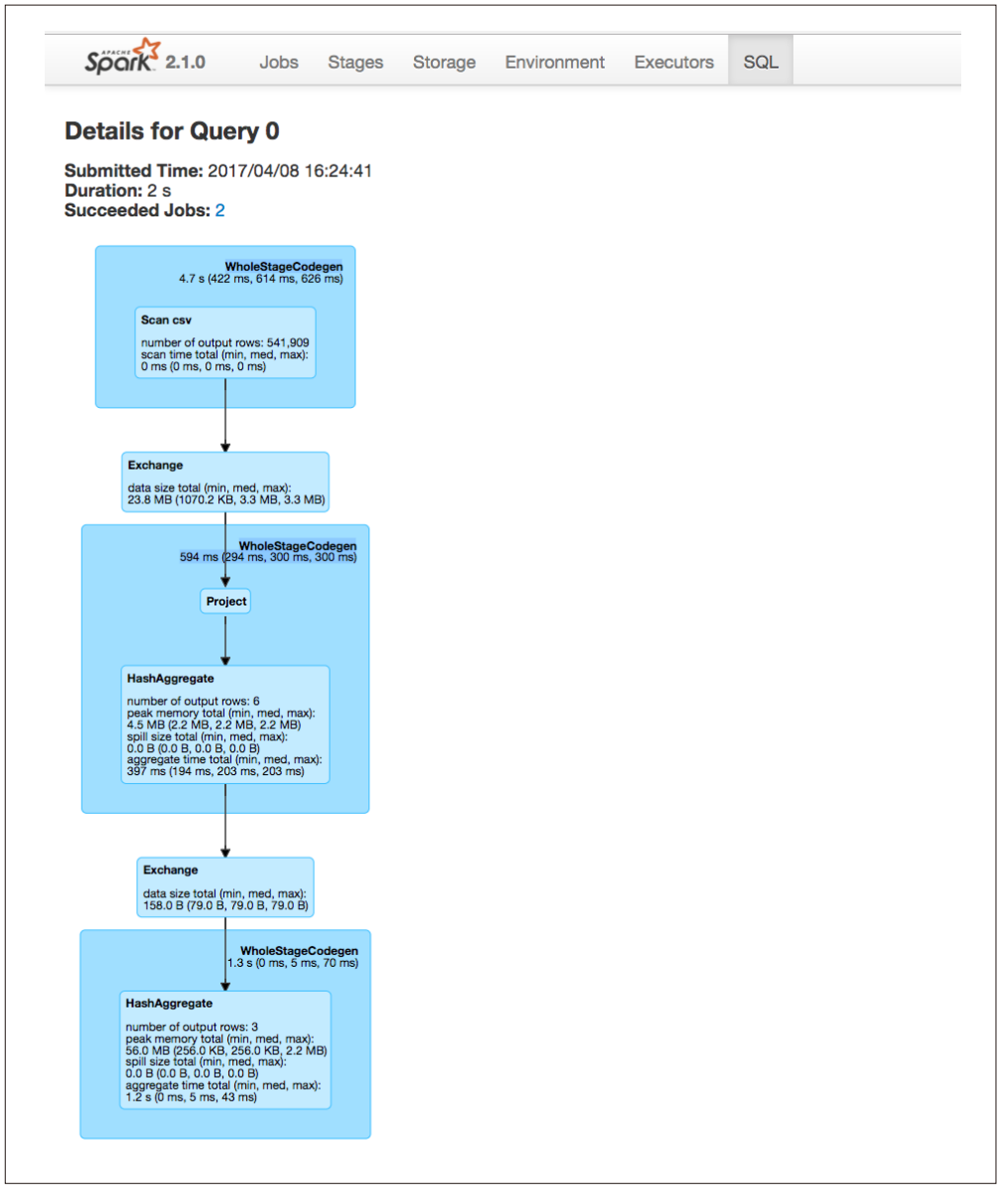


图18-3：SQL选项卡

首先我们来看看代表Spark各阶段联系的有向无环图（DAG），每个蓝色框代表Spark任务的一个阶段，所有这些阶段都代表一个Spark作业。我们来仔细看看每个阶段，以便我们能够更好地理解每个阶段发生的事情。图18-4展示了第一阶段。

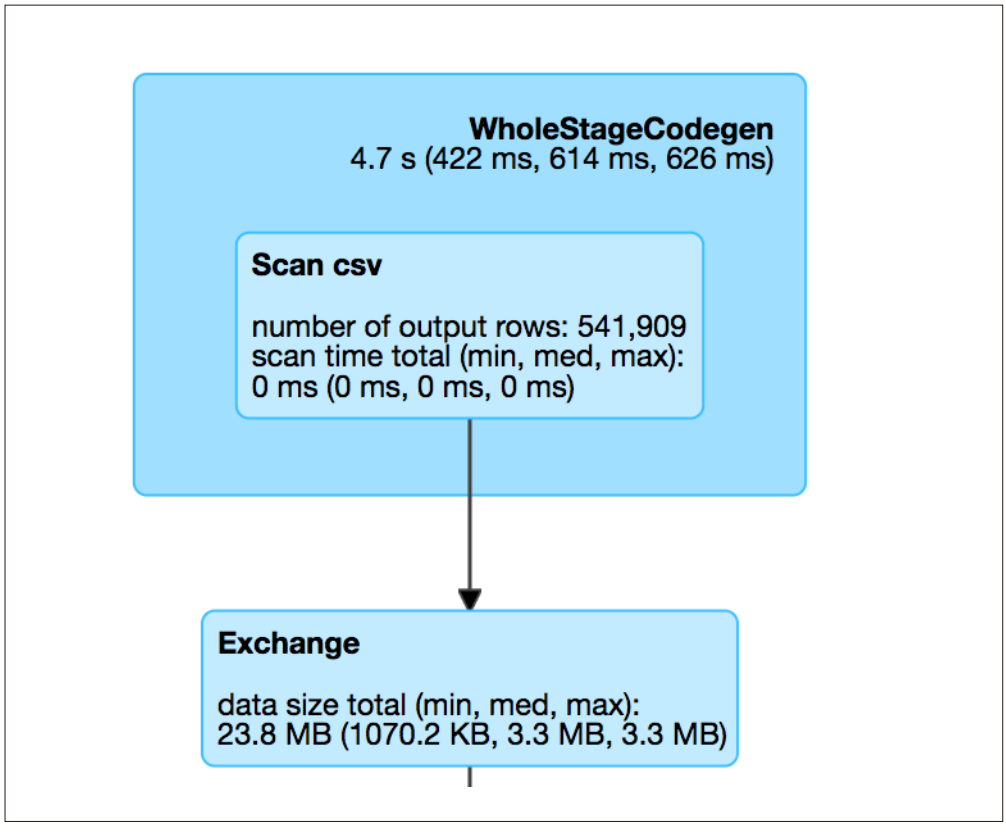


图18-4：第一阶段

标记为WholeStateCodegen的顶部框，代表对CSV文件的完整扫描。下面的蓝框代表一次强制执行的shuffle过程，因为我们调用了repartition函数，它将我们的（尚未指定分区数的）原始数据集划分为两个分区。

下一步是投影操作（选择/添加/过滤列）和聚合操作。请注意，在图18-5中输出行数是6，这等于输出行的数量与执行聚合操作时的分区数量的乘积。这是因为Spark在对数据进行洗牌以准备最后阶段之前，会对每个分区执行聚合操作（该种情况下是基于散列的聚合）。

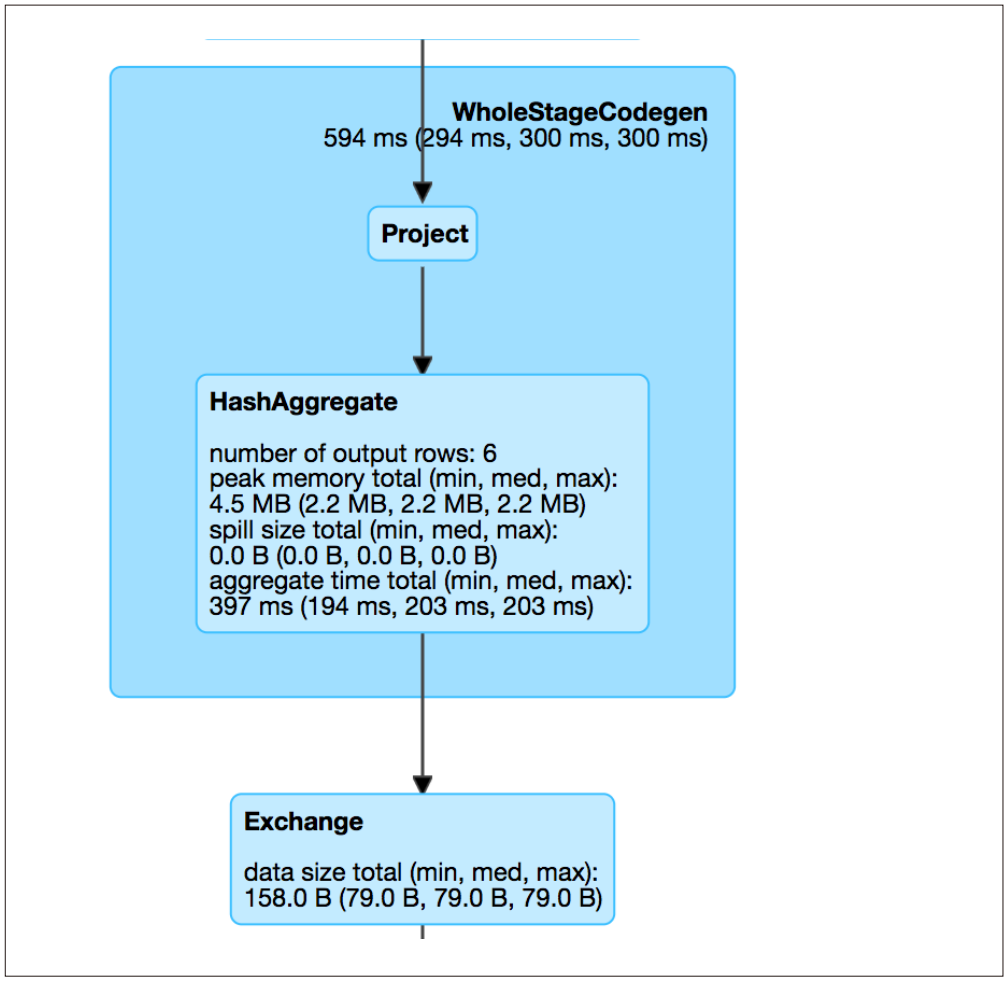


图18-5：第二阶段

最后一个阶段聚合第二阶段所有分区的聚合结果，将来自两个分区的最后三行结果合并，并作为总查询的结果输出（见图18-6）。

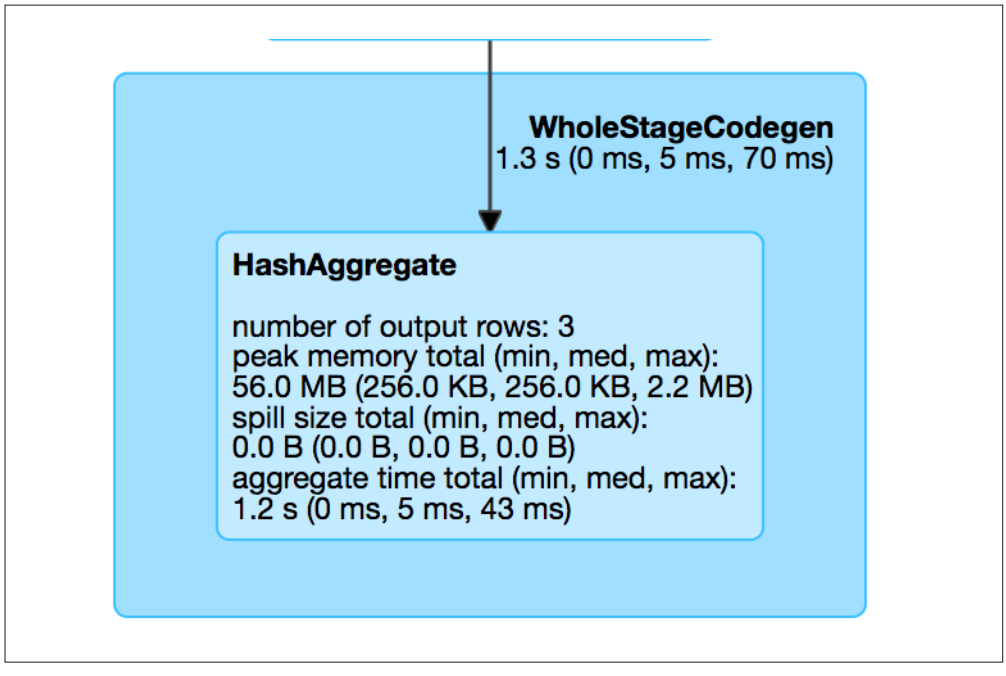


图18-6：第三阶段

让我们进一步看看作业的执行情况。在“作业”选项卡上的“Succeeded Jobs”旁边单击2，如图18-7所示，我们的作业分为三个阶段（这与我们在SQL选项卡上看到的三个阶段相对应）。

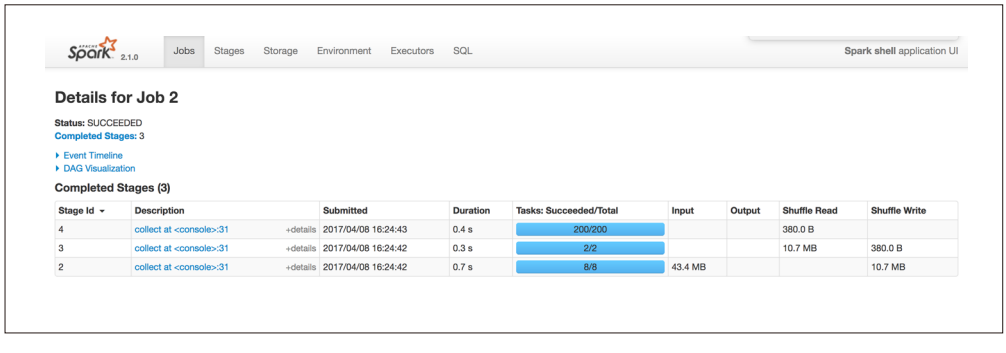
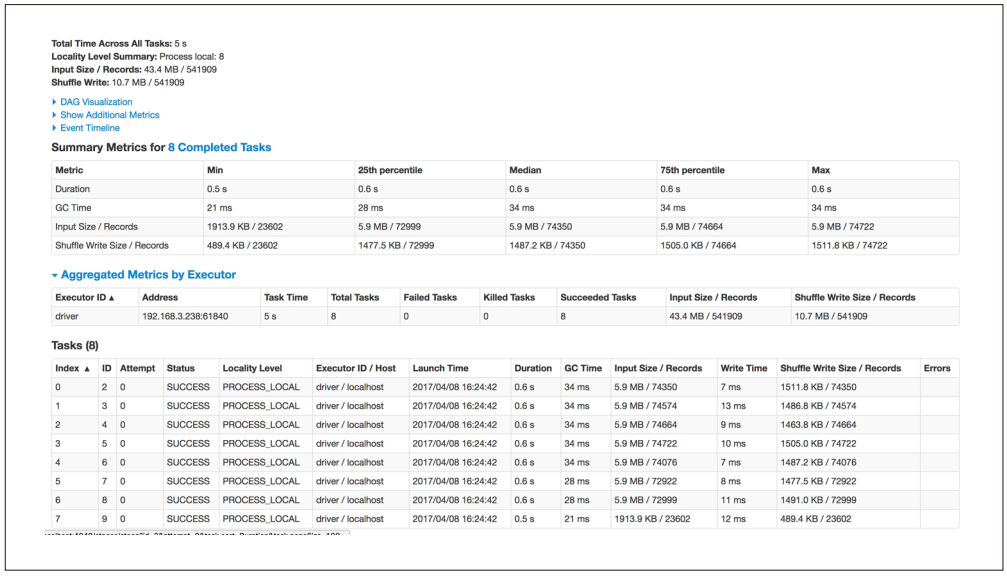


图18-7：Jobs选项卡

这些阶段或多或少具有与图18-6所示内容相同的信息，单击其中一个的标签将显示某阶段的详细信息。在这个例子中，有三个阶段，三个阶段分别有8个任务，2个任务和200个任务。在深入分析细节之前，让我们了解一下为什么会出现这种情况。

第一阶段有8个任务。输入的CSV文件是可拆分的，Spark将对输入数据均匀分布到机器的不同内核上进行并行处理。如果这发生在集群级别上，那么就涉及一个重要优化，如何存储文件。第二阶段有2个任务，因为我们显式地调用了重新分区操作，并将数据移动到2个分区中。最后一个阶段有200个任务，因为默认的shuffle分区是200个。

接下来查看下一级的细节，单击第一阶段后，其中包含8个任务，如图18-8所示。



Total Time Across All Tasks: 5 s
Locality Level Summary: Process local: 8
Input Size / Records: 43.4 MB / 541909
Shuffle Write: 10.7 MB / 541909

▸ DAG Visualization
▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.6 s	0.6 s	0.6 s	0.6 s
GC Time	21 ms	28 ms	34 ms	34 ms	34 ms
Input Size / Records	1913.9 KB / 23602	5.9 MB / 72999	5.9 MB / 74350	5.9 MB / 74664	5.9 MB / 74722
Shuffle Write Size / Records	489.4 KB / 23602	1477.5 KB / 72999	1487.2 KB / 74350	1505.0 KB / 74664	1511.8 KB / 74722

▾ Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
driver	192.168.3.238:61840	5 s	8	0	0	8	43.4 MB / 541909	10.7 MB / 541909

Tasks (8)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74350	7 ms	1511.8 KB / 74350	
1	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74574	13 ms	1486.8 KB / 74574	
2	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74664	9 ms	1463.8 KB / 74664	
3	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74722	10 ms	1505.0 KB / 74722	
4	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74076	7 ms	1487.2 KB / 74076	
5	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	28 ms	5.9 MB / 72922	8 ms	1477.5 KB / 72922	
6	8	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	28 ms	5.9 MB / 72999	11 ms	1491.0 KB / 72999	
7	9	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.5 s	21 ms	1913.9 KB / 23602	12 ms	489.4 KB / 23602	

图18-8：Spark任务

Spark提供了大量关于该作业在运行时所做工作的详细信息。在图18-8中的顶部，注意“Summary Metrics”部分，这里显示了关于各种指标的统计概要，注意警惕不均匀的数据分布（第19章将详细介绍）。在本例中，各指标分布比较均匀，没有大的波动。在底部的表格中，可以检查每个执行器的运行情况（在本例中，每个核心对应一个执行器），这可以帮助判断某个执行器是否严重过载。

Spark还提供了一组更详细的性能指标（虽然大多数普通用户可能不需要这些信息），如图18-8所示，请单击“Show Additional Metrics”，然后根据你想要查看的内容，全选（或取消全选）或者选择单个度量标准。

对每个阶段的分析与此类似，读者可自行练习。

其他Spark UI标签

其余的Spark选项卡，包括Storage、Environment和Executors选项卡的功能都很容易理解。Storage选项卡显示有关集群上缓存的RDD/DataFrame信息，这可以帮助查看某些数据是否已经从缓存中取出。Environment选项卡显示有关运行环境的信息，包括有关Scala和Java的信息以及各种Spark集群相关属性。

配置Spark UI

你可以对Spark UI进行多种配置。其中许多是网络相关配置，如启用访问控制等。也包括配置Spark UI的各种功能（例如，设置最多存储多少个作业、阶段、任务的信息）。由于篇幅有限，本书不介绍所有配置，请参阅Spark文档中关于Spark UI配置的相关表格。

Spark REST API

除了Spark UI之外，你还可以通过REST API查询Spark的状态和性能指标（可以访问*http://localhost:4040/api/v1*），利用REST API也是在Spark之上构建可视化和监视工具的一种方式。通过Web UI获得的大多数信息也可以通过REST API获得，只是它不包含SQL的相关信息。如果要根据Spark UI中提供的信息构建自己的监控平台，REST API将是一个有用的工具。由于篇幅有限，本书不会包含所有API路径列表，请参阅Spark文档中有关REST API路径的相关表格。

Spark UI历史记录服务器

通常情况下，Spark UI仅在SparkContext运行时可用，因此在应用程序崩溃或结束后如何才能使用Spark UI呢？为此，Spark提供了一个名为Spark历史记录服务器（Spark History Server）的工具，它允许你重建Spark UI和REST API，但是前提是应用程序配置为保存事件日志（event log）。你可以在Spark文档中找到有关如何使用此工具的最新信息。

使用历史记录服务器首先需要配置应用程序将事件日志存储到特定位置，这需要启用`spark.eventLog.enabled`和配置`spark.eventLog.dir`来指定事件日志的存放位置。一旦存储了事件，就可以将历史记录服务器作为独立应用程序运行，并且会根据这些日志自动重建Web UI。某些集群管理器和云服务也会自动配置日志记录，并默认运行历史记录服务器。

历史记录服务器还有许多其他配置，由于篇幅有限，本书不对Spark历史纪录服务器

的相关皮遏制做具体介绍，请参阅Spark文档中关于Spark历史记录服务器配置的相关表格。

调试和Spark抢救方案

前面的章节定义了一些Spark的核心“生命体征”，我们可以通过监控它们来检查Spark应用程序的健康状况。在本章的剩余部分，我们将解决Spark调试中遇到的问题：我们将总结执行Spark作业中表现出来的一些问题迹象和症状，包括你观察到的迹象（例如，执行缓慢的任务）以及Spark本身提示的症状（例如，`OutOfMemoryError`）。存在许多方面可能会影响Spark作业，因此本书不可能涵盖所有内容。但我们将讨论一些你可能遇到的比较常见的Spark问题，除了介绍问题可能的表现形式，我们还会针对这些问题给出一些可行的解决方案。

有关解决问题的建议可以参考第16章讨论的配置工具。

Spark作业未启动

这个问题是经常遇到的，尤其是在你部署一个新Spark集群之后或者迁移到一个新的硬件执行环境之后。

表现形式

- Spark作业无法启动。
- 除了驱动器节点，Spark UI不显示集群中其他执行器节点的信息。
- Spark UI报告疑似不正确的信息。

应对措施

这通常是由于你的集群或应用程序的资源需求配置不正确。Spark在分布式设置中对网络文件系统，和其他资源都有一定的假设，在设置集群的过程中，你可能会错误地执行了某些配置，导致驱动器节点无法与执行器节点进行通信，这可能是因为你没有正确打开某个指定的IP或端口，这很可能是集群和主机的配置问题。另一种可能是，应用程序为每个执行器进程请求了过多的资源，以至于大于集群管理器当前的空闲资源，在这种情况下，驱动器进程将永远等待执行器进程启动。

- 确保节点可以在指定的端口上相互通信。最好是打开工作节点的所有端口，除非你有严格的安全限制。

- 确保你的Spark资源配置正确，并且确保集群管理器已针对Spark进行了正确的配置。先尝试运行一个简单的应用程序，看看是否正常工作。一个常见的问题可能是，每个执行器进程被配置了太大的内存，而超过了集群管理器的内存资源配额。因此，请先（通过UI）检查内存资源是否足够，并且检查`spark-submit`的内存配置。

执行前错误

这种错误可能发生在：旧的应用程序可以运行，而当你在旧应用程序基础上添加了一些代码导致程序无法工作。

表现形式

- 命令不执行，并输出了大量错误消息。
- 通过Spark UI看不到任何作业、阶段或任务的运行。

应对措施

在检查并确认Spark UI的Environment选项卡显示应用程序的正确信息后，请仔细检查你的代码。很多时候，可能是一个简单的拼写错误或错误的列名称导致Spark作业不能编译到底层的Spark执行计划中。

- 请查看Spark返回的错误，并确认不是代码的问题，例如提供错误的输入文件路径或字段名称等是常见的代码错误。
- 反复仔细检查以确认集群的网络连接正常，检查驱动器节点、执行器节点，以及存储系统之间的网络连接情况。
- 可能是库或classpath的路径配置问题，导致加载了外部库的错误版本。请试着一步步删减代码以缩小错误的范围和定位错误，直到最后找到一个能够重现问题的较小的代码片段（例如，只读一个数据集）。

执行期间错误

如果你在遇到错误之前Spark应用程序已经在运行了，出现的错误属于执行期间错误。这可能是普通计划作业在执行一定时间之后遇到的错误，也可能是一个交互式作业在用户提交某个查询之后产生的错误。

表现形式

- 一个Spark作业在集群上成功运行，但下一个作业失败。
- 多步骤查询中的某个步骤失败。
- 一个已经成功运行过的程序在第二次运行时失败了。
- 难以解析错误信息。

应对措施

- 检查数据是否存在和输入数据的格式是否正确，输入数据可能会随时间而改变，这有可能对应用程序产生意想不到的后果。
- 如果在运行查询时（即启动任务之前）就弹出了错误，则在建立查询计划时可能出现了分析错误。这可能是因为查询中引用的列名称拼写错误，或者是因为你引用的列、视图或表不存在。
- 仔细读stack trace错误跟踪日志，以尝试找到某些组件错误的线索（例如，错误发生在哪个运算符和哪个阶段）。
- 用确保格式正确的输入数据集来隔离问题，排除数据的问题之后，也可以尝试删除部分代码逻辑，逐步缩减代码，直到找到一个可以产生错误的较小代码版本来定位问题。
- 如果作业运行一段时间然后失败，可能是由于输入数据本身存在问题，可能特定行的数据模式不正确。例如，有时你的模式指定数据不应该包含空值，但你的数据确实包含空值，这可能会导致某些转换操作失败。
- 你自己的代码在处理数据时也可能会崩溃。在这种情况下，Spark会向你显示代码抛出的异常，你将在Spark UI上看到标记为“失败”的任务，并且你还可以通过查看日志来了解在失败的时候正在做什么任务。尝试在代码中添加更多日志以确定哪个正在处理的数据记录有问题，这点会很有帮助。

缓慢任务或落后者

此问题在优化应用程序时非常常见，这可能是由于工作负载没有被均匀分布在集群各节点上（导致负载“倾斜”），或者是由于某台计算节点比其他计算节点速度慢（例如，由于硬件问题）。

表现形式

以下都可能是该问题的表现形式：

- Spark阶段中只剩下少数任务未完成，这些任务运行了很长时间。
- 在Spark UI中可以观察到这些缓慢的任务始终在相同的数据集上发生。
- 各阶段都有这些缓慢任务。
- 扩大Spark集群规模并没有太大的效果，有些任务仍然比其他任务耗时更长。
- 在Spark指标中，某些执行器进程读取和写入的数据量比其他执行器进程大得多。

应对措施

缓慢任务通常被称为“落后者”，有很多原因会导致缓慢任务，但最常见的原因是你的数据不均匀地分布到DataFrame或RDD分区上。发生这种情况时，一些执行器节点可能需要比其他执行器节点更多的工作量。一个特别常见的情况是，你使用按键分组操作，对应其中一个键的数据比其他键多得多。在这种情况下，当你查看Spark UI时，你会看到某些节点shuffle的数据比其他大得多。

- 尝试增加分区数以减少每个分区被分配到的数据量。
- 尝试通过另一种列组合来重新分区。例如，当你使用ID列进行分区时，如果ID是倾斜分布的，那么就容易产生落后者。或者当你使用存在许多空值的列进行分区时，许多对应空值列的行都被集中分配到一台节点上，也会造成落后者，在后一种情况下，首先筛选出空值可能会有所帮助。
- 尽可能分配给执行器进程更多的内存。
- 监视有缓慢任务的执行器节点，并确定该执行器节点在其他作业上也总是执行缓慢任务，这说明集群中可能存在一个不健康的执行器节点，例如，磁盘空间不足的节点。
- 如果在执行连接操作或聚合操作时产生缓慢任务，请参阅本章后面的“缓慢连接操作”和“缓慢聚合操作”。
- 检查用户定义函数（UDF）是否在其对象分配或业务逻辑中有资源浪费的情况。如果可能，尝试将它们转换为DataFrame代码。
- 确保你的UDF或用户定义的聚合函数（UDAF）在足够小的数据上可以运行。通常情况下，聚合操作要将大量数据存入内存以处理对某个key的聚合操作，从而

导致该执行器比其他执行器要完成更多的工作。

- 打开推测执行功能，这将为缓慢任务在另外一台节点上重新运行一个任务副本，关于该功能将在本章后面的“缓慢读写问题”中介绍。如果缓慢问题是由于硬件节点的原因，推测执行功能将会有所帮助，因为任务会被迁移到更快的节点上运行。然而，推测执行也会付出代价，第一是因为它会消耗额外的资源，另外，对于一些使用最终一致性的存储系统，如果写操作不是幂等的，则可能会产生重复冗余的输出数据（第17章讨论了推测执行的具体配置）。
- 使用Dataset时可能会出现另一个常见问题。由于Dataset执行大量的对象实例化并将记录转换为用户定义函数中的Java对象，这可能会导致大量垃圾回收。如果你使用Dataset，请查看Spark UI中的垃圾回收指标，以确定它们是否是导致缓慢任务的原因。

缓慢任务和落后者可能是调试中最困难的问题之一，因为有太多可能性会导致缓慢任务。然而，大多数情况是某种数据倾斜导致的，所以首先最好是在Spark UI上检查跨任务的不均衡负载。

缓慢的聚合操作

如果你的聚合操作速度较慢，请先查看“缓慢任务”部分的解决方案。尝试过那些之后，你可能会继续看到同样的问题。

表现形式

- 在执行groupby操作时产生缓慢任务。
- 聚合操作之后的作业也执行的非常缓慢。

应对措施

这个问题不能总是能够得到解决。如果你的作业中需要对存在数据倾斜的某个key执行聚合操作，那么如果你想在它们上执行聚合操作就是很慢。

- 在聚合操作之前增加分区数量可能有助于减少每个任务中处理的不同key的数量。
- 增加执行器进程的内存配额也可以帮助缓解此问题。如果一个key拥有大量数据，这将允许其执行器进程更少地与磁盘交互数据并更快完成任务，尽管它可能仍然比处理其他key的执行器进程要慢得多。

- 如果你发现聚合操作之后的任务也很慢，这意味着你的数据集在聚合操作之后可能仍然不均衡。尝试调用repartition并对数据进行随机重新分区。
- 确保涉及的所有过滤操作和select操作在聚合操作之前完成，这样可以保证只对需要执行聚合操作的数据进行处理，避免处理无关数据。Spark的查询优化器将自动为结构化API执行此操作。
- 确保空值被正确地表示（建议使用Spark的null关键字），不要用“ ”或“EMPTY”之类的空值表示。Spark优化器通常会在作业执行初期来跳过对null空值的处理，但它无法为你自己定义的空值形式进行此优化。
- 一些聚合操作本身也比其他聚合操作慢。例如，collect_list和collect_set是非常慢的聚合函数，因为它们必须将所有匹配的对象返回给驱动器进程，所以在代码中应该尽量避免使用这些聚合操作。

缓慢的连接操作

连接和聚合都需要数据shuffle操作，所以它们在问题的表现形式和对问题的处理方式上类似。

表现形式

- 连接操作阶段需要很长时间。可能是一项任务也可能是许多任务。
- 连接操作之前的阶段和之后的阶段似乎都很正常。

应对措施

- 许多连接操作类型可以被（手动或自动）转化到其他连接操作类型。第8章介绍了如何选择不同的连接操作类型。
- 试验不同的连接操作顺序是提高连接操作性能的一个方法，如果其中一些连接操作会过滤掉大量数据的话，应该先执行这些连接操作。
- 在执行连接操作前对数据集进行分区，这对于减少在集群之间的数据移动非常有用，特别是在多个连接操作中使用相同的数据集时，尝试不同的数据分区方法对提高连接操作性能是值得一试的。请注意数据分区不是“免费”的，而是以数据的shuffle操作为代价。
- 数据倾斜也可能导致连接操作速度变慢。如前所述，对于数据倾斜往往没有什么好办法，但是增加执行器节点的数量可能会有一定效果。
- 确保涉及的所有过滤操作和select操作在连接操作之前完成，这样可以保证只对

需要执行聚合操作的数据进行处理。

- 与聚合操作一样，使用`null`表示空值，而不是使用像`""`或`"EMPTY"`这样的空值表示。
- 如果Spark不知道有关输入DataFrame或输入数据表的统计信息，则无法选择合适的连接操作查询计划。如果你知道要执行连接操作的一个数据表很小，或使用Spark提供的数据统计命令能获得数据表的大小，则可以强制采用广播这个小数据表的方法实现连接操作（如第8章所述）。

缓慢的读写操作

缓慢的I/O可能很难诊断原因，尤其是对于网络文件系统。

表现形式

- 从分布式文件系统或外部存储系统上读取数据缓慢。
- 往网络文件系统或blob存储上写入数据缓慢。

应对措施

- 开启推测执行（将`spark.speculation`设置为`true`）有助于解决缓慢读写的问题。推测执行功能启动执行相同操作的任务副本，如果第一个任务只是一些暂时性问题，推测执行可以很好地解决读写操作慢的问题。推测执行是一个强大的工具，与支持数据一致性的文件系统兼容良好。但是，如果使用支持最终一致性的云存储系统，例如Amazon S3，它可能会导致重复的数据写入，因此请检查你使用的存储系统连接器是否支持。
- 确保网络连接状态良好。你的Spark集群可能因为没有足够的网络带宽而导致读写存储系统缓慢。
- 如果在相同节点上运行Spark和HDFS等分布式文件系统，确保Spark与文件系统的节点主机名相同。Spark将考虑数据局部性进行任务调度，用户将在Spark UI的“locality”列中看到该调度情况。我们将在下一章讨论更多关于数据局部性的问题。

驱动器的OutOfMemoryError错误或者驱动器无响应

这通常是一个非常严重的问题，它会使Spark应用程序崩溃。这通常是由于驱动器进程收集了过多的数据，从而导致内存不足。

表现形式

- Spark应用程序无响应或崩溃。
- 在驱动器进程的错误日志中发现OutOfMemoryErrors错误或垃圾回收消息。
- 命令需要很长时间才能完成或根本不运行。
- 交互性非常低或根本不存在。
- 驱动器程序JVM的内存使用率很高。

应对措施

发生这种情况有多种可能的原因，诊断出具体原因通常很难。

- 你的代码可能尝试使用诸如collect之类的操作将过大的数据集收集到驱动器节点。
- 你可能使用了广播连接，但是广播的数据太大。设置Spark的最大广播连接数可以更好地控制广播消息的大小。
- 应用程序长时间运行导致驱动器进程生成大量对象，并且无法释放它们。Java的*jmap*工具可以打印堆内存维护对象数量的直方图，这样可以查看哪些对象正在占用驱动器进程JVM的内存。但是请注意，运行*jmap*将需要暂停该JVM。
- 如果可能的话可以增加驱动器进程的内存分配，让它可以处理更多的数据。
- 由于两者之间的数据转换需要占用JVM中的大量内存，因此如果你使用其他语言绑定（如Python），JVM可能会发生内存不足的问题。请查看你的问题是否特定于你选择的语言，并将更少的数据带回驱动器节点，或者将其写入文件而不是将其作为内存中的对象返回。
- 如果你与其他用户（例如，通过SQL JDBC服务器和某些Notebook环境）共享SparkContext，请不要让其他用户在驱动器中执行可能导致分配大量内存的操作（例如，在代码中创建过大的数组，或者加载过大的数据集）。

执行器的OutOfMemoryError错误或执行器无响应

Spark应用程序有时候可以从这类错误中自动恢复，但是不一定，这取决于导致这种错误的根本原因。

表现形式

- 在执行器的错误日志中发现OutOfMemoryErrors或垃圾回收消息。这可以在Spark UI中找到它们。
- 执行器崩溃或无响应。
- 某些节点上的缓慢任务始终无法恢复。

应对措施

- 尝试增加执行器进程的可用内存和执行器节点的数量。
- 尝试通过相关的Python配置来增加PySpark工作节点的大小。
- 在执行器的日志中查找垃圾回收的错误消息。一些正在运行的任务，特别你使用了UDF的任务，可能会创建大量需要垃圾回收的对象，对数据进行重新分区以增加并行性，减少每个任务处理的记录数量，并确保所有执行者获得大体相同的工作负载。
- 正如我们前面讨论过的，确保使用null代表空值，而不应该使用""或"EMPTY"来表示空值。
- 在使用RDD或Dataset时，由于对象实例化可能会导致产生执行器的内存溢出错误。尽可能避免使用用户定义函数UDF，而尽可能使用Spark的结构化操作。
- 使用Java监视工具（如*jmap*）获取执行器的对象占用内存情况的直方图，查看哪类对象占用的空间最多。
- 如果执行器进程被放置在同时有其他工作负载运行的物理节点上，例如key-value存储，尝试将Spark作业与其他工作负载隔离。

结果返回意外的空值

表现形式

- 转换操作后，意外得到空值。
- 以前可以运行的生产作业不再正确运行，或不再产生正确的结果。

应对措施

- 在处理过程中数据格式可能已经更改，但是业务逻辑并没有及时调整。也就是说，以前的代码不再有效。

- 使用累加器对记录或某些类型的记录进行计数，以及在跳过记录时的解析或处理错误数。这可能会有所帮助，因为它会帮助你发现下面这样的问题，你认为你正在解析某种格式的数据，但实际上不是。大多数情况下，用户在将原始数据解析为某种格式时，会将累加器放在其UDF中，并在那里执行计数时。这使你可以计算有效的和无效的记录，并基于此信息调试程序。
- 确保你的转换操作解析成了正确的查询计划。Spark SQL有时会执行隐式强制类型转换，可能会导致错误结果。例如，SQL表达式`SELECT 5*"23"`的结果为115，因为字符串“25”以整数形式转换为值25，但表达式`SELECT 5 *""`的结果为`null`，因为将空字符串转换为整数会返回`null`。确保你的中间结果数据集具有正确模式（通过对数据使用`printSchema`检查），并在最终查询计划中检查所有`CAST`操作。

磁盘空间不足错误

表现形式

- 作业失败并显示“no space left on disk”错误。

应对措施

- 最简单的方法是增加更多的磁盘空间。在云环境中，你可以扩大集群规模（增加计算节点数量），或者挂载额外的存储。
- 如果集群的存储空间有限，由于数据倾斜，某些节点的存储可能会首先耗尽。如前所述对数据重新分区可能对此有所帮助。
- 还可以尝试调整存储配置。例如，有些配置决定日志在被彻底删除之前应该保留多长时间，可以缩短日志的保留时间以尽快释放日志占用的磁盘空间。有关执行器日志的更多信息，请参阅第16章。
- 尝试再相关机器上手动删除一些有问题的旧日志文件或旧shuffle文件。当然这只是个治标不治本的方法。

序列化错误

表现形式

作业失败并显示序列化错误。

应对措施

- 使用结构化API时，这种情况非常罕见。但当你使用UDF或RDD对执行器执行一些自定义逻辑时，或序列化到执行器的任务或共享的数据无法序列化时，就常会遇到序列化错误。当你处理无法序列化为UDF或函数的代码或数据时，或当你处理某些无法序列化的奇怪数据类型时，就会出现该错误。如果你正在使用（或打算使用Kryo序列化），请确认确实注册了你的类，以便它们能够被序列化。
- 在Java或Scala类中创建UDF时，不要在UDF中引用封闭对象的任何字段。因为这会导致Spark序列化整个封闭对象，这可能会产生错误。正确的做法是，将相关字段复制到与封闭对象相同作用域内的局部变量，然后再使用它们。

小结

本章介绍了一些可用于监控和调试Spark作业和应用程序的主要工具，以及一些最常见的错误和解决方案。与调试任何复杂的软件一样，我们建议基于调试原则一步一步地来找出错误所在。添加日志记录语句来确定程序在哪里崩溃，以及确定每个阶段处理的数据模式，使用隔离方法尽可能地将问题定位到最小的代码范围，并从那里开始找问题。对于并行计算特有的数据倾斜问题，可以使用Spark UI快速了解每个任务的负载。在第19章，我们将具体介绍使用各种工具来进行性能调优。

第19章

性能调优

第18章介绍了Spark用户界面（UI）和基本的错误应对方法，使用第18章介绍的工具，能够确保你的作业运行稳定。但是，有时你还需要它们运行得更快或更高效，这就是本章要介绍的内容，本章将讨论一些使Spark作业运行更快的有效方法。

正如监测一样，可以尝试从不同层面来优化程序。例如，如果Spark集群网络状况很好，这将使Spark作业运行的更快，因为依赖于网络性能的shuffle操作往往是Spark作业中开销最大的一个步骤。但是用户通常难以优化已有的网络环境，因此我们将更多地讨论通过代码优化或配置优化来提高Spark应用程序性能。

对于Spark作业，可以从多个方面进行优化，以下是一些优化方向：

- 代码级设计选择（例如，选择RDD还是DataFrame）。
- 静息数据。
- 连接操作。
- 聚合操作。
- 处理中的数据。
- 应用程序属性。
- 执行器节点的JVM。
- 工作节点。
- 集群部署属性。

这份清单并不全面，但它们基本包含了本章将要讨论的优化方向。此外，有两种方法可以指定Spark作业之外的执行特性：第一，可以通过合理配置和改变运行环境来间接地优化程序，这种方法会对此集群上的其他Spark应用程序或作业一起产生作用；第二，可以尝试直接指定某个Spark作业、某个阶段、或某个任务的执行特性，这些设置非常有针对性，不会对整体有什么影响。在直接优化和间接优化方面，各自都有很多内容会介绍。

提高性能最好的方法是实现良好的监测功能和作业历史跟踪功能，没有这些信息，就很难了解你的优化策略是否真的提高了性能。

间接性能优化

如前所述，你可以通过一些间接的方法来优化Spark作业的性能。我们不会介绍“改善硬件”之类显而易见的性能优化方法，而是更关注于介绍一些对普通用户来说更实际的方法。

设计选择

尽管正确的设计选择是优化性能的一种常见方式，但它通常不是首要考虑的优化方法。在设计应用程序时，做出正确的设计选择非常重要，因为它不仅有助于你写出更好的Spark应用程序，而且更有利于提高应用程序的健壮性，使其适应各种外部变化。前文中我们已经讨论了一些设计选择策略，但是本章会再次讨论一些最基本的设计选择问题。

Scala、Java、Python、R语言的选择

没有哪个语言始终是最好的，使用何种语言往往取决于实际应用。例如，如果你想在执行大规模ETL作业后执行一些单节点机器学习任务，我们会建议将你的抽取-转换-加载程序（Extract-Transform-Load，ETL）使用SparkR代码实现，然后利用R机器学习生态系统的支持，运行单节点机器学习算法。这可以获得R和Spark各自的优点，同时避免各自的缺点。正如我们多次提到的那样，Spark的结构化API在速度和稳定性方面对各种语言都一致，你可以使用你最喜欢的语言，或者选择最适合你应用程序的语言。

当你需要在结构化API中创建自定义转换操作时，事情会变得更加复杂，比如RDD转换或用户定义函数（UDF）。在这种情况下，因为实际的执行方式，R和Python并不一定是最好的选择。在定义跨语言函数时，严格保证类型和操作实现也更加困难。我

们发现，主要用Python来编写应用程序，然后在必要的情况下用Scala编写部分代码，或者用Scala编写UDF是一种很好的策略。它在整体可用性、可维护性和性能之间取得良好的平衡。

DaraFrame、SQL、Dataset、与RDD的选择

这个问题也经常出现，答案很简单。在所有语言环境下，DataFrame、Dataset和SQL在速度上都是相同的。这意味着如果你使用任何语言的DataFrame，性能是相同的。但是，在定义UDF时，如果使用Python或R编写的话会影响性能，用Java和Scala编写的话会好一些。如果你想纯粹地优化性能，那么你应该尝试使用DataFrame和SQL。虽然所有的DataFrame，SQL和Dataset代码都可以编译成RDD，但Spark的优化引擎会比手动编写出的RDD代码“更好”，并且这种避免手动编码对工作量的节约可是相当可观的。此外Spark SQL引擎发布新的版本时，你新添加的优化操作可能都会失效了。

最后，如果想使用RDD，我们推荐使用Scala或Java编写。如果这不可行，我们建议你将应用程序中调用RDD的次数限制到最低。这是因为当Python运行RDD代码时，大量数据将被序列化到Python进程或将从Python进程序列化出来。处理大规模数据时这个开销非常大，并且也会降低稳定性。

虽然它与性能调优并不完全相关，但需要注意的是，Spark的每种语言都支持哪些功能还有一些区别。我们在第16章曾讨论过。

RDD的对象序列化

在第Ⅲ部分中，我们简要讨论了可以在RDD转换中使用的序列化库。当你使用自定义数据类型时，你可能会使用Kryo对它们进行序列化，因为它比Java序列化更紧凑，效率更高。但是，使用Kryo进行序列化需要首先在你的应用程序中注册将要序列化的类，这是非常不方便的。

你可以通过将`spark.serializer`设置为`org.apache.spark.serializer.KryoSerializer`来使用Kryo序列化。你还需要通过`spark.kryo.classesToRegister`显式地注册Kryo序列化器的类。Kryo文档中还介绍了许多控制序列化过程的高级参数。

要注册你的类，请使用你刚创建的`SparkConf`并传入类的名称：

```
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
```

集群配置

集群配置可以显著地提升性能，但由于硬件和应用场景之间的差异，很难使用统一的策略。一般来说，监控机器本身的性能是优化集群配置的最有效的方法，尤其是当涉及在单个集群上运行多个应用程序（无论它们是否为Spark应用程序）时。

集群/应用程序的规模和共享

这可以归结为资源共享和调度问题。但是，在集群级别或应用程序级别如何共享资源存在许多选项，可以查看第16章最后和第17章的部分配置信息。

动态分配

Spark提供了一种机制，可根据工作负载动态调整应用程序占用的资源。这意味着你的应用程序可以在资源不再使用时将资源返回给集群，并在有需求时再次请求资源使用。如果多个应用程序共享Spark集群中的资源，此功能特别有用。此功能在默认情况下处于禁用状态，并在所有粗粒度集群管理器上可用，即独立模式、YARN模式和Mesos粗粒度模式。如果你想启用此功能，则应将`spark.dynamicAllocation.enabled`设置为`true`。Spark文档提供了许多可以调整的相关参数。

调度

在前几章中，我们讨论了许多不同的潜在优化策略，你可以利用调度池（Scheduler Pools）来优化Spark作业的并行度，或利用动态分配或设置`max-executor-cores`来优化Spark应用程序的并行度。调度优化方面确实有一些研究成果和实验结果，不幸的是，除了将`spark.scheduler.mode`设置为FAIR之外，还没有更好的解决方案来帮助多个用户更好的共享资源。也可以设置`--max-executor-cores`，它指定应用程序需要的最大执行核心数量，指定此值可确保你的应用程序不会占满集群上的所有资源。你还可以根据你的集群管理器更改默认值，方法是将配置`spark.cores.max`参数。集群管理器还提供了一些调度原语，这些原语在优化多个Spark应用程序时可能会有所帮助，如第16章和第17章中所讨论的。

静息数据

更为常见的情况是，当你保存数据时，你的组织中的其他人可以访问相同的数据集以进行不同的分析任务，因此它会被多次读取。确保你的数据以高性能读取方式存储，这对于一个成功的大数据项目来说是非常重要的，这包括选择存储系统、选择数据格式，以及针对某存储格式的数据分区功能。

基于文件的长期数据存储

有许多不同的文件格式，包括简单的CSV文件、二进制blob、以及如Apache Parquet这种更复杂的格式。优化Spark作业的最简单方法之一是在存储数据时选择最高效的存储格式。

一般而言，你应该始终支持结构化的二进制类型来存储数据，尤其是当你需要经常访问它时。尽管像“CSV”这样的文件看起来结构良好，但它们解析速度很慢，而且通常存在边界案例和某些痛点。例如，在阅读大量文件时，不恰当地忽略换行符通常会导致很多麻烦。一般来说，最有效的文件格式是Apache Parquet，Parquet将数据存储在具有列存储格式的二进制文件中，并且还跟踪有关每个文件的一些统计信息，以便快速跳过那些查询不需要的数据。另外Spark支持Parquet数据源，使其与Spark可以很好地集成。

可拆分的文件类型和压缩

无论选择哪种文件格式，都应确保它是“可拆分”（Splittable）的，这意味着不同的任务可以并行读取文件的不同部分。第18章介绍了为什么这是重要的，当我们读入文件时，所有核心都能够并行工作，就是因为该文件是可拆分的。如果我们没有使用可拆分的文件类型（比如JSON文件），我们将需要在单个机器上读取整个文件，这大大降低了并行性。

文件的可分割能力主要取决于压缩格式。ZIP文件或TAR归档文件不能被拆分，这意味着即使在ZIP中有10个文件，并且拥有10个核心，也只有一个核心可以读取该数据，因为我们无法并行访问ZIP文件，这是对资源的不良使用。相比之下，如果是由像Hadoop或Spark这样的并行处理框架输出的gzip、bzip2或lz4压缩文件，它们通常是可拆分的。对于你自己的输入数据，使其可拆分的最简单方法是将其作为单独的文件上传，理想情况下每个文件不超过几百兆字节。

表分区

第9章讨论过表分区，表分区是指将文件基于关键字存储在分开的目录中。像Apache Hive这样的存储管理器支持表分区，许多Spark的内置数据源也支持。正确地对数据进行分区，允许Spark在查询特定范围的数据时跳过许多不相关的文件。例如，如果用户在其查询中经常按“date”或“customerId”进行过滤，那就按这些列对数据进行分区，这将大大减少用户在大多数查询中读取的数据量，从而显着提高速度。

然而，分区的一个缺点是，如果分割粒度太细，可能会导致很多小文件，当列出存储系统中的所有文件时这会产生巨大开销。

分桶

第9章还讨论了分桶，分桶允许Spark根据可能执行的连接操作或聚合操作对数据进行“预先分区”。这可以提高性能和稳定性，因为分桶技术可以帮助数据在分区间持续分布，而不是倾斜到一两个分区。例如，如果在读取后频繁地根据某一列进行连接操作，则可以使用分桶来确保数据根据这一列的值进行了良好的分区。这可以减少在连接操作之前的shuffle操作，并因此有助于加快数据访问。分桶通常与分区协同工作，它是物理分割数据的第二种方式。

文件的数量

除了将数据分桶和分区之外，你还需要考虑文件的数量和存储文件的大小。如果有很多小文件，获得文件列表和找到每个文件的开销将非常大。例如，如果你正在从Hadoop分布式文件系统（HDFS）读取数据，每个数据块大小（默认情况）最大为128 MB。这意味着，如果你有30个文件，每个文件的实际内容大小为5 MB，尽管2个文件块就可以存储下这些文件内容（内容总共150 MB），但是也需要请求30个块。

尽管并没有一种万全之策来存储你的数据。但我们在下面了总结了可能的权衡方式。拥有大量小文件将使调度程序更难以找到数据并启动读取任务，这可能会增加作业的网络开销和调度开销。减少文件数量，让每个文件更大，则可以减轻调度程序的开销，但也会使任务运行更长时间。在这种情况下，你可以启动比输入文件个数更多的任务来增加并行性，Spark会将每个文件分割并分配个多个任务，前提是你使用的是可拆分格式。一般来说，我们建议你调整文件大小，使其每个文件至少包含几十兆字节的数据。

在编写数据时控制数据分区的一种方法是通过Spark 2.2版本引入的写入选项。要控制每个文件中有多少条记录，可以为写入操作指定`maxRecordsPerFile`选项。

数据局部性

另一个在共享集群环境中很重要的优化考虑是数据局部性。数据局部性指定了某些节点的存储偏好，哪些节点应该存储哪些数据，进而不必通过网络交换数据。如果你在运行Spark的相同集群机器上运行存储系统，并且系统支持局部性提示，则Spark将尝试调度与每个输入数据块在物理上更近的任务，例如HDFS存储就提供了这个选项。有几种配置会影响局部性，但如果Spark检测到它正在使用本地存储系统，则通常会默认使用它。你还将在Spark Web UI中看到标记为“local”的数据读取任务。

统计信息收集

Spark包含一个基于成本的查询优化器，它在使用结构化API时根据输入数据的属性来计划查询。但是，基于成本的优化器需要收集（并维护）关于数据表的统计信息。这些统计信息包含两类：数据表级的和列级的统计信息。统计信息收集仅适用于命名表，不适用于任意的DataFrame或RDD。

要收集表级统计信息，可以运行以下命令：

```
ANALYZE TABLE table_name COMPUTE STATISTICS
```

要收集列级统计信息，可以命名特定列：

```
ANALYZE TABLE table_name COMPUTE STATISTICS FOR
COLUMNS column_name1, column_name2, ...
```

列级别的统计信息收集速度较慢，但这些信息为基于成本的列级优化器提供了如何使用这些列的更多信息。这两种统计信息可以帮助优化连接操作、聚合操作、过滤操作，以及其他一些潜在的操作等（例如，自动选择何时进行广播连接。这是Spark的一个快速发展的部分，因此更多基于统计数据的不同优化在将来可能被加入到Spark中。

你可以在JIRA问题列表上跟踪基于成本优化的改进进度，你还可以阅读SPARK-16026设计文档以了解有关此功能的更多信息。在编写本书时，基于统计信息的优化是Spark项目进展非常快的一个方向。

Shuffle配置

配置Spark的外部Shuffle服务（在第16章和第17章中讨论过）一般情况下可以提高性能，因为它允许节点读取来自远程机器的Shuffle数据，即使这些机器上的执行器正忙于进行其他工作（例如，垃圾收集）。但是，这是以复杂性和维护为代价的，在实际部署中，这种策略可能并不值得。除了配置这个外部服务外，还有一些Shuffle配置，比如每个执行器的并发连接数，当然，这些配置通常都有很好的默认值。

另外，对于基于RDD的作业，序列化格式对Shuffle性能有很大影响，Kryo序列化通常比Java序列化更快。此外，对于所有Spark作业来说，Shuffle的分区数量都很重要。如果你的分区太少，那么只有少量节点在工作，这可能会造成数据倾斜。但是如果你有太多的分区，那么启动每一个任务的需要的开销可能会占据所有的资源。为了更好地平衡，你的Shuffle中为每个输出分区设置至少几十兆字节的数据。

内存压力和垃圾收集

在执行Spark作业的过程中，由于缺乏足够的内存或“内存压力”，执行器节点或驱动器节点可能难以完成其任务。当应用程序在执行过程中占用过多的内存，或者垃圾回收运行过于频繁，或者在JVM中创建了大量对象，而垃圾回收机制没有对这些对象及时回收的情况下，就会产生内存压力过大的情况。缓解此问题的一个策略是确保尽可能使用结构化API，这不仅会提高执行Spark作业的效率，而且还会大大降低内存压力，因为结构化API不会生成JVM对象，Spark SQL只是在其内部格式上执行计算。

Spark文档包含了一些关于RDD和UDF应用程序的垃圾回收机制调整的重要说明。

评估垃圾回收的影响

垃圾回收调优的第一步是统计垃圾回收发生的频率和时间。你可以使用`spark.executor.extraJavaOptions`配置参数添加`-verbose:gc -XX:+ PrintGCDetails -XX:+ PrintGCTimeStamps`到Spark的JVM选项来完成此操作。下次运行Spark作业时，每次发生垃圾回收时都会在工作节点的日志中打印相关信息，这些日志位于集群工作节点上（在其工作目录中的stdout文件中），而不在驱动器中。

垃圾收集调优

为了进一步对垃圾回收调优，你首先需要了解JVM中有关内存管理的一些基本信息：

- Java堆空间分为两个区域：新生代（Young）和老年代（Old）。新生代用于保存短寿命的对象，而老年代用于保存寿命较长的对象。
- 新生代被进一步划分为三个区域：Eden，Survivor1，以及Survivor2。

以下是对垃圾回收过程的简单描述：

- 当Eden区域满了时，在Eden上运行一个小型垃圾回收系统，Eden和Survivor1区域中活跃的对象被复制到Survivor2区域中。
- Survivor1区域与Survivor2区域交换。
- 如果一个对象足够旧，或者如果Survivor2区域已满，则该对象将移至老年代（Old）区域。
- 最后，当老年代区域接近装满时，将调用完整的垃圾回收。这涉及遍历堆中的所有对象，删除未引用的对象，以及移动其他对象以填充未使用的空间，所以它通常是最慢的垃圾回收操作。

Spark中垃圾回收调优的目标是确保只有长寿命的数据对象存储在老年代（Old）区域中，并且新生代（Young）区域的大小足以存储所有短期对象。这将有助于避免完整的垃圾回收处理在任务执行过程中创建的临时对象。以下是可能有用的一些步骤。

收集垃圾回收统计信息以确定它是否过于频繁运行。如果在任务完成之前多次调用完整的垃圾回收，则意味着没有足够的内存可用于执行任务，因此应该减少Spark用于缓存的内存大小（`spark.Memory.fraction`）。

如果有很多小型的垃圾回收但大型垃圾回收很少时，为Eden区域分配更多内存空间将会有所帮助。你可以将Eden区域的大小设置为略大于每个任务可能需要的内存总量。如果Eden区域的大小确定为E，则可以使用选项`-Xmn=4/3 × E`来设置Young区域的大小（比例系数4/3也是为了给Survivor区域腾出空间）。

例如，如果你的任务正在从HDFS中读取数据，则可以使用从HDFS中读取的数据块的大小来估计该任务使用的内存量。请注意，解压缩之后块的大小通常是压缩块大小的两倍或三倍。所以如果你想要三到四个任务的工作空间，而HDFS块大小为128 MB，我们可以估计Eden区域的大小为43128MB。

试着通过`-XX:+UseG1GC`选项来使用GIGG垃圾回收器。如果垃圾回收成为瓶颈，而且你无法通过调整Young和Old区域大小的方法来降低它的开销的话，那么使用GIGG可能会提高性能。注意，对于较大执行器的堆内存大小，使用`-XX:G1HeapRegionSize`增加G1区域大小非常重要。

监测垃圾回收的频率和所花费的时间如何随新设置的变化而发生变化。

我们的经验是，垃圾回收调优的效果取决于你的应用程序和可用内存量。关于垃圾回收调优的方法还有很多，你可以从网上找到，但在较高的层次上，设置完整垃圾回收的频率可以帮助减少开销。你可以通过在作业配置中设置`spark.executor.extraJavaOptions`来指定执行器的垃圾回收选项。

直接性能优化

在上一节中，我们谈到了适用于所有作业的一些通用的性能优化方法。在跳转到本节以及解决方案之前，一定要浏览前几页，这些解决方案旨在为特定阶段和作业的问题提供急救方案，但它们需要分别检查和优化每个阶段或每个作业。

并行度

如果需要加速某一个特定阶段，你应该做的第一件事就是增加并行度。通常，如果一个阶段需要处理大量的数据，我们建议你分配集群中每个CPU核心至少有两到三个任务。你可以通过`spark.default.parallelism`属性来设置它，并同时根据集群中的核心数量来调整`spark.sql.shuffle.partitions`属性。

过滤优化

提高性能的另一个常见手段是将过滤器尽可能移动到Spark任务的最开始。有时，这些过滤器可以被放置在数据的源头，这意味着你可以避免读取和处理与最终结果无关的数据。启用分区和分桶也有助于实现此目的。尽可能早的过滤掉大量数据，你会发现你的Spark作业几乎总是运行的更快。

重分区和合并

重新分区的调用可能会导致Shuffle操作。但是，但是通过平衡集群中的数据可以从整体上优化作业的执行，所以这个代价是值得的。一般来说，你应该试着Shuffle尽可能少的数据。因此，如果要减少DataFrame或RDD中整个分区的数量，请先尝试使用`coalesce`方法，该方法不会执行数据Shuffle，而是将同一节点上的多个分区合并到一个分区中。相对较慢的`repartition`方法会跨网络Shuffle数据以实现负载均衡，重新分区在执行连接操作之前或者调用`cache`方法之前调用会产生非常好的效果。请记住，重新分区并不是没有开销的，但它可以提高应用程序的整体性能和并行度。

自定义分区

如果你的作业仍然缓慢或不稳定，你可能需要尝试在RDD级别上执行自定义分区。你需要定义一个自定义分区函数，该函数将整个集群中的数据组织到比DataFrame级别更高的精度级别。这种方法通常是不需要的，但它是你提升性能的一个选择。有关更多信息，请参阅第Ⅱ部分。

用户定义函数

一般来说，避免使用UDF是一个很好的优化策略。UDF是昂贵的，因为它们强制将数据表示为JVM中的对象，有时在查询中要对一个记录执行多次此操作。你应该尽可能多地使用结构化API来执行操作，因为它们将以比高级语言更高效的方式执行转换操作。目前还有一些正在研究的工作可以批量向UDF提供数据，例如Python的Vectorized UDF扩展，它使用Pandas数据框（data frame）一次提供多条记录。我们在第18章中讨论了UDF及其成本。

临时数据存储（缓存）

在重复使用相同数据集的应用程序中，最有用的优化之一是缓存。缓存将DataFrame，数据表或RDD放入集群中执行器的临时存储区（内存或磁盘），这将使后续读取更快。虽然缓存可能听起来像我们应该一直做的事情，但这并不总是一件好事。这是因为缓存数据会导致序列化，反序列化和存储开销。例如，如果你只打算（在稍后的转换操作中）只处理这个数据集一次，缓存它只会降低速度。

缓存的案例很简单：当你在Spark中处理数据时，无论是在交互式会话还是独立应用程序中，你通常都希望重用某个数据集（例如DataFrame或RDD）。例如，在交互式数据科学会话中，你可能会加载并清理数据，然后重复使用它来尝试多个统计模型。或者在独立应用程序中，你可以运行重复使用相同数据集的迭代算法。你可以告诉Spark使用DataFrame或RDD上的cache方法来缓存数据集。

缓存是一种惰性的操作，这意味着只有在访问它们时才会对其进行缓存。 RDD API和结构化API在实际执行缓存方面有所不同，因此在我们介绍存储级别之前首先观察一些有趣的细节。当我们缓存RDD时，我们缓存实际的物理数据（即比特位）。当再次访问此数据时，Spark会返回正确的数据。这是通过RDD引用完成的。但是，在结构化API中，缓存是基于物理计划完成的。这意味着我们高效地将物理计划存储为我们的键（而不是对象引用）并在执行结构化作业之前执行查询。这可能会导致混乱（缓存访问冲突），因为有时你可能期望访问原始数据，但由于其他人已经缓存了数据，你实际上正在访问它们的缓存版本。在使用此功能时请记住这一特性。

你可以使用不同的存储级别来缓存数据，指定要使用的存储类型。表19-1列出了这些存储级别。

表19-1：数据存储级别

存储级别	意义
MEMORY_ONLY	将RDD存储为JVM中反序列化后的Java对象。如果内存装不下该RDD，则某些分区将不会被缓存，在每次需要时都会重新计算这些部分。这是默认的存储级别
MEMORY_AND_DISK_SER (Java and Scala)	将RDD存储为JVM中反序列化的Java对象。如果内存装不下该RDD，则再磁盘上存储内存装不下的分区，并在需要时从磁盘读取它们

表19-1：数据存储级别（续）

存储级别	意义
MEMORY_ONLY_SER (Java and Scala)	将RDD存储为序列化的Java对象（每个分区一个字节数组）。这通常比反序列化的对象更节省空间，尤其是在使用快速序列化工具时，但读取CPU开销更大
MEMORY_AND_DISK	与MEMORY_ONLY_SER类似，但将不适合内存的分区存储到磁盘，而不是每次需要时动态重新计算它们
DISK_ONLY	仅将RDD分区存储在磁盘上
MEMORY_ONLY_2，MEMORY_AND_DISK_2，etc。	与上一级别相同，但每个分区被复制到两个集群节点上
OFF_HEAP (experimental)	与MEMORY_ONLY_SER类似，但将数据存储在堆外内存中。这需要启用堆外内存

有关这些选项的更多信息，请参阅第16章的“内存管理配置”。

图19-1提供了一个简单的过程说明。我们从一个CSV文件加载一个初始的DataFrame，然后使用转换操作再从它派生出一些新的DataFrame。我们可以通过添加一行代码来缓存它们以避免重复计算原始的DataFrame（包括加载和解析CSV文件的重复计算）。

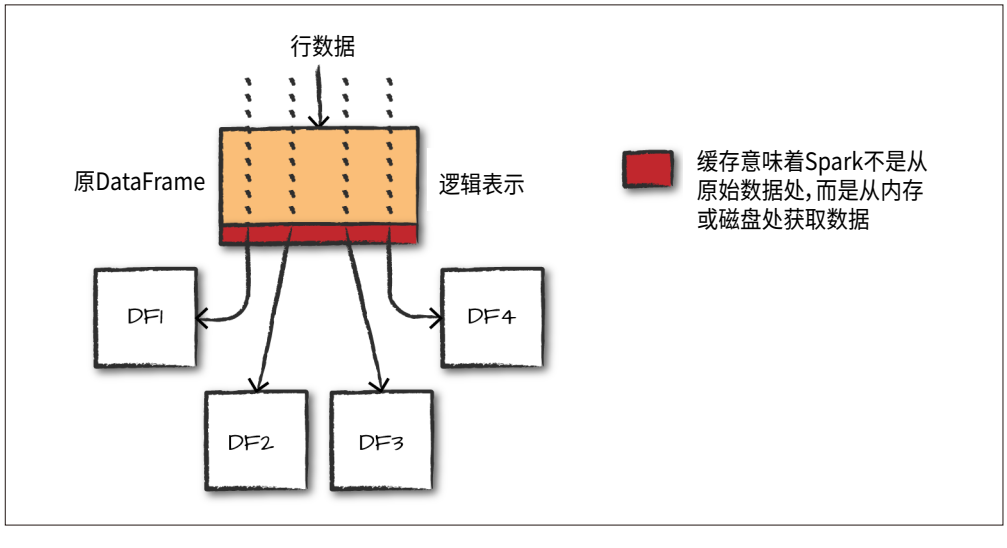


图19-1：一个缓存的DataFrame

现在我们来看看代码：

```
# in Python
# 原来的加载代码并不会缓存DataFrame
DF1 = spark.read.format("csv")\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .load("/data/flight-data/csv/2015-summary.csv")
DF2 = DF1.groupBy("DEST_COUNTRY_NAME").count().collect()
DF3 = DF1.groupBy("ORIGIN_COUNTRY_NAME").count().collect()
DF4 = DF1.groupBy("count").count().collect()
```

你会在这里看到我们的“惰性”创建的DataFrame（DF1）以及另外三个访问DF1中数据的DataFrame，所有的下游DataFrame都共享该父DataFrame（DF1），并在执行上述代码时重复相同的工作。在这种情况下，它只是读取和解析原始CSV数据，但这可能是一个相当繁重的过程，特别是处理大型数据集时。

在本书作者的机器上，这些命令需要一两秒钟才能运行结束。幸运的是，缓存可以帮助加快速度。当我们缓存DataFrame时，Spark会在第一次计算数据时将数据保存在内存或磁盘中。然后，当任何其他查询出现时，它们只会引用存储在内存中的DataFrame，而不是原始文件。你可以使用DataFrame的cache方法执行此操作：

```
DF1.cache()
DF1.count()
```

我们使用上面的count方法来触发缓存数据操作（执行一个Action动作操作来强制将它缓存在内存中）。这是因为缓存本身是惰性的，仅在你首次对DataFrame执行动作操作时才会真正地缓存数据。现在数据被缓存了，以前的命令会更快执行。下面运行以下代码：

```
# in Python
DF2 = DF1.groupBy("DEST_COUNTRY_NAME").count().collect()
DF3 = DF1.groupBy("ORIGIN_COUNTRY_NAME").count().collect()
DF4 = DF1.groupBy("count").count().collect()
```

当我们运行这个代码时，它将运行时间减少了一半以上！这可能看起来并不出众，但想象一下缓存一个大型数据集或需要大量计算才能创建的数据集（不仅仅是读取文件），节省的开销可能是相当可观的。这对于迭代机器学习工作负载也很有用，因为他们通常需要多次访问相同的数据，我们很快就会接触到迭代机器学习的任务。

Spark中的cache命令默认将数据置于内存中，如果集群的内存已满，则只缓存数据集的一部分。对于更多控制，还有一个persist方法，它通过StorageLevel对象指定缓存数据的位置，在内存中、磁盘上或两者混合。

连接操作

连接操作是最常需要优化的操作，当谈到优化连接时，最有用的就是了解每一种连接操作是做什么用的和怎么具体实现的，这对你最有帮助。此外，对于Spark而言，等值连接（equi-join）是最容易优化的，因此应尽可能优先选择等值连接。除此之外，其他简单的优化包括，通过改变连接顺序来让过滤发生在内部链接中，这可以极大的加快执行速度。另外，使用广播连接提示可帮助Spark在创建查询计划时做出智能的计划决策，如第8章描述的那样。避免笛卡尔连接或甚至避免完全外连接通常是提高稳定性和性能优化的简单方法，因为当你查看整个数据流而不仅仅是一个特定的工作区时，这些连接通常可以优化为不同过滤样式的连接。最后，本章其他小节介绍的技术也可以对连接性能产生重大影响。例如，在做连接操作之前收集数据表统计信息将有助于Spark做出智能的连接决策。此外，适当将数据分桶还可以帮助Spark在执行连接时避免大规模的shuffle操作。

聚合操作

大多数情况下，除了在聚合之前过滤数据之外，可以优化特定聚合操作的方法并不多。但是，如果你使用的是RDD，则精确地控制这些聚合的执行方式会非常有用（例如，用`reduceByKey`而不是`groupByKey`）。与此同时，这些策略还可以提高代码的速度和稳定性。

广播变量

我们在前几章中介绍了广播连接（Broadcast Join）和广播变量（Broadcast Variable），这些都是很好的优化选择。基本思路是，如果某一大块数据被程序中的多个UDF访问，则可以将其广播让每个节点上存储一个只读副本，并避免在每项作业中重新发送此数据。例如，把查找表（Lookup Table）或机器学习模型作为广播变量可能很有用。你也可以通过使用SparkContext创建广播变量来广播任意对象，然后只需在你的任务中引用这些变量，如我们在第14章所讨论的。

小结

有许多不同的方法可以优化Spark应用程序的性能，使其运行速度更快，成本更低。一般来说，你需要优先考虑的主要因素包括：①尽可能地通过分区和高效的二进制格式来读取较少的数据；②确保使用分区的集群具有足够的并行度，并通过分区方法减少数据倾斜；③尽可能多地使用诸如结构化API之类的高级API来使用已经优化过的

成熟代码。与其他任何软件优化工作一样，你也应该确保你是在优化正确的操作，第18章描述的Spark监视工具帮助让你了解哪些阶段花费时间最长，你需要将精力集中在这些阶段上。一旦确定了你认为可以优化的操作，就可以尝试本章介绍的各种性能优化方法。

第 V 部分

流处理

第20章

流处理基础

许多大数据应用都需要支持流处理。一旦应用程序计算出有价值的东西（比如一个客户行为报告或一个新的机器学习模型），公司就会想要在生产环境中连续不断地更新这个结果以获得更有价值的信息。因此，各大厂商都开始应用流处理技术，甚至在新应用的首个版本中就使用流处理。

幸运的是，Apache Spark一直支持流处理。在2012年，该项目合并了Spark Streaming及其DStream API，这是第一个能够使用例如`map`和`reduce`这样的高级函数进行流处理的API。现在，数以百计的组织在生产中使用DStreams来实现大型实时应用程序，通常每小时处理TB级的数据。虽然在很多方面与弹性分布式数据集(RDD) API相似，但是Dstream API是基于Java/Python对象的相对较低级别操作，它限制了更高级别的优化。因此，在2016年，Spark项目增加了结构化流处理（Structured Streaming），这是一个直接建立在DataFrame之上的新式流处理API，支持多种优化策略，并且可以很简单地与其他DataFrame和Dataset代码集成在一起。在Apache Spark 2.2中，结构化流处理API（Structured Streaming API）已经很稳定，并且在Spark社区中也得到了迅速地应用。

在本书中，我们将只关注结构化流处理API，它直接与我们在本书前面讨论的DataFrame和Dataset API集成在一起，同时它也是编写新式流处理应用程序的一种框架。如果你对DStream感兴趣，许多其他的书籍涵盖了这个API的功能介绍，包括一些关于Spark流处理的专门书籍，如Francois Garillot和Gerard Maas的《Learning Spark Streaming》(O'Reilly，2017)。然而，与RDD和DataFrame一样，本书重点介绍的结构化流处理包括了DStream的大多数功能，并且由于代码生成和Catalyst优化器，通常会表现更好。

在讨论Spark的流处理API之前，我们首先形式化地定义流处理和批处理。本章将讨论这一领域中的一些核心概念，虽然不能涵盖该领域的全部内容，但你将了解大部分概念，这些足够让你能够理解这个领域的系统。

什么是流处理？

流处理是连续处理新到来的数据以更新计算结果的行为。在流处理中，输入数据是无边界的，没有预定的开始或结束。它是一系列到达流处理系统的事件（例如，信用卡交易，单击网站动作，或从物联网IoT传感器读取的数据），用户应用程序对此事件流可以执行各种查询操作（例如，跟踪每种事件类型的发生次数，或将这些事件按照某时间窗口聚合）。应用程序在运行时将输出多个版本的结果，或者在某外部系统（如键值存储）中持续保持最新的聚合结果。

当然，我们可以将流处理与批处理（batch processing）进行比较，批处理是在固定大小输入数据集上进行计算的。通常，这可能是数据仓库中的大规模数据集，其包含来自应用程序的所有历史事件（例如，过去一个月的所有网站访问记录或传感器记录的数据）。批处理也可以进行查询计算，与流处理差不多，但只计算一次结果。

虽然流处理和批处理是不同的，但在实践中他们经常需要一起使用。例如，流式应用程序通常需要将输入流数据与批处理作业定期产生的数据连接起来，流式作业的输出通常是在批处理作业中要查询的文件或数据表。此外，应用程序中的任何业务逻辑都需要在流处理和批处理执行之间保持一致：例如，如果你有一个自定义代码来计算用户的计费金额，而流处理和批处理运行出来的结果不同那就出问题了!为了满足这些需求，结构化流处理从一开始就设计成可以轻松地与其他Spark组件进行交互，包括批处理应用程序。结构化流处理开发人员提出了一个叫做连续应用程序（continuous application）的概念，它把包括流处理、批处理和交互式作业等全部作用在同一数据集上的处理环节串联起来，从而构建端到端应用程序，提供最终的处理结果或产品。结构化流处理注重使用简单的端到端的方式构建此类应用程序，而不是仅仅在流数据级别上针对每条记录进行处理。

流处理应用场景

我们将流处理定义为在无边界数据集上的增量处理，但显然这并不能直观的体现流处理的应用场景，也不能体现其必要性。在我们介绍流处理的优缺点之前，先来了解一下为什么你需要使用流处理，我们将描述来自底层流处理系统中具有不同需求的六个常见应用场景。。

通知和警报

可能最明显的流处理应用场景是通知和警报。给定一系列事件，如果发生某一事件或一系列连续特殊事件，则应触发通知或警报。这不一定意味着触发自动决策或者触发预先制定的决策，警报还可以通知人们以采取一些必要的行动。比如，流处理系统向仓库管理员发出警报，提示他们需要从仓库中的某个位置取出货物，并将其装运给客户，在任何情况下，这类通知都需要及时。

实时报告

许多组织使用流处理系统来运行员工可以实时查看的动态仪表板。例如，这本书的作者每天都要使用Databricks 公司（本书两位作者都受雇于该公司）的实时报告仪表板，该仪表盘即利用结构化流处理运行。我们使用这些仪表板来监视整个平台的使用情况、系统负载、在线时间，甚至是新功能在推出时以及其他应用程序中的使用情况。

增量 ETL

最常见的流处理应用程序之一是减少公司在信息检索时必须忍受的延迟时间，简而言之，“把批处理任务用流处理方式执行”。Spark批处理作业通常用于抽取-转换-加载任务（ETL任务，Extract-Transform-Load），将原始数据转换为结构化格式 (如Parquet) 以支持高效查询。使用结构化流处理，这些作业可以在几秒钟内处理新数据，使下游用户能够更快地查询信息。在这个应用场景中，关键是一次性地以容错的方式处理数据： 我们不希望它在加载到数据仓库之前丢失任何输入数据，而且我们不想两次加载相同的数据。此外，流处理系统需要支持数据仓库的事务级别的更新操作，以避免查询结果与部分写入数据的冲突。

实时数据更新来提供实时服务

流处理系统经常用于为另一个应用程序提供实时的数据处理结果。例如，类似谷歌分析（Google Analytics）这样的Web分析产品可能会持续跟踪每个网页的访问次数，并使用流处理系统将这些计数持续更新，当用户与产品UI交互时，此Web应用程序将查询最新的计数。支持此用例需要流处理系统对 key–value 存储 (或其他服务系统)执行同步的增量更新，而且通常与ETL案例中的一样，这些更新需要是事务性的以保证应用程序的数据一致性。

实时决策

流处理系统的实时决策包括分析新的输入，根据业务逻辑自动作出决策来应对新数

据。比如，一个银行希望自动验证客户信用卡上的新交易记录是否为一个可疑交易，这种判断可以基于其最近的历史记录，如果认定可疑的盗刷行为则拒绝交易。此决策需要在处理每个交易时实时进行，因此开发人员需要在流处理系统中实现此业务逻辑，并用于处理交易信息的数据流。这种类型的应用程序需要维护每个用户的大量状态，以跟踪其当前的支出模式，并自动将这些状态与每个新交易进行比较。

在线机器学习

与实时决策用例相近的是在线机器学习。在这种情况下，你可能希望基于多个用户的流和历史数据的组合来训练模型。这个示例可能比前面提到的信用卡交易用例更复杂，因为公司可能希望基于所有客户行为来不断更新模型并基于它验证每笔交易，而不是去根据一个客户的行为采用固定规则来做出响应。这是流处理系统中最具挑战性的应用场景，因为它需要对多个客户的数据进行聚合操作、与静态数据集执行连接操作、与机器学习库集成等，并同时需要降低延迟响应时间。

流处理的优点

我们已经了解了一些使用流处理的应用场景，下面具体介绍一下流处理的优点。在大多数情况下，批处理更容易理解、更容易调试、也更容易编写应用程序。此外，批量处理数据也使得数据处理的吞吐量大大高于许多流处理系统。然而，流处理对于以下两种情况非常必要。首先，流处理可以降低延迟时间：当你的应用程序需要快速响应时间（在分钟、秒或毫秒级别上），你需要一个可以将状态保存在内存中的流处理系统，以获得更好的性能，我们介绍的许多决策和警报案例都属于这种情况。其次，流处理在更新结果方面也比重复的批处理作业更有效，因为它会自动增量计算。例如，如果我们要计算过去24小时内的 Web 流量统计数据，那么简单的批处理作业实现可能会在每次运行时遍历所有数据，总是处理24小时内的数据。与此相反，流处理系统可以记住以前计算的状态，只计算新数据。如果你告诉流处理系统个小时更新一次报告，则每次只需处理1小时的数据 (自上次报告以来的新数据)。在批理系统中，你需要手动实现这种增量计算，以获得相同的性能，从而导致要做大量额外的工作，而这些工作在流处理系统中会自动完成。

流处理的挑战

我们讨论了流处理的应用场景和优点，但你可能知道，从来没有免费的午餐。让我们讨论一下流处理的一些挑战。

为了更好地说明，我们举个实际例子。假设我们的应用程序接收来自传感器（例如，

汽车内部）的输入消息，该传感器在不同时间报告其值。然后，我们希望在该数据流中搜索特定值或特定模式，一个挑战就是输入记录可能会无序地到达应用程序。例如，因延迟和重新传输，我们可能会收到以下顺序的更新序列，其中time字段显示实际测量的时间：

```
{value: 1, time: "2017-04-07T00:00:00"}
{value: 2, time: "2017-04-07T01:00:00"}
{value: 5, time: "2017-04-07T02:00:00"}
{value: 10, time: "2017-04-07T01:30:00"}
{value: 7, time: "2017-04-07T03:00:00"}
```

在任何数据处理系统中，我们都可以构造逻辑来执行一些基于接收单值“5”的操作。在流处理系统中，我们也可以快速响应此单个事件。然而，如果你只想根据收到的特定值序列触发某个动作，比如2，然后10，接着5，事情就复杂多了。在批处理的情况下，这并不特别困难，因为我们可以简单地按time字段对所有事件进行排序，以发现10在2和5之间到来。然而，对于流处理系统来说，这是比较困难的。原因是流处理系统将单独接收每个事件，并且需要跟踪事件的某些状态以记住值为2和5的事件，并意识到值为10的事件是在它们之间，这种在流中记住事件状态的需求带来了更多的挑战。例如，如果你有一个海量的数据集(例如，数以百万计的传感器流) 并且状态本身是巨大的，应该如何处理？如果系统中的机器出现故障了，丢失了一些状态怎么办？如果负载不平衡，一台机器的速度非常慢怎么办？当对某些事件的分析完成之后，你的应用程序如何向下游消费者发出信号 (例如，模式 2-10-5没有发生)？是否应该等待固定的时间或无限期地记住某个状态？当你希望部署流式应用程序时，所有这些挑战和其他问题 (例如处理事务性的输入和输出) 都会出现。

总结一下，我们在前面几段所述的挑战如下：

- 基于应用程序时间戳(也称为事件时间) 处理无序数据。
- 维持大量的状态。
- 支持高吞吐量。
- 即使有机器故障也仅需对事件进行一次处理。
- 处理负载不平衡和拖延者（straggler）。
- 快速响应时间。
- 与其他存储系统中的数据连接。
- 确定在新事件到达时如何更新输出。

- 事务性地向输出系统写入数据。
- 在运行时更新应用程序的业务逻辑。

这些主题都是大型流处理系统研究和开发的热点问题。为了了解不同的流处理系统如何应对这些挑战，我们将介绍一些最常见的设计概念。

流处理设计要点

为了应对我们所描述的流处理难题，包括高吞吐量、低延迟时间和无序数据处理，有多种方法来设计流处理系统。在下一节中描述Spark的选择之前，我们在这里描述最常见的设计选项。

记录级别API与声明式API

设计流处理API 最简单的方法就是将每个事件传递给应用程序，并使用自定义代码进行响应。这是许多早期的流处理系统 (如 Apache Storm) 的实现方法，当应用程序需要完全控制数据处理时，它具有重要的地位。提供这种记录级别API（record-at-a-time API）的流处理系统只是给用户提供一个获取每条流数据记录的接口，然而这些系统的缺点是，我们前面描述的大多数复杂因素 (如状态维护) 都要完全由应用程序负责。例如，使用记录级别API，用户需要负责在较长时间段内跟踪状态，在一段时间后删除它以清除空间，并在发生故障后以不同方式响应重复事件。在这些系统上实现正确的流处理程序相当困难。从本质上讲，像记录级别API这种低级 API 需要深厚的专业知识才能开发和维护。

因此，许多后来出现的流处理系统提供了声明式API，应用程序为了响应每个新事件指定要计算的内容，而不是如何计算，也不需要考虑如何从失败中恢复。例如，Spark的原始 DStream API 提供了基于map、reduce和filter等数据流上操作的功能性API。在内部，DStream API自动跟踪每个操作处理的数据量，可靠地保存相关状态，并在需要时从失败中恢复计算，诸如 Google DataFlow和 Apache Kafka Stream等系统也提供了类似的功能性API。Spark的结构化流处理则更进一步，从功能操作切换到关系操作(类似 SQL)，从而在不需要编程的情况下实现更丰富的自动执行优化。

基于事件时间与基于处理时间

对于具有声明性 API 的系统，第二个问题是系统是否支持事件时间。事件时间（event time）是根据数据源插入记录中的时间戳来处理数据的概念，而不是流处理应用程序

在接收记录时的时间（称为处理时间，processing time）。特别是，当使用事件时间时，记录可能会出现无序状况(例如，如果记录从不同的网络路径返回)，并且不同的数据源可能也无法同步（对于标记相同事件时间的记录，某些记录可能比其他记录晚到达）。如果你的应用程序从可能产生延迟的远程数据源 (如移动电话或物联网设备)收集数据，则基于事件时间的处理方式就非常必要。如果不基于事件时间，你可能在某些数据延迟到达时无法发现某些重要的模式。相比之下，如果应用程序只处理本地事件 (例如，在同一个数据中心中生成的数据)，则可能不需要复杂的事件时间处理。

在使用事件时间时，有几个问题需要重点关注，包括以允许系统合并延迟事件的方式跟踪状态，以及确定何时在事件时间内输出给定时间窗口的结果是安全的（即当系统已收到某事件时间点之前的所有输入）。因此，许多声明性系统（包括结构化流处理）对其所有API中集成的事件时间都具有“原生”支持，因此可以在整个程序中自动处理这些问题。

连续处理与微批量处理

最后一个设计选择通常是确定连续处理（Continuous Processing）模式还是微批量处理（Micro-batch Processing）模式。在基于连续处理的系统中，系统中的每个节点都不断地侦听来自其他节点的消息并将新的更新输出到其子节点。例如，假设你的应用程序在多个输入流上实现了map-reduce运算。在连续处理系统中，实现map的每个节点将从输入数据源一个一个地读取记录，根据函数逻辑执行计算，并将它们发送到相应的reducer。当reducer获得新记录时，将更新其状态。最关键的是这种情况是发生在每个记录上的，如图 20-1所示。

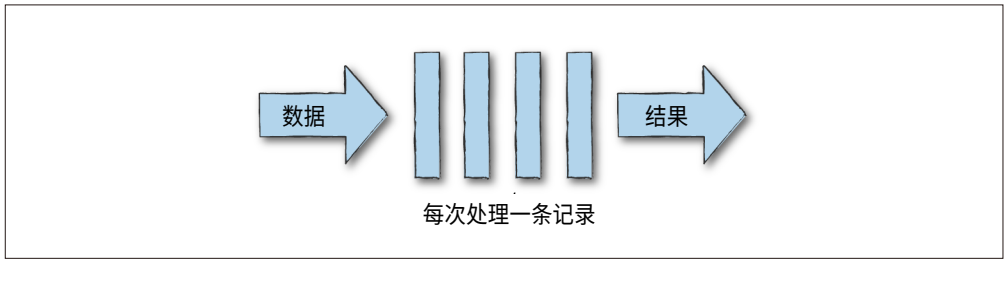


图20-1：连续处理

在输入数据速率较低时，连续处理提供尽可能低的处理延迟，这是连续处理的优势，因为每个节点都会立即响应新消息。但是，连续处理系统通常具有较低的最大吞吐量，因为它们处理每条记录的开销都很大 (例如，调用操作系统将数据包发送到下游

节点)。此外，连续处理系统通常具有固定的计算拓扑，如果不停止整个系统，在运行状态下是不能够改动的，这也可能会导致负载均衡的问题。

相比之下，微批量处理系统等待积累少量的输入数据（比如500ms的数据），然后使用分布式任务集合并行处理每个批次，类似于在Spark中执行批处理作业。微批量处理系统通常可以在每个节点上实现高吞吐量，因为它们利用与批处理系统相同的优化操作 (例如，向量化处理，Vectorized Processing)，并且不会引入频繁处理单个记录的额外开销，如图 20-2所示。

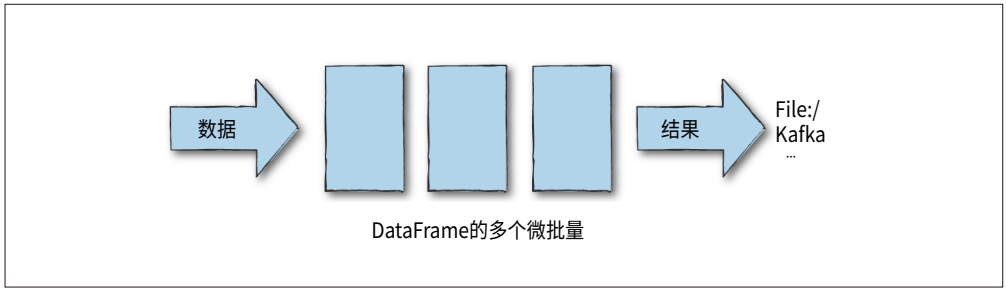


图20-2：微批量处理

因此，微批量处理只需要较少的节点就可以应对相同生产速率的数据。微批量处理系统还可以使用动态负载平衡技术来应对不断变化的工作负载 (例如，增加或减少任务数量)。然而，缺点是由于等待累积一个微批量而导致更长的响应延迟。在实际生产中，在处理大规模数据并需要分布式计算的流处理应用程序往往优先考虑吞吐量，因此Spark最开始就实现了微批量处理。但是在结构化流处理中，正在积极开发基于相同API来支持连续处理模式。

在这两种执行模式之间进行选择时，最主要因素是你期望的延迟和总的操作成本(TCO)。根据应用程序的不同，微批量处理系统可能将延迟从100毫秒延长到秒。在这种机制下，通常需要较少的节点就可以达到要求的吞吐量，因而降低了运营成本 (包括由于节点故障次数较少带来的维护成本降低)。为了获得更低的延迟，你应该考虑一个连续处理系统，或使用一个微批量处理系统与快速服务层（fast serving layer）结合以提供低延迟查询（例如，将数据输出到 MySQL 或 Apache Cassandra，可以在几毫秒内将其提供给客户端）。

Spark的流处理API

我们已经讨论了流处理的一些高级设计方法，但到目前为止，我们还没有详细讨论

Spark的流处理 API。正如在本章开头说的那样，Spark包括两种流处理 API，Spark早期的 DStream API 纯粹是基于微批量处理模式的，它是声明式 (基于功能的) API，但不支持事件时间。新的结构化流处理 API添加了更高级别的优化、事件时间，并且支持连续处理。

DStream API

Spark原始的DStream API自2012年首次发布以来已被广泛应用。例如，在Datanami 2016 调查结果中DStream是使用最广泛的流处理引擎。由于高级 的API 接口和简单的语义，许多公司在大规模实际生产中都采用Spark流处理。与 RDD 代码的交互（如与静态数据的连接）也是Spark流处理原生支持的。运行Spark流处理并不比运行正常的Spark作业困难。但是，DStream API 有几个限制。首先，它完全基于Java/Python 对象和函数，而不是DataFrame和Dataset中丰富的结构化表概念，这不利于执行引擎进行优化。其次，DStream API 纯粹基于处理时间（要处理事件时间操作，应用程序需要自己实现它们）。最后，DStream只能以微批量方式运行，并在其API的某些部分暴露了微批量的持续时间，这使得其难以支持其他执行模式。

结构化流处理

结构化流处理是基于Spark结构化API的高级流处理 API，它适用于运行结构化处理的所有环境，例如Scala，Java，Python，R 和 SQL。与DStream一样，它是基于高级操作的声明式 API，但是，通过构建本书上一部分介绍的结构化数据模型，结构化流处理可以自动执行更多类型的优化。但是，与 DStream不同，结构化流处理对标记事件时间的数据具有原生支持 (所有窗口操作都自动支持它)。在Apache Spark 2.2中，系统只支持微批量模型，但Databricks 的Spark团队正在扩展对连续处理的支持，最终将支持连续处理的执行模式，这应该在Spark 2.3 版本上支持。

更重要的是，除了简化流处理之外，结构化流处理还旨在使用Apache Spark轻松构建端到端连续应用程序，这些应用程序结合了流处理、批处理和交互式查询。例如，结构化流处理不使用 DataFrame之外的 API，只需编写一个正常的 DataFrame (或 SQL) 计算并在数据流上应用它。当数据到达时，结构化流处理将以增量方式自动更新此计算的结果。这非常有益于简化编写端到端数据处理应用程序，开发人员不需要维护批处理代码版本和流处理版本，并且避免这两个版本代码失去同步的风险。另外一个例子是，结构化流处理可以将数据输出到由Spark SQL (如Parquet表) 可用的标准接收器，从而便于从另一个Spark应用程序查询你的流状态。在未来版本的 Apache Spark

中，我们期待越来越多的项目组件与结构化流处理集成进来，包括 MLlib 中的在线学习算法。

总的来看，结构化流处理易于使用且更加高效，它是Spark流处理DStream API 的演化，因此我们将只关注更新的结构化流处理API。许多概念，例如从转换图中生成计算，也适用于 DStream，但我们不在本书中介绍，你可以参考其他书籍。

小结

本章介绍了流处理的基本概念和思路，本章介绍的设计方法帮助读者了解如何针对给定应用程序设计流处理系统。你也应该理解了DStream和结构化流处理的作者做了哪些权衡，以及使用结构化流处理时直接支持 DataFrame为什么如此重要，因为它避免了重写应用程序逻辑。在接下来的章节中，我们将深入介绍结构化流处理，了解如何使用它。

第21章

结构化流处理基础

我们已经简要介绍了流处理，下面将介绍结构化流处理。在本章中，我们将再次介绍结构化流处理背后的一些关键概念，然后将介绍一些代码示例，以展示系统的易用性。

结构化流处理基础

我们在第20 章结束时讨论过，结构化流处理是建立在Spark SQL 引擎上的流处理框架。结构化流处理不是独立的 API，而是使用了Spark中现有的结构化 API (包括DataFrame、Dataset和 SQL)，这意味着你熟悉的所有操作都支持。用户可以用在静态数据上批处理的计算方式来表达流处理计算。在指定流处理操作后，结构化流处理引擎将负责对新到达系统的数据执行增量地、连续地查询。用于计算的逻辑指令将在本书第Ⅱ部分讨论的Catalyst引擎上执行，包括查询优化、代码生成等。除了核心结构化流处理引擎之外，结构化流处理还包括一些专门用于流处理的特性。例如，结构化流处理支持仅执行一次的端到端处理，并且支持基于检查点和预写式日志（write-ahead log）的错误恢复功能。

结构化流处理背后的主要思想是将数据流视为连续追加数据的数据表。然后，该作业定期检查新的输入数据，对其进行处理，在需要时更新位于状态存储区的某些内部状态，并更新其结果。API的设计原则是，批处理和流处理的查询代码是一致的，不需要更改查询代码，你只需要指定是以批处理还是以流处理方式运行该查询即可。在内部，结构化流处理将自动找出“增量”查询的方法，即在新数据到达时有效地更新其结果，并以容错的方式运行。如图21-1所示。

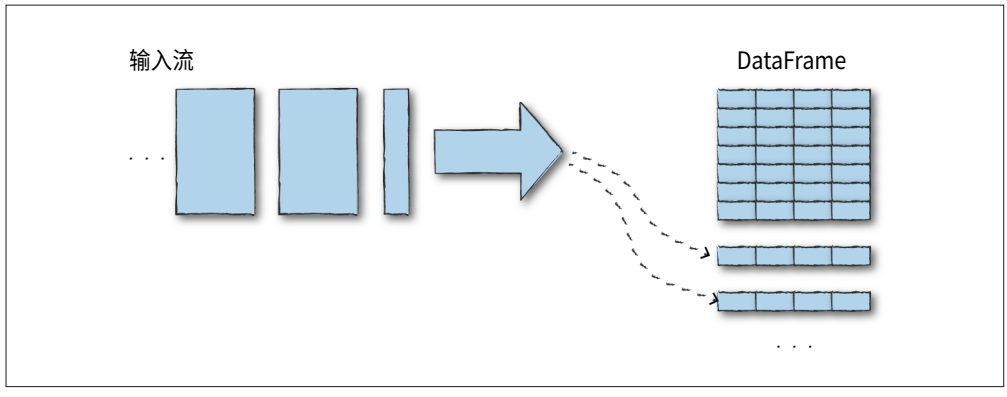


图21-1：结构化流处理的输入

简单来说，结构化流即是“以流处理方式处理的DataFrame”，这使得使用流处理应用程序非常容易，因为你可能已经有了流处理的代码！但是，结构化流处理的查询类型有一定的限制，以及你必须考虑到一些特定于流的新概念，例如事件时间和无序数据。我们将在本章和下面的章节中讨论这些内容。

最后，通过与Spark其余部分集成，结构化流处理使用户能够构建连续应用程序。连续应用程序是一个端到端的应用程序，它通过组合各种工具来实时对数据作出响应，包括流处理作业、批处理作业、流数据和离线数据之间的连接，以及交互式的ad-hoc查询（即席查询）。由于当前大多数流处理作业都是在更大的连续应用程序中部署的，因此Spark开发者试图使其在一个框架中简单地指定整个应用程序，并在这些不同部分的数据获得一致的结果。例如，你可以使用结构化流处理连续地更新一个（用户通过Spark SQL 执行交互式查询的）数据表，或者为 MLlib 训练的机器学习模型提供流处理服务，或者将数据流与在任一Spark数据源中离线数据连接，使用不同工具的组合来构建流处理也可以，但是也会使得应用程序更复杂。

核心概念

既然我们已经介绍了高级概念，下面来讨论结构化流处理作业中的一些重要概念。你会发现并没有太多内容，这是因为结构化流处理设计地很简单。阅读一些其他大数据流处理的书籍，你会注意到，他们一开始就引入术语，如针对倾斜数据的分布式流处理拓扑（这是讽刺但很准确）和其他复杂的措辞。Spark的目标是自动处理这些问题，让用户以简单的方式运行Spark流处理程序。

转换操作和动作操作

结构化流处理同样涉及转换操作和动作操作的概念，结构化流处理的转换操作就是第Ⅱ部分中的转换操作，虽然有一些限制，这些限制通常是在引擎无法增量执行某些类型的查询上，但一些限制会在新版本的Spark中去除。在结构化流处理中只有一个动作操作，即启动流处理，然后运行并持续输出结果。

输入源

Spark支持从多种输入数据源以流式读取数据进行结构化流处理。在Spark 2.2版本中，支持的输入源包括：

- Apache Kafka 0.10。
- 类似HDFS 或 S3 这样的分布式文件系统 (Spark将持续读取某目录下的新文件)。
- 用于测试的套接字（socket）源。

本章后面我们将深入讨论这些内容，但值得一提的是，Spark的作者正在开发稳定的数据源 API，这样你就可以用它来构建自己的数据流连接器。

接收器

你需要指定数据源从其读取数据流，也需要指定接收器来设置流处理之后的结果集的去处。接收器（sink）和执行引擎还负责可靠地跟踪数据处理的进度。下面是Spark 2.2版本中支持的接收器：

- Apache Kafka 0.10。
- 几乎任何文件格式的文件。
- 用于在输出记录上运行任意计算的 foreach 接收器。
- 用于测试的控制台接收器。
- 用于调试的内存接收器。

我们将在本章稍后讨论源代码时更详细地讨论这些内容。

输出模式

为结构化流处理作业指定一个接收器只解决了一半问题，还需要定义Spark将数据以何种方式写入接收器。例如，是否只想追加新信息？当我们会随着时间的推移收到有

关它们的更多信息时，是否希望更新数据记录（例如，更新给定网页的点击次数）？是否希望每次都完全覆盖结果集 (即始终使用总点击数更新文件)？为此，我们需要定义了一个输出模式，类似于我们如何定义静态结构化 API 中的输出模式。Spark支持的输出模式如下：

- append（追加，只向输出接收器中添加新记录）。
- update（更新，更新有变化的记录）。
- complete（完全，重写所有输出）。

值得注意的是，某些查询和某些接收器只支持某些固定的输出模式，这将在后面讨论。例如，假设你的作业只是在流上执行`map`操作，随着新记录的到达，输出数据将无限地增长，因此使用complete模式，将所有数据一次性重新写入一个新的文件，则是没有意义的。相比之下，如果你正在针对有限数量的键进行聚合操作，则complete模式和update模式是有意义的，但append模式不行，因为某些键的值需要随着时间的推移不断更新。

触发器

输出模式定义了数据以何种方式被输出，触发器则定义了数据何时被输出，即结构化流处理应何时检查处理新输入数据并更新其结果。默认情况下，结构化流处理将在处理完最后一组输入数据后立即检查是否有新的输入记录，从而尽力保证低延迟。但是，当接收器是一组文件时，可能产生许多小输出文件。因此，Spark还支持基于处理时间的触发器（trigger，仅在固定时间间隔内检查新数据），将来还可能支持其他类型的触发器。

事件时间处理

结构化流处理也支持事件时间处理（event-time processing），根据可能无序到达的记录中标注的时间戳进行数据处理。现在你将需要了解两个关键点，我们将在下一章中更深入地讨论这两个问题。所以如果你现在还不完全清楚，也不必担心。

事件时间数据

事件时间（event-time）表示嵌入到数据中的时间字段。这意味着，不是根据它到达系统的时间处理数据，而是根据生成数据的时间对其进行处理，即使由于上传速度慢或网络延迟导致流处理应用程序中的输入记录乱序到达。在结构化流处理中实现事件时间处理很简单，由于系统将输入数据视作数据表，因此事件时间只是该表中的一个

字段，应用程序可以使用标准 SQL 运算符进行分组、聚合和窗口化。但是，在内部当它知道你的其中一列是事件时间字段时，结构化流处理可以采取一些特殊的操作，包括进行查询优化或确定何时可以安全地忘记某时间窗口的状态。许多这些操作可以使用watermarks进行控制。

watermarks

watermarks是流处理系统的一项功能，它允许你指定在事件时间内查看数据的延迟程度。例如，某个应用程序需要处理来自移动设备的日志，由于上传延迟可能会导致日志延迟30分钟。支持事件时间 (包括结构化流处理) 的系统通常允许设置watermarks来限制它们记住旧数据的时长。watermarks还可用于控制何时输出特定事件时间窗口的结果(例如，一直等待，直到它的watermarks过期)。

结构化流处理实例

让我们来看一个使用结构化流的应用示例。我们将使用异质性人类活动识别数据集，数据是从包括智能手机和智能手表等设备的各种传感器采集的读数，特别是加速度计和陀螺仪，设备以最高频率进行采样。这些传感器采集用户活动的信息并将其记录下来，如骑自行车、坐、站、步行等。其中涉及多个不同的智能手机和智能手表，一共采集了9个用户的信息。可以在“activity data”文件夹中下载该数据。

此数据集相当大。如果它对你的机器来说太大，你可以删除一些文件，它也仍然可用。

读取数据集的静态版本到DataFrame：

```
// in Scala
val static = spark.read.json("/data/activity-data/")
val dataSchema = static.schema

# in Python
static = spark.read.json("/data/activity-data/")
dataSchema = static.schema
```

下面是结构：

```
root
 |-- Arrival_Time: long (nullable = true)
 |-- Creation_Time: long (nullable = true)
 |-- Device: string (nullable = true)
```

```
 |-- Index: long (nullable = true)
 |-- Model: string (nullable = true)
 |-- User: string (nullable = true)
 |-- _corrupt_record: string (nullable = true)
 |-- gt: string (nullable = true)
 |-- x: double (nullable = true)
 |-- y: double (nullable = true)
 |-- z: double (nullable = true)
```

下面是该DataFrame 的一些样本：

```
+-------------+-----------------+--------+-----+------+----+--------+-----+-----
| Arrival_Time|     Creation_Time|  Device|Index| Model|User|_c...ord|.  gt|    x
|1424696634224|14246966322623685|nexus4_1|   62|nexus4|   a|    null|stand|-0...
...
|1424696660715|14246966587238l726|nexus4_1| 2342|nexus4|   a|    null|stand|-0...
+-------------+-----------------+--------+-----+------+----+--------+-----+-----
```

你可以在前面的示例中看到，其中包括一些时间戳、型号信息、用户信息和设备信息列。 gt 字段记录用户当时正在进行的活动。

接下来我们创建该数据集的流式版本，它会将数据集中的每个输入文件一个一个地读出来，就好像是一个流一样。

流式 DataFrame基本与静态 DataFrame相同，我们需要在Spark应用程序中创建它们，然后对它们执行转换操作，以使数据成为正确的格式。基本上，所有静态的结构化 API 中可用的转换操作都适用于流式DataFrame。但是，一个小的区别是结构化流处理不允许你在未明确指定它的情况下执行模式推断，需要通过设置spark.sql.streaming.schemaInference为 true来显式指定模式推断。鉴于此，我们将从一个具有已知可用的数据模式的文件中读取模式dataSchema 对象，并将dataSchema 对象从静态 DataFrame 应用到流式DataFrame。如前所述，在你的数据可能 (意外地)被更改时，应避免在生产应用中使用模式推断：

```
// in Scala
val streaming = spark.readStream.schema(dataSchema)
  .option("maxFilesPerTrigger", 1).json("/data/activity-data")

# in Python
streaming = spark.readStream.schema(dataSchema).option("maxFilesPerTrigger", 1)\
  .json("/data/activity-data")
```

本章将在稍后讨论maxFilesPerTrigger属性，它控制Spark将以什么样的速度读取文件夹中的文件。将该值设置的低一些，比如将流限制为每次触发读取一个文件，这有助于演示如何以增量方式运行结构化流处理，但在生产中可能需要设置为更大的值。

就像其他的Spark API 一样，流式DataFrame的创建和执行是惰性的。特别是，我们现在可以在最终调用启动流的操作之前在流式DataFrame上指定转换操作。在这种情况下，我们将展示一个简单的转换，将按gt列对数据进行分组和计数，这是用户在该时间点执行的活动：

```
// in Scala
val activityCounts = streaming.groupBy("gt").count()

# in Python
activityCounts = streaming.groupBy("gt").count()
```

由于此代码是在一台普通计算机上以本地模式编写的，所以我们要将shuffle分区设置为一个较小的数值，以避免创建太多的shuffle分区：

```
spark.conf.set("spark.sql.shuffle.partitions", 5)
```

在设置了转换操作之后，我们只需要调用动作操作来启动查询。另外，正如本章前面提到的，我们需要为查询结果指定输出目标或输出接收器Sink。对于这个基本示例，我们将编写一个内存接收器，它将结果保存为内存中的表。

在指定这个接收器的过程中，我们需要指定Spark将如何输出这些数据。在本例中，我们使用*complete*输出模式，在每个触发操作之后重写所有key键以及它们的计数：

```
// in Scala
val activityQuery = activityCounts.writeStream.queryName("activity_counts")
  .format("memory").outputMode("complete")
  .start()

# in Python
activityQuery = activityCounts.writeStream.queryName("activity_counts")\
  .format("memory").outputMode("complete")\
  .start()
```

我们正在写出我们的数据流!你会注意到，我们设置了唯一的查询名称来代表流，即activity_counts，我们将格式指定为内存表，并设置了输出模式。

当我们运行前面的代码时，还要包括下面这一行代码：

```
activityQuery.awaitTermination()
```

执行此代码后，流式计算将在后台启动。查询对象是该活跃流查询的句柄，也需要指定我们希望使用activitQuery.awaitTermination()来等待查询终止，以防止查询程序还在运行而驱动器已经终止的情况。我们之后会略掉这行代码，但是在生产应用中必须都有，否则你的流处理程序将无法运行。

Spark可以列出这个流和其他我们Spark Session中活跃的流。通过运行下面代码，我们可以看到这些流的列表：

```
spark.streams.active
```

Spark为每个流分配一个 UUID，所以如果需要，你可以遍历流的列表，并选择上面的一个。在这个例子中，我们把流赋值给了一个变量，所以不需要通过UUID获取流。

现在这个流正在运行，我们可以通过查询内存中的数据表来查看流聚合的结果。此表叫做 activity_counts，与流的名称相同。要查看这个输出表中的当前数据，我们只需要简单地查询它!我们将在一个简单的循环中执行此操作，每秒打印一次流查询的结果：

```
// in Scala
for( i <- 1 to 5 ) {
    spark.sql("SELECT * FROM activity_counts").show()
    Thread.sleep(1000)
}

# in Python
from time import sleep
for x in range(5):
    spark.sql("SELECT * FROM activity_counts").show()
    sleep(1)
```

在前面的查询运行时，你应该看到每个活动的计数随时间而变化。例如，第一次调用 show 显示以下结果 (因为我们在流读取第一个文件的同时查询了它)：

```
+---+-----+
| gt|count|
+---+-----+
+---+-----+
```

上一个显示调用展示以下结果。请注意，当你亲自运行此代码时，结果可能会有所不同，因为你可能会在不同的时间启动它：

```
+----------+-----+
|        gt|count|
+----------+-----+
|       sit| 8207|
...
```

```
|      null| 6966|
|      bike| 7199|
+----------+-----+
```

通过这个简单的例子，结构化流处理的优势应该很清晰了。你可以执行在批处理中使用的相同操作，代码只需做少量修改（实际上只是指定它是流的修改操作）就可以直接在数据流上运行。本章的其余部分将介绍可用于结构化流处理的各种操作、数据源和接收器的一些细节。

结构化流上的转换操作

正如我们提到的，流数据的转换几乎包括第Ⅱ部分中介绍的所有静态 DataFrame 转换操作。所有的选择、筛选和简单转换都是支持的，所有 DataFrame 函数和单个列操作也都支持。对在流处理中没有意义的转换操作会稍有限制，例如，在 Apache Spark 2.2 中，用户无法对没有被聚合的流进行sort排序，并且不能在不使用状态处理（Stateful Processing，在下一章介绍）的情况下执行多级别聚合。随着结构化流处理的不断发展，这些限制可能会被取消，因此建议你检查新版本文档的更新。

选择和筛选

结构化流处理支持所有的选择转换和筛选转换，对所有 DataFrame 函数和对单个列的操作也都支持。我们使用下面的选择和筛选操作演示一个简单的示例，由于我们Key没有任何改变，我们将使用追加append输出模式，以便将新结果追加到输出表中：

```
// in Scala
import org.apache.spark.sql.functions.expr
val simpleTransform = streaming.withColumn("stairs", expr("gt like '%stairs%'"))
  .where("stairs")
  .where("gt is not null")
  .select("gt", "model", "arrival_time", "creation_time")
  .writeStream
  .queryName("simple_transform")
  .format("memory")
  .outputMode("append")
  .start()

# in Python
from pyspark.sql.functions import expr
simpleTransform = streaming.withColumn("stairs", expr("gt like '%stairs%'"))\
  .where("stairs")\
  .where("gt is not null")\
  .select("gt", "model", "arrival_time", "creation_time")\
  .writeStream\
  .queryName("simple_transform")\
  .format("memory")\
```

```
  .outputMode("append")\
  .start()
```

聚合

结构化流处理对聚合操作有很好的支持，你可以指定任意聚合，如在结构化 API 中看到的那样。例如，你可以根据电话型号和活动信息进行cube分组后，再计算x、y、z加速平均值等这样复杂的聚合操作（请翻回到第7章查看你可以在流上运行的潜在聚合操作）。

```
// in Scala
val deviceModelStats = streaming.cube("gt", "model").avg()
  .drop("avg(Arrival_time)")
  .drop("avg(Creation_Time)")
  .drop("avg(Index)")
  .writeStream.queryName("device_counts").format( “memory").outputMode( “complete")
  .start()

# in Python
deviceModelStats = streaming.cube("gt", "model").avg()\
  .drop("avg(Arrival_time)")\
  .drop("avg(Creation_Time)")\
  .drop("avg(Index)")\
  .writeStream.queryName("device_counts").format("memory")\
  .outputMode("complete")\
  .start()
```

查询该表可以看到下面结果：

```
SELECT * FROM device_counts
```

```
+----------+------+------------------+--------------------+--------------------+
|        gt| model|            avg(x)|              avg(y)|              avg(z)|
+----------+------+------------------+--------------------+--------------------+
|       sit|  null|-3.682775300344...|1.242033094787975...|-4.22021191297611...|
|     stand|  null|-4.415368069618...|-5.30657295890281...|2.264837548081631...|
...
|      walk|nexus4|-0.007342235359...|0.004341030525168...|-6.01620400184307...|
|stairsdown|nexus4|0.0309175199508...|-0.02869185568293...| 0.11661923308518365|
...
+----------+------+------------------+--------------------+--------------------+
```

除了对数据集中的原始列上的这些聚合外，结构化流处理对表示事件时间的列具有特殊支持，包括watermark支持和窗口操作。我们将在第22章中更详细地讨论这些内容 。

在Spark 2.2 中，聚合的一个限制是不支持“链”起来的多个聚合（流聚合上的聚合）。但是，你可以通过写出数据到中间数据接收器 (如Kafka或文件库) 来实现此目的。在未来这将很快支持，结构化流处理社区已经添加了此功能。

连接操作

从 Apache Spark 2.2开始，结构化流处理支持流式DataFrame与静态 DataFrame的连接操作。Spark 2.3将增加对连接多个流的支持。你可以执行多列连接，并从静态数据源中补充流数据：

```
// in Scala
val historicalAgg = static.groupBy("gt", "model").avg()
val deviceModelStats = streaming.drop("Arrival_Time", "Creation_Time", "Index")
  .cube("gt", "model").avg()
  .join(historicalAgg, Seq("gt", "model"))
  .writeStream.queryName("device_counts").format("memory").outputMode("complete")
  .start()

# in Python
historicalAgg = static.groupBy("gt", "model").avg()
deviceModelStats = streaming.drop("Arrival_Time", "Creation_Time", "Index")\
  .cube("gt", "model").avg()\
  .join(historicalAgg, ["gt", "model"])\
  .writeStream.queryName("device_counts").format("memory")\
  .outputMode("complete")\
  .start()
```

在Spark 2.2 中，不支持完全外连接、流在右侧的左连接、流在左侧的右连接。结构化流处理也不支持流和流的连接，但这也是正在开发的一个特性。

输入和输出

本节深入介绍数据源、接收器和输出模式在结构化流处理中的工作方式。具体地说，我们将讨论流处理系统中数据是在什么地点什么时间以怎样的方式（where、when、how）流进流出系统的。在编写本书时，结构化流处理支持多个数据源和接收器，包括Apache Kafka、文件，以及用于测试和调试的数据源和接收器。随着时间的推移，可能会添加更多的数据源，因此请务必检看最新版本的文档介绍。本章将讨论特定存储系统的数据源和接收器，但实际上你可以混用它们 (例如，使用Kafka输入源和文件接收器)。

从哪里读取和向哪里写入(数据源和接收器)

结构化流处理支持多个可用于生产的数据源和接收器 (文件和 Apache Kafka)，以及一些调试工具，如内存表接收器。我们在本章的开头提到了这些，现在我们来讨论一下每个细节。

文件数据源和接收器

也许你能想到的最简单的数据来源是简单的文件源，这很容易解释和理解，虽然基本上任何文件源都应该可行，但我们在实践中常看到的是Parquet、文本、JSON 和 CSV。

使用文件数据源/接收器与Spark静态文件源之间的唯一区别是，通过流我们可以控制在每个触发器读取的文件数，即maxFilesPerTrigger选项，这个我们在之前就看到过。

请记住，为流式作业添加到输入目录中的任何文件都需要以原子形式显示，否则Spark将在你完成之前处理部分写出文件。在可以显示部分写入的文件系统 (如本地文件系统或 HDFS) 中，最好是在外部目录中写文件并在完成后将其移动到输入目录中。在亚马逊 S3中，对象通常只在完全写完后出现。

Kafka数据源和接收器

Apache Kafka是一种分布式的数据流发布和订阅系统。Kafka允许你发布和订阅与消息队列类似的记录流，它们以容错的方式存储，可以认为Kafka类似一个分布式缓冲区。Kafka将记录流分类存储在主题（topic）中，每条记录都包含一个键、一个值和一个时间戳，主题包含不可改变的记录序列，每个记录在序列中的位置成为偏移量（offset），读取数据称为订阅（subscrib）主题，写数据类似把数据发布（publish）到主题上。

Spark可以以批量方式和流方式从Kafka上读取 DataFrame。

从Spark 2.2版本开始，结构化流处理支持Kafka 0.10 版，将来可能扩展到新版本，所以一定要查看更新文档，了解更多有关支持Kafka版本的信息。在你阅读Kafka时，只有几个选项需要指定。

从Kafka源读取

读取数据之前，首先需要选择下列选项之一：`assign`，`subscribe`或者`subscribePattern`，当从Kafka中读取时只能选择其中一个。assign是一种细粒度的方法，它不仅指定主题，而且还需要指定你想读取的主题分区（topic partition），这通

过一个JSON字符串指定，{"topicA"： [0，1]，"topicB"： [2，4]}。subscribe和subscribePattern是通过指定主题列表（前者）和指定模式（后者）来订阅一个或多个主题的方法。

其次，你需要指定kafka提供的kafka.bootstrap.servers来连接到服务。

在指定了这些选项后，还有几个其他选项可以指定：

startingOffsets和endingOffsets
: startingOffsets指定查询的起始点，可以是从最早的偏移量earliest；也可以是最新的偏移量latest；或者是一个 JSON 字符串，它指定每个TopicPartition的起始偏移量。在 JSON 中，-2代表最早的偏移量，-1是最新的。例如，JSON格式可以写成是{"topicA"： {"0"：23，"1"：-1}，"topicB"： {"0"：-2}}。这仅当新的流式查询启动时才适用，并且将始终从查询中断的偏移量位置开始恢复，查询过程中新发现的分区将从最早偏移量开始。endingOffsets是查询的终止偏移量。

failOnDataLoss
: 在可能丢失数据 (例如，删除主题或偏移量超出范围) 时是否停止查询，这可能是一个误报，如果它不能按预期工作，也可以禁用它，默认值为 true。

maxOffsetsPerTrigger
: 在给定触发器中读取的总偏移量。

还有一些选项可以设置Kafka消费者超时、提取重试次数、和间隔时间。

要从Kafka读取，请在结构化流处理中执行以下操作：

```
// in Scala
// 订阅一个主题
val ds1 = spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribe", "topic1")
  .load()
// 订阅多个主题
val ds2 = spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribe", "topic1,topic2")
  .load()
// 按照某模式订阅
val ds3 = spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribePattern", "topic.*")
  .load()
```

在Python中非常相似：

```
# in Python
# 订阅一个主题
df1 = spark.readStream.format("kafka")\
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")\
  .option("subscribe", "topic1")\
  .load()
# 订阅多个主题
df2 = spark.readStream.format("kafka")\
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")\
  .option("subscribe", "topic1,topic2")\
  .load()
# 按照某模式订阅
df3 = spark.readStream.format("kafka")\
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")\
  .option("subscribePattern", "topic.*")\
  .load()
```

数据源中的每一行都具有以下模式：

- 键（key）：二进制（binary）。
- 值（value）：二进制（binary）。
- 主题（topic）：字符串（string）。
- 分区（partition）：整型（int）。
- 偏移量（offset）：长整型（long）。
- 时间戳（timestamp）：长整型（long）。

Kafka中的每条消息都有可能以某种方式序列化，在结构化 API 或用户定义函数 (UDF) 中使用原生的Spark函数，可以将消息解析为更结构化的格式。一个常见的模式是使用 JSON 或 Avro 读写Kafka。

写入Kafka接收器

写入Kafka与从Kafka读取数据，除了几个参数不同之外其他几乎相同。你仍然需要指定Kafka的引导服务器，你需要唯一额外提供的是，指定主题topic的列，或者把topic作为选项给出。例如，下面两种写法是等效的：

```
// in Scala
ds1.selectExpr("topic", "CAST(key AS STRING)", "CAST(value AS STRING)")
  .writeStream.format("kafka")
  .option("checkpointLocation", "/to/HDFS-compatible/dir")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .start()
```

```
ds1.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
  .writeStream.format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("checkpointLocation", "/to/HDFS-compatible/dir")\
  .option("topic", "topic1")
  .start()

# in Python
df1.selectExpr("topic", "CAST(key AS STRING)", "CAST(value AS STRING)")\
  .writeStream\
  .format("kafka")\
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")\
  .option("checkpointLocation", "/to/HDFS-compatible/dir")\
  .start()
df1.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")\
  .writeStream\
  .format("kafka")\
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")\
  .option("checkpointLocation", "/to/HDFS-compatible/dir")\
  .option("topic", "topic1")\
  .start()
```

foreach 接收器

foreach接收器类似于Dataset API中的foreachPartitions，此操作在每个分区上并行执行任意操作。这种写法最开始在 Scala 和 Java中使用，但将来其他语言很有可能也支持它。要使用 foreach 接收器，必须实现 ForeachWriter 接口，它在 Scala/Java 文档中有所描述，其中包含三个方法：open，process和close。在触发操作生成输出行序列的时候，就会调用相关的方法。

以下是一些重要的细节：

- 写出器writer必须是可序列化的（Serializable），它可以是 UDF 或Dataset映射函数。
- 每个执行器上，这三个方法 (open，process，close)将会被调用。
- 写出器必须在open方法中执行其所有初始化，如打开连接或启动事务。常见的错误是，初始化发生在open 方法之外 (在使用的类中)，比如在驱动器而不是执行器上发生。

因为 Foreach 接收器运行任意用户代码，所以使用它时必须考虑的一个关键问题是容错。如果结构化流处理要求你的接收器写出一些数据，但随后崩溃，它无法知道你的原始写入是否成功。因此，API 提供了一些附加参数来保证你只处理一次。

首先，在ForeachWriter中调用open函数会接收两个参数，这两个参数唯一标识了

需要执行操作的行集。version参数是一个单调递增的 ID，每次触发后自动增加，partitionId 是任务中输出分区的ID。 open 方法应返回是否处理这组行，如果你在外部跟踪接收器的输出，并看到这组行已经输出 (例如，你在存储系统中看到了最后一个version和partitionId已经输出)，则可以让open方法返回false来跳过处理这组行。如果要处理，则返回true，ForeachWriter将再次打开来写出每次触发的数据。

接下来，假设open方法返回true，那么process方法将处理数据中的每条记录，这是非常简单的，就是处理或写出数据。

最后，每当open方法被调用，那不管它返不返回true，close方法都会被调用（除非在这之前节点崩溃）。如果Spark在处理过程中出现错误，则close方法将接收到该错误，你应该在close方法中清理任何程序占用的资源 。

同时ForeachWriter接口可以帮你高效实现自己的接收器，包括跟踪哪些触发器的数据已经被写入的逻辑，以及在发生错误时如何安全恢复的逻辑。可以看下面的ForeachWriter的例子来理解 ：

```
//in Scala
datasetOfString.write.foreach(new ForeachWriter[String] {
  def open(partitionId: Long, version: Long): Boolean = {
    // 打开数据连接
  }
  def process(record: String) = {
    // 将字符串写入连接
  }
  def close(errorOrNull: Throwable): Unit = {
    // 关闭连接
  }
})
```

用于测试的数据源和接收器

Spark还包括几个用于测试的数据源和接收器，可用于流查询的调试 (仅在开发过程中使用，而不能应用于生产，因为它们不为你的应用程序提供端到端的容错能力)：

Socket数据源

套接字（socket）数据源允许你通过 TCP 套接字向流发送数据。通过指定要从中读取数据的主机和端口来启动它，Spark将打开一个新的 TCP 连接，以从该地址读取数据。在生产中请不要使用套接字数据源，因为套接字位于驱动器上，并且不提供端到端的容错保证。

下面是设置从地址端口为localhost：9999的socket数据源读取流数据的简单示例：

```
// in Scala
val socketDF = spark.readStream.format("socket")
  .option("host", "localhost").option("port", 9999).load()

# in Python
socketDF = spark.readStream.format("socket")\
  .option("host", "localhost").option("port", 9999).load()
```

如果你想将数据写入应用程序，则需要运行侦听端口9999的服务器，在类UNIX的系统上，你可以使用 NetCat 工具，这允许你可以往端口9999的连接中键入文本。在启动Spark应用程序之前运行下面的命令，然后在里面写入文本数据：

```
nc -lk 9999
```

套接字源将返回一个文本字符串的数据表，输入数据中每一行对应数据表中的一行。

控制台接收器

控制台接收器允许你将一些流式查询写入控制台，对调试很有用，但它不支持容错。写到控制台很简单，将流查询的一些行打印到控制台。它支持追加append和完全complete输出模式：

```
activityCounts.format("console").write()
```

内存接收器

设置内存接收器是测试流处理系统的一个简单方法。它与控制台接收器类似，除了它不是打印到控制台，它将数据收集到驱动器，然后使数据整理为可用于交互式查询的内存表。这个接收器不是容错的，所以不应该在生产中使用它，但是在开发过程中用于测试，那自然是很合适的。它支持追加append和完全complete输出模式：

```
// in Scala
activityCounts.writeStream.format("memory").queryName("my_device_table")
```

如果要将数据输出到表中，以便在生产中进行交互式 SQL 查询，作者建议在分布式文件系统上使用Parquet文件接收器 (例如，S3)，这样你就可以在任何Spark应用程序中查询数据。

数据是怎么被输出的 (输出模式)

现在我们知道数据输出到哪里，下面我们讨论结果数据集输出时的形式，这就是我们所说的输出模式（output mode）。正如我们说过的，它与静态 DataFrame的保存模式是相同的概念。结构化流支持三种输出模式，我们一个一个来介绍。

追加（append）模式

追加模式是默认的模式，也是最容易理解的。当新行添加到结果数据表中时，它们将根据指定的触发器 (后面会解释) 输出到接收器。假设接收器提供容错能力，此模式确保每行输出一次 （并且仅一次）。当你使用带有事件时间和watermarks的追加模式（第22 章将会讨论）时，只有最终结果才会输出到接收器。

完全（complete）模式

完全模式将结果表的整个状态输出到接收器。当你使用有状态数据时这就很有用，因为每行都可能会随时间而变化，也可能因为你正在写入的接收器不支持行级更新。可以把它想象为在上一批量运行后流的状态。

更新（update）模式

更新模式类似于完全模式，但只把与上一个批量写出中不同的行才会被写出到接收器。当然，你的接收器必须支持行级更新以支持此模式。如果查询不包含聚合操作，则它和追加模式是等价的。

该使用哪种模式?

结构化流处理在有些情况是限制某种模式的，因为要保证应用到查询的模式必须有意义。例如，如果查询只是执行map操作，则结构化流处理不允许complete完全模式，因为这将要求它记住自作业开始以来的所有输入记录并重写整个输出表，随着作业的进行，这一定会变得非常耗时。我们将在下一章更详细地讨论何时支持何种模式，同时也会说明事件时间处理和watermarks。如果选择的模式不可用，则在启动流时，Spark流将引发异常。

这里有一个Spark官方文档中列出的表，但是在将来一定会发生变化，你需要经常检查最新版本的文档。

表21-1显示何时使用何种输出模式。

表21-1：Spark 2.2 结构化流处理的输出模式

查询类型	查询类型 (续)	支持输出模式	备注
聚合查询	具有watermark的事件时间聚合	追加 更新 完全	追加模式使用waternark除去旧的聚合状态。这意味着当表插入新行时，Spark将只保留在“watermark” 以下的行。更新模式还使用watermark删除旧的聚合状态。根据定义，完全模式不会除去旧的聚合状态，因为此模式保留结果表中的所有数据
	其他聚合	完全， 更新	由于没有定义watermark (仅在其他类别中定义)，所以不会丢弃旧的聚合状态。不支持追加模式是因为聚合可以更新会违反此模式的语义
使用mapGroupsWithState的查询		更新	
使用flatMapGroupsWithState的查询	追加模式	追加	flatMapGroupsWithState 允许聚合
	更新模式	更新	flatMapGroupsWithState 不允许聚合
其他查询		追加 更新	不支持完全模式，因为在结果表中保留所有未聚合的数据是不可行的

数据是何时被输出的（触发器）

为了控制数据何时输出到接收器，我们设置一个触发器（Trigger）。默认情况下，当上一个触发器完成处理时，结构化流处理将立即启动数据。你可以使用触发器来确保不输出过多的更新以避免对接收器造成太大的负载，也可以使用触发器来尝试控制文件大小。目前有一种基于处理时间的周期性触发器类型，以及一个手动控制的触发器类型来触发一次数据，将来可能会增加更多的触发器。

处理时间触发器

对于处理时间触发器（processing time trigger），只需用字符串来制定处理时间周期(你也可以使用Scala中的Duration或者Java中的TimeUnit)。下面将显示字符串格式。

```
// in Scala
import org.apache.spark.sql.streaming.Trigger

activityCounts.writeStream.trigger(Trigger.ProcessingTime("100 seconds"))
  .format("console").outputMode("complete").start()

# in Python
activityCounts.writeStream.trigger(processingTime='5 seconds')\
  .format("console").outputMode("complete").start()
```

ProcessingTime触发器将等待给定持续时间的倍数才能输出数据。例如，触发时间为一分钟，触发器将在12：00、12：01、12：02 等处触发。如果由于先前的处理尚未完成而错过触发时间，则Spark将等待到下一个触发点 (即下一分钟)，而不是在上一次处理完成后立即运行。

一次性触发

你也可以设置一次性触发器（once tirgger）来运行流处理作业，这可能看起来很奇怪，但它实际上非常有用。在开发过程中，你可以一次只测试一个触发器的数据。在生产过程中，可以使用一次性触发器以较低的速率手动运行作业 (例如，只是偶尔将新数据输出到摘要表中)。由于结构化流处理仍然完全跟踪处理的所有输入文件和计算状态，这比编写你自己的自定义逻辑来跟踪批处理作业更容易，并且比运行连续作业节省了大量资源。

```
// in Scala
import org.apache.spark.sql.streaming.Trigger

activityCounts.writeStream.trigger(Trigger.Once())
  .format("console").outputMode("complete").start()

# in Python
activityCounts.writeStream.trigger(once=True)\
  .format("console").outputMode("complete").start()
```

流式Dataset API

关于结构化流处理最后一点要注意的是，你不仅限于使用DataFrame API来处理流数据，你还可以用类型安全的方式使用Dataset执行相同的计算。你可以将流式DataFrame转换为Dataset，这与静态数据的处理方式相同。与静态Dataset一样，

Dataset的元素需要是 Scala case 类或 Java bean 类。另外，在流DataFrame 和Dataset上的操作与在静态数据上的操作方式相同，在流上运行时也会变成流式执行计划。

下面是一个示例，使用与第11章中相同的数据集：

```
// in Scala
case class Flight(DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String,
  count: BigInt)
val dataSchema = spark.read
  .parquet("/data/flight-data/parquet/2010-summary.parquet/")
  .schema
val flightsDF = spark.readStream.schema(dataSchema)
  .parquet("/data/flight-data/parquet/2010-summary.parquet/")
val flights = flightsDF.as[Flight]
def originIsDestination(flight_row: Flight): Boolean = {
  return flight_row.ORIGIN_COUNTRY_NAME == flight_row.DEST_COUNTRY_NAME
}
flights.filter(flight_row => originIsDestination(flight_row))
  .groupByKey(x => x.DEST_COUNTRY_NAME).count()
  .writeStream.queryName("device_counts").format("memory").outputMode("complete")
  .start()
```

小结

应该清楚的是，使用结构化流处理是编写流式应用程序的有效方法。批处理作业程序的代码几乎不需要怎么改动，就可以转换为流处理作业程序。如果需要这个作业与数据处理应用的其他部分进行交互的话，这种简单的直接转换能力对于工程师来说很有帮助。第22章会深入讨论两个高级的流处理概念：事件时间处理和有状态处理。然后，第23章将讨论在生产中运行结构化流处理时需要做的事情。

第22章

事件时间和有状态处理

第21 章介绍了Spark流处理核心概念和基本API，本章深入讨论事件时间（event-time）和有状态处理（stateful processing）。事件时间处理是一个热门话题，因为我们根据记录创建时间而非处理时间来分析信息，这种处理方式的主要思想是，在作业执行的生命周期中，在结果输出到接收器之前，Spark将维护相关状态并在作业执行过程中不断更新它。

在开始使用代码来展示它们如何工作之前，我们先详细地介绍以下这些概念。

事件时间

事件时间是一个很重要的概念，而Spark的DStream API 不支持有关事件时间的处理。在流处理系统中，每个事件实际上有两个相关的时间：实际发生的时间(事件时间)，以及被处理或到达流处理系统的时间(处理时间)。

事件时间

事件时间是嵌入在数据本身中的时间，事件时间通常为事件实际发生的时间，尽管不是必须的。事件时间很重要，因为它提供了一种更加健壮的方式来比较事件。而面临的挑战是事件数据可能会延迟或乱序，所以流处理系统必须能够处理乱序或延迟的数据。

处理时间

处理时间是流处理系统实际接收数据的时间。它通常不如事件时间重要，因为数据何时被处理属于实现细节，并且处理时间也不可能是乱序的，因为它是流处理

系统在某个时刻的一个属性，而不是像事件时间那样的是由外部系统标注的时间。

这些解释是很好但比较抽象，所以我们用一个更具体的例子来解释。假设我们有一个位于旧金山的数据中心。一个事件同时发生在两个地方，一个在厄瓜多尔，另一个在弗吉尼亚 (见图 22-1)。

图22-1：全球范围的事件时间

由于数据中心的位置，弗吉尼亚的事件可能会先于厄瓜多尔的事件之前到达我们的数据中心。如果我们要根据处理时间分析这些数据，看起来弗吉尼亚州的事件发生在厄瓜多尔事件之前，当然我们知道这是错误的。但是如果根据事件时间来分析数据，我们会看到这些事件同时发生。

正如我们之前提到的，处理系统中记录的一系列事件的顺序不能保证它也是按照事件时间的顺序。这可能有点不直观，但值得强调。计算机网络是不可靠的，这意味着事件数据可能会有丢包、传输阻塞、重复发送的情况。由于对于单个事件数据，不能保证它并不发生这些情况，因此我们必须承认，从信息源到流处理系统的传输途中，这些事件数据可能会发生很多意外情况。出于这个原因，我们需要根据事件时间进行操作，并参考数据中包含的信息来处理整个流，而不应该参考数据到达系统的时间，也就是说我们应该根据事件发生的时间来比较事件数据。

有状态处理

在本章另外一个要讨论的主题是有状态处理。实际上我们已经在第21 章已经谈到了很多次有状态处理，只有当你需要长时间使用或更新中间结果信息（状态）时，才需要进行有状态处理（在微批量处理模型和面向记录模型中都是如此）。当你使用事件时间时或在键上执行聚合操作时，有状态处理就可能会发生，无论是否涉及事件时间。

在大多数情况下，当你执行有状态操作时，Spark会为你处理所有复杂的事情。例如，在实现分组操作时，结构化流处理会为你维护并更新信息，你只需指定处理逻辑。在执行有状态操作时，Spark会将中间结果信息存储在状态存储中。Spark当前的状态存储实现是一个内存状态存储，它通过将中间状态存储到检查点目录来实现容错。

任意有状态处理

上面描述的有状态处理功能足以解决许多流处理问题，但是，有时关于应该存储什么状态、如何更新状态，以及何时移除状态（显式指定或通过一个超时限制）等这些问题，你需要进行细粒度控制，这叫做任意（或定制）有状态处理，并且Spark允许你在整个流处理过程中存储任何你想要的信息。这提供了极大的灵活性和强大的功能，你可以轻松处理复杂的业务逻辑。我们用一些例子来说明：

- ·你想要记录有关电子商务网站上用户会话的信息。例如，你可能希望跟踪用户在本次会话过程中访问的页面，以便在下一次会话中实时地提供建议。每个用户的会话的启动时间和停止时间可以是任意的。
- ·贵公司希望报告Web应用程序中的错误，但仅在用户会话期间发生5个错误事件时才会报告错误。你可以使用基于计数的窗口来执行此操作，只有发生某种类型的事件5次时，才输出结果。
- 你希望删除流数据里的重复记录。为此，你需要在找到重复数据之前一直跟踪看到的每条记录。

现在我们已经解释了在本章中的核心概念，让我们用一些可以效仿的例子来说明，并解释以这种方式处理时需要考虑的一些重要注意事项。

事件时间基础知识

让我们接着上一章使用的数据集，在使用事件时间时，它只是数据集中的一列，而这正是我们需要密切关注的，我们只需如同下面示例中的一样简单地使用该列：

```
// in Scala
spark.conf.set("spark.sql.shuffle.partitions", 5)
val static = spark.read.json("/data/activity-data")
val streaming = spark
  .readStream
  .schema(static.schema)
  .option("maxFilesPerTrigger", 10)
  .json("/data/activity-data")

# in Python
spark.conf.set("spark.sql.shuffle.partitions", 5)
static = spark.read.json("/data/activity-data")
streaming = spark\
  .readStream\
  .schema(static.schema)\
  .option("maxFilesPerTrigger", 10)\
  .json("/data/activity-data")

streaming.printSchema()

root
 |-- Arrival_Time: long (nullable = true)
 |-- Creation_Time: long (nullable = true)
 |-- Device: string (nullable = true)
 |-- Index: long (nullable = true)
 |-- Model: string (nullable = true)
 |-- User: string (nullable = true)
 |-- gt: string (nullable = true)
 |-- x: double (nullable = true)
 |-- y: double (nullable = true)
 |-- z: double (nullable = true)
```

在此数据集中，有两个基于时间的列。 `Creation_Time` 列是事件的创建时间，而 `Arrival_Time` 是事件从上游某处到达服务器的时间，本章将使用 `Creation_Time` 。正如我们在上一章中使用的，这个例子是从一个文件中读取流数据，但如果你已经有一个集群正在运行，那么将它更改为Kafka数据源是很简单的。

事件时间的窗口

事件时间分析的第一步是将时间戳列转换为合适的Spark SQL 时间戳类型。我们目前的列是纳秒 (表示为long长整型)，因此我们将做一些操作，将其转换成为适当的格式：

```
// in Scala
val withEventTime = streaming.selectExpr(
  "*",
  "cast(cast(Creation_Time as double)/1000000000 as timestamp) as event_time")

# in Python
withEventTime = streaming\.selectExpr(
  "*",
  "cast(cast(Creation_Time as double)/1000000000 as timestamp) as event_time")
```

我们现在准备对事件时间进行操作。请注意，这里就像在批处理操作中所做的那样，没有特殊的 API 或 DSL。简单地使用列做聚合操作，与批处理中使用列的方法一样，这样就可以使用事件时间。

滚动窗口

最简单的操作就是计算给定时间窗口中某事件的发生次数。图 22-2描述了基于输入数据和键执行简单求和的过程。

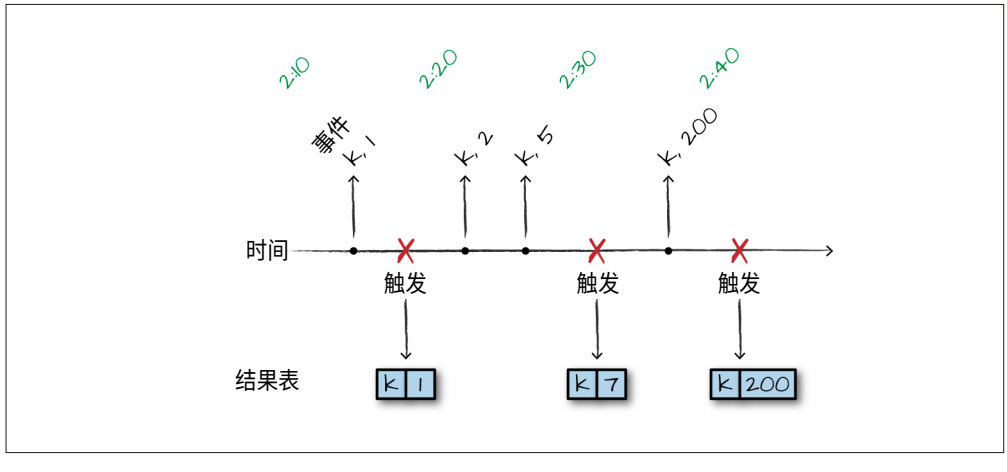


图22-2：滚动窗口

我们正在对某时间窗口内中的键值执行聚合。每次触发器运行时，我们都会更新结果表 (怎么更新取决于输出模式)，这将根据自上次触发器以来收到的数据进行操作。在使用该实际数据集 (和图 22-2) 的情况下，我们在10分钟的时间窗口中进行，并且这些时间窗口不会发生任何重叠（一个事件只可能落入一个时间窗口)。并且结果也将实时更新，如果系统上游添加了新的事件，则结构化流处理将相应地更新这些计数。因为这里是完全输出模式，无论我们是否处理完了整个数据集，Spark都会输出整个结果表：

```
// in Scala
import org.apache.spark.sql.functions.{window, col}
withEventTime.groupBy(window(col("event_time"), "10 minutes")).count()
  .writeStream
  .queryName("events_per_window")
  .format("memory")
  .outputMode("complete")
  .start()

# in Python
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes")).count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

我们把数据写到了内存接收器中以便于调试，我们可以在运行流处理之后使用SQL查询它：

```
spark.sql("SELECT * FROM events_per_window").printSchema()

SELECT * FROM events_per_window
```

该查询的结果如下，结果其实取决于你在运行查询时已经处理了的数据量：

```
+------------------------------------------+-----+
|window                                    |count|
+------------------------------------------+-----+
|[2015-02-23 10:40:00.0,2015-02-23 10:50:00.0]|11035|
|[2015-02-24 11:50:00.0,2015-02-24 12:00:00.0]|18854|
...
|[2015-02-23 13:40:00.0,2015-02-23 13:50:00.0]|20870|
|[2015-02-23 11:20:00.0,2015-02-23 11:30:00.0]|9392 |
+------------------------------------------+-----+
```

作为参考，下面是我们从上一个查询中得到的模式schema：

```
root
 |-- window: struct (nullable = false)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- count: long (nullable = false)
```

注意时间窗口实际上是一个结构体（一个复杂类型），使用此方法，我们可以查询该结构体以获得特定时间窗口的开始时间和结束时间。

重要的是我们可以在多个列上执行聚合操作，包括事件时间列。就像在上一章中看到的那样，我们甚至可以使用cube之类的复杂分组方法来执行聚合。下面代码涉及到执

行多键聚合，但这确实适用于任何窗口式聚合 (或有状态计算)：

```
// in Scala
import org.apache.spark.sql.functions.{window, col}
withEventTime.groupBy(window(col("event_time"), "10 minutes"), "User").count()
  .writeStream
  .queryName("events_per_window")
  .format("memory")
  .outputMode("complete")
  .start()

# in Python
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes"), "User").count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

滑动窗口

前面例子是在给定窗口中简单计数，但是多个时间窗口是可以重叠的，图 22-3的例子来帮助说明滑动窗口。

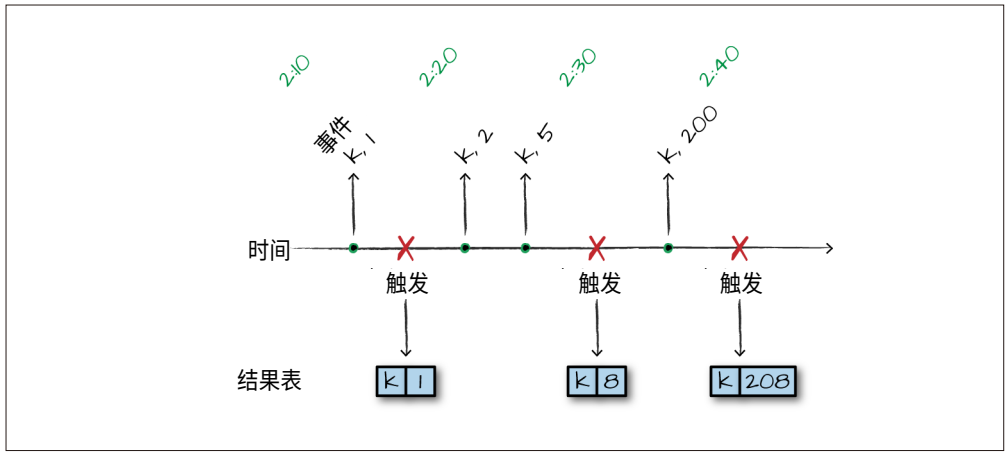


图22-3：滑动窗口

图中正在运行一个以1小时为增量的滑动窗口，但我们希望每10分钟获得一次状态。这意味着，我们将随着时间的推移更新这些值，并考虑之前1小时的数据。在这个例子中，我们设置10分钟的窗口，每隔5分钟开始。因此，每个事件都将落到两个不同的时间窗口，你也可以根据需要进行调整：

```
// in Scala
import org.apache.spark.sql.functions.{window, col}
withEventTime.groupBy(window(col("event_time"), "10 minutes", "5 minutes"))
  .count()
  .writeStream
  .queryName("events_per_window")
  .format("memory")
  .outputMode("complete")
  .start()

# in Python
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes", "5 minutes"))\
  .count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

我们按如下方法查询内存表：

```
SELECT * FROM events_per_window
```

此查询返回以下结果。请注意，这里每隔5分钟就启动一个时间窗口，而不是 10分钟，这样就会产生两个重叠的时间窗口，这与之前的例子不一样。

```
+------------------------------------------+-----+
|window                                    |count|
+------------------------------------------+-----+
|[2015-02-23 14:15:00.0,2015-02-23 14:25:00.0]|40375|
|[2015-02-24 11:50:00.0,2015-02-24 12:00:00.0]|56549|
...
|[2015-02-24 11:45:00.0,2015-02-24 11:55:00.0]|51898|
|[2015-02-23 10:40:00.0,2015-02-23 10:50:00.0]|33200|
+------------------------------------------+-----+
```

使用水位处理延迟数据

前面的例子虽然很好，但有一个缺陷。我们未指定系统可以接受延迟多久的迟到数据，这意味着Spark将需要永久存储这些中间数据，因为我们没有指定一个过期时间，如果数据迟到超过一定时间阈值，我们将不会处理它。这适用于所有基于事件时间的有状态处理，我们必须指定此水位，以便确定过期数据，这样就可以免于由于长时间不清理过期数据而造成系统存储压力过大。

具体来说，水位是给定事件或事件集之后的一个时间长度，在该时间长度之后我们不希望再看到来自该时间长度之前的任何数据。这种数据到达延迟可能是由于网络延

迟、设备断开连接、或多种其他原因造成的。在 DStream API 中，过去没有一种处理延迟数据的可靠方法，如果某个事件是在某个时间窗口发生的，但在对该时间窗口的批处理开始时该事件还未到达处理系统，则它将显示在其他的批处理批次中。结构化流处理可以处理这种问题，在基于事件时间的有状态处理中，一个时间窗口的状态或数据是与处理时间解耦的。这意味着随着更多事件的进入，结构化流处理将会继续更新包含更多信息的窗口。

让我们回到本章开头介绍事件时间的示例上，如图22-4所示。

图22-4：事件时间水位

在这个例子中，假设经常看到拉丁美洲客户事件数据的延迟，所以我们指定一个10分钟的水位。当这样做时，我们命令Spark应忽略前一个事件发生 10 分钟后才收到的任何事件，而希望处理在10分钟内的所有事件。之后，Spark应该移除中间状态，并根据输出模式输出结果。正如本章开头提到的，我们需要指定水位，因为如果不这样做，我们需要保持所有中间结果用来更新窗口状态，所有窗口状态有可能持续被更新。在处理事件时间时，这就带来一个核心问题：“到底保留延迟最长多久的数据？”这个问题的答案就是，为你的数据设置水位。

返回到我们的数据集中，如果知道一般情况下通常会在几分钟内收到上游生产的数据，但偶尔也遇到过在事件发生后5个小时才收到数据的情况 (可能是用户失去了手机连接)，我们按照以下方式指定水位：

```
// in Scala
import org.apache.spark.sql.functions.{window, col}
withEventTime
  .withWatermark("event_time", "5 hours")
  .groupBy(window(col("event_time"), "10 minutes", "5 minutes"))
  .count()
  .writeStream
  .queryName("events_per_window")
  .format("memory")
  .outputMode("complete")
  .start()

# in Python
from pyspark.sql.functions import window, col
withEventTime\
  .withWatermark("event_time", "30 minutes")\
  .groupBy(window(col("event_time"), "10 minutes", "5 minutes"))\
  .count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

这非常了不起，但我们的查询语句几乎没有任何变化，只是增加了另一个配置。现在，结构化流处理会等待10分钟窗口中最晚时间戳之后的30分钟 ，然后才确定该窗口的结果。我们可以查询数据表并查看所有中间结果，因为我们使用的是完整输出模式，它们将随着时间的推移而持续更新。在追加输出模式下，信息直到窗口关闭时才会输出。

```
SELECT * FROM events_per_window
```

```
+------------------------------------------+-----+
|window                                    |count|
+------------------------------------------+-----+
|[2015-02-23 14:15:00.0,2015-02-23 14:25:00.0]|9505 |
|[2015-02-24 11:50:00.0,2015-02-24 12:00:00.0]|13159|
...
|[2015-02-24 11:45:00.0,2015-02-24 11:55:00.0]|12021|
|[2015-02-23 10:40:00.0,2015-02-23 10:50:00.0]|7685 |
+------------------------------------------+-----+
```

此时，你已经了解了处理延时数据的方法，Spark为你完成了所有繁杂的任务。再次强调一下，如果你没有指定要接受迟到多晚的数据，那么Spark会永久将这些数据保留在内存中，指定水位可使其从内存中释放，从而使流处理能够长时间持续运行。

在流中删除重复项

在一次一记录（record-at-a-time）系统中，更困难的操作之一是删除数据流中的重复项，你必须一次对一批记录进行操作，才可能查找重复项，处理系统需要很高的协调开销。重复项删除是许多应用程序需要支持的一个重要能力，尤其是当消息可能从上游系统多次传递时。物联网(IoT)应用程序就是这样一个例子，上游产品在非稳定的网络环境中生产消息，并且相同的消息可能会被多次发送，你的下游应用程序和聚合操作需要保证每个消息只出现一次。

结构化流处理可以轻松地让消息系统支持“至少一次”的语义，并根据记录任意键的方法来丢弃重复消息，将其转换为“只一次”的语义。为了消除重复数据，Spark将维护许多用户指定的键，并忽略重复项。

与其他有状态处理应用程序一样，你需要指定一个水位，以确保维护的状态在流处理过程中不会无限增长。

让我们熟悉下去重的过程，目标是删除每个用户的重复事件数量。在下面的例子中请注意是如何将事件时间列指定为重复列和需要去重的列，一个重要假设是重复事件具有相同的时间戳和标识符，具有两个不同时间戳的行是两个不同的记录：

```
// in Scala
import org.apache.spark.sql.functions.expr

withEventTime
  .withWatermark("event_time", "5 seconds")
  .dropDuplicates("User", "event_time")
  .groupBy("User")
  .count()
  .writeStream
  .queryName("deduplicated")
  .format("memory")
  .outputMode("complete")
  .start()

# in Python
from pyspark.sql.functions import expr

withEventTime\
  .withWatermark("event_time", "5 seconds")\
  .dropDuplicates(["User", "event_time"])\
  .groupBy("User")\
  .count()\
  .writeStream\
```

```
    .queryName("pydeduplicated")\
    .format("memory")\
    .outputMode("complete")\
    .start()
```

结果如下所示，随着读取数据流中更多的数据，这个结果将随着时间的推移而不断更新：

```
+----+-----+
|User|count|
+----+-----+
|   a| 8085|
|   b| 9123|
|   c| 7715|
|   g| 9167|
|   h| 7733|
|   e| 9891|
|   f| 9206|
|   d| 8124|
|   i| 9255|
+----+-----+
```

任意有状态处理

本章的第一部分演示了Spark如何根据用户配置来维护信息和更新窗口。但是当你有更复杂的窗口概念时，情况会有所不同，而这是任意有状态处理（Arbitrary Stateful Processing）的长处。本节包含几个不同应用场景的示例，以及向你展示如何设置业务逻辑的示例。有状态处理仅在Spark 2.2中的Scala语言支持中可用，将来版本可能会支持更多。

执行有状态处理时，可能需要执行以下操作：

- 根据给定键的计数创建窗口。
- 如果在特定时间范围内发生多个特定事件，则发出警报。
- 在不确定时间内维护用户会话，保存这些会话以便稍后进行一些分析。

在一天的最后，如果执行这类处理时，你将需要执行以下两件事情：

- 映射数据中的组，对每组数据进行操作，并为每个组生成至多一行。 这个用例的相关API是`mapGroups WithState`。
- 映射数据中的组，对每组数据进行操作，并为每个组生成一行或多行。 这个用例的相关API是`flatMapGroups WithState`。

当我们说对每组数据进行“操作”时，你可以独立于其他数据组去任意更新该组，也就是说，你也可以定义不符合翻滚窗口或滑动窗口的任意窗口类型。在执行这种处理方式时获得的一个重要好处是可以控制配置状态超时，使用时间窗口和watermark，这非常简单：当一个窗口开始前watermark已经超时，你只需暂停该窗口。 这不适用于任意有状态处理，因为你是基于用户定义的概念来管理状态，所以你需要适当地设置状态超时。接下来我们更详细地讨论一下。

超时

如第21 章所述，超时时间是指在标记某个中间状态为超时（time-out）之前应该等待多长时间，超时时间是作用在每组上的一个全局参数。超时可以基于处理时间 (`GroupStateTimeout.ProcessingTimeTimeout`)， 也可以基于事件时间 (`GroupStateTimeout. EventTimeTimeout`)。使用超时，请先检查超时时间的设置，你可以通过检查`state.hasTimedOut`标志或检查值迭代器是否为空来获取此信息。你需要设置一些状态（必须定义状态，而不是删除状态）才能设置超时。

基于处理时间的超时，可以通过调用`GroupState.setTimeoutDuration`设置超时时间 (我们将在本节后面看到此部分的代码示例) 。当时钟超过设定的持续时间，会发生超时。如果超时时间设置为D毫秒，可以保证：

- 在时钟时间增加D ms之前，超时不会发生。
- 当查询中存在某个触发(在D ms 之后)，则最终会发生超时。因此，超时最终发生的时间没有严格的上限限制。例如，查询的触发间隔会影响实际发生超时的时间，如果数据流中没有任何数据 (相对于某个组)，就没有触发的机会，直到有数据时才会发生超时函数调用。

由于处理时间超时是基于时钟时间的，所以系统时钟的不同会对超时有影响，这意味着时区不同和时钟偏差是需要给予考虑的。

基于事件时间的超时，用户还必须使用watermark水位在查询中指定事件时间水位。设置后，比水位更旧的数据会被过滤掉。作为开发人员，你可以通过使用`GroupState.setTimeoutTimes tamp(...)` API设置超时时间戳，从而设置水位应参考的时间戳，当水位超出设定的时间戳时将发生超时。当然，你可以通过指定较长的水位来控制超时延迟，或者在处理流时动态更新超时时间。因为可以用代码来实现，所以可以针对特定组来更改超时时间。 此超时提供的保证是，在水位超过设置的超时时间之前，保证不会发生这种情况。

与处理时间超时类似，当超时实际发生时，延迟没有严格的上限，在事件时间已经超时的情况下，只有当流中有数据时，才会提高水位。

值得强调的是，虽然超时很重要，但它们可能并不总是按你预期的那样运行。例如，在编写本书时，结构化流处理不支持异步作业执行，这意味着Spark不会在周期（epoch）结束和下一次开始之间输出数据 (或超时数据)，因为它在那个时候不处理任何数据。此外，如果处理的一批数据中没有记录（记住这是批处理的批次，而不是组），则没有更新，并且不可能有事件时间超时。 这可能会在未来版本中改变。

输出模式

在使用这种任意有状态处理时，最后一个“陷阱”是，并非所有在第21章中讨论的输出模式都被支持。 随着Spark的不断改进，这肯定会发生变化，但在编写本书时，`mapGroupsWithState`仅支持`update`更新输出模式，而`flatMapGroupsWithState`支持`append`追加输出模式和`update`更新输出模式。追加模式意味着只有在超时（超过水位watermark）之后，数据才会显示在结果集中。但这不会自动发生，你需要负责输出正确的一行或者多行。

请参见表21-1了解哪种输出模式可以在何种情况下使用。

mapGroupsWithState

第一个有状态处理的示例使用`mapGroupsWithState`，这类似于用户定义的聚合函数，它将更新数据集作为输入，然后将其解析为对应一个值集合的键。 你需要给出如下几个定义：

- 三个类定义： 输入定义、状态定义、以及可选的输出定义。
- 基于键、事件迭代器和先前状态的一个更新状态的函数。
- 超时时间参数 (如“超时”部分所述)。

通过这些对象和定义，你可以通过创建、随时间更新以及删除它来控制任意状态。我们从一个简单的基于定量的状态更新key键的例子开始，然后继续更复杂的例子，比如会话。

来看我们正在处理传感器数据，我们要查找给定用户在数据集中执行某项活动的第一个和最后一个时间戳，这意味着分组操作的键是用户和活动组合。

当你使用mapGroupsWithState时，输出始终是每个键（或组）只对应一行。 如果你希望每个组都有多个输出，则应该使用flatMapGroupsWithState（稍后会介绍）。

我们来建立输入、状态和输出定义：

```
case class InputRow(user:String, timestamp:java.sql.Timestamp, activity:String)
case class UserState(user:String,
  var activity:String,
  var start:java.sql.Timestamp,
  var end:java.sql.Timestamp)
```

还有设置函数，定义如何根据给定行更新状态：

```
def updateUserStateWithEvent(state:UserState, input:InputRow):UserState = {
  if (Option(input.timestamp).isEmpty) {
    return state
  }
  if (state.activity == input.activity) {

    if (input.timestamp.after(state.end)) {
      state.end = input.timestamp
    }
    if (input.timestamp.before(state.start)) {
      state.start = input.timestamp
    }
  } else {
    if (input.timestamp.after(state.end)) {
      state.start = input.timestamp
      state.end = input.timestamp
      state.activity = input.activity
    }
  }

  state
}
```

现在需要通过函数来定义根据每一批次行来更新状态。

```
import org.apache.spark.sql.streaming.{GroupStateTimeout, OutputMode, GroupState}
def updateAcrossEvents(user:String,
  inputs: Iterator[InputRow],
  oldState: GroupState[UserState]):UserState = {
  var state:UserState = if (oldState.exists) oldState.get else UserState(user,
        "",
        new java.sql.Timestamp(6284160000000L),
        new java.sql.Timestamp(6284160L)
    )
```

```
    // 我们直接指定了可以用于比较的旧日期，
    // 并且根据数据值进行及时更新

    for (input <- inputs) {
      state = updateUserStateWithEvent(state, input)
      oldState.update(state)
    }
    state
  }
```

当定义好了这些后，就可以通过传递相关信息来执行查询。在指定mapGroupsWithState时，需要指定是否需要超时给定组的状态，它告诉系统如果在一段时间后没有收到更新的状态应该做什么。在我们这个例子中，你希望无限期地维护状态，因此指定Spark不会超时。

使用该update输出模式，以便获得用户活动的更新：

```
import org.apache.spark.sql.streaming.GroupStateTimeout
withEventTime
  .selectExpr("User as user",
    "cast(Creation_Time/1000000000 as timestamp) as timestamp", "gt as activity")
  .as[InputRow]
  .groupByKey(_.user)
  .mapGroupsWithState(GroupStateTimeout.NoTimeout)(updateAcrossEvents)
  .writeStream
  .queryName("events_per_window")
  .format("memory")
  .outputMode("update")
  .start()

SELECT * FROM events_per_window order by user, start
```

以下是查询结果：

```
+----+--------+-------------------+-------------------+
|user|activity|              start|                end|
+----+--------+-------------------+-------------------+
|   a|    bike|2015-02-23 13:30:...|2015-02-23 14:06:...|
|   a|    bike|2015-02-23 13:30:...|2015-02-23 14:06:...|
...
|   d|    bike|2015-02-24 13:07:...|2015-02-24 13:42:...|
+----+--------+-------------------+-------------------+
```

我们数据集的一个有趣的地方是，在任意时间执行的最后一项活动总是“bike”，这与实验的运行方式可能有关，每个参与者按相同序列执行活动。

示例： 基于计数的窗口

典型的窗口操作是把落入开始时间和结束时间之内的所有事件都来进行计数或求和。但是，有时候不是基于时间创建窗口，而是基于大量事件创建它们，而不考虑状态和事件时间，并在该数据窗口上执行一些聚合。例如，我们可能想要为接收到的每500个事件计算一个值，而不管它们何时收到。

下一个示例分析本章中的活动数据集，并定期输出每个设备的平均读数，根据事件计数创建一个窗口，并在每次为该设备累积500个事件时输出该累积结果。你为此任务定义了两个case类：包括输入行格式（它只是一个设备和一个时间戳），以及状态和输出行（其中包含收集记录的当前计数，设备ID，以及窗口中事件数组）。

下面是各种自描述的case类定义：

```
case class InputRow(device: String, timestamp: java.sql.Timestamp, x: Double)
case class DeviceState(device: String, var values: Array[Double],
  var count: Int)
case class OutputRow(device: String, previousAverage: Double)
```

现在你可以定义一个函数，根据输入的一行来更新状态，你也可以用许多其他方式编写这个例子，这个例子帮你了解如何基于给定的行来更新状态：

```
def updateWithEvent(state:DeviceState, input:InputRow):DeviceState = {
  state.count += 1
  //将x维的值保存为数组
  state.values = state.values ++ Array(input.x)
  state
}
```

现在来定义一个基于多输入行的更新函数。在下面的示例中我们可以看到，有一个特定的键、包含一系列输入的迭代器、还有旧的状态，然后随着时间推移在接收到新事件后我们持续更新这个旧状态。它根据每个设备上发生事件的计数数量，针对每个设备返回更新的输出行。 这种情况非常简单，在一定事件数量之后，更新状态并重置它，然后创建一个输出行。可以在输出表中看到此行：

```
import org.apache.spark.sql.streaming.{GroupStateTimeout, OutputMode,
  GroupState}

def updateAcrossEvents(device:String, inputs: Iterator[InputRow],
  oldState: GroupState[DeviceState]):Iterator[OutputRow] = {
  inputs.toSeq.sortBy(_.timestamp.getTime).toIterator.flatMap { input =>
    val state = if (oldState.exists) oldState.get
      else DeviceState(device, Array(), 0)

    val newState = updateWithEvent(state, input)
    if (newState.count >= 500) {
      // 某个窗口完成之后用空的DeviceState代替该状态，
```

```
        // 并输出旧状态中过去500项的平均值
        oldState.update(DeviceState(device, Array(), 0))
        Iterator(OutputRow(device,
          newState.values.sum / newState.values.length.toDouble))
      }
      else {
        // 更新此处的DeviceState，
        // 并不输出任何结果记录
        oldState.update(newState)
        Iterator()
      }
    }
  }
```

现在，可以运行你的流处理任务了。你会注意到你需要显式说明输出模式，即append模式。你还需要设置一个GroupStateTimeout，这个超时时间指定了窗口输出的等待的时间（即使它没有达到所需的计数）。在这种情况下，如果设置一个无限的超时时间，则意味着如果一个设备一直没有累积到要求的500计数阈值，该状态将一直保持为“不完全”并且不会输出到结果表。

在updateAcrossEvents函数中指定这两个参数之后，就可以启动流处理了：

```
import org.apache.spark.sql.streaming.GroupStateTimeout

withEventTime
  .selectExpr("Device as device",
    "cast(Creation_Time/1000000000 as timestamp) as timestamp", "x")
  .as[InputRow]
  .groupByKey(_.device)
  .flatMapGroupsWithState(OutputMode.Append,
    GroupStateTimeout.NoTimeout)(updateAcrossEvents)
  .writeStream
  .queryName("count_based_device")
  .format("memory")
  .outputMode("append")
  .start()
```

启动流后，就可以执行实时查询。结果如下：

```
SELECT * FROM count_based_device

+--------+-------------------+
|  device|     previousAverage|
+--------+-------------------+
|nexus4_1|      4.660034012E-4|
|nexus4_1|0.001436279298199...|
...
|nexus4_1|1.049804683999999...|
|nexus4_1|-0.01837188737960...|
+--------+-------------------+
```

你可以看到每个窗口的统计值都在变化，新的统计结果加入到了结果集中。

flatMapGroupsWithState

我们的第二个有状态处理的示例使用flatMapGroupsWithState，这与mapGroupsWithState非常相似，不同之处在于一个键不是只对应最多一个输出，而是一个键可以对应多个输出。这可以为我们提供更多的灵活性，并且与mapGroupsWithState具有相同的基本结构。我们需要定义下面几项。

- 三个类定义：输入定义、状态定义，以及可选的输出定义。
- 一个函数，输入参数为一个键、一个多事件迭代器和先前状态。
- 超时事件参数 (如“超时”部分所述)。

通过这些对象和定义，我们可以通过创建状态、（随着时间推移）更新状态，以及删除状态来控制任意状态。接下来我们提供一个会话化（Sessionization）的例子。

示例： 会话化

会话是未指定的时间窗口，其中可能发生一系列事件。通常你需要将这些不同的事件记录在数组中，以便将其与将来的其他会话进行比较。在一次会话中，用户可能会设计各种不同的逻辑来维护和更新状态，以及定义窗口何时结束（如计数）或通过简单的超时来结束。我们以前面的例子为基础，并把它更严格地定义为一个会话。

有时候你可能有一个明确的会话ID，你可以在你的函数中使用它。这是很容易实现的，因为你只是执行简单的聚合，甚至可能不需要你自己定义的有状态逻辑。在下面这个例子中，将根据用户ID和一些时间信息即时创建会话，并且如果你在五秒内没有看到该用户的新事件，则会话将终止。你还会注意到，此代码使用超时的方式与其他示例中的不同。

你可以遵循创建类的相同流程，定义我们的单个事件更新函数，然后定义多事件更新函数：

```
case class InputRow(uid:String, timestamp:java.sql.Timestamp, x:Double,
  activity:String)
case class UserSession(val uid:String, var timestamp:java.sql.Timestamp,
  var activities: Array[String], var values: Array[Double])
case class UserSessionOutput(val uid:String, var activities: Array[String],
  var xAvg:Double)

def updateWithEvent(state:UserSession, input:InputRow):UserSession = {
// 处理异常数据
```

```
    if (Option(input.timestamp).isEmpty) {
      return state
    }

    state.timestamp = input.timestamp
    state.values = state.values ++ Array(input.x)
    if (!state.activities.contains(input.activity)) {
      state.activities = state.activities ++ Array(input.activity)
    }
    state
  }

  import org.apache.spark.sql.streaming.{GroupStateTimeout, OutputMode,
    GroupState}

  def updateAcrossEvents(uid:String,
    inputs: Iterator[InputRow],
    oldState: GroupState[UserSession]):Iterator[UserSessionOutput] = {

    inputs.toSeq.sortBy(_.timestamp.getTime).toIterator.flatMap { input =>
      val state = if (oldState.exists) oldState.get else UserSession(
      uid,
      new java.sql.Timestamp(6284160000000L),
      Array(),
      Array())
      val newState = updateWithEvent(state, input)

      if (oldState.hasTimedOut) {
        val state = oldState.get
        oldState.remove()
        Iterator(UserSessionOutput(uid,
        state.activities,
        newState.values.sum / newState.values.length.toDouble))
      } else if (state.values.length > 1000) {
        val state = oldState.get
        oldState.remove()
        Iterator(UserSessionOutput(uid,
        state.activities,
        newState.values.sum / newState.values.length.toDouble))
      } else {
        oldState.update(newState)
        oldState.setTimeoutTimestamp(newState.timestamp.getTime(), "5 seconds")
        Iterator()
      }

    }
  }
```

你会看到，我们只希望看到最多延迟5秒的事件，我们将忽略太晚到达的事件。在这个有状态操作中，我们将使用EventTimeTimeout来基于事件时间设置超时：

```
  import org.apache.spark.sql.streaming.GroupStateTimeout
```

```
withEventTime.where("x is not null")
  .selectExpr("user as uid",
    "cast(Creation_Time/1000000000 as timestamp) as timestamp",
    "x", "gt as activity")
  .as[InputRow]
  .withWatermark("timestamp", "5 seconds")
  .groupByKey(_.uid)
  .flatMapGroupsWithState(OutputMode.Append,
    GroupStateTimeout.EventTimeTimeout)(updateAcrossEvents)
  .writeStream
  .queryName("count_based_device")
  .format("memory")
  .start()
```

查询此表将显示此时间段内每个用户的输出行：

```
SELECT * FROM count_based_device

+---+--------------------+--------------------+
|uid|          activities|                xAvg|
+---+--------------------+--------------------+
|  a|  [stand, null, sit]|-9.10908533566433...|
|  a|   [sit, null, walk]|-0.00654280428601...|
...
|  c|[null, stairsdown...|-0.03286657789999995|
+---+--------------------+--------------------+
```

如你所料，其中包含许多活动的会话比具有较少活动的会话具有更高的x轴平均值。将这个例子扩展到你自己领域的应用也应该是很简单的。

小结

本章介绍了结构化流处理中的一些更高级的主题，包括事件时间和有状态处理，这实际上是可以帮助你构建应用程序逻辑并将流数据转化为价值的用户指南。接下来，将讨论为了使Spark流处理应用程序应用于实际生产中，我们需要做些什么。

第23章

生产中的结构化流处理

本书流处理部分的前几章从用户的角度讨论了结构化流处理，这是你的应用程序的核心。本章将介绍在开发流处理应用程序后，在生产环境中稳健运行结构化流处理时所需的操作工具。

结构化流处理在Apache Spark 2.2.0中可以被直接应用到生产中，此版本具有生产使用所需的所有功能并具有稳定的API，许多组织已经在生产中使用该系统。坦率地说，它与运行其他可运行在生产中的Spark应用程序没有多大区别。 事实上，通过诸如支持事务处理的数据源/接收器以及一次处理等功能，结构化数据流处理的设计人员试图尽可能地简化操作。本章将向你介绍完成特定于结构化流式处理的一些关键操作任务，这会补充我们在第Ⅱ部分中了解的关于Spark操作的内容。

容错和检查点

流处理应用程序中最需要重视的问题是故障恢复。故障是不可避免的，你可能会丢失集群中的一台机器，在没有适当迁移的情况下数据模式可能发生意外更改，或者你可能需要重新启动集群或应用程序。在上面任何一种情况下，结构化流处理允许你仅通过重新启动应用程序来恢复它。为此，你必须将应用程序配置为使用检查点和预写日志，这两者都由引擎自动处理。具体来说，你必须配置查询以写入可靠文件系统上的检查点位置（例如，HDFS，S3或其他兼容的文件系统）。然后，结构化流处理将周期性地保存所有相关进度信息（例如，一个触发操作处理的偏移量范围），以及当前中间状态值，将它们保存到检查点位置。在失败情况下，你只需重新启动应用程序，

确保指向相同的检查点位置，它将自动恢复其状态并在其中断的位置重新开始处理数据。你不必手动管理此状态，结构化流处理可以自动完成这些过程。

要使用检查点，请在启动应用程序之前通过writeStream上的checkpointLocation选项指定检查点位置。如下所示：

```
// in Scala
val static = spark.read.json("/data/activity-data")
val streaming = spark
  .readStream
  .schema(static.schema)
  .option("maxFilesPerTrigger", 10)
  .json("/data/activity-data")
  .groupBy("gt")
  .count()
val query = streaming
  .writeStream
  .outputMode("complete")
  .option("checkpointLocation", "/some/location/")
  .queryName("test_stream")
  .format("memory")
  .start()

# in Python
static = spark.read.json("/data/activity-data")
streaming = spark\
  .readStream\
  .schema(static.schema)\
  .option("maxFilesPerTrigger", 10)\
  .json("/data/activity-data")\
  .groupBy("gt")\
  .count()
query = streaming\
  .writeStream\
  .outputMode("complete")\
  .option("checkpointLocation", "/some/python/location/")\
  .queryName("test_python_stream")\
  .format("memory")\
  .start()
```

如果你丢失了检查点目录或它内部的信息，则应用程序将无法从故障中恢复，你将不得不从头开始重新启动流处理。

更新应用程序

检查点对于生产中运行应用程序来说大概是最重要的事情，这是因为检查点将存储所有的信息，包括流到目前为止已处理的所有信息，以及它可能存储的中间状态等。但是，检查点有一个小问题，当你更新流处理应用程序时，你不得不对旧的检查点数据

进行推理。在更新应用程序时，必须确保你的更新不是一个中断性的更改。下面我们针对两种更新类型进行详细介绍：更新应用程序代码和运行新的Spark版本。

更新流处理应用程序代码

结构化流式处理允许在应用程序两次重启之间对应用程序代码进行某些更改，最重要的是，只要具有相同的类型签名，就可以更改用户定义函数 (UDF)，此功能对于 bug 修复非常有用。例如，假设你的应用程序开始接收新类型的数据，并且当前逻辑中的某个数据解析函数崩溃，使用结构化流处理，你可以使用该函数的新版本重新编译应用程序，并在之前流处理的崩溃位置开始流处理。

尽管像添加新列或更改UDF这样的小调整不算是破坏性的改变，并且不需要新的检查点目录，但有某些更大的更改需要新的检查点目录。例如，如果你更新流应用程序以添加用于聚合操作的新键，或甚至更改查询本身，则Spark无法为新查询从旧的检查点目录构建所需的状态。在这些情况下，结构化流处理将会抛出一个异常，表示无法从检查点目录开始，必须从头开始用一个新的（空）目录作为检查点位置。

更新你的Spark版本

对于基于补丁（patch）的版本更新，结构化流处理应用程序应该能够从旧检查点目录重新启动（例如，从Spark 2.2.0迁移到2.2.1，再到2.2.2）。检查点格式被设计为向前兼容，因此它可能被破坏的原因是由于关键bug修复。如果Spark发行版无法从旧检查点恢复，则会在其发行说明中明确说明。结构化数据流处理开发人员会尽量保持在微调版本更新（例如，Spark 2.2.x到2.3.x）之间的格式兼容，但是你应该检查发行说明以了解每次升级是否支持旧的检查点格式。无论哪种情况，如果无法从检查点启动，则需要使用新的检查点目录再次启动应用程序。

调整应用程序的大小

一般而言，集群的大小应该能够轻松地处理超出数据速率突发的情况，你应该在应用程序和集群中监视如下所述的关键指标。一般来说，如果你的输入速率比你的处理率高得多，那么就需要扩展你的集群或应用程序。根据你的资源管理器和部署情况，可能需将执行器动态添加到你的应用程序中。当时机到来，你也可以用相同的方式缩减你的应用程序，删除执行器（可能通过你的云服务提供商的资源管理器）或以较低的资源数重新启动你的应用程序。 这些更改可能会导致一些处理延迟（因为当执行器被删除时，数据会被重新计算或分区）。最后，关于是否值得创建一个具有更复杂资源管理功能的系统是一个商业决策。

虽然有时需要对集群或应用程序进行底层结构的更改，但有时候的更改可能只需要用新的配置来重新启动应用程序或流。例如，在流处理正在运行时，不支持更改spark.sql.shuffle.partitions配置（即使更改但实际上也不会改变随机分区的数量），这需要重新启动流处理，而不一定是整个应用程序，更重量级的更改（如更改Spark应用程序配置）可能需要重新启动应用程序。

度量和监视

流处理应用程序中的度量和监视与第18章中介绍到的针对一般Spark应用程序的度量（Metrics）与监视（Monitoring）工具相同，但是结构化流处理增加了更多选项，以帮助你更好地了解流处理应用程序的状态。你可以用两个关键 API 来查询流查询的状态，并查看其最近的执行进度。借助这两个API，可以了解你的流处理是否按照预期运行。

查询状态

查询状态是最基本的监视API，因此我们从这里开始介绍，它回答“我的流正在执行什么处理？”这样的问题，此信息可以在`startStream`返回的查询对象的`status`字段中报告。例如，假如有一个简单的计数流，它提供由下列查询定义的IOT设备的计数（这里我们使用上一章中的相同查询，省略了初始化代码）：

```
query.status
```

要获取给定查询的结果状态，只需运行`query.status`命令即可返回流的当前状态，它返回数据流在那个点当前正在发生事情的细节。以下是查询此状态时将返回的结果：

```
{
  "message" : "Getting offsets from ...",
  "isDataAvailable" : true,
  "isTriggerActive" : true
}
```

上面的代码段描述了从结构化流数据源获取偏移量（因此“消息”字段描述为获取偏移量）。有各种各样的消息来描述流的状态。

我们在这里演示了在Spark shell中调用status命令的方式，但是对于独立应用程序，你可能没有附加shell来运行任意代码。 在这种情况下，你可以通过实现监视服务器来显示其状态。例如一个监听某端口的小型HTTP服务器，该服务器在监听端口收到请求时返回`query.status`。或者，你可以使用稍后描述的`StreamingQueryListener` API来监听更多事件，它提供更丰富的功能接口。

近期进度

虽然查询当前状态很有用，但查看查询执行进度的能力同样重要。进度API（progress API）允许我们回答诸如“我们处理元组的速度怎样？”或“元组从数据源到达的速度有多快？”等问题。通过运行query.recentProgress，你将获得更多基于时间的信息，如处理速率和批处理持续时间。流查询的进度信息还包括有关数据流的输入源和输出接收器的信息。

```
query.recentProgress
```

以下是我们运行代码之后的版本的结果，版本的结果类似：

```
Array({
  "id" : "d9b5eac5-2b27-4655-8dd3-4be626b1b59b",
  "runId" : "f8da8bc7-5d0a-4554-880d-d21fe43b983d",
  "name" : "test_stream",
  "timestamp" : "2017-08-06T21:11:21.141Z",
  "numInputRows" : 780119,
  "processedRowsPerSecond" : 19779.89350912779,
  "durationMs" : {
    "addBatch" : 38179,
    "getBatch" : 235,
    "getOffset" : 518,
    "queryPlanning" : 138,
    "triggerExecution" : 39440,
    "walCommit" : 312
  },
  "stateOperators" : [ {
    "numRowsTotal" : 7,
    "numRowsUpdated" : 7
  } ],
  "sources" : [ {
    "description" : "FileStreamSource[/some/stream/source/]",
    "startOffset" : null,
    "endOffset" : {
      "logOffset" : 0
    },
    "numInputRows" : 780119,
    "processedRowsPerSecond" : 19779.89350912779
  } ],
  "sink" : {
    "description" : "MemorySink"
  }
})
```

正如你从刚才所显示的输出中所看到的，这包括有关流状态的许多详细信息。需要注意的是，这是一个某个时间点的快照（根据我们询问进度的时间）。为了持续获得有关流处理的状态，你需要重复执行此API查询以获取更新的状态。查询结果输出中的大部分字段是容易理解的，但是，我们来详细介绍一下更为重要的字段。

输入速率和处理速率

输入速率（input rate）是指数据从输入源流向结构化流处理系统的速度，处理速率（process rate）是应用程序处理分析数据的速度。在理想的情况下，输入速率和处理速率应变化一致。另一种情况是输入速率远大于处理速率，当这种情况发生时，流处理就延迟落后了，你需要扩展集群以处理更大的负载。

每个批次的处理时间

几乎所有流式传输系统都使用批处理以任何合理的吞吐量运行（有些可选择高延迟以换取更低的吞吐量）。结构化流处理实现了两者，随着时间的推移，结构化流处理会以不同的批量大小来处理事件，因此你可能会看到批处理每个批次处理时间的持续变化。当执行选项为连续处理时，此度量值就没有太大意义。

一般来说，最好的做法是将每个批次的持续时间以及输入和处理速率的变化用可视化的方法演示出来，相对于持续的文本报告，可视化方法可以更好帮助用户理解数据流的状态。

Spark用户界面

第18章中详细介绍了Spark的Web UI，Spark用户界面还可以显示结构化流处理应用程序的任务、作业和数据处理的各项指标。在Spark用户界面上，每个流处理应用程序将显示为一系列短作业，每个都有一个触发器。但是，你可以使用相同的 UI 来查看应用程序中的指标、查询计划、任务持续时间和日志。DStream API与结构化流处理的一个区别是结构化流处理不使用流标签（Streaming Tab）。

警报

了解和查看结构化流查询的度量标准是重要的第一步，这涉及到持续监视仪表板和各种指标，以发现潜在的问题。你需要强大的自动警报功能，以便在作业失败时通知你，或者在处理速率跟不上输入数据速率的情况下通知你，整个过程是自动的，不需要人工参与。将现有警报工具与Spark集成有几种方法，通常是基于我们之前介绍的近期进度（recent progress）API。例如，你可以直接将度量标准提供给监控系统，如开源的Coda Hale Metrics库或Prometheus，或者你可以简单地记录它们并使用Splunk等日志聚合系统。 除了对查询过程进行监视和警报之外，你还可以监视集群运行状态和整个应用程序（如果你一起运行多个查询），并提供警报服务。

使用流侦听器进行高级监视

我们已经接触过结构化流处理中的一些高级监控工具，通过一些胶连逻辑（glue logic），你可以使用queryProgress API将监视事件输出到你所选的监视平台上（例如日志聚合系统或Prometheus仪表板）。除了这些方法之外，还有一种更低级但更强大的方式来监视应用程序的执行过程，这就是使用StreamingQueryListener类。

StreamingQueryListener类允许你从流查询中接收异步更新，以便自动将此信息输出到其他系统，并实现强大的监视和警报机制。首先开发自己的对象来扩展StreamingQueryListener，然后将其附加到正在运行的SparkSession。一旦使用sparkSession.streams.addListener()附加了自定义侦听器，你的类将在查询开始或停止时收到通知，或者在活跃查询上有进度变化时收到通知。以下是来自结构化流处理文档的一个监听器示例：

```
val spark: SparkSession = ...

spark.streams.addListener(new StreamingQueryListener() {
    override def onQueryStarted(queryStarted: QueryStartedEvent): Unit = {
        println("Query started: " + queryStarted.id)
    }
    override def onQueryTerminated(
      queryTerminated: QueryTerminatedEvent): Unit = {
        println("Query terminated: " + queryTerminated.id)
    }
    override def onQueryProgress(queryProgress: QueryProgressEvent): Unit = {
        println("Query made progress: " + queryProgress.progress)
    }
})
```

流侦听器使你可以用自定义代码处理每次进度更新或状态更改，并将其传递给外部系统。例如，以下StreamingQueryListener的实现代码是用于将所有查询进度信息转发给Kafka，一旦从Kafka读取数据以便获取实际度量指标，你必须解析此JSON字符串：

```
class KafkaMetrics(servers: String) extends StreamingQueryListener {
  val kafkaProperties = new Properties()
  kafkaProperties.put(
    "bootstrap.servers",
    servers)
  kafkaProperties.put(
    "key.serializer",
    "kafkashaded.org.apache.kafka.common.serialization.StringSerializer")
  kafkaProperties.put(
    "value.serializer",
    "kafkashaded.org.apache.kafka.common.serialization.StringSerializer")
```

```
    val producer = new KafkaProducer[String, String](kafkaProperties)

    import org.apache.spark.sql.streaming.StreamingQueryListener
    import org.apache.kafka.clients.producer.KafkaProducer

    override def onQueryProgress(event:
      StreamingQueryListener.QueryProgressEvent): Unit = {
      producer.send(new ProducerRecord("streaming-metrics",
        event.progress.json))
    }
    override def onQueryStarted(event:
      StreamingQueryListener.QueryStartedEvent): Unit = {}
    override def onQueryTerminated(event:
      StreamingQueryListener.QueryTerminatedEvent): Unit = {}
  }
```

通过使用StreamingQueryListener接口，你甚至可以运行结构化流处理应用程序来监视同一个（或另一个）集群上的结构化流处理应用程序。你也可以用这种方式管理多个流。

小结

在本章中，我们介绍了在生产环境中运行结构化流处理所需的主要工具，即容错检查点和各种允许你观察应用程序的运行状态的监视API。在面向生产环境的Spark中，许多概念和工具是相似的，所以你应该能够运用你已有的各种知识来帮助部署流处理应用程序。一定要阅读第Ⅳ部分，看看其他有用的工具，以监测Spark应用程序。

第 VI 部分

高级分析与机器学习

第24章

高级分析和机器学习概览

到目前为止，我们已经概述了常用的数据流API，该部分开始将深入探讨Spark提供的更有针对性的高级数据分析API。除了大规模SQL分析和流处理，Spark还提供了对数据统计、机器学习和图分析的支持，我们称这些任务为高级数据分析任务。本部分将涵盖Spark的高级数据分析工具，包括：

- 数据预处理（数据清洗和特征工程）。
- 监督学习。
- 推荐系统。
- 无监督学习。
- 图分析。
- 深度学习。

本章将概述高级数据分析，提供一些应用场景，以及介绍高级分析任务的基本流程。之后我们将介绍上面列出的分析工具，并介绍如何应用它们。

本书不打算从基础开始介绍关于机器学习的一切知识，不会涉及到严格的数学定义和公式，这并不是因为它们不重要，而是因为其中包含太多的内容本书无法全部涵盖。本书不是一本教你每一个算法数学基础的算法指南，也没有深入的实现策略分析。这里包含的章节为用户提供了一个指南，目的是概述使用。

高级分析简介

高级分析是指各种旨在发现数据规律，或根据数据做出预测和推荐等核心问题的技术。机器学习最佳的模型结构要根据要执行的任务制定，最常见的任务包括：

- 监督学习，包括分类和回归，其目标是根据数据项的各种特征预测每个数据项的标签。
- 推荐系统，根据行为向用户推荐产品。
- 无监督学习，包括聚类，异常检测，以及主题建模，其目的是发现数据中的结构。
- 图分析任务，如发现社交网络中的模式。

在详细讨论Spark的API之前，让我们了解一下这些机器学习和高级数据分析任务的应用场景。虽然我们努力使这部分的介绍尽可能通俗易懂，但有时你可能需要查阅其他资料来完全理解其中所指。此外，我们将在之后的几章里引用以下的书籍，因为他们是学习数据分析非常好的资源（这些资料可以在网上免费获得）：

- 《An Introduction to Statistical Learning》，Gareth James，Daniela Witten，Trevor Hastie和Robert Tibshirani。。我们称这本书为“ISL”。
- 《Elements of Statistical Learning》，Trevor Hastie，Robert Tibshirani和Jerome Friedman。我们称这本书为“ESL”。
- 《Deep Learning》，Ian Goodfellow，Yoshua Bengio和Aaron Courville。我们称这本书为“DLB”。

监督学习

监督学习可能是最常见的机器学习任务，它的目标很简单：使用有标签的历史数据（通常被称为因变量）来训练模型，基于该模型和新数据点的各特征来预测该数据点的标签。一个例子是根据一个人的年龄（特征）预测他的收入（因变量）。它的训练过程一般是通过梯度下降这种优化算法实现的，训练算法从一个初始基本模型开始，并且在每次迭代期间会调整模型的各参数来逐渐提升模型准确度。这一过程的结果是一个训练好的模型，可以利用它来对新数据进行预测。在训练和做出预测的过程中我们需要完成很多不同的任务，如在使用模型之前需要测试训练好的模型的效果，其基本原理很简单：基于历史数据进行训练，确保它在未训练过的数据上的泛化性，然后在新数据上预测。

我们可以根据要预测的变量类型进一步组织监督学习，之后会提到这个内容。

分类

分类是监督学习的一种常见任务，分类是训练一个算法来预测因变量的类别（属于离散的，有限的一组值）的行为。最常见的情况是二元分类，模型预测一个给定的项属于两组中的哪一个。典型的例子是辨别垃圾邮件。使用一组已经确定为垃圾邮件或非垃圾邮件的历史电子邮件，训练一个模型来分析这些邮件其中的单词和各种属性，并对它们进行预测。一旦模型的性能让我们满意，我们就使用该模型来对模型从未见过的邮件进行预测，自动判别它是否是垃圾邮件。

当我们分类的项多于两类时，我们称其为多分类问题。例如，我们有四种不同类型的电子邮件（而不再是之前提到的两个类别）：垃圾邮件，私人邮件，工作相关邮件和其他。有很多分类任务的应用场景，包括：

预测疾病

医生或医院可能有患者的行为和生理特征的历史数据集，他们可以用这份历史数据集来训练模型（在应用之前需要评估它的准确度和伦理问题），然后利用它来预测患者是否有心脏疾病。这可以是个二分类问题（健康或不健康），也可以是一个多分类问题（健康的心脏，或不同的疾病之一）。

图像分类

有许多来自像苹果、谷歌、脸书公司的应用，它们通过你过往上传的人物图片训练出的一个分类模型，来预测给定一张图片中的人物。另一种常见应用场景是图片的分类或标注图片中的对象。

预测客户流失

一个更商业化的用例是预测客户流失，预测哪些客户可能不再使用这项服务。你可以基于已经流失和未流失的客户数据集，训练出一个二元分类模型，并用它来预测当前客户是否有可能会流失。

买还是不买

公司经常想预测网站的访问者会不会购买某产品。为了实现这种预测，它们可能会用到用户浏览模式或者用户位置等特征。

还有其他很多分类任务的应用场景。我们会在第26章介绍更多的应用场景和Spark的分类API。

回归

在分类问题中因变量是一系列离散的值，而在回归问题中，我们要预测连续的变量（实数）。最简单的情况是想预测出某一个实数值而非一个类别，其余的过程大致相同，这就是为什么他们都是监督学习，我们要基于历史数据来预测我们从未见过的数据。回归有很多典型的应用场景：

销售预测

商店想利用历史销售数据来预测商品销售情况。销售情况可能会和很多因素有关，也就是说可能会基于很多输入变量，但是一个简单的例子就是基于上周的销售数据来预测下周的数据。

身高预测

根据父母的身高，要预测他们孩子的身高。

预测节目的观众人数

像Netflix这样的多媒体公司要预测会有多少用户观看某一个节目。

我们会在第27章介绍更多的应用场景和Spark的回归方法。

推荐系统

推荐系统是高级分析最直观的应用之一。通过研究用户对多种商品的显式偏好（通过评级）或隐式偏好（通过观察到的行为），基于用户之间的相似性或商品之间的相似性来推荐给用户他们可能喜欢的商品。通过查看这些相似性，推荐系统可以把其相似用户喜欢的商品推荐给他，或者把其相似用户已经购买的商品推荐给他。推荐系统是Spark任务中很常见的应用场景，并且Spark非常适合处理大数据推荐。下面是一些推荐系统的应用场景：

电影推荐

Netflix虽然不一定使用Spark来处理大数据，但Netflix确实要为其用户做大规模推荐，它通过研究用户在Netflix上看过和不看什么电影的观影记录来做预测。此外，Netflix公司有可能考虑到用户之间打分模式的相似性。

产品推荐

亚马逊将产品推荐作为提高销售额的主要手段之一。例如，根据购物车中的物品，亚马逊推荐与曾经加入过购物车的类似的其他物品。同样，每一个产品页面上，亚马逊展示出由其他用户购买的同类产品。

我们会在第28章介绍更多的推荐系统应用场景和Spark生成推荐的方法。

无监督学习

无监督学习是试图在一组给定的数据中寻找模式或发现隐层结构的方法。这不同于监督学习，因为没有因变量（标签）来做预测。

一些使用无监督学习的应用场景包括：

异常检测

鉴于一些常规事件经常发生，我们可能希望当非常规事件发生时给出预警。例如，安全管理人员可能希望当路上出现一个奇怪物体（损坏车辆，溜冰鞋或骑自行车的人）的时候收到通知。

用户分类

给定一组用户的行为数据，我们可能要更好地了解某些用户与其他用户共享哪些特性。例如，游戏公司可能基于像在某游戏中花费的时间来聚类用户。该模型可能揭示休闲玩家与铁杆玩家完全不同的行为模式，并根据这种差异性，给每个玩家提供不同的建议或奖励。

主题建模

给定一组文件，我们可以分析其中所含的词组来看看他们之间是否有某种潜在关系。例如，提供一些关于数据分析的网页，主题建模算法可以基于一个主题中比较常见的词将这些网页标记成机器学习主题，SQL主题，流处理主题的页面等。

很容易了解到，用户分类可以帮助一个平台更好地迎合每一组的用户。但是，很难判断这种用户分类的方法是否是“正确的”。出于这个原因，我们很难判别一个特定的模型是好是坏。我们将在第29章详细讨论无监督学习。

图分析

虽然相比于分类和回归，图分析不是很常用，但它是一个强大的工具。从根本上讲，图分析是在研究给定顶点（对象）和边（表示这些对象之间的关系）的结构。例如，顶点可能代表了人与产品，边可能代表购买行为。通过观察顶点和边的属性，我们可以更好地理解它们和图形整体结构之间的关联。由于图表达的是关系，任何可以抽象成关系的数据都可以作为图分析的一个应用场景，包括：

欺诈预测

Capital One公司采用Spark的图分析功能更好的了解欺诈网络，通过使用历史欺诈信息（如电话号码，地址或名字），他们发现欺诈信贷的请求或交易。举例来说，一个欺诈电话号码的两跳范围内（电话拨出方和接听方构成一个边连接，从某电话号码A拨出给B，B再拨出给C，则B和C都处于该电话号码的两跳范围内）的任何用户帐户可能会被认为是可疑的。

异常检测

通过观察个体之间的网络连接方式，可以标记出异常值，以便进行手动分析。例如，如果在我们的数据中通常每个顶点都有10条边，而给定的一个顶点只有一条边相连，这可能是值得研究的奇怪现象。

分类

给定一个网络中顶点的属性信息，就可以根据其他点与该点的连接情况来对其他点进行分类。例如，如果某个人被标记为社会网络中的一个影响力者，我们可以将其他具有类似网络结构的人归类为有影响力者。

推荐

谷歌的原始网页推荐算法PageRank是以分析网站关系来对网页重要性排名的图算法。例如，有很多链接的网页比没有链接的网页更重要。

我们将在第30章讨论更多的图分析应用场景。

高级分析过程

你应该牢牢掌握机器学习和高级分析的一些基本应用场景，但是了解应用场景只是实际高级分析过程的一小部分。在准备数据进行分析、测试不同的建模方法、以及评估这些模型方面，还有很多工作要做。本节将涵盖整个分析过程和我们要采取的步骤，这涉及很多，不仅仅是刚刚提及的任务之一，而且要客观实际地评估其效果，以判定我们是不是要将该模型应用于真实场景（见图24-1）。

整个过程涉及以下步骤（有时候稍有不同）：

1. 搜集与你的任务相关的数据。
2. 清理和检查数据以更好地理解它。
3. 执行特征工程以使数据以适合的形式为算法使用（例如，将数据转换为数值向量）。

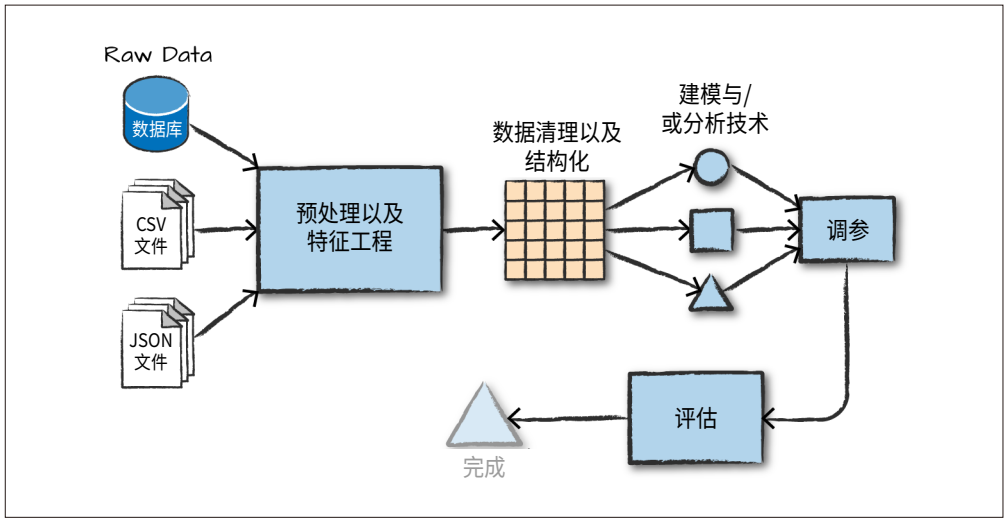


图24-1：机器学习工作流程

4. 使用该数据的一部分作为训练集训练一个或多个模型，生成一些候选模型。

5. 利用从未被用作训练的数据子集来实际客观地评价结果，从而评估比较模型的效果，这可以让你更好地了解模型到底怎么样。

6. 利用上述过程的结果和/或使用模型进行预测、检测异常、或解决更通用的业务难题。

这些步骤并不是对每一个高级分析任务都适用，但是此工作流确实是一个通用框架可以助你在高级分析中获得成功。正如我们在本章前面对各种高级分析任务所做的那样，让我们分解流程以更好地理解每个步骤的总体目标。

数据采集

当然，不先收集数据就无法创建训练数据集，通常这意味着至少要收集用来训练模型的数据集。Spark是个很棒的工具，因为它支持多种数据源和并能够处理各种大小的数据集。

数据清理

收集合适的数据后，你将需要清理和检查它，这通常是所谓探索性数据分析或EDA的一部分。EDA一般是指采用交互式查询和可视化的方法，以便更好地了解数据的分布、数据相关性和其他细节。在这个过程中，你可能会发现你需要删除一些未标注的数据或者可能错误标注的数据。无论是哪种情况，了解你的数据中有什么是一件好

事，这样可以避免在数据方面犯各种错误。结构化API中的许多Spark函数都提供一种简单的方式来清洗和报告数据。

特征工程

收集和清理数据集之后，需要将它转换成适合于机器学习算法一种形式，这通常是数值特征。特征工程可以造就也可以毁掉一个机器学习应用，所以这是要认真对待的一个任务。特征工程的过程包括各种任务，诸如正则化数据，增加变量来表示其他变量的相互作用，操纵类别变量，并将它们转换为适当的格式以输入到我们的机器学习模型。在MLlib（Spark的机器学习库）中，所有的变量通常必须作为浮点型向量输入（不管他们实际上代表什么）。我们在第25章将深入探讨特征工程的过程，Spark提供了使用多种机器学习统计技术来处理数据。

下面的几个步骤（模型训练校正和评估）并不适用于所有场景，这是一个一般的工作流程，根据你希望实现的最终目标可能会有很大的偏差。

训练模型

现在我们拥有了历史信息的数据集（例如，垃圾邮件或不是垃圾邮件），我们也有一个具体的任务（例如，分类垃圾邮件）。下一步，我们将要训练一个模型来根据输入预测正确的输出。在训练过程中，模型的内部参数将根据模型对输入数据的分类效果发生变化。举例来说，要检测垃圾邮件，我们的算法可能会发现某些词的出现更能用来识别垃圾邮件，因此模型中与这些词相关的权重会更高。最后，训练之后的模型发现，某些单词在分类邮件方面比其他词有更大的影响力（因为它们与垃圾邮件有一些相关性）。训练过程的输出就是我们所说的模型，然后模型可被用于理解数据或做未来的预测。为了做出预测，你需要给模型输入数据，它会基于对这些输入数据的数学运算来产生输出。以分类为例，给定邮件的属性，它会通过比较训练过的历史垃圾邮件和非垃圾邮件来预测该邮件是否为垃圾邮件。

然而，仅仅训练模型并不是最终目标，我们要充分利用我们的模型理解数据的本质。因此，我们必须回答这个问题：我们的模型是怎么完成任务的呢？这就是模型的调整和评估。

模型调整和评估

你可能注意到前面我们提到的，你应该将数据分割成多个部分，并且只使用其中一部分进行训练。这是在机器学习过程中的一个重要步骤，因为当你建立一个高级分析模

型，需要能泛化到它以前没有见过的数据。将我们的数据集分成多个部分，使我们能够客观地对其中一部分数据（一组未用来训练模型的数据）对训练后的模型的有效性进行测试，目标是看你的模型是否了解有关此数据的本质，或者它是否只注意到特定于训练集的内容（又称为过拟合），这就是它为什么把被称为测试集。在训练模型的过程中，我们还可能采取另一个单独的数据子集，并将其视为另一种类型的测试集(称为验证集)，以便尝试不同的超参数(影响训练过程的参数)，并在没有过拟合到测试集的情况下比较相同模型的不同变化。

配置适当的训练集、验证集和测试集对成功运用机器学习解决问题是非常重要的。如果我们不正确地分割这些数据集，那么就很容易导致过拟合（模型对新数据泛化能力弱）。这本书中我们不会涵盖该问题，但是几乎所有的机器学习书籍都会讲到。

为了继续使用前面介绍的分类示例，我们有三组数据：训练模型的训练集、用于测试我们所训练模型的不同变体的验证集，以及用于测试哪个模型变体最好的测试集。

使用模型

在训练后得到一个模型之后，你现在就可以使用模型了！把你的模型用于实际生产中，可能是一个巨大的挑战。我们将在本章后面讨论一些策略。

Spark的高级分析工具包

前面概述的仅仅是一个任务流程的例子，还有很多情况或潜在的工作流程没有涵盖。此外，你可能会注意到，我们几乎根本没有讨论Spark。本节将讨论Spark对高级数据分析的支持，Spark包括执行高级分析的几个核心包和许多扩展包，最重要的是MLlib，它提供了用于构建机器学习流程的接口。

什么是MLlib？

MLlib基于Spark，并属于Spark项目的一个软件包，它提供各种API接口用于收集和清理数据、特征工程和特征选择、训练和微调大型有监督和无监督机器学习模型、并在生产中使用这些模型。

MLlib实际上由两个利用不同的核心数据结构的包组成。`org.apache.spark.ml`软件包包括使用DataFrame的接口，还提供了用于构建机器学习流程的高层次接口，它将有助于标准化执行上述步骤。较低级别的软件包`org.apache.spark.mllib`，包括Spark低级别的RDD API接口。这本书将专注于DataFrame API。RDD API是低级别的接口，它处于维护模式，也就是说处于更新bug而不会再有新功能。这在以前的Spark书籍里面涉及很多了，所以在这里我们忽略。

何时以及为什么要使用MLlib（相比于scikit-learn,Tensorflow或foo软件包）

MLlib 可能听起来像很多你听说过的其他机器学习库，例如 Python的 scikit-learn或各种执行类似任务的 R 软件包，那你为什么要使用 MLlib？基于单机有许多用于执行机器学习任务的工具，虽然有几个很好的工具可供选择，但这些基于单机的工具或者无法训练的大数据，或者处理时间太长。这意味着单机工具通常是MLlib的补充，当你遇到这些特别需要扩展性的问题的时候，就要利用Spark的能力。

利用Spark的可扩展能力，有两个关键的应用场景。第一个，你希望利用Spark进行数据预处理和特征生成，以减少从大量数据中生成训练和测试集所需的时间，你可以利用单机学习库对这些给定的数据集进行训练。第二个，当输入数据或模型变得太难或不方便在单机上处理时，可以使用Spark，Spark使分布式机器学习变得非常简单。

一个重要提示是，虽然利用Spark可以使训练和数据预处理工作变得很简单，但仍需要牢记一些复杂性，尤其是在部署训练好的模型时。例如，Spark并没有提供内置的方式来支持模型中的低延迟预测，因此你可能希望将模型导出到另一个系统或自定义应用程序来做到这一点，MLlib 通常支持导出模型到其他工具。

高级MLlib概念

在MLlib有几个基本的“结构”类型：转换器（transformer），估计器（estimator），评估器（evaluator）和流水线（pipeline）。通过结构化的表示，在定义端到端的机器学习流水线时你会考虑这些类型，他们将提供共同的语言来定义哪些属于流水线的哪一部分。图24-2说明使用Spark开发机器学习模型时需遵循的总体开发流程。

转换器是将原始数据以某种方式进行转换的函数。它可能会（从另外两个变量）创建一个新的变量，对某一列进行归一化，或仅仅将一个`Integer`类型值变为`Double`类型值输入模型。转换器的一个例子是将字符串类型变量转换为MLlib能够使用的数值型值，它主要用于数据预处理和特征工程阶段，以DataFrame作为输入并生成一个新的DataFrame作为输出，如图24-3所示。

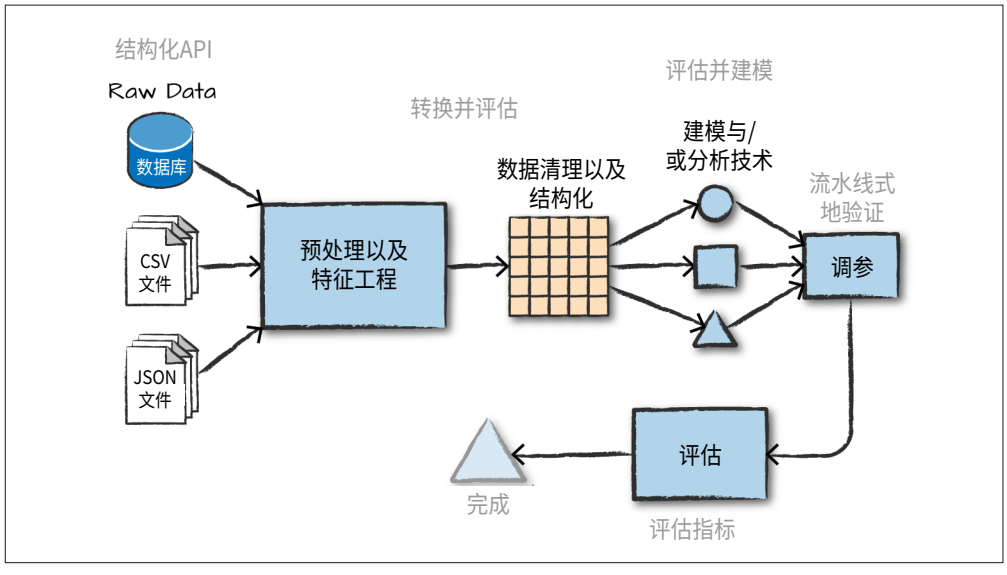


图24-2：Spark中的机器学习工作流

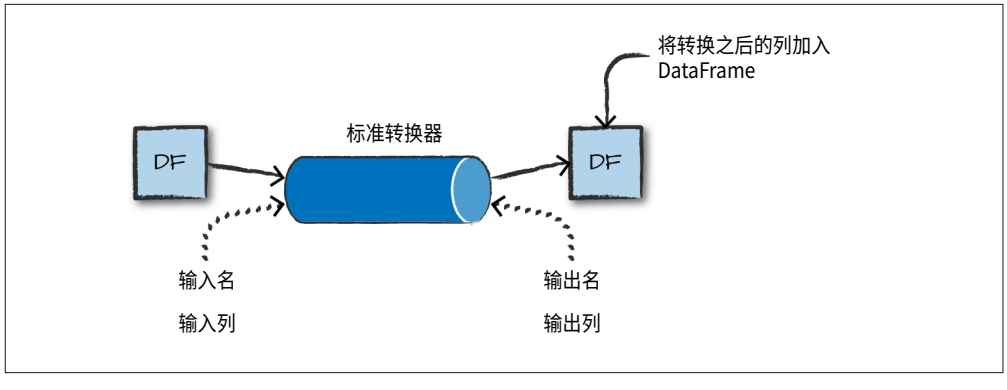


图24-3：标准转换器

估计器可能用来做两件事其中之一。第一，估计器可以做为数据初始化的转换器。例如，为了对数值数据进行归一化，我们需要基于某列中的当前值来初始化我们的转换器。这需要传递两次数据，第一次传递生成初始化值，第二次实际在数据上应用生成函数。第二，根据Spark的命名规则，基于数据训练模型的算法也称为估计器。

评估器允许我们根据某种效果评价指标（如受试者工作特征曲线，即ROC曲线）来评价给定模型的表现如何。在使用评估器从我们测试的模型中选择最佳模型之后，就可以使用该模型进行预测。

从较高的层次上，我们可以一个一个地指定转换器、估计器和评估器，但是也可以把我们的步骤指定为流水线中的阶段，这种流水线类似于 scikit-learn 中的流水线的概念。

低级别的数据类型

除了构建流水线的结构类型外，还有几种低级别的数据类型可能在 MLlib 中使用(Vector是最常见的)。每当我们将一组特征数据传递到机器学习模型中时，我们必须将其组织成Double类型的向量，此向量可以是稀疏的 (其中大多数元素为零)，也可以是稠密的(其中有许多非重复值)。向量是以不同的方式创建的，要创建稠密向量，我们需要指定数组中的所有值，要创建稀疏向量，我们可以指定向量的大小、索引和非零元素的值。你可能已经猜到，稀疏是大多数值为零时情况下的最好表达形式，因为这是一种更易压缩的表示形式。下面是如何手动创建向量的示例：

```
// in Scala
import org.apache.spark.ml.linalg.Vectors
val denseVec = Vectors.dense(1.0, 2.0, 3.0)
val size = 3
val idx = Array(1,2) // 向量中非0元素的位置
val values = Array(2.0,3.0)
val sparseVec = Vectors.sparse(size, idx, values)
sparseVec.toDense
denseVec.toSparse

# in Python
from pyspark.ml.linalg import Vectors
denseVec = Vectors.dense(1.0, 2.0, 3.0)
size = 3
idx = [1, 2] # 向量中非0元素的位置
values = [2.0, 3.0]
sparseVec = Vectors.sparse(size, idx, values)
```

令人费解的是，存在可在 DataFrame中使用和其他只能在 RDD 中使用的数据类型。mllib包中包含RDD的实现，而DataFrame的实现位于ml包中。

MLlib的执行

前面我们已经概述了你可能会遇到的核心概念，下面就来创建一个简单的流水线来演示每个组件。我们将使用一个规模较小的合成数据集，这将有助于说明。在进一步讨论之前，我们先读取数据：

```
// in Scala
var df = spark.read.json("/data/simple-ml")
df.orderBy("value2").show()

# in Python
df = spark.read.json("/data/simple-ml")
df.orderBy("value2").show()
```

这是数据样本：

```
+-----+----+------+------------------+
|color| lab|value1|            value2|
+-----+----+------+------------------+
|green|good|     1|14.386294994851129|
...
|  red| bad|    16|14.386294994851129|
|green|good|    12|14.386294994851129|
+-----+----+------+------------------+
```

此数据集包含以下标签、好或坏的分类lab、代表颜色的分类color，以及两个数值变量value1和value2。虽然数据是合成的，让我们想象一下，这个数据集代表公司客户的健康状况，其中color列代表由客服代表所做的某种类别的健康评级，lab列代表真正的客户健康状况，另外两个值是一些数值型的行为度量 (例如，在线时间和花费开销)。假设我们想要训练一个分类模型，我们希望根据这些值预测一个二元变量（标签）。

除了JSON，还有一些特定的数据格式用于监督学习，包括 LIBSVM。这些格式具有真正的值标签和稀疏输入数据，Spark可以使用其数据源 API 读取和写入这些格式。下面是一个如何使用数据源 API

```
spark.read.format("libsvm").load(
  "/data/sample_libsvm_data.txt")
```

有关LIBSVM的更多信息，请参阅该相关文档。

转换器执行特征工程

如前所述，转换器帮助我们以某种方式操纵当前的列数据，操纵这些列通常是为了构建特征（我们会将它们输入到模型中）。转换器可以用来减少特征的数量、添加更多特征、操作当前的列或简单地帮助我们纠正数据格式，转换器将向 DataFrame 添加新列。

当我们使用Mllib时，所有Spark机器学习算法（在后面的章节中会讨论一些例外情况）的输入是由Double类型（表示标签）和Vector[Double]类型（表示特征）组成。

当前的数据不符合这一要求，因此我们需要将其转换为正确的格式。

要在我们的例子做到这一点，我们要指定一个RFormula。它是指定机器学习转换的声明式语言，用来根据数据集自动生成特征和标签，一旦你理解了语法这其实很简单。RFormula支持R语言运算符的一个有限子集，它对于实际应用中的简单模型和操作有很好的支持（我们会在第25章展示解决这个问题的手动方式）。基本RFormula操作符是：

~
: 目标（标签）和项（特征）的分隔符号。

+
: 合并项，“+ 0”表示删除空格(this means that the yintercept of the line that we will fit will be 0)。

-
: 删除项，“- 1”表示删除空格，(this means that the yinterceptof the line that we will fit will be 0—yes, this does the same thing as “+0”和“+ 0”起相同作用）。

:
: 交互（数值乘法，或类别二值化）。

.
: 除了目标列的全部列。

为了应用具有此语法的转换，我们需要导入相关的类，然后我们需要定义公式。在该例子中，我们希望使用所有列 (.符号)，在value1和color列之间添加交互，在value2和color列之间添加交互，将这些特征视为新特性：

```
// in Scala
import org.apache.spark.ml.feature.RFormula
val supervised = new RFormula()
.setFormula("lab ~ . + color：value1 + color：value2")

# in Python
from pyspark.ml.feature import RFormula
supervised = RFormula(formula="lab ~ . + color：value1 + color：value2")
```

此时我们以声明方式指定了如何将数据更改为模型训练要使用的格式。接下来的步骤是，将RFormula转换器应用到数据上，让它发现每个列的可能值。不是所有的转换器都有这个要求，但因为RFormula会自动处理分类变量，它需要确定哪些列是用于分类

的，哪些不是，还有分类的列的具体值。出于这个原因，我们要调用fit方法。之后，它返回“训练好”的转换器，我们可以使用它来转换数据。

我们使用 RFormula转换器，因为它使一些转换很容易实现。当我们在第25章谈到Mllib的转换器的时候会说明其他类似的转换操作集合，并列出RFormula组成部分。

现在我们说明了这些细节并继续准备DataFrame：

```
// in Scala
val fittedRF = supervised.fit(df)
val preparedDF = fittedRF.transform(df)
preparedDF.show()

# in Python
fittedRF = supervised.fit(df)
preparedDF = fittedRF.transform(df)
preparedDF.show()
```

这是训练和转换过程的输出：

```
+-----+----+------+------------------+--------------------+-----+
|color| lab|value1|            value2|            features|label|
+-----+----+------+------------------+--------------------+-----+
|green|good|     1|14.386294994851129|(10,[1,2,3,5,8],[...|  1.0|
...
|  red| bad|     2|14.386294994851129|(10,[0,2,3,4,7],[...|  0.0|
+-----+----+------+------------------+--------------------+-----+
```

在输出中，我们可以看到我们的转换结果，就是名为features的列，它包含以前的原始数据。后面发生的事情其实很简单，RFormula在调用fit函数时检查我们的数据，并输出一个根据指定的公式转换数据的对象(称为 RFormulaModel)，这个 "训练好" 的转换器在类型签名中始终称为Model。当我们使用这个转换器时，Spark会自动将我们的分类变量转换为Double类型，这样我们就可以将它输入到一个 (尚未指定的) 机器学习模型中。特别是，它为每个可能的color类别分配一个数值，为color和 value1/value2之间的交互变量创建特征，并将它们全部放到一个向量中。然后，我们调用该对象上的transform函数，以便将我们的输入数据转换为想要的输出数据。

至此，你(预) 处理了数据，并添加了一些特征，现在是时候在这个数据集上实际训练一个模型 (或一组模型) 了。为了做到这一点，你首先需要准备一个测试集进行评估。

有一个好的测试集可能是最重要的事情，能确保你训练的模型以一种可靠的方式在实际环境中使用。创建不具有代表性的测试集，或不使用测试集进行超参数优化，一定会导致模型在实际场景下的应用时表现欠佳。不要忽略创建测试集，如果想要了解你的模型的真实表现效果，这非常重要！

我们现在创建一个基于随机拆分的简单测试集 (我们将在本章使用此测试集)：

```
// in Scala
val Array(train, test) = preparedDF.randomSplit(Array(0.7, 0.3))

# in Python
train, test = preparedDF.randomSplit([0.7, 0.3])
```

估计器

现在，我们已经将我们的数据转换为正确的格式，也创建了有用的特征，终于到了使用我们模型的时候了。在本例中，我们将使用一种称为逻辑回归（logistic regression）的分类算法。我们创建一个`LogisticRegression`的实例，使用默认配置和超参数，然后设置标签列和特征列。我们设置的`label`和`features`列，实际上是Spark MLlib 估计器的默认使用的标签，在后面的章节中我们省略它们：

```
// in Scala
import org.apache.spark.ml.classification.LogisticRegression
val lr = new LogisticRegression().setLabelCol("label").setFeaturesCol("features")

# in Python
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(labelCol="label",featuresCol="features")
```

我们实际上在训练这个模型之前，让我们检查一下参数，这也是提示每个特定模型的所有可用参数选项的好方法：

```
// in Scala
println(lr.explainParams())

# in Python
print lr.explainParams()
```

虽然输出太大，无法在此处重现，但它展示了对Spark实现逻辑回归的所有参数的解释，`explainParams`方法在MLlib 可用的所有算法中都有实现。

在实例化未经过训练的算法后，该把数据输入进去了。在本例中，它将返回`LogisticRegressionModel`：

```
// in Scala
val fittedLR = lr.fit(train)

# in Python
fittedLR = lr.fit(train)
```

这段代码将启动一个Spark job来训练模型。与你在书中看到的转换过程相反，机器学习模型的拟合是即刻实现的。

完成后，可以使用模型进行预测。从逻辑上说，这就是从特征到标签的转换。我们使用transform方法进行预测。例如，我们可以transform我们的训练数据集，看看我们的模型对训练数据会输出什么标签，并将其与真实标签进行比较，这又是我们可以操纵的另一个 DataFrame。用下面的代码段来执行该预测：

```
fittedLR.transform(train).select("label", "prediction").show()
```

结果是：

```
+-----+----------+
|label|prediction|
+-----+----------+
|  0.0|       0.0|
...
|  0.0|       0.0|
+-----+----------+
```

我们下一步将是手动评估此模型，并计算性能指标，如真正率（true positive rate），假负率（false negative rate）等。然后，我们可以尝试一组不同的参数以查看这些性能指标是否更好，虽然这个过程很有效，但是很乏味。Spark可以帮助你避免手动尝试不同的模型和性能指标，它允许你将Spark作业指定为包含所有转换的声明式流水线，以及允许调整超参数。

超参数概述

虽然我们前面提到过超参数，现在来正式定义超参数。超参数是影响训练过程的配置参数，诸如模型结构和正则化，它们在训练开始之前被设置。例如，逻辑回归有一个超参数（正则化参数），它决定了我们的数据在训练阶段执行什么程度的正则化 (正则化是一种避免模型过拟合的技术)。你将在接下来的内容中看到，我们以不同的超参数值(例如，不同的正则化参数值)来启动流水线作业，以便比较同一模型不同版本的效果。

流水线化工作流

正如你可能注意到的那样，如果你执行大量的转换，则编写所有步骤并跟踪DataFrame的过程最终会非常繁琐，这就是Spark包含Pipeline流水线概念的原因所在。流水线允许你设置相关转换的数据流，并以估计器结束，估计器可以根据用户指定自动调整，最后得到一个优化后的模型，即可以直接使用。图24-4展示了这一过程。

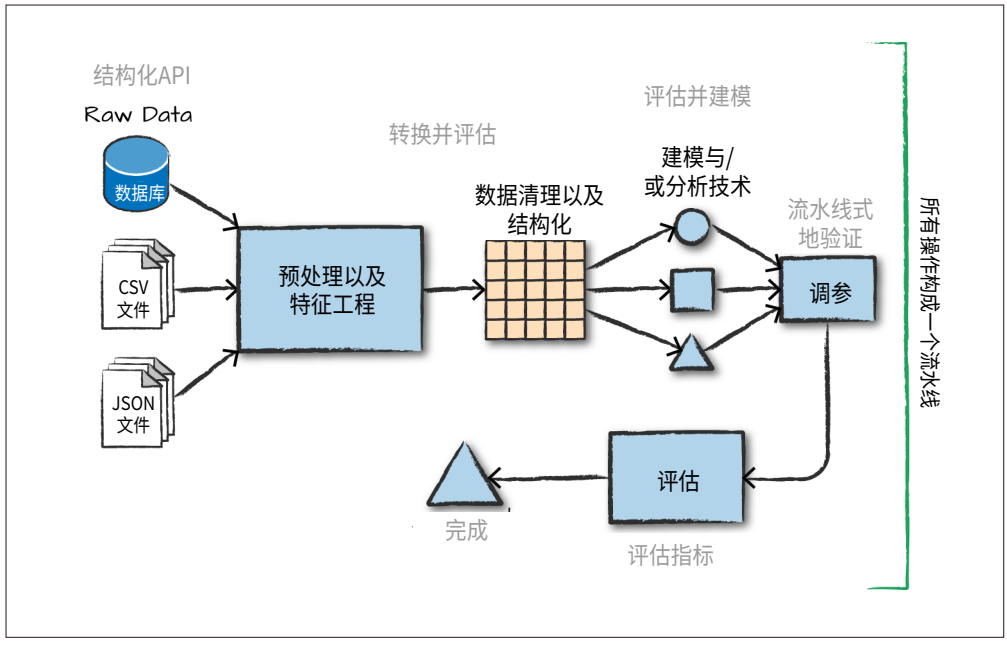


图24-4：流水线式的机器学习工作流

请注意，转换器或模型的实例在不同流水线上不会被重用，在创建另一个流水线之前，始终要创建模型的新实例。

为了确保不会过拟合，我们将创建一个测试集并根据验证集调整我们的超参数(请注意，我们基于原始数据集创建此验证集，而不是前面提到的preparedDF)：

```
// in Scala
val Array(train, test) = df.randomSplit(Array(0.7, 0.3))

# in Python
train, test = df.randomSplit([0.7, 0.3])
```

配置完成后，需要创建流水线中的基本阶段，一个阶段仅仅代表转换器或估计器。在我们的例子中，我们将有两个估计器。创建RFomula 将首先分析我们的数据，以了解

输入特征的类型，然后转换它们以创建新的特征。随后，创建LogisticRegression 对象，它是通过训练来生成模型的算法：

```
// in Scala
val rForm = new RFormula()
val lr = new LogisticRegression().setLabelCol("label").setFeaturesCol("features")

# in Python
rForm = RFormula()
lr = LogisticRegression().setLabelCol("label").setFeaturesCol("features")
```

现在，不是去手动使用我们的转换然后调整我们的模型，而是只需在整个流水线的阶段创建它们，如下面的代码片段所示：

```
// in Scala
import org.apache.spark.ml.Pipeline
val stages = Array(rForm, lr)
val pipeline = new Pipeline().setStages(stages)

# in Python
from pyspark.ml import Pipeline
stages = [rForm, lr]
pipeline = Pipeline().setStages(stages)
```

训练与评估

现在已经设定好了逻辑流水线，下一步是训练。在这个例子中，我们不只训练一个模型，我们将通过指定不同的超参数组合来让Spark训练多个不同的模型。然后，我们将使用评估器选择最佳模型，并将它作出的预测与我们的验证数据进行比较。我们可以测试整个流水线上各种不同的超参数组合，甚至当使用RFormula操作原始数据的时候也是如此。下面代码演示了如何执行此操作：

```
// in Scala
import org.apache.spark.ml.tuning.ParamGridBuilder
val params = new ParamGridBuilder()
  .addGrid(rForm.formula, Array(
    "lab ~ . + color:value1",
    "lab ~ . + color:value1 + color:value2"))
  .addGrid(lr.elasticNetParam, Array(0.0, 0.5, 1.0))
  .addGrid(lr.regParam, Array(0.1, 2.0))
  .build()

# in Python
from pyspark.ml.tuning import ParamGridBuilder
params = ParamGridBuilder()\
  .addGrid(rForm.formula, [
    "lab ~ . + color:value1",
    "lab ~ . + color:value1 + color:value2"])\
  .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
```

```
  .addGrid(lr.regParam, [0.1, 2.0])\
  .build()
```

在当前的参数设置中，有三个超参数与默认值不同：

- 两个不同版本的RFormula。
- 对于ElasticNet参数有三个不同的选择。
- 对于正则化参数有两种不同的选择。

这给了我们总共12种不同的参数组合，意味着我们将训练12个不同版本的逻辑回归模型。我们将在第26章解释Elastic Net参数以及正则化选项。

现在已建立了各种参数组合，是指定我们的评估过程的时候了。评估器将自动地和客观地把多个模型基于同一评估指标进行比较，后面的章节会介绍针对分类和回归的评估器。但在这个例子中，我们使用BinaryClassificationEvaluator，它有许多潜在的评估指标，我们将在第26章中讨论。在这个例子中，我们将使用areaUnderROC，它是受试者工作特征曲线的总面积，这是评估分类任务效果的常用评估指标：

```
// in Scala
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
val evaluator = new BinaryClassificationEvaluator()
  .setMetricName("areaUnderROC")
  .setRawPredictionCol("prediction")
  .setLabelCol("label")

# in Python
from pyspark.ml.evaluation import BinaryClassificationEvaluator
evaluator = BinaryClassificationEvaluator()\
  .setMetricName("areaUnderROC")\
  .setRawPredictionCol("prediction")\
  .setLabelCol("label")
```

现在我们有了指定数据应该如何转换的流水线，我们将对逻辑回归模型执行模型选择，尝试不同的超参数，利用areaUnderROC指标比较性能来评估其效果。

正如我们所讨论的，机器学习中在验证集（而非测试集）上对比超参数效果可以很好地避免过拟合问题，出于这个原因，我们不用（之前创建的）测试集来调整这些参数。Spark提供了自动调整超参数的两种选择，我们可以使用TrainValidationSplit，这将简单地将数据任意随机分成两个不同的组，或使用CrossValidator，其通过将数据集分割成不重叠的，随机分配的k个折叠来执行K-折叠交叉验证（K-fold cross-validation）：

```
// in Scala
```

```
import org.apache.spark.ml.tuning.TrainValidationSplit
val tvs = new TrainValidationSplit()
  .setTrainRatio(0.75) // 这里0.75也是其默认值
  .setEstimatorParamMaps(params)
  .setEstimator(pipeline)
  .setEvaluator(evaluator)

# in Python
from pyspark.ml.tuning import TrainValidationSplit
tvs = TrainValidationSplit()\
  .setTrainRatio(0.75)\
  .setEstimatorParamMaps(params)\
  .setEstimator(pipeline)\
  .setEvaluator(evaluator)
```

下面我们运行构建的流水线，运行该流水线将对每一个模型版本都进行测试。注意`tvsFitted`的类型是`TrainValidationSplitModel`，当拟合一个模型，输出一个“model”类型：

```
// in Scala
val tvsFitted = tvs.fit(train)

# in Python
tvsFitted = tvs.fit(train)
```

当然，要在测试集上评估它的效果！

```
evaluator.evaluate(tvsFitted.transform(test)) // 0.9166666666666667
```

我们也可以查看模型的训练信息总结。要做到这一点，我们要从流水线中提取它，将它转换为适当的类型，并打印结果。在整个接下来的几个章节中我们会对每个模型提供的各项指标进行讨论。下面是我们查看结果的方法：

```
// in Scala
import org.apache.spark.ml.PipelineModel
import org.apache.spark.ml.classification.LogisticRegressionModel
val trainedPipeline = tvsFitted.bestModel.asInstanceOf[PipelineModel]
val TrainedLR = trainedPipeline.stages(1).asInstanceOf[LogisticRegressionModel]
val summaryLR = TrainedLR.summary
summaryLR.objectiveHistory // 0.6751425885789243，0.5543659647777687，0.473776...
```

此处展示了我们的算法在训练中每次迭代的训练效果。这会很有帮助，因为可以观察到我们的算法正逼近最佳模型，通常是在开始时评价指标会有较大的变化，但随着时间的推移，这些值应该变得越来越小。

持久化和应用模型

现在，我们训练好了这个模型，我们可以将它保存到磁盘上，以便在以后的预测中使

用它：

```
tvsFitted.write.overwrite().save("/tmp/modelLocation")
```

在写出模型之后，我们可以将其加载到另一个Spark程序中进行预测。为此，我们需要使用特定算法的“model”版本来从磁盘上来加载我们保存的模型。例如，如果使用CrossValidator，则必须在持久化版本中读取CrossValidatorModel，如果要手动使用LogisticRegression，则必须使用LogisticRegressionModel。在我们的例子中使用TrainValidationSplit，它输出 TrainValidationSplitModel：

```
// in Scala
import org.apache.spark.ml.tuning.TrainValidationSplitModel
val model = TrainValidationSplitModel.load("/tmp/modelLocation")
model.transform(test)
```

部署模式

在Spark中有几种不同的部署模式，用于实际应用机器学习模型。图24-5展示了常见的工作流程。

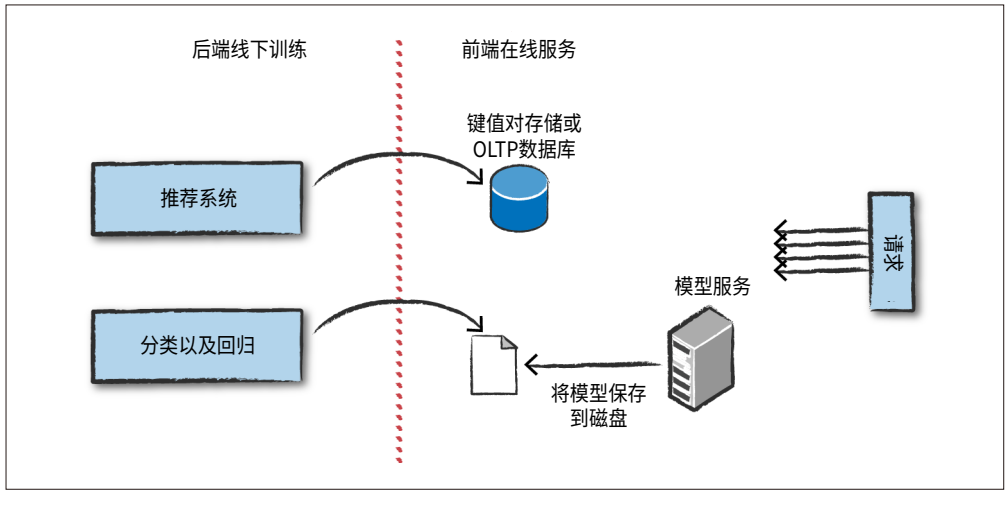


图24-5：实际生产中的应用

下面是关于如何部署Spark模型的各种选项。这些都是一般的选项，可以在图24-5中找到对应的项。

- 离线训练机器学习 (ML) 模型，然后向其提供离线数据。离线数据指的是存储下

来用于分析的数据而不是你需要快速得到应答的数据。Spark很适合这种部署方式。

- 离线训练模型，然后把训练结果放到一个数据库（通常是一个键值存储）中。这非常适用于推荐系统，但不适用于像分类或回归这样的应用，因为这些应用不是为某一用户查询返回一个值，而是必须基于输入计算输出值。
- 离线训练模型，持久化模型到磁盘上用于之后提供服务。如果使用Spark作为服务模块，这将不是一个低延迟的解决方案，因为Spark作业启动很耗时，即使不是在集群上运行。此外它不能很好的并行化，所以你很可能不得不使用负载均衡器，集成多个模型副本提供服务，并集成一些自己的REST API。这个问题有一些有趣的潜在解决方案，但当前还没有标准化的方法。
- 手动（或通过其他一些软件）将分布式模型转换成可以在一台机器上更快运行的模型。当不对Spark中的原始数据做很多操作时这种方法很不错，但可能很难维护和更新。当前也有几种解决方案，例如MLlib可以将某些模型导出为PMML（一种通用的模型交换格式）文件。
- 在线训练算法并在线使用。可以和结构化流处理一起使用，但是对于某些模型来说会很复杂。

虽然这些都是一些可选选项，但也有很多其他方式用于部署模型和管理模型。这是一个正在蓬勃发展的领域，目前有许多潜在的创新工作正在进行。

小结

在本章中，我们讨论了高级分析和 MLlib 的核心概念，还展示了如何使用它们。下一章将深入讨论数据预处理，包括用于特征工程和数据清理的Spark工具。然后，我们将详细描述 MLlib 中的每个算法，以及一些用于图分析和深度学习的工具。

第25章

预处理和特征工程

每一个数据科学家都知道数据分析中最大的挑战（以及最耗时的工作）就是数据预处理，这并不是说它是一个特别复杂的编程过程，而是它需要特别了解你的数据，并了解你的模型需要什么才能利用这些数据。本章介绍了如何使用Spark执行预处理和特征工程的详细内容。我们将介绍需要了解的核心知识，以便根据数据的结构来训练MLlib 模型，并且讨论Spark为执行数据预处理提供的各种工具。

根据应用场景格式化模型

要想为Spark不同的高级分析工具进行数据预处理就要考虑最终的目标。下面的列表说明了 MLlib 中每个高级分析任务的输入数据结构要求：

- 在大多数分类和回归算法中，用`Double`类型的一列数据代表标签，`Vector`类型（可能稠密也可能稀疏）的一列数据代表特征。
- 在推荐系统中，数据由一个用户列，一个项目（观看的电影或购买的书）列，以及一个评分列组成。
- 在无监督学习时，需要使用`Vector`类型（可能稠密也可能稀疏）的一列数据表示特征。
- 图分析中，需要DataFrame数据类型表示顶点和边。

以这些格式表示数据的最好方式是使用转换器。转换器是以DataFrame作为输入输出的函数，本章只涉及了相关的适合某应用场景的特定转换器，而不会涵盖所有转换器。

Spark的org.apache.spark.ml.feature包中有很多转换器，Python对应的包是pyspark.ml.feature. Spark MLlib会持续更新和加入新的转换器，所以本书不可能列举出一个列表。最新的信息可以在Spark 的官方文档上找到。

处理之前需要从几个不同的采样数据集中读取数据，其中的每一个都有我们这一章中需要处理的属性。

```
// in Scala
val sales = spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/data/retail-data/by-day/*.csv")
  .coalesce(5)
  .where("Description IS NOT NULL")
val fakeIntDF = spark.read.parquet("/data/simple-ml-integers")
var simpleDF = spark.read.json("/data/simple-ml")
val scaleDF = spark.read.parquet("/data/simple-ml-scaling")

# in Python
sales = spark.read.format("csv")\
  .option("header", "true")\
  .option("inferSchema", "true")\
  .load("/data/retail-data/by-day/*.csv")\
  .coalesce(5)\
  .where("Description IS NOT NULL")
fakeIntDF = spark.read.parquet("/data/simple-ml-integers")
simpleDF = spark.read.json("/data/simple-ml")
scaleDF = spark.read.parquet("/data/simple-ml-scaling")
```

除了真实的销售数据之外，我们还将使用几个简单的合成数据集，FakeIntDF，simpleDF和scaleDF的行数都很少，这将有助于你专注于正在执行的具体数据操作，而不是数据集之间的差异性。因为我们得多次使用这些销售数据，所以我们将对其进行缓存，以便我们可以从内存中高效地读取它，而不是每次需要时再从磁盘上读取它。下面我们查看数据集的前几行来更好地了解数据格式。

```
sales.cache()
sales.show()

+---------+---------+--------------------+--------+-------------------+---------
|InvoiceNo|StockCode|         Description|Quantity|        InvoiceDate|UnitPr...
+---------+---------+--------------------+--------+-------------------+---------
|   580538|    23084|  RABBIT NIGHT LIGHT|      48|2011-12-05 08:38:00|     1...
...
|   580539|    22375|AIRLINE BAG VINTA...|       4|2011-12-05 08:39:00|     4...
+---------+---------+--------------------+--------+-------------------+---------
```

值得注意的是，我们在这里过滤掉 null 值。当前 MLlib 并不总是能很好地处理null值，这也是常见的报错的原因，也是需要在调试时的首先考虑的。之后每个Spark的版本都会改进对null值的处理。

转换器

我们在前一章中讨论过转换器，但值得在这里重复一下。转换器是以某种方式转换原始数据的函数，这可能是(从两个其他变量)创建一个新的交互变量、规范化一列、或简单地将其转换为Double类型以便输入到模型中。转换器主要用于数据预处理或特征生成。

Spark的转换器只包括一种转换方法，这是因为它不会根据输入数据进行更改。图 25-1 是一个简单的示例，左侧是一个要被转换的输入 DataFrame ，右侧是输出DataFrame，它增加了一个代表转换输出的新列。

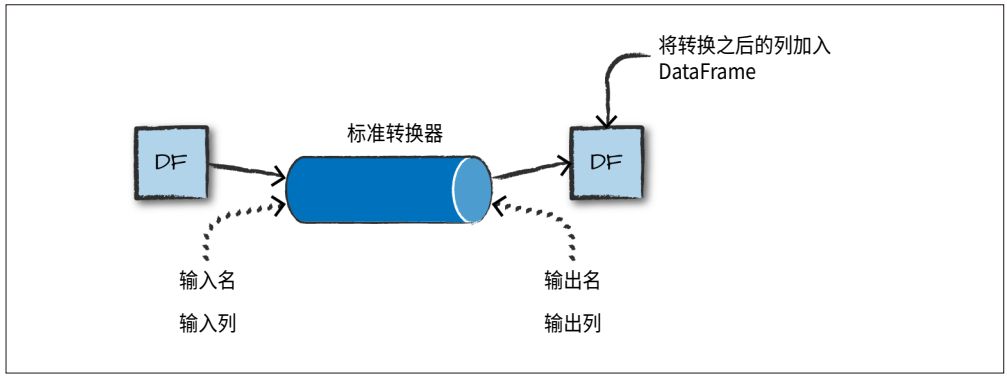


图25-1：Spark的转换器

Tokenizer是转换器的一个例子，它根据给定的字符对一个字符串进行分词，这不会从我们的数据中学习到任何东西，它只是应用了一个函数，我们将在本章后面更深入地讨论分词器。下面是一个代码片断，它展示了如何构建一个分词器来处理输入列，如何转换数据，然后从该转换中输出：

```
// in Scala
import org.apache.spark.ml.feature.Tokenizer
val tkn = new Tokenizer().setInputCol("Description")
tkn.transform(sales.select("Description")).show(false)
```

```
+---------------------------------+-----------------------------------------+
|Description                      |tok_7de4dfc81ab7__output                 |
+---------------------------------+-----------------------------------------+
```

```
|RABBIT NIGHT LIGHT                |[rabbit, night, light]                    |
|DOUGHNUT LIP GLOSS                |[doughnut, lip, gloss]                    |
...
|AIRLINE BAG VINTAGE WORLD CHAMPION |[airline, bag, vintage, world, champion]  |
|AIRLINE BAG VINTAGE JET SET BROWN  |[airline, bag, vintage, jet, set, brown]  |
+-----------------------------------+------------------------------------------+
```

预处理的估计器

预处理的另一个工具是估计器。如果要执行的转换必须使用与输入列有关的数据或信息进行初始化 (通常传递输入列本身来派生)，那就有必要使用估计器。例如，如果要将我们列中的值缩放为以0为均值和具有单位方差的值，则需要处理整个数据，以便计算用于将数据归一化为以0为均值和具有单位方差的值。从效果上来看，估计器是可以根据你的特定输入数据配置的转换器。简单来说，你可以盲目应用转换 ("普通的 " 转换器类型)，或者根据数据执行转换(估计器类型) 。图 25-2 是使用对特定输入数据集拟合的估计器的一个简单示例，它生成随后转换器需要的输入数据，再通过转换器以追加新列 (转换后的数据)。

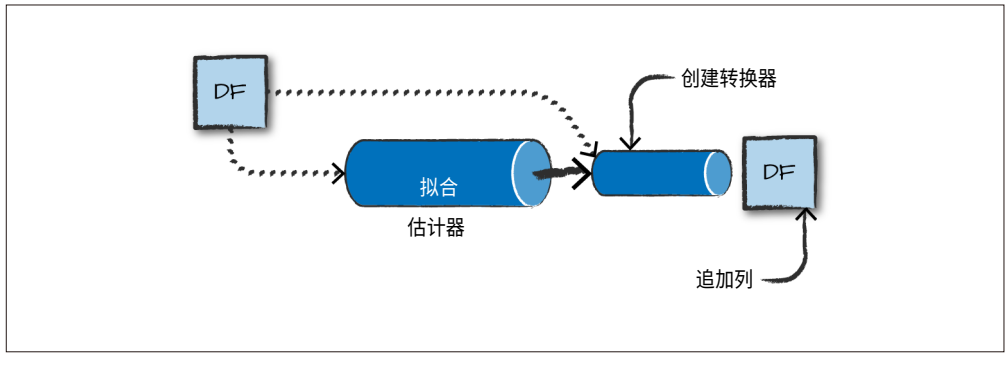


图25-2：Spark 估计器

此类估计器的一个示例是StandardScaler，根据该列中的值范围对输入列进行缩放，使其在每个维度中的数据都保持平均值为0方差为1的形式，为此，它必须首先处理一遍数据来创建转换器。下面是显示整个过程的示例代码片断以及输出：

```
// in Scala
import org.apache.spark.ml.feature.StandardScaler
val ss = new StandardScaler().setInputCol("features")
ss.fit(scaleDF).transform(scaleDF).show(false)
```

```
+---+--------------+------------------------------------------------------------+
|id |features      |stdScal_d66fbeac10ea__output                                |
+---+--------------+------------------------------------------------------------+
|0  |[1.0,0.1,-1.0]|[1.1952286093343936,0.02337622911060922,-0.5976143046671968]|
```

```
...
|1  |[3.0,10.1,3.0]|[3.5856858280031805,2.3609991401715313,1.7928429140015902]  |
+---+--------------+------------------------------------------------------------+
```

我们的示例会都使用估计器和转换器，并在本章后面部分中会涉及更多的估计器（并添加 Python 中的示例）。

转换器特性

所有的转换器都要求你至少指定`inputCol`和`outputCol`，分别代表输入和输出的列名，这可以通过`setInputCol`和`setOutputCol`来设置。虽然有一些默认值 (你可以在文档中找到这些)，但最好手动指定它们以使其清晰明了。除了输入和输出列，所有的转换器都有不同的可调参数(本章中每当我们提到一个参数时，你必须使用`set()`方法设置)。在 Python 中还有另一个方法，就是将这些值设置为对象构造函数的关键字参数，从下一章开始的示例中将排除它，以保持一致性。估计器要求你将转换器首先拟合到给定数据集，然后在结果对象上调用`transform`。

Spark MLlib 存储有关它在每个 DataFrame 中使用列的元数据作为列本身的特性。这样，它就可以正确地区分出，存储的Doubles类型列实际上是表示一系列离散的分类变量，而不是连续值。但是在打印模式或 DataFrame时However, metadata won't show up when you print the schema or the DataFrame。

高级转换器

如我们在上一章中看到的`RFormula`这种高级转换器，允许你在其中简明地指定多个转换。这些属于“高层”操作，使你避免一个一个地对数据操作或转换。一般情况下，你应该尝试使用最高级别的转换器，以尽量减少发生错误的风险，并帮助你专注于业务问题而不是实现细节。虽然这并非总能实现，但这是一个努力的方向。

RFormula

`RFormula`在具有 "常规" 格式的数据时是最易用的转换器。Spark从 R 语言借用这个转换器概念，可以简单地以声明式地方法为你的数据指定一组转换。有了这个转换器，值可以是数字或类别变量，你不需要从字符串中提取值或以任何方式操作它们。通过执行独热编码（one-hot encoding）的操作，`RFormula` 将自动处理代表类别的输入 (字符串形式)。简而言之，独热编码将一组值转换为一组二进制列，它指定一个数据点

是否具有每个特定值 (我们将在本章后面的更深入讨论独热编码)。使用RFormula，数值列将被强制转换为Double类型而不会被热编码。如果标签列是String类型，则首先会使用StringIndexer将其转换为Double类型。

将数值列转换为Double类型而不是独热编码形式的策略具有一些重要含义。如果是具有数值型的类别变量，则它们将仅仅转换为Double类型，这将隐式地指定次序。确保输入类型进行预期的转换非常重要，如果处理没有顺序关系的类别变量，则应将它们强制转换为String类型。你还可以手动索引列 (请参见本章后面的“使用类别特征”)。

RFormula允许你在声明式语言指定转换。一旦理解了语法，使用RFormula是很简单的。目前，RFormula支持有限的 R 运算符子集。在实际工作中，对于简单的转换来说它非常适合。基本运算符为：

~
: 目标（标签）和项（特征）的分隔符号。

+
: 合并项;，“+ 0”表示删除空格。

-
: 删除项，“ - 1”表示删除空格，和“+ 0”起相同作用(this means the y-intercept of the line that we will fit will be 0)。

:
: 交互（数值乘法，或类别二值化）。

.
: 除了目标列的全部列。

你可能猜到了，RFormula默认使用label和features的名字来标记它输出的标签和特征集合 (对于监督学习)。本章后面所述的模型在默认情况下也使用这些列名，这样就可以很容易地将转换成DataFrame的结果传递到模型中进行训练。如果你还没有理解，不用担心，当我们在后面章节中开始使用模型时，就会变得很清楚。

下面例子中使用RFormula，在这个例子中，我们希望使用所有可用的变量 (.)，然后在value1和color之间与value2和color之间指定交互，作为附加特征生成：

```
// in Scala
import org.apache.spark.ml.feature.RFormula
```

```
val supervised = new RFormula()
  .setFormula("lab ~ . + color:value1 + color:value2")
supervised.fit(simpleDF).transform(simpleDF).show()

# in Python
from pyspark.ml.feature import RFormula

supervised = RFormula(formula="lab ~ . + color:value1 + color:value2")
supervised.fit(simpleDF).transform(simpleDF).show()

+-----+----+------+------------------+--------------------+-----+
|color| lab|value1|            value2|            features|label|
+-----+----+------+------------------+--------------------+-----+
|green|good|     1|14.386294994851129|(10,[1,2,3,5,8],[...|  1.0|
| blue| bad|     8|14.386294994851129|(10,[2,3,6,9],[8....|  0.0|
...
|  red| bad|     1| 38.97187133755819|(10,[0,2,3,4,7],[...|  0.0|
|  red| bad|     2|14.386294994851129|(10,[0,2,3,4,7],[...|  0.0|
+-----+----+------+------------------+--------------------+-----+
```

SQL转换器

使用SQLTransformer允许你利用Spark的大量 SQL 相关操作库，就像 MLlib 转换一样。在 SQL 中使用的任何 SELECT 语句都是可行的转换，唯一需要更改的是不要使用表名，只需使用关键字*THIS*。如果要将某些 DataFrame 操作正式编纂为预处理步骤，或者在超参数调整期间尝试不同的 SQL 表达式，则可能需要使用SQLTransformer。还请注意，此转换的输出将作为列追加到输出DataFrame。

你可能希望使用SQLTransformer来表示最原始格式的数据上的所有操作，以便为不同的操作创建不同的转换器，这为你提供了构建和测试不同流水线的好处，全部仅仅通过简单地交换转换器就可以。下面是使用 SQLTransformer 的基本一个例子：

```
// in Scala
import org.apache.spark.ml.feature.SQLTransformer

val basicTransformation = new SQLTransformer()
  .setStatement("""
    SELECT sum(Quantity), count(*), CustomerID
    FROM __THIS__
    GROUP BY CustomerID
  """)

basicTransformation.transform(sales).show()

# in Python
from pyspark.ml.feature import SQLTransformer

basicTransformation = SQLTransformer()\
  .setStatement("""
```

```
    SELECT sum(Quantity), count(*), CustomerID
    FROM __THIS__
    GROUP BY CustomerID
  """)

basicTransformation.transform(sales).show()
```

这是输出的样例：

```
-------------+--------+----------+
|sum(Quantity)|count(1)|CustomerID|
+-------------+--------+----------+
|          119|      62|   14452.0|
...
|          138|      18|   15776.0|
+-------------+--------+----------+
```

对于这些转换的其他样例，请参考第Ⅱ部分。

VectorAssembler

VectorAssembler 是在几乎每条流水线中都使用的工具，它将所有特征组合成一个大的向量，然后将其传递到估计器。它通常在机器学习流水线的最后一步使用，并将多个Boolean，Double或者Vector类型的列作为输入。如果你要使用各种转换器执行一些操作，并需要收集所有这些结果，则这种方法就很有用。

下面代码段的输出明确说明了它是如何工作的：

```
// in Scala
import org.apache.spark.ml.feature.VectorAssembler
val va = new VectorAssembler().setInputCols(Array("int1", "int2", "int3"))
va.transform(fakeIntDF).show()

# in Python
from pyspark.ml.feature import VectorAssembler
va = VectorAssembler().setInputCols(["int1", "int2", "int3"])
va.transform(fakeIntDF).show()

+----+----+----+-----------------------------------------+
|int1|int2|int3|VectorAssembler_403ab93eacd5585ddd2d__output|
+----+----+----+-----------------------------------------+
|   1|   2|   3|                            [1.0,2.0,3.0]|
|   4|   5|   6|                            [4.0,5.0,6.0]|
|   7|   8|   9|                            [7.0,8.0,9.0]|
+----+----+----+-----------------------------------------+
```

处理连续型特征

连续型特征（Continuous Feature）是指从正无穷大到负无穷大的连续型数值。有两个常用的用于处理连续型特征的转换器。首先，你可以通过一个称为分桶的过程将连续特征转换为类别特征，也可以根据不同的要求来缩放和归一化特征。这些转换器将仅用于Double类型，因此请确保已将任何其他数值类型转换为Double类型：

```
// in Scala
val contDF = spark.range(20).selectExpr("cast(id as double)")

# in Python
contDF = spark.range(20).selectExpr("cast(id as double)")
```

分桶

实现分桶的最直接方法是使用Bucketizer，这将把一个给定的连续型特征分解到你指定的桶中，你需要指定如何根据Double类型值的数组或列表创建桶。这很有用，因为你可能希望简化数据集中的特征，或者简化它们的表示形式以便以后进行解释。例如，假设你有一个代表一个人体重的列，并且你希望根据这些信息预测一些值。在某些情况下，创建三个分别代表“超重”、“平均值”和“偏轻”的桶可能更简单。

要设置桶，就需要设置它的边界。例如，在contDF数据集上设置5.0，10.0，250.0的拆分值就会失败，因为没有涵盖所有可能的输入范围。指定桶时，传入拆分的值必须满足三要求：

- 拆分数组中的最小值必须小于 DataFrame 中的最小值。
- 拆分数组中的最大值必须大于 DataFrame 中的最大值。
- 你需要在拆分数组中指定至少三个值，这将创建两个桶。

我们通过split方法指定桶边界，但是没有真正的拆分，这样的Bucketizer让人困惑。

要覆盖所有可能的范围，使用Scala.Double.NegativeInfinity可能是另一个选择，用Scala.Double.PositiveInfinity代表最大值可以覆盖所有可能的范围。在Python中，我们用float("inf")，float("-inf")。

为了处理null或NaN值，我们必须指定handleInvalid参数，我们可以保留这些值（设置keep），报错（设置error或null）或跳过（设置skip）那些行。下面是分桶的例子：

```
// in Scala
import org.apache.spark.ml.feature.Bucketizer
val bucketBorders = Array(-1.0, 5.0, 10.0, 250.0, 600.0)
val bucketer = new Bucketizer().setSplits(bucketBorders).setInputCol("id")
bucketer.transform(contDF).show()

# in Python
from pyspark.ml.feature import Bucketizer
bucketBorders = [-1.0, 5.0, 10.0, 250.0, 600.0]
bucketer = Bucketizer().setSplits(bucketBorders).setInputCol("id")
bucketer.transform(contDF).show()

+----+----------------------------------------+
|  id|Bucketizer_4cb1be19f4179cc2545d__output|
+----+----------------------------------------+
| 0.0|                                     0.0|
...
|10.0|                                     2.0|
|11.0|                                     2.0|
...
+----+----------------------------------------+
```

除了基于硬编码值进行分桶外，另一个选项是基于数据的百分比进行拆分。这是通过QuantileDiscretizer完成的，它将值放入用户指定的存储桶中，并由近似分位数的值确定拆分。例如，数据中的90分位数表示90%的数据小于该值。通过使用setRelativeError 设置近似分位数计算的相对误差，可以控制桶的拆分方式。Spark这样做是通过允许你指定数据上的桶数，进而据此相应地拆分数据。下面是一个示例：

```
// in Scala
import org.apache.spark.ml.feature.QuantileDiscretizer
val bucketer = new QuantileDiscretizer().setNumBuckets(5).setInputCol("id")
val fittedBucketer = bucketer.fit(contDF)
fittedBucketer.transform(contDF).show()

# in Python
from pyspark.ml.feature import QuantileDiscretizer
bucketer = QuantileDiscretizer().setNumBuckets(5).setInputCol("id")
fittedBucketer = bucketer.fit(contDF)
fittedBucketer.transform(contDF).show()

+----+----------------------------------------+
|  id|quantileDiscretizer_cd87d1a1fb8e__output|
+----+----------------------------------------+
| 0.0|                                     0.0|
...
```

```
| 6.0|                                      1.0|
| 7.0|                                      2.0|
...
|14.0|                                      3.0|
|15.0|                                      4.0|
...
+----+-----------------------------------------+
```

高级分桶技术

刚才介绍的分桶技术是将数据分桶最常用的方法，但是当前Spark中也有许多其他的方法。从数据流的角度来看，所有这些过程都是相同的，把连续型数据划分到桶中进而得到类别变量，区别在于取得这些桶的算法。我们刚才只是看了容易解释和运行的例子，但MLlib也有更高级的技术例如局部敏感哈希（LSH）来完成近似分桶的任务。

缩放和归一化

我们已经明白如何利用分桶技术来创建连续型变量的分组。另一个常见的任务是缩放（Scaling）和归一化（Normalization）连续型数据，虽然并不总是必要的，但这样做通常会带来更好的效果。当你的数据集包含基于不同度量的多个列时，你就可能希望这样做。例如，假设我们有一个包含两列的DataFrame，一列是重量（以盎司为单位），一列是高度（以英尺为单位）。如果你没有缩放或归一化数据，算法将对高度变化不太敏感，因为英尺的高度值比盎司的重量值小得多，这就是一个应该缩放数据的情况。

归一化的例子可能涉及转换数据，使得每个点的值都表示其与该列的平均值的距离。用以前的例子，我们可能想知道某个体的身高与平均身高差了多少，许多算法假设输入数据是归一化后的。

正如你可能想象的那样，有许多算法可以应用到我们的数据来进行缩放或归一化，在这里列举这些算法没什么必要，因为许多其他文档和机器学习库都已涵盖这些算法。如果你对这个概念不太熟悉，请阅读上一章中提到的书籍推荐列表。一定要记住根本目标，那就是我们希望数据具有相同的度量，以便可以很容易地以合理的方式比较。在 MLlib 中，它总是基于Vector类型的列，MLlib 将查看给定列中的所有行 (Vector 类型)，然后将这些向量中的每个维度视为一种特殊的列，之后它将分别对每个维度应用缩放或归一化函数。

一个简单的例子可以是一列中的下列向量：

```
1, 2
3, 4
```

当我们应用我们的缩放 (而非归一化) 函数时，“3”和“1”将根据另外两个值进行调整，而“2”和“4”也将进行调整，这通常被称为逐分量比较（component-wise comparisons）。

StandardScaler

`StandardScaler`将一组特征值归一化成平均值为0而标准偏差为1的一组新值。`withStd`标志表示将数据缩放到单位标准差，而`withMean`标志(默认情况下为 false) 表示将使数据在缩放之前进行中心化（变量减去它的均值）。

稀疏向量中心化非常耗时，因为一般会将它们转化为稠密向量，所以谨慎中心化你的数据。

下面是一个使用`StandardScaler`的例子：

```
// in Scala
import org.apache.spark.ml.feature.StandardScaler
val sScaler = new StandardScaler().setInputCol("features")
sScaler.fit(scaleDF).transform(scaleDF).show()

# in Python
from pyspark.ml.feature import StandardScaler
sScaler = StandardScaler().setInputCol("features")
sScaler.fit(scaleDF).transform(scaleDF).show()
```

实际输出如下：

```
+---+--------------+------------------------------------------------------------+
|id |features      |StandardScaler_41aaa6044e7c3467adc3__output                 |
+---+--------------+------------------------------------------------------------+
|0  |[1.0,0.1,-1.0]|[1.1952286093343936,0.02337622911060922,-0.5976143046671968]|
...
|1  |[3.0,10.1,3.0]|[3.5856858280031805,2.3609991401715313,1.7928429140015902]  |
+---+--------------+------------------------------------------------------------+
```

MinMaxScaler

`MinMaxScaler`将向量中的值基于给定的最小值到最大值按比例缩放。如果指定的最小值为0且最大值为 1，则所有值都将介于0和1之间：

```
// in Scala
import org.apache.spark.ml.feature.MinMaxScaler
val minMax = new MinMaxScaler().setMin(5).setMax(10).setInputCol("features")
val fittedminMax = minMax.fit(scaleDF)
fittedminMax.transform(scaleDF).show()

# in Python
from pyspark.ml.feature import MinMaxScaler
minMax = MinMaxScaler().setMin(5).setMax(10).setInputCol("features")
fittedminMax = minMax.fit(scaleDF)
fittedminMax.transform(scaleDF).show()

+---+--------------+----------------------------------------+
| id|      features|MinMaxScaler_460cbafafbe6b9ab7c62__output|
+---+--------------+----------------------------------------+
|  0|[1.0,0.1,-1.0]|                            [5.0,5.0,5.0]|
...
|  1|[3.0,10.1,3.0]|                          [10.0,10.0,10.0]|
+---+--------------+----------------------------------------+
```

MaxAbsScaler

最大绝对值缩放将每个值除以该特征的最大绝对值来缩放数据。因此，所有的值最终都会在-1和1之间。该转换器不会平移数据也不会中心化数据：

```
// in Scala
import org.apache.spark.ml.feature.MaxAbsScaler
val maScaler = new MaxAbsScaler().setInputCol("features")
val fittedmaScaler = maScaler.fit(scaleDF)
fittedmaScaler.transform(scaleDF).show()

# in Python
from pyspark.ml.feature import MaxAbsScaler
maScaler = MaxAbsScaler().setInputCol("features")
fittedmaScaler = maScaler.fit(scaleDF)
fittedmaScaler.transform(scaleDF).show()

+---+--------------+------------------------------------------------------------+
|id |features      |MaxAbsScaler_402587e1d9b6f268b927__output                   |
+---+--------------+------------------------------------------------------------+
|0  |[1.0,0.1,-1.0]|[0.3333333333333333,0.009900990099009901,-0.3333333333333]|
...
|1  |[3.0,10.1,3.0]|[1.0,1.0,1.0]                                               |
+---+--------------+------------------------------------------------------------+
```

ElementwiseProduct

`ElementwiseProduct`允许用一个缩放向量对某向量中的每个值以不同的尺度进行缩放。例如，给定下面的向量和行“1，0.1，-1”，输出将是“10，1.5，-20”。当然缩放向量的维度必须与要进行缩放的向量的维度匹配：

```
// in Scala
import org.apache.spark.ml.feature.ElementwiseProduct
import org.apache.spark.ml.linalg.Vectors
val scaleUpVec = Vectors.dense(10.0, 15.0, 20.0)
val scalingUp = new ElementwiseProduct()
  .setScalingVec(scaleUpVec)
  .setInputCol("features")
scalingUp.transform(scaleDF).show()

# in Python
from pyspark.ml.feature import ElementwiseProduct
from pyspark.ml.linalg import Vectors
scaleUpVec = Vectors.dense(10.0, 15.0, 20.0)
scalingUp = ElementwiseProduct()\
  .setScalingVec(scaleUpVec)\
  .setInputCol("features")
scalingUp.transform(scaleDF).show()

+---+--------------+-------------------------------------------------+
| id|      features|ElementwiseProduct_42b29ea5a55903e9fea6__output|
+---+--------------+-------------------------------------------------+
|  0|[1.0,0.1,-1.0]|                                  [10.0,1.5,-20.0]|
...
|  1|[3.0,10.1,3.0]|                                 [30.0,151.5,60.0]|
+---+--------------+-------------------------------------------------+
```

Normalizer

Normalizer允许我们使用某个幂范数来缩放多维向量，Normalizer的作用范围是每一行，使每一个行向量的范数变换为一个单位范数。它通过参数“p”设置是几范数。例如，我们可以使用p = 1表示曼哈顿范数 (或曼哈顿距离)，p = 2表示欧几里德范数等。如果是一阶范数，将每一行的规整为一阶范数为1的向量，一阶范数即所有值绝对值之和。例如，下面例子中第一行[1.0，0.1，-1.0]的一阶范数（曼哈顿范数）是1.0+0.1+1.0=2.1，所以会得到[1.0/2.1，0.1/2.1，-1.0/2.1]=[0.47619047619047...]。

下面是使用Normalizer的例子。

```
// in Scala
import org.apache.spark.ml.feature.Normalizer
val manhattanDistance = new Normalizer().setP(1).setInputCol("features")
manhattanDistance.transform(scaleDF).show()

# in Python
from pyspark.ml.feature import Normalizer
manhattanDistance = Normalizer().setP(1).setInputCol("features")
manhattanDistance.transform(scaleDF).show()

+---+--------------+------------------------------+
| id|      features|normalizer_1bf2cd17ed33__output|
+---+--------------+------------------------------+
|  0|[1.0,0.1,-1.0]|          [0.47619047619047...|
```

```
|  1| [2.0,1.1,1.0]|           [0.48780487804878...|
|  0|[1.0,0.1,-1.0]|           [0.47619047619047...|
|  1| [2.0,1.1,1.0]|           [0.48780487804878...|
|  1|[3.0,10.1,3.0]|           [0.18633540372670...|
+---+--------------+-------------------------------+
```

使用类别特征

处理类型特征（Categorical Feature）最常见的任务是索引（Indexing）。索引将列中的一个类别变量转换为数值进而可以嵌入进机器学习算法，虽然这从概念上来说是简单的，但有一些细节是十分重要的，这样可以使Spark以一种稳定且可重复的方式来实现索引。

通常，我们建议在预处理时重新索引每个类别变量，这是为了保证一致性。因为从长远来看，这有助于模型的维护，因为编码可能会随着时间的推移而发生变化。

StringIndexer

实现索引的最简单方法是通过`StringIndexer`，它将字符串映射到不同的数字 id。Spark的`StringIndexer`还会创建附加到DataFrame的元数据，用来指定哪些输入字符串对应于哪些输出数值，这样我们以后就可以从索引的数值反向推出原始的类别输入：

```
// in Scala
import org.apache.spark.ml.feature.StringIndexer
val lblIndxr = new StringIndexer().setInputCol("lab").setOutputCol("labelInd")
val idxRes = lblIndxr.fit(simpleDF).transform(simpleDF)
idxRes.show()

# in Python
from pyspark.ml.feature import StringIndexer
lblIndxr = StringIndexer().setInputCol("lab").setOutputCol("labelInd")
idxRes = lblIndxr.fit(simpleDF).transform(simpleDF)
idxRes.show()

+-----+----+------+------------------+--------+
|color| lab|value1|            value2|labelInd|
+-----+----+------+------------------+--------+
|green|good|     1|14.386294994851129|     1.0|
...
|  red| bad|     2|14.386294994851129|     0.0|
+-----+----+------+------------------+--------+
```

我们还可以应用`StringIndexer`于非字符串的列。在这种情况下，这些非字符串列将在被索引之前转换为字符串：

```
// in Scala
val valIndexer = new StringIndexer()
  .setInputCol("value1")
  .setOutputCol("valueInd")

valIndexer.fit(simpleDF).transform(simpleDF).show()

# in Python
valIndexer = StringIndexer().setInputCol("value1").setOutputCol("valueInd")
valIndexer.fit(simpleDF).transform(simpleDF).show()

+-----+----+------+------------------+--------+
|color| lab|value1|            value2|valueInd|
+-----+----+------+------------------+--------+
|green|good|     1|14.386294994851129|     1.0|
...
|  red| bad|     2|14.386294994851129|     0.0|
+-----+----+------+------------------+--------+
```

请记住StringIndexer是一个必须符合输入数据的估计器，它必须看到所有输入之后进而确定输入到ID的映射关系。如果在输入"a""b"和"c"上训练 StringIndexer，然后再输入"d"的话，则默认情况下会返回错误。另一个可能是，如果输入值不是训练过程中看到的值，则跳过该值对应的行。对于上一个示例，输入"d" 值将导致该行完全跳过。我们可以在训练索引器或流水线之前或之后设置此选项，将来可能会添加更多选项，但在Spark 2.2 版本中，你只能跳过或抛出无效输入的错误。

```
valIndexer.setHandleInvalid("skip")
valIndexer.fit(simpleDF).setHandleInvalid("skip")
```

将索引值转回文本

当检查机器学习的结果时，你希望将索引的数值类型映射回原始的类别值。由于MLlib 分类模型使用索引值进行预测，因此这种转换帮助将模型预测值 (索引值) 转换回原始代表的类别，我们可以使用 IndexToString 来执行此处理。你会注意到，我们不必指定输出的是String类型，因为Spark的 MLlib 会为你维护此元数据信息，你可以选择指定转换哪列。

```
// in Scala
import org.apache.spark.ml.feature.IndexToString
val labelReverse = new IndexToString().setInputCol("labelInd")
labelReverse.transform(idxRes).show()

# in Python
from pyspark.ml.feature import IndexToString
labelReverse = IndexToString().setInputCol("labelInd")
labelReverse.transform(idxRes).show()
```

```
+-----+----+------+------------------+--------+-------------------------------+
|color| lab|value1|            value2|labelInd|IndexToString_415...2a0d__output|
+-----+----+------+------------------+--------+-------------------------------+
|green|good|     1|14.386294994851129|     1.0|                           good|
...
|  red| bad|     2|14.386294994851129|     0.0|                            bad|
+-----+----+------+------------------+--------+-------------------------------+
```

向量索引

VectorIndexer是处理在数据集向量内部具有类别变量的有用工具。此工具将自动查找输入向量内部的类别特征，并将它们转换为具有从0开始的类别索引。例如，在下面示例的 DataFrame 中，向量中的第一列是具有两个不同类别的类别变量，而其余变量是连续的。通过在VectorIndexer中设置maxCategories为2，我们指示Spark使用两个或更少的值代表向量中的某一列，并将其转换为一个类别变量。当你知道最多有多少个类别时，这会很有帮助，你可以指定它，并且它将自动对类别值进行相应的索引。Spark会根据这个参数改变数据值，所以是有大量重复值的连续变量，这些变量就会被意外地转换为类别变量：

```
// in Scala
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.linalg.Vectors
val idxIn = spark.createDataFrame(Seq(
  (Vectors.dense(1, 2, 3),1),
  (Vectors.dense(2, 5, 6),2),
  (Vectors.dense(1, 8, 9),3)
)).toDF("features", "label")
val indxr = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("idxed")
  .setMaxCategories(2)
indxr.fit(idxIn).transform(idxIn).show

# in Python
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.linalg import Vectors
idxIn = spark.createDataFrame([
  (Vectors.dense(1, 2, 3),1),
  (Vectors.dense(2, 5, 6),2),
  (Vectors.dense(1, 8, 9),3)
]).toDF("features", "label")
indxr = VectorIndexer()\
  .setInputCol("features")\
  .setOutputCol("idxed")\
  .setMaxCategories(2)
indxr.fit(idxIn).transform(idxIn).show()

+-------------+-----+-------------+
|     features|label|        idxed|
```

```
+-------------+-----+-------------+
|[1.0,2.0,3.0]|    1|[0.0,2.0,3.0]|
|[2.0,5.0,6.0]|    2|[1.0,5.0,6.0]|
|[1.0,8.0,9.0]|    3|[0.0,8.0,9.0]|
+-------------+-----+-------------+
```

独热编码

索引类别变量只是整个工作的一部分，独热编码是在类别变量索引之后执行的非常常见的数据转换。这是因为索引并不总是能够以（对下游处理模型来说）正确的方式表示类别变量。例如，当对 "color" 列进行索引时，你会注意到某些颜色的值 (或索引号) 比其他的高 (在我们的例子中，蓝色是 1，绿色是 2)。

这是不正确的，因为它给出的这种数学表达（即机器学习算法的输入）似乎暗示了绿色>蓝色，这在颜色类别的情况下是没有意义的。为了避免这种情况出现，可以使用 OneHotEncoder，它将不同的类别值转换为向量中的一个布尔值元素(1 或 0) 。当对颜色类别进行编码时，我们可以看到这些颜色类别不再被排序，从而使下游模型 (如线性模型) 更容易处理：

```
// in Scala
import org.apache.spark.ml.feature.{StringIndexer, OneHotEncoder}
val lblIndxr = new StringIndexer().setInputCol("color").setOutputCol("colorInd")
val colorLab = lblIndxr.fit(simpleDF).transform(simpleDF.select("color"))
val ohe = new OneHotEncoder().setInputCol("colorInd")
ohe.transform(colorLab).show()

# in Python
from pyspark.ml.feature import OneHotEncoder, StringIndexer
lblIndxr = StringIndexer().setInputCol("color").setOutputCol("colorInd")
colorLab = lblIndxr.fit(simpleDF).transform(simpleDF.select("color"))
ohe = OneHotEncoder().setInputCol("colorInd")
ohe.transform(colorLab).show()
```

```
+-----+--------+---------------------------------------+
|color|colorInd|OneHotEncoder_46b5ad1ef147bb355612__output|
+-----+--------+---------------------------------------+
|green|     1.0|                          (2,[1],[1.0])|
| blue|     2.0|                              (2,[],[])|
...
|  red|     0.0|                          (2,[0],[1.0])|
|  red|     0.0|                          (2,[0],[1.0])|
+-----+--------+---------------------------------------+
```

文本数据转换器

文本类型数据的输入很棘手，因为它往往需要大量的操作才能映射到一个机器学习模

型可用的格式。通常会看到两种文本：自由格式文本和字符串类别变量。因为我们已经讨论了类别变量，所以本节主要侧重于自由格式的文本。

文本分词

分词是将任意格式的文本转变成一个“符号”（“token”）列表或者一个单词列表的过程。要做到这一点最简单的方法是使用Tokenizer类，该转换器将由空格分开的包含若干单词的字符串转换成单词数组。例如对于我们的数据，我们可能想将Description属性转换成一个token列表。

```
// in Scala
import org.apache.spark.ml.feature.Tokenizer
val tkn = new Tokenizer().setInputCol("Description").setOutputCol("DescOut")
val tokenized = tkn.transform(sales.select("Description"))
tokenized.show(false)

# in Python
from pyspark.ml.feature import Tokenizer
tkn = Tokenizer().setInputCol("Description").setOutputCol("DescOut")
tokenized = tkn.transform(sales.select("Description"))
tokenized.show(20, False)
```

```
+---------------------------------+------------------------------------------+
|Description                      DescOut                                    |
+---------------------------------+------------------------------------------+
|RABBIT NIGHT LIGHT               |[rabbit, night, light]                    |
|DOUGHNUT LIP GLOSS               |[doughnut, lip, gloss]                    |
...
|AIRLINE BAG VINTAGE WORLD CHAMPION |[airline, bag, vintage, world, champion]  |
|AIRLINE BAG VINTAGE JET SET BROWN  |[airline, bag, vintage, jet, set, brown]  |
+---------------------------------+------------------------------------------+
```

我们不仅可以基于空格符创建Tokenizer，也可以用RegexTokenizer指定的正则表达式来分词。正则表达式的格式应符合 Java 正则表达式 (RegEx) 语法：

```
// in Scala
import org.apache.spark.ml.feature.RegexTokenizer
val rt = new RegexTokenizer()
  .setInputCol("Description")
  .setOutputCol("DescOut")
  .setPattern(" ") // 这是最简单的表达式
  .setToLowercase(true)
rt.transform(sales.select("Description")).show(false)

# in Python
from pyspark.ml.feature import RegexTokenizer
rt = RegexTokenizer()\
  .setInputCol("Description")\
  .setOutputCol("DescOut")\
  .setPattern(" ")\
```

```
  .setToLowercase(True)
rt.transform(sales.select("Description")).show(20, False)

+-----------------------------------+---------------------------------------+
|Description                        DescOut                                 |
+-----------------------------------+---------------------------------------+
|RABBIT NIGHT LIGHT                 |[rabbit, night, light]                 |
|DOUGHNUT LIP GLOSS                 |[doughnut, lip, gloss]                 |
...
|AIRLINE BAG VINTAGE WORLD CHAMPION |[airline, bag, vintage, world, champion]  |
|AIRLINE BAG VINTAGE JET SET BROWN  |[airline, bag, vintage, jet, set, brown]  |
+-----------------------------------+---------------------------------------+
```

使用RegexTokenizer的另一种方法是输出与提供模式匹配的值，而不是将其用作分隔符，将gaps参数设置为false就可以了。在下面例子中，用空格符作为模式将返回所有空格，这显然没什么用，但如果我们采用模式捕获单词，则会返回那些单词：

```
// in Scala
import org.apache.spark.ml.feature.RegexTokenizer
val rt = new RegexTokenizer()
  .setInputCol("Description")
  .setOutputCol("DescOut")
  .setPattern(" ")
  .setGaps(false)
  .setToLowercase(true)
rt.transform(sales.select("Description")).show(false)

# in Python
from pyspark.ml.feature import RegexTokenizer
rt = RegexTokenizer()\
  .setInputCol("Description")\
  .setOutputCol("DescOut")\
  .setPattern(" ")\
  .setGaps(False)\
  .setToLowercase(True)
rt.transform(sales.select("Description")).show(20, False)

+-----------------------------------+------------------+
|Description                        DescOut            |
+-----------------------------------+------------------+
|RABBIT NIGHT LIGHT                 |[ ,  ]            |
|DOUGHNUT LIP GLOSS                 |[ ,  ,  ]         |
...
|AIRLINE BAG VINTAGE WORLD CHAMPION |[ ,  ,  ,  ,  ]   |
|AIRLINE BAG VINTAGE JET SET BROWN  |[ ,  ,  ,  ,  ]   |
+-----------------------------------+------------------+
```

删除常用词

分词后的一个常见任务是过滤停用词（stop word），这些常用词在许多分析中没有什么意义，因此应被删除。英语中经常出现的停用词包括“the”“and”“but”。

Spark通过调用以下方法查看默认的停用词列表，如有需要可以不区分大小写。(在Spark 2.2版本中 所支持的停用词语言包括：“丹麦语”、“荷兰语”、“英语”、“芬兰语”、“法语”、“德语”、“匈牙利语”、“意大利语”、“挪威语”、“葡萄牙语”、“俄语”、“西班牙语”、“瑞典语”和“土耳其文”)：

```
// in Scala
import org.apache.spark.ml.feature.StopWordsRemover
val englishStopWords = StopWordsRemover.loadDefaultStopWords("english")
val stops = new StopWordsRemover()
  .setStopWords(englishStopWords)
  .setInputCol("DescOut")
stops.transform(tokenized).show()

# in Python
from pyspark.ml.feature import StopWordsRemover
englishStopWords = StopWordsRemover.loadDefaultStopWords("english")
stops = StopWordsRemover()\
  .setStopWords(englishStopWords)\
  .setInputCol("DescOut")
stops.transform(tokenized).show()
```

以下输出说明了其运行原理：

```
+--------------------+--------------------+-----------------------------------+
|         Description|             DescOut|StopWordsRemover_4ab18...6ed__output|
+--------------------+--------------------+-----------------------------------+
...
|SET OF 4 KNICK KN...|[set, of, 4, knic...|               [set, 4, knick, k...|
...
+--------------------+--------------------+-----------------------------------+
```

注意在输出列中单词of被删除了，因为它是一个及其常见的单词，与任何下游操作都不相关，只会给数据集增加噪声。

创建词组合

字符串分词和过滤停用词之后，会得到可作为特征的一个词集合，观察这些单词的组合（通常是关联词）很有趣。单词组合在技术上称为n-gram，即长度为n的单词序列，长度为1的n-gram称为unigram，长度为2的单词序列称为bigram，而长度为3的单词序列称为trigram(长度再大的就叫做four-gram，five-gram等)。创建n-gram要考虑单词的次序，将一个含有三个单词的句子转换为bigram将得到两个 bigram。创建n-gram的目的是为了更好地捕获句子结构和更多的信息，而不是逐个单词查看。让我们创建一些n-gram来说明这个概念。

"Big Data Processing Made Simple" 的 bigram包括：

- "Big Data"。
- "Data Processing"。
- "Processing Made"。
- "Made Simple"。

它的trigram是：

- "Big Data Processing"。
- "Data Processing Made"。
- "Processing Made Simple"。

使用n-gram，我们可以查看经常共同出现的单词序列，并将它们用作机器学习算法的输入，这比逐个单词处理可以创建更好的特征（以空格分隔）：

```
// in Scala
import org.apache.spark.ml.feature.NGram
val unigram = new NGram().setInputCol("DescOut").setN(1)
val bigram = new NGram().setInputCol("DescOut").setN(2)
unigram.transform(tokenized.select("DescOut")).show(false)
bigram.transform(tokenized.select("DescOut")).show(false)

# in Python
from pyspark.ml.feature import NGram
unigram = NGram().setInputCol("DescOut").setN(1)
bigram = NGram().setInputCol("DescOut").setN(2)
unigram.transform(tokenized.select("DescOut")).show(False)
bigram.transform(tokenized.select("DescOut")).show(False)
```

```
+----------------------------------------+-----------------------------------
DescOut                                  |ngram_104c4da6a01b__output       ...
+----------------------------------------+-----------------------------------
|[rabbit, night, light]                  |[rabbit, night, light]           ...
|[doughnut, lip, gloss]                  |[doughnut, lip, gloss]           ...
...
|[airline, bag, vintage, world, champion] |[airline, bag, vintage, world, cha...
|[airline, bag, vintage, jet, set, brown] |[airline, bag, vintage, jet, set, ...
+----------------------------------------+-----------------------------------
```

Bigrams的结果是:

```
+----------------------------------------+-----------------------------------
DescOut                                  |ngram_6e68fb3a642a__output       ...
+----------------------------------------+-----------------------------------
|[rabbit, night, light]                  |[rabbit night, night light]      ...
|[doughnut, lip, gloss]                  |[doughnut lip, lip gloss]        ...
...
```

```
|[airline, bag, vintage, world, champion]  |[airline bag, bag vintage, vintag...
|[airline, bag, vintage, jet, set, brown]  |[airline bag, bag vintage, vintag...
+-----------------------------------------+----------------------------------
```

单词转换成数值表示

一旦有了词特征，就可以开始对单词和单词组合进行计数，以便在我们的模型中使用。最简单的方法就是统计文档（我们的例子就是一行）中每个单词的二值计数，即存在该单词还是不存在该单词。这种简单方法可以规范化文档大小和出现次数，并获得可以用于文档分类的数值特征。此外，我们还可以使用CountVectorizer来对单词进行计数，或使用TF–IDF转换（接下来讨论）来对单词重新赋值权重：

一个CountVectorizer对分词之后数据进行操作，并执行以下两项操作：

1. 在fit（拟合）过程中，在全部文档中查找一个单词集合，然后计算在所有文档中的这些单词的出现次数。
2. 在转换过程中计算 DataFrame 列每行中给定单词的出现次数，并输出包含在该行中的单词的向量。

在概念上，此转换器将每行看作文档（document），把每个单词看作项（term），把所有项的集合看作词库（vocabulary）。这些都是可调的参数，我们可以设置最小项频率参数（minTF）来决定词库中是否包含的某项(这样可以有效地删除低频单词)，可以设置词库中的项需要至少出现在多少个文档中（用于从词库中删除低频单词的另一种方法），最后设置总的最大单词量（vocabSize）。最后默认情况下CountVectorizer 将输出文档中各项的计数，若要只返回word是否存在于文档中，可以使用setBinary (true)。下面是使用CountVectorizer的示例：

```
// in Scala
import org.apache.spark.ml.feature.CountVectorizer
val cv = new CountVectorizer()
  .setInputCol("DescOut")
  .setOutputCol("countVec")
  .setVocabSize(500)
  .setMinTF(1)
  .setMinDF(2)
val fittedCV = cv.fit(tokenized)
fittedCV.transform(tokenized).show(false)

# in Python
from pyspark.ml.feature import CountVectorizer
cv = CountVectorizer()\
  .setInputCol("DescOut")\
  .setOutputCol("countVec")\
  .setVocabSize(500)\
```

```
  .setMinTF(1)\
  .setMinDF(2)
fittedCV = cv.fit(tokenized)
fittedCV.transform(tokenized).show(False)
```

虽然输出看起来有点复杂，但实际上它只是一个稀疏向量，包含总的词汇量、词库中某单词的索引，以及该单词的计数：

```
+--------------------------------+-----------------------------------------+
DescOut                          |countVec                                 |
+--------------------------------+-----------------------------------------+
|[rabbit, night, light]          |(500,[150,185,212],[1.0,1.0,1.0])        |
|[doughnut, lip, gloss]          |(500,[462,463,492],[1.0,1.0,1.0])        |
...
|[airline, bag, vintage, world,...|(500,[2,6,328],[1.0,1.0,1.0])           |
|[airline, bag, vintage, jet, s...|(500,[0,2,6,328,405],[1.0,1.0,1.0,1.0,1.0])|
+--------------------------------+-----------------------------------------+
```

词频-逆文档频率

另一种将文本转换为数值表示的方法是使用词频-逆文档频率 (TF–IDF)。最简单的情况是，TF–IDF度量一个单词在每个文档中出现的频率，并根据该单词出现过的文档数进行加权，结果是在较少文档中出现的单词比在许多文档中出现的单词权重更大。在实践中，像“the”这样的词的权重非常低，因为它在所有文档中都经常出现，而一个更专业的词如“streaming”将出现在较少的文档中，从而有更高的权重值。在某种程度上，TF–IDF 帮助查找有类似主题的文档。让我们来看看一个例子，首先，我们将检查我们的数据中包含“red”一词的一些文档：

```
// in Scala
val tfIdfIn = tokenized
  .where("array_contains(DescOut, 'red')")
  .select("DescOut")
  .limit(10)
tfIdfIn.show(false)

# in Python
tfIdfIn = tokenized\
  .where("array_contains(DescOut, 'red')")\
  .select("DescOut")\
  .limit(10)
tfIdfIn.show(10, False)

+---------------------------------------+
DescOut                                 |
+---------------------------------------+
|[gingham, heart, , doorstop, red]      |
...
|[red, retrospot, oven, glove]          |
```

```
|[red, retrospot, plate]                |
+---------------------------------------+
```

我们可以在这些文档中看到一些重复出现的单词，但这些词至少提供了一个粗略的主题表示形式。现在，让我们输入到 TF–IDF中。为此，我们将对每个单词进行哈希运算并将其转换为数值表示形式，然后根据逆文档频率对词库中的每个单词进行加权。哈希是与CountVectorizer类似的过程，但是不可逆，也就是说从我们的输出索引，无法反过来获得输入词 (多个单词可能映射到相同的输出索引)：

```
// in Scala
import org.apache.spark.ml.feature.{HashingTF, IDF}
val tf = new HashingTF()
  .setInputCol("DescOut")
  .setOutputCol("TFOut")
  .setNumFeatures(10000)
val idf = new IDF()
  .setInputCol("TFOut")
  .setOutputCol("IDFOut")
  .setMinDocFreq(2)

# in Python
from pyspark.ml.feature import HashingTF, IDF
tf = HashingTF()\
  .setInputCol("DescOut")\
  .setOutputCol("TFOut")\
  .setNumFeatures(10000)
idf = IDF()\
  .setInputCol("TFOut")\
  .setOutputCol( “IDFOut")\
  .setMinDocFreq(2)

// in Scala
idf.fit(tf.transform(tfIdfIn)).transform(tf.transform(tfIdfIn)).show(false)

# in Python
idf.fit(tf.transform(tfIdfIn)).transform(tf.transform(tfIdfIn)).show(10, False)
```

因为输出太大我们无法在此展示。请注意“red”已经对应到了一个数值，并且每个文档中都要包含该值。还请注意，代表“red”的项的权重非常低，因为它出现在每个文档中。输出格式是稀疏的Vector，我们以如下形式输入到机器学习模型中：

```
(10000，[2591，4291，4456]，[1.0116009116784799，0.0，0.0])
```

此向量使用三个不同的值表示： 总的词汇量、文档中出现的每个单词的哈希值，以及这些单词的权重。这类似于CountVectorizer的输出。

Word2Vec

Word2Vec 是一个基于深度学习的工具，它用于计算一组单词的向量表示形式。我们的目标是在这个向量空间中相似的词彼此接近，这样我们就可以对单词本身进行归纳。该模型易于训练和使用，在一些自然语言处理应用中被证明是很有效的，包括实体识别、消除歧义、解析、标注和机器翻译。

Word2Vec 是基于语义来捕获单词之间的关系的。例如，如果v~king，v~queen，v~man和v~women代表这四个词的向量，那么我们通常会得到一个这样的表示：v ~ king　v ~ man + v ~ women ~=v ~ queen。为此，Word2Vec 使用一种名为 "skipgrams" 的技术将单词组成的句子转换为向量表示 (可选为特定大小)。它通过构建一个词汇表来做到这一点，然后对于每个句子，它删除一个token，并训练模型来预测 "n-gram" 特征中缺少的token。Word2Vec 在连续的、无格式约束的（以token形式表示的）文本中最有效。

下面是文档中的一个简单示例：

```
// in Scala
import org.apache.spark.ml.feature.Word2Vec
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.sql.Row
// 输入数据：每一行都是来自某句话或文档的词袋
val documentDF = spark.createDataFrame(Seq(
  "Hi I heard about Spark".split(" "),
  "I wish Java could use case classes".split(" "),
  "Logistic regression models are neat".split(" ")
).map(Tuple1.apply)).toDF("text")
// 学习一个从单词到向量的映射函数
val word2Vec = new Word2Vec()
  .setInputCol("text")
  .setOutputCol("result")
  .setVectorSize(3)
  .setMinCount(0)
val model = word2Vec.fit(documentDF)
val result = model.transform(documentDF)
result.collect().foreach { case Row(text: Seq[_], features: Vector) =>
  println(s"Text: [${text.mkString(", ")}] => \nVector: $features\n")
}

# in Python
from pyspark.ml.feature import Word2Vec
# 输入数据：每一行都是来自某句话或文档的词袋
documentDF = spark.createDataFrame([
    ("Hi I heard about Spark".split(" "), ),
    ("I wish Java could use case classes".split(" "), ),
    ("Logistic regression models are neat".split(" "), )
], ["text"])
# 学习一个从单词到向量的映射函数
```

```
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text",
  outputCol="result")
model = word2Vec.fit(documentDF)
result = model.transform(documentDF)
for row in result.collect():
    text, vector = row
print("Text: [%s] => \nVector: %s\n" % (", ".join(text), str(vector)))

Text: [Hi, I, heard, about, Spark] =>
Vector: [-0.008142343163490296,0.02051363289356232,0.03255096450448036]

Text: [I, wish, Java, could, use, case, classes] =>
Vector: [0.043090314205203734,0.035048123182994974,0.023512658663094044]

Text: [Logistic, regression, models, are, neat] =>
Vector: [0.038572299480438235,-0.03250147425569594,-0.01552378609776497]
```

Spark的 Word2Vec 实现包括多种可调参数，可在相关文档中找到。

特征操作

尽管几乎每个 ML 的转换器都在某种程度上操纵特征空间，但下面的算法和工具是自动扩展输入特征向量或将它们降维的常用方法。

PCA

主成分分析(PCA)是一种数学方法，用于找到我们的数据中最重要的成分(主成分) 。它通过创建新的特征集("成分")来更改数据的特征表示形式，每个新特征都是原始特征的组合。PCA 的作用在于它可以创建一组更小的、更有意义的特征集合输入到你的模型中，从而减少运行时间成本。

如果你有一个特别大的输入数据集，并且希望减少特征数，则需要使用 PCA。这种情况经常出现在文本分析中，其整个特征空间很大，许多特征都无关紧要。使用 PCA，我们可以找到最重要的特征组合，只把它们输入到机器学习模型中。PCA 使用参数指定要创建的输出特征的数量，这通常应该比输入向量的尺寸小得多。

选择正确的k值很难，这里无法给出特别有效的方法。有关详细信息，请查阅 ESL 和 ISL的相关章节。

下面来训练k=2的PCA：

```
// in Scala
import org.apache.spark.ml.feature.PCA
val pca = new PCA().setInputCol( “features").setK(2)
pca.fit(scaleDF).transform(scaleDF).show(false)

# in Python
from pyspark.ml.feature import PCA
pca = PCA().setInputCol( “features").setK(2)
pca.fit(scaleDF).transform(scaleDF).show(20, False)

+---+--------------+----------------------------------------+
|id |features      |pca_7c5c4aa7674e__output                |
+---+--------------+----------------------------------------+
|0  |[1.0,0.1,-1.0]|[0.0713719499248418,-0.4526654888147822]  |
...
|1  |[3.0,10.1,3.0]|[-10.872398139848944,0.030962697060150646]|
+---+--------------+----------------------------------------+
```

交互作用

在某些情况下，你可能对数据集中的特定变量有深入理解。例如，你可能知道两个变量之间的某种作用是下游估计器中包含的一个重要变量，特征转换器允许你手动创建两个变量之间的交互作用。交互作用只是将两个特征变量相乘，但是一个典型的线性模型不会对数据中每个特征对都这样做。此转换器当前仅在 Scala 中可用，但可以从任何语言使用 RFormula 调用。我们建议用户使用 RFormula 而不是通过Interation手动创建交互作用。

多项式扩展

多项式扩展（Polynomial Expansion）基于所有输入列生成交互变量。通过多项式扩展，可以指定各种程度的交互作用。例如，对于一个二阶多项式，Spark把特征向量中的每个值乘以所有其他值，然后将结果存储为特征。例如，如果我们有两个输入特征使用二阶多项式 (2 × 2)，我们将得到四个输出特征。如果我们有三个输入特征，我们将得到九个输出特征 (3 × 3)。我们使用三阶多项式将得到27输出特性(3 × 3 × 3)等。当你希望看到特定特征之间的交互作用，但不确定要考虑哪些特征之间的交互作用时，此转换非常有用。

多项式扩展会扩大特征空间，从而导致高计算成本和过拟合效果，所以请小心使用它，特别是在高维的情况下。

这里有一个二阶多项式的例子：

```
// in Scala
import org.apache.spark.ml.feature.PolynomialExpansion
val pe = new PolynomialExpansion().setInputCol( “features").setDegree(2)
pe.transform(scaleDF).show(false)

# in Python
from pyspark.ml.feature import PolynomialExpansion
pe = PolynomialExpansion().setInputCol( “features").setDegree(2)
pe.transform(scaleDF).show()

+---+--------------+------------------------------------------------------------+
|id |features      |poly_9b2e603812cb__output                                   |
+---+--------------+------------------------------------------------------------+
|0  |[1.0,0.1,-1.0]|[1.0,1.0,0.1,0.1,0.010000000000000002,-1.0,-1.0,-0.1,1.0]   |
...
|1  |[3.0,10.1,3.0]|[3.0,9.0,10.1,30.299999999999997,102.00999999999999,3.0... |
+---+--------------+------------------------------------------------------------+
```

特征选择

通常你会拥有大量可选的特征，并希望选择一个较小的子集用于训练。例如，许多特征之间可能是紧密相关的（所以只需选用一个），或者使用太多的特征可能导致过拟合。此过程称为特征选择。一旦你训练了一个模型，有许多方法来评估特征的重要性，但另一种选择是做一些粗略的预过滤。Spark提供一些简单的工具来做特征选择，例如ChiSqSelector。

ChiSqSelector

ChiSqSelector利用统计测试来确定与我们试图预测的标签无关的特征，并删除不相关的特征。它经常与类别数据一起使用，以减少你输入到模型中的特征数量，也可以降低文本数据的维数(以频率或计数的形式)。由于这种方法是基于卡方测试（Chi-Square test）的，有几种不同的方法可以选择“最佳”特征。这些方法是numTopFeatures，它基于p-value排序；percentile，它采用输入特征的比例 (而不是仅是top N 个特征)；fpr，它设置截断p-value。

我们将使用本章前面创建的CountVectorizer输出来演示这一点：

```
// in Scala
import org.apache.spark.ml.feature.{ChiSqSelector, Tokenizer}
val tkn = new Tokenizer().setInputCol( “Description").setOutputCol( “DescOut")
val tokenized = tkn
  .transform(sales.select( “Description", “CustomerId"))
  .where( “CustomerId IS NOT NULL")
```

```
val prechi = fittedCV.transform(tokenized)
val chisq = new ChiSqSelector()
  .setFeaturesCol("countVec")
  .setLabelCol("CustomerId")
  .setNumTopFeatures(2)
chisq.fit(prechi).transform(prechi)
  .drop("customerId", "Description", "DescOut").show()
```

```
# in Python
from pyspark.ml.feature import ChiSqSelector, Tokenizer
tkn = Tokenizer().setInputCol("Description").setOutputCol("DescOut")
tokenized = tkn\
  .transform(sales.select("Description", "CustomerId"))\
  .where("CustomerId IS NOT NULL")
prechi = fittedCV.transform(tokenized)\
  .where("CustomerId IS NOT NULL")
chisq = ChiSqSelector()\
  .setFeaturesCol("countVec")\
  .setLabelCol("CustomerId")\
  .setNumTopFeatures(2)
chisq.fit(prechi).transform(prechi)\
  .drop("customerId", "Description", "DescOut").show()
```

高级主题

有一些关于转换器和估计器的高级主题。此处我们讨论两个最常见的：持久化转换器和编写自定义转换器。

持久化转换器

一旦使用了估计器来配置转换器，将其写入磁盘并在需要时加载它 (如在另一个Spark会话中使用) 是很有用的，我们在上一章中接触持久化整条流水线时介绍了这一点。要单独保存转换器，我们使用内置的转换器 (或标准转换器) 上的write方法并指定位置：

```
// in Scala
val fittedPCA = pca.fit(scaleDF)
fittedPCA.write.overwrite().save("/tmp/fittedPCA")
```

```
# in Python
fittedPCA = pca.fit(scaleDF)
fittedPCA.write().overwrite().save("/tmp/fittedPCA")
```

之后就可以加载该转换器：

```
// in Scala
import org.apache.spark.ml.feature.PCAModel
val loadedPCA = PCAModel.load("/tmp/fittedPCA")
loadedPCA.transform(scaleDF).show()
```

```
# in Python
from pyspark.ml.feature import PCAModel
loadedPCA = PCAModel.load("/tmp/fittedPCA")
loadedPCA.transform(scaleDF).show()
```

编写自定义转换器

如果要将你自己的某些业务逻辑编码并放入 ML Pipeline中，或传递给超参数搜索等，则编写自定义转换器可能很有用处。通常，应尽量使用内置模块（例如，SQLTransformer），因为它们经过了优化可以高效运行。但有时我们没有合适的内置模块，就需要自定义一个转换器。下面创建一个简单的分词器（tokenizer）来演示：

```
import org.apache.spark.ml.UnaryTransformer
import org.apache.spark.ml.util.{DefaultParamsReadable, DefaultParamsWritable,
  Identifiable}
import org.apache.spark.sql.types.{ArrayType, StringType, DataType}
import org.apache.spark.ml.param.{IntParam, ParamValidators}

class MyTokenizer(override val uid: String)
  extends UnaryTransformer[String, Seq[String],
    MyTokenizer] with DefaultParamsWritable {

  def this() = this(Identifiable.randomUID( “myTokenizer"))

  val maxWords: IntParam = new IntParam(this, “maxWords",
     “The max number of words to return.",
  ParamValidators.gtEq(0))

  def setMaxWords(value: Int): this.type = set(maxWords, value)

  def getMaxWords: Integer = $(maxWords)

  override protected def createTransformFunc: String => Seq[String] = (
    inputString: String) => {
      inputString.split( “\\s").take($(maxWords))
  }

  override protected def validateInputType(inputType: DataType): Unit = {
    require(
      inputType == StringType, s"Bad input type: $inputType. Requires String.")
  }

  override protected def outputDataType: DataType = new ArrayType(StringType,
    true)
}

// 这使得可以通过该对象读回数据
object MyTokenizer extends DefaultParamsReadable[MyTokenizer]
val myT = new MyTokenizer().setInputCol( “someCol").setMaxWords(2)
myT.transform(Seq( “hello world. This text won't show.").toDF( “someCol")).show()
```

```
myT.transform(Seq("hello world. This text won't show.").toDF("someCol")).show()
```

当你必须根据实际输入数据自定义转换时，还可以编写自定义估计器。但是，这并不像编写独立转换器那样常用，因此本书不作介绍。自定义估计器的一个方法是查看我们之前看到的简单估计器，并修改其代码以适合你的应用场景，比如StandardScaler就不错。

小结

本章介绍了Spark中许多最常见的预处理转换。有几个特定领域的转换我们没有讲述(例如，离散余弦变换)，但你可以在文档中找到更多的信息。随着社区的壮大，在这方面Spark也在不断发展。

特征工程工具的另一个重要方面是一致性。在上一章中，我们讨论了流水线的概念，这是打包和训练端到端 ML 工作流必不可少的工具。在下一章中，我们将讨论你可能遇到的各种机器学习任务以及每个任务可用的算法。

第26章

分类

分类（Classification）是在给定一组输入特征的情况下预测标签、类别、类或离散变量的任务。与其他机器学习任务（例如回归）相比，关键区别在于分类任务的输出标签是一组可能值的有限集合（例如，三个类别）。

应用场景

第24章介绍了分类有许多应用场景，现实世界中还有几个可以应用分类的应用场景。

预测信贷风险

融资公司在向公司或个人提供贷款之前，可能会查看很多信息。最后决定是否提供贷款，是否提供贷款是一个二进制分类问题。

新闻分类

分类算法还可以被训练来预测新闻文章的主题(体育、政治、商业等)。

对人类行为分类

根据从传感器 (如手机的重力感应器或智能手表) 采集的数据，可以预测用户的行为活动，而输出将是一组有限类别的集合(如行走、睡眠、站立或运动)。

分类的类型

在继续介绍分类计数之前，先回顾一下几种不同的分类类型。

二元分类

最简单的分类类型是二元分类（Binary Classification），只会预测出两个标签。例如欺诈分析，某一交易归类为欺诈行为或是非欺诈行为；垃圾邮件分类，一个给定的电子邮件归类为垃圾邮件或是非垃圾邮件。

多分类

输出多于两个标签的分类任务是多分类（Multiclass Classification），预测结果是从多于两个的候选标签集合中选出来的一个标签。例如，Facebook从一张照片中预测是哪个人，气象学家预测天气 (雨天，晴天，多云等)。注意始终有一组有限的候选类别来预测，而不是无限制的类别候选，有时候多分类也称为多项式分类。

多标签分类

最后一种分类任务是多标签分类（Multilabel Classification），一个给定的输入可对应多个标签。例如，你可能希望根据这本书的内容预测书的主题，虽然这可以是一个多分类问题，但更适合于多标签分类，因为一本书可以涉及多个主题，划分到多个分类中。多标签分类的另一个例子是识别图像中的多个物体，在这个例子中，预测输出物体的个数不一定是固定的，不同图像可以检测出不同数量的物体。

MLlib中的分类模型

Spark有一些即用的二分类和多分类的模型。以下是Spark中可用的分类模型：

- Logistic回归（Logistic regression）。
- 决策树（Decision trees）。
- 随机森林（Random forests）。
- 梯度提升决策树（Gradient-boosted trees）。

Spark本身不支持多标签分类，为了训练出多标签分类模型，你必须训练每一个标签模型然后人工地将它们结合起来。当人工创建好模型后，可以使用评估多种模型效果的内置工具（在本章结尾讨论）。

本章将从下面几个方面介绍每种模型：

- 对模型及其思路的简单解释。

- 模型超参数（采用不同的超参数初始化模型）。
- 训练参数（影响模型训练过程的参数）。
- 预测参数（影响预测效果的参数）。

正如我们在第24章中讨论的，可以在ParamGrid中设置超参数和训练参数。

模型的可扩展性

选择模型时，模型的可扩展性是一个重要的考虑因素。总的来说，Spark支持训练大规模机器学习模型（注意，这些模型的训练都是大规模的，基于单节点的训练还有其他很多优秀的工具）。表26-1简单地列出了各个模型的可扩展性能，据此你可以找到适合你特定任务的最佳模型（如果可扩展性是你的核心考虑因素的话）。实际的可扩展性将取决于你的配置，机器性能和其他细节，但该表可以帮助你有做个大概可行的选择。

表26-1：模型可伸缩性参考

模型	特征数量	训练样例数	输出类别
逻辑回归	100万~1000万	无限	特征×类别数<1000万
决策树	1000	无限	特征×类别数<10000
随机森林	10000	无限	特征×类别数<100000
梯度提升树	1000	无限	特征×类别数<10000

可以看到，几乎所有这些模型都可以适应规模庞大的输入数据集，而且Spark开发团队也在进一步提升可扩展性能。无限制训练实例的原因是因为这些训练采用了随机梯度下降和L-BFGS等方法进行训练，这些方法专门针对大数据集进行了优化，并避免了可能存在的限制训练实例数量的若干问题。

下面开始先加载数据再来详细看看这些分类模型：

```
// in Scala
val bInput = spark.read.format( “parquet").load( “/data/binary-classification")
  .selectExpr( “features", “cast(label as double) as label")

# in Python
bInput = spark.read.format( “parquet").load( “/data/binary-classification")\
  .selectExpr( “features", “cast(label as double) as label")
```

像其他高级分析章节一样，本章不会介绍每个模型的数学基础。有关分类的介绍，请参阅第4章中的ISL和ESL。

逻辑回归

逻辑回归是最常见的分类方法之一。它是一种线性模型，为输入的每个特征赋以权重之后将它们组合在一起，从而获得该输入属于特定类的概率。这些权重很有用，因为它们很好地表示了特征重要性。如果某特征的权重很大，则说明该特征的变化对结果有显著影响 (假定执行了标准化)，而较小的权重意味着该特征不是那么重要。

请参见ISL 4.3和ESL 4.4获得更多相关信息。

模型超参数

模型的超参数决定了模型本身的基本结构配置。逻辑回归包含以下超参数：

`family`
: 可以设置为“multinomial”（两个或更多个不同的标签，对应多分类）或“binary”（仅两个不同的标签，对应二分类）。

`elasticNetParam`
: 从0到1的浮点值。该参数依照弹性网络正则化的方法将L1正则化和L2正则化混合(即两者的线性组合)。对L1或L2的选择取决于你的特定用例，其指导原则如下：L1正则化 (值1)将在模型中产生稀疏性，因为(对输出的影响不大的)某些特征权重将变为零 ，因此它可以作为一个简单的特征选择方法。L2正则化(值0)不会造成稀疏，因为特定特征的相应权重只会趋于零而不会等于零。弹性网络给出了两者之间最好的结合-我们可以选择一个介于0和1之间的值，以指定L1和L2正则化的混合。在大多数情况下，你应该通过多次测试来调整这个值。

`fitIntercept`
: 可以是true或false。此超参数决定是否适应截距。通常情况下，如果我们没有对训练数据执行标准化，则需要添加截距。

`regParam`
: 大于等于0的值。确定在目标函数中正则化项的权重，它的选择和数据集的噪音情况和数据维度有关，最好尝试多个值 (如0、0.01、0.1、1)。

`standardization`

可以是true或false。设置它决定在将输入数据传递到模型之前是否要对其进行标准化，可以在第25章了解更多信息。

训练参数

训练参数用于指定我们如何执行训练，下面是逻辑回归的训练参数。

`maxIter`

迭代次数。更改此参数可能不会对结果造成很大的影响，所以它不应该是你要调整的首个参数。默认值是100。

`tol`

此值指定一个用于停止迭代的阈值，达到该阈值说明模型已经优化的足够好了。指定该参数后，算法可能在达到maxIter 指定的次数之前停止迭代，默认值为1.0E-6。这也不应该是你要调整的第一个参数。

`weightCol`

权重列的名称，用于赋予某些行更大的权重。如果有衡量每个训练样本重要性的方法，并对这些样本赋予了不同的训练权重值，这就是个很有用的工具。例如，在10000个样例中你知道哪些样本的标签比其他样本的标签更精确，就可以赋予那些更有用的样本以更大的权值。

预测参数

这些参数指定模型如何实际进行预测而又不影响训练。以下是逻辑回归的预测参数：

`threshold`

一个0~1的Double值。此参数是预测时的概率阈值，你可以根据需要调整此参数以平衡误报（false positive）和漏报（false negative）。例如，如果误报的成本高昂，你可能希望使该预测阈值非常高。

`thresholds`

该参数允许你在进行多分类的时候指定每个类的阈值数组，它和之前的threshold类似。

示例

下面是一个使用 LogisticRegression （逻辑回归）模型的简单示例。我们没有指定任

何参数，而是使用默认值，并且数据符合正确的列命名。在实际中，你一般也不需要设置许多参数：

```
// in Scala
import org.apache.spark.ml.classification.LogisticRegression
val lr = new LogisticRegression()
println(lr.explainParams()) // 查看所有的参数
val lrModel = lr.fit(bInput)

# in Python
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression()
print lr.explainParams() #查看所有的参数
lrModel = lr.fit(bInput)
```

一旦模型被训练好了，你就可以观察系数和截距项来获取有关模型的信息。系数对应于各特征的权重 (每个特征权重乘以各特征来计算预测值)，而截距项是斜线截距的值（如果我们在指定模型时选择了适当的截距参数fitIntercept）。查看系数有助于检查构建的模型，也有助于理解各特征如何影响预测：

```
// in Scala
println(lrModel.coefficients)
println(lrModel.intercept)
# in Python

print lrModel.coefficients
print lrModel.intercept
```

对于多项式分类模型（上面是针对二分类模型的），`lrModel.coefficientMatrix`和`lrModel.interceptVector`可以用来得到系数和截距值，它们会返回Matrix类型和Vector类型的值代表每个类别的相应值。

模型摘要

逻辑回归提供了一个模型摘要（Model Summary），给出了最终训练模型的相关信息，它类似于我们在许多 R 语言机器学习包中看到的摘要。模型摘要目前仅可用于二分类逻辑回归问题，但将来可能会添加多分类逻辑回归的摘要。利用二分类摘要，可以得到关于模型本身的各种信息，包括 ROC 曲线下的面积、f值、准确率、召回率和ROC曲线。请注意，对于曲线下面积，不考虑实例权重，因此如果你想了解考虑权重的情况，则必须手动处理，未来版本的Spark中可能会提供此功能。你可以使用以下API 查看摘要：

```
// in Scala
import org.apache.spark.ml.classification.BinaryLogisticRegressionSummary
val summary = lrModel.summary
```

```
val bSummary = summary.asInstanceOf[BinaryLogisticRegressionSummary]
println(bSummary.areaUnderROC)
bSummary.roc.show()
bSummary.pr.show()

# in Python
summary = lrModel.summary
print summary.areaUnderROC
summary.roc.show()
summary.pr.show()
```

模型到达最终结果状态的速度会显示在目标历史（objective history）中。我们可以查看模型摘要的目标历史记录来获得：

```
Summary.objectiveHistory
```

它是一个Double类型的数组，包含了每次训练迭代时我们的模型到底表现如何。此信息有助于帮助用户了解是否已经进行了足够次数的迭代训练或是否需要调整其他参数。

决策树

决策树（Decision Tree）是一种更友好和更易于理解的分类方法，因为它类似于人类经常使用的简单决策模型。例如，如果你必须预测某人是否会买冰激凌，一个很好的特征可能是这个人是否喜欢冰淇淋。在伪代码中，如果person.likes(“ice_cream”)返回true，他们会买冰淇淋，否则，他们不会买冰淇淋。决策树模型即是基于所有输入构建一棵树形结构，在预测时通过判断各种可能的分支来给出预测结果。这使它被常常做为一个最先被试用的模型，因为它易于推理，易于检查，并且对数据的结构只需要很少的假设。简而言之，它并不是试图训练系数来拟合函数而是简单地创造了一棵树来进行预测。该模型还支持多分类，并将输出各种预测结果和它们的概率。

虽然决策树模型简单，但它有一些缺点，它可能会非常快地就会出现过拟合的情况。意思是，在无约束的条件下决策树会基于每个训练样例从根节点开始创建一条判断路径，这意味着它会对模型的训练集中的所有信息进行编码。 这很糟糕，因为这样的话模型就不能泛化到新的数据（可能会得到很差的预测准确度）。但是也有一些方法通过限制分支结构（例如限制高度）来避免过拟合的问题，可以从ISL 8.1 和 ESL 9.2 了解更多信息。

模型超参数

有许多不同的方法来配置和训练决策树，下面是Spark实现支持的超参数：

`maxDepth`

因为我们正在训练生成一棵树结构，所以指定最大深度以避免过拟合数据集是很有帮助的。默认值为5。

`maxBins`

在决策树中，连续特征被转换为类别特征，maxBins确定应基于连续特征创建多少个槽（bin，相当于类别特征个数），更多的槽提供更细的粒度级别。该值必须大于或等于 2，并且也需要大于或等于数据集中任何类别特征中的类别数。默认值为 32。

`impurity`

建立一个 "树"时，在模型应该分支时需要进行配置。不纯度（impurity）表示是否应该在某叶子结点拆分的度量(信息增益)。该参数可以设置为 "entropy" 或 "gini" (默认值)，这是两个常用的不纯度度量。

`minInfoGain`

此参数确定可用于分割的最小信息增益。较大的值可以防止过拟合，这通常需要通过测试决策树模型不同变体来确定。默认值是0。

`minInstancePerNode`

此参数确定需要在一个节点结束训练的实例最小数目。可以把这看成是控制最大深度的另一种方式，可以通过限制深度来防止过拟合，或者可以指定在一个叶子节点上的最少训练样本来防止过拟合。如果它不满足，我们会 "修剪" （prune）树，直到满足要求。较大的值可以防止过拟合。默认值为 1，但可以是大于1的任何数值。

训练参数

还有一些针对如何执行训练的训练参数可以配置，下面是决策树的训练参数：

`checkpointInterval`

检查点（checkpointing）是一种在训练过程中保存模型的方法，此方法可以保证当集群节点因某种原因崩溃而不影响整个训练过程。将该值设置为10，表示模型每10次迭代都会保存检查点，将此设置为-1以关闭检查点。需要将此参数与`checkpointDir`(检查点的目录) 和`useNodeIdCache = true`一起设置。有关检查点的详细信息，请参阅Spark文档。

预测参数

决策树只有一个预测参数：thresholds。请参阅本章前面的“逻辑回归”中的阈值解释。

下面是一个使用决策树分类器的小示例：

```
// in Scala
import org.apache.spark.ml.classification.DecisionTreeClassifier
val dt = new DecisionTreeClassifier()
println(dt.explainParams())
val dtModel = dt.fit(bInput)

# in Python
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier()
print dt.explainParams()
dtModel = dt.fit(bInput)
```

随机森林和梯度提升树

这些方法是决策树的扩展，其实不必在所有数据上训练一棵树，而是在不同的数据子集上训练多个树。这样做的思路是，多棵决策树成为某一特定领域的“专家”，而其他决策树则可以成为其他特定领域的专家。通过结合这些不同的专家，就可以达到超过任何个人的“群众的智慧”。此外，这些方法有助于防止过度拟合。

随机森林（Random Forest）和梯度提升树（Gradient-Boosted Trees）是组合决策树的两种截然不同的方法。在随机森林中，我们只训练大量的树，然后平均他们的结果做出预测。利用梯度提升树，每棵树进行加权预测 (因为一些树对某些类别的预测能力比其他树更强)。 它们的参数大致相同，我们将在下面说明。 但是它也有一个局限性，就是梯度提升树目前仅支持二分类任务。

有几种常用的工具用于学习基于树的模型。例如，XGBoost库提供了可用于在Spark上运行的集成包。

关于这些增强树模型的更多信息，请参考ISL 8.2和ESL 10.1。

模型的超参数

随机森林和梯度提升树具有与前面介绍的决策树模型相同的模型超参数。此外，它们各自还添加了几个自己的超参数。

仅适合随机森林

`numTrees`

用于训练的树的总数。

`featureSubsetStrategy`

此参数确定拆分时应考虑多少特征，它可以是“auto”“all”“sqrt”“log2”，或数字“n”。当输入为“n”时，模型将在训练过程中使用n个特征数，当n在范围1至特征数量之间时，模型将在训练期间使用 n个特征。这里没有一个适合所有情况的解决方案，所以在你的流水线任务中需要尝试不同的值。

仅适合梯度提升树（GBT）

`lossType`

这是梯度提升树在训练过程中优化的损失函数。目前只支持logistic loss损失。

`maxIter`

迭代次数。改变这个值可能不会改变你的结果太多，所以它不应该是你首要调整的参数。默认值是100。

`stepSize`

代表算法的学习速度。较大的步长（step size）意味着在两次迭代训练之间较大的变化。该参数可以帮助优化过程，是在训练中应该多次测试的参数。默认值为0.1，可以是从0到1的任何数值。

训练参数

这些模型只有一个训练参数`checkpointInterval`。有关检查点的详细信息，请参阅本章前面的“决策树”中的说明。

预测参数

这些模型与决策树具有相同的预测参数。有关的详细信息，请参考决策树模型下的预测参数。

下面是使用这些分类方法的简短代码示例：

```
// in Scala
import org.apache.spark.ml.classification.RandomForestClassifier
val rfClassifier = new RandomForestClassifier()
println(rfClassifier.explainParams())
val trainedModel = rfClassifier.fit(bInput)

// in Scala
import org.apache.spark.ml.classification.GBTClassifier
val gbtClassifier = new GBTClassifier()
println(gbtClassifier.explainParams())
val trainedModel = gbtClassifier.fit(bInput)

# in Python
from pyspark.ml.classification import RandomForestClassifier
rfClassifier = RandomForestClassifier()
print rfClassifier.explainParams()
trainedModel = rfClassifier.fit(bInput)

# in Python
from pyspark.ml.classification import GBTClassifier
gbtClassifier = GBTClassifier()
print gbtClassifier.explainParams()
trainedModel = gbtClassifier.fit(bInput)
```

朴素贝叶斯

朴素贝叶斯（Naive Bayes）分类是基于贝叶斯定理的分类方法。该模型背后的核心假设是，数据的所有特征都是相互独立的。当然，严格的独立性是有点不现实，但即使不符合这条假设，仍然可以生成有用的模型。 朴素贝叶斯分类通常用于文本或文档分类，尽管它也可以用作更通用的分类任务。有两种不同的模型类型：多元贝努利模型（multivariate Bernoulli model），其中指示器变量代表文档中的一个单词是否存在;多项式模型（ multinomial model），其中使用所有单词计数。

当谈到朴素贝叶斯一个重要的注意事项是，所有的输入特征值必须为非负数。

ISL 4.4和ESL 6.6中有关于这些模型更多的背景信息。

模型超参数

这些配置是我们为确定模型的基本结构而指定的：

`modelType`
: “bernoulli”或“multinomial”。有关此选项的详细信息，请参阅上一节。

`weightCol`

允许对不同的数据点赋值不同的权值。有关此超参数的说明，请参阅本章前面的“训练参数”。

训练参数

用来指定我们如何执行训练过程：

`smoothing`

它指定使用加法平滑（additive smoothing）时的正则化量，该设置有助于平滑分类数据，并通过改变某些类的预期概率来避免过拟合问题。默认值是1。

预测参数

像所有其他模型一样，朴素贝叶斯也使用相同的预测参数`thresholds`。有关阈值的说明，请参阅本章之前的解释。

下面是一个使用朴素贝叶斯分类的示例。

```
// in Scala
import org.apache.spark.ml.classification.NaiveBayes
val nb = new NaiveBayes()
println(nb.explainParams())
val trainedModel = nb.fit(bInput.where("label != 0"))

# in Python
from pyspark.ml.classification import NaiveBayes
nb = NaiveBayes()
print nb.explainParams()
trainedModel = nb.fit(bInput.where("label != 0"))
```

请注意，在本示例数据集中具有负值的特征。在该数据集中，具有负特征的行对应于带有标签 "0" 的行。因此，我们只是要过滤它们 (通过标签)，而不是进一步处理它们来演示朴素贝叶斯的API。

分类评估器和自动化模型校正

正如我们在第24章中看到的那样，评估器允许我们指定模型的终止标准。当只有一个评估器时，它的作用并不太大。但是当我们在流水线中使用它时，我们可以自动地尝试模型和转换器的各种参数，尝试所有参数的组合，以查看哪些性能最好。评

估器在这条流水线和参数网格（Parameter Grid）上下文中最有用。对于分类，有两个评估器对应两列：一个模型预测的标签和一个真正的标签。对于二分类，我们使用 BinaryClassificationEvaluator，它支持优化两个不同的指标“areaUnderROC”和“areaUnderPR”。对于多分类，需要使用 MulticlassClassificationEvaluator，它支持优化“f1”“weightedPrecision”“weightedRecall”和“accuracy”。

要使用评估器，需要建立流水线，指定我们想测试的参数，然后运行一下看看结果。第24章 包含有代码样例。

详细的评价指标

MLlib 还包含一些工具，让你同时评估多个分类指标。不幸的是，这些度量类还没有从底层 RDD 框架中移植到Spark基于DataFrame的机器学习包中。因此，在编写本书时，你仍然需要创建一个 RDD 来使用它们。将来此功能很可能会移植到 DataFrame 上，那时候下面介绍的就不再是查看度量标准的最佳方法 (尽管你仍然可以使用这些 api)。

我们可以使用三种不同的分类指标：

- 二分类指标。
- 多分类指标。
- 多标签分类指标。

所有这些评测标准都遵循相同的近似样式，将生成的输出值与真实值进行比较，模型将计算所有相关的度量标准，然后可以查询每个指标的值：

```
// in Scala
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
val out = model.transform(bInput)
  .select("prediction", "label")
  .rdd.map(x => (x(0).asInstanceOf[Double], x(1).asInstanceOf[Double]))
val metrics = new BinaryClassificationMetrics(out)

# in Python
from pyspark.mllib.evaluation import BinaryClassificationMetrics
out = model.transform(bInput)\
  .select("prediction", "label")\
  .rdd.map(lambda x: (float(x[0]), float(x[1])))
metrics = BinaryClassificationMetrics(out)
```

这样做之后，可以使用与逻辑回归类似的API，在此度量指标对象上查看典型的度量指标：

```
// in Scala
metrics.areaUnderPR
metrics.areaUnderROC
println("Receiver Operating Characteristic")
metrics.roc.toDF().show()

# in Python
print metrics.areaUnderPR
print metrics.areaUnderROC
print "Receiver Operating Characteristic"
metrics.roc.toDF().show()
```

One-vs-Rest分类

有一些MLlib模型不支持多分类，在这些情况下，用户可以利用一个one-vs-rest分类器来执行多分类，它是在给定一个二分类器的情况下，执行多分类任务。这背后的思想是，对于每一个你希望预测的类，one-vs-rest分类器将把该分类转化为一个二分类问题，将目标类做为一类，把其余的其他类别做为另外一类。这样，分类预测就变成了一个二分类问题 (是此类或不是此类)。

One-vs-rest是作为估计器实现的。对于基分类器，它创建k个分类器实例，为k个类别中的每个分类创建一个对应的二分类问题，训练类i的分类器预测标签是否为i，将类i与所有其他类区分开来。

预测是通过对每个二分类器进行评估来完成的，将最有可能的分类器的索引输出为标签。

Spark文档中有一个使用one-vs-rest分类的很好的例子。

多层感知器

多层感知器（Multilayer Perceptron）是基于（可配置层数和层宽度的）神经网络的分类器。我们将在第31章讨论。

小结

在本章中，我们介绍了Spark提供的用于分类的工具，根据其特征可以预测每个数据点的分类标签。在下一章中，我们将介绍回归，它的输出是连续的，而不是离散的。

第27章

回归

回归（Regression）是分类的逻辑延伸。回归不是仅仅根据一组特质值预测单个离散值，而是预测实数值（表示为连续变量）。

回归可能比分类更难，因为从数学角度来看，有无限数量的可能输出值。此外，我们的目标是优化预测值和真实值之间的一些误差度量，而不是准确率。除此之外，回归和分类是很类似的。因此，我们将看到许多分类和回归有一些相同的基础概念。

应用场景

下面是一系列使用回归的应用场景，你也可以考虑自己熟悉领域中潜在的回归问题：

预测电影收视率

如果有电影和电影观众的信息，比如有多少人看过预告片或者在社交媒体上分享过，可以预测有多少人会真去看这部电影。

预测公司营收

鉴于目前的增长轨迹、市场和季节性，可以预测一个公司在未来将获得多少收入。

预测作物产量

如果有关于作物生长区域的信息以及全年的天气情况信息，可以预测某一特定区域位置的农作物总产量。

MLlib中的回归模型

MLlib 中有几个基本的回归模型，其中一些模型在第26章中讨论分类问题时介绍过，其他的模型则只是和回归问题有关。下面是当前Spark 2.2支持的回归模型，将来的版本会有所扩充：

- 线性回归（Linear regression）。
- 广义线性回归（Generalized linear regression）。
- 保序回归（Isotonic regression）。
- 决策树（Decision trees）。
- 随机森林（Random forest）。
- 梯度提升树（Gradient-boosted trees）。
- 生存分析（Survival regression）。

本章将通过以下几个方面来介绍每个特定模型的基础知识：

- 简单解释模型和算法背后的思想。
- 模型超参数(初始化模型的不同方法)。
- 训练参数 (影响模型训练方式的参数)。
- 预测参数 (影响如何进行预测的参数)。

你可以使用`ParamGrid`对超参数和训练参数进行搜索，就像第24章中说的那样。

模型的可扩展性

MLlib 中的回归模型全部都适合处理大型数据集。表27-1是一个简单的模型扩展性参考表，它将帮助你为特定任务选择最佳模型 (如果可扩展性是你要考虑的核心因素)。这些将取决于你的配置、计算机性能和其他因素。

表27-1：回归的可扩展性参考表

模型	特征数量	训练样例
线性回归	100万~10000万	无限
广义线性回归	4096	无限
保序回归	N/A	百万级别

表27-1：回归的可扩展性参考表（续）

模型	特征数量	训练样例
决策树	1000	无限
随机森林	10000	无限
梯度提升树	1000	无限
存活分析	100万~1000万	无限

与高级分析部分其他章节一样，本章不会介绍每个模型的数学基础。 有关回归的介绍，请参阅ISL和ESL的第3章。

首先读入一些数据，它将在本章后面部分也用到：

```
// in Scala
val df = spark.read.load("/data/regression")

# in Python
df = spark.read.load("/data/regression")
```

线性回归

线性回归假定输入特征的线性组合 (每个特征乘以权重的总和) 将得到（有一定高斯误差的）输出结果。这个 (高斯误差) 线性的假设并不总是正确的，但它确实是一个简单的可解释模型，而且一般不会产生过拟合。像逻辑回归一样，Spark实现了 ElasticNet 正则化，允许你结合使用L1 和 L2 正则化。

有关更多信息，请参见 ISL 3.2 和 ESL 3.2。

模型超参数

线性回归与逻辑回归具有相同的模型超参数。有关详细信息，请参阅第26 章。

训练参数

线性回归也与逻辑回归具有相同的训练参数。有关此主题的详细内容，请参阅第26章。

示例

下面是对读入数据集使用线性回归的简短示例：

```
// in Scala
import org.apache.spark.ml.regression.LinearRegression
val lr = new LinearRegression().setMaxIter(10).setRegParam(0.3)\
  .setElasticNetParam(0.8)
println(lr.explainParams())
val lrModel = lr.fit(df)

# in Python
from pyspark.ml.regression import LinearRegression
lr = LinearRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)
print lr.explainParams()
lrModel = lr.fit(df)
```

训练摘要

正如逻辑回归，可以得到模型的详细训练信息。代码字体方法可以快速访问这些度量标准，它包含了一些传统的衡量回归模型效果的指标，可以帮助看到模型是如何拟合曲线的。

summary方法返回具有多个字段的摘要对象，我们接下来一个一个来分析：残差（residuals）是我们输入进模型中每个特征的权重；目标历史（objective history）记录了每次迭代训练的情况；均方根误差（root mean squared error）是衡量曲线对数据的拟合程度的度量，通过检查每个预测值与实际值之间的距离来确定；R方（R-squared）变量是由模型捕获的预测变量方差与总方差比例的度量。

还有许多度量方法和模型摘要信息可能适合不同的应用场景，本章只介绍API，不会介绍所有的度量方法（请参考API文档来获取更详细信息）。

下面是线性回归模型摘要的一些属性：

```
// in Scala
val summary = lrModel.summary
summary.residuals.show()
println(summary.objectiveHistory.toSeq.toDF.show())
println(summary.rootMeanSquaredError)
println(summary.r2)

# in Python
summary = lrModel.summary
summary.residuals.show()
print summary.totalIterations
print summary.objectiveHistory
print summary.rootMeanSquaredError
```

```
print summary.r2
```

广义线性回归

在本章中看到的标准线性回归实际上是称为广义线性回归的算法系列的一部分。对这种算法，Spark中有两种的实现，一个是针对处理非常大的特征集（本章前面介绍的简单线性回归）进行了优化，另一个则更为通用，包括对更多算法的支持，目前并不支持处理大量的特性值。

线性回归的广义形式使你可以更细粒度地控制使用各种回归模型。例如，它可以选择不同的噪声分布，包括高斯分布(线性回归)、二项式分布 (逻辑回归)、泊松分布(泊松回归) 和伽玛分布 (伽玛回归)。广义模型还支持设置链接函数，指定线性预测器与分布函数均值之间的关系。表 27-2列出了每种分布类型的可用链接函数。

表27-2：？？？？？

分布	响应类型	支持的链接函数
高斯	连续	Identity *，Log，Reverse
二 项	二 进 制	Logit *，Probit，CLogLog
泊 松	计数	Log *，Identity，Sqrt
伽 玛	连续	Inverse*，Idenity，Log
Tweedie	零膨胀连续	Power link function

*表示每个分布类型的规范链接函数。

可以从ISL 3.2 和 ESL 3.2 获得关于广义线性模型的更多信息。

Spark 2.2 的一个限制是，广义线性回归最大只接受4096个特征作为输入。在之后的Spark版本中可能会改进，所以请留意参考文档。

模型超参数

这些配置是用于确定模型本身的基本结构而指定的，除了 `fitIntercept` 和 `regParam`（ 在24章的“回归”中介绍的 ）之外，广义线性回归还包括几个其他超参数：

`family`

指定在模型中使用的误差分布。支持的选项包括Poisson（泊松分布），binomial（二项式分布），gamma（伽马分布），Caussian（高斯分布）和tweedie（tweedie分布）。

`link`

链接函数的名称，指定线性预测器与分布函数平均值之间的关系。支持的选项是cloglog、probit、logit、reverse、sqrt、identity和log（默认是identity）

`solver`

指定的优化算法。当前唯一支持的优化算法是 irls (迭代重加权最小二乘法)。

`variancePower`

Tweedie分布方差函数中的幂，它刻画了分布的方差和平均值之间的关系，仅适用于 Tweedie 分布。支持的值为0和 [1，无穷大)，默认值为0。

`linkPower`

Tweedie分布的乘幂链接函数索引。

训练参数

训练参数与逻辑回归的训练参数相同。有关详细信息，请参阅第26章。

预测参数

此模型添加了一个预测参数：

`linkPredictionCol`

指定一个列名，为每个预测保存我们的链接函数的输出。

示例

下面是一个使用`GeneralizedLinearRegression`的示例：

```
// in Scala
import org.apache.spark.ml.regression.GeneralizedLinearRegression
val glr = new GeneralizedLinearRegression()
  .setFamily( “gaussian")
  .setLink( “identity")
  .setMaxIter(10)
  .setRegParam(0.3)
  .setLinkPredictionCol( “linkOut")
println(glr.explainParams())
```

```
val glrModel = glr.fit(df)

# in Python
from pyspark.ml.regression import GeneralizedLinearRegression
glr = GeneralizedLinearRegression()\
  .setFamily( "gaussian")\
  .setLink( "identity")\
  .setMaxIter(10)\
  .setRegParam(0.3)\
  .setLinkPredictionCol( "linkOut")
print glr.explainParams()
glrModel = glr.fit(df)
```

训练摘要

与前一节中介绍的简单线性模型一样，Spark也为广义线性模型提供训练摘要，它可以帮助确保模型适合该训练集的数据。必须注意的是，这并不会提供更适合的运行算法，但它可以提供更多的信息。此信息包括许多不同的度量标准，用于分析算法的适应性，包括一些最常见的度量方法：

R squared
: R方指标，用户评估拟合度。

The residuals
: 标签与预测值之间的差异。

请务必检查模型的摘要对象，以查看所有可用方法。

决策树

在回归中使用的决策树与用于分类的决策树很相似。主要的区别是，用于回归分析的决策树不是在每个叶子节点上输出离散的标签，而是一个连续的数值，但是可解释性和模型结构仍然适用。简而言之，回归决策树只是创建一个树来预测数值输出，而不是试图训练系数来建模一个函数。这点很重要，因为与广义线性回归不同，我们可以预测输入数据中的非线性函数，这也造成了过拟合的风险，因此在调整和评估这些模型时，我们需要小心。

在第26章已经介绍了决策树的内容。有关此主题的详细信息，请参阅 ISL 8.1 和 ESL 9.2。

模型超参数

用于回归的决策树模型超参数与用于分类的决策树超参数相同，只是在`impurity`参数

上稍有变化。有关其他超参数的详细信息，请参见第26 章：

`impurity`
: 该参数指示模型是否应在某叶子节点分割（根据信息增益），回归树当前支持的唯一度量方法是“variance”。

训练参数

除了超参数，分类树和回归树还采用相同的训练参数。有关这些参数，请参见第26章的"训练参数"。

示例

下面是使用决策树进行回归任务的例子：

```
// in Scala
import org.apache.spark.ml.regression.DecisionTreeRegressor
val dtr = new DecisionTreeRegressor()
println(dtr.explainParams())
val dtrModel = dtr.fit(df)

# in Python
from pyspark.ml.regression import DecisionTreeRegressor
dtr = DecisionTreeRegressor()
print dtr.explainParams()
dtrModel = dtr.fit(df)
```

随机森林和梯度提升树

随机森林（Random Forest）和梯度提升树（Gradient-Boosted Tree）模型可应用于分类和回归，它们与决策树具有相同的基本概念，不是训练一棵树而是很多树来做回归分析。在随机森林模型中，大量的非相关树被训练然后平均。使用梯度提升树，每棵树进行加权预测 (一些树对某些类的预测能力比其他的强)。随机森林和梯度提升树回归具有与相应分类模型相同的模型超参数和训练参数，只有纯度度量impurity不同（但是它与`DecisionTreeRegressor`的`impurity`一样设置）。

请查看ISL8.2和ESL0.1获得更多的关于决策树的其他内容。

模型超参数

这些模型与我们在上一章中看到的回归决策树的参数相同。有关这些参数的详细说

明，请参阅第26章的“模型超参数”。只有一点，与单个回归树相同，impurity参数当前仅支持variance。

训练参数

就像在第26章中所描述的，这些模型支持与分类树相同的checkpointInterval参数。

示例

这里有一个如何使用这两个模型进行回归的例子：

```
// in Scala
import org.apache.spark.ml.regression.RandomForestRegressor
import org.apache.spark.ml.regression.GBTRegressor
val rf = new RandomForestRegressor()
println(rf.explainParams())
val rfModel = rf.fit(df)
val gbt = new GBTRegressor()
println(gbt.explainParams())
val gbtModel = gbt.fit(df)

# in Python
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.regression import GBTRegressor
rf = RandomForestRegressor()
print rf.explainParams()
rfModel = rf.fit(df)
gbt = GBTRegressor()
print gbt.explainParams()
gbtModel = gbt.fit(df)
```

高级方法

之前所涉及的方法是执行回归的通用方法，它们提供了许多人使用的基本回归类型，但还有一些其他方法。下一章还将介绍一些更专业的回归模型，我们省略代码示例，因为它们遵循与其他算法相同的模式。

生存分析（加速失效时间模型）

统计学家一般在对照实验中使用生存分析来了解个体的存活率。Spark实现了加速失效时间模型（生存分析变体），而不是描述实际生存时间来对生存时间进行记录，Spark实现这种生存分析变体，是因为更著名的 Cox Proportional Hazard模型是半参数化的，也不能很好地扩展到大型数据集上。相比之下，加速失效时间模型是因为每个实例（行）都独立地贡献结果，这与Cox生存模型有不同的假设，因此这两个模型互

相补充而不能取代其一。本书不会讲解这些不同的假设，详细信息请参阅L. J. Wei的论文（*http://bit.ly/2rKxqcW*）。

生存回归对输入的要求与其他回归模型的要求非常相似，我们将根据特征值调整系数。然而有一处不同，它引入了一个censor变量列。在科学分析中，测试对象会检查某个体何时退出，因为他们在实验结束时的状态可能是未知的。这一点很重要，因为我们不能假设在研究的某个中间点个体会输出结果。

请参阅文档中的AFT部分获取更多关于生存回归的信息（*http://bit.ly/2nht2wD*）。

保序回归

保序回归（Isotonic Regression）是另一种特殊的回归模型，具有一些特别的要求。本质上，保序回归指定了一个总是单调递增的分段线性函数，它不会下降。 这意味着如果你的数据在训练目标图中总是向上和向右，这是一个合适的模型。 如果它随着输入会有波动变化，那么这是不合适的。

图27-1说明了保序回归的情况，便于理解。

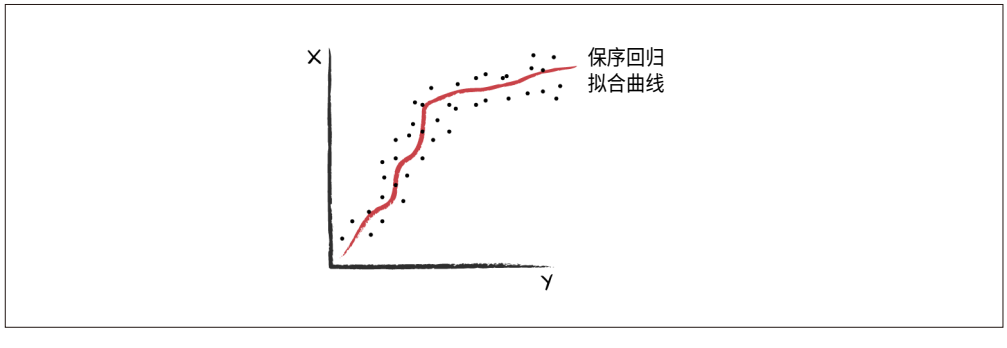


图27-1：保序回归曲线

请注意这条曲线为什么要比简单的线性回归拟合地更好，可以参考Spark文档关于如何使用该模型的说明。

评估器和自动化模型校正

回归具有与分类相同的核心模型优化功能，我们可以指定一个评估器，选择一个度量标准来优化，然后训练流水线在这一部分上执行参数调整。用于回归任务的评估器称为 `RegressionEvaluator` ，支持许多常见的回归度量标准。与分类评估器一

样，RegressionEvaluator 需要两项输入，一个表示预测值，另一个表示真实标签的值。支持的优化度量值包括均方根误差（“rmse”）、均方误差（“mse”）、r2度量（“r2”）和平均绝对误差(“mae”)。

要使用RegressionEvaluator，需要建立流水线，指定我们想要测试的参数，然后运行它。Spark将自动选择最佳的模型并将其返回给我们：

```
// in Scala
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.GeneralizedLinearRegression
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
val glr = new GeneralizedLinearRegression()
  .setFamily( “gaussian")
  .setLink( “identity")
val pipeline = new Pipeline().setStages(Array(glr))
val params = new ParamGridBuilder().addGrid(glr.regParam, Array(0, 0.5, 1))
  .build()
val evaluator = new RegressionEvaluator()
  .setMetricName( “rmse")
  .setPredictionCol( “prediction")
  .setLabelCol( “label")
val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEvaluator(evaluator)
  .setEstimatorParamMaps(params)
  .setNumFolds(2) // 至少应该是3，但是这里数据太少所以设置成为2
val model = cv.fit(df)

# in Python
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import GeneralizedLinearRegression
from pyspark.ml import Pipeline
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
glr = GeneralizedLinearRegression().setFamily( “gaussian").setLink( “identity")
pipeline = Pipeline().setStages([glr])
params = ParamGridBuilder().addGrid(glr.regParam, [0, 0.5, 1]).build()
evaluator = RegressionEvaluator()\
  .setMetricName( “rmse")\
  .setPredictionCol( “prediction")\
  .setLabelCol( “label")
cv = CrossValidator()\
  .setEstimator(pipeline)\
  .setEvaluator(evaluator)\
  .setEstimatorParamMaps(params)\
  .setNumFolds(2) # 至少应该是3，但是这里数据太少所以设置成为2
model = cv.fit(df)
```

度量标准

评估器允许我们根据一个特定的度量指标值来评估和调整模型，但我们也可以通

过RegressionMetrics对象获得多个回归度量指标。与上一章中的分类指标一样，RegressionMetrics 也是处理在RDD上的（预测值，标签值）对。下面看看如何检查之前训练好的模型的结果。

```
// in Scala
import org.apache.spark.mllib.evaluation.RegressionMetrics
val out = model.transform(df)
  .select(“prediction", “label")
  .rdd.map(x => (x(0).asInstanceOf[Double], x(1).asInstanceOf[Double]))
val metrics = new RegressionMetrics(out)
println(s"MSE = ${metrics.meanSquaredError}")
println(s"RMSE = ${metrics.rootMeanSquaredError}")
println(s"R-squared = ${metrics.r2}")
println(s"MAE = ${metrics.meanAbsoluteError}")
println(s"Explained variance = ${metrics.explainedVariance}")

# in Python
from pyspark.mllib.evaluation import RegressionMetrics
out = model.transform(df)\
  .select(“prediction", “label").rdd.map(lambda x: (float(x[0]), float(x[1])))
metrics = RegressionMetrics(out)
print “MSE: “ + str(metrics.meanSquaredError)
print “RMSE: “ + str(metrics.rootMeanSquaredError)
print “R-squared: “ + str(metrics.r2)
print “MAE: “ + str(metrics.meanAbsoluteError)
print “Explained variance: “ + str(metrics.explainedVariance)
```

请参考Spark文档查看包含的最新方法。

小结

在本章中，我们讨论了Spark回归的基本原理，包括我们如何训练模型以及如何确定训练结束。在下一章中，我们将了解推荐引擎，这是 MLlib 中比较流行的应用之一。

第28章

推荐系统

推荐系统很容易理解，通过研究人们的明确偏好（通过评级）或隐含偏好（通过观察到的行为），可以绘制用户与其他用户之间、或他们喜欢的产品之间的相似性，利用这种潜在的相似性，推荐引擎可以向用户进行新的推荐。

应用场景

推荐系统引擎是大数据的最佳应用场景之一，大规模地收集用户过往信息作为训练数据还算比较容易，而且这些数据可以用于很多领域中来建立用户与新内容之间的联系。Spark就是很多公司锁采用大规模推荐引擎工具：

电影推荐

Amazon、Netflix和HBO都希望向用户推荐用户感兴趣的电影和电视内容。Netflix使用Spark为其用户做大规模电影的推荐。

课程推荐

一所学校可能想通过学习相似学生喜欢或选修的课程来向学生推荐课程，以往的课程注册数据可以作为这项任务的训练数据。

在Spark中有一个经常被使用的推荐算法，交替最小二乘法（ALS）。该算法利用一种叫做协同过滤（collaborative filtering）的方法，这种方法仅使用用户过去的采购数据做推荐，它并不需要或使用用户或项目的其他任何特征，它还支持多种ALS变体（例如，显式或隐式反馈）。除了ALS，Spark还提供了在购物篮分析中用于发现关联规则的频繁模式挖掘（Frequent Pattern Mining）技术。最后，Spark的RDD API提供了低级的矩阵分解运算，但是本书不会讲解这部分。

基于交替最小二乘法的协同过滤

ALS（交替最小二乘法）为每个用户和物品建立k维的特征向量，从而可以通过用户和物品向量的点积来估算该用户对该物品的评分值，所以只需要用户-物品对的评分数据作为输入数据集，其中有三列：用户 id 列、物品（如电影）id 列和评分列。评分可以是显式的，即我们想要直接预测的数值等级；或隐式的，在这种情况下，每个分数表示用户和物品之间的交互强度 (例如，访问特定页的次数)，它衡量用户对该物品的偏好程度。根据输入的DataFrame，模型将产生特征向量，可以使用它来预测用户对尚未评分的物品的评级。

在实践中需要注意的一个问题是，该算法偏好推荐那些非常常见的或者有大量信息物品，如果你正在推出一种还没有用户熟悉的新产品，该算法就不会推荐它。此外，如果新用户加入平台，他们可能在训练集中没有任何打分记录，算法也将不知道推荐他们些什么。这些是称之为冷启动（Cold Start）问题，我们将在本章后面讨论。

在可扩展性方面，Spark在此任务中受欢迎的一个原因是MLlib中的算法和实现可以扩展到数百万用户，数百万物品和数十亿的评级。

模型超参数

这些配置用于确定模型的结构以及具体的协同过滤问题：

`rank`

rank（秩）确定了用户和物品特征向量的维度，这通常是通过实验来调整，一个重要权衡是过高的秩导致过拟合，而过低的秩导致不能做出最好的预测。默认值为10。

`alpha`

在基于隐式反馈（用户行为）的数据上进行训练时，alpha设置偏好的基线置信度（Baseline Confidence），这个值越大则越认为用户和他没有评分的物品之间没有关联。默认值1.0，需要通过实验调参。

`regParam`

控制正则化参数来防止过拟合，需要测试不同的值来找到针对你的问题的最优的值。默认值为0.1。

`implicitPrefs`

此布尔值指定是在隐式数据(`true`)还是显式数据(`false`)上进行训练(请参阅前面的讨论以知晓显式和隐式之间的差异)，应根据模型输入的数据来设置此值。如

果数据是基于对产品的被动交互(如通过单击或网页访问)，则应设置为隐式。与此相反，如果数据是显式分级的(例如，用户给这家餐厅4/5的星级)，则应设置为显式。默认值是显式。

`nonnegative`
: 如果设置为true，则将非负约束置于最小二乘问题上，并且只返回非负特征向量，这可以提高某些应用程序的性能。默认值为false。

训练参数

交替最小二乘法的训练参数与其他模型中的训练参数不同。这是因为我们将会对数据在整个集群中的分布方式进行更低级的控制，分布在集群中的数据组称为数据块，确定在每个块中放置的数据对训练时间有很大影响 (不是最终结果)。通常的经验法则是每个数据块大概分配一百万到五百万个评分值，如果每个数据块的数据量少于这个数字，则太多的数据块可能会影响性能。

`numUserBlocks`
: 这将确定将用户数据拆分成多少个数据块。默认值为10。

`numItemBlocks`
: 这将确定将物品数据拆分为多少个数据块。默认值为10。

`maxIter`
: 训练的迭代次数。改变它可能不会太影响结果，所以这不应该是调参的首要参数。默认值为10。在检查训练历史记录之后，如果发现迭代多少次之后误差曲线还没有平缓，可以考虑增大该值。

`checkpointInterval`
: 设置检查点可以在训练过程中保存模型状态，以便更快地从节点故障中恢复。可以使用SparkContext。setCheckpointDir设置检查点目录。

`seed`
: 指定随机种子帮助复现实验结果。

预测参数

预测参数用来指定如何用训练好的模型进行预测。在我们的例子中，有一个参数，就是冷启动策略 (通过coldStartStrategy设置)，该参数用来设置模型应为未出现过在训练集中的用户或物品推荐什么。

当在实际生产中使用模型时，冷启动的挑战就会凸显出来，新用户和/或物品没有评分的历史记录，因此模型无法做出推荐。当在Spark的`CrossValidator`或`TrainValidationSplit`中使用简单随机拆分时，也可能发生这种情况，遇到不在训练集中的用户和/或物品非常常见。

默认情况下，Spark在遇到实际模型中不存在的用户和/或物品时，预测值将使用NaN。当遇到新用户或物品时，可以配置系统使用某些默认的推荐。但是，这在训练过程中是不可取的，因为它会破坏评估器正确评估模型的能力，这使得我们无法合理地选择模型。Spark用户可以将`coldStartStrategy`参数设置为`drop`，以便把在DataFrame中预测为NaN值的行删除，排除NaN数据后将通过非NaN数据计算评估指标。`drop`和`nan`（默认值）是当前仅支持的冷启动策略。

示例

本示例将使用MovieLens电影评分数据集，它还没有被使用过，这个数据集具有与电影推荐相关的信息。我们将首先使用此数据集来训练模型：

```
// in Scala
import org.apache.spark.ml.recommendation.ALS
val ratings = spark.read.textFile( “/data/sample_movielens_ratings.txt")
  .selectExpr( “split(value , '::') as col")
  .selectExpr(
     “cast(col[0] as int) as userId",
     “cast(col[1] as int) as movieId",
     “cast(col[2] as float) as rating",
     “cast(col[3] as long) as timestamp")
val Array(training, test) = ratings.randomSplit(Array(0.8, 0.2))
val als = new ALS()
  .setMaxIter(5)
  .setRegParam(0.01)
  .setUserCol( “userId")
  .setItemCol( “movieId")
  .setRatingCol( “rating")
println(als.explainParams())
val alsModel = als.fit(training)
val predictions = alsModel.transform(test)

# in Python
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row
ratings = spark.read.text( “/data/sample_movielens_ratings.txt")\
  .rdd.toDF()\
  .selectExpr( “split(value , '::') as col")\
  .selectExpr(
     “cast(col[0] as int) as userId",
     “cast(col[1] as int) as movieId",
     “cast(col[2] as float) as rating",
```

```
    "cast(col[3] as long) as timestamp")
training, test = ratings.randomSplit([0.8, 0.2])
als = ALS()\
  .setMaxIter(5)\
  .setRegParam(0.01)\
  .setUserCol("userId")\
  .setItemCol("movieId")\
  .setRatingCol("rating")
print als.explainParams()
alsModel = als.fit(training)
predictions = alsModel.transform(test)
```

现在可以针对每个用户或影片输出评分最高的k个推荐。模型的recommendForAllUsers方法返回对应某个userId的DataFrame，包含推荐电影的数组，以及每个影片的评分。recommendForAllItems返回对应某个movieId的DataFrame以及最有可能给该影片打高分的前几个用户：：

```
// in Scala
alsModel.recommendForAllUsers(10)
  .selectExpr("userId", "explode(recommendations)").show()
alsModel.recommendForAllItems(10)
  .selectExpr("movieId", "explode(recommendations)").show()

# in Python
alsModel.recommendForAllUsers(10)\
  .selectExpr("userId", "explode(recommendations)").show()
alsModel.recommendForAllItems(10)\
  .selectExpr("movieId", "explode(recommendations)").show()
```

推荐系统的评估器

在涉及到冷启动策略时，可以在使用 ALS 时设置自动化的模型评估器。有一件事可能不太明显，就是这个推荐问题其实只是一种回归问题，由于我们正在预测给定用户的评分值，我们希望通过优化措施以减少预测的用户评分和真实值之间的差别。我们可以使用与第27章中相同的RegressionEvaluator来执行，可以将其放在流水线中以自动化训练过程。执行此操作时，还应将冷启动策略设置为drop而不是NaN，然后在生产系统中实际进行预测时将其切换回NaN：

```
// in Scala
import org.apache.spark.ml.evaluation.RegressionEvaluator
val evaluator = new RegressionEvaluator()
  .setMetricName("rmse")
  .setLabelCol("rating")
  .setPredictionCol("prediction")
val rmse = evaluator.evaluate(predictions)
println(s"Root-mean-square error = $rmse")
```

```
# in Python
from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator()\
  .setMetricName( “rmse")\
  .setLabelCol( “rating")\
  .setPredictionCol( “prediction")
rmse = evaluator.evaluate(predictions)
print( “Root-mean-square error = %f" % rmse)
```

度量指标

推荐结果可以使用标准的回归度量指标和一些特定于推荐的度量指标来衡量。毫无疑问，在评估推荐结果的问题上，比单纯的基于回归的评估更为复杂，这些指标对于评估最终模型特别有用。

回归度量指标

我们可以继续使用回归的度量指标来评估推荐结果，我们可以简单地查看每个用户和项目的实际评级与预测值的接近程度：

```
// in Scala
import org.apache.spark.mllib.evaluation.{
  RankingMetrics,
  RegressionMetrics}
val regComparison = predictions.select("rating", "prediction")
  .rdd.map(x => (x.getFloat(0).toDouble,x.getFloat(1).toDouble))
val metrics = new RegressionMetrics(regComparison)
```

```
# in Python
from pyspark.mllib.evaluation import RegressionMetrics
regComparison = predictions.select("rating", "prediction")\
  .rdd.map(lambda x: (x(0), x(1)))
metrics = RegressionMetrics(regComparison)
```

排名指标

我们还有另一个工具，就是排名指标，RankingMetric将推荐结果与给定用户的实际评分值 (或实际偏好) 进行比较，它不关注具体的评分值，而是关注算法是否能够把已经打分的物品再次推荐给用户。这需要我们做一些数据准备工作，可能需要参考第 II 部分复习下某些方法。首先，需要为给定的用户收集一组高评分的电影。在我们的例子中，我们将使用一个相当低的阈值： 电影评分在2.5以上的电影，选择一个合适的阈值属于商业决策。

```
// in Scala
import org.apache.spark.mllib.evaluation.{RankingMetrics, RegressionMetrics}
```

```
import org.apache.spark.sql.functions.{col, expr}
val perUserActual = predictions
  .where("rating > 2.5")
  .groupBy("userId")
  .agg(expr("collect_set(movieId) as movies"))

# in Python
from pyspark.mllib.evaluation import RankingMetrics, RegressionMetrics
from pyspark.sql.functions import col, expr
perUserActual = predictions\
  .where("rating > 2.5")\
  .groupBy("userId")\
  .agg(expr("collect_set(movieId) as movies"))
```

现在我们有一个用户的集合，连同他们对电影的真实打分，我们将为每个用户依据算法取得前10个推荐电影，我们再观察前10个推荐是否出现在排名前10的电影集合中。如果我们训练了一个很好的模型，它将正确地推荐用户真正喜欢的电影，否则它还没有能学习到用户的喜好：

```
// in Scala
val perUserPredictions = predictions
  .orderBy(col("userId"), col("prediction").desc)
  .groupBy("userId")
  .agg(expr("collect_list(movieId) as movies"))

# in Python
perUserPredictions = predictions\
  .orderBy(col("userId"), expr("prediction DESC"))\
  .groupBy("userId")\
  .agg(expr("collect_list(movieId) as movies"))
```

现在我们有两个 DataFrame，一个预测值，另一个是真实的打分最高的电影集合。我们可以将它们传递到 RankingMetrics 对象中，此对象接受这些组合的RDD作为参数，如以下连接操作和 RDD 转换的例子所示：

```
// in Scala
val perUserActualvPred = perUserActual.join(perUserPredictions, Seq("userId"))
  .map(row => (
    row(1).asInstanceOf[Seq[Integer]].toArray,
    row(2).asInstanceOf[Seq[Integer]].toArray.take(15)
  ))
val ranks = new RankingMetrics(perUserActualvPred.rdd)

# in Python
perUserActualvPred = perUserActual.join(perUserPredictions, ["userId"]).rdd\
  .map(lambda row: (row[1], row[2][:15]))
ranks = RankingMetrics(perUserActualvPred)
```

现在，可以根据这个排名来度量我们的推荐算法。例如，可以根据平均精精度值指标来查看算法的准确度如何，还可以查看某排名位置之前的准确度，如查看在哪个排名位置开始正向预测开始大量失败：

```
// in Scala
ranks.meanAveragePrecision
ranks.precisionAt(5)

# in Python
ranks.meanAveragePrecision
ranks.precisionAt(5)
```

频繁模式挖掘

除了 ALS 之外，MLlib提供的另一个推荐系统工具是频繁模式挖掘（Frequent Pattern Mining）。频繁模式挖掘，有时称为购物篮分析（market basket analysis），根据原始数据发现关联规则。例如，鉴于大量的交易，你可能会发现购买热狗的用户几乎总是购买热狗面包。这种技术可以应用于推荐中，尤其是当人们在购物时 (无论是在线还是离线)。Spark实现了频繁模式挖掘的FP-growth算法，有关此算法的更多信息，请参见Spark文档和 ESL 14.2。

小结

在这一章中，我们讨论了在Spark实践中最流行的机器学习算法——用于推荐系统的交替最小二乘法，了解了如何训练、调整和评估这个模型。在下一章中，我们将讨论无监督学习并来讨论聚类。

第29章

无监督学习

本章将介绍Spark可用于无监督学习的工具，主要关注聚类。一般来说，无监督学习的使用频率低于有监督学习，因为它的最终结果通常难以评估好坏。这些挑战随着数据规模变大可能会加剧，例如，高维空间中的聚类可能因为高维的属性而生成奇怪的聚类结果，这被称为维度灾难。维度灾难体现了这样一个事实：随着特征空间随着维度扩展，它变得越来越稀疏。这意味着，随着维数的增加，填充此空间以获得具有统计意义的结果所需的数据会迅速增加。此外，高维度会带来更多噪音，这可能会使你的模型在噪音数据中训练从而导致奇怪的结果。因此，在模型可扩展性表中，我们不但包括了计算力的限制，以及一些统计学上的建议，这些是启发式的指导，而不是严格要求。

就其核心内容，无监督学习尝试发现数据中的模式或获得数据底层结构的简明表示形式。

应用场景

下面是一些可能的应用场景，这些模式可能会揭示我们的数据中并不明显的主题、异常或分组：

查找数据中的异常

如果数据集中的大多数值都集中在一个较大的组中，而其外只有几个小组，则这些小组可能需要进一步研究。

主题建模

通过检查大量的文本，可以找到在这些不同文档中存在的共同主题。

模型的可扩展性

就像其他模型一样，模型扩展性限制非常重要。

表29-1：聚类模型可扩展性

模型	建议	计算限制	训练例子
k -means	最大值为50~100	特征乘以聚类 < 1000万	不限
二分k-means	最大值为50~100	特征乘以聚类 < 1000万	不限
GMM	最大值为50~100	特征乘以聚类 < 1000万	不限
LDA	可解释的数字	1000个主题	不限

先加载一些数值数据：

```
// in Scala
import org.apache.spark.ml.feature.VectorAssembler

val va = new VectorAssembler()
  .setInputCols(Array("Quantity", "UnitPrice"))
  .setOutputCol("features")

val sales = va.transform(spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/data/retail-data/by-day/*.csv")
  .limit(50)
  .coalesce(1)
  .where("Description IS NOT NULL"))

sales.cache()

# in Python
from pyspark.ml.feature import VectorAssembler
va = VectorAssembler()\
  .setInputCols(["Quantity", "UnitPrice"])\
  .setOutputCol("features")

sales = va.transform(spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/data/retail-data/by-day/*.csv")
  .limit(50)
  .coalesce(1)
  .where("Description IS NOT NULL"))
```

```
sales.cache()
```

k-means

k-means是最受欢迎的聚类算法之一。在此算法中，用户在数据集中随机选择数量为k的数据点作为处理聚类的聚类中心，未分配的点基于他们与这些聚类中心的相似度(以欧氏距离计算)然后被 "分配"到离它们最近的聚类中。分配之后，再根据被分配到一个聚类的数据点再计算聚类的新中心 (称为质心)，并重复该过程，直到到达有限的迭代次数或直到收敛 (即质心位置停止改变)。这种方法并不一定合理，例如，如果实际的一个聚类可能被错误地拆分为两个，仅仅是因为开始的时候选择了两个初始点，因此以不同方式选择初始点来多次执行k-means会取得比较好的效果。

选择正确的聚类个数k是保证kmeans算法有效的一个关键，这也是一个难点。对于聚类数量该设置为多少，本书也并没有什么建议，所以你可能需要根据你期望的最终结果，不得不尝试不同的值来看看效果。

有关k-means的详细信息，请参阅 ISL 10.3 和 ESL 14.3。

模型超参数

这些是用来指定模型基本结构的配置：

k

指定你希望的聚类数量。

训练参数

`initMode`

初始化模式是确定质心初始位置的算法。支持的选项为`random`（随机初始化）和`k-means| |`(默认值)。后者是`k-means||`的并行化变体，虽然本书不介绍细节，但后一种方法的思想不是简单地选择随机初始化位置，而是选择具有良好分布的聚类中心，以产生更好的聚类。

`initSteps`

k-means||模式初始化所需要的步数。必须大于0，默认值是2

`maxIter`

迭代次数。改变这个值可能不会对最终结果改变很大，所以这不应该是你首先考虑调整的参数。默认20。

tol

该阈值指定质心改变小到该程度后，就认为模型已经优化的足够了，可以在迭代maxIter次之前停止运行。默认值为 0.0001。

此算法对这些参数通常是具有鲁棒性的，主要是一个均衡问题，运行更多的初始化步骤和迭代次数可能会生成更好的聚类，代价则是更长的训练时间：

示例

```
// in Scala
import org.apache.spark.ml.clustering.KMeans
val km = new KMeans().setK(5)
println(km.explainParams())
val kmModel = km.fit(sales)

# in Python
from pyspark.ml.clustering import KMeans
km = KMeans().setK(5)
print km.explainParams()
kmModel = km.fit(sales)
```

k-means评估指标摘要

k-means包括可用于评估模型的摘要类。此类为*k*-means运行成功提供了一些常用的衡量标准(这些方法是否适用于你的问题集是另一个问题)，*k*-means摘要包括有关创建聚类的信息以及聚类的相对大小（示例数）。

我们还可以使用computeCost计算类内平方误差和，这可以帮助衡量聚类内数据点与每个聚类中心点相距有多近。*k*-means的隐式目标是，基于给定的聚类数量*k*，来最小化聚类内平方误差的和：

```
// in Scala
val summary = kmModel.summary
summary.clusterSizes // 中心点的数量
kmModel.computeCost(sales)
println("Cluster Centers: ")
kmModel.clusterCenters.foreach(println)

# in Python
summary = kmModel.summary
print summary.clusterSizes # 中心点的数量
kmModel.computeCost(sales)
centers = kmModel.clusterCenters()
print("Cluster Centers: ")
for center in centers:
print(center)
```

二分k-means

二分*k*-means是*k*-means的变体，关键区别在于，它不是通过"自下而上"（bottom-up）地聚类，而是自上而下（top-down）的聚类方法。它将首先创建一个组，然后将该组再拆分成较小的组，以此类推，直到满足用户指定的k个组。这通常是比k-means更快捷的方法，将得到与*k*-means不同的结果。

模型超参数

用来指定模型的基本配置：

k

聚类数量。

训练参数

`minDivisibleClusterSize`

指定一个可分聚类中的最少的数据点数 (如果大于或等于1.0) 或数据点的最小比例 (如果小于1.0)，当聚类中数据点数小于该值时，聚类就不可再分割了。默认值为 1.0，这意味着每个聚类中必须至少有一个点。

`maxIter`

迭代次数。改变迭代次数可能不会对聚类的最终结果造成太大影响，所以这不应该是调参的首要选项。默认是20。

该模型中的大多数参数都应进行调参以找到最佳结果，没有适用于所有数据集的规则。

示例

```
// in Scala
import org.apache.spark.ml.clustering.BisectingKMeans
val bkm = new BisectingKMeans().setK(5).setMaxIter(5)
println(bkm.explainParams())
val bkmModel = bkm.fit(sales)

# in Python
from pyspark.ml.clustering import BisectingKMeans
bkm = BisectingKMeans().setK(5).setMaxIter(5)
bkmModel = bkm.fit(sales)
```

二分k-means摘要

二分*k*-means包括一个摘要类，们可以使用它来评估模型。它和*k*-means摘要类大部分相同，包括有关创建的聚类信息以及聚类的相对大小 (数据点数量)：

```
// in Scala
val summary = kmModel.summary
summary.clusterSizes // 中心点的数量
kmModel.computeCost(sales)
println("Cluster Centers: ")
kmModel.clusterCenters.foreach(println)
```

```
# in Python
summary = kmModel.summary
print summary.clusterSizes # 中心点的数量
kmModel.computeCost(sales)
centers = kmModel.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

高斯混合模型

高斯混合模型（Gaussian mixture models，GMM）是另一种流行的聚类算法，它的假设不同于二分*k*-means和*k*-means算法，那些算法尝试通过降低数据点和聚类中心之间的距离平方和来对数据进行分组，而高斯混合模型假设每个聚类中的数据点符合高斯分布。这意味着数据点在聚类边缘 (根据高斯分布) 的可能性较小，而数据点在中心附近的概率更高。每个高斯聚类的均值和标准差各不相同，可以是任意大小 (因此可能是各不相同的椭圆形的形状)。在训练过程中仍然需要用户指定聚类的k值。

一种简单理解高斯混合模型的方法是，它们就像*k*-means的软聚类版本（软聚类soft clustering即每个数据点可以划分到多个聚类中），而*k*-means创建硬聚类（即每个点仅在一个聚类中），高斯混合模型GMM依照概率而不是硬性边界进行聚类。

有关的更多信息，请参见 ESL 14.3。

模型超参数

这些参数用来确定模型的基本结构：

k

聚类数量。

训练参数

`maxIter`

迭代次数。改变这个值可能不会对最终聚类结果有太大影响，所以它不应该是你首先考虑调整的参数。默认为100。

`tol`

指定一个阈值来代表将模型优化到什么程度就够了。越小的阈值将需要更多的迭代次数作为代价（不会超过`maxIter`），也可以得到更高的准确度。默认值为0.01。

与*k*-means模型一样，这些训练参数一般不会受到聚类数量*k*的影响。

示例

```
// in Scala
import org.apache.spark.ml.clustering.GaussianMixture
val gmm = new GaussianMixture().setK(5)
println(gmm.explainParams())
val model = gmm.fit(sales)

# in Python
from pyspark.ml.clustering import GaussianMixture
gmm = GaussianMixture().setK(5)
print gmm.explainParams()
model = gmm.fit(sales)
```

高斯模型摘要

与其他聚类算法一样，高斯混合模型包括一个摘要类来帮助模型评估。这包括创建的聚类信息，如高斯混合的权重、均值和协方差，这可以帮助我们进一步了解数据的隐藏信息：

```
// in Scala
val summary = model.summary
model.weights
model.gaussiansDF.show()
summary.cluster.show()
summary.clusterSizes
summary.probability.show()

# in Python
summary = model.summary
print model.weights
model.gaussiansDF.show()
summary.cluster.show()
summary.clusterSizes
```

```
summary.probability.show()
```

LDA主题模型

隐含狄利克雷分布（Latent Dirichlet Allocation，LDA）是一种通常用于对文本文档执行主题建模的分层聚类模型。LDA 试图从与这些主题相关联的一系列文档和关键字中提取高层次的主题，然后它将每个文档解释为多个输入主题的组合。你可以使用两种实现方案：在线LDA（online LDA）和EM算法。一般来说，当有更多的输入样本时，在线的LDA表现得更好；当有较大的输入词库时，EM优化器效果更好。此方法还可以扩展到成百上千个主题。

要把文本数据输入进LDA中，首先要将其转化为数值格式，这可以通过`CountVectorizer`来实现。

模型超参数

这些参数是用来指定确定模型基本结构的：

k

用于指定从数据中提取的主题数量。默认值是10，并且必须是整数。

`docConcentration`

文档分布的浓度（Concentration）参数向量 (通常称为 "alpha")，它是狄利克雷分布的参数，值越大意味着越平滑 (正则化程度更高)。

如果未经用户设置，则`docConcentration`自动取值。如果设置为单值向量[alpha]，则 alpha 被重复复制到长度为*k*的向量中，`docConcentration`向量的长度必须为 。

`topicConcentration`

主题分布的浓度参数向量（通常命名为“beta”或“eta”），它是一个对称狄利克雷分布的参数。如果用户未设置，则`topicConcentration`将被自动设置。

训练参数

这些配置是用来指导训练过程的：

`maxIter`

最大迭代次数。改变这个值可能对结果不会有多大影响，所以这不应该是你首先考虑调整的参数。默认值20。

optimizer
: 指定是使用EM还是在线训练方法来优化LDA 模型。默认为online。

learningDecay
: 学习率，即指数衰减率。应该介于 (0.5、1.0] 之间以保证渐近收敛。默认值为0.51，仅适用于在线优化程序。

learningOffset
: 一个正数值的学习参数，在前几次迭代中会递减。较大的值使前期迭代次数较少。默认值为1024.0，仅适用于在线优化程序。

optimizeDocConcentration
: 指示docConcentration(文档主题分布的狄利克雷参数)是否在训练过程中进行优化。默认值为true，但仅适用于在线优化程序。

subsamplingRate
: 在微型批量梯度下降的每次迭代中采样的样本比例，范围是(0、1]。默认值为0.5，仅适用于在线优化程序。

seed
: 该模型还支持指定一个随机种子，用于实验重现。

checkpointInterval
: 和在第26章中看到的一样，是用于设置检查点的参数。

预测参数

topicDistributionCol
: 将每个文档的主题混合分布输出作为一列保存起来。

示例

```
// in Scala
import org.apache.spark.ml.feature.{Tokenizer, CountVectorizer}
val tkn = new Tokenizer().setInputCol("Description").setOutputCol("DescOut")
val tokenized = tkn.transform(sales.drop("features"))
val cv = new CountVectorizer()
  .setInputCol("DescOut")
  .setOutputCol("features")
  .setVocabSize(500)
  .setMinTF(0)
  .setMinDF(0)
  .setBinary(true)
val cvFitted = cv.fit(tokenized)
```

```
val prepped = cvFitted.transform(tokenized)

# in Python
from pyspark.ml.feature import Tokenizer, CountVectorizer
tkn = Tokenizer().setInputCol("Description").setOutputCol("DescOut")
tokenized = tkn.transform(sales.drop("features"))
cv = CountVectorizer()\
  .setInputCol("DescOut")\
  .setOutputCol("features")\
  .setVocabSize(500)\
  .setMinTF(0)\
  .setMinDF(0)\
  .setBinary(True)
cvFitted = cv.fit(tokenized)
prepped = cvFitted.transform(tokenized)

// in Scala
import org.apache.spark.ml.clustering.LDA
val lda = new LDA().setK(10).setMaxIter(5)
println(lda.explainParams())
val model = lda.fit(prepped)

# in Python
from pyspark.ml.clustering import LDA
lda = LDA().setK(10).setMaxIter(5)
print lda.explainParams()
model = lda.fit(prepped)
```

训练完模型后，你将看到一些排名靠前的主题，返回单词的索引。我们必须使用训练的`CountVectorizerModel`来找到这些单词的真实语义，例如训练后发现的前3个相关主题是hot、home和brown。

```
// in Scala
model.describeTopics(3).show()
cvFitted.vocabulary

# in Python
model.describeTopics(3).show()
cvFitted.vocabulary
```

这种方法会产生所用词的详细信息以及特定单词的强调，这些有助于更好地理解潜在的主题。由于篇幅限制，我们无法显示（所有的）输出。使用类似API，其实有更多的评估方法，如对数似然（log likelihood）和困惑度（perplexity）。这些工具的目标是帮助用户根据数据的分布情况优化主题的数量，你应该将这些指标应用到一个保留集（holdout set）上，以减少模型的整体困惑度。另一种选择是以提高保留集的对数似然为目标的优化。我们可以通过将数据集传递到以下函数来计算相应的指标值：`model.logLikelihood` 和`model.logPerplexity`。

小结

本章介绍了Spark支持的一些常用的无监督学习算法。下一章将走出MLlib，讨论Spark之外的高级分析生态系统。

第30章

图分析

上一章介绍了一些常规的无监督学习技术，本章将深入介绍一个更专业的工具集：图处理。图是由节点或顶点(任意对象) 和边(定义这些节点之间的关系) 组成的数据结构，图分析是分析这些关系的过程，比如这个图可能是你的朋友圈，每个顶点或节点代表一个人，每一条边代表一个人与人之间的关系。图30-1给出了一个图结构的示例。

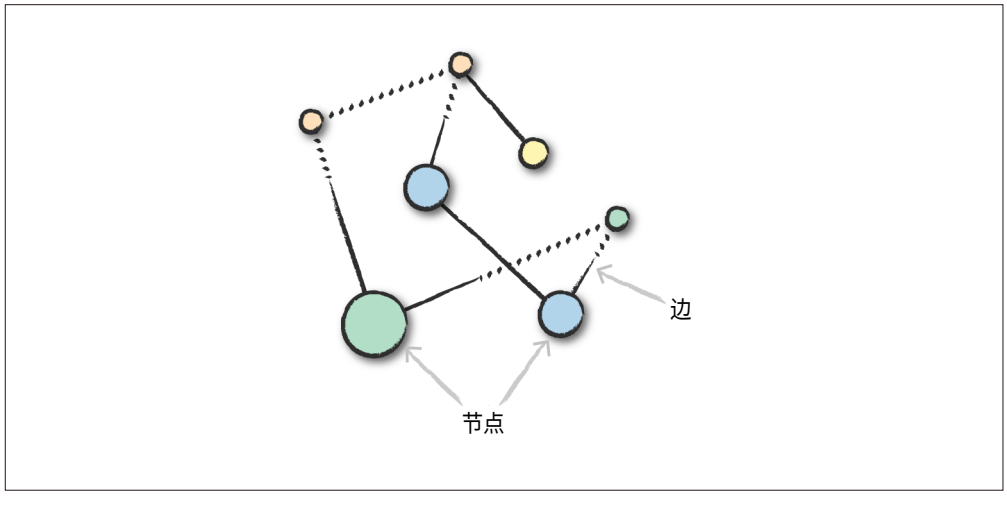


图30-1：具有七个节点和七条边的图

此关系图为无向图，因为边没有指定“起始”顶点和“终止”顶点，即没有方向。除此之外还有有向图，有向图中的边指定起始顶点和结束顶点。图30-2就是一个有向图，其中的边是有方向的。

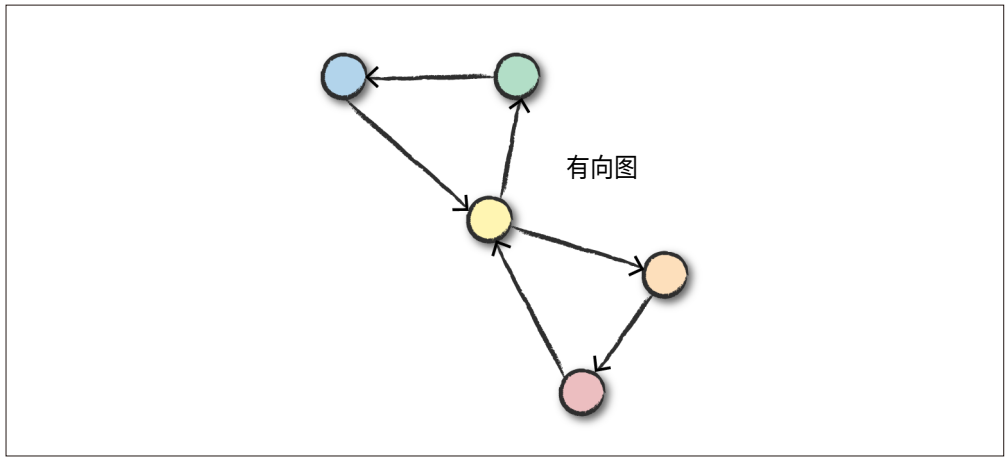


图30-2：有向图

图30-2中的边和顶点也可以有与之关联的数据。在朋友关系图的例子中，边的权重可能代表不同的朋友之间的亲密关系，例如普通朋友之间的边权重较低，而结婚了的两人之间的边权重较大。我们可以通过查看节点之间的通信频率为边添加权重，每个顶点 (人) 也可能有数据，如人名。

图结构是刻画关系数据和许多问题的一种很自然的数据结构，而Spark提供了几种用来图分析的有用方法。可能的一些商业应用场景包括检测信用卡欺诈、模式发现、确定引文网络中的论文重要性 (例如，哪些论文引用量最高)，以及网页排名，就像 Google 著名的PageRank算法。

Spark一直提供一个基于RDD的库来执行图处理： GraphX。它提供了一个非常低级的接口，非常强大，但就像 RDD一样不易使用或优化。GraphX 仍然是Spark的核心部分。我们仍然在GraphX上构建生产应用程序，而且仍然可以看到一些功能扩展。GraphX API 提供很全面的的记录，因为它自创建以来没有什么变化。然而，一些Spark的开发者 (包括一些 GraphX 的原创者) 最近基于Spark创建了一个新一代的图分析库： GraphFrame。GraphFrames 扩展了GraphX ，提供 DataFrame API 和对Spark不同语言的支持，以便 Python 用户可以使用该工具利用其可扩展性。在这本书中，我们将集中讨论 GraphFrame。

GraphFrame当前是一个Spark外部包，在启动Spark应用程序的时候需要进行加载操作，但未来可能会合并到Spark的核心模块中。在大多数情况下，GraphX和GraphFrame之间的性能差别不大 (除了 GraphFrame的用户体验改进)。使用

GraphFrame还有一些微量开销，但在多数情况下它试图在适当的时候调用 GraphX。对大多数人来说，用户体验提升的收益远远大于这个轻微的性能开销。

GraphFrame与图数据库比较会怎么样？

Spark不是数据库，Spark是一个分布式计算引擎，但它不长期存储数据或执行事务。你可以基于Spark建立一个图计算任务，但这与数据库有根本的不同。GraphFrame相比于许多图数据库，可以扩展到更大规模，处理更大的图数据集，并且在图分析上性能很好，但不支持事务处理和服务。

本章将向你展示如何使在Spark上使用 GraphFrame执行图分析。我们将采用湾区共享单车门户（Bay Area Bike Share portal）开放的共享单车数据来做分析。

在编写本书的过程中，此数据集发生了很大变化 (甚至命名!)。我们在本书的资料库的data目录下包含了该数据集的副本，请确保使用该数据集执行下面的实验。

首先需要指向合适的包，如果从命令行执行此操作，你需要运行：

```
./bin/spark-shell --packages graphframes:graphframes:0.5.0-spark2.2-s_2.11

// in Scala
val bikeStations = spark.read.option("header","true")
  .csv("/data/bike-data/201508_station_data.csv")
val tripData = spark.read.option("header","true")
  .csv("/data/bike-data/201508_trip_data.csv")

# in Python
bikeStations = spark.read.option("header","true")\
  .csv("/data/bike-data/201508_station_data.csv")
tripData = spark.read.option("header","true")\
  .csv("/data/bike-data/201508_trip_data.csv")
```

构建图

第一步是构建图。要做到这一点，我们需要定义顶点和边，这是具有一些命名列的DataFrame。在我们的例子中，我们正在创建一个有向图，具有从源地址指向目的地址的边。在共享单车行程数据集中，包含从行程的开始位置到行程的结束位置。为了定义图，我们基于GraphFrames库中列的命名约定。在顶点表中，我们将车站名标识

符定义为 id (在该示例中是字符串类型)。在边表中，我们将每条边的源顶点ID 标记为src ，目的地顶点ID为dst：

```
// in Scala
val stationVertices = bikeStations.withColumnRenamed("name", "id").distinct()
val tripEdges = tripData
  .withColumnRenamed("Start Station", "src")
  .withColumnRenamed("End Station", "dst")

# in Python
stationVertices = bikeStations.withColumnRenamed("name", "id").distinct()
tripEdges = tripData\
  .withColumnRenamed("Start Station", "src")\
  .withColumnRenamed("End Station", "dst")
```

接下来可以基于这些顶点DataFrame和边 DataFrame创建一个 GraphFrame 对象来表示我们的图。我们还将利用缓存，因为我们将在以后的查询中频繁访问此数据：

```
// in Scala
import org.graphframes.GraphFrame
val stationGraph = GraphFrame(stationVertices, tripEdges)
stationGraph.cache()

# in Python
from graphframes import GraphFrame
stationGraph = GraphFrame(stationVertices, tripEdges)
stationGraph.cache()
```

现在，我们可以看到这个图的基本信息 (并查询原始 DataFrame 以确保我们看到的是预期的结果)：

```
// in Scala
println(s"Total Number of Stations: ${stationGraph.vertices.count()}")
println(s"Total Number of Trips in Graph: ${stationGraph.edges.count()}")
println(s"Total Number of Trips in Original Data: ${tripData.count()}")

# in Python
print "Total Number of Stations: " + str(stationGraph.vertices.count())
print "Total Number of Trips in Graph: " + str(stationGraph.edges.count())
print "Total Number of Trips in Original Data: " + str(tripData.count())
```

返回的结果是：

```
Total Number of Stations: 70
Total Number of Trips in Graph: 354152
Total Number of Trips in Original Data: 354152
```

查询图

操作图的最基本的方法是查询，诸如执行行程计数和根据某目的地进行过滤操作等，GraphFrame提供类似操作DataFrame的对顶点和边的简单访问操作。请注意，除了ID、源顶点和目的地顶点之外，我们的图还保留了数据中的其他列，因此也可以在需要时查询它们：

```
// in Scala
import org.apache.spark.sql.functions.desc
stationGraph.edges.groupBy("src", "dst").count().orderBy(desc("count")).show(10)

# in Python
from pyspark.sql.functions import desc
stationGraph.edges.groupBy("src", "dst").count().orderBy(desc("count")).show(10)

+--------------------+--------------------+-----+
|                 src|                 dst|count|
+--------------------+--------------------+-----+
|San Francisco Cal...|     Townsend at 7th| 3748|
|Harry Bridges Pla...|Embarcadero at Sa...| 3145|
...
|     Townsend at 7th|San Francisco Cal...| 2192|
|Temporary Transba...|San Francisco Cal...| 2184|
+--------------------+--------------------+-----+
```

我们还可以通过任何有效的 DataFrame 表达式进行过滤操作。在这个例子中，我们想看看一个特定的站点和往返于该站的次数：

```
// in Scala
stationGraph.edges
  .where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")
  .groupBy("src", "dst").count()
  .orderBy(desc("count"))
  .show(10)

# in Python
stationGraph.edges\
  .where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")\
  .groupBy("src", "dst").count()\
  .orderBy(desc("count"))\
  .show(10)

+--------------------+--------------------+-----+
|                 src|                 dst|count|
+--------------------+--------------------+-----+
|San Francisco Cal...|     Townsend at 7th| 3748|
|     Townsend at 7th|San Francisco Cal...| 2734|
...
|   Steuart at Market|     Townsend at 7th|  746|
|     Townsend at 7th|Temporary Transba...|  740|
+--------------------+--------------------+-----+
```

子图

子图就是一个大图中的小图。我们在最后前一节中看到了如何查询给定的一组边和顶点，我们可以使用此查询功能创建子图：

```
// in Scala
val townAnd7thEdges = stationGraph.edges
  .where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")
val subgraph = GraphFrame(stationGraph.vertices, townAnd7thEdges)

# in Python
townAnd7thEdges = stationGraph.edges\
  .where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")
subgraph = GraphFrame(stationGraph.vertices, townAnd7thEdges)
```

然后，我们可以将以下算法应用于原始图或应用于子图上。

模式发现

Motif是图的结构化模式的一种表现形式。当指定一个motif时，查询的数据中的模式而不是实际的数据。在 GraphFrame中，我们采用具体领域语言 (类似于Neo4J的Cypher语言) 来指定查询。此语言允许我们指定顶点和边的组合，并给它们指定名称。例如，如果要指定给定顶点a通过边ab连接到另一个顶点b，就要指定(a)-[ab]->(b)，小括号或中括号内的名称不表示值，而是结果 DataFrame 中命名匹配顶点和边的列名。如果不打算查询结果值，则可以省略名称 (例如(a)-[]->())。

我们来对共享单车数据执行查询。简单来说，找到在三个站点之间形成“三角”模式的所有行程。 我们用下面的表达来声明motif，使用find方法在GraphFrame中查找该模式。(a)表示起始站点，[ab]表示从(a)到下一站(b)的边。对于站点(b)到站点(c)，以及从站点(c)到站点(a)，我们重复此操作：

```
// in Scala
val motifs = stationGraph.find("(a)-[ab]->(b); (b)-[bc]->(c); (c)-[ca]->(a)")

# in Python
motifs = stationGraph.find("(a)-[ab]->(b); (b)-[bc]->(c); (c)-[ca]->(a)")
```

图30-3提供了此查询的可视表达。

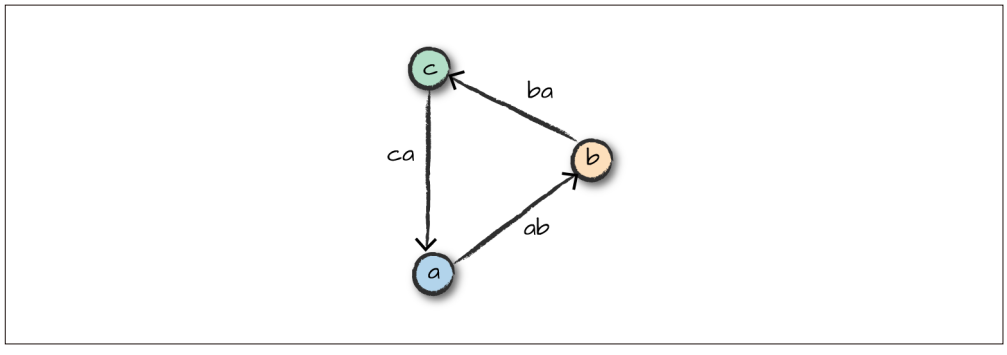


图30-3：在三角查询中发现的三角模式

运行此查询所得到的 DataFrame 包含顶点a，b和c的嵌套字段以及各自的边，然后可以采用DataFrame查询它们。例如，给定一辆自行车，自行车从 a 站到 b 站，再到 c 站，再到车站 a 的最短行程是哪条？ 下面代码将我们的时间戳解析为Spark时间戳，然后进行比较以确保它是相同的自行车，从一个站到另一个站以及每个行程的开始时间都是正确的：

```
// in Scala
import org.apache.spark.sql.functions.expr
motifs.selectExpr("*",
    "to_timestamp(ab.`Start Date`, 'MM/dd/yyyy HH:mm') as abStart",
    "to_timestamp(bc.`Start Date`, 'MM/dd/yyyy HH:mm') as bcStart",
    "to_timestamp(ca.`Start Date`, 'MM/dd/yyyy HH:mm') as caStart")
  .where("ca.`Bike #` = bc.`Bike #`").where("ab.`Bike #` = bc.`Bike #`")
  .where("a.id != b.id").where("b.id != c.id")
  .where("abStart < bcStart").where("bcStart < caStart")
  .orderBy(expr("cast(caStart as long) - cast(abStart as long)"))
  .selectExpr("a.id", "b.id", "c.id", "ab.`Start Date`", "ca.`End Date`")
  .limit(1).show(false)

# in Python
from pyspark.sql.functions import expr
motifs.selectExpr("*",
    "to_timestamp(ab.`Start Date`, 'MM/dd/yyyy HH:mm') as abStart",
    "to_timestamp(bc.`Start Date`, 'MM/dd/yyyy HH:mm') as bcStart",
    "to_timestamp(ca.`Start Date`, 'MM/dd/yyyy HH:mm') as caStart")\
  .where("ca.`Bike #` = bc.`Bike #`").where("ab.`Bike #` = bc.`Bike #`")\
  .where("a.id != b.id").where("b.id != c.id")\
  .where("abStart < bcStart").where("bcStart < caStart")\
  .orderBy(expr("cast(caStart as long) - cast(abStart as long)"))\
  .selectExpr("a.id", "b.id", "c.id", "ab.`Start Date`", "ca.`End Date`")
  .limit(1).show(1, False)
```

可以看到最快的路程是大约20分钟，对于三个不同的人 (假设是不同的) 使用相同的自行车来说，这是相当快的!

还要注意，我们必须过滤这个例子中通过模式发现返回的三角行程。通常，查询中使用的不同顶点 ID不会强制匹配不同的顶点，所以如果需要不同的顶点，则应执行此类型的过滤。GraphFrames最强大的功能之一是，你可以将模式发现与 DataFarme 查询合并到结果表中，以进一步缩小、排序或聚合找到的模式。

图算法

图只是数据的逻辑表示形式。图论提供了许多用于分析此格式数据的算法，GraphFrame支持许多图算法。随着Spark项目的发展，新的图算法也会不断添加到GraphFrame中，因此支持的图算法会越来越多。

PageRank

广受欢迎的图算法之一是PageRank。谷歌联合创始人Larry Page提出PageRank来对网页进行排名。对PageRank如何工作的完整解释超出了本书的范围，我们引用维基百科的解释如下：

> PageRank通过计算网页链接的数量和质量来确定网站的重要性。它根本的假设是，重要的网站可能会被更多其他网站所链接。

PageRank在网页排名之外的领域也有广泛应用，我们可以将这一方法应用到我们的共享单车数据集中，并得到重要的单车站（特别是那些有很多共享单车行经的车站）。在这个例子中，重要的单车站将被分配更大的排名值：

```
// in Scala
import org.apache.spark.sql.functions.desc
val ranks = stationGraph.pageRank.resetProbability(0.15).maxIter(10).run()
ranks.vertices.orderBy(desc("pagerank")).select("id", "pagerank").show(10)

# in Python
from pyspark.sql.functions import desc
ranks = stationGraph.pageRank(resetProbability=0.15, maxIter=10)
ranks.vertices.orderBy(desc("pagerank")).select("id", "pagerank").show(10)
```

```
+--------------------+------------------+
|                  id|          pagerank|
+--------------------+------------------+
|San Jose Diridon ...| 4.051504835989922|
|San Francisco Cal...|3.3511832964279518|
...
|     Townsend at 7th| 1.568456580534273|
|Embarcadero at Sa...|1.5414242087749768|
+--------------------+------------------+
```

图算法API： 参数和返回值

GraphFrame中的大多数算法都是接受参数的方法(例如，在这个PageRank示例中的 resetProbability)，大多数算法返回的是一个新的 GraphFrame 。算法的结果存储在GraphFrame 的顶点和/或边上，或者是返回一个DataFrame 。对于PageRank，算法返回一个 GraphFrame，可以从新的PageRank列中提取每个顶点的PageRank值。

根据计算机上的可用资源，图算法计算可能需要一些时间，在运行实际数据之前，可以先尝试少量的数据来提前看看结果。Databricks 社区版的PageRank处理该数据集，需要大约20秒的时间来运行，有一些用户发现在他们自己的机器上需要更长的时间。

有趣的是，我们看到 Caltrain 站排名相当高。这是有道理的，因为很多单车行程都与此站相关，人们要么从家到Caltrain站来上班，要么从Caltrain站回家。

入度出度指标

我们的图是有向图，这是由于单车行程是有向的，从一个站点开始，并在另一个站点结束。一个常见的任务是计算进出某一站点的次数。为了测量进出车站的行程数，我们将分别使用一种名为“入度”和“出度”的度量，如图30-4所示。

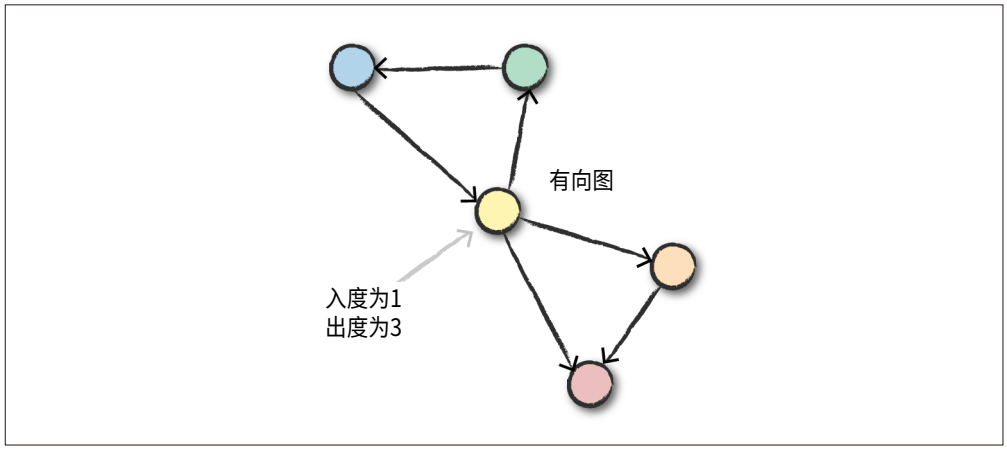


图30-4：入度和出度

它在社交网络中很有用，因为某些用户可能拥有比出连接（即他们关注的人）更多的入连接（即关注他们的人）。 使用以下查询，可以在社交网络中找到更具影响力的人。 GraphFrame提供了一种查询图顶点入度和出度的简单方法：

```
// in Scala
val inDeg = stationGraph.inDegrees
inDeg.orderBy(desc("inDegree")).show(5, false)

# in Python
inDeg = stationGraph.inDegrees
inDeg.orderBy(desc("inDegree")).show(5, False)
```

按照车站入度由高到低排序得到的查询结果：

```
+----------------------------------------+--------+
|id                                      |inDegree|
+----------------------------------------+--------+
|San Francisco Caltrain (Townsend at 4th)|34810   |
|San Francisco Caltrain 2 (330 Townsend) |22523   |
|Harry Bridges Plaza (Ferry Building)    |17810   |
|2nd at Townsend                         |15463   |
|Townsend at 7th                         |15422   |
+----------------------------------------+--------+
```

我们可以以相同的方式查询出度：

```
// in Scala
val outDeg = stationGraph.outDegrees
outDeg.orderBy(desc("outDegree")).show(5, false)

# in Python
outDeg = stationGraph.outDegrees
outDeg.orderBy(desc("outDegree")).show(5, False)
```

```
+-------------------------------------------+---------+
|id                                         |outDegree|
+-------------------------------------------+---------+
|San Francisco Caltrain (Townsend at 4th)   |26304    |
|San Francisco Caltrain 2 (330 Townsend)    |21758    |
|Harry Bridges Plaza (Ferry Building)       |17255    |
|Temporary Transbay Terminal (Howard at Beale)|14436    |
|Embarcadero at Sansome                     |14158    |
+-------------------------------------------+---------+
```

这两个值的比值是一个有趣的指标，更高的比值告诉我们在哪里有大量的行程结束(但很少有行程从这里开始)，而较低的值告诉我们行程通常开始于哪里(但很少有行程在这里结束)：

```
// in Scala
val degreeRatio = inDeg.join(outDeg, Seq("id"))
```

```
  .selectExpr("id", "double(inDegree)/double(outDegree) as degreeRatio")
degreeRatio.orderBy(desc("degreeRatio")).show(10, false)
degreeRatio.orderBy("degreeRatio").show(10, false)

# in Python
degreeRatio = inDeg.join(outDeg, "id")\
  .selectExpr("id", "double(inDegree)/double(outDegree) as degreeRatio")
degreeRatio.orderBy(desc("degreeRatio")).show(10, False)
degreeRatio.orderBy("degreeRatio").show(10, False)
```

查询结果：

```
+----------------------------------------+------------------+
|id                                      |degreeRatio       |
+----------------------------------------+------------------+
|Redwood City Medical Center             |1.5333333333333334|
|San Mateo County Center                 |1.4724409448818898|
...
|Embarcadero at Vallejo                  |1.2201707365495336|
|Market at Sansome                       |1.2173913043478262|
+----------------------------------------+------------------+

+------------------------------+------------------+
|id                            |degreeRatio       |
+------------------------------+------------------+
|Grant Avenue at Columbus Avenue|0.5180520570948782|
|2nd at Folsom                 |0.5909488686085761|
...
|San Francisco City Hall       |0.7928849902534113|
|Palo Alto Caltrain Station    |0.8064516129032258|
+------------------------------+------------------+
```

广度优先搜索

广度优先搜索将搜索我们的图以了解如何根据图中的边连接两组节点。在我们的数据集上，我们可能需要广度优先搜索来查找到不同站点间的最短路径，但该算法也适用于通过 SQL 表达式指定的节点集合。我们可以指定maxPathLength来设定最大的边数量，还可以指定edgeFilter来筛选不符合要求的边，比如非业务时间内的行程。

我们将选择两个接近的站点，因此不会运行太长时间。 但是，当在具有远距离连接的稀疏图上时，可以执行一些有趣的图遍历。请关注一下这些车站（特别是其他城市的车站），看看是否可以获得遥远车站的连接：

```
// in Scala
stationGraph.bfs.fromExpr("id = 'Townsend at 7th'")
  .toExpr("id = 'Spear at Folsom'").maxPathLength(2).run().show(10)
```

```
# in Python
stationGraph.bfs(fromExpr="id = 'Townsend at 7th'",
  toExpr="id = 'Spear at Folsom'", maxPathLength=2).show(10)

+--------------------+--------------------+--------------------+
|                from|                  e0|                  to|
+--------------------+--------------------+--------------------+
|[65,Townsend at 7...|[913371,663,8/31/...|[49,Spear at Fols...|
|[65,Townsend at 7...|[913265,658,8/31/...|[49,Spear at Fols...|
...
|[65,Townsend at 7...|[903375,850,8/24/...|[49,Spear at Fols...|
|[65,Townsend at 7...|[899944,910,8/21/...|[49,Spear at Fols...|
+--------------------+--------------------+--------------------+
```

连通分量

连通分量（connected component）是一个(无向的)子图，它与自身子图有连接，但不连接到其他子图，如图30-5中所示。

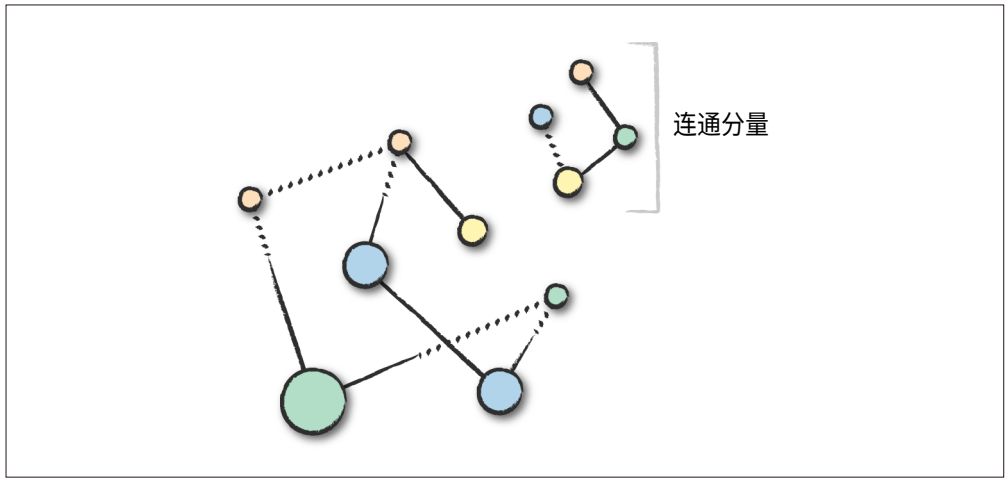


图30-5：连通分量

连通分量算法与我们的单车行程问题不太相关，因为该算法假定为无向图。但我们仍然可以运行该算法，它只是假设边数据没有方向性。事实上，如果我们看看共享单车位置地图，我们可能将得到两个截然不同的连通分量 (见图 30-6)。

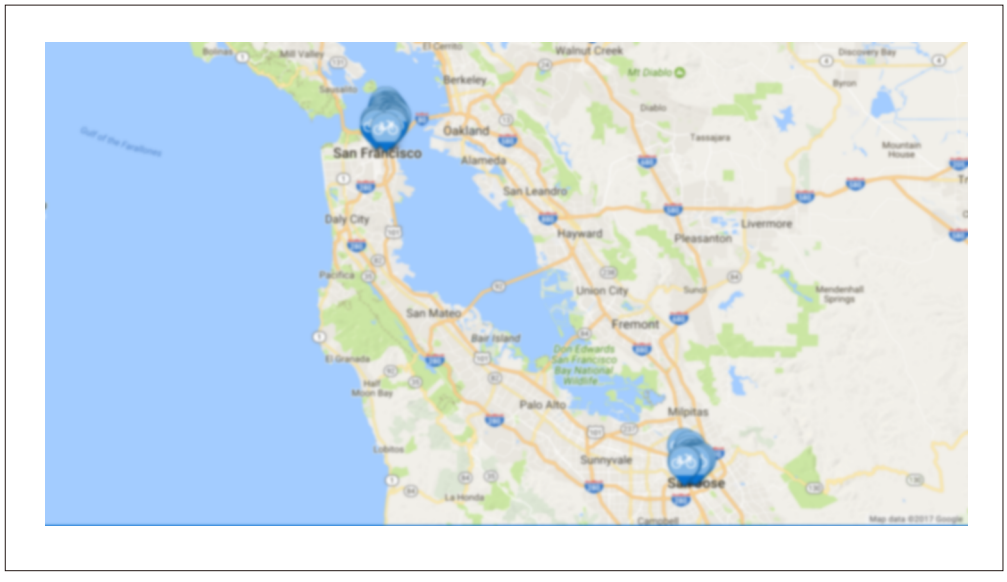


图30-6：湾区共享单车位置地图

要运行此算法，你需要设置一个检查点目录，它将在每次迭代后存储作业的状态。这样如果作业崩溃，你就可以继续重新启动。它可能是目前 GraphFrame中运行代价最大的算法之一，因此预计会运行很长时间。

在本地计算机上运行此算法可能需要做数据采样，就像我们在下面的代码示例中所做的那样 (使用示例中的采样可以帮助你获得结果，而不会使Spark应用程序由于垃圾回收问题而崩溃)：

```
// in Scala
spark.sparkContext.setCheckpointDir("/tmp/checkpoints")

# in Python
spark.sparkContext.setCheckpointDir("/tmp/checkpoints")

// in Scala
val minGraph = GraphFrame(stationVertices, tripEdges.sample(false, 0.1))
val cc = minGraph.connectedComponents.run()

# in Python
minGraph = GraphFrame(stationVertices, tripEdges.sample(False, 0.1))
cc = minGraph.connectedComponents()
```

从这个查询中，我们得到两个连通分量，但不一定是我们期望的。我们的示例可能没有使用所有正确的数据或信息，因此我们可能需要更多的计算资源来进一步查询：

```
// in Scala
cc.where("component != 0").show()

# in Python
cc.where("component != 0").show()
```

```
+----------+-----------------+---------+-----------+---------+------------+-----
|station_id|               id|      lat|       long|dockcount|    landmark|in...
+----------+-----------------+---------+-----------+---------+------------+-----
|        47|   Post at Kearney|37.788975|-122.403452|       19|San Franc...|  ...
|        46|Washington at K...|37.795425|-122.404767|       15|San Franc...|  ...
+----------+-----------------+---------+-----------+---------+------------+-----
```

强连通分量

GraphFrame包括另一种作用于有向图的算法，即强连通分量（strongly connected components），它考虑到了方向性。强连通分量是一个子图，子图中所有顶点对之间都具有连接路径。

```
// in Scala
val scc = minGraph.stronglyConnectedComponents.maxIter(3).run()

# in Python
scc = minGraph.stronglyConnectedComponents(maxIter=3)
scc.groupBy("component").count().show()
```

高级任务

以上只列举了 GraphFrame支持的的一部分算法和功能，GraphFrame库还包括其他一些功能，例如通过消息传递接口（message-passing interface，MPI）编写自己的算法、三角形计数，以及与GraphX之间的转换等。可以在 GraphFrame文档中找到更多信息。

小结

在这一章，我们讲解了 GraphFrame，它是Apache Spark中的执行图分析的库。这里只做了基础讲解，因为这种处理技术不一定是人们在执行高级分析时的首选工具。然而作为一个强大的处理关系数据的工具，它在诸多领域中起着非常重要的作用。下一章将讨论更前沿的功能，特别是深度学习。

第31章

深度学习

深度学习是Spark最激动人心的发展领域之一，因为它能够解决一些以前机器学习方法很难处理的问题，特别是处理非结构化数据 (如图像、音频和文本)。本章将介绍Spark如何与深度学习结合在一起，以及一些与Spark和深度学习一起使用的方法。

由于深度学习仍然是一个新的领域，许多最新的工具都是在外部库中实现的。本章不会把重点放在Spark核心库上，而是那些基于Spark的外部创新库。我们首先讨论几个在Spark上使用深度学习的高级方法，以及讨论何时使用哪一个，之后会讨论支持的库。与往常一样，我们将列举端到端的示例。

要想充分理解本章内容，你至少应该了解深度学习的基本知识以及Spark的基本知识。本书的开头部分提到了一份非常棒的资料，这本书叫做“深度学习”，是由这个领域的一些顶级研究人员编著的。

什么是深度学习？

要定义深度学习，首先应该给出神经网络的概念。神经网络是具有权重和激活函数的一个节点网络，这些节点被组织到若干个堆叠在一起的层上，每个层的部分节点或全部节点连接到网络上的前一层。通过这种层结构，这些简单的函数可以学会识别输入中复杂的信号：一层识别简单线条，下一层识别圆圈和正方形，再下一层识别复杂的纹理，最后一层识别完整的对象。目标是通过调整与每个连接相关的权重和网络中每

个节点的值来训练网络，使某些输入与某些输出相关联。图31-1展示了一个简单的神经网络。

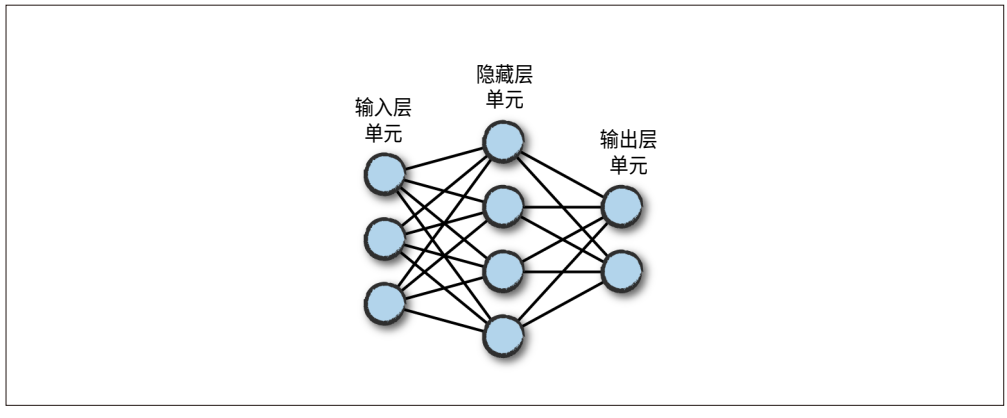


图31-1：神经网络

深度学习或深层神经网络将许多这些层堆叠在一起，组成各种不同的体系结构中。神经网络本身已经发展了数十年，它的流行度也起起伏伏。然而就在最近，由于规模更大的数据集 (例如ImageNet目标识别数据集)、性能强大的硬件（集群和GPU），以及新的训练算法等的出现，我们可以训练规模更大的神经网络，获得的效果比以前的许多机器学习方法都更好。由于增加了更多的数据，传统的机器学习技术通常不能获得稳定的表现效果，可能已经达到了它们的上限。深度学习可以从大量的数据和信息中受益，其输入数据的规模通常比其他机器学习方法的数据集要大几个数量级。深度神经网络现在已经成为计算机视觉、语音处理和一些自然语言处理任务的标准方法，相比于以前的手工模型它们能够学习到更好的特征，它们也还活跃在机器学习的其他领域。Apache Spark在大数据处理和并行计算能力方面的优势使其成为深度学习的非常合适的框架。

研究人员和工程师们在加快这些神经网络的计算方面投入了大量的精力。现在，使用神经网络或深度学习最流行的方法是使用现成的框架，它可能是由研究所或公司实现，本书所写之时正是 TensorFlow、MXNet、Keras 和 PyTorch大受欢迎的时候。这一领域正在迅速发展，所以你需要时刻关注最新动态，看看是否有新的框架出现。

在Spark中使用深度学习

在大多数情况下，无论你的目标应用是什么，在Spark中使用深度学习可用来主要做以下三个任务：

推断（Inference）

使用深度学习最简单方法是采用训练好的模型，然后用Spark并行地将其应用于大型数据集上。例如，你可以使用像 ImageNet 这样训练好的标准图像分类模型，并将其应用于你自己的图像上来识别熊猫、花卉或汽车。许多组织发布大型的基于通用数据集训练好的模型 (例如，用于对象检测的Faster R-CNN 和 YOLO)，因此你可以在你偏好的深度学习框架上提取模型，然后使用Spark并行地使用它。使用PySpark，你可以简单地在`map`函数中调用TensorFlow或PyTorch等框架来实现分布式推断，虽说这样，但还有一些库可以做进一步优化，而不仅仅是在map函数中调用这些库。

特征化（Featurization）与迁移学习（Transfer Learning）

比前面所讲更复杂的是将现有模型用作特征提取器（featurizer），而不再是用它来获得最终的输出。许多深度学习模型因为接受过端到端任务的训练，在较低层次学习到了有用的特征表示。例如，对 ImageNet 数据集进行训练的分类器还将学习所有自然图像中存在的低级特征(如边缘和纹理) 。然后，我们可以使用这些特征为原始数据集未包括的新问题学习模型，此方法称为迁移学习，通常涉及训练好的模型的最后几层，并使用你感兴趣的数据对其进行再训练。如果你没有大量的训练数据，迁移学习也特别有用： 从头开始训练一个完整的网络需要一个像ImageNet这样的包含成千上万张图像的数据集,以避免过拟合，但在许多商业环境中没有这么多数据集。相比之下，迁移学习即使只有几千张图像也可以运行，因为它更新的参数较少。

模型训练（Model training）

Spark还可以用来从头训练一个新的深度学习模型。这里有两种常用的方法。首先，你可以使用Spark集群在多个服务器上并行训练某个单一模型，通过服务器之间的通信进行更新。或者，一些库允许用户并行地训练相似模型的多个实例，以尝试各种模型结构和超参数，从而加速模型搜索和优化的过程。在这两种情况下，使用Spark的深度学习库可以很方便地将数据从 RDD和 DataFrame传递给深度学习算法。最后，即使你不希望并行训练模型，也可以使用这些库从集群中提取数据，并根据TensorFlow等框架的原始数据格式将其导出到单机训练脚本中。

在这三种情况下，深度学习代码通常作为更大应用程序的一部分运行，其中包括提取、转换和加载 (ETL) 步骤来分析输入数据、来自各种源的I/O，以及可能的批处理或流式推断。对于应用程序的其他部分，你可以简单地使用本书前面介绍的DataFrame、RDD 和 MLlib API。Spark的优点之一在于很容易将这些步骤合并到一个并行工作流中。

深度学习库

在这一节中，我们将介绍Spark中一些最受欢迎的深度学习库，我们将描述库的主要应用场景，并提供参考资料或示例。这份深度学习库的清单并不是完整的，因为这一领域发展迅速，建议你访问每个库的官网和Spark最新的文档以获取最新更新。

MLlib对神经网络的支持

Spark的 MLlib 当前提供对一个深度学习算法的原生支持：多层感知机分类器的`ml.classification.MultilayerPerceptronClassifier`类。该类仅限于训练层数相对浅的网络，该网络包含具有sigmoid激活函数的完全连接层和具有softmax激活函数的输出层。 当在现有的基于深度学习的特征提取器上使用迁移学习时，此类对于训练这个分类模型的最后几层非常有用。例如，它可以添加到本章后面描述的深度学习库之上，以快速执行 Keras 和 TensorFlow 模型的迁移学习。

TensorFrame

TensorFrame是一个用于推断和面向迁移学习的库，它便于在Spark DataFrame和TensorFlow 之间传递数据。它支持 Python 和 Scala 接口，提供一个简单而优化的接口来支持数据从 TensorFlow 和Spark之间的传递。特别是，在Spark DataFrame上使用TensorFrame来应用模型通常比直接调用 TensorFlow 模型的一个 Python `map`函数更有效，因为启动时间更少，数据传输更快。TensorFrame在处理推断问题上特别有用，无论是在流处理和批处理情形下，还是在迁移学习中，你可以应用一个训练好的模型来处理原始数据，提起它的特征，然后使用`MultilayerPerceptronClassifier`来学习最后几层，或者甚至在数据上应用简单的逻辑回归或随机森林分类器。

BigDL

BigDL是由英特尔公司针对Apache Spark开发的分布式深度学习框架，它旨在支持大模型的分布式训练以及这些模型的快速应用。BigDL 相比于其他库的一个主要优点是，它主要优化使用CPU的训练而不是 GPU训练，从而使其在基于 CPU 的集群(例如，Apache Hadoop) 上运行效率更高。BigDL 提供高级 API 可以构建神经网络，并在默认情况下自动分布式化所有操作。它还可以训练由Keras DL 库所描述的模型。

TensorFlowOnSpark

TensorFlowOnSpark是一个广泛使用的库，可以用以并行方式在Spark集群上训练TensorFlow 模型。TensorFlow 包括一些基础配置来进行分布式训练，但它仍然需要依靠集群管理器来管理硬件和数据通信，它不与集群管理器或框架之外的分布式I/O层一起使用。TensorFlowOnSpark 在Spark作业中启动 TensorFlow 的现有分布式模式，并自动将数据从Spark RDD 或 DataFrame传递给 TensorFlow 任务。如果你已经知道如何使用TensorFlow的分布式模式，就可以轻松地采用TensorFlowOnSpark在Spark集群内启动你的任务，并将来自任何Spark支持的输入源的数据传递给其他Spark库(例如，DataFrame 转换)进行处理。TensorFlowOnSpark 最初是由Yahoo!开发的，在其他大型组织中也有实际的生产使用。该项目还与Spark的 ML Pipelines API（机器学习流水线 API）进行了集成。

DeepLearning4J

DeepLearning4j是基于Java 和 Scala 的一个开源的、分布式的深度学习项目，支持单节点和分布式训练。相比于Python深度学习框架，其优点之一是它主要为 JVM 设计，使不希望在开发过程中添加 Python 库的人感觉更方便。它包括各种各样的训练算法，既支持CPU，也支持GPU。

Deep Learning Pipelines

Deep Learning Pipelines（深度学习流水线）是Databricks公司的一个开源软件包，它将深度学习功能集成到Spark的ML Pipelines API中。目前，TensorFlow和TensorFlow支持下的Keras深度学习引擎已经支持，它侧重于两个目标：

- 将这些框架并入标准的Spark API (如ML Pipelines和Spark SQL)，使之易于使用。
- 默认情况下将计算分布式化。

例如，Deep Learning Pipelines提供一个`DeepImageFeaturizer`类，在Spark的ML Pipeline API 中充当转换器，允许你使用短短几行代码构建迁移学习流水线 (例如，通过在顶部添加感知器或逻辑回归分类程序)。同样，该库还支持使用 MLlib 的网格搜索和交叉验证 API对多个模型参数进行并行网格搜索。最后，用户可以在Spark SQL的用户自定义函数中使用机器学习模型，使其可供使用SQL或流数据的分析人员使用。

在编写本书的时候，Deep Learning Pipelines正在不断的发展演化，因此我们建议你访问其网站以了解最新的更新。

表 31-1总结了各种深度学习库及其支持的主要应用场景。

表31-1：深度学习库

库	基础深度学习框架	用例
BigDL	BigDL	分布式训练，推断，ML Pipeline集成
DeepLearning4J	DeepLearning4J	推断，迁移学习，分布式训练
Deep Learning Pipelines	TensorFlow，Keras	推断，迁移学习，多模型训练，ML Pipeline，Spark SQL 集成
MLlib 感知器	Spark	分布式训练，ML Pipeline集成
TensorFlowOnSpark	TensorFlow	分布式训练，ML Pipeline集成
TensorFrames	TensorFlow	推断，迁移学习，DataFrame 集成

虽然不同公司有多种方法集成Spark和深度学习库,但目前与MLlib 和 DataFrame集成最好的方式是Deep Learning Pipelines。这个库旨在提高Spark对图像和张量数据的支持 (在Spark 2.3版本中，这些都被支持)，并使所有Spark功能在ML Pipeline API 中可用。它设计友好的API使其成为今天在Spark上实现深度学习的最简单的方法，这是本章余下部分的重点。

Deep Learning Pipelines的一个简单示例

正如我们所描述的，Deep Learning Pipelines通过将流行的深度学习框架和ML Pipeline与Spark SQL 集成在一起，为可扩展的深度学习提供了高级 API。

建立在Spark的ML Pipelines上的Deep Learning Pipelines是为了训练，而基于Spark DataFrame和SQL的流水线是为了部署模型。它包括常用深度学习方法的高级 API，可以用几行代码高效地实现以下这些操作：

- 用Spark DataFrame处理图像。
- 应用大规模深度学习模型处理图像数据和张量数据，这些模型可以是你自己设计的也可以是标准模型。
- 使用常用的预训练模型进行迁移学习。
- 将模型导出为Spark SQL 函数，以便让更多用户都使用深度学习。

- 分布式深度学习超参数通过ML Pipelines进行优化。

Deep Learning Pipelines目前只提供 Python 的 API，它的设计目标是与现有的 Python 深度学习包 (如 TensorFlow 和 Keras) 完美兼容，可以随意调用这些包。

配置

Deep Learning Pipelines是一个Spark包，所以我们将像加载 GraphFrame一样加载它。Deep Learning Pipelines在Spark 2.x 版本上开始支持，可以在此处（*https://spark-packages.org/package/databricks/spark-deep-learning*）找到该软件包。你首先需要安装几个 Python 依赖包，包括TensorFrames、TensorFlow、Keras 和 h5py。确保驱动器节点和工作节点上都已经安装了这些依赖包。

我们将使用来自 TensorFlow retraining tutorial上的flowers数据集。如果你在一个包含若干台机器的集群上运行这个程序，你将需要把下载的数据上传到一个分布式文件系统上。我们在本书的GitHub代码库中包含了该图像数据集的示例。.

图像数据和DataFrame

在Spark中处理图像数据的一个挑战是，很难将图像数据转换成一个DataFrame。Deep Learning Pipelines包括一些工具函数，便于分布式加载和解码图像数据。这是一个更新很快的部分，目前它是ML Pipeline的一部分，Spark 2.3版本中包含基本的图像加载和表示。本章所有示例都会与即将推出的Spark版本兼容：

```
from sparkdl import readImages
img_dir = '/data/deep-learning-images/'
image_df = readImages(img_dir)
```

结果DataFrame 中包含路径以及图像和它的一些元数据：

```
image_df.printSchema()

root
 |-- filePath: string (nullable = false)
 |-- image: struct (nullable = true)
 |    |-- mode: string (nullable = false)
 |    |-- height: integer (nullable = false)
 |    |-- width: integer (nullable = false)
 |    |-- nChannels: integer (nullable = false)
 |    |-- data: binary (nullable = false)
```

迁移学习

现在我们有了数据，可以开始执行一些简单的迁移学习。我们可以利用别人创建的模型，修改它以更好地匹配我们的目标。首先，我们将加载每类花的数据，并创建训练集和测试集：

```
from sparkdl import readImages
from pyspark.sql.functions import lit
tulips_df = readImages(img_dir + "/tulips").withColumn("label", lit(1))
daisy_df = readImages(img_dir + "/daisy").withColumn("label", lit(0))
tulips_train, tulips_test = tulips_df.randomSplit([0.6, 0.4])
daisy_train, daisy_test = daisy_df.randomSplit([0.6, 0.4])
train_df = tulips_train.unionAll(daisy_train)
test_df = tulips_test.unionAll(daisy_test)
```

下一步我们将用一个名为DeepImageFeaturizer的转换器，我们可以利用它的一个名为Inception的预训练模型，它是一个成功用于图像模式识别的神经网络模型。我们使用这个训练好的模型可以很好地识别图像中的各种常见对象和动物，它是Keras 库支持的标准预训练模型之一。然而，这个特殊的神经网络并没有被训练来识别雏菊和玫瑰。因此，我们将使用迁移学习使其适合处理我们的问题： 区分不同的花卉类型。

请注意，我们可以使用ML Pipeline（机器学习流水线）概念，并将它与Deep Learning Pipelines一起使用：DeepImageFeaturizer只是一个 ML转换器，此外我们仅添加了一个逻辑回归模型来做分类，我们也可以用其他分类模型。下面的代码段演示如何添加此模型 (注意，这可能需要一段时间才能完成，因为它是一个相当耗费资源的过程)：

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from sparkdl import DeepImageFeaturizer
featurizer = DeepImageFeaturizer(inputCol="image", outputCol="features",
  modelName="InceptionV3")
lr = LogisticRegression(maxIter=1, regParam=0.05, elasticNetParam=0.3,
  labelCol="label")
p = Pipeline(stages=[featurizer, lr])
p_model = p.fit(train_df)
```

一旦我们训练好了模型，就可以使用第25章中使用的相同的分类评估器，我们可以指定要评估的指标，然后进行评估：

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
tested_df = p_model.transform(test_df)
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Test set accuracy = " + str(evaluator.evaluate(tested_df.select(
  "prediction", "label"))))
```

对于我们的DataFrame 例子，我们可以检查在以前的训练中出错的行和图像：

```
from pyspark.sql.types import DoubleType
from pyspark.sql.functions import expr
# 自定义函数将数值转换为浮点型
def _p1(v):
  return float(v.array[1])
p1 = udf(_p1, DoubleType())
df = tested_df.withColumn("p_1", p1(tested_df.probability))
wrong_df = df.orderBy(expr("abs(p_1 - label)"), ascending=False)
wrong_df.select("filePath", "p_1", "label").limit(10).show()
```

使用大规模深度学习模型

Spark的DataFrame结构可以将深度学习模型应用于大规模数据集，Deep Learning Pipelines提供了一套转换器用于处理大规模的TensorFlow 数据流图和 TensorFlow 支持的 Keras 模型。此外，其他流行的图像模型也可以应用，不需要TensorFlow 或 Keras 代码，由 TensorFrames 库支持的转换器可以有效地处理Spark任务中模型分布和数据分布。

使用流行的模型

有许多标准的深度学习模型用来处理图像数据。如果你的任务与模型适合解决的问题非常相似 (例如，使用 ImageNet 类进行对象识别)，或者仅仅尝试一下，则只需指定模型名称即可使用转换器DeepImagePredictor。Deep Learning Pipelines支持Keras中的各种标准模型，这些模型在其网站上已经列出。下面是使用 DeepImagePredictor 的示例：

```
from sparkdl import readImages, DeepImagePredictor
image_df = readImages(img_dir)
predictor = DeepImagePredictor(
  inputCol="image",
  outputCol="predicted_labels",
  modelName="InceptionV3",
  decodePredictions=True,
  topK=10)
predictions_df = predictor.transform(image_df)
```

注意，使用该基本模型后，predicted_labels列很大概率是“daisy”（雏菊）。从概率值的差异可以看出，神经网络可以识别出两种花型。我们的迁移学习示例显示出，它能够从基本模型开始正确地学习出雏菊（daisy）和郁金香（tulip）之间的差异：

```
df = p_model.transform(image_df)
```

使用自定义的Keras模型

Deep Learning Pipelines还允许我们使用Spark以分布式方式应用 Keras 模型。为此，请查阅 KerasImageFileTransformer上的用户指南，它将加载一个 Keras 模型并将其应用于 DataFrame 列。

使用TensorFlow模型

Deep Learning Pipelies通过与 TensorFlow 的集成，可创建使用 TensorFlow 操作图像的自定义转换器。例如，你可以创建一个转换器来更改图像的大小或修改颜色。为此，请使用 `TFImageTransformer` 类。

将模型集成到SQL函数

另一种选择是将模型集成到 SQL函数，这样的话，任何熟悉SQL的用户能够使用深度学习模型。使用此函数后，生成的 UDF 函数将生成特定模型的输出并写出到一列。例如，可以使用`register KeraImageUDF`类将Inception v3应用于各种图像 ：

```
from keras.applications import InceptionV3
from sparkdl.udf.keras_image_model import registerKerasImageUDF
from keras.applications import InceptionV3
registerKerasImageUDF("my_keras_inception_udf", InceptionV3(weights="imagenet"))
```

这样，深度学习的力量可以被所有Spark用户所享用，而不仅仅是构建模型的专家。

小结

本章讨论了在Spark中使用深度学习的几种常用方法。涵盖了各种可用的库，然后通过一些常见任务的基本示例进行演示。Spark的深度学习部分发展非常迅速，并将继续发展改进，所以建议读者经常查询这些库以及时了解更新！随着时间的推移，本书作者希望保持这一章跟踪到最新的发展。本章讨踪发展前沿。

第VII部分

生态系统

第32章

语言支持：Python(PySpark)和R(SparkR和sparklyr)

本章将介绍 Apache Spark的一些对编程语言方面的支持。本书已经列举了大量的PySpark 例子，第1章宏观上讨论了Spark如何运行其他语言的代码。本章将介绍Spark对以下语言的支持：

- PySpark。
- SparkR。
- Sparklyr。

图32-1展示了这些特定语言的基本体系结构。

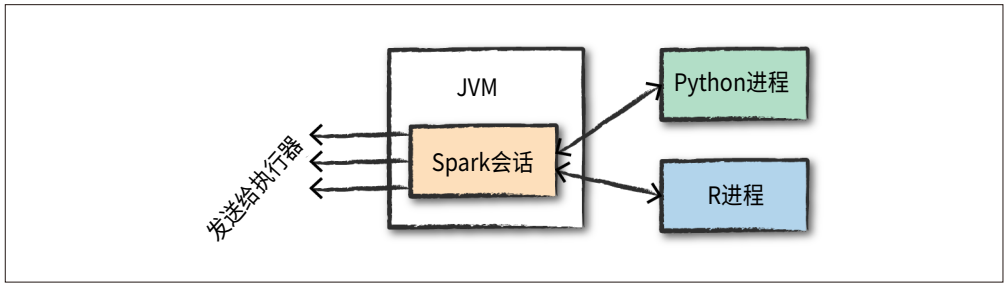


图32-1：Spark的驱动器

现在我们深入地来看看每一种语言。

PySpark

本书中涉及到了很多PySpark例子，事实上，PySpark 与Scala 和 SQL 几乎出现在本书的每一章中。因此这部分比较简短也易于理解，只介绍与Spark本身有关的细节。在第1章中曾经介绍过，Spark 2.2支持安装 PySpark的pip方法，简单使用命令pip install pyspark就能在本机安装。这个方法已被支持，所以可能会有一些 bug 待修复，但当前可以在项目中使用。

PySpark中的一些重要区别

如果你使用的是结构化 API，则代码的运行速度应该与Scala代码一样快，除非在Python中使用 UDF（用户定义函数）。如果使用 UDF，则可能会影响性能。有关为什么会发生这种情况的详细信息，请参阅第6章。

如果你使用的是非结构化 API (特别是 RDD)，则性能将受到影响（但是会获得更好的灵活性）。我们在第12章中讨论了这个原因，根本原因在于，Spark需要额外的开销来将Spark和JVM格式数据与Python数据之间进行转换，这既包括数据也包括函数，这就是所谓的序列化过程。我们并不是说使用RDD是没有意义的，只是在使用RDD时要注意它的性能开销。

集成Pandas

PySpark强大之处还在于它能够跨编程模型工作。例如，通常是采用Spark执行大规模的ETL操作，然后驱动器收集（适合单机大小的）结果数据，之后利用Pandas进行进一步处理。这样，你可以在任务的不同阶段使用最适合的工具来完成任务，Spark处理大数据，而Pandas处理小数据：

```
import pandas as pd
df = pd.DataFrame({"first": range(200), "second": range(50,250)})

sparkDF = spark.createDataFrame(df)

newPDF = sparkDF.toPandas()
newPDF.head()
```

这样，Spark处理大数据和小数据都很轻松。Spark社区将持续专注于改进与其他优秀项目的互操作性，因此Spark和 Python 的集成将继续得到改进。例如，在本书编

写时，社区正致力于支持向量化UDF（SPARK-21190），它添加了一个mapBatches API，支持在Python中将Spark DataFrame作为一系列Pandas DataFrame进行处理，而不需要将每一行都转换为Python对象。该特性已在Spark 2.3中得到支持。

Spark 中的R

本章的余下部分将主要介绍R语言,它是Spark最新官方支持的语言。R语言是统计计算和图像处理领域的语言，它类似于S语言，是由John Chambers（和本书任何作者没有什么关系）和他的同事们在贝尔实验室开发出的一门语言和环境。R 语言已经存在数十年了，在统计学领域和那些从事数值计算的研究中一直很受欢迎。R语言在Spark体系中越来越重要，Spark也为R语言提供了一些简单地分布式计算的开源接口。。

在单机数据分析和高级分析方面备受欢迎的 R 语言是Spark的绝佳补充。将R集成到Spark中有两种实现：SparkR和`sparklyr`，这些软件包采用略有不同的方法来提供类似的功能。SparkR 提供了DataFrame API，它类似于R中的`data.frame`，而`sparklyr`基于很受欢迎的`dplyr`程序包来访问结构化数据。你可以在代码中使用你喜欢的一种，但随着时间的推移，我们预计社区可能会整合这两种实现形成统一的集成包。

这两个软件包我们都会介绍，帮助你选择你喜欢的API。尽管有稍微不同的用户群，但这两个项目基本上都很成熟而且具有良好的社区支持，它们都支持Spark的结构化API和机器学习。我们将在接下来的章节中介绍它们的不同。

SparkR

SparkR是一个基于R语言的软件包(它最初是加州大学伯克利分校、Databricks 和麻省理工学院 CSAIL 之间的合作研究项目)，它提供了类似R API 的 Apache Spark前端接口。除了一些如惰性评估API 语义的不同之外，其概念上类似于 R 内置的data.frame API。SparkR 是Spark官方项目的一部分，并且有Spark项目的充分支持。有关更多信息，请参见 SparkR 的文档。

使用 SparkR的利弊

建议你使用 SparkR 而不是 PySpark 的原因如下：

- 你熟悉 R，并希望用简单的几步就可以利用Spark的能力。
- 你希望利用特定于 R 的功能或库 (如非常好用的`ggplot2`库)，并处理大数据。

R 是一种强大的编程语言，在涉及到某些任务时，它比其他语言有许多优势。然而，它也有缺点，如不能处理分布式数据。SparkR 的目的是填补这一空白，使用户能够成功地以类似于 PySpark 和Pandas的概念方式同时处理小数据集和大数据集。

启动

让我们来看看如何使用 SparkR。当然，你需要在系统上安装 R才能实现本章所讲的。为了启动 shell，请在Spark主文件夹中运行**./bin/sparkR** 以启动SparkR，这将自动为你创建一个 SparkSession。如果要从 RStudio中 运行 SparkR，则执行如下操作：

```
library(SparkR)
spark <- sparkR.session()
```

一旦启动了 shell，我们就可以运行Spark命令。例如，我们可以读取一个 CSV 文件，就像我们在第9 章中看到的：

```
retail.data <- read.df(
  "/data/retail-data/all/",
  "csv",
  header="true",
  inferSchema="true")
print(str(retail.data))
```

我们可以从这个 SparkDataFrame 中读取一些行并将它们转换为标准 R `data.frame`类型：

```
local.retail.data <- take(retail.data, 5)
print(str(local.retail.data))
```

关键概念

我们看到了一些非常基本的代码，现在来重申下关键的概念。首先，SparkR 仍然是Spark。基本上，你在整本书中看到的所有工具都能直接应用于 SparkR。它按照与 PySpark 相同的原则运行，几乎与 PySpark有相同的功能。

如图 32-1所示，有一个网关将 R 进程连接到包含 SparkSession 的 JVM，SparkR 将用户代码转换为可以跨集群的结构化Spark操作。这使得在使用结构化 API 时，它的效率与 Python 和 Scala 不相上下。SparkR不支持RDD 或其他低级API。

虽然 SparkR 不如PySpark 或 Scala流行，但它仍然很受欢迎且在持续改进。对于那些想充分了解Spark并有效利用 SparkR 的用户，我们建议阅读本书的第 I 部分和 第 II 部

分，在其他章节时，请随意尝试使用 SparkR 代替 Python 或 Scala程序。你会发现当你一旦入门，就很容易在不同的语言之间进行转换。

本章的剩下部分将解释 SparkR和“标准”R之间最重要的区别，以帮助你使用SparkR更快地提高工作效率。

我们应该讲解的第一件事是本地类型和Spark类型之间的不同。data.frame类型与Spark版DataFrame的核心区别在于它在内存中可用，并且通常可以直接在特定进程中使用，而SparkDataFrame只是一系列操作的逻辑表示形式。因此，当我们操作data.frame时，我们将立刻看到结果，而在SparkDataFrame上，我们将使用你所了解的转换操作和动作操作的概念来操作数据。

一旦有了SparkDataFrame，我们可以将其转换为data.frame，类似于我们使用Spark读取数据的方式。我们还可以使用以下代码将其转换为本地data.frame (使用本章前面benzhangqiangm“启动”中创建的 SparkDataFrame)：

```
# collect函数将数据从Spark集群加载到本地环境
collect(count(groupBy(retail.data, "country")))
# createDataFrame将data.frame从本地发送到Spark集群
```

此差异对于终端用户有所影响，适用于本地data.frames的某些函数或假设并不适用于Spark。例如，我们不能索引 SparkDataFrame的每一行。此外，我们不能在SparkDataFrame 中更改点值,但可以在本地data.frames中这么做。

函数屏蔽

当用户使用 SparkR 时，一个经常出现的"陷阱" 是，某些功能被 SparkR 屏蔽。引用SparkR 时，收到以下消息：

```
The following objects are masked from 'package：stats'：

cov，filter，lag，na.omit，predict，sd，var，window

The following objects are masked from 'package：base'：

as.data.frame，colnames，...
```

这意味着，如果我们希望调用这些被屏蔽的函数，则需要对调用它们的包进行显式指定，或者至少了解哪个函数会屏蔽另一个函数。“?”有助于确定这些冲突：

```
?na.omit # 根据包加载顺序引用SparkR
?stats::na.omit # 显式引用stats
?SparkR::na.omit # 显式引用SparkR的空值过滤
```

仅适用于 SparkDataFrame的SparkR 函数

函数屏蔽暗示了一个事实，在引入 SparkR 包之后，以前处理对象的函数可能不能用了，这是因为 SparkR 函数只适用于Spark对象。例如，我们不能在标准data.frame上使用sample函数，因为Spark也包含该函数名：

```
sample(mtcars) # 失败报错
```

你必须显式指定base包的sample函数。另外，函数签名在两个函数之间也有所不同，这意味着即使你熟悉某个特定库的语法和参数顺序，在SparkR中该顺序也不一定相同：

```
base::sample(some.r.data.frame) # some.r.data.frame与R的data.frame类型相同
```

数据操作

SparkR 中的数据操作在概念上与其他语言环境中Spark DataFrame API的应用相同。主要区别在于语法，因为我们运行的是R 代码而不是其他语言代码。在本书的其他章节中看到的聚合、过滤和许多功能也可在 R 中使用。在大多数情况下，你可以查看在本书中找到的函数或操作的名称，并通过运行*?<function-name>*来确定它们在 SparkR 中是否可用。其实绝大多数情况是可用的，因为确实支持了很多结构化 SQL 函数：

```
?to_date #to_date DataFrame列操作
```

SQL 的操作也基本相同。我们可以像操作DataFrame一样使用SQL 命令。例如，我们可以在数据集中查询包含“production”一词的所有表：

```
tbls <- sql("SHOW TABLES")

collect(
  select(
    filter(tbls, like(tbls$tableName, "%production%")),
    "tableName",
    "isTemporary"))
```

我们还可以使用流行的`magrittr`包使此代码更具可读性，利用流水线操作符以更实用、更可读的语法把转换操作链接起来：

```
library(magrittr)
```

```
tbls %>%
  filter(like(tbls$tableName, "%production%")) %>%
  select("tableName", "isTemporary") %>%
  collect()
```

数据源

SparkR兼容Spark支持的所有数据源，包括第三方软件包。可以在下面的代码段中看到，只是使用稍微不同的语法来进行设置：

```
retail.data <- read.df(
  "/data/retail-data/all/",
  "csv",
  header="true",
  inferSchema="true")
flight.data <- read.df(
  "/data/flight-data/parquet/2010-summary.parquet",
  "parquet")
```

关于数据源，第9章包含更多的信息。

机器学习

机器学习是 R 语言以及Spark的最重要的组成部分，SparkR 也支持一些Spark MLlib算法。通常情况下，在 Scala 或 Python 中支持这些算法后的1-2个Spark版本后，SparkR也会开始支持这些算法。在Spark 2.1 中，SparkR 支持以下算法：

- `spark.glm or glm`：广义线性模型。
- `spark.survreg`：加速失效时间（AFT）生存分析模型。
- `spark.naiveBayes`：朴素贝叶斯模型。
- `spark.kmeans`：k-means 模型。
- `spark.logit`：逻辑回归模型。
- `spark.isoreg`：保序回归模型。
- `spark.gaussianMixture`：高斯混合模型。
- `spark.lda`：隐含狄利克雷分布（LDA）模型。
- `spark.mlp`：多层感知机分类模型。
- `spark.gbt`：用于回归和分类的梯度提升树模型。
- `spark.randomForest`：用于回归和分类的随机森林模型。

- spark.als：交替最小二乘（ALS）矩阵分解模型。
- spark.kstest：Kolmogorov-Smirnov测试。

SparkR 内部使用 MLlib 来训练模型，这意味着第VI部分大多数内容都与SparkR 用户相关。用户可以调用summary以打印拟合模型的摘要，调用predict对新数据进行预测，write.ml/read.ml以保存/加载拟合的模型。SparkR 支持 R 的一部分用于模型拟合的公式运算符，包括 ~，.，：，+和-。以下是在retail数据集上运行的简单回归示例：

```
model <- spark.glm(retail.data, Quantity ~ UnitPrice + Country,
  family='gaussian')
summary(model)
predict(model, retail.data)

write.ml(model, "/tmp/myModelOutput", overwrite=T)
newModel <- read.ml("/tmp/myModelOutput")
```

虽然并非所有模型都支持详细的摘要输出，如我们在 glm 中看到的那样，但各种模型的API是一致的。有关特定模型或预处理技术的更多信息，请参见第VI部分中相应的章节 .

虽然这些支持与 R 的大量统计算法和分析库集合相比相形见绌，许多用户并不需要处理大数据，并不需要为其机器学习算法的实际训练和使用提供可扩展性支持。用户可以使用Spark在大数据上构建训练集，然后将该数据集收集到本地，以在本地data.frame实现训练。

用户定义的函数

在 SparkR 中，有几种运行用户定义函数的方法。用户定义函数是在本机语言中创建并在支持该语言的服务器上运行的函数。在大多数情况下，这些运行方式与 Python UDF 运行的方法相同，都是通过对JVM函数执行序列化。

有如下几种不同类型的 UDF定义方式：

第一，spark.lapply允许在Spark中运行带有不同参数值的多个函数实例。这是执行网格搜索和结果对比的好方法：

```
families <- c("gaussian", "poisson")
train <- function(family) {
  model <- glm(Sepal.Length ~ Sepal.Width + Species, iris, family = family)
  summary(model)
}
```

```
# 返回模型汇总列表
model.summaries <- spark.lapply(families, train)

# 打印每个模型的汇总信息
print(model.summaries)
```

第二，dapply和dapplyCollect允许你使用自定义代码处理 SparkDataFrame 数据。特别是，这些函数将在 SparkDataFrame 的每个分区，将其转换为执行器内的R data.frame，然后在该分区上调用你的R代码 (表示为R data.frame)。然后，它们将返回结果：dapply的SparkDataFrame或者dapplyCollect.的本地data.frame。

使用dapply将返回一个 SparkDataFrame，你必须指定转换操作后的输出模式，以便Spark了解要返回的数据类型。例如，如果你根据key对数据进行分区，下面的代码将允许你在 SparkDataFrame 中为每个分区训练一个本地 R 模型：

```
df <- withColumnRenamed(createDataFrame(as.data.frame(1:100)), "1:100", "col")
outputSchema <- structType(
  structField("col", "integer"),
  structField("newColumn", "double"))

udfFunc <- function (remote.data.frame) {
  remote.data.frame['newColumn'] = remote.data.frame$col * 2
  remote.data.frame
}
# 输出SparkDataFrame，所以要求数据模式schema
take(dapply(df, udfFunc, outputSchema), 5)
# 将结果聚集起来不需要schema，
# 但是如果结果太大，则导致失败
dapplyCollect(df, udfFunc)
```

最后，gapply和gapplyCollect函数将 UDF 应用于一组数据，其方式类似于dapply。事实上，这两种方法基本上是相同的，只不过一个操作在泛型 SparkDataFrame 上，另一个则适用于分组的 DataFrame。gapply 函数将在每组上应用此函数，并将该组对应的key作为第一个参数传递给你定义的函数，这样就可以确保每个组执行特定的自定义函数：

```
local <- as.data.frame(1:100)
local['groups'] <- c("a", "b")

df <- withColumnRenamed(createDataFrame(local), "1:100", "col")

outputSchema <- structType(
  structField("col", "integer"),
  structField("groups", "string"),
  structField("newColumn", "double"))
```

```
udfFunc <- function (key, remote.data.frame) {
  if (key == "a") {
    remote.data.frame['newColumn'] = remote.data.frame$col * 2
  } else if (key == "b") {
    remote.data.frame['newColumn'] = remote.data.frame$col * 3
  } else if (key == "c") {
    remote.data.frame['newColumn'] = remote.data.frame$col * 4
  }

  remote.data.frame
}
# 输出SparkDataFrame，所以需要shema
take(gapply(df,
            "groups",
            udfFunc,
            outputSchema), 50)

gapplyCollect(df,
              "groups",
              udfFunc)
```

SparkR 作为Spark的一部分将持续发展增强，如果你熟悉R语言和Spark，SparkR将会是一个强有力的工具。

sparklyr

sparklyr是RStudio 团队开发的较新的软件包，它基于流行的dplyr包处理结构化数据。这个软件包与SparkR有明显的不同，它的开发者对Spark和R之间的集成有自己独特的见解。sparklyr去除掉一些Spark概念，如 SparkSession，而采用自己的方法。另外，sparklyr会首先采用R方法，而不是 SparkR的与Python 和 Scala API密切匹配的方法。这种做法可能是为了坚持R语言的原生性，sparklyr是由 RStudio (RStudio是一个受欢迎的 R语言的IDE)团队成员在 R 社区中创建的，而不是由Spark社区创建的。到底是sparklyr好还是SparkR更好，应该完全取决于终端用户的偏好。

简而言之，sparklyr对于熟悉dplyr的R用户更易于使用，其总体功能略少于 SparkR (随着时间的推移可能会发生变化)。sparklyr为Spark提供了完整的dplyr程序，可以是用户能够在本地计算机轻松地运行dplyr代码，同时支持分布式运行。dplyr后端体系的含义是你在本地data.frame对象上使用的函数可以直接以分布式方式应用于分布式的Spark DataFrame上。实质上，扩展不需要修改代码。由于函数同时适用于单节点和分布式 DataFrame，因此该体系结构解决了当今 SparkR 的核心难题之一，函数屏蔽导致的奇怪调试方案。此外，使用sparklyr比使用 SparkR 更容易过渡。与 SparkR 一样，sparklyr是一个不断演化改进的项目，在本书发布时，sparklyr项目将进一步

发展。对于最新的内容，请访问 sparklyr网站。以下部分将提供简单比较，而不会深入研究这个项目。首先介绍一些关于 sparklyr 的实际操作示例，需要做的第一件事是安装软件包：

```
install.packages("sparklyr")
library(sparklyr)
```

关键概念

sparklyr略掉了一些Spark的基本概念，也略掉了一些在本书中讨论过的内容，这可能是因为一般的 R 用户对这些概念并不熟悉(而且可能不是太相关)。例如，使用spark_connect而非SparkSession才能连接到Spark集群：

```
sc <- spark_connect(master = "local")
```

返回的变量是远程dplyr数据源，此连接并不是本书使用的SparkContext(尽管它类似于SparkContext），而是一个纯sparklyr概念，表示与Spark集群的连接。此函数主要是你将如何定义在Spark环境中使用的配置接口，通过此接口，你可以为整个Spark集群指定初始化配置：

```
spark_connect(master = "local", config = spark_config())
```

这可以通过使用 R 中的配置包来指定你希望在Spark集群上的配置。这些详细信息可以在 sparklyr部署文档中查到。

使用此变量，我们可以在本地 R 进程上操作远程Spark数据，因而spark_connect结果与SparkContext大致相同，终端用户可以通过它们执行一些管理员角色的任务。

没有DataFrame

sparklyr不用SparkDataFrame类型的概念，而是使用了与其他 dplyr 数据源类似的表(在Spark内部操作时它们仍然映射为DataFrame）。这将与典型的 R 工作流更匹配，即使可以用 dplyr 和 magrittr 在功能上定义对数据源表中的转换，但是某些Spark的内置函数和 API 可能也无法访问，除非 dplyr 也支持它们。

数据操作

一旦连接到集群，就可以像处理本地dplyr data.frame一样使用dplyr函数来操作它。这种架构选择让熟悉R的用户可以直接用相同的代码执行大规模的数据转换操作，不需要R用户重新学习新的语法或概念。

尽管sparklyr确实改善了 R 终端用户的体验，但代价是削弱了sparklyr提供给用户的功能，因为 R 的概念不一定是Spark的概念。例如，sparklyr不支持使用dapply、gapply和lapply在 SparkR 中创建和应用的用户定义函数。当sparklyr也在持续发展改进，它可能在未来会添加此类功能，但在编写此书时，还没有这些功能。sparklyr的开发进展很迅速，正在不断添加更多新功能，请参阅sparklyr主页以了解更多信息。

执行 SQL

尽管没有完全直接集成Spark，用户可以使用DBI库对集群执行任意的 SQL 代码，这和在前几章中看到的SQL接口几乎相同：

```
library(DBI)
allTables <- dbGetQuery(sc, "SHOW TABLES")
```

此 SQL 接口提供了一个到SparkSession的低级接口。例如，用户可以使用 DBI 接口在Spark集群上设置Spark SQL 的某些属性：

```
setShufflePartitions <- dbGetQuery(sc, "SET spark.sql.shuffle.partitions=10")
```

不幸的是，DBI 和 spark_connect都不提供用于设置特定于Spark属性的接口，但你必须在连接到集群时指定它。

数据源

用户可以使用sparklyr来利用许多Spark支持的数据源。例如，你应该能够使用任意数据源来执行创建数据表的语句。但是，只有 CSV、JSON 和Parquet格式才能使用以下函数定义：

```
spark_write_csv(tbl_name, location)
spark_write_json(tbl_name, location)
spark_write_parquet(tbl_name, location)
```

机器学习

sparklyr还支持我们在前几章中看到的一些核心机器学习算法。(在此编写时) 支持的算法列表包括：

- ml_kmeans：*k*-means聚类。
- ml_linear_regression：线性回归。
- ml_logistic_regression：逻辑回归。

- `ml_survival_regression`：生存回归。
- `ml_generalized_linear_regression`：广义线性回归。
- `ml_decision_tree`：决策树。
- `ml_random_forest`：随机森林。
- `ml_gradient_boosted_trees`：梯度提升树。
- `ml_pca`：主成分分析。
- `ml_naive_bayes`：朴素贝叶斯。
- `ml_multilayer_perceptron`：多层感知器。
- `ml_lda`：隐含狄利克雷分布LDA。
- `ml_one_vs_rest`：one vs rest方法（使用二分类器来完成多类分类任务）。

但是，整个项目在一直在发展，请访问 MLlib 以了解更多信息。

小结

SparkR和`sparklyr`是Spark项目中快速发展的领域，所以请访问它们的网站了解每一项的最新更新。此外，随着新的成员、工具、集成和软件包加入社区，整个Spark项目在继续发展。下一章将讨论Spark社区和其他有用的资源。

第33章

生态系统和社区

Spark最大的优势在于数量庞大的资源、工具和贡献者。在写作本书的时候，Spark代码库有超过1000个贡献者，比其他好多项目多好几个数量级，这是其他项目想都不敢想象的，同时也证明了Spark惊人的社区力量（无论是贡献者和管理人员都有很大的规模）。Spark的发展没有出现减缓的迹象，因为许多大公司和小型企业都在寻求加入社区。这个社区环境孕育了大量的项目，它们补充和扩展了Spark的功能，其中包括官方提供的Spark软件包以及用户可以在Spark上使用的非官方扩展软件包。

Spark软件包

Spark提供一个Spark软件包仓库：Spark Packages，这些软件包在第9章和第24章中曾经讨论过。Spark Packages提供可以很容易与社区分享的Spark应用程序。GraphFrame就是这么一个例子，它基于Spark的结构化 API来执行图分析，使用它要比使用更低级的 (GraphX) API 要容易得多。还有很多其他的软件包，包括许多支持机器学习和深度学习的软件包，它们基于Spark并扩展其功能。

除了这些高级分析包，还有一些解决特定领域问题的软件包。医疗保健和基因组学有许多大数据处理的需求，例如，ADAM 项目对Spark Catalyst引擎做了专门的内部优化，为基因组处理提供可扩展的 API和CLI。另一个软件包Hail是一个开源的、可扩展的框架，用于探索和分析基因组数据，它针对VCF 和其他格式的测序或微阵列数据，提供可扩展的算法进行大数据分析，支持在笔记本电脑上处理GB级数据或在集群上处理TB级数据。

在写作本书的时候，有将近400种不同的软件包可供选择，用户在编译自己项目时可以引用Spark包 (如本书的GitHub库中所示)。你还可以下载编译好的 jar包，并将它们包含在class path中，无需在编译过程中显式地添加。另外，你还可以在运行时通过参数传递给spark-shell 或Spark命令行工具来引用Spark包。

常用包列表

如上所述，有近400个Spark包，你不必引用所有的包，因为你可以在Spark package网站上搜索具体的Spark包。下面是一些比较常用的软件包：

Spark Cassandra Connector

该连接器用于从Cassandra数据库中读数据和向Cassandra数据库写数据。

Spark Redshift Connector

该连接器用于从Redshift数据库中读数据和向Redshift数据库写数据。

Spark bigquery

该连接器用于从Google的BigQuery数据库中读数据和向BigQuery数据库写数据。

Spark Avro

该软件包用户读写 Avro 文件。

Elasticsearch

该软件包用于从 Elasticsearch 中读数据和向Elasticsearch写数据。

Magellan

它基于Spark执行地理空间数据分析。。

GraphFrames

它允许你使用 DataFrame执行图分析。

Spark Deep Learning

允许你结合使用深度学习和Spark。

使用Spark包

有两种核心方法可以在项目中包含Spark包。在 Scala 或 Java 中，可以在构建项目时将其包括为生成依赖项，也可以在运行时指定包 (对于 Python 或 R)。下面回顾一下包含这些信息的方式。

在Scala中

在“*build.sbt*”文件中添加以下resolver来将Spark包作为依赖项。例如，我们可以添加此resolver：

```
// 允许包含Spark软件包
resolvers += "bintray-spark-packages" at
  "https://dl.bintray.com/spark-packages/maven/"
```

现在已经添加好了该行，可以为Spark包引入一个库依赖：

```
libraryDependencies ++= Seq(
...
  // spark 软件包
  "graphframes" % "graphframes" % "0.4.0-spark2.1-s_2.11",

)
```

这将包括 GraphFrames 库。软件包的版本可能稍有不同，但总可以在 "Spark Packages" 网站上找到此信息。

在Python中

在编写此书时，没有明确的方法让 Python 包引用Spark包作为依赖项，必须在运行时设置这些依赖项。

在运行时

我们看到了如何在 Scala 包中指定Spark包，也可以在运行时包含这些包，其实就是在`spark-shell`和`spark-submit`提交作业时包含一个软件包参数。例如，若要包括magellan库，可以执行下面代码：

```
$SPARK_HOME/bin/spark-shell --packages harsha2010：magellan：1.0.4-s_2.11
```

外部包

除了官方的Spark包，还有一些非官方的软件包，它们基于Spark构建或利用Spark的功能。一个典型的例子是流行的梯度提升决策树框架XGBoost，它利用Spark调度各个分区上的分布式训练程序。其中一些是GitHub上的开源项目，可以用搜索引擎发现已有的项目，不必自己编写。

社区

Spark有一个庞大健壮的社区，还有不断增长的用户群体，他们将Spark构建进他们自己的产品中，并撰写使用说明。在本书写作之时，Github 上有超过1000个软件库的贡献者。

Spark官网提供最新的社区信息，包括邮件列表、改进提议和项目提交者名单。这个网站还包括Spark新版本、文档和发行说明等许多资源。

Spark峰会

Spark峰会（Spark Summit）每年不同时间在全球多个地区召开。这是进行Spark相关讨论的重要活动，数以千计的终端用户和开发人员参加这些峰会，了解有关Spark的前沿信息以及各种应用案例，在几天内有数以百计的比赛和培训课程。2016年召开了三次峰会： 纽约 (美东Spark峰会)，旧金山 (美西Spark峰会) 和阿姆斯特丹 (欧洲Spark峰会)。在2017年，在波士顿、旧金山和都柏林召开了Spark峰会。在2018年和以后的将来，还会有更多的峰会召开。可以在Spark峰会网站上查找更多信息。

有数以百计的且免费可用的Spark峰会视频，帮助用户学习应用案例和学习Spark开发，还有使用Spark的战略原则来帮助你充分利用Spark提供的功能。你可以在 Web站点上浏览过去的Spark峰会讲座和视频。

本地见面会

Meetup.com上有很多Spark相关的见面会团体。图33-1展示了*Meetup.com*上Spark相关的见面会的地图。

图33-1：Spark见面会地图

Spark的“官方见面会组”在湾区（由本书的作者之一创建），可以在这里(*https://www.meetup.com/spark-users/*)找到。然而，世界各地有600多个Spark相关的见面会，总计近35万名成员。这些见面会的数量和规模会继续增长，所以一定会在你附近的地区找到一个。

小结

这个简短的章节讨论了Spark提供的非技术性资源。最重要的一点是，Spark最大的财富之一是Spark社区，我们对社区参与Spark的发展感到自豪，并且乐于听到有公司、学术机构和个人基于Spark构建或开发了什么项目。

我们衷心希望你喜欢这本书，我们期待着在Spark峰会上见到你!