小白学院
www.xueai8.com

面向零基础小白的大数据入门教程

Spark 实用教程

基于 Spark 3.1.2

预览版

只要不放弃，蜗牛也可以爬到金字塔的顶端

前　言

大数据分析一直是个热门话题，需要大数据分析的场景也越来越多。Apache Spark 是一个用于快速、通用、大规模数据处理的开源项目。现在，Apache Spark 已经成为一个统一的大数据处理平台，拥有一个快速的统一分析引擎，可用于大数据的批处理、实时流处理、机器学习和图计算。

2009 年，Spark 诞生于伯克利大学 AMP 实验室，最初属于伯克利大学的研究性项目。它于 2010 年被正式开源，于 2013 年被转交给 Apache 软件基金会，并于 2014 年成为 Aparch 基金的顶级项目，整个过程不到五年时间。Apache Spark 诞生以后，迅速发展成为了大数据处理技术中的佼佼者，目前已经成为大数据处理领域炙手可热的技术，其发展势头非常强劲。

自 2010 年首次发布以来，Apache Spark 已经成为最活跃的大数据开源项目之一。如今，Apache Spark 实际上已经是大数据处理、数据科学、机器学习和数据分析工作负载的统一引擎，是从业人员以及希望进入大数据行业人员必须要学习和掌握的大数据技术之一。但是作为大数据的初学者，在学习 Spark 时通常会遇到以下几个难题：

- 缺少面向零基础小白的 Spark 入门教程。
- 缺少系统化的 Spark 大数据教程。
- 现有的 Spark 资料、教程或图书过时陈旧或者碎片化。
- 官方全英文文档难以阅读和理解。
- 缺少必要的数据集、可运行的实验案例及学习平台。
- .....

特别是 Spark 3 发布以后，性能得到了极大的提升，并且增加了对数据湖等下一代大数据技术的支持。为此，既是为了自己能更系统更及时地跟进 Spark 的演进和迭代，另一方面也是为了（感同身受地）解决面向零基础小白学习 Spark（以及其他大数据技术）的入门难度，编写了这一本《Spark 实用教程》。个人以为，本书具有以下几个特点：

- 面向零基础小白，知识点深浅适当，代码完整易懂。
- 内容全面系统，包括架构原理、开发环境及程序部署、流和批计算、图云计算等，并特别包含了 Delta Lake、Iceberg、Hudi 等数据湖内容。
- 版本先进，所有代码均基于 Spark 3.1.2。

个人认为，本书特别适合想要入门 Apache Spark 大数据分析、大数据 OLAP 引擎、流计算的同学、希望系统大数据参考教材的老师以及想要了解最新 Spark 技术应用的从业人员。

当然，因为水平所限，行文以内容难免错误，请大家见谅，并予以反馈，我会在后续的版本重构中不断提升内容质量。

小白学院

本书导学

为了读者能更好地利用本书，特别给出以下学习建议。

❑ 本书只提供电子版。

一般来说，纸质图书的出版周期较长，而大数据技术更新很快。因此为了能及时跟进 Spark 的最新版本，本书只提供电子 版本。（当然，如果有幸有哪位出版社的编辑看上本书，作者本人也特别乐意进一步合作）

❑ 本书正式版本提供书中全部代码。

本书正式版会配套提供书中所有经过测试的 Scala 代码（Python 版本正在编写当中，估计不久就可以和大家见面了），以及数据集和教学视频。最好的学习方法就是动手实践。

❑ 配书依赖 www.xueai8.com 提供的个人大数据学习平台 PBLP（个人大数据学习平台）。

大数据的学习，最大的拦路虎其实是大数据平台。很多初学者花费大量的时间在大数据平台和环境的搭建上，并且在运行时经常出现莫名其妙的问题。即使好不容易自己搭建了一个大数据平台，却与教程、教材、学习资料等的运行环境不匹配或不兼容，学习起来磕磕绊绊。

个人大数据学习平台（PBLP，Personal Big Data Learning Platform）是 www.xueai8.com 依据开源大数据框架搭建好的大数据学习平台（目前为 Apache Hadoop 3.2.2 和 Spark 3.1.2），并搭载了本教程中所有代码和案例所使用的数据集，同时本教程的所有代码、案例、数据集路径、操作截图等，均基于此 PBLP 大数据平台测试和运行。读者可以直接下载此平台，在 VMWare 15+以上打开，即可轻松运行本书中所有代码和案例。

当然，对于有经验的用户，或者已经熟悉了本书内容之后，可以随意修改此平台的安装配置，因为它是完全使用开源大数据组件搭建而成的。

PBLP 下载地址，请大家访问 http://www.xueai8.com 首页相关的下载链接。

❑ 本书提供如下答疑和反馈渠道。

QQ：185314368。如有任何疑问及问题反馈，都可以加此 QQ 进行咨询。

❑ 如何获取本书的最新版本。

本书会根据 Apache Spark 版本的更新迭代而定期更新。如果想要获取本书的最新版本，请访问 http://www.xueai8.com，本书的最新版本会及时在该网站上发布。

❑ 如何获取本书的正式版和配套代码。

如果您拿到的是预览版本，并感觉本书对您有一定的帮助，那么可以从 http://www.xueai8.com 获取本书正式版及配套代码（Scala 或 Python）下载地址。请注意：本书正式版是收费版本。

目　　录

第 1 章　Spark 架构原理与集群搭建

Apache Spark 是一个用于快速、通用、大规模数据处理的开源项目。它类似于 Hadoop 的 MapReduce，但对于执行批处理来说速度更快、更高效。Apache Spark 可以部署在大量廉价的硬件设备上，以创建大数据并处计算集群。

Apache Spark 作为一个用于大数据处理的内存并行计算框架，它利用内存缓存和优化执行来获得更快的性能，并且支持以任何格式读取/写入 Hadoop 数据，同时保证了高容错性和可扩展性。现在，Apache Spark 已经成为一个统一的大数据处理平台，拥有一个快速的统一分析引擎，可用于大数据的批处理、实时流处理、机器学习和图计算。

自 2010 年首次发布以来，Apache Spark 已经成为最活跃的大数据开源项目之一。如今，Apache Spark 实际上已经是大数据处理、数据科学、机器学习和数据分析工作负载的统一引擎。

1.1 Spark 简介

2009 年，Spark 诞生于伯克利大学 AMP 实验室，最初属于伯克利大学的研究性项目。它于 2010 年被正式开源，于 2013 年被转交给 Apache 软件基金会，并于 2014 年成为 Aparch 基金的顶级项目，整个过程不到五年时间。Apache Spark 诞生以后，迅速发展成为了大数据处理技术中的佼佼者，目前已经成为大数据处理领域炙手可热的技术，其发展势头非常强劲。

下图演示了 Spark 的内存计算模型。Spark 一次性从 HDFS 中读取所有的数据并以分布式的方式缓存在计算机集群中各节点的内存中。

下图是 Spark 用于迭代算法的内存数据共享表示：

Spark 与其他分布式计算平台相比有许多独特的优势，例如：

- 用于迭代机器学习和交互式数据分析的更快的执行平台。
- 用于批处理、SQL 查询、实时流处理、图处理和复杂数据分析的单一技术栈。
- 通过隐藏分布式编程的复杂性，提供高级 API 来供用户开发各种分布式应用程序。
- 对各种数据源的无缝支持，如 RDBMS、HBase、Cassandra、Parquet、MongoDB、HDFS、Amazon S3，等等。

Spark 隐藏了编写核心 MapReduce 作业的复杂性，并通过简单的函数调用提供了大部分功能。由于它的简单性，它受到了用户的广泛应用和认同，比如数据科学家、数据工程师、统计学家，以及 R /Python/Scala/Java 开发人员。由于 Spark 采用了内存计算，并采用函数式编程，提供了大量高阶函数和算子，因此它具有以下三个显著特性：速度、易用性和灵活性。

在 2014 年，Spark 赢得了 Daytona GraySort 竞赛，该竞赛是对 100 TB 数据进行排序的行业基准（1 万亿条记录）。来自 Databricks 的提交声称 Spark 能够以比之前的 Hadoop MapReduce 所创造的世界记录的速度快三倍的速度对 100 TB 的数据进行排序，并且使用的资源减少了 10 倍。

Spark 可以连接到许多不同的数据源，包括文件(CSV，JSON，Parquet，AVRO)、MySQL、MongoDB、HBase 和 Cassandra。此外，它还可以连接到特殊用途的引擎和数据源，如 ElasticSearch、Apache Kafka 和 Redis。这些引擎支持 Spark 应用程序中的特定功能，如搜索、流、缓存等。Spark 提供了 DataSource API 以支持到各种数据源(包括自定义数据源)的 Spark 连接。

Spark 提供了四种编程语言接口，分别是 Java、Scala、Python 和 R。因为 Apache Spark 本身是用 Scala 构建的，所以 scala 是首选语言。由于 Spark 内置了对 Scala、Java、R 和 Python 的支持，因此大多数的开发人员和数据工程师能够利用整个 Spark 栈来应用不同的应用场景。

1.2 Spark 技术栈

Spark 提供了一个统一的数据处理引擎，称为 Spark 栈。Spark 栈的基础是其 Core 核心模块（称为 Spark Core）。Spark Core 提供了管理和运行分布式应用程序的所有必要功能，如调度、协调和容错。此外，它还为数据处理提供了强大的通用编程抽象，称为弹性分布式数据集（RDD，resilient distributed datasets）。

在 Spark Core 之上是一个组件集合，其中每个组件都是为特定的数据处理工作而设计的，它们建立在 Spark Core 的强大基础引擎之上的。Spark 技术栈如下图所示：

下面我们分别了解 Spark Core 引擎和各个功能组件。

1.2.1 Spark Core

Spark Core 由两个部分组成：分布式计算基础设施和 RDD 编程抽象。

其中分布式计算基础设施的职责包括：

- 负责集群中多节点上的计算任务的分发、协调和调度
- 处理计算任务失败
- 高效地跨节点传输数据（即数据传输 shuffling）

Spark 的高级用户需要对 Spark 分布式计算基础设施有深入的了解，从而能够有效地设计高性能的 Spark 应用程序。

Spark Core 在某种程度上类似于操作系统的内核。它是通用的执行引擎，它既快速又容错。整个 Spark 生态系统是建立在这个核心引擎之上的。它主要用于工作调度、任务分配和跨 worker 节点的作业监控。此外它还负责内存管理，与各种异构存储系统交互，以及各种其他操作。

Spark Core 的主要编程抽象是弹性分布式数据集（RDD），RDD 是一个不可变的、容错的对象集合，它可以在一个集群中进行分区，因此可以并行操作。本质上，RDD 为 Spark 应用程序开发人员提供了一组 APIs，使这些开发人员能够轻松高效地执行大规模的数据处理，而不必担心数据驻留在集群上的什么位置或处理机器故障。

Spark 可以从各种数据源创建 RDD，如 HDFS、本地文件系统、Amazon S3、其他 RDD、NoSQL 数据存储，等等。RDD 适应性很强，会在失败时自动重建。RDD 是通过惰性并行转换构建的，它们可能被缓存和分区，可能会也可能不会被具体化。

1.2.2 Spark SQL

Spark SQL 是构建在 Spark Core 之上的组件，被设计用来在结构化数据上执行查询、分析操作。因为 Spark SQL 的灵活性、易用性和良好性能，现在它是 Spark 技术栈中最受欢迎、应用最多的组件。

Spark SQL 提供了一种名为 DataFrame 的分布式编程抽象。DataFrame 是分布式二维表集合，类似于 SQL 表或 Python 的 Pandas 库中的 DataFrame。可以从各种的数据源构造 DataFrame，如 Hive、Parquet、JSON、关系型数据库(如 MySQL 等)、以及 Spark RDD。这些数据源可以具有各种模式。

Spark SQL 可以用于不同格式的 ETL 处理，然后进行即席查询分析。Spark SQL 附带一个名为 Catalyst 的优化器框架，它能解析 SQL 查询并自动进行优化以提高效率。Spark SQL 利用 Catalyst 优化器来执行许多分析数据库引擎中常见的优化类型。Spark SQL 的座右铭是“write less code, read less data, and let the optimizer do the hard work”。

1.2.3 Spark streaming 和 Structured Streaming

为了解决企业的数据实时处理需求，Spark 提供了流处理组件，它具有容错能力和可扩展性。Spark 支持实时数据流的实时数据分析。因为具有统一的 Spark 技术栈，所以在 Spark 中可以很容易地将批处理和交互式查询以及流处理结合起来。

目前的 Spark 流处理模块实际上包含两代流处理引擎，分别是第一代的 Spark Streaming 和第二代的 Spark Structured Streaming。其中 Spark Streaming 是基于 RDD 的，而 Spark Structured Streaming 是基于 DataFrame 的。

Spark Streaming 和 Spark Structured Streaming 模块能够以高吞吐量和容错的方式处理来自各种数据源的实时流数据。数据可以从像 Kafka、Flume、Kinesis、Twitter、HDFS 或 TCP 套接字这样的资源中摄取。

在第一代 Spark Streaming 处理引擎中，主要的数据抽象是离散化流（DStream），它通过将输入数据分割成小批量（基于时间间隔）来实现增量流处理模型，该模型可以定期地组合当前的处理状态以产生新的结果。换句话说，一旦传入的数据被分成微批，每批数据都将被视为一个 RDD，并将其复制到集群中，这样它们就可以被作为基本的 RDD 进行处理。通过在 DStreams 上应用一些更高级别的操作，可以产生其他的 DStream。Spark 流的最终结果可以被写回 Spark 所支持的各种数据存储，或者可以被推送到任何仪表盘进行可视化。

从 Spark 2.1 开始，Spark 引入了一个新的可扩展和容错的流处理引擎，称为结构化流（Structured Streaming）。结构化流构建在 Spark SQL 引擎之上，它进一步简化了流处理应用程序开发，处理流计算就像在静态数据上表示批计算一样。随着新的流数据的持续到来，结构化流引擎将自动地、增量地、持续地执行流处理逻辑。结构化流提供的一个新的重要特性是基于事件时间（Event Time）处理输入流数据的能力。在结构化流引擎中还支持端到端的、精确一次性保证。

1.2.4 Spark MLlib

MLlib 是 Spark 栈中内置的机器学习库，它的目标是使机器学习变得可扩展并且更容易。MLlib 提供了执行各种统计分析的必要功能，如相关性、抽样、假设检验等等。该组件还开箱即用的提供了常用的机器学习算法实现，如分类、回归、聚类和协同过滤。

Spark 机器学习库实际上包含两种，分别是基于 RDD 的第一代机器学习库(Spark 0.8 引入)和基于 DataFrame 的第二代机器学习库(Spark 2.0 引入)。目前基于 RDD 的机器学习库已经处理维护模式，因此

本书的机器学习部分基于第二代机器学习库来进行讲解，它受益于 Spark SQL 引擎中的 Catalyst 优化器和 Tungsten 项目，以及这些组件所提供的许多优化。

机器学习工作流程包括收集和预处理数据、构建和部署模型、评估结果和改进模型。在现实世界中，预处理步骤需要付出很大的努力。这些都是典型的多阶段工作流，涉及昂贵的中间读/写操作。通常，这些处理步骤可以在一段时间内多次执行。Spark 机器学习库引入了一个名为 ML 管道的新概念，以简化这些预处理步骤。管道是一个转换序列，其中一个阶段的输出是另一个阶段的输入，形成工作流链。

除了提供超过 50 种常见的机器学习算法之外，Spark MLlib 库提供了一些功能抽象，用于管理和简化许多机器学习模型构建任务，如特征化，用于构建、评估和调优模型的管道，以及模型的持久性（以帮助将模型从开发转移到生产环境）。

1.2.5 Spark GraphX

GraphX 是 Spark 的统一图分析框架。它被设计成一个通用的分布式数据流框架，取代了专门的图处理框架。它具有容错特性，并且利用内存进行计算。

GraphX 是一种嵌入式图处理 API，用于操纵图（例如，社交网络）和执行图并行计算（例如，Google 的 Pregel）。它结合了 Spark 栈上的图并行和数据并行系统的优点，以统一探索性数据分析、迭代图计算和 ETL 处理。它扩展了 RDD 抽象来引入弹性分布式图（Resilient Distributed Graph - RDG），这是一个有向图，具有与每个顶点和边相关联的属性。

GraphX 组件包括一组通用图处理算法，包括 PageRank、K-Core、三角计数、LDA、连接组件、最短路径，等等。

目前的 Spark GraphX 组件是基于 RDD 的，社区正在构建基于 DataFrame（以及其底层的 Catalyst 优化器和 Tungsten 项目）的 DataFrame 版本，称为“GraphFrame”，目前还没有集成到 Spark 发行版中，但已经得到了广泛的应用。本书第 11 章会详细讲解 GraphFrame 的安装和使用。

1.2.6 SparkR

SparkR 项目将 R 的统计分析和机器学习能力与 Spark 的可扩展性集成在一起。它解决了 R 的局限性，即它处理单个机器内存中所需要的大量数据的能力。R 程序现在可以通过 SparkR 在分布式环境中进行扩展。

SparkR 实际上是一个 R 包，它提供了一个 R shell 来利用 Spark 的分布式计算引擎。有了 R 丰富的用于数据分析的内置包，数据科学家可以交互式地分析大型数据集。

注：本书将不涉及 SparkR 的内容。

1.3 Spark 架构原理

在深入了解 Spark 的架构之前，一定要对 Spark 的核心概念和各种核心组件有一个深入的理解。这些核心概念和组件包括：

- Spark 集群
- 资源管理系统
- Spark 应用程序
- Spark Driver
- Spark Executor

1.3.1 Spark 集群和资源管理系统

Spark 本质上是一个分布式系统，设计目的是用来高效、快速地处理海量数据。这个分布式系统通常部署在一个计算机集合上，称为“Spark 集群”。为了高效和智能地管理这个集群，通常依赖于一个资源管理系统，如 Apache YARN 或 Apache Mesos。

资源管理系统内部有两个主要组件：集群管理器（cluster manage）和工作节点（worker）。它有点像主从（master-slave）架构，其中集群管理器充当主节点，工作节点充当集群中的从节点。集群管理器跟踪与 Worker 节点及其当前状态相关的所有信息。集群管理器维护的信息包括：

- Worker 节点的状态(busy/available)
- Worker 节点位置
- Worker 节点内存
- Worker 节点的总 CPU 核数

集群管理器知道 Worker 节点的位置，其内存大小，以及每个 Worker 的 CPU 核数量。集群管理器的主要职责之一是管理 Worker 节点并根据 Worker 节点的可用性和容量为它们分配任务(Task)。每个 Worker 节点都向集群管理器提供自己可用的资源（内存、CPU 等），并负责执行集群管理器分配的任务。如下图所示：

1.3.2 Spark 应用程序

Spark 应用程序也采用了主从架构，其中 Spark Driver 是 master，Spark Executors 是 slave。每一个组件都作为一个独立的 JVM 进程运行在 Spark 集群上。Spark 应用程序由一个且只有一个 Spark Driver 和一个或多个 Spark Executors 组成。

Spark 应用程序由两部分组成，分别是：

- 应用程序数据处理逻辑，使用 Spark API 表示；
- Spark 驱动程序（Spark Driver）。

应用程序数据处理逻辑（即 Task）是用 Java 或 Scala 或 Python 或 R 这几种语言编写的数据处理逻辑代码。它可以简单到几行代码来执行一些数据处理操作，也可以复杂到训练一个大型机器学习模型（这个模型需要多次迭代，可能要运行很多个小时才能完成）。

Spark 驱动程序是运行应用程序 main()函数并创建 SparkSession 的进程。它是 Spark 应用程序的主控制器，负责组织和监控一个 Spark 应用程序的执行。它与集群管理器进行交互，以确定哪台机器来运行数据处理逻辑。Driver 及其子组件（Spark Session 和 Scheduler）负责如下职责：

- 向集群管理器请求内存和 CPU 资源；
- 将应用程序逻辑分解为阶段(stage)和任务（task）；
- 请求集群管理器启动名为 Executor 的进程（在运行 task 的节点上）；
- 向 Executor 发送 Tasks(应用程序数据处理逻辑)，每个 Task 都在一个单独的 CPU Core 上执行；
- 与每个 Executor 协调以收集计算结果并将它们合并在一起。

Spark 应用程序的入口点是通过一个名为 SparkSession 的类来实现的。一旦 Driver 程序被启动之后，它启动并配置 SparkSession 的一个实例。SparkSession 是访问 Spark 运行时的主要接口。SparkSession 对

象连接到一个集群管理器，并提供了设置配置的工具，以及用于表示数据处理逻辑的 API。

除此之外，还需要一个客户端组件。客户端进程负责启动 Driver 程序。客户端进程可以是一个用于运行程序的 spark-submit 脚本，也可以是一个 spark-shell 脚本或一个使用 Spark API 的自定义应用程序。客户端进程为 Spark 程序准备 class path 和所有配置选项，并传递应用程序参数（如果有的话）给运行在 Driver 中的程序。

1.3.3 Spark Driver 和 Executor

每个 Spark 应用程序都有一个 Driver 进程。Spark Driver 包含多个组件，负责将用户代码转换为在集群上执行的实际作业，如下图所示：

Spark Driver 中各个组件的功能如下：

- SparkContext：表示到 Spark 集群的连接，可用于在该集群上创建 RDD、累加器和广播变量。
- DAGScheduler：计算每个作业的 stages 的 DAG，并将它们提交给 TaskScheduler，确定任务的首选位置(基于缓存状态或 shuffle 文件位置)，并找到运行作业的最优调度。
- TaskScheduler：负责将任务（Tasks）发送到集群，运行它们，在出现故障时重试，并减少掉队的情况。
- SchedulerBackend：用于调度系统的后端接口，允许插入不同的实现(Mesos、YARN、单机、本地)。
- BlockManager：提供用于在本地和远程将 block 块放入和检索到各种存储(内存、磁盘和非堆)中的接口。

每个 Spark 应用程序都有一组 Executor 进程。每个 Executor 都是一个 JVM 进程，扮演 slave 角色，专门分配给特定的 Spark 应用程序，执行命令，以任务（Task）的形式执行数据处理逻辑。每个任务在一个单独的 CPU 核心上执行。

Executors 驻留在 Worker 节点上，一旦集群管理器建立连接，就可以直接与 Driver 通信，接受来自 Driver 的任务(Tasks)，执行这些任务，并将结果返回给 Driver。每个 Executor 都有几个并行运行任务的任务槽（Task Slots）。可以将任务槽的数量设置 CPU 核心数量的 2 倍或 3 倍。尽管这些任务槽通常被称为 Spark 中的 CPU Cores，但它们是作为线程实现的，并且不需要与机器上的物理 CPU Cores 数量相对应。另外，每个 Spark Executor 都有一个 Block Manager 组件组成，Block Manager 负责管理数据块。这些块可以缓存 RDD 数据、中间处理的数据或广播数据。当可用内存不足时，它会自动将一些数据块移动到磁盘。Block Manager 还有一个职责是执行跨节点的数据复制。

在启动一个 Spark 应用程序时，可以向资源管理器请求该应用程序所需的 Executor 数量，以及每个 Executor 应该拥有的内存大小和 CPU 核数。要计算出适当数量的 Executor、内存大小和 CPU 数量，需要了解将要处理的数据量、数据处理逻辑的复杂性以及 Spark 应用程序完成处理逻辑所需的持续时间。

1.4 Spark 程序部署模式

Spark Driver 程序的运行有两种基本的方式：集群部署模式和客户端部署模式。

集群部署模式，如下图所示。在这种模式下，Driver 进程作为一个单独的 JVM 进程运行在集群中，集群负责管理其资源(主要是 JVM 堆内存)。

客户端部署模式，如下图所示。在这种模式下，Driver 进程运行在客户端的 JVM 进程中，并与受集群管理的 Executors 进行通信。

选择不同的部署模式将影响如何配置 Spark 和客户端 JVM 的资源需求。通常我们使用客户端部署式，在这种模式下，我们可以在客户端获取并显示作业执行情况。

1.5 安装和配置 Spark 集群

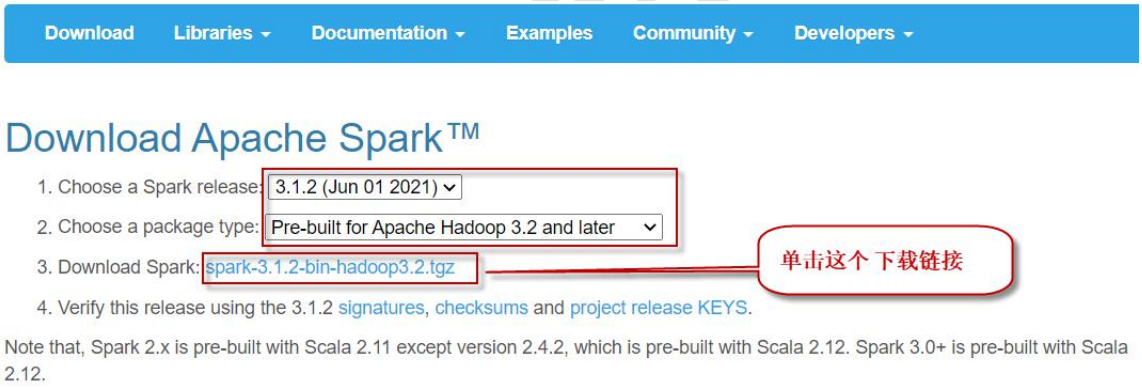
为了学习 Spark，最好在我们自己的计算机上本地安装 Spark。通过这种方式，我们可以轻松地尝试 Spark 特性或使用小型数据集测试数据处理逻辑。

Spark 是用 Scala 编程语言编写的，而 Scala 需要运行在 JVM 上。因此，在安装 Spark 之前，确保已经在自己的计算机上安装了 Java（JDK 8）。

1.5.1 安装 Spark 程序

要在自己的计算机上本地安装 Spark，请按以下步骤操作。

1）下载预先打包的二进制文件到“~/software”目录下，它包含运行 Spark 所需的 JAR 文件。下载地址如下：http://spark.apache.org/downloads.html。目前最新版是 3.1.2。



2）将其解压缩到“~/bigdata/”目录下，并重命名为 spark-3.1.2。执行命令如下：

```
$ cd ~/bigdata
$ tar -zxvf ~/software/spark-3.1.2-bin-hadoop3.2.tgz
$ mv spark-3.1.2-bin-hadoop3.2 spark-3.1.2
```

3）配置环境变量。打开"/etc/profile"文件：

```
$ cd
$ sudo nano /etc/profile
```

在文件最后，添加如下内容：

```
export SPARK_HOME=/home/hduser/bigdata/spark-3.1.2
export PATH=$SPARK_HOME/bin:$PATH
```

保存文件并关闭。

4）执行/etc/profile 文件使得配置生效：

```
$ source /etc/profile
```

1.5.2 了解 Spark 目录结构

查看解压缩后的 Spark 安装目录，会发现其中包含多个目录：

其中几个主要目录作用如下表所示：

目录	描述
bin	包含各种可执行文件，以启动Scala或Python中的Spark shell、提交Spark应用程序和运行Spark示例
conf	包含用于Spark的各种配置文件
data	包含用于各种Spark示例的小示例数据文件
examples	包含所有Spark示例的源代码和二进制文件
jars	包含运行Spark所需的二进制文件
sbin	包含管理Spark集群的可执行文件

1.5.3 配置 Spark 集群

Spark 的配置文件位于 conf 目录下。conf 目录下会默认存在下表中这几个文件，均为 Spark 的配置示例模板文件：

文件名	说明
fairscheduler.xml.template	Hadoop公平调度配置模板文件
log4j.properties.template	Spark Driver节点的日志配置模板文件
metrics.properties.template	Metrics系统性能监控工具的配置模板文件
spark-defaults.conf.template	Spark运行时的属性配置模板文件
spark-env.sh.template	Spark环境变量配置模板文件
workers.template	Spark集群的Worker节点配置模板文件

这些模板文件，均不会被 Spark 读取，需要将.template 后缀去除，Spark 才会读取这些文件。这些配置文件中，在 Spark 集群中主要需要关注的是 spark-env.sh、spark-defaults.conf 和 workers 这四个配置文件。

接下来，我们对 Spark 进行配置，包括其运行环境和集群配置参数。请按以下步骤执行：

1）从模板文件复制一份 spark-env.sh。执行以下命令：

```
$ cd ~/spark-3.1.2/conf/
$ cp spark-env.sh.template spark-env.sh
```

2）编辑 spark-env.sh。执行以下命令：

```
$ nano spark-env.sh
```

加入以下内容，并保存：（请将 JDK 和 Hadoop 修改为你自己安装的版本和路径）

```
export JAVA_HOME=/usr/local/jdk1.8.0_251
export HADOOP_CONF_DIR=/home/hduser/bigdata/hadoop-3.2.2/etc/hadoop
export YARN_CONF_DIR=/home/hduser/bigdata/hadoop-3.2.2/etc/hadoop
export SPARK_HOME=/home/hduser/bigdata/spark-3.1.2
export SPARK_DIST_CLASSPATH=$(/home/hduser/bigdata/hadoop-3.2.2/bin/hadoop classpath)
```

3）编辑 workers。执行以下命令：

```
$ cp workers.template workers
$ nano workers
```

然后将其中的 localhost 删除，将集群中所有 Worker 节点的机器名或 IP 地址填写进去，一个一行。例如，我们的 PBDP 平台机器名是 xueai8，因此 workers 文件内容修改如下：

```
xueai8
```

4）将配置好的 Spark 目录拷贝到集群中其他的节点，放在相同的路径下。

1.5.4 验证 Spark 安装

Spark 配置完成后就可以直接使用，不需要像 Hadoop 运行启动命令。下面我们通过运行 Spark 自带的蒙特卡罗求圆周率 π 值示例，以验证 Spark 是否安装成功。

Spark 支持以本地模式运行 Spark 程序，或者以集群模式运行 Spark 程序。

在本地模式下，直接使用 spark-submit 命令来提交示例程序 jar 包运行即可。命令如下：

```
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[*] \
  ./examples/jars/spark-examples_2.12-3.1.2.jar
```

执行过程如下所示：

执行结果如下图中所示：

或者，也可以 standalone 模式(需要先执行./sbin/start-all.sh 启动 Spark 集群)：

```
$ cd ~/bigdata/spark-3.1.2
$ ./sbin/start-all.sh
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://xueai8:7077 \
  ./examples/jars/spark-examples_2.12-3.1.2.jar
```

执行过程如下所示：

执行结果如下图中所示：

1.6 配置 Spark 历史服务器

当我们提交一个 Spark 应用程序时，会创建一个 SparkContext，它提供了 Spark Web UI 来监视应用程序的执行。监控包括以下内容。

- Spark 使用的配置；
- Spark Jobs、stages 和 tasks 细节；
- DAG 执行；
- Driver 和 Executor 资源利用率；
- 应用程序日志等等。

当应用程序完成处理后，SparkContext 将终止，因此 Web UI 也将终止。如果我们还想看到已经完成的应用程序的监控信息，那么我们就必须配置一个单独的 Spark 历史记录服务器。

Spark History Server（历史记录服务器）是一个用户界面，用于监控已完成的 Spark 应用程序的指标和性能。它是 Spark 的 web UI 的扩展，保存了所有已完成的应用程序的历史(事件日志)及其运行时信息，允许我们稍后检查度量并及时监控应用程序。当我们试图改进应用程序的性能时，历史度量非常有用，我们可以将以前的运行度量与最近的运行度量进行比较。

Spark History server 可以保存事件日志的历史信息，用于如下操作：

- 所有通过 spark-submit 提交的应用程序；
- 通过 REST API 提交的；
- 运行的每一个 spark-shell；
- 通过 NoteBook 提交的作业。

1.6.1 历史服务器配置

为了存储所有提交的应用程序的事件日志，首先，Spark 需要在应用程序运行时收集信息。默认情况下，Spark 不收集事件日志信息。我们可以通过在 spark-defaults.conf 中设置下面的配置来启用它：

- 将配置项 spark.eventLog.enabled 设置为 true 来启用事件日志功能。
- 使用 spark.history.fs.logDirectory 和 spark.eventLog.dir 指定存储事件日志历史的位置。默认位置是 file:///tmp/spark-events。需要提前创建该目录。

请按以下步骤操作。

1）在 Spark 安装目录下，创建存储事件日志历史的文件夹。在终端窗口中，执行如下命令：

```
$ cd ~/bigdata/spark-3.1.2
$ mkdir spark-events
```

2）在 spark-defaults.conf 文件中启用事件日志记录功能。首先从模板文件拷贝一份，并去掉.template 后缀，得到 spark-defaults.conf 文件。命令如下：

```
$ cd ~/bigdata/spark-3.1.2/conf
$ cp spark-defaults.conf.template spark-defaults.conf

# 编辑 spark-defaults.conf
$ nano spark-defaults.conf
```

将下面的配置项添加到文件末尾：

```
# 启用存储事件日志
spark.eventLog.enabled true

# 存储事件日志的位置
spark.eventLog.dir file:///home/hduser/bigdata/spark-3.1.2/spark-events

# 历史服务器读取事件日志的位置
spark.history.fs.logDirectory file:///home/hduser/bigdata/spark-3.1.2/spark-events
```

```
# 日志记录周期
spark.history.fs.update.interval 10s

# 历史服务器端口号
spark.history.ui.port 18080
```

Spark 通过为每个应用程序创建一个子目录来保存运行的每个应用程序的历史，并在该目录中记录与该应用程序相关的事件。

还可以设置如 HDFS 目录这样的位置，以便历史文件可以被历史服务器读取。例如：

```
spark.eventLog.dir hdfs://xueai8:8020/user/spark/spark-events
```

如果想要为 org.apache.spark.deploy.history 的日志记录器（logger）启用 INFO 日志记录级别的话，可以将下面这行配置添加到 conf/log4j.properties 文件中：

```
log4j.logger.org.apache.spark.deploy.history=INFO
```

当启用事件日志记录时，默认的行为是保存所有日志，这会导致存储空间随时间增长。要启用自动清理功能，请编辑 spark-defaults.conf 文件并编辑以下选项：

```
# 设置日志清除周期
spark.history.fs.cleaner.enabled true
spark.history.fs.cleaner.interval 1d
spark.history.fs.cleaner.maxAge 7d
```

对于这些设置，将启用自动清理，每天执行清理，并删除超过 7 天的日志。

1.6.2 启动 Spark 历史服务器

通过在终端窗口中执行以下命令，来启动 Spark 历史服务器。

```
$ SPARK_HOME/sbin/start-history-server.sh
```

如果未明确指定，start-history-server.sh 使用默认配置文件 spark-defaults.conf。另外，它也可以接受--properties-file [propertiesFile]命令行选项，该选项指定带有自定义 Spark 属性的属性文件。

```
$ SPARK_HOME/sbin/start-history-server.sh --properties-file history.properties
```

使用更显式的 spark-class 方法来启动 Spark History Server，则可以更容易地跟踪执行，因为可以看到日志被打印到标准输出（直接输出到终端）。

```
$ SPARK_HOME/bin/spark-class org.apache.spark.deploy.history.HistoryServer
```

如果在 Windows 上运行 Spark，可以通过启动下面的命令来启动历史记录服务器。

```
$ SPARK_HOME/bin/spark-class.cmd org.apache.spark.deploy.history.HistoryServer
```

监控 Spark 应用程序

默认情况下，历史记录服务器监听 18080 端口，可以使用 http://localhost:18080/从浏览器访问它。

在每个 App ID 上单击，可以得到该 Spark 应用程序的 job、stage、task、executor 的详细环境信息。

停止 Spark 历史服务器

通过在终端窗口中执行以下命令，来停止 Spark 历史服务器。

```
$ SPARK_HOME/sbin/stop-history-server.sh
```

使用 Spark 历史服务器，我们可以跟踪所有已完成的应用程序，因此需要启用此功能以保持历史记录。在进行性能调优时，这些指标会派上用场。

附：Spark 历史服务器相关的配置参数描述

- spark.history.updateInterval
  - 默认值：10
  - 以秒为单位，更新日志相关信息的时间间隔。
- spark.history.retainedApplications
  - 默认值：50
  - 在内存中保存 Application 历史记录的个数，如果超过这个值，旧的应用程序信息将被删除，当再次访问已被删除的应用信息时需要重新构建页面。
- spark.history.ui.port
  - 默认值：18080
  - HistoryServer 的 web 端口。
- spark.history.kerberos.enabled
  - 默认值：false
  - 是否使用 kerberos 方式登录访问 HistoryServer，对于持久层位于安全集群的 HDFS 上是有用的，如果设置为 true，就要配置下面的两个属性。
- spark.history.kerberos.principal
  - 默认值：用于 HistoryServer 的 kerberos 主体名称。
- spark.history.kerberos.keytab
  - 用于 HistoryServer 的 kerberos keytab 文件位置。
- spark.history.ui.acls.enable
  - 默认值：false
  - 授权用户查看应用程序信息的时候是否检查 acl。如果启用，只有应用程序所有者和 spark.ui.view.acls 指定的用户可以查看应用程序信息;否则，不做任何检查
- spark.eventLog.enabled
  - 默认值：false
  - 是否记录 Spark 事件，用于应用程序在完成后重构 webUI
- spark.eventLog.dir
  - 默认值：file:///tmp/spark-events
  - 保存日志相关信息的路径，可以是 hdfs://开头的 HDFS 路径，也可以是 file://开头的本地路径，都需要提前创建
- spark.eventLog.compress
  - 默认值：false
  - 是否压缩记录 Spark 事件，前提 spark.eventLog.enabled 为 true，默认使用的是 snappy。

以 spark.history 开头的需要配置在 spark-env.sh 中的 SPARK_HISTORY_OPTS，以 spark.eventLog 开头的配置在 spark-defaults.conf。

1.7 使用 spark-shell 进行交互式分析

在进行数据分析的时候，通常需要进行交互式数据探索和数据分析。为此，Spark 提供了一个交互式的工具 Spark Shell。通过 Spark Shell，用户可以和 Spark 进行实时交互，以进行数据探索、数据清洗和整理以及交互式数据分析等工作。

spark-shell 命令格式如下所示：

```
$ ./bin/spark-shell [options]
```

要查看完整的参数选项列表，可以执行“spark-shell --help”命令，如下：

```
$ spark-shell --help
```

1.7.1 运行模式--master

Spark 的运行模式取决于传递给 SparkContext 的 Master URL 的值。参数选项“--master”表示当前的 Spark Shell 要连接到哪个 master（即告诉 Spark 使用哪种集群类型）。

如果是 local[*]，就是使用本地模式启动 spark-shell，其中，中括号内的星号(*)表示需要使用几个 CPU 核，也就是启动几个线程模拟 Spark 集群。如果不指定，则默认为 local。

当运行 spark-shell 命令时，像下面这样定义这个参数：

```
$ spark-shell --master <master_connection_url>
```

<master_connection_url>根据所使用的集群的类型而变化。Master URL（即--master 参数）的值如下表所示：

1.7.2 启动和退出 spark-shell

以下操作均在终端窗口中进行。

1）启动 Spark Shell 方式一：local 模式

```
$ cd ~/bigdata/spark-3.1.2
$ ./bin/spark-shell
```

然后可以看到如下的启动过程：

从上图中可以看出，Spark Shell 在启动时，已经帮我们创建好了 SparkContext 对象的实例 sc 和 SparkSession 对象的实例 spark，我们可以在 Spark Shell 中直接使用 sc 和 spark 这两个对象。另外，默认情况下，启动的 Spark Shell 采用 local 部署模式。

在创建 SparkContext 对象的实例 sc 之后，它将等待资源。一旦资源可用，sc 将设置内部服务并建立到 Spark 执行环境的连接。

退出 Spark Shell，使用如下命令:

```
scala> :quit
```

2）启动 Spark Shell 方式二：standalone 模式

首先要确保启动了 Spark 集群，

```
$ cd ~/bigdata/spark-3.1.2
$ ./sbin/start-all.sh
```

使用 jps 命令查看启动的进程。如果有 master 和 worker 进程，说明 Spark 集群已经启动。

然后启动 spark-shell，并指定--master spark://xueai8:7077 参数，以 standalone 模式运行：

```
$ ./bin/spark-shell --master spark://xueai8:7077
```

在 Master URL 中指定的 xueai8 是当前的机器名。

1.7.3 Spark Shell 常用命令

可以在 Spark Shell 中键入以下命令，查看 Spark Shell 常用的命令：

```
scala> :help
```

如下图所示：

例如，可以使用":history"命令查看历史操作记录，使用":quit"命令退出 shell 界面。

可以在 Spark Shell 里面输入 scala 代码进行调试：

1.7.4 SparkContext 和 SparkSession

在 Spark 2.0 中引入了 SparkSession 类，以提供与底层 Spark 功能交互的单一入口点。这个类具有用于从非结构化文本文件读取数据的 API，以及各种格式的结构化和二进制数据，包括 JSON、CSV、Parquet、ORC 等。此外，SparkSession 还提供了检索和设置与 spark 相关的配置的功能。

SparkContext 在 Spark 2.0 中，成为了 SparkSession 的一个属性对象。

一旦一个 Spark shell 成功启动，它就会初始化一个 SparkSession 类的实例，名为 spark，以及一个 SparkContext 类的实例，名为 sc。这个 spark 变量和 sc 变量可以在 Spark shell 中直接使用。我们可以使用:type 命令来验证这一点。

```
scala> :type spark
scala> :type sc
```

执行过程如下图所示：

查看当前使用的 Spark 版本号，使用如下命令:

```
scala> spark.version
scala> sc.version
```

执行过程如下图所示：

要查看在 Spark shell 中配置的默认配置，可以访问 Spark 的 conf 变量。下面的命令显示 Spark shell 中默认的配置信息：

```
scala> spark.conf.getAll.foreach(println)
```

执行过程如下图所示：

1.7.5 Spark Web UI

每次初始化 SparkSession 对象时，Spark 都会启动一个 web UI，提供关于 Spark 环境和作业执行统

计信息的信息。web UI 默认端口是 4040，但是如果这个端口已经被占用（例如，被另一个 Spark web UI），Spark 会增加该端口号，直到找到一个空闲的。

在启动一个 Spark Shell 时，将看到与此类似的输出行（除非关闭了 INFO log 消息）：

```
Spark context Web UI available at http://xueai8:4040
```

如下图所示：

注意，可以通过将 spark.ui.enabled 配置参数设为 false 来禁用 Spark web UI。可以用 spark.ui.port 参数来改变它的端口。

示例 Spark web UI 欢迎页面如下图所示。这个 web UI 是从一个 Spark shell 启动的，所以它的名字被设置为 Spark shell，如图右上角所示。

在运行 spark-submit 命令时，也可以使用--conf spark.app.name=<new_name>在命令行上设置程序名称，但不能在启动 Spark shell 时更改应用程序名称。在这种情况下，它总是默认为 Spark shell。

在 Spark web UI 的 Environment 页面，可以查看影响 Spark 应用程序的配置参数的完整列表。如下图所示：

1.8 使用 spark-submit 提交 Spark 应用程序

对于公司大数据的批量处理或周期性数据分析/处理任务，通常采用编写好的 Spark 程序，并通过 spark-submit 指令的方式提交给 Spark 集群进行具体的任务计算，spark-submit 指令可以指定一些向集群申请资源的参数。

Spark 安装包附带有 spark-submit.sh 脚本文件（适用于 Linux、Mac）和 spark-submit.cmd 命令文件（适用于 Windows）。这些脚本可以在$SPARK_HOME/bin 目录下找到。

spark-submit 命令是一个实用程序，通过指定选项和配置向集群中运行或提交 Spark 或 PySpark 应用程序(或 job 作业)，提交的应用程序可以用 Scala、Java 或 Python 编写。spark-submit 命令支持以下功能。

- 在 Yarn、Kubernetes、Mesos、Stand-alone 等不同的集群管理器上提交 Spark 应用。
- 在 client 客户端部署模式或 cluster 集群部署模式下提交 Spark 应用。

spark-submit 命令内部使用 org.apache.spark.deploy.SparkSubmit 类和我们指定的选项和命令行参数。下面是一个带有最常用命令选项的 spark-submit 命令。

```
./bin/spark-submit \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key<=<value> \
  --driver-memory <value>g \
  --executor-memory <value>g \
  --executor-cores <number of cores>  \
  --jars  <comma separated dependencies>
  --class <main-class> \
  <application-jar> \
  [application-arguments]
```

也可以像下面这样提交应用程序，而不使用脚本。

```
./bin/spark-class org.apache.spark.deploy.SparkSubmit <options & arguments>
```

1.8.1 spark-submit 指令的各种参数说明

在 Linux 环境下，可通过“spark-submit --help”命令来了解 spark-submit 指令的各种参数说明。

```
$ cd ~/bigdata/spark-3.1.2
$ ./bin/spark-submit --help
```

spark-submit 的完整语法如下：

```
$ ./bin/spark-submit [options] <app jar | python file> [app options]
```

其中 options 的主要标志参数说明如下：

- --master：指定使用哪个集群管理器来运行应用程序。Spark 目前支持 Yarn、Mesos、Kubernetes、Stand-alone 和 local。
- --deploy-mode：是否要在本地("client")启动驱动程序，或者在集群中("cluster")的一台 worker 机器上。在 client 模式下，驱动程序在调用 spark-submit 的机器上本地运行。客户端模式主要用于交互和调试目的。在 cluster 模式下，驱动程序会被发送到集群的一个 worker 节点上去运行，该节点在应用程序的 Spark Web UI 上显示为 driver。集群模式用于运行生产作业。默认是 client 模式。
- --class：应用程序的主类(带有 main 方法的类),如果运行 Java 或 Scala 程序
- --name：应用程序易读的名称，这将显示在 Spark 的 web UI 上
- --jars：一系列 jar 文件的列表，会被上传并设置到应用程序的 classpath 上。如果你的应用程序依赖于少量的第三方 JAR 包，可以将它们加到这里(逗号分隔)
- --files：使用逗号分隔的文件。通常，这些文件可以来自 resource 文件夹。使用此选项，Spark 将所有这些文件提交到集群。这个标志参数可被用于想要分布到每个节点上的数据文件。(注：使用--files 指定的文件被上传到集群)
- --py-files：一系列文件的列表，会被添加到应用程序的 PYTHONPATH。这可以包括.py、.egg 或.zip 文件。
- --executor-memory：executor 使用的内存数量，以字节为单位。可以指定不同的后缀如"512m"或"15g"。
- --driver-memory：driver 进程所使用的内存数量，以字节为单位。可以指定不同的后缀如"512m"或"15g"。
- --verbose：显示详细信息。例如，将 spark 应用程序使用的所有配置写入日志文件。
- --config：用于指定应用程序配置、shuffle 参数、运行时配置。

关于 dirver 和 executor 资源（cpu 核和内存）配置，我们需要深入了解一下。在提交应用程序时，我们可以指定需要为 driver 和 executor 提供多少内存和核数。下表是这些资源相关的选项说明。

选项	说明
--driver-memory	Spark driver需要使用的内存
--driver-cores	Spark driver需要使用的CPU内核数
--num-executors	要使用的执行器executor总数
--executor-memory	executor进程使用的内存量
--executor-cores	executor进程使用的CPU核数
--total-executor-cores	要使用的执行器executor内核总数

下面这个示例将 Spark 应用程序运行在 Standalone 集群上，采用 cluster 集群部署模式，指定每个 executor 分配 5G 内存和 8 个核。

```
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
```

```
  --master spark://192.168.231.132:7077 \
  --deploy-mode cluster \
  --executor-memory 5G \
  --executor-cores 8 \
  $SPARK_HOME/examples/jars/spark-examples_3.1.2.jar  80
```

下面这个示例将 Spark 应用程序运行在 YARN 集群上，采用 cluster 集群部署模式，指定 driver 进程分配 8G 内存，每个 executor 分配 16G 内存和 2 个核。：

```
$ ./bin/spark-submit \
    --master yarn \
    --deploy-mode cluster \
    --driver-memory 8g \
    --executor-memory 16g \
    --executor-cores 2  \
    --class org.apache.spark.examples.SparkPi \
    $SPARK_HOME/examples/jars/spark-examples_3.1.2.jar  80
```

下面的示例使用集群部署模式将应用程序提交给 yarn 集群管理器，并指定 8g driver 内存，指定每个 executor 有 16g 内存和 2 个内核。

```
$ ./bin/spark-submit \
    --verbose
    --master yarn \
    --deploy-mode cluster \
    --driver-memory 8g \
    --executor-memory 16g \
    --executor-cores 2  \
    --files /path/log4j.properties,/path/file2.conf,/path/file3.json
    --class org.apache.spark.examples.SparkPi \
    $SPARK_HOME/examples/jars/spark-examples_3.1.2.jar  80
```

Spark-submit 使用--config 支持几种配置，这些配置用于指定应用程序配置、shuffle 参数、运行时配置。这些配置对于用 Java、Scala 和 Python 编写的 Spark 应用程序(PySpark)来说是相同的。下表列出了几种常用的配置 key 及其说明。

选项	说明
spark.sql.shuffle.partitions	为宽shuffle转换(join连接和聚合)创建的分区数。
spark.executor.memoryOverhead	在集群模式下为每个executor进程分配的额外内存量，这通常是用于JVM开销的内存。（PySpark不支持）
spark.serializer	org.apache.spark.serializer. JavaSerializer (default) org.apache.spark.serializer.KryoSerializer
spark.sql.files.maxPartitionBytes	读取文件时为每个分区使用的最大字节数。默认128 MB。
spark.dynamicAllocation.enabled	指定是否根据工作负载动态增加或减少executor的数量。默认为true。
spark.dynamicAllocation.minExecutors	启用动态分配时使用的最小executor数量。
spark.dynamicAllocation.maxExecutors	启用动态分配时使用的最大executor数量。
spark.extraJavaOptions	指定JVM选项。

更多配置参数请参考：https://spark.apache.org/docs/latest/configuration.html

请看下面的示例：

```
$ ./bin/spark-submit \
    --master yarn \
    --deploy-mode cluster \
    --conf "spark.sql.shuffle.partitions=20000" \
    --conf "spark.executor.memoryOverhead=5244" \
```

```
    --conf "spark.memory.fraction=0.8" \
    --conf "spark.memory.storageFraction=0.2" \
    --conf "spark.serializer=org.apache.spark.serializer.KryoSerializer" \
    --conf "spark.sql.files.maxPartitionBytes=168435456" \
    --conf "spark.dynamicAllocation.minExecutors=1" \
    --conf "spark.dynamicAllocation.maxExecutors=200" \
    --conf "spark.dynamicAllocation.enabled=true" \
    --conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" \
    --files /path/log4j.properties,/path/file2.conf,/path/file3.json \
    --class org.apache.spark.examples.SparkPi \
    $SPARK_HOME/examples/jars/spark-examples_repace-spark-3.1.2.jar    80
```

也可以在$SPARK_HOME/conf/spark-defaults.conf 文件中将这些配置设置为全局的，以应用于每个 Spark 应用程序。

也可以通过编程方式使用 SparkConf 进行设置。如下代码片段所示：

```
val config = new SparkConf()
config.set("spark.sql.shuffle.partitions","300")
val spark = SparkSession.builder().config(config).master("local[3]")
      .appName("SparkExamples")
      .getOrCreate();

val arrayConfig = spark.sparkContext.getConf.getAll
for (conf <- arrayConfig)
      println(conf._1 +", "+ conf._2)

// 使用 SparkConf 的 get()方法获取特定配置项的值
print("spark.sql.shuffle.partitions ==> " + spark.sparkContext.getConf.get("spark.sql.shuffle.partitions"))
// 显示如下的值：spark.sql.shuffle.partitions ==> 300
```

这几个地方配置的优先顺序是，首先选择代码中的 SparkConf，然后是命令行 spark-submit --config 选项，最后是 spark-defaults.conf 中提到的配置。

无论使用哪种语言，大多数选项都是相同的，但是也有少数选项是特定于某种语言的。

1）用于 Scala 或 Java 程序的参数

例如，要运行用 Scala 或 Java 编写的 Spark 应用程序，需要使用以下额外的选项：

选项	说明
--jars	如果你在一个文件夹中有所有的依赖jar，可以使用spark-submit --jars选项传递所有这些jar。所有的jar文件都应该用逗号分隔。例如，--jars jar1.jar,jar2.jar,jar3.jar。
--packages	用此命令时将处理所有传递依赖项。
--class	指定想运行的Scala或Java类。这应该是带有包名的完全限定名，例如org.apache.spark.examples.SparkPi。

注：使用--jars 和--packages 指定的文件被上传到集群。

例如:

```
$ ./bin/spark-submit \
    --master yarn \
    --deploy-mode cluster \
    --conf "spark.sql.shuffle.partitions=20000" \
    --jars "dependency1.jar,dependency2.jar"
    --class com.xueai8.WordCountExample \
    spark-hello.jar
```

2）用于 PySpark（Python）程序的参数

当想要 spark-submit 一个 PySpark 应用程序时，需要指定想要运行的.py 文件，并为依赖库指定.egg 文件或.zip 文件。

下面是一些特定于 PySpark 应用程序的选项和配置。除此之外，也可以使用上面提到的大多数选项和配置。

PySpark专用配置	说明
--py-files	使用--py-files添加.py、.zip或.egg文件。
--config spark.executor.pyspark.memory	PySpark为每个executor进程使用的内存量。
--config spark.pyspark.driver.python	用于PySpark driver的Python二进制可执行文件。
--config spark.pyspark.python	用于PySpark driver和executor的Python二进制可执行文件。

注：使用--py-files 指定的文件在集群运行应用程序之前被上传到集群。还可以提前上传这些文件，并在 PySpark 应用程序中引用它们。

下面是提交 PySpark 应用程序的示例：

```
$ ./bin/spark-submit \
    --master yarn \
    --deploy-mode cluster \
    wordcount.py
```

下面的示例使用其他 python 文件作为依赖项。

```
$ ./bin/spark-submit \
    --master yarn \
    --deploy-mode cluster \
    --py-files file1.py,file2.py,file3.zip
    wordcount.py
```

1.8.2 提交 SparkPi 程序，计算圆周率π值

Spark 安装包中自带了一个使用蒙特卡罗方法求圆周率π值的程序。下面我们使用 spark-submit 将其提交到 Spark 集群上以 standalone 模式运行，以掌握 spark-submit 提交 Spark 程序的方法。

请按以下步骤操作。

1）打开终端窗口。

2）确保已经启动了 Spark 集群(standalone)模式(启动方式见上一节)

3）进入到 Spark 主目录下，执行以下操作：

```
$ cd ~/bigdata/spark-3.1.2
$ ./bin/spark-submit --master spark://xueai8:7077 --class org.apache.spark.examples.SparkPi
  examples/jars/spark-examples_2.12-3.1.2.jar   10
```

说明：

- --master 参数指定要连接的集群管理器，这里是 standalone 模式。
- --calss 参数指定要执行的主类名称(带包名的全限定名称)。
- 接下来的一个参数是所提交的.jar 包。
- 最后一个参数是需要传入应用程序内的外部参数。本例中的 10 指的是 Spark 程序创建的分区的数量，也就是计算的并行度。计算 PI 的任务被划分为 10 个任务(PI 是通过迭代算法计算的)。

运行结果如下图所示：

. . . . . .

1.8.3 提交 Spark 程序到 YARN 集群上执行

也可以将 Spark 程序运行在 YARN 集群上，由 YARN 来管理集群资源。下面我们使用 spark-submit 将 SparkPi 程序提交到 Spark 集群上以 YARN 模式运行。

请按以下步骤执行。

1）打开终端窗口

2）不需要启动 Spark 集群。启动 YARN 集群：

```
$ start-dfs.sh
$ start-yarn.sh
```

执行过程如下图所示：

3）进入到 Spark 主目录下，执行以下操作：

```
$ cd ~/bigdata/spark-3.1.2
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  ./examples/jars/spark-examples_2.12-3.1.2.jar   10
```

执行过程如下图所示：

执行结果如下图所示：

1.9 小结

- Spark 运行时体系结构的典型组件是客户端进程、driver（驱动程序）和 executors。
- Spark 可以在两种部署模式下运行：客户端部署模式和集群部署模式。这取决于驱动程序(driver)的位置。
- Spark 支持三个集群管理器：Spark 独立集群、YARN 和 Mesos。Spark 本地模式是 Spark 独立集群的特殊情况。集群管理器管理为不同的 Spark 应用程序的 Spark executors（调度）资源。
- Spark 本身在一个应用程序中调度 CPU 和内存资源，以两种可能的模式：FIFO 调度和公平调度。
- 数据本地化意味着 Spark 尝试将任务尽可能地靠近数据位置；存在五个位置水平。
- Spark 通过将内存划分为存储内存、shuffle 内存和堆的其余部分，直接管理对其 executors 可用的内存。
- Spark 可以通过配置文件，使用命令行参数，使用系统环境变量，并及编程方式进行配置。
- Spark web UI 展示了关于运行作业(jobs)、阶段(stages)和任务(tasks)的有用信息。
- Spark local mode 在单个 JVM 中运行整个集群，这对于测试目的非常有用。
- Spark local cluster 模式是在本地机器上运行的全 Spark 独立集群，master 进程在客户端 JVM 中运行。

第 2 章 开发和部署 Spark 应用程序

要开发 Spark 应用程序，业界普遍采用 IntelliJ IDEA 这样的集成开发环境。在使用集成开发环境工具开发 Spark 应用程序时，需要依赖很多 Hadoop 和 Sparkt 等的依赖库，目前企业中普遍采用一些项目管理和构建工具（如 Maven、SBT 等）来管理依赖，以及项目的编译和打包等。Spark 官方推荐的是使用 SBT（Scala Build Tool，Scala 构建工具）来构建 Spark 项目、管理依赖、编译和打包项目。

2.1 使用 IntelliJ IDEA 开发 Spark SBT 应用程序

IntelliJ IDEA，一般简称 IDEA，是 Java 语言开发的集成环境。IntelliJ 在业界被公认为最好的 Java 开发工具之一，尤其在智能代码助手、代码自动提示、重构、J2EE 支持、Ant、JUnit、CVS 整合、代码审查、创新的 GUI 设计等方面的功能可以说是超常的。IDEA 是 JetBrains 公司的产品，这家公司总部位于捷克共和国的首都布拉格，开发人员以严谨著称的东欧程序员为主。

下载地址：https://www.jetbrains.com/idea/download/#section=windows

IDEA 每个版本提供 Community 和 Ultimate 两个版本，如下图所示，其中 Community 是完全免费的，而 Ultimate 版本可以使用 30 天，过这段时间后需要收费。开发 Spark 应用程序，下载一个最新的 Community 版本即可。

2.1.1 安装 IntelliJ IDEA

1.首先在官网下载 IDEA https://www.jetbrains.com/idea/download/#section=windows。
双击 exe 文件，安装。

2.傻瓜式安装，一路 Next

接着 Next，选择安装路径。路径要记牢哦，后面会用到。

3.耐心等待安装

OK，到这里就安装完成了。

安装完成后，我们就可以启动 IntelliJ IDEA 了。可以通过两种方式启动：

- 到 IntelliJ IDEA 安装所在目录下，进入 bin 目录双击 idea.sh 启动 IntelliJ IDEA；
- 在命令行终端中，进入$IDEA_HOME/bin 目录，输入./idea.sh 进行启动

2.1.2 配置 IntelliJ IDEA Scala 环境

在 IDEA 中开发 scala 程序（以及 spark 程序）需要安装 scala 插件。IDEA 默认情况下并没有安装 Scala 插件，需要手动进行安装。安装过程并不复杂，下面将演示如何进行安装。

1）在 IDEA 启动界面上选择“Configure-->Plugins”选项(或者在项目界面，选择“File > Settings... > Plugins”)，然后弹出插件管理界面，在该界面上列出了所有安装好的插件。由于 Scala 插件没有安装，需要点击“Install JetBrains plugins”进行安装，如下图所示：

2）待安装的插件很多，可以通过查询或者字母顺序找到 Scala 插件，选择插件后在界面的右侧出现该插件的详细信息，点击绿色按钮“Install plugin”安装插件，如下图所示：

3）安装过程将出现安装进度界面，通过该界面了解插件安装进度，如下图所示：

4）安装完成后，将看到一个按钮，用于重新启动 IntelliJ IDE。继续并点击它来重启我们的 IntelliJ：

5）最后，重启 IntelliJ，让插件生效。

现在我们已经安装了 IntelliJ IDEA、Scala 插件和 SBT，可以开始构建 Spark 程序了。

2.1.3 创建 IntelliJ SBT 项目

SBT 之于 Scala 就像 Maven 之于 Java，用于管理项目依赖，构建项目。使用 IntelliJ IDEA 开发 Spark 应用程序，使用 SBT 作为构建管理器，这是也官方推荐的开发方式(安装 Scala 插件时,该 Scala 插件自带 SBT 工具)。

请按以下步骤，使用 IntelliJ IDEA 创建一个新的 Spark SBT 项目。

1）启动 IntelliJ IDEA，在开始界面中，选择【create new project】，创建一个新项目。如下图所示：

2）接下来，依次选择【Scala】|【sbt】，然后单击【Next】按钮。如下图所示：

3）在接下来的向导窗口中，将项目命名为“HelloSpark”，指定项目存放的位置，并选择合适的 sbt 和 Scala 版本。如下图所示：

4）单击【Finish】按钮继续。IntelliJ 应该创建一个具有默认目录结构的新项目。生成所需的所有文件夹可能需要一到两分钟，最终的文件夹结构应该是这样的:

让我们了解一下生成的项目结构，说明如下：

- .idea：这些是 IntelliJ 配置文件。
- project：编译期间使用的文件。例如，在 build.properties 中指定编译项目时使用的 SBT 版本。
- src：源代码。大多数代码应该放在 main/scala 目录下。测试脚本放在 test/scala 文件夹下。
- target：当编译项目时，会生成这个文件夹。
- build.sbt：sbt 配置文件。使用该文件导入第三方库和文档。

2.1.4 配置 SBT 构建文件

在开始编写 Spark 应用程序之前，我们需要将 Spark 库和文档导入 IntelliJ，这需要在 build.sbt 文件中进行配置。编辑 build.sbt 文件，创建工程时生成的初始内容如下：

```
name := "HelloSpark"

version := "1.0"

scalaVersion := "2.12.14"
```

向其中添加 Spark Core 和 Spark SQL 依赖，内容如下：

```
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "3.1.2",
  "org.apache.spark" %% "spark-sql" % "3.1.2"
)
```

说明：SBT 依赖库的内容和格式，可以到 mvn repository 查询。

保存文件后，IntelliJ 将自动导入运行 Spark 所需的库和文档，因此要确保你的电脑是可以连网的。结果如下所示：

2.1.5 准备数据文件

接下来，我们将构建一个简单的 Spark 应用程序，用来对莎士比亚文集（shakespeare.txt）执行单词计数任务。

我们需要在两个地方保存 shakespeare.txt 数据集。一个在项目中用于本地系统测试，另一个在 HDFS (Hadoop 分布式文件系统)中用于集群测试。

将 PBDP 大数据平台中的/home/hduser/data/spark/shakespeare.txt 文件上传到 HDFS：

1）确保已经启动了 HDFS；

2）在终端窗口中，执行以下命令，上传文件到 HDFS 上：

```
$ cd /home/hduser/data/spark
$ hdfs dfs -put shakespeare.txt /data/spark/
```

2.1.6 创建 Spark 应用程序

现在，我们准备开始编写 Spark 应用程序。

1）回到项目中，在 src/main 下创建一个名为 resources 的文件夹(如果它不存在的话)，并将 shakespear .txt 拷贝到该文件夹下。

2）接下来，在 src/main/scala 下创建一个新类。右击“scala > New > Scala Class”，如下图所示：

3)接下来，IDE 会询问是创建一个 class、object 还是 trait。选择 object，将该文件命名为“HelloWord”，如下图所示：

4）编辑该源文件，代码如下：

```
object HelloWord {
```

```
  def main(args: Array[String]): Unit ={
    println("Hello World!")
  }
}
```

5）运行程序。在文件上任何位置单击右键，在弹出的环境菜单中，选择【Run 'HelloWord'】。

6）如果一切正确，该 IDE 应该在下方的控制台窗口输出“HelloWorld!”。如下图所示：

7）现在我们知道环境已经正确设置，接下来用以下代码替换文件中原来的内容:

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.SparkSession

object HelloWord {

  def main(args: Array[String]) {

    // 在 windows 下开发时设置
    System.setProperty("HADOOP_USER_NAME", "hduser")

    // 创建一个 SparkContext 来初始化 Spark
    // Spark 2.0 以前的用法
    // val conf = new SparkConf().setMaster("local").setAppName("Word Count")
    // val sc = new SparkContext(conf)

    // Spark 2.0 以后的用法
    val spark = SparkSession.builder().master("local[*]").appName("Word Count").getOrCreate()
    val sc = spark.sparkContext

    // 将文本加载到 Spark RDD 中，它是文本中每一行的分布式表示
    val file = "src/main/resources/shakespeare.txt"
    val textFile = sc.textFile(file)

    // transformation 转换
    val counts = textFile.flatMap(line => line.split(" "))
                         .map(word => (word, 1))
                         .reduceByKey(_ + _)

    counts.collect.foreach(println)
    System.out.println("全部单词: " + counts.count());

    // 将单词计数结果保存到指定文件中
    val output = "tmp/shakespeareWordCount"
    counts.saveAsTextFile(output)
  }

}
```

8）和刚才一样，右键单击，并选择【Run ' HelloScala'】来运行程序。这将运行 Spark 作业并打印莎士比亚作品中出现的每个单词的频率，预期输出如下:

9）此外，如果浏览指定的目录：

```
counts.saveAsTextFile ("tmp/shakespeareWordCount");
```

将找到程序的输出：

注意我们在代码中设置了下面这一行代码:

```
.master("local[*]")
```

这告诉 Spark 使用这台计算机在本地运行，而不是在分布式模式下运行。要在多台机器上运行 Spark，我们需要更改此值。(稍后我们将看到如何更改)

2.1.7 部署分布式 Spark 应用程序

现在我们已经了解了如何在 IDE 中直接部署应用程序。这是一种快速构建和测试应用程序的好方法。但是在生产环境中，Spark 通常会处理存储在 HDFS 等分布式文件系统中的数据。Spark 通常也以集群模式运行(即分布在许多机器上)。

接下来，我们修改代码，使其能部署到 Spark 分布式集群上运行。请按以下步骤操作。

1）修改源代码，如下面代码中粗体部分所示：

```
import org.apache.spark.{SparkConf, SparkContext}

object HelloScala {

  def main(args: Array[String]) {

    // 创建一个 SparkContext 来初始化 Spark
    val spark = SparkSession.builder().appName("Word Count").getOrCreate()
    val sc = spark.sparkContext

    // 将文本加载到 Spark RDD 中，它是文本中每一行的分布式表示
    val file = "hdfs://localhost:8020/data/spark_demo/shakespeare.txt"
    val textFile = sc.textFile(file)

    // word count
    val counts = textFile.flatMap(line => line.split(" "))
                         .map(word => (word, 1))
                         .reduceByKey(_ + _)

    counts.collect.foreach(println)
    System.out.println("全部单词: " + counts.count());

    // 将单词计数结果保存到指定输出目录中
    val output = "hdfs://localhost:8020/data/spark_demo/shakespeareWordCount"
    counts.saveAsTextFile(output)
  }

}
```

这告诉 Spark 读写 HDFS，而不是本地。

2）创建 JAR 文件

我们将把这些代码打包到一个已编译的 jar 文件中，该文件可以部署在 Spark 集群上。

在 IntelliJ IDEA 菜单栏选择【Tools】|【Start SBT Shell】，在编辑窗口下方打开 sbt shell 交互窗口，然后就可以应用 sbt 的 clean、compile、package 等命令进行操作。这里我们执行打包命令：

```
> package
```

打包过程如下图所示：

这将在项目的“target/scala-2.11”下创建一个名为“hellospark_2.11-1.0”的编译过的 jar 文件。

3）将该 jar 包提交到 Spark 集群上执行。使用 spark-submit 运行我们的代码。

```
$ cd ~/bigdata/spark-3.1.2
$ ./bin/spark-submit --class HelloWord --master local[*] ./hellospark_2.11-1.0.jar
```

提交任务时需要指定主类、要运行的 jar 包和运行模式(本地或集群):

4）控制台应该打印莎士比亚作品中出现的每个单词的频率，如下所示:

```
...
(comutual,1)
(ban-dogs,1)
(rut-time,1)
(ORLANDO],4)
(Deceitful,1)
(commits,3)
(GENTLEWOMAN,4)
(honors,10)
(returnest,1)
(topp'd?,1)
(compass?,1)
(toothache?,1)
(miserably,1)
(hen?,1)
(luck?,2)
(call'd,162)
(lecherous,2)
...
```

5）此外，可通过 HDFS 或 Web UI 查看输出文件的内容：

```
$ hdfs dfs -cat /data/spark_demo/shakespeareWordCount/part-00000
```

2.1.8 远程调试 Spark 程序

在本节中，我们将学习如何将正在运行的 Spark 程序连接到调试器，调试器允许我们设置断点并逐行执行代码。在直接从 IDE 运行时，调试 Spark 和其他任何程序一样，但是调试远程集群需要一些配置。

1）在计划提交 Spark 作业的机器上，从终端运行以下代码:

```
export SPARK_SUBMIT_OPTS=-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005
```

2）提交 Spark job 作业，运行时会出现程序挂起，监听端口中。如下：

```
spark-submit --class HelloWord --master local[*] --driver-java-options    bigdata/hellospark_2.11-1.0.jar
```

3）在 IntelliJ IDEA 中配置 remote debug：在 IntelliJ 中选择菜单项【Run】|【Edit Configurations】，编辑运行配置信息。如下：

4）然后单击左上角的+按钮，选择面板左侧的 Remote 项，增加一个新的远程配置。在面板右侧，使用主机 ip 地址填写 Host（Port 字段默认为 5005，不需要修改），这将允许在端口 5005 上关联调试器。需要确保端口 5005 能够接收入站连接。

5）从 IDE 中 debug 此调试配置，调试器将附加此调试配置，并且远程 Spark 程序将在断点处停止。还可以检查程序中活动变量的值。在试图确定代码中的 bug 时，这是非常宝贵的。

2.2 使用 IntelliJ IDEA 开发 Spark Maven 应用程序

虽然 Spark 官方推荐使用 SBT 来构建 Spark 项目，但仍然有很多人习惯使用 Maven。接下来我们演示如何使用 IntelliJ IDEA 开发 Spark Maven 应用程序。

集成开发环境 IntelliJ IDEA 的安装，请参考上一节中的安装步骤。请确保已经安装了 JDK 8 和 Scala 插件。

2.2.1 创建 IntelliJ Maven 项目

首先启动 IntelliJ IDEA，依次选择 File > New > Project > Maven，从模板创建 Scala Maven 项目。如下图所示：

在接下来的向导窗口中，命名项目如下：

```
GroupId: com.xueai8
ArtifactId: sparkexamples
Version: 1.0-SNAPSHOT
```

然后单击 Next 按钮继续。如下图所示：

接下来，设置 Maven 的 settings 文件和 repository 位置。选择默认就好了。

最后，选择项目名称和位置。这些字段应该自动填充，所以使用默认值就好了。然后按下【Finish】键，开始生成项目。

IntelliJ 应该创建一个具有默认目录结构的新项目。生成所有文件夹可能需要一到两分钟。

让我们了解一下项目结构：

- .idea：这些是 IntelliJ 配置文件。
- src/main/scala：源代码。源代码应该位于此目录下。而 test 文件夹应该保留用于测试脚本。
- target：当对项目编译时会产生此目录。
- pom.xml：Maven 配置文件。使用这个文件导入第三方库和文档。

2.2.2 验证 SDK 安装和配置

在继续之前，让我们先验证几个 IntelliJ 设置：

- 验证导入 Maven 项目是否自动打开。
  - File > Settings... > Build, Execution, Deployment > Build Tools > Maven > Importing

- 验证项目的 Project SDK 和 Project language level 设置为 Java 版本：
  - File > Project Structure... > Project

- 验证模块的 Language level 设置为 Java 版本：
  - File > Project Structure... > Modules，将右侧的的 Language level 值设置如下图：

- 验证 Global Libraries 已经配置了 Scala SDK
  - File > Project Structure... > Global Libraries，点击右侧的加号（+），浏览并选择本地安装的 Scala SDK 安装目录，设置如下图：

2.2.3 项目依赖和管理配置

在开始编写 Spark 应用程序之前，我们需要将 Spark 库和文档导入 IntelliJ。要导入 Spark 库，我们将使用依赖管理器 Maven。打开 pom.xml 配置文件，按以下步骤编辑。

1）首先，将 Scala 版本修改为最新的版本，我使用的是 2.12.14：

```
<properties>
    <scala.version>2.12.14</scala.version>
 </properties>
```

2）删除如下部分：

```
      <plugin>
        <groupId>org.scala-tools</groupId>
        <artifactId>maven-scala-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <scalaVersion>${scala.version}</scalaVersion>
          <args>
            <arg>-target:jvm-1.8</arg>
          </args>
        </configuration>
      </plugin>
```

3）删除不必要的文件。

从项目结构中，删除如下部分：

- 删除 src/test
- 删除 src/main/scala/org.xueai8.App

4）添加 Spark 依赖到 Maven pom.xml 文件中。

```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.12</artifactId>
      <version>3.1.2</version>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.12</artifactId>
      <version>3.1.2</version>
      <scope>compile</scope>
    </dependency>
```

5）最终的 pom.xml 文件如下所示：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.xueai8</groupId>
  <artifactId>sparkexamples</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <scala.version>2.12.11</scala.version>
  </properties>

  <dependencies>
    <!--scala-->
    <dependency>
      <groupId>org.scala-lang</groupId>
      <artifactId>scala-library</artifactId>
      <version>${scala.version}</version>
    </dependency>

    <!--spark-->
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.12</artifactId>
      <version>3.1.2</version>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
```

```
        <artifactId>spark-sql_2.12</artifactId>
        <version>3.1.2</version>
    </dependency>
  </dependencies>

  <build>
    <sourceDirectory>src/main/scala</sourceDirectory>
    <testSourceDirectory>src/test/scala</testSourceDirectory>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

保存文件后，IntelliJ 将自动导入运行 Spark 所需的库和文档。这个过程有可能会持续较长时间。

2.2.4 测试程序

接下来，我们就可以开发基于 Maven 的 Spark 程序。

1）创建源程序

在 src/main/scala 上单击右键，创建一个 Scala Object，命名为 HelloWorld。编辑代码如下：

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
  }
}
```

有些时候，pom.xml 中的依赖项不会自动加载，因此，需要重新导入依赖项或重启 IntelliJ。

2）然后在文件任何空白地方，单击右键，在弹出的菜单中选择【Run 'MyApp'】，如果得到如下的输出结果，则一切 OK！

```
Hello World!
```

2.2.5 项目编译和打包

对于 Maven 项目，可以简单地在终端窗口运行如下的打包命令，就会自动编译并打 jar 包：

```
$ mvn clean package
```

如下图所示：

如果一切顺利，会在项目中生成 target 目录，打好的 jar 包就位于此目录下。如下图所示：

2.3 使用 Java 开发 Spark 应用程序

除了提供 Scala 和 Python API 接口，Spark 还支持使用 Java 语言来开发 Spark 应用程序。在这一节中，我们将使用 IntelliJ IDEA 作为开发工具，使用 Java 作为开发语言，并使用 Maven 作为构建管理器。在本教程的最后，将了解如何设置 IntelliJ、如何使用 Maven 管理依赖项、如何将 Spark 应用程序打包和部署到集群，以及如何将实时程序连接到调试器。

2.3.1 创建一个新的 IntelliJ 项目

启动 IntelliJ IDEA，选择 File > New > Project > Maven，单击 Next 按钮。如下图所示：

在接下来的向导窗口中，命名项目如下：

- GroupId: xlw
- ArtifactId: myspark
- Version: 1.0-SNAPSHOT

然后单击 Next 按钮继续。

最后，选择项目名称和位置。这些字段应该自动填充，所以让我们运行默认值:

IntelliJ 应该创建一个具有默认目录结构的新项目。生成所有文件夹可能需要一到两分钟。

让我们了解一下项目结构：

- .idea: 这些是 IntelliJ 配置文件。
- src: 源代码。大多数的代码应该位于 main 目录下。而 test 文件夹应该保留用于测试脚本。
- target: 当对项目编译时会产生此目录。
- pom.xml: Maven 配置文件。使用这个文件导入第三方库和文档。

2.3.2 验证 SDK 安装和配置

在继续之前，让我们先验证几个 IntelliJ 设置：

- 验证导入 Maven 项目是否自动打开。
  - File > Settings... > Build, Execution, Deployment > Build Tools > Maven > Importing
- 验证项目的 Project SDK 和 Project language level 设置为 Java 版本：
  - File > Project Structure... > Project
- 验证模块的 Language level 设置为 Java 版本：
  - File > Project Structure... > Modules，将右侧的的 Language level 值设置如下图：
- 验证 Global Libraries 已经配置了 Scala SDK
  - File > Project Structure... > Global Libraries，点击右侧的加号（+），浏览并选择本地安装的

Scala SDK 安装目录，设置如下图：

2.3.3 安装和配置 Maven

在开始编写 Spark 应用程序之前，我们需要将 Spark 库和文档导入 IntelliJ。要执行此操作，我们将使用 Maven。如果我们想让 IntelliJ 识别 Spark 代码，这是必要的。要导入 Spark 库，我们将使用依赖管理器 Maven。向 pom.xml 文件中添加以下行：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>xlw</groupId>
    <artifactId>myspark</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.11</artifactId>
            <version>2.3.2</version>
        </dependency>
    </dependencies>

</project>
```

保存文件后，IntelliJ 将自动导入运行 Spark 所需的库和文档。这个过程有可能会持续较长时间。

2.3.4 创建 Spark 应用程序

对于第一个 Spark 应用程序，我们将构建一个简单的程序，它对收集的莎士比亚作品（shakespear .txt）

执行单词计数。

稍后，我们将希望 Spark 从 HDFS (Hadoop 分布式文件系统)检索这个文件，所以现在让我们把它上传到 HDFS。

将 PBDP 大数据平台中的/home/hduser/data/spark/shakespeare.txt 文件上传到 HDFS：

1）确保已经启动了 HDFS；

2）在终端窗口中，执行以下命令，上传文件到 HDFS:

```
$ cd /home/hduser/data/spark
$ hdfs dfs -put shakespeare.txt /data/spark_demo/
```

现在准备创建应用程序。在 IDE 中打开文件夹 src/main/resources，这个文件夹应该已经生动生成了。将 shakespeare.txt 放在该文件夹下。

接下来，选择文件夹 src/main/java，在其上单击右键，然后选择 New > Java Class，创建一个名为 Main.java 的类。

编辑 Main.java 的源代码如下：

```
package xlw.myspark;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

现在转到 IDE 顶部的“Run”下拉菜单并选择 Run。然后选择 Main。如果一切设置正确，IDE 应该打印“Hello World”。

现在我们知道环境已经正确设置，用以下代码替换文件内容：

```
package xlw.myspark;

import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.SparkConf;
import scala.Tuple2;

import java.util.Arrays;

public class Main {

    public static void main(String[] args){

        // 创建要初始化的 SparkContext
        SparkConf conf = new SparkConf().setMaster("local").setAppName("Word Count");

        // 创建一个 Java 版本的 Spark Context
        JavaSparkContext sc = new JavaSparkContext(conf);
```

```
        // 加载文件文件到一个 Spark RDD，这是每行文本的分布式表示
        JavaRDD<String> textFile = sc.textFile("./src/main/resources/shakespeare.txt");

        // 单词计数
        JavaPairRDD<String, Integer> counts = textFile
                .flatMap(s -> Arrays.asList(s.split("[ ,]")).iterator())
                .mapToPair(word -> new Tuple2<>(word, 1))
                .reduceByKey((a, b) -> a + b);
        counts.foreach(p -> System.out.println(p));
        System.out.println("全部单词: " + counts.count());

        // 将统计结果保存到指定文件
        counts.saveAsTextFile("./shakespeareWordCount");
    }

}
```

如前所述，单击 Run > Run 以运行该文件。这应该运行 Spark 作业并打印莎士比亚中出现的每个单词的频率。

程序创建的文件位于上面代码中指定的目录中，在示例中我们使用./shakespeareWordCount。

现在我们已经了解了如何在 IDE 中直接部署应用程序。这是一种快速构建和测试应用程序的好方法，但是这有点不切实际，因为 Spark 只在一台机器上运行。在生产环境中，Spark 通常会处理存储在 HDFS 等分布式文件系统中的数据。Spark 通常也以集群模式运行(即分布在许多机器上)。

在接下来的部分中，我们将学习如何部署分布式 Spark 应用程序。

2.3.5 部署 Spark 应用程序

在本节中，我们将针对 Spark 集群进行部署。尽管我们仍然在一台机器上运行 Spark，但我们将使用 HDFS 和 Spark Standalone 集群资源管理器。

将源代码修改如下，注意粗体字部分的改变：

```
package xlw.myspark;

import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.SparkConf;
import scala.Tuple2;

import java.util.Arrays;

public class Main {

    public static void main(String[] args){

        // 创建要初始化的 SparkContext
        SparkConf conf = new SparkConf().setMaster("local").setAppName("Word Count");
```

```
        // 创建一个 Java 版本的 Spark Context
        JavaSparkContext sc = new JavaSparkContext(conf);

        // 加载文件文件到一个 Spark RDD，这是每行文本的分布式表示
        JavaRDD<String> textFile = sc.textFile("hdfs://localhost:8020/data/spark_demo/shakespeare.txt");

        // 单词计数
        JavaPairRDD<String, Integer> counts = textFile
                .flatMap(s -> Arrays.asList(s.split("[ ,]")).iterator())
                .mapToPair(word -> new Tuple2<>(word, 1))
                .reduceByKey((a, b) -> a + b);
        counts.foreach(p -> System.out.println(p));
        System.out.println("全部单词: " + counts.count());

        // 将统计结果保存到指定文件
        counts.saveAsTextFile("hdfs://localhost:8020/data/spark_demo/shakespeareWordCount");
    }

}
```

这告诉 Spark 读写 HDFS 上的数据文件，而不是本地数据文件。请确保保存该源代码文件。

接下来，我们将把这些代码打包到一个编译后的 jar 文件中，该文件可以部署在集群上。为了简化我们的工作，我们将创建一个 assembly jar：一个包含代码和代码所依赖的所有 jar 包的单个 jar 文件。通过将代码打包为程序集，我们可以确保在代码运行时，所有依赖 jar 包(在 pom.xml 中定义)都将出现。

打开一个终端并 cd 到包含 pom.xml 的目录。运行“mvn package”命令打包。这将会在 target 目录下创建一个编译过的名为“myspark-1.0-SNAPSHOT.jar”的 jar 包。

将“myspark-1.0-SNAPSHOT.jar”这个 jar 包拷贝到虚拟机的用户主目录下。然后打开另一个终端窗口，使用 spark-submit 运行我们的代码。我们需要指定主类、要运行的 jar 和运行模式(本地或集群)：

```
$ cd ~/bigdata/spark-3.1.2
$ spark-submit --class "xlw.myspark.Main" --master local ./myspark-1.0-SNAPSHOT.jar
```

控制台应该打印莎士比亚作品中出现的每个单词的频率，如下所示：

```
...
(comutual,1)
(ban-dogs,1)
(rut-time,1)
(ORLANDO],4)
(Deceitful,1)
(commits,3)
(GENTLEWOMAN,4)
(honors,10)
(returnest,1)
(topp'd?,1)
(compass?,1)
(toothache?,1)
(miserably,1)
(hen?,1)
(luck?,2)
```

```
(call'd,162)
(lecherous,2)
...
```

2.3.6 远程调试 Spark 应用程序

在本节中，我们将学习如何将正在运行的 Spark 程序连接到调试器，调试器允许我们设置断点并逐行调试代码。在直接从 IDE 运行时，调试 Spark 和其他任何程序一样，但是调试远程集群需要一些配置。

1）在计划提交 Spark 作业的机器上，从终端运行以下代码:

```
export SPARK_SUBMIT_OPTS=-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005
```

2）提交 Spark job 作业，运行时会出现程序挂起，监听端口中。如下：

```
spark-submit --class HelloScala   --master local   --driver-java-options   bigdata/helloscala_2.11-1.0.jar
```

3）在 IntelliJ IDEA 中配置 remote debug：在 IntelliJ 中选择菜单项 Run -> Edit Configurations，编辑运行配置信息。如下：

4）然后单击左上角的+按钮，选择面板左侧的 Remote 项，增加一个新的远程配置。在面板右侧，使用主机 ip 地址填写 Host（Port 字段默认为 5005，不需要修改），这将允许在端口 5005 上关联调试器。需要确保端口 5005 能够接收入站连接。

5）从 IDE 中 debug 此调试配置，调试器将附加此调试配置，并且远程 Spark 程序将在断点处停止。还可以检查程序中活动变量的值。在试图确定代码中的 bug 时，这是非常宝贵的。

1

2.4 使用 Zeppelin 进行交互式分析

Apache Zeppelin 是一款基于 Web 的 NoteBook，支持交互式数据分析。使用 Zeppelin，可以使用丰富的预构建语言后端（或解释器）制作精美的数据驱动、交互式和协作文档。目前，Apache Zeppelin 支持 Apache Spark、Python、JDBC、Markdown 和 Shell 等多种解释器。

特别是，Apache Zeppelin 提供了内置的 Apache Spark 集成。我们不需要为它构建单独的模块、插件或库。Apache Zeppelin 与 Spark 集成，提供了如下功能：

- 自动注入 SparkContext 和 SQLContext；
- 从本地文件系统或 maven 存储库加载运行时 jar 依赖项；
- 取消作业并显示进度。

Apache Zeppelin 专注于企业级应用，Zeppelin Notebook 可以满足以下企业用户以下需求：

- 数据摄取
- 数据发现
- 数据分析
- 数据可视化与协作

接下来，我们学习如何安装 Zeppelin 和配置 Zeppelin 解释器，并演示如何使用 Zepplin Notebook 作为 Spark 的交互式数据分析工具进行大数据的分析和数据可视化。

2.4.1 下载 zeppelin 安装包

Apache Zeppelin 的下载地址为：http://zeppelin.apache.org/download.html。请选择图中所示的版本：

将下载的安装包拷贝到~/software 目录下。

2.4.2 安装和配置 Zeppelin

请按以下步骤安装和配置 Zeppelin。

1）将下载的安装包解压缩到~/bigdata 目录下，并改名为 zeppelin-0.9.0：

```
$ cd ~/bigdata
$ tar xvf ~/software/zeppelin-0.9.0-bin-netinst.tgz
$ mv zeppelin-0.9.0-bin-netinst zeppelin-0.9.0
```

2）配置环境变量：

```
$ cd
$ sudo nano /etc/profile
```

在文件最后，添加如下内容：

```
export ZEPPELIN_HOME=/home/hduser/bigdata/zeppelin-0.9.0
export PATH=$PATH:$ZEPPELIN_HOME/bin
```

保存文件并关闭。

3）执行/etc/profile 文件使得配置生效：

```
$ source /etc/profile
```

4）打开 conf/zeppelin-env.sh 文件：（默认没有，从模板复制一份）

```
$ cd ~/bigdata/zeppelin-0.9.0/conf
$ cp zeppelin-env.sh.template zeppelin-env.sh
$ nano zeppelin-env.sh
```

在文件最后添加如下两行内容：

```
export JAVA_HOME=/usr/local/jdk1.8.0_281
export SPARK_HOME=/home/hduser/bigdata/spark-3.1.2
```

5）打开 zeppelin-site.xml 文件：（默认没有，从模板复制一份）

```
$ cd ~/bigdata/zeppelin-0.9.0/conf
$ cp zeppelin-site.xml.template zeppelin-site.xml
$ gedit zeppelin-site.xml
```

修改如下两个属性，设置新的端口号，以避免与 Spark Web UI 默认端口发生冲突：

```
<property>
  <name>zeppelin.server.port</name>
  <value>9090</value>
  <description>Server port.</description>
</property>

<property>
  <name>zeppelin.server.ssl.port</name>
  <value>9443</value>
  <description>Server ssl port. (used when ssl property is set to true)</description>
</property>
```

6）启动 zeppelin 服务

在终端窗口中，执行以下命令，启动 zeppelin 服务：

```
$ zeppelin-daemon.sh start
```

执行如下图所示：

7）关闭 zeppelin 服务

在终端窗口中，执行以下命令，停止 zeppelin 服务：

```
$ zeppelin-daemon.sh stop
```

2.4.3 配置 Spark 解释器

说明：如果是使用 Spark local 模式，此一步骤省略。如果是使用 Spark standalone 模式，需要配置 Spark 解释器。

首先启动浏览器，在浏览器地址栏输入 URL：http://xueai8:9090/，打开访问界面，如下图。点击右上角的小三角按钮，打开下拉菜单，点击“Interpreter”菜单项，打开解释器配置界面。

打开的解释器配置界面如下图所示。按图中所示找到 spark 解释器，然后修改 master 属性值为 spark://xueai8:7077(这实际上是连接到的集群管理器，我们这里使用的是 spark standalone 模式。这相当于启动 Spark Shell 时指定--master 参数），然后单击【Save】按钮保存。其它参数酌情设置。

2.4.4 创建和执行 notebook 文件

回到浏览器 zeppelin 首页，点击按钮，创建一个新的 notebook 文件，如下图所示：

然后在弹出的创建窗口，填写相应信息，然后单击【Create】按钮即可：

执行 Spark 交互式操作-Scala 语言

说明：如果是在 standalone 模式下使用 Zeppelin，请先启动 Spark 集群。

在新打开的 notebook 界面，执行 Spark 代码，如下图所示：

执行 Spark 交互式操作-Python 语言

新创建一个 notebook。

在新打开的 notebook 界面，执行 Python 代码。需要在第一行键入“%pyspark”，以告诉 zeppelin 使用 pyspark 解释器。如下图所示：

2.5 小结

- Apache Spark 自带了 spark-shell 命令行工具，通过它可以实现交互式执行 Spark 指令。

- Apache Spark 自带了 spark-submit 作业提交工具，通过它可以将 jar 包形式的作业提交到 Spark 集群上运行。
- 开发 Apache Spark 应用程序，可以使用多种工具。最受企业欢迎的 IntelliJ IDEA 集成开发环境。
- 我们可以使用 IntelliJ IDEA + Maven 构建 Spark 项目，也可以使用 IntelliJ IDEA + SBT 构建使用 Scala API 开发的 Spark 项目。
- 对于大数据分析人员来说，最佳的交互式大数据分析工具是 Zeppelin Notebook。Apache Zeppelin 专注于企业级应用，Zeppelin Notebook 可以满足以下企业用户以下需求：数据摄取、数据发现、数据分析、数据可视化与协作。

第 3 章　Spark 核心编程

Spark Core 模块包含 Spark 的基本功能，包括任务调度组件、内存管理、故障恢复、与存储系统交互等。在 Spark Core 模块中，核心的数据抽象被称为“弹性分布式数据集(RDD)”。RDD 是 Spark Core 的用户级 API，要真正理解 Spark 的工作原理，就必须理解 RDD 的本质。

Spark 为 Scala、Java、R 和 Python 编程语言提供了 APIs。Spark 本身是用 Scala 编写的，但 Spark 通过 PySpark 支持 Python。PySpark 构建在 Spark 的 Java API 之上(使用 Py4J)。通过 Spark（PySpark）上的交互式 shell，可以对大数据进行交互式数据分析。数据科学界大多选择 Scala 或 Python 来进行 Spark 程序开发和数据分析。

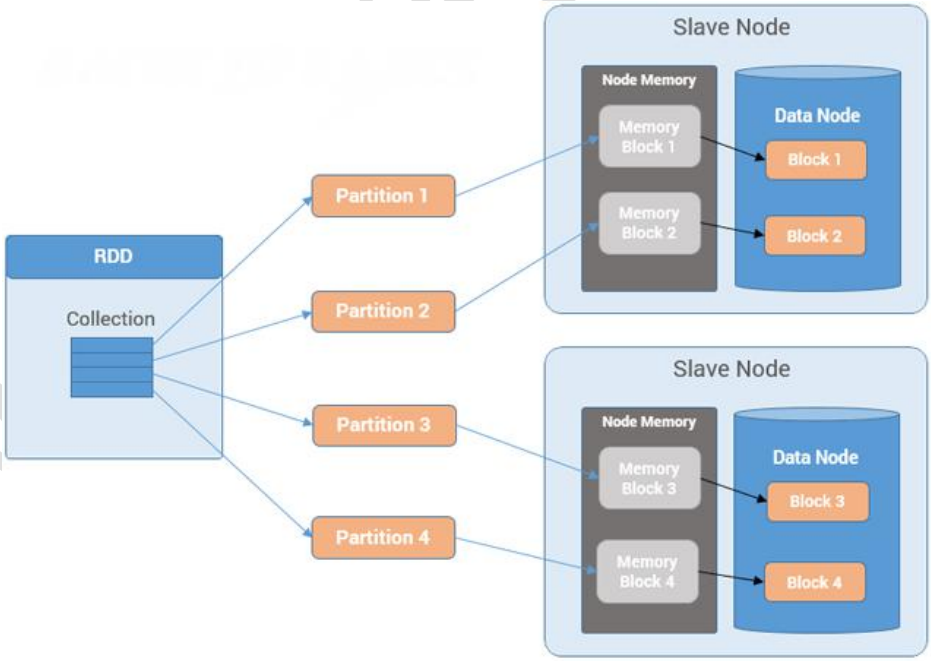
3.1 理解数据抽象 RDD

在 Spark 的编程接口中，每一个数据集都被表示为一个对象，称为 RDD。RDD 是一个只读的(不可变的)、分区的(分布式的)、容错的、延迟计算的、类型推断的和可缓存的记录集合。

所谓 RDD (Resilient Distributed Dataset，弹性分布式数据集)，指的是：

- Resilient：不可变的、容错的
- Distributed：数据分散在不同节点（机器，进程）
- Dataset：一个由多个分区组成的数据集

Spark RDD 是对跨集群分布的各个分区的引用的集合。参考下图理解：



RDD 是 Resillient Distributed Dataset(弹性分布式数据集)的简称，是分布式内存的一个抽象概念，提供了一种高度受限的共享内存模型。通常 RDD 很大，会被分成很多个分区，分别保存在不同的节点上。RDD 是不可变的、容错的、并行的数据结构，允许用户显式地将中间结果持久化到内存中，控制分区以优化数据放置，并使用一组丰富的操作符来操作它们。

RDD 被设计成不可变的，这意味着我们不能具体地修改数据集中由 RDD 表示的特定行。如果调用

一个 RDD 操作来操纵 RDD 中的行，该操作将返回一个新的 RDD。原 RDD 保持不变，新的 RDD 将以我们希望的方式包含数据。RDD 的不变性本质上要求 RDD 携带"血统"信息，Spark 利用这些信息有效地提供容错能力。

RDD 提供了一组丰富的常用数据处理操作。它们包括执行数据转换、过滤、分组、连接、聚合、排序和计数的能力。关于这些操作需要注意的一点是，它们在粗粒度级别上进行操作，这意味着相同的操作应用于许多行，而不是任何特定的行。

RDD 结构

综上所述，RDD 只是一个逻辑概念，它可能并不对应磁盘或内存中的物理数据。根据 Spark 官方描述，RDD 由以下五部分组成：

- 一组 partition（分区），即组成整个数据集的块；
- 每个 partition（分区）的计算函数（用于计算数据集中所有行的函数）；
- 所依赖的 RDD 列表（即父 RDD 列表）；
- （可选的）对于 key-value 类型的 RDD，则包含一个 Partitioner（默认是 HashPartitioner）；
- （可选的）每个 partition 数据驻留在集群中的位置（可选）；如果数据存放在 HDFS 上，那么它就是块所在的位置。

Spark 运行时使用这 5 条信息来调度和执行通过 RDD 操作表示的用户数据处理逻辑。前三段信息组成“血统”信息，Spark 将其用于两个目的。第一个是确定 RDDs 的执行顺序，第二个是用于故障恢复目的。

容错

Spark 通过使用“血统”信息重建失败的部分，自动地代表其用户处理故障。每一个 RDD 或 RDD 分区都知道如何在出现故障时重新创建自己。它有转换的日志，或者血统(lineage)，可依据此从稳定存储器或另一个 RDD 中重新创建自己的。因此，任何使用 Spark 的程序都可以确保内置的容错能力，而不考虑底层数据源和 RDD 类型。

RDD 特性

作为 Spark 中最核心的数据抽象，RDD 具有以下特征：

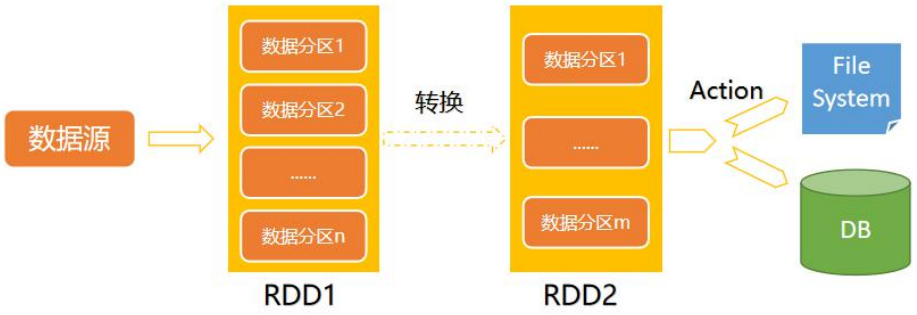
- In-Memory：RDD 会优先使用内存；
- Immutable（Read-Only）：一旦创建不可修改；
- Lazy evaluated：惰性执行；
- Cacheable：可缓存，可复用；
- Parallel：可并行处理；
- Typed：强类型，单一类型数据；
- Partitioned：分区的；
- Location-Stickiness：可指定分区优先使用的节点。

3.2 RDD 编程模型

在 Spark 中，使用 RDD 对数据进行处理，通常遵循如下的模型：

- 首先，将待处理的数据构造为 RDD；
- 对 RDD 进行一系列操作，包括 Transformation 和 Action 两种类型操作；
- 最后，输出或保存计算结果。

这个处理流程可以用下图表示：



接下来我们通过一个具体的示例来掌握 RDD 编程的一般流程。

3.2.1 单词计数应用程序

下面我们使用 Spark RDD 来实现经典的单词计数应用程序。

【示例】使用 Spark RDD 实现单词计数。这里我们使用 Zeppelin 作为开发工具，大家可以根据自己的喜好选择任意其他工具。

（1）首先准备一个文本文件 wc.txt，内容如下：

```
good good study
day day up
```

（2）将该文本文件上传到 HDFS 的"/data/spark_demo/"目录下：

```
$ hdfs dfs -put wc.txt /data/spark_demo/
```

（3）在 Zeppelin 中新建一个 notebook。在 notebook 的单元格中，执行代码。

（4）读取数据源文件，构造一个 RDD

```
val source = "hdfs://localhost:8020/data/wc.txt"
var textFile = sc.textFile(source)
```

（5）将每行数据按空格拆分成单词 - 使用 flatMap 转换

```
val words = textFile.flatMap(line => line.split(" "))
```

（6）将各个单词加上计数值 1 - 使用 map 转换

```
val wordPairs = words.map(word => (word,1))
```

（7）对所有相同的单词进行聚合相加求各单词的总数 - 使用 reduceByKey 转换

```
val vordCounts = wordPairs.reduceByKey((a,b) => a + b)
```

（8）返回结果给 Driver 程序，这一步才触发 RDD 开始实际的计算 - Action

```
wordCounts.collect()
```

（9）或者，也可以将计算结果保存到文件中 - Action

```
val sink = "hdfs://localhost:8020/data/result"
wordCounts.saveAsTextFile(sink)
```

在 Zeppelin 中交互式数据处理过程如下图所示：

以上代码也可以精简为下面一句：

```
val source = "hdfs://localhost:8020/data/wc.txt"
sc.textFile(source)
   .flatMap(_.split(" ")).
   .map((_,1)).
   .reduceByKey(_+_).
   .collect.
   .foreach(println)
```

如果在 IntelliJ IDEA 或 Eclipse 中开发，完整的代码如下：

```
package com.xueai8

import org.apache.spark.sql.SparkSession

object WordCount {
   def main(args: Array[String]): Unit = {

      // 创建 SparkSession 实例 - 入口
      val spark = SparkSession.builder.master("local[*]").appName("HelloWorld").getOrCreate

      // 加载数据源，构造 RDD
      val textFiles = spark.sparkContext.textFile("input/word.txt")

      // 对每一行数据分词，并扁平化
      val words = textFiles.flatMap(line => line.split(" "))

      // 将每个单词转换为元组形式（单词,1）
      val wordTuples = words.map(word => (word, 1))

      // 对同一个 key(单词)执行 reduce 操作
      val wordCounts = wordTuples.reduceByKey(_ + _)

      // 将结果输出
      wordCounts.collect.foreach(println)

      // 将结果保存到文件中
      wordCounts.saveAsTextFile("output/wordcount")
   }
}
```

3.2.2 理解 SparkSession

从 Spark 2.0 开始，SparkSession 已经成为 Spark 与 RDD、DataFrame 和 Dataset 一起工作的入口点。在 2.0 之前，SparkContext 曾经是一个入口点。在这里，我将主要通过定义和描述如何创建 SparkSession 和使用 spark-shell 默认的 SparkSession 变量来解释什么是 SparkSession。

什么 SparkSession？

SparkSession 在版本 Spark 2.0 中引入，全限定名称为 org.apache.spark.sql.SparkSession。它是 Spark

底层功能的入口点，用于编程创建 Spark RDD、DataFrame 和 DataSet。SparkSession 的实例对象 spark 在 spark-shell 中是默认可用的，它可以通过 SparkSession 构建器模式以编程方式创建。

正因为 SparkSession 是 Spark 的一个入口点，创建 SparkSession 实例将是使用 RDD、DataFrame 和 Dataset 编写程序的第一个语句。SparkSession 将使用 SparkSession.builder()构建器模式创建。

虽然 SparkContext 是 2.0 之前一个入口点，但它并没有被 SparkSession 完全取代，SparkContext 的许多特性仍然可用，并在 Spark 2.0 和以后的版本中使用。我们还应该知道 SparkSession 内部使用 SparkSession 提供的配置创建 SparkConfig 和 SparkContext。

SparkSession in spark-shell

调用 spark-shell 时，默认提供了 spark 对象，它是 SparkSession 类的一个实例。

在 Scala 程序中创建 SparkSession

要在 Scala 或 Python 中创建 SparkSession，需要使用构建器模式方法 builder()并调用 getOrCreate()方法。如果 SparkSession 已经存在，它返回存在的对象，否则创建新的 SparkSession。

```
val spark = SparkSession.builder()
        .master("local[*]")
        .appName("SparkExamples")
        .getOrCreate();
```

3.2.3 理解 SparkContext

SparkContext 从 Spark 1.x 引入的（对于 Java API 来说是 JavaSparkContext），在 2.0 中引入 SparkSession 之前，用来作为 Spark 和 PySpark 的入口点。使用 RDD 编程和连接到 Spark Cluster 的第一步就是创建 SparkContext。SparkContext 是在 org.apache.spark 包中定义的，它用于在集群中通过编程方式创建 Spark RDD、累加器和广播变量。

注意，每个 JVM 只能创建一个 SparkContext。

在任何给定时间，每个 JVM 应该只有一个 SparkContext 实例是活动的。如果想创建另一个新的 SparkContext，应该在创建一个新的 SparkContext 之前停止现有的 SparkContext(使用 stop())。

SparkContext in spark-shell

Spark shell 默认提供了一个名为“sc”的对象，该对象是 SparkContext 类的一个实例。我们可以在需要时直接使用该对象。

在 Scala 程序中创建 SparkContext

当使用 Scala、PySpark 或 Java 编程时，首先需要创建一个 SparkConf 实例，并分配应用名称和设置 master（分别使用 SparkConf 的静态方法 setAppName()和 setMaster()），然后将 SparkConf 对象作为参数传递给 SparkContext 构造器来创建 SparkContext。实现代码如下所示:

```
val sparkConf = new SparkConf().setAppName("sparkexamples").setMaster("local[*]")
val sc = new SparkContext(sparkConf)
```

SparkContext 构造函数在 2.0 中已经弃用，因此建议使用静态方法 getOrCreate()来创建 SparkContext。该函数用于获取或实例化 SparkContext，并将其注册为一个单例对象。

```
SparkContext.getOrCreate(sparkConf)
```

一旦创建了 Spark Context 对象，就可以使用它来创建 Spark RDD。

在 Spark 2.x 中创建 SparkContext

自从 Spark 2.0 以来，我们主要使用 SparkSession，SparkContext 中的大多数方法也存在于 SparkSession 中，并且 SparkSession 内部创建了 SparkContext 并公开了 sparkContext 变量供使用。

```
val sc = spark.sparkContext
```

3.3 创建 RDD

在对数据进行任何 transformation 或 action 操作之前，必须先将这些数据构造为一个 RDD。Spark 提供了创建 RDDs 的三种方法，分别为：

- 第一种方法是将现有的集合并行化。
- 另一种方法是加载外部存储系统中的数据集，比如文件系统。
- 第三种方法是在现有 RDD 上进行转换来得到新的 RDD。

3.3.1 将现有的集合并行化以创建 RDD

创建 RDD 的第一种方法是将对象集合并行化，这意味着将其转换为可以并行操作的分布式数据集。这种方法最简单，是开始学习 Spark 的好方法，因为它不需要任何数据文件。这种方法通常用于快速尝试一个特性或在 Spark 中做一些试验。对象集合的并行化是通过调用 SparkContext 类的 parallelize 方法实现的。请看下面的代码：

```
// 可以从列表中创建
val list1 = List(1,2,3,4,5,6,7,8,9,10)
val rdd1 = sc.parallelize(list1)
rdd1.collect

// 通过并行集合（range）创建 RDD
val list2 = List.range(1,11)
val rdd2 = sc.parallelize(list2)
rdd2.collect

// 通过并行集合（数组）创建 RDD
val arr = Array(1,2,3,4,5,6,7,8,9,10)
val rdd3 = sc.parallelize(arr)
rdd3.collect

// 通过并行集合（数组）创建 RDD
val strList = Array("明月几时有","把酒问青天","不知天上宫阙","今夕是何年")
val strRDD = sc.parallelize(strList)
strRDD.collect
```

3.3.2 从存储系统读取数据集以创建 RDD

创建 RDD 的第二种方法是从存储系统读取数据集，存储系统可以是本地计算机文件系统、HDFS、

Cassandra、Amazon S3 等等。Spark 可以从 Hadoop 支持的任何数据源创建 RDD，包括其本地文件系统、HDFS、Cassandra、HBase、Amazon S3 等。Spark 支持 Hadoop InputFormat 支持的任何格式。请看下面的代码：

```
// 或者，也可以从文件系统中加载数据创建 RDD
val file = "/data/spark_demo/rdd/wc.txt"        // hdfs
val rdd1 = sc.textFile(file)
```

SparkContext 类的 textFile 方法假设每个文件是一个文本文件，并且每行由一个新行分隔。此 textFile 方法返回一个 RDD，它表示所有文件中的所有行。需要注意的重要一点是，textFile 方法是延迟计算的，这意味着如果指定了错误的文件或路径，或者错误地拼写了目录名，那么在采取其中一项 action 操作之前，这个问题不会出现（被发现）。

处理 JSON 数据文件

假设我们有如下的员工信息文件 peoples.json，以 JSON 格式：

```
{"name":"张三"}
{"name":"李四", "age":30}
{"name":"王老五", "age":19}
```

下面是构造 RDD 进行处理的代码实现，我们用到的 scala.util.parsing.json.JSON 类。

```
package com.xueai8.rdd

import org.apache.spark.sql.SparkSession

import scala.util.parsing.json.JSON

object ReadJsonDemo {
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 实例 - 入口
    val spark = SparkSession.builder.master("local[*]").appName("HelloWorld").getOrCreate

    // 读取文件，构造 RDD
    val jsonRDD = spark.sparkContext.textFile("input/json/peoples.json")

    // 解析
    val result = jsonRDD.map(line => JSON.parseFull(line))

    // 输出，使用模式匹配
    result.collect.foreach(r => r match{
      case Some(map) => println(map)
      case None          => println("解析失败")
      case other         => println("未知数据结构: " + other)
    })
  }
}
```

执行以上代码，输出结果如下：

```
Map(name -> 张三)
Map(name -> 李四, age -> 30.0)
Map(name -> 王老五, age -> 19.0)
```

3.3.3 从已有的 RDD 转换得到新的 RDD

创建 RDD 的第三种方法是调用现有 RDD 上的一个转换操作。例如，下面的代码通过对 rdd4 的转换得到一个新的 RDD - rdd5：

```
// 字符转为大写，得到一个新的 RDD
val rdd5 = rdd4.map(line => line.toUpperCase)
rdd5.collect
```

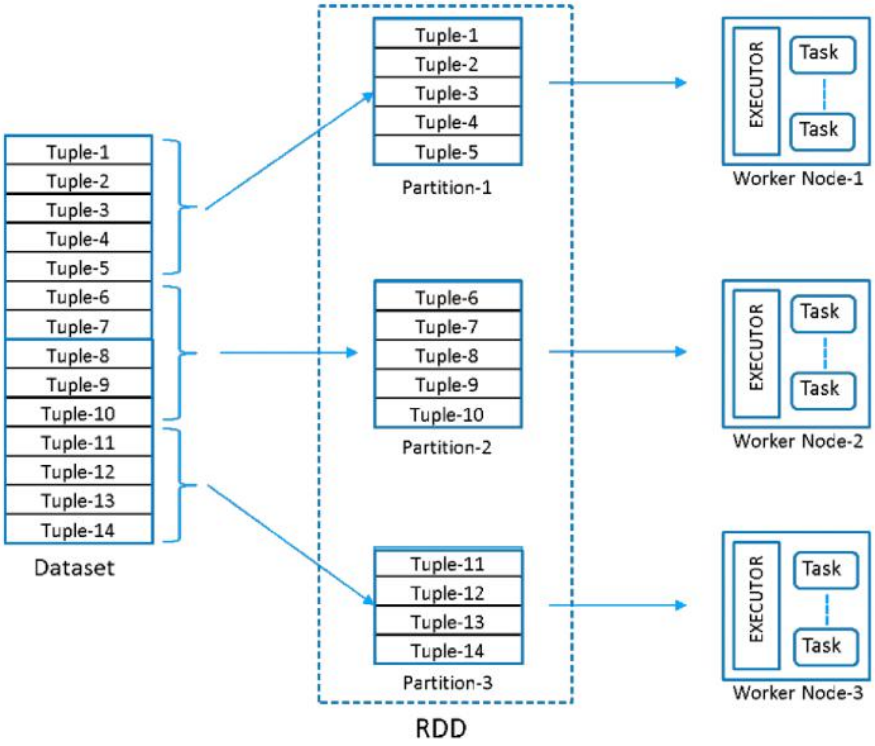
注：关于 map 函数，在稍后部分讲解。

3.3.4 创建 RDD 时指定分区数量

Spark 在集群的每个分区上运行一个任务（task），因此必须谨慎地决定优化计算工作。尽管 Spark 会根据集群自动设置分区数量，但我们可以通过将其作为第二个参数传递给并行化函数。例如：

```
sc.parallelize(data,3)        // 3 个分区
```

下图表示创建一个 RDD，包含 14 条记录(或元组)，分区为 3，分布在三个节点上：



3.4 操作 RDD

创建了 RDD 之后，就可以编写 Spark 程序对 RDD 进行操作。RDD 操作分为两种类型：转换(Transformation)和动作(action)。转换(Transformation)是用来创建 RDDs 的方法，而动作(action)是使用 RDDs 的方法。

3.4.1 RDD 上的 Transformation 和 Action

RDD 支持两种类型的操作：transformations 和 actions。

Transformation 是定义如何构建 RDD 的延迟操作。大多数转换都接受单个函数参数。所有这些方法都将一个数据源转换为另一个数据源。每当在任何 RDD 上执行转换时，都会生成一个新的 RDD，如下图所示：

RDD 操作在粗粒度级别上操作，这在前面已经描述过。数据集中的每一行都表示为 Java 对象，这个 Java 对象的结构对于 Spark 来说是不透明的。RDD 的用户可以完全控制如何操作这个 Java 对象。

RDD 是不可变的（只读的）数据结构，因此任何转换都会产生新的 RDD。转换操作被延迟计算，我们称为“惰性转换”，这意味着 Spark 将延迟对被调用的操作的执行，直到采取 action。换句话说,转换操作仅仅记录指定的转换逻辑，并在稍后的时候应用它们。当调用 action 操作将触发对它之前的所有转换的求值，它将向驱动程序返回一些结果，或者将数据写入存储系统，如 HDFS 或本地文件系统。延迟计算概念背后的一个重要优化技术是在执行期间将类似的转换折叠或组合为单个操作的能力，即优化转换步骤。例如，如果动作是返回第一行，Spark 就只计算单个分区，然后跳过其余部分。

简而言之，RDD 是不可变的，RDD 转换是延迟计算的，RDD action 是即时计算的，并触发数据处理逻辑的计算。而在 RDD 的内部实现机制中，底层接口则是基于迭代器的，从而使得数据访问变得更高效，也避免了大量中间结果对内存的消耗。

通过应用程序操作 RDD 与操作数据的本地集合类似。请看下面这个简单的代码：

```
val lines = sc.textFile("hdfs://path/to/the/file")
val filteredLines = lines.filter(line => line.contains("spark")).cache()
val result = filteredLines.count()
```

上面这段代码的意思是，从 HDFS 上加载指定的日志文件，找出包含单词"spark"的行数。其在内存中的计算和转换过程可用如下的图来表示：

（1）一个 300MB 的日志文件，分布式存储在 HDFS 上，如下图所示：

（2）调用这行代码，将其加载到分布式的内存中：

```
val lines = sc.textFile("hdfs://path/to/the/file")
```

（3）执行下面这行代码，过滤满足条件的行(即只包含单词"spark"的行)，这是原始数据集的一个子集，并将这个中间结果缓存到内存中：

```
val filteredLines = lines.filter(line => line.contains("spark")).cache()
```

（4）执行最后一行代码，统计过滤后的行数，返回给驱动程序 Driver：

```
val result = filteredLines.count()
```

3.4.2 RDD Transformation 操作

Transformation 是操作 RDD 并返回一个新的 RDD，如 map()和 filter()方法，而 action 是返回一个结果给驱动程序或将结果写入存储的操作，并开始一个计算，如 count()和 first()。

Spark 对于 transformation RDD 是延迟计算的，只在遇到 action 时才真正进行计算。许多转换是作用于元素范围内的，也就是一次作用于一个元素。

现在假设有一个 RDD，包含元素为{1, 2, 3, 3}。首先，让我们构造出一个 RDD：

```
// 构造一个 RDD
val rdd = sc.parallelize(List(1,2,3,3))
```

接下来，学习普通 RDD 上的各种转换操作方法：

- 1）map(func)

```
// map 转换
val rdd1 = rdd.map(x => x + 1)    // tansformation
rdd1.collect          // action
```

```
rdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[34] at map at <console>:29
res59: Array[Int] = Array(2, 3, 4, 4)
```

map 转换练习：分析以下转换后的结果。

```
val a = sc.parallelize(List("animal", "human", "bird", "rat"), 3)
val b = a.map(_.length)
val c = a.zip(b)
c.collect
```

- mapPartitions(func)

通过对这个 RDD 的每个分区应用一个函数来返回一个新的 RDD。

mapPartitions()可以作为 map()和 foreach()的替代方法。可以对每个分区调用 mapPartitions()，而对 RDD 中的每个元素调用 map()和 foreach()。因此，可以根据每个分区而不是每个元素进行初始化。

```
// 构造 RDD
val x = spark.sparkContext.parallelize(Array(1,2,3,4,5,6,7,8,9,10), 2)

// mapPartitions 转换
x.mapPartitions(iter => Iterator(iter.toArray))
 .collect
 .foreach(item => println(item.toList))

// 自定义函数（迭代器求和）
def f(i:Iterator[Int]) = {
    // 每个分区的每个元素翻倍
    Tuple1(i.sum).productIterator
}

// 应用 mapPartitions 转换，求每个分区的元素和
val y = x.mapPartitions(f)

// 返回结果
y.collect
```

- mapPartitionsWithIndex(func)

通过对这个 RDD 的每个分区应用一个函数来返回一个新的 RDD，同时跟踪原始分区的索引。mapPartitionsWithIndex 类似于 mapPartitions()，但它提供了第二个参数索引，用于跟踪分区。

```
val x = spark.sparkContext.parallelize(Array(1,2,3,4,5,6,7,8,9,10), 2)

// 定义函数 f
```

```
def f(partitionIndex:Int, i:Iterator[Int]) = {
      (partitionIndex, i.sum).productIterator
}

val y = x.mapPartitionsWithIndex(f)

y.glom.collect
```

❑ 2）flatMap(func)

```
val rdd2 = rdd.flatMap(x => x.to(3))
rdd2.collect
res65: Array[Int] = Array(1, 2, 3, 2, 3, 3, 3)
```

flatMap 转换练习：分析以下转换后的结果。

```
val a = sc.parallelize(1 to 5, 4)
a.flatMap(1 to _).collect

sc.parallelize(List(5, 10, 20), 2).flatMap(x => List(x, x, x)).collect
```

❑ 3）filter(func)

```
val rdd3 = rdd.filter(x => x!=1)
rdd3.collect
res68: Array[Int] = Array(2, 3, 3)
```

filter 转换练习：分析以下转换后的结果。

```
val a = sc.parallelize(1 to 10, 3)
val b = a.filter(_ % 3 == 0)
b.collect
```

❑ sample(withReplacement, fraction, seed)

返回这个 RDD 的一个采样子集。其中各参数含义如下：

- withReplacement：是否可以对元素进行多次采样(采样后替换)
- fraction：抽样因子。对于 without replacement，每个元素被选中的概率，fraction 值必须是[0,1]之间；对于 with replacement，每个元素被选择的期望次数，fraction 值必须大于等于 0。
- seed：用于随机数生成器的种子。

注意：这并不能保证精确地提供给定 RDD 的计数的因子。

```
val rdd5 = rdd.sample(false,0.5)
rdd5.collect
res74: Array[Int] = Array(1)
```

❑ 4）distinct([numPartitions]))：返回一个包含这个 RDD 中不同元素的新 RDD。

```
val rdd4 = rdd.distinct()
rdd4.collect
res71: Array[Int] = Array(1, 2, 3)
```

distinct 转换练习：分析以下转换后的结果。

Scala:

```
val c = sc.parallelize(List("John", "Jack", "Mike", "Jack"), 2)
```

```
c.distinct.collect
```

- keyBy(func): RDD[(K, T)]

当在类型为T的数据集上调用时，返回一个(K, T)元组对的数据集。通过应用func函数创建这个RDD中元素的元组。

```
val   x= sc.parallelize(Array("John", "Fred", "Anna", "James"))
val   y= x.keyBy(w => w.charAt(0))
println(y.collect().mkString(", "))
```

- groupBy(func)，groupBy(func, numPartitions)，groupBy(func, partitioner)。

当在类型为T的数据集上调用时，返回一个(K, Iterable[T])元组的数据集。

返回分组项的RDD。每个组由一个key和一系列映射到该key的元素组成。每个组内元素的顺序不能得到保证，甚至在每次计算结果RDD时可能会有所不同。这个方法有可能会引起数据shuffle。

```
val x = sc.parallelize(Array("Joseph", "Jimmy", "Tina", "Thomas", "James",
                             "Cory", "Christine", "Jackeline", "Juan"), 3)

// 每第一个字符创建一个组
val y = x.groupBy(word => word.charAt(0))

y.collect

// 另一个短的语法
val y = x.groupBy(_.charAt(0))

y.collect
```

- sortBy(func,[ascending],[numPartitions])

返回这个按给定key函数排序的RDD。

```
val data = List(3,1,90,3,5,12)
val rdd = sc.parallelize(data)
rdd.collect

// 默认升序
rdd.sortBy(x => x).collect

// 降序
rdd.sortBy(x => x, false).collect

val result = rdd.sortBy(x => x, false)
result.partitions.size

// 改变分区数为 1
val result = rdd.sortBy(x => x, false, 1)

result.partitions.size
```

上面的实例对rdd中的元素进行升序排序。并对排序后的RDD的分区个数进行了修改，上面的result就是排序后的RDD，默认的分区个数是2，而我们对它进行了修改，所以最后变成了1。

- glom()：RDD[Array[T]]

返回将每个分区中的所有元素合并到一个数组中创建的 RDD，一个分区一个数组。当在类型为 T 的 RDD 上调用时，返回一个 Array[T]的 RDD。

```
// 构造 RDD
val x = spark.sparkContext.parallelize(Array(1,2,3,4,5,6,7,8,9,10), 2)

// 将每个分区中的所元素合并成一个数组并返回
x.glom().collect

// 分区求和
x.glom().map(_.sum).collect
```

- repartition(numPartitions)：随机地重新 shuffle RDD 中的数据，以创建更多或更少的分区，并在它们之间进行平衡。repartition()用于增加或减少 RDD 分区。需要注意的一点是，Sparkcoalesce()是非常昂贵的操作，因为它会跨多个分区转移数据。

```
val rdd = spark.sparkContext.parallelize(Range(0,20))
println("当前分区数：" + rdd.partitions.size)                    // 2

val rdd2 = rdd.repartition(4)
println("重分区后，现在的分区数：" + rdd2.partitions.size)       // 4
```

- coalesce(numPartitions)：将 RDD 中的分区数量减少到 numpartition。coalesce()仅用于以一种有效的方式减少分区数量，适用于过滤大型数据集后更有效地运行操作。这是 repartition()的优化或改进版本，其中使用合并可以降低跨分区的数据移动。需要注意的一点是，Spark repartition()是非常昂贵的操作，因为它会跨多个分区转移数据。

```
val rdd = spark.sparkContext.parallelize(Range(0,20))
println("当前分区数：" + rdd.partitions.size)                    // 2

val rdd3 = rdd.coalesce(1)
println("合并分区后，现在的分区数：" + rdd3.partitions.size)     // 1
```

- randomSplit(weights, seed)

使用提供的权重随机分割这个 RDD，以数组形式返回拆分后的 RDD（即拆分后的 RDD 组成的数组并返回）。其中各参数含义如下：

  - weights：分割的权重，如果它们的和不等于 1，将被标准化。
  - seed：随机种子。

```
// 构造一个 RDD
val rdd1 = spark.sparkContext.parallelize(Array(1,2,3,4,5,6,7,8,9,10))

// 按 80/20 分割数据集
val splitedRDD = rdd1.randomSplit(Array(0.8,0.2))

// 查看
splitedRDD(0).collect                // Array(2, 4, 5, 6, 7, 8, 9, 10)
splitedRDD(0).collect                // Array(1, 3)
```

RDD 集合运算

现在假设有两个 RDD，分别包含{1,2,3,3}和{3,4,5}。首先，让我们构造出这两个 RDD：

```
// 构造这两个 RDD
val rdd1 = sc.parallelize(List(1,2,3,3))
rdd1.collect

val rdd2 = sc.parallelize(List(3,4,5))
rdd2.collect
```

接下来操作这两个 RDD，如下：

- union(otherDataset)

```
val rdd3 = rdd1.union(rdd2)
rdd3.collect
```

```
res79: Array[Int] = Array(1, 2, 3, 3, 3, 4, 5)
```

union 转换练习：分析以下转换后的结果。

```
val a = sc.parallelize(3 to 7, 1)
val b = sc.parallelize(7 to 9, 1)
val c = a.union(b)            // 另一种方式是 (a ++ b).collect
c.collect
```

- intersection(otherDataset)

```
val rdd4 = rdd1.intersection(rdd2)
rdd4.collect
```

```
res82: Array[Int] = Array(3)
```

intersection 转换练习：分析以下转换后的结果。

```
val x = sc.parallelize(1 to 10)
val y = sc.parallelize(5 to 15)
val z = x.intersection(y)
z.collect
```

- subtract 转换

```
val rdd5 = rdd1.subtract(rdd2)
rdd5.collect
```

```
res85: Array[Int] = Array(1, 2)
```

- cartesian(otherDataset)(即笛卡尔集)：当在类型为 T 和 U 的 RDD 上调用时，返回一个(T, U)对(所有元素对)的 RDD。

```
val rdd6 = rdd1.cartesian(rdd2)
rdd6.collect
```

```
res88: Array[(Int, Int)] = Array((1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,3), (3,4), (3,5), (3,3), (3,4), (3,5))
```

cartesian 转换练习：分析以下转换后的结果。

```
val x = sc.parallelize(List(1,2,3))
val y = sc.parallelize(List(10,11,12))
x.cartesian(y).collect
```

- zip(other)：当在类型为 T 和 U 的 RDD 上调用时，返回一个(T, U)对的 RDD，其中元组第一个元素来自第一个 RDD，第二个元素来自第二个 RDD。这类似于拉链操作。假设两个 RDD 具有相同数量的分区和每个分区中相同数量的元素(例如，一个 RDD 通过另一个 RDD 上的 map 生成)。

```
val rdd1 = spark.sparkContext.parallelize(Array("aa","bb","cc"))
val rdd2 = spark.sparkContext.parallelize(Array(1,2,3))

val rdd3 = rdd1.zip(rdd2)
rdd3.collect                    // Array((aa,1), (bb,2), (cc,3))
```

详细 API 说明请参考 http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD。

3.4.3 RDD Action 操作

Action 是返回一个结果给驱动程序或将结果写入存储的操作，并开始一个计算，如 count()和 first()。

一旦创建了 RDD，就只有在执行了 action 时才会执行各种转换。一个 action 的执行结果可以是将数据写回存储系统，或者返回到驱动程序，以便在本地进行进一步的计算。常用的 action 操作函数如下：

- reduce(func)：使用函数 func(接受两个参数并返回一个参数)聚合数据集的元素。这个函数应该是交换律和结合律，这样才能并行地正确地计算它。
- collect()：将 RDD 操作的所有结果返回给驱动程序。这通常对产生足够小的数据集的操作很有用。
- count()：这会返回数据集中的元素数量或 RDD 操作的结果输出。
- first()：返回数据集的第一个元素或 RDD 操作产生的结果输出。它的工作原理类似于 take(1)函数。
- take(n)：返回 RDD 的前 n 个元素。它首先扫描一个分区，然后使用该分区的结果来估计满足该限制所需的其他分区的数量。这个方法应该只在预期得到的数组很小的情况下使用，因为所有的数据都加载到驱动程序的内存中。
- top(n)：按照指定的隐式排序[T]从这个 RDD 中取出最大的 k 个元素，并维护排序。这与 takeOrdered 相反。这个方法应该只在预期得到的数组很小的情况下使用，因为所有的数据都加载到驱动程序的内存中。
- takeSample(withReplacement, num, [seed])：返回一个数组，其中包含来自数据集的元素的 num 个随机样本。它有三个参数，如下：
  - withReplacement/withoutReplacement：这表示采样时有或没有替换（在取多个样本时，它指示是否将旧的样本替换回集合，然后取一个新的样本或者在不替换的情况下取样本）。对于 withReplacement，参数应该是 True 和 False。
  - num：这表示样本中元素的数量。
  - seed：这是一个随机数生成器种子（可选）。
- takeOrdered(n)：返回 RDD 的前 n 个（最小的）元素，并维护排序。这和 top 是相反的。这个方法应该只在预期得到的数组很小的情况下使用，因为所有的数据都加载到驱动程序的内存中。
- saveAsTextFile(path)：将数据集的元素作为文本文件(或文本文件集)写入本地文件系统、HDFS 或任何其他 hadoop 支持的文件系统的给定目录中。Spark 将对每个元素调用 toString，将其转换

为文件中的一行文本。

- countByKey()：仅在类型(K, V)的 RDDs 上可用。返回(K, Int)对的 hashmap 和每个键的计数。
- foreach(func)：在数据集的每个元素上运行函数 func。

假设一个 RDD，包含{1, 2, 3, 3}。下面是一些常用 action 操作代码示例。

```
// 构造 RDD
val rdd1 = spark.sparkContext.parallelize(Array(4,5,1,2,8,9,3,6,7,10))

rdd.count
rdd.first
rdd.collect

// countByValue: 返回每个元素在 RDD 中出现的次数
rdd.countByValue

// take(n)：返回 RDD 中的前 n 个元素
rdd.take(3)

// top(n)：返回 RDD 中的 top(n)个元素
rdd.top(3)

// takeOrdered(n)：返回 n 个元素，基于隐含的顺序
rdd.takeOrdered(3)

// takeSample(withReplacement,num,[seed])：随机返回 num 个元素
rdd.takeSample(false,2)
```

下面我们着重介绍其中几个 action 函数。

- reduce

这是一个 action，并且一个宽依赖操作。例如，在下面的示例中，使用 reduce 计算 List(1,2,3,3)中所有元素的和。

```
// 构造 RDD
val rdd = sc.parallelize(List(1,2,3,3))

// reduce(func)
rdd.reduce((x,y) => x + y)

// 等价
rdd.reduce(_+_)
```

- aggregate(zeroValue)(seqOp,combOp)

类似于 reduce，但用来返回不同的类型。这个函数聚合每个分区的元素，然后使用给定的 combine 组合函数和一个中性的“零值”，对所有分区的结果进行聚合。其中各参数的含义如下：

  - zeroValue：seqOp 操作符的每个分区的累积结果的初始值，combOp 操作符的不同分区的合并结果的初始值—这通常是中性元素(例如，列表连接为 Nil 或求和为 0 或求积为 1)。
  - seqOp：用于在分区内累积结果的运算符。

- combOp：用于合并来自不同分区的结果的关联运算符

这个 aggregate 函数类似于 reduce()和 fold()。但 reduce 和 fold 这两个函数有一个问题，那就是它们的返回值必须与 RDD 的数据类型相同。aggregate()函数就打破了这个限制。比如可以返回(Int, Int)元组，这在要计算平均值的时候很有用。

【示例】使用 Spark RDD aggregate()函数计算 RDD 元素的总和。

```
import org.apache.spark.sql.SparkSession

object AggregateExample2 extends App {

  val spark = SparkSession.builder()
    .appName("SparkByExamples.com")
    .master("local")
    .getOrCreate()

  spark.sparkContext.setLogLevel("ERROR")

  // 构造 RDD
  val listRdd = spark.sparkContext.parallelize(List(1,2,3,4,5,3,2))

  // 定义分区计算函数
  def param0= (accu:Int, v:Int) => accu + v

  // 定义分区结果合并计算函数
  def param1= (accu1:Int,accu2:Int) => accu1 + accu2

  // 执行 aggregate 计算
  val result = listRdd.aggregate(0)(param0,param1)
  println("输出: " + result)
}
```

执行以上代码，输出结果如下：

```
输出: 20
```

【示例】使用 aggregate 计算 List(1,2,3,3)中所有元素的平均值。

要算平均值，需求计算出两个值，一个是 RDD 的各元素的累加和，另一个是元素计数。对于加法计算，要初始化为(0, 0)。

```
import org.apache.spark.sql.SparkSession

object AggregateExample2 extends App {

  val spark = SparkSession.builder()
    .appName("SparkByExamples.com")
    .master("local")
    .getOrCreate()

  spark.sparkContext.setLogLevel("ERROR")

  // 构造一个 RDD，指定有两个分区
  val rdd = sc.parallelize(List(1,2,3,3), 2)
```

```
  // 查看分区数
  println(s"分区数: ${rdd.partitions.size}")

  // 计算所有元素的平均值
  def param1= (accu:Int, v:Int) => accu + v
  def param2= (accu1:Int, accu2:Int) => accu1 + accu2

  val result = rdd.aggregate((0,0))((acc,e)=>(acc._1+e, acc._2+1),
                                    (acc1,acc2)=>(acc1._1+acc2._1, acc1._2+acc2._2))
  val avg = result._1/result._2.toDouble

  println(s"RDD 中所有元素的平均值是：${avg}")
}
```

执行以上代码，输出结果如下：

```
分区数: 2
RDD 中所有元素的平均值是：2.25
```

3.4.4 RDD 上的描述性统计操作

Spark 在包含数值数据的 RDD 上提供了许多描述性统计操作。描述性统计都是在数据的单次传递中计算的。请看下面的代码：

```
// 构造一个 RDD
val rdd1 = sc.parallelize(1 to 20 by 2)

rdd1.collect
rdd1.sum
rdd1.max
rdd1.min
rdd1.count
rdd1.mean
rdd1.variance()
rdd1.stdev()
```

用直方图可视化数据分布：

```
// 方法 1
// rdd1.histogram(Array(1.0, 10.0, 20.9))
rdd1.histogram(Array(1.0, 8.0, 20.9))

// 方法 2
rdd1.histogram(3)
```

如果是多次调用描述性统计方法，则可以使用 StatCounter 对象。可以通过调用 stats()方法返回一个 StatCounter 对象：

```
// 通过调用 stats()方法返回一个 StatCounter 对象
val status = rdd1.stats()
```

```
status.count
status.mean
status.stdev
status.max
status.min
status.sum
status.variance
```

```
status: org.apache.spark.util.StatCounter = (count: 10, mean: 10.000000, stdev: 5.744563, max: 19.000000, min: 1.000000)
```

3.5 Key-Value Pair RDD

有一类特殊的 RDD，其元素是以<key,value>对的形式出现，我们称之为"Pair RDD"。针对 key/value pair RDD，Spark 专门提供了一些操作，这些操作只在 key/value 对的 RDD 上可用。

3.5.1 创建 Pair RDD

Spark 在包含 key/value 对的 RDDs 上提供了专门的 transformation API，包括 reduceByKey、groupByKey、sortByKey 和 join 等。Pair RDD 让我们能够在 key 上并行操作，或者跨网络重新组织数据。Key/value RDD 常被用于执行聚合操作，以及常被用来完成初始的 ETL(extract, transform, load)以获取 key/value 格式数据。

注意，除了 count 操作之外，大多数操作通常都涉及到 shuffle，因为与 key 相关的数据可能并不总是驻留在单个分区上。

创建 Pair RDD

创建 Pair RDD 的方式有多种。第一种创建方式：从文件中加载。

请看下面的代码：

```
val file = "/data/spark_demo/rdd/wc.txt"
val lines = sc.textFile(file)

val pairRDD = lines.flatMap(line => line.split(" ")).map(word => (word,1))    // 通过转换，生成 Pair RDD
pairRDD.collect
```

第二种方式：通过并行集合创建 Pair RDD

请看下面的代码：

```
val rdd = sc.parallelize(Seq("Hadoop","Spark","Hive","Spark"))
val pairRDD = rdd.map(word => (word,1))
pairRDD.collect

val a = sc.parallelize(List("black", "blue", "white", "green", "grey"), 2)
// 通过应用指定的函数来创建该 RDD 中元素的元组(参数函数生成对应的 key)，返回一个 pair RDD
val b = a.keyBy(_.length)
b.collect
```

练习：创建一个 pair RDD，使用每行的第一个单词作为 key。

```
val text = Seq("good good study","day day up")
val rdd = sc.parallelize(text)
val pairs = rdd.map(x => (x.split(" ")(0),x))                // 返回元组 tuple
pairs.collect
```

3.5.2 操作 Pair RDD

假设有一个 Pair RDD {(1,2),(3,4),(3,6)}。

```
// 构造 pair rdd
val pairRDD = sc.parallelize(Seq((1,2),(3,4),(3,6)))
pairRDD.collect
res29: Array[(Int, Int)] = Array((1,2), (3,4), (3,6))
```

1) keys：返回所有的 key。

```
val p3 = pairRDD.keys
p3.collect
```

2) values：返回所有的 value。

```
val p4 = pairRDD.values
p4.collect
```

3) mapValues(func)：将函数应用到 Pair RDD 中的每个元素上，只改变 value，不改变 key。

```
val p6 = pairRDD.mapValues(x => x*x)
p6.collect
```

4) flatMapValues(func)：传入(K, U)对，传出(K,TraversableOnce[U])。通过一个 flatMap 函数传递 key-value pair RDD 中的每个 value，而不改变 key 值；这也保留了原始的 RDD 分区。

```
val p7 = pairRDD.flatMapValues(x => (x to 5))
p7.collect
```

5) sortByKey([ascending], [numPartitions])：

这是一个 transformation 操作。按照 key 进行排序，默认是升序。当对(K, V)对的数据集(其中 K 实现 Ordered)调用时，返回一个(K, V)对的数据集(按键升序或降序排序)，按布尔类型的 ascending 参数中指定的顺序。

```
val p5 = pairRDD.sortByKey()
p5.collect

// pairRDD.sortByKey(ascending=false).collect
pairRDD.sortByKey(false).collect
```

6) groupByKey([numPartitions])：这是一个 transformation 操作。它将 RDD 中每个 key 的值分组成一个序列。当对(K, V)对的数据集调用时，返回(K, Iterable<V>)对的数据集。

```
val p2 = pairRDD.groupByKey()
p2.collect
```

7) reduceByKey(func, [numPartitions])，reduceByKey(partitioner, func])：

这是一个 transformation 操作。它按照 key 来合并值(相同 key 的值进行合并)。当对一个(K, V)对的数据集调用时，返回一个(K, V)对的数据集，其中每个 key 的值使用给定的 reduce 函数 func 进行聚合，该 reduce 函数的类型必须是(V,V) => V。

```
val p1 = pairRDD.reduceByKey((x,y) => x + y)
p1.collect
```

8) foldByKey(zeroValue, [numPartitions])(func)，foldByKey(zeroValue, [partitioner])(func)

这是一个 transformation 操作。它使用一个关联函数和一个初始值来合并每个键的值，这个初始值可以任意次数地添加到结果中，并且不能改变结果(例如，列表连接为 Nil，加法为 0，乘法为 1)。

```
val p7 = pairRDD.foldByKey(1)((a,b) => a + b)
p7.collect                              // 返回 Array((1,3), (3,11))
```

9) aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions])：

这是一个 transformation 操作。当对一个(K, V)对的数据集调用时，返回一个(K, U)对的数据集，其中每个 key 的值使用给定的 combine 函数和一个中性的“零”值进行聚合。允许与输入值类型不同的聚合值类型，同时避免不必要的分配。（详细用法在 3.5.6 中讲解）

10) combineByKey(createCombiner, mergeValue, mergeCombiners, numPartitions, mapSideCombine)：

这是一个 transformation 操作。合并相同 key 的值，使用不同的结果类型。（详细用法在 3.5.7 中讲解）

- createCombiner:在第一次遇到 Key 时创建组合器函数，将 RDD 数据集中的 V 类型 value 值转换 C 类型值（V => C）
- mergeValue：合并值函数，再次遇到相同的 Key 时，将 createCombiner 的 C 类型值与这次传入的 V 类型值合并成一个 C 类型值（C,V）=>C
- mergeCombiners:合并组合器函数，将 C 类型值两两合并成一个 C 类型值
- partitioner：使用已有的或自定义的分区函数，默认是 HashPartitioner
- mapSideCombine：是否在 map 端进行 Combine 操作,默认为 true

11) subtractByKey

这是一个 transformation 操作。它返回这样一个 RDD：其中的 pair 对的键 key 只在当前 RDD 中有而在 other RDD 中没有。它有三个重载的方法：

- subtractByKey[W](other)
- subtractByKey[W](other, numPartitions)
- subtractByKey[W](other, partitioner)

12) sampleByKey(withReplacement, fractions, seed)：

这是一个 transformation 操作。返回按 key 采样的 RDD 的一个子集(通过分层采样)。

13) sampleByKeyExact(withReplacement, fractions, seed)：

这是一个 transformation 操作。返回按 key 采样的 RDD 的一个子集(通过分层采样)，对于每一层（具有相同 key 的一组对），包含精确的 math.ceil(numItems * samplingRate)个元素。

14) 连接操作

- join(otherDataset, [numPartitions])：当对类型(K, V)和(K, W)的数据集调用时，返回(K， (V, W))对的数据集，其中包含每个 key 的所有元素对。通过 leftOuterJoin、rightOuterJoin 和 fullOuterJoin 支持外连接。
- leftOuterJoin：左外连接。
- rightOuterJoin：右外连接。
- fullOuterJoin：全外连接。

15) cogroup(otherDataset, [numPartitions])：

这是一个 transformation 操作。当对类型(K, V)和(K, W)的数据集调用时，返回一个(K，(Iterable<V>，Iterable<W>))元组的数据集。这个操作也称为 groupWith。

16) groupWith[W](other)：cogroup 的别名。

这是一个 transformation 操作。groupWith[W1, W2](other1, other2)：cogroup 的别名。当对类型(K, V)、(K, W1)和(K, W2)的数据集调用时，返回一个(K， (Iterable<V>， Iterable<W1>， Iterable<W2>))元组的数据集。

groupWith[W1, W2, W3](other1, other2, other3)：cogroup 的别名。当对类型(K, V)、(K, W1)、(K, W2)和(K, W3)的数据集调用时，返回一个(K， (Iterable<V>， Iterable<W1>， Iterable<W2>， Iterable<W3>))元组的数据集。

17) partitionBy(partitioner)：

这是一个 transformation 操作。返回使用指定分区器分区的 RDD 的一个副本。

18) repartitionAndSortWithinPartitions(partitioner)：

根据给定的分区程序对 RDD 进行重新分区，并在每个结果分区中根据键对记录进行排序。这比调用 repartition 然后在每个分区内排序更有效，因为它可以将排序下推到 shuffle 机制中。

Pair RDD 上的 action 操作

1）countByKey()：

这是一个 action 操作。计算每个 key 的元素数量，将结果收集到一个本地 Map 中（Map[K, Long]）。

```
val stus = List(("计算机系","张三"),("数学系","李四"),("计算机系","王老五"),("数学系","赵老六"))
val rdd1 = spark.sparkContext.parallelize(stus)

val kvRDD1 = rdd1.keyBy(_._1)       // 转换为 pair rdd
```

```
kvRDD1.countByKey                    // action 操作，返回 Map(计算机系  -> 2,  数学系  -> 2)
```

2）collectAsMap()

这是一个 action 操作。将这个 RDD 中的键值对作为 Map 返回给 master。这不会返回一个 multimap(所以如果一个键有多个值，每个键在返回的 map 中只保留一个值)。这个方法只应该在结果数据很小的情况下使用，因为所有的数据都被加载到驱动程序的内存中。

详细 API 说明请参考：

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions

3.5.3 关于 sortByKey

Pair RDD 的 sortByKey 函数作用于 Key-Value 形式的 RDD，并对 Key 进行排序。它是在 org.apache.spark.rdd.OrderedRDDFunctions 中实现的。该函数返回的 RDD 一定是 ShuffledRDD 类型的，因为对源 RDD 进行排序，必须进行 Shuffle 操作，而 Shuffle 操作的结果 RDD 就是 ShuffledRDD。

其实这个函数的实现很优雅，里面用到了 RangePartitioner，它可以使得相应的范围 Key 数据分到同一个 partition 中，然后内部用到了 mapPartitions 对每个 partition 中的数据进行排序，而每个 partition 中数据的排序用到了标准的 sort 机制，避免了大量数据的 shuffle。

请看下面这个示例：

```
val a = sc.parallelize(List("xlw", "snail", "xueai8", "39657", "about"), 2)
val b = sc. parallelize (1 to a.count.toInt , 2)

val c = a.zip(b)            // 拉链操作

c.sortByKey().collect     // 按 key 排序
```

执行以上代码，输出结果如下：

```
Array((39657,4), (about,5), (snail,2), (xlw,1), (xueai8,3))
```

3.5.4 关于 groupByKey

Pair RDD 的 groupByKey 函数以迭代器的形式收集每个 key 的值。顾名思义，groupByKey 函数会把同一个 key 的所有值分到一组中。与 reduceByKey 不同的是，它不会对最终输出进行任何形式的操作，它只是对数据进行分组并以迭代器的形式返回。它是一个 transformation 转换操作，这意味着它的计算是惰性的。

假设，现在有这么一组数据：

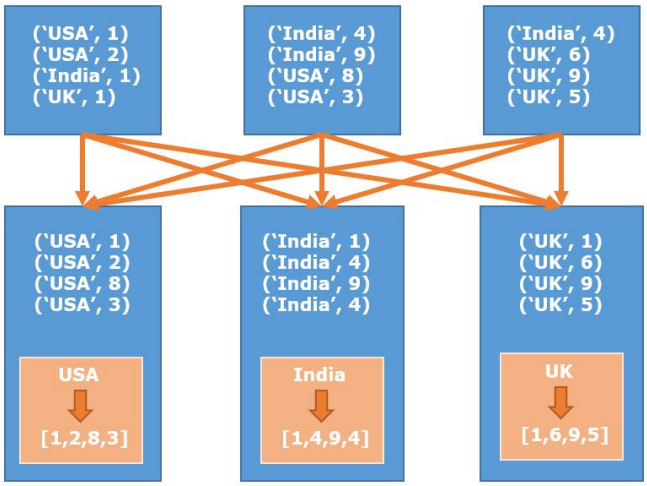
```
val data = Array(("USA", 1), ("USA", 2), ("India", 1),
                 ("UK", 1), ("India", 4), ("India", 9),
                 ("USA", 8), ("USA", 3), ("India", 4),
                 ("UK", 6), ("UK", 9), ("UK", 5)
            )
```

将其构造为具有 3 个分区的一个 RDD：

```
val rdd = sc.parallelize(data, 3)
```

现在，因为在源 RDD 中，每个 key 都可以存在于任何分区中，所以当对这个 RDD 执行 groupByKey 转换时，它需要将同一个 key 所所有数据 shuffle 到单个分区（除非源 RDD 已经按 key 分区了）。这种

shuffling 使这种转换成为一种宽依赖的转换。如下图所示：



这个函数有三个变体：

- groupByKey()：将 RDD 中每个 key 的值分组为单个序列。
- groupByKey(numPartition)：参数用于指定结果 RDD 中的分区数。
- groupByKey(partitioner)：使用 partitioner 在结果 RDD 中创建分区。

【示例】使用 groupByKey 函数对 Pair RDD 进行分组。

```
// 假设，现在有这么一组数据：
val data = Array(("USA", 1), ("USA", 2), ("India", 1),
                 ("UK", 1), ("India", 4), ("India", 9),
                 ("USA", 8), ("USA", 3), ("India", 4),
                 ("UK", 6), ("UK", 9), ("UK", 5)
)

// 将其构造为具有 3 个分区的一个 RDD：
val rdd = sc.parallelize(data, 3)

// 查看分区数
println(s"rdd 的分区数是：${rdd.getNumPartitions}")

// 按 key 对 Pair RDD 中的元素分组
val rdd2 = rdd.groupByKey(2)

// 查看分区数
println(s"rdd2 的分区数是：${rdd2.getNumPartitions}")

// 查看结果
println("\n 分组以后：")
rdd2.collect.foreach(println)
```

执行以上代码，输出结果如下：

```
rdd 的分区数是：3
rdd2 的分区数是：2

分组以后：
(UK,CompactBuffer(1, 6, 9, 5))
(USA,CompactBuffer(1, 2, 8, 3))
```

```
(India,CompactBuffer(4, 9, 1, 4))
```

关于 groupByKey，它具有以下特点：

- groupByKey 是一个 transformation 转换操作，因此它的计算是惰性的；
- 它是一种宽依赖的操作，因为它从多个分区 shuffle 数据并创建另一个 RDD；
- 此操作开销很大，因为它不使用分区本地的组合器（combiner）来减少数据传输；
- 当需要对分组数据进行进一步聚合时，不建议使用；
- groupByKey 总是会导致对 RDD 执行哈希分区。

3.5.5 关于 reduceByKey

Pair RDD 的 reduceByKey 函数使用 reduce 函数合并每个 key 的值，它是一个 transformation 操作，这意味着它是延迟计算的。我们需要传递一个相关函数作为参数，该函数将被应用到源 Pair RDD 上，并创建一个新的 Pair RDD。这个操作是一个宽依赖的操作，因为有可能发生跨分区数据 shuffling。

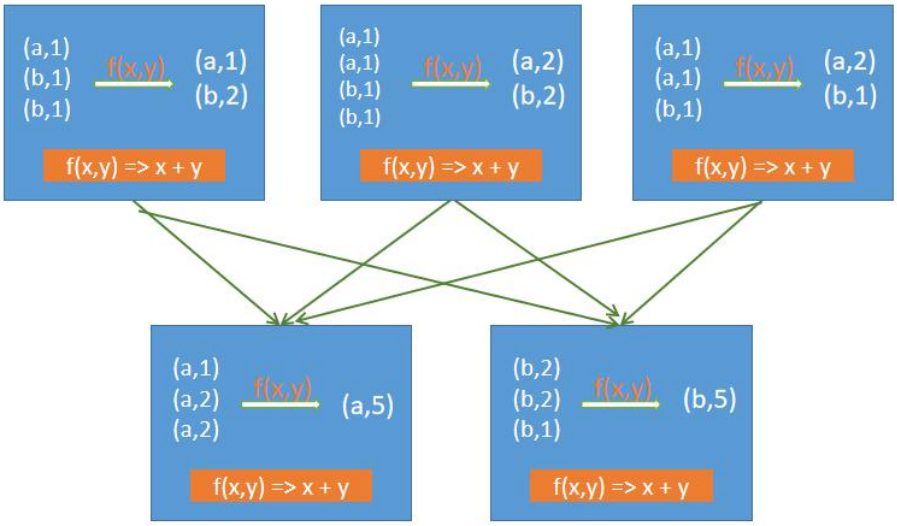
假设，现在有这么一组数据：

```
val data = Array(("a", 1), ("b", 1), ("a", 1), ("a", 1), ("b", 1),
                 ("b", 1), ("a", 1), ("b", 1), ("a", 1), ("b", 1)
               )
```

将其构造为具有 3 个分区的一个 RDD：

```
val rdd = sc.parallelize(data, 3)
```

当 reduceByKey 函数重复地应用于具有多个分区的同一组 RDD 数据时，它首先使用 reduce 函数在本地执行合并，然后跨分区发送记录以准备最终结果。也就是说，在跨分区发送数据之前，它还使用相同的 reduce 函数在本地合并数据，以优化数据转换。如下图所示：



这个 reduceByKey 函数有三个变体：

- reduceByKey(function)：将使用现有的分区器生成散列分区输出。
- reduceByKey(function, [numPartition])：将使用现有的分区器生成散列分区输出。
- reduceByKey(partitioner, function)：使用指定的 Partitioner 对象生成输出。

【示例】使用 reduceByKey 函数对 Pair RDD 进行分组求和。

```
// 假设，现在有这么一组数据：
```

```
val data = Array(("a", 1), ("b", 1), ("a", 1), ("a", 1), ("b", 1),
                 ("b", 1), ("a", 1), ("b", 1), ("a", 1), ("b", 1)
)

// 将其构造为具有 3 个分区的一个 RDD：
val rdd = sc.parallelize(data, 3)

// 在 rdd 上应用 reduceByKey 操作
val rdd2 = rdd.reduceByKey((acc, n) => (acc + n))
rdd2.collect.foreach(println)

println()
// 也可以使用更简洁的方式
val rdd3 = rdd.reduceByKey(_ + _)
rdd3.collect.foreach(println)

println()
// 或者，也可以单独定义 combiner 函数（当逻辑比较复杂时）
def sumFunc(acc:Int, n:Int) =  acc + n
val rdd4 = rdd.reduceByKey(sumFunc)
rdd4.collect.foreach(println)
```

执行以上代码，输出结果如下：

```
(a,5)
(b,5)

(a,5)
(b,5)

(a,5)
(b,5)
```

3.5.6 关于 aggregateByKey

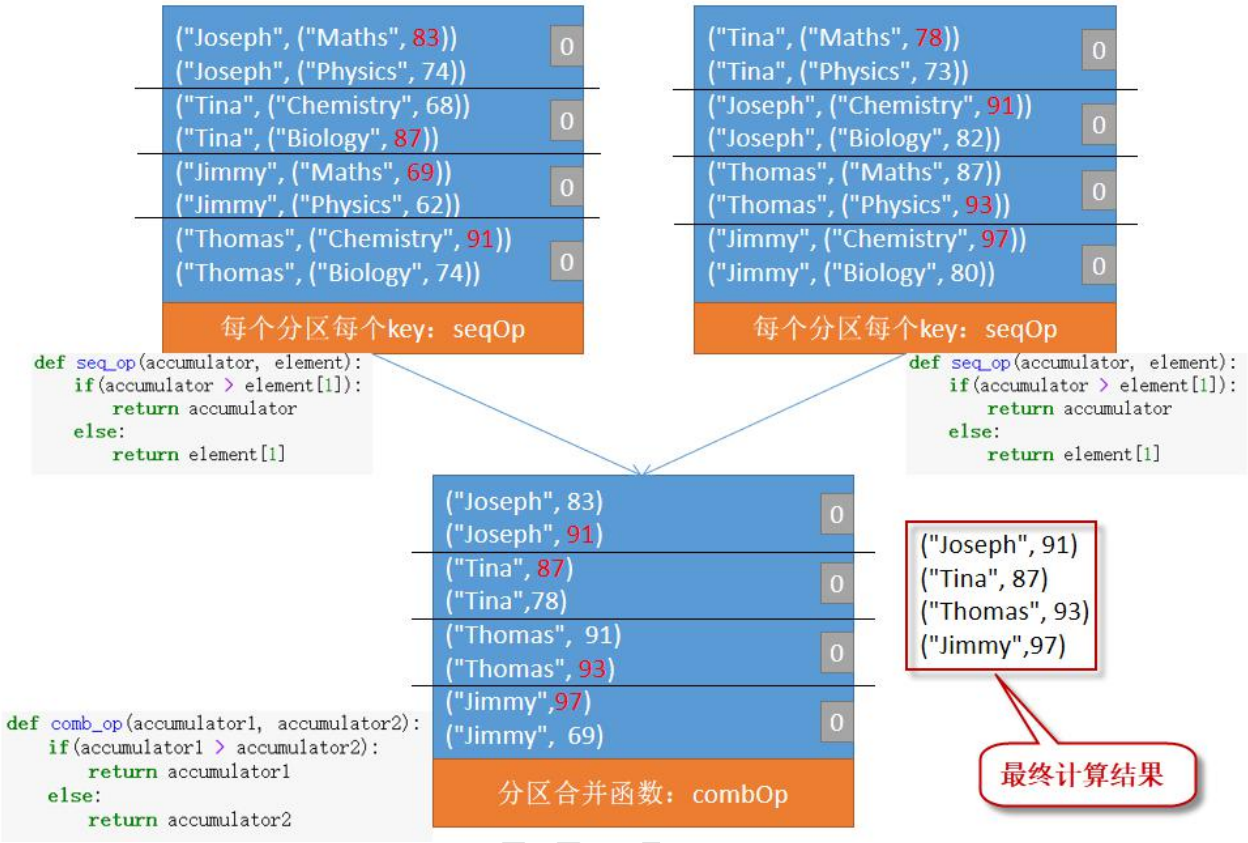
Apache Spark aggregateByKey 函数聚合每个 key 的值，使用给定的 combine 函数和一个中性的“零值”，并为该 key 返回不同类型的值。

这个 aggregateByKey 函数总共接受 3 个参数：

- zeroValue：它是累加值或累加器的初值。如果聚合类型是对所有的值求和，那么它可以是 0。如果聚合目标是找出最小值，这个值可以是 Double.MaxValue。如果聚合目标是找出最大值，这个值可以使用 Double.MinValue。或者，如果我们只是想要一个各自的集合作为每个 key 的输出的话，也可以使用一个空的 List 或 Map 对象。
- seqOp：是聚合单个分区的所有值的操作。它将一种类型[V]的数据转换/合并为另一种类型[U]的序列操作函数。
- combOp：类似于 seqOp，进一步聚合来自不同分区的所有聚合值。它将多个转换后的类型[U]合并为一个单一类型[U]的组合操作函数。

例如，我们有这样的 RDD：PairRDD[String, (String, Double)]。其中，key 是学生姓名，数据类型为 String，值是课程名称和成绩，数据类型为（String,Double）。现在我们要找出每个学生的最好成绩，过

程如下：



【示例】给出每个学生每门功课的分数，请使用 aggregateByKey 函数计算每个学生的最好成绩。

```
// 使用 key-value 对创建 PairRDD studentRDD
val studentRDD = sc.parallelize(Array(
      ("Joseph", "Maths", 83), ("Joseph", "Physics", 74), ("Joseph", "Chemistry", 91),
      ("Joseph", "Biology", 82), ("Jimmy", "Maths", 69), ("Jimmy", "Physics", 62),
      ("Jimmy", "Chemistry", 97), ("Jimmy", "Biology", 80), ("Tina", "Maths", 78),
      ("Tina", "Physics", 73), ("Tina", "Chemistry", 68), ("Tina", "Biology", 87),
      ("Thomas", "Maths", 87), ("Thomas", "Physics", 93), ("Thomas", "Chemistry", 91),
      ("Thomas", "Biology", 74), ("Cory", "Maths", 56), ("Cory", "Physics", 65),
      ("Cory", "Chemistry", 71), ("Cory", "Biology", 68), ("Jackeline", "Maths", 86),
      ("Jackeline", "Physics", 62), ("Jackeline", "Chemistry", 75), ("Jackeline", "Biology", 83),
      ("Juan", "Maths", 63), ("Juan", "Physics", 69), ("Juan", "Chemistry", 64), ("Juan", "Biology", 60)), 4)

// 定义 Seqencial Operation 和 Combiner Operations
// Sequence operation : 从单个分区查找最大成绩
def seqOp = (accumulator: Int, element: (String, Int)) =>
      if(accumulator > element._2) accumulator else element._2

// Combiner Operation : 从所有分区累加器中找出最大成绩
def combOp = (accumulator1: Int, accumulator2: Int) =>
      if(accumulator1 > accumulator2) accumulator1 else accumulator2

// Zero Value: 在我们的情况下，零值将是零，因为我们正在寻找最大的成绩
```

```
val zeroVal = 0
val aggrRDD = studentRDD.map(t => (t._1, (t._2, t._3))).aggregateByKey(zeroVal)(seqOp, combOp)

// 查看输出结果
aggrRDD.collect foreach println
```

输出结果如下：

```
(Tina,87)
(Thomas,93)
(Jackeline,83)
(Joseph,91)
(Juan,69)
(Jimmy,97)
(Cory,71)
```

【示例】在上例的基础上，请修改代码，要求要同时输出每个学生最高成绩及该成绩所属的课程。

```
//
```

输出结果如下：

```
(Tina,(Biology,87))
(Thomas,(Physics,93))
(Jackeline,(Biology,83))
(Joseph,(Chemistry,91))
(Juan,(Physics,69))
(Jimmy,(Chemistry,97))
(Cory,(Chemistry,71))
```

【示例】在上例的基础上，请修改代码，要求计算所有学生的平均成绩。

```
//
```

输出结果如下：

```
(Tina,76.0)
(Thomas,86.0)
(Jackeline,76.0)
(Joseph,82.0)
(Juan,64.0)
(Jimmy,77.0)
(Cory,65.0)
```

aggregateByKey 函数的特点总结如下：

- 性能方面的 aggregateByKey 是一个优化的转换；
- aggregateByKey 是一个宽依赖的转换；
- 当聚合需求加上输入和输出 RDD 类型不同时，我们应该使用 aggregateByKey；
- 当聚合需求加上输入和输出 RDD 类型相同时，我们应该使用 reduceByKey。

3.5.7 关于 combineByKey

Pair RDD 的 combineByKey 转换与 Hadoop MapReduce 编程中的 combiner 非常相似。是一个宽依赖的操作，因为它在最后阶段需要 shuffle。它内部会按分区合并元素。

Pair RDD 的 combineByKey 是一个通用函数，它使用一组自定义的聚合函数组合每个 key 的元素。内部 combineByKey 函数通过应用聚合函数有效地组合了 Pair RDD 分区的值。combineByKey 转换的主要目标是将任何 PairRDD[(K,V)]转换为 RDD[(K,C)]，其中 C 是键 K 下所有值的任何聚合的结果。

Pair RDD 的 combineByKey 函数使用如下三个函数作为参数：

- createCombiner：在第一次遇到 Key 时创建组合器函数，将 RDD 数据集中的 V 类型 value 值转换 C 类型值（V => C）；
- mergeValue：mergeValue：合并值函数，再次遇到相同的 Key 时，将 createCombiner 的 C 类型值与这次传入的 V 类型值合并成一个 C 类型值（C,V）=>C。
- mergeCombiners：合并组合器函数，将 C 类型值两两合并成一个 C 类型值。

createCombiner 函数：

- 这个函数是 combineByKey 函数的第一个参数；
- 它是每个 key 的第一个聚合步骤；
- 当在一个分区中发现任何新的 key 时，它将被执行；
- 这个 lambda 函数的执行在每个单独的值上对一个节点的分区都是局部的；
- 这个函数类似于 aggregateByKey 函数的第一个参数(即 zeroVal)。

mergeValue 函数：

- 第二个函数在将下一个后续值赋给 combiner 时执行；
- 它还在节点的每个分区上本地执行，并组合所有值；
- 这个函数的参数是一个累加器和一个新值；
- 它在现有累加器中组合了一个新值；
- 这个函数类似于 aggregateByKey 转换的第二个参数(即 seqOp)。

mergeCombiners 函数：

- 这是最后的函数，用于组合如何跨分区合并单个 key 的两个累加器(即 combiner)以生成最终的预期结果；
- 参数是两个累加器(即 combiner)；
- 合并来自不同分区的单个 key 的结果；
- 这个函数类似于 aggregateByKey 函数的第三个参数(即 combOp)。

下面我们通一个示例来理解 combineByKey 的用法。

首先，假设我们有一个由 studentName、subjectName 和 marks 构成的 RDD，我们想要得到每个学生的平均成绩。下面是用 combineByKey 变换求解的步骤。

【示例】给出每个学生每门功课的分数，请使用 combinerByKey 函数计算每个学生的平均成绩。

在这个例子中，因为要计算平均成绩，需要做 sum 和 count 聚合。所以这里的 createCombiner 函数应该用一个元组(sum, count)来初始化它。对于初始聚合，它应该是(value,1)。

在这个例子中，mergeValue 有一个累加器元组(sum, count)。因此，每当我们得到一个新值，marks 将被添加到第一个元素，而第二个值(即计数器)将增加 1。

在这个例子中，mergeCombiners 合并来自各个分区的结果(sum, count)。因此，对于同一个 key，需要将各个元组中的 sum 和 count 都累加，得到每个学生的总成绩和总课目数。

代码实现如下所示：

```
// 创建一个 PairRDD，命名为 studentRDD
val studentRDD = sc.parallelize(Array(
      ("Joseph", "Maths", 83), ("Joseph", "Physics", 74), ("Joseph", "Chemistry", 91),
      ("Joseph", "Biology", 82), ("Jimmy", "Maths", 69), ("Jimmy", "Physics", 62),
      ("Jimmy", "Chemistry", 97), ("Jimmy", "Biology", 80), ("Tina", "Maths", 78),
      ("Tina", "Physics", 73), ("Tina", "Chemistry", 68), ("Tina", "Biology", 87),
      ("Thomas", "Maths", 87), ("Thomas", "Physics", 93), ("Thomas", "Chemistry", 91),
      ("Thomas", "Biology", 74), ("Cory", "Maths", 56), ("Cory", "Physics", 65),
      ("Cory", "Chemistry", 71), ("Cory", "Biology", 68), ("Jackeline", "Maths", 86),
      ("Jackeline", "Physics", 62), ("Jackeline", "Chemistry", 75), ("Jackeline", "Biology", 83),
      ("Juan", "Maths", 63), ("Juan", "Physics", 69), ("Juan", "Chemistry", 64),
      ("Juan", "Biology", 60)), 3)

// 定义 createCombiner 函数
def createCombiner = (tuple: (String, Int)) =>
      (tuple._2.toDouble, 1)

// 定义 mergeValue 函数
def mergeValue = (accumulator: (Double, Int), element: (String, Int)) =>
      (accumulator._1 + element._2, accumulator._2 + 1)

// 定义 mergeCombiner 函数
def mergeCombiner = (accumulator1: (Double, Int), accumulator2: (Double, Int)) =>
      (accumulator1._1 + accumulator2._1, accumulator1._2 + accumulator2._2)

// 使用 combineByKey 计算平均成绩
val combRDD = studentRDD.map(t => (t._1, (t._2, t._3)))
                        .combineByKey(createCombiner, mergeValue, mergeCombiner)
                        .map(e => (e._1, e._2._1/e._2._2))

// 查看输出
combRDD.collect foreach println
```

执行以上代码，输出结果如下：

```
(Tina,76.5)
(Thomas,86.25)
(Jackeline,76.5)
(Joseph,82.5)
(Juan,64.0)
(Jimmy,77.0)
(Cory,65.0)
```

combineByKey 转换函数的特点总结如下：

- combineByKey 是一个通用的转换，而 groupByKey、reduceByKey 和 aggregateByKey 转换的内部实现使用了 combineByKey；
- combineByKey 转换可灵活执行 map 或 reduce 端 combine；

- combineByKey 转换的使用更加复杂；
- 总是需要实现三个函数：createCombiner、mergeValue、mergeCombiner；
- combineByKey 是一个转换操作，因此它的计算是惰性的；
- 它是一个宽依赖的操作，因为它在聚合的最后阶段 shuffle 数据并创建另一个 RDD；

3.5.8 Pair RDD 的连接操作

可以对两个 Pair RDD 按 key 进行 join 连接。Spark 提供了类似于关系数据库中的连接，分别是 join、leftOuterJoin、rightOuterJoin、fullOuterJoin。

假设有两个 RDD，分别是{(1,2),(3,4),(3,6)}和{(3,9)}。首先，我们构造两个 RDD：

```
val pairRDD1 = sc.parallelize(Seq((1,2),(3,4),(3,6)))
val pairRDD2 = sc.parallelize(Seq((3,9)))
```

接下来，对两个 RDD 进行转换操作：

1) subtractByKey：接 key 做差集运算。

```
val r1 = pairRDD1.subtractByKey(pairRDD2)
r1.collect
```

```
res73: Array[(Int, Int)] = Array((1,2))
```

2) join：内连接

```
val r2 = pairRDD1.join(pairRDD2)
r2.collect
```

```
res76: Array[(Int, (Int, Int))] = Array((3,(4,9)), (3,(6,9)))
```

3) leftOuterJoin: 左外连接

```
val r3 = pairRDD1.leftOuterJoin(pairRDD2)
r3.collect
```

```
res79: Array[(Int, (Int, Option[Int]))] = Array((1,(2,None)), (3,(4,Some(9))), (3,(6,Some(9))))
```

4) rightOuterJoin：右外连接

```
val r4 = pairRDD1.rightOuterJoin(pairRDD2)
r4.collect
```

```
res82: Array[(Int, (Option[Int], Int))] = Array((3,(Some(4),9)), (3,(Some(6),9)))
```

5) fullOuterJoin：全外连接

```
val r5 = pairRDD1.fullOuterJoin(pairRDD2)
r5.collect
```

```
res10: Array[(Int, (Option[Int], Option[Int]))] = Array((1,(Some(2),None)), (3,(Some(4),Some(9))), (3,(Some(6),Some(9))))
```

6) cogroup: 对来自两个 RDD 的数据按 key 分组。

```
val r6 = pairRDD1.cogroup(pairRDD2)
r5.collect
```

```
res85: Array[(Int, (Iterable[Int], Iterable[Int]))] = Array((1,(CompactBuffer(2),CompactBuffer())), (3,(CompactBuffer(4, 6),CompactBuffer(9))))
```

请执行如下的 join 操作练习，进一步掌握 Pair RDD 的 join 连接操作：

```
val a = sc.parallelize(List("blue", "green", "orange"), 3)
val b = a.keyBy(_.length)

val c = sc.parallelize(List("black", "white", "grey"), 3)
val d = c.keyBy(_.length)

b.join(d).collect

// leftOuterJoin
b.leftOuterJoin(d).collect

// rightOuterJoin
b.rightOuterJoin(d).collect

// fullOuterJoin
b.fullOuterJoin(d).collect
```

3.6 持久化 RDD

Spark 中最重要的功能之一是跨操作在内存中持久化(或缓存)数据集。当持久化一个 RDD 时，每个节点在内存中存储它计算的任何分区，并在该数据集(或从该数据集派生的数据集)上的其他操作中重用它们。这使得将来的操作要快得多(通常超过 10 倍)。缓存是迭代算法和快速交互使用的关键工具。

3.6.1 缓存 RDD

在 Spark 中，RDD 采用惰性求值的机制，每次遇到 action 操作，Spark 都会从头重新计算 RDD 及其所有的依赖。这对于迭代计算而言，代价是很大的，迭代计算经常需要多次重复使用同一组数据。

下面就是多次计算同一个 RDD 的例子。

```
val list = List("Hadoop", "Spark", "Hive")
val input = sc.parallelize(list)
val result = input.map(x => x.toUpperCase)

println(result.count())                          // action 操作，触发一次真正从头到尾的计算
println(result.collect().mkString(","))          // action 操作，触发一次真正从头到尾的计算
```

可以通过持久化（缓存）机制避免这种重复计算的开销。

Cache 机制是 Spark 提供的一种将数据缓存到内存(或磁盘)的机制，主要用途是使得中间计算结果可以被重用。

要缓存 RDD，常用到两个函数，cache()和 persist()。可以使用 persist()方法对一个 rdd 标记为持久化。之所以说“标记为持久化”，是因为出现 persist()语句的地方，并不会马上计算生成 rdd 并把它持久化，而是要等到遇到第一个 action 操作触发真正计算以后，才会把计算结果进行持久化。持久化后的 rdd 分区将会被保留在计算节点的内存中，被后面的 action 操作重复使用。

如果一个数据集被要求参与几个 action，那么持久化该数据集会节省大量的时间、CPU 周期、磁盘

输入/输出和网络带宽。容错机制也适用于缓存分区。当由于节点故障而丢失任何分区时，它将使用血统图重新计算。

下表列举了 Spark 所支持的持久化级别：

持久化级别	内存使用情况	CPU时间	位于内存中	位于磁盘中	说明
MEMORY_ONLY	高	低	是	否	RDD作为反序列化的Java对象存储在JVM中。如果RDD不适合内存，那么一些分区将不会被缓存，并在每次需要它们时动态地重新计算。这是默认级别。
MEMORY_ONLY_SER（Java和Scala）	低	高	是	否	将RDD存储为序列化的Java对象(每个分区一个字节数组)。这比反序列化对象更节省空间，特别是在使用快速序列化器时，但读取时需要更多CPU。
MEMORY_AND_DISK	高	中等	部分	部分	将RDD作为反序列化的Java对象存储在JVM中。如果RDD不适合内存，那么将不适合的分区存储在磁盘上，并在需要的时候从那里读取它们。
MEMORY_AND_DISK_SER（Java和Scala）	低	高	部分	部分	类似于MEMORY_ONLY_SER，但是将不适合内存的分区溢出到磁盘，而不是在每次需要它们时动态地重新计算它们。
DISK_ONLY	低	高	否	是	仅在磁盘上存储RDD分区
MEMORY_ONLY_2, MEMORY_AND_DISK_2等.					与上面的级别相同，但是在两个集群节点上复制每个分区。
OFF_HEAP(实验)					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。

注意：在 Python 中，存储的对象将始终使用 Pickle 库进行序列化，所以是否选择序列化级别并不重要。Python 中可用的存储级别包括 MEMORY_ONLY、MEMORY_ONLY_2、MEMORY_AND_DISK、MEMORY_AND_DISK_2、DISK_ONLY 和 DISK_ONLY_2。

Spark 还会在随机操作(如 reduceByKey)中自动保存一些中间数据，甚至不需要用户调用 persist。这样做是为了避免在节点在转移期间失败时重新计算整个输入。

重写上一示例，加入对 RDD 进行缓存的代码。代码如下所示：

```
val list = List("Hadoop","Spark","Hive")
val input = sc.parallelize(list)
val result = input.map(x => x.toUpperCase)

// 会调用 persist(MEMORY_ONLY)
// 但是，语句执行到这里，并不会缓存 rdd，这时 rdd 还没有被计算生成
result.persist(StorageLevel.MEMORY_ONLY)       // = result.cache()

// 第一次 action 操作，触发一次真正从头到尾的计算，
// 这时才会执行上面的 rdd.cache()，把这个 rdd 放到缓存中
println(result.count())

// 第二次 action 操作，不需要触发从头到尾的计算，只需要重复使用上面缓存中的 rdd
println(result.collect().mkString(","))

// 把持久化的 RDD 从缓存中移除
result.unpersist()
```

如果可用内存不足，Spark 会将持久的分区溢写到磁盘上。开发人员可以使用 unpersist 删除不需要的 RDD。Spark 会自动监控缓存，并使用 LRU(Least Recently Used)算法删除旧分区。

Spark 的缓存不仅能将数据缓存到内存，也能使用磁盘，甚至同时使用内存和磁盘，这种缓存的不

同存储方式，称作“StorageLevel(存储级别)”。可以这样使用：

```
rdd.persist(StorageLevel.MEMORY_ONLY)。
```

cache 方法本质上是 persist(StorageLevel.MEMORY_ONLY)，也就是说 persist 可以指定 StorageLevel，而 cache 不行。Spark 目前支持的存储级别如下：

- NONE (default)
- DISK_ONL
- DISK_ONLY_2
- MEMORY_ONLY (cache 操作使用的级别)
- MEMORY_ONLY_2
- MEMORY_ONLY_SER
- MEMORY_ONLY_SER_2
- MEMORY_AND_DISK
- MEMORY_AND_DISK_2
- MEMORY_AND_DISK_SER
- MEMORY_AND_DISK_SER_2
- OFF_HEAP

其中：

- 2 代表存储份数为 2，也就是有 2 个备份存储。
- SER 代表存储序列化后的数据。
- DISK_ONLY 后面没跟 SER，但其实只能是存储序列化后的数据。

缓存策略(StorageLevel)

RDD 块可以在多个存储(内存、磁盘、堆外)中以序列化或非序列化格式缓存。

- MEMORY_ONLY：数据仅以非序列化格式缓存在内存中
- MEMORY_AND_DISK：数据缓存在内存中。如果没有足够的内存可用，则将从内存中清除的块序列化到磁盘。当重新计算很昂贵且内存资源稀缺时，建议使用这种操作模式。
- DISK_ONLY：数据仅以序列化格式缓存在磁盘上。
- OFF_HEAP：块被缓存到堆外，例如，在 Alluxio[2]上。

上面的缓存策略还可以使用序列化来以序列化格式存储数据。序列化增加了处理成本，但减少了大型数据集的内存占用。将“_SER”后缀附加到上述策略名上。例如，MEMORY_ONLY_SER 、MEMORY_AND_DISK_SER、DISK_ONLY 和 OFF_HEAP 始终以序列化格式写入数据。

数据也可以通过在 StorageLevel 上添加“_2”后缀来复制到另一个节点：例如，MEMORY_ONLY_2、MEMORY_AND_DISK_SER_2。在集群的一个节点(或执行器)出现故障时，复制有助于加速恢复。

如何选择缓存策略？

Spark 的存储级别意味着在内存使用和 CPU 效率之间提供不同的权衡。建议通过以下步骤来选择一个:

- 如果 RDDs 适合默认存储级别(MEMORY_ONLY)，那么就保留它们。这是 CPU 效率最高的选项，允许 RDDs 上的操作尽可能快地运行。
- 如果没有，可以尝试使用 MEMORY_ONLY_SER 并选择一个快速序列化库，以使对象更节省空间，但访问速度仍然相当快。(Java 和 Scala)

- 不要溢出到磁盘，除非计算数据集的函数非常昂贵，或者它们过滤了大量数据。否则，重新计算分区的速度可能与从磁盘读取分区的速度一样快。
- 如果希望快速恢复故障(例如，如果使用 Spark 为来自 web 应用程序的请求提供服务)，请使用复制的存储级别。通过重新计算丢失的数据，所有存储级别都提供了完全的容错能力，但是复制的存储级别允许在 RDD 上继续运行任务，而无需等待重新计算丢失的分区。

Spark 自动监视每个节点上的缓存使用情况，并以最近最少使用(LRU)的方式删除旧的数据分区。如果希望手动删除一个 RDD，而不是等待它从缓存中删除，那么可以使用 RDD.unpersist()方法。

3.6.2 检查点 RDD

通过链接任意数量的 transformations，RDD lineage 可以任意增长。Spark 提供了一种方法，可以将整个 RDD 持久化到稳定的存储器中，存储的数据包括 RDD 计算后的数据和分区器。然后，在发生节点故障时，Spark 不需要从一开始就重新计算丢失的 RDD 碎片，而是从存储的快照那里开始计算 lineage 中其余的部分。这个特性称为“检查点(check point)”。

简单来说，checkpoint（检查点）是一种截断 RDD 依赖链并把 RDD 数据持久化到存储系统(通常是 HDFS 或本地)的过程。它的主要作用是截断 RDD 依赖关系，防止任意增长的 lineage 导致的堆栈溢出。在检查点之后，RDD 的依赖项，以及它的父 RDD 的信息会被擦除，因为重新计算不再需要它们了。

必须先调用 SparkContext.setCheckpointDir()方法来设置保存数据的目录，然后通过调用 checkpoint 操作来对 RDD 设置检查点，这时该 RDD 将被保存到检查点目录中的一个文件中，所有对其父 RDD 的引用将被删除。

RDD checkpoint 的调用必须在这个 RDD 上执行任何作业之前。当在 RDD 上调用 checkpoint()方法时，仅是将此 RDD 标记为检查点。必须在 RDD 上调用了 action 操作才能完成检查点。

【示例】设置 RDD checkpoint。

```
val sc = spark.sparkContext                                  // 获取 SparkContext 实例

sc.checkpoint("hdfs://xueai8:8020/ck/rdd")                   // 设置 RDD 的检查点目录

val data = sc.textFile("hdfs://master:8020/input")           // 加载数据源

val rdd = data.map(...).reduceByKey(...)                     // 执行一系列 transformation

rdd.checkpoint                  // 标记对 RDD 做 checkpoint,不会真正执行,直到遇到第一个 action 算子
rdd.count                       // 第一个 action 算子，触发之前的代码执行
```

3.6.3 Checkpoint vs Cache

Cache 用于缓存，采用临时保存，Executor 挂掉会导致数据丢失，但是数据可以重新计算。

Checkpoint 用于截断依赖链，reliable 方式下 Executor 挂掉不会丢失数据，数据一旦丢失不可恢复。

尽管 Spark 会自动管理(包括创建和回收)cache 和 persist 持久化的数据，但是 checkpoint 持久化的数据需由用户自己管理。checkpoint 会清除 RDD 的血统信息，避免血统过长导致序列化开销增大，而 cache 和 persist 不会清除 RDD 的血统。

3.7 数据分区

数据分区（partition）是 Spark 中的重要概念，是 Spark 在集群中的多个节点之间划分数据的机制。它是 RDD 的最小单元，RDD 是由分布在各个节点上的分区组成的。Spark 使用分区来管理数据，分区的数量决定了 task 的数量，每个 task 对应着一个数据分区。这些分区有助于并行化分布式数据处理。

默认情况下，为每个 HDFS block 块创建一个分区，该分区默认为 128MB（Spark 2.x）。例如，当从本地文件系统加载一个文本文件到 Spark 时，文件的内容被分成几个分区，这些分区均匀地分布在集群中的节点上。在同一个节点上可能会出现不止一个分区。所有这些分区的总和形成了 RDD。这就是“弹性分布数据集”中“分布”一词的来源。

在下面这张图中，加载的数据集被分割为 3 个分区，分别存储在集群的 3 个节点上。每个 RDD 维护一个分区的列表和一个可选的首选位置列表，用于计算分区。可以从 RDD 的 partitions 字段获得 RDD 分区的列表。它是一个数组 Array，所以可以通过读取 RDD 的 partitions.size 字段来获得该 RDD 分区的数量。

使用下面的代码查看 RDD 的分区数量：

```
// 构造一个 pair rdd
val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3)))

// 查看分区数量
pairs.partitions.size
```

分区的数量可以在创建 RDD 时指定。例如，在调用 textFile 和 parallelize 方法创建 RDD 时，可手动指定分区个数。方法签名如下：

```
deftextFile(path: String, minPartitions: Int = defaultMinPartitions): RDD[String]
defparallelize[T](seq: Seq[T], numSlices: Int = defaultParallelism)(implicit arg0: ClassTag[T]): RDD[T]
```

如果未指定 RDD 的分区数量，则在创建 RDD 时，Spark 将使用默认分区数，默认值为“spark.default.parallelism”配置的参数。

默认情况下，Spark 尝试从 RDD 附近的节点将数据读入 RDD。因为 Spark 通常访问分布式分区数据，为了优化转换（transformation）操作，它创建分区来保存数据块。

Spark 试图维护每个分区的首选位置列表。分区的首选位置是一个分区的数据所驻留的主机名或 executors 的列表，这样计算就可以更靠近数据了。这些信息可以获取然后用于基于 HDFS 数据（HadoopRDD）的 RDD 和缓存的 RDD。

如果 Spark 获得了首选位置的列表，Spark 调度程序会试着在数据实际存在的执行器上运行任务，这样就不需要进行数据传输。这对性能有很大的影响。

有五个级别的数据本地化配置：

- PROCESS_LOCAL：在缓存分区的 execute 上执行一个 task 任务。
- NODE_LOCAL：在分区可用的节点上执行一个任务。
- RACK_LOCAL：如果机架信息在集群中可用（目前仅在 YARN 上），则在与分区相同的机架上执行任务。
- NO_PREF：没有任何首选的位置与任务（task）相关联.
- ANY：默认。

3.7.1 调整 RDD 分区数

RDD 从数据源生成的时候，数据通常是随机分配到不同的分区或者保持数据源的分区。RDD 分区数的多少，会对 Spark 程序的执行产生一定的影响。因为除了影响整个集群中的数据分布之外，它还直接决定了将要运行 RDD 转换的任务的数量。

如果分区数量太少，则直接影响是集群计算资源不能被充分利用。例如分配 8 个核，但分区数量为 4，则将有一半的核没有利用到。此外，因为数据集可能会变得太大，当无法装入 executor 的内存中时可能会导致内存问题。

如果分区数量太多，虽然计算资源能够充分利用，但会导致 task 数量过多，而 task 数量过多会影响执行效率，主要是 task 在序列化和网络传输过程带来较大的时间开销。

根据 Spark RDD Programming Guide 上的建议，集群节点的每个核分配 2-4 个分区比较合理，也就是说，建议将分区数设置为集群中 CPU 核数的三到四倍。

> One important parameter for parallel collections is the number of *partitions* to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to `parallelize` (e.g. `sc.parallelize(data, 10)`). Note: some places in the code use the term slices (a synonym for partitions) to maintain backward compatibility.

在某些情况下，为了更有效地分配工作负载或避免内存问题，需要显式地重新划分 RDD。例如，从 HDFS 上加载压缩文件时，因为压缩文件不能被分片，所以只能有一个 RDD 分区，即使在 sc.textFile("xxx.gz", 100)中指定了分区数。这里，就需要对 rdd 调用 repartition(N)方法进行重分区。

最主要的两种调整数据分区的方法是 coalesce 和 repartition 函数。从函数接口可以看到，reparation 是直接调用 coalesce(numPartitions, shuffle=True)，不同的是，reparation 函数可以增加或减少分区数量，调用 repartition 函数时，还会产生 shuffle 操作（该操作与 HiveQL 的 DISTRIBUTE BY 操作类似。）。而 coalesce 函数可以控制是否 shuffle，但当 shuffle 为 false 时，只能减小 partition 数，而无法增大。

函数 coalesce 用于减少或增加分区的数量。完整的方法签名如下：

```
coalesce (numPartitions: Int, shuffle: Boolean = false)
```

第二个（可选）布尔参数 shuffle 指定是否应该执行 shuffle(默认为 false)。

coalesce 转换用于更改分区的数量。它可以触发的 RDD shuffling，这取决于 shuffle 标志(默认禁用，即 false)。（repartition(N)方法相当于 coalesce(N, true)方法）。在减少分区时，coalesce 并没有对所有数据进行移动，仅仅是在原来分区的基础之上进行了合并而已，这样的操作可以减少数据的移动，所以效率较高。

【示例】对 RDD 进行重分区。

```
// 在创建时，指定分区个数
val rdd1 = sc.parallelize(Seq(1,2,3,4,5,6,7,8), 4)
rdd1.collect

rdd1.partitions.size                    // rdd1 的分区数量，目前为 4

// 对于通过转换得到的新 RDD，直接调用 repartition 方法重新分区
val rdd2 = rdd1.map((x) => x*x)         // 转换得到 rdd2
rdd2.collect

val rdd3 = rdd2.repartition(8)          // 重新分区，得到 rdd3
rdd3.partitions.size                    // rdd3 的分区数量，这时为 8
```

输出结果如下所示：

3.7.2 使用数据分区器

当需要对 Pair RDD 进行重分区时，RDD 的分区由 org.apache.spark.Partitioner 对象执行，该分区器对象将一个分区索引赋给每个 RDD 元素（在每个 key 和分区 ID 间建立起映射，分区 ID 的值从 0 到 numPartitions - 1）。Spark 内置提供了两个 Partitioner 分区器实现，分别是：

- org.apache.spark.HashPartitioner
- org.apache.spark.RangePartitioner。

HashPartitioner 是 Spark 的默认分区器，它基于一个元素的 Java 散列码（或者是 Pair RDDs 中的 key 的散列码）计算的分区索引。计算公式如下：

```
partitionIndex = key hashCode % numberOfPartitions
```

分区索引是准随机的；因此，分区很可能不会完全相同大小。然而，在具有相对较少分区的大型数据集中，该算法可能会在其中均匀地分布数据。

当使用 HashPartitioner 时，数据分区的默认数量是由 Spark 配置参数“spark.default.parallelism”决定的。如果该参数没有被用户指定，那么它将被设置为集群中的核的数量。

RangePartitioner 将已排序的 RDD 的数据划分为大致相等的范围。它对传递给它的 RDD 的内容进行了采样，并根据采样数据确定了范围边界。一般不太可能直接使用 RangePartitioner。

partitionBy()方法

当在 Pair RDD 上调用 partitionBy 方法进行重分区时，需要向它传递一个参数：期望的 Partitioner（分区器）对象。如果分区器与之前使用的分区器相同，则保留分区，RDD 保持不变。否则，就会安排一次 shuffle，并创建一个新的 RDD。

【示例】在调用 partitionBy 方法进行重分区时，使用指定的分区器。

```
val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3)))
pairs.partitioner                       // 查看所使用的分区器
pairs.partitions.size                   // 初始分区数量

val repairs = pairs.repartition(4)      // 重新分区
repairs.partitions.size                 // 重分区之后的数量

import org.apache.spark.HashPartitioner

val partitionedRDD = pairs.partitionBy(new HashPartitioner(2))    // 使用指定的分区器
partitionedRDD.persist()                // 持久化，以便后续操作重复使用 partitioned
partitionedRDD.partitioner              // 查看所使用的分区器
partitionedRDD.partitions.size          // 查看分区数量
```

上面代码中，传给 partitionBy()的参数值 2，代表分区的数量，它将控制有多少并行的 tasks 执行未来的 RDD 上的操作(如 join)。一般来说，这个值至少与集群中核的数量一样多。

只有当一个数据集在面向 key 的操作中被重用多次的情况下(例如 join 操作)，控制分区才有意义。例如，不带 partitionBy()的 join 过程如下图所示：

而使用了 partitionBy()的 join 过程如下图所示：

另外，partitionBy 也经常用于存储 RDD 时控制输出的结果文件。因为 RDD 的每个分区对应一个输出文件，所以可以通过 partitionBy 来调整分区，从而控制输出结果文件的数量。

自定义数据分区器

当需要精确地在分区中放置数据时，也可以自定义分区器。自定义分区只可以在 Pair RDD 上使用。

大多数 Pair RDD 转换有两个额外的重载方法：一个接受额外的 Int 参数（所需的分区数量），另一个则接受一个（定制）Partitioner 类型的附加参数。其中第一个方法使用默认的 HashPartitioner。例如，下面两行代码是相等的，因为它们都应用了 100 个分区的 HashPartitioner：

```
rdd.foldByKey(afunction, 100)          // 使用默认的 HashPartitioner
rdd.foldByKey(afunction, new HashPartitioner(100))
```

如果 Pair RDD 转换没有指定一个分区器，那么所使用的分区数量将是父 RDD（转换为这个分区的 RDD）的最大分区数。如果父 RDD 中没有一个定义了分区器，那么将使用 HashPartitioner，分区数由 spark.default.parallelism 参数指定。

另一种改变 Pair RDD 中分区之间数据的默认位置的方法是使用默认的 HashPartitioner，但是根据某种算法更改 key 的散列码(hash code)。

【示例】使用自定义分区器对 Pair RDD 进行重分区，使得 Pair RDD 中所有偶数写到一个输出文件，所有奇数写到另一个输出文件。

```
// 自定义分区器
class UsridPartitioner(numParts:Int) extends org.apache.spark.Partitioner{

    //覆盖分区数
    override def numPartitions: Int = numParts

    //覆盖分区号获取函数
    override def getPartition(key: Any): Int = {
        key.toString.toInt%10          // 取 key 的最后一位数字
    }
}

// 构造一个 RDD
val rdd1 = spark.sparkContext.parallelize(1 to 100000)
rdd1.getNumPartitions

// 使用自定义的分区器进行重分区，并指定分区数量
val rdd2 = rdd1.map((_,null)).partitionBy(new NumberPartitioner(2))
rdd2.getNumPartitions

// 将结果输出到文件中存储
rdd2.map(_._1).saveAsTextFile("file:///home/hduser/data/spark/files-output2")

// 加载上面存储的结果文件 1，会发现都是偶数
sc.textFile("file:///home/hduser/data/spark/files-output2/part-00000")
  .map(_.toInt)
  .takeOrdered(10)
```

```
// 加载上面存储的结果文件 2，会发现都是奇数
sc.textFile("file:///home/hduser/data/spark/files-output2/part-00001")
  .map(_.toInt)
  .takeOrdered(10)
```

也可以在终端窗口中使用如下的 Linux 命令来查看生成的结果文件：

```
$ ls /home/hduser/data/spark/files-output2
$ head -10 /home/hduser/data/spark/files-output2/part-00000
$ head -10 /home/hduser/data/spark/files-output2/part-00001
```

【示例】使用自定义分区器对 Pair RDD 进行重分区，使得 Pair RDD 根据 key 值的最后一位数字，写到不同的文件。

```
//自定义分区器，需继承 Partitioner 类
class UsridPartitioner(numParts:Int) extends org.apache.spark.Partitioner{
    //覆盖分区数
    override def numPartitions: Int = numParts
    //覆盖分区号获取函数
    override def getPartition(key: Any): Int = {
        key.toString.toInt%10          // 取 key 的最后一位数字
    }
}

//模拟 5 个分区的数据
val data=sc.parallelize(1 to 10,5)

//根据尾号转变为 10 个分区，分写到 10 个文件
val result = data.map((_,1)).partitionBy(new UsridPartitioner(10))

// 保存结果
result.saveAsTextFile("/data/spark_demo/rdd/partition-output")
```

3.7.3 避免不必要的 shuffling

Spark shuffle 是一种重新分配或重新分区数据的机制，以便数据在不同的分区上分组，根据数据大小，可能需要使用 spark.sql.shuffle.partitions 配置或通过代码来减少或增加 RDD/DataFrame 的分区数量。

当需要将来自多个分区的数据组合起来以构建新的分区时，就会发生 shuffle。例如，当按 key 对元素进行分组时，Spark 需要检查 RDD 的所有分区，找到具有相同 key 的元素，然后对它们进行物理分组，从而形成新的分区。

Spark 中的某些操作会触发 shuffle 事件，例如 gropByKey()、reducebyKey()、join()、union()、groupBy()、aggregateByKey()等转换操作。在 shuffle 之前和之后的任务分别被称为 map 和 reduce 任务。map 任务的结果被写到中间文件（通常只针对操作系统的文件系统缓存），并通过 reduce 任务读取。Spark shuffle 是一项昂贵的操作，因为它在执行器之间甚至在集群的工作节点之间移动数据，涉及磁盘 I/O、数据序列化和反序列化以及网络 I/O，所以在 Spark 作业中尽量减少 shuffle 的次数是很重要的。当在 Spark 作业上遇到性能问题时，应该查看涉及转换的 Spark 转换。

例如，在创建 RDD 时，Spark 并不需要将所有 key 的数据存储在一个分区中，因为在创建 RDD 时，我们无法为数据集设置键。因此，当我们运行 reduceByKey()操作来聚合 key 上的数据时，Spark 会执行以下操作：

- Spark 首先在所有分区上运行 map 任务，为单个 key 将所有值分组。
- map 任务的结果保存在内存中。
- 当结果与内存不匹配时，Spark 将数据存储到磁盘中。
- Spark 将 map 后的数据跨分区进行 shuffle，有时还会将 shuffle 后的数据存储到磁盘中，以便在需要重新计算时重用。
- 运行垃圾回收。
- 最后基于 key 在每个分区上运行 reduce 任务。

虽然 reduceByKey()会触发数据 shuffle，但不会改变分区数，因为子 RDD 从父 RDD 继承了分区大小。

尽管大多数 RDD 转换不需要进行 shuffle，但在特定条件下其中一些转换会进行 shuffle。因此，为了最小化 shuffle 出现的次数，需要了解这些条件。

- 在明确改变分区的时候会进行 shuffling。当使用一个自定义分区器时，总是会引发 shuffling。当在转换过程中使用一个与之前的 HashPartitioner 不同的 HashPartitioner 时（分区数量不同），也会引起 shuffling。例如，下面的代码总是会引起 shuffling：

```
rdd.aggregateByKey(zeroValue, 100)(seqFunc, comboFunc).collect()
rdd.aggregateByKey(zeroValue, new CustomPartitioner())(seqFunc, comboFunc).collect()
```

- 由删除分区引起的 shuffle。有时候，转换会导致 shuffle，尽管使用的是默认的分区器。例如，在下面的代码中，第二行不会引起 shuffle，但是第三行会：

```
val rdd:RDD[Int] = sc.parallelize(1 to 10000)

rdd.map(x => (x, x*x)).map(_.swap).count()                    // 不会引起 shuffle
rdd.map(x => (x, x*x)).reduceByKey((v1, v2)=>v1+v2).count()   // 会引起 shuffle
```

下面列出了会引起 shuffle 的转换操作：

- 会引起 RDD 的分区器改变的 Pair RDD 转换：aggregateByKey, foldByKey, reduceByKey, groupByKey, join, leftOuterJoin, rightOuterJoin, fullOuterJoin, cogroup, subtractByKey；
- RDD 转换：subtract, intersection, 以及 groupWith；
- sortByKey ：总是会引起一个 shuffle；
- 重分区操作，如 repartition、partitionBy 或 coalesce(shuffle=true)。

注意：countByKey()不会引起 shuffle 操作。

影响 shuffling 的参数

Spark 有两个 shuffle 实现：基于 sort 排序的实现和基于 hash 的实现。从 Spark 1.2 版本开始默认的是基于 sort 排序的 shuffle，因为它的内存更高效，文件更少。可以通过设置“spark.shuffle.manager”参数的值为 hash 或 sort 来定义要使用哪个 shuffle 实现。

“spark.shuffle.consolidateFiles”参数指定在一个 shuffle 期间是否要合并中间文件。出于性能上的考虑，如果使用 ext4 或 XFS 文件系统，建议将这个参数设为 true（默认值是 false）。

一般来说，shuffling 可能需要大量的内存用于聚合和 join 分组。设置“spark.shuffle.spill”参数指定用于这些任务的内存数量是否应该被限制（默认为 true）。这种情况下，任何多余的数据都会溢写到磁盘上。对于内存的限制由参数“spark.shuffle.memoryFraction”指定(默认是 0.2)。此外，“spark.shuffle.spill.compress”参数告诉 Spark 是否为溢写的数据使用压缩（默认情况下也是 true）。

溢写阈值不应该设得太高，否则会导致内存溢出异常。但是如果溢写阈值设得太低了，溢写就会频繁发生，所以找到一个好的平衡点是很重要的。在大多数情况下，保持默认值应该就很好。

此外，还有一些参数也很有用：

- spark.shuffle.compress：指定是否压缩中间文件(默认是 true)。
- spark.shuffle.spill.batchSize：指定当溢写到磁盘时将被序列化或反序列化的对象的数量。默认值是 10,000。
- spark.shuffle.service.port：如果启用了外部 shuffle 服务，则指定服务器将侦听的端口。

3.7.4 基于数据分区的操作

Spark 提供了一种方法，可以将一个函数应用于整个 RDD，而不是单独地应用于每个分区。我们可以可以重写其中许多方法，只在分区中映射数据，从而避免 shuffle。作用在分区上的 RDD 操作有：mapPartitions，mapPartitionsWithIndex，以及 glom。

下表列出了 Spark 提供的基于分区的操作函数：

函数名	被调用	返回内容	在RDD[T]上的函数签名
mapPartitions()	该分区内元素的迭代 器	返回元素的迭代器	f: (Iterator[T]) -> Iterator[U]
mapPartitionsWithIndex()	集成分区号和该分区内元素的迭代器	返回元素的迭代器	f: (Int, Iterator[T]) -> Iterator[U]
foreachPartition()	元素的迭代器	无	f: (Iterator[T]) -> Unit

各函数的详细说明如下：

- mapPartitions：接受 map 函数，但是函数必须有签名 Iterator T=>Iterator U。它可以用于在每个分区中迭代元素，并为新的 RDD 创建分区。
- mapPartitionsWithIndex：不同之处在于它的 map 函数也接受分区的索引：(Int，Iterator T)=Iterator U。然后，分区的索引就可以在 map 函数中使用。

这两个函数可以将输入 Iterator 转换为带有 Scala Iterator 函数的新迭代器。这两个转换都接受一个额外的可选参数 preservePartitioning，默认是 false。如果它被设置为 true，新的 RDD 将保留父 RDD 的分区。如果它被设置为 false，那么分区器将被移除，以及我们之前讨论的所有结果。

mapPartitions 可以帮助更有效地解决一些问题。例如，如果 map 函数涉及到昂贵的设置（比如打开数据库连接），那么在每个分区上执行一次比每个元素一次要好得多。

当基于每个分区操作时，Spark 为函数提供了一个 Iterator(该分区中的元素)。对于要返回的值，返回一个 Iterable（结果的迭代器）。

用 GLOM 变换收集分区数据

glom 将每个分区的元素聚集到一个数组中，并以这些数组作为元素返回一个新的 RDD。新 RDD 中元素的数量等于其分区的数量。在这个过程中，分区器会被删除。

在下面的示例代码中，创建一个具有 30 个分区的 RDD 并对其执行 glom 转换。新 RDD 中的数组对象的计数，包含来自每个分区的数据，也是 30。代码如下：

```
// 用随机生成 500 个 100 以内的整数构造一个 list
val list = List.fill(500)(scala.util.Random.nextInt(100))

// 构造一个 RDD，并用 glom 方法收集分区数据
val rdd = sc.parallelize(list, 30).glom()
println
```

```
// 返回 RDD 数据
rdd.collect()

// 统计 RDD 中元素的数量
rdd.count()
```

输出结果如下所示：

```
list: List[Int] = List(88, 59, 78, 94, 34, 47, 49, 31, 84, 47, ...)
rdd: org.apache.spark.rdd.RDD[Array[Int]] = MapPartitionsRDD[0]
res0: Array[Array[Int]] = Array(Array(88, 59, 78, 94,...), ...)
res1: Long = 30
```

可以看出，glom 方法将每个分区中的元素聚集到一个数组中，并以这些数组作为新 RDD 的元素。因为有 30 个分区，所以新 RDD 中的元素个数是 30 个。glom 可以作为一种快速的方法将所有 RDD 的元素放到一个数组中。

使用 foreachPartition 执行分区迭代

```
val rdd=sc.parallelize(Seq(1,2,3,4,5),3)

rdd.foreachPartition(partiton=>{          // iterator 只能被执行一次
        partiton.foreach(line=>{
          // save(line)   落地数据
        })
    })
```

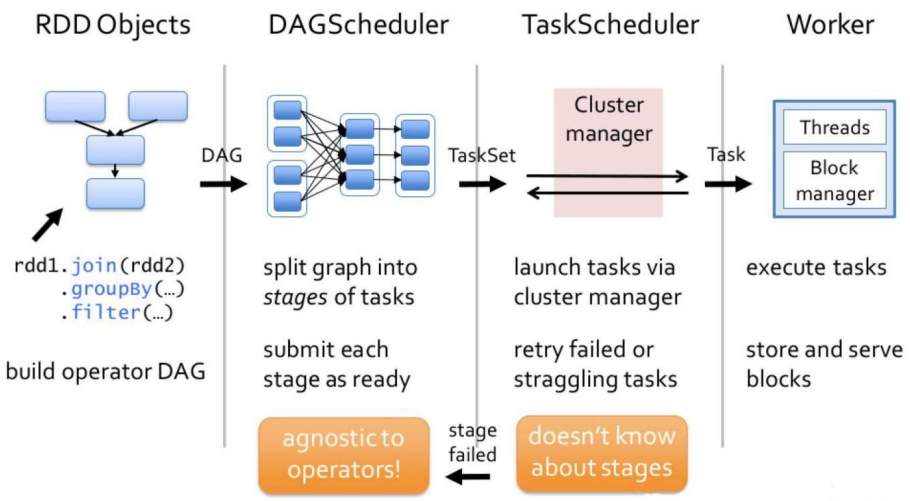
```
val rdd = sc.parallelize(Seq(1,2,3,4,5),3)

rdd.mapPartitions(partiton => {
        partiton.map(line=>{
          // save line
        })
    })
rdd.count()      //   需要 action，来触发执行
```

3.8 理解代码执行过程

Spark 的执行模型是基于 directed acyclic graphs(DAGs) - 有向无环图的。其中 RDD 是顶点，依赖是边。每次在一个 RDD 上执行一个转换时，新创建一个新的顶点(一个新的 RDD)和一条新的边(一个依赖)。新的 RDD 依赖于旧的 RDD，因此边的方向是从子 RDD 到父 RDD。这个依赖图也称为 RDD lineage（RDD 血统）。

Spark RDD 调度过程

如下图所示，Spark 对 RDD 执行调度的过程，创建 RDD 并生成 DAG，由 DAGScheduler 分解 DAG 为包含多个 Task（即 TaskSet）的 Stages，再将 TaskSet 发送至 TaskScheduler，由 TaskScheduler 来调度每个 Task，并分配到 Worker 节点上执行，最后得到计算结果。



当程序创建RDD和应用操作符代码时，Spark会创建一个计算图。当用户运行一个Action(比如collect)时，图被提交给DAG Scheduler。DAG Scheduler将操作符图分为(map和reduce)两个阶段(stages)。阶段由基于输入数据分区的任务组成。DAG Scheduler将操作符连接在一起(形成管道)以优化图。例如，多个map操作符可以在一个stage中进行调度。这种优化是Spark性能的关键。DAG Scheduler的最终结果是一组stages。这些stages被传递给Task Scheduler。Task Scheduler通过集群管理器(Spark Standalone/Yarn/Mesos)启动任务(Task)。Task Scheduler不知道stages之间的依赖关系。

以下面的代码片段为例，来理解代码的执行过程。

```
val tokenized = sc.textFile("/data/spark_demo/rdd/wc.txt").flatMap(_.split(" "))
val wordCounts = tokenized.map((_, 1)).reduceByKey(_ + _)
wordCounts.collect
```

Stage

例如，Spark 在对 Job 中的所有操作划分 Stage 时，一般会按照倒序进行，依据 RDD 之间的依赖关系（宽依赖或窄依赖）进行划分。即从 Action 开始，当遇到窄依赖类型的操作时，则划分到同一个执行阶段；遇到宽依赖操作，则划分一个新的执行阶段，且新的阶段为之前阶段的 Parent，之前的阶段称作 Child Stage，然后依次类推递归执行。Child Stage 需要等待所有的 Parent Stage 执行完之后才可以执行，这时 Stage 之间根据依赖关系构成了一个大粒度的 DAG。

如下图所示，为一个复杂的 DAG Stage 划分示意图：

上图为一个 Job，该 Job 生成的 DAG 划分成了 3 个 Stage。上图的 Stage 划分过程是这样的：从最后的Action开始，从后往前推，当遇到操作为NarrowDependency时，则将该操作划分为同一个Stage，当遇到操作为ShuffleDependency时，则将该操作划分为新的一个Stage。

Task

Task 为一个 Stage 中的一个执行单元，也是 Spark 中的最小执行单元，一般来说，一个 RDD 有多少个 Partition，就会有多少个 Task，因为每一个 Task 只是处理一个 Partition 上的数据。在一个 Stage 内，所有的 RDD 操作以串行的 Pipeline 方式，由一组并发的 Task 完成计算，这些 Task 的执行逻辑完全相同，只是作用于不同的 Partition。每个 Stage 里面 Task 的数目由该 Stage 最后一个 RDD 的 Partition 个数决定。

Spark 中 Task 分为两种类型，ShuffleMapTask 和 ResultTask，位于最后一个 Stage 的 Task 为 ResultTask，其他阶段的属于 ShuffleMapTask。ShuffleMapTask 和 ResultTask 分别类似于 Hadoop 中的 Map 和 Reduce。

Spark 中主要有两种调度器：DAGScheduler 和 TaskScheduler，DAGScheduler 主要是把一个 Job 根据 RDD 间的依赖关系，划分为多个 Stage，对于划分后的每个 Stage 都抽象为一个由多个 Task 组成的任务集（TaskSet），并交给 TaskScheduler 来进行进一步的任务调度。TaskScheduler 负责对每个具体的 Task 进行调度。

DAGScheduler

当创建一个 RDD 时，每个 RDD 中包含一个或多个分区，当执行 Action 操作时，相应的产生一个 Job，而一个 Job 会根据 RDD 间的依赖关系分解为多个 Stage，每个 Stage 由多个 Task 组成（即 TaskSet），每个 Task 处理 RDD 中的一个 Partition。一个 Stage 里面所有分区的任务集合被包装为一个 TaskSet 交给 TaskScheduler 来进行任务调度。这个过程由是由 DAGScheduler 来完成的。DAGScheduler 对 RDD 的调度过程如下图所示：

TaskScheduler

DAGScheduler 将一个 TaskSet 交给 TaskScheduler 后，TaskScheduler 会为每个 TaskSet 进行任务调度，Spark 中的任务调度分为两种：FIFO（先进先出）调度和 FAIR（公平调度）调度。

- FIFO 调度：即谁先提交谁先执行，后面的任务需要等待前面的任务执行。这是 Spark 的默认的调度模式。
- FAIR 调度：支持将作业分组到池中，并为每个池设置不同的调度权重，任务可以按照权重来决定执行顺序。

在 Spark 中使用哪种调度器可通过配置 spark.scheduler.mode 参数来设置，可选的参数有 FAIR 和 FIFO，默认是 FIFO。

FIFO 调度算法为 FIFOSchedulingAlgorithm，该算法的 comparator 方法的 Scala 源代码如下：

```
override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
    val priority1 = s1.priority   // priority 实际为 Job ID
    val priority2 = s2.priority
    var res = math.signum(priority1 - priority2)
    if (res == 0) {
      val stageId1 = s1.stageId
      val stageId2 = s2.stageId
      res = math.signum(stageId1 - stageId2)
    }
    res < 0
}
```

根据以上代码，FIFO 调度算法实现的是：对于两个调度任务 s1 和 s2，首先比较两个任务的优先级（Job ID）大小，如果 priority1 比 priority2 小，那么返回 true，表示 s1 的优先级比 s2 的高。由于 Job ID 是顺序生成的，先生成的 Job ID 比较小，所以先提交的 Job 肯定比后提交的 Job 优先级高，也即先提交的 Job 会被先执行。

如果 s1 和 s2 的 priority 相同，表示为同一个 Job 的不同 Stage，则比较 Stage ID，Stage ID 小

则优先级高。

FAIR 调度算法为 FairSchedulingAlgorithm，该算法的 comparator 方法的 Scala 源代码如下：

```
override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
    val minShare1 = s1.minShare
    val minShare2 = s2.minShare
    val runningTasks1 = s1.runningTasks
    val runningTasks2 = s2.runningTasks
    val s1Needy = runningTasks1 < minShare1
    val s2Needy = runningTasks2 < minShare2
    val minShareRatio1 = runningTasks1.toDouble / math.max(minShare1, 1.0)
    val minShareRatio2 = runningTasks2.toDouble / math.max(minShare2, 1.0)
    val taskToWeightRatio1 = runningTasks1.toDouble / s1.weight.toDouble
    val taskToWeightRatio2 = runningTasks2.toDouble / s2.weight.toDouble

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

由以上代码可以看到，FAIR 任务调度主要由两个因子来控制（关于 FAIR 调度的配置，可参考\${SPARK_HOME}/conf/fairscheduler.xml.template 文件）：

weight：相对于其它池，它控制池在集群中的份额。默认情况下，所有池的权值为 1。例如，如果给定一个特定池的权重为 2，它将获得比其它池多两倍的资源。设置高权重(比如 1000)也可以实现池与池之间的优先级。如果设置为-1000，则该调度池一有任务就会马上运行。

minShare：最小 CPU 核心数，默认是 0，它能确保池总是能够快速地获得一定数量的资源(例如 10 个核)，在权重相同的情况下，minShare 越大，可以获得更多的资源。

对以上代码的理解：

- 如果 s1 所在的任务池正在运行的任务数量比 minShare 小，而 s2 所在的任务池正在运行的任务数量比 minShare 大，那么 s1 会优先调度。反之，s2 优先调度。
- 如果 s1 和 s2 所在的任务池正在运行的 task 数量都比各自 minShare 小，那么 minShareRatio 小的优先被调度。
- 如果 s1 和 s2 所在的任务池正在运行的 task 数量都比各自 minShare 大，那么

taskToWeightRatio 小的优先被调度。

- 如果 minShareRatio 或 taskToWeightRatio 相同，那么最后比较各自 Pool 的名字。

3.8.1 Spark 执行模型

上述代码段的执行分两个阶段进行。

1）逻辑计划：在此阶段，使用一组转换创建一个 RDD，它通过构建一个计算链(一系列 RDD)作为转换图来跟踪驱动程序中的这些转换，从而生成一个称为"Lineage Graph"的 RDD。

转换可以进一步分为两种类型：

(a) 窄转换：操作的管道，可以作为一个阶段执行，并且不需要在分区之间移动数据（shuffle）—例如，map、filter 等等。

(b) 宽转换：在这里，每个操作都需要对数据进行 shuffle，因此，对于每个宽转换，都将创建一个新的 stage—例如 reduceByKey，等等。

我们可以使用 toDebugString 来查看 lineage 图：

```
wordCounts.toDebugString
```

2）物理计划：在这个阶段，一旦我们在 RDD 上触发了一个 action 操作，DAG Scheduler 就会查看 RDD lineage，并与 TaskSchedulerImpl 一起提出带有阶段和任务（stages and tasks）的最佳执行计划，并将作业并行地执行到一组任务中。

```
wordCounts.collect
```

一旦我们执行了一个 action 操作，SparkContext 就会触发一个作业并注册该 RDD，直到（作为 DAGScheduler 的一部分的）第一个阶段（stage）。

现在，在进入下一个阶段(宽转换)之前，它将检查是否有任何要 shuffle 的分区数据，以及是否有它所依赖的任何缺失的父操作结果，如果缺少任何这样的阶段，那么它通过使用 DAG(Directed Acyclic Graph，有向无环图)来重新执行这部分操作，这使得它具有容错能力。

在缺少任务（tasks）的情况下，它将任务分配给 executors。每个任务（task）都被分配给 executor 的 CoarseGrainedExecutorBackend。它从 Namenode 获取块信息。现在，它执行计算并返回结果。

接下来，DAGScheduler 查找新运行的阶段并触发下一个阶段(reduceByKey)操作。ShuffleBlockFetcherIterator 获取要 shuffle 的块。现在 reduce 操作被分成两个任务并执行。在完成每个任务时，执行程序将结果返回给驱动程序。一旦作业完成，结果就会显示出来。

3.8.2 理解数据依赖

当 RDD1 经过 transformation 生成了 RDD2，就称作 RDD2 依赖 RDD1，RDD1 是 RDD2 的父 RDD，他们是父子关系。

先看一个例子

```
val r00 = sc.parallelize(0 to 9)
val r01 = sc.parallelize(0 to 90 by 10)
val r10 = r00 cartesian r01
val r11 = r00.map(n => (n, n))
val r12 = r00 zip r01
```

```
val r13 = r01.keyBy(_ / 20)
val r20 = Seq(r11, r12, r13).foldLeft(r10)(_ union _)
```

我们看下 RDD 之间的依赖关系图

RDD 的依赖关系网又叫 RDD 的血统(lineage)，可以看做是 RDD 的逻辑执行计划。

存在两类基本的依赖：窄依赖和宽依赖。窄依赖可进一步被分为 one-to-one 依赖和 range 依赖。range 依赖只用于 union 转换-它们在单个的依赖中合并多个父 RDD。One-to-one 依赖被用在所有其他不要求 shuffle 的情况下。

窄依赖（NarrowDependency）：每个父 RDD 的一个分区最多被子 RDD 的一个分区所使用，即 RDD 之间是一对一的关系。窄依赖的情况下，如果下一个 RDD 执行时，某个分区执行失败（数据丢失），只需要重新执行父 RDD 的对应分区即可进行数恢复。例如 map、filter、union 等算子都会产生窄依赖。

宽依赖(WideDependency，或 ShuffleDependency)：是指一个父 RDD 的分区会被子 RDD 的多个分区所使用，即 RDD 之间是一对多的关系。当遇到宽依赖操作时，数据会产生 Shuffle，所以也称之为 ShuffleDependency。宽依赖情况下，如果下一个 RDD 执行时，某个分区执行失败（数据丢失），则需要将父 RDD 的所有分区全部重新执行才能进行数据恢复。例如 groupByKey、reduceByKey、sortByKey 等操作都会产生宽依赖。

RDD 依赖关系如下图所示：

窄依赖：map, union, filter

宽依赖：groupByKey, distinct, join

RDD 可以被看作是一组分区，带有一个所依赖的父 RDD 列表，以及一个函数来计算一个分配给其父 RDD 的分区。

父 RDD 与子 RDD 之间的依赖关系记录在子 RDD 的属性中(deps: Seq[Dependency[_]])，数据类型为 Dependency(可以有多个)，Dependency 中保存了父 RDD 的引用，这样通过 Dependency 就能找到父 RDD。

Dependency 不仅描述了 RDD 之间的依赖关系，还进一步描述了不同 RDD 的 partition 之间的依赖关系。

依据 partition 之间依赖关系的不同 Dependency 分为两大类：

- NarrowDependency 窄依赖，1 个父分区只对应 1 个子分区，这时父 RDD 不需要改变分区方式。如：map、filter、union，co-paritioned join
- ShuffleDependency Shuffle 依赖(宽依赖)，1 个父分区对应多个子分区，这种情况父 RDD 必须重新分区，才能符合子 RDD 的需求。如：groupByKey、reduceByKey、sortByKey，（not co-paritioned）join

NarrowDependency

NarrowDependency 是一个抽象类，一共有 3 中实现类，也就是说有 3 种 NarrowDependency。

- OneToOneDependency：一对一依赖，比如 map,
- RangeDependency：范围依赖，如 union

- PruneDependency：裁剪依赖，过滤掉部分分区，如 PartitionPruningRDD

ShuffleDependency
出现 shuffle 依赖表示父 RDD 与子 RDD 的分区方式发生了变化。

有时，一个父 RDD 的每个分区都被单个的 RDD 使用，这被称为“窄依赖”。窄的依赖关系是我们期待的，因为当一个父 RDD 分区丢失时，只需要重新计算一个子分区。另一方面，计算一个单独的子 RDD 分区，如果其中涉及到如 gruop by key 这样的操作，这将会依赖于多个父 RDD 分区。因此，在几个子 RDD 分区中创建数据时，需要依次从每个父 RDD 分区获得数据，这种依赖性被称为“宽依赖”。在窄依赖的情况下，可以将双亲和子 RDD 分区保留在单个节点上。但这在宽依赖的情况下是不可能的，因为父数据分散在几个分区上。所以在宽依赖这种情况下，数据会跨分区进行 shuffle。数据 shuffle 是一种资源密集型的操作，应该尽可能避免。另一个宽依赖具有的问题是，如果单个父 RDD 分区丢失，所有的 RDD 分区都需要重新计算。

例如，有以下这样的代码：

```
// 构造一个 RDD，默认分区
val rdd1= sc.parallelize(Seq(3, 3, 9, 2, 8, 5, 6, 7.0), 4)

// 窄依赖 map 到元组(x,1)
val rdd2 = rdd1.map(x => (x, 1))

// 宽依赖 groupByKey
val rdd3 = rdd2.groupByKey()
```

执行结果如下所示：

```
rdd1: org.apache.spark.rdd.RDD[Double] = ParallelCollectionRDD[14] at parallelize at <cc
rdd2: org.apache.spark.rdd.RDD[(Double, Int)] = MapPartitionsRDD[15] at map at <console>
rdd3: org.apache.spark.rdd.RDD[(Double, Iterable[Int])] = ShuffledRDD[16] at groupByKey
```

可使用下图理解其中的分区依赖 - 宽依赖和窄依赖：

输出的 DAG 图的文本表示如下：(先输出最后的 RDD)

```
rdd3.toDebugString
```

输出内容如下图所示：

每一个 job 都根据 shuffle 的发生点被划分为几个 stages(阶段)。每个 stage 的结果保存在磁盘上作为 executor 机器上的中间文件。在下一阶段的 stage 中，每个分区从这些中间文件接收属于它的数据，并且在后续的转换和最终 collect 过程中继续执行。

对于每个 stage 和每个分区，tasks 都被创建并发送给 executors。如果该 stage 以一个 shuffle 结束，那么该创建的 tasks 将是 shuffle-map tasks。在完成特定 stage 的所有任务之后，驱动程序为下一个 stage 创建 tasks，并将其发送给 executors，等等。这将重复到最后一个阶段，它需要将结果返回给驱动程序。为最后阶段创建的任务称为 result tasks。

下面这个图阐述了所有这些概念。

```
// 创建一个集合，包含 500 个随机整数
```

```
val list = List.fill(500)(scala.util.Random.nextInt(10))

// 并行化一个随机的整数列表，并创建一个带有五个分区的 RDD
val listrdd = sc.parallelize(list, 5)

// 对 RDD 进行 map 转换，并创建一个 pair RDD
val pairs = listrdd.map(x => (x, x*x))

// 按 key 对 RDD 的值求和
val reduced = pairs.reduceByKey((v1, v2)=>v1+v2)

// 对 RDD 的分区进行 map 转换，以创建其 key-value 对的字符串表示
val finalrdd = reduced.mapPartitions(iter => iter.map({case(k,v)=>"K="+k+",V="+v}))
finalrdd.collect()

// 输出 RDD 的 DAG 的文本表示
println(finalrdd.toDebugString)
```

这些转换在现实生活中没有意义或用处，但是它们有助于阐明 RDD 依赖性的概念。由此产生的 DAG 如图所示。

输出的 DAG 图的文本表示如下：(先输出最后的 RDD)

每一个 job 都根据 shuffle 的发生点被划分为几个 stages(阶段)。下图显示了本示例中创建的两个 stages：

stage1 包含了导致 shuffle 的转换：parallelize、map 和 reduceByKey。stage1 的结果保存在磁盘上作为 executor 机器上的中间文件。在 stage2 中，每个分区从这些中间文件接收属于它的数据，并且在第二个 map 转换和最终 collect 过程中继续执行。

对于每个 stage 和每个分区，tasks 都被创建并发送给 executors。如果该 stage 以一个 shuffle 结束，那么该创建的 tasks 将是 shuffle-map tasks。在完成特定 stage 的所有任务之后，驱动程序为下一个 stage 创建 tasks，并将其发送给 executors，等等。这将重复到最后一个阶段（在本例中是 stage2），它需要将结果返回给驱动程序。为最后阶段创建的任务称为 result tasks。

3.8.3 通过 Spark WebUI 查看代码执行过程

Spark-UI 有助于理解代码执行流和完成特定任务所需的时间。可视化有助于发现在执行期间发生的任何潜在问题，并进一步优化 spark 应用程序。

作业完成后，就可以看到作业的详细信息，比如阶段的数量、作业执行期间调度的任务的数量。

单击完成的作业，我们可以查看 DAG 可视化，例如，作为它的一部分的不同的宽变换和窄变换。

可以看到每个阶段的执行时间。

在单击作业的某个特定阶段时，它将显示数据块位于何处、数据大小、使用的执行程序、使用的内存和完成特定任务所需的时间等详细信息。它还显示了发生的 shuffle 次数。

此外，我们可以单击 Executors 选项卡查看使用的 Executor 和 Driver。

现在我们已经了解了 Spark 的内部工作方式，可以通过使用 Spark UI、日志和调整 Spark eventlistener 来确定执行流，从而在提交 Spark 作业时确定最佳解决方案。

3.9 使用共享变量

Spark 中的第二个抽象是可以在并行操作中使用的共享变量。默认情况下，当 Spark 作为不同节点上的一组任务并行运行一个函数时，它会将函数中使用的每个变量的副本发送给每个任务。有时候，一个变量需要在任务之间共享，或者在任务和驱动程序之间共享。Spark 支持两种类型的共享变量：广播变量(broadcast variable)和累加器(accumulator)，广播变量可用于在所有节点的内存中缓存一个值，累加器是只“添加”到其中的变量，比如计数器和 sum。

通常，当传递给 Spark 操作(如 map 或 reduce)的函数在远程集群节点上执行时，它会在函数中使用的所有变量的单独副本上工作。这些变量被复制到每台机器，而对远程机器上的变量的更新不会传播回驱动程序。

广播变量和累加器能够维护一个全局状态，或者在 Spark 程序中的任务和分区之间共享数据。

3.9.1 广播变量

广播变量允许程序员在每台机器上保持一个缓存的只读变量，而不是将其副本与任务一起发送。例如，可以使用它们以有效的方式为每个节点提供一个大型输入数据集的副本。Spark 还尝试使用高效的广播算法来分发广播变量，以降低通信成本。

广播变量可以从整个集群中共享和访问，但它们不能被 executors 修改。驱动程序创建一个广播变量，executors 读取它。

如果有大量的数据，而这些数据是大多数 executors 所需要的，那么应该使用广播变量。通常，在驱动程序中创建的变量，由执行任务所需的任务，被序列化并随这些任务一起发送。但是一个驱动程序可以在几个作业中重用相同的变量，并且一些任务可能会被发送到同一个 executor，作为同一作业的一部分。因此，一个潜在的大变量可能会被串行化并在网络上传输超过必要的次数。在这些情况下，最好使用广播变量，因为它们可以以一种更优化的方式传输数据，而且只传输一次。

Spark 操作通过一组阶段（stage）执行，通过分布式“shuffle”操作进行分隔。Spark 自动广播每个阶段（stage）中任务所需的公共数据。以这种方式广播的数据以序列化的形式缓存，并在运行每个任务之前反序列化。这意味着，只有当跨多个阶段（stage）的任务需要相同的数据，或者以反序列化的形式缓存数据非常重要时，显式地创建广播变量才有用。

广播变量是用 SparkContext.broadcast(value)方法创建的，它返回一个 Broadcast 类型的对象。值可以是任何可序列化的对象。然后，executors 可以使用 Broadcast.value 方法读取它。

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
```

```
scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

创建 broadcast 变量之后，应该在集群上运行的任何函数中使用它，而不是使用值 v，这样 v 就不会被多次发送到节点。此外，对象 v 在广播后不应进行修改，以确保所有节点获得广播变量的相同值(例如，如果稍后将变量发送到新节点)。

当不再需要广播变量时，可以销毁它。所有关于它的信息都将被删除（从 executors 和驱动程序），并且该变量将不可用。如果试图在调用 destroy 之后访问它，将抛出一个异常。

另一种方法是调用 unpersist，它只从 executors 的缓存中删除变量值。如果您尝试在 unpersist 之后使用它，它将再次被发送到 executors。

影响广播变量的配置参数有：

- spark.broadcast.compress：spark.io.compression.codec
- spark.broadcast.blockSize：默认是 4096
- spark.python.worker.reuse：默认是 true

请看下面的示例：

```
val broads = sc.broadcast(3)            //创建广播变量，变量可以是任意类型

val lists = List(1,2,3,4,5)             // 创建一个测试的 List
val listRDD = sc.parallelize(lists)     // 构造一个 rdd

val results = listRDD.map(x => x * broads.value)      //map 操作数据

println("结果是：")
results.collect.foreach(println)        // 遍历结果
```

```
结果是：
3
6
9
12
15
```

3.9.2 累加器

累加器(Accumulators)是只能 add 的跨 executors 之间共享的变量。可以使用它们来实现 Spark job 中的全局求和与计数。

可以创建命名或未命名的累加器。如下图所示，一个指定的累加器(在这个实例计数器中)将显示在 web UI 中，用于修改该累加器的阶段（stage）。Spark 在“Tasks”表中显示由任务修改的每个累加器的值。

在 UI 中跟踪累加器对于理解运行阶段（stages）的进度非常有用(注意：Python 中还不支持这一点)。

可以通过调用 SparkContext.longAccumulator()或 SparkContext.doubleAccumulator()来创建数值累加器，以分别累积 Long 或 Double 类型的值。然后，可以使用 add 方法将运行在集群上的任务添加到集群

中。但是，他们无法读取它的值。只有驱动程序可以读取累加器的值，使用它的 value 方法。下面是使用累加器的示例：

```
val acc = sc.longAccumulator("My Accumulator")
acc.value         // 0

val list = sc.parallelize(Array(1, 2, 3, 4))

// 在 executors 上执行
list.foreach(x => acc.add(x))

// 在 driver 上执行
acc.value         // 10
```

在 Spark 的执行模型中，只有当计算被触发(例如，由一个 action)时，Spark 才会添加累加器。

【示例】删除指定数组是既能被 2 整除也能被 3 整除的数，并统计删除了多少个。

```
import org.apache.spark.sql.SparkSession

object AccDemo1 {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder().master("local[*]").appName("accumulator").getOrCreate()

    // 定义计数器
    val acc = spark.sparkContext.longAccumulator("my_counter")

    // 构造 RDD
    val rdd1 = spark.sparkContext.parallelize(1 to 12)

    rdd1.filter(number => {
      if(number%2 ==0 && number%3==0){
        acc.add(1)
        false
      }else{
        true
      }
    }).collect()
      .foreach(println)

    println(s"删除了符号条件的元素有${acc.value}个")
  }
}
```

【示例】数据清洗示例。将下面数据集中任意关键字段为空的条目剔除，并以打印语句输出删除条目数；关键字段定义为{星级}。

数据集样本如下：

```
SEQ,酒店,国家,省份,城市,商圈,星级,业务部门,房间数,图片数,评分,评论数
aba_2066,马尔康嘉绒大酒店,中国,四川,阿坝,,四星级/高档,OTA,85,,4.143799782,108
aba_2069,阿坝马尔康县澜峰大酒店,中国,四川,阿坝,,,低星,115,,3.977930069,129
aba_2094,阿坝鑫鸿大酒店,中国,四川,阿坝,四姑娘山,二星及其他,低星,,,,
aba_2096,九寨沟管理局荷叶迎宾馆,中国,四川,阿坝,九寨沟沟口,二星及其他,低星,49,,3.972340107,394
```

```
aba_2097,九寨沟风景名胜区管理局贵宾楼饭店,中国,四川,阿坝,九寨沟沟口,三星级/舒适,低星,50,,4.12789011,585
aba_2098,九寨沟九鑫山庄,中国,四川,阿坝,九寨沟沟口,四星级/高档,OTA,60,,4.04046011,161
aba_2102,九寨沟冈拉美朵酒店,中国,四川,阿坝,九寨沟沟口,四星级/高档,OTA,198,,3.471659899,12
aba_2109,若尔盖大藏酒店古格王朝店,中国,四川,阿坝,西部旅游牧场,五星级/豪华,OTA,94,,3.263220072,62
aba_2111,若尔盖大藏酒店圣地店,中国,四川,阿坝,西部旅游牧场,四星级/高档,OTA,188,,3.921580076,119
aba_2117,九寨沟保利新九寨宾馆,中国,四川,阿坝,漳扎镇,五星级/豪华,OTA,329,,4.353809834,269
aba_2134,九寨沟名人酒店,中国,四川,阿坝,漳扎镇,三星级/舒适,低星,292,,4.539999962,57
aba_2150,九寨沟仁智度假酒店,中国,四川,阿坝,九寨沟沟口,三星级/舒适,低星,137,,3.782749891,173
aba_2152,九寨沟药泉山庄,中国,四川,阿坝,黄龙机场、川主寺,四星级/高档,OTA,128,,3.821099997,310
aba_2156,松潘黄龙寺华龙山庄,中国,四川,阿坝,黄龙风景区,四星级/高档,OTA,154,,3.315500021,107
aba_2213,阿坝山之旅背包客栈,中国,四川,阿坝,四姑娘山,二星及其他,客栈,20,,,13
aba_2217,阿坝若尔盖大酒店,中国,四川,阿坝,,二星及其他,低星,71,,4.267769814,2
aba_2233,九寨沟云天海大酒店,中国,四川,阿坝,九寨沟沟口,三星级/舒适,低星,100,,2.783930063,
aba_2243,阿坝九旅假日酒店,中国,四川,阿坝,九寨沟沟口,四星级/高档,OTA,140,,3.00515008,49
aba_2248,九寨沟川主寺岷江源大酒店,中国,四川,阿坝,黄龙机场、川主寺,,低星,228,,3.211859941,109
```

实现代码如下：

```
package com.mm.rdd

import org.apache.spark.sql.SparkSession

object AccDemo2 {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder().master("local[*]").appName("accumulator").getOrCreate()

    // 定义计数器
    val starCounter = spark.sparkContext.longAccumulator("star_counter")

    // 加载外部数据源，构造 RDD
    val rdd1 = spark.sparkContext.textFile("input/hotel/sample2.csv")

    // 过滤掉标题行
    val rdd2 = rdd1.filter(line => !line.startsWith("SEQ"))

    // 将关键字段有缺失值的记录删除：即将字段{星级、评论数、评分}中任意字段为空的数据删除
    // 并打印输出删除条目数 - 使用计数器统计
    val rdd3 = rdd2.map(_.split(",", -1))
      .filter(arr => {
        // 如果"星级"字段为空
        if(arr(6)==null || arr(6).trim.isEmpty){
          starCounter.add(1)      // 全局计数器 + 1
          false
        }else{
          true
        }
      })

    // 显示
    println(s"过滤前记录数${rdd2.count}，过滤后记录后${rdd3.count}")
    println(s"删除的星级字段缺失的记录数是：${starCounter.value}")
  }
```

```
}
```

执行以上代码，输出结果如下：

```
过滤前记录数 19，过滤后记录后 17
删除的星级字段缺失的记录数是：2
```

如果我们把关键字段定义为{星级、评分、评论数}，并分别统计每个字段的缺失值，则实现如下：

```
package com.mm.rdd

import org.apache.spark.sql.SparkSession

object AccDemo3 {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder().master("local[*]").appName("accumulator").getOrCreate()

    // 定义计数器
    val starCounter = spark.sparkContext.longAccumulator("star_counter")
    val scoreCounter = spark.sparkContext.longAccumulator("score_counter")
    val commCounter = spark.sparkContext.longAccumulator("comm_counter")

    // 加载外部数据源，构造 RDD
    val rdd1 = spark.sparkContext.textFile("input/hotel/sample2.csv")

    // 过滤掉标题行
    val rdd2 = rdd1.filter(line => !line.startsWith("SEQ"))

    // 将关键字段有缺失值的记录删除：即将字段{星级、评论数、评分}中任意字段为空的数据删除
    // 并打印输出删除条目数 - 使用计数器统计
    val rdd3 = rdd2.map(_.split(",",-1))
      .filter(arr => {
        var flag = true          // 定义标志变量
        // 如果"星级"字段为空
        if(arr(6)==null || arr(6).trim.isEmpty){
          starCounter.add(1)     // 全局计数器 + 1
          flag = false
        }

        if(arr(10)==null || arr(10).trim.isEmpty){
          scoreCounter.add(1)    // 全局计数器 + 1
          flag = false
        }

        if(arr(11)==null || arr(11).trim.isEmpty){
          commCounter.add(1)     // 全局计数器 + 1
          flag = false
        }

        flag
      })

    // 显示
    println(s"过滤前记录数${rdd2.count}，过滤后记录后${rdd3.count}")
```

```
    println(s"删除的星级字段缺失的记录数是：${starCounter.value}")
    println(s"删除的评分字段缺失的记录数是：${scoreCounter.value}")
    println(s"删除的星级字段缺失的记录数是：${commCounter.value}")

    // 存储清洗结果
    rdd3.map(arr => arr.mkString(",")).coalesce(1).saveAsTextFile("output/hotel")
  }
}
```

执行以上代码，输出结果如下：

```
过滤前记录数 19，过滤后记录后 14
删除的星级字段缺失的记录数是：2
删除的评分字段缺失的记录数是：2
删除的星级字段缺失的记录数是：2
```

3.10 Spark RDD 编程案例

下面我们演示几个使用 Spark RDD 的常用案例。

3.10.1 Top N 问题

【示例】给出一个员工信息名单，找出收入最高的前 10 名员工（Top N 问题）。

样本数据 employees.csv 内容如下：

```
ename,title,department,Full or Part-Time,Salary or Hourly,Typical Hours,Annual Salary,Hourly Rate
张三,paramedic i/c,fire,f,salary,,91080.00,
李四,lieutenant,fire,f,salary,,114846.00,
王老五,sergeant,police,f,salary,,104628.00,
赵六,police officer,police,f,salary,,96060.00,
钱七,clerk iii,police,f,salary,,53076.00,
周扒皮,firefighter,fire,f,salary,,87006.00,
吴用,law clerk,law,f,hourly,35,,14.51
```

实现代码如下。

```
// RDD 实现
val inputPath = "file:///home/hduser/data/spark_demo/employees.csv"
val rdd = sc.textFile(inputPath);

// def sortBy[K](f: (T) ⇒ K, ascending: Boolean = true, numPartitions: Int = this.partitions.length)
val sortedData = rdd.map(_.split(","))
                    .sortBy(t => if(t(6).length>0) t(6).toFloat else 0.0, false)

val top = sortedData.take(10)

top.foreach(emp => println(emp.toList.mkString(",")))
```

执行以上代码，得到如下的结果：

```
李四,lieutenant,fire,f,salary,,114846.00
王老五,sergeant,police,f,salary,,104628.00
赵六,police officer,police,f,salary,,96060.00
张三,paramedic i/c,fire,f,salary,,91080.00
周扒皮,firefighter,fire,f,salary,,87006.00
```

```
钱七,clerk iii,police,f,salary,,53076.00
吴用,law clerk,law,f,hourly,35,,14.51
```

3.10.2 电影数据集分析

下面的示例使用 Spark RDD 实现对电影数据集进行分析。在这里我们使用推荐领域一个著名的开放测试数据集 movielens。我们将使用其中的电影评分数据集 ratings.csv 以及电影数据集 movies.csv。

【例】请找出平均评分超过 4.0 的电影，列表显示。

实现过程和代码如下：

1、加载数据，构造 RDD：

```
// 加载数据，构造 RDD
val ratings = "/data/spark_demo/movielens/ratings.csv"          // 评分数据集
val movies = "/data/spark_demo/movielens/movies.csv"            // 电影数据集

val ratingsRDD = sc.textFile(ratings)
val moviesRDD = sc.textFile(movies)

ratingsRDD.count                // 评分数据集中数据总记录数量
ratingsRDD.cache                // 缓存评分数据集

moviesRDD.count                 // 电影数据集中数据总记录数量
moviesRDD.cache                 // 缓存电影数据集
```

输出结果如下：

可以看到，评分数据集 ratingsRDD 中总共有 100837 条评论记录，电影数据集 moviesRDD 中总共有 9743 部电影信息。

2、对评分数据集和电影数据集进行简单探索，了解数据：

```
// 对评分数据集进行简单探索，了解数据：
ratingsRDD.take(5).foreach(println)
println                         // 换行
// 对电影数据集进行简单探索，了解数据：
moviesRDD.take(5).foreach(println)
```

输出结果如下所示：

3、处理评分数据集，包括：忽略标题行；抽取(movieId,rating)字段。代码如下所示：

```
val rating = ratingsRDD.filter(line => !line.startsWith("userId")).                 // 去掉标题行
                        map(line => {
                            val fileds = line.split(",")                            // 拆分一行记录
                            (fileds(1).trim.toInt, fileds(2).trim.toDouble)
                        })

rating.take(5).foreach(println)
```

输出结果如下：

可以看到，已经去掉了标题行，并在 rating RDD 中只保留了 movieId（电影 ID）和 rating（评分）

这两个字段。

4、对 rating RDD 进行转换，按 key（即 movieId）进行分组，并计算每一组的平均值（也就是每部电影的平均评分）。代码如下所示：

```
// 获得(movieid,ave_rating)
val movieScores = rating.groupByKey().
                        map(t => {
                            val avg = t._2.sum / t._2.size        // 计算平均得分
                            (t._1, avg)                           // 构造元组，元素为电影 Id 和平均评分
                        })

// 查看前 5 条数据
movieScores.take(5).foreach(println)
```

输出结果如下：

可以看出，经过这一步处理后，返回的是一个元组，元组元素为(movieId,平均得分)。但是这个结果对用户来说并不友好，因为它只显示了电影的 ID。下一步显示对应的电影名称及其平均得分。

5、处理电影数据集，包括：忽略标题行；抽取(movieId,movieName)字段。代码如下所示：

```
// 抽取出(MovieID,MovieName)
val movieskey = moviesRDD.filter(line => !line.startsWith("movieId")).        // 去年标题行
                          map(line=>{
                              val fileds = line.split(",")                    // 拆分一行记录
                              (fileds(0).toInt,fileds(1))
                          })
movieskey.take(5).foreach(println)
```

输出结果如下所示：

6、最后一步，将 movieScores RDD 和 movieskey RDD 进行 join 连接，从而得到每部电影的名称及其得分。代码如下所示：

```
// 通过 join 连接,可以得到<movieId,movieName,averageRating>
val result = movieScores.join(movieskey).
                        filter(f => f._2._1>4.0).
                        map(f => (f._1,f._2._2,f._2._1))

result.take(5).foreach(println)
```

输出结果如下：

3.10.3 合并小文件

使用 SparkContext 的 wholeTextFiles 方法和 colleasc 方法，可以实现对小文件的合并。

```
package com.xueai8.rdd

import org.apache.spark.sql.SparkSession

/**
  *
```

```
 * 加载整个目录中的文件，使用 wholeTextFiles
 */
object WordCount3 {
  def main(args: Array[String]): Unit = {

    // 创建 SparkSession 实例 - 入口
    val spark = SparkSession.builder.master("local[*]").appName("HelloWorld").getOrCreate

    // 加载数据源，构造 RDD
    val textFiles = spark.sparkContext.wholeTextFiles("input/files")

    textFiles.map(_._2).coalesce(1).saveAsTextFile("output/one")
  }
}
```

3.10.4 使用 Spark RDD 实现二次排序

什么是二次排序？二次排序就是对于<key,value>类型的数据，不但按 key 排序，而且每个 Key 对应的 value 也是有序的。

假设我们有以下输入文件 data.txt，其中逗号分割的分别是年、月和总数：

```
2018,5,22
2019,1,24
2018,2,128
2019,3,56
2019,1,3
2019,2,-43
2019,4,5
2019,3,46
2018,2,64
2019,1,4
2019,1,21
2019,2,35
2019,2,0
```

我们想要对这些数据排序，期望的输出结果如下：

```
2018-2   64,128
2018-5   22
2019-1   3,4,21,24
2019-2   -43,0,35
2019-3   46,56
2019-4   5
```

Spark 二次排序解决方案如下：需要将年和月组合起来构成一个 Key，将第三列作为 value，并使用 groupByKey 函数将同一个 Key 的所有 Value 全部分组到一起，然后对同一个 Key 的所有 Value 进行排序即可。

```
// 加载数据集
val inputPath = "file:///home/hduser/data/spark/data.txt"
val inputRDD = sc.textFile(inputPath)

// 实现二次排序
```

```
val sortedRDD = inputRDD
                        .map(line => {
                           val arr = line.split(",")
                           val key = arr(0) + "-" + arr(1)
                           val value = arr(2)
                           (key,value)
                        })
                        .groupByKey()
                        .map(t => (t._1, t._2.toList.sortWith(_.toInt < _.toInt).mkString(",")))
                        .sortByKey(true)                // true:升序，false:降序

// 结果输出
sortedRDD.collect.foreach(t => println(t._1 + "\t" + t._2))
```

第 4 章　Spark SQL

Spark SQL 是 Spark 用于处理结构化和半结构化数据的接口，允许使用关系操作符表示分布式内存计算。结构化数据被认为是任何有模式的数据，如 JSON、Hive 表、Parquet。模式意味着为每个记录拥有一组已知的字段。半结构化数据是指模式和数据之间没有分离。

与基本的 Spark RDD API 不同，Spark SQL 提供的接口为 Spark 提供了有关数据结构和正在执行的计算的更多信息。在内部，Spark SQL 使用这些额外的信息来执行额外的优化。

Spark 无疑是 Apache 软件基金会能够构想出的最成功的项目之一。他们引入了 Spark SQL，将关系处理与 Spark 的函数式编程 API 集成在一起。因此，Spark SQL 被设计用来集成关系型处理和函数式编程的功能，这样复杂的逻辑就可以在分布式计算设置中实现、优化和扩展。

4.1 Spark SQL 数据抽象

尽管 Spark 提供了一个函数式编程 API 来操作分布式的数据集合，但使用 RDD 进行编码有些复杂和混乱，而且有时很慢。Spark SQL 提供了使用结构化和半结构化数据的三个主要功能：

- 它提供了由 Python、Java 和 Scala 语言所支持的 DataFrame/Dataset 抽象，以简化使用结构化数据集的工作。DataFrame/Datase 类似于关系数据库中的表。
- 它可以读写各种结构化格式的数据(如 JSON、Hive 表、Parquet)。
- 它允许在 Spark 程序内部和通过标准数据库连接器(JDBC/ ODBC)连接到 Spark SQL 的外部工具(如 Tableau 等商业智能工具)中使用 SQL 查询数据。

Spark SQL 提供了一个统一的接口，用于在分布式存储系统(如 Cassandra 或 HDFS (Hive, Parquet, JSON))中访问数据，使用专门的 DataFrameReader 和 DataFrameWriter 对象。

Spark SQL 允许对 Hadoop HDFS 或与 Hadoop 兼容的文件系统(如 S3)中的大量数据执行类似 SQL 的查询。它可以访问来自不同数据源(文件或表)的数据。

Spark SQL 的主要数据抽象是 Dataset，它表示结构化数据(具有已知模式的记录)。这种结构化数据表示 Dataset 支持使用存储在 JVM 堆外的托管对象中的压缩柱状格式的紧凑二进制表示。它可以通过减少内存使用和 GC 来加快计算速度。

Dataset 是一个带有 transformation 和 action 的结构化查询执行管道的编程接口(就像在旧的 Spark Core 中的 RDD API 一样)。在内部，结构化查询是(逻辑和物理)关系运算符和表达式的 Catalyst 树。

Spark DataFrame 是一个分布式的记录集合，被组织成命名的列。DataFrames 可以从各种各样的数据源构建，例如：结构化数据文件、Hive 中的表、外部数据库或现有的 RDDs。

DataFrame API 支持 Scala、Java、Python 和 R 等多种语言。在 Scala 和 Java 中，DataFrame 由 Row 构成的 Dataset 表示。在 Scala API 中，DataFrame 只是 Dataset[Row]的一个类型别名。而在 Java API 中，用户需要使用 Dataset<Row>来表示 DataFrame。

DataFrame API 为 Spark 带来了以下两个特性：

- 内置对各种数据格式的支持，如 Parquet、Hive 和 JSON。尽管如此，通过 Spark SQL 的外部数据源 API，DataFrames 可以访问各种各样的第三方数据源，如数据库和 NoSQL 存储。
- 提供了更健壮、功能丰富的 DSL（特定领域语言），它的功能是为常见任务设计的，例如：
  - 元数据
  - 抽样
  - 关系数据处理—投影、过滤、聚合、连接
  - UDF（用户自定义函数）

Spark 2.0 版本对 API 进行了统一，并扩展了 SQL 功能，包括对子查询的支持。在 Spark 2.0 中，DataFrame API 已与 Dataset API 合并，从而统一了跨 Spark 库的数据处理功能。DataFrame、Dataset 和 SQL 查询共享相同的执行和优化管道；因此，使用这些结构中的任何一个(或使用任何受支持的编程 API)都不会影响性能。

Spark SQL 的核心是 Catalyst 优化器，它利用 Scala 的高级特性(例如模式匹配)来提供可扩展的查询优化器。开发人员编写的基于 DataFrame 的高级代码被转换为 Catalyst 表达式，然后再通过这个执行和优化管道转换为低级 Java 字节码。

DataFrame 在 RDD 上提供了巨大的性能改进，因为它有 2 个强大的功能：

1、自定义内存管理（又名 Project Tungsten）

数据以二进制格式存储在堆外内存中（off-heap memory）。这节省了大量的内存空间。此外，也不涉及垃圾收集开销。通过提前了解数据的模式并有效地以二进制格式存储，还可以避免昂贵的 Java 序列化。

2、优化的执行计划(又名 Catalyst Optimizer)

使用 Spark catalyst optimiser 创建用于执行的查询计划。在经过一些步骤准备好优化的执行计划之后，最终的执行仅在 rdd 内部进行，但这对用户是完全隐藏的。

4.2 Spark SQL 架构组成

Spark SQL 主要由三层组成:

- 语言 API：Spark 支持 Python、HiveQL、Scala、Java 等语言。
- SchemaRDD：由于 Spark SQL 工作在模式、表和记录上，所以可以使用 SchemaRDD 或 DataFrame 作为临时表。
- 数据源：对于 Spark core，数据源通常是一个文本文件、Avro 文件等。而 Spark SQL 的数据源通常是 JSON 文档、Parquet 文件、HIVE 表和 Cassandra 数据库等。

Spark SQL 模块为 Spark 机器学习应用程序、流应用程序、图应用程序和许多其他类型的应用程序体系结构提供了基础，其组件如下图所示。

Spark SQL 的组件包括：

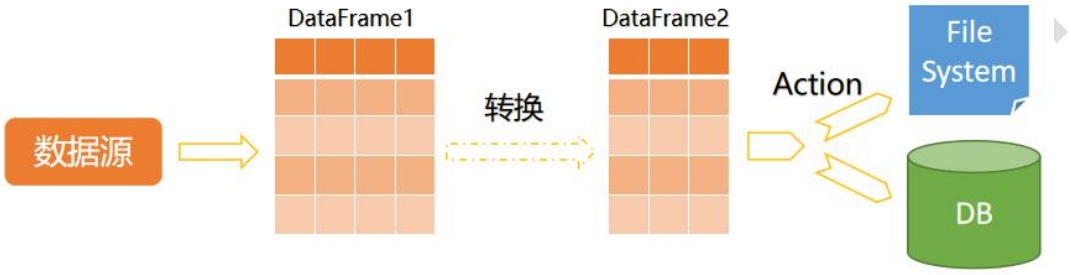
- Spark SQL DataFrame：RDD 有一些缺点。首先，没有处理结构化数据的相关准备，也没有用于处理结构化数据的优化引擎。其次，基于属性，开发人员必须优化每个 RDD。Spark DataFrame 是一个分布式的数据集合，按顺序排列成指定的列。Spark DataFrame 与关系型数据库中的表非常相似。
- Spark SQL Dataset：在 Spark 1.6 版本中，引入了接口是 Dataset。这个接口综合了 RDD 的优点

以及 Spark SQL 的优化执行引擎的优点。为了实现 JVM 对象和表格表示之间的转换，使用了编码器的概念。使用 JVM 对象，可以接收数据集，并且必须使用 map、filter 等函数转换来修改它们。Dataset API 在 Scala 和 Java 中都可用，但在 Python 中不支持。

- Spark Catalyst Optimizer：Catalyst optimizer 是 Spark SQL 中使用的优化器，所有由 Spark SQL 和 DataFrame DSL 编写的查询都是通过这个工具进行优化的。这个优化器比 RDD 更好，因此系统的性能得到了提高。

4.3 Spark SQL 编程模型

所有的 Spark SQL 应用程序都以特定的步骤来工作。这些工作步骤如下图所示：



也就是说，每个 Spark SQL 应用程序都由相同的基本部分组成：

- 从数据源加载数据，构造 DataFrame/Dataset；
- 对 DataFrame/Dataset 执行转换（transformation）操作；
- 将最终的 DataFrame/Dataset 存储到指定位置。

下面我们实现一个完整的的 Spark SQL 应用程序。

【示例】加载 json 文件中的人员信息，并进行统计。请按以下步骤执行。

1）准备数据源文件。

Spark 安装目录中自带了一个 people.json 文件，位于“examples/src/main/resources/”目录下。其内容如下：

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

我们将这个 people.json 文件，拷贝到项目的 resources 目录下。

2）创建一个 Spark 项目，并创建一个 Scala 源文件，编辑代码如下：

```
package com.xueai8.sql

import org.apache.spark.sql.SparkSession

/**
  * Created by www.xueai8.com
  *
  * Spark SQL 编程模型
  * 实现：加载 json 文件中的人员信息，并进行计算。
  */
object Demo01 {
  def main(args: Array[String]): Unit = {
```

```
    // 1）创建 SparkSession
    val spark = SparkSession
      .builder()
      .master("local[*]")
      .appName("Spark SQL basic example")
      .getOrCreate()

    // 用于隐式转换，如将 rdd 转换为 DataFrame
    import spark.implicits._

    // 2）加载数据源，构造 DataFrame
    val input = "./src/main/resources/people.json"
    val df = spark.read.json(input)

    // 3）执行转换操作
    // -- 找出年龄超过 21 岁的人
    val resultDF = df.where($"age" > 21)
    resultDF.show()    // 显示 DataFrame 数据

    // 4）将结果保存到 csv 文件中
    val output = "tmp/people-output"
    resultDF.write.format("csv").save(output)
  }
}
```

3）执行以上代码，在控制台中可以看到输出结果如下：

4）查看存储结果的 csv 文件。如下图中所示：

在本例中，我们使用了本地文件系统。大家可自行修改代码，使用 HDFS 文件系统。

读数据格式

所有读取数据的 API 都遵循以下调用格式：

```
DataFrameReader.format(...).option("key", "value").schema(...).load()
```

例如：

```
spark.read.format("csv")
    .option("mode", "FAILFAST")          // 读取模式
    .option("inferSchema", "true")       // 是否自动推断 schema
    .option("path", "path/to/file(s)")   // 文件路径
    .schema(someSchema)                  // 使用预定义的 schema
    .load()
```

读取模式有以下三种可选项：

写数据格式

所有写数据的 API 都遵循以下调用格式：

```
DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save()
```

例如：

```
dataframe.write.format("csv")
     .option("mode", "OVERWRITE")            // 写模式
     .option("dateFormat", "yyyy-MM-dd")     // 日期格式
     .option("path", "path/to/file(s)")
     .save()
```

写数据模式有以下四种可选项：

4.4 程序入口 SparkSession

在上一节的程序中，我们在代码的开始，首先创建了一个 SparkSession 的实例 spark。

在 Spark 2.0 中，SparkSession 表示在 Spark 中操作数据的统一入口点。要创建一个基本的 SparkSession，需要使用 SparkSession.builder()：

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .getOrCreate()

// 用于隐式转换，如将 RDDs 转换为 DataFrames
import spark.implicits._
```

在 Spark 程序中，我们使用构建器设计模式实例化 SparkSession 对象。然而，在 REPL 环境中(即在 Spark shell 会话中)，SparkSession 会被自动创建，并通过名为 spark 的实例对象提供给我们使用。

SparkSession 对象可以用来配置 Spark 的运行时配置属性。例如，Spark 和 Yarn 管理的两个主要资源是 CPU 和内存。如果想为 Spark executor 设置内核数量和堆大小，那么可以通过分别设置 spark.executor.cores 和 spark.executor.memory 属性来实现这一点。例如，在下面的代码片段中，我们分别设置 Spark executor 运行时属性的核为 2 个，内存为 4G。

```
spark.conf.set("spark.executor.cores", "2")
spark.conf.set("spark.executor.memory", "4g")
```

下面是在 Spark SQL 代码中创建 SparkSession 对象的常用代码模板：

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

val conf = SparkConf()
// conf.set("spark.app.name", application_name)
conf.set("spark.master", master)              // master='yarn-client'
conf.set("spark.executor.cores", `num_cores`)
conf.set("spark.executor.instances", `num_executors`)
conf.set("spark.locality.wait", "0")
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");

val spark = SparkSession
```

```
    .builder()
    .appName(application_name)
    .config(conf=conf).
    getOrCreate()
```

可以使用 SparkSession 对象从各种源读取数据，例如 CSV、JSON、JDBC、Stream 等等。此外，它还可以用来执行 sql 语句、注册用户定义函数(UDF)和使用 Dataset 和 DataFrame。

注：Spark 2.0 中的 SparkSession 为 Hive 特性提供了内置支持，包括使用 HiveQL 编写查询、访问 Hive UDF 以及从 Hive 表中读取数据的能力。要使用这些特性，不需要已有的 Hive 安装。

4.5 Spark SQL 支持的数据类型

Spark DataFrame/Dataset 就像分布在内存中的表，具有指定的列和模式，其中每个列都有特定的数据类型：整数、字符串、数组、map、real、日期、时间戳等。在我们看来，Spark DataFrame 就像一张具有行和列的二维表。

在使用 Spark 定义模式之前，我们首先来了解一下有哪些可用的泛型和结构化数据类型。然后，我们将学习如何使用模式创建 DataFrame。

4.5.1 Spark SQL 基本数据类型

与支持的编程语言相匹配，Spark 支持基本的内部数据类型。这些数据类型可以在 Spark 应用程序中声明，也可以在模式中定义。例如，在 Scala 中，我们可以定义或声明一个特定的列，类型为 String、Byte、Long 或 Map 等。

下表列出了 Spark 支持的基本 Scala 数据类型。除了 DecimalType，它们都是类 DataTypes 的子类型。

下表列出了 Spark 支持的基本 Python 数据类型。

4.5.2 Spark SQL 复杂数据类型

对于复杂的数据分析，我们不在可能只处理简单或基本的数据类型。有时遇到的数据是复杂的，通常是结构化的或嵌套的，需要 Spark 来处理这些复杂的数据类型。下表列出了 Spark 支持的 Scala 结构化数据类型。

下表列出了 Spark 支持的 Python 结构化数据类型。

4.5.3 模式-Schema

Spark 中的模式（Schema）为一个 DataFrame/Dataset 定义了列名和关联数据类型。最常见的情况是，当我们从外部数据源读取结构化数据时，需要用到模式。预先定义模式，而不是采用“读时模式”方法，

有三个好处：

- 减轻了 Spark 推断数据类型的责任。
- 可以防止 Spark 仅为了读取文件的大部分以确定模式而创建单独的作业，这对于大型数据文件来说是昂贵和耗时的。
- 如果数据与模式不匹配，可以尽早发现错误。

因此，当我们想从数据源读取大文件时，最好是预先定义模式。

Spark 支持以两种方式定义模式：

- 第一种是通过编程方式定义 schema；
- 第二种是使用数据定义语言(Data Definition Language, DDL)字符串。

其中第二种模式定义方式要简单得多，也更容易阅读。

要以编程方式为 DataFrame 定义一个模式，假设这个 DataFrame 有三个命名列，author、title 和 pages，可以使用 Spark DataFrame API。代码如下所示：

Scala：

```
import org.apache.spark.sql.types._

val schema = StructType(Array(
        StructField("author", StringType, false),
        StructField("title", StringType, false),
        StructField("pages", IntegerType, false)
    ))
```

4.6.2 列对象和行对象

在 Spark SQL 中，列是具有 public 方法的对象，由 Column 类型表示。我们可以按名称列出所有列，并且可以使用关系表达式或计算表达式对列的值执行操作。还可以在列上使用逻辑或数学表达式。例如：

```
import org.apache.spark.sql.functions._

blogsDF.select(col("Hits") * 2).show(2)
```

DataFrame 中的 Column 对象不能单独存在；每一列都是记录中一行的一部分，所有的行一起构成一个 DataFrame。正如我们将在本章后面看到的那样，DataFrame 实际上是 Scala 中的 Dataset[Row]。

Spark 中的行是一个通用的 Row 对象，包含一个或多个列。每个列可以是相同的数据类型(例如，整数或字符串)，也可以有不同的类型(整数、字符串、映射、数组等)。因为 Row 是 Spark 中的一个对象，是字段的一个有序集合，因此我们可以在 Spark 中实例化 Row，并通过从 0 开始的索引访问它的字段。如下面的代码所示：

```
import org.apache.spark.sql.Row

// Create a Row
val blogRow = Row(6, "Reynold", "Xin", "https://tinyurl.6", 255568, "3/2/2015", Array("twitter", "LinkedIn"))

// Access using index for individual items
blogRow(1)                  // res62: Any = Reynold
```

4.6 构造 DataFrame

Spark DataFrame 的创建类似于 RDD 的创建。为了访问 DataFrame API，需要 SparkSession 作为入口点。在本节中，我们将演示如何从各种数据源创建 DataFrames：

有多种方式可用来创建 DataFrames：

- 简单创建单列和多列 DataFrame；
- 转换已经存在的 RDD；
- 运行 SQL 查询；
- 加载外部数据。

4.6.1 简单创建单列和多列 DataFrame

SparkSession 有一个函数叫 range，可以很容易地创建单列 DataFrame，带有列名 id 和类型 LongType。请执行以下的代码：

```
// 创建单列 DataFrame，默认列名是 id，类型是 LongType
val df0 = spark.range(5).toDF()
df0.printSchema
df0.show
```

输出结果如下所示：

还可以指定列名：

```
val df1 = spark.range(5).toDF("num")
df1.show
```

输出结果如下所示：

另外，还可以指定范围的起始(含)和结束值(不含)：

```
val df2 = spark.range(5,10).toDF("num")
df2.show
```

输出结果如下所示：

另外，还可以指定步长：

```
val df3 = spark.range(5,15,2).toDF("num")
df3.show
```

输出结果如下所示：

请注意，toDF 采用的是元组列表，而不是标量元素。

通过将一个元组集合转换为一个 DataFrame，可创建多列 DataFrame。这需要使用 SparkSession 对象的 toDF 方法。toDF 方法将列标签列表作为可选的参数，以指定转换后的 DataFrame 的标题行。请执行下面的代码：

```
// 用于隐式转换，如将 rdd 转换为 DataFrame
```

```
import spark.implicits._

// 元组序列
val movies = Seq(("马特·达蒙", "谍影重重:极限伯恩", 2007L),
                 ("马特·达蒙", "心灵捕手", 1997L))

// 将元组转为 DataFrame
val moviesDF = movies.toDF("演员", "电影", "年份")

// 输出模式
moviesDF.printSchema

// 显示
moviesDF.show
```

输出结果如下所示：

通过元组来创建单列或多列 DataFrames，每个元组类似于一行。可以选择标题列；否则，Spark 会创建一些模糊的名称，比如_1、_2。列的类型推断是隐式的。

4.6.2 从 RDD 创建 DataFrame

从 RDD 创建 DataFrame 有三种方式：

- 使用包含 Row 数据(以元组的形式)的 RDD
- 使用 case 类
- 明确指定一个模式(schema)

从 RDD 创建 DataFrame 有多种方式，但是这些方法都必须提供一个 schema。要么显式地提供，要么隐式地提供。

下面的示例中，调用 RDD 的 toDF 显式函数，将 RDD 转换到 DataFrame，使用指定的列名。列的类型是从 RDD 中的数据推断出来的。

```
// 用于隐式转换，如将 rdd 转换为 DataFrame
import spark.implicits._

val persons = List(("张三",23),("李四",18),("王老五",35))
val personRDD = spark.sparkContext.parallelize(persons)        // RDD[(String, Int)]
val personsDF = personRDD.toDF("name", "age")                  // rdd to DataFrame
personsDF.printSchema()
personsDF.show()
```

我们在这里创建了一个 RDD 然后把它转换成元组，然后被发送到 toDF 方法。请注意，toDF 采用的是元组列表，而不是标量元素。每个元组类似于一行。我们可以选择列名，否则，Spark 会自行创建一些模糊的名称，比如_1、_2。列的类型推断是隐式的。

执行上面的代码，输出结果如下：

```
```

如果已经有了 RDD 的数据，Spark SQL 就支持两种不同的方法，将现有的 RDD 转换成 DataFrame：

- 第一种方法使用反射来推断 RDD 的模式，该模式包含特定类型（case 类）的对象。

- 第二种方法是通过一个编程接口，先构造出一个模式，然后将其应用到现有的 RDD 中。虽然这种方法比较冗长，但是在事先不知道列类型的情况下，这种方法允许我们自行构造 DataFrame。

首先看第一种方法。使用反射模式，可以实现从 RDD 到 DataFrame 间的隐式转换。请看下面的示例：

```
// 定义一个 case class
case class Person(name:String,age:Long)
......

import spark.implicits._

val peoples = List(
  Person("张三",29),
  Person("李四",30),
  Person("王老五",19)
)
val peopleDF = spark.sparkContext.parallelize(peoples).toDF()
peopleDF.printSchema()
peopleDF.show()
```

输出结果如下所示：

```

```

如果上面示例中的集合元素不是 Person 对象，而是字符串的话，那么需要我们自己通过转换来构造出 Person 集合来。如下所示：

```
// 定义一个 case class
case class Person(name:String,age:Long)
......

// 创建一个 List 集合
val peoples = List("张三,29","李四,30","王老五,19")

// 构造一个 RDD，经过转换为 RDD[Person]类型后，再转换为 DataFrame
val peopleRDD = spark.sparkContext.parallelize(peoples)
val peopleDF = peopleRDD
  .map(_.split(","))
  .map(x => Person(x(0),x(1).trim.toLong))
  .toDF()
peopleDF.printSchema()
peopleDF.show()
```

接下来我们看第二种方法。我们也可以先定义好特定的模式（schema），然后使用该模式（schema）创建一个 DataFrame。这需要使用 SparkSession 的方法 createDataFrame 来创建。创建过程如下面的代码所示：

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
......

import spark.implicits._

// 构造一个 RDD
```

```
val peopleRDD = spark.sparkContext.parallelize(
  Seq(
    Row("张三",30),
    Row("李四",25),
    Row("王老五",35)
  )
)

// 指定一个 Schema(模式)
val fields = Seq(
  StructField("name", StringType, nullable = true),
  StructField("age", IntegerType, nullable = true)
)
val schema = StructType(fields)

// 从给定的 RDD 应用给定的 Schema 创建一个 DataFrame
val peopleDF = spark.createDataFrame(peopleRDD, schema)

// 查看 DataFrame Schema
peopleDF.printSchema

// 输出
peopleDF.show
```

输出结果如下所示：

4.6.3 读取文本文件创建 DataFrame

Spark 提供了一个接口，DataFrameReader，用来从众多的数据源读取数据到 DataFrame，以各种格式，如 JSON、CSV、Parquet、Text、Avro、ORC 等。

同样，要将 DataFrame 以特定格式写回数据源，Spark 使用 DataFrameWriter。

加载存储系统中的文件到 DataFrame，可通过 SparkSession 的 read 字段，它是 DataFrameReader 的一个实例。它有个 load()方法，可直接从配置的数据源加载数据。另外它还有五个快捷方法：text、csv、json、orc 和 parquet，相当于先调用 format()方法再调用 load()方法。

文本文件是最常见的数据存储文件。Spark DataFrame API 允许开发者将文本文件的内容转换成 DataFrame。

让我们仔细看看下面的例子，以便更好地理解（这里我们使用 Spark 自带的数据文件）：

```
val file = "/data/spark_demo/resources/people.txt"
val txtDF = spark.read.format("text").load(file)         // 加载文本文件
// val txtDF = spark.read.text(file)     // 等价上一句，快捷方法

txtDF.printSchema                        // 打印 schema
txtDF.show                               // 输出
```

输出内容如下所示：

Spark 会自动推断出模式，并相应地创建一个单列的 DataFrame。因此，没有必要为文本数据定义模式。不过，当加载大数据文件时，定义一个 schema 要比让 Spark 来进行推断效率更高。

4.6.4 读取 CSV 文件创建 DataFrame

在 Spark 2.x 中，加载 CSV 文件是非常简单的。请看下面的示例。

```
// 数据源文件
val file = "./src/main/resources/people.csv"

val peopleDF = spark.read.format("csv")
   .option("sep", ";")                    // 字段使用;分隔符
   .option("inferSchema", "true")         // 指定自动推断模式
   .option("samplingRatio", 0.001)        // 根据抽样进行模式推断
   .option("header", "true")              // 说明有标题行
   .load(file)

// 使用快捷方法
/*
val peopleDF = spark.read
   .option("sep", ";")                    // 字段使用;分隔符
   .option("inferSchema", "true")         // 指定模式自动推断
   .option("samplingRatio", 0.001)        // 根据抽样进行模式推断
   .option("header", "true")              // 说明有标题行
   .csv(file)
*/

peopleDF.printSchema                      // 打印 schema
peopleDF.show()                           // 显示
```

输出结果如下所示：

```

```

在上面的示例中，使用了模式推断。对于大型的数据源，指定一个 schema 要比让 Spark 来进行推断效率更高。在下面的代码中，我们提供一个 schema：

```
import org.apache.spark.sql.types._
......

// 数据源文件
val file = "./src/main/resources/people.csv"

// 构造 schema
val fields = Seq(
   StructField("p_name", StringType, nullable = true),
   StructField("p_age", LongType, nullable = true),
   StructField("p_job", StringType, nullable = true)
)
val schema = StructType(fields)

// 读取数据源，创建 DataFrame
val peopleDF = spark.read
   .option("sep", ";")                    // 字段使用;分隔符
   .option("header", "true")              // 说明有标题行
   .schema(schema)                        // 指定使用的 schema
```

```
    .csv(file)

peopleDF.printSchema()                    // 打印 schema
peopleDF.show()                           // 显示
```

输出结果如下所示：

```
```

可以看出，它返回了由行和命名列组成的 DataFrame，该 DataFrame 具有模式中指定的类型。

注：如果一个文件是由 DataFrame 以 Parquet 格式写出存储的，则模式被保留为 Parquet 元数据的一部分。在这种情况下，随后读入 DataFrame 不需要手动提供模式。Parquet 是一种流行的柱状格式，是 Spark 的默认格式；它使用 snappy 压缩来压缩数据。

也可以使用 csv 格式读取 tsv 文件。所谓 tsv 文件，指的是以制表符（tab）作为字段分隔符的文件。在下面的示例中，加载 tsv 文件到 DataFrame 中：

```
val file = "./src/main/resources/people.tsv"

// 读取数据源，创建 DataFrame
val peopleDF = spark.read
  .option("sep", "\t")                    // 字段使用 Tab 分隔符
  .option("inferSchema", "true")          // 指定模式自动推断
  .option("header", "true")               // 说明有标题行
  .csv(file)

peopleDF.printSchema()                    // 打印 schema
peopleDF.show()                           // 显示
```

输出结果如下所示：

```
```

然而，在使用 SparkSessionc 对象的 read 字段读取 CSV 时的选项列表很长，而且很难找到。所以在下表总结了在加载 CSV 文件时可以指定的 option 选项参数和值：

Option	默认值	说明	引入版本
sep	,	设置单个字符作为每个字段和值的分隔符	v2.0.0
encoding	UTF-8	根据给定的编码类型解码CSV文件	v2.0.0
quote	"	设置用于转义引用值的单个字符，其中分隔符可以是值的一部分。如果要关闭引号，需要设置的不是null，而是一个空字符串	v2.0.0
escape	\	设置用于转义已引用值中的引号的单个字符	v2.0.0
comment	空字符串	设置用于跳过以该字符开头的行的单个字符。默认情况下，它是禁用的	v2.0.0
header	false	是否使用第一行作为列的名称。不支持两行标题。	v2.0.0
inferSchema	false	告诉Spark是否尝试基于列值推断列类型（即从数据自动推断输入模式）。它需要额外传递一次数据。	v2.0.0
ignoreLeadingWhiteSpace	false	指示是否跳过正在读取的值中的前导空格	v2.0.0
ignoreTrailingWhiteSpace	false	指示是否跳过正在读取的值中的尾部空格	v2.0.0
nullValue	空字符串	设置null值的字符串表示形式。从2.0.1开始，这适用于所有受支持的类型，包括字符串类型。	v2.0.0
nanValue	NaN	设置非数字"值"的字符串表示形式	v2.0.0
positiveInf	Inf	设置正无穷值的字符串表示形式	v2.0.0
negativeInf	-Inf	设置负无穷值的字符串表示形式	v2.0.0
dateFormat	yyyy-MM-dd	设置指示日期格式的字符串。自定义日期格式遵循java.text.SimpleDateFormat的格式。这适用于日期(date)类型	v2.0.0

timestampFormat	yyyy-MM-dd'T'HH:mm:ss.SSSXXX	设置指示时间戳格式的字符串。自定义日期格式遵循java.text.SimpleDateFormat的格式。这适用于时间戳(timestamp)类型	v2.1.0
maxColumns	20480	定义一个记录可以有多少列的硬限制	
maxCharsPerColumn	-1	定义允许读取任何给定值的最大字符数。默认值是1000000	
mode	PERMISSIVE	允许在解析期间处理损坏记录的模式。它支持以下不区分大小写的模式。 - PERMISSIVE：当遇到损坏的记录时，将其他字段设置为null，并将格式错误的字符串放入由columnNameOfCorruptRecord配置的字段中。要保存损坏的记录，用户可以在用户定义的模式中设置一个名为columnNameOfCorruptRecord的字符串类型字段。如果一个模式没有该字段，它将在解析期间删除损坏的记录。当解析后的CSV tokens长度小于模式的预期长度时，它会为额外字段设置null。 - DROPMALFORMED：忽略整个损坏的记录。 - FAILFAST：当遇到损坏的记录时抛出异常。	v2.0.0
columnNameOfCorruptRecord		允许重命名新字段，该字段具有由PERMISSIVE模式创建的格式错误字符串。这将覆盖spark.sql.columnNameOfCorruptRecord。默认值是在spark.sql.columnNameOfCorruptRecord中指定的值	v2.2.0
multiLine		解析一条记录，它可能跨越多行	v2.2.0
wholeFile	false	解析一条记录，它可能跨越多行	

4.6.5 读取 JSON 文件创建 DataFrame

Spark SQL 可以自动推断 JSON Dataset 的模式，并将其加载为 Dataset[Row]。这种转换可以在 Dataset[String]或 JSON 文件上使用 SparkSession.read.json()完成。

注意，作为 json 文件提供的文件实际上并不是典型的 JSON 文件。每一行必须包含一个单独的、自包含的有效 JSON 对象。对于常规的多行 JSON 文件，将 multiLine 选项设置为 true。

读取 json 数据源文件时，Spark 会自动从 key 中自动推断模式，并相应地创建一个 DataFrame。因此，没有必要为 JSON 数据定义模式。此外，Spark 极大地简化了访问复杂 JSON 数据结构中的字段所需的查询语法。请看下面的示例。

```
// 数据源文件
// JSON 数据集路径可以是单个文件，也可以是存储文件的目录
val file = "./src/main/resources/people.json"

// 读取数据源，创建 DataFrame
val df = spark.read.json(file)            // json 解析；列名和数据类型隐式地推断

// schema
df.printSchema()

// 显示
df.show()
```

执行以上代码，输出结果如下：

```

```

当然，也可以明确指定一个 schema，覆盖 Spark 的推断 schema。如下面的代码所示：

```
import org.apache.spark.sql.types._
.....
```

```
// 数据源文件
// JSON 数据集路径可以是单个文件，也可以是存储文件的目录
val file = "./src/main/resources/people.json"

// 创建 schema。字段名称要与 json 对象的 key 名称保持一致
val fields = Seq(
  StructField("name",StringType,nullable = true),
  StructField("age",IntegerType,nullable = true)
)

// 读取数据源，创建 DataFrame，使用自定义的 schema
val df = spark.read.schema(StructType(fields)).json(file)

// schema
df.printSchema()

// 显示
df.show()
```

执行上面的代码，输出结果如下所示：

如果 json 格式解析错误，那么默认各行各列值都设为 null。在下面的示例代码中，我们在 schema 中将姓名这一列（name）对应的数据类型错误地设为 boolean 类型，因此 SparkSession 在解析时出现解析错误，因而整个 DataFrame 中的各行各列值均为 null：

```
import org.apache.spark.sql.types._
......

// 数据源文件
// JSON 数据集路径可以是单个文件，也可以是存储文件的目录
val file = "./src/main/resources/people.json"

// 创建 schema。字段名称要与 json 对象的 key 名称保持一致
val fields = Seq(
  StructField("name",StringType,nullable = true),
  StructField("age",IntegerType,nullable = true)
)

// 读取数据源，创建 DataFrame，使用自定义的 schema
val df = spark.read.schema(StructType(fields)).json(file)

// schema
df.printSchema()

// 显示
df.show()
```

输出结果如下所示：

但是这样处理 json 格式解析错误并不是一种好的方式，因为它经常会掩盖错误事实，让用户迷惑。

所以最好的方式是，如果 json 格式解析错误，那么就直接抛出异常（快速失败），而不是全都设为 null 值。在下面的代码中演示了这种处理方式：

```
import org.apache.spark.sql.types._
......

    // 数据源文件
    // JSON 数据集路径可以是单个文件，也可以是存储文件的目录
    val file = "./src/main/resources/people.json"

    // 创建 schema。字段名称要与 json 对象的 key 名称保持一致
    val fields = Seq(
      StructField("name",BooleanType,nullable = true),
      StructField("age",IntegerType,nullable = true)
    )

    // 读取数据源，创建 DataFrame，使用自定义的 schema
    val df = spark.read
            .option("mode","failFast")      // 指定 failFast 模式，告诉 Spark，当面对解析错误时，to fail fast
            .schema(StructType(fields))
            .json(file)

    // schema
    df.printSchema()

    // 显示
    df.show()          // 当执行一个 action 时，Spark 将抛出一个 RuntimeException
```

执行以上代码，输出结果如下所示：

```
......
```

可以看到，这时会立即抛出异常信息，从而不会对用户产生误导。

4.6.6 读取 Parquet 文件创建 DataFrame

Apache Parquet 文件是 Spark SQL 中直接支持的一种常见格式，它们非常节省空间，非常流行。Apache Parquet 是一种高效的、压缩的、面向列的开源数据存储格式。它提供了多种存储优化，允许读取单独的列而非整个文件，这不仅节省了存储空间而且提升了读取效率。它是 Spark 默认的文件格式，支持非常有效的压缩和编码方案，也可用于 Hadoop 生态系统中的任何项目，可以大大提高这类应用程序的性能。

Apache Spark 提供了对读取和写入 Parquet 文件的支持，这些文件自动保存原始数据的模式。Parquet 是一种非常流行的格式，在 Spark 中有一些额外的选项可以用于读写 Parquet 文件。在编写 Parquet 文件时，出于兼容性考虑，所有列都会自动转换为 nullable。

在下面的示例中，先读取 Parquet 文件内容到 DataFrame 中，然后打印其 schema 并输出数据：

```
    // 读取 Parquet 文件
    val parquetFile = "./src/main/resources/users.parquet"

    // Parquet 是默认的格式，因此当读取时我们不需要指定格式
    val usersDF = spark.read.load(parquetFile)
    // 如果我们想要更加明确，我们可以指定 parquet 函数
```

```
    // val usersDF = spark.read.parquet(parquetFile)

    usersDF.printSchema()
    usersDF.show()
```

执行以上代码，输出结果如下所示：

```
```

4.6.7 读取 ORC 文件创建 DataFrame

ORC 文件是一种自描述的、类型感知的列存储格式数据文件，它针对大型数据的读写进行了优化，也是大数据中常用的文件格式。Apache Spark 提供了对读取和写入 ORC 文件的支持。

下面的例子演示了如何读取 ORC 文件并创建 DataFrame：

```
    // 数据源文件
    val orcFile = "./src/main/resources/users.orc"

    // 读取 ORC 文件，构造 DataFrame
//      val orcDF = spark.read.orc(orcFile)    // 简洁写法
    val orcDF = spark.read.format("orc").load(orcFile)    // 简洁写法

    orcDF.printSchema()
    orcDF.show()
```

执行以上代码，输出结果如下所示：

```
```

从 Spark 2.3 开始，Spark 支持一个带有新的 ORC 文件格式的向量化 ORC reader。为此，新添加了以下配置。

属性名	默认值	含义
spark.sql.orc.impl	native	The name of ORC实现的名称。可以是native或hive。native意味着本地ORC支持，它构建在Apache ORC 1.4之上。`hive`意味着Hive 1.2.1中的ORC库。
spark.sql.orc.enableVectorizedReader	true	在native实现中启用向量化orc解码。：如果为false，则在native实现中使用新的非向量化ORC reader。

当 spark.sql.orc.impl 设置为 native 以及 spark.sql.orc.enableVectorizedReader 设置为 true 时，会将向量化的 reader 用于本地 ORC 表（例如，使用 USING ORC 子句创建的表）。对于 Hive ORC serde 表，当 spark.sql.hive.convertMetastoreOrc 也被设置为 true 时，也会使用向量化的 reader。

4.6.8 使用 JDBC 从数据库创建 DataFrame

Spark SQL 还包括一个可以使用 JDBC 从其他关系型数据库读取数据的数据源。开发人员可以使用 JDBC 创建来自其他数据库的 DataFrame，只要确保预定数据库的 JDBC 驱动程序是可访问的（需要在 spark 类路径中包含特定数据库的 JDBC 驱动程序）。

说明：Spark 安装程序默认是没有提供数据库驱动的，所以在使用前需要将相应的数据库驱动上传到$SPARK_HOME 目录下的 jars 目录中。本书示例使用的是 MySQL 数据库，所以使用前需要将相应的 mysql-connector-java-x.x.x.jar 包上传到$SPARK_HOME/jars 目录下。

如果是在 Maven 项目中开发，则需要在 pom.xml 中添加如下的依赖项：

```
<dependency>
    <groupId>mysql</groupId>
```

```
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.18</version>
</dependency>
```

如果是在 STB 项目中开发，则需要在 build.sbt 中添加如下的依赖项：

```
libraryDependencies += "mysql" % "mysql-connector-java" % "8.0.18"
```

说明：作者本地安装的 MySQL 数据库版本为“8.0.18”，请大家根据自己的数据库版本进行修改。

需要注意的是，如果是在使用 spark-shell 进行交互式操作的环境下，那么在使用 JDBC 从 Spark 读取 MySQL 数据库数据时，必须在启动 REPL shell 时指定驱动程序的 class path 路径，因此通常在启动 spark-shell 时，使用 driver-class-path 命令行参数指定 JDBC 驱动程序的路径，如下所示：

```
// 启动 shell，带有 driver-class-path 命令行参数
$ spark-shell --driver-class-path mysql-connector-java.jar --jars /usr/share/java/mysql-connector-java.jar
```

在下面的示例中，我们通过 JDBC 读取 MySQL 数据库中的一个 peoples 数据表，并创建 DataFrame。首先，在 MySQL 中执行如下脚本，创建一外名为 xueai8 的数据和一个名为 peoples 的数据表，并向表中插入一些样本记录。

```
mysql> create database xueai8;
mysql> use xueai8;
mysql> create table peoples(id int not null primary key, name varchar(20), age int);
mysql> insert into peoples values(1,"张三",23),(2,"李四",18),(3,"王老五",35);
mysql> select * from peoples;
```

然后编写如下的代码，来读取 peoples 表中的数据到 DataFrame 中。

```
    val DB_URL= "jdbc:mysql://localhost:3306/xueai8" +              // 数据库名为 xueai8
      "?useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true"
    val peoplesDF = spark.read.format("jdbc")
      .option("driver", "com.mysql.cj.jdbc.Driver")                 // 数据库驱动程序类名
      .option("url", DB_URL)                                        // 连接 url
      .option("dbtable", "peoples")                                 // 要读取的表
      .option("user", "root")                                       // 连接账户
      .option("password","admin")                                   // 连接密码
      .load()

    peoplesDF.printSchema()
    peoplesDF.show()
```

执行以上代码，输出结果如下所示：

```

```

也可以将 JDBC 参数放到一个 Map 集合中，做为 options 的参数传入。代码如下：

```
    val DB_URL= "jdbc:mysql://localhost:3306/cdadb" +           // 数据库名为 cdadb
      "?useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true"

    val jdbcMap = Map(
      "url" -> DB_URL,                  // jdbc url
      "dbtable" -> "peoples",           // 要读取的数据表
      "user" -> "root",                 // mysql 账号
      "password" -> "admin"             // mysql 密码
```

```
    )

    // 读取 JDBC 数据源，创建 DataFrame
    val peoplesDF = spark.read.format("jdbc").options(jdbcMap).load()

    peoplesDF.printSchema()
    peoplesDF.show()
```

也可使用 jdbc()快捷方法从关系型数据库中加载数据。例如，上面的示例可以改写为如下的代码：

```
    val DB_URL= "jdbc:mysql://localhost:3306/cdadb" +          // 数据库名为 cdadb
      "?useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true"

    // 创建一个 Properties()对象来保存参数
    import java.util.Properties
    val connectionProperties = new Properties()

    connectionProperties.put("user", "root")                    // 账号
    connectionProperties.put("password", "admin")               // 密码

    val driverClass = "com.mysql.cj.jdbc.Driver"                // mysql 8 驱动
//    val driverClass = "com.mysql.jdbc.Driver"                 // mysql 5 驱动
    connectionProperties.put("Driver", driverClass)
//    connectionProperties.setProperty("Driver", driverClass)   // 等价上一句

    // 使用快捷方式
    val peoplesDF = spark.read.jdbc(DB_URL, "peoples", connectionProperties)

    peoplesDF.printSchema()
    peoplesDF.show()
```

说明：

根据 MySQL 数据库的版本不同（主要是 MySQL 8 有较大的变化），相应的驱动程序名和 JDBC 的连接 URL 也发生变化。

如果读写的是 MySQL 5.x：

- 驱动程序为：com.mysql.jdbc.Driver。
- 连接的 URL 为：jdbc:mariadb://localhost:3306/db?useSSL=false"。

如果评定的是 MySQL 8.x：

- 驱动程序为：com.mysql.cj.jdbc.Driver。
- 连接的 URL 为：
  jdbc:mysql://localhost:3306/hostel?useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true"

还可以使用 query 选项指定用于将数据读入 Spark 的查询语句。指定的查询将被圆括号括起来，并在 FROM 子句中用作子查询。Spark 还将为子查询子句分配一个别名。例如，Spark 将向 JDBC 源发出如下形式的查询。

```
SELECT <columns> FROM (<user_specified_query>) spark_gen_alias
```

使用此选项时有一些限制。

- 不允许同时指定“dbtable”和“query”选项。
- 不允许同时指定“query”和“partitionColumn”选项。当需要指定“partitionColumn”选项时，可以使用“dbtable”选项指定子查询，分区列可以使用作为“dbtable”的一部分提供的子查询别名进行限定。

例如：

```
spark.read.format("jdbc")
            .option("url", jdbcUrl)
            .option("query", "select c1, c2 from t1")
            .load()
```

更多关于 JDBC 配置参数，请参考官方文档。

4.6.9 读取图像文件创建 DataFrame

随着用于图像分类和对象检测的深度学习框架的进展，Apache Spark 中对标准图像处理的需求也越来越大。图像处理和预处理有其特定的挑战，例如，图像有不同的格式(如 jpeg、png 等)、大小和颜色方案，而且没有简单的方法来测试正确性(静默失败)。图像数据源通过提供标准表示来解决这些问题，用户可以针对特定图像表示的细节进行编码和抽象。

Apache Spark 2.3 提供了 ImageSchema.readImages API，它最初是在 MMLSpark 库中开发的，在 Apache Spark 2.4 中成为了一个内置的数据源，因此更容易使用。ImageDataSource 实现了一个 Spark SQL 数据源 API，用于将图像数据作为 DataFrame 加载。

要想使用图像数据源，需要添加相关的依赖。如果使用的是 SBT 项目构建和管理工具的话，在 build.sbt 中添加如下依赖：

```
libraryDependencies += "org.apache.spark" %% "spark-mllib" % "2.4.7"
```

如果使用的是 Maven 项目构建和管理工具的话，在 pom.xml 中添加如下依赖：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-mllib_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
```

在下面的示例中，我们读取 images 目录下的所有图像到 DataFrame 中。

```
    val file = "./src/main/resources/images"
    val imageDF = spark.read.format("image").option("dropInvalid", "true").load(file)

    imageDF.printSchema()
    imageDF
      .select("image.origin", "image.width", "image.height","image.nChannels","image.mode")
      .show(truncate=false))
```

执行以上代码，输出结果如下所示：

4.6.10 读取 Avro 文件创建 DataFrame

从 Apache Spark 2.4 版本开始，Spark SQL 为读取和写入 Apache Avro 数据提供了内置支持，新增一个基于 Databricks 的 spark-avro 模块的原生 AVRO 数据源。

spark-avro 模块是外部的，默认情况下不包括在 spark-submit 或 spark-shell 中。要使用该模块，需要在使用 spark-submit 命令启动应用程序时，使用--packages 选项添加 spark-avro_2.11 及其依赖。例如，

```
$ ./bin/spark-submit --packages org.apache.spark:spark-avro_2.11:2.4.7 ...
```

同样，对于 spark shell，也可以使用--packages 选项添加 spark-avro_2.12 及其依赖。例如，

```
$ ./bin/spark-shell --packages org.apache.spark:spark-avro_2.11:2.4.7 ...
```

因为 spark-avro 模块是外部的，所以 DataFrameReader 或 DataFrameWriter 中没有.avro API。要想使用 Avro 数据源，需要添加相关的依赖。如果使用的是 SBT 项目构建和管理工具的话，在 build.sbt 中添加如下依赖：

```
libraryDependencies += "org.apache.spark" %% "spark-avro" % "2.4.7"
```

如果使用的是 Maven 项目构建和管理工具的话，在 pom.xml 中添加如下依赖：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-avro_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
```

在下面的示例中，我们读取 Spark 自带的 user.avro 数据源文件到 DataFrame 中。要以 Avro 格式加载/保存数据，需要将数据源的 format 选项指定为 avro

```
// 数据源文件
val file = "./src/main/resources/users.avro"

// 读取 Avro 文件，创建 DataFrame
val usersDF = spark.read.format("avro").load(file)

usersDF.printSchema()
usersDF.show()
```

执行以上代码，输出结果如下所示:

```

```

4.7 操作 DataFrame

在 Spark 2.0 中，Spark 将 DataFrame 看作是 Dataset 的一个特例，即 Dataset[Row]。Dataset/DataFrame 为结构化数据操作提供了一种特定于领域的语言（DSL）。其中 DataFrame 所支持的这些操作也被称为“无类型转换”，同时提供 Scala、Java、Python 和 R 语言的支持。而 Dataset 所支持的这些操作也被称为“类型化转换”，只提供强类型的 Scala 和 Java 语言支持的 API。

在本节中，我们将重点讨论可以在 DataFrame 上执行的各种操作。这些操作被分为两类，transformation 和 action。开发人员链接多个操作来选择、过滤、转换、聚合和排序在 DataFrame 中的数

据。底层的 Catalyst 优化器确保了这些操作的高效执行。

4.7.1 多种方式引用列

在学习操作 DataFrame 之前，首先需要掌握 Spark 所提供的引用 DataFrame 中的列的多种方式。在下面的代码中，演示了这些方式：

```
import org.apache.spark.sql.functions._
......

    import spark.implicits._

    // 使用元组序列创建一个 DataFrame
    val kvDF = Seq((1,2),(3,4)).toDF("key","value")
    // kvDF.printSchema()
    // kvDF.show()

    // 要在一个 DataFrame 中显示列名，可以调用 columns 函数
    kvDF.columns

    // 以不同的方式选择特定的列
    kvDF.select("key").show                    // 列为字符串类型
    kvDF.select(col("key")).show               // col 是内置函数，它返回 Column 类型
    kvDF.select(column("key")).show            // column 是内置函数，它返回 Column 类型
    kvDF.select(expr("key")).show              // expr 与 col 方法调用相同
    kvDF.select($"key").show                   // Scala 中构造 Column 类型的语法糖
    kvDF.select('key).show                     // 同上

    // 也可以使用 DataFrame 的 col 函数
    kvDF.select(kvDF.col("key")).show
```

执行上面的代码，都输出相同的内容：

```
+---+
|key|
+---+
|  1|
|  3|
+---+
```

下面的代码选择 key 列，并增加一个新的列，新列的值由 key 列计算得来，为 boolean 值，表示 key 列的值是否大于 1：

```
    kvDF.select('key,'key > 1).show
```

输出内容如下所示：

```
+---+---------+
|key|(key > 1)|
+---+---------+
|  1|    false|
|  3|     true|
+---+---------+
```

也可以给列取一个别名，代码如下所示：

```
// 给列一个别名
kvDF.select('key,'key > 1 as "aa").show
// 或（等价）
kvDF.select('key,('key > 1).alias("aa")).show
```

输出内容如下所示：

```
+---+-----+
|key|   aa|
+---+-----+
|  1|false|
|  3| true|
+---+-----+
```

4.7.2 对 DataFrame 进行转换操作

在演示 DataFrame 的各种操作方法之前，我们先准备好数据。首先加载数据集到 DataFrame 中，代码如下所示：

```
// 加载电影数据集文件到 DataFrame 中
val file = "./src/main/resources/movies.csv"
val movies = spark.read
   .option("header","true")
   .option("inferSchema","true")
   .csv(file)

movies.printSchema()
movies.show(5)
```

执行以上代码，输出结果如下所示：

```
```

DataFrame API 提供有许多函数用来执行关系运算，这些函数模拟了 SQL 关系操作：

- 选择数据：select
- 删除某列：drop
- 过滤数据：where 和 filter(同义的)
- 限制返回的数量：limit
- 重命名列：withColumnRenamed
- 增加一个新的列：withColumn
- 数据分组：groupBy
- 数据排序：orderBy 和 sort（等价的）

接下来，学习 DataFrame 的各种转换操作函数。

1) select 函数：选择指定的列。

下面的代码选取原 DF 中的两列，返回一个新的 DataFrame：

```
movies.select("title","year").show(5)
```

输出结果如下：

下面的代码使用 column 表达式将电影上演的年份转换到年代表示，给赋予一个别名：

```
import spark.implicits._
movies.select($"title",($"year"-$"year" % 10).as("decade")).show(5)
```

输出结果如下所示：

2) selectExpr：使用 SQL 表达式选择列。

下面的代码中，用通配符星号（*）来表示选择所有的列，并增加一个新的列“decade”，新列的值是通过对 year 列值计算得到的电影上映的年代：

```
movies.selectExpr("*","(year - year % 10) as decade").show(5)
```

输出内容如下所示：

下面的代码中使用了 SQL 表达式和内置函数，用来查询电影数量和演员数量这两个值：

```
movies.selectExpr("count(distinct(title)) as movies","count(distinct(actor)) as actors").show()
```

输出结果如下所示：

```
+------+------+
|movies|actors|
+------+------+
|  1409|  6527|
+------+------+
```

可以看出，数据集中的电影共有 1409 部，演员共有 6527 名。

3) filter, where：实现过滤。这两个函数等价。

首先找出 2000 年以前上映的电影，在 filter 函数或 where 函数中指定过滤条件：

```
import spark.implicits._
movies.filter($"year" < 2000).show(5)
// movies.where($"year" < 2000).show(5)          // 等价
```

输出结果如下所示：

找出 2000 年及以后上映的电影：

```
import spark.implicits._
movies.filter($"year" >= 2000).show(5)
movies.where($"year" >= 2000).show(5)
```

输出结果如下所示：

找出 2000 年上映的电影。注意，在 Scala 中，相等比较要求 3 个等号。

```
import spark.implicits._
movies.filter($"year" === 2000).show(5)
movies.where($"year" === 2000).show(5)
```

输出结果如下所示：

找出非 2000 年上映的电影。注意，在 Scala 中，不相等比较使用的操作符是 =!=。

```
import spark.implicits._
movies.select("title","year").filter('year =!= 2000).show(5)
movies.select("title","year").where('year =!= 2000).show(5)
```

输出结果如下所示：

找出 2001 年到 2002 年间上映的电影。

```
import spark.implicits._
movies.filter('year.isin(2001,2002)).show()
movies.where('year.isin(2001,2002)).show()
```

输出结果如下所示：

可使用 OR 和 AND 运算符组合一个或多个比较表达式。在下面的代码中，我们找出 2000 年及以后上映，并且电影名称少于 5 个字符的电影。

```
import spark.implicits._
movies.filter('year >= 2000 && length('title) < 5).show(5)
movies.where('year >= 2000 && length('title) < 5).show(5)
```

另一种实现相同结果的方法是调用 filter 函数两次：

```
import spark.implicits._
movies.where('year >= 2000)
        .where(length('title) < 5)
        .show(5)
```

输出结果如下所示：

4) distinct, dropDuplicates：去重，保持唯一值。

执行下面的代码，找出数据集中共有多少部电影：

```
import spark.implicits._
movies.select("title").distinct.selectExpr("count(title) as movies").show()
movies.dropDuplicates("title").selectExpr("count(title) as movies").show()      // 与上句等价
```

输出结果如下所示：

```
+------+
|movies|
+------+
|  1409|
+------+
```

5) sort(columns), orderBy(columns)：对指定的列排序。

执行下面的代码，先对电影标题去重，并选择指定的三个字段：

```
val movieTitles = movies
   .dropDuplicates("title")
   .selectExpr("title", "length(title) as title_length", "year")
```

接下来按电影名称长度排序显示，默认是升序：

```
import spark.implicits._
movieTitles.sort('title_length).show(5)
```

输出结果如下所示：

按电影名称长度排序降序显示：

```
import spark.implicits._
movieTitles.sort('title_length.desc).show(5)
```

输出结果如下所示：

输出时，先按电影名称长度降序排序，再按上映日期升序排序：

```
import spark.implicits._
movieTitles.orderBy('title_length.desc, 'year).show(5)
```

输出结果如下所示：

6）groupBy()：分组统计。

统计每年上映的电影数量，统计排名靠前的 5 个年份。

```
import spark.implicits._
movies.groupBy("year").count()
        .orderBy($"count".desc)
        .show()
```

执行上面的代码，输出内容如下：

统计上映电影数量超过 2000 部的年份。

```
movies.groupBy("year").count()
        .where($"count" > 2000)
        .show()
```

执行上面的代码，输出内容如下：

7）limit(n)：限制返回的行数。

按演员名字的长度排序，并获得 top 5：

```
import spark.implicits._
val actorNameDF = movies
  .select("actor")
  .distinct
  .selectExpr("actor", "length(actor) as length")
  .orderBy($"length".desc)
  .limit(5)
  .show(false)
```

输出结果如下所示：

8）union(otherDataFrame)：与另一个 DataFrame 进行合并，相当于 SQL 中的 union all。

假设我们现在想在名称为“12”的电影中添加一个缺失的演员，那么创建另一个 DataFrame，其中包含所缺失的演员信息。然后将原数据集与这个含有缺失演员信息的新数据集执行一个 union 操作，将两个数据集合并在一起。

执行下面的代码，获得电影“12”的数据集：

```
import spark.implicits._
val shortNameMovieDF = movies.where($"title" === "12")
shortNameMovieDF.show()
```

输出结果如下所示：

接下来，创建另一个 DataFrame，包含所缺失演员“Brychta,Edita”的信息。然后将这个 DataFrame 和上面的 DataFrame 进行合并，这样就将演员“Brychta,Edita”的信息合并到上面的数据集中了。

```
// 另创建一个 DataFrame
val forgottenActor = Seq(("Brychta, Edita", "12", 2007L))
val forgottenActorDF = forgottenActor.toDF("actor","title","year")

// 通过合并两个 DataFrame，实现添加缺失的演员姓名
val completeShortNameMovieDF = shortNameMovieDF.union(forgottenActorDF)
completeShortNameMovieDF.show()
```

输出结果如下所示：

9）withColumn(colName, column)：向 DataFrame 增加一个新的列。

执行下面的代码，向 movies 这个 DataFrame 增加一个新列“decade”，该列的值是基于“produced_year”这个列的表达式，计算的结果是该电影上映的年代。

```
import spark.implicits._
movies.withColumn("decade", $"year" - $"year" % 10).show(5)
```

输出结果如下所示：

如果传给 withColumn 函数的列名与现有列名相同，则意思着用新值替换旧值。执行下面的代码，替换“produced_year”列的值为年代（原值为年份）：

```
import spark.implicits._
movies.withColumn("year", $"year" - $"year" % 10).show(5)
```

输出结果如下所示：

10）withColumnRenamed(existingColName, newColName)：修改列名。

执行下面同的代码，修改三个列的列名：

```
movies
  .withColumnRenamed("actor", "actor_name")
  .withColumnRenamed("title", "movie_title")
  .withColumnRenamed("year", "produced_year")
  .show(5)
```

输出结果如下所示：

11）drop(columnName1, columnName2)：删除指定的列。

执行下面的代码，从 DataFrame 中删除指定的列。

```
movies.drop("actor", "me").printSchema()
movies.drop("actor", "me").show(5)
```

输出结果如下所示：

从输出结果可以看出，“actor”这一列被删除了，而“me”这一列在原 DataFrame 中并不存在，所以删除不存在的列没有任何影响。

12）sample(fraction), sample(fraction, seed), sample(withReplacement,fraction)：抽样

执行带有无放回和 fraction 的抽样：

```
movies.sample(false, 0.0003).show(3)
```

执行带有有放回和 fraction、seed 的抽样：

```
movies.sample(true, 0.0003, 123456).show(3)
```

执行以上代码，输出内容如下：

13）randomSplit(weights)：随机分割数据集，其中参数 weights 是一个数组 Array，其元素代表各部分所占的比例。

执行下面的代码，将数据集分割为三部分，比例分别为 0.6、0.3 和 0.1：

```
val smallerMovieDFs = movies.randomSplit(Array(0.6, 0.3, 0.1))
smallerMovieDFs(0).show(3)
smallerMovieDFs(1).show(3)
smallerMovieDFs(2).show(3)

// 看看各部分计数之和是否等于 1
// 数据集总数量
println(movies.count())
// 分割后的第 1 个数据集的数量
println(smallerMovieDFs(0).count())
// 3 个数据集之和
println(smallerMovieDFs(0).count() + smallerMovieDFs(1).count() + smallerMovieDFs(2).count()  )
```

输出结果如下所示：

4.7.3 对 DataFrame 进行 action 操作

与 RDD 类似，当在 DataFrame 上执行 action 操作时，会触发真正的计算。这些 action 操作相对都比较简单，在下面的代码中演示了这些 action 方法的使用：

```
// 查看前 5 条数据。第 2 个参数指定当列内容较长时，是否截断显示，false 为不截断
movies.show(5,false)

// 返回数据集中的数量
movies.count
```

```
// 返回数据集中第 1 条数据
movies.first()

// 等价于 first 方法
movies.head()

// 返回数据集中前 3 条数据，以 Array 形式
movies.head(3)

// 返回数据集中前 3 条数据，以 Array 形式
movies.take(3)

// 返回数据集中前 3 条数据，以 List 形式
movies.takeAsList(3)

// 返回一个包含数据集中所有行的数组
movies.collect

// 返回一个包含数据集中所有行的数组，以 List 形式
movies.collectAsList

// 返回数据集的数据类型，以 Array 形式
df.types

// 返回数据集的列名，以 Array 形式
df.columns
```

另外还有一个 describe 函数，用来计算数字列和字符串列的基本统计信息，包括 count、mean、stddev、min 和 max。如果没有给定列，则此函数计算所有数值列或字符串列的统计信息。该函数返回的也是一个 DataFrame。请看下面的代码：

```
val descDF = df.describe()
descDF.printSchema()
descDF.show()
```

输出结果如下所示：

```

```

下面指定计算“year”列的基本统计信息：

```
val descDF = df.describe("year")
descDF.printSchema()
descDF.show()
```

输出结果如下所示：

```

```

下面的代码计算“year”和“actor”这两列的基本统计信息：

```
val descDF = df.describe("year","actor")
descDF.printSchema()
descDF.show()
```

输出结果如下所示：

这个函数用于探索性数据分析，它不能保证结果数据集模式的向后兼容性。如果希望以编程方式计算汇总统计信息，请使用 agg 函数。

Spark 还提供了一个与 describe()函数类似的 summary()函数，用来提供数据集的摘要信息。如果没有给出统计信息，这个函数将计算 count、mean、stddev、min、近似四分位数(25%、50%和 75%的百分位数)和 max。其用法如下下面的代码所示。

```
val summaryDF = df.summary()
summaryDF.printSchema()
summaryDF.show()
```

执行以上代码，输出结果如下所示：

也可以指定想要的统计信息。如下面的代码所示：

```
val summaryDF = df.summary("count", "min", "25%", "75%", "max")
summaryDF.printSchema()
summaryDF.show()
```

执行以上代码，输出结果如下所示：

要对指定的列做一个摘要，首先选择它们，然后再执行 summary()方法。

```
val summaryDF = df.select("year").summary()
summaryDF.printSchema()
summaryDF.show()
```

执行以上代码，输出结果如下所示：

取出 DataFrame Row 中特定字段的方法有两个：

1）getAS 方法。

```
testDF.foreach{
  line =>
    val col1 = line.getAs[String]("col1")
    val col2 = line.getAs[String]("col2")
}
```

在下面这个示例中，加载数据源文件，并输出每个字段的值。

```
// 1）创建 SparkSession
val spark = SparkSession
  .builder
  .master("local[*]")
  .appName("Spark SQL example")
  .getOrCreate()

// 2）加载数据源，构造 DataFrame
val input = "./src/main/resources/people.json"
val df = spark.read.json(input)
```

```
    // 3）取每个字段的值
    df.foreach{
      row =>
        val col1 = row.getAs[String]("name")
        val col2 = row.getAs[Long]("age")
        println(col1 + "\t" + col2)
    }
```

执行上面的代码，输出结果如下所示：

```
Michael    0
Andy       30
Justin     19
```

2）模式匹配。

```
testDF.map{
      case Row(col1:String,col2:Int) =>
        println(col1); println(col2)
        col1
      case _ =>
        ""
    }
```

请看下面的示例代码。

```
    // 1）创建 SparkSession
    val spark = SparkSession
      .builder
      .master("local[*]")
      .appName("Spark SQL basic example")
      .getOrCreate()

    // 2）加载数据源，构造 DataFrame
    val input = "./src/main/resources/people.json"
    val df = spark.read.json(input)

    // 3）通过模式匹配取每个字段的值
    df.collect().map{
      case Row(col1:Long,col2:String) =>
        println(col1 + "\t" + col2)
      case _ => ""
    }
```

执行上面的代码，输出结果如下：

```
30   Andy
19   Justin
```

4.7.4 操作 DataFrame 示例

下面通过一个示例来演示 DataFrame 的 transformation 和 action 操作。

【示例】给出一个员工信息名单，找出收入最高的前 3 名员工（Top N 问题）。

样本数据 employees.csv 内容如下：

```
张三,paramedic i/c,fire,f,salary,,91080.00,
```

```
李四,lieutenant,fire,f,salary,,114846.00,
王老五,sergeant,police,f,salary,,104628.00,
赵六,police officer,police,f,salary,,96060.00,
钱七,clerk iii,police,f,salary,,53076.00,
周扒皮,firefighter,fire,f,salary,,87006.00,
吴用,law clerk,law,f,hourly,35,,14.51
```

其中各个字段的含义依次如下：

“姓名,职业,部门,全职或兼职;固定薪资或计时工资,工作时长,年薪,每小时工资”。

请注意数据集中，“吴用”没有年薪，是计时薪资，所以有最后一个“每小时工资”的字段。而其他人都是固定薪资，所以有“年薪”字段，但是没有“每小时工资”字段。

实现代码如下。

```
// 创建 SparkSession 实例
val spark:SparkSession = SparkSession.builder()
  .master("local[*]")
  .appName("Spark DataFram Demo")
  .getOrCreate()

// 定义文件路径
val file = "src/main/resources/employees.csv"

// 指定列名
val columns = Array("uname", "designation", "department", "jobtype", "NA", "NA2", "salary", "NA3")

// 加载数据文件，并构造 DataFrame
val df = spark.read
  .option("inferSchema", "true")
  .option("header","false")
  .csv(file)
  .toDF(columns:_*)

// 按 salary 列降序排列，并显示
import spark.implicits._
df.orderBy($"salary".desc).show(3)
```

执行以上代码，得到如下的结果：

```
+------+--------------+----------+-------+------+----+--------+----+
| uname|   designation|department|jobtype|    NA| NA2|  salary| NA3|
+------+--------------+----------+-------+------+----+--------+----+
|  李四|     lieutenant|      fire|      f|salary|null|114846.0|null|
|王老五|       sergeant|    police|      f|salary|null|104628.0|null|
|  赵六|police officer|    police|      f|salary|null| 96060.0|null|
+------+--------------+----------+-------+------+----+--------+----+
```

4.8 存储 DataFrame

有时，我们需要将 DataFrame 中的数据写到外部存储系统中，例如，本地文件系统、HDFS 或 Amazon S3。在一个典型的 ETL 数据处理作业中，处理结果很可能需要被写到一些存储系统中。

4.8.1 保存 DataFrame

在 Spark SQL 中，org.apache.spark.sql.DataFrameWriter 类负责将 DataFrame 中的数据写入外部存储系统。在 DataFrame 中有一个变量 write，它实际上就是 DataFrameWriter 类的一个实例。与 DataFrameWriter 交互的模式与 DataFrameReader 的交互模式有点类似。

下面描述了与 DataFrameWriter 交互的常见模式：

```
movies.write.format(...).mode(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save(path)
```

与 DataFrameReader 相似，可以使用 json、orc 和 parquet 文件存储格式，默认格式是 parquet。需要注意的是，save 函数的输入参数是目录名，不是文件名，它将数据直接保存到文件系统，如 HDFS、Amazon S3、或者一个本地路径 URL。

这些方法大多有相应的快捷方式，如 df.write.csv()、df.write.json()、df.write.orc()、df.write.parquet()、df.write.jdbc()等。这些方法相当于先调用 format()方法，再调用 save()方法。

【示例】先读取 json 数据文件，然后进行简单计算，把结果 DataFrame 保存到 csv 存储文件中，最后再加载这个结果文件到 RDD 中。

```
val spark = SparkSession.builder()
   .master("local[*]")
   .appName("Spark Write Demo")
   .getOrCreate()

// 读取 json 文件源，创建 DataFrame
val file = "src/main/resources/people.json"
val df = spark.read.json(file)
// df.show()

// 找出 age 不是 null 的信息，保存到 csv 文件中
import spark.implicits._
val output = "tmp/people-csv-output"
// df.where($"age".isNotNull).write.format("csv").save(output)
df.where($"age".isNotNull).write.csv(output)      // 等价上一句

// 将保存的 csv 数据再次加载到 RDD 中
val textFile = spark.sparkContext.textFile(output)
textFile.foreach(println)
```

注：write.format()支持输出 json、parquet、jdbc、orc、libsvm、csv、text 等格式文件，如果要输出 text 文本文件，可以采用 write.format("text")。但是，需要注意，只有 select()中只存在一个列时，才允许保存成文本文件，如果存在两个列，比如 select("name", "age")，就不能保存成文本文件。

Spark 支持通过 JDBC 方式连接到其他数据库，并将 DataFrame 存储到数据库中。下面这个示例中，将分析结果 DataFrame 写出到 MySQL 数据库中。

【示例】将结果 DataFrame 保存到 MySQL 的数据表中。

首先，在 MySQL 中新建一个测试 Spark 程序的数据库，数据库名称是“spark”，表的名称是“student”。请登录 MySQL 数据库，执行以下 SQL 语句。

```
mysql> create database spark;
mysql> use spark;
mysql> create table student (id int(4), name char(20), gender char(4), age int(4));
```

```
mysql> insert into student values(1,'张三','F',23);
mysql> insert into student values(2,'李四','M',18);
mysql> select * from student;
```

然后，编写 Spark 应用程序连接 MySQL 数据库并且向 MySQL 写入数据。在这里，我们向 spark.student 表中插入两条记录。

```
// 首先导入依赖的包：
import java.util.Properties
import org.apache.spark.sql.types._
import org.apache.spark.sql.Row
import org.apache.spark.sql.SaveMode
...

        // 下面我们设置两条数据表示两个学生信息：
        val studentRDD = spark.sparkContext
                .parallelize(Array("3,王老五,F,44","4,赵小虎,M,27"))
                .map(_.split(","))

        // 下面创建 Row 对象，每个 Row 对象都是 rowRDD 中的一行
        // RDD[String] => RDD[Row]
        val rowRDD = studentRDD.map(stu => Row(stu(0).toInt, stu(1).trim, stu(2).trim, stu(3).toInt))

        // 下面要设置模式信息：
        val schema = StructType(
                Seq(
                  StructField("id", IntegerType, nullable = true),
                  StructField("name", StringType, nullable = true),
                  StructField("gender", StringType, nullable = true),
                  StructField("age", IntegerType, nullable = true)
                )
        )

        // 建立起 Row 对象和模式之间的对应关系，也就是把数据和模式对应起来
        val studentDF = spark.createDataFrame(rowRDD, schema)
        // studentDF.printSchema()
        // studentDF.show()

        // 下面创建一个 prop 变量用来保存 JDBC 连接参数
        val prop = new Properties()
        prop.put("user", "root")                                    // 表示用户名是 root
        prop.put("password", "admin")                               // 表示密码是 hadoop
        prop.put("driver","com.mysql.jdbc.Driver")                  // 表示驱动程序是

        //下面就可以连接数据库，采用 append 模式，表示追加记录到数据库 spark 的 students 表中
        val DB_URL= "jdbc:mysql://localhost:3306/spark?useSSL=false"        // 数据库名为 cda
        studentDF.write
                .mode(SaveMode.Append)                              // .mode("append")
                .jdbc(DB_URL, "student", prop)
```

注：

如果是使用 spark-shell 命令行执行以上写入操作，请按以下说明执行。

1) 首先，将 mysql 驱动程序拷贝到$SPARK_HOME/jars 目录下：

```
$ cp ~/software/mysql-connector-java-5.1.38-bin.jar $SPARK_HOME/jars
```

2) 启动 Spark Shell 时，必须带有--driver-class-path 作为命令行参数，用来指定 mysql 连接驱动 jar 包。

```
$ spark-shell --driver-class-path ~/software/mysql-connector-java-5.1.38-bin.jar
```

3) 启动进入 spark-shell 以后，可以执行以下命令连接数据库，读取数据，并显示：

```
val jdbcDF = spark.read.format("jdbc")
                      .option("url", "jdbc:mysql://localhost:3306/spark?useSSL=false")
                      .option("driver","com.mysql.jdbc.Driver")
                      .option("dbtable", "student")
                      .option("user", "root")
                      .option("password", "admin")
                      .load()
```

或者：

```
val jdbcDF = spark.read.format("jdbc")
                      .options(
                       Map(
                         "url" -> "jdbc:mysql://localhost:3306/spark?useSSL=false",
                         "driver" -> "com.mysql.jdbc.Driver",
                         "dbtable" -> "student",
                         "user" -> "root",
                         "password" -> "admin"
                       )
                      ).load

jdbcDF.show()
```

上面的代码以 MySQL 5 为例。如果使用的是 MySQL 8 数据库，需要相应地修改 URL 连接及驱动程序名称。请参考 4.5.8 节修改。

4.8.2 存储模式

DataFrameWriter 类中的一个重要选项是 save mode，它表示存储模式。在将 DataFrame 中的数据写出到存储系统上时，默认行为是创建一个新表。如果指定的输出目录或同名表已经存在的话，则抛出错误消息。可以使用 Spark SQL 的 SaveMode 特性来更改此行为。

下表列出了各种支持的存储模式（save mode）：

模式	说明
append	将DataFrame数据追加到已经存在于指定目标位置下的文件列表
overwrite	使用DataFrame数据完全覆盖已经存在于指定目标位置下的任何数据文件
error errorIfExists	这是默认模式 如果指定的目标位置存在，那么DataFrameWriter将抛出一个错误
ignore	如果指定的目标位置存在，则简单地什么都不做。换句话说，不写出DataFrame中的数据

这些保存模式不使用任何锁定，也不是原子性的。此外，在执行 overwrite 时，将在写入新数据之前删除数据。

请看下面的示例：

```
// 将数据以 CVS 格式写出，但是使用'#'作为分隔符
movies.write.format("csv").option("sep", "#").save("/tmp/output/csv")
```

```
// 使用 overwrite save mode 写出数据
movies.write.format("csv").mode("overwrite").option("sep", "#").save("/tmp/output/csv")
```

4.8.3 控制输出的分区数量

Spark SQL DataFrame API 在转换操作执行 shuffling 时增加分区。触发 shuffle 的 DataFrame 操作是 join()、union()和所有聚合函数。

在下面的示例中，注意是如何查看一个 DataFrame 拥有的分区数量的。

```
import spark.implicits._

val simpleData = Seq(
      ("张三","销售部","北京",90000,34,10000),
      ("李四","销售部","北京",86000,56,20000),
      ("王老五","销售部","上海",81000,30,23000),
      ("赵老六","财务部","上海",90000,24,23000),
      ("钱小七","财务部","上海",99000,40,24000),
      ("周扒皮","财务部","北京",83000,36,19000),
      ("孙悟空","财务部","北京",79000,53,15000),
      ("朱八戒","市场部","上海",80000,25,18000),
      ("沙悟净","市场部","北京",91000,50,21000)
   )
val df = simpleData.toDF("employee_name","department","city","salary","age","bonus")

println(s"shuffle 前的分区数：${df.rdd.getNumPartitions}")

// groupBy 操作会触发数据 shuffle
val df2 = df.groupBy("city").count()

println(s"shuffle 后的分区数：${df2.rdd.getNumPartitions}")
```

执行上面的代码，输出结果如下：

```
shuffle 前的分区数：2
shuffle 后的分区数：200
```

从输出结果可以看出，经过 shuffle 操作后，DataFrame 的默认分区数变为 200。当 Spark 操作执行数据 shuffling（join()，union()，aggregation 函数）时，DataFrame 会自动将分区数增加到 200。这个默认的 shuffle 分区数来自 Spark SQL 配置 spark.sql.shuffle.partitions，默认设置为 200。

可以使用 SparkSession 对象的 conf 方法或使用 Spark Submit 命令配置来修改这个默认的 shuffle 分区数。例如：

```
spark.conf.set("spark.sql.shuffle.partitions",100)
println(df.groupBy("city").count().rdd.partitions.length)          // 100
```

输出到指定输出目录的文件数量与 DataFrame 拥有的分区数量相对应。在下面的代码示例中，我们加载了一个电影数据集，并将其重分区为 4 个分区，然后保存到指定的位置。

```
// 加载电影数据集文件到 DataFrame 中
val file = "./src/main/resources/movies.csv"

val movies = spark.read
```

```
        .option("header","true")
        .option("inferSchema","true")
        .csv(file)

val movies2 = movies.repartition(4)                    // 将 DataFrame 重分区为 4 个分区
val partitions = movies2.rdd.getNumPartitions          // 查看分区数
println("movies2 的分区数是：" + partitions)

// 保存 movies2
movies2.write.csv("tmp/movies-out-partitions")
```

执行上面的代码，输出结果如下图所示：

在某些情况下，DataFrame 的内容并不大，并且需要写入单个文件。这时可以使用 coalesce 转换函数，将 DataFrame 的分区数量减少到 1，然后把它写出。将上面示例代码中的最后一行修改如下：

```
// 保存 movies2
movies2.coalesce(1).write.csv("tmp/movies-out-partitions")
```

再一次执行代码，这时输出结果文件数如下图所示：

在 Spark SQL 中写出数据时也可以使用分区技术。DataFrameWriter 有一个 partitionBy()方法，它指定数据在写入到磁盘前如何先进行分区。

在 movies 这个 DataFrame 中，“year”列是最适合用来进行分区的候选列。假设我们要写出 movies 这个 DataFrame，用“year”列来分区。DataFrameWriter 将把所有具有相同年份的电影都写在同一个目录中。输出文件夹中的目录数量将对应于 movies 这个 DataFrame 中的年份的数量。

下面的示例中使用 partitionBy 函数来指定 DataFrame 输出时的分区数。

```
// 创建 SparkSession 的实例
val spark = SparkSession.builder()
  .master("local[*]")
  .appName("Spark Basic Example")
  .getOrCreate()

// 加载电影数据集文件到 DataFrame 中
val file = "./src/main/resources/movies.csv"

val movies = spark.read
  .option("header","true")
  .option("inferSchema","true")
  .csv(file)

import spark.implicits._
val outpath = "tmp/movies-out-partitions"
// 输出 movies DataFrame，使用 Parquet 格式并按 year 列进行分区
movies.repartition($"year").write.partitionBy("year").save(outpath)
```

注意，在上面的代码中，我们先用 repartition()方法在内存中按“year”进行分区，然后再调用 repartitionBy()方法按相同的“year”列进行物理分区。这是通常的做法，有利于优化物理分区速度。

执行以上代码，输出结果如下：

可以看到，“tmp/movies-out-partitions”目录下将包含多个子目录，从 year=1961 到 year=2012。图中只截取了其中部分显示。

根据数据集大小、CPU 核数量和内存大小，Spark shuffling 可能对作业有利也可能有害。当处理较少的数据量时，通常应该减少 shuffle 分区，否则将最终得到许多分区文件，每个分区中的记录数量较少。这将导致运行许多需要处理的数据较少的任务。

另一方面，当有太多的数据和较少的分区数同会导致长时间运行的少量任务，这时也可能会得到内存错误。

获得正确的 shuffle 分区大小总是很棘手，需要多次尝试不同的值来获得最优的分区数。当在 Spark 作业上遇到性能问题时，这是需要查找的关键属性之一。

该如何设置分区数量

假设我们要对一个大数据集进行操作，该数据集的分区数也比较大，那么当我们进行一些操作之后，比如 filter 过滤操作、sample 采样操作，这些操作可能会使结果数据集的数据量大量减少。但是 Spark 却不会对其分区进行调整，由此会造成大量的分区没有数据，并且向 HDFS 读取和写入大量的空文件，效率会很低，这种情况就需要我们重新调整分数数量，以此来提升效率。

通常情况下，结果集的数据量减少时，其对应的分区数也应当相应地减少。那么该如何确定具体的分区数呢？

- 分区过少：将无法充分利用群集中的所有可用的 CPU core
- 分区过多：产生非常多的小任务，从而会产生过多的开销

在这两者之间，第一个对性能的影响相对比较大。对于小于 1000 个分区数的情况而言，调度太多的小任务所产生的影响相对较小。但是，如果有成千上万个分区，那么 Spark 会变得非常慢。

Spark 中的 shuffle 分区数是静态的。它不会随着不同的数据大小而变化。上文提到：默认情况下，控制 shuffle 分区数的参数 spark.sql.shuffle.partitions 值为 200，这将导致以下问题：

- 对于较小的数据，200 是一个过大的选择，由于调度开销，通常会导致处理速度变慢。
- 对于大数据，200 很小，无法有效使用群集中的所有资源。

一般情况下，我们可以通过将集群中的 CPU 数量乘以 2、3 或 4 来确定分区的数量。如果要将数据写出到文件系统中，则可以选择一个分区大小，以创建合理大小的文件。

如果写入产生小文件数量过多，这时会产生大量的元数据开销。Spark 和 HDFS 一样，都不能很好的处理这个问题，这被称为“小文件问题”。同时数据文件也不能过大，否则在查询时会有不必要的性能开销，因此要把文件大小控制在一个合理的范围内。

在 Spark 中，有两种方法可以控制输出文件的大小：

- 可以通过分区数量来控制生成文件的数量，从而间接控制文件大小。
- Spark 2.2 引入了一种新的方法，以更自动化的方式控制文件大小，这就是 maxRecordsPerFile 参数，通过它可以控制写入文件的记录数来控制文件大小。

例如，我们可以在写出 DataFrame 结果时，指定如下的选项，Spark 将确保每个输出文件最多包含 5000 条记录。

```
df.write.option(“maxRecordsPerFile”, 5000)
```

4.9 使用类型化的 Dataset

从 Spark2.0 开始，Spark 整合了 Dataset 和 DataFrame，前者是有明确类型的数据集，后者是无明确类型的数据集。DataFrame 实际上是 Dataset[Row]类型，DataFrame 每一行的类型是 Row（Row 是一个通用的非类型化 JVM 对象，不解析的话无法得知每一行的字段名和对应的字段类型）。相反，Dataset 是强类型 JVM 对象的集合。

4.9.1 了解 Dataset

Dataset 是在 Spark 1.6 中添加的一个新接口，它是 Dataframe API 的扩展，是新的抽象。它同时提供了 RDD（强类型，能够使用强大的 lambda 函数）的优点和 Spark SQL 的优化执行引擎的优点。可以从 JVM 对象构造 Dataset，然后使用函数转换(map、flatMap、filter 等)进行操作。Dataset API 可在 Scala 和 Java 中使用。Python 不支持 Dataset API。

Dataset 由类型化对象组成，这意味着转换语法错误（比如方法名中的拼写错误）和分析错误（比如不正确的输入变量类型）可以在编译时捕获。DataFrame 由无类型的 Row 对象组成，这意味着在编译时只能捕获语法错误。Spark SQL 由一个字符串组成，这意味着语法错误和分析错误只在运行时被捕获。Dataset 可以更快地捕获错误，从而节省开发人员的时间，即使是在使用 IDE 时的输入错误。

在 Dataset 中使用 case 类来定义数据模式的结构。使用 case 类，很容易处理 Dataset。case 类中不同属性的名称直接映射到 Dataset 中的字段名称。它给人一种与 RDD 工作的感觉，但实际上它与 DataFrame 的工作原理是一样的。

DataFrame 实际上被视为通用行对象（row object）的数据集。DataFrame=Dataset[Row]。因此，我们可以通过调用 Dataframe 上的 as 方法，将 DataFrame 转换为 Dataset。例如，df.as[MyClass]。

4.9.2 创建 Dataset

创建 Dataset 的方法有多种：

- 第一种方法是使用 DataFrame 类的 as（符号）函数将 DataFrame 转换为 Dataset；
- 第二种方法是使用 SparkSession.createDataset()函数从本地集合对象中创建 Dataset；
- 第三种方法是使用 toDS 隐式转换程序。

不管使用哪一种方法，通常需要做的第一件事都是定义一个领域特定的对象来表示每一行数据（即需要先定义一个代表数据模式的 case 类）。

下面的代码使用第一种方法，将 DataFrame 转换为 Dataset：

```
// 定义 Movie case class
case class Movie(actor_name:String, movie_title:String, produced_year:Long)

def main(args: Array[String]): Unit = {
  val spark:SparkSession = SparkSession.builder()
    .master("local[*]")
    .appName("Spark Dataset Demo")
    .getOrCreate()
```

```
    // 定义文件路径
    val parquetFile = "src/main/resources/movies.parquet"

    // 读取到 DataFrame 中
    val movies = spark.read.parquet(parquetFile)
    movies.printSchema()

    import spark.implicits._
    // 将 DataFrame 转换到强类型的 Dataset
    val moviesDS = movies.as[Movie]

    // 过滤 2010 生产的电影
    moviesDS.filter(movie => movie.produced_year == 2010).show(5)

    // 显示 moviesDS 中第一部电影的 title
    println(moviesDS.first.movie_title)

    // 试一下：拼写错 movie_title，得到编译时错误
    moviesDS.first.movie_tile

    // 使用 map transformation 执行投影(projection)
    // map：返回一个新的 Dataset，其中包含对每个元素应用' func '的结果。
    val titleYearDS = moviesDS.map(m => ( m.movie_title, m.produced_year))
    titleYearDS.printSchema()
    titleYearDS.show()

    // 演示一个类型安全的 transformation-它在编译时失败,因为在字符串类型的列上执行减法
    // 对于 DataFrame 来说，直到运行时才会检查出问题
    movies.select('movie_title - 'movie_title).show()

    // 在编译时就能检查出问题
    moviesDS.map(m => m.movie_title - m.movie_title).show()

    // take action 返回行作为 Movie 对象给 driver
    moviesDS.take(5).foreach(println)
  }
```

在创建 Dataset 的三种方法中，第一种方法是最流行的方法。

下面的代码演示了使用 SparkSession.createDataset()函数从本地集合对象中创建 Dataset。

```
  // 定义 Movie case class
  case class Movie(actor:String, title:String, year:Long)

  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()
```

```
    // 定义一个本地集合
    val localMovies = Seq(
      Movie("郭涛", "疯狂的石头", 2018L),
      Movie("黄渤", "疯狂的石头", 2018L)
    )

    // 常见类型的编码器是通过导入 spark.implicit._自动提供的
    import spark.implicits._

    // 使用 SparkSession.createDataset()函数从本地集合对象中创建 Dataset
    val localMoviesDS1 = spark.createDataset(localMovies)
    localMoviesDS1.printSchema()
    localMoviesDS1.show()
  }
```

在上面的代码中，case class 添加结构信息。Spark 使用这种结构来创建最好的数据布局和编码。执行以上代码，输出结果如下所示：

下面的代码演示了使用 toDS 隐式转换程序的方法。

```
  // 定义 Movie case class
  case class Movie(actor:String, title:String, year:Long)

  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    // 定义一个本地集合
    val localMovies = Seq(
      Movie("郭涛", "疯狂的石头", 2018L),
      Movie("黄渤", "疯狂的石头", 2018L)
    )

    // 常见类型的编码器是通过导入 spark.`implicit`._自动提供的
    import spark.implicits._

    // toDS 隐式方法创建一个 Dataset
    val localMoviesDS2 = localMovies.toDS()
    localMoviesDS2.printSchema()
    localMoviesDS2.show()
```

执行以上代码，输出结果如下所示：

除了上面这几种方式，还可以通过 toDS()方法将一个 range 或 rdd 转换为 Dataset。请看下面的示例。

```
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
```

```
        .master("local[*]")
        .appName("Spark Basic Example")
        .getOrCreate()

    import spark.implicits._

    // 从一个简单的集合来创建一个 Dataset
    val ds1 = List.range(1,5).toDS()
    ds1.printSchema()
    ds1.show()

    // 从 RDD 到 Dataset 间的转换，使用类型推断
    val phones = List(
      ("小米","中国",3999.00),
      ("华为","中国",4999.00),
      ("苹果","美国",5999.00),
      ("三星","韩国",1999.00),
      ("诺基亚","荷兰",999.00)
    )
    val phones_ds = spark.sparkContext.parallelize(phones).toDS()

    phones_ds.printSchema()
    phones_ds.show()
  }
```

输出结果如下：

```
```

Dataset 提供了查看结构和执行计划的 API。下面的代码向我们展示了如何查看前面所创建的 phones_ds 这个 Dataset 的结构和执行的计划。

```
    // 检查数据类型
    phones_ds.dtypes.foreach(println)

    // 查看 schema
    phones_ds.printSchema

    // 检查执行计划
    phones_ds.explain()
```

执行上面的代码，输出结果如下所示：

```
(_1,StringType)
(_2,StringType)
(_3,DoubleType)

StructType(StructField(_1,StringType,true), StructField(_2,StringType,true), StructField(_3,DoubleType,false))

== Physical Plan ==
*(1) SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, scala.Tuple3, true])._1, true, false) AS _1#11, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, scala.Tuple3, true])._2, true, false) AS _2#12, assertnotnull(input[0, scala.Tuple3, true])._3 AS _3#13]
```

```
+- Scan[obj#10]
```

Dataset 可以从 JSON 文件中创建，类似于 DataFrame。请注意，JSON 文件可能包含几个记录，但是每个记录必须在一行上。如果源 JSON 有换行符，那么必须以编程的方式删除它们。JSON 记录可能有数组，并且可以嵌套。它们不需要有统一的模式。

在下面的示例中，从 JSON 文件创建一个 Dataset，JSON 文件中的每一个记录有一个附加的标签和一个数据数组。

```
  // 定义 Movie case class
  case class Movie(actor_name:String, movie_title:String, produced_year:Long)

  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    // 设置文件路径
    val file = "src/main/data/sql/movies.json"
//    val file = "src/main/resources/movies.json"

    import spark.implicits._
    // 使用 case class 从 JSON 创建 Dataset
    val movies = spark.read.json(file).as[Movie]

    // 查看数据
    movies.printSchema()
    movies.show()
  }
```

执行以上代码，输出结果如下所示：

```
```

4.9.3 操作 Dataset

对 Dataset 的操作同样分为 transformation 转换操作和 action 行为操作，而且非常类似 DataFrame 和 RDD。下面是常用的一些 Dataset transformation 操作：

transformation操作	描述
map	返回将输入函数应用到每个元素之后的新Dataseet
filter	返回一个新Dataset，它包含输入函数为true的元素
groupByKey	返回一个KeyValueGroupedDataset，其数据按给定的key函数分组

下面是一些常用 Dataset action 操作：

action操作	描述
show(n)	显示Dataset中前n行数据
take(n)	以数组形式返回Dataset中前n个对象
count	返回Dataset中行数

Map、flatMap、filter 操作示例

.....

类型化分组- groupByKey 操作符

该方法的签名如下：

```
groupByKey[K: Encoder](func: T => K): KeyValueGroupedDataset[K, T]
```

groupByKey 通过输入函数 func 对记录（类型为 T）进行分组，最后返回一个 KeyValueGroupedDataset，以便对其应用聚合。

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.expressions.scalalang.typed
.....

  case class Product(title: String, quantity: Int, price: Double)

  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    import spark.implicits._

    val products = Seq(
      Product("服装", 2, 2000.00),
      Product("服装", 1, 2500.00),
      Product("玩具", 3, 500.00),
      Product("玩具", 2, 500.00),
      Product("玩具", 4, 1000.00)
    )
    val productsDS = products.toDS().cache()

    productsDS.printSchema()

    // 简单分组统计
    productsDS
      .groupByKey(_.title)
      .count()
      .show()

    // 同时统计多个列
    productsDS
      .groupByKey(_.title)
      .agg(
        typed.sum[Product](_.quantity),     // 总数量
        typed.avg[Product](_.price),        // 均价
        typed.count(_.title)                // 类别数
```

```
        )
        .toDF("商品","总价","均价","总数")
        .orderBy($"value".desc)
        .show()
  }
```

Dataset 中，每一行是什么类型是不一定的，在自定义了 case class 之后可以很自由的获得每一行的信息（可以定义字段名和类型）。可以看出，Dataset 在需要访问列中的某个字段时是非常方便的。

执行以上代码，输出结果如下：

```

mapGroups
.....
```

Dataset API 的限制

尽管 Dataset API 是使用 RDD 和 DataFrame 的最佳方式，但它在当前的开发阶段仍然有一些限制：

- 在查询数据集时，所选字段应该像 case class 一样给定特定的数据类型，否则输出将变成 DataFrame。
- Python 和 R 本质上是动态的，因此不支持类型化 Dataset。

4.9.4 类型安全检查

Dataset 应用在以下几个场景时，有助于更早发现异常和错误：

- 当在 filter 或 map 函数中应用 lambda 表达式时。
- 当查询不存在的列时。
- 如果转换回 RDD(弹性分布式数据集)，DataFrame 和 Dataset 是否保留模式。

【示例】Dataset 在类型安全方面与 DataFrame 进行比较。

```

```

4.9.5 编码器 Encoder

Dataset API 的核心是编码器（encoder）。编码器负责 JVM 对象和表格表示之间的转换，这种表示以 Tungsten 二进制格式存储，绕过了 JVM 的内存管理和垃圾收集，提高了内存利用率。

Spark 有自己的 C 风格的内存访问，专门用于解决它所支持的工作流。由此产生的内部表示占用更少的内存，并具有高效的内存管理。紧凑的内存表示导致在 shuffle 操作期间网络负载减少。编码器生成紧凑的字节码，直接在序列化的对象上操作，如过滤、排序和散列，并提供对单个属性的按需访问，而无需将字节反序列化回对象，从而提高性能。在缓存 Dataset 时，尽早了解模式会导致内存中更优的布局。

Dataset API 中的 Encoders 可以有效地序列化和反序列化 JVM 对象，以生成紧凑的字节码。较小的字节码确保更快的执行速度。

4.10 临时视图与执行 SQL 查询

Spark SQL 支持直接应用标准 SQL 语句进行查询。当在 Spark SQL 中编写 SQL 命令时，它们会被翻译为 DataFrame 上的关系操作。在 SQL 语句内，可以访问所有 SQL 表达式和内置函数。

这需要使用 SparkSession 的 sql 函数执行给定的 SQL 查询，该查询会返回一个 DataFrame。

4.10.1 在 Spark 程序中运行 SQL 语句

Spark 提供了几种在 Spark 中运行 SQL 的方法。

- Spark SQL CLI(. / bin / spark-sql)；
- JDBC / ODBC 服务器；
- 在 Spark 应用程序以编程方式。

前两种选择提供了与 Apache Hive 的集成，以利用 Hive 的元数据。Spark SQL 支持使用基本 SQL 语法或 HiveQL 编写的 SQL 查询的执行。

在 Spark shell 中，spark.sql 是自动导入的，所以可以直接使用该函数用来编写 SQL 命令。例如：

```
sql("select current_date() as today , 1 + 100 as value").show()
```

本节只讨论最后一个选项，即在 Spark 应用程序中以编程方式运行 SQL。下面是执行一个不带注册视图的 SQL 语句的简单示例：

```
  def main(args: Array[String]): Unit = {
    val spark:SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Dataset Demo")
      .getOrCreate()

    val infoDF = spark.sql("select current_date() as today , 1 + 100 as value")
    infoDF.show()
  }
```

输出结果如下所示：

```
+----------+-----+
|     today|value|
+----------+-----+
|2021-07-28|  101|
+----------+-----+
```

从输出结果可以看出，执行 sql 函数返回的是一个 DataFrame。

除了使用 read API 将文件加载到 DataFrame 并对其进行查询，Spark 也可以使用 SQL 直接查询该文件。

```
  def main(args: Array[String]): Unit = {
    val spark:SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Dataset Demo")
      .getOrCreate()

    val sqlDF = spark.sql("SELECT * FROM parquet.`src/main/resources/users.parquet`")
```

```
    sqlDF.show()
  }
```

执行以上代码，输出内容如下：

```
+------+--------------+----------------+
|   name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|    [3, 9, 15, 20]|
|    Ben|           red|              []|
+------+--------------+----------------+
```

4.10.2 注册临时视图并查询

DataFrame 和 Dataset 本质上就像数据库中的表一样，可以通过 SQL 语句来查询它们。不过，在可以发出 SQL 查询来操纵它们之前，需要将它们注册为一个临时视图，然后，就可以使用 SQL 查询从临时表中查询数据了。每个临时视图都有一个名字，通过视图的名字来引用该 DataFrame，该名字在 select 子句中用作表名。请看下面的示例。

【示例】使用 SQL 语句来查询电影数据集。

```
  def main(args: Array[String]): Unit = {
    val spark:SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Dataset Demo")
      .getOrCreate()

    // 定义文件路径
    val parquetFile = "src/main/resources/movies.parquet"

    // 读取到 DataFrame 中
    val movies = spark.read.parquet(parquetFile)

    // 现在将 movies DataFrame 注册为一个临时视图
    movies.createOrReplaceTempView("movies")

    // 从视图 view 查询
    spark.sql("select * from movies where actor_name like '%Jolie%' and produced_year > 2009").show()
  }
```

输出结果如下所示：

```
+--------------+---------------+-------------+
|        actor_name|       movie_title|produced_year|
+--------------+---------------+-------------+
|Jolie, Angelina|           Salt|         2010|
|Jolie, Angelina|Kung Fu Panda 2|         2011|
|Jolie, Angelina|    The Tourist|         2010|
+--------------+---------------+-------------+
```

也可以在 sql 函数中混合使用 SQL 语句和 DataFrame 转换 API。请看下面的示例。

【示例】查询电影数据集，找出参演影片超过 30 部的演员。

```
  def main(args: Array[String]): Unit = {
```

```
    val spark:SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Dataset Demo")
      .getOrCreate()

    // 定义文件路径
    val parquetFile = "src/main/resources/movies.parquet"

    // 读取到 DataFrame 中
    val movies = spark.read.parquet(parquetFile)

    // 现在将 movies DataFrame 注册为一个临时视图
    movies.createOrReplaceTempView("movies")

    // 查询参演影片超过 30 部的演员：
    import spark.implicits._
    spark.sql("select actor_name, count(*) as count from movies group by actor_name")
      .where('count > 30)
      .orderBy('count.desc)
      .show()
  }
```

输出结果如下所示：

```
+------------------+-----+
|        actor_name|count|
+------------------+-----+
|  Tatasciore, Fred|   38|
|      Welker, Frank|   38|
|Jackson, Samuel L.|   32|
|      Harnell, Jess|   31|
+------------------+-----+
```

当 SQL 语句较长时，可以利用"""来格式化多行 SQL 语句。请看下面的示例。

【示例】查询电影数据集，使用子查询来计算每年产生的电影数量。

```
  def main(args: Array[String]): Unit = {
    val spark:SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Dataset Demo")
      .getOrCreate()

    // 定义文件路径
    val parquetFile = "src/main/resources/movies.parquet"

    // 读取到 DataFrame 中
    val movies = spark.read.parquet(parquetFile)

    // 现在将 movies DataFrame 注册为一个临时视图
    movies.createOrReplaceTempView("movies")

    // 使用子查询来计算每年产生的电影数量（利用"""来格式化多行 SQL 语句）
```

```
    import spark.implicits._
    spark.sql("""select produced_year, count(*) as count
                 from (select distinct movie_title, produced_year from movies)
                 group by produced_year
              """)
         .orderBy('count.desc)
         .show(5)
  }
```

输出结果如下所示：

```
```

注：

Spark 实现了 ANSI SQL:2003 修订版（最流行的 RDBMS 服务器支持）的一个子集。此外，Spark 2.0 通过包含一个新的 ANSI SQL 解析器扩展了 Spark SQL 功能，支持子查询和 SQL:2003 标准。更具体地说，子查询支持现在包括相关/不相关的子查询，以及 WHERE / HAVING 子句中的 IN / NOT IN 和 EXISTS / NOT EXISTS 谓词。

4.10.3 使用全局临时视图

Spark 为临时视图提供了两个级别的范围。一个是 Spark 会话级别。当 DataFrame 在这个级别上注册时，只有在同一个会话中发出的查询才能引用那个 DataFrame。当 Spark 会话关闭时，会话范围的级别将消失。第二个作用域级别是全局级别的，这意味着可以在所有 Spark 会话中将这些视图用于 SQL 语句。

Spark SQL 中的临时视图是会话范围的，如果创建它的会话终止，它就会消失。如果希望拥有一个在所有会话之间共享的临时视图，并一直保持活动状态，直到 Spark 应用程序终止，那么可以创建一个全局临时视图。全局临时视图绑定到系统保留的数据库 global_temp，必须使用限定名来引用它，例如“SELECT * FROM global_temp.view1”。

如果要注册全局临时视图，使用 createOrReplaceGlobalTempView 方法。例如，将 movies 注册为全局临时视图，叫做“movies_g”，使用如下的代码：

```
movies.createOrReplaceGlobalTempView("movies_g")
```

从全局视图中查询，需要使用关键字"global_temp"来前缀视图名称。代码如下所示：

```
spark.sql("select count(*) as total from global_temp.movies_g").show
```

请看下面的示例。

【示例】跨多个会话使用全局临时视图进行查询。

```
  def main(args: Array[String]): Unit = {
    val spark:SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Dataset Demo")
      .getOrCreate()

    // 读取 json 数据到 DataFrame
    val df = spark.read.json("src/main/resources/people.json")

    // 简单探索
    df.printSchema()
    df.show()
```

```
    // 注册 DataFrame 为全局临时视图
    df.createGlobalTempView("people")

    // 全局临时视图绑定到一个系统保留的数据库`global_temp`上
    spark.sql("SELECT * FROM global_temp.people").show()

    // 全局临时视图是跨会话的
    spark.newSession().sql("SELECT * FROM global_temp.people").show()
  }
```

执行以上代码，输出结果如下：

```

```

4.10.4 直接从数据源注册表

在前面的示例中，我们都是先将数据加载到 DataFrame 中，然后将 DataFrame 注册为临时表或全局表。除此之外，也可以使用 SparkSession 的 sql()方法从注册的数据源直接加载数据。

例如，在下面的代码中，注册了一个 Parquet 文件并加载它的内容。

```
  def main(args: Array[String]): Unit = {
    val spark:SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName("Spark SQL Demo")
      .getOrCreate()

    // 从 parquet 数据源创建临时视图
    spark.sql("create temporary view usersParquet "+
      "using org.apache.spark.sql.parquet "+
      "options(path 'src/main/resources/users.parquet')")

    // 查询临时视图
    spark.sql("select * from usersParquet").show()
  }
```

执行以上代码，输出结果如下：

```
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

下面是另一个内置数据源的例子：从 jdbc 注册一个临时表，然后使用 sql 语句查询该临时表。（注：这里我们连接的是 MySQL 8 数据库，请自行修改为自己的数据库连接参数）

```
  def main(args: Array[String]): Unit = {
    val spark:SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Dataset Demo")
      .getOrCreate()

    // 数据库连接 URL 太长，所以先放在变量中
```

```
    val DB_URL= "jdbc:mysql://localhost:3306/cdadb" +       // 数据库名为 cdadb
      "?useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true"

    // 从 jdbc 数据源创建临时视图
    spark.sql(
      s"""
        create temporary view moviesjdbc
        using org.apache.spark.sql.jdbc
        options(
          url '${DB_URL}',
          dbtable 'peoples',
          user 'root',
          password 'admin'
        )
      """
    )

    // 在临时视图上执行查询
    spark.sql("select * from moviesjdbc").show()
  }
```

执行上面的代码，输出结果如下：

4.10.5 查看和管理表目录

当将 DataFrame/Dataset 注册为临时视图时，Spark 会将该视图的定义存储在“表目录(table catalog)”中。所有已注册的视图都保存在这个元数据目录中，Spark 提供了一个工具用于管理这个表目录。这个管理工具是作为 Catalog 类实现的，通过 SparkSession 的 catalog 字段访问。

可以使用该 Catalog 对象来查看当前注册的表有哪些（显示 catalog 目录中的表），如果没有的话，是一个空的 list 列表。另外，还可以使用该 Catalog 对象来检查一个指定表的列。

【示例】查看当前注册的表有哪些，

```
  def main(args: Array[String]): Unit = {
    val spark:SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName("Spark SQL Demo")
      .getOrCreate()

    // 从 parquet 数据源创建临时视图
    spark.sql("create temporary view usersParquet "+
      "using org.apache.spark.sql.parquet "+
      "options(path 'src/main/resources/users.parquet')")

    // 查看当前数据库中注册的表有哪些
    spark.catalog.listTables().show()

    // 返回给定表/视图或临时视图的所有列
    spark.catalog.listColumns("usersParquet").show()
  }
```

输出结果如下所示：

要获得所有可用的 SQL 函数列表，执行如下所示代码：

```
// 返回在当前数据库中注册的函数列表
spark.catalog.listFunctions.show()
```

输出结果如下图所示：

还可以使用 cacheTable、uncacheTable、isCached 和 clearCache 方法来管理这些元数据，包括缓存表、删除临时视图和清除缓存等。

4.11 缓存 DataFrame/Dataset

DataFrames 可以在内存中进行持久/缓存，就像 RDD 一样。在 DataFrame 类中也可以使用同样熟悉的持久性 APIs（persist 和 unpersist）。然而，在缓存 DataFrame 时，与 RDD 有很大的区别。Spark SQL 知道 DataFrame 中数据的模式（schema），因此它以一种列格式组织数据，并应用任何适用的压缩来最小化空间使用。最终的结果是，当两个都由同一个数据文件支持时，在内存中存储 DataFrame 所需的空间要比存储 RDD 要少得多。

Spark 缓存和持久化是用于迭代和交互 Spark 应用程序的 DataFrame/Dataset 优化技术，以提高作业的性能。Spark SQL 提供了 cache()和 persist()方法用来缓存 DataFrame/Dataset。

4.11.1 缓存方法

使用 cache()和 persist()方法，Spark 提供了一种优化机制来存储 Spark DataFrame 的中间计算，以便在后续操作中重用。

当持久化一个数据集时，每个节点将它的分区数据存储在自己的内存中，并在该数据集的其他操作中重用它们。Spark 在节点上的持久化数据是容错的，这意味着如果数据集的任何分区丢失，它将自动使用创建它的原始转换重新计算。

【示例】DataFrame 缓存应用示例。

```
def main(args: Array[String]): Unit = {
  val spark:SparkSession = SparkSession.builder()
    .master("local[*]")
    .appName("Spark Cache Demo")
    .getOrCreate()

  // 读取 csv 文件源，创建 DataFrame
  val file = "src/main/resources/movies.csv"
  val df = spark.read
    .options(Map("inferSchema"->"true","delimiter"->",","header"->"true"))
    .csv(file)

  // 缓存
  import spark.implicits._
  val df2 = df.where($"year" === "2012").cache()
```

```
    df2.show(5, truncate = false)

    println(s"2012 年上映的电影数量有：${df2.count()}部")

    val df3 = df2.where($"title" === "The Hunger Games")
    println("电影饥饿游戏的主演是：" + df3.collect()(0)(0))
  }
```

执行以上代码，输出内容如下：

```
```

Spark DataFrame 或 Dataset 的 cache()方法内部调用 persist()方法，默认情况下采用的存储级别是“MEMORY_AND_DISK”，因为重新计算底层表的内存列表示非常昂贵。注意，这与“RDD.cache()”的默认缓存级别“MEMORY_ONLY”不同。

该 persist()方法调用 sparkSession.sharedState.cacheManager.cacheQuery()缓存 DataFrame 或 Dataset 的结果集。不管是 cache()还是 persist()方法，都是延迟计算的(只有在执行 action 时才进行计算)。实际上，cache()是 persist(StorageLevel.MEMORY_AND_DISK)的别名。

4.11.2 缓存策略

Spark 支持的所有不同存储级别都可以在 org.apache.spark.storage.StorageLevel 类上获得。存储级别指定如何以及在何处持久化或缓存 Spark DataFrame 和 Dataset。

- MEMORY_ONLY – 这是 RDD cache()方法的默认行为，并将 DataFrame 作为反序列化对象存储到 JVM 内存中。当没有足够的可用内存时，它将不保存某些分区的 DataFrame，并在需要时重新计算这些 DataFrame。这需要更多的内存。但与 RDD 不同的是，这将比 MEMORY_AND_DISK 级别慢，因为它重新计算未保存的分区，并且重新计算底层表的内存列表示非常昂贵。
- MEMORY_ONLY_SER – 这与 MEMORY_ONLY 相同，但不同之处在于它将 RDD/DataFrame 作为序列化对象存储到 JVM 内存中。 它比 MEMORY_ONLY 占用更少的内存(节省空间)，因为它将对象保存为序列化的，并且为了反序列化多占用几个 CPU 周期。
- MEMORY_ONLY_2 – 与 MEMORY_ONLY 存储级别相同，但将每个分区复制到两个集群节点。
- MEMORY_ONLY_SER_2 –与 MEMORY_ONLY_SER 存储级别相同，但将每个分区复制到两个集群节点。
- MEMORY_AND_DISK –这是 DataFrame 或 Dataset 的**默认**行为。在这个存储级别中，数据帧将作为反序列化对象存储在 JVM 内存中。 当所需的存储大于可用内存时，它将一些多余的分区存储到磁盘中，并在需要时从磁盘读取数据。当涉及到 I/O 时，它会较慢。
- MEMORY_AND_DISK_SER – 这与 MEMORY_AND_DISK 存储级别相同，不同之处在于，当空间不可用时，它会在内存和磁盘上序列化数据帧对象。
- MEMORY_AND_DISK_2 –与 MEMORY_AND_DISK 存储级别相同，但将每个分区复制到两个集群节点。
- MEMORY_AND_DISK_SER_2 – 与 MEMORY_AND_DISK_SER 存储级别相同，但将每个分区复制到两个集群节点。
- DISK_ONLY –在这个存储级别中，DataFrame 仅存储在磁盘上，CPU 计算时间长，因为涉及

到 I/O。

- DISK_ONLY_2 – 与 DISK_ONLY 存储级别相同，但将每个分区复制到两个集群节点。

Spark 会自动监控每个 persist()和 cache()调用，并检查每个节点上的使用情况，如果没有使用或使用最近最少使用(LRU)算法，则删除持久化数据。也可以使用 unpersist()方法手动删除。unpersist()将 Dataset/DataFrame 标记为非持久的，并从内存和磁盘中删除它的所有块。例如：

```
val dfPersist = dfPersist.unpersist()
```

4.11.3 缓存表

也可以使用 Spark 的 catalog 来缓存 DataFrame，以可读的名字持久化表。在下面的示例中，演示了如何持久化一个 DataFrame 到表中：

```
// 以人类可读的名字持久化一个 DataFrame

val numDF = spark.range(1000).toDF("id")

// 注册为一个视图
numDF.createOrReplaceTempView("num_df")

// 使用 Spark catalog 来缓存该 numDF，使用名字"num_df"
spark.catalog.cacheTable("num_df")

// 通过 count action 操作来强制持久化
numDF.count
```

cache()和 persist()持久化是 lazy 执行的，而 cacheTable()是 eager 执行的。

4.12 Spark SQL 编程案例

本节通过几个案例的学习，掌握使用 Spark SQL DSL API 和 SQL 进行大数据分析的方法。

4.12.1 实现单词计数

到目前为止，我们已经了解到，在 Spark 中，对数据的处理，有三种方案：

- 使用 RDD;
- 使用 DataFrame/Dataset (DSL API)
- 使用 DataFrame/Dataset (SQL 语句)

强烈建议不要直接使用 RDD，而是使用 DataFrame/Dataset 来对数据进行分析计算。只有这样，才能充分利用 Spark SQL 的 Catalyst 优化器来对分析过程自动进行优化。

下面这个示例中，我们使用 DataFrame 和 SQL 两种方式来实现单词计数功能。

【例】使用 Spark Dataset API 统计某个英文文本中的词频。

实现过程和代码如下所示。

1) 准备数据文件。请自行创建一个纯文本文件 words.txt，并编辑内容如下：

```
good good study
day day up
```

```
to be or not to be
this is a question
```

2) 方法一：使用 DSL API 实现单词计数。

```
def main(args: Array[String]): Unit = {
  // 0) 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark Basic Example")
    .getOrCreate()

  // 定义文件路径
  val filePath = "src/main/resources/words.txt"
  val wordDS = spark.read.textFile(filePath)

  //      wordDS.printSchema()
  //      wordDS.show()

  // implicit object 提供了隐式转换，用于将 Scala 对象(包括 rdd)转换为 Dataset、DataFrame、Columns
  // implicit object 是在 SparkSession 内部定义的
  // implicit object 继承了 SQLImplicits 抽象类
  import spark.implicits._

  // 对 Dataset 进行一系列处理，产生一个包含最终结果的 Dataset
  val wordDF = wordDS
    .flatMap(_.split("\\s+"))
    .filter(_.size>0)
    .groupByKey(_.toLowerCase)
    .count
    .toDF("word","count")

  wordDF.show()

  // 获得前 3 个出现频率最高的词
  val top3 = wordDF.orderBy($"count".desc).limit(3)

  // 输出结果
  top3.show()
}
```

执行以上代码，输出结果如下所示：

```
```

3) 方法二：使用 SQL 语句。

```
def main(args: Array[String]): Unit = {
  // 0) 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark Basic Example")
    .getOrCreate()

  // 读取输入文件
```

```
    val filePath = "src/main/resources/words.txt"
    val wordDS2 = spark.read.textFile(filePath)

    import spark.implicits._

    // 转换操作，然后返回加了列标题的 DataFrame
    val wordDF2 = wordDS2
      .flatMap(_.split("\\s+"))
      .filter(_.size>0)
      .toDF("word")

    // 注册为临时 view
    wordDF2.createOrReplaceTempView("wc_tb")

    // 执行 SQL 查询，分析产生结果
    val sql = """
            select word,count(1) as count
            from wc_tb
            group by word
            order by count desc
        """
    val resultDF = spark.sql(sql)
    resultDF.show()
  }
```

执行以上代码，输出结果如下所示：

4.12.2 用户数据集分析

Spark SQL 支持两种不同的方法将现有的 RDD 转换为 DataFrame/Dataset。

- 第一种方法使用反射推断模式。这种方法使代码更简洁，当在编写 Spark 应用程序时已经了解模式时，这种方法可以很好地工作。
- 第二种方法是通过编程接口自己构造模式，然后将其应用于现有的 RDD。虽然此方法更冗长，但它允许我们在列及其类型直到运行时才知道时构造数据集。

1、使用反射推断模式创建 RDD

Spark SQL 的 Scala 接口支持将包含 case 类的 RDD 自动转换为 DataFrame。case 类定义了表的模式。case 类的参数名使用反射读取，并成为列的名称。case 类还可以嵌套或包含复杂类型，如 Seq 或 Array。可以隐式地将此 RDD 转换为 DataFrame，然后将其注册为表。表可以在后续的 SQL 语句中使用。

【示例】分析 Spark 安装包自带的 people.txt 文件内容。

```
package com.xueai8.sql

import org.apache.spark.sql.SparkSession

object Example02 {

  // 定义 case 类
```

```
  case class Person(name:String, age:Int)

  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    // 用于从 RDD 到 DataFrame 的隐式转换
    import spark.implicits._    // 导入隐式类， rdd->DataFrame 需要

    // 使用反射推断模式创建 RDD
    // 从一个文本文件创建 RDD[Person]，并将它转换为一个 DataFrame
    val file = "input/resources/people.txt"    // 定义文件路径
    val peopleDF = spark.sparkContext
      .textFile(file)                // RDD[String]
      .map(_.split(","))             // RDD[Array[String]]
      .map(atts => Person(atts(0), atts(1).trim.toInt))   // RDD[Person]
      .toDF()                        // DataFrame

    // 将该 DataFrame 注册为一个临时视图
    peopleDF.createOrReplaceTempView("people")

    // 使用 SparkSession 的 sql 方法来运行 SQL 语句
    val sqlStr = "SELECT name, age FROM people WHERE age BETWEEN 13 AND 19"
    val teenagersDF = spark.sql(sqlStr)

    // 结果集中每行的列可以通过字段索引访问
    teenagersDF.map(teenager => "Name: " + teenager(0)).show()

    // 或者也可以通过字段名来访问
    teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show()

    // Dataset[Map[K,V]]没有预定义的 encoders，明确定义
    implicit val mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]]

    // 基本类型和 case 类也可以定义为
    // implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()

    // row.getValuesMap[T]一次检索多个列到一个 Map[String, T]
    val result = teenagersDF.map(teenager => teenager.getValuesMap[Any](List("name", "age"))).collect()
    // Array(Map("name" -> "Justin", "age" -> 19))

    result.foreach(m => println(m))
    result.foreach(m => println(m.mkString("\n")))
  }
}
```

执行上面的代码，输出内容如下：

2、以编程方式指定模式

当不能提前定义 case 类时(例如，记录的结构编码在一个字符串中，或者文本数据集将被解析，字段将针对不同的用户以不同的方式投影)，可以通过三个步骤以编程方式创建 DataFrame。

- 从原始 RDD 创建一个 Row RDD;
- 创建由 StructType 表示的模式，该模式与上一步中创建的 RDD 中的 Row 结构相匹配。
- 通过 SparkSession 提供的 createDataFrame 方法将模式应用到 Row RDD。

例如：

```
package com.snail.sql

import org.apache.spark.sql._
import org.apache.spark.sql.types._

object Example03 {

  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    // 定义文件路径
    val file = "input/resources/people.txt"

    // 创建一个 RDD
    val peopleRDD = spark.sparkContext.textFile(file)

    val fields = Array(
      StructField("name",StringType,nullable = true),
      StructField("age",IntegerType,nullable = true)
    )
    val schema = StructType(fields)

    // 将 RDD 的记录转换为 Rows
    val rowRDD = peopleRDD
      .map(_.split(","))
      .map(atts => Row(atts(0), atts(1).trim.toInt))

    // 将这个 schema 应用到该 RDD
    val peopleDF = spark.createDataFrame(rowRDD, schema)

    // 使用该 DataFrame 创建一个临时视图
    peopleDF.createOrReplaceTempView("people")

    // SQL 可以在使用 DataFrames 创建的临时视图上运行
    val sqlStr = "SELECT name, age FROM people WHERE age BETWEEN 13 AND 19"
    val results = spark.sql(sqlStr)
```

```
    import spark.implicits._    // 注意，需要导入隐式类

    // SQL 查询的结果是 DataFrame，支持所有正常的 RDD 操作
    // 可以通过字段索引或字段名访问结果中一行的列
    results.map(atts => "Name: " + atts(0)).show()
    results.map(atts => "Name: " + atts.getAs[String]("name")).show()
  }
}
```

执行以上代码，输出内容如下：

4.12.4 电商用户评论数据集分析

本案例的数据集包含电子产品类别的大约 169 万条 Amazon 评论。我们可以直接读取 JSON 数据集来创建 Spark SQLDataFrame。

【例】从 JSON 文件中读取一组订单记录进行分析。实现代码如下：

```
  def main(args: Array[String]): Unit = {

    import org.apache.spark.sql.SparkSession

    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()
    val sc = spark.sparkContext

    // 数据集路径
    val filePath = "src/main/resources/amazon/Electronics_5.json"

    // 加载数据文件，创建 DataFrame
    val reviewsDF = spark.read.json(filePath).cache()

    // 简单查看
    reviewsDF.printSchema()
//    reviewsDF.show(5)

    // 创建临时视图
    reviewsDF.createOrReplaceTempView("reviewsTable")

    // 执行 SQL 查询，找出给出综合评分(overall)大于 3 的评论
    val sqlStr =
      """
        SELECT asin, overall, reviewTime, reviewerID, reviewerName
        FROM reviewsTable
        WHERE overall >= 3
```

```
      """
    val selectedDF = spark.sql(sqlStr)
    selectedDF.show(5)

    import spark.implicits._
    val selectedJSONArrayElementDF = reviewsDF
      .select($"asin", $"overall", $"helpful")
      .where($"helpful".getItem(0) < 3)
    selectedJSONArrayElementDF.show(5)

  }
```

执行以上代码，输出结果如下：

4.12.5 航空公司航班数据集分析

我们将使用美国交通部的一些航班信息，探索最导致航班延误的航班属性。使用 Spark Dataset，我们将探索这些航班数据来回答以下问题：当航班延误超过 40 分钟时，

- 哪家航空公司的航班延误次数最多?
- 每周哪几天的航班延误次数最多?
- 哪些始发机场的航班延误次数最多?
- 每天什么时候的航班延误次数最多?

航班数据是 JSON 文件，每个航班记录有以下信息：

属性	含义
id	ID，由由承运人、日期、出发地、目的地、航班号组成
dofW	星期几（1 = Monday星期一，7 = Sunday星期日）
carrier	承运人代码
origin	起始机场代码
dest	目的地机场代码
crsdephour	规定起飞时间hour（scheduled departure hour ）
crsdeptime	规定起飞时间time（scheduled departure time）
depdelay	起飞延误分钟数（departure delay in minutes）
crsarrtime	预定到达时间（scheduled arrival time）
arrdelay	到达延误分钟数（arrival delay minutes）
crselapsedtime	飞行时间
dist	距离（distance）

每条航班信息的格式如下：

```
{
"_id": "AA_2017-01-01_ATL_LGA_1678",
"dofW": 7,
"carrier": "AA",
"origin": "ATL",
"dest": "LGA",
"crsdephour": 17,
"crsdeptime": 1700,
"depdelay": 0.0,
"crsarrtime": 1912,
```

```
"arrdelay": 0.0,
"crselapsedtime": 132.0,
"dist": 762.0
}
```

代码实现如下。

```
```

第 5 章　Spark SQL（高级）

为了帮助执行复杂的分析，Spark SQL 提供了一组强大而灵活的聚合函数、连接多个数据集的函数、一组内置的高性能函数和一组高级分析函数。本章将详细介绍这些主题。另外本章还介绍了 Spark SQL 模块的一些高级功能，并解释 Catalyst 优化器和 Tungsten 引擎所提供的优化和执行效率。

5.1 内置标量函数

DataFrame API 的设计目的是在数据集中操作或转换单个行，如过滤或分组。如果我们想要转换一个数据集中的每一行的列的值，例如将字符串从大写字母转换成驼峰命名形式，那么就需要使用一个函数来实现。Spark SQL 内置了一组常用的函数，同时也提供了用户自定义新函数的简单方法。

为了有效地使用 Spark SQL 执行分布式数据操作，必须熟练使用 Spark SQL 函数。Spark SQL 提供了超过 200 个内置函数，它们被分组到不同的类别中。

按类别来分，SQL 函数可分为四类：

- 标量函数：每一行返回单个的值。
- 聚合函数：每一组行返回单个的值。
- 窗口函数：每一组行返回多个值。
- 用户自定义函数（UDF）：包括自定义的标量函数或聚合函数。

标量函数和聚合函数位于 org.apache.spark.sql.functions 包内。在使用前，需要先导入它：

```
import org.apache.spark.sql.functions._
```

如果是使用 spark-shell 或 zeppelin 进行交互式分析，则会自动导入该包。

Spark 提供了大量的标量函数，主要完成：

- 数学计算：abs、hypot、log、cbrt，等等。
- 字符串操作：length、trim、concat，等等。
- 日期操作：year、date_add，等等。

下面我们详细来了解这些函数及其用法。

5.1.1 日期时间函数

Spark 内置的日期时间函数大致可分为以下三个类别：

- 执行日期时间格式转换的函数；
- 执行日期时间计算的函数；
- 从日期时间戳中提取特定值（如年、月、日等）的函数。

日期和时间转换函数有助于将字符串转换为日期、时间戳或 Unix 时间戳，反之亦然。在内部，它使用 Java 日期格式模式语法。这些函数使用的默认的日期格式是“yyyy-mm-dd HH:mm:ss”。因此，如果日期或时间戳列的日期格式不同，那么需要向这些转换函数传入指定的模式。

下面的示例显示了将字符串类型的日期和时间戳转换为 Spark date 和 timestamp 类型。

```
def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
```

```
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    import spark.implicits._

    // 1）日期和时间转换函数：这些函数使用的默认的日期格式是 yyyy-mm-dd HH:mm:ss
    // 构造一个简单的 DataFrame，注意最后两列不遵循默认日期格式
    val testDate = Seq((1, "2019-01-01", "2019-01-01 15:04:58", "01-01-2019", "12-05-2018 45:50"))
    val testDateTSDF = testDate.toDF("id", "date", "timestamp", "date_str", "ts_str")

    // 将这些字符串转换为 date、timestamp 和 unix timestamp，并指定一个自定义的 date 和 timestamp 格式
    val testDateResultDF = testDateTSDF.select(
      to_date('date).as("date1"),
      to_timestamp('timestamp).as("ts1"),
      to_date('date_str,"MM-dd-yyyy").as("date2"),
      to_timestamp('ts_str,"MM-dd-yyyy mm:ss").as("ts2"),
      unix_timestamp('timestamp).as("unix_ts")
    )
    testDateResultDF.printSchema          // date1 和 ts1 分别为日期和时间戳类型
    testDateResultDF.show(false)
  }
```

执行以上代码，输出结果如下：

将日期或时间戳转换为时间字符串是很容易的，方法是使用 date_format 函数和定制日期格式，或者使用 from_unixtime 函数将 Unix 时间戳（以秒为单位）转换成字符串。请看下面这个转换例子：

```
    testDateResultDF.select(
      date_format('date1, "dd-MM-YYYY").as("date_str"),
      date_format('ts1, "dd-MM-YYYY HH:mm:ss").as("ts_str"),
      from_unixtime('unix_ts,"dd-MM-YYYY HH:mm:ss").as("unix_ts_str")
    ).show()
```

执行以上代码，输出结果如下所示：

```
+----------+-------------------+-------------------+
|  date_str|             ts_str|        unix_ts_str|
+----------+-------------------+-------------------+
|01-01-2019|01-01-2019 15:04:58|01-01-2019 15:04:58|
+----------+-------------------+-------------------+
```

日期-时间计算函数有助于计算两个日期或时间戳的相隔时间，以及执行日期或时间算术运算。下面这个例子演示了日期-时间计算：

```
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    import spark.implicits._
```

```
    // 2) 日期-时间（date-time）计算函数
    val employeeData = Seq(
      ("黄渤", "2016-01-01", "2017-10-15"),
      ("王宝强", "2017-02-06", "2017-12-25")
    ).toDF("name", "join_date", "leave_date")

    employeeData.show()

    // 执行 date 和 month 计算
    employeeData.select(
      'name,
      datediff('leave_date, 'join_date).as("days"),
      months_between('leave_date, 'join_date).as("months"),
      last_day('leave_date).as("last_day_of_mon")
    ).show()

    // 执行日期加、减计算
    val oneDate = Seq("2019-01-01").toDF("new_year")
    oneDate.select(
      date_add('new_year, 14).as("mid_month"),
      date_sub('new_year, 1).as("new_year_eve"),
      next_day('new_year, "Mon").as("next_mon")
    ).show()
  }
```

执行上面的代码，输出结果如下所示：

转换不规范的日期：

```
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    import spark.implicits._

    // 转换不规范的日期：
    val df = Seq(
      "Nov 05, 2018 02:46:47 AM",
      "Nov 5, 2018 02:46:47 PM"
    ).toDF("times")

    df.withColumn(
      "times2",
      from_unixtime(
        unix_timestamp($"times", "MMM d, yyyy hh:mm:ss a"),
        "yyyy-MM-dd HH:mm:ss.SSSSSS"
```

```
        )
    ).show(false)
  }
```

执行以上代码，输出结果如下：

```
+----------------------+--------------------------+
|times                 |times2                    |
+----------------------+--------------------------+
|Nov 05, 2018 02:46:47 AM|2018-11-05 02:46:47.000000|
|Nov 5, 2018 02:46:47 PM |2018-11-05 14:46:47.000000|
+----------------------+--------------------------+
```

在处理时间序列数据（time-series data）时，经常需要提取日期或时间戳值的特定字段（如年、月、小时、分钟和秒）。例如，当需要按季度、月或周对所有股票交易进行分组时，就可以从交易日期提取该信息，并按这些值分组。下面的代码展示了如何从日期或时间戳中提取字段。

```
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    import spark.implicits._

    // 3）提取日期或时间戳值的特定字段（如年、月、小时、分钟和秒）
    // 从一个日期值中提取指定的字段
    val valentimeDateDF = Seq("2019-02-14 13:14:52").toDF("date")
    valentimeDateDF.select(
      year('date).as("year"),                  // 年
      quarter('date).as("quarter"),            // 季
      month('date).as("month"),                // 月
      weekofyear('date).as("woy"),             // 周
      dayofmonth('date).as("dom"),             // 日
      dayofyear('date).as("doy"),              // 天
      hour('date).as("hour"),                  // 小时
      minute('date).as("minute"),              // 分
      second('date).as("second")               // 秒
    ).show()
  }
```

执行以上代码，输出结果如下：

```
+----+-------+-----+---+---+---+----+------+------+
|year|quarter|month|woy|dom|doy|hour|minute|second|
+----+-------+-----+---+---+---+----+------+------+
|2019|      1|    2|  7| 14| 45|  13|    14|    52|
+----+-------+-----+---+---+---+----+------+------+
```

5.1.2 字符串函数

毫无疑问，大多数数据集的大多数列都是 string 类型的。Spark SQL 内置的字符串函数提供了操作

字符串类型列的通用和强大的方法。一般来说，这些函数分为两类。

- 执行字符串转换的函数；
- 执行字符串提取（或替换）的函数，使用正则表达式。

最常见的字符串转换包括去空格、填充、大写转换、小写转换和字符串连接等。下面的代码展示了使用各种内置字符串函数转换字符串的各种方法。

```
def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark Basic Example")
    .getOrCreate()

  import spark.implicits._

  // 使用各种内置字符串函数转换字符串的各种方法。
  val sparkDF = Seq(" Spark ").toDF("name")

  // 去空格
  sparkDF.select(
    trim($"name").as("trim"),          // 去掉"name"列两侧的空格
    ltrim($"name").as("ltrim"),        // 去掉"name"列左侧的空格
    rtrim($"name").as("rtrim")         // 去掉"name"列右侧的空格
  ).show()
}
```

执行上面的代码，输出结果如下所示:

```
+-----+------+------+
| trim| ltrim| rtrim|
+-----+------+------+
|Spark|Spark | Spark|
+-----+------+------+
```

用给定的 pad 字符串将字符串填充到指定长度：

```
def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark Basic Example")
    .getOrCreate()

  import spark.implicits._

  // 使用各种内置字符串函数转换字符串的各种方法。
  val sparkDF = Seq(" Spark ").toDF("name")

  // 首先去掉"Spark"前后的空格，然后填充到 8 个字符长
  sparkDF
    .select(trim($"name").as("trim"))          // 去掉两侧的空格
    .select(
```

```
        lpad($"trim", 8, "-").as("lpad"),                    // 宽度为 8，不够的话，左侧填充"-"
        rpad($"trim", 8, "=").as("rpad")                     // 宽度为 8，不够的话，右侧填充"="
      )
      .show()
  }
```

执行以上代码，输出结果如下所示：

下面的代码中演示了多个字符串转换函数的用法。

```
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    import spark.implicits._

    // 使用 concatenation, uppercase, lowercase 和 reverse 转换一个字符串
    val sentenceDF = Seq(("Spark", "is", "excellent")).toDF("subject", "verb", "adj")
    sentenceDF
      .select(concat_ws(" ",$"subject", $"verb", $"adj").as("sentence"))   // 用空格连接多列值
      .select(
        lower($"sentence").as("lower"),                      // 转小写
        upper($"sentence").as("upper"),                      // 转大写
        initcap($"sentence").as("initcap"),                  // 转首字母大写
        reverse($"sentence").as("reverse")                   // 反转
      )
      .show()

    // 从一个字符转换到另一个字符
    sentenceDF
      .select($"subject", translate($"subject", "pr", "oc").as("translate"))
      .show()
  }
```

执行以上代码，输出结果如下所示：

如果想从字符串列值中替换或提取子字符串，可以使用 regexp_extract 和 regexp_replace 函数。这两个函数通过传入的正则表达式模式来实现替换或提取。Spark 利用 Java 正则表达式库来实现这两个字符串函数的底层实现。

如果要提取字符串的某一部分子字符串，使用 regexp_extract 函数。regexp_extract 函数的输入参数是字符串列、匹配的模式和组索引。在字符串中可能会有多个匹配模式;因此，需要组索引（从 0 开始）来确定是哪一个。如果没有指定模式的匹配，则该函数返回空字符串。以获得 regexp_extract 函数的一个示例。

```
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
```

```
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark function Example")
      .getOrCreate()

    import spark.implicits._

    // 使用 regexp_extract 字符串函数来提取"fox"，使用一个模式
    val strDF = Seq("A fox saw a crow sitting on a tree singing \"Caw! Caw! Caw!\"").toDF("comment")

    // 使用一个模式
    strDF
      .select(regexp_extract($"comment", "[a-z]*o[xw]",0).as("substring"))
      .show()
  }
```

执行以上代码，输出结果如下所示：

```
+---------+
|substring|
+---------+
|      fox|
+---------+
```

如果要提取字符串的某一部分子字符串，使用 regexp_replace 函数。regexp_replace 字符串函数的输入参数是字符串列、匹配的模式、以及替换的值。下面是一个 regexp_replace 函数的示例。

```
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark function Example")
      .getOrCreate()

    import spark.implicits._

    // 用 regexp_replace 字符串函数将“fox”和“Caw”替换为“animal”
    val strDF = Seq("A fox saw a crow sitting on a tree singing \"Caw! Caw! Caw!\"").toDF("comment")

    // 下面两行产生相同的输出
    strDF
      .select(regexp_replace($"comment", "fox|crow", "animal").as("new_comment"))
      .show(false)

    strDF
      .select(regexp_replace($"comment", "[a-z]*o[xw]", "animal").as("new_comment"))
      .show(false)
  }
```

执行以上代码，输出结果如下所示：

在下面这个示例中，我们使用 regexp_replace()函数从从混乱的数据中抽取出手机号。

```
def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark function Example")
    .getOrCreate()

  import spark.implicits._

  val telDF = Seq("135a-123b4-c5678").toDF("tel")
  telDF.withColumn("phone",regexp_replace('tel,"-|\\D","")).show()
}
```

执行以上代码，输出结果如下所示：

```
+---------------+-----------+
|            tel|      phone|
+---------------+-----------+
|135a-123b4-c5678|13512345678|
+---------------+-----------+
```

5.1.3 数学计算函数

Spark SQL 还提供有许多对数值类型列进行计算的函数，其中最经常用到的是 round 函数，它对传入的列值执行一个四舍五入计算。这个函数有两种方法签名：

- def round(e: Column, scale: Int): Column。
- def round(e: Column): Column

下面的示例代码演示了 round 函数的行为。

```
def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark function Example")
    .getOrCreate()

  import spark.implicits._

  val numberDF = Seq((3.14159, -3.14159)).toDF("pie", "-pie")
  numberDF
    .select(
      $"pie",
      round($"pie").as("pie0"),                    // 整数四舍五入
      round($"pie", 2).as("pie1"),                 // 四舍五入，保留小数点后 2 位
      round($"pie", 4).as("pie2"),                 // 四舍五入，保留小数点后 4 位
      $"-pie",
      round($"-pie").as("-pie0"),                  // 整数四舍五入
      round($"-pie", 2).as("-pie1"),               // 四舍五入，保留小数点后 2 位
      round($"-pie", 4).as("-pie2")                // 四舍五入，保留小数点后 4 位
    )
    .show()
```

```
}
```

执行以上代码，输出结果如下所示：

```

```

还有两个数学函数也经常用到，分别是 ceil 和 floor：

- def ceil(e: Column): Column
- def floor(e: Column): Column

请看下面的示例。

```
def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark function Example")
    .getOrCreate()

  import spark.implicits._

  val numberDF = Seq((3.14159, -3.14159)).toDF("v1", "v2")
  numberDF
    .select(
      $"v1",
      ceil($"v1"),                    // 向上取整
      floor($"v1"),                   // 向下取整
      $"v2",
      ceil($"v2"),                    // 向上取整
      floor($"v2")                    // 向下取整
    )
    .show()
}
```

执行以上代码，输出结果如下所示：

```

```

5.1.4 处理集合元素的函数

集合被设计用来处理复杂的数据类型，如 arrays、maps 和 structs。本节将介绍两种特定类型的集合函数。第一种方法是使用 array 数据类型，第二种方法是处理为 JSON 数据格式。

Spark DataFrame 支持复杂数据类型，也就是列值可以是一个集合。我们可以使用数组相关的集合函数来轻松获取数组大小、检查值的存在、或者对数组进行排序。下面的代码包含了处理各种数组相关函数的示例。

```

```

执行以上代码，输出结果如下所示：

```

```

许多非结构化数据集都是以 JSON 的形式存在的。对于 JSON 数据类型的列，使用相关的集合函数将 JSON 字符串转换成 struct（结构体）数据类型。主要的函数是 from_json、get_json_object 和 to_json。一旦 JSON 字符串被转换为 Spark struct 数据类型，我们就可以轻松地提取这些值。下面的代码演示了

from_json 和 to_json 函数的示例。

执行以上代码，输出结果如下所示：

5.1.5 其他函数

除了前面几节介绍的函数外，还有一些函数在特定的场景下非常有用。节将介绍以下几个这样的函数：monotonically_increasing_id、when、coalesce 和 lit。

monotonically_increasing_id 函数

有时需要为数据集中的每一行生成单调递增的惟一但不一定是连续的 id。例如，如果一个 Dataset 有 2 亿行，并且是分区存储的，那么如何确保这些 id 值是惟一的并且同时增加呢？Spark SQL 提供了一个 monotonically_increasing_id 函数，它生成 64 位整数作为 id 值。下面是使用 monotonically_increasing_id 函数的例子。

```
def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark function Example")
    .getOrCreate()

  import spark.implicits._

  // 首先生成一个 DataFrame，它的值分散到 5 个分区中
  val numDF = spark.range(1,11,1,5)

  // 验证的确有 5 个分区
  println("分区数为：" + numDF.rdd.getNumPartitions)

  // 现在生成单调递增的值，并查看所在的分区
  import org.apache.spark.sql.functions._
  numDF.select(
    $"id",
    monotonically_increasing_id().as("m_ii"),
    spark_partition_id().as("partition")
  ).show()
}
```

执行上面的代码，输出结果如下所示：

when 函数

在 DataFrame 中，如果需要根据条件列表来评估一个值并返回一个值，可以使用 when 函数。在下面的示例代码中，使用 when 函数将数字值转换成字符串。

```
def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
  val spark = SparkSession.builder()
```

```
      .master("local[*]")
      .appName("Spark function Example")
      .getOrCreate()

    import spark.implicits._

    // 创建一个具有从 1 到 7 的值的 DataFrame 来表示一周中的每一天
    val dayOfWeekDF = spark.range(1,8)

    // 将每个数值转换成字符串
    import org.apache.spark.sql.functions._
    dayOfWeekDF.select(
      $"id",
      when($"id" === 1, "星期一")
        .when($"id" === 2, "星期二")
        .when($"id" === 3, "星期三")
        .when($"id" === 4, "星期四")
        .when($"id" === 5, "星期五")
        .when($"id" === 6, "星期六")
        .when($"id" === 7, "星期七")
        .as("星期")
    ).show()
  }
```

执行以上代码，输出结果如下所示：

处理默认情况时，可以使用 Column 类的 otherwise 函数。

```
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark function Example")
      .getOrCreate()

    import spark.implicits._

    // 创建一个具有从 1 到 7 的值的 DataFrame 来表示一周中的每一天
    val dayOfWeekDF = spark.range(1,8)

    // 将每个数值转换成字符串
    import org.apache.spark.sql.functions._
    dayOfWeekDF.select(
      $"id",
      when($"id" === 6, "周末")
        .when($"id" === 7, "周末")
        .otherwise("工作日")
        .as("day_type")
    ).show()
  }
```

执行以上代码，输出结果如下所示：

coalesce 函数和 lit 函数

在处理数据时，正确处理 null 值是很重要的。Spark 提供了一个名为 coalesce 的函数，该函数接受一个或多个列值，并返回第一个非空值。而 coalesce 函数中的每个参数都必须是 Column 类型，所以如果想传入字面量值，那么需要使用 lit 函数，将字面量值包装为 Column 类的实例。下面的代码中演示了 coalesce 和 lit 函数的使用。

```
// 创建一个 case class
case class Movie(actor_name:String, movie_title:String, produced_year:Long)

def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark function Example")
    .getOrCreate()

  // 使用 coalesce 来处理列中的 Null 值
  import spark.implicits._
  import org.apache.spark.sql.functions._

  // 构造一个 DataFrame，带有 null 值
  val badMoviesDF = Seq(
    Movie(null, null, 2018L),
    Movie("黄渤", "一出好戏", 2018L)
  ).toDF()

  badMoviesDF.show()

  // 使用 coalese 来处理 title 列中的 null 值
  badMoviesDF
    .select(
      coalesce($"actor_name", lit("路人甲")).as("演员"),
      coalesce($"movie_title", lit("烂片")).as("电影"),
      coalesce($"produced_year", lit("烂片")).as("年份")
    )
    .show()
}
```

执行以上代码，输出结果如下所示：

5.1.6 函数应用示例

这一节，通过几个示例程序来演示 Spark SQL 函数的应用。

【示例】使用 Spark DataFrame 实现二次排序。

假设我们有以下输入文件 data.txt，其中逗号分割的分别是年、月和总数：

```
2018,5,22
2019,1,24
2018,2,128
2019,3,56
2019,1,3
2019,2,-43
2019,4,5
2019,3,46
2018,2,64
2019,1,4
2019,1,21
2019,2,35
2019,2,0
```

我们想要对这些数据排序，期望的输出结果如下：

```
2018-2   64,128
2018-5   22
2019-1   3,4,21,24
2019-2   -43,0,35
2019-3   46,56
2019-4   5
```

实现过程

1、加载数据集。

```
// 加载数据集
val inputPath = "file:///home/hduser/data/spark_demo/data.txt"
val inputDF = spark.read
                    .option("inferSchema","true")
                    .option("header","false")
                    .csv(inputPath)
                    .toDF("year","month","cnt")

inputDF.show()
```

查看结果：

2、 组合 year 和 month 为一列，并取别名“ym”。

```
import org.apache.spark.sql.functions._

import spark.implicits
val df2 = inputDF.select(concat_ws("-",$"year",$"month").as("ym"), $"cnt")
df2.printSchema()
df2.show
```

查看结果：

3、先按“ym”进行分组聚合，然后对每一组的“cnt”列进行排序，并输出。

```
df2.groupBy("ym")
    .agg(sort_array(collect_list("cnt")).as("cnt"))
    .orderBy("ym")
    .show
```

查看结果：

```
```

4、最后，我们可以把上面的代码写到一个 ETL 当中。

```
import org.apache.spark.sql.functions._

val inputPath = "file:///home/hduser/data/spark_demo/data.txt"

import spark.implicits
spark.read
      .option("inferSchema","true")
      .option("header","false")
      .csv(inputPath)
      .toDF("year","month","cnt")
      .select(concat_ws("-",$"year",$"month").as("ym"),$"cnt")
      .groupBy("ym")
      .agg(sort_array(collect_list("cnt")).as("cnt"))
      .orderBy("ym")
      .show
```

输出结果如下：

5.1.7 Spark3 数组函数

Spark 3 增加了一些新的数组函数，其中的 transform 和 aggregate 数组函数是功能特别强大的通用函数。它们提供的功能相当于 Scala 中的 map 和 fold，使 ArrayType 列更容易处理。

.....

5.2 聚合函数

对大数据进行分析通常都需要对数据进行聚合操作。聚合通常需要某种形式的分组，要么在整个数据集上，要么在一个或多个列上，然后对它们应用聚合函数，比如对每个组进行求和、计数或求平均值等。Spark SQL 提供了许多常用的聚合函数。

5.2.1 聚合函数

在 Spark 中，所有的聚合都是通过函数完成的。聚合函数被设计用来在一组行上执行聚合，不管那组行是由 DataFrame 中的所有行还是一组子行组成的。

下表描述了常见的聚合函数：

聚合函数	描述
count(col)	返回每组中成员数量
countDistinct(col)	返回每组中成员唯一数量
approx_count_distinct(col)	返回每组中成员唯一近似数量
min(col)	返回每组中给定列的最小值
max(col)	返回每组中给定列的最大值
sum(col)	返回每组中给定列的值的和

sumDistinct(col)	返回每组中给定列的唯一值的和
avg(col)	返回每组中给定列的值的平均
skewness(col)	返回每组中给定列的值的分布的偏斜度
kurtosis(col)	返回每组中给定列的值的分布的峰度
variance(col)	返回每组中给定列的值的无偏方差
stddev(col)	返回每组中给定列的值的标准差
collect_list(col)	返回每组中给定列的值的集合。返回的集合可能包含重复的值。
collect_set(col)	返回每组中给定列的唯一值的集合

为了演示这些函数的用法，我们将使用“2018 年 11 月 14 日深圳市价格定期监测信息”数据集。这个数据集包含一些主要副食品的监测信息，以 csv 格式存储在文件中。

下面的代码读取一个价格监测信息数据集，并创建 DataFrame。

```
  def main(args: Array[String]): Unit = {
    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    // 读取数据源文件，创建 DataFrame
    val filePath = "src/main/resources/pricewatch.csv"
    val priceDF = spark
      .read
      .option("header","true")
      .option("inferSchema","true")
      .csv(filePath)

    priceDF.printSchema()
    priceDF.show(5)
  }
```

执行以上代码，输出结果如下所示：

```
root
 |-- RECORDID: string (nullable = true)
 |-- JCLB: string (nullable = true)
 |-- JCMC: string (nullable = true)
 |-- BQ: double (nullable = true)
 |-- SQ: double (nullable = true)
 |-- TB: double (nullable = true)
 |-- HB: double (nullable = true)

+--------------------+----+--------+-----+-----+------+------+
|                RECORDID|JCLB|    JCMC|   BQ|   SQ|    TB|    HB|
+--------------------+----+--------+-----+-----+------+------+
|537B9A6E0C836F36E...|null|     椰菜| 2.27|2.261|-0.067| 0.004|
|537B9A6E0C846F36E...|null|  东北米|2.863|2.843|-0.067| 0.007|
|537B9A6E0C856F36E...|null|  早籼米| 3.08|3.044| 0.219| 0.012|
|537B9A6E0C866F36E...|null|  晚籼米|3.217| 3.22| 0.081|-0.001|
|537B9A6E0C876F36E...|null|泰国香米| 9.48| 9.48| 0.047|   0.0|
+--------------------+----+--------+-----+-----+------+------+
```

```
only showing top 5 rows
```

每一行代表从一条商品的价格监测信息。其中各个字段的含义如下：

- JCLB：监测类别。
- JCMC：监测名称。
- BQ：本期价格。
- SQ：上期价格。
- TB：同比价格变化。
- HB：环比价格变化。

找出这个数据集总共有多少行。

```
println("监测的商品数量有：" + priceDF.count())
```

执行以上代码，输出结果如下所示：

```
监测的商品数量有：331
```

下面使用一些常用的聚合函数进行统计。

1）count(col)：统计指定列的数量。

下面的代码用来统计数据集中商品（JCMC）的数量和监测类别（JCLB）的数量。

```
import org.apache.spark.sql.functions._

// 商品数量
priceDF.select(count("JCMC").as("监测商品")).show()

// 当统计一列中的项目数量时，count（col）函数不包括计数中的 null 值
priceDF.select(count("JCLB").as("监测类别")).show()

// 判断"JCLB"列为 null 值的有多少
val nullJclb = priceDF.where(col("JCLB").isNull).count()
println("\"JCLB\"列为 null 值的有：" + nullJclb)
```

执行以上代码，输出结果如下所示：

```
+--------+
|监测商品|
+--------+
|       331|
+--------+

+--------+
|监测类别|
+--------+
|       298|
+--------+

"JCLB"列为 null 值的有：33
```

2）countDistinct(col)：它只计算每个组的唯一项。

下面的代码统计总共有多少个商品类别，多少个商品。

```
// 去重：统计检测的商品类别有多少
// priceDF.select("JCLB").distinct.show()
// priceDF.select("JCLB").distinct.count()
```

```
// 去重：统计检测的商品有多少个
// priceDF.select("JCMC").distinct.show()
// priceDF.select("JCMC").distinct.count()

// countDistinct(col)：它只计算每个组的唯一项,不包括 null
priceDF.select(
    countDistinct("JCLB"),
    countDistinct("JCMC"),
    count("*")
).show()

priceDF.select(
    countDistinct("JCLB").as("监测类别"),
    countDistinct("JCMC").as("监测商品"),
    count("*").as("总数量")
).show()
```

执行以上代码，输出结果如下所示：

```
+--------------------+---------------------+--------+
|count(DISTINCT JCLB)|count(DISTINCT JCMC)|count(1)|
+--------------------+---------------------+--------+
|                   6|                   35|     331|
+--------------------+---------------------+--------+


+--------+--------+------+
|监测类别|监测商品|总数量|
+--------+--------+------+
|       6|      35|   331|
+--------+--------+------+
```

3）approx_count_distinct (col, max_estimated_error=0.05)：近似唯一计数

在一个大数据集里计算每个组中唯一的项的确切数量是一个成本很高且很耗时的操作。在某些用例中，有一个近似惟一的计数就足够了。例如，在线广告业务中，每小时有数亿个广告曝光并且需要生成一份报告来显示每个特定类型的成员段的独立访问者的数量。Spark 实现了 approx_count_distinct 函数用来统计近似惟一计数。因为唯一的计数是一个近似值，所以会有一定数量的误差。这个函数允许我们为指定一个可接受估算误差的值。下面的代码演示了 approx_count_distinct 函数的用法和行为。

```
// 统计 price DataFrame 的"JCMC"列。默认估算误差是 0.05 (5%)
priceDF.select(
        count("JCMC"),
        countDistinct("JCMC"),
        approx_count_distinct("JCMC", 0.05)
    ).show()
```

执行以上代码，输出结果如下所示：

```
+-----------+--------------------+--------------------------+
|count(JCMC)|count(DISTINCT JCMC)|approx_count_distinct(JCMC)|
+-----------+--------------------+--------------------------+
|        331|                  35|                        33|
+-----------+--------------------+--------------------------+
```

4）min(col), max(col)：获取 col 列的最小值和最大值

下面的代码中，我们统计本期价格的最大值和最小值：

```
// 统计本期价格(BQ)最大值和最小值
priceDF.select(
        min("BQ").as("最便宜的"),
        max("BQ").as("最贵的")
    ).show()
```

执行以上代码，输出结果如下所示：

```
+--------+------+
|最便宜的|最贵的|
+--------+------+
|       1.7|43.515|
+--------+------+
```

5）sum(col)：这个函数计算一个数字列中的值的总和。

例如，下面的代码计算本期价格之和：

```
priceDF.select(sum("BQ")).show()
```

执行以上代码，输出结果如下所示：

```
+------------------+
|                sum(BQ)|
+------------------+
|2904.4770000000003|
+------------------+
```

6）sumDistinct(col)：这个函数只汇总了一个数字列的不同值。

例如，下面的代码计算本期价格（唯一值）之和：

```
priceDF.select(sumDistinct("BQ")).show
```

执行以上代码，输出结果如下所示：

```
+------------------+
|  sum(DISTINCT BQ)|
+------------------+
|2544.9589999999994|
+------------------+
```

7）avg(col)：这个函数计算一个数字列的平均值。

这个方便的函数简单地取总并除以项目的数量。例如，下面的代码本期的平均价格：

```
priceDF.select(avg("BQ"), sum("BQ") / count("BQ")).show
```

执行以上代码，输出结果如下所示：

```
+----------------+--------------------+
|              avg(BQ)|(sum(BQ) / count(BQ))|
+----------------+--------------------+
|8.774854984894262|      8.774854984894262|
+----------------+--------------------+
```

8）skewness(col), kurtosis(col)

在统计领域中，数据集中的值分布说明了数据集背后的许多故事。Skewness 是一种度量数据集的值分布的对称性的度量。在正态分布或钟形分布中，倾斜值为 0。正倾斜值表明右边的尾巴比左边的长或更胖。负倾斜值表示相反，左边的尾巴比右边长或更胖。当偏度为 0 时，两边的尾巴是相当的。

Kurtosis 是对分布曲线形状的度量，不管曲线是正常的，平坦的还是尖的。正的 Kurtosis 表示曲线是细而尖的，负的 Kurtosis 表示曲线是宽而平的。下面的代码计算 BQ 列的偏斜度（Skewness）和峰度（Kurtosis）：

```
priceDF.select(skewness("BQ"), kurtosis("BQ")).show
```

执行以上代码，输出结果如下所示：

```
+------------------+-----------------+
|      skewness(BQ)|      kurtosis(BQ)|
+------------------+-----------------+
|2.2954037816302346|4.814228050842151|
+------------------+-----------------+
```

结果似乎表明，BQ 列的分布是不对称的，右边的尾巴比左边尾巴长或宽。峰度值表明分布曲线是尖的。

8）variance(col), stddev(col)

在统计学中，方差（variance）和标准偏差（stddev）用于测量数据的分散性或分布。换句话说，它们被用来说明 values 到平均值的平均距离。当方差值较低时，意味着该值接近均值。方差和标准差是相关的，后者是前者的平方根。

variance 和 stddev 函数分别用于计算方差和标准差。Spark 提供了这些函数的两种不同实现：一种是利用抽样来加速计算，另一种使用全样。下面的代码中，计算 priceDF DataFrame 中的 BQ 列的方差和标准偏差。

```
// 使用方差和标准差的两种变化
priceDF.select(variance("BQ"), var_pop("BQ"), stddev("BQ"), stddev_pop("BQ")).show
```

执行以上代码，输出结果如下所示：

```
+-----------------+-----------------+-----------------+-----------------+
|     var_samp(BQ)|      var_pop(BQ)| stddev_samp(BQ)|     stddev_pop(BQ)|
+-----------------+-----------------+-----------------+-----------------+
|85.96212834860381|85.70242403335124|9.27157636805111|9.257560371574751|
+-----------------+-----------------+-----------------+-----------------+
```

看起来 priceDF DataFrame 中的 BQ 值很分散。

5.2.2 分组聚合

分组聚合不会在 DataFrame 中对全局组执行聚合，而是在 DataFrame 中的每个子组中执行聚合。通过分组执行聚合的过程分为两步。第一步是通过使用 groupBy（col1、col2、……）转换来执行分组，也就是指定要按哪些列分组。与其他返回 DataFrame 的转换不同，这个 groupBy 转换返回一个 RelationalGroupedDataset 类的实例。类 RelationalGroupedDataset 提供了一组标准的聚合函数，可以将它们应用到每个子组中。这些聚合函数有 avg(cols)、count()、mean(cols)、min(cols)、max(cols)和 sum(cols)。除了 count()函数之外，其余所有的函数都在数字列上运行。

下面的代码按 JCLB 分组并执行一个 count 聚合：（请注意，groupBy 列将自动包含在输出中）

```
// 按检测的商品大类分组统计
priceDF.groupBy("JCLB").count().show(false)
```

执行以上代码，输出结果如下所示：

```
+------+-----+
|JCLB  |count|
+------+-----+
|粮食  |55   |
|食用油|37   |
|水产品|27   |
|null  |33   |
|肉奶蛋|7    |
```

```
|蔬菜   |117   |
|肉蛋奶|55    |
+------+-----+
```

下面的代码中，按 JCLB 和 JCMC 分组之后，执行 count 聚合，并按统计数量降序排序：

```
// 按商品大类和商品小类分组统计
priceDF
        .groupBy($"JCLB", $"JCMC")
        .count
        .orderBy($"count".desc)
        .show(false)

priceDF
        .groupBy($"JCLB", $"JCMC")
        .count
        .where($"JCMC" === "花生油")
        .orderBy($"count".desc)
        .show(false)
```

执行以上代码，输出结果如下所示：

有时需要在同一时间对每个组执行多个聚合。例如，除了计数之外，我们还想知道最小值和最大值。RelationalGroupedDataset 类提供一个名为 agg 的功能强大的函数，它接受一个或多个列表达式，这意味着可以使用任何聚合函数。这些聚合函数返回 Column 类的一个实例，这样就可以使用所提供的函数来应用任何列表达式。一个常见的需求是在聚合完成后重命名列，使之更短、更可读、更易于引用。

下面的代码中，按 JCLB 分组之后，执行多个聚合。

```
// 在同一时间对每个组执行多个聚合
import org.apache.spark.sql.functions._

priceDF
        .na.drop
        .groupBy("JCLB")
        .agg(
          count("BQ").as("本期数量"),
          min("BQ").as("本期最低价格"),
          max("BQ").as("本期最高价格"),
          avg("BQ").as("本期平均价格")
        )
        .show()
```

执行以上代码，输出结果如下所示：

这个 agg 功能提供了一种通过基于字符串的 key-value 映射来表达列表达式的附加方法。key 是列名，而 value 是一个聚合函数，它可以是 avg、max、min、sum 或 count。例如：

```
priceDF.na.drop
        .groupBy("JCLB")
        .agg(
          "BQ" -> "count",
```

```
        "BQ" -> "min",
        "BQ" -> "max",
        "BQ" -> "avg"
      )
      .show()
```

执行以上代码，输出结果如下所示：

```
```

函数 collect_list(col)和 collect_set(col)用于在应用分组后收集特定组的所有值。一旦每个组的值被收集到一个集合中，那么就可以自由地以任何我们选择的方式对其进行操作。这两个函数的返回集合之间有一个小的区别，那就是惟一性。collect_list 函数返回一个可能包含重复值的集合，collect_set 函数返回一个只包含唯一值的集合。在下面的示例中，使用 collection_list 函数来收集每个商品大类下的商品名称：

```
import org.apache.spark.sql.functions._
import spark.implicits._

priceDF.na.drop
      .groupBy('JCLB.as("监测类别"))
      .agg(collect_set("JCMC").as("监测的商品"))
      .withColumn("监测商品数量",size('监测的商品))
      .show()
```

执行以上代码，输出结果如下所示：

```
```

5.2.3 数据透视

数据透视是一种通过聚合和旋转把数据行转换成数据列的技术，它是一种将行转换成列同时应用一个或多个聚合时的方法。这样一来，分类值就会从行转到单独的列中。这种技术通常用于数据分析或报告。

数据透视过程从一个或多个列的分组开始，然后在一个列上旋转，最后在一个或多个列上应用一个或多个聚合。因此，当透视数据时，需要确定三个要素：要在行（分组元素）中看到的元素，要在列（扩展元素）上看到的元素，要在数据部分看到的元素（聚合元素）。

在下面的例子中，有一个包含学生信息的数据集，每行包含学生姓名、性别、体重、毕业年份。现在想要知道每个毕业年份每个性别的平均体重：

```
case class Student(name:String, gender:String, weight:Int, graduation_year:Int)

def main(args: Array[String]): Unit = {
  // 创建 SparkSession 的实例
  val spark = SparkSession.builder()
    .master("local[*]")
    .appName("Spark Basic Example")
    .getOrCreate()

  import spark.implicits._

  // 转为 DataFrame
  val studentsDF = Seq(
```

```
        Student("刘宏明", "男", 180, 2015),
        Student("赵薇", "女", 110, 2015),
        Student("黄海波", "男", 200, 2015),
        Student("杨幂", "女", 109, 2015),
        Student("楼一萱", "女", 105, 2015),
        Student("龙梅子", "女", 115, 2016),
        Student("陈知远", "男", 195, 2016)
    ).toDF()

    // studentsDF.show()

    // 计算每年每个性别的平均体重
    studentsDF.groupBy("graduation_year").pivot("gender").avg("weight").show()
  }
```

执行以上代码，输出结果如下所示：

可以利用 agg 函数来执行多个聚合，这会在结果表中创建更多的列：

```
studentsDF
        .groupBy("graduation_year")
        .pivot("gender")
        .agg(
          min("weight").as("min"),
          max("weight").as("max"),
          avg("weight").as("avg")
        ).show()
```

执行以上代码，输出结果如下所示：

如果 pivot 列有许多不同的值，可以选择性地选择生成聚合的值。例如：

```
studentsDF
        .groupBy("graduation_year")
        .pivot("gender", Seq("男"))
        .agg(
          min("weight").as("min"),
          max("weight").as("max"),
          avg("weight").as("avg")
        ).show()
```

执行以上代码，输出结果如下所示：

为 pivot 列指定一个 distinct 值的列表实际上会加速旋转过程。

Spark SQL 支持一个非常有用的特性：谓词子查询。谓词子查询是指操作数为子查询的谓词。Spark 2.0 支持 EXISTS 和 IN 这两种基本形式。Spark 2.0 目前只支持 WHERE 子句中的谓词子查询。

这意味着可以使用来自另一个查询的内容方便地过滤主查询中的行。当只需要从一个表中选择行，而另一个表中存在匹配内容时，这将非常有用。

以下面这个查询为例，它使用谓词子查询来根据在另一个表 interesting_users 中找到的用户筛选

clickstream 中的用户。

```
select count(1)
from clickstream
where user in (select distinct user from interesting_users)
```

5.3 高级分析函数

本节将介绍 Spark SQL 提供的高级分析函数。

5.3.1 使用多维聚合函数

在高级分析函数中，第一个是关于多维聚合的，它对于涉及分层数据分析的用例非常有用，在这种情况下，通常需要在一组分组列中计算子总数和总数。rollup 和 cube 基本上是在多列上进行分组的高级版本，它们通常用于在这些列的组合和排列中生成子总数和大总数。所提供的一组列的顺序被视为分组的层次结构。

rollup

当使用分层数据时，比如不同部门和分部的销售收入数据等，rollup 可以很容易地计算出它们的子总数和总数。rollup 按给定的列集的层次结构，并且总是在层次结构中的第一列启动 rolling up 过程。下面的代码演示如何使用一个 rollup：

```
// 读取超市订单汇总数据
val filePath = "/data/spark_demo/超市订单.csv"
val ordersDF = spark.read.option("header", "true").option("inferSchema","true").csv(filePath )

ordersDF.count
ordersDF.printSchema
ordersDF.show(3)
```

执行以上代码，输出结果如下所示:

```
// 将数据过滤到更小，以便更容易地看到 rollup 的结果
val twoSummary = ordersDF.select($"地区", $"省/自治区", $"订单 ID").
                                where($"地区" === "华东" || $"地区" === "华北")

// 让我们看看数据是什么样子的
twoSummary.count
twoSummary.show()
```

执行以上代码，输出结果如下所示:

```
// 按地区、省/自治区执行 rollup，然后计算计数的总和，最后按 null 排序
twoSummary.rollup($"地区", $"省/自治区").
            agg(count("订单 ID") as "total").
            orderBy($"地区".asc_nulls_last, $"省/自治区".asc_nulls_last).
            show()
```

执行以上代码，输出结果如下所示:

这个输出显示了华东区和华北区的每个城市的子总数，而总计显示在最后一行，并带有在"地区"和"省/自治区"的列上的 null 值。注意带有 asc_nulls_last 选项进行排序，因此 Spark SQL 会将 null 值排序到最后位置。

cube

一个 cube 基本上是一个更高级的 rollup。它在分组列的所有组合中执行聚合。因此，结果包括 rollup 提供的以及其他组合所提供的。在我们的"地区"和"省/自治区"的例子中，结果将包括每个"省/自治区"的聚合。使用 cube 函数的方法类似于如何使用 rollup 函数。请看下面的示例：

```
// 执行 cube
twoSummary.cube($"地区", $"省/自治区").
            agg(count("订单  ID") as "total").
            orderBy($"地区".asc_nulls_last, $"省/自治区".asc_nulls_last).
            show(30)
```

执行以上代码，输出结果如下所示:

在结果表格中，在"地区"列中有 null 值的行表示一个地区中所有城市的聚合。因此，一个 cube 的结果总是比会 rollup 的结果有更多的行。

5.3.2 使用时间窗口聚合

在高级分析函数中，第二个功能是基于时间窗口执行聚合，这在处理来自物联网设备的事务或传感器值等时间序列数据时非常有用。

在 Spark 2.0 中引入了时间窗口的聚合，使其能够轻松地处理时间序列数据，这些数据由一系列的时间顺序数据点组成。这种数据集在金融或电信等行业很常见。例如，股票市场交易数据集有交易日期、开盘价、收盘价、交易量和每个股票代码的其他信息。时间窗口聚合可以帮助回答问题，比如京东股票的周平均收盘价，或者京东股票跨每一周的月移动平均收盘价。

时间窗口函数有几个版本，但是它们都需要一个时间戳类型列和一个窗口长度，该窗口长度可以指定为几秒、几分钟、几小时、几天或几周。窗口长度代表一个时间窗口，它有一个开始时间和结束时间，它被用来确定一个特定的时间序列数据应该属于哪个桶。有两种类型的时间窗口：滚动窗口和滑动窗口。与滚动窗口（也叫固定窗口）相比，滑动窗口需要提供额外的输入参数，用来说明在计算下一个桶时，一个时间窗口应该滑动多少。

下面的例子将使用京东股票交易，可以在雅虎财经网站上找到。下面的代码根据一年的数据计算京东股票的周平均价格。

```
// 加载京东股票 2018 年交易数据
val csvPath = "/data/spark_demo/JD2018.csv"
val jd2018DF = spark.read.option("header", "true").option("inferSchema","true").csv(csvPath)

// 显示该 schema, 第一列是交易日期
jd2018DF.printSchema
jd2018DF.take(1)
```

执行以上代码，输出结果如下所示:

对交易日期进行整理（将 Date 字段的字符串类型转为 Date 类型）

```
val jdOneYearDF = jd2018DF.withColumn("Date",to_date('Date,"yyyy/MM/dd"))
jdOneYearDF.printSchema
jdOneYearDF.take(1)
```

执行以上代码，输出结果如下所示:

使用时间窗口函数来计算 Apple 股票的平均收盘价:

```
// 使用窗口函数计算 groupBy 变换内的周平均价格
// 这是一个滚动窗口的例子，也就是固定窗口
val jdWeeklyAvgDF = jdOneYearDF.groupBy(window('Date, "1 week")).
                                    agg(avg("Close").as("weekly_avg"))

// 结果模式有窗口启动和结束时间
jdWeeklyAvgDF.printSchema

// 按开始时间顺序显示结果，并四舍五入到小数点后 2 位
jdWeeklyAvgDF.orderBy("window.start").
                selectExpr("window.start", "window.end", "round(weekly_avg, 2) as weekly_avg").
                show(5)
// 注：包含起始时间，不包含结束时间
```

执行以上代码，输出结果如下所示:

前面的例子使用了一个星期的滚动窗口，其中没有重叠。因此，每个交易只使用一次来计算移动平均值。下面的例子使用了滑动窗口。这意味着在计算平均每月移动平均值时，一些交易将被多次使用。窗口的大小是四个星期，每个窗口一次滑动一个星期。

```
// 使用时间窗口函数来计算 Apple 股票的月平均收盘价
// 4 周窗口长度，每次 1 周的滑动
val jdMonthlyAvgDF = jdOneYearDF.groupBy(window('Date, "4 week", "1 week")).
                                    agg(avg("Close").as("monthly_avg"))

// 按开始时间显示结果
jdMonthlyAvgDF.orderBy("window.start").
                selectExpr("window.start", "window.end", "round(monthly_avg, 2) as monthly_avg").
                show(5)
```

执行以上代码，输出结果如下所示:

由于滑动窗口间隔是一个星期，所以先前的结果表显示两个连续行的起始时间间隔是一个星期的间隔。在连续两行之间，有大约三周的重叠交易，这意味着一个交易被多次使用来计算移动平均值。

5.3.3 使用窗口函数

在高级分析函数中，第三个是在逻辑分组中执行聚合的能力，这个逻辑分组被称为窗口。使用窗口函数，我们能够轻松地执行例如移动平均、累积和/或每一行的排名这样的计算。它们显著提高了 Spark 的 SQL 和 DataFrame API 的表达能力。

有时需要对一组行进行操作，并为每个输入行返回一个值。窗口函数提供了这种独特的功能，使其易于执行计算，如移动平均、累积和或每一行的 rank。

使用窗口函数有两个主要步骤。

- 第一步是定义一个窗口规范，该规范定义了称为 frame 的行逻辑分组，这是每一行被计算的上下文。
- 第二步是应用一个合适的窗口函数。

窗口规范定义了窗口函数将使用的三个重要组件。第一个组件被称为 partition by，指定用来对行进行分组的列（一个或多个列）。第二个组件称为 order by，它定义了如何根据一个或多个列来排序各行，以及顺序是升序或降序。在这三个组件中，最后一个更复杂，需要详细的解释。最后一个组件称为 frame，它定义了窗口相对于当前行的边界。换句话说，“frame”限制了在计算当前行的值时包括哪些行。可以通过行索引或 order by 表达式的实际值来指定在 window frame 中包含的一系列行。最后一个组件是可选的，有的窗口函数需要，有的窗口函数或场景不需要。窗口规范是使用在 org.apache.spark.sql.expressions.Window 类中定义的函数构建的。rowsBetween 和 rangeBetween 函数分别用来定义行索引和实际值的范围。

窗口函数可分为三种类型：排序函数、分析函数和聚合函数。在下面两个表中分别描述了排序函数和分析函数。对于聚合函数，可以使用前面提到的任何聚合函数作为窗口函数。

关于窗口函数的完整列表，请参考以下链接：https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/functions.html

ranking 函数：

函数名称	描述
rank	返回一个frame内行的排名和排序，基于一些排序规则
dense_rank	类似于rank，但是在不同的排名之间没有间隔，紧密衔接显示
ntile(n)	在一个有序的窗口分区中返回ntile分组ID。比如，如果n是4，那么前25%行得到的ID值为1，第二个%25行得到的ID值为2，依次类推。
row_number	返回一个序列号，每个frame从1开始

分析函数：

函数名称	描述
cume_dist	返回一个frame的值的累积分布。换句话说，低于当前行的行的比例。
lag(col,offset)	返回当前行之前offset行的列值
lead(col,offset)	返回当前行之后offset行的列值

让我们通过一个小的样本数据集来演示窗口函数功能。要求计算过去三个月中每个月的平均销售额。数据文件：MonthlySales.csv，包含 24 个观察数据，两个产品(P1 和 P2)的月销售数据。

```
// 创建一个 DataFrame，包含两个产品的月销售数据
val file = "/data/spark_demo/MonthlySales.csv"
val monthlySales = spark.read.option("header","true").option("inferSchema","true").csv(file)

monthlySales.show()
```

执行以上代码，输出结果如下所示:

```
import org.apache.spark.sql.expressions.Window

// 准备 WindowSpec，为每个产品创建一个包含三个月的滑动窗口
// 负数下标表示在当前行之上（前）的行
val w = Window.partitionBy("Product").orderBy("Month").rangeBetween(-2,0)
```

```
// val w = Window.partitionBy(monthlySales("Product"))
                  .orderBy(monthlySales("Month"))
                  .rangeBetween(-2,0)


// 在该滑动窗口上定义计算，在本例中是一个移动的平均值
val f = avg("Sales").over(w)
// val f = avg(monthlySales("Sales")).over(w)

//应用该滑动窗口和计算。检查结果
monthlySales.select($"Product",$"Sales",$"Month",bround(f,2).alias("MovingAvg")).
                orderBy($"Product",$"Month").
                show()

// 以上两行可以合并到下面这一行
// monthlySales.select($"Product",$"Sales",$"Month",bround(avg("Sales").over(w),2).alias("MovingAvg")).
//                  orderBy($"Product",$"Month").
//                  show()
```

执行以上代码，输出结果如下所示:

【示例】现在假设有两个用户 user01 和 user02，下面这是两个用户的购物交易数据：

用户ID	交易日期	交易金额
user01	2018-07-02	13.35
user01	2018-07-06	27.33
user01	2018-07-04	21.72
user02	2018-07-07	69.74
user02	2018-07-01	59.44
user02	2018-07-05	80.14

有了这个购物交易数据，让我们尝试使用窗口函数来回答以下问题：

- 对于每一个用户，最高的交易金额是多少？
- 每个用户的交易金额和最高交易金额之间的差是多少？
- 每个用户的交易金额相对上一次交易的变化是多少？
- 每个用户的移动平均交易金额是多少？
- 每个用户的交易金额的累计金额是多少？

下面我们应用窗口函数来回答这些问题：

1）为了回答第一个问题，可以将 rank 窗口函数应用于一个窗口规范，该规范按用户 ID 对数据进行分区，并按交易金额对其进行降序排序。rank 窗口函数根据每一 frame 中每一行的排序顺序给每一行分配一个等级。请参考下面解决第一个问题的实际代码。

执行以上代码，输出结果如下所示:

可以看出，用户 user01 的最高交易金额是 27.33，用户 user02 的最高交易金额是 80.14。

2）解决第二个问题的方法是在每个分区的所有行的 amount 列上应用 max 函数。除了按用户 ID 分区之外，它还需要定义一个包含每个分区中所有行的 frame 边界。要定义这个 frame，我们可以使用 Window.rangeBetween 函数，以 Window.unboundedPreceding 作为开始值，以 Window.unboundedFollowing 作为结束值。下面的代码根据前面定义的逻辑定义窗口规范，并将 max 函数应用于它之上。

执行以上代码，输出结果如下所示:

3）解决第三个问题的方法是使用每个分区的当前行的 amount 列减去上一行的 amount 列。获取上一行的指定字段用 lag 函数。除了按用户 ID 分区之外，它还需要定义一个包含每个分区中所有行的 frame 边界。默认 frame 包括所有前面的行和当前行。

执行以上代码，输出结果如下所示:

4）为了计算每个用户按交易日期顺序移动的平均移动数量，我们将利用 avg 函数来根据 frame 中的一组行计算每一行的平均数量。这里希望每一 frame 都包含三行：当前行加上前面的一行和后面的一行。与前面的例子类似，窗口规范将按用户 ID 对数据进行分区，但是每一个 frame 中的行将按交易日期排序。下面代码展示了如何将 avg 函数应用于前面描述的窗口规范。

执行以上代码，输出结果如下所示:

5）为了计算每个用户的交易金额的累积总和，我们将把 sum 函数应用于一个 frame，该 frame 由所有行到当前行组成。其 partition by 和 order by 子句与移动平均示例相同。下面的代码展示了如何将 sum 函数应用于前面描述的窗口规范。

执行以上代码，输出结果如下所示:

窗口规范的默认 frame 包括所有前面的行和当前行。对于前面的例子，没有必要指定 frame，所以应该得到相同的结果。

前面的窗口函数示例是使用 DataFrame APIs 编写的。也可以通过使用 SQL 来实现相同的目标，使用关键字：PARTITION BY, ORDER BY, ROWS BETWEEN, RANGE BETWEEN。frame 边界可以使用以下关键字来指定：UNBOUNDED PRECEDING, UNBOUNDED FOLLOWING, CURRENT ROW, <value> PRECEDING, 和<value> FOLLOWING。

下面的代码展示了使用 SQL 的窗口函数的示例。

执行上面的代码，输出结果应当与使用 DataFrame API 进行窗口操作的结果一样。当使用 SQL 中的窗口函数时，必须在单个语句中指定 partition by、order by 和 frame 窗口。

5.4 用户自定义函数(UDF)

尽管 Spark SQL 为大多数常见用例提供了大量的内置函数，但总会有一些情况下，这些功能都不能提供我们的用例所需要的功能。Spark SQL 提供了一个相当简单的工具来编写用户定义的函数（UDF），并在 Spark 数据处理逻辑或应用程序中使用它们，就像使用内置函数一样。UDF 实际上是我们可以扩展 Spark 的功能以满足特定需求的一种方式。Spark 的 UDF 可以用 Python、Java 或 Scala 来写，它们可以利用和集成任何必要的库。

从概念上讲，UDF 只是一些常规的函数，它们接受一些输入并提供输出。尽管 UDF 可以用 Scala、Java 或 Python 编写，但是必须注意当 UDF 用 Python 编写时，性能差异。UDF 必须在使用 Spark 之前注册，因此 Spark 知道将它们发送到 executor，以便使用和执行。鉴于 executor 是用 Scala 编写的 JVM 进程，他们可以在同一个进程中本地执行 Scala 或 Java UDF。如果一个 UDF 是用 Python 编写的，那么 executor 就不能本地执行它，因此它必须生成一个单独的 Python 进程来执行 Python UDF。除了生成 Python 过程的成本之外，在数据集中的每一行中都要对数据进行序列化，这是一个很大的成本。

5.4.1 用户定义标量函数

使用 UDF 涉及有三个步骤。第一个是编写一个函数并进行测试。第二步是通过将函数名及其签名传递给 Spark 的 udf 函数来注册该函数。最后一步是在 DataFrame 代码或发出 SQL 查询时使用 UDF。在 SQL 查询中使用 UDF 时，注册过程略有不同。下面的代码用一个简单的 UDF 演示前面提到的三个步骤。

```
// 在 Scala 中一个简单的 UDF，将数字等级转换为考查等级

// 导入依赖包
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

// 定义 case class
case class Student(name:String, score:Int)

  def main(args: Array[String]): Unit = {

    // 创建 SparkSession 的实例
    val spark = SparkSession.builder()
      .master("local[*]")
      .appName("Spark Basic Example")
      .getOrCreate()

    // 在 Scala 中一个简单的 UDF，将数字等级转换为考查等级
    import spark.implicits._

    // 创建学生成绩 DataFrame
    val studentDF = Seq(
      Student("张三", 85),
      Student("李四", 90),
      Student("王老五", 55)
    ).toDF()

    // 注册为视图
```

```
    studentDF.createOrReplaceTempView("students")

    spark.sql("select * from students").show()

    // 创建一个函数(普通的 Scala 函数)将成绩转换到考察等级
    def convertGrade(score:Int) : String = {
      score match {
        case `score` if score > 100 => "作弊"
        case `score` if score >= 90 => "优秀"
        case `score` if score >= 80 => "良好"
        case `score` if score >= 70 => "中等"
        case _ => "不及格"
      }
    }

    // 注册为一个 UDF（在 DSL API 中使用时的注册方法）
    val convertGradeUDF = udf(convertGrade(_:Int):String)

    // 使用该 UDF 将成绩转换为字母等级
    studentDF.select($"name",$"score", convertGradeUDF($"score").as("grade")).show()
  }
```

执行以上代码，输出结果如下所示：

在 SQL 查询中使用 UDF 时，注册过程略有不同：

```
// 注册为 UDF，在 SQL 中使用
spark.udf.register("convertGrade", convertGrade(_: Int): String)
spark.sql("select name, score, convertGrade(score) as grade from students").show()
```

执行以上代码，输出结果如下所示：

内置的 DataFrames 函数提供常见的聚合，如 count()、countDistinct()、avg()、max()、min()等。虽然这些函数是为 DataFrames 设计的，但 Spark SQL 也有类型安全的版本，其中一些在 Scala 和 Java 中可以使用强类型数据集。此外，用户不仅限于预定义的聚合函数，还可以创建自己的聚合函数。

5.4.2 无类型的用户定义聚合函数

用户必须继承 UserDefinedAggregateFunction 抽象类来实现自定义的无类型聚合函数。例如，用户定义的计算平均值的聚合函数可以如下(这里为了演示如何创建 UDAF，因为 Spark 内置了 avg 函数)：

虽然 UDAF 编写起来很复杂，但与 Dataset 上的 mapGroups 或简单地编写 RDD 上的 aggregateByKey 等选项相比，UDAF 的性能相当好。然后，我们可以直接在列上使用 UDAF，也可以像对非聚合 UDF 那样将其添加到函数注册中心。

5.4.3 类型安全的用户定义聚合函数

强类型 Dataset 的用户定义聚合函数(UDAF)围绕 Aggregator 抽象类进行。例如，一个类型安全的用

户定义的平计算均值的聚合函数看起来像下面这样：

5.5 join 连接

本节将介绍如何在 Spark SQL 中使用 join transformation 和它支持的各种类型的 join 来执行多个 DataFrame/Dataset 的连接。本节的最后一部分介绍了 Spark SQL 如何在内部执行 join 连接的一些细节。

5.5.1 join 表达式和 join 类型

执行两个数据集的连接需要指定两个内容。第一个是连接表达式，它指定来自每个数据集的哪些列应该用于确定来自两个数据集的哪些行将被包含在连接后的数据集中（确定连接列/等值列）。第二种是连接类型，它决定了连接后的数据集中应该包含哪些内容。

下表描述了在 Spark SQL 中所支持的 join 类型：

类型	描述
内连接(又叫等值连接)	当连接表达式计算结果为true时，返回来自两个数据集的行
左外连接	甚至当连接表达式计算结果为false时，也返回来自左侧数据集的行
右外连接	甚至当连接表达式计算结果为false时，也返回来自右侧数据集的行
外连接	甚至当连接表达式计算结果为false时，也返回来自两侧数据集的行
左反连接	当连接表达式计算结果为false时，只返回来自左侧数据集的行
左半连接	当连接表达式计算结果为true时，只返回来自左侧数据集的行
交叉连接(又名笛卡尔连接)	返回左数据集中每一行和右数据集中每一行合并后的行。行的数量将是两个数据集的乘积

左半连接和左反连接是唯一的只有值来自左表的连接类型。左半连接与过滤左表中只有在右表中存在键的行相同。左反连接也只返回来自左表的数据，但只返回右边表中不存在的记录。

DataFrames 支持自连接，但是我们最终会得到重复的列名。为了能够访问结果，需要将 DataFrames 别名为不同的名称—否则由于名称冲突将无法选择列。一旦您为每个 DataFrame 设置了别名，在结果中您就可以使用 dfName.colName 访问每个 DataFrame 的各个列。例如：

```
val joined = df.as("a").join(df.as("b")).where($"a.name" === $"b.name")
```

Broadcast hash join

在 Spark SQL 中，通过调用 queryExecution.executedPlan 可以看到正在执行的连接类型。与 core Spark 一样，如果其中一个表比另一个表小得多，可能需要广播散列连接。可以向 Spark SQL 提示，一个给定的 DF 应该在 join 连接之前通过在 DataFrame 上调用 broadcast 对其进行广播来进行 join 连接。例如：

```
df1.join(broadcast(df2), "key")
```

Spark 还自动使用 spark.sql.conf. autoBroadcastJoinThreshold 来确定是否应该广播表。

5.5.2 使用 join

下面使用两个小 DataFrame 来演示如何在 Spark SQL 中使用 join 连接。第一个 DataFrame 代表一个员工列表，每一行包含员工姓名和所属部门。第二个 DataFrame 包含一个部门列表，每一行包含一个部门 ID 和部门名称。创建这两个 DataFrame 的代码片段如下：

```
// 员工
case class Employee(first_name:String, dept_no:Long)
```

```
val employeeDF = Seq(Employee("刘宏明", 31),
                     Employee("赵薇", 33),
                     Employee("黄海波", 33),
                     Employee("杨幂", 34),
                     Employee("楼一萱", 34),
                     Employee("龙梅子", null.asInstanceOf[Int])
                    ).toDF

// 部门
case class Dept(id:Long, name:String)

val deptDF = Seq(Dept(31, "销售部"),
                 Dept(33, "工程部"),
                 Dept(34, "财务部"),
                 Dept(35, "市场营销部")
                ).toDF

// 将它们注册为 view 视图，然后就可以使用 SQL 来执行 join 连接
employeeDF.createOrReplaceTempView("employees")
deptDF.createOrReplaceTempView("departments")
```

内连接

这是最常用的连接类型，它使用相等比较的连接表达式，包含来自两个数据集与连接条件相匹配的列。连接的数据集只有当连接表达式结果为真时才包含行。没有匹配列值的行将被排除在连接数据集之外。在 Spark SQL 中，内联接是默认连接类型。下面的示例按 DepartmentID 执行内连接：

```
// 定义相等比较的 join 表达式
val joinExpression = employeeDF.col("dept_no") === deptDF.col("id")

// 执行该 join
employeeDF.join(deptDF, joinExpression, "inner").show

// 不需要指定该 join 类型，因为"inner"是默认的
employeeDF.join(deptDF, joinExpression).show

// 使用 SQL
spark.sql("select * from employees JOIN departments on dept_no == id").show
```

执行上面的代码，输出结果如下所示：

连接表达式可以在 join 转换中指定，也可以使用 where 变换。如果列名是唯一的，则可以使用简写引用 join 表达式中的列。如果没有，则需要通过使用 col 函数指定特定列来自哪个 DataFrame。下面的代码演示了表示一个 join 表达式的不同方式：

```
// join 表达式的简写版本
employeeDF.join(deptDF, 'dept_no === 'id).show

// 在 join transformation 内指定 join 表达式
employeeDF.join(deptDF, employeeDF.col("dept_no") === deptDF.col("id")).show
```

```
// 使用 where transformation 指定 join 表达式
employeeDF.join(deptDF).where('dept_no === 'id).show
```

执行上面的代码，输出结果如下所示：

左外连接

这个join类型的连接后的数据集包括来自内连接的所有行加上来自左边数据集的连接表达式的计算结果为 false 的所有行。对于那些不匹配的行，它将为右边的数据集的列填充 NULL 值。下面是一个做左外连接的例子：

```
// 连接类型既可以是"left_outer"，也可以是"leftouter"
employeeDF.join(deptDF, 'dept_no === 'id, "left_outer").show

// 使用 SQL
spark.sql("select * from employees LEFT OUTER JOIN departments on dept_no == id").show
```

执行上面的代码，输出结果如下所示：

右外连接

这种 join 类型的行为类似于左外连接类型的行为，除了将相同的处理应用于右边的数据集之外。换句话说，连接后的数据集包括来自内连接的所有行加上来自右边数据集的连接表达式的计算结果为 false 的所有行。对于那些不匹配的行，它将为左边数据集的列填充 NULL 值。下面是一个右外连接的示例。

```
// 连接类型既可以是"right_outer"，也可以是"rightouter"
employeeDF.join(deptDF, 'dept_no === 'id, "right_outer").show

// 使用 SQL
spark.sql("select * from employees RIGHT OUTER JOIN departments on dept_no == id").show
```

执行上面的代码，输出结果如下所示：

外连接 (又叫全外连接)

这种 join 类型的行为实际上与将左外连接和右外连接的结果结合起来是一样的。下面是全外连接的示例。

```
// 使用 join 转换
employeeDF.join(deptDF, 'dept_no === 'id, "outer").show

// 使用 SQL
spark.sql("select * from employees FULL OUTER JOIN departments on dept_no == id").show
```

执行上面的代码，输出结果如下所示：

左反连接

这种 join 类型能够发现来自左边数据集的哪些行在右边的数据集上没有任何匹配的行，而连接后的

数据集只包含来自左边数据集的列。下面是执行 left anti-join 的例子：

```
// 使用 join 转换
employeeDF.join(deptDF, 'dept_no === 'id, "left_anti").show

// 使用 SQL
spark.sql("select * from employees LEFT ANTI JOIN departments on dept_no == id").show
```

注意：没有 right anti-join 类型。

执行上面的代码，输出结果如下所示：

左半连接

这种 join 类型的行为类似于内连接类型，除了连接后的数据集不包括来自右边数据集的列。我们可以将这种 join 类型看作与 left anti-join 相反，在这里，连接后的数据集只包含匹配的行。下面是一个左半连接的示例：

```
// 使用 join 转换
employeeDF.join(deptDF, 'dept_no === 'id, "left_semi").show

// 使用 SQL
spark.sql("select * from employees LEFT SEMI JOIN departments on dept_no == id").show
```

执行上面的代码，输出结果如下所示：

交叉连接 (又称笛卡尔连接)

执行交叉连接的代码如下所示：

```
// 使用 crossJoin transformation 并显示该 count
employeeDF.crossJoin(deptDF).count                    // Long = 24

// 使用 SQL，并显示前 30 行以观察连接后的数据集中所有的行
spark.sql("select * from employees CROSS JOIN departments").show(30)
```

执行上面的代码，输出结果如下所示：

5.5.3 处理重复列名

有时，在 join 两个具有一个或多个有相同名称的列的 DataFrames 之后，会出现一个意想不到的问题。当这种情况发生时，连接后的 DataFrame 会有多个同名的列。在这种情况下，在对连接后的 DataFrame 进行某种转换时，就不太好引用其中一列。下面模拟这个过程。

```
// 向 deptDF 增加一个新的列，列名为 dept_no
val deptDF2 = deptDF.withColumn("dept_no", 'id)

deptDF2.printSchema
```

执行上面的代码，输出结果如下所示：

```
root
 |-- id: long (nullable = false)
 |-- deptname: string (nullable = true)
 |-- dept_no: long (nullable = false)
```

现在，使用 employeeDF 连接 deptDF2，基于 dept_no 列进行连接：

```
val dupNameDF = employeeDF.join(deptDF2, employeeDF.col("dept_no") === deptDF2.col("dept_no"))

dupNameDF.printSchema
```

执行上面的代码，输出结果如下所示：

注意，dupNameDF DataFrame 现在有两个名称相同的列，都叫 dept_no。当试图在 depNameDF DataFrame 中投影 dept_no 列时，Spark 会抛出一个错误。例如，执行下面的语句，选择 dept_no 列：

```
dupNameDF.select("dept_no")
```

执行上面的代码，输出结果如下所示：

要解决这个问题，可以有以下几种方法。

1）使用原始的 DataFrame

连接后的 DataFrame 记得在连接过程中哪些列来自哪个原始的 DataFrame。为了消除某个特定列来自哪个 DataFrame 的歧义，可以告诉 Spark 以其原始的 DataFrame 名称作为前缀。请看下面的代码。

```
// 解决方法一：明确来自哪个 DataFrame
// dupNameDF.select(employeeDF.col("dept_no")).show
dupNameDF.select(deptDF2.col("dept_no")).show
```

执行上面的代码，输出结果如下所示：

2）join 之前重命名列

为了避免列名称的模糊性问题，另一种方法是使用 withColumnRenamed 转换来重命名其中一个 DataFrames 中的列。

```
// 解决方法二：join 之前重命名列，使用 withColumnRenamed 转换来重命名其中一个 DataFrames 中的列
val deptDF3 = deptDF2.withColumnRenamed("dept_no","dept_id")
deptDF3.printSchema

val dupNameDF2 = employeeDF.join(deptDF3, 'dept_no === 'dept_id)
dupNameDF2.printSchema

dupNameDF2.select("dept_no").show
```

执行上面的代码，输出结果如下所示：

3）使用一个连接后的列名

在两个 DataFrams 中，当连接的列名是相同的时，在 join 函数中指定一个连接列名即可，这会自动

从连接后的 DataFrame 中删除重复列名。但是，如果这是一个自连接，也就是说 join 一个 DataFrame 本身，那么就没有办法引用其他重复的列名。在这种情况下，需要使用第一个技术来重命名一个 DataFrame 的列。请看下面的代码：

```
val noDupNameDF = employeeDF.join(deptDF2, "dept_no")

noDupNameDF.printSchema
```

执行上面的代码，输出结果如下所示：

```
root
 |-- dept_no: long (nullable = false)
 |-- empname: string (nullable = true)
 |-- id: long (nullable = false)
 |-- deptname: string (nullable = true)
```

注意，在 noDupNameDF DataFrame 中只有一个 dept_no 列。

5.5.4 join 实现概述

join 连接是 Spark 中最昂贵的操作之一。在较高的层次上，有两种不同的策略可以用来连接两个数据集，它们是 shuffle hash join 和 broadcast join。选择特定策略的主要标准是基于两个数据集的大小。当两个数据集的大小都很大时，就会使用 shuffle hash join 策略。当其中一个数据集的大小足够小，可以容纳进 executors 的内存时，就会使用 broadcast join 策略。下面的部分给出了每个连接策略如何工作的细节。

Shuffle Hash Join

shuffle hash join 实现由两个步骤组成。第一步是计算在每个数据集的每一行的连接表达式中的列的 hash 值，然后将这些具有相同 hash 值的行 shuffle 到同一分区。为了确定某一行将被移动到哪个分区，Spark 执行一个简单的算术操作，它通过分区的数量来计算 hash 值的模。一旦第一步完成，第二步就将那些具有相同列 hash 值的行的列组合起来。在较高的层次上，这两个步骤与 MapReduce 编程模型的步骤很相似。

下图演示了该 shuffle hash join 中 shuffling 的过程。正如前面提到的，这是一个昂贵的操作，因为它需要通过网络在多个机器上移动大量数据。当在网络上移动数据时，数据通常会经过数据序列化和反序列化过程。想象一下，在两个大型数据集上执行一个 join 连接，其中每个数据集的大小为 100 GB。在这个场景中，它需要移动大约 200 GB 的数据。在 join 两个大型数据集时，不可能完全避免 shuffle hash join，但重要的是要注意在可能的情况下减少 join 它们的频率。

Broadcast Hash Join

只有当其中一个数据集足够小到可以装入内存时，这种 join 策略才适用。知道 shuffle hash join 是一项昂贵的操作，broadcast hash join 避免了对两个数据集都进行 shuffling，而是只对较小的数据进行 shuffling。与 shuffle hash join 策略类似，这个策略也包含两个步骤。第一步是将整个小数据集的副本广播到较大数据集的每个分区上。第二步是遍历较大数据集中的每一行，并在较小的数据集中按匹配列值查找对应的行。下图演示了较小数据集的广播：

很容易理解，在可能的情况下，首选 broadcast hash join。Spark SQL 在大多数情况下都可以根据在

读取数据时对数据集的一些统计数据自动判断是否使用 broadcast hash join 或 shuffle hash join。然而，在使用 join 转换时，提供一个提示给 Spark SQL 来使用 broadcast hash join 是可行的。下面的示例代码演示了这种做法：

```
// 提供一个提示，使用一个 broadcast hash join 来广播 deptDF
import org.apache.spark.sql.functions.broadcast

// 输出执行计划以验证 broadcast hash join 策略被使用了
employeeDF.join(broadcast(deptDF), employeeDF.col("dept_no") === deptDF.col("id")).explain()

// 使用 SQL
spark.sql("select /*+ MAPJOIN(departments) */ * from employees JOIN departments on dept_no == id").explain()
```

执行上面的代码，输出结果如下所示：

从输出的物理计划可以看到，利用到了 broadcast hash join。

5.5.5 Dataset join

连接 Dataset 是通过 joinWith 完成的，其行为与常规关系连接类似，只是结果是不同记录类型的元组。在连接之后使用这一点有点尴尬，但也使 self 连接更加容易，因为不需要首先对列进行别名。

连接两个 Dataset：

```
val result: Dataset[(RawPanda, CoffeeShop)] = pandas.joinWith(coffeeShops, $"zip" === $"zip")
```

自连接一个 Dataset：

```
val result: Dataset[(RawPanda, RawPanda)] = pandas.joinWith(pandas, $"zip" === $"zip")
```

使用自连接和 lit(true)，可以生成 Dataset 的笛卡尔乘积，这可能很有用，但也说明了连接(特别是自连接)容易导致不可用的数据大小。

与 DataFrame 一样，可以指定所需的连接类型(例如，inner, left_outer, right_outer, left_semi)，改变仅在一个 Dataset 中显示的记录的处理方式。缺失的记录由 null 值表示，所以要小心。

5.6 深入理解数据分区

数据分区对数据处理性能至关重要，尤其是在 Spark 中处理大量数据时更是如此。Spark 中的分区不会跨节点，尽管一个节点可以包含多个分区。在处理时，Spark 为每个分区分配一个任务（task），每个工作线程一次只能处理一个任务（task）。因此，如果分区太少，应用程序就不能利用集群中所有可用的内核，这会导致数据倾斜问题；如果分区太多，Spark 管理太多的小任务会带来开销。默认情况下，每个线程将把数据读入一个分区。

那么该如何合理地进行数据分区呢？

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import year,month, dayofmonth
from datetime import date,timedelta
from pyspark.sql.types import IntegerType,DateType,StringType,StructType,StructField

appName = "PySpark Partition Example"
 master = "local[8]"
```

```
# 创建 Spark Session
spark = SparkSession.builder.appName(appName).master(master).getOrCreate()
print(spark.version)

# 填充样本数据(100 条记录)
start_date = date(2019, 1, 1)
data = []

for i in range(0, 50):
    data.append({"Country":"CN", "Date":start_date + timedelta(days=i), "Amount":10+i})
    data.append({"Country":"AU", "Date":start_date + timedelta(days=i), "Amount":10+i})

schema = StructType([StructField('Country', StringType(), nullable=False),
                     StructField('Date', DateType(), nullable=False),
                     StructField('Amount', IntegerType(), nullable=False)])

df = spark.createDataFrame(data, schema)
df.show()
```

显示结果如下（部分）：

```
# 查看当前的分区数（默认分区数）
print(df.rdd.getNumPartitions())

# 将 data frame 写入到文件系统，一个分区将会对应一个文件
df.write.mode("overwrite").csv("/data/spark_demo/partitioner-out", header=True)
```

在 Spark 中可以使用两个函数来重新分区数据，coalesce 就是其中之一。

```
# 使用 coalesce 函数重分区
df = df.coalesce(1)
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("/data/spark_demo/partitioner-out", header=True)
```

如果指定的重分区数大于当前的分区数，则保持当前分区数不变，这是因为 coalesce 函数不涉及数据重组，它导致的是窄依赖。

重分区的另一种方法是 repartition。使用此函数时将发生数据重新洗牌（shuffle）。它的参数可以是分区数（整数），也可以是分区列。

下面是按整数分区（指定分区数）

```
df = df.repartition(4)
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("/data/spark_demo/partitioner-out", header=True)
```

Spark 将尝试将数据均匀地分布到每个分区。如果总分区数大于实际的记录数(或 RDD 大小)，有些分区将是空的，多个空的分区只会生成一个文件。在运行上述代码之后，数据将被重新分配到 4 个分区，并生成 4 个切分文件。

下面是按 Country 列分区：

```
df = df.repartition("Country")
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("/data/spark_demo/partitioner-out", header=True)
```

上面的脚本将创建 200 个分区(Spark 默认情况下创建 200 个分区)。然而，只有三个分片文件生成：

- 一个文件存储 CN 国家的数据。
- 另一个文件存储 AU 国家的数据。
- 另一个是空的。

类似地，我们也可以按 Date 列对数据进行分区（默认也是 200 个分区）：

```
df = df.repartition("Date")
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("/data/spark_demo/partitioner-out", header=True)
```

如果查看数据，可能会发现数据可能没有像预期的那样被正确分区，例如，有的分区文件包含两个国家和不同日期的数据，如下图所示。这是因为 Spark 默认使用散列分区作为分区函数（可以使用范围分区函数或自定义分区函数，但不在这里讲解）。

在现实世界中，可能会将数据划分为多个列。例如，我们可以实现如下分区策略:

```
/data
    /spark_demo
        /partitioner-out
            /year=2019
                /month=01
                    /day=01
                        /Country=CN
                            /part....csv
```

使用这种分区策略，我们可以根据日期和国家轻松检索数据。为了实现上述分区策略，我们需要派生一些新列(year、month、date)。

```
df = df.withColumn("Year", year("Date"))\
       .withColumn("month", month("Date"))\
       .withColumn("Day", dayofmonth("Date"))
df = df.repartition("Year", "Month", "Day", "Country")
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("/data/spark_demo/partitioner-out", header=True)
```

当查看保存的文件时，可能会发现所有的新列也被保存了，并且文件仍然混合了不同的子分区。为了改进这一点，我们需要将写分区 key 与重分区 key 匹配。

将重分区键与写分区键匹配

为了匹配分区键，我们只需要改变最后一行来添加一个 partitionBy 函数：

```
df = df.withColumn("Year", year("Date"))\
       .withColumn("month", month("Date"))\
       .withColumn("Day", dayofmonth("Date"))
df = df.repartition("Year", "Month", "Day", "Country")
print(df.rdd.getNumPartitions())
df.write.partitionBy("Year","Month","Day","Country")\
       .mode("overwrite")
       .csv("/data/spark_demo/partitioner-out", header=True)
```

打开这些文件，还会发现所有的分区列/键都被从序列化的数据文件中删除了。这样，存储成本也更低。使用分区数据，我们还可以轻松地将数据追加到新的子文件夹中，而不必对整个数据集进行操作。

读取分区数据

现在让我们用以下条件从分区文件中读取数据：

- Year= 2019
- Month=2
- Day=1
- Country=CN

```
df = spark.read.csv("/data/spark_demo/partitioner-out/Year=2019/Month=2/Day=1/Country=CN", header=True)
print(df.rdd.getNumPartitions())
df.show()
```

结果如下所示：

类似地，我们可以查询第二个月的所有数据

```
df = spark.read.csv("/data/spark_demo/partitioner-out/Year=2019/Month=2", header=True)
print(df.rdd.getNumPartitions())
df.show()
```

结果如下所示：

如果要查询所有 Country=CN 的数据呢？

可以使用通配符。分区发现中的所有文件格式都支持通配符。

```
df = spark.read.option("basePath","/data/spark_demo/partitioner-out")\
        .csv("/data/spark_demo/partitioner-out/Year=*/Month=*/Day=*/Country=CN", header=True)
print(df.rdd.getNumPartitions())
df.show()
```

结果如下所示：

可以在路径的任何部分使用通配符进行分区发现。例如，下面的代码查找 Country=AU 的第二个月的数据：

```
df = spark.read.option("basePath","/data/spark_demo/partitioner-out")\
        .csv("/data/spark_demo/partitioner-out/Year=*/Month=2/Day=*/Country=AU", header=True)
print(df.rdd.getNumPartitions())
df.show()
```

结果如下所示：

5.7 读写 Hive 表

Spark SQL 还支持读取和写入存储在 Apache Hive 中的数据。Spark 支持两种 SQL 方言：Spark 的

SQL 方言和 Hive 查询语言(HQL)。我们可以通过 Spark SQL 访问已经存在的 Hive 表并使用已经存在的、社区构建的 Hive UDFs。没有现有 Hive 部署的用户仍然可以启用 Hive 支持。

5.7.1 Spark SQL 的 Hive 配置

要通过 Spark SQL 访问 Hive 数据表，需要先进行以下配置。

1）配置 Hive 支持

.....

2）拷贝 JDBC 驱动

3）启动 Hive Metastore Server

.....

5.7.2 Spark SQL 读写 Hive 表

。。。。。。

5.7.3 分桶和排序

对于基于文件的输出，还可以进行分桶和排序。DataFrameWriter 提供有 bucketBy 和 sortBy 方法用于控制输出文件的目录结构。如果要对输出结果分桶存储的话，可以使用“write.bucketBy(<分桶数>,<分桶字段>)”方法。

修改上面示例代码，让 movies 结果集以 parquet 格式按电影名称（title 列）分桶存储。

执行以上代码，输出内容和存储数据结构如下图中所示：

有可能对单个表同时使用分区和分桶以及排序。将以上代码修改如下。

执行以上代码，输出结果和存储结构如下图所示：

提示：

（1）目前 bucketBy 需要和 saveAsTable()结合使用，而不能和 save()一起来用。

（2）目前只支持 sortBy(...)和 bucketBy(...)结合使用，并且排序列不应该是分区列的一部分。

5.7.2 Spark Hive ETL 实现

.....

5.8 性能调优

Spark 提供了许多配置来改进和调优 Spark SQL 工作负载的性能，这些配置可以通过编程方式完成，

也可以使用 Spark submit 在全局级别应用。对于某些工作负载，可以通过在内存中缓存数据或启用一些实验性选项来提高性能。

参考：

Spark 性能调优和最佳实践：https://sparkbyexamples.com/spark/spark-performance-tuning/

5.8.1 在内存中缓存数据

Spark SQL 可以通过调用 Spark.catalog.cachetable(“tableName”)或 datafame .cache()来使用内存中的列格式来缓存表。然后，Spark SQL 将只扫描所需的列，并自动调整压缩，以最小化内存使用和 GC 压力。可以调用 spark.catalog.uncacheTable(“tableName”)来从内存中删除该表。

可以使用 SparkSession 上的 setConf 方法或使用 SQL 运行 SET key=value 命令来配置内存缓存。

属性名	默认	含义
spark.sql.inMemoryColumnarStorage.compressed	true	当设置为true时，Spark SQL将根据数据的统计信息自动为每一列选择一个压缩编解码器。
spark.sql.inMemoryColumnarStorage.batchSize	10000	控制用于列缓存的批的大小。更大的批处理大小可以提高内存利用率和压缩，但是在缓存数据时存在OOM风险。

缓存时使用柱状格式

当从 Dataframe/SQL 缓存数据时，请使用内存中的列格式。当对列执行 Dataframe/SQL 操作时，Spark 只检索所需的列，这将减少数据检索和内存使用。通过将 spark.sql.inMemoryColumnarStorage.compressed 配置设置为 true，可以使 Spark 使用内存中的柱状存储。

```
spark.conf.set("spark.sql.inMemoryColumnarStorage.compressed", true)
```

Spark CBO（Cost-Based Optimizer，基于成本的优化器）

当使用多个 join 连接时，请使用基于成本的优化器，因为它可以基于表和列统计信息改进查询计划。默认情况下是启用的，如果是禁用的，可以通过设置 spark.sql.cbo.enabled 为 true 来启用它。

```
spark.conf.set("spark.sql.cbo.enabled", true)
```

注：在执行 Join 查询之前，需要运行 ANALYZE TABLE 命令，指出要联接的所有列。此命令为基于成本的优化器收集表和列的统计信息，以找出最佳查询计划。

```
ANALYZE TABLE table_name COMPUTE STATISTICS FOR COLUMNS col1,col2
```

注：关于 ANALYZE TABLE 命令，请参考官方文档：

https://spark.apache.org/docs/3.0.0-preview/sql-ref-syntax-aux-analyze-table.html

为 Shuffle 分区使用最优值

当执行触发数据 shuffle 的操作(如聚合和 join 连接)时，Spark 默认创建 200 个分区。这是因为配置属性 spark.sql.shuffle.partitions 默认设置为 200。之所以设置这个 200 默认值，是因为 Spark 不知道 shuffle 操作后要使用的最佳分区大小。大多数情况下，这个值会导致性能问题，因此，根据数据大小更改它。如果我们有非常庞大的数据，那么就需要有更大的分区数字，如果有较小的数据集，那么就需要较小的分区数字。

```
spark.conf.set("spark.sql.shuffle.partitions",30)                    // 默认值是 200
```

我们需要调优此值和其他值，直到达到性能基准。

当 join 数据可以装入内存时，使用 Broadcast Join

在 Spark 中可用的所有不同的 join 策略中，广播散列连接(broadcast hash join)提供了更好的性能。只有当一个连接表足够小，能够在广播阈值范围内装入内存时，才可以使用此策略。

如果数据很大时，可以使用下面的配置来增加该阈值大小：

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold",10485760)  // 默认是 100 MB
```

Spark 3.0 - 在 SQL 上使用 coalesce & repartition

在使用 Spark SQL 查询时，我们可以通过在查询中使用 COALESCE、REPARTITION 和 REPARTITION_BY_RANGE 来根据数据大小增加或减少分区。

```
SELECT /*+ COALESCE(3) */ * FROM EMP_TABLE
SELECT /*+ REPARTITION(3) */ * FROM EMP_TABLE
SELECT /*+ REPARTITION(c) */ * FROM EMP_TABLE
SELECT /*+ REPARTITION(3, dept_col) */ * FROM EMP_TABLE
SELECT /*+ REPARTITION_BY_RANGE(dept_col) */ * FROM EMP_TABLE
SELECT /*+ REPARTITION_BY_RANGE(3, dept_col) */ * FROM EMP_TABLE
```

Spark 3.0 - 启用“自适应查询执行”

自适应查询执行，请参考：https://sparkbyexamples.com/spark/spark-adaptive-query-execution/

自适应查询执行是 3.0 的一个特性，它通过在运行时使用每个阶段完成后收集的统计数据重新优化查询计划，从而提高查询性能。可以通过设置 spark.sql.adaptive.enabled 配置属性为 true 来启用它。

```
spark.conf.set("spark.sql.adaptive.enabled",true)
```

Spark 3.0 - 合并 Shuffle 后分区（Coalescing Post Shuffle Partitions）

在 Spark 3.0 中，在作业的每个阶段之后，Spark 通过查看完成阶段的指标来动态地确定分区的最佳数量。为了使用它，需要启用以下配置。

```
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled",true)
```

Spark 3.0 -优化倾斜连接（Optimizing Skew Join）

有时，我们可能会遇到分区中分布不均匀的数据，这称为数据倾斜。诸如 join 之类的操作在这些分区上执行得非常慢。通过启用 AQE, Spark 检查 stage 统计信息，确定是否存在 Skew 连接，并通过将较大的分区划分为较小的分区(与其他表/DataFrame 上的分区大小匹配)来优化它。

```
spark.conf.set("spark.sql.adaptive.skewJoin.enabled",true)
```

上面介绍了通过不同的配置，以提高 Spark SQL 查询和应用程序的性能。在具体应用进，需要调优这些配置的值以及执行器 CPU 内核和执行器内存，直到满足我们的需求为止。

5.8.2 合理的配置选项

还可以使用以下选项来调优查询执行的性能。在将来的版本中，随着自动执行更多的优化，这些选项可能会被弃用。

属性名	默认	含义
spark.sql.files.maxPartitionBytes	134217728 (128 MB)	在读取文件时，要塞进到单个分区中的最大字节数。
spark.sql.files.openCostInBytes	4194304 (4 MB)	打开一个文件的估计成本，通过同时扫描的字节数来衡量。这是在

		将多个文件放入一个分区时使用的。最好是过度估计，那么带有小文件的分区将比带有大文件的分区(这是优先计划的)更快。
spark.sql.broadcastTimeout	300	广播连接中广播等待时间的超时(秒)
spark.sql.autoBroadcastJoinThreshold	10485760 (10 MB)	配置将在执行join时广播到所有工作节点的表的最大大小(以字节为单位)。通过将此值设置为-1，可以禁用广播。
spark.sql.shuffle.partitions	200	在为连接或聚合重组(shuffling)数据时，配置要使用的分区数量。

5.8.3 广播 SQL 查询提示

BROADCAST 提示引导 Spark 在将每个指定表与另一个表或视图连接时广播它们。当 Spark 决定连接方法时，broadcast hash join (即 BHJ)是首选，即使统计数据高于 spark.sql.autoBroadcastJoinThreshold 配置。当指定联接的双方时，Spark 广播统计数据较低的一方。注意，Spark 并不保证总是选择 BHJ，因为并非所有情况(例如完全外部连接)都支持 BHJ。当选择广播嵌套循环连接时，我们仍然遵循提示。

```
import org.apache.spark.sql.functions.broadcast
broadcast(spark.table("src")).join(spark.table("records"), "key").show()
```

5.9 查询优化器

Catalyst 是 Spark SQL 查询优化器，用于获取查询计划，并将其转换为 Spark 可以运行的执行计划。它在确保用 DataFrame API 或 SQL 编写的数据处理逻辑高效、快速地运行方面发挥了重要作用。它的设计目的是最小化端到端查询响应时间，并具有可扩展性，这样 Spark 用户可以将用户代码注入优化器，以执行自定义优化。

当我们在 DataFrame/Dataset 上应用关系和函数转换时，Spark SQL 建立了一个语法树来表示我们的查询计划，称为逻辑计划。Spark 能够在逻辑计划上应用许多优化，还可以使用基于成本的模型在相同逻辑计划的多个物理计划之间进行选择。

5.7.1 窄转换和宽转换

我们将介绍 Spark 如何将 Dataset 的 Transformation 和 action 转换为执行模型。为了理解应用程序如何在集群上运行，一件重要事情是了解 Dataset transformation，它们分为两种类型，窄类型和宽类型，在解释执行模型之前，我们将首先讨论这两种类型。

窄转换和宽转换

回顾一下，transformation 将从现有 Dataset 创建一个新的 Dataset。在从现有 Dataset 创建新的 Dataset 时，narrow transformation 不必在分区之间移动数据。例如，在执行 filter 和 select 转换时，就是 narrow transformation：

```
// select 和 filter 是 narrow transformations
df.select($"carrier",$"origin", $"dest", $"depdelay", $"crsdephour")
  .filter($"carrier" === "AA")
  .show(2)
```

在一个称为流水线（pipelining）的过程中，可以对内存中的 Dataset 执行多个窄转换，从而使窄转换非常有效。

在创建新的 Dataset 时，wide transformation 会导致数据在分区之间移动，在一个被称为 shuffle 的过程中。通过 wide transformation shuffle，数据通过网络发送到其他节点并写到磁盘，从而导致网络和磁盘 I/O，并使 shuffle 成为一项昂贵的操作。例如，groupBy、agg、sortBy 和 orderBy 就是宽转换：

```
df.groupBy("carrier").count.show
```

5.7.2 Spark 执行模型

Spark 执行模型可以定义为三个阶段：创建逻辑计划，将其转换为物理计划，然后在集群上执行任务。Spark Catalyst 将用户写入的数据处理逻辑转换为逻辑计划，然后使用启发式方法对其进行优化，最后将逻辑计划转换为物理计划。最后根据物理计划生成代码。

在 Catalyst 中，针对逻辑执行计划和物理执行计划，分别提供了逻辑优化和物理优化。

- 逻辑优化：这包括优化器将筛选谓词下推到数据源的能力，以及执行跳过不相关数据的能力。例如，对于 Parquet 文件，可以跳过整个块，并且可以通过字典编码将字符串上的比较转换为开销更少的整数比较。
- 物理优化：这包括智能地在广播连接和 shuffle 连接之间选择连接以减少网络流量的能力，执行低级别的优化，例如消除昂贵的对象分配和减少虚拟函数调用。

Catalyst 优化器优化过程如下：

逻辑计划

逻辑计划是以操作符和表达式树的形式对用户数据处理逻辑进行内部表示。通过 DataFrames/dataset(或 SQL 查询)上的 transformation 转换构造逻辑计划，一开始逻辑计划是未解析的。Spark 优化器是多阶段的，在执行任何优化之前，它需要解析表达式的引用和类型。

Catalyst 解析和优化逻辑计划的过程如下：

-

下图显示了所有这些步骤：

Spark 2.2 中引入了基于成本的优化，使 Catalyst 能够更智能地根据正在处理的数据统计选择正确的 join 连接类型。基于成本的优化依赖于参与过滤器或连接条件的列的详细统计信息，这就是引入统计信息收集框架的原因。统计信息的示例包括基数、不同值的数量、最大/最小、平均/最大长度，等等。

物理计划

逻辑计划优化后，Spark 将生成物理计划。

.....。

5.7.4 Catalyst 实践

本节将展示如何使用 DataFrame 类的 explain 函数来显示逻辑和物理计划。要查看逻辑计划和物理计划，可以调用 explain(true)函数。否则，这个函数只显示物理计划。

在下面的示例中，首先以 Parquet 格式读取电影数据，然后根据 produced_year 进行过滤，然后增加一个名为 produced_decade 的列，并投影 movie_title 和 produced_decade 列，最后根据 produced_decade 来过滤行。这里的目标是证明 Catalyst 执行了谓词下推和过滤条件优化。

【示例】catalyst 实践示例：查看逻辑计划和物理计划。

执行以上代码，可以看到如下的内容：

如果仔细分析优化的逻辑计划，将看到它将两个过滤条件合并到一个过滤器中。物理计划表明，Catalyst 既下推了 produced_year 的过滤，也执行了 FileScan 这一步的投影修剪。

【示例】带有 groupBy 操作的 catalyst 实践示例。

执行上面的代码，可以看到如下的输出内容：

在上面的代码中，在 explain 之后，我们看到 moviesDS3 的物理计划由一个 FileScan、Filter、Project、HashAggregate、Exchange 和 HashAggregate 组成。

其中的 Exchange 是由 groupBy transformation 引起的 shuffle。Spark 在 Exchange 中 shuffle 数据之前对每个分区执行 hash 聚合。在 Exchange 之后，对之前的子聚合进行 hash 聚合。

【示例】带有缓存和 groupBy 操作的 catalyst 实践示例。

执行上面的代码，可以看到如下的输出内容：

注意，在这个 DAG 中，如果 moviesDS2 被缓存，Catalyst 将使用内存扫描而不是文件扫描。

5.7.5 Catalyst 实践 2

通过调用 explain(true)方法，可以看到一个 Dataset/DataFrame 的逻辑和物理计划；如果调用的 explain(false)方法，则只显示物理计划。

在下面的代码中，我们看到 df2 的 DAG 由一个 FileScan、一个在 depdelay 列上的 Filter 过滤器和一个 Project 投影(选择列)组成。

5.7.5 可视化 Spark 程序执行

Spark Web UI 接口对于调优任务是必不可少的。我们可以使用使用该接口来监视 Spark 应用程序的执行，显示的信息包括调度器阶段和任务的列表、RDD 大小和内存使用的摘要、环境信息以及关于正在运行的执行程序的信息。

可以使用以下 URL 在 web 浏览器中实时查看关于 Spark 作业的有用信息：http://<driver-node>:4040。

对于已经完成的 Spark 应用程序，可以使用 Spark 历史服务器在 web 浏览器中查看这些信息，这时使用的 URL 为：http:// <server-url >: 18080。更详细的参考在这里）

【示例】直观地研究 Spark SQL 应用程序执行。

。。。。。。

5.10 项目 Tungsten

Tungsten 是 Spark 项目的代码名，该项目对 Apache Spark 的执行引擎进行了更改，包括针对 Spark 所需操作类型调优的专用内存数据结构、改进的代码生成和专用的连接协议，重点是提高内存和 CPU 使用效率。

从 2015 年开始，Spark 的设计者注意到，Spark 的工作负载越来越多地受到 CPU 和内存的瓶颈，而不是 I/O 和网络通信。

从 2015 年开始，Spark 设计者注意到，因为硬件方面的进步（比如 10 Gbps 的网络连接和高速 SSD ），Spark 工作负载越来越多地受到 CPU 和内存的瓶颈，而不是 I/O 和网络通信。在 Spark 2.0 之前，大部分 CPU 周期都花在了无用的工作上，例如进行虚函数调用或将中间数据读写到 CPU 缓存或内存。项目 Tungsten 的创建就是为了提高 Spark 应用程序中内存和 CPU 的使用效率。它基于现代编译器和大规模并行处理(MPP)技术的思想，并将性能提升到接近现代硬件的极限。

在 Tungsten 项目中采用了以下三个措施：

- 通过使用堆外管理技术来显式地管理内存，以消除 JVM 对象模型的开销并最小化垃圾收集。
- 使用智能缓存感知（cache-aware）算法和数据结构来利用内存层次结构。
- 通过将多个操作符组合到单个 Java 函数中，使用全阶段的代码生成来最小化虚函数调用。

此外，集成在 Spark 2.0 中的第二代 Tungsten 执行引擎，还在以下方面做了改进：

- 内存管理的改进：集中在以紧凑的二进制格式存储 Java 对象以减少 GC 开销，密集的内存数据格式以减少溢出效应(例如，Parquet 格式)，和对于理解数据类型(在 DataFrames、Datasets 和 SQL)的操作符直接工作在内存二进制格式上而不是序列化/反序列化等等。内存数据的柱状布局避免了不必要的 I/O，并加速了现代 CPU 和 GPU 上的分析处理性能。
- 智能缓存感知：为了提高数据处理的速度，通过更有效地使用 L1/ L2/L3 CPU 缓存，Spark 算法和数据结构利用内存层次与缓存感知计算。
- Catalyst 优化器处理：分析、逻辑优化、物理规划和代码生成，以将部分查询编译成 Java 字节码。Catalyst 现在支持基于规则和基于成本的优化。
- Spark SQL“全阶段 Java 代码生成”：通过为 SQL 查询中的操作符集(如果可能的话)生成单个字节码优化函数(函数折叠)，从而优化了 CPU 使用。该技术消除了虚拟函数调用，并使用 CPU 寄存器存储中间数据，这进而大大提高了 CPU 效率和运行时性能。
- 向量化技术：这利用了现代 CPU 设计，允许 CPU 对向量进行操作，向量是来自多个记录的列值的数组。在向量化中，引擎以列格式将多行进行批处理，每个操作符在批处理中迭代数据。但是，它仍然需要将中间数据放在内存中，而不是将它们保存在 CPU 寄存器中。因此，只在不可能完成全阶段代码生成时才使用向量化。

下面的例子显示在 DataFrame 中过滤和求和整数的物理计划，我们通过检查物理计划来了解全阶段代码生成计划。在 explain()输出中，当一个操作符被标记为星号*时，就意味着该操作的全阶段代码生成已经启用。

【示例】了解全阶段代码生成计划。

```
spark.range(1000)
      .filter("id > 100")
      .selectExpr("sum(id)")
      .explain())
```

输出的物理计划如下：

```
== Physical Plan ==
*(2) HashAggregate(keys=[], functions=[sum(id#264L)])
+- Exchange SinglePartition, ENSURE_REQUIREMENTS, [id=#442]
   +- *(1) HashAggregate(keys=[], functions=[partial_sum(id#264L)])
      +- *(1) Filter (id#264L > 100)
         +- *(1) Range (0, 1000, step=1, splits=2)
```

全阶段代码生成将过滤和汇总整数的逻辑组合成一个单一的 Java 函数。

5.11 Spark SQL 编程案例

本节通过几个案例的学习，掌握使用 Spark SQL 进行大数据分析的复杂用法方法。

5.11.1 业务分析示例

【示例】业务分析示例。使用 nw 数据集，回答以下问题：

- 每个客户下了多少订单?
- 每个国家的订单有多少?
- 每月/年有多少订单?
- 每个客户的年销售总额是多少?
- 客户每年的平均订单是多少?

实现代码如下所示：

```
```

5.11.3 MovieLens 数据集分析

本节使用 Spark SQL 实现对电影数据集进行分析。在这里我们使用推荐领域一个著名的开放测试数据集 movielens。MovieLens 数据集包括电影元数据信息和用户属性信息。我们将使用其中的 users.dat 和 ratings.dat 两个数据集。

【例】使用 Spark Dataset API 统计看过“Lord of the Rings,The(1978)”的用户的年龄和性别分布（提示该影片的 id 是“2116”）。实现过程和代码如下。

1) 定义两个类型 case class 类，分别定义用户和评分的 schema。

```
case class User(userID:Long, gender:String, age:Integer, occupation:String, zipcode:String)
case class Rating(userID:Long, movieID:Long, rating:Integer, timestamp:Long)
```

2) 读取用户数据集 users.dat，并注册为临时表 users。

```
    import org.apache.spark.sql.functions._

    // 创建 SparkSession 的实例
```

```
val spark = SparkSession.builder()
  .master("local[*]")
  .appName("Spark Basic Example")
  .getOrCreate()
val sc = spark.sparkContext

// 定义文件路径
val usersFile = "src/main/resources/ml-1m/users.dat"

// 获得 RDD
val rawUserRDD = sc.textFile(usersFile)

// 支持 RDD 到 DataFrame 的隐式转换
import spark.implicits._

// 对 RDD 进行转换操作，最后转为 DataFrame
val userDF = rawUserRDD
  .map(_.split("::"))
  .map(x=>User(x(0).toLong, x(1), x(2).toInt, x(3), x(4)))
  .toDF()

// 查看用户数据
userDF.printSchema()
userDF.show(5)
```

输出结果如下所示：

```

```

3) 读取评分数据集 ratings.dat，并注册成临时表 ratings

```
// 定义文件路径
val ratingsFile = "src/main/resources/ml-1m/ratings.dat"

// 生成 RDD
val rawRatinngRDD = sc.textFile(ratingsFile)

// 对 RDD 进行转换，最后转为 DataFrame
val ratingDF = rawRatinngRDD
    .map(_.split("::"))
    .map(x=>Rating(x(0).toLong, x(1).toLong,x(2).toInt,x(3).toLong))
    .toDF

// 查看
ratingDF.printSchema()
ratingDF.show(5)
```

输出结果如下所示：

```

```

4) 将两个 DataFrame 注册为临时表，对应的表名分别为“users”和“ratings”。

```
userDF.createOrReplaceTempView("users")
ratingDF.createOrReplaceTempView("ratings")
```

5) 通过 SQL 处理临时表 users 和 ratings 中的数据，并输出最终结果。为了简单起见，避免三表连接操作，这里直接使用了 movieID。

```
val MOVIE_ID = "2116"
val sqlStr = s"""select age,gender,count(*) as total_peoples
                  from users as u join ratings as r on u.userid=r.userid
                  where movieid=${MOVIE_ID} group by gender,age"""
val resultDF = spark.sql(sqlStr)

// 显示 resultDF 的内容
resultDF.show()
```

输出结果如下所示：

6) 以交叉表的形式统计不同年龄不同性别用户数。

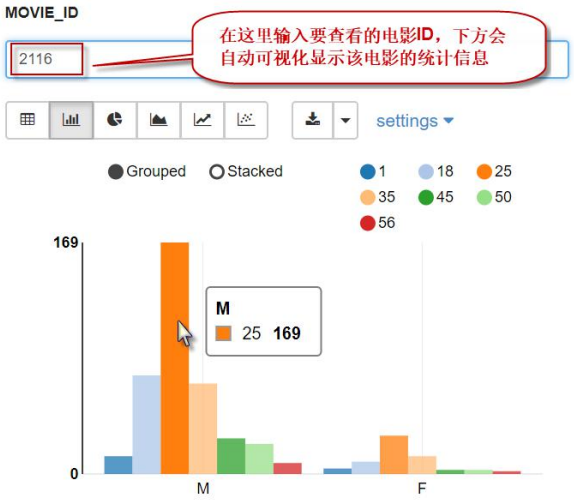
```
import org.apache.spark.sql.functions._
resultDF
    .groupBy("age")
    .pivot("gender")
    .agg(sum("total_peoples").as("cnt"))
    .show()
```

输出结果如下所示：

7) 在 zeppelin 中，支持查询结果的可视化显示。在 zeppelin 的单元格中，执行以下语句，可视化显示数据(注：第一行必须输入%sql)。

```
%sql
select age,gender,count(*) as total_peoples
from users as u join ratings as r
        on u.userid=r.userid
where movieid=${MOVIE_ID=2116}
group by gender,age
```

输出结果如下所示：



第 6 章　Spark Streaming

除了批数据处理之外，流数据处理已经成为任何想要利用实时数据的价值来提高其竞争优势或改善用户体验的企业的必备能力。

在很多领域，如股市走向分析、气象数据测控、网站用户行为分析等，由于数据产生快，实时性强，数据量大，所以很难统一采集并入库存储后再做处理，这便导致传统的数据处理架构不能满足需要。流计算的出现，就是为了更好地解决这类数据在处理过程中遇到的问题。与传统架构不同，流计算模型在数据流动的过程中实时地进行捕捉和处理，并根据业务需求对数据进行计算分析，最终把结果保存或者分发给需要的组件。

随着物联网的爆发，互联设备通过传感器、摄像头、加速度计、激光雷达和深度传感器收集越来越多的信息。从制造业到汽车、医疗技术、能源、公用事业和可穿戴技术等各个行业都存在联网产品。在人工智能和 5G 融合的帮助下，被收集的数据量只会不断扩大。据估计，一辆完全自动驾驶的汽车将包含超过 60 个微处理器和传感器，每年产生超过 300TB 的数据。或者，反过来说，在一小时的长途旅行中，联网车辆之间将发送多达 25GB 的信息(相当于 100 个小时的视频)。

有了这些海量数据，捕获、聚合和分析数据就成了一个挑战。并非所有数据都有用，但自动驾驶汽车、有毒气体监测、医疗保健和安全设备等对时间敏感的数据存在滞后风险。数据在瞬间的延迟(来自于，例如，一辆汽车无法识别道路上的行人，或者一个故障的胰岛素泵)进入云端并返回到设备可能是灾难性的或致命的。

Apache Spark 的统一数据处理平台受欢迎的一个因素是能够执行流数据处理和批处理数据处理。

6.1 Spark DStream

第一代 Spark 流式处理引擎是在 2012 年引入的，这个引擎的主要编程抽象称为离散化流，即 DStream。它的工作方式是使用微批处理模型将传入的数据流分成批处理，然后由 Spark 批处理引擎处理。当 RDD 是主要的编程抽象模型时，这是很有意义的。每批都由 RDD 在内部表示。下图显示了 DStream 的工作方式。

一个 DStream 可以从像 Kafka、AWS Kinesis、一个文件、或者一个 socket 套接字这样的来源的输入数据流中创建。在创建 DStream 时，需要的关键信息之一是批间隔，这个批间隔可以在是秒或几毫秒。有了 DStream，就可以在数据输入流上应用一个高级的数据处理函数，如 map、filter、reduce 或 reduceByKey。此外，还可以执行窗口操作，比如通过提供窗口长度和滑动间隔来减少和计算一个固定/滚动或滑动窗口。一个重要的注意事项是，窗口长度和滑动间隔必须是批处理间隔的倍数。例如，如果批间隔是 3 秒，并且使用了固定/滚动间隔，那么窗口长度和滑动间隔可以是 6 秒。在 DStream 中支持跨批次数据执行计算时保持任意状态，但这是一个手动过程，而且有点麻烦。一个 DStream 还可以与另一个 DStream 或代表静态数据的 RDD 连接（join）起来。在所有处理逻辑完成之后，可以使用 DStream 将数据写到外部系统，如数据库、文件系统或 HDFS。

下面是一个小小的单词记数 Spark DStream 应用程序。通过这个程序我们可以了解一个典型的 Spark

DStream 应用程序。下面的示例包含了单词计数应用程序的代码：

```
import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}

val ssc = new StreamingContext(sc, Seconds(1))

val host = "localhost"
val port = 9999

val lines = ssc.socketTextStream(host, port, StorageLevel.MEMORY_AND_DISK_SER)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()

ssc.start()
ssc.awaitTermination()
```

在组装一个 DStream 应用程序时，有几个重要的步骤。

- DStream 应用程序的入口点是 StreamingContext，其中一个必需的输入是批间隔，它定义了一个时间持续时间，Spark 用来将输入的数据输入 RDD 进行处理。它也代表了一个触发点，在该触发点 Spark 应该执行流应用程序计算逻辑。例如，如果批处理间隔是 3 秒，那么就会触发所有到达 3 秒间隔内的数据；在该间隔之后，它将把这批数据转换为 RDD，并根据提供的处理逻辑对其进行处理。
- 一旦创建了 StreamingContext，下一步就是通过定义输入源来创建实例 DStream。前面的例子将输入源定义为读取文本行的 socket 套接字。
- 接下来，将为新创建的 DStream 提供处理逻辑。前面例子中的处理逻辑并不复杂。一旦一个系列的 RDD 在一秒后可用，那么 Spark 就会执行将每一行代码分割成单词的逻辑，将每个单词转换成单词的元组和 1 的计数，最后总结同一个单词的计数。
- 最后，计数在控制台上打印出来。
- 流应用程序是一个长期运行的应用程序；因此，它需要一个信号来启动接收和处理传入的数据流的任务。这个信号是通过调用 StreamingContext 的 start()函数来实现的，这通常是在文件的末尾完成的。awaitTermination()函数用于等待流应用程序的执行停止，以及在流应用程序运行时防止驱动程序退出的机制。在一个典型的程序中，一旦执行最后一行代码，它就会退出。然而，一个长时间运行的流应用程序需要在启动时继续运行，并且只有当显式地停止它时才会结束。

注：

要运行该程序，你需要首先运行一个 Netcat 服务器（在一个单独的终端窗口）：$ nc -lk 9999

然后再另一个终端窗口，运行流程序代码：$ bin/run-example NetworkWordCount

6.2 Spark 流处理示例

对于实时数据处理，Spark 使用“微批处理”。这意味着 Spark 流会将特定时间段内的数据块打包成 RDD。如下图所示：

6.2.1 编写 Spark Streaming 程序

下面通过一个示例"为证券公司建立仪表盘程序"，来说明编写 Spark 流应用程序的步骤。在这个示例中，数据源是证券公司实时发布的市场交易数据，比如证券、股票等的买卖。我们现在需要构建一个实时仪表板应用程序，用来实现回答以下问题：

- 计算每秒钟的销售和购买订单数量；
- 根据购买或出售的总金额来计算前 5 个客户；
- 找出过去一个小时内前 5 个交易量最多的股票。

数据集说明：

我们准备了一个文件 orders.txt，包含 500,000 行数据，每行代表一个买/卖订单。每一行包含如下用逗号分隔的元素：

- Order 时间戳 — 格式为 yyyy-mm-dd hh:MM:ss
- Order ID — 订单 ID，这是连续递增的整数
- Client ID — 客户 ID，这是从 1 到 100 范围内的整数
- Stock 代码 — 共 80 个股票代码
- 股票买卖的数量 — 从 1 到 1000
- 股票买卖的价格 — 从 1 到 100
- 字符 B 或 S — 代表一个订单的买(B)/卖(S)事件

以可使用如下命令查看文件的前 5 行内容：

```
$ head -n 5 ~/spark_demo/streaming/orders.txt
```

内容如下所示：

```
2016-03-22 20:25:28,1,80,EPE,710,51.00,B
2016-03-22 20:25:28,2,70,NFLX,158,8.00,B
2016-03-22 20:25:28,3,53,VALE,284,5.00,B
2016-03-22 20:25:28,4,14,SRPT,183,34.00,B
2016-03-22 20:25:28,5,62,BP,241,36.00,S
```

以上这些格式的数据就是我们要处理的流程序的数据。

在这个示例中，我们采用文件数据源。采用的思路和步骤如下：

1）编写一个名为 splitAndSend.sh 的脚本文件，它会将 orders.txt 这个股票交易订单文件拆分为 50 个小文件，每个小文件 10000 行；

2）然后，运行该脚本，它会周期性地（每 3 秒钟）将这些小文件移动到一个指定的 HDFS 目录；

3）我们编写的 Spark 流处理程序会监视这些指定的 HDFS 目录。一旦发现有新的数据文件，就实时进行处理，并将结果写出到 HDFS 中。

脚本 splitAndSend.sh 的代码如下：

```
```

暂时不需要启动该脚本，稍后我们会用到。

接下来实现我们的 Spark 流处理程序。

1、计算每秒钟的销售和购买订单数量

为了简单起见，我们从一个 HDFS 文件读取数据，并将结果写回到 HDFS。

确保 executor 的可用核数不少于 2，因为每个 Spark Streaming receiver 都必须使用一个 core（技术上，它是一个线程）来读取传入的数据，并且至少需要另一个 core 来执行程序的计算。

1）启动 Spark Shell，并指定 core 数：

```
$ spark-shell --master local[4]
```

2）创建流上下文环境：

```
import org.apache.spark._
import org.apache.spark.streaming._

val ssc = new StreamingContext(sc, Seconds(5))
```

在构造 StreamingContext 时，第二个参数为 Duration 对象，除了可以使用 Seconds 外，还可以使用 Milliseconds 和 Minutes 对象来指定一个周期。

上面代码中的 StreamingContext 构造器重用了一个已经存在的 SparkContext，但是 SparkStreaming 还能从一个新的 SparkContext 开始，如果给定一个 Spark 配置对象的话：

```
val conf = new SparkConf().setMaster("local[4]").setAppName("App name")
val ssc = new StreamingContext(conf, Seconds(5))
```

注：以上两行代码可以在单独的应用程序中使用，但是不能在 Spark shell 中使用，因为在同一个 JVM 中不能实例化两个 Spark contexts。

3）创建一个离散流

StreamingContext 提供了 textFileStream 方法，监控一个指定的目录(任何 Hadoop 兼容的目录，如 HDFS、S3、ClusterFS 和本地目录)，并读取该目录下新创建的文件。该方法只有一个参数：被监控的目录的名称。但是只对流程序开始后新拷贝到该目录下的文件进行处理。

下面的代码指定要监控的目录：

```
val inputPath = "/data/spark_demo/streaming/orders-input"
val fileStream = ssc.textFileStream(inputPath)
```

4）使用离散流

构造 DStream，并对其进行转换计算。

```
// 导入相关的包
import java.sql.Timestamp

// case class 类
case class Order(time: java.sql.Timestamp,
                 orderId:Long,
                 clientId:Long,
                 symbol:String,
                 amount:Int,
                 price:Double,
                 buy:Boolean)
```

订单 DStream 的每个 RDD 现在都包含 Order 对象。

包含两个元素元组的 DStreams 会自动地转换为 PairDStreamFunctions 对象。然后，就可以使用处理 pair RDD 的方法了，如 combineByKey、reduceByKey、flatMapValues、各种 join 等。如果将 orders 映射到包含订单类型作为 key 并且计数为 value 的元组，那么就可以使用 reduceByKey。

例如，计算每个订单类型（购买或出售）的次数，代码如下所示：

```
val numPerType = orders.map(o => (o.buy, 1L)).reduceByKey((c1,c2) => c1 + c2)
```

```
numPerType.print()
```

5）将结果保存到文件

使用 DStream 的 saveAsTextFiles 方法保存计算结果到一个文件中。每个 mini-batch RDD 被保存到一个名为<your_prefix>-<time_in_milliseconds>.<your_suffix>, 或<your_prefix>-<time_in_milliseconds>的文件夹中。这意味着，每 3 秒钟(本例中)，就有一个新的目录被创建。每个目录中，对应 RDD 的每个分区，包含一个对应的文件，名为 part-xxxxx，其中 xxxxx 是该分区号。因为在本例中，我们可以肯定每个 RDD 最多包含两个元素，所以可以对 DStream 重分区为只有一个分区，然后再保存到文件中(一个目录中就只有一个文件了)。代码如下：

```
val resultPath = "/data/spark_demo/streaming/orders-output/result"
numPerType.repartition(1).saveAsTextFiles(resultPath, "txt")
```

6）启动和停止流计算

通过下面的命令开始流计算：

```
ssc.start()
```

如果是在独立的应用程序中，还需要加上下面这句，否则驱动程序的主线程会退出：

```
ssc.awaitTermination()
```

7）新打开另一个终端窗口，在这个新的终端窗口中，执行以下代码（执行 splitAndSend.sh 脚本）：

```
$ cd ~/data/spark_demo/streaming/
$ hdfs dfs -mkdir /tmp/streaming
$ ./splitAndSend.sh /data/spark_demo/streaming/orders-input
```

如果流的输入目录是本地，需要在命令后加上 local 参数：

```
$ ./splitAndSend.sh /data/spark_demo/streaming/orders-input local
```

8）接下来，可以等待所有的文件都被处理完，或者也可以马上从 shell 停止运算流：

```
ssc.stop(false)
```

参数 false 告诉 streaming context 不要停止该 Spark context。不能重新启动已经停止的 streaming context，但是可以重用已经存在的 Spark context 来创建一个新的 streaming context。

9）查看最后结果：

```
$ hdfs dfs -cat /data/spark_demo/streaming/orders-output/result-*/*
```

计算结果如下（部分计算结果）：

```
-------------------------------------------
Time: 1557107545000 ms
-------------------------------------------
(false,5064)
(true,4936)

-------------------------------------------
Time: 1557107550000 ms
-------------------------------------------
(false,4992)
(true,5008)
```

部分计算结果

2、根据购买或出售的总金额来计算前 5 个客户

这是一个 Top N 问题。要根据购买或出售的总金额来计算前五名客户，就必须跟踪每个客户购买或出售的总金额。因此，随着时间的推移，流应用程序需要保存之前的计算状态。

换句话说，我们必须跟踪一个持续时间和不同小批量的状态。这个概念如下图所示：新数据定期以小批量的速度出现，每个 DStream 都是一个处理数据并产生结果的程序。通过使用 Spark Streaming 方法来更新状态，DStreams 可以将来自状态的持久数据与当前 mini-batch 的新数据结合起来。结果是更强大的流程序。

Spark Streaming 提供了两种执行状态计算的方法：updateStateByKey 和 mapWithState。这两种方法都可以从 PairDStreamFunctions 访问；换句话说，它们只适用于包含 key-value 元组的 DStream。因此，使用这些方法之前，必须创建这样的 DStream。

查看 API 文档，可以看到 updateStateByKey 函数定义如下：

```
updateStateByKey[S](updateFunc: (Seq[V], Option[S]) ⇒ Option[S])
```

该函数返回一个新的“state”DStream，其中每个 key 的状态通过对该 key 的前一个状态和新值应用给定的函数来更新。在每一批处理中，即使没有新值，也会为每个状态调用 updateFunc。使用哈希分区来生成具有 Spark 默认分区数的 RDDs。如果 updateFunc 这个状态更新函数返回 None，那么相应的状态 key-value 对将被消除。

下面的代码在前面已经计算出的 orders 基础上继续，根据买卖股票的总金额来计算出前 5 个客户。

3、同时输出卖出/买入多少手以及交易额居前 5 年客户 ID

为了在每个批周期内只输出两个 DStream 的结果(top 5 clients 和买/卖数量)一次，必须首先合并它们到一个 DStream。两个 DStream 可以使用各种 join 方法，或 cogroup 方法，或 union 方法，按 key 进行合并。

要合并两个 DStreams，它们的元素必须是相同的类型。我们将把 top5Clients 和 numPerType 这两个 DStreams 的元素转换为元组，这个元组的第一个元素是 key，代表输出的结果类型，用来描述这条数据显示的是 BUYS（买入）或 SELLS（卖出）或 TOP5CLIENTS（Top 5 客户的 ID），第二个元素是字符串列表。形式如下所示：

```
(卖出,List(4980))
(买入,List(5020))
(TOP5CLIENTS,List(34, 21, 92, 94, 69))
```

实现代码如下：

```
val buySellList = numPerType.map(t =>
        if(t._1) ("买入", List(t._2.toString))
        else ("卖出", List(t._2.toString))
    )

// glom()返回一个新的 DStream，其中通过对 DStream 的每个 RDD 应用 glom()生成每个 RDD。
// 对 RDD 应用 glom()可以将每个分区中的所有元素合并到一个数组中。
val top5clList = top5clients.repartition(1).map(x => x._1.toString).glom().map(arr => ("TOP5CLIENTS", arr.toList))

// 然后，将两个 DStream 合并在一起
val finalStream = buySellList.union(top5clList)
```

```
// 保存组合后的 DStream：
finalStream.repartition(1).saveAsTextFiles("/data/spark_demo/streaming/orders-output/result2", "txt")

// 在启动之前，先设置检查点目录
sc.setCheckpointDir("/tmp/streaming/checkpoint")
```

完整代码如下：

```
```

启动了 streaming context 后，另打开一个终端窗口，在这个新的终端窗口执行 splitAndSend.sh 脚本。最后生成的结果类似下面这样：

```
......
```

等待所有的文件都被处理完后，从 shell 停止运算流：

```
ssc.stop(false)
```

mapWithState 方法是从 Spark 1.6 引入的，比 updateStateByKey 要新。它带来了一些性能改进，可以维护的 key 的数量是 updateStateByKey 的 10 倍，而且它的速度是 updateStateKey 的 6 倍（原因是在没有新 key 到达的情况下避免处理）。该方法的签名如下：

```
mapWithState[StateType, MappedType](spec: StateSpec[K, V, StateType, MappedType])
```

这个方法通过对流的每个 key-value 元素应用一个函数来返回一个 MapWithStateDStream，同时维护每个唯一 key 的一些状态数据。该转换的映射函数和其他规范(例如，分区器、超时、初始状态数据等)可以使用 StateSpec 类指定。

对于流中接收到的每个 key-value 元素，都将调用传递给 StateSpec（以及随后的 mapWithState）的函数，该函数的输入参数为新收到的 key 和 value，以及每个 key 已经存在的状态。状态数据可以作为映射函数中 state 类型的参数访问。

所提供的函数接收到的 State 对象持有 key 的状态，并且有几个有用的方法来操纵它：

- exists—如果该状态被定义，返回 true
- get—获得该状态值
- remove—为一个 key 删除该状态
- update—为一个 key 更新或设置新状态值

mapWithState 的使用示例如下：

```
// 维护一个整数状态并返回一个字符串的映射函数
def mappingFunction(key: String, value: Option[Int], state: State[Int]): Option[String] = {
  // 使用 state.exists(), state.get(), state.update()  和  state.remove()来维护状态，并返回必要的字符串
  // ......
}

val spec = StateSpec.function(mappingFunction).numPartitions(10)

val mapWithStateDStream = keyValueDStream.mapWithState[StateType, MappedType](spec)
```

例如，下面的函数使用 mapWithState，将获得与之前的 updateStateByKey 相同的 amountState DStream。它接收的前两个参数是当前流元素的客户 ID（key）和其交易金额（value），第三个参数是该客户 ID 之前已经存在的状态，即之前存储的交易金额。

```
val updateAmountState = (clientId:Long, amount:Option[Double], state:State[Double]) => {
```

```
    // 获取新传入的交易金额;如果是第一次遇到这个 key，则为 0
    val total = amount.getOrElse(0.toDouble)
    // 如果有已经的状态（以前的交易金额）
    if(state.exists()){
        // 累加交易金额
        total += state.get()
    }
    // 用新值更新状态
    state.update(total)
    // 返回带有客户 ID 和新 state 值的 tuple
    Some((clientId, total))
}
```

像下面这样使用这个函数：

```
val amountState = amountPerClient.mapWithState(StateSpec.function(updateAmountState)).stateSnapshots()
```

如果没有最后一个 stateSnapshots 方法，就会得到一个带有客户 ID 和交易总金额的 DStream，但是只针对那些在当前的小批量中订单到达的客户。stateSnapshots 给了一个带有所有状态（所有客户）的 DStream。

4、使用窗口操作进行时间限制的计算

继续前面的例子。还有最后一个任务要完成：每 5 秒报告一次过去一个小时内前 5 个交易量最多的股票。这与之前的任务不同，因为它是有时间限制的。在 Spark Streaming 中，这种类型的计算是通过“窗口操作”完成的。下图是一个滑动窗口的示意图：

在我们的任务中，窗口的持续时间是一个小时。但是滑动周期与 mini-batch 持续时间相同(5 秒)，因为我们想要每 5 秒报告一次交易量排名前五的股票。

要创建一个窗口的 DStream，可以使用 reduceByKeyAndWindow 这个窗口方法。

```
reduceByKeyAndWindow(reduceFunc: (V, V) ⇒ V, windowDuration: Duration): DStream[(K, V)]
```

在这个方法中需要指定 reduce 函数以及窗口持续时间（也可以指定滑动周期，如果它与 mini-batch 周期不同的话）。所以要计算每个股票每个窗口的交易金额，可以使用如下的代码（放在 finalStream 变量初始化之前）：

```

```

启动了 streaming context 后，另打开一个终端窗口，在这个新的终端窗口执行 splitAndSend.sh 脚本。最后生成的结果类似下面这样：

```
...
```

等待所有的文件都被处理完后，从 shell 停止运算流：

```
ssc.stop(false)
```

完整代码如下所示：

```

```

window 方法并不是唯一可用的窗口操作。还有很多其他方法，其中有些适用于普通的 DStreams，而另一些则仅用于 pair DStreams(byKey 函数)。下面列出了这些方法：

窗口操作	描述
window(winDur, [slideDurl])	返回一个新的DStream，其中每个RDD包含在该DStream上的一个滑动时间窗

	口中看到的所有元素。slideDurl默认等于微批处理持续时间。
countByWindow(winDur, slideDur)	返回一个新的DStream，其中每个RDD都有一个单独的元素，该元素是通过计算该DStream上滑动窗口中的元素数量生成的。哈希分区用于生成具有Spark默认分区数的RDDs。
countByValueAndWindow(winDur, slideDur, [numParts])	返回一个新的DStream，其中每个RDD在该DStream上的一个滑动窗口中包含RDDs中不同元素的计数。哈希分区用于生成带有numpartition分区的RDDs(如果没有指定numpartition，则使用Spark的默认分区数)。
reduceByWindow(reduceFunc, winDur, slideDur)	返回一个新的DStream，其中每个RDD都有一个单独的元素，该元素是通过reduce该DStream上滑动窗口中的所有元素生成的。
reduceByWindow(reduceFunc, invReduceFunc, winDur, slideDur)	返回一个新的DStream，其中每个RDD都有一个单独的元素，该元素是通过reduce该DStream上滑动窗口中的所有元素生成的。然而，reduce是增量地使用旧窗口的reduce后的值: 1) reduce输入窗口的新值(例如，添加新计数) 2) “反向减少（inverse reduce）”窗口中遗留的旧值(例如，减去旧计数)。这比没有“inverse reduce”函数的reduceByWindow更有效。然而，它只适用于“可逆约简函数”("invertible reduce functions")。
groupByKeyAndWindow(winDur, [slideDur], [numParts/partitioner]	通过在该DStream上的滑动窗口上应用groupByKey创建一个新的DStream。类似于DStream.groupByKey()，但它应用于一个滑动窗口。还可以指定分区的数量或使用的分区器。 只应用于Pair DStream。
reduceByKeyAndWindow(reduceFunc, winDur, [slideDur], [numParts/partitioner])	通过在滑动窗口上应用reduceByKey返回一个新的DStream。与DStream.reduceByKey()类似，但应用于一个滑动窗口。还可以指定分区的数量或使用的分区器。 只应用于Pair DStream。
reduceByKeyAndWindow(reduceFunc, invReduceFunc, winDur, [slideDur], [numParts], [filterFunc])	通过在滑动窗口上应用增量reduceByKey返回一个新的DStream。使用旧窗口的约简值计算新窗口的约简值: 1）reduce输入窗口的新值(例如，添加新计数) 2）“反向减少（inverse reduce）”窗口中遗留的旧值(例如，减去旧计数) 这比没有“inverse reduce”函数的reduceByKeyAndWindow更有效。然而，它只适用于“可逆约简函数”。哈希分区用于生成具有Spark默认分区数的RDDs。

大家可能会注意到，替代使用 reduceByKeyAndWindow 方法，在之前的示例中，我们还可以先使用 window 方法，然后再使用 reduceByKey 方法。

6.2.2 使用外部数据源-Kafka

Kafka 通常用于构建实时流数据管道，以可靠地在系统之间移动数据，还用于转换和响应数据流。Kafka 作为集群在一个或多个服务器上运行。Kafka 的一些关键概念描述如下：

- Topic：消息发布到的类别或流名称的高级抽象。主题可以有 0、1 或多个消费者，这些消费者订阅发布到该主题的消息。用户为每个新的消息类别定义一个新主题；
- Producers：向主题发布消息的客户端；
- Consumers：使用来自主题的消息的客户端；
- Brokers：复制和持久化消息数据的一个或多个服务器。

此外，生产者和消费者可以同时对多个主题进行读写。每个 Kafka 主题都是分区的，写入每个分区的消息都是顺序的。分区中的消息具有一个偏移量，用来惟一标识分区内的每个消息。（Kafka 官网）

主题的分区是分布式的，每个 Broker 处理对分区共享的请求。每个分区在 Brokers（数量可配置的）之间复制。Kafka 集群在一段可配置的时间内保留所有已发布的消息。Apache Kafka 使用 Apache ZooKeeper 作为其分布式进程的协调服务。

Kafka 的数据源可能是在生产型流应用程序中最常用的数据源。为了有效地处理这个数据源，我们

需要一定的 Kafka 使用基本知识。

官方提供了 Spark 连接器用于集成 Kafka。在这一节，我们使用一个 shell 脚本发送股票交易数据给 Kafka topic。Spark Streaming job 将从这个 topic 读取交易数据并将计算出的结果写到另一个 Kafka topic。然后我们将使用 Kafka 的 kafka-console-consumer.sh 脚本来接收并显示结果。处理流程如下图所示：

1）设置 Kafka

要设置 Kafka，首先需要下载它：http://kafka.apache.org/downloads.html。注意，要选择与你的 Spark 版本相兼容的版本。(查看版本兼容性)

然后，解压缩下载的 Kafka 压缩包到~/bigdata/目录下：

```
$ cd ~/bigdata
$ tar -xvfz kafka_2.11-0.10.1.0.tgz
```

Kafka 依赖于 Apache ZooKeeper，所以在启动 Kafka 之前，要先启动它：

```
$ cd ~/bigdata/kafka_2.11-0.10.1.0
$ ./bin/zookeeper-server-start.sh config/zookeeper.properties &
```

这将在 2181 端口启动 ZooKeeper 进程，并让 ZooKeeper 在后台工作。接下来，启动 Kafka 服务器：

```
$ ./bin/kafka-server-start.sh config/server.properties &
```

最后，需要创建用于发送股票买卖数据和计算结果数据的主题(topic)：

```
$ ./bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic orders
$ ./bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic metrics
```

查看已有的主题：

```
$ ./bin/kafka-topics.sh --list --zookeeper localhost:2181
```

2）重构代码，使用 Kafka

重新启动 spark shell，在启动时添加 Kafka 库和 Spark Kafka 连接器库到类路径。需要使用 package 参数让 Spark 为我们下载这些文件：(具体版本号要参阅官网)

```
$ spark-shell --master spark://localhost:7077 --packages
org.apache.spark:spark-streaming-kafka-0-10_2.11:2.3.2,org.apache.kafka:kafka_2.11:0.10.1.0
```

3）使用 Spark Kafka 连接器

Kafka 的连接器有两个版本。第一个是基于 receiver 的连接器，第二个是较新的 direct 连接器。在某些情况下，当使用基于 receiver 连接器的连接器时，相同的消息可能会被多次使用；direct 连接器使实现传入消息的只执行一次处理成为可能。基于 receiver 的连接器效率也较低（它需要设置一个 write-ahead log(预写日志)，这会减慢计算速度）。我们将在本节中使用 direct 连接器。

完整的流处理代码如下：

```

```

4）运行

另外再打开一个终端窗口，执行脚本 streamOrders.sh。这个脚本会从 orders.txt 文件中流式读取行并发送给 orders Kafka topic。（注：orders.txt 文件要和这个脚本文件在同一目录下，另外 Kafka bin 目录要

位于 Path 中-脚本中要调用 kafka-console-producer.sh 脚本)

```
$ chmod +x streamOrders.sh
$ ./streamOrders.sh localhost:9092
```

5）另外再打开一个终端窗口，在这个终端窗口中，启动 kafka-console-consumer.sh 脚本，它会消费来自 metrics topic 的消息，查看我们的流程序的输出内容：

```
$ ./bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic metrics
```

输出结果大致如下：

6）等待所有的文件都被处理完后，从 shell 停止运算流：

```
ssc.stop(false)
```

6.2.3 Spark Streaming 作业性能

对于 Spark Streaming 程序，我们通常希望的是：

- 低延迟：尽可能快地处理到来的记录
- 可扩展：跟上输入数据流的增加
- 容错性：在发生节点故障时，继续获取数据，不要丢失任何数据

在对 Spark Streaming job 调优时，需要提及几个参数，确保程序容错。有了 Spark 流，我们需要决定的第一个参数是 mini-batch duration 时间。没有精确的方法来确定它的值，因为它依赖于作业执行的类型和集群的容量。我们可以通过 Spark web UI 的流页面获得帮助。

（一）获得良好的性能

如果正在运行一个 Spark Streaming 应用程序(StreamingContext)，Streaming 标签页会自动出现在 web UI 上。它显示了几个有用的图表，如下图所示，有以下指标：

- Input Rate：每秒钟到来的记录的数量
- Scheduling Delay：新的 mini-batches 等待其 job 被调度的时间
- Processing Time：处理每一个 mini-batch 的 job 所需的时间
- Total Delay：处理一批数据的总时间

Total Delay 应该低于 mini-batch duration，并且它或多或少应该是一个常数。如果它持续增长，从长远来看，计算是不可持续的，你将不得不减少处理时间(Processing time)，增加并行度，或者限制输入率(Input Rate)。

降低处理时间(processing time)

如果开始看到调度延迟，第一步是尝试优化程序，并减少每个批次的处理时间(processing time)。需要避免不必要的 shuffles。如果将数据发送到外部系统，需要在 partitions 分区中重用连接，并使用某种连接池。

还可以尝试增加 mini-batch 的持续时间，因为 Spark Streaming 作业调度、task 序列化和数据 shuffling 时间将会在更大的数据集上进行，从而减少每个记录的处理时间。但是，设置 mini-batch 的时间过长会

增加每个 mini-batch 的内存需求；此外，较低的输出频率可能无法满足业务需求。

额外的集群资源也可以帮助减少处理时间。例如，增加内存可能会降低垃圾收集频率，而增加 CPU Cores 可能会提高处理速度。

增加并行度

但是为了有效地使用所有的 CPU Cores 并获得更多的吞吐量，需要增加并行性。可以在几个层面上增加它。首先，可以在输入源上做到这一点。例如，Kafka 有分区的概念，而分区决定了消费者可以实现的并行度。如果使用的是 Kafka direct connector，它将自动处理并行性，并将 Spark Streaming 中的消费线程数量与 Kafka 的分区数量相匹配。

但是如果使用的是基于 receiver 的连接器，可以通过创建多个 DStreams 并将它们组合在一起来增加消费者的并行性：

```
val stream1 = ...
val stream2 = ...
val stream = stream2.union(stream2)
```

在下一个层次上，可以通过显式地将 DStreams 重新分区到更高的分区数（通过使用 repartition 方法）来增加并行性。一般的经验法则是，receivers 的数量不应该超过可用的 cores 数量或 executors 的数量。

限制 Input Rate

最后，如果不能减少处理时间或增加并行性，并且仍然会遇到越来越多的调度延迟，那么可能需要限制数据被摄入的速率。手动限制摄入速率有两个参数：spark.stream.receiver.maxRate 用于基于 receiver 的消费者和 spark.streaming.kafka.maxRatePerPartition 用于 Kafka direct 消费者。前者限制了基于 receiver 的流的记录数量，后者是每个 Kafka 分区的记录数量(不只一个 Kafka 分区可以被单个 direct Kafka stream 读取)。两者都表示每秒记录的数量，并且不是默认设置的。

还可以通过设置 spark.streaming.backpressure.enabled 参数为 true 来启用 backpressure 特性。如果调度延迟开始出现，它将自动限制应用程序接收到的最大消息数量。但是，如果设置了，速率永远不会超过前两个参数的值。

（二）实现容错

Streaming 应用程序通常是长时间运行的应用程序，并且驱动程序和 executor 进程的失败是意料之中的。Spark Streaming 使在这些故障中存活下来并且数据零丢失成为可能。

从 executor 故障中恢复

通过运行在 executors 中的 receiver 接收的数据在集群中被复制。如果运行一个 receiver 的 executor 出现故障，则驱动程序将在另一个节点上重新启动该 executor，并且数据将被恢复。不需要特别启用这种行为。Spark 自动完成这些。

从驱动程序故障中恢复

在驱动程序失败的情况下，与 executor 的连接丢失，应用程序需要重新启动。集群管理器可以自动重启驱动程序进程（如果在 Spark Standalone 集群中运行时，通过在 YARN 中使用集群模式，或者在 Mesos 中使用 Marathon，提交时带有--supervise 选项），则可以自动重新启动驱动程序。

一旦驱动程序重新启动，Spark Streaming 就可以通过读取流上下文的检查点状态来恢复以前的 streaming 应用程序的状态。Spark 的 StreamingContext 有一种特殊的初始化方法来利用这个特性。该方

法是 StreamingContext.getOrCreate()，它使用检查点目录和初始化上下文的函数。这个函数需要执行所有通常的步骤来实例化你的流并初始化一个 StreamingContext 对象。getOrCreate 方法首先检查 checkpoint 目录，看看是否存在任何状态。如果有的话，它会加载先前的检查点状态并跳过 StreamingContext 初始化。否则，它调用初始化函数。

要在本章的示例中使用这个，需要重新组织代码：

ssc.checkpoint 告诉 Streaming context 定期将 streaming 的状态保存到指定的目录中。就是这样。如果驱动程序重新启动，应用程序将能够从它中断的地方继续运行。

在驱动程序重新启动时，还需要另一件事来防止数据丢失。Spark Streaming receivers 可以将它们接收到的所有数据在这些数据处理之前写入预写日志。他们向输入源（如果输入源允许消息被承认）确认消息是在写入到预写日志之后才收到的。如果一个 receiver（及其 executor）重新启动，它将从预写日志中读取所有未处理的数据，因此不会发生数据丢失。Kafka direct connector 并不需要预写日志来防止数据丢失，因为 Kafka 提供了该功能。

预写日志默认是没有启用的。我们需要显式地启用它们，这需要设置参数 spark.streaming.receiver.writeAheadLog.enable 为 true。

第 7 章　Spark 结构化流

Spark 2.0 引入了更高级别的新的流处理 API，叫做 Structured Streaming(结构化流)。结构化流是一种快速、容错、精确一次的有状态流处理方法。它支持流分析，而无需考虑流的底层机制。在结构化流处理模型中，可以将输入看作不断增长的表的数据。触发器指定检查输入新数据到达的时间间隔。

在 Spark 2.0 的结构化流中，查询表示输入上的查询或操作，如映射、筛选和合并，结果表示在每个触发器间隔中根据指定的操作更新的最终表。输出定义了在每个时间间隔内将结果写入数据接收器的部分。

这个可伸缩和容错的高级流 API 构建在 Spark SQL 引擎之上，与 SQL 查询和 DataFrame/Dataset API 紧密集成。主要优点是使用相同的 Spark DataFrame 和 Dataset API 以及 Spark 引擎计算出操作所需的增量和连续执行，简化实时、连续的大数据应用程序的开发。结构化流将批处理和流处理计算统一起来，并可以连接（join）流和批数据。

此外，还可以使用查询管理 API 来管理多个并行运行的流查询。例如，可以列出正在运行的查询、停止和重新启动查询、在失败的情况下检索异常等等。

7.1 结构化流简介

Spark 结构化流提供了快速、可扩展、容错、端到端的精确一次性流处理，而用户无需对流进行推理。结构化流操作直接工作在 DataFrame(或 DataSets)上。不再有“流”的概念，只有流式 DataFrames 和普通 DataFrames。流式 DataFrames 是作为 append-only 表实现的。在流数据上的查询返回新的 DataFrame，使用它们就像在批处理程序中一样。

使用 Spark 结构化流的模型如下图所示：

结构化流的关键思想是将实时数据流视为一个不断追加的表。这就产生了一个新的流处理模型，它与批处理模型非常相似。当一组新的数据到达时，将这些新到达的数据作为一组新的行添加到输入表。我们将把流计算表示为标准的批处理查询，就像在静态表上的查询一样，Spark 将其作为无边界输入表上的增量查询运行。

将输入数据流视为“输入表”。流上到达的每个数据项都像是向输入表追加了一个新行。如下图所示：

因此，无论是静态还是流式数据，只需像在静态数据表上那样启动类似于批处理的查询，Spark 就会在无界输入表上作为增量查询运行它。因此，开发人员在输入表上定义一个查询，对于静态有界表和动态无界表都是一样的。

在结构化流处理模型中，用户使用批处理 API 在输入表上进行查询，Spark SQL Planner 在流数据上增量执行，整个过程如下图所示：

考虑传入数据流的方式，只不过是一个不断追加的表。这样就能够利用现有的用于 DataFrame 和 Dataset 的结构化 API（在 Scala、Java 或 Python 中），执行流计算，并且随着新的流数据的到来，由结构化流引擎负责增量地、持续地运行它们。

传统上，当从流式应用程序发送数据到外部存储系统时，确保没有重复的数据或数据丢失是开发人员的责任。这是流应用程序开发人员提出的一个痛点。在内部，结构流引擎已经提供了一个端到端的、精确一次性的保证，现在同样的保证被扩展到外部存储系统，只要这些系统支持事务。

从根本上说，结构化流由 Spark SQL 的 Catalyst 优化器负责优化。因此，它使开发人员不再担心底层的管道，在处理静态或实时数据流时，使查询更高效。

7.2 结构化流编程模型

假设我们想维护从监听 TCP 套接字的数据服务器接收到的文本数据的运行时单词计数。让我们看看如何使用结构化流来表达这一需求。

【例】实现 Spark 运行时单词计数流程序。（先分步执行，最后是完整的程序）

首先需要使用 Netcat(在大多数类 unix 系统中可以找到的小型实用程序)作为数据服务器。在 Linux 的终端中，执行如下命令，启动 Netcat 服务器，使其保持运行。

```
$ nc -lk 9999
```

注：如果没有 Netcat 服务器的话，可以使用如下命令先安装：

```
$ sudo yum install -y nc
```

然后，编写我们的结构化流程序并执行。请按以下步骤操作。

1）先导入必要的类并创建本地 SparkSession。

```

```

2）接下来，创建一个流 DataFrame，它表示从所监听的 localhost:9999 服务器接收到的文本数据，并对该 DataFrame 进行转换以进行单词计数。

```
// 创建表示从连接到 localhost:9999 的输入行流的 DataFrame

```

3）在流数据上设置查询。

为此，我们进行设置，以便每次更新计数时都将完整的计数集(由 outputMode("complete")指定)打印到控制台。然后使用 start()启动流计算。

```
// 开始运行将运行时单词计数打印到控制台的查询
val query = wordCounts.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```

执行此代码后，流计算将在后台开始。其中 query 对象是该活动流查询的句柄，使用 awaitTermination()等待查询终止，以防止查询处于活动状态时进程退出。对于生产和长期运行的流应用程序，有必要调用 StreamingQuery.awaitTermination()函数，这是一个阻塞调用，它会防止 Driver 驱动程序退出，并允许流查询持续运行和当新数据到达数据源时处理新数据。

4） 切换到 Netcat 窗口，输入几行任意的内容，单词之间用空格分割。例如：

```
good good study
study day day up
```

5）切换回流程序执行窗口，查看输出结果。会看到类似如下这样的输出结果：

```

```

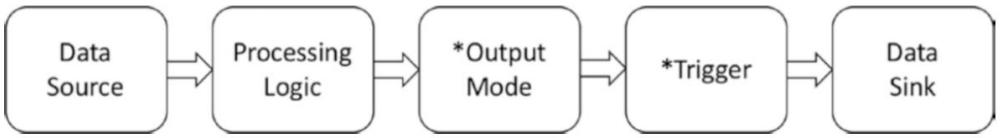
有时我们希望停止流查询来改变输出模式、触发器或其他配置。可以使用 StreamingQuery.stop()函数来阻止数据源接收新数据，并停止在流查询中逻辑的连续执行。下面的代码是管理流查询的一个示例：

7.3 结构化流核心概念

Spark Structured Streaming 应用程序包括以下主要部分：

- 指定一个或多个流数据源。
- 提供了以 DataFrame 转换的形式操纵传入数据流的逻辑。
- 定义输出模式和触发器（都有默认值，所以是可选的）。
- 最后指定一个将结果写出到的数据接收器（data sink）。

下图概述了前面提到的步骤。可选的选项用星号标记。



下面的部分将详细描述这些概念。

数据源（Data Sources）

对于批处理，数据源是驻留在某些存储系统上的静态数据集，如本地文件系统、HDFS 或 S3。结构化流的数据源是完全不同的。他们生产的数据是连续的，可能永远不会结束，而且生产速率也会随着时间而变化。

Spark 结构化流提供了以下开箱即用的数据源：

- Kafka 源：要求 Apache Kafka 的版本是 0.10 或更高版本。这是生产环境中最流行的数据源。连接和读取来自 Kafka 主题的数据需要提供一组特定的设置。
- 文件源：文件位于本地文件系统、HDFS 或 S3 上。当新的文件被放入一个目录中时，这个数据源将会把它们挑选出来进行处理。支持常用的文件格式，如文本、CSV、JSON、ORC 和 Parquet。在处理这个数据源时，一个好的实践是先完全地写出输入文件，然后再将它们移动到这个数据源的路径中。（例如，流程序监控的是 HDFS 上的 A 目录，那么先将输入文件写出到 HDFS 的 B 目录中，再从 B 目录将它们移动到 A 目录）
- Socket 源：这仅用于测试目的。它从一个监听特定的主机和端口的 socket 上读取 UTF-8 数据。
- Rate 源：这仅用于测试和基准测试。这个源可以被配置为每秒产生许多事件，其中每个事件由时间戳和一个单调递增的值组成。这是学习结构化流时使用的最简单的源。

数据源需要提供的一个重要的属性是一种跟踪流中的读位置的方法，用于结构化的流来传递端到端、精确一次性保证。例如，Kafka 的数据源提供了一个 Kafka 的偏移量来跟踪一个主题分区的读位置。这个属性决定一个特定的数据源是否具有容错能力。

下表描述了每个开箱即用数据源的一些选项。

数据源	是否容错	配置
File	是	path：输入目录的路径 maxFilesPerTrigger：每个触发器读取新行的最大数量 latestFirst：是否处理最新的文件(根据modification time)
Socket	否	要求有如下的参数： host：要连接到的主机

		port：要连接到的端口号
Rate	是	rowsPerSecond：每秒钟生成的行的数量 rampUpTime：在到达rowsPerSecond之前的时间，以秒为单位 numPartitions：分区的数量
Kafka	是	kafka.bootstrap.servers：Kafka brokers列表，以逗号分隔的host:port subscribe：主题列表，以逗号分隔

Apache Spark 2.3 引入了 DataSource V2 APIs，这是一组官方支持的接口，用于 Spark 开发人员开发定制数据源，这些数据源可以很容易地与结构化流集成。有了这些定义良好的 API 集，在不久的将来，自定义结构化流源的数量将急剧增加。

输出模式

输出模式是一种方法，可以告诉结构流如何将输出数据写入到 sink 中。这个概念对于 Spark 中的流处理来说是独一无二的。输出模式有三个选项。

- append 模式：如果没有指定输出模式，这是默认模式。在这种模式下，只有追加到结果表的新行才会被发送到指定的输出接收器。只有自上次触发后在结果表中附加的新行将被写入外部存储器。这仅适用于结果表中的现有行不会更改的查询。
- complete 模式：此模式将数据完全从内存写入接收器，即整个结果表将被写到输出接收器。当对流数据执行聚合查询时，就需要这种模式。
- update 模式：只有自上次触发后在结果表中更新的行才会被写到输出接收器中。对于那些没有改变的行，它们将不会被写出来。注意，这与 complete 模式不同，因为此模式不输出未更改的行。

触发器类型

触发器是另一个需要理解的重要概念。结构化流引擎使用触发器信息来确定何时在流应用程序中运行提供的流计算逻辑。下表描述了不同的触发类型。

类型	描述
未指定(默认)	对于默认类型，Spark将使用微批模型，并且当前一批数据完成处理后，立即处理下一批数据
固定周期	对于这个类型，Spark将使用微批模型，并基于用户提供的周期处理这批数据。如果因为任何原因导致上一批数据的处理超过了该周期，那么前一批数据完成处理后，立即处理下一批数据。换句话说，Spark将不会等到下一个周期区间边界。
一次性	这个触发器类型意味着用于一次性处理可用的批数据，并且一旦该处理完成的话，Spark将立即停止流程序。当数据量特别低时，这个触发器很有用，因此，构建一个集群并每天处理几次数据更划算。
持续	这个触发器类型调用新的持续处理模型，该模型是设计用于非常低延迟需求的特定流应用程序的。这是Spark 2.3中新的实验性处理模式。这个时候将支持“最少一次性”保证。

数据接收器（Data Sinks）

数据接收器是用来存储流应用程序的输出的。不同的 sinks 可以支持不同的输出模式，并且具有不同的容错能力，了解这一点很重要。Spark 结构化流支持以下几种数据接收器：

- Kafka sink：要求 Apache Kafka 的版本是 0.10 或更高版本。有一组特定的设置可以连接到 Kafka 集群。
- File sink：这是文件系统、HDFS 或 S3 的目的地。支持常用的文件格式，如文本、CSV、JSON、ORC、Parquet。
- Foreach sink：这是为了在输出中的行上运行任意计算。
- Console sink：这仅用于测试和调试目的，以及在处理低容量数据时。每个触发器上输出被打印到控制台。

- Memory sink：这是在处理低容量数据时进行测试和调试的目的。它使用驱动程序的内存来存储输出。

下表列出了每个 sink 的各种选项：

名称	支持的输出模式	是否容错	配置
File	Append	是	path：这是输入目录的路径。支持所有流行的文件格式。详细信息可查看DataFrameWriter API。
Foreach	Append Update Complete	依情况而定	这是一个非常灵活的接收器，它是特定于实现的
Console	Append Update Complete	否	numRows：这是每个触发器输出的行的数量。默认是20行。 truncate：如果每一行太长的话，是否截断。默认是true。
Memory	Append Complete	否	N/A
Kafka	Append Update Complete	是	kafka.bootstrap.servers：Kafka brokers列表，以逗号分隔的host:port topic：这是写入数据的Kafka主题

数据接收器必须支持的一个重要的属性（用于结构化的流交付端到端、精确一次性保证）是处理重做的幂等性。换句话说，它必须能够处理使用相同数据的多个写（在不同的时间发生），结果就像只有一个写一样。多重写是在故障场景中重新处理数据的结果。

水印(Watermarking)

数字水印是流处理引擎中常用的一种技术，用于处理迟到的数据。流应用程序开发人员可以指定一个阈值，让结构化的流引擎知道数据在事件时间（event time）内的预期延迟时间。有了这个信息，超过这个预期延迟时间到达的迟到数据会被丢弃。更重要的是，结构化流使用指定的阈值来确定何时可以丢弃旧状态。没有这些信息，结构化流将需要无限期地维护所有状态，这将导致流应用程序的内存溢出问题。任何执行某种聚合或连接的生产环境下的结构化流应用程序都需要指定水印。这是一个重要的概念，关于这个主题的更多细节将在后面的部分中讨论和说明。

7.4 使用数据源

上一节描述了结构化流提供的每个内置源。本节将更详细地介绍这些数据源，并将提供使用它们的示例代码。

7.4.1 使用 Socket 数据源

套接字数据源很容易使用，只需要提供主机和端口号，但仅限于学习和测试使用，不在生产环境中使用。下面这个示例应用 socket 数据源。

1）在启动套接字数据源的流式查询之前，首先使用一个网络命令行实用工具，如 Mac 上的 nc 或 Windows 上的 netcat，启动一个套接字服务器。打开一个终端窗口，执行下面的命令，启动带有端口号 9999 的套接字服务器：

```
$ nc -lk 9999
```

2）另外打开第二个终端，启动 spark shell：

```
$ spark-shell --master spark://localhost:7077
```

3）在 Spark Shell 中，执行以下结构化流处理代码：

```
// 从 Socket 数据源读取流数据
val socketDF = spark.readStream
      .format("socket")
      .option("host", "localhost")
      .option("port", "9999")
      .load()

val words = socketDF.as[String].flatMap(_.split(" "))
val wordCounts = words.groupBy("value").count()

val query = wordCounts.writeStream
      .format("console")
      .outputMode("complete")
      .start()
```

4）回到第一个终端窗口，任意输入一些单词，以空格分隔，并回车。多输入一些行，然后在第二个终端窗口观察流计算输出。

在第二个终端窗口观察到的输出结果：

5）当完成测试 Socket 数据源时，可以通过调用 stop 函数来停止流查询。在停止流查询之后，在第一个终端中输入任何东西都不会导致在 Spark shell 中显示任何东西。

```
query.stop
```

7.6.2 使用 Rate 数据源

与 Socket 数据源类似，Rate 数据源是为测试和学习目的而设计的。它支持以下这些选项：

- rowsPerSecond：每秒应该生成多少行，例如，指定为 100。默认是 1。如果这个数字很高，那么就可以提供下一个可选配置 rampUpTime。
- rampUpTime：在生成速度变为 rowsPerSecond 之前需要多长时间用来提升，例如，5s。默认是 0s。使用比秒更细的粒度将被截断为整数秒。
- numPartitions：生成行的分区数。默认是 Spark 默认并行度。

Rate 源将尽力达到 rowsPerSecond，但是查询可能受到资源限制，可以调整 numPartitions 以帮助达到所需的速度。

Rate 源产生的每一段数据只包含两列：时间戳和自动增加的值。下面的示例包含打印 Rate 数据源数据的代码。请启动 Spark Shell，执行以下代码：

```

```

观察到每秒输出 10 条数据。其中部分批次数据如下所示：

```

```

值得注意的一件事是，value 列中的数字保证在所有分区中都是连续的。下面的代码展示了三个分区的输出结果。

```
import org.apache.spark.sql.functions._
...


```

观察到输出结果如下所示：

前面的输出显示了这 10 行分布在三个分区上，并且这些值是连续的，就好像它们是为单个分区生成的一样。

7.6.3 使用 File 数据源

文件数据源是最容易理解和使用的。Spark Structured Streaming 开箱即用地支持所有常用的文件格式，包括文本、CSV、JSON、ORC 和 Parquet。要获得支持的文件格式的完整列表，请参考 DataStreamReader 接口。

File 数据源支持以下选项配置：

- path：输入目录的路径，对所有文件格式都通用。
- maxFilesPerTrigger：每个触发器中考虑处理的最大新文件数(默认：no max) 。
- latestFirst：是否先处理最新的文件，当有大量文件积压时很有用(默认：false)。
- fileNameOnly：是否仅根据文件名而不是根据完整路径检查新文件(默认:false)。 将此值设置为“true”后，以下文件将被认为是相同的文件，因为它们的文件名都是一样的，均为"dataset.txt”：
  - "file:///dataset.txt"
  - "s3://a/dataset.txt"
  - "s3n://a/b/dataset.txt"
  - "s3a://a/b/c/dataset.txt"

下面是使用 File 数据源的流程序模板代码：

下面我们通过一个示例程序来演示如何使用结构化流读取文件数据源。

【示例】移动电话的开关机等事件会保存在 json 格式的文件中。现在编写 Spark 结构化流处理程序来读取这些事件并处理。请按以下步骤操作。

1）准备数据

在本示例中，我们使用文件数据源，该数据源以 json 文件的格式记录了一小组移动电话动作事件。每个事件由三个字段组成：

- id：表示手机的唯一 ID。在样例数据集中，电话 ID 将类似于 phone1、phone2、phone3 等。
- action：表示用户所采取的操作。该操作的可能值是"open"或"close"。
- ts：表示用户 action 发生时的时间戳。这是事件时间(event time)。

我们准备了三个存储移动电话事件数据的 JSON 文件，三个文件的内容如下：

file1.json

```
{"id":"phone1","action":"open","ts":"2018-03-02T10:02:33"}
{"id":"phone2","action":"open","ts":"2018-03-02T10:03:35"}
{"id":"phone3","action":"open","ts":"2018-03-02T10:03:50"}
{"id":"phone1","action":"close","ts":"2018-03-02T10:04:35"}
```

file2.json

```
{"id":"phone3","action":"close","ts":"2018-03-02T10:07:35"}
{"id":"phone4","action":"open","ts":"2018-03-02T10:07:50"}
```

file3.json

```
{"id":"phone2","action":"close","ts":"2018-03-02T10:04:50"}
{"id":"phone5","action":"open","ts":"2018-03-02T10:10:50"}
```

为了模拟数据流的行为，我们将把这三个 JSON 文件复制到项目的“src/main/resources/mobile”目录下。

2）先导入相关的依赖包。

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
```

3）为手机事件数据创建模式（schema）

默认情况下，结构化流在从基于文件的数据源读取数据时需要一个模式（因为最初目录可能是空的，因此结构化的流无法推断模式）。但是，可以设置配置参数 spark.sql.streaming.schemaInference 的值为 true 来启用模式推断。在这个例子中，我们将显式地创建一个模式，代码如下所示：

```
// 为手机事件数据创建一个 schema
val fields = Array(
  StructField("id", StringType, nullable = false),
  StructField("action", StringType, nullable = false),
  StructField("ts", TimestampType, nullable = false)
)
val mobileDataSchema = StructType(fields)
```

3）读取流文件数据源，创建 DataFrame，并将 action 列值转换为大写。

```
// 监听的文件目录
val dataPath = "src/main/resources/mobile"

// 读取指定目录下的源数据文件，一次一个
val mobileDF = spark.readStream
  .option("maxFilesPerTrigger", 1)
  .option("mode","failFast")
  .schema(mobileDataSchema)
  .json(dataPath)

// mobileSSDF.isStreaming
// mobileSSDF.printSchema()

// 将所有"action"列值转换为大写
import spark.implicits._
val upperDF = mobileDF.select($"id",upper($"action"),$"ts")
```

4）将结果 DataFrame 输出到控制台显示。

```
// 结果输出到控制台
val query = upperDF.writeStream
  .format("console")
  .option("truncate","false")
  .outputMode("append")
  .start()
```

5）执行流处理程序，输出结果如下所示。

```

```

完整的代码如下所示。

```

```

思考：本例中我们将源数据文件放在本项目的路径下。大家可以思考一下，如果将数据放在 HDFS

的指定路径下，该怎么处理？

7.6.4 使用 Kafka 数据源

Kafka 通常用于构建实时流数据管道，以可靠地在系统之间移动数据，还用于转换和响应数据流。Kafka 作为集群运行在一个或多个服务器上。Kafka 的一些关键概念描述如下：

- Topic：主题。消息发布到的类别或流名称的高级抽象。主题可以有 0、1 或多个消费者，这些消费者订阅发布到该主题的消息。用户为每个新的消息类别定义一个新主题；
- Producers：生产者。向主题发布消息的客户端；
- Consumers：消费者。使用来自主题的消息的客户端；
- Brokers：服务器。复制和持久化消息数据的一个或多个服务器。

此外，生产者和消费者可以同时对多个主题进行读写。每个 Kafka 主题都是分区的，写入每个分区的消息都是顺序的。分区中的消息具有一个偏移量，用来惟一标识分区内的每个消息。（Kafka 官网）

主题的分区是分布式的，每个 Broker 处理对分区共享的请求。每个分区在 Brokers（数量可配置的）之间复制。Kafka 集群在一段时间内（可配置的）保留所有已发布的消息。Kafka 使用 ZooKeeper 作为其分布式进程的协调服务。

说明：Kafka 的数据源可能是在生产型流应用程序中最常用的数据源。为了有效地处理这个数据源，我们需要一定的 Kafka 使用基本知识。

在使用 Kafka 数据源时，我们的流程序实际上充当了 Kafka 的消费者。因此，流程序所需要的信息与 Kafka 的消费者所需要的信息相似。下表列出了配置 Kafka 数据源一些选项：

Option	值	描述
kafka.bootstrap.servers	host1:port1, host2:port2	Kafka服务器列表，逗号分隔。
subscribe	topic1, topic2	这个数据源要读取的主题名列表，以逗号分隔。
subscribePattern	topic.*	使用正则模式表示要读取数据的主题，比subscribe要灵活。
assign	{topic1:[1,2], topic2:[3,4]}	指定要读取数据的主题的分区。这个信息必须是json格式。

其中必需的信息是要连接的 Kafka 服务器的列表，以及一个或多个从其读取数据的主题。为了支持选择从哪个主题和主题分区来读取数据的各种方法，它支持三种不同的方式来指定这些信息。我们只需要选择最适合自身用例的那个即可。

还有一些可选配置选项，都有自己的默认值，列出在下表中。

Option	默认值	值	描述

startingOffsets	latest	earliest, latest
		每个主题的开始偏移位置，json格式字符串，例如：
		{
		"topic1":{"0":45, "1":-1},
		"topic2":{"0":-2}
		}
			earliest：意味着主题的开始处
			latest：意味着主题中的任何最新数据
			当使用JSON字符串格式时，-2代表在一个特定分区中的earliest offset，-1代表在一个特定分区中的latest offset

endingOffsets	latest	Latest
		json格式字符串，例如：
		{
		"topic1":{"0":45, "1":-1},
		"topic2":{"0":-1}
		}
			latest：意味着主题中的最新数据
			当使用JSON字符串格式时，-1代表在一个特定分区中的latest offset。当然-2不适用于此选项

maxOffsetsPerTrigger	none	Long，例如，500	此选项是一种速率限制机制，用于控制每个触发器间隔要处理的记录数量。如果指定了一个值，它表示所有分区的记录总数，而不是每个

			分区的记录总数。

注：这里只列出了部分。更详细的 option 设置选项可以参考：

https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html

设置 Kafka

要设置 Kafka，首先需要下载它：http://kafka.apache.org/downloads.html。注意，要选择与 Spark 版本相兼容的版本。在我们的平台上，使用 kafka_2.11-2.4.1.tgz 这个版本。

1）解压缩下载的 Kafka 压缩包到~/bigdata/目录下：

```
$ cd ~/bigdata
$ tar -xvfz kafka_2.11-2.4.1.tgz
```

2）Kafka 依赖于 Apache ZooKeeper，所以在启动 Kafka 之前，要先启动它：

```
$ cd ~/bigdata/kafka_2.11-2.4.1
$ ./bin/zookeeper-server-start.sh config/zookeeper.properties
```

这将在 2181 端口启动 ZooKeeper 进程，并让 ZooKeeper 在后台工作。

3）接下来，启动 Kafka 服务器：

```
$ ./bin/kafka-server-start.sh config/server.properties
```

4）创建主题：

```
$ ./bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

5）查看已有的主题：

```
$ ./bin/kafka-topics.sh --list --zookeeper localhost:2181
```

6）删除一个主题。

```
$ ./bin/kafka-topics.sh --delete --bootstrap-server localhost:9092 --topic test
```

需要修改启动的配置文件 server.properties，设置“delete.topic.enable=true”（默认设置为 false）。

默认情况下，Kafka 的数据源并不是 Spark 的内置数据源，因此如果要开发读取 Kafka 数据的 Spark 结构化流处理程序，必须添加 Kafka 的依赖包到 classpath 中。

如果我们要从 spark shell 使用 Kafka 数据源，那么需要在启动 spark shell 时将依赖的 JAR 包添加到 classpath 中。有两种方式可以做到：

- 手动将依赖包添加到 classpath。

首先将 spark-sql-kafka-0-10_2.12-3.1.2.jar 、 spark-streaming-kafka-0-10-assembly_2.12-3.1.2.jar 和 kafka_2.12-2.4.1.jar 包拷贝到~/bigdata/spark-3.1.2/jars/目录下。然后启动 spark-shell：

```

```

- 使用 package 参数让 Spark 为我们下载这些文件：(具体版本号要参阅官网)

```

```

- 使用 package 参数让 Spark 为我们下载这些文件：(具体版本号要参阅官网)

```

```

如果是使用 IDE 开发并使用 SBT 来管理依赖，则需要在 build.sbt 中添加如下依赖配置以使用 Kafka 数据源。

```
libraryDependencies += "org.apache.spark" %% "spark-sql-kafka-0-10" % "2.4.7"
libraryDependencies += "org.apache.kafka" %% "kafka" % "2.4.1"
```

如果是使用 IDE 开发并使用 Maven 来管理依赖，则需要在 pom.xml 中添加如下依赖配置以使用

Kafka 数据源。

下面我们通过一个示例来演示如何编写 Spark 结构化流处理程序来读取 Kafka 中的数据。

【示例】编写 Spark 结构化流程序作为 Kafka 的消费者程序，Kafka 作为流数据源。

在这个示例中，我们使用 Kafka 自带的生产者脚本向 Kafka 的 test 主题发送内容，而 Spark 结构化流程序会订阅该主题。一旦它收到了订阅的消息，马上输出到控制台中。程序处理流程如下图所示：

首先，我们编写 Spark 结构化流程序代码。实现如下：

要运行这个流程序，请按以下步骤进行操作：

1）启动 zookeeper 服务

Kafka 依赖于 Apache ZooKeeper，所以在启动 Kafka 之前，要先启动它。

打开一个终端窗口，执行如下命令：

```
$ cd ~/bigdata/kafka_2.11-2.4.1
$ ./bin/zookeeper-server-start.sh config/zookeeper.properties
```

等待 30 秒左右 ZooKeeper 启动。

2）接下来，启动 Kafka 服务器。

另打开一个终端窗口，执行如下命令：

```
$ cd ~/bigdata/kafka_2.11-2.4.1
$ ./bin/kafka-server-start.sh config/server.properties
```

等待 30 秒左右 Kafka 启动。

3）查看和创建 Kafka 主题(如果已经有了 test 主题，则此步略过)。

另外打开第三个终端窗口，执行如下命令，创建 test 主题：

```
$ cd ~/bigdata/kafka_2.11-2.4.1
$ ./bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

查看已有的主题，使用如下的命令：

```
$ ./bin/kafka-topics.sh --list --zookeeper localhost:2181
```

4）启动流程序，开始接收从 Kafka“test”主题订阅的消息。

5）向 Kafka“test”主题发送消息

另外打开第四个终端窗口，执行如下命令，生产消息并发布给“test”主题。

```
$ cd ~/bigdata/kafka_2.11-2.4.1
$ ./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

然后，随意键入一些消息。例如：

```
> good good study
> day day up
```

6）回到流程序执行窗口，如果一切正常，应该可以看到在控制台输出收到的订阅消息，如下：

从上面的输出内容可以看出，从 Kafka 中读取的数据每一列都有固定的格式，如下表所示：

列	类型
key	binary
value	binary
topic	string
partition	int
offset	long
timestamp	long
timestampType	int

在从 Kafka 读取消息时，有多种不同的方式。下面这个例子包含了一些指定 Kafka 的主题、分区和从 Kafka 读取消息的偏移量的不同的变化方式。

7.6.5 使用自定义数据源

在 Spark 2.3 之前，Data Source API 有一些限制，而且扩展性不是很好。因此，对于 Spark 开发人员来说，构建自定义数据源非常具有挑战性。

从 Spark 2.3 开始，Spark 结构化流引入了 Data Source V2 API 来解决 V1 中的问题，并提供一组干净、可扩展且易于使用的新 API。Data Source V2 API 仅在 Scala 中可用。

所有自定义数据源都必须实现一个名为 DataSourceV2 的标记接口，然后它可以选择是实现接口 ContinuousReadSupport 还是 MicroBatchReadSupport，或者两者都实现。例如，KafkaSourceProvider.scala 实现了这两个接口，因为它允许用户根据用例选择使用哪种处理模式。这两个接口行为中都有一个工厂方法，分别用于创建 ContinuousReader 或 MicroBatchReader 的实例。自定义数据源实现的大部分将用于实现在这两个接口中定义的 API。

7.5 流 DataFrame 操作

前面的例子表明，一旦配置和定义了数据源，DataStreamReader 将返回一个 DataFrame 的实例。这意味着我们可以使用大多数熟悉的操作和 Spark SQL 函数来表达应用程序流计算逻辑。但是要注意，并不是所有的 DataFrame 操作都受流式 DataFrame 支持的，比如 limit、distinct 和 sort 就不能在流 DataFrame 上使用，这是因为它们在流数据处理的上下文中不适用。

7.5.1 选择、投影和聚合操作

结构化流的一个优点是具有一组用于 Spark 的批处理和流处理的统一 API。使用流数据格式的 DataFrame，可以应用任何 select 和 filter 转换，以及任何作用在个别列上的 Spark SQL 函数。此外，基本聚合和高级分析函数也可用于流 DataFrame。

【示例】移动电话事件数据流分析。

移动电话的开关机等事件会保存在 json 格式的文件中。现在编写 Spark 结构化流处理程序来读取这些事件并处理。请按以下步骤操作。

1）准备数据

在本示例中，我们使用文件数据源，该数据源以 json 文件的格式记录了一小组移动电话动作事件。每个事件由三个字段组成：

- id：表示手机的唯一 ID。在样例数据集中，电话 ID 将类似于 phone1、phone2、phone3 等。
- action：表示用户所采取的操作。该操作的可能值是"open"或"close"。
- ts：表示用户 action 发生时的时间戳。这是事件时间(event time)。

我们准备了三个存储移动电话事件数据的 JSON 文件，三个文件的内容如下：

file1.json

```
{"id":"phone1","action":"open","ts":"2018-03-02T10:02:33"}
{"id":"phone2","action":"open","ts":"2018-03-02T10:03:35"}
{"id":"phone3","action":"open","ts":"2018-03-02T10:03:50"}
{"id":"phone1","action":"close","ts":"2018-03-02T10:04:35"}
```

file2.json

```
{"id":"phone3","action":"close","ts":"2018-03-02T10:07:35"}
{"id":"phone4","action":"open","ts":"2018-03-02T10:07:50"}
```

file3.json

```
{"id":"phone2","action":"close","ts":"2018-03-02T10:04:50"}
{"id":"phone5","action":"open","ts":"2018-03-02T10:10:50"}
```

为了模拟数据流的行为，我们将把这三个 JSON 文件复制到项目的“src/main/resources/mobile”目录下。

2）先导入相关的依赖包。

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
```

3）为手机事件数据创建模式（schema）

默认情况下，结构化流在从基于文件的数据源读取数据时需要一个模式（因为最初目录可能是空的，因此结构化的流无法推断模式）。但是，可以设置配置参数 spark.sql.streaming.schemaInference 的值为 true 来启用模式推断。在这个例子中，我们将显式地创建一个模式，代码如下所示：

```

```

3）读取流文件数据源，创建 DataFrame，并将 action 列值转换为大写。

```

```

4）执行过滤、投影、聚合等转换操作。

```
    t
```

5）将结果 DataFrame 输出到控制台显示。

```

```

6）执行流处理程序，输出结果如下所示。

```

```

完整的代码如下。

```

```

在这个示例中，我们采用的输出模式是“complete”。在没有聚合操作的情况下，不能使用“complete”输出模式；在有聚合操作的情况下，不能使用“append”模式。

需要注意，在流 DataFrame 中，不支持以下 DataFrame 转换（因为它们太过复杂，无法维护状态，或者由于流数据的无界性）：

- 在流 DataFrame 上的多个聚合或聚合链。
- limit 和 take N 行。
- distinct 转换。
- 在没有任何聚合的情况下对流 DataFrame 进行排序。

任何使用不受支持的操作的尝试都会导致一个 AnalysisException 异常以及类似“XYZ 操作不受流 streaming DataFrame/Datasets 支持”这样的消息。

7.5.2 执行 join 操作

可以用一个流 DataFrame 来 join 连接另一个静态的 DataFrame 或者流 DataFrame。然而，join 连接是一个复杂的操作，其中最棘手的在于并不是所有的数据都是在连接时是可用的流 DataFrame。因此，join 连接的结果是在每个触发器点上增量地生成的。

从 Spark 2.3 开始，结构化流支持 join 两个流 DataFrames。考虑到流 DataFrame 的无界性，结构化的流必须维护两个流 DataFrames 的过去数据，以匹配任何未来的、尚未收到的数据。

下面我们通过一个 IOT 示例来学习两个流 DataFrame 的连接操作。

【示例】在某个数据中心，通过不同的传感器采集不同类型的实时数据。其中第一个传感器采集不同机架的实时温度读数；第二个传感器采集不同机架的实时负载信息。这些数据都存储在 json 格式的数据文件中。

包含了数据中心中不同位置机架的温度读数，数据如下所示：

file1_temp.json：

```
{"temp_location_id":"rack1","temperature":99.5,"temp_taken_time":"2017-06-02T08:01:01"}
{"temp_location_id":"rack2","temperature":100.5,"temp_taken_time":"2017-06-02T08:06:02"}
{"temp_location_id":"rack3","temperature":101.0,"temp_taken_time":"2017-06-02T08:11:03"}
{"temp_location_id":"rack4","temperature":102.0,"temp_taken_time":"2017-06-02T08:16:04"}
```

包含了同一数据中心中每台计算机的负载信息，数据如下所示：

file2_load.json：

```
{"load_location_id":"rack1","load":199.5,"load_taken_time":"2017-06-02T08:01:02"}
{"load_location_id":"rack2","load":1105.5,"load_taken_time":"2017-06-02T08:06:04"}
{"load_location_id":"rack3","load":2104.0,"load_taken_time":"2017-06-02T08:11:06"}
{"load_location_id":"rack4","load":1108.0,"load_taken_time":"2017-06-02T08:16:08"}
{"load_location_id":"rack4","load":1108.0,"load_taken_time":"2017-06-02T08:21:10"}
```

我们需要编写一个 Spark 流处理程序，连接这个两个流数据集，统计每个机架实时的温度和负载。实现代码如下所示（注意其中的连接条件和时间约束条件设置）：

```

```

注*：不是必须的。如果是非标准时间戳格式，使用这个 option 来指定解析格式。

执行以上代码，可以得到如下的输出结果：

```

```

那么，在这个结果表上，我们可以进一步执行 Spark SQL 查询操作。

当连接一个静态 DataFrame 和一个流 DataFrame 时，以及当连接两个流 DataFrames 时，外连接会受到更多的限制。下表提供了相关的一些细节。

左侧 + 右侧	连接类型	说明
静态数据 + 流数据	内连接	支持

静态数据 + 流数据	左外连接	不支持
静态数据 + 流数据	右外连接	支持
静态数据 + 流数据	全外连接	不支持
流数据 + 流数据	内连接	支持
流数据 + 流数据	左外连接	有条件地支持。必须在右侧指定水印以及时间约束
流数据 + 流数据	右外连接	有条件地支持。必须在左侧指定水印以及时间约
流数据 + 流数据	全连接	不支持

7.6 使用数据接收器

流应用程序的最后一步通常是将计算结果写入一些外部系统或存储系统。结构化流提供了 5 个内置数据接收器（Data Sink）。其中三个是用于生产的，两个用于测试目的。下面的部分将详细介绍每个 Data Sink。

7.6.1 使用 File Data Sink

File data sink 是一个非常简单的数据接收器，需要提供的唯一必需选项是输出目录。由于 File data sink 是容错的，结构化的流将需要一个检查点位置来写进度信息和其他元数据，以帮助在出现故障时进行恢复。

【示例】配置 Rate 数据源，每秒产生 10 行数据，将生成的数据行发送到两个分区，并将数据以 JSON 格式写出到指定的目录。

实现的代码如下。

```
def main(args: Array[String]): Unit = {

  val spark = SparkSession.builder()
    .master("local")
    .appName("file sink")
    .getOrCreate()

  // spark.sparkContext.setLogLevel("WARN")          // 设置日志级别

  // 将数据从 Rate 数据源写出到 File Sink
  val rateSourceDF = spark.readStream
    .format("rate")
    .option("rowsPerSecond","10")                   // 每秒产生 10 条数据
    .option("numPartitions","2")                    // 两个分区
    .load()

  val query = rateSourceDF.writeStream
    .outputMode("append")
    .format("json")                                 // or "csv"
    .option("path", "tmp/output")                   // 设置输出目录
    .option("checkpointLocation", "tmp/ck")         // 设置 checkpoint
    .start()

  // 等待流程序结束
```

```
    query.awaitTermination()
  }
```

由于分区的数量被配置为两个分区，所以每当结构化流在每个触发点上写出数据时，就会将两个文件写到输出目录中。因此，如果检查输出目录的话，将会看到带有名称的文件，这些名称以 part-00000 或 part-00001 开头。Rate 数据源配置为每秒 10 行，并且有两个分区；因此，每个输出包含 5 行，内容如下所示。

part-00000-*.json：

```
{"timestamp":"2021-02-03T17:56:46.283+08:00","value":0}
{"timestamp":"2021-02-03T17:56:46.483+08:00","value":2}
{"timestamp":"2021-02-03T17:56:46.683+08:00","value":4}
{"timestamp":"2021-02-03T17:56:46.883+08:00","value":6}
{"timestamp":"2021-02-03T17:56:47.083+08:00","value":8}
{"timestamp":"2021-02-03T17:56:47.283+08:00","value":10}
{"timestamp":"2021-02-03T17:56:47.483+08:00","value":12}
{"timestamp":"2021-02-03T17:56:47.683+08:00","value":14}
{"timestamp":"2021-02-03T17:56:47.883+08:00","value":16}
{"timestamp":"2021-02-03T17:56:48.083+08:00","value":18}
```

part-00001-*.json：

```
{"timestamp":"2021-02-03T17:56:46.383+08:00","value":1}
{"timestamp":"2021-02-03T17:56:46.583+08:00","value":3}
{"timestamp":"2021-02-03T17:56:46.783+08:00","value":5}
{"timestamp":"2021-02-03T17:56:46.983+08:00","value":7}
{"timestamp":"2021-02-03T17:56:47.183+08:00","value":9}
{"timestamp":"2021-02-03T17:56:47.383+08:00","value":11}
{"timestamp":"2021-02-03T17:56:47.583+08:00","value":13}
{"timestamp":"2021-02-03T17:56:47.783+08:00","value":15}
{"timestamp":"2021-02-03T17:56:47.983+08:00","value":17}
{"timestamp":"2021-02-03T17:56:48.183+08:00","value":19}
```

7.6.2 使用 Kafka Data Sink

在结构化的流中，将流 DataFrame 的数据写入 Kafka 的 data sink，要比从 Kafka 的数据源中读取数据要简单得多。

Kafka 的 data sink 可以配置为下表中列出的四个选项：

Option	值	描述
kafka.bootstrap.servers	host1:port1 host2:port2	Kafka服务器列表，用逗号分隔
topic	字符串,如"topic1"	这是单个的主题(topic)名称
key	一个字符串, 或二进制	这个key用来决定一个Kafka消息应该被发送到哪个分区。所有具有相同key的Kafka消息将被发送到同一分区。这是一个可选项。
value	一个字符串, 或二进制	这是消息的内容。对于Kafka，它只是一个字节数组，对Kafka没有任何意义

其中有三个选项是必需的。重点要理解的是 key 和 value，它们与 Kafka 消息的结构有关。正如前面提到的，Kafka 的数据单元是一个消息，本质上是一个 key-value 对。这个 value 的作用就是保存消息的实际内容，而它对 Kafka 没有任何意义。就 Kafka 而言，value 只是一堆字节。然而，key 被 Kafka 认为是一个元数据，它和 value 一起被保存在 Kafka 的信息中。当一个消息被发送到 Kafka 并且一个 key 被提供时，Kafka 将其作为一种路由机制来确定一个特定的 Kafka 消息应该被发送到哪一个分区，按照对

该 key 哈希并对 topic 的分区数求余。这意味着所有具有相同 key 的消息都将被路由到同一个分区。如果消息中没有提供 key，那么 Kafka 就不能保证消息被发送到哪个分区，而 Kafka 使用了一个循环算法来平衡分区之间的消息。

提供 topic 主题名称有两种方法。第一种方法是在设置 Kafka data sink 时在配置中提供主题名称，第二种方法是在流 DataFrame 中定义一个名为 topic 的列，该列的值将用作 topic 主题的名称。

如果名为 key 的列存在于流 DataFrame 中，那么该列的值将用作消息的 key。因为该 key 是一个可选的元数据，所以在流 DataFrame 中不是必须有这一列。

另一方面，必须提供 value 值，而 Kafka 的 data sink 则期望在流 DataFrame 中有一个名为 value 的列。

Column	Type
key (optional)	string or binary
value (required)	string or binary
topic (*optional)	string

如果要开发以 Kafka 为 Data Sink 的 Spark 结构化流处理程序，必须添加 Kafka 的依赖包到 classpath 中。

如果我们要从 spark shell 使用 Kafka 数据源，那么需要在启动 spark shell 时将依赖的 JAR 包添加到 classpath 中。有两种方式可以做到：

- 手动将依赖包添加到 classpath。

首先将 spark-sql-kafka-0-10_2.11-2.4.7.jar 和 kafka-2.4.1.jar 包拷贝到~/bigdata/spark-2.4.7/jars/目录下。然后启动 spark-shell：

```
$ cd ~/bigdata/spark-2.4.7
$ ./spark-shell --master spark://localhost:7077
```

- 使用 package 参数让 Spark 为我们下载这些文件：(具体版本号要参阅官网)

```
$ cd ~/bigdata/spark-2.4.7
$ ./spark-shell --master spark://localhost:7077 --packages
org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.7,org.apache.kafka:kafka_2.11:2.4.1
```

如果是使用 IDE 开发并使用 SBT 来管理依赖，则需要在 build.sbt 中添加如下依赖配置以使用 Kafka 数据源。

```
libraryDependencies += "org.apache.spark" %% "spark-sql-kafka-0-10" % "2.4.7"
libraryDependencies += "org.apache.kafka" %% "kafka" % "2.4.1"
```

如果是使用 IDE 开发并使用 Maven 来管理依赖，则需要在 pom.xml 中添加如下依赖配置以使用 Kafka 数据源。

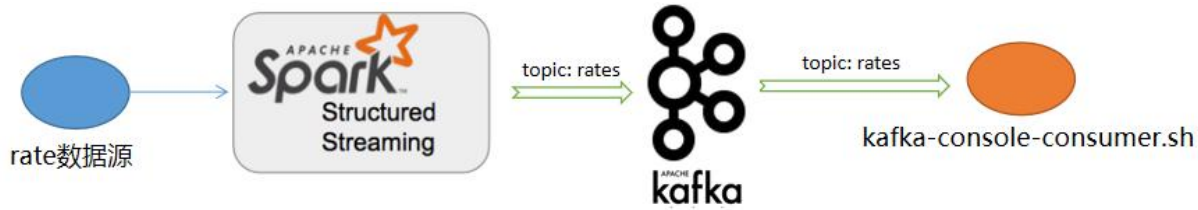
```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql-kafka-0-10_2.11</artifactId>
    <version>2.4.7</version>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka</artifactId>
    <version>2.4.1</version>
```

```
</dependency>
```

下面我们编写一个 Spark 结构化流应用程序，它读取 Rate 数据源，将数据写到 Kafka 的指定主题中。

【示例】编写 Spark 结构化流应用程序作为 Kafka 的生产者，将从 Rate 数据源读取的消息写入到 Kafka 的“rates”主题中。

在这个示例中，Spark 结构化流程序会向 Kafka 的“rates”主题发送消息（本例为读取自 Rate 数据源的数据），我们用 Kafka 自带的消费者脚本程序订阅该主题。一旦它收到了订阅的消息，马上输出。程序处理流程如下图所示：



首先，我们编写 Spark 结构化流程序代码。实现如下：

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._
......

  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .master("local")
      .appName("kafka sink")
      .getOrCreate()

    // spark.sparkContext.setLogLevel("WARN")          // 设置日志级别

    // 以每秒 10 行的速度设置 Rate 数据源，并使用两个分区
    val ratesSinkDF = spark.readStream
      .format("rate")
      .option("rowsPerSecond","10")
      .option("numPartitions","2")
      .load()

    // 转换 ratesSinkDF 以创建一个"key"列和"value"列
    // value 列包含一个 JSON 字符串，该字符串包含两个字段：timestamp 和 value
    val ratesSinkForKafkaDF = ratesSinkDF
      .select(
        col("value").cast("string") as "key",
        to_json(struct("timestamp","value")) as "value"
      )

    // 设置一个流查询，使用 topic "rates"，将数据写到 Kafka
    val query = ratesSinkForKafkaDF.writeStream
      .outputMode("append")
      .format("kafka")
      .option("kafka.bootstrap.servers", "localhost:9092")
      .option("topic","rates")
```

```
        .option("checkpointLocation", "tmp/rates")
        .start()

    // 等待流程序结束
    query.awaitTermination()
  }
```

要运行这个流程序，请按以下步骤进行操作：

1）启动 zookeeper 服务

Kafka 依赖于 Apache ZooKeeper，所以在启动 Kafka 之前，要先启动它。

打开一个终端窗口，执行如下命令：

```
$ cd ~/bigdata/kafka_2.11-2.4.1
$ ./bin/zookeeper-server-start.sh config/zookeeper.properties
```

等待 30 秒左右 ZooKeeper 启动。

2）接下来，启动 Kafka 服务器。

另打开一个终端窗口，执行如下命令：

```
$ cd ~/bigdata/kafka_2.11-2.4.1
$ ./bin/kafka-server-start.sh config/server.properties
```

等待 30 秒左右 Kafka 启动。

3）查看和创建 Kafka 主题(如果已经有了 rates 主题，则此步略过)。

另外打开第三个终端窗口，执行如下命令，创建 test 主题：

```
$ cd ~/bigdata/kafka_2.11-2.4.1
$ ./bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic rates
```

查看已有的主题，使用如下的命令：

```
$ ./bin/kafka-topics.sh --list --zookeeper localhost:2181
```

4）在第三个终端窗口，运行 Kafka 自带的消费者脚本，订阅“rates”主题消息。

```
$ ./bin/kafka-console-consumer.sh --bootstrap-server 192.168.190.145:9092 --topic rates
```

5）启动上面的流处理程序。

6）回到第三个终端窗口（即运行消费者脚本的窗口），如果一切正常，应该可以看到在终端输出收到的订阅消息，如下（部分）：

```
{"timestamp":"2021-02-03T19:14:59.352+08:00","value":1960}
{"timestamp":"2021-02-03T19:14:59.552+08:00","value":1962}
{"timestamp":"2021-02-03T19:14:59.752+08:00","value":1964}
{"timestamp":"2021-02-03T19:14:59.952+08:00","value":1966}
{"timestamp":"2021-02-03T19:15:00.152+08:00","value":1968}
{"timestamp":"2021-02-03T19:14:59.452+08:00","value":1961}
{"timestamp":"2021-02-03T19:14:59.652+08:00","value":1963}
{"timestamp":"2021-02-03T19:14:59.852+08:00","value":1965}
.....
```

7.6.3 使用 Foreach Data Sink

与结构化流提供的其他内置 data sinks 相比，foreach data sink 是一个很有意思的数据接收器，因为它根据数据应该如何被写出、何时写出数据以及将数据写入何处，提供了完整的灵活性，但这种灵活性和可扩展性是有要求的，即由我们自己来负责在使用这个数据接收器时写出数据的逻辑。

要使用 foreach 接收器，必须实现 ForeachWriter 接口（注意，该接口只支持 Scala/Java），它包含三个方法：open、process 和 close。只要有一个触发器的输出生成一系列的行，这些方法就会被调用。下面是使用这个 data sink 的一些细节。

- ForeachWriter 抽象类实现的一个实例将在驱动程序端被创建，它将被发送到 Spark 集群中的 executors 执行。这有两个条件。首先，ForeachWriter 的实现必须是可序列化的，否则，它的实例不能通过网络发送到 executors。第二，如果在创建过程中有任何初始化都将发生在驱动程序端。因此，如果想打开一个数据库连接或套接字连接，那就不应该在类初始化期间发生，而是在其他地方。
- 在流 DataFrame 中分区的数量决定了有多少个 ForeachWriter 实现的实例被创建。这类似于 Dataset.foreachPartition 方法。
- 在 ForeachWriter 抽象类中定义的三类方法将在 executors 上被调用。
- 执行初始化（例如打开数据库连接或套接字连接）的最佳位置，是在 open 方法中。然而，每当有数据被写出来时，就会调用 open 方法；因此，这种逻辑必须是智能和高效的。
- open 方法签名有两个输入参数：分区 ID 和版本。返回类型是布尔型。这两个参数的组合惟一地表示一组需要被写出来的行。这个版本的值是一个单调递增的 ID，随着每个触发器的增加而增加。根据分区 ID 的值和版本参数，open 方法需要决定它是否需要写出行序列，并将适当的布尔值返回到结构流引擎。
- 如果 open 方法返回 true，那么对于触发器输出的每一行就会调用 process 方法。
- 无论何时调用 open 方法，不管它返回什么值，close 方法也都会被调用。如果在调用 process 方法时出现错误，则该错误将被传递到 close 方法中。调用 close 方法的目的是给用户一个机会来清理在 open 或 process 方法调用期间创建的任何必要状态。只有当 executor 的 JVM 崩溃或者 open 方法抛出一个 Throwable 异常时，才不会调用 close 方法。

简而言之，这个 data sink 为我们提供了一个在写出流 DataFrame 的数据时足够的灵活性。下面通过一个示例来演示如何使用这种类型的 Data Sink。

【示例】编写 Spark 结构化流应用程序，通过将 Rate 数据源中的数据写入控制台，包含了一个 ForeachWriter 抽象类的简单实现。

首先导入依赖的类：

```
import org.apache.spark.sql.{ForeachWriter, Row, SparkSession}
```

接下来定义一个 ForeachWriter 抽象类的实现，并实现它的三个方法：open、process 和 close。

```
// 自定义一个 ConsoleWriter，继承自 ForeachWriter
class ConsoleWriter(private var pId:Long = 0, private var ver:Long = 0) extends ForeachWriter[Row] {

  // 初始化方法
  def open(partitionId: Long, version: Long): Boolean = {
    pId = partitionId        // 分区 id
    ver = version            // 版本号
    println(s"open => ($partitionId, $version)")
    true
  }

  // 业务处理方法
  def process(row: Row): Unit = {
    println(s"writing => $row")
  }
```

```
    // 做一些清理性的工作
    def close(errorOrNull: Throwable): Unit = {
      println(s"close => ($pId, $ver)")
    }
  }
```

然后，在流数据写出时，指定 forach 方法使用上面自定义的 ConsoleWriter。

```
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .master("local")
      .appName("foreach sink")
      .getOrCreate()

    // spark.sparkContext.setLogLevel("WARN")          // 设置日志级别

    // 以每秒 10 行的速度设置 Rate 数据源，并使用两个分区
    val ratesSourceDF = spark.readStream
      .format("rate")
      .option("rowsPerSecond","10")
      .option("numPartitions","2")
      .load()

    // 设置 Foreach data sink
    val query = ratesSourceDF.writeStream
      .foreach(new ConsoleWriter)
      .start()

    // 等待流程序结束
    query.awaitTermination()
  }
```

当开始执行时，可以看到控制台的输出，类似下面这样：

```
open => (0, 1)
writing => [2021-02-03 19:52:53.194,0]
writing => [2021-02-03 19:52:53.394,2]
writing => [2021-02-03 19:52:53.594,4]
writing => [2021-02-03 19:52:53.794,6]
writing => [2021-02-03 19:52:53.994,8]
close => (0, 1)
open => (1, 1)
writing => [2021-02-03 19:52:53.294,1]
writing => [2021-02-03 19:52:53.494,3]
writing => [2021-02-03 19:52:53.694,5]
writing => [2021-02-03 19:52:53.894,7]
writing => [2021-02-03 19:52:54.094,9]
close => (1, 1)
open => (0, 2)
writing => [2021-02-03 19:52:54.194,10]
writing => [2021-02-03 19:52:54.394,12]
writing => [2021-02-03 19:52:54.594,14]
writing => [2021-02-03 19:52:54.794,16]
```

```
writing => [2021-02-03 19:52:54.994,18]
close => (0, 2)
open => (1, 2)
writing => [2021-02-03 19:52:54.294,11]
writing => [2021-02-03 19:52:54.494,13]
writing => [2021-02-03 19:52:54.694,15]
writing => [2021-02-03 19:52:54.894,17]
writing => [2021-02-03 19:52:55.094,19]
close => (1, 2)
......
```

7.6.4 使用 Console Data Sink

这个数据接收器非常简单，但它不是一个容错的 data sink。它主要用于学习和测试，不能在生产环境下使用。它只有两种选项配置：要显示的行数，以及输出太长时是否截断。这些选项都有一个默认值，如下表所示。这个数据接收器的底层实现使用与 DataFrame.show 方法相同的逻辑来显示流 DataFrame 中的数据。

Option	默认值	描述
numRows	20	在控制台输出的行的数量
truncate	true	当每一行的内容超过20个字符时，是否截断显示

下面通过一个示例来了解这些 option 参数的用法。

【示例】编写 Spark 结构化流程序，读取 rate 数据源流数据，并将流数据处理结果写出到控制台，每次输出不超过 30 行。

代码实现如下。

```
def main(args: Array[String]): Unit = {

  val spark = SparkSession.builder()
    .master("local")
    .appName("file sink")
    .getOrCreate()

  // spark.sparkContext.setLogLevel("INFO")        // 设置日志级别

  // Rate 数据源读出数据
  val rateSourceDF = spark.readStream
    .format("rate")
    .option("rowsPerSecond","10")                  // 每秒产生 10 条数据
    .option("numPartitions","2")                   // 两个分区
    .load()

  // 将结果 DataFrame 写出到控制台
  val query = rateSourceDF.writeStream
    .outputMode("append")
    .format("console")                             // console data sink
    .option("truncate",value = false)              // 不截断显示
    .option("numRows",30)                          // 每次输出 30 行
    .start()
```

```
    // 等待流程序结束
    query.awaitTermination()
  }
```

执行上面的程序，输出结果如下：

```
-------------------------------------------
Batch: 1
-------------------------------------------
+----------------------+-----+
|timestamp             |value|
+----------------------+-----+
|2021-02-03 20:07:36.987|0    |
|2021-02-03 20:07:37.187|2    |
|2021-02-03 20:07:37.387|4    |
|2021-02-03 20:07:37.587|6    |
|2021-02-03 20:07:37.787|8    |
|2021-02-03 20:07:37.087|1    |
|2021-02-03 20:07:37.287|3    |
|2021-02-03 20:07:37.487|5    |
|2021-02-03 20:07:37.687|7    |
|2021-02-03 20:07:37.887|9    |
+----------------------+-----+
.....
```

7.6.5 使用 Memory Data Sink

与 Console data sink 类似，这个数据接收器也很容易理解和使用。事实上，它非常简单，不需要任何配置。它也不是一个容错的 data sink，主要用于学习和测试，不在生产环境中使用。它收集的数据被发送给 Driver，并作为内存中的表存储在 Driver 中。换句话说，可以发送到 Memory data sink 的数据量是由 Driver 中 JVM 拥有的内存大小决定的。在设置这个 data sink 时，可以指定一个查询名称作为 DataStreamWriter.queryName 函数参数，然后就可以对内存中的表发出 SQL 查询。与 Console data sink 不同的是，一旦数据被发送到内存中的表，就可以使用几乎所有在 Spark SQL 组件中可用的特性进一步分析或处理数据。（如果数据量很大，并且不适合内存，那么最好的选择就是使用 File data sink 以 Parquet 格式来写出数据）

下面通过一个示例来了解 Memory 数据接收器的用法。

【示例】编写 Spark 结构化流程序，读取 rate 数据源流数据，并流数据处理结果写出到内存表中，然后对该内存表发出。

代码实现如下。

```

```

注：上面的代码如下在 IDEA 中运行，可能会有问题。可尝试在 spark-shell 中执行以查看效果。

需要注意的一点是，即使在流查询 ratesSQ 停止之后，内存中的 rates 仍然会存在。然而，一旦一个新的流查询以相同的名称开始，那么来自内存中的数据就会被截断。

7.6.6 Data Sink 与输出模式

在了解了有哪些 data sink 之后，还有很重要的一点是要了解每种类型的 data sink 支持哪些输出。关于 data sink 和所支持的输出模式，请参考下表（关于输出模式的详细信息将在下一节中介绍）：

sink	支持的输出模式	备注
File	Append	只支持写出新行，没有更新
Kafka	Append, Update, Complete	
Foreach	Append, Update, Complete	依赖于ForeachWriter实现
Console	Append, Update, Complete	
Memory	Append, Complete	不支持 in-place 更新

7.7 深入研究输出模式

在 Spark 结构化流中，输出模式可以是 complete、update 或 append，其中 complete 输出模式意味着每次都要写入全部的结果表，update 输出模式写入从上批处理中已更改的行，而 append 输出模式仅写入新行。

一般来说，有两种类型的流查询。

- 第一种类型称为“无状态类型”，它只对流入的流数据进行基本的转换，然后将数据写到一个 data sink 上。
- 第二种类型称为“有状态类型”，它需要保持一定数量的状态，不管它是隐式还是显式地完成。

有状态类型通常执行某种聚合或使用像 mapGroupsWithState 或 flatMapGroupsWithState 这样的结构化流 API，可以维护特定用例所需的任意状态，例如，维护用户会话数据。

7.7.1 无状态流查询

无状态流查询的典型用例是实时流 ETL，它可以连续读取实时流数据，比如在线服务连续生成的 PV 事件，以捕获哪些页面正在被哪些用户浏览。在这种用例中，它通常执行以下操作：

- 过滤、转换和清洗
  - 真实世界的数据是混乱和肮脏的，而且这种结构可能不太适合重复分析。
- 转换为更有效的存储格式
  - 像 CVS 和 JSON 这样的文本文件格式易读性虽然好，但对于重复分析来说是低效的，特别是如果数据量很大，比如几百 TB 字节。更有效的二进制格式，如 PRC、Parquet 或 Avro，通常用于减少文件大小和提高分析速度。
- 按某些列划分数据
  - 在将数据写到 data sink 时，可以根据常用列的值对数据进行分区，以加快组织中不同团队的重复分析。

以前的任务在将数据写到 data sink 之前不需要流查询来维护任何类型的状态。随着新数据的出现，它被清理、转换，并可能进行重组，并立即被写出。因此，这种**无状态流类型的唯一适用的输出模式是 Append**。Complete 输出模式是不适用的，因为这需要结构化的流来维护所有以前的数据，这些数据可能太大而无法维护。Update 输出模式也不适用，因为只有新数据被写出来。然而，当这种 update 输出模式被用于无状态流查询时，结构化流就会识别这个并将其与 Append 输出模式相同对待。当不适当的输出模式用于流查询时，结构化的流引擎会抛出异常。

下面的代码展示了在使用不适当的输出模式时会发生什么情况：

```
```

当提交以上代码执行时，我们会收到如下这样的异常信息：

```
// 在分析阶段来自结构化流的一个异常
```

```
org.apache.spark.sql.AnalysisException: Complete output mode not supported
when there are no streaming aggregations on streaming DataFrames/Datasets;
```

7.7.2 有状态流查询

当一个有状态的流查询通过一个 group by 转换执行一个聚合时，这个聚合的状态是由结构化的流引擎隐式地维护的。随着更多的数据到达，新数据聚合的结果被更新到结果表中。在每个触发点上，根据输出模式，更新后的数据或结果表中的所有数据都被写到一个 data sink 上。这意味着使用 Append 输出模式是不合适的，因为这违反了输出模式的语义，该模式指定只有附加到结果表的新行将被发送到指定的输出接收器。换句话说，**只有 Complete 和 Update 输出模式适合于有状态查询类型**，并且聚合状态隐式地由结构化流引擎负责维护。使用 Complete 输出模式的流查询的输出总是等于或超过使用 Update 输出模式的相同流查询的输出。

下面的示例用来说明 Update 和 Complete 模式之间的输出差异。

【示例】移动电话的开关机等事件会保存在 json 格式的文件中。现在编写 Spark 结构化流处理程序来读取这些事件并统计不同的 action 发生的数量。请按以下步骤操作。

1）准备数据

在本示例中，我们使用文件数据源，该数据源以 json 文件的格式记录了一小组移动电话动作事件。每个事件由三个字段组成：

- id：表示手机的唯一 ID。在样例数据集中，电话 ID 将类似于 phone1、phone2、phone3 等。
- action：表示用户所采取的操作。该操作的可能值是"open"或"close"。
- ts：表示用户 action 发生时的时间戳。这是事件时间(event time)。

我们准备了两个存储移动电话事件数据的 JSON 文件，两个文件的内容如下：

action.json

```
{"id":"phone1","action":"open","ts":"2018-03-02T10:02:33"}
{"id":"phone2","action":"open","ts":"2018-03-02T10:03:35"}
{"id":"phone3","action":"open","ts":"2018-03-02T10:03:50"}
{"id":"phone1","action":"close","ts":"2018-03-02T10:04:35"}
{"id":"phone3","action":"close","ts":"2018-03-02T10:07:35"}
```

newaction.json

```
{"id":"phone4","action":"open","ts":"2018-03-02T10:07:50"}
{"id":"phone2","action":"crash","ts":"2018-03-02T11:09:13"}
{"id":"phone5","action":"swipe","ts":"2018-03-02T11:17:29"}
```

注意这两个数据文件中，action 字段值的区别。在 action.json 文件中，包含两类 action 值，分别为“open”和“close”，而在 newaction.json 文件中，包含三类 action 值，分别为“open”、“crash”和“swipe”。

为了模拟数据流的行为，我们将把这两个 JSON 文件复制到项目的“src/main/resources/mobile2”目录下。

编写 Spark 结构化流程序，读取文件流数据源并执行聚合操作，然后输出结果。这里我们使用的输入模式是“complete”。实现代码如下：

```

```

执行上面的代码，流查询输出如下：

```

```

观察上面的输出结果，在“Batch 1”中输出的聚合结果，包含了“Batch 0”中的状态。这说明在上面代码中，带有“complete”输出模式的流查询的输出包含了结果表中的所有 action 类型。

接下来，将上面代码中的输出改为使用“update”输出模式，其余代码保持不变：

执行上面的代码，流查询输出如下：

观察上面的输出结果，在“Batch 1”中输出的聚合结果，并不包含“Batch 0”中的状态。这说明在上面代码中，带有“update”输出模式的流查询的输出只包含文件 newaction.json 中的 actions，这些 actions 结果（包含更新的 open action）以前从未出现过。

同样的，如果有状态查询类型使用了不适当的输出模式，那么结构化的流引擎会抛出异常信息。例如，我们继续修改上面的代码，将输出模式设为“append”：

因为执行的有“groupBy”聚合操作，所以会产生下面这样的异常：

```
org.apache.spark.sql.AnalysisException: Append output mode not supported when there are streaming aggregations on streaming DataFrames/DataSets without watermark;
```

也有一个例外情况：如果向带有聚合的有状态的流查询提供一个水印，那么所有的输出模式都是适用的。这是因为结构化的流引擎将删除旧的聚合状态数据，这些数据比指定的水印要“老”，这意味着一旦水印被逾越，新的行就可以被添加到结果表中。这时 Append 输出的语义是有意义的。

7.8 深入研究触发器

触发器设置决定了结构化的流引擎何时运行在流查询中表达的流计算逻辑，其中包括所有的转换，以及将数据写入到 data sink。换句话说，触发器设置控制什么时候数据被写到 data sink 上，以及使用哪种处理模式。从 Spark 2.3 开始，引入了一种称为“连续的（Continuous）”新处理模式。

到目前为止，我们所有的流查询示例都没有指定触发器类型，而是使用了默认触发器类型，因为。**这个默认的触发器类型选择微批模式作为处理模式**，而流查询中的逻辑执行不是基于时间的，而是在前一批数据完成处理后立即执行的。这意味着，在数据被写入的频率方面，可预测性会降低。

如果需要更强的可预测性，那么可以指定固定的间隔触发器。示例：

7.8.1 固定间隔触发器

固定间隔触发器可以根据用户提供的时间间隔，例如，每 30 秒，在特定的时间间隔内执行流查询中的逻辑。在处理模式方面，这个触发器类型使用微批处理模式。这个间隔可以用字符串格式指定，也可以作为 Scala Duration 或 Java TimeUnit 来指定。下面的代码包含了使用固定间隔触发器的示例。

提交执行上面的代码，可以得到类似下面这样的输出结果：

因为我们指定 rate 数据源每秒产生 2 行，而触发器指示每 3 秒钟计算一批，所以可以看到每 3 秒有 6 行输出。

也可以使用 Scala Duration 类型指定触发器的时间间隔，如下所示：

固定间隔触发器并不总是保证流查询的执行会精确地在每个用户指定的时间间隔内发生。这有两个原因：

- 第一个原因很明显，如果没有数据到达处理，那么就没有什么可处理的，因此没有任何东西被写入到 data sink 中。
- 第二个原因是，当前一批的处理时间超过指定间隔时间时，流查询的下一个执行将在处理完成后立即启动。换句话说，它不会等待下一个时间间隔边界。

7.8.2 一次性的触发器

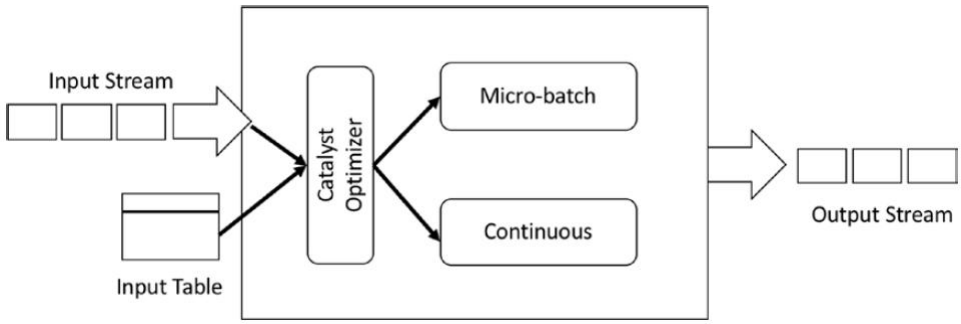
顾名思义，一次性触发器以微批处理模式在流查询中执行逻辑，并将数据写到 data sink 一次，然后处理停止。这种触发类型的存在，在开发和生产环境中都很有用。在开发阶段，通常流计算逻辑是以迭代的方式开发的，在每个迭代中，我们都希望测试逻辑。这个触发器类型简化了开发-测试-迭代。对于生产环境，这种触发器类型适合于流入流数据量较低的情况，这时只需要每天运行几次数据处理逻辑。

指定这个一次性触发器类型非常简单。下面的代码演示了如何使用一次性的触发器类型：

7.8.3 连续性的触发器

最后一个触发类型称为**连续（Continuous）触发类型**。这是在 Spark 2.3 中新引入的实验性的处理模式，以解决需要端到端的毫秒级延迟的情况。连续处理是 Apache Spark 的新执行引擎，每次处理事件的延迟非常低(以毫秒为单位)。在这个新的处理模式中，结构化流启动长时间运行的任务，以持续读取、处理和写入数据到一个 data sink。这意味着，一旦传入的数据到达数据源，它就会立即被处理并写入到 data sink，则端到端延迟是几毫秒。此外，还引入了一个异步检查点机制，用于记录流查询的进度，以避免中断长时间运行的任务，从而提供一致的毫秒级延迟。

利用这个 Continuous 触发器类型的一个比较好的案例是信用卡欺诈性交易检测。在较高的层次上，结构流引擎根据触发器类型确定要使用哪种处理模式，如下图所示。



Spark 一直通过微批处理提供流处理能力，这种方法的主要缺点是每个任务/微批处理必须定期收集和调度，通过这种方式 Spark 可以提供的最佳(最小)延迟大约是 1 秒。不存在单一事件/消息处理的概念。Spark 试图通过连续处理来克服这些限制，以提供低延迟的流处理。

为了启用这些特性，Spark 对其底层代码进行两项主要更改。

- 创建可以连续读取消息(而不是微批处理)的新数据源和数据接收器—称为 DataSourceV2。
- 创建一个新的执行引擎- ContinuousProcessing，它使用 ContinuousTrigger 并使用 DataSourceV2 启动长运行任务。ContinuousTrigger 内部使用 ProcessingTimeExecutor(与 ProcessingTime 触发器相同)。

DataSource V2

DataSource V2 具有一次读写记录的能力。例如，KafkaSource 有 get()和 next()方法来读取每条记录，而不是 V1 中的 getBatch()方法（注意：即使每次读取一个记录，kafkconsumer 仍然会有一些缓冲）。KafkaSink 持续运行，等待新记录提交到主题，并且一次写入/提交记录。

目前支持的读取数据源有：

- KafkaSource(简称 kafka)
- RateSource(简称 rate) - 仅用于测试目的

目前支持的写出 Sink 有：

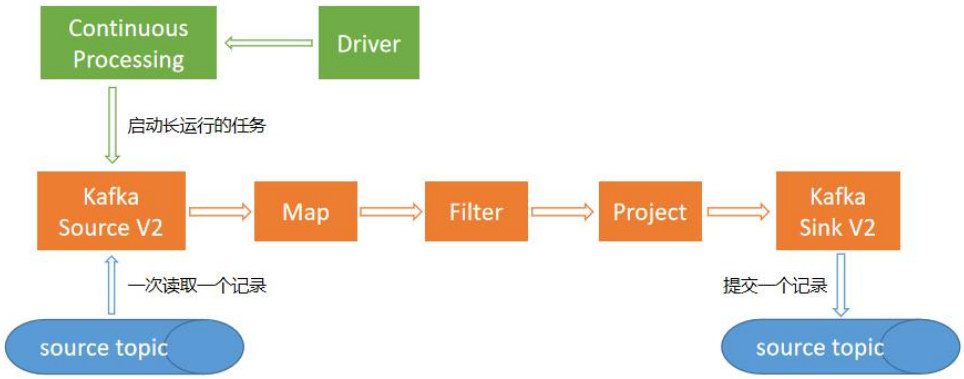
- KafkaSink
- ConsoleSink - 仅用于测试目的
- MemorySink - 仅用于测试目的

ContinuousExecution 引擎

这是允许在 Spark 中进行低延迟处理的第二个主要更改。当触发器集为 ContinuousTrigger 时(数据源和接收器也应该是 DataSourceV2 类型)，选择 ContinuousExeuction 引擎作为 StreamExecution。该引擎支持的操作目前是有限的，它主要支持映射、筛选和投影。不支持聚合、连接、窗口等操作。这背后的想法是，对于这样的操作，我们需要等待一段时间来收集数据，在那些用例中，基于微批处理的引擎就足够了。需要非常低延迟(以毫秒为单位)的用例适合这个连续模型。

在 Spark 2.3 中，在持续处理模式中只有投影和选择操作是允许的，如 select、where、map、flatmap 和 filter。在这种处理模式下，除了聚合函数之外，所有 Spark SQL 函数都是受支持的。另外需要特别注意的是，在连续处理中不支持水印，因为这涉及到收集数据。

例如，在使用连续流读取 Kakfa 时，其执行过程如下图所示：



要使用流查询的连续处理模式，需要指定一个连续触发器（Continuous trigger），其中包含一个期望的检查点间隔，如下面的代码所示。

提交执行以上代码，可以在控制台上看到类似下面这样的输出内容：

上面代码中的 ratesDF streaming DataFrame 设置为两个分区；因此，结构化流在连续处理模式下启动了两个正在运行的任务，所以输出显示所有的偶数在一起，所有奇数出现在一起。

7.9 理解结构化流执行机制

在 DataFrames/Datasets 上执行的操作的执行机制如下图所示：

为了启用结构化流功能，Planner 轮询来自源的新数据，并在将其写入接收器之前递增地对其执行计算（请注意，Planner 知道如何将流逻辑计划转换为连续的递增执行计划）。此外，应用程序所需的任何正在运行的聚合都作为由预写日志(WAL)支持的内存状态进行维护。内存中的状态数据是在增量执行中生成和使用的。这些应用程序的容错要求包括能够恢复和重演系统中的所有数据和元数据。Planner 在执行之前将偏移量写入容错的 WAL 持久存储，如 HDFS，如图所示:

如果 Planner 在当前增量执行中失败，重新启动的 Planner 将从 WAL 读取并重新执行所需偏移量的精确范围。通常，Kafka 等源也是容错的，并生成原始事务数据，给出由 Planner 恢复的适当的偏移量。状态数据通常保存在 Spark workers 中一个版本化的键值映射中，并由 HDFS 上的 WAL 所支持。Planner 确保在失败后使用状态的正确版本重新执行事务。此外，sinks 的设计是幂等的，并且可以在不重复提交输出的情况下处理重新执行。因此，WAL 中的偏移跟踪、状态管理和容错源和 sinks 提供了端到端精确一次的保证。

我们可以使用 explain 方法列出结构化流的物理计划，如下所示:

```
spark.streams.active(0).explain
```

第 8 章　Spark 结构化流（高级）

前一章介绍了流处理的核心概念，Spark 结构化流处理引擎提供的特性，以及将流应用程序组合在一起的基本步骤。本章将涵盖结构化流的事件时间（event-time）处理和有状态处理特性，并解释结构化流提供的支持，以帮助流应用程序对故障进行容错，并监控流应用程序的状态和进展。

8.1 事件时间和窗口聚合

基于数据创建时间处理传入的实时数据的能力是一个优秀的流处理引擎的必备功能。这一点很重要，因为要真正理解并准确地从流数据中提取见解或模式，需要能够根据数据或事件发生的时间来处理它们。

8.1.1 固定窗口聚合

一个固定的窗口（也就是一个滚动的窗口）操作本质上是根据一个固定的窗口长度将一个流入的数据流离散到非重叠的桶中。对于每一片输入的数据，根据它的事件时间（event time）将它放置到其中一个桶中。执行聚合仅仅是遍历每个桶并在每个桶上应用聚合逻辑（例如计数或求和）。下图说明了固定窗口聚合逻辑。

下面我们通过一个示例程序来演示如何使用结构化流读取文件数据源。

【示例】移动电话的开关机等事件会保存在 json 格式的文件中。现要求编写 Spark 结构化流处理程序来读取并分析这些移动电话数据，统计每 10 分钟内不同电话操作（如 open 或 close）发生的数量。

这实际上是在一个 10 分钟长的固定窗口上对移动电话操作事件的数量进行 count 聚合。请按以下步骤操作。

1）准备数据

在本示例中，我们使用文件数据源，该数据源以 json 文件的格式记录了一小组移动电话动作事件。每个事件由三个字段组成：

- id：表示手机的唯一 ID。在样例数据集中，电话 ID 将类似于 phone1、phone2、phone3 等。
- action：表示用户所采取的操作。该操作的可能值是"open"或"close"。
- ts：表示用户 action 发生时的时间戳。这是事件时间(event time)。

我们准备了四个存储移动电话事件数据的 JSON 文件，四个文件的内容如下：

file1.json

```
{"id":"phone1","action":"open","ts":"2018-03-02T10:02:33"}
{"id":"phone2","action":"open","ts":"2018-03-02T10:03:35"}
{"id":"phone3","action":"open","ts":"2018-03-02T10:03:50"}
{"id":"phone1","action":"close","ts":"2018-03-02T10:04:35"}
```

file2.json

```
{"id":"phone3","action":"close","ts":"2018-03-02T10:07:35"}
{"id":"phone4","action":"open","ts":"2018-03-02T10:07:50"}
```

file3.json

```
{"id":"phone2","action":"close","ts":"2018-03-02T10:04:50"}
```

```
{"id":"phone5","action":"open","ts":"2018-03-02T10:10:50"}
```

newaction.json 文件内容：

```
{"id":"phone2","action":"crash","ts":"2018-03-02T11:09:13"}
{"id":"phone5","action":"swipe","ts":"2018-03-02T11:17:29"}
```

为了模拟数据流的行为，我们将把这四个 JSON 文件复制到项目的“src/main/resources/mobile”目录下。

2）编辑源代码，内容如下。

```

```

3）执行流处理程序，输出结果如下所示。

```

```

可以看出，当用窗口执行聚合时，输出窗口实际上是一个 struct 类型，它包含开始和结束时间。在上面的代码中，我们分别取 window 的 start 和 end 列，并按 start 窗口开始时间排序。可以看，每 10 分钟做为一个窗口进行统计。

3）除了在 groupBy 转换中指定一个窗口之外，还可以从事件本身指定额外的列。对上面的例子稍做修改，使用一个窗口并在 action 列上执行聚合，实现对每个窗口和该窗口中 action 类型的 count 计数。代码如下所示：

```

```

输出结果如下所示：

```

```

8.1.2 滑动窗口聚合

除了固定窗口类型之外，还有另一种称为滑动窗口（sliding window）的窗口类型。定义一个滑动窗口需要两个信息，窗口长度和一个滑动间隔，滑动间隔通常比窗口的长度要小。由于聚合计算在传入的数据流上滑动，因此结果通常比固定窗口类型的结果更平滑。因此，这种窗口类型通常用于计算移动平均。关于滑动窗口，需要注意的一点是，由于重叠的原因，一块数据可能会落入多个窗口，如下图所示。

下面通过一个示例程序使用和理解滑动窗口。

【示例】应用滑动窗口聚合解决一个 IOT 流数据分析需求：在一个数据中心中，按一定的时间间隔周期性地检测每个服务器机架的温度，并生成一个报告，显示每一个机架在窗口长度 10 分钟、滑动间隔 5 分钟的平均温度。

请按以下步骤操作。

1）准备数据

在本示例中，我们使用文件数据源，该数据源以 json 文件的格式记录了某数据中心两个机架的温度数据。每个事件由三个字段组成：

- rack：表示机器的唯一 ID，字符串类型。
- temperature：表示采集到的温度值，double 类型。
- ts：表示该事件发生时的时间戳。这是事件时间(event time)。

两个数据文件的内容如下：

file1.json：

```
{"rack":"rack1","temperature":99.5,"ts":"2017-06-02T08:01:01"}
{"rack":"rack1","temperature":100.5,"ts":"2017-06-02T08:06:02"}
{"rack":"rack1","temperature":101.0,"ts":"2017-06-02T08:11:03"}
```

```
{"rack":"rack1","temperature":102.0,"ts":"2017-06-02T08:16:04"}
```

file2.json：

```
{"rack":"rack2","temperature":99.5,"ts":"2017-06-02T08:01:02"}
{"rack":"rack2","temperature":105.5,"ts":"2017-06-02T08:06:04"}
{"rack":"rack2","temperature":104.0,"ts":"2017-06-02T08:11:06"}
{"rack":"rack2","temperature":108.0,"ts":"2017-06-02T08:16:08"}
```

为了模拟数据流的行为，我们将把这两个 JSON 文件复制到项目的“src/main/resources/iotd”目录下。

2）代码编写。

实现的流查询代码如下：

```
```

在上面的代码中，首先读取温度数据，然后在 ts 列上构造一个长 10 分钟、每 5 分钟进行滑动的滑动窗口，并在这个窗口上执行 groupBy 转换。对于每个滑动窗口，avg()函数被应用于 temperature 列。

3）执行以上代码，输出结果如下所示：

```
```

上面的输出显示在合成数据集中有 5 个窗口。注意每个窗口的开始时间间隔为 5 分钟，这是因为在 groupBy 转换中指定的滑动间隔的长度。

在上面的分析结果中，可以看出 avg_temp 列所代表的机架平均温度在上升。那么大家思考一下，机架平均温度的上长，是因为其中某个机架的温度升高从而导致平均温度的升高？还是所有机架的温度都在升高?

4）为了弄清楚到底是哪些机架在不断升温，我们重构上面的代码，把 rack 列添加到 groupBy 转换中，代码如下所示（只显示不同的代码部分）：

```
```

执行以上代码，输出结果如下所示：

```
```

从上面的输出结果表中，可以清楚地看出来，机架 1 的平均温度低于 103，而机架 2 升温的速度要远快于机架 1，所以应该关注的是机架 2。

8.2 水印

在流处理引擎中，水印是一种常用的技术，用于处理延迟数据，以及限制维护它所需的状态数量。

8.2.1 限制聚合状态数量

通过应用于事件时间上的窗口聚合（固定窗口聚合或滑动窗口聚合），在 Spark 结构化流中可以很容易地执行常见的和复杂的流处理操作。虽然表面上看似乎很容易，但在其内部，结构化流引擎和 Spark SQL 引擎协同工作，在执行流聚合时，以容错的方式维护中间聚合结果。

。。。。。。

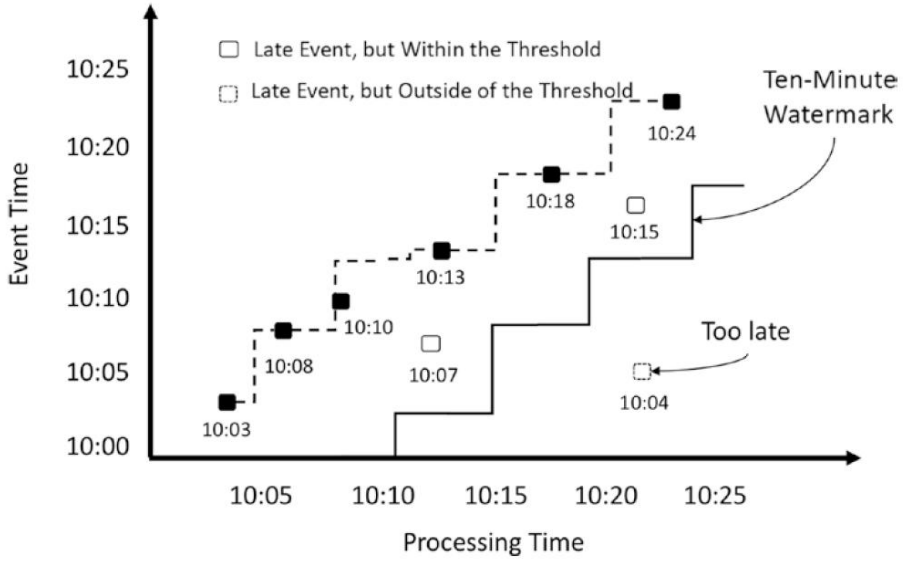
指定水印的最大好处之一是能让结构化流引擎可以安全地删除比水印更古老的窗口的聚合状态。**生产环境下执行任何类型聚合的流应用程序都应该指定一个水印来避免内存不足的问题**。

8.2.2 处理迟到的数据

在现实世界中，流数据往往会不按顺序到达，以及因为网络拥挤、网络中断或数据生成器（如移动

设备等）不在线而延迟到达。作为一个实时流应用程序的开发人员，必须要知道想要怎样处理比某个阈值晚一些的数据。换句话说，数据延迟到达时间量是多少时才是可以接受的，或者说对这个时间量之后迟到的数据置之不理？这取决于应用场景。

从结构化流的角度来看，水印是事件时间（event time）的移动阈值，它位于目前所见的最新事件时间之后。随着新事件不断到达，这将导致水印也不断移动。例如，在下面这张图中，水印被定义为 10 分钟。水印线由实线表示，它在最新事件时间线（由虚线表示）后面。每个矩形框代表一段数据，正文是其事件时间。其中事件时间 10:07 的那批数据虽然有点迟到（大约在 10:12 被处理）；然而，这仍然落在 10:03 和 10:13 之间的阈值上，也就是在 10 分钟的水印线之前。因此，它可以正常被处理。事件时间 10:15 的那批数据也是如此。但是，事件时间为 10:04 的那批数据到达非常晚，大约 10:22，落在了水印线之外，因此它将被忽略，而不会被处理。



要在结构化流中指定水印作为流 DataFrame 的一部分，只需要使用 Watermark API 并提供两个参数：事件时间列和阈值，这些数据可以是秒、分钟或小时。

【示例】使用水印来处理延迟到达的移动电话操作事件数据。

请按以下步骤操作。

1）准备数据

在本示例中，我们使用文件数据源，代表移动电话操作事件数据存储在两个 JSON 格式的文件中。每个事件由三个字段组成：

- id：表示手机的唯一 ID，字符串类型。
- action：表示用户所采取的操作。该操作的可能值是"open"或"close"。
- ts：表示用户 action 发生时的时间戳。这是事件时间(event time)。

两个数据文件的内容如下：

file1.json：

```
{"id":"phone1","action":"open","ts":"2018-03-02T10:15:33"}
{"id":"phone2","action":"open","ts":"2018-03-02T10:22:35"}
{"id":"phone3","action":"open","ts":"2018-03-02T10:33:50"}
```

file2.json：代表迟到的数据。

```
{"id":"phone4","action":"open","ts":"2018-03-02T10:29:35"}
{"id":"phone5","action":"open","ts":"2018-03-02T10:11:35"}
```

注意观察这两个数据文件中的数据。数据以这样一种方式设置，即 file1.json 文件中的每一行进入了它自己的 10 分钟窗口，那么 file1.json 的处理会形成三个窗口：10:10:00-10:20:00、10:20:00-10:30:00 和 10:30:00-10:40:00。我们指定水印为 10 分钟。file2.json 文件中的数据代表迟到的数据，其中第一行落在 10:20:00-10:30:00 窗口中，所以即使它到达的时间较晚，它的时间戳仍然在水印的阈值范围内，因此它将被处理。file2.json 文件中的最后一行数据的时间戳在 10:10:00-10:20:00 窗口中，由于它超出了水印的阈值，所以它将被忽略，而不会被处理。

为了模拟数据流的行为，我们将把这两个 JSON 文件复制到项目的“src/main/resources/mobile3”目录下。

2）代码编写。

实现的流查询代码如下：

3）执行程序，输出源数据的结构：

当它读取到第一个流数据文件 file1.json 时，输出结果如下。

正如期望的，每一行都落在它自己的窗口内。

当它读取到第一个流数据文件 file2.json 时，输出结果如下。

注意到窗口 10:20:00-10:30:00 的 count 现在被更新为 2，窗口 10:10:00- 10:20:00 没有变化。如前所述，因为 file2.json 文件中的最后一行的时间戳落在 10 分钟的水印阈值之外，因此它不会被处理。

4）如果删除对 Watermark API 的调用，那么输出结果如下所示。

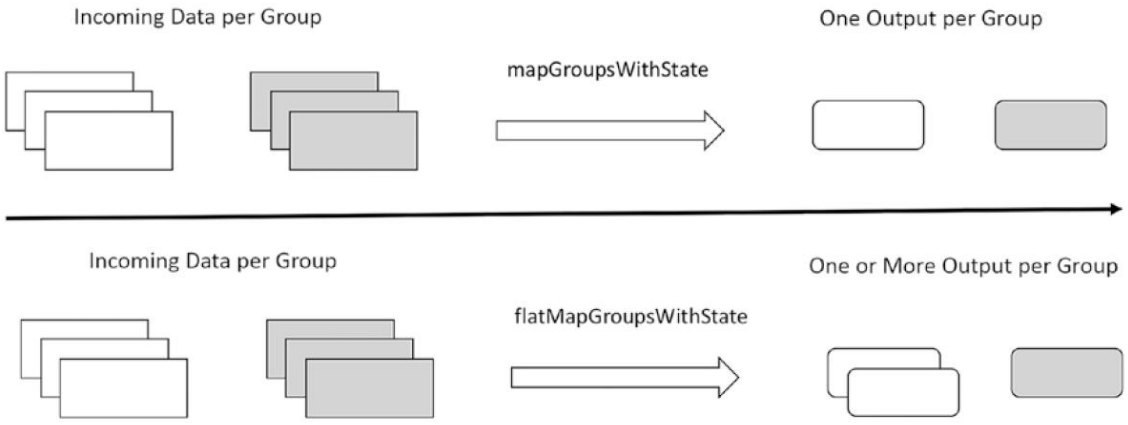
可以看出，因为没有指定水印，所以迟到的数据也不会被删除，所以对窗口 10:10:00- 10:20:00 的 count 计数被更新为 2。（思考：这样好吗？Spark 怎么知道迟到的数据会迟到多长时间？）

8.3 任意有状态处理

如前所述，按 key 或事件窗口聚合的中间状态由结构化流自动维护。然而，并不是所有的 event-time 处理都可以通过简单地在一个或多个列上聚合，并且在没有窗口的情况下得到满足。例如，在 IOT 实时温度监控程序中，当看到三个连续的温度读数超过 100 度时，需要发出一个警报。另一个例子是关于维护用户会话，其中每个会话的长度不是由固定的时间决定的，而是由用户的活动和缺乏活动决定的。要解决这两个示例和类似的应用场景，就需要能够在每组数据上应用任意处理逻辑，以控制每组数据的窗口长度，并在触发器点上保持任意状态。这就需要应用结构化流的任意状态处理。

8.3.1 结构化流的任意有状态处理

结构化流为流应用程序提供了一种回调机制来执行任意的有状态处理，并且它将负责确保中间状态的维护和以容错的方式存储。这种处理方式基本上可以归结为执行以下任务之一的能力，如下图所示：



这两个任务是：

- 映射一组数据，对每组数据应用任意处理，然后每组只输出一行。
- 映射一组数据，对每组数据进行任意处理，然后每组输出任意数量的行，包括 none。

对于其中每一个任务 task，结构化流都提供了一个特定的 API 来处理它。在第一个 API 中，API 被称为 mapGroupsWithState，而对于第二个，API 被称为 flatMapGroupsWithState。这些 API 从 Spark 2.2 开始引入，并且只使用 Scala 或 Java 绑定在类型化 Dataset API 上工作。

当使用任何类型的回调机制时，重要的是要清楚地了解何时以及多长时间调用它一次以及输入参数的详细信息。在这个特殊的例子中，顺序是这样的：

- 要在一个流 DataFrame 上执行任意的有状态处理，必须首先通过调用 groupByKey 转换并提供进行分组的列来指定分组；然后它返回 KeyValueGroupedDataset 类的一个实例。
- 从 KeyValueGroupedDataset 类的一个实例中，可以调用 mapGroupsWithState 或 flatMapGroupsWithState 函数。这两个 API 都需要一组不同的输入参数。
- 当调用 mapGroupsWithState 函数时，需要提供超时类型和用户定义的回调函数。超时部分将在稍后解释。
- 当调用 flatMapGroupsWithState 函数时，需要提供一个输出模式、超时类型和一个用户定义的回调函数。

8.3.2 处理状态超时

在带有水印的事件时间聚合的情况下，中间状态的超时是由结构化流内部管理的，我们无法来影响它。而结构化流任意状态处理提供了控制中间状态超时的灵活性。由于有能力维护任意状态，所以对于某些特定的用例来说，控制中间状态超时是有意义的。

结构化流有状态处理提供了三种不同的超时类型。

第一个是基于处理时间，另一个基于事件时间。超时类型是在全局级别配置的，这意味着它适用于特定的流 DataFrame 中的所有 groups。可以为每个单独的组配置超时值，并且可以随意更改。如果中间状态被配置了超时，那么在处理回调函数中给定的值列表之前，检查它是否超时了，这是很重要的。

在某些场景中，不需要超时，第三个超时类型是为这个场景设计的。

下面的部分将通过几个示例来演示如何实现任意状态处理。

8.3.3 任意状态处理实战

本节将通过处理两个用例来演示结构化流中的任意状态处理。

- 第一个是关于从数据中心计算机机架温度数据中提取模式，并维护中间状态下每个机架的状态。每当遇到三个连续的 100 度及以上的温度时，机架状态将升级到 warning（警告）级别。这个例子将使用 mapGroupsWithState API。
- 第二个例子是关于用户会话化的，它将根据用户与网站的交互来跟踪用户状态。这个例子将使用 flatMapGroupsWithState API。

无论哪种 API 将被用于为用例执行任意的状态处理，都需要一组通用的步骤，包括：

- 定义几个类来表示输入数据、中间状态和输出。
  - 输入事件(I)
  - 要保持的任意状态(S)
  - 输出(O)(如果合适，这种类型可能与状态表示相同)
- 定义状态转换函数。有了上面这些类型，我们就可以制定实现自定义状态处理逻辑的状态转换函数。需要定义两个状态转换函数，第一个是回调函数，由结构化流来调用。第二个函数包含在每组数据上执行的任意状态处理逻辑，以及维护状态的逻辑。
- 决定一个超时类型和一个适当的超时值。

【示例】数据中心计算机机架温度数据复杂事件模式识别和处理。

在本例中，我们感兴趣的模式是从同一个机架上采集到连续三个温度读数在 100 度或以上、并且两个连续高温读数之间的时间差必须在 60 秒内。当检测到这种模式时，该特定机架的状态将升级为 warning（警告）状态。如果下一个进入的温度读数低于 100 度阈值，那么机架状态就会降级为正常值。

请按以下步骤操作。

1）准备数据

在本示例中，我们使用文件数据源，事件数据存储在 JSON 格式的文件中。每个事件由三个字段组成：

- rack：表示机架的唯一 ID，字符串类型。
- temperature：表示采集到的温度值，double 类型。
- ts：表示事件发生时的时间戳。这是事件时间(event time)。

这里提供了三个数据文件。其中 file1.json 的内容显示机架 rack1 的温度，在 100 度上下交替变化。文件 file2.json 显示 rack2 的温度，连续升温。在文件 file3.json 中，rack3 也在升温，但温度读数超过一分钟的距离。这三个数据文件的内容如下。

file1.json：

```
{"rack":"rack1","temperature":99.5,"ts":"2017-06-02T08:01:01"}
{"rack":"rack1","temperature":100.5,"ts":"2017-06-02T08:02:02"}
{"rack":"rack1","temperature":98.3,"ts":"2017-06-02T08:02:29"}
{"rack":"rack1","temperature":102.0,"ts":"2017-06-02T08:02:44"}
```

file2.json：

```
{"rack":"rack1","temperature":97.5,"ts":"2017-06-02T08:02:59"}
{"rack":"rack2","temperature":99.5,"ts":"2017-06-02T08:03:02"}
{"rack":"rack2","temperature":105.5,"ts":"2017-06-02T08:03:44"}
{"rack":"rack2","temperature":104.0,"ts":"2017-06-02T08:04:06"}
```

```
{"rack":"rack2","temperature":108.0,"ts":"2017-06-02T08:04:49"}
```

file3.json：

```
{"rack":"rack2","temperature":108.0,"ts":"2017-06-02T08:06:40"}
{"rack":"rack3","temperature":100.5,"ts":"2017-06-02T08:06:20"}
{"rack":"rack3","temperature":103.7,"ts":"2017-06-02T08:07:35"}
{"rack":"rack3","temperature":105.3,"ts":"2017-06-02T08:08:53"}
```

为了模拟数据流的行为，我们将把这三个 JSON 文件复制到项目的“src/main/resources/iot3”目录下。

2）首先导入项目所依赖的包。

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.streaming.{GroupState, GroupStateTimeout, OutputMode}
import org.apache.spark.sql.types._
```

3）接下来，准备两个 case class。对于这个用例，机架温度输入数据由类 RackInfo 来表示（输入事件 I），中间状态和输出都由一个名为 RackState 的类表示（任意状态 S）。代码如下。

4）然后定义两个函数。第一个被称为 updateRackState，它包含了模式检测的核心逻辑，用来检测在 60 秒内产生三个连续超过 100 度的高温读数，在每个 group 上执行。第二个函数叫做 updateAcrossAllRackStatus，它是一个回调函数，它将被传递到 mapGroupsWithState API。这个函数确保根据事件时间的顺序来处理机架温度读数。实现代码如下。

设置步骤现在已经完成。

5）现在在结构化流应用程序中将回调函数连接到 mapGroupsWithState。实现代码如下所示：

6）执行以上程序。当读取到第一个 file1.json 数据文件时，输出结果如下：

当读取到第二个 file2.json 数据文件时，输出结果如下：

当读取到第三个 file3.json 数据文件时，输出结果如下：

从以上的输出结果中，分以看到，rack1 有一些温度读数超过 100 度；然而，它们不是连续的，因此输出状态处于正常水平。在文件 file2.json 中 rack2 有三个连续的温度读数超过 100 度，而每一个和前一个之间的时间间隔小于 60 秒，所以 rack2 的状态处于 warning（警告）级别。rack3 有三个连续的温度度数超过 100 度；然而，每一个和前一个之间的时间间隔超过了 60 秒。因此，它的地位处于正常水平。

【示例】基于用户会话活动的复杂事件模式识别和处理。

在这个例子中，会话处理逻辑是基于用户活动的。当用户采取 login 动作时，会创建一个会话，并且在用户采取 logout 动作时结束会话。当没有用户活动持续 30 分钟时，会话将自动结束。

分析：本例需要利用前面描述的超时特性来执行此检测。就输出而言，每当会话开始或结束时，该信息将被发送到输出。输出信息由用户 ID、会话开始和结束时间以及访问页面的数量组成。这个用例使用 flatMapGroupsWithState API 执行用户会话化，它支持每组输出多于一行的能力。

请按以下步骤操作。

1）准备数据

在本示例中，我们使用文件数据源，事件数据存储在 JSON 格式的文件中。每个事件由四个字段组成：

- user：表示用户的唯一 ID，字符串类型。
- action：表示用户的行为，比如“login”、“click”、“send”、“view”或“logout”，字符串类型。
- page：表示用户浏览的页面，字符串类型。
- ts：表示事件发生时的时间戳。这是事件时间(event time)。

这个用例的数据由三个 json 文件组成。它们的内容如下所示。

file1.json：

```
{"user":"user1","action":"login","page":"page1", "ts":"2017-09-06T08:08:53"}
{"user":"user1","action":"click","page":"page2", "ts":"2017-09-06T08:10:11"}
{"user":"user1","action":"send","page":"page3", "ts":"2017-09-06T08:11:10"}
```

file2.json：

```
{"user":"user2","action":"login", "page":"page1", "ts":"2017-09-06T08:44:12"}
{"user":"user2","action":"view", "page":"page7", "ts":"2017-09-06T08:45:33"}
{"user":"user2","action":"view", "page":"page8", "ts":"2017-09-06T08:55:58"}
{"user":"user2","action":"view", "page":"page6", "ts":"2017-09-06T09:10:58"}
{"user":"user2","action":"logout","page":"page9", "ts":"2017-09-06T09:16:19"}
```

file3.json：

```
{"user":"user3","action":"login", "page":"page4", "ts":"2017-09-06T09:17:11"}
```

请仔细观察上面的文件内容。文件 file1.json 包含用户 user1 的行为记录，它包含一个 login 动作，但是没有 logout 操作。文件 file2.json 包含用户 user2 的所有活动，包括 login 和 logout 操作。文件 file3.json 只包含用户 user3 的 login 动作。在三个文件中，用户活动的时间戳是以这样一种方式设置的，即 user1 的会话将在 file3.json 被处理时超时。

为了模拟数据流的行为，我们将把这三个 json 文件复制到项目的“src/main/resources/session”目录下。

2）首先导入项目所依赖的包。

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types._
import org.apache.spark.sql.streaming.{GroupState, GroupStateTimeout, OutputMode}
import scala.collection.mutable.ListBuffer
```

3）接下来，准备三个 case class。对于这个用例，用户活动输入数据由类 UserActivity 来表示。用户会话数据的中间状态由类 UserSessionState 表示，用户会话输出由类 UserSessionInfo 表示。下面是实现这三个类的代码。

```

```

4）接下来定义两个函数。

第一个函数叫做 updateUserActivity，它负责根据单用户活动更新用户会话状态。它根据用户所采取的操作，适当地更新会话开始或结束时间。此外，它还更新了最新的活动时间戳。

```

```

第二个函数叫做 updateAcrossAllUserActivities，是回调函数，它将被传递到 flatMapGroupsWithState 函数。这个函数有两个主要的职责。第一个是处理中间会话状态的超时，并且当出现这种情况时，它会更新用户会话结束时间。另一个责任是确定何时以及什么被发送到输出。所需的输出是当用户会话启动时的一行，当用户会话结束时的另一行。

```

```

设置步骤现在已经完成。

5）在结构化流应用程序中将回调函数连接到 flatMapGroupsWithState 函数。在这个例子中，将利用

超时特性，因此需要设置水印和事件超时类型。

6）执行以上程序。当读取到第一个 file1.json 数据文件时，输出结果如下：

当读取到第一个 file2.json 数据文件时，输出结果如下：

当读取到第一个 file3.json 数据文件时，输出结果如下：

在处理 file1.json 中的用户活动之后，可以看到输出中有一行。这是因为每当函数 updateAcrossAllUserActivities 在用户活动中看到 login 动作时，它就会将 UserSessionInfo 类的一个实例添加到输出 ListBuffer 中。在处理 file2.json 之后，输出中有两行。一种是 login 动作，另一种用于 logout 操作。现在 file3.json 只包含一个带有动作 login 的 user3 的用户活动，但是输出包含两行。user1 的一行是检测 user1 会话超时的结果，这意味着由于 user1 缺乏活动，水印已经传递了该特定会话的超时值。

通过这两个用例的演示，我们看到结构化流中的任意有状态处理特性提供了灵活而强大的方法，可以将用户定义的处理逻辑应用于每个组，并完全控制何时发送何内容到输出，从而实现对复杂事件模式的检测和处理。

8.4 处理重复数据

当数据源多次发送相同的数据时，实时流数据中的数据就会产生重复。在流处理中，由于流数据的无界性，去除重复数据是一种非常具有挑战性的任务。

不过，Spark 结构化流使得流应用程序能够轻松地执行数据去重，因此这些应用程序可以通过在到达时删除重复的数据来保证精确一次处理。结构化流所提供的数据去重特性可以与水印一起工作，也可以不使用水印。不过，需要注意的一点是，在执行数据去重时，如果没有指定水印的话，在流应用程序的整个生命周期中，结构化流需要维护的状态将无限增长，这可能会导致内存不足的问题。使用水印，比水印更老的数据会被自动删除，以避免重复的可能。

在结构化流中执行重复数据删除操作的 API 很简单，它只有一个输入参数，该输入参数是用来惟一标识每一行的列名的列表。这些列的值将被用于执行重复检测，并且结构化流将把它们存储为中间状态。下面我们通过一个示例来演示这个 API 的使用。

【示例】处理重复到达的移动电话操作事件数据。

请按以下步骤操作。

1）准备数据

在本示例中，我们使用文件数据源，代表移动电话操作事件数据存储在两个 JSON 格式的文件中。每个事件由三个字段组成：

- id：表示手机的唯一 ID，字符串类型。
- action：表示用户所采取的操作。该操作的可能值是"open"或"close"。
- ts：表示用户 action 发生时的时间戳。这是事件时间(event time)。

两个数据文件的内容如下：

file1.json 文件：

```
{"id":"phone1","action":"open","ts":"2018-03-02T10:15:33"}
```

```
{"id":"phone2","action":"open","ts":"2018-03-02T10:22:35"}
{"id":"phone3","action":"open","ts":"2018-03-02T10:23:50"}
```

观察上面的数据，每一行都是唯一的 id 和 ts 列。

file2.json 文件：

```
{"id":"phone1","action":"open","ts":"2018-03-02T10:15:33"}
{"id":"phone2","action":"open","ts":"2018-03-02T10:22:35"}
{"id":"phone4","action":"open","ts":"2018-03-02T10:29:35"}
{"id":"phone5","action":"open","ts":"2018-03-02T10:01:35"}
```

观察上面的数据，前两行是 file1.json 中前两行的重复，第三行是唯一的，第四行也是唯一的，但延迟到达(所以在我们后面的代码中应该不被处理)。

为了模拟数据流的行为，我们将把这两个 JSON 文件复制到项目的“src/main/resources/mobile4”目录下。

2）编写流处理代码，在代码中我们基于 id 列进行分组 count 聚合。id 和 ts 列共同定义为 key。

4）执行上面的程序代码。

当读取到 file1.json 源数据文件时，输出结果如下所示：

当读取到 file2.json 源数据文件时，输出结果如下所示：

如预期所料，当读取到 file2.json 源数据文件时，输出结果中只有一行显示在控制台中。原因是前两行是 file1.json 中前两行的重复，因此它们被过滤掉了（去重）。最后一行的时间戳是 10:10:00，这被认为是迟到数据，因为时间戳比 10 分钟的水印阈值更迟。因此，最后一行没有被处理，也被删除掉了。

8.5 容错

当开发重要的流应用程序并将其部署到生产环境中时，最重要的考虑之一就是故障恢复。根据墨菲定律，任何可能出错的地方都会出错。机器将会有故障，软件将会有缺陷。当有故障时，结构化流提供了一种方法来重新启动或恢复流应用程序，并从停止的地方继续。

要利用这种恢复机制，需要配置流应用程序使用检查点和预写日志。理想情况下，检查点的位置应该是一个可靠的、容错的文件系统的路径，比如 HDFS 或 S3。结构化流将定期保存所有的进度信息到检查点位置，例如正在处理的数据的偏移细节和中间状态值。向流查询添加检查点位置非常简单，只需要在流查询中添加一个选项，将 checkpointLocation 作为名称和路径作为值。请参见下面的示例。

```
// 向一个流查询添加 checkpointLocation 选项
val userSessionSQ = userSessionDS.writeStream
    .format("console")
    .option("truncate",false)
    .option("checkpointLocation","/ck/location")          // 设置检查点
    .outputMode("append")
    .start()
```

如果查看指定的检查点位置，应该看到以下子目录：commits、metadata、offsets、sources 和 stats。这些目录中的信息是专用于于特定流查询的；因此，每个流查询都必须使用不同的检查点位置。

就像大多数软件应用程序一样，流应用程序将随着时间的推移而不断发展，因为需要改进处理逻辑或性能或者修复 bug。重要的是要记住，这可能会如何影响在检查点位置保存的信息，并知道哪些更改被认为是安全的。概括地说，有两类变化。一个是对流应用程序代码的更改，另一个是对 Spark 运行时

的更改。

8.5.1 流应用程序代码更改

检查点位置的信息被设计为对流应用程序的变化有一定的弹性。有一些变化将被认为是不相容的变化。第一个是通过改变 key 列、添加更多的 key 列、或者删除一个现存的 key 列来改变聚合的方式。第二种方法是改变用于存储中间状态的类结构，例如，当一个字段被移除或者字段的类型从字符串转换为整数时。当在重新启动期间检测到不兼容的更改时，结构化流将通过一个异常进行通知。在这种情况下，必须使用新的检查点位置，或者删除先前检查点位置的内容。

8.5.2 运行时更改

检查点格式被设计为向前兼容，这样当 Spark 跨越补丁版本或小版本的更新时（例如从 Spark 2.2.0 升级到 2.2.1 或从 Spark 2.2.x 升级到 2.3.x)，流应用程序应该能够从一个旧的检查点重新启动。唯一的例外是当有严重的 bug 修复时。不过通常不需要担心，当 Spark 引入不兼容的变更时，它会在发行说明中清楚地进行说明。

如果由于不兼容的问题，无法启动一个使用现有检查点位置的流应用程序，那么就需要使用一个新的检查点位置，并且可能还需要为应用程序提供一些关于偏移量的信息来读取数据。

8.6 流查询度量指标和监控

与其他长时间运行的应用程序（如在线服务）类似，我们有必要了解流应用程序正在进行的进展、传入的数据速率、或者中间状态所消耗的内存数量等信息。结构化流提供了一些 API 来提取关于最近执行进度的信息，以及在流应用程序中监控所有流媒体查询的异步方式。

8.6.1 流查询指标

关于流式查询的最基本的有用信息是它的当前状态。通过调用 StreamingQuery.status 函数，可以以可读的格式检索和显示这些信息。返回的对象是类型 StreamingQueryStatus，它将状态信息转换成 JSON 格式。下面的示例代码展示了状态信息的样子。

```
// 以 JSON 格式查询状态信息
// 从上面的示例中使用一个流查询
query.status
```

输出：

```
res11: org.apache.spark.sql.streaming.StreamingQueryStatus =
{
  "message" : "Waiting for data to arrive",
  "isDataAvailable" : false,
  "isTriggerActive" : false
}
```

很明显，在当前状态函数被调用的时候，前面的状态提供了关于流查询的基本信息。为了从最近的进展中获得更多的细节，例如传入的数据速率、处理速率、水印、数据源的偏移量，以及一些关于中间状态的信息，可以调用 StreamingQuery.recentProgress 函数。这个函数返回 StreamingQueryProgress 类的

实例的一个数组，它将细节转换成 JSON 格式。默认情况下，每个流查询都被配置为保持 100 个进度更新，这个数字可以通过更新 Spark 配置 spark.sql.streaming.numRecentProgressUpdates 来改变。要查看最新的流查询进度，可以调用函数 StreamingQuery.lastProgress。下面是展示了流查询进度的示例。

```
// 流查询进度结点
。。。。。。
```

查看上面显示的流进度的详细信息，有一些重要的关键指标值得注意。输入率表示从输入源流入流应用程序的传入数据量。处理速率代表流应用程序处理传入数据的速度有多快。在理想状态下，处理速率应该高于输入速率，如果不是这样，那么需要考虑在 Spark 集群中增加节点的数量。如果流应用程序通过隐式 groupBy 转换或显式地通过任意状态处理 APIs 来保持状态，那么关注 stateOperators 部分中的指标是很重要的。

Spark UI 在 job、stages 和 task 级别上提供了丰富的度量标准。流应用程序中的每个触发器都被映射到 Spark UI 中的一个 job，在那里查询计划和任务持续时间可以很容易地检查。

8.6.2 监控流查询

结构化流提供了一种回调机制，可以在流应用程序中异步接收事件和流查询的进展。这是通过 StreamingQueryListener 接口完成的，它告诉我们什么时候启动了流查询，什么时候它已经取得了一些进展，什么时候它被终止了。这个接口的一个实现可以控制如何处理所提供的信息。一个明显的实现是将这些信息发送到 Kafka topic 或其他用于离线分析的发布-订阅系统，或者另一个流应用程序进行处理。下面的代码包含一个 StreamingQueryListener 接口的简单实现；它将信息打印到控制台。

```
。。。。。。
```

一旦有了 StreamingQueryListener 的实现，接下来就是用 StreamQueryManager 注册它，StreamQueryManager 可以处理多个监听器。请参阅下面的代码，了解如何注册和注销监听器。

```
// 注册和注销一个 StreamingQueryListener 实例，使用 StreamQueryManager
Val listener = new ConsoleStreamingQueryListener

// 注册
spark.streams.addListener(listener)

// 注销
spark.streams.removeListener(listener)
```

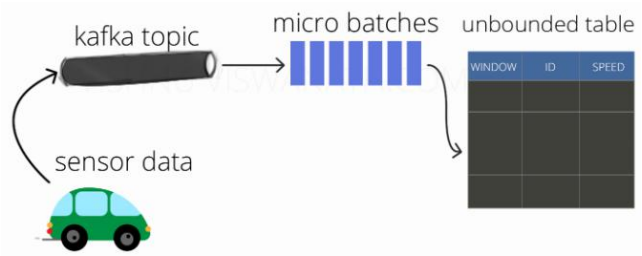
需要记住的一点是，每个监听器都从流应用程序中的所有流查询接收流查询事件。如果需要将特定的事件处理逻辑应用到某个流查询中，那么它可以利用该流查询名称。

8.7 结构化流案例：运输公司车辆超速实时监测

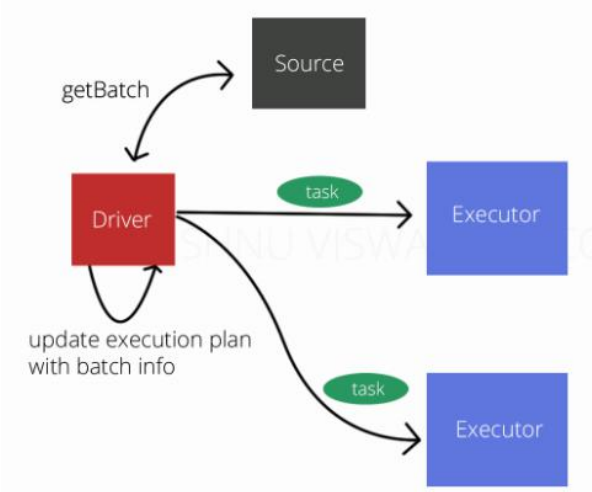
让我们想象一个车队管理解决方案，其中车队中的车辆启用了无线网络功能。每辆车定期报告其地理位置和许多操作参数，如燃油水平、速度、加速度、轴承、发动机温度等。利益相关者希望利用这一遥测数据流来实现一系列应用程序，以帮助他们管理业务的运营和财务方面。

使用到目前为止我们所知道的结构化流特性，我们已经可以实现许多用例，比如使用事件时间窗口来监控每天行驶的公里数，或者通过应用过滤器来找到燃油不足预警的车辆。

假设我们开了一家运输公司，需要检查车辆是否超速。我们将创建一个简单的接近实时的流应用程序来计算车辆每几秒钟的平均速度。



Spark 在处理实时数据流时，它会等待一个非常小的间隔，比如 1 秒(或者甚至 0 秒—即尽可能地快)。将在此间隔期间收到的所有事件合并到一个微批处理中。这个微批处理然后由驱动程序调度，作为任务在执行器中执行。完成微批处理执行后，将再次收集并调度下一批。这种调度是经常进行的，以给人以流执行的印象。



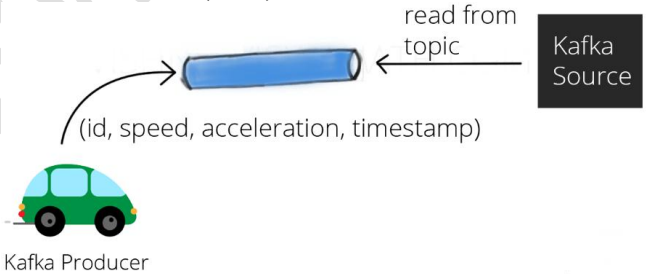
我们使用 Kafka 作为流数据源，将从 Kafka 的“cars”主题来读取这些事件。

。。。。。。

为了模拟车辆向我们发送传感器数据，我们将创建一个 Kafka producer，它将 id、speed、acceleration 和 timestamp 写入 Kafka 的“cars”主题。

.....

注意，这里的时间戳是在源处生成事件(消息)的时间。



接下来，我们将原始数据解析到一个 case class 类中，这样我们就有了一个可以使用的结构。

这会产生 CarEvent 类型的 DataSet。

执行聚合

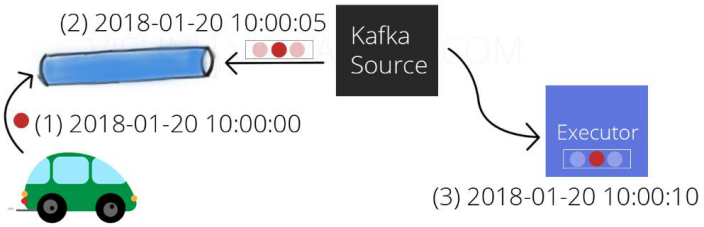
我们从求每辆车的平均速度开始。这可以通过对 carId 执行 groupby 并应用 avg 聚合函数来实现。

在结构化流媒体中，可以使用触发器控制微批处理的时间间隔。在 Spark 中，触发器被设置为指定

在检查新数据是否可用之前等待多长时间。如果没有设置触发器，一旦完成前一个微批处理执行，Spark 将立即检查新数据的可用性。

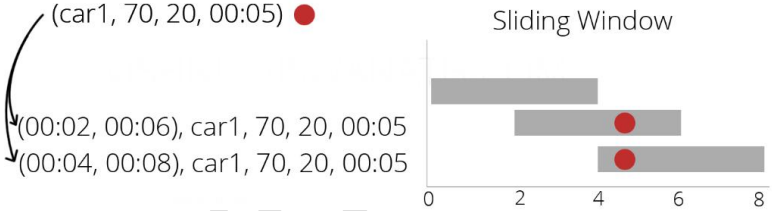
事件时间和处理时间

EventTime 是在源处生成事件的时间，而 ProcessingTime 是系统处理事件的时间。一些流处理系统还需要考虑另外一个时间，即 IngestionTime——事件/消息被摄入到系统中的时间。理解 EventTime 和 ProcessingTime 之间的区别很重要。



计算车辆在过去 5 秒内的平均速度。为此，我们需要根据事件时间将事件分组为 5 秒间隔时间组。这种分组称为窗口（Windowing）。

在 Spark 中，窗口是通过在 groupBy 子句中添加额外的 key 来实现的。对于每个消息，它的 EventTime(传感器生成的时间戳)用于标识消息属于哪个窗口。基于窗口的类型(滚动/滑动)，一个事件可能属于一个或多个窗口。如下图所示，为一个滚动窗口。



为了实现窗口化，Spark 添加了一个名为“window”的新列，并将提供的“timestamp”列分解为一个或多个行(基于它的值、窗口的大小和滑动)，并在该列上执行 groupBy。这将隐式地将属于一个时间间隔的所有事件拉到同一个“窗口”中。

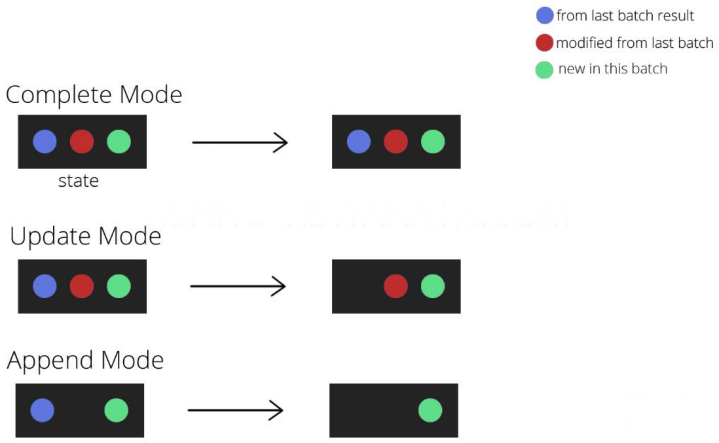
这里我们根据“window”和 carId 对 cars 数据进行分组。注意，window()是 Spark 中的一个函数，它返回一个列。

使用下面的方法定义大小为 4 秒、滑动为 2 秒的滑动窗口：

这将产生一个 carId、平均速度和相应时间窗口的 DataFrame。输出如下所示：

输出我们产生的结果到一个 sink—一个 Kafka 主题。

Spark 提供了三种输出模式—Complete、Update 和 Append。在处理微批处理后，Spark 更新状态和输出结果的方式各不相同。



在每个微批处理期间，Spark 更新前批处理中的一些 key 的值，有些是新的，有些保持不变。在 complete 模式下，输出所有的行，而在 update 模式下，只输出新的和更新的行。Append 模式略有不同，在 append 模式中，不会有任何更新的行，它只输出新行。

在使用 Kafka 接收器时，检查点位置是必须的，它支持故障恢复和精确地一次处理。

运行应用程序的输出如下所示:

请注意，结构化流 API 隐式地跨批维护聚合函数的状态。例如，在上面的例子中，第二个微批计算的平均速度将是第 1 和第 2 批接收到的事件的平均速度。作为用户，我们不需要自定义状态管理。但随着时间的推移，维护一个庞大的状态也会带来成本。这可以通过使用水印来实现控制。

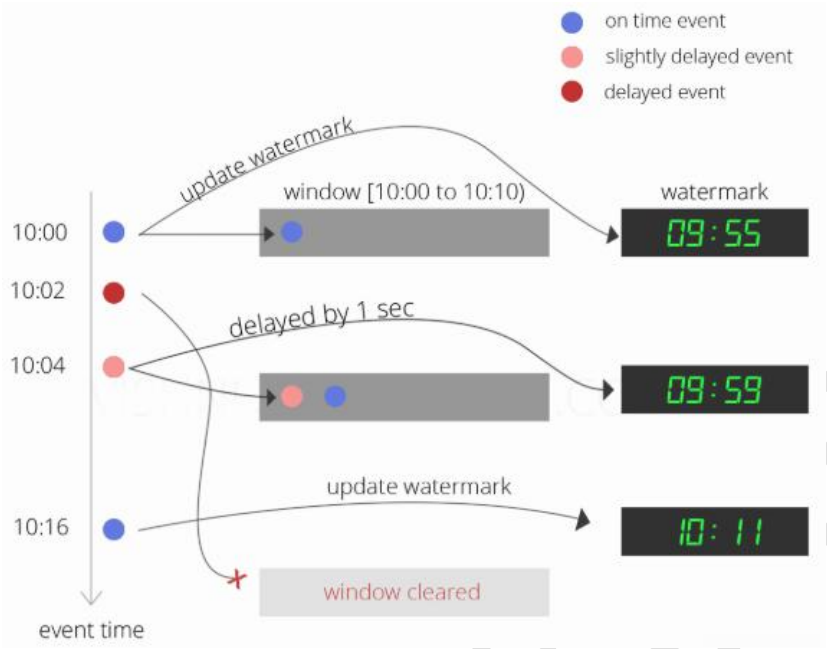
在 Spark 中，水印用于根据当前最大事件时间决定何时清除状态。基于我们所指定的延迟，水印滞后于目前所看到的最大事件时间。例如，如果 dealy 是 3 秒，当前最大事件时间是 10:00:45，那么水印是在 10:00:42。这意味着 Spark 将保持结束时间小于 10:00:42 的窗口的状态。

需要理解的一个细微但重要的细节是，当使用基于 EventTime 的处理时，只有当接收到具有更高时间戳值的消息/事件时，时间才会前进。可以将它看作 Spark 内部的时钟，但与每秒钟滴滴计时(基于处理时间)的普通时钟不同，该时钟只在接收到具有更高时间戳的事件时移动。

让我们看一个示例，看看在消息到达较晚时如何工作。我们将集中于[10:00 到 10:10]之间的单个窗口和 5 秒的最大延迟。例如：

```
.withWatermark("timestamp", "5 seconds")
```

引擎跟踪的最大事件时间，在每个触发器开始时将水印设置为(最大事件时间-5 秒钟)。



说明：

。。。。。。

完整代码：

```
.....
```

执行过程：

```

```

8.8 结构化流案例：实时订单分析

接下来，我们使用 Spark 结构化流实现"股票交易仪表板"流处理程序。

下面我们使用结构化流重写上一章的“股票交易仪表板”代码，以了解 Spark Structured Streaming 是如何处理流数据的。请按以下步骤操作：

```

```

8.9 结构化流案例：IP 欺诈检测

```

```

第 9 章 GraphX 图处理库

主要内容：

- 使用 GraphX API
- 构建图和操作图
- 使用 GraphX 内置图算法
- 案例：用 GraphX API 实现航班数据查询

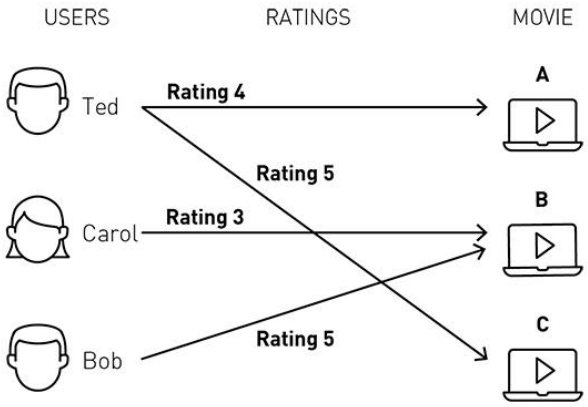
数据通常以记录或行集合的形式存储和处理。它表示为一个二维表，数据分为行和列。然而，集合或表不是表示数据的唯一方法。有时，图比集合提供更好的数据表示。

对于面向图的数据，图为处理数据提供了易于理解和直观的模型。此外，还可以使用专门的图算法来处理面向图的数据。这些算法为不同的分析任务提供了有效的工具。

使用图来表示连接数据的应用场景有以下几种。

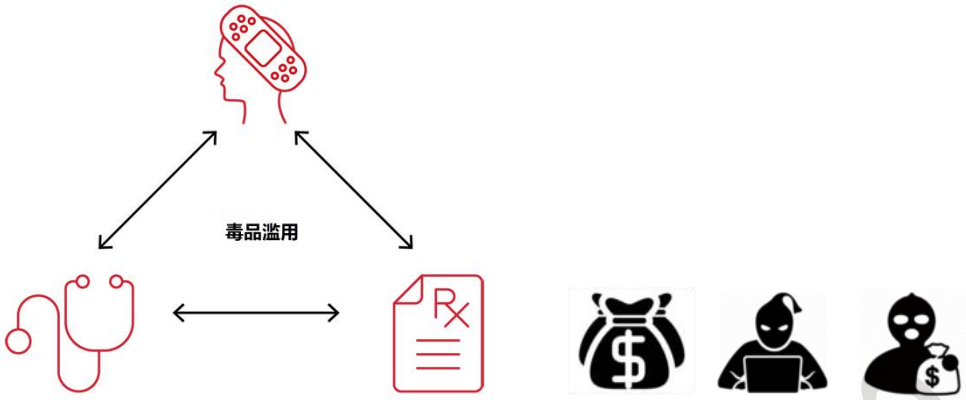
推荐引擎

推荐算法可以使用这样的图：节点是用户和产品以及它们各自的属性和边是用户对产品的评级或购买。图算法可以计算相似用户如何评价或购买相似产品的权重。



欺诈检测

图对于银行、医疗保健和网络安全中的欺诈检测算法非常有用。在医疗保健领域，图算法可以探索病人、医生和药房处方之间的联系。在银行业，图算法可以探索信用卡申请人与电话号码和地址之间的关系，或信用卡客户与商户交易之间的关系。在网络安全中，图算法可以发现数据漏洞。图分析可以用来监控金融交易，发现涉及金融欺诈和洗钱的人。



灾难检测系统

图形可以用来探测灾害，如飓风、地震、海啸、森林火灾、火山等，从而提供警告，提醒人们。

Page Rank

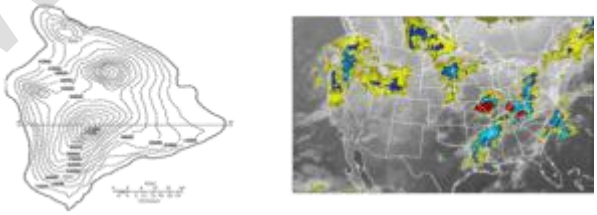
页面排名可以用于在任何网络中寻找有影响力的人，如论文引文网络或社交媒体网络。

经营分析

当与机器学习一起使用时，图有助于理解客户的购买趋势。例如优步、麦当劳等。

地理信息系统

图被广泛用于开发地理信息系统的功能，如流域划分和天气预报。



Google Pregel

Pregel 是谷歌的可扩展和容错平台，它的 API 足够灵活，可以表示任意的图算法。

图的分布式或者并行处理其实是把图拆分成很多的子图，然后分别对这些子图进行计算，计算的时

候可以分别迭代进行分阶段的计算，即对图进行并行计算。图和图结构的算法是特定行业的核心，可用在后勤物流、交通运输、路由选择、社交网络等领域。

Spark 平台提供了 GraphX 库，它是有效处理大规模面向图数据的图处理 API。GraphX 对 Spark RDD 进行了弹性分布属性图的扩展。属性图是一个有向多图，可以有多条边并行。每条边和顶点都有用户定义的属性。平行边允许相同顶点之间存在多种关系。

9.1 Spark 图处理

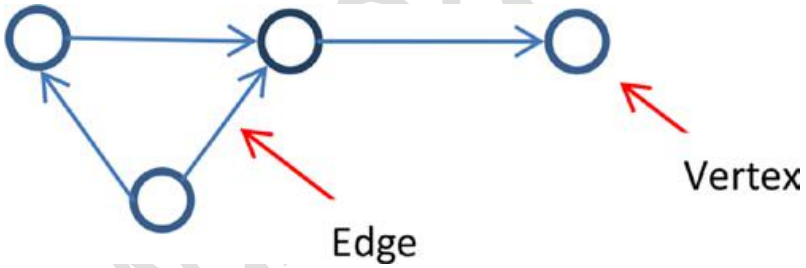
图（graph），作为链接对象的数学概念，由顶点（vertices，图中的对象）和连接顶点的边（edges）组成。

一旦表示为图，有些问题就变得更容易解决；它们自然地产生了图算法。例如，使用传统的数据组织方法(如关系数据库)呈现分层数据会是很复杂的，那么就可以用图来简化。除了使用它们来代表社交网络和网页之间的链接外，图算法还在生物学、计算机芯片设计、旅行、物理、化学等领域都有应用。

首先介绍一下本章中用到的基本图相关术语。

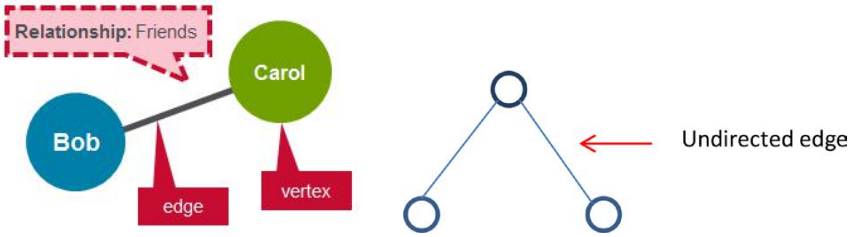
9.1.1 图基本概念

图是一种数学结构，用来建模对象之间的关系。图是由顶点和连接它们的边组成的。顶点是对象，而边是它们之间的关系。顶点是图中的一个节点。一条边连接图中的两个顶点。一般来说，顶点代表一个实体，边代表两个实体之间的关系。图在概念上等价于顶点和边的集合。

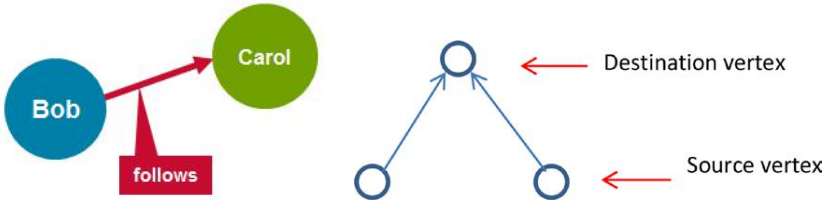


图可以有向也可以无向。

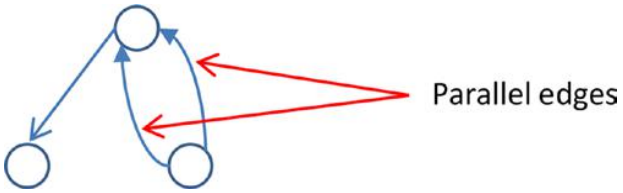
无向图是具有没有方向的边的图。无向图中的边没有源顶点或目标顶点。如下图所示，用户 Bob 和 Carol 构成图的顶点，它们之间的关系构成边：



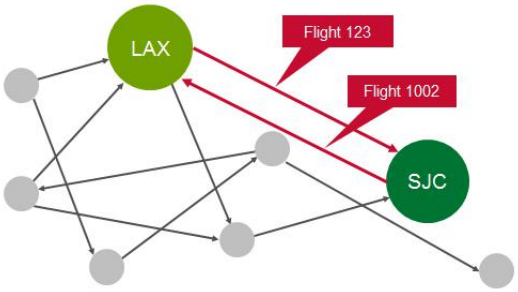
有向图就是边具有方向的图。有向图中的边具有源顶点和目标顶点。例如，Twitter follower 就是一个有向图。用户 Bob 可以 follow 用户 Carol，而不需要暗示用户 Carol 也 follow 用户 Bob。



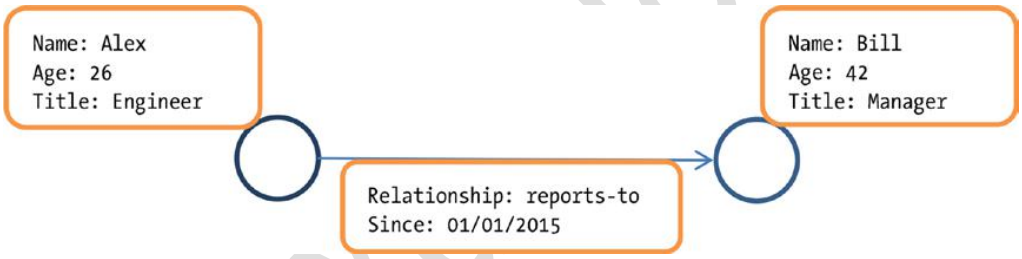
有向多图是一个有向图，它包含由两条或多条平行边连接的顶点对。有向多图中的平行边是具有相同起始和终止顶点的边。它们用于表示一对顶点之间的多个关系。



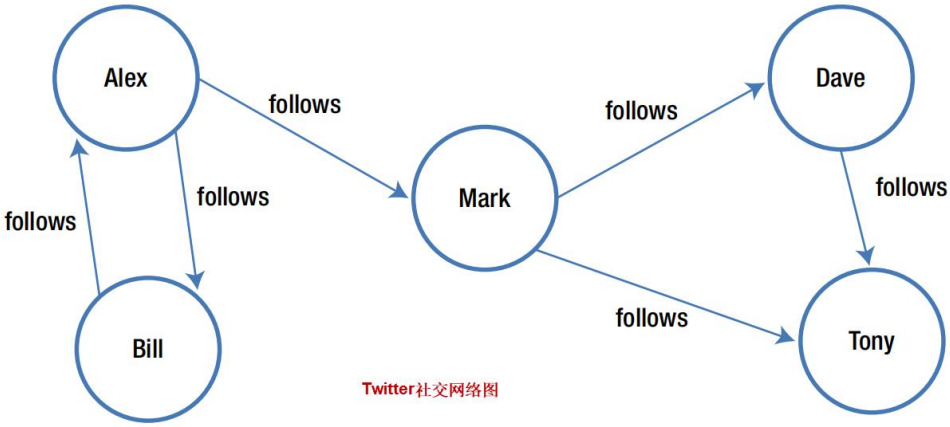
属性图是一个有向多图，它具有与顶点和边相关联的数据。属性图中的每个顶点都有一个或多个属性。类似地，每条边都有一个标签或属性。属性图如下所示：



属性图为处理面向图的数据提供了丰富的抽象。它是用图建模数据的最流行的形式。下面的图显示了一个属性图，表示公司中的两个员工及其关系。



属性图的另一个例子是表示 Twitter 上的社交网络的图。用户具有姓名、年龄、性别和位置等属性。此外，用户可以关注其他用户，并可能有追随者。在表示 Twitter 上的社交网络的图中，顶点表示用户，边表示“关注”关系。下图显示了一个简单的示例。

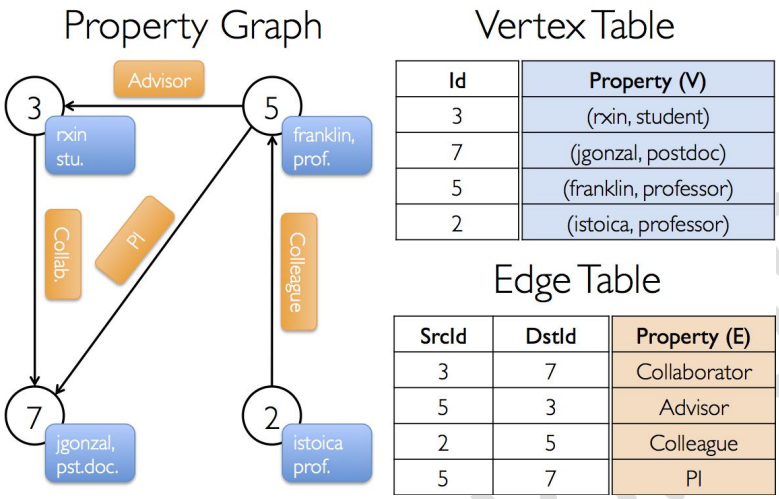


9.1.2 介绍 Spark GraphX 图处理库

Spark GraphX 是一个分布式图分析框架，它是用于图并行计算的 Spark 组件，扩展了 Spark 以用于大规模图处理。它建立在一个称为“图论”的数学分支上，为图分析提供了比 Spark Core API 更高层次

的抽象。Spark GraphX 基于 Spark 平台提供了对图计算和图挖掘简洁易用的接口，极大的方便了对分布式图处理的需求。

GraphX 的核心抽象是 Resilient Distributed Property Graph，一种顶点和边都带属性的有向多图，我们称为“Spark 属性图”。它扩展了 Spark RDD 的抽象，有 Table 和 Graph 两种视图，而只需要一份物理存储。两种视图都有自己独有的操作符，从而获得了灵活操作和执行效率。

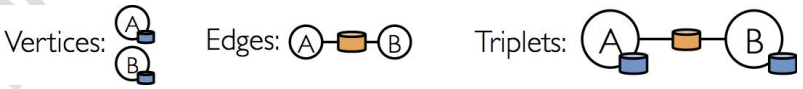


对 Graph 视图的所有操作，最终都会转换成其关联的 Table 视图的 RDD 操作来完成。一个图的计算在逻辑上等价于一系列 RDD 的转换过程。因此，Graph 最终具备了 RDD 的 3 个关键特性：不变性、分布性和容错性。其中最关键的是不变性。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX 会有一定程度的不变顶点和边的复用优化，对用户透明。

两种视图底层共用的物理数据，由 RDD[VertexPartition]和 RDD[EdgePartition]这两个 RDD 组成。点和边实际都不是以表 Collection[tuple]的形式存储的，而是由 VertexPartition/EdgePartition 在内部存储一个带索引结构的分片数据块，以加速不同视图下的遍历速度。不变的索引结构在 RDD 转换过程中是共用的，降低了计算和存储开销。

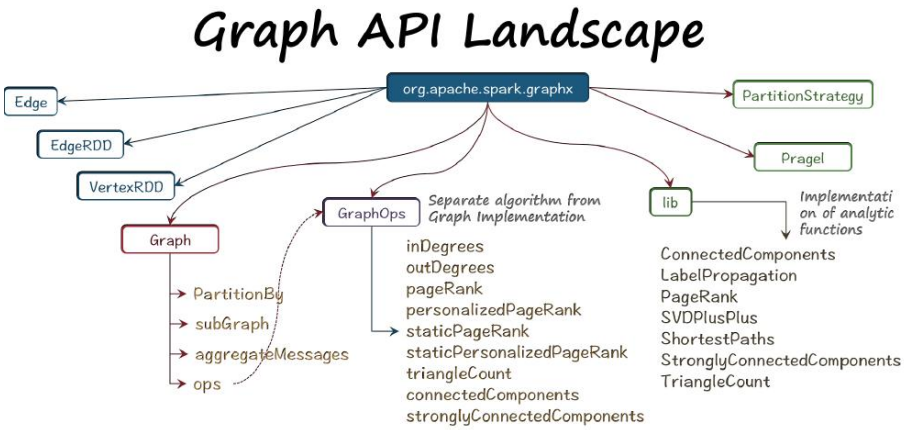
图的分布式存储采用点分割模式，而且使用 partitionBy 方法，由用户指定不同的划分策略。

在 Spark GraphX 中，边是有方向的，并且边和顶点都有附属于它们的属性对象。顶点表示对象，边表示这些对象之间的各种关系。例如，在包含关于网页和链接的数据的图中，附加到顶点的属性对象可能包含关于网页的 URL、标题、日期等的信息，并且附加到边的属性对象可能包含对链接的描述（一个 <a> HTML 标记的内容）。



GraphX API 提供了用于表示面向图数据的数据类型，以及用于图分析的基本的图运算符和图算法实现，它还提供了一个优化过的 Pregel API 实现。另外，GraphX 包含了一个快速增长的图算法和图构建器的集合，用以简化图分析任务。

在 org.apache.spark.graphx 包下有 Edge、EdgeRDD、VertexRDD、Graph 和 EdgeTriplet 等对象；Graph 对象具有如 triplets、persist、subgraph 等对象。而 GraphX 所实现的图算法位于 GraphOps 对象下(图算法和图实现分离)，在 lib 下是一些分析函数，如 SVD++、ShortestPath 等。



注：Spark GraphX API 对 Python 或 Java 不可用（只支持 Scala 的 API）。

Graph 类是 Spark GraphX 提供的主要类，它代表一个图对象，用于表示属性图的抽象，并提供对顶点和边的访问，以及用于转换图的各种操作。与 RDD 类似，它是不可变的、分布式的、容错的。在 Spark 中顶点和边是由两个特殊的 RDD 实现的：

- VertexRDD。包含元组，元组由两个元素组成：Long 型的顶点 ID 和任意类型的属性对象。
- EdgeRDD。包含 Edge 对象，它由源和目标顶点 ID（分别为 srcId 和 dstId）和任意类型的属性对象(attr 字段)组成。

VertexRDD

VertexRDD 表示属性图中顶点的分布式集合。它提供属性图中顶点的集合视图。VertexRDD 只为每个顶点存储一个条目。此外，它还为快速连接索引条目。

每个顶点都由一个 key-value 对表示，其中 key 是惟一的 id，value 是与该顶点关联的数据。key 的数据类型是 VertexId，它本质上是 64 位 Long 型。value 可以是任何类型的。

VertexRDD 是一个泛型类或参数化类，它需要一个类型参数。它被定义为 VertexRDD[VD]，其中类型参数 VD 指定与图中每个顶点关联的属性或属性的数据类型。例如，VD 可以是 Int、Long、Double、String 或用户定义的类型。因此，图中的顶点可以用 VertexRDD[String]、VertexRDD[Int]或用户定义类型的 VertexRDD 来表示。

Edge

Edge 类是属性图中的有向边的抽象表示。Edge 类的一个实例包含源顶点 id、目标顶点 id 和边属性。Edge 也是一个需要类型参数的泛型类。类型参数指定边属性的数据类型。例如，边可以具有 Int、Long、Double、String 或用户定义类型的属性。

EdgeRDD

EdgeRDD 表示属性图中的边的分布式集合。EdgeRDD 类是通过 edge 属性的数据类型参数化的。

EdgeTriplet

EdgeTriplet 类的一个实例表示一条边和它连接的两个顶点的组合。它存储一条边的属性和它连接的两个顶点。它还包含边的源顶点和目标顶点的唯一标识符。EdgeTriplet 如下图所示：

EdgeTriplets 的集合表示属性图的表格视图。

GraphX

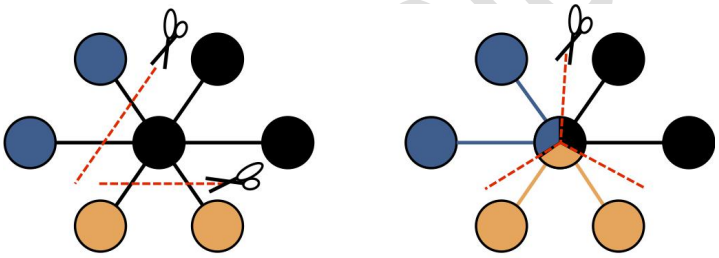
Graph 是 GraphX 用于表示属性图的抽象；Graph 类的一个实例表示一个属性图。与 RDD 类似，它是不可变的、分布式的和容错的。GraphX 使用顶点分区启发式在集群中划分和分布图形。如果一台机器出现故障，它将在另一台机器上重新创建分区。

实际上，Graph 类以 RDD 的形式提供对顶点和边的访问。顶点或边的 RDD 可以像任何其他 RDD 一样操作。所有的 RDD 方法都可用于转换或分析与顶点和边相关的数据。Graph 类还提供了通过图视图转换和分析顶点和边的方法。因此，图类的实例可以作为一对集合或一个图形来处理。

Graph 类不仅提供了修改顶点和边属性的方法，还提供了修改属性图结构的方法。由于 Graph 是不可变的数据类型，任何修改属性或结构的操作符都返回一个新的属性图。

GraphX 的图存储模式

图存储的两种方式：点分割存储和边分割存储。GraphX 使用的是顶点分割（Vertex-Cut）方式存储图。这种存储方式特点是任何一条边只会出现在一台机器上，每个点有可能分布到不同的机器上。



当点被分割到不同机器上时，是相同的镜像，但是有一个点作为主点,其他的点作为虚点，当点的数据发生变化时,先更新主点的数据，然后将所有更新好的数据发送到虚点所在的所有机器，更新虚点。

GraphX 不是沿着边分割图，而是沿着顶点分割图，这样可以同时减少通信和存储开销。这样做的好处是在边的存储上是没有冗余的，而且对于某个点与它的邻居的交互操作，只要满足交换律和结合律，就可以在不同的机器上面执行，网络开销较小。但是这种分割方式会存储多份点数据，更新点时，会发生网络传输，并且有可能出现同步问题。

GraphX 在进行图分割时，分配边的确切方法取决于划分策略。用户可以通过用 Graph.partitionBy 算子重新划分图来选择不同的策略。默认的分区策略是使用图构造时提供的边的初始分区。

有几种不同的分区(partition)策略，它通过 PartitionStrategy 专门定义这些策略。在 PartitionStrategy 中，总共定义了 EdgePartition2D、EdgePartition1D、RandomVertexCut 以及 CanonicalRandomVertexCut 这四种不同的分区策略。

- RandomVertexCut。这个方法比较简单，通过取源顶点和目标顶点 id 的哈希值来将边分配到不同的分区。这个方法会产生一个随机的边分割，两个顶点之间相同方向的边会分配到同一个分区。
- CanonicalRandomVertexCut。这种分割方法和前一种方法没有本质的不同。不同的是，哈希值

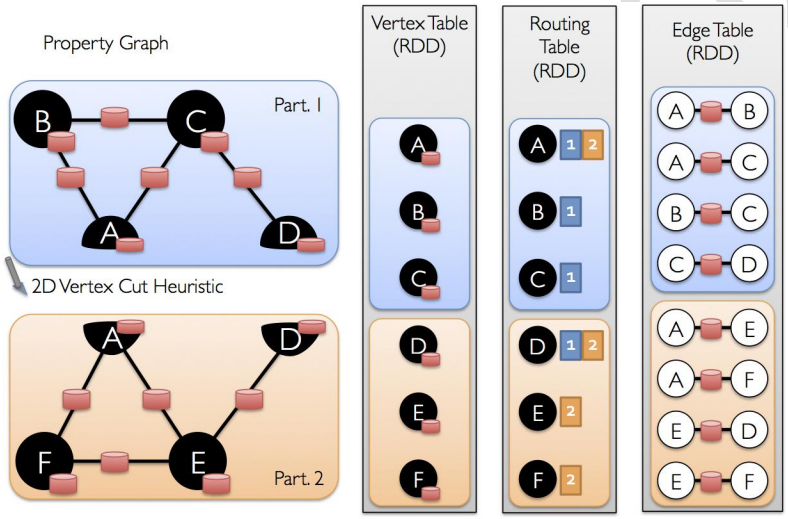
的产生带有确定的方向（即两个顶点中较小 id 的顶点在前）。两个顶点之间所有的边都会分配到同一个分区，而不管方向如何。

- EdgePartition1D。这种方法仅仅根据源顶点 id 来将边分配到不同的分区。有相同源顶点的边会分配到同一分区。
- EdgePartition2D。这种分割方法同时使用到了源顶点 id 和目的顶点 id。它使用稀疏边连接矩阵的 2 维区分来将边分配到不同的分区，从而保证顶点的备份数不大于 2 * sqrt(numParts)的限制。这里 numParts 表示分区数。

Graphx 使用的点分割图存储方式，用三个 RDD 存储图数据信息：

- VertexTable(id, data)：id 为顶点 id， data 为顶点属性。
- EdgeTable(pid, src, dst, data)：pid 为分区 id ，src 为源顶点 id ，dst 为目的顶点 id，data 为边属性。
- RoutingTable(id, pid)：id 为顶点 id ，pid 为分区 id。

点分割存储实现如下图所示：



9.1.3 使用 GraphX API 构建图

使用 GraphX 可以为完整的图分析工作流或管道提供一个集成平台。图分析管道通常包括以下步骤：

(1) 读取原始数据。
(2) 预处理数据(如清洗数据)。
(3) 提取顶点和边来创建属性图。
(4) 切分子图。
(5) 运行图算法。
(6) 分析结果。
(7) 对图的另一部分重复步骤 5 和 6。

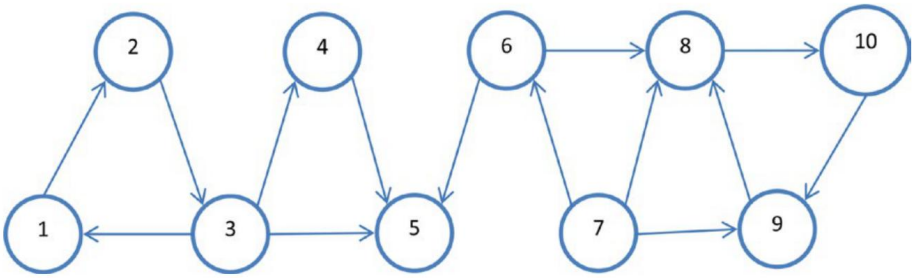
GraphX 的 Graph 对象是用户操作图的入口，它包含了边(edges)、顶点(vertices)以及 triplets 三部分，并且这三部分都包含相应的属性，可以携带额外的信息。

构建图的入口方法有两种，分别是根据边构建和根据边的两个顶点构建。

- 根据边构建图：Graph.fromEdges。
- 根据边的两个顶点数据构建：Graph.fromEdgeTuples。

不管是根据边构建图还是根据边的两个顶点数据构建，最终都是使用 GraphImpl 来构建的，即调用了 GraphImpl 的 apply 方法。

构建图的过程很简单，分为三步，它们分别是构建边 EdgeRDD、构建顶点 VertexRDD、生成 Graph 对象。在本节中，我们将学习如何使用 GraphX 构造和转换图。我们以一个社交网络为例，它有 10 个顶点和 14 条边，如下图所示：



我们创建上边这个属性图，表示类似于 Twitter 的用户网络的社交网络。在这个属性图中，顶点表示用户，有向边表示“follows”关系。顶点中的数字代表顶点 id。

构造图

Spark GraphX 库的 Graph 对象提供了从 RDD 构造图的工厂方法。一种常用的方法是使用一个包含元组（由一个顶点 ID 和一个顶点属性对象组成）的 RDD 和一个包含 Edge 对象的 RDD 来实例化一个 Graph 对象。我们将使用这个方法来构造上图中的示例图。

前面代码片段中的 Graph 对象从顶点 RDD 和边 RDD 创建 Graph 类的实例。

9.1.4 使用 GraphX API 查看图属性

正如我们所说，在 GraphX 中表示图的主类是 Graph。但是还有一个 GraphOps 类，它的方法被隐式添加到了 Graph 对象。其中包括获取顶点节点数量、入度、出度等的属性和方法。

GraphOps API：http://spark.apache.org/docs/latest/api/java/org/apache/spark/graphx/GraphOps.html

下面查看图的一些属性：

```
// 边的数量
val numEdges = socialGraph.numEdges
println("边的数量：" + numEdges)
```

输出如下：

```
边的数量：15
```

```
// 顶点的数量
val numVertices = socialGraph.numVertices
```

```
println("顶点的数量：" + numVertices)
```

输出如下：

```
顶点的数量：11
```

```
// 顶点的入度
val inDegrees = socialGraph.inDegrees
println("顶点的入度：")
inDegrees.collect.foreach(println)
```

输出如下：

```
// 顶点的出度
val outDegrees = socialGraph.outDegrees
println("顶点的出度：")
outDegrees.collect.foreach(println)
```

输出如下：

```
// 顶点的出入度总和
val degrees = socialGraph.degrees
println("顶点的出入度总和：")
degrees.collect.foreach(println)
```

输出如下：

```
// 获得属性图中顶点的集合视图
val vertices = socialGraph.vertices       // 获得所有的顶点
println("属性图中顶点的集合视图：")
vertices.collect.foreach(println)
```

输出如下：

```
// 获得属性图中边的集合视图
val edges = socialGraph.edges                // 获得所有的边
println("属性图中边的集合视图：")
edges.collect.foreach(println)
```

输出如下：

在 org.apache.spark.graphx 包中有一个 EdgeTriplet 类，它的一个实例表示一条边和该边连接的两个顶点的组合，存储一条边的属性和它连接的两个顶点。EdgeTriplet 类继承自 Edge 类，增加了分别包含源属性和目标属性的 srcAttr 和 dstAttr 成员。它还包含边的源顶点和目标顶点的惟一标识符。

输出如下：

9.1.5 使用 GraphX API 操作图

正如我们所说，在 GraphX 中表示图的主类是 Graph。但是还有一个 GraphOps 类，它的方法被隐式添加到了 Graph 对象。在确定要使用哪个方法时，我们需要考虑这两个类。在这一节中提到的一些方法来自 Graph 类，还有一些方法来自于 GraphOps 类。

属性转换操作

Graphx 类提供有一个高阶函数 mapVertices，用来对顶点属性进行转换，其参数函数返回一个新的顶点属性。mapVertices 用来更新顶点属性。从图的构建我们知道，顶点属性保存在边分区中，所以我们需要改变的是边分区中的属性。例如，我们需要将所有表示用户信息的顶点属性 user 的年龄增加 1 岁，执行代码如下：

输出结果如下：

Graphx 类还提供有一个高阶函数 mapEdges，用来对边属性进行转换，其参数函数返回一个新的边属性。mapEdges 方法用来更新边属性，它需要一个 map 函数，它接受分区 ID 和该分区中的边的迭代器，并返回转换后的迭代器（其中对于每个输入边都包含一个新 edge 属性对象（而不是新的 edge））。下面的代码使用 mapEdges 方法将每个边的属性从整数修改为字符串：

输出结果如下：

Graphx 类还提供有一个高阶函数 mapTriplets，用来更新边属性。它用在转换边属性时如果需要相邻的顶点值的情况下，使用 map 函数一次转换一个分区的每个边属性，并将相邻顶点属性传递给分区。下面的代码修改源图中的边属性，并返回一个新的边属性：

输出结果如下：

结构转换操作

Graphx 类还提供有对整个结构进行转换的方法，例如反转一个图结构，或者从一个给定的图中提取子图。在 Spark GraphX 库中的这些结构操作，允许用户将图作为一个单独的单元处理，生成一个新的图。其中 reverse 方法反转属性图中所有边的方向，它返回一个新的属性图。如下面的代码所示：

输出结果如下：

Graphx 类还提供了 groupEdges 高阶函数，用于将两个顶点之间的多条边合并为一条边。为了得到正确的结果，必须使用 partitionBy 对图进行分区。

输出结果如下：

groupEdges 操作合并多重图中的并行边(如顶点对之间重复的边)。在大量的应用程序中，并行的边可以合并（它们的权重合并）为一条边从而降低图的大小。

选择图子集

图上的另一个重要操作是选择子图（即只选择图的一部分）。有三种方法来实现这一点：

- subgraph：根据提供的谓词选择顶点和边。
- mask：只选择在另一张图中显示的顶点。
- filter：前两个的组合。

Graphx 类提供了 subgraph 方法，对每个顶点和边应用用户指定的过滤器。它返回源图的子图。概念上，它类似于 RDD filter 方法。下面是 subgraph 的签名：

```
def subgraph(
        epred: EdgeTriplet[VD, ED] => Boolean = (x => true),
        vpred: (VertexId, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
```

该方法接收两个 predicates（判断）作为参数，第一个 predicates（epred）接收一个 EdgeTriplet 作为参数。如果它返回 true，那么特定的边将被包含在结果图中。第二个 predicates（vpred）接收一个顶点 ID 及其属性对象作为参数。返回的子图，只包含满足这两个 predicates 的顶点和边。如果没有 vpred 函数参数，subgraph 函数会在新图中保留所有原始顶点。在新图中不再存在顶点的边会自动被删除。subgraph 方法的实现分两步：先过滤 VertexRDD，然后再过滤 EdgeRDD。

请看下面的示例：

输出结果如下：

subgraph 操作利用顶点和边的判断式（predicates），返回的图仅仅包含满足顶点判断式的顶点、满足边判断式的边以及满足顶点判断式的 triple。subgraph 操作可以用于很多场景，如获取感兴趣的顶点和边组成的图或者获取清除断开连接后的图。

Graphx 类还提供了 mask 方法，mask 函数是在 GraphX 中过滤图的另一种方法。有了 mask，你可以把一个图投射到另一个图上，只保留在第二个图中存在的那些顶点和边，而不用考虑两个图的属性对象。mask 唯一的参数是第二个图。它限制图只包含另一个图（用作掩码的图）中的顶点和边，但保留此图的属，非常类似于 SQL 语法中的"create ... like ..."语句。请看下面的代码：

输出结果如下：

mask 操作构造一个子图，这个子图包含输入图中包含的顶点和边。它的实现很简单，顶点和边均做 inner join 操作即可。这个操作可以和 subgraph 操作相结合，基于另外一个相关图的特征去约束一个图。

过滤图内容的第三个函数是 filter，它位于 org.apache.spark.graphx.GraphOps 类中。它与 subgraph 和 mask 函数都有关。该函数的签名如下：

它需要三个参数：一个预处理函数以及边和顶点判断函数。通过预处理函数把原来的图转换成另一

个图，然后使用提供的边和顶点判断函数进行修剪。得到的图将用作原始图的 mask（掩码）。换句话说，我们可以使用 filter 函数将两个步骤合并到一个步骤中。

此函数可用于基于某些属性过滤图，而无需更改程序中的顶点和边值。例如，我们可以删除图中出度为 0 的顶点：

输出结果如下：

连接操作

在许多情况下，有必要将外部数据加入到图中。例如，我们可能有额外的用户属性需要合并到已有的图中或者我们可能想从一个图中取出顶点特征加入到另外一个图中。这些任务可以用 join 连接操作完成。连接操作用来更新现有属性或向图中的顶点添加新属性。

在 org.apache.spark.graphx.GraphOps 类中，定义了一个 joinVertices 方法，它使用一个新的顶点集合（作为输入提供给源图）更新源图中的顶点，没有匹配的顶点保留其原始值。它有以下签名：

```
def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD) : Graph[VD, ED]
```

例如，假设顶点 3 和 4 的用户的年龄不正确，使用 joinVertices 方法来修改这两个用户的年龄。

输出结果如下：

在 org.apache.spark.graphx.Graph 类中还提供了一个 outerJoinVertices 方法，用来向源图中的顶点添加新属性。这个函数用于根据外部数据用新的值更新顶点。它有以下签名：

```
def outerJoinVertices[U:ClassTag, VD2:ClassTag](other: RDD[(VertexId, U)])
        (mapFunc: (VertexId, VD, Option[U]) => VD2) : Graph[VD2, ED]
```

这个函数需要提供两个参数：一个 RDD（包含带有顶点 IDs 和新顶点对象的元组），以及一个映射函数（将旧的顶点属性对象（类型 VD）和来自输入 RDD（类型 U）的新顶点对象组合在一起）。如果在输入 RDD 中对于某个特定的顶点 ID 没有对象存在，则该映射函数接收 None。

现在假设我们想向源图中的每个用户增加一个新的 city 属性，请看下面的示例代码：

输出结果如下：

joinVertices 和 outerJoinVertices 的区别

这两个方法都是根据给定的另一个图(原图的每个顶点 id 至多对应此图的的一个顶点 id)把原图中的顶点的属性值根据指定的 mapFunc 函数进行修改，返回一个新图，新图的顶点类型不变。但当图中的某个顶点 id 在另一个图中不存在时，它们的处理不同。

- joinVertices 的操作是会保留原图中该顶点属性的原值。
- outerJoinVertices 操作是使用 None 值作为该顶点的属性值。

聚合操作

GraphX 中提供的聚合操作有三个，分别是 aggregateMessages、collectNeighborIds 和 collectNeighbors。

在 org.apache.spark.graphx.Graph 类中提供了 aggregateMessages 方法，用来从相邻顶点和连接边聚合每个顶点的值。它返回顶点 id 和聚合消息的 pair RDD。这个方法使用两个用户定义的函数来进行聚

合，并为属性图中的每个三元组调用这些函数。

通过在边级别上表示计算，我们实现了最大的并行性。这是图 API 中支持邻域级计算的核心函数之一。它有以下方法签名：

```
def aggregateMessages[A: ClassTag](
        sendMsg: EdgeContext[VD, ED, A] => Unit,
        mergeMsg: (A, A) => A,
        tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[A]
```

它的主要功能是向邻边发消息，合并邻边收到的消息，返回 messageRDD。该接口有三个参数，分别为发消息函数、合并消息函数以及 tripletFields 属性。

其中 sendMsg 函数为图中的每条边接收一个 EdgeContext 对象，如果需要，则使用该 EdgeContext 向顶点发送消息。EdgeContext 对象包含源和目标顶点的 ID 和属性对象、边的属性对象、以及向邻近顶点发送消息的两种方法：sendToSrc 和 sendToDst。这个 sendMsg 函数可以使用 EdgeContext 来决定发送给每个顶点的消息。

而 mergeMsg 函数聚合去向同一顶点的消息。该函数用于在每个 edge 分区中每个顶点收到的消息合并，并且它还用于合并不同分区 vertexId 相同的消息。

最后，tripletFields 参数指定应该提供哪些字段作为 EdgeContext 的一部分。可能的值是 TripletFields 类中的静态字段（None，EdgeOnly，Src，Dst，和 All），默认是 TripletFields.ALL。

方法 aggregateMessages 的使用分两步：

- 在第一步（方法的第一个参数）中，消息被发送到目标顶点或源顶点（类似于 MapReduce 中的 Map 函数）
- 在第二步（方法的第二个参数）中，聚合是在这些消息上完成的（类似于 MapReduce 中的 Reduce 函数）。

在下面的示例中，我们计算社交图中每个用户的关注者中最年长的追随者的年龄（即粉丝的最大年龄）：

```

```

图结构处理示例

在下面这个图结构处理示例中，我们创建一个图，然后删除缺失的顶点，并合并边属性。

```

```

输出结果如下：

```

```

9.1.6 使用 GraphX Pregel API

图本身是递归数据结构，顶点的属性依赖于它们邻居的属性，这些邻居的属性又依赖于自己邻居的属性。所以许多重要的图算法都是迭代的重新计算每个顶点的属性，直到满足某个确定的条件。

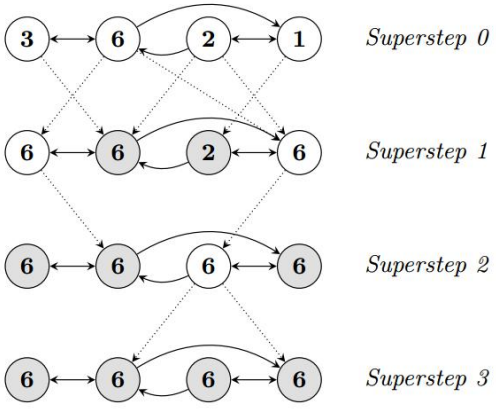
Pregel 是一种面向图算法的分布式编程框架，采用迭代的计算模型：在每一轮，每个顶点处理上一轮收到的消息，并发出消息给其它顶点，并更新自身状态和拓扑结构（出、入边）等。GraphX 实现了一个类似 pregel 的批量同步消息传递 API，这是基于 Bulk Synchronous Parallel (BSP，分布式批同步) 算法的。

分布式批同步（Bulk Synchronous Parallell,BSP）计算模式

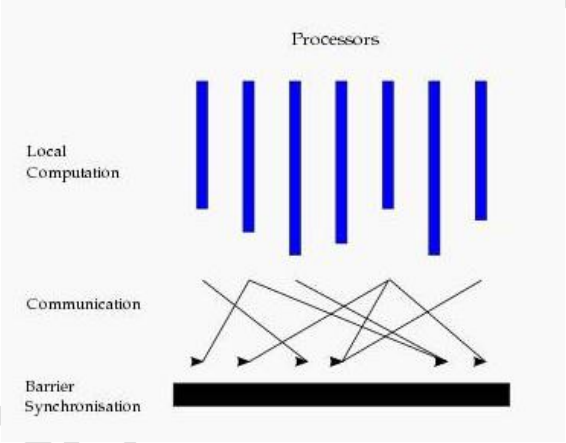
（1）BSP 基本原理

在 BSP 中，一次计算过程由一系列全局超步组成，每一个超步由并发计算、通信和同步三个步骤组成。同步完成，标志着这个超步的完成及下一个超步的开始。

BSP 模式的准则是批量同步(bulk synchrony)，其独特之处在于超步(super step)概念的引入。一个 BSP 程序同时具有水平和垂直两个方面的结构。从垂直上看,一个 BSP 程序由一系列串行的超步(super step)组成,如图所示:



从水平上看，在一个超步中，所有的进程并行执行局部计算。一个超步可分为三个阶段，如图所示:



- 本地计算阶段，每个处理器只对存储在本地内存中的数据进行本地计算。
- 全局通信阶段，对任何非本地数据进行操作。
- 栅栏同步阶段，等待所有通信行为的结束。

（2）BSP 模型特点

BSP 模型有如下几个特点:

- 将计算划分为一个一个的超步(super step)，有效避免死锁;
- 将处理器和路由器分开，强调了计算任务和通信任务的分开，而路由器仅仅完成点到点的消息传递，不提供组合、复制和广播等功能，这样做既掩盖具体的互连网络拓扑，又简化了通信协议;
- 采用障碍同步的方式、以硬件实现的全局同步是可控的粗粒度级，提供了执行紧耦合同步式并行算法的有效方式。

GraphX 中实现的这个更高级的 Pregel 操作是一个约束到图拓扑的批量同步（bulk-synchronous）并行消息抽象。Pregel 操作符在一系列的超步（super steps）中执行。在每一次超步中，顶点的计算都是并行的，并且执行用户定义的同一个函数。每个顶点可以修改其自身的状态信息或以它为起点的出边的信

息，从前序超步中接受消息，并传送给其后续超步，或者修改整个图的拓扑结构。边，在这种计算模式中并不是核心对象，没有相应的计算运行在其上。因此，在每一个超步中，总是按如下顺序执行：

- 顶点从上一个 super step 接收它们的入站(inbound)消息的总和；
- 为顶点属性计算一个新的值；
- 在下一个 super step 中向相邻的顶点发送消息。

消息通过边 triplet 的一个函数被并行计算，消息的计算既会访问源顶点特征也会访问目的顶点特征。在超步中，没有收到消息的顶点会被跳过。当没有消息遗留时，Pregel 操作将结束迭代并返回最终的图。

Pregel 计算模型中有三个重要的函数，分别是 vertexProgram、sendMessage 和 messageCombiner。

- vertexProgram：用户定义的顶点运行程序。它作用于每一个顶点，负责接收进来的信息，并计算新的顶点值。
- sendMsg：发送消息。
- mergeMsg：合并消息。

输出结果如下：

注意，与标准的 Pregel 实现不同的是，GraphX 的 Pregel 实现中的顶点仅仅能发送信息给邻居顶点，并且可以利用用户自定义的消息函数并行地构造消息。这些限制允许对 GraphX 进行额外的优化。

下面是另一个使用 Pregel 实现的图算法：最短路径算法。

输出结果如下所示：

上面的例子中，Vertex Program 函数定义如下：

```
(id, dist, newDist) => math.min(dist, newDist)
```

这个函数的定义显而易见，当两个消息来的时候，取它们当中路径的最小值。同理 Merge Message 函数也是同样的含义。

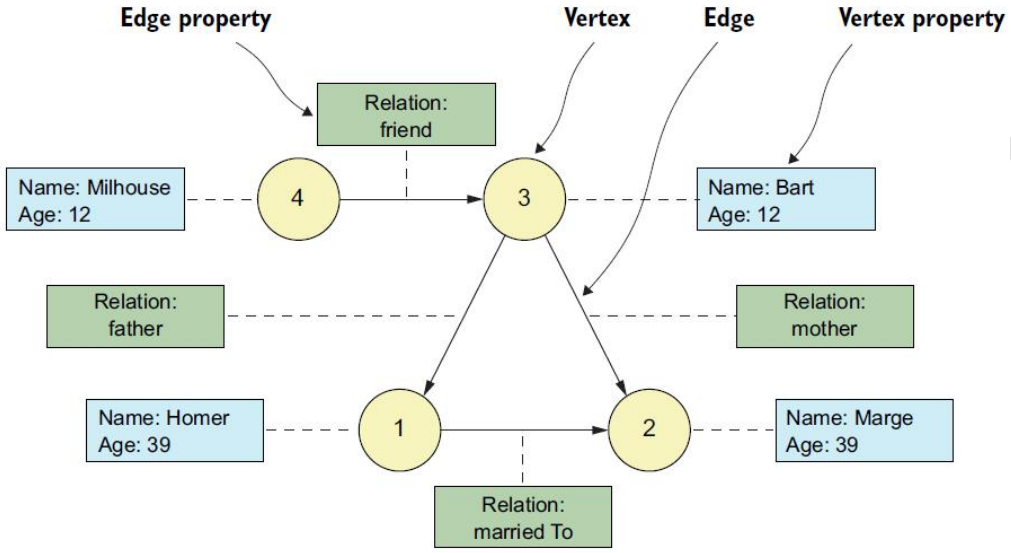
Send Message 函数中，会首先比较 triplet.srcAttr + triplet.attr 和 triplet.dstAttr，即比较加上边的属性后，这个值是否小于目的节点的属性，如果小于，则发送消息到目的顶点。

Pregel 框架的缺点

这个模型虽然简单，但是缺陷明显，那就是对于邻居数很多的顶点，它需要处理的消息非常庞大，而且在这个模式下，它们是无法被并发处理的。所以对于符合幂律分布的自然图，这种计算模型下很容易发生假死或者崩溃。

9.1.7 使用 GraphX 分析社交网络数据

在这一节，我们将构建一个社交网络图，它由 4 个顶点和 4 条边组成，如下图所示。



在这个图中，顶点代表人，边代表他们之间的关系。每个顶点（Vertex）有一个顶点 ID（圆中的数字）和附属他的顶点属性（关于这个人的姓名和年龄的信息）。边（Edge）属性包含关于关系类型的信息。让我们使用 Spark GraphX API 来构建这个图。

执行以上代码，输出结果如下所示：

接下来，我们想要对人和人之间的关系添加更多的信息，然后可以使用这些信息进行额外的计算。例如，可以添加关于两个人何时开始他们的关系以及他们见面的频率等信息。这里我们把边的属性修改为 Relationship 类（它只是 String 所表示的关系属性的包装器，在这里起到演示的作用）。

以上代码执行后的输出内容如下：

现在，假设我们想要在图中附上孩子、朋友和同事的数量，并添加一个标识他们是否结婚的标志。首先需要更改顶点属性对象，以便它们能够存储这些新属性。我们使用 PersonExt case class 来表示这个新的属性对象：

```
// 新的属性对象
case class PersonExt(name:String, age:Int, children:Int=0, friends:Int=0, married:Boolean=false)
```

接下来我们需要更改图的顶点来使用这个新的属性类。mapVertices 方法映射顶点的属性对象，需要给它的 map 函数传入一个顶点 ID 和一个顶点属性对象并返回新的属性对象：

```
// 修改顶点属性
val newGraphExt = newgraph.mapVertices((vid, person) => PersonExt(person.name, person.age))
```

```
// 查看修改后的顶点信息
newGraphExt.vertices.collect().foreach(println)
```

执行以上代码，输出内容如下：

现在，newGraphExt 中的所有顶点都有这些新属性，但它们都默认为 0，因此图中没有人结婚、有孩子、或者有朋友。如何计算正确的值？这就需要 aggregateMessages 方法，该方法用于在图的每个顶点上运行一个函数，并可选择性地向相邻的顶点发送消息。该方法收集并聚合发送到每个顶点的所有消息，并将它们存储在一个新的 VertexRDD 中。

接下来使用 aggregateMessages 来为 newGraphExt 图的每个顶点计算 PersonExt 对象的附加属性（朋友、孩子和 married 标志）。

输出结果如下：

在上面的代码中，sendMsg 函数发送给顶点的消息是元组，其中包含 children 的数量（Int）、朋友的数量（Int）、以及这个人是否已婚（Boolean）。只有当被检查的边是适当的类型时，这些值才被设置为 1（或 true）。第二个函数（mergeMsg）将每个顶点的所有值加起来，这样得到的元组就包含了总和。

因为 marriedTo 和 friendOf 关系只有一个边表示（为了节省空间），但是在现实中是双向的关系，在这些情况下，sendMsg 会向源和目标顶点发送相同的消息。mother 和 father 的关系是单向的，用于计算子女的数量，所以这些信息只发送一次。

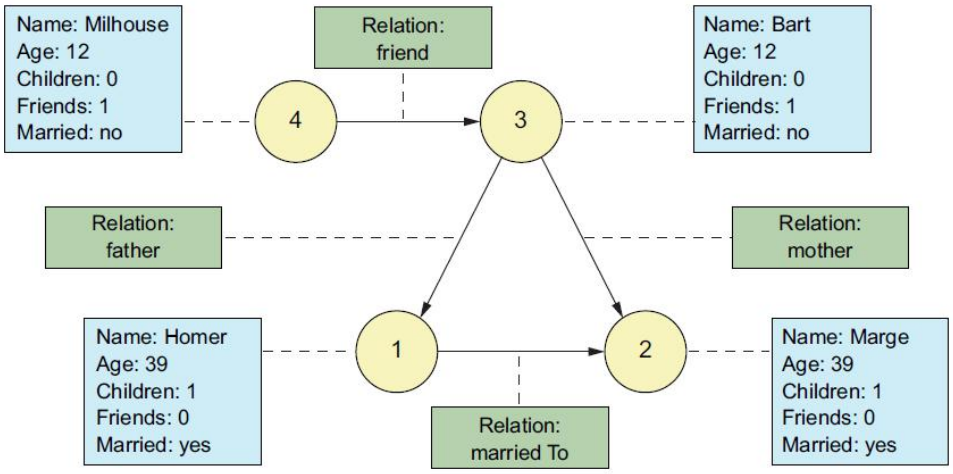
由此产生的 RDD 与原始的 VertexRDD 一样有七个元素，因为所有的顶点都接收到消息。结果是一个 VertexRDD，所以我们的工作没有完成（因为还没有得到结果 Graph）。要用这些新发现的值更新原始图，我们需要使用新的顶点来连接旧图。

下面的语句将连接 newGraphExt 图和来自 aggVertices 的新信息：

上面的代码执行后的输出内容如下：

在上面的代码中，如果对于一个已经存在的顶点有输入消息的话，映射函数将输入消息中的求和值复制到新的 PersonExt 对象，并保存 name 和 age 属性；否则它返回原始的 PersonExt 对象。

得到的图如下所示：



接下来，我们从上图（即 graphAggr）中选择这些有孩子的人。可以使用 subgraph 方法，并给它指定选择谓词，顶点和边必须满足才能被选中。

```
// val parents = graphAggr.subgraph(_ => true, (vertexId, person) => person.children > 0)
val parents = graphAggr.subgraph(vpred = (vertexId, person) => person.children > 0)
parents.vertices.collect.foreach(println)
```

执行上面的代码，输出内容如下：

```
(1,PersonExt(Homer,39,1,0,true))
(2,PersonExt(Marge,39,1,0,true))
```

查看所选的子图的边和顶点，会发现只有 Marge 和 Homer，有一条边连接着他们。

```
parents.edges.collect.foreach(println)
```

输出内容如下：

```
Edge(1,2,Relationship(marriedTo))
```

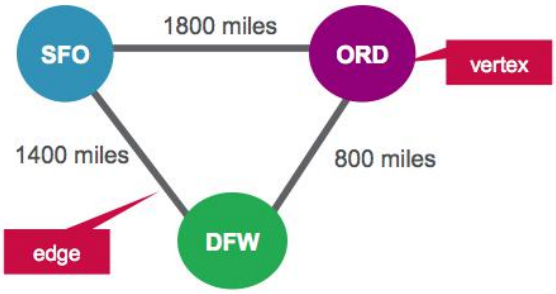
9.1.8 使用 GraphX 分析航班数据

使用 Spark GraphX 分析实时飞行数据，提供接近实时的计算结果，并使用 Google Data Studio 将结果可视化（ Google Data Studio is a product under Google Analytics 360 Suite. ）。

下面是一个简单的示例，我们将分析三个航班。每次航班，我们都有以下信息：

Originating Airport	Destination Airport	Distance
SFO	ORD	1800 miles
ORD	DFW>	800 miles
DFW	SFO>	1400 miles

在这个场景中，我们将机场表示为顶点，而航线表示为边。对于我们的图，我们将有三个顶点，每个顶点代表一个机场。机场之间的距离为航线属性，如下图所示：



其中代表机场的顶点表如下：

ID	Property
1	SFO
2	ORD
3	DFW

代表航线的边表如下：

SrcId	DestId	Property
1	2	1800
2	3	800
3	1	1400

首先，我们将导入 GraphX 包：

```
import org.apache.spark.graphx.{Edge, Graph}
import org.apache.spark.sql.SparkSession
```

然后构建图来表示机场及航线数据。

执行以上代码，输出内容如下：

接下来，我们需要回答以下问题：

1) 有多少个机场?
2) 有多少条航线?
3) 哪些航线的距离 ＞1000 英里?
4) 显示所有机场和航线的信息。
5) 排序并输出最长距离的航线。

代码如下所示：

执行以上代码，输出内容如下：

【作业】有三个顶点，分别代表 California 的三个城市中心：Santa Clara，Fremont，San Francisco。各城市间的距离如下：

源	目标	距离(miles)
Santa Clara, CA	Fremont, CA	20
Fremont, CA	San Francisco, CA	44
San Francisco, CA	Santa Clara, CA	53

请尝试用 GraphX 进行简单分析

参考实现代码如下：

9.2 GraphX 内置图算法

GraphX 为一些常见的图形算法提供了内置实现。这些算法可以作为对 Graph 类的方法调用。使用图算法可以很好地解决许多问题。

在本节中，我们将介绍几个 Spark GraphX 所实现了的图算法：

- 最短路径算法—找到一组顶点的最短路径。
- Page Rank 算法—根据进出的边数来计算图中顶点的相对重要性。
- 连通组件算法—找到不同的子图，如果它们存在于图中的话。
- 强连通的组件算法—找到双连通顶点的集群。

9.2.1 预处理数据集

在本节中，将使用的数据集可以从斯坦福大学获得。这个数据集包含了维基百科文章的一个子集，对于本节，我们只需要其中的两个：articles.tsv 和 links.tsv。其中 articles.tsv 包含唯一的文章名称（每行一个），links.tsv 包含与源和目标条目名称的链接，由制表符分隔。现要求用户将两篇文章用尽可能少的链接连接到一起。

首先对该数据集进行预处理，以构建图模型。在下面的代码中，先加载内容是文章名称的 articles.tsv 文件，删除空行和注释行，然后使用 zipWithIndex，为每个文章名称分配一个惟一的编号（ID）：

输出内容如下：

类似地，加载内容是文章链接的 links.tsv，代码如下：

输出内容如下：

然后，解析每个链接行以获得两个链接的文章名称，然后通过 join 连接 articles RDD 中的文章名称，使用文章 ID 来替换每个名称：

由此产生的 RDD 包含元组（元组包括源和目标的文章索引），输出内容如下：

接下来，我们就可以用它来构建一个 Graph 对象模型。这里，我们使用了 fromEdgeTuples 方法，它的作用是从边集合（被编码为顶点 ID 对）中构造一个图，该图带有边属性（其中包含重复边的计数或 1(如果 uniqueEdges 为 None)），以及顶点属性（包含每个顶点的总度数）。代码如下：

输出内容如下：

查看文章的数量和图中顶点的数量：

```
println("文章的数量：" + articles.count())
println("图中顶点的数量：" + wikigraph.vertices.count())
```

输出结果如下：

```
文章的数量：4604
图中顶点的数量：4592
```

可以看到在图中，文章和顶点的数量有细微的差别。这是因为链接文件中缺少一些文章。这可以通过在 linkIndexes RDD 中统计所有唯一文章名称来检查：

```

```

输出内容如下：

```

```

9.2.2 最短路径算法

从图中的某个顶点出发到达另外一个顶点的所经过的边的权重和最小的一条路径，称为最短路径。Dijkstra 算法是典型最短路径算法，用于计算一个节点到其他节点的最短路径。它的主要特点是以起始点为中心向外层层扩展(广度优先搜索思想)，直到扩展到终点为止。算法思想是，每次找到离源点最近的一个顶点，然后以该顶点为中心，然后得到源点到其他顶点的最短路径。

对于图中的每个顶点，最短路径算法会找到为了到达起始顶点所需的最小的边数。Spark 使用 ShortestPaths 对象实现了最短路径算法。它只有一个名为 run 的方法，该方法接受一个图和一个具有里程碑意义的顶点 ID 的序列。返回的图的顶点包含一个带有到每个地标的最短路径映射，其中地标性顶点 ID 是 key，最短路径长度是 value。

例如，如果我们想要将 Wikispeedia 中的“14th_century”与“Rainbow”这两个页面连接起来，那么需要的最小点击数是多少次？下面我们使用 Spark GraphX 来进行计算。

```

```

执行以上代码，输出内容如下：

```
Rainbow 的顶点 ID 是：3425
14th_century 的顶点 ID 是：10
```

这样就获得了两个页面顶点的 ID。然后调用 ShortestPaths 的 run 方法，用 wikigraph 和 Seq 中的一个 ID 作为参数，计算到给定地标顶点集的最短路径。

```

```

输出内容如下所示：

```
(10,Map(3425 -> 3))
```

可以看出，要将“14th_century”与“Rainbow”页面连接起来，最少需要三次点击。

但是，ShortestPaths 不会给出从一个顶点到另一个顶点的路径，只给出了需要到达它的边的数量。要在最短路径上找到顶点，需要编写自己的最短路径算法的实现，这超出了本书的范围。

9.2.3 页面排名

页面排名算法是由谷歌联合创始人 Larry Page 发明的。它通过计算输入边来确定顶点在图中的重要性。它被广泛用于分析 web 页面在 web 图中的相对重要性。它首先为每个顶点分配一个页面等级（PR）值为 1。它将这个 PR 值除以输出边的数量，然后将结果添加到所有相邻顶点的 PR 值。重复这个过程，

直到 PR 值的变化不再超过公差参数。

因为页面的 PR 值除以输出边的数量，所以页面的影响会根据它所引用的页面的数量按比例缩小。排名最高的页面是具有最少输出链接数量和最大输入链接数量的页面。PageRank 还研究了指向目标页面的页面的重要性。因此，如果一个给定的 web 页面有来自高级页面的传入链接，那么它的排名将会被排得更高。

我们可以通过调用图的 pageRank 方法并传入公差参数来在图上运行页面排名算法。公差参数决定了页面等级值可以改变的数量，并且仍然被认为是收敛的。值越小意味着越精确，但是算法也需要更多的时间来收敛。计算结果是另一个图，其顶点包含 PR 值。

下面的代码从 Wikispeedia 的文章中找出 10 个排名最高的页面。

输出内容如下所示：

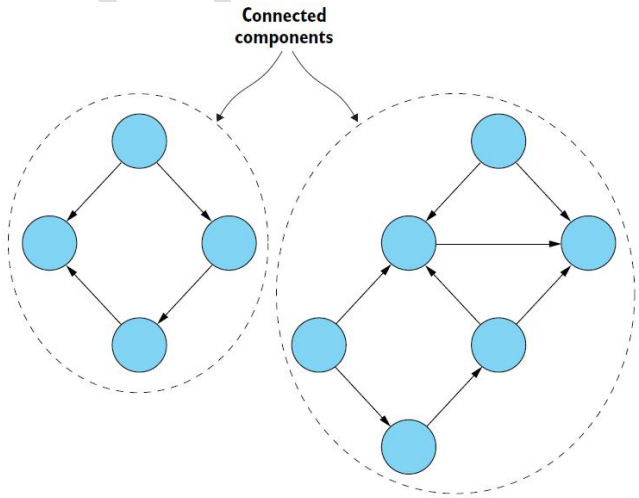
从输出结果中可以看出，top10 数组现在包含了 10 个最高排名的页面的顶点 ID 和它们的 PR 值，但是我们仍然不知道这些 ID 属于哪个页面。这可以使用 articles RDD 来 join 连接这个数组，以找出这些最高排名的文章名称：

输出内容如下所示：

结果中每个元组的第一个元素是顶点 ID；第二个元素包含 PR 值和页面名称。正如结果中所看到的，关于 United States 的页面是数据集中最具影响力的页面。

9.2.4 连通组件

连通组件是图的一个子图。在这个子图中，每个顶点都可以通过子图中的边来到达另一个顶点。如果一个图只包含一个连通的组件，并且它的所有顶点都可以从其他顶点到达，那么这个图就是连通的。下面这张图中显示了具有两个连通组件的图：来自一个连通组件的顶点不能从另一个连通组件到达。



在许多情况下，寻找连通的组件是很重要的。比如，在运行其他算法之前，应该先检查此图是否是连通的，因为这可能会影响到结果并影响到结论。

要在一个 GraphX 图上找到连通的组件，可以调用其 connectedComponents 方法（隐式地从 GraphOps 对象中获得）。连通的组件由该连通组件中的最低顶点 ID 表示。下面计算 Wikispeedias 图中每个顶点

的连通组件隶属关系。

输出内容如下所示：

wikiCC 图的顶点的属性对象包含它们所属的连通组件的最低顶点 ID。要找到这些 ID 的页面名称，可以再次将它们与 articles RDD 进行 join 连接：

输出内容如下所示：

如图中所见，Wikispeedia 图有两个独立的 web 页面集群。第一个是由与Áedán mac Gabráin有关的页面的顶点 ID 标识的，第二个是 Direct Debit 页面的顶点 ID。

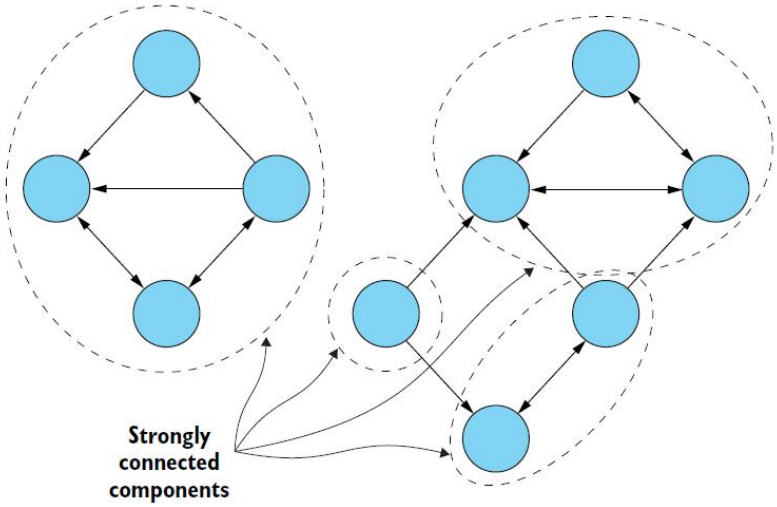
让我们看一看在每个集中群有多少页面：

输出内容如下：

可以看出，第二个集群只有三页，这意味着 Wikispeedia 具有很好的连通性。

9.2.5 强连通组件

强连通组件（简称 SCC）也是子图，其中所有顶点都连接到子图中的每个其他顶点（不一定是直接的）。SCC 中的所有顶点都需要彼此可访问（沿着边的方向）。下图是一个带有四个强连通组件的图。与连通组件不同，SCC 可以通过它们的一些顶点相互连接。



强连通组件在图论和其他领域有许多应用。例如，对于推荐引擎等应用程序，可以使用强连通的组件来分析组中的类似行为或关注的重点。

在 Spark 中，使用 GraphOps 对象的 stronglyConnectedComponents 方法来计算强连通组件，它也隐式地添加到 Graph 对象中。执行时只需要给它指定要执行的最大迭代次数即可。

输出内容如下：

与上一节连通组件算法一样，由 SCC 算法生成的图的顶点包含了它们所属的强连通组件的最低顶点 ID。该 wikiSCC 图包含 519 个强连通的组件：

输出内容如下：

最大的强连通组件是哪一个？

输出内容如下：

最大的 SCC 有 4051 个顶点，这太多了，无法在这里显示。我们可以在几个较小的 SCC 中检查属于这些顶点的页面的名称。对于 SCC 找到的接下来的三个最大值， 第一个 SCC 的成员是各大洲的国家名单列表：

输出内容如下：

第二个 SCC 包含关于一个恒星（HD_217107）和它的两个行星的页面：

输出内容如下：

第三个 SCC 的成员是英格兰的地区：

输出内容如下：

9.3 案例：分析家庭成员关系

现有两个文件，vertexs.csv 和 edges.csv，分别存储一个家庭的成员及他们的关系。下面使用 Spark GraphX 分析该家庭成员的关系。

实现代码如下：

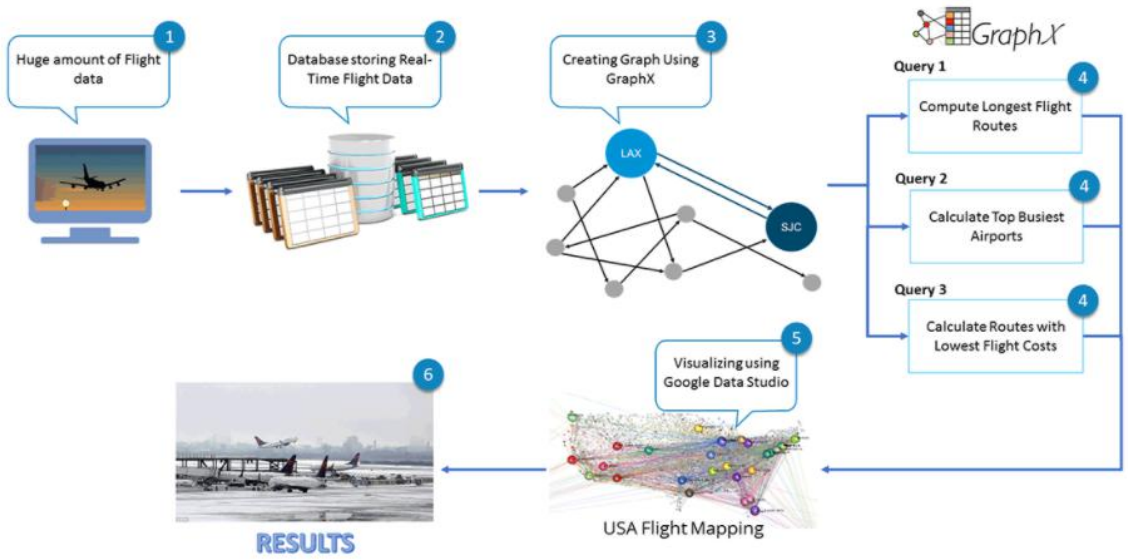
执行以上代码，输出内容如下：

9.4 案例：分析真实航班数据

我们使用 2014 年 1 月份的航班数据。对于每次航班，我们都有以下信息：

字段	描述	示例值
dOfM(String)	日期	1
dOfW(String)	星期	4
carrier(String)	航空公司代码	AA
tailNum(String)	飞机尾号的唯一标识符	N787AA
flnum(Int)	航班号码	21
org_id(String)	起飞机场ID	12478
origin(String)	起飞机场代码	JFK
dest_id(String)	目的地机场ID	12892
dest(String)	目的地机场代码	LAX
crsdeptime(Double)	规定起飞时间	900
deptime(Double)	实际起飞时间	855
depdelaymins(Double)	起飞延误(分钟)	0
crsarrtime(Double)	预定到达时间	1230
arrtime(Double)	实际到达时间	1237
arrdelaymins(Double)	到达延误(分钟)	7
crselapsedtime(Double)	飞行时间	390
dist(Int)	距离	2475

处理过程如下：



在这个场景中，我们将机场表示为顶点，而航线表示为边。我们对机场和航线的可视化很感兴趣，我们想知道有多少机场起飞或抵达。

然后，定义一个 case calss：

接下来，加载数据文件，分别构造机场信息的 RDD 和航线信息的 RDD。并分别以这两个 RDD 为顶点和边，来创建图模型。有两种方式可以实现：一种是直接加载为 RDD，另一种是先加载为 DataFrame，再转换为 RDD。

方式一：直接加载为 RDD，再将元素封装为 Flight 类，得到 RDD[Flight]。

执行以上代码，输出内容如下：

方式二：先加载为 DataFrame/Dataset，再转为 RDD[Flight]

将源数据加载到 RDD 中之后，接下来着手构建图模型。定义机场作为顶点，每个顶点是一个元组，由机场 ID 和机场名称组成 (flight.org_id, flight.origin)。例如：

ID	Property(V)
10397	ATL

边是机场之间的航线。边必须具有源、目标和属性。在我们的例子中，边包括:

Edge origin id	→	src (Long)
Edge destination id	→	dest (Long)
Edge property distance	→	distance (Long)

构建图模型的代码实现如下：

执行以上代码，输出内容如下：

我们已经根据真实航班数据，构建了一个图模型。接下来对该图模型进行分析，尝试回答以下问题：

- 有多少机场？
- 有多少航线？
- 哪些航线的距离大于 1000 英里？
- 根据距离对航线进行排序。
- 所有机场中，最高的出度、入度和出入度和分别是多少？
- 找出入港航班最多的 3 个机场。
- 找出出港航班最多的 3 个机场。
- 找出成本最低的航班。

实现代码如下：

执行以上代码，输出内容如下：

第 10 章 GraphFrames

图提供了一种强大的方法来分析数据集中的连接。对于面向图的数据，图为处理数据提供了易于理解和直观的模型。此外，还可以使用专门的图算法来处理面向图的数据。这些算法为不同的分析任务提供了有效的工具。

Spark GraphX 组件是位于 Spark Core 之上的分布式图处理框架，用于图并行和数据并行计算。它建立在一个称为"图论"的数学分支上，使在 Spark 中运行图算法成为可能。但是，GraphX API 存在一些限制。首先，GraphX 只支持 Scala 语言，所以无法使用 Python 进行大型的图计算。其次，GraphX 只能在 RDD 上工作，因此不能从 DataFrame 和 Catalyst 查询优化器提供的性能改进中受益。

GraphFrames 是一个开源的 Spark 包，创建它的目的是解决以上这两个问题。它具有以下特点：

- 提供一组 Python API。
- 适用于 DataFrame。

GraphFrames 扩展了 Spark GraphX 以提供 DataFrame API，使分析更容易使用、更有效，并简化了数据管道。GraphFrames 集成了 GraphX 和 DataFrame，使得用户可以在不将数据移动到专门的图数据库的情况下执行图模式查询。

GraphFrames 和 GraphX 的比较如下表所示。

	GraphFrame	GraphX
核心API	Scala, Java, Python	仅Scala
编程抽象	DataFrame	RDD
用例	算法，查询、Motif查找	算法
顶点ID	任何类型（Catalyst支持的）	Long
顶点属性/边属性	任意数量的DataFrame列	任何类型（VD,ED）
返回类型	GraphFrame/DataFrame	Graph[VD,ED]

GraphFrames 源码地址：https://github.com/graphframes/graphframes。

GraphFrames JAR 包下载地址：https://spark-packages.org/package/graphframes/graphframes。

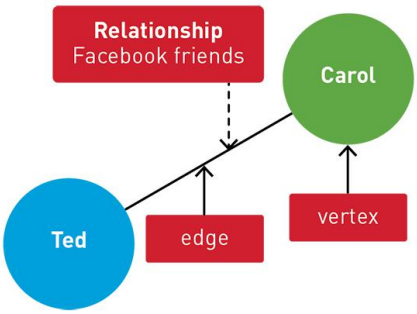
GraphFrames 文档地址：https://graphframes.github.io/graphframes/docs/_site/index.html。

10.1 图基本概念

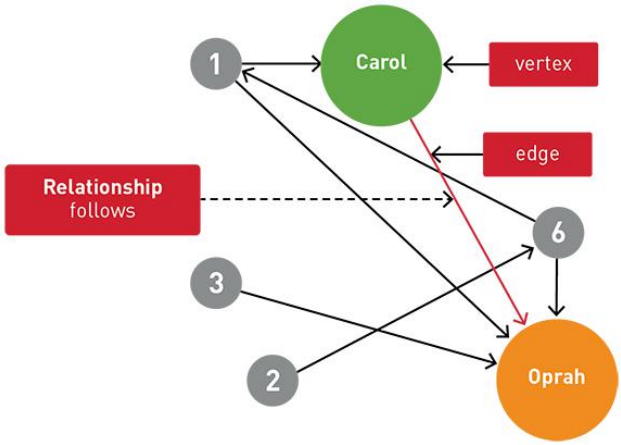
基于大数据集的图分析在社会网络、通信网络、网络图、交通网络、产品联购网络等各个领域都变得越来越重要。通常，以表格或关系格式从源数据创建图，然后在其上运行搜索和图算法等应用程序。

图是一种数学结构，用来建模对象之间的关系。图是由顶点和连接它们的边组成的。顶点是对象，边是它们之间的关系。

常规图是每个顶点的边数相同的图。Facebook 上的好友就是一个典型的例子。如果 Ted 是 Carol 的朋友，那么 Carol 也是 Ted 的朋友。

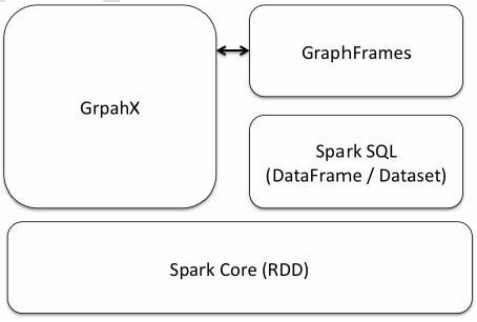


有向图就是边有方向的图。有向图的一个例子是 Twitter follower。Carol 可以 follow Oprah，而不需要暗示 Oprah follow Carol。



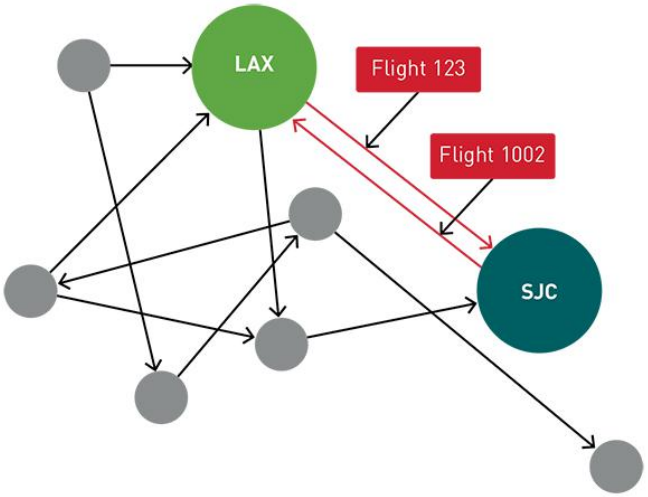
10.2 GraphFrame 图处理库简介

Spark GraphFrames 提供了一个声明性 API，可用于大型图模型上的交互式查询和独立程序。由于 GraphFrames 是在 Spark SQL 上实现的，它支持在计算过程中进行并行处理和优化。



GraphFrames API 中的主要编程抽象是一个 GraphFrame。从概念上讲，它由两个 DataFrame 组成，分别表示图的顶点和边。顶点和边可以有多个属性，这些属性也可以用于查询。例如，在社交网络中，顶点可以包含名称、年龄、位置和其他属性，而边可以表示节点（社交网络中的人）之间的关系。因为 GraphFrame 模型可以支持用户定义的每个顶点属性和边属性，因此它等价于属性图模型。此外，可以使用模式来定义视图，以匹配网络中各种形状的子图。

Spark GraphFrames 支持分布式属性图的图计算。属性图是一个有向多图，它可以并行地有多条边。每个边和顶点都有与之相关的用户定义属性。平行边允许相同顶点之间有多个关系。



使用 GraphFrames，顶点和边均表示为 DataFrame，这可以充分利用 Spark SQL 查询的优点，并支持 DataFrame 数据源，如 Parquet、JSON、CSV 等等。

GraphFrames 优化了计算的关系和图部分的执行。可以使用关系运算符、模式和对算法的调用指定这些计算。

10.3 GraphFrame 的基本使用

在本节中，我们将使用 Spark GraphFrames 来建立图模型并进行研究。图的顶点和边被存储为 DataFrame，并且支持基于 Spark SQL 和 DataFrame 的查询对它们进行操作。由于 DataFrame 可以支持各种数据源，我们可以从关系表、文件(JSON、Parquet、Avro 和 CSV)等等读取输入顶点和边信息。

顶点 DataFrame 必须包含一个叫做“id”的列，它为每个顶点指定唯一的 ID。类似地，边 DataFrame 必须包含名为“src”(源顶点的 ID)和“dst”(目标顶点的 ID)的两列。顶点 DataFrame 和边 DataFrame 都可以包含额外的属性列。

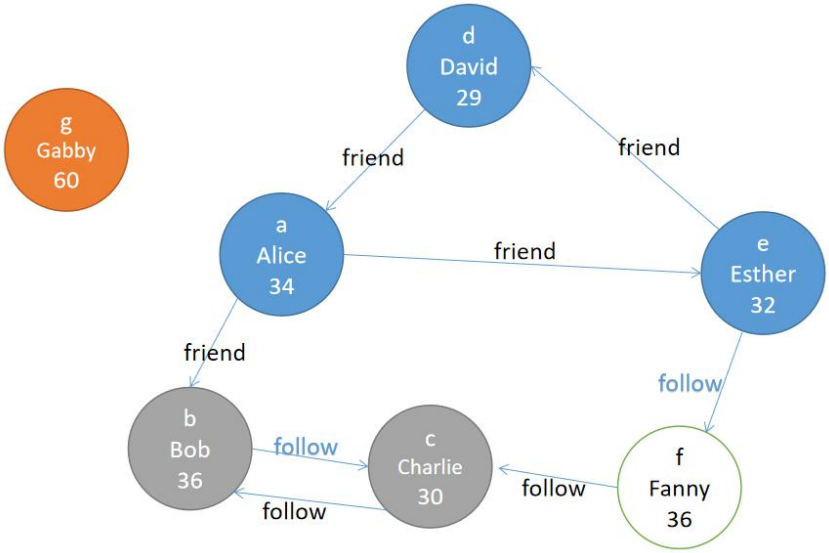
GraphFrames 公开了一个简洁的语言集成 API，它统一了图分析和关系查询，集成了图算法、模式匹配和查询。机器学习代码、外部数据源和 UDF 可以与 GraphFrames 集成来构建更复杂的应用程序。

10.3.1 添加 GraphFrame 依赖

.....。

10.3.2 构造图模型

假设我们现在要分析如下的一个社交网络：



10.3.3 简单图查询

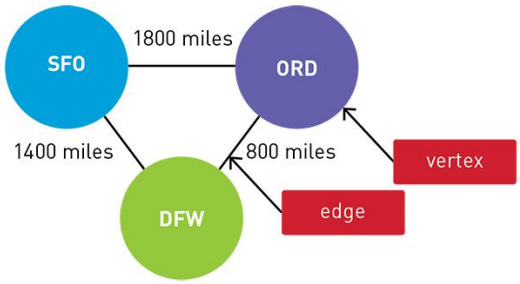
.....

10.3.4 示例：简单航线数据分析

作为一个简单的例子，我们将分析 3 个航线信息。每条航线，我们都有以下信息：

Originating Airport	Destination Airport	Distance
SFO	ORD	1800 miles
ORD	DFW	800 miles
DFW	SFO	1400 miles

用图表示如下：



在构建的图模型中，我们将机场表示为顶点，航线表示为边。图中有三个顶点，每个顶点代表一个机场。每个顶点都有机场代码作为 ID，机场所在城市名称作为属性。表示机场的顶点表如下：

id	city
SFO	San Francisco
ORD	Chicago
DFW	Texas

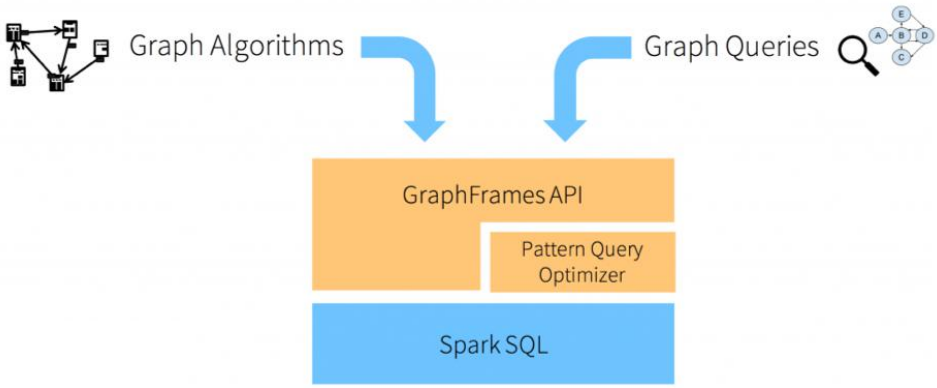
边具有源 ID、目标 ID 和作为属性的距离。表示航线的边的表如下：

Src	Dst	Distance	Delay
SFO	ORD	1800	40
ORD	DFW	800	0
DFW	SFO	1400	10

10.4 应用 motif 模式查询

图分析有两种形式：图算法和图模式查询。

GraphFrame 集成了图算法和图查询，支持跨图和 Spark SQL 查询的优化，而不需要将数据移动到专门的图数据库。



10.4.1 简单 motif 查询

Graph Motif 是在图中重复出现的子图或模式，表示顶点之间的交互或关系。图查询在图中搜索符合 motif 模式的结构，找到 motif 可以帮助我们执行查询来发现图中的结构模式。例如，我们可以使用 Motif 来分析用户购买产品的网络图，根据表示产品的图的结构属性及其属性和它们之间的关系洞察用户行为（找出经常同时购买的商品）。这些信息可用于推荐和/或广告引擎。

例如，我们可以搜索这样的模式：A follows B，B follows C，但是 A 并不 follows C。找到这样的结果后，我们就可以把 C 推荐给 A。

Motif 的语法形式为：

```
g.find("(start)-[pass]->(end)")。
```

其中 g 为图对象，start 为起点，pass 为经过的边，end 为目标点。下面是一个 GraphFrames Motif 查询模式，用来找到从 A 到 B 并且从 B 到 C，但没有从 A 到 C 的边的结构：

```
graph.find("(a)-[]->(b); (b)-[]->(c);!(a)-[]->(c)")
```

10.4.2 状态查询

无状态查询，没有指定任务限制条件，虽易于表达，但只能在查询完后再进行过滤，会返回一个较大的数据集。大多数 motif 查询是无状态的，如上一节的示例所示。

Motif 支持更复杂的查询，这些查询沿着 motif 中的路径携带状态。通过将 GraphFrame motif 查找与结果集上的过滤器相结合来表达这些查询，过滤器使用序列操作来构造一系列 DataFrame 列。

例如，我们要在图 g 中查找 4 个顶点（人）构成的关系链，要求其中至少有两个是“friend”-朋友关系。在本例中，要维护的状态是属性为“friend”的边的计数。

。。。。。。

10.5 构建子图

GraphFrames 提供了通过对边和顶点进行过滤来构建子图的 API。这些过滤器可以组合在一起来使用。例如，下面的子图只包含年龄在 30 岁以上的朋友。。

10.6 GraphFrames 内置图算法

和 GraphX 类似，GraphFrames 也实现了一些标准的图算法：

- 广度优先搜索(BFS)算法
- 连通分量算法
- 强连通分量算法
- 标签传播算法
- PageRank 算法
- 最短路径算法
- 三角计数算法

下面我们学习如何使用 GraphFrames 自带的这些图算法。

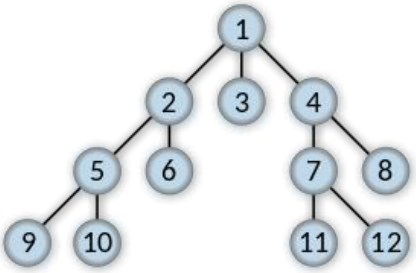
10.6.1 广度优先搜索(BFS)算法

广度优先搜索算法（Breadth-First-Search），是一种图搜索算法。简单的说，BFS 是从根节点开始，沿着树(图)的宽度遍历树(图)的节点。如果所有节点均被访问，则算法中止。BFS 属于盲目搜索。

算法步骤：

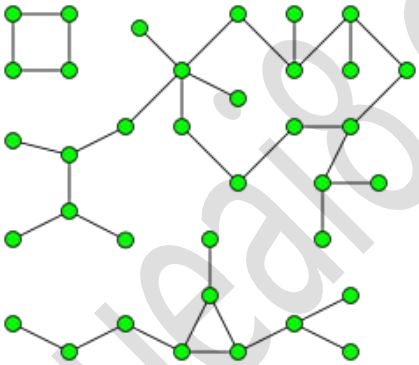
- 首先将根节点放入队列中。
- 从队列中取出第一个节点，并检验它是否为目标。如果找到目标，则结束搜寻并回传结果。否则将它所有尚未检验过的直接子节点加入队列中。
- 若队列为空，表示整张图都检查过了——亦即图中没有欲搜寻的目标。结束搜寻并回传“找不到目标”。
- 重复步骤 2。

如下图，其广度优先算法的遍历顺序为：1->2->3->4->5->6->7->8->9->10->11->12。



10.6.2 连通分量算法

在图论中，无向图的一个组件(有时称为连通组件)是一个子图，其中任意两个顶点通过路径相互连接，并且在超图中不与任何其他的顶点连接。例如，图中显示的图形有三个组件。没有关联边的顶点本身就是一个组件。一个自身连通的图只有一个分量，由整个图组成。

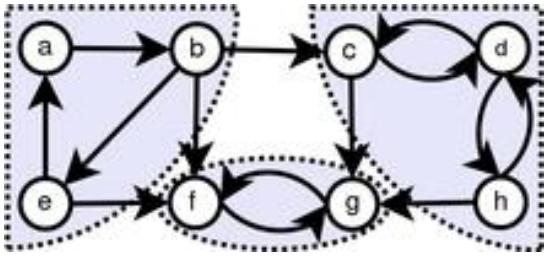


连通分量算法寻找孤立的集群或孤立的子图。这些集群是图中的连接顶点的集合，其中每个顶点都可以从同一集合中的任何其他顶点到达。该算法返回一个包含每个顶点和该顶点所在连通组件的GraphFrame。

10.6.3 强连通分量算法

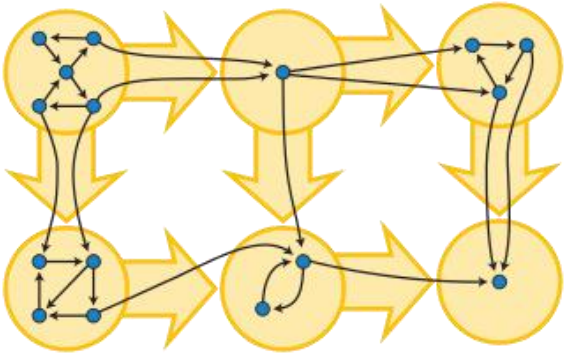
在有向图的数学理论中，如果一个图的每个顶点都能从其他顶点到达，那么这个图就被称为强连通图。任意有向图的强连通分量构成子图的划分，子图本身是强连通的。在线性时间内，可以测试图的强连通性，或者找到它的强连通部分。

如果有向图的每对顶点之间在每个方向上都有一条路径，则称为强连通图。也就是说，从第一个顶点到第二个顶点存在一条路径，从第二个顶点到第一个顶点存在另一条路径。



如果每个强连通分量都收缩到一个顶点，那么由此产生的图就是一个有向无环图（DAG）。例如，下面图中黄色有向无环图是蓝色有向图的浓缩。它是通过将蓝色图的每个强连通部分缩并成一个黄色的

顶点而形成的。



强连通分量算法计算每个顶点的强连通分量(SCC)，并返回一个图，其中每个顶点分配给包含该顶点的 SCC。

10.6.4 标签传播算法

标签传播是一种半监督机器学习算法，它将标签分配给之前未标记的数据点。在算法开始时，数据点的一个(通常很小的)子集有标签(或分类)。这些标签将在整个算法过程中传播到未标记的点。

在复杂的网络中，真实的网络往往具有社区结构。标签传播是一种查找社区的算法。与其他算法相比，标签传播在运行时间和网络结构所需的先验信息量(不需要预先知道参数)方面具有优势。缺点是它产生的不是唯一的解决方案，而是许多解决方案的集合。

在初始条件下，节点携带一个标签，该标签表示它们所属的社区。社区中的成员关系会随着邻近节点所拥有的标签而变化。此更改取决于节点一度内标签的最大数量。每个节点都用一个唯一的标签初始化，然后这些标签在网络中扩散。因此，紧密相连的组很快就会达到一个共同的标签。当许多这样密集(一致)的团体在整个网络中被创建时，他们继续向外扩展，直到不可能这样做。

该过程有 5 个步骤：

1) 初始化网络中所有节点上的标签。对于给定的节点 x, Cx (0) = x。
2) 设 t = 1。
3) 将网络中的节点按随机顺序排列，并设为 X。
4) 对于以该顺序选取的每个 x∈x, 令 Cx(t) = f(Cxi1(t), …, Cxim(t), Cxi(m+1) (t−1), …, Cxik (t−1)。这里返回相邻区域中出现频率最高的标签。如果有多个最高频率的标签，随机选择一个标签。
5) 如果每个节点都有其相邻节点的最大数量的标签，则停止该算法。否则，设 t = t + 1，转到(3)重复执行这个过程。

标签传播算法（Label Propagation Algorithm (LPA)）运行静态标签传播算法来检测网络中的社区。

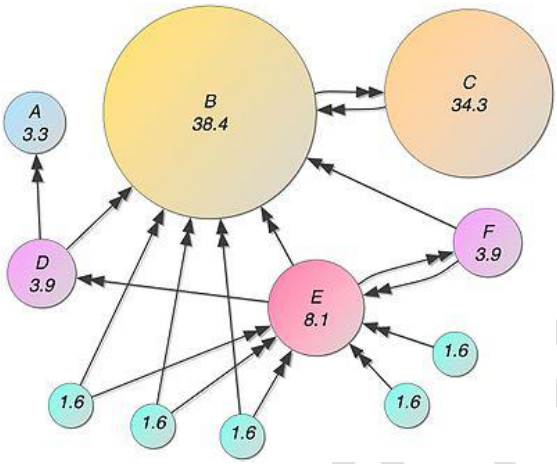
网络中的每个节点最初都被分配给它自己的社区。在每个 superstep 中，节点将它们的社区从属关系发送给所有邻居，并将它们的状态更新为传入消息的模式社区从属关系。

10.6.5 PageRank 算法

PageRank (PR)是谷歌搜索引擎使用的一种算法，用于在搜索引擎结果中对网页进行排名。它用节点代表网页，用边代表网页间的链接，并按链接页面的数量和排名加上每个链接的页面的数量和排名计算页面重要性。根据 Google 的描述：

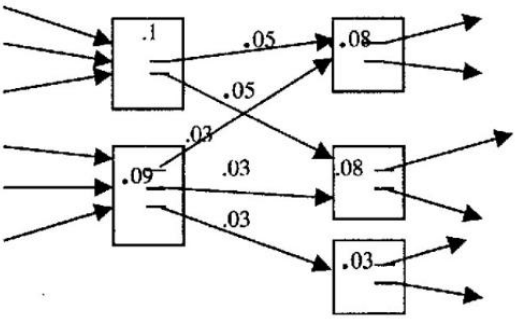
“PageRank 通过计算链接的数量和质量来粗略估计网站的重要性。潜在的假设是，更重要的网站有可能收到更多来自其他网站的链接。”

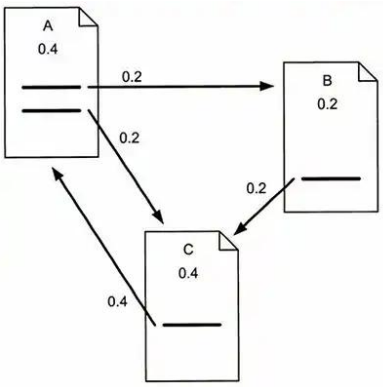
PageRank 是以谷歌的创始人之一 Larry Page 的名字命名的。PageRank 是衡量网站页面重要性的一种方法。PageRank 算法在测量图中某个顶点的重要性时非常有用。



PageRank 算法原理

PageRank 的计算充分利用了两个假设：数量假设和质量假设。用以上两个假设，PageRank 算法刚开始赋予每个网页相同的重要性得分，通过迭代递归计算来更新每个页面节点的 PageRank 得分，直到得分稳定为止。





步骤如下：

- 1）在初始阶段：网页通过链接关系构建起 Web 图，每个页面设置相同的 PageRank 值，通过若干轮的计算，会得到每个页面所获得的最终 PageRank 值。随着每一轮的计算进行，网页当前的 PageRank 值会不断得到更新。
- 2）在一轮中更新页面 PageRank 得分的计算方法：在一轮更新页面 PageRank 得分的计算中，

每个页面将其当前的 PageRank 值平均分配到本页面包含的出链上，这样每个链接即获得了相应的权值。而每个页面将所有指向本页面的入链所传入的权值求和，即可得到新的 PageRank 得分。当每个页面都获得了更新后的 PageRank 值，就完成了一轮 PageRank 计算。

PageRank 计算得出的结果是网页的重要性评价。

在 GraphFrames 中，PageRank 算法有两种实现：

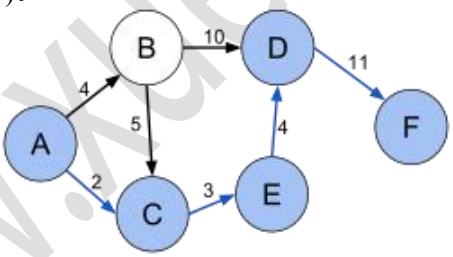
- 第一种实现使用 org.apache.spark.graphx.graph 接口与 aggregateMessages，并运行 PageRank 为固定的迭代次数。这可以通过设置 maxIter 来执行。
- 第二种实现使用 org.apache.spark.graphx.Pregel，并运行 PageRank，直到收敛，这可以通过设置 tol 参数来运行。

这两种实现都支持非个性化和个性化 PageRank，其中设置 sourceId 将对该顶点的结果进行个性化处理。(个性化 PageRank 算法的目标是要计算所有节点相对于用户 u 的相关度。个性化的 pagerank 跟传统 pagerank 不同的是，每次重新游走时，总是从用户 u 节点开始。另外，每个节点权重初始化时，个性化的 pagerank 是这样子的，假如对用户 u 推荐，则对用户 u 节点初始化为 1，其他节点都初始化为 0)

运行 PageRank，识别图中的重要顶点。

10.6.6 最短路径算法

在图论中，最短路径问题是在图中两个顶点(或节点)之间找到一条路径，使其组成边的权值之和最小。对于有向图，路径的定义要求连续的顶点由合适的有向边连接。例如，下面的加权有向图中顶点 A 和 F 之间的最短路径是(A, C, E, D, F)。



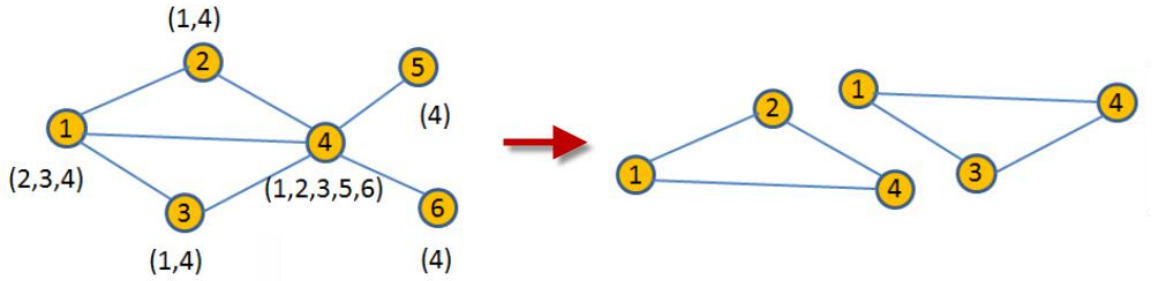
GraphFrames 中最短路径算法实现是 shortestPaths，它计算从每个顶点到给定的地标顶点集的最短路径（但是不考虑权值），地标由顶点 ID 指定。注意，这考虑了边的方向。

例如，在下面的代码中，我们计算了图中每个顶点到地标顶点“a”和“d”之间的最短路径。

```
val paths = g.shortestPaths.landmarks(Seq("a", "d")).run()
paths.show(false)
```

10.6.7 三角计数算法

顶点三角形计数算法是一种社区检测图算法，用于计算图中每个顶点所属的三角形的数量。三角形被定义为由三条边(a-b, b-c, c-a)连接的三个节点，其中每个顶点都与三角形中的其他两个顶点有关系。在一个图中，如果一条边的两个点如果有共同邻居点，那么这三个点就构成了三角形结构。



三角计数在社交网络分析中很受欢迎，它被用来检测社区并测量这些社区的凝聚力。它也可以用来确定一个图的稳定性，并经常被用作计算网络指标的一部分，如聚类系数。采用三角形计数算法计算局部聚类系数。

聚类系数有两种类型:

- 局部聚类系数。一个节点的局部聚类系数是它的邻居也被连接的可能性。这个分数的计算涉及到三角形计数。
- 全局聚类系数。全局聚类系数是局部聚类系数的归一化和（normalized sum）。

在 GraphFrames 中，通过 triangleCount 实现了该算法，它返回一个 GraphFrame，其中包含通过每个顶点的三角形数量。

例如，

.....

10.7 保存和加载 GraphFrame

由于 GraphFrame 是在 DataFrames API 上构建的，因此它们支持保存和加载各种数据源。在下面的代码中，我们展示了如何将顶点和边保存到 HDFS 上的 Parquet 文件中，然后从持久化存储中重新创建顶点和边 DataFrame，并创建图模型。

10.8 深入理解 GraphFrame

下面我们将简要介绍 GraphFrame 的内部结构及其执行计划和分区。

由于 GraphFrame 是基于 Spark SQL DataFrame 构建的，因此可以通过查看物理计划来理解图操作的执行，如下所示：

```
// 查看 GraphFrame 物理执行计划
g.edges.filter("salerank < 100").explain(true)
```

10.9 案例：亚马逊产品共同购买网络分析

【示例】亚马逊产品 2003 年 6 月 1 日联购网分析。

数据集：网络由爬取亚马逊网站收集。它是基于亚马逊网站的“购买了该商品的顾客同时也购买了”

功能。如果产品 i 经常与产品 j 共同购买，则图中包含从 i 到 j 的有向边。

数据集统计：

Dataset statistics	
Nodes	403394
Edges	3387388
Nodes in largest WCC	403364 (1.000)
Edges in largest WCC	3387224 (1.000)
Nodes in largest SCC	395234 (0.980)
Edges in largest SCC	3301092 (0.975)
Average clustering coefficient	0.4177
Number of triangles	3986507
Fraction of closed triangles	0.06206
Diameter (longest shortest path)	21
90-percentile effective diameter	7.6

请按以下步骤执行。

执行结果如下：

10.10 案例：亚马逊销售数据分析

在本节中，我们将分析一个建模为图的 JSON 数据集。我们使用一个包含 Amazon 产品元数据的数据集；该数据集包含关于约 548,552 种产品的产品信息和评论。

数据集：这些数据由爬取亚马逊网站收集（2006 年夏天），包含产品元数据和关于 548,552 种不同产品(书籍、音乐 cd、dvd 和 VHS 录像带)的评论信息。

https://snap.stanford.edu/data/amazon-meta.html

每个产品都有以下信息：

- 标题
- 销售排名
- 与当前产品共同购买的类似产品清单
- 详细的产品分类
- 产品评论：时间，客户，评级，投票数，发现评论有帮助的人数

Dataset statistics	
Products	548,552
Product-Project Edges	1,788,725
Reviews	7,781,990
Product category memberships	2,509,699
Products by product group	
Books	393561
DVDs	19828
Music CDs	103144
Videos	26132

为了处理简单，原始数据集被转换为 JSON 格式文件，每行代表一条完整的记录。使用本章提供的

Java 程序(Preprocess.java)进行转换。

10.11 案例：真实航班数据查询

这一节，我们将使用美国交通部的一些飞行信息，探索导致航班延误最多的航班属性：当航班延误超过 40 分钟时，哪一家航空公司、一周中哪几天、哪几个小时的航班延误次数最多?

在我们的用例中，边是机场之间的航班。一条边必须有 src 和 dst 列，并且可以有多个关系列。在我们的例子中，边包括：

id	src	dst	distance	depdelay	carrier	crsdephour
SFO_ORD_2017-01-01_AA	SFO	ORD	1800	40	AA	17

数据集说明

本案例所使用的数据集有两个，均为 json 格式。分别是存储航班信息的 flightdata2018.json 文件和存储机场信息的 airports.json 文本。现分别对这两个数据集加以说明。

flightdata2018.json：

- id: 航班 ID，由航空公司、日期、出发机场、目的机场、航班号组成；
- flDate：飞行日期（yyyy-MM-dd）；
- month：月份；
- dofW: 星期几 (1=Monday, 7=Sunday)；
- carrier: 航空公司代码；
- origin: 出发机场代码；
- dest: 目的机场代码；
- crsdephour: 预定起飞时间（hour）
- crsdeptime: 预定起飞时间（time）
- depdelay: 起飞延迟(分钟)
- crsarrtime: 预定到达时间
- arrdelay: 到达延迟(分钟)
- crselapsedtime: 飞行时间
- dist: 距离

airports.json：

- id：id
- city：机场所在城市
- state：机场所在州

现在我们可以查询该 GraphFrame 来回答以下问题:

- 有多少个机场?
- 有多少航班?

- 哪些航班距离最长?
- 最繁忙的航线有哪些？
- 哪个机场的进出航班最多?
- 哪些航班的平均延误时间最长?
- 一天当中，哪些时间段航班的平均延误时间较长?
- 超过 1500 英里的长途航班中延误时间最长的是?
- 从亚特兰大(Atlanta)起飞的各航班的平均延误时间是多少?
- 从亚特兰大起飞的航班一天当中哪些时段延误最严重?
- 根据 PageRank 算法，对机场的重要性进行排名。
- 每个机场的出港航班平均延误时间是多少？
- 查找两地没有直航的航线。
- 查找从每个机场到 LGA 的最短路径，我们使用 shortestPaths 算法
- LAX 机场和 LGA 机场之间是否有直飞航班？
- 查找 LAX 和 LGA 之间的转接航班(转机)。

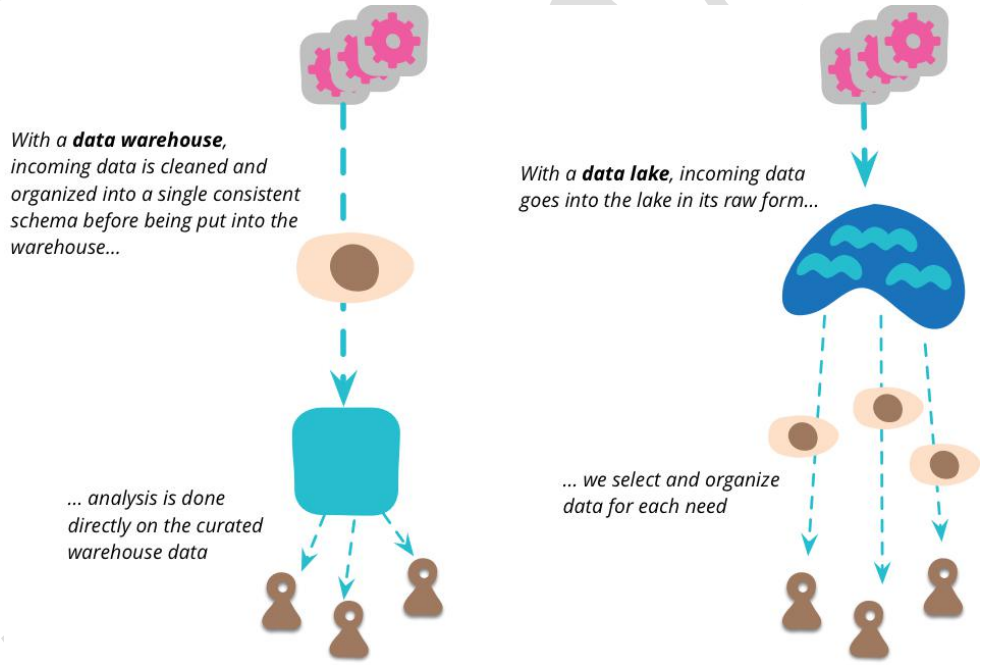
第 12 章 Delta Lake 数据湖

12.1 数据湖概述

随着数据量增长到新的、前所未有的水平，新的工具和技术正在出现，以处理这种增长。其中一个发展的领域是数据湖。

传统上，数据仓库工具用于从数据中驱动商业智能（BI）。业界随后认识到，数据仓库通过强制 schema on write 限制了智能的潜力。显然，在收集数据时不能考虑到所收集数据集的所有方面。这意味着，强制执行一个模式或删除一些看起来无用的项，从长远来看可能会损害商业智能。此外，数据仓库技术无法跟上数据增长的步伐。由于数据仓库通常基于数据库和结构化数据格式，因此它不足以应对当前数据驱动世界所面临的挑战。

因为在人工智能、ML 和大数据的时代，数据仓库对于实时决策并不是很有用，因为处理时间长，还有其他缺点。



这导致了数据湖的出现，这些数据湖针对非结构化和半结构化数据进行了优化，可以轻松地扩展到 PB 级，并允许更好地集成各种工具，以帮助企业最大限度地利用其数据。

捕获和分析实际任何类型的数据的能力已经成为一种关键的业务能力。无论格式如何，数据湖都可以广泛接受新数据。这明显不同于传统关系数据库中充满规则、高度结构化的存储。

数据湖是企业数据的中央存储池，我们从各种渠道向它输入信息。这些源可能以非结构化、半结构化和结构化的格式包括从数据库到原始音频和视频片段的任何内容。相反，数据仓库只存放结构化数据。

数据湖被划分为一个或多个数据区域，具有不同程度的转换和清洁度。原始区域是所有其他湖泊区域数据构建的基础。（“store now, find value later”）

数据湖是一个通常重载的术语，在不同的上下文中有不同的含义，但是在所有数据湖定义中有几个重要的属性是一致的：

- 支持非结构化和半结构化数据。
- 可扩展到 PB 及以上级规模。
- 使用类 SQL 的接口与存储的数据交互。
- 能够尽可能无缝地连接各种分析工具。
- 最后，现代数据湖通常是解耦存储和分析工具的组合。

过去几年见证了 Hadoop 作为事实上的大数据平台的崛起和随后的衰落。最初，HDFS 充当存储层，而 Hive 充当分析层。经过大力推广，Hadoop 能够达到几百 TB，允许对半结构化数据进行 SQL 之类的查询，而且在当时已经足够快了。

后来，数据量增长到新的规模，企业的需求变得更加雄心勃勃，即用户现在期望更快的查询时间，更好的可扩展性，易于管理等等。这时，Hive 和 HDFS 开始为新的更好的技术平台让路。

为了解决 Hadoop 的复杂性和扩展性方面的挑战，业界现在正转向一种分解的架构，使用 REST API 将存储层和分析层松耦合起来。这使得每一层都更加独立(在扩展和管理方面)，并允许为每一项工作使用更合适（完美）的工具。例如，在这个分解的模型中，用户可以选择对批处理工作负载使用 Spark 进行分析，而对 SQL 繁重的工作负载使用 Presto, Spark 和 Presto 都使用相同的后端存储平台。

这种方法现在正迅速成为标准。常用的存储平台是对象存储平台，如 AWS S3、Azure Blob 存储、GCS、Ceph、MinIO 等等。分析平台从简单的基于 Python 和 R 的笔记本到 Tensorflow、Spark、Presto、Splunk、Vertica 等多种多样。

新的分解模型(为每个作业使用正确的工具)通常在可扩展性和易于管理方面更好。这种方法在架构上也非常适合。但仍有数据一致性和管理方面的挑战需要解决。

- File 或 Tables：分解模型是指存储系统将数据视为对象或文件的集合。但是终端用户对数据的物理排列不感兴趣，他们想要看到数据的更合乎逻辑的视图。RDBMS 数据库在实现这种抽象方面做得很好。随着大数据平台逐渐成为未来的数据平台，现在人们期望这些系统能够以一种用户友好的方式行事，即不强制用户了解物理存储的任何信息。
- SQL 接口：如上所述，用户不再愿意考虑底层平台的低效率。例如，现在还期望数据湖是 ACID 兼容的，这样终端用户就不会有额外的开销来确保与数据相关的保证。
- 变更管理：在这种规模下管理数据的另一个非常重要的方面是，能够回滚并查看何时发生了什么变化，以及检查特定的细节。目前，使用对象存储的版本管理可以实现这一点，但正如我们前面所看到的，这是在较低的物理细节层，在较高的逻辑级别上可能没有用处。现在用户期望在逻辑层看到版本控制。

可用性挑战和数据一致性需求导致了一个新的软件项目类别。这些项目位于存储平台和分析平台之间，在以本地方式处理对象存储平台的同时，为最终用户提供强 ACID 保证。下面我们从更高的层次来看看其中的一些项目，看看它们之间的比较。

1）Delta Lake

Delta Lake 是一个将 ACID 事务引入 Apache Spark 的开源平台。Delta Lake 是由 Databricks 开发的，可运行在现有的存储平台(S3、HDFS、Azure)之上，并且完全兼容 Apache Spark API。特别地它提供了如下特性：

- Spark 上的 ACID 事务：可序列化的隔离级别确保用户读取数据时永远不会看到不一致的数据。

Delta Lake 允许在现有的数据湖之上添加事务层。既然在它之上有了事务事务，就可以确保拥有可靠、高质量的数据，并且可以对其进行各种计算。

- 可扩展的元数据处理：利用 Spark 的分布式处理能力，轻松处理具有数十亿个文件的 PB 级规模的表的所有元数据。
- 流和批处理的统一：Delta Lake 中的一个表是一个批处理表，同时也是一个流源和 sink。流数据摄入、批量历史回填、交互式查询都是开箱即用的。
- 模式强制：自动处理模式演变，以防止在摄入期间插入坏记录。
- 时间旅行：数据版本化功能，使用户能够关注特定的时间点，支持回滚、完整的历史审计跟踪和可重复的机器学习实验。
- upsert 和 delete：支持 merge、update 和 delete 操作，以支持复杂的使用，如 change-data-capture, slowly-changing-dimension (SCD)操作、流更新，等等。

2）Apache Iceberg

Apache Iceberg 是一种用于大型分析数据集的开放表格格式。Iceberg 向 Presto 和 Spark 添加了使用高性能格式的表，其工作方式与 SQL 表类似。Iceberg 的重点是避免不愉快的意外，帮助模式演化和避免无意的数据删除。用户不需要了解分区就可以获得快速查询。

- 模式演化支持添加、删除、更新或重命名，并且没有副作用。
- 隐藏分区可以防止用户错误导致不正确的结果或非常慢的查询。
- 分区布局演化可以在数据卷或查询模式更改时更新表的布局。
- 时间旅行支持使用完全相同的表快照的可重复查询，或者让用户轻松地检查更改。
- 版本回滚允许用户通过将表重新设置为良好状态来快速纠正问题。

3）Apache Hive

Apache Hive 数据仓库软件已经出现一段时间了。随着新的挑战的出现，Hive 现在正试图解决一致性和可用性问题。它方便了使用 SQL 读取、写入和管理分布存储中的大型数据集。结构可以投射到存储中已经存在的数据上。使用 ACID 语义的事务已经被添加到 Hive 中，以解决以下应用场景：

- 数据流摄取：使用 Apache Flume、Apache Storm 或 Apache Kafka 等工具将数据流传输到 Hadoop 集群中。虽然这些工具可以以每秒数百行或更多行的速度写入数据，但 Hive 只能每 15 分钟到 1 小时添加一次分区。添加分区通常会很快导致表中分区数量过多。这些工具可以将数据流传输到现有的分区中，但这会导致读取器进行脏读(即，在开始查询之后会看到写入的数据)，并在目录中留下许多小文件，从而对 NameNode 造成压力。Hive 现在支持这样的场景，即允许读取器获得一致的数据视图，并避免过多的文件。
- 缓慢变化的维度（SCD）：在典型的星型模式数据仓库中，维度表会随着时间缓慢变化。这些更改导致插入单个记录或更新记录(取决于所选择的策略)。Hive 从 0.14 开始就能够支持这一功能。
- 数据重申：有时收集的数据被发现是不正确的，需要纠正。从 Hive 0.14 开始，可以通过 INSERT、UPDATE 和 DELETE 来支持这些用例。
- 使用 SQL MERGE 语句进行批量更新。

下表列出了上面三种工具的一个高层次比较：

平台	支持的分析工具	文件格式	回滚	压缩和清理	时间旅行
Delta Lake	Apache Spark	Parquet	支持	支持，手动处理	支持
Apache Iceberg	Apache Spark, Presto	Parquet, ORC	支持	不支持	支持
Apache Hive	Hive, Apache Spark	ORC	不支持	手动 & 自动处理	不支持

12.2 Delta Lake 介绍

Apache Spark 没有提供数据处理系统最基本的特性，比如 ACID 事务和其他数据管理功能。为了克服这个不足，Databricks(由 Apache Spark 的第一批工程师建立的组织)发布了 Delta Lake，这是 Spark 的一个开源存储中间层。

基本上，Delta Lake 是一个文件系统，它在对象存储上存储批和流数据，以及用于表结构和模式实施的 Delta 元数据。将数据放入湖中是使用 Delta ACID API 完成的，而将数据从湖中取出是使用 Delta JDBC 连接器完成的。Delta 数据不能被其他 SQL 查询引擎查询。

它的行为非常类似于已解耦的数据库，如 Snowflake 和 BigQuery：一个独立的事务层，用于对象存储，具有专有的元数据、ACID API 和 JDBC 连接器。

我们的程序不直接与存储层交互，而是通过使用 Delta Lake API 与数据湖通信来读写数据。Delta Lake 提供了一个统一的平台，在单一平台上支持批处理和流处理工作负载。它充当计算和存储层之间的中间服务。

Delta Lake 提供了 ACID 事务、快照隔离、数据版本控制和回滚，以及模式强制，以更好地处理模式更改和数据类型更改。在 Delta Lake 表上进行的所有事务都直接存储到磁盘。

Delta Lake 支持两个隔离级别：Serializable 和 WriteSerializable。WriteSerializable 比快照隔离更强大，它提供了可用性和性能的最佳组合，并且是默认设置。最强级别的 Serializable 隔离确保串行序列与表的历史记录完全匹配。

Delta Lake 通过管理提交（commit）的并发性来负责并发的读写访问。这是通过使用乐观锁实现的。这意味着：

- 当一个提交执行开始时，线程快照当前 DeltaLog。
- 提交 actions 完成后，线程会检查 DeltaLog 是否被另一个线程同时更新。
  - 如果没有，它将在 DeltaLog 中记录提交。
  - 否则，它更新其 DeltaTable 视图，并在重新处理步骤之后(如果需要)再次尝试注册提交。

这确保了隔离属性。

在 Delta Lake 表上进行的所有事务都直接存储到磁盘上。这个过程满足 ACID 持久性的特性，这意味着即使在系统出现故障时，它也将持续存在。

模式实施特性通过提供指定模式并帮助实施的能力，帮助防止数据湖中摄入不良数据。它通过提供合理的错误消息，防止错误数据进入系统，甚至在数据被摄入数据湖之前就进入系统，从而防止数据损坏。

在默认情况下，Spark 只支持数据读模式，而不支持写入模式，这在数据一致性上造成了一个大问题。解决问题是很棘手的：

- 哪一个是错误的工作?
- 哪个数据文件导致了这个问题?
- 有多少行是错误的?
- 我们如何纠正它?

- 使系统处于一致状态需要多长时间?
- 我们什么时候可以重新启动失败的作业?

Delta Lake Architecture

铜表:

- 因为数据来自不同的来源，这可能是脏的。因此，它是原始数据的垃圾场。
- 通常保留时间很长(年)。
- 避免容易出错的解析。

银表:

- 由中间数据组成，并应用了一些清理操作。
- 是可查询的，便于调试。

金表:

- 由准备使用的干净数据组成
- 可以使用 Spark 或 presto 读取数据。

如下图所示：

Delta Lake 的核心是事务日志，这是一个中央存储库，用于跟踪用户所做的所有更改，它保证了客户端启动的事务的原子性、一致性、隔离性和持久性。它以 JSON 文件的形式记录每一个更改，并按照更改的顺序进行记录。如果有人做了更改，但是删除了它，仍然会有一个记录来简化审计。它作为真实的单一来源，使用户能够访问 Delta Table 状态的最后一个版本。它提供了序列化能力，这是隔离级别的最强级别。如下图所示：

因此，在 Delta Lake 中，读取一张表也会重放这张表的历史记录，比如表的重命名、修改 Schema 等等操作。

Delta Lake 将客户端执行的每个活动分隔为原子提交，每个原子提交由 actions 组成。成功完成提交的所有 actions 可以确保 DeltaLog 记录提交的操作。对于任何失败的作业，commit 不会记录在 DeltaLog 中。

更详细来说，在 Delta Lake 中的每个 JSON 文件都是一次 commit，这个 commit 是原子性的，保存了事务相关的详细记录。另外，Delta Lake 还可以保证多个用户同时 commit 而不会产生冲突，它用的是一种基于乐观锁处理的方式。这种解决冲突的方案适用于“写比较少、读取比较多”的场景。

假设我们要处理一个非常大的表，有百万级别的文件，那么如何高效的处理元数据呢？Delta Lake 用 Spark 来读取事务日志，然后 Delta Lake 隔一段时间对 commit 做一次合并，之后可以从 Checkpoint 开始应用后续的 commit。

下面以公共零售设置中的 DELTA LAKE 应用场景为例，介绍使用 Delta Lake 实现流合并模式。

涉及多个销售点的零售场景的一个典型模式是，每个销售点定期将其各自的交易汇总信息上传到一

个中心实体(例如，公司总部)，在这个实体中，可以对总销售额进行聚合和分析。

出于我们的目的，让我们假设这些销售地点每个都提交每日的摘要文件，其中详细说明当天发生的每个销售交易。让我们进一步假设这些位置存在于多个时区，操作时间不同，并且这些摘要文件(可能还需要在最后完成之前执行一些人工操作)将在不同时间发送。

出于数据处理的目的，可以将上面的过程想象成一个流场景，这很容易在 Spark 中实现。但实际上，这种性质的上传通常会失败、延迟，甚至重新提交并添加或修改，从而引入需要额外的协调。因此，我们需要一个足够健壮的流目的地，以适应业务流程的动态特性。

Delta Lake 非常适合这种情况。而且，在插入和更新经常同时出现的情况下，我们真正想要的是“merge”模式，也称为“upsert”模式。

在我们的这个例子中，文件通过外部进程被上传到 Data Lake 中一个日期特定的文件夹中，并且通过命名约定与其他文件进行区分。命名约定包括日期、时间和 StoreId(例如，201906210916_1000.json)。

JSON 格式的样本文件内容如下，我们可以看到它包括一个存储标识符、日期和一个交易数组。(注意，出于性能考虑，不建议使用多行 JSON，但这里有意将其用于反映真实的场景。)

我们首先定义销售文档的模式。

然后，我们在销售摘要文件被上传到的文件夹上创建一个流 DataFrame。在此步骤中，我们还将销售事务分散到行上(对 Transactions 列使用 explode 表函数)，以便于稍后进行销售聚合。

我们还派生了一个日期键（DateKey），用于聚合和构建 merge 语句来处理重新提交。

接下来，创建一个 Delta 表：

理想情况下，我们希望直接流到增量表，而不是执行批处理 upsert。

最后，我们查询 Delta Lake 表并查看三个商店的合计销售总额。

后来，我们发现第一个商店重新提交了它的每日销售摘要，尽管之前提交的文件没有被修改，既没有被覆盖也没有被删除。另外，第四个商店现在已经提交了它的摘要文件。

商店 1000 的修改后的销售摘要包含一个额外的事务，以及对现有事务的更改。

但是请注意，处理或检索这些更新的结果不需要进一步的努力。我们只执行与以前完全相同的查询，但是现在我们接收更新后的结果。我们的流合并无缝、透明地协调了数据更改!

这里的关键要点是，由于 Delta Lake 的存在，一旦我们的流程就位，就不需要进一步的操作来更新或协调传入的数据。

12.3 Delta Lake 使用

本教程介绍了 Delta Lake 的许多特性，包括模式实施和模式演化、批处理和流工作负载之间的互操作性、时间旅行以及 Delete 和 Merge 等 DML 命令。展示如何从交互式、批处理和流查询读取和写入增

量表。

Delta Lake 构建数据湖
Delta Lake = Parquet 文件 + Meta 文件 + 一组操作的 API

Delta 0.8.0，支持 Spark 3.0。Delta 最新发布了 1.0.0，支持 Spark 3.1.x。

12.3.1 安装 Delta Lake

当前 Delta Lake 的最新版本为 1.0.0，兼容 Spark 3.1.x。如果使用的是 Spark 2.x.x，请使用 Delta Lake 0.6.1 版本。

交互式安装

Python:
运行以下命令安装或升级 Pyspark(3.0 或以上)：

```
$ pip install --upgrade pyspark
```

然后，使用 Delta Lake 包和其他配置运行 PySpark：

Scala:

项目安装

Maven：pom.xml

SBT: build.sbt

```
libraryDependencies += "io.delta" %% "delta-core" % "0.8.0"
```

对于 Python 项目：

Zeppelin 解释器配置

将 delta-core_2.12-0.8.0.jar 包拷贝到$SPARK_HOME/jars/目录下。
然后，在 Zeppelin 的 Spark 解释器中，添加如下内容：

```
spark.sql.extensions                io.delta.sql.DeltaSparkSessionExtension
spark.sql.catalog.spark_catalog     org.apache.spark.sql.delta.catalog.DeltaCatalog
```

12.3.2 数据读写

将 DataFrame 数据写入到 Delta Lake。

```
val data = spark.range(0, 5).toDF("number")
// data.show
data.write.format("delta").save("file:///home/hduser/data/delta-table/demo")
```

这些操作使用从 DataFrame 推断出来的模式创建一个新的 Delta 表。

从 Delta Lake 读取数据到 DataFrame。

```
val df = spark.read.format("delta").load("file:///home/hduser/data/delta-table/demo")
df.show()
```

修改表数据，使用标准 DataFrame API（注意：模式要保持一致）

Delta Lake 支持几种使用标准 DataFrame API 修改表的操作。下面这个例子运行一个批处理作业来覆盖表中的数据：

```
val data2 = spark.range(5, 10).toDF("number")
data2.write.format("delta").mode("overwrite").save("file:///home/hduser/data/delta-table/demo")
```

再次读取数据，应该只看到添加的值 5-9，因为覆盖了以前的数据。

```
val df2 = spark.read.format("delta").load("file:///home/hduser/data/delta-table/demo")
df2.show()
```

不覆盖的条件更新

Delta Lake 提供了编程 API，用于有条件地将数据更新、删除和合并(upsert)到表中。

使用时间旅行读取旧版本的数据

可以使用时间旅行查询 Delta 表的以前快照。如果希望访问覆盖的数据，可以在覆盖第一组数据之前使用 versionAsOf 选项查询表的快照。

```
// val df3 = spark.read.format("delta").option("versionAsOf", 0).load("file:///home/hduser/data/delta-table/demo")
// val df3 = spark.read.format("delta").load("file:///home/hduser/data/delta-table/demo")
// df3.show()
df3.count()
```

向表写入数据流

还可以使用结构化流写入 Delta 表。

// Delta Lake 事务日志保证精确一次性处理，即使有其他流或批查询并发地运行在该表上。

// 默认情况下，流以追加模式（append mode）运行，它将新记录添加到表中

从表中读取更改流

当流写入 Delta 表时，我们也可以从该表中读取流源。

例如，我们可以启动另一个流查询，以打印对 Delta 表所做的所有更改。

另外，还可以指定结构化流从哪个版本开始，这通过提供 startingVersion 或 startingTimestamp 选项来获得从那一点开始的改变。

构建将 JSON 数据读入 Delta 表、修改表、读取表、显示表历史以及优化表的管道

要创建 Delta 表，可以使用现有的 Apache Spark SQL 代码并将格式从 parquet、csv、json 等更改为 Delta。

```
```

12.3.3 文件移除

Delta Lake 进行了版本控制，因此可以轻松地恢复或访问到数据的旧版本。因此，在某些情况下，Delta lake 需要存储多个版本的数据才能实现回滚功能。

存储相同数据的多个版本可能代价昂贵，因此 Delta lake 包含了一个 vacuum 命令，可以删除数据的旧版本。下面我们解释如何使用 vacuum 命令及其适用场景。

首先使用下面的 people1.csv 文件来构造一个 Delta Lake 表：

```
first_name,last_name,country
miguel,cordoba,colombia
luisa,gomez,colombia
li,li,china
wang,wei,china
hans,meyer,germany
mia,schmidt,germany
```

加载该文件，创建 Delta Lake：

```
```

下面查看该 Delta Lake 的内容：

```
val path = "/delta_data_lake/vacuum_example/"
val df = spark.read.format("delta").load(path)
df.show()
```

输出结果如下所示：

```
```

下面我们用另一个 people2.csv 文件覆盖 Delta Lake 中的数据：

```
first_name,last_name,country
lou,bega,germany
bradley,nowell,usa
```

覆盖 Delta Lake 的代码如下：

```
```

下面我们查看一下覆盖后的数据：

```
val df = spark.read.format("delta").load(outputPath)
df.show()
```

覆盖后的数据如下：

```
```

在终端窗口中，执行如下的命令来查看当前的文件系统：

```
$ hdfs dfs -ls /delta_data_lake/vacuum_example/
```

下面是第二次写入后文件系统的内容：

可以看到，虽然第一次写入的数据不再读到我们的 DataFrame 中，但它仍然存储在文件系统中。所以我们可以回滚到数据的旧版本。

下面显示版本 0 时的 Delta Lake 内容：

```
val df = spark.read.format("delta").option("versionAsOf", 0).load(outputPath)
df.show()
```

输出内容如下：

Delta lake 提供了一个 vacuum 命令，用于删除数据的旧版本(任何比指定的保留期限早的数据)。让我们运行 vacuum 命令并验证文件系统中是否删除了文件。

```
import io.delta.tables._

val deltaTable = DeltaTable.forPath(spark, outputPath)
deltaTable.vacuum(0.000001)
```

我们将保留时间设置为 0.000001 小时，这样我们就可以立即运行这个 vacuum 命令。

注：当设置保留的时间小于 168 hours 时，需要设置：

```
spark.databricks.delta.retentionDurationCheck.enabled = false
```

运行 vacuum 命令后文件系统的样子如下：

我们可以看看 00000000000000001.json 文件，以了解 Delta 如何知道要删除哪些文件：

JSON 文件的 remove 部分表明 part-00000-d7ec54f9-....snappy.parquet 可以在执行 vacuum 命令时删除。在执行 vacuum 命令后，我们就无法访问 Delta lake 的 0 版本了：

```
val df = spark.read.format("delta").option("versionAsOf", 0).load(outputPath)
df.show()
```

这时候访问 0 版本，会抛出以下错误信息：

对于旧的数据，保存期系统默认为 7 天。因此 deltable .vacuum()不会做任何事情，除非我们等待 7 天（168 个小时）才能运行该命令。需要设置特殊的命令来调用保留时间小于 7 天的 vacuum 方法，否则会抛出错误信息。我们需要更新 Spark 配置，以允许小于 168 小时的保留时间。

如果是使用 Zeppelin Notebook，则可以在 Spark 解释器中添加这个配置，如下图所示：

spark.databricks.delta.retentionDurationCheck.enabled	false

有的时候，vacumme 什么也不会做。我们看一下下面这个示例，仅添加数据到 Delta Lake，这样的话，运行 vacumm 命令将不会做什么事情。

dogs1.csv：

```
first_name,breed
fido,lab
spot,bulldog
```

dogs2.csv：

```
first_name,breed
fido,beagle
lou,pug
```

下面的代码中，将 dog1.csv 和 dogs2.csv 写成 Delta Lake 文件。

以下是在这两个文件被写入后 Delta Lake 所包含的内容：

查看对应的文件系统，如下所示：

下面是 0000000000000000.json 文件的内容：

下面是 0000000000000001.json 文件的内容：

所有 JSON 文件都不包含任何 remove 行，因此 vacuum 不会删除任何文件。所以当执行下面这段代码后没有改变任何东西：

```
import io.delta.tables._

val path = "/delta_lake_data/vacuum_example2"
val deltaTable = DeltaTable.forPath(spark, path)
deltaTable.vacuum(0.000001)
```

由上面的内容可知，如果想节省数据存储成本，可以使用 vacuum()从 Delta Lake 中删除文件。

在运行 Overwirte 操作后，经常会有重复的文件。在_delta_log/下的 JSON 文件中，任何比指定的保留时间更早且被标记为 remove 的文件都将在执行 vacuum 时被删除。

12.3.4 压缩小文件

数据湖可以积累很多小文件，特别是当它们被增量更新时。文件小，读取速度就会变慢。我们可以用 Spark 压缩 Delta Lake 中的小文件。通过压缩将小文件连接到大文件中是保持快速读取的一种重要的数据湖维护技术。

下面我们创建一个包含 1000 个文件的 Delta 数据湖，然后将文件夹压缩为只包含 10 个文件。

查看相应的日志文件/delta_data_lake/compact/_delta_log/00000000000000000000.json，会看到其中包含有 1000 行类似下面这样的 json 行：

让我们将数据压缩到只包含 10 个文件。

现在/delta_data_lake/compact 包含 1010 个文件，其中 1000 个原始未压缩的文件以及 10 个压缩过的文件。

这时查看/delta_data_lake/compact/_delta_log/00000000000000000001.json，会看到其中包含有 10 行类似下面这样的 json 行：

另外还包含有 1000 行类似下面这样的 json 行：

该 JSON 文件中的 remove 部分表明当运行 vacuum 命令时，part-00097-.....-c000.snappy.parquet 文件

会被删除。

Delta lake 包括一个删除旧版本数据的 vacuum 命令。可以运行 vacuum()命令删除旧的数据文件，这样就不必存储未压缩的数据。

```
deltaTable.vacuum(0.000001)
```

注：当设置保留的时间小于 168 hours 时，需要设置：

```
spark.databricks.delta.retentionDurationCheck.enabled = false
```

这里我们将保留时间设置为 0.000001 小时，这样我们就可以立即运行这个 vacuum 命令。在执行 vacuum 命令后，我们无法访问 Delta lake 的 0 版本，否则会出现错误。如下所示：

```
```

运行上面的代码，会出现类似下面这样的错误信息：

```
.
.....
```

关于 dataChange=false

Delta 事务协议能够将事务日志中的条目标记为 dataChange=false，表示它们只重新安排已经是表的一部分的数据。这非常强大，因为它允许执行压缩和其他读性能优化，而不会破坏使用 Delta 表作为流源的能力。我们应该将其公开为用于重写的 DataFrame 写入器选项。

Delta lake 目前在数据压缩时设置 dataChange=true，这对下游流消费者来说是一个重大改变。当文件被压缩时，用户可以选择设置 dataChange=false，这时 Delta lake 将被更新，这样压缩对于下游流客户来说不是一个中断操作。

合并分区的 Delta Lake

假设我们的数据存储在/some/path/data 文件夹中，并按 year 字段进行分区。进一步假设 2019 目录包含 5 个文件，我们希望将其压缩为一个文件。

下面是我们如何压缩 2019 分区。

```
```

12.3.5 时间旅行

Delta Lake 支持增量更新。Delta Lake 使用事务日志存储数据湖元数据，并通过事务日志允许我们在给定的时间点进行时间旅行和数据探索。下面我们通过一个示例来演示 Delta Lake 中的时间旅行特性。

首先，我们从一个 CSV 文件创建一个 Delta Lake。下面是我们将要使用的 CSV 数据。

people.csv：

```
first_name,last_name,country
miguel,cordoba,colombia
luisa,gomez,colombia
li,li,china
wang,wei,china
hans,meyer,germany
mia,schmidt,germany
```

接下来，我们使用 Spark SQL API 读取这个 CSV 文件到一个 DataFrame 中，然后将该 DataFrame 作为一个 Delta 数据湖写出。代码如下：

执行上面的代码后，person_data_lake 目录将包含以下文件：

数据以 Parquet 格式保存在文件中，元数据以_delta_log/00000000000000000000.json 格式保存在文件中。JSON 文件包含关于写事务、数据模式以及添加了什么文件的信息。让我们检查一下 JSON 文件的内容。

增量更新 Delta 数据湖

下面我们使用一些纽约市出租车数据来构建并增量更新一个 Delta 数据湖。

首先我们构建一个初始的 Delta 数据湖。代码如下：

这段代码会创建一个 Parquet 文件和一个_delta_log/00000000000000000000.json 文件。查看文件系统结构，如下所示：

接下来让我们检查 Delta 数据湖的内容。代码如下：

得到的输出内容如下所示：

在第一次加载后该 Delta Lake 包含 5 行数据。

下面我们将另一个文件加载到同一个 Delta Lake 中，使用 SaveMode.Append 模式。代码如下：

这段代码创建了一个 Parquet 文件和一个_delta_log/ 00000000000000001.json 文件。文件系统目录 incremental_data_lake 现在包含如下这些文件：

再一次加载该 Delta Lake。在加载文件后，现在 Delta Lake 包含 10 行数据：

得到的输出内容如下所示：

时间旅行

Delta 提供了进行时间旅行的支持，并在给定数据加载时探索数据湖的状态。下面我们编写一个查询来检查第一次数据加载之后(忽略第二次数据加载)增量更新的 Delta 数据湖。

得到的输出内容如下所示：

在上面的代码中，option("versionAsOf",0)告诉 Delta 只抓取_delta_log/00000000000000000000.json 中的文件，而忽略_delta_log/00000000000000000001. json 中的文件。

假设我们正在数据湖上训练一个机器学习模型，并希望在实验时保持数据不变。Delta Lake 让我们在训练模型时很容易使用单一版本的数据。

也可以轻松访问 Delta Lake 事务日志的完整历史。

得到的输出内容如下所示：

Delta history 历史表的模式如下：

```
fullHistoryDF.printSchema()
```

得到的输出内容如下所示：

我们还可以通过时间戳获取 Delta 表版本。

这与抓取我们的 Delta 表的版本 1 是一样的(大家可以检查一下事务日志历史输出看看为什么)：

12.3.6 合并更新列和执行 upsert

这一节，我们将解释如何更新表列，以及如何使用 merge 命令执行 upserts。

Parquet 文件是不可变的，不支持更新。Delta Lake 提供了 merge 语句来提供类似于更新的接口，但实际上，这些并不是真正的更新，不会改变底层文件。

Delta Lake 只是在重写整个 Parquet 文件。这将使大型数据集上的 upsert 或 update 列语句非常慢。

merge 示例

下面我们通过一个示例来理解和掌握使用 merge 命令执行 upserts 的方法。首先让我们将一些 CSV 数据转换为 Delta Lake：

events.csv：

```
eventType,websitePage
click,homepage
clck,about page
mouseOver,logo
```

读取该数据文件，并写出到 Delta Lake 中：

让我们看一下_delta_log/ 00000000000000000000.json 事务日志文件中存储了什么内容。

让我们来展示一下数据湖的内容：

可以得到如下的输出结果：

注意，第二行数据在 eventType 字段中有一个输入错误。它说的是“clck”而不是“click”。让我们写一些代码来更新这个错误。

执行上面的代码，然后可以检查 Delta Lake 的内容，确认拼写错误是否已经修正。

得到如下的输出结果：

Parquet 文件是不可变的，那么 Delta 是怎样做到修复拼写错误呢?

让我们看一下_delta_log/ 00000000000000000001.json 来弄清楚背后的情况。

merge 命令将一个新文件写入文件系统。让我们检查一下新文件的内容。

执行上面的代码，输出如下的内容：

下面是运行 merge 命令后文件系统中的所有文件。

因此，merge 命令将所有数据写入一个全新的文件中。不幸的是，它不能进入现有的 Parquet 文件，只更新需要更改的单元格。

写出所有数据将使 merge 运行要慢得多。

upsert 示例

首先我们用下面的数据集，建立另一个小 Delta Lake。

original_data.csv：

```
date,eventId,data
2019-01-01,4,take nap
2019-02-05,8,play smash brothers
2019-04-24,9,speak at spark summit
```

其后使用下面的代码，将上面的数据构建为 Delta Lake：

看看 Delta Lake 的初始状态：

得到如下的输出内容：

接下来我们用另一种更“妈妈友好”的事件说法来更新 Delta Lake。首先构造这个妈妈友好的数据。

mom_friendly_data.csv：

在上面的新数据集中，第 4 和第 8 个事件的描述会让妈妈感到骄傲。66 号事件将被添加到湖中，让妈妈感觉良好。

下面我们执行 upsert 操作。

执行上面的代码，然后我们查看一下 upsert 之后 Delta Lake 的内容。

输出内容如下：

upserts 的事务日志

在_delta_log /00000000000000000000.json 文件中包含单个项，用于添加的单个 Parquet 文件。

由_delta_log /00000000000000000001.json 文件显示，upserts 在事务日志中添加了大量记录。

下面我们创建一个小辅助方法，以便轻松检查这些 Parquet 文件的内容：

依次查看各个 Parquet 文件的内容。
执行：
```
displayEventParquetFile("part-00000-36aafda3-530d-4bd7-a29b-9c1716f18389-c000")
```
得到内容如下：

执行：
```
displayEventParquetFile("part-00026-fcb37eb4-165f-4402-beb3-82d3d56bfe0c-c000")
```
得到内容如下：

执行：
```
displayEventParquetFile("part-00139-eab3854f-4ed4-4856-8268-c89f0efe977c-c000")
```
得到内容如下：

执行：
```
displayEventParquetFile("part-00166-0e9cddc8-9104-4c11-8b7f-44a6441a95fb-c000")
```
得到内容如下：

执行：
```
displayEventParquetFile("part-00178-147c78fa-dad2-4a1c-a4c5-65a1a647a41e-c000")
```
得到内容如下：

这个更新代码创建了数量惊人的 Parquet 文件。

12.3.7 小结

Delta Lake 提供了统一数据科学、数据工程和生产工作流程的功能，这是机器学习生命周期的理想选择。它提供了时间旅行，允许数据更改在需要时回滚或复制。它确保数据以正确的格式被处理，即模式强制。它还提供了模式演化，防止现有模型因模式更改而崩溃。Delta 通过在一个存储系统中获得多个存储系统的好处，从而降低了复杂性。Delta 支持更简单的数据架构，使组织能够专注于从数据中提取价值。

12.4 Delta 架构

。。。。。。

第 13 章 Iceberg 数据湖

Apache Iceberg 是一种相对较新的开源表格式，用于存储 PB 级数据集。Iceberg 很容易地融入到现有的大数据生态系统中。通过使用保存在每个表上的大量元数据，Iceberg 提供了传统上其他表格式无法提供的函数。这包括模式演进、分区演进和表版本回滚，而不需要进行代价高昂的表重写或表迁移。

13.1 在 Spark 3 中使用 Iceberg

方式一，在启动 spark-shell 时指定：

```
$ spark-shell --packages org.apache.iceberg:iceberg-spark3-runtime:0.11.1
```

方式二，将 iceberg-spark3-runtime:0.11.1.jar 包添加到 Spark 的 jars 目录下。

方式三，如果是使用 Maven 开发，则添加如下依赖：

```
```

13.2 配置和使用 catalog

在介绍如何使用 Iceberg 之前，先简单地介绍一下 Iceberg catalog 的概念。catalog 是 Iceberg 对表进行管理（create、drop、rename 等）的一个组件。目前 Iceberg 主要支持 HiveCatalog 和 HadoopCatalog 两种 Catalog。其中 HiveCatalog 将当前表 metadata 文件路径存储在 Metastore，这个表 metadata 文件是所有读写 Iceberg 表的入口，所以每次读写 Iceberg 表都需要先从 Metastore 中取出对应的表 metadata 文件路径，然后再解析这个 Metadata 文件进行接下来的操作。而 HadoopCatalog 将当前表 metadata 文件路径记录在一个文件目录下，因此不需要连接 Metastore。

Catalog 名称在 SQL 查询中用于标识一个表。通过添加属性 spark.sql.catalog.(catalog-name)及其值的实现类来创建和命名 catalog。

Iceberg 提供了两种实现，两个 catalog 都是使用嵌套在 catalog 名称下的属性来配置的：

- org.apache.iceberg.spark.SparkCatalog：支持 Hive Metastore 或 Hadoop warehouse 作为 catalog。
- org.apache.iceberg.spark.SparkSessionCatalog：将对 Iceberg 表的支持添加到 Spark 的内置 catalog 中，并将非 Icegerg 表委托给内置 catalog。

例如，下面的命令为$PWD/warehouse 下的表创建一个名为 local 的基于路径的 hadoop catalog，并对 Iceberg 表添加 Spark 内置 catalog 的支持。

```
```

说明 1：

Iceberg 0.11.0 及以后的版本向 Spark 增加了一个扩展模块，以添加新的 SQL 命令，如 CALL 调用存储过程或 ALTER TABLE…WRITE ORDERED BY。使用这些 SQL 命令需要使用以下 Spark 属性将 Iceberg 扩展添加到 Spark 环境中。（SQL 扩展在 Spark 2.4 中不可用）

```
spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions
```

作者注：在实测过程中，Iceberg 0.11.0 和 Icegerg 0.11.1 在添加了该扩展模块以后，均会引发执行错

误。所以目前是来说还应该是个 Bug。

说明 2：Spark 2.4 中的 catalog

当使用 Iceberg 0.11.1 时，Spark 2.4 可以从多个 Iceberg catalog 或从表位置加载表。Spark 2.4 中的 catalog 配置与 Spark 3.0 中的 catalog 类似，但只支持 Iceberg catalog。

13.2.1 使用 catalog

Spark 3.0 增加了一个 API 来可插拔式地插入表 catalog，用于加载、创建和管理 Iceberg 表。例如，要创建一个名为 hive_prod 的 Iceberg catalog 来从 Hive metastore 加载表，使用如下的配置：

```
```

如果省略 uri，则使用与 Spark 相同的 uri，即在 hive-site.xml 中的 hive.metastore.uris。

注：基于 hive 的 catalog 只加载 Iceberg 表。要在同一个 Hive metastore 中加载非 Iceberg 表，需要使用 session catalog。

Iceberg 还支持 HDFS 中基于目录的 catalog，可以使用 type=hadoop 来配置：

```
```

在上面的例子中，hive_prod 和 hadoop_prod 可被用于前缀将从这些 catalog 加载的数据库和表名称。

```
SELECT * FROM hive_prod.db.table        -- 从 catalog hive_prod 加载 db.table
```

Spark 3 会跟踪当前的 catalog 和命名空间，这些可以从表名中省略。

```
USE hive_prod.db;
SELECT * FROM table                     -- 从 catalog hive_prod 加载 db.table
```

要查看当前的 catalog 和命名空间，运行：

```
SHOW CURRENT NAMESPACE
```

13.2.2 替换 session catalog

要向 Spark 的内置目录添加 Iceberg 表支持，配置 spark_catalog 使用 Iceberg 的 SparkSessionCatalog。

```
spark.sql.catalog.spark_catalog = org.apache.iceberg.spark.SparkSessionCatalog
spark.sql.catalog.spark_catalog.type = hive
```

Spark 内置的 catalog 支持在 Hive Metastore 中跟踪现有的 v1 和 v2 表。这将配置 Spark 使用 Iceberg 的 SparkSessionCatalog 作为该 session catalog 的包装器。当表不是 Iceberg 表时，将使用内置 catalog 来加载它。

这个配置可以对 Iceberg 表和非 Iceberg 表使用相同的 Hive Metastore。

13.2.3 加载自定义的 catalog

Spark 通过指定 catalog-impl 属性支持加载自定义的 Iceberg Catalog 实现。当 catalog-impl 被设置时，type 的值将被忽略。下面是一个例子：

```
```

13.2.4 运行时配置

可以在运行时对 Spark 3 读写 Iceberg 进行动态配置。

读选项配置

Spark read options 是在配置 DataFrameReader 时被传递，如下所示：

Spark option	Default	Description
snapshot-id	(latest)	要读取的表快照的ID
as-of-timestamp	(latest)	以毫秒为单位的时间戳：所使用的快照将是此时的当前快照
split-size	As per table property	重写这个表的read.split.target-size和read.split.metadata-target-size
lookback	As per table property	重写这个表的read.split.planning-lookback
file-open-cost	As per table property	重写这个表的read.split.open-file-cost
vectorization-enabled	As per table property	重写这个表的read.parquet.vectorization.enabled
batch-size	As per table property	重写这个表的read.parquet.vectorization.batch-size

写选项配置

Spark write options 在配置 DataFrameWriter 时被传递，如下所示:

Spark option	Default	Description
write-format	write.format.default	写操作使用的文件格式：parquet，avro或 orc
target-file-size-bytes	As per table property	覆盖这个表的write.target-file-size-bytes
check-nullability	true	对字段设置nullable检查
snapshot-property.custom-key	null	在快照摘要中添加具有自定义key和相应value的条目
fanout-enabled	false	覆盖这个表的write.spark.fanout.enabled
check-ordering	true	检查输入模式和表模式是否相同

全部配置，请参考 http://iceberg.apache.org/configuration/#catalog-properties。

13.2.5 读写 Iceberg 表

添加 catalog

这个命令为$PWD/warehouse 下的表创建一个名为 local 的基于路径的目录，并将对 Iceberg 表的支持添加到 Spark 的内置目录中：

如果是在 Zeppelin 中，则使用%spark.conf 解释器来进行配置，如下所示：

创建表

要在 Spark 中创建 Iceberg 表，使用 spark-shell 或 spark.sql(...)来运行 CREATE TABLE 命令：

```
-- local 是上面定义的基于路径的 catalog
CREATE TABLE local.db.table (id bigint, data string) USING iceberg
```

Iceberg catalog 支持所有的 SQL DDL 命令，包括：

写入
一旦表被创建，就可以使用 insert INTO 来插入数据：

可以看到类似下面这样的查询结果：

Iceberg 还向 Spark 添加了行级 SQL 更新，MERGE INTO 和 DELETE FROM：

Iceberg 支持使用新的 v2 DataFrame write API 来写 DataFrame：

读取
要使用 SQL 读取数据，在 SELECT 查询中使用 Iceberg 表名：

查询结果如下所示：

SQL 也是检查表的推荐方法。要查看一个表中的所有快照，使用 snapshots 元数据表：

```
SELECT * FROM local.db.table.snapshots;
```

也支持 DataFrame 读取，可以使用 spark.table 按名称引用表：

```
val df = spark.table("local.db.table")
df.count()
```

13.3 DDL 命令

Iceberg 使用 Apache Spark 的 DataSourceV2 API 实现数据源和 catalog。Spark DSv2 是一个不断发展的 API，在 Spark 版本中提供了不同级别的支持。Spark 2.4 不支持 SQL DDL。

注：Spark 2.4 不能使用 DDL 创建 Iceberg 表，应该使用 Spark 3.x API 或 Iceberg API。

13.3.1 创建和删除表

Spark 3.0 可以使用 USING Iceberg 子句在任何 Iceberg catalog 中创建表：

```
CREATE TABLE prod.db.sample (
    id bigint COMMENT 'unique id',
    data string
) USING iceberg
```

Iceberg 将把 Spark 中的列类型转换为相应的 Iceberg 类型。

表创建命令，包括 CTAS 和 RTAS，支持所有 Spark create 子句，包括：

- PARTITION BY (partition-expressions)：配置分区。

- LOCATION '(fully-qualified-uri)' ：设置表位置。
- COMMENT 'table documentation' ：设置一个表的描述信息。
- TBLPROPERTIES ('key'='value', ...)：设置表配置信息。

表创建命令也可以使用 USING 子句设置默认格式。这只支持 SparkCatalog，因为 Spark 对内置 catalog 的 USING 子句处理方式不同。

PARTITION BY

使用 partition by 创建一个分区表。

```
```

PARTITIONED BY 子句支持转换表达式来创建隐藏的分区。

```
```

支持的转换有：

- years(ts)：按年分区。
- months(ts)：按月分区。
- days(ts)或 date(ts)：按天分区。
- hours(ts)或 date_hour(ts)：按天和小时分区。
- bucket(N, col)：按哈希值对 N 个分桶取模来分区。
- truncate(L, col)：以截断为 L 的值进行分区。
  - 字符串会被截断到给定的长度。
  - Integer 和 Long 会被截断到 bins：truncate(10,i)产生分区 0、10、20、30、...

CTAS

create table ... as select

在使用 SparkCatalog 时，Iceberg 支持 CTAS 作为原子操作。在使用 SparkSessionCatalog 时 CTAS 虽然是受支持的，但不是原子的。

```
CREATE TABLE prod.db.sample
USING iceberg
AS SELECT ...
```

RTAS

replace table ... as select

当使用 SparkCatalog 时，Iceberg 支持 RTAS 作为原子操作。在使用 SparkSessionCatalog 时 RTAS 虽然是受支持的，但不是原子的。

原子表替换使用 SELECT 查询的结果创建一个新的快照，但保留表历史。

```
...
```

如果更改，模式和分区规范将被替换。为了避免修改表的模式和分区，请使用 INSERT OVERWRITE 而不是 REPLACE TABLE。REPLACE TABLE 命令中的新表属性将与任何现有的表属性合并。如果更改了现有的表属性，则会更新它们，否则它们就会被保存下来。

如果想要删除一张表，使用 drop table 命令。如下：

```
DROP TABLE prod.db.sample
```

13.3.2 修改表

在 Spark 3 中，Iceberg 支持 ALTER TABLE 命令对表结构进行修改，包括:

- 重命名表
- 设置或删除表属性
- 添加、删除和重命名列
- 添加、删除和重命名嵌套字段
- 新排序顶级列和嵌套的 struct 字段
- 扩大 int、float 和 decimal 字段的类型
- 使必需列为可选列

此外，可以使用 SQL 扩展来添加对分区演化和设置表的写顺序的支持。

ALTER TABLE ... RENAME TO

```
ALTER TABLE prod.db.sample RENAME TO prod.db.new_name
```

ALTER TABLE ... SET TBLPROPERTIES

```
ALTER TABLE prod.db.sample SET TBLPROPERTIES (
    'read.split.target-size'='268435456'
)
```

Iceberg 使用表属性来控制表行为。UNSET 用于删除属性：

```
ALTER TABLE prod.db.sample UNSET TBLPROPERTIES ('read.split.target-size')
```

ALTER TABLE ... ADD COLUMN

要向 Iceberg 添加一个列，可以使用 ALTER TABLE 中的 ADD COLUMNS 子句：

```

```

可以同时添加多个列，以逗号分隔。

嵌套的列应该使用完整的列名来标识：

```

```

在 Spark 2.4.4 及以后版本中，可以通过添加 FIRST 或 AFTER 子句在任何位置添加列：

```

```

ALTER TABLE ... RENAME COLUMN

Iceberg 允许对任何字段进行重命名。要重命名一个字段，使用 RENAME COLUMN：

```
ALTER TABLE prod.db.sample RENAME COLUMN data TO payload
ALTER TABLE prod.db.sample RENAME COLUMN location.lat TO latitude
```

注意，嵌套的重命名命令只重命名叶字段。上面的命令将 location.lat 重命名为 location.latitude。

ALTER TABLE ... ALTER COLUMN

Alter column 可用于扩展类型、使字段可选、设置注释和重新排序字段。

如果修改是安全的，Iceberg 允许改变列类型。安全的列类型修改：

- int => bigint
- float => double

- decimal(P,S) => decimal(P2,S) ，当 P2 > P (规模不能改变)

```
ALTER TABLE prod.db.sample ALTER COLUMN measurement TYPE double
```

若要从 struct 中添加或删除列，使用具有嵌套列名的“ADD COLUMN”或 “DROP COLUMN”。

列注释也可以使用 ALTER COLUMN 进行更新：

```
ALTER TABLE prod.db.sample ALTER COLUMN measurement TYPE double COMMENT '单位是 bps'
ALTER TABLE prod.db.sample ALTER COLUMN measurement COMMENT 'unit is kilobytes per second'
```

Iceberg 允许使用 FIRST 和 AFTER 子句对结构中的顶级列或列进行重新排序：

```
ALTER TABLE prod.db.sample ALTER COLUMN col FIRST
ALTER TABLE prod.db.sample ALTER COLUMN nested.col AFTER other_col
```

可以使用 SET NOT NULL 和 DROP NOT NULL 来更改是否允许为空。

```
ALTER TABLE prod.db.sample ALTER COLUMN id DROP NOT NULL
```

注：ALTER 列不用于更新结构(struct)类型。使用 ADD COLUMN 和 DROP COLUMN 来添加或删除结构字段。

ALTER TABLE ... DROP COLUMN
要删除列，使用 ALTER TABLE ... DROP COLUMN：

```
ALTER TABLE prod.db.sample DROP COLUMN id
ALTER TABLE prod.db.sample DROP COLUMN point.z
```

13.3.3 修改表 SQL 扩展

这些命令在 Spark 3.x 中可用，当使用 Iceberg SQL 扩展时。

ALTER TABLE ... ADD PARTITION FIELD
Iceberg 支持使用 ADD PARTITION FIELD 向规范中添加新的分区字段：

```
ALTER TABLE prod.db.sample ADD PARTITION FIELD catalog            -- identity transform
```

也支持分区转换：

```
```

添加分区字段是一个元数据操作，不会更改任何现有的表数据。新数据将与新分区一起写入，但现有数据将保留在旧分区布局中。对于元数据表中的新分区字段，旧数据文件将具有空值。

当表的分区发生变化时，动态分区覆盖行为也会改变，因为动态覆盖会隐式地替换分区。要显式地覆盖，使用新的 DataFrameWriterV2 API。

注意：要使用转换从每日分区迁移到每小时分区，不需要删除每日分区字段。保留该字段可确保现有元数据表查询继续工作。

警告：当分区发生变化时，动态分区覆盖行为将发生变化。例如，如果按天分区转换到按小时分区，

重写将覆盖按小时划分的分区，但不再覆盖按天划分的分区。

ALTER TABLE ... DROP PARTITION FIELD

可以使用 DROP PARTITION FIELD 删除分区字段。

注意，尽管分区被删除了，但列仍然存在于表模式中。

删除分区字段是一个元数据操作，不会更改任何现有的表数据。新数据将与新分区一起写入，但现有数据将保留在旧分区布局中。

警告：删除分区字段时要小心，因为它将更改元数据表(如文件)的模式，并可能导致元数据查询失败或产生不同的结果。

ALTER TABLE ... WRITE ORDERED BY

可以配置 Iceberg 表的排序顺序，该顺序用于自动排序某些引擎中写入表的数据。例如，MERGE INTO 在 Spark 中将使用表的排序。

要设置表的写顺序，使用 WRITE ORDERED BY：

注：表写顺序不能保证查询的数据顺序。它只影响数据写入表的方式。

13.4 查询数据

Iceberg 使用 Apache Spark 的 DataSourceV2 API 实现数据源和 catalog。Spark DSv2 是一个不断发展的 API，在 Spark 版本中提供了不同级别的支持:

13.4.1 使用 SQL 查询

在 Spark 3 中，表使用包含 catalog 名的标识符。

```
-- catalog: prod, namespace: db, table: table
SELECT * FROM prod.db.table
```

元数据表，如 history 和 snapshots，可以使用 Iceberg 表名作为命名空间。

例如，从 prod.db.table 的 files 元数据表中读取，运行：

```
SELECT * FROM prod.db.table.files
```

13.4.2 使用 DataFrame 查询

要将表加载到一个 DataFrame，使用 table：

```
val df = spark.table("prod.db.table")
```

在 Spark 3 和 Spark 2.4 中，Iceberg 0.11.1 都为 DataFrameReader 增加了多 catalog 支持。

可以通过 Spark 的 DataFrameReader 接口加载路径 path 和表名。表的加载方式取决于标识符的指定方式。当使用 spark.read.format("iceberg").path(table)或 spark.table(table)时，table 变量可以有以下几种形式：

- file:/path/to/table：加载一个给定路径的 HadoopTable。
- tablename：加载 currentCatalog.currentNamespace.tablename。
- catalog.tablename：从指定的 catalog 加载 tablename。
- namespace.tablename：从当前 catalog 加载 namespace.tablename。
- catalog.namespace.tablename：从指定的 catalog 加载 namespace.tablename。
- namespace1.namespace2.tablename：从当前 catalog 加载 namespace1.namespace2.tablename。

上面的列表是按先后顺序排列的。例如：匹配的 catalog 将优先于任何名称空间解析。

Time travel

为了选择特定的表快照或某个时间点的快照，Iceberg 支持两个 Spark 读选项：

- snapshot-id：选择一个特定的表快照。
- as-of-timestamp：选择某个时间戳(以毫秒为单位)的当前快照。

```
// 时间旅行到 1986-10-26 01:21:00
```

注：Spark 目前不支持在 DataFrameReader 命令中使用 option with table。所有选项将被无声地忽略。当尝试时间旅行或使用其他选项时，不要使用 table。Options 选项将在 Spark 3.1 - Spark-32592 中支持 table。

Spark 的 SQL 语法还不支持时间旅行。

Spark 2.4

Spark 2.4 需要使用 iceberg format 的 DataFrame reader，因为 2.4 不支持直接的 SQL 查询：

```
```

要在 Spark 2.4 版的 Iceberg 表上运行 SQL SELECT 语句，需要将 DataFrame 注册为一个临时表：

```
```

13.4.3 探索表

为了检查表的历史、快照和其他元数据，Iceberg 支持元数据表。

元数据表通过在原始表名之后添加元数据表名来标识。例如，db.table 的历史记录是通过 db.table.history 读取的。

注：从 Spark 3.0 开始，检查的表名格式(catalog.database.table.metadata)与 Spark 的默认 catalog(spark_catalog)不兼容。如果已经替换了默认目录，则可能需要使用 DataFrameReader API 来检查表。

History

要显示表历史，执行以下命令：

```
SELECT * FROM prod.db.table.history
```

输出结果如下：

注：这显示了一个回滚的提交。该示例有两个具有相同父节点的快照，其中一个不是当前表状态的祖先。

Snapshots

要显示一个表的有效快照，执行命令：

```
SELECT * FROM prod.db.table.snapshots
```

输出结果如下：

还可以将快照连接到表历史记录。例如，下面的查询将显示表历史，以及写每个快照的应用程序 ID：

输出结果如下：

Files

要显示一个表的数据文件和每个文件的元数据，运行：

```
SELECT * FROM prod.db.table.files
```

输出结果如下所示：

Manifests

要显示一个表的文件清单(manifests)和每个文件的元数据，运行：

```
SELECT * FROM prod.db.table.manifests
```

输出结果如下所示：

使用 DataFrame 进行探索

在 Spark 2.4 或 Spark 3 中可以通过 DataFrameReader API 加载元数据表：

```
// 命名的元数据表
spark.read.format("iceberg").load("prod.db.table.files").show(truncate = false)

// Hadoop path 表
spark.read.format("iceberg").load("hdfs://nn:8020/path/to/table#files").show(truncate = false)
```

13.5 写数据

Iceberg 使用 Apache Spark 的 DataSourceV2 API 实现数据源和 catalog。Spark DSv2 是一个不断发展的 API，在 Spark 版本中提供了不同级别的支持：

13.5.1 使用 SQL 写

Spark 3 支持 SQL 中的 INSERT INTO、MERGE INTO 和 INSERT OVERWRITE，以及新的 DataFrameWriterV2 API。

INSERT INTO

若要向表添加新数据，使用 INSERT INTO。

```
INSERT INTO prod.db.table VALUES (1, 'a'), (2, 'b')

INSERT INTO prod.db.table SELECT ...
```

MERGE INTO

Spark 3 增加了对 MERGE INTO 查询的支持，可以表示行级更新。

通过重写包含在 overwrite 提交中需要更新的行的数据文件，Iceberg 支持 MERGE INTO。

建议使用 MERGE INTO 而不是 INSERT OVERWRITE，因为 Iceberg 可以仅替换受影响的数据文件，而且如果表的分区发生变化，被动态重写覆盖的数据可能会发生变化。

使用来自另一个查询的一组更新(称为 source)，MERGE INTO 更新一个表(称为 target 表)。使用类似于连接条件的 ON 子句可以找到 target 表中某一行的更新。MERGE INTO 的语法如下：

```
MERGE INTO prod.db.target t      -- target 表
USING (SELECT ...) s             -- source 更新
ON t.id = s.id                   --  用于查找目标行更新的条件
WHEN ...                         --  更新
```

对目标表中行的更新使用 WHEN MATCHED…THEN…列出。多个 MATCHED 子句可以添加条件，以确定何时应用每个匹配。使用第一个匹配的表达式。

```
WHEN MATCHED AND s.op = 'delete' THEN DELETE
WHEN MATCHED AND t.count IS NULL AND s.op = 'increment' THEN UPDATE SET t.count = 0
WHEN MATCHED AND s.op = 'increment' THEN UPDATE SET t.count = t.count + 1
```

不匹配的 source 行(更新)可以插入：

```
WHEN NOT MATCHED THEN INSERT *
```

插入还支持其他条件：

```
WHEN NOT MATCHED AND s.event_time > still_valid_threshold THEN INSERT (id, count) VALUES (s.id, 1)
```

源(source)数据中只有一条记录可以更新目标表中的任何给定行，否则将抛出错误。

INSERT OVERWRITE

INSERT OVERWRITE 可以用查询的结果替换表中的数据。OVERWRITE 是 Iceberg 表的原子操作。

要被 INSERT OVERWRITE 替换的分区取决于 Spark 的分区覆盖模式和表的分区。MERGE INTO 能仅重写受影响的数据文件，并且具有更容易理解的行为，因此建议使用 MERGE INTO 而不是 INSERT OVERWRITE。

Spark 的默认覆盖模式是静态的（static），但是在写 Iceberg 表时建议使用动态覆盖模式。静态覆盖模式通过将 PARTITION 子句转换为过滤器来确定要覆盖表中的哪些分区，但是 PARTITION 子句只能引用表的列。

动态覆盖模式是通过设置 spark.sql.sources.partitionOverwriteMode=dynamic 来配置的。

为了演示动态和静态覆盖的行为，考虑一个由以下 DDL 定义的 logs 表：

```
CREATE TABLE prod.my_app.logs (
    uuid string NOT NULL,
    level string NOT NULL,
    ts timestamp NOT NULL,
    message string)
USING iceberg
PARTITIONED BY (level, hours(ts))
```

动态覆盖

当 Spark 的覆盖模式是动态的时，包含 SELECT 查询产生的行的分区将被替换。

例如，下面的查询从示例的 logs 表中删除重复的日志事件。

```
INSERT OVERWRITE prod.my_app.logs
SELECT uuid, first(level), first(ts), first(message)
FROM prod.my_app.logs
WHERE cast(ts as date) = '2020-07-01'
GROUP BY uuid
```

在 dynamic 模式下，这将用 SELECT 结果中的行替换任何分区。因为所有行的日期都限制在 7 月 1 日，所以只替换当天的 hours。

静态覆盖

当 Spark 的覆盖模式为静态时，PARTITION 子句被转换为用于从表中删除的过滤器。如果省略了 PARTITION 子句，所有的分区都将被替换。

由于在上面的查询中没有 PARTITION 子句，当以静态模式运行时，它将删除表中所有现有的行，但只写入 7 月 1 日的日志。

要覆盖已加载的分区，添加一个与 SELECT 查询过滤器对齐的 PARTITION 子句：

```
INSERT OVERWRITE prod.my_app.logs
PARTITION (level = 'INFO')
SELECT uuid, first(level), first(ts), first(message)
FROM prod.my_app.logs
WHERE level = 'INFO'
GROUP BY uuid
```

注意，这种模式不能像动态示例查询那样替换小时分区，因为 PARTITION 子句只能引用表中的列，而不能引用隐藏分区。

DELETE FROM

Spark 3 增加了对 DELETE FROM 查询的支持，可以从表中删除数据。

Delete 查询接受一个过滤器来匹配要删除的行。

```
DELETE FROM prod.db.table
WHERE ts >= '2020-05-01 00:00:00' and ts < '2020-06-01 00:00:00'
```

如果删除过滤器匹配表的整个分区，Iceberg 将执行一个只删除元数据的操作。如果过滤器匹配表中的各个行，那么 Iceberg 将只重写受影响的数据文件。

13.5.2 使用 DataFrame 写

Spark 3 引入了新的 DataFrameWriterV2 API，用于使用 DataFrame 写入表。推荐使用 v2 API 有以下几个原因：

- 支持 CTAS、RTAS 和过滤器覆盖；
- 所有操作都一致地按名称将列写入表；
- partitionedBy 支持隐藏分区表达式；
- 覆盖行为是显式的，可以是动态的，也可以是用户提供的过滤器；
- 每个操作的行为都对应于 SQL 语句：
  - df.writeTo(t).create() 等价于 CREATE TABLE AS SELECT
  - df.writeTo(t).replace() 等价于 REPLACE TABLE AS SELECT
  - df.writeTo(t).append() 等价于 INSERT INTO
  - df.writeTo(t).overwritePartitions() 等价于 dynamic INSERT OVERWRITE

仍然支持 v1 DataFrame write API，但不推荐使用。

注：在 Spark 3 中使用 v1 DataFrame API 写数据时，可以使用 saveAsTable 或 insertInto 来加载带有 catalog 目录的表。使用 format("iceberg")加载一个独立的表引用，该引用不会自动刷新查询使用的表。

追加数据

要将一个 DataFrame 添加到 Iceberg 表，使用 append：

```
val data: DataFrame = ...
data.writeTo("prod.db.table").append()
```

在 Spark 2.4 中，使用 append 模式和 iceberg 格式的 v1 API：

```

```

覆盖数据

要动态地覆盖分区，可以使用 overwritePartitions()：

```
val data: DataFrame = ...
data.writeTo("prod.db.table").overwritePartitions()
```

要显式覆盖分区，使用 overwrite 来提供一个过滤器：

```
data.writeTo("prod.db.table").overwrite($"level" === "INFO")
```

在 Spark 2.4 中，用 overwrite 模式和 iceberg 格式覆盖 Iceberg 表中的值：

```

```

创建表

要运行一个 CTAS 或 RTAS，使用 create、replace 或 createOrReplace 操作：

```

```

Create 和 replace 操作支持表配置方法，如 partitionedBy 和 tableProperty：

```

```

13.5.3 写分区表

Iceberg 通过实现隐藏分区，使用户能够简单地进行分区。与其强迫用户在查询时提供单独的分区过滤器，Iceberg 在底层处理分区和查询的所有细节。用户不需要维护分区列，甚至不需要了解物理表的布局，就可以得到准确的查询结果。

在每个 Iceberg 表文件夹中，都有一个元数据文件夹和一个数据文件夹。metadata 文件夹包含关于分区规范的信息、它们的惟一 id，以及将单个数据文件与适当的分区规范 id 连接起来的清单。data 文件夹包含构成整个 Iceberg 表的所有表数据文件。当向带有分区的表写入数据时，Iceberg 会在 data 文件夹中创建几个子文件夹。每个分区都使用分区描述和值进行命名。例如，标题为 time 并按 month 划分的列将有以下文件夹：time_month=2008-11、time_month=2008-12，等等。我们将在下面的例子中直接看到这一点。在多个列上分区的数据会创建多个文件夹，每个顶级文件夹包含一个子文件夹，用于存放每个二级分区值。

在对分区表进行写操作之前，Iceberg 要求根据每个任务(Spark 分区)的分区规范对数据进行排序。这既适用于 SQL 写入，也适用于 DataFrame 写入。

注：显式排序是必须的，因为从 Spark 3.0 开始，在写入之前，Spark 不允许 Iceberg 请求排序。

注：全局排序(orderBy/sort)和本地排序(sortWithinPartitions)都可以满足需求。

让我们根据下面的 sample 表来写数据：

要将数据写入 sample 表，数据需要按 days(ts)、category 排序。

如果用 SQL 语句插入数据，可以使用 ORDER BY 来实现，如下所示：

如果使用 DataFrame 插入数据，则可以使用 orderBy/sort 来触发全局排序，或使用 sortWithinPartitions 来触发局部排序。局部排序如下：

对于大多数分区转换，可以简单地将原始列添加到排序条件中，bucket 除外。对于 bucket 分区转换，需要在 Spark 中注册 Iceberg 转换函数，以便在排序过程中指定它。

让我们看看另一个有桶分区的示例表：

需要注册函数来处理 bucket，如下所示：

这里我们将 bucket 函数注册为 iceberg_bucket16，它可以在 sort 子句中使用。

如果用 SQL 语句插入数据，可以像下面这样使用该函数：

如果用 DataFrame 插入数据，可以像下面这样使用该函数：

13.5.4 类型适配

Spark 和 Iceberg 支持不同的类型集。Iceberg 会自动进行类型转换，但不是针对所有组合，因此在设计表中的列类型之前，可能希望了解 Iceberg 中的类型转换。

Spark 类型到 Iceberg 类型

此类型转换表描述了如何将 Spark 类型转换为 Iceberg 类型。这个转换既适用于创建 Iceberg 表，也适用于通过 Spark 写入 Iceberg 表。

Iceberg 类型到 Spark 类型

此类型转换表描述了如何将 Iceberg 类型转换为 Spark 类型。转换应用于通过 Spark 从 Iceberg 表读取数据。

13.6 使用存储过程维护表

要在 Spark 中使用 Iceberg，首先需要配置 Spark catalogs。只有在 Spark 3.x 中使用 Iceberg SQL 扩展时，存储过程才可用。

13.6.1 用法

可以通过 CALL 从任何配置的 Iceberg catalog 中使用过程。所有过程都位于命名空间 system 中。
CALL 支持通过名称(推荐)或位置传递参数。不支持混合位置参数和命名参数。

命名参数
所有过程参数都被命名。当通过名称传递参数时，参数可以按任意顺序传递，任何可选参数都可以省略。

```
CALL catalog_name.system.procedure_name(arg_name_2 => arg_2, arg_name_1 => arg_1)
```

位置参数
当按位置传递参数时，如果结尾参数是可选的，则只能省略它们。

```
CALL catalog_name.system.procedure_name(arg_1, arg_2, ... arg_n)
```

13.6.2 快照管理

rollback_to_snapshot
将表回滚到指定的快照 ID。
要回滚到特定的时间，使用 rollback_to_timestamp。
此过程将使所有引用受影响表的缓存 Spark 计划失效。
例如，将表 db.sample 回滚到快照 ID 1：

```
CALL catalog_name.system.rollback_to_snapshot('db.sample', 1)
```

rollback_to_timestamp
将表回滚到某个时间点的当前快照。
此过程将使所有引用受影响表的缓存 Spark 计划失效。
例如，将 db.sample 回滚到一天前：

```
CALL catalog_name.system.rollback_to_timestamp('db.sample', date_sub(current_date(), 1))
```

set_current_snapshot
设置表的当前快照 ID。
与 rollback 不同，快照不需要是当前表状态的祖先。
此过程将使所有引用受影响表的缓存 Spark 计划失效。
例如，为 db.sample 设置当前快照为 1：

```
CALL catalog_name.system.set_current_snapshot('db.sample', 1)
```

cherrypick_snapshot
cherry-pick 从快照更改到当前表状态。
在不修改或删除原始快照的情况下，cherry-picking 从现有快照创建一个新的快照。
只有 append 和动态覆盖快照可以被 cherry-picked。
此过程将使所有引用受影响表的缓存 Spark 计划失效。

例如，cherry-pick 快照 1：

```
CALL catalog_name.system.cherrypick_snapshot('my_table', 1)
```

使用命名参数 cherry-pick 快照 1：

```
CALL catalog_name.system.cherrypick_snapshot(snapshot_id => 1, table => 'my_table' )
```

13.6.3 元数据管理

可以使用 Iceberg 存储过程执行许多维护操作。

expire_snapshots

在 Iceberg 中，每次写/更新/删除/upsert/压缩都会生成一个新快照，同时保留旧数据和元数据，以便快照隔离和时间旅行。expire_snapshots 过程可以用来删除不再需要的旧快照及其文件。

此过程将删除仅旧快照所需要的旧快照和数据文件。这意味着 expire_snapshots 将不会删除未过期快照所需要的文件。

例如，删除超过 10 天的快照，但保留最近的 100 个快照：

```
CALL hive_prod.system.expire_snapshots('db.sample', date_sub(current_date(), 10), 100)
```

删除所有比当前时间戳老的快照，但保留最后 5 个快照：

```
CALL hive_prod.system.expire_snapshots(table => 'db.sample', older_than => now(), retain_last => 5)
```

remove_orphan_files

用于删除 Iceberg 表的任何元数据文件中没有引用的文件，因此可以认为是“孤立的”文件。

例如，通过在这个表上执行 remove_orphan_files 命令，列出所有可能需要删除的文件，而不是实际删除它们：

```
CALL catalog_name.system.remove_orphan_files(table => 'db.sample', dry_run => true)
```

删除表 db.sample 所不知道的 tablelocation/data 文件夹中的任何文件。

```
CALL catalog_name.system.remove_orphan_files(table => 'db.sample', location => 'tablelocation/data')
```

rewrite_manifests

重写表清单(manifest)以优化扫描计划。

manifest 中的数据文件是按照分区规范中的字段进行排序的。此过程使用 Spark 作业并行运行。

此过程将使所有引用受影响表的缓存 Spark 计划失效。

例如，重写表 db.sample 中的清单文件并将其与表分区对齐。

```
CALL catalog_name.system.rewrite_manifests('db.sample')
```

重写表 db.sample 中的清单，并禁用 Spark 缓存的使用。这样做可以避免执行器的内存问题。

```
CALL catalog_name.system.rewrite_manifests('db.sample', false)
```

13.6.4 表迁移

snapshot 和 migrate 过程有助于测试和迁移现有的 Hive 或 Spark 表到 Iceberg。

snapshot

在不更改源表的情况下，创建用于测试的表的轻量级临时副本。

可以对新创建的表进行更改或写入，而不会影响源表，但是快照使用原始表的数据文件。当在快照上运行插入或覆盖时，新文件被放置在快照表的位置，而不是原始表的位置。

测试完快照表后，运行 DROP TABLE 清理快照表。

例如，在 catalog 的 default 位置为 db.snap 创建一个孤立的 Iceberg 表，它引用表 db.sample，命名为 db.snap。

```
CALL catalog_name.system.snapshot('db.sample', 'db.snap')
```

迁移一个引用表 db.sample 的孤立的 Iceberg 表 db.snap，在一个手动指定的位置/tmp/temptable/。

```
CALL catalog_name.system.snapshot('db.sample', 'db.snap', '/tmp/temptable/')
```

migrate

用装载源数据文件的 Iceberg 表替换表。表模式、分区、属性和位置将从源表复制。

如果任何表分区使用不支持的格式，迁移将失败。支持的格式有 Avro、Parquet 和 ORC。现有数据文件被添加到 Iceberg 表的元数据中，可以使用从原始表模式创建的 name-to-id 映射来读取。

为了在测试时保持原始表不变，可以使用 snapshot 创建共享源数据文件和模式的新临时表。

例如，将 Spark 的 default catalog 中的 db.sample 表迁移到一个 Iceberg 表，并添加一个属性 foo，设置为 bar。

```
CALL catalog_name.system.migrate('spark_catalog.db.sample', map('foo', 'bar'))
```

Migrate db.sample in the current catalog to an Iceberg table without adding any additional properties:

将当前 catalog 中的 db.sample 迁移到一个 Iceberg 表，而不添加任何其他属性。

```
CALL catalog_name.system.migrate('db.sample')
```

13.7 Spark 结构化流

Iceberg 使用 Apache Spark 的 DataSourceV2 API 实现数据源和目录实现。Spark DSv2 是一个不断发展的 API，在 Spark 版本中提供了不同级别的支持。

从 Spark 3.0 开始，支持 DataFrame 读写。

13.7.1 流写入

要将流查询的值写入 Iceberg 表，使用 DataStreamWriter：

```

```

其中 tableIdentifier 可以是：

- HDFS 表的完全限定路径，例如 hdfs://nn:8020/path/to/table。
- 如果表由 catalog 跟踪，则为表名，如 database.table_name。

Iceberg 不支持“continuous processing”，因为它不提供“commit”输出的接口。

Iceberg 支持 append 和 complete 的输出模式：

- append：将每个微批的行追加到表中。
- complete：每个微批替换表内容。

表应该在开始流查询之前创建。

对分区表进行写入

在对分区表进行写操作之前，Iceberg 要求根据每个任务(Spark 分区)的分区规范对数据进行排序。对于批处理查询，鼓励执行显式排序来满足需求，但是这种方法会带来额外的延迟，因为重分区和排序被认为是流工作负载的繁重操作。为了避免额外的延迟，可以启用 fanout writer 来消除这种需求。

Fanout writer 按每个分区值打开文件，直到写任务完成才关闭这些文件。不鼓励批处理查询使用此功能，因为对输出行进行显式排序对于批处理工作负载来说并不昂贵。

13.7.2 维护流表

流查询可以快速创建新的表版本，这将创建大量的表元数据来跟踪这些版本。强烈建议通过调优提交速率、旧快照过期和自动清理元数据文件来维护元数据。

调优提交速率

高提交率会产生大量的数据文件、清单和快照，这会导致表难以维护。我们鼓励至少 1 分钟的触发间隔，并在需要时增加间隔。

过期旧快照

每个写入表的微批都会生成一个新的快照，在表元数据中跟踪该快照直到过期，从而删除不再需要的元数据和任何数据文件。快照随着频繁提交而迅速累积，因此强烈建议定期维护由流查询写的表。

压缩数据文件

微批处理中写入的数据量通常很小，这可能导致表元数据跟踪大量小文件。将小文件压缩为大文件可以减少表所需的元数据，并提高查询效率。

重写清单

为了优化流工作负载上的写延迟，Iceberg 可以使用不自动压缩清单的“fast”追加来写新快照。这可能导致大量的小清单文件。清单可以被重写以优化查询和压缩。

13.8 使用 Spark 构建 Iceberg 数据湖

现在进入我们的示例，该示例使用 Iceberg 的 HiveCatalog API 从 Hive metastore 创建和加载 Iceberg 表。为了简洁起见，我们使用了一个简单的数据集，它模拟了 X 公司开发的某些软件产品的日志表，其中列有 time、log_id 和 log_message。注意，数据中的时间戳显示为长数据类型，对应于它们的 UNIX 时间戳，以秒为单位：

在本例中，该表创建于 2008 年。从那时起，X 公司获得了几个客户，现在希望日志事件更频繁地发生。他们认为，从 2009 年开始，按日志事件的天数划分会更有用。我们将通过手动添加必要的数据来完成这个场景。

首先，用以下命令启动 spark-shell。在本例中，我们使用了 Spark 3.1.1、Hive metastore 2.3.4 和 Iceberg 包 0.11.0。

```
$SPARK_HOME/bin/spark-shell --packages org.apache.iceberg:iceberg-spark3-runtime:0.11.0 \
    --conf spark.sql.catalog.my_catalog=org.apache.iceberg.spark.SparkCatalog \
    --conf spark.sql.catalog.my_catalog.type=hive
```

注：Iceberg 目前仅通过其 HiveCatalog 接口支持分区演进，而 HadoopTable 接口的用户无法访问相同的功能。

启动 Spark shell 后，导入本例所需的 Iceberg 包：

现在，我们在名为 default 的名称空间中创建表(logtable)，这个名称空间最初是按事件月份划分的。为此，我们创建了一个 HiveCatalog 对象和一个 TableIdentifier 对象。我们还必须定义表模式和初始分区规范，以向 createTable 函数提供这些信息：

然后将数据添加到表中。如果使用这里提供的数据集，请将 file_location 变量设置为计算机上数据 CSV 文件的路径。在以下命令中，我们仅添加 2009 年 1 月 1 日之前的数据，模拟公司 X 示例中设置的场景。如果使用自己的数据集，请确保在写入表时对分区列上的数据进行排序(如下所示)：

查看结果

接下来，我们定义一个新的 PartitionSpec 对象，指定构建它的模式(前面定义的)以及所需的分区和源列：

```
val newspec = PartitionSpec.builderFor(logtable.schema()).day("time").build()
```

然后，我们通过定义一个 TableOperations 对象来更新表的分区规范。从这个对象中，我们可以定义基本元数据版本和根据演进分区规范的新元数据版本。必须在 TableOperations 对象上执行提交操作，以正式实现所需的更改:

在我们的 X 公司场景中，他们已经开发了表的分区规范，并且添加了新的日志。我们通过使用以下代码手动向表中添加新日志来模拟这一点。在这个写操作中，我们只添加在 2009 年 1 月 1 日或之后发生的数据：

正如我们看到的，在 Iceberg 中进行分区演进后，不需要重写整个表。如果导航到 logtable 的 data 文件夹，将看到 Iceberg 已经根据分区值对数据文件进行了组织—2009 年 1 月 1 日之前的时间戳是按 month 组织的；该日期和之后的时间戳按 day 进行组织。

新表 spec 和数据如下所示。

```
logtable.spec
```

同样显示表格。

```
spark.table("hive_catalog.default.logtable").show
```

输出结果如下（旧的数据）：

如果使用的是 Spark 2.4.x，使用下面的语句来显示表格。

```
spark.read.format("iceberg").load("default.logtable").show
```

输出结果如下（新的数据）：

X 公司现在想查询员工休假期间发生的所有日志事件，以确保他们不会错过任何重大错误。查询如下所示，跨越多个分区布局，但仍然无缝工作，用户不需要指定任何额外的信息或知道任何关于表的分区：

```
spark.table("default.logtable").createOrReplaceTempView("logtable")
spark.sql("SELECT * FROM logtable WHERE time > '2008-12-14' AND time < '2009-1-14'").show
```

第 14 章 Hudi 数据湖

Apache Hudi 摄取并管理通过 DFS (hdfs 或云存储)存储的大型分析数据集。Hudi 为大数据带来了流处理，在提供新数据的同时，比传统的批处理效率高出一个数量级。

Hudi 特性：

- 支持快速、可插入索引的 upsert。
- 支持回滚的原子发布数据。
- 写入器和查询之间的快照隔离。
- 保存点用于数据恢复。
- 使用统计管理文件大小和布局。
- 异步压缩行和列数据。
- 跟踪沿袭（lineage）的时间线元数据。
- 利用聚类优化数据湖布局。

14.1 Hudi 特性

Apache Hudi(发音为“hoodie”)在 hadoop 兼容存储上提供流原语：

- 更新/删除记录(如何更改表中的记录?)
- 更改流(如何获取更改的记录?)

正我们将讨论一些关键概念和术语，这些概念和术语对于理解和有效使用这些原语非常重要。

14.1.1 Timeline

其核心是，Hudi 维护了在不同时刻(instant)在表上执行的所有操作的时间轴(timeline)，这有助于提供表的瞬时视图，同时还有效地支持按到达顺序检索数据。一个 Hudi 瞬态(instant)由以下组件组成

- Instant action：对表执行的 action 类型。
- Instant time：瞬间时间通常是一个时间戳(例如:20190117010349)，它按照 action 的开始时间顺序单调地增加。
- state：瞬间的当前状态。

Hudi 保证在时间轴上执行的 action 是原子的 & 时间轴基于 instant time 一致。

执行的关键 action 包括：

- COMMITS - commit 表示将一批记录原子地写到一个表中。
- CLEANS - 清除表中不再需要的旧版本文件的后台活动。
- DELTA_COMMIT - 增量提交指的是将一批记录原子地写入 MergeOnRead 类型表，其中部分/所有数据都可以写入增量日志。
- COMPACTION - 协调 Hudi 内部不同数据结构的后台活动，例如：将更新从基于行的日志文件

移动到柱状格式。在内部，压缩显示为时间轴上的一个特殊提交。

- ROLLBACK - 指示提交/增量提交未成功&回滚，删除了这种写过程中产生的任何部分文件。
- SAVEPOINT - 将某些文件组标记为“已保存”，这样 cleaner 就不会删除它们。在灾难/数据恢复场景下，它有助于将表恢复到时间轴上的某个点。

任何给定的瞬间（instant）都可能处于下列状态之一：

- REQUESTED - 表示某个 action 已被计划，但尚未启动。
- INFLIGHT - 表示当前正在执行的 action。
- COMPLETED - 表示完成时间轴上的 action。

上面的例子显示了在 10:00 到 10:20 之间发生在 Hudi 表上的 upserts 事件，大约每 5 分钟发生一次，commit 元数据和其他后台清理/压缩一起留在了 Hudi 时间轴上。要做的一个关键观察是，提交时间表示数据的到达时间(10:20AM)，而实际数据组织反映的是实际时间或事件时间，这是数据的目的(从 07:00 开始的每小时桶)。这是在权衡数据的延迟和完整性时的两个关键概念。

当出现延迟到达的数据时(预计 9 点到达>的数据在 1 小时后 10 点 20 分到达)，我们可以看到 upsert 将新数据生成到更老的时间桶/文件夹中。在时间轴的帮助下，尝试获取 10:00 小时以来成功提交的所有新数据的增量查询能够非常有效地只使用更改的文件，而不必扫描所有> 07:00 的时间桶。

14.1.2 File Layout

Hudi 将表组织成 DFS 上的一个 basepath 下的目录结构。表被分成几个分区，这些分区是包含该分区的数据文件的文件夹，非常类似于 Hive 表。每个分区都由它的 partitionpath(它是相对于 basepath 的)唯一标识。

在每个分区中，文件被组织成由 file id 唯一标识的文件组(file groups)。每个文件组包含几个文件片(file slices)，其中每个片包含一个在特定提交/压缩瞬间产生的基本文件(*.parquet)，以及一组日志文件(*.log.*)，这些日志文件包含自生成基本文件以来对基本文件的插入/更新。

14.1.3 Index

Hudi 通过索引机制将给定的 hoodie key(记录键+分区路径)一致地映射到文件 id，从而提供了高效的 upserts。记录键和文件组/文件 id 之间的映射，一旦记录的第一个版本被写入到一个文件中，就不会改变。简而言之，映射文件组包含一组记录的所有版本。

14.1.4 Table Types & Queries

Hudi 表类型定义了数据是如何在 DFS 上被索引和布局的，以及上面的原语和时间轴活动是如何在这样的组织上实现的(即数据是如何写入的)。反过来，查询类型定义了如何将底层数据公开给查询(即如何读取数据)。

表类型	支持的查询类型
Copy On Write	快照查询 + 增量查询
Merge On Read	快照查询+增量查询+读优化查询

表类型

Hudi 支持如下的表类型：

- Copy On Write：只使用柱状文件格式存储数据(如 parquet)。更新简单地版本化并通过写期间执行一个同步合并来重写文件。
- Merge On Read：使用柱状(如 parquet)+基于行的(如 avro)文件格式的组合来存储数据。更新被记录到增量文件中，然后压缩以同步或异步地生成新版本的柱状文件。

下表总结了这两种表类型之间的权衡。

	Copy On Write	Merge On Read
数据延迟	较高	较低
查询延迟	较低	较高
更新成本(I / O)	较高（重写整个parquet）	较低（追加到增量日志文件）
parquet文件大小	较小（高更新(I/O)成本）	较高（低更新(I/O)成本）
写入放大率	较高	较低（取决于压缩策略）

查询类型

Hudi 支持如下的查询类型：

- 快照查询：查询查看给定提交或压缩操作时表的最新快照。在对 merge on read 表进行合并的情况下，它通过动态合并最新文件片的基本文件和增量文件来公开近实时的数据(几分钟)。对于 copy on write 表，它提供了对现有 parquet 表的就地替代，同时提供了 upsert/delete 和其他写端特性。
- 增量查询：查询只看写入表的新数据，从一个给定的提交/压缩。这有效地提供了更改流以支持增量数据管道。
- 读优化查询：查询查看给定提交/压缩操作时表的最新快照。仅在最新的文件切片中公开基/柱状文件，并保证与非 hudi 柱状表相比具有相同的柱状查询性能。

下表总结了不同查询类型之间的权衡。

	快照	读优化
数据延迟	较低	较高
查询延迟	较高（合并基础/柱状文件+基于行的增量/日志文件）	较低（原始基础/柱状文件性能）

14.1.5 Copy On Write 表

Copy-On-Write 表中的文件片只包含基文件/柱状文件，并且每次提交都会产生基文件的新版本。换句话说，我们隐式地压缩了每次提交，这样就只存在列数据。因此，写入放大(1 字节输入数据需要写入的字节数)要高得多，而读放大为零。**这是分析工作负载非常需要的属性**，分析工作负载主要是大量读取。

下面演示了当数据写入 copy-on-write 表并在其上运行两个查询时，这是如何从概念上工作的。

在写入数据时，对现有文件组的更新会为该文件组生成一个新片，并标记提交瞬时时间（instant time），而插入则会分配一个新文件组，并为该文件组写入其第一个片。这些文件切片和它们的提交瞬时时间在上面用颜色编码。对这样一个表运行的 SQL 查询(例如：select count(*)计算分区中的总记录)，首先检查最近提交的时间轴，并过滤每个文件组的所有文件片(除了最新的)。如图中所见，旧的查询没

有看到以粉红色编码的当前正提交的文件，而是在提交后开始的新查询获得新数据。因此查询不受任何写失败或部分写的影响，只在已提交的数据上运行。

对写表进行复制的目的是从根本上改善表的管理方式：

- 第一级支持在文件级别上自动更新数据，而不是重写整个表/分区。
- 能够增量地使用更改，而不是浪费的扫描或摸索启发式。
- 严格控制文件大小以保持出色的查询性能(小文件会严重影响查询性能)。

14.1.6 Merge On Read 表

Merge-On-Read 表是 Copy-On-Write 表的一个超集，从某种意义上说，它仍然通过在最新的文件片中只公开基/柱状文件来支持对表的读优化查询。此外，它将每个文件组的传入 upserts 存储到一个基于行的增量日志中，以便在查询期间实时应用增量日志到每个文件 id 的最新版本，从而支持快照查询。因此，这种表类型尝试智能地平衡读和写放大，以提供近实时的数据。这里最重要的变化是对压缩器的更改，它现在会仔细选择哪些增量日志文件需要压缩到它们的柱状基本文件中，以检查查询性能(较大的增量日志文件将导致查询端合并数据的合并时间更长)。

下面演示了表的工作方式，并显示了两种查询类型—快照查询和读优化查询。

在这个示例中发生了很多有趣的事情，这些事情带来了方法中的微妙之处。

- 我们现在每 1 分钟左右提交一次，这是我们在其他表类型中无法做到的。
- 在每个文件 id 组中，现在有一个增量日志文件，它保存对基本柱状文件中记录的传入更新。在这个示例中，增量日志文件保存了 10:05 到 10:10 之间的所有数据。与前面一样，基本柱状文件仍然使用提交进行版本控制。因此，如果只看基本文件，那么表布局看起来就像 copy-on-write 表。
- 定期压缩过程协调来自增量日志的这些更改，并生成一个新版本的基文件，就像示例中的 10:05 所发生的那样。
- 查询同一底层表有两种方式：Read Optimized 查询和 Snapshot 查询，这取决于我们选择的是查询性能还是数据的新鲜度。
- 关于何时提交的数据可用于查询的语义以一种微妙的方式对读优化查询进行更改。注意，这样的查询在 10:10 运行，不会看到 10:05 之后的数据，而快照查询总是看到最新的数据。
- 当我们触发压缩&它决定压缩什么持有解决这些困难问题的所有关键。通过实现压缩策略，我们积极地压缩最新的分区，而不是旧的分区，我们可以确保经过读优化的查询在 X 分钟内以一致的方式查看发布的数据。

对 merge-on-read 表进行合并的目的是在 DFS 之上直接实现近乎实时的处理，而不是将数据复制到特定的系统，后者可能无法处理数据量。这个表还有一些次要的好处，比如通过避免数据的同步合并来减少写入放大，也就是说，在批处理中，每 1 字节的数据写入的数据量。

14.2 在 Spark 3 中使用 Hudi

使用 Spark 数据源，我们可以插入和更新具有默认表类型（Copy on Write）的 Hudi 表。在每次写操作之后，我们还将展示如何同时读取快照和增量读取数据。

Hudi 当前最新版本是 0.8.0，使用 Spark-2.4.3+ 以及 Spark 3.x 版本。

14.2.1 配置 Hudi

运行 spark-shell 与 Hudi 如下：

```
// spark-shell for spark 3
```

注意：

- spark-avro 模块需要在--packages 中指定，因为 spark-shell 默认不包含它。
- spark-avro 和 spark 版本必须匹配（上面我们都是用 3.1.2 或 2.4.7）。
- 如果使用 spark-avro_2.12，则需要使用 hudi-spark-bundle_2.12/hudi-spark3-bundle_2.12。如果使用 spark-avro_2.11，则需要使用 hudi-spark-bundle_2.11/hudi-spark3-bundle_2.11。

方式二，将以下两个 jar 包拷贝到 Spark 的 jars 目录下。

-

方式三，如果是使用 Maven 开发，则添加如下依赖：

```
// for spark 3
```

如果是使用 Zeppelin，需要：

- 把 spark-avro_2.12-3.1.2.jar 拷贝到$ZEPPELIN_HOME/lib/目录下；
- 在运行所有的 Spark 代码之前，先执行如下配置：

```
%spark.conf
spark.jars.packages org.apache.hudi:hudi-spark3-bundle_2.12:0.8.0,org.apache.spark:spark-avro_2.12:3.1.2
spark.serializer       org.apache.spark.serializer.KryoSerializer
```

注：如果不生效，请配置 Zeppelin spark 解释器的 spark.jars.packages 属性。

14.2.2 初始设置

设置表名、基本路径和数据生成器，以生成示例所需要的记录。

```

```

DataGenerator 可以根据这里的示例旅行模式生成示例插入和更新：

```

```

14.2.3 插入数据

生成一些新的行程数据，将它们加载到 DataFrame 中，并将 DataFrame 写入 Hudi 表中，如下所示。

```
// 将数据加载到 DataFrame

```

这里使用分区字段（partition field (region/country/city)）。

14.2.4 查询数据

将数据文件加载到一个 DataFrame 中。

在上面的代码中，load(basePath)使用“/partitionKey=partitionValue”文件夹结构用于 Spark 自动分区发现。因为我们的分区路径(region/country/city)从基本路径嵌套了 3 个级别，所以我们使用 load(basePath + "/*/*/*/*")。

下面的查询提供对摄入数据的快照查询。

14.2.5 更新数据

这类似于插入新数据。使用数据生成器生成对现有旅程的更新，加载到 DataFrame 中，并将 DataFrame 写入 hudi 表。

注意，保存模式现在是 Append。一般来说，除非第一次创建表，否则总是使用 append 模式。再次查询数据将显示更新的行程。每个写操作都会生成一个由时间戳表示的新提交。

14.2.6 增量查询

Hudi 还提供了获取自给定提交时间戳以来更改的记录流的功能。这可以通过使用 Hudi 的增量查询来实现，并提供一个开始时间，更改需要从该时间流开始。如果想要在提交之后进行的所有更改(这是常见的情况)，则不需要指定 endTime。

```
// spark-shell
```

这将使用 fare > 20.0 的过滤器给出在 beginTime 提交之后发生的所有更改。查询结果如下：

这个特性的独特之处在于，它现在允许我们在批处理数据上编写流管道。

14.2.7 时间点查询

看看如何查询特定时间的数据。具体的时间可以通过将 endTime 指向特定的提交时间，将 beginTime 指向“000”(表示尽可能早的提交时间)来表示。

```
// spark-shell
```

可以看到，查询出来的是追加操作前的数据。

14.2.8 删除数据

删除传入的 HoodieKeys 记录。

```
// spark-shell
```

注：删除操作只支持 Append 模式。

14.2.9 插入覆盖表

生成一些新的行程数据，在 Hudi 元数据级别上逻辑地覆盖表。Hudi cleaner 将最终清理前一个表快照的文件组。这比删除旧表并在 Overwrite 模式下重新创建要快。

```
// spark-shell
```

14.2.10 插入覆盖

生成一些新的行程数据，覆盖输入中出现的所有分区。对于批处理 ETL 作业，此操作可以比 upsert 更快，后者是一次性重新计算整个目标分区(而不是增量地更新目标表)。这是因为，我们能够完全绕过 upsert 写路径中的索引、预组合和其他重新分区步骤。

```
// spark-shell
```