小白学苑
http://www.xueai8.com

面向零基础小白的大数据入门教程

PySpark 实用教程

基于 Spark 3.1.2 和 Python3.7

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。

Apache Spark 支持 Java、Scala、Python 和 R 语言，并提供了相应的 API。而在数据科学领域，Python 是应用最广的数据处理语言。但是作为大数据的初学者，在学习 PySpark 时通常会遇到以下几个难题：

- 缺少面向零基础小白的 PySpark 入门教程。
- 缺少系统化的 PySpark 大数据教程。
- 现有的 PySpark 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 PySpark 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 PySpark（以及其他大数据技术）的入门难度，编写了这一本《PySpark 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、综合项目案例等。
- 版本先进，所有代码均基于 Spark 3.1.2 和 Python 3.7。

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、希望拥有大数据系统参考教材的教师以及想要了解最新 Spark/PySpark 技术应用的从业人员。

当然，因为水平所限，行文以及内容难免有错误之处，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。

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 Python 代码，以及数据集和教学视频。最好的学习方法就是动手实践。

❑ 本书示例依赖小白学苑(http://www.xueai8.com)提供的个人大数据学习平台 PBLP。

对于大数据的初学者，最大的拦路虎是缺少一个成熟稳定的大数据平台环境。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）是小白学苑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、Python 3.7），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，请大家访问小白学苑(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。或者访问小白学苑首页，咨询网站管理员。

❑ 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，请访问小白学苑(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，那么可以联系(http://www.xueai8.com)小白学苑管理，获取本书正式版及配套代码的下载地址。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 和 PySpark 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。

1.3.4 PySpark 架构

PySpark 构建在 Spark 的 Java API 之上。数据在 Python 中处理，在 JVM 中缓存和 Shuffle。PySpark 架构如下图所示：

在 Python 驱动程序中，当 PySpark 的 Python 解释器启动时，SparkContext 使用 Py4J 启动 JVM 并创建 JavaSparkContext，并通过 Socket 套接字与之通信。JVM 作为实际的 Spark 驱动程序运行，并通过 JavaSparkContext 与集群中的 Spark Executor 进行通信。Py4J 只在驱动程序上用于 Python 和 JavaSparkContext 对象之间的本地通信（大型数据传输是通过一种不同的机制执行的）。

对 SparkContext 对象的 Python API 调用被转换为对 JavaSparkContext 的 Java API 调用。例如，PySpark 的 sc.textFile()的实现分派为对 JavaSparkContext 的. textFile()方法的调用，该方法最终与 Spark Executor JVM 通信，以从 HDFS 加载文本数据。

集群上的 Spark Executor 为每个 core 启动一个 Python 解释器，当它们需要执行用户代码时，通过管道与该解释器通信，发送用户代码和要处理的数据。

本地 PySpark 客户端中的 PythonRDD 对应于本地 JVM 中的 PythonRDD 对象。与此 RDD 关联的数据实际上作为 Java 对象存在于 Spark JVM 中。例如，在 Python 解释器中运行 sc.textFile()将调用 JavaSparkContext 的 textFile()方法，该方法将数据作为 Java 字符串对象加载到集群中。类似地，使用 newAPIHadoopFile 加载 Parquet/Avro 文件将以 Java Avro 对象的形式加载对象。

PySpark 目前使用 Python cPickle 序列化器序列化数据。PySpark 使用 cPickle 序列化数据，因为它相当快，并且支持几乎任何 Python 数据结构。当在 Python RDD 上进行 API 调用时，任何相关的代码(例如 Python lambda 函数)都会通过“cloudpickle(一个由 PiCloud 构建的定制模块)”进行序列化，并分发给执行器。然后将数据从 Java 对象转换为与 Python 兼容的表示(例如 pickle 对象)，并通过管道流向与 executor 相关的 Python 解释器。

任何必要的 Python 处理都在解释器中执行，结果数据作为 RDD（默认情况下作为 pickle 对象）存储回 JVM 中。

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

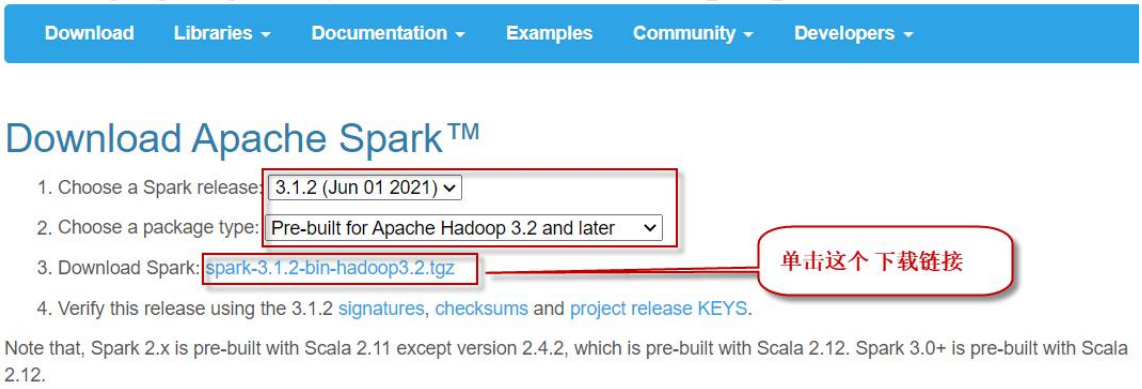
为了学习 Spark，最好在我们自己的计算机上本地安装 Spark。通过这种方式，我们可以轻松地尝试 Spark 特性或使用小型数据集测试数据处理逻辑。

Apache 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
```

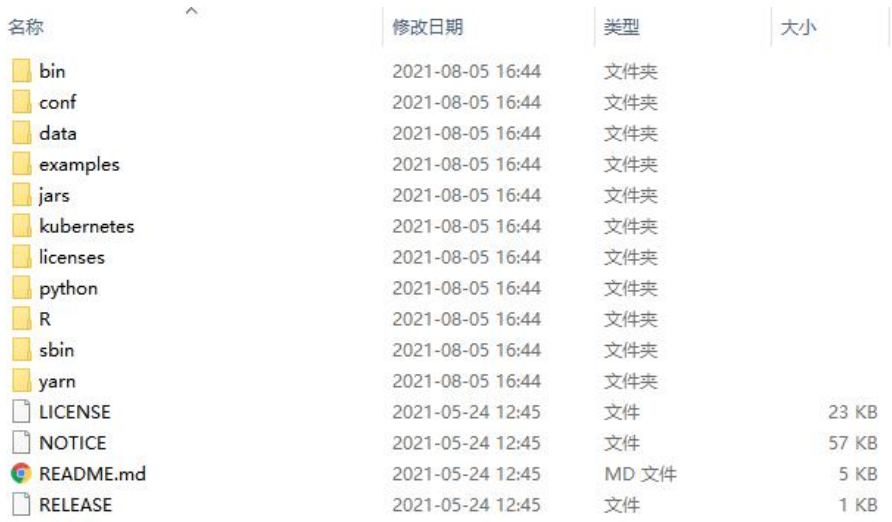
保存文件并关闭。

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：



名称	修改日期	类型	大小
bin	2021-08-05 16:44	文件夹	
conf	2021-08-05 16:44	文件夹	
data	2021-08-05 16:44	文件夹	
examples	2021-08-05 16:44	文件夹	
jars	2021-08-05 16:44	文件夹	
kubernetes	2021-08-05 16:44	文件夹	
licenses	2021-08-05 16:44	文件夹	
python	2021-08-05 16:44	文件夹	
R	2021-08-05 16:44	文件夹	
sbin	2021-08-05 16:44	文件夹	
yarn	2021-08-05 16:44	文件夹	
LICENSE	2021-05-24 12:45	文件	23 KB
NOTICE	2021-05-24 12:45	文件	57 KB
README.md	2021-05-24 12:45	MD 文件	5 KB
RELEASE	2021-05-24 12:45	文件	1 KB

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/PySpark 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：

1.5.4 验证 PySpark 安装

PySpark 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 PySpark 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。

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

在本地（Local）模式下，进入到 Spark 主目录下，直接使用 spark-submit 命令来提交示例程序 pi.py

运行即可。命令如下：

执行过程如下所示：

执行结果如下图中所示：

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

说明：

- --master 参数指定要连接的集群管理器，这里是 standalone 模式
- 最后一个参数是所提交的 python 程序

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。

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）。

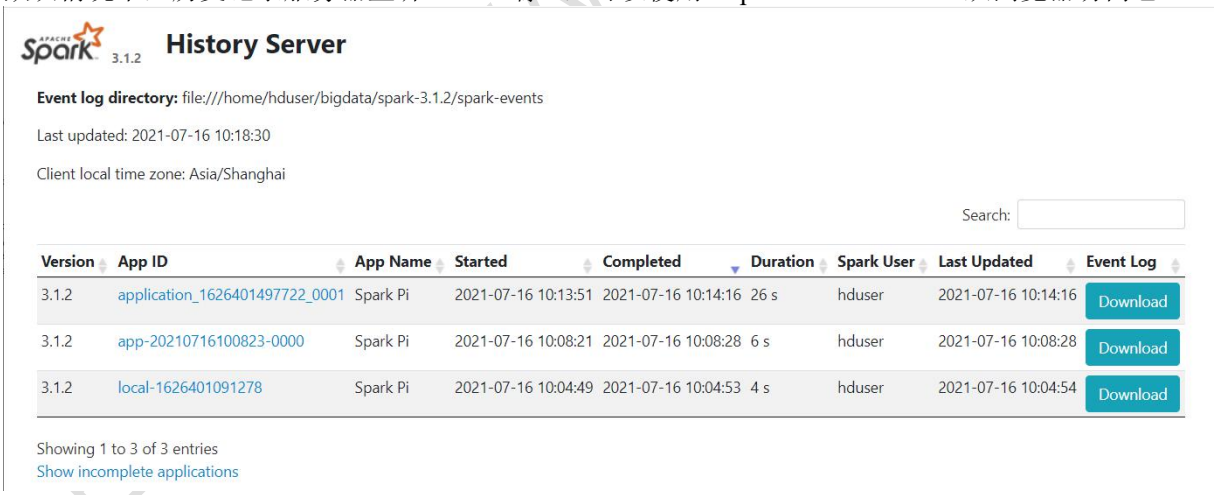
```
$ SPARK_HOME/bin/spark-class org.apache.spark.deploy.history.HistoryServer
```

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 pyspark 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，PySpark 提供了一个交互式的工具 pyspark shell。通过 Spark Shell，用户可以和 PySpark 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。

pyspark shell 命令格式如下所示：

```
$ ./bin/pyspark [options]
```

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

```
$ pyspark --help
```

1.7.1 运行模式--master

Spark/PySpark 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 pyspark shell 要连接到哪个 master（即告诉 Spark/PySpark 使用哪种集群类型）。

如果是 local[*]，就是使用本地模式启动 pyspark 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 pyspark shell 命令时，像下面这样定义这个参数：

```
$ pyspark --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：

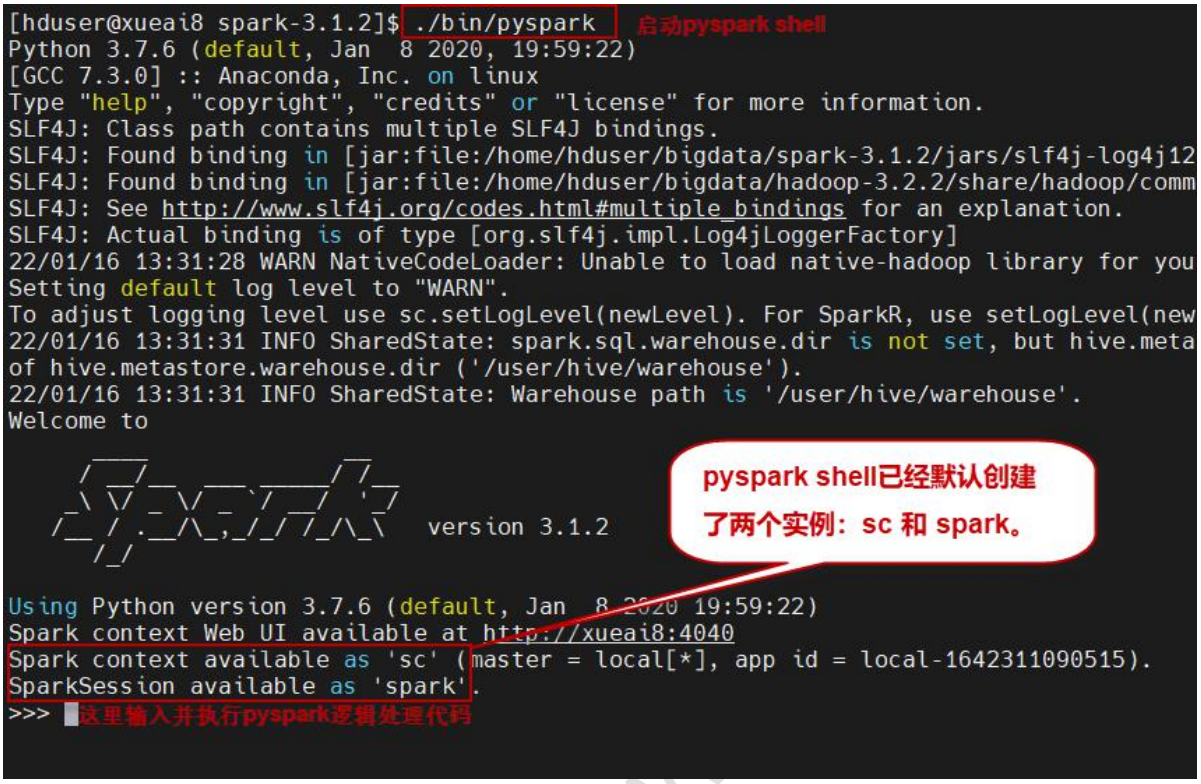
1.7.2 启动和退出 pyspark shell

以下操作均在终端窗口中进行。

1）启动 pyspark shell 方式一：local 模式

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

然后可以看到如下的启动过程：



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

退出 pyspark shell，使用如下命令:

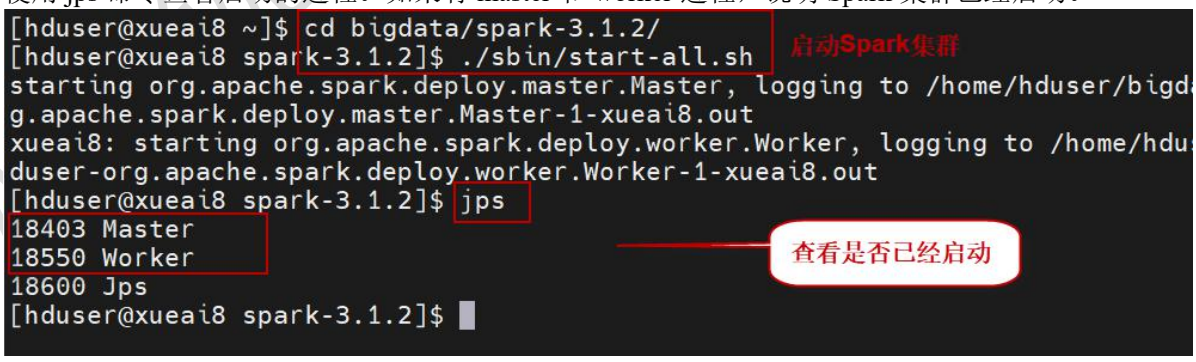
```
>>> exit()
```

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
```

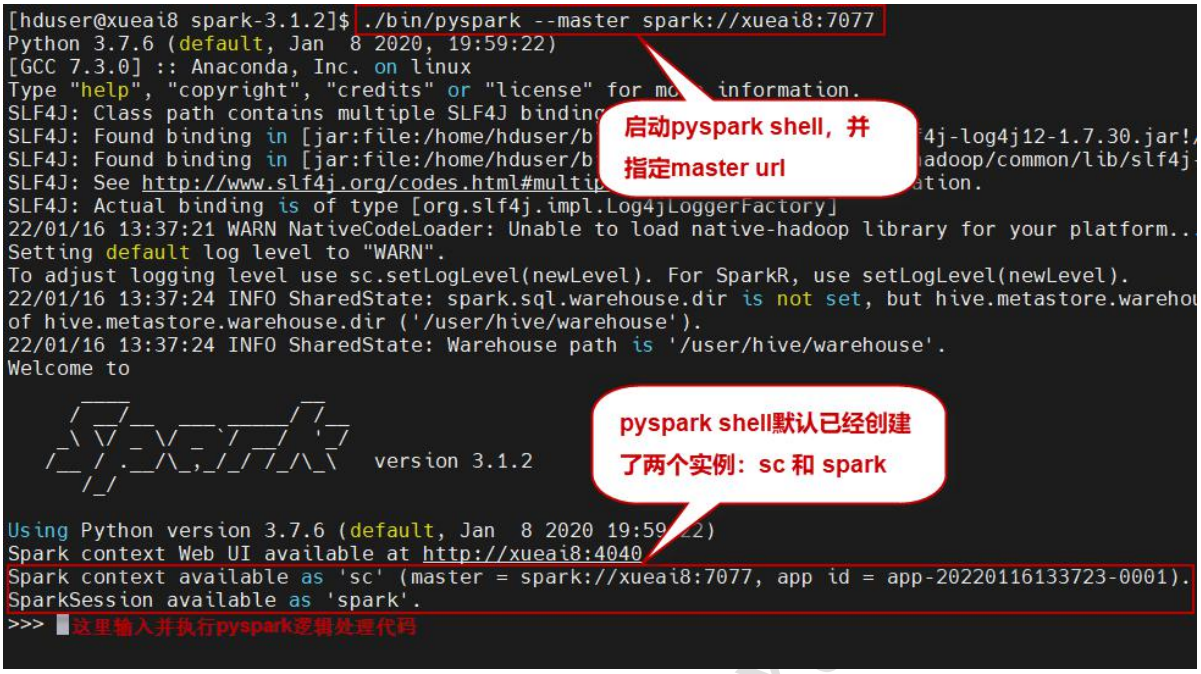
使用 jps 命令查看启动的进程。如果有 master 和 worker 进程，说明 Spark 集群已经启动。



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

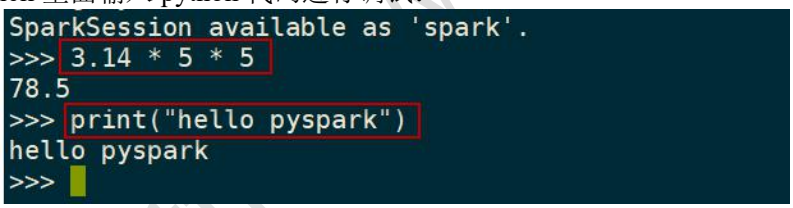
```
$ ./bin/pyspark --master spark://xueai8:7077
```

在 Master URL 中指定的 xueai8 是当前的机器名。



1.7.3 pyspark shell 常用命令

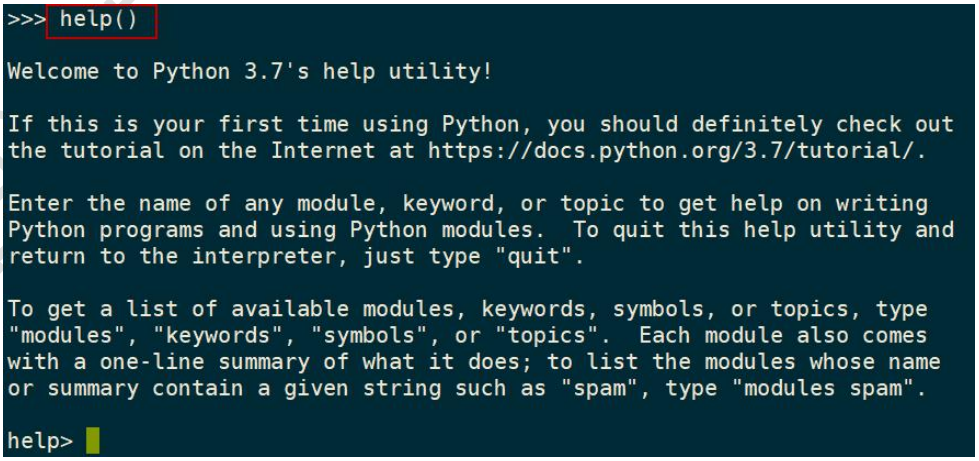
可以在 pyspark shell 里面输入 python 代码进行调试：



可以 pyspark shell 中键入以下命令，查看 pyspark shell 常用的命令：

```
>>> help()
```

如下图所示：



可以 help 模式下键入模块的名称，查看该模块的使用说明：

```
help> pyspark.sql
```

会显示如下的使用说明界面：

```
Help on package pyspark.sql in pyspark:

NAME
    pyspark.sql - Important classes of Spark SQL and DataFrames:

DESCRIPTION
        - :class:`pyspark.sql.SparkSession`
          Main entry point for :class:`DataFrame` and SQL functionality.
        - :class:`pyspark.sql.DataFrame`
          A distributed collection of data grouped into named columns.
        - :class:`pyspark.sql.Column`
          A column expression in a :class:`DataFrame`.
        - :class:`pyspark.sql.Row`
          A row of data in a :class:`DataFrame`.
        - :class:`pyspark.sql.GroupedData`
          Aggregation methods, returned by :func:`DataFrame.groupBy`.
        - :class:`pyspark.sql.DataFrameNaFunctions`
          Methods for handling missing data (null values).
        - :class:`pyspark.sql.DataFrameStatFunctions`
          Methods for statistics functionality.
        - :class:`pyspark.sql.functions`
          List of built-in functions available for :class:`DataFrame`.
        - :class:`pyspark.sql.types`
          List of data types available.
        - :class:`pyspark.sql.Window`
          For working with window functions.

PACKAGE CONTENTS
    catalog
    column
    conf
    context
    dataframe
    functions
    group
    readwriter
:
```

按 q 键退出 help 帮助界面，输入 exit()命令，返回 shell 命令行。

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以及各种格式的结构化数据和二进制数据文件读取数据的 API，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 pyspark 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 pyspark shell 中直接使用。我们可以使用 type()函数来验证这一点。

```
>>> type(sc)
>>> type(spark)
```

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:

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

执行过程如下图所示：

要查看在 pypark 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 pyspark shell 中默认的配置信息：

```
>>> for item in spark.sparkContext.getConf().getAll():
...         print(item)
```

执行过程如下图所示：

或者，也可以使用如下的命令：

```
>>> for item in sc.getConf().getAll():
...     print(item)
```

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 会增加该端口号值，直到找到一个空闲的端口号为止。

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。

下图所示是 Spark Web UI 欢迎页面的一个示例。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 PySpark 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 PySpark 应用程序(或 job 作业)。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。

下面是一个带有最常用命令选项的 spark-submit 命令。

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：
-
-
-
-
- 。

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。

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。：

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。

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：

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。SparkConf 提供了如下几个常用的设置方法：

- set(key, value)：设置一个配置属性。
- setMaster(value)：设置 master URL。
- setAppName(value)：设置一个应用程序名称。
- get(key, defaultValue=None)：获得指定 key 的配置值。
- setSparkHome(value)：在 worker 节点上设置 Spark 安装路径。

使用 SparkConf 进行设置如下代码片段所示：

或者也可以调用 config(key, value)方法进行设置。如下代码片段所示：

通过 SparkContext 的 getConf()方法可以获取配置对象，查看所有的配置项。如下代码片段所示：

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:

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：

```
```

下面的示例使用其他 python 文件作为依赖项。

```
```

1.8.2 提交 pi.py 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 PySpark 集群上以 standalone 模式运行，以掌握 spark-submit 提交 PySpark 程序的方法。

请按以下步骤操作。

- 。

运行结果如下图所示：

. . . . . .

1.8.3 提交 PySpark 程序到 YARN 集群上执行

也可以将 PySpark 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 pi.py 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 Hadoop/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 --master yarn examples/src/main/python/pi.py
```

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 PySpark 应用程序

要开发 PySpark 应用程序，通常可以采用以下几种开发方式和开发环境：

- 使用 pyspark shell，交互式执行；
- 使用 PyCharm IDE 集成开发环境，先开发测试，然后部署执行；
- 使用 Jupyter Notebook，交互式开发；
- 使用 Zeppelin Notebook，交互式开发。

在上一章，我们已经了解了如何使用 pyspark shell 以交互式方式执行 PySpark 代码。但是 pyspark shell 并不适合在生产（工作）环境下使用。在生产（工作）环境中，我们可以根据自己的需求选择后面三种开发和执行方式。

在本配置之前，读者需要先在 Linux/CentOS 上安装好 Python 3。作者是通过 Anaconda 安装的，所以这里的配置中引用的 Python 为 Anaconda 所带的 Python。如果读者的安装的 Python 3 与本书不一致，请自行修改。

推荐：为避免繁琐易错的配置，推荐直接下载使用小白学苑提供的 PBLP 个人大数据学习平台。该平台已经配置好了 Hadoop + PySpark + Zeppelin + Jupyter 的大数据学习和开发环境。

2.1 使用 PyCharm 开发 PySpark 应用程序

在这一节，向大家介绍如何使用 PyCharm 这个 IDE 来开发 PySpark 应用程序。

我们将使用 PyCharm Community Edition 作为 IDE。在本教程的最后，将了解如何使用 PyCharm 设置 PySpark，以及如何将代码部署到集群中。

2.1.1 安装 PyCharm

1、首先去 Pycharm 官网，下载 PyCharm 安装包（去官网下载），根据自己电脑的操作系统进行选择。PyCharm 分为收费的企业版和免费的社区版。开发 PySpark 应用程序，使用社区版即可。对于 windows 系统选择下图的框选的安装包。

2、双击下载的安装包，进行安装。一路 Next 即可。安装完成，会在电脑桌面生成如下的启动图标，双击它可以启动 PyCharm：

3、下面是 PyCharm 中常用的一些快捷键：

- Ctrl + Enter：在下方新建行但不移动光标；
- Shift + Enter：在下方新建行并移到新行行首；
- Ctrl + /：注释(取消注释)选择的行；
- Ctrl+d：对光标所在行的代码进行复制。

2.1.2 创建一个新的 PyCharm 项目

要创建一个新的项目，首先启动 PyCharm，选择 File > New Project，选择 Pure Python，并在右侧指定项目代码所在的位置，将项目命名为 HelloSpark。单击 Create 按钮创建此项目。

这样就创建了一个空的项目 HelloSpark。
在本地文件系统，将 shakespeare.txt 拷贝到项目的根目录下，比如~/PythonProjects/HelloSpark。

2.1.3 安装 pyspark 包

要安装 pyspark 包，请在 Pycharm 中导航到 File > Settings ...，打开项目设置面板，如下图所示：

在项目设置面板中，选择左侧的 Project:HelloSpark > Project Interpreter，然后在右侧单击右边栏的绿色＋按钮：

在上方的搜索框内输入 pyspark，搜索并选择 pyspark，然后的面板右侧选择指定的版本（本书使用的 Spark 版本是 2.4.7），最后单击单击 Install Package 按钮安装。如下图所示：

2.1.4 创建 PySpark 应用程序

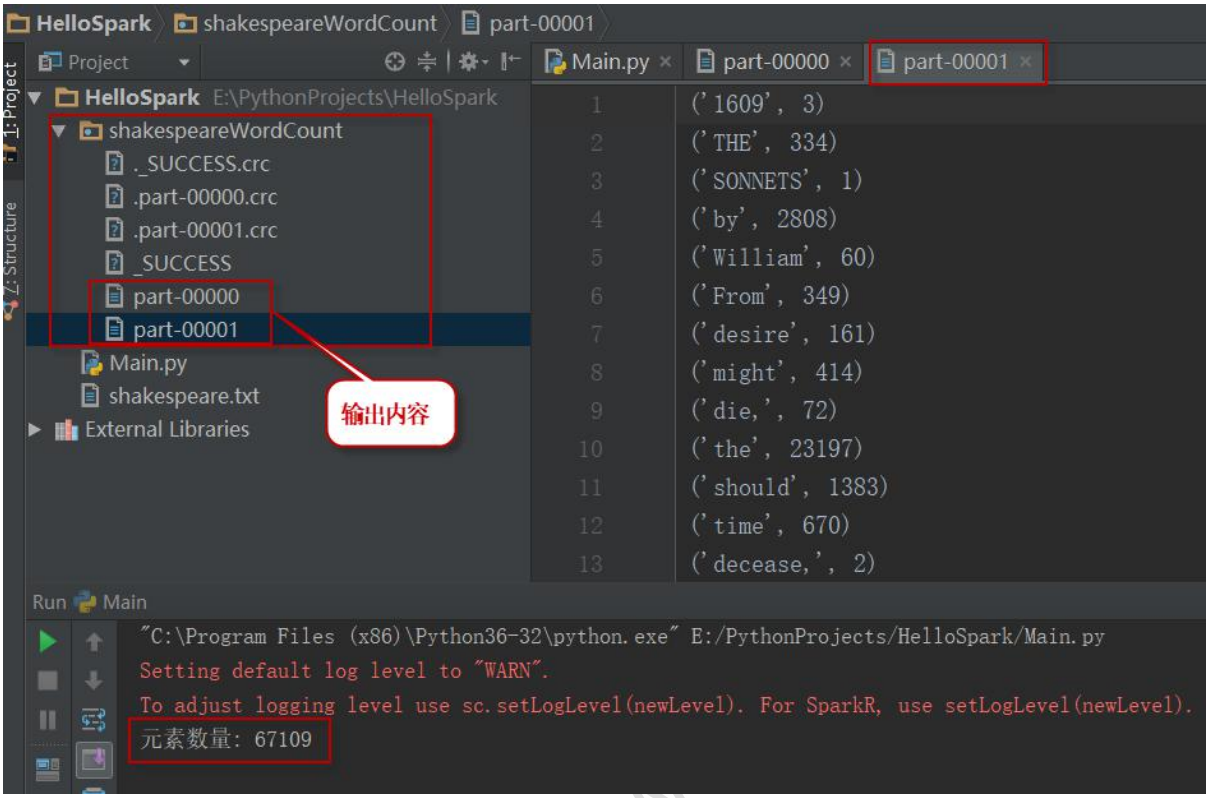
在刚创建的 HelloSpark 项目目录上，单击右键，选择 New > File，创建一个新的 Python 源文件，并命名为 Main.py。

编辑 Main.py 文件内容如下：

```
```

要运行程序，请从 IDE 菜单中单击 Run > Run…并选择 Main。如下图所示：

程序计算结果保存在 shakespeareWordCount 文件夹中，该文件夹与源代码的文件夹位于相同的目录下。如下图所示：



2.1.5 部署到集群中运行

接下来我们将 Spark 作业部署到集群上运行。

.....

4）输出结果被保存到 HDFS 上：/data/spark_demo/shakespeareWordCount。其内容如下：

```
(u'fawn', 11)
(u'Fame,', 3)
(u'mustachio', 1)
(u'protested,', 1)
(u'sending.', 3)
(u'offendeth', 1)
(u'instant;', 1)
(u'scold', 4)
(u'Sergeant.', 1)
(u'nunnery', 1)
(u'Sergeant,', 2)
...
```

2.2 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。

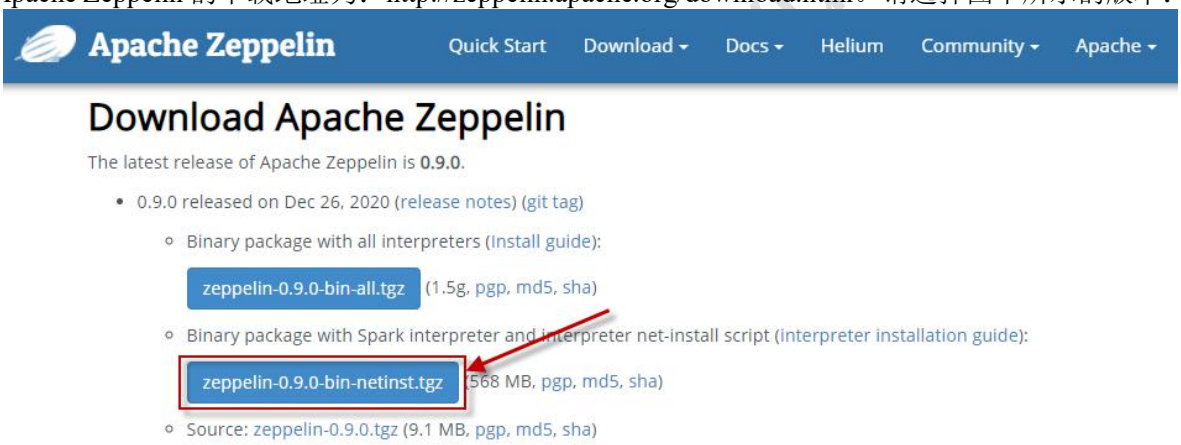
Apache Zeppelin 专注于企业级应用，Zeppelin Notebook 可以满足以下企业用户以下需求：

- 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。

2.2.1 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。

2.2.2 安装和配置 Zeppelin

请按以下步骤安装和配置 Zeppelin。

2.2.3 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，添加一个 SPARK_HOME 属性，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，这里我们使用的是 spark

standalone 模式。这相当于启动 pyspark shell 时指定--master 参数）。

再新增加两个 PySpark Python 的相关设置，如下图所示。

然后单击【Save】按钮保存。

2.2.4 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-Python 语言

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：

2.3 Jupyter Notebook 进行交互式分析

数据分析师最喜欢的一个交互式分析工具是 Jupyter Notebook，因此也希望在应用 Spark 进行大数据分析时也使用 Jupyter。下面我们就配置 PySpark 与 Jupyter 的组合。

有两种方法可以使 PySpark 在 Jupyter Notebook 中可用：

- 配置 PySpark 驱动程序使用 Jupyter Notebook：运行 pyspark 将自动打开一个 Jupyter Notebook。
- 加载一个普通的 Jupyter Notebook，并使用 findSpark 包加载 PySpark。

第一种方法更快，但是特定于 Jupyter 笔记本；第二种方法是一种更广泛的方法，可以在自己喜欢的 IDE 中使用 PySpark。

2.3.1 配置 PySpark Driver 使用 Jupyter Notebook

请按以下步骤配置和启动 Spark 及 Jupyter Notebook。

如下图所示：

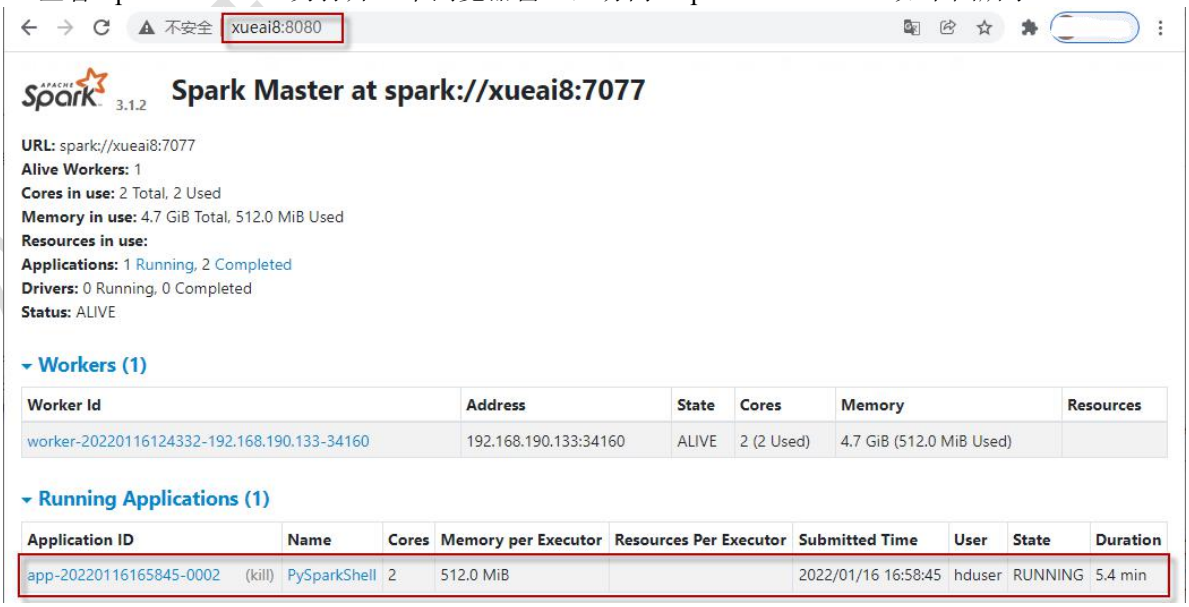
注意，不要关闭此窗口，保持服务一直处于运行状态。

4）在 Windows 下，通过浏览器访问 Jupyter。到 Windows 下，打开浏览器，粘贴上一步复制的 URL，回车访问，打开 notebook 页面。如下图所示：

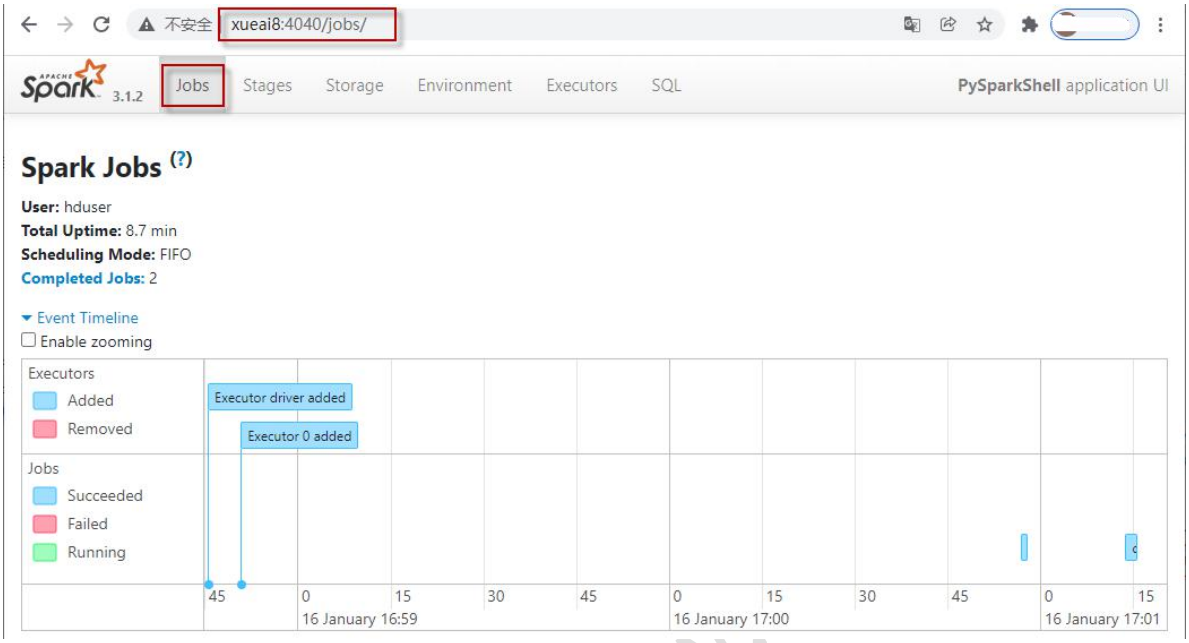
5）编写 PySpark 代码。新建一个 notebook，输入以下代码执行：

```
In [1]: from pyspark.sql import SparkSession

In [2]: spark = SparkSession.builder \
            .master("spark://xueai8:7077") \
            .appName("pyspark demo") \
            .getOrCreate()

In [3]: spark.version
Out[3]: '3.1.2'

In [4]: numbers = [1,2,3,4,5,6,7,8,9,0]

        rdd = spark.sparkContext.parallelize(numbers)

In [5]: rdd.collect()
Out[5]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

In [6]: rdd.map(lambda n: n*n).collect()
Out[6]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 0]
```

6）查看 Spark Web UI。另打开一个浏览器窗口，访问 http://xueai8:8080。如下图所示：



7）查看正在执行的 Spark 作业的 Web UI。另打开一个浏览器窗口，访问 http://xueai8:4040，可以看到正在执行和已经执行完成的作业情况。如下图所示：



2.3.2 使用 findSpark 包

在 Jupyter 笔记本中使用 PySpark 还有另一种方法是，使用 findSpark 包使 Spark 上下文在代码中可用。

findSpark 包不是特定于 Jupyter 笔记本的，我们也可以在自己喜欢的 IDE 中使用这个技巧。

2.4 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 编程语言提供了编程 API。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：

- ；
-
-
-

- 。

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：

-
-
-
-
-
-
- 。

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 Jupyter Notebook 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）首先启动 HDFS 集群和 Spark 集群。

```
$ start-dfs.sh

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

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

（2）首先准备一个文本文件 word.txt，内容如下：

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

（3）将该文本文件上传到 HDFS 的"/data/spark/"目录下：

```
$ hdfs dfs -put word.txt /data/spark/
```

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

（5）首先创建 SparkSession 和 SparkContext 的实例。

```
from pyspark.sql import SparkSession

# 构建 SparkSession 和 SparkContext 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark demo") \
    .getOrCreate()

sc = spark.sparkContext
```

（6）读取数据源文件，构造一个 RDD

```
source = "/data/spark/word.txt"
textFile = sc.textFile(source)
```

（7）将每行数据按空格拆分成单词 - 使用 flatMap 转换

```
words = textFile.flatMap(lambda line: line.split(" "))
```

（8）将各个单词加上计数值 1 - 使用 map 转换

```
wordPairs = words.map(lambda word: (word,1))
```

（9）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

```
wordCounts = wordPairs.reduceByKey(lambda a,b: a + b)
```

（10）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()
```

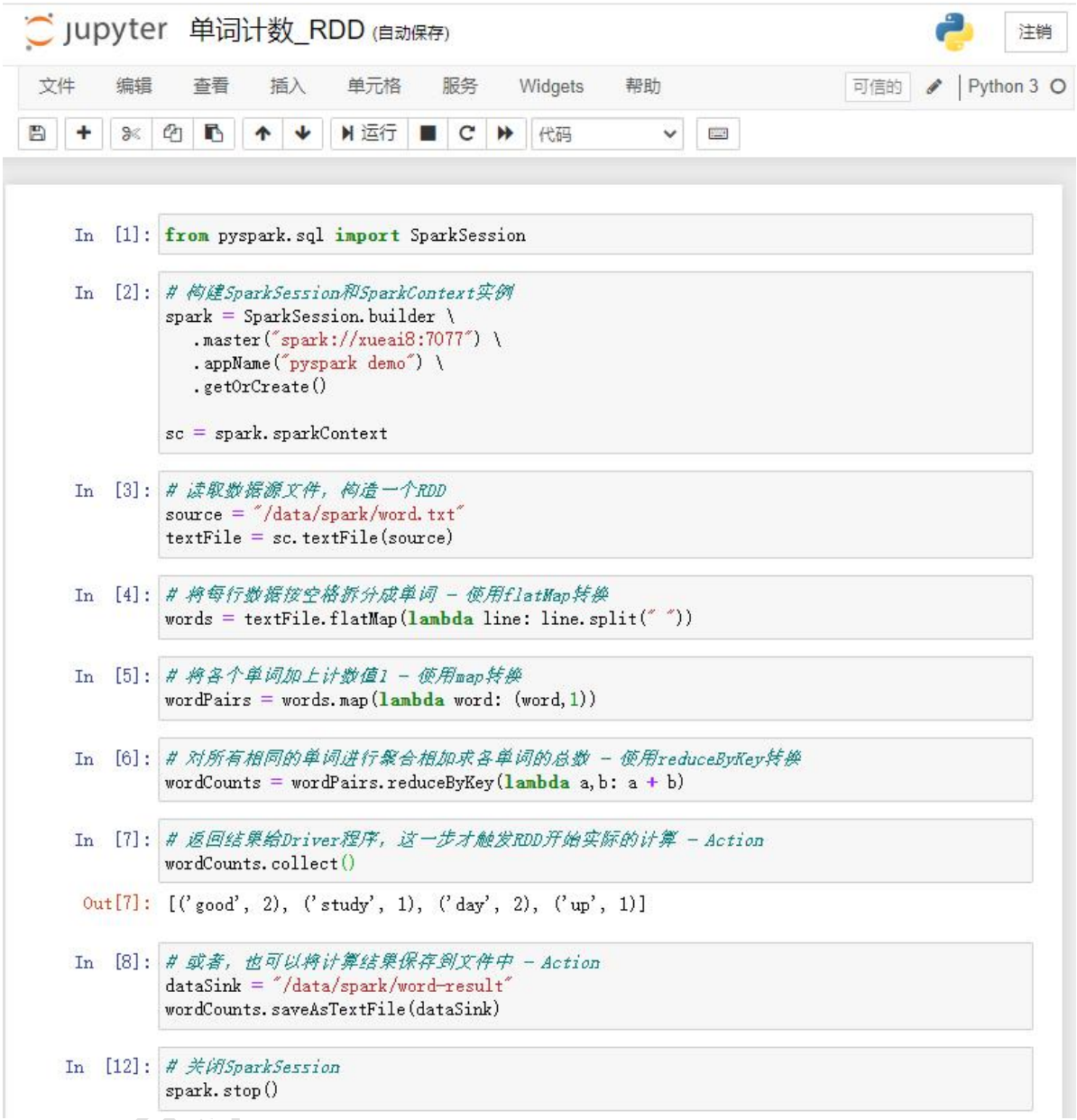
可以看到输出结果如下：

```
[('good', 2), ('study', 1), ('day', 2), ('up', 1)]
```

（11）或者，也可以将计算结果保存到文件中 - Action

```
dataSink = "/data/spark/word-result"
wordCounts.saveAsTextFile(dataSink)
```

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：

```
source = "/data/spark/word.txt"
sc.textFile(source) \
   .flatMap(lambda line: line.split(" ")) \
   .map(lambda word: (word,1)) \
   .reduceByKey(lambda a,b: a + b) \
   .collect()
```

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？

。。。。。。

在 pyspark shell 中的 SparkSession

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

在 PySpark 程序中创建 SparkSession

要在 PySpark 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。

。。。。。。

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()方法)。

SparkContext in pyspark shell

。。。。。。。

在 PySpark 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:

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。

```
sc = 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。

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：

执行过程及输出结果如下图所示：

```
In [1]:
from pyspark.sql import SparkSession

spark = SparkSession.builder \
   .master("spark://xueai8:7077") \
   .appName("pyspark demo") \
   .getOrCreate()

sc = spark.sparkContext
```

```
In [2]:
# 通过并行集合（数组）创建RDD
array1 = [1,2,3,4,5,6,7,8,9,10]
rdd1 = sc.parallelize(array1)
# rdd1 = sc.parallelize([1,2,3,4,5,6,7,8,9,10])

rdd1.collect()
```

```
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
In [3]:
# 或者并行化list
list2 = list(range(11))    # range(start=0, end, step=0)
rdd2 = sc.parallelize(list2)

rdd2.collect()
```

```
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
In [4]:
# 通过并行集合（数组）创建RDD
strList = ["明月几时有","把酒问青天","不知天上宫阙","今夕是何年"]
strRDD = sc.parallelize(strList)

strRDD.collect()
```

```
Out[4]: ['明月几时有', '把酒问青天', '不知天上宫阙', '今夕是何年']
```

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：

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（因此也不会被发现）。

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：

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 RDD 的方法，而动作(action)是使用 RDD 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：

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

```
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：

```
filteredLines = lines.filter(lambda 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：

```
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。

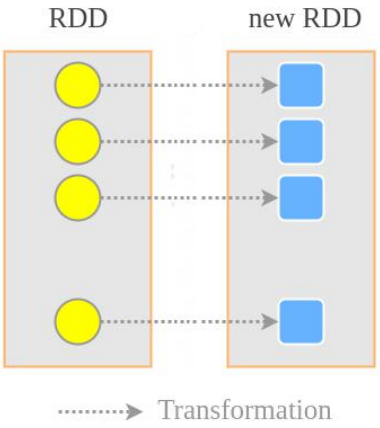
现在假设有一个 RDD，包含元素为{1, 2, 3, 3}。首先，让我们构造出这个 RDD：

```
.....

# 构造一个 RDD
rdd = sc.parallelize([1,2,3,3])
```

接下来，学习普通 RDD 上的各种转换操作方法：

- map(func)

map 是使用函数转换每个 RDD 元素并返回一个新的 RDD。



输出结果如下：

```
[2,3,4,4]
```

map 转换练习：分析以下转换后的结果。

```
```

❑ mapPartitions(func)

mapPartitions 是一个转换操作，应用于 RDD 中的每个分区。它是 RDD 的一个属性，它将一个函数应用到 RDD 的分区上，并返回一个新的 RDD。

在执行 MapPartitions 时没有数据移动或 shuffling。输出返回的行数与输入行相同。

需要在每个分区上一次性调用的数据模型的大量初始化可以通过使用 MapPartitions 来完成。RDD 将数据存储在一个分区中，在该分区中，操作应用于每个元素，而在 MapPartitions 中，该函数应用于 RDD 数据模型中的每个分区。因此，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。

```
```

执行以上代码，输出结果如下：

```
[3, 6]
```

mapPartitions 将结果保存在内存中，直到所有的行都在分区中处理完毕。

这个函数可以用来创建在每个分区中应用一次的逻辑，比如创建连接、终止连接。

mapPartitions 转换练习：分析以下转换后的结果。

```
```

❑ mapPartitionsWithIndex(func)

mapPartitionsWithIndex(f)类似于 map，但在每个分区上单独运行 f 函数，并提供分区的索引。请看下面的示例代码：

```
```

执行以上代码，输出结果如下：

```
[(0, 1),
 (0, 2),
 (0, 3),
 (0, 4),
 (0, 5),
```

```
]
```

可将 mapPartitionsWithIndex 用于确定分区内的数据倾斜。请看下面的示例代码：

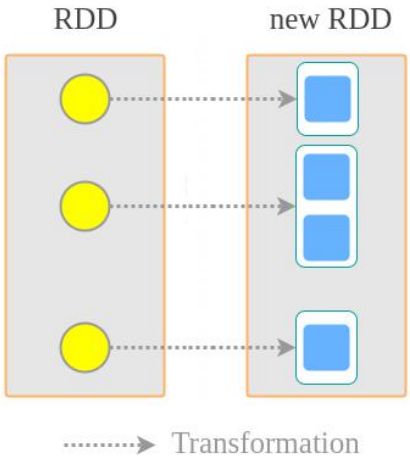
```
```

执行以上代码，输出结果如下：

```
[[0, 5], [1, 6]]
```

- ❑ flatMap(func)

使用一个函数来转换每个 RDD 元素，这个函数可以将多个元素返回到新的 RDD。




```
```

执行以上代码，输出结果如下所示：

```
```

flatMap 转换练习：分析以下转换后的结果。

```
```

- ❑ filter(func)

```
```

执行以上代码，输出结果如下所示：

```
```

filter 转换练习：分析以下转换后的结果。

```
```

- ❑ 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。

```
```

执行以上代码，输出结果如下所示：

```
[2, 3]
```

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

执行以上代码，输出结果如下所示：

```
[2, 1, 3]
```

distinct 转换练习：分析以下转换后的结果。

- 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。

执行以上代码，输出结果如下所示：

```
[('J', 'John'), ('F', 'Fred'), ('A', 'Anna'), ('J', 'James')]
```

- 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。

执行以上代码，输出结果如下所示：

```
('J', ['Joseph', 'Jimmy', 'James', 'Jackeline', 'Juan'])
('T', ['Tina', 'Thomas'])
('C', ['Cory', 'Christine'])
```

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

返回这个按给定 key 函数排序的 RDD。

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。

- 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，PySpark 的 repartition()是非常昂贵的操作，因为它会跨多个分区转移数据。

- 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。

- 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。

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：

接下来操作这两个 RDD，如下：

- union(otherDataset)

union 转换练习：分析以下转换后的结果。

- intersection(otherDataset)

intersection 转换练习：分析以下转换后的结果。

- subtract(otherDataset)

- 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。

执行上面的代码，输出结果如下：

cartesian 转换练习：分析以下转换后的结果。

- 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)。

详细 API 说明请参考：

http://www.xueai8.com

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：

- 。
-
-
-
- 。

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。

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。

- 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。

例如，有一个 RDD，包含元素[1,2,3,4]，分区数为 2。那么可以通过 aggregate 函数来计算 RDD 元素的平均值。计算过程如下图所示。

【示例】使用 Spark RDD aggregate()函数计算 RDD 元素的平均值。

分析：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)。

```
from pyspark.sql import SparkSession

# 构建 SparkSession 和 SparkContext 实例
```

执行以上代码，输出结果如下：

```
(20, 7)
```

更多 aggregate 函数的练习，如下。

本节练习：

示例 1：给定一个文本文件，统计一个某字符所出现的行数。

示例 2：找出文本文件中单行文本所包含的单词数量的最大值

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：

用直方图可视化数据分布：

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：

执行以上代码，输出结果如下：

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 RDD 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：

第二种方式：通过并行集合创建 Pair RDD。请看下面的代码：

也可以使用 keyBy()函数自定义 key 的分组规则。

执行以上代码，输出结果如下：

3.5.2 操作 Pair RDD

假设有一个 Pair RDD {(1,2),(3,4),(3,6)}。

1) keys：返回所有的 key。

2) values：返回所有的 value。

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。

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。

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。

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。

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。

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)。

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）

- ❑
- ❑
- ❑
- ❑
- ❑

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：

- ❑
- ❑
- ❑
- ❑
- ❑
- ❑

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。

例如，在下面的代码中，创建了两个 Pair RDD，并执行 join 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]）。

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。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：

```

```

执行以上代码，输出结果如下：

```
[('39657', 3), ('about', 4), ('snail', 1), ('xlw', 0), ('xueai8', 2)]
```

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：

```

```

将其构造为具有 3 个分区的一个 RDD：

```
x = spark.sparkContext.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 的所有数据 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。

```

```

执行以上代码，输出结果如下：

```

```

再看下面这个示例。

```

```

执行以上代码，输出结果如下：

```
http://www.xueai8.com
```

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：

将其构造为具有 3 个分区的一个 RDD：

```
x = spark.sparkContext.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。

执行以上代码，输出结果如下：

注意：由于数据集可以有非常多的键(key)，所以 reduceByKey()不是作为一个返回值给用户程序的 action 实现的。相反，它返回一个新的 RDD，由每个键(key)和该键的 reduce 值组成。

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。

输出结果如下：

```
('Jimmy', 97)
('Tina', 87)
('Thomas', 93)
('Joseph', 91)
('Cory', 71)
('Jackeline', 86)
('Juan', 69)
```

【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：

```
('Jimmy', ('Chemistry', 97))
('Tina', ('Biology', 87))
('Thomas', ('Physics', 93))
('Joseph', ('Chemistry', 91))
('Cory', ('Chemistry', 71))
('Jackeline', ('Maths', 86))
('Juan', ('Physics', 69))
```

【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：

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

aggregateByKey 函数的特点总结如下：

- 性能方面的 aggregateByKey 是一个优化的 transformation 操作；

- 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，它在最后阶段需要 shuffle 数据。另外，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：

- 。

mergeValue 函数：

- 。

mergeCombiners 函数：

- 。

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：

```
```

执行以上代码，输出结果如下：

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

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：

接下来，对两个 RDD 进行转换操作：

1) subtractByKey：接 key 做差集运算。

2) join：内连接

3) leftOuterJoin：左外连接

4) rightOuterJoin：右外连接

5) fullOuterJoin：全外连接

6) cogroup：对来自两个 RDD 的数据按 key 分组。

执行以上代码，输出结果如下：

```
1: 2
3: 4 6 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：

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。

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：

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。

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：

```
```

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。方法定义如下：

```
SparkContext.textFile(name, minPartitions=None, use_unicode=True)
SparkContext.parallelize(c, numSlices=None)
```

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。

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。

。。。。。。

【示例】对 RDD 进行重分区。

```
```

3.7.2 使用数据分区函数

当需要对 Pair RDD 进行重分区时，RDD 的分区由 portable_hash 函数执行，该分区程序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）。

。。。。。。：

```
```

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 portable_hash 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。

partitionBy()方法

可以使用 partitionBy()方法对 Pair RDD 进行重分区，该方法定义如下：

```
RDD.partitionBy(numPartitions, partitionFunc=<function portable_hash>)
```

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：期望的分区数量。如果分区程序与之前使用的分区程序相同，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，使用指定的分区数。

```
```

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。

自定义数据分区程序

当需要精确地在分区中放置数据时，也可以自定义分区程序。自定义分区程序只可以在 Pair RDD 上使用。

如果 Pair RDD 转换没有指定一个分区程序，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 中没有定义一个分区程序，那么将使用 portable_hash，分区数由 spark.default.parallelism 参数指定。

例如，在下面这个示例中，有一个整数列表[10,20,30,40,50,10,20,35]。现在想将它分为两个分区，比如 P1 和 P2。P1 包含所有小于 30 的元素，p2 包含所有大于 30 的元素。

【示例】使用自定义分区程序对 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。

```
```

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。

```
```

repartition() vs partitionBy()

从前面的内容可知，repartition()方法和 partitionBy()方法都可以调整 RDD 的分区数。但这两个方法是有区别的。repartition()不能指定分区函数，它主要用于根据内核数量和数据量指定分区的数量。repartition()实际上是通过在现有值上添加随机键来在内部使用 pair RDD，因此它不能对输出数据分布提供强有力的保证。

两者的区别可以通过以下代码来体会：

执行以上代码，输出结果如下：

实际上，Spark 中的任何重分区都是使用 Pair RDD 来处理的。如果需要，Spark 只需添加虚拟键或虚拟值使其工作。

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。

- 。。。。。。。

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

.....。

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

。。。。。。

Stage

.....。

Task

.....。

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

.....

TaskScheduler

DAGScheduler 将一个 TaskSet 交给 TaskScheduler 后，TaskScheduler 会为每个 TaskSet 进行任务调度，.....。

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。

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。

。。。。。。

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 Spark 资源调度策略

Spark 应用程序的资源被作为 executors（JVM 进程）和 CPU（任务槽-task slots）进程调度，然后将内存分配给它们。当前正在运行的集群的集群管理器和 Spark 调度程序授予用于执行 Spark 作业的资源。

集群管理器启动 driver 所请求的 executor 进程，并且当运行在集群部署模式下时启动 driver 进程本身。集群管理器还可以重新启动和停止它已经启动的进程，并且可以设置 executor 进程可以使用的最大 CPU 数量。

一旦应用程序的 driver 和 executors 开始运行，Spark 调度程序就会直接与它们通信，并决定哪些 executors 将运行哪些 tasks，这称为“作业调度”，它会影响集群中的 CPU 资源使用情况。间接地，它也会影响内存使用，因为在单个 JVM 中运行的任务越多，使用的堆内存就越多。然而，内存并不是像 CPU 那样在任务级别上直接管理的。Spark 管理由集群管理器分配的 JVM 堆内存，将其分成几个部分。

在集群中运行的每个 Spark 应用程序都会被分配一组专用的 executors。如果在单个集群中运行多个 Spark 应用程序（或其他类型的应用程序），它们就会争夺集群的资源。

因此，存在两个层次的 Spark 资源调度：

- 集群资源调度：为不同的 Spark 应用程序的 Spark executors 分配资源
- Spark 资源调度：用于在单个应用程序内调度 CPU 和内存资源

3.9.1 CPU 资源分配策略

Spark 分配 CPU 资源的方式有两种：FIFO（先进先出）调度和 fair scheduling（公平调度）。可通过 Spark 参数 spark.scheduler.mode 设置调度器模式，它有两个可能的值：FAIR 和 FIFO。

（1）FIFO 调度

“先到先服务”。请求资源的第一个 job 占用了所有必需的（和可用的）executor task slots（假设每个 job 作业只包含一个 stage 阶段）。如下图所示：

FIFO 调度是默认的调度器模式，如果应用程序是一个只运行一个作业的单用户应用程序，那么它的效果最好。

（2）FAIR 调度

公平调度以循环的方式在竞争的 Spark jobs 中均匀地分配可用资源（executor 线程）。对于同时运行多个作业的多用户应用程序来说，这是一个更好的选择。如下图所示：

3.9.2 Spark 内存管理

要设置分配给 executors 的内存数量，使用 spark.executor.memory 参数。大小可以使用 g 和 m 做为后缀。默认的 executor 内存大小是 1G。

Spark 保留部分内存用于缓存的数据存储和临时 shuffle 数据。Spark 为这些缓存数据和 shuffle 数据设置堆（heap），使用参数有 spark.storage.memoryFraction(默认是 0.6)和 spark.shuffle.memoryFraction(默认是 0.2)。因为堆的这些部分可以在 Spark 能够测量和限制它们之前增长，所以必须设置两个额外的安全参数：spark.storage.safetyFraction(默认是 0.9)和 spark.shuffle.safetyFraction(默认是 0.8)。安全参数按指定的数量降低内存比例，如下图所示：

默认存储的堆的实际部分是 0.6 x 0.9（安全系数乘以存储内存系数），等于 54%。类似地，用于 shuffle 数据的堆的一部分是 0.2 x 0.8（安全分数乘以 shuffle 内存分数），它等于 16%。

然后，剩下 30%的堆预留给其他 Java 对象和运行任务所需的资源（然而，应该只期望 20%）。

可使用 spark.driver.memory 设置 driver 内存。当使用 spark shell 和 spark-submit 脚本（在集群和客户端部署模式中）启动应用程序时，这个参数就应用了。

如果从另一个应用程序（客户端模式）以编程方式启动，则使用-Xmx Java 选项来设置包含进程的 Java 堆的大小上限。

Spark on YARN 的 executors 的内存组成

在 YARN 上运行时，Spark 的 executors 的内存布局如图所示：

如果设置 spark.executor.memoryOverhead 的值不够高的话，会导致难以诊断的问题。确保指定至少 1024 MB。

。。。。。。

以编程方式配置 executor 资源

当在用 spark.executor.cores、spark.executor.instances 和 spark.executor.memory 以编程方式创建 Spark context 上下文对象时，也可以指定 executor 资源。但是，如果应用程序在 YARN 集群模式下运行，那么驱动程序 driver 就在它自己的 container 中运行，与 executor 的 containers 并行启动运行，因此这些参数将不会产生任何影响。在这种情况下，最好在 spark-defaults.conf 文件中或在命令行上设置这些参数。

3.10 使用共享变量

除了 RDD，Spark 中还提供了另一个数据抽象“共享变量”。共享变量可以在并行操作中使用。默认情况下，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，它会将函数中使用的每个变量的副本发送给每个任务，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。但是有时候，我们需要在任务之间共享同一个变量，或者在任务和驱动程序之间共享同一个变量。

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。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。

3.10.1 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。

.....

请看下面的示例：

Broadcast VS Cache

……。

手动实现 broadcast hash join

。 。 。 。 。 。

Partial manual broadcast hash join

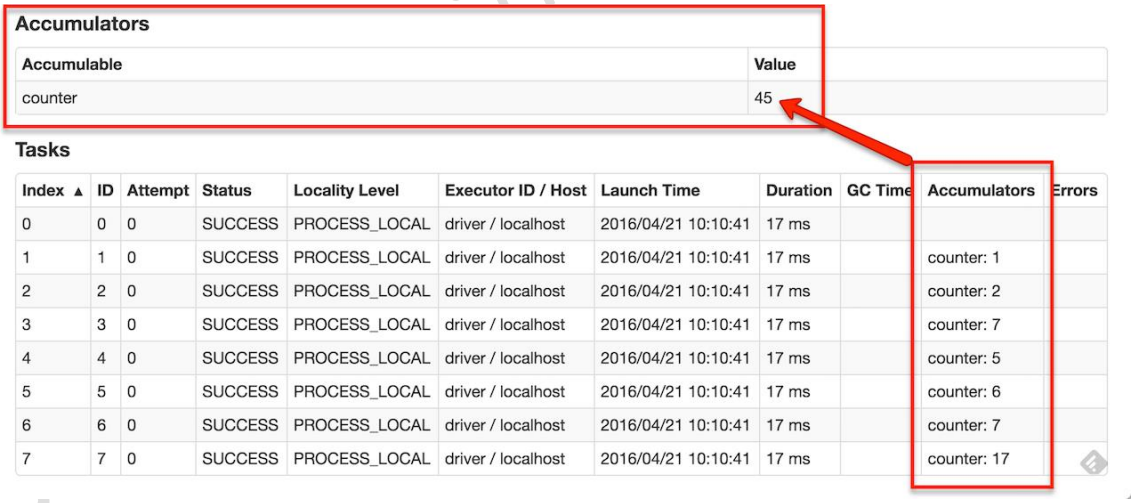
。 。 。 。 。 。

3.10.2 累加器

累加器(Accumulators)用来把 executor 端变量信息聚合到 driver 端。在 driver 程序中定义的变量，在 executor 端的每个 Task 都会得到这个变量的一份新的副本，每个 task 更新这些副本的值后，传回 driver 端进行 merge。

累加器是跨 executors 之间共享的分布式只写共享变量，每个 executor 之间是不可见的，只有 driver 端可以对其读操作。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。



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

……。

下面是使用累加器的示例：

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。

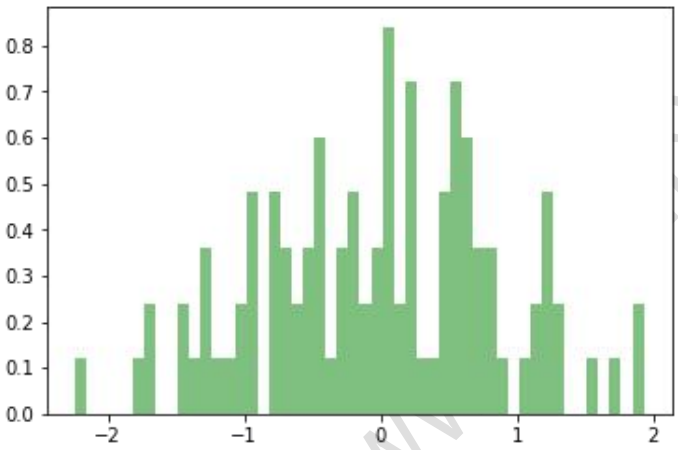
【示例】删除指定数组是既能被 2 整除也能被 3 整除的数，并统计删除了多少个。

3.11 PySpark RDD 可视化

PySpark RDD 还没有任何绘图功能。如果想绘制一些内容，可以将数据从 SparkContext 中取出并放入“本地”Python 会话中，在那里可以使用 Python 的任意一个绘图库来处理它。对于 RDD，调用.collect()方法，将数据返回到 driver 再绘制。

请看下面的示例。

执行以上代码，执行结果如下所示：



3.12 PySpark RDD 编程案例

本节我们应用前面所学到的知识，实现几个常见的算法场景。

3.12.1 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）。

3.12.2 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。

3.12.3 数据聚合计算

【示例】给定一些销售数据，数据采用键值对的形式(公司，收入)，求出每个公司的总收入和平均收入。(提示：可直接用 sc.parallelize 在内存中生成数据，在求每个公司总收入时，先分三个分区进行求和，然后再把三个分区进行合并。只需要编写 RDD combineByKey 函数的前三个参数的实现)

执行以上代码，输出结果如下：

```
('company-1', 259, 86.33333333333333)
('company-3', 262, 87.33333333333333)
('company-2', 259, 86.33333333333333)
```

3.12.4 合并小文件

【示例】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。

3.12.5 实现二次排序

【示例】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：

第 4 章 PySpark SQL

4.6.4 读取 Parquet 文件创建 DataFrame

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：

```
file = "file:///home/hduser/data/spark/resources/users.parquet"

# 读取 parquet 文件
# Parquet 文件是自描述的，因此模式得以保留
# 加载 Parquet 文件的结果也是一个 DataFrame
parquet_df = spark.read.load(file,format="parquet")
# parquet_df = spark.read.parquet(file)        # 简洁写法

# 输出模式和内容
parquet_df.printSchema()
parquet_df.show()
```

执行过程如下所示：

```
file = "file:///home/hduser/data/spark/resources/users.parquet"

# 读取parquet文件
# Parquet文件是自描述的，因此模式得以保留
# 加载Parquet文件的结果也是一个DataFrame
parquet_df = spark.read.load(file,format="parquet")
# parquet_df = spark.read.parquet(file)      # 简洁写法

# 输出模式和内容
parquet_df.printSchema()
parquet_df.show()
```

```
root
 |-- name: string (nullable = true)
 |-- favorite_color: string (nullable = true)
 |-- favorite_numbers: array (nullable = true)
 |    |-- element: integer (containsNull = true)
```

```
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

4.6.5 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
orcFile = "file:///home/hduser/data/spark/resources/users.orc"

# 读取 ORC 文件，构造 DataFrame
orc_df = spark.read.format("orc").load(orcFile)
# orc_df = spark.read.load(orcFile,format="orc")
# orc_df = spark.read.orc(orcFile)    # 简洁写法

orc_df.printSchema()
orc_df.show()
```

执行过程如下所示：

```
# 数据源文件
orcFile = "file:///home/hduser/data/spark/resources/users.orc"

# 读取ORC文件，构造DataFrame
orc_df = spark.read.format("orc").load(orcFile)
# orc_df = spark.read.load(orcFile,format="orc")
# orc_df = spark.read.orc(orcFile)    # 简洁写法

orc_df.printSchema()
orc_df.show()
```

```
root
 |-- name: string (nullable = true)
 |-- favorite_color: string (nullable = true)
 |-- favorite_numbers: array (nullable = true)
 |    |-- element: integer (containsNull = true)

+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

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。

4.6.6 使用 JDBC 从数据库创建 DataFrame

PySpark 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。

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。

```
jdbc_df1 = spark.read \
    .format('jdbc') \
    .options(url = 'jdbc:mysql://localhost:3306/xueai8',
             dbtable = 'peoples',
             user = 'root',
             password = 'admin') \
    .load()

# 获取表数据作为一个 DataFrame
jdbc_df1.show()
```

执行过程如下所示：

```
jdbc_df1 = spark.read \
    .format('jdbc') \
    .options(url = 'jdbc:mysql://localhost:3306/xueai8',
             dbtable = 'peoples',
             user = 'root',
             password = 'admin') \
    .load()

# 获取表数据作为一个DataFrame
jdbc_df1.show()
```

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  1|  张三| 23|
|  2|  李四| 18|
|  3|王老五| 35|
+---+------+---+
```

也可以使用快捷方法 jdbc(url, table, properties={"user": "username", "password": "password"})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：

```
jdbc_df3 = spark.read \
```

```
    .jdbc("jdbc:mysql://localhost:3306/xueai8",
          "peoples",
          properties={"user": "root", "password": "admin"})

# 获取表数据作为一个 DataFrame
jdbc_df3.show()
```

执行过程如下所示：

```
jdbc_df3 = spark.read \
    .jdbc("jdbc:mysql://localhost:3306/xueai8",
          "peoples",
          properties={"user": "root", "password": "admin"})

# 获取表数据作为一个DataFrame
jdbc_df3.show()
```

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  1|  张三| 23|
|  2|  李四| 18|
|  3|王老五| 35|
+---+------+---+
```

在读取 JDBC 关系型数据库中的表数据时，也可以指定相应的 DataFrame 的列数据类型。

```
# 指定读取时的 dataframe 列数据类型
jdbc_df2 = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:mysql://localhost:3306/xueai8") \
    .option("dbtable", "peoples") \
    .option("user", "root") \
    .option("password", "admin") \
    .option("customSchema", "id DECIMAL(38, 0), name STRING, age LONG") \
    .load()

jdbc_df2.printSchema()
jdbc_df2.show()
```

执行过程如下所示：

```
# 指定读取时的dataframe列数据类型
jdbc_df2 = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:mysql://localhost:3306/xueai8") \
    .option("dbtable", "peoples") \
    .option("user", "root") \
    .option("password", "admin") \
    .option("customSchema", "id DECIMAL(38, 0), name STRING, age LONG") \
    .load()

jdbc_df2.printSchema()
jdbc_df2.show()
```

```
root
 |-- id: decimal(38,0) (nullable = true)
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)

+---+------+---+
| id|  name|age|
+---+------+---+
|  1|  张三| 23|
|  2|  李四| 18|
|  3|王老五| 35|
+---+------+---+
```

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：

```
jdbc_df4 = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:mysql://localhost:3306/xueai8") \
    .option("query", "select name,age from peoples") \
    .option("user", "root") \
    .option("password", "admin") \
    .load()

jdbc_df4.printSchema()
jdbc_df4.show()
```

执行过程如下所示：

```
jdbc_df4 = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:mysql://localhost:3306/xueai8") \
    .option("query", "select name,age from peoples") \
    .option("user", "root") \
    .option("password", "admin") \
    .load()

jdbc_df4.printSchema()
jdbc_df4.show()
```

```
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)

+------+---+
|  name|age|
+------+---+
|  张三| 23|
|  李四| 18|
|王老五| 35|
+------+---+
```

4.6.7 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 PySpark SQL 数据源 API，用于将图像数据作为 DataFrame 加载。

在下面的示例中，我们读取 images 目录下的所有图像到 DataFrame 中。

```
image_file = "file:///home/hduser/data/spark/resources/images/*"
image_df = spark.read.format("image").option("dropInvalid", "true").load(image_file)

image_df.printSchema()
image_df \
    .select("image.origin","image.width","image.height","image.nChannels","image.mode") \
    .show(truncate=False)
```

执行过程如下所示：

```
image_file = "file:///home/hduser/data/spark/resources/images/*"
image_df = spark.read.format("image").option("dropInvalid", "true").load(image_file)

image_df.printSchema()
image_df \
    .select("image.origin","image.width","image.height","image.nChannels","image.mode") \
    .show(truncate=False)
```

```
root
 |-- image: struct (nullable = true)
 |    |-- origin: string (nullable = true)
 |    |-- height: integer (nullable = true)
 |    |-- width: integer (nullable = true)
 |    |-- nChannels: integer (nullable = true)
 |    |-- mode: integer (nullable = true)
 |    |-- data: binary (nullable = true)
```

```
+------------------------------------------------------------------------------+-----+------+---------+----+
|origin                                                                        |width|height|nChannels|mode|
+------------------------------------------------------------------------------+-----+------+---------+----+
|file:///home/hduser/data/spark/resources/images/white_lines.jpg               |2436 |1366  |3        |16  |
|file:///home/hduser/data/spark/resources/images/bridge_trees_example.jpg      |1280 |720   |3        |16  |
|file:///home/hduser/data/spark/resources/images/udacity_sdc.png               |320  |213   |4        |24  |
|file:///home/hduser/data/spark/resources/images/curved_lane.jpg               |1280 |720   |3        |16  |
+------------------------------------------------------------------------------+-----+------+---------+----+
```

从上面的示例中，我们可以看出，使用图像数据源，我们可以从目录加载图像文件，加载的 DataFrame 有一个 StructType 列:“image”，其中包含作为 image schema 存储的图像数据。该 image 列的 schema 是：

- origin：StringType（代表该头像的文件路径）
- height：IntegerType（图像的高度）
- width：IntegerType（图像的宽度）
- nChannels：IntegerType（图像通道的数量）
- mode：IntegerType（OpenCV 兼容的类型）
- data：BinaryType（以 opencv 兼容顺序的图像字节：在大多数情况下按行排列的 BGR）

关于 nChannels，指的是颜色通道的数量。典型值是 1 为灰度图像，3 为彩色图像(例如，RGB)， 4 为带有 alpha 通道的彩色图像。

关于 mode：整数标志，提供如何解释数据字段的信息。它指定数据存储的数据类型和通道顺序。该字段的值被期望(但不是强制的)映射到下面显示的 OpenCV 类型之一。OpenCV 类型定义为 1、2、3 或 4 个通道和像素值的一些数据类型。

OpenCV 中类型到数字的映射(数据类型 x 通道数量)：

Type	C1	C2	C3	C4
CV_8U	0	8	16	24
CV_8S	1	9	17	25
CV_16U	2	10	18	26
CV_16S	3	11	19	27
CV_32S	4	12	20	28
CV_32S	5	13	21	29
CV_64F	6	14	22	30

关于 data：以二进制格式存储的图像数据。图像数据表示为一个三维数组，其维度形状(高度、宽度、nChannels)和类型为 t 的数组值由 mode 字段指定。数组按行主顺序存储。

对于可加载的图像，Spark 规范明确针对当前行业中最常见的用例：二进制、int32、int64、浮点或双数据的多通道矩阵，可以轻松地放入 JVM 堆中。以下是一些限制：

- 图像的总大小应该限制在 2GB 以下(大致)。
- 颜色通道的含义是特定于应用程序的，不受标准的约束(与 OpenCV 标准一致)。
- 不支持气象学、医学等领域使用的专门格式。
- 这种格式专门用于图像，并不试图解决 Spark 中表示 n 维张量的更一般的问题。

4.6.8 读取 Avro 文件创建 DataFrame

从 Apache Spark 2.4 版本开始，PySpark 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。

因为 spark-avro 模块是外部的，所以 DataFrameReader 或 DataFrameWriter 中没有.avro API。因此，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。

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
avro_file = "file:///home/hduser/data/spark/resources/users.avro"

# 读取 Avro 文件，创建 DataFrame
users_df = spark.read.format("avro").load(avro_file)

users_df.printSchema()
users_df.show()
```

执行过程如下所示：

```
# 数据源文件
avro_file = "file:///home/hduser/data/spark/resources/users.avro"

# 读取Avro文件，创建DataFrame
users_df = spark.read.format("avro").load(avro_file)

users_df.printSchema()
users_df.show()
```

```
root
 |-- name: string (nullable = true)
 |-- favorite_color: string (nullable = true)
 |-- favorite_numbers: array (nullable = true)
 |    |-- element: integer (containsNull = true)

+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

Avro 的数据源选项可以使用 DataFrameReader 或 DataFrameWriter 上的.option 方法来设置。

属性名	默认值	含义	应用范围
avroSchema	None	用户以JSON格式提供的可选Avro模式。记录字段的数据类型和命名在从Avro读取时应与Avro数据类型匹配，在写入Avro文件时应与Spark的内部数据类型(例如StringType、IntegerType)匹配;否则，读/写操作将失败。	read write
recordName	topLevelRecord	写入结果中的顶级记录名，这在Avro规范中是必需的。	write
recordNamespace	""	在写入结果中的记录命名空间。	write
ignoreExtension	True	该选项控制在read中忽略没有.avro扩展名的文件。 如果启用了该选项，则加载所有文件(有和没有.avro扩展名)。	read
compression	snappy	compression选项允许指定写入中使用的压缩编解码器。目前支持的编解码器有uncompressed, snappy, deflate, bzip2 和xz。 如果未设置此选项，则会考虑配置 spark.sql.avro.compression.codec配置项。	write

Avro 的配置可以在 SparkSession 上使用 setConf 方法，或者使用 SQL 运行 SET key=value 命令。

属性名	默认值	含义
spark.sql.legacy.replaceDatabricksSparkAvro.enabled	True	如果将其设置为true，则数据源提供程序com.databricks.spark.avro被映射到内置但外部的avro数据源模块，以便向后兼容。
spark.sql.avro.compression.codec	snappy	用于写AVRO文件的压缩编解码器。支持的编解码器：uncompressed, deflate, snappy, bzip2 和xz。默认的编解码器是snappy。
spark.sql.avro.deflate.level	-1	写AVRO文件中使用的deflate codec的压缩水平（级别）。有效值必须在1到9(含9)或-1之间。默认值为-1，对应于当前实现中的6级。

4.7 操作 DataFrame

在 Spark 2.0 中，DataFrame 为结构化数据操作提供了一种特定于领域的语言（DSL）。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 引用列的多种方式

在学习操作 DataFrame 之前，首先需要掌握 PySpark 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。

首先，构造一个 DataFrame。

```
# 将元组转为 DataFrame
kvDF = spark.createDataFrame([(1,2),(3,4)],["key","value"])

kvDF.printSchema()
kvDF.show()
```

执行过程如下所示：

```
# 将元组转为DataFrame
kvDF = spark.createDataFrame([(1,2),(3,4)],["key","value"])

kvDF.printSchema()
kvDF.show()
```

```
root
 |-- key: long (nullable = true)
 |-- value: long (nullable = true)

+---+-----+
|key|value|
+---+-----+
|  1|    2|
|  3|    4|
+---+-----+
```

调用 columns 函数，可以获得一个 DataFrame 所有列名的列表。

```
kvDF.columns
```

执行过程如下所示：

```
kvDF.columns
```

```
['key', 'value']
```

如果我们仅仅是为了获取列值，而不对列值做任何计算和比较，则直接引用列名字符串即可。

```
kvDF.select("key").show()                    # 列为字符串类型
```

执行过程如下所示：

```
kvDF.select("key").show()                    # 列为字符串类型
```

```
+---+
|key|
+---+
|  1|
|  3|
+---+
```

如果要对列做任何计算和比较等操作，则需要获得列的对象类型，这需要使用内置的 col 函数来选择列（这时获得的是一个 pyspark.sql.Column 对象）。

```
from pyspark.sql.functions import col
```

```
kvDF.select(col("key")).show()                # 列为 Column 类型
kvDF.where(col("key") > 1).show()             # 列为 Column 类型
```

执行过程如下所示：

```
from pyspark.sql.functions import col


kvDF.select(col("key")).show()            # 列为Column类型
kvDF.where(col("key") > 1).show()         # 列为Column类型
```

```
+---+
|key|
+---+
|  1|
|  3|
+---+

+---+-----+
|key|value|
+---+-----+
|  3|    4|
+---+-----+
```

也可以使用 expr 函数，它与 col 函数相同，返回一个 pyspark.sql.Column 对象，不过它允许使用表达式作为参数。

```
from pyspark.sql.functions import expr

kvDF.select(expr("key")).show()
kvDF.select(expr("key > 1")).show()
kvDF.select(expr("key") > 1).show()
```

执行过程如下所示：

```
from pyspark.sql.functions import expr

kvDF.select(expr("key")).show()
kvDF.select(expr("key > 1")).show()
kvDF.select(expr("key") > 1).show()
```

```
+---+
|key|
+---+
|  1|
|  3|
+---+

+---------+
|(key > 1)|
+---------+
|    false|
|     true|
+---------+

+---------+
|(key > 1)|
+---------+
|    false|
|     true|
+---------+
```

也可以直接通过 DataFrame 来引用。

```
kvDF.select(kvDF.key).show()
kvDF.select(kvDF["key"]).show()
```

执行过程如下所示：

```
kvDF.select(kvDF.key).show()
kvDF.select(kvDF["key"]).show()
```

```
+---+
|key|
+---+
|  1|
|  3|
+---+

+---+
|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。

```
from pyspark.sql.functions import col

kvDF.select(col("key"), col("key") > 1).show()
```

执行过程如下所示：

```
from pyspark.sql.functions import col

kvDF.select(col("key"), col("key") > 1).show()
```

```
+---+---------+
|key|(key > 1)|
+---+---------+
|  1|    false|
|  3|     true|
+---+---------+
```

也可以给列取一个别名，代码如下所示：

```
kvDF.select(col("key"), (col("key") > 1).alias("a")).show()

# 或
kvDF.select(kvDF.key, (kvDF.key > 1).alias("a")).show()
```

执行过程如下所示：

```
kvDF.select(col("key"), (col("key") > 1).alias("a")).show()

# 或
kvDF.select(kvDF.key, (kvDF.key > 1).alias("a")).show()
```

```
+---+-----+
|key|    a|
+---+-----+
|  1|false|
|  3| true|
+---+-----+

+---+-----+
|key|    a|
+---+-----+
|  1|false|
|  3| true|
+---+-----+
```

也可以使用正则表达式来选择列。

```
kvDF.select(kvDF.colRegex("`^k.*`")).show()
```

执行过程如下所示：

```
kvDF.select(kvDF.colRegex("`^k.*`")).show()
```

```
+---+
|key|
+---+
|  1|
|  3|
+---+
```

4.7.2 对 DataFrame 进行转换操作

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）

在进一步演示 DataFrame 的各种操作方法之前，我们先准备好要用到的数据。这里我们将使用一个电影数据集 movies.csv，它位于 PBLP 平台的~/data/spark/movies2/目录下。

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
file = "file:///home/hduser/data/spark/movies2/movies.csv"
movies = spark.read \
        .option("header","true") \
        .option("inferSchema","true") \
        .csv(file)

movies.printSchema()
movies.show(5)
```

执行过程如下所示：

```
# 加载电影数据集文件到DataFrame中
file = "file:///home/hduser/data/spark/movies2/movies.csv"
movies = spark.read \
      .option("header","true") \
      .option("inferSchema","true") \
      .csv(file)

movies.printSchema()
movies.show(5)
```

```
root
 |-- actor: string (nullable = true)
 |-- title: string (nullable = true)
 |-- year: integer (nullable = true)

+-----------------+-------------+----+
|            actor|        title|year|
+-----------------+-------------+----+
|McClure, Marc (I)|Freaky Friday|2003|
|McClure, Marc (I)| Coach Carter|2005|
|McClure, Marc (I)|  Superman II|1980|
|McClure, Marc (I)|    Apollo 13|1995|
|McClure, Marc (I)|     Superman|1978|
+-----------------+-------------+----+
only showing top 5 rows
```

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 title 和 year 两列，返回一个新的 DataFrame：

```
movies.select("title","year").show(5)
```

执行过程如下所示：

```
movies.select("title","year").show(5)
```

```
+--------------+----+
|         title|year|
+--------------+----+
|Freaky Friday|2003|
| Coach Carter|2005|
|  Superman II|1980|
|    Apollo 13|1995|
|     Superman|1978|
+--------------+----+
only showing top 5 rows
```

下面的代码将电影上演的年份转换到年代表示，使用 col 函数，并赋予一个别名：

```
# 使用将年份列转换到年代列
movies2 = movies.select(col('title') ,(col('year') - col('year') % 10).alias("decade"))
movies2.show(5)
```

执行过程如下所示：

```
# 使用将年份列转换到年代列
movies2 = movies.select(col('title') ,(col('year') - col('year') % 10).alias("decade"))
movies2.show(5)
```

```
+--------------+------+
|         title|decade|
+--------------+------+
|Freaky Friday|  2000|
| Coach Carter|  2000|
|  Superman II|  1980|
|    Apollo 13|  1990|
|     Superman|  1970|
+--------------+------+
only showing top 5 rows
```

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)
```

执行过程如下所示：

```
movies.selectExpr("*","(year - (year % 10)) as decade").show(5)
```

```
+-----------------+--------------+----+------+
|            actor|         title|year|decade|
+-----------------+--------------+----+------+
|McClure, Marc (I)|Freaky Friday|2003|  2000|
|McClure, Marc (I)| Coach Carter|2005|  2000|
|McClure, Marc (I)|  Superman II|1980|  1980|
|McClure, Marc (I)|    Apollo 13|1995|  1990|
|McClure, Marc (I)|     Superman|1978|  1970|
+-----------------+--------------+----+------+
only showing top 5 rows
```

在 selectExpr()方法中，不但支持使用 SQL 表达式，还支持直接使用 SQL 内置函数。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()
```

执行过程如下所示：

```
movies.selectExpr("count(distinct(title)) as movies",
                  "count(distinct(actor)) as actors").show()
```

```
+------+------+
|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：

```
movies.filter('year < 2000').show(5)
# movies.where('year < 2000').show(5)    # 等价
```

执行过程如下所示：

```
movies.filter('year < 2000').show(5)
# movies.where('year < 2000').show(5)   # 等价
```

```
+------------------+--------------------+----+
|             actor|               title|year|
+------------------+--------------------+----+
|McClure, Marc (I)|         Superman II|1980|
|McClure, Marc (I)|           Apollo 13|1995|
|McClure, Marc (I)|            Superman|1978|
|McClure, Marc (I)|  Back to the Future|1985|
|McClure, Marc (I)|Back to the Futur...|1990|
+------------------+--------------------+----+
only showing top 5 rows
```

找出 2000 年及以后上映的电影，代码如下：

```
movies.filter('year >= 2000').show(5)
# movies.where('year >= 2000').show(5)   # 等价
```

执行过程如下所示：

```
movies.filter('year >= 2000').show(5)
# movies.where('year >= 2000').show(5)  # 等价
```

```
+------------------+--------------------+----+
|             actor|               title|year|
+------------------+--------------------+----+
|McClure, Marc (I)|       Freaky Friday|2003|
|McClure, Marc (I)|        Coach Carter|2005|
|Cooper, Chris (I)|  Me, Myself & Irene|2000|
|Cooper, Chris (I)|              Capote|2005|
|Cooper, Chris (I)|The Bourne Supremacy|2004|
+------------------+--------------------+----+
only showing top 5 rows
```

找出 2000 年上映的电影，代码如下：

```
# 相等比较
movies.filter('year = 2000').show(5)
```

```
# movies.where('year = 2000').show(5)  # 等价
```

执行过程如下所示：

```
# 相等比较
movies.filter('year = 2000').show(5)
# movies.where('year = 2000').show(5)  # 等价
```

```
+-----------------+--------------------+----+
|            actor|               title|year|
+-----------------+--------------------+----+
|Cooper, Chris (I)|  Me, Myself & Irene|2000|
|Cooper, Chris (I)|         The Patriot|2000|
|  Jolie, Angelina|Gone in Sixty Sec...|2000|
|   Yip, Fran$oise|      Romeo Must Die|2000|
|   Danner, Blythe|    Meet the Parents|2000|
+-----------------+--------------------+----+
only showing top 5 rows
```

不等比较使用的操作符是!=。找出非 2000 年上映的电影，代码如下：

```
# 不等比较使用的操作符是 !=
movies.select("title","year").filter('year != 2000').show(5)
# movies.select("title","year").where('year != 2000').show(5)  # 等价
# movies.select("title","year").filter(col('year') != 2000).show(5)  # 等价
```

执行过程如下所示：

```
# 不等比较使用的操作符是 !=
movies.select("title","year").filter('year != 2000').show(5)
# movies.select("title","year").where('year != 2000').show(5)  # 等价
# movies.select("title","year").filter(col('year') != 2000).show(5)  # 等价
```

```
+-------------+----+
|        title|year|
+-------------+----+
|Freaky Friday|2003|
| Coach Carter|2005|
|  Superman II|1980|
|    Apollo 13|1995|
|     Superman|1978|
+-------------+----+
only showing top 5 rows
```

支持离散值匹配。例如，找出 2001 年到 2002 年间上映的电影，实现代码如下：

```
movies.filter(col("year").isin([2001,2002])).show(5, truncate=False)
# movies.where(col("year").isin([2001,2002])).show(5, truncate=False)  # 等价
```

执行过程如下所示：

```
movies.filter(col("year").isin([2001,2002])).show(5, truncate=False)
# movies.where(col("year").isin([2001,2002])).show(5, truncate=False)  # 等价
```

```
+--------------------+-----------------------------------------+----+
|actor               |title                                    |year|
+--------------------+-----------------------------------------+----+
|Cooper, Chris (I)   |The Bourne Identity                      |2002|
|Cassavetes, Frank   |John Q                                   |2002|
|Knight, Shirley (I)|Divine Secrets of the Ya-Ya Sisterhood|2002|
|Jolie, Angelina     |Lara Croft: Tomb Raider                  |2001|
|Cueto, Esteban      |Collateral Damage                        |2002|
+--------------------+-----------------------------------------+----+
only showing top 5 rows
```

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。

```
movies.filter('year >= 2000' and length('title') < 5).show(5)
# movies.where('year >= 2000' and length('title') < 5).show(5)    # 等价
```

执行过程如下所示：

```
movies.filter('year >= 2000' and length('title') < 5).show(5)
# movies.where('year >= 2000' and length('title') < 5).show(5)   # 等价
```

```
+----------------+-----+----+
|           actor|title|year|
+----------------+-----+----+
|Jolie, Angelina| Salt|2010|
| Cueto, Esteban|  xXx|2002|
|  Butters, Mike|  Saw|2004|
| Franko, Victor|   21|2008|
|  Ogbonna, Chuk| Salt|2010|
+----------------+-----+----+
only showing top 5 rows
```

另一种实现相同结果的方法是调用 filter 或 where 函数两次：

```
movies.filter('year >= 2000').filter(length('title') < 5).show(5)
# movies.where('year >= 2000').where(length('title') < 5).show(5)    # 等价
```

执行过程如下所示：

```
movies.filter('year >= 2000').filter(length('title') < 5).show(5)
# movies.where('year >= 2000').where(length('title') < 5).show(5)   # 等价
```

```
+----------------+-----+----+
|           actor|title|year|
+----------------+-----+----+
|Jolie, Angelina| Salt|2010|
| Cueto, Esteban|  xXx|2002|
|  Butters, Mike|  Saw|2004|
| Franko, Victor|   21|2008|
|  Ogbonna, Chuk| Salt|2010|
+----------------+-----+----+
only showing top 5 rows
```

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：

```
movies.select("title").distinct().selectExpr("count(title) as movies").show()
```

```
movies.dropDuplicates(["title"]).selectExpr("count(title) as movies").show()    # 与上句等价
```

执行过程如下所示：

```
movies.select("title").distinct().selectExpr("count(title) as movies").show()
movies.dropDuplicates(["title"]).selectExpr("count(title) as movies").show()
```

```
+------+
|movies|
+------+
|  1409|
+------+

+------+
|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，然后按电影名称长度排序显示，默认是升序：

```
movieTitles = movies.dropDuplicates(["title"]).selectExpr("title", "length(title) as title_length", "year")
movieTitles.sort('title_length').show(15)                                    # 默认是升序
```

执行过程如下所示：

```
movieTitles = movies.dropDuplicates(["title"]) \
                    .selectExpr("title", "length(title) as title_length", "year")
movieTitles.sort('title_length').show(15)                                    # 默认是升序
```

```
+-----+------------+----+
|title|title_length|year|
+-----+------------+----+
|   X2|           2|2003|
|   Up|           2|2009|
|   12|           2|2007|
|   21|           2|2008|
|   RV|           2|2006|
|  Saw|           3|2004|
|  Rio|           3|2011|
|  Elf|           3|2003|
|  Hop|           3|2011|
|  JFK|           3|1991|
|  xXx|           3|2002|
|  300|           3|2006|
|  Ali|           3|2001|
| Antz|           4|1998|
| Jack|           4|1996|
+-----+------------+----+
only showing top 15 rows
```

按电影名称长度排序降序显示：

```
movieTitles.sort(col('title_length').desc()).show(15,False)          # 降序
# movieTitles.sort(desc('title_length')).show(15,False)              # 同上，降序
```

执行过程如下所示：

```
movieTitles.sort(col('title_length').desc()).show(15,False)      # 降序
# movieTitles.sort(desc('title_length')).show(15,False)          # 同上，降序
```

```
+--------------------------------------------------------------------------+------------+----+
|title                                                                     |title_length|year|
+--------------------------------------------------------------------------+------------+----+
|Borat: Cultural Learnings of America for Make Benefit Glorious Nation of Kazakhstan|83          |2006|
|The Chronicles of Narnia: The Lion, the Witch and the Wardrobe            |62          |2005|
|Hannah Montana & Miley Cyrus: Best of Both Worlds Concert                 |57          |2008|
|Istoriya pro Richarda, milorda i prekrasnuyu Zhar-ptitsu                  |56          |1997|
|The Chronicles of Narnia: The Voyage of the Dawn Treader                  |56          |2010|
|Pirates of the Caribbean: The Curse of the Black Pearl                    |54          |2003|
|Indiana Jones and the Kingdom of the Crystal Skull                        |50          |2008|
|Percy Jackson & the Olympians: The Lightning Thief                        |50          |2010|
|Interview with the Vampire: The Vampire Chronicles                        |50          |1994|
|The Lord of the Rings: The Fellowship of the Ring                         |49          |2001|
|Wallace & Gromit in The Curse of the Were-Rabbit                          |48          |2005|
|Master and Commander: The Far Side of the World                           |47          |2003|
|Lemony Snicket's A Series of Unfortunate Events                           |47          |2004|
|Magnificent Desolation: Walking on the Moon 3D                            |46          |2005|
|Sweeney Todd: The Demon Barber of Fleet Street                            |46          |2007|
+--------------------------------------------------------------------------+------------+----+
only showing top 15 rows
```

orderBy 函数与 sort()函数等价。例如：

```
movieTitles.orderBy(col('title_length').desc()).show(5, truncate=False)
```

执行过程如下所示：

```
movieTitles.orderBy(col('title_length').desc()).show(5, truncate=False)
```

```
+--------------------------------------------------------------------------+------------+----+
|title                                                                     |title_length|year|
+--------------------------------------------------------------------------+------------+----+
|Borat: Cultural Learnings of America for Make Benefit Glorious Nation of Kazakhstan|83          |2006|
|The Chronicles of Narnia: The Lion, the Witch and the Wardrobe            |62          |2005|
|Hannah Montana & Miley Cyrus: Best of Both Worlds Concert                 |57          |2008|
|The Chronicles of Narnia: The Voyage of the Dawn Treader                  |56          |2010|
|Istoriya pro Richarda, milorda i prekrasnuyu Zhar-ptitsu                  |56          |1997|
+--------------------------------------------------------------------------+------------+----+
only showing top 5 rows
```

也可以按不同的顺序对两列或多列进行排序。例如，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，实现代码如下：

```
movieTitles.orderBy(col('title_length').desc(), 'year').show(10, truncate=False)
```

执行过程如下所示：

```
movieTitles.orderBy(col('title_length').desc(), 'year').show(10, truncate=False)
```

```
+--------------------------------------------------------------------------+------------+----+
|title                                                                     |title_length|year|
+--------------------------------------------------------------------------+------------+----+
|Borat: Cultural Learnings of America for Make Benefit Glorious Nation of Kazakhstan|83          |2006|
|The Chronicles of Narnia: The Lion, the Witch and the Wardrobe            |62          |2005|
|Hannah Montana & Miley Cyrus: Best of Both Worlds Concert                 |57          |2008|
|Istoriya pro Richarda, milorda i prekrasnuyu Zhar-ptitsu                  |56          |1997|
|The Chronicles of Narnia: The Voyage of the Dawn Treader                  |56          |2010|
|Pirates of the Caribbean: The Curse of the Black Pearl                    |54          |2003|
|Interview with the Vampire: The Vampire Chronicles                        |50          |1994|
|Indiana Jones and the Kingdom of the Crystal Skull                        |50          |2008|
|Percy Jackson & the Olympians: The Lightning Thief                        |50          |2010|
|The Lord of the Rings: The Fellowship of the Ring                         |49          |2001|
+--------------------------------------------------------------------------+------------+----+
only showing top 10 rows
```

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。

```
movies.groupBy("year").count().orderBy(col("count").desc()).show(5)
```

执行过程如下所示：

```
movies.groupBy("year").count().orderBy(col("count").desc()).show(5)
```

```
+----+-----+
|year|count|
+----+-----+
|2006| 2078|
|2004| 2005|
|2007| 1986|
|2005| 1960|
|2011| 1926|
+----+-----+
only showing top 5 rows
```

统计上映电影数量超过 2000 部的年份。

```
movies.groupBy("year").count().where("count > 2000").show()
```

执行过程如下所示：

```
movies.groupBy("year").count().where("count > 2000").show()
```

```
+----+-----+
|year|count|
+----+-----+
|2006| 2078|
|2004| 2005|
+----+-----+
```

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：

```
# 首先创建一个带有演员名字及名字长度的 DataFrame，
# 再按长度对名字排序，并获得 top 10
actor_top_10 = movies \
        .select("actor") \
        .distinct() \
        .selectExpr("actor", "length(actor) as length") \
        .orderBy(col("length").desc()) \
        .limit(10)

actor_top_10.show(truncate=False)
```

执行过程如下所示：

```
# 首先创建一个带有演员名字及名字长度的DataFrame,
# 再按长度对名字排序，并获得top 10
actor_top_10 = movies \
      .select("actor") \
      .distinct() \
      .selectExpr("actor", "length(actor) as length") \
      .orderBy(col("length").desc()) \
      .limit(10)

actor_top_10.show(truncate=False)
```

```
+-----------------------------+------+
|actor                        |length|
+-----------------------------+------+
|Badalamenti II, Peter Donald|28    |
|Driscoll, Timothy 'TJ' James|28    |
|Phillips, Christopher (III) |27    |
|Marshall-Fricker, Charlotte |27    |
|Shepard, Maridean Mansfield |27    |
|Martino, Nicholas Alexander |27    |
|Pahlavi, Shah Mohammad Reza |27    |
|Lawrence, Mark Christopher  |26    |
|Van de Kamp Buchanan, Ryan  |26    |
|Lough Haggquist, Catherine  |26    |
+-----------------------------+------+
```

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，我们可以采用如下的方法：另外创建一个 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。这样就将缺失的演员添加到了原数据集中。

首先执行下面的代码，获得电影“12”的数据集：

```
shortNameMovieDF = movies.where('title == "12"')
shortNameMovieDF.show()
```

执行过程如下所示：

```
shortNameMovieDF = movies.where('title == "12"')
shortNameMovieDF.show()
```

```
+--------------------+-----+----+
|               actor|title|year|
+--------------------+-----+----+
|    Efremov, Mikhail|   12|2007|
|     Stoyanov, Yuriy|   12|2007|
|     Gazarov, Sergey|   12|2007|
|Verzhbitskiy, Viktor|   12|2007|
+--------------------+-----+----+
```

接下来，另外创建一个 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。

```
# 构造缺失演员信息到一个 DataFramme
forgottenActor = [("Brychta, Edita", "12", 2007)]
forgottenActorDF = spark.createDataFrame(forgottenActor,shortNameMovieDF.schema)
```

```
# 现在添加缺失的演员姓名
completeShortNameMovieDF = shortNameMovieDF.union(forgottenActorDF)
completeShortNameMovieDF.show()
```

执行过程如下所示：

```
# 构造缺失演员信息到一个DataFramme
forgottenActor = [("Brychta, Edita", "12", 2007)]
forgottenActorDF = spark.createDataFrame(forgottenActor,shortNameMovieDF.schema)

# 现在添加缺失的演员姓名
completeShortNameMovieDF = shortNameMovieDF.union(forgottenActorDF)
completeShortNameMovieDF.show()
```

```
+--------------------+-----+----+
|               actor|title|year|
+--------------------+-----+----+
|    Efremov, Mikhail|   12|2007|
|     Stoyanov, Yuriy|   12|2007|
|     Gazarov, Sergey|   12|2007|
|Verzhbitskiy, Viktor|   12|2007|
|      Brychta, Edita|   12|2007|
+--------------------+-----+----+
```

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“year”这个列的表达式，计算的结果是该电影上映的年代。

```
movies.withColumn("decade", (col('year') - col('year') % 10)).show(5)
```

执行过程如下所示：

```
movies.withColumn("decade", (col('year') - col('year') % 10)).show(5)
```

```
+-----------------+--------------+----+------+
|            actor|         title|year|decade|
+-----------------+--------------+----+------+
|McClure, Marc (I)|Freaky Friday|2003|  2000|
|McClure, Marc (I)| Coach Carter|2005|  2000|
|McClure, Marc (I)|  Superman II|1980|  1980|
|McClure, Marc (I)|    Apollo 13|1995|  1990|
|McClure, Marc (I)|     Superman|1978|  1970|
+-----------------+--------------+----+------+
only showing top 5 rows
```

如果传给 withColumn 函数的列名与现有列名相同，则意味着用新值替换旧值。执行下面的代码，替换“year”列的值为年代（原值为年份）：

```
# 现在用新值替换 year
movies.withColumn("year", (col('year') - col('year') % 10)).show(5)
```

执行过程如下所示：

```
# 现在用新值替换year
movies.withColumn("year", (col('year') - col('year') % 10)).show(5)
```

```
+------------------+--------------+----+
|             actor|         title|year|
+------------------+--------------+----+
|McClure, Marc (I)|Freaky Friday|2000|
|McClure, Marc (I)| Coach Carter|2000|
|McClure, Marc (I)|  Superman II|1980|
|McClure, Marc (I)|    Apollo 13|1990|
|McClure, Marc (I)|     Superman|1970|
+------------------+--------------+----+
only showing top 5 rows
```

10）withColumnRenamed(existingColName, newColName)：修改列名。

执行下面的代码，修改三个列的列名：

```
movies.withColumnRenamed("actor", "actor_name") \
      .withColumnRenamed("title", "movie_title") \
      .withColumnRenamed("year", "produced_year") \
      .show(5)
```

执行过程如下所示：

```
movies.withColumnRenamed("actor", "actor_name") \
      .withColumnRenamed("title", "movie_title") \
      .withColumnRenamed("year", "produced_year") \
      .show(5)
```

```
+------------------+--------------+-------------+
|       actor_name|  movie_title|produced_year|
+------------------+--------------+-------------+
|McClure, Marc (I)|Freaky Friday|         2003|
|McClure, Marc (I)| Coach Carter|         2005|
|McClure, Marc (I)|  Superman II|         1980|
|McClure, Marc (I)|    Apollo 13|         1995|
|McClure, Marc (I)|     Superman|         1978|
+------------------+--------------+-------------+
only showing top 5 rows
```

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)
```

执行过程如下所示：

```
movies.drop("actor", "me").printSchema()
movies.drop("actor", "me").show(5)
```

```
root
 |-- title: string (nullable = true)
 |-- year: integer (nullable = true)

+--------------+----+
|         title|year|
+--------------+----+
|Freaky Friday|2003|
| Coach Carter|2005|
|  Superman II|1980|
|    Apollo 13|1995|
|     Superman|1978|
+--------------+----+
only showing top 5 rows
```

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。

执行带有无放回和比例因子的抽样：

```
movies.sample(false, 0.0003).show(3)
```

执行带有有放回和比例因子、种子的抽样：

```
movies.sample(true, 0.0003, 123456).show(3)
```

执行过程如下所示：

```
# 带有无放回和fraction的抽样
movies.sample(False, 0.0003).show(3)

# 带有有放回和fraction、seed的抽样
movies.sample(True, 0.0003, 123456).show(3)
```

```
+-----------------+--------------------+----+
|            actor|               title|year|
+-----------------+--------------------+----+
|    Dilley, Leslie|      Pay It Forward|2000|
|Davis, Warwick (I)|Harry Potter and ...|2001|
|   Walsh, M. Emmet|My Best Friend's ...|1997|
+-----------------+--------------------+----+
only showing top 3 rows

+-------------+--------------------+----+
|       actor|               title|year|
+-------------+--------------------+----+
|Piddock, Jim|      The Prestige|2006|
|Reed, Tanoai|The Stepford Wives|2004|
|Moyo, Masasa|   Angels & Demons|2009|
+-------------+--------------------+----+
only showing top 3 rows
```

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：

```
# 权重需要是一个数组
smallerMovieDFs = movies.randomSplit([0.6, 0.3, 0.1])

# 看看各部分计数之和是否等于 1
t1 = movies.count()                    # 31393
print(t1)

# 分割后的第 1 个数据集的数量
t2 = smallerMovieDFs[0].count()        # 18881
print(t2)

# 3 个子数据集之和
t3 = smallerMovieDFs[0].count() + smallerMovieDFs[1].count() + smallerMovieDFs[2].count()     # 31393
print(t3)
```

执行过程如下所示：

```
# 权重需要是一个数组
smallerMovieDFs = movies.randomSplit([0.6, 0.3, 0.1])

# 看看各部分计数之和是否等于1
t1 = movies.count()                    # 31393
print(t1)

# 分割后的第1个数据集的数量
t2 = smallerMovieDFs[0].count()        # 18881
print(t2)

# 3个子数据集之和
t3 = smallerMovieDFs[0].count() + smallerMovieDFs[1].count() + smallerMovieDFs[2].count()    # 31393
print(t3)
```

```
31394
18774
31394
```

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,truncate=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)

# 返回一个包含数据集中所有行的数组
movies.collect()

# 返回数据集的列名，以列表形式
movies.columns                          # ['actor', 'title', 'year']
```

4.7.4 对 DataFrame 进行描述性统计操作

DataFrame 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：

```
descDF = df.describe()

descDF.printSchema()
descDF.show()
```

执行过程如下所示：

```
descDF = movies.describe()

descDF.printSchema()
descDF.show()
```

```
root
 |-- summary: string (nullable = true)
 |-- actor: string (nullable = true)
 |-- title: string (nullable = true)
 |-- year: string (nullable = true)

+-------+--------------------+--------------------+--------------------+
|summary|               actor|               title|                year|
+-------+--------------------+--------------------+--------------------+
|  count|               31394|               31394|               31393|
|   mean|                null|  312.61538461538464|2002.7964514382188|
| stddev|                null|   485.7043414390151| 6.377135379933117|
|    min|    Aaron, Caroline|'Crocodile' Dunde...|                1961|
|    max|von Sydow, Max (I)|                 xXx|                2012|
+-------+--------------------+--------------------+--------------------+
```

下面指定计算“year”列的基本统计信息：

```
descDF2 = movies.describe("year")

descDF2.printSchema()
descDF2.show()
```

执行过程如下所示：

```
descDF2 = movies.describe("year")

descDF2.printSchema()
descDF2.show()
```

```
root
 |-- summary: string (nullable = true)
 |-- year: string (nullable = true)

+-------+------------------+
|summary|              year|
+-------+------------------+
|  count|             31393|
|   mean|2002.7964514382188|
| stddev| 6.377135379933117|
|    min|              1961|
|    max|              2012|
+-------+------------------+
```

下面的代码计算“year”和“actor”这两列的基本统计信息：

```
descDF3 = movies.describe("year","actor")

descDF3.printSchema()
descDF3.show()
```

执行过程如下所示：

```
descDF3 = movies.describe("year","actor")

descDF3.printSchema()
descDF3.show()
```

```
root
 |-- summary: string (nullable = true)
 |-- year: string (nullable = true)
 |-- actor: string (nullable = true)

+-------+------------------+--------------------+
|summary|              year|               actor|
+-------+------------------+--------------------+
|  count|             31393|               31394|
|   mean|2002.7964514382188|                null|
| stddev| 6.377135379933117|                null|
|    min|              1961|    Aaron, Caroline|
|    max|              2012|von Sydow, Max (I)|
+-------+------------------+--------------------+
```

注：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。

PySpark 还提供了一个与 describe()函数类似的 summary()函数，用于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。

```
summaryDF = movies.summary()
```

```
summaryDF.printSchema()
summaryDF.show()
```

执行过程如下所示：

```
summaryDF = movies.summary()

summaryDF.printSchema()
summaryDF.show()
```

```
root
 |-- summary: string (nullable = true)
 |-- actor: string (nullable = true)
 |-- title: string (nullable = true)
 |-- year: string (nullable = true)

+-------+--------------------+--------------------+--------------------+
|summary|               actor|               title|                year|
+-------+--------------------+--------------------+--------------------+
|  count|               31394|               31394|               31393|
|   mean|                null|  312.61538461538464|2002.7964514382188|
| stddev|                null|   485.7043414390151| 6.377135379933117|
|    min|    Aaron, Caroline|'Crocodile' Dunde...|                1961|
|    25%|                null|                21.0|                1999|
|    50%|                null|                21.0|                2004|
|    75%|                null|               300.0|                2008|
|    max|von Sydow, Max (I)|                 xXx|                2012|
+-------+--------------------+--------------------+--------------------+
```

也可以指定想要的统计信息。如下面的代码所示：

```
summaryDF2 = movies.summary("count", "min", "25%", "75%", "max")

summaryDF2.printSchema()
summaryDF2.show()
```

执行过程如下所示：

```
summaryDF2 = movies.summary("count", "min", "25%", "75%", "max")

summaryDF2.printSchema()
summaryDF2.show()
```

```
root
 |-- summary: string (nullable = true)
 |-- actor: string (nullable = true)
 |-- title: string (nullable = true)
 |-- year: string (nullable = true)

+-------+--------------------+--------------------+-----+
|summary|               actor|               title| year|
+-------+--------------------+--------------------+-----+
|  count|               31394|               31394|31393|
|    min|    Aaron, Caroline|'Crocodile' Dunde...| 1961|
|    25%|                null|                21.0| 1999|
|    75%|                null|               300.0| 2008|
|    max|von Sydow, Max (I)|                 xXx| 2012|
+-------+--------------------+--------------------+-----+
```

如果要对指定的列做一个摘要，首先选择这些列，然后再执行 summary()方法。

```
summaryDF3 = movies.select("year").summary()

summaryDF3.printSchema()
summaryDF3.show()
```

执行过程如下所示：

```
summaryDF3 = movies.select("year").summary()

summaryDF3.printSchema()
summaryDF3.show()
```

```
root
 |-- summary: string (nullable = true)
 |-- year: string (nullable = true)

+-------+--------------------+
|summary|                year|
+-------+--------------------+
|  count|               31393|
|   mean|2002.7964514382188|
| stddev| 6.377135379933117|
|    min|                1961|
|    25%|                1999|
|    50%|                2004|
|    75%|                2008|
|    max|                2012|
+-------+--------------------+
```

4.7.5 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 位于 PBLP 平台的/home/hduser/data/spark/目录下，其内容如下：

```
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
```

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。

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# 构建 SparkSession 实例
```

```
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 定义文件路径
file = "file:///home/hduser/data/spark/employees.csv"

# 指定列名
columns = ["uname", "designation", "department", "jobtype", "NA", "NA2", "salary", "NA3"]

# 加载数据文件，并构造 DataFrame
df = spark.read \
        .option("inferSchema", "true") \
        .option("header","true") \
        .csv(file) \
        .toDF(*columns)

# 按 salary 列降序排列，并显示
df.orderBy(col("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|
+-----+---------------+----------+-------+----+----+--------+----+
only showing top 3 rows
```

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。而在一个典型的 ETL 数据处理作业中，处理结果通常需要被写到一些存储系统中，例如 HDFS 或 Hive。下面我们学习如何存储这些 DataFrame。

4.8.1 保存 DataFrame

在 PySpark SQL 中，pyspark.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。json 数据文件位于 PBLP 平台的/home/hduser/data/spark/resources/目录下。

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 读取 json 文件源，创建 DataFrame
file = "file:///home/hduser/data/spark/resources/people.json"
df = spark.read.json(file)

df.printSchema
df.show()

# 保存目录位置
output = "file:///home/hduser/data/spark/resources/people-csv-output"
df.where(col("age").isNotNull()).write.format("csv").save(output)
# df.where(col("age").isNotNull()).write.csv(output)                    # 等价上一句

# 将保存的 csv 数据再次加载到 RDD 中
textFile = spark.sparkContext.textFile(output)
for row in textFile.collect():
    print(row)
```

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。

PySpark 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。

```
from pyspark.sql.types import *
```

```
# 设置两条数据表示两个学生信息
students = [
    (3, "王老五", "F", 44),
    (4, "赵小虎", "M", 27)
]

# 指定一个 Schema(模式)
fields = [
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("gender", StringType(), True),
    StructField("age", IntegerType(), True)
]
schema = StructType(fields)

# 将元组转为 DataFrame
stusDF = spark.createDataFrame(students, schema=schema)

# stusDF.printSchema()
# stusDF.show()

# 下面创建一个 prop 变量用来保存 JDBC 连接参数
props = {
    "driver": "org.mariadb.jdbc.Driver",
    "user": "root",
    "password": "admin"
}

# 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 student 表中
url = "jdbc:mysql://localhost:3306/spark?useSSL=false"
stuDF.write.mode("append").jdbc(url=url, table="spark.students", properties=props)
```

这时再在 MySQL 数据库中执行如下查询：

```
mysql> select * from students;
```

可以看到，数据已经正确保存了：

```
MariaDB [spark]> select * from students;
+------+-----------+--------+------+
| id   | name      | gender | age  |
+------+-----------+--------+------+
|    1 | 张三      | F      |   23 |
|    2 | 李四      | M      |   18 |
|    3 | 王老五    | 男     |   46 |
|    4 | 赵小花    | 女     |   27 |
+------+-----------+--------+------+
4 rows in set (0.00 sec)
```

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 PySpark 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）：

Scala/Java	任何语言	说明
SaveMode.Append	"append"	当保存一个DataFrame到数据源时，如果数据/表已经存在的话，该DataFrame的内容将被追加到已经存储的数据。
SaveMode.Overwrite	"overwrite"	当保存一个DataFrame到数据源时，如果数据/表已经存在的话，已经存在的数据将被该DataFrame的内容所覆盖。
SaveMode.ErrorIfExists (默认)	"error"或"errorIfExists" (默认)	当保存一个DataFrame到数据源时，如果数据已经存在的话，将抛出一个异常。
SaveMode.Ignore	"ignore"	当保存一个DataFrame到数据源时，如果数据/表已经存在的话，save操作不会保存该DataFrame的内容，并且不会改变已经存在的数据。这类似于SQL中的create table if not exists。

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

PySpark SQL DataFrame API 在转换操作执行 shuffling 时增加分区。诸如 join()、union()这些 DataFrame 操作以及所有聚合函数操作会触发数据 shuffle。

在下面的示例中，请注意是如何查看一个 DataFrame 拥有的分区数量的。

```
from pyspark.sql import SparkSession

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

simpleData = [
      ("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)
]
schema = ["employee_name","department","city","salary","age","bonus"]
df = spark.createDataFrame(simpleData, schema=schema)

print("shuffle 前的分区数：",df.rdd.getNumPartitions())
```

```
# groupBy 操作会触发数据 shuffle
df2 = df.groupBy("city").count()

print("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)
print(df.groupBy("city").count().rdd.getNumPartitions())                # 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
file = "file:///home/hduser/data/spark/movies2/movies.csv"

movies = spark.read.option("header","true").option("inferSchema","true").csv(file)

movies2 = movies.repartition(4)                    # 将 DataFrame 重分区为 4 个分区

# 查看分区数
print("movies2 的分区数是：", movies2.rdd.getNumPartitions())    # movies2 的分区数是： 4

# 保存 movies2
movies2.write.csv("file:///home/hduser/data/spark/movies2/movies-out-partitions")
```

执行上面的代码，输出结果如下图所示：

```
[hduser@xueai8 movies-out-partitions]$ ll
总用量 1248
-rw-r--r-- 1 hduser hduser 318343 2月  11 09:50 part-00000-34dfa393-da58-40dc-b2c3-238030adc158-c000.csv
-rw-r--r-- 1 hduser hduser 317823 2月  11 09:50 part-00001-34dfa393-da58-40dc-b2c3-238030adc158-c000.csv
-rw-r--r-- 1 hduser hduser 317743 2月  11 09:50 part-00002-34dfa393-da58-40dc-b2c3-238030adc158-c000.csv
-rw-r--r-- 1 hduser hduser 318439 2月  11 09:50 part-00003-34dfa393-da58-40dc-b2c3-238030adc158-c000.csv
-rw-r--r-- 1 hduser hduser      0 2月  11 09:50 _SUCCESS
[hduser@xueai8 movies-out-partitions]$                    每个分区对应一个输出文件
```

在某些情况下，DataFrame 的内容并不大，并且需要写入到单个文件中。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.mode("overwrite").csv("file:///home/hduser/data/spark/movies2/movies-out-partitions")
```

再一次执行代码，这时输出结果文件数如下图所示：

```
[hduser@xueai8 movies-out-partitions]$ ll
总用量 1244
-rw-r--r-- 1 hduser hduser 1272348 2月  11 09:57 part-00000-da33373f-aa71-4ad3-a6b7-05a3e0076ce2-c000.csv
-rw-r--r-- 1 hduser hduser       0 2月  11 09:57 _SUCCESS
[hduser@xueai8 movies-out-partitions]$                    只有一个输出结果文件
```

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。

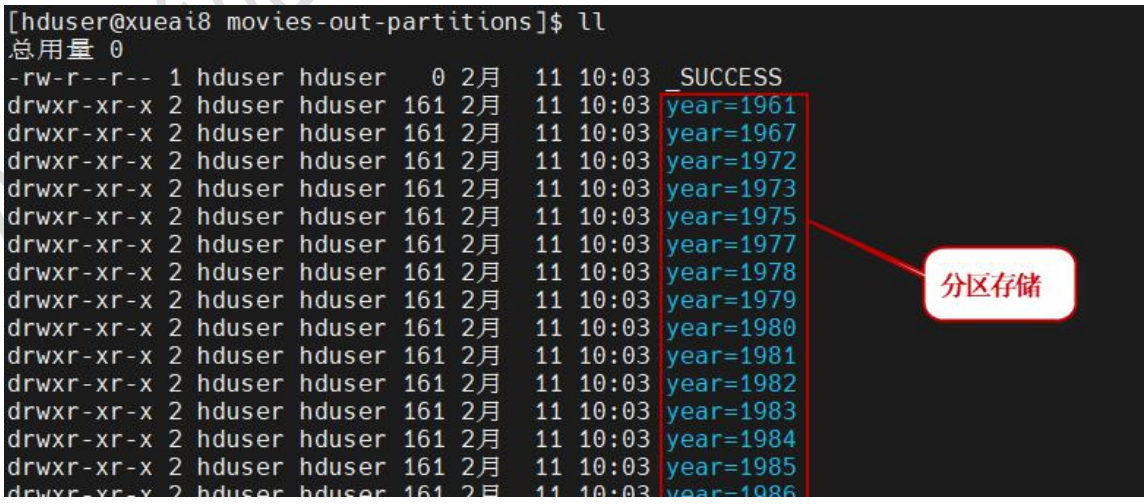
在 movies 这个 DataFrame 中，“year”列是最适合用来进行分区的候选列。假设我们要写出 movies 这个 DataFrame，用“year”列来分区。DataFrameWriter 将把所有具有相同年份的电影都写在同一个目录中。输出文件夹中的目录数量将对应于 movies 这个 DataFrame 中的年份的数量。

下面的示例中使用 partitionBy 函数来指定 DataFrame 输出时的分区数。

```
from pyspark.sql import SparkSession

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 加载电影数据集文件到 DataFrame 中
file = "file:///home/hduser/data/spark/movies2/movies.csv"

movies = spark.read \
        .option("header","true") \
        .option("inferSchema","true") \
        .csv(file)

# 查看分区数
print("movies 的分区数是：", movies.rdd.getNumPartitions())

# 指定输出路径
outpath = "file:///home/hduser/data/spark/movies2/movies-out-partitions"

# 输出 movies DataFrame，使用 Parquet 格式并按 year 列进行分区
movies.repartition("year").write.mode("overwrite").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，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。所以分区过少或过多，都可能是不利的：

- 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。

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？

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。

如何控制输出文件的大小？

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 临时视图与执行 SQL 查询

PySpark SQL 支持直接应用标准 SQL 语句进行查询。当在 PySpark 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。

4.9.1 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。

- PySpark 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。PySpark SQL 支持使用基本 SQL 语法或 HiveQL 编写的 SQL 查询的执行。

在 pyspark shell 或 Zeppelin notebook 中，会自动导入 spark.sql，所以可以直接使用该函数用来编写 SQL 命令。例如：

```
spark.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：

```
from pyspark.sql import SparkSession

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

infoDF = spark.sql("select current_date() as today , 1 + 100 as value")
infoDF.show()
```

执行过程如下所示：

```
from pyspark.sql import SparkSession

# 构建SparkSession实例
spark = SparkSession.builder \
   .master("spark://xueai8:7077") \
   .appName("pyspark sql demo") \
   .getOrCreate()
```

```
infoDF = spark.sql("select current_date() as today , 1 + 100 as value")
infoDF.show()
```

```
+----------+-----+
|     today|value|
+----------+-----+
|2022-02-11|  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，PySpark 也可以使用 SQL 语句直接查

询文件。例如：

```
sqlDF = spark.sql("SELECT * FROM parquet.`/data/spark/resources/users.parquet`")
sqlDF.show()
```

执行过程如下所示：

```
sqlDF = spark.sql("SELECT * FROM parquet.`/data/spark/resources/users.parquet`")
sqlDF.show()
```

```
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

注意，这种方式必须使用 HDFS 文件路径。如果是在 PBLP 平台下，因为配置了 Hive 集成，所以还需要先启动 Hive 的 MetaStore 服务：

```
$ hive --service metastore
```

4.9.2 注册临时视图并查询

DataFrame 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。

```
from pyspark.sql import SparkSession

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 定义文件路径
parquetFile = "file:///home/hduser/data/spark/movies2/movies.parquet"

# 读取到 DataFrame 中
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
sql_str = "select * from movies where actor_name like '%Jolie%' and produced_year>2009"
spark.sql(sql_str).show()
```

执行以上代码，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。

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 定义文件路径
parquetFile = "file:///home/hduser/data/spark/movies2/movies.parquet"

# 读取到 DataFrame 中
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 actor_name, count(*) as count from movies group by actor_name") \
      .where('count > 30') \
      .orderBy(col("count").desc()) \
      .show()
```

执行以上代码，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。

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# 构建 SparkSession 实例
spark = SparkSession.builder \
```

```
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 定义文件路径
parquetFile = "file:///home/hduser/data/spark/movies2/movies.parquet"

# 读取到 DataFrame 中
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 produced_year, count(*) as count
                from (select distinct movie_title, produced_year from movies)
                group by produced_year
            """) \
       .orderBy(col("count").desc()) \
       .show(5)
```

执行以上代码，输出结果如下所示：

```
+-------------+-----+
|produced_year|count|
+-------------+-----+
|         2011|   86|
|         2004|   86|
|         2006|   86|
|         2005|   85|
|         2008|   82|
+-------------+-----+
only showing top 5 rows
```

注：

PySpark 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，PySpark 2.0 通过包含一个新的 ANSI SQL 解析器扩展了 PySpark 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。

4.9.3 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。

PySpark 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。

```
from pyspark.sql import SparkSession

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 加载 JSON 文件数据到 DataFrame
json_file = "file:///home/hduser/data/spark/resources/people.json"
df = spark.read.json(json_file)

# 显示 DataFrame 内容
df.show()

# 查看 schema
df.printSchema()

# 将该 DataFrame 注册为一个全局临时视图 (注意，需要 hive --service metastore)
df.createGlobalTempView("people")

# 全局临时视图绑定到系统保存的数据库'global_temp'
spark.sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|    name|
# +----+-------+
# |null|Michael|
# |  30|    Andy|
# |  19| Justin|
# +----+-------+

# 全局临时视图是跨 session 会话的
spark.newSession().sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|    name|
# +----+-------+
# |null|Michael|
# |  30|    Andy|
# |  19| Justin|
# +----+-------+
```

4.9.4 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。

```
from pyspark.sql import SparkSession

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark 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 'file:///home/hduser/data/spark/resources/users.parquet')")

# 查询临时视图
spark.sql("select * from usersParquet").show()
```

执行过程如下所示：

```
# 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 'file:///home/hduser/data/spark/resources/users.parquet')")

# 查询临时视图
spark.sql("select * from usersParquet").show()
```

```
+------+--------------+----------------+
|  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。

```
from pyspark.sql import SparkSession

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 从 jdbc 数据源创建临时视图
spark.sql(
      """
         create temporary view peoplesjdbc
         using org.apache.spark.sql.jdbc
```

```
        options(
          url 'jdbc:mysql://localhost:3306/xueai8',
          dbtable 'peoples',
          user 'root',
          password 'admin'
        )
      """
    )

# 在临时视图上执行查询
spark.sql("select * from peoplesjdbc").show()
```

执行过程如下所示：

```
# 从jdbc数据源创建临时视图
spark.sql(
      """
        create temporary view peoplesjdbc
        using org.apache.spark.sql.jdbc
        options(
          url 'jdbc:mysql://localhost:3306/xueai8',
          dbtable 'peoples',
          user 'root',
          password 'admin'
        )
      """
    )

# 在临时视图上执行查询
spark.sql("select * from peoplesjdbc").show()
```

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  1|  张三| 23|
|  2|  李四| 18|
|  3|王老五| 35|
+---+------+---+
```

4.9.5 查看和管理表目录

当将 DataFrame 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，

```
from pyspark.sql import SparkSession

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark 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 'file:///home/hduser/data/spark/resources/users.parquet')")

# 查看当前数据库中注册的表有哪些
print(spark.catalog.listTables())

# 返回给定表/视图或临时视图的所有列
print(spark.catalog.listColumns("usersParquet"))
```

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()
```

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。

4.10 缓存 DataFrame

可以在内存中对 DataFrame 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 API（persist 和 unpersist）。然而，DataFrame 的缓存与 RDD 有很大的不同。PySpark 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 所需的空间比存储 RDD 所需的空间要少得多。

Spark 缓存和持久化是用于迭代和交互 Spark 应用程序的 DataFrame 优化技术，以提高作业的性能。PySpark SQL 提供了 cache()和 persist()方法用来缓存 DataFrame。

4.10.1 缓存方法

使用 cache()和 persist()方法，Spark 提供了一种优化机制来存储 PySpark 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。

```
from pyspark.sql.functions import col
from pyspark.sql import SparkSession

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 读取 csv 文件源，创建 DataFrame
file = "file:///home/hduser/data/spark/movies2/movies.csv"
```

```
df = spark.read \
        .options(inferSchema="true",delimiter=",",header="true") \
        .csv(file)

# 缓存
df2 = df.where(col("year") == "2012").cache()
df2.show(5, truncate = False)

print(f"2012 年上映的电影数量有{df2.count()}部。" )

df3 = df2.where(col("title") == "The Hunger Games")
print("电影饥饿游戏的主演是：" ,df3.collect()[0][0])
print("电影饥饿游戏的主演是：" ,df3.first()[0])
```

执行以上代码，输出内容如下：

```
+-----------------+-----------------+----+
|actor            |title            |year|
+-----------------+-----------------+----+
|Cassavetes, Frank|Battleship       |2012|
|Danner, Blythe   |The Lucky One    |2012|
|Manji, Rizwan    |The Dictator     |2012|
|Harrelson, Woody |The Hunger Games|2012|
|Hardy, Tom (I)   |This Means War   |2012|
+-----------------+-----------------+----+
only showing top 5 rows

2012年上映的电影数量有601部。
电影饥饿游戏的主演是： Harrelson, Woody
电影饥饿游戏的主演是： Harrelson, Woody
```

PySpark DataFrame 的 cache()方法内部调用了 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。

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。

4.10.2 缓存策略

PySpark 支持的所有不同存储级别指定如何以及在何处持久化或缓存 Spark DataFrame。

- 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 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：

```
dfPersist = dfPersist.unpersist()
```

4.10.3 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：

```
# 以可读的名字持久化一个 DataFrame
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()               # 1000
```

cache()和 persist()持久化是 lazy 执行的，而 cacheTable()是 eager 执行的。

4.11 PySpark SQL 可视化

PySpark 还没有任何绘图功能。如果想绘制一些内容，可以将数据从 SparkContext 中取出并放入“本地”Python 会话中，在那里可以使用 Python 的任意一个绘图库来处理它。

对于 PySpark SQL 中的 DataFrame，可以先将它转成 Pandas 的 DataFrame，再应用 Python 绘图库进行绘制。

4.11.1 PySpark DataFrame 转换到 Pandas

在 PySpark 中，很容易通过一行代码将 PySpark DataFrame 转换为 Pandas DataFrame：

```
df_pd = df.toPandas()
```

在下面的示例中，演示了如何将 PySpark DataFrame Row 对象列表转换为 Pandas DataFrame。

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import collect_list,struct
from pyspark.sql.types import ArrayType, StructField, StructType, StringType, IntegerType, DecimalType

from decimal import Decimal
import pandas as pd

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark rdd demo") \
    .getOrCreate()

# List
data = [ ('Category A', 1, Decimal(12.40)),
           ('Category B', 2, Decimal(30.10)),
           ('Category C', 3, Decimal(100.01)),
           ('Category A', 4, Decimal(110.01)),
           ('Category B', 5, Decimal(70.85))
         ]

# 创建 schema
schema = StructType([
      StructField('Category', StringType(), False),
      StructField('ItemID', IntegerType(), False),
      StructField('Amount', DecimalType(scale=2), True)
])

# 将 List 转换为 RDD
rdd = spark.sparkContext.parallelize(data)

# 创建 DataFrame
df = spark.createDataFrame(rdd, schema)
df.printSchema()
df.show()

# 将 PySpark DataFrame 转换为 Pandas DataFrame
df_pd = df.toPandas()
df_pd.info()

# 查看 Pandas DataFrame
print(df_pd)
```

执行以上代码，PySpark DataFrame 的输出内容如下：

```
root
 |-- Category: string (nullable = false)
 |-- ItemID: integer (nullable = false)
 |-- Amount: decimal(10,2) (nullable = true)

+----------+------+------+
|  Category|ItemID|Amount|
+----------+------+------+
|Category A|     1| 12.40|
|Category B|     2| 30.10|
|Category C|     3|100.01|
|Category A|     4|110.01|
|Category B|     5| 70.85|
+----------+------+------+
```

Pandas DataFrame 的信息输出如下：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Category  5 non-null      object
 1   ItemID    5 non-null      int32
 2   Amount    5 non-null      object
dtypes: int32(1), object(2)
memory usage: 228.0+ bytes
```

Pandas DataFrame 的内容输出如下：

```
    Category  ItemID  Amount
0  Category A       1   12.40
1  Category B       2   30.10
2  Category C       3  100.01
3  Category A       4  110.01
4  Category B       5   70.85
```

如果 PySpark SQL DataFrame 具有嵌套结构，该如何转换呢？

例如，在 PySpark 中进行聚合是很常见的操作。下面的代码片段按照 Category 属性对上面的 PySpark DataFrame 进行分组。

```
# 聚合但仍然保留所有原始属性
df_agg = df.groupby("Category").agg(collect_list(struct("*")).alias('Items'))
df_agg.printSchema()
df_agg.show(truncate=False)
```

执行以上代码，输出内容如下：

```
root
 |-- Category: string (nullable = false)
 |-- Items: array (nullable = false)
 |    |-- element: struct (containsNull = false)
 |    |    |-- Category: string (nullable = false)
 |    |    |-- ItemID: integer (nullable = false)
 |    |    |-- Amount: decimal(10,2) (nullable = true)

+----------+-----------------------------------------------------+
|Category  |Items                                                |
+----------+-----------------------------------------------------+
|Category C|[{Category C, 3, 100.01}]                            |
|Category B|[{Category B, 2, 30.10}, {Category B, 5, 70.85}]     |
|Category A|[{Category A, 1, 12.40}, {Category A, 4, 110.01}]    |
+----------+-----------------------------------------------------+
```

新的 Spark DataFrame 的 schema 有两个属性：Category 和 Items。这个 Items 属性是 pyspark.sql.Row 对象的数组或列表。要将这个 pyspark.sql.Row 列表转换为 Pandas DataFrame，我们可以使用 foreach 函数来转换 Items 属性。

```
def to_pandas(row):
    print(f'为类别{row["Category"]}创建一个 Pandas DataFrame')
    items = [item.asDict() for item in row["Items"]]
    df_pd_items = pd.DataFrame(items)
    print(df_pd_items)

# 将每个类别的 Items 转换为 Pandas DataFrame
for row in df_agg.collect():
    print(to_pandas(row))
```

在上面的代码片段中，首先将 Row list 转换为字典列表，然后使用 pd.DataFrame 函数将该列表转换为 Pandas 的 DataFrame。因为 list 元素是字典对象，它有 key，所以我们不需要为 pd.DataFrame 函数指定 columns 参数。

执行以上代码，输出结果如下：

```
为类别Category C创建一个Pandas DataFrame
     Category  ItemID  Amount
0  Category C       3  100.01
None
为类别Category B创建一个Pandas DataFrame
     Category  ItemID Amount
0  Category B       2  30.10
1  Category B       5  70.85
None
为类别Category A创建一个Pandas DataFrame
     Category  ItemID  Amount
0  Category A       1   12.40
1  Category A       4  110.01
None
```

4.11.2 PySpark SQL DataFrame 可视化

PySpark 还没有任何绘图功能。如果想绘制一些内容，可以将数据从 Spark 上下文中取出并放入“本地”Python 会话中，在那里可以使用 Python 的任意一个绘图库来处理它。

对于 PySpark SQL 的 DataFrame 绘图，我们可以先将它转成 Pandas 的 DataFrame，再绘制。请看下面的示例。

```
from pyspark.sql import SparkSession
import random

# 构建 SparkSession 实例
spark = SparkSession.builder \
    .master("spark://xueai8:7077") \
    .appName("pyspark sql demo") \
    .getOrCreate()

# 创建一个包含 3 个数字列和一个类别(颜色)的 spark dataframe
A = [random.normalvariate(0,1) for i in range(100)]
```

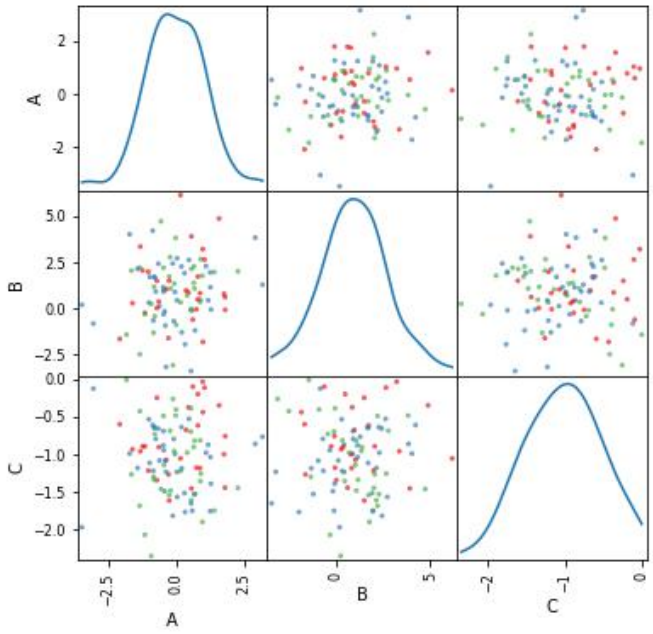
```
B = [random.normalvariate(1,2) for i in range(100)]
C = [random.normalvariate(-1,0.5) for i in range(100)]
col = [random.choice(['#e41a1c', '#377eb8', '#4eae4b']) for i in range(100)]

df = spark.createDataFrame(zip(A,B,C,col), ["A","B","C","col"])
df.printSchema()
df.show(5)

from pandas.plotting import scatter_matrix

# 转换为 pandas 并绘图
pdf = df.toPandas()

stuff = scatter_matrix(pdf, alpha=0.7, figsize=(6, 6), diagonal='kde', color=pdf.col)
```

执行以上代码，输出的 PySpark SQL DataFrame 内容如下：

```
root
 |— A: double (nullable = true)
 |— B: double (nullable = true)
 |— C: double (nullable = true)
 |— col: string (nullable = true)
```

```
+--------------------+--------------------+--------------------+-------+
|                   A|                   B|                   C|    col|
+--------------------+--------------------+--------------------+-------+
|  1.384946838507259|  1.9723595337779216| -1.7433008332887872|#4eae4b|
|-0.9812217969597337|  1.7046046382241848|  -1.113379241508142|#377eb8|
|  0.575954311201969|-0.10590577165060955| -1.7451690616923745|#377eb8|
|-1.0115365029459253| 0.30510247214884667| -0.5342195326699025|#377eb8|
|-3.0459506903597946| -0.8299767662273825|-0.12795069365362288|#377eb8|
+--------------------+--------------------+--------------------+-------+
only showing top 5 rows
```

转换为 Pandas DataFrame 后，绘制结果如下：



4.12 PySpark SQL 编程案例

本节通过几个案例的学习，掌握使用 PySpark 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(DSL API)
- 使用 DataFrame(SQL 语句)

强烈建议不要直接使用 RDD，而是使用 DataFrame 来对数据进行分析计算。只有这样，才能充分利用 PySpark 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 PySpark API 统计某个英文文本中的词频。

实现过程和代码如下所示。

。。。。。。

4.12.2 用户数据集分析

【示例】分析 Spark 安装包自带的 people.txt 文件内容，找出年龄在 13 岁到 19 岁之间的人。

。。。。。。

4.12.3 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 PySpark DataFrame，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 flights20170102.json，位于 PBLP 平台的“/home/hduser/data/flightdelay/”目录下。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 PySpark SQL（高级）

为了帮助执行复杂的分析，PySpark 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 PySpark 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 PySpark SQL 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。PySpark 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 PySpark SQL 执行分布式数据操作，必须熟练使用 PySpark SQL 函数。PySpark 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 pyspark.sql.functions 包内。在使用前，需要先导入它：

```
from pyspark.sql.functions import *
```

如果是使用 pyspark shell 或 zeppelin 进行交互式分析，则会自动导入该包。

5.2 内置标量函数

PySpark SQL 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。

5.2.1 日期时间函数

PySpark SQL 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。

。。。。。。

5.2.2 字符串函数

毫无疑问，大多数数据集的大多数列都是字符串类型的。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。

。。。。。。

5.2.3 数学计算函数

PySpark 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。

。。。。。。

5.2.4 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。

PySpark 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。

。。。。。。

5.2.5 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。本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？PySpark 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。

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。

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。

执行以上代码，输出结果如下所示：

5.2.6 函数应用示例

这一节，通过几个示例程序来演示 PySpark SQL 函数的应用。

【示例】使用 PySpark 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

。。。。。。

5.2.7 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。

- pyspark.sql.functions.exists(col, f)

如果函数对数组中的任何值有返回 true, exists 返回 true。下面我们创建一个 DataFrame，然后运行 pyspark.sql.functions.exists 函数用于附加一个 even_best_number_exists 列。

执行以上代码，输出结果如下：

```
root
 |-- person_id: string (nullable = true)
 |-- best_numbers: array (nullable = true)
 |    |-- element: long (containsNull = true)

+---------+------------+
|person_id|best_numbers|
+---------+------------+
|        a|   [3, 4, 5]|
|        b|     [8, 12]|
|        c|     [7, 13]|
|        d|        null|
+---------+------------+

+---------+------------+-----------------------+
|person_id|best_numbers|even_best_number_exists|
+---------+------------+-----------------------+
|        a|   [3, 4, 5]|                   true|
|        b|     [8, 12]|                   true|
|        c|     [7, 13]|                  false|
|        d|        null|                   null|
+---------+------------+-----------------------+
```

- pyspark.sql.functions.forall(col, f)

用来遍历数组中的每个元素。

执行以上代码，输出结果如下：

```
root
 |-- words: array (nullable = true)
 |    |-- element: string (containsNull = true)

+-------------------------+
|words                    |
+-------------------------+
|[ants, are, animals]     |
|[italy, as, interesting] |
|[brazilians, love, soccer]|
|null                     |
+-------------------------+

+-------------------------+-----------------------+
|words                    |uses_alliteration_with_a|
+-------------------------+-----------------------+
|[ants, are, animals]     |true                   |
|[italy, as, interesting] |false                  |
|[brazilians, love, soccer]|false                 |
|null                     |null                   |
+-------------------------+-----------------------+
```

- pyspark.sql.functions.filter(col, f)

该函数用于过滤数组中的每个元素。

执行以上代码，输出结果如下：

```
root
 |-- words: array (nullable = true)
 |    |-- element: string (containsNull = true)

+-------------------------+
|words                    |
+-------------------------+
|[bad, bunny, is, funny]  |
|[food, is, bad, tasty]   |
|null                     |
+-------------------------+

+-------------------------+-----------------------+
|words                    |filtered_words         |
+-------------------------+-----------------------+
|[bad, bunny, is, funny]  |[bunny, is, funny]     |
|[food, is, bad, tasty]   |[food, is, tasty]      |
|null                     |null                   |
+-------------------------+-----------------------+
```

在下面这个示例中，只保存月份大于 6 的数组元素值。

执行以上代码，输出结果如下：

```
+--------------------------+
|after_second_quarter      |
+--------------------------+
|[2018-09-20, 2019-07-01]  |
+--------------------------+
```

- pyspark.sql.functions.transform(col, f)

相当于 map 操作，用来转换数组中的每个元素。

执行以上代码，输出结果如下：

```
root
 |-- places: array (nullable = true)
 |    |-- element: string (containsNull = true)

+----------------------+
|places                |
+----------------------+
|[New York, Seattle]   |
|[Barcelona, Bangalore]|
|null                  |
+----------------------+

+----------------------+-----------------------------------------+
|places                |fun_places                               |
+----------------------+-----------------------------------------+
|[New York, Seattle]   |[New York is fun!, Seattle is fun!]      |
|[Barcelona, Bangalore]|[Barcelona is fun!, Bangalore is fun!]   |
|null                  |null                                     |
+----------------------+-----------------------------------------+
```

该方法也支持引用数组元素的索引。例如，在下面的示例中，我们将数组中索引为偶数的元素取反。

- pyspark.sql.functions.aggregate(col, initialValue, merge, finish=None)

相当于 Scala 中的 fold 操作，用来执行数组元素的聚合。

执行以上代码，输出结果如下：

在下面这个示例中，我们使用 aggregate 来计算一个数组元素的平均值。

执行以上代码，输出结果如下：

- pyspark.sql.functions.zip_with(left, right, f)

拉链操作，用来组合两个数组中的相同索引位置处的元素。如果一个数组较短，则在应用函数之前在末尾添加空值以匹配较长的数组的长度。

执行以上代码，输出结果如下：

在下面这个示例中，使用 zip_with 函数来合并两个数组列的元素值以计算新的列。

```
df = spark.createDataFrame([(1, [1, 3, 5, 8], [0, 2, 4, 6])], ("id", "xs", "ys"))
```

```
df.select(zip_with("xs", "ys", lambda x, y: x ** y).alias("powers")).show(truncate=False)
```

执行以上代码，输出结果如下：

5.3 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。PySpark SQL 提供了许多常用的聚合函数。

5.3.1 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。

注：该数据集位于 PBLP 平台的/home/hduser/data/spark/目录下。当在使用 IDEA 编写代码时，可以将其拷贝到项目的 src/main/resources/目录下。

下面的代码读取一个价格监测信息数据集，并创建 DataFrame。

```
```

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。

```
print(f"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。

。。。。。。

5.3.2 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。通常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）

```

```

执行以上代码，输出结果如下所示：

监测类别	监测的商品	监测商品数量
粮食	[早籼米，散装面粉，泰国香米，东北米，袋装面粉，晚籼米]	6
食用油	[花生油，菜籽油，豆油，调和油]	4
水产品	[带鱼，草鱼，大头鱼]	3
肉奶蛋	[其中:精瘦肉，鸡蛋，牛肉，肋排骨，猪肉，纯牛奶，白条鸡]	7
蔬菜	[蔬菜均价，西红柿，大白菜，土豆，椰菜，菠菜，生菜，菜心，茄子，、土豆，其中：青椒，黄瓜，芹菜，萝卜]	14
肉蛋奶	[其中:精瘦肉，鸡蛋，牛肉，肋排骨，纯牛奶（元/250ml），猪肉，纯牛奶，白条鸡]	8

5.3.3 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：

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：

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：

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。

5.4 高级分析函数

本节将介绍 PySpark SQL 提供的高级分析函数。

5.4.1 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：

。。。。。。

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 函数的方法类似于使用 rollup 函数。请看下面的示例：

。。。。。。

5.4.2 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。可以使用时间窗口聚合分析时序数据，比如京东方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。

下面的例子将使用苹果股票历史交易数据。下面的代码根据一年的股票交易数据计算苹果股票的周平均价格。

。。。。。。

5.4.3 使用窗口分析函数

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。有时需要对一组数据进行操作，并为每组输入行返回一个值，而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，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。

窗口函数（Window Function）是 SQL2003 标准中定义的一项新特性，并在 SQL2011、SQL2016 中又加以完善，添加了若干拓展。

窗口函数不同于我们熟悉的常规函数及聚合函数，它为每行数据进行一次计算，特点是输入多行（一个窗口）、返回一个值。

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。
- 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。

【示例】要求计算每个产品在过去的三个月中的平均销售额。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。

注：数据集位于 PBLP 平台的/home/hduser/data/spark/目录下。

。。。。。。

【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？
- 每个用户的累计交易金额是多少？

下面我们应用窗口函数来回答这些问题：

。。。。。。

5.5 用户自定义函数(UDF)

尽管 PySpark 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。PySpark 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。

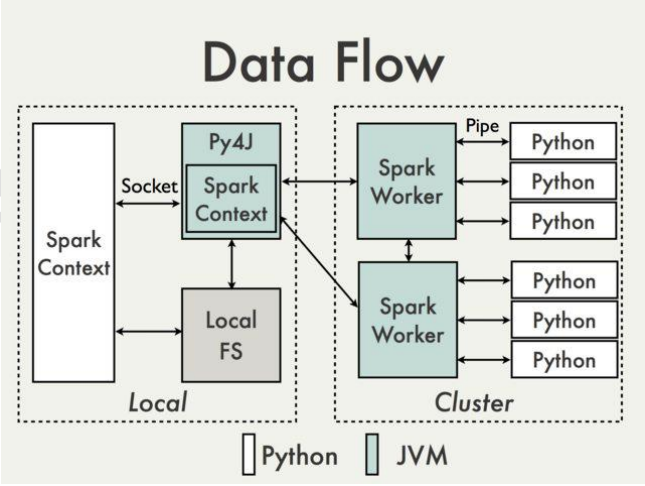
UDF 用于扩展框架的函数，并在多个 DataFrame 上重用这些函数。UDF 实际上是我们可以扩展 Spark 的功能以满足特定需求的一种方式。

当创建 UDF 时，需要非常仔细地设计它们，否则可能会遇到优化和性能问题。UDF 是 PySpark 的黑盒，因此它不能应用优化，我们将失去 PySpark 在 Dataframe 上做的所有优化。如果可能的话，应该使用 Spark SQL 内置函数，因为这些函数提供了优化。考虑只在现有的内置 SQL 函数不能满足业务需求时才创建 UDF。

5.5.1 内部原理

PySpark 实际上是用 Scala 编写的 Spark Core 的包装器。当在 Python 中启动 SparkSession 时，。。。。。。对于结果行，整个序列化/反序列化过程再次以相反的方向发生，以便实际的 filter()可以应用于结果集。

在 PySpark 中使用任意 Python 函数时的整个数据流如下图所示：



由上可知，就执行时间而言，在分布式 Java 系统中执行 Python 函数是非常昂贵的，这是因为来回过度地复制数据。

简单总结一下这种低级的变化：只要我们避免所有类型的 Python UDF，一个 PySpark 程序将会和基于 Scala 的 Spark 程序一样快。如果我们不能避免 udf,我们至少应该尽量让它们尽可能高效。

5.5.2 创建和使用 UDF

在 PySpark 中，使用 UDF 涉及有三个步骤。

- 第一步是用 Python 语法创建一个函数并进行测试。
- 第二步是通过将函数名传递给 PySpark SQL 的 udf()函数来注册它。
- 第三步是在 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。

。。。。。。

【示例】把 DataFrame 中名字字符串中每个单词的第一个字母都转换成大写字母。

。。。。。。

5.5.3 特殊处理

执行顺序

需要注意的一点是，PySpark 并不保证子表达式的求值顺序，这意味着表达式不能保证从左到右或任何其他固定的顺序求值。PySpark 为了查询优化和计划而重新排序执行，因此 AND、OR、WHERE 和 HAVING 表达式会产生副作用。因此，在设计和使用 UDF 时，必须非常小心，特别是空值处理，因为这些结果是运行时异常。

例如，在下面的示例中，不能保证 Name is not null 将首先被执行。如果 convertUDF(Name) like '%John%'先执行的话，有可能得到一个运行时错误（如果 Name 为空的话）。

。。。。。。

空值检查

以下几点需要记住：

- 最好的做法是在 UDF 函数内部检查空值 null，而不是在函数外部检查空值 null。
- 在任何情况下，如果不能在 UDF 中做 null 检查，至少使用 if 或 case when 来检查 null，并有条件地调用 UDF。

例如，下面的代码中我们创建一个 null 值安全的 UDF。

。。。。。。

5.6 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。

5.6.1 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 PySpark 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：

```
joined = df.alias("a").join(df.alias("b")).where(col("a.name") == col("b.name"))
```

Broadcast hash join

在 PySpark SQL 中，通过调用 queryExecution.executedPlan 可以看到正在执行的连接类型。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（broadcast hash join）。可以向 Spark SQL 提供一个提示：某个 DataFrame 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。

5.6.2 使用 join

下面使用两个小型 DataFrame 来演示如何在 PySpark 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：

上述代码分别构造了两个 DataFrame，内容如下：

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：

。。。。。。

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：

.....

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。

。。。。。。

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。

。。。。。。

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：

。。。。。。

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：

。。。。。。

交叉连接

又称笛卡尔连接，是指在 sql 语句中没有写出表连接的条件或者表的连接条件不能约束两个表的连接，优化器把第一个表的每一条记录和第二个表的所有记录相连接。如果第一个表的记录数为 m, 第二个表的记录数为 n,则会产生 m*n 条记录数。

执行交叉连接的代码如下所示。

。。。。。。

5.6.3 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 DataFrame 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，就不太方便引用其中一列。下面模拟这个过程。

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。

2）在 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。

3）使用一个连接列名

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。请看下面的代码：

执行上面的代码，输出结果如下所示：

```
root
 |— dept_no: long (nullable = true)
 |— name: string (nullable = true)
 |— name: string (nullable = true)

+———+———+———+
|dept_no|  name|  name|
+———+———+———+
|     34|  杨幂|财务部|
|     34|楼一萱|财务部|
|     31|刘宏明|销售部|
|     33|  赵薇|工程部|
|     33|黄海波|工程部|
+———+———+———+
```

注意，在 noDupNameDF 中只保留有一个 dept_no 列。

注意，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。

5.6.4 join 实现概述

两个数据集的 join 连接是 Spark 中最昂贵的操作之一。从一个较高的层面上来看，有两种不同的 join 连接策略，分别是 shuffle hash join 和 broadcast join。选择其中哪一种策略，主要标准是基于两个数据集

的大小。当两个数据集的规模都很大时，就会使用 shuffle hash join 策略。当其中一个数据集的大小足够小，小到可以容纳进 executors 的内存时，就会使用 broadcast join 策略。下面分别详细介绍这两种策略。

（1）Shuffle Hash Join

shuffle hash join 的实现由两个步骤组成。首先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。在较高的层次上来看，这两个步骤与 MapReduce 编程模型的步骤很相似。

下图演示了 shuffle hash join 策略中 shuffling 的过程。正如前面提到的，这是一个成本很高的操作，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。在这种情况下，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 的次数。

（2）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。我们知道，shuffle hash join 是一项成本高昂的操作，而 broadcast hash join 策略避免对两个数据集都进行 shuffling，而是只对较小的数据集进行 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过程。

很容易理解，在可能的情况下，首选 broadcast hash join。PySpark 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使用 broadcast hash join 还是 shuffle hash join。然而，也可以在使用 join 连接时，明确提供一个提示给 PySpark SQL 来使用 broadcast hash join。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
from pyspark.sql.functions import broadcast

# 输出执行计划以验证 broadcast hash join 策略被使用了
employeeDF.join(broadcast(deptDF), employeeDF.dept_no == deptDF.dept_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：

```
== Physical Plan ==
*(2) BroadcastHashJoin [dept_no#1L], [dept_id#13L], Inner, BuildRight, false
:- *(2) Filter isnotnull(dept_no#1L)
:  +- *(2) Scan ExistingRDD[name#0,dept_no#1L]
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]),false), [id=#46]
   +- *(1) Filter isnotnull(dept_id#13L)
      +- *(1) Scan ExistingRDD[dept_id#13L,name#14]
```

从输出的物理计划可以看到，利用到了 broadcast hash join。

5.7 读写 Hive 表

PySpark SQL 还支持读取和写入存储在 Apache Hive 中的数据。PySpark 支持两种 SQL 方言：PySpark 的 SQL 方言和 Hive 查询语言(HQL)。PySpark SQL 支持 HiveQL 语法，同时支持 Hive SerDes 和 UDF，可以访问现有的 Hive 仓库。

我们可以通过 PySpark SQL 访问已经存在的 Hive 表并使用已经存在的、社区构建的 Hive UDF。没有部署 Hive 的用户仍然可以启用 Hive 支持。

Spark 中与每个表相关联的是它的相关元数据，它是关于表及其数据的信息：模式、描述、表名、数据库名、列名、分区、实际数据所在的物理位置等。所有这些都存储在一个中央元存储中。

Spark 默认情况下没有为 Spark 表提供单独的元存储，而是使用 Apache Hive metastore，位于/user/hive/warehouse，用于持久化关于表的所有元数据。但是，可以通过将 Spark 配置变量 spark.sql.warehouse.dir 设置为另一个位置来更改默认位置，该位置可以设置为本地或外部分布式存储。

5.7.1 PySpark SQL 的 Hive 配置

要通过 PySpark 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

Hive Metastore Server 是 Spark SQL 应用程序将要连接到的服务器，用于获取 Hive 表的元数据。启动 Hive Metastore Server 的命令如下：

5.7.2 PySpark SQL 读写 Hive 表

可以使用 SparkSession 或 DataFrameReader 的 table()方法从一个 Hive metastore 的注册表中加载一个 DataFrame。例如，我们使用 table 方法加载 Hive 表 mytable 中的数据：

```
spark.read.table("mytable").show()
```

或者：

```
spark.table("mytable").show()
```

将 DataFrame 保存到 Hive 表中，使用 DataFrameWriter 的 saveAsTable()方法，它会将 DataFrame 数据保存到 Hive 表中，并在 Hive metastore 中注册。例如：

```
spark.range(0,10).write.saveAsTable("test_tb")
spark.table("test_tb").show()
```

默认情况下，如果不指定自定义表路径的话，Spark 将把数据写到 warehouse 目录下的默认表路径。当删除表时，默认的表路径也将被删除。

为 Hive 表指定存储格式

当创建一个 Hive 表时，需要定义这个表如何从/向文件系统读取/写入数据，即“输入格式”和“输出格式”。还需要定义此表应该如何将数据反序列化为 Row，或将 Row 序列化为数据，即“serde”。下表中的 OPTIONS 可以用来指定存储格式（"serde"、"input format"、"output format"），例如：

```
CREATE TABLE src(id int) USING hive OPTIONS(fileFormat 'parquet')
```

默认情况下，我们将以纯文本的形式读取表文件。请注意，在创建表时还不支持 Hive 存储处理程序，我们可以在 Hive 端使用存储处理程序创建一个表，并使用 Spark SQL 读取它。

属性名	含义
fileFormat	文件格式是一种存储格式规范的包，包括"serde"、"输入格式"和"输出格式"。目前支持6种文件格式:"sequencefile"、"rcfile"、"orc"、"parquet"、"textfile"和"avro"。
inputFormat, outputFormat	这两个选项指定相应的' InputFormat '和' OutputFormat '类的名称， 例如。"org.apache.hadoop.hive.ql.io.orc.OrcInputFormat"。 这两个选项必须成对出现，如果已经指定了“fileFormat”选项，则不能指定它们。
serde	此选项指定serde类的名称。 当指定"fileFormat"选项时，如果给定的"fileFormat"已经包含serde的信息，则不要指定此选项。 目前"sequencefile"、"textfile"和"rcfile"不包含serde信息，可以对这3种文件格式使用此选项。
fieldDelim, escapeDelim, collectionDelim, mapkeyDelim, lineDelim	这些选项只能用于"textfile"文件格式。它们定义如何将带分隔符的文件读入行。

请看下面的示例程序。

【示例】使用 PySpark SQL 读取 Hive 元数据及读写 Hive 表数据。请安以下步骤操作。

请注意：

- 当我们将任何表作为分区表处理时，分区列变得区分大小写。
- 分区列应该在 DataFrame 中以相同的名称出现(区分大小写)。

【示例】使用 Spark SQL 读取 Hive 元数据及读写 Hive 表数据。本示例中将使用的数据是 MovieLens 数据集。我们将使用其中的 movies、ratings 和 tags 数据集。

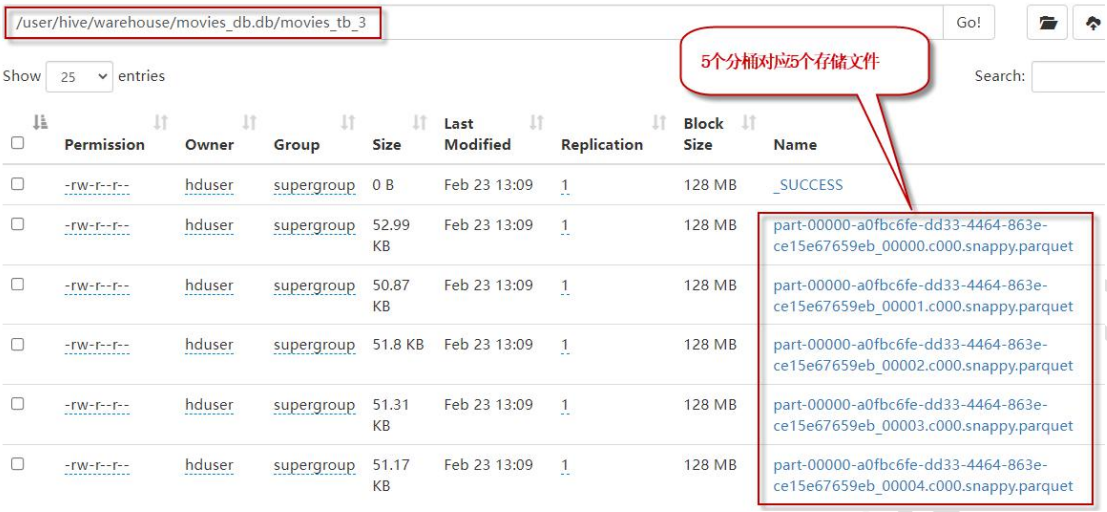
请安以下步骤操作。

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，然后通过查看实际的数据存储结构，如下图中所示：



可以看到，我们将数据存储到 5 个桶中，对应 HDFS 上 5 个存储文件。

有可能对单个表同时使用分区和分桶以及排序。请看下面的示例。

执行以上代码，输出结果如下图所示：

```
+--------------------+--------------------+----+
|               actor|               title|year|
+--------------------+--------------------+----+
|        Abrell, Brad|Men in Black 3|2012|
|      Adams, January|Men in Black 3|2012|
|       Armisen, Fred|  The Dictator|2012|
|        Arnett, Will|Men in Black 3|2012|
|Arthur, Michelle (I)|    Battleship|2012|
+--------------------+--------------------+----+
only showing top 5 rows
```

然后查看写入的实际存储结构，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。

5.8 PySpark SQL 编程案例

本节通过几个案例的学习，掌握使用 PySpark SQL 进行大数据分析的复杂用法。

5.8.1 电商订单数据分析

本示例使用一个电商数据集 nw。该数据集位于 PBLP 平台的/home/hduser/data/spark/目录下。

【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?

请按以下步骤执行。

。。。。。。

5.8.2 电影评分数据集分析

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，它已经位于 PBLP 平台的/home/hduser/data/spark/目录下。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”）。

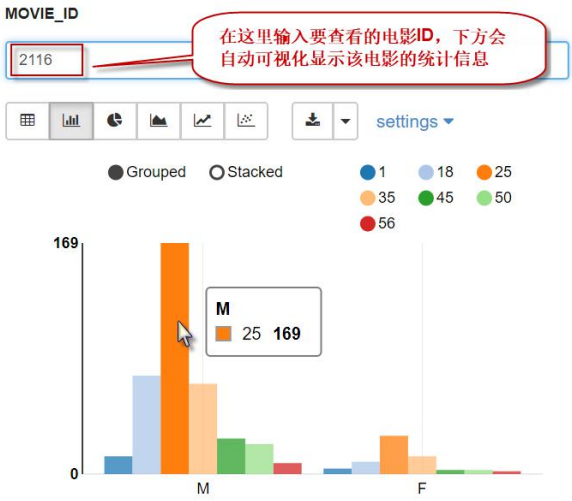
请按以下步骤执行。

。。。。。。

（7）如何使用 zeppelin 作为交互式分析环境，则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：



5.9 小结

- 聚合是大数据分析领域中最常用的特性之一。Spark SQL 提供了许多常用的聚合函数，如 sum、count、avg 等等。pivoting 聚合提供了一种很好的方法来总结数据，并将列转换成行。
- 做任何有用且有意义的数据处理通常需要 join 两个或多个数据集。Spark SQL 支持在 SQL 世界中存在的许多标准 join 类型。
- Spark SQL 附带了一组丰富的内置函数，它应该涵盖了处理字符串、数学、日期和时间等方面的大部分常见需求。如果它们都没有满足某个用例的特定需求，那么就很容易编写一个用户定义的函数，该函数既可以用于 DataFrame APIs 和 SQL 查询。
- 窗口函数是强大的和高级的分析函数，因为它们可以计算输入组中的每一行的值。它们对于计算移动平均、累积和或每一行的秩（rank）特别有用。
- 与 RDD 相比，DataFrames 得益于 Spark SQL 的高效存储格式、Catalyst 优化器以及直接对序列化数据执行某些操作的能力。

第 6 章 PySpark 结构化流（初级）

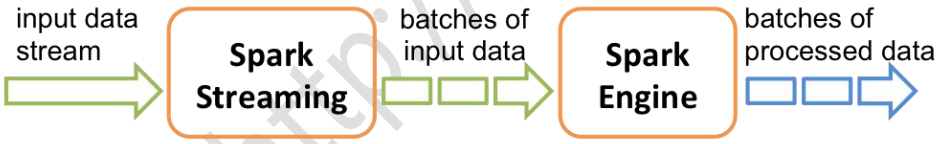
在很多领域，如股市走向分析、气象数据测控、网站用户行为分析等，由于数据产生快，实时性强，数据量大，所以很难统一采集并入库存储后再做处理，这便导致传统的数据处理架构不能满足需要。流计算的出现，就是为了更好地解决这类数据在处理过程中遇到的问题。与传统架构不同，流计算模型在数据流动的过程中实时地进行捕捉和处理，并根据业务需求对数据进行计算分析，最终把结果保存或者分发给需要的组件。

另外，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。而 Apache Spark 的统一数据处理平台能够同时执行流数据处理和批处理数据处理。

6.1 PySpark DStream 流简介

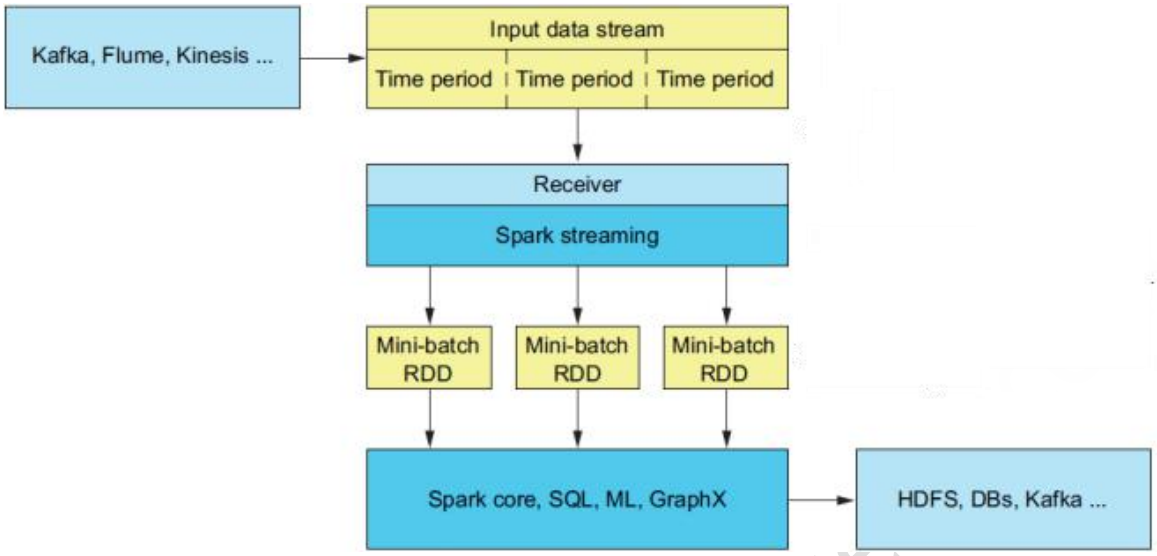
第一代 Spark 流式处理引擎 Spark Streaming 是在 2012 年引入的，这个引擎的主要编程抽象称为离散流，即 DStream，它表示连续的数据流。下图显示了 DStream 的工作方式。



Spark Streaming 是对核心 Spark API 的扩展，支持可扩展、高吞吐量、容错的实时数据流处理。一个 DStream 可以从 Kafka、AWS Kinesis 或 TCP 套接字等许多输入源获取并创建，也可以通过对其他 DStreams 应用高级操作创建，并且可以使用复杂的算法来处理，这些算法由高级函数(如 map、reduce、join 和 window)表示。

对于实时数据处理，Spark Streaming 使用“微批处理”模型。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，这意味着 Spark 流会将特定时间段内的数据块打包成 RDD，然后由 Spark 批

处理引擎处理。在内部，DStream 被表示为 RDD 的序列。如下图所示：



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 PySpark DStream 应用程序。通过这个程序我们可以了解一个典型的 PySpark DStream 应用程序。

【示例】实时统计文本流的单词数量（单词计数的实时处理版本）。

要运行该程序，请按以下步骤执行。

。。。。。。

（1）需要首先运行一个 Netcat 服务器。打开一个终端窗口，执行：

```
$ nc -lk 9999
```

（2）然后打开第二个终端窗口，使用如下命令将 network_wordcount.py 程序提交到 PySpark 集群上运行。（请确保已经启动了 Spark 集群，并假设 network_wordcount.py 位于~/pyspark_demos/chapter06/目录下）

```
$ cd ~/bigdata/spark-3.1.2
$ ./bin/spark-submit --master spark://xueai8:7077 ~/pyspark_demos/chapter06/network_wordcount.py localhost 9999
```

执行过程如下所示：

```
[hduser@xueai8 ~]$ cd bigdata/spark-3.1.2/
[hduser@xueai8 spark-3.1.2]$ ./bin/spark-submit --master spark://xueai8:7077 ~/pyspark_demos/chapter06/network_wordcount.py localhost 9999
```

（3）回到第一个终端窗口，任意输入一些内容，单词之间以空格分隔，例如：

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

执行过程如下所示：

```
[hduser@xueai8 ~]$ nc -lk 9999
good good study
day day up
```

（4）在第二个终端窗口，查看流程序输出的计算结果。

```
-------------------------------------------
Time: 2022-02-25 10:41:45
-------------------------------------------
('good', 2)
('study', 1)

-------------------------------------------
Time: 2022-02-25 10:41:48
-------------------------------------------
('day', 2)
('up', 1)
```

6.2 PySpark 结构化流简介

Apache Spark 2.0 引入了更新的高级别的流处理 API，叫做 Structured Streaming（结构化流）。这个可扩展和容错的高级流 API 构建在 Spark SQL 引擎之上，与 SQL 查询和 DataFrame API 紧密集成。主要优点是使用相同的 Spark DataFrame 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。

结构化流是一种基于 Spark SQL 引擎的快速、可扩展、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。

PySpark 结构化流提供了快速、可扩展、容错、端到端的精确一次性有状态流处理。它支持流分析，而无需考虑流的底层机制。结构化流操作直接工作在 DataFrame 上。不再有“流”的概念，只有流式 DataFrame 和普通 DataFrame。流式 DataFrame 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。

使用 PySpark 结构化流的模型如下图所示：

在结构化流处理模型中，是将实时数据流视为一个不断增长的表。触发器指定检查输入新数据到达的时间间隔。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。

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。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 的结构化 API，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。

以单词计数示例，当查询启动时，Spark 将持续检查来自套接字连接的新数据。如果有新数据，Spark 将运行一个“增量”查询，将以前的运行计数与新数据结合起来，计算更新的计数。

可以使用统一入口点 SparkSession 从流源创建流 DataFrame，并对它们应用与静态 DataFrame 相同

的操作。有一些内置的源，例如文件，支持的文件格式是文本、csv、json、orc、parquet。

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。

6.3 PySpark 结构化流编程模型

假设有一个监听 TCP 套接字的数据服务器，我们想维护该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 PySpark 运行时单词计数流程序。（下面先分步执行，最后是完整的程序）

请按以下步骤操作。

要运行这个结构化流程序，请按以下步骤操作。

（1）首先需要使用 Netcat 作为数据服务器。在 Linux 的第一个终端窗口中，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
```

（2）然后打开第二个终端窗口，使用如下命令将 structed_wordcount.py 程序提交到 PySpark 集群上运行。（请确保已经启动了 Spark 集群，并假设 structed_wordcount.py 位于~/pyspark_demos/chapter06/目录下）

```
$ cd ~/bigdata/spark-3.1.2
$ ./bin/spark-submit --master spark://xueai8:7077 ~/pyspark_demos/chapter06/structed_wordcount.py localhost 9999
```

执行过程如下所示：

```
[hduser@xueai8 ~]$ cd bigdata/spark-3.1.2/
[hduser@xueai8 spark-3.1.2]$ ./bin/spark-submit --master spark://xueai8:7077 ~/pyspark_demos/chapter06/structed_wordcount.py localhost 9999
```

（3）切换到第一个终端窗口，任意输入一些内容，单词之间以空格分隔，例如：

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

执行过程如下所示：

```
[hduser@xueai8 ~]$ nc -lk 9999
good good study
day day up
```

（4）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：

```
-------------------------------------------
Batch: 1
-------------------------------------------
22/02/25 12:10:17 INFO CodeGenerator: Code generated in 34.165973 ms
22/02/25 12:10:17 INFO CodeGenerator: Code generated in 10.95491 ms
+-----+-----+
| word|count|
+-----+-----+
|study|    1|
| good|    2|
+-----+-----+
```

```
-------------------------------------------
Batch: 2
-------------------------------------------
+-----+-----+
| word|count|
+-----+-----+
|  day|    2|
|study|    1|
|   up|    1|
| good|    2|
+-----+-----+
```

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。

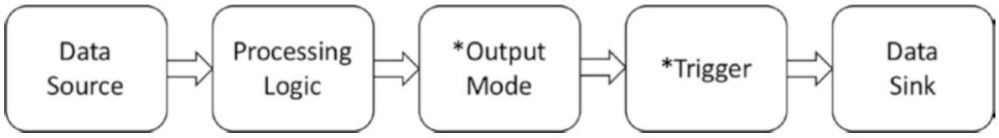
```
# 这是阻塞调用
mobileSQ.awaitTermination()

# 停止流查询
mobileSQ.stop()
```

6.4 PySpark 结构化流核心概念

PySpark 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。

PySpark 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。

6.5 使用数据源

上一节描述了结构化流提供的各个内置源。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。

6.5.1 使用 Socket 数据源

Socket（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，不要在生产环境中使用。下面这个示例应用 socket 数据源。

6.5.2 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。

。。。。。。

6.5.3 使用 File 数据源

文件数据源是最容易理解和使用的。PySpark 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。

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：

```
# 使用 File 数据源，读取 json 文件
mobileSSDF = spark.readStream \
    .schema(mobileDataSchema) \
    .json("<directoryname>")

# 如果我们指定 maxFilesPerTrigger 为 1，表示一个文件一个文件地处理
mobileSSDF = spark.readStream \
    .schema(mobileDataSchema) \
    .option("maxFilesPerTrigger",1) \
```

```
    .json("<directory name>")

# 如果我们指定 latestFirst 为 true，则表示首先处理新产生的文件
mobileSSDF = spark.readStream \
    .schema(mobileDataSchema) \
    .option("latestFirst", "true") \
    .json("<directory name>")
```

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 PySpark 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：

注：这三个 JSON 文件位于 PBLP 平台的/home/hduser/data/spark/stream/mobile/目录下。

2）先导入相关的依赖包，并构造一个 SparkSession 实例。

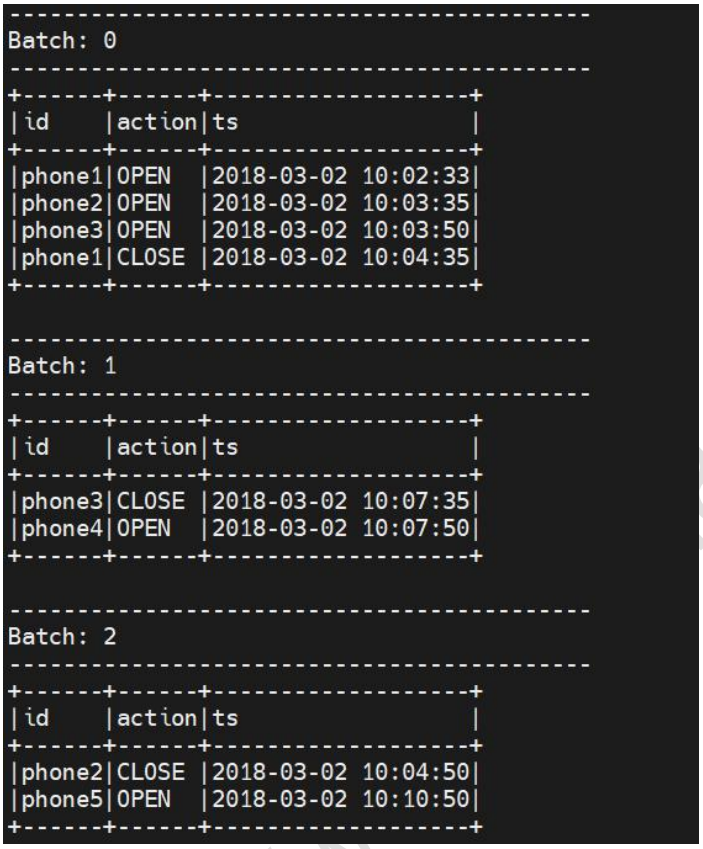
3）为手机事件数据创建模式（schema）

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）将结果 DataFrame 输出到控制台显示。

5）执行流处理程序，输出结果如下所示。



6.5.4 使用 Kafka 数据源

Apache Kafka 作为集群在一个或多个服务器上运行，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 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 中主题（Topic）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。如果想要进一步学习 Kafka，可参考如下两个资源：

- Kafka 官网：https://kafka.apache.org/
- 小白学苑：Kafka 教程。

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 PBLP 平台上，使用 kafka_2.12-2.4.1.tgz 这个版本。

下面我们通过一个示例来演示如何编写 PySpark 结构化流处理程序来读取 Kafka 中的数据。

【示例】编写 PySpark 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

等待 30 秒左右 ZooKeeper 启动。

2）接下来，启动 Kafka 服务器。

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)。

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

然后，随意键入一些消息。例如：

```
[hduser@xueai8 kafka_2.12-2.4.1]$ ./bin/kafka-console-producer.sh --broker-list xueai8:9092 --topic test
>good good study
>day day up
>aaa bbb ccc
>
```

随意输入一些内容

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：

```
>>> from pyspark.sql.types import *
>>> from pyspark.sql.functions import *
>>> dataDF = spark.readStream \
...       .format("kafka") \
...       .option("kafka.bootstrap.servers", "xueai8:9092") \
...       .option("subscribe", "test") \
...       .option("startingOffsets", "earliest") \
...       .load()
>>> dataDF.printSchema()
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)

>>> kvstream = dataDF.selectExpr("CAST(value as string)","topic","partition","offset")
>>> query = kvstream.writeStream.outputMode("append").format("console").start()
22/02/25 17:19:54 WARN StreamingQueryManager: Temporary checkpoint location created which
7d-7361-4778-b730-b1895d66463d. If it's required to delete it under any circumstances, ple
. Important to know deleting temp checkpoint folder is best effort.
-------------------------------------------
Batch: 0
-------------------------------------------
+---------------+-----+---------+------+
|          value|topic|partition|offset|
+---------------+-----+---------+------+
|good good study| test|        0|     0|
|     day day up| test|        0|     1|
+---------------+-----+---------+------+

query.stop()
>>>
```

流DataFrame Schema

流计算输出结果

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：

6.6 流 DataFrame 操作

前面的例子表明，一旦配置和定义了数据源，pyspark.sql.streaming.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 DataFrame 关系操作和 PySpark 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 就不能在流式 DataFrame 上使用，这是因为它们在流数据处理的上下文中不适用。

6.6.1 选择、投影和聚合操作

PySpark 结构化流的一个优点是具有一组统一 API 用于 PySpark 的批处理和流处理。使用流数据格式的 DataFrame，可以应用任何 select 和 filter 转换，以及任何作用在列上的 PySpark SQL 函数。此外，基本聚合和高级分析函数也可用于流 DataFrame。下面我们通过一个示例程序来演示这些用法。

【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 PySpark 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：

这三个 JSON 文件位于 PBLP 平台的/home/hduser/data/spark/stream/mobile/目录下。

2）先导入相关的依赖包，并构造一个 SparkSession 实例。

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。（注意，这里为了简化测试，我们使用了本地文件路径。在生产环境下，请使用 HDFS 路径）

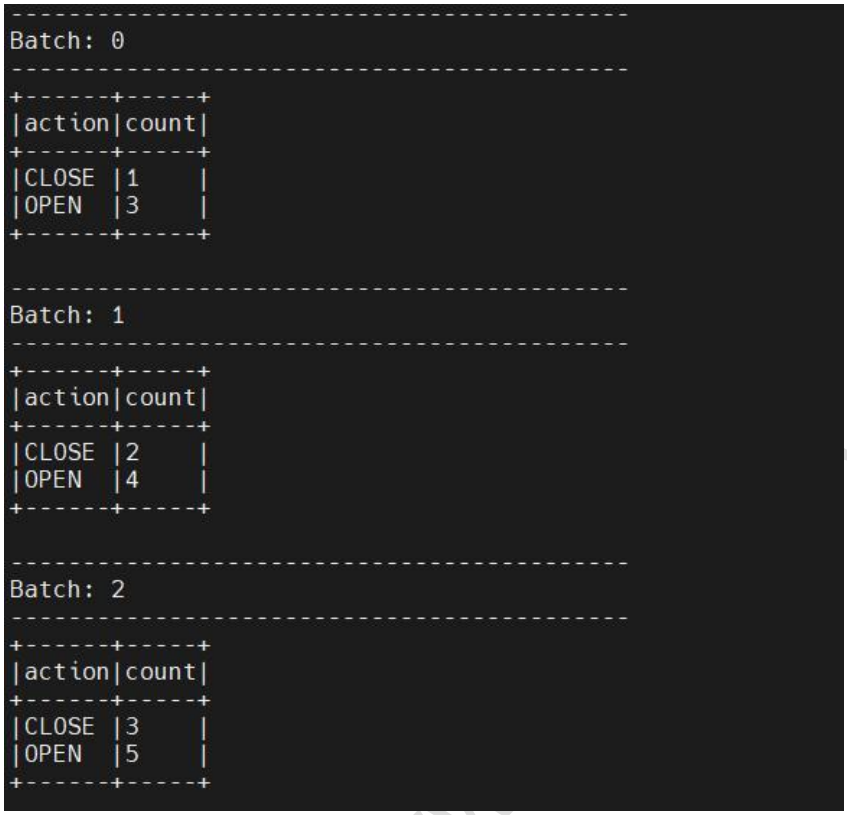
4）将 action 列值转换为大写，并执行过滤、投影、聚合等转换操作。

5）将结果 DataFrame 输出到控制台显示。

#)

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。

6）执行流处理程序，输出结果如下所示。



在 PySpark Structured Streaming 结构化流程序中，也可以创建一个视图来应用 SQL 查询。修改上面示例的代码如下：

执行以上代码，会看到相同的输出结果。

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。

6.6.2 执行 join 操作

从 Spark 2.3 开始，结构化流支持对两个流 DataFrame 执行 join 操作。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。

可以用一个流式 DataFrame 来 join 连接另一个静态的 DataFrame 或者流式 DataFrame。然而，join 连接是一个复杂的操作，其中最棘手的在于并非所有的数据在连接时都是可用的流 DataFrame。因此，join 连接的结果是在每个触发器点上增量地生成的。

下面我们通过一个 IOT（物联网）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。

file1_temp.json 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：

file2_load.json 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：

注：以上两个数据文件位于 PBLP 平台的/home/hduser/data/spark/stream/temp/目录下。

现在我们需要编写一个 PySpark 结构化流处理程序，对上面这两个流数据集执行 join 连接，以统计每个机架实时的温度和负载。实现代码如下所示（请注意其中的连接条件和时间约束条件设置方法）：

说明：上面代码中 timestampFormat 选项不是必须的。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：

```
-------------------------------------------
Batch: 0
-------------------------------------------
+----------------+-----------+-------------------+----------------+------+-------------------+
|temp_location_id|temperature|temp_taken_time    |load_location_id|load  |load_taken_time    |
+----------------+-----------+-------------------+----------------+------+-------------------+
|rack2           |100.5      |2017-06-02 08:06:02|rack2           |1105.5|2017-06-02 08:06:04|
|rack1           |99.5       |2017-06-02 08:01:01|rack1           |199.5 |2017-06-02 08:01:02|
|rack3           |101.0      |2017-06-02 08:11:03|rack3           |2104.0|2017-06-02 08:11:06|
|rack4           |102.0      |2017-06-02 08:16:04|rack4           |1108.0|2017-06-02 08:16:08|
|rack4           |102.0      |2017-06-02 08:16:04|rack4           |1108.0|2017-06-02 08:21:10|
+----------------+-----------+-------------------+----------------+------+-------------------+
```

那么，在这个结果表上，我们可以进一步执行 Spark SQL 查询操作。

流 DataFrame 连接限制

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

6.7 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。PySpark 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，如下表所示。

Sink	支持的模式	选项参数	是否容错	备注
File Sink	append	path：输出目录路径，必须指定。 retention：输出文件的生命周期(TTL)。批量提交	是(exactly-once)	支持写入分区表。

		的比TTL时间早的输出文件，最终将被排除在元数据日志中。这意味着读取接收器输出目录的读取器查询可能无法处理它们。可以提供时间的字符串格式的值(如“12h”、“7d”等)。 默认情况下是禁用的。		按时间进行分区可能是有用的。
Kafka Sink	append update complete	参见后面的6.7.2示例	是(at-least-once)	
Foreach Sink	append update complete	无	是(at-least-once)	
ForeachBatch Sink	append update complete	无	依赖于实现	
Console Sink	append update complete	numRows：每个触发器打印的行数，默认20 truncate：是否截断过长的输出，默认true	否	
Memory Sink	append complete	无	否 在 complete 模式下，重启查询将重新创建全表	

下面的部分将详细介绍每个 Data Sink。

6.7.1 使用 File Data Sink

文件数据接收器（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。由于文件数据接收器是容错的，因此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。

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}
```

6.7.2 使用 Kafka Data Sink

在 PySpark 结构化流中，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 中这一列不是必须的。

另一方面，必须提供 value 值，而 Kafka 的 data sink 也要求在流 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 PySpark 结构化流处理程序，必须添加 Kafka 的依赖包到 classpath 中。

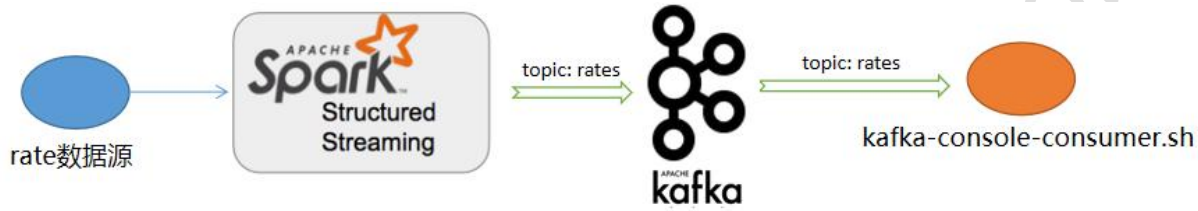
如果我们要从 pyspark shell 使用 Kafka 数据源，那么需要在启动 pyspark shell 时将依赖的 JAR 包添

加到 classpath 中。具体方式可参考 6.5.4 节。

下面我们编写一个 PySpark 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 PySpark 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，PySpark 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 PySpark 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

等待 30 秒左右 ZooKeeper 启动。

2）接下来，启动 Kafka 服务器。

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)。

查看已有的主题，使用如下的命令：

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。

5）启动上面的流处理程序。执行过程如下图所示：

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）：

```
[hduser@xueai8 kafka_2.12-2.4.1]$ ./bin/kafka-console-consumer.sh --bootstrap-server xueai8:9092 --topic rates
{"timestamp":"2022-02-26T08:17:56.181+08:00","value":1}
{"timestamp":"2022-02-26T08:17:56.081+08:00","value":0}
{"timestamp":"2022-02-26T08:17:56.281+08:00","value":2}
{"timestamp":"2022-02-26T08:17:56.481+08:00","value":4}
{"timestamp":"2022-02-26T08:17:56.681+08:00","value":6}
{"timestamp":"2022-02-26T08:17:56.881+08:00","value":8}
{"timestamp":"2022-02-26T08:17:57.081+08:00","value":10}
{"timestamp":"2022-02-26T08:17:57.281+08:00","value":12}
{"timestamp":"2022-02-26T08:17:57.481+08:00","value":14}
{"timestamp":"2022-02-26T08:17:57.681+08:00","value":16}
{"timestamp":"2022-02-26T08:17:56.381+08:00","value":3}
{"timestamp":"2022-02-26T08:17:57.881+08:00","value":18}
{"timestamp":"2022-02-26T08:17:56.581+08:00","value":5}
{"timestamp":"2022-02-26T08:17:58.081+08:00","value":20}
```

6.7.3 使用 Foreach 和 ForeachBatch Data Sink

与结构化流提供的其他内置 data sinks 相比，foreach 和 foreachBatch data sink 是很有意思的数据接收器。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。

foreach 和 foreachBatch 操作允许我们对流查询的输出应用任意操作和编写逻辑，但它们的应用场景略有不同—foreach 允许在每一行上自定义写逻辑，而 foreachBatch 允许对每个微批处理的输出进行任意操作和自定义逻辑。

ForeachBatch

使用 foreachBatch(…)，可指定一个函数，该函数在流查询的每个微批的输出数据上执行。它接受两个参数：微批输出数据的 DataFrame 和微批的唯一 ID。其使用基本模式如下：

```
def foreach_batch_function(df, epoch_id):
    # 转换并写 df
    pass

streamingDF.writeStream.foreachBatch(foreach_batch_function).start()
```

使用 foreachBatch，我们可以执行以下操作：

- 重用现有的批处理数据源—对于许多存储系统来说，可能还没有可用的流接收器，但可能已经存在用于批处理查询的数据写入器。使用 foreachBatch，我们可以在每个微批处理的输出上使用批处理数据写入器。
- 写入多个位置—如果想将流查询的输出写入多个位置，那么可以简单地写输出 DataFrame 多次。但是，每次尝试写出都可能导致重新计算输出数据(包括可能重新读取输入数据)。为了避免重新计算，应该缓存输出 DataFrame，将其写入多个位置，然后取消缓存。例如：

```
def foreach_batch_function(df, epoch_id):
    df.persist()
    df.write.format(...).save(...)          // location 1
    df.write.format(...).save(...)          // location 2
    df.unpersist()
    pass

streamingDF.writeStream.foreachBatch(foreach_batch_function).start()
```

- 应用额外的 DataFrame 操作—许多 DataFrame 操作在流式 DataFrame 中是不支持的，因为在这些情况下 Spark 不支持生成增量计划。使用 foreachBatch，可以对每个微批处理输出应用其中一些操作。但是，我们必须自己推理执行该操作的端到端语义。

默认情况下，foreachBatch 只提供 at-least-once（至少一次）写保证。但是，可以使用提供给该函数的 batchId 来消除输出的重复，并获得 exactly-once（精确一次）的保证。另外，foreachBatch 不能用于连续（continuous）处理模式，因为它从根本上依赖于流查询的微批处理执行。如果以连续模式写入数据，则使用 foreach 代替。（从 Spark 2.4 开始，foreachBatch 可以在 Python 中使用）

Foreach

在不能选择使用 foreachBatch 时（例如，对应的批处理数据写入器不存在，或者连续处理模式不存在），那么可以使用 foreach 来表达自定义写入器逻辑。具体来说，可以通过将数据写入逻辑分为三个

方法来表示：open、process 和 close。（从 Spark 2.4 开始，foreach 可以在 Python 中使用）

在 Python 中，可以用两种方式调用 foreach：在一个函数中或在一个对象中。使用函数的方式提供了一种简单的方法来表达处理逻辑，但不允许在失败导致重新处理某些输入数据时对生成的数据进行去重。对于这种情况，必须在对象中指定处理逻辑。

使用函数方式时，该函数接收一行作为输入，其基本模式如下：

```
def process_row(row):
    # 将 row 写到存储中
    pass

query = streamingDF.writeStream.foreach(process_row).start()
```

使用对象方式时，该对象具有一个 process 方法，以及可选的 open 和 close 方法，其基本模式如下：

```
class ForeachWriter:

    def open(self, partition_id, epoch_id):
        # 打开连接。这个方法是可选的
        pass

    def process(self, row):
        # 将 row 写入连接。这个方法是必须的
        pass

    def close(self, error):
        # 关闭连接。这个方法是可选的
        pass

query = streamingDF.writeStream.foreach(ForeachWriter()).start()
```

ForeachWriter 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 类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。
- 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。
- 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。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 PySpark 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 类的简单实现。

首先定义一个 ForeachWriter 类的实现，并实现它的三个方法：open、process 和 close。

然后，在流数据写出时，指定 forach 方法使用上面自定义的 ForeachConsoleWriter。

注意：以上代码，请在 Local 模式下执行，因为 print()打印函数在集群模式下 driver 端是看不到内容的。

当开始执行时，可以看到控制台的输出，类似下面这样：

```
open => (0,0)                                                   (0 + 1) / 1]
close => (0, 0)
open => (1,1)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 13, 415000), value=1)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 13, 615000), value=3)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 13, 815000), value=5)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 15000), value=7)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 215000), value=9)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 415000), value=11)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 615000), value=13)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 815000), value=15)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 15000), value=17)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 215000), value=19)
close => (1, 1)
open => (0,1)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 13, 315000), value=0)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 13, 515000), value=2)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 13, 715000), value=4)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 13, 915000), value=6)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 115000), value=8)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 315000), value=10)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 515000), value=12)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 715000), value=14)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 14, 915000), value=16)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 115000), value=18)
close => (0, 1)
open => (1,2)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 415000), value=21)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 615000), value=23)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 815000), value=25)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 16, 15000), value=27)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 16, 215000), value=29)
close => (1, 2)
open => (0,2)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 315000), value=20)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 515000), value=22)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 715000), value=24)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 15, 915000), value=26)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 16, 115000), value=28)
close => (0, 2)
open => (0,3)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 16, 315000), value=30)
writing => Row(timestamp=datetime.datetime(2022, 2, 26, 10, 30, 16, 515000), value=32)
```

6.7.4 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。

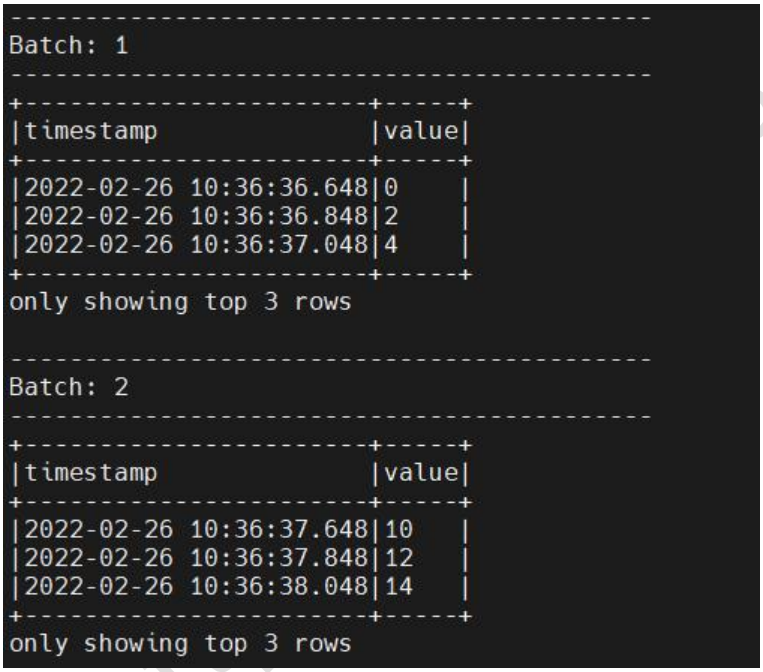
Option	默认值	描述
numRows	20	在控制台输出的行的数量
truncate	true	当每一行的内容超过20个字符时，是否截断显示

下面通过一个示例来了解这些 option 参数的用法。

【示例】编写 PySpark 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。

执行上面的程序，输出结果如下（部分）：



6.7.5 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 函数的参数，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中，就可以使用几乎所有在 PySpark 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 PySpark 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：运行此代码，如果集成了 Hive，则需要启动 metastore：hive --service metastore

需要注意的一点是，即使在流查询停止之后，内存中的 rates_tb 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。

6.8 深入研究输出模式

在 PySpark 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。

6.8.1 无状态流查询

无状态流查询的典型应用是执行实时的流 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。在这种应用场景中，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。

前面部分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 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（使用无状态流查询时使用 Complete 输出模式）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：

```
.
```

6.8.2 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 PySpark 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

```
```

newaction.json

```
```

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”。

注：这两个 JSON 文件位于 PBLP 平台的/home/hduser/data/spark/stream/mobile2/目录下。

在下面编写 PySpark 结构化流程序中，读取文件流数据源并执行聚合操作，然后输出结果。这里我们使用的输出模式是“complete”。实现代码如下：

```
```

执行上面的代码，流查询输出如下：

```
-------------------------------------------
Batch: 0
-------------------------------------------
+------+-----+
|action|count|
+------+-----+
|close |2    |
|open  |3    |
+------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+------+-----+
|action|count|
+------+-----+
|close |2    |
|crash |1    |
|open  |4    |
|swipe |1    |
+------+-----+

query.stop()
```

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：

```
>>> -------------------------------------------
Batch: 0
-------------------------------------------
+------+-----+
|action|count|
+------+-----+
|close |2    |
|open  |3    |
+------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+------+-----+
|action|count|
+------+-----+
|crash |1    |
|open  |4    |
|swipe |1    |
+------+-----+

query.stop()
```

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 close 统计结果，但是包含了 open 统计项。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：

```
.....
```

注：

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。

6.9 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。下表列出了 PySpark 结构化流所支持的不同类型的触发器。

触发器类型	描述
未指定（默认）	如果没有显式指定触发器设置，那么默认情况下，查询将以微批处理模式执行，在这种模式下，上一个微批处理完成后将立即生成新的微批。
固定间隔的微批	查询将以微批模式执行，其中微批将在用户指定的时间间隔启动。 ✓ 如果前一个微批处理在间隔内完成，那么引擎将等待直到间隔结束，然后启动下一个微批处理。 ✓ 如果前一个微批需要的时间比间隔时间长(即一个间隔边界丢失)，那么下一个微批将在前一个微批完成后立即开始(即不会等待下一个间隔边界)。 ✓ 如果没有新数据可用，则不会启动微批处理。
一次性微批	查询将只执行一个微批处理来处理所有可用数据，然后自行停止。这在希望定期启动集群、处理自上一阶段以来可用的所有内容、然后关闭集群的场景中非常有用。在某些情况下，这可能会导致显著的成本节约。
连续，具有固定的检查点间隔	查询将在新的低延迟、连续处理模式下执行。

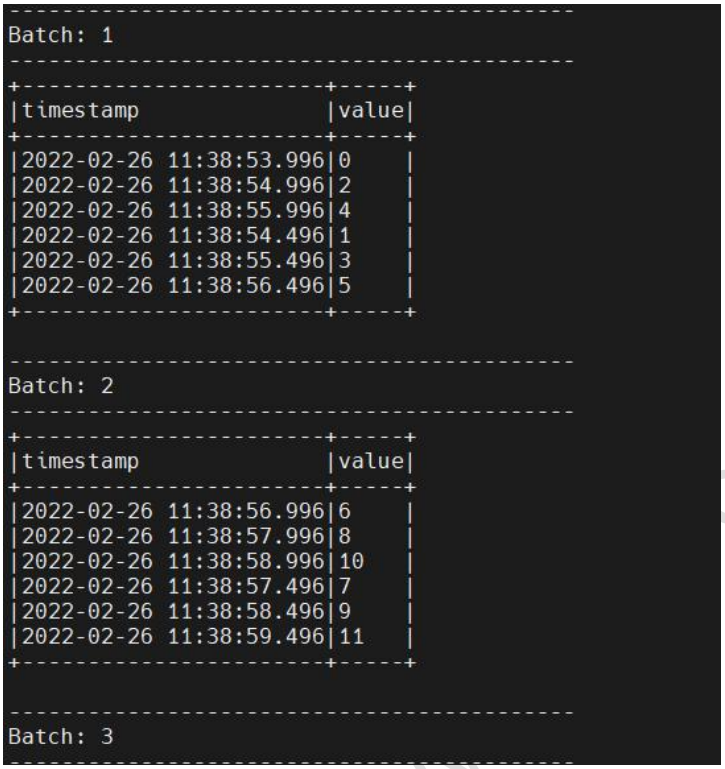
到目前为止，我们所有的流查询示例都没有指定触发器类型，而是使用了默认触发器类型，因为。**这个默认的触发器类型选择微批模式作为处理模式**，而流查询中的逻辑执行不是基于时间的，而是在前一批数据完成处理后立即执行的。这意味着，在数据被写入的频率方面，可预测性会降低。

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：

6.9.1 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。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：



从上面的输出可以看出，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。

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。

6.9.2 一次性触发器

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。

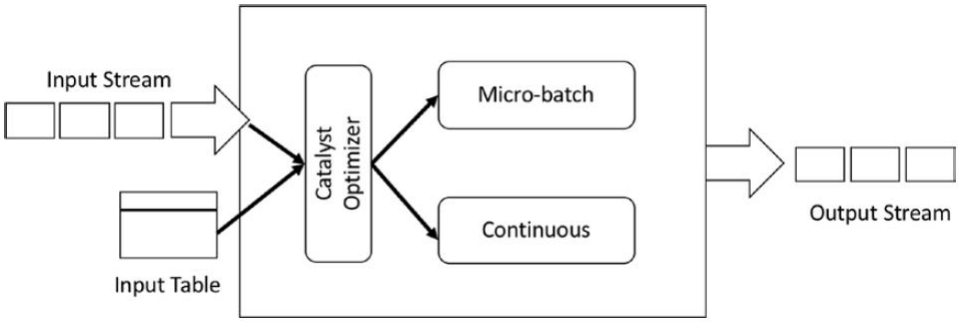
指定这个一次性触发器类型非常简单。下面的代码演示了如何使用一次性的触发器类型：

6.9.3 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。



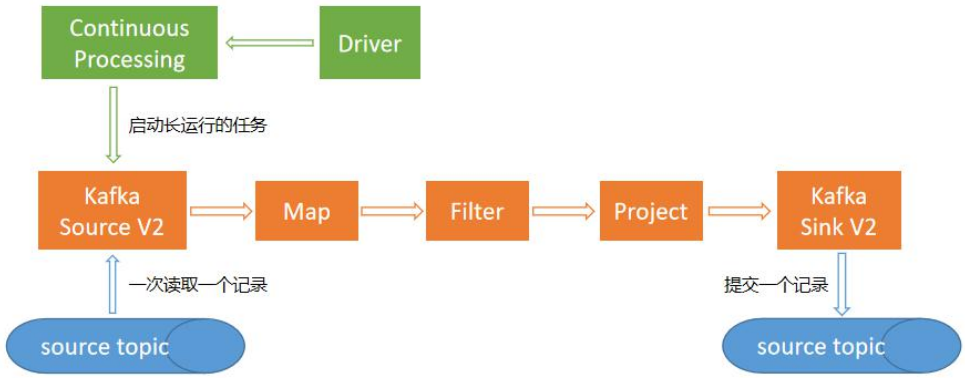
Spark 一直通过微批处理提供流处理能力，这种方法的主要缺点是每个任务/微批处理必须定期收集和调度，通过这种方式 Spark 可以提供的最佳(最小)延迟大约是 1 秒。不存在单一事件/消息处理的概念。Spark 试图通过连续处理来克服这些限制，以提供低延迟的流处理。

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一称为 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)。

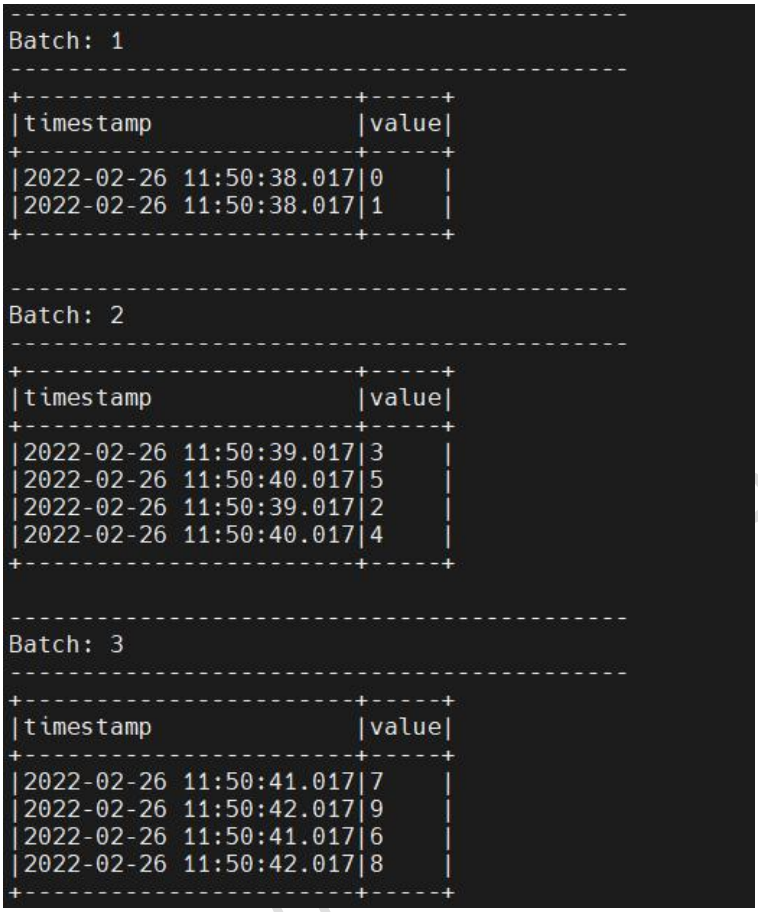
在持续处理模式中只有投影和选择操作是允许的，如 select、where、map、flatmap 和 filter。在这种处理模式下，除了聚合函数之外，所有 PySpark SQL 函数都是受支持的。另外需要特别注意的是，在连续处理中不支持水印，因为这涉及到收集数据。

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：



上面代码中的流 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。

6.10 小结

结构化流是 Apache Spark 的第二代流处理引擎。它提供了一种简单的方法来构建容错和可伸缩的流应用程序。这一章涵盖了很多领域，包括流处理领域的核心概念和结构化流的核心部分。

- 流处理是一个令人兴奋的领域，它可以帮助解决大数据时代的许多新的有趣的用例。
- 构建生产环境下的流数据应用程序要比构建批处理数据处理应用程序更具挑战性，因为无界数据的性质和数据到达率和无序数据的不可预测性。
- 为了有效地构建流数据应用程序，必须熟悉流处理领域中的三个核心概念，分别是数据传递语义、时间概念和窗口。
- 现在有很多可供挑选的流处理引擎。最受欢迎的是 Apache Flink、Apache Samza、Apache Kafka 和 Apache Spark。
- 结构化的流处理引擎是为开发人员设计的，目的是构建端到端的流应用程序，可以使用基于 PySpark SQL 引擎优化和坚实基础之上的简单编程模型实时地对数据做出反应。
- 结构化流的独特想法是将流数据视为一个无界的输入表，并且随着一组新数据的到来，将其视为附加到输入表的一组新行。
- 流查询中的核心组件是数据源、流操作、输出模式、触发器和数据接收器（data sink）。
- 结构化流提供了一组内置的数据源和数据接收器。内置的数据源是 File、Kafka、Socket 和 Rate。

内置的数据接收器是 File、Kafka、Console 和 Memory。

- 输出模式决定数据如何输出到数据接收器，有三个选项：Append、Update 和 Complete。
- 触发器是一个结构流引擎的机制，用来决定何时运行流计算。有几种选项可供选择：micro-batch、fixed interval micro-batch、one-time micro-batch 和 continuous。最后一个是用于要求毫秒级延迟的场景，它目前处于实验状态。

第 7 章 PySpark 结构化流（高级）

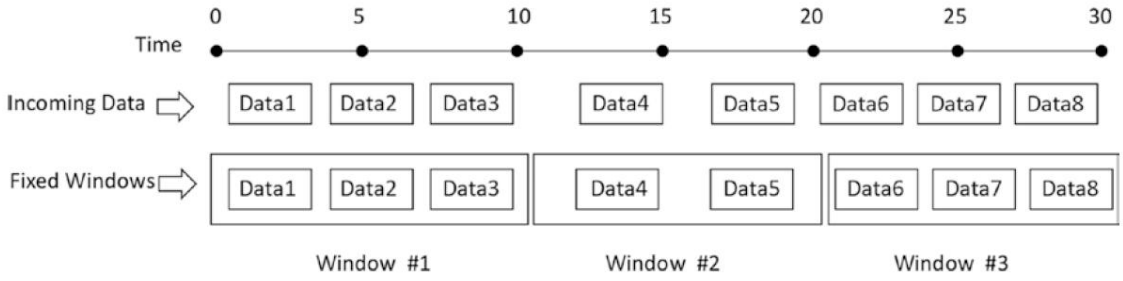
前一章介绍了流处理的核心概念，PySpark 结构化流处理引擎提供的特性，以及将流应用程序组合在一起的基本步骤。本章将涵盖结构化流的事件时间（event-time）处理和有状态处理特性，并解释结构化流提供的支持，以帮助流应用程序对故障进行容错，并监控流应用程序的状态和进展。

7.1 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。

7.1.1 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 PySpark 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。

分析：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：

注：这四个 JSON 文件位于 PBLP 平台的/home/hduser/data/spark/stream/mobile3/目录下。

2）编辑源代码，内容如下。

3）执行流处理程序，可以看到输出的 Schema 如下：

```
root
 |-- window: struct (nullable = false)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- count: long (nullable = false)
```

输出的统计结果如下：

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-------------------+-------------------+-----+
|start              |end                |count|
+-------------------+-------------------+-----+
|2018-03-02 10:00:00|2018-03-02 10:10:00|4    |
+-------------------+-------------------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+-------------------+-------------------+-----+
|start              |end                |count|
+-------------------+-------------------+-----+
|2018-03-02 10:00:00|2018-03-02 10:10:00|6    |
+-------------------+-------------------+-----+

-------------------------------------------
Batch: 2
-------------------------------------------
+-------------------+-------------------+-----+
|start              |end                |count|
+-------------------+-------------------+-----+
|2018-03-02 10:00:00|2018-03-02 10:10:00|7    |
|2018-03-02 10:10:00|2018-03-02 10:20:00|1    |
+-------------------+-------------------+-----+

-------------------------------------------
Batch: 3
-------------------------------------------
+-------------------+-------------------+-----+
|start              |end                |count|
+-------------------+-------------------+-----+
|2018-03-02 10:00:00|2018-03-02 10:10:00|7    |
|2018-03-02 10:10:00|2018-03-02 10:20:00|1    |
|2018-03-02 11:00:00|2018-03-02 11:10:00|1    |
|2018-03-02 11:10:00|2018-03-02 11:20:00|1    |
+-------------------+-------------------+-----+
```

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：

执行流处理程序，可以看到输出的 Schema 如下：

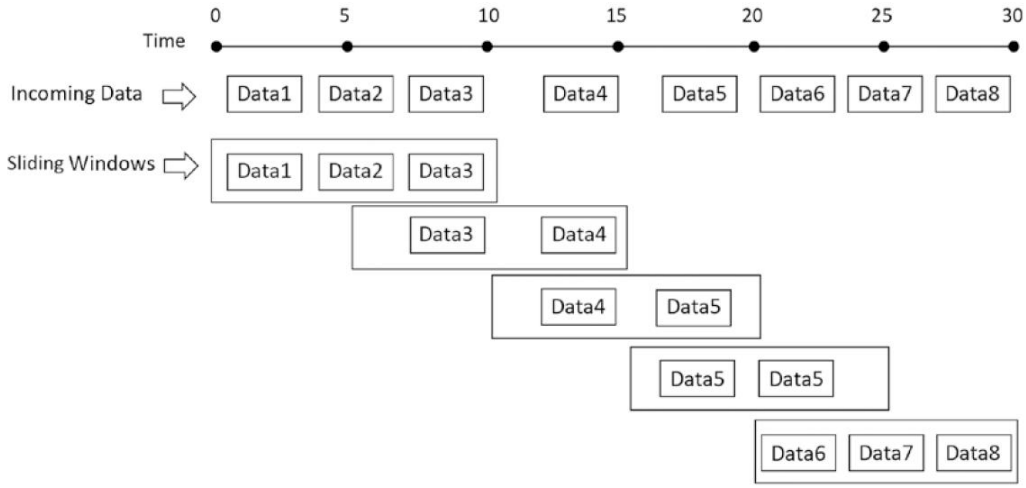
```
root
 |-- window: struct (nullable = false)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- action: string (nullable = true)
 |-- count: long (nullable = false)
```

输出的统计结果如下：

```
>>> -------------------------------------------
Batch: 0
-------------------------------------------
+-------------------+-------------------+------+-----+
|start              |end                |action|count|
+-------------------+-------------------+------+-----+
|2018-03-02 10:00:00|2018-03-02 10:10:00|open  |3    |
|2018-03-02 10:00:00|2018-03-02 10:10:00|close |1    |
+-------------------+-------------------+------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+-------------------+-------------------+------+-----+
|start              |end                |action|count|
+-------------------+-------------------+------+-----+
|2018-03-02 10:00:00|2018-03-02 10:10:00|open  |4    |
|2018-03-02 10:00:00|2018-03-02 10:10:00|close |2    |
+-------------------+-------------------+------+-----+

-------------------------------------------
Batch: 2
-------------------------------------------
+-------------------+-------------------+------+-----+
|start              |end                |action|count|
+-------------------+-------------------+------+-----+
|2018-03-02 10:00:00|2018-03-02 10:10:00|open  |4    |
|2018-03-02 10:00:00|2018-03-02 10:10:00|close |3    |
|2018-03-02 10:10:00|2018-03-02 10:20:00|open  |1    |
+-------------------+-------------------+------+-----+

-------------------------------------------
Batch: 3
-------------------------------------------
+-------------------+-------------------+------+-----+
|start              |end                |action|count|
+-------------------+-------------------+------+-----+
|2018-03-02 10:00:00|2018-03-02 10:10:00|open  |4    |
|2018-03-02 10:00:00|2018-03-02 10:10:00|close |3    |
|2018-03-02 10:10:00|2018-03-02 10:20:00|open  |1    |
|2018-03-02 11:00:00|2018-03-02 11:10:00|crash |1    |
|2018-03-02 11:10:00|2018-03-02 11:20:00|swipe |1    |
+-------------------+-------------------+------+-----+
```

7.1.2 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：

注：这两个 JSON 文件位于 PBLP 平台的/home/hduser/data/spark/stream/iot2/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）执行流处理程序，可以看到输出的 Schema 如下：

```
root
 |-- window: struct (nullable = true)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- avg_temp: double (nullable = true)
```

输出的统计结果如下：

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-------------------+-------------------+--------+
|start              |end                |avg_temp|
+-------------------+-------------------+--------+
|2017-06-02 07:55:00|2017-06-02 08:05:00|99.5    |
|2017-06-02 08:00:00|2017-06-02 08:10:00|101.25  |
|2017-06-02 08:05:00|2017-06-02 08:15:00|102.75  |
|2017-06-02 08:10:00|2017-06-02 08:20:00|103.75  |
|2017-06-02 08:15:00|2017-06-02 08:25:00|105.0   |
+-------------------+-------------------+--------+

query.stop()
```

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，机架平均温度的上升，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）：

执行流处理程序，可以看到输出的 Schema 如下：

```
root
 |-- window: struct (nullable = true)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- rack: string (nullable = true)
 |-- avg_temp: double (nullable = true)
```

输出的统计结果如下：

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-----+-------------------+-------------------+--------+
|rack |start              |end                |avg_temp|
+-----+-------------------+-------------------+--------+
|rack1|2017-06-02 07:55:00|2017-06-02 08:05:00|99.5    |
|rack1|2017-06-02 08:00:00|2017-06-02 08:10:00|100.0   |
|rack1|2017-06-02 08:05:00|2017-06-02 08:15:00|100.75  |
|rack1|2017-06-02 08:10:00|2017-06-02 08:20:00|101.5   |
|rack1|2017-06-02 08:15:00|2017-06-02 08:25:00|102.0   |
|rack2|2017-06-02 07:55:00|2017-06-02 08:05:00|99.5    |
|rack2|2017-06-02 08:00:00|2017-06-02 08:10:00|102.5   |
|rack2|2017-06-02 08:05:00|2017-06-02 08:15:00|104.75  |
|rack2|2017-06-02 08:10:00|2017-06-02 08:20:00|106.0   |
|rack2|2017-06-02 08:15:00|2017-06-02 08:25:00|108.0   |
+-----+-------------------+-------------------+--------+

query.stop()
```

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。

7.2 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。

7.2.1 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 PySpark 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 PySpark 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。

事实上，任何时候在流查询上执行聚合时，都必须维护中间聚合状态。这个状态保存在 key-value 对结构中，类似于散列映射（hash map），其中 key 是 group name，value 是中间聚合值。在上一节的例子中，通过滑动窗口和机架 ID 进行聚合，其中 key 就是窗口的开始和结束时间以及机架名称的组合值，而 value 则是平均温度。

中间状态存储在 Spark executors 的内存中、版本化的 key-value 状态存储中，并将其写到一个预写日志中（该日志应该被配置为驻留在像 HDFS 这样的稳定存储系统中）。在每个触发点上，该状态都在内存中的状态存储中读取和更新，然后写入到预写日志中。在失败的情况下，当 Spark 结构化的流应用程序重新启动时，状态从预写日志中还原，从那个点恢复。这种容错的状态管理显然会在结构化流引擎中产生一些资源和处理开销。因此，开销的大小与它需要维护的状态量成正比，因此，保持状态的数量在一个可以接受大小是很重要的；换句话说，状态的规模不应该无限增长。

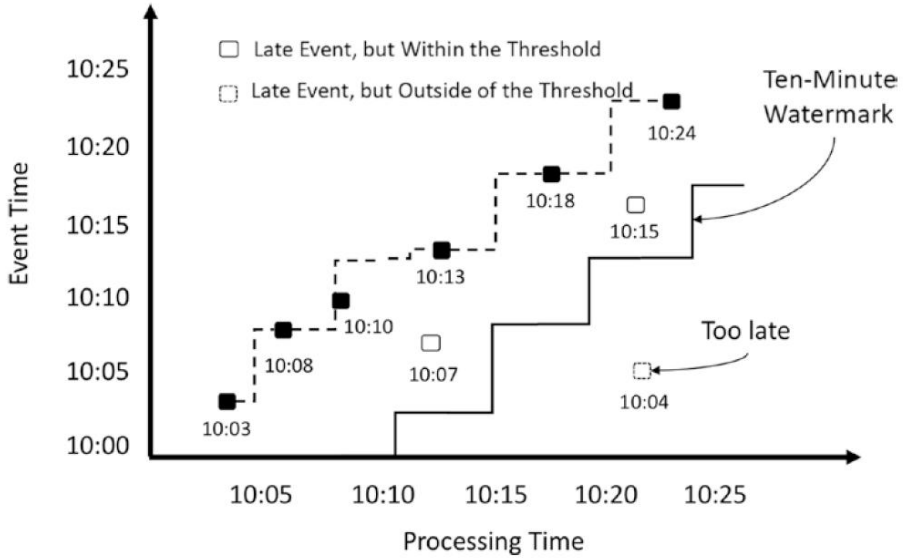
考虑到滑动窗口的性质，窗口的数量将会无限增长。这意味着执行滑动窗口聚合会导致中间状态无限地增长，因此，必须有一种方法可以删除不再更新的旧状态。在 Spark 结构化流处理技术中，这是通过一种叫做“水印（watermarking）”的技术完成的。

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**。

7.2.2 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：

注：这两个 JSON 文件位于 PBLP 平台的/home/hduser/data/spark/stream/mobile4/目录下。

2）代码编写。

实现的流查询代码如下：

3）执行流处理程序，可以看到输出的 Schema 如下：

```
root
 |-- window: struct (nullable = false)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- action: string (nullable = true)
 |-- count: long (nullable = false)
```

当它读取到第一个流数据文件 file1.json 时，输出结果如下。

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-------------------+-------------------+------+-----+
|start              |end                |action|count|
+-------------------+-------------------+------+-----+
|2018-03-02 10:10:00|2018-03-02 10:20:00|open  |1    |
|2018-03-02 10:20:00|2018-03-02 10:30:00|open  |1    |
|2018-03-02 10:30:00|2018-03-02 10:40:00|open  |1    |
+-------------------+-------------------+------+-----+
```

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。

```
-------------------------------------------
Batch: 1
-------------------------------------------
+-------------------+-------------------+------+-----+
|start              |end                |action|count|
+-------------------+-------------------+------+-----+
|2018-03-02 10:20:00|2018-03-02 10:30:00|open  |2    |
+-------------------+-------------------+------+-----+
```

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，如下 ：

那么输出结果如下所示。

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-------------------+-------------------+------+-----+
|start              |end                |action|count|
+-------------------+-------------------+------+-----+
|2018-03-02 10:10:00|2018-03-02 10:20:00|open  |1    |
|2018-03-02 10:20:00|2018-03-02 10:30:00|open  |1    |
|2018-03-02 10:30:00|2018-03-02 10:40:00|open  |1    |
+-------------------+-------------------+------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+-------------------+-------------------+------+-----+
|start              |end                |action|count|
+-------------------+-------------------+------+-----+
|2018-03-02 10:10:00|2018-03-02 10:20:00|open  |2    |
|2018-03-02 10:20:00|2018-03-02 10:30:00|open  |2    |
+-------------------+-------------------+------+-----+
```

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？）

水印是一个有用的特性，因此理解聚合状态被正确清理的条件是很重要的：

- 输出模式不能是 complete 模式，必须是 update 或 append 模式。原因是，complete 模式的语义规定必须维护所有聚合数据，并且不违反这些语义，水印不能删除任何中间状态。
- 通过 groupBy 转换的聚合必须直接位于 event-time 列或 event-time 列的窗口上。
- 在 Watermark API 和 groupBy 转换中指定的 event-time 列必须是同一个。

- 当设置一个流 DataFrame 时，必须在调用 groupBy 转换之前调用 Watermark API；否则，它将被忽略。

7.3 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，PySpark 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，在流 DataFrame 上调用 dropDuplicates 方法即可。该方法只有一个输入参数，该输入参数是用来标识每一行的惟一列名的列表。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：

注：这两个 JSON 文件位于 PBLP 平台的/home/hduser/data/spark/stream/mobile5/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：

```
-------------------------------------------
Batch: 0
-------------------------------------------
+------+-----+
|id    |count|
+------+-----+
|phone1|1    |
|phone2|1    |
|phone3|1    |
+------+-----+
```

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。

7.4 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。

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。

流应用程序有可能会随着时间的推移而发生变化，因为需要改进处理逻辑或性能或者修复 bug，这可能会如何影响在检查点位置保存的信息。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。

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。

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。

7.5 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。

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，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
{
  "id" : "9ba6691d-7612-4906-b64d-9153544d81e9",
  "runId" : "c6d79bee-a691-4d2f-9be2-c93f3a88eb0c",
  "name" : null,
  "timestamp" : "2018-04-23T17:20:12.023Z",
  "batchId" : 0,
  "numInputRows" : 3,
  "inputRowsPerSecond" : 250.0,
  "processedRowsPerSecond" : 1.728110599078341,
  "durationMs" : {
    "addBatch" : 1548,
    "getBatch" : 8,
    "getOffset" : 36,
    "queryPlanning" : 110,
    "triggerExecution" : 1736,
    "walCommit" : 26
  },
  "eventTime" : {
    "avg" : "2017-09-06T15:10:04.666Z",
    "max" : "2017-09-06T15:11:10.000Z",
    "min" : "2017-09-06T15:08:53.000Z",
    "watermark" : "1970-01-01T00:00:00.000Z"
  },
  "stateOperators" : [ {
```

```
    "numRowsTotal" : 1,
    "numRowsUpdated" : 1,
    "memoryUsedBytes" : 16127
  } ],
  "sources" : [ {
    "description" : "FileStreamSource[file:<path>/data/input]",
    "startOffset" : null,
    "endOffset" : {
      "logOffset" : 0
    },
    "numInputRows" : 3,
    "inputRowsPerSecond" : 250.0,
    "processedRowsPerSecond" : 1.728110599078341
  } ],
  "sink" : {
    "description" : "org.apache.spark.sql.execution.streaming.
    ConsoleSinkProvider@37dc4031"
  }
}
```


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。

7.6 案例：运输公司车辆超速实时监测

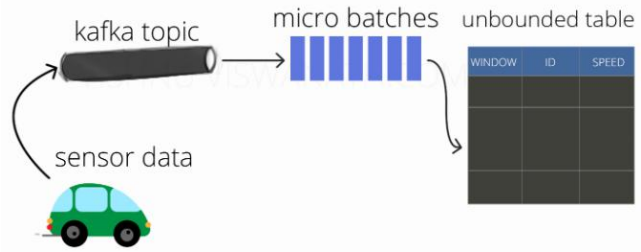
让我们想象一个物流公司运输车队管理解决方案，其中车队中的车辆启用了无线网络功能。每辆车定期报告其地理位置和许多操作参数，如燃油水平、速度、加速度、轴承、发动机温度等。公司管理层希望利用这一遥测数据流来实现一系列应用程序，以帮助他们在业务的运营和财务方面提高管理效率。

使用到目前为止我们所知道的 PySpark 结构化流特性，我们可以实现其中许多用例，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。下面我们要实现的是其中一个功能，即监控运输车辆是否超速。

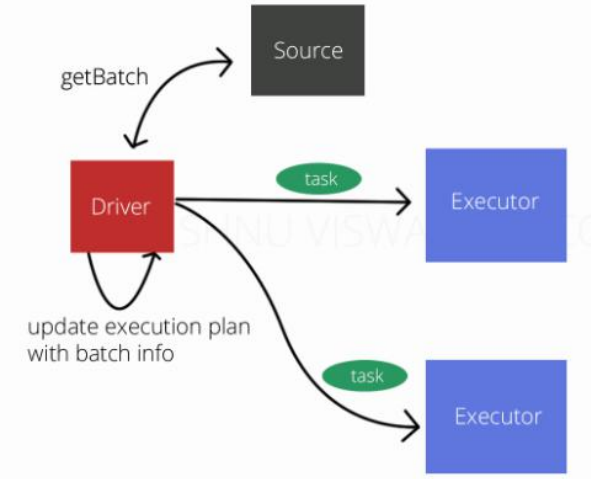
我们将创建一个简单的近实时的 PySpark 结构化流应用程序来计算车辆每几秒钟的平均速度，并且将超速信息发送到下游。在本例中，我们将集成 Kafka 作为流程序的数据源和 Data Sink，如下图所示：

7.6.1 实现技术剖析

PySpark 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。在结构化流中，微批数据被构造为一个流式的 DataFrame（即无界表）。

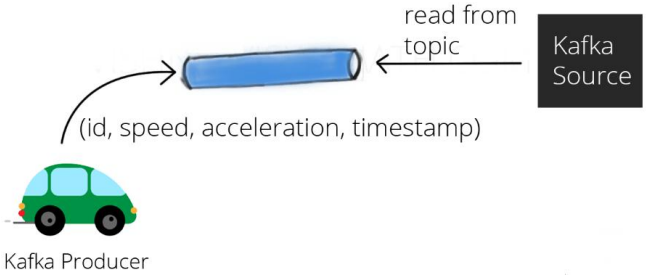


这个微批处理然后由驱动程序调度，作为任务在执行器中执行。完成微批处理执行后，将再次收集并调度下一批。这种调度是经常进行的，以给人以流执行的印象。



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，我们提供一个已经创建好的 Kafka 生产者程序 fastcars_fat.jar（这是一个 Java 程序执行 jar 包，它位于 PBLP 平台的/home/hduser/jars/路径下），它将 id、speed、acceleration 和 timestamp 写入 Kafka 的“cars”主题。注意，这里的 timestamp 时间戳是在事件源处生成事件(消息)的时间，即代表事件时间。



接下来，我们对从 Kafka cars 主题拉取的原始数据解析解析，这样我们就有了一个可以使用的结构。

这会产生一个如下结构的流 DataFrame：

```
>>> convertedDF.printSchema()
root
 |-- car_id: string (nullable = true)
 |-- speed: integer (nullable = true)
 |-- acceleration: double (nullable = true)
 |-- ts: timestamp (nullable = true)
```

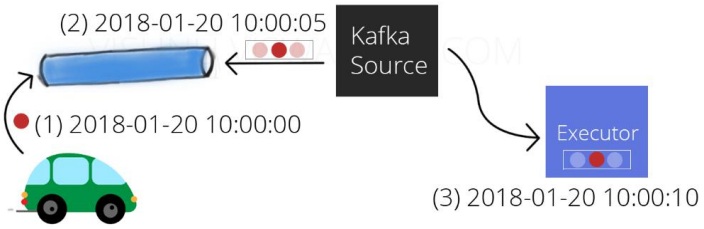
执行聚合

我们从求每辆车的平均速度开始。这可以通过对 car_id 执行 groupBy 并应用 avg 聚合函数来实现。

在结构化流程序中，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。

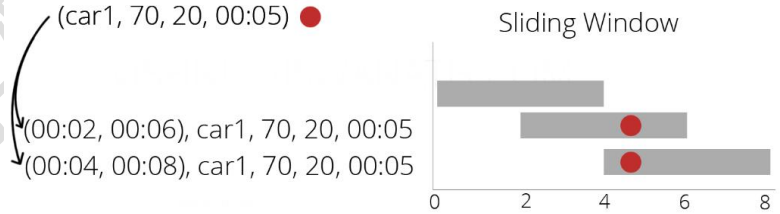
事件时间和处理时间

事件时间（EventTime）是在数据源处生成事件的时间，而处理时间（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。



这里我们需要计算车辆在过去 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 PySpark 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，它的事件时间（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，PySpark 添加了一个名为“window”的新列，并将提供的“ts”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 car_id 对 convertedDF 数据进行分组。注意，window()是 PySpark 中的一个函数，它返回一个列。

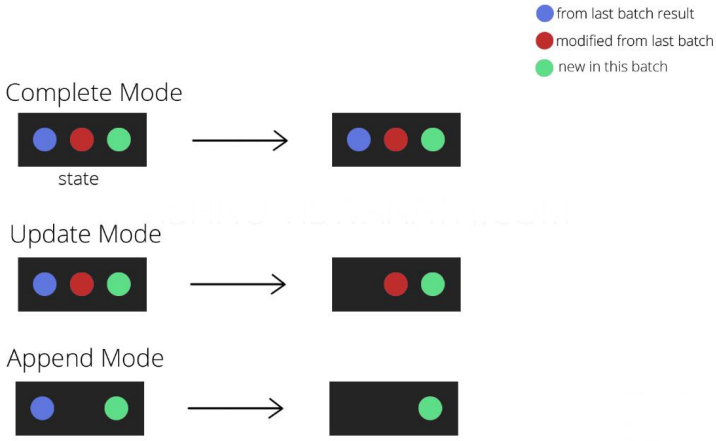
使用下面的方法定义大小为 4 秒、滑动为 2 秒的滑动窗口：

这将产生一个 car_id、平均速度和相应时间窗口的 DataFrame。输出如下所示：

```
-------------------------------------------
Batch: 17
-------------------------------------------
+------------------------------------------+------+-----+
|window                                    |car_id|speed|
+------------------------------------------+------+-----+
|{2022-02-27 11:44:24, 2022-02-27 11:44:28}|car3  |108.0|
|{2022-02-27 11:44:24, 2022-02-27 11:44:28}|car6  |112.0|
+------------------------------------------+------+-----+
```

输出结果到一个 Kafka 主题

PySpark 提供了三种输出模式—complete、update 和 append。在处理微批处理后，PySpark 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，PySpark 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：

```
[hduser@xueai8 kafka_2.12-2.4.1]$ bin/kafka-console-consumer.sh --topic fastcars --bootstrap-server xueai8:9092
car0, 119.0
car1, 111.0
car0, 107.0
car2, 111.0
car0, 107.0
car1, 116.0
```

检测到的超速车辆的编号和速度信息

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。

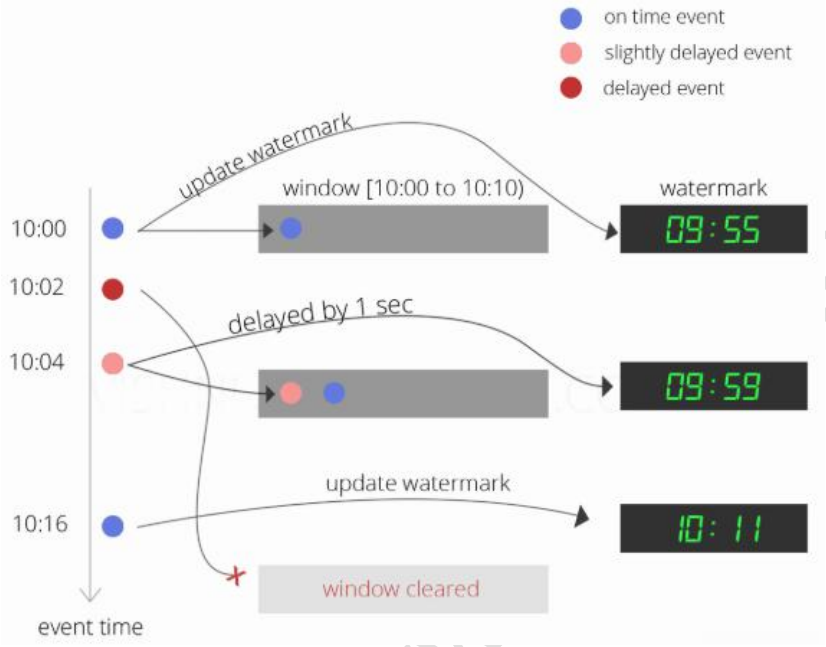
在 PySpark 中，水印用于根据当前最大事件时间决定何时清除状态。基于我们所指定的延迟，水印滞后于目前所看到的最大事件时间。例如，如果 dealy 是 3 秒，当前最大事件时间是 10:00:45，那么水印是在 10:00:42。这意味着 Spark 将保持结束时间小于 10:00:42 的窗口的状态。

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会如何去处理。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("ts", "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：

- 一个时间戳为 10:00 的事件到达并落在窗口[10:00,10:10]，水印被更新为 timestamp - 5。
- 在源处生成时间戳为 10:02 的事件，但会被延迟。这个事件应该落在窗口[10:00,10:10]。
- 时间戳为 10:04 的事件在 10:05 较晚到达，但它仍然落在窗口[10:00,10:10)，因为当前水印为 09:55，即小于窗口结束时间。水印更新到 10:04 - 00:05 = 09:59。
- 一个时间戳为 10:16 的事件到达，它将水印更新为 10:11。(此事件将落在窗口[10:10,10:20)，但在这里不相关)。
- 带有时间戳 10:02 的延迟事件到达，但是窗口[10:00,10:10)被清除，因此该事件将被删除。

设置水印将确保状态不会永远增长。另外，请注意一个延迟事件是如何处理的，而另一个是如何被忽略了(因为已经太迟了)。

7.6.2 完整实现代码

下面给出本案例的完整代码，如下。

7.6.3 执行步骤演示

本案例涉及到与 Kafka 的集成，以及使用自定义的数据源程序，所以执行步骤稍稍有点复杂，请按以下说明操作。

测试数据源

本案例使用 PBLP 平台内置的 Kafka 生产者程序。该程序是以 Java 开发、编译和打包的扫行 jar 包，它位于 PBLP 平台的/home/hduser/jars/路径下，名为 fastcars_fat.jar，在 PBLP 平台上可使用 jar 命令来运行它。请按以下步骤测试和掌握该数据源程序的使用。

1）启动 zookeeper。在第一个终端窗口，执行以下命令：

2）接下来，启动 Kafka 服务器。在第二个终端窗口，执行以下命令：

3）创建主题 cars。在第三个终端窗口，执行以下命令：

4）运行 kafka 消费者脚本，指定要连接的 Kafka 服务器名称和端口号，以及要连接的主题。继续在第三个终端窗口，执行以下命令：

5）运行模拟数据源程序 jar 包。打开第四个终端窗口，执行以下命令：

其中参数 xueai8:9092 是要连接的 Kafka 服务器名称和端口号，参数 cars 是连接的主题。这两个参数可以根据自己的需要修改，但是要和消费者监听的服务器和主题一致。

执行过程如下图所示：

6）回到第三个终端窗口（运行消费者脚本的窗口），可以看到输出了接收到的数据信息，如下图所示：

7）如果要终止数据源程序或消费者脚本的运行，同时按 Ctrl + C 键即可。

执行流程序，实时监测超速信息

接下来，我们就可以运行上节开发的 PySpark 结构化流程序，来实时监测运输车辆的车速，并及时将监测到的超速车辆信息发送到 Kafka 的 fastcars 主题。请按以下步骤操作。

1）启动 zookeeper（如果已经启动，略过此步骤）。在第一个终端窗口，执行以下命令：

2）接下来，启动 Kafka 服务器（如果已经启动，略过此步骤）。在第二个终端窗口，执行以下命令：

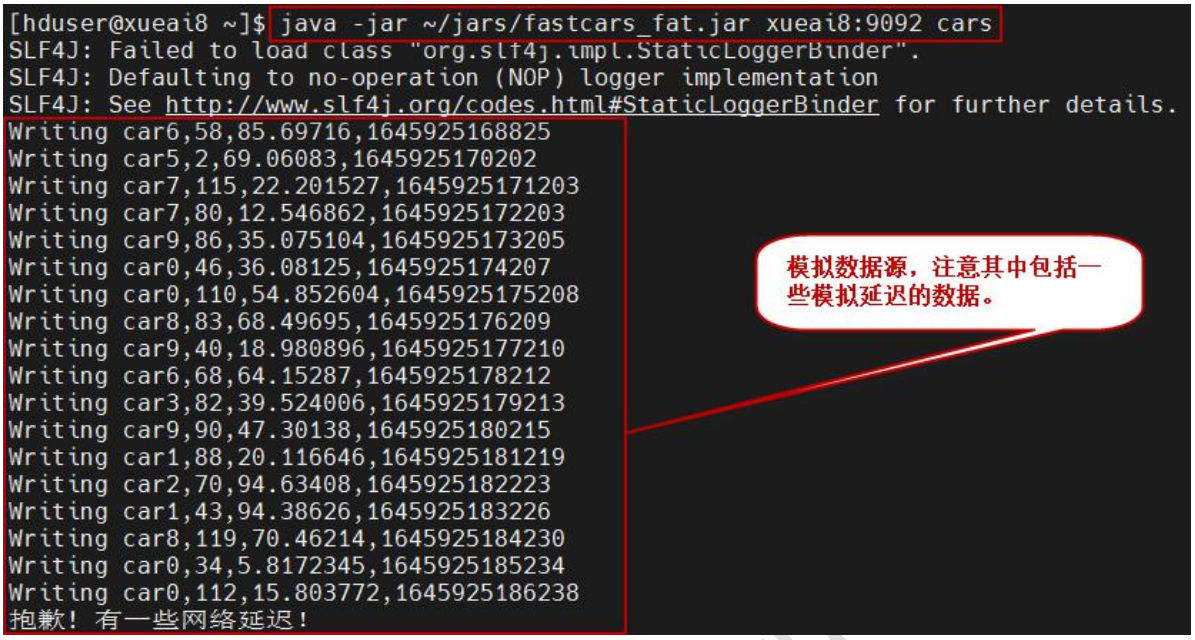
3）创建主题 fastcars（如果已经创建，略过此步骤）。在第三个终端窗口，执行以下命令：

4）运行 kafka 消费者脚本，监听超速车辆信息。继续在第三个终端窗口，执行以下命令：

5）运行流程序。打开第四个终端窗口，启动 pyspark shell，运行上节开发的流程序代码。（也可以保存到.py 文件中，使用 spark-submit 命令提交执行）

6）运行模拟数据源程序 jar 包。打开第五个终端窗口，执行以下命令：

执行过程如下图所示：



7）回到第三个终端窗口，如果一节正常，那么可以看到接收到的超速信息，如下图所示：

8）如果要结束程序的运行，按以下顺序关闭：

先关闭数据源程序，Ctrl + C；

再关闭流处理程序，键入 exit()函数，回车；

然后关闭消费者脚本程序，Ctrl + C；

先关闭 Kafka 集群，Ctrl + C；

最后关闭 Zookeeper 集群，Ctrl + C。

7.7 小结

PySpark 结构化的流引擎提供了许多高级功能和灵活性，可以构建复杂而精巧的流应用程序。

- 任何严肃的流处理引擎都必须支持按事件发生时处理传入数据的能力。结构化流不仅支持这样做的能力，而且还支持基于固定和滑动窗口的窗口聚合。此外，它还将以容错的方式自动维护中间状态。
- 当流应用程序处理越来越多的数据时，维护中间状态会带来内存耗尽的风险。引入了一个水印，使人们更容易对迟到数据进行推理，并删除不再需要的中间状态。
- 任意的有状态处理支持一种用户定义的方法来处理各组的值并保持它的中间状态。结构化流提供了一种通过回调 API 来实现这一功能的简单方法，并且可以灵活地在输出中每个 group 生成一个或多个行。
- 结构化流提供了一个端到端的、exactly-once 的保证。这是通过使用检查点和预写日志机制实现的。通过提供驻留在容错文件系统上的检查点位置，可以轻松地打开它们。通过读取保存在检查点位置的信息，可以很容易地重新启动流应用程序，并在故障发生前从它们中断的地方恢复。

- 生产环境下的流应用程序需要能够深入了解流查询的状态和指标。结构化流提供了流查询状态的简短摘要，以及关于传入数据速率、处理速率和一些关于中间状态内存消耗的详细信息。为了监视所有流查询的生命周期及其详细的进展，您可以注册 StreamingQueryListener 接口的一个或多个实例。

第 8 章 PySpark 大数据分析综合案例

本章的综合案例涉及数据的采集（使用爬虫程序）、数据集成、数据预处理、大数据存储、Hive 数据仓库应用、大数据 ELT 实现和大数据结果展现等全流程所涉及的各种典型操作，涵盖 Linux、MySQL、Hadoop 3、Flume、PySpark 3.x.x、SparkMlib、IntelliJ IDEA（简称 IDEA）、Spring MVC 框架、ECharts 组件等系统和软件的使用方法。通过本项目，将有助于读者综合运用主流大数据技术以及各种工具软件，掌握大数据离线批处理的全流程操作。

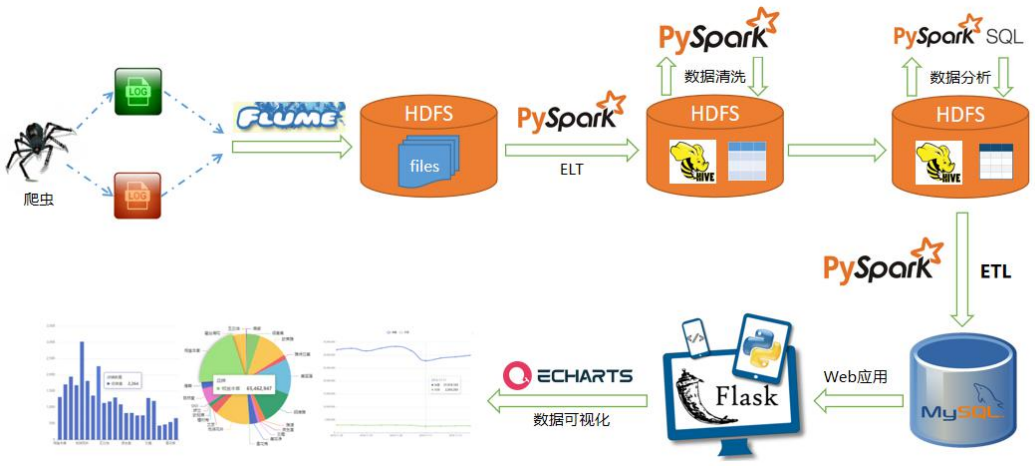
8.1 项目需求说明

综合运行大数据分析和可视技术，对使用爬虫程序从互联网上采集到的某招聘网站招聘岗位数据进行多维度分析，并可视化展示分析结果。

。。。。。。

8.2 项目架构设计

本项目架构流程如下：



项目流程：

1. 数据采集：使用 Python 爬虫程序；
2. 数据集成：使用 Flume 自动监测并导入采集到的数据文件到 HDFS 中存储；
3. 数据 ELT：使用 PySpark 建立 ELT 管道，将集成到的数据文件导入到 Hive 数据仓库中 ODS 层；
4. 数据清洗：使用 PySpark + Hive 进行数据清洗和整理；
5. 数据分析：使用 PySpark SQL + Hive 进行数据多维度分析；
6. 数据 ETL：使用 PySpark 建立 ETL 管道，将分析结果导出到 MySQL 数据库；
7. 数据可视化：使用 Python Flask + ECharts 实现分析结果网页可视展示。

8.3 项目实现-数据采集

编写 Python 爬虫程序，从 51job 招聘网站上爬取 5 个热门城市（北京、上海、广州、深圳、杭州）的大数据相关岗位（大数据、数据分析、Java、Python）的招聘信息。

本案例我们提供了两套实现方案。一个是使用 requests 库来编写爬虫程序，另一个是使用 Scrapy 框架来编写爬虫程序。请大家根据自己的需要选择其中一个即可。

8.3.1 爬虫程序实现：使用 requests 库

。。。。。。

8.3.2 爬虫程序实现：使用 Scrapy 框架

。。。。。。

8.4 项目实现-数据集成

本步骤使用 Flume 自动监测并导入采集到的数据文件到 HDFS 中存储。

8.4.1 Flume 简介

。。。

8.4.2 安装和配置 Flume

。。。

。。。

8.4.3 实现数据集成

。。。

8.5 项目实现-数据 ELT

使用 PySpark 建立 ELT 管道，将集成到 HDFS 中的数据文件导入到 Hive 数据仓库中 ODS 层；

PySpark SQL 内置提供了对众多常用数据源的支持，可以很容易地使用它来构建 ELT 管道。下面是实现将数据从 HDFS 存储 ELT 到 Hive 贴源层（ODS 层）的代码实现：

。。。。。。

8.6 项目实现-数据清洗与整理

使用 PySpark 对大数据进行清洗，包括去重、错误数据处理、空值处理、属性转换、属性提取等数据预处理任务。

```
```

。。。

8.7 项目实现-数据分析

使用 PySpark SQL + Hive 从多个维度对整理后的数据集进行分析，并将分析结果存入数据集市。

```
```

。。。

8.8 项目实现-数据 ETL

使用 PySpark 建立 ETL 管道，将 Hive 数据集市中的分析结果导出到 MySQL 数据库中。

PySpark SQL 内置提供了对众多常用数据源的支持，可以很容易地使用它来构建 ETL 管道。下面是实现将数据从 Hive 数据集市 ETL 到 MySQL 的代码实现：

```
```

。。。

8.9 项目实现-数据可视化

开发 Python Flask Web 项目，使用 ECharts 作为可视化组件，展示分析结果。

```
```

。。。