# 简介

## Scala简介

Scala是一门多范式的编程语言，一种类似java的编程语言，设计初衷是实现可伸缩的语言、并集成面向对象编程和函数式编程的各种特性。

## Spark简介

### Spark概述

Apache Spark是一个基于内存计算的分布式计算框架，旨在简化运行于计算机集群上的并行程序的编写。该框架对资源调度，任务的提交、执行和跟踪，节点间的通信以及数据并行处理的内在底层操作都进行了抽象。它提供了一个更高级别的API用于处理分布式数据。从这方面说，它与Apache Hadoop等分布式处理框架类似。但在底层架构上，Spark与它们有所不同。

Spark支持四种运行模式。

* 本地单机模式：所有Spark进程都运行在同一个Java虚拟机（Java Vitural Machine，JVM）中。Spark能通过内置的单机集群调度器来在本地运行。此时，所有的Spark进程运行在同一个Java虚拟机中。这实际上构造了一个独立、多线程版本的Spark环境。本地模式很适合程序的原型设计、开发、调试及测试。同样，它也适应于在单机上进行多核并行计算的实际场景。Spark的本地模式与集群模式完全兼容，本地编写和测试过的程序仅需增加少许设置便能在集群上运行。
* 集群单机模式：使用Spark自己内置的任务调度框架。
* 基于Mesos：Mesos是一个流行的开源集群计算框架。
* 基于YARN：即Hadoop 2，它是一个与Hadoop关联的集群计算和资源调度框架。

基于Spark的基础平台扩展了5个主要的Spark库，包括支持结构化数据的Spark SQL、处理实时数据的Spark Streaming、用于机器学习的MLlib、用于图计算的GraphX、用于统计分析的SparkR，各种程序库与Spark核心API高度整合在一起，并在持续不断改进。

### Spark生态

Spark大数据计算平台包含许多子模块，构成了整个Spark的生态系统，其中Spark为核心。

伯克利将整个Spark的生态系统称为伯克利数据分析栈（BADS），其结构图如图1-1所示。

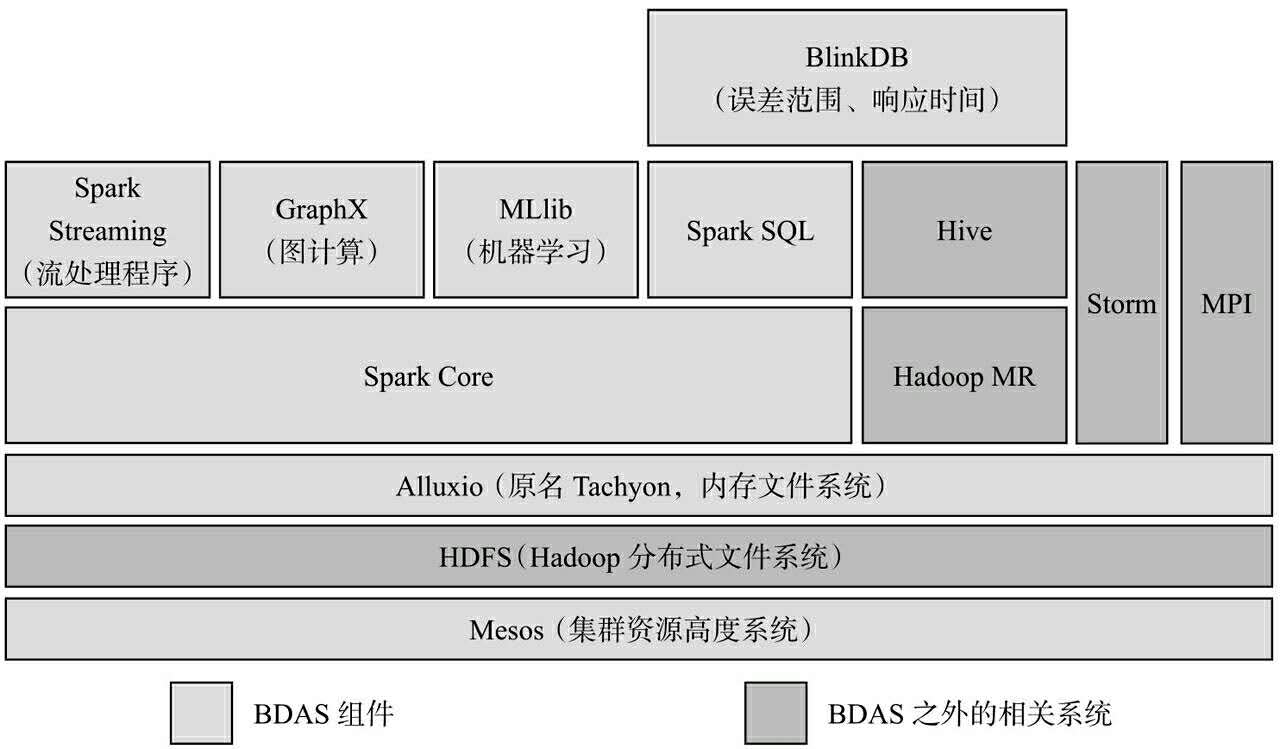


图 1‑1BADS

1. Spark Core

Spark Core是整个BDAS的核心组件，是一种大数据分布式处理框架，不仅实现了MapReduce的算子map函数和reduce函数及计算模型，还提供如filter、join、groupByKey等更丰富的算子。Spark将分布式数据抽象为弹性分布式数据集（RDD）、实现了应用任务调度、RPC、序列化和压缩，并为运行在其上的上层组件提供API。

1. Mesos

Mesos是Apache下的开源分布式资源管理框架，被称为分布式系统的内核，提供了类似YARN的功能，实现了高效的资源任务调度。

1. MLlib

MLlib是Spark对常用的机器学习算法的实现库，同时包括相关的测试和数据生成器。MLlib目前支持4种常见的机器学习问题：二元分类、回归、聚类以及协同过滤，还包括一个底层的梯度下降优化基础算法。

MLlib基于RDD，天生就可以与Spark SQL、GraphX、Spark Streaming无缝集成，MLlib是MLBase的一部分，MLBase通过边界定义，力图将MLBase打造成一个机器学习平台，让机器学习开发的门槛更低，让一些并不了解机器学习的用户也能方便地使用MLBase这个工具来处理自己的数据。

MLlib支持将本地向量和矩阵存储在单个机器中，也包括有一个或更多的RDD支持的分布式矩阵。在目前的实现中，本地向量和矩阵都是为公共接口服务的简单数据模式，MLlib使用了线性代数包Breeze。在监督学习中使用到的样本在MLlib中成为标记点。

Spark MLlib架构由底层基础、算法库和应用程序三部分构成。底层基础包括Spark的运行库、进行线性代数相关技术的矩阵库和向量库。算法库包括Spark MLlib实现的具体机器学习算法，以及为这些算法提供的各类评估方法；主要实现算法包括建立在广义线性回归模型的分类和回归，以及协同过滤、聚类和决策树。在最新的Spark 1.5.0版本中还新增了基于前馈神经网络的分类器算法MultilayerPerceptronClassif?ier(MLPC)，频繁项挖掘算法Pref?ixSpan、AssociationRules，实现Kolmogorov-Smirnov检验等等算法，随着版本的演进，算法库也会越来越强大。应用程序包括测试数据的生成以及外部数据的加载等功能。

Spark的ML库基于DataFrame提供高性能API，帮助用户创建和优化实用的机器学习流水线（pipeline），包括特征转换独有的Pipelines API。相比较MLlib，变化主要体现在：

* 从机器学习的Library开始转向构建一个机器学习工作流的系统，ML把整个机器学习的过程抽象成Pipeline，一个Pipeline是由多个Stage组成，每个Stage是Transformer或者Estimator。
* ML框架下所有的数据源都是基于DataFrame，所有模型也尽量都基于Spark的数据类型表示，ML的API操作也从RDD向DataFrame全面转变。

1. Spark Streaming

Spark Streaming是一种构建在Spark上的实时计算框架，它扩展了Spark处理大规模流式数据的能力。其吞吐量能够超远现有主流的流式处理框架Storm，并提供丰富的API用于流式数据计算。

Spark Streaming构建在Spark上处理Stream数据的框架，基本的原理是将Stream数据分成小的时间片断（几秒），以类似batch批量处理的方式来处理这小部分数据。

Spark Streaming属于核心Spark API的扩展，它支持高吞吐量和容错的实时流数据处理，它可以接受来自Kafka、Flume、Twitter、ZeroMQ或TCP Socket的数据源，使用复杂的算法表达和高级功能来进行处理，如Map、Reduce、Join、Window等，处理的结果数据能够存入文件系统、数据库。还可以直接使用内置的机器学习算法、图形处理算法来处理数据。

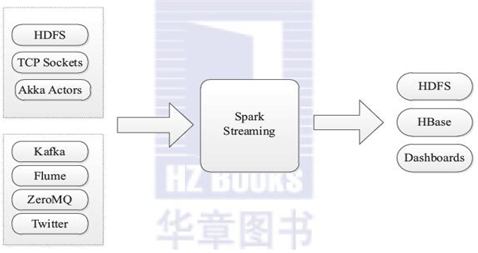


图 1‑2Spark数据流

Spark Streaming的数据处理流程，接收到实时数据后，先传给Spark Engine对数据进行分批次处理，最后生成该批次最后的结果。一个周期开始后，接收实时数据，然后进行处理，必须在一个周期内处理完，如果无法处理完，就需要将周期时间设置更长一点。本周期结束后，接着进行下一个周期，如此循环。

Spark Streaming是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是Spark Core，也就是把Spark Streaming的输入数据按照batch size（如1秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成Spark中的RDD（Resilient Distributed Dataset），然后将Spark Streaming中对DStream的Transformation操作变为针对Spark中对RDD的Transformation操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加或者存储到外部设备。



图 1‑3Spark计算流程

Spark Streaming提供一种名为离散流（DStream）的高级抽象连续数据流。DStream直接支持Kafka、Flume的数据源创建，或者通过高级操作其他DStream创建，一个DStream是一个序列化的RDD。

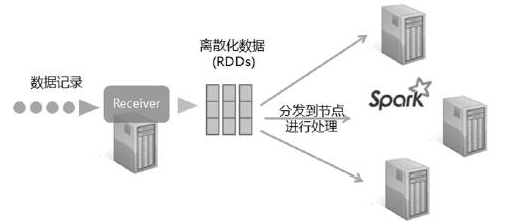


图 1‑4Spark计算流程

Spark为每个运行中的程序都提供了一个Web界面，用于监控程序的运行，Spark Streaming程序运行时，界面上会多出一个名为Streaming标签页，专用于流式计算。需要重点关注两个数据：Scheduling Delay和Processing Time。这两者加起来就是Spark Streaming一次周期计算的总时间。如果这两个时间在持续增加，或者两者之和超过了创建StreamingContext时设置的周期间隔时间，那很可能是集群资源紧张了，可以增加周期，或者优化集群或程序。

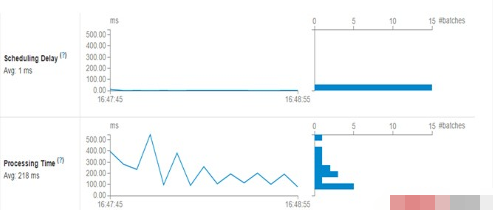


图 1‑5Spark页面监控

1. GraphX

GraphX是Spark中用于图和图并行计算的API，可以认为是GraphLab和Pregel在Spark上的重写及优化。

GraphX是用于图和并行图计算的新Spark API。从上层来看，GraphX通过引入弹性分布式属性图（resilient distributed property graph）扩展了Spark RDD。这种图是一种伪图，图中的每个边和节点都有对应的属性。

为了支持图计算，GraphX给出了一系列基础的操作（例如，subgraph、joinVertices、和MapReduceTriplets）以及基于Pregel API的优化变体。除此之外，GraphX还包含了一个不断扩展的图算法和构建器集合，以便简化图分析的任务。

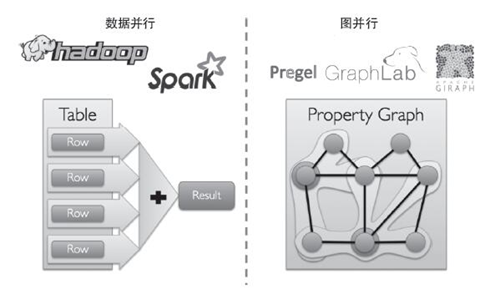


图 1‑6基于GraphX的并行图计算与其他方式的比较

1. Spark SQL

Shark是构建在Spark和Hive基础上的数据仓库。它提供了能够查询Hive中所存储数据的一套SQL接口，兼容现有的Hvie QL语法。熟悉Hive QL或者SQL的用户可以基于Shark进行快速的Ad-Hoc、Reporting等类型的SQL查询。2014年7月1日，Spark社区推出了Spark SQL，重新实现了SQL解析等原来Hive完成的工作，Spark SQL在功能上全覆盖了原有的Shark，具备更优秀的性能。

1. Alluxio

Alluxio是一个分布式内存文件系统，可以理解为内存中的HDFS。为了提供更高的性能、将数据存储剥离JAVA Heap。用户可以基于Alluxio实现RDD或者文件的跨应用共享，并提供高容错机制，保证数据的可靠性。

1. BlinkDB

BlinkDB是一个用于在海量数据上进行交互式SQL的近似查询引擎。它允许用户在查询准确性和查询响应时间之间做出权衡，执行相似查询。

### Spark架构

Spark架构采用了分布式计算中的Master-Slave模型。集群中运行Master进程的节点称为Master，同样，集群中包含有Worker进程的节点为Slave。Master负责控制整个集群的运行；Worker节点相当于分布式系统中的计算节点，它接收Master节点指令并返回计算进程到Master；Executor负责任务的执行；Client是用户提交应用的客户端；Driver负责协调提交后的分布式应用。架构如图 1‑7所示：

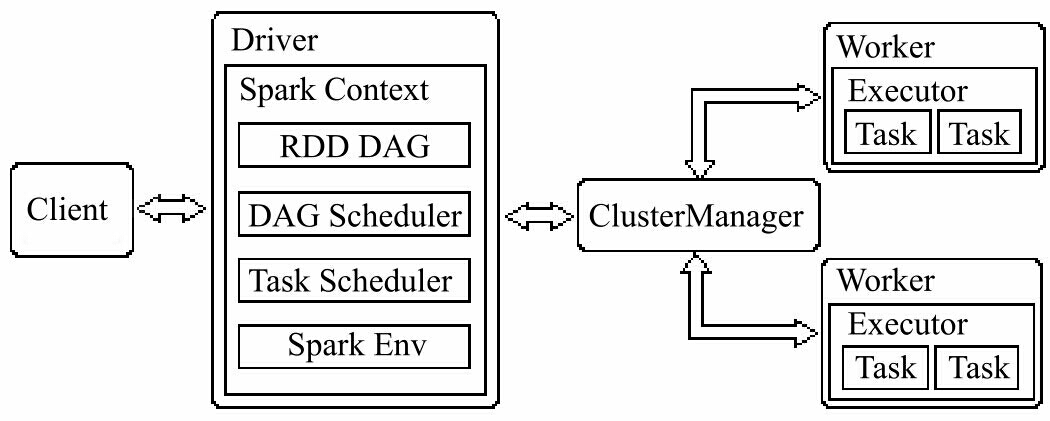


图 1‑7Spark架构

在Spark应用的执行过程中，Driver和Worker是相互对应的。Driver是应用逻辑执行的起点，负责Task任务的分发和调度；Worker负责管理计算节点并创建Executor来并行处理Task任务。Task执行过程中所需的文件和包由Driver序列化后传输给对应的Worker节点，Executor对相应分区的任务进行处理。

Spark的具体流程如下：

1. 用户在Client提交了应用；
2. Master找到Worker，并启动Driver；
3. Driver向资源管理器（YARN模式）或者Master（Standalone模式）申请资源，并将应用转化为RDD Graph；
4. DAG Scheduler将RDD Graph转化为Stage的有向无环图提交Task Scheduler；
5. Task Scheduler提交任务给Executor执行。

# 开发环境搭建

## JAVA安装

1. 下载地址：<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
2. 解压：jdk-8u91-linux-x64.tar.gz
3. 配置环境变量（vi /etc/profile）

export JAVA\_HOME=/home/dyst/JDK1.8

export JRE\_HOME=${JAVA\_HOME}/jre

export CLASSPATH=.:${JAVA\_HOME}/lib:${JRE\_HOME}/lib

1. 测试：java –version

注：以上是linux运行环境的JDK安装方法，在windows开发环境上使用适当的方法安装。

## SCALA安装

1. 下载scala-2.11.8.tgz，下载地址：<http://www.scala-lang.org/download/>
2. 解压scala-2.11.8.tgz，tar –zxvf scala-2.11.8.tgz
3. 配置环境变量

export SCALA\_HOME=/home/dyst/scala

export PATH=$PATH:$SCALA\_HOME/bin

1. 测试

在命令行中输入：scala，即可进入scala命令行。

注：以上是linux运行环境的SCALA安装方法，在windows开发环境上使用适当的方法安装。

## HADOOP安装

下载地址：<https://archive.apache.org/dist/hadoop/common/>

spark基于hadoop，为了访问HDFS（Hadoop Distributed File System，Hadoop分布式文件系统）以及标准或定制的Hadoop输入源，Spark的编译需要与Hadoop的版本对应。本文使用版本为hadoop-2.7.4.tar.gz，解压后路径：E:\hadoop-2.7.4。

配置环境变量：

HADOOP\_HOME=E:\hadoop-2.7.4

PATH=%HADOOP\_HOME%/bin

需要winutils.exe文件，在GitHub上提供了winutils的windows的版本，项目地址是：<https://github.com/srccodes/hadoop-common-2.2.0-bin>（或者<https://github.com/steveloughran/winutils>，选择相应的hadoop版本号，并下载winutils.exe文件），直接下载此项目的zip包，下载后是文件名是hadoop-common-2.2.0-bin-master.zip，将里面的winutils.exe文件拷贝到%HADOOP\_HOME%/bin下。

## SPARK安装

下载地址：<https://spark.apache.org/downloads.html>

本文使用版本为spark-2.2.0-bin-hadoop2.7.tgz.tar，解压后路径：E:\spark-2.2.0-bin-hadoop2.7，注意：Spark的文件目录路径名，不能出现空格，类似于“Program Files”这样的文件夹名是不被允许的。

配置环境变量：

SPARK\_HOME=E:\spark-2.2.0-bin-hadoop2.7

PATH=%SPARK\_HOME%/bin

设置完系统变量后，在任意目录下的cmd命令行中，直接执行spark-shell命令，即可开启Spark的交互式命令行模式。

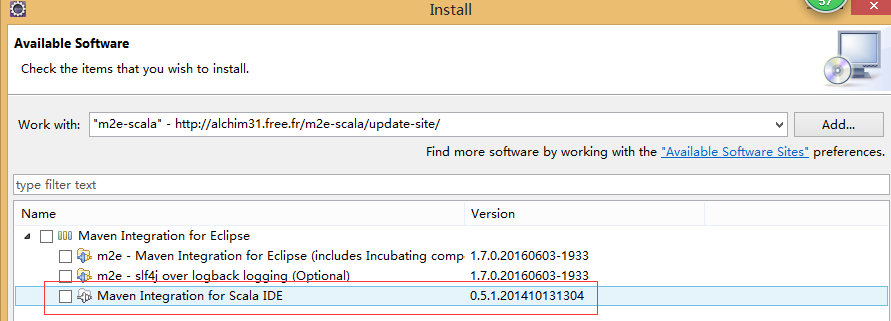
注：Spark的在线文档<http://spark.apache.org/docs/latest/涵盖了进一步学习Spark所需的各种资料>。

## Scala IDE For Eclipse

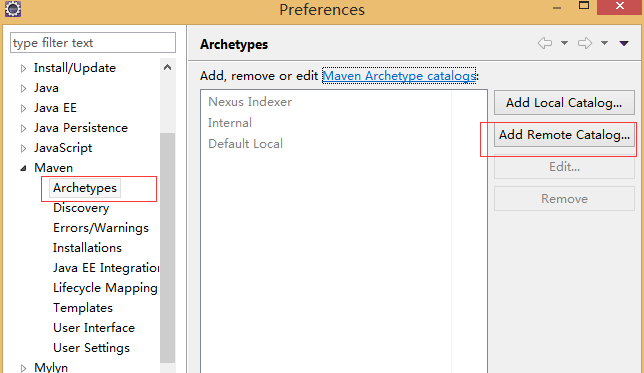
1. 下载地址：<http://scala-ide.org/>
2. 解压放到指定目录下即可

## Maven-Scala插件安装

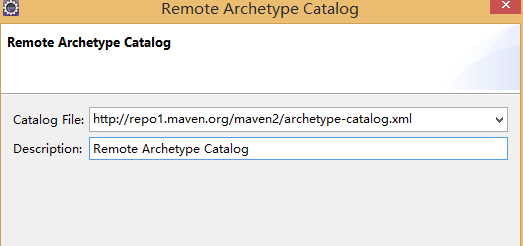
通过http://alchim31.free.fr/m2e-scala/update-site/下载安装（可能网络问题导致经常安装不成功），如果eclipse已经有Maven插件了，可只选择第三项。



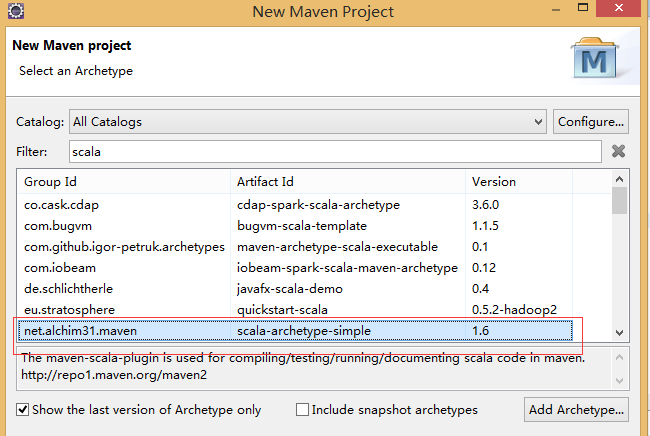
添加远程的原型或模板目录：<http://repo1.maven.org/maven2/archetype-catalog.xml>。



新建Archetype



## 创建maven工程，选择scala插件；add scala nature



## spark工程打包

在Scala工程中，右击“WordCount.scala”，选择“Export”，并在弹出框中选择“Java” –> “JAR File”，进而将该程序编译成jar包。

# 生产环境搭建

## SSH免密码登录

配置好SSH免密码登录之后，机器之间无须用户输入密码即可相互访问。下面介绍安装与配置过程：

1. 安装SSH：yum install ssh
2. 生成公私钥：cd ~/.ssh，然后执行ssh-keygen -t rsa，在目录下将生成id\_rsa.pub与id\_rsa两个密钥文件。其中id\_rsa.pub为公钥。
3. 将公钥id\_rsa.pub拷贝到其他机器上：scp -r id\_rsa.pub <用户名>@IP地址:~/.ssh/authorized\_keys
4. 更改authorized\_keys文件权限：chmod 600 ~/.ssh/authorized\_keys

每台服务器都必须执行一遍。

## 安装Jdk

1. 下载地址：<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
2. 解压：jdk-8u91-linux-x64.tar.gz
3. 配置环境变量（vi /etc/profile）

export JAVA\_HOME=/home/dyst/JDK1.8

export JRE\_HOME=${JAVA\_HOME}/jre

export CLASSPATH=.:${JAVA\_HOME}/lib:${JRE\_HOME}/lib

1. 测试：java –version

## 安装Scala

1. 下载地址：<http://www.scala-lang.org/download/>
2. 解压：tar –zxvf scala-2.11.8.tgz
3. 配置环境变量（vi /etc/profile）

export SCALA\_HOME=/home/dyst/scala

export PATH=$PATH:$SCALA\_HOME/bin

1. 测试

在命令行中输入：scala，即可进入scala命令行。

## 安装hadoop

为了访问HDFS（Hadoop Distributed File System，Hadoop分布式文件系统）以及标准或定制的Hadoop输入源，Spark的编译需要与Hadoop的版本对应。

1. 登录hadoop官网http://hadoop.apache.org/releases.html，下载地址：（其他地址https://archive.apache.org/dist/hadoop/common/，下载稳定本版stable）
2. 解压：tar –zxvf hadoop-2.7.2.tar.gz
3. 配置环境变量（vi /etc/profile）

export HADOOP\_INSTALL=/home/dyst/hadoop

export HADOOP\_HOME=/home/dyst/hadoop

export HADOOP\_CONF\_DIR=$HADOOP\_HOME/etc/hadoop

export YARN\_HOME=$HADOOP\_INSTALL

export YARN\_CONF\_DIR=$YARN\_HOME/etc/hadoop

export HADOOP\_HDFS\_HOME=$HADOOP\_INSTALL

export HDFS\_CONF\_DIR=$HADOOP\_HOME/etc/hadoop

export PATH=$PATH:$HADOOP\_INSTALL/bin:$HADOOP\_INSTALL/sbin

export HADOOP\_MAPRED\_HOME=$HADOOP\_INSTALL

export HADOOP\_COMMON\_HOME=$HADOOP\_INSTALL

执行source /etc/profile使其生效。

1. 配置hadoop

进入Hadoop配置目录：/home/dyst/hadoop/etc/hadoop，配置Hadoop。

1. 配置hadoop-env.sh

export JAVA\_HOME=/home/dyst/JDK1.8

1. 配置yarn-env.sh

默认访问端口：8088

1. 配置slaves文件

添加hadoop集群工作节点，机器名或ip。

如：

dyst74

dyst76

dyst77

dyst78

1. 配置core-site.xml

<property>

<name>fs.defaultFS</name>

<value>hdfs://Master:9000</value>

</property>

<property>

<name>hadoop.tmp.dir</name>

<value>file:/root/bigdata/tmp</value>

</property>

<property>

<name>io.file.buffer.size</name>

<value>131702</value>

</property>

1. 配置hdfs-site.xml

创建namenode和datanode目录，并配置路径。执行：mkdir -p /hdfs/namenode和mkdir -p /hdfs/datanode命令。

配置hdfs-site.xml文件：

<property>

<name>dfs.namenode.name.dir</name>

<value>file:/hdfs/namenode</value>

</property>

<property>

<name>dfs.datanode.data.dir</name>

<value>file:/hdfs/datanode</value>

</property>

<property>

<name>dfs.replication</name>

<value>3</value>

</property>

<property>

<name>dfs.namenode.secondary.http-address</name>

<value>Master:9001</value>

</property>

<property>

<name>dfs.webhdfs.enabled</name>

<value>true</value>

</property>

1. 配置mapred-site.xml

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value>

</property>

<property>

<name>mapreduce.jobhistory.address</name>

<value>Master:10020</value>

</property>

<property>

<name>mapreduce.jobhistory.webapp.address</name>

<value>Master:19888</value>

</property>

1. 配置yarn-site.xml

<property>

<name>yarn.nodemanager.aux-services</name>

<value>mapreduce\_shuffle</value>

</property>

<property>

<name>yarn.nodemanager.auxservices.mapreduce.shuffle.class</name>

<value>org.apache.hadoop.mapred.ShuffleHandler</value>

</property>

<property>

<name>yarn.resourcemanager.address</name>

<value>Master:8032</value>

</property>

<property>

<name>yarn.resourcemanager.scheduler.address</name>

<value>Master:8030</value>

</property>

<property>

<name>yarn.resourcemanager.resource-tracker.address</name>

<value>Master:8031</value>

</property>

<property>

<name>yarn.resourcemanager.admin.address</name>

<value>Master:8033</value>

</property>

<property>

<name>yarn.resourcemanager.webapp.address</name>

<value>Master:8088</value>

</property>

1. 启动与测试（在主节点上操作）
2. 格式化HDFS文件系统

$HADOOP\_HOME/bin/hdfs namenode -format

1. 启动集群：$HADOOP\_HOME/sbin/start-all.sh
2. 访问

hdfs访问：<http://master-ip:50070>

yarn访问：http://master-ip:8088

使用jps查看进程。

## 安装spark

1. 下载地址：<https://spark.apache.org/downloads.html>
2. 解压：tar –zxvf spark-1.6.1-bin-hadoop2.6.tgz
3. 配置环境变量（vi /etc/profile）

export SPARK\_HOME=/home/dyst/spark

export PATH=$PATH:$SPARK\_HOME/bin:$SPARK\_HOME/sbin

export SPARK\_MASTER\_IP=10.42.127.79

export SPARK\_MASTER\_PORT=7077

执行source /etc/profile使其生效。

1. 对每一台机器操作。打开/etc/hosts，加入集群中Master及各个Worker节点的ip与hostname配对。

x.x.x.x dyst79

x.x.x.x dyst74

x.x.x.x dyst75

x.x.x.x dyst76

x.x.x.x dyst77

x.x.x.x dyst78

1. 配置spark-env.sh

/home/dyst/spark/conf/目录下，使用cp命令复制一份spark-env.sh：

cp spark-env.sh.template spark-env.sh

添加如下配置：

###jdk安装目录

export JAVA\_HOME=/home/dyst/jdk1.8

###scala安装目录

export SCALA\_HOME=/home/dyst/scala

###hadoop集群目录

export HADOOP\_HOME=/home/dyst/hadoop

###hadoop集群的配置文件目录

export HADOOP\_CONF\_DIR=/home/dyst/hadoop/etc/hadoop

###spark集群的master节点的ip或主机名

export SPARK\_MASTER\_IP=dyst79

###指定的worker节点能够最大分配给Excutors的内存大小

export SPARK\_WORKER\_MEMORY=3g

|  |  |  |
| --- | --- | --- |
| 序号 | 配置 | 描述 |
| 1 | JAVA\_HOME | jdk安装目录 |
| 2 | SCALA\_HOME | scala安装目录 |
| 3 | HADOOP\_HOME | hadoop集群目录 |
| 4 | HADOOP\_CONF\_DIR | hadoop集群的配置文件目录 |
| 5 | SPARK\_LOCAL\_IP | Spark本机ip地址 |
| 6 | SPARK\_PUBLIC\_DNS | 主节点和工作节点的公共域名 |
| 7 | SPARK\_CLASSPATH | 配置spark运行需要增加并加载的类或类路径 |
| 8 | SPARK\_LOCAL\_DIRS | Spark用于写中间数据，如RDD Cache，Shuffle，Spill等数据的位置 |
| 9 | MESOS\_NATIVE\_JAVA\_LIBRARY |  |
| 10 | SPARK\_EXECUTOR\_INSTANCES |  |
| 11 | SPARK\_MASTER\_WEBUI\_PORT |  |

1. 配置slaves，spark集群节点

在每台Worker节点机器上的/home/dyst/spark/conf/目录下，使用cp命令复制一份slaves：

cp -r slaves.template slaves

然后加入以下内容：

dyst74

dyst75

dyst76

dyst77

dyst78

1. 启动与测试（在主节点上操作）

到/home/dyst/spark/sbin目录下执行./start-all.sh(停止：./stop-all.sh)

spark集群访问：<http://master:8080/>

进入spark的bin目录，启动./spark-shell控制台

sbin/start-master.sh：在机器上启动一个master实例

sbin/start-slaves.sh：在每台机器上启动一个slave实例

sbin/start-all.sh：同时启动一个master实例和所有slave实例

sbin/stop-master.sh：停止master实例

sbin/stop-slaves.sh：停止所有slave实例

sbin/stop-all.sh：停止master实例和所有slave实例

1. 添加依赖包（在各节点上操作）

在/home/dyst/spark/conf目录下，cp –r spark-defaults.conf.template spark-defaults.conf，然后在spark-defaults.conf中添加依赖：

spark.driver.extraClassPath /home/dyst/spark/lib/spark-streaming-kafka\_2.10-1.6.0.jar:/home/dyst/spark/lib/kafka\_2.10-0.8.2.1.jar:/home/dyst/spark/lib/mysql-connector-java-5.1.32.jar:/home/dyst/spark/lib/metrics-core-2.2.0.jar:/home/dyst/spark/lib/kafka-clients-0.8.2.1.jar:/home/dyst/spark/lib/elasticsearch-spark\_2.10-2.2.0.jar

spark.executor.extraClassPath /home/dyst/spark/lib/spark-streaming-kafka\_2.10-1.6.0.jar:/home/dyst/spark/lib/kafka\_2.10-0.8.2.1.jar:/home/dyst/spark/lib/mysql-connector-java-5.1.32.jar:/home/dyst/spark/lib/metrics-core-2.2.0.jar:/home/dyst/spark/lib/kafka-clients-0.8.2.1.jar:/home/dyst/spark/lib/elasticsearch-spark\_2.10-2.2.0.jar

1. 最后将配置及依赖包复制到其他机器，重启spark。

## 安装过程中遇到的问题

### Hadoop datanode 无法启动，报错：java.io.IOException: All specified directories are failed to load

解决办法：

可能是由于在关机的时候，按照正常顺序退出集群造成的，导致dfs中的id不同。

进入dfs/data，修改VERSION文件即可，将version里面的内容修改成和master一直的。

或者直接删除dfs，格式化hdfs即可（./hdfs namenode -format）

### hadoop创建文件夹与授权

hadoop fs -mkdir /checkpoint(创建文件夹)

hadoop fs -mkdir /checkpoint/ssyj

hadoop dfs –chmod 777 /checkpoint（修改文件夹权限）

### spark性能优化

# Scala语言基础

## Hello World

**package** com.kermit.scala.helloworld

**object** HelloScala {

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

println("Hello World!");

}

}

## Scala基础语法

对象：对象有状态和行为。例如：狗有状态 - 颜色，名字，品种，它有行为 - 摇摆，吠叫，吃东西。对象是类的一个实例。

类：类可以被定义为一个模板/蓝本以描述行为/指示其类型支持对象。

方法：方法本质上是一个行为。类可以包含许多方法。它是在将逻辑写入方法中，数据进行操作和所有动作被执行。

字段：每个对象都有其独特的一组临时变量，这是所谓的字段。对象的状态是由分配给这些字段中的值创建的。

1. 基础语法
2. **区分大小写** -  Scala是大小写敏感的，这意味着标识Hello 和 hello在Scala中会有不同的含义。
3. **类名** - 对于所有的类名的第一个字母要大写。  
   如果需要使用几个单词来构成一个类的名称，每个单词的第一个字母要大写。示例：*class MyFirstScalaClass*
4. **方法名称** - 所有的方法名称的第一个字母用小写。  
   如果若干单词被用于构成方法的名称，则每个单词的第一个字母应大写。示例：*def myMethodName()*
5. **程序文件名** - 程序文件的名称应该与对象名称完全匹配。保存文件时，应该保存它使用的对象名称（记住Scala是区分大小写），并追加“.scala”为文件扩展名。 （如果文件名和对象名称不匹配，程序将无法编译）。示例: 假设“HelloWorld”是对象的名称。那么该文件应保存为'HelloWorld.scala“
6. **def main(args: Array[String])** - Scala程序从main()方法开始处理，这是每一个Scala程序的强制程序入口部分。
7. Scala标识符
8. 文字标识符：字母数字标识符开始以字母或下划线，可以使用字母，数字或下划线。“$”字符在Scala中是保留关键字，标识符不能使用。
9. 运算符标识符：运算符识别符由一个或多个运算符字符。操作字符是可打印的ASCII字符，如+, :, ?, ~ 或#。
10. 混合标识符：混合标识符由一个字母数字识别符，随后是一个下划线和运算符标识。
11. 立即数标识符：一个文字标识是包含在反引号(` . . . `)的任意字符串。
12. Scala关键字

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| abstract |  | case |  | catch | |  | class |
| def |  | do |  | else |  | | extends |
| false |  | final |  | finally |  | | for |
| forSome |  | if |  | implicit |  | | import |
| lazy |  | match |  | new |  | | null |
| object |  | override |  | package |  | | private |
| protected |  | return |  | sealed |  | | super |
| this |  | throw |  | trait |  | | try |
| true |  | type |  | val |  | | var |
| while |  | with |  | yield |  | |  |
| - |  | : |  | = |  | | => |
| <- |  | <: |  | <% |  | | >: |
| # |  | @ |  |  |  | |  |

1. Scala访问修饰符
2. 私有成员

私有成员只能看到里面包含的成员定义的类或对象。下面是一个例子：

class Outer {

class Inner {

private def f() { println("f") }

class InnerMost {

f() // OK

}

}

(new Inner).f() // Error: f is not accessible

}

1. 保护成员

受保护的成员是从该成员定义的类的子类才能访问。下面是一个例子：

package p {

class Super {

protected def f() { println("f") }

}

class Sub extends Super {

f()

}

class Other {

(new Super).f() // Error: f is not accessible

}

}

1. 公共成员

未标示私有或受保护的每一个成员是公开的。不需要明确使用修饰符public。这样的成员可以从任何地方访问。

1. 保护范围

Scala中的访问修饰符可以增加使用修饰符。形式：private[X]或protected[X]的修饰符意味着访问私有或受保护“达到”X，其中X代表了一些封闭的包，类或单个对象。

1. Scala中的注释

Scala支持单行和多行注释非常类似于Java。多行注释可以嵌套，但必须正确嵌套。可用的任何注释和所有字符都将被Scala编译器忽略。

## Scala数据类型

|  |  |
| --- | --- |
| **数据类型** | **描述** |
| Byte | 8位有符号值。范围从-128到127 |
| Short | 16位有符号值。范围从-32768至32767 |
| Int | 32 位有符号值。范围从 -2147483648 to 2147483647 |
| Long | 64位有符号值。 从-9223372036854775808到9223372036854775807 |
| Float | 32位IEEE754单精度浮点数 |
| Double | 64位IEEE754双精度浮点数 |
| Char | 16位无符号Unicode字符。范围由U+0000至U+FFFF |
| String | 字符序列 |
| Boolean | 无论是字面true或false字面 |
| Unit | 对应于没有值 |
| Null | 空或空引用 |
| Nothing | 每一个其他类型的子类型; 包括无值 |
| Any | Any类型的超类型;任何对象是任何类型 |
| AnyRef | 任何引用类型的超类型 |

## 运算符

1. 算术运算符

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **描述** |  | **示例** |
| + | 两个操作数相加 |  | A + B = 30 |
| - | 从第一操作减去第二操作数 |  | A - B = -10 |
| \* | 两个操作数相乘 |  | A \* B = 200 |
| / | 通过分子除以分子 |  | B / A = 2 |
| % | 模运算，整数除法后的余数 |  | B % A = 0 |

1. 关系运算符

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **示例** |
| == | 检查两个操作数的值是否相等，如果是的话那么条件为真。 | (A == B)不为 true. |
| != | 检查两个操作数的值是否相等，如果值不相等，则条件变为真。 | (A != B) 为 true. |
| > | 检查左边的操作数的值是否大于右操作数的值，如果是的话那么条件为真。 | (A > B) 不为 true. |
| < | 检查左边的操作数的值是否小于右操作数的值，如果是的话那么条件为真。 | (A < B) 为 true. |
| >= | 检查左边的操作数的值是否大于或等于右操作数的值，如果是的话那么条件为真。 | (A >= B) 不为 true. |
| <= | 检查左边的操作数的值是否小于或等于右操作数的值，如果是的话那么条件为真。 | (A <= B) 为 true. |

1. 逻辑运算符

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **示例** |
| && | 所谓逻辑与操作。如果两个操作数为非零则条件为真。 | (A && B) 为 false. |
| || | 所谓的逻辑或操作。如果任何两个操作数是非零则条件变为真。 | (A || B) 为 true. |
| ! | 所谓逻辑非运算符。使用反转操作数的逻辑状态。如果条件为真，那么逻辑非操作符作出结果为假。 | !(A && B) 为  true. |

1. 位运算符

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **示例** |
| & | 二进制和运算符副本位的结果，如果它存在于两个操作数。 | (A & B) = 12, 也就是 0000 1100 |
| | | 二进制或操作拷贝，如果它存在一个操作数。 | (A | B) = 61, 也就是 0011 1101 |
| ^ | 二进制异或运算符的副本，如果它被设置在一个操作数而不是两个比特。 | (A ^ B) = 49, 也就是 0011 0001 |
| ~ | 二进制的补运算符是一元的，具有“翻转”位的效应。 | (~A ) = -61, 也就是 1100 0011在2补码形式，由于一个带符号二进制数。 |
| << | 二进制左移位运算符。左边的操作数的值向左移动由右操作数指定的位数。 | A << 2 = 240, 也就是 1111 0000 |
|  |  |  |
| >> | 二进制向右移位运算符。左边的操作数的值由右操作数指定的位数向右移动。 | A >> 2 = 15, 也就是 1111 |
| >>> | 右移补零操作。左边的操作数的值由右操作数指定的位数向右移动，并转移值以零填充。 | A >>>2 = 15 也就是 0000 1111 |

1. 赋值运算符

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **示例** |
| = | 简单的赋值操作符，分配值从右边的操作数左侧的操作数 | C = A + B 将分配 A + B 的值到 C |
| += | 加法和赋值运算符，它增加了右操作数左操作数和分配结果左操作数 | C += A 相当于 C = C + A |
| -= | 减和赋值运算符，它减去右操作数从左侧的操作数和分配结果左操作数 | C -= A 相当于 C = C - A |
| \*= | 乘法和赋值运算符，它乘以右边的操作数与左操作数和分配结果左操作数 | C \*= A 相当于 C = C \* A |
| /= | 除法和赋值运算符，它把左操作数与右操作数和分配结果左操作数 | C /= A 相当于 C = C / A |
| %= | 模量和赋值运算符，它需要使用两个操作数的模量和分配结果左操作数 | C %= A 相当于 C = C % A |
| <<= | 左移位并赋值运算符 | C <<= 2 等同于 C = C << 2 |
| >>= | 向右移位并赋值运算符 | C >>= 2 等同于 C = C >> 2 |
| &= | 按位与赋值运算符 | C &= 2 等同于C = C & 2 |
| ^= | 按位异或并赋值运算符 | C ^= 2 等同于 C = C ^ 2 |
| |= | 按位或并赋值运算符 | C |= 2 等同于 C = C | 2 |

## 变量定义

1. val 变量名:变量类型 = 值

一旦初始化，val就不能再被赋值。

1. var变量名:变量类型 = 值

可以在生命周期中多次被赋值。

**var** a : *String* = "a" //完整定义

**var** b = "b" //自动识别变量类型

## 函数定义

### 函数声明

函数的定义以def开始。然后是函数名，跟着是括号里带有冒号分分隔的参数列表。函数的每个参数必须带有前缀冒号的类型标注，因为Scala编译器无法推断函数的参数类型。括号后紧接“:变量类型”，其定义该函数的返回结果类型。跟在函数结果类型之后的是一个等号和一对包含了函数体的花括号。函数定义有以下几种形式：

1. 规规矩矩的写法，带有等号、大括号和返回值类型的形式

def myFunc(var p1 : Int) : Int = {

//something

}

def myFunc(var p1 : Int) : Unit = {

//something

}

1. 非unit返回值的情形下，省略返回值，让程序根据代码块，自行判断。注意，这里等号还是要的

def myFunc(var p1 : Int) = {

//something

}

1. unit返回值的情况下，直接省略返回值类型和等号

def myFunc(var p1 : Int) {

//something

// return unit

}

1. 函数只有一行的情形下省略返回值和大括号

def max2(x: Int, y: Int) = if (x > y) x else y

def greet() = println("Hello, world!")

### 函数进阶

**def** methodA(x : Int) = x + 100

**var** a = (x : Int) => x + 100

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

println(methodA(2))

println(a(2))

}

## 流程控制

### for循环

**var** i = 0

**for**(i <- 1 to 10){//1到10

println(i)

}

i = 0

**for**(i <- 1 until 10){//1到9

println(i)

}

**var** files = (**new** java.io.File(".")).listFiles();

**for**(file <- files) {

println(file.getAbsolutePath)

}

//跳出循环

**var** a = 0;

**val** numList = List(1,2,3,4,5,6,7,8,9,10);

**val** loop = **new** Breaks;

loop.breakable {

**for**(a <- numList){

println( "Value of a: " + a );

**if**( a == 4 ){

loop.break;

}

}

}

//相当于两层循环语句

**for**(i <- 1 to 2;j <- 1 to 3){

println("i:" + i + ",j:" + j);

}

//相当于两层循环语句，过滤条件

**for**(i <- 1 to 2;j <- 1 to 3 **if** i != j){

println("i:" + i + ",j:" + j);

}

### do-while循环

**def** methodA() : Unit = {

**var** str = ""

**do**{

str = ~~readLine~~()

println(str)

} **while**(str != "")

}

### while循环

**var** a:Int = 10

**while**(a < 20){

println(a)

a+=1

}

### if语句

**def** methodA(a : Int) {

**if**(a <= 3){

println("a <= 3: " + a);

} **else** **if**(a <= 5){

println("a <= 5: " + a);

} **else** **if**(a <= 10){

println("a <= 10: " + a);

} **else** {

println("a > 10: " + a)

}

}

## 异常处理

**try** {

**var** i = 6/0

} **catch**{

**case** re : *RuntimeException* => println("The Exception is:" + re.getMessage)

**case** e : *Exception* => println("The Exception is:" + e.getMessage)

} **finally** {

println("==========end============")

}

## 元组Tuple

Scala的元组结合件多个固定数量在一起，使它们可以被传来传去作为一个整体。不像一个数组或列表，元组可以容纳不同类型的对象，但它们也是不可改变的。

val t = (1, "hello", Console)

val t = new Tuple3(1, "hello", Console)

一个元组的实际类型取决于它包含的元素和这些元素的类型的数目。因此，该类型 (99, "Luftballons") 是 Tuple2[Int, String].  ('u', 'r', "the", 1, 4, "me") 的类型是 Tuple6[Char, Char, String, Int, Int, String]

元组类型Tuple1，Tuple2，Tuple3等。至少目前为22的上限在Scala，如果需要更多，那么可以使用一个集合，而不是一个元组。对于每个TupleN类型，其中1<= N <= 22。

//交换元素（2元组有效） 可以使用Tuple.swap方法来交换一个Tuple2的元素

**var** t1 = **new** **Tuple2**(100, "you")

println("Swapped Tuple: " + t1.swap )

//访问元素

**var** t2 = (1, "love", 5l)

**var** t3 = ("I", "Love", "China", "！")

println(t3.\_1)

println(t3.\_2)

println(t3.\_3)

println(t3.\_4)

//迭代遍历

t3.productIterator.foreach{ j => println("Value = " + j) }

## 数组Array

Scala中提供了一种数据结构-数组，其中存储相同类型的元素的固定大小的连续集合。数组用于存储数据的集合，但它往往是更加有用认为数组作为相同类型的变量的集合。

### 数组声明

var z:Array[String] = new Array[String](3)

or

var z = new Array[String](3)

var z = Array("Zara", "Nuha", "Ayan")

### 遍历数组

**var** array = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)

**for**(i <- 0 until array.length){

//println("Array1：" + array(i))

println("for until：" + array.apply(i))

}

**for**(value <- array){

println("for each：" + value)

}

array.foreach { x => println("foreach varry：" + x) }

### 复制数据

**var** array1 = Array(1, 2, 3, 4)

**var** array2 = **new** Array[Int](10)

array1.copyToArray(array2)

**for**(value <- array2){

println("array2：" + value)

}

### 获取元素

1. 获取第一个元素

**var** array3 = Array("Love", "China")

println("array3 first:" + array3.head)//获取第一个元素

1. 获取最后一个元素

**var** array3 = Array("Love", "China")

println("array3 last:" + array3.last)//最后一个元素

### 添加元素

1. 将一个元素添加至列表前面

**var** array3 = Array("Love", "China")

**var** array4 = array3.+:("I")//将一个元素添加到列表前面

println("array4 first:" + array4.head)

1. 将一个元素添加至列表后面

**var** array3 = Array("Love", "China")

**var** array4 = array3.+:("I")//将一个元素添加到列表前面

println("array4 first:" + array4.head)

**var** array5 = array4.:+("！")//将一个元素添加到列表后面

println("array5 last:" + array5.last)

### 连接数组

**var** array5 = Array(" I Love China !")

**var** array6 = Array(" What about you ?")

**var** array7 = array5.++(array6) //array5 + array6

//var array7 = array5.++:(array6) //array6 + array5

**for**(value <- array7){

println("array7：" + value)

}

### 元素去重

**var** array8 = Array(1, 2, 3, 3, 2, 5, 7,1)

**var** array9 = array8.distinct

**for**(value <- array9){

println("array9：" + value)

}

### 判断是否存在符合某一条件的元素

**var** array8 = Array(1, 2, 3, 3, 2, 5, 7,1)

array8.exists { x => x > 10 }//false

### 过滤元素，仅获取符合条件的元素

**var** array9 = Array(1, 2, 3, 3, 2, 5, 7,1)

**var** varry10 = array9.filter { x => x > 4 }

**for**(value <- varry10){

println("varry10：" + value)

}

### scala.collection.mutable.ArrayBuffer

**var** abf = **new** ArrayBuffer[*String*]()

abf += "1" //添加一个元素

abf += ("9", "3") //添加一个数组的元素

abf ++= Array("7", "6") //添加一个数组的元素

abf.insert(2, "5", "4")//在下标2位置插入元素，原来的元素位置往前移动

abf.remove(2)//删除下标为2的元素

abf.remove(2, 2)//从下标为2开始，删除两个元素

abf.trimEnd(3) //删除最后3个元素

abf.foreach { x => print(x + " ") }

abf.sorted

## Map集合

Scala中的映射是键/值对的集合。任何值可以根据它的键进行检索。键是在映射唯一的，但值不一定是唯一的。映射也被称为哈希表。有两种类型的映射，不可变以及可变的。可变和不可变的对象之间的区别在于，当一个对象是不可变的，对象本身不能被改变。

默认情况下，Scala中使用不可变的映射。如果想使用可变集，必须明确地导入scala.collection.mutable.Map类。如果想在同一个同时使用可变和不可变的映射，那么可以继续参考不可变的映射作为映射，但可以参考可变集合为mutable.Map。

### Map声明

// Empty hash table whose keys are strings and values are integers:

var A:Map[Char,Int] = Map()

// A map with keys and values.

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")

### 遍历

**var** mapA = Map("China" -> "I am a chinese.", "America" -> "She is a American")

**for**((key, value) <- mapA){

println("key:" + key + ", value:" + value);

}

mapA.keys.foreach { x =>

print("key:" + x)

println(" value:" + mapA(x)) }

mapA.foreach{case(x, y) => println("key:" + x + " value:" + mapA(x))}

### 添加/去除键值对

**var** mapA = Map("China" -> "I am a chinese.", "America" -> "She is a American")

mapA += ("France" -> "What?") //增加一个键值对

mapA += ("A" -> "1", "B" -> "2") //增加多个键值对

mapA -= "America" //去除一个键值对

mapA.keys.foreach { x =>

print("key:" + x)

println(" value:" + mapA(x)) }

### 串联

//连接

**val** mapB = Map("A" -> "1", "B" -> "2")

**val** mapC = Map("C" -> "3", "D" -> "4")

**var** mapD =mapB.++:(mapC)

mapD.keys.foreach { x =>

print("-key:" + x)

println(" value:" + mapD(x)) }

### 检查是否存在映射键

**val** mapF = Map("C" -> "3", "D" -> "4")

**if**(!mapF.contains("E")){//true

}

### 根据键获取值

**val** mapG = Map("A" -> "1", "B" -> "2")

println(mapG("C"))

println(mapG.getOrElse("C", "--"))//推荐使用

### 更新

**var** mapH = Map("A" -> "1", "B" -> "2")

mapH += ("A" -> "2", "B" -> "4", "C" -> "5")

mapH.keys.foreach { x =>

print("-key:" + x)

println(" value:" + mapH(x)) }

## 类与对象

类是对象的抽象，而对象是类的具体实例。类是抽象的，不占用内存，而对象是具体的，占用存储空间。类是用于创建对象的蓝图，它是一个定义包括在特定类型的对象中的方法和变量的软件模板。

### 类

#### 类的定义与实例化

Scala中的类不声明为public，一个Scala源文件中可以有多个类。

Scala 的类定义可以有参数，称为类参数，类参数在整个类中都可以访问。

**class** Point(**val** xc: Int, **val** yc: Int) {

**var** x: Int = xc

**var** y: Int = yc

**def** move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("x 的坐标点: " + x);

println ("y 的坐标点: " + y);

}

}

可以使用 new 关键字来创建类的对象：

**object** TestClass01 {

**def** main(args: Array[*String*]) {

**val** pt = **new** Point(10, 20);

// 移到一个新的位置

pt.move(10, 10);

}

}

#### 类的继承

**class** Point(**val** xc: Int, **val** yc: Int) {

**var** x: Int = xc

**var** y: Int = yc

**def** move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("x 的坐标点: " + x);

println ("y 的坐标点: " + y);

}

}

**class** Location(**override** **val** xc: Int, **override** **val** yc: Int,

**val** zc :Int) **extends** Point(xc, yc){

**var** z: Int = zc

**def** move(dx: Int, dy: Int, dz: Int) {

x = x + dx

y = y + dy

z = z + dz

println ("x 的坐标点 : " + x);

println ("y 的坐标点 : " + y);

println ("z 的坐标点 : " + z);

}

}

Scala 使用 extends 关键字来继承一个类。实例中 Location 类继承了 Point 类。Point 称为父类(基类)，Location 称为子类。override val xc 为重写了父类的字段。

继承会继承父类的所有属性和方法，Scala允许继承一个父类。

Scala重写一个非抽象方法，必须用override修饰符。

#### 单例对象（伴生对象）

在 Scala 中，是没有 static 这个东西的，但是它也为我们提供了单例模式的实现方法，那就是使用关键字 object。

Scala 中使用单例模式时，除了定义的类之外，还要定义一个同名的object对象，它和类的区别是，object对象不能带参数。

当单例对象与某个类共享同一个名称时，他被称作是这个类的伴生对象：companion object。你必须在同一个源文件里定义类和它的伴生对象。类被称为是这个单例对象的伴生类：companion class。类和它的伴生对象可以互相访问其私有成员。

// 私有构造方法

**class** Marker **private**(**val** color:*String*) {

println("创建" + **this**)

**override** **def** toString(): *String* = "颜色标记："+ color

}

// 伴生对象，与类共享名字，可以访问类的私有属性和方法

**object** Marker {

**private** **val** markers: *Map*[*String*, Marker] = Map(

"red" -> **new** Marker("red"),

"blue" -> **new** Marker("blue"),

"green" -> **new** Marker("green")

)

**def** apply(color:*String*) = {

**if**(markers.contains(color)) markers(color) **else** **null**

}

**def** getMarker(color:*String*) = {

**if**(markers.contains(color)) markers(color) **else** **null**

}

}

**object** TestClass02 {

**def** main(args: Array[*String*]) {

println(Marker("red"))

// 单例函数调用，省略了.(点)符号

println(Marker getMarker "blue")

}

}

运行结果：

创建颜色标记：red

创建颜色标记：blue

创建颜色标记：green

颜色标记：red

颜色标记：blue

## 泛型

范型就是定义以类型为参数的类或接口（Scala中为特征）的功能。在JAVA也存在泛型的类，如：java.util.List<T>

java.util.List<String> strs = new java.util.ArrayList<String>();

java.util.List<int> ints = new java.util.ArrayList<int>();

**class** A[K](**var** a:K){

**override** **def** toString() : *String* = "A输入的是：" + a

}

//B类的泛型K，确定了A的类型，构造方法参数类型是A[K]

**class** B[K](**var** a:A[K]){

**override** **def** toString() : *String* = "B输入的是：" + a

}

**class** C[K](**var** a:Int, **var** b:A[K]){

**override** **def** toString() : *String* = "C输入的是：" + a + "==" + b

}

**object** TestClass03 {

**def** main(args: Array[*String*]) {

println(**new** A[Int](4))

println(**new** A[*String*]("字符串类型"))

println(**new** B[Int](**new** A[Int](4)))

println(**new** B[*String*](**new** A[*String*]("字符串类型")))

println(**new** C[Int](3, **new** A[Int](4)))

println(**new** C[*String*](3, **new** A[*String*]("字符串类型")))

}

}

# Spark开发

每个Spark应用程序都由一个主函数及各种并行操作运行在集群上。spark的两个抽象概念：

1. RDD（Resilient Distributed Dataset，弹性分布式数据集）
2. 共享变量：broadcast variables（广播变量）和*accumulators（*累加*）*

每个spark程序或shell会话如下流程：

1. 从外部数据中创建RDDs
2. 对RDDs进行转换成新的RDDs，例如filter（）操作
3. 若要对RDDs进行重复使用，则需persist（）持久化
4. actions例如count（），first（），启动并行的计算，spark会对计算任务进行优化和执行。

## Spark集群

Spark集群由两类程序构成：一个驱动程序和多个执行程序。本地模式时所有的处理都运行在同一个JVM内，而在集群模式时它们通常运行在不同的节点上。

### Spark集群管理

Spark应用程序作为独立的进程运行在集群上，由主程序（称为驱动程序Driver Program）中的SparkContext对象协调。

## Spark运行模式

Spark运行模式主要包括Local模式、Standalone模式、YARN模式与Mesos模式。

### Local模式

Local模式即本地运行模式，在单个节点上运行，使用Local[N]启用一个或多个线程（最大值与核数有关）来运行spark程序，主要用于开发调试Spark应用程序。如：

**val** conf = **new** SparkConf().setAppName("Test").setMaster("local[4]")

**val** sc = **new** SparkContext(conf)

### Standalone模式

即独立模式，自带完整的服务，可单独部署到一个集群中，无需依赖任何其他资源管理系统。

需要将Spark复制到集群机器中的每个节点，然后分别启动每个节点即可；Spark Standalone模式的集群由Master与Worker节点组成，程序通过与Master节点交互申请资源，Worker节点启动Executor运行。

### YARN模式

Spark on YARN的原理就是依靠yarn来调度Spark，比默认的Spark运行模式性能要好的多，前提是首先部署好hadoop HDFS并且运行在yarn上，然后就可以开始部署spark on yarn了。

YARN模式分为两种：

* yarn-client

在yarn-client模式下，Spark driver在客户机上运行，然后向YARN申请运行exeutor以运行Task，即Driver和YARN是分开的，Driver程序作为YARN集群的一个客户端，这是一种CS模式。

提交命令：./spark-submit --name SparkWordCount --class spark.examples.SparkWordCount --master yarn-client --executor-memory 512M --total-executor-cores 1 SparkWordCount.jar README.md

采用yarn-client方式，因为driver在客户端，所以可以通过webUI访问driver的状态，默认是http://hadoop.master:4040访问，而YARN通过http://haoop.master:8088访问。

* yarn-cluster

在yarn-cluster模式下，Spark driver将作为一个ApplicationMaster在YARN集群中先启动，然后再由ApplicationMaster向RM申请资源启动executor以运行Task。也就是说，在这种部署方式下，Driver程序运行在YARN集群上。

提交命令：./spark-submit --name SparkWordCount --class spark.examples.SparkWordCount --master yarn-cluster --executor-memory 512M --total-executor-cores 1 SparkWordCount.jar README.md

访问http://hadoop.master:8088查看任务执行结果。

## SparkContext类与SparkConf类

任何Spark程序的编写都是从SparkContext（或用Java编写时的JavaSparkContext）开始的。SparkContext的初始化需要一个SparkConf对象，后者包含了Spark集群配置的各种参数。

初始化后，我们便可用SparkContext对象所包含的各种方法来创建和操作分布式数据集和共享变量。

**val** conf = **new** SparkConf().setAppName("Test Spark App").setMaster("local[4]")

**val** sc = **new** SparkContext(conf)

这段代码会创建一个4线程的SparkContext对象，并将其相应的任务命名为Test Spark APP。setMaster()用来设置运行模式。简化为：

**val** sc = **new** SparkContext("local[4]", "Test Spark App")

### 加载文件textFile(……)

def textFile(path: String, minPartitions: Int): RDD[String]

* path: String 是一个URI，这个URI可以是HDFS、本地文件（全部的节点都可以），或者其他Hadoop支持的文件系统URI；加载HDFS文件和本地文件都是使用textFile，区别是添加前缀（hdfs://和file://）进行标识，从本地文件读取文件直接返回MapPartitionsRDD,而从HDFS读取的文件先转成HadoopRDD,然后隐式转换成MapPartitionsRDD。

1. 从当前程序目录读取一个或多个文件

**val** textFile = sc.textFile("a1.txt", 2);

println(textFile.first())//获取第一项

//加载多个文件，如果开启多个线程，则每个线程同时读取一个文件

**val** textFile = sc.textFile("a1.txt,a2.txt", 2);

textFile.foreach { x => println(x) }

1. 读取本地一个文件

//windows

**val** textFile = sc.textFile("file:///E:/a1.txt", 2);

textFile.foreach { x => println(x) }

//linux

**val** textFile = sc.textFile("file:///home/dyst/a1.txt", 2);

textFile.foreach { x => println(x) }

1. 读取一个文件夹下的所有文件

**val** textFile = sc.textFile("file:///E:/txt/", 2);

textFile.foreach { x => println(x) }

1. 采用通配符读取文件

**val** textFile = sc.textFile("file:///E:/txt/\*/\*1.txt", 2);

textFile.foreach { x => println(x) }

1. 从hdfs读取文件

val path = "hdfs://master:9000/examples/examples/src/main/resources/people.txt"

val rdd1 = sc.textFile(path,2)

* minPartitions=math.min(defaultParallelism, 2)是指定数据的分区，如果不指定分区，当你的核数大于2的时候，不指定分区数那么就是2，当你的数据大于128M时候，Spark是为每一个块（block）创建一个分片（hadoop-2.X之后为128m一个block）
* 返回的是一个字符串类型的RDD，也就是RDD的内部形式是Iterator[(String)]

## 理解闭包

使用Spark的一个难点为：理解程序在集群中执行时变量和方法的生命周期。RDD操作可以在变量范围之外修改变量，这是一个经常导致迷惑的地方。例如下面例子的做法是错误的：

object RddTest3 {

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

val conf = new SparkConf().setAppName("rddtest3").setMaster("local[4]")

val sc = new SparkContext(conf)

var counter = 0

var a = sc.parallelize(List(3, 2, 8, 7, 10, 6), 2)

a.foreach { x => counter += x }//这种操作没有任何意义，必须要使用累加变量

print(counter)//结果：0

//停止

sc.stop();

}

}

在集群模式下，情况会变得复杂，上面的代码并不会按照预期的方式执行。为了执行这个job，Spark把处理RDD的操作分割成多个任务，每个任务将被一个executor处理。在执行之前，Spark首先计算闭包(closure)。闭包是必须对executor可见的变量和方法，在对RDD进行运算时将会用到这些变量和方法(在本例子中指foreach())。这个闭包会被序列化，并发送给每个executor。在local模式下，只有一个executor，所以所有的变量和方法都使用同一个闭包。在其他模式下情况跟local模式不一样，每个executor在不同的worker节点上运行，每个executor都有一个单独的闭包。

在这里，发送给每个executor的闭包内的变量是当前变量的副本，因此当counter在foreach中被引用时，已经不是在driver节点上的counter了。在driver节点的内存中仍然有一个counter，但这个counter对executors不可见。executor只能操作序列化的闭包中的counter副本。因此，最终counter的值仍然是0，因为所有对counter的操作都是在序列化的闭包内的counter上进行的。

在类似这种场景下，为了保证良好的行为确保，应该使用累加器。Spark中的累加器专门为在集群中多个节点间更新变量提供了一种安全机制。

### broadcast variables（广播变量）

broadcast 就是将数据从一个节点发送到其他各个节点上去。这样的场景很多，比如 driver 上有一张表，其他节点上运行的 task 需要 lookup 这张表，那么 driver可以先把这张表 copy 到这些节点，这样 task 就可以在本地查表了。

Broadcast（广播）是相对较为常用方法功能，通常使用方式，包括共享配置文件，map数据集，树形数据结构，kafka生产者（避免频繁创建连接）等，为能够更好更快速为TASK任务使用相关变量。

广播变量不能改变变量的值，其为**只读**变量。

创建并使用广播变量的过程：

* 在一个类型T的对象obj上使用SparkContext.brodcast(obj)方法，创建一个Broadcast[T]类型的广播变量，obj必须满足Serializable；
* 通过广播变量的.value()方法访问其值。

例如下面的代码首先生成了一个集合变量，把这个变量通过sparkContext的broadcast函数进行广播，最后在rdd的每一个partition的迭代时，使用这个广播变量。

var sparkConf = new SparkConf().setMaster("local[2]").setAppName("producer04");

var ssc = new StreamingContext(sparkConf, Milliseconds(5000));

val values = List[Int](1,2,3)

val broadcastValues = ssc.**sparkContext.broadcast**(values)

rdd.mapPartitions(iter => {

broadcastValues.value.foreach(println)

})

val conf = new SparkConf().setAppName("Test Spark App").setMaster("local[4]")

val sc = new SparkContext(conf)

val values = List[Int](1,2,3)

val broadcastValues = sc.**broadcast**(values)

### accumulators（累加器）

当有多个值需要被跟踪记录，或者一个值需要在并行程序的多处进行更新时，使用累加器的计数功能变得尤其方便。例如，原始数据中经常有一部分的无效数据。当无效数据的比例很高时，为了防止产生垃圾输出，需要使用累加器对数据中的有效数据和无效数据分别进行计数。

一个累加器可以通过在原始值v上调用SparkContext.accumulator(v)。然后，集群上正在运行的任务就可以使用add方法或+=操作对该累加器进行累加操作。只有driver程序可以读取累加器的值，读取累加器的值使用value方法。以下是求和示例：

val conf = new SparkConf().setAppName("Test Spark App").setMaster("local[4]")

val sc = new SparkContext(conf)

val accumulator = sc.**accumulator**(0.0)//初始化累加器

val a1 = sc.makeRDD(1 to 4, 3)

val b1 = a1.flatMap { x =>

{

accumulator += x //统计x的和

1 to x

}

} //a1.flatMap(1 to \_)

println(b1.collect().mkString)

println(accumulator)//计算结果：10

在上述例子中，我们无法在flatMap方法中使用一个外部变量，只能初始化一个accumulators变量。

Spark支持Integer、Double（doubleAccumulator），Long（longAccumulator）和Float类型的累加器。除此之外，Spark还提供了自定义累加器类型和聚合操作（如查找最大值等加操作以外的操作）的API，但要保证定义的操作满足交换律和结合律。

## RDD弹性分布式数据集

RDD（Resilient Distributed Dataset，弹性分布式数据集）是Spark的核心概念之一。一个RDD代表一系列的"记录"（严格来说，某种类型的对象）。这些记录被分配或分区到一个集群的多个节点上。Spark中的RDD具备容错性，即当某个节点或任务失败时（因非用户代码错误的原因而引起，如硬件故障、网络不通等），RDD会在余下的节点上自动重建，以便任务能最终完成。

RDD（Resilient Distributed Datasets，弹性分布式数据集）是一个容错的、并行的数据结构，可以让用户显示地将数据存储到磁盘或内存中，并控制数据的分区。RDD还提供了一组丰富的操作来操作这些数据，诸如map、flatMap、filter等转换操作实现了monad模式，很好地契合了Scala的集合操作。除此之外，RDD还提供诸如join、groupBy、reduceByKey等更为方便的操作，以支持常见的数据运算。

RDD是Spark的核心数据结构，通过RDD的依赖关系形成Spark的调度顺序。所谓Spark应用程序，本质是一组对RDD的操作。

RDD从直观上可以看作一个数组，本质上是逻辑分区记录的集合。在集群中，一个RDD可以包含多个分布在不同节点上的分区，每个分区是一个dataset片段，如图 1‑1所示。

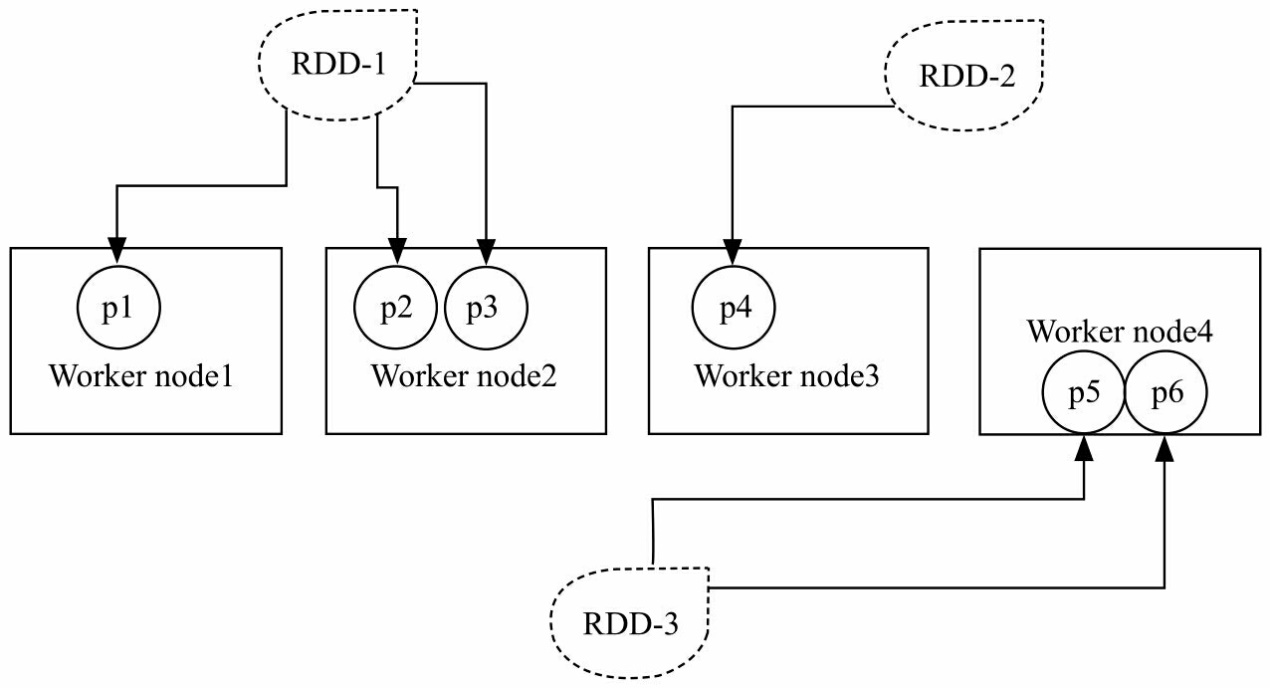


图 5‑1RDD分区

### RDD的两种创建方式

RDD可以从普通数组创建出来，也可以从文件系统或者HDFS中的文件创建出来。

1. 将普通数组转换成RDD，并3个分区存放

**var** a = sc.makeRDD(1 to 9, 3)

**或**

**val** collection = List("a", "b", "c", "d", "e")

**val** rddFromCollection = sc.parallelize(collection, 3)

1. 读取文件来创建RDD，文件中的每一行就是RDD中的一个元素

**val** textFile= sc.textFile("D:\\a.txt")

### RDD的两种操作算子

1. Transformation（变换）：Transformation类型的算子不是立刻执行，而是延迟执行。也就是说从一个RDD变换为另一个RDD的操作需要等到Action操作触发时，才会真正执行。

如：map(func)、filter(func)、flatMap(func)、mapPartitions(func)等操作。

1. Action（行动）：Action类型的算子会触发Spark提交作业，并将数据输出到Spark系统。

如：count()、first()、countByKey()等操作。

RDD将操作分为两类：Transformation与Action。无论执行了多少次Transformation操作，RDD都不会真正执行运算，只有当Action操作被执行时，运算才会被触发。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。

### RDD依赖

RDD存在两种相互依赖关系：窄依赖与宽依赖。

窄依赖是指父Rdd的分区最多只能被一个子Rdd的分区所引用，即一个父Rdd的分区对应一个子Rdd的分区，或者多个父Rdd的分区对应一个子Rdd的分区。（一个父只能有一个子，但一个子可以有多个父）

宽依赖是指子RDD的分区依赖于父RDD的多个分区或所有分区，即存在一个父RDD的一个分区对应一个子RDD的多个分区。（一个父有多个子）

不同的RDD操作会产生不同的依赖，如map操作会产生窄依赖，join操作会产生宽依赖。

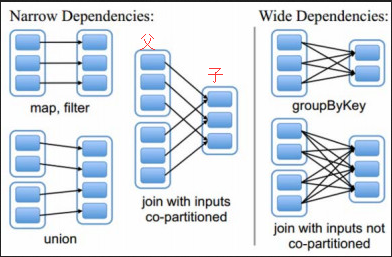


图 5‑2窄依赖与宽依赖

### Transformation算子

#### filter

def filter(f: T => Boolean): RDD[T]：参数是函数，函数会过滤掉不符合条件的元素，返回值是新的RDD。

**val** textFilter = textFile.filter(line => line.contains("run"))//获取包含run字串的行

println(textFilter.count())//行数

#### map

map(func)：数据集中的每个元素经过用户自定义的函数转换形成一个新的RDD，新的RDD叫MappedRDD。

任何原RDD中的元素在新RDD中都有且只有一个元素与之对应。

将textFile每一行文本中的句子使用split(" ")进行分词，并统计分词后的单词数。创建一个基于单词数的新RDD

**val** count = textFile.map(line =>line.split(" ").size)

**val** a = sc.parallelize(1 to 9, 3)//1到9的RDD

**val** b = a.map(x => x\*2)//每个元素乘以2，形成一个新的RDD

#### mapPartitions

mapPartitions是map的一个变种。map的输入函数是应用于RDD中每个元素，而mapPartitions的输入函数是应用于每个分区，也就是把每个分区中的内容作为整体来处理的。

**var** a = sc.makeRDD(1 to 9, 3)

**var** b = a.mapPartitions{iterator => {

**var** res = List[Int]()

**var** i = 1

**while**(iterator.hasNext) {

**var** x = iterator.next

println("===" + i + "===" + x)

i += 1

res.::(x \* 2)

}

res.iterator

}}

println("===========================" + b.collect().mkString)

#### mapPartitionsWithIndex

#### mapValues

原RDD中的Key保持不变，与新的Value一起组成新的RDD中的元素。因此，该函数只适用于元素为KV对的RDD。

**val** c = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", " eagle"), 2)//1到9的RDD

**val** d = c.map {x => (x.length(), x) }

**val** e = d.mapValues { "x" + \_ + "x"}

println(e.collect().mkString)

结果：

(3,xdogx)(5,xtigerx)(4,xlionx)(3,xcatx)(7,xpantherx)(6,x eaglex)

#### flatMap

与map类似，区别是原RDD中的元素经map处理后只能生成一个元素，而原RDD中的元素经flatmap处理后可生成多个元素。

**val** a1 = sc.makeRDD(1 to 4, 3)

**val** b1 = a1.flatMap { x => 1 to x } //a1.flatMap(1 to \_)

println(b1.collect().mkString)

结果：

1121231234 🡪 1 1 2 1 2 3 1 2 3 4

#### union

#### intersection

#### distinct

#### aggregateByKey

#### flatMapValues

flatMapValues类似于mapValues，不同的在于flatMapValues应用于元素为KV对的RDD中Value。每个一元素的Value被输入函数映射为一系列的值，然后这些值再与原RDD中的Key组成一系列新的KV对。

**val** a2 = sc.makeRDD(List((1,2),(3,4),(3,6)))

**val** b2 = a2.flatMapValues { x => x to 5 }

println(b2.collect().mkString)

结果：

(1,2)(1,3)(1,4)(1,5)(3,4)(3,5)

上述例子中原RDD中每个元素的值被转换为一个序列（从其当前值到5），比如第一个KV对(1,2), 其值2被转换为2，3，4，5。然后其再与原KV对中Key组成一系列新的KV对(1,2),(1,3),(1,4),(1,5)。

#### reduceByKey

reduceByKey是reduce的变种，将RDD中相同key的元素两两传递给输入函数，同时产生一个新的值，新产生的值与RDD中下一个元素再被传递给输入函数，最后每一个key都是唯一，且仅对应一个value值。

**val** a4 = sc.makeRDD(List(("a",2),("a",4),("b",6),("c",6),("c",2),("d",1)))

**var** b4 = a4.reduceByKey((x, y) => x + y) //a4.reduceByKey(\_ + \_)

println(b4.collect().mkString)

结果：(d,1)(a,6)(b,6)(c,8)

#### sortBy

该函数最多可以传三个参数：  
　　**第一个参数**是一个函数，该函数的也有一个带T泛型的参数，返回类型和RDD中元素的类型是一致的；（必须）

**第二个参数**是ascending，从字面的意思大家应该可以猜到，是的，这参数决定排序后RDD中的元素是升序还是降序，默认是true，也就是升序；

**第三个参数**是numPartitions，该参数决定排序后的RDD的分区个数，默认排序后的分区个数和排序之前的个数相等，即为this.partitions.size

**val** conf = **new** SparkConf().setAppName("rddtest2").setMaster("local[4]")

**val** sc = **new** SparkContext(conf)

**var** a = sc.parallelize(List(3, 2, 8, 7, 10, 6), 2)

println("a==" + a.collect().mkString)

**var** b = a.sortBy(x => x, **true**, 2).collect().mkString

println("b==" + b)

//停止

sc.stop();

#### sortByKey

该函数最多可以传三个参数：

**第一个参数**是ascending，从字面的意思大家应该可以猜到，是的，这参数决定排序后RDD中的元素是升序还是降序，默认是true，也就是升序；

**第二个参数**是numPartitions，该参数决定排序后的RDD的分区个数，默认排序后的分区个数和排序之前的个数相等，即为this.partitions.size

**val** a2 = sc.makeRDD(List((1,2),(7,4),(3,6),(5,6),(10,6)))

println("a2==" + a2.collect().mkString)

**val** b2 = a2.sortByKey(**false**, 2).collect().mkString

println("b2==" + b2)

#### groupByKey

该函数根据key分组。以下为分组统计：

**val** a5 = sc.makeRDD(List(("a",2),("a",4),("b",6),("c",6),("c",2),("d",1)))

**var** b5 = a5.groupByKey().map(t => (t.\_1, t.\_2.sum));

println(b5.collect().mkString)

#### join

#### cogroup

#### cartesian

#### pipe

#### coalesce()和repartition()

两个方法都是对RDD进行重新分区，但coalesce()操作属于窄依赖，而repartition()是shuffle为true的coalesce()。

分两种情况：

* 当分区较多而每个分区的数据较少时，此时需要减少分区，提高执行效率，这时需要使用coalesce()；（如果设置的分区比之前的多，默认shuffle为false，此时不会起作用）
* 当每个分区的数据量较大时，需要增加分区，以提高并行度，这时需要使用repartition()。

#### repartitionAndSortWithinPartitions

### Action算子

#### reduce

reduce将RDD中元素两两传递给输入函数，同时产生一个新的值，新产生的值与RDD中下一个元素再被传递给输入函数直到最后只有一个值为止。

**val** a3 = sc.parallelize(1 to 10)

**val** b3 = a3.reduce((x, y) => x + y)

println(b3)

结果：55

#### collect

将RDD转换成数组。

**var a3 = sc.parallelize(List(3, 2, 8, 7, 10, 6), 2)**

**var b3 = a3.collect()**

#### count

获取RDD中元素的个数。

**var a3 = sc.parallelize(List(3, 2, 8, 7, 10, 6), 2)**

**print(a3.count())**

**结果：6**

#### first

获取RDD中第一个元素。

**var a3 = sc.parallelize(List(3, 2, 8, 7, 10, 6), 2)**

**print(a3.first())**

**结果：3**

#### take(n)

返回RDD中的n个元素组成的数组，从第一个元素开始取。

**var a3 = sc.parallelize(List(3, 2, 8, 7, 10, 6), 2)**

**var b3 = a3.take(3)//包含3个元素，分别是3，2，8**

#### countByKey()

#### foreach(func)

### RDD缓存

Spark也支持将数据集存进一个集群的内存缓存中，当数据被反复访问时，如在查询一个小而“热”数据集，或运行像PageRank的迭代算法时，是非常有用的。

RDD可以使用persist()方法或cache()方法进行持久化。数据将会在第一次action操作时进行计算，并在各个节点的内存中缓存。Spark的缓存具有容错机制，如果一个缓存的RDD的某个分区丢失了，Spark将按照原来的计算过程，自动重新计算并进行缓存。

另外，每个持久化的RDD可以使用不同的存储级别进行缓存，例如，持久化到磁盘、已序列化的Java对象形式持久化到内存(可以节省空间)、跨节点间复制、以off-heap的方式存储在 Tachyon。

在Spark中有时候很多地方都会用到同一个RDD，按照常规的做法的话，那么每个地方遇到Action操作的时候都会对同一个算子计算多次，这样会造成效率低下的问题。下面sc.textFile被执行了两次，如果对sc.textFile后的RDD进行缓存，则只执行一次即可。（textFile及缓存操作都是Tranformation操作）

val textFile = sc.textFile("file:///E:/txt/file/a1.txt", 2)//加载一个文件

val wordCount = textFile.map(line => line.split(" ").size)

.reduce((a, b) => if(a > b) a else b)

val words = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a+b)

words.foreach(x => println(x.\_1 + "==" + x.\_2))

在不同的操作中把数据集缓存（cache）或存留（persist）在内存（memory）中。当持久化一个RDD后，每个节点都会把计算的分片的结果保存在内存中，之后可以对此数据集在其他action中再次使用。这使得后续的action变得迅速。

cache()和persist()都是用于将一个RDD进行缓存的，这样在之后使用的过程中就不需要重新计算了，可以大大节省程序运行时间。

cache和persist的区别：cache只有一个默认的缓存级别MEMORY\_ONLY ，而persist可以根据情况设置其它的缓存级别。

|  |  |
| --- | --- |
| **持久化级别** | **含义解释** |
| MEMORY\_ONLY | 使用未序列化的Java对象格式，将数据保存在内存中。如果内存不够存放所有的数据，则数据可能就不会进行持久化。那么下次对这个RDD执行算子操作时，那些没有被持久化的数据，需要从源头处重新计算一遍。这是默认的持久化策略，使用cache()方法时，实际就是使用的这种持久化策略。 |
| MEMORY\_AND\_DISK | 使用未序列化的Java对象格式，优先尝试将数据保存在内存中。如果内存不够存放所有的数据，会将数据写入磁盘文件中，下次对这个RDD执行算子时，持久化在磁盘文件中的数据会被读取出来使用。 |
| MEMORY\_ONLY\_SER | 基本含义同MEMORY\_ONLY。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。  如果内存不能全部存储RDD,那么使用MEMORY\_ONLY\_SER，并挑选一个快速序列化库将对象序列化，以节省内存空间。使用这种存储级别，计算速度仍然很快。 |
| MEMORY\_AND\_DISK\_SER | 基本含义同MEMORY\_AND\_DISK。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。 |
| DISK\_ONLY | 使用未序列化的Java对象格式，将数据全部写入磁盘文件中。 |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, 等等 | 对于上述任意一种持久化策略，如果加上后缀\_2，代表的是将每个持久化的数据，都复制一份副本，并将副本保存到其他节点上。这种基于副本的持久化机制主要用于进行容错。假如某个节点挂掉，节点的内存或磁盘中的持久化数据丢失了，那么后续对RDD计算时还可以使用该数据在其他节点上的副本。如果没有副本的话，就只能将这些数据从源头处重新计算一遍了。 |

移除数据：Spark自动监控各个节点上的缓存使用率，并以最近最少使用的方式(LRU)将旧数据块移除内存。如果想手动移除一个RDD，而不是等待该RDD被Spark自动移除，可以使用RDD.unpersist()方法。

### 打印RDD的元素(Printing elements of an RDD)

打印一个RDD的元素也是一个常用的语法，带引RDD元素可以使用方法rdd.foreach(println)或rdd.map(println)。在本地模式下，该方法将生成预期的输出并打印RDD所有的元素。然而，在集群模式下各个executor调用stdout，将结果打印到executor的stdout中。因为不是打印到driver节点上，所以在driver节点的stdout上不会看到这些输出。如果想将RDD的元素打印到driver节点上，可以使用collect()方法将RDD发送到driver节点上，然后再打印该RDD:rdd.collect().foreach(println)。这个操作可能会导致driver节点内存不足，因为collect()方法将RDD全部的数据都发送到一台节点上。如果仅仅打印RDD的部分元素，一个安全的方法是使用take()方法：rdd.take(100).foreach(println)。

### Shuffle操作(Shuffle operations)

shuffle是Spark将多个分区的数据重新分组重新分布数据的机制。shuffle是一个复杂且代价较高的操作，它需要完成将数据在executor和机器节点之间进行复制的工作。

通过reduceByKey操作的例子，来理解shuffle过程。reduceByKey操作生成了一个新的RDD，原始数据中相同key的所有记录的聚合值合并为一个元组，这个元组中的key对应的值为执行reduce函数之后的结果。这个操作的挑战是，key相同的所有记录不在同一各分区种，甚至不在同一台机器上，但是该操作必须将这些记录联合运算。

在Spark中，通常一条数据不会垮分区分布，除非为了一个特殊的操作在必要的地方才会跨分区分布。在计算过程中，一个分区由一个task进行处理。因此，为了组织所有的数据让一个reduceByKey任务执行，Spark需要进行一个all-to-all操作。all-to-all操作需要读取所有分区上的数据的所有的key，以及key对应的所有的值，然后将多个分区上的数据进行汇总，并将每个key对应的多个分区的数据进行计算得出最终的结果，这个过程称为shuffle。

会引起shuffle过程的操作有：

* 1. repartition操作，例如：repartition、coalesce
  2. ByKey操作(除了counting相关操作)，例如：groupByKey、reduceByKey
  3. join操作，例如：cogroup、join

shuffle是一个代价比较高的操作，它涉及磁盘IO、数据序列化、网络IO。为了准备shuffle操作的数据，Spark启动了一系列的map任务和reduce任务，map任务完成数据的处理工作，reduce完成map任务处理后的数据的收集工作。这里的map、reduce来自MapReduce，跟Spark的map操作和reduce操作没有关系。

某些shuffle操作会大量消耗堆内存空间，因为shuffle操作在数据转换前后，需要在使用内存中的数据结构对数据进行组织。需要特别说明的是，reduceByKey和aggregateByKey在map时会创建这些数据结构，ByKey操作在reduce时创建这些数据结构。当内存满的时候，Spark会把溢出的数据存到磁盘上，这将导致额外的磁盘IO开销和垃圾回收开销的增加。

如果Spark应用长期保持对RDD的引用，或者垃圾回收不频繁，这将导致垃圾回收的周期比较长。这意味着，长期运行Spark任务可能会消耗大量的磁盘空间。临时数据存储路径可以通过SparkContext中设置参数spark.local.dir进行配置。

## 把函数传递到Spark

Spark的API，在很大程度上依赖于把驱动程序中的函数传递到集群上运行。这有两种推荐的实现方式。

1. 使用匿名函数的语法：这可以让代码更加简洁。
2. 使用全局单例对象的静态方法。比如，可以定义函数对象object MyFunctions，然后将该对象的MyFunctions.func1方法传递给Spark。如下所示：

object MyFunctions {

def func1(s: String) : (Int, String) = {

return (s.length(), s);

}

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

val conf = new SparkConf().setAppName("Test Spark App").setMaster("local[4]")

val sc = new SparkContext(conf)

val c = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", " eagle"), 2)

val d = c.map { MyFunctions.func1 } //相当于匿名函数 x => (x.length(), x)

println(d.collect().mkString)

}

}

如果传递的是一个类实例方法的引用（而不是一个单例对象），在传递方法的时候，应该同时传递包含该方法的类对象。

/\*\*

\* spark算子中用到scala类，该类必须序列化

\*/

class MyClass extends Serializable{

def func1(s: String) : (Int, String) = {

return (s.length(), s);

}

def doStuff(rdd: RDD[String]): RDD[(Int, String)] = { rdd.map(func1) }//类似写成：rdd.map(this.func1)

}

类似地，访问外部对象的字段时将引用整个对象。

/\*\*

\* spark算子中用到scala类，该类必须序列化

\*/

class MyClass1 extends Serializable{

val field = "Hello"

def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(x => field + x) }//类似写成：rdd.map(x => this.field + x)

}

上述rdd.map(x => field + x)等同于rdd.map(x => this.field + x)，但这样写法会导致频繁访问外部对象，为了避免这种问题，最简单的方式是把field拷贝到本地变量，而不是去外部访问它。如下所示：

/\*\*

\* spark算子中用到scala类，该类必须序列化

\*/

class MyClass1 extends Serializable{

val field = "Hello"

def doStuff(rdd: RDD[String]): RDD[String] = {

rdd.map(x => field + x)

}//类似写成：rdd.map(x => this.field + x)

def doStuff2(rdd: RDD[String]): RDD[String] = {

val field\_ = this.field

rdd.map(x => field\_ + x)

}

}

## Spark Streaming

Spark Streaming是Spark核心API的扩展，它支持可伸缩、高吞吐量、可容错的数据流处理。数据可以从很多数据源获取，例如Kafka, Flume, Kinesis, or TCP sockets，并且可以使用如map, reduce, join and window等高级API方法组合处理。最后可以将处理结果存放到文件系统、数据库或在图表上展示。



在Spark内部，它的工作流程如下：Spark Streaming接收实时数据流，并将数据分成批处理，然后交给spark引擎处理，并最终分批生成处理结果。



### 创建一个Spark Streaming程序

引入：

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-streaming\_2.11</artifactId>

<version>2.2.0</version>

</dependency>

//Local模式即本地运行模式，在单个节点上运行，使用Local[N]启用一个或多个线程（最大值与核数有关）来运行spark程序，主要用于开发调试Spark应用程序

**var** sparkConf = **new** SparkConf().setMaster("local[4]").setAppName("SparkStreaming01");

//指定运行周期 运行周期为500毫秒，表示流式计算每间隔500毫秒执行一次。这个时间的设置需要综合考虑程序的延时需求和集群的工作负载，应该大于每次的运行时间。

//Milliseconds(x) Seconds(x)

**var** ssc = **new** StreamingContext(sparkConf, Milliseconds(5000));

//处理过程

//正式启动计算

ssc.start()

//等待执行结束

ssc.awaitTermination()

//停止

ssc.stop()

注意：当本地运行Spark Streaming程序时，不要使用“local”或“local [1]”作为master URL。 这两者意味着只有一个线程将用于在本地运行任务。 如果您正在使用基于receiver（e.g. sockets, Kafka, Flume, etc）的Input DStream，则单线程将用于运行接收器，不会留出线程来处理接收到的数据。 因此，当在本地运行时，始终使用“local [n]”作为master URL，其中n>要运行的接收器数量。

### DStream

Spark Streaming提供了一个高级抽象的概念：discretized stream or DStream，表示连续的数据流。

DStream可以由Kafka, Flume, and Kinesis等数据源输入流创建，也可以使用一些高级API在DStream上。

在Spark内部，DStream是一系列的RDDS。

Discretized Stream或者Dstream是SparkStreaming提供的最基础的抽象。它表示一系列的数据流，这些数据流可能来自于原始的输入，或者来源于其他的DStream转换生成的。Dstream本身是由一段 RDD组成，RDD本身是spark对于不可修改的分布式数据的抽象。每个RDD表示一个时间间隔范围内的数据，如下所示：



任何在DStream上执行的操作，最后都会映射到对应的RDD上面。对于内部的RDD操作最后都是由spark引擎来完成的。

### Input DStreams

Input DStreams表示从数据源接收的数据流。数据源可以分成以下两类：

* 基本数据源（Basic sources）：file systems, and socket connections
* 高级数据源（Advanced sources）：Kafka, Flume, Kinesis等

Spark Streaming应用程序中并行接收多个数据流，则可以创建多个 Input DStream。

### 自定义接收数据流receiverStream(……)

SparkStreaming 能够接收任意类型的流式数据，不单单只是内建的Flume,Kafka,Kinesis,files,sockets等等。当然若要支持其他的数据流，则需要开发者自定义程序来接收对应的数据源，只需实现自定义的Receiver类，并使用receiverStream方法将其放入集群中运行。

/\*\*

\*nc -lk 9999

\*/

**object** CustomReceiver {

**def** main(args: Array[*String*]) {

// Create the context with a 1 second batch size

**val** sparkConf = **new** SparkConf().setAppName("CustomReceiver").setMaster("local[4]")

**val** ssc = **new** StreamingContext(sparkConf, Seconds(1))

// Create an input stream with the custom receiver on target ip:port and count the

// words in input stream of \n delimited text (eg. generated by 'nc')

**val** lines = ssc.receiverStream(**new** CustomReceiver("192.168.0.120", 9999))

**val** words = lines.flatMap(\_.split(" "))

**val** wordCounts = words.map(x => (x, 1)).reduceByKey(\_ + \_)

wordCounts.print()

ssc.start()

ssc.awaitTermination()

}

}

//实现Receiver类

**class** CustomReceiver(host: *String*, port: Int) **extends** Receiver[*String*](StorageLevel.MEMORY\_AND\_DISK\_2) {

**def** onStart() {

//接收器开始运行时触发方法，在该方法内需要启动一个线程，用来接收数据。

**new** Thread("Socket Receiver") {

**override** **def** run() {

receive()

}

}.start()

}

//接收器结束运行时触发的方法，在该方法内需要确保停止接收数据（即完成停止接收数据）。

**def** onStop() {

// There is nothing much to do as the thread calling receive()

// is designed to stop by itself isStopped() returns false

}

/\*\* Create a socket connection and receive data until receiver is stopped \*/

**private** **def** receive() {

**var** socket: Socket = **null**

**var** userInput: *String* = **null**

**var** reader: BufferedReader = **null**;

**try** {

//连接socke，并获取输入流

socket = **new** Socket(host, port)

reader = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream(), StandardCharsets.UTF\_8))

//获取一行

userInput = reader.readLine()

**while**(!isStopped && userInput != **null**) {

//保存数据 一旦接收完数据，则必须要进行数据的存储，并交由SparkStreaming 来处理，

//Spark以store(data)方法来支持此流程。由于数据格式的不同，当然store方法必须要

//支持各种类型的数据存储。store方法是以一次存储一条记录或者一次性收集全部的序列化对象。

store(userInput)

//获取下一行

userInput = reader.readLine()

}

} **catch** {

**case** e: *Exception* =>

//关闭

**if**(reader != **null**) {

reader.close()

}

**if**(socket != **null**) {

socket.close()

}

//重启 在Receiver中任何的异常都需要捕获并正确得当地处理，以防止出现故障而不被发现。

//故障发生后receiver提供了restart(exception)方法用来重新启动该Receiver，

//重启过称为，异步调用onStop方法，然后调用onstart方法重新接收消息。

restart("Error:" + host + ":" + port, e)

//spark中还提供直接停止接收的方法，stop(exception) 调用该方法后会同步调用onStop方法，并终止接收数据流。

//提供了reportError(Exception)方法，用来将错误信息发送给Driver端，但是只是简单的上报错误，并没有调用onstop或者onstart方法。

}

}

}

### 接收Socket文本的数据流socketTextStream(……)

socketTextStream(……)是通过TCP Socket接收的文本数据创建了一个DStream。

**var** sparkConf = **new** SparkConf().setMaster("local[2]").setAppName("tcp01");

**var** ssc = **new** StreamingContext(sparkConf, Milliseconds(5000));

//从tcp中读取信息

**val** lines = ssc.socketTextStream("localhost", 9999)

**val** words = lines.flatMap(line => line.split(" ")) //lines.flatMap(\_.split(" "))

**val** pairs = words.map(word => (word, 1))

**val** wordCounts = pairs.reduceByKey((x, y) => x+y) //pairs.reduceByKey(\_ + \_)

wordCounts.print()

ssc.start() // Start the computation

ssc.awaitTermination() // Wait for the computation to terminate

ssc.stop()

### 接收Socket序列化的数据流socketStream(……)

socketStream函数与socketTextStream函数不同的是，socketStream的数据是序列化后的字节流，需要传递一个函数将字节流转换为对象；而socketTextStream的数据是文本类型的数据。

**object** TestReadTcp02 {

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

**var** sparkConf = **new** SparkConf().setMaster("local[2]").setAppName("tcp01");

**var** ssc = **new** StreamingContext(sparkConf, Milliseconds(5000));

//从tcp中读取信息

**val** lines = ssc.socketStream[*String*]("192.168.0.120", 9999, myDeserialize, StorageLevel.MEMORY\_AND\_DISK\_SER)

**val** words = lines.flatMap(line => line.split(" ")) //lines.flatMap(\_.split(" "))

**val** pairs = words.map(word => (word, 1))

**val** wordCounts = pairs.reduceByKey((x, y) => x+y) //pairs.reduceByKey(\_ + \_)

wordCounts.print()

ssc.start() // Start the computation

ssc.awaitTermination() // Wait for the computation to terminate

ssc.stop()

}

//将字节流转换为对象函数

**def** myDeserialize(data:InputStream): *Iterator*[*String*] = {

**val** obj = **null**

//将流反序列化成对象

Iterator(obj)

}

}

### rawSocketStream(……)

### 监控hadoop文件系统fileStream(……)与textFileStream(dataDirectory)

创建一个输入流，用于监控与hadoop兼容的文件系统，用于新文件，并使用给定的键值类型和输入格式来读取它们。文件必须通过将它们从同一个文件系统中的另一个位置“移动”到被监控的目录。

fileStream是Spark Streaming Basic Source的一种，用于“近实时”地分析HDFS（或者与HDFS API兼容的文件系统）指定目录（假设：dataDirectory）中新近写入的文件，dataDirectory中的文件需要满足以下约束条件：

1. 这些文件格式必须相同，如：统一为文本文件；
2. 这些文件在目录dataDirectory中的创建形式比较特殊：必须以原子方式被“移动”或“重命名”至目录dataDirectory中；
3. 一旦文件被“移动”或“重命名”至目录dataDirectory中，文件不可以被改变，例如：追加至这些文件的数据可能不会被处理。

fileStream的实现原理是比较简单的：以固定的时间间隔（duration）不断地探测目录（dataDirectory），每次探测时将时间段(now - duration, now]内新写入的文件（即文件的最近修改时间处于时间区间(now - duration, now]）封装为RDD交由Spark处理。

#### fileStream(……)

Spark提供了三个重载的fileStream方法：

1. def fileStream[K, V, F <: InputFormat[K, V]](directory: String)
2. def fileStream[K, V, F <: InputFormat[K, V]](directory: String, filter: Path => Boolean, newFilesOnly: Boolean)
3. def fileStream[K, V, F <: InputFormat[K, V]](directory: String, filter: Path => Boolean, newFilesOnly: Boolean, conf: Configuration)

参数说明：

* directory：指定待分析文件的目录；
* filter：用户指定的文件过滤器，用于过滤directory中的文件；
* newFilesOnly：应用程序启动时，目录directory中可能已经存在一些文件，如果newFilesOnly值为true，表示忽略这些文件；如果newFilesOnly值为false，表示需要分析这些文件；
* conf：用户指定的Hadoop相关的配置属性；

#### textFileStream(dataDirectory)

如果分析的文件是文本文件，Spark提供了一个便利的方法textFileStream(dataDirectory)。

def textFileStream(directory: String): DStream[String]

**object** TextFileStream01 {

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

**var** sparkConf = **new** SparkConf().setMaster("local[5]").setAppName("TextFileStream01");

**var** ssc = **new** StreamingContext(sparkConf, Milliseconds(10\*1000));

**val** lines = ssc.textFileStream("hdfs://192.168.0.119:9000/hadoop/txt/")

**val** words = lines.flatMap(line => line.split(" ")) //lines.flatMap(\_.split(" "))

**val** pairs = words.map(word => (word, 1))

**val** wordCounts = pairs.reduceByKey((x, y) => x+y) //pairs.reduceByKey(\_ + \_)

wordCounts.print()

ssc.start() // Start the computation

ssc.awaitTermination() // Wait for the computation to terminate

ssc.stop()

}

}

### queueStream(queueOfRDDs)

### Checkpointing

一个 Streaming Application 往往需要7\*24不间断的跑，所以需要有抵御意外的能力（比如机器或者系统挂掉，JVM crash等）。为了让这成为可能，Spark Streaming需要 checkpoint 足够多信息至一个具有容错设计的存储系统才能让 Application 从失败中恢复。Spark Streaming 会 checkpoint 两种类型的数据。

1. Metadata（元数据） checkpointing - 保存定义了 Streaming 计算逻辑至类似 HDFS 的支持容错的存储系统。用来恢复driver，元数据包括：配置（用于创建该 streaming application 的所有配置）、DStream 操作（DStream 一些列的操作）、未完成的batches（那些提交了 job 但尚未执行或未完成的 batches）
2. Data checkpointing - 保存已生成的RDDs至可靠的存储。这在某些 stateful 转换中是需要的，在这种转换中，生成 RDD 需要依赖前面的 batches，会导致依赖链随着时间而变长。为了避免这种没有尽头的变长，要定期将中间生成的 RDDs 保存到可靠存储来切断依赖链。

metadata checkpointing主要用来恢复 driver；而 RDD数据的 checkpointing 对于stateful 转换操作是必要的。

启用checkpoint满足以下任一条件：

* 使用了stateful转换：如果 application中使用了updateStateByKey或reduceByKeyAndWindow等stateful操作，必须提供checkpoint目录来允许定时的RDD checkpoint；updateStateByKey的使用需要checkpoint,隔几次记录一次到磁盘中

UpdateStateByKey的主要功能

1、Spark Streaming中为每一个Key维护一份state状态，这个state类型可以是任意类型的的， 可以是一个自定义的对象，那么更新函数也可以是任意类型的。

2、通过更新函数对该key的状态不断更新，对于每个新的batch而言，Spark Streaming会在使用updateStateByKey的时候为已经存在的key进行state的状态更新（对于每个新出现的key，会同样的执行state的更新函数操作）

3、如果要不断的更新每个key的state，就涉及到了状态的保存和容错，这个时候就需要开启checkpoint机制和功能 ( one.checkpoint(Durations.seconds(10)) //容错更好就需要牺牲性能，容错不需要太高，时间可以设置的长一些

* 希望能从意外中恢复driver

如果streaming app没有stateful操作，也允许driver挂掉后再次重启的进度丢失，就没有启用checkpoint的必要了。

启用 checkpoint：需要设置一个支持容错的、可靠的文件系统（如HDFS、s3等）目录来保存checkpoint数据。通过调用streamingContext.checkpoint(checkpointDirectory)来完成。另外，如果你想让你的application能从driver失败中恢复，你的application要满足：

* 若application为首次重启，将创建一个新的 StreamContext 实例；
* 如果application是从失败中重启，将会从checkpoint目录导入checkpoint数据来重新创建StreamingContext实例。

**import** org.apache.spark.streaming.StreamingContext

**import** org.apache.spark.SparkConf

**import** org.apache.spark.streaming.Milliseconds

**import** org.apache.kafka.common.serialization.StringDeserializer

**import** org.apache.spark.streaming.kafka010.KafkaUtils

**import** org.apache.spark.streaming.kafka010.ConsumerStrategies.\_

**import** org.apache.spark.streaming.kafka010.LocationStrategies.\_

**object** Test {

**def functionToCreateContext(): StreamingContext = {**

**var sparkConf = new SparkConf().setMaster("local[2]").setAppName("checkpoint01");**

**var ssc = new StreamingContext(sparkConf, Milliseconds(5000));**

**ssc.checkpoint("/checkpoint/ssyj") // 设置在HDFS上的checkpoint目录**

**return ssc**

**}**

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

**val** ssc = StreamingContext.getOrCreate("/checkpoint/ssyj", functionToCreateContext)

**val** kafkaParams = Map[*String*, Object]("bootstrap.servers" -> "192.168.1.100:9092",

"key.deserializer" -> classOf[StringDeserializer],

"value.deserializer" -> classOf[StringDeserializer],

"group.id" -> "kermit",

"auto.offset.reset" -> "latest",

"enable.auto.commit" -> (**true**: java.lang.Boolean))

//读取消息

**var** stream = KafkaUtils.createDirectStream[*String*, *String*](ssc, PreferConsistent,

Subscribe[*String*, *String*](Array("ssyj\_out"), kafkaParams))

//设置通过间隔时间，定时持久checkpoint到hdfs上

stream.checkpoint(Milliseconds(5000\*5))

**var** words = stream.flatMap(s => s.value().split(" "))

**var** pairs = words.map(word => (word, 1))

**var** wordCounts = pairs.reduceByKey(\_ + \_)

wordCounts.print()

ssc.start() // Start the computation

ssc.awaitTermination() // Wait for the computation to terminate

ssc.stop()

}

}

随着 streaming application 的持续运行，checkpoint 数据占用的存储空间会不断变大。因此，需要小心设置checkpoint 的时间间隔。设置得越小，checkpoint 次数会越多，占用空间会越大；如果设置越大，会导致恢复时丢失的数据和进度越多。一般推荐设置为 batch duration 的5~10倍。

### Window Operations

Spark Streaming提供窗口操作（Window Operation），如下图所示：



上图中，红色实线表示窗口当前的滑动位置，虚线表示前一次窗口位置，窗口每滑动一次，落在该窗口中的RDD被一起同时处理，生成一个窗口DStream（windowed DStream），窗口操作需要设置两个参数（这两个参数必须是原始DStream 批处理间隔（batch interval）的整数倍）

* 窗口长度（window length），即窗口的持续时间，上图中的窗口长度为3;
* 滑动间隔（sliding interval），窗口操作执行的时间间隔，上图中的滑动间隔为2;

窗口操作肯定会有很多的重复计算，但在实际应用中，需要这种重复计算，而不需要避免重复计算，如

* 博中每小时的热门微博，每隔1分钟计算一次，相当于重复计算了59分钟的内容
* 商家想看前5分钟的销售额，每隔30秒看一次，也需要基于窗口的操作

**import** org.apache.spark.streaming.StreamingContext

**import** org.apache.spark.SparkConf

**import** org.apache.spark.streaming.Milliseconds

**import** org.apache.spark.streaming.kafka010.KafkaUtils

**import** org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent

**import** org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe

**import** org.apache.kafka.common.serialization.StringDeserializer

**import** org.apache.spark.streaming.kafka010.HasOffsetRanges

**import** org.apache.spark.streaming.kafka010.OffsetRange

**import** org.apache.spark.TaskContext

**import** org.apache.kafka.common.serialization.StringDeserializer

**import** org.apache.spark.streaming.Seconds

**object** TestKafka010Consumer03 {

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

**var** sparkConf = **new** SparkConf().setMaster("local[2]").setAppName("consumer03");

**var** ssc = **new** StreamingContext(sparkConf, Seconds(5));

**val** kafkaParams = Map[*String*, Object]("bootstrap.servers" -> "192.168.1.100:9092",

"key.deserializer" -> classOf[StringDeserializer],

"value.deserializer" -> classOf[StringDeserializer],

"group.id" -> "kermit",

"auto.offset.reset" -> "latest",

"enable.auto.commit" -> (**true**: java.lang.Boolean))

//读取消息

**var** stream = KafkaUtils.createDirectStream[*String*, *String*](ssc, PreferConsistent,

Subscribe[*String*, *String*](Array("ssyj\_out"), kafkaParams))

**var** words = stream.flatMap(s => s.value().split(" "))

**var** pairs = words.map(word => (word, 1))

//每隔十秒计算前三十秒的数据 这两个参数必须是原始DStream 批处理间隔（batch interval）的整数倍

**val** wordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b), Seconds(30), Seconds(10))

wordCounts.print()

ssc.start() // Start the computation

ssc.awaitTermination() // Wait for the computation to terminate

ssc.stop()

}

}

## Spark Streaming整合Kafka0.10

### 单个DStream消费者案例

//设置程序名

**val** sparkConf = **new** SparkConf().setMaster("local[2]").setAppName("ssyj")

//指定运行周期 运行周期为500毫秒，表示流式计算每间隔500毫秒执行一次。这个时间的设置需要综合考虑程序的延时需求和集群的工作负载，应该大于每次的运行时间。

//Milliseconds(x) Seconds(x)

**val** ssc = **new** StreamingContext(sparkConf, Milliseconds(5000))

//bootstrap.servers：kafka节点地址，ip1:port1,ip2:port2

//group.id：用来标识消费者，程序会启动多个线程读取，用group.id标识这些消费者都是“一样”的，一个线程读取完毕后，偏移量往前移，这样可以避免重复读取，

//auto.offset.reset：重置偏移量

// earliest：当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，从头开始消费

// latest：当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，消费新产生的该分区下的数据

// none：topic各分区都存在已提交的offset时，从offset后开始消费；只要有一个分区不存在已提交的offset，则抛出异常

//enable.auto.commit：是否自动提交已拉取消息的offset。提交offset即视为该消息已经成功被消费，该组下的Consumer无法再拉取到该消息。

**val** kafkaParams = Map[*String*, Object]("bootstrap.servers" -> "192.168.1.100:9092",

"key.deserializer" -> classOf[StringDeserializer],

"value.deserializer" -> classOf[StringDeserializer],

"group.id" -> "kermit",

"auto.offset.reset" -> "latest",

"enable.auto.commit" -> (**true**: java.lang.Boolean))

//读取消息

**var** stream = KafkaUtils.createDirectStream[*String*, *String*](ssc, PreferConsistent,

Subscribe[*String*, *String*](Array("ssyj\_out"), kafkaParams))

//每一个stream都是一个ConsumerRecord

//stream.map(s =>(s.key(),s.value())).print();

//对每次抓取的数据进行处理，每次都是一个RDD

stream.foreachRDD { rdd =>

**val** offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges

//1.按分区划分数据，并循环抓取

rdd.foreachPartition(

partitionOfRecords => {

//偏移量

**val** o: OffsetRange = offsetRanges(TaskContext.get.partitionId)

println(s"${o.topic} ${o.partition} ${o.fromOffset} ${o.untilOffset}")

//循环一个分区的信息

**while** (partitionOfRecords.hasNext) {

**val** msg = partitionOfRecords.next()

println("message: " + msg.value())

}

})

//2.循环rdd

rdd.foreach(f => println(f.value()))

}

//对每次抓取的数据进行处理，每次都是一个RDD

stream.foreachRDD( rdd =>

//1.按分区划分数据，并循环抓取

rdd.foreachPartition( partitionOfRecords => {

partitionOfRecords.foreach{

**case** x => {

println(x.value())

}

}

})

)

//正式启动计算 ssc.stop()

ssc.start()

//等待执行结束

ssc.awaitTermination()

//停止

ssc.stop()

### （待测试）多个DStream消费者案例

在Kafka中，一个主题（topic）可以有N个分区。理想的情况下，我们希望在多个分区上并行读取。

kafka主题的分区数量对于性能来说非常重要，而这个值一般是消费者parallelism的最大数量（消费者最大并发数）：如果一个主题拥有N个分区，那么你的应用程序最大程度上只能进行N个线程的并行。

可以通过设置kafka组ID来标识一组相同的消费者，同一组消费者中的所有消费者将分别从一个指定Kafka主题中的读取消息，同时，同一个消费组中所有消费者从话题中读取的线程数最大值即是N（等同于分区的数量），多余的线程将会闲置。

例如：

应用程序使用“terran”消费者组id对一个名为“zerg.hydra”的kafka主题进行读取，这个主题拥有10个分区。如果你的消费者应用程序只配置一个线程对这个主题进行读取，那么这个线程将从10个分区中进行读取。

同上，但是这次你会配置5个线程，那么每个线程都会从2个分区中进行读取。

同上，这次你会配置10个线程，那么每个线程都会负责1个分区的读取。

同上，但是这次你会配置多达14个线程。那么这14个线程中的10个将平分10个分区的读取工作，剩下的4个将会被闲置。

可以从以下两个方面控制线程数：

* **Input DStreams的数量：**使用多个input DStreams将跨多个节点并行进行读取操作。
* **Input DStreams上的消费者线程数量：在同一个节点上开启多个消费者线程。**

**控制input DStreams的数量（以下是多个input DStreams线程共用一个消费者，存在线程不安全隐患，针对每个线程必须开启一个消费者）**：

**val** kafkaParams = Map[*String*, Object]("bootstrap.servers" -> "192.168.1.102:9092",

"key.deserializer" -> classOf[StringDeserializer],

"value.deserializer" -> classOf[StringDeserializer],

"group.id" -> "kermit",

"auto.offset.reset" -> "latest",

"enable.auto.commit" -> (**true**: java.lang.Boolean))

//读取消息

**val** numInputDStreams = 5

**var** streams = (1 to numInputDStreams).map { \_ => KafkaUtils.createDirectStream[*String*, *String*](ssc, PreferConsistent,

Subscribe[*String*, *String*](Array("ssyj\_out"), kafkaParams))}

//合并，窄依赖

**val** unifiedStream = ssc.union(streams)

//重新分区

unifiedStream.repartition(5)

**控制每个input DStream上消费者线程的数量：**

### 普通生产者案例

**val** props = **new** HashMap[*String*, Object]()

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "192.168.1.102:9092")

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer")

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer")

**val** producer = **new** KafkaProducer[*String*, *String*](props)

**var** list = List("a", "b", "c", "d", "e", "f")

**for**(element <- list){

**val** message = **new** ProducerRecord[*String*, *String*]("ssyj\_out", **null**, element)

producer.send(message)

}

producer.close()

### 从Kafka获取消息，再发送出去（缺点：对于每个partition的每条记录，都需要创建KafkaProducer）

**var** sparkConf = **new** SparkConf().setMaster("local[2]").setAppName("producer01");

**var** ssc = **new** StreamingContext(sparkConf, Milliseconds(5000));

**val** kafkaParams = Map[String, Object]("bootstrap.servers" -> "192.168.1.102:9092",

"key.deserializer" -> classOf[StringDeserializer],

"value.deserializer" -> classOf[StringDeserializer],

"group.id" -> "kermit",

"auto.offset.reset" -> "latest",

"enable.auto.commit" -> (**true**: java.lang.Boolean))

//读取消息

**var** stream = KafkaUtils.createDirectStream[String, String](ssc, PreferConsistent,

Subscribe[String, String](Array("ssyj\_out"), kafkaParams))

//对每次抓取的数据进行处理，每次都是一个RDD

stream.foreachRDD( rdd =>

//按分区划分数据，并循环抓取

rdd.foreachPartition( partitionOfRecords => {

partitionOfRecords.foreach{

**case** x => {

//发送出去，形成接收-发送-接收...循环执行

**val** props = **new** HashMap[String, Object]()

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "192.168.1.102:9092")

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer")

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer")

**val** producer = **new** KafkaProducer[String, String](props)

**val** message = **new** ProducerRecord[String, String]("ssyj\_out", **null**, x.value())

producer.send(message)

producer.close()

}

}

})

)

ssc.start() // Start the computation

ssc.awaitTermination() // Wait for the computation to terminate

ssc.stop()

### 封装并序列化KafkaProducer

**5.7.4**的缺点很明显，对于每个partition的每条记录，都需要创建KafkaProducer，然后利用producer进行输出操作。我们不能将KafkaProducer的新建任务放在foreachPartition外边，因为KafkaProducer是不可序列化的（not serializable）。显然这种做法是不灵活且低效的，因为每条记录都需要建立一次连接。

**import** org.apache.spark.SparkConf

**import** org.apache.spark.streaming.StreamingContext

**import** org.apache.spark.streaming.Milliseconds

**import** org.apache.spark.streaming.kafka010.KafkaUtils

**import** org.apache.kafka.common.serialization.StringDeserializer

**import** org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent

**import** org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe

**import** java.util.HashMap

**import** org.apache.kafka.clients.producer.ProducerConfig

**import** org.apache.kafka.clients.producer.KafkaProducer

**import** org.apache.kafka.clients.producer.ProducerRecord

**import** java.util.concurrent.Future

**import** org.apache.kafka.clients.producer.RecordMetadata

**import** org.apache.spark.broadcast.Broadcast

//构造方法中的参数为匿名函数并返回KafkaProducer[K, V]

**class** KafkaSink[K, V](createProducer: () => KafkaProducer[K, V]) **extends** Serializable {

//创建KafkaProducer，懒加载

**lazy** **val** producer = createProducer()

/\*\*

\* 发送方法

\* topic：主题

\* partition：指定分区

\* key：消息key

\* value：消息value

\*/

**def** send(topic: *String*, partition: Integer, key: K, value: V): Future[RecordMetadata] =

producer.send(**new** ProducerRecord[K, V](topic, partition, key, value))

/\*\*

\* 发送方法

\* topic：主题

\* key：消息key

\* value：消息value

\*/

**def** send(topic: *String*, key: K, value: V): Future[RecordMetadata] =

producer.send(**new** ProducerRecord[K, V](topic, key, value))

/\*\*

\* 发送方法

\* topic：主题

\* value：消息value

\*/

**def** send(topic: *String*, value: V): Future[RecordMetadata] =

producer.send(**new** ProducerRecord[K, V](topic, value))

}

//KafkaSink类的伴生对象（相当于单例对象），必须在同一个源文件里定义类和它的伴生对象

//类和它的伴生对象可以互相访问其私有成员

**object** KafkaSink {

//apply方法接受构造参数变成对象

**def** apply[K, V](config: HashMap[*String*, Object]): KafkaSink[K, V] = {

//匿名函数，

**val** createProducerFunc = () => {

//创建生产者

**val** producer = **new** KafkaProducer[K, V](config)

//注册一个JVM关闭的钩子，以确保在JVM退出前将生产者关闭。

//这个钩子可以在以下几种场景被调用：

//1）程序正常退出

//2）使用System.exit()

//3）终端使用Ctrl+C触发的中断

//4）系统关闭

//5）使用Kill pid命令干掉进程

sys.addShutdownHook {

// Ensure that, on executor JVM shutdown, the Kafka producer sends

// any buffered messages to Kafka before shutting down.

producer.close()

}

//返回

producer

}

//创建序列化对象

**new** KafkaSink[K, V](createProducerFunc)

}

//实现兼容个java.util.Properties类型的参数设置

//**def** apply[K, V](config: java.util.Properties): KafkaSink[K, V] = {}

}

**object** TestKafka010Producer03 {

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

**var** sparkConf = **new** SparkConf().setMaster("local[2]").setAppName("producer01");

**var** ssc = **new** StreamingContext(sparkConf, Milliseconds(5000));

//kafka参数

**val** props = **new** HashMap[*String*, Object]()

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "192.168.1.102:9092")

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer")

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer")

//创建生产者并广播到每个节点

**val** kafkaProducer = ssc.sparkContext.broadcast(KafkaSink[*String*, *String*](props))

//发送消息

**val** msg = "Start the computation";

kafkaProducer.value.send("ssyj\_out", **null**, msg)

//ssc.start() // Start the computation

//ssc.awaitTermination() // Wait for the computation to terminate

//ssc.stop()

}

}

### 接收后发送（序列化后）

//构造方法中的参数为匿名函数并返回KafkaProducer[K, V]

**class** KafkaSink[K, V](createProducer: () => KafkaProducer[K, V]) **extends** Serializable {

//创建KafkaProducer，懒加载

**lazy** **val** producer = createProducer()

/\*\*

\* 发送方法

\* topic：主题

\* partition：指定分区

\* key：消息key

\* value：消息value

\*/

**def** send(topic: *String*, partition: Integer, key: K, value: V): Future[RecordMetadata] =

producer.send(**new** ProducerRecord[K, V](topic, partition, key, value))

/\*\*

\* 发送方法

\* topic：主题

\* key：消息key

\* value：消息value

\*/

**def** send(topic: *String*, key: K, value: V): Future[RecordMetadata] =

producer.send(**new** ProducerRecord[K, V](topic, key, value))

/\*\*

\* 发送方法

\* topic：主题

\* value：消息value

\*/

**def** send(topic: *String*, value: V): Future[RecordMetadata] =

producer.send(**new** ProducerRecord[K, V](topic, value))

}

//KafkaSink类的伴生对象（相当于单例对象），必须在同一个源文件里定义类和它的伴生对象

//类和它的伴生对象可以互相访问其私有成员

**object** KafkaSink {

//apply方法接受构造参数变成对象

**def** apply[K, V](config: HashMap[*String*, Object]): KafkaSink[K, V] = {

//匿名函数，

**val** createProducerFunc = () => {

//创建生产者

**val** producer = **new** KafkaProducer[K, V](config)

//注册一个JVM关闭的钩子，以确保在JVM退出前将生产者关闭。

//这个钩子可以在以下几种场景被调用：

//1）程序正常退出

//2）使用System.exit()

//3）终端使用Ctrl+C触发的中断

//4）系统关闭

//5）使用Kill pid命令干掉进程

sys.addShutdownHook {

// Ensure that, on executor JVM shutdown, the Kafka producer sends

// any buffered messages to Kafka before shutting down.

producer.close()

}

//返回

producer

}

//创建序列化对象

**new** KafkaSink[K, V](createProducerFunc)

}

**def** apply[K, V](config: java.util.Properties): KafkaSink[K, V] = {

//匿名函数，

**val** createProducerFunc = () => {

//创建生产者

**val** producer = **new** KafkaProducer[K, V](config)

//注册一个JVM关闭的钩子，以确保在JVM退出前将生产者关闭。

sys.addShutdownHook {

producer.close()

}

//返回

producer

}

//创建序列化对象

**new** KafkaSink[K, V](createProducerFunc)

}

}

**object** TestKafka010Producer03 {

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

**var** sparkConf = **new** SparkConf().setMaster("local[2]").setAppName("producer03");

**var** ssc = **new** StreamingContext(sparkConf, Milliseconds(2000));

//生产者kafka参数

**val** props = **new** HashMap[*String*, Object]()

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "192.168.1.102:9092")

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer")

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, "org.apache.kafka.common.serialization.StringSerializer")

//创建生产者并广播到每个节点

**val** kafkaProducer = ssc.sparkContext.broadcast(KafkaSink[*String*, *String*](props))

//消费者kafka参数

**val** kafkaParams = Map[*String*, Object]("bootstrap.servers" -> "192.168.1.102:9092",

"key.deserializer" -> classOf[StringDeserializer],

"value.deserializer" -> classOf[StringDeserializer],

"group.id" -> "kermit",

"auto.offset.reset" -> "latest",

"enable.auto.commit" -> (**true**: java.lang.Boolean))

//读取消息

**var** streams = KafkaUtils.createDirectStream[*String*, *String*](ssc, PreferConsistent,

Subscribe[*String*, *String*](Array("ssyj\_out"), kafkaParams))

//对每次抓取的数据进行处理，每次都是一个RDD

streams.foreachRDD { rdd =>

rdd.foreachPartition( partitionOfRecords => {//每个分区

partitionOfRecords.foreach {//每个分区的每条记录

**case** x => {

kafkaProducer.value.send("ssyj\_out", **null**, x.value())

}

}

}

)

}

ssc.start() // Start the computation

ssc.awaitTermination() // Wait for the computation to terminate

ssc.stop()

}

}

### （待测试）由多个DStream消费者接收数据，然后由封装并序列化的KafkaProducer生产者将数据再发送出去

## Spark SQL, DataFrames and Datasets

Spark SQL 是一个用来处理结构化数据的spark组件。它提供了一个叫做DataFrames的可编程抽象数据模型，并且可被视为一个分布式的SQL查询引擎。主要用于结构化数据处理和对Spark数据执行类SQL的查询。通过Spark SQL，可以针对不同格式的数据执行ETL操作（如JSON，Parquet，数据库）然后完成特定的查询操作。SparkSQL的前身是Shark，给熟悉RDBMS但又不理解MapReduce的技术人员提供快速上手的工具，Hive应运而生，Shark是当时唯一运行在Hadoop上的SQL-on-Hadoop工具。

但是，随着Spark的发展，对于野心勃勃的Spark团队来说，Shark对于Hive的太多依赖（如采用Hive的语法解析器、查询优化器等等），制约了Spark的One Stack Rule Them All的既定方针，制约了Spark各个组件的相互集成，所以提出了SparkSQL项目。SparkSQL抛弃原有Shark的代码，汲取了Shark的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive兼容性等，重新开发了SparkSQL代码；由于摆脱了对Hive的依赖性，SparkSQL无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便。

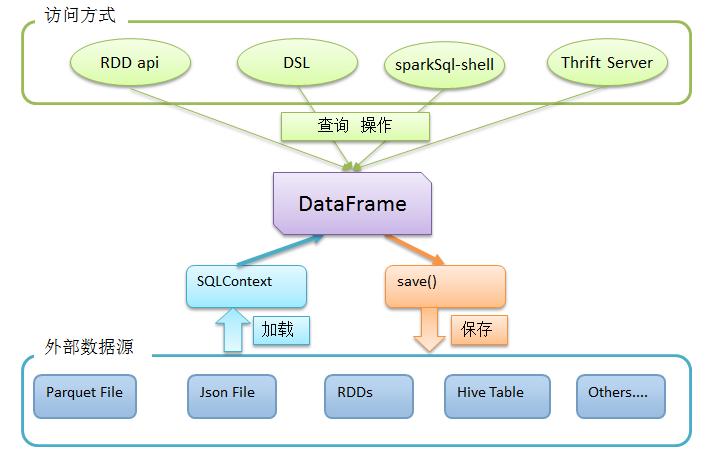
2014年6月1日Shark项目和SparkSQL项目的主持人Reynold Xin宣布：停止对Shark的开发，团队将所有资源放SparkSQL项目上，至此，Shark的发展画上了句话，但也因此发展出两个直线：SparkSQL和Hive on Spark。

### Spark SQL工作原理

首先，利用sqlContext从外部数据源加载数据为DataFrame

然后，利用DataFrame上丰富的api进行查询、转换

最后，将结果进行展现或存储为各种外部数据形式



DataFrame是由“命名列”（类似关系表的字段定义）所组织起来的一个分布式数据集合。你可以把它看成是一个关系型数据库的表。

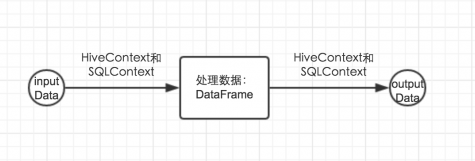
DataFrame可以通过多种来源创建：结构化数据文件，hive的表，外部数据库，或者RDDs。

sqlContext支持从各种各样的数据源中创建DataFrame，内置支持的数据源有parquetFile，jsonFile，外部数据库，hive表，RDD等。

### sparkSQL层级

用sparkSQL来解决我们的需求时，就经历了三步：读入数据 -> 对数据进行处理 -> 写入最后结果。

这三个步骤用的主要类有三个：读入数据和写入最后结果用到两个类HiveContext和SQLContext，对数据进行处理用到的是DataFrame类，此类是你把数据从外部读入到内存后，数据在内存中进行存储的基本数据结构。



### SQLContext

#### SQLContext连接数据库（以MySql为例）

连接数据库时，需用使用到以下参数属性：

|  |  |  |
| --- | --- | --- |
| **序号** | **属性名** | **描述** |
| 1 | url | 数据库连接地址 |
| 2 | user | 用户名 |
| 3 | password | 密码 |
| 4 | dbtable | 需要操作的数据库表名，也可以是子查询，但必须带括号且起别名，如(select id, login\_name, user\_name from user) u |
| 5 | driver | 驱动类 |
| 6 | partitionColumn, lowerBound, upperBound | 指定分区字段，并字段值的范围 |
| 7 | numPartitions | 分成多少个区 |
| 8 | fetchsize | 循环每次抓取数据返回的行数（默认值根据驱动设定），该属性参数仅用于查询。 |
| 9 | batchsize | 批量插入，每次插入的条数，默认1000条，该属性参数仅用于写操作。 |
| 10 | 事务隔离级别 | * 未提交读（read uncommitted）：当事务A更新某条数据时，不容许其他事务来更新该数据，但可以读取。 * 提交读（read committed）：当事务A更新某条数据时，不容许其他事务进行任何操作包括读取，但事务A读取时，其他事务可以进行读取、更新 * 重复读（repeatable read）：当事务A更新数据时，不容许其他事务进行任何操作，但当事务A进行读取时，其他事务只能读取，不能更新。 * 序列化（serializable）：最严格的隔离级别，事务必须依次进行 |
| 11 | truncate | 保存模式，该属性参数仅用于写操作：  SaveMode.Append  SaveMode.Overwrite  SaveMode.Ignore |
| 12 | createTableOptions |  |
| 13 | createTableColumnTypes | 定义列及类型：jdbcDF.write  .option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)")  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties) |

##### 不指定查询条件

调用方法：def jdbc(url: String, table: String, properties: Properties): DataFrame

**val** conf = **new** SparkConf().setAppName("SQLContext\_Mysql01\_1").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//url：连接数据库串

//table：操作的表

//properties：参数

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

**val** userDataFrame = sqlContext.read.jdbc(url, "user", properties)

//数量

println(userDataFrame.count())

//打印

userDataFrame.rdd.foreach { x => println(x.get(1) + "==" + x.get(2)) }

sc.stop()

##### 指定查询条件\_01

**val** conf = **new** SparkConf().setAppName("SQLContext\_Mysql01\_3").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//url: String：连接数据库串

//table: String：操作的表

//connectionProperties: Properties：参数

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

**val** userDataFrame = sqlContext.read.jdbc(url, "( select login\_name,user\_name,dept\_name from user where login\_name like '%08%' ) u", properties)

.select("login\_name", "user\_name", "dept\_name")

//记录数量

println(userDataFrame.count())

//分区数量

println(userDataFrame.rdd.partitions.size)

//打印

userDataFrame.rdd.foreachPartition {

partitionOfRecords => {

partitionOfRecords.foreach {

x => println(x.get(1) + "==" + x.get(2))

}

}

}

sc.stop()

##### 指定查询条件\_02

**val** conf = **new** SparkConf().setAppName("SQLContext\_Mysql03\_2").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//url: String：连接数据库串

//table: String：操作的表

//predicates: Array[String] 根据任意字段进行分区

//connectionProperties: Properties：参数

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

**val** userDataFrame = sqlContext.read.jdbc(url, "user", Array[String](" login\_name like '%08%' "), properties)

//记录数量

println(userDataFrame.count())

//分区数量

println(userDataFrame.rdd.partitions.size)

//打印

userDataFrame.rdd.foreach {

x => println(x.get(1) + "==" + x.get(2))

}

sc.stop()

##### 查询指定列

**val** conf = **new** SparkConf().setAppName("SQLContext\_Mysql01\_3").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//url：连接数据库串

//table：操作的表

//properties：参数

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc?user=root&password=jcbkzhzx110"

**val** properties = **new** Properties()

**val** userDataFrame = sqlContext.read.jdbc(url, "user", properties).select("login\_name", "user\_name", "dept\_name")

//数量

println(userDataFrame.count())

//打印

userDataFrame.rdd.foreach { x => println(x.get(0) + "==" + x.get(1) + "==" + x.getAs("dept\_name")) }

sc.stop()

##### （并发读取）指定数据库字段的范围

调用方法：def jdbc(

    url: String,

    table: String,

    columnName: String,

    lowerBound: Long,

    upperBound: Long,

    numPartitions: Int,

connectionProperties: Properties): DataFrame

**val** conf = **new** SparkConf().setAppName("SQLContext\_Mysql02").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//url: String：连接数据库串

//table: String：操作的表

//columnName: String：指定列 通过指定数据库中某个字段的范围（这个字段必须是数字）

//lowerBound: Long：上限

//upperBound: Long：下限

//numPartitions: Int：分区的个数

//connectionProperties: Properties：参数

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc?user=root&password=jcbkzhzx110"

**val** properties = **new** Properties()

//id取值范围6000至30000

**val** userDataFrame = sqlContext.read.jdbc(url, "user", "id", 6000,

30000, 2, properties)

//数量

println(userDataFrame.count())

//打印

userDataFrame.rdd.foreach { x => println(x.get(1) + "==" + x.get(2)) }

sc.stop()

##### （并发读取）根据任意字段进行分区

def jdbc(

    url: String,

    table: String,

    predicates: Array[String],

connectionProperties: Properties): DataFrame

**val** conf = **new** SparkConf().setAppName("SQLContext\_Mysql03").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//url: String：连接数据库串

//table: String：操作的表

//predicates: Array[String] 根据任意字段进行分区

//connectionProperties: Properties：参数

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc?user=root&password=jcbkzhzx110"

**val** properties = **new** Properties()

//根据id分5个区：

// id > 0 and id <= 10000

// id > 10000 and id <= 20000

// id > 20000 and id <= 30000

// id > 30000 and id <= 40000

// id > 40000

**var** predicates = Array[*String*]("id > 0 and id <= 10000", "id > 10000 and id <= 20000",

"id > 20000 and id <= 30000", "id > 30000 and id <= 40000", "id > 40000")

**val** userDataFrame = sqlContext.read.jdbc(url, "user", predicates, properties)

//记录数量

println(userDataFrame.count())

//分区数量

println(userDataFrame.rdd.partitions.size)

//打印

userDataFrame.rdd.foreachPartition {

partitionOfRecords => {

partitionOfRecords.foreach {

x => println(x.get(1) + "==" + x.get(2))

}

}

}

sc.stop()

##### 通过load的方式来读取数据

**val** conf = **new** SparkConf().setAppName("SQLContext\_Mysql04").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**val** userDataFrame = sqlContext.read.format("jdbc").options(

Map("url" -> "jdbc:mysql://localhost:3306/dc",

"user" -> "root",

"password" -> "jcbkzhzx110",

"dbtable" -> " (select id, login\_name, user\_name from user) u ",

"fetchsize" -> "2000")

).load().select("id", "login\_name", "user\_name")

//记录数量

println(userDataFrame.count())

//打印

userDataFrame.rdd.foreach {

x => println(x.get(0) + "==" + x.get(1) + "==" + x.get(2))

}

sc.stop()

### HiveContext

### DataFrame

#### show：展示数据

1. show()：展示前20条数据
2. show(numRows: Int)：显示numRows条
3. show(truncate: Boolean)：是否最多只显示20个字符，默认为true。
4. show(numRows: Int, truncate: Boolean)：综合前面的显示记录条数，以及对过长字符串的显示格式。
5. show(numRows: Int, truncate: Int)：显示numRows条，且显示前truncate个字符

**val** conf = **new** SparkConf().setAppName("DataFrame01").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

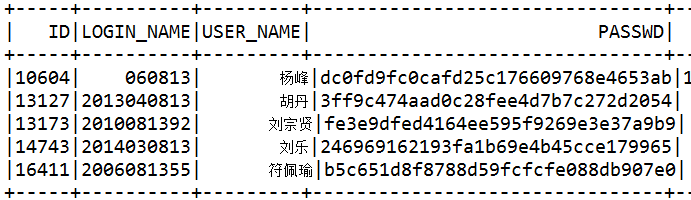
properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

**val** userDataFrame = sqlContext.read.jdbc(url, "user", Array[*String*](" login\_name like '%0813%' "),

properties).**show(5, 100)**

sc.stop()



#### collect：获取所有数据到数组

**val** conf = **new** SparkConf().setAppName("DataFrame02").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//获取所有数据到数组

**val** userCollect = sqlContext.read.jdbc(url, "user", Array[*String*](" login\_name like '%0813%' "),

properties).collect()

//打印

userCollect.foreach {

x => println(x.get(0) + "==" + x.get(1) + "==" + x.get(2))

}

sc.stop()

#### collectAsList：获取所有数据到List

**val** conf = **new** SparkConf().setAppName("DataFrame03").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//获取所有数据到List

**val** userList = sqlContext.read.jdbc(url, "user", Array[*String*](" login\_name like '%0813%' "),

properties).collectAsList()

//打印

**for**(i <- 0 until userList.size()) {

println(userList.get(i).get(0) + "==" + userList.get(i).get(1) + "==" + userList.get(i).get(2))

}

sc.stop()

#### describe(cols: String\*)：获取指定字段的统计信息

可以动态的传入一个或多个字段名，用于统计数值类型字段的统计值，比如count（记录总数）， mean（平均数），stddev（标准差），min（最小值），max（最大值）等，结果仍然为DataFrame对象。可以用在金额、数量等字段。

**val** conf = **new** SparkConf().setAppName("DataFrame04").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

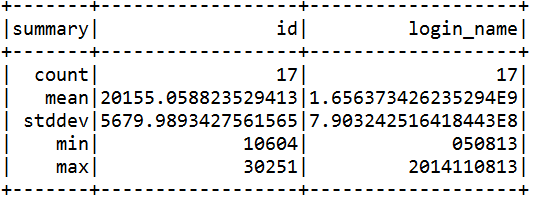
properties.setProperty("password", "jcbkzhzx110")

//统计id,login\_name字段

**val** userList = sqlContext.read.jdbc(url, "user", Array[*String*](" login\_name like '%0813%' "),

properties).describe("id" , "login\_name").show()

sc.stop()



#### first, head, take, takeAsList：获取若干行记录

1. first：获取第一行记录
2. head：获取第一行记录，head(n: Int)获取前n行记录
3. take(n: Int) ：获取前n行数据
4. takeAsList(n: Int) ：获取前n行数据，并以List的形式展现

**val** conf = **new** SparkConf().setAppName("DataFrame05").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//获取数据，返回DataFrame类型的数据

**val** userDataFrame = sqlContext.read.jdbc(url, "user", Array[*String*](" login\_name like '%0813%' "),

properties)

//获取第一行

**var** firtRow = userDataFrame.first()

println("firt：" + firtRow.get(0) + "==" + firtRow.get(1) + "==" + firtRow.get(2))

//获取第一行

**var** headRow = userDataFrame.head()

println("head：" + headRow.get(0) + "==" + headRow.get(1) + "==" + headRow.get(2))

//获取前n行

**var** headRows = userDataFrame.head(5)

headRows.foreach {

x => println("head(n)：" + x.get(0) + "==" + x.get(1) + "==" + x.get(2))

}

//获取前n行

**var** takeRows = userDataFrame.take(5)

takeRows.foreach {

x => println("take(n)：" + x.get(0) + "==" + x.get(1) + "==" + x.get(2))

}

//获取前n行，返回list

**var** takeListRows = userDataFrame.takeAsList(5)

**for**(i <- 0 until takeListRows.size()) {

println("takeList(n)：" + takeListRows.get(i).get(0) + "==" + takeListRows.get(i).get(1) + "==" + takeListRows.get(i).get(2))

}

sc.stop()

结果：

firt：10604==060813==杨峰

head：10604==060813==杨峰

head(n)：10604==060813==杨峰

head(n)：13127==2013040813==胡丹

head(n)：13173==2010081392==刘宗贤

head(n)：14743==2014030813==刘乐

head(n)：16411==2006081355==符佩瑜

take(n)：10604==060813==杨峰

take(n)：13127==2013040813==胡丹

take(n)：13173==2010081392==刘宗贤

take(n)：14743==2014030813==刘乐

take(n)：16411==2006081355==符佩瑜

takeList(n)：10604==060813==杨峰

takeList(n)：13127==2013040813==胡丹

takeList(n)：13173==2010081392==刘宗贤

takeList(n)：14743==2014030813==刘乐

takeList(n)：16411==2006081355==符佩瑜

#### 过滤条件

##### where(conditionExpr: String)：SQL语言中where关键字后的条件

**val** conf = **new** SparkConf().setAppName("DataFrame06").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

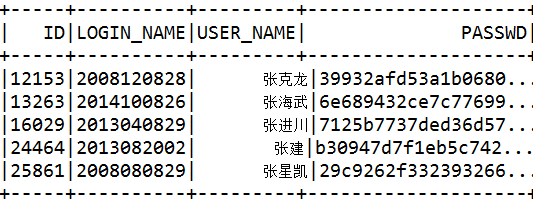
properties.setProperty("password", "jcbkzhzx110")

//获取数据，返回DataFrame类型的数据

**val** userDataFrame = sqlContext.read.jdbc(url, "user", properties)

.where(" login\_name like '%082%' and user\_name like '%张%' ").show()

sc.stop()



##### filter：根据字段进行筛选

与where使用方法相同。

#### 查询指定字段

##### select：获取指定字段值

**val** conf = **new** SparkConf().setAppName("DataFrame07").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

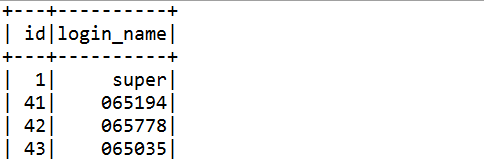
properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//获取数据，返回DataFrame类型的数据

**val** userDataFrame = sqlContext.read.jdbc(url, "user", properties).select("id", "login\_name").show()

sc.stop()



##### selectExpr：可以对指定字段进行特殊处理

可以直接对指定字段调用UDF函数，或者指定别名等。传入String类型参数，得到DataFrame对象。   
　　示例，查询id字段四舍五入，login\_name字段取别名name。

**val** conf = **new** SparkConf().setAppName("DataFrame08").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

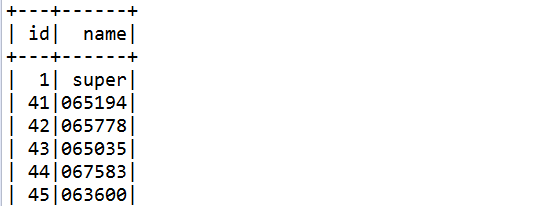
properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//获取数据，返回DataFrame类型的数据

**val** userDataFrame = sqlContext.read.jdbc(url, "user", properties).selectExpr("round(id) as id", "login\_name as name").show()

sc.stop()



##### col：获取指定字段

只能获取一个字段，返回对象为Column类型。

val idCol = jdbcDF.col(“id”)

##### apply：获取指定字段

只能获取一个字段，返回对象为Column类型。

val idCol1 = jdbcDF.apply("id")

val idCol2 = jdbcDF("id")

##### drop：去除指定字段，保留其他字段

返回一个新的DataFrame对象，其中不包含去除的字段，一次只能去除一个字段。

jdbcDF.drop("id")

jdbcDF.drop(jdbcDF("id"))

#### limit

limit方法获取指定DataFrame的前n行记录，得到一个新的DataFrame对象。和take与head不同的是，limit方法不是Action操作。

jdbcDF.limit(3).show(false)

#### 排序

##### orderBy、sort

orderBy和sort：按指定字段排序，默认为升序。

**val** conf = **new** SparkConf().setAppName("DataFrame09").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

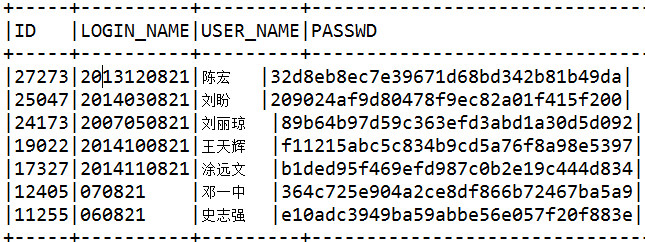
properties.setProperty("password", "jcbkzhzx110")

//获取数据，返回DataFrame类型的数据

**val** userDataFrame = sqlContext.read.jdbc(url, "user", properties).where(" login\_name like '%0821%'")

userDataFrame.orderBy(**userDataFrame("id").desc**).show(**false**)

sc.stop()



##### sortWithinPartitions

和sort方法功能类似，区别在于sortWithinPartitions方法返回的是按Partition排好序的DataFrame对象。

#### 分组

##### groupBy：根据字段进行group by操作

groupBy方法有两种调用方式，可以传入String类型的字段名，也可传入Column类型的对象。返回值是一个org.apache.spark.sql.GroupedData数据类型，在GroupedData的API中提供了group by之后的操作。操作如下：

* max(colNames: String\*)方法，获取分组中指定字段或者所有的数字类型字段的最大值，只能作用于数字型字段
* min(colNames: String\*)方法，获取分组中指定字段或者所有的数字类型字段的最小值，只能作用于数字型字段
* mean(colNames: String\*)方法，获取分组中指定字段或者所有的数字类型字段的平均值，只能作用于数字型字段
* sum(colNames: String\*)方法，获取分组中指定字段或者所有的数字类型字段的和值，只能作用于数字型字段
* count()方法，获取分组中的元素个数

**val** conf = **new** SparkConf().setAppName("DataFrame10").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

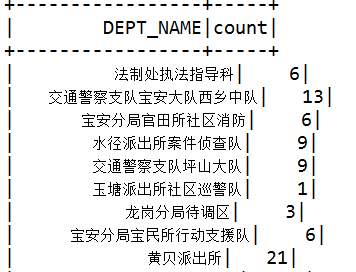
properties.setProperty("password", "jcbkzhzx110")

//获取数据，返回DataFrame类型的数据

**val** userDataFrame = sqlContext.read.jdbc(url, "user", properties)

.groupBy("DEPT\_NAME").count().show()

sc.stop()



#### 去重

##### distinct：返回一个不包含重复记录的DataFrame

返回当前DataFrame中不重复的Row记录。

**val** conf = **new** SparkConf().setAppName("DataFrame11").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//不去重统计，记录只有一个字段

**val** userCount1 = sqlContext.read.jdbc(url, "user", properties)

.select("user\_name").count()

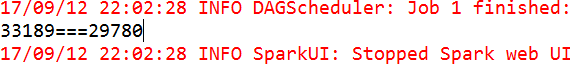
//去重统计，记录只有一个字段

**val** userCount2 = sqlContext.read.jdbc(url, "user", properties)

.select("user\_name").distinct().count()

println(userCount1 + "===" + userCount2)

sc.stop()



##### dropDuplicates：根据指定字段去重

根据指定字段去重。类似于select distinct a, b操作。如果不指定字段，效果与distinct一样。

**val** conf = **new** SparkConf().setAppName("DataFrame12").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//不去重统计

**val** userCount1 = sqlContext.read.jdbc(url, "user", properties)

.distinct().count()

//dropDuplicates不指定字段去重统计

**val** userCount2 = sqlContext.read.jdbc(url, "user", properties)

.dropDuplicates().count()

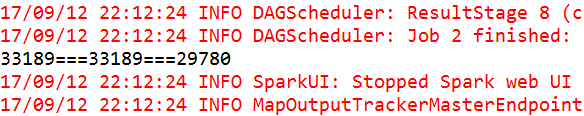
//dropDuplicates指定字段去重统计

**val** userCount3 = sqlContext.read.jdbc(url, "user", properties)

.dropDuplicates("user\_name").count()

println(userCount1 + "===" + userCount2 + "===" + userCount3)

sc.stop()



#### 聚合

聚合操作调用的是agg方法，该方法有多种调用方式。一般与groupBy方法配合使用（如分组统计等）。avg, max, min, sum, count。

**val** conf = **new** SparkConf().setAppName("DataFrame13").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

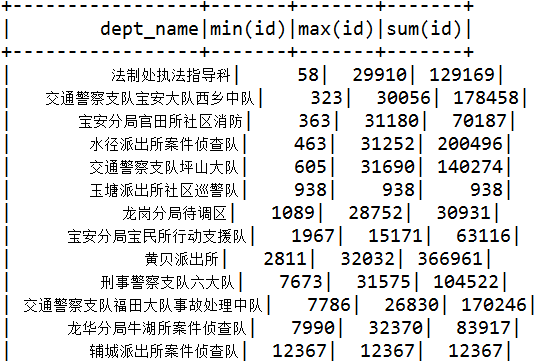
properties.setProperty("password", "jcbkzhzx110")

//分组取最小值、最大值、统计

**val** userCount1 = sqlContext.read.jdbc(url, "user", properties)

.groupBy("dept\_name").agg("id" -> "min", "id" -> "max", "id" -> "sum").show()

sc.stop()



#### union

union、unionAll方法：对两个DataFrame进行组合。

**val** conf = **new** SparkConf().setAppName("DataFrame14").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

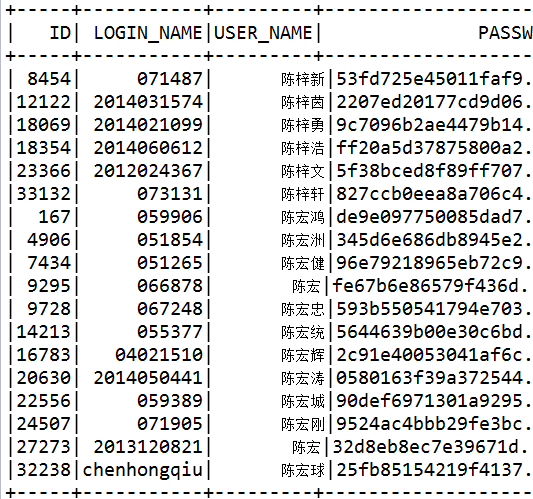
//两个DataFrame

**val** userDataFrame1 = sqlContext.read.jdbc(url, "user", properties).where("user\_name like '%陈梓%'")

**val** userDataFrame2 = sqlContext.read.jdbc(url, "user", properties).where("user\_name like '%陈宏%'")

userDataFrame1.union(userDataFrame2).show()

sc.stop()

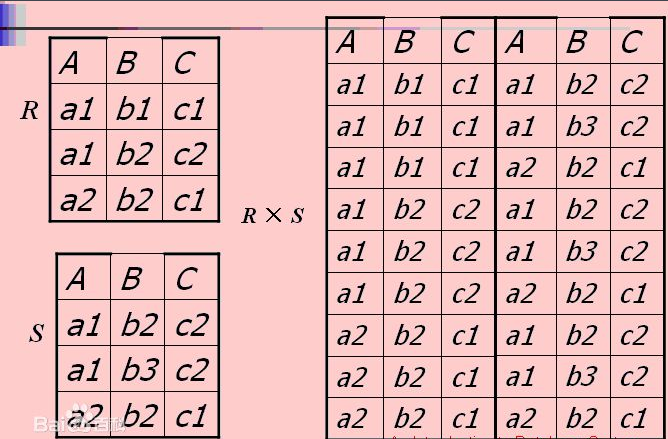


#### join

重点来了。在SQL语言中用得很多的就是join操作，DataFrame中同样也提供了join的功能。接下来隆重介绍join方法。在DataFrame中提供了六个重载的join方法。

##### 笛卡尔积

笛卡尔乘积是指在数学中，两个集合X和Y的笛卡尓积（Cartesian product），又称直积，表示为X×Y，第一个对象是X的成员而第二个对象是Y的所有可能有序对的其中一个成员。



示例：（X×Y条数据）未通过

**val** conf = **new** SparkConf().setAppName("DataFrame15").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//两个DataFrame

**val** userDataFrame1 = sqlContext.read.jdbc(url, "user", properties).selectExpr("LOGIN\_NAME", "DEPT\_ID as DEPT\_NO")

**val** deptDataFrame2 = sqlContext.read.jdbc(url, "department", properties).select("DEPT\_NO", "DEPT\_NAME")

userDataFrame1.join(deptDataFrame2).show()

sc.stop()

}

##### using一个字段形式

指定一个共同的字段，然后以该字段的值进行连接（两个DataFrame的该字段的值相等才能相关联），两个DataFrame中有相同的一个列名，结果中该字段只显示一次。（内部实现是内连接inner）

**val** conf = **new** SparkConf().setAppName("DataFrame16").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//两个DataFrame

**val** userDataFrame1 = sqlContext.read.jdbc(url, "user", properties).selectExpr("LOGIN\_NAME", "DEPT\_ID as DEPT\_NO")

**val** deptDataFrame2 = sqlContext.read.jdbc(url, "department", properties).select("DEPT\_NO", "DEPT\_NAME")

userDataFrame1.join(deptDataFrame2, "DEPT\_NO").show(100)

sc.stop()

##### using多字段形式

**val** conf = **new** SparkConf().setAppName("DataFrame17").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/dc"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//两个DataFrame

**val** userDataFrame1 = sqlContext.read.jdbc(url, "user", properties).selectExpr("LOGIN\_NAME", "DEPT\_ID as DEPT\_NO", "DEPT\_NAME")

**val** deptDataFrame2 = sqlContext.read.jdbc(url, "department", properties).select("DEPT\_NO", "DEPT\_NAME")

userDataFrame1.join(deptDataFrame2, Seq("DEPT\_NO", "DEPT\_NAME")).show(100)

sc.stop()

##### 指定join类型

两个DataFrame的join操作有inner（AB交集），outer（全连接），left\_outer，right\_outer，leftsemi类型。“full”,”outer”,”full\_outer”,”fullouter”代表全连接。

“left”,”left\_outer”或者”leftouter”代表左连接。

“right”,”right\_outer”及“rightouter”代表右连接。

“full”,”outer”,”full\_outer”,”fullouter”代表全连接。

**val** conf = **new** SparkConf().setAppName("DataFrame18").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/test"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//两个DataFrame

**val** aDataFrame1 = sqlContext.read.jdbc(url, "a", properties).select("id", "dept\_id", "name")

**val** bDataFrame2 = sqlContext.read.jdbc(url, "b", properties).selectExpr("id as dept\_id", "name as dept\_name")

aDataFrame1.join(bDataFrame2, Seq("dept\_id"), "left\_outer").show(100)

sc.stop()

##### 使用Column类型来join

**val** conf = **new** SparkConf().setAppName("DataFrame19").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/test"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//两个DataFrame

**val** aDataFrame1 = sqlContext.read.jdbc(url, "a", properties).select("id", "dept\_id", "name")

**val** bDataFrame2 = sqlContext.read.jdbc(url, "b", properties).selectExpr("id", "name as dept\_name")

aDataFrame1.join(bDataFrame2, **aDataFrame1("dept\_id") === bDataFrame2("id")**).show(100)

sc.stop()

* + - * 1. ***在指定join字段同时指定join类型***

**val** conf = **new** SparkConf().setAppName("DataFrame19").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/test"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//两个DataFrame

**val** aDataFrame1 = sqlContext.read.jdbc(url, "a", properties).select("id", "dept\_id", "name")

**val** bDataFrame2 = sqlContext.read.jdbc(url, "b", properties).selectExpr("id", "name as dept\_name")

aDataFrame1.join(bDataFrame2, aDataFrame1("dept\_id") === bDataFrame2("id"), "left\_outer").show(100)

sc.stop()

#### 获取指定字段统计信息

stat方法可以用于计算指定字段或指定字段之间的统计信息，比如方差，协方差等。这个方法返回一个DataFramesStatFunctions类型对象。

下面代码演示根据c4字段，统计该字段值出现频率在30%以上的内容。在jdbcDF中字段c1的内容为"a, b, a, c, d, b"。其中a和b出现的频率为2 / 6，大于0.3

jdbcDF.stat.freqItems(Seq ("c1") , 0.3).show()

#### 获取两个DataFrame中共有的记录

**val** conf = **new** SparkConf().setAppName("DataFrame20").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/test"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//两个DataFrame

**val** aDataFrame1 = sqlContext.read.jdbc(url, "a", properties).select("id")

**val** bDataFrame2 = sqlContext.read.jdbc(url, "b", properties).select("id")

aDataFrame1.intersect(bDataFrame2).show()

sc.stop()

#### 获取一个DataFrame中有另一个DataFrame中没有的记录

**val** conf = **new** SparkConf().setAppName("DataFrame21").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/test"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//两个DataFrame

**val** aDataFrame1 = sqlContext.read.jdbc(url, "a", properties).select("id")

**val** bDataFrame2 = sqlContext.read.jdbc(url, "b", properties).select("id")

aDataFrame1.except(bDataFrame2).show()

sc.stop()

#### 操作字段名

##### withColumnRenamed：重命名DataFrame中的指定字段名

如果指定的字段名不存在，不进行任何操作。

**val** conf = **new** SparkConf().setAppName("DataFrame21").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/test"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//DataFrame

**val** aDataFrame1 = sqlContext.read.jdbc(url, "a", properties).select("id", "name")

aDataFrame1.withColumnRenamed("id", "pk\_id").show()

sc.stop()



##### withColumn：往当前DataFrame中新增一列

**val** conf = **new** SparkConf().setAppName("DataFrame22").setMaster("local[2]")

**val** sc = **new** SparkContext(conf)

**val** sqlContext = **new** SQLContext(sc)

//返回DataFrame类型的数据

**var** url = "jdbc:mysql://localhost:3306/test"

**val** properties = **new** Properties()

properties.setProperty("user", "root")

properties.setProperty("password", "jcbkzhzx110")

//DataFrame

**val** aDataFrame1 = sqlContext.read.jdbc(url, "a", properties).select("id", "name")

aDataFrame1.withColumn("id2", aDataFrame1("id")).show()

sc.stop()



#### 行转列

有时候需要根据某个字段内容进行分割，然后生成多行，这时可以使用explode方法。

## SparkSession

在Spark的早期版本，sparkContext是进入Spark的切入点。我们都知道RDD是Spark中重要的API，然而它的创建和操作得使用sparkContext提供的API；对于RDD之外的其他东西，我们需要使用其他的Context。比如对于流处理来说，我们得使用StreamingContext；对于SQL得使用sqlContext；而对于hive得使用HiveContext。然而DataSet和Dataframe提供的API逐渐称为新的标准API，我们需要一个切入点来构建它们，所以在 Spark 2.0中我们引入了一个新的切入点(entry point)：SparkSession。

流处理：StreamingContext

RDD：sparkContext

DataSet和Dataframe：SparkSession

在 Spark 2.0中引入了一个新的切入点（entry point）：SparkSession。SparkSession实质上是SQLContext和HiveContext的组合（未来可能还会加上StreamingContext），所以在SQLContext和HiveContext上可用的API在SparkSession上同样是可以使用的。SparkSession内部封装了sparkContext，所以计算实际上是由sparkContext完成的。

### 创建SparkSession

SparkSession可以通过建造者模式创建。如果SparkContext存在，那么SparkSession将会重用它；但是如果SparkContext不存在，则创建它。

1. 创建一个SparkContext

**val** spark = SparkSession.builder()

.master("local[2]")

.appName("SparkSession01")

//.config("spark.some.config.option", "some-value")

.getOrCreate()

1. 创建一个HiveContext

**val** spark = SparkSession.builder()

.master("local[2]")

.appName("SparkSession01")

.enableHiveSupport()

//.config("spark.some.config.option", "some-value")

.getOrCreate()

# Spark案例

## 找出文本中行拥有最多的单词数是多少

**val** conf = **new** SparkConf().setAppName("textfile02").setMaster("local[4]")

**val** sc = **new** SparkContext(conf)

//读取一个

**val** textFile = sc.textFile("file:///E:/txt/file/a1.txt", 2)//加载一个文件

**val** wordCount = textFile.map(line => line.split(" ").size)

.reduce((a, b) => **if**(a > b) a **else** b)

print(wordCount)

sc.stop()

## 统计单词数

**val** conf = **new** SparkConf().setAppName("textfile02").setMaster("local[4]")

**val** sc = **new** SparkContext(conf)

//读取一个

**val** textFile = sc.textFile("file:///E:/txt/file/a1.txt", 2)//加载一个文件

**val** words = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a+b)

words.foreach(x => println(x.\_1 + "==" + x.\_2))

sc.stop()