



第一课 Spark ML基础入门

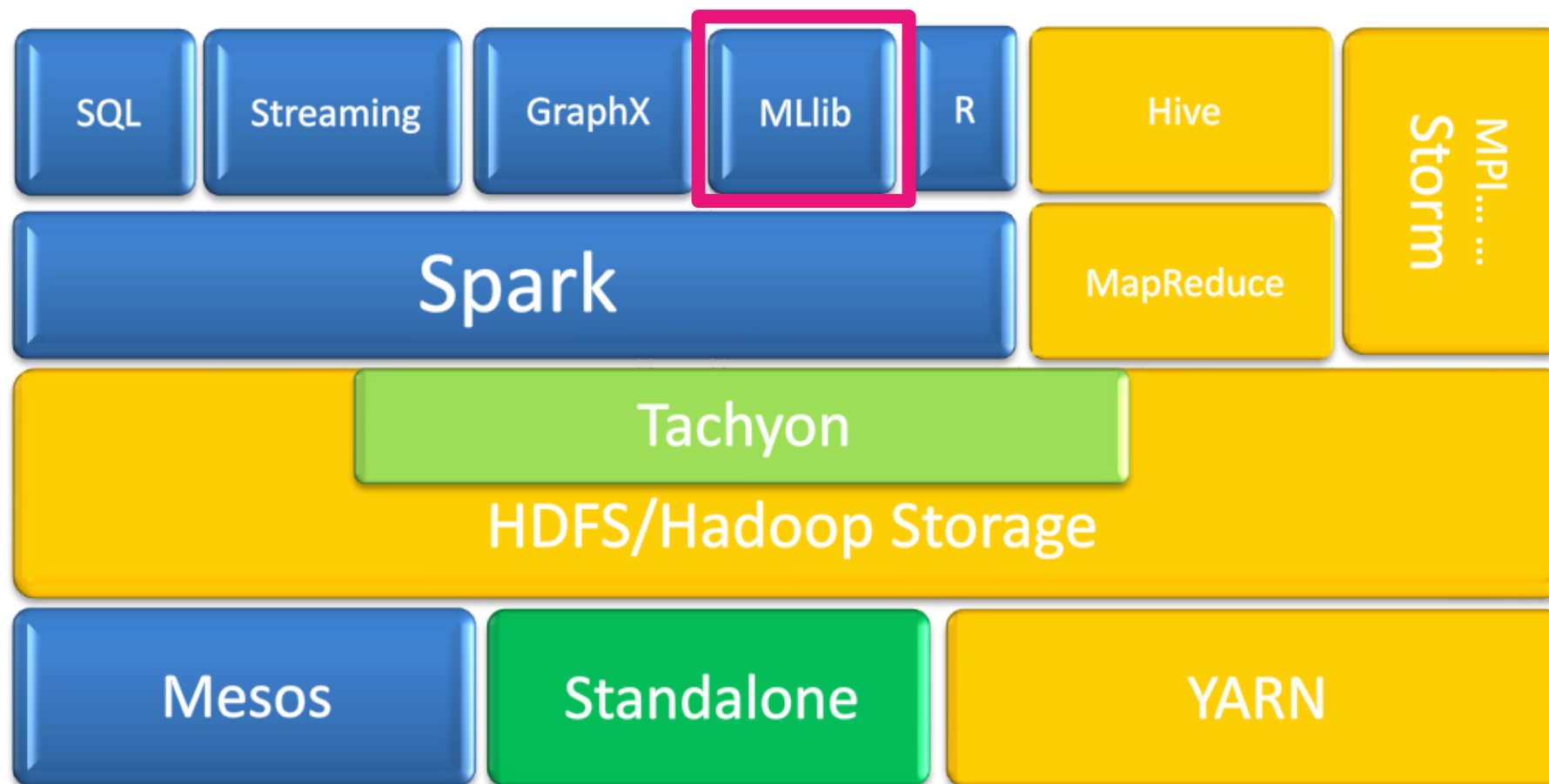
- 1、Spark介绍
- 2、Spark ML介绍
- 3、课程的基础环境
- 4、Spark SparkSession
- 5、Spark Datasets操作
- 6、Datasets操作的代码实操

个人介绍

- 黄美灵，网名：sunbow，Spark爱好者，现从事移动互联网的计算广告和数据变现工作。
- 《Spark MLlib机器学习：算法、源码及实战详解》作者
- CSDN博客专家
- <http://blog.csdn.net/sunbow0>

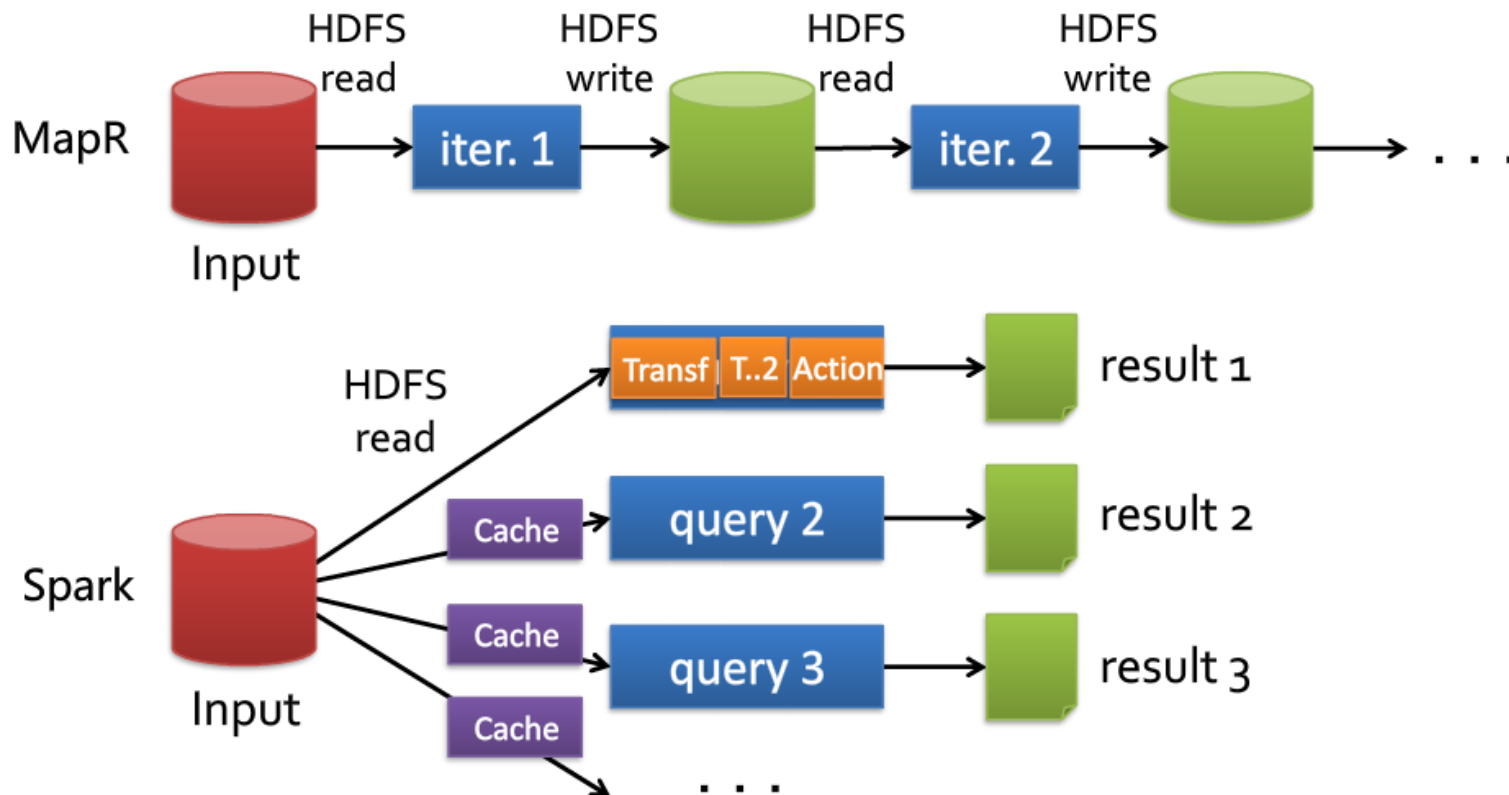


1、Spark概述



1、Spark概述

- MapReduce每次读写，都需要序列化到磁盘。一个复杂任务，需要多次处理，几十次磁盘读写。
- Spark只需要一次磁盘读写，大部分处理在内存中进行。



I/O and serialization can take **90%** of the time

1、Spark概述

■ spark-shell (交互窗口模式)

运行**Spark**-shell需要指向申请资源的standalone spark集群信息，其参数为MASTER，还可以指定executor及driver的内存大小。

■ `sudo spark-shell --executor-memory 2g --driver-memory 1g --executor-cores 2 --num-executors 4 --master spark://192.168.180.156:7077`

spark-shell启动完后，可以在交互窗口中输入**Scala**命令，进行操作，其中spark-shell已经默认生成spark对象，可以用：

```
val df1 = spark.read.csv("/user/t01.csv")
```

读取数据资源等。

2、Spark Mllib 介绍

MLlib: Main Guide

- Basic statistics
- Pipelines
- Extracting, transforming and selecting features
- Classification and Regression
- Clustering
- Collaborative filtering
- Frequent Pattern Mining
- Model selection and tuning
- Advanced topics

MLlib: RDD-based API Guide

- Data types
- Basic statistics
- Classification and regression
- Collaborative filtering
- Clustering
- Dimensionality reduction
- Feature extraction and transformation
- Frequent pattern mining
- Evaluation metrics
- PMML model export
- Optimization (developer)

Machine Learning Library (MLlib) Guide

MLlib is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:

- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- Persistence: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc.

Announcement: DataFrame-based API is primary API

The MLlib RDD-based API is now in maintenance mode.

As of Spark 2.0, the [RDD-based APIs](#) in the `spark.mllib` package have entered maintenance mode. The primary Machine Learning API for Spark is now the [DataFrame-based API](#) in the `spark.ml` package.

What are the implications?


- MLlib will still support the RDD-based API in `spark.mllib` with bug fixes.
- MLlib will not add new features to the RDD-based API.
- In the Spark 2.x releases, MLlib will add features to the DataFrames-based API to reach feature parity with the RDD-based API.
- After reaching feature parity (roughly estimated for Spark 2.3), the RDD-based API will be deprecated.
- The RDD-based API is expected to be removed in Spark 3.0.

Why is MLlib switching to the DataFrame-based API?

- DataFrames provide a more user-friendly API than RDDs. The many benefits of DataFrames include Spark Datasources, SQL/DataFrame queries, Tungsten and Catalyst optimizations, and uniform APIs across languages.
- The DataFrame-based API for MLlib provides a uniform API across ML algorithms and across multiple languages.
- DataFrames facilitate practical ML Pipelines, particularly feature transformations. See the [Pipelines guide](#) for details.

What is "Spark ML"?

- Spark2.2、Spark2.1、Spark2.0
- `sudo spark-shell --executor-memory 2g --driver-memory 1g --total-executor-cores 2 --num-executors 1 --master spark://192.168.180.100:7077`

 **Spark Master at spark://192.168.180.156:7077**

URL: spark://192.168.180.156:7077
REST URL: spark://192.168.180.156:6066 (cluster mode)
Alive Workers: 1
Cores in use: 2 Total, 2 Used
Memory in use: 2.0 GB Total, 2.0 GB Used
Applications: 1 Running, 8 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160412143006-192.168.180.156-42236	192.168.180.156:42236	ALIVE	2 (2 Used)	2.0 GB (2.0 GB Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160506184703-0008 (kill)	Spark shell	2	2.0 GB	2016/05/06 18:47:03	root	RUNNING	1.2 min

1、DataSet介绍

1.1 DataSet演进历史



Spark 0.0
第一代

Spark 1.3
第二代

Spark 1.6
第三代

弹性分布式数据集

列方式组织的分布式数据集

Encoder的分布式数据集

	<table><tr><th>Name</th><th>Age</th><th>Heigth</th></tr><tr><td>String</td><td>Int</td><td>Double</td></tr><tr><td>String</td><td>Int</td><td>Double</td></tr><tr><td>String</td><td>Int</td><td>Double</td></tr></table>	Name	Age	Heigth	String	Int	Double	String	Int	Double	String	Int	Double	<table><tr><th>Name</th><th>Age</th><th>Heigth</th></tr><tr><td>Encoders.STRING()</td><td>Encoders.INT()</td><td>Encoders.DOUBLE()</td></tr><tr><td>Encoders.STRING()</td><td>Encoders.INT()</td><td>Encoders.DOUBLE()</td></tr><tr><td>Encoders.STRING()</td><td>Encoders.INT()</td><td>Encoders.DOUBLE()</td></tr></table>	Name	Age	Heigth	Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()	Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()	Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()
Name	Age	Heigth																								
String	Int	Double																								
String	Int	Double																								
String	Int	Double																								
Name	Age	Heigth																								
Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()																								
Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()																								
Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()																								
<table><tr><td>Person</td></tr><tr><td>Person</td></tr><tr><td>Person</td></tr></table>	Person	Person	Person	<table><tr><td>String</td><td>Int</td><td>Double</td></tr><tr><td>String</td><td>Int</td><td>Double</td></tr><tr><td>String</td><td>Int</td><td>Double</td></tr></table>	String	Int	Double	String	Int	Double	String	Int	Double	<table><tr><td>Encoders.STRING()</td><td>Encoders.INT()</td><td>Encoders.DOUBLE()</td></tr><tr><td>Encoders.STRING()</td><td>Encoders.INT()</td><td>Encoders.DOUBLE()</td></tr><tr><td>Encoders.STRING()</td><td>Encoders.INT()</td><td>Encoders.DOUBLE()</td></tr></table>	Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()	Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()	Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()			
Person																										
Person																										
Person																										
String	Int	Double																								
String	Int	Double																								
String	Int	Double																								
Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()																								
Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()																								
Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()																								
<table><tr><td>Person</td></tr><tr><td>Person</td></tr><tr><td>Person</td></tr></table>	Person	Person	Person	<table><tr><td>String</td><td>Int</td><td>Double</td></tr><tr><td>String</td><td>Int</td><td>Double</td></tr><tr><td>String</td><td>Int</td><td>Double</td></tr></table>	String	Int	Double	String	Int	Double	String	Int	Double	<table><tr><td>Encoders.STRING()</td><td>Encoders.INT()</td><td>Encoders.DOUBLE()</td></tr><tr><td>Encoders.STRING()</td><td>Encoders.INT()</td><td>Encoders.DOUBLE()</td></tr><tr><td>Encoders.STRING()</td><td>Encoders.INT()</td><td>Encoders.DOUBLE()</td></tr></table>	Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()	Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()	Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()			
Person																										
Person																										
Person																										
String	Int	Double																								
String	Int	Double																								
String	Int	Double																								
Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()																								
Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()																								
Encoders.STRING()	Encoders.INT()	Encoders.DOUBLE()																								

RDD[Person]
非结构化数据

DataFrame
结构化数据

Dataset
已序列化的结构数据

DATAGURU专业数据分析社区

1.1 DataSet演进历史

1、Spark第一代API : RDD

优点:

- 1) 编译时类型安全 , 编译时就能检查出类型错误。
- 2) 面向对象的编程风格 , 直接通过类名点的方式来操作数据。

`idAge.filter(_.age > "")` // 编译时报错, int不能跟String比

`idAgeRDDPerson.filter(_.age > 25)` // 直接操作一个个的person对象

缺点:

- 1) 序列化和反序列化的性能开销 , 无论是集群间的通信, 还是IO操作都需要对对象的结构和数据进行序列化和反序列化。
- 2) GC的性能开销 , 频繁的创建和销毁对象, 势必会增加GC。

1.1 DataSet演进历史

Spark第二代API : DataFrame

DataFrame核心特征：

Schema：包含了以ROW为单位的每行数据的列的信息；Spark通过Schema就能够读懂数据，因此在通信和IO时就只需要序列化和反序列化数据，而结构的部分就可以省略了。

off-heap：Spark能够以二进制的形式序列化数据(不包括结构)到off-heap中，当要操作数据时，就直接操作off-heap内存。

Tungsten：新的执行引擎；

Catalyst：新的语法解析框架；

1.1 DataSet演进历史

Spark第二代API : DataFrame

优点 :

off-heap就像地盘, schema就像地图, Spark有地图又有自己地盘了, 就可以自己说了算, 不再受JVM的限制, 也就不再收GC的困扰了, 通过schema和off-heap, DataFrame解决了RDD的缺点。对比RDD提升计算效率、减少数据读取、底层计算优化;

缺点:

DataFrame解决了RDD的缺点, 但是却丢了RDD的优点。DataFrame不是类型安全的, API也不是面向对象风格的。

```
// API不是面向对象的
```

```
idAgeDF.filter(idAgeDF.col("age") > 25)
```

```
// 不会报错, DataFrame不是编译时类型安全的
```

```
idAgeDF.filter(idAgeDF.col("age") > "")
```

1.1 DataSet演进历史

Spark第三代API : DataSet

DataSet的核心：Encoder。

- 1) 编译时的类型安全检查；性能极大的提升，内存使用极大降低、减少GC、极大的减少网络数据的传输、极大的减少采用scala和java变成代码的差异性。
- 2) DataFrame每一个行对应了一个Row。而Dataset的定义更加宽松，每一个record对应了一个任意的类型。DataFrame只是Dataset的一种特例。
- 3) 不同于Row是一个泛化的无类型JVM object, **Dataset是由一系列的强类型JVM object组成的，Scala的case class或者Java class定义。因此DataSet可以在编译时进行类型检查。**
- 4) DataSet以Catalyst逻辑执行计划表示，并且数据以编码的二进制形式被存储，不需要反序列化就可以执行sorting、shuffle等操作。
- 5) **DataSet创立需要一个显式的Encoder，把对象序列化为二进制。**

1.2 DataSet对MLlib的影响

1) 基于MLlib RDD的API现在处于维护模式。

从Spark 2.0开始，spark.mllib软件包中的基于RDD的API已进入维护模式。Spark的主要学习API现在是spark.ml包中基于DataFrame的API。

MLlib仍然会在spark.mllib中支持基于RDD的API，并提供错误修复。

MLlib不会为基于RDD的API添加新功能。

预计将在Spark 3.0中删除基于RDD的API。

1.2 DataSet对MLlib的影响

2) 基于MLlib DataFrames的API。

在Spark 2.x版本中，MLlib将向基于DataFrames的API添加功能，以实现与基于RDD的API的功能奇偶校验。达到功能奇偶校验（Spark 2.2）后，将不推荐使用基于RDD的API。

3) MLlib的发展

MLlib中基于DataFrame的API将成为主流，MLlib的API更加偏向于底层，可以灵活多变的修改逻辑，MLlib的API不会被ML替代。

1.3 Spark Sql API介绍

Spark Sql 核心API		
org.apache.spark.sql	hide	focus
<ul style="list-style-type: none">AnalysisExceptionColumnColumnNameDataFrameNaFunctionsDataFrameReaderDataFrameStatFunctionsDataFrameWriterDatasetDatasetHolderEncoderEncodersExperimentalMethodsForeachWriterfunctionsKeyValueGroupedDatasetLowPrioritySQLImplicitsRelationalGroupedDatasetRowRowFactoryRuntimeConfigSaveModeSparkSessionSparkSessionExtensionsSQLContextSQLImplicitsTypedColumnUDFRegistration	<ul style="list-style-type: none">sparkSessionDatasetColumnRowEncoderfunctionsSQLImplicits	<p>Spark入口</p> <p>SparkSession统一封装了SparkConf, SparkContext, SQLContext; SparkSession是spark的唯一切入点, 通过SparkSession可以配置Spark运行参数、读取文件、创建数据、使用SQL等。</p> <p>Dataset</p> <p>统一Dataset接口, 其中DataFrame为Dataset[Row]特例; 其中Dataset基本实现了类似rdd的所有操作算子。</p> <p>Dataset的列对象</p> <p>包括对列的基本操作函数。</p> <p>DataFrame的行对象</p> <p>包括对行的基本操作函数。</p> <p>序列化</p> <p>支持常用的数据类型, 可以直接序列化; 也支持采用case class 自定义数据对象进行序列化</p> <p>Dataset内置函数</p> <p>支持丰富的操作函数, 包括: 聚合函数、Collection函数、日期时间函数、数值计算函数、Misc函数、Non-aggregate函数、排序函数、字符串函数、Window函数、自定义UDF函数。</p> <p>隐式转换</p> <p>其中scala对象、rdd转成DF/DS, DF/DS使用Map/Flatmap方法等, 需要采用隐式转换。引入格式:</p> <pre>val spark = SparkSession().... import spark.implicits._</pre>

1.3 Spark Sql API介绍

- **SparkSession** : Spark的一个全新的切入点，统一Spark入口
- [Spark2.0](#)中引入了SparkSession的概念，它为用户提供了一个统一的切入点来使用Spark的各项功能，包括是SQLContext和HiveContext的组合（未来可能还会加上StreamingContext），用户不但可以使用DataFrame和Dataset的各种API，学习Spark的难度也会大大降低。

```
//0 构建Spark对象
val spark = SparkSession
    .builder
    .appName("test")
    .enableHiveSupport()
    .getOrCreate()
```

2、DataSet基本操作

2.1 DataSet的创建

1、DataSet的创建操作			
操作名称	操作函数	用法	说明
创建Dataset	spark.createDataset	createDataset[T](data: Seq[T])(implicit arg0: Encoder[T]): Dataset[T]	通过Seq集合生成Dataset，其中数据类型已知，如果有多列，采用case class指定格式
		createDataset[T](data: List[T])(implicit arg0: Encoder[T]): Dataset[T]	通过List集合生成Dataset，其中数据类型已知，如果有多列，采用case class指定格式
		createDataset[T](data: RDD[T])(implicit arg0: Encoder[T]): Dataset[T]	通过RDD转成Dataset，其中数据类型已知，如果有多列，采用case class指定格式
		rdd.toDS	通过RDD转成Dataset
	spark.range	range(start: Long, end: Long, step: Long): Dataset[Long]	产生序列dataset，指定起始值、序列间隔
		range(start: Long, end: Long, step: Long, numPartitions: Int): Dataset[Long]	产生序列dataset，指定起始值、序列间隔，并指定分区数量
创建DataFrame	spark.createDataFrame	createDataFrame(rows: List[Row], schema: StructType): DataFrame	通过List集合生成DataFrame
		createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame	通过RDD转成DataFrame
		rdd.toDF	通过RDD转成DataFrame

2.1 DataSet的创建

读取文件	spark.read	spark.read.csv(path: String)	读取csv文件
		spark.read.options(options: Map[String, String]).chema(schema).csv(path: String)	读取csv文件，可以设置读取的参数，例如：分隔符，表头等，以及设置数据格式。
		spark.read.text(path: String)	读取txt文件
		spark.read.json(path: String)	读取json文件
读取HIVE表数据	spark.sql	spark.sql(sqlText: String): DataFrame	读取hive表数据

2.1 DataSet的创建

```
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.Encoder
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import org.apache.spark.sql._

case class Person(name: String, age: Int, height: Int)
case class Peples(age: Int, names: String)

import spark.implicits._
spark.sparkContext.setCheckpointDir("hdfs://[redacted]/spark_checkpoint")
```

```
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.Encoder
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import org.apache.spark.sql._
defined class Person
defined class Peples
import spark.implicits._
```

2.1 DataSet的创建

```
//1 产生序列dataset  
val numDS = spark.range(5, 100, 5)  
numDS.orderBy(desc("id")).show(5)  
numDS.describe().show()
```

```
numDS: org.apache.spark.sql.Dataset[Long] = [id: bigint]
```

```
+---+
```

```
| id|
```

```
+---+
```

```
| 95|
```

```
| 90|
```

```
| 85|
```

```
| 80|
```

```
| 75|
```

```
+---+
```

```
only showing top 5 rows
```

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

```
| summary|          id|
```

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

```
| count|          19|
```

```
| mean|          50.0|
```

2.1 DataSet的创建

```
//2 集合转成Dataset
```

```
val seq1 = Seq(Person("Michael", 29, 170), Person("Andy", 30, 165), Person("geta", 29, 170))  
val ds1 = spark.createDataset(seq1)  
ds1.show()
```

```
seq1: Seq[Person] = List(Person(Michael,29,170), Person(Andy,30,165), Person(geta,29,170))
```

```
ds1: org.apache.spark.sql.Dataset[Person] = [name: string, age: int ... 1 more field]
```

```
+-----+-----+  
|  name|age|height|  
+-----+-----+  
|Michael| 29|   170|  
|   Andy| 30|   165|  
|   geta| 29|   170|  
+-----+-----+
```

Took 1 sec. Last updated by sunbowhuang at July 16 2017, 2:42:43 PM.

2.1 DataSet的创建

```
//3 集合转成DataFrame
val df1 = spark.createDataFrame(seq1).withColumnRenamed("_1", "language").withColumnRenamed("_2", "percent")
ds1.orderBy(desc("age")).show(10)
```

```
df1: org.apache.spark.sql.DataFrame = [name: string, age: int ... 1 more field]
```

```
+-----+-----+
|  name|age|height|
+-----+-----+
|  Andy| 30|   165|
|Michael| 29|   170|
|  geta| 29|   170|
+-----+-----+
```

Took 5 sec. Last updated by sunbowhuang at July 16 2017, 2:42:57 PM.

2.1 DataSet的创建

```
//4 rdd转成DataFrame
val array1 = Array(("Michael", 29, 170), ("Andy", 30, 165), ("geta", 29, 170))
val rdd1 = spark.sparkContext.parallelize(array1, 3).map(f => Row(f._1, f._2, f._3))
val schema = StructType(
  StructField("name", StringType, false) ::
  StructField("age", IntegerType, true) :: Nil)
val rddToDataFrame = spark.createDataFrame(rdd1, schema)
rddToDataFrame.orderBy(desc("name")).show(false)
```

```
array1: Array[(String, Int, Int)] = Array((Michael,29,170), (Andy,30,165), (geta,29,170))
rdd1: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[17] at map at <cons
schema: org.apache.spark.sql.types.StructType = StructType(StructField(name,StringType,false),
rddToDataFrame: org.apache.spark.sql.DataFrame = [name: string, age: int]
```

```
+-----+-----+
|name    |age|
+-----+-----+
|geta    |29 |
|Michael |29 |
|Andy    |30 |
+-----+-----+
```

2.1 DataSet的创建

```
//5 rdd转成Dataset/DataFrame
val rdd2 = spark.sparkContext.parallelize(array1, 3).map(f => Person(f._1, f._2, f._3))
val ds2 = rdd2.toDS()
val df2 = rdd2.toDF()
ds2.orderBy(desc("name")).show(10)
df2.orderBy(desc("name")).show(10)
```

```
rdd2: org.apache.spark.rdd.RDD[Person] = MapPartitionsRDD[23] at map at <console>:44
ds2: org.apache.spark.sql.Dataset[Person] = [name: string, age: int ... 1 more field]
df2: org.apache.spark.sql.DataFrame = [name: string, age: int ... 1 more field]
```

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

```
|  name|age|height|
```

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

```
|  geta| 29|   170|
```

```
|Michael| 29|   170|
```

```
|  Andy| 30|   165|
```

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

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

```
|  name|age|height|
```

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

```
|  geta| 29|   170|
```

```
|Michael| 29|   170|
```

```
|  Andy| 30|   165|
```

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

2.1 DataSet的创建

```
//6 rdd转成Dataset  
val ds3 = spark.createDataset(rdd2)  
ds3.show(10)
```

```
ds3: org.apache.spark.sql.Dataset[Person] = [name: string, age: int ... 1 more field]  
+-----+---+-----+  
|   name|age|height|  
+-----+---+-----+  
|Michael| 29|   170|  
|   Andy| 30|   165|  
|   geta| 29|   170|  
+-----+---+-----+
```


2.1 DataSet的创建

```
//7 读取文件
val df4 = spark.read.csv("hdfs://10.10.10.100:8020/t01.csv")
df4.show()
```

```
df4: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 1 more field]
```

```
+-----+-----+
|  _c0 | _c1 | _c2 |
+-----+-----+
|Michael| 29 |170|
|  Andy| 30 |165|
|  geta| 29 |170|
| Justin| 29 |170|
|   Big| 25 |175|
+-----+-----+
```

2.1 DataSet的创建

```
//8 读取文件 , 详细参数
val schema2 = StructType(
  StructField("name", StringType, false) ::
  StructField("age", IntegerType, false) ::
  StructField("height", IntegerType, true) :: Nil)
val df7 = spark.read.
  options(Map(("delimiter", ","), ("header", "false"))).
  schema(schema2).
  csv("hdfs://[REDACTED]/t01.csv").show()
```

```
schema2: org.apache.spark.sql.types.StructType = StructType(StructField(name,StringType,true))
```

```
+-----+-----+
|  name|age|height|
+-----+-----+
|Michael| 29|   170|
|  Andy| 30|   165|
|  geta| 29|   170|
| Justin| 29|   170|
|   Big| 25|   175|
+-----+-----+
```

```
df7: Unit = ()
```

2.2 DataSet的基础函数

2、DataSet的基础函数

操作名称	操作函数	用法	说明
存储类型	checkpoint	checkpoint(): Dataset[T]	为当前Dataset设置检查点，数据会存储到checkpoint目录中，其中目录需要在sparkContext中设置。
		checkpoint(eager: Boolean): Dataset[T]	
	cache	cache(): Dataset.this.type	将数据集缓存到内存。
	persist	persist(): Dataset.this.type	设置数据集的存储级别。默认：MEMORY_AND_DISK
		persist(newLevel: StorageLevel): Dataset.this.type	设置数据集的存储级别。支持：MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, MEMORY_AND_DISK_SER, DISK_ONLY, MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.
	unpersist	unpersist(): Dataset.this.type	将Dataset删除。
		unpersist(blocking: Boolean): Dataset.this.type	将Dataset删除。

2.2 DataSet的基础函数

结构属性	columns	columns: Array[String]	以数组的形式返回数据集的列名。
	dtypes	dtypes: Array[(String, String)]	以数组的形式返回Dataset的列名和数据类型。
	explain	explain(): Unit	返回Dataset的执行物理计划。
		explain(extended: Boolean): Unit	返回Dataset的执行计划（逻辑和物理）。
rdd数据互转	rdd	rdd: RDD[T]	DataFrame转成rdd
	toDF	toDF(): DataFrame	将rdd转成DataFrame
		toDF(colNames: String*): DataFrame	
保存文件	write	write: DataFrameWriter[T]	Dataset保存到文件中
	writeStream	writeStream: DataStreamWriter[T]	Dataset流式数据保存到文件中

2.2 DataSet的基础函数

DS转临时sql表	createTempView	createTempView(viewName: String): Unit	将DS转换成临时表，可以用sql进行操作，生命周期为当前SparkSession。其中Replace可以覆盖已有临时表。
		createOrReplaceTempView(viewName: String): Unit	
		createOrReplaceGlobalTempView(viewName: String): Unit	将DS转换成临时表，可以用sql进行操作，生命周期为当前Spark作业。其中Replace可以覆盖已有临时表。
		createGlobalTempView(viewName: String): Unit	

2.2 DataSet的基础函数

```
//1 Dataset存储类型
val seq1 = Seq(Person("Michael", 29, 170), Person("Andy", 30, 165), Person("geta", 29, 170))
val ds1 = spark.createDataset(seq1)
ds1.show()
ds1.checkpoint()
ds1.cache()
ds1.persist(MEMORY_ONLY)
ds1.count()
ds1.show()
ds1.unpersist(true)

//2 Dataset结构属性
ds1.columns
ds1.dtypes
ds1.explain()

//3 Dataset rdd数据互转
val rdd1 = ds1.rdd
val ds2 = rdd1.toDS()
ds2.show()
val df2 = rdd1.toDF()
df2.show()
```

2.2 DataSet的基础函数

```
//4 Dataset 保存文件
ds1.select("name", "age", "height").write.format("csv").save("hdfs://192.168.1.100:8020/peple01.csv")
// 读取保存的文件
val schema2 = StructType(
  StructField("name", StringType, false) ::
  StructField("age", IntegerType, false) ::
  StructField("height", IntegerType, true) :: Nil)
val out = spark.read.
  options(Map(("delimiter", ","), ("header", "false"))).
  schema(schema2).
  csv("hdfs://192.168.1.100:8020/peple01.csv")
out.show(10)
```


2.2 DataSet的基础函数

//4 结果保存

```
user_recommend.createOrReplaceTempView("df_to_hive_table")
val insertSql1 = "insert overwrite table " + recommend_table +
" partition(ds=" + sampleDate + ", source='" + source2 + "') select * from df_to_hive_table"
println(insertSql1)
spark.sql(insertSql1)
```

2.3 DataSet的Actions操作

3、DataSet的Actions操作			
操作名称	操作函数	用法	说明
显示数据集： 以表格形式显示数据集	show	show(): Unit	默认显示前20行数据
		show(truncate: Boolean): Unit	truncate是否对超过20个字符的字符串进行截断
		show(numRows: Int, truncate: Boolean): Unit	numRows可以指定显示的行数，truncate为TRUE表示截断。
		show(numRows: Int, truncate: Int): Unit	numRows可以指定显示的行数，truncate大于0表示截断。
获取数据集	collect	collect(): Array[T]	以数组形式返回数据集
		collectAsList(): List[T]	以Java list形式返回数据集
	first	first(): T	返回数据集中的第1行数据，类似于take(1)
	head	head(): T	返回数据集中的第1行数据，类似于take(1)
		head(n: Int): Array[T]	返回数据集中的前n行数据，类似于take(n)
	take	take(n: Int): Array[T]	返回数据集中的前n行数据
		takeAsList(n: Int): List[T]	list形式返回数据集前n行数据

2.3 DataSet的Actions操作

统计数据集	count	<code>count(): Long</code>	返回数据集的行数。
	describe	<code>describe(cols: String*): DataFrame</code>	计算数据集中指定列的统计信息，统计指标包括：count，mean，stddev，min，max。
聚集	reduce	<code>reduce(func: (T, T) => T): T</code>	对数据集的每一行执行聚集(func)函数，该函数必须是可交换的
		<code>reduce(func: ReduceFunction[T]): T</code>	可以用自己定义的聚集(func)函数

2.3 DataSet的Actions操作

```
//1 显示数据集
```

```
val seq1 = Seq(Person("Michael", 29, 170), Person("Andy", 30, 165), Person("geta", 29, 170), Person("agg", 18, 160))
```

```
val ds1 = spark.createDataset(seq1)
```

```
ds1.show()
```

```
ds1.show(2)
```

```
ds1.show(2, true)
```

```
seq1: Seq[Person] = List(Person(Michael,29,170), Person(Andy,30,165), Person(geta,29,170), Person(agg,18,160))
```

```
ds1: org.apache.spark.sql.Dataset[Person] = [name: string, age: int ... 1 more field]
```

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

```
|  name|age|height|
```

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

```
|Michael| 29|  170|
```

```
|  Andy| 30|  165|
```

```
|  geta| 29|  170|
```

```
|   agg| 18|  160|
```

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

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

```
|  name|age|height|
```

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

```
|Michael| 29|  170|
```

```
|  Andy| 30|  165|
```

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

```
only showing top 2 rows
```

2.3 DataSet的Actions操作

```
//2 获取数据集
val c1 = ds1.collect()
val c2 = ds1.collectAsList()
val h1 = ds1.head()
val h2 = ds1.head(3)
val f1 = ds1.first()
val t1 = ds1.take(2)
val t2 = ds1.takeAsList(2)
```

```
c1: Array[Person] = Array(Person(Michael,29,170), Person(Andy,30,165), Person(geta,29,170), Person(agg,18,160))
c2: java.util.List[Person] = [Person(Michael,29,170), Person(Andy,30,165), Person(geta,29,170), Person(agg,18,160)]
h1: Person = Person(Michael,29,170)
h2: Array[Person] = Array(Person(Michael,29,170), Person(Andy,30,165), Person(geta,29,170))
f1: Person = Person(Michael,29,170)
t1: Array[Person] = Array(Person(Michael,29,170), Person(Andy,30,165))
t2: java.util.List[Person] = [Person(Michael,29,170), Person(Andy,30,165)]
```

2.3 DataSet的Actions操作

```
//3 统计数据集
ds1.count()
ds1.describe().show()
ds1.describe("age").show()
ds1.describe("age","height").show()
```

```
res7: Long = 4
```

```
+-----+-----+-----+-----+
|summary|name|          age|          height|
+-----+-----+-----+-----+
|  count|   4|           4|           4|
|   mean|null|        26.5|        166.25|
| stddev|null|5.686240703077327|4.787135538781694|
|   min|Andy|          18|          160|
|   max|geta|          30|          170|
+-----+-----+-----+-----+
```

```
+-----+-----+
|summary|          age|
+-----+-----+
|  count|           4|
|   mean|        26.5|
| stddev|5.686240703077327|
|   min|          18|
```

2.3 DataSet的Actions操作

//4 聚集

```
ds1.reduce((f1, f2) => Person("sum", (f1.age + f2.age), (f1.height + f2.height)))
```

```
scala> ds1.reduce((f1, f2) => Person("sum", (f1.age + f2.age), (f1.height + f2.height)))  
res111: Person = Person(sum,106,665)
```

2.4 DataSet的类型化的转化操作

4、DataSet的转换操作			
操作名称	操作函数	用法	说明
别名操作	as	as(alias: Symbol): Dataset[T]	返回一个具有别名的新数据集。
		as(alias: String): Dataset[T]	
		alias(alias: String): Dataset[T]	
		alias(alias: Symbol): Dataset[T]	
map操作	map	map[U](func: (T) => U)(implicit arg0: Encoder[U]): Dataset[U]	返回一个新的数据集，其中对每一行进行func操作。
		map[U](func: MapFunction[T, U], encoder: Encoder[U]): Dataset[U]	
	mapPartitions	mapPartitions[U](f: MapPartitionsFunction[T, U], encoder: Encoder[U]): Dataset[U]	返回一个新的数据集，其中对每一个分区进行func操作。
		mapPartitions[U](func: (Iterator[T]) => Iterator[U])(implicit arg0: Encoder[U]): Dataset[U]	
	flatMap	flatMap[U](func: (T) => TraversableOnce[U])(implicit arg0: Encoder[U]): Dataset[U]	返回一个新的数据集，其中对每一个分行进行func操作，每一行会被映射为多行。
		flatMap[U](f: FlatMapFunction[T, U], encoder: Encoder[U]): Dataset[U]	

2.4 DataSet的类型化的转化操作

过滤操作	filter	<code>filter(condition: Column): Dataset[T]</code>	过滤操作，按照列的逻辑过滤
		<code>filter(conditionExpr: String): Dataset[T]</code>	过滤操作，按照sql的逻辑过滤
		<code>filter(func: (T) ⇒ Boolean): Dataset[T]</code>	过滤操作，按照元素的逻辑过滤
		<code>filter(func: FilterFunction[T]): Dataset[T]</code>	过滤操作，按照自定义函数的逻辑过滤
	where	<code>where(condition: Column): Dataset[T]</code>	过滤操作，按照列的逻辑过滤
		<code>where(conditionExpr: String): Dataset[T]</code>	过滤操作，按照sql的逻辑过滤
去重操作	distinct	<code>distinct(): Dataset[T]</code>	对相同行数据去除重复数据
	dropDuplicates	<code>dropDuplicates(): Dataset[T]</code>	删除重复行数据，可以指定列进行删除重复数据。
		<code>dropDuplicates(colNames: Seq[String]): Dataset[T]</code>	
		<code>dropDuplicates(colNames: Array[String]): Dataset[T]</code>	
加法/减法操作	except	<code>except(other: Dataset[T]): Dataset[T]</code>	对两个Dataset进行减法操作
	union	<code>union(other: Dataset[T]): Dataset[T]</code>	对两个Dataset进行合并操作
	intersect	<code>intersect(other: Dataset[T]): Dataset[T]</code>	对两个Dataset进行交集操作

2.4 DataSet的类型化的转化操作

Select操作	select	select[U1](c1: TypedColumn[T, U1]): Dataset[U1]	对Dataset选择列，其中可以对列进行自带udf/自定义udf函数操作
排序操作	sort	sort(sortCol: String, sortCols: String*): Dataset[T]	对Dataset进行全局排序，支持多列，可以按照升序、降序
		sort(sortExprs: Column*): Dataset[T]	
	sortWithinPartitions	sortWithinPartitions(sortCol: String, sortCols: String*): Dataset[T]	对Dataset进行每个分区内排序，支持多列，可以按照升序、降序
		sortWithinPartitions(sortExprs: Column*): Dataset[T]	
	orderBy	orderBy(sortCol: String, sortCols: String*): Dataset[T]	对Dataset进行全局排序，支持多列，可以按照升序、降序
		orderBy(sortExprs: Column*): Dataset[T]	
抽样/分割操作	randomSplit	randomSplit(weights: Array[Double]): Array[Dataset[T]]	按照权重进行数据分割，第1个参数为分割权重数组，第2个参数为随机种子
		randomSplit(weights: Array[Double], seed: Long): Array[Dataset[T]]	
	randomSplitAsList	randomSplitAsList(weights: Array[Double], seed: Long): List[Dataset[T]]	
	sample	sample(withReplacement: Boolean, fraction: Double, seed: Long): Dataset[T]	对数据集进行抽样，withReplacement是抽样时是否放回，fraction是抽样比例，数seed是随机种子
		sample(withReplacement: Boolean, fraction: Double): Dataset[T]	

2.4 DataSet的类型化的转化操作

调整分区操作	repartition	repartition(numPartitions: Int): Dataset[T]	对数据集重新分区，可以指定分区数量，也可以指定分区函数。
		repartition(numPartitions: Int, partitionExprs: Column*): Dataset[T]	
		repartition(partitionExprs: Column*): Dataset[T]	
	coalesce	coalesce(numPartitions: Int): Dataset[T]	对数据集重进行重分区，默认不进行shuffle，适合将分区进行缩减
列操作	drop	drop(colNames: String*): DataFrame	对数据集删除列。
		drop(col: Column): DataFrame	
	withColumn	withColumn(colName: String, col: Column): DataFrame	对数据集增加列。
	withColumnRenamed	withColumnRenamed(existingName: String, newName: String): DataFrame	对数据集中的列重命名。

2.4 DataSet的类型化的转化操作

join操作	join	join(right: Dataset[_]): DataFrame	<p>join操作，如果两个数据集关联的字段名相同，可以指定字段名，如：df1.join(df2, Seq("user_id", "user_name"))。</p> <p>如果字段名不同，可以指定表名+字段名进行关联，如：df1.join(df2, \$"df1Key" === \$"df2Key")。</p> <p>还可以指定joinType，支持：inner, cross, outer, full, full_outer, left, left_outer, right, right_outer, left_semi, left_anti.</p>
		join(right: Dataset[_], usingColumn: String): DataFrame	
		join(right: Dataset[_], usingColumns: Seq[String]): DataFrame	
		join(right: Dataset[_], usingColumns: Seq[String], joinType: String): DataFrame	
		join(right: Dataset[_], joinExprs: Column): DataFrame	
		join(right: Dataset[_], joinExprs: Column, joinType: String): DataFrame	

2.4 DataSet的类型化的转化操作

分组操作	groupBy	groupBy(cols: Column*): RelationalGroupedDataset	对数据集进行分组，以便进行聚合操作，通常配合聚合操作函数使用，如：agg, avg等。
		groupBy(col1: String, cols: String*): RelationalGroupedDataset	
	groupByKey	groupByKey[K](func: (T) => K)(implicit arg0: Encoder[K]): KeyValueGroupedDataset[K, T]	对[k,v]数据集进行分组，按照k进行分组，其中数据按给func分组。
		groupByKey[K](func: MapFunction[T, K], encoder: Encoder[K]): KeyValueGroupedDataset[K, T]	
	cube	cube(cols: Column*): RelationalGroupedDataset	根据指定列为当前数据集创建一个多维多维数据集，以便进行聚合操作，通常配合聚合操作函数使用，如：agg, avg等。
		cube(col1: String, cols: String*): RelationalGroupedDataset	
聚合操作	agg	rollup(col1: String, cols: String*): RelationalGroupedDataset	根据指定列为当前数据集创建一个多维多维数据集，以便进行聚合操作，通常配合聚合操作函数使用，如：agg, avg等。
		rollup(cols: Column*): RelationalGroupedDataset	
		agg(expr: Column, exprs: Column*): DataFrame	可以指定列或者多列进行聚合操作，操作格式支持： ds.groupBy().agg(max(\$"age"), avg(\$"salary")) ds.groupBy().agg(Map("age" -> "max", "salary" -> "avg")) ds.groupBy().agg("age" -> "max", "salary" -> "avg")
		agg(exprs: Map[String, String]): DataFrame	
		agg(exprs: Map[String, String]): DataFrame	
		agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame	

2.4 DataSet的类型化的转化操作

```
//1 map操作,flatMap操作
val seq1 = Seq(Peples(20, "Michael,Andy,Geta"), Peples(25, "Sundy,Canas,Pand"))
val df1 = spark.createDataset(seq1)
val df2 = df1.map { x => (x.age + 1, x.names) }.show()
val df3 = df1.flatMap { x =>
  val a = x.age
  val s = x.names.split(",").map { x => (a, x) }
  s
}.show()
```

```
df1: org.apache.spark.sql.Dataset[Peples] = [age: int, names: string]
```

```
+---+-----+
| _1|          _2|
+---+-----+
| 21|Michael,Andy,Geta|
| 26|Sundy,Canas,Pand|
+---+-----+
```

```
df2: Unit = ()
```

```
+---+-----+
| _1|    _2|
+---+-----+
| 20|Michael|
| 20|  Andy|
| 20|  Geta|
| 25|Sundy|
| 25|Canas|
| 25|Pand|
+---+-----+
```

2.4 DataSet的类型化的转化操作

//2 filter操作,where操作

```
val seq2 = Seq(Person("Michael", 29, 170), Person("Andy", 30, 165), Person("Andy", 30, 165), Person("geta", 18, 170))
val ds4 = spark.createDataset(seq2)
ds4.filter("age >= 20 and height >= 170").show()
ds4.filter($"age" >= 20 && $"height" >= 170).show()
ds4.filter { x => x.age > 20 && x.height >= 170 }.show()
ds4.where("age >= 20 and height >= 170").show()
ds4.where($"age" >= 20 && $"height" >= 170).show()
```

```
seq2: Seq[Person] = List(Person(Michael,29,170), Person(Andy,30,165), Person(Andy,30,165), Person(geta,18,170))
```

```
ds4: org.apache.spark.sql.Dataset[Person] = [name: string, age: int ... 1 more field]
```

```
+-----+---+-----+
|  name|age|height|
```

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

```
|Michael| 29|  170|
```

```
|  gigt| 29|  170|
```

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

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

```
|  name|age|height|
```

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

```
|Michael| 29|  170|
```

```
|  gigt| 29|  170|
```

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

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

```
|  name|age|height|
```

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

```
|Michael| 29|  170|
```

2.4 DataSet的类型化的转化操作

```
//3 去重操作
ds4.distinct().show()
ds4.dropDuplicates("age").show()
ds4.dropDuplicates("age", "height").show()
ds4.dropDuplicates(Seq("age", "height")).show()
ds4.dropDuplicates(Array("age", "height")).show()
```

```
+-----+-----+
|  name|age|height|
```

```
+-----+-----+
|   agg| 18|   160|
|  Andy| 30|   165|
|  geta| 18|   170|
|  gigt| 29|   170|
|Michael| 29|   170|
```

```
+-----+-----+
+-----+-----+
|  name|age|height|
```

```
+-----+-----+
|Michael| 29|   170|
|   Andy| 30|   165|
|   geta| 18|   170|
```

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


2.4 DataSet的类型化的转化操作

//4 加法/减法操作

```
val seq3 = Seq(Person("Cici", 16, 160), Person("Andy", 30, 165), Person("Pecith", 22, 170), Person("Hanso
val df5 = spark.createDataset(seq3)
ds4.except(df5).show()
ds4.union(df5).show()
ds4.intersect(df5).show()
```

```
seq3: Seq[Person] = List(Person(Cici,16,160), Person(Andy,30,165), Person(Pecith,22,170), Person(Hanson,18,170)
df5: org.apache.spark.sql.Dataset[Person] = [name: string, age: int ... 1 more field]
```

```
+-----+-----+
|  name|age|height|
+-----+-----+
|   agg| 18|   160|
|  geta| 18|   170|
|  gigt| 29|   170|
|Michael| 29|   170|
+-----+-----+
+-----+-----+
|  name|age|height|
+-----+-----+
|Michael| 29|   170|
|   Andy| 30|   165|
|   Andy| 30|   165|
|  geta| 18|   170|
|   agg| 18|   160|
```

2.4 DataSet的类型化的转化操作

```
//5 select操作
df5.select("name", "age").show()
df5.select(expr("height + 1").as[Int]).show()
```

```
+-----+----+
|  name|age|
+-----+----+
|  Cici| 16|
|  Andy| 30|
|Pecith| 22|
|Hanson| 18|
+-----+----+
+-----+
|(height + 1)|
+-----+
|           161|
|           166|
|           171|
|           171|
+-----+
```

2.4 DataSet的类型化的转化操作

```
//6 排序操作
df5.sort("age").show()
df5.sort($"age".desc, $"height".desc).show()
df5.orderBy("age").show()
df5.orderBy($"age".desc, $"height".desc).show()
```

```
|Hanson| 18|   170|
|  Cici| 16|   160|
+-----+---+-----+
+-----+---+-----+
|  name|age|height|
+-----+---+-----+
|  Cici| 16|   160|
|Hanson| 18|   170|
|Pecith| 22|   170|
|  Andy| 30|   165|
+-----+---+-----+
+-----+---+-----+
|  name|age|height|
+-----+---+-----+
|  Andy| 30|   165|
|Pecith| 22|   170|
|Hanson| 18|   170|
|  Cici| 16|   160|
```

2.4 DataSet的类型化的转化操作

```
//7 分割抽样操作
val df6 = ds4.union(df5)
val rands = df6.randomSplit(Array(0.3, 0.7))
rands(0).count()
rands(1).count()
rands(0).show()
rands(1).show()
val df7 = df6.sample(false, 0.5)
df7.count()
df7.show()
```

```
df6: org.apache.spark.sql.Dataset[Person] = [name: string, age: int ... 1 more field]
rands: Array[org.apache.spark.sql.Dataset[Person]] = Array([name: string, age: int ... 1 more field], [name: s
res27: Long = 5
res28: Long = 5
+-----+---+-----+
|  name|age|height|
+-----+---+-----+
|Michael| 29|   170|
|  geta| 18|   170|
|  Andy| 30|   165|
| Pecith| 22|   170|
| Hanson| 18|   170|
+-----+---+-----+
+----+---+-----+
|name|age|height|
+----+---+-----+
```

2.4 DataSet的类型化的转化操作

```
//8 列操作
val df8 = df6.drop("height")
df8.columns
df8.show()
val df9 = df6.withColumn("add2", $"age" + 2)
df9.columns
df9.show()
val df10 = df9.withColumnRenamed("add2", "age_new")
df10.columns
df10.show()
```

```
df8: org.apache.spark.sql.DataFrame = [name: string, age: int]
```

```
res34: Array[String] = Array(name, age)
```

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

```
|   name|age|
```

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

```
|Michael| 29|
```

```
|   Andy| 30|
```

```
|   Andy| 30|
```

```
|   geta| 18|
```

```
|   agg| 18|
```

```
|   gigt| 29|
```

```
|   Cici| 16|
```

```
|   Andy| 30|
```

```
| Pecith| 22|
```

```
| Hanson| 18|
```

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

2.4 DataSet的类型化的转化操作

```
//8 列操作
val df8 = df6.drop("height")
df8.columns
df8.show()
val df9 = df6.withColumn("add2", $"age" + 2)
df9.columns
df9.show()
val df10 = df9.withColumnRenamed("add2", "age_new")
df10.columns
df10.show()
df6.withColumn("add_col", lit(1)).show
```

2.4 DataSet的类型化的转化操作

```
df9: org.apache.spark.sql.DataFrame = [name: string, age: int ... 2 more fields]
```

```
res36: Array[String] = Array(name, age, height, add2)
```

```
+-----+---+-----+-----+
|  name|age|height|add2|
+-----+---+-----+-----+
|Michael| 29|   170|  31|
|  Andy| 30|   165|  32|
|  Andy| 30|   165|  32|
|  geta| 18|   170|  20|
|   agg| 18|   160|  20|
|  gigt| 29|   170|  31|
|  Cici| 16|   160|  18|
|  Andy| 30|   165|  32|
| Pecith| 22|   170|  24|
| Hanson| 18|   170|  20|
+-----+---+-----+-----+
```

2.4 DataSet的类型化的转化操作

```
//8 join 操作
val seq4 = Seq(Score("Cici", 85), Score("Andy", 70), Score("Pecith", 90), Score("Bibi", 88))
val df11 = spark.createDataset(seq4)

val df12 = df5.join(df11, Seq("name"), "inner")
df12.show()
val df13 = df5.join(df11, Seq("name"), "left")
df13.show()
```

```
seq4: Seq[Score] = List(Score(Cici,85), Score(Andy,70), Score(Pecith,90), Score(Bibi,88))
```

```
df11: org.apache.spark.sql.Dataset[Score] = [name: string, score: int]
```

```
df12: org.apache.spark.sql.DataFrame = [name: string, age: int ... 2 more fields]
```

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

```
| name|age|height|score|
```

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

```
| Cici| 16|   160|   85|
```

```
| Andy| 30|   165|   70|
```

```
|Pecith| 22|   170|   90|
```

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

```
df13: org.apache.spark.sql.DataFrame = [name: string, age: int ... 2 more fields]
```

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

```
| name|age|height|score|
```

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

```
| Cici| 16|   160|   85|
```

```
| Andy| 30|   165|   70|
```

```
|Pecith| 22|   170|   90|
```

```
|Hanson| 18|   170| null|
```


2.4 DataSet的类型化的转化操作

```
//9 分组聚合 操作
```

```
val df14 = ds4.union(df5).groupBy("height").agg(avg("age").as("avg_age"))  
df14.show()
```

```
df14: org.apache.spark.sql.DataFrame = [height: int, avg_age: double]
```

```
+-----+-----+  
|height|avg_age|  
+-----+-----+  
|   165|   30.0|  
|   160|   17.0|  
|   170|   23.2|  
+-----+-----+
```

2.5 DataSet内置函数：Aggregate functions



Aggregate functions		
操作名称	用法	说明
近似去重统计	approx_count_distinct(columnName: String, rsd: Double): Column	近似计算分组下的distinct items数量 rsd允许最大估计误差（默认= 0.05）
	approx_count_distinct(e: Column, rsd: Double): Column	
	pprox_count_distinct(columnName: String): Column	
	approx_count_distinct(e: Column): Column	
平均值统计	avg(columnName: String): Column	计算分组下平均值
	avg(e: Column): Column	
多行合并成数组	collect_list(columnName: String): Column	在分组中指定列进行合并，多行数据合并成一个数组，其中list包含重复数据，set中不包含重复数据
	collect_list(e: Column): Column	
	collect_set(columnName: String): Column	
	collect_set(e: Column): Column	
相关系数	corr(columnName1: String, columnName2: String): Column	返回两列的Pearson相关系数
	corr(column1: Column, column2: Column): Column	
计数统计	count(columnName: String): TypedColumn[Any, Long]	计数统计，其中Distinct是去重统计
	count(e: Column): Column	
	countDistinct(columnName: String, columnNames: String*): Column	
	countDistinct(expr: Column, exprs: Column*): Column	

2.5 DataSet内置函数：Aggregate functions



方差统计	covar_pop(columnName1: String, columnName2: String): Column	返回两列的总体协方差。 返回两列的样本协方差。
	covar_pop(column1: Column, column2: Column): Column	
	covar_samp(columnName1: String, columnName2: String): Column	
	covar_samp(column1: Column, column2: Column): Column	
取第1行数据	first(columnName: String): Column	返回分组中第1行数据
	first(e: Column): Column	
	first(columnName: String, ignoreNulls: Boolean): Column	
	first(e: Column, ignoreNulls: Boolean): Column	
取最后1行数据	last(columnName: String): Column	返回分组中最后1行数据
	last(e: Column): Column	
	last(columnName: String, ignoreNulls: Boolean): Column	
	last(e: Column, ignoreNulls: Boolean): Column	
最大值	max(columnName: String): Column	返回分组中最大数据
	max(e: Column): Column	
平均值	mean(columnName: String): Column	返回分组中平均数据
	mean(e: Column): Column	
最小值	min(columnName: String): Column	返回分组中最小数据
	min(e: Column): Column	

2.5 DataSet内置函数：Aggregate functions



聚合标记	grouping(columnName: String): Column	对指定列进行标记是否聚合，1为聚合，0为不聚合
	grouping(e: Column): Column	
	grouping_id(colName: String, colNames: String*): Column	
	grouping_id(cols: Column*): Column	
峰度 (Kurtosis)	kurtosis(columnName: String): Column	返回分组中峰度数据
	kurtosis(e: Column): Column	
偏度 (Skewness)	skewness(columnName: String): Column	返回分组中偏度数据
	skewness(e: Column): Column	
样本标准差 (Standard Deviation)	stddev(columnName: String): Column	标准差
	stddev(e: Column): Column	
	stddev_samp(columnName: String): Column	
	stddev_samp(e: Column): Column	
	stddev_pop(columnName: String): Column	
	stddev_pop(e: Column): Column	
总体标准差 (population standard deviation)	stddev_samp(columnName: String): Column	标准差
	stddev_samp(e: Column): Column	

2.5 DataSet内置函数：Aggregate functions



求和	sum(columnName: String): Column	求和操作，支持去重求和
	sum(e: Column): Column	
	sumDistinct(columnName: String): Column	
	sumDistinct(e: Column): Column	
总体方差(Variance)	var_pop(columnName: String): Column	方差
	var_pop(e: Column): Column	
样本方差(Variance)	variance(columnName: String): Column	方差
	variance(e: Column): Column	
	var_samp(columnName: String): Column	
	var_samp(e: Column): Column	

2.5 DataSet内置函数：Collection functions



Collection functions		
操作名称	用法	说明
判断数组是否包含指定数据	<code>array_contains(column: Column, value: Any): Column</code>	如果数组为空，则返回null，如果数组包含值，则返回true，否则返回false。
行转多列，将数组中的元素转成每一行	<code>explode(e: Column): Column</code>	为给定数组中的每个元素创建一个新行。
	<code>explode_outer(e: Column): Column</code>	与explode不同，如果array / map为空或为null，则生成null。
解析JSON字符串	<code>from_json(e: Column, schema: String, options: Map[String, String]): Column</code>	schema为json的格式类型，options为解析参数
	<code>from_json(e: Column, schema: DataType): Column</code>	
	<code>from_json(e: Column, schema: StructType): Column</code>	
	<code>from_json(e: Column, schema: DataType, options: Map[String, String]): Column</code>	
	<code>from_json(e: Column, schema: StructType, options: Map[String, String]): Column</code>	
	<code>from_json(e: Column, schema: DataType, options: Map[String, String]): Column</code>	
	<code>from_json(e: Column, schema: StructType, options: Map[String, String]): Column</code>	

2.5 DataSet内置函数：Collection functions



读取json数据	get_json_object(e: Column, path: String): Column	基于指定的json路径从json字符串中提取json对象
	json_tuple(json: Column, fields: String*): Column	为json列创建一个新行
转成json对象	to_json(e: Column): Column	将StructType或StructType的ArrayType的列转换为具有指定结构的JSON字符串。
	to_json(e: Column, options: Map[String, String]): Column	
	to_json(e: Column, options: Map[String, String]): Column	
数组或者json对象 转成多行	json_tuple(json: Column, fields: String*): Column	为json列中每个数据创建一个新行
	posexplode(e: Column): Column	为给定数组中的每个元素创建一个新行
集合大小	size(e: Column): Column	集合长度
数组排序	sort_array(e: Column, asc: Boolean): Column	对数据进行排序
	sort_array(e: Column): Column	

2.5 DataSet内置函数：Date time functions



Date time functions

操作名称	用法	说明
日期、时间操作	add_months(startDate: Column, numMonths: Int): Column	日期加法，月
	current_date(): Column	返回当前日期，新增加一列日期
	current_timestamp(): Column	返回当前时间，新增加一列时间戳
	date_add(start: Column, days: Int): Column	日期加法，日
	date_format(dateExpr: Column, format: String): Column	将日期/时间戳/字符串转换指定format格式的字符串
	date_sub(start: Column, days: Int): Column	日期减法，日
	datediff(end: Column, start: Column): Column	计算日期间隔
	dayofmonth(e: Column): Column	取日期的月份
	dayofyear(e: Column): Column	取日期的年
	from_unixtime(ut: Column, f: String): Column	取unixtime
	from_unixtime(ut: Column): Column	取unixtime
	from_utc_timestamp(ts: Column, tz: String): Column	取utc
	hour(e: Column): Column	取日期的小时
	last_day(e: Column): Column	给定一个日期列，返回给定日期所属的月份的最后一天。例如，从2015年7月31日起，输入“2015-07-27”返回“2015-07-31”，是2015年7月的最后一天。

2.5 DataSet内置函数：Date time functions



日期、时间操作

minute(e: Column): Column	取日期的分钟
month(e: Column): Column	取日期的月份
months_between(date1: Column, date2: Column): Column	计算日期间隔，月
next_day(date: Column, dayOfWeek: String): Column	给定一个日期列，返回下一个周期的日期。 例如next_day ('2015-07-27' , "Sunday") 返回2015-08-02， 因为那是2015-07-27之后的第一个星期天。
quarter(e: Column): Column	取日期的季度
second(e: Column): Column	取日期的秒
to_date(e: Column, fmt: String): Column	将列转换为具有指定格式的DateType
to_date(e: Column): Column	
to_timestamp(s: Column, fmt: String): Column	将时间字符串转换为具有指定格式的Unix时间戳（以秒为单位）
to_timestamp(s: Column): Column	
to_utc_timestamp(ts: Column, tz: String): Column	将时间字符串转换为具有指定格式的utc时间戳
trunc(date: Column, format: String): Column	格式转换
unix_timestamp(s: Column, p: String): Column	
unix_timestamp(s: Column): Column	
unix_timestamp(): Column	

2.5 DataSet内置函数：Date time functions



日期、时间操作	weekofyear(e: Column): Column	取第几周
	window(timeColumn: Column, windowDuration: String): Column	生成指定列的时间戳的滚动时间窗口
	window(timeColumn: Column, windowDuration: String, slideDuration: String): Column	
	window(timeColumn: Column, windowDuration: String, slideDuration: String, startTime: String): Column	
	year(e: Column): Column	取年

2.5 DataSet内置函数：Math functions

Math functions		
操作名称	用法	说明
数值计算	acos	反余弦
	asin	反正弦
	atan	反正切
	atan2	atan2(y,x)所表达的意思是坐标原点为起点，指向(x,y)的射线在坐标平面上与x轴正方向之间的角的角度。
	bin	二进制
	bround	四舍五入
	cbrt	立方根
	ceil	四舍五入
	conv	基数转换
	cos	余弦
	cosh	双曲余弦
	degrees	将以弧度测量的角度转换为以度为单位的近似等效角度。
	exp	指数
	expm1	计算给定值的指数减去1

2.5 DataSet内置函数：Math functions

数值计算	factorial	阶乘
	floor	四舍五入
	hex	十六进制
	hypot	计算 $\sqrt{a^2 + b^2}$
	log	对数
	log10	对数，10为底
	log1p	对数，减1
	log2	对数，2为底
	pmod	取模
	pow	幂
	radians	将以度为单位的角度转换为以弧度测量的大致相等的角度。
	rint	格式转换
	round	取整
	shiftLeft	左移
	shiftRight	右移
	shiftRightUnsigned	无符号移动
	signum	符号函数

2.5 DataSet内置函数：Math functions

数值计算	sin	正弦
	sinh	双曲正弦
	sqrt	开方
	tan	正切
	tanh	双曲正切
	unhex	十六进制转二进制

2.5 DataSet内置函数：String functions

String functions		
操作名称	用法	说明
字符串操作	ascii(e: Column): Column	计算字符串列的第一个字符的数值
	base64(e: Column): Column	计算二进制列的BASE64编码
	concat(exprs: Column*): Column	将多个字符串列连接成一个字符串。
	concat_ws(sep: String, exprs: Column*): Column	将多个字符串列连接成一个字符串，可以指定分割符。
	decode(value: Column, charset: String): Column	解码
	encode(value: Column, charset: String): Column	编码
	format_number(x: Column, d: Int): Column	数字格式化
	format_string(format: String, arguments: Column*): Column	数字格式化
	initcap(e: Column): Column	单词首字母大写
	instr(str: Column, substring: String): Column	在给定的字符串中找到substr的第一个出现位置
	length(e: Column): Column	长度
	levenshtein(l: Column, r: Column): Column	计算两个给定字符串列的Levenshtein距离
	locate(substr: String, str: Column, pos: Int): Column	在位置pos之后，找到字符串列中第一次出现substr的位置。
	locate(substr: String, str: Column): Column	找到第一次出现的substr的位置。
	lower(e: Column): Column	将字符串列转换为小写

2.5 DataSet内置函数：String functions

字符串操作

lpad(str: Column, len: Int, pad: String): Column	左截取
ltrim(e: Column): Column	左去除空格
regexp_extract(e: Column, exp: String, groupIdx: Int): Column	正则匹配
regexp_replace(e: Column, pattern: Column, replacement: Column): Column	
regexp_replace(e: Column, pattern: String, replacement: String): Column	
repeat(str: Column, n: Int): Column	重复字符n次
reverse(str: Column): Column	反转字符串列
rpadd(str: Column, len: Int, pad: String): Column	右截取
rtrim(e: Column): Column	右去除空格
soundex(e: Column): Column	返回指定表达式的soundex代码
split(str: Column, pattern: String): Column	字符串分割
substring(str: Column, pos: Int, len: Int): Column	子串
substring_index(str: Column, delim: String, count: Int): Column	子串
translate(src: Column, matchingString: String, replaceString: String): Column	字符翻译
trim(e: Column): Column	两端去除空格
unbase64(e: Column): Column	解码
upper(e: Column): Column	将字符串列转换为大写

2.5 DataSet内置函数：Non-aggregate functions

Non-aggregate functions		
操作名称	用法	说明
非聚合函数操作	<code>abs(e: Column): Column</code>	计算绝对值
	<code>array(colName: String, colNames: String*): Column</code>	创建一个数组列。 输入列必须都具有相同的数据类型。
	<code>array(cols: Column*): Column</code>	
	<code>bitwiseNOT(e: Column): Column</code>	按位计算NOT
	<code>broadcast[T](df: Dataset[T]): Dataset[T]</code>	广播数据集
	<code>coalesce(e: Column*): Column</code>	返回不为null的第一列，如果所有输入都为空，则返回null。
	<code>col(colName: String): Column</code>	根据给定的列名返回一个列。
	<code>column(colName: String): Column</code>	
	<code>expr(expr: String): Column</code>	将字符串解析为列表达式。例如： <code>df.groupBy(expr("length(word)")).count()</code>
	<code>greatest(columnName: String, columnNames: String*): Column</code>	返回指定列的最大值，跳过空值。
	<code>greatest(exprs: Column*): Column</code>	
	<code>input_file_name(): Column</code>	为当前Spark任务的文件名创建一个字符串列。
	<code>isnan(e: Column): Column</code>	如果列是NaN，返回true。

2.5 DataSet内置函数：Non-aggregate functions

非聚合函数操作	<code>isnull(e: Column): Column</code>	如果列为空，则返回true。
	<code>least(columnName: String, columnNames: String*): Column</code>	返回指定列的最小值，跳过空值。
	<code>least(exprs: Column*): Column</code>	
	<code>lit(literal: Any): Column</code>	创建字面量列。
	<code>map(cols: Column*): Column</code>	map操作
	<code>monotonically_increasing_id(): Column</code>	一个产生单调递增的64位整数的列表达式。
	<code>nanvl(col1: Column, col2: Column): Column</code>	如果不是NaN，则返回col1，如果col1是NaN，则返回col2
	<code>negate(e: Column): Column</code>	相反数， <code>df.select(-df("amount"))</code>
	<code>not(e: Column): Column</code>	逻辑非
	<code>rand(): Column</code>	随机数，[0-1]之间。可以支持正态分布
	<code>rand(seed: Long): Column</code>	
	<code>randn(): Column</code>	
	<code>spark_partition_id(): Column</code>	Partition ID
	<code>struct(colName: String, colNames: String*): Column</code>	创建一个新的结构列
	<code>struct(cols: Column*): Column</code>	
	<code>typedLit[T](literal: T)(implicit arg0: scala.reflect.api.JavaUniverse.TypeTag[T]): Column</code>	创建字面量列
	<code>when(condition: Column, value: Any): Column</code>	条件判断取值，例如： <code>people.select(when(people("gender") === "male", 0) .when(people("gender") === "female", 1) .otherwise(2))</code>

Thanks

FAQ时间