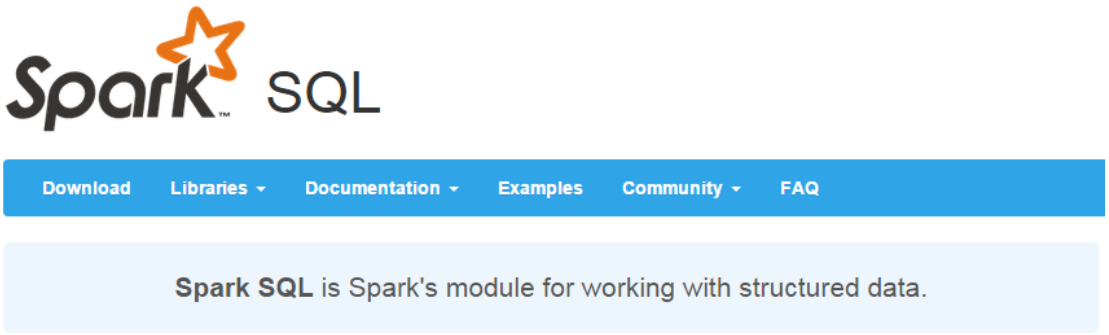
1 SparkSQL



Spark SQL 是 Spark 用来处理**结构化数据**的一个模块。

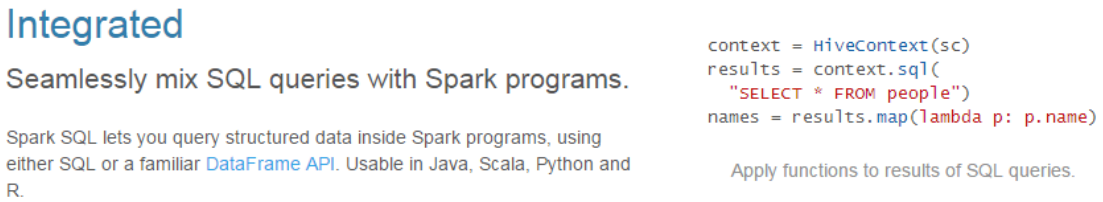
1.1 SparkSQL 的概念

Spark SQL 是一个用来处理结构化数据的 spark 组件，也可被视为一个**分布式的 SQL 查询引擎**。与基础的 Spark RDD API 不同，Spark SQL 提供了查询结构化数据及计算结果等信息的接口。在内部，Spark SQL 使用这个额外的信息去执行额外的优化.有几种方式可以跟 Spark SQL 进行交互，包括 SQL 和 Dataset API(DSL)。

1.2 Spark SQL 优点

Spark SQL，是将 Spark SQL 转换成 RDD，然后提交到集群执行，执行效率非常快!

1.易整合



2.统一的数据访问方式

Uniform Data Access

Connect to any data source the same way.

DataFrames and SQL provide a common way to access a variety of data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC. You can even join data across these sources.

```
context.jsonFile("s3n://...")
  .registerTempTable("json")
results = context.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

Query and join different data sources.

3.兼容 Hive

hive on Spark

hive 依然比较流行，报表，结构化的数据

udf uers defined function 用户自定义函数 sum avg count(*) myudf(ip) ➔ 10 进制数据

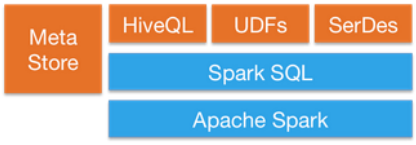
udaf aggregate 输入多个，返回一个结果

hive On spark

Hive Compatibility

Run unmodified Hive queries on existing data.

Spark SQL reuses the Hive frontend and metastore, giving you full compatibility with existing Hive data, queries, and UDFs. Simply install it alongside Hive.



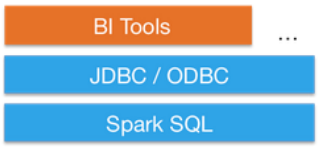
Spark SQL can use existing Hive metastores, SerDes, and UDFs.

4.标准的数据连接

Standard Connectivity

Connect through JDBC or ODBC.

A server mode provides industry standard JDBC and ODBC connectivity for business intelligence tools.



Use your existing BI tools to query big data.

1.3 DataFrames

spark core --- rdd
sparksql --- dataframe
mysql 关系型数据库， accesslog(province varchar(20),cnts int) “河南省”，100
rdd + shema 信息
schema: 结构 字段名称，字段的类型，是否可以为空

1.3.1 什么是 DataFrames（1.3）

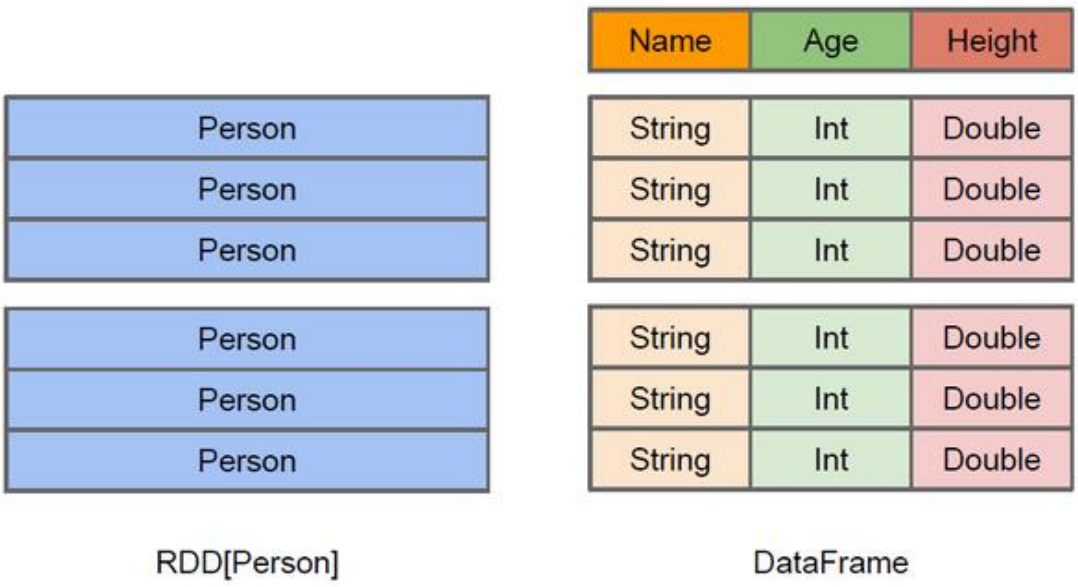
dataframe 是从 1.3 版本引入的

与 RDD 相似，DataFrame 也是一个不可变分布式的数据集合。但与 RDD 不同的是，数据都被组织到有名字的列中，就像关系型数据库中的表一样,除了数据之外，还记录着数据的结构信息，即 **schema**。设计 DataFrame 的目的就是要让对大型数据集的处理变得更简单，它让开发者可以为分布式的数据集指定一个模式，进行更高层次的抽象。还提供了特定领域内专用的 API 来处理分布式数据。

从 API 易用性的角度上 看，DataFrame API 提供的是一套高层的关系操作，比函数式的 RDD API 要更加友好，门槛更低。由于与 R 和 Pandas 的 DataFrame 类似，Spark DataFrame 很好地继承了传统单机数据分析的开发体验。

dataframe = rdd + schema

dataframe 就是带着 schema 信息的 rdd



1.3.2 创建 DataFrames

1.在本地创建一个文件，有三列，分别是 id、name、age，用空格分隔，然后上传到 hdfs 上

```
hdfs dfs -put person.txt /
```

2.在 spark shell 执行下面命令，读取数据，将每一行的数据使用列分隔符分割

```
val lineRDD = sc.textFile("hdfs://hdp-01:9000/person.txt").map(_.split(" "))
```

3.定义 case class（相当于表的 schema）

```
case class Person(id:Int, name:String, age:Int)
```

4.将 RDD 和 case class 关联

```
val personRDD = lineRDD.map(x => Person(x(0).toInt, x(1), x(2).toInt))
```

5.将 RDD 转换成 DataFrame

```
val personDF = personRDD.toDF
```

6.对 DataFrame 进行处理

```
personDF.show
```

1.4 Spark-shell 中操作 DataFrame

1.4.1 DSL 风格语法

DSL : domain-specific language

//查看 DataFrame 中的内容

```
personDF.show
```

//查看 DataFrame 部分列中的内容

```
personDF.select(personDF.col("name")).show
personDF.select(col("name"), col("age")).show
personDF.select(personDF ("name")).show
personDF.select("name").show
```

//打印 DataFrame 的 Schema 信息

```
personDF.printSchema
personDF.schema
```

//查询所有的 name 和 age，并将 age+1

```
personDF.select(col("id"), col("name"), col("age") + 1).show
personDF.select(personDF("id"), personDF("name"), personDF("age") + 1).show
personDF.select(personDF("name"),personDF.col("age")+10 as "tenyears").show
```

//过滤 age 大于等于 18 的

```
personDF.filter(col("age") >= 18).show
```

//按年龄进行分组并统计相同年龄的人数

```
personDF.groupBy("age").count().show()
```

1.4.2 SQL 风格语法

如果想使用 SQL 风格的语法，需要将 DataFrame 注册成表

```
personDF.registerTempTable("t_person")
```

//查询年龄最大的前两名

```
sqlContext.sql("select * from t_person order by age desc limit 2").show
```

//显示表的 Schema 信息

```
sqlContext.sql("desc t_person").show
```

1.5 DataFrame 开发操作

1.5.1 SQL 风格语法

1.5.1.1导入 jar 包

使用 sparksql 需要导入 sparksql 的依赖 jar 包

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>${spark.version}</version>
</dependency>
```

1.5.1.2 toDF 方法

使用 toDF 方式，通过反射，指定 schema 信息

```
object SqlDemo1 {
  def main(args: Array[String]): Unit = {
    val conf: SparkConf = new SparkConf()
    conf.setMaster("local").setAppName("sql")
```

```
    val sc: SparkContext = new SparkContext(conf)
// 创建 sparksql 的操作对象
    val sqlContext: SQLContext = new SQLContext(sc)
    // 准备数据 先从集合中准备
    val data: RDD[String] = sc.parallelize(Array("laoduan 9999 30","laozhao 99 32","zs 99 28"))
    // 数据预处理操作
    val maped: RDD[Boy] = data.map({
      arr =>
        // 获取各属性值
        val fields: Array[String] = arr.split(" ")
        val name = fields(0)
        val fv = fields(1).toDouble
        val age = fields(2).toInt
        // 封装成对象
        Boy(name, fv, age)
    })
    // RDD 类型的数据，想要使用 sql 来操作，就必须进行转换 转换成 DataFrame
    // 需要导入 sqlContext 对象的隐式转换
    import sqlContext.implicits._
    // 通过 toDF 方法，把 RDD 转换成了 DataFrame 类型
    val boyDF: DataFrame = maped.toDF
    // 想要使用 sql 语法来查询，还需要把 df 对象注册成一张表
    boyDF.registerTempTable("boy")
    // 使用 sql 语句执行查询,返回 DataFrame 对象， lazy 执行的，
    val resDF: DataFrame = sqlContext.sql("select * from boy order by fv desc ,age asc")
    // 调用 action 方法执行
    resDF.show()
  }
}
// case class 用于封装数据
case class Boy(name:String,fv:Double,age:Int)
```

1.5.1.3 自定义 schema

使用 StructType 方式，指定 schema 信息

```
  def main(args: Array[String]): Unit = {
    val conf: SparkConf = new SparkConf()
    conf.setMaster("local").setAppName("sql")
    val sc: SparkContext = new SparkContext(conf)
// 创建 sparksql 的操作对象
    val sqlContext: SQLContext = new SQLContext(sc)
    // 准备数据 先从集合中准备
```

```
    val data: RDD[String] = sc.parallelize(Array("laoduan 9999 30","laozhao 99 32","zs 99 28"))

    // 数据预处理操作
    val rdd= data.map({
      arr =>
        // 获取各属性值
        val fields: Array[String] = arr.split(" ")
        val name = fields(0)
        val fv = fields(1).toDouble
        val age = fields(2).toInt
      // 不封装 case class，需要使用 Row 对象来封装
       Row(name,fv,age)
    })
    // 将 RDD 关联 scheme
    // structtype 里面封装的是对象的类型信息
    val schema:StructType=StructType(
      List(
        StructField("name",StringType),
        StructField("fv",DoubleType),
        StructField("age",IntegerType)
      ))
    val boyDF: DataFrame = sqlContext.createDataFrame(rdd,schema)
    // 想要使用 sql 语法来查询，还需要把 df 对象注册成一张表
    boyDF.registerTempTable("boy")
    // 使用 sql 语句执行查询,返回 DataFrame 对象， lazy 执行的，
    val resDF: DataFrame = sqlContext.sql("select * from boy order by fv desc ,age asc")
    // 调用 action 方法执行
    resDF.show()
  }
```

1.5.2 DSL 风格语法

```
def main(args: Array[String]): Unit = {
  val conf: SparkConf = new SparkConf()
  conf.setMaster("local").setAppName("sql")
  val sc: SparkContext = new SparkContext(conf)
```

```
  // 创建 sparksql 的操作对象
  val sqlContext: SQLContext = new SQLContext(sc)
  // 准备数据 先从集合中准备
  val data: RDD[String] = sc.parallelize(Array("laoduan 9999 30", "laozhao 99 32", "zs 99 28"))

  // 数据预处理操作
  val rdd = data.map({
    arr =>
      // 获取各属性值
      val fields: Array[String] = arr.split(" ")
      val name = fields(0)
      val fv = fields(1).toDouble
      val age = fields(2).toInt
      // 不封装 case class，需要使用 Row 对象来封装
      Row(name, fv, age)
  })
  // 将 RDD 关联 scheme
  // structtype 里面封装的是对象的类型信息
  val schema: StructType = StructType(
    List(
      StructField("name", StringType),
      StructField("fv", DoubleType),
      StructField("age", IntegerType)
    ))
  val boyDF: DataFrame = sqlContext.createDataFrame(rdd, schema)
  // 指定列名
  val df1: DataFrame = boyDF.select("name", "fv")
  // 指定过滤条件
  val df2 = df1.where("fv > 90")
  // 排序  默认升序  sort 和 orderBy 方法都可实现
  //      val df3 = df2.sort("fv")
  // 如果按照指定的排序，需要导入隐式转换
  import sqlContext.implicits._
  val df3 = df2.orderBy($"fv" desc)
  // 可以一行搞定
  val by: Dataset[Row] = boyDF.select("name","fv").where("fv > 90").orderBy($"fv" desc)
  // 查看结果
  df3.show()
}
```

DataFrame 的 wordcount

```
val conf = new SparkConf()
```

```
  .setMaster("local")
  .setAppName(DFwc.getClass.getSimpleName)
val sc = new SparkContext(conf)
// 获取 sql 的操作对象
val ssc = new SQLContext(sc)

val file = sc.textFile("wc.log").flatMap(_.split(" ")).map(Row(_))

val schema = StructType(List(StructField("word", StringType)))
val pdf = ssc.createDataFrame(file, schema)

// sql 语法风格
pdf.registerTempTable("t_wc")
ssc.sql("select word,count(*) as cnts from t_wc group By word order by cnts desc").show()

// DSL 语法风格
import ssc.implicits._
pdf.select("word").groupBy("word").count().orderBy($"count" desc).show()
```

1.5.3 spark 1.x SQL 的用法总结

1.创建 SparkContext
2.创建 SQLContext
3.创建 RDD
4.创建一个样例类，并定义类的成员变量
5.整理数据并关联 class
6.将 RDD 转换成 DataFrame（导入隐式转换）
7.将 DataFrame 注册成临时表
8.书写 SQL（Transformation）
9.执行 Action

1.创建 SparkContext
2.创建 SQLContext
3.创建 RDD
4.创建 StructType（schema）
5.整理数据将数据跟 Row 关联
6.通过 rowRDD 和 schema 创建 DataFrame
7.将 DataFrame 注册成临时表
8.书写 SQL（Transformation）
9.执行 Action

1.6 Datasets and DataFrames

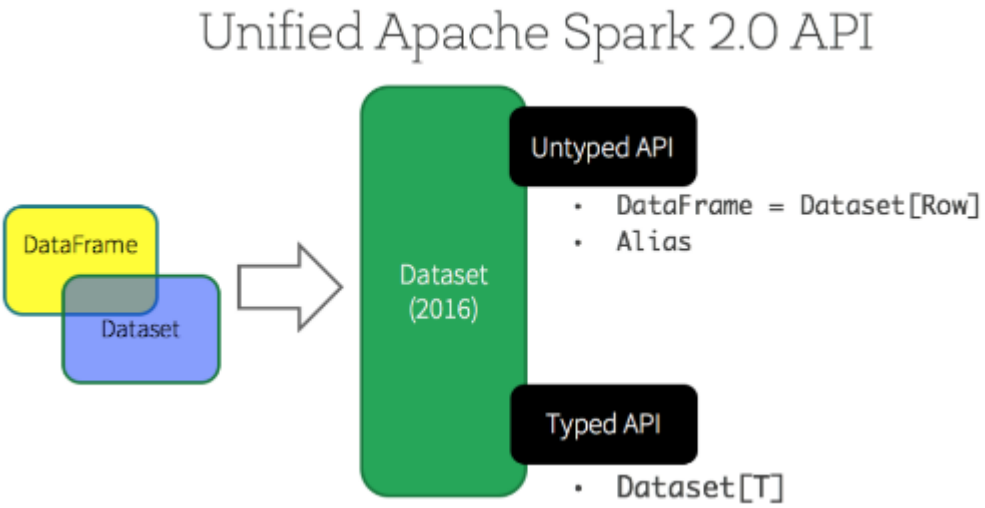
spark1.6 版本引入，dataset
在 spark2.x 后，dataset 和 DataFrame 合二为一。 DataFrame = Dataset[Row]

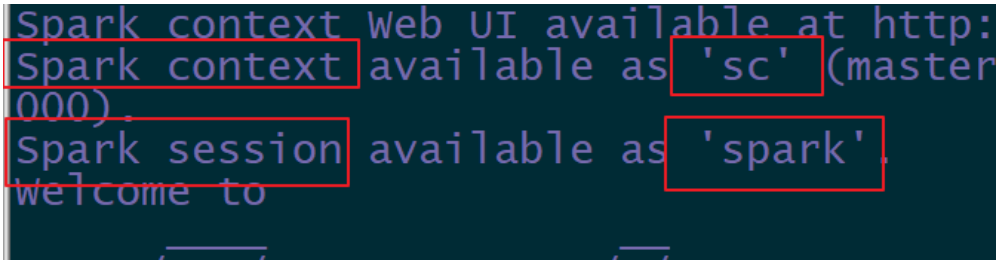
Dataset 是一个分布式的数据集合,Dataset 是在 Spark 1.6 中被添加的新接口，它提供了 RDD 的优点（强类型化，能够使用强大的函数）与 Spark SQL 执行引擎的优点.一个 Dataset 可以从 JVM 对象来 构造 并且使用转换功能（map, flatMap, filter, 等等）.

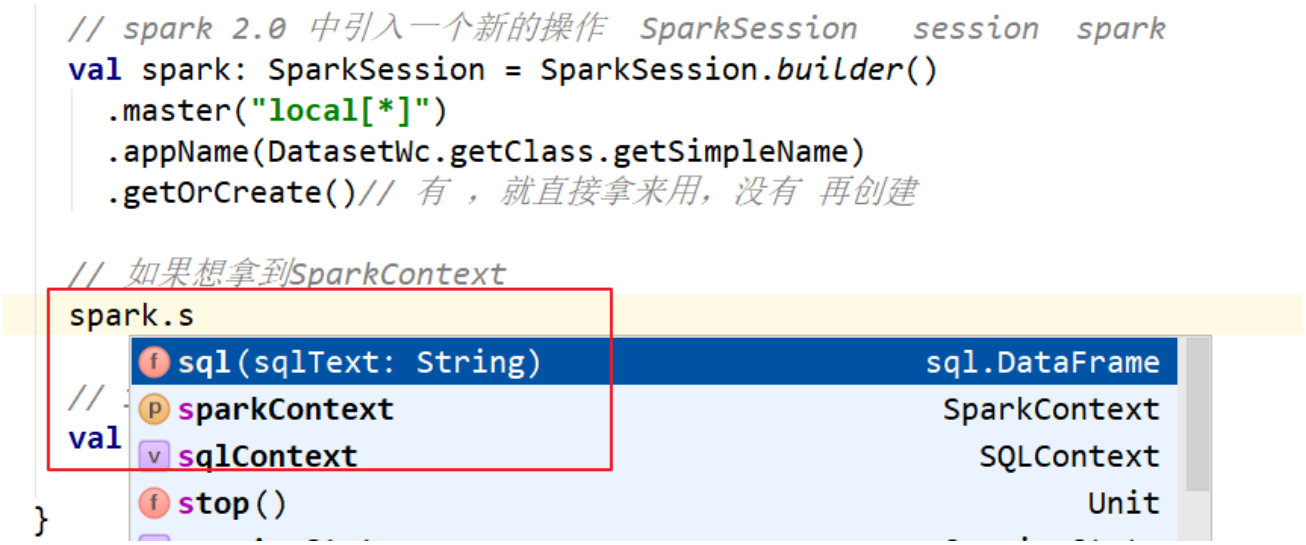
spark2.x 后，dataset 和 DataFrame 合并，一个 DataFrame 是一个 Dataset 组成的指定列.

DataFrame = Dataset[Row]

在 Scala 和 Java 中，一个 DataFrame 所代表的是一个多个 Row (行) 的 Dataset (数据集合). 在 Scala API 中，DataFrame 仅仅是一个 Dataset[Row]类型的别名. 然而，在 Java API 中，用户需要去使用 Dataset<Row> 去代表一个 DataFrame.







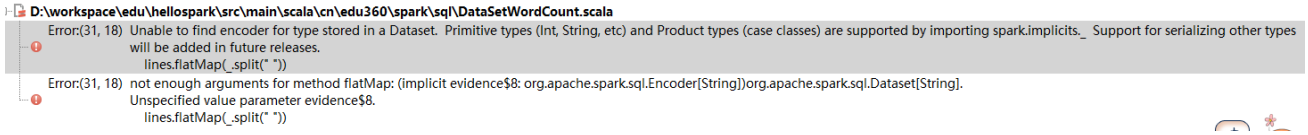
1.7 DataSet 编程

sqlWordCount：

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

  // 在 spark2.0 中，想要使用 DataSet DataFrame 以及 sql 程序执行的入口是 SparkSession
  // 通过 builder 方法，获取 sparksession 对象
  val session: SparkSession = SparkSession.builder()
    .appName("xxx")
    .master("local")
    .getOrCreate()  // 类似于单例，有就拿来使用，没有再创建

  // 读取数据 调用 read textFile 方法
  val lines: Dataset[String] = session.read.textFile("wc.txt")
  // 收集数据到 driver 端瞅一瞅
  val collected: Array[String] = lines.collect()
  //      println(collected.toBuffer)

  // DataSet 中的数据也是有 schema 信息的
  //      lines.show()
  // 必须导入隐式转换 才能使用 DataSet 的方法
  import session.implicits._
  val words: Dataset[String] = lines.flatMap(_.split(" "))
  // 用 sql 的方式执行
  // 创建一个临时视图（类似于表）
  words.createTempView("t_words")

  val sql: DataFrame = session.sql("select value word,count(*) count from t_words group by word order by count desc")
  sql.show()
```

```
  session.close()
}
```

如果缺少隐式转换，就会报错：



DataSet DSL 风格的 WordCount

```
  def main(args: Array[String]): Unit = {
    // 在 spark2.0 中，想要使用 DataSet DataFrame 以及 sql 程序执行的入口是 SparkSession
    // 通过 builder 方法，获取 sparksession 对象
    val session: SparkSession = SparkSession.builder()
      .appName("xxx")
      .master("local")
      .getOrCreate()  // 类似于单例，有就拿来使用，没有再创建

    // 读取数据 调用 read textFile 方法
    val lines: Dataset[String] = session.read.textFile("wc.txt")

    // 收集数据到 driver 端瞅一瞅
    val collected: Array[String] = lines.collect()
    //      println(collected.toBuffer)
    // DataSet 中的数据也是有 schema 信息的
    //      lines.show()

    // 必须导入隐式转换 才能使用 DataSet 的方法
    import session.implicits._
    val words: Dataset[String] = lines.flatMap(_.split(" "))

    // 不需要再 进行单词和 1 的组装了，因为操作的是一张表，直接分组后调用聚合函数（实际上是一种表达式，spark 会默认转化为方法）
    // 此时必须导入默认的函数
    import org.apache.spark.sql.functions._
    val res: DataFrame = words.groupBy($"value").agg(count("*"))
    // 默认的列名是 value|count(1) ，可类似于 sql 中的指定别名   还可通过 sort 方法进行排序
   val res2:DataFrame =   words.groupBy($"value" as "word").agg(count("*") as "counts").sort($"counts" desc)
    // 分组后求组内 count  使用 count 方法一样可以实现
    val res3: DataFrame = words.groupBy($"value" as "word").count() // .sort($"count" desc)

    // 继续对 res3 调用 count 方法  返回的结果 相当于数据表中的行数
     val res4: Long = res3.count()
```

```
    // 重命名列名
    val sort: Dataset[Row] = words.groupBy($"value" as
"word").count().withColumnRenamed("count","countss").sort($"countss" desc)
    sort.show()
    // words.show()
    // 展示结果
//    res3.show()
//    println(s"counts = $res4")
    session.close()
  }
```

1.8 RDD、DataFrame 和 DataSet 的区别

DataFrame 和 DataSet 可以转化为 RDD 类型

```
// DataFrame Dataset  都可以通过.rdd 方法转化为 rdd
val rdd: RDD[Row] = personDF.rdd
```

1.8.1 RDD 的优点与缺点

优点：

编译时类型安全：编译时就能检查出类型错误

面向对象的编程风格：直接通过类名点的方式来操作数据

缺点：

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

GC 的性能开销：频繁的创建和销毁对象，势必会增加 GC

1.8.2 DataFrame 的优点与缺点

DataFrame 引入了 schema 和 off-heap

schema : RDD 每一行的数据, 结构都是一样的. 这个结构就存储在 schema 中. 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 : 意味着 JVM 堆以外的内存, 这些内存直接受操作系统管理（而不是 JVM）。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. 由于 Spark 理解 schema, 所以知道该如何操作

优点：通过 schema 和 off-heap, DataFrame 解决了 RDD 的缺点，DataFrame 不受 JVM 的限制，没有 GC 的困扰

缺点：DataFrame 不是类型安全的, API 也不是面向对象风格的

1.8.3 DataSet 的优点

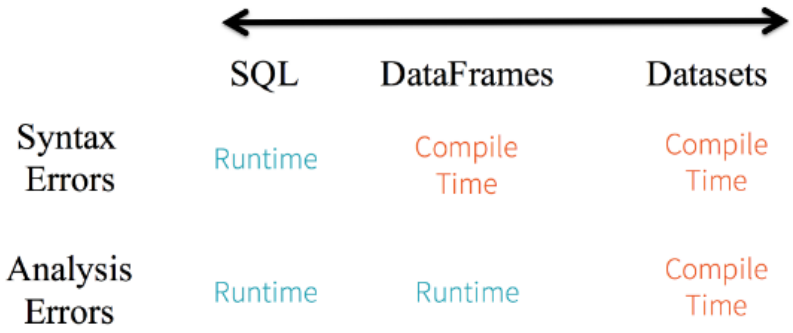
DataSet 结合了 RDD 和 DataFrame 的优点（类型安全、面向对象、不受 JVM 限制、没有 GC 开销）

1.8.3.1 静态类型与运行时类型安全

如果你用的是 Spark SQL 的查询语句，要直到运行时才会发现有语法错误（这样做代价很大），而如果你用的是 DataFrame 和 Dataset，你在编译时就可以捕获错误（这样就节省了开发者的时间和整体代价）。也就是说，当你在 DataFrame 中调用了 API 之外的函数时，编译器就可以发现这个错。不过，如果你使用了一个不存在的字段名字，那就要到运行时才能发现错误了。

因为 Dataset API 都是用 lambda 函数和 JVM 类型对象表示的，所有不匹配的类型参数都可以在编译时发现。而且在使用 Dataset 时，你的分析错误也会在编译时被发现，这样就节省了开发者的时间和代价。

所有这些最终都被解释成关于类型安全的图谱，内容就是你的 Spark 代码里的语法和分析错误。在图谱中，Dataset 是最严格的一端，但对于开发者来说也是效率最高的。



1.8.3.2 方便易用的结构化 API

虽然结构化可能会限制 Spark 程序对数据的控制，但它却提供了丰富的语义，和方便易用的特定领域内的操作，后者可以被表示为高级结构。事实上，用 Dataset 的高级 API 可以完成大多数的计算。比如，比用 RDD 数据行的数据字段进行 agg、select、sum、avg、map、filter 或 groupBy 等操作更简单。

1.8.3.3 性能与优化

首先，因为 DataFrame 和 Dataset API 都是基于 Spark SQL 引擎构建的，它使用 Catalyst 来生成优化后的逻辑和物理查询计划。所有 R、Java、Scala 或 Python 的 DataFrame/Dataset API，所有的关系型查询的底层使用的都是相同的代码优化器，因而会获得空间和速度上的效率。尽管有类型的 Dataset[T] API 是对数据处理任务优化过的，无类型的 Dataset[Row]（别名 DataFrame）却运行得更快，适合交互式分析。

1.9 多表关联 join

```
    // 获取到一个 sparksession 对象
 val session = SparkSession.builder()
  .master("local[*]")
  .appName(JoinDemo.getClass.getSimpleName)
  .getOrCreate()
import session.implicits._

// 多表的关联查询
/** 第一个表
  * name age fv address
  *
  * 第二个:
  * address  aname
  * hn  河南省
  *
  */
val data1: Dataset[String] = session.createDataset(List("dd 25 100 hn", "nn 26 100 sd",
"tt 18 100 gz", "lz 30 100 heb", "cangls 40 100 jp"))

val data2: Dataset[String] = session.createDataset(List("hn 河南省", "sd 山东省", "heb
黑龙江省", "jp 日本省"))
// 数据的切分
val splitData1: Dataset[(String, Int, Int, String)] = data1.map({
  t =>
    val lines = t.split(" ")
    val name = lines(0)
    val age = lines(1).toInt
    val fv = lines(2).toInt
    val pro = lines(3)
    (name, age, fv, pro)
})

// 指定 shema
val df1: DataFrame = splitData1.toDF("name", "age", "fv", "pro")

val splitData2: Dataset[(String, String)] = data2.map {
  t =>
    val lines = t.split(" ")
    (lines(0), lines(1))
}
val df2: DataFrame = splitData2.toDF("pro1", "pname")
```

```
//    要读两个数据进行join1.10
// SQL 语法
// 注册两张表，然后进行join 关联查询
//    df1.createTempView("t_pinfo")
//    df2.createTempView("t_proinfo")

//    session.sql("select * from t_pinfo inner join t_proinfo on t_pinfo.pro =
t_proinfo.pro ")
//    session.sql("select * from t_pinfo right join t_proinfo on t_pinfo.pro =
t_proinfo.pro ")
//    .show()


// DSL 语法风格
// 先进行join  然后再通过where 指定join 的条件
//    df1.join(df2).where(df1("pro") === df2("pro"))

// 如果有相同的字段，可以直接使用该字段名称
//    df1.join(df2,"pro")

// 默认使用的是inner join  我们可以使用自定的
//    inner`, `cross`, `outer`, `full`, `full_outer`, `left`, `left_outer`,
//    *                `right`, `right_outer`, `left_semi`, `left_anti`.
//      df1.join(df2,df1("pro") === df2("pro1"))

// 利用第3 个参数来指定 join 的类型
df1.join(df2, $"pro" === $"pro1", "left")
  .show()
```

案例：求 ip 地址归属地，利用 sparksql 的多表关联实现

iprules: 注册成一张表
accesslog