# 2019

# 企业级数据仓库实战

# Spark SQL

学习资料：

Spark SQL基础：http://spark.apache.org/docs/latest/sql-programming-guide.html

Scala基础：https://www.runoob.com/scala/scala-tutorial.html

SparkSQL语法：http://spark.apache.org/docs/latest/api/sql/index.html

主要内容：

1、SparkSession VS. SparkContext VS. SQLContext VS. HiveContext

2、DataFrame VS. DataSet VS. RDD

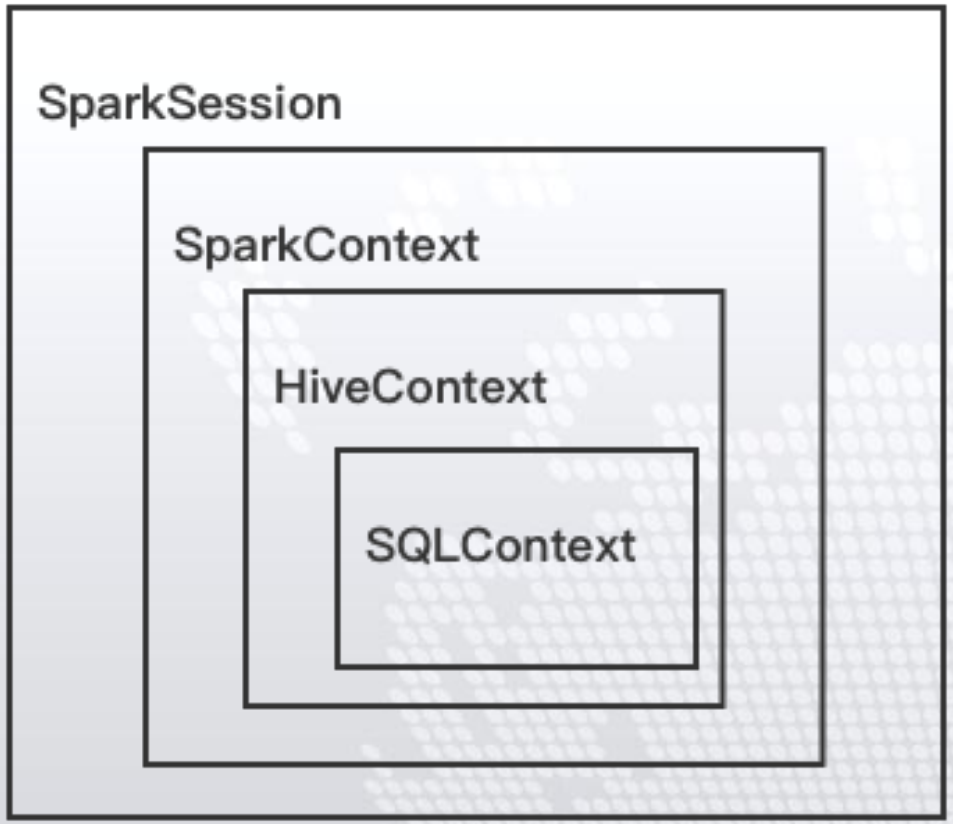
3、如何创建DataFrame

4、如何生成DataSet

5、如何加载和保存数据

6、总结

# SparkSession VS. SparkContext VS. SQLContext VS. HiveContext




Spark 1.6 => SparkContext,SQLContext,HiveContext

```
val conf = new SparkConf().setMaster(master).setAppName(appName)
val sc = new SparkContext(conf)
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

Spark 2.0 => SparkSession

```
val ss = SparkSession.builder().master(master)
        .appName(appName)
        .getOrCreate()
```

# SparkSession VS. SparkContext VS. SQLContext VS. HiveContext

## SparkContext

（1）驱动程序使用SparkContext与集群连接并建立通信

（2）通过SparkContext，可以使用SQLContext和HiveContext

（3）使用SparkContext，可以针对Spark作业配置相应的参数

备注：spark-shell中系统已经存在了一个SparkContext并分配给了sc.正常编程时需要通过如下语句创建一个SparkContext

```
//set up the spark configuration
val sparkConf = new SparkConf().setAppName(“appName”).setMaster("local")
//get SparkContext using the SparkConf
val sc = new SparkContext(sparkConf)
```

其中sparkConf为对应的配置项，用来设置运行主机、应用名等，而SparkContext有多个构造函数，这里我们使用的是截图中的构造函数

**Instance Constructors**

new **SparkContext**(master: String, appName: String, sparkHome: String = null, jars: Seq[String] = Nil, environment: Map[String, String] = Map())

Alternative constructor that allows setting common Spark properties directly

new **SparkContext**(master: String, appName: String, conf: SparkConf)

Alternative constructor that allows setting common Spark properties directly

new **SparkContext**()

Create a SparkContext that loads settings from system properties (for instance, when launching with ./bin/spark-submit).

new **SparkContext**(config: SparkConf)

**config** a Spark Config object describing the application configuration. Any settings in this config overrides the default configs as well as system properties.

# SparkSession VS. SparkContext VS. SQLContext VS. HiveContext

## SQLContext

SQLContext是使用SparkSQL的入口，使用上一步创建的SparkContext创建SparkSQL

备注：编程时需要通过如下语句创建一个SQLContext

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

注意此方法已经被SparkSession替代

```
class SQLContext extends Logging with Serializable
```

The entry point for working with structured data (rows and columns) in Spark 1.x.

As of Spark 2.0, this is replaced by SparkSession. However, we are keeping the class here for backward compatibility.

| | |
|---|---|
| *Self Type* | SQLContext |
| *Annotations* | @Stable() |
| *Source* | SQLContext.scala |
| *Since* | 1.0.0 |

▶ Linear Supertypes

▶ Known Subclasses

# SparkSession VS. SparkContext VS. SQLContext VS. HiveContext

## HiveContext

HiveContext是使用Hive的入口，HiveContext具有SQLContext的全部功能，即HiveContext是SQLContext的超集

备注：编程时需要通过如下语句创建一个SQLContext

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

注意此方法已经被SparkSession.builder.enableHiveSupport 替代

# SparkSession VS. SparkContext VS. SQLContext VS. HiveContext

## SparkSession

SparkSession是Spark 2.0后引入的，通过访问SparkSession，我们可以自动访问SparkContext，创建SparkSession的方法如下：

```
val spark = SparkSession
            .builder()
            .appName("appName")
            .config("spark.some.config.option", "some-value")
            .getOrCreate()
```

SparkSession是Spark的新入口，取代了旧的SQLContext和HiveContext.请注意，保留旧的SQLCONTEXT和HiveContext是为了向后兼容。一旦可以访问SparkSession，就可以开始使用DataFrame和Dataset。

如果想要使用Hive，可以通过如下方式进行创建

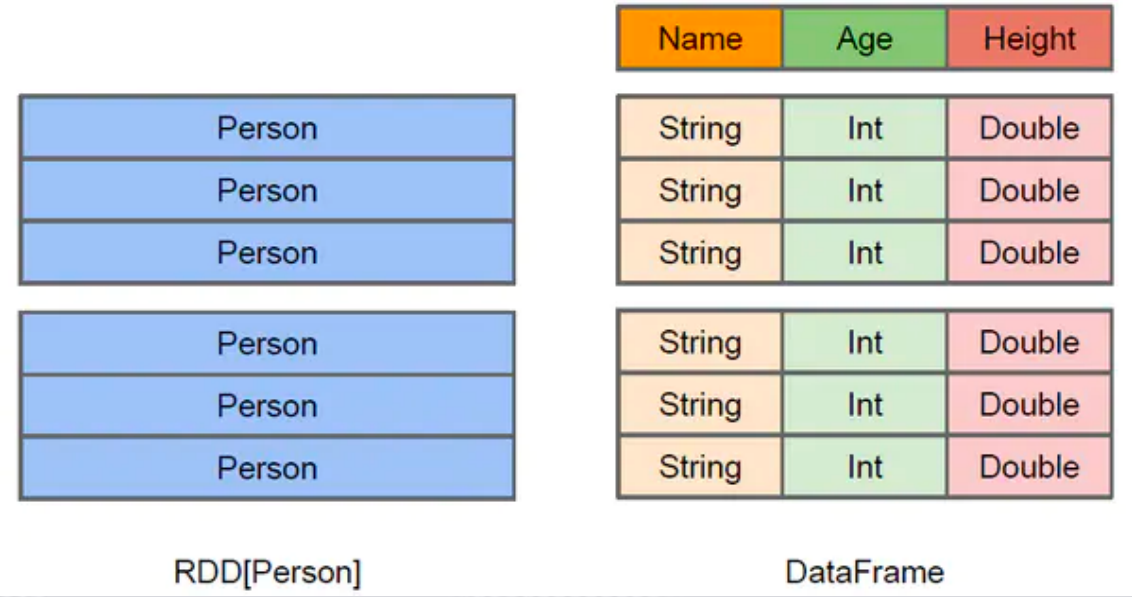
```
val spark = SparkSession
            .builder()
            .appName("appName")
            .config("spark.sql.warehouse.dir", warehouseLocation)
           .enableHiveSupport()
           .getOrCreate()
```

# DataFrame VS. DataSet VS. RDD

**RDD**：RDD代表弹性分布式数据集，它记录的是只读分区集合。RDD是Spark的基本数据结构，它允许程序员以容错的方式在大型集群的内存中进行计算

**DataFrame**：与RDD类似，DataFrame也是一个不可变的分布式数据集合，两者不同点在于，DataFrame中包含数据的Schema信息，从而可以进行更高级别的抽象，简而言之，DataFrame=RDD+Schema

**DataSet**：是DataFrame的父类，当DataSet中存储Row时，两者等价DataSet[Row]=DataFrame
其中Row是一个类型，跟Car、Person这些类型一样，所有的表结构信息都用Row来表示。



# Spark程序的入口：SparkSession

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

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

# 如何创建DataFrame

## 方法一：加载一个数据源

常见的数据源有Parquet文件、ORC文件、JSON文件、JSON文件、AVRO文件、Hive表、JDBC连接数据库等

## 方法二：通过一个RDD进行转换

Spark SQL有两种方法将RDD转为DataFrame。

**1. 使用反射机制**

*Spark SQL*的*Scala*接口支持自动将包含*case class*对象的*RDD*转为*DataFrame*。
对应的*case class*定义了表的*schema*。*case class*的参数名通过反射，映射为表的字段名。
*case class*还可以嵌套一些复杂类型，如*Seq*和*Array*。
*RDD*隐式转换成*DataFrame*后，可以进一步注册成表。
随后，你就可以对表中数据使用*SQL*语句查询了。

**2. 编程方式构建一个schema**

如果不能事先通过*case class*定义*schema*（例如，记录的字段结构是保存在一个字符串，或者其他文本数据集中，需要先解析，又或者字段对不同用户有所不同），那么你可能需要按以下三个步骤，以编程方式的创建一个*DataFrame*：

- 从已有的*RDD*创建一个包含*Row*对象的*RDD*
- 用*StructType*创建一个*schema*，和步骤*1*中创建的*RDD*的结构相匹配
- 把得到的*schema*应用于包含*Row*对象的*RDD*，调用这个方法来实现这一步：*SparkSession.createDataFrame*

# 通过加载数据源创建DataFrame

常见的数据源有Parquet文件、ORC文件、JSON文件、JSON文件、AVRO文件、Hive表、JDBC连接数据库等

```
//Parquet文件
val usersDF = sparkSession.read.load(“/spark/resources/users.parquet")
//JSON文件
val peopleDF=sparkSession.read.json(“/spark/resources/people.json")
//AVRO文件
val usersDF = sparkSession.read.format(“avro").load("/spark/resources/users.avro")
//Hive表
val courseDF = sparkSession.sql("select * from dw.course”)
// JDBC
val jdbcDF = spark.read .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)
```

# 通过RDD进行转换生成DataFrame

**使用反射机制**

```
case class People(name: String, age: Long)
//为了支持RDD到DataFrame的隐式转换
import sparkSession.implicits._
//创建一个包含Person对象的RDD
val peopleDF = sparkSession.sparkContext
  .textFile("/spark/resources/people.txt")
  .map(_.split(","))
  .map(attributes => People(attributes(0), attributes(1).trim.toInt))
  .toDF()
//将其注册成table
peopleDF.createOrReplaceTempView("people")
// sqlContext.sql方法可以直接执行SQL语句
val teenagersDF = sparkSession.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")
// SQL查询的返回结果是一个DataFrame，且能够支持所有常见的RDD算子
//查询结果中每行的字段可以按字段索引访问:
teenagersDF.map(teenager => "Name: " + teenager(0)).show()
//或者按字段名访问:
teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show()
```

# 通过RDD进行转换生成DataFrame

**编程方式构建一个schema**

```
//为了支持RDD到DataFrame的隐式转换
import sparkSession.implicits._
//创建一个RDD
val peopleRDD = sparkSession.sparkContext.textFile("/spark/resources/people.txt")
//数据的Schema被编码在一个字符串中
val schemaString = "name age"
//生成Schema
val fields = schemaString.split(" ")
  .map(fieldName => StructField(fieldName, StringType, nullable = true))
val schema = StructType(fields)
//将RDD[people]的各个记录转换为Rows，即：得到一个包含Row对象的RDD
val rowRDD = peopleRDD.map(_.split(","))
  .map(attributes => Row(attributes(0), attributes(1).trim))
//将schema应用到包含Row对象的RDD上，得到一个DataFrame
val peopleDF = sparkSession.createDataFrame(rowRDD, schema)
//将DataFrame注册为table
peopleDF.createOrReplaceTempView("people")
//执行SQL语句
val results = sparkSession.sql("SELECT name FROM people")
//并且其字段可以以索引访问，也可以用字段名访问
results.map(attributes => "Name: " + attributes(0)).show()
```

# 如何生成DataSet

```
case class Person(name:String,age:Long)
import sparkSession.implicits._
//对普通类型数据的Encoder是由 importing sqlContext.implicits._ 自动提供的
val caseClassDS = Seq(Person("Andy",32)).toDS()
caseClassDS.show()
val primitivesDS =  Seq(1,2,3).toDS()
primitivesDS.map(_+1).show()
//注意这里默认使用的HDFS路径
// DataFrame 只需提供一个和数据schema对应的class即可转换为 Dataset。Spark会根据字段名进行映射。
val hdfs_path = "/spark/resources/people.json"
val peopleDS = sparkSession.read.json(hdfs_path).as[Person]
peopleDS.show()
```

# 如何生成临时视图和全局视图

```
//加载json文件 生成DataFrame
val df=sparkSession.read.json("/spark/resources/people.json")
//打印DataFrame的内容
df.show()
//将DataFrame注册成一个临时的视图
df.createOrReplaceTempView("people")
//使用SQL语句进行操作
sparkSession.sql("SELECT age,name FROM people").show()
//将DataFrame注册全局临时视图
df.createGlobalTempView("people")
//访问全局临时视图需要指定global_temp数据库
sparkSession.sql("SELECT * FROM global_temp.people").show()
//全局临时视图与临时视图的区别在于前者是跨session的
sparkSession.newSession().sql("SELECT * FROM global_temp.people").show()
```

# 加载与保存

```
import spark.implicits._
//加载一个parquet文件
val usersDF = spark.read.load("/spark/resources/users.parquet")
//以追加的方式写入磁盘
usersDF.select("name", "favorite_color").write
  .mode(SaveMode.Append)
  .save("/LoadSave/20200512/usersDF/namesAndFavColors.parquet")
//使用saveAsTable方式写入Hive
peopleDF.write
  .format("Hive")
  .mode(SaveMode.Append)
  .saveAsTable("people_bucketed")
```

备注：

_SUCCESS文件：空文件，是一个标识文件

# 加载与保存

其中SaveMode主要有如下集中方式：

| Scala/Java | Any Language | Meaning |
| --- | --- | --- |
| `SaveMode.ErrorIfExists` (default) | `"error"` or `"errorifexists"` (default) | When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown. |
| `SaveMode.Append` | `"append"` | When saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data. |
| `SaveMode.Overwrite` | `"overwrite"` | Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame. |
| `SaveMode.Ignore` | `"ignore"` | Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected not to save the contents of the DataFrame and not to change the existing data. This is similar to a `CREATE TABLE IF NOT EXISTS` in SQL. |

# 总结

RDD、DataFrame 、DataSet三者是可以相互转换的，其中

RDD —> DataFrame：使用反射机制或者使用编程的方式构造一个schema

RDD —> DataSet:反射+.toDS

DataFrame —> DataSet：.as[case class]

DataFrame —> RDD：.rdd

DataSet —> RDD：.rdd

DataSet —> DataFrame:.toDF

# 总结

1.如果你想要丰富的语义、高层次的抽象，和特定情景的API，使用DataFrame或DataSet。

2.如果你的处理要求涉及到filters, maps, aggregation, averages, sum, SQL queries, columnar access或其他lambda匿名函数，使用DataFrame或DataSet。

3.如果希望在编译时获得更高的类型安全性，需要类型化的JVM对象，利用Tungsten编码进行高效的序列化、反序列化，使用DataSet。

4.如果你想统一和简化spark的API，使用DataFrame或DataSet。

5.如果你是一个R用户，使用DataFrame。

6.如果你是一个Python用户，使用DataFrame，如果你需要更多的控制功能，尽量回到RDD。

THANK YOU FOR YOUR GUIDANCE.

# 谢谢