亚太研究院

Spark 官方文档翻译

Spark SQL 编程指南

（V1.2.0）

翻译者　韩保礼

Spark 官方文档翻译团成员

前 言

伴随着大数据相关技术和产业的逐步成熟，继 Hadoop 之后，Spark 技术以集大成的无可比拟的优势，发展迅速，将成为替代 Hadoop 的下一代云计算、大数据核心技术。

Spark 是当今大数据领域最活跃最热门的高效大数据通用计算平台，基于 RDD，Spark 成功的构建起了一体化、多元化的大数据处理体系，在“One Stack to rule them all”思想的引领下，Spark 成功的使用 Spark SQL、Spark Streaming、MLLib、GraphX 近乎完美的解决了大数据中 Batch Processing、Streaming Processing、Ad-hoc Query 等三大核心问题，更为美妙的是在 Spark 中 Spark SQL、Spark Streaming、MLLib、GraphX 四大子框架和库之间可以无缝的共享数据和操作，这是当今任何大数据平台都无可匹敌的优势。

在实际的生产环境中，世界上已经出现很多一千个以上节点的 Spark 集群，以 eBay 为例，eBay 的 Spark 集群节点已经超过 2000 个，Yahoo！等公司也在大规模的使用 Spark，国内的淘宝、腾讯、百度、网易、京东、华为、大众点评、优酷土豆等也在生产环境下深度使用 Spark。2014 Spark Summit 上的信息，Spark 已经获得世界 20 家顶级公司的支持，这些公司中包括 Intel、IBM 等，同时更重要的是包括了最大的四个 Hadoop 发行商，都提供了对 Spark 非常强有力的支持。

与 Spark 火爆程度形成鲜明对比的是 Spark 人才的严重稀缺，这一情况在中国尤其严重，这种人才的稀缺，一方面是由于 Spark 技术在 2013、2014 年才在国内的一些大型企业里面被逐步应用，另一方面是由于匮乏 Spark 相关的中文资料和系统化的培训。为此，Spark 亚太研究院和 51CTO 联合推出了“Spark 亚太研究院决胜大数据时代 100 期公益大讲堂”，来推动 Spark 技术在国内的普及及落地。

具体视频信息请参考 http://edu.51cto.com/course/course_id-1659.html

与此同时，为了向 Spark 学习者提供更为丰富的学习资料，Spark 亚太研究院去年 8 月发起并号召，结合网络社区的力量构建了 Spark 中文文档专家翻译团队，翻译了 Spark 中文文档 V1.1.0 版本。2014 年 12 月，Spark 官方团队发布了 Spark 1.2.0 版本，为了让学习者了解到最新的内容，Spark 中文文档专家翻译团队又对 Spark 1.2.0 版本进行了部分更新，在此，我谨代表 Spark 亚太研究院及广大 Spark 学习爱好者向专家翻译团队所有成员热情而专业的工作致以深刻的敬意！

当然，作为相对系统的 Spark 中文文档，不足之处在所难免，大家有任何建议或者意见都可以发邮件到 marketing@sparkinchina.com；同时如果您想加入 Spark 中文文档翻译团队，也请发邮件到 marketing@sparkinchina.com 进行申请；Spark 中文

文档的翻译是一个持续更新的、不断版本迭代的过程，我们会尽全力给大家提供更高质量的 Spark 中文文档翻译。

最后，也是最重要的，请允许我荣幸的介绍一下我们的 Spark 中文文档 1.2.0 版本翻译的专家团队成员，他们分别是（排名不分先后）：

- 傅智勇, 《快速开始(v1.2.0)》
- 王宇舟,《Spark 机器学习库 (v1.2.0)》
- 武扬，《在 Yarn 上运行 Spark (v1.2.0)》《Spark 调优(v1.2.0)》
- 徐骄，《Spark 配置(v1.2.0)》《Spark 作业调度(v1.2.0)》
- 蔡立宇，《Bagel 编程指南(v1.2.0)》
- harli，《Spark 编程指南 (v1.2.0)》
- 韩保礼，《Spark SQL 编程指南(v1.2.0)》
- 李丹丹，《文档首页(v1.2.0)》
- 李军，《Spark 实时流处理编程指南(v1.2.0)》
- 俞杭军，《使用 Maven 编译 Spark(v1.2.0)》
- 王之，《给 Spark 提交代码(v1.2.0)》
- Ernest ,《集群模式概览(v1.2.0)》《监控与相关工具(v1.2.0)》《提交应用程序(v1.2.0)》

Life is short, You need Spark!

Spark 亚太研究院院长 王家林
2015 年 2 月

Spark SQL 编程指南（V1.2.0）

（翻译者：韩保礼）

Spark SQL Programming Guide，原文档链接：

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

目录

第 1 章 Spark SQL 概述

1.1Scala 版

Spark SQL 允许在 Spark 中执行以 SQL、HiveQL 或 Scala 表示的关系型查询语句。此组件的核心是一种新型的 RDD :SchemaRDD。SchemaRDDs 由行对象(Row objects)组成，同时还有一个描述每行中列元素数据类型的模式（schema）。一个 SchemaRDD 类似于一个传统关系型数据库中的表。一个 SchemaRDD 可以从现有的 RDD、Parquet 文件，JSON 数据集，或运行 HiveQL 以处理存储在 Apache Hive 中的数据中创建。

本文中所有例子使用的样本数据都能够在 spark-shell 中运行。

目前，Spark SQL 还是 alpha 版本。但我们会尽量减少 API 的变化，一些 API 可能会在将来的版本中有所变化。

1.2Java 版

Spark SQL 允许在 Spark 中执行以 SQL、HiveQL 表示的关系型查询语句。此组件的核心是一种新型的 RDD :JavaSchemaRDD。JavaSchemaRDDs 由行对象(Row objects)组成，同时还有一个描述每行中列元素数据类型的模式(schema)。一个 JavaSchemaRDD 类似于一个传统关系型数据库中的表。一个 JavaSchemaRDD 可以从现有的 RDD、Parquet 文件，JSON 数据集，或运行 HiveQL 以处理存储在 Apache Hive 中的数据中创建。

目前，Spark SQL 还是 alpha 版本。但我们会尽量减少 API 的变化，一些 API 可能会在将来的版本中有所变化。

1.3Python 版

Spark SQL 允许在 Spark 中执行以 SQL、HiveQL 表示的关系型查询语句。此组件的核心是一种新型的 RDD：SchemaRDD。JavaSchemaRDDs 由行对象（Row objects）组成，同时还有一个描述每行中列元素数据类型的模式（schema）。一个 SchemaRDD 类似于一个传统关系型数据库中的表。一个 SchemaRDD 可以从现有的 RDD、Parquet 文件，JSON 数据集，或运行 HiveQL 以处理存储在 Apache Hive 中的数据中创建。

本文中所有例子使用的样本数据都能够在 pyspark shell 中运行。

目前，Spark SQL 还是 alpha 版本。但我们会尽量减少 API 的变化，一些 API 可能会在将来的版本中有所变化。

第 2 章 Spark SQL 入门

不同编程语言创建相应的 SQLContext 示例如下表：

Scala	使用 Spark 所有相关功能的入口点是：SQLContext 类，或者它的子类。为了创建一个 SQLContext 基类，你首先需要的是一个 SparkContext。
	`val sc: SparkContext // An existing SparkContext.`
	`val sqlContext = new org.apache.spark.sql.SQLContext(sc)`
	`// createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.`
	`import sqlContext.createSchemaRDD`

Java	使用 Spark 所有相关功能的入口点是：JavaSQLContext 类，或者它的子类。为了创建一个 JavaSQLContext 基类，你首先需要的是一个 JavaSparkContext。
	`JavaSparkContext sc = ...; // An existing JavaSparkContext.`
	`JavaSQLContext sqlContext = new org.apache.spark.sql.api.java.JavaSQLContext(sc);`

Python	使用 Spark 所有相关功能的入口点是：SQLContext 类，或者它的子类。为了创建一个 SQLContext 基类，你首先需要的是一个 SparkContext。
	`from pyspark.sql import SQLContext`
	`sqlContext = SQLContext(sc)`

除了 SQLContext 基类，你还可以创建一个 HiveContext，它提供了涵盖 SQLContext 功能的一个超集。附加功能包括：使用更完整的 HiveQL 解析器编写查询语句、访问 HiveUDFs、从 Hive 表中读取数据。要使用 HiveContext，你不需要有一个现有的 Hive 设置，所有 SQLContext 可访问的数据源都是可用的。

我们对 HiveContext 进行了单独打包，以避免在 Spark 的默认版本中添加对 Hive 的依赖。如果这些依赖关系对你的应用程序没有影响，那么推荐在 Spark 1.2 中使用 HiveContext。未来版本将着力使 SQLContext 拥有现有 HiveContext 的功能。

用于查询解析的 SQL 特定变体（specific varient）也可以选择使用 spark.sql.dialect 选项。你可以利用 SQLContext 中 setConf 方法或在 SQL 中使用 SET key=value 的命令，来改变这个参数。对于 SQLContext，唯一可用的方言是“sql”，它采用了由 Spark SQL 提供的一个简单 SQL 解析器。对于 HiveContext，默认为“hiveql”，但“sql”也是可以的。由于 HiveQL 解析器更完整，建议大多数情况下使用“hiveql”。

第 3 章数据源

Spark SQL 通过 SchemaRDD 接口，支持不同数据源的操作。SchemaRDD 可以像普通 RDDs 一样运行，也可以被注册为一张临时表。将 SchemaRDD 注册为一张表，可以让你在它的数据之上运行 SQL 查询。本章主要介绍将数据加载到 SchemaRDD 的几种方法。

3.1.RDDs

Spark SQL 支持两种不同的方法将现有的 RDDS 转变成 SchemaRDDs。第一种方法使用反射（reflection）来推断包含特定类型对象的 RDD 的格式。这种基于反射方法使得代码更简洁且运行良好，因为当你在写 Spark 应用的时候，你早已经知道了它的格式 。

创建 SchemaRDDs 的第二种方法是通过一个编程接口——允许你构建一种格式 然后将其应用到现有的 RDD。虽然这种方法比较繁琐，但可以让你在不知道 RDD 的列和它们的类型时构建 SchemaRDDs。

3.1.1.使用反射推断模式

Scala 版

Scala 接口支持自动将含有样本类（case classes）的 RDD 转变为 SchemaRDD。样本类定义了表的模式（schema）。样本类中的参数名通过反射被读取，然后转变为列的名称。样本类也可以嵌套或包含复杂类型，如序列或数组。这种 RDD 可以隐式转换为 SchemaRDD，然后注册为一个表。这张表可以在后续的 SQL 语句中使用。示例代码如下表：

```
// sc is an existing SparkContext.

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.

import sqlContext.createSchemaRDD


// Define the schema using a case class.

// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work a
round this limit,

// you can use custom classes that implement the Product interface.

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

```
// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt))
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are SchemaRDDs and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

Java 版

Spark SQL 支持自动将含有 JavaBeans 的 RDD 转变成 SchemaRDD。BeanInfo 中定义了表的模式，并由反射读取。目前，Spark SQL 不支持包含嵌套或复杂类型（如列表或数组）的 JavaBeans。你可以创建一个 JavaBean，其实现 Serializable 接口，并具有所有属性的 getter 和 setter 方法。如下表：

```
public static class Person implements Serializable {
  private String name;
  private int age;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
```

```
  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }
}
```

通过调用 applySchema 方法将模式应用到现有的 RDD，并为 JavaBean 提供 Class 对象，代码如下表：

```
// sc is an existing JavaSparkContext.
JavaSQLContext sqlContext = new org.apache.spark.sql.api.java.JavaSQLContext(sc);

// Load a text file and convert each line to a JavaBean.
JavaRDD<Person> people = sc.textFile("examples/src/main/resources/people.txt").map(
  new Function<String, Person>() {
    public Person call(String line) throws Exception {
      String[] parts = line.split(",");

      Person person = new Person();
      person.setName(parts[0]);
      person.setAge(Integer.parseInt(parts[1].trim()));

      return person;
    }
  });

// Apply a schema to an RDD of JavaBeans and register it as a table.
```

```
JavaSchemaRDD schemaPeople = sqlContext.applySchema(people, Person.class);

schemaPeople.registerTempTable("people");


// SQL can be run over RDDs that have been registered as tables.

JavaSchemaRDD teenagers = sqlContext.sql("SELECT name FROM people WHERE age >=
 13 AND age <= 19")


// The results of SQL queries are SchemaRDDs and support all the normal RDD op
erations.

// The columns of a row in the result can be accessed by ordinal.

List<String> teenagerNames = teenagers.map(new Function<Row, String>() {

  public String call(Row row) {

    return "Name: " + row.getString(0);

  }

}).collect();
```

Python 版

Spark SQL 可以将含有行对象的 RDD 转变为 SchemaRDD，并推断数据类型。将键/值对列表作为参数(kwargs)传递给 Row 类进行构造行。此列表的键定义了表的列名，其类型通过查阅第一行进行推断得到。由于目前我们只查阅第一行数据，因此 RDD 的第一行不能有丢失的数据。在未来的版本中，我们计划通过查阅更多的数据，进行更完善的模式推断，类似于在 JSON 文件之上的推断。示例代码如下表：

```
# sc is an existing SparkContext.

from pyspark.sql import SQLContext, Row

sqlContext = SQLContext(sc)


# Load a text file and convert each line to a Row.

lines = sc.textFile("examples/src/main/resources/people.txt")

parts = lines.map(lambda l: l.split(","))

people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))
```

```
# Infer the schema, and register the SchemaRDD as a table.

schemaPeople = sqlContext.inferSchema(people)

schemaPeople.registerTempTable("people")



# SQL can be run over SchemaRDDs that have been registered as a table.

teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")



# The results of SQL queries are RDDs and support all the normal RDD operations.

teenNames = teenagers.map(lambda p: "Name: " + p.name)

for teenName in teenNames.collect():

  print teenName
```

3.1.2.以编程的方式指定模式

Scala 版

当样本类不能提前确定时（例如，当记录的结构由字符串或文本数据集编码而成，它在解析时，字段将会对不同的用户有不同的投影结果），SchemaRDD 可以由以下三个步骤创建：

1. 从原始 RDD 创建一个含有 Rows 的 RDD;
2. 创建一个由 StructType 表示的模式，它与第 1 步中创建的 RDD 的 Rows 结构相一致。
3. 通过调用 SQLContext 中 applySchema 方法将模式应用到含有 Rows 的 RDD。

例如：

```
// sc is an existing SparkContext.

val sqlContext = new org.apache.spark.sql.SQLContext(sc)



// Create an RDD

val people = sc.textFile("examples/src/main/resources/people.txt")



// The schema is encoded in a string

val schemaString = "name age"
```

```scala
// Import Spark SQL data types and Row.
import org.apache.spark.sql._

// Generate the schema based on the string of schema
val schema =
  StructType(
    schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, true)))

// Convert records of the RDD (people) to Rows.
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))

// Apply the schema to the RDD.
val peopleSchemaRDD = sqlContext.applySchema(rowRDD, schema)

// Register the SchemaRDD as a table.
peopleSchemaRDD.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val results = sqlContext.sql("SELECT name FROM people")

// The results of SQL queries are SchemaRDDs and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
results.map(t => "Name: " + t(0)).collect().foreach(println)
```

Java 版

当 JavaBean 类不能提前确定时(例如 ,当记录的结构由字符串或文本数据集编码而成，它在解析时，字段将会对不同的用户有不同的投影结果)，SchemaRDD 可以由以下三个步骤创建：

1. 从原始 RDD 创建一个含有 Rows 的 RDD;
2. 创建一个由 StructType 表示的模式，它与第 1 步中创建的 RDD 的 Rows 结构相一致。
3. 通过调用 JavaSQLContext 中 applySchema 方法将模式应用到含有 Rows 的 RDD。

例如：

```
// Import factory methods provided by DataType.
import org.apache.spark.sql.api.java.DataType
// Import StructType and StructField
import org.apache.spark.sql.api.java.StructType
import org.apache.spark.sql.api.java.StructField
// Import Row.
import org.apache.spark.sql.api.java.Row

// sc is an existing JavaSparkContext.
JavaSQLContext sqlContext = new org.apache.spark.sql.api.java.JavaSQLContext(sc);

// Load a text file and convert each line to a JavaBean.
JavaRDD<String> people = sc.textFile("examples/src/main/resources/people.txt");

// The schema is encoded in a string
String schemaString = "name age";

// Generate the schema based on the string of schema
List<StructField> fields = new ArrayList<StructField>();
for (String fieldName: schemaString.split(" ")) {
  fields.add(DataType.createStructField(fieldName, DataType.StringType, true));
}
StructType schema = DataType.createStructType(fields);

// Convert records of the RDD (people) to Rows.
JavaRDD<Row> rowRDD = people.map(
  new Function<String, Row>() {
    public Row call(String record) throws Exception {
      String[] fields = record.split(",");
      return Row.create(fields[0], fields[1].trim());
```

```
    }

  });


// Apply the schema to the RDD.

JavaSchemaRDD peopleSchemaRDD = sqlContext.applySchema(rowRDD, schema);


// Register the SchemaRDD as a table.

peopleSchemaRDD.registerTempTable("people");


// SQL can be run over RDDs that have been registered as tables.

JavaSchemaRDD results = sqlContext.sql("SELECT name FROM people");


// The results of SQL queries are SchemaRDDs and support all the normal RDD operations.

// The columns of a row in the result can be accessed by ordinal.

List<String> names = results.map(new Function<Row, String>() {

  public String call(Row row) {

    return "Name: " + row.getString(0);

  }

}).collect();
```

Python 版

当 kwargs 的字典不能提前确定时（例如，当记录的结构由字符串或文本数据集编码而成，它在解析时，字段将会对不同的用户有不同的投影结果），SchemaRDD 可以由以下三个步骤创建：

1. 从原始 RDD 创建一个含有元组或列表的 RDD;
2. 创建一个由 StructType 表示的模式，它与第 1 步中创建的 RDD 的元组或列表结构相一致。
3. 通过调用 SQLContext 中 applySchema 方法将模式应用到 RDD。

例如：

```
# Import SQLContext and data types

from pyspark.sql import *


```

```
# sc is an existing SparkContext.
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a tuple.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: (p[0], p[1].strip()))

# The schema is encoded in a string.
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)

# Apply the schema to the RDD.
schemaPeople = sqlContext.applySchema(people, schema)

# Register the SchemaRDD as a table.
schemaPeople.registerTempTable("people")

# SQL can be run over SchemaRDDs that have been registered as a table.
results = sqlContext.sql("SELECT name FROM people")

# The results of SQL queries are RDDs and support all the normal RDD operations.
names = results.map(lambda p: "Name: " + p.name)
for name in names.collect():
  print name
```

3.2.Parquet 文件

Parquet 文件是一种列式存储格式的文件，被很多数据处理系统支持。Spark SQL 支持读取和写入 Parquet 文件，并可自动保留原始数据的格式（schema）。

3.2.1.以编程方式加载数据

Scala 版

使用 3.1.1.1 中示例代码中的数据，示例代码如下表：

```
// sqlContext from the previous example is used in this example.

// createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.

import sqlContext.createSchemaRDD


val people: RDD[Person] = ... // An RDD of case class objects, from the previous example.


// The RDD is implicitly converted to a SchemaRDD by createSchemaRDD, allowing it to be stored
 using Parquet.

people.saveAsParquetFile("people.parquet")


// Read in the parquet file created above.  Parquet files are self-describing so the schema is
 preserved.

// The result of loading a Parquet file is also a SchemaRDD.

val parquetFile = sqlContext.parquetFile("people.parquet")


//Parquet files can also be registered as tables and then used in SQL statements.

parquetFile.registerTempTable("parquetFile")

val teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")

teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

Java 版

使用 3.1.1.2 中示例代码中的数据，示例代码如下表：

```
// sqlContext from the previous example is used in this example.


JavaSchemaRDD schemaPeople = ... // The JavaSchemaRDD from the previous example.
```

```
// JavaSchemaRDDs can be saved as Parquet files, maintaining the schema information.
schemaPeople.saveAsParquetFile("people.parquet");

// Read in the Parquet file created above.  Parquet files are self-describing so the sche
ma is preserved.
// The result of loading a parquet file is also a JavaSchemaRDD.
JavaSchemaRDD parquetFile = sqlContext.parquetFile("people.parquet");

//Parquet files can also be registered as tables and then used in SQL statements.
parquetFile.registerTempTable("parquetFile");
JavaSchemaRDD teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AN
D age <= 19");
List<String> teenagerNames = teenagers.map(new Function<Row, String>() {
  public String call(Row row) {
    return "Name: " + row.getString(0);
  }
}).collect();
```

Python 版

使用 3.1.1.3 中示例代码中的数据，示例代码如下表：

```
# sqlContext from the previous example is used in this example.

schemaPeople # The SchemaRDD from the previous example.

# SchemaRDDs can be saved as Parquet files, maintaining the schema information.
schemaPeople.saveAsParquetFile("people.parquet")

# Read in the Parquet file created above.  Parquet files are self-describing so the schema is
preserved.
# The result of loading a parquet file is also a SchemaRDD.
parquetFile = sqlContext.parquetFile("people.parquet")
```

```
# Parquet files can also be registered as tables and then used in SQL statements.
parquetFile.registerTempTable("parquetFile");
teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenNames = teenagers.map(lambda p: "Name: " + p.name)
for teenName in teenNames.collect():
  print teenName
```

3.2.2.配置

可以用 SQLContext 中 setConf 方法或使用 SQL 运行 SET key=value 命令，来完成 Parquet 配置。Parquet 配置属性如下表：

属性名称	默认值	描述
spark.sql.parquet.binaryAsString	false	一些 Parquet 处理系统，特别是 Impala 和老版本的 Spark SQL，当写出 Parquet 模式时，它们并不区分二进制数据或字符串数据。这个属性告诉 Spark SQL 将二进制数据解释为字符串型，以提供与其他系统的兼容性。
spark.sql.parquet.cacheMetadata	false	启动 Parquet 格式元数据缓存。可以加速静态数据查询
spark.sql.parquet.compression.codec	gzip	写 Parquet 文件时，设置压缩编码的值。可用的值包括：uncompressed，snappy，gzip，lzo。
spark.sql.parquet.filterPushdown	false	打开 Parquet 过滤器下推优化功能。此功能是默认关闭的，因为一个已知的 bug：Paruet1.6.0rc3（PARQUET-136）。但是，如果你的表不包含任何空字符串或二进制列，可安全的打开此功能。
spark.sql.hive.convertMetastoreParquet	true	若设置为 false，Spark SQL 将使用 Hive SerDe 处理 parquet 表，而不是内置功能。

3.3Json 数据集

3.3.1.Scala 版

Spark SQL 可以自动推断出一个 JSON 数据集的架构（schema），并将其加载为 SchemaRDD。这种转换可以利用 SQLContext 提供的两种方法中的一个来完成：

- jsonFile ——从含有 JSON 文件的目录中加载数据。其中，JSON 文件中的每一行均是一个 JSON 对象。
- jsonRdd ——从现有的 RDD 中加载数据。其中，RDD 的每个元素是一个包含一个 JSON 对象的字符串。

注意 :提供作为 jsonFile 的文件并不是典型的 JSON 文件。各行必须包含一个独立的，有效的 JSON 对象。否则， JSON 文件将会加载失败。

示例代码如下表：

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

// A JSON dataset is pointed to by path.
// The path can be either a single text file or a directory storing text files.
val path = "examples/src/main/resources/people.json"
// Create a SchemaRDD from the file(s) pointed to by path
val people = sqlContext.jsonFile(path)

// The inferred schema can be visualized using the printSchema() method.
people.printSchema()
// root
//  |-- age: integer (nullable = true)
//  |-- name: string (nullable = true)

// Register this SchemaRDD as a table.
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
```

```scala
// Alternatively, a SchemaRDD can be created for a JSON dataset represented by
// an RDD[String] storing one JSON object per string.
val anotherPeopleRDD = sc.parallelize(
  """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)
val anotherPeople = sqlContext.jsonRDD(anotherPeopleRDD)
```

3.3.2.Java 版

Spark SQL 可以自动推断出一个 JSON 数据集的架构（schema），并将其加载为 JavaSchemaRDD。这种转换可以利用 JavaSQLContext 提供的两种方法中的一个来完成：

- jsonFile ——从含有 JSON 文件的目录中加载数据。其中，JSON 文件中的每一行均是一个 JSON 对象。
- jsonRdd ——从现有的 RDD 中加载数据。其中，RDD 的每个元素是一个包含一个 JSON 对象的字符串。

注意 :提供作为 jsonFile 的文件并不是典型的 JSON 文件。各行必须包含一个独立的，有效的 JSON 对象。否则， JSON 文件将会加载失败。

示例代码如下表：

```java
// sc is an existing JavaSparkContext.
JavaSQLContext sqlContext = new org.apache.spark.sql.api.java.JavaSQLContext(sc);

// A JSON dataset is pointed to by path.
// The path can be either a single text file or a directory storing text files.
String path = "examples/src/main/resources/people.json";
// Create a JavaSchemaRDD from the file(s) pointed to by path
JavaSchemaRDD people = sqlContext.jsonFile(path);

// The inferred schema can be visualized using the printSchema() method.
people.printSchema();
// root
//  |-- age: integer (nullable = true)
//  |-- name: string (nullable = true)
```

```
// Register this JavaSchemaRDD as a table.
people.registerTempTable("people");

// SQL statements can be run by using the sql methods provided by sqlContext.
JavaSchemaRDD teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19");

// Alternatively, a JavaSchemaRDD can be created for a JSON dataset represented by
// an RDD[String] storing one JSON object per string.
List<String> jsonData = Arrays.asList(
  "{\"name\":\"Yin\",\"address\":{\"city\":\"Columbus\",\"state\":\"Ohio\"}}");
JavaRDD<String> anotherPeopleRDD = sc.parallelize(jsonData);
JavaSchemaRDD anotherPeople = sqlContext.jsonRDD(anotherPeopleRDD);
```

3.3.3.Python 版

Spark SQL 可以自动推断出一个 JSON 数据集的架构（schema），并将其加载为 JavaSchemaRDD。这种转换可以利用 JavaSQLContext 提供的两种方法中的一个来完成：

- jsonFile ——从含有 JSON 文件的目录中加载数据。其中，JSON 文件中的每一行均是一个 JSON 对象。
- jsonRdd ——从现有的 RDD 中加载数据。其中，RDD 的每个元素是一个包含一个 JSON 对象的字符串。

注意 :提供作为 jsonFile 的文件并不是典型的 JSON 文件。各行必须包含一个独立的，有效的 JSON 对象。否则， JSON 文件将会加载失败。

示例代码如下表：

```
# sc is an existing SparkContext.
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

# A JSON dataset is pointed to by path.
# The path can be either a single text file or a directory storing text files.
path = "examples/src/main/resources/people.json"
# Create a SchemaRDD from the file(s) pointed to by path
people = sqlContext.jsonFile(path)
```

```
# The inferred schema can be visualized using the printSchema() method.
people.printSchema()
# root
#  |-- age: integer (nullable = true)
#  |-- name: string (nullable = true)

# Register this SchemaRDD as a table.
people.registerTempTable("people")

# SQL statements can be run by using the sql methods provided by sqlContext.
teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# Alternatively, a SchemaRDD can be created for a JSON dataset represented by
# an RDD[String] storing one JSON object per string.
anotherPeopleRDD = sc.parallelize([
  '{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}'])
anotherPeople = sqlContext.jsonRDD(anotherPeopleRDD)
```

3.4.Hive 表

Spark SQL 还支持读取和写入存储在 Apache Hive 中的数据。然而，由于 Hive 有大量的依赖关系，它并不包括在默认的 Spark 组件中。为了使用 Hive，你必须使用“-Phive ”或“-Phive-thriftserver”构建(build)Spark。这个命令构建了一个包含 Hive 的新的集成 jar 包。需要注意到 Hive 的集成 jar 包必须存在于所有的工作节点上，因为他们需要访问 Hive 的序列化和反序列化库（SerDes），以访问到存储在 Hive 中的数据。

将 hive-site.xml 文件放到 conf/目录下，以完成 Hive 的配置。

3.4.1.Scala 版

当运行 Hive 时，你必须构造一个 HiveContext，它继承于 SQLContext，并增加了对在 MetaStore 中查询表的支持，以及使用 HiveQL 编写查询语句的支持。即使没有一个现有的 Hive 部署，你仍然可以创建一个 HiveContext。当不使用 hive-site.xml 的配置时，上下文自动在当前目录中创建 metastore_db 和 warehouse。示例代码如下表：

```scala
// sc is an existing SparkContext.

val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)



sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")

sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")



// Queries are expressed in HiveQL

sqlContext.sql("FROM src SELECT key, value").collect().foreach(println)
```

3.4.2.Java 版

当运行 Hive 时，你必须构造一个 JavaHiveContext，它继承于 JavaSQLContext，并增加了对在 MetaStore 中查询表的支持，以及使用 HiveQL 编写查询语句的支持。JavaHiveContext 除了提供 sql 方法，也提供了 hql 方法，允许查询语句能够在 HiveQL 中使用。示例代码如下表：

```java
// sc is an existing JavaSparkContext.

JavaHiveContext sqlContext = new org.apache.spark.sql.hive.api.java.HiveContext(sc);



sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)");

sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src
");



// Queries are expressed in HiveQL.

Row[] results = sqlContext.sql("FROM src SELECT key, value").collect();
```

3.4.3.Python 版

当运行 Hive 时，你必须构造一个 HiveContext，它继承于 SQLContext，并增加了对在 MetaStore 中查询表的支持，以及使用 HiveQL 编写查询语句的支持。HiveContext 除了提供 sql 方法，也提供了 hql 方法，允许查询语句能够在 HiveQL 中使用。示例代码如下表：

```python
# sc is an existing SparkContext.

from pyspark.sql import HiveContext
```

```
sqlContext = HiveContext(sc)



sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")

sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")



# Queries can be expressed in HiveQL.

results = sqlContext.sql("FROM src SELECT key, value").collect()
```

第 4 章性能调优

对于某些工作负荷，我们可以通过将数据缓存到内存中，或改变一些实验参数，来改善 Spark SQL 性能。

4.1.缓存数据

Spark SQL 通过调用 cacheTable("tableName")，以内存列存储的格式，缓存表。然后，Spark SQL 将只扫描所需的列，并会自动调整压缩，以减少内存使用量和 GC 的压力。你可以调用 uncacheTable("tableName")，将表从内存中删除。

但请注意，如果你调用 cache，而不是 cacheTable，表将不以内存列式存储的格式进行缓存。因此强烈建议在此用例中使用 cacheTable。

可以调用 SQLContext 中的 setConf 方法，或使用 SQL 运行 SET key=value 命令，来完成内存缓存的配置。配置属性如下表：

属性名称	默认值	描述
spark.sql.inMemoryColumnarStorage.compressed	false	当设置为 true 时，Spark SQL 会根据数据的统计，自动选择压缩编码。
spark.sql.inMemoryColumnarStorage.batchSize	1000	控制列缓存时的 batchSize。较大的 batchSize 可提高内存的利用率和压缩，但当缓存数据时，增加了内存泄露（OOMs）的风险。

4.2.其他配置选项

下列选项也可以用来调整查询执行的性能。这些选项可能会在将来的版本中被弃用，因为很多的优化都将自动执行。

属性名称	默认值	描述
spark.sql.autoBroadcastJoinThreshold	10000	在执行 join 操作时，配置一张表将被广播到所有工作节点的最大字节数。设置这个值为-1，则表是不能被广播出去的。需注意的是，目前的数据统计只支持 Hive Metastore 表，并且需要运行命令“ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan”。
spark.sql.codegen	false	如果为 true，在特定查询的表达式求值中，代码将会被动态生成。对于一些复杂表达式的查询，此选项可以得到显著的速度提升。但是，对于简单的查询，这实际上会减缓查询的执行。
spark.sql.shuffle.partitions	200	在进行数据的 shuffle（如 join、aggregation）操作时，配置所用分区的数量。

第 5 章其他 SQL 接口

Spark SQL 还支持直接运行 SQL 查询的接口，而不需要编写任何代码。

5.1.运行 Thrift JDBC/ODBC Server

Thrift JDBC Server 使用的是 HIVE0.12 的 HiveServer2 实现。你能够使用 Spark 或者 hive0.12 版本的 beeline 脚本与 JDBC Server 进行交互。

在 Spark 目录下，执行如下命令，运行 JDBC/ODBC Server：

```
./sbin/start-thriftserver.sh
```

此脚本接受所有"bin/spark-submi"命令行选项，再加上"--hiveconf"选项来指定 Hive 的属性。你可以运行命令：./sbin/start-thriftserver.sh –help，获得所有参数的完整列表说明。此服务的默认监听端口是 10000。你可以通过两种方式修改 host 及 port：修改环境变量，即，

```
export HIVE_SERVER2_THRIFT_PORT=<listening-port>
```

```
export HIVE_SERVER2_THRIFT_BIND_HOST=<listening-host>

./sbin/start-thriftserver.sh \

  --master <master-uri> \

  ...
```

或系统属性：

```
./sbin/start-thriftserver.sh \

  --hiveconf hive.server2.thrift.port=<listening-port> \

  --hiveconf hive.server2.thrift.bind.host=<listening-host> \

  --master <master-uri>

  ...
```

你可以使用 beeline 测试 Thrift JDBC/ODBC Server：

```
./bin/beeline
```

使用如下命令，在 beeline 中连接到 JDBC/ODBC Server：

```
beeline> !connect jdbc:hive2://localhost:10000
```

Beeline 会要求你输入用户名和密码。在非安全模式下，密码为空，因此只需输入你的用户名。在安全模式下，请参考 beeline 文档的说明进行操作。

只需要将你的 hive-site.xml 文件放在 conf/目录下，即可完成 Hive 的配置。

您也可以使用 Hive 中自带的 beeline 脚本。

5.2.运行 Spark SQL CLI

Spark SQL CLI 是一个便捷的工具，它以本地模式运行 Hive metastore 服务，执行从命令行中输入的查询语句。但 Spark SQL CLI 不能与 Thrift JDBC Server 交互。

在 Spark 目录下，执行如下命令，运行 Spark SQL CLI：

```
./bin/spark-sql
```

只需要将你的 hive-site.xml 文件放在 conf/目录下，即可完成 Hive 的配置。你也可以运行命令：./bin/spark-sql --help，获得所有参数的完整列表说明。

第 6 章 Spark SQL 的兼容性

6.1.Shark 用户的迁移指南

6.1.1 调度（scheduling）

用户可以通过设置变量“spark.sql.thriftserver.scheduler.pool”的值，为一次 JDBC 客户端会话设置一个合适的调度池（Fair Scheduler pool），示例代码如下：

```
SET spark.sql.thriftserver.scheduler.pool=accounting;
```

6.1.2 Reducer 数目（Reducer number）

在 Shark 中，默认的 reducer 数量为 1，设置 reducer 数量的属性为：mapred.reduce.tasks。而 Saprk SQL 不用这个属性，用的是 spark.sql.shuffle.partitions，其默认值为：200。用户可以用 SET 命令，自定义这个属性：

```
SET spark.sql.shuffle.partitions=10;

SELECT page, count(*) c

FROM logs_last_month_cached

GROUP BY page ORDER BY c DESC LIMIT 10;
```

你也可以在 hive-site.xml 中设置这个属性，重写默认的属性值。

目前，“mapred.reduce.tasks”仍然可被识别，并自动转变为“spark.sql.shuffle.partitions”属性。

6.1.3 缓存（caching）

Shark 中的表属性“shark.cache”被删除了，以“_cache”结尾的表名的表也不再自动缓存。相反，我们提供了 CACHE TABLE 和 UNCACHE TABLE 操作，让用户显式地控制表的缓存。示例代码如下：

```
CACHE TABLE logs_last_month;

UNCACHE TABLE logs_last_month;
```

注意：与缓存 RDD 一样，CACHE TABLE tbl 是延迟性（lazy）操作。这个命令仅仅是标记了 tbl，以确保当被计算的时候，表的分区被缓存到内存中了。只有当涉及这张表的查询语句执行的时候，这张表才会被真正缓存。若想立即缓存这张表，你可以在执行 CACHE TABLE 后，再执行 count 命令，代码如下：

```
CACHE TABLE logs_last_month;

SELECT COUNT(1) FROM logs_last_month;
```

以下三个缓存相关功能还未实现：

- 用户自定义分区级的缓存回收策略（eviction policy）
- RDD 重载
- 内存级的缓存写操作

6.2 兼容 Apache Hive

Spark SQL 与 Hive MetaStore、SerDes 和 UDFs 相兼容。目前 ,Spark SQL 基于 Hive 0.12.0 及 0.13.1。

6.2.1 在现有的 Hive 仓库中部署

Spark SQL Thrift JDBC Server 与已安装的 Hive 兼容 ,且“开箱即用”(out of the box), 即你不需要修改现有的 Hive MetaStore，或改变数据位置、表的分区。

6.2.2 可支持的 Hive 功能

Spark SQL 支持 Hive 的大部分功能，例如：

- Hive 查询语句，包括：
  - ✓ SELECT
  - ✓ GROUP BY
  - ✓ ORDER BY
  - ✓ CLUSTER BY
  - ✓ SORT BY
- Hive 的所有操作符，包括：
  - ✓ Relational operators (=, ⇔, = = , < > , < , > , > = , < = , etc)
  - ✓ Arithmetic operators (+, -, *, /, %, etc)
  - ✓ Logical operators (AND, &&, OR, ||, etc)
  - ✓ Complex type constructors
  - ✓ Mathematical functions (sign, ln, cos, etc)
  - ✓ String functions (instr, length, printf, etc)
- 用户自定义函数（UDF）
- 用户自定义聚集函数（UDFA）
- 用户自定义序列化格式（SerDes）
- Joins

  - ✓ JOIN
  - ✓ {LEFT|RIGHT|FULL} OUTER JOIN
  - ✓ LEFT SEMI JOIN
  - ✓ CROSS JOIN
- Unions
- Sub-queries
  - ✓ SELECT col FROM (SELECT a + b AS col from t1) t2
- Sampling
- Explain
- Partitioned tables
- View
- All Hive DDL Functions，包括：
  - ✓ CREATE TABLE
  - ✓ CREATE TABLE AS SELECT
  - ✓ ALTER TABLE
- 大多数 Hive 数据类型，包括：
  - ✓ TINYINT
  - ✓ SMALLINT
  - ✓ INT
  - ✓ BIGINT
  - ✓ BOOLEAN
  - ✓ FLOAT
  - ✓ DOUBLE
  - ✓ STRING
  - ✓ BINARY
  - ✓ TIMESTAMP
  - ✓ ARRAY<>
  - ✓ MAP<>
  - ✓ STRUCT<>

6.2.3 不支持的 Hive 功能

下面列出了不支持的 Hive 功能，其中，大部分功能是很少在 Hive 部署中使用的。

1. 主要的 Hive 功能

- ~~Spark SQL 目前不支持使用动态分区将数据插入到表中。~~
- 有 buckets 的表：bucket 是 Hive 表分区中的 hash 分区。Spark SQL 目前不支持 buckets 表

2. 复杂 Hive 功能

- Hive 中表的分区可以使用不同的输入格式；而在 Spark SQL 中，所有的表分区需要使用相同的输入格式
- Non-equi outer join:对于使用有 “non-equi” 连接条件(如 key<10)的外连接用例，这种用例是不常见的，并且 Spark SQL 在空元组处，会输出错误答案。
- UNION 类型和 DATE 类型
- Unique join
- Single query multi insert
- 列统计信息收集：Spark SQL 不捎带扫描并收集目前列的统计信息，仅支持填充 Hive MetaStore 的 sizeInBytes 值。

3. Hive 输入/输出格式

- 输出到客户端的文件格式：Spark SQL 只支持 TextOutputFormat 格式。
- Hadoop archive

4. Hive 优化

一小部分的 Hive 优化策略在 Spark 中也是不被支持的。一部分优化策略（如索引）在 Spark SQL 的内存计算面前显得微不足道；另一部分则会在 Spark SQL 的未来版本中被支持。

- 块级位图索引和虚拟列（用于建立索引）
- 自动将 join 操作转变为 map join 操作 :当一个大表与多个小表进行 join 操作时 ,Hive 会自动将 join 操作转变为 map join 操作。我们将会在下个版本中加入这种自动转换机制。
- 对于 join 操作和 groupby 操作 ,Hive 会自动决定 reducer 的数量 :目前 ,在 Spark SQL 中，你需要使用命令 “SET spark.sql.shuffle.partitions=[num_tasks];”，来控制 post-shuffle 操作的并行度。
- 元数据可供查询：Hive 可以通过只使用元数据来返回查询的结果，但 Spark SQL 仍然会启动任务来计算结果。
- 偏斜数据标志（Skew data flag）: Spark SQL 并不遵循 Hive 中偏斜数据标志。
- STREAMTABLE hint in join：Spark SQL 不遵循 STREAMTABLE 的提示。
- 将查询得到的多个小文件合并：如果结果输出中包含很多小文件，Hive 能够选择性地将这些小文件合并成几个大文件，这样可以避免 HDFS 中元数据的溢出。但 Spark SQL 不支持此操作。

第 7 章语言集成关系型查询

目前，语言集成关系型查询（Language-Integrated Relational Queries）是试验性的，只在 scala 中支持。

Spark SQL 同样支持使用领域语言来编写查询。再次，使用上面例子中的数据：

```
// sc is an existing SparkContext.

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// Importing the SQL context gives access to all the public SQL functions and implicit c
onversions.

import sqlContext._

val people: RDD[Person] = ... // An RDD of case class objects, from the first example.



// The following is the same as 'SELECT name FROM people WHERE age >= 10 AND age <= 19'

val teenagers = people.where('age >= 10).where('age <= 19).select('name)

teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

DSL 使用 Scala 中的标识符（symbols）来代表基础表中的一列，以前缀（'）为标识。隐式转换将这些标识符转换为能够被 SQL 执行引擎识别的表达式。支持这些功能的函数列表可以查阅 ScalaDoc。

第 8 章 Spark SQL 数据类型

- 值类型
  - ✓ ByteType：用 1 个字节表示整数值，取值范围：-128~127。
  - ✓ ShortType：用 2 个字节表示整数值，取值范围：-32768~32767。
  - ✓ IntegerType: :用 4 个字节表示整数值 ,取值范围 :-2147483648~2147483647。
  - ✓ LongType：用 8 个字节表示整数值，取值范围：
  - ✓ -9223372036854775808~9223372036854775807。
  - ✓ FloatType：用 4 个字节表示单精度浮点数。
  - ✓ DoubleType：用 8 个字节表示双精度浮点数。
  - ✓ DecimalType ： 代表任意精度的有符号十进制数。通过内部的 java.math.BigDecimal 支持。BigDecimalBigDecimal 由任意精度的整数非标度值 (unscaled value) 和 32 位的整数标度(scale)组成。
- 字符串类型
  - ✓ StringType：表示字符串值。
- 二进制类型
  - ✓ BinaryType：表示二进制值。
- 布尔类型
  - ✓ BooleanType：表示布尔值。
- 日期类型
  - ✓ TimestampType：表示包括年，月，日，小时，分钟和秒的值。

✓ DateType：表示包括年，月，日的值。

- 复杂类型
  - ✓ ArrayType(elementType, containsNull)：表示一系列数据类型为 elementType 的元素值。containsNull 表示 ArrayType 中的元素是否可为空值。
  - ✓ MapType(keyType, valueType, valueContainsNull)：表示一组 key-value 键值对的值。Key 的类型由 keyType 决定，value 的类型由 valueType 决定。MapType 类型中，key 值不能为空。valueContainsNull 表示 value 值是否可以为空。
  - ✓ StructType(fields)：表示一系列结构为 StructFields (fields)的值。
    - ■ StructField(name, dataType, nullable)：表示 StructType 中的一个字段(field)。参数 name 定义了这个字段的名称。参数 dataType 定义了这个字段的数据类型。nullable 表示这个字段的 value 是否可以为空。

8.1.Scala 版

Spark SQL 中所有的数据类型都位于包：org.apache.spark.sql。你可以在代码中加入以下语句，使用这些数据类型：

```
import org.apache.spark.sql._
```

数据类型	在 Scala 中的值类型	使用或创建数据类型的 API
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Int	IntegerType
LongType	Long	LongType
FloatType	Float	FloatType
DoubleType	Double	DoubleType
DecimalType	scala.math.sql.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array[Byte]	BinaryType
BooleanType	Boolean	BooleanType
TimestampType	java.sql.Timestamp	TimestampType
DateType	java.sql.Date	DateType

ArrayType	scala.collection.Seq	ArrayType(elementType, [containsNull]) Note: containsNull 默认值为 false。
MapType	scala.collection.Map	MapType(keyType,valueType, [valueContainsNull]) Note: valueContainsNull 默认值为 true。
StructType	org.apache.spark.sql.Row	StructType(fields) Note:fields 表示一系列类型为 StructFields 的参数。因此，filelds 不允许有重名。
StructField	scala 中的值类型就是这个字段的数据类型。（例如，Int 型 StructField 的数据类型即为 IntegerType)	StructField(name,dataType,nullable)

8.2.Java 版

Spark SQL 中所有的数据类型都位于包：org.apache.spark.sql.api.java。要使用或创建某数据类型，需要使用包 org.apache.spark.sql.api.java.DataType 中提供的工厂方法。

数据类型	在 Scala 中的值类型	使用或创建数据类型的 API
ByteType	byte or Byte	DataType.ByteType
ShortType	short or Short	DataType.ShortType
IntegerType	int or Integer	DataType.IntegerType
LongType	long or Long	DataType.LongType
FloatType	float or Float	DataType.FloatType
DoubleType	double or Double	DataType.DoubleType
DecimalType	java.math.BigDecimal	DataType.DecimalType
StringType	String	DataType.StringType
BinaryType	byte[]	DataType.BinaryType
BooleanType	boolean or Boolean	DataType.BooleanType
TimestampType	java.sql.Timestamp	DataType.TimestampType

DateType	java.sql.Date	DataType.DateType
ArrayType	java.util.List	DataType.createArrayType(elementType) Note: containsNull 默认值为 false。 DataType.createArrayType(elementType, containsNull).
MapType	java.util.Map	DataType.createMapType(keyType,valueType) Note: valueContainsNull 默认值为 true。 DataType.createMapType(keyType,valueType, valueContainsNull)
StructType	org.apache.spark.sql.api.java	DataType.createStructType(fields) Note:fields 表示一系列类型为 StructFields 的参数。因此，filelds 不允许有重名。
StructField	java 中的值类型就是这个字段的数据类型。（例如，Int 型 StructField 的数据类型即为 IntegerType)	DataType.createStructField(name,dataType,nullable)

8.3.Python 版

Spark SQL 中所有的数据类型都位于包 :pyspark.sql。你可以在代码中加入以下语句，使用这些数据类型：

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

数据类型	在 Python 中的值类型	使用或创建数据类型的 API
ByteType	int or long Note：请确保整数的取值范围为：-128~127。	ByteType()
ShortType	int or long Note：请确保整数的取值范围为：-32768~32767。	ShortType()
IntegerType	int or long	IntegerType()
LongType	long	LongType()

	Note：请确保整数的取值范围为：-9223372036854775808~9223372036854775807。否则，请将数据转换为 decimal.Decimal ，并使用 DecimalType。	
FloatType	float Note：运行时，数据将转变为 4 字节的单精度浮点型数据。	FloatType()
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	list, tuple, or array	ArrayType(elementType, [containsNull]) Note: containsNull 默认值为 false。
MapType	dict	MapType(keyType, valueType, [valueContainsNull]) Note: valueContainsNull 默认值为 true。
StructType	list or tuple	StructType(fields) Note:fields 表示一系列类型为 StructFields 的参数。因此，filelds 不允许有重名。
StructField	python 中的值类型就是这个字段的数据类型。(例如，Int 型 StructField 的数据类型即为 IntegerType)	StructField(name,dataType,nullable)

Spark 亚太研究院

Spark 亚太研究院是中国最专业的一站式大数据 Spark 解决方案供应商和高品质大数据企业级完整培训与服务供应商，以帮助企业规划、架构、部署、开发、培训和使用 Spark 为核心，同时提供 Spark 源码研究和应用技术训练。针对具体 Spark 项目，提供完整而彻底的解决方案。包括 Spark 一站式项目解决方案、Spark 一站式项目实施方案及 Spark 一体化顾问服务。

官网： www.sparkinchina.com

视频课程：

《大数据 Spark 实战高手之路》 国内第一个 Spark 视频系列课程

从零起步，分阶段无任何障碍逐步掌握大数据统一计算平台 Spark，从 Spark 框架编写和开发语言 Scala 开始，到 Spark 企业级开发，再到 Spark 框架源码解析、Spark 与 Hadoop 的融合、商业案例和企业面试，一次性彻底掌握 Spark，成为云计算大数据时代的幸运儿和弄潮儿，笑傲大数据职场和人生！

- 第一阶段：熟练的掌握 Scala 语言
  课程学习地址：http://edu.51cto.com/pack/view/id-124.html
- 第二阶段：精通 Spark 平台本身提供给开发者 API
  课程学习地址：http://edu.51cto.com/pack/view/id-146.html
- 第三阶段：精通 Spark 内核
  课程学习地址：http://edu.51cto.com/pack/view/id-148.html
- 第四阶段：掌握基于 Spark 上的核心框架的使用
  课程学习地址：http://edu.51cto.com/pack/view/id-149.html
- 第五阶段：商业级别大数据中心黄金组合：Hadoop+ Spark
  课程学习地址：http://edu.51cto.com/pack/view/id-150.html
- 第六阶段：Spark 源码完整解析和系统定制
  课程学习地址：http://edu.51cto.com/pack/view/id-151.html

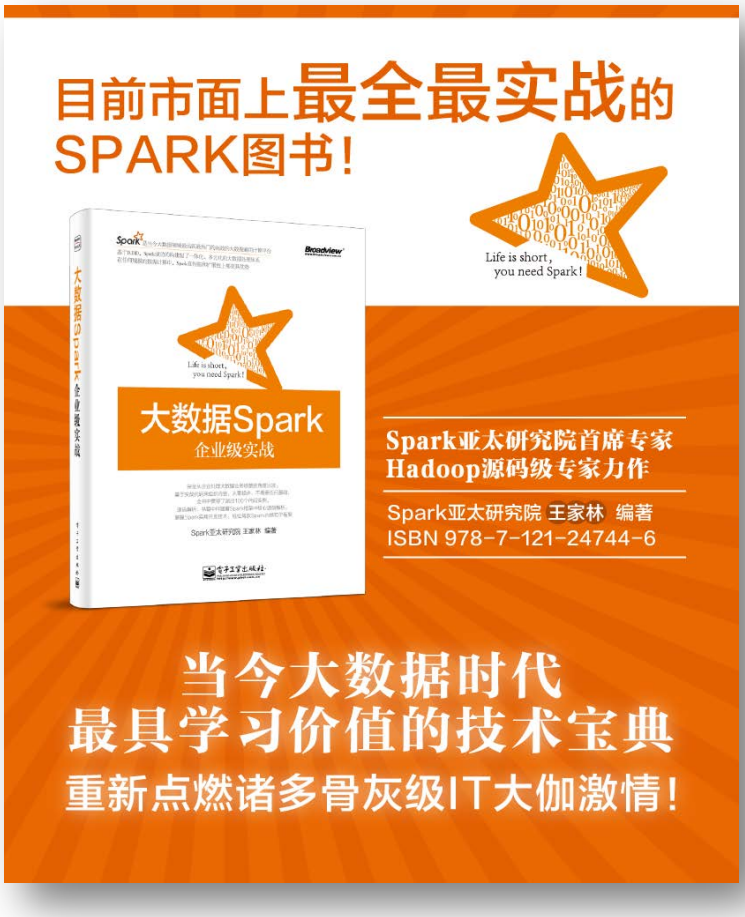
图书：

《大数据 spark 企业级实战》

京东购买官网：http://item.jd.com/11622851.html

当当购买官网：http://product.dangdang.com/23631792.html

亚马逊购买官网：http://www.amazon.cn/dp/B00RMD8KI2/



咨询电话：4006-998-758

QQ 交流群： 1 群：317540673（已满）
2 群：297931500（已满）
3 群：317176983
4 群：324099250

微信公众号：spark-china