亚太研究院

Spark 官方文档翻译

Spark SQL 编程指南

(v1.1.0)

翻译者　徐骄

Spark 官方文档翻译团成员

前 言

世界上第一个Spark 1.1.0 中文文档问世了！

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，历经 1 个月左右的艰苦努力和反复修改，Spark中文文档V1.1 终于完成。尤其值得一提的是，在此次中文文档的翻译期间，Spark官方团队发布了Spark 1.1.0 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主动提出基于最新的Spark 1.1.0 版本，更新了所有已完成的翻译内容，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，作为世界上第一份相对系统的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中文文档第一个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(v1.1.0)》（和唐海东翻译的是同一主题，大家可以对比参考）
- 吴洪泽,《Spark机器学习库（v1.1.0)》（其中聚类和降维部分是蔡立宇翻译）
- 武扬，《在Yarn上运行Spark (v1.1.0)》《Spark 调优(v1.1.0)》
- 徐骄，《Spark配置(v1.1.0)》《Spark SQL编程指南(v1.1.0)》（Spark SQL和韩保礼翻译的是同一主题，大家可以对比参考）
- 蔡立宇，《Bagel 编程指南(v1.1.0)》
- harli，《Spark 编程指南（v1.1.0)》
- 吴卓华，《图计算编程指南(1.1.0)》
- 樊登贵，《EC2(v1.1.0)》《Mesos(v1.1.0)》
- 韩保礼，《Spark SQL编程指南(v1.1.0)》（和徐骄翻译的是同一主题，大家可以对比参考）
- 颜军，《文档首页(v1.1.0)》
- Jack Niu，《Spark实时流处理编程指南(v1.1.0)》
- 俞杭军，《sbt-assembly》《使用Maven编译Spark(v1.1.0)》
- 唐海东，《快速开始(v1.1.0)》（和傅智勇翻译的是同一主题，大家可以对比参考）
- 刘亚卿，《硬件配置(v1.1.0)》《Hadoop 第三方发行版(v1.1.0)》《给Spark提交代码(v1.1.0)》
- 耿元振《集群模式概览(v1.1.0)》《监控与相关工具(v1.1.0)》《提交应用程序(v1.1.0)》
- 王庆刚，《Spark作业调度(v1.1.0)》《Spark安全(v1.1.0)》
- 徐敬丽，《Spark Standalone 模式 (v1.1.0)》

另外关于Spark API的翻译正在进行中，敬请大家关注。

Life is short, You need Spark!

Spark亚太研究院院长 王家林
2014 年 10 月

Spark 亚太研究院决胜大数据时代 100 期公益大讲堂

简 介

作为下一代云计算的核心技术，Spark性能超Hadoop百倍，算法实现仅有其 1/10 或 1/100,是可以革命Hadoop的目前唯一替代者，能够做Hadoop做的一切事情，同时速度比Hadoop快了 100 倍以上。目前Spark已经构建了自己的整个大数据处理生态系统，国外一些大型互联网公司已经部署了Spark。甚至连Hadoop的早期主要贡献者Yahoo现在也在多个项目中部署使用Spark；国内的淘宝、优酷土豆、网易、Baidu、腾讯、皮皮网等已经使用Spark技术用于自己的商业生产系统中，国内外的应用开始越来越广泛。Spark正在逐渐走向成熟，并在这个领域扮演更加重要的角色，刚刚结束的 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发行商都提供了对非常强有力的支持Spark的支持.

鉴于Spark的巨大价值和潜力，同时由于国内极度缺乏Spark人才，Spark亚太研究院在完成了对Spark源码的彻底研究的同时，不断在实际环境中使用Spark的各种特性的基础之上，推出了Spark亚太研究院决胜大数据时代 100 期公益大讲堂，希望能够帮助大家了解Spark的技术。同时，对Spark人才培养有近一步需求的企业和个人，我们将以公开课和企业内训的方式，来帮助大家进行Spark技能的提升。同样，我们也为企业提供一体化的顾问式服务及Spark一站式项目解决方案和实施方案。

Spark亚太研究院决胜大数据时代 100 期公益大讲堂是国内第一个Spark课程免费线上讲座，每周一期，从 7 月份起，每周四晚 20:00-21:30，与大家不见不散！老师将就Spark内核剖析、源码解读、性能优化及商业实战案例等精彩内容与大家分享，干货不容错过！

时间：从 7 月份起，每周一期，每周四晚 20:00-21:30
形式：腾讯课堂在线直播
学习条件：对云计算大数据感兴趣的技术人员
课程学习地址：http://edu.51cto.com/course/course_id-1659.html

Spark SQL 编程指南（V1.1.0）

（翻译者：徐骄）

Spark SQL Programming Guide，原文档链接：http://spark.apache.org/docs/latest/sql-programming-guide.html

目录

1， 综述.....6
2， 入门指南.....6
3， 数据源.....7
1、 RDDs.....7
1） 使用反射机制推断RDD模式.....7
2） 使用编程方式定义RDD模式.....11
2、 Parquet文件.....16
1） 编程来加载数据.....16
2） 配置.....19
3、 JSON数据集.....19
4、 Hive表.....23
4， 执行调优.....25
1、 在内存中缓存数据.....25
2、 其他配置选项.....26
5， 其他的SQL 接口.....27
1、 运行Thrift JDBC.....27
2、 运行Spark SQL CLI.....27
6，与其他系统的兼容性.....28
1、 Spark用户的迁移向导.....28
1） 调度.....28
2） Reducer数量.....28
3） 缓存.....28
2、 与Apache Hive的兼容性.....29
1） 在显存的Hive数据仓库中部署.....29
2） 支持的Hive特性.....29
3） 不支持的Hive函数.....30
7，集成语言的关系型查询.....30
8，Spark SQL数据类型.....31

1， 综述

Spark SQL 允许在 Spark 中执行使用 SQL、HiveQL 或 Scala 表示的关系型查询。此处的核心组件是一个新类型的 RDD——SchemaRDD。SchemaRDDs 由行对象以及用来描述每行中各列数据类型的模式组成。每个 SchemaRDD 类似于关系型数据库中的一个表。SchemaRDD 的创建可以来自于已存在的 RDD 或 Parquet 文件，或 JSON 数据集或运行 HiveQL 而不将其结果存于 Hive。

本文当中所有的示例使用的示例数据包含在了 Spark 版本中并且可以使用 spark-shell 运行。

Spark SQL 目前是一个测试组件，同时我们会最小化 API 的修改，未来的发布版本可能会有 API 的修改。

2， 入门指南

Spark 中的入口是 SQLContext 类，或者其衍生类。要创建一个基本的 SQLContext，首先需要的是一个 SparkContext。以下提供 3 种语言的 SQLContext 创建方式：

【Scala】

```
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】

```
JavaSparkContext sc = ...; // An existing JavaSparkContext.

JavaSQLContext sqlContext = new org.apache.spark.sql.api.java.JavaSQLContext(sc);
```

【Python】

```
from pyspark.sql import SQLContext

 sqlContext = SQLContext(sc)
```

除了基本的 SQLContext 之外，你还可以创建 HiveContext，后者提供了基本

SQLContext 所提供的功能的父集。还有其他的特性包括：查询语句可以使用更完整的 HiveQL 解析器、访问 Hive 的 UDFs、从 Hive 表中读取数据。要使用 HiveContext，Hive 的安装不是必须的，所有 SQLContext 可用的数据源对 HiveContext 仍然可用。HiveContext 只是被包装起来了，以避免包含默认的 Spark 中的所有 Hive 依赖，如果这些依赖对你的应用程序不会造成影响，那么我们建议使用能够 Spark 1.2 版本中的 HiveContext。未来的一些特性将会集中于将 SQLContext 打造得与 HiveContext 相同。

用来解析查询语句的特定的 SQL 也可以使用 spark.sql.dialect 选项选择，这个参数能够通过 SQLContext 的 setConf 方法或者在 SQL 中使用一条 SET key=value 命令进行修改。SparkContext 中提供的唯一一个“方言”是”sql”，它使用了 Spark SQL 提供的一个简单 SQL 解析器；而在 HiveContext 中，尽管也提供了”sql”，但其默认的是”hiveql”。由于 HiveQL 解析器相对更完整，所以更加推荐使用。

3， 数据源

Spark SQL 支持通过 SchemaRDD 接口操作各种各样的数据源。SchemaRDD 可以像普通 RDD 一样的操作，也可以被注册成一个临时表。将一个 SchemaRDD 注册成一个表可以使你基于其数据来运行 SQL 查询。这个部分描述各种将数据加载到 SchemaRDD 的方法。

1、 RDDs

Spark SQL 支持两种不同的方法将已存在的 RDDs 转换成 SchemaRDDs。第一种方法是使用反射通过 RDD 包含的对象类型来推断其模式。这种基于反射的方式会产生很多简介的代码。并且在你写 Spark 应用程序时如果已经知道其模式了，这种方式会非常好用的。

第二种创建 SchemaRDDs 的方法是，你可以通过一个编程接口构造模式，然后将其应用到已存在的 RDD 上。虽然这种方式显得比较冗余，但你可以用它在不知道列及其类型的时候构造 SchemaRDDs。

1） 使用反射机制推断 RDD 模式

【Scala】

Spark SQL 的 Scala 接口支持从包含 case class 的 RDD 到 SchemaRDD 的自动转换。case class 定义了表的模式，其参数名由反射机制读取，并成为表的列名。Case class 可以

嵌套或者包含如序列或数组这样的复杂类型。RDD 可以被隐式转换成 SchemaRDD 然后注册成表，这些表可以在后续的 SQL 语句中使用。

```
// 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 around 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】

Spark SQL 支持 JavaBeans 的 RDD 到 SchemaRDD 的自动转换。使用反射机制得到的 BeanInfo，定义了表的模式。目前，Spark SQL 不支持嵌套的 JavaBeans 或者包含如序列或数组这样的复杂类型。你可以通过创建一个实现了 Serializable 接口的类来创建 JavaBean，对其所有属性都有相应的 getters 和 setters。

```
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;
    }
  }
```

可以通过调用 applySchema 将模式应用到已存在的 RDD 上，并为 JavaBean 提供类对象。

```
  // 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 operations.
    // 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】

Spark SQL 通过推断其数据类型，将一个行对象的 RDD 转换成 SchemaRDD。行是通过给给 Row 类传递一系列的键值对作为 kwargs 构造而成的。所有的键定义了表的列名，类型由第一行推断得出。因为我们目前值查看第一行，所以 RDD 中的第一行不能有缺失数据就变得尤为重要。在将来的版本中，我们打算查看更多的数据进行更完整的推断，类似于在 JSON 文件中执行的推断。

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

```
    # Load a text file and convert each line to a dictionary.
    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 operat
ions.
    teenNames = teenagers.map(lambda p: "Name: " + p.name)
    for teenName in teenNames.collect():
      print teenName
```

2） 使用编程方式定义 RDD 模式

【Scala】

当不能提前定义 case class 时（例如，记录的结构被编码成了一个字符串，或者即将被解析的文本数据集，或者即将为不同用户分别映射的属性），可以通过三个步骤用编程方式来创建 SchemaRDD。

1. 从原始 RDD 创建 RowS 的 RDD
2. 创建由 StructType 表示的模式去匹配第一步中创建的 RDD 的 RowS 结构
3. 通过 SQLContext 的 applySchema 方法将模式应用到 RowS 的 RDD

示例：

```
    // 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"

// 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】

当不能提前定义 JavaBean class 时（例如，记录的结构被编码成了一个字符串，或者即将被解析的文本数据集，或者即将为不同用户分别映射的属性），可以通过三个步骤用编程方式来创建 SchemaRDD。

1. 从原始 RDD 创建 RowS 的 RDD
2. 创建由 StructType 表示的模式去匹配第一步中创建的 RDD 的 RowS 结构
3. 通过 JavaSQLContext 的 applySchema 方法将模式应用到 RowS 的 RDD

示例：

```
// 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, t
rue));
    }
    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】

当不能提前定义 kwargs 字典时（例如，记录的结构被编码成了一个字符串，或者即将被解析的文本数据集，或者即将为不同用户分别映射的属性），可以通过三个步骤用编程方式来创建 SchemaRDD。

1. 从原始 RDD 创建元组或列表 RDD
2. 创建由 StructType 表示的模式去匹配第一步中创建的 RDD 的元组或列表结构
3. 通过 SQLContext 的 applySchema 方法将模式应用到 RowS 的 RDD

示例：

```
# 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 sc
hemaString.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 operat
ions.
    names = results.map(lambda p: "Name: " + p.name)
    for name in names.collect():
      print name
```

2、 Parquet 文件

Parquet 是一种面向列的二进制文件格式，这种文件格式同时也被很多其他的数据处理系统所支持，如 Impala。Spark SQL 支持读写 Parquet 文件，同时将自动保留原始数据的模式。

1） 编程来加载数据

使用上述示例的数据：

【Scala】

```
    // sqlContext from the previous example is used in this example.
    // createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.
    import sqlContext.createSchemaRDD
```

```scala
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】

```java
// 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 schema 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 state
ments.
    parquetFile.registerTempTable("parquetFile");
    JavaSchemaRDD teenagers = sqlContext.sql("SELECT name FROM parquetFile WH
ERE age >= 13 AND age <= 19");
    List<String> teenagerNames = teenagers.map(new Function<Row, String>()
 {
     public String call(Row row) {
       return "Name: " + row.getString(0);
     }
    }).collect();
```

【Python】

```
    # 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 informat
ion.
    schemaPeople.saveAsParquetFile("people.parquet")

    # Read in the Parquet file created above.  Parquet files are self-describin
g 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 state
ments.
    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
```

2） 配置

Parquet 可以通过使用 SQLContext 的 setConf 方法配置，或者通过使用 SQL 运行 SET key=value 进行配置。

属性名称	默认值	属性的意义
spark.sql.parquet.binaryAsString	false	在其他的使用 Parquet 的系统中，特别是在 Impala 和老版本的 Spark SQL 中，当写出成 Parquet 格式时，并不对二进制数据和字符串进行区分。这个属性将告诉 Spark SQL 把二进制数据解释成字符串以提供这些系统的完整系。
spark.sql.parquet.cacheMetadata	false	开启 Parquet 模式元数据的缓存。可以加速静态数据的查询。
spark.sql.parquet.compression.codec	snappy	设置在写 Parquet 文件时的压缩编码。能够被接受的有效值包括：uncompressed,snappy,gzip,lzo。

3、 JSON 数据集

【Scala】

Spark SQL 可以自动推断 JSON 数据集的模式，并将其加载为 SchemaRDD，此转换可以通过使用 SQLContext 中下述方法中的一种实现：

1. jsonFile ： 从 JSON 文件目录中加载数据，其中文件的每一行都是一个 JSON 对象
2. jsonRdd ： 从现有 RDD 加载数据，现有 RDD 中的每一行都是包含了一个 JSON 对象的字符串

```
// 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 tex
t 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() metho
d.
 people.printSchema()
// root
//  |-- age: IntegerType
//  |-- name: StringType


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


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


// Alternatively, a SchemaRDD can be created for a JSON dataset represe
nted 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(anotherPeopleRD
```

```
D)
```

【Java】

Spark SQL 可以自动推断 JSON 数据集的模式，并将其加载为 JavaSchemaRDD，此转换可以通过使用 JavaSQLContext 中下述方法中的一种实现：

1. jsonFile ： 从 JSON 文件目录中加载数据，其中文件的每一行都是一个 JSON 对象
2. jsonRdd ： 从现有 RDD 加载数据，现有 RDD 中的每一行都是包含了一个 JSON 对象的字符串

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

    // A JSON dataset is pointed to by path.
    // The path can be either a single text file or a directory storing text fi
les.
    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: IntegerType
    //  |-- name: StringType

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

    // SQL statements can be run by using the sql methods provided by sqlContex
t.
    JavaSchemaRDD teenagers = sqlContext.sql("SELECT name FROM people WHERE a
```

```
ge >= 13 AND age <= 19");


    // Alternatively, a JavaSchemaRDD can be created for a JSON dataset represe
nted 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);
```

【Python】

Spark SQL 可以自动推断 JSON 数据集的模式，并将其加载为 SchemaRDD，此转换可以通过使用 SQLContext 中下述方法中的一种实现：

1. jsonFile ： 从 JSON 文件目录中加载数据，其中文件的每一行都是一个 JSON 对象
2. jsonRdd ： 从现有 RDD 加载数据，现有 RDD 中的每一行都是包含了一个 JSON 对象的字符串

```
# 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() metho
d.
people.printSchema()
```

```
    # root
    #  |-- age: IntegerType
    #  |-- name: StringType

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

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

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

4、 Hive 表

Spark SQL 还支持存储在 Apache Hive 中的数据读写。然而，因为 Hive 有大量的依赖关系，所以默认并不包含在 Spark assembly 中。如果要使用 Hive，就必须先运行”sbt/sbt –Phive assembly/assembly”（或在 maven 中使用-Phive）。这个命令构建了一个新的包含 Hive 的 assembly jar 包。注意，为了访问存储在 Hive 中的数据，这个 Hive assembly jar 包将需要访问 Hive 序列化和反序列化库 ,所以它们(Hive assembly jar 包）必须在所有的 worker 节点上都有。

Hive 的配置直接通过将 hive-site.xml 文件置于 conf/下即可。

【Scala】

当在 Spark 中使用 Hive 时，你必须构建一个 HiveContext，它（HiveContext）继承自 SQLContext，并且为查找 MetaStore 中的表以及使用 HiveQL 进行查询提供支持。没有部署 Hive 的用户也可以创建 HiveContext，此时没有 hive-site.xml 文件，那么上下文会在当前目录下自动的创建 metastore_db 和 warehouse 两个目录。

```
// 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)
```

【Java】

当在 Spark 中使用 Hive 时，你必须构建一个 JavaHiveContext，它（JavaHiveContext）继承自 JavaSQLContext，并且为查找 MetaStore 中的表以及使用 HiveQL 进行查询提供支持。除了 sql 方法之外，JavaHiveContext 还提供了 hql 方法允许用户使用 HiveQL 进行查询。

```
// 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();
```

【Python】

当在 Spark 中使用 Hive 时，你必须构建一个 HiveContext，它（HiveContext）继承自 SQLContext，并且为查找 MetaStore 中的表以及使用 HiveQL 进行查询提供支持。除了 sql 方法之外，HiveContext 还提供了 hql 方法允许用户使用 HiveQL 进行查询。

```
# 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.t
xt' INTO TABLE src")


    # Queries can be expressed in HiveQL.
    results = sqlContext.sql("FROM src SELECT key, value").collect()
```

4， 执行调优

有些工作负载可以通过缓存数据到内存或对某些试验性选项调优来提升其性能。

1、 在内存中缓存数据

Spark SQL 通过调用 cacheTable(“tablename”)使用内存中的列式格式来对表进行缓存，Spark SQL 将仅扫描需要的列并且会自动进行调整尽量压缩以最小化内存使用率和 GC 的压力。还可以调用 uncacheTable(“tableName”)将该表从内存中移除。

注意，如果你调用的是 cache 而不是 cacheTable，那么表不会以列式格式缓存与内存之中，因此我们强烈建议在调优时使用 cacheTable。

内存缓存的配置可以通过 SQLContext 的 setConf 方法或者运行 SQL 的 SET key=value 命令进行设置。

属性名称	默认值	属性的意义
spark.sql.inMemoryColumnarStorage.compressed	false	若设置为 true，Spark SQL 将基于数据的信息统计量为每个列自动选择压缩编码。
spark.sql.inMemoryColumnarStorage.batchSize	1000	控制列式缓存的批量大小。批量越大可以提高内存的使用率和压缩率 ,但是会有内存溢

		出的风险。

2、 其他配置选项

下列选项也可以被用于查询调优。可能这些选项会与后期的发布版本中自动执行的优化重复。

属性名称	默认值	属性的意义
spark.sql.autoBroadcastJoinThreshold	10000	配置在执行join操作时 ,将要被广播给所有worker节点的表的最大值（单位：字节）。将此值设置为-1 ,广播将被取消。注意 , 当前的信息统计量在运行 `ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan` 命令只对Hive的MetaStore进行支持。
spark.sql.codegen	false	若设为 true ,查询中的表达式将在运行时动态生成编码。这个选项对于包含复杂表达式的查询会有显著的速度提升 ,而对简单的查询 ,实际上这个选项可能会降低其查询速度。
spark.sql.shuffle.partitions	200	此选项配置在执行连接操作和聚合操作的 Shuffle 数据时所使用的分区数量。

5， 其他的 SQL 接口

Spark SQL 还支持直接运行 SQL 查询而不需要任何编码。

1、 运行 Thrift JDBC

此处的 Thrift JDBC server 对应于 Hive 0.12 中的 HiveServer2，可以使用 beeline（Hive 0.12 和 Spark 中都有的一个 CLI 脚本）测试 JDBC server。

启动 JDBC server，在 Spark 目录中运行下列命令：

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

JDBC server 默认的监听端口是 10000，如果要监听自定义的主机和端口，需要设置两个环境变量 :HIVE_SERVER2_THRIFT_PORT 和 HIVE_SERVER2_THRIFT_BIND_HOST。可以运行./sbin/start-thriftserver.sh –help 查看完整的选项列表。接下来你可以使用 beeline 来测试 Thrift JDBC server:

```
./bin/beeline
```

在 beeline 中连接 JDBC server：

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

Beeline 要求你输入用户命和密码。在非安全模式下，只要输入机器上你的用户名以及空白密码即可进入。而在安全模式下，请按照 Beeline 文档给出的指令提示输入。

Hive 的配置直接将 hive-site.xml 放置在 conf/下即可完成。

你也可以使用 Hive 中的 Beeline。

2、 运行 Spark SQL CLI

Spark SQL 的 CLI 对于以本地模式运行 Hive 元数据服务（metastore service）以及执行来自命令行的查询都是一个非常方便的工具。注意 ,Spark SQL CLI 不能与 Thrift JDBC server 通信。

启动 Spark SQL CLI，在 Spark 目录中运行下面的命令：

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

Hive 的配置直接将 hive-site.xml 放置在 conf/下即可完成。运行./bin/spark-sql –help 可以列出完整的可选列表。

6， 与其他系统的兼容性

1、 Spark 用户的迁移向导

1） 调度

为 JDBC 会话设置 Fair 调度器池 ,要设置 spark.sql.thriftserver.scheduler.pool 变量：

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

2） Reducer 数量

在 Shark 中 ,默认的 reducer 只有 1 个 ,由 mapred.reduce.tasks 属性控制 ,在 Spark SQL 中由 spark.sql.shuffle.partitions 属性覆盖，后者的默认值是 200.用户可以通过 SET 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;
```

你也能够在 hive-site.xml 中设置这个属性覆盖掉默认值。

现在属性仍然是被识别的，并且会被自动的转换成 spark.sql.shuffle.partitions 属性。

3） 缓存

表 shark.cache 属性不存在了，且以_cached 结尾的表也不会自动被缓存。但是我们提供了 CACHE TABLE 和 UNCACHE TABLE 语句让用户可以精确的控制表的缓存：

```
CACHE TABLE logs_last_month;

UNCACHE TABLE logs_last_month;
```

注意：CACHE TABLE tbl 是懒值性质的，与 RDD 中的.cache 类似。这个命令是标记 tbl 以确保在计算时它的分区会被缓存，但是在真正执行到出发 tbl 的查询之前，它实际上是不会被缓存的，这就是 lazy 性质（懒值性质）。如果你要强制缓存某个表，你可以通过执行 CACHE TABLE 之后立即对该表计数：

```
CACHE TABLE logs_last_month;

SELECT COUNT(1) FROM logs_last_month;
```

有些缓存相关的特性目前还未被支持：

用户自定义分区级别缓存回收策略

RDD 重加载

基于策略的内存中的写缓存

2、 与 Apache Hive 的兼容性

Spark SQL 设计目标之一就是兼容 Hive MetaStore，序列化&反序列化以及 UDFs，基于这点，目前 Spark SQL 是基于 Hive 0.12.0。

1） 在显存的 Hive 数据仓库中部署

Spark SQL 的 Thrift JDBC server 被设计成“开箱即用”的兼容于现有的 Hive 安装，不需要修改现有的 Hive MetaStore 或数据位置或表的分区。

2） 支持的 Hive 特性

Spark SQL 支持很多 Hive 的主要特性，例如：

- Hive 查询语句，包括：SELECT/GROUP BY/ORDER BY/CLUSTER BY/SORT BY
- 所有的 Hive 运算符，包括：关系运算符（=,⇔，==，<>，<,>,>=,<=等）、算数运算符（+,-,*,/,%）、逻辑运算符（AND,&&,OR,||等）、复杂类型构造器、算数函数（sign,ln,cos 等）、字符串函数（instr,length,printf 等）
- 用户自定义函数（UDF）
- 用户自定义聚合函数（UDAF）
- 用户自定义序列化格式（SerDes）
- 连接操作：JOIN/{LEFT|RIGHT|FULL} OUTER JOIN/LEFT SEMI JOIN/CROSS JOIN
- Unions
- 子查询： SELECT col FROM (SELECT a+b AS col from t1) t2
- 取样
- EXPLAIN
- 分区表
- 所有的 HiveDDL 操作函数，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/TIME

STAMP/ARRAY<>/MAP<>/STRUCT<>

3） 不支持的 Hive 函数

下面列出的是目前还不支持的 Hive 特性，大多数这些特性很少在 Hive 部署中使用。

- 主要的 Hive 特性：Spark SQL 目前不支持使用动态分区进行表插入；桶：桶是 Hive 表分区中的哈希分区。Spark SQL 还不支持桶。
- 较少用的 Hive 特性：1）分区表使用不同的输入格式，在 Spark SQL 中，所有的表分区必须是相同的输入格式；2）非等值外连接，对于较少使用的非等值外连接，Spark SQL 对 NULL 元组会输出错误结果；3）UNION 类型和 DATE 类型；4）Unique 连接；5）单查询多插入；6）多列的统计信息收集，Spark SQL 不会即时进行背负式扫描以收集列的统计信息，而只支持 Hive MetaStore 中字节大小属性的计算。
- Hive 的输入输出格式：1）CLI 的文件格式，返回至 CLI 的显示结果，Spark SQL 只支持 TextOutputFormat；2）Hadoop archive。
- Hive 优化：Spark 中一小部分 Hive 优化目前还没有囊括进来。其中一些（如索引）因为 Spark SQL 的内存计算模型而变得不那么重要了，其他的一些优化特性在后期发布的 Spark SQL 中会分别加入。目前还未被支持的 Hive 优化有：

块级别的位图索引和虚拟列（用于建索引）；

将 join 自动转换成 map join：一个大表在连接多个小表时，Hive 会自动将其转换成 map join，我们会在下一个发布版中提供这种自动转换；

对join和groupby操作的reducer数目的自动决策：在目前的Spark SQL中，你需要通过语句 “SET spark.sql.shuffle.partitions=[num_tasks];” 来控制post-shuffle的并行度；

只查询元数据：对那些仅使用元数据就能返回结果的查询，Spark SQL 仍然会启动很多任务(tasks)来计算结果。

数据倾斜标记：Spark SQL 不遵循 Hive 中的数据倾斜标记；

Join 中提示的 STREAMTABLE：Spark SQL 不遵循 STREAMTABLE 提示；

合并查询结果产生的大量小文件：如果输出结果包含很多小文件，Hive 可以选择合并这些小文件以避免 HDFS 元数据的溢出，而 Spark SQL 还不支持这个特性。

7，集成语言的关系型查询

集成语言的查询目前还在试验中且仅支持 Scala。

Spark SQL 还支持领域语言编写的查询。再次使用上述示例中的数据举例：

```
// 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 conversions.
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 标记表示当下面表中的列 这些标记是以一个单引号作为前缀的标识符。这些标记会隐式转换成被 SQL 执行引擎评估的表达式。所支持的函数列表可以在 ScalaDoc 中找到。

8，Spark SQL 数据类型

- 数值类型

ByteType：表示 1 个字节大小的有符号整数，其范围是-128~128；

ShortType：表示 2 个字节大小的有符号整数，其范围是-32768~32767；

IntegerType：表示 4 个字节大小的有符号整数，其范围是-2147483648~2147483647；

LongType：表示 8 个字节大小的有符号整数，其范围是-9223372036854775808~9223372036854775807；

FloatType：表示 4 个字节大小的单精度浮点数；

DoubleType：表示 8 个字节大小的双精度浮点数；

DecimalType；

- 字符串类型

StringType：表示字符串值；

- 二进制类型

BinaryType：表示字节序列的值；

- 布尔类型

BooleanType：表示布尔值；

- 时间类型

TimestampType：表示由年，月，日，时，分，秒组成的时间值；

- 复杂类型

ArrayType（elementType,containsNull）：表示由 elementType 类型的元素序列组成的值。containsNull 用于表示 ArrayType 的元素是否可以为空。

MapType（keyType,valueType,valueContainsNull）：表示由一系列 key-value 对组成的集合。Key 的数据类型为 keyType ,value 的数据类型为 valueType。MapType 中 key 不能为 Null 值，valueContainsNull 用来表示 value 是否可以包含 null 值。

StructType（fields）：表示一系列由 StructField（fields）描述的结构：

StructField（name,dataType,nullable）：表示 StructType 中的一个属性。Name 表示属性的名字，dataType 表示属性的数据类型，nullable 表示属性的值是否可以为 null。

【Scala】

针对 Scala，Spark SQL 的所有数据类型位于 org.apache.spark.sql 包中，可以通过下面的语句访问：

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

数据类型	Scala 中的值类型	访问或创建数据类型的 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
TimestampTyp	Java.sql.Timestamp	TimestampType

e		
ArrayType	Scala.collection.Seq	ArrayType(elementType,[containsNull] 注意：containsNull 默认值为 false
MapType	Scala.collection.Map	MapType(keyType,valueType,[valueContainsNull] 注意：valueContainsNull 默认值为 true
StructType	Org.apache.spark.sql.Row	StructType(fields) 注意：fields 是 StructField 的序列，两个 fields 不能同名
StructField	Field 数据类型在 Scala 中的值类型（例如，StructField 的数据类型是 IntegerType，则此值为 Int）	StructField(name,dataType,nullable)

【Java】

针对 Java，Spark SQL 的所有数据类型位于 org.apache.spark.sql.api.java 包中，要创建或访问一个数据类型，请使用 org.apache.spark.sql.api.java.DataType 提供的工厂方法。

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

ArrayType	java.util.List	DataType.createArrayType(elementType,[containsNull] 注意：containsNull 默认值为 false
MapType	java.util.Map	DataType.createMapType(keyType,valueType,[valueContainsNull] 注意：valueContainsNull 默认值为 true
StructType	org.apache.spark.sql.api.java	DataType.createStructType(fields) 注意：fields 是 StructField 的列表或数组，两个 fields 不能同名
StructField	Field 数据类型在 Java 中的值类型（例如，StructField 的数据类型是 IntegerType，则此值为 Int）	DataType.createStructField(name,dataType,nullable)

【Python】

针对 Python，Spark SQL 位于 pyspark.sql 包中，可以通过下面语句访问：

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

数据类型	Scala 中的值类型	访问或创建数据类型的 API
ByteType	int 或 long 注意：该数将在运行时转换成 1 个字节的有符号整数。请确定该数在-128~127	ByteType()
ShortType	int 或 long 注意：该数将在运行时转换成 2 个字节的有符号整数。请确定该数在-32768~32767	ShortType()
IntegerType	int 或 long	IntegerType()
LongType	long 注意：该数将在运行时转换成 8 个字节的有符号整数。请确定该数在-9223372036854775808~9223372036854775807	LongType()
FloatType	float	FloatType()

	注意：该数将在运行时转换成 4 个字节的单精度浮点数。	
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	Datetime.datetime	TimestampType()
ArrayType	list,tuple,array	ArrayType(elementType,[containsNull] () 注意：containsNull 默认值为 false
MapType	dict	MapType(keyType,valueType,[valueContainsNull] 注意：valueContainsNull 默认值为 true
StructType	List,tuple	StructType(fields) 注意：fields 是 StructField 的序列，两个 fields 不能同名
StructField	Field 数据类型在 Python 中的值类型（例如，StructField 的数据类型是 IntegerType，则此值为 Int）	StructField(name,dataType,nullable)

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

近期活动

- 2014 年亚太地区规格最高的 Spark 技术盛会！
- 面向大数据、云计算开发者、技术爱好者的饕餮盛宴！
- 云集国内外 Spark 技术领军人物及灵魂人物！
- 技术交流、应用分享、源码研究、商业案例探询！

时间：2014 年 12 月 6-7 日
地点：北京珠三角万豪酒店
Spark亚太峰会网址：http://www.sparkinchina.com/meeting/2014yt/default.asp

- 如果你是对 Spark 有浓厚兴趣的初学者，在这里你会有绝佳的入门和实践机会！
- 如果你是 Spark 的应用高手，在这里以“武”会友，和技术大牛们尽情切磋！
- 如果你是对 Spark 有深入独特见解的专家，在这里可以尽情展现你的才华！

比赛时间：
2014 年 9 月 30 日—12 月 3 日
Spark开发者大赛网址：http://www.sparkinchina.com/meeting/2014yt/dhhd.asp

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

近期公开课：

《决胜大数据时代：Hadoop、Yarn、Spark 企业级最佳实践》

集大数据领域最核心三大技术：Hadoop 方向 50%：掌握生产环境下、源码级别下的 Hadoop 经验，解决性能、集群难点问题；Yarn 方向 20%：掌握最佳的分布式集群资源管理框架，能够轻松使用 Yarn 管理 Hadoop、Spark 等；Spark 方向 30%：未来统一的大数据框架平台，剖析 Spark 架构、内核等核心技术，对未来转向 SPARK 技术，做好技术储备。课程内容落地性强，即解决当下问题，又有助于驾驭未来。

开课时间： 2014 年 10 月 26-28 日北京、2014 年 11 月 1-3 日深圳

咨询电话：4006-998-758

QQ 交流群： 1 群：317540673 （已满）
2 群：297931500

微信公众号：spark-china