《Flink编程基础（Scala版）》

教材官网：http://dblab.xmu.edu.cn/post/flink/

温馨提示：编辑幻灯片母版，可以修改每页PPT的厦大校徽和底部文字

第7章 Table API&SQL

（PPT版本号：2021年3月版本）

扫一扫访问教材官网

林子雨

厦门大学计算机科学系

E-mail: ziyulin@xmu.edu.cn

主页：**http://dblab.xmu.edu.cn/linziyu**

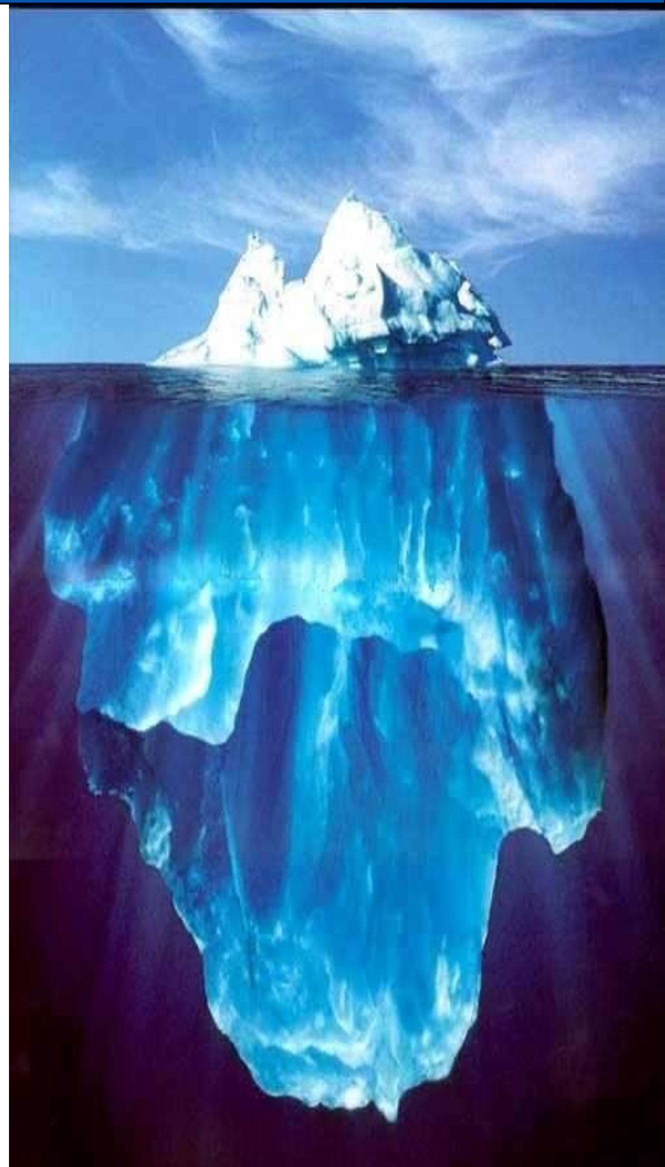
提纲

- 7.1 编程模型
- 7.2 Flink Table API
- 7.3 Flink SQL
- 7.4 自定义函数

高校大数据课程

公 共 服 务 平 台

百度搜索厦门大学数据库实验室网站访问平台



7.1 编程模型

7.1.1 程序执行原理
7.1.2 程序结构
7.1.3 TableEnvironment

7.1.1 程序执行原理

如图7-1中所示，一段使用SQL或TableAPI 编写的程序，从输入到编译为可执行的 JobGraph，主要经历如下几个阶段：

•将SQL文本或Table API代码转化为逻辑执行计划（Logical Plan）；
•通过优化器把逻辑查询计划优化为物理执行计划（Physical Plan）；
•通过代码生成技术生成Transformation后，进一步编译为可执行的JobGraph，然后提交给Flink集群运行。

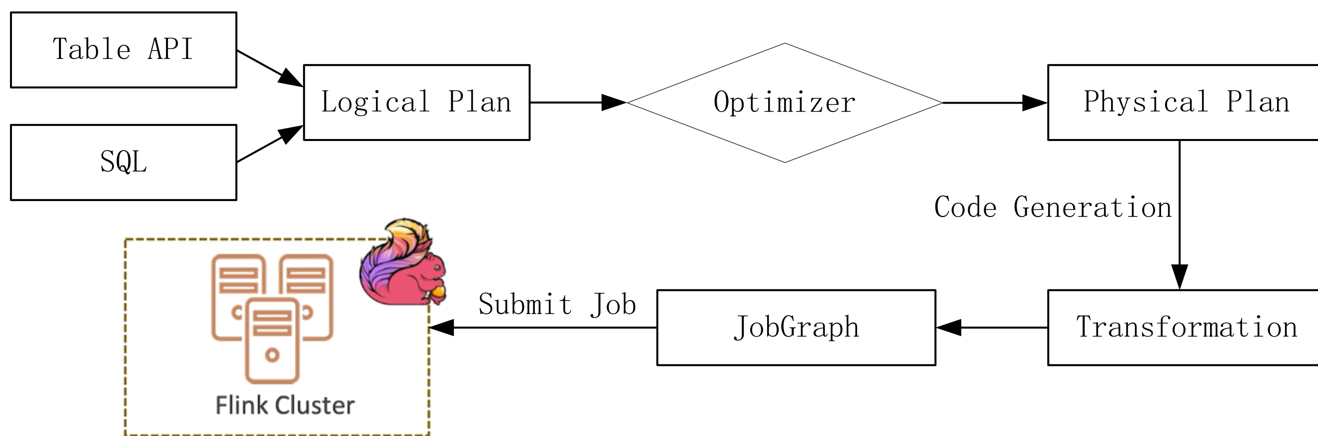


图 Table API&SQL程序执行原理

7.1.2 程序结构

基于Table API&SQL的数据处理应用程序主要包括以下6个步骤：

- 获取运行时；
- 获取TableEnvironment；
- 注册表；
- 定义查询；
- 输出结果；
- 启动程序。

7.1.2 程序结构

程序基本框架如下：

```
//获取运行时
env=...;
//获取TableEnvironment对象
tableEnv = ...;
//注册一个表（输入数据）
tableEnv.connect(...).createTemporaryTable("table1");
//注册一个表（输出数据）
tableEnv.connect(...).createTemporaryTable("outputTable");
//定义查询：通过Table API的查询创建一个Table对象
tapiResult = tableEnv.from("table1").select(...);
//定义查询：通过SQL查询的查询创建一个Table对象
sqlResult = tableEnv.sqlQuery("SELECT ... FROM table1 ... ");
//输出结果
tapiResult.insertInto("outputTable");
//启动程序
tableEnv.execute("Table API and SQL");
```

7.1.2 程序结构

需要注意的是，在pom.xml文件中，需要加入相应的flink-table-api-scala-bridge_2.12依赖库，库中包含了Table API&SQL接口，具体如下：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-scala-bridge_2.12</artifactId>
  <version>1.11.2</version>
</dependency>
```

此外，如果要在本地用IDE（比如IntelliJ IDEA或Eclipse）调试程序，则还需要加入如下依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-blink_2.12</artifactId>
  <version>1.11.2</version>
</dependency>
```

7.1.2 程序结构

不管是批处理还是流处理，还需要在pom.xml文件中引入flink-streaming-scala_2.12依赖库，具体如下：

```
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-scala_2.12</artifactId>
        <version>1.11.2</version>
</dependency>
```

对于批处理应用，还需要在pom.xml文件中引入flink-scala_2.12依赖库，具体如下：

```
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-scala_2.12</artifactId>
        <version>1.11.2</version>
</dependency>
```

7.1.2 程序结构

如果要实现用户自定义函数或者要与Kafka交互，则还需要加入以下依赖库：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-common</artifactId>
  <version>1.11.2</version>
</dependency>
```

7.1.3 TableEnvironment

使用Table API和SQL创建Flink应用程序，需要在环境中创建TableEnvironment。TableEnvironment提供了注册内部表、执行Flink SQL语句、注册自定义函数、将DataStream或DataSet转换成表等功能。

对于流处理，TableEnvironment的创建方法如下：

```
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.table.api.EnvironmentSettings
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment

val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment
val bsSettings =
EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode()
.build()
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.1.3 TableEnvironment

对于批处理，TableEnvironment的创建方法如下：

```
import org.apache.flink.table.api.{EnvironmentSettings, TableEnvironment}

val bbSettings =
EnvironmentSettings.newInstance().useBlinkPlanner().inBatchMode().build()
val bbTableEnv = TableEnvironment.create(bbSettings)
```

7.1.4注册表

在Flink中，表可以分为视图（View）和常规的表（Table）。视图可以从一个已经存在的表对象中创建，通常是一个查询结果。常规的表则是描述来自外部的数据，比如文件或者数据库等。下面是创建一个视图的代码示例：

```
//创建 TableEnvironment
val tableEnv = ...

//使用环境对象从外部表中查询，并将结果创建一张表
val projTable: Table = tableEnv.from("X").select(...)

//使用表projTable 注册成临时表 "projectedTable"
tableEnv.createTemporaryView("projectedTable", projTable)
```

在Flink中，创建和注册表的另一个方式是通过表连接器（Table Connector）。在Table API&SQL中，Flink可以通过表连接器（Table Connector）直接连接外部系统，将批量或者流式数据从外部系统中获取到Flink系统中，或者从Flink系统中将数据发送到外部系统。连接器描述了存储表数据的外部系统。存储系统（例如 Apache Kafka 或者常规的文件系统）都可以通过这种方式来创建表。具体使用方法如下：

```
tableEnvironment
  .connect(...)  //指定表连接器的描述符
  .withFormat(...) //指定数据格式
  .withSchema(...) //指定表结构
  .inAppendMode() //指定更新模式
  .createTemporaryTable("MyTable") //注册表
```

7.1.4注册表

1.表连接器

Flink提供了一些内置的表连接器，包括文件系统连接器、Kafka连接器、Elasticsearch连接器、JDBC SQL连接器等。这里介绍文件系统连接器和JDBC SQL连接器的用法。
文件系统连接器允许用户从本地或者分布式文件系统中读取和写入数据。下面是一个具体实例：

```
bsTableEnv.connect(
      new FileSystem()
        .path("file:///home/hadoop/stockprice.txt")
    )
```

7.1.4注册表

JDBC SQL连接器允许用户从那些支持JDBC驱动程序的关系数据库中读取和写入数据。下面是一个具体实例：

```
CREATE TABLE MyUserTable (
  id BIGINT,
  name STRING,
  age INT,
  status BOOLEAN,
  PRIMARY KEY (id) NOT ENFORCED
) WITH (
   'connector' = 'jdbc',
   'url' = 'jdbc:mysql://localhost:3306/mydatabase',
   'table-name' = 'users'
);
```

7.1.4注册表

上面这个实例中，WITH从句中只提供了3个参数，实际上可以提供更多的参数，具体参数及其含义可以参见表7-1。

参数名称	是否必须	含义
connector	必须	确定连接器类型，对于JDBC连接器就是“jdbc”
url	必须	要连接的JDBC数据库的地址
table-name	必须	要连接的JDBC表的名称
driver	可选	JDBC驱动类的名称，如果没有提供，则自动从url中获取
username	可选	JDBC数据库的用户名，必须和密码一起提供
password	可选	JDBC数据库的密码，必须和用户名一起提供

7.1.4注册表

2.表格式

为了支持表连接器传输不同格式类型的数据，Flink提供了常用的表格式（Table Format），例如CSV格式和JSON格式等。可以调用TableEnvironment的withFormat()方法来指定表格式。

7.1.4注册表

这里介绍CSV格式的用法，其他格式的用法可以参考Flink官网。CSV格式指定分隔符切分数据记录中的字段，具体如下：

```
.withFormat(
  new Csv()
    .field("field1", Types.STRING)    //根据顺序指定字段名称和类型（必选）
    .field("field2", Types.TIMESTAMP) //根据顺序指定字段名称和类型（必选）
    .fieldDelimiter(",")           //指定列切割符，默认使用","（可选）
    .lineDelimiter("\n")           //指定行切割符，默认使用"\n"（可选）
    .quoteCharacter('"')            //指定字符串中的单个字符，默认为空（可选）
    .commentPrefix('#')            //指定注释的前缀，默认为空（可选）
    .ignoreFirstLine()            //是否忽略第一行（可选）
    .ignoreParseErrors()            //是否忽略解析错误的数据（可选）
)
```

7.1.4注册表

此外，还需要在pom.xml中加入如下依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-csv</artifactId>
    <version>1.11.2</version>
</dependency>
```

7.1.4注册表

3.表模式

表模式（Table Schema）定义了表的数据结构，包括字段名称、字段类型等信息。同时，表模式会和表格式相匹配，在表数据输入或者输出的过程中完成模式的转换。表模式的定义方法如下：

```
.withSchema(
  new Schema()
    .field("MyField1", Types.INT)     // 根据顺序指定第1个字段的名称和类型
    .field("MyField2", Types.STRING)  // 根据顺序指定第2个字段的名称和类型
    .field("MyField3", Types.BOOLEAN) // 根据顺序指定第3个字段的名称和类型
)
```

7.1.4注册表

4.更新模式

对于Stream类型的表数据，需要标记出是由于INSERT、UPDATE、DELETE中的哪种操作更新的数据，在Table API中通过更新模式（Update Mode）来指定数据更新的类型，通过指定不同的更新模式，来确定是哪种更新操作的数据与外部系统进行交互。更新模式的定义方法如下：

```
.connect(...)
  .inAppendMode()  //仅交互INSERT操作更新数据
  .inUpsertMode()  //仅交互INSERT、UPDATE、DELETE操作更新数据
  .inRetractMode()  //仅交互INSERT和DELETE操作更新数据
```

7.1.4注册表

5.应用实例

（1）读取文件

这里给出一个简单的Table API数据处理应用程序。假设已经存在一个文本文件“/home/hadoop/stockprice.txt”，其内容如下：

```
stock_2,1602031562148,43.5
stock_1,1602031562148,22.9
stock_0,1602031562153,8.3
stock_2,1602031562153,42.1
stock_1,1602031562158,22.2
```

7.1.4注册表

下面我们编写一个程序，使用Table API进行查询操作。程序内容如下：

```
package cn.edu.xmu.dblab

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._

case class StockPrice(stockId:String,timeStamp:Long,price:Double)

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)
```

7.1.4注册表

```
//获取EnvironmentSettings
val bsSettings = EnvironmentSettings
  .newInstance()
  .useBlinkPlanner()
  .inStreamingMode()
  .build()

//获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.1.4注册表

```
//创建数据源
val stockTable = bsTableEnv.connect(
  new FileSystem()
    .path("file:///home/hadoop/stockprice.txt")
).withFormat(new Csv())
  .withSchema(new Schema()
    .field("stockId", DataTypes.STRING())
    .field("timeStamp", DataTypes.BIGINT())
    .field("price", DataTypes.DOUBLE())
  ).createTemporaryTable("stocktable")
```

7.1.4注册表

```
    //使用Table API查询
val stock = bsTableEnv.from("stocktable") .select($"stockId",$"timeStamp",$"price")

    //打印输出
stock.toAppendStream[(String, Long, Double)].print()

    //程序触发执行
bsEnv.execute("TableAPITest")
  }
}
```

7.1.4注册表

该程序执行以后的输出结果如下：

```
(stock_2,1602031562148,43.5)
(stock_1,1602031562148,22.9)
(stock_0,1602031562153,8.3)
(stock_2,1602031562153,42.1)
(stock_1,1602031562158,22.2)
```

7.1.4注册表

（2）读写MySQL数据库

现在需要读取这4条数据显示到屏幕上，并且向student表中新插入一条数据("95003","3",97)

sno	cno	grade
95001	1	94
95001	2	89
95002	1	91
95002	2	86

7.1.4注册表

完成上述功能的程序代码如下：

```
package cn.edu.xmu.dblab

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.descriptors.{Csv, FileSystem, Schema}

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)
```

7.1.4注册表

```
//获取EnvironmentSettings
val bsSettings = EnvironmentSettings
  .newInstance()
  .useBlinkPlanner()
  .inStreamingMode()
  .build()

//获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)

//创建一个数据流
val dataStream = bsEnv.fromElements(Tuple3("95003","3",97))

//把数据流转换成表（这个知识点会在第7.1.7节介绍）
val table1 = bsTableEnv.fromDataStream(dataStream)
```

7.1.4注册表

```
//创建表student
  val sinkDDL: String =
    """
      |create table student (
      | sno varchar(20) not null,
      | cno varchar(20) not null,
      | grade int
      |) with (
      | 'connector.type' = 'jdbc',
      | 'connector.url' = 'jdbc:mysql://localhost:3306/flink',
      | 'connector.table' = 'student',
      | 'connector.driver' = 'com.mysql.jdbc.Driver',
      | 'connector.username' = 'root',
      | 'connector.password' = '123456'
      | )
      """.stripMargin
```

7.1.4注册表

```
    //执行SQL语句
bsTableEnv.executeSql(sinkDDL)

//注册表
val mystudent=bsTableEnv.from("student")

//执行SQL查询（这个知识点会在第7.1.5节介绍）
val result=bsTableEnv.sqlQuery(s"select sno,cno,grade from $mystudent")

//打印输出
result.toRetractStream[(String,String,Int)].print()

    //把数据插入到student中
table1.executeInsert("student")

//触发程序执行
    bsEnv.execute("MySQLConnector")
  }
}
```

此外，需要在pom.xml文件中额外添加如下两个依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-jdbc_2.12</artifactId>
    <version>1.11.2</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.40</version>
</dependency>
```

7.1.5 查询表

1.Table API

Table API是基于Table类的，该类表示一个表（流或批处理），并提供使用关系操作的方法。这些方法返回一个新的Table对象，该对象表示对输入Table进行关系操作的结果。 一些关系操作由多个方法调用组成，例如table.groupBy(...).select(...)，其中，groupBy(...)指定table的分组，而select(...) 则是在 table 分组上进行投影操作。

7.1.5 查询表

下面是一个简单的Table API聚合查询实例：

```
//获取TableEnvironment
val tableEnv = ...

//注册一个表，名称为Orders

//扫描注册的Orders表
val orders = tableEnv.from("Orders")

//计算来自法国的所有顾客的收益
val revenue = orders
  .filter($"cCountry" === "FRANCE")
  .groupBy($"cID", $"cName")
  .select($"cID", $"cName", $"revenue".sum AS "revSum")

//执行表的转换
//执行查询
```

7.1.5 查询表

2.SQL

Flink SQL是基于Apache Calcite实现的。Calcite是为不同计算平台和数据源提供统一动态数据管理服务的高层框架。Calcite在各种数据源上构建了标准的SQL语言，并提供多种查询优化方案，而且，Calcite引擎也适用于流处理场景。Calcite的目标是为不同计算平台和数据源提供统一的查询引擎，并以SQL语言访问不同数据源。

Calcite执行SQL查询的主要步骤如下；
（1）将SQL解析成未经校验的抽象语法树，抽象语法树是和语言无关的形式；
（2）验证抽象语法树，主要验证SQL语句是否合法，验证后的结果是RelNode树；
（3）优化RelNode树并生成物理查询计划；
将物理执行计划转换成特定平台的执行代码，如Flink的DataStream应用程序。

7.1.5 查询表

下面的示例演示了如何指定查询并将结果作为表对象返回：

```
//获取TableEnvironment
val tableEnv = ...

//注册一个表，名称为Orders

//计算来自法国的所有顾客的收益
val revenue = tableEnv.sqlQuery("""
  |SELECT cID, cName, SUM(revenue) AS revSum
  |FROM Orders
  |WHERE cCountry = 'FRANCE'
  |GROUP BY cID, cName
  """.stripMargin)

//执行表的转换
//执行查询
```

7.1.5 查询表

3.应用实例

假设已经存在一个文本文件“/home/hadoop/stockprice.txt”，文件内容与7.1.4节中的相同。下面我们编写一个程序，这个程序分别使用了Table API和SQL进行查询操作。

```
package cn.edu.xmu.dblab

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

7.1.5 查询表

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)

    //获取EnvironmentSettings
    val bsSettings = EnvironmentSettings
      .newInstance()
      .useBlinkPlanner()
      .inStreamingMode()
      .build()

    //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.1.5 查询表

```
//创建数据源
 val stockTable = bsTableEnv.connect(
   new FileSystem()
     .path("file:///home/hadoop/stockprice.txt")
 ).withFormat(new Csv())
   .withSchema(new Schema()
     .field("stockId", DataTypes.STRING())
     .field("timeStamp", DataTypes.BIGINT())
     .field("price", DataTypes.DOUBLE())
   ).createTemporaryTable("stocktable")

 //使用Table API查询
 val stock = bsTableEnv.from("stocktable")
val stock1 = stock.select($"stockId",$"price").filter('stockId==="stock_1")
```

7.1.5 查询表

```
  //注册表
bsTableEnv.createTemporaryView("stockSQLTable",stock)

   //设置SQL语句
   val sql=
     """
        |select stockId,price from stockSQLTable
        |where stockId='stock_2'
        |""".stripMargin

   //执行SQL查询
val stock2=bsTableEnv.sqlQuery(sql)
//打印输出
   stock1.toAppendStream[(String, Double)].print("stock_1")
stock2.toAppendStream[(String,Double)].print("stock_2")
   //程序触发执行
   bsEnv.execute("TableAPIAndSQLTest")
  }
}
```

7.1.5 查询表

程序执行以后会输出如下结果：

```
stock_1> (stock_1,22.9)
stock_2> (stock_2,43.5)
stock_1> (stock_1,22.2)
stock_2> (stock_2,42.1)
```

7.1.6 输出表

表通过写入TableSink来实现输出。TableSink是一个通用接口，用于支持多种文件格式（如CSV、Apache Parquet、Apache Avro等）、存储系统（如JDBC、Apache HBase、Apache Cassandra、Elasticsearch等）或消息队列系统（如 Apache Kafka、RabbitMQ等）。

下面的程序框架演示了如何输出到CSV文件中：

```
//获取TableEnvironment
val tableEnv = ...

//创建一个输出表
val schema = new Schema()
    .field("a", DataTypes.INT())
    .field("b", DataTypes.STRING())
    .field("c", DataTypes.BIGINT())

tableEnv.connect(new FileSystem().path("/path/to/file"))
    .withFormat(new Csv().fieldDelimiter('|').deriveSchema())
    .withSchema(schema)
    .createTemporaryTable("CsvSinkTable")
//使用Table API或者SQL计算一个结果表
val result: Table = ...
//把结果表写入到已经注册的TableSink
result.executeInsert("CsvSinkTable")
```

这里给出一个具体的程序实例，代码如下：

```
package cn.edu.xmu.dblab

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._

case class StockPrice(stockId:String,timeStamp:Long,price:Double)

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)
```

```
//获取EnvironmentSettings
val bsSettings = EnvironmentSettings
  .newInstance()
  .useBlinkPlanner()
  .inStreamingMode()
  .build()

//获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

```
//创建数据源
val stockTable = bsTableEnv.connect(
  new FileSystem()
    .path("file:///home/hadoop/stockprice.csv")
).withFormat(new Csv())
  .withSchema(new Schema()
    .field("stockId", DataTypes.STRING())
    .field("timeStamp", DataTypes.BIGINT())
    .field("price", DataTypes.DOUBLE())
  ).createTemporaryTable("stocktable")

//使用Table API查询
val stock = bsTableEnv.from("stocktable")
val stock1 = stock.select("stockId,price").filter('stockId==="stock_1")
```

```
//创建一个输出表
val schema = new Schema()
  .field("stockId", DataTypes.STRING())
  .field("price", DataTypes.DOUBLE())

bsTableEnv.connect(new FileSystem().path("file:///home/hadoop/output.csv"))
  .withFormat(new Csv().fieldDelimiter('|').deriveSchema())
  .withSchema(schema)
  .createTemporaryTable("CsvSinkTable")

//把查询结果stock1发送给已经注册的TableSink
stock1.executeInsert("CsvSinkTable")

//打印输出
stock1.toAppendStream[(String, Double)].print("stock_1")

//程序触发执行
bsEnv.execute("TableSinkTest")
  }
}
```

上面程序中，stockprice.txt的文件内容和7.1.4节的相同。程序执行以后，可以看到在本地文件系统中生成了一个output.csv文件，文件里面的内容如下：

```
stock_1|22.9
stock_1|22.2
```

7.1.7 DataStream/DataSet与Table的相互转换

Flink提供了两种计划器（Planner），即Flink原生计划器和Blink计划器。两套方案都可以和DataStream API集成，也就是说，可以将一个DataStream转换成一个Table，也可以将Table转换成DataStream。只有原生的方案可以和DataSet API集成。Blink 在基于批数据时，不能与流数据合并处理。注意，下面关于DataSet的讨论都是对基于批处理的原生方案进行的。

7.1.7 DataStream/DataSet与Table的相互转换

1.通过DataSet或DataStream创建视图

在TableEnvironment中可以将DataStream或DataSet注册成视图。结果视图的模式（Schema）取决于注册的 DataStream 或 DataSet 的数据类型。需要注意的是，通过 DataStream或DataSet创建的视图只能注册成临时视图。

7.1.7 DataStream/DataSet与Table的相互转换

下面是一个具体实例：

```
// 获取TableEnvironment

val tableEnv: StreamTableEnvironment = ...

//创建一个DataStream
val stream: DataStream[(Long, String)] = ...

// 把这个DataStream注册成为视图"myTable"，视图的两个字段是"f0"和"f1"
tableEnv.createTemporaryView("myTable", stream)

// 把这个DataStream注册成为视图"myTable2"，视图的两个字段是
"myLong"和"myString"
tableEnv.createTemporaryView("myTable2", stream, 'myLong, 'myString)
```

7.1.7 DataStream/DataSet与Table的相互转换

2.将DataStream或DataSet转换成表

与在TableEnvironment中注册DataStream或DataSet不同，DataStream和DataSet还可以直接转换成表。如果我们想在Table API的查询中使用表，这种方式是非常便捷的。

下面是一个具体实例：

```
//获取TableEnvironment
val tableEnv = ...

//创建一个DataStream
val stream: DataStream[(Long, String)] = ...

//把一个DataStream转换成一个表，表的默认字段是"_1"和"_2"
val table1: Table = tableEnv.fromDataStream(stream)

//把这个DataStream注册成为表，表的两个字段是"myLong"和"myString"
val table2: Table = tableEnv.fromDataStream(stream, $"myLong", $"myString")
```

7.1.7 DataStream/DataSet与Table的相互转换

3.将表转换成 DataStream 或 DataSet

Table可以被转换成DataStream或DataSet。通过这种方式，定制的DataSet或DataStream 程序就可以在Table API或者SQL的查询结果上运行了。将表转换为DataStream或者DataSet时，需要指定生成的DataStream或者DataSet的数据类型，即Table的每行数据要转换成的数据类型。通常最方便的选择是转换成Row。

（1）将表转换成 DataStream

流式查询（Streaming Query）的结果表会动态更新，即当新记录到达查询的输入流时，查询结果会改变。因此，像这样将动态查询结果转换成DataStream需要对表的更新方式进行编码。

7.1.7 DataStream/DataSet与Table的相互转换

将表转换为 DataStream有两种模式：
（a）Append Mode: 仅当动态表通过INSERT进行修改时，才可以使用此模式，即它仅是追加操作，并且之前输出的结果永远不会更新。
（b）Retract Mode: 任何情形都可以使用此模式。它使用Boolean值对INSERT和DELETE 操作的数据进行标记。

```
//获取TableEnvironment
val tableEnv: StreamTableEnvironment = ...

//创建一个具有两个字段的表， (String name, Integer age)
val table: Table = ...

//把表转换成一个DataStream，每个元素类型为Row
val dsRow: DataStream[Row] = tableEnv.toAppendStream[Row](table)

//把表转换成一个DataStream，每个元素类型为Tuple2[String, Int]
val dsTuple: DataStream[(String, Int)] =
tableEnv.toAppendStream[(String, Int)](table)
```

7.1.7 DataStream/DataSet与Table的相互转换

（2）将表转换成DataSet

将表转换成 DataSet 的过程如下：

```
//获取TableEnvironment
val tableEnv = BatchTableEnvironment.create(env)

//创建一个具有两个字段的表，(String name, Integer age)
val table: Table = ...

//把表转换成一个DataSet，每个元素类型为Row
val dsRow: DataSet[Row] = tableEnv.toDataSet[Row](table)

//把表转换成一个DataSet，每个元素类型为Tuple2[String, Int]
val dsTuple: DataSet[(String, Int)] = tableEnv.toDataSet[(String, Int)](table)
```

3.从数据类型到表模式的映射

表可以由DataStream或者DataSet转换而来，但是，表中的模式和DataStream/DataSet的字段有时候并不是完全匹配的，通常情况下，需要在创建表的时候，修改字段的映射关系。Flink提供了两种从数据类型到表模式的映射，一种是基于字段位置，另一种是基于字段名称。

7.1.7 DataStream/DataSet与Table的相互转换

（1）基于字段位置的映射

基于字段位置的映射，是根据数据集中字段位置偏移来确认表中的字段。这种映射方式可以在保持字段顺序的同时，为字段提供更有意义的名称。

```
//获取TableEnvironment
val tableEnv: StreamTableEnvironment = ...

//创建数据集
val stream: DataStream[(Long, Int)] = ...

//把DataStream转换成表，使用默认的字段名称"_1"和"_2"
val table: Table = tableEnv.fromDataStream(stream)

//把DataStream转换成表，只使用一个字段名称 "myLong"
val table: Table = tableEnv.fromDataStream(stream, $"myLong")

//把DataStream转换成表，使用两个字段名称 "myLong"和"myInt"
val table: Table = tableEnv.fromDataStream(stream, $"myLong", $"myInt")
```

7.1.7 DataStream/DataSet与Table的相互转换

（2）基于字段名称的映射

字段名称映射是指在DataStream或DataSet数据集中，使用数据中的字段名称进行映射。与使用偏移位置相比，字段名称映射更加灵活，适用于包括自定义POJO类的所有数据类型。映射中的所有字段均按名称引用，并且可以通过as重命名。字段可以被重新排序和映射。如果没有指定任何字段名称，则使用默认的字段名称和复合数据类型的字段顺序。

7.1.7 DataStream/DataSet与Table的相互转换

```
//获取TableEnvironment
val tableEnv: StreamTableEnvironment = ...

//创建数据集
val stream: DataStream[(Long, Int)] = ...

//把DataStream转换成表，使用默认字段名称"_1"和"_2"
val table: Table = tableEnv.fromDataStream(stream)

//把DataStream转换成表，只使用一个字段名称"_2"
val table: Table = tableEnv.fromDataStream(stream, $"_2")

//把DataStream转换成表，并交换两个字段的顺序
val table: Table = tableEnv.fromDataStream(stream, $"_2", $"_1")

//把DataStream转换成表，交换两个字段的顺序，并且重命名为"myInt"和"myLong"
val table: Table = tableEnv.fromDataStream(stream, $"_2" as "myInt", $"_1" as "myLong")
```

7.1.7 DataStream/DataSet与Table的相互转换

4.原子数据类型

Flink将基础数据类型（Integer、Double、String）或者通用数据类型（不可再拆分的数据类型）视为原子类型。原子类型的DataStream或者DataSet会被转换成只有一条属性的表。属性的数据类型可以由原子类型推断出，还可以重新命名属性。

```
//获取TableEnvironment
val tableEnv: StreamTableEnvironment = ...

//创建数据集
val stream: DataStream[Long] = ...

//把DataStream转变成表，使用默认的字段名称"f0"
val table: Table = tableEnv.fromDataStream(stream)

//把DataStream转变成表，使用字段名称"myLong"
val table: Table = tableEnv.fromDataStream(stream, $"myLong")
```

7.1.7 DataStream/DataSet与Table的相互转换

5.Tuple类型和Case Class类型

Flink支持Scala的内置Tuple类型，Tuple类型的DataStream和DataSet都能被转换成表。可以通过提供所有字段名称来重命名字段（基于位置映射）。如果没有指明任何字段名称，则会使用默认的字段名称。如果引用了原始字段名称（即_1、_2 … …），则 API 会假定映射是基于名称的而不是基于位置的。基于名称的映射可以通过 as 对字段和投影进行重新排序。

```
//获取TableEnvironment
val tableEnv: StreamTableEnvironment = ...

//创建数据集
val stream: DataStream[(Long, String)] = ...

//把DataStream转换成表，使用重命名的默认名称"_1"和"_2"
val table: Table = tableEnv.fromDataStream(stream)

//把DataStream转换成表，使用字段名称"myLong"和"myString" (基于位置)
val table: Table = tableEnv.fromDataStream(stream, $"myLong", $"myString")
```

7.1.7 DataStream/DataSet与Table的相互转换

```
//把DataStream转换成表，使用重新排序的字段"_2"和"_1" (基于名称)
val table: Table = tableEnv.fromDataStream(stream, $"_2", $"_1")

//把DataStream转换成表，使用映射后的字段"_2" (基于名称)
val table: Table = tableEnv.fromDataStream(stream, $"_2")

//把DataStream转换成表，使用重新排序和重新命名的字段"myString"和
"myLong" (基于名字)
val table: Table = tableEnv.fromDataStream(stream, $"_2" as "myString", $"_1"
as "myLong")
```

7.1.7 DataStream/DataSet与Table的相互转换

```
//定义Case Class
case class Person(name: String, age: Int)
val streamCC: DataStream[Person] = ...

//把DataStream转变成表使用默认的字段名字'name和'age
val table = tableEnv.fromDataStream(streamCC)

//把DataStream转变成表，使用字段名称'myName和'myAge (基于位置)
val table = tableEnv.fromDataStream(streamCC, $"myName", $"myAge")

//把DataStream转换成表，使用重新排序和重新命名的字段"myAge"和
"myName" (基于名称)
val table: Table = tableEnv.fromDataStream(stream, $"age" as "myAge",
$"name" as "myName")
```

7.1.7 DataStream/DataSet与Table的相互转换

6.POJO 类型

Flink支持POJO 类型作为复合类型。在不指定字段名称的情况下将POJO类型的 DataStream或DataSet转换成Table时，将使用原始POJO类型字段的名称。名称映射需要原始名称，并且不能按位置进行。字段可以使用别名（带有 as 关键字）来重命名，重新排序和投影。

```
// 获取TableEnvironment
val tableEnv: StreamTableEnvironment = ...

// Person是一个POJO对象，具有字段名称"name"和"age"
val stream: DataStream[Person] = ...

// 把DataStream转变成表，使用默认的字段名称"age"和"name" (字段根据名称进行排序)
val table: Table = tableEnv.fromDataStream(stream)
```

7.1.7 DataStream/DataSet与Table的相互转换

```
// 把DataStream转变成表，使用重命名字段"myAge"和"myName" (基于名称)
val table: Table = tableEnv.fromDataStream(stream, $"age" as "myAge", $"name" as "myName")

// 把DataStream转换成表，使用映射后的字段名称"name" (基于名称)
val table: Table = tableEnv.fromDataStream(stream, $"name")

// 把DataStream转变成表，使用映射后的和重命名的字段"myName" (基于名称)
val table: Table = tableEnv.fromDataStream(stream, $"name" as "myName")
```

7.1.7 DataStream/DataSet与Table的相互转换

7.Row类型

Row类型支持任意数量的字段以及具有null值的字段。字段名称可以通过RowTypeInfo指定，也可以在将Row的DataStream或DataSet转换为Table时指定。Row类型的字段映射支持基于名称和基于位置两种方式。字段可以通过提供所有字段的名称的方式重命名（基于位置映射）或者分别选择进行投影/排序/重命名（基于名称映射）。

7.1.7 DataStream/DataSet与Table的相互转换

```
//获取TableEnvironment
val tableEnv: StreamTableEnvironment = ...

// Row类型的DataStream，具有两个字段"name"和"age" ， 字段由`RowTypeInfo`声明
val stream: DataStream[Row] = ...

// 把DataStream转变成表，使用默认的字段名称"name"和"age"
val table: Table = tableEnv.fromDataStream(stream)

// 把DataStream转变成表，使用重命名的字段名称"myName"和"myAge" (基于位置)
val table: Table = tableEnv.fromDataStream(stream, $"myName", $"myAge")
```

7.1.7 DataStream/DataSet与Table的相互转换

```
// 把DataStream转变成表，使用重命名的字段"myName"和"myAge" (基于名称)
val table: Table = tableEnv.fromDataStream(stream, $"name" as "myName", $"age" as "myAge")

// 把DataStream转变成表，使用映射后的字段"name" (基于名称)
val table: Table = tableEnv.fromDataStream(stream, $"name")

// 把DataStream转变成表，使用映射后的并且重命名的字段"myName" (基于名称)
val table: Table = tableEnv.fromDataStream(stream, $"name" as "myName")
```

7.1.7 DataStream/DataSet与Table的相互转换

8.应用实例

这里给出一个简单的Table API和SQL数据处理应用程序。假设已经存在一个文本文件“/home/hadoop/stockprice.txt”，其内容与7.1.4节中的相同。
　下面我们编写一个程序，这个程序分别使用了Table API和SQL进行查询操作。程序内容如下：

```
package cn.edu.xmu.dblab

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

7.1.7 DataStream/DataSet与Table的相互转换

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)

    //获取EnvironmentSettings
val bsSettings = EnvironmentSettings
      .newInstance()
      .useBlinkPlanner()
      .inStreamingMode()
      .build()

    //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.1.7 DataStream/DataSet与Table的相互转换

```
    //创建数据源
val inputData=bsEnv.readTextFile("file:///home/hadoop/stockprice.txt")

    //设置对数据集的转换操作逻辑
val dataStream=inputData.map(line=>{
      val arr=line.split(",")
      StockPrice(arr(0),arr(1).toLong,arr(2).toDouble)
})

    //从DataStream生成表
val
stockTable=bsTableEnv.fromDataStream(dataStream,$"stockId",$"timeStam
p",$"price")

    //使用Table API查询
val stock1=stockTable.select($"stockId",$ "price").filter('stockId==="stock_1")

    //注册表
bsTableEnv.createTemporaryView("stocktable",stockTable)
```

7.1.7 DataStream/DataSet与Table的相互转换

```
    //设置SQL语句
val sql=
      """
        |select stockId,price from stocktable
        |where stockId='stock_2'
        |""".stripMargin

    //执行SQL查询
val stock2=bsTableEnv.sqlQuery(sql)

    //把结果打印输出
stock1.toAppendStream[(String,Double)].print("stock_1")
stock2.toAppendStream[(String,Double)].print("stock_2")

//程序触发执行
    bsEnv.execute("TableAPIAndSQLDemo ")
  }
}
```

7.1.7 DataStream/DataSet与Table的相互转换

程序执行后会输出如下结果：

```
stock_2> (stock_2,43.5)
stock_1> (stock_1,22.9)
stock_2> (stock_2,42.1)
stock_1> (stock_1,22.2)
```

7.1.8 时间概念

对于在Table API和SQL接口中的算子，其中部分需要依赖于时间属性，例如GroupBy Windows类算子等，因此，对于这类算子需要在表模式中指定时间属性。

1.事件时间的指定

在DataStream到Table转换时定义事件时间的方法如下：

```
//方案1

//基于stream中的事件产生时间戳和水位线
val stream: DataStream[(String, String)] =
inputStream.assignTimestampsAndWatermarks(...)

//声明一个额外的逻辑字段作为事件时间属性
val table = tEnv.fromDataStream(stream, $"user_name", $"data",
$"user_action_time".rowtime)
```

7.1.8 时间概念

```
//方案2

//从第一个字段获取事件时间，并且产生水位线
val stream: DataStream[(Long, String, String)] =
inputStream.assignTimestampsAndWatermarks(...)

//第一个字段已经用作事件时间抽取了，不用再用一个新字段来表示事件时间了
val table = tEnv.fromDataStream(stream, $"user_action_time".rowtime,
$"user_name", $"data")

//使用方法

val windowedTable = table.window(Tumble over 10.minutes on
$"user_action_time" as "userActionWindow")
```

7.1.8 时间概念

2.处理时间的指定

在 DataStream 到 Table 转换时定义处理时间的方法如下：

```
val stream: DataStream[(String, String)] = ...

//声明一个额外的字段作为时间属性字段
val table = tEnv.fromDataStream(stream, $"UserActionTimestamp",
$"user_name", $"data", $"user_action_time".proctime)

val windowedTable = table.window(Tumble over 10.minutes on
$"user_action_time" as "userActionWindow")
```

7.2 Flink Table API

7.2.1 Table API应用实例
7.2.2 扫描、投影和过滤
7.2.3 列操作
7.2.4 聚合操作
7.2.5 连接操作
7.2.6 集合操作
7.2.7 排序操作
7.2.8 插入操作
7.2.9 基于行的操作

7.2.1 Table API应用实例

对单词进行词频统计并打印输出，具体程序代码如下：

```
package cn.edu.xmu.dblab

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._

import scala.collection.mutable

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)
```

7.2.1 Table API应用实例

```
    //获取EnvironmentSettings
    val bsSettings = EnvironmentSettings
      .newInstance()
      .useBlinkPlanner()
      .inStreamingMode()
      .build()

    //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)

    //生成测试数据
val data = Seq("Flink", "Spark", "HBase", "Spark", "Hadoop", "Flink", "Hive")

    //创建数据源
val source = bsEnv.fromCollection(data).toTable(bsTableEnv, 'word)
```

7.2.1 Table API应用实例

```
    //单词统计核心逻辑
    val result = source
      .groupBy('word)  //单词分组
      .select('word, 'word.count)  //单词统计

    //打印输出计算结果
result.toRetractStream[(String, Long)].print()

    //程序触发执行
    bsEnv.execute
  }
}
```

7.2.1 Table API应用实例

程序执行以后的输出结果如下：

```
(true,(Flink,1))
(true,(Spark,1))
(true,(HBase,1))
(false,(Spark,1))
(true,(Spark,2))
(true,(Hadoop,1))
(false,(Flink,1))
(true,(Flink,2))
(true,(Hive,1))
```

7.2.2 扫描、投影和过滤

1.from

from的用法和SQL中的FROM从句的用法类似，用于对一个已经注册的表的扫描。具体用法如下：

```
val stock: Table = tableEnv.from("stocktable")
```

其中，stocktable是系统中已经注册的表名称

2.fromValues

fromValues的用法和SQL中的VALUES从句的用法类似，它会从用户提供的行中生成一个表。具体用法如下：

```
val table = tableEnv.fromValues(
   row(1, "ABC"),
   row(2L, "ABCDE"))
```

7.2.2 扫描、投影和过滤

3.select

select和SQL语句中的SELECT用法类似，会执行选择操作。具体用法如下：

```
val stock = tableEnv.from("stocktable")
val result = stock.select($"stockId", $"price" as "stockPrice")
```

可以使用“*”选择表中的所有列，具体如下：

```
val stock = tableEnv.from("stocktable")
val result = stock.select($"*")
```

4.as

as用于对字段进行重命名操作，具体如下：

```
val stock = tableEnv.from("stocktable").as("myStockId","myTimeStamp","myPrice")
```

5.where

where和SQL语句中的WHERE从句的用法类似，会执行条件筛选操作。具体如下：

```
val stock = tableEnv.from("stocktable")
val result = stock.where($"stockId" === "stock_1")
```

6.filter

filter用于对表中的行进行过滤操作。具体如下：

```
val stock = tableEnv.from("stocktable")
val result = stock.filter($"stockId" === "stock_1")
```

7.2.3 列操作

1.addColumns

addColumns会为表增加一个列，如果表中已经存在这个列，则会报错。具体如下：

```
val stock = tableEnv.from("stocktable")
val result = stock.addColumns(concat($"stockId", "_good"))
```

增加一个列以后的效果类似如下：

```
(stock_2,1602031562148,43.5,stock_2_good)
(stock_1,1602031562148,22.9,stock_1_good)
(stock_0,1602031562153,8.3,stock_0_good)
(stock_2,1602031562153,42.1,stock_2_good)
(stock_1,1602031562158,22.2,stock_1_good)
```

2.addOrReplaceColumns

addOrReplaceColumns会为表增加一个列，如果表中已经存在同名的列，则表中已经存在的这个列会被替换掉。具体如下：

```
val stock = tableEnv.from("stocktable")
val result = stock.addOrReplaceColumns(concat($"stockId", "_good") as "goodstock")
```

3.dropColumns

dropColumns用于执行列的删除操作，只有已经存在的列才能被删除。具体如下：

```
val stock = tableEnv.from("stocktable")
val result = stock.dropColumns($"price")
```

4.renameColumns

renameColumns用于对列进行重命名。具体如下：

```
val stock = tableEnv.from("stocktable")
val result = stock.renameColumns($"stockId" as "id", $"price" as "stockprice")
```

7.2.4 聚合操作

1. groupBy聚合

groupBy和SQL语句中的GROUPBY从句的用法类似，它会根据分组键对表中的行进行分组。分组以后，就可以用于后续的聚合操作。具体如下：

```
val stock = tableEnv.from("stocktable")
val result = stock.groupBy($"stockId").select($"stockId",
$"price".sum().as("price_sum"))
```

2.基于窗口的groupBy聚合

基于窗口的groupBy聚合和DataStream API、DataSet API中提供的窗口一致，都是将流式数据集根据窗口类型切分成有界数据集，然后在有界数据集上进行聚合类计算。

7.2.4 聚合操作

（1）滚动窗口

对于滚动窗口的情形，可以使用如下方式实现基于窗口的groupBy聚合：

```
val stock = tableEnv.from("stocktable")
val result: Table = stock
    .window(Tumble over 5.seconds() on $"timeStamp" as "w") //定义窗口
    .groupBy($"stockId", $"w") //根据键和窗口进行分组
    .select($"stockId", $"w".start, $"w".end, $"w".rowtime, $"price".sum as "price_sum")
```

其中，over操作符指定窗口的长度，on操作符指定事件时间字段。

7.2.4 聚合操作

（2）滑动窗口

对于滑动窗口的情形，可以使用如下方式实现基于窗口的groupBy聚合：

```
val stock = tableEnv.from("stocktable")
val result: Table = stock
    .window(Slide over 5.seconds() every 1.seconds() on $"timeStamp" as "w") //定义窗口
    .groupBy($"stockId", $"w") //根据键和窗口进行分组
    .select($"stockId", $"w".start, $"w".end, $"w".rowtime, $"price".sum as "price_sum")
```

7.2.4 聚合操作

（3）会话窗口

对于会话窗口的情形，可以使用如下方式实现基于窗口的groupBy聚合：

```
val stock = tableEnv.from("stocktable")
val result: Table = stock
    .window(Session withGap 5.seconds() on $"timeStamp" as "w") //定义窗口
    .groupBy($"stockId", $"w") //根据键和窗口进行分组
    .select($"stockId", $"w".start, $"w".end, $"w".rowtime, $"price".sum as "price_sum")
```

7.2.4 聚合操作

（4）应用实例

下面是运用滚动窗口执行基于窗口的groupBy聚合计算的一个实例：

```
package cn.edu.xmu.cn

import java.text.SimpleDateFormat

import
org.apache.flink.api.common.eventtime.{SerializableTimestampAssign
er, TimestampAssigner, TimestampAssignerSupplier, Watermark,
WatermarkGenerator, WatermarkGeneratorSupplier, WatermarkOutput,
WatermarkStrategy}
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

7.2.4 聚合操作

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

    //获取运行时
    val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置时间特性
bsEnv.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    //设置并行度为1
bsEnv.setParallelism(1)

    //获取EnvironmentSettings
    val bsSettings = EnvironmentSettings
      .newInstance()
      .useBlinkPlanner()
      .inStreamingMode()
      .build()
```

7.2.4 聚合操作

```
    //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)

    //创建数据源
    val source = bsEnv.socketTextStream("localhost", 9999)

    //指定针对数据流的转换操作逻辑
    val stockDataStream = source
      .map(s => s.split(","))
      .map(s=>StockPrice(s(0).toString,s(1).toLong,s(2).toDouble))

    //为数据流分配时间戳和水位线
val watermarkDataStream =
stockDataStream.assignTimestampsAndWatermarks(new
MyWatermarkStrategy)
```

7.2.4 聚合操作

```
    //从DataStream生成表
val
stockTable=bsTableEnv.fromDataStream(watermarkDataStream,$"stockId
",$"timeStamp".rowtime,$"price")

    //使用Table API查询
    val result: Table = stockTable
      .window(Tumble over 5.seconds() on $"timeStamp" as "w") // 定义窗口
      .groupBy($"stockId", $"w") // 根据键和窗口进行分组
      .select($"stockId",$"price".sum as "price_sum")

    //打印输出
result.toRetractStream[(String, Double)].print()

    //程序触发执行
    bsEnv.execute("TableAPIandSQL")
  }
```

7.2.4 聚合操作

```
//指定水位线生成策略
 class MyWatermarkStrategy extends WatermarkStrategy[StockPrice] {
   override def
createTimestampAssigner(context:TimestampAssignerSupplier.Context):Ti
mestampAssigner[StockPrice]={
      new SerializableTimestampAssigner[StockPrice] {
        override def extractTimestamp(element: StockPrice, recordTimestamp:
Long): Long = {
          element.timeStamp //从到达消息中提取时间戳
        }
      }
    }
```

7.2.4 聚合操作

```
    override def createWatermarkGenerator(context:WatermarkGeneratorSupplier.Context): WatermarkGenerator[StockPrice] ={
      new WatermarkGenerator[StockPrice](){
        val maxOutOfOrderness = 10000L //设定最大延迟为10秒
        var currentMaxTimestamp: Long = 0L
        var a: Watermark = null
        val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS")
```

7.2.4 聚合操作

```
        override def onEvent(element: StockPrice, eventTimestamp: Long, output:WatermarkOutput): Unit = {
          currentMaxTimestamp = Math.max(eventTimestamp, currentMaxTimestamp)
          a = new Watermark(currentMaxTimestamp - maxOutOfOrderness)
          output.emitWatermark(a)
          println("timestamp:" + element.stockId + "," + element.timeStamp + "|" + format.format(element.timeStamp) + "," + currentMaxTimestamp + "|" + format.format(currentMaxTimestamp) + "," + a.toString)
        }

        override def onPeriodicEmit(output:WatermarkOutput): Unit = {
          // 没有使用周期性发送水印，因此这里没有执行任何操作
        }
      }
    }
  }
}
```

7.2.4 聚合操作

在Linux终端中，使用如下命令启动NC程序：

```
$ nc -lk 9999
```

然后运行程序GroupByWindowAggregation，再在NC终端内逐行输入如下数据：

```
stock_1,1602031567000,8.14
stock_1,1602031568000,8.22
stock_1,1602031575000,8.14
stock_1,1602031577000,8.14
stock_1,1602031593000,8.14
```

程序运行以后的输出结果如下：

```
(true,(stock_1,16.36))
(true,(stock_1,16.28))
```

7.2.4 聚合操作

3.distinct

distinct和SQL中的DISTINCT从句的作用类似，返回的是具有唯一值的记录。

```
val stock = tableEnv.from("stocktable")
val result = stock.distinct()
```

7.2.5 连接操作

1.内连接

内连接操作和SQL中的JOIN从句的功能类似，会对两个表进行连接。参与连接的两个表必须都存在具有唯一值的字段，并且具有至少一个等值连接谓词。具体实例如下：

```
val left = ds1.toTable(tableEnv, $"a", $"b", $"c")
val right = ds2.toTable(tableEnv, $"d", $"e", $"f")
val result = left.join(right).where($"a" === $"d").select($"a", $"b", $"e")
```

7.2.5 连接操作

2.外连接

外连接操作和SQL中的EFT/RIGHT/FULL OUTER JOIN的功能类似，会对两个表进行连接操作。参与连接的两个表必须都存在具有唯一值的字段，并且具有至少一个等值连接谓词。具体实例如下：

```
val left = tableEnv.fromDataSet(ds1, $"a", $"b", $"c")
val right = tableEnv.fromDataSet(ds2, $"d", $"e", $"f")

val leftOuterResult = left.leftOuterJoin(right, $"a" === $"d").select($"a", $"b", $"e")
val rightOuterResult = left.rightOuterJoin(right, $"a" === $"d").select($"a", $"b", $"e")
val fullOuterResult = left.fullOuterJoin(right, $"a" === $"d").select($"a", $"b", $"e")
```

7.2.6 集合操作

1.union

union操作和SQL中的UNION从句类似，会对两个表进行连接操作，并且去除重复的记录，要求两个表必须具有相同的字段类型。具体实例如下：

```
val left = ds1.toTable(tableEnv, $"a", $"b", $"c")
val right = ds2.toTable(tableEnv, $"a", $"b", $"c")
val result = left.union(right)
```

2.unionAll

unionAll操作和SQL中的UNION ALL从句类似，会对两个表进行连接操作，要求两个表必须具有相同的字段类型。具体实例如下：

```
val left = ds1.toTable(tableEnv, $"a", $"b", $"c")
val right = ds2.toTable(tableEnv, $"a", $"b", $"c")
val result = left.unionAll(right)
```

3.intersect

intersect操作和SQL中的INTERSECT从句类似，对两个表进行intersect操作以后，会返回两个表的交集，也就是在两个表中都存在的记录。如果一个记录在一个表或两个表中出现了两次以上，则在返回的结果中只会出现一次，也就是说，在返回的结果集中不会存在重复的记录。此外，还要求两个表必须具有相同的字段类型。具体实例如下：

```
val left = ds1.toTable(tableEnv, $"a", $"b", $"c")
val right = ds2.toTable(tableEnv, $"e", $"f", $"g")
val result = left.intersect(right)
```

4.intersectAll

intersectAll操作与SQL中的INTERSECT ALL从句类似，会返回两个表的交集，也就是在两个表中都存在的记录。如果一个记录在一个表或两个表中出现了两次以上，则在返回的结果中也会出现相应的次数，也就是说，在返回的结果集中会存在重复的记录。此外，还要求两个表必须具有相同的字段类型。具体实例如下：

```
val left = ds1.toTable(tableEnv, $"a", $"b", $"c")
val right = ds2.toTable(tableEnv, $"e", $"f", $"g")
val result = left.intersectAll(right)
```

5.minus

minus和SQL中的EXCEPT从句类似，它返回的结果是那些在左表中存在、但是右表中不存在的记录。左表中重复的记录，在结果中只会出现一次。此外，还要求两个表必须具有相同的字段类型。具体实例如下：

```
val left = ds1.toTable(tableEnv, $"a", $"b", $"c")
val right = ds2.toTable(tableEnv, $"a", $"b", $"c")
val result = left.minus(right)
```

6.MinusAll

minusAll操作和SQL中的EXCEPT ALL从句类似，它返回的结果是那些在左表中存在、但是右表中不存在的记录。如果一条记录在左表中出现了n次，并且在右表中出现了m次，那么，在返回的结果中会出现n-m次。此外，还要求两个表必须具有相同的字段类型。具体实例如下：

```
val left = ds1.toTable(tableEnv, $"a", $"b", $"c")
val right = ds2.toTable(tableEnv, $"a", $"b", $"c")
val result = left.minusAll(right)
```

7.in

in操作和SQL中的IN从句类似，当一个表达式存在于给定的查询表当中时，in操作会返回true，并且要求这个查询表只能有一个列，而且这个列和这个表达式必须具有相同的数据类型。具体实例如下：

```
val left = ds1.toTable(tableEnv, $"a", $"b", $"c")
val right = ds2.toTable(tableEnv, $"a")
val result = left.select($"a", $"b", $"c").where($"a".in(right))
```

7.2.7 排序操作

orderBy操作和SQL中的ORDER BY从句类似，返回的结果是有序的。具体实例如下：

```
val in = ds.toTable(tableEnv, $"a", $"b", $"c")
val result = in.orderBy($"a".asc)
```

7.2.8 插入操作

executeInsert操作和SQL中的INSERT INTO从句类似，可以向一个已经注册的表中插入记录。需要注意的是，已经注册的输出表的模式必须和查询的模式相匹配。具体实例如下：

```
val stock: Table = bsTableEnv.from("stocktable")
stock.executeInsert("OutStocks")
```

7.2.9 基于行的操作

1.map

执行map操作时，可以使用用户自定义的Scala函数，也可以使用系统内置的Scala函数。具体实例如下：

```
package cn.edu.xmu.dblab

import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._
import org.apache.flink.table.functions.ScalarFunction
import org.apache.flink.types.Row

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

7.2.9 基于行的操作

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)

    //获取EnvironmentSettings
    val bsSettings = EnvironmentSettings
      .newInstance()
      .useBlinkPlanner()
      .inStreamingMode()
      .build()

    //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.2.9 基于行的操作

```
//创建数据源
val stockTable = bsTableEnv.connect(
  new FileSystem()
    .path("file:///home/hadoop/stockprice.txt")
).withFormat(new Csv())
  .withSchema(new Schema()
    .field("stockId", DataTypes.STRING())
    .field("timeStamp", DataTypes.BIGINT())
    .field("price", DataTypes.DOUBLE())
  ).createTemporaryTable("stocktable")

//使用Table API查询
val stock = bsTableEnv.from("stocktable")
val func = new MyMapFunction()
val result = stock.map(func($"stockId")).as("a","b")
result.toRetractStream[(String,String)].print()
```

7.2.9 基于行的操作

```
    //程序触发执行
    bsEnv.execute("TableAPIandSQL")
  }

  class MyMapFunction extends ScalarFunction {
    def eval(a: String): Row = {
      Row.of(a, "my-" + a)
    }
    override def getResultType(signature: Array[Class[_]]): TypeInformation[_] =
      Types.ROW(Types.STRING, Types.STRING())
  }
}
```

程序的执行结果如下：

```
(true,(stock_2,my-stock_2))
(true,(stock_1,my-stock_1))
(true,(stock_0,my-stock_0))
(true,(stock_2,my-stock_2))
(true,(stock_1,my-stock_1))
```

7.2.9 基于行的操作

2.flatMap

这里给出一个flatMap操作的实例，具体如下：

```
package cn.edu.xmu.dblab
import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._
import org.apache.flink.table.functions.{ScalarFunction, TableFunction}
import org.apache.flink.types.Row

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
object TableAPIFlatMap {
  def main(args: Array[String]): Unit = {
    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment
    //设置并行度为1
bsEnv.setParallelism(1)
```

7.2.9 基于行的操作

```
//获取EnvironmentSettings
 val bsSettings = EnvironmentSettings
   .newInstance()
   .useBlinkPlanner()
   .inStreamingMode()
   .build()

 //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.2.9 基于行的操作

```
//创建数据源
 val stockTable = bsTableEnv.connect(
   new FileSystem()
     .path("file:///home/hadoop/stockprice2.txt")
 ).withFormat(new Csv())
   .withSchema(new Schema()
     .field("stockId", DataTypes.STRING())
     .field("timeStamp", DataTypes.BIGINT())
     .field("price", DataTypes.DOUBLE())
   ).createTemporaryTable("stocktable")
```

7.2.9 基于行的操作

```
  //使用Table API查询
   val stock = bsTableEnv.from("stocktable")
   val func = new MyFlatMapFunction()
   val result = stock.flatMap(func($"stockId")).as("a","b")
result.toRetractStream[(String,Int)].print()

   //程序触发执行
   bsEnv.execute("TableAPIandSQL")
  }
```

7.2.9 基于行的操作

```
class MyFlatMapFunction extends TableFunction[Row] {
  def eval(str: String): Unit = {
    if (str.contains("#")) {
      str.split("#").foreach({ s =>
        val row = new Row(2)
        row.setField(0, s)
        row.setField(1, s.length)
        collect(row)
      })
    }
  }
  override def getResultType: TypeInformation[Row] = {
    Types.ROW(Types.STRING, Types.INT)
  }
 }
}
```

7.2.9 基于行的操作

假设stockprice2.txt文件内容如下：

stock#01,1602031567000,8.17
stock#02,1602031568000,8.22
stock#01,1602031575000,8.14

则程序执行结果如下：

(true,(stock,5))
(true,(01,2))
(true,(stock,5))
(true,(02,2))
(true,(stock,5))
(true,(01,2))

7.2.9 基于行的操作

3.聚合

聚合（aggregate）操作会使用聚合函数对表进行操作。需要注意的是，必须在聚合函数后面再跟上select操作，而这个select操作是不支持聚合操作的。具体实例如下：

```
package cn.edu.xmu.dblab

import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.api.java.typeutils.RowTypeInfo
import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._
import org.apache.flink.table.functions.{AggregateFunction, ScalarFunction, Tab
import org.apache.flink.types.Row

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

7.2.9 基于行的操作

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)

    //获取EnvironmentSettings
    val bsSettings = EnvironmentSettings
      .newInstance()
      .useBlinkPlanner()
      .inStreamingMode()
      .build()

    //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.2.9 基于行的操作

```
//创建数据源
val stockTable = bsTableEnv.connect(
  new FileSystem()
    .path("file:///home/hadoop/stockprice.txt")
).withFormat(new Csv())
  .withSchema(new Schema()
    .field("stockId", DataTypes.STRING())
    .field("timeStamp", DataTypes.BIGINT())
    .field("price", DataTypes.DOUBLE())
  ).createTemporaryTable("stocktable")
```

7.2.9 基于行的操作

```
    //使用Table API查询
    val stock = bsTableEnv.from("stocktable")
    val myAggFunc = new MyMinMax()
    val result = stock
      .groupBy($"stockId")
      .aggregate(myAggFunc($"price") as ("x", "y"))
      .select($"stockId", $"x", $"y")
result.toRetractStream[(String,Double,Double)].print()

    //程序触发执行
    bsEnv.execute("TableAPIandSQL")
  }
```

7.2.9 基于行的操作

```
case class MyMinMaxAcc(var min: Double, var max: Double)
class MyMinMax extends AggregateFunction[Row, MyMinMaxAcc] {
  def accumulate(acc: MyMinMaxAcc, value: Double): Unit = {
    if (value < acc.min) {
      acc.min = value
    }
    if (value > acc.max) {
      acc.max = value
    }
  }
  override def createAccumulator(): MyMinMaxAcc = MyMinMaxAcc(0.0, 0.0)
  def resetAccumulator(acc: MyMinMaxAcc): Unit = {
    acc.min = 0.0
    acc.max = 0.0
  }
  override def getValue(acc: MyMinMaxAcc): Row = {
    Row.of(java.lang.Double.valueOf(acc.min), java.lang.Double.valueOf(acc.max))
  }
  override def getResultType: TypeInformation[Row] = {
    new RowTypeInfo(Types.DOUBLE(), Types.DOUBLE())
  }
```

7.2.9 基于行的操作

4.基于分组和窗口的聚合

基于分组和窗口的聚合操作，会对表进行分组和聚合，并且通常会有一个或多个分组键。需要注意的是，必须在聚合函数后面再跟上select操作，而这个select操作是不支持聚合操作的。

```
package cn.edu.xmu.dblab
 import java.text.SimpleDateFormat
import org.apache.flink.api.common.eventtime.{SerializableTimestampAssigner,
TimestampAssigner, TimestampAssignerSupplier, Watermark,
WatermarkGenerator, WatermarkGeneratorSupplier, WatermarkOutput,
WatermarkStrategy}
import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.api.java.typeutils.RowTypeInfo
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.functions.AggregateFunction
import org.apache.flink.types.Row
```

7.2.9 基于行的操作

```
object Test2 {
  def main(args: Array[String]): Unit = {
    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置时间特性
bsEnv.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    //设置并行度为1
bsEnv.setParallelism(1)

    //获取EnvironmentSettings
    val bsSettings = EnvironmentSettings
      .newInstance()
      .useBlinkPlanner()
      .inStreamingMode()
      .build()
    //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.2.9 基于行的操作

```
//创建数据源
val source = bsEnv.socketTextStream("localhost", 9999)

  //指定针对数据流的转换操作逻辑
  val stockDataStream = source
    .map(s => s.split(","))
    .map(s=>StockPrice(s(0).toString,s(1).toLong,s(2).toDouble))

//为数据流分配时间戳和水位线
val watermarkDataStream =
stockDataStream.assignTimestampsAndWatermarks(new
MyWatermarkStrategy)

//从DataStream生成表
  val
stockTable=bsTableEnv.fromDataStream(watermarkDataStream,$"stockId",$"ti
meStamp".rowtime,$"price")
```

7.2.9 基于行的操作

```
 //使用Table API查询
   val myAggFunc = new MyMinMax()
   val result = stockTable
     .window(Tumble over 5.seconds on $"timeStamp" as "w")
     .groupBy($"stockId",$"w")
     .aggregate(myAggFunc($"price") as ("x", "y"))
     .select($"stockId", $"x", $"y")

   //打印输出
result.toRetractStream[(String,Double,Double)].print()

   //程序触发执行
   bsEnv.execute("TableAPIandSQL")
 }

 case class MyMinMaxAcc(var min: Double, var max: Double)
```

7.2.9 基于行的操作

```
class MyMinMax extends AggregateFunction[Row, MyMinMaxAcc] {

  def accumulate(acc: MyMinMaxAcc, value: Double): Unit = {
    if (value < acc.min) {
      acc.min = value
    }
    if (value > acc.max) {
      acc.max = value
    }
  }

  override def createAccumulator(): MyMinMaxAcc = MyMinMaxAcc(0.0, 0.0)

  def resetAccumulator(acc: MyMinMaxAcc): Unit = {
    acc.min = 0.0
    acc.max = 0.0
  }
```

7.2.9 基于行的操作

```
override def getValue(acc: MyMinMaxAcc): Row = {
    Row.of(java.lang.Double.valueOf(acc.min), java.lang.Double.valueOf(acc.max))
  }

  override def getResultType: TypeInformation[Row] = {
    new RowTypeInfo(Types.DOUBLE(), Types.DOUBLE())
  }
}
```

7.2.9 基于行的操作

```
//指定水位线生成策略
 class MyWatermarkStrategy extends WatermarkStrategy[StockPrice] {
   override def
createTimestampAssigner(context:TimestampAssignerSupplier.Context):TimestampAssigner[StockPrice]={
     new SerializableTimestampAssigner[StockPrice] {
       override def extractTimestamp(element: StockPrice, recordTimestamp: Long): Long = {
         element.timeStamp //从到达消息中提取时间戳
       }
     }
   }
```

7.2.9 基于行的操作

```
    override def
createWatermarkGenerator(context:WatermarkGeneratorSupplier.Context):
WatermarkGenerator[StockPrice] ={
      new WatermarkGenerator[StockPrice](){
        val maxOutOfOrderness = 10000L //设定最大延迟为10秒
        var currentMaxTimestamp: Long = 0L
        var a: Watermark = null
        val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS")

        override def onEvent(element: StockPrice, eventTimestamp: Long,
output:WatermarkOutput): Unit = {
          currentMaxTimestamp = Math.max(eventTimestamp, currentMaxTimestamp)
          a = new Watermark(currentMaxTimestamp - maxOutOfOrderness)
          output.emitWatermark(a)
          println("timestamp:" + element.stockId + "," + element.timeStamp + "|" +
format.format(element.timeStamp) + "," + currentMaxTimestamp + "|" +
format.format(currentMaxTimestamp) + "," + a.toString)
        }
```

7.2.9 基于行的操作

```
        override def onPeriodicEmit(output:WatermarkOutput): Unit = {
          // 没有使用周期性发送水印，因此这里没有执行任何操作
        }
      }
    }
  }
}
```

7.2.9 基于行的操作

输入数据如下：

```
stock_1,1602031567000,8.14
stock_2,1602031568000,18.22
stock_2,1602031575000,8.14
stock_1,1602031577000,18.21
stock_1,1602031593000,8.98
```

7.2.9 基于行的操作

程序执行的输出结果如下：

timestamp:stock_1,1602031567000|2020-10-07 08:46:07.000,1602031567000|2020-10-07 08:46:07.000,Watermark @ 1602031557000 (2020-10-07 08:45:57.000)
timestamp:stock_2,1602031568000|2020-10-07 08:46:08.000,1602031568000|2020-10-07 08:46:08.000,Watermark @ 1602031558000 (2020-10-07 08:45:58.000)
timestamp:stock_2,1602031575000|2020-10-07 08:46:15.000,1602031575000|2020-10-07 08:46:15.000,Watermark @ 1602031565000 (2020-10-07 08:46:05.000)
timestamp:stock_1,1602031577000|2020-10-07 08:46:17.000,1602031577000|2020-10-07 08:46:17.000,Watermark @ 1602031567000 (2020-10-07 08:46:07.000)
timestamp:stock_1,1602031593000|2020-10-07 08:46:33.000,1602031593000|2020-10-07 08:46:33.000,Watermark @ 1602031583000 (2020-10-07 08:46:23.000)
(true,(stock_1,0.0,8.14))
(true,(stock_2,0.0,18.22))
(true,(stock_1,0.0,18.21))
(true,(stock_2,0.0,8.14))

7.3 Flink SQL

7.3.1 应用实例
7.3.2 数据查询与过滤操作
7.3.3 聚合操作
7.3.4 连接操作
7.3.5 集合操作

7.3.1 应用实例

SELECT语句需要使用TableEnvironment.sqlQuery()方法加以指定。这个方法会以Table的形式返回SELECT的查询结果。Table可以被用于随后的SQL与Table API查询、 转换为DataSet或DataStream或输出到TableSink。SQL与Table API的查询可以进行无缝融合、整体优化并翻译为单一的程序。SELECT语句也可以通过TableEnvironment.executeSql() 方法来执行，将选择的结果收集到本地。该方法返回TableResult对象用于包装查询的结果。

7.3.1 应用实例

```
package cn.edu.xmu.dblab

import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._
import org.apache.flink.table.functions.ScalarFunction
import org.apache.flink.types.Row

case class StockPrice(stockId:String,timeStamp:Long,price:Double)

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)
```

7.3.1 应用实例

```
//获取EnvironmentSettings
   val bsSettings = EnvironmentSettings
     .newInstance()
     .useBlinkPlanner()
     .inStreamingMode()
     .build()

   //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.3.1 应用实例

```
//创建数据源
   val stockTable = bsTableEnv.connect(
     new FileSystem()
       .path("file:///home/hadoop/stockprice.txt")
   ).withFormat(new Csv())
     .withSchema(new Schema()
       .field("stockId", DataTypes.STRING())
       .field("timeStamp", DataTypes.BIGINT())
       .field("price", DataTypes.DOUBLE())
     ).createTemporaryTable("stocktable")
```

7.3.1 应用实例

```
//创建输出表
    val outTable = bsTableEnv.connect(
      new FileSystem()
        .path("file:///home/hadoop/out.txt")
    ).withFormat(new Csv())
      .withSchema(new Schema()
        .field("stockId", DataTypes.STRING())
        .field("price", DataTypes.DOUBLE())
      ).createTemporaryTable("outtable")
```

7.3.1 应用实例

```
//使用SQL查询
    bsTableEnv.executeSql(
      "INSERT INTO outtable SELECT stockId,price FROM stocktable
WHERE stockId LIKE '%stock_1%'")

    //使用SQL查询
    val stock = bsTableEnv.from("stocktable")
    val result = bsTableEnv.sqlQuery(s"SELECT stockId,price FROM
$stock")
    result.toRetractStream[(String,Double)].print()

 //程序触发执行
    bsEnv.execute("TableAPIandSQL")
  }
}
```

7.3.2 数据查询与过滤操作

可以通过SELECT语句查询表中的数据，并使用WHERE语句设定过滤条件，将符合条件的数据筛选出来。

```
SELECT * FROM stock

SELECT stockId, price AS stockprice FROM stock
SELECT * FROM stock WHERE stockId = 'stock_1'
SELECT * FROM stock WHERE price > 10
```

7.3.3 聚合操作

1.GROUP BY聚合

GROUP BY聚合的实例如下：

```
SELECT stockId, AVG(price) as avg_price
FROM stock
GROUP BY stockId
```

2.GROUP BY窗口聚合

GROUP BY窗口聚合的实例如下：

```
SELECT stockId, AVG(price)
FROM stock
GROUP BY TUMBLE(timeStamp, INTERVAL '1' DAY), stockId
```

7.3.3 聚合操作

3.DISTINCT

DISTINCT的具体实例如下：

```
SELECT DISTINCT stockId FROM stock
```

4. HAVING

HAVING的具体实例如下：

```
SELECT AVG(price)
FROM stock
GROUP BY stockId
HAVING AVG(price) > 20
```

7.3.4 连接操作

1.内连接
目前仅支持等值连接，具体实例如下：

```
SELECT *
FROM stock INNER JOIN stock_info ON stock.stockId = stock_info.stockId
```

2.外连接
目前仅支持等值连接，具体实例如下：

```
SELECT * FROM stock LEFT JOIN stock_info ON stock.stockId =
stock_info.stockId

SELECT * FROM stock RIGHT JOIN stock_info ON stock.stockId =
stock_info.stockId

SELECT * FROM stock FULL OUTER JOIN stock_info ON
stock.stockId = stock_info.stockId
```

7.3.5 集合操作

1.UNION

UNION操作的具体实例如下：

```
SELECT *
FROM (
    (SELECT stockId FROM stock WHERE stockId='stock_1')
  UNION
    (SELECT stockId FROM stock WHERE stockId='stock_2'
)
```

7.3.5 集合操作

2.UNION ALL

UNION ALL操作的具体实例如下：

```
SELECT *
FROM (
    (SELECT stockId FROM stock WHERE stockId=’stock_1’)
  UNION ALL
    (SELECT stockId FROM stock WHERE stockId=’stock_2’
)
```

7.3.5 集合操作

3.Intersect/Except

Intersect/Except操作的具体实例如下：

```
SELECT *
FROM (
    (SELECT stockId FROM stock WHERE price > 10.0)
  INTERSECT
    (SELECT stockId FROM stock WHERE stockId='stock_1')
)
SELECT *
FROM (
    (SELECT stockId FROM stock WHERE price > 10.0)
  EXCEPT
    (SELECT stockId FROM stock WHERE stockId='stock_1')
)
```

7.3.5 集合操作

4.IN

若表达式在给定的表查询中存在，则返回true。查询表必须由单个列构成，且该列的数据类型需与表达式保持一致。IN操作的具体实例如下：

```
SELECT stockId, price
FROM stock
WHERE stockId IN (
    SELECT stockId FROM newstock
)
```

7.3.5 集合操作

5.EXISTS

若子查询的结果多于一行，将返回true 。仅支持可以通过join和group重写的操作。具体实例如下：

```
SELECT stockId, price
FROM stock
WHERE stockId EXISTS (
    SELECT stockId FROM newstock
)
```

7.3.5 集合操作

6.ORDER BY

ORDER BY的具体实例如下：

```
SELECT *
FROM stock
ORDER BY timeStamp
```

7. LIMIT

LIMIT的具体用法如下：

```
SELECT *
FROM stock
ORDER BY timeStamp
LIMIT 3
```

7.4 自定义函数

7.4.1 标量函数
7.4.2 表值函数
7.4.3 聚合函数

7.4.1 标量函数

自定义标量函数可以把 0 到多个标量值映射成 1 个标量值，数据类型里列出的任何数据类型都可作为求值方法的参数和返回值类型。想要实现自定义标量函数，我们需要扩展 org.apache.flink.table.functions里面的ScalarFunction并且实现一个或者多个求值方法。标量函数的行为取决于我们写的求值方法。求值方法必须是public的，而且名字必须是eval。对于不支持的输出结果类型，可以通过实现TableFunction接口中的getResultType()对输出结果的数据类型进行转换。

下面给出一个实例，介绍如何创建一个基本的标量函数，以及如何在Table API和SQL里调用这个函数。函数用于SQL查询前要先经过注册；而在用于Table API 时，函数可以先注册后调用，也可以“内联”后直接使用。

7.4.1 标量函数

```
package cn.edu.xmu.dblab

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._
import org.apache.flink.table.functions.ScalarFunction
import org.apache.flink.types.Row

case class StockPrice(stockId:String,timeStamp:Long,price:Double)

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)
```

7.4.1 标量函数

```
//获取EnvironmentSettings
  val bsSettings = EnvironmentSettings
    .newInstance()
    .useBlinkPlanner()
    .inStreamingMode()
    .build()

  //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.4.1 标量函数

```
//创建数据源
   val stockTable = bsTableEnv.connect(
     new FileSystem()
       .path("file:///home/hadoop/stockprice.txt")
   ).withFormat(new Csv())
     .withSchema(new Schema()
       .field("stockId", DataTypes.STRING())
       .field("timeStamp", DataTypes.BIGINT())
       .field("price", DataTypes.DOUBLE())
     ).createTemporaryTable("stocktable")
val stock = bsTableEnv.from("stocktable")
```

7.4.1 标量函数

```
//在 Table API 里不经注册直接“内联”调用函数
val result1 = stock.select(call(classOf[SubstringFunction], $"stockId",6,7))

    //注册函数
bsTableEnv.createTemporarySystemFunction("SubstringFunction",
classOf[SubstringFunction])

    //在Table API里调用注册好的函数
    val result2 = stock.select(call("SubstringFunction", $"stockId",6,7))
val result3 = bsTableEnv.sqlQuery("SELECT SubstringFunction(stockId, 6, 7)
FROM stocktable")

    //打印输出
    result1.toAppendStream[Row].print("result1")
    result2.toAppendStream[Row].print("result2")
    result3.toAppendStream[Row].print("result3")
    bsEnv.execute("ScalarFunctionDemo ")
  }
```

7.4.1 标量函数

```
//用户自定义函数
class SubstringFunction extends ScalarFunction {
  def eval(s: String, begin: Integer, end: Integer): String = {
    s.substring(begin, end)
  }
}
}
```

7.4.2 表值函数

跟自定义标量函数一样，自定义表值函数的输入参数也可以是 0 到多个标量。但是跟标量函数只能返回一个值不同的是，它可以返回任意多行。返回的每一行可以包含 1 到多列，如果输出行只包含 1 列，会省略结构化信息并生成标量值，这个标量值在运行阶段会隐式地包装进行里。

要定义一个表值函数，我们需要扩展org.apache.flink.table.functions下的TableFunction，可以通过实现多个名为eval的方法对求值方法进行重载。像其他函数一样，输入和输出类型也可以通过反射自动提取出来。表值函数返回的表的类型取决于TableFunction类的泛型参数 T，不同于标量函数，表值函数的求值方法本身不包含返回类型，而是通过collect(T)方法来发送要输出的行。

在Table API中，表值函数是通过.joinLateral(...)或者.leftOuterJoinLateral(...)来使用的。joinLateral算子会把外表（算子左侧的表）的每一行跟跟表值函数返回的所有行（位于算子右侧）进行交叉连接（cross join）。leftOuterJoinLateral算子也是把外表（算子左侧的表）的每一行跟表值函数返回的所有行（位于算子右侧）进行交叉连接，并且如果表值函数返回 0行也会保留外表的这一行。

7.4.2 表值函数

```
package cn.edu.xmu.dblab

import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.api.java.typeutils.RowTypeInfo
import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.annotation.{DataTypeHint, FunctionHint}
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._
import org.apache.flink.table.functions.{AggregateFunction,
ScalarFunction, TableFunction}
import org.apache.flink.types.Row

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

7.4.2 表值函数

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)

    //获取EnvironmentSettings
    val bsSettings = EnvironmentSettings
      .newInstance()
      .useBlinkPlanner()
      .inStreamingMode()
      .build()

    //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.4.2 表值函数

```
//创建数据源
   val stockTable = bsTableEnv.connect(
     new FileSystem()
       .path("file:///home/hadoop/stockprice.txt")
   ).withFormat(new Csv())
     .withSchema(new Schema()
       .field("stockId", DataTypes.STRING())
       .field("timeStamp", DataTypes.BIGINT())
       .field("price", DataTypes.DOUBLE())
     ).createTemporaryTable("stocktable")
```

7.4.2 表值函数

```
//使用Table API查询
    val stock = bsTableEnv.from("stocktable")
    val result = stock
      //.joinLateral(call(classOf[MySplitFunction], $"stockId")
       .leftOuterJoinLateral(call(classOf[MySplitFunction], $"stockId"))
       .select($"stockId", $"word", $"length")
result.toRetractStream[(String,String,Long)].print()

    //程序触发执行
    bsEnv.execute("TableFunctionDemo")
  }
```

7.4.2 表值函数

```
//通过注解指定返回类型
 @FunctionHint(output = new DataTypeHint("ROW<word STRING, length INT>"))
 class MySplitFunction extends TableFunction[Row] {
   def eval(str: String): Unit = {
     //使用collect(...)把行发送（emit）出去
     str.split("_").foreach(s => collect(Row.of(s, Int.box(s.length))))
   }
 }
}
```

7.4.3 聚合函数

自定义聚合函数是把一个表（一行或者多行，每行可以有一列或者多列）聚合成一个标量值。自定义聚合函数是通过扩展AggregateFunction来实现的。AggregateFunction的工作过程如下：

•需要一个 累加器（accumulator），它是一个数据结构，存储了聚合的中间结果。通过调用 AggregateFunction 的 createAccumulator() 方法，可以创建一个空的累加器；

•对于每一行数据，会调用 accumulate() 方法来更新累加器。当所有的数据都处理完了之后，通过调用 getValue方法来计算和返回最终的结果。

每个 AggregateFunction 必须要实现3个方法：createAccumulator()、accumulate()和getValue()。对于不支持的输出结果类型，可以通过实现TableFunction接口中的getResultType()对输出结果的数据类型进行转换。

7.4.3 聚合函数

```
package cn.edu.xmu.dblab

import org.apache.flink.api.common.typeinfo.TypeInformation
import org.apache.flink.api.java.typeutils.RowTypeInfo
import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala._
import org.apache.flink.table.api._
import org.apache.flink.table.descriptors._
import org.apache.flink.table.functions.{AggregateFunction,
ScalarFunction, TableFunction}
import org.apache.flink.types.Row

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

7.4.3 聚合函数

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

    //获取运行时
val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment

    //设置并行度为1
bsEnv.setParallelism(1)

    //获取EnvironmentSettings
    val bsSettings = EnvironmentSettings
      .newInstance()
      .useBlinkPlanner()
      .inStreamingMode()
      .build()

    //获取TableEnvironment
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
```

7.4.3 聚合函数

```
//创建数据源
   val stockTable = bsTableEnv.connect(
     new FileSystem()
       .path("file:///home/hadoop/stockprice.txt")
   ).withFormat(new Csv())
     .withSchema(new Schema()
       .field("stockId", DataTypes.STRING())
       .field("timeStamp", DataTypes.BIGINT())
       .field("price", DataTypes.DOUBLE())
     ).createTemporaryTable("stocktable")
```

7.4.3 聚合函数

```
 //使用Table API查询
    val stock = bsTableEnv.from("stocktable")
    val myCountFunction = new MyCountFunction()
    val result = stock
      .groupBy($"stockId")
      .aggregate(myCountFunction() as ("x"))
      .select($"stockId", $"x")
result.toRetractStream[(String,Long)].print()

    //程序触发执行
    bsEnv.execute("SelfAggFunc")
  }
```

7.4.3 聚合函数

```
  case class MyCountAccumulator(var count:Long)

  class MyCountFunction extends AggregateFunction[Row,
MyCountAccumulator] {
    def accumulate(acc: MyCountAccumulator): Unit = {
      acc.count = acc.count + 1
    }

    override def createAccumulator(): MyCountAccumulator =
MyCountAccumulator(0)

    override def getValue(acc: MyCountAccumulator): Row =
Row.of(java.lang.Long.valueOf(acc.count))

    override def getResultType: TypeInformation[Row] = {
      new RowTypeInfo(Types.LONG())
    }
  }
}
```

7.5 本章小结

关系型编程接口，因其强大且灵活的表达能力，能够让用户通过非常丰富的接口对数据进行处理，有效降低了用户的使用成本，近年来逐渐成为主流大数据处理框架主要的接口形式之一。Table API和SQL是Flink提供的关系型编程接口，能够让用户通过使用结构化编程接口高效地构建Flink应用。同时，Table API和SQL能够统一处理批量和实时计算业务，无需切换修改任何应用代码就能够基于同一套API编写流式应用和批量应用，从而达到真正意义上的批流统一。本章详细介绍了如何使用Table API和SQL来构建应用程序。

附录A：主讲教师林子雨简介

主讲教师：林子雨

单位：厦门大学计算机科学系
E-mail: ziyulin@xmu.edu.cn
个人网页：http://dblab.xmu.edu.cn/post/linziyu
数据库实验室网站：http://dblab.xmu.edu.cn

扫一扫访问个人主页

林子雨，男，1978年出生，博士（毕业于北京大学），全国高校知名大数据教师，现为厦门大学计算机科学系副教授，曾任厦门大学信息科学与技术学院院长助理、晋江市发展和改革局副局长。中国计算机学会数据库专业委员会委员，中国计算机学会信息系统专业委员会委员。国内高校首个“数字教师”提出者和建设者，厦门大学数据库实验室负责人，厦门大学云计算与大数据研究中心主要建设者和骨干成员，2013年度、2017年度和2020年度厦门大学教学类奖教金获得者，荣获2019年福建省精品在线开放课程、2018年厦门大学高等教育成果特等奖、2018年福建省高等教育教学成果二等奖、2018年国家精品在线开放课程。主要研究方向为数据库、数据仓库、数据挖掘、大数据、云计算和物联网，并以第一作者身份在《软件学报》《计算机学报》和《计算机研究与发展》等国家重点期刊以及国际学术会议上发表多篇学术论文。作为项目负责人主持的科研项目包括1项国家自然科学青年基金项目(No.61303004)、1项福建省自然科学青年基金项目(No.2013J05099)和1项中央高校基本科研业务费项目(No.2011121049)，主持的教改课题包括1项2016年福建省教改课题和1项2016年教育部产学协作育人项目，同时，作为课题负责人完成了国家发改委城市信息化重大课题、国家物联网重大应用示范工程区域试点泉州市工作方案、2015泉州市互联网经济调研等课题。中国高校首个“数字教师”提出者和建设者，2009年至今，“数字教师”大平台累计向网络免费发布超过1000万字高价值的研究和教学资料，累计网络访问量超过1000万次。打造了中国高校大数据教学知名品牌，编著出版了中国高校第一本系统介绍大数据知识的专业教材《大数据技术原理与应用》，并成为京东、当当网等网店畅销书籍；建设了国内高校首个大数据课程公共服务平台，为教师教学和学生学习大数据课程提供全方位、一站式服务，年访问量超过200万次，累计访问量超过1000万次。

附录B：大数据学习路线图

大数据学习路线图访问地址：http://dblab.xmu.edu.cn/post/10164/

附录C：林子雨大数据系列教材

了解全部教材信息：http://dblab.xmu.edu.cn/post/bigdatabook/

附录D：《大数据导论（通识课版）》教材

开设全校公共选修课的优质教材

本课程旨在实现以下几个培养目标：

- 引导学生步入大数据时代，积极投身大数据的变革浪潮之中
- 了解大数据概念，培养大数据思维，养成数据安全意识
- 认识大数据伦理，努力使自己的行为符合大数据伦理规范要求
- 熟悉大数据应用，探寻大数据与自己专业的应用结合点
- 激发学生基于大数据的创新创业热情

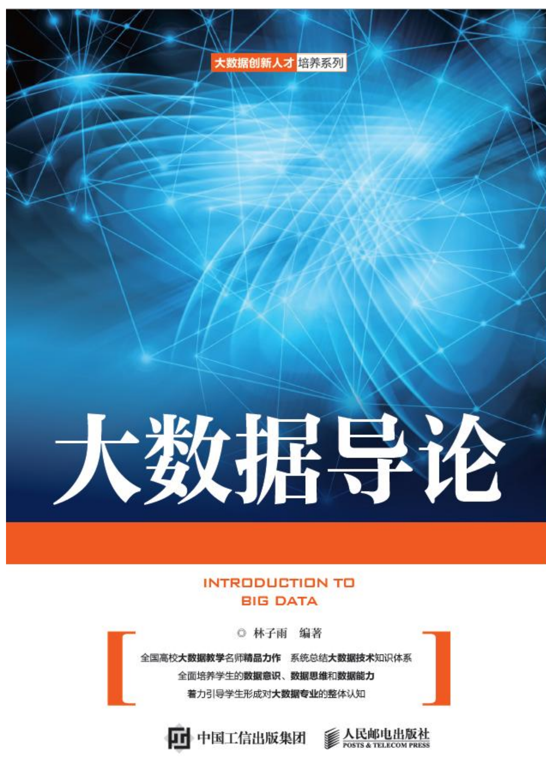
高等教育出版社 ISBN:978-7-04-053577-8 定价：32元 版次：2020年2月第1版

教材官网：http://dblab.xmu.edu.cn/post/bigdataintroduction/

附录E：《大数据导论》教材

- 林子雨 编著《大数据导论》
- 人民邮电出版社，2020年9月第1版
- ISBN:978-7-115-54446-9 定价：49.80元

教材官网：http://dblab.xmu.edu.cn/post/bigdata-introduction/



开设大数据专业导论课的优质教材

扫一扫访问教材官网

附录F：《大数据技术原理与应用（第3版）》教材

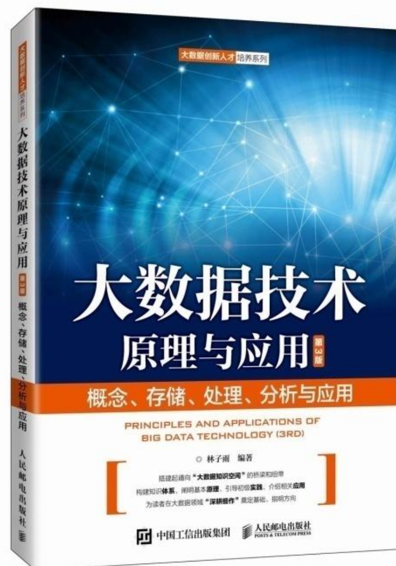
《大数据技术原理与应用——概念、存储、处理、分析与应用（第3版）》，由厦门大学计算机科学系林子雨博士编著，是国内高校第一本系统介绍大数据知识的专业教材。人民邮电出版社 ISBN:978-7-115-54405-6 定价：59.80元

扫一扫访问教材官网

全书共有17章，系统地论述了大数据的基本概念、大数据处理架构Hadoop、分布式文件系统HDFS、分布式数据 库HBase、NoSQL数据库、云数据库、分布式并行编程模型MapReduce、Spark、流计算、Flink、图计算、数据可视化以及大数据在互联网、生物医学和物流等各个领域的应用。在Hadoop、HDFS、HBase、MapReduce、Spark和Flink等重要章节，安排了入门级的实践操作，让读者更好地学习和掌握大数据关键技术。

本书可以作为高等院校计算机专业、信息管理等相关专业的大数据课程教材，也可供相关技术人员参考、学习、培训之用。

欢迎访问《大数据技术原理与应用——概念、存储、处理、分析与应用》教材官方网站：
http://dblab.xmu.edu.cn/post/bigdata3



附录G：《大数据基础编程、实验和案例教程（第2版）》

本书是与《大数据技术原理与应用（第3版）》教材配套的唯一指定实验指导书

大数据教材

1+1黄金组合

厦门大学林子雨编著

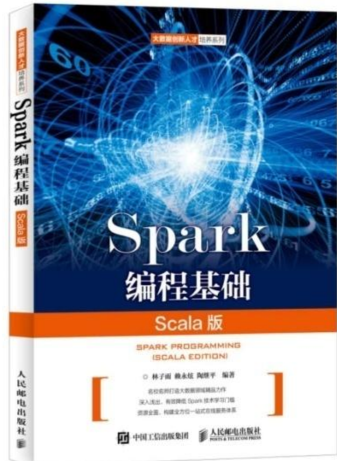
配套实验指导书

- 步步引导，循序渐进，详尽的安装指南为顺利搭建大数据实验环境铺平道路
- 深入浅出，去粗取精，丰富的代码实例帮助快速掌握大数据基础编程方法
- 精心设计，巧妙融合，八套大数据实验题目促进理论与编程知识的消化和吸收
- 结合理论，联系实际，大数据课程综合实验案例精彩呈现大数据分析全流程

林子雨编著《大数据基础编程、实验和案例教程（第2版）》

清华大学出版社 ISBN:978-7-302-55977-1 定价：69元 2020年10月第2版

附录H：《Spark编程基础（Scala版）》



《Spark编程基础（Scala版）》

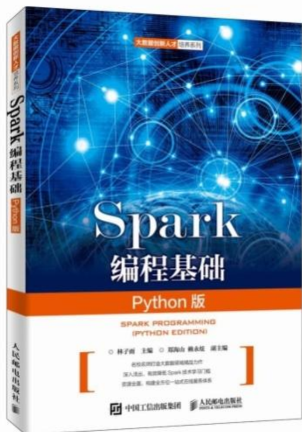
厦门大学 林子雨，赖永炫，陶继平 编著

披荆斩棘，在大数据丛林中开辟学习捷径
填沟削坎，为快速学习Spark技术铺平道路
深入浅出，有效降低Spark技术学习门槛
资源全面，构建全方位一站式在线服务体系

人民邮电出版社出版发行，ISBN:978-7-115-48816-9
教材官网：http://dblab.xmu.edu.cn/post/spark/

本书以Scala作为开发Spark应用程序的编程语言，系统介绍了Spark编程的基础知识。全书共8章，内容包括大数据技术概述、Scala语言基础、Spark的设计与运行原理、Spark环境搭建和使用方法、RDD编程、Spark SQL、Spark Streaming、Spark MLlib等。本书每个章节都安排了入门级的编程实践操作，以便读者更好地学习和掌握Spark编程方法。本书官网免费提供了全套的在线教学资源，包括讲义PPT、习题、源代码、软件、数据集、授课视频、上机实验指南等。

附录I：《Spark编程基础（Python版）》



《Spark编程基础（Python版）》

厦门大学 林子雨，郑海山，赖永炫　编著

披荆斩棘，在大数据丛林中开辟学习捷径
填沟削坎，为快速学习Spark技术铺平道路
深入浅出，有效降低Spark技术学习门槛
资源全面，构建全方位一站式在线服务体系

人民邮电出版社出版发行，ISBN:978-7-115-52439-3

教材官网： http://dblab.xmu.edu.cn/post/spark-python/

本书以Python作为开发Spark应用程序的编程语言，系统介绍了Spark编程的基础知识。全书共8章，内容包括大数据技术概述、Spark的设计与运行原理、Spark环境搭建和使用方法、RDD编程、Spark SQL、Spark Streaming、Structured Streaming、Spark MLlib等。本书每个章节都安排了入门级的编程实践操作，以便读者更好地学习和掌握Spark编程方法。本书官网免费提供了全套的在线教学资源，包括讲义PPT、习题、源代码、软件、数据集、上机实验指南等。

附录J：高校大数据课程公共服务平台

高校大数据课程

公 共 服 务 平 台

http://dblab.xmu.edu.cn/post/bigdata-teaching-platform/

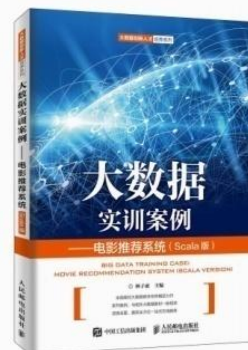
扫一扫访问平台主页

扫一扫观看3分钟FLASH动画宣传片

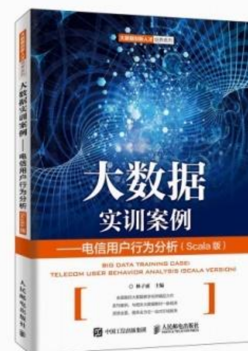
附录K：高校大数据实训课程系列案例教材

为了更好满足高校开设大数据实训课程的教材需求，厦门大学数据库实验室林子雨老师团队联合企业共同开发了《高校大数据实训课程系列案例》，目前已经完成开发的系列案例包括：

《电影推荐系统》（已经于2019年5月出版）
《电信用户行为分析》（已经于2019年5月出版）
《实时日志流处理分析》
《微博用户情感分析》
《互联网广告预测分析》
《网站日志处理分析》



系列案例教材将于2019年陆续出版发行，教材相关信息，敬请关注网页后续更新！
http://dblab.xmu.edu.cn/post/shixunkecheng/



扫一扫访问大数据实训课程系列案例教材主页

Thank You!