精通Apache Flink读书笔记—3、4

3、批处理API

流处理有其价值所在，但很多场景下用不到也没必要使用流处理。有时候，批处理也能发挥很好的作用。Flink支持批处理，而且认为批处理是流处理的一种特殊形式。

这块做下简单的解释，为什么说批是流的特殊情况?

```
流的简单处理形式就是来一条处理一条，但是如果将到达的数据buffer起来，当到达一定的条件时，再

这也是为什么说在Flink中，批是流的一种特殊形式了。
```

这个问题的具体解释，见：Batch is a special case of streaming。

批处理 API的流程和流处理一样，也有获得执行环境、source、transformation、sink等步骤。

3.1、数据源

批处理API的数据源可以是文件或者java collections。DataSet API提供了很多预定义的source函数，当然你也可以自定义数据源。先看看内建的数据源。

3.1.1、基于文件

基于文件的数据源，是一行一行读取，且每行读到的数据作为字符串处理。

```
readTextFile(Stringpath)：默认读取TextInputFormat格式，每行作为一个字符串；

readTextFileWithValue(Stringpath)：返回StringValues，StringValues作为mutable字

readCsvFile(Stringpath)：返回Java POJOS或者tuples；

readFileofPremitives(path, delimiter, class)：解析一行数据到指定的class；

readHadoopFile(FileInputFormat, Key, Value, path)：读取Hadoop文件，指定路径、文

readSequenceFile(Key, Value, path)：读取SequenceFile格式的文件，同样需指定key、v
```

关于读取HDFS文件：

```
// read a file from the specified path of type TextInputFormat
DataSet<Tuple2<LongWritable, Text>> tuples =
 env.readHadoopFile(new TextInputFormat(), LongWritable.class, Text.class, "hdfs://nnHost:nnPort/path/to/file");
```

这里还可以通过JDBC读取关系数据库中的表数据：

```
// Read data from a relational database using the JDBC input format
DataSet<Tuple2<String, Integer> dbData =
    env.createInput(
      // create and configure input format
      JDBCInputFormat.buildJDBCInputFormat()
                     .setDrivername("org.apache.derby.jdbc.EmbeddedDriver")
                     .setDBUrl("jdbc:derby:memory:persons")
                     .setQuery("select name, age from persons")
                     .finish(),
      // specify type information for DataSet
      new TupleTypeInfo(Tuple2.class, STRING_TYPE_INFO, INT_TYPE_INFO)
    );
```

注意：基于文件的数据源，支持递归遍历，循环读取文件中的数据作为数据源。但是我们需要设置recursive.file.enumeration为true以激活此功能。

```
// enable recursive enumeration of nested input files
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// create a configuration object
Configuration parameters = new Configuration();

// set the recursive enumeration parameter
parameters.setBoolean("recursive.file.enumeration", true);

// pass the configuration to the data source
DataSet<String> logs = env.readTextFile("file:///path/with.nested/files")
                          .withParameters(parameters);
```

3.1.2、基于Collection

Flink DataSet API也支持读取java集合中的数据。

```
1  fromCollection(Collection)
2
3  fromCollection(Iterator, Class)：也可以读取iterator，其数据本身的类型为指定的clas
4
5  fromElements(T)：读取sequence对象；
6
7  fromParallelCollection(SplittableIterator, Class)：读取并行iterator；
8
9  generateSequence(from, to)：读取一定范围的sequnce对象。
```

3.1.3、通用数据源

```
1  readFile(inputFormat, path):指定路径，指定FileInputFormat;
```

```
2
3 createInput(inputFormat)
```

3.1.4、压缩文件

Flink支持自动读取压缩文件，扩展名为.gz、.gzip、.deflate的压缩文件。

注意读取压缩文件，不能并行处理，因此加载解压的时间会稍微有点长。

3.2、Transformations

在transformation部分，有些算子操作和流处理的是一样的，这里不做一一介绍。只介绍一些在流处理中没有的操作。

Distinct

可以讲DataSet中的元素去重，这在流处理中无法做到，因为流是无界的，要去重也必须在一定的有界范围内去重，例如窗口。但是目前Flink流处理中还不支持。

```
1 DataSet<Tuple2<Integer, Double>> output = input.distinct();
```

Cross

两个DataSet进行笛卡尔积操作，将会产生非常大的数据集。建议设置DataSet的大小或crossWithTiny() 和crossWithHuge()来限制一个DataSet的大小。轻易不要用。

```
1 DataSet<Integer> data1 = // [...]
2 DataSet<String> data2 = // [...]
3 DataSet<Tuple2<Integer, String>> result = data1.cross(data2);
```

Range partition

根据指定的key，将dataSet范围分片。

```
1 DataSet<Tuple2<String,Integer>> in = // [...]
2 DataSet<Integer> result = in.partitionByRange(0)
3                            .mapPartition(new PartitionMapper());
```

Sort partition

根据key，将dataSet按照key的升序或降序重分片。

```
1 DataSet<Tuple2<String,Integer>> in = // [...]
2 DataSet<Integer> result = in.sortPartition(1, Order.ASCENDING)
3                            .mapPartition(new PartitionMapper());
```

First-n

随机取出dataSet的前10个元素，first-n也可以应用在分组后的数据集上。

```
DataSet<Tuple2<String,Integer>> in = // [...]
// regular data set
DataSet<Tuple2<String,Integer>> result1 = in.first(3);
// grouped data set
DataSet<Tuple2<String,Integer>> result2 = in.groupBy(0)
                                            .first(3);
// grouped-sorted data set
DataSet<Tuple2<String,Integer>> result3 = in.groupBy(0)
                                            .sortGroup(1, Order.ASCENDING)
                                            .first(3);
```

例如应用在sortGroup集合上，first(3)将返回排序后的前3个数据。

3.3、广播变量

广播变量允许用户将特定的DataSet发送到各个节点的内存中。值得注意的是，由于是发送dataSet，因此这个dataSet的大小不能太大。

广播变量的使用主要分为2步：

```
(1) 将dataSet广播出去：withBroadcastSet(DataSet, String)
(2) 获取：在其他operator中，通过继承RichXXFunction，重写open方法来获得：   getRunt
```

例如：

```
// 1. The DataSet to be broadcasted
DataSet<Integer> toBroadcast = env.fromElements(1, 2, 3);

DataSet<String> data = env.fromElements("a", "b");

data.map(new RichMapFunction<String, String>() {
    @Override
    public void open(Configuration parameters) throws Exception {
      // 3. Access the broadcasted DataSet as a Collection
      Collection<Integer> broadcastSet = getRuntimeContext().getBroadcastVariable("broadcastSetName");
    }


    @Override
    public String map(String value) throws Exception {
        ...
    }
}).withBroadcastSet(toBroadcast, "broadcastSetName"); // 2. Broadcast the DataSet
```


这里还有一个K-mean算法的例子，也用到了广播变量：K-Means Algorithm。

3.4、Data Sinks

这块的内容比较简单，直接看一些例子：

```
// text data
DataSet<String> textData = // [...]

// write DataSet to a file on the local file system
textData.writeAsText("file:///my/result/on/localFS");

// write DataSet to a file on a HDFS with a namenode running at nnHost:nnPort
textData.writeAsText("hdfs://nnHost:nnPort/my/result/on/localFS");

// write DataSet to a file and overwrite the file if it exists
textData.writeAsText("file:///my/result/on/localFS", WriteMode.OVERWRITE);

// tuples as lines with pipe as the separator "a|b|c"
DataSet<Tuple3<String, Integer, Double>> values = // [...]
values.writeAsCsv("file:///path/to/the/result/file", "\n", "|");

// this writes tuples in the text formatting "(a, b, c)", rather than as CSV lines
values.writeAsText("file:///path/to/the/result/file");

// this writes values as strings using a user-defined TextFormatter object
values.writeAsFormattedText("file:///path/to/the/result/file",
    new TextFormatter<Tuple2<Integer, Integer>>() {
        public String format (Tuple2<Integer, Integer> value) {
            return value.f1 + " - " + value.f0;
        }
    });
```


自定义Data Sink的例子：

```
DataSet<Tuple3<String, Integer, Double>> myResult = [...]

// write Tuple DataSet to a relational database
myResult.output(
    // build and configure OutputFormat
    JDBCOutputFormat.buildJDBCOutputFormat()
                    .setDrivername("org.apache.derby.jdbc.EmbeddedDriver")
                    .setDBUrl("jdbc:derby:memory:persons")
                    .setQuery("insert into persons (name, age, height) values (?,?,?)")
                    .finish()
    );
```


这里举了一个JDBC sink到数据库的例子。如果想sink到oracle这种不开源的数据库，则需要通过maven引入oracle的jar包，具体操作可参见：maven3 手动安装本地jar到仓库

3.4、Connectors

Flink DataSet API支持许多connectors，用于对外部存储的读写。

3.4.1 文件系统

Flink支持HDFS、S3, Google CloudStorage, Alluxio等，我们需要在pom中引入文件系统的依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-hadoop-compatibility_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

为了使用Hadoop文件系统，需要确保：

```
1、Flink配置文件flink-conf.yaml已经设置了fs.hdfs.hadoopconf的配置
2、在hadoop的配置文件中，要有这些组件的入口，例如S3，Alluxio等的配置
3、要将这些文件系统需要的class文件放到Flink所有节点的lib目录下，如果不方便放，则可以通过
```

例如S3，你需要在core-site.xml中作如下配置：

```
<!-- configure the file system implementation -->
<property>
  <name>fs.s3.impl</name>
  <value>org.apache.hadoop.fs.s3native.NativeS3FileSystem</value>
</property>

<!-- set your AWS ID -->
<property>
  <name>fs.s3.awsAccessKeyId</name>
  <value>putKeyHere</value>
</property>

<!-- set your AWS access key -->
<property>
  <name>fs.s3.awsSecretAccessKey</name>
  <value>putSecretHere</value>
</property>
```

例如Alluxio，在core-site.xml中添加：

```
<property>
  <name>fs.alluxio.impl</name>
  <value>alluxio.hadoop.FileSystem</value>
</property>
```

其他的connector这里不再一一举例，贴张官网一张connector的图：

- S3 (tested)
- Google Cloud Storage Connector for Hadoop (tested)
- Alluxio (tested)
- XtreemFS (tested)
- FTP via Hftp (not tested)
- and many more.

3.4.2 mongoDB

mongoDB：Access MongoDB。

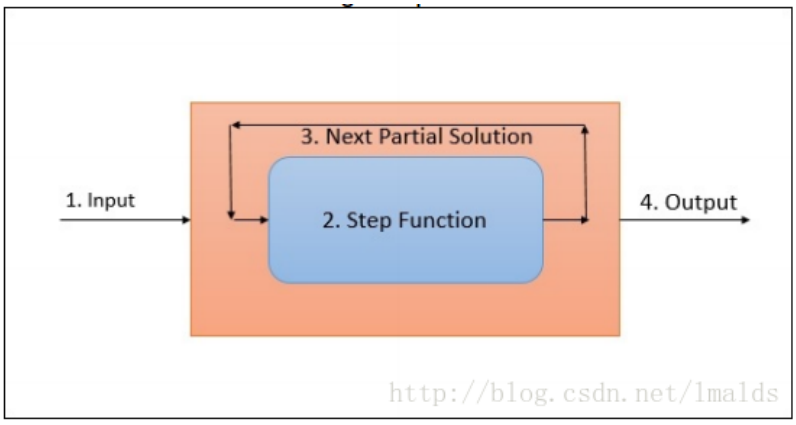
最后，对于Flink DataSet Sink，我们可以通过addSink()自定义一些输出，例如输出到**InfluxDB、oracle、mysql、HBase**等。这里不再做详细介绍。

3.5、迭代

迭代主要用于机器学习、图计算等，Flink通过step函数来支持迭代运算。

3.5.1、迭代器算子

迭代器算子包括以下几步：




```
1、迭代输入：要么来自source，要么是上一个迭代的输出；
2、step函数：应用在DataSet数据集上；
3、Next Partial Solution：step函数都有输出，用于下一次迭代的输入；
4、output：迭代结束或者通过设置一些条件，终止迭代。
```

终止迭代的方式有很多，例如：

```
1、设置迭代次数
2、自定义迭代终止条件
```

例如下面计算pi的例子：

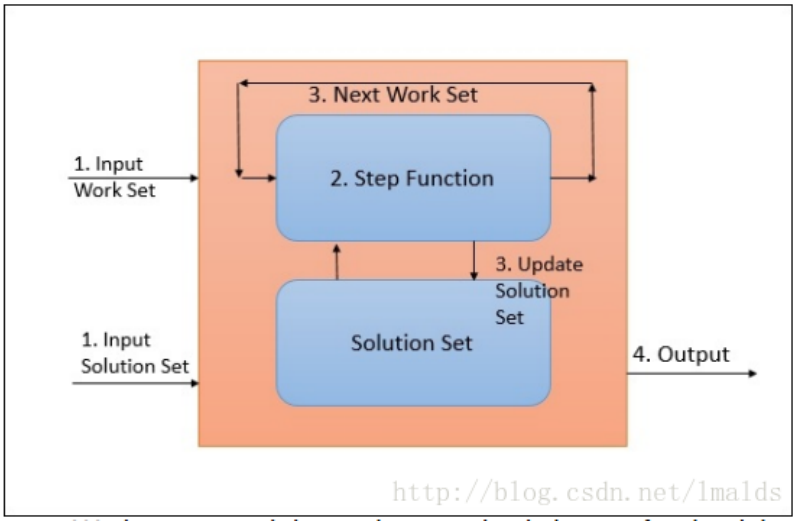
```java
final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// Create initial IterativeDataSet
IterativeDataSet<Integer> initial = env.fromElements(0).iterate(10000);

DataSet<Integer> iteration = initial.map(new MapFunction<Integer, Integer>() {
    @Override
    public Integer map(Integer i) throws Exception {
        double x = Math.random();
        double y = Math.random();

        return i + ((x * x + y * y < 1) ? 1 : 0);
    }
});

// Iteratively transform the IterativeDataSet
DataSet<Integer> count = initial.closeWith(iteration);

count.map(new MapFunction<Integer, Double>() {
    @Override
    public Double map(Integer count) throws Exception {
        return count / (double) 10000 * 4;
    }
}).print();

env.execute("Iterative Pi Example");
```


通过iterate()方法设置最大跌代次数，并将DataSet转换为IterativeDataSet；之后的step函数则是map函数，closeWith(DataSet)代表传递给下一次迭代的数据集是什么，即Next Partial Solution要表达的。

3.5.2、增量迭代

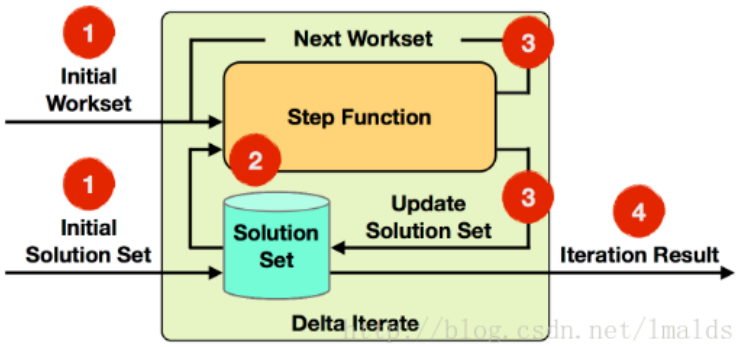
增量迭代与上一节降到的普通迭代的区别是：增量迭代是更新上一步迭代的结果，而不是全部重新计算一次。

增量迭代会使得计算更加有效，时间更短。下面展示增量迭代的数据流：




```
1、迭代输入
2、step函数
3、Next Work Set/ Update Solution：分为workSet和solutionSet，solutionSet维护这
```

```
4、output
```



这里详细说明增量迭代如何开发：

```
// read the initial data sets
DataSet<Tuple2<Long, Double>> initialSolutionSet = // [...]

DataSet<Tuple2<Long, Double>> initialDeltaSet = // [...]

int maxIterations = 100;
int keyPosition = 0;

DeltaIteration<Tuple2<Long, Double>, Tuple2<Long, Double>> iteration = init
    .iterateDelta(initialDeltaSet, maxIterations, keyPosition);

DataSet<Tuple2<Long, Double>> candidateUpdates = iteration.getWorkset()
    .groupBy(1)
    .reduceGroup(new ComputeCandidateChanges());

DataSet<Tuple2<Long, Double>> deltas = candidateUpdates
    .join(iteration.getSolutionSet())
    .where(0)
    .equalTo(0)
    .with(new CompareChangesToCurrent());

DataSet<Tuple2<Long, Double>> nextWorkset = deltas
    .filter(new FilterByThreshold());

iteration.closeWith(deltas, nextWorkset)
    .writeAsCsv(outputPath);
```

通过调用 iterateDelta(DataSet, int, int) 或者 iterateDelta(DataSet, int, int[]) 来生成一个 DeltaIteration。
之后通过iteration.getWorkset() 和 iteration.getSolutionSet()来获得workset 和 solution set。

通过workSet以及solutionSet的join操作，每次迭代时对workSet应用solutionSet中的状态值，实现

了增量迭代的效果。

增量迭代的详细介绍，可以参考：Data Analysis with Flink: A case study and tutorial。

3.6、批处理用例

这里可以参考dataArtisans的flink training的例子：flink-training-exercises、Apache Flink Training。

3.7、总结

本章主要介绍Flink DataSet API，下一章开始介绍Flink生态中的Table API。

4、Table API 数据处理

Flink提供了一个table接口来进行批处理和流处理，这个接口叫做Table API。一旦dataset/datastream被注册为table后，就可以引用聚合、join和select等关系型的操作了。

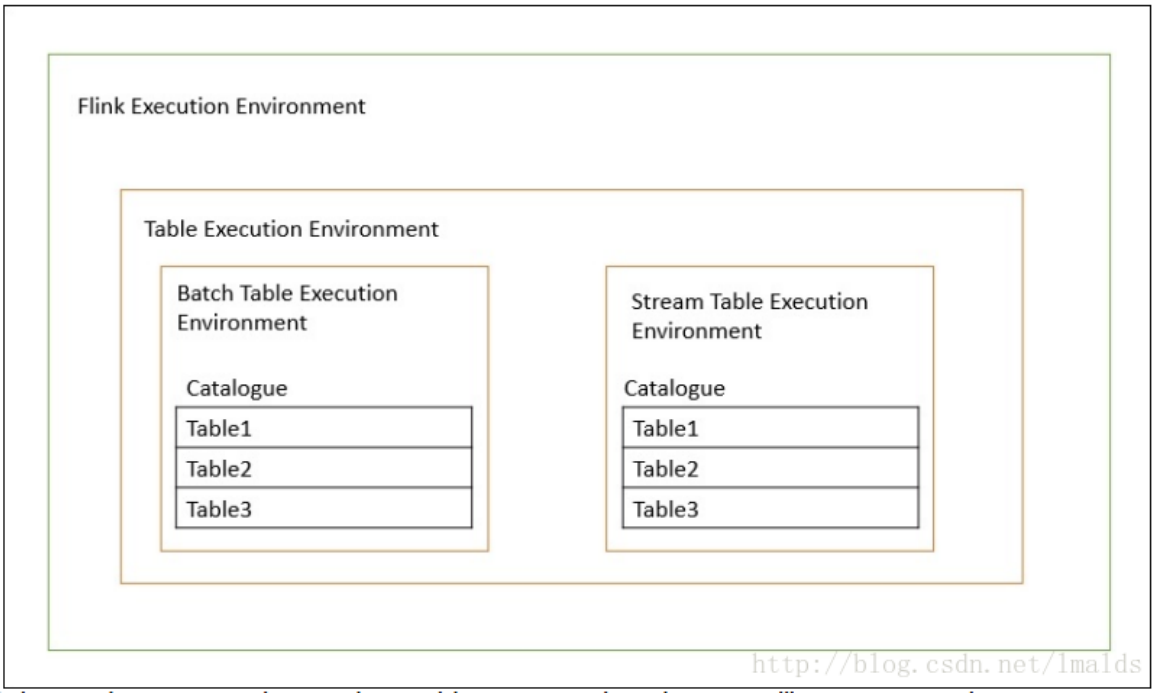
Table同样可以通过标准SQL来操作，操作执行后，需要将table转换为dataSet/datastream。Flink内部中使用开源框架Apache Calcite来优化这些转换操作。

为了使用Table API，我们首先需要引入依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

4.1、Tables注册

我们首先要在TableEnvironment中将dataset或datastream注册，而TableEnvironment中维护者table的基本信息，细节如下：



4.1.1、注册dataset

为了在dataset中使用SQL，我们需要将dataset注册为一个table。

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment(env);

// register the DataSet cust as table "Customers" with fields derived from
tableEnv.registerDataSet("Customers", cust)

// register the DataSet ord as table "Orders" with fields user, product, an
tableEnv.registerDataSet("Orders", ord, "user, product, amount");
```

4.1.2、注册datastream

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnv
StreamTableEnvironment tableEnv = TableEnvironment.getTableEnvironment(env)

// register the DataStream cust as table "Customers" with fields derived fr
tableEnv.registerDataStream("Customers", cust)

// register the DataStream ord as table "Orders" with fields user, product,
tableEnv.registerDataStream("Orders", ord, "user, product, amount");
```

4.1.3、注册table

```
// works for StreamExecutionEnvironment identically
```

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment(env);

// convert a DataSet into a Table
Table custT = tableEnv
  .toTable(custDs, "name, zipcode")
  .where("zipcode = '12345'")
  .select("name")

// register the Table custT as table "custNames"
tableEnv.registerTable("custNames", custT)
```

4.1.4、注册外部数据源

```
// works for StreamExecutionEnvironment identically
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment(env);

TableSource custTS = new CsvTableSource("/path/to/file", ...)

// register a `TableSource` as external table "Customers"
tableEnv.registerTableSource("Customers", custTS)
```

通过TableSource 可以访问但存储在数据库（mysql、hbase等）、文件系统（CSV, Apache Parquet, Avro, ORC等）以及消息系统（Apache Kafka, RabbitMQ）中。

当前Flink预定义的TableSource如下：

Available Table Sources

Class name	Maven dependency	Batch?	Streaming?	Description
CsvTableSouce	flink-table	Y	Y	A simple source for CSV files.
Kafka08JsonTableSource	flink-connector-kafka-0.8	N	Y	A Kafka 0.8 source for JSON data.
Kafka09JsonTableSource	flink-connector-kafka-0.9	N	Y	A Kafka 0.9 source for JSON data.

可以看到，KafkaJsonTableSource还只能用在流中，将dataStream转换为TableSource。

4.1.4.1 CSV table source

CSV source默认存在flink-table的API Jar包中，因此你无需再引入其他的依赖。

CsvTableSource 可以配置以下属性：

```
path：文件路径
fieldNames：table的字段名
fieldTypes：table字段的类型
```

```
fieldDelim：列分隔符，默认是“，”
rowDelim：行分隔符，默认是"\n"
quoteCharacter：对于String值可选的属性,默认是null
ignoreFirstLine：忽略第一行，默认是false
ignoreComments：可选择的前缀，用于注释，默认是null
lenient：跳过错误的记录，默认是false
```

下面展示一个例子：

```
CsvTableSource csvTableSource = new CsvTableSource(
    "/path/to/your/file.csv",
    new String[] { "name", "id", "score", "comments" },
    new TypeInformation<?>[] {
      Types.STRING(),
      Types.INT(),
      Types.DOUBLE(),
      Types.STRING()
    },
    "#",     // fieldDelim
    "$",     // rowDelim
    null,    // quoteCharacter
    true,    // ignoreFirstLine
    "%",     // ignoreComments
    false); // lenient
```


4.1.4.2 Kafka JSON table source

为了使用KafkaJsonTableSource，你需要添加依赖（这里kafka 0.9为例）：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka-0.9_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

你可以像下面这样创建Table Source：

```
// The JSON field names and types
String[] fieldNames =  new String[] { "id", "name", "score"};
Class<?>[] fieldTypes = new Class<?>[] { Integer.class, String.class, Doubl

KafkaJsonTableSource kafkaTableSource = new Kafka08JsonTableSource(
    kafkaTopic,
    kafkaProperties,
    fieldNames,
    fieldTypes);
```

```
tableEnvironment.registerTableSource("kafka-source", kafkaTableSource);
Table result = tableEnvironment.ingest("kafka-source");
```

4.2、访问已经注册的表

对于BatchTableEnvironment，我们通过：

```
tableEnvironment.scan("tableName")
```

对于StreamTableEnvironment，我们通过：

```
tableEnvironment.ingest("tableName")
```

4.3、operators

Flink Table API提供了各种各样的operators，其中大部分都支持java和scala。

4.3.1 select

根SQL中的select很像，查询Table中的字段：

```
Table result = in.select("id, name");
Table result = in.select("*");
```

4.3.2 where和filter

这两个等价，过滤作用：

```
Table in = tableEnv.fromDataSet(ds, "a, b, c");
Table result = in.where("b = 'red'");
```

```
Table in = tableEnv.fromDataSet(ds, "a, b, c");
Table result = in.filter("a % 2 = 0");
```

4.3.3 as

对字段重命名：

```
Table result = in.select("a, c as d");
```

4.3.4 groupBy

和SQL的groupBy操作很像，根据某个属性分组：

```
Table in = tableEnv.fromDataSet(ds, "a, b, c");
Table result = in.groupBy("a").select("a, b.sum as d");
```

4.3.5 join

两个表join。至少指出一个连接条件，可以通过where或filter指定：

```
Table employee = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table dept = tableEnv.fromDataSet(dept, "d_id, d_name");

Table result = employee.join(dept).where("deptId =
d_id").select("e_id, e_name, d_name");
```

4.3.6 leftOuterJoin

在SQL中，left join等价于left Outer Join，但是在flink中，表达left join不能忽略中间的outer：

```
Table employee = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table dept = tableEnv.fromDataSet(dept, "d_id, d_name");

Table result = employee.leftOuterJoin(dept).where("deptId =
d_id").select("e_id, e_name, d_name");
```

当然，你也可以简写：

```
Table left = tableEnv.fromDataSet(ds1, "a, b, c");
Table right = tableEnv.fromDataSet(ds2, "d, e, f");
Table result = left.leftOuterJoin(right, "a = d").select("a, b, e");
```

4.3.7 rightOuterJoin

右连接：

```
Table employee = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table dept = tableEnv.fromDataSet(dept, "d_id, d_name");

Table result = employee.rightOuterJoin(dept).where("deptId =
d_id").select("e_id, e_name, d_name");
```

当然，你也可以简写：

```
Table left = tableEnv.fromDataSet(ds1, "a, b, c");
Table right = tableEnv.fromDataSet(ds2, "d, e, f");
Table result = left.rightOuterJoin(right, "a = d").select("a, b, e");
```

4.3.8 fullOuterJoin

full join，左右两边join不到时，全部保留，左边或右边用null填充。

```
Table employee = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table dept = tableEnv.fromDataSet(dept, "d_id, d_name");

Table result = employee.fullOuterJoin(dept).where("deptId =
d_id").select("e_id, e_name, d_name");
```

当然，你可以简写：

```
Table left = tableEnv.fromDataSet(ds1, "a, b, c");
Table right = tableEnv.fromDataSet(ds2, "d, e, f");
Table result = left.fullOuterJoin(right, "a = d").select("a, b, e");
```

4.3.9 union

跟SQL的union一样，将两个相似（字段类型和个数一样）的table union起来，但是，它起到了**去重**的作用：

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table result = employee1.union(employee2);
```

4.3.10 unionAll

跟SQL的union all操作一样，但是它不去重：

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table result = employee1.unionAll(employee2);
```

4.3.11 intersect

和SQL中的intersect一样，求两个表的交集，即两个table中都存在的数据。但是结果**去重**，即结果中没有重复的数据存在：

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table result = employee1.intersect(employee2);
```

4.3.12 intersectAll

求交集，但是不去重：

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
```

```
Table result = employee1.intersectAll(employee2);
```

4.3.13 minus

和SQL的minus一样，求差集。即左边table存在但右边table不存在的记录，结果去重：

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table result = employee1.minus(employee2);
```

4.3.14 minusAll

求差集，但是结果不去重，即左边如果有重复数据，结果并不去重：

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table result = employee1.minusAll(employee2);
```

4.3.15 distinct

和SQL的distinct一样：

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table result = employee1.distinct();
```

4.3.16 orderBy

按照某个字段全局排序：

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
Table result = employee1.orderBy("e_id.asc");
```

4.3.17 limit

配合orderBy一起使用，即在orderBy的结果上，从第n+1条开始取，limit支持两种参数：

第一种写法是一个参数，代表从第6个数据开始取，知道后边所有的数据：

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");
//returns records from 6th record
Table result = employee1.orderBy("e_id.asc").limit(5);
```

第二种写法是2个参数，第一个参数代表从第4个数据开始取，往后取5条数据：

```
//returns 5 records from 4th record
```

```
Table result1 = employee1.orderBy("e_id.asc").limit(3,5);
```

4.3.18 数据类型

flink使用TypeInformation来自动识别table和sql以及java中的数据类型，当前的匹配如下：

Table API	SQL	Java type
Types.STRING	VARCHAR	java.lang.String
Types.BOOLEAN	BOOLEAN	java.lang.Boolean
Types.BYTE	TINYINT	java.lang.Byte
Types.SHORT	SMALLINT	java.lang.Short
Types.INT	INTEGER, INT	java.lang.Integer
Types.LONG	BIGINT	java.lang.Long
Types.FLOAT	REAL, FLOAT	java.lang.Float
Types.DOUBLE	DOUBLE	java.lang.Double
Types.DECIMAL	DECIMAL	java.math.BigDecimal
Types.DATE	DATE	java.sql.Date
Types.TIME	TIME	java.sql.Time
Types.TIMESTAMP	TIMESTAMP(3)	java.sql.Timestamp
Types.INTERVAL_MONTHS	INTERVAL YEAR TO MONTH	java.lang.Integer
Types.INTERVAL_MILLIS	INTERVAL DAY TO SECOND(3)	java.lang.Long

4.4、flink SQL

通过sql()方法注册给TableEnviroment，我们可以在Table API中使用SQL。目前SQL功能适用于DataSet批处理和DataStream流处理，但是目前流处理支持的SQL力度有限，如果SQL语句中出现flink不能识别的语法，则会抛出TableException异常。

Flink内部通过Apache Calcite对SQL进行解析、优化和执行。

4.4.1 SQL on DataSet

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment(env);

// read a DataSet from an external source
DataSet<Tuple3<Long, String, Integer>> ds = env.readCsvFile(...);
// register the DataSet as table "Orders"
tableEnv.registerDataSet("Orders", ds, "user, product, amount");
// run a SQL query on the Table and retrieve the result as a new Table
Table result = tableEnv.sql(
  "SELECT SUM(amount) FROM Orders WHERE product LIKE '%Rubber%'");
```

当前版本（Flink 1.2）在DataSet中的SQL，支持的语法包括select（filter）、projection、等价join、分组、非distinct的聚合操作，排序等，但不支持以下语法：

```
1、毫秒精度timestamp和intervals
2、不支持interval
3、类似于COUNT(DISTINCT name)不支持
4、非等价连接和笛卡尔积不支持
5、Grouping sets操作不支持
```

4.4.2 SQL on DataStream

目前1.2版本中的SQL on DataStream还支持select、from、where和union操作，其他的类似聚合类的、join等操作还不支持。

通过SELECT STREAM进行：

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment tableEnv =
TableEnvironment.getTableEnvironment(env);

DataStream<Tuple3<Long, String, Integer>> ds = env.addSource(...);
// register the DataStream as table "Products"
tableEnv.registerDataStream("Products", ds, "id, name, stock");

// run a SQL query on the Table and retrieve the result as a new Table
Table result = tableEnv.sql(
"SELECT STREAM * FROM Products WHERE name LIKE '%Apple%'");
```

4.5、用例

这里，我单独开了一篇博客，以2016年中超联赛射手榜的榜单为源数据，对这份榜单使用Flink SQL来进行了简单的统计，详见：Apache Flink SQL示例

4.6、总结

这篇我们介绍了Table API以及基于SQL的API。通过TableEnvironment在dataset、datastream和table之间的转换以及注册。下一篇文章我们将介绍复杂事件处理：Flink CEP。