> 不要轻易凝视深渊

FlinkSQl之Table的操作

注册表

表（Table）的概念

TableEnvironment可以注册目录Catalog，**并可以基于Catalog注册表。**它会维护一个Catalog-Table表之间的map。表（Table）是由一个“标识符”来指定的，由3部分组成：**Catalog名、数据库（database）名和对象名（表名）**。如果没有指定目录或数据库，就使用当前的默认值。

> 表可以是常规的（Table，表），或者虚拟的（View，视图）。常规表（Table）一般可以用来描述外部数据，**比如文件、数据库表或消息队列的数据，也可以直接从 DataStream转换而来。**视图可以从现有的表中创建，通常是table API或者SQL查询的一个结果。

连接到文件系统（Csv格式）

> 连接外部系统在Catalog中注册表，直接调用tableEnv.connect()就可以，里面参数要传入一个ConnectorDescriptor，也就是connector描述器。对于文件系统的connector而言，**flink内部已经提供了，就叫做FileSystem()。**

代码

```
tableEnv
.connect( new FileSystem().path("sensor.txt"))  // 定义表数据来源，外部连接
  .withFormat(new OldCsv())    // 定义从外部系统读取数据之后的格式化方法
  .withSchema( new Schema()
    .field("id", DataTypes.STRING())
    .field("timestamp", DataTypes.BIGINT())
    .field("temperature", DataTypes.DOUBLE())
  )    // 定义表结构
  .createTemporaryTable("inputTable")    // 创建临时表
```

> 这是旧版本的csv格式描述器。由于它是非标的，跟外部系统对接并不通用，所以将被弃用，以后会被一个符合RFC-4180标准的新format描述器取代。新的描述器就叫Csv()，但flink没有直接提供，需要引入依赖flink-csv：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-csv</artifactId>
    <version>1.10.0</version>
</dependency>
```

> 代码非常类似，只需要把withFormat里的OldCsv改成Csv就可以了。

连接到Kafka

> **kafka的连接器flink-kafka-connector中，1.10版本的已经提供了Table API的支持。**我们可以在 connect方法中直接传入一个叫做Kafka的类，这就是kafka连接器的描述器ConnectorDescriptor。【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

连接代码

```
tableEnv.connect(
  new Kafka()
    .version("0.11") // 定义kafka的版本
    .topic("sensor") // 定义主题
    .property("zookeeper.connect", "localhost:2181")
    .property("bootstrap.servers", "localhost:9092")
)
  .withFormat(new Csv())
  .withSchema(new Schema()
  .field("id", DataTypes.STRING())
  .field("timestamp", DataTypes.BIGINT())
  .field("temperature", DataTypes.DOUBLE())
)
  .createTemporaryTable("kafkaInputTable")
```

> 当然也可以连接到ElasticSearch、MySql、HBase、Hive等外部系统，实现方式基本上是类似的。！

连接到数据源，获得到数据了，数据已经变成表了，那么就可以开始查询了，Flink给我们提供了两种查询方式：Table API和 SQL。

查询表

Table API

Table API是集成在Scala和Java语言内的查询API。**与SQL不同，Table API的查询不会用字符串表示**，而是在宿主语言中一步一步调用完成的。

> Table API基于代表一张“表”的Table类，并提供一整套操作处理的方法API。这些方法会返回一个新的Table对象，这个对象就表示对输入表应用转换操作的结果。有些关系型转换操作，可以由多个方法调用组成，构成链式调用结构。例如table.select(...).filter(...)，其中select（...）表示选择表中指定的字段，filter(...)表示筛选条件。

代码中的实现如下：

```
val sensorTable: Table = tableEnv.from("inputTable")

val resultTable: Table = senorTable
.select("id, temperature")
.filter("id ='sensor_1'")
```

SQL查询

> Flink的SQL集成，基于的是ApacheCalcite，它实现了SQL标准。在Flink中，用常规字符串来定义SQL查询语句。SQL 查询的结果，是一个新的 Table。

代码实现如下：

```
val resultSqlTable: Table = tableEnv.sqlQuery("select id, temperature from
inputTable where id ='sensor_1'")
```

OR

```
val resultSqlTable: Table = tableEnv.sqlQuery(
  """
    |select id, temperature
    |from inputTable
    |where id = 'sensor_1'
  """.stripMargin)
```

当然，也可以加上聚合操作，比如我们统计每个sensor温度数据出现的个数，做个count统计

```
val aggResultTable = sensorTable
.groupBy('id)
.select('id, 'id.count as 'count)
```

SQL实现代码如下

```
val aggResultSqlTable = tableEnv.sqlQuery("select id, count(id) as cnt from
inputTable group by id")
```

> 这里Table API里指定的字段，前面加了一个单引号'，这是Table API中定义的Expression类型的写法，可以很方便地表示一个表中的字段。**字段可以直接全部用双引号引起来，也可以用半边单引号+字段名的方式。**以后的代码中，一般都用后一种形式。

转换表【DataStream->表】

> Flink允许我们把Table和DataStream做转换：**我们可以基于一个DataStream，先流式地读取数据源**，然后map成样例类，再把它转成Table。Table的列字段（column fields），就是样例类里的字段，这样就不用再麻烦地定义schema了。

转换代码

```
val inputStream: DataStream[String] = env.readTextFile("sensor.txt")
val dataStream: DataStream[SensorReading] = inputStream
  .map(data => {
    val dataArray = data.split(",")
    // 样例类还是好用
    SensorReading(dataArray(0), dataArray(1).toLong, dataArray(2).toDouble)
  })

val sensorTable: Table = tableEnv.fromDataStream(dataStream)
val sensorTable2 = tableEnv.fromDataStream(dataStream, 'id, 'timestamp as 'ts)
```

数据类型与 Table schema的对应

> 在上节的例子中，DataStream 中的数据类型，与表的 Schema 之间的对应关系，是按照样例类中的字段名来对应的（name-based mapping），所以还可以用as做重命名。

另外一种对应方式是，**直接按照字段的位置来对应（position-based mapping）**，对应的过程中，就可以直接指定新的字段名了。

基于名称的对应代码

```
val sensorTable = tableEnv.fromDataStream(dataStream, 'timestamp as 'ts, 'id as 'myId, 'temperature)
```

基于位置的对应代码

```
val sensorTable = tableEnv.fromDataStream(dataStream, 'myId, 'ts)
```

> Flink的DataStream和 DataSet API支持多种类型。组合类型，比如元组（内置Scala和Java元组）、POJO、Scala case类和Flink的Row类型等，允许具有多个字段的嵌套数据结构，**这些字段可以在Table的表达式中访问。**
>
> 其他类型，则被视为原子类型。元组类型和原子类型，一般用位置对应会好一些；如果非要用名称对应，也是可以的：**元组类型，默认的名称是 *“1”, “2”*；而原子类型，默认名称是 ”f0”。**

创建临时视图（Temporary View）

> 创建临时视图的第一种方式，就是直接从DataStream转换而来。同样，可以直接对应字段转换；也可以在转换的时候，指定相应的字段。

基于DataStream创建视图

```
tableEnv.createTemporaryView("sensorView", dataStream)
tableEnv.createTemporaryView("sensorView", dataStream, 'id, 'temperature, 'timestamp as 'ts)
```

基于Table创建视图

```
tableEnv.createTemporaryView("sensorView", sensorTable)
```

输出表

> 表的输出，是通过将数据写入 TableSink 来实现的。TableSink 是一个通用接口，可以支持不同的文件格式、存储数据库和消息队列。

具体实现，输出表最直接的方法，就是通过 Table.insertInto() 方法将一个 Table 写入注册过的 TableSink 中。

```
// 注册输出表
tableEnv.connect(
  new FileSystem().path("…\\resources\\out.txt")
) // 定义到文件系统的连接
  .withFormat(new Csv()) // 定义格式化方法，Csv格式
  .withSchema(new Schema()
  .field("id", DataTypes.STRING())
  .field("temp", DataTypes.DOUBLE())
) // 定义表结构
  .createTemporaryTable("outputTable") // 创建临时表

resultSqlTable.insertInto("outputTable")
```

更新模式（Update Mode）

> 在流处理过程中，表的处理并不像传统定义的那样简单。

对于流式查询（Streaming Queries），需要**声明如何在（动态）表和外部连接器之间执行转换。**与外部系统交换的消息类型，由更新模式（update mode）指定。

Flink Table API中的更新模式有以下三种：

（1）追加模式（Append Mode）

> 在追加模式下，表（动态表）和外部连接器只交换插入（Insert）消息。

（2）撤回模式（Retract Mode）

> 在撤回模式下，表和外部连接器交换的是：添加（Add）和撤回（Retract）消息。

- 插入（Insert）会被编码为添加消息；
- 删除（Delete）则编码为撤回消息；
- 更新（Update）则会编码为，已更新行（上一行）的撤回消息，和更新行（新行）的添加消息。

在此模式下，不能定义key，这一点跟upsert模式完全不同。

（3）Upsert（更新插入）模式

> 在Upsert模式下，动态表和外部连接器交换Upsert和Delete消息。这个模式需要一个唯一的key，通过这个key可以传递更新消息。为了正确应用消息，外部连接器需要知道这个唯一key的属性。

- 插入（Insert）和更新（Update）都被编码为Upsert消息；
- 删除（Delete）编码为Delete信息。

这种模式和Retract模式的主要区别在于，Update操作是用单个消息编码的，所以效率会更高。

输出到Kafka

> 除了输出到文件，也可以输出到Kafka。我们可以结合前面Kafka作为输入数据，构建数据管道，kafka进，kafka出。

代码如下

```
// 输出到 kafka
tableEnv.connect(
  new Kafka()
```

```
    .version("0.11")
    .topic("sinkTest")
    .property("zookeeper.connect", "localhost:2181")
    .property("bootstrap.servers", "localhost:9092")
)
  .withFormat( new Csv() )
  .withSchema( new Schema()
    .field("id", DataTypes.STRING())
    .field("temp", DataTypes.DOUBLE())
  )
  .createTemporaryTable("kafkaOutputTable")

resultTable.insertInto("kafkaOutputTable")
```

输出到ElasticSearch

> ElasticSearch的connector可以在upsert（update+insert，更新插入）模式下操作，这样就可以使用Query定义的键（key）与外部系统交换UPSERT/DELETE消息。另外，对于“仅追加”（append-only）的查询，connector还可以在append 模式下操作，这样就可以与外部系统只交换insert消息。

es目前支持的数据格式，只有Json，而flink本身并没有对应的支持，所以还需要引入依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-json</artifactId>
    <version>1.10.0</version>
</dependency>
```

代码实现

```
// 输出到es
tableEnv.connect(
  new Elasticsearch()
    .version("6")
    .host("localhost", 9200, "http")
    .index("sensor")
    .documentType("temp")
)
  .inUpsertMode()           // 指定是 Upsert 模式
  .withFormat(new Json())
  .withSchema( new Schema()
    .field("id", DataTypes.STRING())
    .field("count", DataTypes.BIGINT())
  )
  .createTemporaryTable("esOutputTable")

aggResultTable.insertInto("esOutputTable")
```

输出到MySql

引入依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-jdbc_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```

> jdbc连接的代码实现比较特殊，**因为没有对应的java/scala类实现ConnectorDescriptor**，所以不能直接tableEnv.connect()。不过Flink SQL留下了执行DDL的接口：tableEnv.sqlUpdate()。

对于jdbc的创建表操作，天生就适合直接写DDL来实现，所以我们的代码可以这样写

```
// 输出到 Mysql
val sinkDDL: String =
  """
    |create table jdbcOutputTable (
    |  id varchar(20) not null,
    |  cnt bigint not null
    |) with (
    |  'connector.type' = 'jdbc',
    |  'connector.url' = 'jdbc:mysql://localhost:3306/test',
    |  'connector.table' = 'sensor_count',
    |  'connector.driver' = 'com.mysql.jdbc.Driver',
    |  'connector.username' = 'root',
    |  'connector.password' = '123456'
    |)
  """.stripMargin

tableEnv.sqlUpdate(sinkDDL)
aggResultSqlTable.insertInto("jdbcOutputTable")
```

转换表【表->DataStream】

> 表可以转换为DataStream或DataSet。这样，自定义流处理或批处理程序就可以继续在 Table API或SQL查询的结果上运行了。将表转换为DataStream或DataSet时，**需要指定生成的数据类型，即要将表的每一行转换成的数据类型**。通常，最方便的转换类型就是Row。当然，因为结果的所有字段类型都是明确的，我们也经常会用元组类型来表示。

表作为流式查询的结果，是动态更新的。所以，**将这种动态查询转换成的数据流，同样需要对表的更新操作进行编码，进而有不同的转换模式。**

Table API中表到DataStream有两种模式：

- 追加模式（Append Mode）

> 用于表只会被插入（Insert）操作更改的场景。

- 撤回模式（Retract Mode）

> 用于任何场景。有些类似于更新模式中Retract模式，它只有Insert和Delete两类操作。
>
> 得到的数据会增加一个Boolean类型的标识位（返回的第一个字段），用它来表示到底是新增的数据（Insert），还是被删除的数据（老数据， Delete）。

代码实现

```
val resultStream: DataStream[Row] = tableEnv.toAppendStream[Row](resultTable)

val aggResultStream: DataStream[(Boolean, (String, Long))] =
tableEnv.toRetractStream[(String, Long)](aggResultTable)

resultStream.print("result")
aggResultStream.print("aggResult")
```

所以，没有经过groupby之类聚合操作，可以直接用 toAppendStream 来转换；**而如果经过了聚合，有更新操作，一般就必须用 toRetractDstream。**

Query的解释和执行

> Table API提供了一种机制来解释（Explain）计算表的逻辑和优化查询计划。这是通过TableEnvironment.explain（table）方法或TableEnvironment.explain（）方法完成的。

explain方法会返回一个字符串，描述三个计划：

- 未优化的逻辑查询计划
- 优化后的逻辑查询计划
- 实际执行计划

在代码中查看执行计划

```
val explaination: String = tableEnv.explain(resultTable)
println(explaination)
```

Query的解释和执行过程，老planner和blink planner大体是一致的，又有所不同。整体来讲，Query都会表示成一个逻辑查询计划，然后分两步解释：

- 优化查询计划
- 解释成 DataStream 或者 DataSet程序

而Blink版本是批流统一的，**所以所有的Query，只会被解释成DataStream程序**；另外在批处理环境TableEnvironment下，Blink版本要到tableEnv.execute()执行调用才开始解释。