Flink

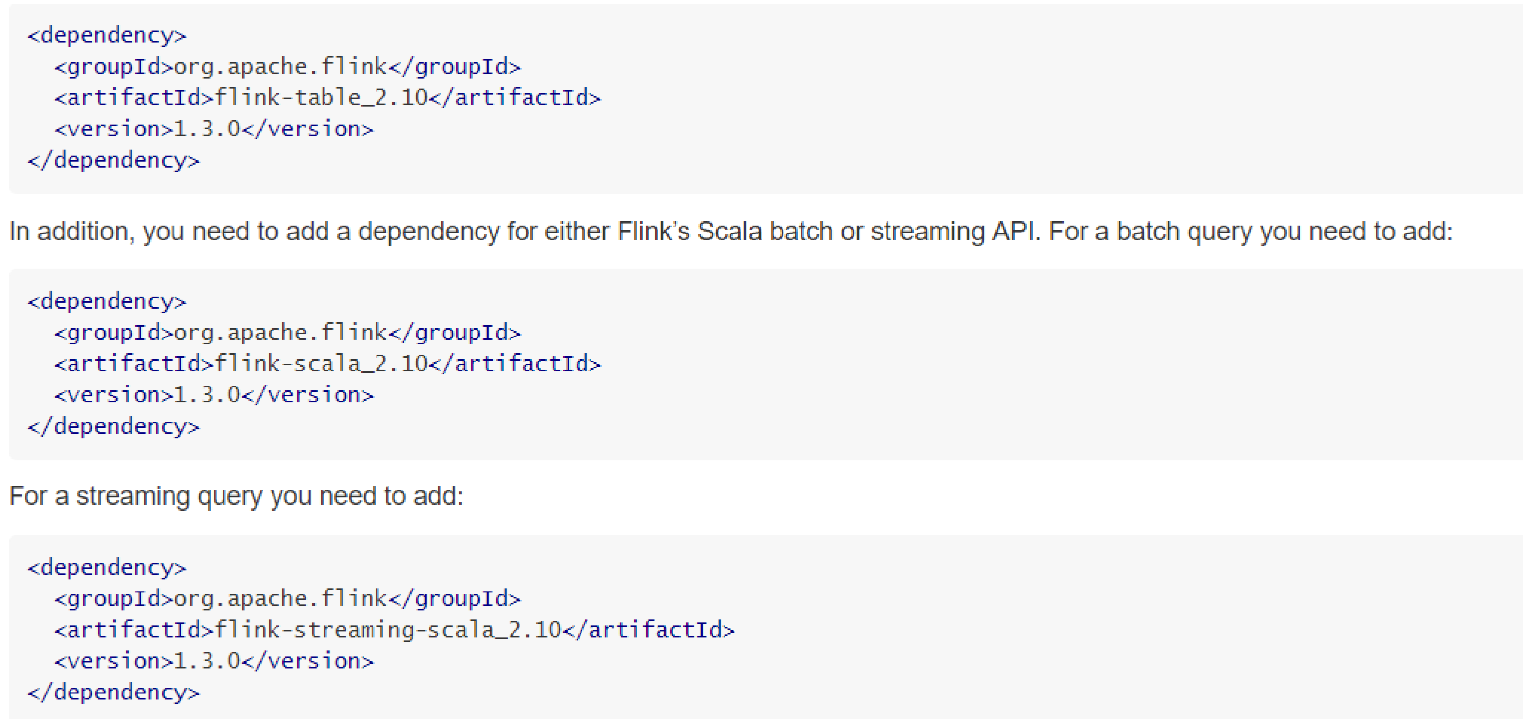
Flink Table&SQL

Table API & SQL

Flink两个相关API分别是Table API 和SQL统一流处理和批处理，可以使用Scala或Java语言进行查询，例如：Select、join、filter。

Table API和SQL接口和Flink’s DataStream 和DataSet API紧密整合，我们可以使用Table API分析CEP Pattern或者在进行图计算之前使用SQL对数据进行预处理。

开发Setup
需要引入flinnk-table，flink batch 、flink streaming依赖。



```xml
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table_2.10</artifactId>
  <version>1.3.0</version>
</dependency>
```

In addition, you need to add a dependency for either Flink’s Scala batch or streaming API. For a batch query you need to add:

```xml
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.10</artifactId>
  <version>1.3.0</version>
</dependency>
```

For a streaming query you need to add:

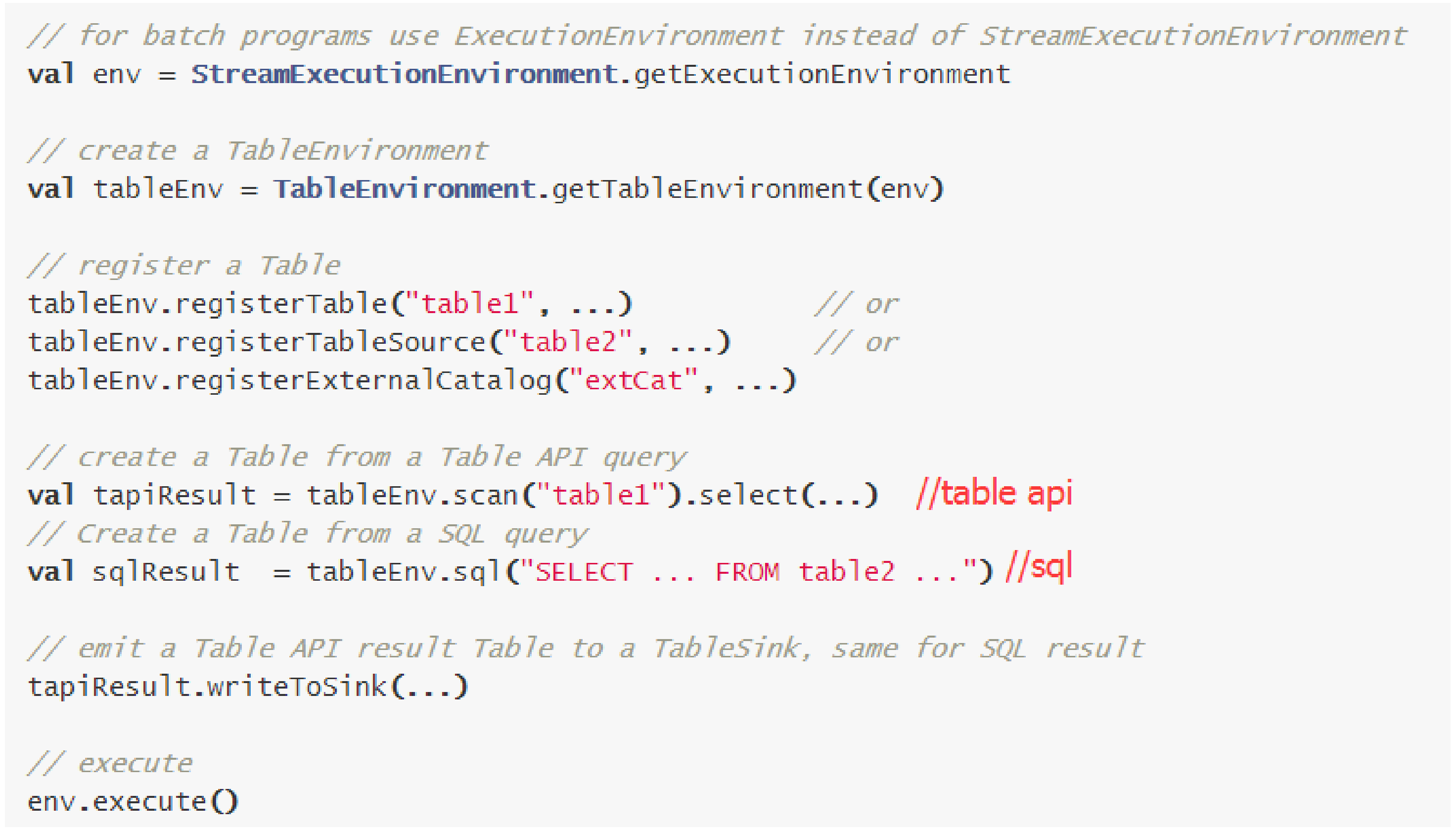
```xml
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.10</artifactId>
  <version>1.3.0</version>
</dependency>
```

Table API & SQL

Apache Calcite是一个动态数据管理框架，（Apache Calcite is a dynamic data management framework.）
Flink SQL是基于Apache Calcite实现的，由于Apache Calcite存在bug，所以有如下建议：
由于Apache Calcite存在的问题会阻止用户代码垃圾回收，因此为了能够方便回收，我们建议
作业打jar时候不将flink-table依赖打进作业jar中，而是将flink table依赖存储到flink安装目录
的lib中。

Concepts & Common API

Table API和SQL整合在一个共同的API中，该API的核心概念是Table，Table为输入数据或者输出数据提供查询。接下来探讨Table api 和SQL查询的程序结构，如何注册一个Table、查询Table、emit一个Table
程序结果一般如下：

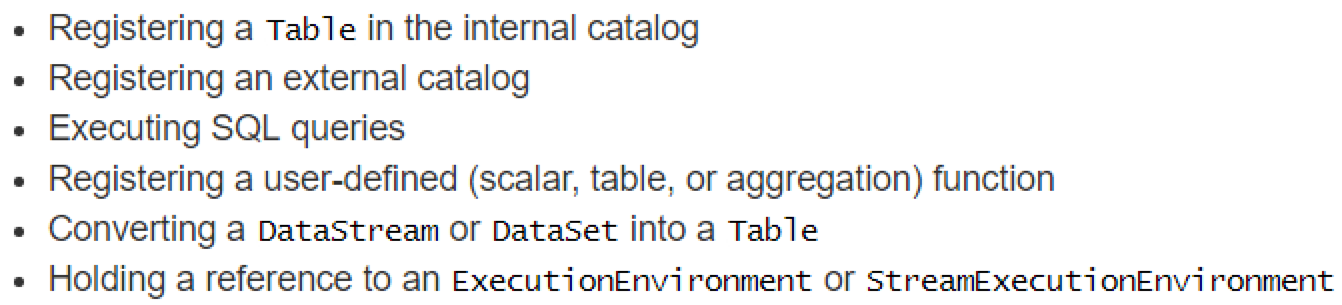


```
// for batch programs use ExecutionEnvironment instead of StreamExecutionEnvironment
val env = StreamExecutionEnvironment.getExecutionEnvironment

// create a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// register a Table
tableEnv.registerTable("table1", ...)           // or
tableEnv.registerTableSource("table2", ...)     // or
tableEnv.registerExternalCatalog("extCat", ...)

// create a Table from a Table API query
val tapiResult = tableEnv.scan("table1").select(...)  //table api
// Create a Table from a SQL query
val sqlResult  = tableEnv.sql("SELECT ... FROM table2 ...") //sql

// emit a Table API result Table to a TableSink, same for SQL result
tapiResult.writeToSink(...)

// execute
env.execute()
```

如上程序分别使用了Table API和SQL查询

Concepts & Common API

TableEnvironment是Table API和SQL的核心概念，TableEnvironment 负责如下事情：



- Registering a `Table` in the internal catalog
- Registering an external catalog
- Executing SQL queries
- Registering a user-defined (scalar, table, or aggregation) function
- Converting a `DataStream` or `DataSet` into a `Table`
- Holding a reference to an `ExecutionEnvironment` or `StreamExecutionEnvironment`

每一个Table必须和一个TableEnvironment绑定，不同TableEnvironment的Table在同一个查询中是不可以组合的！
TableEnvironment 的创建是使用TableEnvironment静态getTableEnvironment方法创建，getTableEnvironment方法有重载，一个参数或者两个参数，两个参数的第二个参数一般是TableConfig，用来优化查询。

Register a Table in the Catalog

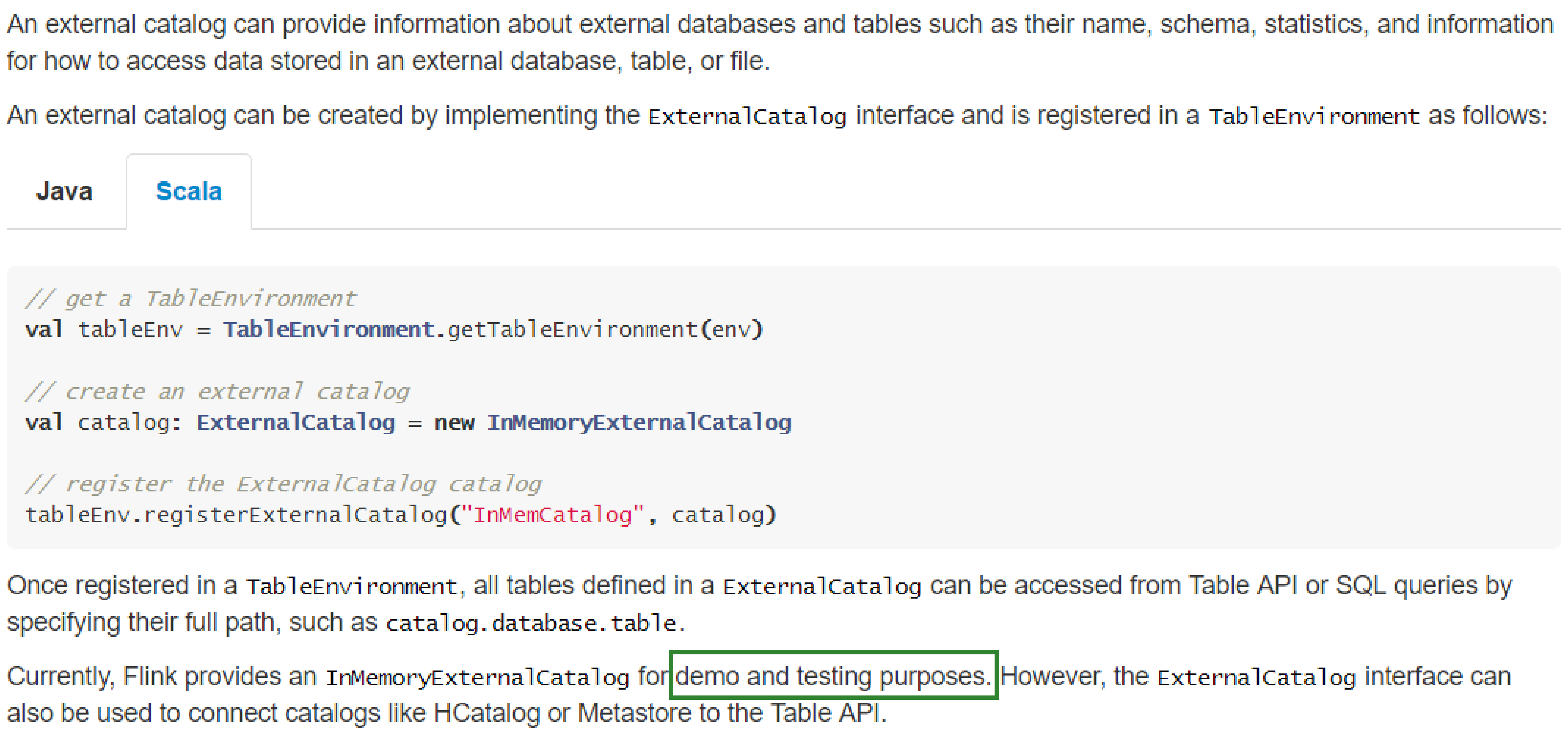
TableEnvironment内部有一个表目录，目录由表明进行组织。Table API或SQL通过访问注册的表明进行访问Table。

Table Source有如下种：
1：存在的Table
2：例如外部文件数据、database中的数据
3：DataStream或者DataSet

Register a Table

```
// get a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// Table is the result of a simple projection query
val projTable: Table = tableEnv.scan("X").project(...)

// register the Table projTable as table "projectedX"
tableEnv.registerTable("projectedTable", projTable)
```

Register an External Catalog



An external catalog can provide information about external databases and tables such as their name, schema, statistics, and information for how to access data stored in an external database, table, or file.

An external catalog can be created by implementing the `ExternalCatalog` interface and is registered in a `TableEnvironment` as follows:

Java | Scala

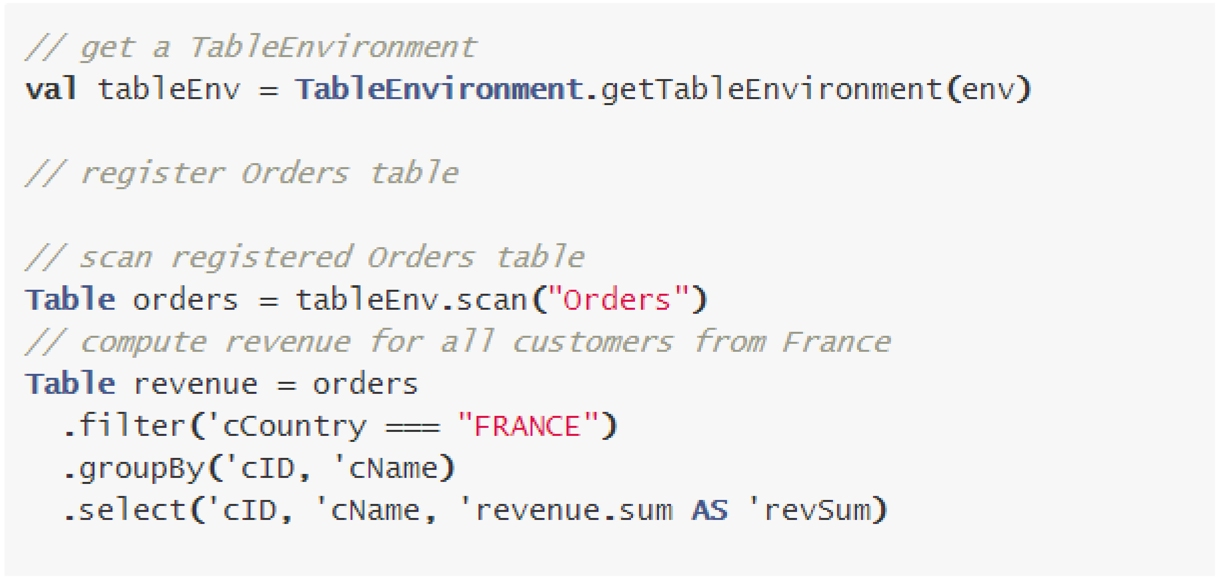
```scala
// get a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// create an external catalog
val catalog: ExternalCatalog = new InMemoryExternalCatalog

// register the ExternalCatalog catalog
tableEnv.registerExternalCatalog("InMemCatalog", catalog)
```

Once registered in a `TableEnvironment`, all tables defined in a `ExternalCatalog` can be accessed from Table API or SQL queries by specifying their full path, such as `catalog.database.table`.

Currently, Flink provides an `InMemoryExternalCatalog` for demo and testing purposes. However, the `ExternalCatalog` interface can also be used to connect catalogs like HCatalog or Metastore to the Table API.

Query a Table

使用Table API查询：



```
// get a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// register Orders table

// scan registered Orders table
Table orders = tableEnv.scan("Orders")
// compute revenue for all customers from France
Table revenue = orders
  .filter('cCountry === "FRANCE")
  .groupBy('cID, 'cName)
  .select('cID, 'cName, 'revenue.sum AS 'revSum)
```

使用方法的形式进行查询，例如：table.groupBy(...).select()

注意：

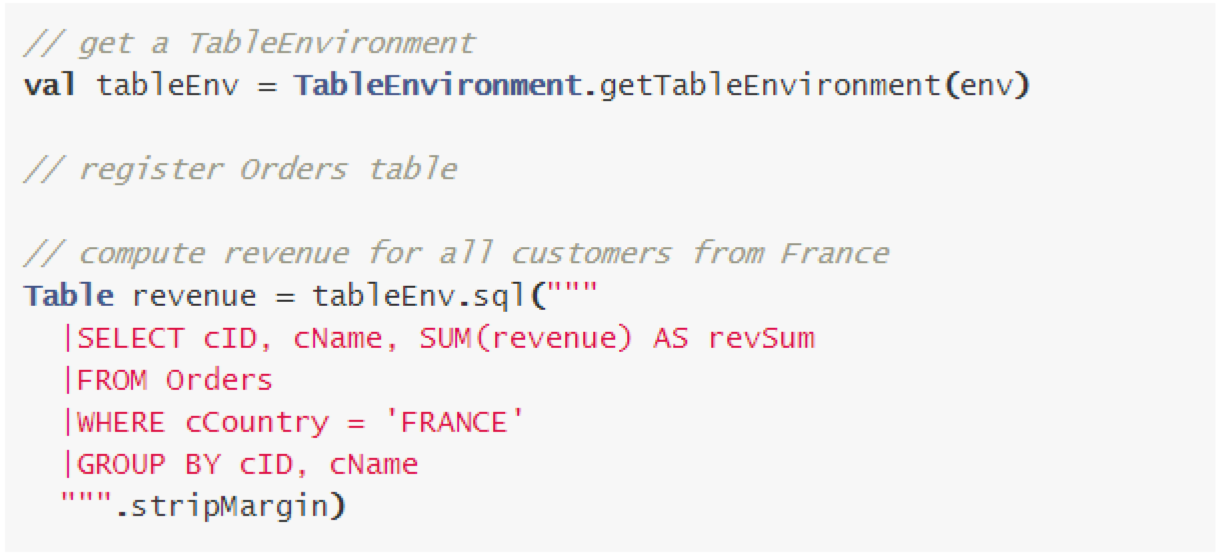
Table API 使用了Scala隐式转换，所以必须导入：

1： org.apache.flink.api.scala._

2： org.apache.flink.table.api.scala._

Query a Table

使用SQL查询：



```
// get a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// register Orders table

// compute revenue for all customers from France
Table revenue = tableEnv.sql("""
  |SELECT cID, cName, SUM(revenue) AS revSum
  |FROM Orders
  |WHERE cCountry = 'FRANCE'
  |GROUP BY cID, cName
  """.stripMargin)
```

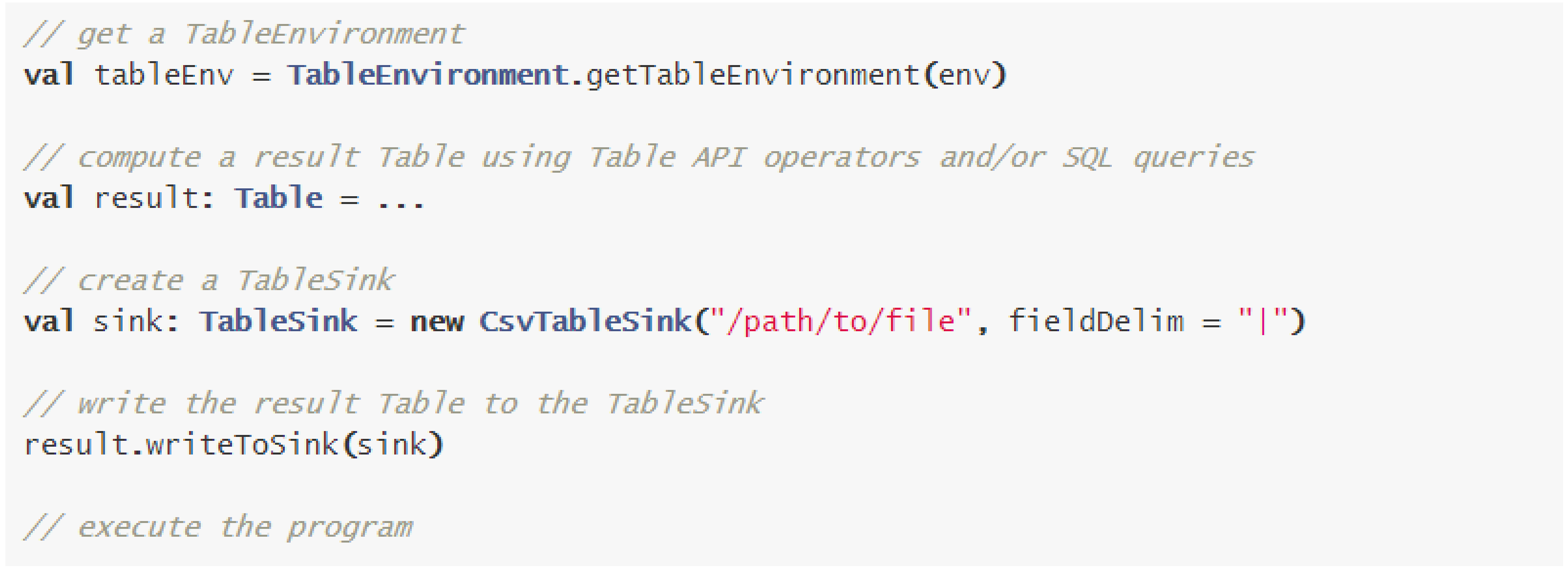
注意：(Mixing Table API and SQL)

由于**Table api**和**SQL**查询返回的对象都是**Table**，因此可以使用**SQL**和**Table API**混合查询！

Emit a Table

Emit a Table:
 为了emit table，table可以被写到TableSink中。TableSink是一个泛型接口支持各种各样的持久化。

Batch Table 仅可以被写到BatchTableSink中；然而Stream Table需要AppendStreamTableSink，RetractStreamTableSink或者UpsertStreamTableSink。



```
// get a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// compute a result Table using Table API operators and/or SQL queries
val result: Table = ...

// create a TableSink
val sink: TableSink = new CsvTableSink("/path/to/file", fieldDelim = "|")

// write the result Table to the TableSink
result.writeToSink(sink)

// execute the program
```

Translate and Execute a Query

Table API and SQL查询会被翻译成 DataStream 或 DataSet 程序执行，一个查询在内部代表一个逻辑查询计划并且被翻译成两个阶段：

1：逻辑计划的优化

2：翻译成DataStream 或 DataSet 程序

Table API 或SQL 查询何时被翻译？

1：Table被emit时候 例如： Table.writeToSink()被调用

2： Table被转成DataStream 或 DataSet时候

一旦Table API或SQL查询翻译完毕，Table API或SQL查询就和常规的DataStream or DataSet 程序一样调用当StreamExecutionEnvironment.execute() or ExecutionEnvironment.execute() 时候然后执行。

Integration with DataStream and DataSet API

由于Table和DataSet或DataStream都可相互转换，因此Table API和SQL查询程序可以嵌套DataStream 和DataSet 程序中，

Implicit Conversion for Scala

在使用Scala开发时候必须导入隐式转换：

```
org.apache.flink.table.api.scala._
org.apache.flink.api.scala._
```

Register a DataStream or DataSet as Table

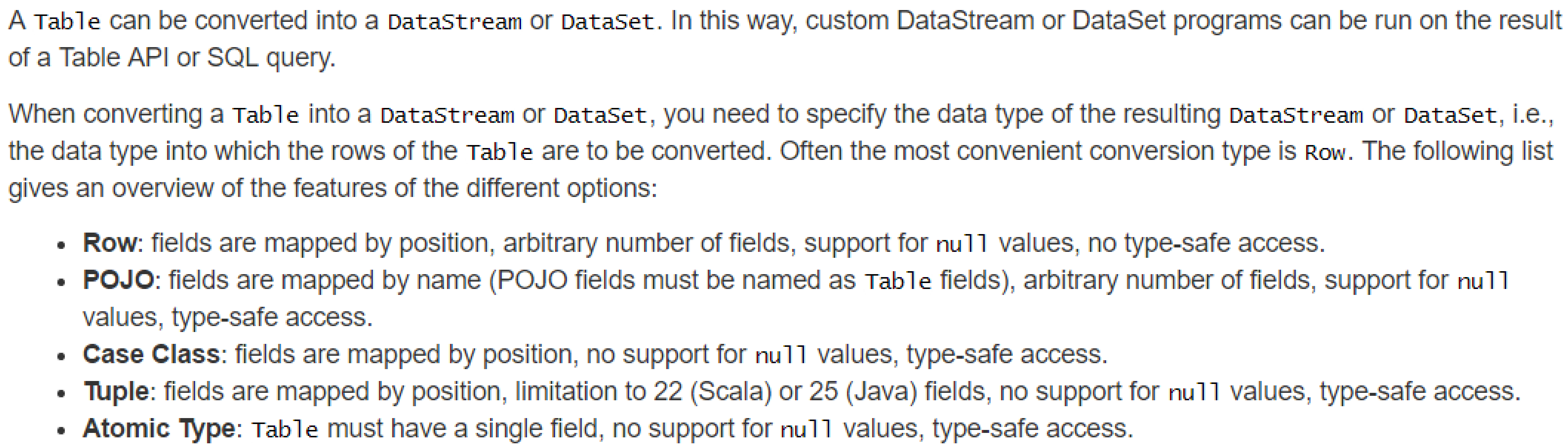
```scala
// get TableEnvironment
// registration of a DataSet is equivalent
val tableEnv = TableEnvironment.getTableEnvironment(env)

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

// register the DataStream as Table "myTable" with fields "f0", "f1"
tableEnv.registerDataStream("myTable", stream)

// register the DataStream as table "myTable2" with fields "myLong", "myString"
tableEnv.registerDataStream("myTable2", stream, 'myLong, 'myString)
```

Note: The name of a `DataStream Table` must not match the `^_DataStreamTable_[0-9]+` pattern and the name of a `DataSet Table` must not match the `^_DataSetTable_[0-9]+` pattern. These patterns are reserved for internal use only.

Convert a DataStream or DataSet into a Table

```
// get TableEnvironment
// registration of a DataSet is equivalent
val tableEnv = TableEnvironment.getTableEnvironment(env)

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

// convert the DataStream into a Table with default fields '_1, '_2
val table1: Table = tableEnv.fromDataStream(stream)

// convert the DataStream into a Table with fields 'myLong, 'myString
val table2: Table = tableEnv.fromDataStream(stream, 'myLong, 'myString)
```

Convert a Table into a DataStream or DataSet



A `Table` can be converted into a `DataStream` or `DataSet`. In this way, custom DataStream or DataSet programs can be run on the result of a Table API or SQL query.

When converting a `Table` into a `DataStream` or `DataSet`, you need to specify the data type of the resulting `DataStream` or `DataSet`, i.e., the data type into which the rows of the `Table` are to be converted. Often the most convenient conversion type is `Row`. The following list gives an overview of the features of the different options:

- **Row**: fields are mapped by position, arbitrary number of fields, support for `null` values, no type-safe access.
- **POJO**: fields are mapped by name (POJO fields must be named as `Table` fields), arbitrary number of fields, support for `null` values, type-safe access.
- **Case Class**: fields are mapped by position, no support for `null` values, type-safe access.
- **Tuple**: fields are mapped by position, limitation to 22 (Scala) or 25 (Java) fields, no support for `null` values, type-safe access.
- **Atomic Type**: `Table` must have a single field, no support for `null` values, type-safe access.

Convert a Table into a DataStream

A `Table` that is the result of a streaming query will be updated dynamically, i.e., it is changing as new records arrive on the query's input streams. Hence, the `DataStream` into which such a dynamic query is converted needs to encode the updates of the table.

There are two modes to convert a `Table` into a `DataStream`:

1. **Append Mode**: This mode can only be used if the dynamic `Table` is only modified by `INSERT` changes, i.e, it is append-only and previously emitted results are never updated.
2. **Retract Mode**: This mode can always be used. It encodes `INSERT` and `DELETE` changes with a `boolean` flag.

```
// get TableEnvironment.
// registration of a DataSet is equivalent
val tableEnv = TableEnvironment.getTableEnvironment(env)

// Table with two fields (String name, Integer age)
val table: Table = ...

// convert the Table into an append DataStream of Row
val dsRow: DataStream[Row] = tableEnv.toAppendStream[Row](table)

// convert the Table into an append DataStream of Tuple2[String, Int]
val dsTuple: DataStream[(String, Int)] dsTuple =
  tableEnv.toAppendStream[(String, Int)](table)

// convert the Table into a retract DataStream of Row.
//   A retract stream of type X is a DataStream[(Boolean, X)].
//   The boolean field indicates the type of the change.
//   True is INSERT, false is DELETE.
val retractStream: DataStream[(Boolean, Row)] = tableEnv.toRetractStream[Row](table)
```

Retract多一个flag，true表示：insert，false表示：delete

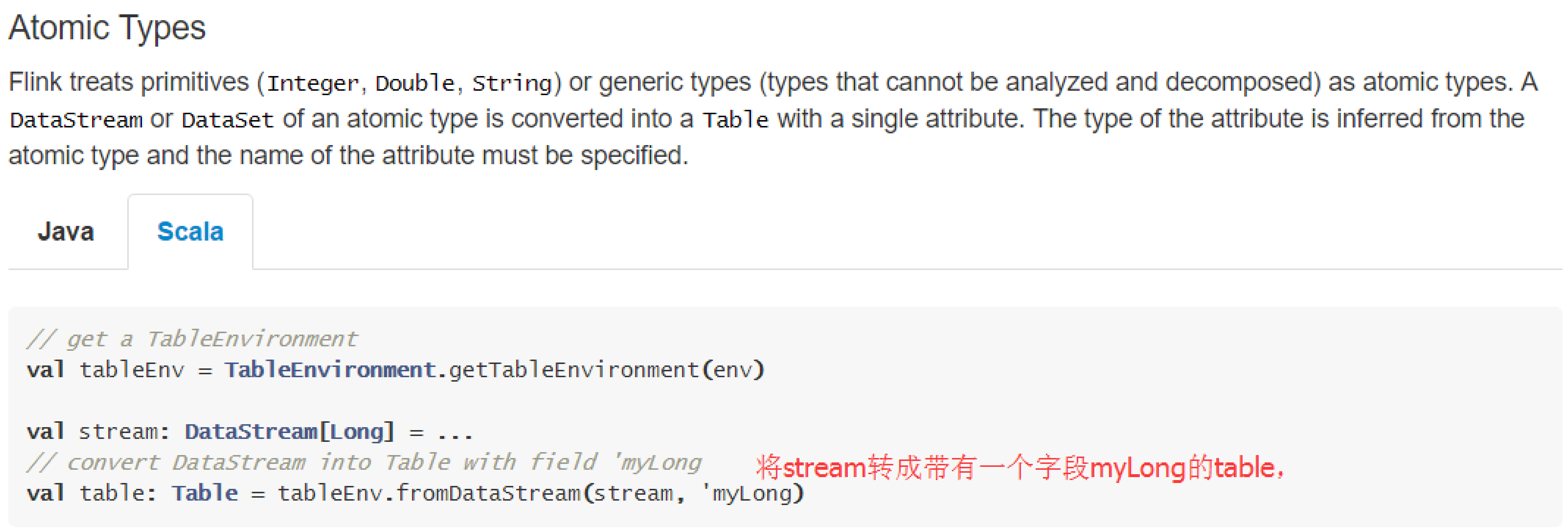
Convert a Table into a DataSet

```scala
// get TableEnvironment
// registration of a DataSet is equivalent
val tableEnv = TableEnvironment.getTableEnvironment(env)

// Table with two fields (String name, Integer age)
val table: Table = ...

// convert the Table into a DataSet of Row
val dsRow: DataSet[Row] = tableEnv.toDataSet[Row](table)

// convert the Table into a DataSet of Tuple2[String, Int]
val dsTuple: DataSet[(String, Int)] = tableEnv.toDataSet[(String, Int)](table)
```

Mapping of Data Types to Table Schema

DataSet或DataStream向Table转换时候数据类型的对应关系：

数据类型和Table Schema映射：
Flink’s DataStream and DataSet支持各种各样的数据类型，例如： Tuples (built-in Scala and Flink Java tuples), POJOs, case classes, and atomic types。



Atomic Types

Flink treats primitives (`Integer`, `Double`, `String`) or generic types (types that cannot be analyzed and decomposed) as atomic types. A `DataStream` or `DataSet` of an atomic type is converted into a `Table` with a single attribute. The type of the attribute is inferred from the atomic type and the name of the attribute must be specified.

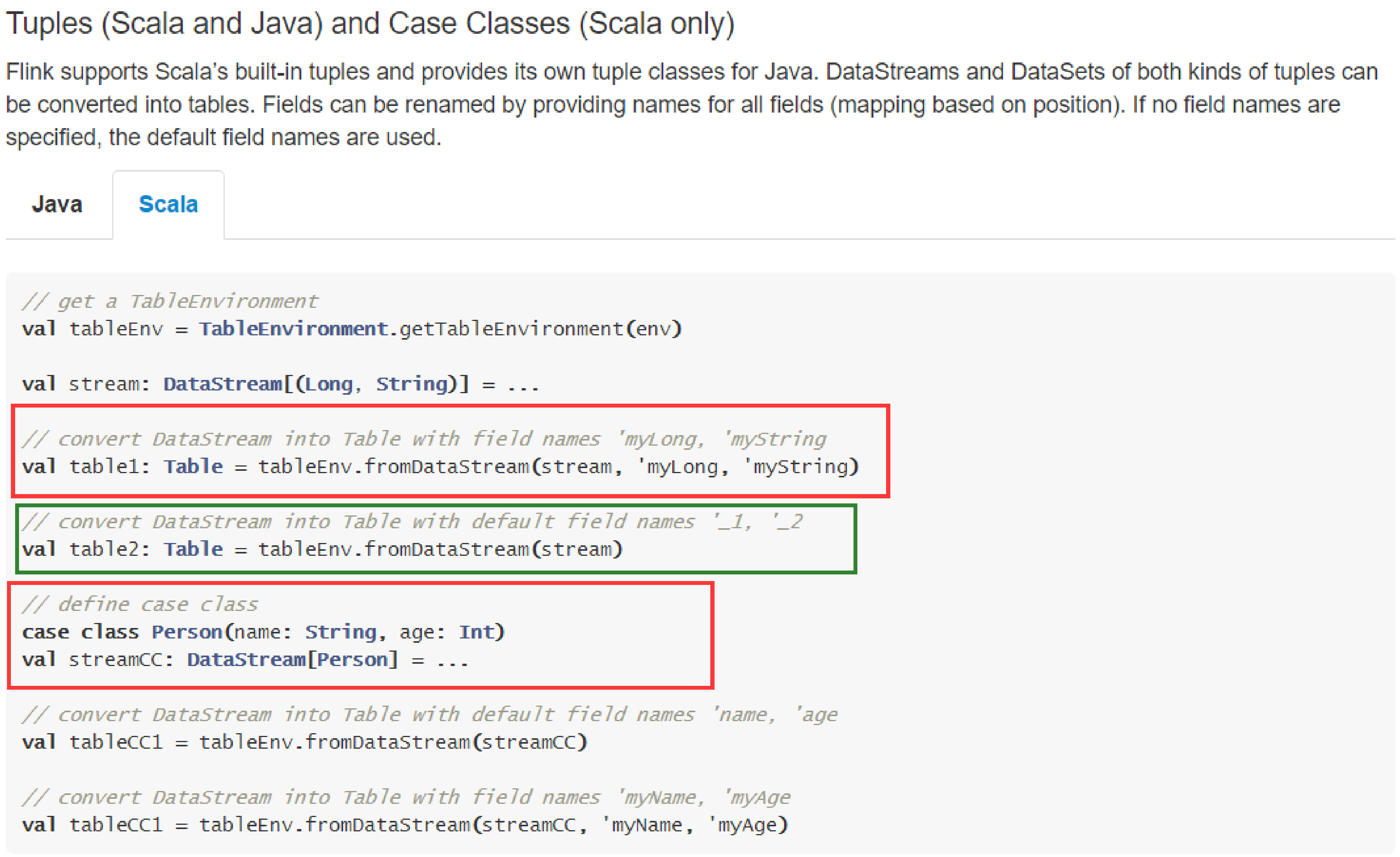
Java | **Scala**

```
// get a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

val stream: DataStream[Long] = ...
// convert DataStream into Table with field 'myLong
val table: Table = tableEnv.fromDataStream(stream, 'myLong)
```

将stream转成带有一个字段myLong的table，

Mapping of Data Types to Table Schema

DataSet或DataStream向Table转换时候数据类型的对应关系：



Tuples (Scala and Java) and Case Classes (Scala only)

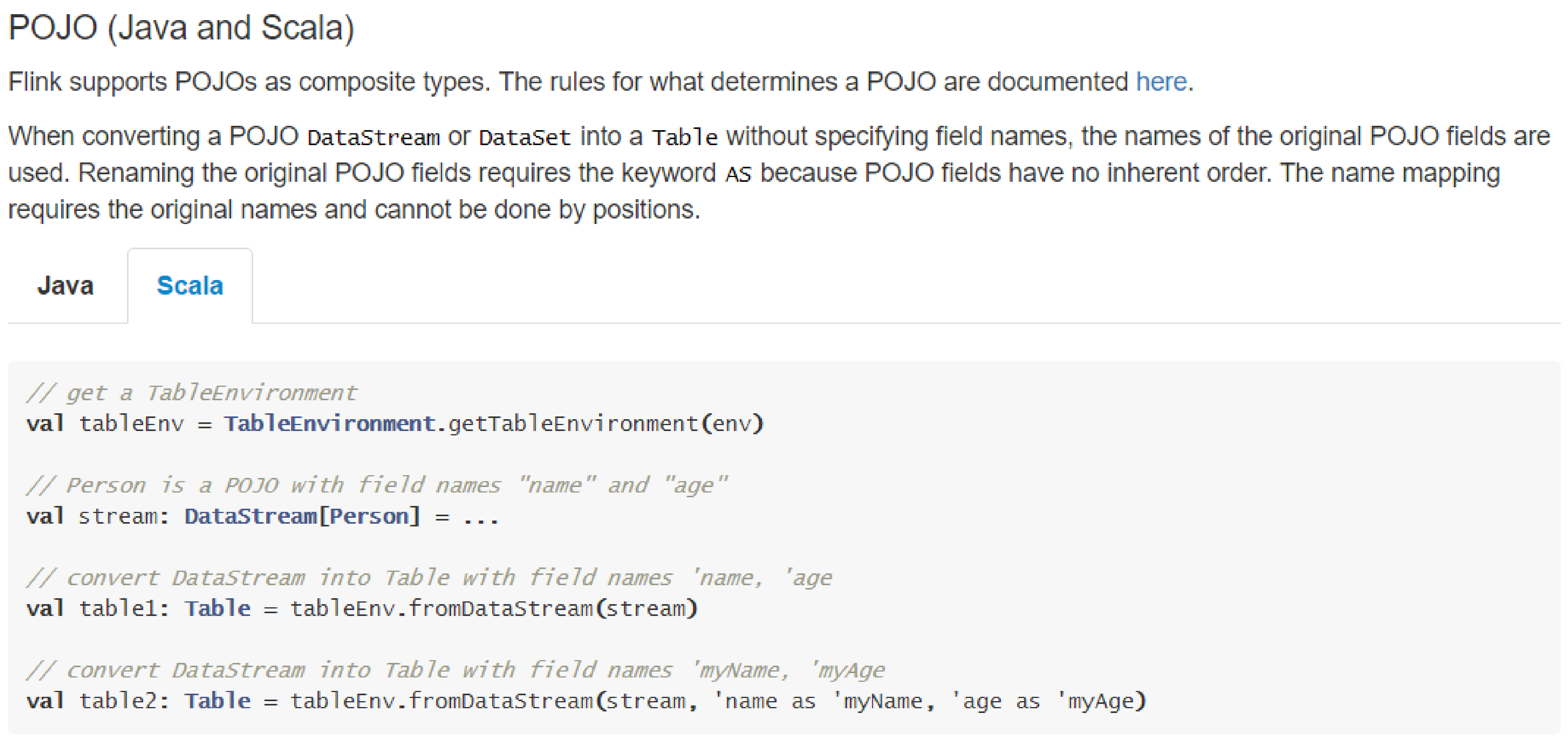
Flink supports Scala's built-in tuples and provides its own tuple classes for Java. DataStreams and DataSets of both kinds of tuples can be converted into tables. Fields can be renamed by providing names for all fields (mapping based on position). If no field names are specified, the default field names are used.

Java | **Scala**

```scala
// get a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

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

// convert DataStream into Table with field names 'myLong, 'myString
val table1: Table = tableEnv.fromDataStream(stream, 'myLong, 'myString)

// convert DataStream into Table with default field names '_1, '_2
val table2: Table = tableEnv.fromDataStream(stream)

// define case class
case class Person(name: String, age: Int)
val streamCC: DataStream[Person] = ...

// convert DataStream into Table with default field names 'name, 'age
val tableCC1 = tableEnv.fromDataStream(streamCC)

// convert DataStream into Table with field names 'myName, 'myAge
val tableCC1 = tableEnv.fromDataStream(streamCC, 'myName, 'myAge)
```

Mapping of Data Types to Table Schema

DataSet或DataStream向Table转换时候数据类型的对应关系：



POJO (Java and Scala)

Flink supports POJOs as composite types. The rules for what determines a POJO are documented here.

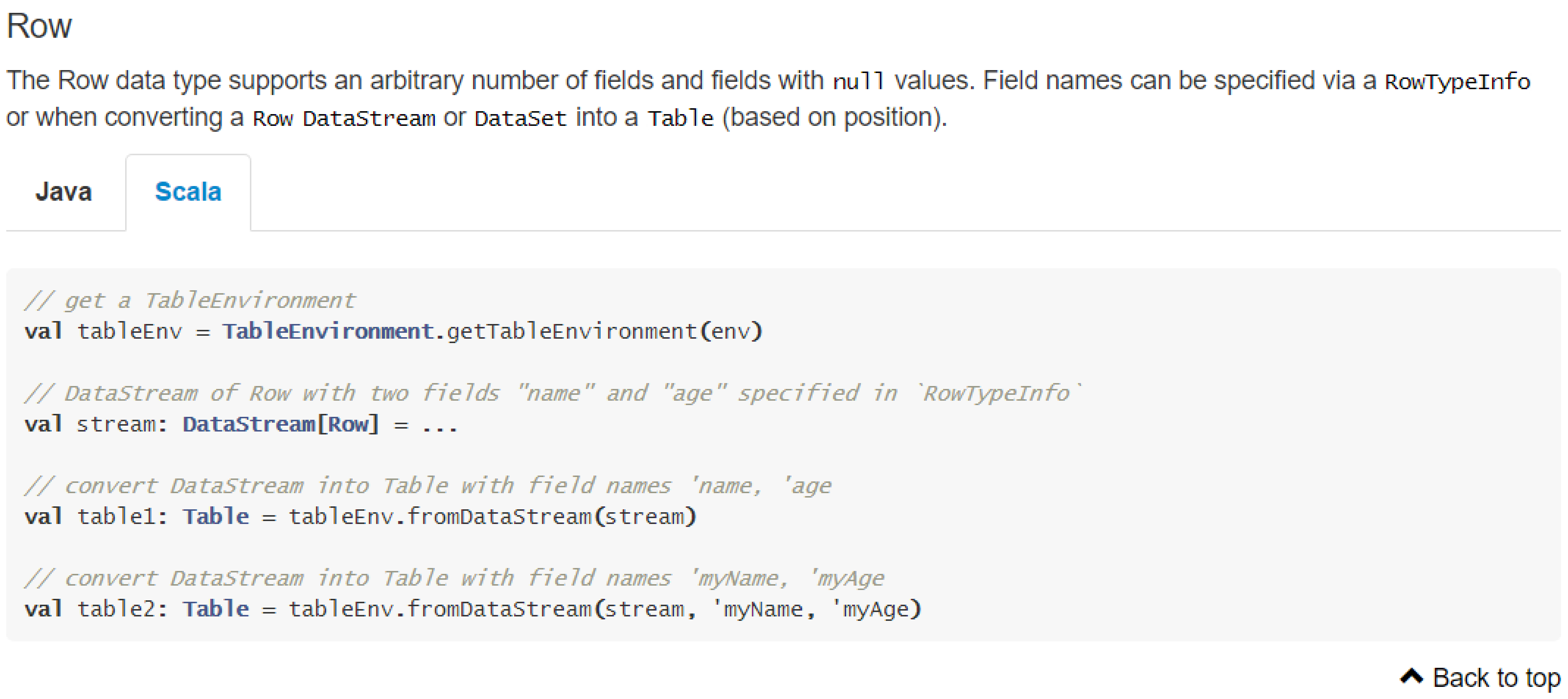
When converting a POJO `DataStream` or `DataSet` into a `Table` without specifying field names, the names of the original POJO fields are used. Renaming the original POJO fields requires the keyword `AS` because POJO fields have no inherent order. The name mapping requires the original names and cannot be done by positions.

Java | **Scala**

```scala
// get a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// Person is a POJO with field names "name" and "age"
val stream: DataStream[Person] = ...

// convert DataStream into Table with field names 'name, 'age
val table1: Table = tableEnv.fromDataStream(stream)

// convert DataStream into Table with field names 'myName, 'myAge
val table2: Table = tableEnv.fromDataStream(stream, 'name as 'myName, 'age as 'myAge)
```

Mapping of Data Types to Table Schema

DataSet或DataStream向Table转换时候数据类型的对应关系：



Row

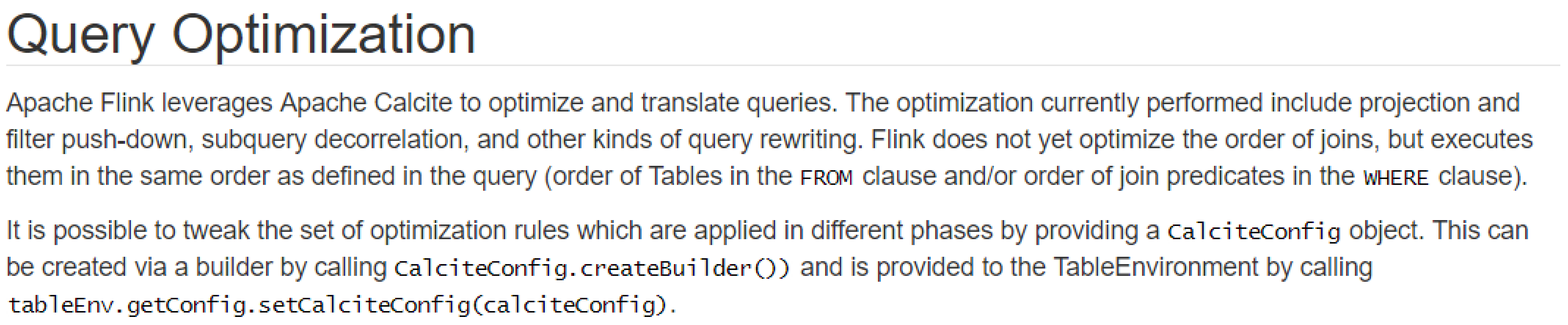
The Row data type supports an arbitrary number of fields and fields with `null` values. Field names can be specified via a `RowTypeInfo` or when converting a `Row DataStream` or `DataSet` into a `Table` (based on position).

Java | **Scala**

```scala
// get a TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// DataStream of Row with two fields "name" and "age" specified in `RowTypeInfo`
val stream: DataStream[Row] = ...

// convert DataStream into Table with field names 'name, 'age
val table1: Table = tableEnv.fromDataStream(stream)

// convert DataStream into Table with field names 'myName, 'myAge
val table2: Table = tableEnv.fromDataStream(stream, 'myName, 'myAge)
```

Back to top

Query Optimization



Query Optimization

Apache Flink leverages Apache Calcite to optimize and translate queries. The optimization currently performed include projection and filter push-down, subquery decorrelation, and other kinds of query rewriting. Flink does not yet optimize the order of joins, but executes them in the same order as defined in the query (order of Tables in the `FROM` clause and/or order of join predicates in the `WHERE` clause).

It is possible to tweak the set of optimization rules which are applied in different phases by providing a `CalciteConfig` object. This can be created via a builder by calling `CalciteConfig.createBuilder())` and is provided to the TableEnvironment by calling `tableEnv.getConfig.setCalciteConfig(calciteConfig)`.

Explaining a Table

The Table API provides a mechanism to explain the logical and optimized query plans to compute a `Table`. This is done through the `TableEnvironment.explain(table)` method. It returns a String describing three plans:

1. the Abstract Syntax Tree of the relational query, i.e., the unoptimized logical query plan,
2. the optimized logical query plan, and
3. the physical execution plan.

The following code shows an example and the corresponding output:

Java **Scala**

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tEnv = TableEnvironment.getTableEnvironment(env)

val table1 = env.fromElements((1, "hello")).toTable(tEnv, 'count, 'word)
val table2 = env.fromElements((1, "hello")).toTable(tEnv, 'count, 'word)
val table = table1
  .where('word.like("F%"))
  .unionAll(table2)

val explanation: String = tEnv.explain(table)
println(explanation)
```