Flink Table API & SQL编程指南(1)

Apache Flink提供了两种顶层的关系型API，分别为Table API和SQL，Flink通过Table API&SQL实现了批流统一。其中Table API是用于Scala和Java的语言集成查询API，它允许以非常直观的方式组合关系运算符（例如select，where和join）的查询。Flink SQL基于Apache Calcite 实现了标准的SQL，用户可以使用标准的SQL处理数据集。Table API和SQL与Flink的DataStream和DataSet API紧密集成在一起，用户可以实现相互转化，比如可以将DataStream或者DataSet注册为table进行操作数据。值得注意的是，**Table API and SQL**目前尚未完全完善，还在积极的开发中，所以并不是所有的算子操作都可以通过其实现。

依赖

从Flink1.9开始，Flink为Table & SQL API提供了两种planner,分别为Blink planner和old planner，其中old planner是在Flink1.9之前的版本使用。主要区别如下：

尖叫提示：对于生产环境，目前推荐使用old planner.

- `flink-table-common`: 通用模块，包含 Flink Planner 和 Blink Planner 一些共用的代码
- `flink-table-api-java`: java语言的Table & SQL API，仅针对table(处于早期的开发阶段，不推荐使用)
- `flink-table-api-scala`: scala语言的Table & SQL API，仅针对table(处于早期的开发阶段，不推荐使用)
- `flink-table-api-java-bridge`: java语言的Table & SQL API，支持DataStream/DataSet API(推荐使用)
- `flink-table-api-scala-bridge`: scala语言的Table & SQL API，支持DataStream/DataSet API(推荐使用)
- `flink-table-planner`:planner 和runtime. planner为Flink1,9之前的old planner(推荐使用)
- `flink-table-planner-blink`: 新的Blink planner.
- `flink-table-runtime-blink`: 新的Blink runtime.
- `flink-table-uber`: 将上述的API模块及old planner打成一个jar包，形如flink-table-*.jar，位与/lib目录下
- `flink-table-uber-blink`:将上述的API模块及Blink 模块打成一个jar包，形如fflink-table-blink-*.jar，位与/lib目录下

Blink planner & old planner

Blink planner和old planner有许多不同的特点，具体列举如下：

- Blink planner将批处理作业看做是流处理作业的特例。所以，不支持Table 与DataSet之间的转换，批处理的作业也不会被转成DataSet程序，而是被转为DataStream程序。
- Blink planner不支持 `BatchTableSource`，使用的是有界的StreamTableSource。
- Blink planner仅支持新的 `Catalog`，不支持`ExternalCatalog` (已过时)。
- 对于FilterableTableSource的实现，两种Planner是不同的。old planner会谓词下推到`PlannerExpression`(未来会被移除)，而Blink planner 会谓词下推到 `Expression`(表示一个产生计算结果的逻辑树)。
- 仅仅Blink planner支持key-value形式的配置，即通过Configuration进行参数设置。
- 关于PlannerConfig的实现，两种planner有所不同。
- Blink planner 会将多个sink优化成一个DAG(仅支持TableEnvironment，StreamTableEnvironment不支持)，old planner总是将每一个sink优化成一个新的DAG，每一个DAG都是相互独立的。
- old planner不支持catalog统计，Blink planner支持catalog统计。

Flink Table & SQL程序的pom依赖

根据使用的语言不同，可以选择下面的依赖，包括scala版和java版，如下：

```
<!-- java版 -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java-bridge_2.11</artifactId>
  <version>1.10.0</version>
  <scope>provided</scope>
</dependency>
<!-- scala版 -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-scala-bridge_2.11</artifactId>
  <version>1.10.0</version>
  <scope>provided</scope>
</dependency>
```

除此之外，如果需要在本地的IDE中运行Table API & SQL的程序，则需要添加下面的pom依赖：

```xml
<!-- Flink 1.9之前的old planner -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner_2.11</artifactId>
  <version>1.10.0</version>
  <scope>provided</scope>
</dependency>
<!-- 新的Blink planner -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-blink_2.11</artifactId>
  <version>1.10.0</version>
  <scope>provided</scope>
</dependency>
```

另外，如果需要实现自定义的格式(比如和kafka交互)或者用户自定义函数，需要添加如下依赖：

```xml
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-common</artifactId>
  <version>1.10.0</version>
  <scope>provided</scope>
</dependency>
```

Table API & SQL的编程模板

所有的Table API&SQL的程序(无论是批处理还是流处理)都有着相同的形式，下面将给出通用的编程结构形式：

```java
// 创建一个TableEnvironment对象，指定planner、处理模式(batch、streaming)
TableEnvironment tableEnv = ...;
// 创建一个表
tableEnv.connect(...).createTemporaryTable("table1");
// 注册一个外部的表
tableEnv.connect(...).createTemporaryTable("outputTable");
// 通过Table API的查询创建一个Table 对象
Table tapiResult = tableEnv.from("table1").select(...);
// 通过SQL查询的查询创建一个Table 对象
Table sqlResult  = tableEnv.sqlQuery("SELECT ... FROM table1 ... ");
// 将结果写入TableSink
tapiResult.insertInto("outputTable");
// 执行
tableEnv.execute("java_job");
```

注意：Table API & SQL的查询可以相互集成，另外还可以在DataStream或者DataSet中使用Table API & SQL的API，实现DataStreams、 DataSet与Table之间的相互转换。

创建TableEnvironment

TableEnvironment是Table API & SQL程序的一个入口，主要包括如下的功能：

- 在内部的catalog中注册Table
- 注册catalog
- 加载可插拔模块
- 执行SQL查询
- 注册用户定义函数
- `DataStream` 、 `DataSet` 与Table之间的相互转换
- 持有对 `ExecutionEnvironment` 、 `StreamExecutionEnvironment` 的引用

一个Table必定属于一个具体的TableEnvironment，不可以将不同TableEnvironment的表放在一起使用(比如join，union等操作)。

TableEnvironment是通过调用 `BatchTableEnvironment.create()` 或者StreamTableEnvironment.create()的静态方法进行创建的。另外，默认两个planner的jar包都存在与classpath下，所有需要明确指定使用的planner。

```
// **********************
// FLINK 流处理查询
// **********************
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironm
ent;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.java.StreamTableEnvironment;

EnvironmentSettings fsSettings = EnvironmentSettings.newInstance().useOl
dPlanner().inStreamingMode().build();
StreamExecutionEnvironment fsEnv = StreamExecutionEnvironment.getExecuti
onEnvironment();
StreamTableEnvironment fsTableEnv = StreamTableEnvironment.create(fsEnv,
 fsSettings);
//或者TableEnvironment fsTableEnv = TableEnvironment.create(fsSettings);

// ********************
// FLINK 批处理查询
// ********************
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.table.api.java.BatchTableEnvironment;

```

```
23
24  ExecutionEnvironment fbEnv = ExecutionEnvironment.getExecutionEnvironmen
25  t();
26  BatchTableEnvironment fbTableEnv = BatchTableEnvironment.create(fbEnv);
27
28  // **********************
29  // BLINK 流处理查询
30  // **********************
31  import org.apache.flink.streaming.api.environment.StreamExecutionEnvironm
32  ent;
33  import org.apache.flink.table.api.EnvironmentSettings;
34  import org.apache.flink.table.api.java.StreamTableEnvironment;
35
36  StreamExecutionEnvironment bsEnv = StreamExecutionEnvironment.getExecuti
37  onEnvironment();
38  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBl
39  inkPlanner().inStreamingMode().build();
40  StreamTableEnvironment bsTableEnv = StreamTableEnvironment.create(bsEnv,
41   bsSettings);
    // 或者 TableEnvironment bsTableEnv = TableEnvironment.create(bsSettings)
    ;

    // ********************
    // BLINK 批处理查询
    // ********************
    import org.apache.flink.table.api.EnvironmentSettings;
    import org.apache.flink.table.api.TableEnvironment;

    EnvironmentSettings bbSettings = EnvironmentSettings.newInstance().useBl
    inkPlanner().inBatchMode().build();
    TableEnvironment bbTableEnv = TableEnvironment.create(bbSettings);
```

在catalog中创建表

临时表与永久表

表可以分为临时表和永久表两种，其中永久表需要一个catalog(比如Hive的Metastore)俩维护表的元数据信息，一旦永久表被创建，只要连接到该catalog就可以访问该表，只有显示删除永久表，该表才可以被删除。临时表的生命周期是Flink Session，这些表不能够被其他的Flink Session访问，这些表不属于任何的catalog或者数据库，如果与临时表相对应的数据库被删除了，该临时表也不会被删除。

创建表

虚表(Virtual Tables)

一个Table对象相当于SQL中的视图(虚表)，它封装了一个逻辑执行计划，可以通过一个catalog创建，具体如下：

```
// 获取一个TableEnvironment
TableEnvironment tableEnv = ...;
// table对象，查询的结果集
Table projTable = tableEnv.from("X").select(...);
// 注册一个表，名称为 "projectedTable"
tableEnv.createTemporaryView("projectedTable", projTable);
```

外部数据源表(Connector Tables)

可以把外部的数据源注册成表，比如可以读取MySQL数据库数据、Kafka数据等

```
tableEnvironment
  .connect(...)
  .withFormat(...)
  .withSchema(...)
  .inAppendMode()
  .createTemporaryTable("MyTable")
```

扩展创建表的标识属性

表的注册总是包含三部分标识属性：catalog、数据库、表名。用户可以在内部设置一个catalog和一个数据库作为当前的catalog和数据库，所以对于catalog和数据库这两个标识属性是可选的，即如果不指定，默认使用的是“current catalog”和 “current database”。

```
TableEnvironment tEnv = ...;
tEnv.useCatalog("custom_catalog");//设置catalog
tEnv.useDatabase("custom_database");//设置数据库
Table table = ...;
// 注册一个名为exampleView的视图，catalog名为custom_catalog
// 数据库的名为custom_database
tableEnv.createTemporaryView("exampleView", table);

// 注册一个名为exampleView的视图，catalog的名为custom_catalog
// 数据库的名为other_database
tableEnv.createTemporaryView("other_database.exampleView", table);

// 注册一个名为'View'的视图，catalog的名称为custom_catalog
// 数据库的名为custom_database，'View'是保留关键字，需要使用``(反引号)
tableEnv.createTemporaryView("`View`", table);

// 注册一个名为example.View的视图，catalog的名为custom_catalog，
// 数据库名为custom_database
```

```
tableEnv.createTemporaryView("`example.View`", table);

// 注册一个名为'exampleView'的视图， catalog的名为'other_catalog'
// 数据库名为other_database'
tableEnv.createTemporaryView("other_catalog.other_database.exampleView",
 table);
```

查询表

Table API

Table API是一个集成Scala与Java语言的查询API，与SQL相比，它的查询不是一个标准的SQL语句，而是由一步一步的操作组成的。如下展示了一个使用Table API实现一个简单的聚合查询。

```
// 获取TableEnvironment
TableEnvironment tableEnv = ...;
//注册Orders表

// 查询注册的表
Table orders = tableEnv.from("Orders");
// 计算操作
Table revenue = orders
  .filter("cCountry === 'FRANCE'")
  .groupBy("cID, cName")
  .select("cID, cName, revenue.sum AS revSum");
```

SQL

Flink SQL依赖于Apache Calcite，其实现了标准的SQL语法，如下案例：

```
// 获取TableEnvironment
TableEnvironment tableEnv = ...;

//注册Orders表

// 计算逻辑同上面的Table API
Table revenue = tableEnv.sqlQuery(
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY cID, cName"
  );

// 注册"RevenueFrance"外部输出表

```

```
// 计算结果插入"RevenueFrance"表
tableEnv.sqlUpdate(
    "INSERT INTO RevenueFrance " +
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY cID, cName"
  );
```

输出表

一个表通过将其写入到TableSink，然后进行输出。TableSink是一个通用的支持多种文件格式(CSV、Parquet, Avro)和多种外部存储系统(JDBC, Apache HBase, Apache Cassandra, Elasticsearch)以及多种消息对列(Apache Kafka, RabbitMQ)的接口。

批处理的表只能被写入到 `BatchTableSink`,流处理的表需要指明AppendStreamTableSink、RetractStreamTableSink或者 `UpsertStreamTableSink`

```
// 获取TableEnvironment
TableEnvironment tableEnv = ...;

// 创建输出表
final Schema schema = new Schema()
    .field("a", DataTypes.INT())
    .field("b", DataTypes.STRING())
    .field("c", DataTypes.LONG());

tableEnv.connect(new FileSystem("/path/to/file"))
    .withFormat(new Csv().fieldDelimiter('|').deriveSchema())
    .withSchema(schema)
    .createTemporaryTable("CsvSinkTable");

// 计算结果表
Table result = ...
// 输出结果表到注册的TableSink
result.insertInto("CsvSinkTable");
```

Table API & SQL底层的转换与执行

上文提到了Flink提供了两种planner，分别为old planner和Blink planner，对于不同的planner而言，Table API & SQL底层的执行与转换是有所不同的。

Old planner

根据是流处理作业还是批处理作业，Table API &SQL会被转换成DataStream或者DataSet程序。一个查询在内部表示为一个逻辑查询计划，会被转换为两个阶段:

- 1.逻辑查询计划优化
- 2.转换成DataStream或者DataSet程序

上面的两个阶段只有下面的操作被执行时才会被执行：

- 当一个表被输出到TableSink时，比如调用了Table.insertInto()方法
- 当执行更新查询时，比如调用TableEnvironment.sqlUpdate()方法
- 当一个表被转换为DataStream或者DataSet时

一旦执行上述两个阶段，Table API & SQL的操作会被看做是普通的DataStream或者DataSet程序，所以当`StreamExecutionEnvironment.execute()`或者`ExecutionEnvironment.execute()`被调用时，会执行转换后的程序。

Blink planner

无论是批处理作业还是流处理作业，如果使用的是Blink planner，底层都会被转换为DataStream程序。在一个查询在内部表示为一个逻辑查询计划，会被转换成两个阶段：

- 1.逻辑查询计划优化
- 2.转换成DataStream程序

对于`TableEnvironment` and `StreamTableEnvironment`而言，一个查询的转换是不同的

首先对于TableEnvironment，当TableEnvironment.execute()方法执行时，Table API & SQL的查询才会被转换，因为TableEnvironment会将多个sink优化为一个DAG。

对于StreamTableEnvironment，转换发生的时间与old planner相同。

与DataStream & DataSet API集成

对于Old planner与Blink planner而言，只要是流处理的操作，都可以与DataStream API集成，**仅仅只有Old planner才可以与DataSet API集成**，由于Blink planner的批处理作业会被转换成DataStream程序，所以不能够与DataSet API集成。值得注意的是，下面提到的table与DataSet之间的转换仅适用于Old planner。

Table API & SQL的查询很容易与DataStream或者DataSet程序集成，并可以将Table API & SQL的查询嵌入DataStream或者DataSet程序中。DataStream或者DataSet可以转换成表，反之，表也可以被转换成DataStream或者DataSet。

从DataStream或者DataSet中注册临时表(视图)

尖叫提示：只能将DataStream或者DataSet转换为临时表(视图)

下面演示DataStream的转换，对于DataSet的转换类似。

```
// 获取StreamTableEnvironment
StreamTableEnvironment tableEnv = ...;
DataStream<Tuple2<Long, String>> stream = ...
// 将DataStream注册为一个名为myTable的视图，其中字段分别为"f0", "f1"
tableEnv.createTemporaryView("myTable", stream);
// 将DataStream注册为一个名为myTable2的视图,其中字段分别为"myLong", "myString"
tableEnv.createTemporaryView("myTable2", stream, "myLong, myString");
```

将DataStream或者DataSet转化为Table对象

可以直接将DataStream或者DataSet转换为Table对象，之后可以使用Table API进行查询操作。下面演示DataStream的转换，对于DataSet的转换类似。

```
// 获取StreamTableEnvironment
StreamTableEnvironment tableEnv = ...;
DataStream<Tuple2<Long, String>> stream = ...
// 将DataStream转换为Table对象，默认的字段为"f0", "f1"
Table table1 = tableEnv.fromDataStream(stream);
// 将DataStream转换为Table对象，默认的字段为"myLong", "myString"
Table table2 = tableEnv.fromDataStream(stream, "myLong, myString");
```

将表转换为DataStream或者DataSet

当将Table转为DataStream或者DataSet时，需要指定DataStream或者DataSet的数据类型。通常最方便的数据类型是row类型，Flink提供了很多的数据类型供用户选择，具体包括Row、POJO、样例类、Tuple和原子类型。

将表转换为DataStream

一个流处理查询的结果是动态变化的，所以将表转为DataStream时需要指定一个更新模式，共有两种模式：**Append Mode**和**Retract Mode**。

- **Append Mode**

如果动态表仅只有Insert操作，即之前输出的结果不会被更新，则使用该模式。如果更新或删除操作使用追加模式会失败报错

- **Retract Mode**

始终可以使用此模式。返回值是boolean类型。它用true或false来标记数据的插入和撤回，返回true代表数据插入，false代表数据的撤回。

```
// 获取StreamTableEnvironment.
StreamTableEnvironment tableEnv = ...;
// 包含两个字段的表(String name, Integer age)
Table table = ...
// 将表转为DataStream，使用Append Mode追加模式，数据类型为Row
DataStream<Row> dsRow = tableEnv.toAppendStream(table, Row.class);
// 将表转为DataStream，使用Append Mode追加模式，数据类型为定义好的TypeInformati
on
TupleTypeInfo<Tuple2<String, Integer>> tupleType = new TupleTypeInfo<>(
  Types.STRING(),
  Types.INT());
DataStream<Tuple2<String, Integer>> dsTuple =
  tableEnv.toAppendStream(table, tupleType);
// 将表转为DataStream，使用的模式为Retract Mode撤回模式，类型为Row
// 对于转换后的DataStream<Tuple2<Boolean, X>>，X表示流的数据类型，
// boolean值表示数据改变的类型，其中INSERT返回true，DELETE返回的是false
DataStream<Tuple2<Boolean, Row>> retractStream =
  tableEnv.toRetractStream(table, Row.class);
```

将表转换为DataSet

```
// 获取BatchTableEnvironment
BatchTableEnvironment tableEnv = BatchTableEnvironment.create(env);
// 包含两个字段的表(String name, Integer age)
Table table = ...
// 将表转为DataSet数据类型为Row
DataSet<Row> dsRow = tableEnv.toDataSet(table, Row.class);
// 将表转为DataSet，通过TypeInformation定义Tuple2<String, Integer>数据类型
TupleTypeInfo<Tuple2<String, Integer>> tupleType = new TupleTypeInfo<>(
  Types.STRING(),
  Types.INT());
DataSet<Tuple2<String, Integer>> dsTuple =
  tableEnv.toDataSet(table, tupleType);
```

表的Schema与数据类型之间的映射

表的Schema与数据类型之间的映射有两种方式：分别是基于字段下标位置的映射和基于字段名称的映射。

基于字段下标位置的映射

该方式是按照字段的顺序进行一一映射，使用方式如下：

```
// 获取StreamTableEnvironment
StreamTableEnvironment tableEnv = ...;
DataStream<Tuple2<Long, Integer>> stream = ...
// 将DataStream转为表，默认的字段名为"f0"和"f1"
Table table = tableEnv.fromDataStream(stream);
// 将DataStream转为表，选取tuple的第一个元素，指定一个名为"myLong"的字段名
Table table = tableEnv.fromDataStream(stream, "myLong");
// 将DataStream转为表，为tuple的第一个元素指定名为"myLong"，为第二个元素指定myInt
的字段名
Table table = tableEnv.fromDataStream(stream, "myLong, myInt");
```

基于字段名称的映射

基于字段名称的映射方式支持任意的数据类型包括POJO类型，可以很灵活地定义表Schema映射，所有的字段被映射成一个具体的字段名称，同时也可以使用"as"为字段起一个别名。其中Tuple元素的第一个元素为f0,第二个元素为f1，以此类推。

```
// 获取StreamTableEnvironment
StreamTableEnvironment tableEnv = ...;
DataStream<Tuple2<Long, Integer>> stream = ...
// 将DataStream转为表，默认的字段名为"f0"和"f1"
Table table = tableEnv.fromDataStream(stream);
// 将DataStream转为表，选择tuple的第二个元素，指定一个名为"f1"的字段名
Table table = tableEnv.fromDataStream(stream, "f1");
// 将DataStream转为表，交换字段的顺序
Table table = tableEnv.fromDataStream(stream, "f1, f0");
// 将DataStream转为表，交换字段的顺序，并为f1起别名为"myInt"，为f0起别名为"myLong
Table table = tableEnv.fromDataStream(stream, "f1 as myInt, f0 as myLong
");
```

原子类型

Flink将`Integer`，`Double`，`String`或者普通的类型称之为原子类型，一个数据类型为原子类型的DataStream或者DataSet可以被转成单个字段属性的表，这个字段的类型与DataStream或者DataSet的数据类型一致，这个字段的名称可以进行指定。

```
//获取StreamTableEnvironment
StreamTableEnvironment tableEnv = ...;
// 数据类型为原子类型Long
DataStream<Long> stream = ...
// 将DataStream转为表，默认的字段名为"f0"
Table table = tableEnv.fromDataStream(stream);
// 将DataStream转为表，指定字段名为myLong"
Table table = tableEnv.fromDataStream(stream, "myLong");
```

Tuple类型

Tuple类型的DataStream或者DataSet都可以转为表，可以重新设定表的字段名(即根据tuple元素的位置进行一一映射，转为表之后，每个元素都有一个别名)，如果不为字段指定名称，则使用默认的名称(java语言默认的是f0,f1,scala默认的是_1),用户也可以重新排列字段的顺序，并为每个字段起一个别名。

```
// 获取StreamTableEnvironment
StreamTableEnvironment tableEnv = ...;
//Tuple2<Long, String>类型的DataStream
DataStream<Tuple2<Long, String>> stream = ...
// 将DataStream转为表，默认的字段名为 "f0", "f1"
Table table = tableEnv.fromDataStream(stream);
// 将DataStream转为表，指定字段名为 "myLong", "myString"(按照Tuple元素的顺序位置)
Table table = tableEnv.fromDataStream(stream, "myLong, myString");
// 将DataStream转为表，指定字段名为 "f0", "f1"，并且交换顺序
Table table = tableEnv.fromDataStream(stream, "f1, f0");
// 将DataStream转为表，只选择Tuple的第二个元素，指定字段名为"f1"
Table table = tableEnv.fromDataStream(stream, "f1");
// 将DataStream转为表，为Tuple的第二个元素指定别名为myString，为第一个元素指定字段名为myLong
Table table = tableEnv.fromDataStream(stream, "f1 as 'myString', f0 as 'myLong'");
```

POJO类型

当将POJO类型的DataStream或者DataSet转为表时，如果不指定表名，则默认使用的是POJO字段本身的名称，原始字段名称的映射需要指定原始字段的名称，可以为其起一个别名，也可以调换字段的顺序，也可以只选择部分的字段。

```
// 获取StreamTableEnvironment
StreamTableEnvironment tableEnv = ...;
//数据类型为Person的POJO类型，字段包括"name"和"age"
DataStream<Person> stream = ...
// 将DataStream转为表，默认的字段名称为"age", "name"
Table table = tableEnv.fromDataStream(stream);
//  将DataStream转为表，为"age"字段指定别名myAge，为"name"字段指定别名myName
Table table = tableEnv.fromDataStream(stream, "age as myAge, name as myName");
//  将DataStream转为表，只选择一个name字段
Table table = tableEnv.fromDataStream(stream, "name");
//  将DataStream转为表，只选择一个name字段，并起一个别名myName
Table table = tableEnv.fromDataStream(stream, "name as myName");
```

Row类型

Row类型的DataStream或者DataSet转为表的过程中，可以根据字段的位置或者字段名称进行映射，同时也可以为字段起一个别名，或者只选择部分字段。

```
// 获取StreamTableEnvironment
StreamTableEnvironment tableEnv = ...;
// Row类型的DataStream，通过RowTypeInfo指定两个字段"name"和"age"
DataStream<Row> stream = ...
// 将DataStream转为表，默认的字段名为原始字段名"name"和"age"
Table table = tableEnv.fromDataStream(stream);
// 将DataStream转为表，根据位置映射，为第一个字段指定myName别名，为第二个字段指定myAge别名
Table table = tableEnv.fromDataStream(stream, "myName, myAge");
// 将DataStream转为表，根据字段名映射，为name字段起别名myName，为age字段起别名myAge
Table table = tableEnv.fromDataStream(stream, "name as myName, age as myAge");
// 将DataStream转为表，根据字段名映射，只选择name字段
Table table = tableEnv.fromDataStream(stream, "name");
// 将DataStream转为表，根据字段名映射，只选择name字段，并起一个别名"myName"
Table table = tableEnv.fromDataStream(stream, "name as myName");
```

查询优化

Old planner

Apache Flink利用Apache Calcite来优化和转换查询。当前执行的优化包括投影和过滤器下推，去相关子查询以及其他类型的查询重写。Old Planner目前不支持优化JOIN的顺序，而是按照查询中定义的顺序执行它们。

通过提供一个`CalciteConfig`对象，可以调整在不同阶段应用的优化规则集。这可通过调用`CalciteConfig.createBuilder()`方法来进行创建，并通过调用`tableEnv.getConfig.setPlannerConfig(calciteConfig)`方法将该对象传递给TableEnvironment。

Blink planner

Apache Flink利用并扩展了Apache Calcite来执行复杂的查询优化。这包括一系列基于规则和基于成本的优化(cost_based)，例如：

- 基于Apache Calcite的去相关子查询
- 投影裁剪
- 分区裁剪

- 过滤器谓词下推
- 过滤器下推
- 子计划重复数据删除以避免重复计算
- 特殊的子查询重写，包括两个部分：
  - 将IN和EXISTS转换为左半联接(left semi-join)
  - 将NOT IN和NOT EXISTS转换为left anti-join
- 调整join的顺序，需要启用 `table.optimizer.join-reorder-enabled`

注意： IN / EXISTS / NOT IN / NOT EXISTS当前仅在子查询重写的结合条件下受支持。

查询优化器不仅基于计划，而且还可以基于数据源的统计信息以及每个操作的细粒度开销(例如io，cpu，网络和内存）,从而做出更加明智且合理的优化决策。

高级用户可以通过`CalciteConfig`对象提供自定义优化规则，通过调用tableEnv.getConfig.setPlannerConfig(calciteConfig)，将参数传递给TableEnvironment。

查看执行计划

SQL语言支持通过explain来查看某条SQL的执行计划，Flink Table API也可以通过调用explain()方法来查看具体的执行计划。该方法返回一个字符串用来描述三个部分计划，分别为：

1. 关系查询的抽象语法树，即未优化的逻辑查询计划，
2. 优化的逻辑查询计划
3. 实际执行计划

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecution
Environment();
StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);
DataStream<Tuple2<Integer, String>> stream1 = env.fromElements(new Tuple
2<>(1, "hello"));
DataStream<Tuple2<Integer, String>> stream2 = env.fromElements(new Tuple
2<>(1, "hello"));
Table table1 = tEnv.fromDataStream(stream1, "count, word");
Table table2 = tEnv.fromDataStream(stream2, "count, word");
Table table = table1
  .where("LIKE(word, 'F%')")
  .unionAll(table2);
// 查看执行计划
String explanation = tEnv.explain(table);
System.out.println(explanation);
```

执行计划的结果为：

```
== 抽象语法树 ==
LogicalUnion(all=[true])
  LogicalFilter(condition=[LIKE($1, _UTF-16LE'F%')])
    FlinkLogicalDataStreamScan(id=[1], fields=[count, word])
  FlinkLogicalDataStreamScan(id=[2], fields=[count, word])

== 优化的逻辑执行计划 ==
DataStreamUnion(all=[true], union all=[count, word])
  DataStreamCalc(select=[count, word], where=[LIKE(word, _UTF-16LE'F%')]
)
    DataStreamScan(id=[1], fields=[count, word])
  DataStreamScan(id=[2], fields=[count, word])

== 物理执行计划 ==
Stage 1 : Data Source
        content : collect elements with CollectionInputFormat

Stage 2 : Data Source
        content : collect elements with CollectionInputFormat

        Stage 3 : Operator
                content : from: (count, word)
                ship_strategy : REBALANCE

                Stage 4 : Operator
                        content : where: (LIKE(word, _UTF-16LE'F%')), se
lect: (count, word)
                        ship_strategy : FORWARD

                        Stage 5 : Operator
                                content : from: (count, word)
                                ship_strategy : REBALANCE
```

小结

本文主要介绍了Flink TableAPI &SQL，首先介绍了Flink Table API &SQL的基本概念，然后介绍了构建Flink Table API & SQL程序所需要的依赖，接着介绍了Flink的两种planner，还介绍了如何注册表以及DataStream、DataSet与表的相互转换，最后介绍了Flink的两种planner对应的查询优化并给出了一个查看执行计划的案例。