Flink DataSet和SQL编程

学习目标：
1. 掌握开发DataSet程序的流程
2. 掌握DataSet输入源的类型和如何设置输入源
3. 掌握DataSet转换操作有哪些
4. 掌握如何调试DataSet程序
5. 掌握Flink Table API和SQL的含义和原理
6. 了解Table执行过程和执行计划
7. 掌握如何将DataStream和DataSet与Table进行相互转换

1 Flink DataSet编程

Flink中的DataSet程序是实现数据集转换的常规程序（例如，过滤，映射，连接，分组）。数据集最初是从某些来源创建的（例如，通过读取文件或从本地集合创建）。结果通过接收器返回，接收器可以例如将数据写入（分布式）文件或标准输出（例如命令行终端）。Flink程序可以在各种环境中运行，独立运行或嵌入其他程序中。执行可以在本地JVM中执行，也可以在许多计算机的集群上执行。

1.1 DataSet示例

通过maven命令创建Flink工程：

```
mvn archetype:generate -DarchetypeGroupId=org.apache.flink -
DarchetypeArtifactId=flink-quickstart-java -DarchetypeVersion=1.9.1
```

我们以单词统计程序为例，演示Flink DataSet程序的开发过程：

```
public class WordCountExample {
    public static void main(String[] args) throws Exception {
        final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

        DataSet<String> text = env.fromElements(
            "Who's there?",
            "I think I hear them. Stand, ho! Who's there?");

        DataSet<Tuple2<String, Integer>> wordCounts = text
            .flatMap(new LineSplitter())
            .groupBy(0)
            .sum(1);

        wordCounts.print();
    }

    public static class LineSplitter implements FlatMapFunction<String,
Tuple2<String, Integer>> {
        @Override
        public void flatMap(String line, Collector<Tuple2<String, Integer>> out)
{
            for (String word : line.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
```

```
            }
        }
    }
}
```

1.2 输入源

数据源创建初始数据集，例如来自文件或Java集合。创建数据集的一般机制是在InputFormat后面抽象的 。Flink附带了几种内置格式，可以从通用文件格式创建数据集。其中许多都在*ExecutionEnvironment*上有快捷方法。

基于文件：

- readTextFile(path)/ TextInputFormat- 按行读取文件并将其作为字符串返回。
- readTextFileWithValue(path)/ TextValueInputFormat- 按行读取文件并将它们作为StringValues返回。StringValues是可变字符串。
- readCsvFile(path)/ CsvInputFormat- 解析逗号（或其他字符）分隔字段的文件。返回元组或POJO的DataSet。支持基本的java类型及其Value对应的字段类型。
- readFileOfPrimitives(path, Class)/ PrimitiveInputFormat- 解析新行（或其他字符序列）分隔的原始数据类型（如String或）的文件Integer。
- readFileOfPrimitives(path, delimiter, Class)/ PrimitiveInputFormat- 解析新行（或其他字符序列）分隔的原始数据类型的文件，例如String或Integer使用给定的分隔符。

基于集合：

- `fromCollection(Collection)` - 从Java.util.Collection创建数据集。集合中的所有元素必须属于同一类型。
- `fromCollection(Iterator, Class)` - 从迭代器创建数据集。该类指定迭代器返回的元素的数据类型。
- `fromElements(T ...)` - 根据给定的对象序列创建数据集。所有对象必须属于同一类型。
- `fromParallelCollection(SplittableIterator, Class)` - 并行地从迭代器创建数据集。该类指定迭代器返回的元素的数据类型。
- `generateSequence(from, to)` - 并行生成给定间隔中的数字序列。

通用：

- `readFile(inputFormat, path)` / `FileInputFormat` - 接受文件输入格式。
- `createInput(inputFormat)` / `InputFormat` - 接受通用输入格式。

例子：

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

// 根据给定的元素创建一个DataSet
DataSet<String> value = env.fromElements("Foo", "bar", "foobar", "fubar");

// 生成一个数字型序列号
DataSet<Long> numbers = env.generateSequence(1, 10000000);

// 从CSV文件中读取三个字段
DataSet<Tuple3<Integer, String, Double>> csvInput =
env.readCsvFile("hdfs:///the/CSV/file")
                            .types(Integer.class, String.class, Double.class);

// 读取CSV文件中的三个字段，并初始化赋值Person对象
DataSet<Person>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
```

```
                        .pojoType(Person.class, "name", "age", "zipcode");


// 从本地系统中读取文本文件
DataSet<String> localLines = env.readTextFile("file:///path/to/my/textfile");

// 从HDFS中读取文件
DataSet<String> hdfsLines =
env.readTextFile("hdfs://nnHost:nnPort/path/to/my/textfile");

// 从HDFS的CSV文件中读取5个字段，使用其中的两个
DataSet<Tuple2<String, Double>> csvInput =
env.readCsvFile("hdfs:///the/CSV/file")
                               .includeFields("10010")  // 使用第一个和第四个
                          .types(String.class, Double.class);

// 顺序读取HDFS文件中的字段
DataSet<Tuple2<IntWritable, Text>> tuples =
 env.createInput(HadoopInputs.readSequenceFile(IntWritable.class, Text.class,
"hdfs://nnHost:nnPort/path/to/file"));

// 通过JDBC读取关系型数据库
DataSet<Tuple2<String, Integer> dbData =
    env.createInput(
      JDBCInputFormat.buildJDBCInputFormat()
                     .setDrivername("org.apache.derby.jdbc.EmbeddedDriver")
                     .setDBUrl("jdbc:derby:memory:persons")
                     .setQuery("select name, age from persons")
                     .setRowTypeInfo(new
RowTypeInfo(BasicTypeInfo.STRING_TYPE_INFO, BasicTypeInfo.INT_TYPE_INFO))
                     .finish()
    );

// 注意：
// Flink的程序编译器需要推断由InputFormat返回的数据项的数据类型，
// 如果此信息无法自动推断，则需要手动提供类型信息。
```

1.3 转换操作

数据转换将一个或多个DataSet转换为新的DataSet。程序可以将多个转换组合成复杂的程序集。

1. **Map**

   采用一个元素并生成一个元素。

   ```
   data.map(new MapFunction<String, Integer>() {
     public Integer map(String value) { return Integer.parseInt(value); }
   });
   ```

2. **FlatMap**

   采用一个元素并生成零个，一个或多个元素。

```
data.flatMap(new FlatMapFunction<String, String>() {
  public void flatMap(String value, Collector<String> out) {
    for (String s : value.split(" ")) {
      out.collect(s);
    }
  }
});
```

3. **MapPartition**

   在单个函数调用中转换并行分区。该函数将分区作为`Iterable`流来获取，并且可以生成任意数量的结果值。每个分区中的元素数量取决于并行度和先前的操作。

```
data.mapPartition(new MapPartitionFunction<String, Long>() {
  public void mapPartition(Iterable<String> values, Collector<Long> out) {
    long c = 0;
    for (String s : values) {
      c++;
    }
    out.collect(c);
  }
});
```

4. **Filter**

   计算每个元素的布尔函数，并保留函数返回true的元素。 **重要信息：**系统假定该函数不会修改元素，否则可能会导致错误的结果。

```
data.filter(new FilterFunction<Integer>() {
  public boolean filter(Integer value) { return value > 1000; }
});
```

5. **Reduce**

   通过将两个元素重复组合成一个元素，将一组元素组合成一个元素。Reduce可以应用于完整数据集或分组数据集。

```
data.reduce(new ReduceFunction<Integer> {
  public Integer reduce(Integer a, Integer b) { return a + b; }
});
```

   如果将reduce应用于分组数据集，则可以通过提供`CombineHintto`来指定运行时执行reduce的组合阶段的方式`setCombineHint`。在大多数情况下，基于散列的策略应该更快，特别是如果不同键的数量与输入元素的数量相比较小（例如1/10）。

6. **ReduceGroup**

   将一组元素组合成一个或多个元素。ReduceGroup可以应用于完整数据集或分组数据集。

```
data.reduceGroup(new GroupReduceFunction<Integer, Integer> {
  public void reduce(Iterable<Integer> values, Collector<Integer> out) {
    int prefixSum = 0;
    for (Integer i : values) {
      prefixSum += i;
      out.collect(prefixSum);
    }
  }
});
```

7. **Aggregate**

   将一组值聚合为单个值。聚合函数可以被认为是内置的reduce函数。聚合可以应用于完整数据集或分组数据集。

```
Dataset<Tuple3<Integer, String, Double>> input = // [...]
DataSet<Tuple3<Integer, String, Double>> output = input.aggregate(SUM,
0).and(MIN, 2);
```

   您还可以使用简写语法进行最小，最大和总和聚合。

```
Dataset<Tuple3<Integer, String, Double>> input = // [...]
DataSet<Tuple3<Integer, String, Double>> output = input.sum(0).andMin(2);
```

8. **Distinct**

   返回数据集的不同元素。它相对于元素的所有字段或字段子集从输入DataSet中删除重复条目。

```
data.distinct();
```

   使用reduce函数实现Distinct。您可以通过提供 `CombineHint` to 来指定运行时执行reduce的组合阶段的方式 `setCombineHint` 。在大多数情况下，基于散列的策略应该更快，特别是如果不同键的数量与输入元素的数量相比较小（例如1/10）。

9. **Join**

   通过创建在其Key上相等的所有元素对来连接两个数据集。可选地使用JoinFunction将元素对转换为单个元素，或使用FlatJoinFunction将元素对转换为任意多个（包括无）元素。

```
result = input1.join(input2)
               .where(0)       // key of the first input (tuple field 0)
               .equalTo(1);    // key of the second input (tuple field 1)
```

   您可以通过 Join Hints 指定运行时执行连接的方式。提示描述了通过分区或广播进行连接，以及它是使用基于排序还是基于散列的算法。

   如果未指定提示，系统将尝试估算输入大小，并根据这些估计选择最佳策略。

```
// This executes a join by broadcasting the first data set
// using a hash table for the broadcast data
result = input1.join(input2,
JoinHint.BROADCAST_HASH_FIRST).where(0).equalTo(1);
```

   请注意，连接转换仅适用于等连接。其他连接类型需要使用OuterJoin或CoGroup表示。

10. **OuterJoin**

在两个数据集上执行左，右或全外连接。外连接类似于常规（内部）连接，并创建在其键上相等的所有元素对。

```
input1.leftOuterJoin(input2) // rightOuterJoin or fullOuterJoin for right or
full outer joins
      .where(0)              // key of the first input (tuple field 0)
      .equalTo(1)            // key of the second input (tuple field 1)
      .with(new JoinFunction<String, String, String>() {
          public String join(String v1, String v2) {
             // NOTE:
             // - v2 might be null for leftOuterJoin
             // - v1 might be null for rightOuterJoin
             // - v1 OR v2 might be null for fullOuterJoin
          }
      });
```

11. **CoGroup**

    reduce操作的二维变体。将一个或多个字段上的每个输入分组，然后加入组。每对组调用转换函数。

```
data1.coGroup(data2)
     .where(0)
     .equalTo(1)
     .with(new CoGroupFunction<String, String, String>() {
         public void coGroup(Iterable<String> in1, Iterable<String> in2,
Collector<String> out) {
           out.collect(...);
         }
     });
```

12. **Cross**

    构建两个输入的笛卡尔积（交叉乘积），创建所有元素对。可选择使用CrossFunction将元素对转换为单个元素

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

    注：交叉是一个潜在的**非常**计算密集型操作它甚至可以挑战大的计算集群！建议使用*crossWithTiny（）*和*crossWithHuge（）*来提示系统的DataSet大小。

13. **Union**

    生成两个数据集的并集。

```
DataSet<String> data1 = // [...]
DataSet<String> data2 = // [...]
DataSet<String> result = data1.union(data2);
```

14. **Rebalance**

    均匀地重新平衡数据集的并行分区以消除数据偏差。只有类似Map的转换可能会使用重新平衡转换。

```
DataSet<String> in = // [...]
DataSet<String> result = in.rebalance()
                           .map(new Mapper());
```

15. **Hash-Partition**

    散列分区给定键上的数据集。键可以指定为位置键，表达键和键选择器功能。

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

16. **Range-Partition**

    范围分区给定键上的数据集。键可以指定为位置键，表达键和键选择器功能。

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

17. **自定义分区**

    使用自定义分区程序功能基于特定分区的键分配记录。密钥可以指定为位置键，表达式键和键选择器功能。 *注意* : 此方法仅适用于单个字段键。

```
DataSet<Tuple2<String,Integer>> in = // [...]
DataSet<Integer> result = in.partitionCustom(partitioner, key)
                            .mapPartition(new PartitionMapper());
```

18. **排序分区**

    本地按指定顺序对指定字段上的数据集的所有分区进行排序。可以将字段指定为元组位置或字段表达式。通过链接sortPartition（）调用来完成对多个字段的排序。

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

1.4 输出源

数据接收器（sink）使用DataSet并用于存储或返回它们。使用OutputFormat描述数据接收器操作 。Flink带有各种内置输出格式，这些格式封装在DataSet上的操作后面：

- `writeAsText()` / `TextOutputFormat` - 按字符串顺序写入元素。通过调用每个元素的*toString（）*方法获得字符串。
- `writeAsFormattedText()` / `TextOutputFormat` - 按字符串顺序编写元素。通过为每个元素调用用户定义的*format（）*方法来获取字符串。
- `writeAsCsv(...)` / `CsvOutputFormat` - 将元组写为逗号分隔值文件。行和字段分隔符是可配置的。每个字段的值来自对象的*toString（）*方法。
- `print()` / `printToErr()` / `print(String msg)` / `printToErr(String msg)` - 在标准输出/标准错误流上打印每个元素的*toString（）*值。可选地，可以提供前缀（msg），其前缀为输出。这有助于区分不同的*打印*调用。如果并行度大于1，则输出也将以生成输出的任务的标识符为前缀。
- `write()` / `FileOutputFormat` - 自定义文件输出的方法和基类。支持自定义对象到字节的转换。

- `output()`/`OutputFormat`- 大多数通用输出方法，用于非基于文件的数据接收器（例如将结果存储在数据库中）。

可以将DataSet输入到多个操作。程序可以编写或打印数据集，同时对它们执行其他转换。

例子

标准数据接收方法：

```
// 文本类型的数据集
DataSet<String> textData = // [...]

// 将数据集保存到本地文件
textData.writeAsText("file:///my/result/on/localFS");

// 将数据集保存到HDFS系统中
textData.writeAsText("hdfs://nnHost:nnPort/my/result/on/localFS");

// 将数据集保存成文件，如果该文件存在，则覆盖该文件
textData.writeAsText("file:///my/result/on/localFS", WriteMode.OVERWRITE);

// 将数据集保存到本地csv文件，数据集各字段用|分割
DataSet<Tuple3<String, Integer, Double>> values = // [...]
values.writeAsCsv("file:///path/to/the/result/file", "\n", "|");

// 按照用户自定义的形式将字符型数据集保存到文本文件
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;
        }
    });
```

使用自定义输出格式：

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

// 将Tuple类型的数据集保存到关系型数据库中
myResult.output(
    // 创建JDBC配置
    JDBCOutputFormat.buildJDBCOutputFormat()
                    .setDrivername("org.apache.derby.jdbc.EmbeddedDriver")
                    .setDBUrl("jdbc:derby:memory:persons")
                    .setQuery("insert into persons (name, age, height) values
(?,?,?)")
                    .finish()
    );
```

1.5 调试程序

在对分布式集群中的大型数据集运行数据分析程序之前，最好确保实现的算法按预期工作。因此，实施数据分析程序通常是检查结果，调试和改进的增量过程。

Flink提供了一些很好的功能，通过支持IDE内的本地调试，测试数据的注入和结果数据的收集，显着简化了数据分析程序的开发过程。本节提供了一些如何简化Flink程序开发的提示。

1.5.1 本地执行环境

A `LocalEnvironment` 在创建它的同一JVM进程中启动Flink系统。如果从IDE启动LocalEnvironment，则可以在代码中设置断点并轻松调试程序。

创建LocalEnvironment并使用如下：

```
final ExecutionEnvironment env = ExecutionEnvironment.createLocalEnvironment();

DataSet<String> lines = env.readTextFile(pathToTextFile);
// build your program

env.execute();
```

1.5.2 收集数据源和接收器

通过创建输入文件和读取输出文件来为分析程序提供输入并检查其输出是麻烦的。Flink具有特殊的数据源和接收器，由Java集合支持以简化测试。一旦程序经过测试，源和接收器可以很容易地被读取/写入外部数据存储（如HDFS）的源和接收器替换。

```
final ExecutionEnvironment env = ExecutionEnvironment.createLocalEnvironment();

// Create a DataSet from a list of elements
DataSet<Integer> myInts = env.fromElements(1, 2, 3, 4, 5);

// Create a DataSet from any Java collection
List<Tuple2<String, Integer>> data = ...
DataSet<Tuple2<String, Integer>> myTuples = env.fromCollection(data);

// Create a DataSet from an Iterator
Iterator<Long> longIt = ...
DataSet<Long> myLongs = env.fromCollection(longIt, Long.class);
```

集合数据接收器指定如下：

```
DataSet<Tuple2<String, Integer>> myResult = ...

List<Tuple2<String, Integer>> outData = new ArrayList<Tuple2<String, Integer>>
();
myResult.output(new LocalCollectionOutputFormat(outData));
```

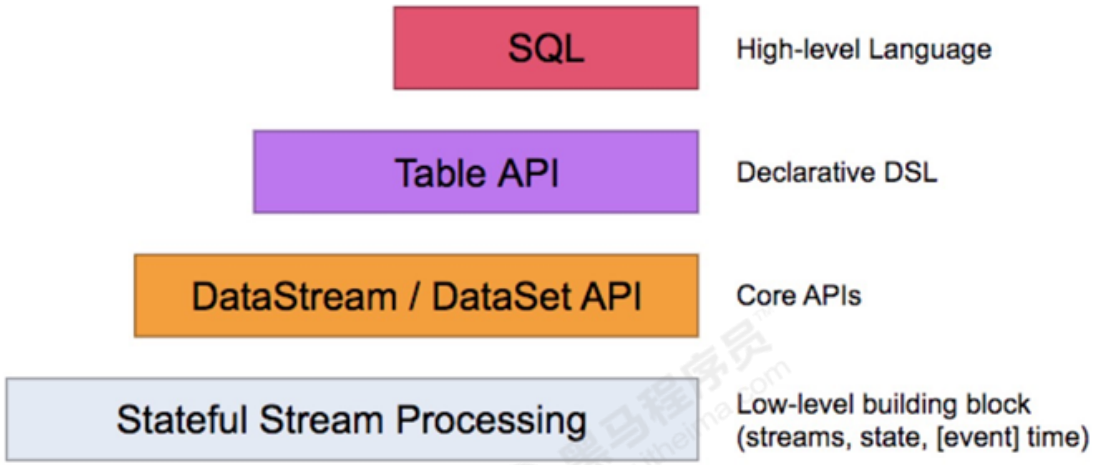
注意：目前，集合数据接收器仅限于本地执行，作为调试工具。

注意：目前，集合数据源要求实现数据类型和迭代器 `Serializable`。此外，收集数据源不能并行执行。

2 Flink Table API和SQL编程

Flink针对标准的流处理和批处理提供了两种相关的API，Table API和sql。Table API允许用户以一种很直观的方式进行select 、filter和join操作。Flink SQL支持基于 Apache Calcite实现的标准SQL。针对批处理和流处理可以提供相同的处理语义和结果。Flink Table API、SQL接口和Flink的DataStream API、DataSet API是紧密联系在一起的。

架构原理：



1577088928687

如果项目中想要使用Table API 和SQL的话，必须要添加下面依赖:

```
<properties>
    <scala.binary.version>2.11</scala.binary.version>
    <!--  其他依赖包的版本... -->
</properties>

<dependencies>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-table-api-java-bridge_${scala.binary.version}
</artifactId>
        <version>${flink.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-table-planner_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
    </dependency>
    <!-- 其他依赖包 -->
</dependencies>
```

2.1 Table API和SQL的程序示例

批处理和流式传输的所有Table API和SQL程序都遵循相同的模式。以下代码示例显示了Table API和SQL程序的常见结构。

批处理和流式传输的 Table API 和SQL程序都遵循相同的模式。以下代码示例显示了常见的程序结构：

```
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.java.StreamTableEnvironment;

import java.util.Arrays;

public class StreamTable {
    /**
```

```
     * 1. 定义数据结构： order pojo
     * 2 设置执行环境
     * 3. 定义table环境
     * 4. 定义数据源/输入源： from collection
     * 5. datastream转化table
     * 6. 执行table查询
     * 7. 执行flink程序
     * @param args
     */
    public static void main(String[] args) throws Exception {
        // 1. 设置执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);

        // 2. 定义输入源
        DataStream<Order> orderA = env.fromCollection(Arrays.asList(
                new Order(1L, "book1", 3),
                new Order(2L, "book2", 4),
                new Order(3L, "book3", 2)
        ));

        DataStream<Order> orderB = env.fromCollection(Arrays.asList(
                new Order(4L, "book4", 3),
                new Order(5L, "book5", 2),
                new Order(6L, "book6", 1)
        ));
        //将数据流转换成flink table
        Table tableA = tEnv.fromDataStream(orderA, "user, product, amount");
        Table tableB = tEnv.fromDataStream(orderB, "user, product, amount");

        //执行查询操作
        Table result = tEnv.sqlQuery("SELECT * FROM " + tableA + " WHERE amount
> 2 union all " +
                "select * from " + tableB + " where amount < 2"); //Flink SQL操作

        //Flink Table API
        Table result2 = tableA.where("amount > 2")
            .unionAll(tableB.where("amount < 2"));


        // 3. 定义输出源
        tEnv.toAppendStream(result, Order.class).print();

        // 4. 执行flink
        env.execute();
    }

   public static class Order{
        public Long user;
        public String product;
        public int amount;

        public Order(){}

        public Order(Long user, String product, int amount){
            this.user = user;
            this.product = product;
```

```
            this.amount = amount;
        }

        @Override
       public String toString(){
            return "Order{" +
                    "user=" + user +
                    ", product=" + product +
                    ", amount=" + amount+
                    "}";
        }
    }
}
```

TableEnvironment 是 Table API 和SQL集成的核心概念，它负责：

- 在内部目录中注册表
- 注册外部目录
- 执行SQL查询
- 注册用户定义的函数
- DataStream 或 DataSet 转换为 Table
- 持有 BatchExecutionEnvironment 或 StreamExecutionEnvironment 的引用

2.2 TableEnvironment

TableEnvironment是Table API和SQL集成的核心概念，是用来创建 Table API 和 SQL 程序的上下文执行环境 ，也是 Table & SQL 程序的入口，Table & SQL 程序的所有功能都是围绕 TableEnvironment 这个核心类展开的。它负责：

- 在内部目录中注册表
- 注册外部目录
- 执行SQL查询
- 注册用户定义的（指标，表或聚合）函数
- 转换： `DataStream` 或 `DataSet` 转换为 `Table`
- 持有对 `ExecutionEnvironment` 或 `StreamExecutionEnvironment` 的引用

在 Flink 1.8 中，一共有 7 个 TableEnvironment ，在最新的 Flink 1.9 中，社区进行了重构和优化，只保留了 5 个TableEnvironment 。在实现上是 5 个面向用户的接口，在接口底层进行了不同的实现。5 个接口包括一个 TableEnvironment 接口，两个 BatchTableEnvironment 接口，两个 StreamTableEnvironment 接口，5 个接口文件完整路径如下：

- org/apache/flink/table/api/TableEnvironment.java
- org/apache/flink/table/api/java/BatchTableEnvironment.java
- org/apache/flink/table/api/scala/BatchTableEnvironment.scala
- org/apache/flink/table/api/java/StreamTableEnvironment.java
- org/apache/flink/table/api/scala/StreamTableEnvironment.scala

其中TableEnvironment 是顶级接口，是所有 TableEnvironment 的基类 ，BatchTableEnvironment 和 StreamTableEnvironment 都提供了 Java 实现和 Scala 实现 ，分别有两个接口。

另一方面，TableEnvironment 作为统一的接口，其统一性体现在两个方面，一是对于所有基于JVM的语言(即 Scala API 和 Java API 之间没有区别)是统一的；二是对于 unbounded data （无界数据，即流数据）和 bounded data （有界数据，即批数据）的处理是统一的。TableEnvironment 提供的是一个纯 Table 生态的上下文环境，适用于整个作业都使用 Table API & SQL 编写程序的场景。

- 两个 StreamTableEnvironment 分别用于 Java 的流计算和 Scala 的流计算场景，流计算的对象分别是 Java 的 DataStream 和 Scala 的 DataStream。相比 TableEnvironment，

StreamTableEnvironment 提供了 DataStream 和 Table 之间相互转换的接口，如果用户的程序除了使用 Table API & SQL 编写外，还需要使用到 DataStream API，则需要使用 StreamTableEnvironment。

- 两个 BatchTableEnvironment 分别用于 Java 的批处理场景和 Scala 的批处理场景，批处理的对象分别是 Java 的 DataSet 和 Scala 的 DataSet。相比 TableEnvironment，BatchTableEnvironment 提供了 DataSet 和 Table 之间相互转换的接口，如果用户的程序除了使用 Table API & SQL 编写外，还需要使用到 DataSet API，则需要使用 BatchTableEnvironment。

```
// 流处理
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.java.StreamTableEnvironment;

StreamExecutionEnvironment sEnv =
StreamExecutionEnvironment.getExecutionEnvironment();
// 创建流查询的TableEnvironment
StreamTableEnvironment sTableEnv = StreamTableEnvironment.create(sEnv);

// 批处理
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.table.api.java.BatchTableEnvironment;

ExecutionEnvironment bEnv = ExecutionEnvironment.getExecutionEnvironment();
// 创建批查询的TableEnvironment
BatchTableEnvironment bTableEnv = BatchTableEnvironment.create(bEnv);
```

2.3. 在表空间中注册表

`TableEnvironment` 是维护按名称注册的表的目录。有两种类型的表，*输入表*和*输出表*。输入表可以在表API和SQL查询中引用，并提供输入参数。输出表可用于将Table API或SQL查询的结果发送到外部系统。

1. 可以从各种来源注册输入表：

- Table API或SQL查询的结果转换成 `Table` 对象。
- `TableSource` ，访问外部数据，例如文件，数据库或消息中间件。
- DataStream或DataSet程序创建的 `DataStream` 或 `DataSet` 。

2. 可以使用 `TableSink` 注册输出表。

2.3.1. 注册表

```
//获得StreamTableEnvironment（BatchTableEnvironment的用法类似）
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

//从表空间中查找表，并执行select操作
Table projTable = tableEnv.scan("tableName").select(...);

// 用一个表注册成另外一个表
tableEnv.registerTable("projectedTable", projTable);
```

2.3.2. 注册TableSource

`TableSource` 提供对外部数据的访问，存储在存储系统中，例如数据库（MySQL，HBase，...），具有特定编码的文件（CSV，Apache [Parquet，Avro，ORC] ……）或消息系统（Apache Kafka，RabbitMQ，……）。

```
// 创建StreamTableEnvironment（BatchTableEnvironment类似）
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

// 创建TableSource
TableSource csvSource = new CsvTableSource("/path/to/file", ...);

// 将TableSource注册成表：CsvTable
tableEnv.registerTableSource("CsvTable", csvSource);
```

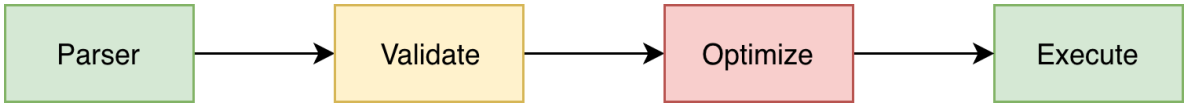
2.3.3. 注册TableSink

已注册 `TableSink` 可用于将表API或SQL查询的结果发送到外部存储系统，例如数据库，键值存储，消息队列或文件系统（在不同的编码中，例如，CSV，Apache [Parquet] ，Avro，ORC]，..... ）。

```
// 创建StreamTableEnvironment（BatchTableEnvironment类似）
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

// 创建TableSink
TableSink csvSink = new CsvTableSink("/path/to/file", ...);

// 定义字段名称和类型
String[] fieldNames = {"a", "b", "c"};
TypeInformation[] fieldTypes = {Types.INT, Types.STRING, Types.LONG};

// 将TableSink注册成表：CsvSinkTable
tableEnv.registerTableSink("CsvSinkTable", fieldNames, fieldTypes, csvSink);
```

2.4 Table执行过程分析

Flink Table API&SQL 利用了Apache Calcite的查询优化框架，为流式数据和批数据的关系查询保留统一的接口。使用Calcite作为SQL解析与处理引擎有Hive、Drill、Flink、Phoenix和Storm等平台。



2.4.1 Flink Sql 执行流程

一条stream sql从提交到calcite解析、优化最后到flink引擎执行，一般分为以下几个阶段:

1. **Sql Parser**: 将sql语句通过java cc解析成AST(语法树)，在calcite中用SqlNode表示AST;
2. **Sql Validator**: 结合数字字典去验证sql语法；
3. **生成Logical Plan**: 将sqlNode表示的AST转换成LogicalPlan, 用relNode表示;
4. **生成 optimized Logical Plan**: 先基于calcite rules 去优化logical Plan, 再基于flink定制的一些优化rules去优化logical Plan；
5. **生成Flink Physica lPlan**: 这里也是基于flink里头的rules，将optimized LogicalPlan转成成Flink的物理执行计划；
6. **将物理执行计划转成Flink Execution Plan**: 就是调用相应的tanslateToPlan方法转换和利用CodeGen元编程成Flink的各种算子。

2.4.2 Flink Table Api 执行流程

如果是通过table api来提交任务的话，也会经过calcite优化等阶段，基本流程和直接运行sql类似:

1. **table api parser**: flink会把table api表达的计算逻辑也表示成一颗树，用treeNode去表式; 在这棵树上的每个节点的计算逻辑用Expression来表示。

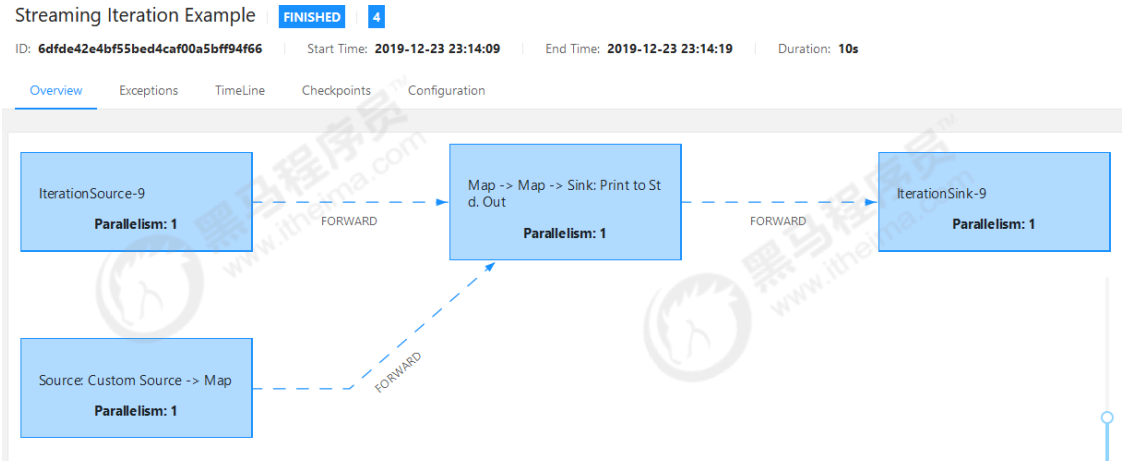
2. **Validate**: 会结合数字字典(catalog)将树的每个节点的Unresolved Expression进行绑定，生成Resolved Expression；
3. **生成Logical Plan**: 依次遍历数的每个节点，调用construct方法将原先用treeNode表达的节点转成成用calcite 内部的数据结构relNode 来表达。即生成了LogicalPlan, 用relNode表示;
4. **生成 optimized Logical Plan**: 先基于calcite rules 去优化logical Plan, 再基于flink定制的一些优化rules去优化logical Plan；
5. **生成Flink Physical Plan**: 这里也是基于flink里头的rules，将optimized LogicalPlan转成成Flink的物理执行计划；
6. **将物理执行计划转成Flink Execution Plan**: 就是调用相应的tanslateToPlan方法转换和利用CodeGen元编程成Flink的各种算子。

2.5 执行计划

Flink会根据客户端提交程序的一些参数，以及集群中机器（TaskManager）的数量去自动优化选取一个它认为合适的执行策略（使数据在DAG中流动计算）；了解flink为job选取的执行计划对我们理解flink是如何执行客户端任务是非常有帮助的。

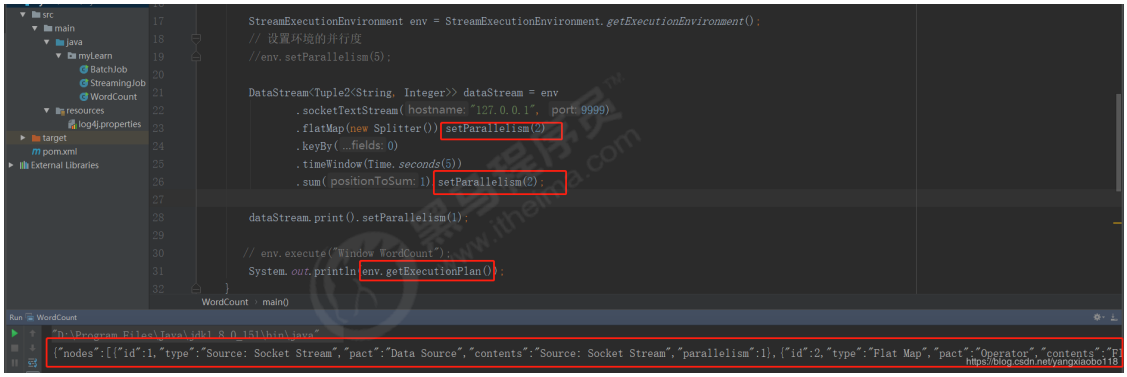
flink提供了最少两种执行计划的可视化的方式，方便我们了解自己编码客户端的执行计划，从而针对性的进行调试。

- **提交Flink工程，在管理页面上查看执行计划**

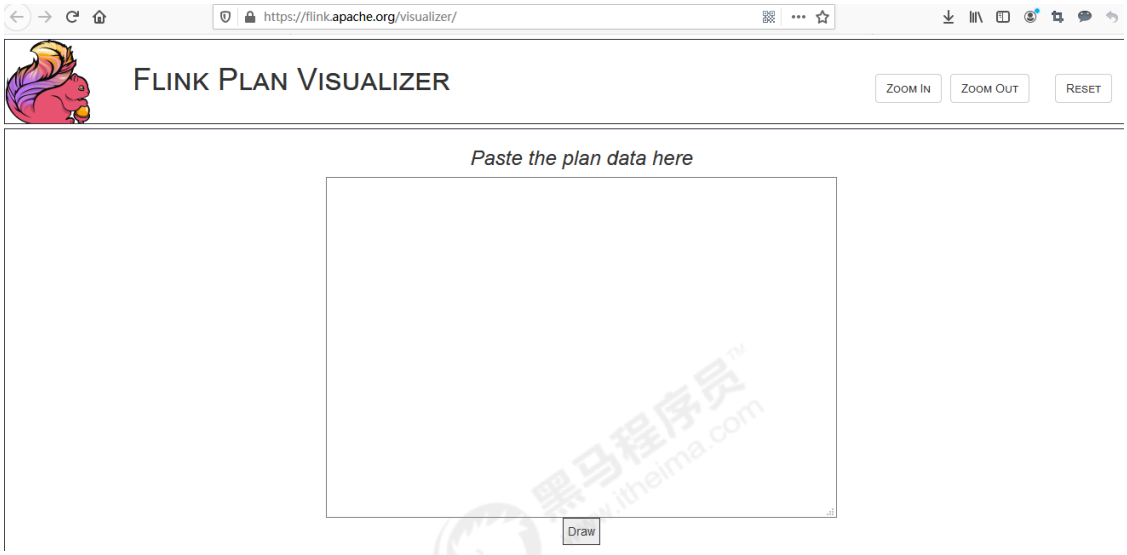


- **Plan Visualization Tool**

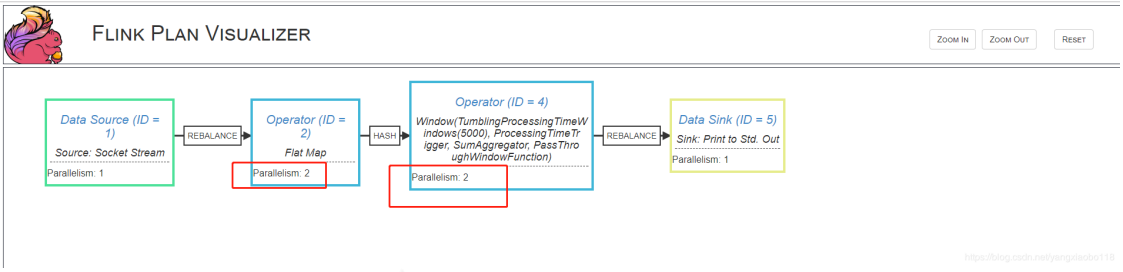
1. 通过flink上下文环境的getExecutionPlan()API输出一段描述执行计划的JSON数据



2. 将输出的信息贴到flink提供的在线可视化工具（https://flink.apache.org/visualizer）
3. 点击Draw效果如下：



4. 点击Draw按钮，效果如下：



2.6 流和批与Table集成编程

表API和SQL查询可以轻松集成并嵌入到DataStream和DataSet程序中。例如，可以查询外部表（例如来自RDBMS），进行一些预处理，例如过滤，投影，聚合或与元数据连接，然后使用DataStream或DataSet API进一步处理数据。 相反，Table API或SQL查询也可以应用于DataStream或DataSet程序的结果。

这种相互作用可以通过转换`DataStream`或`DataSet`转换来实现，`Table`反之亦然。

2.6.1 将DataStream或DataSet注册为表

`DataStream`或`DataSet`可以在`TableEnvironment`表中注册。结果表的模式取决于已注册`DataStream`或的数据类型`DataSet`。

```
// get StreamTableEnvironment
// registration of a DataSet in a BatchTableEnvironment is equivalent
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

DataStream<Tuple2<Long, String>> stream = ...

// 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");
```

注意: 表名：`^_DataStreamTable_[0-9]+`和`^_DataSetTable_[0-9]+`是内部关键字，不能作为我们创建的表名。

2.6.2 将DataStream或DataSet转换为表

除了使用 `TableEnvironment` 注册 `DataStream` 或 `DataSet` ，还可以直接将它们转化成 `Table` 。这个特性在直接使用Table API查询时非常便利。

```
// get StreamTableEnvironment
// registration of a DataSet in a BatchTableEnvironment is equivalent
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

DataStream<Tuple2<Long, String>> stream = ...

// Convert the DataStream into a Table with default fields "f0", "f1"
Table table1 = tableEnv.fromDataStream(stream);

// Convert the DataStream into a Table with fields "myLong", "myString"
Table table2 = tableEnv.fromDataStream(stream, "myLong, myString");
```

2.6.3 将表转换为DataStream或DataSet

`Table` 可以转换为 `DataStream` 或 `DataSet` 。通过这种方式，可以在Table API或SQL查询的结果上运行自定义DataStream或DataSet程序。

当转换一个 `Table` 成 `DataStream` 或 `DataSet` ，需要指定将所得的数据类型 `DataStream` 或 `DataSet` ，即，数据类型到其中的行 `Table` 是要被转换。通常最方便的转换类型是 `Row` 。以下列表概述了不同选项的功能：

- **Row**：字段按位置，任意数量的字段映射，支持 `null` 值，无类型安全访问。
- **POJO**：字段按名称映射（POJO字段必须命名为 `Table` 字段），任意数量的字段，支持 `null` 值，类型安全访问。
- **Case类**：字段按位置映射，不支持 `null` 值，类型安全访问。
- **元组**：字段按位置映射，限制为22（Scala）或25（Java）字段，不支持 `null` 值，类型安全访问。
- **原子类型**： `Table` 必须具有单个字段，不支持 `null` 值，类型安全访问。

2.6.4 将表转换为DataStream

`Table` 流数据查询的结果将动态更新，即它正在改变，因为新记录的查询的输入流到达。因此， `DataStream` 转换这种动态查询需要对表的更新进行编码。

将a转换 `Table` 为a 有两种模式 `DataStream` ：

1. **追加模式**：只有在动态 `Table` 仅通过 `INSERT` 更改修改时才能使用此模式，即，它仅附加，并且以前发出的结果永远不会更新。
2. **缩进模式**：始终可以使用此模式。它用标志编码 `INSERT` 和 `DELETE` 改变 `boolean` 。

```
// get StreamTableEnvironment.
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

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

// convert the Table into an append DataStream of Row by specifying the class
DataStream<Row> dsRow = tableEnv.toAppendStream(table, Row.class);

// convert the Table into an append DataStream of Tuple2<String, Integer>
//   via a TypeInformation
TupleTypeInfo<Tuple2<String, Integer>> tupleType = new TupleTypeInfo<>(
  Types.STRING(),
```

```
  Types.INT());
DataStream<Tuple2<String, Integer>> dsTuple =
  tableEnv.toAppendStream(table, tupleType);

// convert the Table into a retract DataStream of Row.
//   A retract stream of type X is a DataStream<Tuple2<Boolean, X>>.
//   The boolean field indicates the type of the change.
//   True is INSERT, false is DELETE.
DataStream<Tuple2<Boolean, Row>> retractStream =
  tableEnv.toRetractStream(table, Row.class);
```

2.6.5 将表转换为DataSet

```
// get BatchTableEnvironment
BatchTableEnvironment tableEnv = BatchTableEnvironment.create(env);

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

// convert the Table into a DataSet of Row by specifying a class
DataSet<Row> dsRow = tableEnv.toDataSet(table, Row.class);

// convert the Table into a DataSet of Tuple2<String, Integer> via a
TypeInformation
TupleTypeInfo<Tuple2<String, Integer>> tupleType = new TupleTypeInfo<>(
  Types.STRING(),
  Types.INT());
DataSet<Tuple2<String, Integer>> dsTuple =
  tableEnv.toDataSet(table, tupleType);
```

2.7 Mysql数据源

pom文件：

```
<dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-jdbc_2.12</artifactId>
            <version>1.9.1</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.48</version>
        </dependency>
```

程序：

```
import org.apache.flink.api.common.typeinfo.BasicTypeInfo;
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.io.jdbc.JDBCInputFormat;
import org.apache.flink.api.java.operators.DataSource;
import org.apache.flink.api.java.typeutils.RowTypeInfo;
import org.apache.flink.table.api.Table;
```

```
import org.apache.flink.table.api.java.BatchTableEnvironment;
import org.apache.flink.types.Row;


/**
 * 将mysql数据转换成flink table
 */
public class TableJdbc {
    public static void main(String[] args) throws Exception{
        //1. 定义执行环境
        ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
        BatchTableEnvironment batchTableEnvironment =
BatchTableEnvironment.create(env);

        //定义数据类型
        TypeInformation[] fieldTypes = new TypeInformation[]{
                BasicTypeInfo.INT_TYPE_INFO,
                BasicTypeInfo.STRING_TYPE_INFO,
                BasicTypeInfo.INT_TYPE_INFO
        };
        RowTypeInfo rowTypeInfo = new RowTypeInfo(fieldTypes);

        //定义JDBC参数
        JDBCInputFormat jdbcInputFormat = JDBCInputFormat.buildJDBCInputFormat()
                .setDrivername("com.mysql.jdbc.Driver")
                .setDBUrl("jdbc:mysql://ip:3306/test01?characterEncoding=utf8")
                .setUsername("***")
                .setPassword("***")
                .setQuery("select * from users")
                .setRowTypeInfo(rowTypeInfo)
                .finish();

        //定义数据源
        DataSource source = env.createInput(jdbcInputFormat);

        //注册flink table
        batchTableEnvironment.registerDataSet("myTable", source);

        //执行查询
        Table table = batchTableEnvironment.sqlQuery("select * from myTable");

        //返回查询结果
        DataSet result = batchTableEnvironment.toDataSet(table, Row.class);

        //输出
        result.print();

    }
}
```

3 学习总结