

【多易教育】FlinkSql

入门到精通



JUST DO IT
多易教育

1. FlinkSql 快速认识

1.1. 基本原理和架构

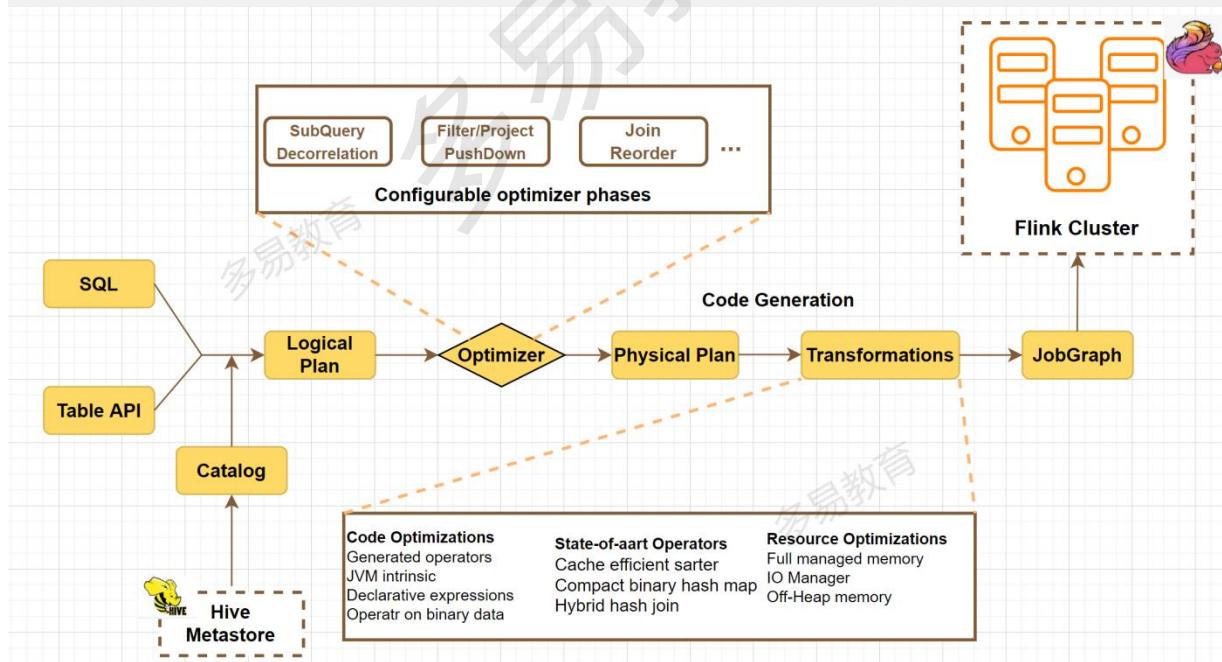
flink sql 是架构于 flink core 之上用 sql 语义方便快捷地进行结构化数据处理的上层库；

(非常类似 sparksql 和 sparkcore 的关系)

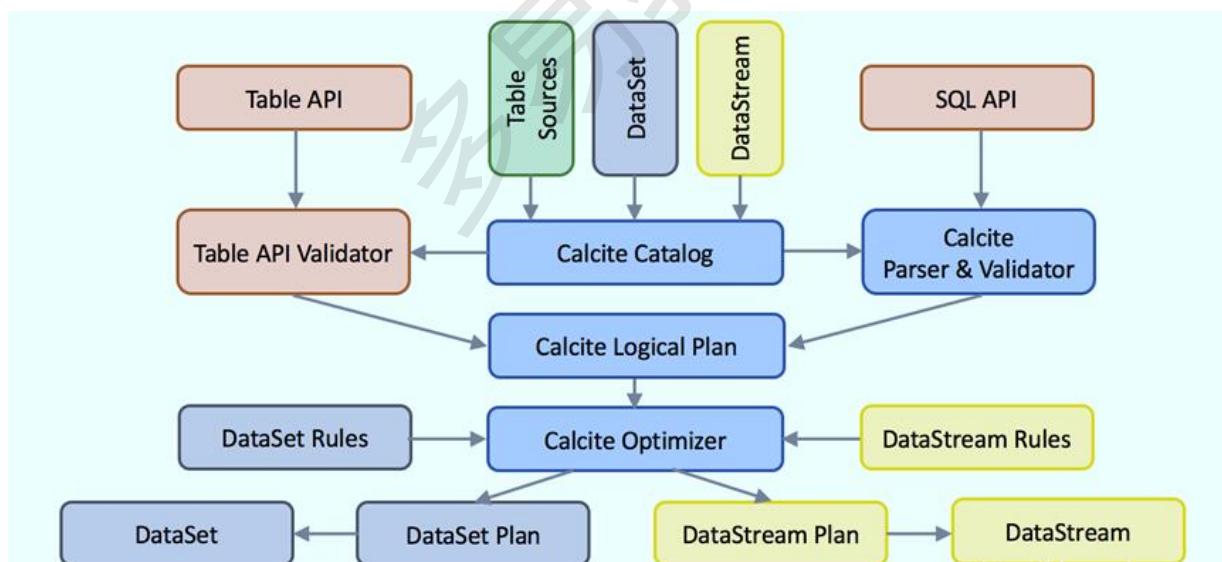
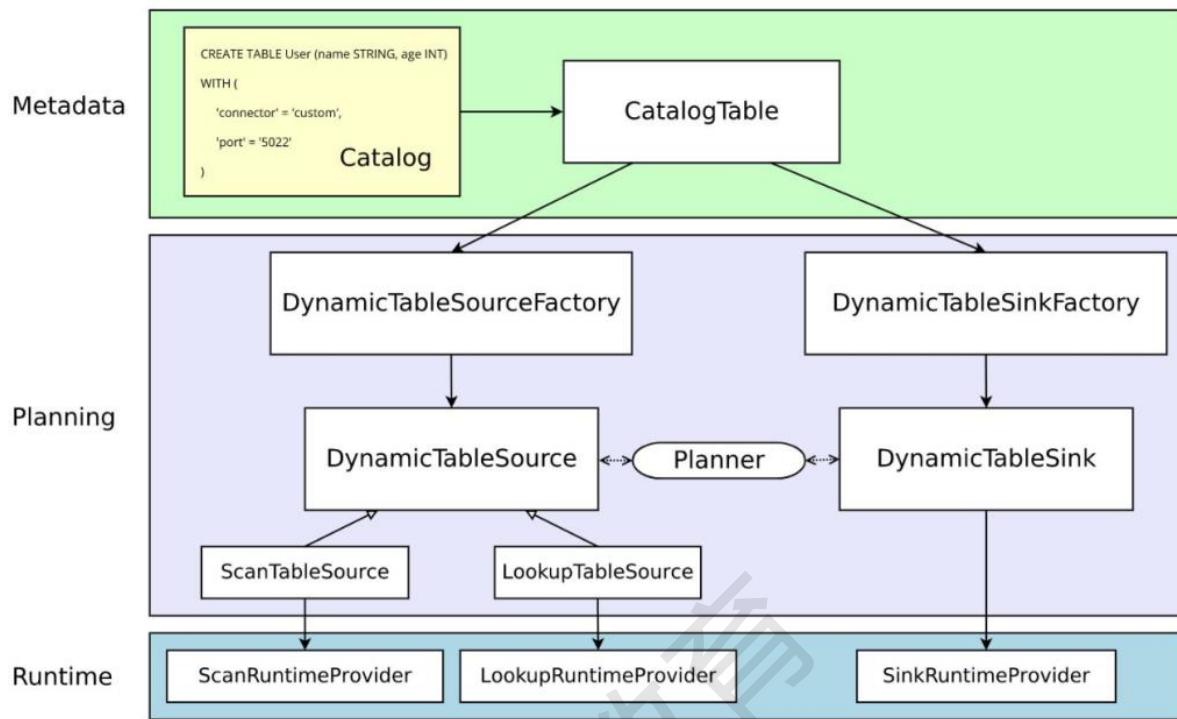
整体架构和工作流程

核心工作原理如下：

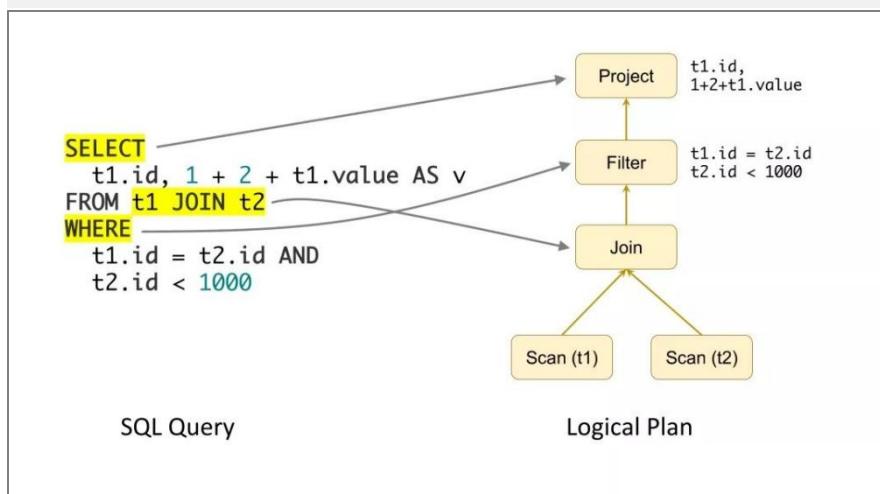
- 将源数据流（数据集），绑定元数据（schema）后，注册成 catalog 中的表（table、view）；
- 然后由用户通过 table Api 或者 table sql 来表达计算逻辑；
- 由 table-planner 利用 apache calcite 进行 sql 语法解析，绑定元数据得到逻辑执行计划；
- 再用 Optimizer 进行优化后，得到物理执行计划
- 物理计划经过代码生成器生成代码，得到 Transformation Tree
- Transformation Tree 转成 JobGraph 后提交到 flink 集群执行



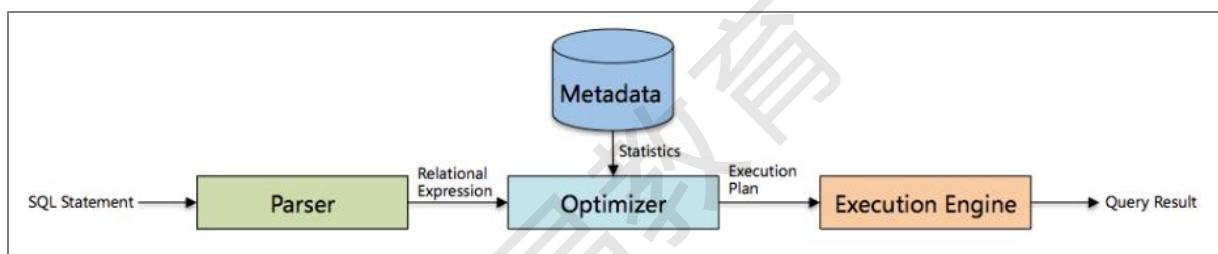
关于元数据管理 catalog



关于逻辑执行计划



关于查询优化



Flinksql 中有两个优化器

- RBO（基于规则的优化器）
- CBO（基于代价（成本）的优化器）

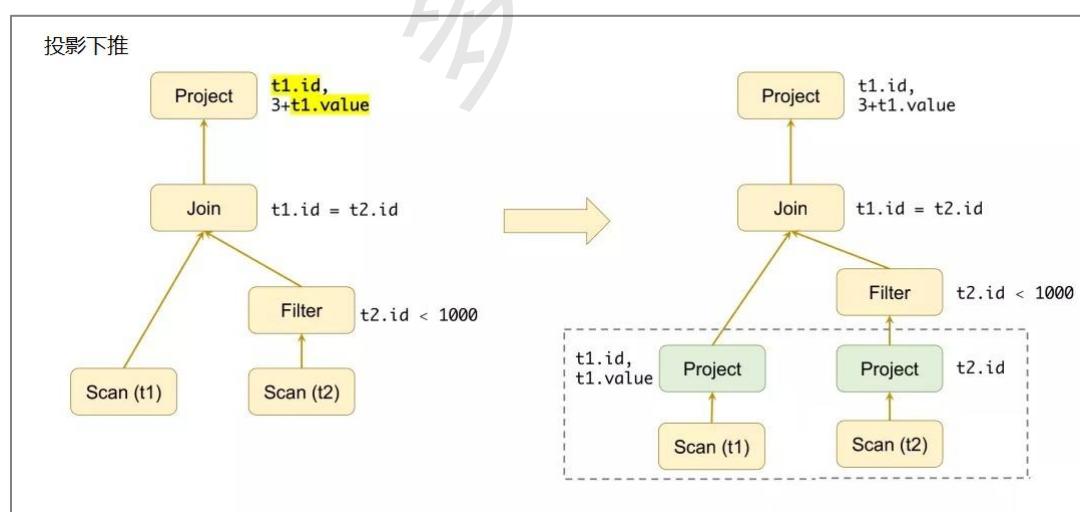
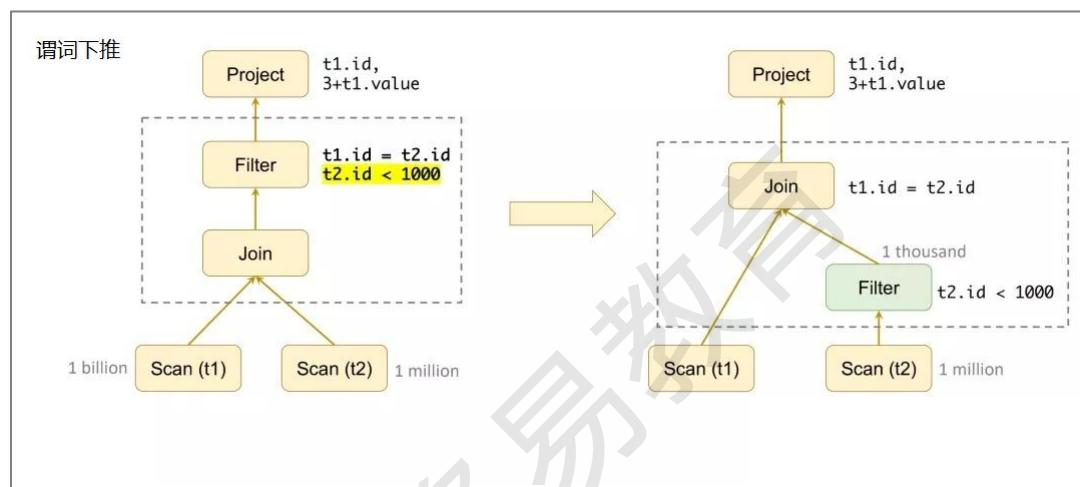
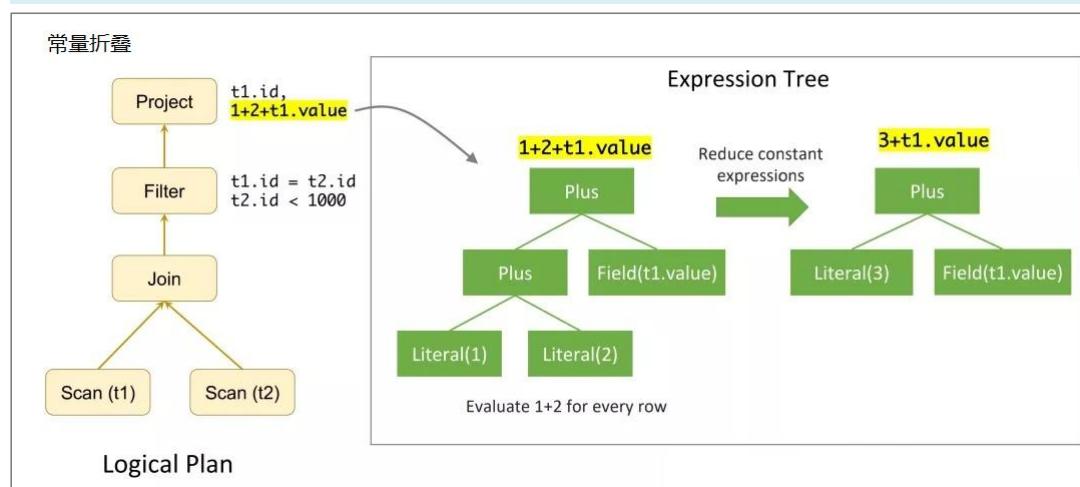
RBO（基于规则的优化器）

遍历一系列规则（[RelOptRule](#)），只要满足条件就对原来的计划节点（表达式）进行转换或调整位置，生成最终的执行计划。

常见的规则包括：

- 分区裁剪（Partition Prune）、列裁剪
- 谓词下推（Predicate Pushdown）、投影下推（Projection Pushdown）、聚合下推、limit 下推、sort 下推
- 常量折叠（Constant Folding）
- 子查询内联转 join 等。

RBO 优化示意图举例



CBO (基于代价的优化器)

会保留原有表达式，基于统计信息和代价模型，尝试探索生成等价关系表达式，最终取代价最小的执行计划。

CBO 的实现有两种模型

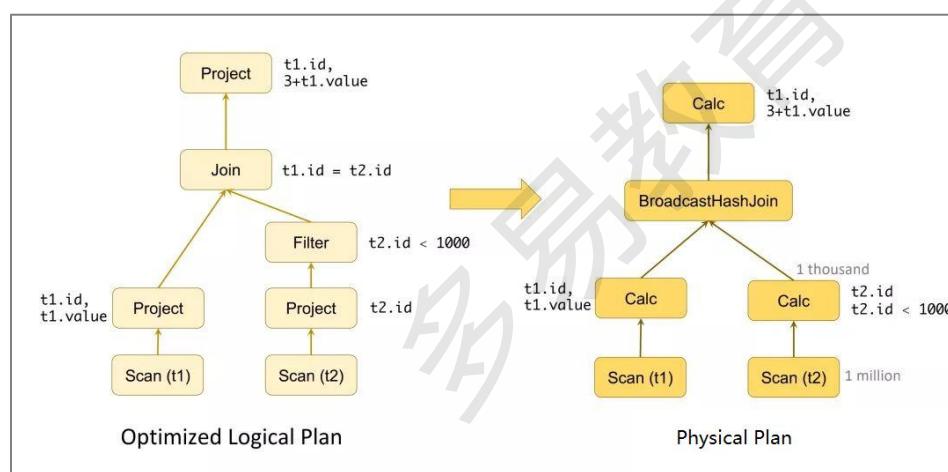
- Volcano 模型
- Cascades 模型

这两种模型思想很是相似，不同点在于 Cascades 模型一边遍历 SQL 逻辑树，一边优化，从而进一步裁剪掉一些执行计划。

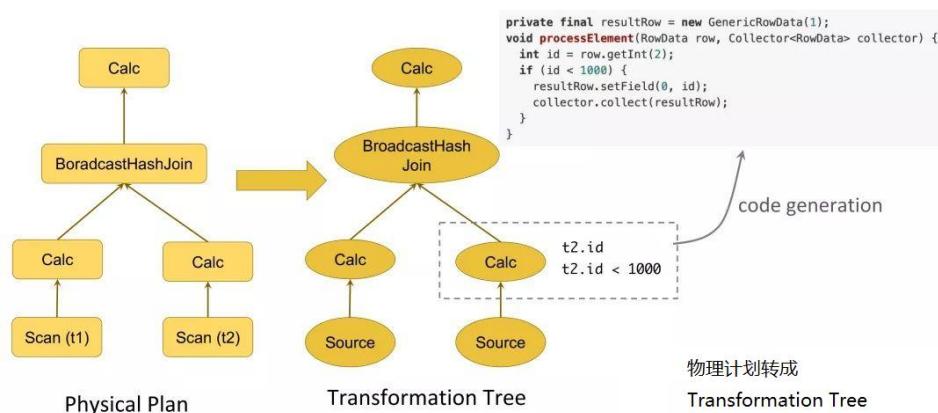
CBO 优化示意图举例

根据代价 cost 选择批处理 join 有方式(sortmergejoin, hashjoin, broadcasthashjoin)。

比如前文中的例子，再 filter 下推之后，在 $t2.id < 1000$ 的情况下，由 1 百万数据量变为了 1 千条，计算 cost 之后，使用 broadcasthashjoin 最合适。



物理计划=>Transformation Tree



1.2. 动态表特性

与 spark、hive 等组件中的“表”的最大不同之处：flinksql 中的表是动态表！

这是因为：

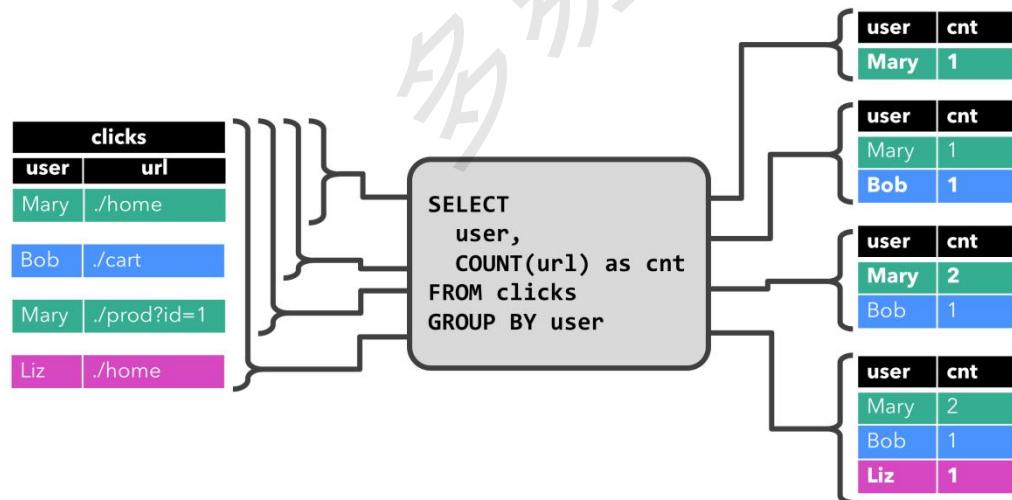
- flink 对数据的核心抽象是“无界（或有界）的数据流”
- 对数据处理过程的核心抽象是“流式持续处理”

因而，flinksql 对“源表（动态表）”的计算及输出结果（结果表），也是流式、动态、持续的；

- 数据源的数据是持续输入
- 查询过程是持续计算
- 查询结果是持续输出

如下图所示：

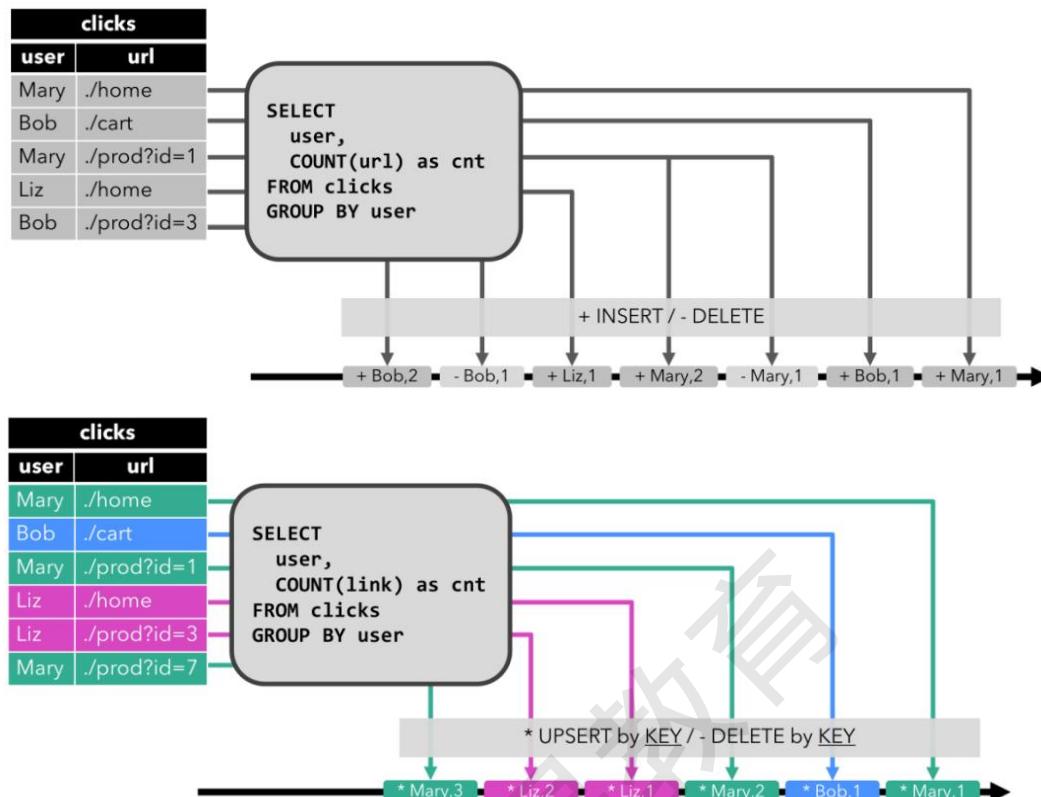
- “源表 clicks”是流式动态的；
- “聚合查询的输出结果表”，也是流式动态的



这其中的动态，不仅体现在“数据追加”，对于输出结果表来说，“动态”还包含对“前序输出结果”的“撤回（删除）”、“更新”等模式；

而 flinksql 如何将这种对于“前序输出的修正”表达给下游呢？

它的核心设计是在底层的数据流中为每条数据添加“ChangelogMode（修正模式）标记”，而添加了这种 Changelog 标记的底层数据流，取名为 changelogStream



在 flink1.12 之前，动态表所对应的底层 stream，有 3 种：

- Append-only stream
- Retract stream
- Upsert stream

现在，统称为 changelog stream

tenv.to		
toDataStream(Table table, Class<T> targetClass)		DataStream<T>
toDataStream(Table table)		DataStream<Row>
toDataStream(Table table, AbstractDataType<?> targetType)		DataStream<T>
toChangelogStream(Table table)		DataStream<Row>
toChangelogStream(Table table, Schema targetSchema)		DataStream<Row>
toChangelogStream(Table table, Schema targetSchema, ChangelogMode changelogMode)		DataStream<Row>
toAppendStream(Table table, Class<T> clazz)		DataStream<T>
toAppendStream(Table table, TypeInformation<T> typeInfo)		DataStream<T>
toRetractStream(Table table, Class<T> clazz)		DataStream<Tuple2<Boolean, T>>
toRetractStream(Table table, TypeInformation<T> typeInfo)		DataStream<Tuple2<Boolean, T>>

2. FlinkSql 编程概览

2.1. FlinkSql 程序结构

所需依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-api-java-bridge_2.12</artifactId>
    <version>${flink.version}</version>
</dependency>
```

Flinksql 编程 4 步曲

- 1) 创建 flinksql 编程入口
- 2) 将数据源定义（映射）成表（视图）
- 3) 执行 sql 语义的查询（sql 语法或者 tableapi）
- 4) 将查询结果输出到目标表

```
import org.apache.flink.table.api.*;
import org.apache.flink.connector.datagen.table.DataGenOptions;

// 创建 table 编程入口环境.
TableEnvironment tableEnv = TableEnvironment.create(/* ... */);

// 创建一个源表
tableEnv.createTemporaryTable("SourceTable", TableDescriptor.forConnector("kafka")
    .schema(Schema.newBuilder()
        .column("f0", DataTypes.STRING())
        .build())
    .option(DataGenOptions.ROWS_PER_SECOND, 100)
    .build());

// 创建一个目标表
tableEnv.executeSql("CREATE TEMPORARY TABLE SinkTable WITH ('connector' = 'kafka') LIKE SourceTable");
// 执行查询并输出结果
tableEnv.executeSql("INSERT INTO SinkTable SELECT * FROM SourceTable");
```

2.2. Table Environment

flinksql 的编程，总是从一个入口环境 **TableEnvironment** 开始；

TableEnvironment 的主要功能如下：

- 注册 catalogs
 - 向 catalog 注册表
 - 加载可插拔模块（目前有 hive module，以用于扩展支持 hive 的语法、函数等）
 - 执行 sql 查询（sql 解析，查询计划生成，job 提交）
 - 注册用户自定义函数
 - 提供 datastream 和 table 之间的互转
- ✓ 创建方式 1（直接创建 TableEnvironment）

```
EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()
    //inBatchMode()
    .build();
```

```
TableEnvironment tEnv = TableEnvironment.create(settings);
```

- ✓ 创建方式 2（从 StreamExecutionEnvironment 创建，这样便于结合 sql 和 stream 编程）

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);
```

2.3. 两种编程方式

Flinksql 提供两种编程方式：

- Table Api
- Table Sql

类比 sparksql 的 dataframe api 编程和 sql 编程，即可快速理解；

2.3.1. [Table Api] 方式

- Table API 是一个与编程语言（scala、java、python）集成的查询 API；
与 SQL 不同，查询逻辑不是以字符串表达，而是在“宿主语言”中调用所提供的类、方法等。
- 复杂的运算可以通过调用多个方法组成；
如： table.filter(...).groupBy(...).select(...).join(...).on(...)

TableAPI 代码示例

```
// 创建入口
TableEnvironment tableEnv = ...;

// 注册 orders 表
// 注册结果表 sinkTable

// 利用 env.from 方法，得到 Table 对象
Table orders = tableEnv.from("Orders");
Table sinkTable = tableEnv.from("sinkTable");

// 执行查询： "中国区"每个客户的总收入
Table revenue = orders
    .filter($"cCountry").isEqual("China")
    .groupBy($"cID"), $"cName")
    .select($"cID"), $"cName", $"revenue")
    .sum().as("revSum");

// 计算结果插入目标表
revenue.insertInto(sinkTable);
```

上述 tableapi 的计算逻辑，就等价于如下 sql 语句：

```
insert into sinkTable
select
    cID,
    cName,
    sum(revenue) as revSum
from orders
where cCountry = 'China'
group by cID,cName
```

2.3.2. [Table Sql] 方式

用“sql 字符串”形式进行基于表的关系运算逻辑表达

```
// 创建入口
TableEnvironment tableEnv = ...;

// 注册 orders 源表
tableEnv.createTable("orders",....)
// 注册 RevenueChina 输出表
tableEnv.createTable("RevenueChina",....)

// 计算中国区每个客户的总收入，并输出到结果表
tableEnv.executeSql(
    "INSERT INTO RevenueChina " +
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = CHINA" +
    "GROUP BY cID, cName"
);
```

2.3.3. 混搭方式

表 API 和 SQL 查询可以很容易地混合，因为 Table 对象可以和 sql 表进行方便的互转：

- 可以让 SQL 查询返回 Table 对象，进而调用 TableAPI；
- 可以用 env.from("sql 表名") 引用 sql 表得到 Table 对象，进而调用 TableAPI；
- 可以用 env.createTemporaryView("sql 表名", table 对象)，将 Table 对象注册成 sql 表，进而用 sql

具体详情参见后续章节中的《表定义详解》；

3. 完整入门示例

需求背景

- 有如下数据通过 socket 端口输入

```
{"guid":1,"sessionId":"s01","eventId":"e01","eventTime":1000}  
{"guid":1,"sessionId":"s01","eventId":"e02","eventTime":2000}  
{"guid":1,"sessionId":"s01","eventId":"e03","eventTime":3000}  
{"guid":2,"sessionId":"s02","eventId":"e02","eventTime":2000}  
{"guid":2,"sessionId":"s02","eventId":"e02","eventTime":3000}  
{"guid":2,"sessionId":"s02","eventId":"e01","eventTime":4000}  
{"guid":2,"sessionId":"s02","eventId":"e03","eventTime":5000}
```

● 需求

每个用户每次会话中各类行为的发生次数，并将结果输出到控制台

3.1. 获取数据流

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
StreamTableEnvironment tenv = StreamTableEnvironment.create(env);  
  
DataStreamSource<String> source = env.socketTextStream("doitedu", 9999);  
DataStream<EventBean> stream = source.map(s -> JSON.parseObject(s, EventBean.class));
```

3.2. [Table Api] 方式查询

```
// 将 dataStream 注册成临时视图  
tenv.createTemporaryView("t_event", stream);  
  
// 查询每个用户每次会话中每类事件的发生次数  
tenv.executeSql("select guid,sessionId,eventId,count(1) from t_event group by guid,sessionId,eventId").print();
```

3.3. [Table Sql] 方式查询

```
// 从 dataStream 创建 Table 对象  
Table table = tenv.fromDataStream(stream);  
// 查询每个用户每次会话中每类事件的发生次数  
Table select =  
    table.groupBy($"guid"), $"sessionId"), $"eventId")  
        .aggregate($"eventId".count().as("event_cnt"))
```

```
.select("guid"), $("sessionId"), $("eventId"), $("event_cnt"));  
select.execute().print();
```



4. 表的概念及类别

4.1. 表的标识结构

每一个表的标识由 3 部分组成：

- ✓ catalog name (常用于标识不同的“源”，比如 hive catalog, inner catalog 等)
- ✓ database name (通常语义中的“库”)
- ✓ table name (通常语义中的“表”)

```
TableEnvironment tEnv = ...;
tEnv.useCatalog("a_catalog");
tEnv.useDatabase("db1");

Table table = ...;

// 注册在默认 catalog 的默认 database 中
tableEnv.createTemporaryView("a_view", table);

// 注册在默认 catalog 的指定 database 中
tableEnv.createTemporaryView("db2.a_view", table);

// 注册在指定 catalog 的指定 database 中
tableEnv.createTemporaryView("x_catalog.db3.a_view", table);
```

一个 flinksql 程序在运行时，tableEnvironment 通过持有一个 map 结构来记录所注册的 catalog；

```
public final class CatalogManager {
    private static final Logger LOG = LoggerFactory.getLogger(CatalogManager.class);
    private final Map<String, Catalog> catalogs;
    private final Map<ObjectIdentifier, CatalogBaseTable> temporaryTables;
```

4.2. 表与视图

FlinkSQL 中的表，可以是 virtual 的（view 视图）和 regular 的（table 常规表）

- table 描述了一个物理上的外部数据源，如文件、数据库表、kafka 消息 topic
- view 则基于表创建，代表一个或多个表上的一段计算逻辑（就是对一段查询计划的逻辑封装）；

不管是 table 还是 view，在 tableAPI 中得到的都是 Table 对象

4.3. 临时与永久

- 临时表（视图）： 创建时带 temporary 关键字（create temporary view, createtemporary table）
- 永久表（视图）： 创建时不带 temporary 关键字（create view , create table ）

```
// sql 定义方式
tableEnv.executeSql("create view view_1 as select .. from projectedTable")
tableEnv.executeSql("create temporary view  view_2 as select .. from projectedTable")

tableEnv.executeSql("create table (id int,...) with ('connector'= ...)")
tableEnv.executeSql("create temporary table (id int,...) with ('connector'= ...)")

// tableapi 方式
tenv.createTable("t_1",tableDescriptor);
tenv.createTemporaryTable("t_1",tableDescriptor);

tenv.createTemporaryView("v_1",dataStream,schema);
tenv.createTemporaryView("v_1",table);
```

临时表与永久表的本质区别： schema 信息是否被持久化存储

- 临时表（视图）

表 schema 只维护在所属 flink session 运行时内存中；

当所属的 flink session 结束后表信息将不复存在；且该表无法在 flink session 间共享；

- 常规表（视图）

表 schema 可记录在外部持久化的元数据管理器中（比如 hive 的 metastore）；

当所属 flink session 结束后，该表信息不会丢失；且在不同 flink session 中都可访问到该表的信息；

5. 表定义概览

5.1. [Table Api] Table 创建概览

```
tenv.from
  m from(String path) Table
  m from(TableDescriptor descriptor) Table
  m fromChangelogStream(DataStream<Row> dataStream) Table
  m fromChangelogStream(DataStream<Row> dataStream, Schema schema) Table
  m fromChangelogStream(DataStream<Row> dataStream, Schema schema, ChangelogMode changelogMode) Table
  m fromDataStream(DataStream<T> dataStream) Table
  m fromDataStream(DataStream<T> dataStream, Schema schema) Table
  m fromDataStream(DataStream<T> dataStream, String fields) Table
  m fromDataStream(DataStream<T> dataStream, Expression... fields) Table
  m fromTableSource(TableSource<?> source) Table
  m fromValues(Object... values) Table
  m fromValues(Iterable<?> values) Table
  m fromValues(Expression... values) Table
  m fromValues(AbstractDataType<?> rowType, Object... values) Table
  m fromValues(AbstractDataType<?> rowType, Iterable<?> values) Table
  m fromValues(AbstractDataType<?> rowType, Expression... values) Table
```

Ctrl+向下箭头 和 Ctrl+向上箭头 将在编辑器中向下和向上移动文本光标 [下一提示](#)

```
Table table3 = table2
    .select($( name: "id"), $( name: "name"), $( name: "score")) Table
    .where($( name: "name").like(pattern: "Mr%"))
    .groupBy($( name: "name")) GroupedTable
    .select($( name: "name"), $( name: "score").avg());
```

Table 对象获取方式解析：

- 从已注册的表
- 从 TableDescriptor (连接器/format/schema/options)
- 从 DataStream
- 从 Table 对象上的查询 api 生成
- 从测试数据

涉及的核心参数：

- 已注册的表名 (catalog_name.database_name.object_name)
- TableDescriptor (表描述器, 核心是 connector 连接器)
- Datastream (底层流)
- 测试数据值

5.2. [Table Api] Table 创建示例

通过已注册的表名生成 Table 对象

```
Table t1 = tenv.from("t1"); // 通过已经在 env 的 catalog 中注册的表名，获得 Table 对象
```

通过 DataStream 生成 Table 对象

- 自动推断 schema (反射手段)

```
DataBean bean1 = new DataBean(1, "s1", "e1", "pg1", 1000);
DataBean bean2 = new DataBean(2, "s2", "e3", "pg1", 1000);
DataStreamSource<DataBean> dataStream1 = env.fromElements(bean1, bean2);
Table table1 = tenv.fromDataStream(dataStream1);
```

- 手动定义 schema

```
Table table = tenv.fromDataStream(dataStream2, Schema.newBuilder()
    .column("f0", DataTypes.STRUCTURED(
        DataBean2.class,
        DataTypes.FIELD("guid", DataTypes.INT()),
        DataTypes.FIELD("uuid", DataTypes.STRING()),
        DataTypes.FIELD("eventId", DataTypes.STRING()),
        DataTypes.FIELD("pageId", DataTypes.STRING()),
        DataTypes.FIELD("ts", DataTypes.BIGINT())
    )).build());
```

通过 tableEnv 的 fromValues 方法获得 Table 对象 (快速测试用)

```
Table table = tenv.fromValues(
    DataTypes.ROW(
        DataTypes.FIELD("id", DataTypes.INT()),
        DataTypes.FIELD("name", DataTypes.STRING()),
        DataTypes.FIELD("info", DataTypes.MAP(DataTypes.STRING(), DataTypes.STRING())),
        DataTypes.FIELD("ts1", DataTypes.TIMESTAMP(3)),
        DataTypes.FIELD("ts3", DataTypes.TIMESTAMP_LTZ(3)),
        /*DataTypes.FIELD("ts5", DataTypes.TIMESTAMP_WITH_TIME_ZONE(3)),*/
    ),
    Row.of(1, "a", info, "2022-06-03 13:59:20.200", 1654236105000L, "14:13:00.200")
);
```

通过 Table 上调用查询 api，生成新的 Table 对象 (本质上就是 view)

```
Table table = table3.select($"guid"), $"uuid");
```

5.3. [Table Sql] 表创建概览

```
tenv.create
  (m) createTemporaryView(String path, DataStream<T> dataStream)           void
  (m) createTemporaryView(String path, Table view)                           void
  (m) createTemporaryView(String path, DataStream<T> dataStream, Schema schema) void
  (m) createTemporaryView(String path, DataStream<T> dataStream, String fields)   void
  (m) createTemporaryView(String path, DataStream<T> dataStream, Expression... fields) void
  (m) createTable(String path, TableDescriptor descriptor)                      void
  (m) createTemporaryTable(String path, TableDescriptor descriptor)            void
  Ctrl+向下箭头 和 Ctrl+向上箭头 将在编辑器中向下和向上移动文本光标 下一提示          :
```

```
tenv.executeSql( statement: "create table t_name(....) with ('connector'='....')");
tenv.executeSql( statement: "create temporary table t_name(....) with ('connector'='....')");
tenv.executeSql( statement: "create temporary view t_name(....) with ('connector'='....')");
tenv.executeSql( statement: "create temporary view as select ... from t_source");
```

注册 sql 表（视图）方式解析

- 从已存在的 datastream 注册
- 从已存在的 Table 对象注册
- 从 TableDescriptor（连接器）注册
- 执行 Sql 的 DDL 语句来注册

5.4. [Table Sql] 表创建示例

将已存在的 Table 对象注册成 sql 视图

```
tenv.createTemporaryView("t1",table);
```

将 datastream 注册成 sql 视图

```
DataBean bean1 = new DataBean(1, "s1", "e1", "pg1", 1000);
DataBean bean2 = new DataBean(1, "s1", "e1", "pg1", 1000);
DataStreamSource<DataBean> dataStream1 = env.fromElements(bean1, bean2);
```

```
// 1.自动推断 schema
tenv.createTemporaryView("t1",dataStream1);

// 2.也可以手动指定 schema
Schema schema = Schema.Builder.column...build();
tenv.createTemporaryView("t1",dataStream1,schema);

tenv.executeSql("desc t1");
tenv.executeSql("select * from t1");
```

通过 connector 注册 sql 表

```
tenv.createTable("t1",TableDescriptor.forConnector("filesystem")
    .option("path", "file://d:/a.txt")
    .format("csv")
    .schema(Schema.newBuilder()
        .column("guid",DataTypes.STRING())
        .column("name",DataTypes.STRING())
        .column("age",DataTypes.STRING())
        .build())
    .build());
```

通过 sql DDL 语句定义 sql 表

```
tenv.executeSql(
    "CREATE TABLE age_info
    + "("
    + "    id      INT,
    + "    name    string,
    + "    gender  string,
    + "    age     int,
    + "    PRIMARY KEY (id) NOT ENFORCED   "
    + ") WITH (
    + "    'connector' = 'mysql-cdc',
    + "    'hostname' = 'doit01',
    + "    'port' = '3306',
    + "    'username' = 'root',
    + "    'password' = 'ABC123.abc123',
    + "    'database-name' = 'abc',
    + "    'table-name' = 'age_info'
    + ")");
tenv.executeSql("select * from age_info").print();
```

6. catalog 详解

6.1. 什么是 catalog

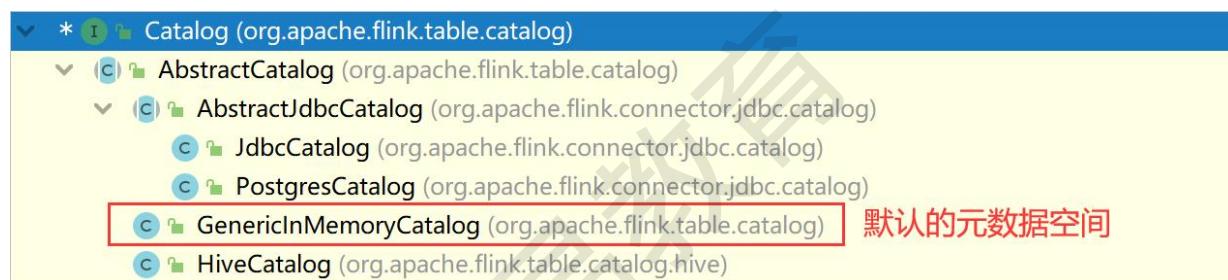
一句话： catalog 就是一个元数据空间，简单说就是记录、获取元数据（表定义信息）的实体；

flinksql 在运行时，可以拥有多个 catalog，它们由 catalogManager 模块来注册、管理；

CatalogManager 中可以注册多个元数据空间：

1 环境创建之初，就会初始化一个默认的元数据空间

- 空间名称： default_catalog
- 空间实现类： GenericInMemoryCatalog



```
public class GenericInMemoryCatalog extends AbstractCatalog {

    public static final String DEFAULT_DB = "default";
    // 用于记录本 catalog 空间中所有 database 的 linkedHashMap
    private final Map<String, CatalogDatabase> databases;
    // 用于记录本 catalog 空间中所有 table 的 linkedHashMap
    private final Map<ObjectPath, CatalogBaseTable> tables;
```

```
private CatalogManager(
    String defaultCatalogName, Catalog defaultCatalog, DataTypeFactory typeFactory) {
    checkArgument(
        !StringUtils.isNullOrWhitespaceOnly(defaultCatalogName),
        "Default catalog name cannot be null or empty");
    checkNotNull(defaultCatalog, "Default catalog cannot be null");
    // 用于记录 session 中所有的 catalog
    catalogs = new LinkedHashMap<>();
    // 放入一个 defaultCatalog
    catalogs.put(defaultCatalogName, defaultCatalog);
```

```
// 设置当前的 catalog 为 default_catalog
currentCatalogName = defaultCatalogName;

// 设置当前的 database 为 default_database
currentDatabaseName = defaultCatalog.getDefaultDatabase();

// 构造一个空的 hashmap, 用来记录 session 中注册的临时表
temporaryTables = new HashMap<>();

// right now the default catalog is always the built-in one
builtInCatalogName = defaultCatalogName;

this.typeFactory = typeFactory;

}
```

2 用户还可以向环境中注册更多的 catalog, 如下代码新增注册了一个 hivecatalog

```
// 创建了一个 hive 元数据空间的实现对象
HiveCatalog hiveCatalog = new HiveCatalog("hive", "default", "/hiveconf/dir");

// 将 hive 元数据空间对象注册到 环境中
tenv.registerCatalog("mycatalog",hiveCatalog);
```

6.2. 深入测试 catalog

通过源码和测试代码, 深入理解 catalog

```
package cn.doitedu.flinksql.demos;

public class Demo5_CatalogDemo {

    public static void main(String[] args) {

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 环境创建之初, 底层会自动初始化一个 元数据空间实现对象 (default_catalog => GenericInMemoryCatalog)
        StreamTableEnvironment tenv = StreamTableEnvironment.create(env);

        // 创建了一个 hive 元数据空间的实现对象
        HiveCatalog hiveCatalog = new HiveCatalog("hive", "default", "d:/conf/hiveconf");
        // 将 hive 元数据空间对象注册到 环境中
        tenv.registerCatalog("mycatalog",hiveCatalog);

        tenv.executeSql(
```

```
"create temporary table `mycatalog`.`default`.`t_kafka`      "
+ "(
+ "  id int,
+ "  name string,
+ "  age int,
+ "  gender string
+ ")
+ " WITH (
+ "  'connector' = 'kafka',
+ "  'topic' = 'doit30-3',
+ "  'properties.bootstrap.servers' = 'doitedu:9092',
+ "  'properties.group.id' = 'g1',
+ "  'scan.startup.mode' = 'earliest-offset',
+ "  'format' = 'json',
+ "  'json.fail-on-missing-field' = 'false',
+ "  'json.ignore-parse-errors' = 'true'
+ ")
);
tenv.executeSql(
"create temporary table `t_kafka2`      "
+ "(
+ "  id int,
+ "  name string,
+ "  age int,
+ "  gender string
+ ")
+ " WITH (
+ "  'connector' = 'kafka',
+ "  'topic' = 'doit30-3',
+ "  'properties.bootstrap.servers' = 'doitedu:9092',
+ "  'properties.group.id' = 'g1',
+ "  'scan.startup.mode' = 'earliest-offset',
+ "  'format' = 'json',
+ "  'json.fail-on-missing-field' = 'false',
+ "  'json.ignore-parse-errors' = 'true'
+ ")
);
tenv.executeSql("create view if not exists `mycatalog`.`default`.`t_kafka_view` as select id,name,age from
`mycatalog`.`default`.`t_kafka`");
```

```

tenv.listCatalogs();

tenv.executeSql("show catalogs").print();
tenv.executeSql("use catalog default_catalog");
tenv.executeSql("show databases").print();
tenv.executeSql("use default_database");
tenv.executeSql("show tables").print();

System.out.println("-----");

tenv.executeSql("use catalog mycatalog");
tenv.executeSql("show databases").print();
tenv.executeSql("use `default`");
tenv.executeSql("show tables").print();
}

}

```

可以在上述代码运行前，设置 debug 断点，然后就可以清晰看到 catalogManager 中的各个元数据空间及临时表空间的情况

评估表达式(Enter)或添加监视(Ctrl+Shift+Enter)

- > └ this = {StreamTableEnvironmentImpl@5283}
- └ catalogManager = {CatalogManager@5284} **两个元数据管理空间**
 - └ f catalogs = {LinkedHashMap@5296} size = 2
 - > └ "default_catalog" -> {GenericInMemoryCatalog@5306} **内存catalog**
 - > └ "mycatalog" -> {HiveCatalog@5308} **hive catalog**
- └ temporaryTables = {HashMap@5297} size = 2
 - > └ {ObjectIdentifier@5313} <"mycatalog`.`default`.`t_kafka"> {ResolvedCatalogTable@5314}
 - > └ {ObjectIdentifier@5315} <"default_catalog`.`default_database`.`t_kafka2"> {ResolvedCatalogTable@5316}
- > f currentCatalogName = "default_catalog"
- > f currentDatabaseName = "default_database"
- > f schemaResolver = {DefaultSchemaResolver@5300}
- > f builtInCatalogName = "default_catalog"
- > f typeFactory = {DataTypeFactoryImpl@5301}

6.3. 临时表与永久表的底层差异

结论 1：如果选择 hive 元数据空间来创建表、视图，则

- 永久表（视图）的元信息，都会被写入 hive 的元数据管理器中，从而可以实现永久存在
- 临时表（视图）的元信息，并不会写入 hive 的元数据管理其中，而是放在 catalogManager 的一个 temporaryTables 的内存 hashmap 中记录
- 临时表空间中的表名（全名）如果与 hive 空间中的表名相同，则查询时会优先选择临时表空间的表

结论 2：如果选择 GenericInMemoryCatalog 元数据空间来创建表、视图，则

- 永久表（视图）的元信息，都会被写入 GenericInMemoryCatalog 的元数据管理器中（内存中）
- 临时表（视图）的元信息，放在 catalogManager 的一个 temporaryTables 的内存 hashmap 中记录
- 无论永久还是临时，当 flink 的运行 session 结束后，所创建的表（永久、临时）都将不复存在

6.4. 如何理解 hive catalog

flinksql 利用 hive catalog 来建表（查询、修改、删除表），本质上只是利用了 hive 的 metastore 服务；

更具体来说，flinksql 只是把 flinksql 的表定义信息，按照 hive 元数据的形式，托管到 hive 的 metastore 中而已！

当然，hive 中也能看到这些托管的表信息，但是，并不能利用它底层的 mapreduce 或者 spark 引擎来查询这些表；

因为 mapreduce 或者 spark 引擎，并不能理解 flinksql 表定义中的信息，也无法为这些定义信息提供相应的组件去读取数据（比如，mr 或者 spark 就没有 flinksql 中的各种 connector 组件）

7. 表定义详解

定义表时所需的核心要素

- 表名 (catalog_name.database_name.object_name)
- TableDescriptor

TableDescriptor 核心要素

- Connector 连接器
- Format 数据格式
- Schema 表结构 (字段)
- Option 连接器参数

7.1. schema 字段定义详解

7.1.1. physical column

物理字段：源自于“外部存储”系统本身 schema 中的字段

如 kafka 消息的 key、value (json 格式) 中的字段；
mysql 表中的字段；hive 表中的字段；parquet 文件中的字段.....

7.1.2. computed column

表达式字段 (逻辑字段)：在物理字段上施加一个 sql 表达式，并将表达式结果定义为一个字段

TableApi 中的定义方式

```
Schema.newBuilder()  
// 声明表达式字段 age_exp, 它来源于物理字段 age+10  
.columnByExpression("age_exp", "age+10")
```

Sql DDL 中的定义方式

```
CREATE TABLE MyTable (
    `user_id` BIGINT,
    `price` DOUBLE,
    `quantity` DOUBLE,
    `cost` AS price * quantity, -- cost 来源于: price*quantity
) WITH (
    'connector' = 'kafka'
    ...
);
```

7.1.3. metadata column

元数据字段：来源于 connector 从外部存储系统中获取到的“外部系统元信息”

比如， kafka 的消息，通常意义上的数据内容是在 record 的 key 和 value 中的，而实质上（底层角度来看）， kafka 中的每一条 record，不光带了 key 和 value 数据内容，还带了这条 record 所属的 topic，所属的 partition，所在的 offset，以及 record 的 timestamp 和 timestamp 类型等“元信息”

而 flink 的 connector 可以获取并暴露这些元信息，并允许用户将这些信息定义成 flinksql 表中的字段；

TableApi 中的定义方式

```
Schema.newBuilder()
.columnByMetadata("topic", DataTypes.STRING())
```

Sql DDL 中的定义方式

```
CREATE TABLE MyTable (
    `user_id` BIGINT,
    `name` STRING,
    -- 元数据字段，来源于 kafka record 的 timestamp
    `record_time` TIMESTAMP_LTZ(3) METADATA FROM 'timestamp'
) WITH (
    'connector' = 'kafka'
    ...
);
```

7.1.4. 主键约束

- 单字段主键约束语法:

```
id INT PRIMARY KEY NOT ENFORCED,  
name STRING
```

- 多字段主键约束语法:

```
id,  
name,  
PRIMARY KEY(id,name) NOT ENFORCED
```

7.1.5. 表字段定义完整实例

- sql 的 DDL 方式

```
TableEnvironment tenv = TableEnvironment.create(EnvironmentSettings.inStreamingMode());  
  
// 建表（数据源表）  
// {"id":4,"name":"zs","nick":"tiedan","age":18,"gender":"male"}  
tenv.executeSql(  
    "create table t_person  
    + "("  
        + " id int,           " // 物理字段  
        + " name string,      " // 物理字段  
        + " nick string,       "  
        + " age int,           "  
        + " gender string,     "  
        + " guid as id,        " // 表达式字段（逻辑字段）  
        + " big_age as age + 10, " // 表达式字段（逻辑字段）  
        + " offs bigint metadata from 'offset',      " // 元数据字段  
        + " ts TIMESTAMP_LTZ(3) metadata from 'timestamp'  " // 元数据字段  
    /*+ " PRIMARY KEY(id,name) NOT ENFORCED */ // -- 主键约束  
    + ")"  
    + " WITH ("  
        + " 'connector' = 'kafka',  
        + " 'topic' = 'doit30-4',  
        + " 'properties.bootstrap.servers' = 'doitedu:9092',  "  
        + " 'properties.group.id' = 'g1',  
        + " 'scan.startup.mode' = 'earliest-offset',          "  
        + " 'format' = 'json',  
    )
```

```
+ "  'json.fail-on-missing-field' = 'false',          "
+ "  'json.ignore-parse-errors' = 'true'           "
+ " )"
);

tenv.executeSql("desc t_person").print();
tenv.executeSql("select * from t_person").print();
```

● api 的方式

```
// 建表（数据源表）
// {"id":4,"name":"zs","nick":"tiedan","age":18,"gender":"male"}
tenv.createTable("t_person",
    TableDescriptor
        .forConnector("kafka")
        .schema(Schema.newBuilder()
            .column("id", DataTypes.INT()) // column 是声明物理字段到表结构中来
            .column("name", DataTypes.STRING()) // column 是声明物理字段到表结构中来
            .column("nick", DataTypes.STRING()) // column 是声明物理字段到表结构中来
            .column("age", DataTypes.INT()) // column 是声明物理字段到表结构中来
            .column("gender", DataTypes.STRING()) // column 是声明物理字段到表结构中来
            .columnByExpression("guid","id") // 声明表达式字段
            /*.columnByExpression("big_age",$("age").plus(10))*/ // 声明表达式字段
            .columnByExpression("big_age","age + 10") // 声明表达式字段
            // isVirtual 是表示：当这个表被 sink 表时，该字段是否出现在 schema 中
            .columnByMetadata("offs",DataTypes.BIGINT(),"offset",true) // 声明元数据字段
            .columnByMetadata("ts",DataTypes.TIMESTAMP_LTZ(3),"timestamp",true) // 声明元数据字段
            /*.primaryKey("id","name")*/
            .build())
        .format("json")
        .option("topic","doit30-4")
        .option("properties.bootstrap.servers","doitedu:9092")
        .option("properties.group.id","g1")
        .option("scan.startup.mode","earliest-offset")
        .option("json.fail-on-missing-field","false")
        .option("json.ignore-parse-errors","true")
        .build()
);

tenv.executeSql("select * from t_person").print();
```

7.2. format 概述

connector 连接器在对接外部存储时，根据外部存储中的数据格式不同，需要用到不同的 format 组件；

format 组件的作用就是：告诉连接器，如何解析外部存储中的数据及映射到表 schema；

format 组件的使用要点

- 导入 format 组件的 jar 包依赖
- 指定 format 组件的名称
- 设置 format 组件所需的参数（不同 format 组件有不同的参数配置需求）

```
CREATE TABLE user_behavior (
    user_id BIGINT,
    item_id BIGINT,
    category_id BIGINT,
    behavior STRING,
    ts TIMESTAMP(3)
) WITH (
    'connector' = 'kafka',
    'topic' = 'user_behavior',
    'properties.bootstrap.servers' = 'localhost:9092',
    'properties.group.id' = 'testGroup',
    'format' = 'csv',
    'csv.ignore-parse-errors' = 'true',
    'csv.allow-comments' = 'true'
)
```

flinksql 目前支持的 format 如下

Formats	Supported Connectors
CSV	Apache Kafka, Upsert Kafka, Amazon Kinesis Data Streams, Filesystem
JSON	Apache Kafka, Upsert Kafka, Amazon Kinesis Data Streams, Filesystem, Elasticsearch
Apache Avro	Apache Kafka, Upsert Kafka, Amazon Kinesis Data Streams, Filesystem
Confluent Avro	Apache Kafka, Upsert Kafka
Debezium CDC	Apache Kafka, Filesystem
Canal CDC	Apache Kafka, Filesystem
Maxwell CDC	Apache Kafka, Filesystem
OGG CDC	Apache Kafka, Filesystem
Apache Parquet	Filesystem
Apache ORC	Filesystem
Raw	Apache Kafka, Upsert Kafka, Amazon Kinesis Data Streams, Filesystem

7.2.1. json format 详解

所需依赖

```
<dependency>
<groupId>org.apache.flink</groupId>
<artifactId>flink-json</artifactId>
<version>1.15.0</version>
</dependency>
```

可用参数

format 组件名: json
json.fail-on-missing-field 缺失字段是否失败
json.ignore-parse-errors 是否忽略 json 解析错误
json.timestamp-format.standard json 中的 timestamp 类型字段的格式
json.map-null-key.mode 可取: FAIL, DROP, LITERAL
json.map-null-key.literal 替换 null 的字符串
json.encode.decimal-as-plain-number

官方详细文档: [JSON | Apache Flink](#)

数据类型映射

Flink SQL type	JSON type
CHAR / VARCHAR / STRING	string
BOOLEAN	boolean
BINARY / VARBINARY	string with encoding: base64
DECIMAL	number
TINYINT	number
SMALLINT	number
INT	number
BIGINT	number
FLOAT	number
DOUBLE	number
DATE	string with format: date
TIME	string with format: time
TIMESTAMP	string with format: date-time
TIMESTAMP_WITH_LOCAL_TIME_ZONE	string with format: date-time (with UTC time zone)
INTERVAL	number
ARRAY	array

Flink SQL type	JSON type
MAP / MULTISET	object
ROW	object

基本使用

- 假设 kafka 或文件中有如下数据：

```
{"id":10,"name":"nick","age":28,"regTime":"2022-06-06 12:30:30.100"}
```

- 映射成 flinksql 表

```
// {"id":10,"name":"nick","age":28,"regTime":"2022-06-06 12:30:30.100"}
```

```
Schema schema = Schema.newBuilder()
```

```
.column("id", DataTypes.INT())
.column("name", DataTypes.STRING())
.column("age", DataTypes.INT())
.column("regTime", DataTypes.TIMESTAMP(3))
.build();
```

```
tenv.createTable("t1",
```

```
TableDescriptor.forConnector("filesystem")
.option("path", "file:///d:/json.txt")
.format("json")
.schema(schema)
.build());
```

```
tenv.executeSql("select id,name,age,regTime from t1").print();
```

```
/*
```

```
+-----+
| op | id | name | age | regTime |
+-----+
| +I | 10 | nick | 28 | 2022-06-06 12:30:30.100 |
+-----+
```

```
*/
```

复杂 json 格式解析 (嵌套对象)

- 有如下嵌套 json

```
{"id":10,"name":{"nick":"doe","formal":"doit edu"}}
```

- 映射成 flinksql 表

```
Schema schema = Schema.newBuilder()
    .column("id", DataTypes.INT())
    .column("name", DataTypes.ROW(
        DataTypes.FIELD("nick", DataTypes.STRING()),
        DataTypes.FIELD("formal", DataTypes.STRING())
    ))
    .build();
```

- 查询

```
select id,name.nick,name.formal from t
```

复杂 json 格式解析 (嵌套数组, 数组内又嵌套对象)

- 有如下嵌套 json

```
{"id":1,"friends":[{"name":"a","info":{"addr":"bj","gender":"male"}},{"name":"b","info":{"addr":"sh","gender":"female"}}]}
```

- 映射成 flinksql 表

```
Schema schema3 = Schema.newBuilder()
    .column("id", DataTypes.INT())
    .column("friends", DataTypes.ARRAY(
        DataTypes.ROW(
            DataTypes.FIELD("name", DataTypes.STRING()),
            DataTypes.ROW(
                DataTypes.FIELD("addr", DataTypes.STRING()),
                DataTypes.FIELD("gender", DataTypes.STRING())
            )
        )))
    .build();
```

- 查询

```
select id,friends[1].name,friends[1].info.addr from t1
```

7.2.2. csv format 详解

- 添加依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-csv</artifactId>
    <version>1.14.4</version>
</dependency>
```

- 可用参数

```
format = csv
csv.field-delimiter = ','
csv.disable-quote-character = false
csv.quote-character = "'"
csv.allow-comments = false
csv.ignore-parse-errors = false    是否忽略解析错误
csv.array-element-delimiter = ';'  数组元素之间的分隔符
csv.escape-character = none      转义字符
csv.null-literal = none        null 的字面量字符串
```

- 代码示例

```
tenv.executeSql(
    "create table t_csv(
        + " id int,
        + " name string,
        + " age  string
        + ") with (
        + " 'connector' = 'filesystem',
        + " 'path' = 'data/csv/',
        + " 'format'='csv',
        + " 'csv.disable-quote-character' = 'false',
        + " 'csv.quote-character' = '|',
        + " 'csv.ignore-parse-errors' = 'true',
        + " 'csv.null-literal' = '\\N',
        + " 'csv.allow-comments' = 'true'
        + ")
);
tenv.executeSql("desc t_csv").print();
tenv.executeSql("select * from  t_csv").print();
```

7.3. watermark 与时间属性详解

时间属性定义，主要是用于各类基于时间的运算操作（如基于时间窗口的查询计算）

7.3.1. eventTime 与 waterMark 定义

```
// guid,uuid,eventId,pageId,ts
DataStreamSource<String> ss = env.socketTextStream("doitedu", 9999);
// 换成 pojo 类型的 datastream
SingleOutputStreamOperator<DataBean> ds = ss.map(s -> {
    String[] arr = s.split(",");
    return new DataBean(Integer.parseInt(arr[0]), arr[1], arr[2], arr[3], Long.parseLong(arr[4]));
});
// 分配 watermark 策略
SingleOutputStreamOperator<DataBean> ds2 = ds.assignTimestampsAndWatermarks(
    WatermarkStrategy.<DataBean>forMonotonousTimestamps()
        .withTimestampAssigner((element, recordTimestamp) -> element.ts));

// 转成 table
Table table2 = tenv.fromDataStream(ds2, Schema.newBuilder()
    // 声明表达式字段，并声明为 processing time 属性字段
    .columnByExpression("pt", "proctime()")
    // 声明表达式字段（来自 ts）
    .columnByExpression("rt", "to_timestamp_ltz(ts,3)")
    // 将 rt 字段指定为 event time 属性字段，并基于它指定 watermark 策略： = rt
    .watermark("rt", "rt")
    // 将 rt 字段指定为 event time 属性字段，并基于它指定 watermark 策略： = rt - 8s
    .watermark("rt", "rt - interval '8' second")
    // 将 rt 字段指定为 event time 属性字段，并沿用“源头流”的 watermark
    .watermark("rt", "SOURCE_WATERMARK()") // 得到与源头 watermark 完全一致
    .build());
table2.printSchema();
```

```
watermark for timeAttrColumn AS timeAttrColumn - interval '1' second,
ptime AS proctime()
```

核心要点：

- 需要一个 timestamp(3)类型字段（可以是物理字段，也可以是表达式字段，也可以是元数据字段）
- 需要用一个 watermarkExpression 来指定 watermark 策略（有 2 种表达式）

7.3.2. 表与流之间 watermark 传承

流转表时

流转表的过程中，无论“源流”是否存在 watermark，都不会自动传递 watermark

如需时间运算（如时间窗口等），需要在转换定义中显式声明 watermark 策略：

- 先设法定义一个 timestamp(3)或者 timestamp_ltz(3)类型的字段（可以来自于数据字段，也可以来自于一个元数据： rowtime

```
rt as to_timestamp_ltz(eventTime) -- 从一个 bigint 得到一个 timestamp (3) 类型的字段
```

```
rt timestamp(3) metadata from 'rowtime' -- 从一个元数据 rowtime 得到一个 timestamp(3)类型的字段
```

- 然后基于该字段，用 watermarkExpression 声明 watermark 策略

```
watermark for rt AS rt - interval '1' second
```

```
watermark for rt AS source_watermark() -- 代表使用底层流的 watermark 策略
```

表转流时

前提：源表定义了 watermark 策略；

则将表转成流时，将会自动传递源表的 watermark；

可通过类似如下代码进行测试：

```
/*
 * 前提： table2 已经是一个存在 watermark 的表对象，转成 datastream 后，打印 watermark
 */
tenv.toDataStream(table2)
    .process(new ProcessFunction<Row, String>() {
        @Override
        public void processElement(Row value, Context ctx, Collector<String> out) throws Exception {
            long wt = ctx.timerService().currentWatermark();
            System.out.println(wt + " => " + value);
        }
    }).print();
```

7.3.3. processing time

定义一个表达式字段，并用表达式 `proctime()` 将其声明为 processing time 即可；

```
// 转成 table
Table table = tenv.fromDataStream(ds, Schema.newBuilder()
    // 声明表达式字段，并声明为 processing time 属性字段
    .columnByExpression("pt", "proctime()")
```

7.4. connector 详解

7.4.1. connector 概述

- ✓ connector 通常是用于对接外部存储建表（源表或目标表）时的映射器、桥接器
- ✓ connector 本质上是对 flink 的 table source /table sink 算子的封装；

连接器使用的核心要素：

- ✓ 导入连接器 jar 包依赖
- ✓ 指定连接器类型名
- ✓ 指定连接器所需的参数（不同连接器有不同的参数配置需求）
- ✓ 获取连接器所提供的元数据

连接器使用的速览示例

- 导入依赖

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka</artifactId>
  <version>1.14.4</version>
</dependency>
```

- 配置参数

[Table Api] 方式

```
tenv.createTable( path: "kfkTable",
    TableDescriptor
        .forConnector("kafka")
        .format("csv")
        .option("topic", "abc")
        .option("properties.bootstrap.servers", "doitedu:9092")
        .option("scan.startup.mode", "earliest-offset")
    .build());
```

[Table Sql] 方式

```
CREATE TABLE KafkaTable (
    `user_id` BIGINT,
    `item_id` BIGINT,
    `behavior` STRING,
    `ts` TIMESTAMP(3) METADATA FROM 'timestamp'
) WITH (
    'connector' = 'kafka',
    'topic' = 'user_behavior',
    'properties.bootstrap.servers' = 'localhost:9092',
    'properties.group.id' = 'testGroup',
    'scan.startup.mode' = 'earliest-offset',
    'format' = 'csv'
)
```

FlinkSql 内置支持的 connector

Name	Version	Source	Sink
Filesystem		Bounded and Unbounded Scan, Lookup	Streaming Sink, Batch Sink
Elasticsearch	6.x & 7.x	Not supported	Streaming Sink, Batch Sink
Apache Kafka	0.10+	Unbounded Scan	Streaming Sink, Batch Sink
Amazon Kinesis Data Streams		Unbounded Scan	Streaming Sink
JDBC		Bounded Scan, Lookup	Streaming Sink, Batch Sink
Apache HBase	1.4.x & 2.2.x	Bounded Scan, Lookup	Streaming Sink, Batch Sink
Apache Hive	Supported Versions	Unbounded Scan, Bounded Scan, Lookup	Streaming Sink, Batch Sink

7.4.2. kafka connector 详解

产生的数据以及能接受的数据流，是 **append-only** 流（只有 +I 这种 changemode）

所需依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_2.11</artifactId>
    <version>1.14.4</version>
</dependency>
```

建表语句示例

```
CREATE TABLE KafkaTable (
    `user_id` BIGINT,
    `item_id` BIGINT,
    `behavior` STRING,
    `ts` TIMESTAMP(3) METADATA FROM 'timestamp'
) WITH (
    'connector' = 'kafka',
    'topic' = 'user_behavior',
    'properties.bootstrap.servers' = 'localhost:9092',
    'properties.group.id' = 'testGroup',
    'scan.startup.mode' = 'earliest-offset',
    'format' = 'csv'
)
```

可用的元数据（来自于 kafka 的 record 中的 metadata）

Key	Data Type	Description	R/ W
topic	STRING NOT NULL	所属 topic 名称.	R
partition	INT NOT NULL	record 所属 partition	R
headers	MAP NOT NULL	record 的 headers: Map<STRING,bytes>.	R/W
leader-epoch	INT NULL	record 所属的 leader-epoch.	R
offset	BIGINT NOT NULL	record 的 offset.	R
timestamp	TIMESTAMP_LTZ(3) NOT NULL	record 的 timestamp.	R/W
timestamp-type	STRING NOT NULL	Timestamp 类型: NoTimestampType, CreateTime, LogAppendTime	R

可用的参数（部分展示）

connector
 topic / topic-pattern
 properties.bootstrap.servers
 properties.* 传给 kafka-client 的（会自动去除 properties. 前缀）
 format / key.format / value.format
 key.fields 如 'fields1;fields2'
 scan.startup.mode 可取：earliest-offset, latest-offset, group-offsets, timestamp, specific-offsets
 scan.startup.specific-offsets
 sink.partitionner 默认 default, 可取 fixed, round-robin, 自定义类 'cn.doitedu.MyPartitioner'
 sink.delivery-guarantee 可取 at-least-once, exactly-once, none
 sink.parallelism 默认为 none, 根据算子 chain 决定

详细参数见官方文档：[Kafka | Apache Flink](#)

超级完整使用示例

- 假设 kafka 中有如下数据

- kafka 的消息中有 json 格式的 key (key 内容需要映射到表 schema 中)
- kafka 的消息中有 json 格式的 value (value 内容需要映射到表 schema 中)
- key 和 value 的 json 数据内容中还有同名的字段
- kafka 的消息中有 header (header 内容需要映射到表 schema 中)

- 使用 kafka-connector 映射源表

```

CREATE TABLE KafkaTable (
  `meta_time` TIMESTAMP(3) METADATA FROM 'timestamp',
  `partition` BIGINT METADATA VIRTUAL,
  `offset` BIGINT METADATA VIRTUAL,
  `inKey_k1` BIGINT,
  `inKey_k2` STRING,
)
  
```

```
`guid` BIGINT,  
`eventId` STRING,  
`k1` STRING,  
`k2` STRING,  
`eventTime` BIGINT,  
`headers` MAP<STRING,BYTES> METADATA FROM 'headers'  
 ) WITH (  
 'connector' = 'kafka',  
 'topic' = 'abc',  
 'properties.bootstrap.servers' = 'doitedu:9092',  
 'properties.group.id' = 'testGroup',  
 'scan.startup.mode' = 'earliest-offset',  
 'key.format'='json',  
 'key.json.ignore-parse-errors' = 'true',  
 'key.fields'='inKey_k1;inKey_k2',  
 'key.fields-prefix'='inKey_',  
 'value.format'='json',  
 'value.json.fail-on-missing-field'='false',  
 'value.fields-include' = 'EXCEPT_KEY'  
 );
```

● 使用 kafka-connector 映射目标表

```
CREATE TABLE KafkaTable2 (  
 `meta_time` TIMESTAMP(3) METADATA FROM 'timestamp',  
 `partition` BIGINT METADATA VIRTUAL,  
 `offset` BIGINT METADATA VIRTUAL,  
 `inKey_k1` BIGINT,  
 `inKey_k2` STRING,  
 `guid` BIGINT,  
 `eventId` STRING,  
 `k1` STRING,  
 `k2` STRING,  
 `eventTime` BIGINT,  
 `headers` MAP<STRING,BYTES> METADATA FROM 'headers'  
 ) WITH (  
 'connector' = 'kafka',  
 'topic' = 'abc',  
 'properties.bootstrap.servers' = 'doitedu:9092',  
 'properties.group.id' = 'testGroup',
```

```
'scan.startup.mode' = 'earliest-offset',
'key.format'='json',
'key.json.ignore-parse-errors' = 'true',
'key.fields'='inKey_k1;inKey_k2',
'key.fields-prefix'='inKey_',
'value.format'='json',
'value.json.fail-on-missing-field'='false',
'value.fields-include' = 'EXCEPT_KEY'
)
```



7.4.3. upsert-kafka-connector

基本工作机制

- 作为 source

根据所定义的主键，将读取到的数据转换为 +I/-U/+U 记录，如果读到 null，则转换为-D 记录；

-- kafka 中假设有如下数据

1,zs,18

1,zs,28

kafka - connector 产生出 appendonly 流

+I [1,zs,18]

+I [1,zs,28]

upsert-kafka-connector 产生出 upsert 模式的 changelog 流

+I [1,zs,18]

-U [1,zs,18]

+U [1,zs,28]

- 作为 sink

对于 -U/+U/+I 记录，都以正常的 append 消息写入 kafka;

对于-D 记录，则写入一个 null 到 kafka 来表示 delete 操作；

代码示例

```
tenv.executeSql(  
    "CREATE TABLE upsert_kafka (          ""  
        + " province STRING,          ""  
        + " pv BIGINT,          ""  
        + " PRIMARY KEY (province) NOT ENFORCED  ""  
        + " ) WITH (          ""  
        + " 'connector' = 'upsert-kafka',          ""  
        + " 'topic' = 'upsert_kafka2',          ""  
        + " 'properties.bootstrap.servers' = 'doitedu:9092',  ""  
        + " 'key.format' = 'csv',          ""  
        + " 'value.format' = 'csv'          ""  
        + " )          ""
```

```
);
```

```
DataStreamSource<Row> stream = env.fromElements(
```

```
    Row.ofKind(RowKind.INSERT, "sx", 1),  
    Row.ofKind(RowKind.INSERT, "sx", 2),  
    Row.ofKind(RowKind.INSERT, "gx", 1),  
    Row.ofKind(RowKind.INSERT, "sx", 2),  
    Row.ofKind(RowKind.INSERT, "gx", 2));
```

```
tenv.createTemporaryView("s", stream, Schema.newBuilder()
```

```
.column("f0", DataTypes.STRING().notNull())  
.column("f1", DataTypes.INT())  
.build());
```

```
// 将查询结果（changelog 流），写入 kafka
```

```
tenv.executeSql("insert into upsert_kafka select f0,sum(f1) as pv from s group by f0");
```

```
/*
```

```
+-----+-----+  
| op |      f0 |      pv |  
+-----+-----+  
| +I |      sx |      1 |  
| -U |      sx |      1 |  
| +U |      sx |      3 |  
| +I |      gx |      1 |  
| -U |      sx |      3 |  
| +U |      sx |      5 |  
| -U |      gx |      1 |  
| +U |      gx |      3 |  
+-----+
```

```
*/
```

```
// 从 kafka 再读出上面的 changelog 结果
```

```
tenv.executeSql("  select * from upsert_kafka").print();
```

```
/*
```

```
+-----+-----+  
| op |      province |      pv |  
+-----+-----+  
| +I |      sx |      1 |  
| -U |      sx |      1 |  
| +U |      sx |      3 |
```

+I	gx	1
-U	sx	3
+U	sx	5
-U	gx	1
+U	gx	3
*/		

7.4.4. jdbc connector 详解

jdbc connector 有如下特性

- 可作为 **scan source** , 底层产生 Bounded Stream
- 可作为 **lookup source**, 底层是“事件驱动”式查询
- 可作为 Batch 模式的 sink
- 可作为 Stream 模式下的 append sink 和 upsert sink

官方文档: [JDBC | Apache Flink](#)

所需依赖

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-jdbc</artifactId>
  <version>1.14.4</version>
</dependency>
```

根据所连接的数据库不同, 需要相应的 jdbc 驱动, 比如连接 mysql

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.21</version>
</dependency>
```

建表语句示例

```
-- 将 mysql 的 users 表，注册到 flinksql 中
CREATE TABLE MyUserTable (
    id BIGINT,
    name STRING,
    age INT,
    status BOOLEAN,
    PRIMARY KEY (id) NOT ENFORCED
) WITH (
    'connector' = 'jdbc',
    'url' = 'jdbc:mysql://doitedu:3306/mydatabase',
    'table-name' = 'users'
);
```

-- 将一个查询结果，通过 jdbc 连接器表插入到 mysql 表中

```
INSERT INTO MyUserTable
SELECT id, name, age, status FROM T;
```

-- 从 jdbc 连接器表查询数据（scan 模式）

```
SELECT id, name, age, status FROM MyUserTable;
```

-- 将 jdbc 连接器表作为一个维表进行时态关联（temporal join 详见后续的查询详解章节）

```
SELECT * FROM myTopic
LEFT JOIN MyUserTable FOR SYSTEM_TIME AS OF myTopic.proctime
ON myTopic.key = MyUserTable.id;
```

幂等写出

jdbc connector 可以利用目标数据库的特性，实现幂等写出；

幂等写出可以避免在 failover 发生后的可能产生的数据重复；

实现幂等写出，本身并不需要对 jdbc connector 做额外的配置，只需要：指定主键字段，jdbc connector 就会利用目标数据库的 upsert 语法（如下），来实现幂等写出；

不同数据库的 upsert 语法不同，如下：

Database	Upsert 语法
MySQL	INSERT .. ON DUPLICATE KEY UPDATE ..
Oracle	MERGE INTO .. USING(..) ON(..) WHEN MATCHED THEN UPDATE SET(..) WHEN NOT MATCHED THEN INSERT(..)

Database	Upsert 语法
	VALUES(..)
PostgreSQL	INSERT .. ON CONFLICT .. DO UPDATE SET ..

分区并行读取 (partitioned scan)

jdbc connector 持有一个多并行度的 source task，因而可以多并行度加快表数据的读取；

通过设置如下参数即可实现多并行读取：

- scan.partition.column: 划分并行任务的参照列.
- scan.partition.num: 任务并行数.
- scan.partition.lower-bound: 首分区的参照字段最小值.
- scan.partition.upper-bound: 末分区的参照字段最大值.

分区参照字段必须是： numeric, date, 或 timestamp 类型

数据类型映射

连接外部数据库，必须要 format 组件来指定数据的解析规则；

- ✓ 因为：数据库，本身就是结构化数据，本身就有 schema 定义；
- ✓ 但是：外部数据库中的数据类型，与 flinksql 中的类型体系，难以做到完全一致（不同数据库总的类型体系就是各有差别）
- ✓ 所以：在 flinksql 表和外部数据库表之间，存在数据类型的映射转换；

类型映射详表：

MySQL type	Oracle type	PostgreSQL type	Flink SQL type
TINYINT			TINYINT
SMALLINT		SMALLINT	
TINYINT UNSIGNED		INT2 SMALLSERIAL SERIAL2	SMALLINT

MySQL type	Oracle type	PostgreSQL type	Flink SQL type
INT		INTEGER	INT
MEDIUMINT		SERIAL	
SMALLINT UNSIGNED			
BIGINT		BIGINT	BIGINT
INT UNSIGNED		BIGSERIAL	
BIGINT UNSIGNED			DECIMAL(20, 0)
BIGINT		BIGINT	BIGINT
FLOAT	BINARY_FLOAT	REAL FLOAT4	FLOAT
DOUBLE	BINARY_DOUBLE	FLOAT8	DOUBLE
DOUBLE PRECISION		DOUBLE PRECISION	
NUMERIC(p, s)	SMALLINT FLOAT(s)	NUMERIC(p, s)	DECIMAL(p, s)
DECIMAL(p, s)	DOUBLE PRECISION REAL NUMBER(p, s)	DECIMAL(p, s)	
BOOLEAN		BOOLEAN	BOOLEAN
TINYINT(1)			
DATE	DATE	DATE	DATE
TIME [(p)]	DATE	TIME [(p)] [WITHOUT TIMEZONE]	TIME [(p)] [WITHOUT TIMEZONE]
DATETIME [(p)]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]
CHAR(n)	CHAR(n)	CHAR(n)	
VARCHAR(n)	VARCHAR(n)	CHARACTER(n)	
TEXT	CLOB	VARCHAR(n) CHARACTER VARYING(n) TEXT	STRING
BINARY			
VARBINARY	RAW(s)		
BLOB	BLOB	BYTEA	BYTES
		ARRAY	ARRAY

可用参数

connector 连接器名称: jdbc

url 数据库地址串

table-name 表名

driver 驱动类名

username 数据库登录用户名

password 数据库登录密码

.....

更多参数，详见官方文档：[JDBC | Apache Flink](#)

7.4.5. filesystem connector

filesystem connector 表特性

- 可读可写
- 作为 source 表时，支持持续监视读取目录下新文件，且每个新文件只会被读取一次
- 作为 sink 表时，支持 多种文件格式、分区、文件滚动、压缩设置等功能

参数举例

path

format

source.monitor-interval

sink.rolling-policy.file-size

sink.rolling-policy.rollover-interval

sink.rolling-policy.check-interval

auto-compaction=false 自动压缩

compaction.file-size = rolling file size 压缩文件目标大小

sink.partition-commit.trigger 分区提交触发器，可用 process-time,partition-time (需要 watermark)

sink.partition-commit.delay 提交延迟，可用 '1 d', '1 h', '1 s' 等

sink.partition-commit.policy.kind = success-file，提交策略，可取 metastore

sink.parallelism = none sink 并行度

代码示例

```
package cn.doitedu.flinksql.demos;
```

```
public class Demo13_FileSystemConnectorTest {
```

```
public static void main(String[] args) throws Exception {  
  
    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
    env.enableCheckpointing(1000, CheckpointingMode.EXACTLY_ONCE);  
    env.getCheckpointConfig().setCheckpointStorage("file:///d:/checkpoint");  
    env.setRuntimeMode(RuntimeExecutionMode.STREAMING);  
  
    EnvironmentSettings environmentSettings = EnvironmentSettings.inStreamingMode();  
    StreamTableEnvironment tenv = StreamTableEnvironment.create(env, environmentSettings);  
  
    // 建表来映射 mysql 中的 flinktest.stu  
    tenv.executeSql(  
        "CREATE TABLE fs_table (\n" +  
            " user_id STRING,\n" +  
            " order_amount DOUBLE,\n" +  
            " dt STRING,\n" +  
            " `hour` STRING\n" +  
            ") PARTITIONED BY (dt, `hour`) WITH (\n" +  
            " 'connector'='filesystem',\n" +  
            " 'path'='file:///d:/filetable/',\n" +  
            " 'format'='json',\n" +  
            " 'sink.partition-commit.delay'='1 h',\n" +  
            " 'sink.partition-commit.policy.kind'='success-file',\n" +  
            " 'sink.rolling-policy.file-size' = '8M',\n" +  
            " 'sink.rolling-policy.rollover-interval'='30 min',\n" +  
            " 'sink.rolling-policy.check-interval'='10 second'\n" +  
        ")"  
    );  
  
    // u01,88.8,2022-06-13,14  
    SingleOutputStreamOperator<Tuple4<String, Double, String, String>> stream = env  
        .socketTextStream("doitedu", 9999)  
        .map(s -> {  
            String[] split = s.split(",");  
            return Tuple4.of(split[0], Double.parseDouble(split[1]), split[2], split[3]);  
        }).returns(new TypeHint<Tuple4<String, Double, String, String>>() {  
    });  
  
    tenv.createTemporaryView("orders",stream);  
  
    tenv.executeSql("insert into fs_table select * from orders");
```

```
    env.execute();
}

@Data
@NoArgsConstructor
@AllArgsConstructor
public static class Bean1 {
    public int id;
    public String gender;
}

@Data
@NoArgsConstructor
@AllArgsConstructor
public static class Bean2 {
    public int id;
    public String name;
}
```

7.4.6. 其他三方 connector 举例

flink-doris-connector (用于读写 doris)

```
CREATE TABLE cdc_mysql_source (
    id int
    ,name VARCHAR
    ,PRIMARY KEY (id) NOT ENFORCED
) WITH (
    'connector' = 'mysql-cdc',
    'hostname' = 'doitedu',
    'port' = '3306',
    'username' = 'root',
    'password' = 'password',
    'database-name' = 'db_doit',
    'table-name' = 't_doit'
);

-- 支持删除事件同步(sink.enable-delete='true'),需要 Doris 表开启批量删除功能
CREATE TABLE doris_sink (
    id INT,
    name STRING
)
WITH (
    'connector' = 'doris',
    'fenodes' = 'doitedu:8030',
    'table.identifier' = 'db_doit.t_doit',
    'username' = 'root',
    'password' = '',
    'sink.properties.format' = 'json',
    'sink.properties.strip_outer_array' = 'true',
    'sink.enable-delete' = 'true'
);

insert into doris_sink select id,name from cdc_mysql_source;
```

flink-hudi-connector (用于读写 hudi)

```
CREATE TABLE hudi_join
(
    id      INT PRIMARY KEY NOT ENFORCED,
```

```
    fname string,
    lname string,
    age int,
    gender string,
    score double
) PARTITIONED BY (gender)
WITH (
    'connector' = 'hudi',
    'path' = 'hdfs://doitedu:8020/lake/hudi_join',
    'table.type' = 'COPY_ON_WRITE',
    'connector' = 'hudi',
    'hive_sync.enable' = 'true',
    'hive_sync.mode' = 'jdbc',
    'hive_sync.metastore.uris' = 'thrift://doitedu:9083',
    'hive_sync.jdbc_url'='jdbc:hive2://doitedu:10000',
    'hive_sync.table'='hudi_join',
    'hive_sync.db'='default',
    'hive_sync.username'='root',
    'hive_sync.password'='',
    'read.streaming.enabled' = 'true',
    'read.streaming.check-interval' = '1'
);
```

7.5. 表定义完整语法

```
CREATE TABLE [IF NOT EXISTS] [catalog_name.][db_name.]table_name
(
    { <physical_column_definition> | <metadata_column_definition> | <computed_column_definition> }[ , ...n]
    [ <watermark_definition> ]
    [ <table_constraint> ][ , ...n]
)
[COMMENT table_comment]
[PARTITIONED BY (partition_column_name1, partition_column_name2, ...)]
[WITH (key1=val1, key2=val2, ...)]
[ LIKE source_table [( <like_options> )] ]

<physical_column_definition>:
    column_name column_type [ <column_constraint> ] [COMMENT column_comment]
```

```
<column_constraint>:  
[CONSTRAINT constraint_name] PRIMARY KEY NOT ENFORCED  
  
<table_constraint>:  
[CONSTRAINT constraint_name] PRIMARY KEY (column_name,...) NOT ENFORCED  
  
<metadata_column_definition>:  
column_name column_type METADATA [ FROM metadata_key ] [ VIRTUAL ]  
  
<computed_column_definition>:  
column_name AS computed_column_expression [COMMENT column_comment]  
  
<watermark_definition>:  
WATERMARK FOR rowtime_column_name AS watermark_strategy_expression  
  
<source_table>:  
[catalog_name.][db_name.]table_name  
  
<like_options>:  
{  
    { INCLUDING | EXCLUDING } { ALL | CONSTRAINTS | PARTITIONS }  
    | { INCLUDING | EXCLUDING | OVERWRITING } { GENERATED | OPTIONS | WATERMARKS }  
}[...]
```

8. cdc 连接器

8.1. cdc 的通用概念

CDC, Change Data Capture, 变更数据获取的简称, 使用 CDC 我们可以从数据库中获取已提交的更改并将这些更改发送到下游, 供下游使用。这些变更可以包括 INSERT, DELETE, UPDATE 等, 用户可以在以下的场景下使用 CDC:

- 使用 flink sql 进行数据同步, 将数据从一个地方同步到其他地方, 比如从 mysql 到 doris
- 可以在源数据库上实时的物化一个聚合视图
- 因为只是增量同步, 所以可以实时的低延迟的同步数据

市面上有大量的 cdc 工具, 知名度最高的如: [canal](#), [Debezium](#)

如 canal, 可以捕获到 mysql 的 binlog, 形成 cdc 日志形式的输出供给下游使用;

8.2. flinksql 对 cdc 的支持

flinksql 底层的设计, 就天然支持 cdc

- 在 flinksql 中, cdc 数据几乎等价于 changelog:
cdc 的数据格式, 是对变化数据的表达模式; changelog 也如是;
- flinksql 对 cdc 的支持, 核心就在对 record 的 rowkind (+I/-U/+U/-D) 进行适配;

flinksql 中操作 cdc 数据, 通过 cdc 连接器即可

- 不需要开发者做太多的底层维护操作
- 使用各种 cdc connector 就可以获得输入的 cdc 模式数据并转成 changelog 流 (映射成表)
- 对 cdc 表进行各种 sql 查询时, flink 也已做好了适配
- 并且还可以把计算结果的 cdc 数据, 通过 cdc connector 输出到目标存储系统

flink 的 cdc connector, 在核心包中是没有集成的, 需要额外引入依赖;

目前, 官方有一个专门的开源项目, 提供了若干种常用的 cdc-connector;

项目地址:

[ververica/flink-cdc-connectors: CDC Connectors for Apache Flink® \(github.com\)](https://github.com/ververica/flink-cdc-connectors)

8.3. mysql-cdc-connector 使用

8.3.1. mysql binlog 配置

- 修改 mysql 配置文件， /etc/my.cnf

```
server-id=1
log_bin=/var/lib/mysql/mysql-bin.log
expire_logs_days=7
binlog_format=ROW
max_binlog_size=100M
binlog_cache_size=16M
max_binlog_cache_size=256M
relay_log_recovery=1
sync_binlog=1
innodb_flush_log_at_trx_commit=1
```

(注意：先复制到 notepad++，然后转为 unix 格式，然后再粘贴到 linux)

- 配置完成后，重启 mysql 服务

```
systemctl restart mysqld
```

- 查看 binlog 是否生效

```
mysql> show variables like 'log_%';
+-----+-----+
| Variable_name          | Value   |
+-----+-----+
| log_bin                 | ON      |
| log_bin_basename        | /var/lib/mysql/mysql-bin    |
| log_bin_index           | /var/lib/mysql/mysql-bin.index |
```

- 测试观察 binlog

```
# 创建一个库
mysql> create database doitedu;

# 创建一个表
mysql> use doitedu;
```

```
mysql> create table t1(id int,name varchar(20));

# 插入一条数据
mysql> insert into t1 values(1,'zs');

# 更新一条数据
mysql> update t1 set name='bb' where id=1;

# 查看 master 正在写入的 binlog 文件
mysql> show master status;
+-----+-----+-----+-----+
| File      | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+
| mysql-bin.000001 |    772 |           |           |           |
+-----+-----+-----+-----+

# 列出当前所有的 binlog 文件
mysql> show binary logs;
+-----+-----+
| Log_name | File_size |
+-----+-----+
| mysql-bin.000001 |    772 |
+-----+-----+

# 查看指定 binlog 文件的内容
mysql> show binlog events in 'mysql-bin.000001';
mysql> show binlog events; # 或者不指定文件，则查看的是第一个 binlog 文件
# 可以清楚看到 binlog 中已经有了上述一系列 sql 操作的事件记录
+-----+-----+-----+-----+-----+-----+
| Log_name | Pos | Event_type | Server_id | End_log_pos | Info |
+-----+-----+-----+-----+-----+-----+
| mysql-bin.000001 |  4 | Format_desc |   1 |     123 | Server ver: 5.7.38-log, Binlog ver: 4 |
| mysql-bin.000001 | 123 | Previous_gtids |   1 |     154 | |
| mysql-bin.000001 | 154 | Anonymous_Gtid |   1 |     219 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS' |
| mysql-bin.000001 | 219 | Query |   1 |     322 | create database doitedu |
| mysql-bin.000001 | 322 | Anonymous_Gtid |   1 |     387 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS' |
| mysql-bin.000001 | 387 | Query |   1 |     507 | use `doitedu`; create table t1(id int,name varchar(20)) |
| mysql-bin.000001 | 507 | Anonymous_Gtid |   1 |     572 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS' |
| mysql-bin.000001 | 572 | Query |   1 |     647 | BEGIN |
| mysql-bin.000001 | 647 | Table_map |   1 |     698 | table_id: 108 (doitedu.t1) |
| mysql-bin.000001 | 698 | Write_rows |   1 |     741 | table_id: 108 flags: STMT_END_F |
| mysql-bin.000001 | 741 | Xid |   1 |     772 | COMMIT /* xid=15 */ |
| mysql-bin.000001 | 772 | Anonymous_Gtid |   1 |     837 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS' |
```

mysql-bin.000001 837 Query 1 912 BEGIN
mysql-bin.000001 912 Table_map 1 963 table_id: 108 (doitedu.t1)
mysql-bin.000001 963 Update_rows 1 1015 table_id: 108 flags: STMT_END_F
mysql-bin.000001 1015 Xid 1 1046 COMMIT /* xid=19 */
-----+-----+-----+-----+-----+-----+-----+

8.3.2. flink-mysql-cdc 依赖配置

8.3.3. flink-mysql-cdc 代码测试

测试数据

mysql 中有如下源表：

用 flink- mysql-cdc 连接器，映射该源表，并进行查询计算

完整代码

```
package cn.doitedu.flinksqldemos;

/*
 * @Author: deep as the sea
 * @Site: <a href="www.51doit.com">多易教育</a>
 * @QQ: 657270652
 * @Date: 2022/6/13
 * @Desc: 学大数据，到多易教育
 *      mysql 的 cdc 连接器使用测试
 */
public class Demo14_MysqlCdcConnector {

    public static void main(String[] args) {
```

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(1000, CheckpointingMode.EXACTLY_ONCE);
env.getCheckpointConfig().setCheckpointStorage("file:///d:/checkpoint");

StreamTableEnvironment tenv = StreamTableEnvironment.create(env);

// 建 cdc 连接器源表
tenv.executeSql("CREATE TABLE flink_score (\n" +
    "    id INT,\n" +
    "    name string,\n" +
    "    gender string,\n" +
    "    score double,\n" +
    "    PRIMARY KEY(id) NOT ENFORCED\n" +
    ") WITH (\n" +
    "    'connector' = 'mysql-cdc',\n" +
    "    'hostname' = 'doitedu',\n" +
    "    'port' = '3306',\n" +
    "    'username' = 'root',\n" +
    "    'password' = 'root',\n" +
    "    'database-name' = 'flinktest',\n" +
    "    'table-name' = 'score'\n" +
    ")");
}

// 简单查询
tenv.executeSql("select * from flink_score")/*.print0*/;

// groupby 聚合查询
tenv.executeSql("select gender,avg(score) as avg_score from flink_score group by gender")/*.print0*/;

// 建一个目标表，用来存放查询结果： 每种性别中，总分最高的前 2 个人
tenv.executeSql(
    "create table flink_rank(\n" +
        "    gender string ,\n" +
        "    name string,\n" +
        "    score_amt double,\n" +
        "    rn bigint ,\n" +
        "    primary key(gender,rn) not enforced \n" +
    ") with (\n" +
        "    'connector' = 'jdbc',\n" +
        "    'url' = 'jdbc:mysql://doitedu:3306/flinktest',\n" +
        "    'table-name' = 'score_rank',\n" +
        "    'username' = 'root',\n" +
        "    'password' = 'root' \n" +
    )");
}
```

```
        ");  
  
        // 执行复杂查询，并将结果以 upsert 模式写入 mysql  
        tenv.executeSql("insert into flink_rank \n" +  
            "SELECT\n" +  
            "    gender,\n" +  
            "    name,\n" +  
            "    score_amt,\n" +  
            "    rn\n" +  
            "from(\n" +  
            "    SELECT\n" +  
            "        gender,\n" +  
            "        name,\n" +  
            "        score_amt,\n" +  
            "        row_number() over(partition by gender order by score_amt desc) as rn\n" +  
            "    from \n" +  
            "    (\n" +  
            "        SELECT\n" +  
            "        gender,\n" +  
            "        name,\n" +  
            "        sum(score) as score_amt\n" +  
            "        from flink_score\n" +  
            "        group by gender,name\n" +  
            ") o1\n" +  
            "    ) o2\n" +  
            "where rn<=2");  
    }  
}
```

9.表与流的集成

9.1. changelogStream

动态表概念回顾

如前文第一章所述，flinksql 中的表相关概念，是持续输入，持续查询，持续数据的动态概念；

很多关系运算，如分组聚合，分组 topn，全局 topn，表关联等运算查询，在动态表的概念中，其输出结果不是一个简单的“结果记录”不断追加的状态，而是一个需要对“前序”结果进行修正的状态；

为了能向下游传递其输出结果的动态信息，flink 设计了一种 DataStream，叫做 changelogStream（改变日志流）；

现象观察

- 有如下测试数据：

```
DataType schema = DataTypes.ROW(
    DataTypes.FIELD("id", DataTypes.INT()),
    DataTypes.FIELD("event", DataTypes.STRING()),
    DataTypes.FIELD("ts", DataTypes.BIGINT())
);
Table table = tenv.fromValues(
    schema,
    row(1, "e1", 1000),
    row(1, "e1", 2000),
    row(1, "e2", 3000),
    row(1, "e3", 4000),
    row(1, "e3", 5000),
    row(1, "e2", 6000),
    row(1, "e2", 7000)
);
tenv.createTemporaryView("t",table);
```

● 执行查询 1：求每种事件的发生次数

```
tenv.executeSql("select event,count(1) as event_cnt from t group by event").print();
```

op	event	event_cnt
+I	e1	1
-U	e1	1
+U	e1	2
+I	e2	1
-U	e2	1
+U	e2	2
-U	e2	2
+U	e2	3
+I	e3	1
-U	e3	1
+U	e3	2

● 执行查询 2：求发生次数最多的前 2 种事件

```
tenv.executeSql("select event,event_cnt " +  
"from (" +  
"select event,event_cnt,row_number() over(partition by event order by event_cnt desc) as rn " +  
"from (select event,count(1) as event_cnt from t group by event) o1 " +  
") o2 " +  
"where rn<=2").print();
```

op	event	event_cnt
+I	e1	1
-D	e1	1
+I	e1	2
+I	e2	1
-D	e2	1
+I	e2	2
-D	e2	2
+I	e2	3
+I	e3	1
-D	e3	1
+I	e3	2

changelogStream 的本质

- changelogStream 依然是一个 DataStream
- 与普通 stream 不同的是，changelog 流中的数据类型是 Row
- 而在 Row 中，有一个属性 RowKind 来标识一行数据属于哪种“更新类型”

```
public final class Row implements Serializable {  
  
    private static final long serialVersionUID = 3L;  
  
    /** 行的 change 模式 (INSERT/UPDATE_BEFORE/UPDATE_AFTER/DELETE *)  
     *  
    private RowKind kind;  
  
    /** 数据存储数组，按字段顺序存储.*/  
    private final @Nullable Object[] fieldByPosition;  
  
    /** 数据存储 map，按 字段名-字段值 形式存储 */  
    private final @Nullable Map<String, Object> fieldByName;  
  
    /** 上述两种存储结构之间的转换关系映射 */  
    private final @Nullable LinkedHashMap<String, Integer> positionByName;
```

RowKind

```
public enum RowKind {  
  
    INSERT("+I", (byte) 0),  
    UPDATE_BEFORE("-U", (byte) 1),  
    UPDATE_AFTER("+U", (byte) 2),  
    DELETE("-D", (byte) 3);  
  
    private final String shortString;  
    private final byte value;  
  
    RowKind(String shortString, byte value) {  
        this.shortString = shortString;  
        this.value = value;  
    }  
}
```

9.2. 表=>流的方法一览

tenv.to		
toDataStream(Table table, Class<T> targetClass)		DataStream<T>
toDataStream(Table table)		DataStream<Row>
toDataStream(Table table, AbstractDataType<?> targetType)		DataStream<T>
toChangelogStream(Table table)		DataStream<Row>
toChangelogStream(Table table, Schema targetSchema)		DataStream<Row>
toChangelogStream(Table table, Schema targetSchema, ChangelogMode changelogMode)		DataStream<Row>
toAppendStream(Table table, Class<T> clazz)		DataStream<T>
toAppendStream(Table table, TypeInformation<T> typeInfo)		DataStream<T>
toRetractStream(Table table, Class<T> clazz)		DataStream<Tuple2<Boolean, T>>
toRetractStream(Table table, TypeInformation<T> typeInfo)		DataStream<Tuple2<Boolean, T>>

9.3. 流=>表的方法一览

tenv.createTemporaryView		
createTemporaryView(String path, DataStream<T> dataStream)		void
createTemporaryView(String path, DataStream<T> dataStream, Schema schema)		void
createTemporaryView(String path, DataStream<T> dataStream, String fields)		void
createTemporaryView(String path, DataStream<T> dataStream, Expression... fields)		void

9.4. 数据类型映射

类型映射工具： org.apache.flink.table.types.utils.TypeInfoDataTypesConverter

9.5. 表流互转代码示例

10. 表输出详解

10.1. 输出方式

利用 table 对象的 api 输出

```
table.executeInsert()  
  m executeInsert(String filePath)          TableResult  
  m executeInsert(TableDescriptor descriptor) TableResult  
  m executeInsert(String filePath, boolean overWrite) TableResult  
  m executeInsert(TableDescriptor descriptor, boolean overWrite) TableResult
```

将 table 对象转成 datastream，然后用 sink 算子输出

```
tenv.toStream()  
  m toChangelogStream(Table table)          DataStream<Row>  
  m toChangelogStream(Table table, Schema schema) DataStream<Row>  
  m toChangelogStream(Table table, Schema schema, boolean autoCreate) DataStream<Row>  
  m toDataStream(Table table)                DataStream<Row>  
  m toDataStream(Table table, Class<T> aClass) DataStream<T>  
  m toDataStream(Table table, AbstractDataType<T> type) DataStream<T>  
  m toAppendStream(Table table, Class<T> aClass) DataStream<T>  
  m toAppendStream(Table table, TypeInformation<T> type) DataStream<T>  
  m toRetractStream(Table table)             DataStream<Tuple2<Boolean, T>>  
  m toRetractStream(Table table, TypeInformation<T>) DataStream<Tuple2<Boolean, T>>
```

用 insert 语句输出

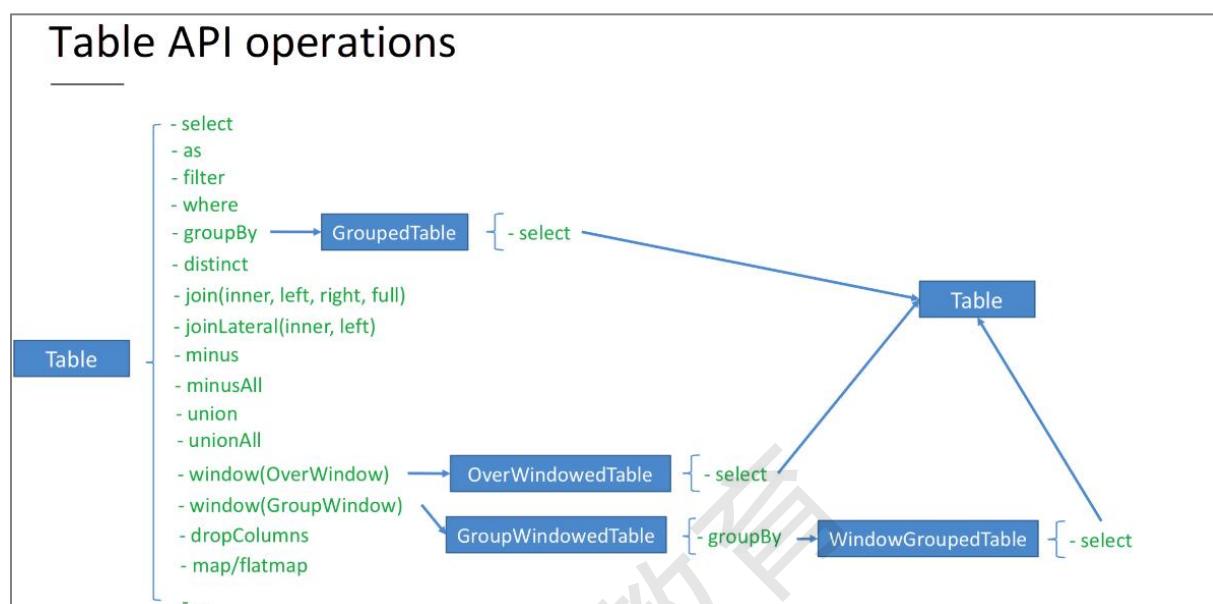
```
tenv.executeSql("insert into dest_table select ... from t1");
```

10.2. 多语句批量执行

```
// 构建一个 statementSet  
StatementSet stmtSet = tenv.createStatementSet();  
// 添加 insert  
stmtSet.addInsert("t_dest", table);  
stmtSet.addInsert(TableDescriptor.forConnector(...).build(), t1);  
stmtSet.addInsertSql("insert into t_dest select ... from t1");  
// 批量执行  
stmtSet.execute();
```

11. 表查询详解

11.1. table api 概览



11.2. 基本查询

- select

```
SELECT order_id, price + tax FROM Orders
```

- where 过滤

```
SELECT order_id, price
FROM orders
WHERE price+10 > 100;
```

- group by 聚合

```
SELECT order_id, SUM(price)
FROM orders
WHERE price > 100
```

```
GROUP BY order_id  
HAVING SUM(price) >200;
```

- with 语法

```
WITH orders_with_total AS (  
    SELECT order_id, price + tax AS total  
    FROM Orders  
)  
SELECT order_id, SUM(total)  
FROM orders_with_total  
GROUP BY order_id;
```

- distinct 去重

```
SELECT DISTINCT uid FROM Orders
```

- limit

- order by

11.3. 高阶聚合

11.3.1. 高阶聚合方式

```
group by cube(维度 1, 维度 2, 维度 3)
group by grouping sets( (维度 1, 维度 2), (维度 1, 维度 3), (维度 2), () )
group by rollup(省, 市, 区)
```

11.3.2. 语法示例

```
select
    province,
    city,
    region,
    count(distinct uid) as u_cnt
from t
group by cube(province,city,region)
```

```
select
    province,
    city,
    region,
    count(distinct uid) as u_cnt
from t
group by rollup(province,city,region)
```

```
select
    province,
    city,
    region,
    count(distinct uid) as u_cnt
from t
group by grouping sets([(province,city),(province,city,region)])
```

11.4. 时间窗口 TVF (表值函数)

从 Flink 1.13 开始，提供了时间窗口聚合计算的 TVF 语法

- 表值函数的使用约束：

1. 在窗口上做分组聚合，必须带上 window_start 和 window_end 作为分组 key
2. 在窗口上做 topN 计算，必须带上 window_start 和 window_end 作为 partition 的 key.
3. 带条件的 join，必须包含 2 个输入表的 window start 和 window end 等值条件.

```
select
```

```
.....  
from TABLE(TUMBLE(TABLE t_x, DESCRIPTOR(rt), interval '10' minute))
```

11.4.1. 支持的时间窗口类型

- 滚动窗口 (Tumble Windows)

```
TUMBLE(TABLE t_action, DESCRIPTOR(时间属性字段), INTERVAL '10' SECONDS[ 窗口长度 ] )
```

- 滑动窗口 (Hop Windows)

```
HOP(TABLE t_action, DESCRIPTOR(时间属性字段), INTERVAL '5' SECONDS[ 滑动步长 ], INTERVAL '10' SECONDS[ 窗口长度 ] )
```

- 累计窗口 (Cumulate Windows)

```
CUMULATE(TABLE t_action, DESCRIPTOR(时间属性字段), INTERVAL '5' SECONDS[ 更新步长 ], INTERVAL '10' SECONDS[ 窗口最大长度 ] )
```

- 会话窗口 (Session Windows)

暂不支持！

11.4.2. 语法示例

```
select
    window_start,
    window_end,
    channel,
    count(distinct guid) as uv
from table(
    tumble(table t_applog.descriptor(rw), interval '5' minutes)
)
group by window_start,window_end,channel
```

11.5. window 聚合

11.6. window topn

语法要义： 在 TVF 上使用 row_number() over()

代码示例，有如下数据表，其中 bidtime 被声明为了 rowtime 属性

```
-- bidtime,price,item,supplier_id
2020-04-15 08:05:00.000,4.00,C,supplier1
2020-04-15 08:07:00.000,2.00,A,supplier1
2020-04-15 08:09:00.000,5.00,D,supplier2
2020-04-15 08:11:00.000,3.00,B,supplier2
2020-04-15 08:09:00.000,5.00,D,supplier3
2020-04-15 08:11:00.000,6.00,B,supplier3
2020-04-15 08:11:00.000,6.00,B,supplier3
```

- 示例 1：

-- 10 分钟滚动窗口中的交易金额最大的前 2 笔订单

```
SELECT
```

```
*
```

```
FROM
(
SELECT
    bidtime,
    price,
    item,
    supplier_id,
    row_number() over(partition by window_start,window_end order by price desc) as rn
FROM TABLE(TUMBLE(table t_bid,descriptor(rt),interval '10' minute))
)
WHERE rn<=2
```

● 示例 2：

```
-- 10 分钟滚动窗口内交易总额最高的前两家供应商，及其交易总额和交易单数
select
  *
from
(
  select
    window_start,window_end,
    supplier_id,
    price_amt,
    bid_cnt,
    row_number() over(partition by window_start,window_end order by price_amt desc) as rn
  from (
    select
      window_start,
      window_end,
      supplier_id,
      sum(price) as price_amt,
      count(1) as bid_cnt
    from table( tumble(table t_bid,descriptor(rt),interval '10' minutes) )
    group by window_start,window_end,supplier_id
  )
)
where rn<=2
```

11.7. window join 查询

语法要义

- 在 TVF 上使用 join
- 参与 join 的两个表都需要定义时间窗口
- join 的条件中必须包含两表的 window_start 和 window_end 的等值条件

支持的 join 方式

- inner/left/right/full
- semi (即: where id in ...)
- anti (即: where id not in ...)

代码示例

```
package cn.doitedu.flinksql.demos;

/*
 * @Author: deep as the sea
 * @Site: <a href="www.51doit.com">多易教育</a>
 * @QQ: 657270652
 * @Date: 2022/6/16
 * @Desc: 学大数据, 到多易教育
 *      各种窗口 JOIN 的代码示例
 */
public class Demo17_TimeWindowJoin {

    public static void main(String[] args) {

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        StreamTableEnvironment tenv = StreamTableEnvironment.create(env);

        /**
         * 1,a,1000
         */
```

```
* 2,b,2000
* 3,c,2500
* 4,d,3000
* 5,e,12000
*/
DataStreamSource<String> s1 = env.socketTextStream("doitedu", 9998);
SingleOutputStreamOperator<Tuple3<String, String, Long>> ss1 = s1.map(s -> {
    String[] arr = s.split(",");
    return Tuple3.of(arr[0], arr[1], Long.parseLong(arr[2]));
}).returns(new TypeHint<Tuple3<String, String, Long>>() {
});

/**
 * 1,bj,1000
 * 2,sh,2000
 * 4,xa,2600
 * 5,yn,12000
 */
DataStreamSource<String> s2 = env.socketTextStream("doitedu", 9999);
SingleOutputStreamOperator<Tuple3<String, String, Long>> ss2 = s2.map(s -> {
    String[] arr = s.split(",");
    return Tuple3.of(arr[0], arr[1], Long.parseLong(arr[2]));
}).returns(new TypeHint<Tuple3<String, String, Long>>() {
});

// 创建两个表
tenv.createTemporaryView("t_left", ss1, Schema.newBuilder()
    .column("f0", DataTypes.STRING())
    .column("f1", DataTypes.STRING())
    .column("f2", DataTypes.BIGINT())
    .columnByExpression("rt", "to_timestamp_ltz(f2,3)")
    .watermark("rt", "rt - interval '0' second")
    .build());

tenv.createTemporaryView("t_right", ss2, Schema.newBuilder()
    .column("f0", DataTypes.STRING())
    .column("f1", DataTypes.STRING())
    .column("f2", DataTypes.BIGINT())
    .columnByExpression("rt", "to_timestamp_ltz(f2,3)")
    .watermark("rt", "rt - interval '0' second")
    .build());
```

```
// 各类窗口 join 示例
// INNER
tenv.executeSql(
    "SELECT  \n" +
    "  a.f0,a.f1,a.f2,b.f0,b.f1  \n" +
    "from \n" +
    "(  select * from table(tumble(table t_left,descriptor(rt),interval '10' second))  ) a\n" +
    "join \n" +
    "( select * from table(tumble(table t_right,descriptor(rt),interval '10' second))  ) b\n" +
    "on a.window_start=b.window_start and a.window_end = b.window_end and a.f0=b.f0"
)/*.print()*/;

// left / right / full outer
tenv.executeSql(
    "SELECT  \n" +
    "  a.f0,a.f1,a.f2,b.f0,b.f1  \n" +
    "from \n" +
    "(  select * from table(tumble(table t_left,descriptor(rt),interval '10' second))  ) a\n" +
    "full join \n" +
    "( select * from table(tumble(table t_right,descriptor(rt),interval '10' second))  ) b\n" +
    "on a.window_start=b.window_start and a.window_end = b.window_end and a.f0=b.f0"
)/*.print()*/;

// semi Join ==> where ... in ....
tenv.executeSql(
    "SELECT\n" +
    "  a.f0,a.f1,a.f2  \n" +
    "from \n" +
    "(  select * from table(tumble(table t_left,descriptor(rt),interval '10' second))  ) a\n" +
    "where f0 in  \n" +
    "(\n" +
    "  select f0 from \n" +
    "  (select * from table(tumble(table t_right,descriptor(rt),interval '10' second))) b\n" +
    "  where a.window_start=b.window_start and a.window_end=b.window_end \n" +
    ")"
).print();

// semi Join ==> where ... not in ....
tenv.executeSql(
    "SELECT\n" +
    "  a.f0,a.f1,a.f2  \n" +
```

```

    "from \n" +
    "(" select * from table(tumble(table t_left,descriptor(rt),interval '10' second)) ) a\n" +
    "where f0 not in \n" +
    "(\n" +
    "  select f0 from \n" +
    "  (select * from table(tumble(table t_right,descriptor(rt),interval '10' second))) b\n" +
    "  where a.window_start=b.window_start and a.window_end=b.window_end \n" +
    ")"
    ).print();
}
}

```



FlinkSQL 提供了非常丰富的 join 功能，为实现各类关联场景提供强大的功能支撑

Join 类别	别名	数据更新	Join 方式	是否 Retract	状态清理
Regular	双流 Join	左、右流触发	Inner/outer Equi-Join	Outer join 会	TTL
Interval	区间 Join	左、右流触发	Inner/outer Join 等值和时间条件	不会	where 条件时间范围
Lateral	UDTF Join	左流触发	Inner/Left Join	不会	无
Temporal	时态 Join	左流触发	Inner/Left Join	不会	TTL
Lookup	维表 Join	左流触发	Inner/Left Join	不会	基于 JVM, LRU

11.8. regular join

常规 join 的实现逻辑所在类:

`org.apache.flink.table.runtime.operators.join.stream.StreamingJoinOperator`

常规 join, flink 底层是会对两个参与 join 的输入流中的数据进行状态存储的;
所以, 随着时间的推进, 状态中的数据量会持续膨胀, 可能会导致过于庞大, 从而降低系统的整体效率;

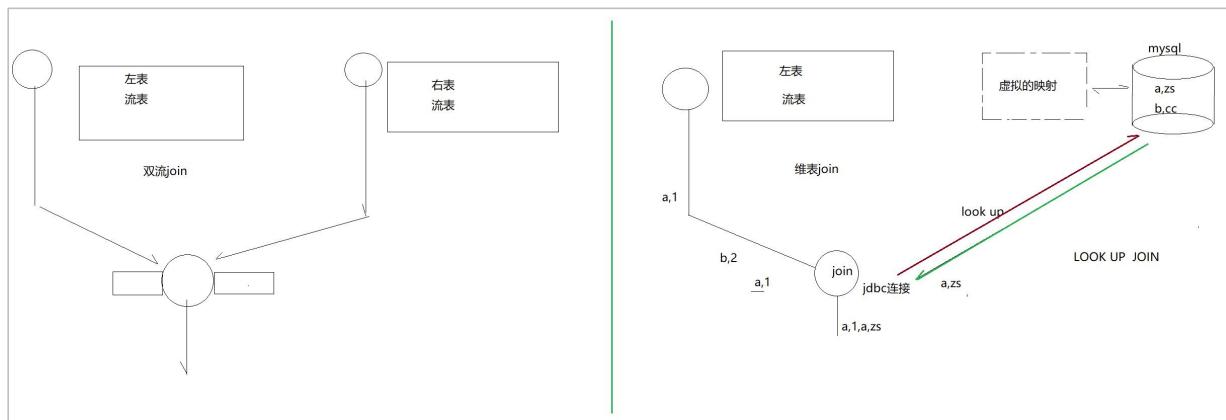
可以如何去缓解: 自己根据自己业务系统数据特性(估算能产生关联的左表数据和右表数据到达的最大时间差), 根据这个最大时间差, 去设置 ttl 时长;

```
StreamTableEnvironment tenv = StreamTableEnvironment.create(env);
// 设置 table 环境中的状态 ttl 时长
tenv.getConfig().getConfiguration().setLong("table.exec.state.ttl",60*60*1000L);
```

- inner
- left join
- right join
- full join

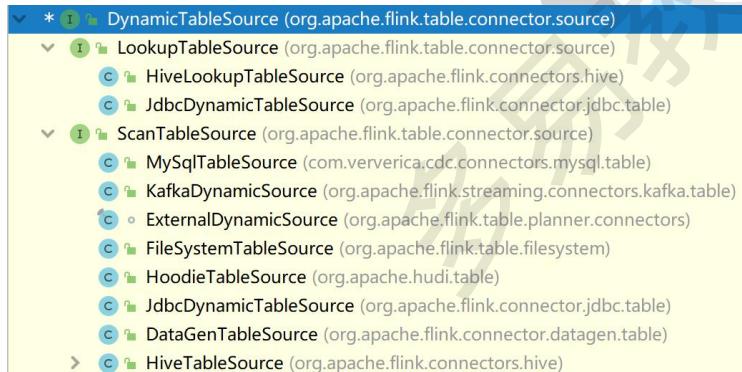
11.9. lookup join

11.9.1. Lookup join 的内在原理



Lookup join 跟其它的 join 有较大的不同，其内在原理得从源码说起，在 flinksql 中，所有的 source connector 都实现自 [DynamicTableSource](#)

而 [DynamicTableSource](#) 下有两个子接口：[LookupTableSource](#) 和 [ScanTableSource](#)



其中，[ScanTableSource](#) 是用的最多的常规 TableSource，它会持续、完整读取源表形成 flink 中的核心数据抽象：“数据流”；

而 [LookupTableSource](#)，则并不对源表持续、完整读取，而是在需要的时候，才根据一个（或多个）查询 key，去临时性地查询源表得到一条（或多条）数据；

lookup join 为了提高性能，lookup 的连接器会将查询过的维表数据进行缓存（默认未开启此机制），可以通过参数开启，比如 jdbc-connector 的 lookup 模式下，有如下参数：

- `lookup.cache.max-rows = (none)` 未开启
- `lookup.cache.ttl = (none)` ttl 缓存清除的时长

以 JdbcDynamicTableSource 为例

```
public class JdbcDynamicTableSource
    implements ScanTableSource,
              LookupTableSource,
              SupportsProjectionPushDown,
              SupportsLimitPushDown {
```

它实现了上述两种接口，因而它是两种读取模式的混合封装体

因而，它也实现了上述两个接口中各自的一个重要方法：

- getLookupRuntimeProvider
- getScanRuntimeProvider

```
① JdbcDynamicTableSource.java <
  4 个语法
69
70 ② @Override
71  public LookupRuntimeProvider getLookupRuntimeProvider(LookupContext context) {
72      // JDBC only support non-nested look up keys
73      String[] keyNames = new String[context.getKeys().length];
74      for (int i = 0; i < keyNames.length; i++) {
75          int[] innerKeyArr = context.getKeys()[i];
76          Preconditions.checkArgument(
77              condition: innerKeyArr.length == 1, errorMessage: "JDBC only support non-nested look up keys");
78          keyNames[i] = physicalSchema.getFieldNames()[innerKeyArr[0]];
79      }
80      final RowType rowType = (RowType) physicalSchema.toRowDataType().getLogicalType();
81
82      return TableFunctionProvider.of(
83          new JdbcRowDataLookupFunction(
84              options,
85              lookupOptions,
86              physicalSchema.getFieldNames(),
87              physicalSchema.getFieldDataTypes(),
88              keyNames,
89              rowType));
90
91  @Override
92  public ScanRuntimeProvider getScanRuntimeProvider(ScanContext runtimeProviderContext) {
93      final JdbcRowDataInputFormat.Builder builder =
94          JdbcRowDataInputFormat.builder()
95              .setDrivername(options.getDriverName())
96              .setDBUrl(options.getDbURL())
97              .setUsername(options.getUsername().orElse(null))
98              .setPassword(options.getPassword().orElse(null))
99              .setAutoCommit(readOptions.getAutoCommit());
100
101     if (readOptions.getFetchSize() != 0) {
102         builder.setFetchSize(readOptions.getFetchSize());
103     }
104     final JdbcDialect dialect = options.getDialect();
105     String query =
106         dialect.getSelectFromStatement()
```

可以看到, JdbcDynamicTableSource提供了lookupProvider和scanProvider

对于 lookupRuntimeProvider 来说，最重要的是其中的： JdbcRowDataLookupFunction

JdbcRowDataLookupFunction

```
// LookupFunction 中最重要的方法就是 eval
public void eval(Object... keys) {
    RowData keyRow = GenericRowData.of(keys);
```

```
// 对于传入的 keys, 先从缓存中获取要查询的数据
if (cache != null) {
    List<RowData> cachedRows = cache.getIfPresent(keyRow);
    if (cachedRows != null) {
        for (RowData cachedRow : cachedRows) {
            // 如果缓存中拿到了数据, 则直接输出
            collect(cachedRow);
        }
        return;
    }
}

// 否则, 用 jdbc 去查询
for (int retry = 0; retry <= maxRetryTimes; retry++) {
    try {
        // 构建 jdbc 查询语句 statement
        statement.clearParameters();
        statement = lookupKeyRowConverter.toExternal(keyRow, statement);
        // 执行查询语句, 并获得 resultset
        try (ResultSet resultSet = statement.executeQuery()) {
            if (cache == null) {
                while (resultSet.next()) {
                    collect(jdbcRowConverter.toInternal(resultSet));
                }
            } else {
                ArrayList<RowData> rows = new ArrayList<>();
                // 迭代 resultset
                while (resultSet.next()) {
                    // 转成内部数据类型 RowData
                    RowData row = jdbcRowConverter.toInternal(resultSet);
                    // 将数据装入一个 list 后一次性输出
                    rows.add(row);
                    collect(row);
                }
                // 并将查到的数据, 放入缓存
                rows.trimToSize();
                cache.put(keyRow, rows);
            }
        }
    }
}
```

11.9.2. Lookup join 的代码示例

```
package cn.doitedu.flinksql.demos;

/*
 * @Author: deep as the sea
 * @Site: <a href="www.51doit.com">多易教育</a>
 * @QQ: 657270652
 * @Date: 2022/6/16
 * @Desc: 学大数据，到多易教育
 * 常规 join 示例
 * 常规 join 的底层实现，是通过在用状态来缓存两表数据实现的
 * 所以，状态体积可能持续膨胀，为了安全起见，可以设置状态的 ttl 时长，来控制状态的体积上限
 */
public class Demo18_LookupJoin {
    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        StreamTableEnvironment tenv = StreamTableEnvironment.create(env);
        // 设置 table 环境中的状态 ttl 时长
        tenv.getConfig().getConfiguration().setLong("table.exec.state.ttl", 60 * 60 * 1000L);

        /*
         * 1,a
         * 2,b
         * 3,c
         * 4,d
         * 5,e
         */
        DataStreamSource<String> s1 = env.socketTextStream("doitedu", 9998);
        SingleOutputStreamOperator<Tuple2<Integer, String>> ss1 = s1.map(s -> {
            String[] arr = s.split(",");
            return Tuple2.of(Integer.parseInt(arr[0]), arr[1]);
        }).returns(new TypeHint<Tuple2<Integer, String>>() {
    });

        // 创建主表（需要声明处理时间属性字段）
    }
}
```

```
tenv.createTemporaryView("a", ss1, Schema.newBuilder()
    .column("f0", DataTypes.INT())
    .column("f1", DataTypes.STRING())
    .columnByExpression("pt", "proctime()") // 定义处理时间属性字段
    .build());

// 创建 lookup 维表 (jdbc connector 表)
tenv.executeSql(
    "create table b(  \n" +
    "    id int ,\n" +
    "    name string,\n" +
    "    gender STRING,\n" +
    "    primary key(id) not enforced  \n" +
    ") with (\n" +
    "    'connector' = 'jdbc',\n" +
    "    'url' = 'jdbc:mysql://doitedu:3306/flinktest',\n" +
    "    'table-name' = 'stu2',\n" +
    "    'username' = 'root',\n" +
    "    'password' = 'root'\n" +
    ")"
);

// lookup join 查询
tenv.executeSql("select a.* ,c.*  from  a  JOIN  b FOR SYSTEM_TIME AS OF a.pt AS c  \n" +
    "    ON a.f0 = c.id").print();
env.execute();
}

}
```

11.10. interval join

```
SELECT *FROM Orders o,Shipments s
WHERE o.id = s.order_id
AND o.order_time BETWEEN s.ship_time - INTERVAL '4' HOUR AND s.ship_time
```

- 时间区间条件的可用语法:
 - ltime = rtime
 - ltime >= rtime AND ltime < rtime + INTERVAL '10' MINUTE
 - ltime BETWEEN rtime - INTERVAL '10' SECOND AND rtime + INTERVAL '5' SECOND
-
- 使用场景举例:

比如广告曝光流 和 广告观看事件流的 join

11.11. temporal join (时态 join/版本 join)

要义： 左表的数据永远去关联右表数据的对应时间上的最新版本

-- 有如下交易订单表（订单 id, 金额, 货币, 时间）

1,88,e,1000

2,88,e,2000

3,68,e,3000

-- 有如下汇率表（货币, 汇率, 更新时间）

e,1.0,1000

e,2.0,3000

-- temporal join 的结果如下

1,88,e,1000 ,1.0

2,88,e,2000 ,1.0

3,68,e,3000 ,2.0

```
-- 创建表 orders.
-- append-only 表
CREATE TABLE orders (
    order_id    STRING,
    price       DECIMAL(32,2),
    currency    STRING,
    order_time  TIMESTAMP(3),
    WATERMARK FOR order_time AS order_time
) WITH /* ... */;

-- 创建汇率表（典型的 versioned 动态表，带时间属性字段）.
-- 比如，从 cdc 过来的表
CREATE TABLE currency_rates (
    currency STRING,
    conversion_rate DECIMAL(32, 2),
    update_time TIMESTAMP(3) METADATA FROM `values.source.timestamp` VIRTUAL,
    WATERMARK FOR update_time AS update_time,
    PRIMARY KEY(currency) NOT ENFORCED
) WITH (
    'connector' = 'kafka',
    'value.format' = 'debezium-json',
    /* ... */
);

SELECT
    order_id,
    price,
    currency,
    conversion_rate,
    order_time
FROM orders
LEFT JOIN currency_rates FOR SYSTEM_TIME AS OF orders.order_time
ON orders.currency = currency_rates.currency;

order_id  price  currency  conversion_rate  order_time
-----  -----  -----  -----  -----
o_001    11.11  EUR        1.14          12:00:00
o_002    12.51  EUR        1.10          12:06:00
```

11.12. over 窗口聚合

```
row_number() over ( )
```

flinksql 中，over 聚合时，指定聚合数据区间有两种方式

- 方式 1，带时间设定区间

```
RANGE BETWEEN INTERVAL '30' MINUTE PRECEDING AND CURRENT ROW
```

- 方式 2，按行设定区间

```
ROWS BETWEEN 10 PRECEDING AND CURRENT ROW
```

```
SELECT order_id, order_time, amount,  
       SUM(amount) OVER (   
           PARTITION BY product  
           ORDER BY order_time  
           RANGE BETWEEN INTERVAL '1' HOUR PRECEDING AND CURRENT ROW  
       ) AS one_hour_prod_amount_sum  
FROM Orders
```

over window 可以单独定义并重复使用，从而简化代码

```
SELECT order_id, order_time, amount,  
       SUM(amount) OVER w AS sum_amount,  
       AVG(amount) OVER w AS avg_amount  
FROM Orders  
WINDOW w AS (   
    PARTITION BY product  
    ORDER BY order_time  
    RANGE BETWEEN INTERVAL '1' HOUR PRECEDING AND CURRENT ROW)
```

12. 内置函数及自定义函数

12.1. ScalaFunction

标量函数

特点： 每次只接收一行的数据，输出结果也是 1 行 1 列

典型的标量函数如： upper(str), lower(str), abs(salary)

12.2. TableFunction

表生成函数

特点：运行时每接收一行数据（一个或多个字段），能产出多行、多列的结果

典型的如： explode(), unnest()

12.3. AggregateFunction

聚合函数

特点：对输入的数据行（一组）进行持续的聚合，最终对每组数据输出一行（多列）结果

典型的如： sum(), max()

12.4. TableAggregateFunction

聚合函数

特点：对输入的数据行（一组）进行持续的聚合，最终对每组数据输出一行或多行（多列）结果

13. FlinkSql 与 hive 集成

Flinksql 与 hive 有两个整合角度

角度 1：元数据整合

角度 2：将 flink 做为 hive 的执行引擎

13.1. idea 中整合 hive catalog

准备 hive 配置文件

准备一个 `hive-site.xml` 配置文件（可放在任何本地目录），至少包含如下内容

```
<configuration>
  <property>
    <name>hive.metastore.uris</name>
    <value>thrift://doitedu:9083</value>
  </property>
</configuration>
```

添加依赖

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-sql-connector-hive-3.1.2_2.11</artifactId>
  <version>1.14.4</version>
  <!--<scope>provided</scope>-->
</dependency>

<dependency>
  <groupId>org.apache.hive</groupId>
  <artifactId>hive-exec</artifactId>
  <version>3.1.3</version>
  <exclusions>
    <exclusion>
      <groupId>org.apache.calcite.avatica</groupId>
      <artifactId>avatica</artifactId>
    </exclusion>
    <exclusion>
```

```
<groupId>org.apache.calcite</groupId>
<artifactId>*</artifactId>
</exclusion>
<exclusion>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>*</artifactId>
</exclusion>
</exclusions>
</dependency>
```

在代码中注册 hive catalog

```
String catalogTypename = "hive";
String defaultDatabase = "default";
String hiveConfDir = "D:/hiveconf/";

// 创建和注册 hive catalog
HiveCatalog hive = new HiveCatalog(catalogTypeName, defaultDatabase, hiveConfDir);
tenv.registerCatalog("myhive", hive);

// set the HiveCatalog as the current catalog of the session
tenv.useCatalog("myhive");

// 打印当前 env 中所有的 catalog
tenv.executeSql("show catalogs").print();

// 查询 hive 中的表
tenv.executeSql("select * from t2").print();
```

13.2. sql-client 中整合 hive catalog

添加依赖

在 flink 安装目录中，创建一个文件夹，如 extlib；

放入所需的依赖：

- ✓ flink-sql-connector-hive-3.1.2_2.11-1.14.4.jar (hive 整合所需)
- ✓ flink-sql-connector-kafka_2.11-1.14.4.jar (示例中需要用到 kafka 连接器)
- ✓ flink-shaded-hadoop-3-uber-3.1.1.7.2.9.0-173-9.0.jar (与 hadoop 整合所需)
- ✓ flink-csv-1.14.3.jar (支持 csv 格式所需)
- ✓ flink-json-1.14.3.jar (支持 json 格式所需)

启动 sql-client

```
[root@doitedu flink-1.14.3]# bin/sql-client.sh -l extlib/
```

注册 hive catalog

```
Flink SQL> create catalog myhive with ('type' = 'hive', 'hive-conf-dir' = '/opt/apps/hive-3.1.2/conf');
```

[INFO] Execute statement succeed.

```
Flink SQL> use catalog myhive;
```

[INFO] Execute statement succeed.

```
Flink SQL> load module hive;
```

[INFO] Execute statement succeed.

```
Flink SQL> use modules hive,core;
```

[INFO] Execute statement succeed.

```
Flink SQL> set table.sql-dialect=hive;
```

[INFO] Session property has been set.

查看 catalog 中的表

```
Flink SQL> show tables;
```

```
+-----+
```

```
| table name |
```

```
+-----+
```

```
|      t |
```

```
|     t2 |
```

```
+-----+
```

可以看到 hive 中的表；

创建 flink 表 (如： kafka connector 表)

```
CREATE TABLE KafkaTable (
    `meta_time` TIMESTAMP(3) METADATA FROM 'timestamp',
    `partition` BIGINT METADATA VIRTUAL,
    `offset` BIGINT METADATA VIRTUAL,
    `inKey_k1` BIGINT,
    `inKey_k2` STRING,
    `guid` BIGINT,
    `eventId` STRING,
    `k1` STRING,
    `k2` STRING,
    `eventTime` BIGINT,
    `headers` MAP<STRING,BYTES> METADATA FROM 'headers'
) WITH (
    'connector' = 'kafka',
    'topic' = 'abc',
    'properties.bootstrap.servers' = 'doitedu:9092',
    'properties.group.id' = 'testGroup',
    'scan.startup.mode' = 'earliest-offset',
    'key.format'='json',
    'key.json.ignore-parse-errors' = 'true',
    'key.fields'='inKey_k1;inKey_k2',
    'key.fields-prefix'='inKey_',
    'value.format'='json',
    'value.json.fail-on-missing-field'='false',
    'value.fields-include' = 'EXCEPT_KEY'
)
```

查询表

Flink SQL> show tables;

```
+-----+  
| table name |  
+-----+  
| kafkatable |  
|     t |  
|    t2 |  
+-----+
```

Flink SQL> select * from kafkatable;



meta_time	partition	offset	inKey_k1
2022-06-05 14:58:42.881	0	0	1
2022-06-05 15:22:55.892	0	1	1

14. SqlClient 使用

14.1. 基本介绍

sql-client 是 flink 安装包中自带的一个命令行工具，用于快捷方便地进行 sql 操作

注意： 首先要启动一个 flink session 集群 (standalone, on yarn)

- 命令核心参数：

-f：指定初始化 sql 脚本文件

-l：指定要添加的外部 jar 包所在的文件夹（来加载文件夹中所有的依赖 jar）

-j：指定一个 jar 包文件路径来加载这个 jar 包依赖

-s：指定要连接的 flink session 集群

```
[root@doitedu flink-1.14.3]# bin/sql-client.sh
```



```
Flink SQL Client BETA
Welcome! Enter 'HELP;' to list all available commands. 'QUIT;' to exit.

Flink SQL> show databases;
+-----+
| database name |
+-----+
| default_database |
+-----+
```

14.2. 完整示例

- 需求背景:

从 kafka 中读取一个 topic 的数据

然后统计每 5 分钟的去重用户数

并且，把结果写入 mysql

- 操作步骤

1 需要额外的依赖 jar 包

- flink-connector-jdbc_2.12-1.14.4.jar
- flink-csv-1.14.3.jar
- flink-json-1.14.3.jar
- flink-sql-connector-kafka_2.12-1.14.4.jar

可以把这些 jar 包直接放在 flink 安装目录的 lib 中，当然，也可以放在任意一个自定义的目录

2 启动 sql-client

```
[root@doitedu flink-1.14.3]# bin/sql-client.sh -l lib/
```

3 需求开发

- 建数据源表（kafka 连接器表）

```
CREATE TABLE events (
    account      STRING ,
    appId        STRING ,
    appVersion   STRING ,
    carrier      STRING ,
    deviceId     STRING ,
    deviceType   STRING ,
    eventId      STRING ,
    ip           STRING ,
    latitude     DOUBLE ,
    longitude    DOUBLE ,
```

```
netType          STRING   ,
osName          STRING   ,
osVersion        STRING   ,
properties       MAP<STRING,STRING>  ,
releaseChannel  STRING   ,
resolution       STRING   ,
sessionId        STRING   ,
`timeStamp`      BIGINT   ,
rt as to_timestamp_ltz(`timeStamp`,3),
watermark for rt as rt - interval '0' second
) WITH (
  'connector' = 'kafka',
  'topic' = 'doit-events',
  'properties.bootstrap.servers' = 'doitedu:9092',
  'properties.group.id' = 'testGroup',
  'scan.startup.mode' = 'earliest-offset',
  'format' = 'json'
)
```

● 目标表的创建

```
CREATE TABLE uv_report (
  window_start timestamp(3),
  window_end  timestamp(3),
  uv bigint
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://doitedu:3306/doitedu',
  'table-name' = 'uv_report',
  'username' = 'root',
  'password'= 'root'
);
```

● 查询 sql 开发

```
INSERT INTO uv_report
SELECT
  window_start,
  window_end,
  count(distinct deviceId) as uv
```

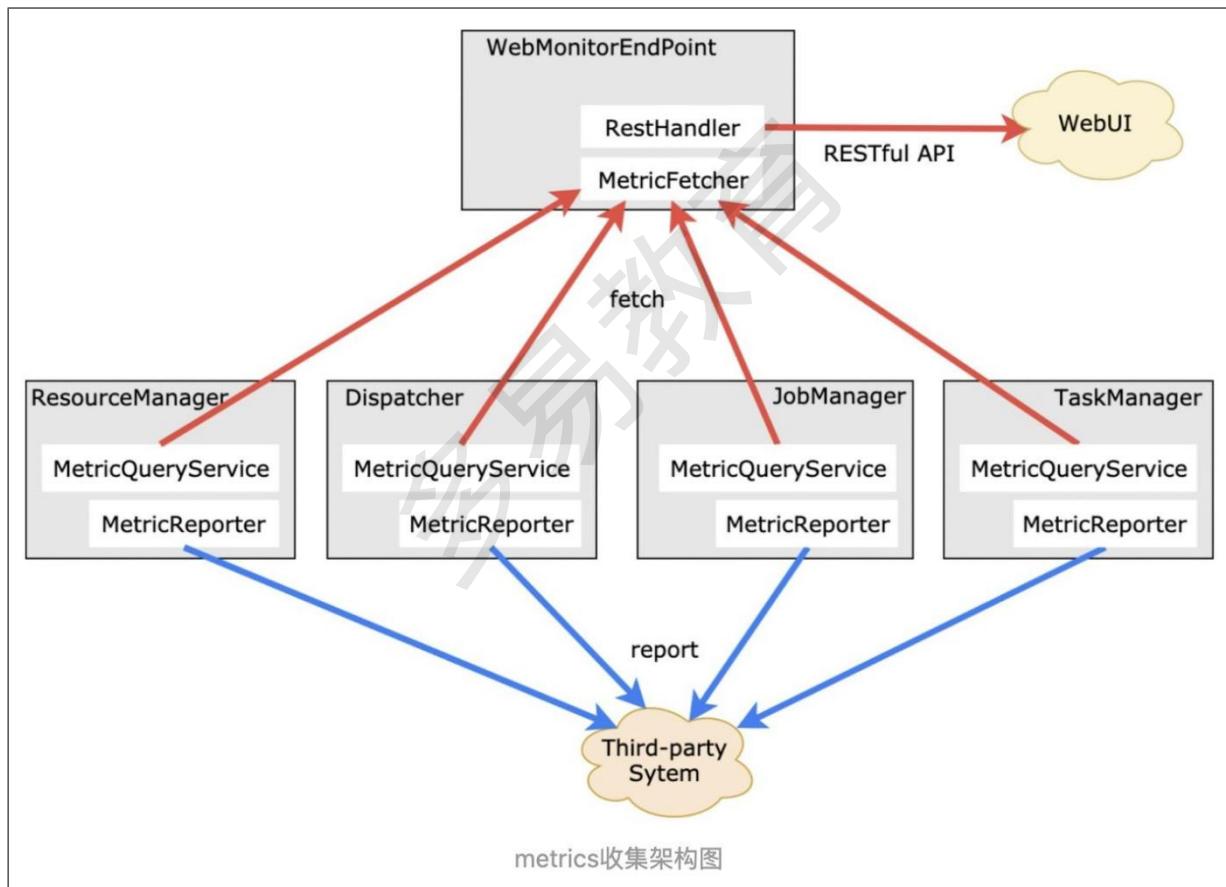
```
FROM TABLE(TUMBLE(table_events,descriptor(rt),interval '1' minute))
group by window_start,window_end
```



15. flink 监控指标 Metric 体系

在 flink 任务运行的过程中，用户通常想知道任务运行的一些基本指标，比如吞吐量、内存和 cpu 使用情况、checkpoint 稳定性等等。而通过 flink metrics 这些指标都可以轻而易举地获取到，避免任务的运行处于黑盒状态，通过分析这些指标，可以更好的调整任务的资源、定位遇到的问题、对任务进行监控；

Metrics 可以暴露给外部系统，通过在 flink 配置文件 conf/flink-conf.yaml 配置即可，flink 原生已经支持了很多 reporter，如 JMX、InfluxDB、Prometheus 等等，同时也支持自定义 reporter；



metrics 的收集有上图所示的两条线路。一种是通过定期 fetch 向 flink 的不同结点或组件查询 metrics，并汇集到一个中心结点并提供 RESTful API，如常用的 WebUI 即是如此。另一种方式是通过 MetricsReporter 分别将各个结点或组件的 metrics 上报到第三方系统，在第三方系统中自定义的分析和监控。两种方式的主要区别在于是否有一个中心结点对 metrics 进行汇总。一般来说，在常用的 flink

任务分析或监控系统中更推荐使用第二种方式，根据使用经验来说 MetricsReporter 上报的方式更加稳定，且可以灵活的使用第三方的存储和分析能力。而 RESTful API 的形式通常用于开发场景中查看 WebUI，使用中偶尔会出现超时或者是 metrics 延迟的情况，主要原因可能包括中心结点内存使用较高或挂掉等等。

flink 提供了如下统计器，来方便用户自定义各类自己的状态度量

- counter
- gauge
- histogram
- meter

15.1. Counter

Counters 通常用来计数，可以通过 inc 或 dec 方法来对计数值进行增加或减少。举例：numRecordsIn、numRecordsOut 分别为 task 读入和输出的数据个数。

```
public class MyMapper extends RichMapFunction<String, String> {  
    private transient Counter counter;  
  
    @Override  
    public void open(Configuration config) {  
        this.counter = getRuntimeContext()  
            .getMetricGroup()  
            .counter("myCustomCounter", new CustomCounter());  
    }  
  
    @Override  
    public String map(String value) throws Exception {  
        this.counter.inc();  
        return value;  
    }  
}
```

15.2. Guage

Gauges 可以根据需要提供各种类型的值，对值的类型没有限制。举例：CPU Load、JVM Heap Used，该类型使用较多。

A Gauge provides a value of any type on demand. (任意类型，不像 counter)

```
public class MyMapper extends RichMapFunction<String, String> {
    private transient int valueToExpose = 0;

    @Override
    public void open(Configuration config) {
        getRuntimeContext()
            .getMetricGroup()
            .gauge("MyGauge", new Gauge<Integer>() {
                @Override
                public Integer getValue() {
                    return valueToExpose;
                }
            });
    }

    @Override
    public String map(String value) throws Exception {
        valueToExpose++;
        return value;
    }
}
```

15.3. Histogram

直方图，用于度量值的统计结果，如平均值、最大值，以及分布情况等。

```
public class MyMapper extends RichMapFunction<Long, Long> {
    private transient Histogram histogram;

    @Override
    public void open(Configuration config) {
```

```
this.histogram = getRuntimeContext()  
    .getMetricGroup()  
    .histogram("myHistogram", new MyHistogram());  
}  
  
@Override  
public Long map(Long value) throws Exception {  
    this.histogram.update(value);  
    return value;  
}  
}
```

第三方扩展直方图使用示例

- 添加依赖

```
<dependency>  
    <groupId>org.apache.flink</groupId>  
    <artifactId>flink-metrics-dropwizard</artifactId>  
    <version>1.14.4</version>  
</dependency>
```

- 使用示例

```
public class MyMapper extends RichMapFunction<Long, Long> {  
    private transient Histogram histogram;  
  
    @Override  
    public void open(Configuration config) {  
        com.codahale.metrics.Histogram dropwizardHistogram =  
            new com.codahale.metrics.Histogram(new SlidingWindowReservoir(500));  
  
        this.histogram = getRuntimeContext()  
            .getMetricGroup()  
            .histogram("myHistogram", new DropwizardHistogramWrapper(dropwizardHistogram));  
    }  
  
    @Override  
    public Long map(Long value) throws Exception {  
        this.histogram.update(value);  
        return value;  
    }  
}
```

15.4. Meter

通常用来度量平均吞吐量。如 numBytesOutPerSecond 表示 task 每秒钟输出字节数。

- 自己实现一个 Meter

```
public class MyMapper extends RichMapFunction<Long, Long> {  
    private transient Meter meter;  
  
    @Override  
    public void open(Configuration config) {  
        this.meter = getRuntimeContext()  
            .getMetricGroup()  
            .meter("myMeter", new MyMeter());  
    }  
  
    @Override  
    public Long map(Long value) throws Exception {  
        this.meter.markEvent();  
        return value;  
    }  
}
```

```
public class MyMeter implements Meter {  
    int cnt = 0;  
    int sum = 0;  
  
    @Override  
    public void markEvent() {  
        sum += RandomUtils.nextInt(100, 200);  
        cnt++;  
    }  
  
    @Override  
    public void markEvent(long l) {  
  
        sum += l;  
        cnt++;  
    }  
  
    @Override  
    public double getRate() {
```

```
    return sum / cnt;  
}
```

```
@Override  
public long getCount() {  
    return cnt;  
}  
}
```

● 使用第三方 Meter

Flink 提供了可使用 Codahale / DropWizard 仪表的包装器。
需要添加依赖（上节中已有）

```
<dependency>  
    <groupId>org.apache.flink</groupId>  
    <artifactId>flink-metrics-dropwizard</artifactId>  
    <version>1.14.4</version>  
</dependency>
```

使用示例

```
public class MyMapper extends RichMapFunction<Long, Long> {  
    private transient Meter meter;  
  
    @Override  
    public void open(Configuration config) {  
        com.codahale.metrics.Meter dropwizardMeter = new com.codahale.metrics.Meter();  
  
        this.meter = getRuntimeContext()  
            .getMetricGroup()  
            .meter("myMeter", new DropwizardMeterWrapper(dropwizardMeter));  
    }  
  
    @Override  
    public Long map(Long value) throws Exception {  
        this.meter.markEvent();  
        return value;  
    }  
}
```

16. 监控套装 Prometheus+Grafana

16.1. Prometheus 介绍

Prometheus 是一套开源的系统监控报警框架。它受启发于 Google 的 Borgmon 监控系统，由工作在 SoundCloud 的前 google 员工在 2012 年创建，作为社区开源项目进行开发，并于 2015 年正式发布。

2016 年，Prometheus 正式加入 Cloud Native Computing Foundation (CNCF) 基金会的项目，成为受欢迎度仅次于 Kubernetes 的项目。2017 年底发布了基于全新存储层的 2.0 版本，能更好地与容器平台、云平台配合。和展示工具 Grafana 有很好的集成性。

prometheus 项目地址：<https://github.com/prometheus/prometheus>
pushgateway 项目地址：<https://github.com/prometheus/pushgateway>

16.2. Grafana 介绍

Grafana 是一款用 Go 语言开发的开源数据可视化工具，可以做数据监控和数据统计，带有告警功能。
项目地址：<https://github.com/grafana/grafana>

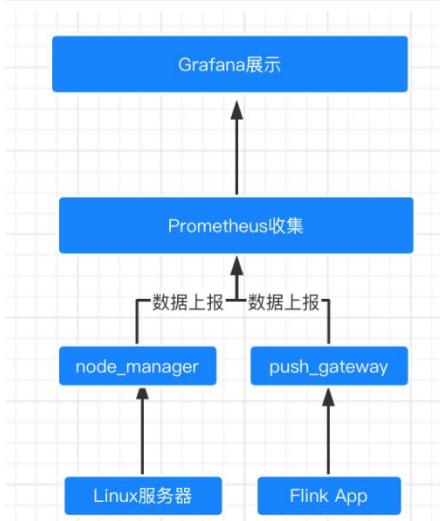




16.3. 监控体系搭建完整示例

prometheus+grafana 监控体系的要点

- Flink App : 需要监控的数据来源
- Prometheus : 收集数据
- nodeExporter : Prometheus 生态中的组件, nodeExport 负责监控运行机器的状态
- Pushgateway: Prometheus 生态中的组件, pushGateway 服务收集 Flink 的指标
- Grafana : 进行可视化展示



16.3.1. Flink 整合配置

(1) 拷贝 flink 的 prometheus 插件 jar 包到 lib

```
cd ${FLINK_HOME}/plugins/metrics-prometheus/  
cp flink-metrics-prometheus-1.14.3.jar /opt/apps/flink-1.14.3/lib/
```

(2) 修改 flink 配置文件

```
cd ${FLINK_HOME}/conf/  
vi flink-conf.yaml
```

```
##### 与 Prometheus 集成配置 #####  
metrics.reporter.promgateway.class:  
org.apache.flink.metrics.prometheus.PrometheusPushGatewayReporter  
# PushGateway 的主机名与端口号  
metrics.reporter.promgateway.host: doitedu  
metrics.reporter.promgateway.port: 9091  
# Flink metric 在前端展示的标签（前缀）与随机后缀  
metrics.reporter.promgateway.jobName: doitedu-flink  
metrics.reporter.promgateway.randomJobNameSuffix: true  
metrics.reporter.promgateway.deleteOnShutdown: false  
metrics.reporter.promgateway.interval: 30 SECONDS
```

16.3.2. Prometheus 安装和配置

(1) pushgateway 安装

```
下载: https://prometheus.io/download/  
解压: tar -zvxf pushgateway-1.4.3.linux-amd64.tar.gz  
启动: ./pushgateway  
访问: http://doitedu:9091/
```

← → ⌛ 🔍 ⚠ 不安全 | doitedu:9091/#

Pushgateway Metrics Status Help

Runtime Information

Started	2022-06-14 21:15:52.809671493 +0800 CST m=+0.024767189
---------	--------------------------------------------------------

Build Information

branch	HEAD
buildDate	20220530-19:02:00
buildUser	root@75e397dd33fe
goVersion	go1.18.2
revision	f9dc1c8664050edbc75916c3664be1df595a1958
version	1.4.3

(2) node_exporter 安裝

下載: <https://prometheus.io/download/>
 解壓: tar -zxvf node_exporter-1.3.1.linux-amd64.tar.gz
 啓動: ./node_exporter
 訪問: http://doitedu:9100/metrics

← → ⌛ ⌂ ⚠ 不安全 | doitedu:9100/metrics

```
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0
go_gc_duration_seconds{quantile="1"} 0
go_gc_duration_seconds_sum 0
go_gc_duration_seconds_count 0
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 8
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="gol.17.3"} 1
```

(3) prometheus 安裝

下載: <https://prometheus.io/download/>

解壓: tar -zxvf prometheus-2.36.1.linux-amd64.targz

編輯 prometheus.yml

```
scrape_configs:
  - job_name: "prometheus"
    static_configs:
```

```
- targets: ["doitedu:9090"]

- job_name: "linux"
  static_configs:
    - targets: ["doitedu:9100"]
      labels:
        instance: "doitedu_linux"

- job_name: "pushgateway"
  static_configs:
    - targets: ["doitedu:9091"]
      labels:
        instance: "pushgateway"
```

- 启动: ./prometheus --config.file=prometheus.yml
- 访问: http://doitedu:9090/targets

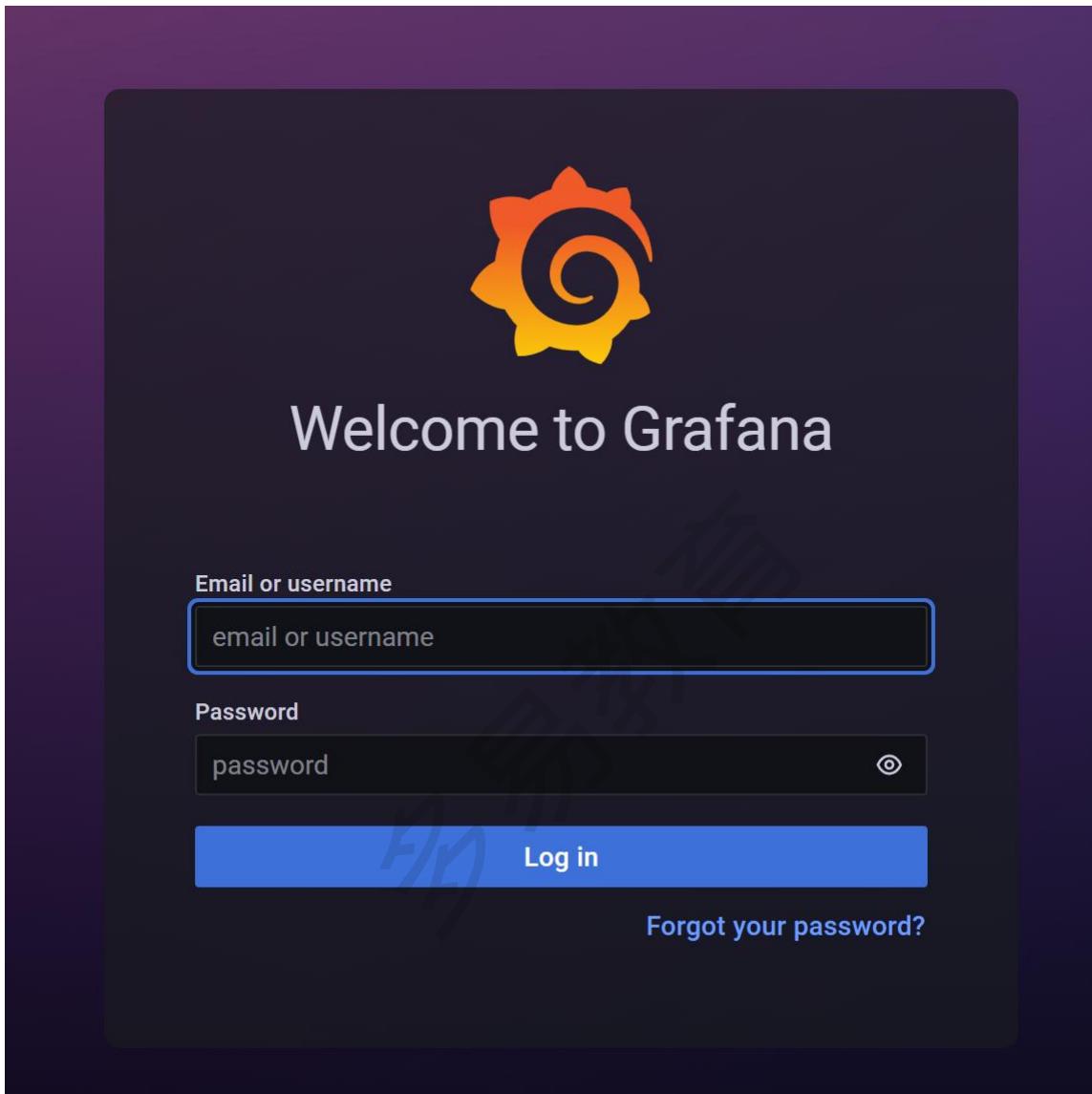
The screenshot shows the Prometheus Targets page. At the top, there are navigation icons and a search bar with the URL doitedu:9090/targets. Below the header, there are two sections: 'Targets' and 'pushgateway'. Under 'Targets', there are two entries: 'linux (1/1 up)' and 'prometheus (1/1 up)'. Each entry has a 'show more' button. A table below lists the targets with columns for Endpoint, State, and Labels. The 'linux' target has the endpoint http://doitedu:9090/metrics, state UP, and labels instance="doitedu:9090" and job="prometheus". The 'prometheus' target has the endpoint http://doitedu:9090/metrics, state UP, and labels instance="doitedu:9090" and job="prometheus". Under 'pushgateway', there is one entry: 'pushgateway (1/1 up)' with a 'show more' button.

Endpoint	State	Labels
http://doitedu:9090/metrics	UP	instance="doitedu:9090" job="prometheus"

16.3.3. Grafana 安装配置

- 下载: wget https://dl.grafana.com/oss/release/grafana-8.5.5.linux-amd64.tar.gz
- 解压: tar -zvxf grafana-8.5.5.linux-amd64.tar.gz

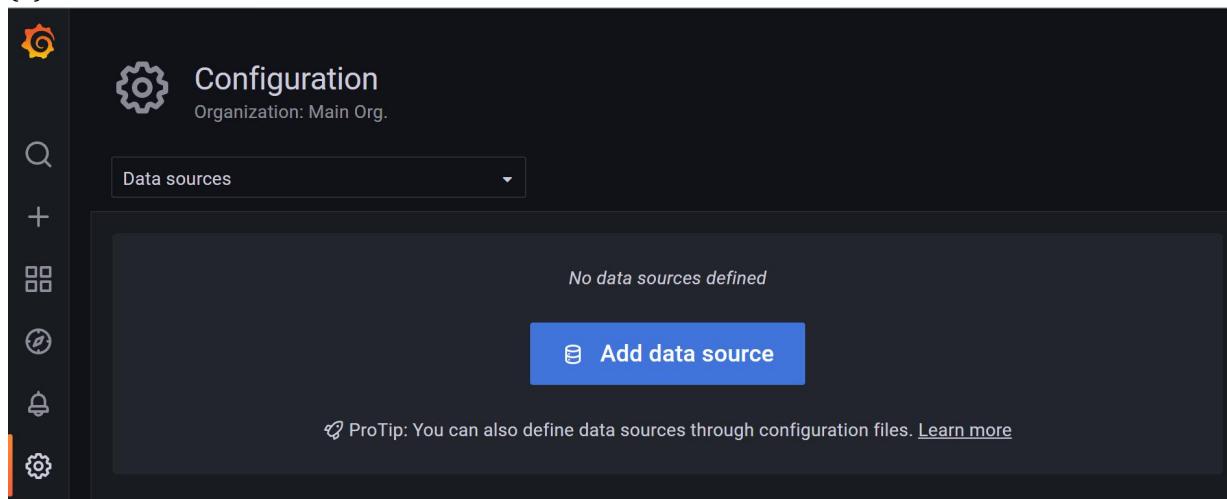
- 启动: ./bin/grafana-server web
- 访问: http://doitedu:3000/login
- 查看进程: netstat -apn | grep -E '9091|3000|9090|9100'



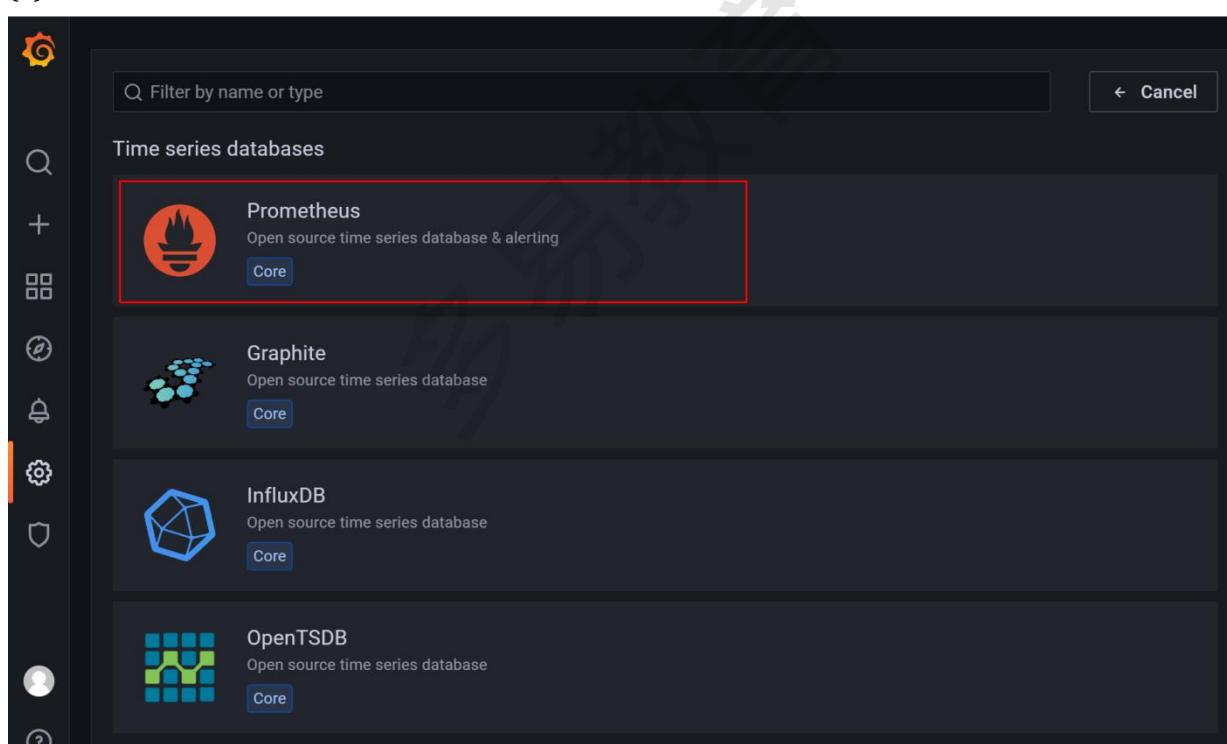
关于密码: 使用 admin 直接登录, 密码随意, 点击 login 后会提示设置密码;

16.3.4. grafana 监控界面操作示例

(1) 点击 Add data source



(2) 点击 Prometheus



(3) 输入 URL 地址，点击保存

Data Sources / Prometheus

Type: Prometheus

Settings

Configure your Prometheus data source below

Or skip the effort and get Prometheus (and Loki) as fully-managed, scalable, and hosted data sources from Grafana Labs with the [free-forever Grafana Cloud plan](#).

Name: Prometheus Default:

HTTP

URL: (highlighted by a red box)

Access: Server (default)

Allowed cookies: New tag (enter key to add)

Timeout: Timeout in seconds

(4) 点击 New dashboard

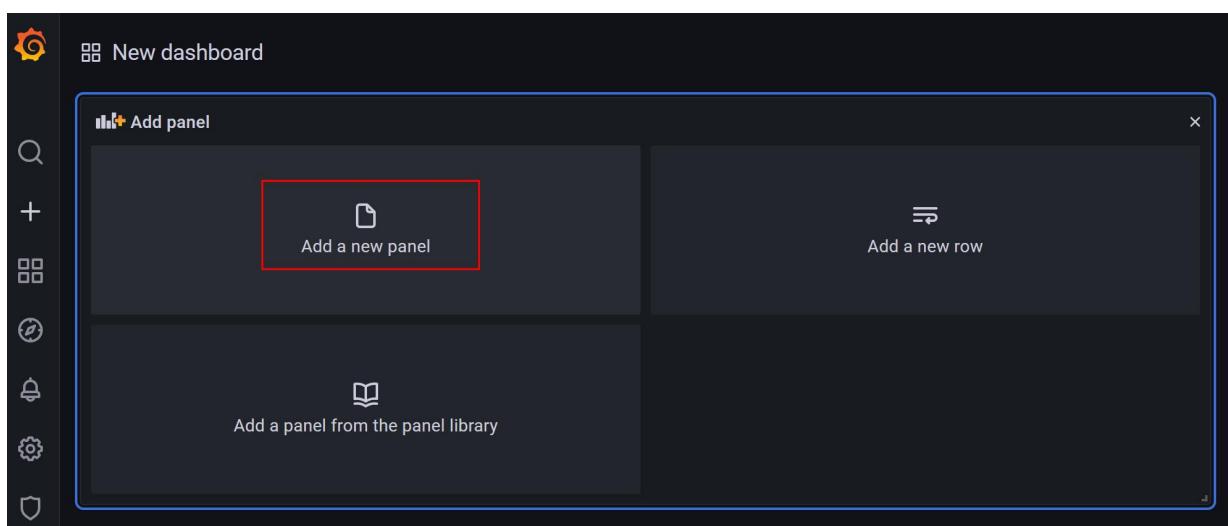
General / Home

Welcome to Grafana

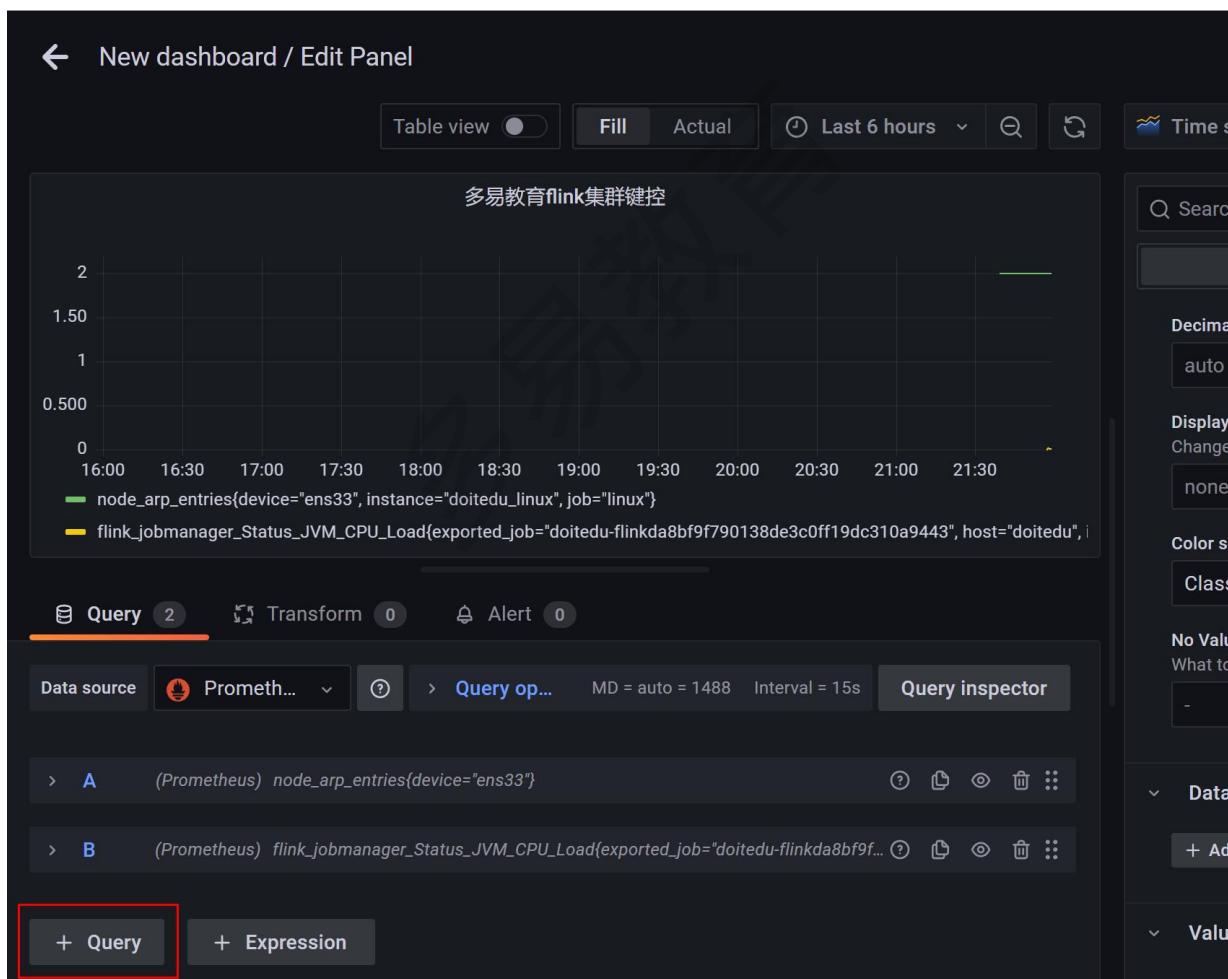
Create

- Dashboard
- Folder
- Import
- Alert rule

below will guide you to quickly finish setting up your Grafana installation.



(5) 点击 Add Quary



(6) 出现图例，表示成功



17. Flink 背压

17.1. 了解背压

什么是背压

在流式处理系统中，如果出现下游消费的速度跟不上上游生产数据的速度，就种现象就叫做背压 (backpressure,有人叫反压，不纠结，本篇叫背压)。本篇主要以 Flink 作为流式计算框架来简单背压机制，为了更好理解，只做简单分享。

背压产生的原因

下游消费的速度跟不上上游生产数据的速度，可能出现的原因如下：

- (1) 节点有性能瓶颈，可能是该节点所在的机器有网络、磁盘等等故障，机器的网络延迟和磁盘不足、频繁 GC、数据热点等原因。
- (2) 数据源生产数据的速度过快，计算框架处理不及时。比如消息中间件 kafka，生产者生产数据过快，下游 flink 消费计算不及时。
- (3) flink 算子间并行度不同，下游算子相比上游算子过小。

背压导致的影响

首先，背压不会直接导致系统的崩盘，只是处在一个不健康的运行状态。

- (1) 背压会导致流处理作业数据延迟的增加。
- (2) 影响到 Checkpoint，导致失败，导致状态数据保存不了，如果上游是 kafka 数据源，在一致性的要求下，可能会导致 offset 的提交不上。

原理：由于 Flink 的 Checkpoint 机制需要进行 Barrier 对齐，如果此时某个 Task 出现了背压，Barrier 流动的速度就会变慢，导致 Checkpoint 整体时间变长，如果背压很严重，还有可能导致 Checkpoint 超时失败。

- (3) 影响 state 的大小，还是因为 checkpoint barrier 对齐要求。导致 state 变大。

原理：接受到较快的输入管道的 barrier 后，它后面数据会被缓存起来但不处理，直到较慢的输入管道的 barrier 也到达。这些被缓存的数据会被放到 state 里面，导致 state 变大。

17.2. 如何查找定位背压

在 web 页面发现 flink 的 checkpoint 生成超时，失败。

ID	Status	Acknowledged	Trigger Time	Latest Acknowledgement	End to End Duration	State Size	Buffered During Alignment
+ 236663	IN_PROGRESS	4/8 (50%)	19:43:19	19:43:19	210ms	5.03 KB	0 B
+ 236662	FAILED	0/8	19:43:09	n/a	41ms	0 B	0 B
+ 236661	COMPLETED	8/8	19:42:44	19:42:46	2s	1.34 GB	0 B
+ 236660	COMPLETED	8/8	19:42:03	19:42:05	2s	1.34 GB	0 B
+ 236659	FAILED	2/8	19:41:28	19:41:28	638ms	2.48 KB	0 B
+ 236658	COMPLETED	8/8	19:41:18	19:41:22	3s	1.34 GB	0 B
+ 236657	FAILED	0/8	19:41:08	n/a	86ms	0 B	0 B
+ 236656	COMPLETED	8/8	19:40:33	19:40:36	3s	1.34 GB	0 B
+ 236655	COMPLETED	8/8	19:39:53	19:39:56	3s	1.34 GB	0 B
+ 236654	COMPLETED	8/8	19:39:43	19:39:45	2s	1.34 GB	0 B

查看 jobmanager 日志

2021-10-17 19:43:19,235 org.apache.flink.runtime.checkpoint.CheckpointCoordinator
-Checkpoint 236663 of job d521558603f6ef25dfd053c665d6afbe expired before completing

在 BackPressure 界面直接可以看到

背压状态可以大致锁定背压可能存在的算子，但具体背压是由于当前 Task 自身处理速度慢还是由于下游 Task 处理慢导致的，需要通过 metric 监控进一步判断。

ID	Start Time	Duration	Overview	Exceptions	TimeLine	Checkpoints	Configuration	Detail	SubTasks	TaskManagers	Watermarks	Accumulators	BackPressure	Metrics
ID: 0964cbac14e135e3760687f88b6f001a	Start Time: 2021-10-12 09:49:26	Duration: 5d 10h 21m											Measurement: 33s ago Back Pressure Status: OK	
								SubTask	Ratio				Status	
								1	0				OK	
								2	0				OK	
								3	0				OK	
								4	0				OK	
								5	0				OK	
								6	0				OK	
								7	0				OK	
								8	0				OK	
								9	0				OK	

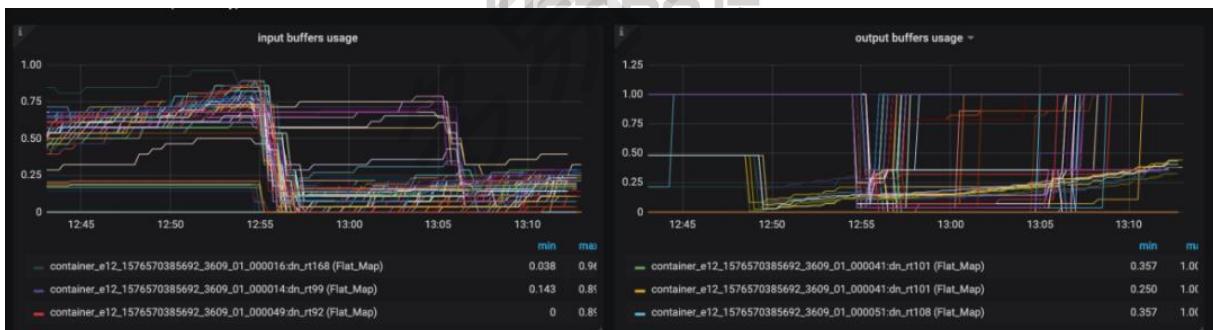
Source: Custom Source -> Map -> Filter -> Flat Map -> Filter -> Map -> Filter -> Sink: Unnamed
Parallelism: 9

Back Pressure Status	
Ratio	Status
1	HIGH
4	LOW

原理：BackPressure 界面会周期性的对 Task 线程栈信息采样，通过线程被阻塞在请求 Buffer 的频率来判断节点是否处于背压状态。计算缓冲区阻塞线程数与总线程数的比值 rate。其中，rate < 0.1 为 OK，0.1 <= rate <= 0.5 为 LOW，rate > 0.5 为 HIGH。

Metrics 监控背压

缓冲区的数据处理不过来，barrier 流动慢，导致 checkpoint 生成时间长，出现超时的现象。input 和 output 缓冲区都占满。



● outPoolUsage 与 inPoolUsage

指标	描述
outPoolUsage	发送端 Buffer 的使用率
inPoolUsage	接收端 Buffer 的使用率

指标可能出现以下情况：

- (1) outPoolUsage 与 inPoolUsage 都低，代表当前 Subtask 正常。
- (2) outPoolUsage 与 inPoolUsage 都高，代表当前 Subtask 下游背压。

(3) outPoolUsage 高，通常是被下游 Task 所影响。

(4) inPoolUsage 高，则表明它有可能是背压的根源。因为通常背压会传导至其上游，导致上游某些 Subtask 的 outPoolUsage 为高。

● inputFloatingBuffersUsage 与 inputExclusiveBuffersUsage

指标	描述
inputFloatingBuffersUsage	每个 Operator 实例对应一个 FloatingBuffers，inputFloatingBuffersUsage 表示 Operator 对应的 FloatingBuffers 使用率。
inputExclusiveBuffersUsage	每个 Operator 实例的每个远程输入通道(RemoteInputChannel)都有自己的一组独占缓冲区(ExclusiveBuffer)，inputExclusiveBuffersUsage 表示 ExclusiveBuffer 的使用率。

指标可能出现以下情况：

(1) floatingBuffersUsage 高，则表明背压正在传导至上游。

(2) floatingBuffersUsage 高、exclusiveBuffersUsage 低，则表明了背压可能存在倾斜。

17.3. 背压的原理

基于 Credit-based Flow Control 的背压机制

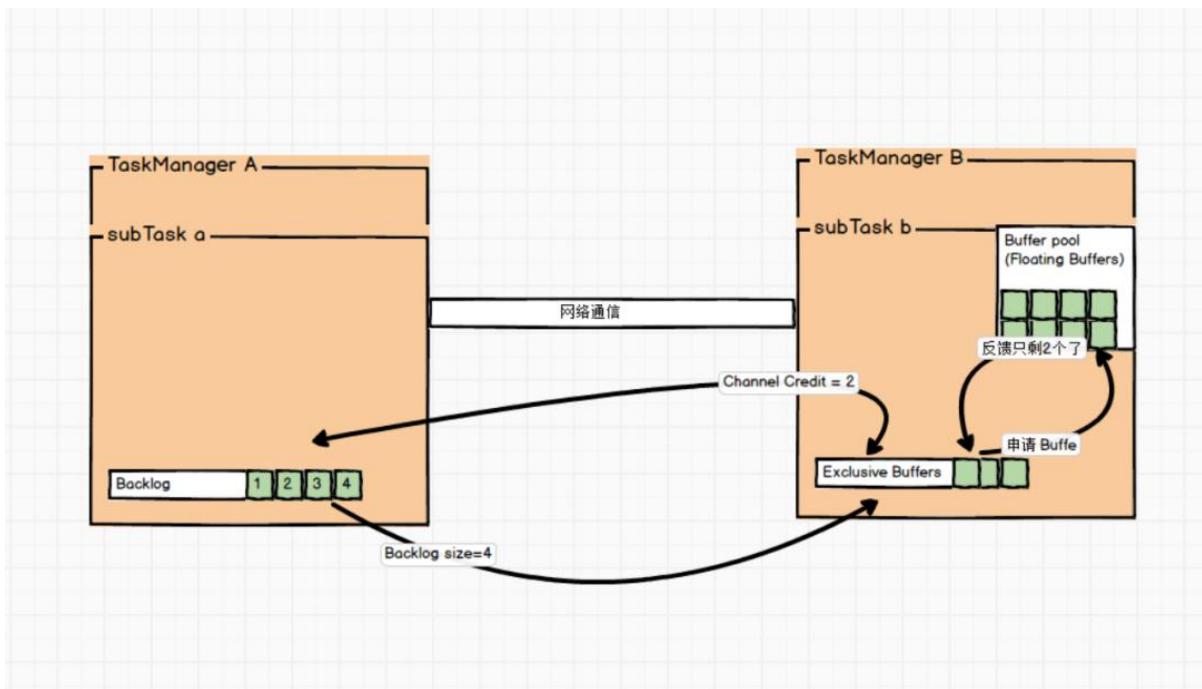
Credit 的反馈策略，保证每次上游发送的数据都是下游 InputChannel 可以承受的数据量。

具体原理是这样的：

(1) 上游 SubTask 给下游 SubTask 发送数据时，会把 Buffer 中要发送的数据和上游 ResultSubPartition 堆积的数据量 Backlog size 发给下游，下游接收到上游发来的 Backlog size 后，会向上游反馈现在的 Credit 值，Credit 值表示目前下游可以接收上游的 Buffer 量，1 个 Buffer 等价于 1 个 Credit。上游接收到下游反馈的 Credit 值后，上游下次最多只会发送 Credit 个数据到下游，保障不会有数据积压在 Socket 这一层。

(2) 当下游 SubTask 反压比较严重时，可能就会向上游反馈 Channel Credit = 0，此时上游就知道下游目前对应的 InputChannel 没有可用空间了，所以就不向下游发送数据了。

(3) 上游会定期向下游发送探测信号，检测下游返回的 Credit 是否大于 0，当下游返回的 Credit 大于 0 表示下游有可用的 Buffer 空间，上游就可以开始向下游发送数据了。



- (1) 上游 SubTask a 发送完数据后，还有 4 个 Buffer 被积压，那么会把发送数据和 Backlog size = 4 一块发送给下游 SubTask b。
- (2) 下游接受到数据后，知道上游积压了 4 个 Buffer，于是向 Buffer Pool 申请 Buffer，由于容量有限，下游 InputChannel 目前仅有 2 个 Buffer 空间。
- (3) SubTask b 会向上游 SubTask a 反馈 Channel Credit = 2。然后上游下一次最多只给下游发送 2 个 Buffer 的数据，这样每次上游发送的数据都是下游 InputChannel 的 Buffer 可以承受的数据量。

参考官网 [【https://flink.apache.org/2019/07/23/flink-network-stack-2.html】](https://flink.apache.org/2019/07/23/flink-network-stack-2.html)

自行了解老版本 TCP-based 背压机制，这里不再阐述。

17.4. 解决背压

Flink不需要一个特殊的机制来处理背压，因为Flink中的数据传输相当于已经提供了应对背压的机制。所以只有从代码上与资源上去做一些调整。

- (1) 背压部分原因可能是由于数据倾斜造成的，我们可以通过 Web UI 各个 SubTask 的 指标值来确认。Checkpoint detail 里不同 SubTask 的 State size 也是一个分析数据倾斜的有用指标。

解决方式把数据分组的 key 预聚合来消除数据倾斜。

- (2) 代码的执行效率问题，阻塞或者性能问题。
- (3) TaskManager 的内存大小导致背压。