Flink流与维表的关联

维表关联是离线计算或者实时计算里面常见的一种处理逻辑，常常用于字段补齐、规则过滤等，一般情况下维表数据放在MySql等数据库里面，对于离线计算直接通过ETL方式加载到Hive表中，然后通过sql方式关联查询即可。

维表是动态表，表里所存储的数据有可能不变，也有可能定时更新，但是更新频率不是很频繁。在业务开发中一般的维表数据存储在关系型数据库如mysql，oracle等，也可能存储在hbase，redis等nosql数据库。

无须借助维表join

当维度数据基本不变，或很少变化时，可以把维度数据插入到redis，并借助于本地缓存，当流消息来的时候，直接根据key去内存中查找即可，无须借助维表join

预加载维表

对于维表数据不是很大的情况，可以把数据直接加载到内存。需要继承RichFunction类(RichFlatMapFunctioin)，重写open方法，可以在open里建立mysql等数据源的连接。同时可以在open里新建一个线程定时加载维表，这样可以无需重启，实现维度数据的周期性更新。

广播维表

同样要求数据量不是很大，利用broadcast state将维度数据流广播到下游task做join。

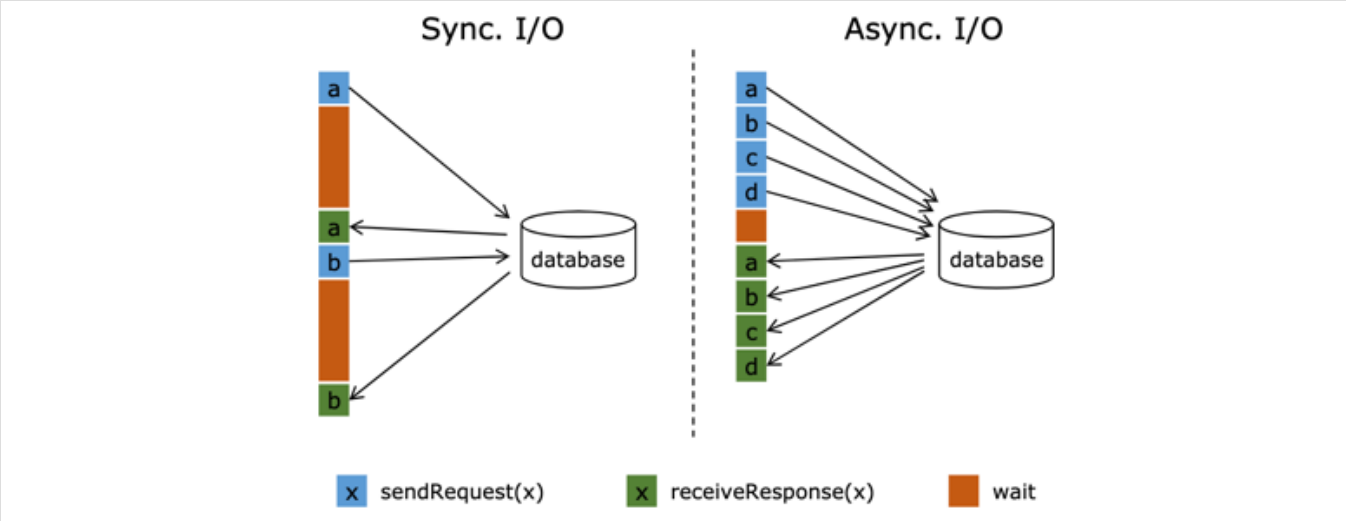
实现方式：

1. 将维度数据发送到kafka作为广播原始流s1
2. 定义状态描述符MapStateDescriptor，调用s1.broadcast()，获得broadcastStream s2
3. 调用非广播流s3.connect(s2)，得到broadcastconnectedStream s4
4. 在keyedBroadcastProcessFunction或BroadcastProcessFunction实现处理逻辑，并作为参数传递给s4.process()

通过AsyncIO实现维表join

AsyncIO

流计算系统中经常需要与外部系统进行交互，比如需要查询外部数据库以关联上用户的额外信息。通常，我们的实现方式是向数据库发送用户`a`的查询请求，然后等待结果返回，在这之前，我们无法发送用户`b`的查询请求。这是一种同步访问的模式，如下图左边所示。



图中棕色的长条表示等待时间，可以发现网络等待时间极大地阻碍了吞吐和延迟。为了解决同步访问的问题，异步模式可以并发地处理多个请求和回复。也就是说，你可以连续地向数据库发送用户`a`、`b`、`c`等的请求，与此同时，哪个请求的回复先返回了就处理哪个回复，从而连续的请求之间不需要阻塞等待，如上图右边所示。这也正是 Async I/O 的实现原理。

使用 Async I/O 的前提是需要一个支持异步请求的客户端。当然，。Flink 提供了非常简洁的API，让用户只需要关注业务逻辑，一些脏活累活比如消息顺序性和一致性保证都由框架处理了，多么棒的事情！

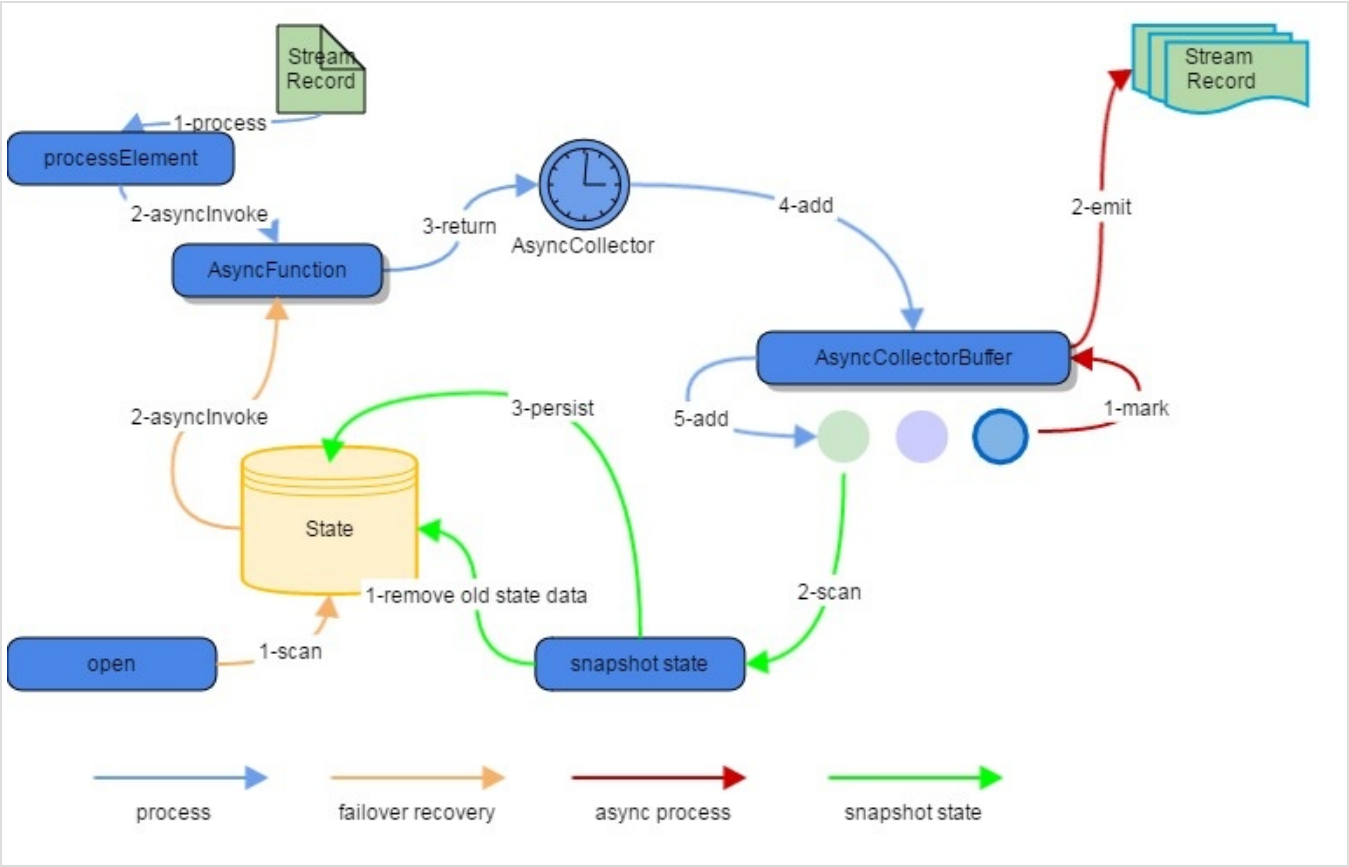
注意：

- 使用AsyncIO，需要外部存储有支持异步请求的客户端**（没有异步请求客户端的话也可以将同步客户端丢到线程池中执行作为异步客户端）**
- 使用AsyncIO，需要继承RichAsyncFunction，重写或实现`open()、close()、asyncInvoke()`三个方法即可。
- 使用AsyncIO，最好结合缓存一起使用，可减少请求外部存储的次数。
- Flink 1.9中，Async I/O 提供了`Timeout`参数来控制请求最长等待时间。默认，异步I/O请求超时时，会引发异常并重启或停止作业。 如果要处理超时，可以重写`AsyncFunction#timeout`方法。
- Flink 1.9中，Async I/O 提供了`Capacity`参数控制请求并发数，一旦`Capacity`被耗尽，会触发反压机制来抑制上游数据的摄入。
- Async I/O 输出提供乱序和顺序两种模式。

> 乱序：用`AsyncDataStream.unorderedWait(...)` API，每个并行的输出顺序和输入顺序可能不一致
>
> 有序：用`AsyncDataStream.orderedWait(...)` API，每个并行的输出顺序和输入顺序一致。为保证顺序，需要在输出的Buffer中排序，该方式效率会低一些。

Async处理原理

流中的记录如何在异步处理过程中与checkpoint进行交互？具体流程如下：



- 蓝色线条为正常的异步处理流程

  - 上游`opertor`发送过来一条记录，进入`asyncInoke`算子
  - `AsyncFunction`处理后的结果收集到`AsyncCollector`中
  - 将`AsyncCollector`中的记录添加到异步缓存`AysncCollectorBuffer`中
- 红色部分是异步处理的流程

  - `AsyncCollectorBuffer`保存了所有的`AsyncCollector`，在调用`AsyncCollector.collect()`时，会在`AsyncCollectorBuffer`中标记buffer中的记录，用于标记已经完成的`Asyncollectors`
  - 一旦`AsyncCollector`获得异步的结果，就会触发一个名为`Emitter`的线程，根据有序或无序的设置向下游operator发送结果
- 绿色的线条是状态保存的流程

  - 先在`snapshot state`中清除掉旧的状态
  - `AsyncCollectorBuffer`完成后，将完成的buffer记录状态同步到`snapshot state`中
  - 将`snapshot state`持久化到`state`
- 橙色部分是用于失败时从checkpoint的恢复流程

  - 从`open`方法中初始化state中的状态信息
  - 获取信息后重新进入`AsyncFunction`，重新进行数据查询

编码

main

```
public static void main(String[] args) throws Exception {
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEr
    env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
    Properties properties = new Properties();
    properties.setProperty("bootstrap.servers", "localhost:9092");
    properties.setProperty("zookeeper.connect", "localhost:2181");
    properties.setProperty("group.id", "test");

    FlinkKafkaConsumer011<String> myConsumer = new FlinkKafkaConsumer011<String>("te
    DataStream<String> stream = env.addSource(myConsumer);

    SampleAsyncFunction asyncFunction = new SampleAsyncFunction();
    DataStream<String> result = AsyncDataStream.orderedWait(stream,asyncFunction,100
    /// 注册成表进行sql查询
    //....
    result.print();
    env.execute("");
}
```

SampleAsyncFunction

```
private static class SampleAsyncFunction extends RichAsyncFunction<String,String> {
    private transient ExecutorService executorService;
    private static java.sql.Connection conn;
    private static PreparedStatement stmt;
    @Override
    public void open(Configuration conf) throws Exception {
        super.open(conf);
        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql:///day15_jdbc", "root", "
            stmt = conn.prepareStatement("");
            executorService = Executors.newFixedThreadPool(30);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public void close() throws Exception {
        super.close();
        stmt.close();
        conn.close();
    }

    @Override
    public void asyncInvoke(String s, ResultFuture<String> resultFuture) throws SQLE
        // 先取缓存查询
        // 缓存没有，则取sql查询

        String result = null;
        ResultSet rs = stmt.executeQuery();
```

```
        if (!rs.isClosed() && rs.next()) {
            result = rs.getString(1);
        }
        resultFuture.complete(Collections.singleton(result));
    }

    @Override
    public void timeout(AsyncUser input, ResultFuture<String> resultFuture) throws E
        logger.warn("Async function for hbase timeout");
        ....
        resultFuture.complete(..);
    }
}
```

此方法是将源表的数据打到`asyncInvoke()`算子，由该算子异步加载数据，将源表和维表组成一个宽表后输出并注册为一个新表，并交由flink执行sql语句。***一般会结合缓存，用来减少与外部系统的交互***，但会有几个弊端：

1. DataStream到Table来回切换不方便
2. 将两个表合并为一个宽表时，如果两个表包含相同属性名称的话需要做区分
3. select语句中的表名要做替换，考虑的情况比较多

实现LookupableTableSource接口，直接可以写sql (UDTF)

Flink 1.9 中维表功能来源于新加入的Blink中的功能，如果你要使用该功能，那就需要自己引入 Blink 的 Planner，而不是引用社区的 Planner。由于新合入的 Blink 相关功能，使得 Flink 1.9 实现维表功能很简单，只要自定义实现 **LookupableTableSource** 接口，同时实现里面的方法就可以进行：

```
public interface LookupableTableSource<T> extends TableSource<T> {
    TableFunction<T> getLookupFunction(String[] lookupKeys);
    AsyncTableFunction<T> getAsyncLookupFunction(String[] lookupKeys);
    boolean isAsyncEnabled();
}
```

- **isAsyncEnabled** 方法主要表示该表是否支持异步访问外部数据源获取数据，当返回 true 时，那么在注册到 TableEnvironment 后，使用时会返回异步函数进行调用，当返回 false 时，则使同步访问函数。

同步访问函数getLookupFunction

getLookupFunction 会返回同步方法，这里你需要自定义 **TableFuntion** 进行实现，**TableFunction** 本质是 **UDTF**,输入一条数据可能返回多条数据，也可能返回一条数据。用户自定义 **TableFunction** 格式如下

```
public class MyLookupFunction extends TableFunction<Row> {
@Override
    public void open(FunctionContext context) throws Exception {
        super.open(context);
    }
    public void eval(Object... paramas) {
```

```
    }
}
```

- open 方法在进行初始化算子实例的进行调用，异步外部数据源的client要在类中定义为 transient,然后在 open 方法中进行初始化，这样每个任务实例都会有一个外部数据源的 client。防止同一个 client 多个任务实例调用，出现线程不安全情况。

- eval 则是 TableFunction 最重要的方法，它用于关联外部数据。当程序有一个输入元素时，就会调用eval一次，用户可以将产生的数据使用 collect() 进行发送下游。paramas 的值为用户输入元素的值，比如在 Join 的时候，使用 A.id = B.id and A.name = b.name, **B 是维表，A 是用户数据表**，paramas 则代表 A.id,A.name 的值。

异步访问函数getAsyncLookupFunction

`getAsyncLookupFunction` 会返回异步访问外部数据源的函数，如果你想使用异步函数，前提是 `LookupableTableSource` 的 `isAsyncEnabled` 方法返回 true 才能使用。使用异步函数访问外部数据系统，一般是外部系统有异步访问客户端，如果没有的话，可以自己使用线程池异步访问外部系统。至于为什么使用异步访问函数，无非就是为了提高程序的吞吐量，不需要每条记录访问返回数据后，才去处理下一条记录。异步函数格式如下：

```
public class MyAsyncLookupFunction extends AsyncTableFunction<Row> {
    @Override
    public void open(FunctionContext context) throws Exception {
        super.open(context);
    }
    public void eval(CompletableFuture<Collection<Row>> future, Object... params) {
    }
}
```

- 外部数据源异步客户端初始化。如果是线程安全的(多个客户端一起使用)，你可以不加 transient 关键字,初始化一次。否则，你需要加上 transient,不对其进行初始化，而在 open 方法中，为每个 Task 实例初始化一个。
- eval 方法中多了一个 CompletableFuture,当异步访问完成时，需要调用其方法进行处理.

编码

main

```
public static void main(String[] args) throws Exception {
    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironm
    EnvironmentSettings settings = EnvironmentSettings.newInstance().useBlinkPlanner
    StreamTableEnvironment tableEnvironment = StreamTableEnvironment.create(env,sett

    Properties properties = new Properties();
    properties.setProperty("bootstrap.servers", "localhost:9092");
    properties.setProperty("zookeeper.connect", "localhost:2181");
    properties.setProperty("group.id", "test");
```

```
10
11      FlinkKafkaConsumer011<String> myConsumer = new FlinkKafkaConsumer011<String>("te
12      DataStream<UserTest> source = env.addSource(myConsumer)
13          .map(new MapFunction<String, UserTest>() {
14              @Override
15              public UserTest map(String s) throws Exception {
16                  UserTest userTest = new UserTest();
17                  userTest.setId(Integer.valueOf(s.split(",")[0]));
18                  userTest.setName(s.split(",")[1]);
19                  return userTest;
20              }
21          });
22
23      tableEnvironment.registerDataStream("ubt",source,"id,name,proctime.proctime");
24
25      MysqlAsyncLookupTableSource tableSource = MysqlAsyncLookupTableSource.Builder
26          .newBuilder().withFieldNames(new String[]{"id","name"})
27          .withFieldTypes(new TypeInformation[] {Types.INT,Types.STRING})
28          .build();
29      tableEnvironment.registerTableSource("info",tableSource);
30
31      String sql = "select a.id,b.name from ubt as a join info FOR SYSTEM_TIME AS OF a
32      Table table = tableEnvironment.sqlQuery(sql);
33      DataStream<Tuple2<Boolean, UserTest>> result = tableEnvironment.toRetractStream(
34      result.process(new ProcessFunction<Tuple2<Boolean,UserTest>, Object>() {
35          @Override
36          public void processElement(Tuple2<Boolean, UserTest> booleanUserTestTuple2,
37              if (booleanUserTestTuple2.f0) {
38                  System.out.println(JSON.toJSONString(booleanUserTestTuple2.f1));
39              }
40          }
41      });
42      env.execute("");
43  }
```

MysqlAsyncLookupTableSource implements LookupableTableSource

```
1   public static class MysqlAsyncLookupTableSource implements LookupableTableSource<Use
2
3       private final String[] fieldNames;
4       private final TypeInformation[] fieldTypes;
5
6       public MysqlAsyncLookupTableSource(String[] fieldNames, TypeInformation[] fieldT
7           this.fieldNames = fieldNames;
8           this.fieldTypes = fieldTypes;
9       }
10
11      @Override
12      public TableFunction<UserTest> getLookupFunction(String[] strings) {
13          return null;
14      }
15
```

```
    @Override
    public AsyncTableFunction<UserTest> getAsyncLookupFunction(String[] strings) {
        return MysqlAsyncLookupFunction.Builder.getBuilder()
            .withFieldNames(fieldNames)
            .withFieldTypes(fieldTypes)
            .build();
    }

    @Override
    public boolean isAsyncEnabled() {
        return true;
    }

    @Override
    public TableSchema getTableSchema() {
        return TableSchema.builder()
            .fields(fieldNames, TypeConversions.fromLegacyInfoToDataType(fieldTypes))
            .build();
    }

    public static final class Builder {
        private String[] fieldNames;
        private TypeInformation[] fieldTypes;

        private Builder() {
        }

        public static Builder newBuilder() {
            return new Builder();
        }

        public Builder withFieldNames(String[] fieldNames) {
            this.fieldNames = fieldNames;
            return this;
        }

        public Builder withFieldTypes(TypeInformation[] fieldTypes) {
            this.fieldTypes = fieldTypes;
            return this;
        }

        public MysqlAsyncLookupTableSource build() {
            return new MysqlAsyncLookupTableSource(fieldNames, fieldTypes);
        }
    }
}
```

MysqlAsyncLookupFunction extends AsyncTableFunction

```
public static class MysqlAsyncLookupFunction extends AsyncTableFunction<UserTest> {
    private final String[] fieldNames;
    private final TypeInformation[] fieldTypes;
```

```
    public MysqlAsyncLookupFunction(String[] fieldNames, TypeInformation[] fieldType
        this.fieldNames = fieldNames;
        this.fieldTypes = fieldTypes;
    }

    // 每一条流数据都会调用此方法进行join
    public void eval(CompletableFuture<Collection<UserTest>> resultFuture, Object...
        // 进行维表查询
    }


    @Override
    public void open(FunctionContext context) {
        // 建立连接
    }

    @Override
    public void close(){}

    public static final class Builder {
        private String[] fieldNames;
        private TypeInformation[] fieldTypes;

        private Builder() {
        }

        public static Builder getBuilder() {
            return new Builder();
        }

        public Builder withFieldNames(String[] fieldNames) {
            this.fieldNames = fieldNames;
            return this;
        }

        public Builder withFieldTypes(TypeInformation[] fieldTypes) {
            this.fieldTypes = fieldTypes;
            return this;
        }

        public MysqlAsyncLookupFunction build() {
            return new MysqlAsyncLookupFunction(fieldNames, fieldTypes);
        }
    }
}
```

注:

```
select a.id,b.name
from ubt as a
```

```
3  join info FOR SYSTEM_TIME AS OF a.proctime as b
4  on a.id=b.id
```

b表后需要跟上 `FOR SYSTEM_TIME AS OF PROCTIME()` 的关键字，其含义是每条到达的数据所关联上的是到达时刻的维表快照，也就是说，当数据到达时，我们会根据数据上的 key 去查询远程数据库，拿到匹配的结果后关联输出。这里的 `PROCTIME` 即 processing time。同时将`a.proctime`字段换成表中的某个时间字段(如`orderdate`)后，还可以关联到`orderdate`时的维表