21 | 场景案例：如何用 Flink SQL CDC 实现实时数据同步?

今天我们来看第二个案例，也就是用 Flink SQL CDC 实现实时数据同步。

那究竟什么是 Flink SQL CDC 呢？毕竟这是个相对还比较新的技术，可能很多人都还没听说过这个技术，所以我们先从它诞生的业务场景说起。

业务场景

如果你是一名后端开发的话，相信十有八九遇到过这样的问题，有时候一种数据库满足不了业务的需求，我们需要将相同的数据，存入多种不同的数据库。

比如，最开始的时候业务比较简单，数据量也很小，数据只需要保存到 MySQL 中，作为主数据库即可。之后，随着业务的发展，数据量变得越来越大，为了提升查询效率，需要将数据写一份到 Redis 缓存。同时，业务查询也变得越来越复杂，为了提供更加灵活和高效的查询分析方式，需要将数据再写一份到 Elasticsearch 里。

面对以上这种情况，你会怎么做呢？一般情况下，我们首先想到的可能就是，改代码！改成类似于下面图 1 这样的方案。

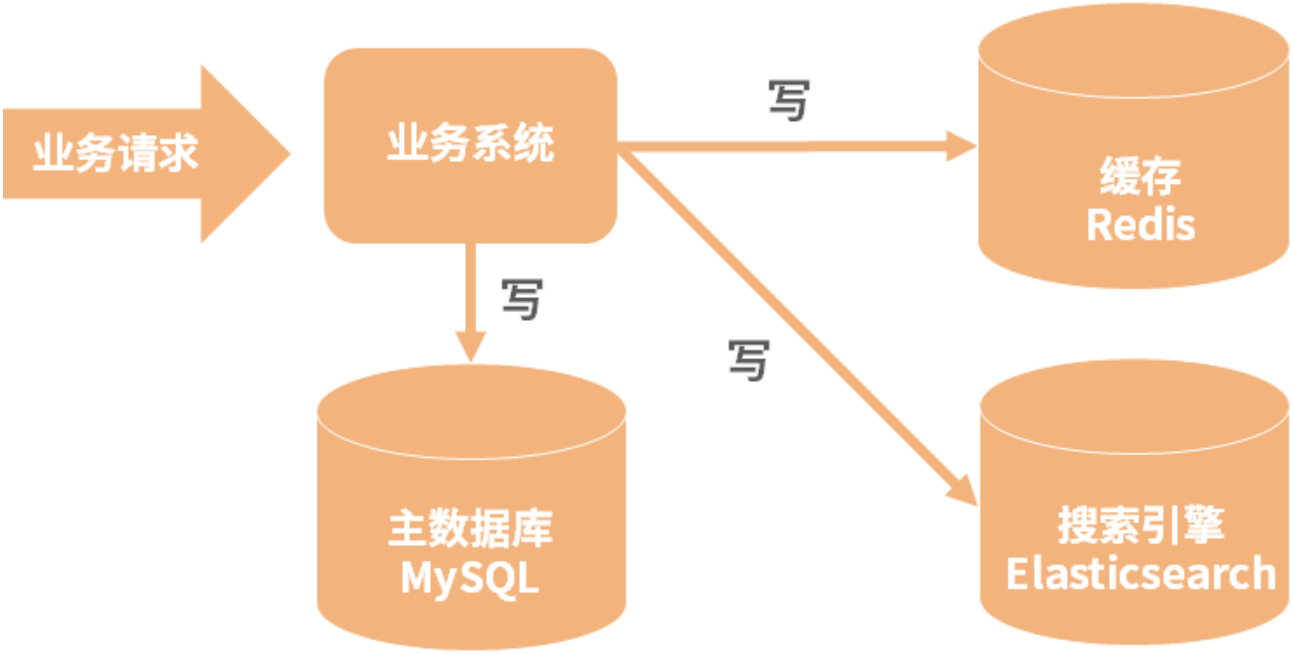

图 1 在业务代码中将数据写入不同数据库

在上面的图 1 中，我们直接在业务代码中，将数据写入了几种不同的数据库里，包括 MySQL、Redis 和 Elasticsearch。

这种方案看着简单，也很容易实现。但这种做法**非常不好**，原因是这样的：

1. 代码严重耦合，每次需要修改业务代码后重新测试和上线才行；
2. 在业务系统中需要多次将数据写入不同数据库，严重影响业务代码性能；
3. 增量同步前，需要先由人工（至少你要写脚本和执行脚本吧）做一次全量同步。

一种好的改进方法，则是**使用消息中间件**，就像下面的图 2 这样。

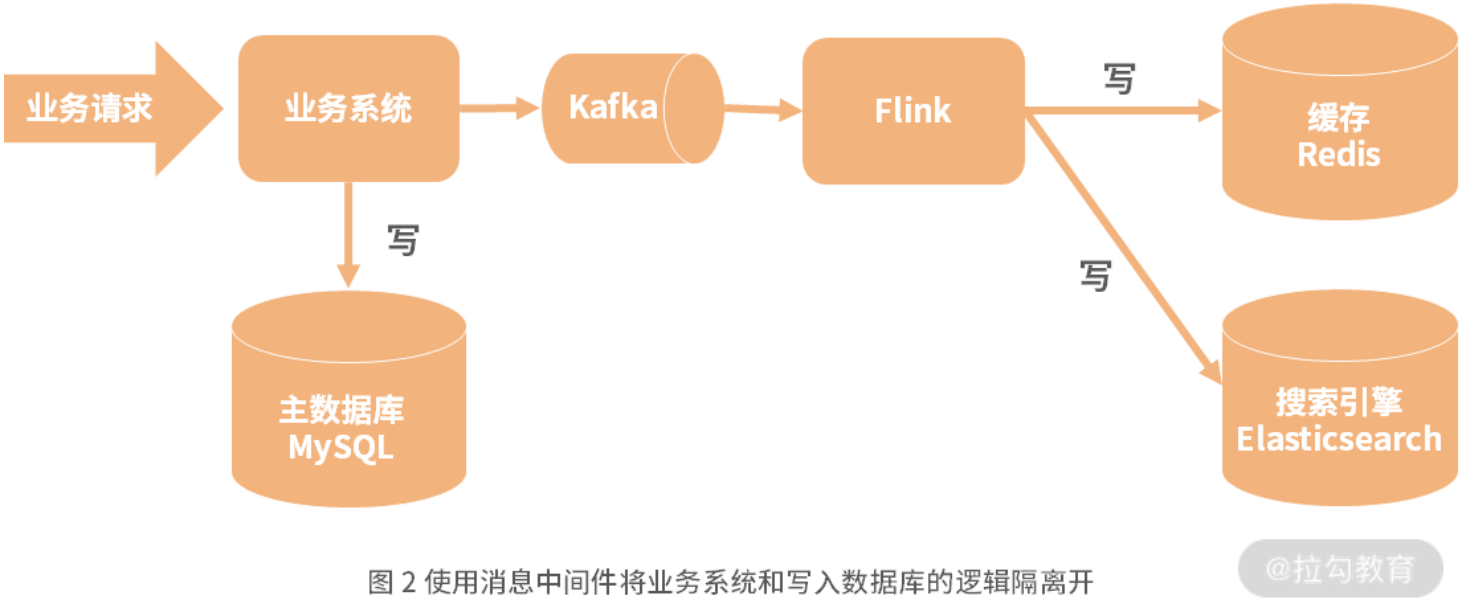

图 2 使用消息中间件将业务系统和写入数据库的逻辑隔离开

在上面的图 2 中，首先由业务系统将数据写入 Kafka，然后由 Flink 从 Kafka 将消息读取出来，最后再写入不同的数据库。

这种做法最大的优点在于**将业务系统和写入数据库的逻辑隔离开，降低了代码的耦合度**，并且在业务中只需要写一次数据到 Kafka 即可，提升了业务系统的性能。

但它还是有几个缺点：

1. 还是需要修改代码，比如在写数据的地方埋点写入 Kafka；
2. 增加了系统的复杂性，因为需要维护 Kafka，并且要开发写入数据库的代码。当需要写入的数据库和表比较多时，这种复杂性就更加严重了；
3. 增量同步前，同样需要先由人工做一次全量同步。

虽然有以上这些缺点，但不管怎样，这种方法在思路上都是值得遵循的。因为它可以解耦，并且性能会更好。特别是当我们有了 Flink 这种神器后，可以直接通过 Flink 从 Kafka 里读取出数据，然后高效地使用流计算技术写入各种数据库。

面对以上图 2 方案的缺点，咱们的 Flink 神器当然没有熟视无睹。所以，它基于 CDC 技术的思路，推出了 Flink CDC 实现。

CDC（Change Data Capture，变化数据捕获）正如其名，是一种捕获数据变化的技术。比如在 MySQL 做同步时，我们设置 MySQL**从数据库**（secondary database）来跟随**主数据库**（primary database）的 binlog 日志，从而将主数据库的所有数据变化同步到从数据库中，这其实就是一个 CDC 的使用场景。

而 Flink CDC 就是一种使用 Flink 流计算框架，来实现 CDC 功能的技术。比如，我们可以通过 Flink CDC 技术，先将主数据库的全量数据同步到另外的数据库中，然后再跟随主数据库的 binlog 日志，将所有增量的数据也实时同步到从数据库中。

由于 Flink CDC 将全量同步和增量同步的操作封装到了一起，并且因为 Flink 还支持 SQL 语句，所以最终我们只需要写几行简单的 SQL，就能轻松解决将同一份数据写入多种不同数据库的问题。

下面的图 3 就展示了使用 Flink CDC 进行数据同步的方案。

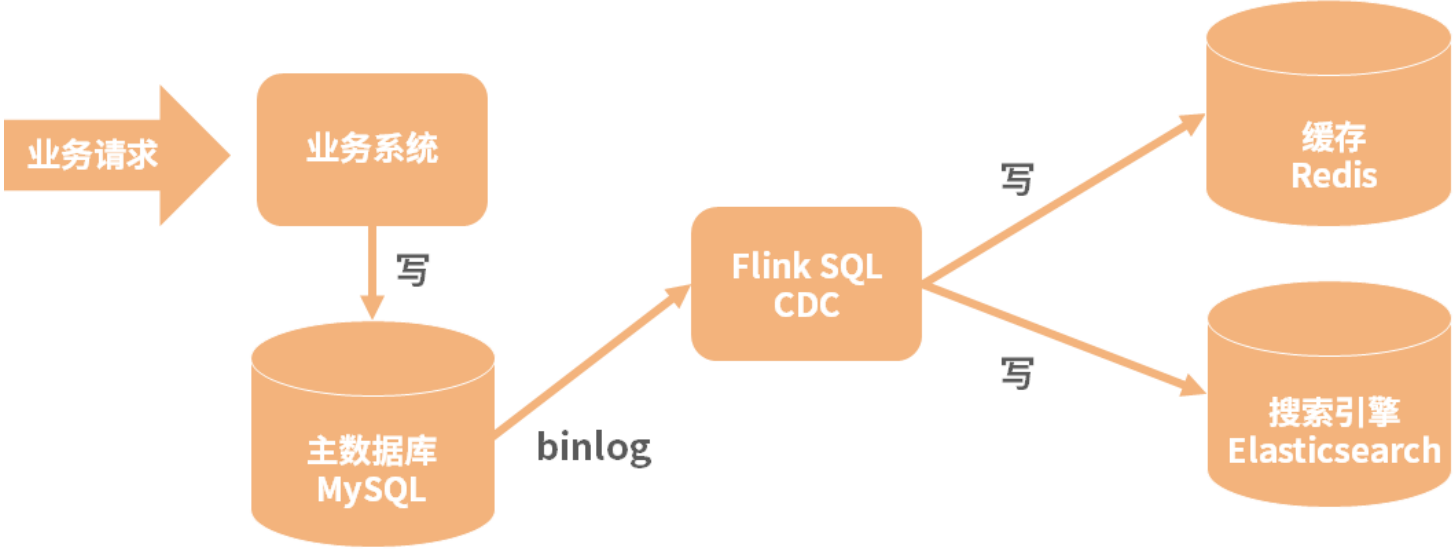

图 3 使用 Flink SQL CDC 进行数据同步

从上面的图 3 可以看出，相比前面图 2 的方案，这里不需要再引入单独的 Kafka，也不需要我们修改业务系统的代码，只需要引入一个 Flink SQL CDC 工具，就能够同时实现数据的全量同步和增量同步。并且，由于 Flink SQL CDC 封装得很好，我们只需要写一些 SQL 就可以了。

Flink CDC 如此神奇，那它究竟是怎样实现的呢？接下来，我们就来看下 Flink CDC 的实现原理。

实现原理

我们以使用 Flink CDC 从 MySQL 中同步数据的情景，来讲解下 Flink CDC 的工作原理。

一般来说，Flink CDC 同步数据需要两个步骤。

- **第一步是将源数据库的数据全量同步到目标数据库；**
- **第二步是跟随源数据库的 binlog 日志，将源数据库的所有变动，以增量数据的方式同步到目标数据库。**

我们先来看将源数据库的数据**全量同步**到目标数据库的过程。Flink CDC 将这个过程称之为“快照”（sanpshot），具体步骤是这样的。

1. Flink CDC 会获取一个全局读锁（global read lock），从而阻塞其他客户端往数据库写入数据。不用担心这个锁定时间会很长，因为它马上就会在第 5 步中被释放掉。
2. 启动一个可重复读语义（repeatable read semantics）的事务，从而确保后续在该事务内的所有“读”操作都是在“一致的快照”（consistent snapshot）中进行。这一步中“可重复读语义”以及后续步骤只涉及“读”操作是非常关键的。因为只有在“可重复读语义”且不存在“写”操作的情况下， MySQL 的“可重复读语义”级别事务才不会存在“幻读”现象（具体原因可以参考这个问题的两个回答），这样才能保证后续做“扫描”和读取 binlog 位置时，它们的数据和时间点是对得上的。这就是“一致的快照”的含义，它保证了同步到目标数据库中的数据是完整的，并且和源数据库中的数据是完全相同的，既不会多一条，也不会少一条。
3. 读取当前 binlog 的位置。
4. 读取 Flink CDC 配置指定的数据库和表定义(schema)。
5. 释放步骤 1 中的全局读锁。这个时候其他的客户端就可以继续往数据库中写入数据了。从步骤 1 到步骤 5，Flink CDC 并没有做非常耗时的任务，所以全局锁定的时间很短，这样对业务运行的影响可以尽量降至最小。
6. 将步骤 4 读取的数据库和表定义，作用到目标数据库上。
7. 对数据库里的表进行全表扫描，将读取出的每条记录，都发送到目标数据库。
8. 完成全表扫描后，提交（commit）步骤 2 时启动的可重复读语义事务。

9. 将步骤 3 读取的 binlog 位置记录下来，表明本次数据全量同步过程（也就是“快照”）成功完成。后续做增量同步时，如果发现没有这个 binlog 位置记录，就意味着数据全量同步过程是失败的，可以重新再做一次步骤 1 到步骤 9，直到全量同步成功为止。

可以看出，数据全量同步的过程还是比较复杂的，但好在 Flink CDC 的 flink-connector-mysql-cdc 连接器插件 已经为我们实现了这个过程，所以我们直接使用它就好了。

完成数据全量同步后，后面的**增量同步**过程就相对简单了，直接跟随源数据库的 binlog 日志，然后将每次的数据变更同步到目标数据库即可。增量同步过程中，Flink 自己会**周期性地执行 checkpoint 操作**，从而记录下当时增量同步到的 binlog 位置。

这样，如果 Flink CDC 作业（job）因为发生故障而重启的话，也能够从最近一次 checkpoint 处，**恢复出故障发生前的状态**，从而继续执行之前的过程。

最后，再配合**写入目标数据库时是幂等性的操作**。这样，就保证了 Flink CDC 的整个数据同步过程，**能够达到 exactly once 级别的数据处理可靠性**。是不是非常惊艳！

以上就是 Flink CDC 的工作原理。接下来，我们就具体展示下，如何使用 Flink CDC 技术，进行实时数据同步。

具体实现

我们这里以将 MySQL 里的数据实时同步到 Elasticsearch 为例。Flink CDC 的具体实现方式有两种，一种是基于 DataStream（参见链接）的方式，另一种是基于 Table & SQL（参见链接）的方式。我们说的 Flink SQL CDC 就是指基于 Table & SQL 方式的 Flink CDC 实现。

由于 Flink SQL 在经过解析之后，最终也会转化为基于 DataStream 的底层代码。所以我先演示直接使用 DataStream 的方式，然后再演示使用 SQL 的方式，这样更加有助于理解。

基于 DataStream 的方式

我们先来看基于 DataStream 的方式。具体代码如下（完整代码和操作说明参看这里）。

```
public class FlinkCdcDemo {
    public static void main(String[] args) throws Exception {
        // 源数据库，下面是以 MySQL 作为源数据库的配置
        SourceFunction<String> sourceFunction = MySQLSource.<String>builder()
                .hostname("127.0.0.1")
                .port(3306)
                .databaseList("db001")
                .username("root")        // 测试用，生产不要用root账号！
                .password("123456")      // 测试用，生产不要用这种简单密码！
                .deserializer(new StringDebeziumDeserializationSchema())
                .build();
        // 目标数据库，下面是以 Elasticsearch 作为目标数据库的配置
        List<HttpHost> httpHosts = new ArrayList<>();
        httpHosts.add(new HttpHost("127.0.0.1", 9200, "http"));
        ElasticsearchSink.Builder<String> esSinkBuilder = new ElasticsearchSink.Builder<>(
                httpHosts,
                new ElasticsearchSinkFunction<String>() {
                    private IndexRequest createIndexRequest(String element) {
                        Map<String, String> json = new HashMap<>();
                        // 这里直接将数据 element 表示为字符串
                        json.put("data", element);
                        return Requests.indexRequest()
                                .index("table001")
                                .source(json);
                    }
                    @Override
                    public void process(String element, RuntimeContext ctx, RequestIndexer indexe
                        // 这里就是将数据同步到目标数据库 Elasticsearch
                        indexer.add(createIndexRequest(element));
                    }
                }
        );
        // 实验时配置逐条插入，生产为了提升性能的话，可以改为批量插入
        esSinkBuilder.setBulkFlushMaxActions(1);
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env
                .addSource(sourceFunction)  // 设置源数据库
                .addSink(esSinkBuilder.build())  // 设置目标数据库
                .setParallelism(1); // 设置并行度为1，以保持消息顺序
        env.execute("FlinkCdcDemo");
    }
}
```

在上面的代码中，我们首先使用 MySQLSource 类配置了一个 MySQL 源数据库。然后，我们再使用 ElasticsearchSink 类配置了一个 Elasticsearch 目标数据库。最后，我们使用 addSource 和 addSink 函数，将源数据库和目标数据库之间的数据同步链路打通。这样，我们就实现了基于 DataStream 的 Flink CDC 实时数据同步功能。

可以看到，上面的代码还是非常简单的。主要的原因在于 MySQLSource 这个 CDC Connector 为我们封装了所有的复杂操作，这些复杂操作就包括我们在实现原理部分讲到的“全量同步”和“增量同步”的实现细节。

不过，上面的代码并不完美，它主要有两个问题：

1. 一是，同步数据时，写入 Elasticsearch 的数据是字符串，而不是经过解析的各个独立字段。这样就会导致很多没有用的字段也保存到了 Elasticsearch ，并且后续查询的效率会非常低。
2. 二是，还是需要写一些代码。虽然上面的代码并不复杂，但是毕竟还是没有 SQL 方便。

为了解决以上两个问题，接下来我们就使用 Table & SQL 的方式来实现 Flink CDC 功能，这就是 Flink SQL CDC。你会发现，我们真的就只需要写几行 SQL 语句，就能轻松解决上面两个问题。

基于 Table & SQL 方式

下面就是基于 Table & SQL 方式实现 Flink CDC 功能的代码（完整代码和操作说明参看这里）。

```
-- 在 Flink SQL Client 里执行以下 SQL。
-- 创建源数据库
CREATE TABLE sourceTable (
  id INT,
  name STRING,
  counts INT,
  description STRING
) WITH (
 'connector' = 'mysql-cdc',
 'hostname' = '192.168.1.7',
 'port' = '3306',
 'username' = 'root',
 'password' = '123456',
 'database-name' = 'db001',
 'table-name' = 'table001'
);
-- 创建目标数据库
CREATE TABLE sinkTable (
  id INT,
  name STRING,
  counts INT
) WITH (
  'connector' = 'elasticsearch-7',
  'hosts' = 'http://192.168.1.7:9200',
  'index' = 'table001'
);
-- 启动 Flink SQL CDC 作业
insert into sinkTable select id, name, counts from sourceTable;
```

看到没！只需要简简单单三个 SQL 语句，就实现了 Flink CDC 的功能。

其中，CREATE TABLE sourceTable 是在配置一个 MySQL 源数据库，CREATE TABLE sinkTable 是在配置一个 Elasticsearch 目标数据库。而 insert into select from 则是指定了需要从源数据库向目标数据库**同步哪些字段**，并且它会触发启动这个 Flink CDC 作业。

当启动 Flink CDC 作业后，如果我们向 MySQL 写入数据，你就可以看到数据从 MySQL 同步到 Elasticsearch 的效果了。

下面的图 4 就是 Flink CDC 数据同步的效果图。

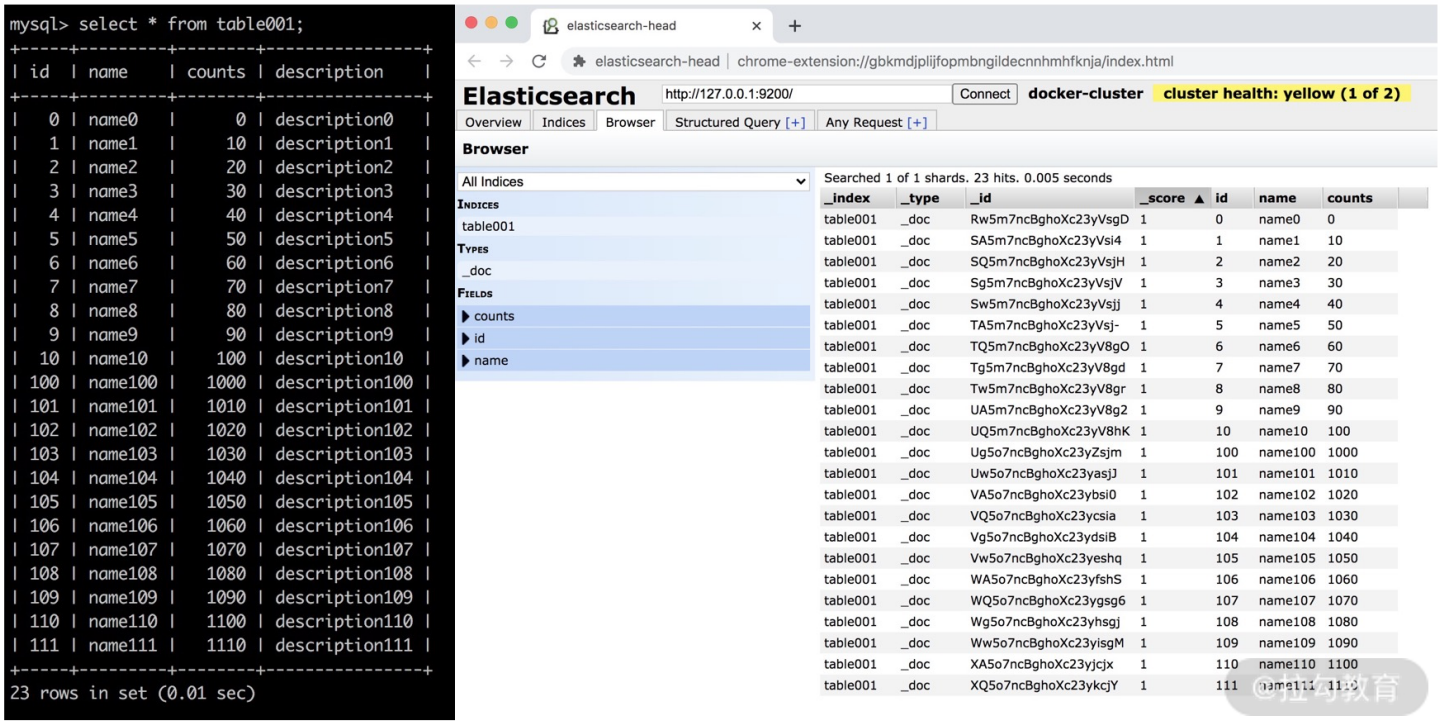

图 4 使用 Flink CDC 实时同步数据的效果图

从上面的图 4 可以看出，左边源数据库 MySQL 里的数据和右边目标数据库 Elasticsearch 里的数据是完全对应的。并且，同步到 Elasticsearch 里数据的字段，也是和我们在 insert into select from 语句里指定的字段是完全一致的。你看，Flink SQL CDC 实现实时数据同步的效果是不是很不错！

最后还需要说明下的是，这里我为了专注于讲解 Flink CDC 的工作原理本身，就使用了相对简单的 SQL 语句。其实，Flink SQL CDC 是可以使用一些更加复杂的 SQL 语句，来实现更加丰富的数据同步功能的。比如，使用 GROUP BY 分组和使用 Window 进行窗口计算等。对于这些更完整和更复杂的 Flink SQL 语句说明，你可以参考 这里的官方文档 。

小结

总的来说，相比 DataStream 的方式，Flink SQL CDC 使用起来会更加方便些。但这两种方式我们都需要掌握，因为目前 Flink SQL CDC 还不算非常成熟，一些 Flink SQL 暂时不支持的功能和插件，还是需要我们自己基于 DataStream 在底层实现。

你的工作中有没有可以使用到 Flink CDC，或者用 Flink CDC 进行改造的场景呢？可以将你的想法或问题写在留言区。

下面是本课时的知识脑图。