基于Canal与Flink实现数据实时增量同步(二)

本文主要从Binlog实时采集和离线处理Binlog还原业务数据两个方面，来介绍如何实现DB数据准确、高效地进入Hive数仓。

背景

在数据仓库建模中，未经任何加工处理的原始业务层数据，我们称之为ODS(Operational Data Store)数据。在互联网企业中，常见的ODS数据有业务日志数据（Log）和业务DB数据（DB）两类。对于业务DB数据来说，从MySQL等关系型数据库的业务数据进行采集，然后导入到Hive中，是进行数据仓库生产的重要环节。如何准确、高效地把MySQL数据同步到Hive中？一般常用的解决方案是批量取数并Load：直连MySQL去Select表中的数据，然后存到本地文件作为中间存储，最后把文件Load到Hive表中。这种方案的优点是实现简单，但是随着业务的发展，缺点也逐渐暴露出来：

- 性能瓶颈：随着业务规模的增长，Select From MySQL -> Save to Localfile -> Load to Hive这种数据流花费的时间越来越长，无法满足下游数仓生产的时间要求。
- 直接从MySQL中Select大量数据，对MySQL的影响非常大，容易造成慢查询，影响业务线上的正常服务。
- 由于Hive本身的语法不支持更新、删除等SQL原语(高版本Hive支持，但是需要分桶+ORC存储格式)，对于MySQL中发生Update/Delete的数据无法很好地进行支持。

为了彻底解决这些问题，我们逐步转向CDC (Change Data Capture) + Merge的技术方案，即实时Binlog采集 + 离线处理Binlog还原业务数据这样一套解决方案。Binlog是MySQL的二进制日志，记录了MySQL中发生的所有数据变更，MySQL集群自身的主从同步就是基于Binlog做的。

实现思路

首先，采用Flink负责把Kafka上的Binlog数据拉取到HDFS上。

然后，对每张ODS表，首先需要一次性制作快照（Snapshot），把MySQL里的存量数据读取到Hive上，这一过程底层采用直连MySQL去Select数据的方式，可以使用Sqoop进行一次性全量导入。

最后，对每张ODS表，每天基于存量数据和当天增量产生的Binlog做Merge，从而还原出业务数据。

Binlog是流式产生的，通过对Binlog的实时采集，把部分数据处理需求由每天一次的批处理分摊到实时流上。无论从性能上还是对MySQL的访问压力上，都会有明显地改善。Binlog本身记录了数据变更的类型（Insert/Update/Delete），通过一些语义方面的处理，完全能够做到精准的数据还原。

实现方案

Flink处理Kafka的binlog日志

使用kafka source，对读取的数据进行JSON解析，将解析的字段拼接成字符串，符合Hive的schema格式，具体代码如下：

```
package com.etl.kafka2hdfs;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONArray;
import com.alibaba.fastjson.JSONObject;
import com.alibaba.fastjson.parser.Feature;
import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.SimpleStringEncoder;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.core.fs.Path;
import org.apache.flink.runtime.state.StateBackend;
import org.apache.flink.runtime.state.filesystem.FsStateBackend;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperat
or;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironm
ent;
import org.apache.flink.streaming.api.functions.sink.filesystem.RollingPol
icy;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingF
ileSink;
import org.apache.flink.streaming.api.functions.sink.filesystem.rollingpol
icies.DefaultRollingPolicy;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;

import java.util.Map;
import java.util.Properties;

/**
 *  @Created with IntelliJ IDEA.
 *  @author : jmx
 *  @Date: 2020/3/27
 *  @Time: 12:52
```

```
 *
 */
public class HdfsSink {
    public static void main(String[] args) throws Exception {
        String fieldDelimiter = ",";
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getE
xecutionEnvironment();
        env.setParallelism(1);

        // checkpoint
        env.enableCheckpointing(10_000);
        //env.setStateBackend((StateBackend) new FsStateBackend("file:///
E://checkpoint"));
        env.setStateBackend((StateBackend) new FsStateBackend("hdfs://km
s-1:8020/checkpoint"));
        CheckpointConfig config = env.getCheckpointConfig();
        config.enableExternalizedCheckpoints(CheckpointConfig.Externalized
CheckpointCleanup.DELETE_ON_CANCELLATION);

        // source
        Properties props = new Properties();
        props.setProperty("bootstrap.servers", "kms-2:9092,kms-3:9092,km
s-4:9092");
        // only required for Kafka 0.8
        props.setProperty("zookeeper.connect", "kms-2:2181,kms-3:2181,km
s-4:2181");
        props.setProperty("group.id", "test123");
        FlinkKafkaConsumer<String> consumer = new FlinkKafkaConsumer<>(
                "qfbap_ods.code_city", new SimpleStringSchema(), props);
        consumer.setStartFromEarliest();
        DataStream<String> stream = env.addSource(consumer);

        // transform
        SingleOutputStreamOperator<String> cityDS = stream
                .filter(new FilterFunction<String>() {
                    // 过滤掉DDL操作
                    @Override
                    public boolean filter(String jsonVal) throws Exceptio
n {
                        JSONObject record = JSON.parseObject(jsonVal, Fe
ature.OrderedField);
                        return record.getString("isDdl").equals("false")
;
                    }
                })
                .map(new MapFunction<String, String>() {

                    @Override
                    public String map(String value) throws Exception {
```

```
                        StringBuilder fieldsBuilder = new StringBuilder()
;
                        // 解析JSON数据
                        JSONObject record = JSON.parseObject(value, Feat
ure.OrderedField);
                        // 获取最新的字段值
                        JSONArray data = record.getJSONArray("data");
                        // 遍历，字段值的JSON数组，只有一个元素
                        for (int i = 0; i < data.size(); i++) {
                            // 获取到JSON数组的第i个元素
                            JSONObject obj = data.getJSONObject(i);
                            if (obj != null) {
                                fieldsBuilder.append(record.getLong("id")
); // 序号id
                                fieldsBuilder.append(fieldDelimiter); // 字
段分隔符
                                fieldsBuilder.append(record.getLong("es")
); //业务时间戳
                                fieldsBuilder.append(fieldDelimiter);
                                fieldsBuilder.append(record.getLong("ts")
); // 日志时间戳
                                fieldsBuilder.append(fieldDelimiter);
                                fieldsBuilder.append(record.getString("ty
pe")); // 操作类型
                                for (Map.Entry<String, Object> entry : o
bj.entrySet()) {

                                    fieldsBuilder.append(fieldDelimiter);
                                    fieldsBuilder.append(entry.getValue()
); // 表字段数据
                                }

                            }
                        }
                        return fieldsBuilder.toString();
                    }

                });

        //cityDS.print();
        //stream.print();

        // sink
        // 以下条件满足其中之一就会滚动生成新的文件
        RollingPolicy<String, String> rollingPolicy = DefaultRollingPoli
cy.create()
                .withRolloverInterval(60L * 1000L) //滚动写入新文件的时间，
默认60s。根据具体情况调节
                .withMaxPartSize(1024 * 1024 * 128L) //设置每个文件的最大大
```

```
小 ,默认是128M, 这里设置为128M
                .withInactivityInterval(60L * 1000L) //默认60秒,未写入数据
处于不活跃状态超时会滚动新文件
                .build();

        StreamingFileSink<String> sink = StreamingFileSink
                //.forRowFormat(new Path("file:///E://binlog_db/city"), n
ew SimpleStringEncoder<String>())
                .forRowFormat(new Path("hdfs://kms-1:8020/binlog_db/code
_city_delta"), new SimpleStringEncoder<String>())
                .withBucketAssigner(new EventTimeBucketAssigner())
                .withRollingPolicy(rollingPolicy)
                .withBucketCheckInterval(1000)  // 桶检查间隔, 这里设置1S
                .build();

        cityDS.addSink(sink);
        env.execute();
    }
}
```

对于Flink Sink到HDFS，`StreamingFileSink`替代了先前的`BucketingSink`，用来将上游数据存储到 HDFS 的不同目录中。它的核心逻辑是分桶，默认的分桶方式是`DateTimeBucketAssigner`，即按照处理时间分桶。处理时间指的是消息到达 Flink 程序的时间，这点并不符合我们的需求。因此，我们需要自己编写代码将事件时间从消息体中解析出来，按规则生成分桶的名称，具体代码如下：

```
1  package com.etl.kafka2hdfs;
2
3  import org.apache.flink.core.io.SimpleVersionedSerializer;
4  import org.apache.flink.streaming.api.functions.sink.filesystem.BucketAssi
5  gner;
6  import org.apache.flink.streaming.api.functions.sink.filesystem.bucketassi
7  gners.SimpleVersionedStringSerializer;
8  import java.text.SimpleDateFormat;
9  import java.util.Date;
10
11 /**
12  *  @Created with IntelliJ IDEA.
13  *  @author : jmx
14  *  @Date: 2020/3/27
15  *  @Time: 12:49
16  *
17  */
18
19 public class EventTimeBucketAssigner implements BucketAssigner<String, S
20 tring> {
21
```

```
    @Override
    public String getBucketId(String element, Context context) {
        String partitionValue;
        try {
            partitionValue = getPartitionValue(element);
        } catch (Exception e) {
            partitionValue = "00000000";
        }
        return "dt=" + partitionValue;//分区目录名称
    }

    @Override
    public SimpleVersionedSerializer<String> getSerializer() {
        return SimpleVersionedStringSerializer.INSTANCE;
    }
    private String getPartitionValue(String element) throws Exception {

        // 取出最后拼接字符串的es字段值，该值为业务时间
        long eventTime = Long.parseLong(element.split(",")[1]);
        Date eventDate = new Date(eventTime);
        return new SimpleDateFormat("yyyyMMdd").format(eventDate);
    }
}
```

离线还原MySQL数据

经过上述步骤，即可将Binlog日志记录写入到HDFS的对应的分区中，接下来就需要根据增量的数据和存量的数据还原最新的数据。Hive 表保存在 HDFS 上，该文件系统不支持修改，因此我们需要一些额外工作来写入数据变更。常用的方式包括：JOIN、Hive 事务、或改用 HBase、kudu。

如昨日的存量数据code_city,今日增量的数据为code_city_delta，可以通过 `FULL OUTER JOIN`，将存量和增量数据合并成一张最新的数据表，并作为明天的存量数据：

```
INSERT OVERWRITE TABLE code_city
SELECT
        COALESCE( t2.id, t1.id ) AS id,
        COALESCE ( t2.city, t1.city ) AS city,
        COALESCE ( t2.province, t1.province ) AS province,
        COALESCE ( t2.event_time, t1.event_time ) AS event_time
FROM
        code_city t1
        FULL OUTER JOIN (
SELECT
        id,
        city,
        province,
```

```
        event_time
FROM
        (-- 取最后一条状态数据
SELECT
        id,
        city,
        province,
        dml_type,
        event_time,
        row_number ( ) over ( PARTITION BY id ORDER BY event_time DESC )
 AS rank
FROM
        code_city_delta
WHERE
        dt = '20200324' -- 分区数据
        ) temp
WHERE
        rank = 1
        ) t2 ON t1.id = t2.id;
```

小结

本文主要从Binlog流式采集和基于Binlog的ODS数据还原两方面，介绍了通过Flink实现实时的ETL，此外还可以将binlog日志写入kudu、HBase等支持事务操作的NoSQL中，这样就可以省去数据表还原的步骤。本文是《基于Canal与Flink实现数据实时增量同步》的第二篇，关于canal解析Binlog日志写入kafka的实现步骤，参见《基于Canal与Flink实现数据实时增量同步一》。

refrence：

[1]https://tech.meituan.com/2018/12/06/binlog-dw.html