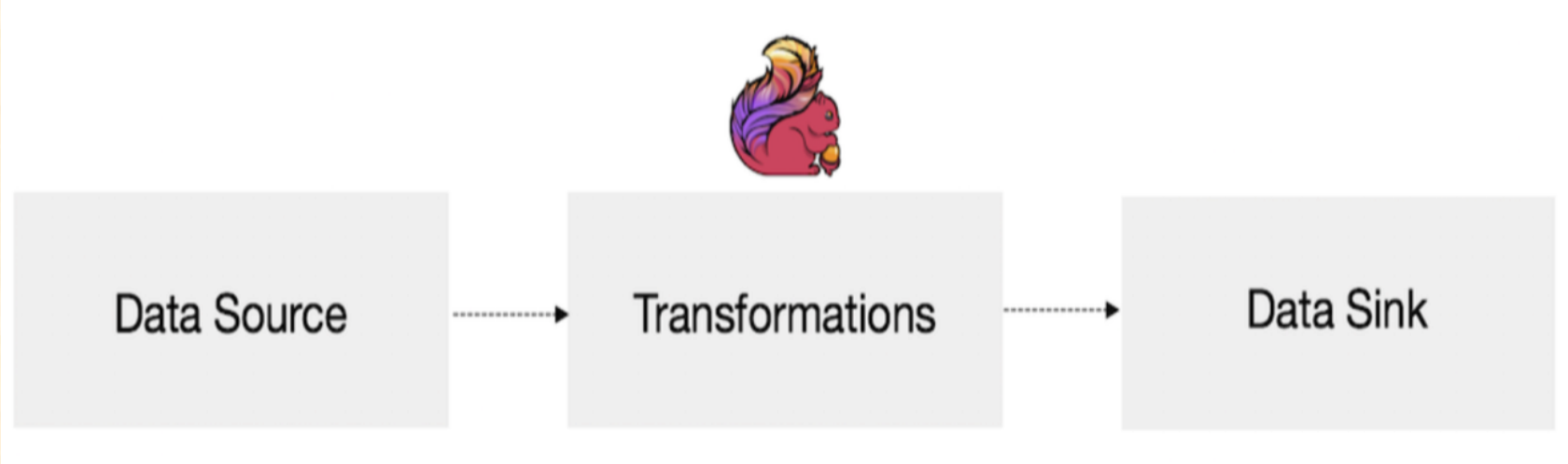
Flink 实时数仓项目实战

QQ群：732021751
QQ或微信：84985152

Flink DataSet API

- 编程基本套路
- DataSource
- Transfamation
- DataSink
- 广播变量
- 分布式缓存
- 函数的参数传递

1. 运行模型



1. WordCount

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

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

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

        wordCounts.print();
    }

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

2. DataSource

- **内置数据源**

➢ **基于文件**

✓ readTextFile(path) / TextInputFormat：逐行读取文件并将其作为字符串(String)返回

✓ readTextFileWithValue(path) / TextValueInputFormat：逐行读取文件并将其作为StringValue返回。StringValue是Flink对String的封装，可变、可序列化，一定程度上提高性能。

✓ readCsvFile(path) / CsvInputFormat：解析以逗号(或其他字符)分隔字段的文件。返回元组或pojo

✓ readFileOfPrimitives(path, Class) / PrimitiveInputFormat

✓ readFileOfPrimitives(path, delimiter, Class) / PrimitiveInputFormat

跟readCsvFile类似，只不过以原生类型返回而不是Tuple。

✓ readSequenceFile(Key, Value, path) / SequenceFileInputFormat：读取SequenceFile，以Tuple2<Key, Value>返回

➢ **基于Collection**

✓ fromCollection(Collection)

✓ fromCollection(Iterator, Class)

✓ fromElements(T ...)

✓ fromParallelCollection(SplittableIterator, Class)

✓ generateSequence(from, to)

➢ **通用**

✓ readFile(inputFormat, path) / FileInputFormat

✓ createInput(inputFormat) / InputFormat

2. DataSource

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

// read text file from local files system
DataSet<String> localLines = env.readTextFile("file:///path/to/my/textfile");

// read text file from a HDFS running at nnHost:nnPort
DataSet<String> hdfsLines = env.readTextFile("hdfs://nnHost:nnPort/path/to/my/textfile");

// read a CSV file with three fields
DataSet<Tuple3<Integer, String, Double>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                               .types(Integer.class, String.class, Double.class);

// read a CSV file with five fields, taking only two of them
DataSet<Tuple2<String, Double>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                               .includeFields("10010")  // take the first and the fourth field
                               .types(String.class, Double.class);

// read a CSV file with three fields into a POJO (Person.class) with corresponding fields
DataSet<Person>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                         .pojoType(Person.class, "name", "age", "zipcode");

// read a file from the specified path of type SequenceFileInputFormat
DataSet<Tuple2<IntWritable, Text>> tuples =
 env.readSequenceFile(IntWritable.class, Text.class, "hdfs://nnHost:nnPort/path/to/file");

// creates a set from some given elements
DataSet<String> value = env.fromElements("Foo", "bar", "foobar", "fubar");

// generate a number sequence
DataSet<Long> numbers = env.generateSequence(1, 10000000);

// Read data from a relational database using the JDBC input format
DataSet<Tuple2<String, Integer> dbData =
    env.createInput(
      JDBCInputFormat.buildJDBCInputFormat()
                     .setDrivername("org.apache.derby.jdbc.EmbeddedDriver")
                     .setDBUrl("jdbc:derby:memory:persons")
                     .setQuery("select name, age from persons")
                     .setRowTypeInfo(new RowTypeInfo(BasicTypeInfo.STRING_TYPE_INFO, BasicTypeInfo.INT_TYPE_INFO))
                     .finish()
    );
```

2. 解析CSV文件

- types(Class ... types)：配置字段的类型，对于已解析字段必须配置
- lineDelimiter(String del)：指定行分隔符，默认'\n'
- fieldDelimiter(String del)：指定字段分隔符，默认','
- includeFields(boolean ... flag), includeFields(String mask), includeFields(long bitMask)指定要去读取的字段， 默认types()来决定，type指定n个类型，则读取前n个字段。
- parseQuotedStrings(char quoteChar)：启用引用字符串解析，默认不启用
- ignoreComments(String commentPrefix)：指定注释前缀。所有以指定的注释前缀开头的行都不会被解析和忽略。默认情况下不会忽略任何行。
- ignoreInvalidLines()：是否忽略无法正确解析的行，默认不忽略直接抛异常
- ignoreFirstLine()：忽略首行，默认不忽略

```
// read a CSV file with three fields
DataSet<Tuple3<Integer, String, Double>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                                .types(Integer.class, String.class, Double.class);

// read a CSV file with five fields, taking only two of them
DataSet<Tuple2<String, Double>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                                .includeFields("10010")  // take the first and the fourth field
                                .types(String.class, Double.class);

// read a CSV file with three fields into a POJO (Person.class) with corresponding fields
DataSet<Person>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                         .pojoType(Person.class, "name", "age", "zipcode");
```

2. 目录的递归遍历

```
// enable recursive enumeration of nested input files
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// create a configuration object
Configuration parameters = new Configuration();

// set the recursive enumeration parameter
parameters.setBoolean("recursive.file.enumeration", true);

// pass the configuration to the data source
DataSet<String> logs = env.readTextFile("file:///path/with.nested/files")
                          .withParameters(parameters);
```

3.Transfamation-Map & FlatMap & MapPartition

- 含义：数据映射(1进1出&1进n出&n进n出)

- MapPartition:类似map，一次处理一个分区的数据【如果在进行map处理的时候需要获取第三方资源，建议使用MapPartition

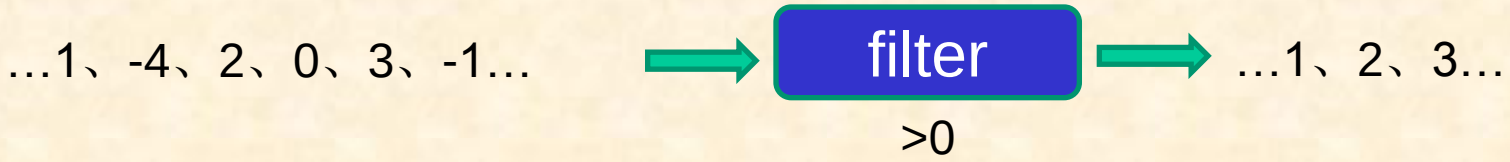
- 使用场景：
  - ✓ 过滤脏数据、数据清洗等

- 案例：看MapPartition代码示例

3. Transfamation-filter

- 含义：数据筛选(满足条件event的被筛选出来进行后续处理)，根据FliterFunction返回的布尔值来判断是否保留元素，true为保留，false则丢弃

- 使用场景：
  - ✓ 过滤脏数据、数据清洗等



3.Transfamation-聚合操作

- Reduce：对数据进行聚合操作，结合当前元素和上一次reduce返回的值进行聚合操作，然后返回一个新的值。

- aggregate：sum、max、min等。

3. Transfamation-Distinct

- 含义：返回数据集中不相同的元素。 它从输入DataSet中删除重复条目，依据元素的所有字段或字段的子集。

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

DataSet<Tuple3<Long,String,Integer>> inputs=env.fromElements(
        Tuple3.of(1L,"zhangsan",28),
        Tuple3.of(3L,"lisi",34),
        Tuple3.of(3L,"wangwu",23),
        Tuple3.of(3L,"zhaoliu",34),
        Tuple3.of(3L,"maqi",25)
);

inputs.distinct( ...fields: 0).print();
```

3. Transfamation-Union

- 含义：生成两个DataSet的并集，两个DataSet必须是相同类型的

```
DataSet<Tuple2<String, Integer>> vals1 = // [...]
DataSet<Tuple2<String, Integer>> vals2 = // [...]
DataSet<Tuple2<String, Integer>> vals3 = // [...]
DataSet<Tuple2<String, Integer>> unioned = vals1.union(vals2).union(vals3);
```

3.Transfamation-inner join(等值连接)

- 含义：两个DataSet连接为一个Dataset。两个数据集的元素在一个或多个key上连接
- 指定join key的四种方式：
  - key expression
  - key-selector function
  - field position(Tuple DataSet only)
  - Case Class(Case Classes only)

3. Transfamation-inner join(等值连接)

- Join 之 Default Join：只join不做其他转换。

Default join生成一个包含两个字段的新元组DataSet。每个元组在第一个字段中保存第一个输入DataSet的连接元素，在第二个字段中保存第二个输入DataSet的匹配元素。

```
//tuple2<用户id，用户姓名>
ArrayList<Tuple2<Integer, String>> data1 = new ArrayList<>();
data1.add(new Tuple2<>(1,"zs"));
data1.add(new Tuple2<>(2,"ls"));
data1.add(new Tuple2<>(3,"ww"));

//tuple2<用户id，用户所在城市>
ArrayList<Tuple2<Integer, String>> data2 = new ArrayList<>();
data2.add(new Tuple2<>(1,"beijing"));
data2.add(new Tuple2<>(2,"shanghai"));
data2.add(new Tuple2<>(3,"guangzhou"));


DataSource<Tuple2<Integer, String>> input1 = env.fromCollection(data1);
DataSource<Tuple2<Integer, String>> input2 = env.fromCollection(data2);

DataSet<Tuple2<Tuple2<Integer, String>,Tuple2<Integer, String>>> joinedData=input1.join(input2)
        .where( ...fields: 0)
        .equalTo( ...fields: 0);

joinedData.print();
```

3. Transfamation-inner join(等值连接)

- Join with Join Function 和 FlatJoinFunction
  - 在Default join生成的二元组上with一个Join Function/ FlatJoinFunction处理连接后的元组
  - Join Function处理连接元组的每条数据都返回一条数据，而FlatJoinFunction会返回n条数据(n可以为0)，类比map和flatmap。

```
    //tuple2<用户id，用户姓名>
    ArrayList<Tuple2<Integer, String>> data1 = new ArrayList<>();
    data1.add(new Tuple2<>(1,"zs"));
    data1.add(new Tuple2<>(2,"ls"));
    data1.add(new Tuple2<>(3,"ww"));
    data1.add(new Tuple2<>(4,"zl"));
    data1.add(new Tuple2<>(5,"mq"));

    //tuple2<用户id，用户所在城市>
    ArrayList<Tuple2<Integer, String>> data2 = new ArrayList<>();
    data2.add(new Tuple2<>(1,"beijing"));
    data2.add(new Tuple2<>(2,"shanghai"));
    data2.add(new Tuple2<>(3,"guangzhou"));


    DataSource<Tuple2<Integer, String>> input1 = env.fromCollection(data1);
    DataSource<Tuple2<Integer, String>> input2 = env.fromCollection(data2);

    DataSet<UserInfo> joinedData=input1.join(input2)
            .where( ...fields: 0)
            .equalTo( ...fields: 0)
            .with(new UserInfoJoinFun());

    joinedData.print();
}

public static class UserInfoJoinFun implements JoinFunction<Tuple2<Integer, String>,Tuple2<Integer, String>,UserInfo> {

    @Override
    public UserInfo join(Tuple2<Integer, String> first, Tuple2<Integer, String> second) throws Exception {
        return UserInfo.of(first.f0,first.f1,second.f1);
    }
}
```

3. Transfamation-OuterJoin

- OuterJoin对两个数据集执行left, right, full outer join。
- Join之后可以使用JoinFunction、 FlatJoinFunction对join后的数据对进行处理。

```
// some POJO
public class Rating {
  public String name;
  public String category;
  public int points;
}

// Join function that joins a custom POJO with a Tuple
public class PointAssigner
         implements JoinFunction<Tuple2<String, String>, Rating, Tuple2<String, Integer>> {

  @Override
  public Tuple2<String, Integer> join(Tuple2<String, String> movie, Rating rating) {
    // Assigns the rating points to the movie.
    // NOTE: rating might be null
    return new Tuple2<String, Double>(movie.f0, rating == null ? -1 : rating.points;
  }
}

DataSet<Tuple2<String, String>> movies = // [...]
DataSet<Rating> ratings = // [...]
DataSet<Tuple2<String, Integer>>
            moviesWithPoints =
            movies.leftOuterJoin(ratings)

                  // key of the first input
                  .where("f0")

                  // key of the second input
                  .equalTo("name")

                  // applying the JoinFunction on joining pairs
                  .with(new PointAssigner());
```

4. DataSink

- writeAsText() / TextOutputFormat：以字符串的形式逐行写入元素。字符串是通过调用每个元素的toString()方法获得的
- writeAsFormattedText() / TextOutputFormat：以字符串的形式逐行写入元素。字符串是通过为每个元素调用用户定义的format()方法获得的。
- writeAsCsv(...) / CsvOutputFormat：将元组写入以逗号分隔的文件。行和字段分隔符是可配置的。每个字段的值来自对象的toString()方法。
- print() / printToErr() / print(String msg) / printToErr(String msg) ()(注: 线上应用杜绝使用，采用抽样打印或者日志的方式)
- write() / FileOutputFormat
- output()/ OutputFormat：通用的输出方法，用于不基于文件的数据接收器(如将结果存储在数据库中)。

5. 广播变量

- 跟DataStream中的Broadcast State有些许类似
- 广播变量允许您将数据集提供给的operator所有并行实例，该数据集将作为集合在operator中进行访问。
- 注意：由于广播变量的内容保存在每个节点的内存中，因此它不应该太大。

```
// 1. The DataSet to be broadcast
DataSet<Integer> toBroadcast = env.fromElements(1, 2, 3);

DataSet<String> data = env.fromElements("a", "b");

data.map(new RichMapFunction<String, String>() {
    @Override
    public void open(Configuration parameters) throws Exception {
      // 3. Access the broadcast DataSet as a Collection
      Collection<Integer> broadcastSet = getRuntimeContext().getBroadcastVariable("broadcastSetName");
    }


    @Override
    public String map(String value) throws Exception {
        ...
    }
}).withBroadcastSet(toBroadcast, "broadcastSetName"); // 2. Broadcast the DataSet
```

6. 分布式缓存

Flink提供了类似于Apache Hadoop的分布式缓存，可以让并行用户函数实例本地化的访问文件。此功能可用于共享包含静态外部数据(如字典或机器学习的回归模型)的文件。

工作方式如下：程序将本地或远程文件系统(如HDFS或S3)的文件或目录作为缓存文件注册到ExecutionEnvironment中的特定名称下。当程序执行时，Flink自动将文件或目录复制到所有worker的本地文件系统。用户函数可以查找指定名称下的文件或目录，并从worker的本地文件系统访问它

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

// register a file from HDFS
env.registerCachedFile("hdfs:///path/to/your/file", "hdfsFile")

// register a local executable file (script, executable, ...)
env.registerCachedFile("file:///path/to/exec/file", "localExecFile", true)

// define your program and execute
...
DataSet<String> input = ...
DataSet<Integer> result = input.map(new MyMapper());
...
env.execute();
```

6. 分布式缓存

```
// extend a RichFunction to have access to the RuntimeContext
public final class MyMapper extends RichMapFunction<String, Integer> {

    @Override
    public void open(Configuration config) {

      // access cached file via RuntimeContext and DistributedCache
      File myFile = getRuntimeContext().getDistributedCache().getFile("hdfsFile");
      // read the file (or navigate the directory)
      ...
    }

    @Override
    public Integer map(String value) throws Exception {
      // use content of cached file
      ...
    }
}
```

7. 参数传递-Constructor方式

- 可以使用constructor或withParameters(Configuration)方法将参数传递给函数。参数被序列化为函数对象的一部分，并传送到所有并行任务实例。

```
DataSet<Integer> toFilter = env.fromElements(1, 2, 3);

toFilter.filter(new MyFilter(2));

private static class MyFilter implements FilterFunction<Integer> {

  private final int limit;

  public MyFilter(int limit) {
    this.limit = limit;
  }

  @Override
  public boolean filter(Integer value) throws Exception {
    return value > limit;
  }
}
```

7.参数传递-withParameters方式

```
DataSet<Integer> toFilter = env.fromElements(1, 2, 3);

Configuration config = new Configuration();
config.setInteger("limit", 2);

toFilter.filter(new RichFilterFunction<Integer>() {
    private int limit;

    @Override
    public void open(Configuration parameters) throws Exception {
      limit = parameters.getInteger("limit", 0);
    }

    @Override
    public boolean filter(Integer value) throws Exception {
      return value > limit;
    }
}).withParameters(config);
```

7.参数传递-ExecutionConfig方式(全局参数)

```
Configuration conf = new Configuration();
conf.setString("mykey","myvalue");
final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
env.getConfig().setGlobalJobParameters(conf);
```

```
public static final class Tokenizer extends RichFlatMapFunction<String, Tuple2<String, Integer>> {

    private String mykey;
    @Override
    public void open(Configuration parameters) throws Exception {
      super.open(parameters);
      ExecutionConfig.GlobalJobParameters globalParams = getRuntimeContext().getExecutionConfig().getGlobalJobPara
meters();
      Configuration globConf = (Configuration) globalParams;
      mykey = globConf.getString("mykey", null);
    }
    // ... more here ...
```

Flink DataSet 容错

- 重启策略-固定延时
- 重启策略-失败率

批处理容错

- 批处理(DataSet API)容错是基于失败重试实现的。
- Flink支持不同的重启策略，这些策略控制在出现故障时如何重新启动job

Restart Strategy	配置项	默认值	说明
固定延迟 (Fixed delay)	restart-strategy: fixed-delay		如果超过最大尝试次数，作业最终会失败。在连续两次重启尝试之间等待固定的时间。
	restart-strategy.fixed-delay.attempts: 3	1	
	restart-strategy.fixed-delay.delay: 10 s	akka.ask.timeout	
失败率 (Failure rate)	restart-strategy: failure-rate		在失败后重新启动作业，但是当超过故障率(每个时间间隔的故障)时，作业最终会失败。在连续两次重启尝试之间等待固定的时间。
	restart-strategy.failure-rate.max-failures-per-interval: 3	1	
	restart-strategy.failure-rate.failure-rate-interval: 5 min	1 minute	
	restart-strategy.failure-rate.delay: 10 s	akka.ask.timeout	
无重启 (No restart)	restart-strategy: none		

Flink DataSet Connectors

- 文件系统
- Flink与Hadoop兼容性
- Flink读写HBase

批处理Connector之文件系统访问

- 为了从文件系统读取数据，Flink内置了对以下文件系统的支持:

文件系统	Scheme	备注
HDFS	hdfs://	支持所有HDFS版本
S3	s3://	通过Hadoop文件系统实现支持
MapR	maprfs://	用户必须手动将所需的jar文件放到lib目录中
Alluxio	alluxio://	通过Hadoop文件系统实现支持

- 注意：Flink允许用户使用实现org.apache.hadoop.fs.FileSystem接口的任何文件系统。

S3、 Google Cloud Storage Connector for Hadoop、 Alluxio、 XtreemFS、 FTP via Hftp and many more

Hadoop 兼容性：使用Hadoop的Input/OutputFormat wrappers连接到其他系统

- Flink与Apache Hadoop MapReduce接口兼容，因此允许重用Hadoop MapReduce实现的代码：
  - 使用Hadoop Writable data type
  - 使用任何Hadoop InputFormat作为DataSource(flink内置HadoopInputFormat)
  - 使用任何Hadoop OutputFormat作为DataSink(flink内置HadoopOutputFormat)
  - 使用Hadoop Mapper作为FlatMapFunction
  - 使用Hadoop Reducer作为GroupReduceFunction
- 引入依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-hadoop-compatibility_2.11</artifactId>
    <version>1.8.0</version>
</dependency>
```

读写HBase实例

- 引入依赖

```
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-shaded-hadoop2</artifactId>
        <version>${flink.version}</version>
        <!--<scope>provided</scope>-->
</dependency>
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-hadoop-compatibility_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
</dependency>
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-hbase_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
</dependency>
```

读写HBase实例

- 准备工作：

➢ 安装hbase(zookeeper、hdfs)

➢ 在hbase中创建namespace和table

```
create_namespace 'learing_flink‘
create 'learing_flink:users',{NAME=>'F',BLOCKCACHE=>true,BLOOMFILTER=>'ROW',DATA_BLOCK_ENCODING =>
'PREFIX_TREE', BLOCKSIZE => '65536'}
```

➢ 写数据到hbase(使用HadoopOutputFormat)

➢ 从Hbase读取数据(使用HadoopInputFormat)

➢ 常用命令

```
//查看数据
scan 'learing_flink:users‘
//count
count 'learing_flink:users‘
//清空表数据
truncate_preserve 'learing_flink:users'
```

Flink SQL

- 概述
- 流处理
- Connector
- Table API
- SQL

1.Table API & SQL介绍

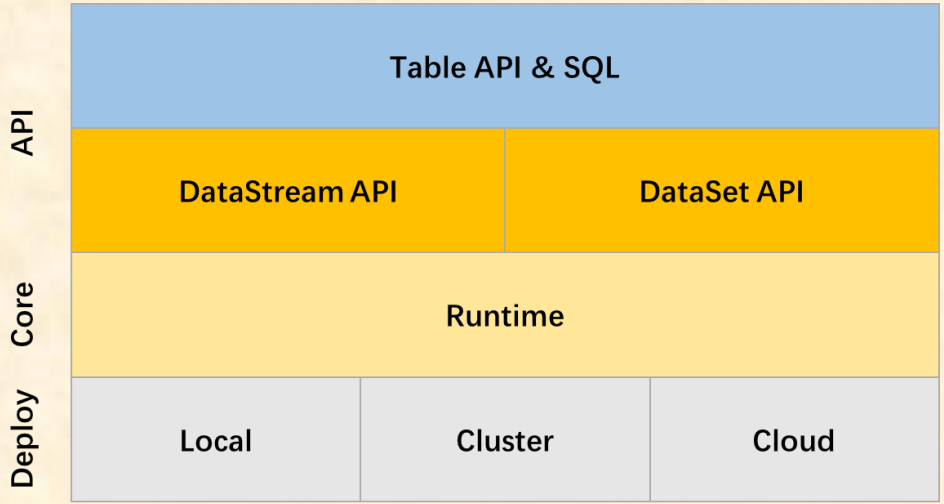
- Apache Flink具有两个关系API：表API和SQL，用于统一流和批处理。 Table API是Scala和Java的语言集成查询API，查询允许组合关系运算符，例如过滤和连接。Flink SQL支持标准的SQL语法。

- Table API和SQL接口彼此集成，Flink的DataStream和DataSet API亦是如此。您可以轻松地在基于API构建的所有API和库之间切换。

- 注意，到目前最新版本为止，Table API和SQL还有很多功能正在开发中。 并非[Table API，SQL]和[stream，batch]输入的每种组合都支持所有操作。

1. 为什么需要Table API & SQL

- Table API 是一种关系型API，类 SQL 的API，用户可以像操作表一样地操作数据，非常的直观和方便。
- SQL 作为一个"人所皆知"的语言，如果一个引擎提供 SQL，它将很容易被人们接受。这已经是业界很常见的现象了。
- Table & SQL API 还有另一个职责，就是流处理和批处理统一的API层。



1.Table API & SQL开发所需依赖

- Table API所需依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner_2.11</artifactId>
  <version>1.8.0</version>
</dependency>
```

- 对于批处理查询（java），需要添加：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java-bridge_2.11</artifactId>
  <version>1.8.0</version>
</dependency>
```

- 对于流式查询，需要添加：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.11</artifactId>
  <version>1.8.0</version>
</dependency>
```

1.Table API & SQL编程套路-stream sql

```
// 创建一个TableEnvironment
StreamTableEnvironment sTableEnv = StreamTableEnvironment.create(sEnv);
//读取数据源
DataStream<String> ds1 = sEnv.readTextFile( filePath: "E:\\工作资料\\大讲台资料
DataStream<User> ds2 = ds1.map(new MapFunction<String, User>() {
    @Override
    public User map(String line) throws Exception {
        String[] split = line.split( regex: ",");
        return new User(split[0], split[1]);
    }
});
//将DataStream转化成Table
Table table = sTableEnv.fromDataStream(ds2);
//注册表,user1
sTableEnv.registerTable( name: "user1",table);
//获取表中所有信息
Table rs = sTableEnv.sqlQuery("select * from user1").select("name");
//将表转化成DataStream
DataStream<String> dataStream = sTableEnv.toAppendStream(rs, String.class);
```

1.Table API & SQL编程套路-batch sql

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
// 创建一个TableEnvironment
BatchTableEnvironment tEnv = BatchTableEnvironment.create(env);
//读取数据源
DataSet<String> ds1 = env.readTextFile( filePath: "E:\\工作资料\\大讲台资料汇

DataSet<User> ds2 = ds1.map(new MapFunction<String, User>() {
    @Override
    public User map(String line) throws Exception {
        String[] split = line.split( regex: ",");
        return new User(split[0], split[1]);
    }
});
//将ds2注册为表user1
tEnv.registerDataSet( name: "user1",ds2, fields: "uid,name");
//查询表数据
Table table = tEnv.sqlQuery("select * from user1").select("name");
//table转换为DataSet
DataSet<String> dataSet = tEnv.toDataSet(table, String.class);
dataSet.print();
```

1.Table API & SQL编程套路-batch table

```
//获取执行环境
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
// 创建一个TableEnvironment
BatchTableEnvironment tEnv = BatchTableEnvironment.create(env);
//读取数据源
DataSet<String> ds = env.readTextFile( filePath: "E:\\工作资料\\大讲台资料汇总
DataSet<User> ds2 = ds.map(new MapFunction<String, User>() {
    @Override
    public User map(String line) throws Exception {
        String[] split = line.split( regex: ",");
        return new User(split[0], split[1]);
    }
});
//将ds2注册为表user
Table user = tEnv.fromDataSet(ds2);
//查询表数据
Table table = user.select("name");
//table转换为DataSet
DataSet<String> dataSet = tEnv.toDataSet(table, String.class);
dataSet.print();
```

2.Table API & SQL 流处理概述

Flink 的 Table API 和SQL支持是用于批处理和流处理的统一API。这意味着 Table API 和SQL查询具有相同的语义，无论它们的输入是有界批量输入还是无界流输入。因为关系代数（relational algebra）和SQL最初是为批处理而设计的，所以对于无界流输入的关系查询不像有界批输入上的关系查询那样容易理解。

2. 流数据上的关系查询

SQL和 Relational algebra 并没有考虑到流数据。因此，在关系代数（和SQL）和流处理之间有一些概念上的差距。

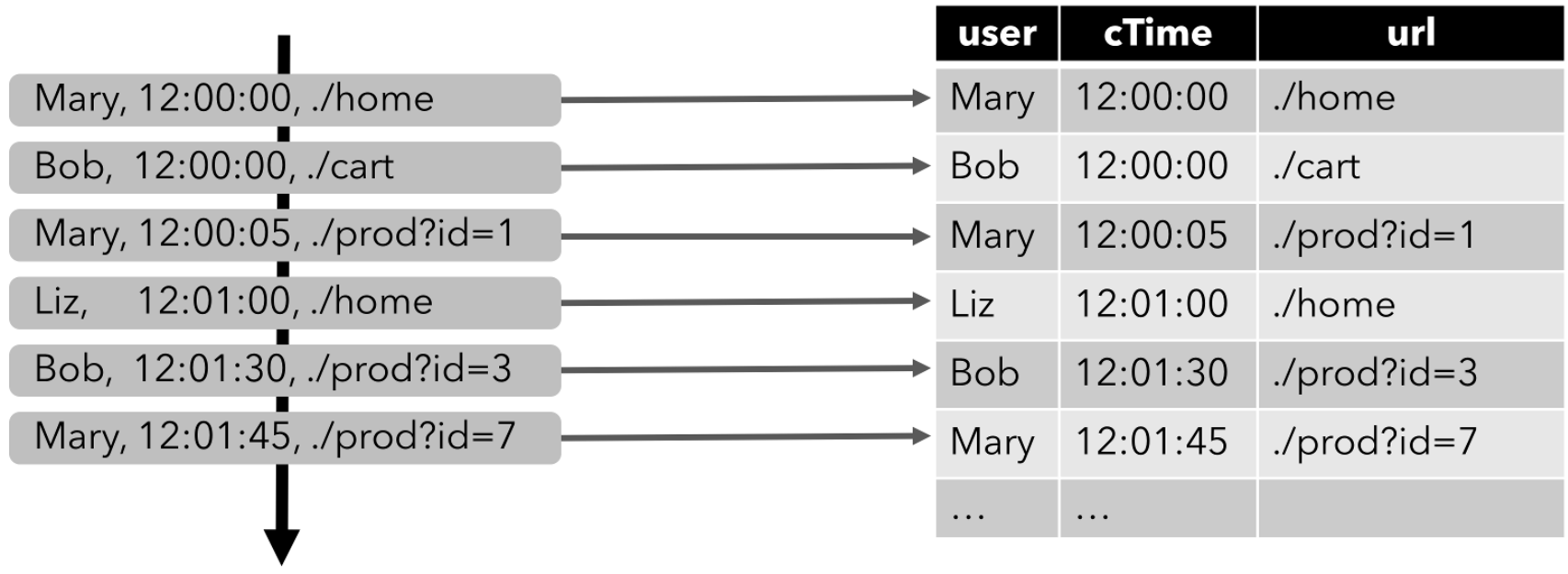
关系代数/ SQL	流处理
1.关系（或表）是有界的（多）元组的集合	1.流是无限的元组序列
2.对批处理数据执行的查询（例如，关系数据库中的表）可以访问完整的输入数据	2.流式查询在启动时无法访问所有数据，必须等待流式传输数据
3.批处理查询在生成固定大小的结果后终止	3.流式查询会根据收到的记录不断更新其结果，并且永远不会完成

2. 动态表和连续查询

动态表（Dynamic table）是 Flink Table API 和SQL支持流数据的核心概念。与表示批处理数据的静态表（static table）相比，动态表会随时间而变化，并且可以像静态批处理表一样查询。查询动态表会生成连续查询（Continuous Query）。连续查询永远不会终止并生成动态表作为结果。查询不断更新其结果表以反映其输入表的更改。

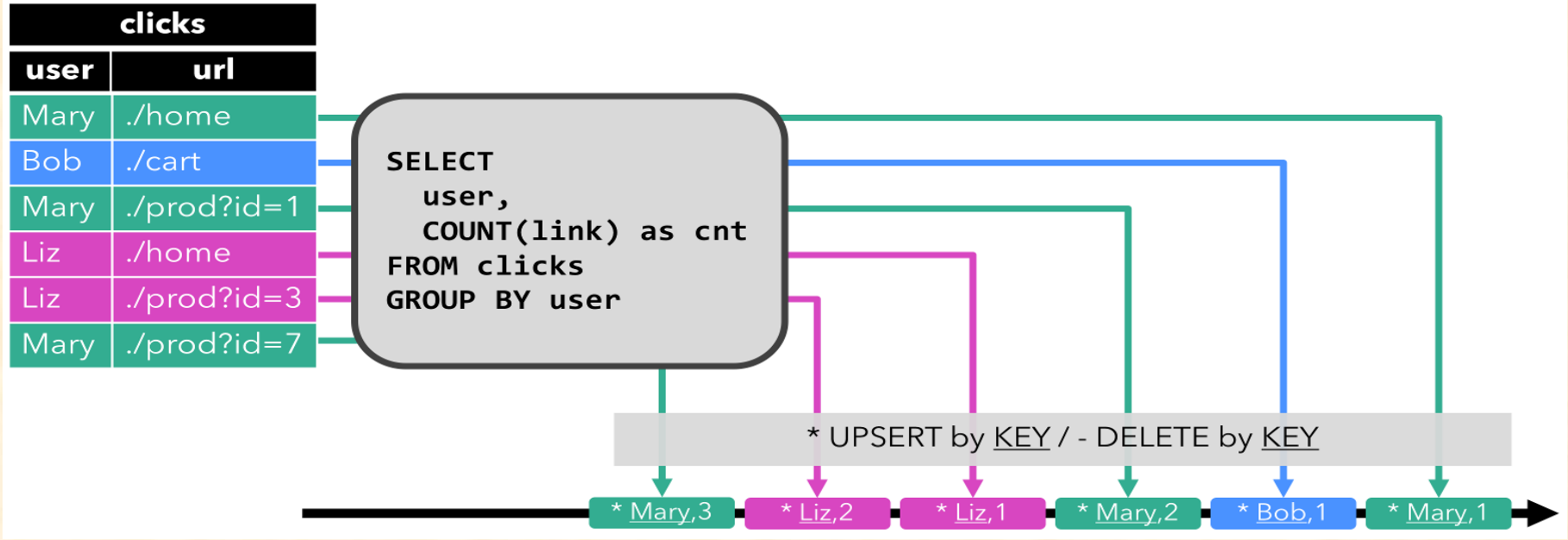
2. 在流上定义表

为了使用关系查询处理流，必须将其转换为表。从概念上讲，流的每个记录都被解释为对结果表的 INSERT 修改。下图显示了点击事件流（左侧）如何转换为表（右侧）。随着更多的点击事件的插入，结果表不断增长。

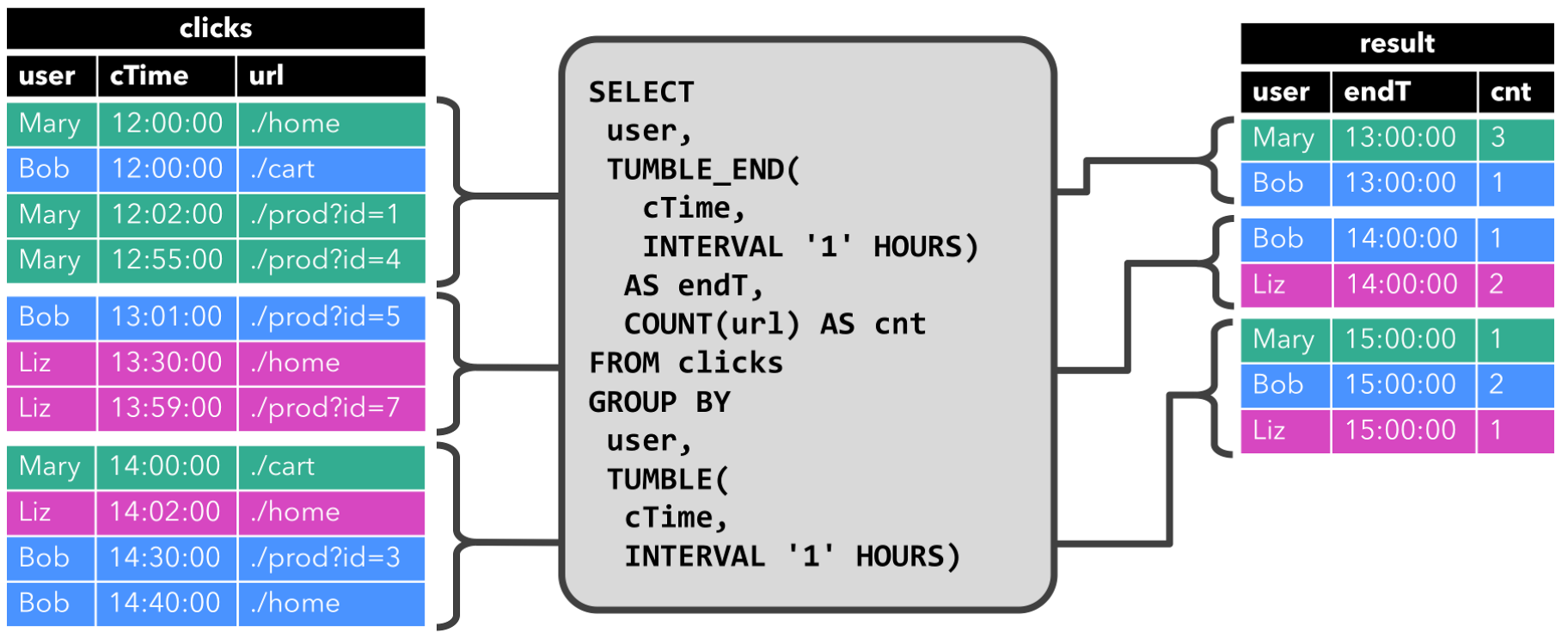


2. 连续查询

在动态表上进行连续查询，并生成新的动态表。与批查询相反，连续查询不会停止更新其结果表。在任何时间点，连续查询的结果在语义上等同于在输入表的快照上以批处理模式执行的相同查询的结果。



2. 连续查询


clicks
user cTime url
Mary 12:00:00 ./home
Bob 12:00:00 ./cart
Mary 12:02:00 ./prod?id=1
Mary 12:55:00 ./prod?id=4
Bob 13:01:00 ./prod?id=5
Liz 13:30:00 ./home
Liz 13:59:00 ./prod?id=7
Mary 14:00:00 ./cart
Liz 14:02:00 ./home
Bob 14:30:00 ./prod?id=3
Bob 14:40:00 ./home
SELECT
user,
TUMBLE_END(
cTime,
INTERVAL '1' HOURS)
AS endT,
COUNT(url) AS cnt
FROM clicks
GROUP BY
user,
TUMBLE(
cTime,
INTERVAL '1' HOURS)
result
user endT cnt
Mary 13:00:00 3
Bob 13:00:00 1
Bob 14:00:00 1
Liz 14:00:00 2
Mary 15:00:00 1
Bob 15:00:00 2
Liz 15:00:00 1

2. 表转换到流

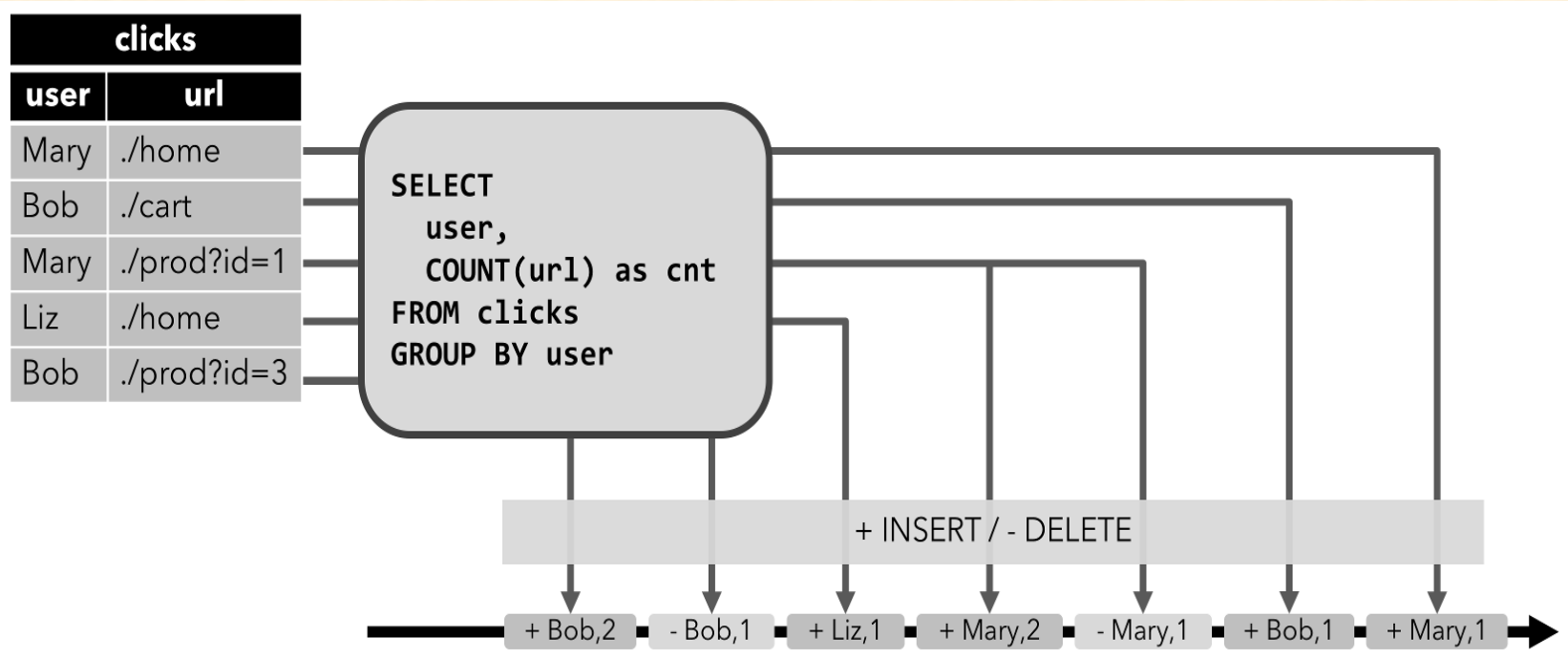
流查询的结果表将被动态更新，即，随着新记录到达查询的输入流，它也在发生变化。因此，将这样的动态查询转换成的DataStream需要对表的更新进行编码。

将表转换为数据流有两种模式:

- Append-only Mode：只有在动态Table仅通过INSERT更改修改时才能使用此模式，即它仅附加，并且以前发出的结果永远不会更新。如果更新或删除操作使用追加模式会失败报错

2. 表转换到流

- Retract Mode：始终可以使用此模式。返回值是boolean类型。它用true或false来标记数据的插入和撤回，返回true代表数据插入，false代表数据的撤回。



3.Flink SQL Connector

Flink的表API和SQL程序可以连接到其他外部系统来读写批处理表和流表。Table source提供对存储在外部系统(如数据库、键值存储、消息队列或文件系统)中的数据的访问。Table Sink将表发送到外部存储系统。

Connectors

Name	Version	Maven dependency	SQL Client JAR
Filesystem		Built-in	Built-in
Elasticsearch	6	`flink-connector-elasticsearch6`	Download
Apache Kafka	0.8	`flink-connector-kafka-0.8`	Not available
Apache Kafka	0.9	`flink-connector-kafka-0.9`	Download
Apache Kafka	0.10	`flink-connector-kafka-0.10`	Download
Apache Kafka	0.11	`flink-connector-kafka-0.11`	Download
Apache Kafka	0.11+ (`universal`)	`flink-connector-kafka`	Download

Formats

Name	Maven dependency	SQL Client JAR
Old CSV (for files)	Built-in	Built-in
CSV (for Kafka)	`flink-csv`	Download
JSON	`flink-json`	Download
Apache Avro	`flink-avro`	Download

3.Flink SQL Connector-Kafka

kafka依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka-0.10_2.11</artifactId>
  <version>${flink.version}</version>
</dependency>
```

Json依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-json</artifactId>
  <version>${flink.version}</version>
</dependency>
```

示例数据：

{"userId":1119,"day":"2017-03-02","begintime":1488326400000,"endtime":1488327000000,"data":[{"package":"com.browser","activetime":120000}]}

4.Flink Table API

Table API 是一个 Scala 和 Java 的语言集成查询API，是基于 Table类。Table类代表了一个流或者批表，并提供方法来使用关系型操作。这些方法返回一个新的 Table 对象，这个新的 Table 对象代表着输入的 Table 应用关系型操作后的结果。

```
//将ds2注册为表table
tEnv.registerDataSet( name: "Order",ds2);
//扫描注册的 Orders 表
Table order = tEnv.scan( ...tablePath: "Order");
//分组计算每个用户的订单总金额
Table rs = order.filter("orderStatus==1")
        .groupBy( fields: "userId")
        .select( fields: "userId,goodsMoney.sum as allmoney");
//table 转换为DataSet
DataSet<Row> rowDataSet = tEnv.toDataSet(rs, Row.class);
rowDataSet.print();
```

5.Flink SQL

Flink SQL 集成是基于 Apache Calcite，Apache Calcite 实现了标准的SQL。

```
//读取数据源
DataSet<String> ds = env.readTextFile( filePath: "E:\\工作资料\\大讲台资料汇总\\大讲台教学相关\\大讲台课
DataSet<TableAPIDemo.Order> ds2 = ds.map(new MapFunction<String, TableAPIDemo.Order>() {
    @Override
    public TableAPIDemo.Order map(String line) throws Exception {
        String[] split = line.split( regex: ",");
        return new TableAPIDemo.Order(split[0], split[1], Integer.parseInt(split[2]), Double.parseDoub
    }
});
//将ds2注册为表table
tEnv.registerDataSet( name: "orders",ds2);
//扫描注册的 Orders 表
Table order = tEnv.sqlQuery("select userId,sum(goodsMoney) as allmoney from orders where orderStatus=1
//table 转换为DataSet
DataSet<Row> rowDataSet = tEnv.toDataSet(order, Row.class);
rowDataSet.print();
```

Flink 实时数仓项目实战

- 项目需求
- 架构设计
- 数据同步
- 实时计算
- 数据可视化
- 多维分析

项目需求

- 某电商平台数据中心经过对Flink的调研学习，决定对原有平台进行升级改造：

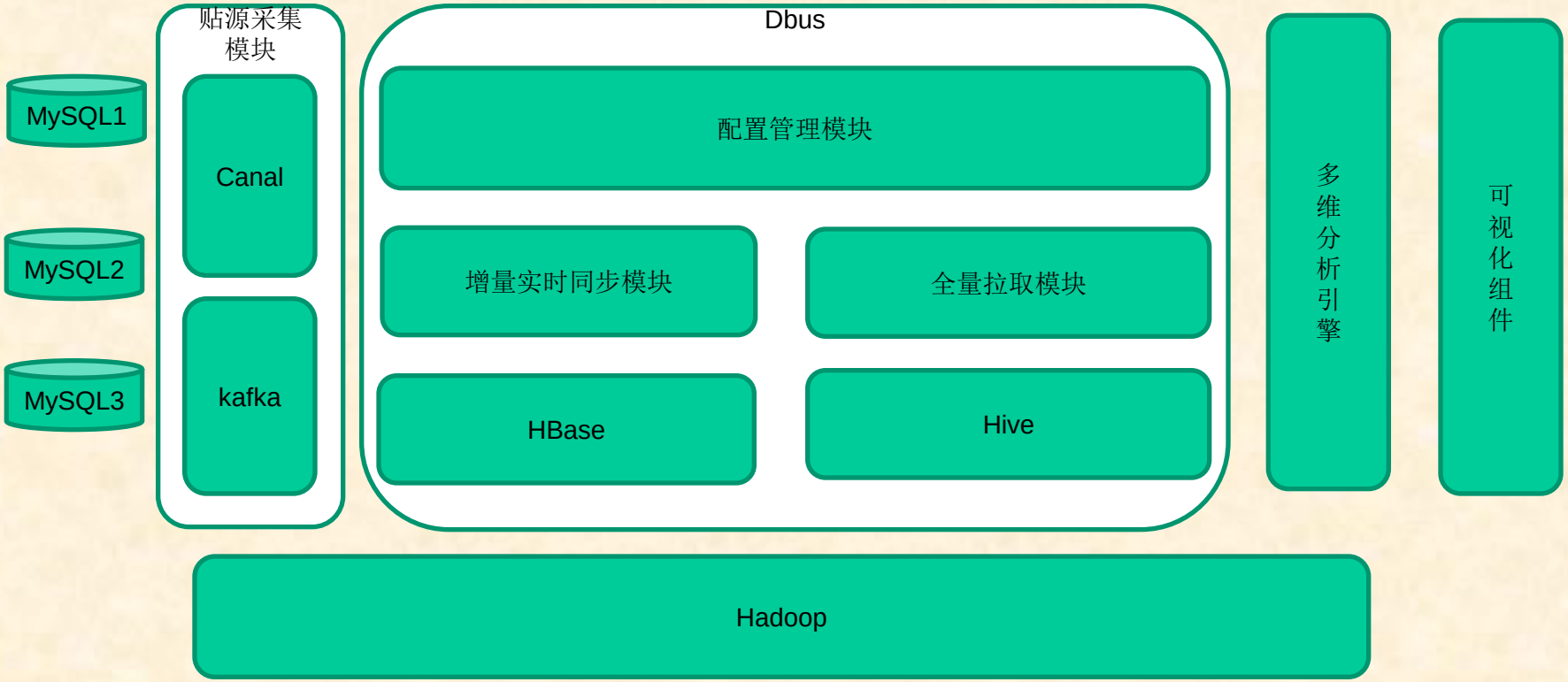
➢ 数据同步
  - 业务库数据实时增量同步到数据仓库
  - 业务数据全量同步到数据仓库

➢ 实时分析指标计算与存储

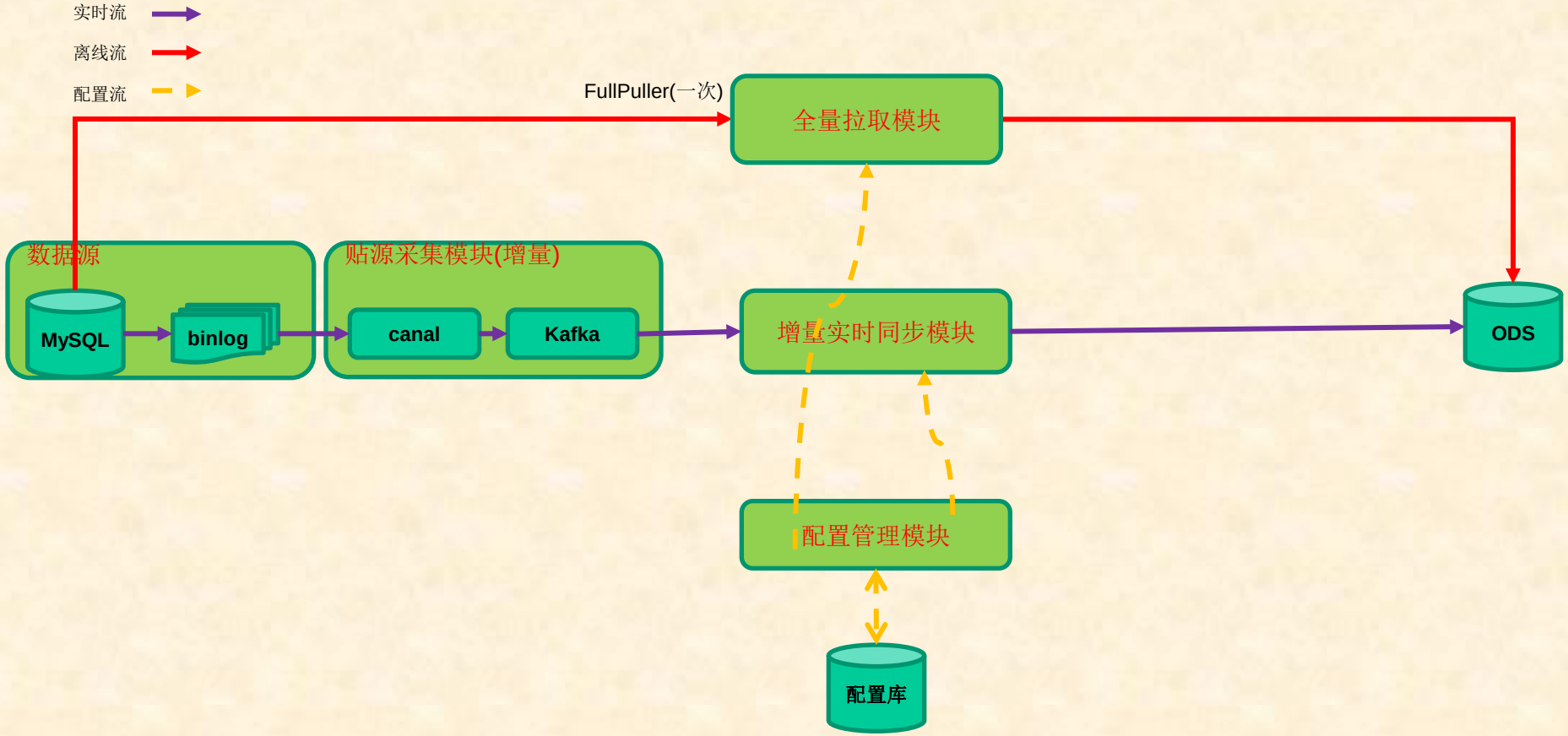
➢ 数据可视化

➢ 多维分析

架构设计


MySQL1
MySQL2
MySQL3
贴源采集模块
Canal
kafka
Dbus
配置管理模块
增量实时同步模块
全量拉取模块
HBase
Hive
多维分析引擎
可视化组件
Hadoop

数据同步-解决方案


实时流
离线流
配置流
FullPuller(一次)
全量拉取模块
数据源
MySQL
binlog
贴源采集模块(增量)
canal
Kafka
增量实时同步模块
ODS
配置管理模块
配置库

数据库建模

- MySQL数据库建模
  - 商品表
  - 订单表
  - flow表
  - 表结构见mysql 表结构.txt

- HBase数据库建模
  - 商品表
  - 订单表
  - 表结构见hbase表结构.txt

数据模拟产生

- 商品数据模拟产生
- 订单数据模拟产生
- 代码解析

全量拉取模块

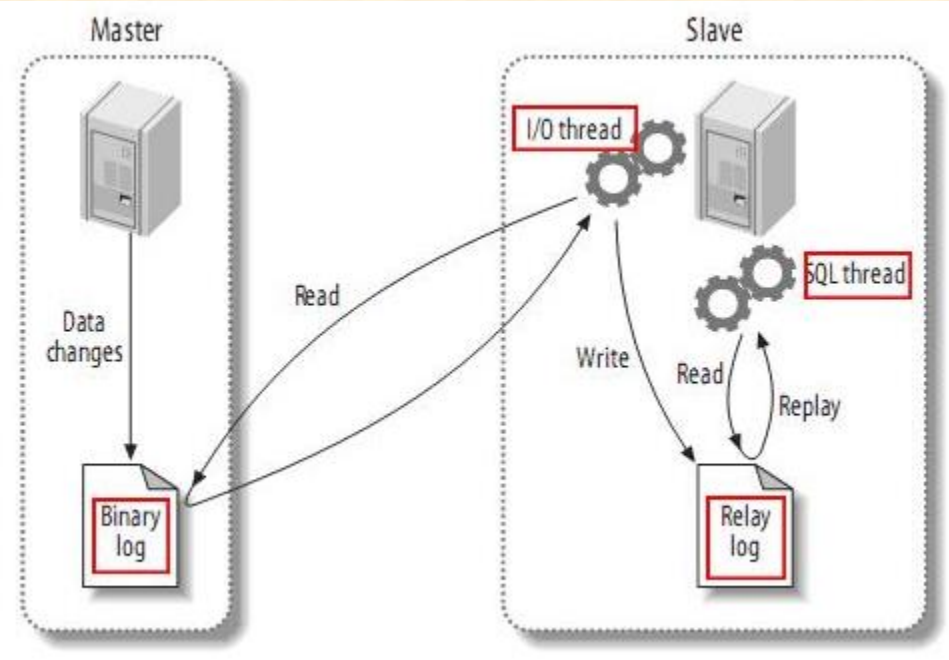
- 为什么需要全量拉取模块
- Flink版Sqoop(flink-jdbc)
- 代码解析

Canal产生背景

早期，阿里巴巴B2B公司因为存在杭州和美国双机房部署，存在跨机房同步的业务需求。不过早期的数据库同步业务，主要是基于trigger的方式获取增量变更，不过从2010年开始，阿里系公司开始逐步的尝试基于数据库的日志解析，获取增量变更进行同步，由此衍生出了增量订阅&消费的业务，从此开启了一段新纪元。

ps. 目前内部版本已经支持mysql和oracle部分版本的日志解析，当前的canal开源版本支持5.7及以下的版本(阿里内部mysql 5.7.13, 5.6.10, mysql 5.5.18和5.1.40/48)

mysql主备复制实现



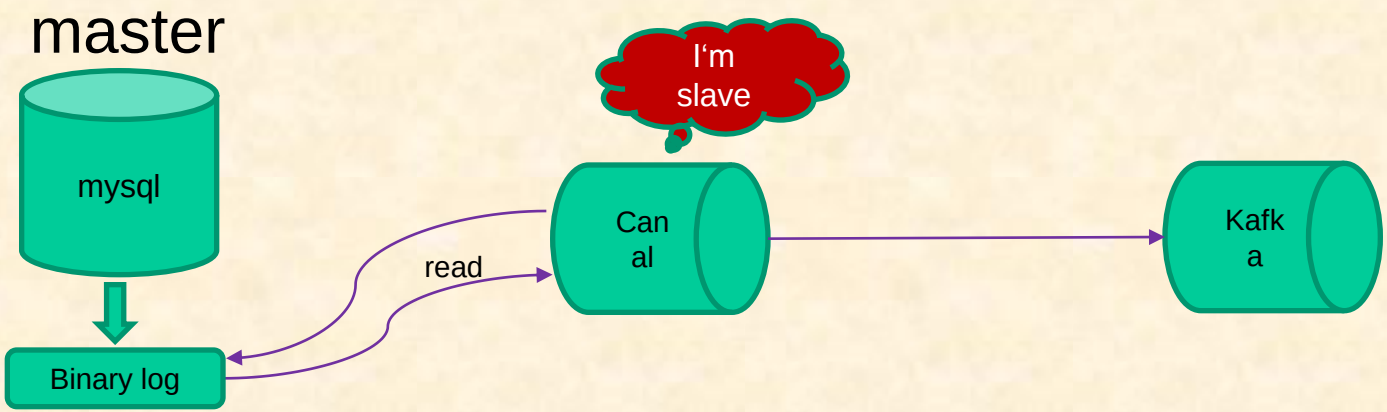
从上层来看，复制分成三步：

1.master将改变记录到二进制日志(binary log)中（这些记录叫做二进制日志事件，binary log events，可以通过show binlog events进行查看）；

2.slave将master的binary log events拷贝到它的中继日志(relay log)；

3.slave重做中继日志中的事件，将改变反映它自己的数据。

canal的工作原理



原理相对比较简单：

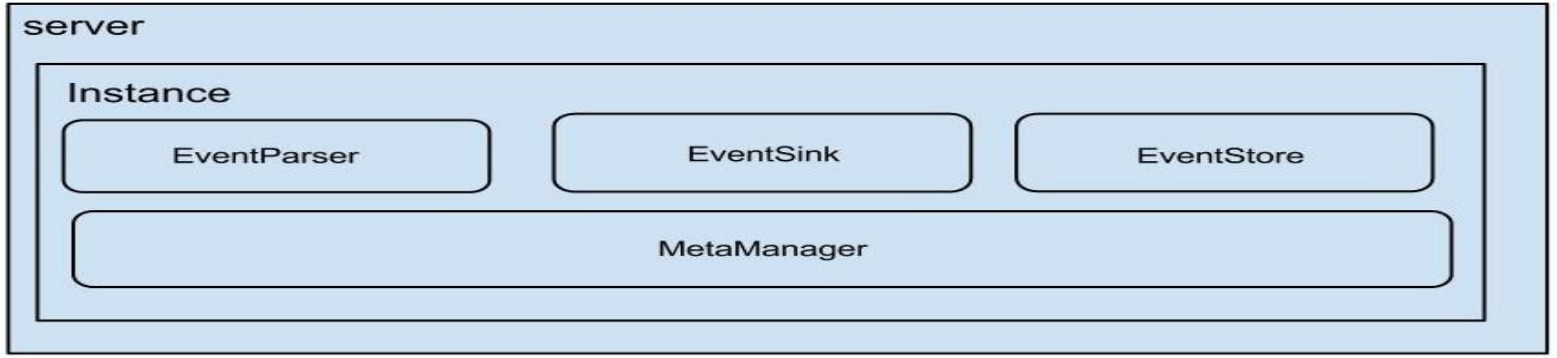
1.canal模拟mysql slave的交互协议，伪装自己为mysql slave，向mysql master发送dump协议

2.mysql master收到dump请求，开始推送binary log给slave(也就是canal)

3.canal解析binary log对象(原始为byte流)

4.canal把解析后的binary log以特性格式的消息推送到Kafka供下游消费

Canal的架构



- 说明：
  - server代表一个canal运行实例，对应于一个jvm
  - instance对应于一个数据队列 （1个server对应1..n个instance）
- instance模块：
  - eventParser（数据源接入，模拟slave协议和master进行交互，协议解析）
  - eventSink（Parser和Store链接器，进行数据过滤，加工，分发的工作）
  - eventStore（数据存储）
  - metaManager（增量订阅&消费信息管理器）

MySQL Binary Log介绍

mysql-binlog是MySQL数据库的二进制日志，用于记录用户对数据库操作的SQL语句（（除了数据查询语句）信息。如果后续我们需要配置主从数据库，如果我们需要从数据库同步主数据库的内容，我们就可以通过binlog来进行同步。

简而言之：

- mysql的binlog是多文件存储，定位一个LogEvent需要通过binlog filename + binlog position，进行定位。
- mysql的binlog数据格式，按照生成的方式，主要分为：statement-based、row-based、mixed。

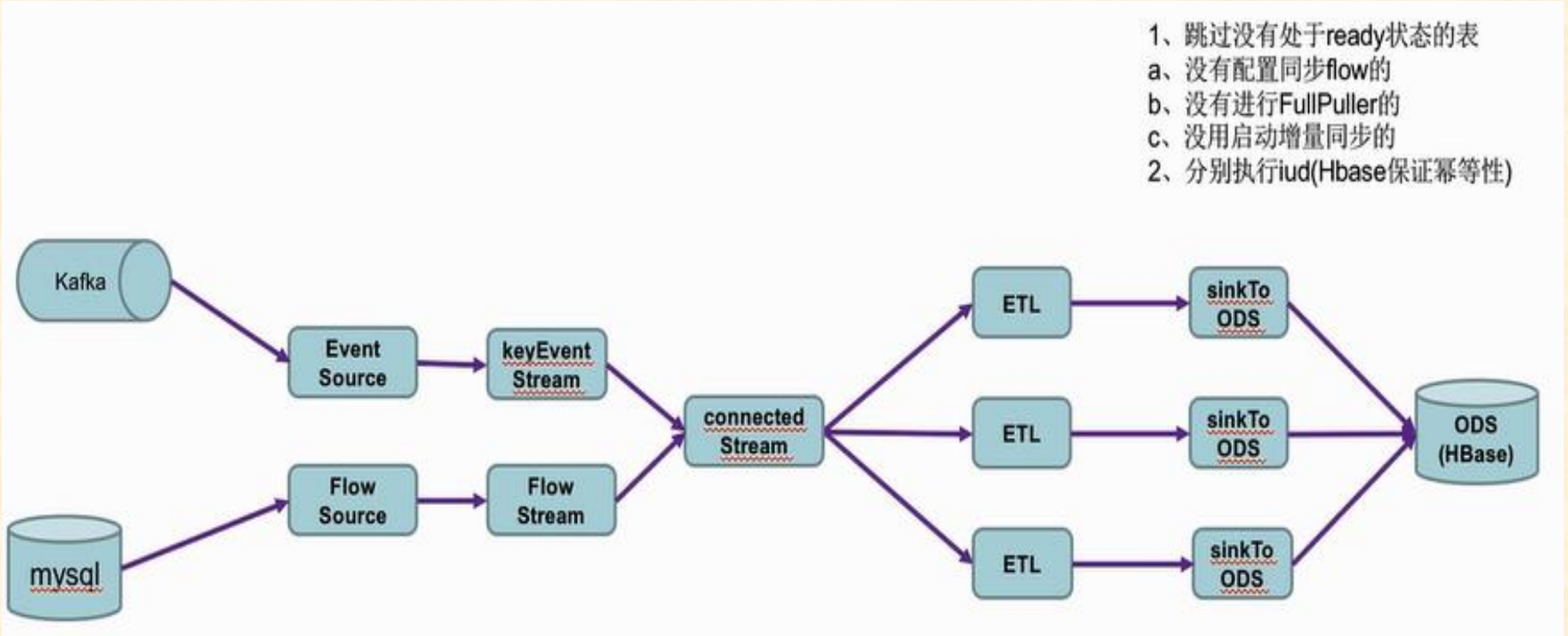
MySQL binlog格式

binlog的格式也有三种：STATEMENT、ROW、MIXED 。

- STATMENT模式：基于SQL语句的复制(statement-based replication, SBR)，每一条会修改数据的sql语句会记录到binlog中。
  - 优点：不需要记录每一条SQL语句与每行的数据变化，这样子binlog的日志也会比较少，减少了磁盘IO，提高性能。
  - 缺点：在某些情况下会导致master-slave中的数据不一致(如sleep()函数， last_insert_id()，以及user-defined functions(udf)等会出现问题)
- 基于行的复制(row-based replication, RBR)：不记录每一条SQL语句的上下文信息，仅需记录哪条数据被修改了，修改成了什么样子了。
  - 优点：不会出现某些特定情况下的存储过程或function、或trigger的调用和触发无法被正确复制的问题。
  - 缺点：会产生大量的日志，尤其是alter table的时候会让日志暴涨。
- 混合模式复制(mixed-based replication, MBR)：以上两种模式的混合使用，一般的复制使用STATEMENT模式保存binlog，对于STATEMENT模式无法复制的操作使用ROW模式保存binlog，MySQL会根据执行的SQL语句选择日志保存方式。

目前canal支持所有模式的增量订阅(但配合同步时，因为statement只有sql，没有数据，无法获取原始的变更日志，所以一般建议为ROW模式)

增量实时同步模块


1、跳过没有处于ready状态的表
a、没有配置同步flow的
b、没有进行FullPuller的
c、没用启动增量同步的
2、分别执行iud(Hbase保证幂等性)
Kafka
mysql
Event Source
Flow Source
keyEvent Stream
Flow Stream
connected Stream
ETL
ETL
ETL
sinkTo ODS
sinkTo ODS
sinkTo ODS
ODS (HBase)

增量实时同步模块难点问题思考

- MQ顺序性问题
- 落地存储幂等性问题（hbase）
- binlog消息顺序性（canal保证）
  - Kafka单分区
  - Flink最后投递时的幂等设计
- 容错（全量拉取可以纠正增量顺序不一致问题）

配置管理模块

- 配置流
  - MySQL数据库名
  - MySQL表名
  - HBase表名
  - HBase列簇名
  - HBase rowkey
  - 等等

Thanks

QQ群：732021751
QQ或微信：84985152