# Flink-流批一体API

## 学习目标

* 了解流处理的相关概念
* 掌握FlinkDataStream-SourceOperator
* 掌握FlinkDataStream-TransformationOperator
* 掌握FlinkDataStream-SinkOperator
* 了解Flink的累加器
* 掌握Flink的广播变量
* 掌握Flink的分布式缓存

## 流处理相关概念

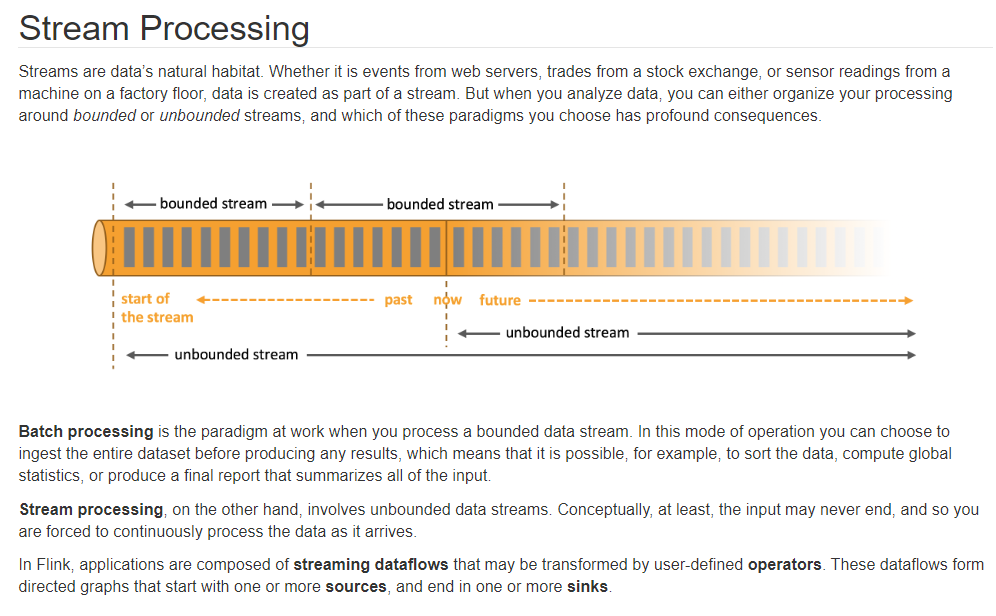
### 数据的时效性

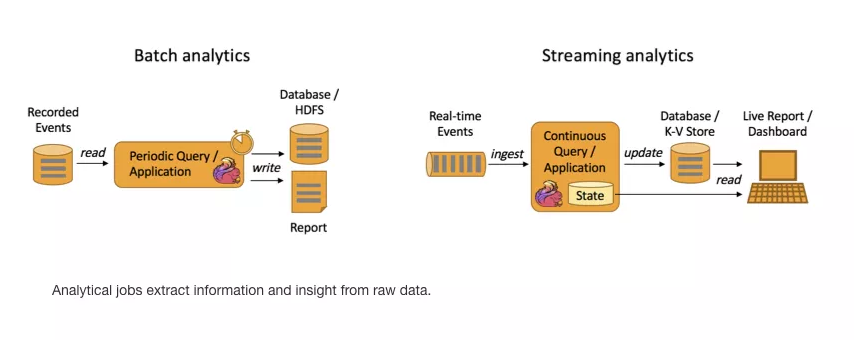
日常工作中，我们一般会先把数据存储在表，然后对表的数据进行加工、分析。既然先存储在表中，那就会涉及到时效性概念。

如果我们处理以年，月为单位的级别的数据处理，进行统计分析，个性化推荐，那么数据的的最新日期离当前有几个甚至上月都没有问题。但是如果我们处理的是以天为级别，或者一小时甚至更小粒度的数据处理，那么就要求数据的时效性更高了。比如：对网站的实时监控、对异常日志的监控，这些场景需要工作人员立即响应，这样的场景下，传统的统一收集数据，再存到数据库中，再取出来进行分析就无法满足高时效性的需求了。

### 流处理和批处理

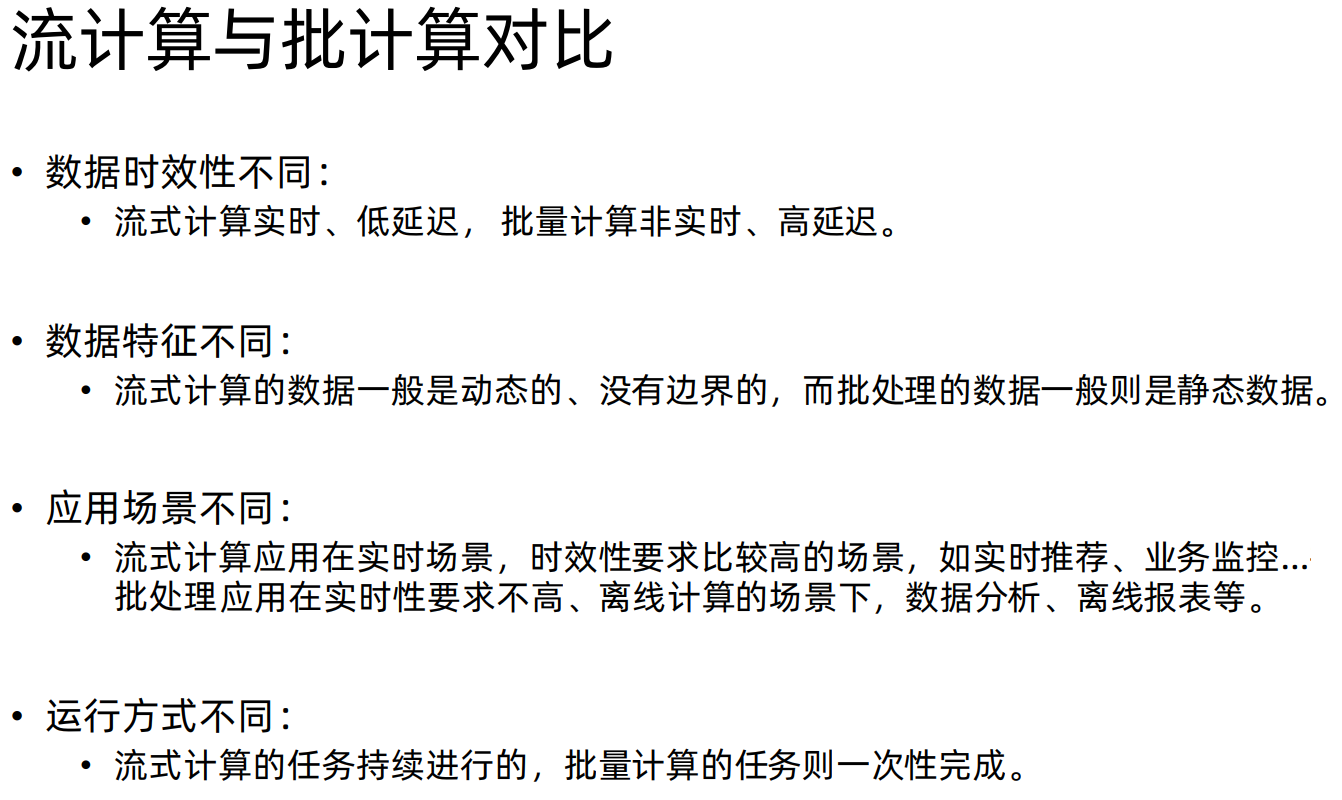
<https://ci.apache.org/projects/flink/flink-docs-release-1.12/learn-flink/>





- Batch Analytics，右边是 Streaming Analytics。批量计算: 统一收集数据->存储到DB->对数据进行批量处理，就是传统意义上使用类似于 Map Reduce、Hive、Spark Batch 等，对作业进行分析、处理、生成离线报表

- Streaming Analytics 流式计算，顾名思义，就是对数据流进行处理，如使用流式分析引擎如 Storm，Flink 实时处理分析数据，应用较多的场景如实时大屏、实时报表。



### 流批一体API

* DataStream API 支持批执行模式

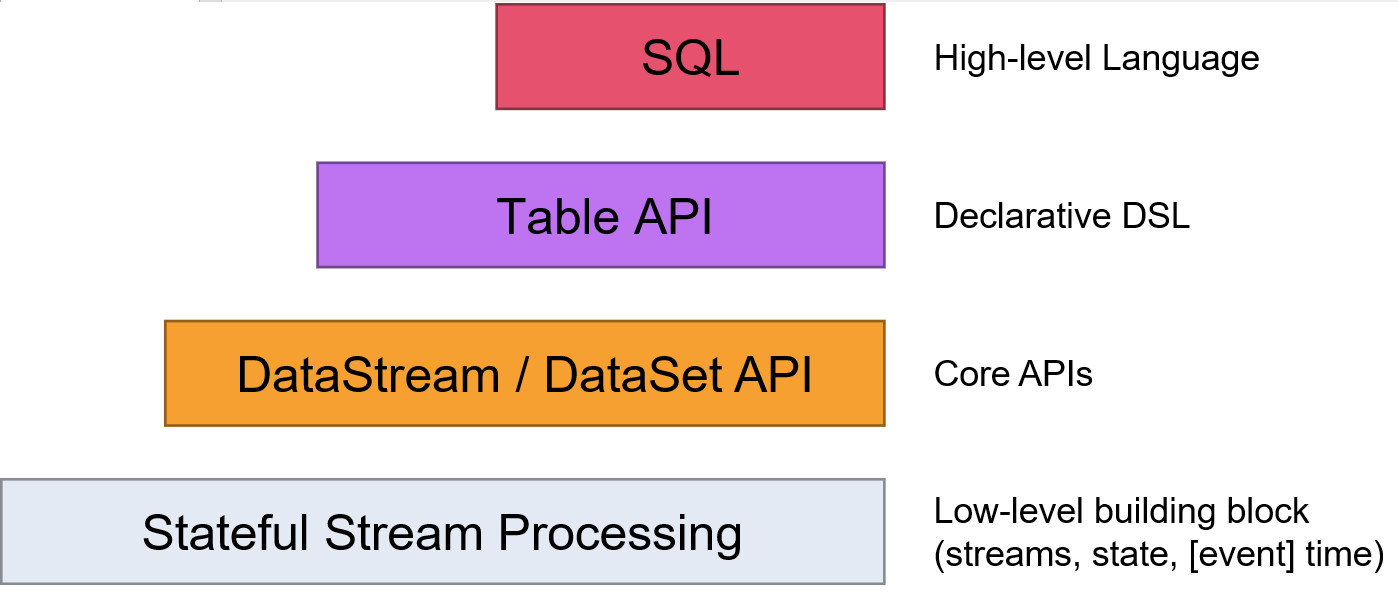
Flink 的核心 API 最初是针对特定的场景设计的，尽管 Table API / SQL 针对流处理和批处理已经实现了统一的 API，但当用户使用较底层的 API 时，仍然需要在批处理（DataSet API）和流处理（DataStream API）这两种不同的 API 之间进行选择。鉴于批处理是流处理的一种特例，将这两种 API 合并成统一的 API，有一些非常明显的好处，比如：

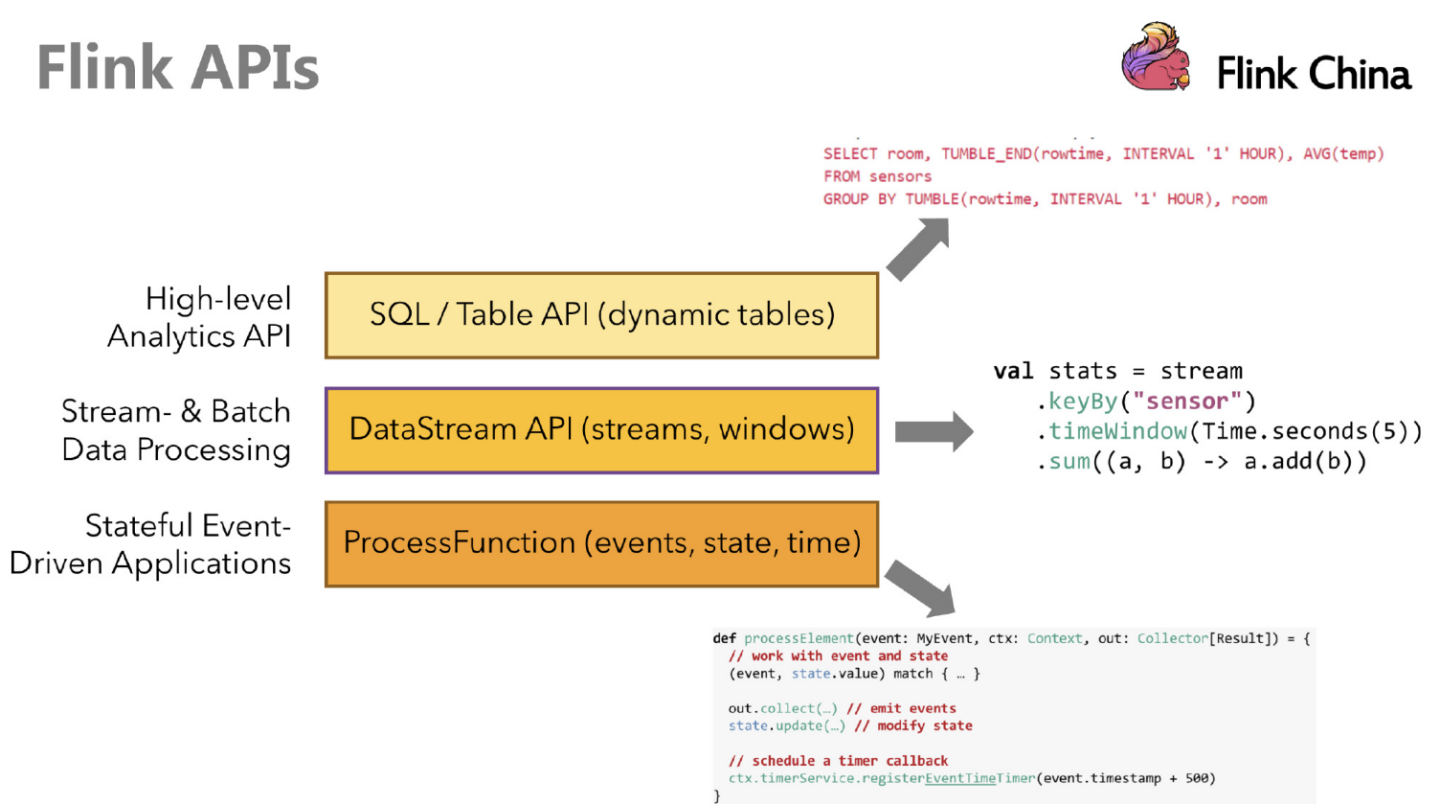
* 可复用性：作业可以在流和批这两种执行模式之间自由地切换，而无需重写任何代码。因此，用户可以复用同一个作业，来处理实时数据和历史数据。
* 维护简单：统一的 API 意味着流和批可以共用同一组 connector，维护同一套代码，并能够轻松地实现流批混合执行，例如 backfilling 之类的场景。

考虑到这些优点，社区已朝着流批统一的 DataStream API 迈出了第一步：支持高效的批处理（FLIP-134）。从长远来看，这意味着 DataSet API 将被弃用（FLIP-131），其功能将被包含在 DataStream API 和 Table API / SQL 中。

* API

Flink提供了多个层次的API供开发者使用，越往上抽象程度越高，使用起来越方便；越往下越底层，使用起来难度越大

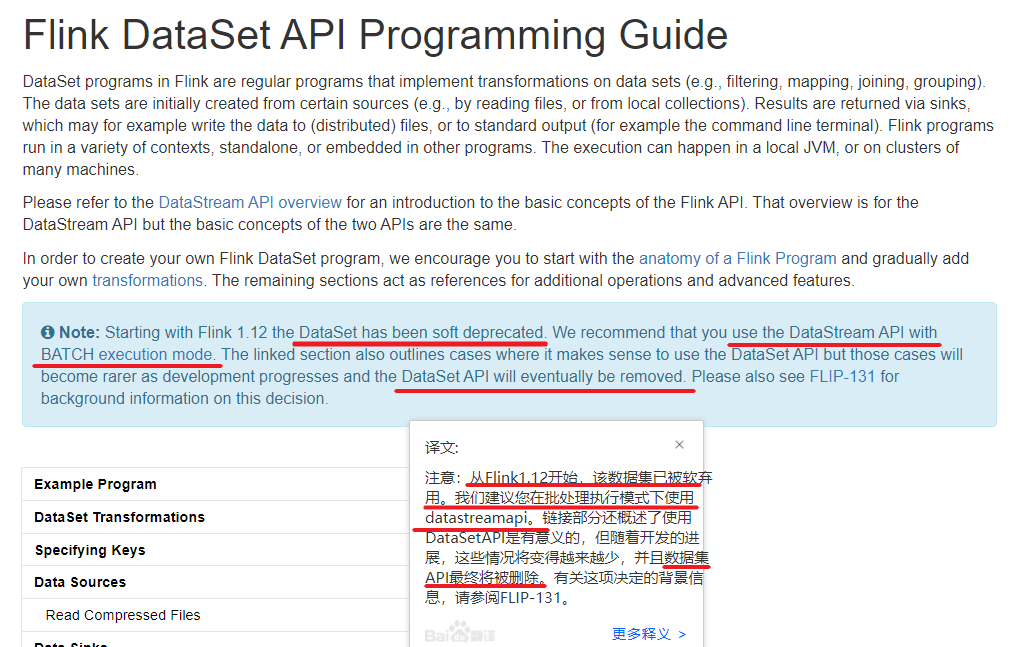




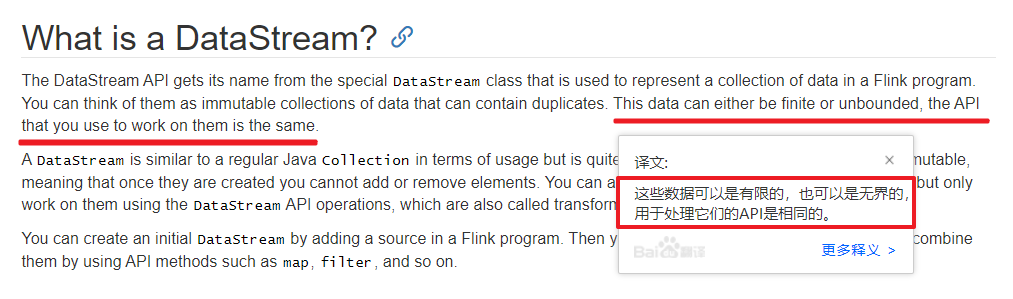
注意：在Flink1.12时支持流批一体，DataSetAPI已经不推荐使用了，所以课程中除了个别案例使用DataSet外，后续其他案例都会优先使用DataStream流式API，既支持无界数据处理/流处理，也支持有界数据处理/批处理！当然Table&SQL-API会单独学习

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/batch/>

<https://developer.aliyun.com/article/780123?spm=a2c6h.12873581.0.0.1e3e46ccbYFFrC>

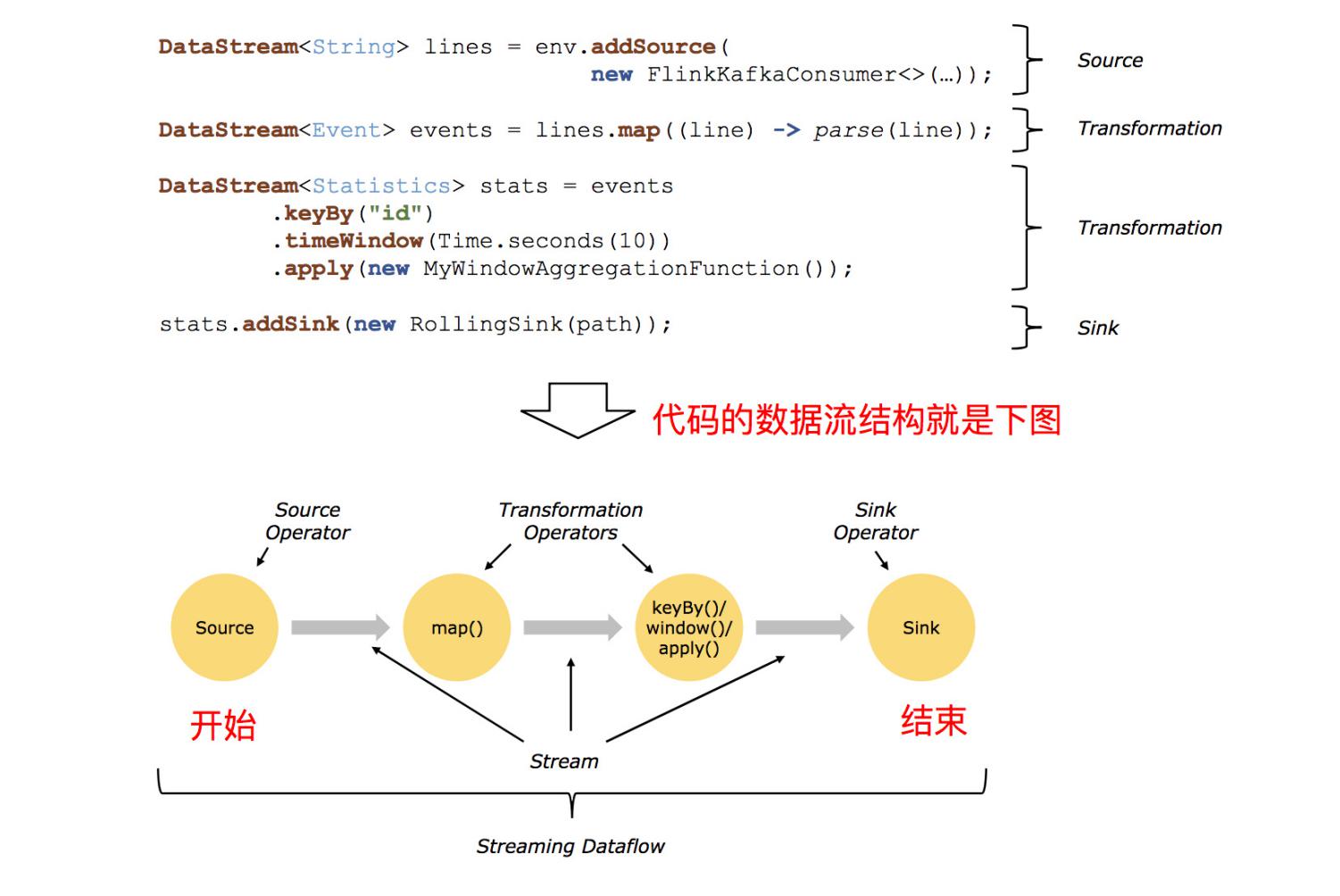
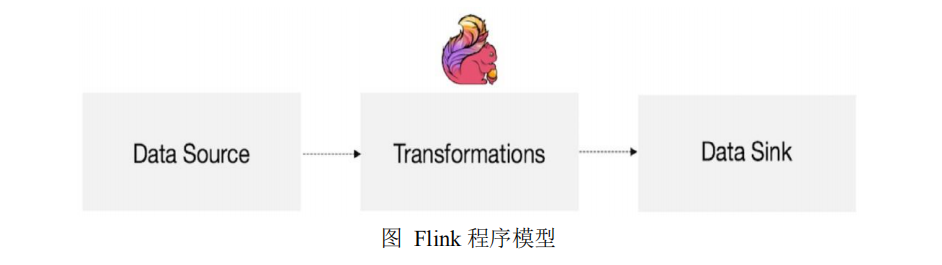


<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/datastream_api.html>

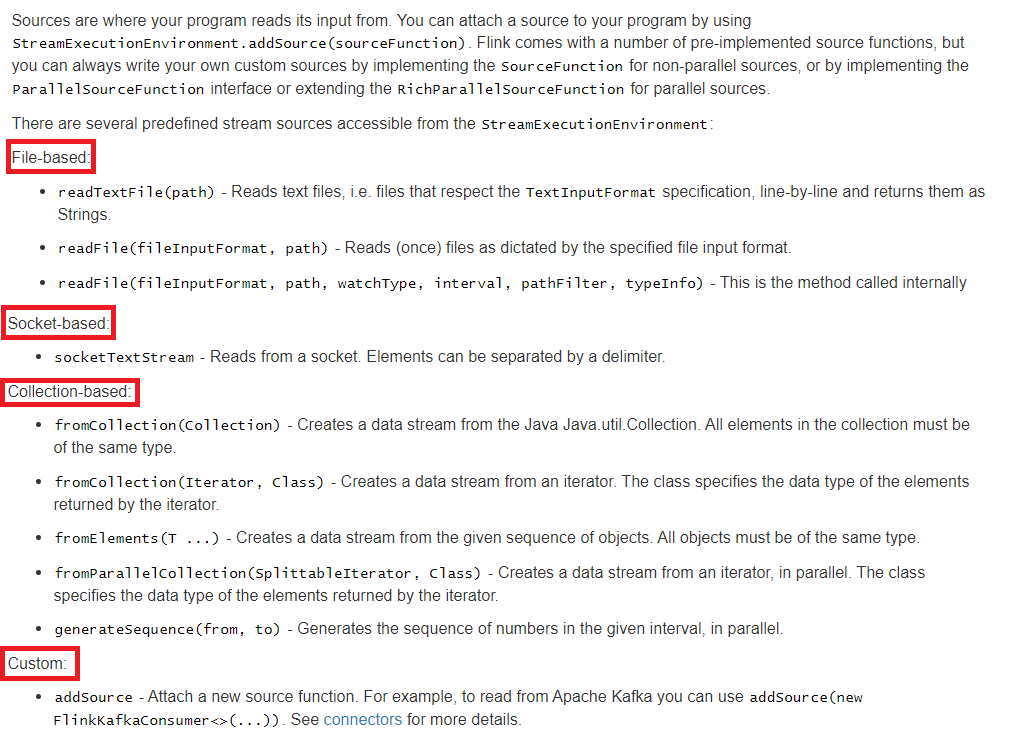


* 编程模型

Flink 应用程序结构主要包含三部分,Source/Transformation/Sink,如下图所示：



## Source



### 预定义Source

#### 基于集合的Source

* API

一般用于学习测试时编造数据时使用

1.env.fromElements(可变参数);

2.env.fromColletion(各种集合);

3.env.generateSequence(开始,结束);

4.env.fromSequence(开始,结束);

* 代码演示:

|  |
| --- |
| **package** cn.itcast.source;  **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  **import** java.util.Arrays;  */\*\*  \* Author itcast  \* Desc  \* 把本地的普通的Java集合/Scala集合变为分布式的Flink的DataStream集合!  \* 一般用于学习测试时编造数据时使用  \* 1.env.fromElements(可变参数);  \* 2.env.fromColletion(各种集合);  \* 3.env.generateSequence(开始,结束);  \* 4.env.fromSequence(开始,结束);  \*/* **public class** SourceDemo01 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***);  *//2.source  // \* 1.env.fromElements(可变参数);* DataStream<String> ds1 = env.fromElements(**"hadoop"**, **"spark"**, **"flink"**);  *// \* 2.env.fromColletion(各种集合);* DataStream<String> ds2 = env.fromCollection(Arrays.*asList*(**"hadoop"**, **"spark"**, **"flink"**));  *// \* 3.env.generateSequence(开始,结束);* DataStream<Long> ds3 = env.generateSequence(1, 10);  *//\* 4.env.fromSequence(开始,结束);* DataStream<Long> ds4 = env.fromSequence(1, 10);  *//3.Transformation  //4.sink* ds1.print();  ds2.print();  ds3.print();  ds4.print();  *//5.execute* env.execute();  } } |

#### 基于文件的Source

* API

一般用于学习测试

env.readTextFile(本地/HDFS文件/文件夹);//压缩文件也可以

* 代码演示:

|  |
| --- |
| **package** cn.itcast.source;  **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  */\*\*  \* Author itcast  \* Desc  \* 1.env.readTextFile(本地/HDFS文件/文件夹);//压缩文件也可以  \*/* **public class** SourceDemo02 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***);  *//2.source  // \* 1.env.readTextFile(本地文件/HDFS文件);//压缩文件也可以* DataStream<String> ds1 = env.readTextFile(**"data/input/words.txt"**);  DataStream<String> ds2 = env.readTextFile(**"data/input/dir"**);  DataStream<String> ds3 = env.readTextFile(**"hdfs://node1:8020//wordcount/input/words.txt"**);  DataStream<String> ds4 = env.readTextFile(**"data/input/wordcount.txt.gz"**);  *//3.Transformation  //4.sink* ds1.print();  ds2.print();  ds3.print();  ds4.print();  *//5.execute* env.execute();  } } |

#### 基于Socket的Source

一般用于学习测试

* 需求:

1.在node1上使用nc -lk 9999 向指定端口发送数据

nc是netcat的简称，原本是用来设置路由器,我们可以利用它向某个端口发送数据

如果没有该命令可以下安装

yum install -y nc

2.使用Flink编写流处理应用程序实时统计单词数量

* 代码实现:

|  |
| --- |
| **package** cn.itcast.source;  **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.api.common.functions.FlatMapFunction; **import** org.apache.flink.api.common.functions.MapFunction; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.KeyedStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.util.Collector;  */\*\*  \* Author itcast  \* Desc  \* SocketSource  \*/* **public class** SourceDemo03 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***);  *//2.source* DataStream<String> linesDS = env.socketTextStream(**"node1"**, 9999);   *//3.处理数据-transformation  //3.1每一行数据按照空格切分成一个个的单词组成一个集合* DataStream<String> wordsDS = linesDS.flatMap(**new** FlatMapFunction<String, String>() {  @Override  **public void** flatMap(String value, Collector<String> out) **throws** Exception {  *//value就是一行行的数据* String[] words = value.split(**" "**);  **for** (String word : words) {  out.collect(word);*//将切割处理的一个个的单词收集起来并返回* }  }  });  *//3.2对集合中的每个单词记为1* DataStream<Tuple2<String, Integer>> wordAndOnesDS = wordsDS.map(**new** MapFunction<String, Tuple2<String, Integer>>() {  @Override  **public** Tuple2<String, Integer> map(String value) **throws** Exception {  *//value就是进来一个个的单词* **return** Tuple2.*of*(value, 1);  }  });   *//3.3对数据按照单词(key)进行分组  //KeyedStream<Tuple2<String, Integer>, Tuple> groupedDS = wordAndOnesDS.keyBy(0);* KeyedStream<Tuple2<String, Integer>, String> groupedDS = wordAndOnesDS.keyBy(t -> t.**f0**);  *//3.4对各个组内的数据按照数量(value)进行聚合就是求sum* DataStream<Tuple2<String, Integer>> result = groupedDS.sum(1);   *//4.输出结果-sink* result.print();   *//5.触发执行-execute* env.execute();  } } |

### 自定义Source

#### 随机生成数据

* API

一般用于学习测试,模拟生成一些数据

Flink还提供了数据源接口,我们实现该接口就可以实现自定义数据源，不同的接口有不同的功能，分类如下：

SourceFunction:非并行数据源(并行度只能=1)

RichSourceFunction:多功能非并行数据源(并行度只能=1)

ParallelSourceFunction:并行数据源(并行度能够>=1)

RichParallelSourceFunction:多功能并行数据源(并行度能够>=1)--后续学习的Kafka数据源使用的就是该接口

* 需求

每隔1秒随机生成一条订单信息(订单ID、用户ID、订单金额、时间戳)

要求:

- 随机生成订单ID(UUID)

- 随机生成用户ID(0-2)

- 随机生成订单金额(0-100)

- 时间戳为当前系统时间

* 代码实现

|  |
| --- |
| **package** cn.itcast.source;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.source.RichParallelSourceFunction;  **import** java.util.Random; **import** java.util.UUID;  */\*\*  \* Author itcast  \* Desc  \*需求  \* 每隔1秒随机生成一条订单信息(订单ID、用户ID、订单金额、时间戳)  \* 要求:  \* - 随机生成订单ID(UUID)  \* - 随机生成用户ID(0-2)  \* - 随机生成订单金额(0-100)  \* - 时间戳为当前系统时间  \*  \* API  \* 一般用于学习测试,模拟生成一些数据  \* Flink还提供了数据源接口,我们实现该接口就可以实现自定义数据源，不同的接口有不同的功能，分类如下：  \* SourceFunction:非并行数据源(并行度只能=1)  \* RichSourceFunction:多功能非并行数据源(并行度只能=1)  \* ParallelSourceFunction:并行数据源(并行度能够>=1)  \* RichParallelSourceFunction:多功能并行数据源(并行度能够>=1)--后续学习的Kafka数据源使用的就是该接口  \*/* **public class** SourceDemo04\_Customer {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***);  *//2.Source* DataStream<Order> orderDS = env  .addSource(**new** MyOrderSource())  .setParallelism(2);   *//3.Transformation   //4.Sink* orderDS.print();  *//5.execute* env.execute();  }  @Data  @NoArgsConstructor  @AllArgsConstructor  **public static class** Order {  **private** String **id**;  **private** Integer **userId**;  **private** Integer **money**;  **private** Long **createTime**;  }  **public static class** MyOrderSource **extends** RichParallelSourceFunction<Order> {  **private** Boolean **flag** = **true**;  @Override  **public void** run(SourceContext<Order> ctx) **throws** Exception {  Random random = **new** Random();  **while** (**flag**){  Thread.*sleep*(1000);  String id = UUID.*randomUUID*().toString();  **int** userId = random.nextInt(3);  **int** money = random.nextInt(101);  **long** createTime = System.*currentTimeMillis*();  ctx.collect(**new** Order(id,userId,money,createTime));  }  }  *//取消任务/执行cancle命令的时候执行* @Override  **public void** cancel() {  **flag** = **false**;  }  } } |

#### MySQL

* 需求:

实际开发中,经常会实时接收一些数据,要和MySQL中存储的一些规则进行匹配,那么这时候就可以使用Flink自定义数据源从MySQL中读取数据

那么现在先完成一个简单的需求:

从MySQL中实时加载数据

要求MySQL中的数据有变化,也能被实时加载出来

* 准备数据

|  |
| --- |
| CREATE TABLE `t\_student` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` varchar(255) DEFAULT NULL,  `age` int(11) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=7 DEFAULT CHARSET=utf8;  INSERT INTO `t\_student` VALUES ('1', 'jack', '18');  INSERT INTO `t\_student` VALUES ('2', 'tom', '19');  INSERT INTO `t\_student` VALUES ('3', 'rose', '20');  INSERT INTO `t\_student` VALUES ('4', 'tom', '19');  INSERT INTO `t\_student` VALUES ('5', 'jack', '18');  INSERT INTO `t\_student` VALUES ('6', 'rose', '20'); |

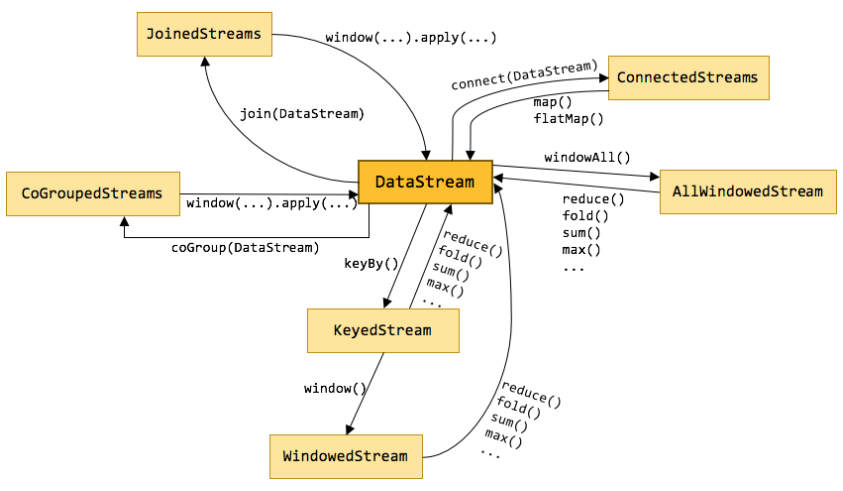
* 代码实现:

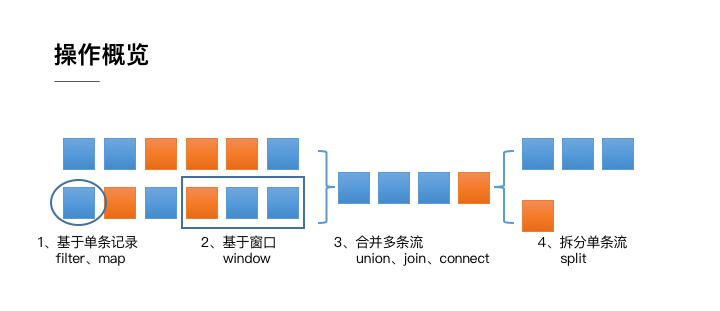
|  |
| --- |
| **package** cn.itcast.source;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.configuration.Configuration; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.source.RichParallelSourceFunction;  **import** java.sql.Connection; **import** java.sql.DriverManager; **import** java.sql.PreparedStatement; **import** java.sql.ResultSet; **import** java.util.concurrent.TimeUnit;  */\*\*  \* Author itcast  \* Desc  \* 需求:  \* 实际开发中,经常会实时接收一些数据,要和MySQL中存储的一些规则进行匹配,那么这时候就可以使用Flink自定义数据源从MySQL中读取数据  \* 那么现在先完成一个简单的需求:  \* 从MySQL中实时加载数据  \* 要求MySQL中的数据有变化,也能被实时加载出来  \*/* **public class** SourceDemo05\_Customer\_MySQL {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source* DataStream<Student> studentDS = env.addSource(**new** MySQLSource()).setParallelism(1);   *//3.Transformation  //4.Sink* studentDS.print();   *//5.execute* env.execute();  }   @Data  @NoArgsConstructor  @AllArgsConstructor  **public static class** Student {  **private** Integer **id**;  **private** String **name**;  **private** Integer **age**;  }   **public static class** MySQLSource **extends** RichParallelSourceFunction<Student> {  **private** Connection **conn** = **null**;  **private** PreparedStatement **ps** = **null**;   @Override  **public void** open(Configuration parameters) **throws** Exception {  *//加载驱动,开启连接  //Class.forName("com.mysql.jdbc.Driver");* **conn** = DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/bigdata"**, **"root"**, **"root"**);  String sql = **"select id,name,age from t\_student"**;  **ps** = **conn**.prepareStatement(sql);  }   **private boolean flag** = **true**;   @Override  **public void** run(SourceContext<Student> ctx) **throws** Exception {  **while** (**flag**) {  ResultSet rs = **ps**.executeQuery();  **while** (rs.next()) {  **int** id = rs.getInt(**"id"**);  String name = rs.getString(**"name"**);  **int** age = rs.getInt(**"age"**);  ctx.collect(**new** Student(id, name, age));  }  TimeUnit.***SECONDS***.sleep(5);  }  }  @Override  **public void** cancel() {  **flag** = **false**;  }  @Override  **public void** close() **throws** Exception {  **if** (**conn** != **null**) **conn**.close();  **if** (**ps** != **null**) **ps**.close();  }  } } |

## Transformation

### 官网API列表

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/stream/operators/>





整体来说，流式数据上的操作可以分为四类。

第一类是对于单条记录的操作，比如筛除掉不符合要求的记录（Filter 操作），或者将每条记录都做一个转换（Map 操作）

第二类是对多条记录的操作。比如说统计一个小时内的订单总成交量，就需要将一个小时内的所有订单记录的成交量加到一起。为了支持这种类型的操作，就得通过 Window 将需要的记录关联到一起进行处理

第三类是对多个流进行操作并转换为单个流。例如，多个流可以通过 Union、Join 或 Connect 等操作合到一起。这些操作合并的逻辑不同，但是它们最终都会产生了一个新的统一的流，从而可以进行一些跨流的操作。

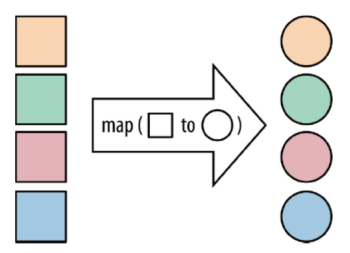
最后， DataStream 还支持与合并对称的拆分操作，即把一个流按一定规则拆分为多个流（Split 操作），每个流是之前流的一个子集，这样我们就可以对不同的流作不同的处理。

### 基本操作-略

#### map

* API

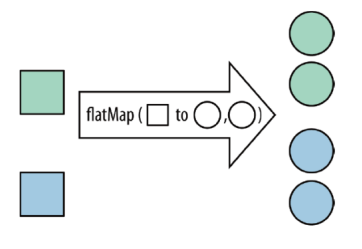
map:将函数作用在集合中的每一个元素上,并返回作用后的结果



#### flatMap

* API

flatMap:将集合中的每个元素变成一个或多个元素,并返回扁平化之后的结果

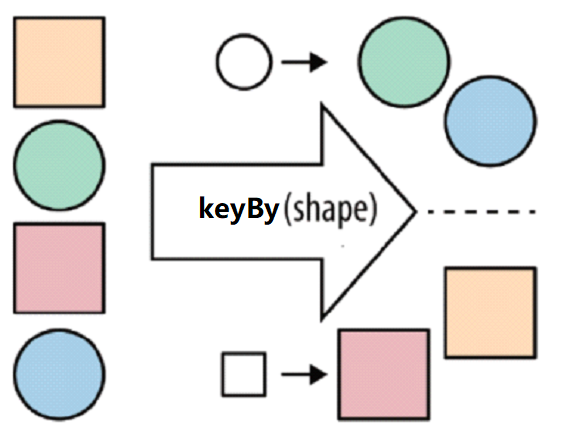


#### keyBy

按照指定的key来对流中的数据进行分组，前面入门案例中已经演示过

注意:

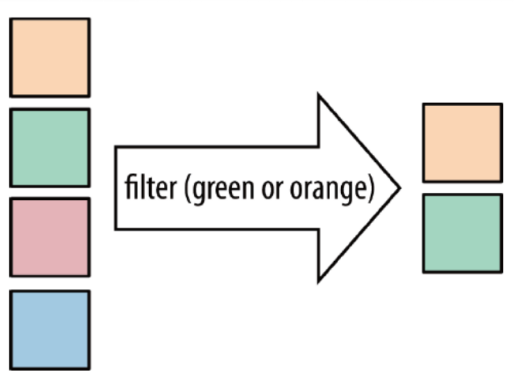
流处理中没有groupBy,而是keyBy



#### filter

* API

filter:按照指定的条件对集合中的元素进行过滤,过滤出返回true/符合条件的元素



#### sum

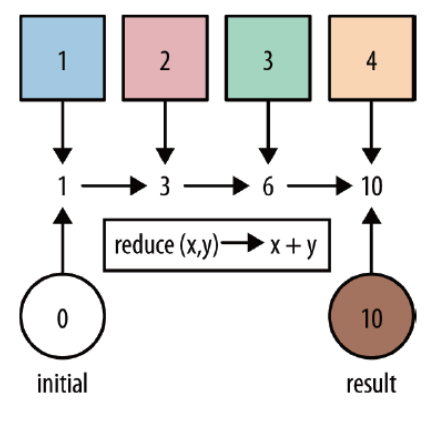
* API

sum:按照指定的字段对集合中的元素进行求和

#### reduce

* API

reduce:对集合中的元素进行聚合



#### 代码演示

* 需求:

对流数据中的单词进行统计，排除敏感词heihei

* 代码演示

|  |
| --- |
| **package** cn.itcast.transformation;  **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.api.common.functions.FilterFunction; **import** org.apache.flink.api.common.functions.FlatMapFunction; **import** org.apache.flink.api.common.functions.MapFunction; **import** org.apache.flink.api.common.functions.ReduceFunction; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.KeyedStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.util.Collector;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** TransformationDemo01 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***);  *//2.source* DataStream<String> linesDS = env.socketTextStream(**"node1"**, 9999);   *//3.处理数据-transformation* DataStream<String> wordsDS = linesDS.flatMap(**new** FlatMapFunction<String, String>() {  @Override  **public void** flatMap(String value, Collector<String> out) **throws** Exception {  *//value就是一行行的数据* String[] words = value.split(**" "**);  **for** (String word : words) {  out.collect(word);*//将切割处理的一个个的单词收集起来并返回* }  }  });  DataStream<String> filtedDS = wordsDS.filter(**new** FilterFunction<String>() {  @Override  **public boolean** filter(String value) **throws** Exception {  **return** !value.equals(**"heihei"**);  }  });  DataStream<Tuple2<String, Integer>> wordAndOnesDS = filtedDS.map(**new** MapFunction<String, Tuple2<String, Integer>>() {  @Override  **public** Tuple2<String, Integer> map(String value) **throws** Exception {  *//value就是进来一个个的单词* **return** Tuple2.*of*(value, 1);  }  });  *//KeyedStream<Tuple2<String, Integer>, Tuple> groupedDS = wordAndOnesDS.keyBy(0);* KeyedStream<Tuple2<String, Integer>, String> groupedDS = wordAndOnesDS.keyBy(t -> t.**f0**);   DataStream<Tuple2<String, Integer>> result1 = groupedDS.sum(1);  DataStream<Tuple2<String, Integer>> result2 = groupedDS.reduce(**new** ReduceFunction<Tuple2<String, Integer>>() {  @Override  **public** Tuple2<String, Integer> reduce(Tuple2<String, Integer> value1, Tuple2<String, Integer> value2) **throws** Exception {  **return** Tuple2.*of*(value1.**f0**, value1.**f1** + value1.**f1**);  }  });   *//4.输出结果-sink* result1.print(**"result1"**);  result2.print(**"result2"**);   *//5.触发执行-execute* env.execute();  } } |

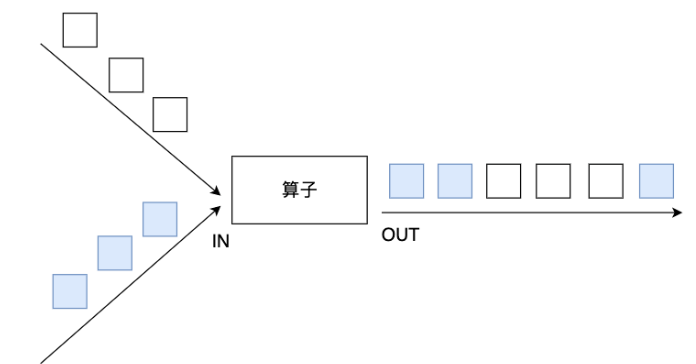
### 合并-拆分

#### union和connect

* API

union：

union算子可以合并多个同类型的数据流，并生成同类型的数据流，即可以将多个DataStream[T]合并为一个新的DataStream[T]。数据将按照先进先出（First In First Out）的模式合并，且不去重。



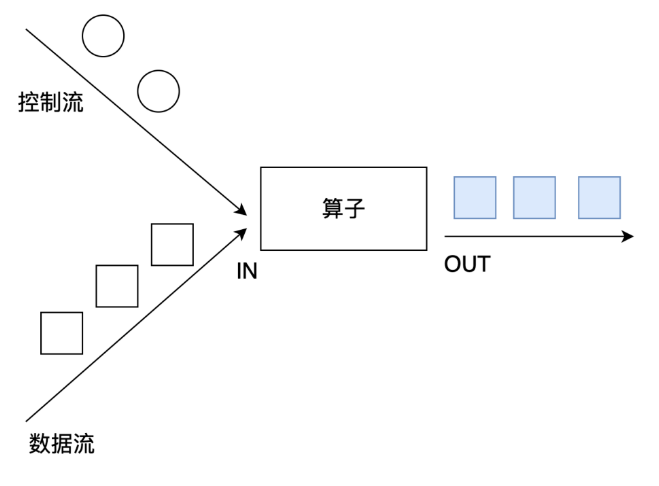
connect：

connect提供了和union类似的功能，用来连接两个数据流，它与union的区别在于：

connect只能连接两个数据流，union可以连接多个数据流。

connect所连接的两个数据流的数据类型可以不一致，union所连接的两个数据流的数据类型必须一致。

两个DataStream经过connect之后被转化为ConnectedStreams，ConnectedStreams会对两个流的数据应用不同的处理方法，且双流之间可以共享状态。



* 需求

将两个String类型的流进行union

将一个String类型和一个Long类型的流进行connect

* 代码实现

|  |
| --- |
| **package** cn.itcast.transformation;  **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.streaming.api.datastream.ConnectedStreams; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.co.CoMapFunction;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** TransformationDemo02 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***);   *//2.Source* DataStream<String> ds1 = env.fromElements(**"hadoop"**, **"spark"**, **"flink"**);  DataStream<String> ds2 = env.fromElements(**"hadoop"**, **"spark"**, **"flink"**);  DataStream<Long> ds3 = env.fromElements(1L, 2L, 3L);   *//3.Transformation* DataStream<String> result1 = ds1.union(ds2);*//合并但不去重 https://blog.csdn.net/valada/article/details/104367378* ConnectedStreams<String, Long> tempResult = ds1.connect(ds3);  *//interface CoMapFunction<IN1, IN2, OUT>* DataStream<String> result2 = tempResult.map(**new** CoMapFunction<String, Long, String>() {  @Override  **public** String map1(String value) **throws** Exception {  **return "String->String:"** + value;  }   @Override  **public** String map2(Long value) **throws** Exception {  **return "Long->String:"** + value.toString();  }  });   *//4.Sink* result1.print();  result2.print();   *//5.execute* env.execute();  } } |

#### split、select和Side Outputs

* API

Split就是将一个流分成多个流

Select就是获取分流后对应的数据

注意：split函数已过期并移除

Side Outputs：可以使用process方法对流中数据进行处理，并针对不同的处理结果将数据收集到不同的OutputTag中

* 需求:

对流中的数据按照奇数和偶数进行分流，并获取分流后的数据

* 代码实现:

|  |
| --- |
| **package** cn.itcast.transformation;  **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.api.common.typeinfo.TypeInformation; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.ProcessFunction; **import** org.apache.flink.util.Collector; **import** org.apache.flink.util.OutputTag;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** TransformationDemo03 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***);   *//2.Source* DataStreamSource<Integer> ds = env.fromElements(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);   *//3.Transformation  /\*SplitStream<Integer> splitResult = ds.split(new OutputSelector<Integer>() {  @Override  public Iterable<String> select(Integer value) {  //value是进来的数字  if (value % 2 == 0) {  //偶数  ArrayList<String> list = new ArrayList<>();  list.add("偶数");  return list;  } else {  //奇数  ArrayList<String> list = new ArrayList<>();  list.add("奇数");  return list;  }  }  });  DataStream<Integer> evenResult = splitResult.select("偶数");  DataStream<Integer> oddResult = splitResult.select("奇数");\*/   //定义两个输出标签* OutputTag<Integer> tag\_even = **new** OutputTag<Integer>(**"偶数"**, TypeInformation.*of*(Integer.**class**));  OutputTag<Integer> tag\_odd = **new** OutputTag<Integer>(**"奇数"**){};  *//对ds中的数据进行处理* SingleOutputStreamOperator<Integer> tagResult = ds.process(**new** ProcessFunction<Integer, Integer>() {  @Override  **public void** processElement(Integer value, Context ctx, Collector<Integer> out) **throws** Exception {  **if** (value % 2 == 0) {  *//偶数* ctx.output(tag\_even, value);  } **else** {  *//奇数* ctx.output(tag\_odd, value);  }  }  });   *//取出标记好的数据* DataStream<Integer> evenResult = tagResult.getSideOutput(tag\_even);  DataStream<Integer> oddResult = tagResult.getSideOutput(tag\_odd);   *//4.Sink* evenResult.print(**"偶数"**);  oddResult.print(**"奇数"**);   *//5.execute* env.execute();  } } |

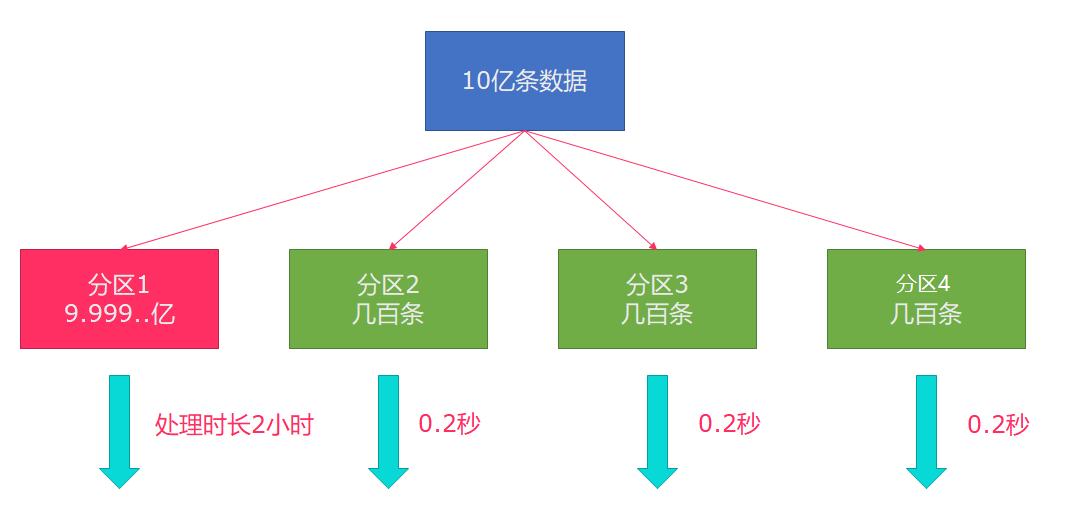
### 分区

#### rebalance重平衡分区

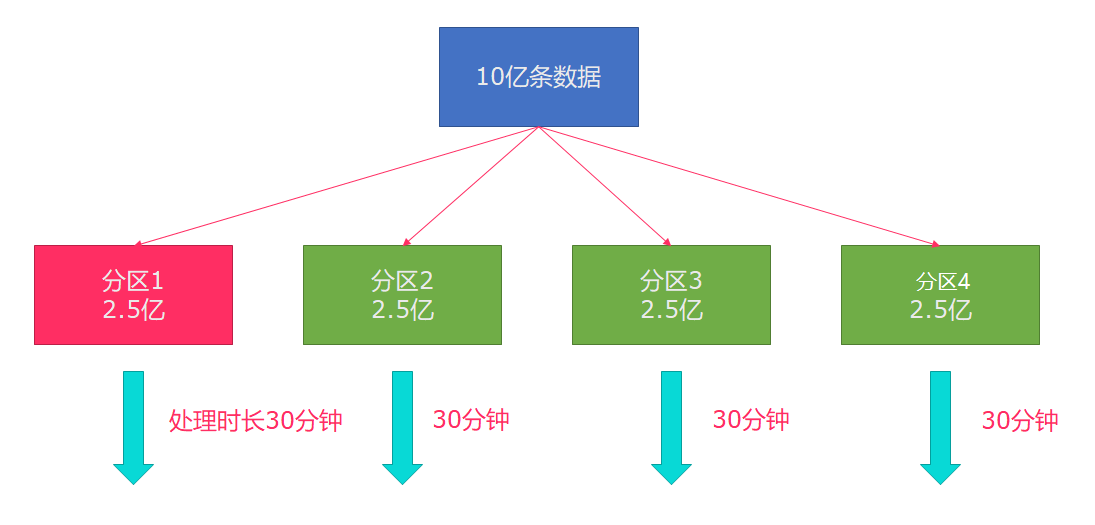
* API

类似于Spark中的repartition,但是功能更强大,可以直接解决数据倾斜

Flink也有数据倾斜的时候，比如当前有数据量大概10亿条数据需要处理，在处理过程中可能会发生如图所示的状况，出现了数据倾斜，其他3台机器执行完毕也要等待机器1执行完毕后才算整体将任务完成；



所以在实际的工作中，出现这种情况比较好的解决方案就是rebalance(内部使用round robin方法将数据均匀打散)



* 代码演示:

|  |
| --- |
| **package** cn.itcast.transformation;  **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.api.common.functions.FilterFunction; **import** org.apache.flink.api.common.functions.RichMapFunction; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** TransformationDemo04 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***).setParallelism(3);   *//2.source* DataStream<Long> longDS = env.fromSequence(0, 100);   *//3.Transformation  //下面的操作相当于将数据随机分配一下,有可能出现数据倾斜* DataStream<Long> filterDS = longDS.filter(**new** FilterFunction<Long>() {  @Override  **public boolean** filter(Long num) **throws** Exception {  **return** num > 10;  }  });   *//接下来使用map操作,将数据转为(分区编号/子任务编号, 数据)  //Rich表示多功能的,比MapFunction要多一些API可以供我们使用* DataStream<Tuple2<Integer, Integer>> result1 = filterDS  .map(**new** RichMapFunction<Long, Tuple2<Integer, Integer>>() {  @Override  **public** Tuple2<Integer, Integer> map(Long value) **throws** Exception {  *//获取分区编号/子任务编号* **int** id = getRuntimeContext().getIndexOfThisSubtask();  **return** Tuple2.*of*(id, 1);  }  }).keyBy(t -> t.**f0**).sum(1);   DataStream<Tuple2<Integer, Integer>> result2 = filterDS.rebalance()  .map(**new** RichMapFunction<Long, Tuple2<Integer, Integer>>() {  @Override  **public** Tuple2<Integer, Integer> map(Long value) **throws** Exception {  *//获取分区编号/子任务编号* **int** id = getRuntimeContext().getIndexOfThisSubtask();  **return** Tuple2.*of*(id, 1);  }  }).keyBy(t -> t.**f0**).sum(1);   *//4.sink  //result1.print();//有可能出现数据倾斜* result2.print();*//在输出前进行了rebalance重分区平衡,解决了数据倾斜   //5.execute* env.execute();  } } |

#### 其他分区

* API



说明:

recale分区。基于上下游Operator的并行度，将记录以循环的方式输出到下游Operator的每个实例。

举例:

上游并行度是2，下游是4，则上游一个并行度以循环的方式将记录输出到下游的两个并行度上;上游另一个并行度以循环的方式将记录输出到下游另两个并行度上。若上游并行度是4，下游并行度是2，则上游两个并行度将记录输出到下游一个并行度上；上游另两个并行度将记录输出到下游另一个并行度上。

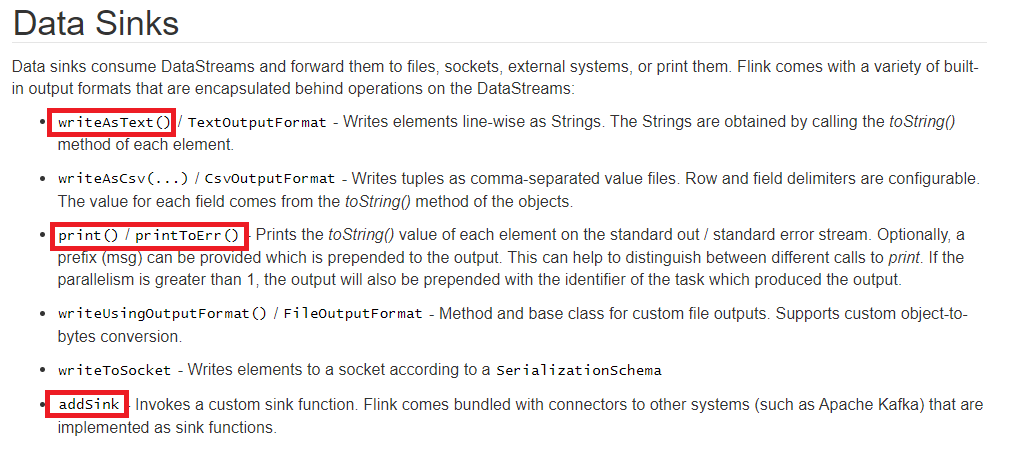
* 需求:

对流中的元素使用各种分区,并输出

* 代码实现

|  |
| --- |
| **package** cn.itcast.transformation;  **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.api.common.functions.FlatMapFunction; **import** org.apache.flink.api.common.functions.Partitioner; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.util.Collector;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** TransformationDemo05 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***);   *//2.Source* DataStream<String> linesDS = env.readTextFile(**"data/input/words.txt"**);  SingleOutputStreamOperator<Tuple2<String, Integer>> tupleDS = linesDS.flatMap(**new** FlatMapFunction<String, Tuple2<String, Integer>>() {  @Override  **public void** flatMap(String value, Collector<Tuple2<String, Integer>> out) **throws** Exception {  String[] words = value.split(**" "**);  **for** (String word : words) {  out.collect(Tuple2.*of*(word, 1));  }  }  });   *//3.Transformation* DataStream<Tuple2<String, Integer>> result1 = tupleDS.global();  DataStream<Tuple2<String, Integer>> result2 = tupleDS.broadcast();  DataStream<Tuple2<String, Integer>> result3 = tupleDS.forward();  DataStream<Tuple2<String, Integer>> result4 = tupleDS.shuffle();  DataStream<Tuple2<String, Integer>> result5 = tupleDS.rebalance();  DataStream<Tuple2<String, Integer>> result6 = tupleDS.rescale();  DataStream<Tuple2<String, Integer>> result7 = tupleDS.partitionCustom(**new** Partitioner<String>() {  @Override  **public int** partition(String key, **int** numPartitions) {  **return** key.equals(**"hello"**) ? 0 : 1;  }  }, t -> t.**f0**);   *//4.sink  //result1.print();  //result2.print();  //result3.print();  //result4.print();  //result5.print();  //result6.print();* result7.print();   *//5.execute* env.execute();  } } |

## Sink



### 预定义Sink

#### 基于控制台和文件的Sink

* API

1.ds.print 直接输出到控制台

2.ds.printToErr() 直接输出到控制台,用红色

3.ds.writeAsText("本地/HDFS的path",WriteMode.OVERWRITE).setParallelism(1)

* 注意:

在输出到path的时候,可以在前面设置并行度,如果

并行度>1,则path为目录

并行度=1,则path为文件名

* 代码演示:

|  |
| --- |
| **package** cn.itcast.sink;  **import** org.apache.flink.core.fs.FileSystem; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  */\*\*  \* Author itcast  \* Desc  \* 1.ds.print 直接输出到控制台  \* 2.ds.printToErr() 直接输出到控制台,用红色  \* 3.ds.collect 将分布式数据收集为本地集合  \* 4.ds.setParallelism(1).writeAsText("本地/HDFS的path",WriteMode.OVERWRITE)  \*/* **public class** SinkDemo01 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();   *//2.source  //DataStream<String> ds = env.fromElements("hadoop", "flink");* DataStream<String> ds = env.readTextFile(**"data/input/words.txt"**);   *//3.transformation  //4.sink* ds.print();  ds.printToErr();  ds.writeAsText(**"data/output/test"**, FileSystem.WriteMode.***OVERWRITE***).setParallelism(2);  *//注意:  //Parallelism=1为文件  //Parallelism>1为文件夹   //5.execute* env.execute();  } } |

### 自定义Sink

#### MySQL

* 需求:

将Flink集合中的数据通过自定义Sink保存到MySQL

* 代码实现:

|  |
| --- |
| **package** cn.itcast.sink;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.configuration.Configuration; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.sink.RichSinkFunction;  **import** java.sql.Connection; **import** java.sql.DriverManager; **import** java.sql.PreparedStatement;  */\*\*  \* Author itcast  \* Desc  \* 使用自定义sink将数据保存到MySQL  \*/* **public class** SinkDemo02\_MySQL {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source* DataStream<Student> studentDS = env.fromElements(**new** Student(**null**, **"tonyma"**, 18));  *//3.Transformation  //4.Sink* studentDS.addSink(**new** MySQLSink());   *//5.execute* env.execute();  }  @Data  @NoArgsConstructor  @AllArgsConstructor  **public static class** Student {  **private** Integer **id**;  **private** String **name**;  **private** Integer **age**;  }   **public static class** MySQLSink **extends** RichSinkFunction<Student> {  **private** Connection **conn** = **null**;  **private** PreparedStatement **ps** = **null**;   @Override  **public void** open(Configuration parameters) **throws** Exception {  *//加载驱动,开启连接  //Class.forName("com.mysql.jdbc.Driver");* **conn** = DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/bigdata"**, **"root"**, **"root"**);  String sql = **"INSERT INTO `t\_student` (`id`, `name`, `age`) VALUES (null, ?, ?)"**;  **ps** = **conn**.prepareStatement(sql);  }   @Override  **public void** invoke(Student value, Context context) **throws** Exception {  *//给ps中的?设置具体值* **ps**.setString(1,value.getName());  **ps**.setInt(2,value.getAge());  *//执行sql* **ps**.executeUpdate();  }   @Override  **public void** close() **throws** Exception {  **if** (**conn** != **null**) **conn**.close();  **if** (**ps** != **null**) **ps**.close();  }  } } |

## Connectors

### JDBC

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/connectors/jdbc.html>

|  |
| --- |
| **package** cn.itcast.connectors;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.connector.jdbc.JdbcConnectionOptions; **import** org.apache.flink.connector.jdbc.JdbcSink; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** ConnectorsDemo\_JDBC {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source* env.fromElements(**new** Student(**null**, **"tonyma"**, 18))  *//3.Transformation  //4.Sink* .addSink(JdbcSink.*sink*(  **"INSERT INTO `t\_student` (`id`, `name`, `age`) VALUES (null, ?, ?)"**,  (ps, s) -> {  ps.setString(1, s.getName());  ps.setInt(2, s.getAge());  },  **new** JdbcConnectionOptions.JdbcConnectionOptionsBuilder()  .withUrl(**"jdbc:mysql://localhost:3306/bigdata"**)  .withUsername(**"root"**)  .withPassword(**"root"**)  .withDriverName(**"com.mysql.jdbc.Driver"**)  .build()));  *//5.execute* env.execute();  }   @Data  @NoArgsConstructor  @AllArgsConstructor  **public static class** Student {  **private** Integer **id**;  **private** String **name**;  **private** Integer **age**;  } } |

### Kafka

#### pom依赖

Flink 里已经提供了一些绑定的 Connector，例如 kafka source 和 sink，Es sink 等。读写 kafka、es、rabbitMQ 时可以直接使用相应 connector 的 api 即可，虽然该部分是 Flink 项目源代码里的一部分，但是真正意义上不算作 Flink 引擎相关逻辑，并且该部分没有打包在二进制的发布包里面。所以在提交 Job 时候需要注意， job 代码 jar 包中一定要将相应的 connetor 相关类打包进去，否则在提交作业时就会失败，提示找不到相应的类，或初始化某些类异常。

<https://ci.apache.org/projects/flink/flink-docs-stable/dev/connectors/kafka.html>



#### 参数设置



以下参数都必须/建议设置上

1.订阅的主题

2.反序列化规则

3.消费者属性-集群地址

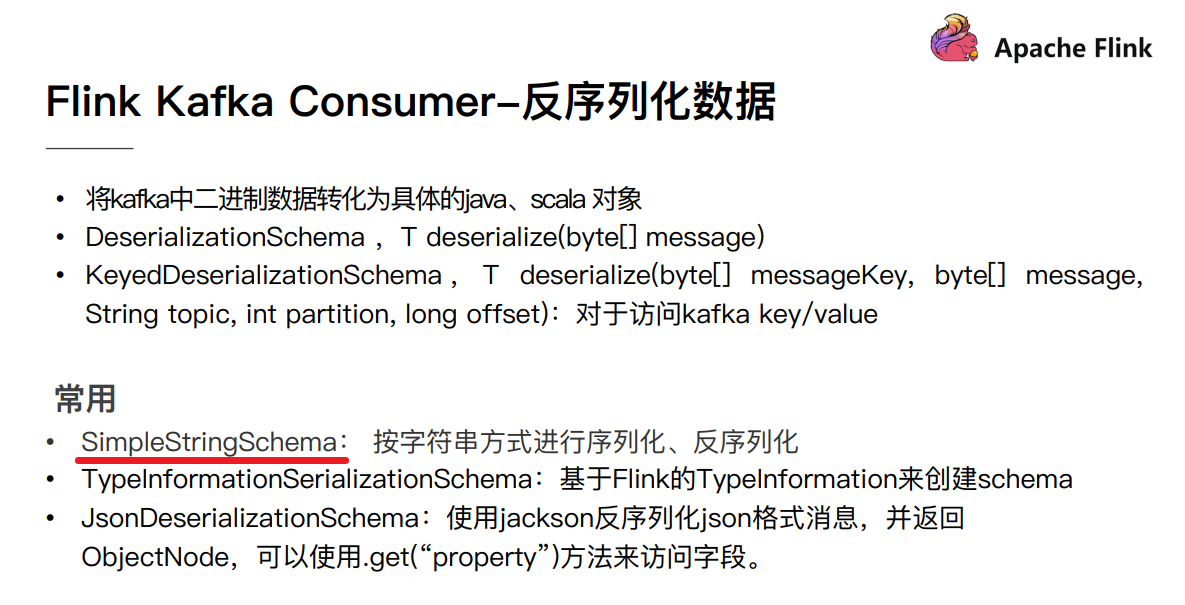
4.消费者属性-消费者组id(如果不设置,会有默认的,但是默认的不方便管理)

5.消费者属性-offset重置规则,如earliest/latest...

6.动态分区检测(当kafka的分区数变化/增加时,Flink能够检测到!)

7.如果没有设置Checkpoint,那么可以设置自动提交offset,后续学习了Checkpoint会把offset随着做Checkpoint的时候提交到Checkpoint和默认主题中

#### 参数说明









实际的生产环境中可能有这样一些需求，比如：

场景一：有一个 Flink 作业需要将五份数据聚合到一起，五份数据对应五个 kafka topic，随着业务增长，新增一类数据，同时新增了一个 kafka topic，如何在不重启作业的情况下作业自动感知新的 topic。

场景二：作业从一个固定的 kafka topic 读数据，开始该 topic 有 10 个 partition，但随着业务的增长数据量变大，需要对 kafka partition 个数进行扩容，由 10 个扩容到 20。该情况下如何在不重启作业情况下动态感知新扩容的 partition？

针对上面的两种场景，首先需要在构建 FlinkKafkaConsumer 时的 properties 中设置 flink.partition-discovery.interval-millis 参数为非负值，表示开启动态发现的开关，以及设置的时间间隔。此时 FlinkKafkaConsumer 内部会启动一个单独的线程定期去 kafka 获取最新的 meta 信息。

针对场景一，还需在构建 FlinkKafkaConsumer 时，topic 的描述可以传一个正则表达式描述的 pattern。每次获取最新 kafka meta 时获取正则匹配的最新 topic 列表。

针对场景二，设置前面的动态发现参数，在定期获取 kafka 最新 meta 信息时会匹配新的 partition。为了保证数据的正确性，新发现的 partition 从最早的位置开始读取。



注意:

开启 checkpoint 时 offset 是 Flink 通过状态 state 管理和恢复的，并不是从 kafka 的 offset 位置恢复。在 checkpoint 机制下，作业从最近一次checkpoint 恢复，本身是会回放部分历史数据，导致部分数据重复消费，Flink 引擎仅保证计算状态的精准一次，要想做到端到端精准一次需要依赖一些幂等的存储系统或者事务操作。

#### Kafka命令

● 查看当前服务器中的所有topic

/export/server/kafka/bin/kafka-topics.sh --list --zookeeper node1:2181

● 创建topic

/export/server/kafka/bin/kafka-topics.sh --create --zookeeper node1:2181 --replication-factor 2 --partitions 3 --topic flink\_kafka

● 查看某个Topic的详情

/export/server/kafka/bin/kafka-topics.sh --topic flink\_kafka --describe --zookeeper node1:2181

● 删除topic

/export/server/kafka/bin/kafka-topics.sh --delete --zookeeper node1:2181 --topic flink\_kafka

● 通过shell命令发送消息

/export/server/kafka/bin/kafka-console-producer.sh --broker-list node1:9092 --topic flink\_kafka

● 通过shell消费消息

/export/server/kafka/bin/kafka-console-consumer.sh --bootstrap-server node1:9092 --topic flink\_kafka --from-beginning

● 修改分区

/export/server/kafka/bin/kafka-topics.sh --alter --partitions 4 --topic flink\_kafka --zookeeper node1:2181

#### 代码实现-Kafka Consumer

|  |
| --- |
| **package** cn.itcast.connectors;  **import** org.apache.flink.api.common.functions.FlatMapFunction; **import** org.apache.flink.api.common.serialization.SimpleStringSchema; **import** org.apache.flink.api.java.tuple.Tuple; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.KeyedStream; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer; **import** org.apache.flink.util.Collector;  **import** java.util.Properties;  */\*\*  \* Author itcast  \* Desc  \* 需求:使用flink-connector-kafka\_2.12中的FlinkKafkaConsumer消费Kafka中的数据做WordCount  \* 需要设置如下参数:  \* 1.订阅的主题  \* 2.反序列化规则  \* 3.消费者属性-集群地址  \* 4.消费者属性-消费者组id(如果不设置,会有默认的,但是默认的不方便管理)  \* 5.消费者属性-offset重置规则,如earliest/latest...  \* 6.动态分区检测(当kafka的分区数变化/增加时,Flink能够检测到!)  \* 7.如果没有设置Checkpoint,那么可以设置自动提交offset,后续学习了Checkpoint会把offset随着做Checkpoint的时候提交到Checkpoint和默认主题中  \*/* **public class** ConnectorsDemo\_KafkaConsumer {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source* Properties props = **new** Properties();  props.setProperty(**"bootstrap.servers"**, **"node1:9092"**);  props.setProperty(**"group.id"**, **"flink"**);  props.setProperty(**"auto.offset.reset"**,**"latest"**);  props.setProperty(**"flink.partition-discovery.interval-millis"**,**"5000"**);*//会开启一个后台线程每隔5s检测一下Kafka的分区情况* props.setProperty(**"enable.auto.commit"**, **"true"**);  props.setProperty(**"auto.commit.interval.ms"**, **"2000"**);  *//kafkaSource就是KafkaConsumer* FlinkKafkaConsumer<String> kafkaSource = **new** FlinkKafkaConsumer<>(**"flink\_kafka"**, **new** SimpleStringSchema(), props);  kafkaSource.setStartFromGroupOffsets();*//设置从记录的offset开始消费,如果没有记录从auto.offset.reset配置开始消费  //kafkaSource.setStartFromEarliest();//设置直接从Earliest消费,和auto.offset.reset配置无关* DataStreamSource<String> kafkaDS = env.addSource(kafkaSource);   *//3.Transformation  //3.1切割并记为1* SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndOneDS = kafkaDS.flatMap(**new** FlatMapFunction<String, Tuple2<String, Integer>>() {  @Override  **public void** flatMap(String value, Collector<Tuple2<String, Integer>> out) **throws** Exception {  String[] words = value.split(**" "**);  **for** (String word : words) {  out.collect(Tuple2.*of*(word, 1));  }  }  });  *//3.2分组* KeyedStream<Tuple2<String, Integer>, Tuple> groupedDS = wordAndOneDS.keyBy(0);  *//3.3聚合* SingleOutputStreamOperator<Tuple2<String, Integer>> result = groupedDS.sum(1);   *//4.Sink* result.print();   *//5.execute* env.execute();  } } |

#### 代码实现-Kafka Producer

* 需求:

将Flink集合中的数据通过自定义Sink保存到Kafka

* 代码实现

|  |
| --- |
| **package** cn.itcast.connectors;  **import** com.alibaba.fastjson.JSON; **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.common.functions.MapFunction; **import** org.apache.flink.api.common.serialization.SimpleStringSchema; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;  **import** java.util.Properties;  */\*\*  \* Author itcast  \* Desc  \* 使用自定义sink-官方提供的flink-connector-kafka\_2.12-将数据保存到Kafka  \*/* **public class** ConnectorsDemo\_KafkaProducer {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source* DataStreamSource<Student> studentDS = env.fromElements(**new** Student(1, **"tonyma"**, 18));  *//3.Transformation  //注意:目前来说我们使用Kafka使用的序列化和反序列化都是直接使用最简单的字符串,所以先将Student转为字符串  //可以直接调用Student的toString,也可以转为JSON* SingleOutputStreamOperator<String> jsonDS = studentDS.map(**new** MapFunction<Student, String>() {  @Override  **public** String map(Student value) **throws** Exception {  *//String str = value.toString();* String jsonStr = JSON.*toJSONString*(value);  **return** jsonStr;  }  });   *//4.Sink* jsonDS.print();  *//根据参数创建KafkaProducer/KafkaSink* Properties props = **new** Properties();  props.setProperty(**"bootstrap.servers"**, **"node1:9092"**);  FlinkKafkaProducer<String> kafkaSink = **new** FlinkKafkaProducer<>(**"flink\_kafka"**, **new** SimpleStringSchema(), props);  jsonDS.addSink(kafkaSink);   *//5.execute* env.execute();   *// /export/server/kafka/bin/kafka-console-consumer.sh --bootstrap-server node1:9092 --topic flink\_kafka* }  @Data  @NoArgsConstructor  @AllArgsConstructor  **public static class** Student {  **private** Integer **id**;  **private** String **name**;  **private** Integer **age**;  } } |

### Redis

* API

通过flink 操作redis 其实我们可以通过传统的redis 连接池Jpoools 进行redis 的相关操作，但是flink 提供了专门操作redis 的RedisSink，使用起来更方便，而且不用我们考虑性能的问题，接下来将主要介绍RedisSink 如何使用。

<https://bahir.apache.org/docs/flink/current/flink-streaming-redis/>

RedisSink 核心类是RedisMapper 是一个接口，使用时我们要编写自己的redis 操作类实现这个接口中的三个方法，如下所示

1.getCommandDescription() ：

设置使用的redis 数据结构类型，和key 的名称，通过RedisCommand 设置数据结构类型

2.String getKeyFromData(T data)：

设置value 中的键值对key的值

3.String getValueFromData(T data);

设置value 中的键值对value的值

* 使用RedisCommand设置数据结构类型时和redis结构对应关系

|  |  |
| --- | --- |
| **Data Type** | **Redis Command [Sink]** |
| HASH | HSET |
| LIST | RPUSH, LPUSH |
| SET | SADD |
| PUBSUB | PUBLISH |
| STRING | SET |
| HYPER\_LOG\_LOG | PFADD |
| SORTED\_SET | ZADD |
| SORTED\_SET | ZREM |

* 需求

将Flink集合中的数据通过自定义Sink保存到Redis

* 代码实现

|  |
| --- |
| **package** cn.itcast.connectors;  **import** org.apache.flink.api.common.functions.FlatMapFunction; **import** org.apache.flink.api.java.tuple.Tuple; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.KeyedStream; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.connectors.redis.RedisSink; **import** org.apache.flink.streaming.connectors.redis.common.config.FlinkJedisPoolConfig; **import** org.apache.flink.streaming.connectors.redis.common.mapper.RedisCommand; **import** org.apache.flink.streaming.connectors.redis.common.mapper.RedisCommandDescription; **import** org.apache.flink.streaming.connectors.redis.common.mapper.RedisMapper; **import** org.apache.flink.util.Collector;  */\*\*  \* Author itcast  \* Desc  \* 需求:  \* 接收消息并做WordCount,  \* 最后将结果保存到Redis  \* 注意:存储到Redis的数据结构:使用hash也就是map  \* key value  \* WordCount (单词,数量)  \*/* **public class** ConnectorsDemo\_Redis {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source* DataStream<String> linesDS = env.socketTextStream(**"node1"**, 9999);   *//3.Transformation  //3.1切割并记为1* SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndOneDS = linesDS.flatMap(**new** FlatMapFunction<String, Tuple2<String, Integer>>() {  @Override  **public void** flatMap(String value, Collector<Tuple2<String, Integer>> out) **throws** Exception {  String[] words = value.split(**" "**);  **for** (String word : words) {  out.collect(Tuple2.*of*(word, 1));  }  }  });  *//3.2分组* KeyedStream<Tuple2<String, Integer>, Tuple> groupedDS = wordAndOneDS.keyBy(0);  *//3.3聚合* SingleOutputStreamOperator<Tuple2<String, Integer>> result = groupedDS.sum(1);   *//4.Sink* result.print();  *// \* 最后将结果保存到Redis  // \* 注意:存储到Redis的数据结构:使用hash也就是map  // \* key value  // \* WordCount (单词,数量)   //-1.创建RedisSink之前需要创建RedisConfig  //连接单机版Redis* FlinkJedisPoolConfig conf = **new** FlinkJedisPoolConfig.Builder().setHost(**"127.0.0.1"**).build();  *//连接集群版Redis  //HashSet<InetSocketAddress> nodes = new HashSet<>(Arrays.asList(new InetSocketAddress(InetAddress.getByName("node1"), 6379),new InetSocketAddress(InetAddress.getByName("node2"), 6379),new InetSocketAddress(InetAddress.getByName("node3"), 6379)));  //FlinkJedisClusterConfig conf2 = new FlinkJedisClusterConfig.Builder().setNodes(nodes).build();  //连接哨兵版Redis  //Set<String> sentinels = new HashSet<>(Arrays.asList("node1:26379", "node2:26379", "node3:26379"));  //FlinkJedisSentinelConfig conf3 = new FlinkJedisSentinelConfig.Builder().setMasterName("mymaster").setSentinels(sentinels).build();   //-3.创建并使用RedisSink* result.addSink(**new** RedisSink<Tuple2<String, Integer>>(conf, **new** RedisWordCountMapper()));   *//5.execute* env.execute();  }   */\*\*  \* -2.定义一个Mapper用来指定存储到Redis中的数据结构  \*/* **public static class** RedisWordCountMapper **implements** RedisMapper<Tuple2<String, Integer>> {  @Override  **public** RedisCommandDescription getCommandDescription() {  **return new** RedisCommandDescription(RedisCommand.***HSET***, **"WordCount"**);  }  @Override  **public** String getKeyFromData(Tuple2<String, Integer> data) {  **return** data.**f0**;  }  @Override  **public** String getValueFromData(Tuple2<String, Integer> data) {  **return** data.**f1**.toString();  }  } } |

## 扩展阅读：其他批处理API

### 累加器

* API

Flink累加器：

Flink中的累加器，与Mapreduce counter的应用场景类似，可以很好地观察task在运行期间的数据变化，如在Flink job任务中的算子函数中操作累加器，在任务执行结束之后才能获得累加器的最终结果。

Flink有以下内置累加器，每个累加器都实现了Accumulator接口。

* + IntCounter
  + LongCounter
  + DoubleCounter
* 编码步骤:

1.创建累加器

private IntCounter numLines = new IntCounter();

2.注册累加器

getRuntimeContext().addAccumulator("num-lines", this.numLines);

3.使用累加器

this.numLines.add(1);

4.获取累加器的结果

myJobExecutionResult.getAccumulatorResult("num-lines")

* 代码实现:

|  |
| --- |
| **package cn.itcast.batch**;  **import org.apache.flink.api.common.JobExecutionResult**; **import org.apache.flink.api.common.accumulators.IntCounter**; **import org.apache.flink.api.common.functions.RichMapFunction**; **import org.apache.flink.api.java.ExecutionEnvironment**; **import org.apache.flink.api.java.operators.DataSource**; **import org.apache.flink.api.java.operators.MapOperator**; **import org.apache.flink.configuration.Configuration**; **import org.apache.flink.core.fs.FileSystem**;  */\*\*  \* Author itcast  \* Desc 演示Flink累加器,统计处理的数据条数  \*/* **public class OtherAPI\_Accumulator** {  **public static void** main(**String**[] args) **throws Exception** {  *//1.env* **ExecutionEnvironment** env = **ExecutionEnvironment**.*getExecutionEnvironment*();   *//2.Source* **DataSource**<**String**> dataDS = env.fromElements("aaa", "bbb", "ccc", "ddd");   *//3.Transformation* **MapOperator**<**String**, **String**> result = dataDS.map(**new RichMapFunction**<**String**, **String**>() {  *//-1.创建累加器* **private IntCounter** elementCounter = **new** IntCounter();  **Integer** count = 0;   *@Override* **public void** open(**Configuration** parameters) **throws Exception** {  **super**.open(parameters);  *//-2注册累加器* getRuntimeContext().addAccumulator("elementCounter", elementCounter);  }   *@Override* **public String** map(**String** value) **throws Exception** {  *//-3.使用累加器* **this**.elementCounter.add(1);  count+=1;  **System**.***out***.println("不使用累加器统计的结果:"+count);  **return** value;  }  }).setParallelism(2);   *//4.Sink* result.writeAsText("data/output/test", **FileSystem**.WriteMode.***OVERWRITE***);   *//5.execute  //-4.获取加强结果* **JobExecutionResult** jobResult = env.execute();  **int** nums = jobResult.getAccumulatorResult("elementCounter");  **System**.***out***.println("使用累加器统计的结果:"+nums);  } } |

### 广播变量

* API

Flink支持广播。可以将数据广播到TaskManager上就可以供TaskManager中的SubTask/task去使用，数据存储到内存中。这样可以减少大量的shuffle操作，而不需要多次传递给集群节点；

比如在数据join阶段，不可避免的就是大量的shuffle操作，我们可以把其中一个dataSet广播出去，一直加载到taskManager的内存中，可以直接在内存中拿数据，避免了大量的shuffle，导致集群性能下降；

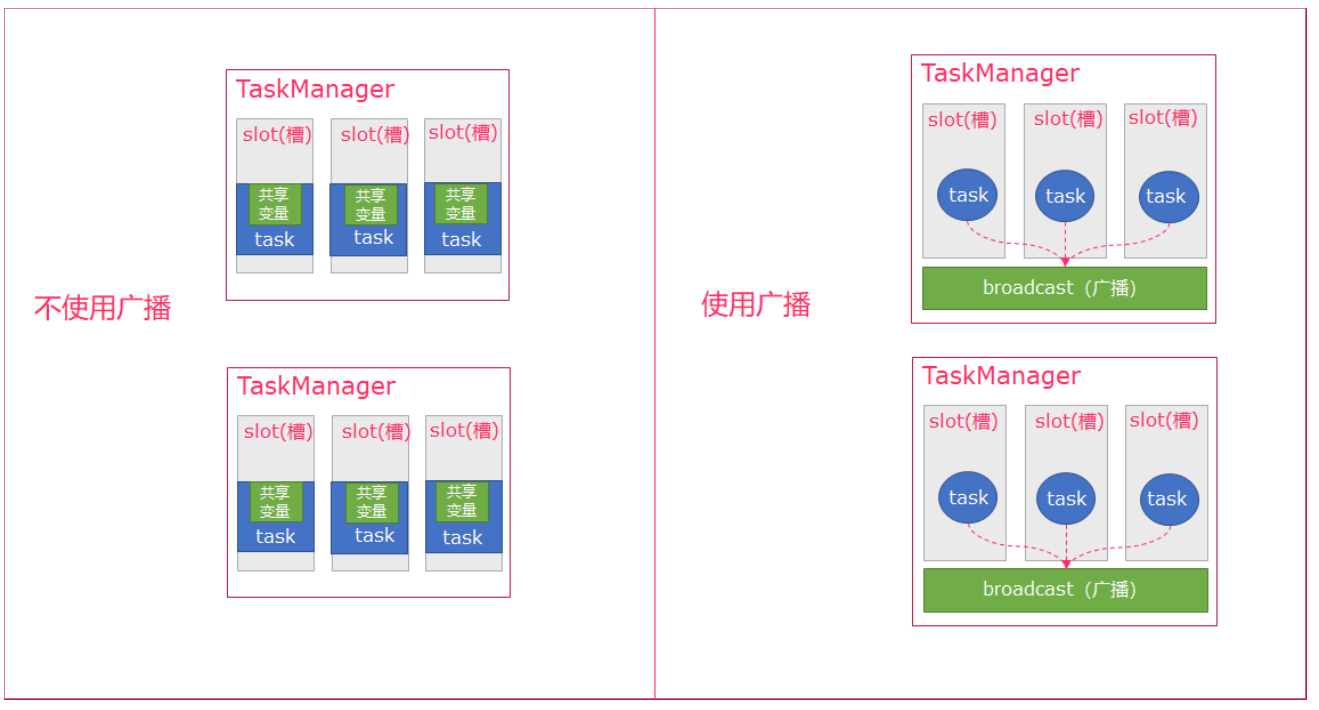
* 图解：

- 可以理解广播就是一个公共的共享变量

- 将一个数据集广播后，不同的Task都可以在节点上获取到

- 每个节点只存一份

- 如果不使用广播，每一个Task都会拷贝一份数据集，造成内存资源浪费



* 注意：

广播变量是要把dataset广播到内存中，所以广播的数据量不能太大，否则会出现OOM

广播变量的值不可修改，这样才能确保每个节点获取到的值都是一致的

* 编码步骤:

1：广播数据

.withBroadcastSet(DataSet, "name");

2：获取广播的数据

Collection<> broadcastSet = getRuntimeContext().getBroadcastVariable("name");

3:使用广播数据

* 需求:

将studentDS(学号,姓名)集合广播出去(广播到各个TaskManager内存中)

然后使用scoreDS(学号,学科,成绩)和广播数据(学号,姓名)进行关联,得到这样格式的数据:(姓名,学科,成绩)

* 代码实现:

|  |
| --- |
| **package cn.itcast.batch**;  **import org.apache.flink.api.common.functions.RichMapFunction**; **import org.apache.flink.api.java.ExecutionEnvironment**; **import org.apache.flink.api.java.operators.DataSource**; **import org.apache.flink.api.java.operators.MapOperator**; **import org.apache.flink.api.java.tuple.Tuple2**; **import org.apache.flink.api.java.tuple.Tuple3**; **import org.apache.flink.configuration.Configuration**;  **import java.util.Arrays**; **import java.util.HashMap**; **import java.util.**List; **import java.util.**Map;  */\*\*  \* Author itcast  \* Desc 演示Flink广播变量  \* 编程步骤：  \* 1：广播数据  \* .withBroadcastSet(DataSet, "name");  \* 2：获取广播的数据  \* Collection<> broadcastSet = getRuntimeContext().getBroadcastVariable("name");  \* 3:使用广播数据  \* <p>  \* 需求:  \* 将studentDS(学号,姓名)集合广播出去(广播到各个TaskManager内存中)  \* 然后使用scoreDS(学号,学科,成绩)和广播数据(学号,姓名)进行关联,得到这样格式的数据:(姓名,学科,成绩)  \*/* **public class OtherAPI\_Broadcast** {  **public static void** main(**String**[] args) **throws Exception** {  *//1.env* **ExecutionEnvironment** env = **ExecutionEnvironment**.*getExecutionEnvironment*();   *//2.Source  //学生数据集(学号,姓名)* **DataSource**<**Tuple2**<**Integer**, **String**>> studentDS = env.fromCollection(  **Arrays**.*asList*(**Tuple2**.*of*(1, "张三"), **Tuple2**.*of*(2, "李四"), **Tuple2**.*of*(3, "王五"))  );   *//成绩数据集(学号,学科,成绩)* **DataSource**<**Tuple3**<**Integer**, **String**, **Integer**>> scoreDS = env.fromCollection(  **Arrays**.*asList*(**Tuple3**.*of*(1, "语文", 50), **Tuple3**.*of*(2, "数学", 70), **Tuple3**.*of*(3, "英文", 86))  );   *//3.Transformation  //将studentDS(学号,姓名)集合广播出去(广播到各个TaskManager内存中)  //然后使用scoreDS(学号,学科,成绩)和广播数据(学号,姓名)进行关联,得到这样格式的数据:(姓名,学科,成绩)* **MapOperator**<**Tuple3**<**Integer**, **String**, **Integer**>, **Tuple3**<**String**, **String**, **Integer**>> result = scoreDS.map(  **new RichMapFunction**<**Tuple3**<**Integer**, **String**, **Integer**>, **Tuple3**<**String**, **String**, **Integer**>>() {  *//定义一集合用来存储(学号,姓名)* Map<**Integer**, **String**> studentMap = **new** HashMap<>();   *//open方法一般用来初始化资源，每个subtask任务只被调用一次  @Override* **public void** open(**Configuration** parameters) **throws Exception** {  *//-2.获取广播数据* List<**Tuple2**<**Integer**, **String**>> studentList = getRuntimeContext().getBroadcastVariable("studentInfo");  **for** (**Tuple2**<**Integer**, **String**> tuple : studentList) {  studentMap.put(tuple.f0, tuple.f1);  }  *//studentMap = studentList.stream().collect(Collectors.toMap(t -> t.f0, t -> t.f1));* }   *@Override* **public Tuple3**<**String**, **String**, **Integer**> map(**Tuple3**<**Integer**, **String**, **Integer**> value) **throws Exception** {  *//-3.使用广播数据* **Integer** stuID = value.f0;  **String** stuName = studentMap.getOrDefault(stuID, "");  *//返回(姓名,学科,成绩)* **return Tuple3**.*of*(stuName, value.f1, value.f2);  }  *//-1.广播数据到各个TaskManager* }).withBroadcastSet(studentDS, "studentInfo");   *//4.Sink* result.print();  } } |

### 分布式缓存

* API解释

Flink提供了一个类似于Hadoop的分布式缓存，让并行运行实例的函数可以在本地访问。

这个功能可以被使用来分享外部静态的数据，例如：机器学习的逻辑回归模型等

* 注意

广播变量是将变量分发到各个TaskManager节点的内存上，分布式缓存是将文件缓存到各个TaskManager节点上；

* 编码步骤:

1：注册一个分布式缓存文件

env.registerCachedFile("hdfs:///path/file", "cachefilename")

2：访问分布式缓存文件中的数据

File myFile = getRuntimeContext().getDistributedCache().getFile("cachefilename");

3：使用

* 需求

将scoreDS(学号, 学科, 成绩)中的数据和分布式缓存中的数据(学号,姓名)关联,得到这样格式的数据: (学生姓名,学科,成绩)

* 代码实现:

|  |
| --- |
| **package cn.itcast.batch**;  **import org.apache.commons.io.FileUtils**; **import org.apache.flink.api.common.functions.RichMapFunction**; **import org.apache.flink.api.java.ExecutionEnvironment**; **import org.apache.flink.api.java.operators.DataSource**; **import org.apache.flink.api.java.operators.MapOperator**; **import org.apache.flink.api.java.tuple.Tuple3**; **import org.apache.flink.configuration.Configuration**;  **import java.io.File**; **import java.util.Arrays**; **import java.util.HashMap**; **import java.util.**List; **import java.util.**Map;  */\*\*  \* Author itcast  \* Desc 演示Flink分布式缓存  \* 编码步骤:  \* 1：注册一个分布式缓存文件  \* env.registerCachedFile("hdfs:///path/file", "cachefilename")  \* 2：访问分布式缓存文件中的数据  \* File myFile = getRuntimeContext().getDistributedCache().getFile("cachefilename");  \* 3：使用  \*  \* 需求:  \* 将scoreDS(学号, 学科, 成绩)中的数据和分布式缓存中的数据(学号,姓名)关联,得到这样格式的数据: (学生姓名,学科,成绩)  \*/* **public class OtherAPI\_DistributedCache** {  **public static void** main(**String**[] args) **throws Exception** {  *//1.env* **ExecutionEnvironment** env = **ExecutionEnvironment**.*getExecutionEnvironment*();   *//2.Source  //注意:先将本地资料中的distribute\_cache\_student文件上传到HDFS  //-1.注册分布式缓存文件  //env.registerCachedFile("hdfs://node01:8020/distribute\_cache\_student", "studentFile");* env.registerCachedFile("data/input/distribute\_cache\_student", "studentFile");   *//成绩数据集(学号,学科,成绩)* **DataSource**<**Tuple3**<**Integer**, **String**, **Integer**>> scoreDS = env.fromCollection(  **Arrays**.*asList*(**Tuple3**.*of*(1, "语文", 50), **Tuple3**.*of*(2, "数学", 70), **Tuple3**.*of*(3, "英文", 86))  );   *//3.Transformation  //将scoreDS(学号, 学科, 成绩)中的数据和分布式缓存中的数据(学号,姓名)关联,得到这样格式的数据: (学生姓名,学科,成绩)* **MapOperator**<**Tuple3**<**Integer**, **String**, **Integer**>, **Tuple3**<**String**, **String**, **Integer**>> result = scoreDS.map(  **new RichMapFunction**<**Tuple3**<**Integer**, **String**, **Integer**>, **Tuple3**<**String**, **String**, **Integer**>>() {  *//定义一集合用来存储(学号,姓名)* Map<**Integer**, **String**> studentMap = **new** HashMap<>();   *@Override* **public void** open(**Configuration** parameters) **throws Exception** {  *//-2.加载分布式缓存文件* **File** file = getRuntimeContext().getDistributedCache().getFile("studentFile");  List<**String**> studentList = **FileUtils**.*readLines*(file);  **for** (**String** str : studentList) {  **String**[] arr = str.split(",");  studentMap.put(**Integer**.*parseInt*(arr[0]), arr[1]);  }  }   *@Override* **public Tuple3**<**String**, **String**, **Integer**> map(**Tuple3**<**Integer**, **String**, **Integer**> value) **throws Exception** {  *//-3.使用分布式缓存文件中的数据* **Integer** stuID = value.f0;  **String** stuName = studentMap.getOrDefault(stuID, "");  *//返回(姓名,学科,成绩)* **return Tuple3**.*of*(stuName, value.f1, value.f2);  }  });   *//4.Sink* result.print();  } } |