# 第二章 Flink 批流一体API开发

## 课程目标

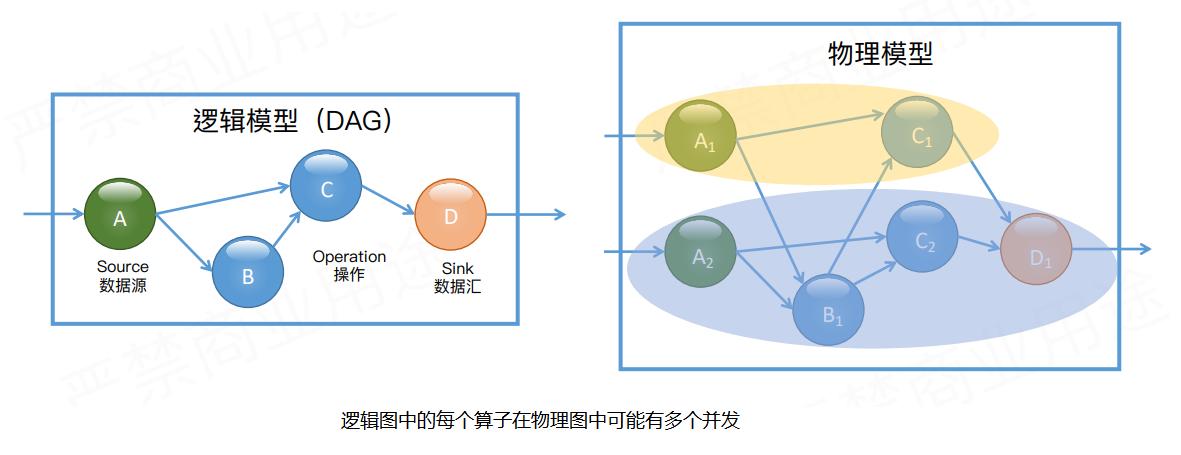
* 了解流处理的相关概念
* 掌握FlinkDataStream-SourceOperator
* 掌握FlinkDataStream-TransformationOperator
* 掌握FlinkDataStream-SinkOperator
* 了解Flink的累加器
* 了解Flink的广播变量
* 了解Flink的分布式缓存

## 流处理相关概念

### 流处理的基本概念

一般来说，由于需要支持**无限数据集**的处理，流处理系统一般采用一种**数据驱动**的处理方式。它会提前设置一些**算子**，然后等到**数据到达**后对数据进行处理。

为了表达复杂的计算逻辑，包括 Flink 在内的分布式流处理引擎一般采用 **DAG 图**来表示整个计算逻辑，其中 DAG 图中的每一个点就代表一个基本的逻辑单元，也就是算子。由于计算逻辑被组织成**有向图**，数据会按照边的方向，从一些特殊的 Source 节点流入系统，然后通过**网络传输**、**本地传输**等不同的数据传输方式在算子之间进行发送和处理，最后会通过 Sink 节点将计算结果发送到某个外部系统或数据库中。



对于分布式流处理引擎，实际运行时物理模型可能比较复杂，由于每个算子都可能有**多个实例**。如图所示，作为 Source 的 **A 算子有两个实例**，中间算子 **C 也有两个实例**。在逻辑模型中，A 和 B 是 C 的**上游节点**，而在对应的物理逻辑中，C 的所有实例和 A、B 的所有实例之间可能都存在**数据交换**。在物理模型中，会根据计算逻辑，采用系统**自动优化**或**人为指定**的方式将计算工作分布到不同的实例中。**只有当算子实例分布到不同进程上时，才需要通过网络进行数据传输，而同一进程中的多个实例之间的数据传输通常是不需要通过网络的**。

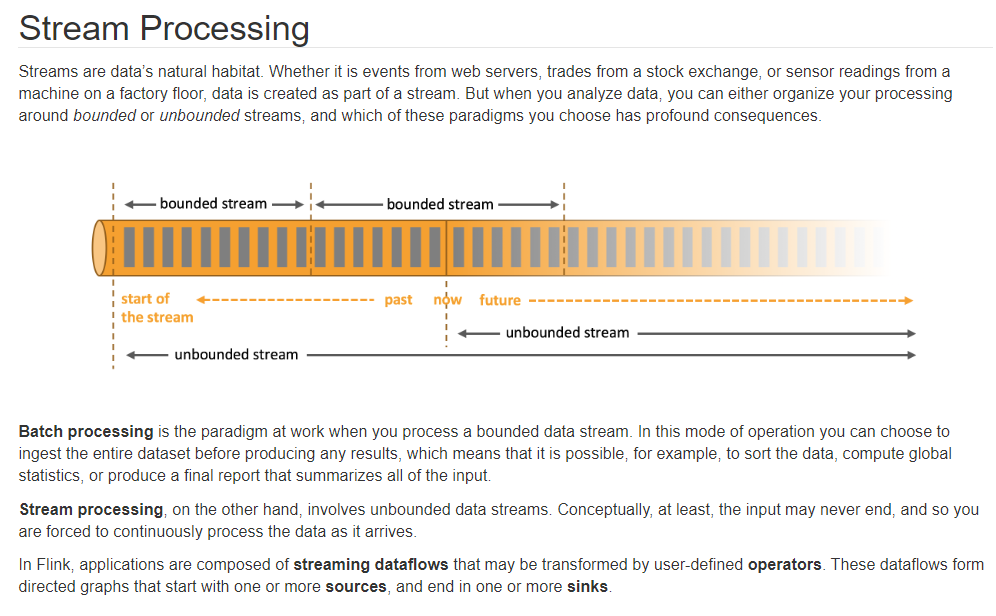


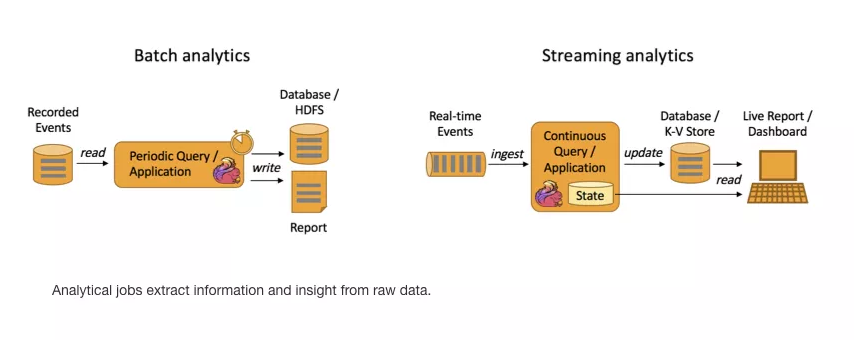
由于流处理的计算逻辑是通过 **DAG 图**来表示的，因此它们的大部分 API 都是围绕构建这种计算**逻辑图**来设计的。例如，对于几年前非常流行的 Apache Storm，它的 Word Count 的示例如表 1 所示。基于 Apache Storm 用户需要在图中添加 Spout 或 Bolt 这种算子，并指定算子之前的连接方式。这样，在完成整个图的构建之后，就可以将图提交到远程或本地集群运行。

与之对比，Apache Flink 的接口虽然也是在构建计算逻辑图，但是 Flink 的 API 定义更加**面向数据本身的处理逻辑**，它把数据流抽象成为一个**无限集**，然后定义了一组集合上的操作，然后在**底层自动构建**相应的 **DAG 图**。可以看出，Flink 的 API 要更“上层”一些。

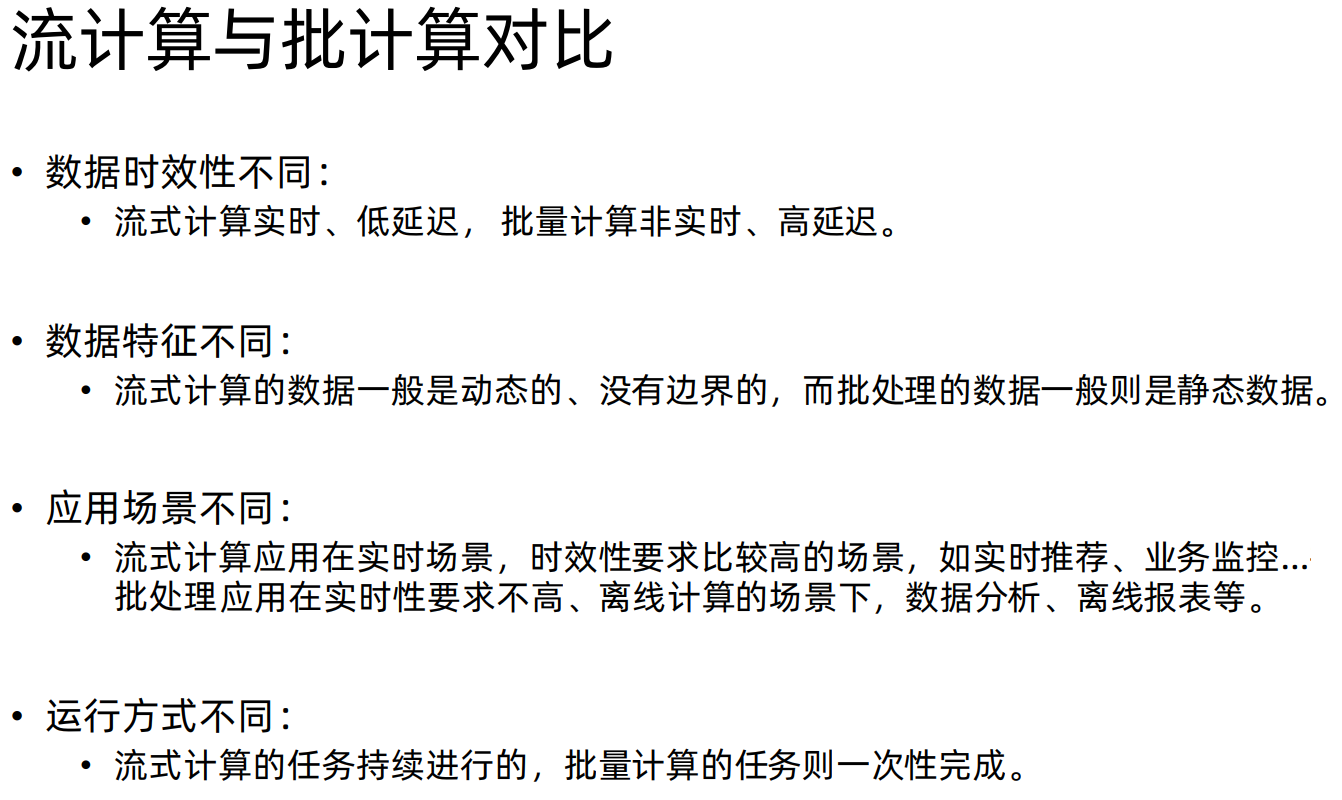
### 流处理和批处理

<https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/learn-flink/overview/>





* 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

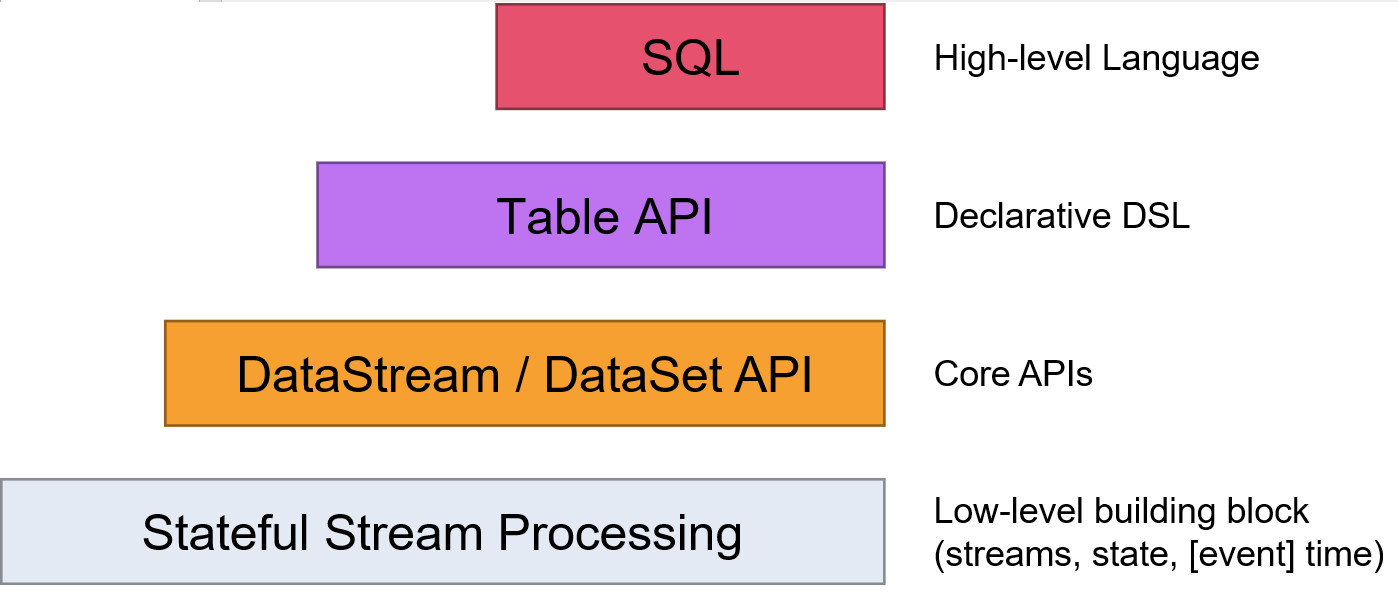
Flink 的核心 API 最初是针对特定的场景设计的，尽管 Table API / SQL 针对流处理和批处理已经实现了统一的 API，但当用户使用较底层的 API 时，仍然需要在批处理（DataSet API）和流处理（DataStream API）这两种不同的 API 之间进行选择。鉴于批处理是流处理的一种特例，将这两种 API 合并成统一的 API，有一些非常明显的好处，比如：

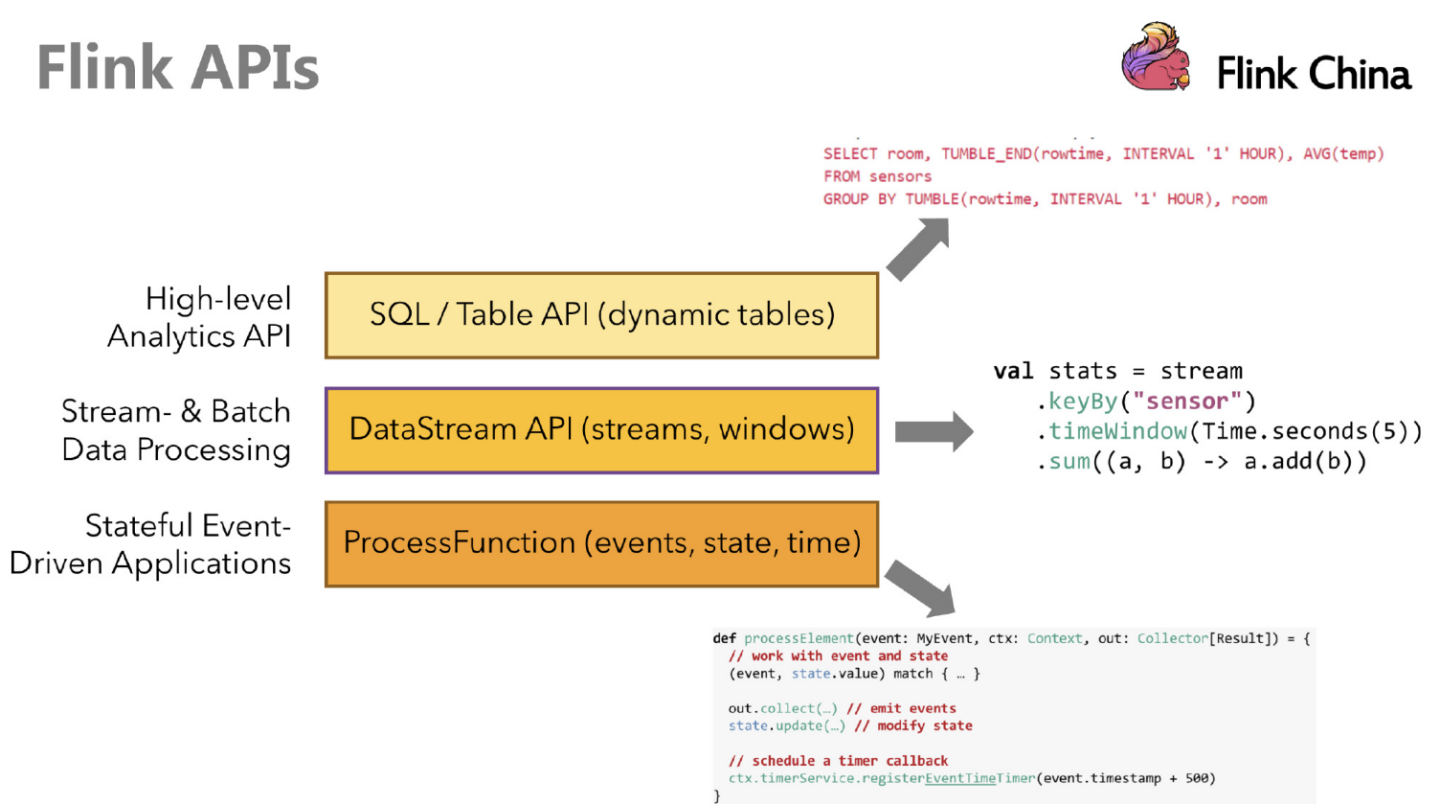
* + 可复用性：作业可以在流和批这两种执行模式之间自由地切换，而无需重写任何代码。因此，用户可以复用同一个作业，来处理实时数据和历史数据。
  + 维护简单：统一的 API 意味着流和批可以共用同一组 connector，维护同一套代码，并能够轻松地实现流批混合执行，例如 backfilling 之类的场景。

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



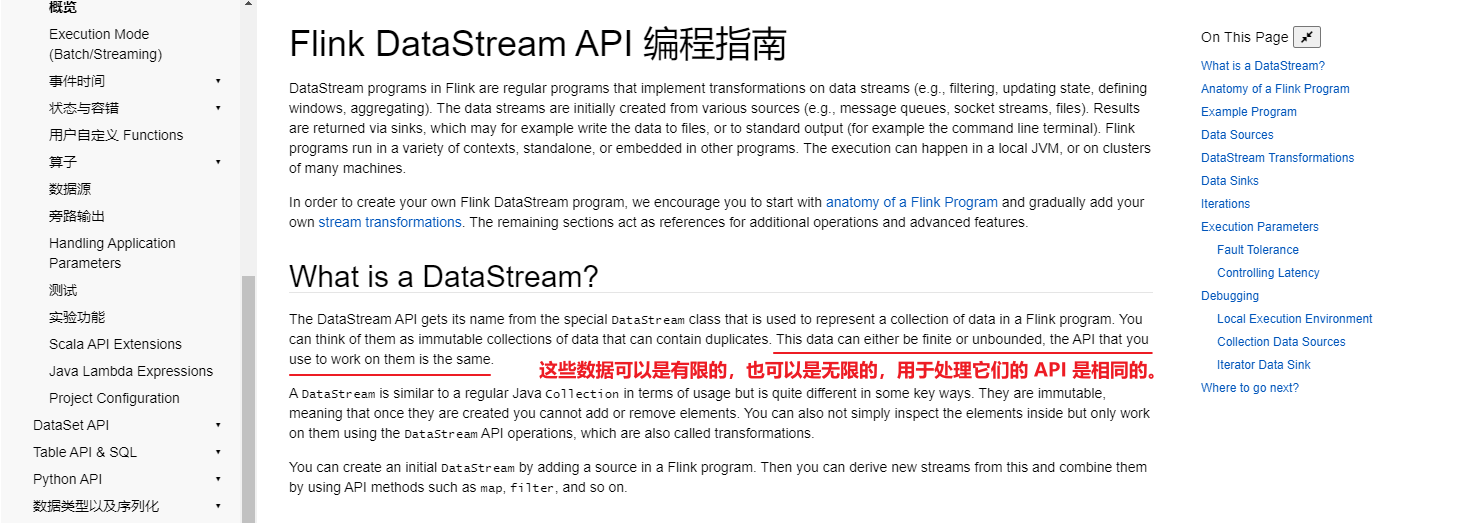


**注意：在Flink1.12时支持流批一体，DataSetAPI已经不推荐使用了，所以课程中除了个别案例使用DataSet外，后续其他案例都会优先使用DataStream流式API，既支持无界数据处理/流处理，也支持有界数据处理/批处理！当然Table&SQL-API会单独学习**

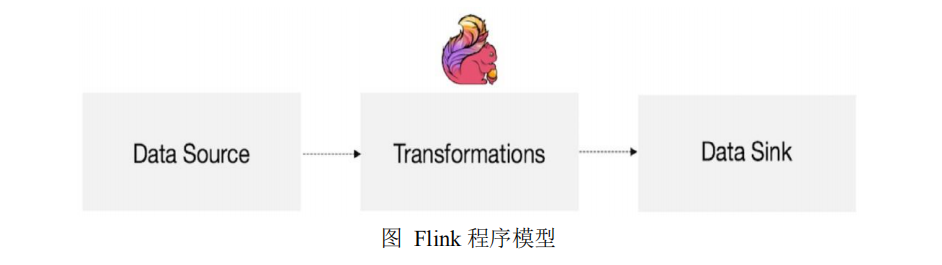
**<https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/dataset/overview/>**

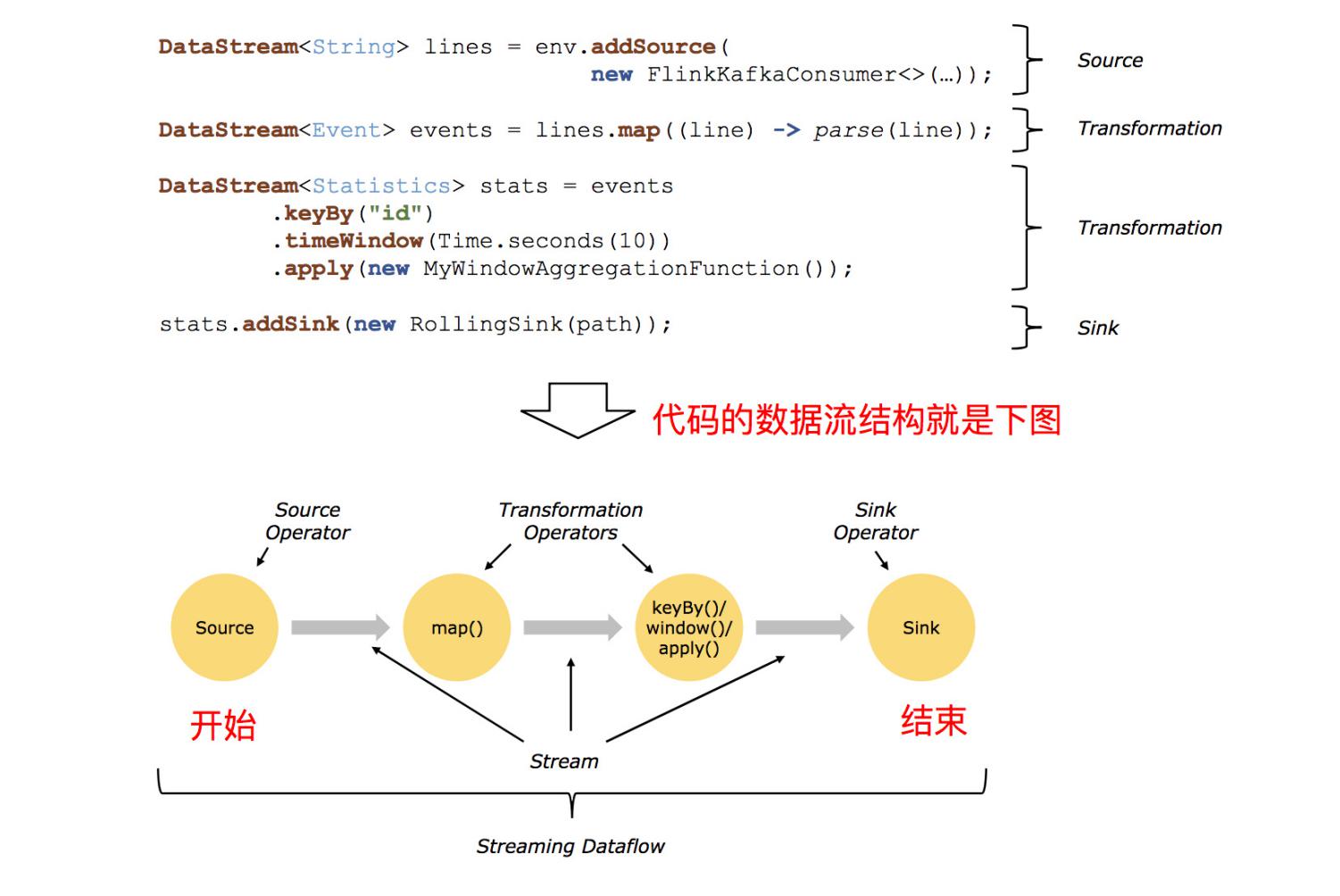


<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/datastream/overview/>



### Flink DataStream API编程模型





### Flink DataStream API 概览

通过之前的入门案例可以发现：

|  |
| --- |
| //1、设置运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  //2、配置数据源读取数据  DataStream<String> text = env.readTextFile ("input");  //3、进行一系列转换  DataStream<Tuple2<String, Integer>> counts = text.flatMap(new Tokenizer()).keyBy(0).sum(1);  //4、配置数据汇写出数据  counts.writeAsText("output");  //5、提交执行  env.execute("Streaming WordCount"); |

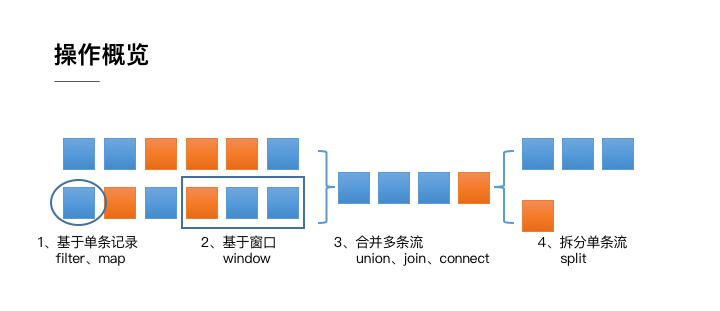
为了实现流式 Word Count：

* 要先获得一个 **StreamExecutionEnvironment** 对象。它是我们构建图过程中的上下文对象。
* 基于这个对象，我们可以添加一些算子
* 对于流处理程序，我们一般需要首先创建一个数据源去接入数据。在这个例子中，我们使用了 Environment 对象中内置的读取文件的数据源
* 上一步之后，我们拿到的是一个 **DataStream** 对象，它可以看作一个无限的数据集，可以在该集合上进行序列的操作
* 例如，在 Word Count 例子中，我们首先将每一条记录（即文件中的一行）分隔为单词，这是通过 FlatMap 操作来实现的。调用 FlatMap 将会在底层的 DAG 图中添加一个 FlatMap 算子。然后，我们得到了一个记录是单词的流
* 我们将流中的单词进行分组（keyBy），然后累积计算每一个单词的数据（sum(1)）
* 计算出的单词的数据组成了一个新的流，我们将它写入到输出文件中

最后，调用 **env.execute** 方法来开始程序的执行。需要强调的是，前面我们调用的所有方法，都不是在实际处理数据，而是在构通表达计算逻辑的 DAG 图。只有当我们将整个图构建完成并显式的调用 **Execute** 方法后，框架才会把计算图提供到集群中，接入数据并执行实际的逻辑。

基于 Flink 的 DataStream API 来编写流处理程序一般需要三步：

* 通过 Source 接入数据
* 进行一系统列的处理以及将数据写出
* 不要忘记显式调用 **Execute** 方式，否则前面编写的逻辑并不会真正执行。



从上面的例子中还可以看出，Flink DataStream API 的核心，就是代表流数据的 **DataStream** 对象。**整个计算逻辑图的构建就是围绕调用 DataStream 对象上的不同操作产生新的 DataStream 对象展开的**。整体来说，DataStream 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。

### Flink DataStream API编程基本步骤

* 获取执行环境（StreamExecutionEnvironment）
* 加载/创建初始数据集
* 对数据集进行各种转换操作（生成新的DataStream）
* 指定将计算的结果存储到哪个位置
* 触发APP执行（**execute**）

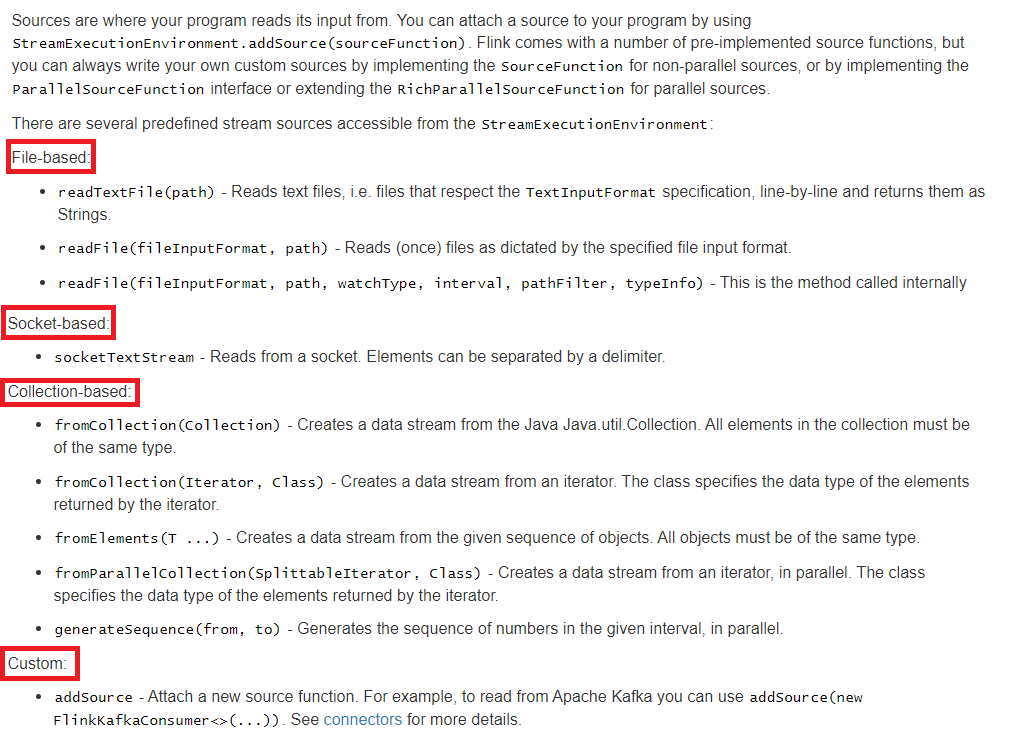
## 输入数据集Data Source

Data Sources 是什么呢？就字面意思其实就可以知道：**数据来源**。

Flink 做为一款流式计算框架，它可用来做批处理，也可以用来做流处理，这个 Data Sources 就是数据的来源地。

flink在**批/流处理**中常见的source主要有两大类。

* 预定义Source
  + 基于本地集合的source（Collection-based-source）
  + 基于文件的source（File-based-source）
  + 基于网络套接字（socketTextStream）
* 自定义Source



### 预定义Source

#### 基于本地集合的source（Collection-based-source）

在flink最常见的创建DataStream方式有四种：

* 使用env.fromElements()，这种方式也支持Tuple，自定义对象等复合形式。

注意：类型要一致，不一致可以用Object接收，但是使用会报错，比如：env.fromElements("haha", 1);



源码注释中有写：



* 使用env.fromCollection(),这种方式支持多种Collection的具体类型，如List，Set，Queue
* 使用env.generateSequence()方法创建基于Sequence的DataStream
* 使用env.fromSequence()方法创建基于开始和结束的DataStream

|  |
| --- |
| /\*\*  \* 使用flink中常见的dataSet对象创建的三种方式：  \* 1）env.fromElements()  \* 2）env.fromCollection()  \* 3）env.generateSequence()  \* 4）env.fromSequence();  \*/ public class BatchFromCollection {  /\*\*  \* 入口方法  \* @param args  \*/  public static void main(String[] args) throws Exception {  /\*\*  \* 开发步骤：  \* 1）获取执行环境（ExecutionEnvironment）  \* 2）加载/创建初始数据集  \* 3）对数据集进行各种转换操作（生成新的DataSet）  \* 4）指定将计算的结果输出  \* 5）提交任务（可选）  \*/   //TODO 1）获取执行环境（ExecutionEnvironment）  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //TODO 2）加载/创建初始数据集  //2.1）env.fromElements()  DataStream<String> ds1 = env.fromElements("hadoop", "spark", "spark", "flink");  List<Tuple2<String, Long>> tuple2List = new ArrayList<>();  tuple2List.add(Tuple2.of("hadoop", 1L));  tuple2List.add(Tuple2.of("spark", 2L));  tuple2List.add(Tuple2.of("flink", 3L));  DataStream<List<Tuple2<String, Long>>> ds2 = env.fromElements(tuple2List);   //2.2）env.fromCollection()  DataStream<String> ds3 = env.fromCollection(Arrays.asList("spark", "flink", "hadoop"));  //可以并行读取集合数据（流式开发中进行测试）  DataStream<Long> parallelCollection = env.fromParallelCollection(new NumberSequenceIterator(0L, 10L), TypeInformation.of(Long.TYPE)).setParallelism(3);   //2.3）env.generateSequence()  DataStream<Long> ds4 = env.generateSequence(1, 10);   DataStream<Long> ds5 = env.fromSequence(1, 10);   //TODO 4）指定将计算的结果输出  ds1.print();  ds2.print();  ds3.print();  ds4.print();  ds5.print();   //TODO 5）提交任务（可选）  env.execute();  } } |

#### 补充-在代码中指定并行度

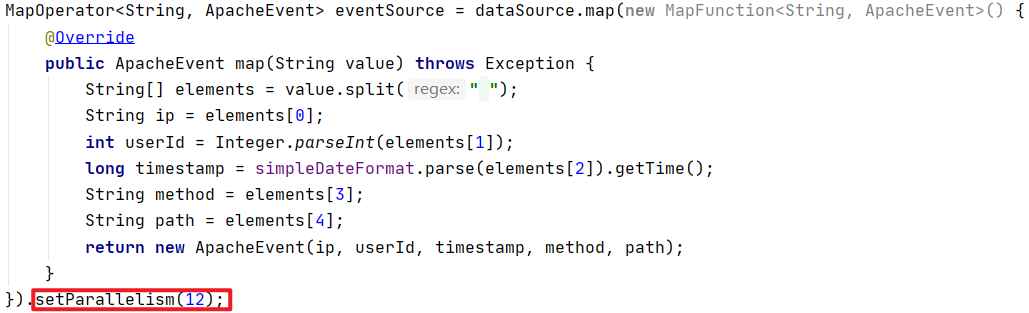
* 指定全局并行度：

env.setParallelism(12);

* 获得全局并行度：

env.getParallelism();

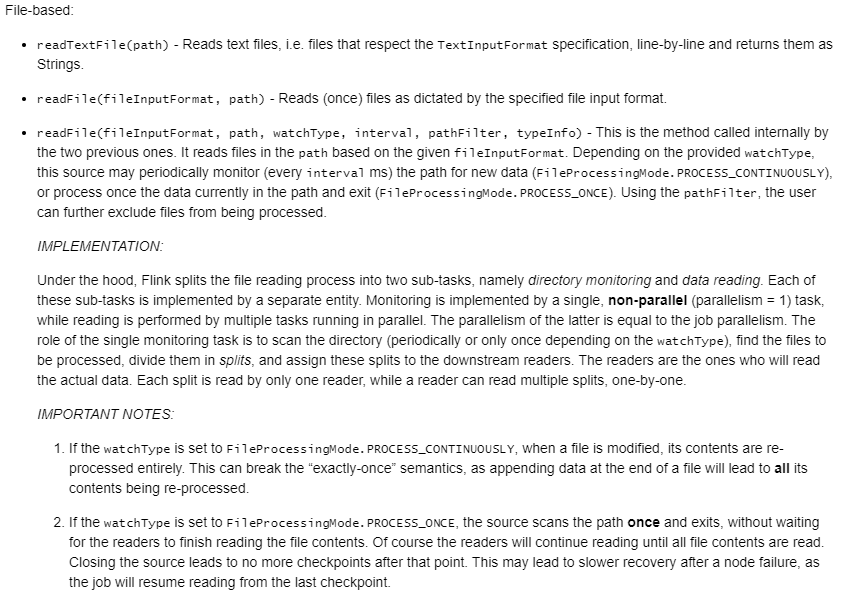
指定算子设置并行度：



获得指定算子的并行度：

eventSource.getParallelism();

#### 基于文件的source（File-based-source）



Flink支持直接从外部文件存储系统中读取文件的方式来创建Source数据源,Flink支持的方式有以下几种:

* readTextFile(path)-TextInputFormat逐行读取文本文件，即遵守规范的文件，并将它们作为字符串返回。
* readFile(fileInputFormat, path) - 按照指定的文件输入格式读取（一次）文件。
* readFile(fileInputFormat, path, watchType, interval, pathFilter, typeInfo)- 这是前两个内部调用的方法。它path根据给定的fileInputFormat. 根据提供的watchType，此源可能会定期（每interval毫秒）监视新数据的路径 ( FileProcessingMode.PROCESS\_CONTINUOUSLY)，或处理当前路径中的数据并退出 ( FileProcessingMode.PROCESS\_ONCE)。使用pathFilter，用户可以进一步排除正在处理的文件。

执行：

Flink 将文件读取过程拆分为两个子任务，即**目录监控**和**数据读取**。这些子任务中的每一个都由单独的实体实现。监控由单个非并行（并行性 = 1）任务实现，而读取由多个并行运行的任务执行。后者的并行度等于作业并行度。单个监控任务的作用是扫描目录（定期或仅一次，取决于watchType），找到要处理的文件，将它们拆分，并将这些拆分分配给下游阅读器。读者是将阅读实际数据的人。每个 split 只能由一个 reader 读取，而一个 reader 可以一个一个读取多个 split。

***重要笔记：***

* 如果watchType设置为**FileProcessingMode.PROCESS\_CONTINUOUSLY**，当文件被修改时，其内容将被完全重新处理。这可能会破坏“恰好一次”语义，因为在文件末尾附加数据将导致其所有内容都被重新处理。
* 如果watchType设置为**FileProcessingMode.PROCESS\_ONCE**，则源将扫描路径一次并退出，无需等待读取器完成读取文件内容。当然，读者会继续阅读，直到所有文件内容都读完。关闭源会导致在那之后不再有检查点。这可能会导致节点故障后恢复速度变慢，因为作业将从最后一个检查点恢复读取。

下面分别演示每个数据源的加载方式。

##### 批的方式读取文本文件

Flink的批处理可以直接通过readTextFile()方法读取文件来创建数据源,方法如下:

|  |
| --- |
| public class BatchFromFile{  public static void main(String[] args) throws Exception {  Configuration configuration = new Configuration();  configuration.setInteger("rest.port", 8181); //设置webui的端口  StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(configuration);   String path = "./data/input/wordcount.txt";   DataStreamSource<String> lines = env.readTextFile(path);   int parallelism = lines.getParallelism();   System.out.println("ReadTextFileDemo创建的DataStream的并行度为：" + parallelism);   lines.print();   env.execute();  } } |

##### 流的方式读取文件文件

Flink的批处理可以直接通过readFile()方法读取文件来创建数据源,方法如下:

|  |
| --- |
| //readFile创建的Source是一个多并行的Source，而且是一个无限的数据流，但是会重复读取数据 public class StreamFromFile {   public static void main(String[] args) throws Exception {   Configuration configuration = new Configuration();  configuration.setInteger("rest.port", 8181); //设置webui的端口  StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(configuration);   String path = "./data/input/wordcount.txt";   DataStreamSource<String> lines = env.readFile(new TextInputFormat(null), path,  FileProcessingMode.PROCESS\_CONTINUOUSLY, 2000);  int parallelism = lines.getParallelism();  System.out.println("readFile创建的DataStream的并行度为：" + parallelism);  lines.print();  env.execute();  } } |

#### 基于Socket的Source

socketTextStream(String hostname, int port) 方法是一个**非并行的Source**，该方法需要传入两个参数，第一个是指定的IP地址或主机名，第二个是端口号，即从指定的Socket读取数据创建DataStream。该方法还有多个重载的方法，其中一个是socketTextStream(String hostname, int port, String delimiter, long maxRetry)，这个重载的方法可以指定行分隔符和最大重新连接次数。这两个参数，默认行分隔符是”\n”，最大重新连接次数为0。

***提示：***

如果使用socketTextStream读取数据，在启动Flink程序之前，必须先启动一个Socket服务，为了方便，Mac或Linux用户可以在命令行终端输入nc -lk 8888启动一个Socket服务并在命令行中向该Socket服务发送数据。Windows用户可以在百度中搜索windows安装netcat命令。

|  |
| --- |
| //socketTextStream创建的DataStream，不论怎样，并行度永远是1 public class StreamSocketSource {   public static void main(String[] args) throws Exception {   //local模式默认的并行度是当前机器的逻辑核的数量  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   int parallelism0 = env.getParallelism();   System.out.println("执行环境默认的并行度：" + parallelism0);   DataStreamSource<String> lines = env.socketTextStream("localhost", 8888);   //获取DataStream的并行度  int parallelism = lines.getParallelism();   System.out.println("SocketSource的并行度：" + parallelism);   SingleOutputStreamOperator<String> words = lines.flatMap(new FlatMapFunction<String, String>() {  @Override  public void flatMap(String line, Collector<String> collector) throws Exception {  String[] words = line.split(" ");  for (String word : words) {  collector.collect(word);  }  }  });   int parallelism2 = words.getParallelism();   System.out.println("调用完FlatMap后DataStream的并行度：" + parallelism2);   words.print();   env.execute();  } } |

### 自定义Source

#### 基于随机生成DataSource

##### 自定义实现SourceFunction接口

除了预定义的Source外，我们还可以通过实现SourceFunction来自定义Source，然后通过StreamExecutionEnvironment.addSource(sourceFunction)添加进来。

* 示例:

自定义数据源, 每1秒钟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
* 开发步骤:
  + 创建订单实体类
  + 创建自定义数据源
    - 死循环生成订单
    - 随机构建订单信息
    - 上下文收集数据
    - 每隔一秒执行一次循环
  + 获取流处理环境
  + 使用自定义Source
  + 打印数据
  + 执行任务

###### 订单实体类

|  |
| --- |
| /\*\*  \* 定义订单的javaBean对象  \*/ @Data @AllArgsConstructor @NoArgsConstructor public class Order {  //订单id  private String id;  //用户id  private String userId;  //订单金额  private int money;  //时间戳  private Long timestamp; } |

###### 主代码

|  |
| --- |
| public class CustomerSourceWithoutParallelDemo {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   DataStreamSource<Order> mySource = env.addSource(new MySource());   mySource.print();  env.execute();  }   public static class MySource implements SourceFunction<Order> {  private boolean isRun = true; // 关闭循环标记  @Override  public void run(SourceContext<Order> ctx) throws Exception {  Random random = new Random();  while (isRun) {  String id = UUID.randomUUID().toString();  String userId = random.nextInt(99) + "";  int money = random.nextInt(999);  long time = System.currentTimeMillis();   ctx.collect(new Order(id, userId, money, time));  Thread.sleep(1000L);  }  }   @Override  public void cancel() {  this.isRun = false;  }  } } |

##### 实现ParallelSourceFunction创建可并行Source

|  |
| --- |
| /\*\*  \* 自定义多并行度Source  \*/ public class CustomerSourceWithParallelDemo {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  DataStreamSource<String> mySource = env.addSource(new MySource()).setParallelism(6);  mySource.print();  env.execute();  }  public static class MySource implements ParallelSourceFunction<String> {  @Override  public void run(SourceContext<String> ctx) throws Exception {  ctx.collect(UUID.randomUUID().toString());  /\*  如果不设置无限循环可以看出，设置了多少并行度就打印出多少条数据  \*/  }   @Override  public void cancel() {}  } } |

##### 实现RichParallelSourceFunction：创建并行并带有Rich功能的Source

|  |
| --- |
| /\*\*  \* 自定义一个RichParallelSourceFunction的实现  \*/ public class CustomerRichSourceWithParallelDemo {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   DataStreamSource<String> mySource = env.addSource(new MySource()).setParallelism(6);  mySource.print();   env.execute();  }   /\*  Rich 类型的Source可以比非Rich的多出有：  - open方法，实例化的时候会执行一次，多个并行度会执行多次的哦（因为是多个实例了）  - close方法，销毁实例的时候会执行一次，多个并行度会执行多次的哦  - getRuntime方法可以获得当前的Runtime对象（底层API）  \*/  public static class MySource extends RichParallelSourceFunction<String> {  @Override  public void open(Configuration parameters) throws Exception {  super.open(parameters);  System.out.println("open......");  }   @Override  public void close() throws Exception {  super.close();  System.out.println("close......");  }   @Override  public void run(SourceContext<String> ctx) throws Exception {  ctx.collect(UUID.randomUUID().toString());  }   @Override  public void cancel() {}  } } |

#### 基于mysql的source操作

上面我们已经使用了自定义数据源和Flink自带的Kafka source，那么接下来就模仿着写一个从 MySQL 中读取数据的 Source。

* 示例

自定义数据源, 读取MySql数据库(test)表(user)数据

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Username** | **Password** | **Name** |
| 1 | Zhangsan | 111111 | 张三 |
| 2 | Lisi | 222222 | 李四 |
| 3 | Wangwu | 333333 | 王五 |
| 4 | Zhaoliu | 444444 | 赵六 |
| 5 | Tianqi | 555555 | 田七 |

* 建表语句

|  |
| --- |
| **DROP** **TABLE** **IF** **EXISTS** `**user**`**;**  **CREATE** **TABLE** `**user**` **(**  `id` **int(**11**)** **NOT** **NULL,**  `username` **varchar(**255**)** **CHARACTER** **SET** utf8 **COLLATE** utf8\_general\_ci **NULL** **DEFAULT** **NULL,**  `password` **varchar(**255**)** **CHARACTER** **SET** utf8 **COLLATE** utf8\_general\_ci **NULL** **DEFAULT** **NULL,**  `name` **varchar(**255**)** **CHARACTER** **SET** utf8 **COLLATE** utf8\_general\_ci **NULL** **DEFAULT** **NULL,**  **PRIMARY** **KEY** **(**`id`**)** **USING** BTREE  **)** ENGINE **=** InnoDB **CHARACTER** **SET** **=** utf8 **COLLATE** **=** utf8\_general\_ci ROW\_FORMAT **=** Compact**;**  -- ----------------------------  -- Records of user  -- ----------------------------  **INSERT** **INTO** `**user**` **VALUES** **(**10**,** 'dazhuang'**,** '123456'**,** '大壮'**);**  **INSERT** **INTO** `**user**` **VALUES** **(**11**,** 'erya'**,** '123456'**,** '二丫'**);**  **INSERT** **INTO** `**user**` **VALUES** **(**12**,** 'sanpang'**,** '123456'**,** '三胖'**);**  **SET** FOREIGN\_KEY\_CHECKS **=** 1**;** |

* 相关依赖

|  |
| --- |
| <!-- 指定mysql-connector的依赖 -->  <dependency>  <groupId>**mysql**</groupId>  <artifactId>**mysql-connector-java**</artifactId>  <version>**5.1.48**</version>  </dependency> |

* 开发步骤
  + 自定义Source,继承自RichSourceFunction
  + 实现open方法
  + 实现run方法
    - 加载驱动
    - 创建连接
    - 创建PreparedStatement
    - 执行查询
    - 遍历查询结果,收集数据
  + 使用自定义Source
  + 打印结果
  + 执行任务
* 参考代码

|  |
| --- |
| /\*\*  \* 自定义Mysql Source  \*/ public class CustomerMysqlSourceDemo {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   // 获得自定义Source对象  DataStreamSource<UserInfo> mysqlSource = env.addSource(new MyMysqlSource());   mysqlSource.print();   env.execute("CustomerMySQLSourceDemo");  }   /\*\*  自定义Mysql Source实现类  \*/  public static class MyMysqlSource extends RichSourceFunction<UserInfo> {  private Connection connection = null; // 定义数据库连接对象  private PreparedStatement ps = null; // 定义PreparedStatement对象   /\*  使用open方法, 这个方法在实例化类的时候会执行一次, 比较适合用来做数据库连接  \*/  @Override  public void open(Configuration parameters) throws Exception {  super.open(parameters);  // 加载数据库驱动  Class.forName("com.mysql.jdbc.Driver");   // 创建数据库连接  String url = "jdbc:mysql://node01:3306/flinkdemo?useUnicode=true&characterEncoding=utf-8&useSSL=false";  this.connection = DriverManager.getConnection(url, "root", "123456");   // 准备PreparedStatement对象  this.ps = connection.prepareStatement("SELECT id, username, password, name FROM user");  }   /\*  使用close方法, 这个方法在销毁实例的时候会执行一次, 比较适合用来关闭连接  \*/  @Override  public void close() throws Exception {  super.close();   // 关闭资源  if (this.ps != null) this.ps.close();  if (this.connection != null) this.connection.close();  }   @Override  public void run(SourceContext<UserInfo> ctx) throws Exception {  ResultSet resultSet = ps.executeQuery();  while (resultSet.next()) {  int id = resultSet.getInt("id");  String username = resultSet.getString("username");  String password = resultSet.getString("password");  String name = resultSet.getString("name");   ctx.collect(new UserInfo(id, username, password, name));  }  }   @Override  public void cancel() {  System.out.println("任务被取消......");  }  }   /\*\*  数据定义类, POJO  \*/  @Data  @AllArgsConstructor  @NoArgsConstructor  public static class UserInfo {  int id;  String username;  String password;  String name;  } } |

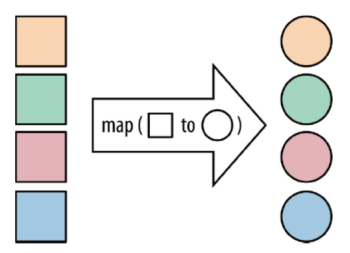
## Transformation

dataStream包括一系列的Transformation操作：

<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/datastream/operators/overview/>

### Map

将DataStream中的每一个元素转换为另外一个元素



* 示例

使用map操作，读取**apache.log**文件中的字符串数据转换成**ApacheLogEvent**对象

如：

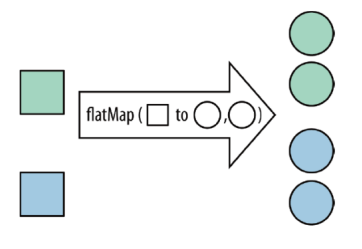
|  |
| --- |
| 86.149.9.216 10001 17/05/2015:10:05:30 GET /presentations/logstash-monitorama-2013/images/github-contributions.png  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:06:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:07:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:08:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:09:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:10:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:16:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:26:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css |

* 步骤
  + 获取ExecutionEnvironment运行环境
  + 使用readTextFile读取数据构建数据源
  + 创建一个**ApacheLogEvent**类
  + 使用map操作执行转换
  + 打印测试
* 参考代码

|  |
| --- |
| /\*\*  \* 使用map操作，读取apache.log文件中的字符串数据转换成ApacheLogEvent对象  \*/ public class MapDemo {  public static void main(String[] args) throws Exception {  /\*\*  \* 获取ExecutionEnvironment运行环境  \* 使用readTextFile读取数据构建数据源  \* 创建一个ApacheLogEvent类  \* 使用map操作执行转换  \* 打印测试  \*/   //TODO 获取ExecutionEnvironment运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //TODO 使用readTextFile读取数据构建数据源  DataStream<String> lines = env.readTextFile("./data/input/apache2.log");   //TODO 创建一个ApacheLogEvent类  //TODO 使用map操作执行转换  /\*\*  \* String：传入值类型  \* ApacheEvent：返回值类型  \*/  SingleOutputStreamOperator<ApacheEvent> apacheEventBean = lines.map(new MapFunction<String, ApacheEvent>() {  @Override  public ApacheEvent map(String line) throws Exception {  String[] elements = line.split(" ");  String ip = elements[0];  int userId = Integer.parseInt(elements[1]);  SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy:HH:mm:ss");  long timestamp = simpleDateFormat.parse(elements[2]).getTime();  String method = elements[3];  String path = elements[4];  return new ApacheEvent(ip, userId, timestamp, method, path);  }  });   //TODO 打印测试  apacheEventBean.print();   env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class ApacheEvent{  String ip; // 访问ip  int userId; // 用户id  long timestamp; // 访问时间戳  String method; // 访问方法  String path; // 访问路径  } } |

### FlatMap

将DataStream中的每一个元素转换为0...n个元素



* 示例

读取**flatmap.log**文件中的数据

如：

|  |
| --- |
| 张三,苹果手机,联想电脑,华为平板 李四,华为手机,苹果电脑,小米平板 |

转换为

|  |
| --- |
| 张三有苹果手机  张三有联想电脑  张三有华为平板  李四有…  …  … |

* 思路

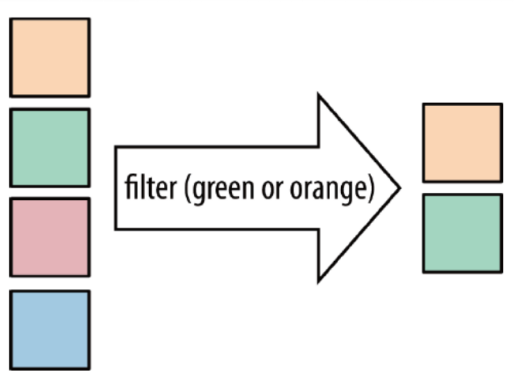
以上数据为一条转换为三条，显然，应当使用flatMap来实现分别在flatMap函数中构建三个数据，并放入到一个列表中

* 步骤
  + 构建批流理运行环境
  + 构建本地集合数据源
  + 使用flatMap将一条数据经过处理转换为三条数据
  + 使用逗号分隔字段
  + 分别构建三条数据
  + 打印输出
* 参考代码

|  |
| --- |
| /\*\*  \* 读取flatmap.log文件中的数据  \* 以上数据为一条转换为三条，显然，应当使用flatMap来实现分别在flatMap函数中构建三个数据，并放入到一个列表中  \*  \* 张三有苹果手机  \* 张三有联想电脑  \* 张三有华为平板  \* 李四有…  \* …  \* …  \*/ public class FlatMapDemo {  public static void main(String[] args) throws Exception {  /\*\*  \* 开发步骤：  \* 构建批处理运行环境  \* 构建本地集合数据源  \* 使用flatMap将一条数据经过处理转换为三条数据  \* 使用逗号分隔字段  \* 分别构建三条数据  \* 打印输出  \*/   //TODO 构建批处理运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //TODO 构建本地集合数据源  DataStream<String> lines = env.readTextFile("./data/input/flatmap.log");   //TODO 使用flatMap将一条数据经过处理转换为三条数据  //TODO 使用逗号分隔字段  //TODO 分别构建三条数据  SingleOutputStreamOperator<String> result = lines.flatMap(new FlatMapFunction<String, String>() {  @Override  public void flatMap(String line, Collector<String> collector) throws Exception {  String[] elements = line.split(",");  collector.collect(elements[0] + "有" + elements[1]);  collector.collect(elements[0] + "有" + elements[2]);  collector.collect(elements[0] + "有" + elements[3]);  }  });   //TODO 打印输出  result.print();   env.execute();  } } |

### [Filter](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)

[过滤出来一些符合条件的元素](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)

[](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)

* [示例：](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)

[读取](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)**[apache.log](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)**[文件中的访问日志数据，过滤出来以下访问IP是](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)**[83.149.9.216](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)**[的访问日志。](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)

|  |
| --- |
| [86.149.9.216 10001 17/05/2015:10:05:30 GET /presentations/logstash-monitorama-2013/images/github-contributions.png](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [10.0.0.1 10003 17/05/2015:10:06:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [10.0.0.1 10003 17/05/2015:10:07:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [10.0.0.1 10003 17/05/2015:10:08:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [10.0.0.1 10003 17/05/2015:10:09:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [10.0.0.1 10003 17/05/2015:10:10:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [10.0.0.1 10003 17/05/2015:10:16:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)  [10.0.0.1 10003 17/05/2015:10:26:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html) |

* [步骤](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)
  + [获取ExecutionEnvironment运行环境](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)
  + [使用fromCollection构建数据源](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)
  + [使用filter操作执行过滤](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)
  + [打印测试](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)
* [参考代码](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html)

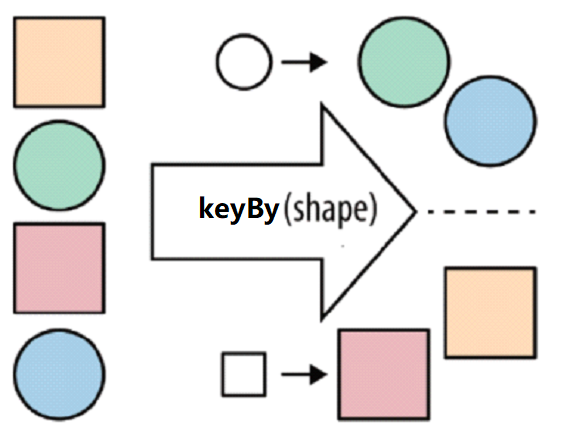
|  |
| --- |
| [/\*\*  \* 读取apache.log文件中的访问日志数据，过滤出来以下访问IP是83.149.9.216的访问日志。  \*/ public class FilterDemo {  public static void main(String[] args) throws Exception {  /\*\*  \* 获取ExecutionEnvironment运行环境  \* 使用fromCollection构建数据源  \* 使用filter操作执行过滤  \* 打印测试  \*/  //TODO 获取ExecutionEnvironment运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //TODO 使用fromCollection构建数据源  DataStream<String> lines = env.readTextFile("./data/input/apache.log");   //TODO 使用filter操作执行过滤(66.249.73.135)  SingleOutputStreamOperator<String> result = lines.filter(new FilterFunction<String>() {  @Override  public boolean filter(String line) throws Exception {  return line.contains("66.249.73.135");  }  });  //TODO 打印测试  result.print();   env.execute();  } }](https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html) |

### KeyBy

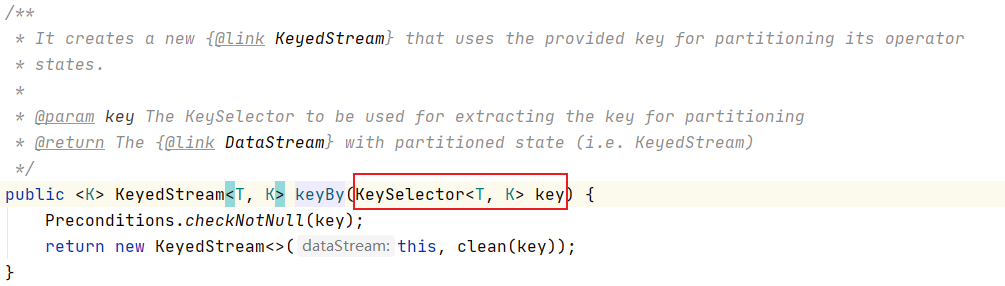
按照指定的key来对流中的数据进行分组，前面入门案例中已经演示过

注意:

流处理中没有**groupBy**,而是**keyBy**



* 函数说明



KeySelector对象可以支持**元组类型**，也可以支持**POJO**

* + 元组类型
    - 单个字段keyBy

|  |
| --- |
| //用字段位置（已经被废弃） wordAndOne.keyBy(0)  //用字段表达式 wordAndOne.keyBy(v -> v.f0) |

* + - 多个字段keyBy

|  |
| --- |
| //用字段位置 wordAndOne.keyBy(0, 1);  //用KeySelector wordAndOne.keyBy(new KeySelector<Tuple2<String, Integer>, Tuple2<String, Integer>>() {  @Override  public Tuple2<String, Integer> getKey(Tuple2<String, Integer> value) throws Exception {  return Tuple2.of(value.f0, value.f1);  } }); |
| 上述可用**lambda**简化 |
| wordAndOne.keyBy(  (KeySelector<Tuple2<String, Integer>, Tuple2<String, Integer>>) value ->  Tuple2.*of*(value.f0, value.f1) ); |

* + POJO

|  |
| --- |
| public class PeopleCount {  private String province;  private String city;  private Integer counts;  public PeopleCount() {  }  *//省略其他代码。。。* } |

* + - 单个字段keyBy

|  |
| --- |
| source.keyBy(a -> a.getProvince());  source.keyBy(PeopleCount::getProvince); |

* + - 多个字段keyBy

|  |
| --- |
| source.keyBy(new KeySelector<PeopleCount, Tuple2<String, String>>() {  @Override  public Tuple2<String, String> getKey(PeopleCount value) throws Exception {  return Tuple2.of(value.getProvince(), value.getCity());  } }); |
| 上述可用lambda简化 |
| map.keyBy(  (KeySelector<PeopleCount, Tuple2<String, String>>) value ->  Tuple2.*of*(value.getProvince(), value.getCity()) ); |

* 示例

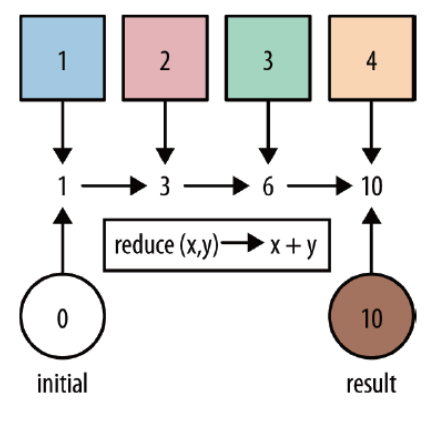
读取socket数据源, 进行单词的计数

* 开发步骤
  + 获取流处理运行环境
  + 设置并行度
  + 获取数据源
  + 转换操作
    - 以空白进行分割
    - 给每个单词计数1
    - 根据单词分组
  + 求和
  + 打印到控制台
  + 执行任务
* 参考代码

|  |
| --- |
| /\*\*  \* 演示keyBy方法  \* -- PS: 也使用了aggregate方法的sum求和  \*/ public class KeyByDemo {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   // 设置全局并行度为1  env.setParallelism(1);  // Get source  DataStreamSource<Tuple2<String, Integer>> source = env.fromElements(  Tuple2.of("篮球", 1),  Tuple2.of("篮球", 2),  Tuple2.of("篮球", 3),  Tuple2.of("足球", 3),  Tuple2.of("足球", 2),  Tuple2.of("足球", 3)  );   // 分组加聚合计算, 类似SQL的group by 后加聚合函数求每个组的数据  // 有一点要注意的是, SQL中是对分组后的每个组的全量数据做聚合计算, 是批计算  // 在流计算内,是来一条计算一条,也就是每个组的数据,挨个进行计算,求和累加,所以结果中最后一个打印的数据才是最终的求和结果  SingleOutputStreamOperator<Tuple2<String, Integer>> sum = source.keyBy(0).sum(1);   // 如果不分组的话, sum的结果是 1+2+3+3+2+3 = 14 分组后是 篮球 6 足球 8  sum.print();  env.execute();  } } |

### Reduce

可以对一个 dataset 或者一个 group 来进行聚合计算，最终聚合成一个元素



* 示例

读取**apache.log**日志，统计ip地址访问pv数量，使用 reduce 操作聚合成一个最终结果

|  |
| --- |
| 86.149.9.216 10001 17/05/2015:10:05:30 GET /presentations/logstash-monitorama-2013/images/github-contributions.png  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  83.149.9.216 10002 17/05/2015:10:06:53 GET /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:06:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:07:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:08:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:09:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:10:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:16:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css  10.0.0.1 10003 17/05/2015:10:26:53 POST /presentations/logstash-monitorama-2013/css/print/paper.css |

结果类似：

**(86.149.9.216,1)**

**(10.0.0.1,7)**

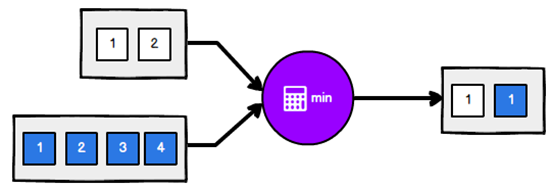
**(83.149.9.216,6)**

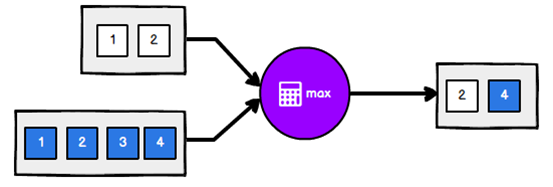
* 步骤
  + 获取 ExecutionEnvironment 运行环境
  + 使用 readTextFile 构建数据源
  + 使用 reduce 执行聚合操作
  + 打印测试
* 参考代码

|  |
| --- |
| /\*\*  \* 读取apache.log日志，统计ip地址访问pv数量，使用 reduce 操作聚合成一个最终结果  \*/ public class ReduceDemo {  public static void main(String[] args) throws Exception {  /\*\*  \* 获取 ExecutionEnvironment 运行环境  \* 使用 readTextFile 构建数据源  \* 使用 reduce 执行聚合操作  \* 打印测试  \*/   //TODO 获取 ExecutionEnvironment 运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //TODO 使用 readTextFile 构建数据源  DataStream<String> lines = env.readTextFile("./data/input/apache.log");   //TODO 使用 reduce 执行聚合操作  SingleOutputStreamOperator<Tuple2<String, Integer>> ipAndOne = lines.map(new MapFunction<String, Tuple2<String, Integer>>() {  @Override  public Tuple2<String, Integer> map(String line) throws Exception {  String[] dataArray = line.split(" ");  return Tuple2.of(dataArray[0], 1);  }  });   SingleOutputStreamOperator<Tuple2<String, Integer>> result = ipAndOne.keyBy(0).reduce(new ReduceFunction<Tuple2<String, Integer>>() {  /\*\*  \* @param tuple1 初始值或者临时累加值  \* @param tuple2 下一个值对象  \* @return  \* @throws Exception  \*/  @Override  public Tuple2<String, Integer> reduce(Tuple2<String, Integer> tuple1, Tuple2<String, Integer> tuple2) throws Exception {  return Tuple2.of(tuple1.f0, tuple1.f1 + tuple2.f1);  }  });   //TODO 打印测试  result.print();   env.execute();  } } |

### minBy和maxBy

获取指定字段的最大值、最小值





#### 第一种场景

* 示例

编写Flink程序，接收socket的单词数据，并以逗号进行单词拆分打印。

|  |
| --- |
| spark,2 spark,5 hadoop,7 hadoop,3 |

* 步骤
  + 获取 ExecutionEnvironment 运行环境
  + 使用 socketTextStream 构建数据源
  + 使用 keyBy 按照单词进行分组
  + 使用 maxBy、minBy对每个分组进行操作
  + 打印测试
* 参考代码

|  |
| --- |
| //分组后，求组内最小值 public class MinMinByDemo {  public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //调用Source创建DataStream  //spark,2  //spark,5  //hadoop,7  //hadoop,3  DataStream<String> lines = env.socketTextStream("node01", 8888);   SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndCount = lines.map(new MapFunction<String, Tuple2<String, Integer>>() {  @Override  public Tuple2<String, Integer> map(String value) throws Exception {  String[] fields = value.split(",");  String word = fields[0];  int count = Integer.parseInt(fields[1]);  return Tuple2.of(word, count);  }  });   KeyedStream<Tuple2<String, Integer>, String> keyed = wordAndCount.keyBy(w -> w.f0);   keyed.minBy(1).print("最小数据>>>");  keyed.maxBy(1).print("最大数据>>>");   env.execute();  } } |

#### 第二种场景

* 示例

编写Flink程序，接收socket的单词数据，并以逗号进行单词拆分打印。

|  |
| --- |
| 辽宁,沈阳,1000 北京,朝阳,8000 辽宁,朝阳,1000 辽宁,朝阳,1000 辽宁,沈阳,2000 北京,朝阳,1000 辽宁,大连,3000 辽宁,铁岭,500 |

* 步骤
  + 获取 ExecutionEnvironment 运行环境
  + 使用 socketTextStream 构建数据源
  + 使用 keyBy 按照单词进行分组
  + 使用 maxBy、minBy对每个分组进行操作
  + 打印测试
* 参考代码

|  |
| --- |
| //分组后，求组内最小值 public class MinMinByDemo2 {   public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  //调用Source创建DataStream  //辽宁,沈阳,1000  //北京,朝阳,8000  //辽宁,朝阳,1000  //辽宁,朝阳,1000  //辽宁,沈阳,2000  //北京,朝阳,1000  //辽宁,大连,3000  //辽宁,铁岭,500  DataStream<String> lines = env.socketTextStream("node01", 8888);  SingleOutputStreamOperator<Tuple3<String, String, Double>> pcm = lines.map(new MapFunction<String, Tuple3<String, String, Double>>() {   @Override  public Tuple3<String, String, Double> map(String value) throws Exception {  String[] fields = value.split(",");  String province = fields[0];  String city = fields[1];  double money = Double.parseDouble(fields[2]);  return Tuple3.of(province, city, money);  }  });   KeyedStream<Tuple3<String, String, Double>, String> keyed = pcm.keyBy(t -> t.f0);   SingleOutputStreamOperator<Tuple3<String, String, Double>> res = keyed.minBy(2, false);   res.print();  env.execute();  } } |

### min max 和 minBy maxBy的区别

min 和 max 方式 对比minBy 和maxBy的区别在：

以min和minBy举例：

首先： min方法(aggregate方法的简写）只能用于元组, 而minBy可以用于集合数据(DataSet)

**另外最重要的区别在于，计算逻辑不同，尽管都是求最小值，但是：**

**Min在计算的过程中，会记录最小值，对于其它的列，会取最后一次出现的，然后和最小值组合形成结果返回**

**minBy在计算的过程中，当遇到最小值后，将第一次出现的最小值所在的整个元素返回。**

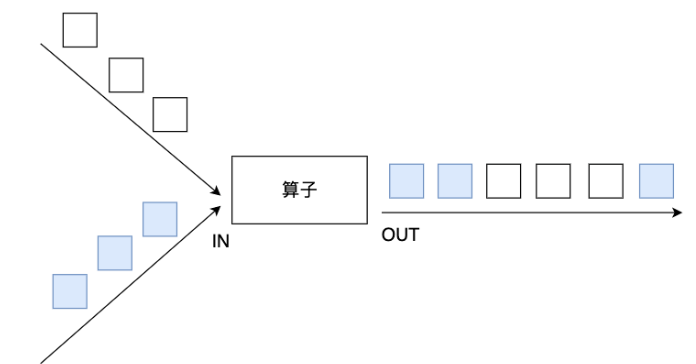
* 示例代码：

|  |
| --- |
| /\*\*  \* min 和 minBy  \* max 和 maxBy  \* 的区别演示  \*/ public class MinVSMinByDemo {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   DataStream<Tuple3<Integer, Integer, Integer>> source = env.fromElements(  Tuple3.of(1, 3, 2),  Tuple3.of(1, 1, 2),  Tuple3.of(1, 2, 3),  Tuple3.of(1, 111, 1),  Tuple3.of(1, 1, 1),  Tuple3.of(1, 2, 0),  Tuple3.of(1, 33, 2)  );   source.keyBy(t->t.f0).min(2).print("min>>>"); // 结果 1,2,0 可以看出 0取的是最小值, 1,2 取的是最后一个  source.keyBy(t->t.f0).minBy(2).printToErr("minBy>>>"); // 结果是1,3,0 可以看出 当遇到最小值后, 就不会变了   /\*  \* min 会取最小值记录下来, 其它字段按照顺序填充, 最终将最后一个元素的其它字段和最小值组合形成结果返回  \* minBy 会判断最小值, 将第一个出现的最小值所在的那一行的结果保留, 最终返回出去  \* max maxBy同理  \*/  env.execute();  } } |

### Union

将多个DataSet合并成一个DataSet

**【注意】：union合并的DataSet的类型必须是一致的**



* 示例

将以下数据进行取并集操作

数据集1

|  |
| --- |
| "hadoop", "hive", "flume" |

数据集2

|  |
| --- |
| "hadoop", "hive", "spark" |

* 步骤
  + 构建批处理运行环境
  + 使用 fromCollection 创建两个数据源
  + 使用 union 将两个数据源关联在一起
  + 打印测试

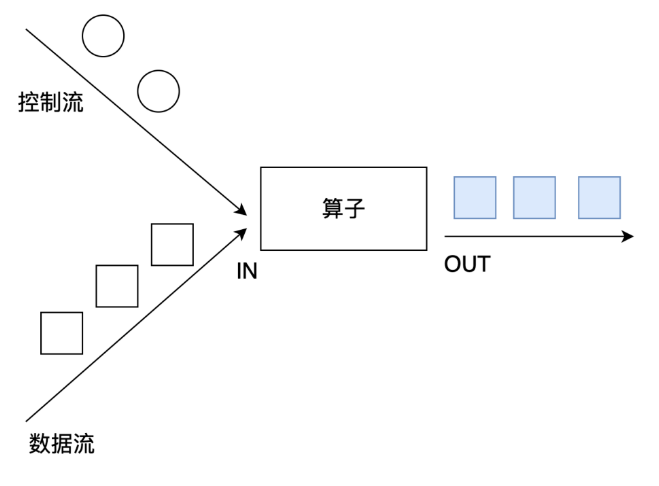
**注意：union可以取并集，但是不会去重。**

* 参考代码

|  |
| --- |
| /\*\*  \* 使用union实现  \* 将以下数据进行取并集操作  \* 数据集1  \* "hadoop", "hive", "flume"  \* 数据集2  \* "hadoop", "hive", "spark"  \*  \* 注意：  \* 1：合并后的数据不会自动去重  \* 2：要求数据类型必须一致  \*/ public class UnionDemo {  public static void main(String[] args) throws Exception {  /\*\*  \* 实现步骤：  \* 1）初始化flink的流处理的运行环境  \* 2）加载/创建数据源  \* 3）处理数据  \* 4）打印输出  \* 5）递交执行作业  \*/  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   DataStreamSource<String> ds1 = env.fromElements("hadoop", "hive", "flume");  DataStreamSource<String> ds2 = env.fromElements("hadoop","hive","spark");  DataStream<String> result = ds1.union(ds2);  result.printToErr();   env.execute();  } } |

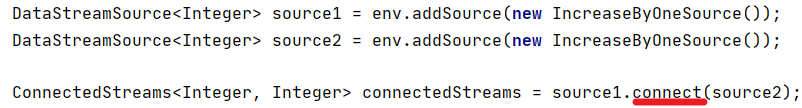
### Connect

DataStream,DataStream → ConnectedStreams：连接两个保持他们类型的数据流，两个数据流被 Connect 之后，只是被放在了一个同一个流中，内部依然保持各自的数据和形式不发生任何变化，两个**流相互独立, 作为对比Union后是真的变成一个流了**



* 示例

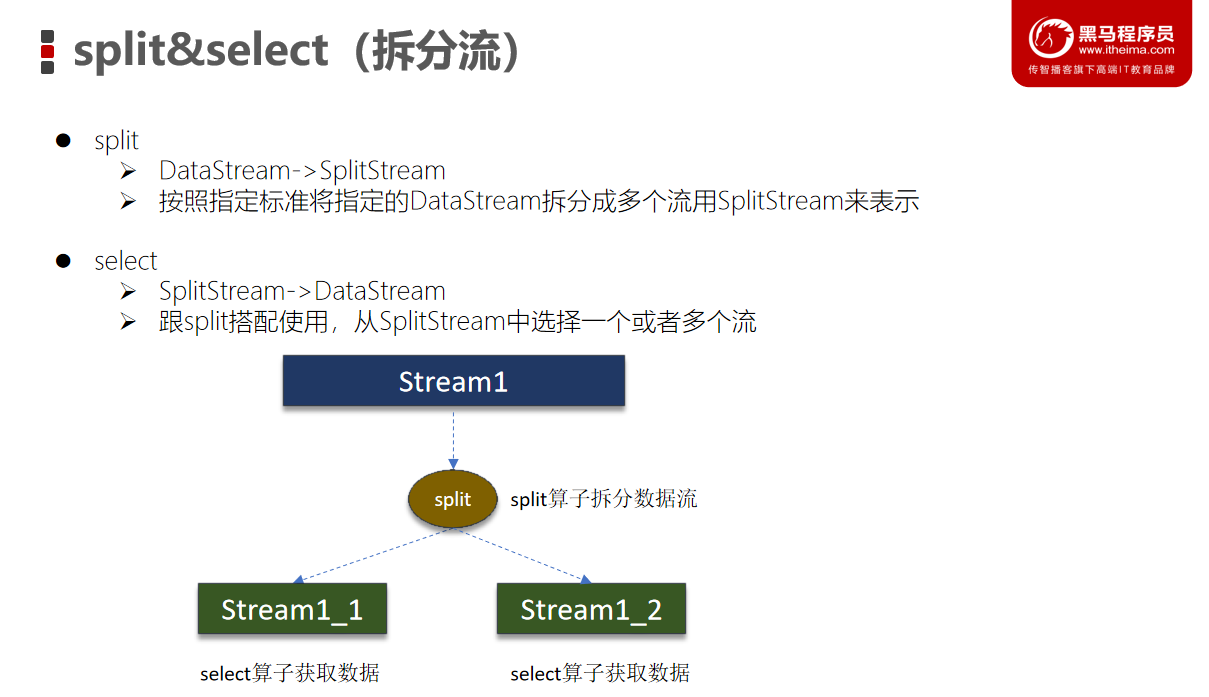
读取两个不同类型的数据源，使用connect进行合并打印。



* 开发步骤
  + 创建流式处理环境
  + 添加两个自定义数据源
  + 使用connect合并两个数据流,创建ConnectedStreams对象
  + 遍历ConnectedStreams对象,转换为DataStream
  + 打印输出,设置并行度为1
  + 执行任务
* 参考代码

|  |
| --- |
| /\*\*  \* 读取两个数据流（生成两个不同类型的数据流），使用connect进行合并输出  \* 和union类似，但是connect只能连接两个流，两个流之间的数据类型可以不同，对两个流的数据可以分别应用不同的处理逻辑  \*/ public class ConnectDemo {  public static void main(String[] args) throws Exception {  /\*\*  \* 实现步骤：  \* 1）初始化flink流处理的运行环境  \* 2）构建两个不同类型数据的数据流  \* 3）对连接后的流数据进行业务处理  \* 4）打印输出  \* 5）启动作业  \*/   //TODO 1）初始化flink流处理的运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //TODO 2）构建两个不同类型数据的数据流  DataStreamSource<Long> longDataStreamSource = env.addSource(new MyNoParallelSource());  DataStreamSource<Long> longDataStreamSource2 = env.addSource(new MyNoParallelSource());    //TODO 3）对连接后的流数据进行业务处理  SingleOutputStreamOperator<String> strDataStreamSource = longDataStreamSource2.map(new MapFunction<Long, String>() {  @Override  public String map(Long aLong) throws Exception {  return "str\_" + aLong;  }  });   ConnectedStreams<Long, String> connectedStreams = longDataStreamSource.connect(strDataStreamSource);  //对连接后的流应用不同的业务逻辑  SingleOutputStreamOperator<Object> result = connectedStreams.map(new CoMapFunction<Long, String, Object>() {  @Override  public Object map1(Long value) throws Exception {  return value;  }   @Override  public Object map2(String value) throws Exception {  return value;  }  });   //TODO 4）打印输出  result.print();   //TODO 5）启动作业  env.execute();  }   public static class MyNoParallelSource implements SourceFunction<Long> {  //定义一个变量，是否循环生成数据  private boolean isRunning = true;  private Long count = 0L;   /\*\*  \* 这是主要的方法，启动一个数据源  \* 实现数据的生成操作  \* @param ctx  \* @throws Exception  \*/  @Override  public void run(SourceContext<Long> ctx) throws Exception {  //不断生成订单数据  while (isRunning){  count+=1;  //收集数据返回  ctx.collect(count);   //每隔一秒钟生成一条订单数据  TimeUnit.SECONDS.sleep(1);  }  }   /\*\*  \* 取消数据的生成操作  \*/  @Override  public void cancel() {  isRunning = false;  }  } } |

### split、select和Side Outputs



**Split**就是将一个DataStream分成两个或者多个DataStream

**Select**就是获取分流后对应的数据

简单认为就是, Split会给数据打上标记

然后通过Select, 选择标记来划分出不同的Stream

效果类似KeyBy分流，但是比KeyBy更自由些，可以自由打标记并进行分流。

**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

* 示例

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

* 开发步骤
  + 创建流处理环境
  + 设置并行度
  + 加载本地集合
  + 数据分流,分为奇数和偶数
  + 获取分流后的数据
  + 打印数据
  + 执行任务
* 参考代码

|  |
| --- |
| /\*\*  \* split就是将一个DataStream拆分成两个或者多个DataStream  \* select就是获取分流后对应的数据  \*/ public class StreamSplitDemo {  public static void main(String[] args) throws Exception {  //TODO 0.env  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);   //TODO 1.source  DataStreamSource<Integer> ds = env.fromElements(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);   //TODO 2.transformation  //需求:对流中的数据按照奇数和偶数拆分并选择  OutputTag<Integer> oddTag = new OutputTag<>("奇数", TypeInformation.of(Integer.class));  //OutputTag<Integer> evenTag = new OutputTag<>("偶数",TypeInformation.of(Integer.class));  OutputTag<Integer> evenTag = new OutputTag("偶数", TypeInformation.of(Integer.class));  //https://www.bilibili.com/video/BV1xJ411n77R  //TypeInformation<Tuple2<String, Long>> info = TypeInformation.of(new TypeHint<Tuple2<String, Long>>(){});  //OutputTag<Tuple3<String,Integer,Integer>> tag = new OutputTag("偶数",TypeInformation.of(new TypeHint<Tuple3<String, Integer, Integer>>(){}));  //OutputTag<Tuple2<String,Integer>> tag2 = new OutputTag<>("",TypeInformation.of(Tuple2<String, Long>));   /\*  public abstract class ProcessFunction<I, O> extends AbstractRichFunction {  public abstract void processElement(I value, ProcessFunction.Context ctx, Collector<O> out) throws Exception;  }  \*/  SingleOutputStreamOperator<Integer> result = ds.process(new ProcessFunction<Integer, Integer>() {  @Override  public void processElement(Integer value, Context ctx, Collector<Integer> out) throws Exception {  //out收集完的还是放在一起的,ctx可以将数据放到不同的OutputTag  if (value % 2 == 0) {  ctx.output(evenTag, value);  } else {  ctx.output(oddTag, value);  }  }  });   DataStream<Integer> oddResult = result.getSideOutput(oddTag);  DataStream<Integer> evenResult = result.getSideOutput(evenTag);   //TODO 3.sink  System.out.println(oddTag);//OutputTag(Integer, 奇数)  System.out.println(evenTag);//OutputTag(Integer, 偶数)  oddResult.print("奇数:");  evenResult.print("偶数:");   //TODO 4.execute  env.execute();  } } |

### [Iterate](https://my.oschina.net/u/4328218/blog/3479223" \t "https://my.oschina.net/u/4328218/blog/_blank)

基本概念：在流中创建“反馈（feedback）”循环，通过将一个算子的输出重定向到某个先前的算子。这对于定义不断更新模型的算法特别有用。

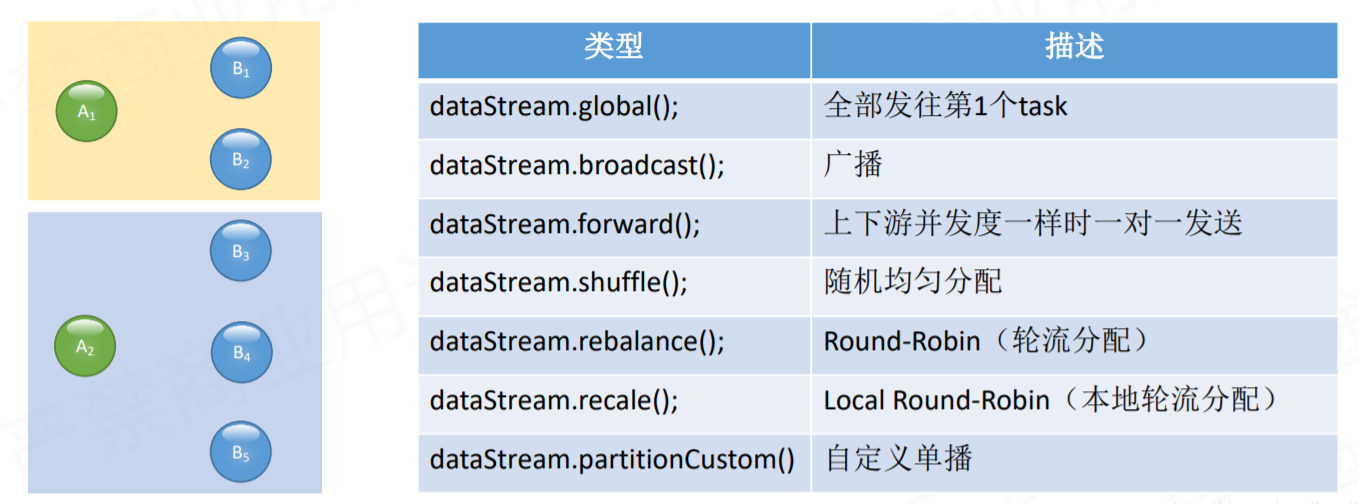
迭代的数据流向：DataStream → IterativeStream → DataStream

* 参考代码

|  |
| --- |
| //Iterate迭代流式计算 public class IterateDemo {   public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //10  DataStreamSource<String> strs = env.socketTextStream("node01", 9999);   DataStream<Long> numbers = strs.map(Long::parseLong);   //调用iterate方法 DataStream -> IterativeStream  //对Nums进行迭代（不停的输入int的数字）  IterativeStream<Long> iteration = numbers.iterate();   //IterativeStream -> DataStream  //对迭代出来的数据进行运算 //对输入的数据应用更新模型，即输入数据的处理逻辑  DataStream<Long> iterationBody = iteration.map(new MapFunction<Long, Long>() {  @Override  public Long map(Long value) throws Exception {  System.out.println("iterate input =>" + value);  return value -= 2;  }  });  //只要满足value > 0的条件，就会形成一个回路，重新的迭代，即将前面的输出作为输入，在进行一次应用更新模型，即输入数据的处理逻辑  DataStream<Long> feedback = iterationBody.filter(new FilterFunction<Long>() {  @Override  public boolean filter(Long value) throws Exception {  return value > 0;  }  });  //传入迭代的条件  iteration.closeWith(feedback);   //不满足迭代条件的最后要输出  DataStream<Long> output = iterationBody.filter(new FilterFunction<Long>() {  @Override  public boolean filter(Long value) throws Exception {  return value <= 0;  }  });   //数据结果  output.printToErr("output value:");  env.execute();  } } |

### 物理分区

Flink 也提供以下方法让用户根据需要在数据转换完成后对数据分区进行更细粒度的配置。



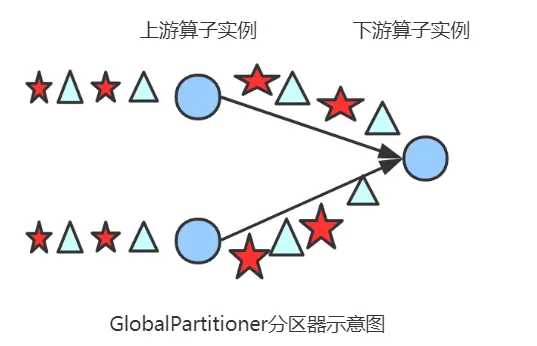
#### Global Partitioner

该分区器会将所有的数据都发送到下游的某个算子实例(subtask id = 0)

* 源码解读



* 图解



* 示例：

编写Flink程序，接收socket的单词数据，并将每个字符串均匀的随机划分到每个分区。

* 示例代码：

|  |
| --- |
| public class GlobalPartitioningDemo {   public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new Configuration());   //Source是一个非并行的Source  //并行度是1  DataStreamSource<String> lines = env.socketTextStream("node01", 9999);   //并行度2  SingleOutputStreamOperator<String> mapped = lines.map(new RichMapFunction<String, String>() {  @Override  public String map(String value) throws Exception {  int indexOfThisSubtask = getRuntimeContext().getIndexOfThisSubtask();  return value + " : " + indexOfThisSubtask;  }  }).setParallelism(1);   //shuffle  DataStream<String> global = mapped.global();   global.addSink(new RichSinkFunction<String>() {  @Override  public void invoke(String value, Context context) throws Exception {  int index = getRuntimeContext().getIndexOfThisSubtask();  System.out.println(value + " -> " + index);  }  });   env.execute();  } } |

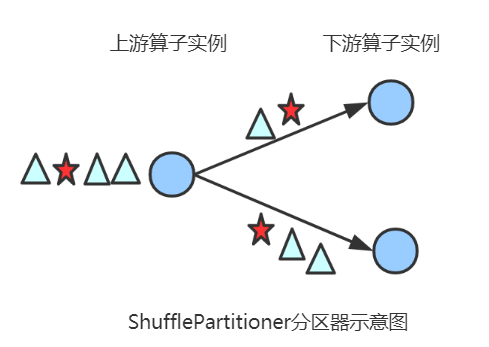
#### Shuffle Partitioner

根据均匀分布随机划分元素。

* 源码解读



* 图解



* 示例：

编写Flink程序，接收socket的单词数据，并将每个字符串均匀的随机划分到每个分区。

* 示例代码：

|  |
| --- |
| public class ShufflePartitioningDemo {  public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new Configuration());   //Source是一个非并行的Source  //并行度是1  DataStreamSource<String> lines = env.socketTextStream("node01", 9999);   //并行度2  SingleOutputStreamOperator<String> mapped = lines.map(new RichMapFunction<String, String>() {  @Override  public String map(String value) throws Exception {  int indexOfThisSubtask = getRuntimeContext().getIndexOfThisSubtask();  return value + " : " + indexOfThisSubtask;  }  }).setParallelism(1);   //shuffle  DataStream<String> shuffled = mapped.shuffle();   shuffled.addSink(new RichSinkFunction<String>() {  @Override  public void invoke(String value, Context context) throws Exception {  int index = getRuntimeContext().getIndexOfThisSubtask();  System.out.println(value + " -> " + index);  }  });  env.execute();  } } |

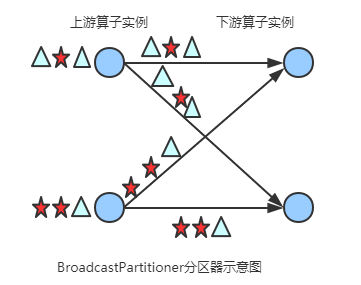
#### Broadcast Partitioner

发送到下游所有的算子实例

* 源码解读



* 图解



* 示例：

编写Flink程序，接收socket的单词数据，并将每个字符串广播到每个分区。

* 示例代码：

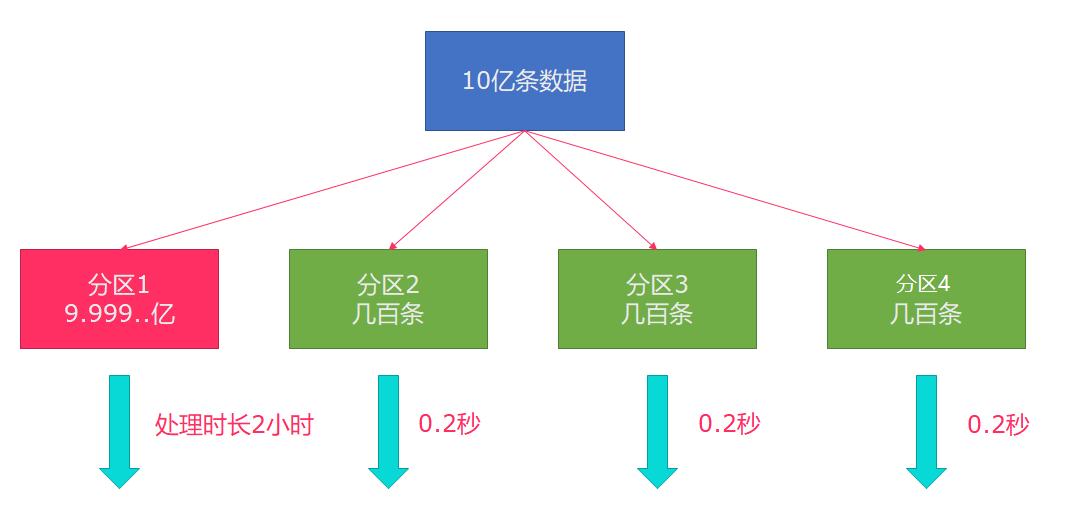
|  |
| --- |
| public class BroadcastPartitioningDemo {  public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new Configuration());   //Source是一个非并行的Source  //并行度是1  DataStreamSource<String> lines = env.socketTextStream("node01", 8888);   //并行度2  SingleOutputStreamOperator<String> mapped = lines.map(new RichMapFunction<String, String>() {  @Override  public String map(String value) throws Exception {  int indexOfThisSubtask = getRuntimeContext().getIndexOfThisSubtask();  return value + " : " + indexOfThisSubtask;  }  }).setParallelism(1);   //广播，上游的算子将一个数据广播到下游所以的subtask  DataStream<String> shuffled = mapped.broadcast();   shuffled.addSink(new RichSinkFunction<String>() {  @Override  public void invoke(String value, Context context) throws Exception {  int index = getRuntimeContext().getIndexOfThisSubtask();  System.out.println(value + " -> " + index);  }  });   env.execute();  } } |

#### Rebalance Partitioner

通过循环的方式依次发送到下游的task

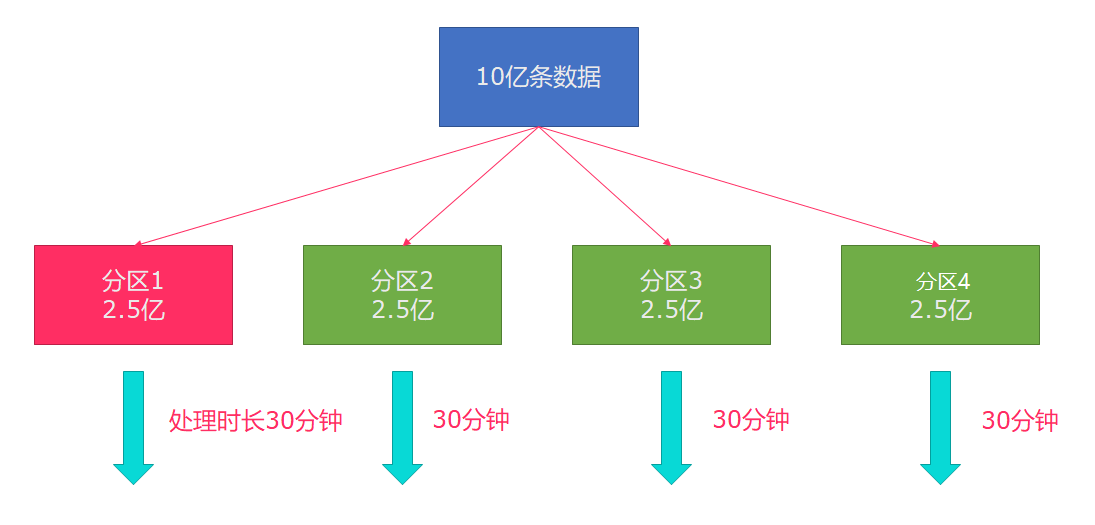
* 问题产生背景

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：



这个时候本来总体数据量只需要10分钟解决的问题，出现了数据倾斜，机器1上的任务需要4个小时才能完成，那么其他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；

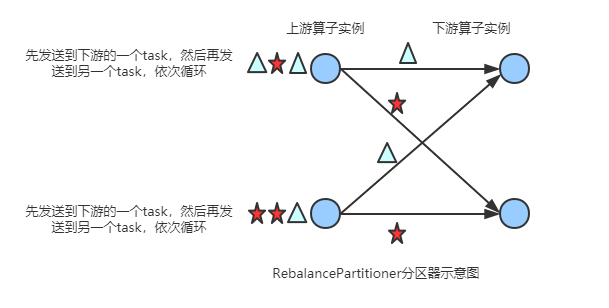
所以在实际的工作中，出现这种情况比较好的解决方案就是本节课要讲解的—rebalance



* 源码解读



* 图解



* 步骤
  + 构建批处理运行环境
  + 使用 env.generateSequence 创建0-100的并行数据
  + 使用 fiter 过滤出来 大于8 的数字
  + 使用map操作传入 RichMapFunction ，将当前子任务的ID和数字构建成一个元组
  + 在RichMapFunction中可以使用 getRuntimeContext.getIndexOfThisSubtask 获取子任务序号
  + 打印测试
* 举例：
  + 在不使用rebalance的情况下，观察每一个线程执行的任务特点

|  |
| --- |
| /\*\*  \* flink的数据倾斜解决方案  \*/ public class RebalanceDemo {  public static void main(String[] args) throws Exception {  /\*\*  \* 构建批处理运行环境  \* 使用 env.generateSequence 创建0-100的并行数据  \* 使用 fiter 过滤出来 大于8 的数字  \* 使用map操作传入 RichMapFunction ，将当前子任务的ID和数字构建成一个元组  \* 在RichMapFunction中可以使用 getRuntimeContext.getIndexOfThisSubtask 获取子任务序号  \* 打印测试  \*/  //TODO 构建批处理运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  //TODO 使用 env.generateSequence 创建0-100的并行数据  DataStream<Long> dataSource = env.generateSequence(0, 100);   //TODO 使用 fiter 过滤出来 大于8 的数字  SingleOutputStreamOperator<Long> filteredDataSource = dataSource.filter(new FilterFunction<Long>() {  @Override  public boolean filter(Long aLong) throws Exception {  return aLong > 8;  }  });  //TODO 使用map操作传入 RichMapFunction ，将当前子任务的ID和数字构建成一个元组  //查看92条数据分别被哪些线程处理的，可以看到每个线程处理的数据条数  //spark中查看数据属于哪个分区使用哪个函数？mapPartitionWithIndex  //TODO 在RichMapFunction中可以使用 getRuntimeContext.getIndexOfThisSubtask 获取子任务序号  SingleOutputStreamOperator<Tuple2<Long, Integer>> tuple2MapOperator = filteredDataSource.map(new RichMapFunction<Long, Tuple2<Long, Integer>>() {  @Override  public Tuple2<Long, Integer> map(Long aLong) throws Exception {  return Tuple2.of(aLong, getRuntimeContext().getIndexOfThisSubtask());  }  });   //TODO 打印测试  tuple2MapOperator.print();   env.execute();  } } |

* + 使用rebalance

|  |
| --- |
| /\*\*  \* flink的数据倾斜解决方案  \*/ public class RebalanceDemo {  public static void main(String[] args) throws Exception {  /\*\*  \* 构建批处理运行环境  \* 使用 env.generateSequence 创建0-100的并行数据  \* 使用 fiter 过滤出来 大于8 的数字  \* 使用map操作传入 RichMapFunction ，将当前子任务的ID和数字构建成一个元组  \* 在RichMapFunction中可以使用 getRuntimeContext.getIndexOfThisSubtask 获取子任务序号  \* 打印测试  \*/  //TODO 构建批处理运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //TODO 使用 env.generateSequence 创建0-100的并行数据  DataStream<Long> dataSource = env.generateSequence(0, 100);   //TODO 使用 fiter 过滤出来 大于8 的数字  SingleOutputStreamOperator<Long> filteredDataSource = dataSource.filter(new FilterFunction<Long>() {  @Override  public boolean filter(Long aLong) throws Exception {  return aLong > 8;  }  });   //解决数据倾斜的问题  DataStream<Long> rebalance = filteredDataSource.rebalance();   //TODO 使用map操作传入 RichMapFunction ，将当前子任务的ID和数字构建成一个元组  //查看92条数据分别被哪些线程处理的，可以看到每个线程处理的数据条数  //spark中查看数据属于哪个分区使用哪个函数？mapPartitionWithIndex  //TODO 在RichMapFunction中可以使用 getRuntimeContext.getIndexOfThisSubtask 获取子任务序号  SingleOutputStreamOperator<Tuple2<Long, Integer>> tuple2MapOperator = rebalance.map(new RichMapFunction<Long, Tuple2<Long, Integer>>() {  @Override  public Tuple2<Long, Integer> map(Long aLong) throws Exception {  return Tuple2.of(aLong, getRuntimeContext().getIndexOfThisSubtask());  }  });   //TODO 打印测试  tuple2MapOperator.print();   env.execute();  } } |

#### Rescale Partitioner

RescalePartitioner,RESCALE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。

* 源码解读



这个看起来也太简单了，并且与RebalancePartitioner的逻辑是相同的？实际上并不是。

Flink 中的执行图可以分成四层：StreamGraph -> JobGraph -> ExecutionGraph -> 物理执行图。

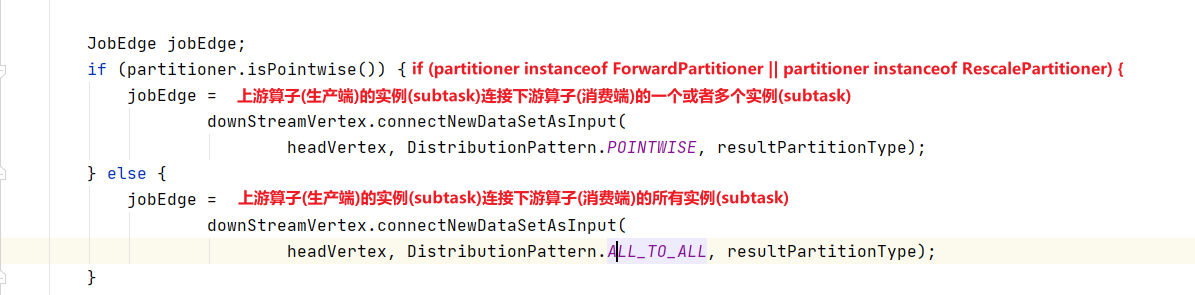
**StreamGraph**：是根据用户通过 Stream API 编写的代码生成的最初的图。用来表示程序的拓扑结构。

**JobGraph**：StreamGraph经过优化后生成了 JobGraph，提交给 JobManager 的数据结构。主要的优化为，将多个符合条件的节点 chain 在一起作为一个节点，这样可以减少数据在节点之间流动所需要的序列化/反序列化/传输消耗。

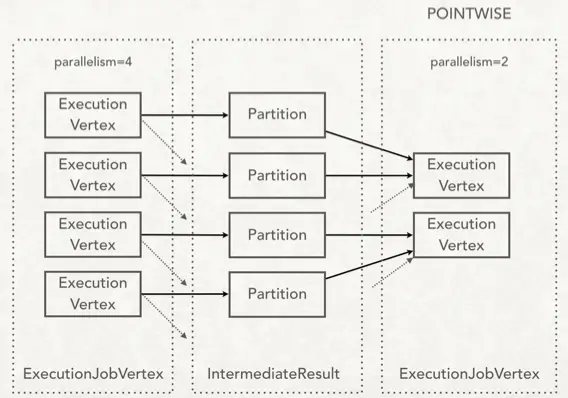
**ExecutionGraph**：JobManager 根据 JobGraph 生成ExecutionGraph。ExecutionGraph是JobGraph的并行化版本，是调度层最核心的数据结构。

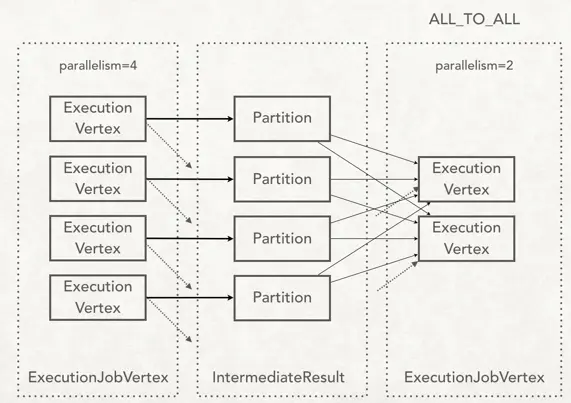
**物理执行图**：JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个TaskManager 上部署 Task 后形成的“图”，并不是一个具体的数据结构。

而StreamingJobGraphGenerator就是StreamGraph转换为JobGraph。在这个类中，把ForwardPartitioner和RescalePartitioner列为POINTWISE分配模式，其他的为ALL\_TO\_ALL分配模式。代码如下：

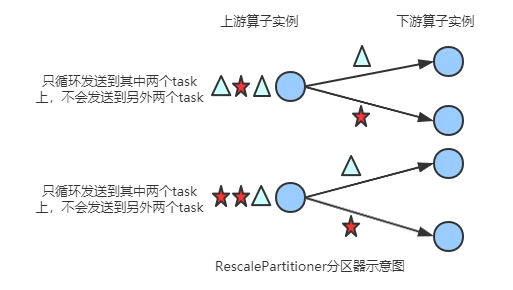


粗略地讲，如果分区逻辑是RescalePartitioner或ForwardPartitioner（**下面会说**），那么采用POINTWISE模式来连接上下游的顶点，对于其他分区逻辑，都用ALL\_TO\_ALL模式来连接。看下面两张图会比较容易理解。



  
也就是说，POINTWISE模式的RescalePartitioner在中间结果传送给下游节点时，会根据并行度的比值来轮询分配给下游算子实例的子集，对TaskManager来说本地性会比较好。而ALL\_TO\_ALL模式的RebalancePartitioner是真正的全局轮询分配，更加均衡，但是就会不可避免地在节点之间交换数据，如果数据量大的话，造成的网络流量会很可观。

* 图解



* 示例：

编写Flink程序，接收socket的单词数据，并将每个字符串广播到每个分区。

* 示例代码：

|  |
| --- |
| public class RescalePartitioningDemo {  public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new Configuration());   //Source是一个非并行的Source  //并行度是1  DataStreamSource<String> lines = env.socketTextStream("node01", 8888);   //并行度2  SingleOutputStreamOperator<String> mapped = lines.map(new RichMapFunction<String, String>() {  @Override  public String map(String value) throws Exception {  int indexOfThisSubtask = getRuntimeContext().getIndexOfThisSubtask();  return value + " : " + indexOfThisSubtask;  }  }).setParallelism(1);   //广播，上游的算子将一个数据广播到下游所以的subtask  DataStream<String> shuffled = mapped.rescale();   shuffled.addSink(new RichSinkFunction<String>() {  @Override  public void invoke(String value, Context context) throws Exception {  int index = getRuntimeContext().getIndexOfThisSubtask();  System.out.println(value + " -> " + index);  }  });   env.execute();  } } |

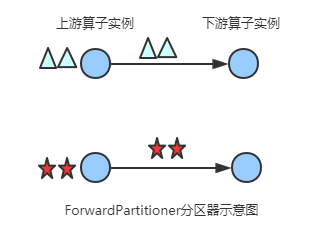
#### Forward Partitioner

发送到下游对应的第一个task，保证上下游算子并行度一致，即上有算子与下游算子是1:1的关系

* 源码解读



* 图解



在上下游的算子没有指定分区器的情况下，如果上下游的算子并行度一致，则使用**ForwardPartitioner**，否则使用**RebalancePartitioner**，对于ForwardPartitioner，必须保证上下游算子并行度一致，否则会抛出异常

* 示例：

编写Flink程序，接收socket的单词数据，并将每个字符串广播到每个分区。

* 示例代码：

|  |
| --- |
| public class ForwardPartitioningDemo {  public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new Configuration());   //Source是一个非并行的Source  //并行度是1  DataStreamSource<String> lines = env.socketTextStream("node01", 8888);   //并行度2  SingleOutputStreamOperator<String> mapped = lines.map(new RichMapFunction<String, String>() {  @Override  public String map(String value) throws Exception {  int indexOfThisSubtask = getRuntimeContext().getIndexOfThisSubtask();  return value + " : " + indexOfThisSubtask;  }  });   //广播，上游的算子将一个数据广播到下游所以的subtask  DataStream<String> shuffled = mapped.forward();   shuffled.addSink(new RichSinkFunction<String>() {  @Override  public void invoke(String value, Context context) throws Exception {  int index = getRuntimeContext().getIndexOfThisSubtask();  System.out.println(value + " -> " + index);  }  });   env.execute();  } } |

#### Custom Partitioning

使用用户定义的 Partitioner 为每个元素选择目标任务。

* 示例：

编写Flink程序，接收socket的单词数据，并将每个字符串写入到指定的分区中。

|  |
| --- |
| hadoop  spark  flink |

* 示例代码：

|  |
| --- |
| public class CustomPartitioningDemo {   public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new Configuration());   //Source是一个非并行的Source  //并行度是1  DataStreamSource<String> lines = env.socketTextStream("node01", 9999);   //并行度2  SingleOutputStreamOperator<Tuple2<String, Integer>> mapped = lines.map(new RichMapFunction<String, Tuple2<String, Integer>>() {  @Override  public Tuple2<String, Integer> map(String value) throws Exception {  int indexOfThisSubtask = getRuntimeContext().getIndexOfThisSubtask();  return Tuple2.of(value, indexOfThisSubtask);  }  });//.setParallelism(2);   //按照指定的规则进行分区  DataStream<Tuple2<String, Integer>> partitioned = mapped.partitionCustom(new Partitioner<String>() {  @Override  public int partition(String key, int numPartitions) {  //System.out.println("key: " + key + " ,下游task的并行度：" + numPartitions);  int res = 0;  if ("spark".equals(key)) {  res = 1;  } else if ("flink".equals(key)) {  res = 2;  } else if ("hadoop".equals(key)) {  res = 3;  }  return res;  }  }, tp -> tp.f0);   partitioned.addSink(new RichSinkFunction<Tuple2<String, Integer>>() {   @Override  public void invoke(Tuple2<String, Integer> value, Context context) throws Exception {  int index = getRuntimeContext().getIndexOfThisSubtask();  System.out.println(value.f0 + " , 上游 " + value.f1 + " -> 下游 " + index);  }  });   env.execute();  } } |

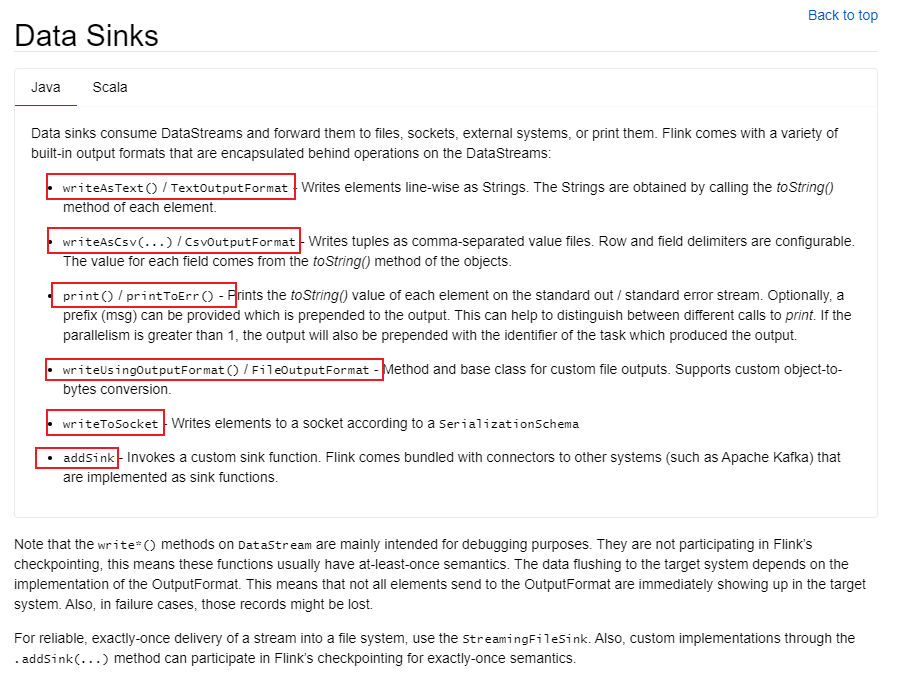
## 数据输出Data Sinks

经过一系列Transformation转换操作后，最后一定要调用**Sink**操作，才会形成一个完整的DataFlow拓扑。只有调用了Sink操作，才会产生最终的计算结果，这些数据可以写入到的文件、输出到指定的网络端口、消息中间件、外部的文件系统或者是打印到控制台。

flink在批处理中常见的sink

* print 打印
* writerAsText 以文本格式输出
* writeAsCsv 以csv格式输出
* writeUsingOutputFormat 以指定的格式输出
* writeToSocket 输出到网络端口
* 自定义连接器（addSink）

参考官网：[https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/datastream/overview/#data-sinks](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/datastream/overview/" \l "data-sinks)



### print 打印

打印是最简单的一个Sink，通常是用来做实验和测试时使用。如果想让一个DataStream输出打印的结果，直接可以在该DataStream调用print方法。另外，该方法还有一个重载的方法，可以传入一个字符，指定一个Sink的标识名称，如果有多个打印的Sink，用来区分到底是哪一个Sink的输出。

|  |
| --- |
| public class PrintSinkDemo {  public static void main(String[] args) throws Exception {  //local模式默认的并行度是当前机器的逻辑核的数量  Configuration configuration = new Configuration();  StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(configuration);   int parallelism0 = env.getParallelism();  System.out.println("执行环境默认的并行度：" + parallelism0);  DataStreamSource<String> lines = env.socketTextStream("node01", 8888);  //获取DataStream的并行度  int parallelism = lines.getParallelism();  System.out.println("SocketSource的并行度：" + parallelism);  lines.print();  lines.addSink(new MyPrintSink()).name("my-print-sink");  env.execute();  }   public static class MyPrintSink extends RichSinkFunction<String> {  private int indexOfThisSubtask;  @Override  public void open(Configuration parameters) throws Exception {  RuntimeContext runtimeContext = getRuntimeContext();  indexOfThisSubtask = runtimeContext.getIndexOfThisSubtask();  }   @Override  public void invoke(String value, Context context) throws Exception {   System.out.println(indexOfThisSubtask + 1 + "> " + value);  }  } } |

下面的结果是WordCount例子中调用print Sink输出在控制台的结果，细心的读者会发现，在输出的单词和次数之前，有一个数字前缀，我这里是1~4，这个数字是该Sink所在subtask的Index + 1。有的读者运行的结果数字前缀是1~8，该数字前缀其实是与任务的并行度相关的，由于该任务是以local模式运行，默认的并行度是所在机器可用的逻辑核数即线程数，我的电脑是2核4线程的，所以subtask的Index范围是0~3，将Index + 1，显示的数字前缀就是1~4了。这里在来仔细的观察一下运行的结果发现：相同的单词输出结果的数字前缀一定相同，即经过keyBy之后，相同的单词会被shuffle到同一个subtask中，并且在同一个subtask的同一个组内进行聚合。一个subtask中是可能有零到多个组的，如果是有多个组，每一个组是相互独立的，累加的结果不会相互干扰。

### writerAsText 以文本格式输出

该方法是将数据以文本格式实时的写入到指定的目录中，本质上使用的是TextOutputFormat格式写入的。每输出一个元素，在该内容后面同时追加一个换行符，最终以字符的形式写入到文件中，目录中的文件名称是该Sink所在subtask的Index + 1。该方法还有一个重载的方法，可以额外指定一个枚举类型的参数writeMode，默认是WriteMode.NO\_OVERWRITE，如果指定相同输出目录下有相同的名称文件存在，就会出现异常。如果是WriteMode.OVERWRITE，会将以前的文件覆盖。

|  |
| --- |
| public class WriteSinkDemo {  public static void main(String[] args) throws Exception {  //local模式默认的并行度是当前机器的逻辑核的数量  Configuration configuration = new Configuration();  StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(configuration);  int parallelism0 = env.getParallelism();  System.out.println("执行环境默认的并行度：" + parallelism0);  DataStreamSource<String> lines = env.socketTextStream("node01", 8888);  //获取DataStream的并行度  int parallelism = lines.getParallelism();  System.out.println("SocketSource的并行度：" + parallelism);  lines.writeAsText("./data/output/3", FileSystem.WriteMode.OVERWRITE).setParallelism(2);  env.execute();  } } |

### writeAsCsv 以csv格式输出

该方法是将数据以csv格式写入到指定的目录中，本质上使用的是CsvOutputFormat格式写入的。每输出一个元素，在该内容后面同时追加一个换行符，最终以csv的形式(类似Excel的格式，字段和字段之间用逗号分隔)写入到文件中，目录中的文件名称是该Sink所在subtask的Index + 1。需要说明的是，该Sink并不是将数据实时的写入到文件中，而是有一个BufferedOutputStream，默认缓存的大小为4096个字节，只有达到这个大小，才会flush到磁盘。另外程序在正常退出，调用Sink的close方法也会flush到磁盘。

|  |
| --- |
| /\*\*  \* @Description 将DataSet数据写入到csv文件中  \*/ public class CsvSink {  public static void main(String[] args) throws Exception {  ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();  //需先建立文件  String filePath = "./data/output/user.csv";  //添加数据  Tuple7<Integer, String, Integer, Integer, String, String, Long> row = new Tuple7<>(15, "zhangsan", 40, 1, "CN", "2020-09-08 00:00:00", 1599494400000L);  //转换为dataSet  DataSet<Tuple7<Integer, String, Integer, Integer, String, String, Long>> dataSet = env.fromElements(row);  //将内容写入到File中，如果文件已存在，将会被复盖  dataSet.writeAsCsv(filePath, FileSystem.WriteMode.OVERWRITE).setParallelism(1);  env.execute("fline file sink");  } } |

### writeUsingOutputFormat 以指定的格式输出

该方法是将数据已指定的格式写入到指定目录中，该方法要传入一个OutputFormat接口的实现类，该接口有很多已经实现好了的实现类，并且可以根据需求自己实现，所以该方法更加灵活。writeAsText和writeAsCsv方法底层都是调用了writeUsingOutputFormat方法。

|  |
| --- |
| public class writeUsingOutputFormatSink {  public static void main(String[] args) throws Exception {  //1：获取流处理运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  //调用env的fromElements创建一个非并行的DataStreamSource  DataStreamSource<String> words = env.fromElements(  "hadoop","spark","flink","hbase","flink","spark"  );  //4：对拆分后的单词，每个单词记一次数  SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndOne = words.map(new MapFunction<String, Tuple2<String, Integer>>() {  @Override  public Tuple2<String, Integer> map(String word) throws Exception {  return Tuple2.of(word, 1);  }  });  DataStream<Tuple2<String, Integer>> result = wordAndOne.keyBy(0).sum(1);  result.writeUsingOutputFormat(new TextOutputFormat<>(new Path("./data/output/5")));  env.execute();  } } |

### writeToSocket 输出到网络端口

该方法是将数据输出到指定的Socket网络地址端口。该方法需要传入三个参数：第一个为ip地址或主机名，第二个为端口号，第三个为数据输出的序列化格式SerializationSchema。输出之前，指定的网络端口服务必须已经启动。

|  |
| --- |
| public class WriteToSocketDemo {  public static void main(String[] args) throws Exception {  //local模式默认的并行度是当前机器的逻辑核的数量  Configuration configuration = new Configuration();  StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(configuration);  int parallelism0 = env.getParallelism();  System.out.println("执行环境默认的并行度：" + parallelism0);  DataStreamSource<String> lines = env.socketTextStream("node01", 8888);  //获取DataStream的并行度  int parallelism = lines.getParallelism();  System.out.println("SocketSource的并行度：" + parallelism);  //lines.writeAsText("file:///Users/xing/Desktop/out");  lines.writeToSocket("localhost", 9999, new SimpleStringSchema());  env.execute();  } } |

### 基于本地集合的sink

* 示例:

|  |
| --- |
| 基于下列数据,分别 进行打印输出,error输出 |
| (19, "zhangsan", 178.8),  (17, "lisi", 168.8),  (18, "wangwu", 184.8),  (21, "zhaoliu", 164.8) |

* 目标:

数据可以输出到：Stdout，Stderr，采集为本地集合

* 代码:

|  |
| --- |
| /\*\*  \* 基于本地集合的SINK  \*/ public class CollectionDemo {  public static void main(String[] args) throws Exception {  //1.env  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //2.source  //DataStream<String> ds = env.fromElements("hadoop", "flink");  DataStream<Tuple2<Integer, String>> dataSource = env.fromElements(  Tuple2.of(1, "zhangsan"),  Tuple2.of(2, "lisi"),  Tuple2.of(3, "wangwu"),  Tuple2.of(4, "zhaoliu")  );   //3.transformation  //4.sink  dataSource.print();  dataSource.printToErr();  //注意:  //Parallelism=1为文件  //Parallelism>1为文件夹   //5.execute  env.execute();  } } |

## Connectors

<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/datastream/jdbc/>

### JDBC Connector

该连接器可以向 JDBC 数据库写入数据。

#### 依赖

添加下面的依赖以便使用该连接器（同时添加 JDBC 驱动）：

|  |
| --- |
| <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-jdbc\_2.11</artifactId>  <version>1.13.1</version> </dependency> |

#### 案例演示

* 需求

从指定的socket读取数据，对单词进行计算，最后将结果写入到MySQL

* Mysql创建表

|  |
| --- |
| CREATE TABLE `t\_wordcount` (  `word` varchar(255) NOT NULL,  `counts` int(11) DEFAULT '0',  PRIMARY KEY (`word`) ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

* 参考代码

|  |
| --- |
| /\*\*  \* 从指定的socket读取数据，对单词进行计算，最后将结果写入到MySQL  \*/ public class JDBCSinkDemo {  public static void main(String[] args) throws Exception {  //创建Flink流计算执行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.enableCheckpointing(5000);  //创建DataStream  //Source  DataStreamSource<String> lines = env.socketTextStream("node01", 9999);   //调用Transformation  SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndOne = lines.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {  @Override  public void flatMap(String line, Collector<Tuple2<String, Integer>> collector) throws Exception {  String[] words = line.split(" ");  for (String word : words) {  //new Tuple2<String, Integer>(word, 1)  collector.collect(Tuple2.of(word, 1));  }  }  });   //分组  KeyedStream<Tuple2<String, Integer>, String> keyed = wordAndOne.keyBy(new KeySelector<Tuple2<String, Integer>, String>() {  @Override  public String getKey(Tuple2<String, Integer> tp) throws Exception {  return tp.f0;  }  });   //聚合  SingleOutputStreamOperator<Tuple2<String, Integer>> summed = keyed.sum(1);   summed.addSink(JdbcSink.sink(  "INSERT INTO t\_wordcount (word, counts) VALUES (?, ?) ON DUPLICATE KEY UPDATE counts = ?",  (ps, t) -> {  ps.setString(1, t.f0);  ps.setInt(2, t.f1);  ps.setInt(3, t.f1);  },  new JdbcConnectionOptions.JdbcConnectionOptionsBuilder()  .withUrl("jdbc:mysql://node03:3306/test?characterEncoding=utf-8")  .withDriverName("com.mysql.jdbc.Driver")  .withUsername("root")  .withPassword("123456")  .build()));   //启动执行  env.execute("JDBCSinkDemo");  } } |

### Apache Kafka Connector

在现实生产环境中，为了保证Flink可以高效地读取数据源中的数据，通常是跟一些分布式消息中件结合使用，例如Apache Kafka。Kafka的特点是分布式、多副本、高可用、高吞吐、可以记录偏移量等。Flink和Kafka整合可以高效的读取数据，并且可以保证Exactly Once（精确一次性语义）

#### 依赖

Apache Flink 集成了通用的 Kafka 连接器，它会尽力与 Kafka client 的最新版本保持同步。该连接器使用的 Kafka client 版本可能会在 Flink 版本之间发生变化。 当前 Kafka client 向后兼容 0.10.0 或更高版本的 Kafka broker。 有关 Kafka 兼容性的更多细节，请参考[Kafka 官方文档](https://kafka.apache.org/protocol.html" \l "protocol_compatibility)。

|  |
| --- |
| <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-kafka\_2.11</artifactId>  <version>1.13.1</version> </dependency> |

#### Kafka Consumer

Flink 的 Kafka consumer 称为 FlinkKafkaConsumer。它提供对一个或多个 Kafka topics 的访问。

构造函数接受以下参数：

* Topic 名称或者名称列表
* 用于反序列化 Kafka 数据的 DeserializationSchema 或者 KafkaDeserializationSchema
* Kafka 消费者的属性。需要以下属性：
  + “bootstrap.servers”（以逗号分隔的 Kafka broker 列表）
  + “group.id” 消费组 ID



##### DeserializationSchema

Flink Kafka Consumer 需要知道如何将 Kafka 中的二进制数据转换为 **Java** 或者 **Scala** 对象。KafkaDeserializationSchema 允许用户指定这样的 schema，每条 Kafka 中的消息会调用 T deserialize(ConsumerRecord<byte[], byte[]> record) 反序列化。

为了方便使用，Flink 提供了以下几种 schemas：

* SimpleStringSchema：按照字符串方式序列化、反序列化
* TypeInformationSerializationSchema（和 TypeInformationKeyValueSerializationSchema) 基于 Flink 的 TypeInformation 创建 schema。 如果该数据的读和写都发生在 Flink 中，那么这将是非常有用的。**此 schema 是其他通用序列化方法的高性能 Flink 替代方案**。
* JsonDeserializationSchema（和 JSONKeyValueDeserializationSchema）将序列化的 JSON 转化为 ObjectNode 对象，可以使用 objectNode.get("field").as(Int/String/...)() 来访问某个字段。 KeyValue objectNode 包含一个含所有字段的 key 和 values 字段，以及一个可选的"metadata"字段，可以访问到消息的 offset、partition、topic 等信息。
* AvroDeserializationSchema 使用静态提供的 schema 读取 Avro 格式的序列化数据。 它能够从 Avro 生成的类（AvroDeserializationSchema.forSpecific(...)）中推断出 schema，或者可以与 GenericRecords 一起使用手动提供的 schema（用 AvroDeserializationSchema.forGeneric(...)）。此反序列化 schema 要求序列化记录不能包含嵌入式架构！

要使用此反序列化 schema 必须添加以下依赖：

|  |
| --- |
| <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-avro</artifactId>  <version>{{site.version }}</version> </dependency> |

##### 配置 Kafka Consumer 开始消费的位置

Flink Kafka Consumer 允许通过配置来确定 Kafka 分区的起始位置。

|  |
| --- |
| final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  FlinkKafkaConsumer<String> myConsumer = new FlinkKafkaConsumer<>(...); myConsumer.setStartFromEarliest(); // 尽可能从最早的记录开始 myConsumer.setStartFromLatest(); // 从最新的记录开始 myConsumer.setStartFromTimestamp(...); // 从指定的时间开始（毫秒） myConsumer.setStartFromGroupOffsets(); // 默认的方法  DataStream<String> stream = env.addSource(myConsumer); |

Flink Kafka Consumer 的所有版本都具有上述明确的起始位置配置方法。

* setStartFromGroupOffsets（**默认方法**）：从 Kafka brokers 中的 consumer 组（consumer 属性中的 group.id 设置）提交的偏移量中开始读取分区。 如果找不到分区的偏移量，那么将会使用配置中的 **auto.offset.reset** 设置。
* setStartFromEarliest() 或者 setStartFromLatest()：从最早或者最新的记录开始消费，在这些模式下，Kafka 中的 committed offset 将被忽略，不会用作起始位置。
* setStartFromTimestamp(long)：从指定的时间戳开始。对于每个分区，其时间戳大于或等于指定时间戳的记录将用作起始位置。如果一个分区的最新记录早于指定的时间戳，则只从最新记录读取该分区数据。在这种模式下，Kafka 中的已提交 offset 将被忽略，不会用作起始位置。
* setStartFromSpecificOffsets：从指定的分区的offset位置开始读取，如指定的offsets中不存在某个分区，该分区从group offset位置开始读取

例子如下：为每个分区指定 consumer 应该开始消费的具体 offset：

|  |
| --- |
| Map<KafkaTopicPartition, Long> specificStartOffsets = new HashMap<>(); specificStartOffsets.put(new KafkaTopicPartition("myTopic", 0), 23L); specificStartOffsets.put(new KafkaTopicPartition("myTopic", 1), 31L); specificStartOffsets.put(new KafkaTopicPartition("myTopic", 2), 43L);  myConsumer.setStartFromSpecificOffsets(specificStartOffsets); |

上面的例子中使用的配置是指定从 myTopic 主题的 0 、1 和 2 分区的指定偏移量开始消费。offset 值是 consumer 应该为每个分区读取的下一条消息。

请注意：如果 consumer 需要读取在提供的 offset 映射中没有指定 offset 的分区，那么它将回退到该特定分区的默认组偏移行为（即 setStartFromGroupOffsets()）。

**请注意**：当 Job 从故障中自动恢复或使用 savepoint 手动恢复时，这些起始位置配置方法不会影响消费的起始位置。在恢复时，每个 Kafka 分区的起始位置由存储在 savepoint 或 checkpoint 中的 offset 确定（有关 checkpointing 的信息，请参阅下一节，以便为 consumer 启用容错功能）。

##### Kafka Consumer 和容错

伴随着启用 Flink 的 checkpointing 后，Flink Kafka Consumer 将使用 topic 中的记录，并以一致的方式定期检查其所有 Kafka offset 和其他算子的状态。如果 Job 失败，Flink 会将流式程序恢复到最新 checkpoint 的状态，并从存储在 checkpoint 中的 offset 开始重新消费 Kafka 中的消息。

因此，设置 checkpoint 的间隔定义了程序在发生故障时最多需要返回多少。

为了使 Kafka Consumer 支持容错，需要在 [执行环境](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/deployment/config/" \l "execution-checkpointing-interval) 中启用拓扑的 checkpointing。

如果未启用 checkpoint，那么 Kafka consumer 将定期向 Zookeeper 提交 offset。

##### Kafka Consumer Topic 和分区发现

###### 分区发现

Flink Kafka Consumer 支持发现动态创建的 Kafka 分区，并使用精准一次的语义保证去消耗它们。在初始检索分区元数据之后（即，当 Job 开始运行时）发现的所有分区将从最早可能的 offset 中消费。

默认情况下，是禁用了分区发现的。若要启用它，请在提供的属性配置中为 **flink.partition-discovery.interval-millis**设置大于 0 的值，表示发现分区的间隔是以毫秒为单位的。

###### Topic 发现

在更高的级别上，Flink Kafka Consumer 还能够使用正则表达式基于 Topic 名称的模式匹配来发现 Topic。请看下面的例子：

|  |
| --- |
| final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  Properties properties = new Properties(); properties.setProperty("bootstrap.servers", "localhost:9092"); properties.setProperty("group.id", "test");  FlinkKafkaConsumer<String> myConsumer = new FlinkKafkaConsumer<>(  java.util.regex.Pattern.compile("test-topic-[0-9]"),  new SimpleStringSchema(), properties);  DataStream<String> stream = env.addSource(myConsumer); ... |

在上面的例子中，当 Job 开始运行时，Consumer 将订阅名称与指定正则表达式匹配的所有主题（以 test-topic 开头并以单个数字结尾）。

要允许 consumer 在作业开始运行后发现动态创建的主题，那么请为 **flink.partition-discovery.interval-millis 设置非负值**。这允许 consumer 发现名称与指定模式匹配的新主题的分区。

**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。

##### Kafka Consumer 提交 Offset 的行为配置

Flink Kafka Consumer 允许有配置如何将 offset 提交回 Kafka broker 的行为。请注意：Flink Kafka Consumer 不依赖于提交的 offset 来实现容错保证。提交的 offset 只是一种方法，用于公开 consumer 的进度以便进行监控。

配置 offset 提交行为的方法是否相同，取决于是否为 job 启用了 checkpointing。

* 禁用 Checkpointing： 如果禁用了 checkpointing，则 Flink Kafka Consumer 依赖于内部使用的 Kafka client 自动定期 offset 提交功能。 因此，要禁用或启用 offset 的提交，只需将 **enable.auto.commit**或者 **auto.commit.interval.ms** 的Key 值设置为提供的 Properties 配置中的适当值。
* 启用 Checkpointing： 如果启用了 checkpointing，那么当 checkpointing 完成时，Flink Kafka Consumer 将提交的 offset 存储在 checkpoint 状态中。 这确保 Kafka broker 中提交的 offset 与 checkpoint 状态中的 offset 一致。 用户可以通过调用 consumer 上的 setCommitOffsetsOnCheckpoints(boolean) 方法来禁用或启用 offset 的提交(默认情况下，这个值是 true )。 注意，在这个场景中，Properties 中的自动定期 offset 提交设置会被完全忽略。

#### Kafka Producer

Flink Kafka Producer 被称为 FlinkKafkaProducer。它允许将消息流写入一个或多个 Kafka topic。

构造器接收下列参数：

* 事件被写入的默认输出 topic
* 序列化数据写入 Kafka 的 SerializationSchema / KafkaSerializationSchema
* Kafka client 的 Properties。下列 property 是必须的：
  + “bootstrap.servers” （逗号分隔 Kafka broker 列表）
* 容错语义

|  |
| --- |
| DataStream<String> stream = ...;  Properties properties = new Properties(); properties.setProperty("bootstrap.servers", "localhost:9092");  FlinkKafkaProducer<String> myProducer = new FlinkKafkaProducer<String>(  "my-topic", // 目标 topic  new SimpleStringSchema() // 序列化 schema  properties, // producer 配置  FlinkKafkaProducer.Semantic.EXACTLY\_ONCE); // 容错  stream.addSink(myProducer); |

#### SerializationSchema

Flink Kafka Producer 需要知道如何将 Java/Scala 对象转化为二进制数据。

KafkaSerializationSchema 允许用户指定这样的 schema。它会为每个记录调用 ProducerRecord<byte[], byte[]> serialize(T element, @Nullable Long timestamp) 方法，产生一个写入到 Kafka 的 ProducerRecord。

用户可以对如何将数据写到 Kafka 进行细粒度的控制。你可以通过 producer record：

* 设置 header 值
* 为每个 record 定义 key
* 指定数据的自定义分区

#### Kafka Producer 和容错

启用 Flink 的 checkpointing 后，FlinkKafkaProducer 可以提供精确一次的语义保证。

除了启用 Flink 的 checkpointing，你也可以通过将适当的 semantic 参数传递给 FlinkKafkaProducer 来选择三种不同的操作模式：

* Semantic.NONE：Flink 不会有任何语义的保证，产生的记录可能会丢失或重复。
* Semantic.AT\_LEAST\_ONCE（默认设置）：可以保证不会丢失任何记录（但是记录可能会重复）
* Semantic.EXACTLY\_ONCE：使用 Kafka 事务提供精确一次语义。无论何时，在使用事务写入 Kafka 时，都要记得为所有消费 Kafka 消息的应用程序设置所需的 isolation.level（read\_committed 或 read\_uncommitted - 后者是默认值）。

**注意事项**

Semantic.EXACTLY\_ONCE 模式依赖于事务提交的能力。事务提交发生于触发 checkpoint 之前，以及从 checkpoint 恢复之后。如果从 Flink 应用程序崩溃到完全重启的时间超过了 Kafka 的事务超时时间，那么将会有数据丢失（Kafka 会自动丢弃超出超时时间的事务）。考虑到这一点，请根据预期的宕机时间来合理地配置事务超时时间。

默认情况下，Kafka broker 将 transaction.max.timeout.ms 设置为 15 分钟。此属性不允许为大于其值的 producer 设置事务超时时间。 默认情况下，FlinkKafkaProducer 将 producer config 中的 transaction.timeout.ms 属性设置为 1 小时，因此在使用 Semantic.EXACTLY\_ONCE 模式之前应该增加 transaction.max.timeout.ms 的值。

在 KafkaConsumer 的 read\_committed 模式中，任何未结束（既未中止也未完成）的事务将阻塞来自给定 Kafka topic 的未结束事务之后的所有读取数据。 换句话说，在遵循如下一系列事件之后：

* 用户启动了 transaction1 并使用它写了一些记录
* 用户启动了 transaction2 并使用它编写了一些其他记录
* 用户提交了 transaction2

即使 transaction2 中的记录已提交，在提交或中止 transaction1 之前，消费者也不会看到这些记录。这有 2 层含义：

* 首先，在 Flink 应用程序的正常工作期间，用户可以预料 Kafka 主题中生成的记录的可见性会延迟，相当于已完成 checkpoint 之间的平均时间。
* 其次，在 Flink 应用程序失败的情况下，此应用程序正在写入的供消费者读取的主题将被阻塞，直到应用程序重新启动或配置的事务超时时间过去后，才恢复正常。此标注仅适用于有多个 agent 或者应用程序写入同一 Kafka 主题的情况。

**注意：**Semantic.EXACTLY\_ONCE 模式为每个 FlinkKafkaProducer 实例使用固定大小的 KafkaProducer 池。每个 checkpoint 使用其中一个 producer。如果并发 checkpoint 的数量超过池的大小，FlinkKafkaProducer 将抛出异常，并导致整个应用程序失败。请合理地配置最大池大小和最大并发 checkpoint 数量。

**注意：**Semantic.EXACTLY\_ONCE 会尽一切可能不留下任何逗留的事务，否则会阻塞其他消费者从这个 Kafka topic 中读取数据。但是，如果 Flink 应用程序在第一次 checkpoint 之前就失败了，那么在重新启动此类应用程序后，系统中不会有先前池大小（pool size）相关的信息。因此，在第一次 checkpoint 完成前对 Flink 应用程序进行缩容，且并发数缩容倍数大于安全系数 FlinkKafkaProducer.SAFE\_SCALE\_DOWN\_FACTOR 的值的话，是不安全的。

#### 代码实现-Kafka Consumer

|  |
| --- |
| /\*\*  \* 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", "node01:9092");  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

|  |
| --- |
| /\*\*   \* 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

该方法是将数据输出到Redis数据库中，Redis是一个基于内存、性能极高的NoSQL数据库，数据还可以持久化到磁盘，读写速度快，适合存储key-value类型的数据。Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。Flink实时计算出的结果，需要快速的输出存储起来，要求写入的存储系统的速度要快，这个才不会造成数据积压。Redis就是一个非常不错的选择。

<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

从指定的socket读取数据，对单词进行计算，将结果写入到Redis中

* 代码实现

|  |
| --- |
| /\*\*  \* 从指定的socket读取数据，对单词进行计算，将结果写入到Redis中  \*/ public class RedisSinkDemo {  public static void main(String[] args) throws Exception {  //创建Flink流计算执行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  //创建DataStream  //Source  DataStreamSource<String> lines = env.socketTextStream("node01", 9999);  //调用Transformation开始  SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndOne = lines.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {  @Override  public void flatMap(String line, Collector<Tuple2<String, Integer>> collector) throws Exception {  String[] words = line.split(" ");  for (String word : words) {  //new Tuple2<String, Integer>(word, 1)  collector.collect(Tuple2.of(word, 1));  }  }  });   //分组  KeyedStream<Tuple2<String, Integer>, String> keyed = wordAndOne.keyBy(new KeySelector<Tuple2<String, Integer>, String>() {  @Override  public String getKey(Tuple2<String, Integer> tp) throws Exception {  return tp.f0;  }  });   //聚合  SingleOutputStreamOperator<Tuple2<String, Integer>> summed = keyed.sum(1);   //Transformation结束   //调用Sink  //summed.addSink()  FlinkJedisPoolConfig conf = new FlinkJedisPoolConfig.Builder().setHost("node03").setPassword("123456").setDatabase(8).build();   summed.addSink(new RedisSink<Tuple2<String, Integer>>(conf, new RedisWordCountMapper()));  //启动执行  env.execute("StreamingWordCount");   }   public static class RedisWordCountMapper implements RedisMapper<Tuple2<String, Integer>> {   @Override  public RedisCommandDescription getCommandDescription() {  return new RedisCommandDescription(RedisCommand.HSET, "WORD\_COUNT");  }   @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

### Flink的广播变量

Flink支持广播变量，就是将数据广播到具体的taskmanager上，数据存储在**内存**中，这样可以减缓大量的**shuffle**操作；

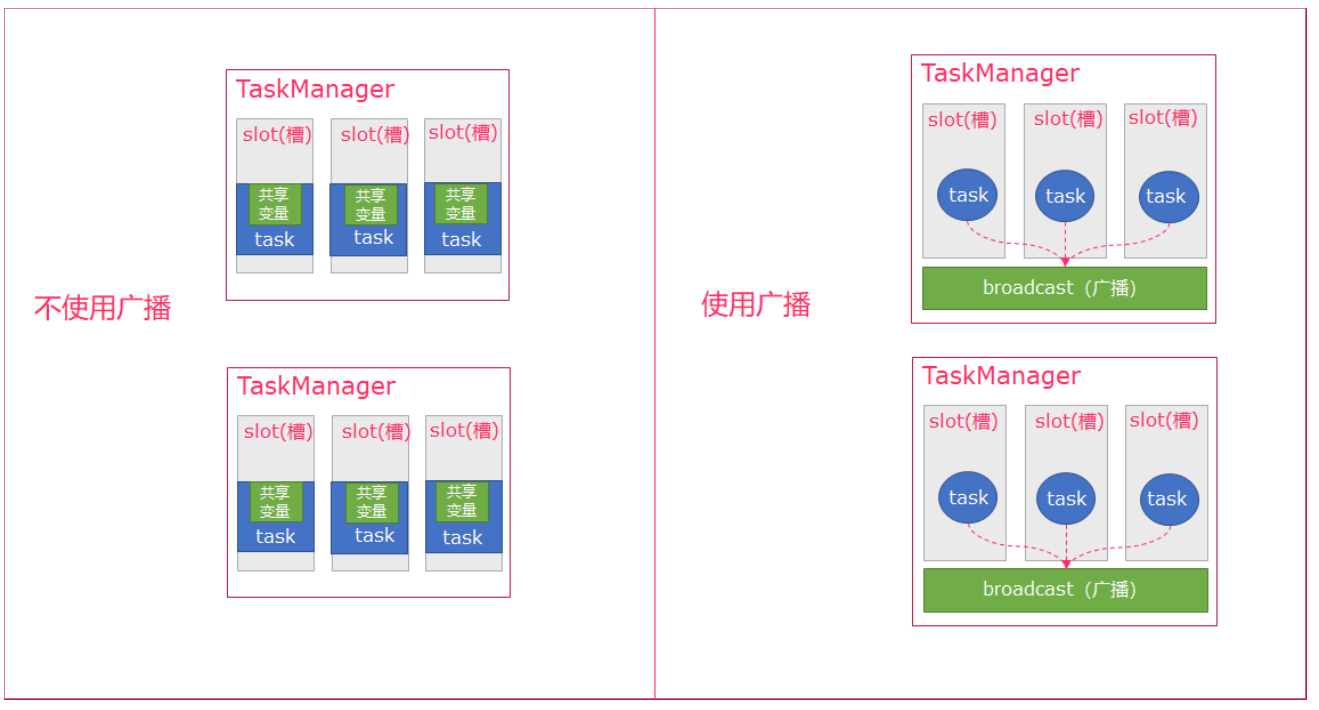
比如在数据join阶段，不可避免的就是大量的shuffle操作，我们可以把其中一个dataSet广播出去，一直加载到taskManager的内存中，可以直接在内存中拿数据，避免了大量的shuffle，导致集群性能下降；

广播变量创建后，它可以运行在集群中的任何**function**上，而不需要多次传递给集群节点。另外需要记住，不应该修改广播变量，这样才能确保每个节点获取到的值都是一致的。

一句话解释，可以理解为是一个公共的共享变量，我们可以把一个dataset 数据集广播出去，然后不同的task在节点上都能够获取到，这个数据在每个节点上只会存在一份。如果不使用broadcast，则在每个节点中的每个task中都需要拷贝一份dataset数据集，比较浪费内存(也就是一个节点中可能会存在多份dataset数据)。

**注意：**因为广播变量是要把dataset广播到**TaskManager**节点内存中，所以广播的数据量不能太大，否则会出现OOM这样的问题

* Broadcast：Broadcast是通过withBroadcastSet(dataset，string)来注册的
* Access：通过getRuntimeContext().getBroadcastVariable(String)访问广播变量



* + 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
* 用法

在需要使用广播的操作后，使用 **withBroadcastSet** 创建广播

在操作中，使用getRuntimeContext.getBroadcastVariable [广播数据类型] ( 广播名 )获取广播变量

* 示例

创建一个 学生 数据集，包含以下数据

|  |
| --- |
| |学生ID | 姓名 |  |------|------|  List((1, "张三"), (2, "李四"), (3, "王五")) |

将该数据，发布到广播。

再创建一个 成绩 数据集，

|  |
| --- |
| |学生ID | 学科 | 成绩 |  |------|------|-----|  List( (1, "语文", 50),(2, "数学", 70), (3, "英文", 86)) |

请通过广播获取到学生姓名，将数据转换为

|  |
| --- |
| List( ("张三", "语文", 50),("李四", "数学", 70), ("王五", "英文", 86)) |

* 步骤
  + 获取批处理运行环境
  + 分别创建两个数据集
  + 使用 RichMapFunction 对 成绩 数据集进行map转换
  + 在数据集调用 map 方法后，调用 withBroadcastSet 将 学生 数据集创建广播
  + 实现 RichMapFunction
    - 将成绩数据(学生ID，学科，成绩) -> (学生姓名，学科，成绩)
    - 重写 open 方法中，获取广播数据
    - 在 map 方法中使用广播进行转换
  + 打印测试
* 参考代码

|  |
| --- |
| /\*\*  \* 演示广播变量  \*/ public class BoardcastDemo {  public static void main(String[] args) throws Exception {  // 获取env  ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();  // 准备学生信息数据集  DataSource<Tuple2<Integer, String>> studentInfoDataSet = env.fromElements(  Tuple2.of(1, "王大锤"),  Tuple2.of(2, "潇潇"),  Tuple2.of(3, "甜甜")  );  // 准备分数信息数据集  DataSource<Tuple3<Integer, String, Integer>> scoreInfoDataSet = env.fromElements(  Tuple3.of(1, "数据结构", 99),  Tuple3.of(2, "英语", 100),  Tuple3.of(3, "C++", 96),  Tuple3.of(5, "Java", 97),  Tuple3.of(3, "Scala", 100)  );   /\*  广播变量的使用分为两步  1. 设置它, 需要在执行具体算子的后面链式调用withBroadcastSet方法  2. 得到它, 在算子内部getRuntimeContext().getBroadcastVariable(广播变量名)来获取  \*/   // 使用map方法进行转换，在map内部获取广播变量,在map方法的后面链式调用withBroadcastSet方法去设置广播变量  MapOperator<Tuple3<Integer, String, Integer>, Tuple3<String, String, Integer>> result = scoreInfoDataSet.map(  new RichMapFunction<Tuple3<Integer, String, Integer>, Tuple3<String, String, Integer>>() {  // 定义一个map用来存储从广播变量中取得的学生信息  Map<Integer, String> map = new HashMap<Integer, String>();   /\*  open 方法在实例化的开始, 会执行一次  \*/  @Override  public void open(Configuration parameters) throws Exception {  // 在open方法内将广播变量中的学生信息写入到map中  List<Tuple2<Integer, String>> broadcastVariable = getRuntimeContext().getBroadcastVariable("student");  for (Tuple2<Integer, String> stu : broadcastVariable) {  this.map.put(stu.f0, stu.f1);  }  }   @Override  public Tuple3<String, String, Integer> map(Tuple3<Integer, String, Integer> value) throws Exception {  int stuId = value.f0;  String stuName = this.map.getOrDefault(stuId, "未知学生姓名");  return Tuple3.of(stuName, value.f1, value.f2);  }  }  ).withBroadcastSet(studentInfoDataSet, "student");   result.print();  } } |

### Flink Accumulators & Counters

**Accumulator**即累加器，与Mapreduce counter的应用场景差不多，都能很好地观察task在运行期间的数据变化

可以在Flink job任务中的算子函数中操作累加器，但是只能在任务执行结束之后才能获得累加器的最终结果。

Flink现在有以下内置累加器。每个累加器都实现了Accumulator接口。

* IntCounter
* LongCounter
* DoubleCounter
* 步骤

|  |
| --- |
| 1. 创建累加器 |
| private IntCounter numLines = new IntCounter(); |
| 1. 注册累加器 |
| getRuntimeContext().addAccumulator("num-lines", this.numLines); |
| 1. 使用累加器 |
| this.numLines.add(1); |
| 1. 获取累加器的结果 |
| myJobExecutionResult.getAccumulatorResult("num-lines") |

* 参考代码
  + 未使用累加器：

|  |
| --- |
| /\*\*  \* 演示未使用累加器  \*/ public class AccumulatorsDemo1 {  public static void main(String[] args) throws Exception {  ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();   DataSource<Long> source = env.generateSequence(1, 10);   MapOperator<Long, Long> map = source.map(new RichMapFunction<Long, Long>() {  int counter = 0;   @Override  public Long map(Long value) throws Exception {  counter += 1;  System.out.println("Thread id: " + getRuntimeContext().getIndexOfThisSubtask() + ",counter: " + counter);  return value;  }  });   map.writeAsText("data/output/accumulators", FileSystem.WriteMode.OVERWRITE);  env.execute();  } } |

* + 使用累加器:

|  |
| --- |
| /\*\*  \* 演示使用累加器  \*/ public class AccumulatorsDemo2 {  public static void main(String[] args) throws Exception {  ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();   DataSource<Long> source = env.generateSequence(1, 10);   /\*  累加器使用为4步：  1. 创建累加器  2. 注册累加器  3. 使用累加器  4. 获得累加器的结果  \*/  MapOperator<Long, Long> map = source.map(new RichMapFunction<Long, Long>() {  // 1. 创建累加器  IntCounter counter = new IntCounter();   // 2. 在open方法中注册累加器  @Override  public void open(Configuration parameters) throws Exception {  super.open(parameters);  // 注册  getRuntimeContext().addAccumulator("sum", counter);  }   @Override  public Long map(Long value) throws Exception {  // 3. 使用累加器  counter.add(1);  System.out.println("Thread id: " + getRuntimeContext().getIndexOfThisSubtask() + ",counter: " + counter.getLocalValue());  return value;  }  });   map.writeAsText("data/output/accumulators", FileSystem.WriteMode.OVERWRITE);  JobExecutionResult jobExecutionResult = env.execute();   // 4. 从执行结果中获取累加器的最终值  int sum = jobExecutionResult.getAccumulatorResult("sum");  System.out.println("Finally Accumulator result is: " + sum);  } } |

* Flink Broadcast和Accumulators的区别
  + Broadcast(广播变量)允许程序员将一个只读的变量缓存在每台机器上，而不用在任务之间传递变量。广播变量可以进行共享，但是不可以进行修改
  + Accumulators(累加器)是可以在不同任务中对同一个变量进行累加操作(简单的说就是由Flink为我们实现线程安全)（对于分布式系统来说，不仅仅累加的各个Task会跨Slot，甚至会跨机器进行累加，所以，传统的累加不可用，需要用Flink提供的累加器才可以的哦）。

### Flink的分布式缓存

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！

缓存的使用流程：

使用ExecutionEnvironment实例对本地的或者远程的文件（例如：HDFS上的文件）,为缓存文件指定一个名字注册该缓存文件！当程序执行时候，**Flink会自动将复制文件或者目录到所有worker节点的本地文件系统中**，函数可以根据名字去该节点的本地文件系统中检索该文件！

**和广播变量的区别：**

* 广播变量广播的是**程序中的变量(DataSet)数据**，分布式缓存广播的是文件
* 广播变量将数据广播到各个TaskManager的内存中，分布式缓存广播到各个TaskManager的本地文件系统
* 用法

使用Flink运行时环境的 registerCachedFile 注册一个分布式缓存

在操作中，使用 getRuntimeContext.getDistributedCache.getFile ( 文件名 )获取分布式缓存

* 示例

创建一个 成绩 数据集

|  |
| --- |
| List( (1, "语文", 50),(2, "数学", 70), (3, "英文", 86)) |

请通过分布式缓存获取到学生姓名，将数据转换为

|  |
| --- |
| List( ("张三", "语文", 50),("李四", "数学", 70), ("王五", "英文", 86)) |

注： 资料\测试数据源\distribute\_cache\_student 文件保存了学生ID以及学生姓名

* 操作步骤
  + 将 distribute\_cache\_student 文件上传到HDFS / 目录下
  + 获取批处理运行环境
  + 创建成绩数据集
  + 对 成绩 数据集进行map转换，将（学生ID, 学科, 分数）转换为（学生姓名，学科，分数）
    - RichMapFunction 的 open 方法中，获取分布式缓存数据
    - 在 map 方法中进行转换
  + 实现 open 方法
    - 使用 getRuntimeContext.getDistributedCache.getFile 获取分布式缓存文件
    - 使用 Scala.fromFile 读取文件，并获取行
    - 将文本转换为元组（学生ID，学生姓名），再转换为List
  + 实现 map 方法
    - 从分布式缓存中根据学生ID过滤出来学生
    - 获取学生姓名
    - 构建最终结果元组
  + 打印测试
* 参考代码

|  |
| --- |
| /\*\*  \* 分布式缓存  \*/ public class DistributedCacheDemo {  public static void main(String[] args) throws Exception {  // Env  ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();   // 准备分数信息数据集  DataSource<Tuple3<Integer, String, Integer>> scoreInfoDataSet = env.fromElements(  Tuple3.of(1, "数据结构", 99),  Tuple3.of(2, "英语", 100),  Tuple3.of(3, "C++", 96),  Tuple3.of(5, "Java", 97),  Tuple3.of(3, "Scala", 100)  );   /\*  分布式缓存和广播变量的使用步骤基本差不多，有一点不同  1. 设置它, 使用env.registerCachedFile来注册分布式缓存.  2. 使用它, 在算子内部调用getRuntimeContext.getDistributedCache.getFile(File)来获取分布式缓存的文件  \*/   // 1. 注册分布式缓存  env.registerCachedFile("data/input/distributed\_student.txt", "student");   // 通过map方法来组合数据和获取分布式缓存文件内容  MapOperator<Tuple3<Integer, String, Integer>, Tuple3<String, String, Integer>> result = scoreInfoDataSet.map(new RichMapFunction<Tuple3<Integer, String, Integer>, Tuple3<String, String, Integer>>() {  // 定义一个map变量来接收学生信息对象  final Map<Integer, String> map = new HashMap<Integer, String>();   // 通过open方法来获得分布式缓存的文件, 并将数据放入map中  @Override  public void open(Configuration parameters) throws Exception {  super.open(parameters);  File distributedFile = getRuntimeContext().getDistributedCache().getFile("student");  BufferedReader bufferedReader = new BufferedReader(new FileReader(distributedFile));  // Lambda方式 // bufferedReader.lines().forEach((String line) -> { // String[] elements = line.split(","); // map.put(Integer.parseInt(elements[0]), elements[1]); // });   // 普通方式  String line = null;  while ((line = bufferedReader.readLine()) != null) {  String[] elements = line.split(",");  map.put(Integer.parseInt(elements[0]), elements[1]);  }  }   @Override  public Tuple3<String, String, Integer> map(Tuple3<Integer, String, Integer> value) throws Exception {  return Tuple3.of(map.getOrDefault(value.f0, "未知学生姓名"), value.f1, value.f2);  }  });   result.print();  } } |