Flink

简介

是一个框架和分布式处理引擎，用于无界有界数据流进行转换

流与批的世界观

批处理的特点是有界、持久、大量，非常适合需要访问全套记录才能完成的计算工作，一般用于离线统计。流处理的特点是无界、实时， 无需针对整个数据集执行操作，而是对通过系统传输的每个数据项执行操作，一般用于实时统计。

在spark的世界观中，一切都是由批次组成的，离线数据是一个大批次，而实时数据是由一个一个无限的小批次组成的。

而在flink的世界观中，一切都是由流组成的，离线数据是有界限的流，实时数据是一个没有界限的流，这就是所谓的有界流和无界流。

无界数据流：无界数据流有一个开始但是没有结束，它们不会在生成时终止并提供数据，必须连续处理无界流，也就是说必须在获取后立即处理event。对于无界数据流我们无法等待所有数据都到达，因为输入是无界的，并且在任何时间点都不会完成。处理无界数据通常要求以特定顺序（例如事件发生的顺序）获取event，以便能够推断结果完整性。

有界数据流：有界数据流有明确定义的开始和结束，可以在执行任何计算之前通过获取所有数据来处理有界流，处理有界流不需要有序获取，因为可以始终对有界数据集进行排序，有界流的处理也称为批处理。

有界数据集

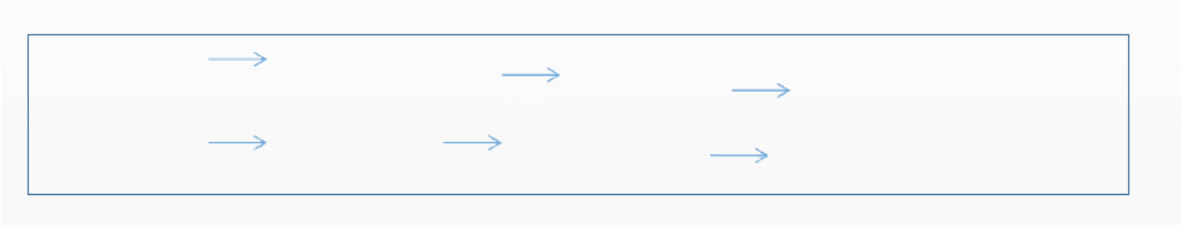
有界数据集对开发者来说都很熟悉，在常规的处理中我们都会从Mysql，文本等获取数据进行计算分析。我们在处理此类数据时，特点就是数据是静止不动的。也就是说，没有再进行追加。又或者说再处理的当时时刻不考虑追加写入操作。所以有界数据集又或者说是有时间边界。在某个时间内的结果进行计算。那么这种计算称之为批计算，批处理。BatchProcessing

无界数据集

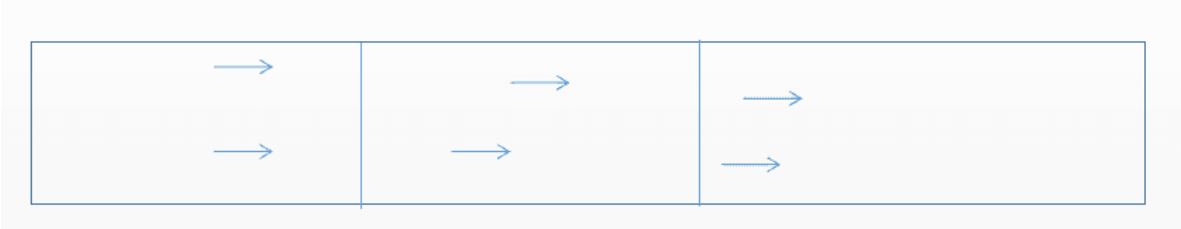
对于某些场景，类似于Kafka持续的计算等都被认定为无界数据集，无界数据集是会发生持续变更的、连续追加的。例如:服务器信令、网络传输流、实时日志信息等。对于此类持续变更、追加的数据的计算方式称之为流计算。StreamingProcessing

场景比较

无界数据集与有界数据集有点类似于池塘和江河，我们在计算池塘中的鱼的数量时只需要把池塘中当前所有的鱼都计算一次就可以了。那么当前时刻，池塘中有多少条鱼就是结果。 无界数据集类似于江河中的鱼，在奔流到海的过程中每时每刻都会有鱼流过而进入大海。那么计算鱼的数量就像是持续追加的。



有界数据集与无界数据集是一个相对模糊的概念，如果数据一条一条的经过处理引擎那么则可以认为是无界的，那么如果每间隔一分钟、一小时、一天进行一次计算那么则认为这一段时间的数据又相对是有界的。有界的数据又可以把数据一条一条的通过计算引擎，造成无界的数据集。所以，有界数据集与无界数据集可以存在互换的。因此业内也就开始追寻 批流统一 的框架。



能够同时实现批处理与流处理的框架有Apache Spark和Apache Flink，而Apache Spark的流处理场景是一个微批场景，也就是它会在特定的时间间隔发起一次计算。而不是每条都会触发计算。也就是相当于把无界数据集切分为小量的有界数据。

Apache Flink基于有界数据集与无界数据集的特点，最终将批处理与流处理混合到同一套引擎当中，用户使用Apache Flink引擎能够同时实现批处理与流处理任务。。

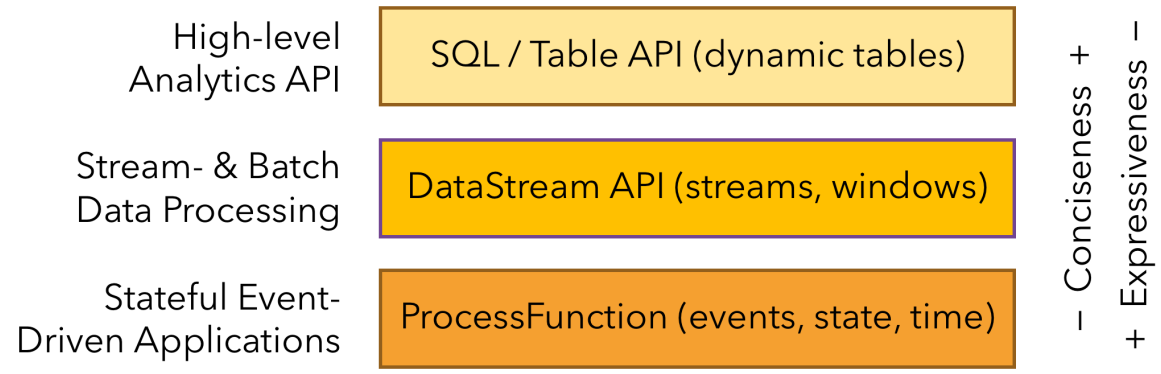
Flink的特点和优点

1、同时支持事件时间和处理时间语义。事件时间语义能够针对无序事件提供精确、一致的结果；处理时间语义能够用在具有极低延迟需求的应用中。

2、提供精确一次（exactly once）的状态一致性保障。

3、层次化的API在表达能力和易用性方面各有权衡。表达能力由强到弱（易用性由弱到强）依次是：ProcessFunction、DataStream API、SQL/Table API。

Flink API提供了通用的流操作原语（如窗口划分和异步操作）以及精确控制时间和状态的接口。



4、提供常见存储系统的连接器，Kafka，Elasticsearch，JDBC

5、checkpoint和savepoint

6、支持高可用性配置（无单点失效），与k8s、Yarn、Apache Mesos紧密集成，快速故障恢复，动态扩缩容作业。

7、提供详细、可自由定制的系统及应用指标（metrics）集合，用于提前定位和响应问题。

8、社区正在努力将Flink发展成为在API及运行时层面都能做到批流统一。

9、对开发者友好，Flink的嵌入式执行模式可将应用自身连同整个Flink系统在单个JVM进程内启动，方便在IDE里运行和调试Flink作业

Flink 运行时的组件

作业管理器（JobManager）
资源管理器（ResourceManager）
任务管理器（TaskManager）
以及分发器（Dispatcher）

作业管理器（JobManager）

控制一个应用程序执行的主进程，也就是说，每个应用程序都会被一个不同的 JobManager 所控制执行。JobManager 会先接收到要执行的应用程序，这个应用程序会包括：作业图（JobGraph）、逻辑数据流图（logical dataflow graph）和打包了所有的类、库和其它资源的 JAR 包。JobManager 会把 JobGraph 转换成一个物理层面的数据流图，这个图被叫做“执行图”（ExecutionGraph），包含了所有可以并发执行的任务。JobManager 会向资源管理器（ResourceManager）请求执行任务必要的资源，也就是任务管理器（TaskManager）上的插槽（slot）。一旦它获取到了足够的资源，就会将执行图分发到真正运行它们的 TaskManager 上。而在运行过程中，JobManager 会负责所有需要中央协调的操作，比如说检查点（checkpoints）的协调。

资源管理器（ResourceManager）

主要负责管理任务管理器（TaskManager）的插槽（slot），TaskManger 插槽是 Flink 中定义的处理资源单元。Flink 为不同的环境和资源管理工具提供了不同资源管理器，比如 YARN、Mesos、K8s，以及 standalone 部署。当 JobManager 申请插槽资源时，ResourceManager 会将有空闲插槽的 TaskManager 分配给 JobManager。如果 ResourceManager 没有足够的插槽来满足 JobManager 的请求，它还可以向资源提供平台发起会话，以提供启动 TaskManager 进程的容器。另外，ResourceManager 还负责终止空闲的 TaskManager，释放计算资源。

任务管理器（TaskManager）

Flink 中的工作进程。通常在 Flink 中会有多个 TaskManager 运行，每一个 TaskManager 都包含了一定数量的插槽（slots）。插槽的数量限制了 TaskManager 能够执行的任务数量。启动之后，TaskManager 会向资源管理器注册它的插槽；收到资源管理器的指令后，TaskManager 就会将一个或者多个插槽提供给 JobManager 调用。JobManager 就可以向插槽分配任务（tasks）来执行了。在执行过程中，一个 TaskManager 可以跟其它运行同一应用程序的 TaskManager 交换数据。

分发器（Dispatcher）

可以跨作业运行，它为应用提交提供了 REST 接口。当一个应用被提交执行时，分发器就会启动并将应用移交给一个 JobManager。由于是 REST 接口，所以 Dispatcher 可以作为集群的一个 HTTP 接入点，这样就能够不受防火墙阻挡。Dispatcher 也会启动一个 Web UI，用来方便地展示和监控作业执行的信息。Dispatcher 在架构中可能并不是必需的，这取决于应用提交运行的方式。

Flink任务提交流程

我们来看看当一个应用提交执行时，Flink 的各个组件是如何交互协作的：

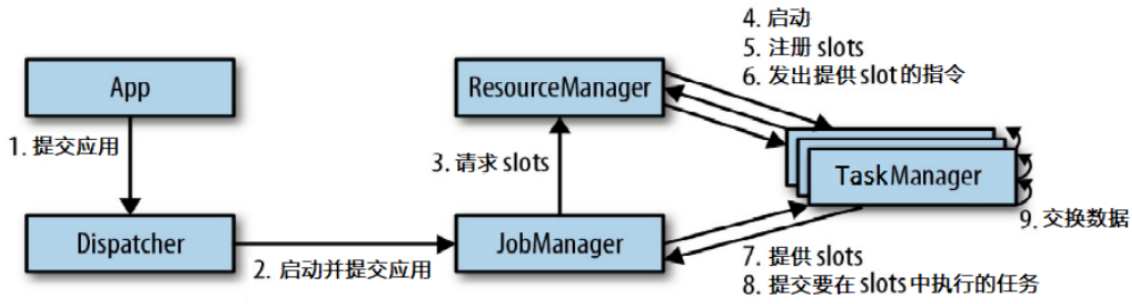


图 任务提交和组件交互流程

图 任务提交和组件交互流程

上图是从一个较为高层级的视角，来看应用中各组件的交互协作。如果部署的集群环境不同（例如 YARN，Mesos，Kubernetes，standalone 等），其中一些步骤可以被省略，或是有些组件会运行在同一个 JVM 进程中。

具体地，如果我们将 Flink 集群部署到 YARN 上，那么就会有如下的提交流程：

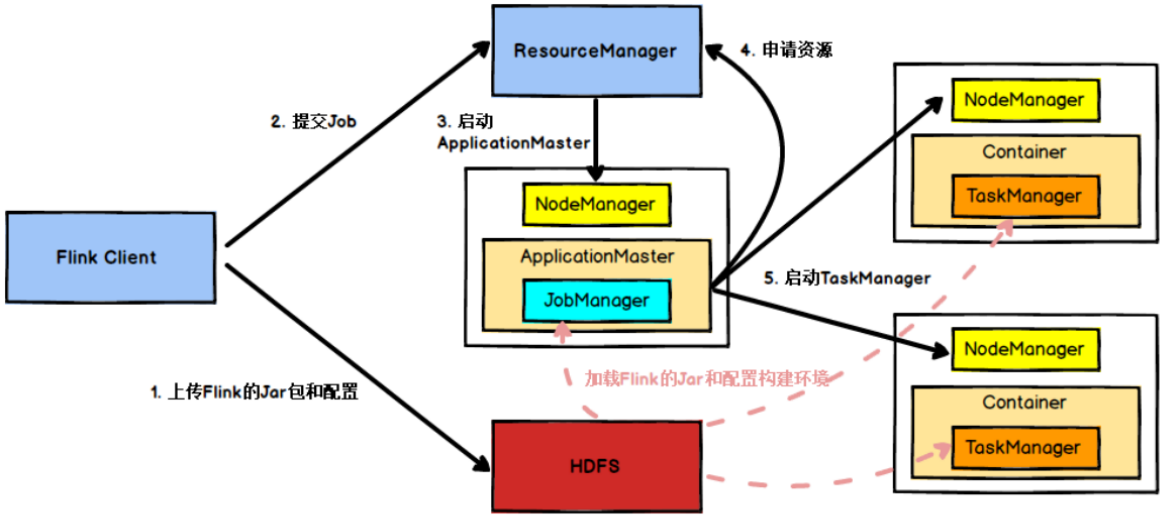


图 Yarn 模式任务提交流程

图 Yarn 模式任务提交流程

```
Flink 任务提交后，Client 向 HDFS 上传 Flink 的 Jar 包和配置，之后向 Yarn ResourceManager 提交任务，ResourceManager 分配 Container 资源并通知对应的 NodeManager 启动 ApplicationMaster，ApplicationMaster 启动后加载 Flink 的 Jar 包和配置构建环境，然后启动 JobManager，之后 ApplicationMaster 向 ResourceManager 申请资源启动 TaskManager ， ResourceManager 分 配 Container 资 源 后 ， 由 ApplicationMaster 通 知 资 源 所 在 节 点 的 NodeManager 启 动 TaskManager ，NodeManager 加载 Flink 的 Jar 包和配置构建环境并启动 TaskManager，TaskManager 启动后向 JobManager 发送心跳包，并等待 JobManager 向其分配任务。
```

Flink任务调度原理

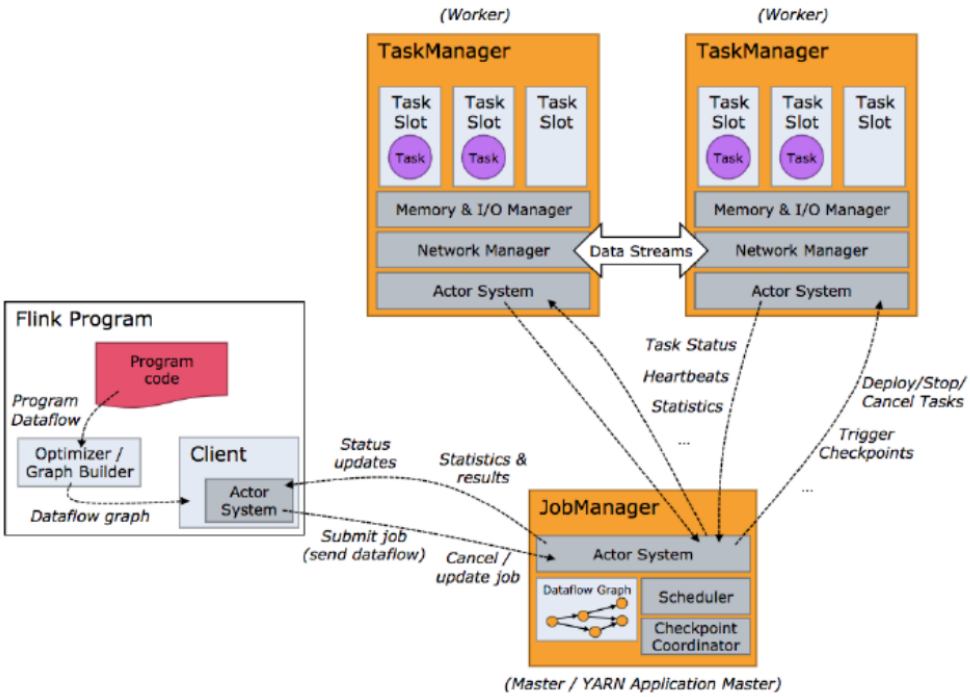


图 任务调度原理

图 任务调度原理

客户端不是运行时和程序执行的一部分，但它用于准备并发送dataflow(JobGraph)给Master(JobManager)，然后，客户端断开连接或者维持连接以等待接收计算结果。

当 Flink 集群启动后，首先会启动一个 JobManger 和一个或多个的 TaskManager。由 Client 提交任务给 JobManager，JobManager 再调度任务到各个 TaskManager 去执行，然后 TaskManager 将心跳和统计信息汇报给 JobManager。TaskManager 之间以流的形式进行数据的传输。上述三者均为独立的 JVM 进程。

1.Client 为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming 的任务），也可以不结束并等待结果返回。

2.JobManager主要负责调度 Job 并协调 Task 做 checkpoint，职责上很像Storm 的 Nimbus。从 Client 处接收到 Job 和 JAR 包等资源后，会生成优化后的执行计划，并以 Task 的单元调度到各个 TaskManager 去执行。

3.TaskManager 在启动的时候就设置好了槽位数（Slot），每个 slot 能启动一个Task，Task 为线程。从 JobManager 处接收需要部署的 Task，部署启动后，与自己的上游建立 Netty 连接，接收数据并处理。

TaskManger 与 Slots

Flink 中每一个 worker(TaskManager)都是一个 **JVM 进程**，它可能会在独立的线程上执行一个或多个 subtask。为了控制一个 worker 能接收多少个 task，worker 通过 task slot 来进行控制（一个 worker 至少有一个 task slot）。

每个 task slot 表示 TaskManager 拥有资源的**一个固定大小的子集**。假如一个 TaskManager 有三个 slot，那么它会将其管理的内存分成三份给各个 slot。资源 slot 化意味着一个 subtask 将不需要跟来自其他 job 的 subtask 竞争被管理的内存，取而

代之的是它将拥有一定数量的内存储备。需要注意的是，这里不会涉及到 CPU 的隔离，slot 目前仅仅用来隔离 task 的受管理的内存。

通过调整 task slot 的数量，允许用户定义 subtask 之间如何互相隔离。如果一个 TaskManager 一个 slot，那将意味着每个 task group 运行在独立的 JVM 中（该 JVM 可能是通过一个特定的容器启动的），而一个 TaskManager 多个 slot 意味着更多的 subtask 可以共享同一个 JVM。而在同一个 JVM 进程中的 task 将共享 TCP 连接（基于多路复用）和心跳消息。它们也可能共享数据集和数据结构，因此这减少了每个 task 的负载。

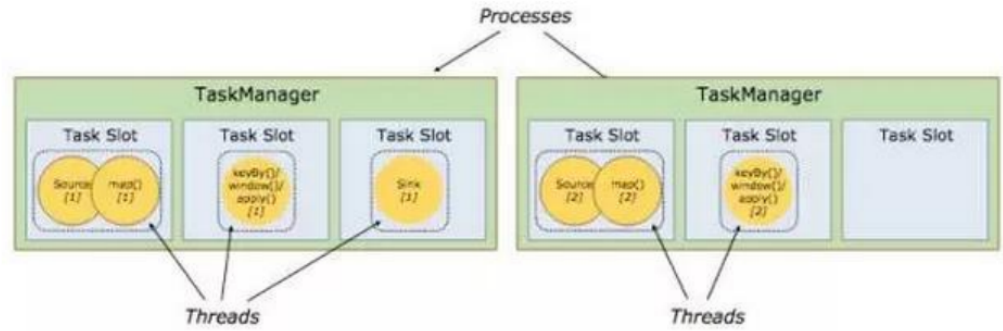


图 TaskManager 与 Slot

图 TaskManager 与 Slot

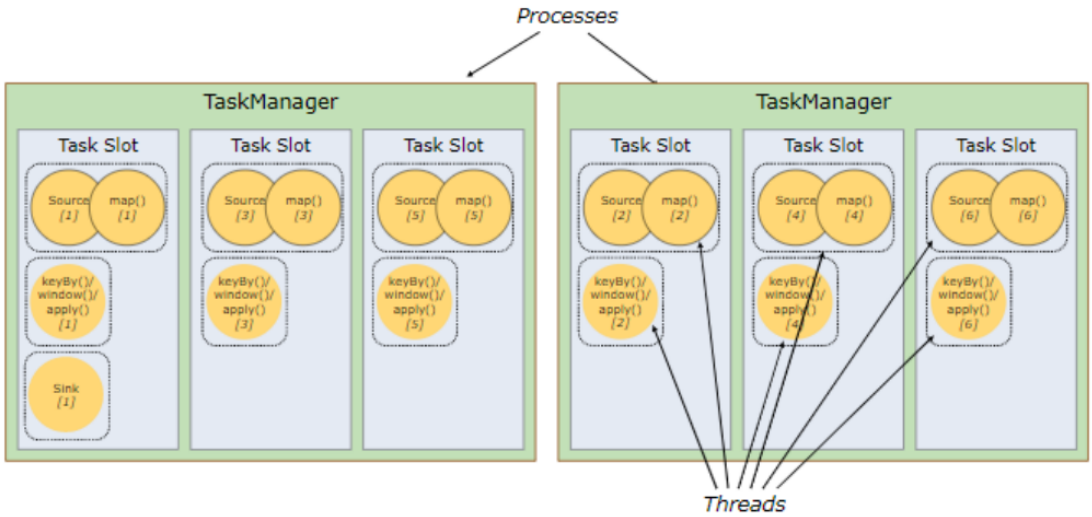


图 子任务共享 Slot

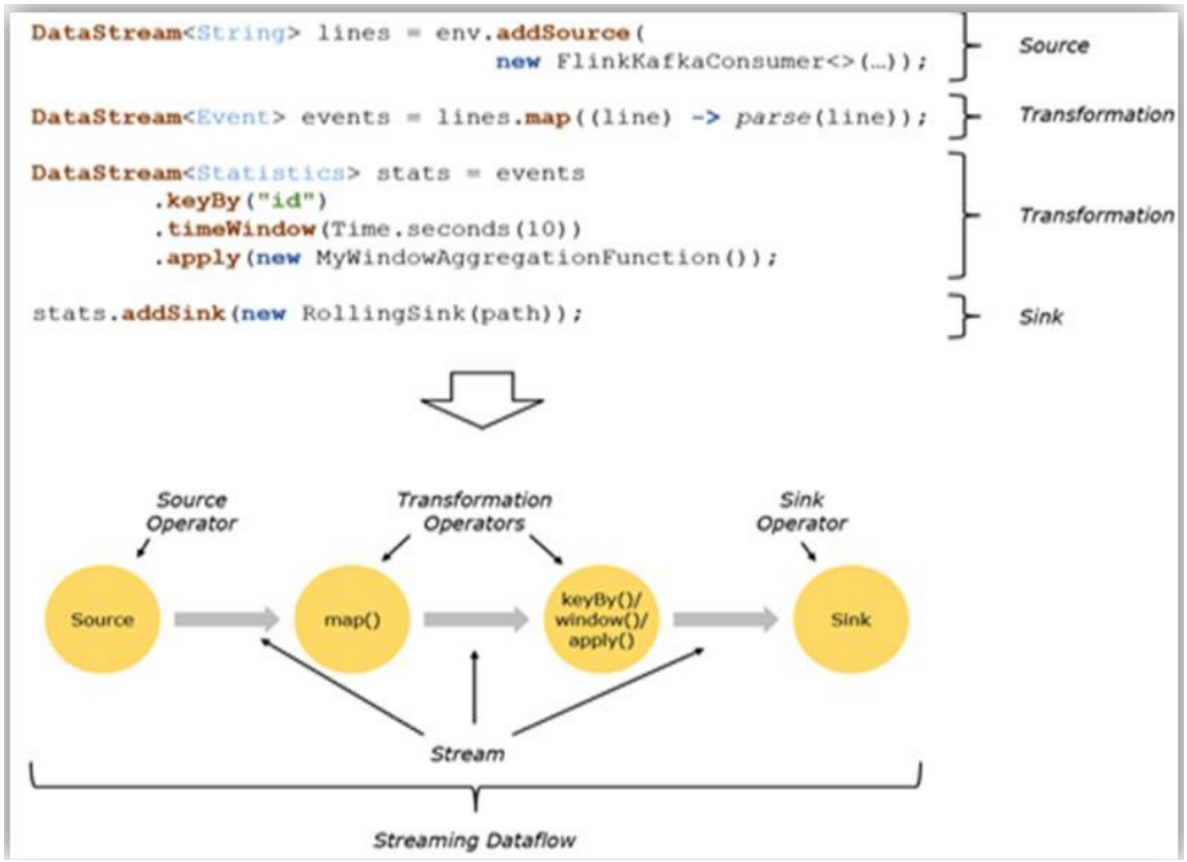
图 子任务共享 Slot

默认情况下，Flink 允许子任务共享 slot，即使它们是不同任务的子任务（前提是它们来自同一个 job）。 这样的结果是，一个 slot 可以保存作业的整个管道。

Task Slot **是静态的概念，是指** **TaskManager** **具有的并发执行能力**，可以通过参数 taskmanager.numberOfTaskSlots 进行配置；
而**并行度** **parallelism** **是动态概念，
即**TaskManager** **运行程序时实际使用的并发能力**，可以通过参数 parallelism.default进行配置。

也就是说，假设一共有 3 个 TaskManager，每一个 TaskManager 中的分配 3 个 TaskSlot，也就是每个 TaskManager 可以接收 3 个 task，一共 9 个 TaskSlot，如果我们设置 parallelism.default=1，即运行程序默认的并行度为 1，9 个 TaskSlot 只用了 1 个，有 8 个空闲，因此，设置合适的并行度才能提高效率。

程序与数据流（DataFlow）



所有的 Flink 程序都是由三部分组成的： **Source** 、**Transformation** 和 **Sink**。Source 负责读取数据源，Transformation 利用各种算子进行处理加工，Sink 负责输出。

在运行时，Flink 上运行的程序会被映射成“逻辑数据流”（dataflows），它包含了这三部分。**每一个** **dataflow** **以一个或多个** **sources** **开始以一个或多个** **sinks** **结**
束。dataflow 类似于任意的有向无环图（DAG）。在大部分情况下，程序中的转换运算（transformations）跟 dataflow 中的算子（operator）是一一对应的关系，但有时候，一个 transformation 可能对应多个 operator。

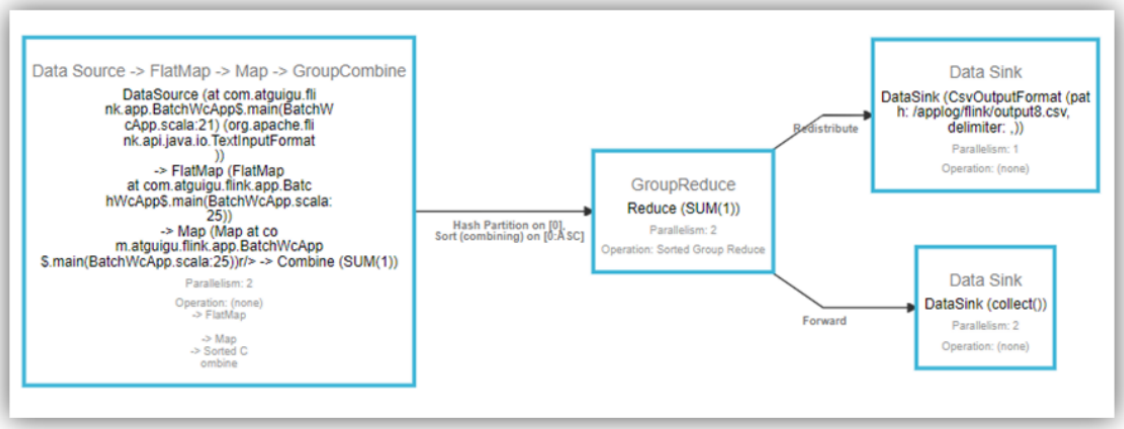


图 程序与数据流

执行图（**ExecutionGraph）**

由 Flink 程序直接映射成的数据流图是 StreamGraph，也被称为逻辑流图，因为它们表示的是计算逻辑的高级视图。为了执行一个流处理程序，Flink 需要将逻辑流图转换为物理数据流图（也叫执行图），详细说明程序的执行方式。

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。

并行度（**Parallelism）

Flink 程序的执行具有**并行、分布式**的特性。
在执行过程中，一个流（stream）包含一个或多个分区（stream partition），而每一个算子（operator）可以包含一个或多个子任务（operator subtask），这些子任务在不同的线程、不同的物理机或不同的容器中彼此互不依赖地执行。
一个特定算子的子任务（**subtask****）的个数被称之为其并行度（****parallelism****）**。
一般情况下，一个流程序的并行度，可以认为就是其所有算子中最大的并行度。一个程序中，不同的算子可能具有不同的并行度。

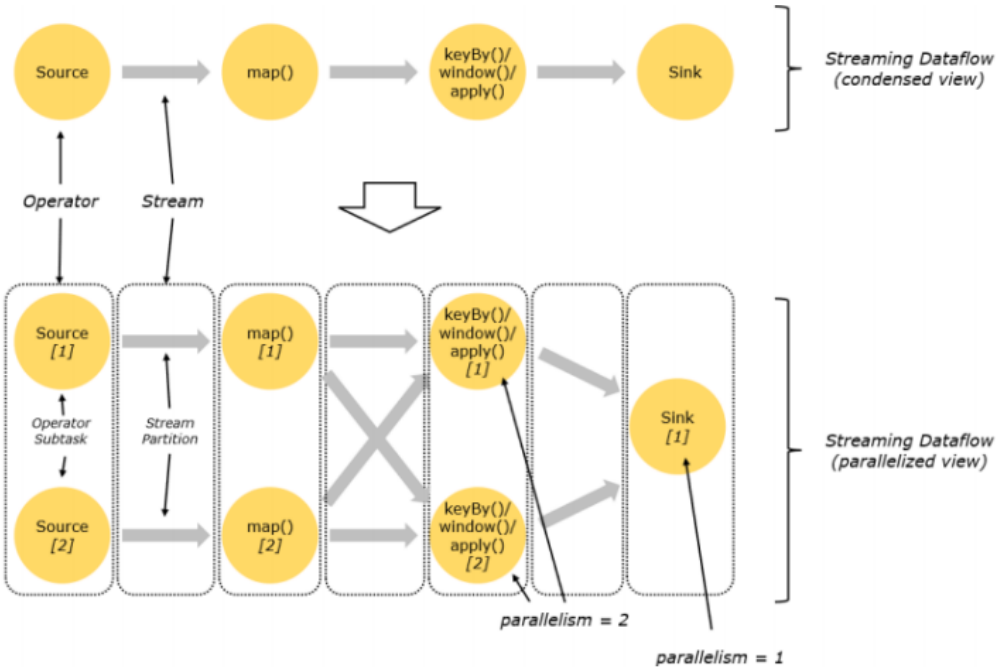


图 并行数据流

Stream 在算子之间传输数据的形式可以是 one-to-one(forwarding)的模式也可以是 redistributing 的模式，具体是哪一种形式，取决于算子的种类。
One-to-one：stream(比如在 source 和 map operator 之间)维护着分区以及元素的顺序。那意味着 map 算子的子任务看到的元素的个数以及顺序跟 source 算子的子任务生产的元素的个数、顺序相同，map、fliter、flatMap 等算子都是 one-to-one 的对应关系。
类似于 spark 中的**窄依赖**

Redistributing：stream(map()跟 keyBy/window 之间或者 keyBy/window 跟 sink 之间)的分区会发生改变。每一个算子的子任务依据所选择的 transformation 发送数据到不同的目标任务。例如，keyBy() 基于 hashCode 重分区、broadcast 和 rebalance 会随机重新分区，这些算子都会引起 redistribute 过程，而 redistribute 过程就类似于 Spark 中的 shuffle 过程。

类似于 spark 中的**宽依赖**

任务链（Operator Chains）

相同并行度的** **one to one** **操作**，Flink 这样相连的算子链接在一起形成一个 task，原来的算子成为里面的一部分。将算子链接成 task 是非常有效的优化：它能减少线程之间的切换和基于缓存区的数据交换，在减少时延的同时提升吞吐量。链接的行为可以在编程 API 中进行指定。

一：没有一张大牌开路再顺的小牌也出不去，说明领导很重要
二：小王一出，基本都会被大王打，说明老大在，老二最好不要发话
三：无论你多会打牌，多会记牌，都抵不过人家的一手好牌，说明运气也很重要
四：假如一堆小牌连不起来，即使你拿着双王王炸，也未必会赢，说明再牛的领导也需要一个会合作的团队
五：必要多的时候，即使拆散自己的牌，也要送走搭档，说明大局很重要
六：你手里边有很厉害的牌，也都是地主先出，说明身份很重要
七：再好的牌，再好的技术，口袋也没钱，也不敢拍牌桌，说明资金很重要
八：一会和另外两家斗，一会和这家斗，一会又和另外一家合作，说明只有永远的利益，没有永远的敌人

体系

1.运行架构

2.API

3.时间语义

4.窗口操作

5.底层API

6.状态管理

7.容错机制

8.状态一致性

开发（慕课网 教学）-> （基础篇）

词频统计(要学会功能拆解)

```
先读取数据，根据不同需求将数据拆开不同的结构
（批处理）
Java  : readTestFlie -> FlapMap -> groupBy -> sum
Scala : readTestFlie ->  FlapMap -> Filter -> Map -> groupBy -> sum
（流处理）
Java: socketTextStream -> FlapMap -> KeyBy -> sum
Scala:socketTextStream -> FlapMap -> Map -> KeyBy -> timeWindow -> sum
```

在执行过程中，一个流（stream）包含一个或多个分区（stream partition），而每一个算子（operator）可以包含一个或多个子任务（operator subtask），这些子任务在不同的线程、不同的物理机或不同的容器中彼此互不依赖地执行。

Flink的编程模型

```
1.获取执行环境
2.获取数据
3.transformation
4.Sink
5.触发执行 (流处理触发)
```

在Flink中可以自定义类去指定key 和sum， Java中可以用封装类去指定，Scala用样例类即可

Java代码

```
public class StreamingWCBean {
    public static void main(String[] args) throws Exception{
        //TODO 创建流处理环境
        final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //TODO 并行度
        env.setParallelism( 1 );
        //TODO 端口
        DataStreamSource<String> source = env.socketTextStream( "localhost", 9999
);
        //TODO 可以新建一个类 接口 FlatMapFunction执行
        SingleOutputStreamOperator<Bean> flatMapDStream = source.flatMap( new
FlatMapFunction<String, Bean>() {
            public void flatMap(String value, Collector<Bean> out) throws
Exception {
                String[] splits = value.toLowerCase().split( "," );
                for (String split : splits) {
                    if (split.length() > 0) {
                        out.collect( new Bean( split, 1 ) );
                    }
                }
            }
        } );
        flatMapDStream.keyBy( "word" )
                .timeWindow( Time.seconds( 5 ) )
                .sum( "count" )
                .print( "job" );
        env.execute( "StreamingWCBean" );
    }
}
```

Scala代码

```
object StreamWcScalaAPP {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置并行度
    env.setParallelism(1)
   val socketDStream = env.socketTextStream("localhost",9999)
    socketDStream.flatMap(_.split(","))
      .filter(_.nonEmpty)
        .map(x => WC(x,1)).
      keyBy("word")
        .timeWindow(Time.seconds(5))
      .sum("count")
      .print()
```

```
    env.execute("StreamWSScalaAPP")
  }
}
case class WC(word:String,count:Int)
```

在Flink中可以实现直接在类中重写方法，也可以自定义一个类将其传参使用，有可以用RichFunction增强函数，方法同普通的函数一样

DataSet

DataSet的数据源

```
DataAPI应用开发程序 -> DataSource(数据源) -> transformation -> Sink(目的地) -> 计数器
-> 分布式缓存
DataSource: 基于文件:readTextFile(掌握) -> (可以用递归文件),readTextFileWithValue,
                    readCsvFile(掌握),->（难
点）,readFileOfPrimitives,readSequenceFile
            基于集合:fromCollection(Seq),fromCollection(Iterator) （掌握）
                    fromElements,fromParallelCollection,generateSequence
            Generic: readFile,createInput
```

关于Csv对象

```
def readCsvFile[T : ClassTag : TypeInformation]( //TODO 注意泛型
      filePath: String,
      lineDelimiter: String = "\n",
      fieldDelimiter: String = ",",
      quoteCharacter: Character = null,
      ignoreFirstLine: Boolean = false,
      ignoreComments: String = null,
      lenient: Boolean = false,
      includedFields: Array[Int] = null,
      pojoFields: Array[String] = null): DataSet[T]
 filePath:路径 （本地或者hdfs）
 lineDelimiter : 分隔行的字符串，默认为换行符.
 fieldDelimiter : 默认逗号分割
 quoteCharacter : 用于带引号的字符串分析的字符，默认情况下禁用。
 ignoreFirstLine : 你的第一行在文件里面是否应该忽略
 ignoreComments :  以给定String开头的行将被忽略，默认情况下处于禁用状态。
 lenient : 解析器是否应静默忽略格式错误的行。
 includedFields : 文件中应读取的字段。默认情况下，所有字段均被读取。(用数组下标去获取数据)
 pojoFields : 用Java的封装类，去描述，用数据数值去确定数据
```

关于Csv对象比较难，下面先将代码放入 Java版本

```
public class JavaDataSetCSVSourceApp {
    public static void main(String[] args) throws Exception{
        ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
        Csv( env );
    }
    //TODO 方式一
    public static void Csv(ExecutionEnvironment env) throws Exception{
        String file = "Flink/data/people.csv";
        CsvReader reader = env.readCsvFile( file );
```

```
        //TODO Java的ignoreFirstLine() 默认为忽略的
        CsvReader csvReader = reader.ignoreFirstLine(); // true
        DataSource<Person> pojoType = csvReader.pojoType( Person.class, "name",
"age", "work" );
        pojoType.print();
    }
}
```

Scala版本

```
object DataSetCSVSourceApp {
  def main(args: Array[String]): Unit = {
    val env = ExecutionEnvironment.getExecutionEnvironment
      Csv(env)
      println("----方式一和方式二的交界处-------")
      CSv(env)
      println("----方式二和方式三的交界处-------")
      CSV(env)
      println("----方式三和方式四的交界处-------")
      csv(env)
    }
  //TODO 方式一
  def Csv(env:ExecutionEnvironment): Unit ={
    val file = "Flink/data/people.csv"
    val unit = env.readCsvFile[(String,Int,String)](file,ignoreFirstLine = true)
    unit.print()
  }
  //TODO 方式二
  def CSv(env:ExecutionEnvironment): Unit ={
    val file = "Flink/data/people.csv"
    val unit = env.readCsvFile[(String,Int,String)](file,ignoreFirstLine =
true,includedFields = Array(0,1,2))
    unit.print()
  }
  //TODO 方式三
  case class Perple(name:String,age:Int,job:String)
  def CSV(env:ExecutionEnvironment): Unit ={
    val file = "Flink/data/people.csv"
    val unit = env.readCsvFile[Perple](file,ignoreFirstLine = true,includedFields
= Array(0,1,2))
    unit.print()
  }
  //TODO 方式四 POJO -> Java实现
  def csv(env: ExecutionEnvironment): Unit = {
    val file = "Flink/data/people.csv"
    val unit = env.readCsvFile[Person](file,ignoreFirstLine = true,pojoFields =
Array("name","age","work"))
    unit.print()
  }
}
```

递归的文件的方式也有难点，需要配置一些参数，附上两个版本的代码

Scala代码

```
object DataSetRecursiveSourceApp {
  def main(args: Array[String]): Unit = {
```

```
    //TODO 递归文件夹 实现方式
    val env = ExecutionEnvironment.getExecutionEnvironment
    recusive(env)
  }
  def recusive(env: ExecutionEnvironment): Unit = {
    val file = "Flink/dahuan"
    //TODO 配置递归参数，参考官网 ： https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/batch/
    val parameters = new Configuration
    parameters.setBoolean("recursive.file.enumeration", true)
    //TODO 读取文件
    env.readTextFile(file).withParameters(parameters).print()
  }
}
```

Java代码

```
public class JavaDataSetRecursiveSourceApp {
    public static void main(String[] args) throws Exception {
        ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
        recusive( env );
    }
    public static void recusive(ExecutionEnvironment env) throws Exception{
        // create a configuration object
        Configuration parameters = new Configuration();
        // set the recursive enumeration parameter
        parameters.setBoolean( "recursive.file.enumeration", true );
        DataSource<String> textFile = env.readTextFile( "Flink/dahuan" ).withParameters(parameters);
        textFile.print();
    }
}
```

DataSet的常见的Transformationn操作

```
map,filer,mappartition,first,flatmap,distinct,join(where,equalTo,apply),
outerJoin(leftouterJoin,rightouterJoin,fullouterJoin),Union,sortBy
cross(笛卡尔积？)
```

连接操作(outerJoin)有难点，附上两个版本代码

Scala代码

```
object DataSetOuterJoinApp {
  def main(args: Array[String]): Unit = {
    val env = ExecutionEnvironment.getExecutionEnvironment
    //TODO 左外连接
    leftouterJoinFunction(env)
    println("-------左外连接和右外连接的分界线----------------")
    //TODO 右外连接
    rightouterJoinFunction(env)
    println("-------右外连接的分界线和全外连接的分界线----------------")
    //TODO 全外连接
    fullouterJoinFunction(env)
  }
```

```
  //TODO 左外连接 先从左边第一个开始
  def leftouterJoinFunction(env: ExecutionEnvironment): Unit ={
    val unit1 = new ListBuffer[(Int,String)]  //TODO 编号，姓名
    unit1.append((1,"PK哥"))
    unit1.append((2,"J哥"))
    unit1.append((3,"小队长"))
    unit1.append((4,"猪头呼"))

    val unit2 = new ListBuffer[(Int,String)]  //TODO 编号，城市
    unit2.append((1,"北京"))
    unit2.append((2,"上海"))
    unit2.append((3,"广州"))
    unit2.append((5,"深圳"))
    //TODO 创建数据源
    val data1 = env.fromCollection(unit1)
    val data2 = env.fromCollection(unit2)

    data1.leftOuterJoin(data2).where(0).equalTo(0).apply((first,second)=>{
      //TODO 既然优先出第一个，那么就要考虑第二个是否为空
      if (second == null) {
        (first._1,first._2,"-")
      }else{
        (first._1,first._2,second._2)
      }
    }).print()
  }
  //TODO 右外连接，先从右边第一个开始
  def rightouterJoinFunction(env:ExecutionEnvironment): Unit ={
    val unit1 = new ListBuffer[(Int,String)]  //TODO 编号，姓名
    unit1.append((1,"PK哥"))
    unit1.append((2,"J哥"))
    unit1.append((3,"小队长"))
    unit1.append((4,"猪头呼"))

    val unit2 = new ListBuffer[(Int,String)]  //TODO 编号，城市
    unit2.append((1,"北京"))
    unit2.append((2,"上海"))
    unit2.append((3,"广州"))
    unit2.append((5,"深圳"))
    //TODO 创建数据源
    val data1 = env.fromCollection(unit1)
    val data2 = env.fromCollection(unit2)

    data1.rightOuterJoin(data2).where(0).equalTo(0).apply((first,second)=>{
      //TODO 既然优先出第二个，那么就要考虑第一个是否为空
      if (first == null) {
        (second._1,"-",second._2)
      }else{
        (first._1,first._2,second._2)
      }
    }).print()
  }
  //TODO 全外连接
  def fullouterJoinFunction(env:ExecutionEnvironment): Unit ={
    val unit1 = new ListBuffer[(Int,String)]  //TODO 编号，姓名
    unit1.append((1,"PK哥"))
    unit1.append((2,"J哥"))
    unit1.append((3,"小队长"))
```

```
    unit1.append((4,"猪头呼"))

    val unit2 = new ListBuffer[(Int,String)]  //TODO 编号，城市
    unit2.append((1,"北京"))
    unit2.append((2,"上海"))
    unit2.append((3,"广州"))
    unit2.append((5,"深圳"))
    //TODO 创建数据源
    val data1 = env.fromCollection(unit1)
    val data2 = env.fromCollection(unit2)

    data1.fullOuterJoin(data2).where(0).equalTo(0).apply((first,second)=>{
      //TODO 都考虑到了就全面排查一下
      if (first == null) {
        (second._1,"-",second._2)
      }else if(second == null){
        (first._1,first._2,"-")
      }else{
        (first._1,first._2,second._2)
      }
    }).print()
  }
}
```

Java代码

```
public class JavaOuterJoinDataSetJoinApp {
    public static void main(String[] args) throws Exception{
        ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
        //TODO 左外连接
        leftOuterJoin(env);
        System.out.println("--------左外连接和右外连接的分界线--------------------
");
        //TODO 右外连接
        rightOuterJoin(env);
        System.out.println("--------右外连接和全外连接的分界线--------------------
");
        //TODO 全外连接
        fullOuterJoin(env);
    }
    //TODO 左外连接
    public static void  leftOuterJoin(ExecutionEnvironment env) throws Exception{
        List<Tuple2<Integer, String>> list1 = new ArrayList<Tuple2<Integer,
String>>();
        list1.add( new Tuple2<Integer, String>(1,"pk哥"));
        list1.add( new Tuple2<Integer, String>(2,"J哥"));
        list1.add( new Tuple2<Integer, String>(3,"小队长"));
        list1.add( new Tuple2<Integer, String>(4,"猪头呼"));

        List<Tuple2<Integer, String>> list2 = new ArrayList<Tuple2<Integer,
String>>();
        list2.add( new Tuple2<Integer, String>(1,"北京"));
        list2.add( new Tuple2<Integer, String>(2,"上海"));
        list2.add( new Tuple2<Integer, String>(3,"广东"));
        list2.add( new Tuple2<Integer, String>(5,"深圳"));
```

```
        DataSource<Tuple2<Integer, String>> data1 = env.fromCollection( list1 );
        DataSource<Tuple2<Integer, String>> data2 = env.fromCollection( list2 );

        data1.leftOuterJoin( data2 )
                .where( 0 )
                .equalTo( 0 )
                .with( new MyleftFunction() )
                .print();
    }
    public static void rightOuterJoin(ExecutionEnvironment env) throws Exception{
        List<Tuple2<Integer, String>> list1 = new ArrayList<Tuple2<Integer,
String>>();
        list1.add( new Tuple2<Integer, String>(1,"pk哥"));
        list1.add( new Tuple2<Integer, String>(2,"J哥"));
        list1.add( new Tuple2<Integer, String>(3,"小队长"));
        list1.add( new Tuple2<Integer, String>(4,"猪头呼"));

        List<Tuple2<Integer, String>> list2 = new ArrayList<Tuple2<Integer,
String>>();
        list2.add( new Tuple2<Integer, String>(1,"北京"));
        list2.add( new Tuple2<Integer, String>(2,"上海"));
        list2.add( new Tuple2<Integer, String>(3,"广东"));
        list2.add( new Tuple2<Integer, String>(5,"深圳"));

        DataSource<Tuple2<Integer, String>> data1 = env.fromCollection( list1 );
        DataSource<Tuple2<Integer, String>> data2 = env.fromCollection( list2 );

        data1.rightOuterJoin( data2 )
                .where( 0 )
                .equalTo( 0 )
                .with( new MyrightFunction() )
                .print();
    }
    public static void fullOuterJoin(ExecutionEnvironment env) throws Exception {
        List<Tuple2<Integer, String>> list1 = new ArrayList<Tuple2<Integer,
String>>();
        list1.add( new Tuple2<Integer, String>(1,"pk哥"));
        list1.add( new Tuple2<Integer, String>(2,"J哥"));
        list1.add( new Tuple2<Integer, String>(3,"小队长"));
        list1.add( new Tuple2<Integer, String>(4,"猪头呼"));

        List<Tuple2<Integer, String>> list2 = new ArrayList<Tuple2<Integer,
String>>();
        list2.add( new Tuple2<Integer, String>(1,"北京"));
        list2.add( new Tuple2<Integer, String>(2,"上海"));
        list2.add( new Tuple2<Integer, String>(3,"广东"));
        list2.add( new Tuple2<Integer, String>(5,"深圳"));

        DataSource<Tuple2<Integer, String>> data1 = env.fromCollection( list1 );
        DataSource<Tuple2<Integer, String>> data2 = env.fromCollection( list2 );

        data1.fullOuterJoin( data2 )
                .where( 0 )
                .equalTo( 0 )
                .with( new MyfullFunction() )
                .print();
    }
}
```

```
//TODO 左外连接接口
class MyleftFunction implements
JoinFunction<Tuple2<Integer,String>,Tuple2<Integer,String>,
Tuple3<Integer,String,String>>{

    public Tuple3<Integer, String, String> join(Tuple2<Integer, String> first,
Tuple2<Integer, String> second) throws Exception {
        if(second == null ){
            return new Tuple3<Integer, String, String>(first.f0,first.f1,"-");
        }else {
            return new Tuple3<Integer, String, String>
(first.f0,first.f1,second.f1);
        }
    }
}
//TODO 右外连接接口
class MyrightFunction implements
JoinFunction<Tuple2<Integer,String>,Tuple2<Integer,String>,
Tuple3<Integer,String,String>>{

    public Tuple3<Integer, String, String> join(Tuple2<Integer, String> first,
Tuple2<Integer, String> second) throws Exception {
        if(first == null ){
            return new Tuple3<Integer, String, String>(second.f0,"-",second.f1);
        }else {
            return new Tuple3<Integer, String, String>
(first.f0,first.f1,second.f1);
        }
    }
}
//TODO 全外连接接口
class MyfullFunction implements
JoinFunction<Tuple2<Integer,String>,Tuple2<Integer,String>,
Tuple3<Integer,String,String>>{

    public Tuple3<Integer, String, String> join(Tuple2<Integer, String> first,
Tuple2<Integer, String> second) throws Exception {
        if(first == null ){
            return new Tuple3<Integer, String, String>(second.f0,"-",second.f1);
        }else if (second == null){
            return new Tuple3<Integer, String, String>(first.f0,first.f1,"-");
        }
        else {
            return new Tuple3<Integer, String, String>
(first.f0,first.f1,second.f1);
        }
    }
}
```

DataSet的常见的Sink操作

```
writeAsText()
writeAsCsv(...)
print()
write()
output()
```

DataSet的高级编程

1.计数器；

```
Flink计数器开发的三部曲，（1）定义注册器
                        （2）注册计数器
                        （3）获取计数器
```

Scala代码

```scala
/**
  * 计数器编程 三部曲
  *  1.定义计数器
  *  2.注册计数器
  *  3.获取计数器
  */
object CounterApp {
  def main(args: Array[String]): Unit = {
    val env = ExecutionEnvironment.getExecutionEnvironment
    val data = env.fromElements("hadoop","spark","hive","kafka")
    val info = data.map(new MyRichMapFunction)
    //TODO 写入文件才能执行
    info.writeAsText("Flink/huan/output1",WriteMode.OVERWRITE)
    val result = env.execute("CounterApp")
    val num = result.getAccumulatorResult[Long]("ele-count-scala")
    println("num:"+num)
  }
}
//TODO 多看源码
class MyRichMapFunction extends RichMapFunction[String,String](){
  //TODO 定义计数器
  val count = new LongCounter()
  override def open(parameters: Configuration): Unit = {
    //TODO 拿到计数器 看addAccumulator()源码
    getRuntimeContext.addAccumulator("ele-count-scala",count)
  }
  //TODO 累加计数器
  override def map(value: String): String = {
    count.add(1)
    value
  }
}
```

Java代码

```java
public class JavaCounterApp {
    public static void main(String[] args) throws Exception {
        final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
        DataSource<String> source = env.fromElements( "hadoop", "spark", "hive",
"kafka" );
        MapOperator<String, String> sink = source.map( new MyrichMapFunction() );
        sink.writeAsText( "Flink/huan/output1", FileSystem.WriteMode.OVERWRITE );
        //TODO 提交完之后获取计数器名字
        JobExecutionResult executionResult = env.execute( "JavaCounterApp" );
        Object num = executionResult.getAccumulatorResult( "ele-counter-java" );
```

```
        System.out.println("num:"+num);
    }
}
class MyrichMapFunction extends RichMapFunction<String, String> {
    //TODO 定义计数器
    LongCounter counter = new LongCounter();
    @Override
    public void open(Configuration parameters) throws Exception {
        //TODO 获取计数器
        getRuntimeContext().addAccumulator( "ele-counter-java", counter );
    }
    public String map(String value) throws Exception {
        //TODO 累加计数器
        counter.add( 1 );
        return value;
    }
}
```

2.分布式缓存功能

Scala代码

```
/**
  * 1.注册一个本地/HDSF文件
  * 2.在open方法中获取分布式缓存的内容即可
  */
object DistributedCacheApp {
  def main(args: Array[String]): Unit = {
    val env = ExecutionEnvironment.getExecutionEnvironment
    val filepath = "Flink/data/dahuan.txt"
    //TODO 注册一个本地/HDSF文件
    env.registerCachedFile(filepath, "huan-scala-db")
    val data = env.fromElements("hadoop", "spark", "hive", "kafka")
    data.map(new RichMapFunction[String, String] {
      //TODO 通过RuntimeContext和DistributedCache访问缓存的文件
      override def open(parameters: Configuration): Unit = {
        val file = getRuntimeContext.getDistributedCache.getFile("huan-scala-db")
        //TODO 此时是Java的集合
        val list:util.List[String] = FileUtils.readLines(file)
        import scala.collection.JavaConverters._
        for (ele <- list.asScala) { //TODO Scala的集合
          println(ele)
        }
      }
      override def map(value: String): String = {
        value
      }
    }).print()
  }
}
```

Java代码

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

```
        //TODO 注册缓冲区文件
        String filepath = "Flink/data/dahuan.txt";
        env.registerCachedFile( filepath,"huan-java-db" );
        DataSource<String> data = env.fromElements( "hadoop", "spark", "kafka",
"storm" );
        data.map( new MyMapper()).print();
    }
}
class MyMapper extends RichMapFunction<String,String >{
    @Override
    public void open(Configuration parameters) throws Exception {
        //TODO 获取缓冲区文件
        File file = getRuntimeContext().getDistributedCache().getFile( "huan-
java-db" );
        List<String> lists = FileUtils.readLines( file );
        for (String list : lists) {
            System.out.println(list);
        }
    }
    public String map(String value) throws Exception {
        return value;
    }
}
```

广播变量

设置广播变量

在某个需要用到该广播变量的算子后调用withBroadcastSet(var1, var2)进行设置，var1为需要广播变量的变量名，var2是自定义变量名，为String类型。注意，被广播的变量只能为DataSet类型，不能为List、Int、String等类型。

获取广播变量

创建该算子对应的富函数类，例如map函数的富函数类是RichMapFunction，该类有两个构造参数，第一个参数为算子输入数据类型，第二个参数为算子输出数据类型。首先创建一个Traversable[]*接口用于接收广播变量并初始化为空，接收类型与算子输入数据类型相对应；然后重写open函数，通过getRuntimeContext.getBroadcastVariable[]*(var)获取到广播变量，var即为设置广播变量时的自定义变量名，类型为String，open函数在算子生命周期的初始化阶段便会调用；最后在map方法中对获取到的广播变量进行访问及其它操作。

注意：只有在某个Operator中使用到不属于该Operator的DataSet时才需要广播变量，在iterate内部可以将某个DataSet直接作为起始节点，不需要使用广播变量

Scala代码

```
object DataSetBroadcastApp {
  //TODO 广播变量
  def main(args: Array[String]): Unit = {
    val env = ExecutionEnvironment.getExecutionEnvironment
    broadcast(env)
  }
  def broadcast(env:ExecutionEnvironment): Unit ={
    //TODO 创建两个元素
    val ds1 = env.fromElements("1", "2", "3", "4", "5")
    val ds2 = env.fromElements("a", "b", "c", "d", "e")
    //TODO 广播数据集
    ds1.map(new RichMapFunction[String,(String,String)] {
      //TODO 定义一个私有的另一个数据集
     private var ds2 : Traversable[String] = null
```

```
      //TODO 因为是Java的集合所以要用隐式转换
      import scala.collection.JavaConverters._
      //TODO 参见官网https://ci.apache.org/projects/flink/flink-docs-release-
1.10/dev/batch/#broadcast-variables
      override def open(parameters: Configuration): Unit = {
        ds2 = getRuntimeContext.getBroadcastVariable[String]("broadcast").asScala
      }
      override def map(value: String): (String, String) = {
        //TODO 定义一个结果
        var result = " "
        for(broadVariable <- ds2){
          result = result + broadVariable + " "
        }
        (value,result)
      }
    }).withBroadcastSet(ds2,"broadcast").print()
  }
}
```

Java代码

```
public class JavaDataSetBroadcastApp {
    //TODO Java实现广播变量
    public static void main(String[] args) throws Exception{
        final  ExecutionEnvironment env =
 ExecutionEnvironment.getExecutionEnvironment();
        broadcast(env);
    }
    public static void broadcast(ExecutionEnvironment env) throws Exception {
        DataSet<Integer> toBroadcast = env.fromElements(1, 2, 3);
        DataSet<String> data = env.fromElements("a", "b");
        MapOperator<String, String> broadcastSet = data.map( new
MyMapperFunction() ).withBroadcastSet( toBroadcast, "huan" );
       broadcastSet.printToErr();
    }
}
class MyMapperFunction extends RichMapFunction<String, String> {
    //TODO 定义一个私有集合
    private List list=  new ArrayList();
    @Override
    public void open(Configuration parameters) throws Exception {
        //TODO 作为集合访问广播数据集.所以要定义集合，用集合添加广播变量，因为返回值是集
合，所以要用集合去添加广播变量
        Collection<Integer> broadcastSet =
getRuntimeContext().getBroadcastVariable( "huan" );
        list.add( broadcastSet );
    }
    @Override
    public String map(String value) throws Exception {
        //TODO value的值就是data   list的值就是toBroadcast
        return value+ " : " +list;
    }
}
```

DataStream

DataStream的数据源（Source）

```
Flink comes with a number of pre-implemented source functions, but you can always write your own custom sources by implementing the SourceFunction for non-parallel sources, or by implementing the ParallelSourceFunction interface or extending the RichParallelSourceFunction for parallel sources.
翻译：
Flink附带了许多预先实现的源函数，但是您始终可以通过为非并行源实现SourceFunction或通过实现ParallelSourceFunction接口或为并行源扩展RichParallelSourceFunction来编写自己的自定义源。
```

所有代码都是一个套路，这里只放一个

Scala版本

```
object DataSteamSourceFunction {
    //TODO 自定义数据源
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    sourceFunction(env)
    env.execute("DataSteamSourceFunction")
  }
  def sourceFunction(env:StreamExecutionEnvironment): Unit ={
    //TODO 自定义数据源的特有类
    val data = env.addSource(new MySource).setParallelism(1)
    data.print().setParallelism(1)
  }
}
//TODO 自定义SourceFunction (看源码一目了然)
class MySource extends SourceFunction[Long]{
  //TODO 初始值为1
  var count = 1L
  var isRunning = true
  //TODO 输出初始值，每次加个1
  override def run(ctx: SourceFunction.SourceContext[Long]): Unit = {
    while (isRunning){
      ctx.collect(count)
      count += 1
      Thread.sleep(1000)
    }
  }
  override def cancel(): Unit = {
    isRunning = false
  }
}
```

Java版本

```
public class JavaDataSteamSourceFunction {
    public static void main(String[] args) throws Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        sourceFunction( env );
        env.execute( "JavaDataSteamSourceFunction" );
    }
```

```
    public static void sourceFunction(StreamExecutionEnvironment env) throws
Exception{
        DataStreamSource<Long> data = env.addSource( new MySourceFunction() );
        data.print( ).setParallelism( 1 );
    }
}
class MySourceFunction implements SourceFunction<Long>{
    private long count = 1;
    private Boolean isRunning = true;

    public void run(SourceContext<Long> ctx) throws Exception {
        while (isRunning){
            ctx.collect( count );
            count += 1;
            Thread.sleep( 1000 );
        }
    }
    public void cancel() {
        isRunning = false;
    }
}
```

DataStream的Transformationn操作的重点

split && select

Scala代码

```
object DataStreamSplitSelectApp {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val data = env.addSource(new MySource)
    val splits =  data.split(new OutputSelector[Long] {
      //TODO value是输入进来的值
      override def select(value: Long): lang.Iterable[String] = {
        val list = new util.ArrayList[String]()
        if(value % 2 ==0){
          list.add("even")
        }else{
          list.add("odd")
        }
        list
      }
    })
    //TODO 看select源码，可以有一个也可以有多个
    splits.select("even","odd").print().setParallelism(1)
    env.execute("DataStreamSplitSelectApp")
  }
}
```

Java代码

```
public class JavaDataStreamSplitSelectApp {
    public static void main(String[] args) throws Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<Long> data = env.addSource( new MySourceFunction() );
```

```
        //TODO 必须要传入OutputSelector
        SplitStream<Long> splits = data.split( new OutputSelector<Long>() {
            //TODO value是输入进来的值
            public Iterable<String> select(Long value) {
                List<String> list = new ArrayList<String>();
                if (value % 2 == 0) {
                    list.add( "even" );
                }
                if (value % 2 == 1) {
                    list.add( "odd" );
                }
                return list;
            }
        } );
        splits.select( "even" ).print( ).setParallelism( 1 );
        env.execute("JavaDataStreamSplitSelectApp");
    }
}
```

DataStream的落地（Sink）

```
总结：1.继承RichSinkFunction<T> T就是你相要写入对象的类型
      2.重写方法：
            open/close方法，
            invoke : 每条记录执行一次
```

附上Java代码：db.properties工具类

```
jdbc.driver = com.mysql.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/flink
jdbc.username=root
jdbc.password=123
```

JDBC工具类

```
public class JDBCUtils {
    //TODO 配置环境
    private static String driver = null;
    private static String url = null;
    private static String user = null;
    private static String password = null;
    //TODO 连接属性
    private static Properties prop = new Properties();
    //TODO 线程池
    private static ThreadLocal<Connection> t1 = new ThreadLocal<Connection>();
    static {
        try {
            //TODO 采用类加载方式读取文件
            InputStream input =
JDBCUtils.class.getClassLoader().getResourceAsStream( "db.properties" );
            prop.load( input );
            driver = prop.getProperty( "jdbc.driver" );
            url = prop.getProperty( "jdbc.url" );
            user = prop.getProperty( "jdbc.username" );
            password = prop.getProperty( "jdbc.password" );
            Class.forName( driver );
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    //TODO 获取连接的方法
    public static Connection getConn() throws Exception{
        //TODO 获取线程
        Connection conn = t1.get();
        if (conn == null){
            conn= DriverManager.getConnection( url,user,password );
            t1.set( conn );
        }
        return conn;
    }
    //TODO 关闭连接
    public static void closeConn() throws  Exception{
        //TODO 获取线程
        Connection conn = t1.get();
        if (conn != null ){
            conn.close();
        }
        //TODO t1设置为空 ，哪怕连接对象不可用了， 但是对象还在，还能给一个新的连接
        t1.set( null );
    }
}
```

定义一个学生类

```
public class Student {
    private int id;
    private String name;
    private int age;
    public Student() { }
    public Student(int id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    @Override
```

```
    public String toString() {
        return "Student[" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                ']';
    }
}
```

定义一个Sink类 继承RichSinkFunction

```
public class SinkToMySQL extends RichSinkFunction<Student> {
    Connection conn;
    PreparedStatement ps;
    //TODO 开始执行
    @Override
    public void open(Configuration parameters) throws Exception {
         conn = JDBCUtils.getConn();
         String sql = "insert into student(id,name,age) values (?,?,?)";
         ps = conn.prepareStatement( sql );
         System.out.println("open");
    }
    //TODO 每条记录插入时调用一次
    public void invoke(Student value, Context context) throws Exception {
        System.out.println("'''''''invoke'''''''");
        ps.setInt(1,value.getId());
        ps.setString( 2,value.getName());
        ps.setInt( 3,value.getAge());
       ps.executeUpdate();
    }
    //TODO 关闭资源
    @Override
    public void close() throws Exception {
        JDBCUtils.closeConn();
        if(ps != null){
            ps.close();
        }
    }
}
```

定义一个map类继承MapFunction

```
public class MyMapFunction implements MapFunction<String,Student> {
    public Student map(String value) throws Exception {
        //TODO 所有的数据按照逗号切分
        String[] splits = value.split( "," );
        //TODO 获取Student对象
        Student stu = new Student();
        //TODO 将student对象的数据按照split实现
        stu.setId( Integer.parseInt( splits[0] ) );
        stu.setName( splits[1] );
        stu.setAge( Integer.parseInt( splits[2] ) );
        //TODO 返回student对象类型
        return stu;
    }
}
```

最后测试

```
public class JavaSinkMysql {
    public static void main(String[] args) throws Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //TODO 定义端口
        DataStreamSource<String> source = env.socketTextStream( "localhost", 7777
);
        //TODO 定义函数
        SingleOutputStreamOperator<Student> studentStream = source.map( new
MyMapFunction());
        //TODO 定义Sink
        studentStream.addSink( new SinkToMySQL());
        //TODO 提交
        env.execute( "JavaSinkMysql");
    }
}
```

总结，充分利用了Java的类与对象的特点，将各种类进行转换，达到了简便的效果

Flink的Time及Windows操作

```
Processing time : 处理时间
Event time : 事件时间
Ingestion time : 接入时间

幂等性：就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。
```

Windows操作

滚动窗口：TumblingWindows （不重叠） Scala代码

```
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val data = env.socketTextStream("localhost",9999)
    data.flatMap(_.split(","))
      .map((_,1))
      .keyBy(0)
      //TODO 添加了timeWindow操作
      .timeWindow(Time.seconds(5))
      .sum(1)
      .print()
      .setParallelism(1)
    env.execute("TumblingWindows")
  }
}
```

Java代码

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

```
        DataStreamSource<String> data = env.socketTextStream( "localhost", 9999
);
        data.flatMap( (FlatMapFunction<String, Tuple2<String, Integer>>) (value,
out) -> {
            String[] splits = value.split( "," );
            for (String split : splits) {
                out.collect( new Tuple2<String, Integer>( split, 1 ) );
            }
        } ).keyBy( 0 )
            //TODO 操作
                .timeWindow( Time.seconds( 5 ) )
                .sum( 1 )
                .print()
                .setParallelism( 1 );
        env.execute( "JavaTumblingWindows" );
    }
}
```

滑动窗口：SlidingWindows（会重叠） Scala代码

```
object SlidingWindows {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    import org.apache.flink.api.scala._
    val data = env.socketTextStream("localhost",9999)
    data.flatMap(_.split(","))
      .map((_,1))
      .keyBy(0)
      //TODO 滑动窗口会重叠，滚动窗口不会重叠
      .timeWindow(Time.seconds(10),Time.seconds(5))
      .sum(1)
      .print()
      .setParallelism(1)
    env.execute("TumblingWindows")
  }
}
```

Java代码

```
public class JavaSlidingWindows {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> data = env.socketTextStream( "localhost", 9999
);
        data.flatMap( new FlatMapFunction<String, Tuple2<String, Integer>>() {
            public void flatMap(String value, Collector<Tuple2<String, Integer>>
out) throws Exception {
                String[] splits = value.split( "," );
                for (String split : splits) {
                    out.collect( new Tuple2<String, Integer>(split,1) );
                }
            }
        } ).keyBy( 0 )
                .timeWindow( Time.seconds( 10 ),Time.seconds( 5 ))
                .sum( 1 )
```

```
                .print( )
                .setParallelism( 1 );
        env.execute( "JavaSlidingWindows" );
    }
}
```

connectors : 连接器

Apache Kafka （Source） 以下均为Scala代码

```
object KafkaConnectorConsumerApp {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // TODO https://ci.apache.org/projects/flink/flink-docs-release-
1.10/dev/connectors/kafka.html
    // TODO 直接去官网
    val topic = "first"
    val properties = new Properties()
    properties.setProperty("bootstrap.servers", "linux-1:9092")
    // only required for Kafka 0.8
    // properties.setProperty("zookeeper.connect", "linux-1:2181")
    properties.setProperty("group.id", "test")
    val data = env.addSource(new FlinkKafkaConsumer011[String](topic, new
SimpleStringSchema(), properties))
    data.print().setParallelism(1)
    env.execute("KafkaConnectorConsumerApp")
  }
}
```

Apache Kafka （Sink）

```
object KafkaConnectorProducerApp {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置检查点模式（仅一次与至少一次）checkpointingMode检查点模式。
    env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)
    //TODO 状态检查点之间的时间间隔（以毫秒为单位）。
    env.enableCheckpointing(4000)
    //TODO 设置检查点在被丢弃之前可能花费的最长时间。
    env.getCheckpointConfig.setCheckpointTimeout(10000)
    //TODO 并发检查点尝试的最大次数。
    env.getCheckpointConfig.setMaxConcurrentCheckpoints(1)
    //TODO 通过socket中接收数据，用过Flink,将数据Sink到kafka
    val data = env.socketTextStream("localhost", 9999)
    // TODO https://ci.apache.org/projects/flink/flink-docs-release-
1.10/dev/connectors/kafka.html
    // TODO 直接去官网
    val topic = "first"
    val properties = new Properties()
    properties.setProperty("bootstrap.servers", "linux-1:9092")
    val sink =  new FlinkKafkaProducer011[String](topic,
      new KeyedSerializationSchemaWrapper[String](new SimpleStringSchema()),
      properties)
    data.addSink(sink)
    env.execute("KafkaConnectorProducerApp")
  }
```

```
}
```

开发（尚硅谷 教学）-> （高级篇）

快速入门一些基本操作

Source （数据源 ）

```
object SourceTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(1)
    //TODO 从自定义的集合中读取数据
    val data = env.fromCollection(List(
      SensorReading("sensor_1", 1547718199, 35.80018327300259),
      SensorReading("sensor_6", 1547718201, 15.402984393403084),
      SensorReading("sensor_7", 1547718202, 6.720945201171228),
      SensorReading("sensor_10", 1547718205, 38.101067604893444)
    ))
    data.print()
    //TODO 从自定义文件读取数据
    val value = env.readTextFile("Flink/source/data,txt")
    value.print()
    //TODO 自定义元素
    val source = env.fromElements(1,2.4,"huan")
    source.print()
    env.execute("SourceTest")
  }
}
//TODO 温度传感器，样例类 id 时间戳，温度值
case class SensorReading(id: String, timestamp: Long, temprature: Double)
```

除了以上代码以外，Flink的数据来源还有kafka，一般情况下都用kafka 。

自定义数据源

```
object MySourceSensor {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val data = env.addSource(new MySensor)
    data.print().setParallelism(1)
    env.execute("MySourceSensor")
  }
}
class MySensor extends SourceFunction[SensorReading] {
  //TODO 定义一个flag,表示数据源是否正常运行
  var isRunning = true
  //TODO 正常数据生成
  override def run(ctx: SourceFunction.SourceContext[SensorReading]): Unit = {
    //TODO 初始化一个随机数发生器
    val rand = new Random(5)
    //TODO 初始化温度传感器
    var curTemp = 1 .to (10).map(
      i => ("sensor_" + i, 60 + rand.nextGaussian() * 10 )
    )
```

```
    //TODO 用无限循环产生数据流
    while (isRunning){
      //TODO 在上一个温度的基础上更新温度值
      curTemp = curTemp.map(
        t => (t._1,t._2 + rand.nextGaussian())
      )
      //TODO 获取当前时间戳
      val curTime = System.currentTimeMillis()

      curTemp.foreach(
        t => ctx.collect(SensorReading(t._1,curTime,t._2))
      )
      //TODO 设置时间间隔
      Thread.sleep(5000)
    }
  }
  override def cancel(): Unit = {
    isRunning = false
  }
}
```

Transform算子操作

基本转换算子

```
object MapFilter {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    val source = "Flink/source/data.txt"
    val data = env.readTextFile(source)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading

 SensorReading(dataArray(0).trim,dataArray(1).trim.toLong,dataArray(2).trim.toDou
ble)
    })
    //TODO 是一步一步去输出，而不是一下子输出
    dataStream.keyBy("id").sum("temprature").print()
    env.execute("MapFilter")
  }
}
```

聚合算子

```
object Reduce {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    val source = "Flink/source/data.txt"
    val data = env.readTextFile(source)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
```

```
      //TODO 返回一个SensorReading

 SensorReading(dataArray(0).trim,dataArray(1).trim.toLong,dataArray(2).trim.toDou
ble)
    })
    //TODO 是一步一步去输出，而不是一下子输出
    dataStream.keyBy("id")
      //TODO 输出当前最新温度值 + 10 ，而时间戳是上一个数据的时间 + 1
      .reduce((x,y) => SensorReading(x.id,x.timestamp+1,y.temprature+10)).print()
    env.execute("MapFilter")
  }
}
```

多流转换算子

（SplitSelect）-> 分流

```
object SplitSelect {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    val source = "Flink/source/data.txt"
    val data = env.readTextFile(source)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
    val splitStream = dataStream.split(t => {
      //TODO 将大于30的温度值赋字符串为higt,否则low
      if (t.temprature > 30) Seq("higt") else Seq("low")
    })
    //TODO 分类好的字符串写上相应的名字
    val higt = splitStream.select("higt")
    val low = splitStream.select("low")
    val all = splitStream.select("higt", "low")
    //TODO 输出
    higt.print("higt")
    low.print("low")
    all.print("all")
    env.execute("SplitSelect")
  }
}
```

（ConnectCoMap）-> 合并流 （只能合并两个）

```
object ConnectCoMap {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    val source = "Flink/source/data.txt"
    val data = env.readTextFile(source)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
```

```
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
    val splitStream = dataStream.split(t => {
      //TODO 将大于30的温度值赋字符串为higt,否则low
      if (t.temprature > 30) Seq("higt") else Seq("low")
    })
    //TODO 分类好的字符串写上相应的名字
    val higt = splitStream.select("higt")
    val low = splitStream.select("low")
    val all = splitStream.select("higt", "low")
    //TODO 当温度值大于30 的时候，合并数值的 id和温度值
    val warning: DataStream[(String, Double)] = higt.map(s => (s.id,
s.temprature))
    //TODO  warning 和 low 连接
     val connectedStream: ConnectedStreams[(String, Double), SensorReading] =
warning.connect(low)
     /**
       * 连接之后做map操作，map操作时，此时有两个流，将两个流分别做成它们相应的需求
       * 当warning 需求时 此时流上方是温度值大于30 ，则输出值的id和温度值 （大于30的温度
值）以及自定义的值
       * 当作low 需求时 ，此时流上方是温度值小于30,则输出id和 自定义的值
       *
       */
    val coMapStream = connectedStream.map(
      /**
        * 此时做操作的时候 ，按照上方的先后顺序来，此时第一个是元组，
        * 第二个是样例类
        * val connectedStream: ConnectedStreams[(String, Double), SensorReading]
= warning.connect(low)
        * [(String, Double), SensorReading]
        * 第一个是元组，(String, Double)
        * 第二个是样例类  SensorReading
        */
      warningdata => (warningdata._1, warningdata._2, "warning"),
      lowData => (lowData.id, lowData.temprature,"healthy")
    )
    //TODO 输出
    coMapStream.print()
    env.execute("ConnectCoMap")
  }
}
```

（Union）-> 合并流 （多个流）

```
object Union {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设行度
    env.setParallelism(1)
    val source = env.readTextFile("Flink/source/data.txt")
    //TODO 返回一个SensorReading
    val dataStream = source.map(data => {
      val dataArray = data.split(",")
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
```

```
    })
    val splitStream = dataStream.split(s => {
      //TODO 将大于30的温度值赋字符串为higt,否则low
      if (s.temprature > 30) {
        Seq("higt")
      } else {
        Seq("low")
      }
    })
    //TODO 分类好的字符串写上相应的名字
    val higt = splitStream.select("higt")
    val low = splitStream.select("low")
    val all = splitStream.select("higt","low")
    //TODO 多流合并
    higt.union(all).union(low).print()
    env.execute("Union")
  }
}
```

自定义函数（UDF）

```
class MyFilter extends FilterFunction[SensorReading]{
  override def filter(value: SensorReading): Boolean = {
    //TODO startsWith 以。。。开始
    value.id.startsWith("sensor_1")
    //TODO contains -> 包含
    //value.id.contains("sensor_6")
  }
}
object MyUDF {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    val source = "Flink/source/data.txt"
    val data = env.readTextFile(source)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading

 SensorReading(dataArray(0).trim,dataArray(1).trim.toLong,dataArray(2).trim.toDou
ble)
    })
    //TODO 是一步一步去输出，而不是一下子输出
    dataStream.filter(new MyFilter).print()
    env.execute("MyUDF")
  }
}
```

Sink（数据落地）

kafka

```
object KafkaSinkTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
    env.setParallelism(1)
    val source = "Flink/source/data.txt"
    val data = env.readTextFile(source)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble).toString  //TODO 转成String 方便序列化输出
    })
    val properties = new Properties()
    properties.setProperty("bootstrap.servers", "localhost:9092")
    // only required for Kafka 0.11
    properties.setProperty("zookeeper.connect", "localhost:2181")
    properties.setProperty("group.id", "test")
    val topic = "first"
    dataStream.addSink(new FlinkKafkaProducer011[String](topic,new
SimpleStringSchema() ,properties))
    dataStream.print()
    env.execute("KafkaSinkTest")
  }
}
```

Redis

```
object RedisSinkTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(1)
    val source = "Flink/source/data.txt"
    val data = env.readTextFile(source)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
    //TODO Redis的固定格式
    val conf = new FlinkJedisPoolConfig.Builder()
        .setHost("localhost")
        .setPort(6379)
        .build()
    //TODO 固定用法 RedisSink ,看源码
    dataStream.addSink(new RedisSink(conf,new MyRedisMapper))
    env.execute("RedisSinkTest")
  }
}
//TODO 继承RedisMapper
class MyRedisMapper extends RedisMapper[SensorReading]{
  //TODO 定义保存数据到redis的命令
  override def getCommandDescription: RedisCommandDescription = {
    //TODO 把传感器的温度值和id保存到哈希表 HSET field value
    //TODO 看RedisCommandDescription的源码
    new RedisCommandDescription(RedisCommand.HSET,"sensor_temprature" )
  }
  //TODO 定义保存到Redis的key
  override def getKeyFromData(data: SensorReading): String = {
    data.id.toString
```

```
  }
  //TODO 定义保存到Redis的value
  override def getValueFromData(data: SensorReading): String = {
    data.temprature.toString
  }
}
```

Elasticsearch

```
object ESSinkTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(1)
    val source = "Flink/source/data.txt"
    val data = env.readTextFile(source)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
    val httpsHosts = new util.ArrayList[HttpHost]()
    httpsHosts.add(new HttpHost("localhost",9200))
    //TODO 创建一个es的Sink 的builder
    val esSinkBuilder = new ElasticsearchSink.Builder[SensorReading](
      httpsHosts,
      new ElasticsearchSinkFunction[SensorReading] {
        /**
          *
          * @param t  -> 当前传进来的数据
          * @param runtimeContext -> 上下文对象
          * @param requestIndexer -> 发送操作请求
          */
        override def process(t: SensorReading, runtimeContext: RuntimeContext,
requestIndexer: RequestIndexer): Unit ={
          println("saving data:" + t)
          //TODO 包装成一个Map或者JsonObject
          val json = new util.HashMap[String,String]()
          json.put("sensor_id",t.id)
          json.put("tempreature",t.temprature.toString)
          json.put("ts",t.timestamp.toString)
          //TODO 创建indexrequest 准备发送数据
            val indexRequest = Requests.indexRequest()
              .index("sensor")
              .`type`("readingdata")
              .source(json)
          //TODO 利用index发送请求，写入数据
          requestIndexer.add(indexRequest)
          println("data saved")
        }
      }
    )
    dataStream.addSink(esSinkBuilder.build())
    env.execute("ESSinkTest")
  }
}
```

JDBC

```
object JDBCSinkTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(1)
    val source = "Flink/source/data.txt"
    val data = env.readTextFile(source)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
    dataStream.addSink(new MyJDBCSink)
    env.execute("JDBCSinkTest")
  }
}
//TODO 继承RichSinkFunction （具有声明周期）
class MyJDBCSink extends RichSinkFunction[SensorReading]{
  //TODO 定义初始化连接 sql , 预编译器
  val url = "jdbc:mysql://localhost:3306/test"
  val user = "root"
  val password = "123"

  var conn : Connection = _
  var insertStmt : PreparedStatement = _
  var updateStmt : PreparedStatement = _
  //TODO 初始化  创建连接和预编译器
  override def open(parameters: Configuration): Unit = {
    conn = DriverManager.getConnection(url,user,password)
    val sql1 = "insert into hh(name,temp) value(?,?)"
    insertStmt =  conn.prepareStatement(sql1)
    var sql2 = "update hh set temp = ? where name = ? "
    updateStmt = conn.prepareStatement(sql2)
  }
  //TODO 调用连接，执行sql
  override def invoke(value: SensorReading, context: SinkFunction.Context[_]):
Unit = {
    //TODO 执行更新语句
    updateStmt.setDouble(1,value.temprature)
    updateStmt.setString(2,value.id)
    updateStmt.execute()
    //TODO 如果update没有查到数据，执行插入
    if(updateStmt.getUpdateCount ==0 ){
      insertStmt.setString(1,value.id)
      insertStmt.setDouble(2,value.temprature)
      insertStmt.execute()
    }
  }
  //TODO 关闭资源，做清理动作 ，三台全部关闭
  override def close(): Unit = {
    insertStmt.close()
    updateStmt.close()
    conn.close()
  }
}
```

WindowAPI

处理无界数据流要把无界数据流变成有界数据流，窗口就是将无限的流切割为有限流的一种方式，它会将流数据分发到有限大小的桶（bucket）中进行分析

窗口函数

```
一般分为两类，
增量聚合函数
    每条数据到来就进行计算，保持一个简单的状态
    ReduceFunction,AggragateFunction
全窗口函数
    先把窗口所有数据收集起来，等到计算的时候会遍历所有数据
    ProcessWindowFunction
```

其他可选API

```
trigger() -> 触发器
    定义Window什么时候关闭，触发计算并输出结果
evitor() -> 移除器
    定义移除某些数据的逻辑
allowedLateness() -> 允许处理迟到的数据
sideOutputLateData() -> 将迟到的数据放入侧输出流
getSideOutput() -> 获取侧输出流
```

Watermark

```
特点：是一个特殊的数据记录
      必须单调递增，以确保任务事件时间时钟向前推进，而不是在后退
      要与数据的时间戳相关
```

引入

```
Event Time的使用一定要指定数据源中的时间戳
对于排好序的数据，只需要指定时间戳就够了，不需要延迟触发
```

MyAssigner 可以有两种类型都继承与TimestampAssigner

TimestampAssigner

- 定义了抽取时间戳，以及生成 watermark 的方法，有两种类型
- AssignerWithPeriodicWatermarks
  - 周期性的生成 watermark：系统会周期性的将 watermark 插入到流中
  - 默认周期是200毫秒，可以使用 ExecutionConfig.setAutoWatermarkInterval() 方法进行设置
  - 升序和前面乱序的处理 BoundedOutOfOrderness，都是基于周期性 watermark 的。
- AssignerWithPunctuatedWatermarks
  - 没有时间周期规律，可打断的生成 watermark

滚动窗口 ——> 小测试

```
object WindowTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    val data = env.socketTextStream("localhost",9999)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
    //TODO 做一个10秒内的最小温度值
    val minTempPerWindowStream: DataStream[(String, Double)] =
dataStream.map(data => (data.id, data.temprature))
    //TODO 以id进行分组
    minTempPerWindowStream.keyBy(x => x._1)
      //TODO 开一个时间窗口 (滚动窗口) 每隔十秒执行一次
      .timeWindow(Time.seconds(10))
      //TODO 用reduce作增量聚合
      .reduce((data1,data2)=> (data1._1,data1._2.min(data2._2)))
      .print(" min Temp")
    dataStream.print("input Stream ")
    env.execute("WindowTest")
  }
}
```

滚动窗口 ——>乱序问题

```
object RollingwindowTimeTest {
```

```
  //TODO 滚动窗口乱序问题
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    //TODO 设置事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //TODO 默认为200毫秒
    env.getConfig.setAutoWatermarkInterval(100L)
    //    val source = "Flink/source/data.txt"
    //    val data = env.readTextFile(source)
    val data = env.socketTextStream("localhost", 9999)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
      //TODO 因为assignAscendingTimestamps是毫秒 ，所以乘以1000变成秒
      // .assignAscendingTimestamps(_.timestamp * 1000)
      //TODO 处理乱序数据的话，有三个方式
      // TODO  经典方式 BoundedOutOfOrdernessTimestampExtractor  延迟一秒钟处理
      .assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.seconds(1)) {
      override def extractTimestamp(element: SensorReading): Long = {
        element.timestamp * 1000
      }
    })
    //TODO 做一个10秒内的最小温度值
    val minTempPerWindowStream: DataStream[(String, Double)] =
dataStream.map(data => (data.id, data.temprature))
    //TODO 以id进行分组
    minTempPerWindowStream.keyBy(x => x._1)
      //TODO 开一个时间窗口 (滚动窗口) 每隔十秒执行一次
      .timeWindow(Time.seconds(10))
      //TODO 用reduce作增量聚合
      .reduce((data1,data2)=> (data1._1,data1._2.min(data2._2)))
      .print(" min Temp")
    dataStream.print("input Stream ")
    env.execute("WindowEventTime")
  }
}
//TODO TimestampAssigner中的两个实现类  -> AssignerWithPeriodicWatermarks和
AssignerWithPunctuatedWatermarks
//TODO 继承 AssignerWithPeriodicWatermarks
class MyAssigner extends AssignerWithPeriodicWatermarks[SensorReading] {

  val bound: Long = 60 * 1000 //TODO 延时为 1 分钟
  var maxTs: Long = Long.MinValue //TODO 观察到的最大时间戳

  override def getCurrentWatermark: Watermark = {
    new Watermark(maxTs - bound)
  }
  override def extractTimestamp(element: SensorReading, previousElementTimestamp:
Long): Long = {
    maxTs = maxTs.max(element.timestamp * 1000)
    element.timestamp * 1000
  }
```

```
}
//TODO 继承AssignerWithPunctuatedWatermarks
class MyAssigners extends AssignerWithPunctuatedWatermarks[SensorReading] {
  override def checkAndGetNextWatermark(lastElement: SensorReading,
extractedTimestamp: Long): Watermark = {
    new Watermark(extractedTimestamp)
  }

  override def extractTimestamp(element: SensorReading, previousElementTimestamp:
Long): Long = {
    element.timestamp * 1000
  }
}
```

滑动窗口 ——>乱序问题

```
object SlidingwindowTest {
  //TODO 滑动窗口乱序问题
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    //TODO 设置事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //TODO 默认为200毫秒
    env.getConfig.setAutoWatermarkInterval(100L)
    //    val source = "Flink/source/data.txt"
    //    val data = env.readTextFile(source)
    val data = env.socketTextStream("localhost", 9999)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
      //TODO 因为assignAscendingTimestamps是毫秒 ，所以乘以1000变成秒
      // .assignAscendingTimestamps(_.timestamp * 1000)
      //TODO 处理乱序数据的话，有三个方式
      // TODO  经典方式 BoundedOutOfOrdernessTimestampExtractor  延迟一秒钟处理
      .assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.seconds(1)) {
      override def extractTimestamp(element: SensorReading): Long = {
        element.timestamp * 1000
      }
    })
    //TODO 做一个15秒内的最小温度值，每隔五秒执行一次
    val minTempPerWindowStream: DataStream[(String, Double)] =
dataStream.map(data => (data.id, data.temprature))
    //TODO 以id进行分组
    minTempPerWindowStream.keyBy(x => x._1)
      //TODO 统计15秒内的最小温度值，隔5s输出一次
      .timeWindow(Time.seconds(15),Time.seconds(5))
      //TODO 用reduce作增量聚合
      .reduce((data1,data2)=> (data1._1,data1._2.min(data2._2)))
      .print(" min Temp")
    dataStream.print("input Stream ")
    env.execute("SlidingwindowTest")
```

```
  }
}
```

ProcessFunction API（底层 API）

```
Flink 提供了 8 个 Process Function：
□ ProcessFunction
□ KeyedProcessFunction
□ CoProcessFunction
□ ProcessJoinFunction
□ BroadcastProcessFunction
□ KeyedBroadcastProcessFunction
□ ProcessWindowFunction
□ ProcessAllWindowFunction
```

KeyedProcessFunction

KeyedProcessFunction 用来操作 KeyedStream。KeyedProcessFunction 会处理流的每一个元素，输出为 0 个、1 个或者多个元素。所有的 Process Function 都继承自RichFunction 接口，所以都有 open()、close()和 getRuntimeContext()等方法。
而KeyedProcessFunction[KEY, IN, OUT]还额外提供了两个方法：
□ processElement(v: IN, ctx: Context, out: Collector[OUT])，流中的每一个元素都会调用这个方法，调用结果将会放在 Collector 数据类型中输出。Context可以访问元素的时间戳，元素的 key，以及 TimerService 时间服务。Context还可以将结果 输出到别的流(side outputs)。
□ onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[OUT])是一个回调函数。当之前注册的定时器触发时调用。参 数 timestamp 为定时器所设定的触发的时间戳。Collector 为输出结果的集合。OnTimerContext 和processElement 的 Context 参数一样，提供了上下文的一些信息，例如定时器触发的时间信息(事件时间或者处理时间)。

TimerService 和 定时器（Timers）

Context 和 OnTimerContext 所持有的 TimerService 对象拥有以下方法：
□ currentProcessingTime(): Long 返回当前处理时间
□ currentWatermark(): Long 返回当前 watermark 的时间戳
□ registerProcessingTimeTimer(timestamp: Long): Unit 会注册当前 key 的 processing time 的定时器。当 processing time 到 达定时时间时，触发 timer。
□ registerEventTimeTimer(timestamp: Long): Unit 会注册当前 key 的 event time 定时器。当水位线大于等于定时器注册的时间 时，触发定时器执行回调函数。
□ deleteProcessingTimeTimer(timestamp: Long): Unit 删除之前注册处理时间定时器。如果没有这个时间戳的定时器，则不执行。
□ deleteEventTimeTimer(timestamp: Long): Unit 删除之前注册的事件时间定时器，如果没有此时间戳的定时器，则不执行。
当定时器 timer 触发时，会执行回调函数 onTimer()。注意定时器 timer 只能在keyed streams 上面使用。

简单测试

```
object ProcessTest {
  //TODO 简单小测试
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
```

```
    //TODO 设置事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //TODO 默认为200毫秒
    env.getConfig.setAutoWatermarkInterval(100L)
    val data = env.socketTextStream("localhost", 9999)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
      .keyBy(_.id)
      .process(new MyProcess)
       .print()
    env.execute("ProcessTest")
  }
}
//TODO 继承这个函数之后要提前做好keyBy工作
class MyProcess extends KeyedProcessFunction[String,SensorReading,String]{
  override def processElement(value: SensorReading, ctx:
KeyedProcessFunction[String, SensorReading, String]#Context, out:
Collector[String]): Unit = {
    ctx.timerService().registerEventTimeTimer(2000L)
  }
}
```

监控温度传感器的温度值，如果温度值在一秒钟之内(processing time)连续上升，则报警。

```
//TODO 监控温度传感器的温度值，如果温度值在一秒钟之内(processing time)连续上升，则报警。
object ProcessFunctionTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    //TODO 设置事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //TODO 默认为200毫秒
    env.getConfig.setAutoWatermarkInterval(100L)
    val data = env.socketTextStream("localhost", 9999)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
      //TODO 因为assignAscendingTimestamps是毫秒 ，所以乘以1000变成秒
      // .assignAscendingTimestamps(_.timestamp * 1000)
      //TODO 处理乱序数据的话，有三个方式
      // TODO  经典方式 BoundedOutOfOrdernessTimestampExtractor  延迟一秒钟处理
      .assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.seconds(1)) {
      override def extractTimestamp(element: SensorReading): Long = {
        element.timestamp * 1000
      }
    })
    val processedStream = dataStream.keyBy(_.id)
      .process(new TempIncreAlert)
```

```
    dataStream.print("input data ")
    processedStream.print("processed data")
    env.execute("ProcessFunctionTest")
  }
}

class TempIncreAlert extends KeyedProcessFunction[String,SensorReading,String]{
  //TODO 定义一个状态，用来保存上一个数据的温度值 ，懒加载就是延迟加载
  lazy val lastTamp : ValueState[Double] =  getRuntimeContext.getState(new
ValueStateDescriptor[Double]("lastTemp",classOf[Double]))

  //TODO 定义一个状态，用来保存定时器的时间戳
  lazy val currentTimer : ValueState[Long] = getRuntimeContext.getState(new
ValueStateDescriptor[Long]("currentTimer",classOf[Long]))

  override def processElement(value: SensorReading, ctx:
KeyedProcessFunction[String, SensorReading, String]#Context, out:
Collector[String]): Unit = {
    //TODO 取出上一个温度值
    val perTemp = lastTamp.value()
    //TODO 更新温度值
    lastTamp.update(value.temprature)
    //TODO 拿出定时器时间戳  一般来讲，拿出来是有值的 ，如果没有值的话，会报错
    val curTimerTs = currentTimer.value()
    //TODO 如果温度上升且没有设过定时器 则注册定时器
    // 如果值中的温度值，大于状态温度值的话 ，注册定时器
    if (value.temprature > perTemp && curTimerTs == 0){
      //TODO 定义一个当前事件的处理时间
      val timerTs = ctx.timerService().currentProcessingTime() + 1000L
      ctx.timerService().registerProcessingTimeTimer(timerTs)
      //TODO 更新定时器 (把上面定义的事件处理时间放里面)
      currentTimer.update(timerTs)
      //TODO 如果状态温度值，大于值中的温度值或状态温度值等于0的话 直接删掉
    }else if( perTemp > value.temprature || perTemp == 0.0 ){
      //TODO 删掉定时器的时间戳
      ctx.timerService().deleteProcessingTimeTimer(curTimerTs)
      //TODO 清空状态
      currentTimer.clear()
    }
  }
  override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[String,
SensorReading, String]#OnTimerContext, out: Collector[String]): Unit = {
    //TODO 输出报警信息 ，清空状态
    out.collect(ctx.getCurrentKey + "温度持续上升")
    currentTimer.clear()
  }
}
```

侧输出流

```
object SideOutputTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    //TODO 设置事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

```
    //TODO 默认为200毫秒
    env.getConfig.setAutoWatermarkInterval(100L)
    val data = env.socketTextStream("localhost", 9999)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
      //TODO 因为assignAscendingTimestamps是毫秒 ，所以乘以1000变成秒
      // .assignAscendingTimestamps(_.timestamp * 1000)
      //TODO 处理乱序数据的话，有三个方式
      // TODO  经典方式 BoundedOutOfOrdernessTimestampExtractor  延迟一秒钟处理
      .assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.seconds(1)) {
      override def extractTimestamp(element: SensorReading): Long = {
        element.timestamp * 1000
      }
    })
    val processedStream = dataStream
      .process(new FreezingAlert)
    dataStream.print("input data ")
    //TODO 打印的是主流
    processedStream.print("processed data")
    //TODO 侧输出流   名字是下面自定义的
    processedStream.getSideOutput(new OutputTag[String]("freezing
alert")).print("alter data")
    env.execute("SideOutputTest")
  }
}
//TODO 没有keyBy操作的情况下直接，继承ProcessFunction
//TODO 冰点报警 如果大于32F,输出报价信息到输出流
class FreezingAlert extends ProcessFunction[SensorReading,SensorReading]{
  //TODO 定义一个报警信息状态
  lazy val alertOutput :OutputTag[String] = new OutputTag[String]("freezing
alert")
  override def processElement(value: SensorReading, ctx:
ProcessFunction[SensorReading, SensorReading]#Context, out:
Collector[SensorReading]): Unit = {
    //TODO 如果温度值大于32
    if(value.temprature > 32.0){
      //TODO 侧输出流
      ctx.output(alertOutput,"freezing alert for"+value.id)
    }else{
      out.collect(value)
    }
  }
}
```

Flink的状态管理

1.有状态就是有数据存储功能。有状态对象(Stateful Bean),就是有实例变量的对象,可以保存数据,是非线程安全的。在不同方法调用间不保留任何状态。
2.无状态就是一次操作,不能保存数据。无状态对象(Stateless Bean),就是没有实例变量的对象.不能保存数据,是不变类,是线程安全的。

如果超过一定的温度范围，则输出报警信息 （方法一）

```
object ProcessFunctionTest2 {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    //TODO 设置事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //TODO 默认为200毫秒
    env.getConfig.setAutoWatermarkInterval(100L)
    val data = env.socketTextStream("localhost", 9999)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
      //TODO 因为assignAscendingTimestamps是毫秒 ，所以乘以1000变成秒
      // .assignAscendingTimestamps(_.timestamp * 1000)
      //TODO 处理乱序数据的话，有三个方式
      // TODO  经典方式 BoundedOutOfOrdernessTimestampExtractor  延迟一秒钟处理
      .assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.seconds(1)) {
      override def extractTimestamp(element: SensorReading): Long = {
        element.timestamp * 1000
      }
    })
    //TODO 如果超过一定的温度范围，则输出报警信息 (超过10度就报警)
    val processFunction2 = dataStream.keyBy(_.id)
      .process(new TempChangeAlert(10.0))
      dataStream.print("input data")
      processFunction2.print("process data")
      env.execute("ProcessFunctionTest2")
  }
}
class TempChangeAlert(d:Double) extends
KeyedProcessFunction[String,SensorReading,(String,Double,Double)]{
  //TODO 定义一个状态变量，保存上一次的温度值
  lazy val lastTempState : ValueState[Double] = getRuntimeContext.getState(new
ValueStateDescriptor[Double]("lastTemp",classOf[Double]))

  override def processElement(value: SensorReading, ctx:
KeyedProcessFunction[String, SensorReading, (String, Double, Double)]#Context,
out: Collector[(String, Double, Double)]): Unit = {
    //TODO 拿到获取上一次的温度值
    val lastTemp = lastTempState.value()
    //TODO 用当前的温度值和上一次的求差，如果大于阀值，则输出报警信息
    //TODO 定义一个差值
    val diff = (value.temprature - lastTemp).abs //TODO 如果相减是负的话，则用abs ->
abs是绝对值
    if(diff > d){
      out.collect(value.id,lastTemp,value.temprature)
    }
    //TODO 否则更新温度值
    lastTempState.update(value.temprature)
  }
```

```
}
```

方法二

```
object ProcessFunctionFlatMapFunctionTest {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    //TODO 设置事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //TODO 默认为200毫秒
    env.getConfig.setAutoWatermarkInterval(100L)
    val data = env.socketTextStream("localhost", 9999)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
      //TODO 因为assignAscendingTimestamps是毫秒 ，所以乘以1000变成秒
      // .assignAscendingTimestamps(_.timestamp * 1000)
      //TODO 处理乱序数据的话，有三个方式
      // TODO  经典方式 BoundedOutOfOrdernessTimestampExtractor  延迟一秒钟处理
      .assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.seconds(1)) {
      override def extractTimestamp(element: SensorReading): Long = {
        element.timestamp * 1000
      }
    })
    //TODO 如果超过一定的温度范围，则输出报警信息 (超过10度就报警)
    val processedStream = dataStream.keyBy(_.id)
      .flatMap(new FlapTempChangeAlert(10.0))
    dataStream.print("input data")
    processedStream.print("process data")
    env.execute("ProcessFunctionFlatMapFunctionTest")
  }
}
class FlapTempChangeAlert(d: Double) extends RichFlatMapFunction[SensorReading,
(String,Double,Double)]{
  //TODO 定义一个状态保存上一个温度值
  private var lastTempState: ValueState[Double] = _
  override def open(parameters: Configuration): Unit = {
    //TODO 初始化的时候声明state变量
    lastTempState = getRuntimeContext.getState(new ValueStateDescriptor[Double]
("lastTempState",classOf[Double]) )
  }
  override def flatMap(value: SensorReading, out: Collector[(String, Double,
Double)]): Unit = {
    //TODO 获取上一个温度值
    val lastTemp = lastTempState.value()
    //TODO 用当前的温度值和上一次的求差，如果大于阀值，则输出报警信息
    //TODO 定义一个差值
    val diff = (value.temprature - lastTemp).abs
    if (diff > d){
      out.collect(value.id,lastTemp,value.temprature)
    }
```

```
    //TODO 否则更新温度值
    lastTempState.update(value.temprature)
  }
  override def close(): Unit = {
    lastTempState.clear()
  }
}
```

方式三（没看懂）

```
object ProcessFlapMapWithState {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    //TODO 设置事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //TODO 默认为200毫秒
    env.getConfig.setAutoWatermarkInterval(100L)
    val data = env.socketTextStream("localhost", 9999)
    val dataStream = data.map(s => {
      val dataArray = s.split(",")
      //TODO 返回一个SensorReading
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
    })
      //TODO 因为assignAscendingTimestamps是毫秒 ，所以乘以1000变成秒
      // .assignAscendingTimestamps(_.timestamp * 1000)
      //TODO 处理乱序数据的话，有三个方式
      // TODO  经典方式 BoundedOutOfOrdernessTimestampExtractor  延迟一秒钟处理
      .assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.seconds(1)) {
      override def extractTimestamp(element: SensorReading): Long = {
        element.timestamp * 1000
      }
    })
    //TODO 如果超过一定的温度范围，则输出报警信息 (超过10度就报警)
    val processStream = dataStream.keyBy(_.id)
      //TODO 无状态  定义一个输出，和一个状态类型
      //TODO 麻烦叭拉的方式，看不懂的玩意，呆哔操作
      .flatMapWithState[(String,Double,Double),Double]{
      //TODO 假如说没有状态的话 也就是数据没有来过，那么就将当前数据温度值存入状态
      case (input:SensorReading,None) => (List.empty,Some(input.temprature))
      //TODO 如果有状态，就应该与上次温度值比较差值，如果大于阀值就报警
      case (input:SensorReading,lastTemp:Some[Double]) => {
        val diff = (input.temprature - lastTemp.get).abs
        if(diff > 10.0 ){
          (List((input.id,lastTemp.get,input.temprature)),
(Some(input.temprature)))
        }else{
          (List.empty,Some(input.temprature))
        }
      }
    }
    dataStream.print(" input data ")
    processStream.print(" process data ")
    env.execute("ProcessFlapMapWithState")
```

```
  }
}
```

一致性问题

```
object Process {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //TODO 设置全局并行度
    env.setParallelism(1)
    //TODO 设置事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //TODO 启动检查点 单位毫秒
    env.enableCheckpointing(6000)
    //TODO 设置检查点级别 一次EXACTLY_ONCE 和 至少一次AT_LEAST_ONCE
    env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)
    //TODO 设置超时时间 单位毫秒
    env.getCheckpointConfig.setCheckpointTimeout(100000)
    //TODO 设置检查点错误失败 boolean类型
    env.getCheckpointConfig.setFailOnCheckpointingErrors(false)
    //TODO 设置最大并发检查点 默认为1
    env.getCheckpointConfig.setMaxConcurrentCheckpoints(1)
    //TODO 两个检查点之间的最小时间间隔 （在检查点之间设置最小暂停） 和上面比较冲突
    env.getCheckpointConfig.setMinPauseBetweenCheckpoints(100)
    //TODO 启动一个外部的检查点 取消删除（DELETE_ON_CANCELLATION） 保留取消
（RETAIN_ON_CANCELLATION）

 env.getCheckpointConfig.enableExternalizedCheckpoints(ExternalizedCheckpointClea
nup.RETAIN_ON_CANCELLATION)
    //TODO 设置重启策略 （RestartStrategies -> 参数） fixedDelayRestart里面的参数，
第一个参数表示的是要尝试的次数 ,第二个表示两次尝试之间的延迟，单位毫秒
    env.setRestartStrategy(RestartStrategies.fixedDelayRestart(3,500))
    //TODO 失败的最大重启次数 5分钟之内，重启3次,每次间隔时间为10秒 这个的Time类型是
import org.apache.flink.api.common.time.Time

 env.setRestartStrategy(RestartStrategies.failureRateRestart(3,Time.seconds(300),
Time.seconds(10)))
    //TODO 默认为200毫秒 自动水印间隔
    env.getConfig.setAutoWatermarkInterval(100L)
    }
  }
```

状态一致性分类

- AT-MOST-ONCE（最多一次）

➢ 当任务故障时，最简单的做法是什么都不干，既不恢复丢失的状态，也不重播丢失的数据。At-most-once 语义的含义是最多处理一次事件。

- AT-LEAST-ONCE（至少一次）

➢ 在大多数的真实应用场景，我们希望不丢失事件。这种类型的保障称为 at-least-once，意思是所有的事件都得到了处理，而一些事件还可能被处理多次。

- EXACTLY-ONCE（精确一次）

➢ 恰好处理一次是最严格的保证，也是最难实现的。恰好处理一次语义不仅仅意味着没有事件丢失，还意味着针对每一个数据，内部状态仅仅更新一次。

Flink+Kafka 端到端状态一致性的保证

- 内部 —— 利用 checkpoint 机制，把状态存盘，发生故障的时候可以恢复，保证内部的状态一致性
- source —— kafka consumer 作为 source，可以将偏移量保存下来，如果后续任务出现了故障，恢复的时候可以由连接器重置偏移量，重新消费数据，保证一致性
- sink —— kafka producer 作为sink，采用两阶段提交 sink，需要实现一个 TwoPhaseCommitSinkFunction