# Streaming(DataStream API)

原文参考:

<https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/datastream_api.html#collection-data-sources>

## Overview

* + 1. Flink DataStream Api 编程指南

在Flink中的DataStream 程序在数据流(data streams)上实现了各种转换(transformation)操作(如,filter,updating,state,window,aggregating 等)。Data Streams 可以从各种数据源(message queue,socket,fiels 等)中被创建。产生的结果可以输出到各种sink(目的地)，比如将它写入到数据文件或一些标准的输出当中。Flink 程序可以在各种环境中运行，如 standlone ,嵌入到其他程序中。Flink能在本地的JVM中执行，也可以在集群中运行（yarn）.

flink Api的基本概念请参照 [basic concepts](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/api_concepts.html)

为了创建你自己的Flink DataStream 程序，我们鼓励你一开始使用 [anatomy of a Flink Program](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/api_concepts.html#anatomy-of-a-flink-program) 并逐步的添加 [stream transformations](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/index.html). 下面的章节将为添加一些operations(翻者注:Flink 中的任何的transformations)和高级特性做一些引用说明

|  |
| --- |
| Example 程序案例 |
| Data Source 数据源 |
| DataStream transformation |
| Data sink 数据输出 |
| Iterations |
| Execution Parametes 执行参数 |
| Fault Tolerance(故障容错) |
| Controlling Latency (延迟控制) |
| Debugging |
| Local Execution Envionment |
| Collection Data Sources |
| Iterator Data Sink |
| Where to go next (下一站)? |

* + 1. Example Program

下面的代码是一段完整的基于窗口的 word count 应用的例子，单词的数量来源于一个5秒窗口的socket . 你可以复制到本地并运行它。

|  |
| --- |
| Java 代码片段 |
| **public class** WindowWordCount {  **public static void** main(String[] args) **throws** Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();   DataStream<Tuple2<String, Integer>> dataStream = env  .socketTextStream(**"localhost"**, 9999)  .flatMap(**new** Splitter())  .keyBy(0)  .timeWindow(Time.*seconds*(5))  .sum(1);   dataStream.print();   env.execute(**"Window WordCount"**);  }   **public static class** Splitter **implements** FlatMapFunction<String, Tuple2<String, Integer>> {  @Override  **public void** flatMap(String sentence, Collector<Tuple2<String, Integer>> out) **throws** Exception {  **for** (String word: sentence.split(**" "**)) {  out.collect(**new** Tuple2<String, Integer>(word, 1));  }  }  } } |

|  |
| --- |
| Scala 代码片段 |
| import org.apache.flink.streaming.api.scala.\_ import org.apache.flink.streaming.api.windowing.time.Time /\*\*  \* Created by yuanhailong on 2018/9/19.  \*/ object WindowWordCount {  def main(args: Array[String]) {   val env = StreamExecutionEnvironment.getExecutionEnvironment  val text = env.socketTextStream("localhost", 9999)   val counts = text.flatMap { \_.toLowerCase.split("\\W+") filter { \_.nonEmpty } }  .map { (\_, 1) }  .keyBy(0)  .timeWindow(Time.seconds(5))  .sum(1)   counts.print()   env.execute("Window Stream WordCount")  } } |

为了运行这个例子，首先你需要启动在命令行终端用netcat 启动一个输入流：

|  |
| --- |
| nc –lk 9999 |

只要输入一些词就会返回一些新的单词。这些词将会成为word count 程序的输入。如果你想看到的结果大于1 。你只要重复的输入5秒钟之内相同的词即可。（如果你的输入不够快，你可以增加窗口大小）

* + 1. Data Sources [数据源]

数据源表示你的程序从哪里读取数据。通过StreamExecutionEnvironment.addSource(sourceFunction). 你能添加数据源到你的程序中。Flink 实现了几种数据源函数(function) ,但你可以通过实现SourceFunction 自定义数据源[翻者注：SourceFunction并行度1]。如果你想要实现多个并行度的数据源函数你可以通过实现ParallelSourceFunction 接口或者扩展RichParallelSourceFunction 。

有一些预先定义的数据源来源于StreamExecutionEnvironment。

file-based[基于文件的]:

* readTextFile(path)：读取文本文件，file 遵循TextInputFormat 规范，文本文件中的数据每一行作为一个字符串返回。
* readFile(fileinputFormat,path): 通过指定文件的输入格式来读取数据文件
* readFile(fileInputFormat, path, watchType, interval, pathFilter) ：这个方法的调用实际是通过上面两个方法中的一个来实现的。它使用给定的fileInputFormat读取指定路径下面的文件。根据提供的watchType. 数据源可能周期性（根据interval ms）的监控Path路径下的新数据(FileProcessingMode.PROCESS\_CONTINUOUSLY)。或者仅处理一次当前路径下面的数据然后退出（FileProcessingMode.PROCESS\_ONCE）。使用pathFilter排除不需要处理的数据。

IMPLEMENTATION（实现）：

在内部，Flink 将读数据程序划分为两个子任务(sub-task) ,也就是目录监控和数据读取。每个子任务通过独立的条目实现。监控是通过并行度为1的任务实现的。然而数据读取时通过多个任务并行实现的。并行度等于Job任务的并行度。目录监控任务去监控目录（根据watchType 周期性的监控或仅读取一次），找到文件，切割文件，并切割文件到下游readers . readers将读取实际的数据。每个切割的文件仅被一个readers 读取。然而一个readers 可以读取多个文件。

IMPORTANT NOTES（特别注意）:

1. 如果watchType 被设置为 FileProcessingMode.PROCESS\_CONTINUOUSLY。当files被修改的时候，它的整个内容将会被重新处理。这就会破坏“exactly-once”的语义，当追加数据到文件的末尾将导致所有的数据都会被重新处理。
2. 如果watchType 被设置为FileProcessingMode.PROCESS\_ONCE. 数据源只会被扫描一次然后退出，无需等待readers完成文件内容的读取[这里指的是监控服务]。当然readers 会持续读取文件内容直到文件内容读取完成.关闭source 将会导致此后的信息不会再有检查点。这将导致在节点失败后恢复变慢，因为Job需要从上一个检查点恢复

Socked-Based:

* socketTextStream: 从Socket中读取数据。通过指定分隔符切割数据

Collection-Based:

* fromCollection(Seq): 从java 的  Java.util.Collection 中创建data stream,集合中所有的元素必须具备相同的数据类型
* fromCollection(Iterator)：从Iterator中创建data stream. 该类指定迭代器返回的元素的数据类型。
* fromElements(elements: \_\*) : 从一系列的对象中创建data stream. 所有的对象必须具备相同的类型
* fromParallelCollection(SplittableIterator):从Iterator中创建data stream. 该类指定迭代器返回的元素的数据类型。
* generateSequence(from, to) ：在给定的范围类生成一系列的数字
  + 1. DataStream transformations

参见 [operators](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/index.html)

* + 1. Data Sinks

Data sinks 消费 DataStream中的数据并将数据输出到files,socket，其他额外系统或print。 Flink 有多种输出格式它封装了DataStream上的背后的多种operators

* writeAsText() / TextOutputFormat: 写元素一行作为一个String . 这个Strings 通过调用每个元素的toString() 方法来获取。
* writeAsCsv(...) / CsvOutputFormat: 用逗号分隔value写元组(tuple). Row 和Filed分隔符可配置。Value通过调用toString（） 方法来获取。
* print() / printToErr()：打印每个元素toString()的value到标准输出。
* writeUsingOutputFormat() / FileOutputFormat：方法和自定义文件输出的基础类。支持自定义的对象到字节的转换
* writeToSocket：根据SerializationSchema 写元素到Socket
* addSink : 调用自定义的sink 函数。Flink 自带了多重sink 函数（如Apache kafka）

注意，在DataStrem上的Write()方法主要是为了调试的目的。他们不会参加flink的chekpoint操作。这就意味着它使用的是“at-least-once”语义。数据如何刷写到目标系统依赖于实现的OutputFormat. 这就意味着不是发送到目标系统的数据会立即展现出来。当然，在失败的场景中，这些数据可能会丢失。

为了可靠性，strema exactly-once 传递到文件系统，可以使用flink-connector-filesystem。

* + 1. Iterations

Iterative streaming(迭代流)程序实现一个step 函数，并将其嵌入到IterativeStream中。由于一个DataStream程序可能永远都不会完成，因此没有最大的迭代次数。相反，你需要指定那些stream需要返回到iteration并且通过split或filter transformation 指定那些需要输出到下游。在这里，我们有一个iteration例子。代码的主体部分是一个简单的map 转换 ，并通过返回的元素区分不同的元素返回给下游。

|  |
| --- |
| **val** iteratedStream **=** someDataStream**.**iterate**(**  iteration => {  val iterationBody = iteration.map(/\* this is executed many times \*/)  (iterationBody.filter(/\* one part of the stream \*/), iterationBody.filter(/\* some other part of the stream \*/))  }) |

例如： 这里有一个程序冲一个整数中持续减1，直到它等于0

|  |
| --- |
| val someIntegers: DataStream[Long] = env.generateSequence(0, 1000)  val iteratedStream = someIntegers.iterate(  iteration => {  val minusOne = iteration.map( v => v - 1)  val stillGreaterThanZero = minusOne.filter (\_ > 0)  val lessThanZero = minusOne.filter(\_ <= 0)  (stillGreaterThanZero, lessThanZero)  }  ) |

* + 1. Execution Parameters

StreamExecutionEnvironment  包含ExecutionConfig  ，ExecutionConfig 允许为Flink运行时设置一些配置参数。

更多的参数参见[execution configuration](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/execution_configuration.html) 。这些参数属于DataStream API:

* setAutoWatermarkInterval(long millseconds):设置watermark发射的频率。通过getAutoWatermarkInterval可以得到当前的watermark的value.
  + - 1. Fault Tolerance(故障容错)

[State & Checkpointing](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/checkpointing.html) 描述了如何启用Flink的checkpoint 机制。

* + - 1. Controlling Latency

默认情况下，数据元素在网络上不是一对一的传输(如果这样将会导致不必要的网络延迟)而是先缓存起来。缓存（在两台机器上实际传输的对象）的大小在flink的配置文件中能被配置。为了更好的吞吐量这往往是一个好方法，但是当数据不足够快的时候会导致一定的数据延迟。为了控制吞吐量和延迟。在execution 环境上你可以通过env.setBufferTimeout(timeoutMillis)设置缓存等待被填满的最大等待时间。这样即使缓存区没有被填满也会被自动发送。这个timeout的默认值时100 ms

Usage:

|  |
| --- |
| val env: LocalStreamEnvironment = StreamExecutionEnvironment.createLocalEnvironment  env.setBufferTimeout(timeoutMillis)  env.generateSequence(1,10).map(myMap).setBufferTimeout(timeoutMillis) |

为了最大的吞吐量。set setBufferTimeout(-1)，这样会移除timeout并且只有当缓存区填满的时候才能被发送。为了最小的延迟，设置timeout = 0 来关闭缓存。Timeout=0 应该要去避免，因为这会引起服务性能下降。

* + 1. Debugging

在分布式集群中运行分布式程序之前，一个很好的办法是确定实现的算法按照期待的方式运行。因此，实现数据分析的程序通常是一个结果检查，调试，改善提高的过程。

Flink 在IDE内通过本地调试的方式提供了数据分析程序开发处理的特性，注入测试，收集数据。本小节将给一些提示如何开发Flink程序。

* + - 1. Local Execution Enviroment

LocalStreamEnvironment 在同一的JVM内启动内创建Flink System.如果你是在IDE里面启动LocalEnvironment 。你可以在你的代码中打断点这样就很容易去调试了。

|  |
| --- |
| val env = StreamExecutionEnvironment.createLocalEnvironment()  val lines = env.addSource(/\* some source \*/)  // build your program  env.execute() |

* + - 1. Collection Data Sources

Flink 为方便调试通过java collections 提供了一些特殊的数据源,一旦程序通过测试，source 和 sink 很容易被替换。

|  |
| --- |
| val env = StreamExecutionEnvironment.createLocalEnvironment()  // Create a DataStream from a list of elements  val myInts = env.fromElements(1, 2, 3, 4, 5)  // Create a DataStream from any Collection  val data: Seq[(String, Int)] = ...  val myTuples = env.fromCollection(data)  // Create a DataStream from an Iterator  val longIt: Iterator[Long] = ...  val myLongs = env.fromCollection(longIt) |

* + - 1. Iterator Data Sink

Flink 为调试和测试的目的提供了收集DataStream 结果的sink .可以像下面这样使用：

|  |
| --- |
| import org.apache.flink.streaming.experimental.DataStreamUtils  import scala.collection.JavaConverters.asScalaIteratorConverter  val myResult: DataStream[(String, Int)] = ...  val myOutput: Iterator[(String, Int)] = DataStreamUtils.collect(myResult.javaStream).asScala |

注：在flink 1.5.0 中 flink-streaming-contrib 被移除了。使用flink-streaming-java 和flink-streaming-scala 来替代

* + 1. Where to go next（下一步）?
* [Operators](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/index.html): stream operators 规范说明
* [Event Time](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_time.html)：介绍flink的时间概念
* [State & Fault Tolerance](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/index.html):解释如何开发有状态的应用
* [Connectors](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/connectors/index.html)：描述有效的输入输出Connectors

## Event Time

# Batch (DataSet API)