# 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field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WriteAs\*()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，stream 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结束，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

|  |
| --- |
| **Event Time / Process Time / Ingestion time** |
| Setting a Time Characteristic |
| **Event Time and Watermarks [事件时间和水位]** |
| Watermarks in parallel Streams [并行流中的水位] |
| Late Elements [延迟元素] |
| Debugging Watermarks |

* + 1. Event Time / Process time / Ingestion Time

Flink 在流式程序中支持不同类型的时间概念:

* Process time:Porcess time 是执行各自operation时候机器系统时间

当streaming 程序采用processing time运行时，所有基于时间的操作（如 windows） 都将使用机器系统时钟来运行各自的operator. 每小时处理时间窗口将包含在系统时钟指向下一个整点时间之间的所有记录.例如，如果一个应用在早上9：15开始，那么第一个小时处理的时间窗口将包含上午9：15到10:00的数据。下一个窗口是10：到11:00

Processing time 是最简单的时间概念，它不需要在streams和机器之间协调。它提供了最高吞吐量以及最低的延迟。然而，在分布式和异步的环境中他不能保证正确性，因为它容易受到记录到达系统的影响，以及系统内部和外部的影响。

* Event time：Event time 的时间是在每台生产设备上发生的时间【既日志记录产生时间】。这个时间在到达Flink之前被嵌入到了日志里面，并且每条日志都能够抽取event timestamp. 在 event time 中，数据处理的时间依赖于数据本身。Event time的程序必须指定如何获取 Event Time Watermarks ,这是event time 内部信号处理的一种机制。Watermark 机制将在后面的章节描述[below](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_time.html#event-time-and-watermarks).

在一个非常完美的世界中，event time 程序产生一个始终一致和确定性的结果，而不用去理会event何时到达以及它的顺序。然而，除非event time确定数据是按次序到达的，否则，在等待event时，会导致事件处理的延迟。由于不可能无限的等待下去，这就限制了应用程序的event time

假设所有的event都到达了，event time 操作处理会如预期一样，并即使在event乱序和有有延迟的情况下都能产生正确和一致的结果。例如，一个小时的时间窗口将包含该时间内所有数据而不用管他们以什么顺序到达和何时到达(更多信息查看 [late events](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_time.html#late-elements))

注意：有时当event time 处理实时数据时，为了保证能够及时正确的处理他们会使用processing time 来进行操作。

* Ingestion time：Ingestion time 是event 进入flink的时间，每行记录在source operator 获得source的当前时间作为timestamp

Ingestion time 位于Event time 和 processing Time 之间 。对比processing time Ingestion time更加昂贵，但是他给定一个可预测的结果，因为Ingestion time给定了一个更加稳定的时间（在source一次性分配），不同的window操作会有相同的时间戳，然而在processing time中每个不同的window 操作都可能在相同的记录上分配不同的时间。

对比Event Time ，Ingestion time 不能处理乱序或延迟的event，但是程序不需要指定watermark.

实际上，Ingestion time 趋向于event time，只不过自动的分配了timestamp和watermark.



* + - 1. Setting a Time Characteristic

Flink DataStream 程序的开始通常都是设置一个时间特性。

下面是Flink程序在一个小时的窗口内做聚合的例子。

|  |
| --- |
| val env = StreamExecutionEnvironment.getExecutionEnvironment  env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)  // alternatively:  // env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime)  // env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)  val stream: DataStream[MyEvent] = env.addSource(new FlinkKafkaConsumer09[MyEvent](topic, schema, props))  stream  .keyBy( \_.getUser )  .timeWindow(Time.hours(1))  .reduce( (a, b) => a.add(b) )  .addSink(...) |

注：为了在这个案例中运行event time例子，程序需要为数据指定event time 和 watermark。或者程序在Source后注入一个 Time Assigner 和 Watermark。

下面的小节将描述时间戳和watermark背后的机制，。至于在Flink DataStream API中如何使用assignment 和watermark，请查阅 [Generating Timestamps / Watermarks](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_timestamps_watermarks.html).

* + 1. Event Time and Watermarks

注:Flink 从DataStream 模型实现了多种技术。 event time 和watermarks 更好的介绍，请参阅：

* [Streaming 101](https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101)
* The [Dataflow Model paper](https://research.google.com/pubs/archive/43864.pdf)

流处理器为支持event time 需要有一个方法去测量event time的进度。例如，当event时间超过一小时后，构建每小时窗口的窗口operator需要得到通知,因此Opertor 能关闭window的处理。

Event time的处理独立于processing time . 例如，一个operator的event time 时间可能稍微落后于processing time(考虑event的延迟)，而两者以相同的速度运行。另一方面，另外一个streaming 程序通过快速的消费缓存在kafka topic(或其他消息队列)的数据可能在几秒钟之内就能处理数周的event time 的数据。

在flink机制中监测event time的进度是通过watermark来实现。Watermark flow携带timestamp作为data stream 的一部分。Watermark 定义了在stream中到达的时间t ,这意味着从带有timestamp的stream中不应有更多的元素

下面的图片这展示了一个带时间戳的event stream和水位线。在这个例子中时间是有序的（通过各自的timestamp）,这意味着在stream中watamark较容易周期性标记



Watermark 更重要的是为无序的streams,正如下面阐述的一样，event不是有序的（通过timestamp） ,在一个通常的watermark定义了stream中的一个点，所有的event 在某个确定的timestamp都应该到达。一旦watermark到达operator，operator可以将其内部event time 时钟 提前到 watermark值。



* + - 1. Watermarks in parallel Streams

Watermarks是在source function 生成或在其后指定的，source function的每个并行子任务通常都生成各自的watermark. watermark在特定的并行source上定义event time .

当watermarks通过stream程序流动时，它将为其后续operator在下游生成一个新的watermarks。

有些operators消费多个input stream, 比如 union ,keyBy ,partition 等函数。这些operators的当前event time 是他们input stream 最小的event time .就像input streams 更新它的event time 一样。

下面的图显示了event 和 watermark 通过并行stream 流动的示意图 。operator 跟踪event time.



注：kafka 支持每个分区的watermark，更多参见[here](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_timestamps_watermarks.html#timestamps-per-kafka-partition).

* + - 1. Late Elements

有可能某些元素将违反水位的条件，这意味着watermark发生后，还会有部分stream 元素。实际上，在真实的世界中，某些元素就是延迟到达的，我们无法指定某个事件时间戳的发生时间。此外，即使延迟的元素是有界的，延迟太多通常是不被建议的，因为在window操作中会引起更多的延迟。

基于这个原因，streaming 能够显示的指定一些延迟的元素，late element的element是指在系统事件时间时钟(由watermark发出信号)已经超过late element时间戳后到达的element,更多资料参见[Allowed Lateness](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#allowed-lateness)

* + - 1. Debugging Watermarks

参见[Debugging Windows & Event Time](https://ci.apache.org/projects/flink/flink-docs-release-1.6/monitoring/debugging_event_time.html)

## State & Fault Tolerance（状态&容错）

* + 1. Overview(概览)

有状态的function和operator通过处理独立的element/event 存储数据，这使得状态成为了构建任何复杂operator的关键组件。

例如：

* 当application搜索某个event模型时，状态将存储到目前为止一系列的事件
* 当每分钟/小时/天发生一个聚合事件时, 状态将保留这些被悬置(还没有到达聚合时间)聚合数据
* 当在一个数据流上训练一个机器学习模型时，状态将保存当前版本的模型参数。
* 当历史数据需要被管理时，状态允许有效的接受过去发生过的event

Flink系统需要知道状态，以便能够使用[checkpoints](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/checkpointing.html) 标记状态容错以及使用streaming application 的[savepoints](https://ci.apache.org/projects/flink/flink-docs-release-1.6/ops/state/savepoints.html)

关于Flink的状态它允许你在应用中进行调整，这意味着你在重新划分并行度的时Flink会管理好状态

Flink [queryable state](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/queryable_state.html)  特征允许你在运行期间接收外部状态。

使用状态进行工作，对 [Flink’s state backends](https://ci.apache.org/projects/flink/flink-docs-release-1.6/ops/state/state_backends.html)[状态恢复]非常有用，Flink 通过指定状态如何存储以及存储在哪里提供了多种状态恢复机制。State可以存储在本地JVM的堆内和堆外内存。Flink管理应用的state, 这意味着flink为了允许能够管理非常大的状态需要自己处理内存管理（状态也可能被溢写到磁盘）。在无需改变任何应用代码的情况下可以配置State backends。

* + 1. Working with state(状态处理)

这篇文档将解析在开发flink应用的时如何使用flink的状态。

|  |
| --- |
| **Keyed State And Operator State** |
| Keyed State |
| Operator State |
| **Raw and Manager State** |
| **Using Managed Keyed State** |
| State Time-to-Live (TTL) |
| State in the Scala DataStream Api |
| **Using Managed Operator State** |
| Stateful Source Functions |

* + - 1. Keyed State And Operator State

在Flink的状态中有两个基本的状态概念 Keyed State 和 Operator State

* + - * 1. Keyed State

Keyed State 状态总是和key相关的并且他只能使用在KeyedStream的function和Operator上。

你可以将Keyed State 看做是一个已经分区分片并的Operator State，它的每个key只有一个状态分区。每个Keyed State 逻辑绑定到唯一的一组< parallel-operator-instance, key >, 由于每个key “属于” 一个keyed operator的并行实例，我们可以简单的认为<operator,key>

Keyed State 进一步组织为所谓的Key Groups，Key Group是 通过Flink 重新分配Keyed State 的原子单元。Key group 数量和定义的最大并行度相同。在执行过程中，每keyed operator 的并行实例都使用一个或多个Key groups

* + - * 1. Operator State

Operator 状态(非 keyed state)，每个operator 状态绑定到一个并行操作实例。在Flink的状态使用中 [Kafka Connector](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/connectors/kafka.html)是一个很好的例子。Kafka的消费者实例的每个分区维护了topic分区和偏移量的映射作为它的Operator State

当并行性发生变化时，operator status 接口支持在并行操作符实例之间重新分配状态。他们之间可以有不同的scema

* + - 1. Raw和 Managed 状态

Keyed State 和 Operator State 存在两种形式 Managed 和 Raw。

Managed State 是 被Flink 运行时管理的数据结构。例如哈希表或RocksDB. 例如”ValueState”,”ListState” 等。Flink 运行时对状态编码并写入到checkpoints

Raw State 是 Operator 保留在它们自己的数据结构内的状态。当发生checkpoint的时候Operator只需要将一系列的字节写入到checkpoint中。对于state的数据结构Flink一无所知，它只仅能看到原始的字节。

所有的datastream 函数都能使用Managed State, 但是raw state 接口只有当实现了Opertator的时才能被使用。我们推荐使用Managed State。因为使用Managed State当Flink的并行度发生变化的时候Flink能自动的重新分布state和更好内存管理。

注意：如果你的managed state 需要自定义序列化逻辑，为了消除兼容性请查阅[corresponding guide](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/custom_serialization.html) 。 Flink的序列化在默认情况下不需要做特殊处理。

* + - 1. Using Managed Keyed Status

Managed keyed stated 接口提供了对不同类型状态的访问，这些状态的作用域是当前输入元素的key . 这意味着这些状态的key 仅能使用在keyedStream上。KeyStream可以通过stream.keyBy()创建。

现在，我将首先探讨可用状态的不同类型，然后我将看下如何在在程序中使用它们。有效状态原语是：

* ValueState<T>:它保留了一个可以被更新和获取的一个value(输入元素key的作用于上面已经提到了，因此这里每个key可能拥有一个value.可以使用update(T)设置该值以及 T value（） 方法获取value
* ListState<T>：他保留了一组元素。你可以追加元素和在当前所有存储的元素上获得一个Iterable 。 添加元素可以使用add(T) 和 addAll(List(T))方法。通过使用terable<T> get() 获取这个迭代器（Iterable ）。你也可以使用update(List(T))方法覆盖已经存在的List.
* ReducingState<T>: 表示一个所有的value聚合后添加到State的单个value. 这个接口类似于ListState，只不过使用add(T)添加的元素通过使用一个特殊的ReduceFunction进行聚合。
* AggregatingState<IN, OUT>：表示所有的value聚合后添加到state的一个value. 对比ReducingState 。 聚合的类型可能和被添加到状态的元素类型不同。这个接口和ListState一样，只不过使用add(IN)添加的元素通过使用一个特殊的AggregateFunction进行聚合。
* FoldingState<T, ACC>:示所有的value聚合后添加到state的一个value. 对比ReducingState. 聚合的类型可能和被添加到状态的元素类型不同，只不过使用add(IN)添加的元素通过使用一个特殊的FoldFunction进行聚合。
* MapState<UK, UV>:它保留了一个list的映射，你可以设置键值对到到state中以及从当前存储的state mapping 中获取一个Iterable 。 Mapping的添加可以通过put(UK, UV) 和 putAll(Map<UK, UV>)方法。Value 通过一个相关的key获取。Iterable 的mappings，keys和values 可以通过entries(), keys()和 values()来分别获取。

所有的类型都可以通过一个clear()方法来清空当前Active key的state.

注意：FoldingState 和FoldingStateDescriptor 在Flink1.4中已经过时，在未来的版本中将会被彻底移除。请使用AggregatingState和AggregatingStateDescriptor替代。

需要牢记的是这些state object（状态对象）只能和state交互。状态不一定存储在内存，他有可能存储在磁盘或者其他地方。第二个要记住的是从你从state中获得的value 取决于输入元素的key. 所以你获得的value可能会在不同的调用中不相同。

通过创建StateDescriptor 可以获得state的句柄(handle). StateDescriptor保留状态的名字（后面你会看到，你创建了若干state,由于他们都会有唯一的名称所以你能引用它们），状态保存的值类型(value type) ，还可能有一些用户函数，比如ReduceFunction 。 根据你想获取的状态类型，你可以要么创建一个ValueStateDescriptor，ListStateDescriptor，ReducingStateDescriptor，FoldingStateDescriptor 要么创建MapStateDescriptor。

使用RuntimeContext来访问State. 因此他只可能在 rich function中。关于rich function 的更多信息请查看[here](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/api_concepts.html#rich-functions) 。这里我们有一个简短的例子。RichFunction中有有效的RuntimeContext为访问state有这些方法：

* ValueState<T> getState(ValueStateDescriptor<T>)
* ReducingState<T> getReducingState(ReducingStateDescriptor<T>)
* ListState<T> getListState(ListStateDescriptor<T>)
* AggregatingState<IN, OUT> getAggregatingState(AggregatingState<IN, OUT>)
* FoldingState<T, ACC> getFoldingState(FoldingStateDescriptor<T, ACC>)
* MapState<UK, UV> getMapState(MapStateDescriptor<UK, UV>)

这是一个FlatMapFunction  的例子并展示了各个部分如何在组合在一起的。

|  |
| --- |
| Java 代码 |
| */\*\*  \* Created by yuanhailong on 2018/9/25.  \*/* **public class** CountWindowAverage **extends** RichFlatMapFunction<Tuple2<Long,Long>,Tuple2<Long,Long>> {  */\*\*  \* ValueState 句柄 第一个元素是数量 第二个元素是求和。  \*/* **private transient** ValueState<Tuple2<Long, Long>> **sum**;   @Override  **public void** flatMap(Tuple2<Long, Long> input, Collector<Tuple2<Long, Long>> out) **throws** Exception {  *//访问 state* Tuple2<Long, Long> currentSum = **sum**.value();   *// 更新数量* currentSum.**f0** += 1;   *//第二个元素求和* currentSum.**f1** += input.**f1**;   *//更新状态* **sum**.update(currentSum);   *//如果状态达到2 ，发送平均值并清空* **if**(currentSum.**f0**>=2){  out.collect(**new** Tuple2<>(input.**f0**,currentSum.**f1**/currentSum.**f0**));  **sum**.clear();  }  }   @Override  **public void** open(Configuration parameters) **throws** Exception {  ValueStateDescriptor<Tuple2<Long,Long>> descriptor=  **new** ValueStateDescriptor<Tuple2<Long, Long>>(  "average",  TypeInformation.of(new TypeHint<Tuple2<Long, Long>>() {})，  Tuple2.of(0L,0L)  );   **sum**=getRuntimeContext().getState(descriptor);  } }  env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L), Tuple2.of(1L, 7L), Tuple2.of(1L, 4L), Tuple2.of(1L, 2L))  .keyBy(0)  .flatMap(new CountWindowAverage())  .print(); |

|  |
| --- |
| Scala 代码 |
| class CountWindowAverage extends RichFlatMapFunction[(Long, Long), (Long, Long)] {  private var sum: ValueState[(Long, Long)] = \_  override def flatMap(input: (Long, Long), out: Collector[(Long, Long)]): Unit = {  // access the state value  val tmpCurrentSum = sum.value  // If it hasn't been used before, it will be null  val currentSum = if (tmpCurrentSum != null) {  tmpCurrentSum  } else {  (0L, 0L)  }  // update the count  val newSum = (currentSum.\_1 + 1, currentSum.\_2 + input.\_2)  // update the state  sum.update(newSum)  // if the count reaches 2, emit the average and clear the state  if (newSum.\_1 >= 2) {  out.collect((input.\_1, newSum.\_2 / newSum.\_1))  sum.clear()  }  }  override def open(parameters: Configuration): Unit = {  sum = getRuntimeContext.getState(  new ValueStateDescriptor[(Long, Long)]("average", createTypeInformation[(Long, Long)])  )  }  }  object ExampleCountWindowAverage extends App {  val env = StreamExecutionEnvironment.getExecutionEnvironment  env.fromCollection(List(  (1L, 3L),  (1L, 5L),  (1L, 7L),  (1L, 4L),  (1L, 2L)  )).keyBy(\_.\_1)  .flatMap(new CountWindowAverage())  .print()  // the printed output will be (1,4) and (1,5)  env.execute("ExampleManagedState")  } |

这个例子实现了一个很简单的窗口计数器。我们的key是元组(tuple)的第一个字段，函数存储count和运行的sum到ValueState中。一旦count到达2时它将发送一个平均值并清空状态。所以他就会又从0开始。注：如果我们的元组（tuple）第一个字段不同的value 它将保存不同的状态

* + - 1. State Time-To-Live(TTL)

任何类型的Keyed State都能够分配Time-To-Live(TTL). 如果说配置了TTL并且State已经过期，stored value 将被清空，下面详细讨论

所有state集合类型支持每个条目级别的TTLs.这就意味着List和Map元素的TTLs是相互独立的。

为了使用state TTL 必须首先构建StateTtlConfig 配置对象。通过传递配置，TTL的功能可以在任何state descriptor 中启用。

|  |
| --- |
| Java 代码 |
| import org.apache.flink.api.common.state.StateTtlConfig**;**  import org.apache.flink.api.common.state.ValueStateDescriptor**;**  import org.apache.flink.api.common.time.Time**;**  StateTtlConfig ttlConfig **=** StateTtlConfig  **.**newBuilder**(**Time**.**seconds**(**1**))**  **.**setUpdateType**(**StateTtlConfig**.**UpdateType**.**OnCreateAndWrite**)**  **.**setStateVisibility**(**StateTtlConfig**.**StateVisibility**.**NeverReturnExpired**)**  **.**build**();**    ValueStateDescriptor**<**String**>** stateDescriptor **=** **new** ValueStateDescriptor**<>(**"text state"**,** String**.**class**);**  stateDescriptor**.**enableTimeToLive**(**ttlConfig**);** |

|  |
| --- |
| Scala 代码 |
| **import** org.apache.flink.api.common.state.StateTtlConfig  **import** org.apache.flink.api.common.state.ValueStateDescriptor  **import** org.apache.flink.api.common.time.Time  **val** ttlConfig **=** **StateTtlConfig**  **.**newBuilder**(Time.**seconds**(**1**))**  **.**setUpdateType**(StateTtlConfig.UpdateType.OnCreateAndWrite)**  **.**setStateVisibility**(StateTtlConfig.StateVisibility.NeverReturnExpired)**  **.**build    **val** stateDescriptor **=** **new** **ValueStateDescriptor[String](**"text state"**,** classOf**[String])**  stateDescriptor**.**enableTimeToLive**(**ttlConfig**)** |

这个配置有若干个选项值得考虑：

第一个方法的参数newBuilder是强制的，它设置TTL.

Update 类型在更新TTL时配置（默认是OnCreateAndWrite）

* StateTtlConfig.UpdateType.OnCreateAndWrite ： 仅创建和写入的时候访问
* StateTtlConfig.UpdateType.OnReadAndWrite ： 包含读取

状态的可见性配置是指过期的value如果在读取的时候没有被清除是否返回（默认NeverReturnExpired）

* StateTtlConfig.StateVisibility.NeverReturnExpired：永不返回过期值
* StateTtlConfig.StateVisibility.ReturnExpiredIfNotCleanedUp：如果有效返回过期value

这个例子中NeverReturnExpired，过期状态的行为就好像它已经不存在了，即使它仍然需要被移除。这个选项在TTL发生后访问变得不在有效特别有用。

另外一个选项ReturnExpiredIfNotCleanedUp 在清空之前允许返回过期数据。

**Notes:**

* State 背后存储了用户最后一次修改的时间戳，这意味着启用这个特征是增量消耗存储的状态。Heap State 在内存中额外存储了一个引用用户对象和原始long 值得java对象。RocksDB状态后端为每个存储value、list或map添加8字节
* 目前仅支持processing time 的TTL
* 之前没有配置TTL尝试恢复State,将导致兼容性失败和StateMigrationException
* TTL配置不是checkpoint和savepoint的一部分而是Flink如何在当前运行的job中如何处理的一种方式。
* 如果用户序列化可以处理Null值那么使用TTL的map state可以处理null 值。如果不支持Null值可以采用NullableSerializer包装，代价是1个字节的额外消费。
  + - * 1. 清空过期状态

目前，过期的值只有在显示的通过调用ValueState.value()读取时才会被删除。

注意:这意味着默认清空下过期的value 如果没有被读取是不会被移除的。可能导致state的增长。这在未来发布版本中被修改。

此外,你可以在获得完整的快照时候激活清空操作这样将减少state大小。 local state在目前的实现中不会被删除。但是当它从上一个快照恢复的时候不会包含过期的数据。在StateTtlConfig可以配：

|  |
| --- |
| Java 代码 |
| import org.apache.flink.api.common.state.StateTtlConfig**;**  import org.apache.flink.api.common.time.Time**;**  StateTtlConfig ttlConfig **=** StateTtlConfig  **.**newBuilder**(**Time**.**seconds**(**1**))**  **.**cleanupFullSnapshot**()**  **.**build**();** |

|  |
| --- |
| Scala 代码 |
| **import** org.apache.flink.api.common.state.StateTtlConfig  **import** org.apache.flink.api.common.time.Time  **val** ttlConfig **=** **StateTtlConfig**  **.**newBuilder**(Time.**seconds**(**1**))**  **.**cleanupFullSnapshot  **.**build |

这个选项不适合在rockdbs state 增量的checkpoint

* + - 1. State In Scala Data Stream Api

除了上面的描述的接口外，Scala Api 在KeyedStream单个ValueState 的map和flatmap函数上有一些快捷方法。这个用户函数获得当前ValueState状态的value.并且他必须返回一个更新的value用户更新state.

|  |
| --- |
| **val** stream**:** **DataStream[(String**, **Int)]** **=** **...**  **val** counts**:** **DataStream[(String**, **Int)]** **=** stream  **.**keyBy**(\_.**\_1**)**  **.**mapWithState**((**in**:** **(String,** **Int),** count**:** **Option[Int])** **=>**  count **match** **{**  **case** **Some(**c**)** **=>** **(** **(**in**.**\_1**,** c**),** **Some(**c **+** in**.**\_2**)** **)**  **case** **None** **=>** **(** **(**in**.**\_1**,** 0**),** **Some(**in**.**\_2**)** **)**  **})** |

* + - 1. Using Managed State Operator

为了使用managed operator state ，一个有状态的函数要么实现CheckpointedFunction接口要么实现ListCheckpointed<T extends Serializable>  接口

* + - * 1. CheckpointedFunction

CheckpointedFunction接口通过重分配不同的schema提供对non-keyed state的访问。实现它必须要实现两个方法

|  |
| --- |
| **void** **snapshotState(**FunctionSnapshotContext context**)** **throws** Exception**;**  **void** **initializeState(**FunctionInitializationContext context**)** **throws** Exception**;** |

无论一个checkpoint合适执行，snapshotState（）方法总是会被调用（既只要执行快照就会执行该方法）。initializeState（） 是每次用户定义的函数初始化的时候被调用或从之前的checkpoint中恢复的时候进行调用。因此，initializeState() 不仅是不同的状态类型被初始化的地方而且也包含了逻辑的恢复。

目前，list-style managed operator state是被支持的。这个state期望是一个序列化的List对象，彼此相互独立，因而在重新调整时可以重新分配state. 换句话说，这些对象是可以重新分配non-keyed state的最佳粒度。根据状态可访问方法，定义了以下重分配方案：

* **Even-split redistribution：**每个operator返回一组state元素，所有列表的整个状态在逻辑上是串联的。在恢复/重分配，该列表被均匀划分为与并行operator相同数量的子列表。每个operator都会得到一个子列表，这个字列表可能是空的或者包含一个或者多个元素。作为一个例子，如果一个包含了element1和element2的operator的checkpoint state 使用的并行度为1 ，那么当增加并行度到2的时，element1 可能在operator第0个实例中而element2可能在实例1中。
* **Union redistribution：**每个operator返回一组state元素，所有列表的整个状态在逻辑上是串联的。在恢复/重分配，每个operator都将获得一个完整的一组state element.

下面的例子是一个有状态的SinkFunction 。 在发送他们到外部之前使用CheckpointedFunction 缓存元素。这里演示了基本的 even-split redistribution 列表状态。

|  |
| --- |
| Java 代码 |
| **package** com.feicheng.java.flink.office.example;  **import** org.apache.flink.api.common.state.ListState; **import** org.apache.flink.api.common.state.ListStateDescriptor; **import** org.apache.flink.api.common.typeinfo.TypeHint; **import** org.apache.flink.api.common.typeinfo.TypeInformation; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.runtime.state.FunctionInitializationContext; **import** org.apache.flink.runtime.state.FunctionSnapshotContext; **import** org.apache.flink.streaming.api.checkpoint.CheckpointedFunction; **import** org.apache.flink.streaming.api.functions.sink.SinkFunction;  **import** java.util.ArrayList; **import** java.util.List;  */\*\*  \* Created by yuanhailong on 2018/9/26.  \*/* **public class** BufferingSink **implements** SinkFunction<Tuple2<String,Integer>> ,CheckpointedFunction{   **private final int threshold**;   **private transient** ListState<Tuple2<String,Integer>> **checkpointedState**;   **private** List<Tuple2<String, Integer>> **bufferedElements**;    **public** BufferingSink(**int** threshold){  **this**.**threshold**=threshold;  **this**.**bufferedElements** = **new** ArrayList<>();  }   @Override  **public void** invoke(Tuple2<String, Integer> value, Context context) **throws** Exception {  **bufferedElements**.add(value);  **if**(**bufferedElements**.size()==**this**.**threshold**){  **for**(Tuple2<String,Integer> element:**bufferedElements**){  *//****TODO 发送到sink*** }  }  }   @Override  *//只要发生快照就会执行该方法* **public void** snapshotState(FunctionSnapshotContext functionSnapshotContext) **throws** Exception {  *//快照繁盛的时候清空之前的快照信息，添加现有的数据到快照* **checkpointedState**.clear();  **for** (Tuple2<String, Integer> element : **bufferedElements**) {  **checkpointedState**.add(element);  }  }   @Override  *//初始化，快照恢复的时候回执行该方法* **public void** initializeState(FunctionInitializationContext context) **throws** Exception {  ListStateDescriptor<Tuple2<String, Integer>> descriptor =  **new** ListStateDescriptor<Tuple2<String, Integer>>(  **"buffered-elements"**,*//名称必须全局唯一* TypeInformation.*of*(**new** TypeHint<Tuple2<String, Integer>>() {}));   **checkpointedState** = context.getOperatorStateStore().getListState(descriptor);    **if**(context.isRestored()){  **for** (Tuple2<String, Integer> element : **checkpointedState**.get()) {  **bufferedElements**.add(element);  }  }  } } |

initializeState 获取到一个FunctionInitializationContext参数。它是用于初始化non-keyed state的“容器”。是Liststate 类型检查点存储的位置。

注：状态初始化类似于keyed state ,使用StateDescriptor 包含状态的名字以及和value有关的一些类型信息。

|  |
| --- |
| ListStateDescriptor<Tuple2<String, Integer>> descriptor =  **new** ListStateDescriptor<Tuple2<String, Integer>>(  **"buffered-elements"**,*//名称必须全局唯一* TypeInformation.*of*(**new** TypeHint<Tuple2<String, Integer>>() {}));   **checkpointedState** = context.getOperatorStateStore().getListState(descriptor); |

State 访问方法的命名包含了重分配模式以及它的数据结构。例如，为了使用list state 在restore上使用union redistribution scheme[详见上文描述] 。通过使用getUnionListState(descriptor) 方法获得state. 如果名字不包含重分配模式那么使用getListState(descriptor) ，这意味着只是简单的使用even-split redistribution 方案。

Container 被初始化之后，我们使用context的isRestored() 方法去检查在失败后是否恢复成功。如果这个值等于true，那么将使用恢复逻辑进行恢复。

正如在BufferingSink 的代码片段中展现的一样，ListState 恢复 state初始化期间保留在类变量中为将来在snapshotState（）中使用。ListState 对象被清空包含之前的checkpoint，然后填充我们想要checkpoint的元素到ListState 。

* + - * 1. ListCheckpointed

ListCheckpointed结果是CheckpointedFunction具有更多限制的变体，在restore上仅支持使用了even-split 重分配方案的list-style 状态。它也必须实现两个方法：

|  |
| --- |
| List**<**T**>** **snapshotState(long** checkpointId**,** **long** timestamp**)** **throws** Exception**;**  **void** **restoreState(**List**<**T**>** state**)** **throws** Exception**;** |

snapshotState() 要返回一个List 对象为checkpoint 和restoreState 处理list上的恢复。如果state没有重分区。在snapshotState() 你总是可以返回Collections.singletonList(MY\_STATE)。

* + - 1. Stateful Source Functions

对比其他operator有状态的Sources必须要更多的关注。为了让state的更新和输出集合原子化（在失败/恢复中保证exactly-once 语义），用户必须从source’s 上下文（context）中获得锁。

|  |
| --- |
| Java 代码 |
| **package** com.feicheng.java.flink.office.example;  **import** org.apache.flink.streaming.api.checkpoint.ListCheckpointed; **import** org.apache.flink.streaming.api.functions.source.RichParallelSourceFunction;  **import** java.util.Collections; **import** java.util.List;  */\*\*  \* Created by yuanhailong on 2018/9/26.  \*/* **public class** CounterSource **extends** RichParallelSourceFunction<Long> **implements** ListCheckpointed<Long> {  */\*\* current offset for exactly once semantics \*/* **private** Long **offset**;   */\*\* flag for job cancellation \*/* **private volatile boolean isRunning** = **true**;   @Override  **public void** run(SourceContext<Long> ctx) **throws** Exception {  **final** Object lock = ctx.getCheckpointLock();  **while** (**isRunning**) {  *// output and state update are atomic* **synchronized** (lock) {  ctx.collect(**offset**);  **offset** += 1;  }  }  }   @Override  **public void** cancel() {  **isRunning** = **false**;  }   @Override  **public** List<Long> snapshotState(**long** l, **long** l1) **throws** Exception {  **return** Collections.*singletonList*(**offset**);  }   @Override  **public void** restoreState(List<Long> state) **throws** Exception {  **for** (Long s : state)  **offset** = s;  } } |

|  |
| --- |
| Scala 代码 |
| **class** **CounterSource**  **extends** **RichParallelSourceFunction[Long]**  **with** **ListCheckpointed[Long]** **{**  @volatile  **private** **var** isRunning **=** **true**  **private** **var** offset **=** 0L  **override** **def** run**(**ctx**:** **SourceFunction.SourceContext[Long]):** **Unit** **=** **{**  **val** lock **=** ctx**.**getCheckpointLock  **while** **(**isRunning**)** **{**  *// output and state update are atomic*  lock**.**synchronized**({**  ctx**.**collect**(**offset**)**  offset **+=** 1  **})**  **}**  **}**  **override** **def** cancel**():** **Unit** **=** isRunning **=** **false**  **override** **def** restoreState**(**state**:** **util.List[Long]):** **Unit** **=**  **for** **(**s **<-** state**)** **{**  offset **=** s  **}**  **override** **def** snapshotState**(**checkpointId**:** **Long,** timestamp**:** **Long):** **util.List[Long]** **=**  **Collections.**singletonList**(**offset**)**  **}** |

当Flink完全了解checkpoint时，可能需要这些信息和外部世界沟通。请查阅 org.apache.flink.runtime.state.CheckpointListener 接口。

* + 1. The BroadCast State Pattern(广播状态模型)

|  |
| --- |
| **Provided APIs** |
| BroadcastProcessFunction 和 KeyedBrodcastProcssFunction |
| **Import Considerations** |

[Working with State](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/state.html)  介绍了operator 要么均匀的分布并行operator任务，要么统一的使用整个状态来恢复并行的任务。

支持的第三种operator类型是Broadcast state. Broadcast state 用于支持某些输入来源于一个stream并且这个stream被下游任务都使用的场景。Broadcast state存储在本地并且用于处理在其它stream上的每个元素。我们可以把它想象为一个低吞吐量的流，它包含了一些规则，我们想从其它stream上过来的元素都使用这个规则进行评估。Broadcast state 和其它Operator state 有一些不同：

* Map 格式
* 它只有在一个输入是 broadcasted stream 和 non-broadcasted 的特殊operator上才有效
* Operator可以使用不同的名字拥有多个broadcast states
  + - 1. Provided Apis

为了说明提供的APis，在呈现所有功能前我们将使用一个例子。当我们运行这个例子的时候。我将使用一个具有不同颜色和形状的stream对象的例子。并且我们想找到具有相同颜色的对象，他们遵循一定的模式比如矩形或者三角形。我们假设这个有趣的模型随着时间而变化。

在这个例子中，第一个Stream的element是一个包含了color和shape属性的Item对象。其他stream包含了Rules

从Item 流开始，由于想要相同color的对，我们只需要通过color作为它的key。这将确保相同具备相同color的元素在相同的物理机上。

|  |
| --- |
| *// key the shapes by color*  KeyedStream**<**Item**,** Color**>** colorPartitionedStream **=** shapeStream  **.**keyBy**(new** KeySelector**<**Shape**,** Color**>(){...});** |

接下来是rules，stream包含这些rules并将它们广播到所有的下游任务。并且这些任务将这些规则保留在本地。因此rules可以对所有进来的Item进行评估。下面的代码片段：

1. 广播所有的rules
2. 使用MapStateDescriptor，它将在rules被存储的地方创建broadcast state

|  |
| --- |
| *// a map descriptor to store the name of the rule (string) and the rule itself.*  MapStateDescriptor**<**String**,** Rule**>** ruleStateDescriptor **=** **new** MapStateDescriptor**<>(**  "RulesBroadcastState"**,**  BasicTypeInfo**.**STRING\_TYPE\_INFO**,**  TypeInformation**.**of**(new** TypeHint**<**Rule**>()** **{}));**  *// broadcast the rules and create the broadcast state*  BroadcastStream**<**Rule**>** ruleBroadcastStream **=** ruleStream  **.**broadcast**(**ruleStateDescriptor**);** |

最后，为了评估从Item Stream进来的元素的规则。我需要

1. 连接（connect）两个stream
2. 指定匹配逻辑

用BroadcastStream 连接一个Stream(keyed 或 non-keyed) 可以通过在non-broadcast stream上通过调用connect()方法来完成。它将返回一个BroadcastConnectedStream。 在这个上面我们可用一个CoProcessFunction调用process() 方法。 这个函数将包含我们的匹配逻辑。函数精确类型依赖于non-broadcasted stream的类型。

* 如果是keyed ，那么这个函数是KeyedBroadcastProcessFunction
* 如果是**non-keyed，那么这个函数是**BroadcastProcessFunction

考虑到我们的non-broadcast是keyed ,下面的代码片段包含上面的调用：

注意：connect要在 non-broadcasted stream上调用。使用BroadcastStream 作为一个参数

|  |
| --- |
| DataStream**<**Match**>** output **=** colorPartitionedStream  **.**connect**(**ruleBroadcastStream**)**  **.**process**(**  *// type arguments in our KeyedBroadcastProcessFunction represent:*  *// 1. the key of the keyed stream*  *// 2. the type of elements in the non-broadcast side*  *// 3. the type of elements in the broadcast side*  *// 4. the type of the result, here a string*  **new** KeyedBroadcastProcessFunction**<**Color**,** Item**,** Rule**,** String**>()** **{**  *// my matching logic*  **}**  **)** |

* + - * 1. BroadcastProcessFunction和KeyedBroadcastProcessFunction

就像CoProcessFunction一样，这些函数有两个方法可以去实现。processBroadcastElement() 代表了在broadcast stream中处理进来的数据而processElement（） 在non-broadcast stream中使用。方法的完整签名如下：

|  |
| --- |
| **public** **abstract** **class** **BroadcastProcessFunction<**IN1**,** IN2**,** OUT**>** **extends** BaseBroadcastProcessFunction **{**  **public** **abstract** **void** **processElement(**IN1 value**,** ReadOnlyContext ctx**,** Collector**<**OUT**>** out**)** **throws** Exception**;**  **public** **abstract** **void** **processBroadcastElement(**IN2 value**,** Context ctx**,** Collector**<**OUT**>** out**)** **throws** Exception**;**  **}** |

|  |
| --- |
| **public** **abstract** **class** **KeyedBroadcastProcessFunction<**KS**,** IN1**,** IN2**,** OUT**>** **{**  **public** **abstract** **void** **processElement(**IN1 value**,** ReadOnlyContext ctx**,** Collector**<**OUT**>** out**)** **throws** Exception**;**  **public** **abstract** **void** **processBroadcastElement(**IN2 value**,** Context ctx**,** Collector**<**OUT**>** out**)** **throws** Exception**;**  **public** **void** **onTimer(long** timestamp**,** OnTimerContext ctx**,** Collector**<**OUT**>** out**)** **throws** Exception**;**  **}** |

首先我们需要注意的是为了能够在broadcast 端能处理元素，两个函数都必须实现processBroadcastElement() 方法，为了能在 non-broadcasted端处理元素实现processElement()方法。

这两个方法在他们提供的context有一些差异。non-broadcast端是ReadOnlyContext而broadcasted 是Context。

这两个context:

1. 给定一个访问broadcast state 方法ctx.getBroadcastState(MapStateDescriptor<K, V> stateDescriptor)
2. 允许查询每个元素的时间戳：ctx.timestamp()
3. 获取当前水位：ctx.currentWatermark()
4. 获取当前processing time : ctx.currentProcessingTime()
5. 发送元素到输出端： ctx.output(OutputTag<X> outputTag, X value)

在getBroadcastState()中的stateDescriptor 应该在上面的.broadcast(ruleStateDescriptor) 中能唯一区分。

不同之处在于每种方法对broadcast state的访问类型。Broadcast是读写访问，non-broadcast只有只读访问。这是应为在Flink中没有交叉任务。所以，为了保证我们的operator的所有并行实例有相同的访问控制，我将读写访问仅给了broadcast端，在所有任务上他们都会看到相同的元素。忽略掉rules规则被破坏一致性导致的不一致性。通常这也是非常难以去调试的结果。

注意：逻辑上实现processBroadcast（）必须在所有的并行实例上有相同的确定性行为。

最后，KeyedBroadcastProcessFunction 是在keyed stream上的一个operator,它公开了一些在BroadcastProcessFunction不是有效的功能。它们是:

* 在processElement  方法中的ReadOnlyContext给了一个访问Flink底层timer（定时器） 服务，当timer(定时器)服务被触发时，onTimer()通过OnTimerContext 被调用， OnTimerContext 和ReadOnlyContext 有相同的功能并增加了：
  + 能够询问定时器触发是event time 或processing time
  + 查询与定时器相关的key

这和KeyedProcessFunction的onTimer() 方法是一致的：

* Context 的processBroadcastElement()方法包含applyToKeyedState(StateDescriptor<S, VS> stateDescriptor, KeyedStateFunction<KS, S> function) 方法，这个方法允许注册一个KeyedStateFunction 到所有key的所有state和提供的stateDescriptor关联。

注：由于没有key 和broadcasted 元素关联，因此注册定时器只可能在KeyedBroadcastProcessFunction的processElement方法上而不能在processBroadcastElement方法上

回到我们最初的例子，KeyedBroadcastProcessFunction 看上去像下面这样：

|  |
| --- |
| **new** KeyedBroadcastProcessFunction**<**Color**,** Item**,** Rule**,** String**>()** **{**  *// store partial matches, i.e. first elements of the pair waiting for their second element*  *// we keep a list as we may have many first elements waiting*  **private** **final** MapStateDescriptor**<**String**,** List**<**Item**>>** mapStateDesc **=**  **new** MapStateDescriptor**<>(**  "items"**,**  BasicTypeInfo**.**STRING\_TYPE\_INFO**,**  **new** ListTypeInfo**<>(**Item**.**class**));**  *// identical to our ruleStateDescriptor above*  **private** **final** MapStateDescriptor**<**String**,** Rule**>** ruleStateDescriptor **=**  **new** MapStateDescriptor**<>(**  "RulesBroadcastState"**,**  BasicTypeInfo**.**STRING\_TYPE\_INFO**,**  TypeInformation**.**of**(new** TypeHint**<**Rule**>()** **{}));**  @Override  **public** **void** **processBroadcastElement(**Rule value**,**  Context ctx**,**  Collector**<**String**>** out**)** **throws** Exception **{**  ctx**.**getBroadcastState**(**ruleStateDescriptor**).**put**(**value**.**name**,** value**);**  **}**  @Override  **public** **void** **processElement(**Item value**,**  ReadOnlyContext ctx**,**  Collector**<**String**>** out**)** **throws** Exception **{**  **final** MapState**<**String**,** List**<**Item**>>** state **=** getRuntimeContext**().**getMapState**(**mapStateDesc**);**  **final** Shape shape **=** value**.**getShape**();**  **for** **(**Map**.**Entry**<**String**,** Rule**>** entry:  ctx**.**getBroadcastState**(**ruleStateDescriptor**).**immutableEntries**())** **{**  **final** String ruleName **=** entry**.**getKey**();**  **final** Rule rule **=** entry**.**getValue**();**  List**<**Item**>** stored **=** state**.**get**(**ruleName**);**  **if** **(**stored **==** **null)** **{**  stored **=** **new** ArrayList**<>();**  **}**  **if** **(**shape **==** rule**.**second **&&** **!**stored**.**isEmpty**())** **{**  **for** **(**Item i **:** stored**)** **{**  out**.**collect**(**"MATCH: " **+** i **+** " - " **+** value**);**  **}**  stored**.**clear**();**  **}**  *// there is no else{} to cover if rule.first == rule.second*  **if** **(**shape**.**equals**(**rule**.**first**))** **{**  stored**.**add**(**value**);**  **}**  **if** **(**stored**.**isEmpty**())** **{**  state**.**remove**(**ruleName**);**  **}** **else** **{**  state**.**put**(**ruleName**,** stored**);**  **}**  **}**  **}**  **}** |

* + - 1. 重要考虑(important Considerations)

Api描述完成 后，下面聚焦于在使用broadcast state状态时需要重点记住的几点：

* 没有交叉任务交流：如上所述，这是为什么只能在BroadcastProcessFunction 的broadcast端修改broadcast state内容的原因。此外，用户需要确保所有进来的每个元素以相同的方式修改broadcast state. 否则，不同的任务可能有不同的内容从而导致结果不一致。
* Broadcast State的event顺序因不同的交叉任务而不同：虽然广播一个流的元素可以保证所有元素(最终)都将到达所有下游任务，但是元素可能以不同的顺序到达每个任务。因此状态的更新必须不能依赖进来event的顺序。
* 所有任务都checkpoint 他们的broadcast state:尽管当checkpoint 发生时所有的元素在他们的broadcast state中都有相同的元素，但是所有的任务checkpoint他们的broadcast state而不仅仅是他们中的一个。这样的设计是为了避免在恢复期间从同一个文件中读取。
* **No RocksDB state backend： Broadcast 保存在内存，必须有相应的内存供给去保留所有的operator state**
  + 1. Checkpointing(检查点)

|  |
| --- |
| **Prequisites(预置条件)** |
| **Enableing and Configuring Checkpointing** |
| Related Config Options |
| **Selecting a State Backend** |
| **State Checkpoint in Iterative Jobs** |
| **Restart Strategies** |

在Flink中的每个function和Operator 都能拥有状态（详见[working with state](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/state.html)） 。有状态的函数在处理独立的element/event时存储数据，是的state成为许多关键operator的一个关键构建块。

为了让state可以容错，Flink需要将state checkpoint(检查点)。 Checkpoints 允许Flink在Stream中恢复存储的数据和位置从而给应用程序当失败恢复的时候具备相同的语义。

[documentation on streaming fault tolerance](https://ci.apache.org/projects/flink/flink-docs-release-1.6/internals/stream_checkpointing.html)  详细的描述了Flink背后的容错机制。

* + - 1. **Prequisites(预置条件)**

Flink的checkpoint机制需要和持久化的stream和state进行交互。一般情况下，必须:

* 持久化的数据可在一定持续的时间对数据进行重放。比如消息队列（apache kafka,RabbitMQ 等）或文件系统（HDFS,S3,GFS,NFS,CEPH）
* 状态的持久化存储，通常是分布式文件系统（HDFS,S3…）
  + - 1. **Enableing and Configuring Checkpointing（启用&配置checkpoint）**

Checkpoint在默认情况下是被禁用的。为了启用checkpoint,在StreamExecutionEnvironment上通过调用enableCheckpointing（n） ，n代表checkpoint的频率，单位是毫秒。

Checkpoint的一些其他参数包含：

* exactly-once vs. at-least-once:在调用enableCheckpointing（n,mode）方法的时候你可以选择他们其中的一个mode。在大多数应用中exactly-once是更好的选择。At-least-once相对而言会有更低的延迟。
* checkpoint timeout:如果在checkpoint在这个时间内没有完成，则会退出checkpoint.
* minimum time between checkpoints：为了让streaming应用在checkpoint之间取得一定的进展，一种可能的定义是在checkpoint之间需要花费多少时间。如果这个value设置为5000，下一个checkpoint将在上一个checkpoint完成后的5秒钟内进行启动，而不用例会checkpoint的持续时间和频率。注checkpoint的时间不能比这个参数小。
* number of concurrent checkpoints：默认情况下当一个checkpoint在处理的时候系统不会触发另一个checkpoint.这样可以确保这个拓扑不会在checkpoint上花费太多时间。但是多个检查点是有可能的，一个有趣的现象处理的管道总会有一些延迟（例如函数从外部系统获取数据）但是我们想要在上面重新处理一下小的失败。

这个选项在checkpoint的minimum time 定义的时候不可用。

* externalized checkpoints：你可以配置定期性的将chekpoint持久化到外部系统，checkpoint写入元数据到外部存储系统，当Job失败的时候不会自动清理checkpoint.这样如果Job失败了你就会有一个可恢复的checkpoint. 更多详情参见[deployment notes on externalized checkpoints](https://ci.apache.org/projects/flink/flink-docs-release-1.6/ops/state/checkpoints.html#externalized-checkpoints).
* fail/continue task on checkpoint errors:如果在执行任务的checkpoint过程中发生错误决定是否任务就失败了，这是默认行为。然而，当他被禁用的时候，任务将通知checkpoint协处理器，并继续运行。

|  |
| --- |
| Java 代码 |
| StreamExecutionEnvironment env **=** StreamExecutionEnvironment**.**getExecutionEnvironment**();**  *// start a checkpoint every 1000 ms*  env**.**enableCheckpointing**(**1000**);**  *// advanced options:*  *// set mode to exactly-once (this is the default)*  env**.**getCheckpointConfig**().**setCheckpointingMode**(**CheckpointingMode**.**EXACTLY\_ONCE**);**  *// make sure 500 ms of progress happen between checkpoints*  env**.**getCheckpointConfig**().**setMinPauseBetweenCheckpoints**(**500**);**  *// checkpoints have to complete within one minute, or are discarded*  env**.**getCheckpointConfig**().**setCheckpointTimeout**(**60000**);**  *// allow only one checkpoint to be in progress at the same time*  env**.**getCheckpointConfig**().**setMaxConcurrentCheckpoints**(**1**);**  *// enable externalized checkpoints which are retained after job cancellation*  env**.**getCheckpointConfig**().**enableExternalizedCheckpoints**(**ExternalizedCheckpointCleanup**.**RETAIN\_ON\_CANCELLATION**);** |

* + - * 1. Related Config Options(相关配置项)

更多参数或默认值在 conf/flink-conf.yaml中进行设置（配置指南查看[*configuration*](https://ci.apache.org/projects/flink/flink-docs-release-1.6/ops/config.html)）

|  |  |  |
| --- | --- | --- |
| 配置项 | 默认值 | 描述 |
| state.backend | 无 | 用于存储和checkpoint的状态 |
| state.backend.async | true | 状态备份是否应该使用异步快照的方式，这个是可配置的。某些状态备份可能不支持异步快照，也有些仅支持异步快照，有些可能直接忽略这个选项。 |
| state.backend.fs.memory-threshold | 1024 | 状态数据文件大小的最小值。所有比他小的state chunks 都被存储在根检查点的元数据文件中。 |
| state.backend.incremental | false | 状态备份是否应该创建增量检查点。增量checkpoint值存储当前checkpoint一些有差异的部分，而不是完整的checkpoint。有些state backend可能不支持增量checkpoint，那么他们会直接会略掉这个选项 |
| state.backend.local-recovery | false |  |
| state.checkpoints.dir | 无 | 这个目录用于配置flink支持的文件系统来存储checkpoint的数据文件和元数据。 |
| state.checkpoints.num-retained | 1 | 需要被保留的已经完成了的checkpoint的最大值 |
| state.savepoints.dir | 无 | Savepoint的目录，调用savepoint命令时将状态写入到这里 |
| taskmanager.state.local.root-dirs | 无 |  |

* + - 1. **Selecting a State Backend（状态备份）**

Flink的 [checkpointing mechanism](https://ci.apache.org/projects/flink/flink-docs-release-1.6/internals/stream_checkpointing.html) 在计时器和有状态的操作（包含connect,window 和 [user-defined state](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/state.html)）中存储所有的一致快照。Checkpoint被存储位置（JobManager 内存，文件系统，数据库等）依赖于**State Backend** 的配置。

默认情况下，state保存在TaskManagers 的内存中，checkpoint保存在JobManager的内存中。大的state更适合持久化，Flink支持各种方法将state和checkpoint存储在其他系统中。通过StreamExecutionEnvironment.setStateBackend(…)可以选择state存储的后端备份系统。

有效的状态备份的更多详细信息可查看[state backends](https://ci.apache.org/projects/flink/flink-docs-release-1.6/ops/state/state_backends.html)

* + - 1. **State Checkpoint in Iterative Jobs（迭代JOB中的State 检查点）**

Flink目前仅支持为没有迭代的JOB提供处理的正确性保证。在迭代的job上启用checkpoint将引起一些异常。为了在迭代的程序中强制启用checkpoint用户需要启动一个标识：env.enableCheckpointing(interval, force = true).

* + - 1. **Restart Strategies(重启策略)**

在失败的情况下为了控制job如何重启Flink提供了不同的重启策略。详情查阅 [Restart Strategies](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/restart_strategies.html).

* + 1. Queryable State(可查询状态)

|  |
| --- |
| **Architecture(架构)** |
| **Activating Queryable state(激活可查询的状态)** |
| **Making State Queryable** |
| Queryable State Stream |
| Managed keyed State |
| **Querying State** |
| Example |
| **Configuration** |
| State Server |
| Proxy |
| **Limitations** |

注意：Queryable state 客户端API 目前还在进行中，它提供的接口不能保证稳定性。在即将发布的Flink版本中有可能中断客户端Api的修改。

简单来说，该特性将Flink管理的Keyed state(见 [Working with State](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/state.html))发布到外部世界并且允许用户查询来自于Flink外部的Job状态。在一些场景中，queryable state 通常和外部系统有一些分布式的operations/transactions，比如key-value存储，在实践中这通常是系统的瓶颈。此外，这个特性在调试中可能特别有用。

注意：当查询一个state对象时，该对象可以在不进行任何异步和复制的情况下从并发线程中访问。这是一种设计上的选择，上面的任何一种情况都可能导致Job的延迟而这是我们想要避免的。使用java 堆空间(MemoryStateBackend , FsStateBackend)的任何state backends 当获取value的时候都不会复制而是直接访问存储的引用value. 由于read-modify-write模式是不安全的，因此他可能导致在并发修改过程中某些queryable state服务的失败。但是RocksDBStateBackend 是安全的。

* + - 1. **Architecture(架构)**

前面描述了如何使用Queryable State,简要的描述他的实体和组成是非常有用的。Queryable State 特性包含了三个主要的条目：

1. QueryableStateClient，运行在Flink 集群外部并提交用户的查询
2. QueryableStateClientProxy，运行在每个taskmanager上（Flink集群内部）并负责接收客户端请求，并将结果返回给客户端
3. QueryableStateServer 运行在每个taskmanager上并负责服务本地存储状态

client 连接到其中的一个代理并发送请求用一个特殊的key【k】管理状态。在[Working with State](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/state.html)已经知道，keyed state 被组织在Key Groups，*并且每个taskmanager都将分配一些key gro*ups. 为了发现那个Taskmanager持有k的key groups ,代理将询问JobManager. 根据这个答案，代理将查询运行在分配了k转台的TaskManager上的QueryableStateServer，并负责返回给客户端

* + - 1. **Activating Queryable State**

为了在你的Flink集群中启用queryable  state,你必须要从你的[Flink distribution](https://flink.apache.org/downloads.html)的opt/folder下复制flink-queryable-state-runtime\_2.11-1.6.1.jar 到lib/folder下，否则 queryable state特性不会被启用。

为了校验集群使用 启用了queryable state正在运行，检查任意一个taskmanager的日志输出行:” Started the Queryable State Proxy Server @ ..." 

* + - 1. **Making State Queryable**

现在你已经在你的集群中启用了 queryable state，是时候看如何使用它了。为了state状态在外部世界可见，通过使用下面的方式现实的创造queryable.

* QueryableStateStream,一个能充当sink 并且能作为queryable state 转入值得接收器的一个便利对象。
* stateDescriptor.setQueryable(String queryableStateName)  方法，这使得用状态描述符的keyed state 表述为queryable

下面的小节展示了如何使用上面描述的两种方法。

* + - * 1. Queryable State Stream

在KeyedStream上调用.asQueryableState(stateName, stateDescriptor) 返回一个QueryableStateStream 。它提供了一个表示queryable state 的values.根据状态的类型。asQueryableState方法有各种类型。

|  |
| --- |
| *// ValueState*  QueryableStateStream **asQueryableState(**  String queryableStateName**,**  ValueStateDescriptor stateDescriptor**)**  *// Shortcut for explicit ValueStateDescriptor variant*  QueryableStateStream **asQueryableState(**String queryableStateName**)**  *// FoldingState*  QueryableStateStream **asQueryableState(**  String queryableStateName**,**  FoldingStateDescriptor stateDescriptor**)**  *// ReducingState*  QueryableStateStream **asQueryableState(**  String queryableStateName**,**  ReducingStateDescriptor stateDescriptor**)** |

注意：这里没有ListState sink ，它的结果将日益增长，导致最后消费过多的内存。

返回的QueryableStateStream 可以看做是一个sink 并且不能被转换。在内部，QueryableStateStream获得一个operator 这个operator使用所有的数据更新queryable state 实例。更新逻辑通过StateDescriptor 提供的asQueryableState 调用。就想下面这样。Keyed stream 所有的记录都将使用状态实例ValueState.update(value):

|  |
| --- |
| stream**.**keyBy**(**0**).**asQueryableState**(**"query-name"**)** |

这个行为就像scala API flatMapWithState.

* + - * 1. Managed Keyed State

管理operator的keyed state （见[Using Managed Keyed State](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/state.html#using-managed-keyed-state)）

|  |
| --- |
| ValueStateDescriptor**<**Tuple2**<**Long**,** Long**>>** descriptor **=**  **new** ValueStateDescriptor**<>(**  "average"**,** *// the state name*  TypeInformation**.**of**(new** TypeHint**<**Tuple2**<**Long**,** Long**>>()** **{}));** *// type information*  descriptor**.**setQueryable**(**"query-name"**);** |

注意:queryableStateName 参数可以任意选择它仅仅用于查询，他不能标识状态自己的名称

* + - 1. Querying state

到目前为止，你已经设置了你的集群采用 queryable state 运行并且你已经定义了queryable的状态。现在是时候如何去查询这些状态了。

为了查看状态我们将使用QueryableStateClient  帮助类。这个类在flink-queryable-state-client jar 中是有效的。它必须显示的包含在pom.xml的依赖中，就想下面这样。

|  |
| --- |
| <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-core</artifactId>  <version>1.6.1</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-queryable-state-client-java\_2.11</artifactId>  <version>1.6.1</version>  </dependency> |

关于这个的更多信息，请查阅[set up a Flink program](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/linking_with_flink.html).

QueryableStateClient 会提交你的查询到内部的代理，然后处理你的查询并返回最终的结果。初始化客户端仅需要提供有效的TaskManager 主机名（记住：queryable state query在每个taskmanger上都有运行）和端口。代理在用这个主机名和端口进行监听。代理的更多配置和state服务端口请查阅[Configuration Section](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/queryable_state.html#Configuration).

|  |
| --- |
| QueryableStateClient client **=** **new** QueryableStateClient**(**tmHostname**,** proxyPort**);** |

客户端准备好了，查询类型V的状态，用类型K的key 进行分配，你可以使用如下方法：

|  |
| --- |
| CompletableFuture**<**S**>** **getKvState(**  JobID jobId**,**  String queryableStateName**,**  K key**,**  TypeInformation**<**K**>** keyTypeInfo**,**  StateDescriptor**<**S**,** V**>** stateDescriptor**)** |

上面代码片段返回了CompletableFuture ，通过带有JobID的queryableStateName 持有queryable state instance的状态值。Key是你感兴趣的key的状态。keyTypeInfo 将告诉Flink如何序列化和反序列化。最后，stateDescriptor 包含了关于请求状态的必要信息，命名类型（Value,Reduce,etc）以及如何序列化和反序列化相关的必要信息。

细心的读者会注意到返回包含类型S的value. 一个状态对象包含了实际的值。它可以是Flink支持的任意的状态类型： ValueState, ReduceState, ListState, MapState, AggregatingState以及过时了的FoldingState。

注意：状态对象不允许修改包含的状态，你可以使用它们获得实际的状态值。例如，使用valueState.get()或迭代 <K, V> 条目，又如，使用mapState.entries()，但是你不可以修改它们，作为一个例子，在返回的list state上调用add()方法将抛出 UnsupportedOperationException.

注意： 客户端是异步的并且多个线程共享。当不需要使用空闲资源的时候需要通过QueryableStateClient.shutdown() 关闭。

* + - * 1. Example

下面的例子通过使用queryable和如何使用value扩展CountWindowAverage 例子（见 [Using Managed Keyed State](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/state.html#using-managed-keyed-state)）

|  |
| --- |
| **public** **class** **CountWindowAverage** **extends** RichFlatMapFunction**<**Tuple2**<**Long**,** Long**>,** Tuple2**<**Long**,** Long**>>** **{**  **private** **transient** ValueState**<**Tuple2**<**Long**,** Long**>>** sum**;** *// a tuple containing the count and the sum*  @Override  **public** **void** **flatMap(**Tuple2**<**Long**,** Long**>** input**,** Collector**<**Tuple2**<**Long**,** Long**>>** out**)** **throws** Exception **{**  Tuple2**<**Long**,** Long**>** currentSum **=** sum**.**value**();**  currentSum**.**f0 **+=** 1**;**  currentSum**.**f1 **+=** input**.**f1**;**  sum**.**update**(**currentSum**);**  **if** **(**currentSum**.**f0 **>=** 2**)** **{**  out**.**collect**(new** Tuple2**<>(**input**.**f0**,** currentSum**.**f1 **/** currentSum**.**f0**));**  sum**.**clear**();**  **}**  **}**  @Override  **public** **void** **open(**Configuration config**)** **{**  ValueStateDescriptor**<**Tuple2**<**Long**,** Long**>>** descriptor **=**  **new** ValueStateDescriptor**<>(**  "average"**,** *// the state name*  TypeInformation**.**of**(new** TypeHint**<**Tuple2**<**Long**,** Long**>>()** **{}));** *// type information*  descriptor**.**setQueryable**(**"query-name"**);**  sum **=** getRuntimeContext**().**getState**(**descriptor**);**  **}**  **}** |

一旦你在Job中使用它，你能获取job ID然后查询这个operator任何key的当前状态。

|  |
| --- |
| QueryableStateClient client **=** **new** QueryableStateClient**(**tmHostname**,** proxyPort**);**  *// the state descriptor of the state to be fetched.*  ValueStateDescriptor**<**Tuple2**<**Long**,** Long**>>** descriptor **=**  **new** ValueStateDescriptor**<>(**  "average"**,**  TypeInformation**.**of**(new** TypeHint**<**Tuple2**<**Long**,** Long**>>()** **{}));**  CompletableFuture**<**ValueState**<**Tuple2**<**Long**,** Long**>>>** resultFuture **=**  client**.**getKvState**(**jobId**,** "query-name"**,** key**,** BasicTypeInfo**.**LONG\_TYPE\_INFO**,** descriptor**);**  *// now handle the returned value*  resultFuture**.**thenAccept**(**response **->** **{**  **try** **{**  Tuple2**<**Long**,** Long**>** res **=** response**.**get**();**  **}** **catch** **(**Exception e**)** **{**  e**.**printStackTrace**();**  **}**  **});** |

* + - 1. Configuration（配置）

下面的配置参数影响queryable 服务和客户端的行为。他们在QueryableStateOptions中定义。

* + - * 1. State Server
* query.server.ports：queryable 服务端口的服务端口范围，如果超过1个task manager 在同一台机器上运行对于避免冲突这个非常有用。这个范围可以是一个端口“9123”和一个范围端口“50100-50200”或“50100-50200,50300-50400,51234” 默认是9067.
* query.server.network-threads： 网络线程数，用于接收状态服务的输入请求（0>=#solt）
* query.server.query-threads：状态服务输入请求的线程数。
  + - * 1. Proxy
* query.proxy.ports：
* query.proxy.network-threads:
* query.proxy.query-threads :

注意：作用同上，一个服务端一个客户端（代理端）

* + - 1. Limitaions(限制)
* Queryable 的生命周期和job的什么中期绑定。
  + 1. State Backends(状态恢复)

Flink 提供了不同的 state Backends，它能如何以及在哪里存储state.

State 可以存储在java的堆内存和非堆内存中。根据你的state backends ，Flink为应用程序管理state,这意味着Flink会管理内存(如果有需要可能溢写到磁盘)并允许应用程序持有非常大的state。默认情况下 flink-conf.yaml的配置决定了所有Flink Job的 state backed.

然而，默认状态可能在每个不同的Job中被覆盖。像下面这样。

关于有效的state backends 信息，它更高级的，限制以及配置参数查看[Deployment & Operations](https://ci.apache.org/projects/flink/flink-docs-release-1.6/ops/state/state_backends.html).

|  |
| --- |
| Java 代码 |
| StreamExecutionEnvironment env **=** StreamExecutionEnvironment**.**getExecutionEnvironment**();**  env**.**setStateBackend**(...);** |

|  |
| --- |
| Scala 代码 |
| **val** env **=** **StreamExecutionEnvironment.**getExecutionEnvironment**()**  env**.**setStateBackend**(...)** |

* + 1. Custom Serialization

如果你的应用程序使用Flink来管理state, 可能需要为特殊的场景实现自定义的序列化逻辑。

这篇文章的目的是需要为他们的状态实现自定义序列化逻辑的用户提供帮助指南，覆盖了如何提供自定义序列化和如何升级到一个可兼容的序列化。如果你是简单的使用Flink 自己的序列化，这篇文章是不相干的，你可以跳过。

* + - 1. Using custom serializers（自定义序列化）

在上面的演示中，当注册一个managed operator和keyed state时，StateDescriptor 必须制定状态的名称，既状态类型相关的信息。这个类型信息（type information）通过使用Flink [type serialization framework](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/types_serialization.html) 为状态创建合适的序列化器。

它也有可能完全抛弃Flink的序列化框架而让Flink使用你自己自定义的序列化器去序列化Managed states，简单的直接实现你的TypeSerializer 。

|  |
| --- |
| Java 代码 |
| **public** **class** **CustomTypeSerializer** **extends** TypeSerializer**<**Tuple2**<**String**,** Integer**>>** **{...};**  ListStateDescriptor**<**Tuple2**<**String**,** Integer**>>** descriptor **=**  **new** ListStateDescriptor**<>(**  "state-name"**,**  **new** **CustomTypeSerializer());**  checkpointedState **=** getRuntimeContext**().**getListState**(**descriptor**);** |

|  |
| --- |
| Scala 代码 |
| **class** **CustomTypeSerializer** **extends** **TypeSerializer[(String**, **Integer)]** **{...}**  **val** descriptor **=** **new** **ListStateDescriptor[(String**, **Integer)](**  "state-name"**,**  **new** **CustomTypeSerializer)**  **)**  checkpointedState **=** getRuntimeContext**.**getListState**(**descriptor**)** |

注:Flink 写状态序列化用状态当做元数据。在restore上的某些案例中，写序列化需要能被反序列化。因此，当你的状态序列化时推荐避免去使用匿名类。匿名类对生成的类名没有保证，他很容易引起写序列化的不可靠性。

* + - 1. Handing serializer upgrades and compatibility(序列化升级和兼容)

Flink 允许改变使用的序列化器去读写manged state,所以不需要锁定任何特殊的序列化器。当一个状态恢复时，新的序列化器为状态注册并检查其兼容性并为状态替换新的序列化器。

兼容的序列化器意味着这个序列化有能力读取先前状态字节的序列化 以及能够写入状态二进制格式。这意味着检查新的序列化兼容性通过TypeSerializer 接口的下面两个方法提供：

|  |
| --- |
| **public** **abstract** TypeSerializerConfigSnapshot **snapshotConfiguration();**  **public** **abstract** CompatibilityResult **ensureCompatibility(**TypeSerializerConfigSnapshot configSnapshot); |

简而言之，每次执行checkpoint，snapshotConfiguration 方法被调用去创建一个状态序列化的point-in-time视图。返回的配置快照和检查点一起作为状态元数据被存储。当使用checkpoint去恢复job时，序列化配置快照将提供相同状态新序列化方法，ensureCompatibility校验新序列化器的兼容性。该方法用于检查新的序列化器是否兼容，以及在不兼容的情况下重新配置序列化器钩子。

注意Flink 自己的序列化器的实现至少需要时间他们自己。当相同的序列化在恢复Job中被使用，序列化将重新配置它们自己去兼容它们先前的配置。

下面的小节阐述了当使用自定义序列化器时实现这两个方法的一个简要指南。

* + - * 1. 实现**snapshotConfiguration 方法**

序列化配置快照在恢复需要获取足够的信息，这些信息保持了新序列化是否兼容或不兼容的状态信息。典型的包含关于序列化参数信息或序列化数据的二进制格式。通常，新的序列化器是否允许它使用之前的序列化字节，并将使用相同的二进制格式写入。

序列化配置快照如何读写完全是自定义的。下面的基类被所有的序列化配置快照实现，这个类是TypeSerializerConfigSnapshot

|  |
| --- |
| **public** **abstract** TypeSerializerConfigSnapshot **extends** VersionedIOReadableWritable **{**  **public** **abstract** **int** **getVersion();**  **public** **void** **read(**DataInputView in**)** **{...}**  **public** **void** **write(**DataOutputView out**)** **{...}**  **}** |

read 和 write 方法定义了如何从checkpoint配置如何读写。基本的实现逻辑包含读写配置快照的版本，所以应该是被扩展而不应该完全覆盖。

配置快照的版本是通过getVersion()方法确定的。Versioning 是序列化配置快照维持兼容性的配置，因为包含在配置里面的信息可能随着时间的迁移而变化。默认情况下，配置快照仅兼容当前版本（通过getVersion返回）。为了指向配置兼容的其他版本，重写getCompatibleVersions 方法返回更多的版本值，当你读取数据的时候，你可以使用getReadVersion 确定写的配置版本并使用指定的版本确定read逻辑。

注意：序列化配置快照的版本对升级序列化是不想干的。抽取相同的序列化可以有不同的配置快照实现，例如，更多的信息添加到配置在未来允许更多综合性兼容检查。

实现TypeSerializerConfigSnapshot的一个限制是空的构造器必须被显示声明，当从checkpoint读取配置快照的时候空的构造器是必须的。

* + - * 1. 实现ensureCompatibility方法

ensureCompatibility 需要包含执行核查关于之前序列化器携带的TypeSerializerConfigSnapshot信息的逻辑。最基本的逻辑之一是：

* 检查序列化是否兼容，同时可能重新配置（如果需要）它自己使它兼容。然后，Flink承认序列化是兼容的。
* Flink在使用新序列化器处理时承认序列化是不兼容的并迁移状态是必须的。

上面的例子可以通过转换ensureCompatibility 方法下面的任意一个进行代码转换。

* **CompatibilityResult.compatible()：**认可新序列化是兼容的，或已经配置被兼容，并且Flink可以使用序列化器处理Job
* **CompatibilityResult.requiresMigration()**: 序列化是不兼容的，或者说不能被重新注册为兼容，在新序列化被使用前必须要状态迁移。状态迁移是通过使用之前是序列化读取状态字节转变为对象来执行的，然后使用新的序列化器进行序列化。
* **CompatibilityResult.requiresMigration(TypeDeserializer deserializer)**:它和CompatibilityResult.requiresMigration()具有相同的语义，但是，在无法找到先前的序列化器或加载该序列化器以读取迁移的恢复状态字节的情况下，可以使用提供的类型反序列化器作为后备方案。
  + - 1. Managed TypeSerializer and TypeSerializerConfigSnapshot classes in user code

## Operator

* + 1. Overview

Operators 转换一个或多个DataStreams到一个新的DataStream.程序可以合并多一个transformations 到一个复杂的数据流拓扑图。

本小节将描述基本的transformations，在应用这些operator后也可以对Flink的operator chaining 进行有效的物理分区。

|  |
| --- |
| **DataStream Transformations** |
| **Physical partitioning** |
| **Task chaining and resource groups** |

* + - 1. DataStream Transformations

|  |  |
| --- | --- |
| Scala 版本 | |
| **Transformation（转换）** | **Description(描述)** |
| **Map**  DataStream -> DataStream | 获取一个元素并产生一个元素。Map函数对输入的数据翻倍：  dataStream**.**map **{** x **=>** x **\*** 2 **}** |
| **FlatMap**  DataStream -> DataStream | 获取一个元素并产生0个，1个或多个元素。flatmap 函数切分语句：  dataStream**.**flatMap **{** str **=>** str**.**split**(**" "**)** **}** |
| **Filter** DataStream → DataStream | 过滤 |
| **KeyBy** DataStream → KeyedStream | 逻辑上分区一个流到多个互不相干的分区，每个分区的元素都具有相同的key.在内部，使用hash 分区。参阅[keys](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/api_concepts.html#specifying-keys) 如何去指定key。这个transformation返回一个KeyedStream  dataStream**.**keyBy**(**"someKey"**)** *// Key by field "someKey"*  dataStream**.**keyBy**(**0**)** *// Key by the first element of a Tuple* |
| **Reduce** KeyedStream → DataStream | 滚动化简keyed 数据流，用当前值和最新值合并元素并得到一个新的值  Reduce function 函数局部求和  keyedStream**.**reduce **{** **\_** **+** **\_** **}** |
| **Fold** KeyedStream → DataStream | 使用初始值折叠data stream. 用当前值和最新值合并元素并得到一个新的值  一个折叠函数，当应用在序列（1,2,3,4,5）发送顺序"start-1", "start-1-2", "start-1-2-3", ...  **val** result**:** **DataStream[String]** **=**  keyedStream**.**fold**(**"start"**)((**str**,** i**)** **=>** **{** str **+** "-" **+** i **})** |
| **Aggregations** KeyedStream → DataStream | 聚合，min和minBy不同在于min返回最小值，而minBy返回这个字段中最小值的元素。  keyedStream**.**sum**(**0**)**  keyedStream**.**sum**(**"key"**)**  keyedStream**.**min**(**0**)**  keyedStream**.**min**(**"key"**)**  keyedStream**.**max**(**0**)**  keyedStream**.**max**(**"key"**)**  keyedStream**.**minBy**(**0**)**  keyedStream**.**minBy**(**"key"**)**  keyedStream**.**maxBy**(**0**)**  keyedStream**.**maxBy**(**"key"**)** |
| **Window** KeyedStream → WindowedStream | Window 被定义在已经分区的KeyedStream上。Window 在每个key中根据相同的一些特征来分组数据。（如：数据在5秒内到达），详见[windows](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html)  dataStream**.**keyBy**(**0**).**window**(TumblingEventTimeWindows.**of**(Time.**seconds**(**5**)))** *// Last 5 seconds of data* |
| **WindowAll** DataStream → AllWindowedStream | Window定义在一个定期的DataStream上。Window根据一些特征分组所有的数据（如：数据在5秒内到达）详见[windows](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html)  警告：在大多数场景中它是**non-parallel 转换（**transformation**）**。所有的数据为了完成WindowAll 操作必须聚集到一个任务中执行。  dataStream**.**windowAll**(TumblingEventTimeWindows.**of**(Time.**seconds**(**5**)))** *// Last 5 seconds of data* |
| **Window Apply** WindowedStream → DataStream AllWindowedStream → DataStream | 一个通用函数应用于整个窗口，下面的函数手动对窗口元素进行求和。  **注意**： 如果你使用windowAll 函数，你需要使用AllWindowFunction 替换。  windowedStream**.**apply **{** **WindowFunction** **}**  *// applying an AllWindowFunction on non-keyed window stream*  allWindowedStream**.**apply **{** **AllWindowFunction** **}** |
| **Window Reduce** WindowedStream → DataStream | 应用一个reduce函数到window 并返回化简的value  windowedStream**.**reduce **{** **\_** **+** **\_** **}** |
| **Window Fold** WindowedStream → DataStream | 在window上使用一个折叠函数。  **val** result**:** **DataStream[String]** **=**  windowedStream**.**fold**(**"start"**,** **(**str**,** i**)** **=>** **{** str **+** "-" **+** i **})** |
| **Aggregations on windows** WindowedStream → DataStream | 对窗口内容进行聚合。  windowedStream**.**sum**(**0**)**  windowedStream**.**sum**(**"key"**)**  windowedStream**.**min**(**0**)**  windowedStream**.**min**(**"key"**)**  windowedStream**.**max**(**0**)**  windowedStream**.**max**(**"key"**)**  windowedStream**.**minBy**(**0**)**  windowedStream**.**minBy**(**"key"**)**  windowedStream**.**maxBy**(**0**)**  windowedStream**.**maxBy**(**"key"**)** |
| **Union** DataStream\* → DataStream | 联合超过2个的data streams 去创建一个新的stream，这个stream包含了来自于所有流的所有元素。  注意：如果你UNION的数据流是他自己那么你在结果流中每个元素都会获得2次。  dataStream**.**union**(**otherStream1**,** otherStream2**,** **...)** |
| **Window Join** DataStream,DataStream → DataStream | 在一个窗口和给定的key上join两个数据流。  dataStream**.**join**(**otherStream**)**  **.**where**(<**key selector**>).**equalTo**(<**key selector**>)**  **.**window**(TumblingEventTimeWindows.**of**(Time.**seconds**(**3**)))**  **.**apply **{** **...** **}** |
| **Window CoGroup** DataStream,DataStream → DataStream | 在一个窗口和给定的key上CoGroup两个数据流。  dataStream**.**coGroup**(**otherStream**)**  **.**where**(**0**).**equalTo**(**1**)**  **.**window**(TumblingEventTimeWindows.**of**(Time.**seconds**(**3**)))**  **.**apply **{}** |
| **Connect** DataStream,DataStream → ConnectedStreams | “Connect”两个数据流并保留他们的类型，在两个流之间在允许共享状态。  someStream **:** **DataStream[Int]** **=** **...**  otherStream **:** **DataStream[String]** **=** **...**  **val** connectedStreams **=** someStream**.**connect**(**otherStream**)** |
| **CoMap, CoFlatMap** ConnectedStreams → DataStream | 类似于一个在connected 数据流上的map和flatMap  connectedStreams**.**map**(**  **(\_** **:** **Int)** **=>** **true,**  **(\_** **:** **String)** **=>** **false**  **)**  connectedStreams**.**flatMap**(**  **(\_** **:** **Int)** **=>** **true,**  **(\_** **:** **String)** **=>** **false**  **)** |
| **Split** DataStream → SplitStream | 根据某些标准切分流到2个或更多的流  **val** split **=** someDataStream**.**split**(**  **(**num**:** **Int)** **=>**  **(**num **%** 2**)** **match** **{**  **case** 0 **=>** **List(**"even"**)**  **case** 1 **=>** **List(**"odd"**)**  **}**  **)** |
| **Select** SplitStream → DataStream | 从Spilt Stream中选择一个或多个流  **val** even **=** split select "even"  **val** odd **=** split select "odd"  **val** all **=** split**.**select**(**"even"**,**"odd"**)** |
| **Iterate** DataStream → IterativeStream → DataStream | 通过重定向一个operator的输出到一些之前的operator在flow中创建一个循环的“feedback”.这对于持续更新模型算法特别有用。下面的代码刚开始启用了一个stream并在主题部分采用了迭代。每个大于0的元素本发送到feedback 渠道，剩下的元素指向下游。见[iterations](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/#iterations)  initialStream**.**iterate **{**  iteration **=>** **{**  val iterationBody **=** iteration**.**map **{***/\*do something\*/***}**  **(**iterationBody**.**filter**(**\_ **>** 0**),** iterationBody**.**filter**(**\_ **<=** 0**))**  **}**  **}** |
| **Extract Timestamps** DataStream → DataStream | 从记录中抽取时间戳，它可以在event time语义的窗口中使用。详见[Event Time](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_time.html).  stream**.**assignTimestamps **{** timestampExtractor **}** |

下面的transformations 仅在元组的数据流上有效

|  |  |
| --- | --- |
| Java | |
| **Transformation** | **Description** |
| **Project** DataStream → DataStream | 选择元组字段子集  DataStream**<**Tuple3**<**Integer**,** Double**,** String**>>** in **=** *// [...]*  DataStream**<**Tuple2**<**String**,** Integer**>>** out **=** in**.**project**(**2**,**0**);** |

* + - 1. Physical partitioning

在一个transformation后Flink也提供了对stream 分区更底层的控制。通过下面的函数：

|  |  |
| --- | --- |
| Scala | |
| **Transformation** | **Description** |
| **Custom partitioning** DataStream → DataStream | 使用用户自定义的分区为每个元素选择目标task  dataStream**.**partitionCustom**(**partitioner**,** "someKey"**);**  dataStream**.**partitionCustom**(**partitioner**,** 0**);** |
| **Random partitioning** DataStream → DataStream | 均匀的随机划分元素  dataStream**.**shuffle**();** |
| **Rebalancing (Round-robin partitioning)** DataStream → DataStream | 为每个分区均衡的划分元素，这对于数据倾斜出现的时候是非常有用的。  dataStream**.**rebalance**();** |
| **Rescaling** DataStream → DataStream | 元素分区到下游operator的子集。如果你想在你想要的地方放置管道，这特别有用，例如。从一个源的每个并行实例扇出到几个映射器的子集，以分发负载，但是又不希望进行完全的均衡。它必须是本地数据传输替代网络上的数据传输，根据配置的值 比如 Taskmanagers 的slots数量  上游操作向其发送元素的下游操作子集取决于上下游操作的并行度。例如，如果上游operation有2个并行度，下游operation的并行度是6，那么其中一个上游operator将分发元素到3个下游operation，而其他的上游operator也将分化到3个下游的operation. 另一方面，下游operation只有2个并行度而上游操作有6个并行度，那么3个上游operation的元素分发到一个下游operation，其他的3个上游operation分发到另一个下游的operation.  如果上下游operation的并行度不是彼此的倍数，那么他们之间会有不同的数量映射  上面的例子详见下面的可视化图形：    dataStream**.**rescale**();** |
| **Broadcasting** DataStream → DataStream | 广播元素到每个分区  dataStream**.**broadcast**();** |

* + - 1. Task chaining and resource groups

Chaining是两个连续的transformations 为了更好的性能而降他们放置到相同的线程当中。如果有可能的话（如：两个连续的map 转换）Flink默认执行chains 操作。如果你需要的话API给了一些细粒度在chain上的控制。

如果你想在你的整个JOB中禁用chaining的话你可以使用StreamExecutionEnvironment.disableOperatorChaining()  方法。一些更加细粒度的控制，下面的函数都是有效的。注意：这些函数只有在DataStream 转换作为他们引用的上一个转换的时候才能被正确使用。例如，你能使用someStream.map(...).startNewChain()，但是你不能使用someStream.startNewChain()。

在Flink中资源组是一个slot,如果你想的话你可以手动在独立的solt中隔离operation

|  |  |
| --- | --- |
| **Scala** | |
| **Transformation** | **Description** |
| Start new chain | 开始一个新的chain,用这个operator启动。两个mapper会被chain在一起，filter没有被第一个mapper chain在一起。  someStream**.**filter**(...).**map**(...).**startNewChain**().**map**(...)** |
| Disable chaining | Operator禁用chain  someStream**.**map**(...).**disableChaining**()** |
| Set slot sharing group | 为operator设置共享slot. Flink会将具有相同slot共享组的operator放到相同的slot中，同时保留其他slot中没有slot共享组的operator. 如果输入的operation在相同的slot共享组中，那么这个slot共享组将继承这个输入的operation。默认slot共享组的名称是“default”,operation通过调用slotSharingGroup("default") 将其放入组中。  someStream**.**filter**(...).**slotSharingGroup**(**"name"**)** |

* + 1. Windows

窗口是处理无线流的核心。窗口切分stream到一个有限大小的“buckets(桶)”中，在这个上面我们可以进行计算。本文档聚焦于Flink的窗口是如何执行的以及程序员是如何从Flink提供的功能中如何受益的。

Flink窗口程序典型的结构就像下面呈现的一样。第一部分代码片段涉及到keyed streams.而第二部分是non-keyed 的一种，他们的区别仅仅是为keyedStream调用了keyBy（…）方法而为non-keyed Stream调用了window(…) 方法。这也将作为其他部分讲解的主要路线图。

**Keyed Windows**

stream

.keyBy(...) <- keyed versus non-keyed windows

.window(...) <- required: "assigner"

[.trigger(...)] <- optional: "trigger" (else default trigger)

[.evictor(...)] <- optional: "evictor" (else no evictor)

[.allowedLateness(...)] <- optional: "lateness" (else zero)

[.sideOutputLateData(...)] <- optional: "output tag" (else no side output for late data)

.reduce/aggregate/fold/apply() <- required: "function"

[.getSideOutput(...)] <- optional: "output tag"

**Non-Keyed Windows**

stream

.windowAll(...) <- required: "assigner"

[.trigger(...)] <- optional: "trigger" (else default trigger)

[.evictor(...)] <- optional: "evictor" (else no evictor)

[.allowedLateness(...)] <- optional: "lateness" (else zero)

[.sideOutputLateData(...)] <- optional: "output tag" (else no side output for late data)

.reduce/aggregate/fold/apply() <- required: "function"

[.getSideOutput(...)] <- optional: "output tag"

在上面的案例中，方括号中的内容是可选的。这说明Flink有多种不同的方式允许你自定义window的窗口逻辑。

|  |
| --- |
| **Window Lifecycle （窗口的生命周期）** |
| **Keyed vs Non-Keyed Stream** |
| **Window Assigners** |
| Tumbling Windows |
| Sliding Windows |
| Session Windows |
| Global Windows |
| **Window Functions** |
| ReduceFunction |
| AggregateFunction |
| FlodFunction |
| ProcessWindowFunction |
| ProcessWindowFunction with Incremental Aggregation |
| Using per-window state in ProcessWindowFunction |
| WindowFunction (Legacy) |
| **Triggers(触发器)** |
| Fire and Purge |
| Default Trigger of WindowAssigners |
| Built-in and Custom Triggers |
| **Evictors(驱逐器)** |
| Allowed Lateness |
| Getting late data as a side output |
| Late elements considerations |
| **Working with window results** |
| Interaction of watermark and windows |
| Consecutive windowed operations |
| **Useful state size considerations** |

* + - 1. Window Lifecycle(窗口的生命周期)

只要一个元素在属于这个窗口内到达那么这个窗口就会被创建，当时间超过指定时间戳加上允许的延迟（ 见[Allowed Lateness](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#allowed-lateness)）后窗口就会被完整的移除。Flink 确保只对基于时间窗口的进行删除而不对其他类型的窗口进行删除，比如：global window(见[Window Assigners](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#window-assigners))。例如，一个5分钟的窗口并且允许延迟一分钟的事件时间窗口策略，当第一个元素在这个时间戳频率内到达的时，Flink会创建一个12:00到12:05的窗口。并且当水位超过12:06分的时候Flink才会移除它。

此外，每个窗口都会有一个触发器（见[Triggers](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#triggers)) 和一个函数（ProcessWindowFunction, ReduceFunction,AggregateFunction or FoldFunction）（见 [Window Functions](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#window-functions)） 附加在它上面。这个函数包含的计算逻辑将会被应用在窗口的内容上，而出指定了窗口是否考虑被应用这些函数的条件。一个触发器的策略可能类似于“当一个元素的数量在窗口中大于4的时候” 或“当水位超过了window的最后大小时候”。触发器可以在任何时间的窗口创建和删除时是否需要删除窗口内容。清除仅涉及数据的内容而不会涉及到元数据。这意味着新的数据依旧可以被添加到这个窗口中。

下面我们将更详细的讲述上面的内容。

* + - 1. Keyed and Non-keyed Windows

第一件事就是去指定你的stream是否应该被keyed还是不需要被keyed.它在窗口定义之前就需要被完成。使用keyBy(…)将切分无限stream到逻辑的 keyed streams。如果keyBy(…)没有被调用。那么你的stream是没有被keyed的。

在keyed stream的场景中，传入数据的任何属性都可以做为key(详见 [here](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/api_concepts.html#specifying-keys))。 Keyed stream 允许你在多个任务中并行计算，以你为每个逻辑keyed stream 相对于其他keyed stream 独立处理。所有具有相同key的元素都会被发送到同一个并行任务中。

在non-keyed stream的场景中，你的原始流不会被切分到多个逻辑流并且所有的窗口逻辑都将在当个任务中执行（并行度为1）

* + - 1. Window Assigners

在指定stream是否需要keyed后，下一步就是定义窗口的window assigner。 window assigner 定义了元素如何被分配到窗口中。为了完成这个目的，你需要在你的代码中选择window(…)或windowAll(…)中指定WindowAssigner 。

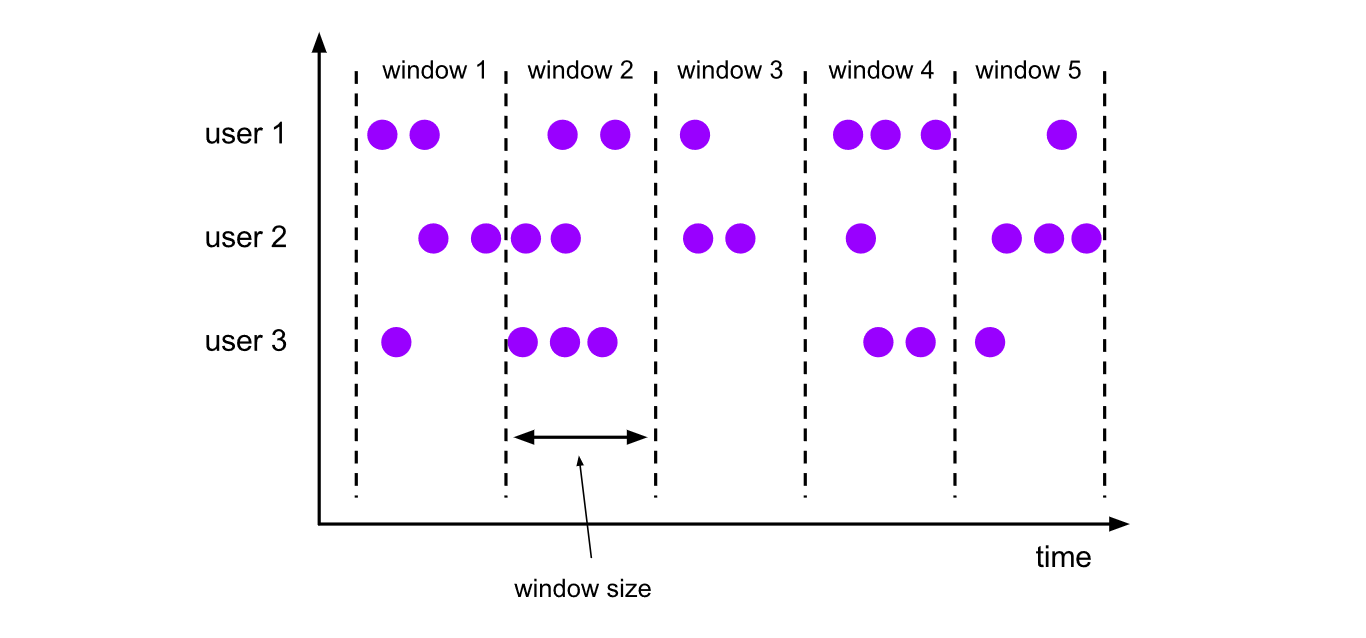
WindowAssigner 负责分输入的元素到一个或多个窗口中。Flink已经预先定义了一些最常用的WindowAssigner ，他们是 tumbling windows, sliding windows, session windows 和 global windows。通过扩展WindowAssigner 可以实现自定义的WindowAssigner 。所有的构建的window assigners（除global assigner外）分配元素都是基于时间的，这个时间既可以是event time 也可以是processing time. 请花一点点时间阅读[event time](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_time.html) 这一小节，它介绍了 processing time 和event time 的区别以及如何生成水位。

基于时间的窗口以一个开始时间（包含）和一个结束时间（排除），他们的区间描述了窗口的大小。在代码中，当使用基于时间的窗口工作的时Flink使用了TimeWindow ，TimeWindow 提供了查询开始和结束时间戳的方法，并且他也提供了maxTimestamp方法，这个方法返回窗口最大允许的时间戳。

下面，我们将展示Flink预先定义的window assigner如何工作和在DataStream程序中如何使用它们。下面的图标展示了每个分配器工作的情况。紫色的圆圈代表了流的元素，他可以通过一些key进行分区。（这个案例中是user1 ,user2 和 user3）,X轴标识时间的进度。

* + - * 1. Tumbling Windows (翻滚窗口)

Thmbling windows 分配器将分配每个元素到一个固定大小的窗口。Thmbling windows 有一个固定的大小并且不会重叠。例如，如果指定的Thmbling windows 大小是5分钟，flink将每5分钟启动一个窗口，就像下入阐述的一样：



下面的代码片段展示了如何使用翻滚窗口

|  |
| --- |
| Scala 代码 |
| **val** input**:** **DataStream[T]** **=** **...**  *// tumbling event-time windows*  input  **.**keyBy**(<**key selector**>)**  **.**window**(TumblingEventTimeWindows.**of**(Time.**seconds**(**5**)))**  **.<**windowed transformation**>(<**window function**>)**  *// tumbling processing-time windows*  input  **.**keyBy**(<**key selector**>)**  **.**window**(TumblingProcessingTimeWindows.**of**(Time.**seconds**(**5**)))**  **.<**windowed transformation**>(<**window function**>)**  *// daily tumbling event-time windows offset by -8 hours.*  input  **.**keyBy**(<**key selector**>)**  **.**window**(TumblingEventTimeWindows.**of**(Time.**days**(**1**),** **Time.**hours**(-**8**)))**  **.<**windowed transformation**>(<**window function**>)** |

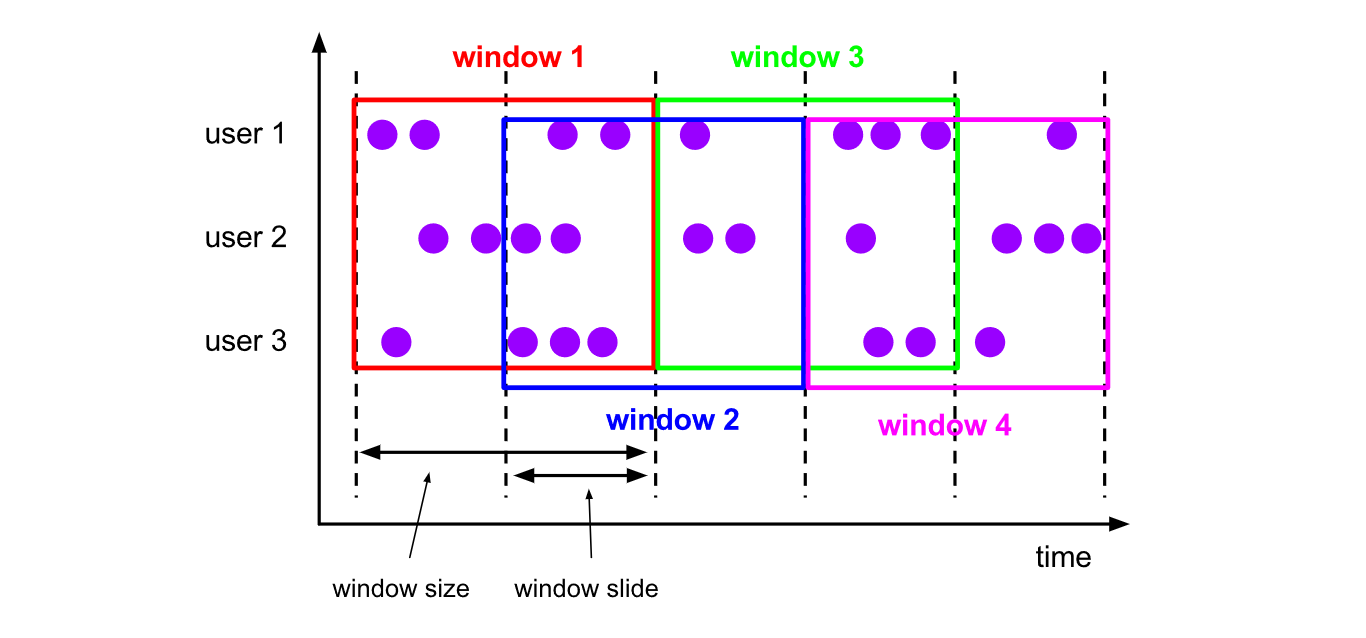
时间的频率使用Time.milliseconds(x), Time.seconds(x), Time.minutes(x)等方法来指定。

在上面展示的案例中，翻滚窗口分配器也能指定一个可选的offset参数，这个参数可用于改变窗口的对齐时间。例如，一个小时的翻滚窗口在没有offset时，你得到的窗口像这样1:00:00.000 - 1:59:59.999, 2:00:00.000 - 2:59:59.999等。如果你想通过给定的offset改变窗口。你能使用一个15分钟的offset ，例如，你将得到1:15:00.000 - 2:14:59.999, 2:15:00.000 - 3:14:59.999 等。使用偏移量一个比较重要的场景时区（默认为UTC-0）。例如，在中国你需要指定offset为Time.hours(-8).

* + - * 1. Sliding Windows(滑动窗口)

*Sliding windows* 分配器分配元素到固定长度的窗口。和*Tumbling window* 分配器类似，窗口的大小是通过*window size* 参数设置的，另外添加一个 window slide参数来控制窗口如何滑动。因此，如果滑动尺寸小于窗口尺寸的时候是*Sliding windows*会被重叠的。在这个用例中元素被分配到多个窗口中。

例如，你的窗口大小是十分钟而滑动大小是5分钟。窗口每5分钟生成一个新的窗口但是包含了最新10分钟的数据，如下图：



下面的代码片段展现了如何使用滑动窗口

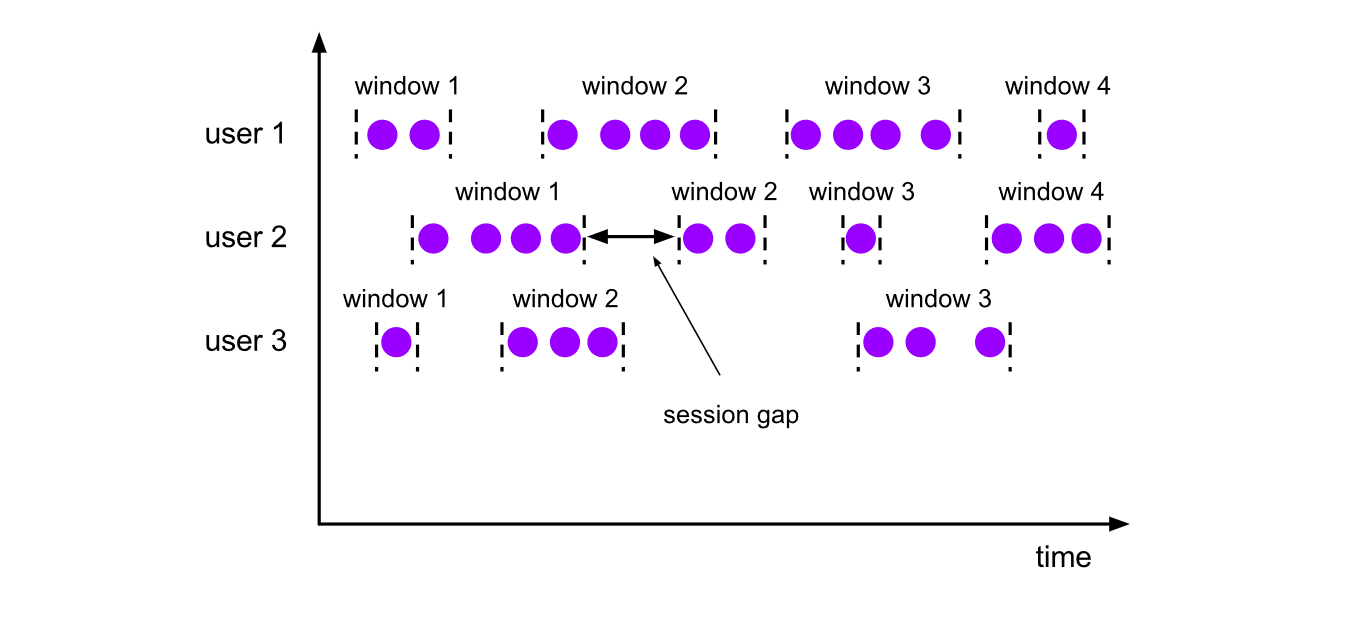
|  |
| --- |
| Scala |
| **val** input**:** **DataStream[T]** **=** **...**  *// sliding event-time windows*  input  **.**keyBy**(<**key selector**>)**  **.**window**(SlidingEventTimeWindows.**of**(Time.**seconds**(**10**),** **Time.**seconds**(**5**)))**  **.<**windowed transformation**>(<**window function**>)**  *// sliding processing-time windows*  input  **.**keyBy**(<**key selector**>)**  **.**window**(SlidingProcessingTimeWindows.**of**(Time.**seconds**(**10**),** **Time.**seconds**(**5**)))**  **.<**windowed transformation**>(<**window function**>)**  *// sliding processing-time windows offset by -8 hours*  input  **.**keyBy**(<**key selector**>)**  **.**window**(SlidingProcessingTimeWindows.**of**(Time.**hours**(**12**),** **Time.**hours**(**1**),** **Time.**hours**(-**8**)))**  **.<**windowed transformation**>(<**window function**>)** |

时间的频率可以通过 Time.milliseconds(x), Time.seconds(x), Time.minutes(x)中任意一个方法来指定。

正如案例中说展现的一样，滑动窗口也可以指定一个offset参数，这个参数可用于改变窗口的对齐时间。例如，一个小时的翻滚窗口在没有offset时，你得到的窗口像这样1:00:00.000 - 1:59:59.999, 2:00:00.000 - 2:59:59.999等。如果你想通过给定的offset改变窗口。你能使用一个15分钟的offset ，例如，你将得到1:15:00.000 - 2:14:59.999, 2:15:00.000 - 3:14:59.999 等。使用偏移量一个比较重要的场景时区（默认为UTC-0）。例如，在中国你需要指定offset为Time.hours(-8).

* + - * 1. Session Windows(会话窗口)

Session window 通过一个活动的会话分配一组元素。相对tumbling windows 和 sliding windows . Session windows 既不会重叠也没有明确的开始和结束时间。Session windows窗口的关闭通过某一个确定的周期内不再接收任何元素。Session window分配器的配置要么通过一个固定的session 间隔要么充函数中抽取时间频率，这个函数定义了如何获取周期。当这个周期过期，对当前的session窗口关闭，接下来的元素会被分配到一个新的窗口中。



下面的代码展示了如何使用Session window

|  |
| --- |
| Scala 代码 |
| **val** input**:** **DataStream[T]** **=** **...**  *// event-time session windows with static gap*  input  **.**keyBy**(<**key selector**>)**  **.**window**(EventTimeSessionWindows.**withGap**(Time.**minutes**(**10**)))**  **.<**windowed transformation**>(<**window function**>)**  *// event-time session windows with dynamic gap*  input  **.**keyBy**(<**key selector**>)**  **.**window**(EventTimeSessionWindows.**withDynamicGap**(new** **SessionWindowTimeGapExtractor[String]** **{**  **override** **def** extract**(**element**:** **String):** **Long** **=** **{**  *// determine and return session gap*  **}**  **}))**  **.<**windowed transformation**>(<**window function**>)**  *// processing-time session windows with static gap*  input  **.**keyBy**(<**key selector**>)**  **.**window**(ProcessingTimeSessionWindows.**withGap**(Time.**minutes**(**10**)))**  **.<**windowed transformation**>(<**window function**>)**  *// processing-time session windows with dynamic gap*  input  **.**keyBy**(<**key selector**>)**  **.**window**(DynamicProcessingTimeSessionWindows.**withDynamicGap**(new** **SessionWindowTimeGapExtractor[String]** **{**  **override** **def** extract**(**element**:** **String):** **Long** **=** **{**  *// determine and return session gap*  **}**  **}))**  **.<**windowed transformation**>(<**window function**>)** |

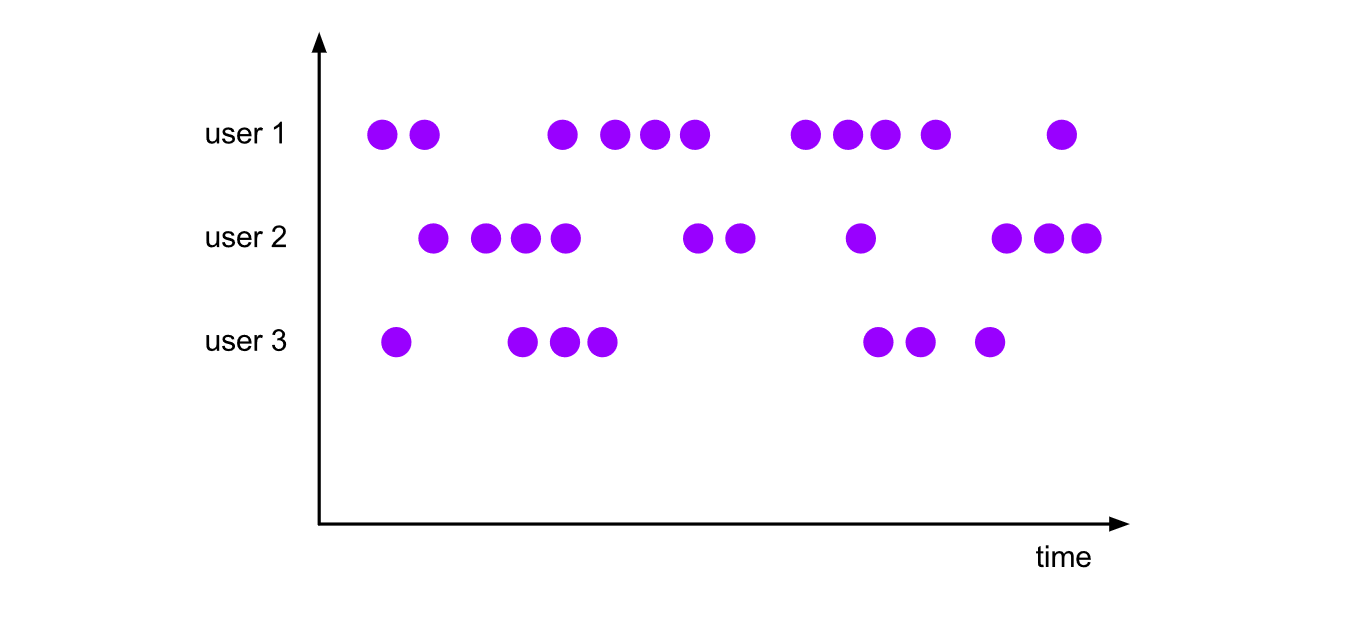
静态间隙可以通过 Time.millisec onds(x), Time.seconds(x), Time.minutes(x)任意一个方法指定

动态间隙需要通过实现SessionWindowTimeGapExtractor 接口来指定。

注意：由于session windows 没有固定的开始和结束时间，那么他和tumbling，sliding窗口不同之处在于对窗口大小的控制。在Flink 内部，sesssion window 将为每次到达的记录都创建要给新的窗口并且只要他们之间比定义的窗口间隙更近那么就会将他们合并在一起。为了可以被合并，session window 必须定义一个合并的[Trigger](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#triggers) 和合并的[Window Function](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#window-functions)。如 ReduceFunction, AggregateFunction, 活 ProcessWindowFunction (FoldFunction不能进行合并.)

* + - * 1. Global Windows(全局窗口)

 global windows 分配器使用相同的key分配所有的元素到单个 global windows . 这种窗口模式只有在你指定了自定义触发器的时候才有用。否则，计算将不会被执行，由于 global windows 没有一个结束点，所以我们可以处理聚合元素。



下面的代码片段展示了 global windows 的使用方式

|  |
| --- |
| Scala 代码 |
| **val** input**:** **DataStream[T]** **=** **...**  input  **.**keyBy**(<**key selector**>)**  **.**window**(GlobalWindows.**create**())**  **.<**windowed transformation**>(<**window function**>)** |

* + - 1. Window Functions

定义了window assigner之后，我们需要指定窗口执行计算的逻辑。负责处理这个操作的是 window function,它用于对已经确定的窗口对窗口中已经准备好的元素进行处理。

窗口函数可以是ReduceFunction, AggregateFunction, FoldFunction or ProcessWindowFunction中的任意一个。前面两个函数更有效，因为Flink会为到达的元素递增的聚合元素。而ProcessWindowFunction 将为窗口内的所有元素获得一个迭代器以及所有元素所属窗口的元信息。

* + - * 1. ReduceFunction

ReduceFunction指定了输入端的两个元素合并后如何去产生具有相同类型的输出元素。Flink 使用ReduceFunction 增量聚合窗口内的元素。

ReduceFunction 的定义看上去想下面这样：

|  |
| --- |
| Scala 代码 |
| **val** input**:** **DataStream[(String**, **Long)]** **=** **...**  input  **.**keyBy**(<**key selector**>)**  **.**window**(<**window assigner**>)**  **.**reduce **{** **(**v1**,** v2**)** **=>** **(**v1**.**\_1**,** v1**.**\_2 **+** v2**.**\_2**)** **}** |

上面的例子是对窗口内所有元素的元组中第二个字段进行求和。

* + - * 1. AggregateFunction

AggregateFunction 是ReduceFunction的通用版本，它有三种类型：输入类型（IN）,累加器类型（ACC）和输出类型（OUT）。输入类型是输入流元素的类型，AggregateFunction 为添加一个元素到累加器提供了一个方法。这个接口也提供了初始化累加器的方法，并从两个累加器合并成一个累加器或从一个累加器中提取输出类型。在下面的案例中我们将看到它是如何工作的。

和ReduceFunction相同，Flink将增量聚合到达窗口内的元素。

AggregateFunction像如下定义：

|  |
| --- |
| Scala 代码 |
| */\*\**  *\* The accumulator is used to keep a running sum and a count. The [getResult] method*  *\* computes the average.*  *\*/*  **class** **AverageAggregate** **extends** **AggregateFunction[(String**, **Long)**, **(Long**, **Long)**, **Double]** **{**  **override** **def** createAccumulator**()** **=** **(**0L**,** 0L**)**  **override** **def** add**(**value**:** **(String,** **Long),** accumulator**:** **(Long,** **Long))** **=**  **(**accumulator**.**\_1 **+** value**.**\_2**,** accumulator**.**\_2 **+** 1L**)**  **override** **def** getResult**(**accumulator**:** **(Long,** **Long))** **=** accumulator**.**\_1 **/** accumulator**.**\_2  **override** **def** merge**(**a**:** **(Long,** **Long),** b**:** **(Long,** **Long))** **=**  **(**a**.**\_1 **+** b**.**\_1**,** a**.**\_2 **+** b**.**\_2**)**  **}**  **val** input**:** **DataStream[(String**, **Long)]** **=** **...**  input  **.**keyBy**(<**key selector**>)**  **.**window**(<**window assigner**>)**  **.**aggregate**(new** **AverageAggregate)** |

上面的例子计算平均值

* + - * 1. FoldFunction

FoldFunction 指定了窗口的输入元素如何和窗口的输出元素类型合并。FoldFunction 对于添加到窗口的元素和当前输出的value是增量调用的。第一个元素使用先前输出类型定义的初始值进行合并。

FoldFunction 定义像下面这样：

|  |
| --- |
| Scala 代码 |
| **val** input**:** **DataStream[(String**, **Long)]** **=** **...**  input  **.**keyBy**(<**key selector**>)**  **.**window**(<**window assigner**>)**  **.**fold**(**""**)** **{** **(**acc**,** v**)** **=>** acc **+** v**.**\_2 **}** |

上面的例子是将所有输入的Long 值附加到初始化为空字符串上。

注意：fold()不能在session windows 和其他任何可可并的窗口。

* + - * 1. ProcessWindowFunction

ProcessWindowFunction 将会得到一个在窗口中包含了所有元素的迭代器，以及一个具有访问时间和状态信息的上下文对象，它提供了比其他窗口更灵活的操作。同时他带来了性能和资源的消耗，因为元素不能被增量聚合而是需要缓存在内部，直到窗口考虑进行处理才会释放。

ProcessWindowFunction  看起来像下面这样。

|  |
| --- |
| Scala 代码 |
| **abstract** **class** **ProcessWindowFunction[IN**, **OUT**, **KEY**, **W** **<:** **Window]** **extends** **Function** **{**  */\*\**  *\* Evaluates the window and outputs none or several elements.*  *\* @param key The key for which this window is evaluated.*  *\* @param context The context in which the window is being evaluated.*  *\* @param elements The elements in the window being evaluated.*  *\* @param out A collector for emitting elements.*  *\* @throws Exception The function may throw exceptions to fail the program and trigger recovery.*  *\*/*  **def** process**(**  key**:** **KEY,**  context**:** **Context,**  elements**:** **Iterable[IN],**  out**:** **Collector[OUT])**  */\*\**  *\* The context holding window metadata*  *\*/*  **abstract** **class** **Context** **{**  */\*\**  *\* Returns the window that is being evaluated.*  *\*/*  **def** window**:** **W**  */\*\**  *\* Returns the current processing time.*  *\*/*  **def** currentProcessingTime**:** **Long**  */\*\**  *\* Returns the current event-time watermark.*  *\*/*  **def** currentWatermark**:** **Long**  */\*\**  *\* State accessor for per-key and per-window state.*  *\*/*  **def** windowState**:** **KeyedStateStore**  */\*\**  *\* State accessor for per-key global state.*  *\*/*  **def** globalState**:** **KeyedStateStore**  **}**  **}** |

注意:key参数是从KeySelector 中抽取的出来的。它被keyBy()方法调用。对于tuple-index或string-field引用，这个key类型总是Tuple，您必须手动将其转换为大小正确的元组以提取键字段。

ProcessWindowFunction  定义像如下这样：

|  |
| --- |
| Scala 代码 |
| **val** input**:** **DataStream[(String**, **Long)]** **=** **...**  input  **.**keyBy**(\_.**\_1**)**  **.**timeWindow**(Time.**minutes**(**5**))**  **.**process**(new** **MyProcessWindowFunction())**  **class** **MyProcessWindowFunction** **extends** **ProcessWindowFunction[(String**, **Long)**, **String**, **String**, **TimeWindow]** **{**  **def** process**(**key**:** **String,** context**:** **Context,** input**:** **Iterable[(String**, **Long)],** out**:** **Collector[String]):** **()** **=** **{**  **var** count **=** 0L  **for** **(**in **<-** input**)** **{**  count **=** count **+** 1  **}**  out**.**collect**(**s"Window ${context.window} count: $count"**)**  **}**  **}** |

这个例子展现了ProcessWindowFunction 对窗口中的元素进行计数。此外，窗口函数添加和窗口输出有关的信息。

**注意**：使用ProcessWindowFunction  做和聚合类似的计数是十分低效的。下一小节将展示ReduceFunction ， AggregateFunction 和ProcessWindowFunction 如何合并以获得增量聚合以及为ProcessWindowFunction添加信息。

* + - * 1. ProcessWindowFunction with Incremental Aggregation

ProcessWindowFunction  可以和ReduceFunction，AggregateFunction 合并也可以和FoldFunction 合并，他们合并后可以增量的聚合到达窗口内的元素。当窗口关闭时，ProcessWindowFunction 将提供聚合的结果。这将允许当访问ProcessWindowFunction附加窗口信息的时候去增量计算。

**注意**：你也可以使用遗留的WindowFunction 替代ProcessWindowFunction 做增量窗口聚合。

Incremental Window Aggreation with ReduceFunction

下面的示例演示如何将递增的ReduceFunction与ProcessWindowFunction结合使用，以返回窗口中最小的事件以及窗口的开始时间。

|  |
| --- |
| Scala 代码 |
| **val** input**:** **DataStream[SensorReading]** **=** **...**  input  **.**keyBy**(<**key selector**>)**  **.**timeWindow**(<**duration**>)**  **.**reduce**(**  **(**r1**:** **SensorReading,** r2**:** **SensorReading)** **=>** **{** **if** **(**r1**.**value **>** r2**.**value**)** r2 **else** r1 **},**  **(** key**:** **String,**  window**:** **TimeWindow,**  minReadings**:** **Iterable[SensorReading],**  out**:** **Collector[(Long**, **SensorReading)]** **)** **=>**  **{**  **val** min **=** minReadings**.**iterator**.**next**()**  out**.**collect**((**window**.**getStart**,** min**))**  **}**  **)** |

Incremental Window Aggration with AggrationFunction

下面的例子展示了一个递增的AggregateFunction和ProcessWindowFunction  合并计算并均值，并将key和发送出去。

|  |
| --- |
| Scala 代码 |
| **val** input**:** **DataStream[(String**, **Long)]** **=** **...**  input  **.**keyBy**(<**key selector**>)**  **.**timeWindow**(<**duration**>)**  **.**aggregate**(new** **AverageAggregate(),** **new** **MyProcessWindowFunction())**  *// Function definitions*  */\*\**  *\* The accumulator is used to keep a running sum and a count. The [getResult] method*  *\* computes the average.*  *\*/*  **class** **AverageAggregate** **extends** **AggregateFunction[(String**, **Long)**, **(Long**, **Long)**, **Double]** **{**  **override** **def** createAccumulator**()** **=** **(**0L**,** 0L**)**  **override** **def** add**(**value**:** **(String,** **Long),** accumulator**:** **(Long,** **Long))** **=**  **(**accumulator**.**\_1 **+** value**.**\_2**,** accumulator**.**\_2 **+** 1L**)**  **override** **def** getResult**(**accumulator**:** **(Long,** **Long))** **=** accumulator**.**\_1 **/** accumulator**.**\_2  **override** **def** merge**(**a**:** **(Long,** **Long),** b**:** **(Long,** **Long))** **=**  **(**a**.**\_1 **+** b**.**\_1**,** a**.**\_2 **+** b**.**\_2**)**  **}**  **class** **MyProcessWindowFunction** **extends** **ProcessWindowFunction[Double**, **(String**, **Double)**, **String**, **TimeWindow]** **{**  **def** process**(**key**:** **String,** context**:** **Context,** averages**:** **Iterable[Double],** out**:** **Collector[(String**, **Double]):** **()** **=** **{**  **val** average **=** averages**.**iterator**.**next**()**  out**.**collect**((**key**,** average**))**  **}**  **}** |

Incremental Window Aggration with FoldFunction

下面的例子展示了如何利用递增的FoldFunction 和ProcessWindowFunction 合并抽取窗口中的时间数量，并返回key和窗口的结束时间

|  |
| --- |
| Scala 代码 |
| **val** input**:** **DataStream[SensorReading]** **=** **...**  input  **.**keyBy**(<**key selector**>)**  **.**timeWindow**(<**duration**>)**  **.**fold **(**  **(**""**,** 0L**,** 0**),**  **(**acc**:** **(String,** **Long,** **Int),** r**:** **SensorReading)** **=>** **{** **(**""**,** 0L**,** acc**.**\_3 **+** 1**)** **},**  **(** key**:** **String,**  window**:** **TimeWindow,**  counts**:** **Iterable[(String**, **Long**, **Int)],**  out**:** **Collector[(String**, **Long**, **Int)]** **)** **=>**  **{**  **val** count **=** counts**.**iterator**.**next**()**  out**.**collect**((**key**,** window**.**getEnd**,** count**.**\_3**))**  **}**  **)** |

* + - * 1. Using per-window state in ProcessWindowFunction

除了访问ProcessWindowFunction 的keyed state(任何的rich 函数都可以)之外，也可以使用Keyed state,但是他的作用域是当前处理的函数窗口。在这种情况下，理解每个窗口状态引用了什么是非常重要的。对于“window”这里涉及：

* 当指定窗口operation时定义窗口：可能是一个小时的tumbling 窗口或一小时滑动一次窗口时长为2小时的滑动窗口。
* 为给定的key定义窗口的实际例子是：这可能是用户id xyz的时间窗口从12:00到13:00。这是基于窗口定义的，有很多窗口是基于作业当前正在处理的key的数量，以及事件属于哪个时间段的.

每个窗口状态都与这两个状态中的后一个相关联。也就是说，如果我们为1000个不同的key处理事件，并且所有key的事件都在[12:00,13:00]时间窗口中，那么就会有1000个窗口实例，每个窗口实例都有自己的keyed窗口状态。

Process()调用接收到上下文的对象有两个方法，它允许访问两种状态类型。

* globalState(), 它允许访问不在窗口作用域keyed state
* windowState(), 它仅允许访问在窗口作用域内的keyed state

如果您预期同一窗口会发生多次触发，那么这个特性将非常有用，因为当您对延迟到达的数据有延迟触发时，或者当您有一个自定义触发器进行推测性的早期触发时，可能会发生这种情况。在这种情况下，您将在每个窗口状态中存储关于先前触发信息或之前触发触发的次数。

当使用窗口状态的时候当一个窗口被清理的时候，窗口状态的清理也是十分重要的。状态清理应定义在clear()方法中。

* + - * 1. WindowFunction(Legacy)

在某些地方能使用ProcessWindowFunction 时也可以使用WindowFunction .WindowFunction是ProcessWindowFunction 的老版本，它提供了更少的上下文信息并且没有一些高级的特性，比如每个窗口的keyed state。这个接口在已经过时。

签名像下面这样：

|  |
| --- |
| Scala 代码 |
| **trait** **WindowFunction[IN**, **OUT**, **KEY**, **W** **<:** **Window]** **extends** **Function** **with** **Serializable** **{**  */\*\**  *\* Evaluates the window and outputs none or several elements.*  *\**  *\* @param key The key for which this window is evaluated.*  *\* @param window The window that is being evaluated.*  *\* @param input The elements in the window being evaluated.*  *\* @param out A collector for emitting elements.*  *\* @throws Exception The function may throw exceptions to fail the program and trigger recovery.*  *\*/*  **def** apply**(**key**:** **KEY,** window**:** **W,** input**:** **Iterable[IN],** out**:** **Collector[OUT])**  **}** |

它能像下面这样去使用：

|  |
| --- |
| Scala |
| **val** input**:** **DataStream[(String**, **Long)]** **=** **...**  input  **.**keyBy**(<**key selector**>)**  **.**window**(<**window assigner**>)**  **.**apply**(new** **MyWindowFunction())** |

* + - 1. Triggers(触发器)

触发器决定了窗口何时被window function被触发。每个WindowAssigner 都有一个默认的触发器。如果默认额触发器不适合你的需求，你可以使用trigger(…) 进行自定义触发器

触发器接口有下面5个方法，它允许对不同的事件作出反应。

* 任何一个元素被添加到窗口的时候调用onElement() 方法
* 当注册event-time 定时触发的时候调用onEventTime() 方法
* 当注册processing-time 定时触发的时候调用onProcessingTime () 方法
* onMerge() 方法是和有状态的触发器相关的，当两个触发器窗口合并一致的时候合并他们的状态，例如，当使用session window时。
* 任何操作被执行需要移除的时候需要调用clear() 方法

关于上面的方法有两点需要注意：

1. 前三个决定如何通过返回的TriggerResult对其调用event进行操作,他们的动作可以是下面中的一个：

* CONTINUE：啥都不干
* FIRE：触发计算
* PURGE：清除窗口中的元素
* FIRE\_AND\_PURGE：触发计算，然后清除窗口中的元素

1. 这当中的任意一个方法都能被用于注册processing-time 和 event-time 定时器。
   * + - 1. Fire and Purge

一旦窗口确定了窗口准备好了进行处理，触发等。它将返回*FIRE* 或 *FIRE\_AND\_PURGE*. 这是窗口operator 发送当前窗口的一个信号。给定一个带有ProcessWindowFunction 窗口，所有的元素都发送到ProcessWindowFunction （可能把它们放在驱逐器之后。）。一个带有educeFunction, AggregateFunction, or FoldFunction 的窗口简单的发送他们聚合的结果

当一个触发器被触发，它要么是FIRE 要么是FIRE\_AND\_PURGE。当是FIRE时他会保留窗口的内容，而FIRE\_AND\_PURGE 会移除窗口内容。默认情况下，预先实现触发器是简单的FIRE 并没有清除窗口的状态。

注意：Purging 只是简单的移除窗口的内容，但是对于窗口的任何元数据信息和任何处罚期状态都是原封不动。

* + - * 1. Default Trigger of WindowAssigners

WindowAssigner 的默认触发器适合大多数场景。例如，所有基于event-time 窗口分配器都是由EventTimeTrigger 默认触发的。这个触发器在水位到达窗口的末端适合进行触发一次。

注意：GlobalWindow 的默认触发器是NeverTrigger ，它永远都不会被触发。因此，当你使用GlobalWindow 的时候必须要自定义一个触发器。

注意：使用trigger() 指定触发器去覆盖WindowAssigner默认的触发器。例如，如果你想为TumblingEventTimeWindows 指定CountTrigger ，窗口将不再随着时间的进度而触发而是通过count（数量）来进行触发。假如你想要基于时间和count都能进行触发你必须要自定义触发器。

* + - * 1. Built-in and Custom Triggers

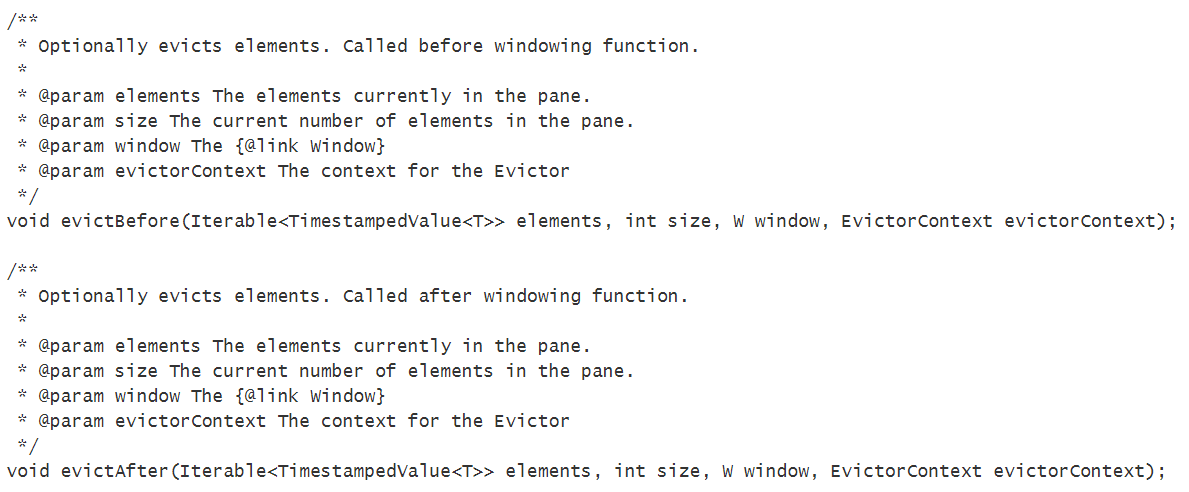
Flink 有几个内置的触发器：

* EventTimeTrigger 上面已经提到过，他说基于event-time 的时间，并检测器watermark来触发的。
* ProcessingTimeTrigger 是基于processing-time
* CountTrigger ：当窗口中的元素超过限定值的时候被触发一次。
* PurgingTrigger ：获取另外一个触发器作为参数并将其转换为它的purging触发器。

如果你需要实现自定义触发器，你需要扩展[Trigger](https://github.com/apache/flink/blob/master/flink-streaming-java/src/main/java/org/apache/flink/streaming/api/windowing/triggers/Trigger.java)抽象类。请注意，API在未来的版本中可能会发生变化。

* + - 1. Evictors(驱逐器)

Flink的窗口模型允许在WindowAssigner 和Trigger 之外添加一个可选的Evictor 。通过使用Evictor(…) 方法来完成这个动作。Evictor 能在触发器触发之后以及在窗口函数被应用之前或之后移除元素。为了做这些，Evictor 接口有两个方法：



evictBefore() 方法包含了在使用窗口函数之前的应用逻辑，而evictAfter（）是包含了在窗口函数之后的逻辑。在应用窗口之前驱逐的元素不会被驱逐器处理。

Flink 内置了下面三种驱逐器，他们是：

* CountEvictor：保留窗口中用户指定的元素数量并清除窗口缓存剩余的元素。L
* DeltaEvictor：取一个DeltaFunction 和一个threshold，计算窗口缓冲区中最后一个元素和每个剩余元素之间的差值，并删除大于或等于阈值的元素。
* TimeEvictor：将一个以毫秒为单位的interval 作为参数，对于给定的窗口，它在其元素中查找最大时间戳max\_ts，并删除时间戳小于max\_ts - interval的所有元素。

默认：默认情况下，所有预先实现的驱逐器都在窗口函数之前应用它们的逻辑

注意：指定驱逐器可以防止任何预聚合，因为在应用计算之前，必须将窗口的所有元素传递给驱逐器。

注意：Flink 不能保证窗口中元素的顺序，这意味着驱逐器虽然能够移除窗口中的开始元素，但是这个元素并不一定是最先或最后到达的。

* + - 1. Alllowed Lateness(允许延迟)

当使用event-time 处理时，元素可能延迟到达。例如，Flink使用wartermark跟踪元素属于哪个event-time 最后时间戳的窗口。更多信息查看[event time](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_time.html) 和 [late elements](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_time.html#late-elements)

默认情况下，当watermark通过窗口的结束点时迟到的元素将被删除。然而，Flink允许为窗口operator指定最大的allowed lateness。允许延迟元素在被删除之前能延迟多长时间，默认是0。watermark经过窗口末端但经过窗口末端加上允许的延迟之前到达的元素，仍然被添加到窗口中。根据所使用的触发器，延迟但未删除的元素可能导致窗口再次触发。这是EventTimeTrigger的情况。

为了让他保持工作，Flink会保持窗口的状态，直到允许延迟时间过期才会被清理。一旦发生这种情况，Flink将移除窗口并删除状态。可以查看 [Window Lifecycle](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#window-lifecycle)小结中的描述。

默认：默认情况下，allowed lateness 设置为0 。既元素在watermark之后到达将会被删除。

你可以想下面这样指定一个allowed lateness：

|  |
| --- |
| Scala |
| **val** input**:** **DataStream[T]** **=** **...**  input  **.**keyBy**(<**key selector**>)**  **.**window**(<**window assigner**>)**  **.**allowedLateness**(<**time**>)**  **.<**windowed transformation**>(<**window function**>)** |

注意：当使用GlobalWindows 时候没有元素需要去考虑延迟，因为global window的最后时间戳是Long.MAX\_VALUE.

* + - * 1. Getting late data as a side output（获取延迟数据作为side输出）

使用Flink的[side output](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/side_output.html) 特性，你可以获得一个数据流，当这个流延迟就被被丢弃。

首先 你可以通过在window stream上使用sideOutputLateData(OutputTag) 获得延迟数据。然后，你就能在window operation结果上得到side-output 流。

|  |
| --- |
| Scala |
| **val** lateOutputTag **=** **OutputTag[T](**"late-data"**)**  **val** input**:** **DataStream[T]** **=** **...**  **val** result **=** input  **.**keyBy**(<**key selector**>)**  **.**window**(<**window assigner**>)**  **.**allowedLateness**(<**time**>)**  **.**sideOutputLateData**(**lateOutputTag**)**  **.<**windowed transformation**>(<**window function**>)**  **val** lateStream **=** result**.**getSideOutput**(**lateOutputTag**)** |

* + - * 1. Late element considerations(延迟元素的考虑)

当指定了 allowed lateness大于0时，wather经过窗口末端后，窗口的内容保持不变。在这些场景中，当一个迟到但是未被删除的元素到达时，他有可能导致另一个窗口被触发。这些触发被称作 late firings，由于他们是通过延迟时间被触发，对比main firings ,main firings是窗口的第一次触发。在session windows的场景中，late firing 可能导致窗口的合并，因为他们可能“桥接”两个窗口之间的间隙。

注意：你应该要意识到，通过late firing发射的元素应作为上一次计算被更新的结果，例如，你的数据流将包含相同计算的多种结果。根据你的应用，你需要考虑数据的副本或者删除他们。

* + - 1. Working with window results

Windowed operation的结果也是一个DataStream.没有window operation 的信息被保留在结果元素中，所以如果你想保持窗口的元数据，你必须要在ProcessWindowFunction 中编码结果元素信息。在结果元素上设置的相关信息是元素timestamp。它设置了处理窗口最大允许timestamp,这个最大允许timestamp是末端timestamp-1(因为窗口的结束时间戳是被排除的).

注意，event-time窗口和processing-time窗口都是如此。例如，在窗口操作之后，元素总是有一个timestamp，但这可以是event-time 时间戳或processing-time时间戳。对于processing-time窗口，这没有什么特别的含义，但是对于event-time窗口，这与wathermark如何与窗口交互一起支持具有相同窗口大小的 [consecutive windowed operations](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#consecutive-windowed-operations) 。在了解wathermark如何与窗口交互之后，我们将介绍这一点。

* + - * 1. Interaction of watermarks and windows(watermark和window交互)

在继续这一小节之前，你可能需要看下[event time and watermarks](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_time.html).

当watermark到达了window operator，触发器有如下来2件事情要做：

* Watermark将触发所有窗口的计算，而这个窗口最大时间戳比新watermark时间戳更小
* Watermark 被转发到下游operations

直觉上，一旦他们收到watermark, watermark会“冲洗”任何在下游操作中被认为较晚的窗口。

* + - * 1. Consecutive windowed operations（连续的窗口操作）

如前所述，计算窗口显示结果的时间戳的方式以及watermark与窗口交互的方式允许串接连续的窗口操作。当您想要执行两个连续的窗口操作时，如果您想使用不同的键，但仍然希望来自相同上游窗口的元素最终位于相同的下游窗口中，那么这可能非常有用。考虑一下这个例子:

|  |
| --- |
| Scala |
| **val** input**:** **DataStream[Int]** **=** **...**  **val** resultsPerKey **=** input  **.**keyBy**(<**key selector**>)**  **.**window**(TumblingEventTimeWindows.**of**(Time.**seconds**(**5**)))**  **.**reduce**(new** **Summer())**  **val** globalResults **=** resultsPerKey  **.**windowAll**(TumblingEventTimeWindows.**of**(Time.**seconds**(**5**)))**  **.**process**(new** **TopKWindowFunction())** |

在本例中，第一个操作的time window[0,5]的结果也会在随后的窗口操作中以time window[0,5]结束。这允许计算每个键的和，然后在第二个操作中计算同一个窗口中的top-k元素。

* + - 1. Useful state size considerations(有用状态大小考虑)

窗口可以被定义在一个很长的时间周期上（如 一天，一周或一个月）从而导致积累一个巨大的状态。当估算窗口计算存储需求时，下面有几个规则需要记住：

1. Flink为每个窗口的每个元素创建一个副本。鉴于此，tumbling窗口的每个元素保持一个副本。与此相反，sliding 窗口会为每个元素创建若干个副本。因此sliding 窗口的周期是一天而滑动是一秒将不会是一个好的主意。
2. ReduceFunction, AggregateFunction, 和FoldFunction都能很显著的降低存储需求，因为他们都会提前聚合元素但是每个窗口仅需要存储一个value . 相反，ProcessWindowFunction 会积攒所有的元素。
3. 使用Evictor 会阻止任何的预聚合。因此窗口的所有元素在应用计算之前都会通过Evictor。

* + 1. Joining

|  |
| --- |
| **Window Join** |
| Tumbling Window Join |
| Sliding Window Join |
| Session Window Join |
| **Interval Join** |

* + - 1. Window Join

Window join 连接两个Strem中的元素，它们共享位于同一个窗口中的key. 窗口可以使用 [window assigner](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/operators/windows.html#window-assigners) 被定义并计算在两个stream上的元素。

然后，将两边的元素传递给用户定义的JoinFunction或FlatJoinFunction，用户可以在这里发布符合连接标准的结果。

通常使用方法如下：

|  |
| --- |
| stream**.**join**(**otherStream**)**  **.**where**(<**KeySelector**>)**  **.**equalTo**(<**KeySelector**>)**  **.**window**(<**WindowAssigner**>)**  **.**apply**(<**JoinFunction**>)** |

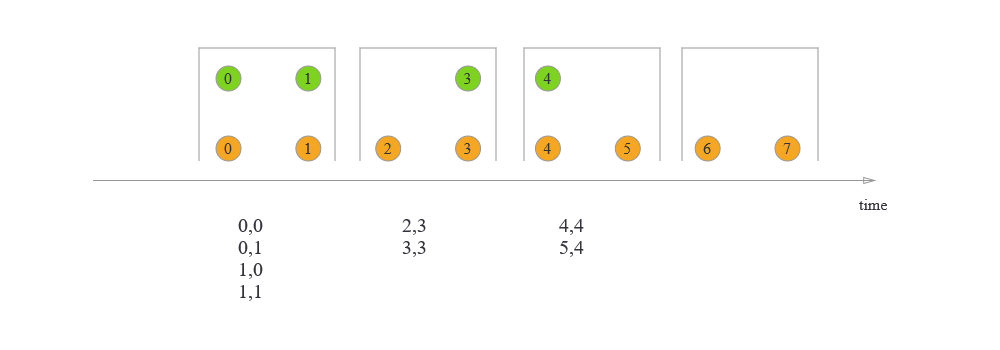
语法上需要注意的点：

* 两个流的元素成对组合的创建行为类似于内部连接，这意味着如果一个流中的元素没有来自另一个流的相应元素可供连接，则不会发出该元素。
* 被join的元素将作为他们的时间戳，但是最大的时间戳依旧在他们各自的窗口中，例如，带有[5,10)作为边界的窗口将导致连接的元素的时间戳为9。

在下一节中，我们将概述不同类型的窗口连接如何使用一些示例场景来表现。

* + - * 1. Tumbling Window Join

当执行tumbling window  join 时，具有 common key和common tumbling window的所有元素作为成对组合被join. 因为这个行为类似于inner join。一个流中没有来自另一个流的元素不会来自于在其tumbling window中另一个流的元素。

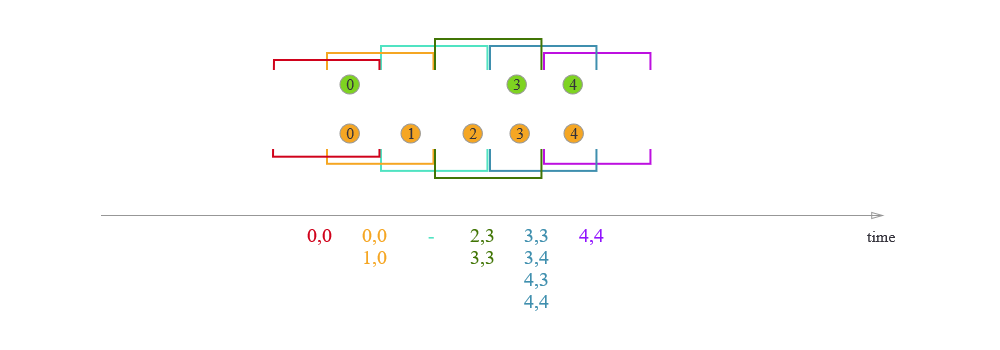


正如上图阐述的一样，我们定义了一个2毫秒的窗口，窗口形式是[0,1],[2,3]…..。这张图展示了每个窗口中所有的元素都将通过JoinFunction成对计算。注意 在窗口[6,7]中没有任何元素被发送，是因为没有绿色stream的元素和橘色元素⑥ 和 ⑦ 进行join

|  |
| --- |
| Scala |
| **import** org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows**;**  **import** org.apache.flink.streaming.api.windowing.time.Time**;**  **...**  **val** orangeStream**:** **DataStream[Integer]** **=** **...**  **val** greenStream**:** **DataStream[Integer]** **=** **...**  orangeStream**.**join**(**greenStream**)**  **.**where**(**elem **=>** */\* select key \*/***)**  **.**equalTo**(**elem **=>** */\* select key \*/***)**  **.**window**(TumblingEventTimeWindows.**of**(Time.**milliseconds**(**2**)))**  **.**apply **{** **(**e1**,** e2**)** **=>** e1 **+** "," **+** e2 **}** |

* + - * 1. Sliding Window Join

当执行一个sliding window join时，所有具有公共key和公共sliding window的元素都是成对组合并传递给JoinFunction或FlatJoinFunction. 在当前滑动窗口中，一个流中没有来自另一个流的元素的元素不会被释放! 请注意，有些元素可能会在一个滑动窗口中连接，但不会在另一个窗口中连接!

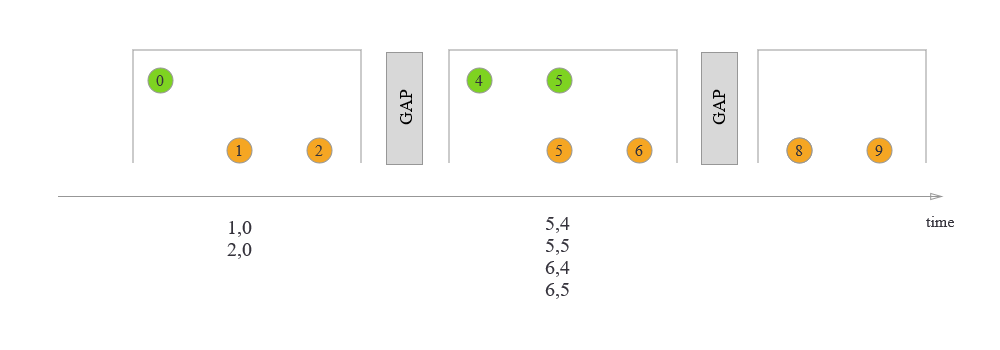


在这个案例中，我们使用滑动的窗口大小为2毫秒，滑动时间为1毫秒，滑动窗口为[-1,0],[0,1],[1,2]…. X轴下是被JOIN的元素，这些元素被传递给每个窗口的JoinFunction。

|  |
| --- |
| Scala |
| **import** org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows**;**  **import** org.apache.flink.streaming.api.windowing.time.Time**;**  **...**  **val** orangeStream**:** **DataStream[Integer]** **=** **...**  **val** greenStream**:** **DataStream[Integer]** **=** **...**  orangeStream**.**join**(**greenStream**)**  **.**where**(**elem **=>** */\* select key \*/***)**  **.**equalTo**(**elem **=>** */\* select key \*/***)**  **.**window**(SlidingEventTimeWindows.**of**(Time.**milliseconds**(**2**)** */\* size \*/***,** **Time.**milliseconds**(**1**)** */\* slide \*/***))**  **.**apply **{** **(**e1**,** e2**)** **=>** e1 **+** "," **+** e2 **}** |

* + - * 1. Session Window Join

当执行session window join 时，具有相同key的所有元素，当他们的“合并”满足session评判标准时成对组合join 并传递他们给JoinFunction或FlatJoinFunction。和inner join相同，所以如果session window仅包含来至于一个stream的元素，不会有任何元素被发送。



这里我们定义了一个1ms间隙切分的session window join . 这里有3个sessions。 前面两个session 从两个stream中join 元素并传递给JoinFunction。在第三个session中没有绿色stream的元素。所以 ⑧ 和⑨ 不能被joined.

|  |
| --- |
| Scala |
| **import** org.apache.flink.streaming.api.windowing.assigners.EventTimeSessionWindows**;**  **import** org.apache.flink.streaming.api.windowing.time.Time**;**  **...**  **val** orangeStream**:** **DataStream[Integer]** **=** **...**  **val** greenStream**:** **DataStream[Integer]** **=** **...**  orangeStream**.**join**(**greenStream**)**  **.**where**(**elem **=>** */\* select key \*/***)**  **.**equalTo**(**elem **=>** */\* select key \*/***)**  **.**window**(EventTimeSessionWindows.**withGap**(Time.**milliseconds**(**1**)))**  **.**apply **{** **(**e1**,** e2**)** **=>** e1 **+** "," **+** e2 **}** |

* + - 1. Interval Join

Interval join用一个公共的key连接两个stream（现在我们称它们为A和B）的元素,其中stream B有一个timestamps，这个timestamps位于stream A 元素中的timestamp的相对时间interval 。

这里也有更多的时间表达式 如：

b.timestamp ∈ [a.timestamp + lowerBound; a.timestamp + upperBound]

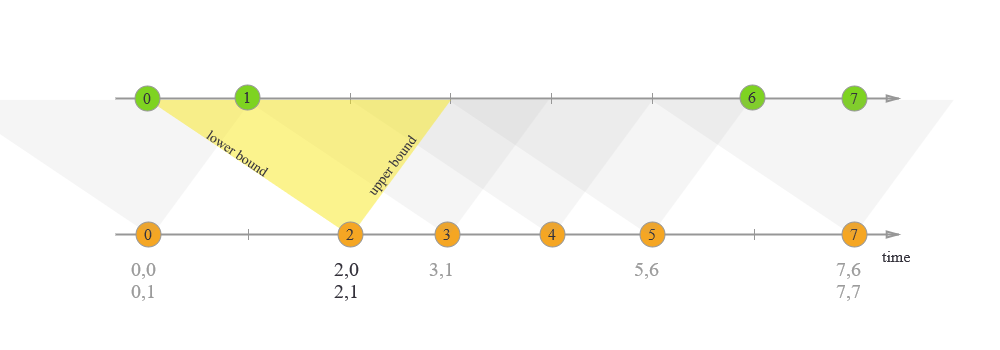
或

a.timestamp + lowerBound <= b.timestamp <= a.timestamp + upperBound

a 和 b元素存在于A和B中，它们共享公用的key.只要lower bound（下边界）总是小于等于upper bound（上边界）那么上边界和下边界既能是负数也能是正数。

当一对元素被传递给ProcessJoinFunction时，它们将使用两个元素中更大的那个时间戳（通过ProcessJoinFunction.Context 可访问）。

注意： interval join 目前仅支持 event time.



上面的例子中，我们join两个流“橘色”和“绿色”，其下边界是-2 毫秒 上边界是+1 毫秒。默认情况下，边界是被排除的，而  .lowerBoundExclusive()和.upperBoundExclusive 可以修改这种行为。

这些符号将转换为：

orangeElem.ts + lowerBound <= greenElem.ts <= orangeElem.ts + upperBound

|  |
| --- |
| Scala |
| **import** org.apache.flink.streaming.api.functions.co.ProcessJoinFunction**;**  **import** org.apache.flink.streaming.api.windowing.time.Time**;**  **...**  **val** orangeStream**:** **DataStream[Integer]** **=** **...**  **val** greenStream**:** **DataStream[Integer]** **=** **...**  orangeStream  **.**keyBy**(**elem **=>** */\* select key \*/***)**  **.**intervalJoin**(**greenStream**.**keyBy**(**elem **=>** */\* select key \*/***))**  **.**between**(Time.**milliseconds**(-**2**),** **Time.**milliseconds**(**1**))**  **.**process**(new** **ProcessJoinFunction[Integer**, **Integer**, **String]** **{**  **override** **def** processElement**(**left**:** **Integer,** right**:** **Integer,** ctx**:** **ProcessJoinFunction[Integer**, **Integer**, **String]#Context,** out**:** **Collector[String]):** **Unit** **=** **{**  out**.**collect**(**left **+** "," **+** right**);**  **}**  **});**  **});** |

* + 1. Process Function

|  |
| --- |
| **The ProcessFunction** |
| **Low-Leval Joins** |
| **Example** |
| **The KeyedProcessFunction** |
| **Timers(定时器)** |
| Fault Tolerance |
| Timer Coalescing(定时调度) |

* + - 1. The ProcessFunction

ProcessFunction是一个底层stream 处理操作，它能访问流应用基本构建块：

* events(流元素)
* state(容错，一致性，仅仅在keyed stream上)
* timers(process time 和 event time. 仅仅在keyed stream上)

ProcessFunction 可以当做是个可访问keyed state 和 timers 的FaltMapFunction。通过调用在输入流中接收每个event去处理event.

为了容错状态，ProcessFunction 可通过RuntimeContext访问Flink的[keyed state](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/stream/state/state.html)。类似于其他有状态的函数能访问keyed state。

Timers(定时器)允许对event-time和processing-time的变化做出响应。每次调用函数processElement(...)会得到一个Context 对象，它提供了访问元素的event time 时间戳和TimerService。TimerService可为将来的 event-/processing-time 注册回调。当定时器特定时间到达时，onTimer(...)方法被调用。在调用期间，所有的状态再次作用到被定时器创建的key，允许定时器去操作keyed state.

注意：如果你想访问keyed state和timers，你必须在keyed stream上应用ProcessFunction 。

stream**.**keyBy**(...).**process**(new** MyProcessFunction**())**

* + - 1. Low-Leval Joins

为了在两个输入上实现底层操作，应用可使用CoProcessFunction。 这个函数绑定到两个不同的输入并为两个不同的输入记录分别调用 processElement1(...) 和processElement2(...) 。

实现一个底层的JOIN 经典模型如下;

* 为一个输入（或者两个）创建state 对象。
* 从输入接收元素后更新state.
* 收到来自于其他输入的元素，检测状态并产生join结果。

例如：你能把客户数据和金融交易数据Join起来，同时保留客户数据的状态。如果你关心的是在无序的event中完成的和确定性的join,当客户数据流的watermark通过交易时间时，你可以使用定时器对交易数据进行评估和join.

* + - 1. Example

下面的例子维持每个key的数量，每当过去一分钟并且对这个key没有更新的时候就发送一个key/count 对：

* count，key 和last-modification-timestamp  都存储在ValueState。通过key确定scope
* ProcessFunction 对每条记录增量的技术和设置last-modification-timestamp
* 函数会在一分钟后进行回调
* 在每次回调时，它检查回调的event-time 时间戳和存储计数的last-modification时间，如果匹配择发送key/count

注意：这个简单的案例已经使用session window 被实现了。这里我们使用ProcessFunction 去阐述它提供的基本模型。

|  |
| --- |
| Scala |
| **import** org.apache.flink.api.common.state.ValueState  **import** org.apache.flink.api.common.state.ValueStateDescriptor  **import** org.apache.flink.streaming.api.functions.ProcessFunction  **import** org.apache.flink.streaming.api.functions.ProcessFunction.Context  **import** org.apache.flink.streaming.api.functions.ProcessFunction.OnTimerContext  **import** org.apache.flink.util.Collector  *// the source data stream*  **val** stream**:** **DataStream[Tuple2[String**, **String]]** **=** **...**  *// apply the process function onto a keyed stream*  **val** result**:** **DataStream[Tuple2[String**, **Long]]** **=** stream  **.**keyBy**(**0**)**  **.**process**(new** **CountWithTimeoutFunction())**  */\*\**  *\* The data type stored in the state*  *\*/*  **case** **class** **CountWithTimestamp(**key**:** **String,** count**:** **Long,** lastModified**:** **Long)**  */\*\**  *\* The implementation of the ProcessFunction that maintains the count and timeouts*  *\*/*  **class** **CountWithTimeoutFunction** **extends** **ProcessFunction[(String**, **String)**, **(String**, **Long)]** **{**  */\*\* The state that is maintained by this process function \*/*  **lazy** **val** state**:** **ValueState[CountWithTimestamp]** **=** getRuntimeContext  **.**getState**(new** **ValueStateDescriptor[CountWithTimestamp](**"myState"**,** classOf**[CountWithTimestamp]))**  **override** **def** processElement**(**value**:** **(String,** **String),** ctx**:** **Context,** out**:** **Collector[(String**, **Long)]):** **Unit** **=** **{**  *// initialize or retrieve/update the state*  **val** current**:** **CountWithTimestamp** **=** state**.**value **match** **{**  **case** **null** **=>**  **CountWithTimestamp(**value**.**\_1**,** 1**,** ctx**.**timestamp**)**  **case** **CountWithTimestamp(**key**,** count**,** lastModified**)** **=>**  **CountWithTimestamp(**key**,** count **+** 1**,** ctx**.**timestamp**)**  **}**  *// write the state back*  state**.**update**(**current**)**  *// schedule the next timer 60 seconds from the current event time*  ctx**.**timerService**.**registerEventTimeTimer**(**current**.**lastModified **+** 60000**)**  **}**  **override** **def** onTimer**(**timestamp**:** **Long,** ctx**:** **OnTimerContext,** out**:** **Collector[(String**, **Long)]):** **Unit** **=** **{**  state**.**value **match** **{**  **case** **CountWithTimestamp(**key**,** count**,** lastModified**)** **if** **(**timestamp **==** lastModified **+** 60000**)** **=>**  out**.**collect**((**key**,** count**))**  **case** **\_** **=>**  **}**  **}**  **}** |

* + - 1. The KeyedProcessFunction

KeyedProcessFunction 是 ProcessFunction的扩展。它提供了在onTimer(...) 方法上对定时器key的访问。

|  |
| --- |
| Scala |
| **override** **def** onTimer**(**timestamp**:** **Long,** ctx**:** **OnTimerContext,** out**:** **Collector[OUT]):** **Unit** **=** **{**  **var** key **=** ctx**.**getCurrentKey  *// ...*  **}** |

* + - 1. Timers

定时器类型在内部都是通过TimerService 进行操作并排队执行。

TimerService 根据key和时间戳删除重复的timers. 也就是说，每个key和时间戳最多有一个timer. 如果多个timer用相同的时间戳进行注册， onTimer() 方法只会被调用一次。

注意：Flink 同步调用onTimer（） 和 processElement() 方法。所以，用户不需要担心并发修改状态的问题。

* + - * 1. Fault Tolerance(容错)

Timer和应用状态一样都具备容错和检查点功能。在失败恢复的场景中或者从savepoint中启动应用，定时器都能被恢复。

注意：在恢复之前应该触发的checkpointed处理时间计时器将立即触发。这可能发生在应用程序从故障恢复或从savepoint启动时。

注意： 除了RocksDB backend / with incremental snapshots / with heap-based 定时器之外，其他Timers总是异步进行checkpointed的。注意：由于定时器是checkpoint state的一部分，因此大量的timers 增加检查点时间。

* + - * 1. Timer Coalescing（定时调度）

由于Flink的每个key和时间戳上只有一个定时器，可以通过减少定时器并合并他们来减少定timers数量。

如果定时器是1秒，你应该向下取整。Timers最多提前1秒触发并不会晚于请求的毫秒数。因此，每个key每秒最多有一个timer.

|  |
| --- |
| Scala |
| **val** coalescedTime **=** **((**ctx**.**timestamp **+** timeout**)** **/** 1000**)** **\*** 1000  ctx**.**timerService**.**registerProcessingTimeTimer**(**coalescedTime**)** |

由于event-time  定时器只有在有watermark进来是被触发，你也可以用下一个watermark进行调度

|  |
| --- |
| Scala |
| **val** coalescedTime **=** ctx**.**timerService**.**currentWatermark **+** 1  ctx**.**timerService**.**registerEventTimeTimer**(**coalescedTime**)** |

Timers 也能停止和移除

停止processing-time  定时器

|  |
| --- |
| Scala |
| **val** timestampOfTimerToStop **=** **...**  ctx**.**timerService**.**deleteProcessingTimeTimer**(**timestampOfTimerToStop**)** |

停止 event-time 定时器：

|  |
| --- |
| Scala |
| **val** timestampOfTimerToStop **=** **...**  ctx**.**timerService**.**deleteEventTimeTimer**(**timestampOfTimerToStop**)** |

注意:如果没有定时器被注册，停止定时器没有任何影响。

* + 1. Async I/O

|  |
| --- |
| **The need for Asynchronous I/O Operations** |
| **Prerequisites** |
| **Async I/O API** |
| Timeout Handling |
| Order of Results |
| Event Time |
| Fault Tolerance Guarantees |
| Implementation Tips |
| Caveat |

本小节解释如何外部存储系统如何使用Flink的asynchronous I/O API. 对于不熟悉异步或事件驱动程序的用户，对事件驱动编程非常有用。

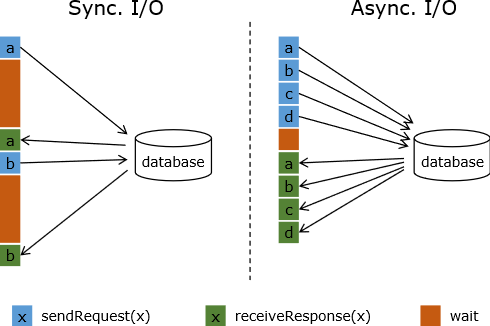
注意：有关asynchronous I/O 详细设计和实现在[FLIP-12: Asynchronous I/O Design and Implementation](https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=65870673)中可以找到

* + - 1. The need for Asynchronous I/O Operations

当和外部系统（stream数据需要存储到数据库中）进行交互时，有一个需要关心的问题是和外部系统的通信延迟不能影响流应用的整体工作。

直接访问外部数据库中的数据，例如在mapFunction函数中，经典的做法是以**synchronous** （同步）的方式进行交互：请求发送到数据库然后MapFunction等待。在多数场景中，这种等待会造成绝大多数的时间浪费。

和database进行异步交互，这就意味着单个并行函数实例能并行处理多个请求，也能并行的接收响应。通过这种方式，通过发送其他的请求和响应这个等待时间就可以被减少很多。至少，这个等待时间会被分摊到多个请求上。这是在大多数场景中获得高吞吐量的方式。



注意：在一些场景中通过调整MapFunction到一个更高的并行度通常可以提高吞吐量，但是它通常带来更高的资源消费：更多的MapFunction 实例意味着更多的tasks,threads，Flink-internal network connections（Flink 内部的网络通信），network connections to the database（数据库的网络连接），buffers（缓存）等。

* + - 1. Prerequisites（预置条件）

如上所述，实现asynchronous I/O到数据库通信，需要客户端数据库支持异步请求。不过多数流行的数据库都提供了这样的客户端。

在没有异步请求的客户端中，一种可尝试的方式是通过创建多个客户端和用线程池处理同步调用去调整同步客户端到到一个受限并发客户端。然而，这种方式通常没有异步客户端有效。

* + - 1. Async I/O API

Flink  Async I/O API  允许用户使用异步客户端。API处理data streams，包含处理顺序，event time,容错，等。

假设目标数据库有异步客户端，为实现一个具有asynchronous I/O 流转换有三部分需要实现：

* 实现一个AsyncFunction ，它用来分发请求
* 一个callback,这个callback用于获取operation的结果并将其交个ResultFuture。
* 在数据流上转换时应用async I/O operation

下面的代码阐述了这个基本模型：

|  |
| --- |
| Scala |
| */\*\**  *\* An implementation of the 'AsyncFunction' that sends requests and sets the callback.*  *\*/*  **class** **AsyncDatabaseRequest** **extends** **AsyncFunction[String**, **(String**, **String)]** **{**  */\*\* The database specific client that can issue concurrent requests with callbacks \*/*  **lazy** **val** client**:** **DatabaseClient** **=** **new** **DatabaseClient(**host**,** post**,** credentials**)**  */\*\* The context used for the future callbacks \*/*  **implicit** **lazy** **val** executor**:** **ExecutionContext** **=** **ExecutionContext.**fromExecutor**(Executors.**directExecutor**())**  **override** **def** asyncInvoke**(**str**:** **String,** resultFuture**:** **ResultFuture[(String**, **String)]):** **Unit** **=** **{**  *// issue the asynchronous request, receive a future for the result*  **val** resultFutureRequested**:** **Future[String]** **=** client**.**query**(**str**)**  *// set the callback to be executed once the request by the client is complete*  *// the callback simply forwards the result to the result future*  resultFutureRequested**.**onSuccess **{**  **case** result**:** **String** **=>** resultFuture**.**complete**(Iterable((**str**,** result**)))**  **}**  **}**  **}**  *// create the original stream*  **val** stream**:** **DataStream[String]** **=** **...**  *// apply the async I/O transformation*  **val** resultStream**:** **DataStream[(String**, **String)]** **=**  **AsyncDataStream.**unorderedWait**(**stream**,** **new** **AsyncDatabaseRequest(),** 1000**,** **TimeUnit.MILLISECONDS,** 100**)** |

**特别注意**：ResultFuture 在第一次调用ResultFuture.complete就完成了，后续的complete 调用豆浆被忽略。

下面两个参数能够控制异步操作：

* **Timeout**:Timeout 定义了异步请求在获取数据前多求就会认为是失败了。这个参数能够保护死亡/失败的请求
* **Capacity**:这个参数定了在同一时刻能处理多少个异步请求。尽管async I/O可以带来更好的吞吐，但是operator在流应用中依然能成为瓶颈。限制并发请求数量可确保不会积累越来越多的待处理请求（pending request）,但是当容量（capacity）被耗尽时会引起反压。
  + - * 1. Timeout Handing(超时处理)

当async I/O 请求超时了，默认情况下将抛出一个异常并重启job. 但是如果你想处理超时的话，你可以重写 AsyncFunction#timeout 方法。

* + - * 1. Order of Results

AsyncFunction 发布的并发请求通常是无序的，为了控制结果记录发送的顺序，Flink提供了下面2个方法：

* **Unordered**：只要异步请求完成结果记录就会被发送。在stream中的record顺序在 async I/O 前后是不同的。这种方式会有更低的延迟和负载，当用processing time作为基本的 time characteristic 时，用AsyncDataStream.unorderedWait(...)启用这种模式。
* **Ordered：**在这个场景中，stream的顺序是被保留的。结果记录和异步请求被触发的顺序是相同的。为了达到这个目的，operator缓存发送之前的所有记录。这通常会在checkpoint中带来一些额外的延迟和负载，因为结果和记录需要在checkpoint状态中维护更长的事件，对比unordered 模式，使用AsyncDataStream.orderedWait(...) 启用这种模式。
  + - * 1. Event Time

当stream应用使用 [event time](https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/event_time.html)处理时，watermark将通过 the asynchronous I/O operator进行处理。有下面两种模式：

* **Unordered：Watermark不会超过记录，这意味着****watermark会建立一个有序的边界。被发送的记录只是在watermark之间是无序的。**
* **Ordered：记录的watermark是被保留的，就像记录的顺序被保留一样。对比**processing time处理没有明显的负载变化。
  + - * 1. Fault Tolerance Guarantees（容错保证）

asynchronous I/O operator 提供exactly-once 的容错性保证。他存储状态在checkpoint中

* + - * 1. Implementation Tips（实现提示）

对于具有callback执行器实现，我们建议使用DirectExecutor，因为callback通常都是非常小的工作，并且DirectExecutor可避免额外的线程到线程的负载。回调通常仅处理ResultFuture的结果。

DirectExecutor 可通过下面两种方式获取：

 org.apache.flink.runtime.concurrent.Executors.directExecutor()

com.google.common.util.concurrent.MoreExecutors.directExecutor().

* + - * 1. Caveat(警告)

**AsyncFunction 不是多线程**

**在这里我们想要明确指出的一个容易混淆的概念是**AsyncFunction 不是多线程。它仅仅是AsyncFunction 的一个实例，它在stream的各自分区的每天记录中循环调用。除非asyncInvoke(…)方法快速返回并依赖于一个回调(由客户端)，否则它不会导致正确的异步I/O。

例如：下面的模型在阻塞asyncInvoke(...)功能中避免异步行为

* 使用数据库client，其lookup/query 方法调用将阻塞，直到返回结果为止
* 阻塞/等待异步客户端在asyncInvoke(…)方法中返回的未来类型对象

## Connectors

# Batch (DataSet API)