**Apache Beam 使用指南（一）**

<https://blog.csdn.net/EdwardWong_?type=blog>

完整项目代码地址：<https://github.com/xiye50070/Apache-Beam-Model.git>

Apache Beam

一、概述

更多 Apache Beam 代码案例：https://github.com/xiye50070/Apache-Beam-Model.git

Apache Beam是一套编程模型和接口范式，是Google在2016年2月份贡献给Apache基金会的Apache孵化项目，其思想源自于Google DataFlow Model这篇论文。论文中作者论证了流处理系统其实是批处理系统的超集，即batch只是固定process/event time window下，利用atWaterMark trigger的数据处理模型。这套理论体系已经成为当前构建分布式流式处理系统的思想指导，而Beam正是这个思维的具象化实现。论文认为一个好的工具需要能够动态根据用户的使用场景、配置进行适应，具体的技术细节由工具本身消化，论文还认为需要摆脱目前一个主流的观点，认为执行引擎负责描述系统的语义。合理设计和构建的批，微批次，流处理系统能够保证同样程度的准确性。而这三种系统在处理无边界数据流时都非常常见。如果我们抽象出一个具有足够普遍性，灵活性的模型，那么执行引擎的选择就只是延迟程度和处理成本之间的选择。

Beam实现了计算逻辑的统一表达，本身并不处理数据，而是用来描述一个数据处理任务应该如何进行的，真正运行处理的是Runner。Beam目前支持Spark2.4+以及Flink1.11+

二、Apache Beam 能做什么

Aache Beam 将批处理和流式数据处理融合在一起，他可以做到一套符合Beam语义的代码，在多种执行引擎上运行，而且目前支持Java、Python、Go和 Scala语言，可以做到用户每个人都可以使用自己熟悉的语言来进行开发

Apache Beam 支持的执行引擎有：

Apache Flink

Apache Spark

Apache Apex

Apache Gearpump (incubating)

Apache Samza

Google Cloud Dataflow

Hazelcast Jet

三、Apache Beam 核心编程模型

Apache Beam 的核心概念有：

编程模型 说明

PCollection 数据集，代表了将要被处理的数据集合，可以是有限的数据集，也可以是无限的数据流。

PTransform 计算过程，代表了将输入数据集处理成输出数据集中间的计算过程。

Pipeline 管道，代表了处理数据的执行任务，可视作一个有向无环图（DAG），PCollections是节点，Transforms是边。

PipelineRunner 执行器，指定了Pipeline将要在哪里，怎样的运行。

ParDo 通用的并行处理的PTranform， 相当于Map/Shuffle/Reduce-style 中的Map，可用于过滤 、类型转换 、抽取部分数据 、 对数据中的每一个元素做计算等

Watermark 标记了多久时间后的延迟数据直接抛弃

Triggers 用来决定什么时候发送每个window的聚合结果

在使用Apache Beam开发时，需要考虑几个关键点

数据输入

输入的数据类型

数据要做什么样的转换（计算）

数据输出

Apache Beam 常用算子：

Element-wise：

算子 说明

Filter 给定一个谓词，过滤掉所有不满足谓词的元素。

FlatMapElements 应用一个函数，该函数将一个集合返回到输入中的每个元素并输出所有结果元素。

Keys 从键值对集合中的每个元素中提取键。

KvSwap 交换键值对集合中每个元素的键和值。

MapElements 将函数应用于输入中的每个元素并输出结果。

ParDo 将用户定义DoFn应用于输入集合中的每个元素的最通用机制。

Partition 根据某些分区函数将每个输入元素路由到特定的输出集合。

Regex 根据正则表达式过滤输入字符串元素。也可以根据匹配的组来转换它们。

Reify 用于在各种 Beam 值的显式和隐式形式之间进行转换的转换。

ToString 将输入集合中的每个元素转换为字符串。

WithKeys 生成一个集合，该集合包含输入集合中的每个元素，转换为键值对，并通过将函数应用于输入元素来选择键。

WithTimestamps 应用一个函数来确定输出集合中每个元素的时间戳，并更新与每个输入关联的隐式时间戳。请注意，仅向前调整时间戳是安全的。

Values 从键值对集合中的每个元素中提取值。

Aggregation

算子 说明

ApproximateQuantiles 使用近似算法使用指定的分位数估计每个聚合内的数据分布。

ApproximateUnique 使用近似算法来估计每个聚合中唯一元素的数量。

CoGroupByKey 类似于GroupByKey，但将与每个键关联的值分组到给定大小的批次中

Combine 根据提供的 转换以组合元素CombineFn。

CombineWithContext 允许访问侧输入和其他上下文的组合的扩展版本。

Count 计算每个聚合中的元素数。

Distinct 生成一个包含来自输入集合的不同元素的集合。

GroupByKey 获取元素的键控集合并生成一个集合，其中每个元素由一个键和与该键关联的所有值组成。

GroupIntoBatches 将与键关联的值批处理成Iterable一定大小的批处理。每个批次都包含与特定键关联的元素。

HllCount 使用 HyperLogLog++ 算法估计不同元素的数量并创建可重新聚合的草图。

Latest 根据隐式时间戳选择每个聚合中的最新元素。

Max 输出每个聚合中的最大元素。

Mean 计算每个聚合中的平均值。

Min 输出每个聚合中的最小元素。

Sample 从每个聚合中随机选择一些元素。

Sum 计算每个聚合中元素的总和。

Top 计算每个聚合中的最大元素。

Other

算子 说明

Create 从内存列表中创建一个集合。

Flatten 给定多个输入集合，生成一个包含来自所有输入集合的所有元素的输出集合。

PAssert 断言PCollection内容的转换，用于本地或与运行器一起测试管道的一部分。

View 将集合转换为视图的操作，可用作ParDo.

Window 根据提供的 .逻辑地将集合的元素划分或分组到有限的窗口中WindowFn。

ApacheBeam编程模型

1、单pipeline的Apache Beam的编程模型为：

2、多个Transform处理相同的PCollection

对于PCollection的多个转换并不会消耗PCollection中的数据，如下图所示，Pipeline从数据源读取输入，并创建了一个PCollection，然后Pipeline将多个Transform应用于同一个PCollection。Transform A 提取PCollection中所有以A开头的字符串，TransformB提取PCollection中所有以B开头的字符串，TransformA个TransformB具有相同的输入PCollection

代码示例：

PCollection<String> dbRowCollection = ...;

PCollection<String> aCollection = dbRowCollection.apply("aTrans", ParDo.of(new DoFn<String, String>(){

@ProcessElement

public void processElement(ProcessContext c) {

if(c.element().startsWith("A")){

c.output(c.element());

}

}

}));

PCollection<String> bCollection = dbRowCollection.apply("bTrans", ParDo.of(new DoFn<String, String>(){

@ProcessElement

public void processElement(ProcessContext c) {

if(c.element().startsWith("B")){

c.output(c.element());

}

}

}));

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

3、单Transform的多输出

对于多输出，在Beam中需要用到TupleTag来标记，每个输出可用用户自定义的Tag来确定，Tag定义好后需要注册，示例代码如下：

// 定义了两个TupleTag

final TupleTag<String> startsWithATag = new TupleTag<String>(){};

final TupleTag<String> startsWithBTag = new TupleTag<String>(){};

PCollectionTuple mixedCollection =

dbRowCollection.apply(ParDo

.of(new DoFn<String, String>() {

@ProcessElement

public void processElement(ProcessContext c) {

if (c.element().startsWith("A")) {

// 以A开头的在这个输出

c.output(c.element());

} else if(c.element().startsWith("B")) {

// 以B开头的在这个输出

c.output(startsWithBTag, c.element());

}

}

})

// 此处需要注册一下Tag

.withOutputTags(startsWithATag,

// 此处可以注册多个Tag

TupleTagList.of(startsWithBTag)));

// 针对TagA的处理方案

mixedCollection.get(startsWithATag).apply(...);

// 针对TagB的处理方案

mixedCollection.get(startsWithBTag).apply(...);

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

完整代码可看文档结尾的案例1

4、合并PCollections

当希望将多个PCollection合并在一起处理时，您可以使用以下操作之一来实现：

Flatten：可以使用Beam的Flatten算子来合并多个相同类型的PCollection；

Join：可以使用Beam的Join算子来实现两个PCollection之间的关系连接，PCollecion必须是KV类型的，并且Key的类型要相同。

以图为例，将第【3】节中分开数据进行合并：

代码示例：

//使用Flatten算子合并两个PCollection

PCollectionList<String> collectionList = PCollectionList.of(aCollection).and(bCollection);

PCollection<String> mergedCollectionWithFlatten = collectionList

.apply(Flatten.<String>pCollections());

// 继续使用合并后生成的新Pcollection

mergedCollectionWithFlatten.apply(...);

1

2

3

4

5

6

7

5、多输入

当Pipeline有多个输入时，可以将多输入读到的数据连接起来，

代码示例：

PCollection<KV<String, String>> userAddress = pipeline.apply(JdbcIO.<KV<String, String>>read()...);

PCollection<KV<String, String>> userOrder = pipeline.apply(KafkaIO.<String, String>read()...);

final TupleTag<String> addressTag = new TupleTag<String>();

final TupleTag<String> orderTag = new TupleTag<String>();

// 合并PCollection

PCollection<KV<String, CoGbkResult>> joinedCollection =

KeyedPCollectionTuple.of(addressTag, userAddress)

.and(orderTag, userOrder)

.apply(CoGroupByKey.<String>create());

joinedCollection.apply(...);

1

2

3

4

5

6

7

8

9

10

11

12

13

14

完整代码可看文档结尾的案例2

ApacheBeam 支持的数据源

1、基于文件

Name Description Javadoc

FileIO General-purpose transforms for working with files: listing files (matching), reading and writing. org.apache.beam.sdk.io.FileIO

AvroIO PTransforms for reading from and writing to Avro files. org.apache.beam.sdk.io.AvroIO

TextIO PTransforms for reading and writing text files. org.apache.beam.sdk.io.TextIO

TFRecordIO PTransforms for reading and writing TensorFlow TFRecord files. org.apache.beam.sdk.io.TFRecordIO

XmlIO Transforms for reading and writing XML files using JAXB mappers. org.apache.beam.sdk.io.xml.XmlIO

TikaIO Transforms for parsing arbitrary files using Apache Tika. org.apache.beam.sdk.io.tika.TikaIO

ParquetIO (guide) IO for reading from and writing to Parquet files. org.apache.beam.sdk.io.parquet.ParquetIO

ThriftIO PTransforms for reading and writing files containing Thrift-encoded data. org.apache.beam.sdk.io.thrift.ThriftIO

2、基于文件系统

姓名 描述 Javadoc

Hadoop文件系统 FileSystem访问Hadoop分布式文件系统文件的实现。 org.apache.beam.sdk.io.hdfs.HadoopFileSystemRegistrar

Gcs文件系统 FileSystem谷歌云存储的实现。 org.apache.beam.sdk.extensions.gcp.storage.GcsFileSystemRegistrar

本地文件系统 FileSystem实现访问磁盘上的文件。 org.apache.beam.sdk.io.LocalFileSystemRegistrar

S3文件系统 FileSystemAmazon S3的实施。 org.apache.beam.sdk.io.aws2.s3.S3FileSystemRegistrar（推荐）org.apache.beam.sdk.io.aws.s3.S3FileSystemRegistrar

3、基于消息队列

Name Description Javadoc

KinesisIO PTransforms for reading from and writing to Kinesis streams. org.apache.beam.sdk.io.aws2.kinesis.KinesisIO (recommended)org.apache.beam.sdk.io.kinesis.KinesisIO

AmqpIO AMQP 1.0 protocol using the Apache QPid Proton-J library org.apache.beam.sdk.io.amqp.AmqpIO

KafkaIO Read and Write PTransforms for Apache Kafka. org.apache.beam.sdk.io.kafka.KafkaIO

PubSubIO Read and Write PTransforms for Google Cloud Pub/Sub streams. org.apache.beam.sdk.io.gcp.pubsub.PubsubIO

JmsIO An unbounded source for JMS destinations (queues or topics). org.apache.beam.sdk.io.jms.JmsIO

MqttIO An unbounded source for MQTT broker. org.apache.beam.sdk.io.mqtt.MqttIO

RabbitMqIO A IO to publish or consume messages with a RabbitMQ broker. org.apache.beam.sdk.io.rabbitmq.RabbitMqIO

SqsIO An unbounded source for Amazon Simple Queue Service (SQS). org.apache.beam.sdk.io.aws2.sqs.SqsIO (recommended)org.apache.beam.sdk.io.aws.sqs.SqsIO

SnsIO PTransforms for writing to Amazon Simple Notification Service (SNS). org.apache.beam.sdk.io.aws2.sns.SnsIO (recommended)org.apache.beam.sdk.io.aws.sns.SnsIOorg.apache.beam.sdk.io.aws2.sns.SnsIO

Pub/Sub Lite I/O transforms for reading from Google Pub/Sub Lite. org.apache.beam.sdk.io.pubsublite.PubSubLiteIO

4、基于数据库

Name Description Javadoc

CassandraIO An IO to read from Apache Cassandra. org.apache.beam.sdk.io.cassandra.CassandraIO

HadoopFormatIO (guide) Allows for reading data from any source or writing data to any sink which implements Hadoop InputFormat or OutputFormat. org.apache.beam.sdk.io.hadoop.format.HadoopFormatIO

HBaseIO A bounded source and sink for HBase. org.apache.beam.sdk.io.hbase.HBaseIO

HCatalogIO (guide) HCatalog source supports reading of HCatRecord from a HCatalog-managed source, for example Hive. org.apache.beam.sdk.io.hcatalog.HCatalogIO

KuduIO A bounded source and sink for Kudu. org.apache.beam.sdk.io.kudu

SolrIO Transforms for reading and writing data from/to Solr. org.apache.beam.sdk.io.solr.SolrIO

ElasticsearchIO Transforms for reading and writing data from/to Elasticsearch. org.apache.beam.sdk.io.elasticsearch.ElasticsearchIO

BigQueryIO (guide) Read from and write to Google Cloud BigQuery. org.apache.beam.sdk.io.gcp.bigquery.BigQueryIO

BigTableIO Read from (only for Java SDK) and write to Google Cloud Bigtable. org.apache.beam.sdk.io.gcp.bigtable.BigtableIO

DatastoreIO Read from and write to Google Cloud Datastore. org.apache.beam.sdk.io.gcp.datastore.DatastoreIO

SnowflakeIO (guide) Experimental Transforms for reading from and writing to Snowflake. org.apache.beam.sdk.io.snowflake.SnowflakeIO

SpannerIO Experimental Transforms for reading from and writing to Google Cloud Spanner. org.apache.beam.sdk.io.gcp.spanner.SpannerIO

JdbcIO IO to read and write data on JDBC. org.apache.beam.sdk.io.jdbc.JdbcIO

MongoDbIO IO to read and write data on MongoDB. org.apache.beam.sdk.io.mongodb.MongoDbIO

MongoDbGridFSIO IO to read and write data on MongoDB GridFS. org.apache.beam.sdk.io.mongodb.MongoDbGridFSIO

RedisIO An IO to manipulate a Redis key/value database. org.apache.beam.sdk.io.redis.RedisIO

DynamoDBIO Read from and write to Amazon DynamoDB. org.apache.beam.sdk.io.aws2.dynamodb.DynamoDBIO (recommended)org.apache.beam.sdk.io.aws.dynamodb.DynamoDBIO

ClickHouseIO Transform for writing to ClickHouse. org.apache.beam.sdk.io.clickhouse.ClickHouseIO

InfluxDB IO to read and write from InfluxDB. org.apache.beam.sdk.io.influxdb.InfluxDbIO

Firestore IO FirestoreIO provides an API for reading from and writing to Google Cloud Firestore. org.apache.beam.sdk.io.gcp.healthcare.FirestoreIO

5、计划中的数据源

Name Language JIRA

Apache DistributedLog Java BEAM-607

Apache Sqoop Java BEAM-67

Couchbase Java BEAM-1893

Memcached Java BEAM-1678

RestIO Java BEAM-1946

NATS IO

四、窗口

ApacheBeam支持的窗口类型有四种

Fixed Time Windows 固定时间窗口

Sliding Time Windows 滑动时间窗口

Per-Session Windows 基于会话的时间窗口

Single Global Window 全局单一时间窗口

Calendar-based Windows (not supported by the Beam SDK for Python or Go) 基于日历的时间窗口（Python和Go不支持）

1、固定时间窗口

最简单的窗口形式，如图所示

代码示例

PCollection<String> items = ...;

PCollection<String> fixedWindowedItems = items.apply(

Window.<String>into(FixedWindows.of(Duration.standardSeconds(60))));

1

2

3

2、滑动时间窗口

滑动时间窗口也表示数据流中的时间间隔，但是由于滑动时间窗口可以重叠，数据集中的部分元素将属于多个窗口。如图所示：

代码示例：

PCollection<String> items = ...;

PCollection<String> slidingWindowedItems = items.apply(

Window.<String>into(SlidingWindows.of(Duration.standardSeconds(30)).every(Duration.standardSeconds(5))));

1

2

3

3、会话窗口

会话窗口函数定义了包含在另一个元素的特定间隙持续时间内的元素的窗口。会话窗口对于时间上不规则分布的数据很有用。例如，表示用户鼠标活动的数据流可能有很长的空闲时间，其中穿插着大量的点击。如果数据在指定的最小间隙持续时间之后到达，这将启动新窗口的开始。如图所示

代码示例：

PCollection<String> items = ...;

PCollection<String> sessionWindowedItems = items.apply(

Window.<String>into(Sessions.withGapDuration(Duration.standardSeconds(600))));

1

2

3

4、单一全局窗口

适用于数据是有界的，可以将所有元素分配给一个全局窗口，代码示例：

PCollection<String> items = ...;

PCollection<String> batchItems = items.apply(

Window.<String>into(new GlobalWindows()));

1

2

3

五、关于ApacheBeam的水位线（水印）机制和触发器机制

1、关于什么是水印和延迟数据

在任何数据处理系统中，数据事件发生的时间（“事件时间”，由数据元素本身的时间戳决定）和实际数据元素在任何阶段被处理的时间之间都有一定的延迟。您的管道（“处理时间”，由处理元素的系统时钟决定）。此外，无法保证数据事件会按照它们生成的顺序出现在您的管道中。

例如，假设我们有一个PCollection使用固定时间窗口的窗口，窗口长度为五分钟。对于每个窗口，Beam 必须在给定的窗口范围内（例如，在第一个窗口中的 0:00 到 4:59 之间）收集具有\*事件时间时间戳的所有数据。\*时间戳超出该范围的数据（5:00 或更晚的数据）属于不同的窗口。

然而，数据并不总是保证按时间顺序到达管道，或者总是以可预测的时间间隔到达。Beam 跟踪watermark，这是系统的概念，即某个窗口中的所有数据预计何时到达管道。一旦水印超过了窗口的末尾，在该窗口中到达的带有时间戳的任何其他元素都被认为是 迟到的数据。

在我们的示例中，假设我们有一个简单的水印，假设数据时间戳（事件时间）和数据出现在管道中的时间（处理时间）之间存在大约 30 秒的延迟时间，那么 Beam 将在 5 点关闭第一个窗口:30。如果数据记录在 5:34 到达，但带有将其置于 0:00-4:59 窗口（例如 3:38）的时间戳，则该记录是迟到的数据。

注意：为简单起见，我们假设我们正在使用一个非常简单的水印来估计延迟时间。在实践中，你PCollection的数据源决定了水印，水印可以更精确也可以更复杂。

Beam 的默认窗口配置尝试确定所有数据何时到达（基于数据源的类型），然后将水印推进到窗口末尾。此默认配置不允许延迟数据。触发器允许您修改和优化PCollection. 您可以使用触发器来决定每个单独的窗口何时聚合并报告其结果，包括窗口如何发出迟到的元素。

2、延迟数据的处理

可以通过在设置的窗口策略时调用.withAllowedLateness()操作来处理延迟数据。下面的代码示例演示了一个窗口策略，该策略将允许在窗口结束后最多两天的延迟数据。

PCollection<String> items = ...;

PCollection<String> fixedWindowedItems = items.apply(

Window.<String>into(FixedWindows.of(Duration.standardMinutes(1)))

.withAllowedLateness(Duration.standardDays(2)));

1

2

3

4

3、为PCollection中的每个元素添加时间戳

ApacheBeam不会为有界数据提供时间戳，如果需要可以使用一下方法添加到元素中

PCollection<LogEntry> unstampedLogs = ...;

PCollection<LogEntry> stampedLogs =

unstampedLogs.apply(ParDo.of(new DoFn<LogEntry, LogEntry>() {

public void processElement(@Element LogEntry element, OutputReceiver<LogEntry> out) {

// Extract the timestamp from log entry we're currently processing.

Instant logTimeStamp = extractTimeStampFromLogEntry(element);

// Use OutputReceiver.outputWithTimestamp (rather than

// OutputReceiver.output) to emit the entry with timestamp attached.

out.outputWithTimestamp(element, logTimeStamp);

}

}));

1

2

3

4

5

6

7

8

9

10

11

4、触发器triggers

在收集数据并将数据分组到窗口中时，Beam 使用触发器来确定何时发出每个窗口（称为 窗格）的聚合结果。如果您使用 Beam 的默认窗口配置和默认触发器，Beam 在估计所有数据已到达时输出聚合结果 ，并丢弃该窗口的所有后续数据。

您可以手动设置触发器来更改此默认行为，Beam提供了很多的预置触发器：

Event time triggers 事件时间触发器：以事件事件为基础进行操作

Processing time triggers 处理时间触发器：以处理时间为基础进行操作

Data-driven triggers 数据驱动触发器：在数据到达窗口时对其检查，满足特定属性时出发

Composite triggers 复合触发器：支持多种触发器的组合

触发器的存在相比于基本的窗口模式，有两个额外的功能：

触发器允许Beam在给定窗口中的所有数据到达之前处理出基于已到达的数据的早期结果

触发器允许 事件时间水印经过窗口后来触发处理延迟的数据

这些功能可以根据用例控制数据流，并在不同的维度之间取得平衡：

完整性：需要有多少数据时就可以开始进行计算

延迟：可以接受的延迟时间是多久，延迟数据到达后是否进行处理

成本：您愿意花费多少计算能力/金钱来降低延迟

例如，时间敏感的业务，重视及时性而不是数据完整性的场景，可能会使用更加严格的基于时间的触发器，该触发器每N秒发出一个窗口。而重视数据完整性但不是很重视及时性的场景，就可以使用Beam的默认触发器，在窗口结束时触发计算。

4.1、事件时间触发器

AfterWatermark触发器是基于事件时间的，触发器根据数据元素所附加的时间戳，在水印通过窗口后发出窗口的内容。此外，您可以配置在您的Pipeline在窗口结束之前或结束之后接受数据时触发触发器，下面的代码示例展示了提前和延时触发

// 在每月末创建订单

AfterWatermark.pastEndOfWindow()

// During the month, get near real-time estimates.

.withEarlyFirings(

AfterProcessingTime

.pastFirstElementInPane()

.plusDuration(Duration.standardMinutes(1))

// 对延时到达的数据进行处理，来修正订单信息

.withLateFirings(AfterPane.elementCountAtLeast(1))

1

2

3

4

5

6

7

8

9

PCollection的默认触发器是基于事件时间的，当Beam的水印通过窗口末尾时发出窗口的结果，如果同时使用默认窗口和默认触发器，则默认触发器仅会触发一次，并且将延迟数据丢弃。

4.2、处理时间触发器

AfterProcessingTime触发器是基于处理时间的，例如，触发器会在收到数据后经过一定的处理时间后发出一个窗口。处理时间由系统时钟决定，而不是数据元素的时间戳。AfterProcessingTime.pastFirstElementInPane()。

处理时间触发器对于不在意数据完整性，在意及时性的场景比较有用，可以在完整数据获得前提前进行数据计算并得到当前的结果。

4.3、数据驱动触发器

Beam 提供了一个数据驱动的触发器，此触发器适用于元素计数，它在当前窗口收集了至少N个元素后触发。这允许窗口发出早起结果（在所有数到来之前）。

5、使用触发器

您可以通过在Transform的结果上调用方法来设置触发器，如下代码示例，为一个PCollection设置了一个处理时间触发器，它在窗口中的第一个元素被处理后一分钟发出当前窗口的结果，代码示例中的最后一行，设置窗口的累积模式。.triggering()Window.into()PCollection.discardingFiredPanes()

PCollection<String> pc = ...;

pc.apply(Window.<String>into(FixedWindows.of(1, TimeUnit.MINUTES))

.triggering(AfterProcessingTime.pastFirstElementInPane()

.plusDelayOf(Duration.standardMinutes(1)))

.discardingFiredPanes());

1

2

3

4

5

5.1、窗口累积模式

在指定触发器时，还必须设置窗口的累积模式。当触发器触发时，它将窗口的当前内容作为窗格发出。由于触发器可以触发多次，因此累积模式决定了系统是在触发器触发时累积窗格还是丢弃它们。

要设置一个窗口来累积触发器触发时生成的窗格，需要在设置触发器时调用.accumulatingFiredPanes()方法。要设置丢弃模式，需要调用.discardingFiredPanes()。

例如，如果每个窗口代表10分钟的运行平均值，但您希望比每10分钟更频繁的显示计算结果，您可以这样做：

使用PCollection的10分钟的固定时间窗口

设置一个基于数据的触发器，每次3个元素到达时触发

以下图为例

如果我们的触发器设置为累加模式，则触发器每次触发时都会发出以下值（每次三个元素到达时触发器都会触发）

First trigger firing: [5, 8, 3]

Second trigger firing: [5, 8, 3, 15, 19, 23]

Third trigger firing: [5, 8, 3, 15, 19, 23, 9, 13, 10]

1

2

3

如果我们的触发器设置为丢弃模式，则触发器在每次触发时会发送以下值：

First trigger firing: [5, 8, 3]

Second trigger firing: [15, 19, 23]

Third trigger firing: [9, 13, 10]

1

2

3

5.2、处理延时数据

如果希望pipeline处理在水印通过窗口结束后到达的延时数据，可以在设置窗口配置时设置允许延迟。这使触发器可以对迟到的数据做出反应。如果设置了允许延迟，默认触发器将在延迟数据到达时立即发出新的结果

在设置窗口函数时调用.withAllowedLateness()方法来设置允许延迟。

PCollection<String> pc = ...;

pc.apply(Window.<String>into(FixedWindows.of(1, TimeUnit.MINUTES))

.triggering(AfterProcessingTime.pastFirstElementInPane()

.plusDelayOf(Duration.standardMinutes(1)))

.withAllowedLateness(Duration.standardMinutes(30));

1

2

3

4

5

6、复合触发器

可以针对场景和业务需求组合多个触发器以形成复合触发器，并且可以指定一个触发器多次触发、最多触发一次或其他自定义条件下触发

Beam 包含以下复合触发器：

您可以通过调用.withEarlyFirings和.withLateFirings将额外的提前触发或延后触发添加到AfterWatermark.pastEndOfWindow。

Repeatedly.forever 指定重复执行的触发器，每当满足触发条件时，都会将窗口结果发出，然后重制并重新开始，使用.orFinally来设置重复停止的条件。

AfterEach.inOrder组合多个触发器按特定顺序触发。

AfterFirst在由多个触发器组成的组合中，满足任一触发器的触发条件时触发所有触发器

AfterAll在由多个触发器组成的组合中，满足所有触发器的触发条件时触发所有触发器

orFinally在由多个触发器组成的组合中，任一触发器触发后停止

例如以下以下场景：

在水印通过窗口末尾之前的推测性触发，以允许更快的处理部分结果

水印通过窗口结束后延迟触发，以允许处理延迟到达的数据

如下面的案例：

水印过了窗口末端

任何数据延迟到达，延迟10分钟

2天后触发器停止执行

.apply(Window

.configure()

.triggering(AfterWatermark

.pastEndOfWindow()

.withLateFirings(AfterProcessingTime

.pastFirstElementInPane()

.plusDelayOf(Duration.standardMinutes(10))))

.withAllowedLateness(Duration.standardDays(2)));

1

2

3

4

5

6

7

8

还可以自定义构建复合触发器，如一下案例，触发器的窗格具有至少100个元素或在一分钟后触发：

Repeatedly.forever(AfterFirst.of(

AfterPane.elementCountAtLeast(100),

AfterProcessingTime.pastFirstElementInPane().plusDelayOf(Duration.standardMinutes(1))))

1

2

3

六、说明示例代码

示例代码1

package com.yss.beam;

import org.apache.beam.sdk.Pipeline;

import org.apache.beam.sdk.coders.StringUtf8Coder;

import org.apache.beam.sdk.options.PipelineOptions;

import org.apache.beam.sdk.options.PipelineOptionsFactory;

import org.apache.beam.sdk.transforms.Create;

import org.apache.beam.sdk.transforms.DoFn;

import org.apache.beam.sdk.transforms.ParDo;

import org.apache.beam.sdk.values.PCollectionTuple;

import org.apache.beam.sdk.values.TupleTag;

import org.apache.beam.sdk.values.TupleTagList;

/\*\*

\* @author MingZhang Wang

\*/

public class BeamMutiOutput {

public static void main(String[] args) {

final TupleTag<String> endWithWorldTag = new TupleTag<String>();

final TupleTag<String> endWithSparkTag = new TupleTag<String>();

final TupleTag<String> endWithFlinkTag = new TupleTag<String>();

PipelineOptions options = PipelineOptionsFactory.create();

Pipeline pipeline = Pipeline.create(options);

PCollectionTuple resultTuple = pipeline

.apply(Create.of("hello world", "hello flink", "hello flink spark"))

.apply(ParDo.of(new DoFn<String, String>() {

@ProcessElement

public void processElement(@Element String word, MultiOutputReceiver out) {

if (word.endsWith("world")) {

out.get(endWithWorldTag).output(word);

} else if (word.endsWith("spark")) {

out.get(endWithSparkTag).output(word);

} else {

out.get(endWithFlinkTag).output(word);

}

}

}).withOutputTags(endWithFlinkTag, TupleTagList.of(endWithSparkTag).and(endWithWorldTag)));

resultTuple.get(endWithFlinkTag).setCoder(StringUtf8Coder.of()).apply(ParDo.of(new DoFn<String, String>() {

@ProcessElement

public void processElement(@Element String element){

System.out.println("flink Tag " + element);

}

}));

resultTuple.get(endWithSparkTag).setCoder(StringUtf8Coder.of()).apply(ParDo.of(new DoFn<String, String>() {

@ProcessElement

public void processElement(ProcessContext context){

String element = context.element();

System.out.println("spark Tag " + element);

context.output(element);

}

}));

resultTuple.get(endWithWorldTag).setCoder(StringUtf8Coder.of()).apply(ParDo.of(new DoFn<String, String>() {

@ProcessElement

public void processElement(ProcessContext context){

String element = context.element();

System.out.println("world Tag " + element);

context.output(element);

}

}));

pipeline.run();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

示例代码2

package com.yss.beam;

import org.apache.beam.sdk.Pipeline;

import org.apache.beam.sdk.coders.Coder;

import org.apache.beam.sdk.coders.KvCoder;

import org.apache.beam.sdk.coders.ListCoder;

import org.apache.beam.sdk.coders.StringUtf8Coder;

import org.apache.beam.sdk.options.PipelineOptions;

import org.apache.beam.sdk.options.PipelineOptionsFactory;

import org.apache.beam.sdk.transforms.\*;

import org.apache.beam.sdk.transforms.join.\*;

import org.apache.beam.sdk.values.\*;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class BeamMutiInput {

public static void main(String[] args) {

PipelineOptions options = PipelineOptionsFactory.create();

Pipeline pipeline = Pipeline.create(options);

final TupleTag<Long> ATag = new TupleTag<Long>(){};

final TupleTag<Long> BTag = new TupleTag<Long>(){};

List<Coder<?>> stringUtf8Coders = new ArrayList<>();

PCollection<KV<String, Long>> inputA = pipeline.apply(Create.of("Ahello Aworld", "Ahello Aflink", "Ahello Aflink spark"))

.apply(FlatMapElements.into(TypeDescriptors.strings()).via(line -> Arrays.asList(line.split(" "))))

.apply(Count.perElement());

PCollection<KV<String, Long>> inputB = pipeline.apply(Create.of("Bhello Aworld", "Bhello Aflink", "Bhello Aflink spark"))

.apply(FlatMapElements.into(TypeDescriptors.strings()).via(line -> Arrays.asList(line.split(" "))))

.apply(Count.perElement());

PCollection<KV<String, CoGbkResult>> joinedCollection = KeyedPCollectionTuple.of(ATag, inputA)

.and(BTag, inputB)

.apply(CoGroupByKey.<String>create())

.apply(ParDo.of(new DoFn<KV<String, CoGbkResult>, KV<String, CoGbkResult>>() {

@ProcessElement

public void processElement(ProcessContext context){

KV<String, CoGbkResult> element = context.element();

System.out.println(element);

context.output(element);

}

}));

pipeline.run();

}

}