1.概述

Apache Beam is an open source, unified model for defining both batch and streaming data-parallel processing pipelines. Using one of the open source Beam SDKs, you build a program that defines the pipeline. The pipeline is then executed by one of Beam’s supported **distributed processing back-ends**, which include [Apache Apex](http://apex.apache.org/), [Apache Flink](http://flink.apache.org/), [Apache Spark](http://spark.apache.org/), and [Google Cloud Dataflow](https://cloud.google.com/dataflow).

Beam is particularly useful for [Embarrassingly Parallel](http://en.wikipedia.org/wiki/Embarassingly_parallel) data processing tasks, in which the problem can be decomposed into many smaller bundles of data that can be processed independently and in parallel. You can also use Beam for Extract, Transform, and Load (ETL) tasks and pure data integration. These tasks are useful for moving data between different storage media and data sources, transforming data into a more desirable format, or loading data onto a new system.

Apache Beam 是一款开源的分布式计算框架，采用统一的模型定义批处理与流式数据并行处理管道。可以在Apache Spark、Flink、Apex和Google Dataflow上运行。

To use Beam, you need to first create a driver program using the classes in one of the Beam SDKs. Your driver program *defines* your pipeline, including all of the inputs, transforms, and outputs; it also sets execution options for your pipeline (typically passed in using command-line options). These include the Pipeline Runner, which, in turn, determines what back-end your pipeline will run on.

要使用Beam, 您首先需要使用一种Beam SDK（可以是Java版本、Python版本或其他语言版本）中类创建一个驱动程序。在你的驱动程序中定义管道(Pipeline), 包括所有输入(Inputs)、转换(Transforms)和输出(Outputs)；同时，它还为管道(Pipeline)设置执行选项 (通常使用命令行选项传递)。其中包括管道运行器（Pipeline Runner）, 它反过来又决定了管道(Pipeline)将在哪种后台(支持Spark、Apex、Flink等分布式计算平台)运行。

The Beam SDKs provide a number of abstractions that simplify the mechanics of large-scale distributed data processing. The same Beam abstractions work with both batch and streaming data sources. When you create your Beam pipeline, you can think about your data processing task in terms of these abstractions. They include:

Beam SDKs提供了许多抽象来简化大规模分布式数据处理的机制。相同的Beam抽象在批处理和流数据源中都起作用。当你创建Beam管道(Pipeline)时, 可以根据这些抽象来考虑你的数据处理任务。主要包括:

* Pipeline: A Pipeline encapsulates your entire data processing task, from start to finish. This includes reading input data, transforming that data, and writing output data. All Beam driver programs must create a Pipeline. When you create the Pipeline, you must also specify the execution options that tell the Pipeline where and how to run.
* Pipeline: 封装从开始到完成的整个数据处理任务，包括读取输入数据、转换数据和输出数据。所有Beam驱动程序都必须创建一个管道(Pipeline)。创建管道(Pipeline)时, 还必须指定执行选项(Execution Options), 以指示管道(Pipeline)在何处以及如何运行。
* PCollection: A PCollection represents a distributed data set that your Beam pipeline operates on. The data set can be *bounded*, meaning it comes from a fixed source like a file, or *unbounded*, meaning it comes from a continuously updating source via a subscription or other mechanism. Your pipeline typically creates an initial PCollection by reading data from an external data source, but you can also create a PCollection from in-memory data within your driver program. From there, PCollections are the inputs and outputs for each step in your pipeline.
* PCollection: 表示Beam管道所运行的分布式数据集(Distributed Data Set)。数据集可以是有界的, 这意味着它可以来自一个固定的源, 如文件, 或者是无界的, 这意味着它可以来自于通过订阅或其他机制不断更新的源。管道通常通过从外部数据源读取数据来创建初始化PCollection, 但也可以从驱动程序中的内存数据来创建 PCollection。PCollections 是管道中每一步的输入和输出。
* PTransform: A PTransform represents a data processing operation, or a step, in your pipeline. Every PTransform takes one or more PCollection objects as input, performs a processing function that you provide on the elements of that PCollection, and produces zero or more output PCollection objects.
* PTransform: 表示管道中的一个数据处理操作或步骤。每个PTransform 都将一个或多个PCollection对象作为输入, 执行您在该PCollection元素上提供的处理函数, 并生成零个或多个PCollection对象输出。
* I/O transforms: Beam comes with a number of “IOs” - library PTransforms that read or write data to various external storage systems.
* I/O转换: Bean附带了许多输入输出库的 PTransforms, 它们可以从各种外部存储系统中读取或写入数据。

A typical Beam driver program works as follows:

一个典型的Beam驱动程序的工作流程如下:

* Create a Pipeline object and set the pipeline execution options, including the Pipeline Runner.
* 创建管道对象, 并设置管道执行选项, 包括管道运行器。
* Create an initial PCollection for pipeline data, either using the IOs to read data from an external storage system, or using a Create transform to build a PCollection from in-memory data.
* 为管道数据创建初始PCollection，方法是使用 IOs 从外部存储系统读取数据, 或者使用创建转换从内存中的数据生成PCollection。
* Apply **PTransforms** to each PCollection. Transforms can change, filter, group, analyze, or otherwise process the elements in a PCollection. A transform creates a new output PCollection *without consuming the input collection*. A typical pipeline applies subsequent transforms to the each new output PCollection in turn until processing is complete. However, note that a pipeline does not have to be a single straight line of transforms applied one after another: think of PCollections as variables and PTransforms as functions applied to these variables: the shape of the pipeline can be an arbitrarily complex processing graph.
* 将PTransforms 应用于每个PCollection。转换可以更改、筛选、分组、分析或以其他方式处理PCollection中的元素。转换创建新的输出PCollection而不消耗输入集合。典型管道依次将后续转换应用于每个新的输出PCollection, 直到处理完成为止。但是, 请注意管道不必是一个接一个地应用的直线变换， 将PCollections 作为变量并将PTransforms 作为应用于这些变量的函数，管道的形状可以是任意复杂的处理图。
* Use IOs to write the final, transformed PCollection(s) to an external source.
* 使用 IOs 将最终的、转换后的 PCollection 写入外部源。
* **Run** the pipeline using the designated Pipeline Runner.
* 使用指定的管道运行器运行管道。

When you run your Beam driver program, the Pipeline Runner that you designate constructs a **workflow graph** of your pipeline based on the PCollection objects you’ve created and transforms that you’ve applied. That graph is then executed using the appropriate distributed processing back-end, becoming an asynchronous “job” (or equivalent) on that back-end.

运行Beam驱动程序时, 指定的管道运行器根据你创建的PCollection对象和已应用的转换来构造管道的工作流图 (Workflow Graph)。然后，使用适当的分布式处理后台执行该图, 成为该后台的等效的异步作业。

## 2. 创建管道

The Pipeline abstraction encapsulates all the data and steps in your data processing task. Your Beam driver program typically starts by constructing a [Pipeline](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/index.html?org/apache/beam/sdk/Pipeline.html) [Pipeline](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/pipeline.py) object, and then using that object as the basis for creating the pipeline’s data sets as PCollections and its operations as Transforms.

管道抽象封装数据处理任务中的所有数据和步骤。Beam驱动程序通常是通过构造管道对象来启动的, 然后使用该对象作为创建管道数据集PCollections 及其操作变换的基础。

To use Beam, your driver program must first create an instance of the Beam SDK class Pipeline (typically in the main() function). When you create your Pipeline, you’ll also need to set some **configuration options**. You can set your pipeline’s configuration options programatically, but it’s ofte-n easier to set the options ahead of time (or read them from the command line) and pass them to the Pipeline object when you create the object.

要使用Beam, 驱动程序必须首先创建Beam SDK类Pipeline的实例 (通常在主函数中)。创建管道时, 还需要设置一些配置选项。可以通过编程的方式设置管道的配置选项；但通常更容易提前设置选项 (或从命令行读取它们), 并在创建对象时将它们传递给管道对象。

*// Start by defining the options for the pipeline.*

PipelineOptions options **=** PipelineOptionsFactory**.**create**();**

*// Then create the pipeline.*

Pipeline p **=** Pipeline**.**create**(**options**);**

### 2.1.配置管道选项

Use the pipeline options to configure different aspects of your pipeline, such as the pipeline runner that will execute your pipeline and any runner-specific configuration required by the chosen runner. Your pipeline options will potentially include information such as your project ID or a location for storing files.

使用管道选项可以配置管道的不同方面, 例如将执行管道的管道运行器以及所选的运行器所需的任何特定配置。管道选项可能包含诸如项目ID或存储文件的位置等信息。

When you run the pipeline on a runner of your choice, a copy of the PipelineOptions will be available to your code. For example, you can read PipelineOptions from a DoFn’s Context.

当你在选择的运行器上运行管道时，管道选项( PipelineOptions)的副本将可用于你的代码。例如, 你可以从 DoFn 的上下文中读取管道选项( PipelineOptions)。

#### 2.1.1.从命令行参数设置管道选项

While you can configure your pipeline by creating a PipelineOptions object and setting the fields directly, the Beam SDKs include a command-line parser that you can use to set fields in PipelineOptions using command-line arguments.

虽然可以通过创建PipelineOptions对象后直接设置其字段来配置管道, 但Beam SDK包括了一个命令行分析器, 可用于使用命令行参数在PipelineOptions 中设置字段。

To read options from the command-line, construct your PipelineOptions object as demonstrated in the following example code:

若要从命令行读取选项, 构造 PipelineOptions对象, 如下面的示例代码所示:

MyOptions options **=** PipelineOptionsFactory**.**fromArgs**(**args**).**withValidation**().**create**();**

This interprets command-line arguments that follow the format:

这将解释遵循以下格式的命令行参数:

--<option>=<value>

**Note:** Appending the method .withValidation will check for required command-line arguments and validate argument values.

Building your PipelineOptions this way lets you specify any of the options as a command-line argument.

**注意**: 追加方法.withValidation将检查所需的命令行参数并验证参数值。通过这种方式构建的PipelineOptions, 可以将任何选项指定为命令行参数。

#### 2.1.2.创建自定义选项

You can add your own custom options in addition to the standard PipelineOptions. To add your own options, define an interface with getter and setter methods for each option, as in the following example:

除了标准PipelineOptions之外, 你还可以添加自己的自定义选项。要添加你自己的选项, 需要为每个选项定义一个带有getter和setter方法的接口, 如下面的示例所示:

**public** **interface** **MyOptions** **extends** PipelineOptions **{**

String **getMyCustomOption();**

**void** **setMyCustomOption(**String myCustomOption**);**

**}**

You can also specify a description, which appears when a user passes --help as a command-line argument, and a default value.

你还可以指定一段描述，当用户传入--help作为命令行参数的时候显示，也可以指定默认值。

You set the description and default value using annotations, as follows:

使用注释设置描述和默认值, 如下所示:

**public** **interface** **MyOptions** **extends** PipelineOptions **{**

@Description**(**"My custom command line argument."**)**

@Default**.**String**(**"DEFAULT"**)**

String **getMyCustomOption();**

**void** **setMyCustomOption(**String myCustomOption**);**

**}**

It’s recommended that you register your interface with PipelineOptionsFactory and then pass the interface when creating the PipelineOptions object. When you register your interface with PipelineOptionsFactory, the --help can find your custom options interface and add it to the output of the --help command. PipelineOptionsFactory will also validate that your custom options are compatible with all other registered options.

建议使用PipelineOptionsFactory 注册你自动定义的管道选项接口, 然后在创建 PipelineOptions对象时传递该接口。当你向PipelineOptionsFactory 注册了接口后,--help可以找到你的自定义选项接口并将其添加到--help 命令的输出中。PipelineOptionsFactory 还将验证你的自定义选项是否与所有其他注册选项兼容。

The following example code shows how to register your custom options interface with PipelineOptionsFactory:

下面的代码示例演示如何使用 PipelineOptionsFactory 注册自定义选项接口:

PipelineOptionsFactory**.**register**(**MyOptions**.**class**);**

MyOptions options **=** PipelineOptionsFactory**.**fromArgs**(**args**)**

**.**withValidation**()**

**.**as**(**MyOptions**.**class**);**

Now your pipeline can accept --myCustomOption=value as a command-line argument.

现在你的管道就可以接受--myCustomOption=value作为命令行参数了。

## 3. PCollections

The [PCollection](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/index.html?org/apache/beam/sdk/values/PCollection.html) PCollection abstraction represents a potentially distributed, multi-element data set. You can think of a PCollection as “pipeline” data; Beam transforms use PCollection objects as inputs and outputs. As such, if you want to work with data in your pipeline, it must be in the form of a PCollection.

PCollectio抽象表示一个潜在的分布式多元素数据集。你可以认为PCollection就是 "管道" 数据;Beam变换使用PCollection对象作为输入和输出。因此, 如果要处理管道中的数据, 它必须是PCollection的形式。

After you’ve created your Pipeline, you’ll need to begin by creating at least one PCollection in some form. The PCollection you create serves as the input for the first operation in your pipeline.

创建管道后, 首先需要以某种方式创建至少一个PCollection。创建的 PCollection 作为管道中第一个操作的输入。

### 3.1. 创建PCollection

You create a PCollection by either reading data from an external source using Beam’s [Source API](https://beam.apache.org/documentation/programming-guide/#pipeline-io), or you can create a PCollection of data stored in an in-memory collection class in your driver program. The former is typically how a production pipeline would ingest data; Beam’s Source APIs contain adapters to help you read from external sources like large cloud-based files, databases, or subscription services. The latter is primarily useful for testing and debugging purposes.

你通过使用Beam的源 API 从外部源读取数据创建PCollection, 也可以通过存储在驱动程序中内存集合类中的数据来创建PCollection。前者通常是生产流水线如何摄取数据；Beam源API 包含适配器, 可帮助您从外部源, 如基于云的大型文件、数据库或订阅服务等中读取数据。后者主要用于测试和调试目的。

#### 3.1.1. 从外部源读取

To read from an external source, you use one of the [Beam-provided I/O adapters](https://beam.apache.org/documentation/programming-guide/#pipeline-io). The adapters vary in their exact usage, but all of them from some external data source and return a PCollection whose elements represent the data records in that source.

要从外部源读取, 请使用由Beam提供的I/O适配器。适配器的确切使用情况不同, 但它们都来自外部数据源, 并返回一个 PCollection, 其元素表示该数据源中的数据记录。

Each data source adapter has a Read transform; to read, you must apply that transform to the Pipeline object itself. TextIO.Readio.TextFileSource, for example, reads from an external text file and returns a PCollection whose elements are of type String, each String represents one line from the text file. Here’s how you would apply TextIO.Read io.TextFileSource to your Pipeline to create a PCollection:

每个数据源适配器都有一个读取转换；若要读取，必须将该转换应用于管道对象本身。例如, TextIO.Read从一个外部文本文件中读取, 并返回一个元素为字符串类型的PCollection，其中，每个字符串表示文本文件中的一行。下面是如何应用TextIO的方法读取数据，并通过管道创建 PCollection:

// **CreatePCollectionFromExternalSource.java**

**public** **static** **void** **main(**String**[]** args**)** **{**

*// Create the pipeline.*

PipelineOptions options **=**

PipelineOptionsFactory**.**fromArgs**(**args**).**create**();**

Pipeline p **=** Pipeline**.**create**(**options**);**

*// Create the PCollection 'lines' by applying a 'Read' transform.*

PCollection**<**String**>** lines **=** p**.**apply**(**

"ReadMyFile"**,**

TextIO**.**read**().**from**(**"protocol://path/to/some/inputData.txt"**));**

**}**

#### 3.1.2. 从内存数据创建PCollection

To create a PCollection from an in-memory Java Collection, you use the Beam-provided Create transform. Much like a data adapter’s Read, you apply Create directly to your Pipeline object itself. As parameters, Create accepts the Java Collection and a Coder object. The Coder specifies how the elements in the Collection should be [encoded](https://beam.apache.org/documentation/programming-guide/#element-type). To create a PCollection from an in-memory list, you use the Beam-provided Create transform. Apply this transform directly to your Pipeline object itself. The following example code shows how to create a PCollection from an in-memory Listlist:

要从内存中的 Java 集合创建 PCollection, 可以使用Beam提供的创建转换(Create)。与数据适配器的读取非常相似, 你可以直接将创建应用于管道对象本身。作为参数, 创建(Create) 接受 Java 集合和一个编码器(Coder)对象。编码器指定如何对集合中的元素如何被编码。要从内存列表中创建PCollection，需要使用Beam提供的创建转换，并将此转换直接应用于管道对象本身。下面的示例代码演示如何从内存中的列表创建PCollection：

**// CreatePCollectionFromInmemoryCollection.java**

**public** **static** **void** **main(**String**[]** args**)** **{**

*// Create a Java Collection, in this case a List of Strings.*

**static** **final** List**<**String**>** LINES **=** Arrays**.**asList**(**

"To be, or not to be: that is the question: "**,**

"Whether 'tis nobler in the mind to suffer "**,**

"The slings and arrows of outrageous fortune, "**,**

"Or to take arms against a sea of troubles, "**);**

*// Create the pipeline.*

PipelineOptions options **=**

PipelineOptionsFactory**.**fromArgs**(**args**).**create**();**

Pipeline p **=** Pipeline**.**create**(**options**);**

*// Apply Create, passing the list and the coder, to create the PCollection.*

p**.**apply**(**Create**.**of**(**LINES**)).**setCoder**(**StringUtf8Coder**.**of**())**

**}**

### 3.2. PCollection的特征

A PCollection is owned by the specific Pipeline object for which it is created; multiple pipelines cannot share a PCollection. In some respects, a PCollection functions like a collection class. However, a PCollection can differ in a few key ways:

PCollection由为其创建的特定管道对象所拥有；多个管道无法共享 PCollection。在某些方面，PCollection 函数类似于集合类。但是，PCollection在几个关键方面可能会有所不同：

#### 3.2.1. 元素类型

The elements of a PCollection may be of any type, but must all be of the same type. However, to support distributed processing, Beam needs to be able to encode each individual element as a byte string (so elements can be passed around to distributed workers). The Beam SDKs provide a data encoding mechanism that includes built-in encoding for commonly-used types as well as support for specifying custom encodings as needed.

PCollection的元素可以是任何类型, 但必须都是同一类型。然而, 为了支持分布式处理, Beam需要能够将每个单独的元素编码为一个字节字符串。这样，元素就可以传递给分布式的工作员(Worker)。Beam SDK提供了一种数据编码机制, 包括常用类型的内置编码, 以及根据需要指定自定义编码的支持。

#### 3.2.2. 不可变性（mmutable）

A PCollection is immutable. Once created, you cannot add, remove, or change individual elements. A Beam Transform might process each element of a PCollection and generate new pipeline data (as a new PCollection), but it does not consume or modify the original input collection.

PCollection 是不可变的。一旦创建, 就不能添加、移除或更改单个元素。Bean转换可能处理 PCollection 的每个元素并生成新的管道数据 (作为新的 PCollection), 但不会消耗或修改原始输入集合。

#### 3.2.3. 随机访问（Random access）

A PCollection does not support random access to individual elements. Instead, Beam Transforms consider every element in a PCollection individually.

PCollection 不支持对单个元素的随机访问。相反, Beam变换会单独考虑 PCollection 中的每个元素。

#### 3.2.4. 大小与边界（Size and boundedness）

A PCollection is a large, immutable “bag” of elements. There is no upper limit on how many elements a PCollection can contain; any given PCollection might fit in memory on a single machine, or it might represent a very large distributed data set backed by a persistent data store.

PCollection是一个大的, 不可变的元素"包"。PCollection可以包含多少元素没有上限；任何给定的PCollection 都可能适合于单个计算机上的内存, 也可能表示由持久数据存储支持的非常大的分布式数据集。

A PCollection can be either **bounded** or **unbounded** in size. A **bounded** PCollection represents a data set of a known, fixed size, while an **unbounded** PCollection represents a data set of unlimited size. Whether a PCollection is bounded or unbounded depends on the source of the data set that it represents. Reading from a batch data source, such as a file or a database, creates a bounded PCollection. Reading from a streaming or continously-updating data source, such as Pub/Sub or Kafka, creates an unboundedPCollection (unless you explicitly tell it not to).

PCollection可以是有界（Bounded）的或无界（Unbounded）的大小。有界 PCollection 表示已知的固定大小的数据集, 而无界PCollection表示无限大小的数据集。PCollection 是有界的还是无界的，取决于它所代表的数据集的来源。从批处理数据源，如文件或数据库中读取会创建有界PCollection。从流式或不断更新的数据源，如Pub/Sub 或Kafka中读取数据会创建一个无界的PCollection，除非你明确地告诉它不要这样做。

The bounded (or unbounded) nature of your PCollection affects how Beam processes your data. A bounded PCollection can be processed using a batch job, which might read the entire data set once, and perform processing in a job of finite length. An unbounded PCollection must be processed using a streaming job that runs continuously, as the entire collection can never be available for processing at any one time.

PCollection 的有界 (或无界) 性质影响Beam处理数据的方式。一个有界的PCollection 可以使用批处理作业，它可能一次性读取整个数据集, 并在有限长度的作业中执行处理。必须使用连续运行的流式作业来处理无界PCollection, 因为任何时候都不可能有效地处理整个集合。

Beam uses [windowing](https://beam.apache.org/documentation/programming-guide/#windowing) to divide a continuously updating unbounded PCollection into logical windows of finite size. These logical windows are determined by some characteristic associated with a data element, such as a **timestamp**. Aggregation transforms (such as GroupByKey and Combine) work on a per-window basis — as the data set is generated, they process each PCollection as a succession of these finite windows.

Beam使用窗口化将不断更新的无界PCollection划分为有限大小的逻辑窗口。这些逻辑窗口由与数据元素关联的某些特性决定, 如时间戳。聚合转换(Aggregation)如 GroupByKey 和Combine, 在每个窗口的基础上工作——随着数据集的生成, 它们将每个 PCollection 作为一些列有限窗口进行处理。

#### 3.2.5. 元素时间戳(Element timestamps)

Each element in a PCollection has an associated intrinsic **timestamp**. The timestamp for each element is initially assigned by the [Source](https://beam.apache.org/documentation/programming-guide/#pipeline-io) that creates the PCollection. Sources that create an unbounded PCollection often assign each new element a timestamp that corresponds to when the element was read or added.

PCollection 中的每个元素都有一个关联的内部时间戳。每个元素的时间戳最初由创建 PCollection的源分配。创建无界PCollection的源通常为每个新元素分配一个与读取或添加元素时对应的时间戳。

**Note**: Sources that create a bounded PCollection for a fixed data set also automatically assign timestamps, but the most common behavior is to assign every element the same timestamp (Long.MIN\_VALUE).

注意: 为固定数据集创建有界PCollection的源也会自动分配时间戳, 但最常见的行为是将每个元素分配相同的戳 (Long.MIN\_VALUE)。

Timestamps are useful for a PCollection that contains elements with an inherent notion of time. If your pipeline is reading a stream of events, like Tweets or other social media messages, each element might use the time the event was posted as the element timestamp.

时间戳对于包含具有固有概念的元素的PCollection很有用。如果管道正在读取事件流，如Tweets或其他社交媒体消息，则每个元素都可能使用事件发布时间作为它的时间戳。

You can manually assign timestamps to the elements of a PCollection if the source doesn’t do it for you. You’ll want to do this if the elements have an inherent timestamp, but the timestamp is somewhere in the structure of the element itself (such as a “time” field in a server log entry). Beam has [Transforms](https://beam.apache.org/documentation/programming-guide/#transforms) that take a PCollection as input and output an identical PCollection with timestamps attached; see [Adding Timestamps](https://beam.apache.org/documentation/programming-guide/#adding-timestamps-to-a-pcollections-elements) for more information about how to do so.

你可以手动为PCollection的元素分配时间戳，如果数据源没有为你分配的话。如果元素具有固有的时间戳，但这时间戳又位于元素本身结构中的某个位置，如服务器日志项中的 "时间" 字段，那你也需要手动为PCollection的元素分配时间戳。Beam有转换能将一个PCollection 作为输入，并输出一个与之相同的、附加有时间戳的 PCollection。

## 4. 转换（Transforms）

Transforms are the operations in your pipeline, and provide a generic processing framework. You provide processing logic in the form of a function object (colloquially referred to as “user code”), and your user code is applied to each element of an input PCollection (or more than one PCollection). Depending on the pipeline runner and back-end that you choose, many different workers across a cluster may execute instances of your user code in parallel. The user code running on each worker generates the output elements that are ultimately added to the final output PCollection that the transform produces.

转换是管道中的操作, 并提供了一个通用的处理框架。你可以函数对象 (也可以叫做"用户代码") 的形式提供处理逻辑, 并将用户代码应用于输入的一个或多个PCollection的每个元素。根据你选择的管道运行器和后台, 许多跨群集的不同工作员可以并行执行你提供的用户代码的实例。在每个工作员上运行的用户代码生成输出元素。这些输出元素会被添加到转换生成的最终输出PCollection中。

The Beam SDKs contain a number of different transforms that you can apply to your pipeline’s PCollections. These include general-purpose core transforms, such as [ParDo](https://beam.apache.org/documentation/programming-guide/#pardo) or [Combine](https://beam.apache.org/documentation/programming-guide/#combine). There are also pre-written [composite transforms](https://beam.apache.org/documentation/programming-guide/#composite-transforms) included in the SDKs, which combine one or more of the core transforms in a useful processing pattern, such as counting or combining elements in a collection. You can also define your own more complex composite transforms to fit your pipeline’s exact use case.

Beam SDK包含许多不同的转换，其中包括一般用途的核心转换，如arDo和Combine。你可以将这些转换应用于你所创建的管道的PCollections。SDK中还包含一些预先编写好的复合转换, 它们将一个或多个核心转换组合在一个有用的处理模式中, 例如在集合中计数或合并元素。您还可以定义自己更复杂的复合转换, 以适应管道的确切使用情况。

### 4.1. 应用转换（Applying transforms）

To invoke a transform, you must **apply** it to the input PCollection. Each transform in the Beam SDKs has a generic apply method (or pipe operator |). Invoking multiple Beam transforms is similar to method chaining, but with one slight difference: You apply the transform to the input PCollection, passing the transform itself as an argument, and the operation returns the output PCollection. This takes the general form:

若要调用转换, 必须将其应用于输入PCollection。Beam SDK中的每个变换都有一个通用的应用方法 (或管道运算符 |)。调用多个Beam转换类似于方法链, 但有一个细微的差别: 将转换应用于输入PCollection, 将转换本身作为参数传递, 并且该操作返回输出 PCollection。这采用一般形式:

**[**Output PCollection**]** **=** **[**Input PCollection**].**apply**([**Transform**])**

Because Beam uses a generic apply method for PCollection, you can both chain transforms sequentially and also apply transforms that contain other transforms nested within (called [composite transforms](https://beam.apache.org/documentation/programming-guide/#composite-transforms) in the Beam SDKs).

由于Beam对PCollection使用泛型应用方法, 因此你可以按顺序链接转换, 也可应用复杂转换，所谓的复杂转换就是嵌套有其他转换的转换。

How you apply your pipeline’s transforms determines the structure of your pipeline. The best way to think of your pipeline is as a directed acyclic graph, where the nodes are PCollections and the edges are transforms. For example, you can chain transforms to create a sequential pipeline, like this one:

应用管道转换的方式决定了管道的结构。最好的方法是将你的管道是看作是一个定向无环图, 其中的节点是PCollections和边是转换（Transforms）。例如, 可以链接一些转换来创建顺序管道, 如下所示:

**[**Final Output PCollection**]** **=** **[**Initial Input PCollection**]**

**.**apply**([**First Transform**])**

**.**apply**([**Second Transform**])**

**.**apply**([**Third Transform**])**

The resulting workflow graph of the above pipeline looks like this.

上述管道的生成工作流图类似于下图所示。

This linear pipeline starts with one input collection, sequentially applies
  three transforms, and ends with one output collection.

Figure: A linear pipeline with three sequential transforms.

图 1 具有三个顺序变换的线性管道。

However, note that a transform does not consume or otherwise alter the input collection–remember that a PCollection is immutable by definition. This means that you can apply multiple transforms to the same input PCollection to create a branching pipeline, like so:

但是, 请注意, 转换不消耗或以其他方式改变输入集合元素，因为PCollection定义为不可变的。这意味着你可以将多个转换应用于相同的输入PCollection 以创建分支管道, 如下所示:

**[**Output PCollection 1**]** **=** **[**Input PCollection**].**apply**([**Transform 1**])**

**[**Output PCollection 2**]** **=** **[**Input PCollection**].**apply**([**Transform 2**])**

The resulting workflow graph from the branching pipeline above looks like this.

从上面的分支管道生成的工作流图看起来像这样。

This pipeline applies two transforms to a single input collection. Each
  transform produces an output collection.

Figure: A branching pipeline. Two transforms are applied to a single PCollection of database table rows.

图 2 分支管道 (两个转换使用数据表行集组成的单一PCollection)

You can also build your own [composite transforms](https://beam.apache.org/documentation/programming-guide/#composite-transforms) that nest multiple sub-steps inside a single, larger transform. Composite transforms are particularly useful for building a reusable sequence of simple steps that get used in a lot of different places.

你还可以构建自己的复合转换, 将多个子步骤嵌套在一个更大的转换中。复合转换对于构建可重用的简单步骤序列非常有用, 在许多不同的地方都可以使用。

### 4.2. 核心Beam转换

Beam provides the following core transforms, each of which represents a different processing paradigm:

Beam提供了以下核心转换, 每个都代表不同的处理范式:

* ParDo
* GroupByKey
* CoGroupByKey
* Combine
* Flatten
* Partition

#### 4.2.1. ParDo

ParDo is a Beam transform for generic parallel processing. The ParDo processing paradigm is similar to the “Map” phase of a Map/Shuffle/Reduce-style algorithm: a ParDo transform considers each element in the input PCollection, performs some processing function (your user code) on that element, and emits zero, one, or multiple elements to an output PCollection.

ParDo是一种通用并行处理的Beam变换。ParDo处理范式类似于Map/Shuffle/Reduce样式算法的Map阶段: ParDo变换考虑输入PCollection 中的每个元素, 在该元素上执行一些处理函数 (用户代码), 并发出零、一个或多个元素到输出PCollection中。

ParDo is useful for a variety of common data processing operations, including:

ParDo对于各种常见的数据处理操作非常有用, 其中包括:

* **Filtering a data set.** You can use ParDo to consider each element in a PCollection and either output that element to a new collection, or discard it.
* 筛选数据集。可以使用ParDo 来考虑 PCollection 中的每个元素, 或者将该元素输出到新的集合中, 或者丢弃它。
* **Formatting or type-converting each element in a data set.** If your input PCollection contains elements that are of a different type or format than you want, you can use ParDo to perform a conversion on each element and output the result to a new PCollection.
* 格式化或类型转换数据集中的每个元素。如果输入PCollection 包含的元素与所需的类型或格式不同, 则可以使用ParDo 对每个元素执行转换, 并将结果输出到新的 PCollection。
* **Extracting parts of each element in a data set.** If you have a PCollection of records with multiple fields, for example, you can use a ParDo to parse out just the fields you want to consider into a new PCollection.
* 提取数据集中每个元素的部分。例如, 如果您有多个字段的记录集合PCollection, 则可以使用ParDo来只解析你需要的字段到新 PCollection 中。
* **Performing computations on each element in a data set.** You can use ParDo to perform simple or complex computations on every element, or certain elements, of a PCollection and output the results as a new PCollection.
* 对数据集中的每个元素执行计算。您可以使用ParDo 对 PCollection 的每个元素或某些元素执行简单或复杂的计算, 并将结果输出为新的 PCollection。

In such roles, ParDo is a common intermediate step in a pipeline. You might use it to extract certain fields from a set of raw input records, or convert raw input into a different format; you might also use ParDo to convert processed data into a format suitable for output, like database table rows or printable strings.

在这些角色中, ParDo是管道中常见的中间步骤。您可以使用它从一组原始输入记录中提取某些字段, 或者将原始输入转换为其他格式。您还可以使用ParDo将已处理的数据转换为适合于输出的格式, 如数据库表的行集或可打印字符串。

When you apply a ParDo transform, you’ll need to provide user code in the form of a DoFn object. DoFn is a Beam SDK class that defines a distributed processing function.

应用ParDo转换时, 需要以 DoFn 对象的形式提供用户代码。DoFn 是一个定义分布式处理函数的Beam SDK 类。

When you create a subclass of DoFn, note that your subclass should adhere to the [Requirements for writing user code for Beam transforms](https://beam.apache.org/documentation/programming-guide/#requirements-for-writing-user-code-for-beam-transforms).

在创建 DoFn 的子类时, 请注意, 子类应遵守编写用于Beam转换的用户代码的要求。

##### 4.2.1.1. 应用 ParDo

Like all Beam transforms, you apply ParDo by calling the apply method on the input PCollection and passing ParDo as an argument, as shown in the following example code:

与所有Beam转换一样, 通过在输入 PCollection 上调用 apply 方法并将ParDo作为参数传递来应用ParDo, 如下面的示例代码所示:

*// The input PCollection of Strings.*

PCollection**<**String**>** words **=** **...;**

*// The DoFn to perform on each element in the input PCollection.*

**static** **class** **ComputeWordLengthFn** **extends** DoFn**<**String**,** Integer**>** **{** **...** **}**

*// Apply a ParDo to the PCollection "words" to compute lengths for each word.*

PCollection**<**Integer**>** wordLengths **=** words**.**apply**(**

*// The DoFn to perform on each element, which*

ParDo**.**of**(new** ComputeWordLengthFn**()));**

*// we define above.# The input PCollection of Strings.*

words **=** **...**

In the example, our input PCollection contains String values. We apply a ParDo transform that specifies a function (ComputeWordLengthFn) to compute the length of each string, and outputs the result to a new PCollection of Integer values that stores the length of each word.

在该示例中, 我们的输入 PCollection 包含字符串值。我们应用一个ParDo变换来指定一个函数 (ComputeWordLengthFn) 来计算每个字符串的长度, 并将结果输出到存储每个单词长度的整数值的新 PCollection。

##### 4.2.1.2.创建DoFn

The DoFn object that you pass to ParDo contains the processing logic that gets applied to the elements in the input collection. When you use Beam, often the most important pieces of code you’ll write are these DoFns–they’re what define your pipeline’s exact data processing tasks.

传递给ParDo的 DoFn 对象包含将被应用于输入集合中的元素的处理逻辑。当你使用Beam时, 通常你会写的最重要的代码片断是这些 DoFn，它们定义了管道的确切数据处理任务。

**Note:** When you create your DoFn, be mindful of the [Requirements for writing user code for Beam transforms](https://beam.apache.org/documentation/programming-guide/#requirements-for-writing-user-code-for-beam-transforms) and ensure that your code follows them.

注意: 创建 DoFn 时, 请注意编写用于Beam转换的用户代码的要求, 并确保代码遵循它们。

A DoFn processes one element at a time from the input PCollection. When you create a subclass of DoFn, you’ll need to provide type parameters that match the types of the input and output elements. If your DoFn processes incoming String elements and produces Integer elements for the output collection (like our previous example, ComputeWordLengthFn), your class declaration would look like this:

DoFn从输入 PCollection一次处理一个元素。创建 DoFn 的子类时, 需要提供与输入和输出元素的类型相匹配的类型参数。如果 DoFn 处理传入的字符串元素并生成输出集合的整数元素 (如前面的示例 ComputeWordLengthFn), 则类声明如下所示:

**static** **class** **ComputeWordLengthFn** **extends** DoFn**<**String**,** Integer**>** **{** **...** **}**

Inside your DoFn subclass, you’ll write a method annotated with @ProcessElement where you provide the actual processing logic. You don’t need to manually extract the elements from the input collection; the Beam SDKs handle that for you. Your @ProcessElementmethod should accept an object of type ProcessContext. The ProcessContext object gives you access to an input element and a method for emitting an output element:

在DoFn 子类中, 你将编写一个用 @ProcessElement 注释的方法来提供实际的处理逻辑。你不需要从输入集合中手动提取元素，Beam SDK已经为你进行了这方面的处理。 @ProcessElement 标注的方法应接受类型为ProcessContext 的对象。ProcessContext 对象使你可以访问输入元素和用于发出输出元素的方法。

**static** **class** **ComputeWordLengthFn** **extends** DoFn**<**String**,** Integer**>** **{**

@ProcessElement

**public** **void** **processElement(**ProcessContext c**)** **{**

*// Get the input element from ProcessContext.*

String word **=** c**.**element**();**

*// Use ProcessContext.output to emit the output element.*

c**.**output**(**word**.**length**());**

**}**

**}**

A given DoFn instance generally gets invoked one or more times to process some arbitrary bundle of elements. However, Beam doesn’t guarantee an exact number of invocations; it may be invoked multiple times on a given worker node to account for failures and retries. As such, you can cache information across multiple calls to your processing method, but if you do so, make sure the implementation **does not depend on the number of invocations**.

给定的 DoFn 实例通常被调用一次或多次来处理一些任意的元素包。但是, Beam不能保证准确的调用次数;可以在给定的工作节点上多次调用它来处理失败和重试。因此, 你可以通过对处理方法的多个调用缓存信息, 但如果这样做, 请确保实现不依赖于调用的次数。

In your processing method, you’ll also need to meet some immutability requirements to ensure that Beam and the processing back-end can safely serialize and cache the values in your pipeline. Your method should meet the following requirements:

在你的处理方法中, 您还需要满足一些不变性要求, 以确保Beam和处理后端可以安全地序列化和缓存管道中的值。你的方法应满足以下要求:

* You should not in any way modify an element returned by ProcessContext.element() or ProcessContext.sideInput() (the incoming elements from the input collection).
* 不应以任何方式修改由 ProcessContext ().element() 或 ProcessContext () sideInput () 返回的元素 (输入集合中的传入元素)。
* Once you output a value using ProcessContext.output() or ProcessContext.sideOutput(), you should not modify that value in any way.
* 使用 ProcessContext ().output() 或 ProcessContext sideOutput () 输出值后, 不应以任何方式修改该值。

##### 4.2.1.3. 轻量级DoFn和其他抽象

If your function is relatively straightforward, you can simplify your use of ParDo by providing a lightweight DoFn in-line, as an anonymous inner class instance a lambda function.

如果你的函数相对简单, 则可以通过提供轻量级的DoFn，作为一个匿名内部类实例或lambda 函数来简化ParDo的使用。

Here’s the previous example, ParDo with ComputeLengthWordsFn, with the DoFn specified as an anonymous inner class instance a lambda function:

下面是前面的示例, 带有ComputeLengthWordsFn的ParDo, 该 DoFn 指定为匿名内部类:

*// The input PCollection.*

PCollection**<**String**>** words **=** **...;**

*// Apply a ParDo with an anonymous DoFn to the PCollection words.*

*// Save the result as the PCollection wordLengths.*

PCollection**<**Integer**>** wordLengths **=** words**.**apply**(**

"ComputeWordLengths"**,** *// the transform name*

*// a DoFn as an anonymous inner class instance*

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

@ProcessElement

**public** **void** **processElement(**ProcessContext c**)** **{**

c**.**output**(**c**.**element**().**length**());**

**}**

**}));**

*# The input PCollection of strings.*

words **=** **...**

If your ParDo performs a one-to-one mapping of input elements to output elements–that is, for each input element, it applies a function that produces exactly one output element, you can use the higher-level MapElementsMap transform. MapElements can accept an anonymous Java 8 lambda function for additional brevity. Here’s the previous example using MapElements Map:

如果ParDo执行输入元素到输出元素的一对一映射, 也就是，对于每个输入元素, 它应用一个只生成一个输出元素的函数, 可以使用更高级的 MapElementsMap 转换。MapElements 可以接受匿名的lambda 函数, 进而精简代码。下面是使用 MapElements 映射来实现的前面的示例:

*// The input PCollection.*

PCollection**<**String**>** words **=** **...;**

*// Apply a MapElements with an anonymous lambda function to the PCollection words.*

*// Save the result as the PCollection wordLengths.*

PCollection**<**Integer**>** wordLengths **=** words**.**apply**(**

MapElements**.**into**(**TypeDescriptors**.**integers**())**

**.**via**((**String word**)** **->** word**.**length**()));**

*# The input PCollection of string.*

words **=** **...**

**Note:** You can use Java 8 lambda functions with several other Beam transforms, including Filter, FlatMapElements, and Partition.

注意: 你可以使用lambda函数和其他几个Beam转换, 包括Filter, FlatMapElements和Partition。

##### 4.2.1.4. SingleOutput

public static class **ParDo.SingleOutput<InputT,OutputT>**

extends [PTransform](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/PTransform.html)<[PCollection](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/values/PCollection.html)<? extends InputT>,[PCollection](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/values/PCollection.html)<OutputT>>

A [PTransform](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/PTransform.html) that, when applied to a PCollection<InputT>, invokes a user-specified DoFn<InputT, OutputT> on all its elements, with all its outputs collected into an output PCollection<OutputT>.

A multi-output form of this transform can be created with [withOutputTags(org.apache.beam.sdk.values.TupleTag<OutputT>, org.apache.beam.sdk.values.TupleTagList)](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/ParDo.SingleOutput.html#withOutputTags-org.apache.beam.sdk.values.TupleTag-org.apache.beam.sdk.values.TupleTagList-).

##### 4.2.1.5. MultiOutput

#### 4.2.2. GroupByKey

GroupByKey is a Beam transform for processing collections of key/value pairs. It’s a parallel reduction operation, analogous to the Shuffle phase of a Map/Shuffle/Reduce-style algorithm. The input to GroupByKey is a collection of key/value pairs that represents a multimap, where the collection contains multiple pairs that have the same key, but different values. Given such a collection, you use GroupByKey to collect all of the values associated with each unique key.

GroupByKey 是用于处理键/值对集合的Beam变换。这是一个并行的化简操作, 类似于Map/Shuffle/Reduce样式算法的Shuffle阶段。GroupByKey的输入是表示多重映射的键/值对的集合, 其中集合包含具有相同键但不同值的多个键值对。给定这样的集合,可以使用GroupByKey收集与每个唯一键关联的所有值。

GroupByKey is a good way to aggregate data that has something in common. For example, if you have a collection that stores records of customer orders, you might want to group together all the orders from the same postal code (wherein the “key” of the key/value pair is the postal code field, and the “value” is the remainder of the record).

GroupByKey是一种聚合具有一些共同点数据的很好的方法。例如, 如果你有一个存储客户订单记录的集合, 你可能希望将同一邮政编码中的所有订单组合在一起 (其中键/值对的 "键" 是邮政编码字段, 而 "值" 是记录的其余部分)。

Let’s examine the mechanics of GroupByKey with a simple example case, where our data set consists of words from a text file and the line number on which they appear. We want to group together all the line numbers (values) that share the same word (key), letting us see all the places in the text where a particular word appears.

让我们用一个简单的例子来检查 GroupByKey 机制, 我们的数据集由文本文件中的单词和它们出现的行号组成。我们要将共享相同单词 (键) 的所有行号 (值) 组合在一起, 让我们看到文本中出现特定单词的所有位置。

Our input is a PCollection of key/value pairs where each word is a key, and the value is a line number in the file where the word appears. Here’s a list of the key/value pairs in the input collection:

我们的输入是键/值对的PCollection, 其中每个单词都是键, 并且该值是该单词出现的文件中的行号。以下是输入集合中的键/值对的列表:

cat, 1

dog, 5

and, 1

jump, 3

tree, 2

cat, 5

dog, 2

and, 2

cat, 9

and, 6

...

GroupByKey gathers up all the values with the same key and outputs a new pair consisting of the unique key and a collection of all of the values that were associated with that key in the input collection. If we apply GroupByKey to our input collection above, the output collection would look like this:

GroupByKey用相同的键收集所有值, 并输出一个新的对, 由唯一键和在输入集合中与该键关联的所有值的集合组成。如果我们将GroupByKey应用于上面的输入集合, 则输出集合将如下所述:

cat, [1,5,9]

dog, [5,2]

and, [1,2,6]

jump, [3]

tree, [2]

...

Thus, GroupByKey represents a transform from a multimap (multiple keys to individual values) to a uni-map (unique keys to collections of values).

因此, GroupByKey 表示从多重映射 (多个键到单个值) 到单向映射 (唯一键到值集合) 的转换。

##### 4.2.2.1 GroupByKey 和无界 PCollections

If you are using unbounded PCollections, you must use either [non-global windowing](https://beam.apache.org/documentation/programming-guide/#setting-your-pcollections-windowing-function) or an [aggregation trigger](https://beam.apache.org/documentation/programming-guide/#triggers) in order to perform a GroupByKey or [CoGroupByKey](https://beam.apache.org/documentation/programming-guide/#cogroupbykey). This is because a bounded GroupByKey or CoGroupByKey must wait for all the data with a certain key to be collected, but with unbounded collections, the data is unlimited. Windowing and/or triggers allow grouping to operate on logical, finite bundles of data within the unbounded data streams.

如果使用无界PCollection, 则必须使用非全局窗口或聚合触发器才能执行 GroupByKey 或CoGroupByKey。这是因为一个有界的GroupByKey 或 CoGroupByKey 必须等待所有的数据使用某个键来收集, 但是对于无界集合, 数据是无限的。窗口和/或触发器允许在无界数据流内分组进行逻辑有限的数据束操作。

If you do apply GroupByKey or CoGroupByKey to a group of unbounded PCollections without setting either a non-global windowing strategy, a trigger strategy, or both for each collection, Beam generates an IllegalStateException error at pipeline construction time.

如果你确实要将GroupByKey或CoGroupByKey 应用于一组无界PCollection, 而不设置既不设置非全局窗口策略，也不设置触发器策略, 则Beam在管道构造时生成 IllegalStateException错误。

When using GroupByKey or CoGroupByKey to group PCollections that have a [windowing strategy](https://beam.apache.org/documentation/programming-guide/#windowing) applied, all of the PCollections you want to group must use the same windowing strategy and window sizing. For example, all of the collections you are merging must use (hypothetically) identical 5-minute fixed windows, or 4-minute sliding windows starting every 30 seconds.

当使用 GroupByKey或CoGroupByKey对应用了窗口策略的PCollections组进行分组时, 要分组的所有PCollections都必须使用相同的窗口策略和窗口大小。例如, 所有要合并的集合必须使用 (假设) 相同的5分钟固定窗口, 或者每30秒启动4分钟滑动窗口。

If your pipeline attempts to use GroupByKey or CoGroupByKey to merge PCollections with incompatible windows, Beam generates an IllegalStateException error at pipeline construction time.

如果管道试图使用GroupByKey或CoGroupByKey将PCollections与不兼容的窗口合并, 则在管道构造时, Beam会产生 IllegalStateException 错误。

#### 4.2.3. CoGroupByKey

CoGroupByKey performs a relational join of two or more key/value PCollections that have the same key type. Consider using CoGroupByKey if you have multiple data sets that provide information about related things. For example, let’s say you have two different files with user data: one file has names and email addresses; the other file has names and phone numbers. You can join those two data sets, using the user name as a common key and the other data as the associated values. After the join, you have one data set that contains all of the information (email addresses and phone numbers) associated with each name.

CoGroupByKey 执行两个或多个具有相同键值类型的键/值 PCollections 的关系连接。（详细信息参见https://beam.apache.org/documentation/pipelines/design-your-pipeline/#multiple-sources）如果有多个数据集提供有关相关内容的信息, 请考虑使用 CoGroupByKey。例如, 假设有两个不同的用户数据文件: 一个文件有名称和电子邮件地址;另一个文件有名称和电话号码。你可以用户名作为公共键，其他数据作为关联值来连接这两个数据集。在连接之后, 你有一个包含与每个名称关联的所有信息 (电子邮件地址和电话号码)的数据集。

If you are using unbounded PCollections, you must use either [non-global windowing](https://beam.apache.org/documentation/programming-guide/#setting-your-pcollections-windowing-function) or an [aggregation trigger](https://beam.apache.org/documentation/programming-guide/#triggers) in order to perform a CoGroupByKey.

如果使用无界 PCollections, 则必须使用非全局窗口或聚合触发器才能执行CoGroupByKey。

In the Beam SDK for Java, CoGroupByKey accepts a tuple of keyed PCollections (PCollection<KV<K, V>>) as input. For type safety, the SDK requires you to pass each PCollection as part of a KeyedPCollectionTuple. You must declare a TupleTag for each input PCollection in the KeyedPCollectionTuple that you want to pass to CoGroupByKey. As output, CoGroupByKey returns a PCollection<KV<K, CoGbkResult>>, which groups values from all the input PCollections by their common keys. Each key (all of typeK) will have a different CoGbkResult, which is a map from TupleTag<T> to Iterable<T>. You can access a specific collection in an CoGbkResult object by using the TupleTag that you supplied with the initial collection.

The following conceptual examples use two input collections to show the mechanics of CoGroupByKey.

在 Java 的Beam SDK 中, CoGroupByKey 接受一个键控 PCollections (PCollection<KV<K, V>>) 的元组作为输入。为了类型安全, SDK要求你将每个 PCollection 作为 KeyedPCollectionTuple的一部分传递。必须为要传递给 CoGroupByKey 的 KeyedPCollectionTuple 中的每个输入PCollection声明一个TupleTag。作为输出，CoGroupByKey返回一个PCollection<KV<K, CoGbkResult>>, 它将所有输入 PCollections的值从它们的共同键分组。每个键 (所有类型为K) 都有一个不同的 CoGbkResult, 它是从 TupleTag <T> 到 Iterable <T> 的映射。通过使用与初始集合一起提供的 TupleTag, 可以访问 CoGbkResult 对象中的特定集合。下面的概念示例使用两个输入集合来显示 CoGroupByKey 的机制。

The first set of data has a TupleTag<String> called emailsTag and contains names and email addresses. The second set of data has a TupleTag<String> called phonesTag and contains names and phone numbers. The first set of data contains names and email addresses. The second set of data contains names and phone numbers.

第一组数据的 TupleTag <String> 称为 emailsTag, 包含名称和电子邮件地址。第二组数据的 TupleTag <String> 称为 phonesTag, 包含名称和电话号码。第一组数据包含名称和电子邮件地址。第二组数据包含名称和电话号码。

**final** List**<**KV**<**String**,** String**>>** emailsList **=** Arrays**.**asList**(**

KV**.**of**(**"amy"**,** "amy@example.com"**),**

KV**.**of**(**"carl"**,** "carl@example.com"**),**

KV**.**of**(**"julia"**,** "julia@example.com"**),**

KV**.**of**(**"carl"**,** "carl@email.com"**));**

**final** List**<**KV**<**String**,** String**>>** phonesList **=** Arrays**.**asList**(**

KV**.**of**(**"amy"**,** "111-222-3333"**),**

KV**.**of**(**"james"**,** "222-333-4444"**),**

KV**.**of**(**"amy"**,** "333-444-5555"**),**

KV**.**of**(**"carl"**,** "444-555-6666"**));**

PCollection**<**KV**<**String**,** String**>>** emails **=** p**.**apply**(**"CreateEmails"**,** Create**.**of**(**emailsList**));**

PCollection**<**KV**<**String**,** String**>>** phones **=** p**.**apply**(**"CreatePhones"**,** Create**.**of**(**phonesList**));**

After CoGroupByKey, the resulting data contains all data associated with each unique key from any of the input collections.

CoGroupByKey执行后, 生成的数据包含与任何输入集合中的每个唯一键关联的所有数据。

**final** TupleTag**<**String**>** emailsTag **=** **new** TupleTag**<>();**

**final** TupleTag**<**String**>** phonesTag **=** **new** TupleTag**<>();**

**final** List**<**KV**<**String**,** CoGbkResult**>>** expectedResults **=** Arrays**.**asList**(**

KV**.**of**(**"amy"**,** CoGbkResult

**.**of**(**emailsTag**,** Arrays**.**asList**(**"amy@example.com"**))**

**.**and**(**phonesTag**,** Arrays**.**asList**(**"111-222-3333"**,** "333-444-5555"**))),**

KV**.**of**(**"carl"**,** CoGbkResult

**.**of**(**emailsTag**,** Arrays**.**asList**(**"carl@email.com"**,** "carl@example.com"**))**

**.**and**(**phonesTag**,** Arrays**.**asList**(**"444-555-6666"**))),**

KV**.**of**(**"james"**,** CoGbkResult

**.**of**(**emailsTag**,** Arrays**.**asList**())**

**.**and**(**phonesTag**,** Arrays**.**asList**(**"222-333-4444"**))),**

KV**.**of**(**"julia"**,** CoGbkResult

**.**of**(**emailsTag**,** Arrays**.**asList**(**"julia@example.com"**))**

**.**and**(**phonesTag**,** Arrays**.**asList**())));**

The following code example joins the two PCollections with CoGroupByKey, followed by a ParDo to consume the result. Then, the code uses tags to look up and format data from each collection.

下面的代码示例将两个 PCollections 用CoGroupByKey 联接, 后跟ParDo以使用结果。然后, 代码使用标记来查找和格式化每个集合中的数据。

PCollection**<**KV**<**String**,** CoGbkResult**>>** results **=**

KeyedPCollectionTuple

**.**of**(**emailsTag**,** emails**)**

**.**and**(**phonesTag**,** phones**)**

**.**apply**(**CoGroupByKey**.**create**());**

PCollection**<**String**>** contactLines **=** results**.**apply**(**ParDo**.**of**(**

**new** DoFn**<**KV**<**String**,** CoGbkResult**>,** String**>()** **{**

@ProcessElement

**public** **void** **processElement(**ProcessContext c**)** **{**

KV**<**String**,** CoGbkResult**>** e **=** c**.**element**();**

String name **=** e**.**getKey**();**

Iterable**<**String**>** emailsIter **=** e**.**getValue**().**getAll**(**emailsTag**);**

Iterable**<**String**>** phonesIter **=** e**.**getValue**().**getAll**(**phonesTag**);**

String formattedResult **=** Snippets**.**formatCoGbkResults**(**name**,** emailsIter**,** phonesIter**);**

c**.**output**(**formattedResult**);**

**}**

**}**

**));**

The formatted data looks like this:

格式化后的数据看起来是这样的：

**final** List**<**String**>** formattedResults **=** Arrays**.**asList**(**

"amy; ['amy@example.com']; ['111-222-3333', '333-444-5555']"**,**

"carl; ['carl@email.com', 'carl@example.com']; ['444-555-6666']"**,**

"james; []; ['222-333-4444']"**,**

"julia; ['julia@example.com']; []"**);**

#### 4.2.4. Combine

[Combine](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/transforms/core.py) is a Beam transform for combining collections of elements or values in your data. Combine has variants that work on entire PCollections, and some that combine the values for each key in PCollections of key/value pairs.

[Combine](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/transforms/core.py) 是用于组合数据中元素或值集合的Beam转换。[Combine](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/transforms/core.py) 具有在整个 PCollection上工作的变量, 以及在键/值对 PCollection中组合每个键的值的变量。

When you apply a Combine transform, you must provide the function that contains the logic for combining the elements or values. The combining function should be commutative and associative, as the function is not necessarily invoked exactly once on all values with a given key. Because the input data (including the value collection) may be distributed across multiple workers, the combining function might be called multiple times to perform partial combining on subsets of the value collection. The Beam SDK also provides some pre-built combine functions for common numeric combination operations such as sum, min, and max.

应用组合转换时, 必须提供包含组合元素或值的逻辑的函数。组合函数应该是交换和关联的, 因为函数在所有值上都不一定用给定的键调用一次。由于输入数据 (包括值集合) 可以分布在多个工作员中, 因此可以多次调用组合函数来执行部分合并值集合的子集。Beam SDK 还为普通数字组合运算 (如 sum、min 和 max) 提供了一些预构建的组合函数。

Simple combine operations, such as sums, can usually be implemented as a simple function. More complex combination operations might require you to create a subclass of CombineFn that has an accumulation type distinct from the input/output type.

简单的组合运算 (如求和) 通常可以作为一个简单的函数来实现。更复杂的组合操作可能要求你创建具有与输入/输出类型不同的累积类型的 CombineFn 子类。

##### 4.2.4.1. 使用简单函数的简单组合

The following example code shows a simple combine function.

下面的例子显示了一个简单的组合函数。

*// Sum a collection of Integer values. The function SumInts implements the interface SerializableFunction.*

**public** **static** **class** **SumInts** **implements** SerializableFunction**<**Iterable**<**Integer**>,** Integer**>** **{**

@Override

**public** Integer **apply(**Iterable**<**Integer**>** input**)** **{**

**int** sum **=** 0**;**

**for** **(int** item **:** input**)** **{**

sum **+=** item**;**

**}**

**return** sum**;**

**}**

**}**

##### 4.2.4.2. 使用CombineFn函数的高级组合

For more complex combine functions, you can define a subclass of CombineFn. You should use CombineFn if the combine function requires a more sophisticated accumulator, must perform additional pre- or post-processing, might change the output type, or takes the key into account.

对于更复杂的组合函数, 可以定义CombineFn的子类。如果组合函数需要更复杂的累加器, 必须执行额外的预处理或后处理, 可能会更改输出类型或者涉及键值, 则应使用 CombineFn。

A general combining operation consists of four operations. When you create a subclass of CombineFn, you must provide four operations by overriding the corresponding methods:

一般组合操作由四个操作组成。在创建 CombineFn 的子类时, 必须通过重写相应的方法来提供四个操作:

1. **Create Accumulator** creates a new “local” accumulator. In the example case, taking a mean average, a local accumulator tracks the running sum of values (the numerator value for our final average division) and the number of values summed so far (the denominator value). It may be called any number of times in a distributed fashion. 创建累加器：创建一个新的 "本地" 累加器。在示例中, 采用中值平均值, 本地累加器跟踪值的运行总和 (最后平均除法的分子值) 和迄今汇总的值 (分母值) 的个数。它可以被分布式地调用任何次数。
2. **Add Input** adds an input element to an accumulator, returning the accumulator value. In our example, it would update the sum and increment the count. It may also be invoked in parallel. 添加输入：将输入元素添加到累加器中, 返回累加器值。在我们的示例中, 它将更新求和并递增计数。也可以并行调用。
3. **Merge Accumulators** merges several accumulators into a single accumulator; this is how data in multiple accumulators is combined before the final calculation. In the case of the mean average computation, the accumulators representing each portion of the division are merged together. It may be called again on its outputs any number of times. 合并累加器：将几个累加器合并为一个累加器;这就是在最终计算之前如何将多个累加器中的数据合并。在中值平均值计算的情况下, 代表除法各部分的累加器合并在一起。它可能会输出上再次调用它任何次数。
4. **Extract Output** performs the final computation. In the case of computing a mean average, this means dividing the combined sum of all the values by the number of values summed. It is called once on the final, merged accumulator. 提取输出：执行最终计算。在计算中值平均值的情况下, 这意味着将所有值的总和除以所求和的值的个数。它在会最后合并的累加器上被调用一次,。

The following example code shows how to define a CombineFn that computes a mean average: 下面的代码示例演示如何定义计算中值平均值的 CombineFn:

**public** **class** **AverageFn** **extends** CombineFn**<**Integer**,** AverageFn**.**Accum**,** Double**>** **{**

**public** **static** **class** **Accum** **{**

**int** sum **=** 0**;**

**int** count **=** 0**;**

**}**

@Override

**public** Accum **createAccumulator()** **{** **return** **new** Accum**();** **}**

@Override

**public** Accum **addInput(**Accum accum**,** Integer input**)** **{**

accum**.**sum **+=** input**;**

accum**.**count**++;**

**return** accum**;**

**}**

@Override

**public** Accum **mergeAccumulators(**Iterable**<**Accum**>** accums**)** **{**

Accum merged **=** createAccumulator**();**

**for** **(**Accum accum **:** accums**)** **{**

merged**.**sum **+=** accum**.**sum**;**

merged**.**count **+=** accum**.**count**;**

**}**

**return** merged**;**

**}**

@Override

**public** Double **extractOutput(**Accum accum**)** **{**

**return** **((double)** accum**.**sum**)** **/** accum**.**count**;**

**}**

**}**

If you are combining a PCollection of key-value pairs, [per-key combining](https://beam.apache.org/documentation/programming-guide/#combining-values-in-a-keyed-pcollection) is often enough. If you need the combining strategy to change based on the key (for example, MIN for some users and MAX for other users), you can define a KeyedCombineFn to access the key within the combining strategy.

如果要组合键值对的 PCollection, 则每个键组合通常就足够了。如果需要组合策略根据键进行更改 (例如, 某些用户的 MIN 和其他用户的 MAX), 则可以定义一个 KeyedCombineFn 来访问组合策略中的键。

##### 4.2.4.3. 将PCollection组合成单值

Use the global combine to transform all of the elements in a given PCollection into a single value, represented in your pipeline as a new PCollection containing one element. The following example code shows how to apply the Beam provided sum combine function to produce a single sum value for a PCollection of integers.

使用全局组合将给定 PCollection 中的所有元素转换为单个值, 在管道中表示为包含一个元素的新PCollection。下面的代码示例演示如何应用Beam提供的 sum 组合函数，该函数可以将整数 PCollection 转换成一个sum值。（注：网站上原本的实例程序运行不了，下面的例子做作了修改）

PCollection**<**Integer**>** pc **=** **...;**

//PCollection<Integer> sum = pc.apply(

// Combine.globally(new Sum.SumIntegerFn()));

PCollection<Integer> sum = pc.apply(Sum.integersGlobally());

##### 4.2.4.4. 组合和全局窗口

If your input PCollection uses the default global windowing, the default behavior is to return a PCollection containing one item. That item’s value comes from the accumulator in the combine function that you specified when applying Combine. For example, the Beam provided sum combine function returns a zero value (the sum of an empty input), while the max combine function returns a maximal or infinite value.

如果输入 PCollection 使用默认的全局窗口, 则默认行为是返回包含一个项的PCollection。该项的值来自在应用组合时指定的组合函数中的累加器。例如, Beam提供的 sum 组合函数返回一个零值 (一个空输入的总和), 而最大组合函数返回最大值或无穷数值。

To have Combine instead return an empty PCollection if the input is empty, specify .withoutDefaults when you apply your Combinetransform, as in the following code example:

如果输入为空, 则将其合并返回空 PCollection, 请在应用 Combine转换时指定. withoutDefaults, 如下面的代码示例所示:

PCollection**<**Integer**>** pc **=** **...;**

//PCollection**<**Integer**>** sum **=** pc**.**apply**(**

// Combine**.**globally**(new** Sum**.**SumIntegerFn**()).**withoutDefaults**());**

PCollection<Integer> sum = pc.apply(

Sum.integersGlobally()**.**withoutDefaults**()**);

##### 4.2.4.5. 组合与非全局窗口

If your PCollection uses any non-global windowing function, Beam does not provide the default behavior. You must specify one of the following options when applying Combine:

如果你的 PCollection 使用任何非全局窗口函数, 则Beam不提供默认行为。在应用组合时, 必须指定下列选项之一:

·

·

* Specify .withoutDefaults, where windows that are empty in the input PCollection will likewise be empty in the output collection. 指定. withoutDefaults, 在输入 PCollection 中为空的窗口同样会在输出集合中为空。
* Specify .asSingletonView, in which the output is immediately converted to a PCollectionView, which will provide a default value for each empty window when used as a side input. You’ll generally only need to use this option if the result of your pipeline’s Combine is to be used as a side input later in the pipeline. 指定. asSingletonView, 其中输出立即转换为 PCollectionView, 当用作侧输入（side input）时, 将为每个空窗口提供默认值。一般情况下, 如果管道组合的结果将用作管道后面的侧输入, 则通常只需要使用此选项。

##### 4.2.4.6. 在键值PCollection中合并值

After creating a keyed PCollection (for example, by using a GroupByKey transform), a common pattern is to combine the collection of values associated with each key into a single, merged value. Drawing on the previous example from GroupByKey, a key-grouped PCollection called groupedWords looks like this:

在创建了键控 PCollection (例如, 通过使用 GroupByKey 转换) 后, 通常的模式是将与每个键关联的值的集合组合到一个合并的值中。从 GroupByKey 的示例中来看, 称为 groupedWords 的键分组 PCollection 如下所示:

cat, [1,5,9]

dog, [5,2]

and, [1,2,6]

jump, [3]

tree, [2]

...

In the above PCollection, each element has a string key (for example, “cat”) and an iterable of integers for its value (in the first element, containing [1, 5, 9]). If our pipeline’s next processing step combines the values (rather than considering them individually), you can combine the iterable of integers to create a single, merged value to be paired with each key. This pattern of a GroupByKey followed by merging the collection of values is equivalent to Beam’s Combine PerKey transform. The combine function you supply to Combine PerKey must be an associative reduction function or a subclass of CombineFn.

在上面的 PCollection 中, 每个元素都有一个字符串键 (例如 "cat") 和一个整数 Iterable (在第一个元素中, 包含 [1、5、9])。如果管道的下一个处理步骤组合了这些值 (而不是单独考虑它们), 则可以组合整数Iterable 来创建单个合并值, 以便与每个键配对。GroupByKey后合并的值集合的这种模式,等价于Beam的Combine PerKey变换。您提供的组合函数PerKey 必须是一个关联Reduce函数或 CombineFn 的子类。

*// PCollection is grouped by key and the Double values associated with each key are combined into a Double.*

PCollection**<**KV**<**String**,** Double**>>** salesRecords **=** **...;**

PCollection**<**KV**<**String**,** Double**>>** totalSalesPerPerson **=**

salesRecords**.**apply**(**Combine**.<**String**,** Double**,** Double**>**perKey**(**

**new** Sum**.**SumDoubleFn**()));**

*// The combined value is of a different type than the original collection of values per key. PCollection has*

*// keys of type String and values of type Integer, and the combined value is a Double.*

PCollection**<**KV**<**String**,** Integer**>>** playerAccuracy **=** **...;**

PCollection**<**KV**<**String**,** Double**>>** avgAccuracyPerPlayer **=**

playerAccuracy**.**apply**(**Combine**.<**String**,** Integer**,** Double**>**perKey**(**

**new** **MeanInts())));**

#### 4.2.5. Flatten

[Flatten](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/transforms/core.py) and is a Beam transform for PCollection objects that store the same data type. Flatten merges multiple PCollection objects into a single logical PCollection.The following example shows how to apply a Flatten transform to merge multiple PCollection objects.

拼合, 是用于存储相同数据类型的PCollection对象集的Beam转换。它将多个 PCollection 对象合并为一个逻辑PCollection。下面的示例演示如何应用拼合转换来合并多个 PCollection 对象。

*// Flatten takes a PCollectionList of PCollection objects of a given type.*

*// Returns a single PCollection that contains all of the elements in the PCollection objects in that list.*

PCollection**<**String**>** pc1 **=** **...;**

PCollection**<**String**>** pc2 **=** **...;**

PCollection**<**String**>** pc3 **=** **...;**

PCollectionList**<**String**>** collections **=** PCollectionList**.**of**(**pc1**).**and**(**pc2**).**and**(**pc3**);**

PCollection**<**String**>** merged **=** collections**.**apply**(**Flatten**.<**String**>**pCollections**());**

##### 4.2.5.1.合并集合中的数据编码

By default, the coder for the output PCollection is the same as the coder for the first PCollection in the input PCollectionList. However, the input PCollection objects can each use different coders, as long as they all contain the same data type in your chosen language.

默认情况下, 输出 PCollection 的编码器与输入 PCollectionList 中第一个 PCollection 的编码器相同。但是, 输入 PCollection 对象可以使用不同的编码, 只要它们在您所选择的语言中都包含相同的数据类型。

##### 4.2.5.2.合并窗口集合

When using Flatten to merge PCollection objects that have a windowing strategy applied, all of the PCollection objects you want to merge must use a compatible windowing strategy and window sizing. For example, all the collections you’re merging must all use (hypothetically) identical 5-minute fixed windows or 4-minute sliding windows starting every 30 seconds. If your pipeline attempts to use Flatten to merge PCollection objects with incompatible windows, Beam generates an IllegalStateException error when your pipeline is constructed.

当使用拼合来合并应用了窗口策略的 PCollection 对象时, 要合并的所有 PCollection 对象都必须使用兼容的窗口化策略和窗口大小。例如, 所有要合并的集合都必须使用 (假设) 相同的5分钟固定窗口或4分钟滑动窗口, 每30秒开始一次。如果管道试图使用拼合将 PCollection 对象与不兼容的窗口合并, 则在构造管道时, Beam会产生一个 IllegalStateException 错误。

#### 4.2.6. Partition

[Partition](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/index.html?org/apache/beam/sdk/transforms/Partition.html) [Partition](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/transforms/core.py) is a Beam transform for PCollection objects that store the same data type. Partition splits a single PCollection into a fixed number of smaller collections.

分区是用于存储相同数据类型的 PCollection 对象的Beam转换。分区将单个 PCollection 拆分为一个固定数目的较小集合。

Partition divides the elements of a PCollection according to a partitioning function that you provide. The partitioning function contains the logic that determines how to split up the elements of the input PCollection into each resulting partition PCollection. The number of partitions must be determined at graph construction time. You can, for example, pass the number of partitions as a command-line option at runtime (which will then be used to build your pipeline graph), but you cannot determine the number of partitions in mid-pipeline (based on data calculated after your pipeline graph is constructed, for instance).

分区根据您提供的分区函数划分 PCollection 的元素。分区函数包含确定如何将输入 PCollection 的元素拆分为每个结果分区 PCollection 的逻辑。分区数必须在图构造时确定。例如, 可以在运行时将分区数作为命令行选项传递 (然后用于生成管道图), 但不能确定中间管道中的分区数 (例如，根据管道图所计算的数据构造)。

The following example divides a PCollection into percentile groups.

下面的示例将 PCollection 划分为百分比组。

*// Provide an int value with the desired number of result partitions, and a PartitionFn that represents the*

*// partitioning function. In this example, we define the PartitionFn in-line. Returns a PCollectionList*

*// containing each of the resulting partitions as individual PCollection objects.*

PCollection**<**Student**>** students **=** **...;**

*// Split students up into 10 partitions, by percentile:*

PCollectionList**<**Student**>** studentsByPercentile **=**

students**.**apply**(**Partition**.**of**(**10**,** **new** PartitionFn**<**Student**>()** **{**

**public** **int** **partitionFor(**Student student**,** **int** numPartitions**)** **{**

**return** student**.**getPercentile**()** *// 0..99*

**\*** numPartitions **/** 100**;**

**}}));**

*// You can extract each partition from the PCollectionList using the get method, as follows:*

PCollection**<**Student**>** fortiethPercentile **=** studentsByPercentile**.**get**(**4**);**

#### 4.2.7. Create

Create<T> takes a collection of elements of type T known when the pipeline is constructed and returns a PCollection<T> containing the elements.

Example of use:

Pipeline p = ...;

PCollection<Integer> pc = p.apply(Create.of(3, 4, 5).withCoder(BigEndianIntegerCoder.of()));

Map<String, Integer> map = ...;

PCollection<KV<String, Integer>> pt =

p.apply(Create.of(map)

.withCoder(KvCoder.of(StringUtf8Coder.of(),

BigEndianIntegerCoder.of())));

Create can automatically determine the Coder to use if all elements have the same run-time class, and a default coder is registered for that class. See [CoderRegistry](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/coders/CoderRegistry.html) for details on how defaults are determined.

If a coder can not be inferred, [Create.Values.withCoder(org.apache.beam.sdk.coders.Coder<T>)](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/Create.Values.html#withCoder-org.apache.beam.sdk.coders.Coder-) must be called explicitly to set the encoding of the resulting PCollection.

A good use for Create is when a PCollection needs to be created without dependencies on files or other external entities. This is especially useful during testing.

#### 4.2.8. Count

[PTransforms](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/PTransform.html) to count the elements in a [PCollection](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/values/PCollection.html).

[perElement()](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/Count.html#perElement--) can be used to count the number of occurrences of each distinct element in the PCollection, [perKey()](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/Count.html#perKey--) can be used to count the number of values per key, and [globally()](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/Count.html#globally--) can be used to count the total number of elements in a PCollection.

[combineFn()](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/Count.html#combineFn--) can also be used manually, in combination with state and with the [Combine](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/org/apache/beam/sdk/transforms/Combine.html) transform.

static <T> Combine.CombineFn<T,?,java.lang.Long>

combineFn()

Returns a Combine.CombineFn that counts the number of its inputs.

static <T> PTransform<PCollection<T>,PCollection<java.lang.Long>>

globally()

Returns a PTransform that counts the number of elements in its input PCollection.

static <T> PTransform<PCollection<T>,PCollection<KV<T,java.lang.Long>>>

perElement()

Returns a PTransform that counts the number of occurrences of each element in its input PCollection.

static <K,V> PTransform<PCollection<KV<K,V>>,PCollection<KV<K,java.lang.Long>>>

perKey()

Returns a PTransform that counts the number of elements associated with each key of its input PCollection.

#### 4.2.9. Filter

#### 4.2.10. FlatMapElements

#### 4.2.11. MapElements

##### SerializableFunction

##### SimpleFunction

#### 4.2.12. Max

#### 4.2.13. Mean

#### 4.2.14 Min

#### 4.2.15 Sum

#### 4.2.16 Keys

#### 4.2.17 Values

### 4.3. 为Beam转换编写用户代码的要求

When you build user code for a Beam transform, you should keep in mind the distributed nature of execution. For example, there might be many copies of your function running on a lot of different machines in parallel, and those copies function independently, without communicating or sharing state with any of the other copies. Depending on the Pipeline Runner and processing back-end you choose for your pipeline, each copy of your user code function may be retried or run multiple times. As such, you should be cautious about including things like state dependency in your user code.

在为Beam转换编写用户代码时, 应牢记执行的分布式性质。例如, 在许多不同的机器上并行运行的函数可能有许多副本, 并且这些拷贝独立工作, 不与任何其他副本通信或共享状态。根据管道运行程序和为管道选择的处理后端, 用户代码函数的每个副本都可以重试或多次运行。因此, 你应该谨慎地在用户代码中包括状态依赖性之类的事情。

In general, your user code must fulfill at least these requirements:

通常, 用户代码至少必须满足以下要求:

* Your function object must be **serializable**. 函数对象必须是可序列化的
* Your function object must be **thread-compatible**, and be aware that the Beam SDKs are not thread-safe. 函数对象必须与线程兼容, 并注意Beam SDK 不是线程安全的。

In addition, it’s recommended that you make your function object **idempotent**.

此外, 建议你将函数对象作为幂等。

**Note:** These requirements apply to subclasses of DoFn (a function object used with the [ParDo](https://beam.apache.org/documentation/programming-guide/#pardo)transform), CombineFn (a function object used with the [Combine](https://beam.apache.org/documentation/programming-guide/#combine) transform), and WindowFn (a function object used with the [Window](https://beam.apache.org/documentation/programming-guide/#windowing) transform).

注意: 这些要求适用于 DoFn 的子类 (与 ParDo转换一起使用的函数对象)、CombineFn (用于组合转换的函数对象) 和 WindowFn (用于窗口转换的函数对象)。

#### 4.3.1. Serializability可序列化

Any function object you provide to a transform must be **fully serializable**. This is because a copy of the function needs to be serialized and transmitted to a remote worker in your processing cluster. The base classes for user code, such as DoFn, CombineFn, and WindowFn, already implement Serializable; however, your subclass must not add any non-serializable members.

为转换提供的任何函数对象都必须是完全可序列化的。这是因为需要序列化函数的副本并将其传输到处理群集中的远程工作员（Worker）。用户代码的基类 (如 DoFn、CombineFn 和 WindowFn) 已经实现可序列化；但是， 子类不能添加任何不可序列化的成员。

Some other serializability factors you should keep in mind are:

你应该牢记的其他一些可序列化因素是:

* Transient fields in your function object are not transmitted to worker instances, because they are not automatically serialized. 函数对象中的瞬态字段不会被传输到工作实例, 因为它们不会自动序列化。
* Avoid loading a field with a large amount of data before serialization. 在序列化前避免加载大量数据的字段。
* Individual instances of your function object cannot share data. 函数对象的单个实例无法共享数据。
* Mutating a function object after it gets applied will have no effect. 在应用函数对象后对其进行变异将不起作用。
* Take care when declaring your function object inline by using an anonymous inner class instance. In a non-static context, your inner class instance will implicitly contain a pointer to the enclosing class and that class’ state. That enclosing class will also be serialized, and thus the same considerations that apply to the function object itself also apply to this outer class. 使用匿名内部类实例在内联声明函数对象时要小心。在非静态上下文中, 内部类实例将隐式包含指向封闭类和该类的状态的指针。该封闭类也将被序列化, 因此应用于函数对象本身的相同注意事项也适用于此外部类。

#### 4.3.2.线程兼容性

Your function object should be thread-compatible. Each instance of your function object is accessed by a single thread on a worker instance, unless you explicitly create your own threads. Note, however, that **the Beam SDKs are not thread-safe**. If you create your own threads in your user code, you must provide your own synchronization. Note that static members in your function object are not passed to worker instances and that multiple instances of your function may be accessed from different threads.

函数对象应与线程兼容。函数对象的每个实例都由工作实例上的单个线程访问, 除非显式创建自己的线程。但是, 请注意, Beam SDK 不是线程安全的。如果你在用户代码中创建自己的线程，则必须提供自己的同步。请注意, 函数对象中的静态成员不会传递给工作实例, 并且可以从不同的线程访问函数的多个实例。

#### 4.3.3.幂等性

It’s recommended that you make your function object idempotent–that is, that it can be repeated or retried as often as necessary without causing unintended side effects. The Beam model provides no guarantees as to the number of times your user code might be invoked or retried; as such, keeping your function object idempotent keeps your pipeline’s output deterministic, and your transforms’ behavior more predictable and easier to debug.

建议你使函数对象具有幂等性，也就是，它可以在必要时重复或重试, 而不会导致意外的副作用。Beam模型不保证你的用户代码可能被调用或重试的次数。因此，保持您的函数对象的幂等性会保持管道的输出确定性， 并且转换的行为更加可预测且更易于调试。

### 4.4. 侧面输入

In addition to the main input PCollection, you can provide additional inputs to a ParDo transform in the form of side inputs. A side input is an additional input that your DoFn can access each time it processes an element in the input PCollection. When you specify a side input, you create a view of some other data that can be read from within the ParDo transform’s DoFn while procesing each element.Side inputs are useful if your ParDo needs to inject additional data when processing each element in the input PCollection, but the additional data needs to be determined at runtime (and not hard-coded). Such values might be determined by the input data, or depend on a different branch of your pipeline.

除了主要的输入 PCollection, 你可以以侧面输入的方式提供额外的输入到ParDo转换中。侧面输入是你的DoFn每次处理输入PCollection中的元素时都可以访问的附加输入。当你指定侧输入时, 可以创建一些其他数据的视图, 可从ParDo转换的 DoFn 中读取，同时处理每个元素。

如果ParDo在处理输入 PCollection 中的每个元素时需要注入额外的数据, 则侧向输入变得很有用，但是附加数据需要在运行时确定 (而不是硬编码)。这些值可能由输入数据确定, 或者取决于管道的不同分支。

#### 4.4.1.传递侧面输入到ParDo

*// Pass side inputs to your ParDo transform by invoking .withSideInputs.*

*// Inside your DoFn, access the side input by using the method DoFn.ProcessContext.sideInput.*

*// The input PCollection to ParDo.*

PCollection**<**String**>** words **=** **...;**

*// A PCollection of word lengths that we'll combine into a single value.*

PCollection**<**Integer**>** wordLengths **=** **...;** *// Singleton PCollection*

*// Create a singleton PCollectionView from wordLengths using Combine.globally and View.asSingleton.*

**final** PCollectionView**<**Integer**>** maxWordLengthCutOffView **=**

wordLengths**.**apply**(**Combine**.**globally**(new** Max**.**MaxIntFn**()).**asSingletonView**());**

*// Apply a ParDo that takes maxWordLengthCutOffView as a side input.*

PCollection**<**String**>** wordsBelowCutOff **=**

words**.**apply**(**ParDo

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

**public** **void** **processElement(**ProcessContext c**)** **{**

String word **=** c**.**element**();**

*// In our DoFn, access the side input.*

**int** lengthCutOff **=** c**.**sideInput**(**maxWordLengthCutOffView**);**

**if** **(**word**.**length**()** **<=** lengthCutOff**)** **{**

c**.**output**(**word**);**

**}**

**}**

**}).**withSideInputs**(**maxWordLengthCutOffView**)**

**);**

#### 4.4.2.侧面输入与窗口

A windowed PCollection may be infinite and thus cannot be compressed into a single value (or single collection class). When you create a PCollectionView of a windowed PCollection, the PCollectionView represents a single entity per window (one singleton per window, one list per window, etc.).

窗口 PCollection 可能是无限的，因此不能压缩为单个值 (或单个集合类)。当你创建窗口 PCollection 的 PCollectionView 时，PCollectionView 代表每个窗口的单个实体 (每个窗口有一个单例，每个窗口有一个列表，等等)。

Beam uses the window(s) for the main input element to look up the appropriate window for the side input element. Beam projects the main input element’s window into the side input’s window set, and then uses the side input from the resulting window. If the main input and side inputs have identical windows, the projection provides the exact corresponding window. However, if the inputs have different windows, Beam uses the projection to choose the most appropriate side input window.

Beam使用主输入元素的窗口为侧面输入元素查找相应的窗口。Beam将主输入元素的窗口投射到侧面输入的窗口中, 然后使用结果窗口中的侧面输入。如果主输入和侧面输入有相同的窗口, 则投影提供精确对应的窗口。但是, 如果输入有不同的窗口, Beam使用投影选择最合适的侧输入窗口。

For example, if the main input is windowed using fixed-time windows of one minute, and the side input is windowed using fixed-time windows of one hour, Beam projects the main input window against the side input window set and selects the side input value from the appropriate hour-long side input window.

例如，如果主输入使用一分钟的固定时间窗口进行窗口化，而侧面输入用一个小时的固定时间窗口进行窗口化，则Beam将主输入窗口投射到侧面输入窗口，并从适当的小时长度的侧面输入窗口选择侧面输入值。

If the main input element exists in more than one window, then processElement gets called multiple times, once for each window. Each call to processElement projects the “current” window for the main input element, and thus might provide a different view of the side input each time.

如果主输入元素存在于多个窗口中, 则 processElement 会多次被调用, 每一个窗口都有一次。对 processElement 的每个调用都为主输入元素投影 "当前" 窗口, 因此每次都可能提供对侧输入的不同视图。

If the side input has multiple trigger firings, Beam uses the value from the latest trigger firing. This is particularly useful if you use a side input with a single global window and specify a trigger.

如果侧面输入有多个触发器触发, 则Beam使用最新触发器触发的值。如果在单个全局窗口中使用侧输入并指定触发器, 则这一点特别有用。

### 4.5.附加输出

While ParDo always produces a main output PCollection (as the return value from apply), you can also have your ParDo produce any number of additional output PCollections. If you choose to have multiple outputs, your ParDo returns all of the output PCollections (including the main output) bundled together.

虽然ParDo总是产生一个主输出 PCollection (作为返回值从应用), 你也可以让你的ParDo产生任何数量的额外输出 PCollections。如果选择多个输出, 则ParDo返回捆绑在一起的所有输出 PCollections (包括主输出)。

#### 4.5.1.多输出标记

*// To emit elements to multiple output PCollections, create a TupleTag object to identify each collection*

*// that your ParDo produces. For example, if your ParDo produces three output PCollections (the main output*

*// and two additional outputs), you must create three TupleTags. The following example code shows how to*

*// create TupleTags for a ParDo with three output PCollections.*

*// Input PCollection to our ParDo.*

PCollection**<**String**>** words **=** **...;**

*// The ParDo will filter words whose length is below a cutoff and add them to*

*// the main ouput PCollection<String>.*

*// If a word is above the cutoff, the ParDo will add the word length to an*

*// output PCollection<Integer>.*

*// If a word starts with the string "MARKER", the ParDo will add that word to an*

*// output PCollection<String>.*

**final** **int** wordLengthCutOff **=** 10**;**

*// Create three TupleTags, one for each output PCollection.*

*// Output that contains words below the length cutoff.*

**final** TupleTag**<**String**>** wordsBelowCutOffTag **=**

**new** TupleTag**<**String**>(){};**

*// Output that contains word lengths.*

**final** TupleTag**<**Integer**>** wordLengthsAboveCutOffTag **=**

**new** TupleTag**<**Integer**>(){};**

*// Output that contains "MARKER" words.*

**final** TupleTag**<**String**>** markedWordsTag **=**

**new** TupleTag**<**String**>(){};**

*// Passing Output Tags to ParDo:*

*// After you specify the TupleTags for each of your ParDo outputs, pass the tags to your ParDo by invoking*

*// .withOutputTags. You pass the tag for the main output first, and then the tags for any additional outputs*

*// in a TupleTagList. Building on our previous example, we pass the three TupleTags for our three output*

*// PCollections to our ParDo. Note that all of the outputs (including the main output PCollection) are*

*// bundled into the returned PCollectionTuple.*

PCollectionTuple results **=**

words**.**apply**(**ParDo

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

*// DoFn continues here.*

**...**

**})**

*// Specify the tag for the main output.*

**.**withOutputTags**(**wordsBelowCutOffTag**,**

*// Specify the tags for the two additional outputs as a TupleTagList.*

TupleTagList**.**of**(**wordLengthsAboveCutOffTag**)**

**.**and**(**markedWordsTag**)));**

#### 4.5.2.在DoFn中发出多个输出

*// Inside your ParDo's DoFn, you can emit an element to a specific output PCollection by passing in the*

*// appropriate TupleTag when you call ProcessContext.output.*

*// After your ParDo, extract the resulting output PCollections from the returned PCollectionTuple.*

*// Based on the previous example, this shows the DoFn emitting to the main output and two additional outputs.*

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

**public** **void** **processElement(**ProcessContext c**)** **{**

String word **=** c**.**element**();**

**if** **(**word**.**length**()** **<=** wordLengthCutOff**)** **{**

*// Emit short word to the main output.*

*// In this example, it is the output with tag wordsBelowCutOffTag.*

c**.**output**(**word**);**

**}** **else** **{**

*// Emit long word length to the output with tag wordLengthsAboveCutOffTag.*

c**.**output**(**wordLengthsAboveCutOffTag**,** word**.**length**());**

**}**

**if** **(**word**.**startsWith**(**"MARKER"**))** **{**

*// Emit word to the output with tag markedWordsTag.*

c**.**output**(**markedWordsTag**,** word**);**

**}**

**}}));**

*# Inside your ParDo's DoFn, you can emit an element to a specific output by wrapping the value and the output tag (str).*

*# using the pvalue.OutputValue wrapper class.*

*# Based on the previous example, this shows the DoFn emitting to the main output and two additional outputs.*

### 4.6.复合变换

Transforms can have a nested structure, where a complex transform performs multiple simpler transforms (such as more than one ParDo, Combine, GroupByKey, or even other composite transforms). These transforms are called composite transforms. Nesting multiple transforms inside a single composite transform can make your code more modular and easier to understand.

转换可以有一个嵌套结构, 其中复杂转换执行多个更简单的转换 (如多个ParDo、组合、GroupByKey 甚至其他复合转换)。这些转换称为复合转换。在单个复合转换内嵌套多个转换可以使代码更模块化, 更易于理解。

The Beam SDK comes packed with many useful composite transforms. See the API reference pages for a list of transforms.

Beam SDK 包装有许多有用的复合变换。有关转换列表, 请参见 API 参考页.

#### 4.6.1. 复合变换示例

The CountWords transform in the [WordCount example program](https://beam.apache.org/get-started/wordcount-example/) is an example of a composite transform. CountWords is a PTransformsubclass that consists of multiple nested transforms.

WordCount 示例程序中的 CountWords 转换是复合转换的一个示例。CountWords 是由多个嵌套转换组成的 PTransform子类。

In its expand method, the CountWords transform applies the following transform operations:

在其expand 方法中, CountWords 转换应用以下转换操作:

1. It applies a ParDo on the input PCollection of text lines, producing an output PCollection of individual words. 它对文本行的输入 PCollection 应用一个ParDo, 产生单个词的输出 PCollection。
2. It applies the Beam SDK library transform Count on the PCollection of words, producing a PCollection of key/value pairs. Each key represents a word in the text, and each value represents the number of times that word appeared in the original data. 它将Beam SDK 库转换计数应用于单词的 PCollection, 产生键/值对的 PCollection。每个键表示文本中的一个单词, 每个值表示 word 在原始数据中出现的次数。

Note that this is also an example of nested composite transforms, as Count is, by itself, a composite transform.

请注意, 这也是嵌套复合转换的一个示例, 因为 Count 本身就是一个复合转换。

Your composite transform’s parameters and return value must match the initial input type and final return type for the entire transform, even if the transform’s intermediate data changes type multiple times.

即使转换的中间数据多次更改, 复合转换的参数和返回值也必须与整个转换的初始输入类型和最终返回类型相匹配。

**public** **static** **class** **CountWords** **extends** PTransform**<**PCollection**<**String**>,**

PCollection**<**KV**<**String**,** Long**>>>** **{**

@Override

**public** PCollection**<**KV**<**String**,** Long**>>** **expand(**PCollection**<**String**>** lines**)** **{**

*// Convert lines of text into individual words.*

PCollection**<**String**>** words **=** lines**.**apply**(**

ParDo**.**of**(new** ExtractWordsFn**()));**

*// Count the number of times each word occurs.*

PCollection**<**KV**<**String**,** Long**>>** wordCounts **=**

words**.**apply**(**Count**.<**String**>**perElement**());**

**return** wordCounts**;**

**}**

**}**

#### 4.6.2.创建复合转换

To create your own composite transform, create a subclass of the PTransform class and override the expand method to specify the actual processing logic. You can then use this transform just as you would a built-in transform from the Beam SDK.

若要创建自己的复合转换, 请创建 PTransform 类的子类别, 并重写expand 方法以指定实际的处理逻辑。然后,可以像使用Beam SDK 中的内置转换一样使用此转换。

For the PTransform class type parameters, you pass the PCollection types that your transform takes as input, and produces as output. To take multiple PCollections as input, or produce multiple PCollections as output, use one of the multi-collection types for the relevant type parameter.

对于 PTransform 类类型参数, 可以传递转换所采用的 PCollection 类型作为输入, 并生成输出。要将多个 PCollections 作为输入, 或生成多个 PCollections 作为输出, 请使用相关类型参数的多集合类型之一。

The following code sample shows how to declare a PTransform that accepts a PCollection of Strings for input, and outputs a PCollection of Integers:

下面的代码示例演示如何声明接受输入的字符串 PCollection 的 PTransform, 并输出整数的 PCollection:

**static** **class** **ComputeWordLengths**

**extends** PTransform**<**PCollection**<**String**>,** PCollection**<**Integer**>>** **{**

**...**

**}**

Within your PTransform subclass, you’ll need to override the expand method. The expand method is where you add the processing logic for the PTransform. Your override of expand must accept the appropriate type of input PCollection as a parameter, and specify the output PCollection as the return value.

在 PTransform 子类中, 你需要重写expand 方法。expand 方法是为 PTransform 添加处理逻辑的位置。你的扩展重写必须接受适当类型的输入 PCollection 作为参数, 并指定输出 PCollection 作为返回值。

The following code sample shows how to override expand for the ComputeWordLengths class declared in the previous example:

下面的代码示例演示如何重写在上一个示例中声明的 ComputeWordLengths 类的展开:

**static** **class** **ComputeWordLengths**

**extends** PTransform**<**PCollection**<**String**>,** PCollection**<**Integer**>>** **{**

@Override

**public** PCollection**<**Integer**>** **expand(**PCollection**<**String**>)** **{**

**...**

*// transform logic goes here*

**...**

**}**

As long as you override the expand method in your PTransform subclass to accept the appropriate input PCollection(s) and return the corresponding output PCollection(s), you can include as many transforms as you want. These transforms can include core transforms, composite transforms, or the transforms included in the Beam SDK libraries. 只要重写 PTransform 子类中的expand 方法以接受适当的输入 PCollection 并返回相应的输出 PCollection, 就可以包含尽可能多的转换。这些转换可以包括核心转换、复合转换或Beam SDK 库中包含的转换。

**Note:** The expand method of a PTransform is not meant to be invoked directly by the user of a transform. Instead, you should call the apply method on the PCollection itself, with the transform as an argument. This allows transforms to be nested within the structure of your pipeline.

注意: PTransform 的展开方法并不意味着由转换的用户直接调用。相反, 您应该调用PCollection 本身的 apply 方法, 并将转换作为参数。这允许转换嵌套在管道结构中。

#### 4.6.3. PTransform风格指南

The [PTransform Style Guide](https://beam.apache.org/contribute/ptransform-style-guide/) contains additional information not included here, such as style guidelines, logging and testing guidance, and language-specific considerations. The guide is a useful starting point when you want to write new composite PTransforms.

PTransform 样式指南包含此处不包括的其他信息, 如样式指南、日志记录和测试指南以及特定于语言的注意事项。当你要编写新的复合 PTransforms 时, 指南是一个有用的起点。

## 5. 管道 I/O

When you create a pipeline, you often need to read data from some external source, such as a file or a database. Likewise, you may want your pipeline to output its result data to an external storage system. Beam provides read and write transforms for a [number of common data storage types](https://beam.apache.org/documentation/io/built-in/). If you want your pipeline to read from or write to a data storage format that isn’t supported by the built-in transforms, you can [implement your own read and write transforms](https://beam.apache.org/documentation/io/io-toc/).

创建管道时, 通常需要从某些外部源 (如文件或数据库) 中读取数据。同样, 您可能希望管道将其结果数据输出到外部存储系统。Beam为许多常见的数据存储类型提供读写转换。如果希望管道读取或写入不受内置转换支持的数据存储格式, 可以实现自己的读写转换。

### 5.1.读取输入数据

Read transforms read data from an external source and return a PCollection representation of the data for use by your pipeline. You can use a read transform at any point while constructing your pipeline to create a new PCollection, though it will be most common at the start of your pipeline.

读取转换从外部源读取数据, 并返回数据的 PCollection 表示形式以供管道使用。在构建管道时, 您可以在任何时候使用读取转换来创建新的 PCollection, 尽管在管道的开始时它将是最常见的。

PCollection**<**String**>** lines **=** p**.**apply**(**TextIO**.**read**().**from**(**"gs://some/inputData.txt"**));**

### 5.2.写出输出数据

Write transforms write the data in a PCollection to an external data source. You will most often use write transforms at the end of your pipeline to output your pipeline’s final results. However, you can use a write transform to output a PCollection’s data at any point in your pipeline.

写转换将 PCollection 中的数据写入外部数据源。您通常会在管道的末尾使用写转换来输出管道的最终结果。但是, 您可以使用写转换在管道的任何位置输出 PCollection 的数据。

output**.**apply**(**TextIO**.**write**().**to**(**"gs://some/outputData"**));**

### 5.3.基于文件的输入和输出数据

#### 5.3.1.从多个位置读取

Many read transforms support reading from multiple input files matching a glob operator you provide. Note that glob operators are filesystem-specific and obey filesystem-specific consistency models. The following TextIO example uses a glob operator (\*) to read all matching input files that have prefix “input-“ and the suffix “.csv” in the given location:

许多读取转换支持从多个输入文件中读取与您提供的一个元运算符匹配的内容。请注意, 该运算符是文件系统特定的, 并服从文件系统特定的一致性模型。下面的 TextIO 示例使用一个跨区域运算符 (\*) 读取所有匹配的输入文件, 它们的前缀为 "input", 并在给定位置添加后缀 ". csv":

p**.**apply**(**“ReadFromText”**,**

TextIO**.**read**().**from**(**"protocol://my\_bucket/path/to/input-\*.csv"**);**

To read data from disparate sources into a single PCollection, read each one independently and then use the [Flatten](https://beam.apache.org/documentation/programming-guide/#flatten) transform to create a single PCollection.

要将来自不同源的数据读取到单个 PCollection 中, 请独立地读取它们, 然后使用拼合转换创建单个 PCollection。

#### 5.3.2.写出到过个输出文件

For file-based output data, write transforms write to multiple output files by default. When you pass an output file name to a write transform, the file name is used as the prefix for all output files that the write transform produces. You can append a suffix to each output file by specifying a suffix.

对于基于文件的输出数据, 默认情况下, 写转换写入多个输出文件。将输出文件名传递给写入转换时, 文件名将用作写入转换生成的所有输出文件的前缀。通过指定后缀, 可以将后缀追加到每个输出文件。

The following write transform example writes multiple output files to a location. Each file has the prefix “numbers”, a numeric tag, and the suffix “.csv”.

下面的写转换示例将多个输出文件写入一个位置。每个文件都有前缀 "数字"、数字标记和后缀 ". csv"。

records**.**apply**(**"WriteToText"**,**

TextIO**.**write**().**to**(**"protocol://my\_bucket/path/to/numbers"**)**

**.**withSuffix**(**".csv"**));**

### 5.4. Beam提供的I/O转换

See the [Beam-provided I/O Transforms](https://beam.apache.org/documentation/io/built-in/) page for a list of the currently available I/O transforms.

有关当前可用 I/O 转换的列表, 请参见https://beam.apache.org/documentation/io/built-in/。

## 6.数据编码与类型安全

When Beam runners execute your pipeline, they often need to materialize the intermediate data in your PCollections, which requires converting elements to and from byte strings. The Beam SDKs use objects called Coders to describe how the elements of a given PCollection may be encoded and decoded.

当Beam运行器执行你的管道时, 他们通常需要在你的PCollections中具体化（实质化）中间数据, 这需要将元素转换成字节字符串。Beam SDK 使用称为编码的对象来描述给定 PCollection的元素如何被编码和解码。

Note that coders are unrelated to parsing or formatting data when interacting with external data sources or sinks. Such parsing or formatting should typically be done explicitly, using transforms such as ParDo or MapElements.

请注意, 在与外部数据源或接收器交互时, 编码器与解析或格式化数据无关。此类解析或格式通常应使用parDo或 MapElements 等转换显式进行。

In the Beam SDK for Java, the type Coder provides the methods required for encoding and decoding data. The SDK for Java provides a number of Coder subclasses that work with a variety of standard Java types, such as Integer, Long, Double, StringUtf8 and more. You can find all of the available Coder subclasses in the [Coder package](https://github.com/apache/beam/tree/master/sdks/java/core/src/main/java/org/apache/beam/sdk/coders).

在 Java 的Beam SDK 中, 类型编码器提供了编码和解码数据所需的方法。Java SDK 提供了许多使用各种标准 Java 类型 (如整数、长、双、StringUtf8 等) 的编码子类。你可以在编码器包中找到所有可用的编码子类。

Note that coders do not necessarily have a 1:1 relationship with types. For example, the Integer type can have multiple valid coders, and input and output data can use different Integer coders. A transform might have Integer-typed input data that uses BigEndianIntegerCoder, and Integer-typed output data that uses VarIntCoder.

请注意, 编码器不一定与类型有1:1 的关系。例如, 整数类型可以有多个有效的编码器, 输入和输出数据可以使用不同的整数编码。转换可能具有使用BigEndianIntegerCoder的整数类型输入数据, 以及使用VarIntCoder的整数类型输出数据。

### 6.1.指定编码器

The Beam SDKs require a coder for every PCollection in your pipeline. In most cases, the Beam SDK is able to automatically infer a Coder for a PCollection based on its element type or the transform that produces it, however, in some cases the pipeline author will need to specify a Coder explicitly, or develop a Coder for their custom type.

BeamSDK中，每个在你的管道中的PCollection都需要一个编码器。在大多数情况下, Beam SDK 可以根据其元素类型或产生它的变换自动推断PCollection的编码器, 但是, 在某些情况下, 管道设计者需要显式指定编码器, 或者为其自定义类型开发编码器。

You can explicitly set the coder for an existing PCollection by using the method PCollection.setCoder. Note that you cannot call setCoder on a PCollection that has been finalized (e.g. by calling .apply on it).

通过使用 PCollection.setCoder 方法, 可以显式设置现有 PCollection 的编码器。请注意, 你不能在已完成的 PCollection 上调用 setCoder (例如, 通过调用apply方法)。

You can get the coder for an existing PCollection by using the method getCoder. This method will fail with an IllegalStateException if a coder has not been set and cannot be inferred for the given PCollection.

通过使用 getCoder 方法, 可以获得现有 PCollection 的编码器。如果未设置编码器, 并且无法为给定的 PCollection 推断出编码器，则此方法将失败并抛出IllegalStateException。

Beam SDKs use a variety of mechanisms when attempting to automatically infer the Coder for a PCollection.

在时, BeamSDK会使用多种机制自动推断 PCollection 编码器。

Each pipeline object has a CoderRegistry. The CoderRegistry represents a mapping of Java types to the default coders that the pipeline should use for PCollections of each type.

每个管道对象都有一个 CoderRegistry。CoderRegistry 表示 Java 类型映射到管道应用于每种类型 PCollections 的默认编码器。

By default, the Beam SDK for Java automatically infers the Coder for the elements of a PCollection produced by a PTransform using the type parameter from the transform’s function object, such as DoFn. In the case of ParDo, for example, a DoFn<Integer, String>function object accepts an input element of type Integer and produces an output element of type String. In such a case, the SDK for Java will automatically infer the default Coder for the output PCollection<String> (in the default pipeline CoderRegistry, this isStringUtf8Coder).

默认情况下, Java 的Beam SDK 会使用转换函数对象 (如 DoFn) 中的类型参数，自动推断出由 PTransform 生成的 PCollection 的元素的编码器。例如, 在ParDo的情况下, DoFn <Integer, String> 函数对象接受类型整数的输入元素, 并生成类型字符串的输出元素。在这种情况下, Java SDK 将自动推断输出 PCollection <String> 的默认编码器 (在默认管道 CoderRegistry 中, 此 isStringUtf8Coder)。

NOTE: If you create your PCollection from in-memory data by using the Create transform, you cannot rely on coder inference and default coders. Create does not have access to any typing information for its arguments, and may not be able to infer a coder if the argument list contains a value whose exact run-time class doesn’t have a default coder registered.

注意: 如果你使用 "创建转换" 从内存中的数据创建 PCollection, 则不能依赖编码器推理和默认编码器。"创建" 不具有对其参数的任何键入信息的访问权限, 如果参数列表中包含的值的确切运行时类（Run-time class）没有注册的默认编码器, 则可能无法推断出编码者。

When using Create, the simplest way to ensure that you have the correct coder is by invoking withCoder when you apply the Createtransform.

使用 Create 时, 确保你拥有正确编码器的最简单方法是在应用 Create 时调用 withCoder。

### 6.2. 默认编码和 CoderRegistry

Each Pipeline object has a CoderRegistry object, which maps language types to the default coder the pipeline should use for those types. You can use the CoderRegistry yourself to look up the default coder for a given type, or to register a new default coder for a given type.

每个管道对象都有一个 CoderRegistry 对象, 它将语言类型映射到管道应用于这些类型的默认编码器。您可以使用 CoderRegistry查找给定类型的默认编码器, 或者为给定类型注册新的默认编码器。

CoderRegistry contains a default mapping of coders to standard Java types for any pipeline you create using the Beam SDK forJava. The following table shows the standard mapping:

CoderRegistry 包含一个默认的编码器映射到标准 JavaPython 类型的任何管道, 你创建使用Beam SDK forJavaPython。下表显示了标准映射:

| **Java Type** | **Default Coder** |
| --- | --- |
| Double | DoubleCoder |
| Instant | InstantCoder |
| Integer | VarIntCoder |
| Iterable | IterableCoder |
| KV | KvCoder |
| List | ListCoder |
| Map | MapCoder |
| Long | VarLongCoder |
| String | StringUtf8Coder |
| TableRow | TableRowJsonCoder |
| Void | VoidCoder |
| byte[ ] | ByteArrayCoder |
| TimestampedValue | TimestampedValueCoder |

#### 6.2.1.查找默认编码器

You can use the method CoderRegistry.getDefaultCoder to determine the default Coder for a Java type. You can access the CoderRegistry for a given pipeline by using the method Pipeline.getCoderRegistry. This allows you to determine (or set) the default Coder for a Java type on a per-pipeline basis: i.e. “for this pipeline, verify that Integer values are encoded using BigEndianIntegerCoder.”

你可以使用CoderRegistry.getDefaultCoder方法来确定 Java 类型的默认编码器。你可以通过Pipeline.getCoderRegistry方法访问给定管道的 CoderRegistry。这样, 你就可以在每个管道的基础上确定 (或设置) Java 类型的默认编码器: 即 "对于此管道, 请验证是否使用 BigEndianIntegerCoder 对整数值进行编码。

#### 6.2.2.设置类型的默认编码器

To set the default Coder for a Java type for a particular pipeline, you obtain and modify the pipeline’s CoderRegistry. You use the method Pipeline.getCoderRegistry to get the CoderRegistry object, and then use the methodCoderRegistry.registerCoder  to register a new Coder for the target type.

若要为特定管道设置 Java类型的默认编码器, 请获取并修改管道的 CoderRegistry。使用方法Pipeline.getCoderRegistry获取 CoderRegistry 对象, 然后使用方法CoderRegistry. registerCoder为目标类型注册新的编码。

The following example code demonstrates how to set a default Coder, in this case BigEndianIntegerCoder, for Integer values for a pipeline.

下面的代码示例演示如何为管道的 Integer 值设置默认编码器 (在本例中为 BigEndianIntegerCoder)。

PipelineOptions options **=** PipelineOptionsFactory**.**create**();**

Pipeline p **=** Pipeline**.**create**(**options**);**

CoderRegistry cr **=** p**.**getCoderRegistry**();**

cr**.**registerCoder**(**Integer**.**class**,** BigEndianIntegerCoder**.**class**);**

#### 6.2.3.使用默认编码器对自定义数据类型进行批注

If your pipeline program defines a custom data type, you can use the @DefaultCoder annotation to specify the coder to use with that type. For example, let’s say you have a custom data type for which you want to use SerializableCoder. You can use the @DefaultCoderannotation as follows:

如果管道程序定义了自定义数据类型, 则可以使用 @DefaultCoder 注释指定要与该类型一起使用的编码器。例如, 假设您有一个要使用 SerializableCoder 的自定义数据类型。可以使用 @DefaultCoderannotation, 如下所示:

@DefaultCoder**(**AvroCoder**.**class**)**

**public** **class** **MyCustomDataType** **{**

**...**

**}**

If you’ve created a custom coder to match your data type, and you want to use the @DefaultCoder annotation, your coder class must implement a static Coder.of(Class<T>) factory method.

如果已经创建了一个自定义编码器来匹配数据类型, 并且要使用 @DefaultCoder 注释, 则编码器类必须实现静态Coder.of (Class <T>) 工厂方法。

**public** **class** **MyCustomCoder** **implements** Coder **{**

**public** **static** Coder**<**T**>** **of(**Class**<**T**>** clazz**)** **{...}**

**...**

**}**

@DefaultCoder**(**MyCustomCoder**.**class**)**

**public** **class** **MyCustomDataType** **{**

**...**

**}**

## 7. 窗口Windowing

Windowing subdivides a PCollection according to the timestamps of its individual elements. Transforms that aggregate multiple elements, such as GroupByKey and Combine, work implicitly on a per-window basis — they process each PCollection as a succession of multiple, finite windows, though the entire collection itself may be of unbounded size.

窗口根据每个元素的时间戳对PCollection进行细分。聚合多个元素的转换，如GroupByKey和Composed，在每个窗口的基础上隐式地工作--它们将每个PCollection作为多个有限窗口的继承处理，尽管整个集合本身可能是无限大小的。

A related concept, called **triggers**, determines when to emit the results of aggregation as unbounded data arrives. You can use triggers to refine the windowing strategy for your PCollection. Triggers allow you to deal with late-arriving data or to provide early results. See the [triggers](https://beam.apache.org/documentation/programming-guide/#triggers) section for more information.

当无界数据到达时，触发器确定何时发出聚合结果。您可以使用触发器来改进PCollection的窗口策略。触发器允许您处理延迟到达的数据或提供早期结果。有关更多信息，请参见 [triggers](https://beam.apache.org/documentation/programming-guide/#triggers)部分。

### 7.1. 窗口的基础知识 （Windowing basics）

Some Beam transforms, such as GroupByKey and Combine, group multiple elements by a common key. Ordinarily, that grouping operation groups all of the elements that have the same key within the entire data set. With an unbounded data set, it is impossible to collect all of the elements, since new elements are constantly being added and may be infinitely many (e.g. streaming data). If you are working with unbounded PCollections, windowing is especially useful.

一些束转换(如GroupByKey和Combine)使用公共密钥对多个元素进行分组。通常，该分组操作对整个数据集中具有相同键的所有元素进行分组。使用无界数据集，不可能收集所有的元素，因为新元素不断地被添加，并且可能无限多(例如流数据)。如果您正在使用无界的PCollection，则窗口特别有用。

In the Beam model, any PCollection (including unbounded PCollections) can be subdivided into logical windows. Each element in a PCollection is assigned to one or more windows according to the PCollection’s windowing function, and each individual window contains a finite number of elements. Grouping transforms then consider each PCollection’s elements on a per-window basis. GroupByKey, for example, implicitly groups the elements of a PCollection by key and window.

在Beam模型中，任何PCollection(包括无界PCollection)都可以细分为逻辑窗口。根据PCollection的窗口功能，将PCollection中的每个元素分配给一个或多个窗口，并且每个单独的窗口包含有限数量的元素。分组转换然后在每个窗口的基础上考虑每个PCollection的元素。例如，GroupByKey隐式地按键和窗口对PCollection元素进行分组。

**Caution:** Beam’s default windowing behavior is to assign all elements of a PCollection to a single, global window and discard late data, even for unbounded *PCollection*s. Before you use a grouping transform such as GroupByKey on an unbounded PCollection, you must do at least one of the following:

注意：BEAM的默认窗口行为是将PCollection的所有元素分配给单个全局窗口，并丢弃后期数据，甚至对于无界的PCollection也是如此。在无界PCollection上使用分组转换(如GroupByKey)之前，必须至少执行以下操作之一：

* Set a non-global windowing function. See [Setting your PCollection’s windowing function](https://beam.apache.org/documentation/programming-guide/#setting-your-pcollections-windowing-function).
* Set a non-default [trigger](https://beam.apache.org/documentation/programming-guide/#triggers). This allows the global window to emit results under other conditions, since the default windowing behavior (waiting for all data to arrive) will never occur.
* 设置一个非全局窗口函数。请参见设置PCollection的窗口功能。
* 设置一个非默认触发器。这允许全局窗口在其他条件下发出结果，因为默认的窗口行为(等待所有数据到达)永远不会发生。

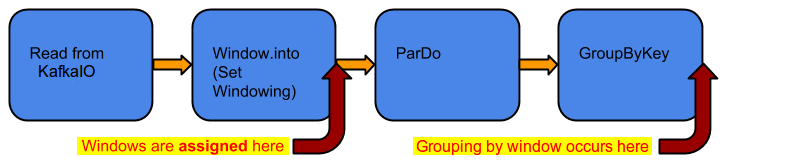
If you don’t set a non-global windowing function or a non-default trigger for your unbounded PCollection and subsequently use a grouping transform such as GroupByKey or Combine, your pipeline will generate an error upon construction and your job will fail.

如果您没有为您的无界PCollection设置非全局窗口函数或非默认触发器，并随后使用GroupByKey或Combine等分组转换，则您的管道将在构造时生成一个错误，您的作业将失败。

#### 7.1.1. 窗口的限制 (Windowing constraints)

After you set the windowing function for a PCollection, the elements’ windows are used the next time you apply a grouping transform to that PCollection. Window grouping occurs on an as-needed basis. If you set a windowing function using the Window transform, each element is assigned to a window, but the windows are not considered until GroupByKey or Combine aggregates across a window and key. This can have different effects on your pipeline. Consider the example pipeline in the figure below:

在你对一个pcollection使用窗函数后，元素的窗口是在你下次申请一个关于PCollection分组变换时使用。窗口分组发生在需要的基础上。如果你设定了一个窗函数使用窗口变换，每个元素被分配给一个窗口，但窗口直到groupbykey或Combine通过一个窗口和键聚集再考虑。这会对你的管道产生不同的影响。考虑下图中的示例管道：



**Figure:** Pipeline applying windowing

In the above pipeline, we create an unbounded PCollection by reading a set of key/value pairs using KafkaIO, and then apply a windowing function to that collection using the Window transform. We then apply a ParDo to the collection, and then later group the result of that ParDo using GroupByKey. The windowing function has no effect on the ParDo transform, because the windows are not actually used until they’re needed for the GroupByKey. Subsequent transforms, however, are applied to the result of the GroupByKey – data is grouped by both key and window.

在上面的管道中，我们通过使用KafkaIO读取一组键/值对来创建一个无界的PCollection，然后使用窗口转换对该集合应用一个窗口函数。然后，我们将Pardo应用于集合，然后使用GroupByKey对该Pardo的结果进行分组。窗口函数对Pardo转换没有任何影响，因为直到GroupByKey需要窗口时才实际使用这些窗口。然而，后续的转换应用于GroupByKey的结果--数据是按键和窗口进行分组的。

#### 7.1.2. 有界pcollections窗口（Windowing with bounded PCollections）

You can use windowing with fixed-size data sets in **bounded** PCollections. However, note that windowing considers only the implicit timestamps attached to each element of a PCollection, and data sources that create fixed data sets (such as TextIO) assign the same timestamp to every element. This means that all the elements are by default part of a single, global window.

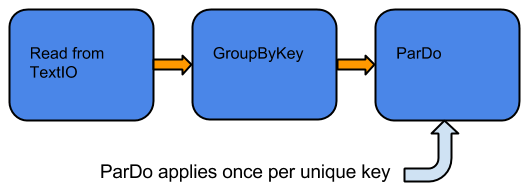
您可以在有界的PCollection中对固定大小的数据集使用窗口化。但是，请注意，窗口只考虑附加到PCollection的每个元素的隐式时间戳，以及创建固定数据集(例如TextIO)的数据源将相同的时间戳分配给每个元素。这意味着在默认情况下，所有元素都是单个全局窗口的一部分。

To use windowing with fixed data sets, you can assign your own timestamps to each element. To assign timestamps to elements, use a ParDo transform with a DoFn that outputs each element with a new timestamp (for example, the [WithTimestamps](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/index.html?org/apache/beam/sdk/transforms/WithTimestamps.html) transform in the Beam SDK for Java).

要对固定数据集使用窗口化，可以为每个元素分配自己的时间戳。若要将时间戳分配给元素，请使用带有DoFn的Pardo转换，该转换以新的时间戳输出每个元素(例如，Beam SDK for Java中的 [WithTimestamps](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/index.html?org/apache/beam/sdk/transforms/WithTimestamps.html)转换)。

To illustrate how windowing with a bounded PCollection can affect how your pipeline processes data, consider the following pipeline:

为了说明使用有界PCollection窗口如何影响管道处理数据的方式，请考虑以下管道：



**Figure:** GroupByKey and ParDo without windowing, on a bounded collection.

In the above pipeline, we create a bounded PCollection by reading a set of key/value pairs using TextIO. We then group the collection using GroupByKey, and apply a ParDo transform to the grouped PCollection. In this example, the GroupByKey creates a collection of unique keys, and then ParDo gets applied exactly once per key.

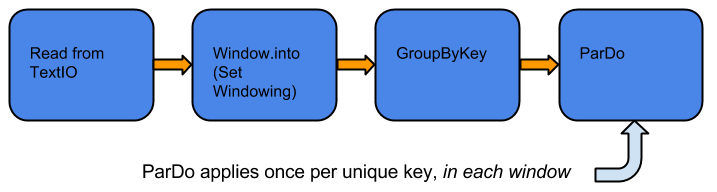
在上面的管道，我们通过阅读一组键/值对用TextIO创造界PCollection。然后我们组采用groupbykey收集，并应用ParDo变换分组pcollection。在这个例子中，groupbykey创建一个独特的收集键，然后ParDo就完全适用于每一次的键。

Note that even if you don’t set a windowing function, there is still a window – all elements in your PCollection are assigned to a single global window.

请注意，即使你不设置窗口函数，这还有一个窗口–所有元素在你的pcollection被分配到一个单一的全局窗口

Now, consider the same pipeline, but using a windowing function:

现在，考虑相同的管道，但使用窗口函数：



**Figure:** GroupByKey and ParDo with windowing, on a bounded collection.

图：GroupByKey和Pardo具有窗口，在一个有限的集合上。

As before, the pipeline creates a bounded PCollection of key/value pairs. We then set a [windowing function](https://beam.apache.org/documentation/programming-guide/#setting-your-pcollections-windowing-function) for that PCollection. The GroupByKey transform groups the elements of the PCollection by both key and window, based on the windowing function. The subsequent ParDo transform gets applied multiple times per key, once for each window.

和前面一样，管道创建了一个有界的键/值对的PCollection。然后，我们为该PCollection设置一个窗口函数。GroupByKey变换基于窗口函数，通过键和窗口对PCollection元素进行分组。随后的ParDo转换获得每个窗口一次的每键多次应用。

### 7.2. 设置窗口的功能（Provided windowing functions）

You can define different kinds of windows to divide the elements of your PCollection. Beam provides several windowing functions, including:

* Fixed Time Windows
* Sliding Time Windows
* Per-Session Windows
* Single Global Window
* Calendar-based Windows (not supported by the Beam SDK for Python)

您可以定义不同类型的窗口来划分PCollection的元素。BEAM提供了几种窗口功能，包括：

·固定时间Windows

·滑动时间Windows

·每次会话Windows

·单个全局窗口

·Calendar-基于Windows(不受BeamSDK for Python支持)

You can also define your own WindowFn if you have a more complex need.

如果您有更复杂的需要，也可以定义自己的WindowFn。

Note that each element can logically belong to more than one window, depending on the windowing function you use. Sliding time windowing, for example, creates overlapping windows wherein a single element can be assigned to multiple windows.

注意，每个元素在逻辑上可以属于多个窗口，这取决于您使用的窗口函数。例如，滑动时间窗口创建重叠窗口，其中单个元素可以分配给多个窗

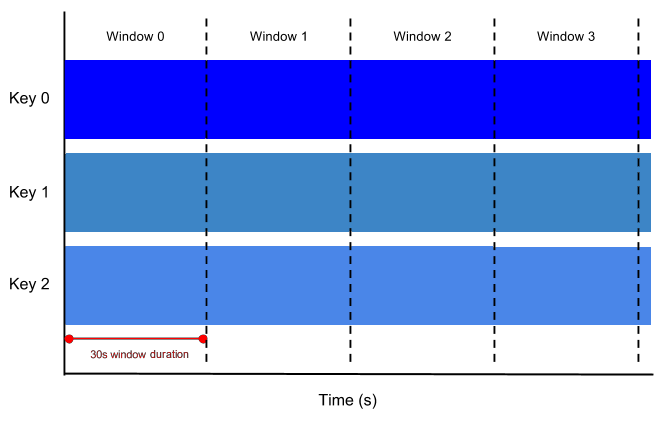
#### . 固定时间窗 （ Fixed time windows）

The simplest form of windowing is using **fixed time windows**: given a timestamped PCollection which might be continuously updating, each window might capture (for example) all elements with timestamps that fall into a five minute interval.

最简单的窗口形式是使用固定时间窗口：给定一个可能正在不断更新的时间戳PCollection，每个窗口可能捕获(例如)所有具有时间戳的元素，这些元素间隔为5分钟。

A fixed time window represents a consistent duration, non overlapping time interval in the data stream. Consider windows with a five-minute duration: all of the elements in your unbounded PCollection with timestamp values from 0:00:00 up to (but not including) 0:05:00 belong to the first window, elements with timestamp values from 0:05:00 up to (but not including) 0:10:00 belong to the second window, and so on.

固定时间窗口表示数据流中持续时间一致、不重叠的时间间隔。考虑持续时间为5分钟的窗口：您的无限PCollection中的所有元素的时间戳值从0：00：00到(但不包括)0：05：00属于第一个窗口，时间戳值从0：05：00到(但不包括)0：10：00的元素属于第二个窗口，依此类推。



**Figure:** Fixed time windows, 30s in duration.

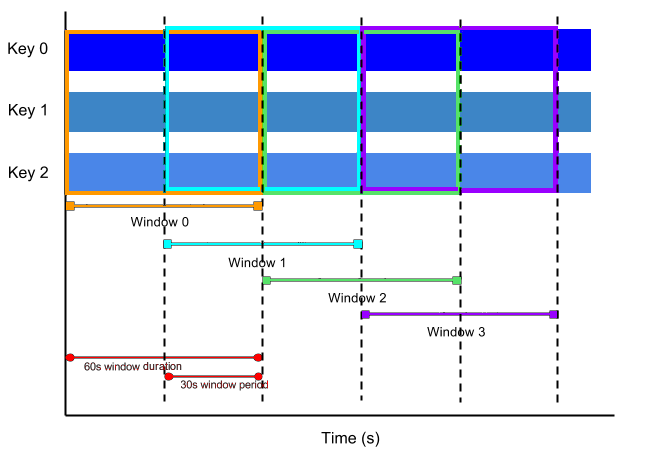
#### 7.2.2. 滑动时间窗（Sliding time windows）

A **sliding time window** also represents time intervals in the data stream; however, sliding time windows can overlap. For example, each window might capture five minutes worth of data, but a new window starts every ten seconds. The frequency with which sliding windows begin is called the period. Therefore, our example would have a window duration of five minutes and a period of ten seconds.

滑动时间窗口也表示数据流中的时间间隔；但是，滑动时间窗口可以重叠。例如，每个窗口可能捕获5分钟的数据，但一个新窗口每10秒启动一次。滑动窗口开始的频率称为周期。因此，我们的示例将有一个持续时间为5分钟和10秒钟的窗口。

Because multiple windows overlap, most elements in a data set will belong to more than one window. This kind of windowing is useful for taking running averages of data; using sliding time windows, you can compute a running average of the past five minutes’ worth of data, updated every ten seconds, in our example.

由于多个窗口重叠，数据集中的大多数元素将属于多个窗口。这种窗口对于获取数据的运行平均值很有用；在我们的示例中，使用滑动时间窗口，您可以计算过去五分钟数据的运行平均值，每十秒钟更新一次。

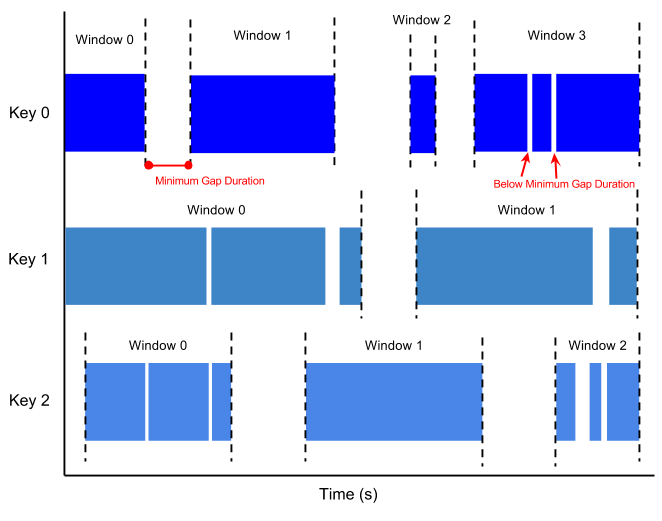


**Figure:** Sliding time windows, with 1 minute window duration and 30s window period.

#### 7.2.3. 会话窗口 （Session windows）

A **session window** function defines windows that contain elements that are within a certain gap duration of another element. Session windowing applies on a per-key basis and is useful for data that is irregularly distributed with respect to time. For example, a data stream representing user mouse activity may have long periods of idle time interspersed with high concentrations of clicks. If data arrives after the minimum specified gap duration time, this initiates the start of a new window.

会话窗口函数定义包含在另一个元素的特定间隔持续时间内的元素的窗口。会话窗口在按键的基础上应用，对于随时间不定期分布的数据非常有用。例如，表示用户鼠标活动的数据流可能具有长时间的空闲时间，其间有较高的点击浓度。如果数据在最小指定的间隙持续时间之后到达，这将启动一个新窗口的开始。



**Figure:** Session windows, with a minimum gap duration. Note how each data key has different windows, according to its data distribution.

会话窗口，具有最小的间隙持续时间。注意每个数据键有不同的窗口，根据它的数据分布。.

#### 7.2.4. 单一全局窗口 （The single global window）

By default, all data in a PCollection is assigned to the single global window, and late data is discarded. If your data set is of a fixed size, you can use the global window default for your PCollection.

默认情况下，PCollection中的所有数据都分配给单个全局窗口，而延迟数据将被丢弃。如果数据集是固定大小的，则可以为PCollection使用全局窗口默认值。

You can use the single global window if you are working with an unbounded data set (e.g. from a streaming data source) but use caution when applying aggregating transforms such as GroupByKey and Combine. The single global window with a default trigger generally requires the entire data set to be available before processing, which is not possible with continuously updating data. To perform aggregations on an unbounded PCollection that uses global windowing, you should specify a non-default trigger for that PCollection.

如果使用的是无界数据集(例如来自流数据源)，则可以使用单个全局窗口，但在应用聚合转换(如GroupByKey和 Combine)时要小心。带有默认触发器的单个全局窗口通常要求在处理之前可以使用整个数据集，而连续更新数据是不可能的。若要在使用全局窗口的无界PCollection上执行聚合，应为该PCollection指定一个非默认触发器。

### 7.3. 设置你的pcollection的窗函数(Setting your PCollection’s windowing function)

You can set the windowing function for a PCollection by applying the Window transform. When you apply the Window transform, you must provide a WindowFn. The WindowFn determines the windowing function your PCollection will use for subsequent grouping transforms, such as a fixed or sliding time window.

您可以通过应用窗口转换来设置PCollection的窗口函数。应用窗口转换时，必须提供WindowsFn。WindowFn确定窗口函数——此函数PCollection将用于后续分组转换(例如固定时间窗口或滑动时间窗口)。

When you set a windowing function, you may also want to set a trigger for your PCollection. The trigger determines when each individual window is aggregated and emitted, and helps refine how the windowing function performs with respect to late data and computing early results. See the [triggers](https://beam.apache.org/documentation/programming-guide/#triggers) section for more information.

设置窗口函数时，还可能希望为PCollection设置触发器。触发器确定何时聚合和发出每个单独的窗口，并帮助细化窗口功能对于后期数据和计算早期结果的执行方式。有关更多信息，请参见[triggers](https://beam.apache.org/documentation/programming-guide/#triggers) 部分。

#### 7.3.1. 固定时间窗（Fixed-time windows）

The following example code shows how to apply Window to divide a PCollection into fixed windows, each one minute in length:

下面的代码示例演示如何应用窗口去分pcollection到固定窗，每个窗口长度为1分钟：

PCollection**<**String**>** items **=** **...;**

PCollection**<**String**>** fixedWindowedItems **=** items**.**apply**(**

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

#### 7.3.2. 滑动时间窗（ Sliding time windows）

The following example code shows how to apply Window to divide a PCollection into sliding time windows. Each window is 30 minutes in length, and a new window begins every five seconds:

下面的示例代码演示如何应用窗口划分PCollection到滑动时间窗口。每个窗口的长度为30分钟，每5秒就有一个新窗口：

PCollection**<**String**>** items **=** **...;**

PCollection**<**String**>** slidingWindowedItems **=** items**.**apply**(**

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

#### 7.3.3. 会话窗口（Session windows）

The following example code shows how to apply Window to divide a PCollection into session windows, where each session must be separated by a time gap of at least 10 minutes:

下面的示例代码演示如何应用窗口划分PCollection到会话窗口，其中每个会话必须间隔至少10分钟：

PCollection**<**String**>** items **=** **...;**

PCollection**<**String**>** sessionWindowedItems **=** items**.**apply**(**

Window**.<**String**>**into**(**Sessions**.**withGapDuration**(**Duration**.**standardMinutes**(**10**))));**

Note that the sessions are per-key — each key in the collection will have its own session groupings depending on the data distribution.

注意，会话是按每个键进行的——集合中的每个键将根据数据分布有自己的会话分组。

#### 7.3.4. 单一全局窗口（Single global window）

If your PCollection is bounded (the size is fixed), you can assign all the elements to a single global window. The following example code shows how to set a single global window for a PCollection:

如果PCollection是有界的(大小是固定的)，则可以将所有元素分配给一个全局窗口。下面的示例代码演示如何为PCollection设置单个全局窗口：

PCollection**<**String**>** items **=** **...;**

PCollection**<**String**>** batchItems **=** items**.**apply**(**

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

### 7.4. 水印和后期数据（Watermarks and late data）

In any data processing system, there is a certain amount of lag between the time a data event occurs (the “event time”, determined by the timestamp on the data element itself) and the time the actual data element gets processed at any stage in your pipeline (the “processing time”, determined by the clock on the system processing the element). In addition, there are no guarantees that data events will appear in your pipeline in the same order that they were generated.

在任何数据处理系统中，在数据事件发生的时间(由数据元素本身的时间戳确定的“事件时间”)到实际数据元素在管道的任何阶段被处理的时间(“处理时间”，由处理该元素的系统上的时钟确定)之间存在一定程度的滞后。此外，无法保证数据事件将以与生成数据事件相同的顺序出现在管道中。

For example, let’s say we have a PCollection that’s using fixed-time windowing, with windows that are five minutes long. For each window, Beam must collect all the data with an event time timestamp in the given window range (between 0:00 and 4:59 in the first window, for instance). Data with timestamps outside that range (data from 5:00 or later) belong to a different window.

例如，假设我们有一个使用固定时间窗口的PCollection，它的窗口只有5分钟长。对于每个窗口，BEAM必须在给定的窗口范围内收集具有事件时间戳的所有数据(例如，在第一个窗口中为0：00到4：59之间)。时间戳超出该范围的数据(5：00或更后面的数据)属于不同的窗口。

However, data isn’t always guaranteed to arrive in a pipeline in time order, or to always arrive at predictable intervals. Beam tracks a watermark, which is the system’s notion of when all data in a certain window can be expected to have arrived in the pipeline. Data that arrives with a timestamp after the watermark is considered **late data**.

然而，数据并不总是按照时间顺序到达管道，或者总是以可预测的间隔到达。BEAM跟踪水印，即某一窗口中的所有管道数据何时到达。在水印之后带有时间戳的数据被认为是后期数据。

From our example, suppose we have a simple watermark that assumes approximately 30s of lag time between the data timestamps (the event time) and the time the data appears in the pipeline (the processing time), then Beam would close the first window at 5:30. If a data record arrives at 5:34, but with a timestamp that would put it in the 0:00-4:59 window (say, 3:38), then that record is late data.

在我们的例子中，假设我们有一个简单的水印，假定在数据的时间戳（事件时间）和出现在管道的时间数据（处理时间）之间存在大约30秒的滞后时间，然后在5:30Beam将关闭第一个窗口。如果一个数据记录到5:34，但有一个时间戳，就把它放在0:00-4:59窗口（也就是说，38），这个记录就是后期数据。

Note: For simplicity, we’ve assumed that we’re using a very straightforward watermark that estimates the lag time. In practice, your PCollection’s data source determines the watermark, and watermarks can be more precise or complex.

注意：为了简单起见，我们假设我们使用了一个非常直观的水印来估计延迟时间。在实践中，你的pcollection数据源确定水印，水印可以更精确的或复杂的。

Beam’s default windowing configuration tries to determines when all data has arrived (based on the type of data source) and then advances the watermark past the end of the window. This default configuration does not allow late data. [Triggers](https://beam.apache.org/documentation/programming-guide/#triggers) allow you to modify and refine the windowing strategy for a PCollection. You can use triggers to decide when each individual window aggregates and reports its results, including how the window emits late elements.

BEAM的默认窗口配置试图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的末尾。此默认配置不允许延迟数据。触发器允许您修改和改进PCollection的窗口策略。您可以使用触发器来决定每个窗口何时聚合和报告其结果，包括窗口如何发出延迟元素。

#### 7.4.1. 延迟数据管理 （ Managing late data）

You can allow late data by invoking the .withAllowedLateness operation when you set your PCollection’s windowing strategy. The following code example demonstrates a windowing strategy that will allow late data up to two days after the end of a window.

您可以在设置PCollection的窗口策略时通过调用.withAllowedLateness操作来允许延迟数据。下面的代码示例演示了一个窗口化策略，该策略允许在窗口结束后两天内的延迟数据。

PCollection**<**String**>** items **=** **...;**

PCollection**<**String**>** fixedWindowedItems **=** items**.**apply**(**

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

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

When you set .withAllowedLateness on a PCollection, that allowed lateness propagates forward to any subsequent PCollectionderived from the first PCollection you applied allowed lateness to. If you want to change the allowed lateness later in your pipeline, you must do so explictly by applying Window.configure().withAllowedLateness().

当您在PCollection上设置.withAllowedLateness时，允许的延迟将从您应用延迟的第一个PCollection一直传播到其派生的后续PCollection中。如果您想在稍后的管道中更改允许的延迟性，则必须通过应用 Window.configure().withAllowedLateness()进行明确的更改。

### 7.5. 给一个pcollection的元素添加时间戳（ Adding timestamps to a PCollection’s elements）

An unbounded source provides a timestamp for each element. Depending on your unbounded source, you may need to configure how the timestamp is extracted from the raw data stream.

无界源为每个元素提供时间戳。根据您的无界源，您可能需要配置如何从原始数据流中提取时间戳。

However, bounded sources (such as a file from TextIO) do not provide timestamps. If you need timestamps, you must add them to your PCollection’s elements.

但是，有界源(例如来自TextIO的文件)不提供时间戳。如果您需要时间戳，则必须将它们添加到PCollection的元素中。

You can assign new timestamps to the elements of a PCollection by applying a [ParDo](https://beam.apache.org/documentation/programming-guide/#pardo) transform that outputs new elements with timestamps that you set.

您可以通过应用Pardo转换将新时间戳分配给PCollection的元素，该转换用您设置的时间戳输出新元素。

An example might be if your pipeline reads log records from an input file, and each log record includes a timestamp field; since your pipeline reads the records in from a file, the file source doesn’t assign timestamps automatically. You can parse the timestamp field from each record and use a ParDo transform with a DoFn to attach the timestamps to each element in your PCollection.

例如，如果您的管道从输入文件中读取日志记录，并且每个日志记录都包含一个时间戳字段；因为您的管道从文件中读取记录，所以文件源不会自动分配时间戳。您可以从每个记录中解析时间戳字段，并使用带有DoFn的Pardo转换将时间戳附加到PCollection中的每个元素。

PCollection**<**LogEntry**>** unstampedLogs **=** **...;**

PCollection**<**LogEntry**>** stampedLogs **=**

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

**public** **void** **processElement(**ProcessContext c**)** **{**

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

Instant logTimeStamp **=** extractTimeStampFromLogEntry**(**c**.**element**());**

*// Use ProcessContext.outputWithTimestamp (rather than*

*// ProcessContext.output) to emit the entry with timestamp attached.*

c**.**outputWithTimestamp**(**c**.**element**(),** logTimeStamp**);**

**}**

**}));**

## 8. 触发器（Triggers）

When collecting and grouping data into windows, Beam uses **triggers** to determine when to emit the aggregated results of each window (referred to as a pane). If you use Beam’s default windowing configuration and [default trigger](https://beam.apache.org/documentation/programming-guide/#default-trigger), Beam outputs the aggregated result when it [estimates all data has arrived](https://beam.apache.org/documentation/programming-guide/#watermarks-and-late-data), and discards all subsequent data for that window.

当收集数据并将数据分组到窗口时，BEAM使用触发器来确定何时发出每个窗口的聚合结果(称为窗格)。如果使用BEAM的默认窗口配置和默认触发器，则BEAM在估计所有数据已到达时输出聚合结果，并为该窗口丢弃所有后续数据。

You can set triggers for your PCollections to change this default behavior. Beam provides a number of pre-built triggers that you can set:

您可以为PCollection设置触发器以更改此默认行为。BEAM提供了许多预先构建的触发器，您可以设置这些触发器：

* **Event time triggers**. These triggers operate on the event time, as indicated by the timestamp on each data element. Beam’s default trigger is event time-based.
* **Processing time triggers**. These triggers operate on the processing time – the time when the data element is processed at any given stage in the pipeline.
* **Data-driven triggers**. These triggers operate by examining the data as it arrives in each window, and firing when that data meets a certain property. Currently, data-driven triggers only support firing after a certain number of data elements.
* **Composite triggers**. These triggers combine multiple triggers in various ways.
* 事件时间触发器。这些触发器在事件时间上操作，如每个数据元素上的时间戳所指示的那样。BEAM的默认触发器是基于事件时间的。
* 处理时间触发器。这些触发器在处理时间上操作—此时间是在管道中的任何给定阶段处理数据元素的时间。
* 数据驱动触发器。这些触发器通过检查数据到达每个窗口时进行操作，并在该数据满足特定属性时触发。目前，数据驱动触发器只支持在一定数量的数据元素之后触发。
* 复合触发器。这些触发器以各种方式组合多个触发器。

At a high level, triggers provide two additional capabilities compared to simply outputting at the end of a window:

在较高级别上，与仅在窗口末尾输出相比，触发器提供了两种额外的功能：

* Triggers allow Beam to emit early results, before all the data in a given window has arrived. For example, emitting after a certain amouint of time elapses, or after a certain number of elements arrives.
* Triggers allow processing of late data by triggering after the event time watermark passes the end of the window.
* 触发器允许BEAM在给定窗口中的所有数据到达之前发出早期结果。例如，在某一时间间隔之后，或在一定数量的元素到达之后发出信号。
* 触发器允许在事件时间水印通过窗口结尾之后触发处理后期数据。

These capabilities allow you to control the flow of your data and balance between different factors depending on your use case:

这些功能允许您根据用例控制数据流并在不同因素之间进行平衡：

* **Completeness:** How important is it to have all of your data before you compute your result?
* **Latency:** How long do you want to wait for data? For example, do you wait until you think you have all data? Do you process data as it arrives?
* **Cost:** How much compute power/money are you willing to spend to lower the latency?
* 完整性：在计算结果之前拥有所有数据有多重要？
* 延迟：你想等待多久？例如，你会等到你认为你拥有了所有的数据吗？当数据到达时，你会处理它吗？
* 成本：你愿意花费多少计算能力/金钱来降低延迟？

For example, a system that requires time-sensitive updates might use a strict time-based trigger that emits a window every N seconds, valuing promptness over data completeness. A system that values data completeness more than the exact timing of results might choose to use Beam’s default trigger, which fires at the end of the window.

例如，需要对时间敏感更新的系统可能使用严格的基于时间的触发器，每N秒发出一次窗口，这比数据的完整性更重要。评估数据完整性的系统可能会选择使用BEAM的默认触发器，该触发器在窗口的末尾触发。

You can also set a trigger for an unbounded PCollection that uses a [single global window for its windowing function](https://beam.apache.org/documentation/programming-guide/#windowing). This can be useful when you want your pipeline to provide periodic updates on an unbounded data set — for example, a running average of all data provided to the present time, updated every N seconds or every N elements.

您还可以为使用单个全局窗口的窗口功能的无界PCollection设置触发器。当您希望管道对无界数据集提供定期更新时，这可能非常有用--例如，提供到当前时间的所有数据的运行平均值，每N秒更新一次或每N个元素更新一次。

### 8.1. 事件时间触发（Event time triggers）

The AfterWatermark trigger operates on event time. The AfterWatermark trigger emits the contents of a window after the [watermark](https://beam.apache.org/documentation/programming-guide/#watermarks-and-late-data)passes the end of the window, based on the timestamps attached to the data elements. The watermark is a global progress metric, and is Beam’s notion of input completeness within your pipeline at any given point. AfterWatermark.pastEndOfWindow() only fires when the watermark passes the end of the window.

 AfterWatermark触发器在事件时间运行。AfterWatermark触发器根据附加到数据元素的时间戳，在水印通过窗口结尾后发出窗口的内容。水印是一个全局进度度量，是BEAM在任意给定点的管道内输入完整性的概念。AfterWatermark.pastEndOfWindow() 只有当水印通过窗口结束时才触发.

In addition, you can use .withEarlyFirings(trigger) and .withLateFirings(trigger) to configure triggers that fire if your pipeline receives data before or after the end of the window.

同时，如果管道在窗口结束之前或之后接收到数据，则可以使用.withEarlyFirings(触发器)和.withLateFirings(触发器)配置触发器。

The following example shows a billing scenario, and uses both early and late firings:

下面的示例显示了一个计费方案，并采用早期和晚期事件：

*// Create a bill at the end of the month.*

AfterWatermark**.**pastEndOfWindow**()**

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

**.**withEarlyFirings**(**

AfterProcessingTime

**.**pastFirstElementInPane**()**

**.**plusDuration**(**Duration**.**standardMinutes**(**1**))**

*// Fire on any late data so the bill can be corrected.*

**.**withLateFirings**(**AfterPane**.**elementCountAtLeast**(**1**))**

*# The Beam SDK for Python does not support triggers.*

#### 8.1.1. 默认触发器 Default trigger

The default trigger for a PCollection is based on event time, and emits the results of the window when the Beam’s watermark passes the end of the window, and then fires each time late data arrives.

PCollection的默认触发器基于事件时间，当波束的水印通过窗口的末尾时发出窗口的结果，然后每次延迟数据到达时触发。

However, if you are using both the default windowing configuration and the default trigger, the default trigger emits exactly once, and late data is discarded. This is because the default windowing configuration has an allowed lateness value of 0. See the Handling Late Data section for information about modifying this behavior.

但是，如果同时使用默认窗口配置和默认触发器，则默认触发器只会发出一次，而延迟数据将被丢弃。这是因为默认的窗口配置允许延迟值为0。有关修改此行为的信息，请参阅“处理延迟数据”部分。

### 8.2. 处理时间触发（Processing time triggers）

The AfterProcessingTime trigger operates on processing time. For example, the AfterProcessingTime.pastFirstElementInPane() trigger emits a window after a certain amount of processing time has passed since data was received. The processing time is determined by the system clock, rather than the data element’s timestamp.

AfterProcessingTime触发器对处理时间进行操作。例如，AfterProcessingTime.pastFirstElementInPane()触发器在收到数据后经过一定的处理时间后发出一个窗口。处理时间是由系统时钟决定的，而不是数据元素的时间戳。

The AfterProcessingTime trigger is useful for triggering early results from a window, particularly a window with a large time frame such as a single global window.

AfterProcessTime触发器用于触发窗口的早期结果，特别是具有大时间帧的窗口，例如单个全局窗口。

### 8.3. 数据驱动触发（Data-driven triggers）

Beam provides one data-driven trigger, AfterPane.elementCountAtLeast(). This trigger works on an element count; it fires after the current pane has collected at least N elements. This allows a window to emit early results (before all the data has accumulated), which can be particularly useful if you are using a single global window.

BEAM提供了一个数据驱动触发器，AfterPane.elementCountAtLeast()。此触发器适用于元素计数；它在当前窗格至少收集了N个元素后触发。这允许一个窗口发出早期结果(在所有数据积累之前)，如果您使用的是单个全局窗口，这将特别有用。

It is important to note that if, for example, you use .elementCountAtLeast(50) and only 32 elements arrive, those 32 elements sit around forever. If the 32 elements are important to you, consider using [composite triggers](https://beam.apache.org/documentation/programming-guide/#composite-triggers) to combine multiple conditions. This allows you to specify multiple firing conditions such as “fire either when I receive 50 elements, or every 1 second”.

重要的是要注意，例如，如果您使用的是.elementCountAtLeast(50)，并且只有32个元素到达，那么这32个元素将永远存在。如果这32个元素对您很重要，请考虑使用复合触发器（[composite triggers](https://beam.apache.org/documentation/programming-guide/#composite-triggers)）组合多个条件。这允许您指定多个触发条件，例如“当我接收50个元素时或每1秒触发一次”。

### 8.4. 设置触发（Setting a trigger）

When you set a windowing function for a PCollection by using the Window transform, you can also specify a trigger.

使用窗口转换为PCollection设置窗口函数时，还可以指定触发器。

You set the trigger(s) for a PCollection by invoking the method .triggering() on the result of your Window.into() transform, as follows:

通过调用Window.into()转换的结果.triggering() ，设置PCollection的触发器，如下所示：

PCollection**<**String**>** pc **=** **...;**

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

**.**triggering**(**AfterProcessingTime**.**pastFirstElementInPane**()**

**.**plusDelayOf**(**Duration**.**standardMinutes**(**1**)))**

**.**discardingFiredPanes**());**

*# The Beam SDK for Python does not support triggers.*

This code sample sets a time-based trigger for a PCollection, which emits results one minute after the first element in that window has been processed. The last line in the code sample, .discardingFiredPanes(), is the window’s **accumulation mode**.

此代码示例为pCollection设置一个基于时间的触发器，该触发器在处理该窗口中的第一个元素在一分钟后发出的结果。代码示例中的最后一行.discardingFiredPanes()是窗口的累积模式。

#### 8.4.1. 窗积累模式（ Window accumulation modes）

When you specify a trigger, you must also set the window’s **accumulation mode**. When a trigger fires, it emits the current contents of the window as a pane. Since a trigger can fire multiple times, the accumulation mode determines whether the system accumulates the window panes as the trigger fires, or discards them.

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，它会将窗口的当前内容作为窗格发出。由于触发器可以多次触发，因此累积模式决定系统是将窗口窗格累加为触发器触发，还是丢弃窗口窗格。

To set a window to accumulate the panes that are produced when the trigger fires, invoke.accumulatingFiredPanes() when you set the trigger. To set a window to discard fired panes, invoke .discardingFiredPanes().

设置一个窗口以累积触发器触发时生成的窗格。当你要设置触发器时调用.accumulatingFiredPanes() 。要设置一个窗口来丢弃触发的窗格，请调用.discardingFiredPanes()。

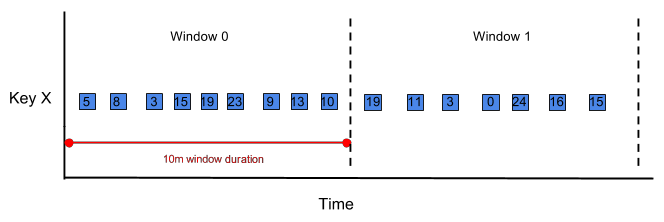
Let’s look an example that uses a PCollection with fixed-time windowing and a data-based trigger. This is something you might do if, for example, each window represented a ten-minute running average, but you wanted to display the current value of the average in a UI more frequently than every ten minutes. We’ll assume the following conditions:

让我们来看看一个使用带有固定时间窗口的pCollection和基于数据的触发器的示例。例如，如果每个窗口表示一个10分钟的平均运行时间，但是您希望在UI中比每隔10分钟更频繁地显示平均值的当前值，那么您可能会这样做。我们将假定以下条件：

* The PCollection uses 10-minute fixed-time windows.
* The PCollection has a repeating trigger that fires every time 3 elements arrive.
* PCollection使用10分钟的固定时间窗口。
* PCollection有一个重复触发器，每次3个元素到达时都会触发。

The following diagram shows data events for key X as they arrive in the PCollection and are assigned to windows. To keep the diagram a bit simpler, we’ll assume that the events all arrive in the pipeline in order.

下图显示了键X的数据事件，当它们到达PCollection并分配给windows时。为了使图表简单一点，我们假设所有事件都按照顺序到达。



##### 8.4.1.1. 积累模式（Accumulating mode）

If our trigger is set to .accumulatingFiredPanes, the trigger emits the following values each time it fires. Keep in mind that the trigger fires every time three elements arrive:

如果我们的触发设置accumulatingfiredpanes，触发器每次发出以下值会触发。记住每次三个元素到达时触发器触发：

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]

##### 8.4.1.2. 丢弃模式（Discarding mode）

If our trigger is set to .discardingFiredPanes, the trigger emits the following values on each firing:

如果我们的触发器设置为.discardingFiredPanes，则触发器在每次触发时发出以下值：

First trigger firing: [5, 8, 3]

Second trigger firing: [15, 19, 23]

Third trigger firing: [9, 13, 10]

#### 8.4.2. 延后数据处理（Handling late data）

If you want your pipeline to process data that arrives after the watermark passes the end of the window, you can apply an allowed lateness when you set your windowing configuration. This gives your trigger the opportunity to react to the late data. If allowed lateness is set, the default trigger will emit new results immediately whenever late data arrives.

如果希望管道处理水印通过窗口结束后到达的数据，则可以在设置窗口配置时应用允许的延迟。这使您的触发器有机会对迟来的数据作出反应。如果允许延迟设置，则当延迟数据到达时，默认触发器将立即发出新结果。

You set the allowed lateness by using .withAllowedLateness() when you set your windowing function:

在设置窗口函数时，您可以使用.withAllowedLateness()设置允许的延迟：

PCollection**<**String**>** pc **=** **...;**

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

**.**triggering**(**AfterProcessingTime**.**pastFirstElementInPane**()**

**.**plusDelayOf**(**Duration**.**standardMinutes**(**1**)))**

**.**withAllowedLateness**(**Duration**.**standardMinutes**(**30**));**

*# The Beam SDK for Python does not support triggers.*

This allowed lateness propagates to all PCollections derived as a result of applying transforms to the original PCollection. If you want to change the allowed lateness later in your pipeline, you can apply Window.configure().withAllowedLateness() again, explicitly.

这允许延迟传播到由原始PCollection通过转换应用而派生的所有PCollection。如果您想在稍后的管道中更改允许的延迟，可以显式地再次应用 Window.configure().withAllowedLateness()

**8.5. 复合触发器（Composite triggers）**

You can combine multiple triggers to form **composite triggers**, and can specify a trigger to emit results repeatedly, at most once, or under other custom conditions.

您可以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来重复、最多一次或在其他自定义条件下发出结果。

#### 8.5.1. 复合触发器的类型（Composite trigger types）

Beam includes the following composite triggers:

BEAM包括以下复合触发器：

* You can add additional early firings or late firings to AfterWatermark.pastEndOfWindow via .withEarlyFirings and.withLateFirings.
* ·您可以通过. AfterWatermark.pastEndOfWindow via .withEarlyFirings and.withLateFirings.

添加额外的早期触发或后期触发。

* Repeatedly.forever specifies a trigger that executes forever. Any time the trigger’s conditions are met, it causes a window to emit results and then resets and starts over. It can be useful to combine Repeatedly.forever with .orFinally to specify a condition that causes the repeating trigger to stop.
* repeatedly.forever指定触发永远执行。每当触发条件满足时，都会导致窗口释放结果，然后重新启动并重新启动。把Repeatedly.forever 和 .orFinally组合起来指定一个导致重复触发器停止的条件是很有用的。
* AfterEach.inOrder combines multiple triggers to fire in a specific sequence. Each time a trigger in the sequence emits a window, the sequence advances to the next trigger.
* AfterEach.inOrder将多个触发器组合在一个特定的序列中触发。每次序列中的触发器发出一个窗口，该序列就会前进到下一个触发器。
* AfterFirst takes multiple triggers and emits the first time any of its argument triggers is satisfied. This is equivalent to a logical OR operation for multiple triggers.
* AfterFirst在它任何触发器第一时间满足时接受多个触发器并发出。这相当于对多个触发器的逻辑或操作。
* AfterAll takes multiple triggers and emits when all of its argument triggers are satisfied. This is equivalent to a logical AND operation for multiple triggers.
* AfterAll 在它的所有参数触发器都满足时接受多个触发器并发出。这相当于多个触发器的逻辑和操作。
* orFinally can serve as a final condition to cause any trigger to fire one final time and never fire again.
* orFinally可以作为最后一个条件来触发任何一次触发，之后不再触发。

#### 8.5.2. AfterWatermark.pastEndOfWindow的组成（ Composition with AfterWatermark.pastEndOfWindow）

Some of the most useful composite triggers fire a single time when Beam estimates that all the data has arrived (i.e. when the watermark passes the end of the window) combined with either, or both, of the following:

一些最有用的复合触发器只触发一次，当BEAM估计所有数据已经到达(即当水印通过窗口结尾时)与下列任一项或两者结合在一起：

* Speculative firings that precede the watermark passing the end of the window to allow faster processing of partial results.
* ·通过窗口末端的水印之前的投机触发，以便窗口更快地处理部分结果。
* Late firings that happen after the watermark passes the end of the window, to allow for handling late-arriving data
* 水印通过窗口结尾之后触发，用来允许处理后到达的数据。

You can express this pattern using AfterWatermark.pastEndOfWindow. For example, the following example trigger code fires on the following conditions:

您可以使用 AfterWatermark.pastEndOfWindow.表示此模式。例如，下面的示例触发器代码在以下条件下触发：

* On Beam’s estimate that all the data has arrived (the watermark passes the end of the window)
* 在Beam的估计下，所有的数据都已经到达（水印通过窗口的尽头）
* Any time late data arrives, after a ten-minute delay
* 在10分钟的延迟后，任何延迟数据到达
* After two days, we assume no more data of interest will arrive, and the trigger stops executing
* ·在两天后，我们假设不会有更多有利益的数据到达，并且触发器停止执行。

**.**apply**(**Window

**.**configure**()**

**.**triggering**(**AfterWatermark

**.**pastEndOfWindow**()**

**.**withLateFirings**(**AfterProcessingTime

**.**pastFirstElementInPane**()**

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

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

*# The Beam SDK for Python does not support triggers.*

#### 8.5.3. 其他复合触发器（Other composite triggers）

You can also build other sorts of composite triggers. The following example code shows a simple composite trigger that fires whenever the pane has at least 100 elements, or after a minute.

还可以构建其他类型的复合触发器。下面的示例代码显示了一个简单的复合触发器，每当窗格中至少有100个元素时或在一分钟后触发该触发器。

Repeatedly**.**forever**(**AfterFirst**.**of**(**

AfterPane**.**elementCountAtLeast**(**100**),**

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

# Design Your Pipeline

* [What to consider when designing your pipeline](https://beam.apache.org/documentation/pipelines/design-your-pipeline/#what-to-consider-when-designing-your-pipeline)
* [A basic pipeline](https://beam.apache.org/documentation/pipelines/design-your-pipeline/#a-basic-pipeline)
* [Branching PCollections](https://beam.apache.org/documentation/pipelines/design-your-pipeline/#branching-pcollections)
  + [Multiple transforms process the same PCollection](https://beam.apache.org/documentation/pipelines/design-your-pipeline/#multiple-transforms-process-the-same-pcollection)
  + [A single transform that produces multiple outputs](https://beam.apache.org/documentation/pipelines/design-your-pipeline/#a-single-transform-that-produces-multiple-outputs)
* [Merging PCollections](https://beam.apache.org/documentation/pipelines/design-your-pipeline/#merging-pcollections)
* [Multiple sources](https://beam.apache.org/documentation/pipelines/design-your-pipeline/#multiple-sources)
* [What’s next](https://beam.apache.org/documentation/pipelines/design-your-pipeline/#whats-next)

This page helps you design your Apache Beam pipeline. It includes information about how to determine your pipeline’s structure, how to choose which transforms to apply to your data, and how to determine your input and output methods.

Before reading this section, it is recommended that you become familiar with the information in the [Beam programming guide](https://beam.apache.org/documentation/programming-guide).

## What to consider when designing your pipeline

When designing your Beam pipeline, consider a few basic questions:

* **Where is your input data stored?** How many sets of input data do you have? This will determine what kinds of Read transforms you’ll need to apply at the start of your pipeline.
* **What does your data look like?** It might be plaintext, formatted log files, or rows in a database table. Some Beam transforms work exclusively on PCollections of key/value pairs; you’ll need to determine if and how your data is keyed and how to best represent that in your pipeline’s PCollection(s).
* **What do you want to do with your data?** The core transforms in the Beam SDKs are general purpose. Knowing how you need to change or manipulate your data will determine how you build core transforms like [ParDo](https://beam.apache.org/documentation/programming-guide/#pardo), or when you use pre-written transforms included with the Beam SDKs.
* **What does your output data look like, and where should it go?** This will determine what kinds of Write transforms you’ll need to apply at the end of your pipeline.

## A basic pipeline

The simplest pipelines represent a linear flow of operations, as shown in figure 1.

A linear pipeline starts with one input collection, sequentially applies
  three transforms, and ends with one output collection.

Figure 1: A linear pipeline.

However, your pipeline can be significantly more complex. A pipeline represents a [Directed Acyclic Graph](https://en.wikipedia.org/wiki/Directed_acyclic_graph) of steps. It can have multiple input sources, multiple output sinks, and its operations (PTransforms) can both read and output multiple PCollections. The following examples show some of the different shapes your pipeline can take.

## Branching PCollections

It’s important to understand that transforms do not consume PCollections; instead, they consider each individual element of a PCollection and create a new PCollection as output. This way, you can do different things to different elements in the same PCollection.

### Multiple transforms process the same PCollection

You can use the same PCollection as input for multiple transforms without consuming the input or altering it.

The pipeline in figure 2 is a branching pipeline. The pipeline reads its input (first names represented as strings) from a database table and creates a PCollection of table rows. Then, the pipeline applies multiple transforms to the **same** PCollection. Transform A extracts all the names in that PCollection that start with the letter ‘A’, and Transform B extracts all the names in that PCollection that start with the letter ‘B’. Both transforms A and B have the same input PCollection.

The pipeline applies two transforms to a single input collection. Each
  transform produces an output collection.

Figure 2: A branching pipeline. Two transforms are applied to a single PCollection of database table rows.

The following example code applies two transforms to a single input collection.

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**());**

**}**

**}**

**}));**

### A single transform that produces multiple outputs

Another way to branch a pipeline is to have a **single** transform output to multiple PCollections by using [tagged outputs](https://beam.apache.org/documentation/programming-guide/#additional-outputs). Transforms that produce more than one output process each element of the input once, and output to zero or more PCollections.

Figure 3 illustrates the same example described above, but with one transform that produces multiple outputs. Names that start with ‘A’ are added to the main output PCollection, and names that start with ‘B’ are added to an additional output PCollection.

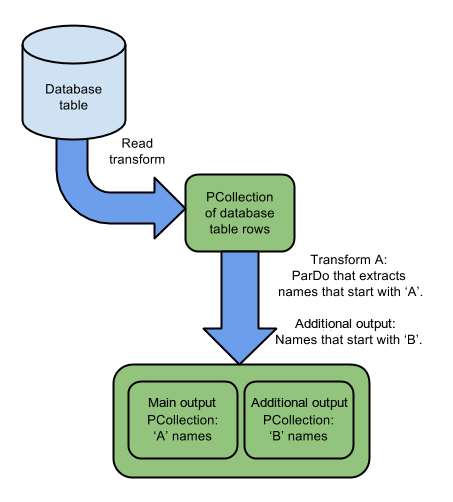


Figure 3: A pipeline with a transform that outputs multiple PCollections.

If we compare the pipelines in figure 2 and figure 3, you can see they perform the same operation in different ways. The pipeline in figure 2 contains two transforms that process the elements in the same input PCollection. One transform uses the following logic:

if (starts with 'A') { outputToPCollectionA }

while the other transform uses:

if (starts with 'B') { outputToPCollectionB }

Because each transform reads the entire input PCollection, each element in the input PCollection is processed twice.

The pipeline in figure 3 performs the same operation in a different way - with only one transform that uses the following logic:

if (starts with 'A') { outputToPCollectionA } else if (starts with 'B') { outputToPCollectionB }

where each element in the input PCollection is processed once.

The following example code applies one transform that processes each element once and outputs two collections.

*// Define two TupleTags, one for each output.*

**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"**))** **{**

*// Emit to main output, which is the output with tag startsWithATag.*

c**.**output**(**c**.**element**());**

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

*// Emit to output with tag startsWithBTag.*

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

**}**

**}**

**})**

*// Specify main output. In this example, it is the output*

*// with tag startsWithATag.*

**.**withOutputTags**(**startsWithATag**,**

*// Specify the output with tag startsWithBTag, as a TupleTagList.*

TupleTagList**.**of**(**startsWithBTag**)));**

*// Get subset of the output with tag startsWithATag.*

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

*// Get subset of the output with tag startsWithBTag.*

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

You can use either mechanism to produce multiple output PCollections. However, using additional outputs makes more sense if the transform’s computation per element is time-consuming.

## Merging PCollections

Often, after you’ve branched your PCollection into multiple PCollections via multiple transforms, you’ll want to merge some or all of those resulting PCollections back together. You can do so by using one of the following:

* **Flatten** - You can use the Flatten transform in the Beam SDKs to merge multiple PCollections of the **same type**.
* **Join** - You can use the CoGroupByKey transform in the Beam SDK to perform a relational join between two PCollections. The PCollections must be keyed (i.e. they must be collections of key/value pairs) and they must use the same key type.

The example in figure 4 is a continuation of the example in figure 2 in [the section above](https://beam.apache.org/documentation/pipelines/design-your-pipeline/#multiple-transforms-process-the-same-pcollection). After branching into two PCollections, one with names that begin with ‘A’ and one with names that begin with ‘B’, the pipeline merges the two together into a single PCollectionthat now contains all names that begin with either ‘A’ or ‘B’. Here, it makes sense to use Flatten because the PCollections being merged both contain the same type.



Figure 4: A pipeline that merges two collections into one collection with the Flatten transform.

The following example code applies Flatten to merge two collections.

*//merge the two PCollections with Flatten*

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

PCollection**<**String**>** mergedCollectionWithFlatten **=** collectionList

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

*// continue with the new merged PCollection*

mergedCollectionWithFlatten**.**apply**(...);**

## Multiple sources

Your pipeline can read its input from one or more sources. If your pipeline reads from multiple sources and the data from those sources is related, it can be useful to join the inputs together. In the example illustrated in figure 5 below, the pipeline reads names and addresses from a database table, and names and order numbers from a Kafka topic. The pipeline then uses CoGroupByKey to join this information, where the key is the name; the resulting PCollection contains all the combinations of names, addresses, and orders.



Figure 5: A pipeline that does a relational join of two input collections.

The following example code applies Join to join two input collections.

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**>();**

*// Merge collection values into a CoGbkResult collection.*

PCollection**<**KV**<**String**,** CoGbkResult**>>** joinedCollection **=**

KeyedPCollectionTuple**.**of**(**addressTag**,** userAddress**)**

**.**and**(**orderTag**,** userOrder**)**

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

coGbkResultCollection**.**apply**(...);**

# Create Your Pipeline

* [Creating Your Pipeline Object](https://beam.apache.org/documentation/pipelines/create-your-pipeline/#creating-your-pipeline-object)
* [Reading Data Into Your Pipeline](https://beam.apache.org/documentation/pipelines/create-your-pipeline/#reading-data-into-your-pipeline)
* [Applying Transforms to Process Pipeline Data](https://beam.apache.org/documentation/pipelines/create-your-pipeline/#applying-transforms-to-process-pipeline-data)
* [Writing or Outputting Your Final Pipeline Data](https://beam.apache.org/documentation/pipelines/create-your-pipeline/#writing-or-outputting-your-final-pipeline-data)
* [Running Your Pipeline](https://beam.apache.org/documentation/pipelines/create-your-pipeline/#running-your-pipeline)
* [What’s next](https://beam.apache.org/documentation/pipelines/create-your-pipeline/#whats-next)

Your Beam program expresses a data processing pipeline, from start to finish. This section explains the mechanics of using the classes in the Beam SDKs to build a pipeline. To construct a pipeline using the classes in the Beam SDKs, your program will need to perform the following general steps:

* Create a Pipeline object.
* Use a **Read** or **Create** transform to create one or more PCollections for your pipeline data.
* Apply **transforms** to each PCollection. Transforms can change, filter, group, analyze, or otherwise process the elements in a PCollection. Each transform creates a new output PCollection, to which you can apply additional transforms until processing is complete.
* **Write** or otherwise output the final, transformed PCollections.
* **Run** the pipeline.

## Creating Your Pipeline Object

A Beam program often starts by creating a Pipeline object.

In the Beam SDKs, each pipeline is represented by an explicit object of type Pipeline. Each Pipeline object is an independent entity that encapsulates both the data the pipeline operates over and the transforms that get applied to that data.

To create a pipeline, declare a Pipeline object, and pass it some [configuration options](https://beam.apache.org/documentation/programming-guide#configuring-pipeline-options).

*// Start by defining the options for the pipeline.*

PipelineOptions options **=** PipelineOptionsFactory**.**create**();**

*// Then create the pipeline.*

Pipeline p **=** Pipeline**.**create**(**options**);**

## Reading Data Into Your Pipeline

To create your pipeline’s initial PCollection, you apply a root transform to your pipeline object. A root transform creates a PCollectionfrom either an external data source or some local data you specify.

There are two kinds of root transforms in the Beam SDKs: Read and Create. Read transforms read data from an external source, such as a text file or a database table. Create transforms create a PCollection from an in-memory java.util.Collection.

The following example code shows how to apply a TextIO.Read root transform to read data from a text file. The transform is applied to a Pipeline object p, and returns a pipeline data set in the form of a PCollection<String>:

PCollection**<**String**>** lines **=** p**.**apply**(**

"ReadLines"**,** TextIO**.**read**().**from**(**"gs://some/inputData.txt"**));**

## Applying Transforms to Process Pipeline Data

You can manipulate your data using the various [transforms](https://beam.apache.org/documentation/programming-guide/#transforms) provided in the Beam SDKs. To do this, you **apply** the trannsforms to your pipeline’s PCollection by calling the apply method on each PCollection that you want to process and passing the desired transform object as an argument.

The following code shows how to apply a transform to a PCollection of strings. The transform is a user-defined custom transform that reverses the contents of each string and outputs a new PCollection containing the reversed strings.

The input is a PCollection<String> called words; the code passes an instance of a PTransform object called ReverseWords to apply, and saves the return value as the PCollection<String> called reversedWords.

PCollection**<**String**>** words **=** **...;**

PCollection**<**String**>** reversedWords **=** words**.**apply**(new** ReverseWords**());**

## Writing or Outputting Your Final Pipeline Data

Once your pipeline has applied all of its transforms, you’ll usually need to output the results. To output your pipeline’s final PCollections, you apply a Write transform to that PCollection. Write transforms can output the elements of a PCollection to an external data sink, such as a database table. You can use Write to output a PCollection at any time in your pipeline, although you’ll typically write out data at the end of your pipeline.

The following example code shows how to apply a TextIO.Write transform to write a PCollection of String to a text file:

PCollection**<**String**>** filteredWords **=** **...;**

filteredWords**.**apply**(**"WriteMyFile"**,** TextIO**.**write**().**to**(**"gs://some/outputData.txt"**));**

## Running Your Pipeline

Once you have constructed your pipeline, use the run method to execute the pipeline. Pipelines are executed asynchronously: the program you create sends a specification for your pipeline to a **pipeline runner**, which then constructs and runs the actual series of pipeline operations.

p**.**run**();**

# Test Your Pipeline

* [Testing Individual DoFn Objects](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#testing-individual-dofn-objects)
  + [Creating a DoFnTester](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#creating-a-dofntester)
  + [Creating Test Inputs](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#creating-test-inputs)
    - [Side Inputs](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#side-inputs)
    - [Additional Outputs](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#additional-outputs)
  + [Processing Test Inputs and Checking Results](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#processing-test-inputs-and-checking-results)
* [Testing Composite Transforms](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#testing-composite-transforms)
  + [TestPipeline](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#testpipeline)
  + [Using the Create Transform](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#using-the-create-transform)
  + [PAssert](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#passert)
  + [An Example Test for a Composite Transform](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#an-example-test-for-a-composite-transform)
* [Testing a Pipeline End-to-End](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#testing-a-pipeline-end-to-end)
  + [Testing the WordCount Pipeline](https://beam.apache.org/documentation/pipelines/test-your-pipeline/#testing-the-wordcount-pipeline)

Testing your pipeline is a particularly important step in developing an effective data processing solution. The indirect nature of the Beam model, in which your user code constructs a pipeline graph to be executed remotely, can make debugging-failed runs a non-trivial task. Often it is faster and simpler to perform local unit testing on your pipeline code than to debug a pipeline’s remote execution.

Before running your pipeline on the runner of your choice, unit testing your pipeline code locally is often the best way to identify and fix bugs in your pipeline code. Unit testing your pipeline locally also allows you to use your familiar/favorite local debugging tools.

You can use [DirectRunner](https://beam.apache.org/documentation/runners/direct), a local runner helpful for testing and local development.

After you test your pipeline using the DirectRunner, you can use the runner of your choice to test on a small scale. For example, use the Flink runner with a local or remote Flink cluster.

The Beam SDKs provide a number of ways to unit test your pipeline code, from the lowest to the highest levels. From the lowest to the highest level, these are:

* You can test the individual function objects, such as [DoFn](https://beam.apache.org/documentation/programming-guide/#pardo)s, inside your pipeline’s core transforms.
* You can test an entire [Composite Transform](https://beam.apache.org/documentation/programming-guide/#composite-transforms) as a unit.
* You can perform an end-to-end test for an entire pipeline.

To support unit testing, the Beam SDK for Java provides a number of test classes in the [testing package](https://github.com/apache/beam/tree/master/sdks/java/core/src/test/java/org/apache/beam/sdk). You can use these tests as references and guides.

## Testing Individual DoFn Objects

The code in your pipeline’s DoFn functions runs often, and often across multiple Compute Engine instances. Unit-testing your DoFnobjects before running them using a runner service can save a great deal of debugging time and energy.

The Beam SDK for Java provides a convenient way to test an individual DoFn called [DoFnTester](https://github.com/apache/beam/blob/master/sdks/java/core/src/test/java/org/apache/beam/sdk/transforms/DoFnTesterTest.java), which is included in the SDK Transforms package.

DoFnTesteruses the [JUnit](http://junit.org/) framework. To use DoFnTester, you’ll need to do the following:

1. Create a DoFnTester. You’ll need to pass an instance of the DoFn you want to test to the static factory method for DoFnTester.
2. Create one or more main test inputs of the appropriate type for your DoFn. If your DoFn takes side inputs and/or produces [multiple outputs](https://beam.apache.org/documentation/programming-guide#additional-outputs), you should also create the side inputs and the output tags.
3. Call DoFnTester.processBundle to process the main inputs.
4. Use JUnit’s Assert.assertThat method to ensure the test outputs returned from processBundle match your expected values.

### Creating a DoFnTester

To create a DoFnTester, first create an instance of the DoFn you want to test. You then use that instance when you create a DoFnTester using the .of() static factory method:

**static** **class** **MyDoFn** **extends** DoFn**<**String**,** Integer**>** **{** **...** **}**

MyDoFn myDoFn **=** **...;**

DoFnTester**<**String**,** Integer**>** fnTester **=** DoFnTester**.**of**(**myDoFn**);**

### Creating Test Inputs

You’ll need to create one or more test inputs for DoFnTester to send to your DoFn. To create test inputs, simply create one or more input variables of the same input type that your DoFn accepts. In the case above:

**static** **class** **MyDoFn** **extends** DoFn**<**String**,** Integer**>** **{** **...** **}**

MyDoFn myDoFn **=** **...;**

DoFnTester**<**String**,** Integer**>** fnTester **=** DoFnTester**.**of**(**myDoFn**);**

String testInput **=** "test1"**;**

#### Side Inputs

If your DoFn accepts side inputs, you can create those side inputs by using the method DoFnTester.setSideInputs.

**static** **class** **MyDoFn** **extends** DoFn**<**String**,** Integer**>** **{** **...** **}**

MyDoFn myDoFn **=** **...;**

DoFnTester**<**String**,** Integer**>** fnTester **=** DoFnTester**.**of**(**myDoFn**);**

PCollectionView**<**List**<**Integer**>>** sideInput **=** **...;**

Iterable**<**Integer**>** value **=** **...;**

fnTester**.**setSideInputInGlobalWindow**(**sideInput**,** value**);**

See the ParDo documentation on [side inputs](https://beam.apache.org/documentation/programming-guide/#side-inputs) for more information.

#### Additional Outputs

If your DoFn produces multiple output PCollections, you’ll need to set the appropriate TupleTag objects that you’ll use to access each output. A DoFn with multiple outputs produces a PCollectionTuple for each output; you’ll need to provide a TupleTagList that corresponds to each output in that tuple.

Suppose your DoFn produces outputs of type String and Integer. You create TupleTag objects for each, and bundle them into a TupleTagList, then set it for the DoFnTester as follows:

**static** **class** **MyDoFn** **extends** DoFn**<**String**,** Integer**>** **{** **...** **}**

MyDoFn myDoFn **=** **...;**

DoFnTester**<**String**,** Integer**>** fnTester **=** DoFnTester**.**of**(**myDoFn**);**

TupleTag**<**String**>** tag1 **=** **...;**

TupleTag**<**Integer**>** tag2 **=** **...;**

TupleTagList tags **=** TupleTagList**.**of**(**tag1**).**and**(**tag2**);**

fnTester**.**setOutputTags**(**tags**);**

See the ParDo documentation on [additional outputs](https://beam.apache.org/documentation/programming-guide/#additional-outputs) for more information.

### Processing Test Inputs and Checking Results

To process the inputs (and thus run the test on your DoFn), you call the method DoFnTester.processBundle. When you call processBundle, you pass one or more main test input values for your DoFn. If you set side inputs, the side inputs are available to each batch of main inputs that you provide.

DoFnTester.processBundle returns a List of outputs—that is, objects of the same type as the DoFn’s specified output type. For a DoFn<String, Integer>, processBundle returns a List<Integer>:

**static** **class** **MyDoFn** **extends** DoFn**<**String**,** Integer**>** **{** **...** **}**

MyDoFn myDoFn **=** **...;**

DoFnTester**<**String**,** Integer**>** fnTester **=** DoFnTester**.**of**(**myDoFn**);**

String testInput **=** "test1"**;**

List**<**Integer**>** testOutputs **=** fnTester**.**processBundle**(**testInput**);**

To check the results of processBundle, you use JUnit’s Assert.assertThat method to test if the List of outputs contains the values you expect:

String testInput **=** "test1"**;**

List**<**Integer**>** testOutputs **=** fnTester**.**processBundle**(**testInput**);**

Assert**.**assertThat**(**testOutputs**,** Matchers**.**hasItems**(...));**

*// Process a larger batch in a single step.*

Assert**.**assertThat**(**fnTester**.**processBundle**(**"input1"**,** "input2"**,** "input3"**),** Matchers**.**hasItems**(...));**

## Testing Composite Transforms

To test a composite transform you’ve created, you can use the following pattern:

* Create a TestPipeline.
* Create some static, known test input data.
* Use the Create transform to create a PCollection of your input data.
* Apply your composite transform to the input PCollection and save the resulting output PCollection.
* Use PAssert and its subclasses to verify that the output PCollection contains the elements that you expect.

### TestPipeline

[TestPipeline](https://github.com/apache/beam/blob/master/sdks/java/core/src/main/java/org/apache/beam/sdk/testing/TestPipeline.java) is a class included in the Beam Java SDK specifically for testing transforms. For tests, use TestPipeline in place of Pipeline when you create the pipeline object. Unlike Pipeline.create, TestPipeline.create handles setting PipelineOptionsinterally.

You create a TestPipeline as follows:

Pipeline p **=** TestPipeline**.**create**();**

**Note:** Read about testing unbounded pipelines in Beam in [this blog post](https://beam.apache.org/blog/2016/10/20/test-stream.html).

### Using the Create Transform

You can use the Create transform to create a PCollection out of a standard in-memory collection class, such as Java List. See [Creating a PCollection](https://beam.apache.org/documentation/programming-guide/#creating-a-pcollection) for more information.

### PAssert

[PAssert](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/index.html?org/apache/beam/sdk/testing/PAssert.html) is a class included in the Beam Java SDK that is an assertion on the contents of a PCollection. You can use PAssertto verify that a PCollection contains a specific set of expected elements.

For a given PCollection, you can use PAssert to verify the contents as follows:

PCollection**<**String**>** output **=** **...;**

*// Check whether a PCollection contains some elements in any order.*

PAssert**.**that**(**output**)**

**.**containsInAnyOrder**(**

"elem1"**,**

"elem3"**,**

"elem2"**);**

Any code that uses PAssert must link in JUnit and Hamcrest. If you’re using Maven, you can link in Hamcrest by adding the following dependency to your project’s pom.xml file:

**<**dependency**>**

**<**groupId**>**org**.**hamcrest**</**groupId**>**

**<**artifactId**>**hamcrest**-**all**</**artifactId**>**

**<**version**>**1.3**</**version**>**

**<**scope**>**test**</**scope**>**

**</**dependency**>**

For more information on how these classes work, see the [org.apache.beam.sdk.testing](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/index.html?org/apache/beam/sdk/testing/package-summary.html) package documentation.

### An Example Test for a Composite Transform

The following code shows a complete test for a composite transform. The test applies the Count transform to an input PCollection of String elements. The test uses the Create transform to create the input PCollection from a Java List<String>.

**public** **class** **CountTest** **{**

*// Our static input data, which will make up the initial PCollection.*

**static** **final** String**[]** WORDS\_ARRAY **=** **new** String**[]** **{**

"hi"**,** "there"**,** "hi"**,** "hi"**,** "sue"**,** "bob"**,**

"hi"**,** "sue"**,** ""**,** ""**,** "ZOW"**,** "bob"**,** ""**};**

**static** **final** List**<**String**>** WORDS **=** Arrays**.**asList**(**WORDS\_ARRAY**);**

**public** **void** **testCount()** **{**

*// Create a test pipeline.*

Pipeline p **=** TestPipeline**.**create**();**

*// Create an input PCollection.*

PCollection**<**String**>** input **=** p**.**apply**(**Create**.**of**(**WORDS**)).**setCoder**(**StringUtf8Coder**.**of**());**

*// Apply the Count transform under test.*

PCollection**<**KV**<**String**,** Long**>>** output **=**

input**.**apply**(**Count**.<**String**>**perElement**());**

*// Assert on the results.*

PAssert**.**that**(**output**)**

**.**containsInAnyOrder**(**

KV**.**of**(**"hi"**,** 4L**),**

KV**.**of**(**"there"**,** 1L**),**

KV**.**of**(**"sue"**,** 2L**),**

KV**.**of**(**"bob"**,** 2L**),**

KV**.**of**(**""**,** 3L**),**

KV**.**of**(**"ZOW"**,** 1L**));**

*// Run the pipeline.*

p**.**run**();**

**}**

## Testing a Pipeline End-to-End

You can use the test classes in the Beam SDKs (such as TestPipeline and PAssert in the Beam SDK for Java) to test an entire pipeline end-to-end. Typically, to test an entire pipeline, you do the following:

* For every source of input data to your pipeline, create some known static test input data.
* Create some static test output data that matches what you expect in your pipeline’s final output PCollection(s).
* Create a TestPipeline in place of the standard Pipeline.create.
* In place of your pipeline’s Read transform(s), use the Create transform to create one or more PCollections from your static input data.
* Apply your pipeline’s transforms.
* In place of your pipeline’s Write transform(s), use PAssert to verify that the contents of the final PCollections your pipeline produces match the expected values in your static output data.

### Testing the WordCount Pipeline

The following example code shows how one might test the [WordCount example pipeline](https://beam.apache.org/get-started/wordcount-example/). WordCount usually reads lines from a text file for input data; instead, the test creates a Java List<String> containing some text lines and uses a Create transform to create an initial PCollection.

WordCount’s final transform (from the composite transform CountWords) produces a PCollection<String> of formatted word counts suitable for printing. Rather than write that PCollection to an output text file, our test pipeline uses PAssert to verify that the elements of the PCollection match those of a static String array containing our expected output data.

**public** **class** **WordCountTest** **{**

*// Our static input data, which will comprise the initial PCollection.*

**static** **final** String**[]** WORDS\_ARRAY **=** **new** String**[]** **{**

"hi there"**,** "hi"**,** "hi sue bob"**,**

"hi sue"**,** ""**,** "bob hi"**};**

**static** **final** List**<**String**>** WORDS **=** Arrays**.**asList**(**WORDS\_ARRAY**);**

*// Our static output data, which is the expected data that the final PCollection must match.*

**static** **final** String**[]** COUNTS\_ARRAY **=** **new** String**[]** **{**

"hi: 5"**,** "there: 1"**,** "sue: 2"**,** "bob: 2"**};**

*// Example test that tests the pipeline's transforms.*

**public** **void** **testCountWords()** **throws** Exception **{**

Pipeline p **=** TestPipeline**.**create**();**

*// Create a PCollection from the WORDS static input data.*

PCollection**<**String**>** input **=** p**.**apply**(**Create**.**of**(**WORDS**)).**setCoder**(**StringUtf8Coder**.**of**());**

*// Run ALL the pipeline's transforms (in this case, the CountWords composite transform).*

PCollection**<**String**>** output **=** input**.**apply**(new** CountWords**());**

*// Assert that the output PCollection matches the COUNTS\_ARRAY known static output data.*

PAssert**.**that**(**output**).**containsInAnyOrder**(**COUNTS\_ARRAY**);**

*// Run the pipeline.*

p**.**run**();**

**}**

**}**

The run method is asynchronous. If you’d like a blocking execution instead, run your pipeline appending the waitUntilFinish method:

p**.**run**().**waitUntilFinish**();**

# Using the Apache Spark Runner

The Apache Spark Runner can be used to execute Beam pipelines using [Apache Spark](http://spark.apache.org/). The Spark Runner can execute Spark pipelines just like a native Spark application; deploying a self-contained application for local mode, running on Spark’s Standalone RM, or using YARN or Mesos.

The Spark Runner executes Beam pipelines on top of Apache Spark, providing:

* Batch and streaming (and combined) pipelines.
* The same fault-tolerance [guarantees](http://spark.apache.org/docs/1.6.3/streaming-programming-guide.html#fault-tolerance-semantics) as provided by RDDs and DStreams.
* The same [security](http://spark.apache.org/docs/1.6.3/security.html) features Spark provides.
* Built-in metrics reporting using Spark’s metrics system, which reports Beam Aggregators as well.
* Native support for Beam side-inputs via spark’s Broadcast variables.

The [Beam Capability Matrix](https://beam.apache.org/documentation/runners/capability-matrix/) documents the currently supported capabilities of the Spark Runner.

***Note:*** support for the Beam Model in streaming is currently experimental, follow-up in the [*mailing list*](https://beam.apache.org/get-started/support/) for status updates.

## Spark Runner prerequisites and setup

The Spark runner currently supports Spark’s 1.6 branch, and more specifically any version greater than 1.6.0.

You can add a dependency on the latest version of the Spark runner by adding to your pom.xml the following:

**<**dependency**>**

**<**groupId**>**org**.**apache**.**beam**</**groupId**>**

**<**artifactId**>**beam**-**runners**-**spark**</**artifactId**>**

**<**version**>**2.4**.**0**</**version**>**

**</**dependency**>**

### Deploying Spark with your application

In some cases, such as running in local mode/Standalone, your (self-contained) application would be required to pack Spark by explicitly adding the following dependencies in your pom.xml:

**<**dependency**>**

**<**groupId**>**org**.**apache**.**spark**</**groupId**>**

**<**artifactId**>**spark**-**core\_2**.**10**</**artifactId**>**

**<**version**>**$**{**spark**.**version**}</**version**>**

**</**dependency**>**

**<**dependency**>**

**<**groupId**>**org**.**apache**.**spark**</**groupId**>**

**<**artifactId**>**spark**-**streaming\_2**.**10**</**artifactId**>**

**<**version**>**$**{**spark**.**version**}</**version**>**

**</**dependency**>**

And shading the application jar using the maven shade plugin:

**<**plugin**>**

**<**groupId**>**org**.**apache**.**maven**.**plugins**</**groupId**>**

**<**artifactId**>**maven**-**shade**-**plugin**</**artifactId**>**

**<**configuration**>**

**<**createDependencyReducedPom**>false</**createDependencyReducedPom**>**

**<**filters**>**

**<**filter**>**

**<**artifact**>\*:\*</**artifact**>**

**<**excludes**>**

**<**exclude**>**META**-**INF**/\*.**SF**</**exclude**>**

**<**exclude**>**META**-**INF**/\*.**DSA**</**exclude**>**

**<**exclude**>**META**-**INF**/\*.**RSA**</**exclude**>**

**</**excludes**>**

**</**filter**>**

**</**filters**>**

**</**configuration**>**

**<**executions**>**

**<**execution**>**

**<**phase**>**package**</**phase**>**

**<**goals**>**

**<**goal**>**shade**</**goal**>**

**</**goals**>**

**<**configuration**>**

**<**shadedArtifactAttached**>true</**shadedArtifactAttached**>**

**<**shadedClassifierName**>**shaded**</**shadedClassifierName**>**

**<**transformers**>**

**<**transformer

implementation**=**"org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"**/>**

**</**transformers**>**

**</**configuration**>**

**</**execution**>**

**</**executions**>**

**</**plugin**>**

After running mvn package, run ls target and you should see (assuming your artifactId is beam-examples and the version is 1.0.0):

beam-examples-1.0.0-shaded.jar

To run against a Standalone cluster simply run:

spark-submit --class com.beam.examples.BeamPipeline --master spark://HOST:PORT target/beam-examples-1.0.0-shaded.jar --runner=SparkRunner

### Running on a pre-deployed Spark cluster

Deploying your Beam pipeline on a cluster that already has a Spark deployment (Spark classes are available in container classpath) does not require any additional dependencies. For more details on the different deployment modes see: [Standalone](http://spark.apache.org/docs/latest/spark-standalone.html), [YARN](http://spark.apache.org/docs/latest/running-on-yarn.html), or [Mesos](http://spark.apache.org/docs/latest/running-on-mesos.html).

## Pipeline options for the Spark Runner

When executing your pipeline with the Spark Runner, you should consider the following pipeline options.

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | **Default Value** |
| runner | The pipeline runner to use. This option allows you to determine the pipeline runner at runtime. | Set to SparkRunnerto run using Spark. |
| sparkMaster | The url of the Spark Master. This is the equivalent of setting SparkConf#setMaster(String) and can either be local[x] to run local with x cores, spark://host:port to connect to a Spark Standalone cluster, mesos://host:port to connect to a Mesos cluster, or yarn to connect to a yarn cluster. | local[4] |
| storageLevel | The StorageLevel to use when caching RDDs in batch pipelines. The Spark Runner automatically caches RDDs that are evaluated repeatedly. This is a batch-only property as streaming pipelines in Beam are stateful, which requires Spark DStream's StorageLevel to be MEMORY\_ONLY. | MEMORY\_ONLY |
| batchIntervalMillis | The StreamingContext's batchDuration - setting Spark's batch interval. | 1000 |
| enableSparkMetricSinks | Enable reporting metrics to Spark's metrics Sinks. | true |

## Additional notes

### Using spark-submit

When submitting a Spark application to cluster, it is common (and recommended) to use the spark-submit script that is provided with the spark installation. The PipelineOptions described above are not to replace spark-submit, but to complement it. Passing any of the above mentioned options could be done as one of the application-arguments, and setting --master takes precedence. For more on how to generally use spark-submit checkout Spark [documentation](http://spark.apache.org/docs/1.6.3/submitting-applications.html#launching-applications-with-spark-submit).

### Monitoring your job

You can monitor a running Spark job using the Spark [Web Interfaces](http://spark.apache.org/docs/1.6.3/monitoring.html#web-interfaces). By default, this is available at port 4040 on the driver node. If you run Spark on your local machine that would be http://localhost:4040. Spark also has a history server to [view after the fact](http://spark.apache.org/docs/1.6.3/monitoring.html#viewing-after-the-fact). Metrics are also available via [REST API](http://spark.apache.org/docs/1.6.3/monitoring.html#rest-api). Spark provides a [metrics system](http://spark.apache.org/docs/1.6.3/monitoring.html#metrics) that allows reporting Spark metrics to a variety of Sinks. The Spark runner reports user-defined Beam Aggregators using this same metrics system and currently supports GraphiteSink and CSVSink, and providing support for additional Sinks supported by Spark is easy and straight-forward.

### Streaming Execution

If your pipeline uses an UnboundedSource the Spark Runner will automatically set streaming mode. Forcing streaming mode is mostly used for testing and is not recommended.

### Using a provided SparkContext and StreamingListeners

If you would like to execute your Spark job with a provided SparkContext, such as when using the [spark-jobserver](https://github.com/spark-jobserver/spark-jobserver), or use StreamingListeners, you can’t use SparkPipelineOptions (the context or a listener cannot be passed as a command-line argument anyway). Instead, you should use SparkContextOptions which can only be used programmatically and is not a common PipelineOptions implementation.

# Using the Apache Flink Runner

**Adapt for:**

* Java SDK

* Python SDK

The Apache Flink Runner can be used to execute Beam pipelines using [Apache Flink](https://flink.apache.org/). When using the Flink Runner you will create a jar file containing your job that can be executed on a regular Flink cluster. It’s also possible to execute a Beam pipeline using Flink’s local execution mode without setting up a cluster. This is helpful for development and debugging of your pipeline.

The Flink Runner and Flink are suitable for large scale, continuous jobs, and provide:

* A streaming-first runtime that supports both batch processing and data streaming programs
* A runtime that supports very high throughput and low event latency at the same time
* Fault-tolerance with exactly-once processing guarantees
* Natural back-pressure in streaming programs
* Custom memory management for efficient and robust switching between in-memory and out-of-core data processing algorithms
* Integration with YARN and other components of the Apache Hadoop ecosystem

The [Beam Capability Matrix](https://beam.apache.org/documentation/runners/capability-matrix/) documents the supported capabilities of the Flink Runner.

## Flink Runner prerequisites and setup

If you want to use the local execution mode with the Flink runner to don’t have to complete any setup.

To use the Flink Runner for executing on a cluster, you have to setup a Flink cluster by following the Flink [setup quickstart](https://ci.apache.org/projects/flink/flink-docs-release-1.1/quickstart/setup_quickstart.html).

To find out which version of Flink you need you can run this command to check the version of the Flink dependency that your project is using:

$ mvn dependency:tree -Pflink-runner |grep flink

...

[INFO] | +- org.apache.flink:flink-streaming-java\_2.10:jar:1.2.1:runtime

...

Here, we would need Flink 1.2.1. Please also note the Scala version in the dependency name. In this case we need to make sure to use a Flink cluster with Scala version 2.10.

For more information, the [Flink Documentation](https://ci.apache.org/projects/flink/flink-docs-release-1.1/) can be helpful.

### Specify your dependency

When using Java, you must specify your dependency on the Flink Runner in your pom.xml.

**<**dependency**>**

**<**groupId**>**org**.**apache**.**beam**</**groupId**>**

**<**artifactId**>**beam**-**runners**-**flink\_2**.**10**</**artifactId**>**

**<**version**>**2.4**.**0**</**version**>**

**<**scope**>**runtime**</**scope**>**

**</**dependency**>**

This section is not applicable to the Beam SDK for Python.

## Executing a pipeline on a Flink cluster

For executing a pipeline on a Flink cluster you need to package your program along will all dependencies in a so-called fat jar. How you do this depends on your build system but if you follow along the [Beam Quickstart](https://beam.apache.org/get-started/quickstart/) this is the command that you have to run:

$ mvn package -Pflink-runner

The Beam Quickstart Maven project is setup to use the Maven Shade plugin to create a fat jar and the -Pflink-runner argument makes sure to include the dependency on the Flink Runner.

For actually running the pipeline you would use this command

$ mvn exec:java -Dexec.mainClass=org.apache.beam.examples.WordCount \

-Pflink-runner \

-Dexec.args="--runner=FlinkRunner \

--inputFile=/path/to/pom.xml \

--output=/path/to/counts \

--flinkMaster=<flink master url> \

--filesToStage=target/word-count-beam--bundled-0.1.jar"

If you have a Flink JobManager running on your local machine you can give localhost:6123 for flinkMaster.

## Pipeline options for the Flink Runner

When executing your pipeline with the Flink Runner, you can set these pipeline options.

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | **Default Value** |
| runner | The pipeline runner to use. This option allows you to determine the pipeline runner at runtime. | Set to FlinkRunnerto run using Flink. |
| streaming | Whether streaming mode is enabled or disabled; true if enabled. Set to true if running pipelines with unbounded PCollections. | false |
| flinkMaster | The url of the Flink JobManager on which to execute pipelines. This can either be the address of a cluster JobManager, in the form "host:port" or one of the special Strings "[local]" or "[auto]". "[local]" will start a local Flink Cluster in the JVM while "[auto]" will let the system decide where to execute the pipeline based on the environment. | [auto] |
| filesToStage | Jar Files to send to all workers and put on the classpath. Here you have to put the fat jar that contains your program along with all dependencies. | empty |
| parallelism | The degree of parallelism to be used when distributing operations onto workers. | 1 |
| checkpointingInterval | The interval between consecutive checkpoints (i.e. snapshots of the current pipeline state used for fault tolerance). | -1L, i.e. disabled |
| numberOfExecutionRetries | Sets the number of times that failed tasks are re-executed. A value of 0 effectively disables fault tolerance. A value of -1 indicates that the system default value (as defined in the configuration) should be used. | -1 |
| executionRetryDelay | Sets the delay between executions. A value of -1 indicates that the default value should be used. | -1 |
| stateBackend | Sets the state backend to use in streaming mode. The default is to read this setting from the Flink config. | empty, i.e. read from Flink config |

See the reference documentation for the [FlinkPipelineOptions](https://beam.apache.org/documentation/sdks/javadoc/2.4.0/index.html?org/apache/beam/runners/flink/FlinkPipelineOptions.html)[PipelineOptions](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/options/pipeline_options.py) interface (and its subinterfaces) for the complete list of pipeline configuration options.

## Additional information and caveats

### Monitoring your job

You can monitor a running Flink job using the Flink JobManager Dashboard. By default, this is available at port 8081 of the JobManager node. If you have a Flink installation on your local machine that would be http://localhost:8081.

### Streaming Execution

If your pipeline uses an unbounded data source or sink, the Flink Runner will automatically switch to streaming mode. You can enforce streaming mode by using the streaming setting mentioned above.

# Submitting Applications

The spark-submit script in Spark’s bin directory is used to launch applications on a cluster. It can use all of Spark’s supported [cluster managers](http://spark.apache.org/docs/2.2.1/cluster-overview.html#cluster-manager-types)through a uniform interface so you don’t have to configure your application specially for each one.

# Bundling Your Application’s Dependencies

If your code depends on other projects, you will need to package them alongside your application in order to distribute the code to a Spark cluster. To do this, create an assembly jar (or “uber” jar) containing your code and its dependencies. Both [sbt](https://github.com/sbt/sbt-assembly) and [Maven](http://maven.apache.org/plugins/maven-shade-plugin/) have assembly plugins. When creating assembly jars, list Spark and Hadoop as provided dependencies; these need not be bundled since they are provided by the cluster manager at runtime. Once you have an assembled jar you can call the bin/spark-submit script as shown here while passing your jar.

For Python, you can use the --py-files argument of spark-submit to add .py, .zip or .egg files to be distributed with your application. If you depend on multiple Python files we recommend packaging them into a .zip or .egg.

# Launching Applications with spark-submit

Once a user application is bundled, it can be launched using the bin/spark-submit script. This script takes care of setting up the classpath with Spark and its dependencies, and can support different cluster managers and deploy modes that Spark supports:

./bin/spark-submit **\**

--class <main-class> **\**

--master <master-url> **\**

--deploy-mode <deploy-mode> **\**

--conf <key>=<value> **\**

... *# other options*

<application-jar> **\**

[application-arguments]

Some of the commonly used options are:

* --class: The entry point for your application (e.g. org.apache.spark.examples.SparkPi)
* --master: The [master URL](http://spark.apache.org/docs/2.2.1/submitting-applications.html#master-urls) for the cluster (e.g. spark://23.195.26.187:7077)
* --deploy-mode: Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client) **†**
* --conf: Arbitrary Spark configuration property in key=value format. For values that contain spaces wrap “key=value” in quotes (as shown).
* application-jar: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes.
* application-arguments: Arguments passed to the main method of your main class, if any

**†** A common deployment strategy is to submit your application from a gateway machine that is physically co-located with your worker machines (e.g. Master node in a standalone EC2 cluster). In this setup, client mode is appropriate. In client mode, the driver is launched directly within the spark-submit process which acts as a client to the cluster. The input and output of the application is attached to the console. Thus, this mode is especially suitable for applications that involve the REPL (e.g. Spark shell).

Alternatively, if your application is submitted from a machine far from the worker machines (e.g. locally on your laptop), it is common to use clustermode to minimize network latency between the drivers and the executors. Currently, standalone mode does not support cluster mode for Python applications.

For Python applications, simply pass a .py file in the place of <application-jar> instead of a JAR, and add Python .zip, .egg or .py files to the search path with --py-files.

There are a few options available that are specific to the [cluster manager](http://spark.apache.org/docs/2.2.1/cluster-overview.html#cluster-manager-types) that is being used. For example, with a [Spark standalone cluster](http://spark.apache.org/docs/2.2.1/spark-standalone.html) with cluster deploy mode, you can also specify --supervise to make sure that the driver is automatically restarted if it fails with non-zero exit code. To enumerate all such options available to spark-submit, run it with --help. Here are a few examples of common options:

*# Run application locally on 8 cores*

./bin/spark-submit **\**

--class org.apache.spark.examples.SparkPi **\**

--master local[8] **\**

/path/to/examples.jar **\**

100

*# Run on a Spark standalone cluster in client deploy mode*

./bin/spark-submit **\**

--class org.apache.spark.examples.SparkPi **\**

--master spark://207.184.161.138:7077 **\**

--executor-memory 20G **\**

--total-executor-cores 100 **\**

/path/to/examples.jar **\**

1000

*# Run on a Spark standalone cluster in cluster deploy mode with supervise*

./bin/spark-submit **\**

--class org.apache.spark.examples.SparkPi **\**

--master spark://207.184.161.138:7077 **\**

--deploy-mode cluster **\**

--supervise **\**

--executor-memory 20G **\**

--total-executor-cores 100 **\**

/path/to/examples.jar **\**

1000

*# Run on a YARN cluster*

export HADOOP\_CONF\_DIR=XXX

./bin/spark-submit **\**

--class org.apache.spark.examples.SparkPi **\**

--master yarn **\**

--deploy-mode cluster **\**  *# can be client for client mode*

--executor-memory 20G **\**

--num-executors 50 **\**

/path/to/examples.jar **\**

1000

*# Run a Python application on a Spark standalone cluster*

./bin/spark-submit **\**

--master spark://207.184.161.138:7077 **\**

examples/src/main/python/pi.py **\**

1000

*# Run on a Mesos cluster in cluster deploy mode with supervise*

./bin/spark-submit **\**

--class org.apache.spark.examples.SparkPi **\**

--master mesos://207.184.161.138:7077 **\**

--deploy-mode cluster **\**

--supervise **\**

--executor-memory 20G **\**

--total-executor-cores 100 **\**

http://path/to/examples.jar **\**

1000

# Master URLs

The master URL passed to Spark can be in one of the following formats:

|  |  |
| --- | --- |
| **Master URL** | **Meaning** |
| local | Run Spark locally with one worker thread (i.e. no parallelism at all). |
| local[K] | Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine). |
| local[K,F] | Run Spark locally with K worker threads and F maxFailures (see [spark.task.maxFailures](http://spark.apache.org/docs/2.2.1/configuration.html#scheduling) for an explanation of this variable) |
| local[\*] | Run Spark locally with as many worker threads as logical cores on your machine. |
| local[\*,F] | Run Spark locally with as many worker threads as logical cores on your machine and F maxFailures. |
| spark://HOST:PORT | Connect to the given [Spark standalone cluster](http://spark.apache.org/docs/2.2.1/spark-standalone.html) master. The port must be whichever one your master is configured to use, which is 7077 by default. |
| spark://HOST1:PORT1,HOST2:PORT2 | Connect to the given [Spark standalone cluster with standby masters with Zookeeper](http://spark.apache.org/docs/2.2.1/spark-standalone.html#standby-masters-with-zookeeper). The list must have all the master hosts in the high availability cluster set up with Zookeeper. The port must be whichever each master is configured to use, which is 7077 by default. |
| mesos://HOST:PORT | Connect to the given [Mesos](http://spark.apache.org/docs/2.2.1/running-on-mesos.html) cluster. The port must be whichever one your is configured to use, which is 5050 by default. Or, for a Mesos cluster using ZooKeeper, use mesos://zk://.... To submit with --deploy-mode cluster, the HOST:PORT should be configured to connect to the [MesosClusterDispatcher](http://spark.apache.org/docs/2.2.1/running-on-mesos.html#cluster-mode). |
| yarn | Connect to a [YARN](http://spark.apache.org/docs/2.2.1/running-on-yarn.html)cluster in client or cluster mode depending on the value of --deploy-mode. The cluster location will be found based on the HADOOP\_CONF\_DIR or YARN\_CONF\_DIR variable. |

# Loading Configuration from a File

The spark-submit script can load default [Spark configuration values](http://spark.apache.org/docs/2.2.1/configuration.html) from a properties file and pass them on to your application. By default it will read options from conf/spark-defaults.conf in the Spark directory. For more detail, see the section on [loading default configurations](http://spark.apache.org/docs/2.2.1/configuration.html#loading-default-configurations).

Loading default Spark configurations this way can obviate the need for certain flags to spark-submit. For instance, if the spark.master property is set, you can safely omit the --master flag from spark-submit. In general, configuration values explicitly set on a SparkConf take the highest precedence, then flags passed to spark-submit, then values in the defaults file.

If you are ever unclear where configuration options are coming from, you can print out fine-grained debugging information by running spark-submitwith the --verbose option.

# Advanced Dependency Management

When using spark-submit, the application jar along with any jars included with the --jars option will be automatically transferred to the cluster. URLs supplied after --jars must be separated by commas. That list is included on the driver and executor classpaths. Directory expansion does not work with --jars.

Spark uses the following URL scheme to allow different strategies for disseminating jars:

* **file:** - Absolute paths and file:/ URIs are served by the driver’s HTTP file server, and every executor pulls the file from the driver HTTP server.
* **hdfs:**, **http:**, **https:**, **ftp:** - these pull down files and JARs from the URI as expected
* **local:** - a URI starting with local:/ is expected to exist as a local file on each worker node. This means that no network IO will be incurred, and works well for large files/JARs that are pushed to each worker, or shared via NFS, GlusterFS, etc.

Note that JARs and files are copied to the working directory for each SparkContext on the executor nodes. This can use up a significant amount of space over time and will need to be cleaned up. With YARN, cleanup is handled automatically, and with Spark standalone, automatic cleanup can be configured with the spark.worker.cleanup.appDataTtl property.

Users may also include any other dependencies by supplying a comma-delimited list of Maven coordinates with --packages. All transitive dependencies will be handled when using this command. Additional repositories (or resolvers in SBT) can be added in a comma-delimited fashion with the flag --repositories. (Note that credentials for password-protected repositories can be supplied in some cases in the repository URI, such as in https://user:password@host/.... Be careful when supplying credentials this way.) These commands can be used with pyspark, spark-shell, and spark-submit to include Spark Packages.

For Python, the equivalent --py-files option can be used to distribute .egg, .zip and .py libraries to executors.

# More Information

Once you have deployed your application, the [cluster mode overview](http://spark.apache.org/docs/2.2.1/cluster-overview.html) describes the components I

# Beam Capability Matrix

Apache Beam provides a portable API layer for building sophisticated data-parallel processing pipelines that may be executed across a diversity of execution engines, or *runners*. The core concepts of this layer are based upon the Beam Model (formerly referred to as the [Dataflow Model](http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf)), and implemented to varying degrees in each Beam runner. To help clarify the capabilities of individual runners, we’ve created the capability matrix below.

Individual capabilities have been grouped by their corresponding ***What*** / ***Where*** / ***When*** / ***How*** question:

* ***What*** results are being calculated?
* ***Where*** in event time?
* ***When*** in processing time?
* ***How*** do refinements of results relate?

For more details on the ***What*** / ***Where*** / ***When*** / ***How*** breakdown of concepts, we recommend reading through the [Streaming 102](http://oreilly.com/ideas/the-world-beyond-batch-streaming-102) post on O’Reilly Radar.

Note that in the future, we intend to add additional tables beyond the current set, for things like runtime characterstics (e.g. at-least-once vs exactly-once), performance, etc.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| (expand details)  **What is being computed?** | | | | |
|  | **Beam Model** | **Google Cloud Dataflow** | **Apache Flink** | **Apache Spark** | **Apache Apex** | **Apache Gearpump** | **Apache Hadoop MapReduce** | **JStorm** | **IBM Streams** | **Apache Samza** |
| **ParDo** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **GroupByKey** | **✓** | **✓** | **✓** | **~** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Flatten** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Combine** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Composite Transforms** | **✓** | **~** | **~** | **~** | **~** | **~** | **✓** | **✓** | **~** | **~** |
| **Side Inputs** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Source API** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **~** | **✓** | **✓** | **✓** |
| **Splittable DoFn (SDF)** | **~** | **✓** | **✓** | **~** | **~** | **~** | **✕** | **✕** | **✕** | **~** |
| **Metrics** | **~** | **~** | **~** | **~** | **✕** | **✕** | **~** | **~** | **~** | **~** |
| **Stateful Processing** | **✓** | **~** | **~** | **✕** | **~** | **✕** | **~** | **~** | **~** | **~** |
|  | | | | |  |  |  |  |  |  |
| (expand details)  **Where in event time?** | | | | |  |  |  |  |  |  |
|  | **Beam Model** | **Google Cloud Dataflow** | **Apache Flink** | **Apache Spark** | **Apache Apex** | **Apache Gearpump** | **Apache Hadoop MapReduce** | **JStorm** | **IBM Streams** | **Apache Samza** |
| **Global windows** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Fixed windows** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Sliding windows** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Session windows** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Custom windows** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Custom merging windows** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
| **Timestamp control** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** |
|  | | | | |  |  |  |  |  |  |
| (expand details)  **When in processing time?** | | | | |  |  |  |  |  |  |
|  | **Beam Model** | **Google Cloud Dataflow** | **Apache Flink** | **Apache Spark** | **Apache Apex** | **Apache Gearpump** | **Apache Hadoop MapReduce** | **JStorm** | **IBM Streams** | **Apache Samza** |
| **Configurable triggering** | **✓** | **✓** | **✓** | **✕** | **✓** | **✕** | **✕** | **✓** | **✓** | **✓** |
| **Event-time triggers** | **✓** | **✓** | **✓** | **✕** | **✓** | **✓** | **✕** | **✓** | **✓** | **✓** |
| **Processing-time triggers** | **✓** | **✓** | **✓** | **✓** | **✓** | **✕** | **✕** | **✓** | **✓** | **✓** |
| **Count triggers** | **✓** | **✓** | **✓** | **✕** | **✓** | **✕** | **✕** | **✓** | **✓** | **✓** |
| **[Meta]data driven triggers** | **✕ (**[**BEAM-101**](https://issues.apache.org/jira/browse/BEAM-101)**)** | **✕** | **✕** | **✕** | **✕** | **✕** | **✕** | **✕** | **✕** | **✕** |
| **Composite triggers** | **✓** | **✓** | **✓** | **✕** | **✓** | **✕** | **✕** | **✓** | **✓** | **✓** |
| **Allowed lateness** | **✓** | **✓** | **✓** | **✕** | **✓** | **✓** | **✕** | **✓** | **✓** | **✓** |
| **Timers** | **✓** | **~** | **~** | **✕** | **✕** | **✕** | **✕** | **~** | **~** | **✕** |
|  | | | | |  |  |  |  |  |  |
| (expand details)  **How do refinements relate?** | | | | |  |  |  |  |  |  |
|  | **Beam Model** | **Google Cloud Dataflow** | **Apache Flink** | **Apache Spark** | **Apache Apex** | **Apache Gearpump** | **Apache Hadoop MapReduce** | **JStorm** | **IBM Streams** | **Apache Samza** |
| **Discarding** | **✓** | **✓** | **✓** | **✓** | **✓** | **✓** | **✕** | **✓** | **✓** | **✓** |
| **Accumulating** | **✓** | **✓** | **✓** | **✕** | **✓** | **✕** | **✕** | **✓** | **✓** | **✓** |
| **Accumulating & Retracting** | **✕ (**[**BEAM-91**](https://issues.apache.org/jira/browse/BEAM-91)**)** | **✕** | **✕** | **✕** | **✕** | **✕** | **✕** | **✕** | **✕** | **✕** |
|  | | | | |  |  |  |  |  |  |

# Using the Apache Flink Runner

The old Flink Runner will eventually be replaced by the Portable Runner which enables to run pipelines in other languages than Java. Please see the [Portability page](https://beam.apache.org/contribute/portability/) for the latest state.

**Adapt for:**

* Java SDK

* Python SDK

The Apache Flink Runner can be used to execute Beam pipelines using [Apache Flink](https://flink.apache.org/). When using the Flink Runner you will create a jar file containing your job that can be executed on a regular Flink cluster. It’s also possible to execute a Beam pipeline using Flink’s local execution mode without setting up a cluster. This is helpful for development and debugging of your pipeline.

The Flink Runner and Flink are suitable for large scale, continuous jobs, and provide:

* A streaming-first runtime that supports both batch processing and data streaming programs
* A runtime that supports very high throughput and low event latency at the same time
* Fault-tolerance with exactly-once processing guarantees
* Natural back-pressure in streaming programs
* Custom memory management for efficient and robust switching between in-memory and out-of-core data processing algorithms
* Integration with YARN and other components of the Apache Hadoop ecosystem

The [Beam Capability Matrix](https://beam.apache.org/documentation/runners/capability-matrix/) documents the supported capabilities of the Flink Runner.

## Flink Runner prerequisites and setup

If you want to use the local execution mode with the Flink runner to don’t have to complete any setup.

To use the Flink Runner for executing on a cluster, you have to setup a Flink cluster by following the Flink [setup quickstart](https://ci.apache.org/projects/flink/flink-docs-stable/quickstart/setup_quickstart.html).

### Version Compatibility

The Flink cluster version has to match the version used by the FlinkRunner. To find out which version of Flink please see the table below:

|  |  |
| --- | --- |
| **Beam version** | **Flink version** |
| 2.7.0, 2.6.0 | 1.5.x |
| 2.5.0, 2.4.0, 2.3.0 | 1.4.x |
| 2.2.0 | 1.3.x with Scala 2.10 |
| 2.2.0, 2.1.x | 1.3.x with Scala 2.10 |
| 2.0.0 | 1.2.x with Scala 2.10 |

For retrieving the right version, see the [Flink downloads page](https://flink.apache.org/downloads.html).

For more information, the [Flink Documentation](https://ci.apache.org/projects/flink/flink-docs-stable/) can be helpful.

### Specify your dependency

When using Java, you must specify your dependency on the Flink Runner in your pom.xml.

**<**dependency**>**

**<**groupId**>**org**.**apache**.**beam**</**groupId**>**

**<**artifactId**>**beam**-**runners**-**flink\_2**.**11**</**artifactId**>**

**<**version**>**2.7**.**0**</**version**>**

**</**dependency**>**

This section is not applicable to the Beam SDK for Python.

## Executing a pipeline on a Flink cluster

For executing a pipeline on a Flink cluster you need to package your program along will all dependencies in a so-called fat jar. How you do this depends on your build system but if you follow along the [Beam Quickstart](https://beam.apache.org/get-started/quickstart/) this is the command that you have to run:

$ mvn package -Pflink-runner

The Beam Quickstart Maven project is setup to use the Maven Shade plugin to create a fat jar and the -Pflink-runner argument makes sure to include the dependency on the Flink Runner.

For actually running the pipeline you would use this command

$ mvn exec:java -Dexec.mainClass=org.apache.beam.examples.WordCount \

-Pflink-runner \

-Dexec.args="--runner=FlinkRunner \

--inputFile=/path/to/pom.xml \

--output=/path/to/counts \

--flinkMaster=<flink master url> \

--filesToStage=target/word-count-beam--bundled-0.1.jar"

If you have a Flink JobManager running on your local machine you can give localhost:8081 for flinkMaster.

## Pipeline options for the Flink Runner

When executing your pipeline with the Flink Runner, you can set these pipeline options.

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | **Default Value** |
| runner | The pipeline runner to use. This option allows you to determine the pipeline runner at runtime. | Set to FlinkRunnerto run using Flink. |
| streaming | Whether streaming mode is enabled or disabled; true if enabled. Set to true if running pipelines with unbounded PCollections. | false |
| flinkMaster | The url of the Flink JobManager on which to execute pipelines. This can either be the address of a cluster JobManager, in the form "host:port" or one of the special Strings "[local]" or "[auto]". "[local]" will start a local Flink Cluster in the JVM while "[auto]" will let the system decide where to execute the pipeline based on the environment. | [auto] |
| filesToStage | Jar Files to send to all workers and put on the classpath. Here you have to put the fat jar that contains your program along with all dependencies. | empty |
| parallelism | The degree of parallelism to be used when distributing operations onto workers. | 1 |
| checkpointingInterval | The interval between consecutive checkpoints (i.e. snapshots of the current pipeline state used for fault tolerance). | -1L, i.e. disabled |
| numberOfExecutionRetries | Sets the number of times that failed tasks are re-executed. A value of 0 effectively disables fault tolerance. A value of -1 indicates that the system default value (as defined in the configuration) should be used. | -1 |
| executionRetryDelay | Sets the delay between executions. A value of -1 indicates that the default value should be used. | -1 |
| stateBackend | Sets the state backend to use in streaming mode. The default is to read this setting from the Flink config. | empty, i.e. read from Flink config |

See the reference documentation for the [FlinkPipelineOptions](https://beam.apache.org/releases/javadoc/2.7.0/index.html?org/apache/beam/runners/flink/FlinkPipelineOptions.html)[PipelineOptions](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/options/pipeline_options.py) interface (and its subinterfaces) for the complete list of pipeline configuration options.

## Additional information and caveats

### Monitoring your job

You can monitor a running Flink job using the Flink JobManager Dashboard. By default, this is available at port 8081 of the JobManager node. If you have a Flink installation on your local machine that would be http://localhost:8081.

### Streaming Execution

If your pipeline uses an unbounded data source or sink, the Flink Runner will automatically switch to streaming mode. You can enforce streaming mode by using the streaming setting mentioned above.