

## 30 | Apache Beam实战冲刺：Beam如何run everywhere?

2019-07-01 蔡元楠 来自北京

《大规模数据处理实战》



你好，我是蔡元楠。

今天我要与你分享的主题是 “Apache Beam 实战冲刺：Beam 如何 run everywhere” 。

你可能已经注意到，自第 26 讲到第 29 讲，从 Pipeline 的输入输出，到 Pipeline 的设计，再到 Pipeline 的测试，Beam Pipeline 的概念一直贯穿着文章脉络。那么这一讲，我们一起来看看一个完整的 Beam Pipeline 究竟是如何编写的。

### Beam Pipeline


一个 Pipeline，或者说是一个数据处理任务，基本上都会包含以下三个步骤：

1. 读取输入数据到 PCollection。
2. 对读进来的 PCollection 做某些操作（也就是 Transform），得到另一个 PCollection。

### 3. 输出你的结果 PCollection。

这么说，看起来很简单，但你可能会有些迷惑：这些步骤具体该怎么做呢？其实这些步骤具体到 Pipeline 的实际编程中，就会包含以下这些代码模块：

#### Java

 复制代码

```
1 // Start by defining the options for the pipeline.
2 PipelineOptions options = PipelineOptionsFactory.create();
3
4 // Then create the pipeline.
5 Pipeline pipeline = Pipeline.create(options);
6
7 PCollection<String> lines = pipeline.apply(
8     "ReadLines", TextIO.read().from("gs://some/inputData.txt"));
9
10 PCollection<String> filteredLines = lines.apply(new FilterLines());
11
12 filteredLines.apply("WriteMyFile", TextIO.write().to("gs://some/outputData.txt"))
13
14 pipeline.run().waitUntilFinish();
```

从上面的代码例子中你可以看到，第一行和第二行代码是创建 Pipeline 实例。任何一个 Beam 程序都需要先创建一个 Pipeline 的实例。Pipeline 实例就是用来表达 Pipeline 类型的对象。这里你需要注意，一个二进制程序可以动态包含多个 Pipeline 实例。

还是以之前的美团外卖电动车处理的例子来做说明吧。

比如，我们的程序可以动态判断是否存在第三方的电动车图片，只有当有需要处理图片时，我们才去创建一个 Pipeline 实例处理。我们也可以动态判断是否存在需要转换图片格式，有需要时，我们再去创建第二个 Pipeline 实例。这时候你的二进制程序，可能包含 0 个、1 个，或者是 2 个 Pipeline 实例。每一个实例都是独立的，它封装了你要进行操作的数据，和你要进行的操作 Transform。

Pipeline 实例的创建是使用 `Pipeline.create(options)` 这个方法。其中 `options` 是传递进去的参数，`options` 是一个 `PipelineOptions` 这个类的实例。我们会在后半部分展开 `PipelineOptions` 的丰富变化。

第三行代码，我们用 `TextIO.read()` 这个 Transform 读取了来自外部文本文件的内容，把所有的行表示为一个 `PCollection`。

第四行代码，用 `lines.apply(new FilterLines())` 对读进来的 `PCollection` 进行了过滤操作。

第五行代码 `filteredLines.apply( "WriteMyFile" , TextIO.write().to( "gs://some/outputData.txt" ))`，表示把最终的 `PCollection` 结果输出到另一个文本文件。

程序运行到第五行的时候，是不是我们的数据处理任务就完成了呢？并不是。

记得我们在第 24 讲、第 25 讲中提过，Beam 是延迟运行的。程序跑到第五行的时候，只是构建了 Beam 所需要的数据处理 DAG 用来优化和分配计算资源，真正的运算完全没有发生。

所以，我们需要最后一行 `pipeline.run().waitUntilFinish()`，这才是数据真正开始被处理的语句。

这时候运行我们的代码，是不是就大功告成呢？别急，我们还没有处理好程序在哪里运行的问题。你一定会好奇，我们的程序究竟在哪里运行，不是说好了分布式数据处理吗？

在上一讲《如何测试 Beam Pipeline》中我们学会了在单元测试环境中运行 Beam Pipeline。就如同下面的代码。和上文的代码类似，我们把 `Pipeline.create(options)` 替换成了 `TestPipeline.create()`。

Java

 复制代码

```
1 Pipeline p = TestPipeline.create();  
2
```

```
3 PCollection<String> input = p.apply(Create.of(WORDS)).setCoder(StringUtf8Coder.of
4
5 PCollection<String> output = input.apply(new CountWords());
6
7 PAssert.that(output).containsInAnyOrder(COUNTS_ARRAY);
8
9 p.run();
```


TestPipeline 是 Beam Pipeline 中特殊的一种，让你能够在单机上运行小规模的数据集。之前我们在分析 Beam 的设计理念时提到过，Beam 想要把应用层的数据处理业务逻辑和底层的运算引擎分离开来。

现如今 Beam 可以做到让你的 Pipeline 代码无需修改，就可以在本地、Spark、Flink，或者在 Google Cloud DataFlow 上运行。这些都是通过 Pipeline.create(options) 这行代码中传递的 PipelineOptions 实现的。

在实战中，我们应用到的所有 option 其实都是实现了 PipelineOptions 这个接口。

举个例子，如果我们希望将数据流水线放在 Spark 这个底层数据引擎运行的时候，我们便可以使用 SparkPipelineOptions。如果我们想把数据流水线放在 Flink 上运行，就可以使用 FlinkPipelineOptions。而这些都是 extends 了 PipelineOptions 的接口，示例如下：

## Java


 复制代码

```
1 options = PipelineOptionsFactory.as(SparkPipelineOptions.class);
2 Pipeline pipeline = Pipeline.create(options);
```

通常一个 PipelineOption 是用 PipelineOptionsFactory 这个工厂类来创建的，它提供了两个静态工厂方法给我们去创建，分别是 PipelineOptionsFactory.as(Class) 和 PipelineOptionsFactory.create()。像上面的示例代码就是用 PipelineOptionsFactory.as(Class) 这个静态工厂方法来创建的。

当然了，更加常见的创建方法是从命令行中读取参数来创建 PipelineOption，使用的是 PipelineOptionsFactory#fromArgs(String[]) 这个方法，例如：

Java

 复制代码

```
1 public static void main(String[] args) {  
2     PipelineOptions options = PipelineOptionsFactory.fromArgs(args).create();  
3     Pipeline p = Pipeline.create(options);  
4 }
```

下面我们来看看不同的运行模式的具体使用方法。

## 直接运行模式

我们先从直接运行模式开始讲。这是我们在本地进行测试，或者调试时倾向使用的模式。在直接运行模式的时候，Beam 会在单机上用多线程来模拟分布式的并行处理。

使用 Java Beam SDK 时，我们要给程序添加 Direct Runner 的依赖关系。在下面这个 maven 依赖关系定义文件中，我们指定了 beam-runners-direct-java 这样一个依赖关系。

 复制代码

```
1 pom.xml  
2 <dependency>  
3     <groupId>org.apache.beam</groupId>  
4     <artifactId>beam-runners-direct-java</artifactId>  
5     <version>2.13.0</version>  
6     <scope>runtime</scope>  
7 </dependency>
```

一般我们会把 runner 通过命令行指令传递进程序。就需要使用 PipelineOptionsFactory.fromArgs(args) 来创建 PipelineOptions。PipelineOptionsFactory.fromArgs() 是一个工厂方法，能够根据命令行参数选择生成不同的 PipelineOptions 子类。



[复制代码](#)

```
1 PipelineOptions options =  
2     PipelineOptionsFactory.fromArgs(args).create();
```

在实验程序中也可以强行使用 Direct Runner。比如：

[复制代码](#)

```
1 PipelineOptions options = PipelineOptionsFactory.create();  
2 options.setRunner(DirectRunner.class);  
3 // 或者这样  
4 options = PipelineOptionsFactory.as(DirectRunner.class);  
5 Pipeline pipeline = Pipeline.create(options);
```

如果是在命令行中指定 Runner 的话，那么在调用这个程序时候，需要指定这样一个参数—runner=DirectRunner。比如：

[复制代码](#)

```
1 mvn compile exec:java -Dexec.mainClass=YourMainClass \  
2     -Dexec.args="--runner=DirectRunner" -Pdirect-runner
```

## Spark 运行模式

如果我们希望将数据流水线放在 Spark 这个底层数据引擎运行的时候，我们便可以使用 Spark Runner。Spark Runner 执行 Beam 程序时，能够像原生的 Spark 程序一样。比如，在 Spark 本地模式部署应用，跑在 Spark 的 RM 上，或者用 YARN。


Spark Runner 为在 Apache Spark 上运行 Beam Pipeline 提供了以下功能：

1. Batch 和 streaming 的数据流水线；
2. 和原生 RDD 和 DStream 一样的容错保证；
3. 和原生 Spark 同样的安全性能；
4. 可以用 Spark 的数据回报系统；

## 5. 使用 Spark Broadcast 实现的 Beam side-input。


目前使用 Spark Runner 必须使用 Spark 2.2 版本以上。

这里，我们先添加 beam-runners-spark 的依赖关系。

 复制代码

```
1 <dependency>
2   <groupId>org.apache.beam</groupId>
3   <artifactId>beam-runners-spark</artifactId>
4   <version>2.13.0</version>
5 </dependency>
6 <dependency>
7   <groupId>org.apache.spark</groupId>
8   <artifactId>spark-core_2.10</artifactId>
9   <version>${spark.version}</version>
10 </dependency>
11 <dependency>
12   <groupId>org.apache.spark</groupId>
13   <artifactId>spark-streaming_2.10</artifactId>
14   <version>${spark.version}</version>
15 </dependency>
```

然后，要使用 SparkPipelineOptions 传递进 Pipeline.create() 方法。常见的创建方法是从命令行中读取参数来创建 PipelineOption，使用的是 PipelineOptionsFactory.fromArgs(String[]) 这个方法。在命令行中，你需要指定 runner=SparkRunner:

 复制代码

```
1 mvn exec:java -Dexec.mainClass=YourMainClass \
2   -Pspark-runner \
3   -Dexec.args="--runner=SparkRunner \
4     --sparkMaster=<spark master url>"
```

也可以在 Spark 的独立集群上运行，这时候 spark 的提交命令，spark-submit。

```
1 spark-submit --class YourMainClass --master spark://HOST:PORT target/...jar 复制代码
```

当 Beam 程序在 Spark 上运行时，你也可以同样用 Spark 的网页监控数据流水线进度。

## Flink 运行模式

Flink Runner 是 Beam 提供的用来在 Flink 上运行 Beam Pipeline 的模式。你可以选择在计算集群上比如 Yarn/Kubernetes/Mesos 或者本地 Flink 上运行。Flink Runner 适合大规模，连续的数据处理任务，包含了以下功能：

1. 以 Streaming 为中心，支持 streaming 处理和 batch 处理；
2. 和 flink 一样的容错性，和 exactly-once 的处理语义；
3. 可以自定义内存管理模型；
4. 和其他（例如 YARN）的 Apache Hadoop 生态整合比较好。

其实看到这里，你可能已经掌握了这里面的诀窍。就是通过 PipelineOptions 来指定 runner，而你的数据处理代码不需要修改。PipelineOptions 可以通过命令行参数指定。那么类似 Spark Runner，你也可以使用 Flink 来运行 Beam 程序。

同样的，首先你需要在 pom.xml 中添加 Flink Runner 的依赖。

```
1 <dependency>
2   <groupId>org.apache.beam</groupId>
3   <artifactId>beam-runners-flink-1.6</artifactId>
4   <version>2.13.0</version>
5 </dependency>
```

复制代码

然后在命令行中指定 flink runner：

```
1 mvn exec:java -Dexec.mainClass=YourMainClass \
```

复制代码




```
2 -Pflink-runner \  
3 -Dexec.args="--runner=FlinkRunner \  
4 --flinkMaster=<flink master url>"
```

## Google Dataflow 运行模式


Beam Pipeline 也能直接在云端运行。Google Cloud Dataflow 就是完全托管的 Beam Runner。当你使用 Google Cloud Dataflow 服务来运行 Beam Pipeline 时，它会先上传你的二进制程序到 Google Cloud，随后自动分配计算资源创建 Cloud Dataflow 任务。

同前面讲到的 Direct Runner 和 Spark Runner 类似，你还是需要为 Cloud Dataflow 添加 beam-runners-google-cloud-dataflow-java 依赖关系：

 复制代码

```
1 <dependency>  
2   <groupId>org.apache.beam</groupId>  
3   <artifactId>beam-runners-google-cloud-dataflow-java</artifactId>  
4   <version>2.13.0</version>  
5   <scope>runtime</scope>  
6 </dependency>
```

我们假设你已经在 Google Cloud 上创建了 project，那么就可以用类似的命令行提交任务：

 复制代码

```
1 mvn -Pdataflow-runner compile exec:java \  
2   -Dexec.mainClass=<YourMainClass> \  
3   -Dexec.args="--project=<PROJECT_ID> \  
4   --stagingLocation=gs://<STORAGE_BUCKET>/staging/ \  
5   --output=gs://<STORAGE_BUCKET>/output \  
6   --runner=DataflowRunner"
```

## 小结

这一讲我们先总结了前面几讲 Pipeline 的完整使用方法。之后一起探索了 Beam 的重要特性，就是 Pipeline 可以通过 PipelineOption 动态选择同样的数据处理流水线在哪里运行。并

且，分别展开讲解了直接运行模式、Spark 运行模式、Flink 运行模式和 Google Cloud Dataflow 运行模式。在实践中，你可以根据自身需要，去选择不同的运行模式。

## 思考题

Beam 的设计模式是对计算引擎动态选择，它为什么要这么设计？

欢迎你把答案写在留言区，与我和其他同学一起讨论。如果你觉得有所收获，也欢迎把文章分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言 (5)



**suncar**

2019-07-01

请问一下老师，可不可以提供几个获取大量测试数据的网址。谢谢

作者回复：谢谢留言！我比较推荐kaggle的datasets。



👍 11



**明翼**

2019-07-02

想问下读者中多少人用beam在生产环境...

共 6 条评论 >

👍 5



**hugo**

2020-10-23

runner是如何在多平台，多语言间实现兼容的？像flink，go runner会在本地调用java runner吗



👍 1



**David**

2020-03-04

请教一下，GCP上同时有Composer/Airflow和Dataflow/Beam两种可以用来完成ETL工作的产品。

是否可以讲一下两者的比较，和在技术上如何进行选型？

谢谢！



1



**ditiki**

2019-07-03

请教两个production遇到的问题.

In a beam pipeline (dataflow), one step is to send http request to schema registry to validate event schema. A groupby event type before this step and static cache are used to reduce calls to schema registry. How does beam (or the underline runner) optimise IO ? Is it a good practice to use a thread pool for asynchronous http calls ?

The event object has a Json (json4s library) payload, each time we try to update the Dataflow pipeline, we get the error says that the Kryo coder generated for the JSON has changed, such that the current pipeline can't be updated in place. We did a work a round by serialise the Json payload to string in a custom coder, which should be very inefficient. Have you ever seen this before ? Does Kryo generate a different coder at each compile time ?

多谢啦！

