30-ApacheBeam实战冲刺：Beam如何runeverywhere-

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

```
// Start by defining the options for the pipeline.
PipelineOptions options = PipelineOptionsFactory.create();

// Then create the pipeline.
Pipeline pipeline = Pipeline.create(options);

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

PCollection<String> filteredLines = lines.apply(new FilterLines());

filteredLines.apply("WriteMyFile", TextIO.write().to("gs://some/outputData.txt"));

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

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

PCollection<String> input = p.apply(Create.of(WORDS)).setCoder(StringUtf8Coder.of());

PCollection<String> output = input.apply(new CountWords());

PAssert.that(output).containsInAnyOrder(COUNTS_ARRAY);

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

```
options = PipelineOptionsFactory.as(SparkPipelineOptions.class);
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

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

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。

```
pom.xml
<dependency>
   <groupId>org.apache.beam</groupId>
   <artifactId>beam-runners-direct-java</artifactId>
   <version>2.13.0</version>
   <scope>runtime</scope>
</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。

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

在实验程序中也可以强行使用Direct Runner。比如：

```
PipelineOptions options = PipelineOptionsFactory.create();
options.setRunner(DirectRunner.class);
// 或者这样
options = PipelineOptionsFactory.as(DirectRunner.class);
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：

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

```
<dependency>
  <groupId>org.apache.beam</groupId>
  <artifactId>beam-runners-spark</artifactId>
  <version>2.13.0</version>
</dependency>
<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>
```

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：

```
mvn exec:java -Dexec.mainClass=YourMainClass \
    -Pspark-runner \
    -Dexec.args="--runner=SparkRunner \
      --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。

```
spark-submit --class YourMainClass --master spark://HOST:PORT target/...jar --runner=SparkRunner
```

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。

```
<dependency>
  <groupId>org.apache.beam</groupId>
```

```
  <artifactId>beam-runners-flink-1.6</artifactId>
  <version>2.13.0</version>
</dependency>
```

然后在命令行中指定flink runner：

```
mvn exec:java -Dexec.mainClass=YourMainClass \
    -Pflink-runner \
    -Dexec.args="--runner=FlinkRunner \
      --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：

```
<dependency>
  <groupId>org.apache.beam</groupId>
  <artifactId>beam-runners-google-cloud-dataflow-java</artifactId>
  <version>2.13.0</version>
  <scope>runtime</scope>
</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：

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

小结

这一节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。

精选留言：

- 路晓明 2019-07-01 17:25:49
  请问一下老师，可不可提供几个获取大量测试数据的网止。谢谢