

## 29 | 如何测试Beam Pipeline?

2019-06-28 蔡元楠 来自北京

《大规模数据处理实战》

29



你好，我是蔡元楠。

今天我要与你分享的主题是“如何测试 Beam Pipeline”。

在上一讲中，我们结合了第 7 讲的内容，一起学习了在 Beam 的世界中我们该怎么设计好对应的设计模式。而在今天这一讲中，我想要讲讲在日常开发中经常会被忽略的，但是又非常重要的一个开发环节——测试。

你知道，我们设计好的 Beam 数据流水线通常都会被放在分布式环境下执行，具体每一步的 Transform 都会被分配到任意的机器上面执行。如果我们在运行数据流水线时发现结果出错了，那么想要定位到具体的机器，再到上面去做调试是不现实的。

当然还有另一种方法，读取一些样本数据集，再运行整个数据流水线去验证哪一步逻辑出错了。但这是一项非常耗时耗力的工作。即便我们可以把样本数据集定义得非常小，从而缩短运

行数据流水线运行所需的时间。但是万一我们所写的是多步骤数据流水线的话，就不知道到底在哪一步出错了，我们必须把每一步的中间结果输出出来进行调试。

基于以上种种的原因，在我们正式将数据流水线放在分布式环境上面运行之前，先完整地测试好整个数据流水线逻辑，就变得尤为重要了。

为了解决这些问题，Beam 提供了一套完整的测试 SDK。让我们可以在开发数据流水线的同时，能够实现对一个 Transform 逻辑的单元测试，也可以对整个数据流水线端到端（End-to-End）地测试。

在 Beam 所支持的各种 Runners 当中，有一个 Runner 叫作 DirectRunner。DirectRunner 其实就是我们的本地机器。也就是说，如果我们指定 Beam 的 Runner 为 DirectRunner 的话，整个 Beam 数据流水线都会放在本地机器上面运行。我们在运行测试程序的时候可以利用这个 DirectRunner 来跑测试逻辑。

在正式讲解之前，有一点是我需要提醒你的。如果你喜欢自行阅读 Beam 的相关技术文章或者是示例代码的话，可能你会看见一些测试代码使用了在 Beam SDK 中的一个测试类，叫作 DoFnTester 来进行单元测试。这个 DoFnTester 类可以让我们传入一个用户自定义的函数（User Defined Function/UDF）来进行测试。

通过 [第 25 讲](#) 的内容我们已经知道，一个最简单的 Transform 可以用一个 ParDo 来表示，在使用它的时候，我们需要继承 DoFn 这个抽象类。这个 DoFnTester 接收的对象就是我们继承实现的 DoFn。在这里，我们把一个 DoFn 看作是一个单元来进行测试了。但这并不是 Beam 所提倡的。

因为在 Beam 中，数据转换的逻辑都是被抽象成 Transform，而不是 Transform 里面的 ParDo 这些具体的实现。**每个 Runner 具体怎么运行这些 ParDo，对于用户来说应该都是透明的。**所以，在 Beam 的 2.4.0 版本之后，Beam SDK 将这个类标记成了 Deprecated，转而推荐使用 Beam SDK 中的 TestPipeline。

所以，我在这里也建议你，在写测试代码的时候，不要使用任何和 DoFnTester 有关的 SDK。

## Beam 的 Transform 单元测试

说完了注意事项，那事不宜迟，我们就先从一个 Transform 的单元测试开始，看看在 Beam 是如何做测试的（以下所有的测试示例代码都是以 Java 为编程语言来讲解）。

一般来说，Transform 的单元测试可以通过以下五步来完成：

1. 创建一个 Beam 测试 SDK 中所提供的 TestPipeline 实例。
2. 创建一个静态（Static）的、用于测试的输入数据集。
3. 使用 Create Transform 来创建一个 PCollection 作为输入数据集。
4. 在测试数据集上调用我们需要测试的 Transform 上并将结果保存在一个 PCollection 上。
5. 使用 PAssert 类的相关函数来验证输出的 PCollection 是否是我所期望的结果。

假设我们要处理的数据集是一个整数集合，处理逻辑是过滤掉数据集中的奇数，将输入数据集中的偶数输出。为此，我们通过继承 DoFn 类来实现一个产生偶数的 Transform，它的输入和输出数据类型都是 Integer。

Java

 复制代码


```
1 static class EvenNumberFn extends DoFn<Integer, Integer> {
2     @ProcessElement
3     public void processElement(@Element Integer in, OutputReceiver<Integer> out) {
4         if (in % 2 == 0) {
5             out.output(in);
6         }
7     }
8 }
```

那我们接下来就根据上面所讲的测试流程，测试这个 EvenNumberFn Transform，来一步步创建我们的单元测试。

### 创建 TestPipeline

第一步，创建 TestPipeline。创建一个 TestPipeline 实例的代码非常简单，示例如下：

Java


 复制代码

```
1 ...  
2 Pipeline p = TestPipeline.create();  
3 ...
```

如果你还记得在 [第 26 讲](#) 中如何创建数据流水线的话，可以发现，TestPipeline 实例的创建其实不用给这个 TestPipeline 定义选项（Options）。因为 TestPipeline 中 create 函数已经在内部帮我们创建好一个测试用的 Options 了。

## 创建静态输入数据集

Java

 复制代码


```
1 ...  
2 static final List<Integer> INPUTS = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
3 ...
```

第二步，创建静态的输入数据集。创建静态的输入数据集的操作就和我们平时所写的普通 Java 代码一样，在示例中，我调用了 Arrays 类的 asList 接口来创建一个拥有 10 个整数的数据集。

## 使用 Create Transform 创建 PCollection

在创建完静态数据集后，我们进入第三步，创建一个 PCollection 作为输入数据集。在 Beam 原生支持的 Transform 里面，有一种叫作 Create Transform，我们可以利用这个 Create Transform 将 Java Collection 的数据转换成为 Beam 的数据抽象 PCollection，具体的做法如下：

## Java

 复制代码

```
1 ...
2 PCollection<Integer> input = p.apply(Create.of(INPUTS)).setCoder(VarIntCoder.of())
3 ...
```

## 调用 Transform 处理逻辑

第四步，调用 Transform 处理逻辑。有了数据抽象 PCollection，我们就需要在测试数据集上调用我们需要测试的 Transform 处理逻辑，并将结果保存在一个 PCollection 上。

## Java

 复制代码


```
1 ...
2 PCollection<String> output = input.apply(ParDo.of(new EvenNumberFn()));
3 ...
```

根据 [第 25 讲](#) 的内容，我们只需要在这个输入数据集上调用 apply 抽象函数，生成一个需要测试的 Transform，并且传入 apply 函数中就可以了。

## 验证输出结果

第五步，验证输出结果。在验证结果的阶段，我们需要调用 PAssert 类中的函数来验证输出结果是否和我们期望的一致，示例如下。


## Java

 复制代码

```
1 ...
2 PAssert.that(output).containsInAnyOrder(2, 4, 6, 8, 10);
3 ...
```

完成了所有的步骤，我们就差运行这个测试的数据流水线了。很简单，就是调用 TestPipeline 的 run 函数，整个 Transform 的单元测试示例如下：

## Java

 复制代码

```
1 final class TestClass {
2     static final List<Integer> INPUTS = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9,
3
4     public void testFn() {
5         Pipeline p = TestPipeline.create();
6         PCollection<Integer> input = p.apply(Create.of(INPUTS)).setCoder(VarIntCoder);
7         PCollection<String> output = input.apply(ParDo.of(new EvenNumberFn()));
8         PAssert.that(output).containsInAnyOrder(2, 4, 6, 8, 10);
9         p.run();
10    }
11 }
```

有一点需要注意的是，TestPipeline 的 run 函数是在单元测试的结尾处调用的，PAssert 的调用必须在 TestPipeline 调用 run 函数之前调用。

## Beam 的端到端测试

在一般的现实应用中，我们设计的都是多步骤数据流水线，就拿我在 [第一讲](#) 中举到的处理美团外卖电动车的图片为例子，其中就涉及到了多个输入数据集，而结果也有可能根据实际情况有多个输出。

所以，我们在做测试的时候，往往希望能有一个端到端的测试。在 Beam 中，端到端的测试和 Transform 的单元测试非常相似。唯一的不同点在于，我们要为所有的输入数据集创建测试数据集，而不是只针对某一个 Transform 来创建。对于在数据流水线的每一个应用到 Write Transform 的地方，我们都需要用到 PAssert 类来验证输出数据集。


所以，端到端测试的步骤也分五步，具体内容如下：

1. 创建一个 Beam 测试 SDK 中所提供的 TestPipeline 实例。

2. 对于多步骤数据流水线中的每个输入数据源，创建相对应的静态（Static）测试数据集。
3. 使用 Create Transform，将所有的这些静态测试数据集转换成 PCollection 作为输入数据集。
4. 按照真实数据流水线逻辑，调用所有的 Transforms 操作。
5. 在数据流水线中所有应用到 Write Transform 的地方，都使用 PAssert 来替换这个 Write Transform，并且验证输出的结果是否我们期望的结果相匹配。

为了方便说明，我们就在之前的例子中多加一步 Transform 和一个输出操作来解释如何写端到端测试。假设，我们要处理数据集是一个整数集合，处理逻辑是过滤掉奇数，将输入数据集中的偶数转换成字符串输出。同时，我们也希望对这些偶数求和并将结果输出，示例如下：

## Java

 复制代码

```
1 final class Foo {
2     static class EvenNumberFn extends DoFn<Integer, Integer> {
3         @ProcessElement
4         public void processElement(@Element Integer in, OutputReceiver<Integer> out)
5             if (in % 2 == 0) {
6                 out.output(in);
7             }
8     }
9 }
10
11 static class ParseIntFn extends DoFn<String, Integer> {
12     @ProcessElement
13     public void processElement(@Element String in, OutputReceiver<Integer> out) {
14         out.output(Integer.parseInt(in));
15     }
16 }
17
18 public static void main(String[] args) {
19     PipelineOptions options = PipelineOptionsFactory.create();
20     Pipeline p = Pipeline.create(options);
21     PCollection<Integer> input = p.apply(TextIO.read().from("filepath/input")).ap
22     PCollection<Integer> output1 = input.apply(ParDo.of(new EvenNumberFn()));
23     output1.apply(ToString.elements()).apply(TextIO.write().to("filepath/evenNumb
24     PCollection<Integer> sum = output1.apply(Combine.globally(new SumInts()));
25     sum.apply(ToString.elements()).apply(TextIO.write().to("filepath/sum"));
26     p.run();
```



```
27     }
28 }
```

从上面的示例代码中你可以看到，我们从一个外部源读取了一系列输入数据进来，将它转换成了整数集合。同时，将我们自己编写的 EvenNumberFn Transform 应用在了这个输入数据集上。得到了所有偶数集合之后，我们先将这个中间结果输出，然后再针对这个偶数集合求和，最后将这个结果输出。

整个数据流水线总共有一次对外部数据源的读取和两次的输出，我们按照端到端测试的步骤，为所有的输入数据集创建静态数据，然后将所有有输出的地方都使用 PAssert 类来进行验证。整个测试程序如下所示：

## Java

 复制代码

```
1 final class TestClass {
2
3     static final List<String> INPUTS =
4         Arrays.asList("1", "2", "3", "4", "5", "6", "7", "8", "9", "10");
5
6     static class EvenNumberFn extends DoFn<Integer, Integer> {
7         @ProcessElement
8         public void processElement(@Element Integer in, OutputReceiver<Integer> out)
9             if (in % 2 == 0) {
10                 out.output(in);
11             }
12     }
13 }
14
15 static class ParseIntFn extends DoFn<String, Integer> {
16     @ProcessElement
17     public void processElement(@Element String in, OutputReceiver<Integer> out) {
18         out.output(Integer.parseInt(in));
19     }
20 }
21
22 public void testFn() {
23     Pipeline p = TestPipeline.create();
24     PCollection<String> input = p.apply(Create.of(INPUTS)).setCoder(StringUtf8Cod
25     PCollection<Integer> output1 = input.apply(ParDo.of(new ParseIntFn())).apply(
26     PAssert.that(output1).containsInAnyOrder(2, 4, 6, 8, 10);
```



```
27     PCollection<Integer> sum = output1.apply(Combine.globally(new SumInts()));
28     PAssert.that(sum).is(30);
29     p.run();
30 }
31 }
```

在上面的示例代码中，我们用 TestPipeline 替换了原来的 Pipeline，创建了一个静态输入数据集并用 Create Transform 转换成了 PCollection，最后将所有用到 Write Transform 的地方都用 PAssert 替换掉，来验证输出结果是否是我们期望的结果。

## 小结

今天我们一起学习了在 Beam 中写编写测试逻辑的两种方式，分别是针对一个 Transform 的单元测试和针对整个数据流水线的端到端测试。Beam 提供的 SDK 能够让我们不需要在分布式环境下运行程序而是本地机器上运行。测试在整个开发环节中是非常的一环，我强烈建议你在正式上线自己的业务逻辑之前先对此有一个完整的测试。

## 思考题

如果让你来利用 Beam SDK 来测试你日常处理的数据逻辑，你会如何编写测试呢？

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 (6)



明翼

2019-07-02

我期望是基于spark或flink讲解的重实践思想，轻知识，这个可以自己下去学





2019-09-10

看来我得先去学学Java了，不会java，看不太懂。



2



**Ming**

2019-06-30

我觉得测试的话，相对麻烦的地方还是在工程脚手架的设计上。

显然代码本身要抽象封装好，确保测试能覆盖生产代码，而不是生产代码的“拷贝”。

但有些在代码之外的问题让我挺好奇的：

如何保证测试数据的格式和生产数据的格式同步？

流处理的测试怎么模拟时间？

团队是如何在流程上确保pipeline必须经过测试才能运行的，是通过CI/CD来自动执行pipeline？还是往往通过人力把关？



2



**之渊**

2020-08-22

我觉得挺好的，对于我们新手来说很友好。

谁都是从新手过来的，感谢大佬对新手的支持。

代码示例：

<https://gitee.com/oumin12345/daimademojihe/tree/master/cloudx/bigdata/src/main/java/test/beam>



**杰洛特**

2019-11-14

TestClass 里的这个 `PCollection<String> output = input.apply(ParDo.of(new EvenNumberFn()));` 里面的泛型是不是写错了？偶数是 Integer 吧？

共 1 条评论 >



**李孟**

2019-07-08

Beam 有类似sparksql的api吗？

作者回复：谢谢你的提问！有Beam SQL，不过现在只支持Java。



