29-如何测试BeamPipeline？

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

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

那我们接下来就根据上面所讲的测试流程，测试这个EvenNumerFn 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

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

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

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

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

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

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

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

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

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

```
...
```

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

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

    public void testFn() {
      Pipeline p = TestPipeline.create();
      PCollection<Integer> input = p.apply(Create.of(INPUTS)).setCoder(VarIntCoder.of());
      PCollection<String> output = input.apply(ParDo.of(new EvenNumberFn()));
      PAssert.that(output).containsInAnyOrder(2, 4, 6, 8, 10);
      p.run();
    }
}
```

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TestPipeliner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，而结果也有可能会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 Transfrom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

```
final class Foo {
  static class EvenNumberFn extends DoFn<Integer, Integer> {
    @ProcessElement
    public void processElement(@Element Integer in, OutputReceiver<Integer> out) {
      if (in % 2 == 0) {
        out.output(in);
      }
    }
  }

  static class ParseIntFn extends DoFn<String, Integer> {
    @ProcessElement
    public void processElement(@Element String in, OutputReceiver<Integer> out) {
      out.output(Integer.parseInt(in));
    }
  }

  public static void main(String[] args) {
    PipelineOptions options = PipelineOptionsFactory.create();
    Pipeline p = Pipeline.create(options);
    PCollection<Integer> input = p.apply(TextIO.read().from("filepath/input")).apply(ParDo.of(new ParseIntF
    PCollection<Integer> output1 = input.apply(ParDo.of(new EvenNumberFn()));
    output1.apply(ToString.elements()).apply(TextIO.write().to("filepath/evenNumbers"));
    PCollection<Integer> sum = output1.apply(Combine.globally(new SumInts()));
    sum.apply(ToString.elements()).apply(TextIO.write().to("filepath/sum"));
    p.run();
  }
}
```

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

```
final class TestClass {

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

  static class EvenNumberFn extends DoFn<Integer, Integer> {
    @ProcessElement
    public void processElement(@Element Integer in, OutputReceiver<Integer> out) {
      if (in % 2 == 0) {
        out.output(in);
      }
    }
  }

  static class ParseIntFn extends DoFn<String, Integer> {
    @ProcessElement
    public void processElement(@Element String in, OutputReceiver<Integer> out) {
```

```
      out.output(Integer.parseInt(in));
    }
  }

  public void testFn() {
    Pipeline p = TestPipeline.create();
    PCollection<String> input = p.apply(Create.of(INPUTS)).setCoder(StringUtf8Coder.of());
    PCollection<Integer> output1 = input.apply(ParDo.of(new ParseIntFn())).apply(ParDo.of(new EvenNumberFn(
    PAssert.that(output1).containsInAnyOrder(2, 4, 6, 8, 10);
    PCollection<Integer> sum = output1.apply(Combine.globally(new SumInts()));
    PAssert.that(sum).is(30);
    p.run();
  }
}
```

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。