36-Facebook游戏实时流处理BeamPipeline实战（下）

你好，我是蔡元楠。

在上一讲中，我们一起对怎样实现一个简易的游戏积分排行榜展开了讨论，也一起研究了如何使用批处理计算的方式在Beam中构建出一个数据流水线来得出排行榜结果。

我们知道，虽然批处理计算可以得到一个完整的结果，但是它也存在着自身的不足，比如会有一定的延时，需要额外的crontab来管理定时任务，增加了维护成本等等。

所以在上一讲的末尾，我们提出了使用实时流处理来改进这些不足，而其中就需要用到窗口、触发器和累加模式这几个概念。

相信学习了第32讲的内容后，你对于窗口在Beam中是如何运作的，已经比较了解了。对于有效时间为一周的积分排行榜来说，我们可以赋予一个“窗口时长为一周的固定窗口”给数据流水线。也就是说，我们最终的结果会按照每一周的时长来得出。

那接下来的问题就剩下我们怎么定义触发器和累加模式了。

首先，我想先讲讲触发器在Beam中是怎么运作的。在第23讲中，我们已经了解了触发器在Beam中的作用。它是用于告诉数据流水线，什么时候需要计算一遍落在窗口中的所有数据的。这在实时流处理中尤为重要。

在实时流处理当中，我们总是需要在数据结果的**完整性**和**延迟性**上做出一些取舍。

如果我们设置的触发器比较频繁，例如说每隔几分钟甚至是几秒钟，或者是在时间上很早发生的话，那就表示我们更倾向于数据流水线的延时比较小，但是不一定能够获得完整的数据。

如果我们设置的触发器是比较长时间的，像每隔一个小时才会触发一次窗口中的计算的话，那就表示我们更希望获得完整的数据集来得到最终结果。

为什么这么说呢？

因为在现实世界中，我们是没有办法保证数据流水线可以在某一刻能够得到在这一刻之前所产生的所有数据的。

就拿获得这个游戏积分排行榜的数据为例子来说明一下。

在现实生活中，可能很多用户是用手机来通关游戏，并且上传通关时间和积分的。有的地方可能因为信号差，上传数据会有很大的延迟。甚至可能有这样的情况：有一些用户是在坐飞机的时候玩的游戏（在飞行模式之下完成各种通关），等到飞机降落，手机重新有了信号之后，数据才被上传到服务器，这时候可能已经有了好几个小时的延时了。

如果提早触发窗口中的数据计算，可能会有很多“迟到”的数据未被纳入最终结果中，而这些“迟到”的数据有可能又会影响到游戏积分排行榜。

所以，在很多复杂的场景下，我们希望尽可能地将所有数据归入正确的时间窗口中，而且还要能够得到正确

的结果。因此，除了要对触发器进行设置之外，我们还需要设置到底应不应该在一些“迟到”的数据来到的时候，重新计算一下整个结果。

在Beam中，我们可以为PCollection设置的触发器有4种模式：

1.**基于事件时间的触发器**（Event-Time Trigger）

如果设置了基于事件时间的触发器，那所有的计算都是基于PCollection中所有元素的事件时间的。

如果我们不显式地设置触发器的话，Beam的默认触发器就是基于事件时间的，如果要显式地设置基于事件时间的触发器，可以使用AfterWatermark类进行设置。

2.**基于处理时间触发器**（Process-Time Trigger）

如果设置了基于处理时间的触发器，那一个PCollection中的所有元素都会在数据流水线中的某一个时刻被处理。如果要显式地设置基于处理时间的触发器，可以使AfterProcessingTime类进行设置。

3.**数据驱动的触发器**（Data-Driven Trigger）

数据驱动的触发器一般是在每个元素到达每个窗口时，通过检查这个元素是否满足某个属性来触发的。

就好像我在第23讲所举的例子一样，检查元素是否在窗口中到达一定的数量，然后触发计算就是数据驱动的触发器的一种，在Beam中，可以使用AfterPane.elementCountAtLeast()函数来配置。

4.**复合触发器**（Composite Trigger）

复合触发器其实就是由上面所说三种基本触发器组合而成的。在第23讲中，我举过一个触发器的例子，例子中至少要等到有两个交易数据到达后才能触发计算。

有同学在留言中问我，如果现实中只有一个数据到达窗口，那岂不是永远都触发不了计算了？其实，这个时候就可以定义一个复合触发器，可以定义成累积有超过两个元素落入窗口中或者是每隔一分钟触发一次计算的复合触发器。

而像我之前提到的，如果我们需要处理“迟到”的数据，那在Beam中又是怎么操作呢？我们可以使用withAllowedLateness这个在Window类里定义好的函数，方法签名如下：

Java

```
public Window<T> withAllowedLateness(Duration allowedLateness);
```

这个函数接收的参数就是我们希望允许多久的“迟到”数据可以被纳入计算中。

最后需要说明的是累加模式。

在Beam中，我们可以设置两种累加模式，分别是**丢弃模式**和**累积模式**。它们可以分别通过Window类里的函数discardingFiredPanes()和accumulatingFiredPanes()来设置。

好了，那现在回到我们的积分排行榜问题当中。

虽然我们对输入数据集设定的窗口是一个窗口时长为1周的固定窗口，但是我们也需要尽可能地在近乎实时的状态下更新排行榜。所以，我们可以设置数据流水线在每5分钟更新一次。

那我们接受“迟到”多久的数据呢？

我在网上查询了一下，现在飞机航班直飞耗时最长的是新加坡飞往纽约的航班，大概需要19个小时。如果玩游戏的用户恰好也在这趟航班上，那么可能数据的延时可能就会超过19个小时了。那我们就设定允许“迟到”20个小时的数据也纳入我们的窗口计算当中。

一般情况下，我们可以从Pub/Sub数据流中读取实时流数据。为了简化数据流水线的逻辑，不在数据流水线中保存中间状态，我们现在假设在实际操作的时候，服务器已经判断好某一用户的分数是否是最高分，如果是最高分的话，再通过Pub/Sub将数据传入流水线。

这时，我们的累加模式可以定义为丢弃模式，也就是只保留最新的结果。

为此，我们可以写出一个Transform来设置所有上述描述的概念，分别是：

1. 设置窗口时长为1周的固定窗口。
2. 每隔5分钟就会计算一次窗口内数据的结果。
3. 允许“迟到”了20个小时的数据被重新纳入窗口中计算。
4. 采用丢弃模式来保存最新的用户积分。

Java

```
static class ConfigUserScores extends PTransform<PCollection<UserScoreInfo>, PCollection<UserScoreInfo>> {
    private final Duration FIXED_WINDOW_SIZE = Duration.standardDays(7);
    private final Duration FIVE_MINUTES = Duration.standardMinutes(5);
    private final Duration TWENTY_HOURS = Duration.standardHours(20);

    @Override
    public PCollection<UserScoreInfo> expand(PCollection<UserScoreInfo> infos) {
      return infos.apply(
          Window.<UserScoreInfo>into(FixedWindows.of(FIXED_WINDOW_SIZE))
              .triggering(
                  AfterWatermark.pastEndOfWindow()
                      .withEarlyFirings(
                          AfterProcessingTime.pastFirstElementInPane().plusDelayOf(FIVE_MINUTES))
                      .withLateFirings(
                          AfterProcessingTime.pastFirstElementInPane().plusDelayOf(FIVE_MINUTES)))
              .withAllowedLateness(TWENTY_HOURS)
              .discardingFiredPanes());
    }
  }
```

有了这个Transform去设定好我们在实时流处理中如何处理数据之后，我们其实只需要修改之前批处理数据流水线中很小的一部分，就可以达到我们想要的结果了。

Java

```
...
pipeline.apply(
            KafkaIO.<String>read()
                .withBootstrapServers("broker_1:9092,broker_2:9092")
                .withTopic("user_scores")
                .withKeyDeserializer(StringDeserializer.class)
                .withValueDeserializer(StringDeserializer.class)
                .withLogAppendTime())
        .apply("ConvertUserScoreInfo", ParDo.of(new ConvertUserScoreInfoFn()))
        .apply("ConfigUserScores", new ConfigUserScores())
        .apply("RetrieveTop100Players", new ExtractUserAndScore())
...
```

如代码所示，真正做出修改的地方就是将读取输入数据集的BigTableIO改成使用KafkaIO来读取。将批处理的两个Filter Transform替换成我们自定义的ConfigUserScores Transform。

到此为止，我们就可以“一劳永逸”，运行一个实时流处理的数据流水线来得到游戏积分排行榜结果了。

小结

今天我们一起设计了一个实时流处理的数据流水线，来完成之前自定义的一个简单游戏积分排行榜。

这里面有不少经验是值得我们在处理现实的应用场景中借鉴的。比如，我们应该考虑数据结果的完整性有多重要、我们能接受多大的延迟、我们是否接受晚来的数据集等等。

这些问题其实又回到了第23讲中提到过的——我们在解决现实问题时，应该回答好的“WWWH”这四个问题。

思考题

今天我们一起探讨了如何利用实时流处理的方式来解决游戏积分排行榜的问题，里面涉及了配置窗口，触发器和累加模式。这些配置可能还不是最优的，你觉得我们还有什么地方可以进行优化的呢？

欢迎你把自己的学习体会写在留言区，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。