29-流计算与消息（一）：通过Flink理解流计算的原理

你好，我是李玥。

在上节课中，我简单地介绍了消息队列和流计算的相关性。在生产中，消息队列和流计算往往是相互配合，一起来使用的。而流计算也是后端程序员技术栈中非常重要的一项技术。在接下来的两节课中，我们一起通过两个例子来实际演练一下，如何使用消息队列配合流计算框架实现一些常用的流计算任务。

这节课，我们一起来基于Flink实现一个流计算任务，通过这个例子来感受一下流计算的好处，同时我还会给你讲解流计算框架的实现原理。下一节课中，我们会把本节课中的例子升级改造，使用Kafka配合Flink来实现Exactly Once语义，确保数据在计算过程中不重不丢。

无论你之前是否接触过像Storm、Flink或是Spark这些流计算框架都没有关系，因为我们已经学习了消息队列的实现原理，以及实现消息队列必备的像异步网络传输、序列化这些知识。在掌握了这些知识和底层的原理之后，再来学习和理解流计算框架的实现原理，你会发现，事情就变得非常简单了。

为什么这么说，一个原因是，对于很多中间件或者说基础框架这类软件来说，它们用到很多底层的技术都是一样；另外一个原因是，流计算和消息队列处理的都实时的、流动的数据，很多处理流数据的方法也是一样的。

哪些问题适合用流计算解决?

首先，我们来说一下，哪些问题适合用流计算来解决？或者说，流计算它的应用场景是什么样的呢?

在这里，我用一句话来回答这个问题：**对实时产生的数据进行实时统计分析，这类场景都适合使用流计算来实现。**

你在理解这句话的时候，需要特别注意的是，这里面有两个“实时”，一个是说，数据是“实时”产生的，另一个是说，统计分析这个过程是“实时”进行的，统计结果也是第一时间就计算出来了。对于这样的场景，你都可以考虑使用流计算框架。

因为流计算框架可以自动地帮我们实现实时的并行计算，性能非常好，并且内置了很多常用的统计分析的算子，比如TimeWindow、GroupBy、Sum和Count，所以非常适合用来做实时的统计和分析。举几个例子：

- 每分钟按照IP统计Web请求次数；
- 电商在大促时，实时统计当前下单量；
- 实时统计App中的埋点数据，分析营销推广效果。

以上这些场景，以及和这些场景类似的场景，都是比较适合用流计算框架来实现的。特别是基于时间维度的统计分析，使用流计算框架来实现是非常方便的。

用代码定义Job并在Flink中执行

接下来，我们用Flink来实现一个实时统计任务：接收NGINX的access.log，每5秒钟按照IP地址统计Web请求的次数。这个统计任务它一个非常典型的，按照Key来进行分类汇总的统计任务，并且汇总是按照一定周期来实时进行的，我们日常工作中遇到的很多统计分析类的需求，都可以套用这个例子的模式来实现，所以我们就以它为例来做一个实现。

假设我们已经有一个实时发送access.log的日志服务，它运行在本地的9999端口上，只要有客户端连接上来，他就会通过Socket给客户端发送实时的访问日志，日志的内容只包含访问时间和IP地址，每条数据的结尾用一个换行符(\n)作为分隔符。这个日志服务就是我们流计算任务的数据源。

我们用NetCat连接到这个服务上，看一下数据格式：

```
$nc localhost 9999
14:37:11 192.168.1.3
14:37:11 192.168.1.2
14:37:12 192.168.1.4
14:37:14 192.168.1.2
14:37:14 192.168.1.4
14:37:14 192.168.1.3
...
```

接下来我们用Scala语言和Flink来实现这个流计算任务。你可以先不用关心如何部署启动Flink，如何设置开发环境这些问题，一起来跟我看一下定义这个流计算任务的代码：

```
object SocketWindowIpCount {

  def main(args: Array[String]) : Unit = {

    // 获取运行时环境
    val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
    // 按照EventTime来统计
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    // 设置并行度
    env.setParallelism(4)

    // 定义输入：从Socket端口中获取数据输入
    val hostname: String = "localhost"
    val port: Int = 9999
    // Task 1
    val input: DataStream[String] = env.socketTextStream(hostname, port, '\n')

    // 数据转换：将非结构化的以空格分隔的文本转成结构化数据IpAndCount
    // Task 2
    input
      .map { line => line.split("\\s") }
      .map { wordArray => IpAndCount(new SimpleDateFormat("HH:mm:ss").parse(wordArray(0)), wordArray(1), 1)

    // 计算：每5秒钟按照ip对count求和

      .assignAscendingTimestamps(_.date.getTime) // 告诉Flink时间从哪个字段中获取


      .keyBy("ip") // 按照ip地址统计
      // Task 3
      .window(TumblingEventTimeWindows.of(Time.seconds(5))) // 每5秒钟统计一次
      .sum("count") // 对count字段求和

    // 输出：转换格式，打印到控制台上

      .map { aggData => new SimpleDateFormat("HH:mm:ss").format(aggData.date) + " " + aggData.ip + " " + ag
      .print()
```

```
    env.execute("Socket Window IpCount")
  }

  /** 中间数据结构 */

  case class IpAndCount(date: Date, ip: String, count: Long)
}
```

我来给你解读一下这段代码。

首先需要获取流计算的运行时环境，也就是这个env对象，对env做一些初始化的设置。然后，我们再定义输入的数据源，这里面就是我刚刚讲的，运行在9999端口上的日志服务。

在代码中，env.socketTextStream(hostname, port, ‘\n’)这个语句中的三个参数分别是主机名、端口号和分隔符，返回值的数据类型是DataStream[String]，代表一个数据流，其中的每条数据都是String类型的。它告诉Flink，我们的数据源是一个Socket服务。这样，Flink在执行这个计算任务的时候，就会去连接日志服务来接收数据。

定义完数据源之后，需要做一些数据转换，把字符串转成结构化的数据IpAndCount，便于后续做计算。在定义计算的部分，依次告诉Flink：时间从date字段中获取，按照IP地址进行汇总，每5秒钟汇总一次，汇总方式就是对count字段求和。

之后定义计算结果如何输出，在这个例子中，我们直接把结果打印到控制台上就好了。

这样就完成了一个流计算任务的定义。可以看到，定义一个计算任务的代码还是非常简单的，如果我们要自己写一个分布式的统计程序来实现一样的功能，代码量和复杂度肯定要远远超过上面这段代码。

总结下来，无论是使用Flink、Spark还是其他的流计算框架，定义一个流计算的任务基本上都可以分为：定义输入、定义计算逻辑和定义输出三部分，通俗地说，也就是：**数据从哪儿来，怎么计算，结果写到哪儿去**，这三件事儿。

我把这个例子的代码上传到了GitHub上，你可以在这里下载，关于如何设置环境、编译并运行这个例子，我在代码中的README中都给出了说明，你可以下载查看。

执行计算任务打印出的计算结果是这样的：

```
1> 18:40:10 192.168.1.2 23
4> 18:40:10 192.168.1.4 16
4> 18:40:15 192.168.1.4 27
3> 18:40:15 192.168.1.3 23
1> 18:40:15 192.168.1.2 25
4> 18:40:15 192.168.1.1 21
1> 18:40:20 192.168.1.2 21
3> 18:40:20 192.168.1.3 31
4> 18:40:20 192.168.1.1 25
4> 18:40:20 192.168.1.4 26
```

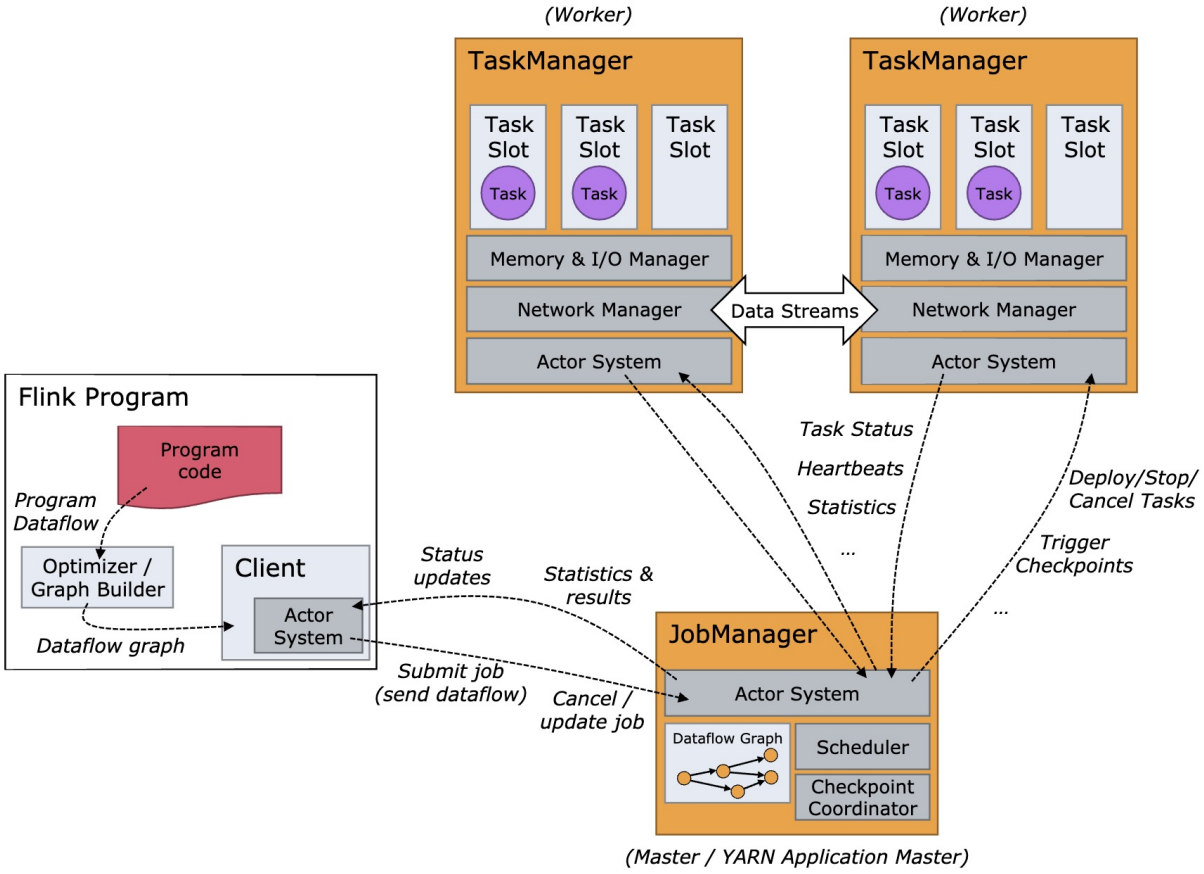
对于流计算的初学者，特别不好理解的一点是，我们上面编写的这段代码，**它只是“用来定义计算任务的代码”，而不是“真正处理数据的代码”。**对于普通的应用程序，源代码编译之后，计算机就直接执行了，这个比较好理解。而在Flink中，当这个计算任务在Flink集群的计算节点中运行的时候，真正处理数据的代码并不是我们上面写的那段代码，而是Flink在解析了计算任务之后，动态生成的代码。

这个有点儿类似于我们在查询MySQL的时候执行的SQL，我们提交一个SQL查询后，MySQL在执行查询遍历数据库中每条数据时，并不是对每条数据执行一遍SQL，真正执行的其实是MySQL自己的代码。SQL只是告诉MySQL我们要如何来查询数据，同样，我们编写的这段定义计算任务的代码，只是告诉Flink我们要如何来处理数据而已。

Job是如何在Flink集群中执行的?

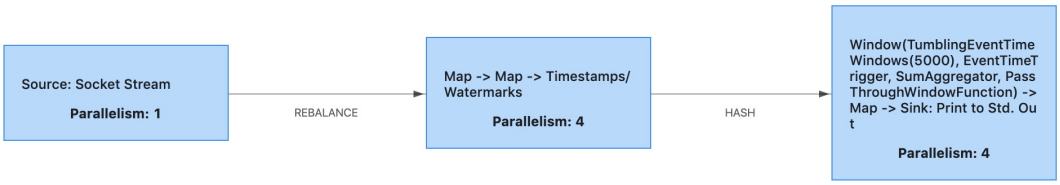
那我们的计算任务是如何在Flink中执行的呢？在讲解这个问题之前，我们先简单看一下Flink集群在运行时的架构。

下面这张图来自于Flink的官方文档。



这张图稍微有点儿复杂，我们先忽略细节看整体。Flink的集群和其他分布式系统都是类似的，集群的大部分节点都是TaskManager节点，每个节点就是一个Java进程，负责执行计算任务。另外一种节点是JobManager节点，它负责管理和协调所有的计算节点和计算任务，同时，客户端和Web控制台也是通过JobManager来提交和管理每个计算任务的。

我们编写好计算任务的代码后，打包成JAR文件，然后通过Flink的客户端提交到JobManager上。计算任务被Flink解析后，会生成一个Dataflow Graph，也叫JobGraph，简称DAG，这是一个有向无环图（DAG），比如我们的这个例子，它生成的DAG是这样的：



图中的每个节点是一个Task，每个Task就是一个执行单元，运行在某一个TaskManager的进程内。你可以想象一下，就像电流流过电路图一样，数据从Source Task流入，进入这个DAG，每流过一个Task，就被这个Task做一些计算和变换，然后数据继续流入下一个Task，直到最后一个Sink Task流出DAG，就自然完成了计算。

对于图中的3个Task，每个Task对应执行了什么计算，完全可以和我们上面定义计算任务的源代码对应上，我也在源代码的注释中，用"//Task n"的形式给出了标注。第一个Task执行的计算很简单，就是连接日志服务接收日志数据，然后将日志数据发往下一个Task。第二个Task执行了两个map变换，把文本数据转换成了结构化的数据，并添加Watermark（水印）。Watermark这个概念可以先不用管，主要是用于触发按时间汇总的操作。第三个Task执行了剩余的计算任务，按时间汇总日志，并输出打印到控制台上。

这个DAG仍然是一个逻辑图，它到底是怎么在Flink集群中执行的呢？你注意到图中每个Task都标注了一个Parallelism（并行度）的数字吗？这个并行度的意思就是，这个Task可以被多少个线程并行执行。比如图中的第二个任务，它的并行度是4，就代表Task在Flink集群中运行的时候，会有4个线程都在执行这个Task，每个线程就是一个SubTask（子任务）。注意，如果Flink集群的节点数够多，这4个SubTask可能会运行在不同的TaskManager节点上。

建立了SubTask的概念之后，我们再重新回过头来看一下这个图中的两个箭头。第一个箭头连接前两个Task，这个箭头标注了REBALANCE（重新分配），因为第一个Task并行度是1，而第二个Task并行度是4，意味着从第一个Task流出的数据将被重新分配给第二个Task的4个线程，也就是4个SubTask（子任务）中，这样就实现了并行处理。这和消息队列中每个主题分成多个分区进行并行收发的设计思想是一样的。

再来看连接第二、第三这两个Task的箭头，这个箭头上标注的是HASH，为什么呢？可以看到，第二个Task中最后一步业务逻辑是：keyBy(“ip”)，也就是按照IP这个字段做一个HASH分流。你可以想一下，第三个Task，它的并行度是4，也就是有4个线程在并行执行汇总。如果要统计每个IP的日志条数，那必须得把相同IP的数据发送到同一个SubTask（子任务）中去，这样在每个SubTask（子任务）中，对于每一条数据，只要在对应IP汇总记录上进行累加就可以了。

反之，要是相同IP的数据被分到多个SubTask（子任务）上，这些SubTask又可能分布在多个物理节点上，那就没办法统计了。所以，第二个Task会把数据按照IP地址做一个HASH分流，保证IP相同的数据都发送到第三个Task中相同的SubTask（子任务）中。这个HASH分流的设计是不是感觉很眼熟？我们之前课程中讲到的，严格顺序消息的实现方法：通过HASH算法，让key相同的数据总是发送到相同的分区上来保证严格顺序，和Flink这里的设计就是一样的。

最后在第三个Task中，4个SubTask并行进行数据汇总，每个SubTask负责汇总一部分IP地址的数据。最终打印到控制台上的时候，也是4个线程并行打印。你可以回过头去看一下输出的计算结果，每一行数据前面的数字，就是第三个Task中SubTask的编号。

到这里，我们不仅实现并运行了一个流计算任务，也理清了任务在Flink集群中运行的过程。

小结

流计算框架适合对实时产生的数据进行实时统计分析。我们通过一个“按照IP地址统计Web请求的次数”的例子，学习了Flink实现流计算任务的原理。首先，我们用一段代码定义了计算任务，把计算任务代码编译成JAR包后，通过Flink客户端提交到JobManager上。

这里需要注意的是，我们编写的代码只是用来定义计算任务，和在Flink节点上执行的真正做实时计算的代码是不一样的。真正执行计算的代码是Flink在解析计算任务后，动态生成的。

Flink分析计算任务之后生成JobGraph，JobGraph是一个有向无环图，数据流过这个图中的节点，在每个节点进行计算和变换，最终流出有向无环图就完成了计算。JobGraph中的每个节点是一个Task，Task是可以并行执行的，每个线程就是一个SubTask。SubTask被JobManager分配给某个TaskManager，在TaskManager进程中的一个线程中执行。

通过分析Flink的实现原理，我们可以看到，流计算框架本身并没有什么神奇的技术，之所以能够做到非常好的性能，主要有两个原因。一个是，它能自动拆分计算任务来实现并行计算，这个和Hadoop中Map Reduce的原理是一样的。另外一个原因是，流计算框架中，都内置了很多常用的计算和统计分析的算子，这些算子的实现都是经过很多大神级程序员反复优化过的，不仅能方便我们开发，性能上也比大多数程序员自行实现要快很多。

思考题

我们在启动Flink集群之前，修改了Flink的一个配置：槽数taskmanager.numberOfTaskSlots。请你课后看一下Flink的文档，搞清楚这个槽数的含义。然后再想一下，这个槽数和我们在计算任务中定义的并行度又是什么关系呢？

欢迎在留言区写下你的思考，如果有任何问题，也欢迎与我交流。

感谢阅读，如果你觉得这篇文章对你有一些启发，也欢迎把它分享给你的朋友。

精选留言：

- 张天屹 2019-10-04 22:16:36
  老师你好，请教个概念，虽然处理统计的程序叫做“流”计算，但是比如每5秒或者1分钟这样的时间间隔统计一次，那么对于有向无环图的每个节点，其实还是按照“批处理的”吗，即必须等这一分钟的结果汇集为一批处理完了，才传输到协议个Task节点。

  作者回复2019-10-05 10:03:58
  不是这样的，每条数据是实时流过的，并不会等待。
  只有做按时间汇聚的那个节点，它会记录汇总的中间结果（注意，它只记录汇总的中间结果，并不知把所有数据都攒起来），在每个时间窗口结束后会在流中产生一条新的数据（也就是统计结果），流往下游。

  比如，每分钟统计数据条数，汇聚的这个节点每流过一条数据，统计值就+1，它不会保存流过的数据，等到时间窗口结束，这个统计就做完了，它直接生成一条统计结果数据，发往下游。

- 川杰 2019-10-02 17:20:57
  如果服务端在处理过程是失败了呢？所以，需要客户端收到服务端明确的告知：”数据我收到并且处理成功了“，才能保证数据不会丢失。
  老师您好，对于上一问题，我还有疑问。
  对于服务端处理过程中的失败，假设场景: 业务A处理完毕后，数据需要落地，结果数据保存时出现异常，无法正常落地。
  对于这种场景，应该是业务处理完毕后就发送确认给消息产生者吧?
  我想表达的意思是，对于服务端这种业务场景，是否使用ACK，应该还是要具体问题具体分析的吧?

  作者回复2019-10-04 10:35:20
  一般应该是数据持久化完成后在发送消费成功确认。

- 川杰 2019-10-01 14:10:02
  老师，请问网上说的ACK机制，在消息队列中到底什么场景下要使用呢?
  我理解，异步线程发送消息后，虽然主线程没法捕获异常，但子线程也可以判断出是否发送成功。那么为什么还要等待接收方返回一个数据处理完的结果呢?

  作者回复2019-10-02 16:46:18

这个原因很简单，你想一下，客户端发送成功并不等于服务端处理成功，如果数据在网络传输过程中丢了呢？如果服务端在处理过程是失败了呢？所以，需要客户端收到服务端明确的告知：”数据我收到并且处理成功了“，才能保证数据不会丢失。