05 | 有向无环图（DAG）：如何描述、分解流计算过程?

今天，我们来聊聊如何用 Java 中最常见的工具类，开发一个简单的流计算框架，你会进一步在源码细节的层面，看到异步和流是如何相通的。另外，虽然这个框架简单，但它是我们从 Java 异步编程，迈入流计算领域的第一步，同时它也反映出了所有流计算框架中，最基础也是最核心的组件，即用于传递流数据的队列，和用于执行流计算的线程。

学完本课时，**你将领悟到“流”独特的计算模式，就像理解了 23 种设计模式后，有助于我们编写优秀的程序一样**。你理解了“流”这种计算模式后，**也有助以后理解各种开源流计算框架**。

在开始做事情前，我们对于自己将来要做的事情，应该是“心中有丘壑”的。所以，我们也应该先知道，该怎样去描述一个流计算过程。

为此，我们首先可以看一些开源流计算框架是怎样做的。

开源流计算框架是怎样描述流计算过程的

首先，我们看下大名鼎鼎的 Spark 大数据框架。在 Spark 中，计算步骤是被描述为有向无环图的，也就是我们常说的 DAG 。在 Spark 的 DAG 中，节点代表了数据（ RDD ，弹性分布式数据集），边则代表转换函数。

下面的图 1 是 Spark 将 DAG 分解为运行时任务的过程。我们可以看出，最左边的 RDD1 到 RDD4 ，以及表示这些 RDDs 之间依赖关系的有向线段，共同构成了一个 DAG 有向无环图。

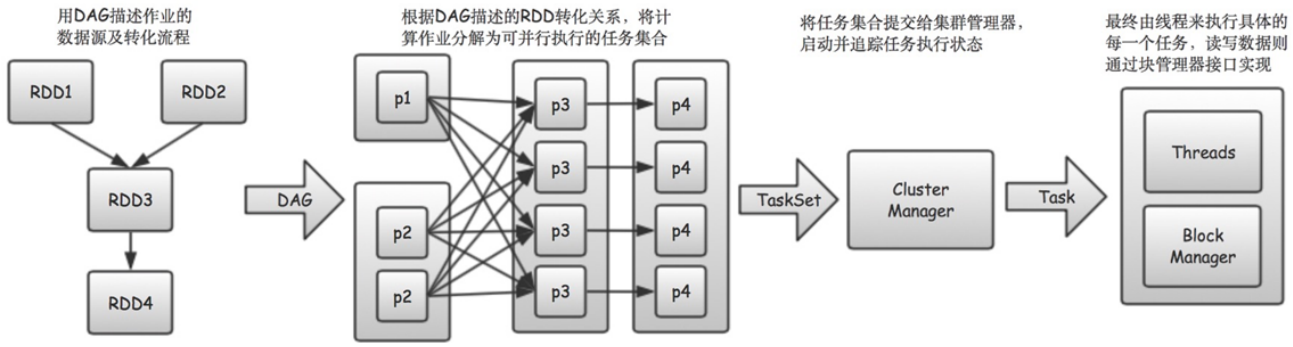


图 1 Spark 中将 DAG 分解为运行时任务的过程

我们可以看到，Spark 是这样将 DAG 解析为最终执行的任务的。首先，DAG 被分解成一系列有依赖关系的并行计算任务集合。然后，这些任务集合被提交到 Spark 集群，再由分配的线程，执行具体的每一个任务。

看完 Spark，我们再来看另外一个最近更加火爆的流计算框架 Flink。在 Flink中，我们是采用了 JobGraph 这个概念，来描述流计算的过程的。下图 2 是 Flink 将 JobGraph 分解为运行时的任务的过程，这幅图来自 Flink 的官方文档 。

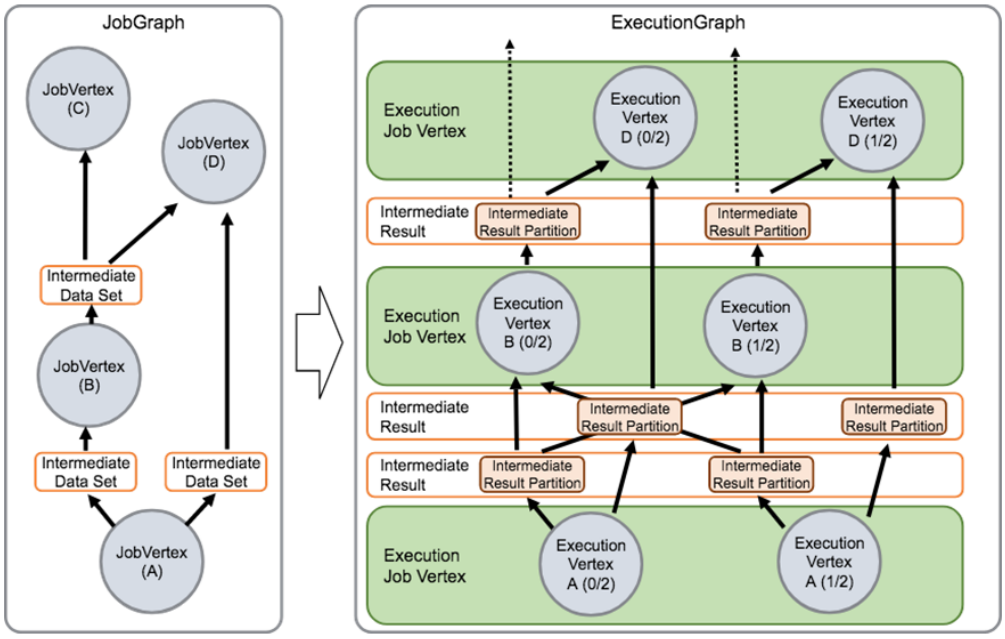


图 2 Flink 中将 DAG 分解为运行时任务的过程

我们很容易看出，左边的 JobGraph 不就是 DAG 有向无环图嘛！其中 JobVertex A 到 JobVertex D，以及表示它们之间依赖关系的有向线段，共同构成了 DAG 有向无环图。这个 DAG 被分解成右边一个个并行且有依赖关系的计算节点，这相当于原始 DAG 的并行化版本。之后在运行时，就是按照这个并行化版本的 DAG 分配线程并执行计算任务。

> 上面介绍的两种流计算框架具体是怎样解析 DAG 的，在本课时你可以暂时不必关心这些细节，只需要知道业界一般都是采用 DAG 来描述流计算过程即可。像其他的一些开源流计算框架，比如 Storm 和 Samza 也有类似的 DAG 概念，这里因为篇幅原因就不一一详细讲解了。

综合这些实例我们可以看出，在业界大家通常都是用 DAG 来描述流计算过程的。

用 DAG 描述流计算过程

所以，接下来我们实现自己的流计算框架，也同样采用了 DAG（有向无环图）来描述流的执行过程。如下图 3 所示。

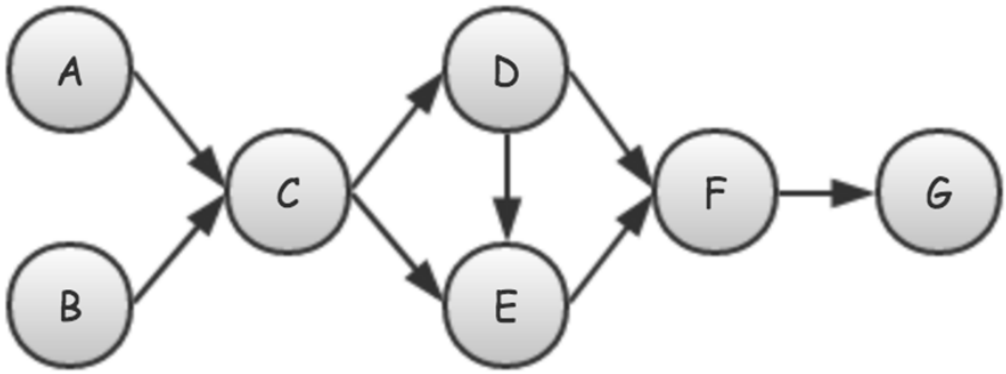


图3 代表流计算过程的有向无环图 DAG

这里，我们对 DAG 的概念稍微做些总结。可以看到上面这个 DAG 图，是由两种元素组成，也就是代表节点的圆圈，和代表节点间依赖关系的有向线段。

DAG 有以下两种不同的表达含义。

- 一是，如果不考虑并行度，那么每个节点表示的是计算步骤，每条边表示的是数据在计算步骤之间的流动，比如图 3 中的 A->C->D。

- 二是，如果考虑并行度，那么每个节点表示的是计算单元，每条边表示的是，数据在计算单元间的流动。这个就相当于将表示计算步骤的 DAG 进行并行化任务分解后，形成的并行化版本 DAG。

上面这样讲可能会有些抽象，下面我们用一个具体的流计算应用场景，来进行更加详细地讲解。

在风控场景中，我们的核心是风控模型和作为模型输入的特征向量。这里我们重点讨论下，如何计算**特征向量的问题。**

在通常的风控模型中，特征向量可能包含几十个甚至上百个特征值，所以为了实现实时风控的效果，需要**并行地计算这些特征值**。否则，如果依次串行计算上百个特征值的话，即使一个特征只需要 100ms，100 个特征计算完也要 10 秒钟了。这样就比较影响用户体验，毕竟刷个二维码还要再等 10 秒钟才能付款，这就很恼人了。

为了实现并行提取特征值的目的，我们设计了下图 4 所示的，特征提取流计算过程 DAG。

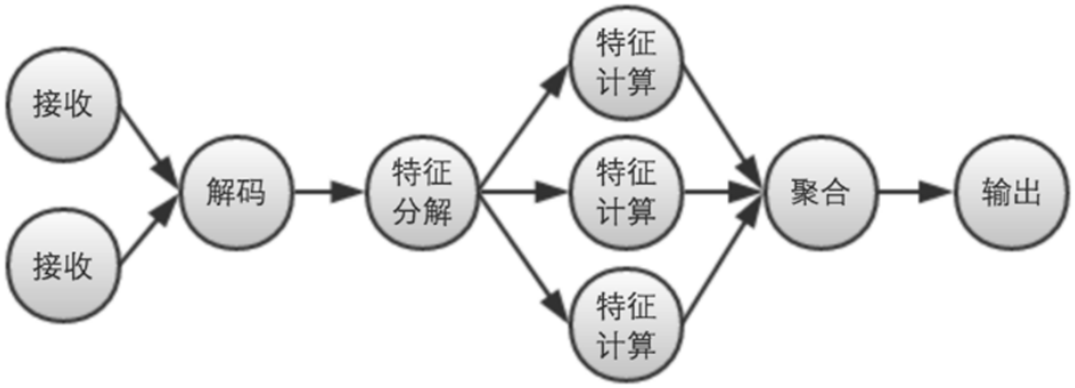


图4 描述风控特征计算过程的 DAG

在上面的图 4 中，假设风控事件先是存放在 Kafka 消息队列里。现在，我们先用两个“接收”节点，将消息从 Kafka 中拉取出来。然后，发送给一个“解码”节点，将事件反序列化为 JSON 对象。接下来，根据风控模型定义的特征向量，将这个 JSON 对象进行“特征分解”为需要并行执行的“特征计算”任务。当所有“特征计算”完成后，再将所有结果“聚合”起来，这样就构成了完整的特征向量。最后，我们就可以将包含了特征向量的事件，“输出”到下游的风险评分模块。

很显然，这里我们采用的是前面所说的第二种 DAG 含义，即并行化的 DAG。

接下来，我们就需要看具体如何，实现这个并行化的 DAG 。看着图 4 这个 DAG，我们很容易想到，可以给每个节点分配一个线程，来执行具体的计算任务。而在节点之间，就用队列（Queue），来作为线程之间传递数据的载体。

具体而言，就是类似于下图 5 所描述的过程。一组线程从其输入队列中取出数据进行处理，然后输出给下游的输入队列，供下游的线程继续读取并处理。

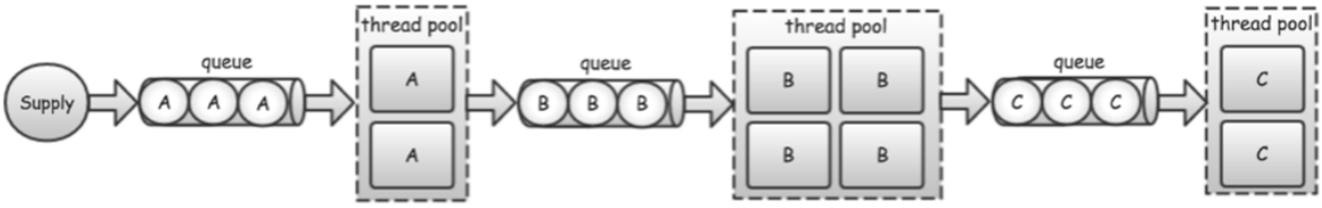


图5 流计算的执行模式

看到这里，你对用 DAG 描述流计算过程，是不是已经做到“心中有丘壑”了？接下来，我们就将心中的丘壑真真实实画出来，做成一幅看得见摸得着的山水画。

用线程和队列实现 DAG

前面说到，我们准备用线程来实现 DAG 的节点，也就是计算步骤或计算单元，具体实现如下面的代码所示。需要注意的是，我这里为了限制篇幅和过滤无效信息，只保留代码的主体部分，对于一些不影响整体理解的代码分支和变量申明等做了删减。本课时的完整代码可以看这里 。

```
public abstract class AbstractStreamService<I, O> {
    private List<Queue<I>> inputQueues;
    private List<Queue<O>> outputQueues;

    private boolean pipeline() throws Exception {
        List<I> inputs = poll(inputQueues);
        List<O> outputs = process(inputs);
        offer(outputQueues, outputs)
    }

    @Override
    public void start() {
        thread = new Thread(() -> {
                while (!stopped) {
                        pipeline()
                }
        });
        thread.start();
    }
}
```

在上面的代码中，我定义了一个抽象类 AbstractStreamService。它的功能是从其输入队列，也就是 inputQueues 中，拉取（poll）消息，然后经过处理（process）后，发送到下游的输入队列，也就是 outputQueues 中去。

在 AbstractStreamService 中，为了在线程和线程之间传输数据，也就是实现 DAG 中节点和节点之间的有向线段，我们还需要定义消息传递的载体，也就是队列 Queue 接口，具体定义如下：

```
public interface Queue<E> {
    E poll(long timeout, TimeUnit unit) throws InterruptedException;
    boolean offer(E e, long timeout, TimeUnit unit) throws InterruptedException;
}
```

上面的接口定义了两个方法，其中 offer 用于上游的节点向下游的节点传递数据，poll 则用于下游的节点向上游的节点拉取数据。

现在，用于描述 DAG 节点的 AbstractStreamService 类，和用于描述 DAG 有向线段的 Queue 接口，都已经定义清楚。接下来就只需要将它们按照 DAG 的各个节点和有向线段组合起来，就可以构成一个完整的流计算过程了。

但这里还有个问题，上面流计算过程没有实现流的“分叉”（Fork）和“聚合”（Join）。而“分叉”和“聚合”的操作，在流计算过程中又是非常频繁出现的。所以，这里我们对问题稍微做些转化，即借用 Future 类，来实现这种 Fork/Join 的计算模式。

我们先看分叉（ Fork ）的实现。

```
private class ExtractorRunnable implements Runnable {
    @Override
    public void run() {
        JSONObject result = doFeatureExtract(event, feature);
        future.set(result);
    }
}

private ListenableFuture<List<JSONObject>> fork(final JSONObject event) {
    List<SettableFuture<JSONObject>> futures = new ArrayList<>();
    final String[] features = {"feature1", "feature2", "feature3"};
    for (String feature : features) {
        SettableFuture<JSONObject> future = SettableFuture.create();
        executorService.execute(new ExtractorRunnable(event, feature, future));
        futures.add(future);
    }
    return Futures.allAsList(futures);
}
```

在上面的代码中，Fork 方法将事件需要提取的特征，分解为多个任务（用 ExtractorRunnable 类表示），并将这些任务提交给专门进行特征提取的执行器（ExecutorService）执行。执行的结果用一个 List<SettableFuture<JSONObject>> 对象来表示，然后通过 Futures.allAsList 将这些 SettableFuture 对象，封装成了一个包含所有特征计算结果的 ListenableFuture<List<JSONObject>> 对象。这样，我们就非常方便地，完成了特征的分解和并行计算。并且，我们得到了一个用于在之后获取所有特征计算结果的 ListenableFuture 对象。

接下来就是聚合（Join）的实现了。

```
private JSONObject join(final ListenableFuture<List<JSONObject>> future) {
    List<JSONObject> features = future.get(extractTimeout, TimeUnit.MILLISECONDS);
    JSONObject featureJson = new JSONObject();
    for (JSONObject feature : features) {
        featureJson.putAll(feature);
    }
    event.put("features", featureJson);
    return event
```

在上面的代码中，由于在 Fork 时已经将所有特征计算的结果，用 ListenableFuture<List<JSONObject>> 对象封装起来，故而在 Join 方法中，用 future.get() 就可以获取所有特征计算结果。而且，为了保证能够在一定的时间内，结束对这条消息的处理，我们还指定了超时时间，也就是 extractTimeout。

当收集了所有的特征后，将它们添加到消息 JSON 对象的 features 字段。至此，我们也就完成了完整特征向量的全部计算过程。

让流计算框架稳定可靠

接下来，我们整体分析下这个风控特征计算过程的 DAG，在实际运行起来时有什么特点。

首先，DAG 中的每个节点都是通过队列隔离开的，每个节点运行的线程都是相互独立的互不干扰，这正是“异步”系统最典型的特征。

然后就是，节点和节点之间的队列，我们并没指定其容量是有限还是无限的，以及是阻塞的还是非阻塞的，这在实际生产环境中会造成一个比较严重的问题。

我们回顾下图 4 所示的风控特征计算过程 DAG，如果“特征计算”节点较慢，而数据“接收”和“解码”节点又很快的话，会出现什么情况呢？毫无疑问，如果没有“反向压力”，数据就会不断地在“队列“中积累起来，直到最终 JVM 内存耗尽，抛出 OOM 异常，程序崩溃退

出。

事实上，由于 DAG 中所有上下游节点之间都是独立运行的，所以这种上下游之间速度不一致的情况随处可见。如果不处理好“反向压力”的问题，系统时时刻刻都有着 OOM 的危险。

所以，那我们应该怎样在流计算框架中加入“反向压力”的能力呢？其实也很简单，只需在实现队列 Queue 接口时，使用容量有限且带阻塞功能的队列即可，比如像下面这样。

```
public class BackPressureQueue<E> extends ArrayBlockingQueue<E> implements Queue<E>{
    public ArrayBlockingQueuePipe(int capacity) {
        super(capacity);
    }
}
```

可以看出，我们实现的 BackPressureQueue 是基于 ArrayBlockingQueue 的。也就是说，它的容量是有限的，而且是一个阻塞队列。这样当下游比上游的处理速度更慢时，数据在队列里积压起来。而当队列里积压的数据达到队列的容量上限时，就会阻塞上游继续往这个队列写入数据。从而，上游也就自动减慢了自己的处理速度。

至此，我们就实现了一个流计算框架，并且这个框架支持反向压力，在生产环境能够安全平稳地运行。

小结

今天，我们用最基础的线程（Thread）和阻塞队列（ArrayBlockingQueue）实现了一个简单的流计算框架。麻雀虽小，但五脏俱全。我们可以从中了解到一个流计算框架的基本骨架，也就是用于传输流数据的队列，以及用于处理流数据的线程。

这个框架足够我们做一些业务逻辑不太复杂的功能模块，但是它有以下问题。

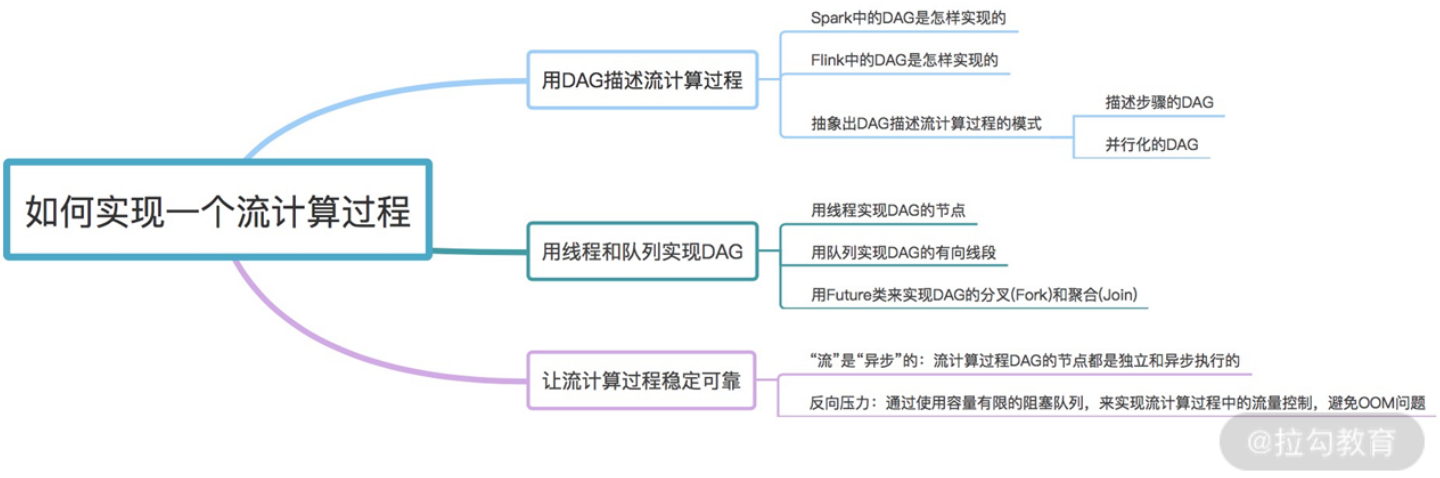
- 一是，能够实现的 DAG 拓扑结构有限。比如，在实现 Fork/Join 功能时，我们还需要借助 SettableFuture和ListenableFuture 的功能，这样对于实现一个 DAG 拓扑来说，并不纯粹和优雅。
- 二是，给每个节点的计算资源只能静态配置，不能根据实际运行时的状况动态分配计算资源。

为了解决这些问题，在接下来的课时中，我们将采用 Java 8 中初次登场的 CompletableFuture 类，来对这个流计算框架进行改造。

到时候，我们将会得到一个更加简洁，但功能更强大的流计算框架。并且我们将能够更加深刻地理解异步系统和流计算系统之间的关联关系。

那么，在学完今天的课程后，你还有什么疑问呢？可以将你的问题放到留言区，我会时刻关注，并在后续文章为你解答哦！

> 本课时精华：



点击此链接查看本课程所有课时的源码

PB 级企业大数据项目实战 + 拉勾硬核内推，5 个月全面掌握大数据核心技能。点击链接，全面赋能！