04 | DAG与流水线：到底啥叫“内存计算”?

2021-03-22 吴磊

进入课程 >

讲述：吴磊

时长 16:15　大小 14.90M

你好，我是吴磊。

在日常的开发工作中，我发现有两种现象很普遍。

第一种是缓存的滥用。无论是 RDD，还是 DataFrame，凡是能产生数据集的地方，开发同学一律用 cache 进行缓存，结果就是应用的执行性能奇差无比。开发同学也很委屈：“Spark 不是内存计算的吗？为什么把数据缓存到内存里去，性能反而更差了？”

第二种现象是关于 Shuffle 的。我们都知道，Shuffle 是 Spark 中的性能杀手，在开发用时要尽可能地避免 Shuffle 操作。不过据我观察，很多初学者都没有足够的动力去重构代码来避免 Shuffle，这些同学的想法往往是：“能把业务功能实现就不错了，费了半天劲去重写代码就算真的消除了 Shuffle，能有多大的性能收益啊。”

以上这两种现象可能大多数人并不在意，但往往这些细节才决定了应用执行性能的优劣。在我看来，造成这两种现象的根本原因就在于，开发者对 Spark 内存计算的理解还不够透彻。所以今天，我们就来说说 Spark 的内存计算都有哪些含义？

第一层含义：分布式数据缓存

一提起 Spark 的“内存计算”的含义，你的第一反应很可能是：Spark 允许开发者将分布式数据集缓存到计算节点的内存中，从而对其进行高效的数据访问。没错，这就是内存计算的**第一层含义：众所周知的分布式数据缓存。**

RDD cache 确实是 Spark 分布式计算引擎的一大亮点，也是对业务应用进行性能调优的诸多利器之一，很多技术博客甚至是 Spark 官网，都在不厌其烦地强调 RDD cache 对于应用执行性能的重要性。

正因为考虑到这些因素，很多开发者才会在代码中不假思索地滥用 cache 机制，也就是我们刚刚提到的第一个现象。但是，这些同学都忽略了一个重要的细节：只有需要频繁访问的数据集才有必要 cache，对于一次性访问的数据集，cache 不但不能提升执行效率，反而会产生额外的性能开销，让结果适得其反。

之所以会忽略这么重要的细节，背后深层次的原因在于，开发者对内存计算的理解仅仅停留在缓存这个层面。因此，当业务应用的执行性能出现问题时，只好死马当活马医，拼命地抓住 cache 这根救命稻草，结果反而越陷越深。

接下来，我们就重点说说内存计算的第二层含义：**Stage 内部的流水线式计算模式。**

在 Spark 中，内存计算有两层含义：第一层含义就是众所周知的分布式数据缓存，第二层含义是 Stage 内的流水线式计算模式。关于 RDD 缓存的工作原理，我会在后续的课程中为你详细介绍，今天咱们重点关注内存计算的第二层含义就可以了。

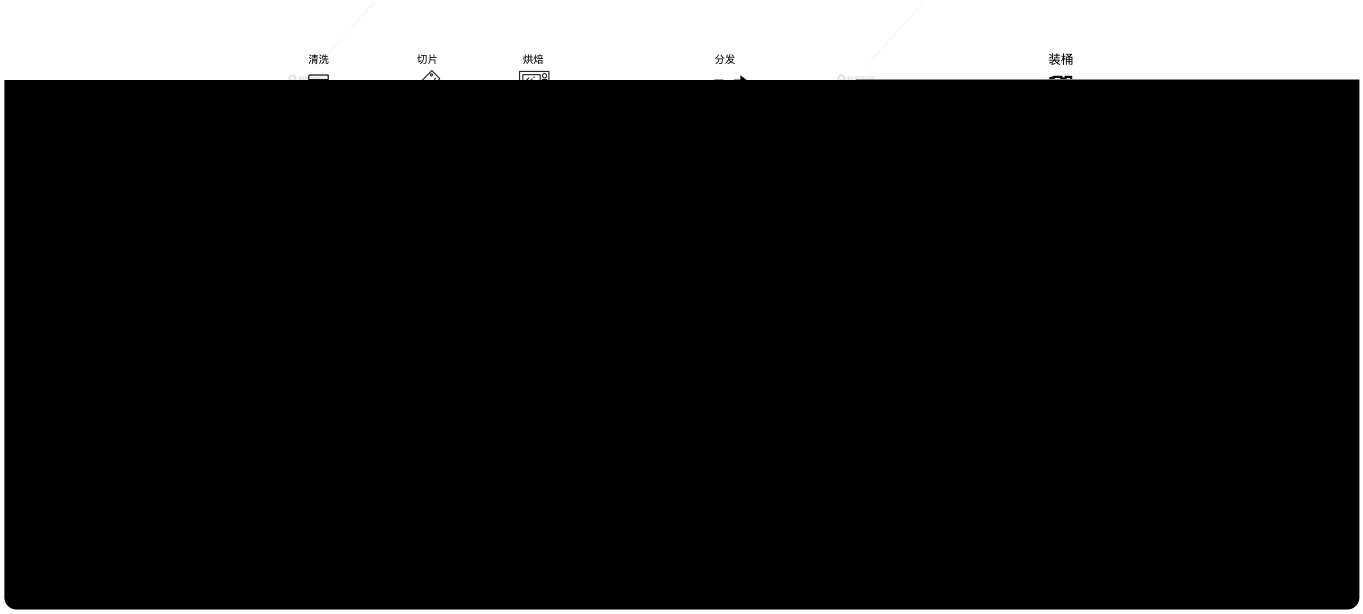
第二层含义：Stage 内的流水线式计算模式

很显然，要弄清楚内存计算的第二层含义，咱们得从 DAG 的 Stages 划分说起。在这之前，我们先来说说什么是 DAG。

什么是 DAG?

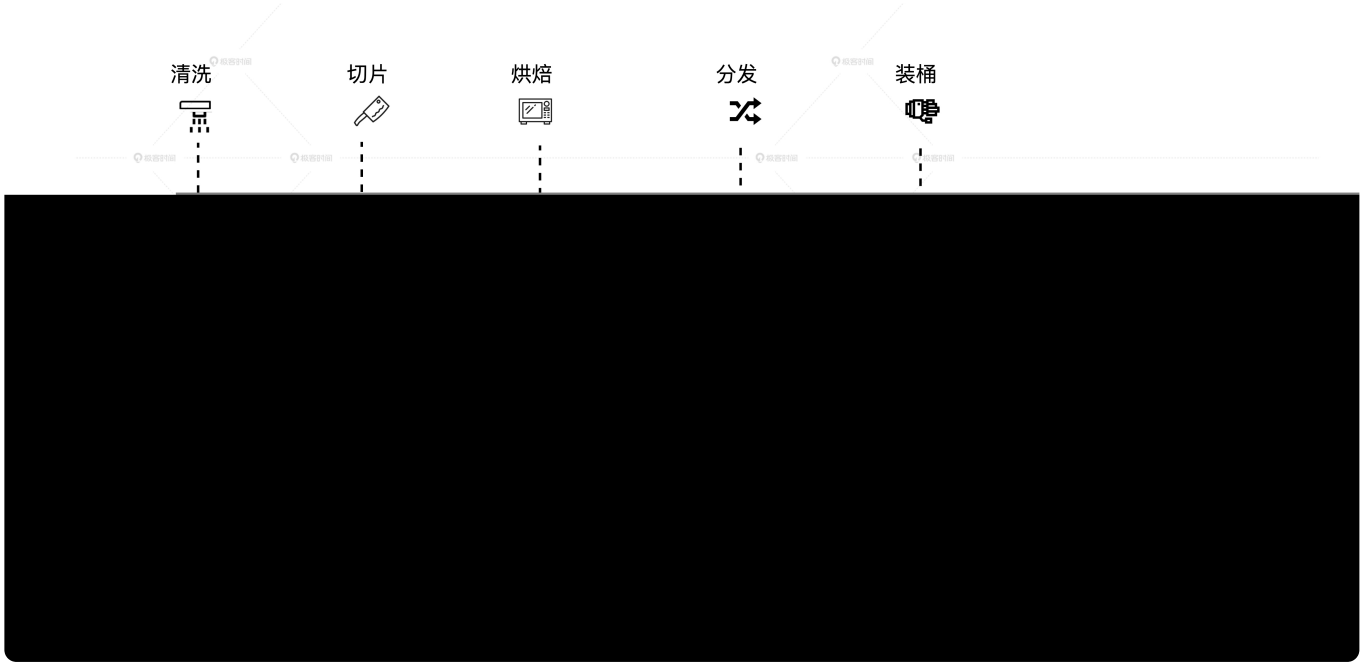
DAG 全称 Direct Acyclic Graph，中文叫有向无环图。顾名思义，DAG 是一种“图”。我们知道，任何一种图都包含两种基本元素：顶点（Vertex）和边（Edge），顶点通常用于表示实体，而边则代表实体间的关系。**在 Spark 的 DAG 中，顶点是一个个 RDD，边则是 RDD 之间通过 dependencies 属性构成的父子关系。**

从理论切入去讲解 DAG，未免枯燥乏味，所以我打算借助上一讲土豆工坊的例子，来帮助你直观地认识 DAG。上一讲，土豆工坊成功地实现了同时生产 3 种不同尺寸的桶装“原味”薯片。但是，在将“原味”薯片推向市场一段时间以后，工坊老板发现季度销量直线下滑，不由得火往上撞、心急如焚。此时，工坊的工头儿向他建议：“老板，咱们何不把流水线稍加改造，推出不同风味的薯片，去迎合市场大众的多样化选择？”然后，工头儿把改装后的效果图交给老板，老板看后甚是满意。



土豆工坊流水线效果图

不过，改造流水线可是个大工程，为了让改装工人能够高效协作，工头儿得把上面的改造设想抽象成一张施工流程图。有了这张蓝图，工头儿才能给负责改装的工人们分工，大伙儿才能拧成一股绳、劲儿往一处使。在上一讲中，我们把食材形态类比成 RDD，把相邻食材形态的关系看作是 RDD 间的依赖，那么显然，流水线的施工流程图就是 DAG。



DAG：土豆工坊流水线的设计流程图

因为 DAG 中的每一个顶点都由 RDD 构成，对应到上图中就是带泥的土豆 potatosRDD，清洗过的土豆 cleanedPotatosRDD，以及调料粉 flavoursRDD 等等。DAG 的边则标记了不同 RDD 之间的依赖与转换关系。很明显，上图中 DAG 的每一条边都有指向性，而且整张图不存在环结构。

那 DAG 是怎么生成的呢？

我们都知道，在 Spark 的开发模型下，应用开发实际上就是灵活运用算子实现业务逻辑的过程。开发者在分布式数据集如 RDD、 DataFrame 或 Dataset 之上调用算子、封装计算逻辑，这个过程会衍生新的子 RDD。与此同时，子 RDD 会把 dependencies 属性赋值到父 RDD，把 compute 属性赋值到算子封装的计算逻辑。以此类推，在子 RDD 之上，开发者还会继续调用其他算子，衍生出新的 RDD，如此往复便有了 DAG。

因此，**从开发者的视角出发，DAG 的构建是通过在分布式数据集上不停地调用算子来完成的。**

Stages 的划分

现在，我们知道了什么是 DAG，以及 DAG 是如何构建的。不过，DAG 毕竟只是一张流程图，Spark 需要把这张流程图转化成分布式任务，才能充分利用分布式集群并行计算的优势。这就好比土豆工坊的施工流程图毕竟还只是蓝图，是工头儿给老板画的一张“饼”，

工头儿得想方设法把它转化成实实在在的土豆加工流水线，让流水线能够源源不断地生产不同风味的薯片，才能解决老板的燃眉之急。

简单地说，从开发者构建 DAG，到 DAG 转化的分布式任务在分布式环境中执行，其间会经历如下 4 个阶段：

回溯 DAG 并划分 Stages

在 Stages 中创建分布式任务

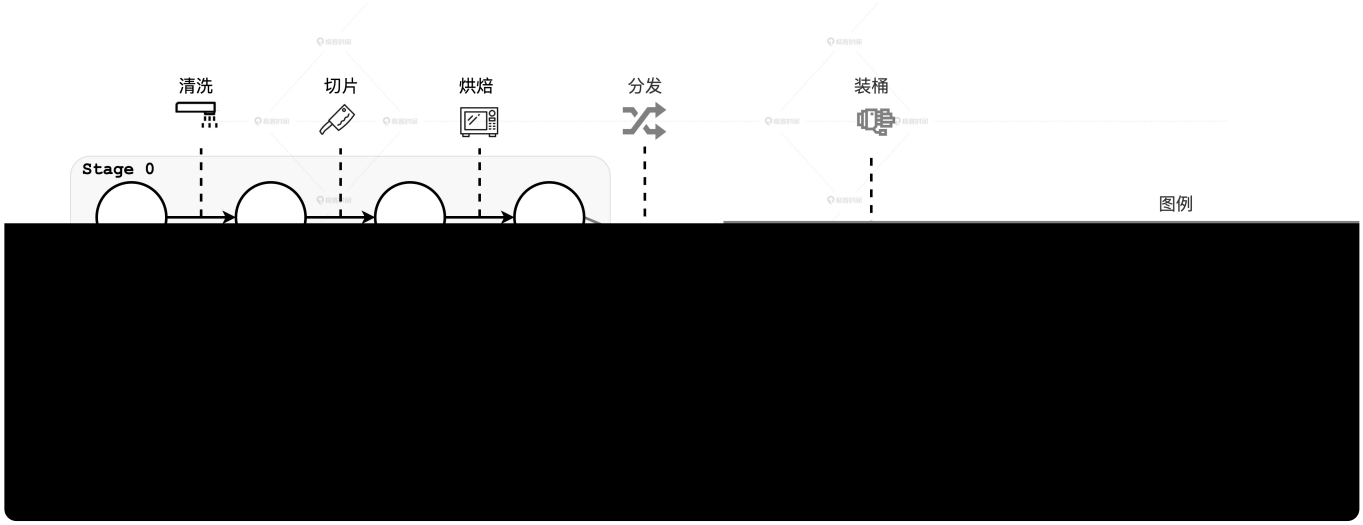
分布式任务的分发

分布式任务的执行

刚才我们说了，内存计算的第二层含义在 stages 内部，因此这一讲我们只要搞清楚 DAG 是怎么划分 Stages 就够了。至于后面的 3 个阶段更偏向调度系统的范畴，所以我会在下一讲给你讲清楚其中的来龙去脉。

如果用一句话来概括从 DAG 到 Stages 的转化过程，那应该是：**以 Actions 算子为起点，从后向前回溯 DAG，以 Shuffle 操作为边界去划分 Stages**。

接下来，我们还是以土豆工坊为例来详细说说这个过程。既然 DAG 是以 Shuffle 为边界去划分 Stages，我们不妨先从上帝视角出发，看看在土豆工坊设计流程图的 DAG 中，都有哪些地方需要执行数据分发的操作。当然，在土豆工坊，数据就是各种形态的土豆和土豆片儿。



DAG以Shuffle为边界划分出3个Stages

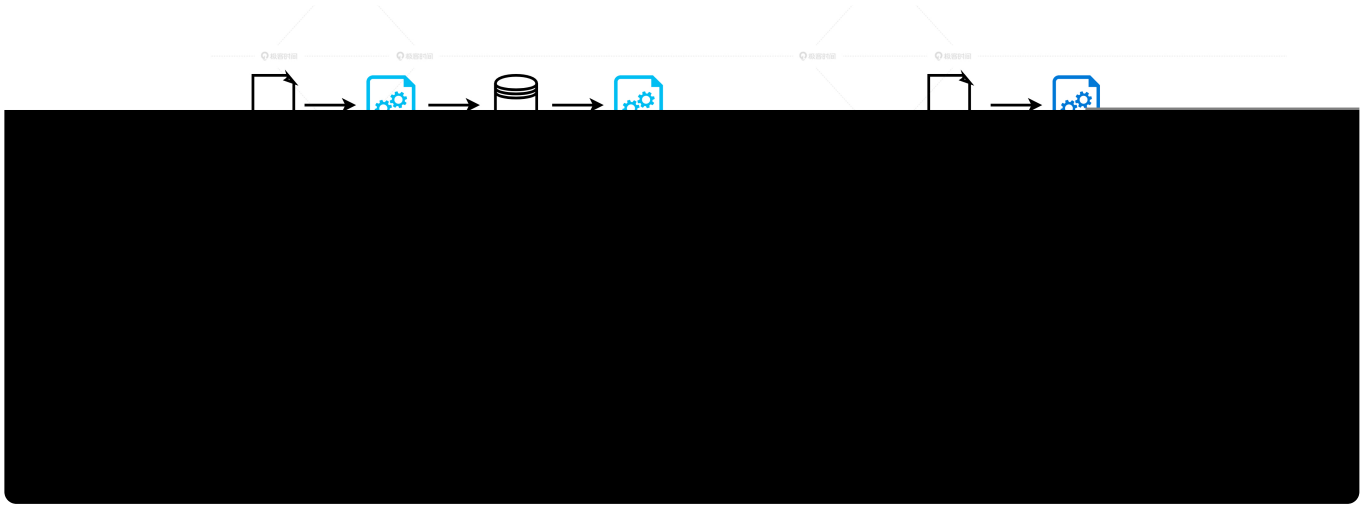
仔细观察上面的设计流程图，我们不难发现，有两个地方需要分发数据。第一个地方是薯片经过烘焙烤熟之后，把即食薯片按尺寸大小分发到下游的流水线上，这些流水线会专门处理固定型号的薯片，也就是图中从 bakedChipsRDD 到 flavouredBakedChipsRDD 的那条线。同理，不同的调料粉也需要按照风味的不同分发到下游的流水线上，用于和固定型号的即食薯片混合，也就是图中从 flavoursRDD 到 flavouredBakedChipsRDD 那条分支。

同时，我们也能发现，土豆工坊的 DAG 应该划分 3 个 Stages 出来，如图中所示。其中，Stage 0 包含四个 RDD，从带泥土豆 potatosRDD 到即食薯片 bakedChipsRDD。Stage 1 比较简单，它只有一个 RDD，就是封装调味粉的 flavoursRDD。Stage 2 包含两个 RDD，一个是加了不同风味的即食薯片 flavouredBakedChipsRDD，另一个表示组装成桶已经准备售卖的桶装薯片 bucketChipsRDD。

你可能会问：“费了半天劲，把 DAG 变成 Stages 有啥用呢？”还真有用！内存计算的第二层含义，就隐匿于从 DAG 划分出的一个又一个 Stages 之中。不过，要弄清楚 Stage 内的流水线式计算模式，我们还是得从 Hadoop MapReduce 的计算模型说起。

Stage 中的内存计算

基于内存的计算模型并不是凭空产生的，而是根据前人的教训和后人的反思精心设计出来的。这个前人就是 Hadoop MapReduce，后人自然就是 Spark。

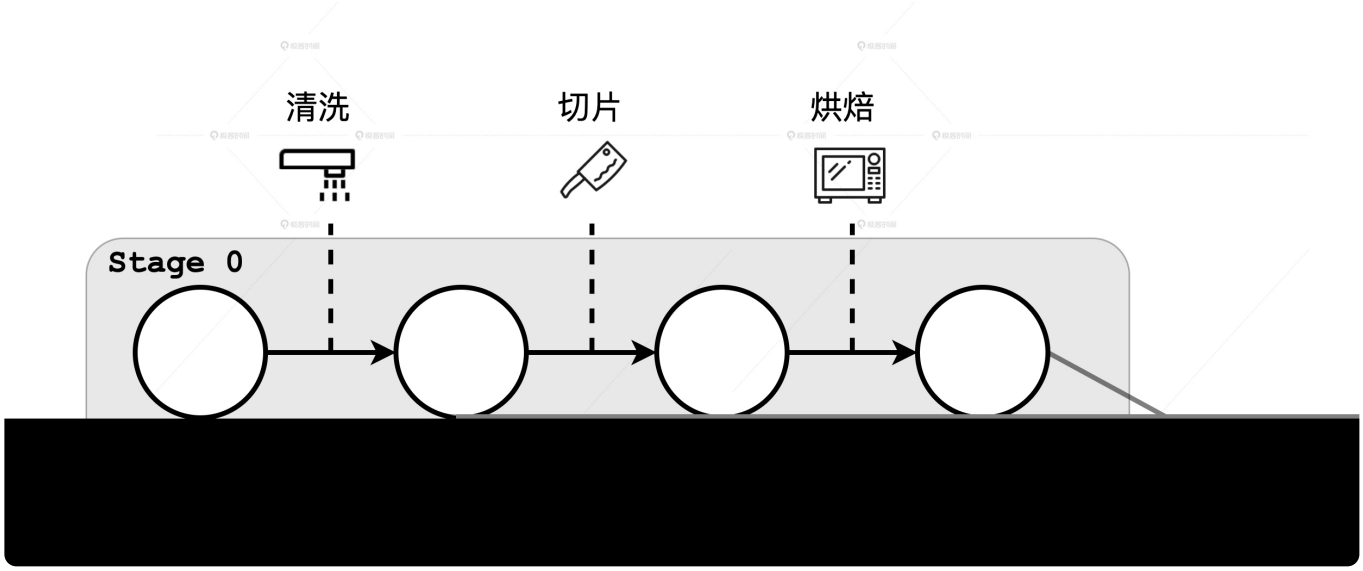


Hadoop MapReduce的计算模型

MapReduce 提供两类计算抽象，分别是 Map 和 Reduce：Map 抽象允许开发者通过实现 map 接口来定义数据处理逻辑；Reduce 抽象则用于封装数据聚合逻辑。MapReduce 计算模型最大的问题在于，所有操作之间的数据交换都以磁盘为媒介。例如，两个 Map 操

作之间的计算通过本地磁盘交换数据，而 Map 与 Reduce 操作之间利用本地磁盘来交换数据。不难想象，这种频繁的磁盘 I/O 必定会拖累用户应用端到端的执行性能。

那么，这和 Stage 内的流水线式计算模式有啥关系呢？我们再回到土豆工坊的例子中，把目光集中在即食薯片分发之前，也就是刚刚划分出来的 Stage 0。这一阶段包含 3 个处理操作，即清洗、切片和烘焙。按常理来说，流水线式的作业方式非常高效，带泥土豆被清洗过后，会沿着流水线被传送到切片机，切完的生薯片会继续沿着流水线再传送到烘焙烤箱，整个过程一气呵成。如果把流水线看作是计算节点内存的话，那么清洗、切片和烘焙这 3 个操作都是在内存中完成计算的。



Stage 0包含清洗、切片、烘焙3个操作

你可能会说：“内存计算也不过如此，跟 MapReduce 相比，不就是把数据和计算都挪到内存里去了吗？”事情可能并没有你想象的那么简单。

在土豆工坊的例子里，Stage 0 中的每个加工环节都会生产出中间食材，如清洗过的土豆、土豆片、即食薯片。我们刚刚把流水线比作内存，这意味着每一个算子计算得到的中间结果都会在内存中缓存一份，以备下一个算子运算，这个过程与开发者在应用代码中滥用 RDD cache 简直如出一辙。如果你曾经也是逢 RDD 便 cache，应该不难想象，采用这种计算模式，Spark 的执行性能不见得比 MapReduce 强多少，尤其是在 Stages 中的算子数量较多的时候。

既然不是简单地把数据和计算挪到内存，那 Stage 内的流水线式计算模式到底长啥样呢？在 Spark 中，**流水线计算模式指的是：在同一 Stage 内部，所有算子融合为一个函数，**

Stage 的输出结果由这个函数一次性作用在输入数据集而产生。这也正是内存计算的第二层含义。下面，我们用一张图来直观地解释这一计算模式。

内存计算的第二层含义

如图所示，在上面的计算流程中，如果你把流水线看作是内存，每一步操作过后都会生成临时数据，如图中的 clean 和 slice，这些临时数据都会缓存在内存里。但在下面的内存计算中，所有操作步骤如 clean、slice、bake，都会被捏合在一起构成一个函数。这个函数一次性地作用在“带泥土豆”上，直接生成“即食薯片”，在内存中不产生任何中间数据形态。

因此你看，所谓内存计算，不仅仅是指数据可以缓存在内存中，更重要的是让我们明白了，通过计算的融合来大幅提升数据在内存中的转换效率，进而从整体上提升应用的执行性能。

这个时候，我们就可以回答开头提出的第二个问题了：费劲去重写代码、消除 Shuffle，能有多大的性能收益?

由于计算的融合只发生在 Stages 内部，而 Shuffle 是切割 Stages 的边界，因此一旦发生 Shuffle，内存计算的代码融合就会中断。但是，当我们对内存计算有了多方位理解以后，就不会一股脑地只想到用 cache 去提升应用的执行性能，而是会更主动地想办法尽量避免 Shuffle，让应用代码中尽可能多的部分融合为一个函数，从而提升计算效率。

小结

这一讲，我们以两个常见的现象为例，探讨了 Spark 内存计算的含义。

在 Spark 中，内存计算有两层含义：第一层含义就是众所周知的分布式数据缓存，第二层含义是 Stage 内的流水线式计算模式。

对于第二层含义，我们需要先搞清楚 DAG 和 Stages 划分，从开发者的视角出发，DAG 的构建是通过在分布式数据集上不停地调用算子来完成的，DAG 以 Actions 算子为起点，从后向前回溯，以 Shuffle 操作为边界，划分出不同的 Stages。

最后，我们归纳出内存计算更完整的第二层含义：同一 Stage 内所有算子融合为一个函数，Stage 的输出结果由这个函数一次性作用在输入数据集而产生。

每日一练

今天的内容重在理解，我希望你能结合下面两道思考题来巩固一下。

1. 我们今天说了，DAG 以 Shuffle 为边界划分 Stages，那你知道 Spark 是根据什么来判断一个操作是否会引入 Shuffle 的呢?
2. 在 Spark 中，同一 Stage 内的所有算子会融合为一个函数。你知道这一步是怎么做到的吗?

期待在留言区看到你的思考和答案，如果对内存计算还有很多困惑，也欢迎你写在留言区，我们下一讲见！

提建议

上一篇 03 | RDD：为什么你必须要理解弹性分布式数据集？

下一篇 05 | 调度系统：“数据不动代码动”到底是什么意思？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。