复习课（二）| MapReduce

2021-11-03 黄金

《大数据经典论文解读》 课程介绍 >

讲述：王惠

时长 08:47 大小 8.05M

你好，我是黄金。欢迎来到第二期复习课，今天我们来回顾 MapReduce 论文的知识点。

MapReduce 介绍

在论文的第 1 节“Introduction”中，作者提到过去 5 年里，Google 的同事们实现了上百个针对特定领域的大数据分析程序，这促使他们思考开发一种更通用的编程模型，让开发者能够专注于分析程序的业务逻辑，而不需要关心分布式领域的复杂问题。

MapReduce 的编程模型也并非是作者首创，而是借鉴了 Lisp 这类函数式编程语言的想。熟悉 Java Stream API 的同学对这种编程模式应该都不陌生，它实际上就是 map、groupingBy、reduce 之类的操作，这种编程模型**分离了程序的业务逻辑和控制逻辑**，使得程序在大规模的分布式环境下运行成为了可能。

另外，尽管 MapReduce 编程模型非常简单，现实中的大多数任务却都可以用这种编程模型来表达，这在函数式编程语言中已经得到了证明，它为 MapReduce 后来广泛地流行奠定了基础。

在论文第 6.1 节“Large-Scale Indexing”中就给出了一个例子，说明用 MapReduce 重写的索引服务带来的显著收益：第一，代码更精简，也更容易理解，原来用来实现某个计算功能的代码有 3800 行，重构后只有 700 行；第二，性能更好，原来改变索引需要几个月，现在只要几天；第三，更容易维护，也更容易提升性能，因为分布式问题都交给了 MapReduce 框架来处理。

那么总结一下，MapReduce 主要有三个特点。**第一，简单的编程模型；第二，丰富的表达能力；第三，能够有效利用分布式系统的资源。**

编程模型

使用 MapReduce 编程模型，只需要实现两个函数，一个是 Map 函数，另一个是 Reduce 函数。

Map 函数，是接受一个 key-value 对，返回一组新的 key-value 对，它通常被用来做数据变换。而 Reduce 函数，是接受一个 key，以及一组相关的 value，然后返回一组新的 value，它通常被用来做数据规约，比如分组计数。如果你对于 API 还不够清楚的话，可以参考下面的代码：

复制代码

```
map: (k1, v1) -> list (k2, v2)
reduce: (k2, list(v2)) -> list(v3)
```

执行概览

在论文的第 3.1 节“Execution Overview”中，描述了 MapReduce 的执行过程。

MapReduce 会启动很多个 Worker 程序，有的负责 map 任务，有的负责 reduce 任务，其中有一个很特殊，叫做 **Master**，它负责整个 MapReduce 任务的调度。MapReduce 把输入数据分成了 M 份，交给不同的 map 任务进行处理。每一份大概占用 16~64MB，和 GFS 的 Block 块大小接近，相当于一个 map 任务处理一个或多个 GFS Block 块。

此外，map 任务的处理结果是写在本地的，而 reduce 任务会通过 RPC 请求对应的机器，获取 map 任务的输出。map 到 reduce 会经过一个**混洗**的过程，任何一个 map 任务的输出都可能被多个 reduce 任务读取。那么，**map 任务的输出如何对应多个 reduce 任务的输入呢？**

map 任务在执行的时候知道存在多少个 reduce 任务，它的输出会按分区函数分成 R 份，每一份对应着一个 reduce 任务的输入。当所有的 reduce 任务执行完成之后，Master 就会通知客户端任务结束。

那么，通过 MapReduce 的执行过程，我们可以看到数据的 Block 会分散在不同机器上，map 和 reduce 的计算任务也分散在不同机器上，从而就有效利用了分布式系统的资源。

容错设计

分布式系统的局部出现故障是难以避免的事情，让我们来看看 MapReduce 都做了哪些设计，来保障系统的可靠性。

MapReduce 程序由多个工作进程构成，进程可以分成两类，一类是 **Master 进程**，只有一个，负责整个 MapReduce 任务的调度。另一类是 **Worker 进程**，负责执行 map 任务和 reduce 任务。Worker 进程只和 Master 进程交互，Worker 和 Worker 进程之间没有交互。

需要注意的是，前面提到混洗过程 map 任务和 reduce 任务之间交互，并不是通过 Worker 和 Worker 进程交互完成的，而是绕过 Worker 进程，通过 RPC 服务交互。Worker 进程的这种独立性使得当 Worker 进程崩溃的时候，只需要由 Master 进程重启，并重新运行其负责的 map 和 reduce 任务即可。

Master 进程会掌握执行计划，通过心跳来获得 Worker 进程的执行进度和状态。如果我们把 Master 进程的状态保存下来，就很容易让 Master 进程从故障中恢复。

不过，在 MapReduce 的论文中，作者并没有实现这一点，这是因为作者认为 Master 进程是单实例，短周期执行，不太可能在运行的过程中出现故障，所以当 Master 进程出错时，作者希望用户重试 MapReduce 任务。

Master 会通过心跳获悉 Worker 进程是否还在工作，不过当 Master 进程和 Worker 进程点之间的网络出现故障时，Master 其实会产生误判，认为还在工作的 Worker 进程停止了工作，然后在其他地方启动了一个新的 Worker 进程，执行相同的任务。

那么，在这种故障恢复的机制下，系统就可能同时存在多个处理相同输入的 map 任务或 reduce 任务。当 map 任务和 reduce 任务执行完成后，Worker 进程向 Master 进程提交任务的过程就必须是**原子性**的，只有这样，当 Master 收到一个任务结束的消息之后，才能忽略其他相同任务结束的消息。

性能优化

除此之外，徐老师在07 讲中还提到过，MapReduce 框架有两个和网络相关的性能优化，一个是**让计算靠近存储**，另一个是**通过 Combiner 减少混洗阶段需要传输的数据**。

计算靠近存储在论文中被称为 Locality Optimization，从论文的“Figure 3”中我们可以看到 Locality Optimization 的威力，由于大部分数据可以从本地读取，它的数据读取速度会是写出速度的 2~3 倍。

然后，论文里还有一个优化，叫做 **Backup Tasks**。在分布式的环境下，会出现一些执行很慢的长尾任务，比如论文的第 5.4 节“Effect of Backup Tasks”中提到，在一个 MapReduce 程序中，程序已经运行了 960 秒，剩下 5 个 reduce 任务还未执行完，而这 5 个任务又花了 300 秒，导致程序的运行时间增加了 44%。

这 5 个任务就是长尾任务，它们执行很慢不是因为需要大量的计算，而是由于所在机器的网络、硬盘等资源突然出现了问题，换一台机器执行其实就能很快完成。那么 MapReduce 的优化方法，正是在其他地方启动一模一样的后备任务，处理相同的输入，谁先完成就以谁的结果为准，这样就比较好地消除了长尾现象。

遗憾与缺陷

在 07 讲的最后，徐老师讲到了 MapReduce 框架有**两个缺陷**。一个是它还没有 100% 做到让用户意识不到“分布式”的存在，二个是性能还是不太理想，Worker 进程初始化时间太长，需要写到文件系统的数据太多。

Worker 进程初始化时间到底有多长呢？在论文的第 5.2 节“Grep”中，就给出了一份数据，这个任务总耗时是 150 秒，其中初始化 Worker 进程的时间是 60 秒，足足占了 40% 的时间。

虽然 MapReduce 框架还有不少缺陷，不过也正是因为这些缺陷的存在，才给了后来者机会，在以后的经典论文中，我们将会看到工程师们是如何激动地弥补这些缺陷的。

好了，针对课程中，MapReduce 论文的解读和学习的总结与回顾就到这里了，我们下节复习课再见。

上一篇 复习课（一）| The Google File System

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。