觉得前端不需要懂算法？

算法是问题的解决步骤，同一个问题可以有多种解决思路，也就会有多种算法，但是算法之间是有好坏之分的，区分标志就是复杂度。

通过复杂度可以估算出耗时/内存占用等性能的好坏，所以我们用复杂度来评价算法。

（不了解复杂度可以看这篇：性能分析不一定得用 Profiler，复杂度分析也行）

开发的时候，大多数场景下我们用最朴素的思路，也就是复杂度比较高的算法也没啥问题，好像也用不到各种高大上的算法，算法这个东西似乎可学可不学。

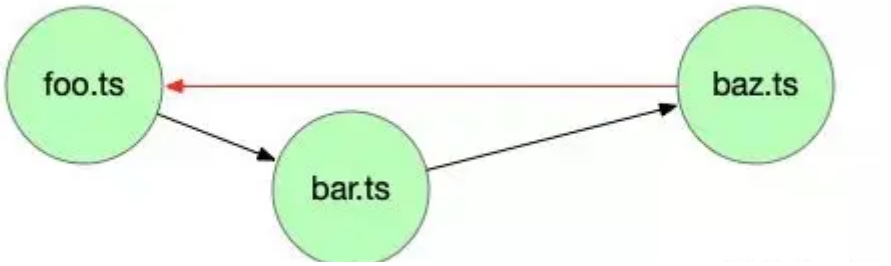
其实不是的，那是因为你没有遇到一些数据量大的场景。

下面我给你举一个我之前公司的具体场景的例子：

体现算法威力的例子

这是我前公司高德真实的例子。

我们会做全源码的依赖分析，会有几万个模块，一个模块依赖另一个模块叫做正向依赖，一个模块被另一个模块依赖叫做反向依赖。我们会先分析一遍正向依赖，然后再分析一遍反向依赖。



分析反向依赖的时候，之前的思路是这样的，对于每一个依赖，都遍历一边所有的模块，找到依赖它的模块，这就是它的反向依赖。

这个思路是很朴素的，容易想到的思路，但是这个思路有没有问题呢？

这个算法的复杂度是 O(n^2)，如果 n 达到了十几万，那性能会很差的，从复杂度我们就可以估算出来。

事实上也确实是这样，后来我们跑一遍全源码依赖需要用 10 几个小时，甚至一晚上都跑不出来。

如果让你去优化，你会怎么优化性能呢？

有的同学可能会说，能不能拆成多进程/多个工作线程，把依赖分析的任务拆成几部分来做，这样能得到几倍的性能提升。

是，几倍的提升很大了。

但是如果说我们后来做了一个改动，性能直接提升了几万倍你信么？

我们的改动方式是这样的：

之前是在分析反向依赖的时候每一个依赖都要遍历一遍所有的正向依赖。但其实正向依赖反过来不就是反向依赖么？

所以我们直接改成了分析正向依赖的时候同时记录反向依赖。

这样根本就不需要单独分析反向依赖了，算法复杂度从 O（n^2）降到了 O(n)。

O(n^2) 到 O(n) 的变化在有几万个模块的时候，就相当于几万倍的性能提升。

这体现在时间上就是我们之前要跑一个晚上的代码，现在十几分钟就跑完了。这优化力度，你觉得光靠多线程/进程来跑能做到么？

这就是算法的威力，当你想到了一个复杂度更低的算法，那就意味着性能有了大幅的提升。

具体的依赖分析的架构可以看这篇文章：

高德 APP 全链路源码依赖分析工程

为什么我们整天说 diff 算法，因为它把 O(n^2) 的朴素算法复杂度降低到了 O(n)，这就意味着 dom 节点有几千个的时候，就会有几千倍的性能提升。

所以，感受到算法的威力了么？

总结

多线程、缓存等手段最多提升几倍的性能，而算法的优化是直接提升数量级的性能，当数据量大了以后，就是几千几万倍的性能提升。

那为什么我们平时觉得算法没用呢？那是因为你处理的数据量太小了，处理几百个数据，你用 O(n^2) O(n^3) 和 O(n) 的算法，都差不了多少。

你处理的场景数据量越大，那算法的重要性越高，因为好的算法和差的算法的差别不是几倍几十倍那么简单，可能是几万倍的差别。

所以，你会见到各大公司都在考算法，没用么？不是的，是太过重要了，直接决定着写出的代码的性能。