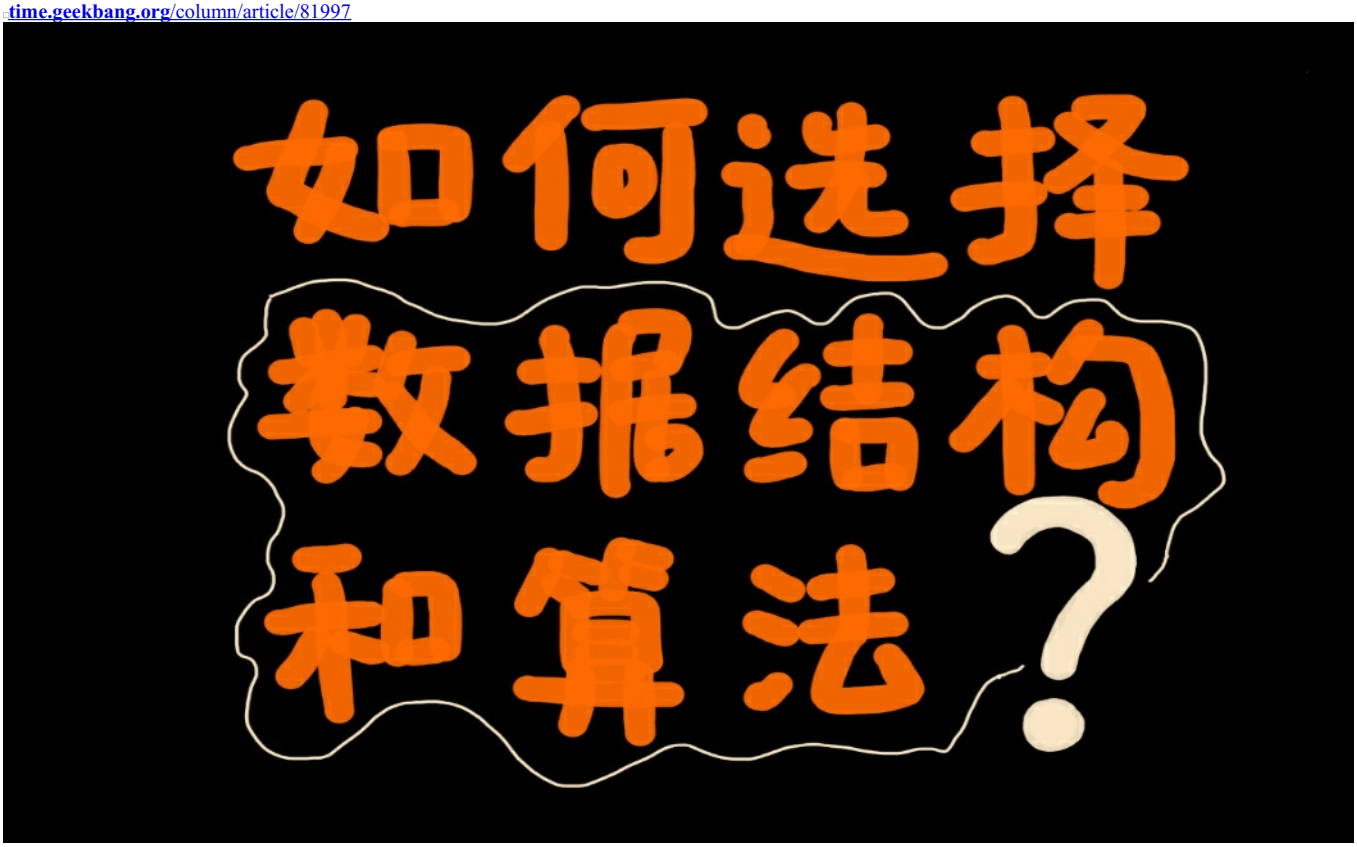
总结课 | 在实际开发中，如何权衡选择使用哪种数据结构和算法？

time.geekbang.org/column/article/81997



你好，我是王争，今天是一篇总结课。我们学了这么多数据结构和算法，在实际开发中，究竟该如何权衡选择使用哪种数据结构和算法呢？今天我们就来聊一聊这个问题，希望能帮你把学习带回实践中。

我一直强调，学习数据结构和算法，不要停留在学院派的思维中，只把算法当作应付面试、考试或者竞赛的花拳绣腿。作为软件开发工程师，我们要把数据结构和算法，应用到软件开发中，解决实际的开发问题。

不过，要想在实际的开发中，灵活、恰到好处地应用数据结构和算法，需要非常深厚的实战经验积累。尽管我在课程中，一直都结合实际的开发场景来讲解，希望带你真枪实弹的演练算法如何解决实际的问题。但是，在今后的软件开发中，你要面对的问题远比我讲的场景要复杂、多变、不确定。

要想游刃有余地解决今后你要面对的问题，光是熟知每种数据结构和算法的功能、特点、时间空间复杂度，还是不够的。毕竟工程上的问题不是算法题。算法题的背景、条件、限制都非常明确，我们只需要在规定的输入、输出下，找最优解就可以了。

而工程上的问题往往都比较开放，在选择数据结构和算法的时候，我们往往需要综合各种因素，比如编码难度、维护成本、数据特征、数据规模等，最终选择一个**工程的最合适解**，而非**理论上的最优解**。

为了让你能做到活学活用，在实际的软件开发中，不生搬硬套数据结构和算法，今天，我们就聊一聊，在实际的软件开发中，如何权衡各种因素，合理地选择使用哪种数据结构和算法？关于这个问题，我总结了六条经验。

1. 时间、空间复杂度不能跟性能划等号

我们在学习每种数据结构和算法的时候，都详细分析了算法的时间复杂度、空间复杂度，但是，在实际的软件开发中，复杂度不能与性能简单划等号，不能表示执行时间和内存消耗的确切数据量。为什么这么说呢？原因有下面几点。

复杂度不是执行时间和内存消耗的精确值

在用大 O 表示法表示复杂度的时候，我们会忽略掉低阶、常数、系数，只保留高阶，并且它的度量单位是语句的执行频度。每条语句的执行时间，并非是相同、确定的。所以，复杂度给出的只能是一个非精确量值的趋势。

代码的执行时间有时不跟时间复杂度成正比

我们常说，时间复杂度是 O(nlogn) 的算法，比时间复杂度是 O(n^2) 的算法，执行效率要高。这样说的一个前提是，算法处理的是大规模数据的情况。对于小规模数据的处理，算法的执行效率并不一定跟时间复杂度成正比，有时还会跟复杂度成反比。

对于处理不同问题的不同算法，其复杂度大小没有可比性

复杂度只能用来表征不同算法，在处理同样的问题，以及同样数据类型的情况下的性能表现。但是，对于不同的问题、不同的数据类型，不同算法之间的复杂度大小并没有可比性。

2. 抛开数据规模谈数据结构和算法都是“耍流氓”

在平时的开发中，在数据规模很小的情况下，普通算法和高级算法之间的性能差距会非常小。如果代码执行频率不高、又不是核心代码，这个时候，我们选择数据结构和算法的主要依据是，其是否简单、容易维护、容易实现。大部分情况下，我们直接用最简单的存储结构和最暴力的算法就可以了。

比如，对于长度在一百以内的字符串匹配，我们直接使用朴素的字符串匹配算法就够了。如果用 KMP、BM 这些更加高效的字符串匹配算法，实际上就大材小用了。因为这对于处理时间是毫秒量级敏感的系统来说，性能的提升并不大。相反，这些高级算法会徒增编码的难度，还容易产生 bug。

3. 结合数据特征和访问方式来选择数据结构

面对实际的软件开发场景，当我们掌握了基础数据结构和算法之后，最考验能力的并不是数据结构和算法本身，而是对问题需求的挖掘、抽象、建模。**如何将一个背景复杂、开放的问题，通过细致的观察、调研、假设，理清楚要处理数据的特征与访问方式，这才是解决问题的重点。**只有理清楚了这些东西，我们才能将问题转化成合理的数据结构模型，进而找到满足需求的算法。

比如我们前面讲过，Trie 树这种数据结构是一种非常高效的字符串匹配算法。但是，如果你要处理的数据，并没有太多的前缀重合，并且字符集很大，显然就不适合利用 Trie 树了。所以，在用 Trie 树之前，我们需要详细地分析数据的特点，甚至还要写些分析代码、测试代码，明确要处理的数据是否适合使用 Trie 树这种数据结构。

再比如，图的表示方式有很多种，邻接矩阵、邻接表、逆邻接表、二元组等等。你面对的场景应该用哪种方式来表示，具体还要看你的数据特征和访问方式。如果每个数据之间联系很少，对应到图中，就是一个稀疏图，就比较适合用邻接表来存储。相反，如果是稠密图，那就比较适合采用邻接矩阵来存储。

4. 区别对待 IO 密集、内存密集和计算密集

如果你要处理的数据存储在磁盘，比如数据库中。那代码的性能瓶颈有可能在磁盘 IO，而并非算法本身。这个时候，你需要合理地选择数据存储格式和存取方式，减少磁盘 IO 的次数。

比如我们在递归那一节讲过最终推荐人的例子。你应该注意到了，当时我给出的代码尽管正确，但其实并不高效。如果某个用户是经过层层推荐才来注册的，那我们获取他的最终推荐人的时候，就需要多次访问数据库，性能显然就不高了。

不过，这个问题解决起来不难。我们知道，某个用户的最终推荐人一旦确定，就不会变动。所以，我们可以离线计算每个用户的最终推荐人，并且保存在表中的某个字段里。当我们要查看某个用户的最终推荐人的时候，访问一次数据库就可以获取到。

刚刚我们讲了数据存储在磁盘的情况，现在我们再来看下，数据存储在内存中的情况。如果你的数据是存储在内存中，那我们还需要考虑，代码是内存密集型的还是 CPU 密集型的。

所谓 CPU 密集型，简单点理解就是，代码执行效率的瓶颈主要在 CPU 执行的效率。我们从内存中读取一次数据，到 CPU 缓存或者寄存器之后，会进行多次频繁的 CPU 计算（比如加减乘除），CPU 计算耗时占大部分。所以，在选择数据结构和算法的时候，要尽量减少逻辑计算的复杂度。比如，用位运算代替加减乘除运算等。

所谓内存密集型，简单点理解就是，代码执行效率的瓶颈在内存数据的存取。对于内存密集型的代码，计算操作都比较简单，比如，字符串比较操作，实际上就是内存密集型的。每次从内存中读取数据之后，我们只需要进行一次简单的比较操作。所以，内存数据的读取速度，是字符串比较操作的瓶颈。因此，在选择数据结构和算法的时候，需要考虑是否能减少数据的读取量，数据是否在内存中连续存储，是否能利用 CPU 缓存预读。

5. 善用语言提供的类，避免重复造轮子

实际上，对于大部分常用的数据结构和算法，编程语言都提供了现成的类和函数实现。比如，Java 中的 HashMap 就是散列表的实现，TreeMap 就是红黑树的实现等。在实际的软件开发中，除非有特殊的要求，我们都可以直接使用编程语言中提供的这些类或函数。

这些编程语言提供的类和函数，都是经过无数验证过的，不管是正确性、鲁棒性，都要超过你自己造的轮子。而且，你要知道，重复造轮子，并没有那么简单。你需要写大量的测试用例，并且考虑各种异常情况，还要团队能看懂、能维护。这显然是一个出力不讨好的事情。这也是很多高级的数据结构和算法，比如 Trie 树、跳表等，在工程中，并不经常被应用的原因。

但这并不代表，学习数据结构和算法是没用的。深入理解原理，有助于你能更好地应用这些编程语言提供的类和函数。能否深入理解所用工具、类的原理，这也是普通程序员跟技术专家的区别。

6. 千万不要漫无目的地过度优化

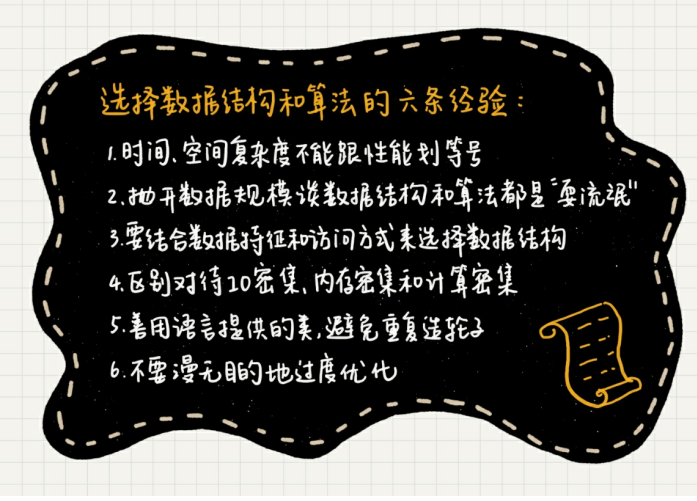
掌握了数据结构和算法这把锤子，不要看哪里都是钉子。比如，一段代码执行只需要 0.01 秒，你非得用一个非常复杂的算法或者数据结构，将其优化成 0.005 秒。即便你的算法再优秀，这种微小优化的意义也并不大。相反，对应的代码维护成本可能要高很多。

不过度优化并不代表，我们在软件开发的时候，可以不加思考地随意选择数据结构和算法。我们要学会估算。估算能力实际上也是一个非常重要的能力。我们不仅要对普通情况下的数据规模和性能压力做估算，还需要对异常以及将来一段时间内，可能达到的数据规模和性能压力做估算。这样，我们才能做到未雨绸缪，写出来的代码才能经久可用。

还有，当你真的要优化代码的时候，一定要先做 Benchmark 基准测试。这样才能避免你想当然地换了一个更高效的算法，但真实情况下，性能反倒下降了。

总结

工程上的问题，远比课本上的要复杂。所以，我今天总结了六条经验，希望你能把数据结构和算法用在刀刃上，恰当地解决实际问题。



我们在利用数据结构和算法解决问题的时候，一定要先分析清楚问题的需求、限制、隐藏的特点等。只有搞清楚了这些，才能有针对性地选择恰当的数据结构和算法。这种灵活应用的实战能力，需要长期的刻意锻炼和积累。这是一个有经验的工程师和一个学院派的工程师的区别。

好了，今天的内容就到这里了。最后，我想听你谈一谈，你在实际开发选择数据结构和算法时，有什么感受和方法呢？

欢迎在留言区写下你的想法，也欢迎你把今天的文章分享给你的朋友，帮助他在数据结构和算法的实际运用中走得更远。