03 | 复杂度分析（上）：如何分析、统计算法的执行效率和资源消耗？

2018-09-26 王争

03 | 复杂度分析（上）：如何分析、统计算法的执行效率和资源消耗？
朗读人：修阳 19′42′′ | 9.04M

我们都知道，数据结构和算法本身解决的是“快”和“省”的问题，即如何让代码运行得更快，如何让代码更省存储空间。所以，执行效率是算法一个非常重要的考量指标。那如何来衡量你编写的算法代码的执行效率呢？这里就要用到我们今天要讲的内容：时间、空间复杂度分析。

其实，只要讲到数据结构与算法，就一定离不开时间、空间复杂度分析。而且，我个人认为，**复杂度分析是整个算法学习的精髓，只要掌握了它，数据结构和算法的内容基本上就掌握了一半。**

复杂度分析实在太重要了，因此我准备用两节内容来讲。希望你学完这个内容之后，无论在任何场景下，面对任何代码的复杂度分析，你都能做到“庖丁解牛”般游刃有余。

为什么需要复杂度分析？

你可能会有些疑惑，我把代码跑一遍，通过统计、监控，就能得到算法执行的时间和占用的内存大小。为什么还要做时间、空间复杂度分析呢？这种分析方法能比我实实在在跑一遍得到的数据更准确吗？

首先，我可以肯定地说，你这种评估算法执行效率的方法是正确的。很多数据结构和算法书籍还给这种方法起了一个名字，叫**事后统计法**。但是，这种统计方法有非常大的局限性。

1. 测试结果非常依赖测试环境

测试环境中硬件的不同会对测试结果有很大的影响。比如，我们拿同样一段代码，分别用 Intel Core i9 处理器和 Intel Core i3 处理器来运行，不用说，i9 处理器要比 i3 处理器执行的速度快很多。还有，比如原本在这台机器上 a 代码执行的速度比 b 代码要快，等我们换到另一台机器上时，可能会有截然相反的结果。

2. 测试结果受数据规模的影响很大

后面我们会讲排序算法，我们先拿它举个例子。对同一个排序算法，待排序数据的有序度不一样，排序的执行时间就会有很大的差别。极端情况下，如果数据已经是有序的，那排序算法不需要做任何操作，执行时间就会非常短。除此之外，如果测试数据规模太小，测试结果可能无法真实地反应算法的性能。比如，对于小规模的数据排序，插入排序可能反倒会比快速排序要快！

所以，**我们需要一个不用具体的测试数据来测试，就可以粗略地估计算法的执行效率的方法**。这就是我们今天要讲的时间、空间复杂度分析方法。

大 O 复杂度表示法

算法的执行效率，粗略地讲，就是算法代码执行的时间。但是，如何在不运行代码的情况下，用“肉眼”得到一段代码的执行时间呢？

这里有段非常简单的代码，求 1,2,3...n 的累加和。现在，我就带你一块来估算一下这段代码的执行时间。

```
1  int cal(int n) {
2    int sum = 0;
3    int i = 1;
4    for (; i <= n; ++i) {
5      sum = sum + i;
6    }
7    return sum;
8  }
```

从 CPU 的角度来看，这段代码的每一行都执行着类似的操作：**读数据-运算-写数据**。尽管每行代码对应的 CPU 执行的个数、执行的时间都不一样，但是，我们这里只是粗略估计，所以可以假设每行代码执行的时间都一样，为 unit_time。在这个假设的基础之上，这段代码的总执行时间是多少呢？

第 2、3 行代码分别需要 1 个 unit_time 的执行时间，第 4、5 行都运行了 n 遍，所以需要 2n*unit_time 的执行时间，所以这段代码总的执行时间就是 (2n+2)*unit_time。可以看出来，**所有代码的执行时间 T(n) 与每行代码的执行次数成正比**。

按照这个分析思路，我们再来看这段代码。

```
1  int cal(int n) {
2    int sum = 0;
3    int i = 1;
4    int j = 1;
5    for (; i <= n; ++i) {
6      j = 1;
7      for (; j <= n; ++j) {
8        sum = sum +  i * j;
9      }
10   }
11 }
```

我们依旧假设每个语句的执行时间是 unit_time。那这段代码的总执行时间 T(n) 是多少呢？

第 2、3、4 行代码，每行都需要 1 个 unit_time 的执行时间，第 5、6 行代码循环执行了 n 遍，需要 2n * unit_time 的执行时间，第 7、8 行代码循环执行了 n^2遍，所以需要 $2n^2$ * unit_time 的执行时间。所以，整段代码总的执行时间 $T(n) = (2n^2+2n+3)$*unit_time。

尽管我们不知道 unit_time 的具体值，但是通过这两段代码执行时间的推导过程，我们可以得到一个非常重要的规律，那就是，**所有代码的执行时间 T(n) 与每行代码的执行次数 n 成正比**。

我们可以把这个规律总结成一个公式。注意，大 O 就要登场了！

$$T(n) = O(f(n))$$

我来具体解释一下这个公式。其中，T(n) 我们已经讲过了，它表示代码执行的时间；n 表示数据规模的大小；f(n) 表示每行代码执行的次数总和。因为这是一个公式，所以用 f(n) 来表示。公式中的 O，表示代码的执行时间 T(n) 与 f(n) 表达式成正比。

所以，第一个例子中的 T(n) = O(2n+2)，第二个例子中的 $T(n) = O(2n^2+2n+3)$。这就是**大 O 时间复杂度表示法**。大 O 时间复杂度实际上并不具体表示代码真正的执行时间，而是表示**代码执行时间随数据规模增长的变化趋势**，所以，也叫作**渐进时间复杂度**（asymptotic time complexity），简称**时间复杂度**。

当 n 很大时，你可以把它想象成 10000、100000。而公式中的低阶、常量、系数三部分并不左右增长趋势，所以都可以忽略。我们只需要记录一个最大量级就可以了，如果用大 O 表示法表示刚讲的那两段代码的时间复杂度，就可以记为：T(n) = O(n)； $T(n) = O(n^2)$。

时间复杂度分析

前面介绍了大 O 时间复杂度的由来和表示方法。现在我们来看下，如何分析一段代码的时间复杂度？我这儿有三个比较实用的方法可以分享给你。

1. 只关注循环执行次数最多的一段代码

我刚才说了，大 O 这种复杂度表示方法只是表示一种变化趋势。我们通常会忽略掉公式中的常量、低阶、系数，只需要记录一个最大阶的量级就可以了。所以，**我们在分析一个算法、一段代码的时间复杂度的时候，也只关注循环执行次数最多的那一段代码就可以了**。这段核心代码执行次数的 n 的量级，就是整段要分析代码的时间复杂度。

为了便于你理解，我还拿前面的例子来说明。

```
1  int cal(int n) {
2    int sum = 0;
3    int i = 1;
4    for (; i <= n; ++i) {
5      sum = sum + i;
6    }
7    return sum;
8  }
```

其中第 2、3 行代码都是常量级的执行时间，与 n 的大小无关，所以对于复杂度并没有影响。循环执行次数最多的是第 4、5 行代码，所以这块代码要重点分析。前面我们也讲过，这两行代码被执行了 n 次，所以总的时间复杂度就是 O(n)。

2. 加法法则：总复杂度等于量级最大的那段代码的复杂度

我这里还有一段代码。你可以先试着分析一下，然后再往下看跟我的分析思路是否一样。

```
1 int cal(int n) {
2    int sum_1 = 0;
3    int p = 1;
4    for (; p < 100; ++p) {
5      sum_1 = sum_1 + p;
6    }
7
8    int sum_2 = 0;
9    int q = 1;
10   for (; q < n; ++q) {
11     sum_2 = sum_2 + q;
12   }
13
14   int sum_3 = 0;
15   int i = 1;
16   int j = 1;
17   for (; i <= n; ++i) {
18     j = 1;
19     for (; j <= n; ++j) {
20       sum_3 = sum_3 +  i * j;
21     }
22   }
23
24   return sum_1 + sum_2 + sum_3;
25 }
```

这个代码分为三部分，分别是求 sum_1、sum_2、sum_3。我们可以分别分析每一部分的时间复杂度，然后把它们放到一块儿，再取一个量级最大的作为整段代码的复杂度。

第一段的时间复杂度是多少呢？这段代码循环执行了 100 次，所以是一个常量的执行时间，跟 n 的规模无关。

这里我要再强调一下，即便这段代码循环 10000 次、100000 次，只要是一个已知的数，跟 n 无关，照样也是常量级的执行时间。当 n 无限大的时候，就可以忽略。尽管对代码的执行时间会有很大影响，但是回到时间复杂度的概念来说，它表示的是一个算法执行效率与数据规模增长的变化趋势，所以不管常量的执行时间多大，我们都可以忽略掉。因为它本身对增长趋势并没有影响。

那第二段代码和第三段代码的时间复杂度是多少呢？答案是 O(n) 和 $O(n^2)$，你应该能容易就分析出来，我就不啰嗦了。

综合这三段代码的时间复杂度，我们取其中最大的量级。所以，整段代码的时间复杂度就为 $O(n^2)$。也就是说：**总的时间复杂度就等于量级最大的那段代码的时间复杂度**。那我们将这个规律抽象成公式就是：

如果 T1(n)=O(f(n))，T2(n)=O(g(n))；那么 T(n)=T1(n)+T2(n)=max(O(f(n)), O(g(n))) =O(max(f(n), g(n))).

3. 乘法法则：嵌套代码的复杂度等于嵌套内外代码复杂度的乘积

我刚讲了一个复杂度分析中的加法法则，这儿还有一个**乘法法则**。类比一下，你应该能“猜到”公式是什么样子的吧？

如果 T1(n)=O(f(n))，T2(n)=O(g(n))；那么 T(n)=T1(n)*T2(n)=O(f(n))*O(g(n))=O(f(n)*g(n)).

也就是说，假设 T1(n) = O(n)，$T2(n) = O(n^2)$，则 $T1(n) * T2(n) = O(n^3)$。落实到具体的代码上，我们可以把乘法法则看成是**嵌套循环**，我举个例子给你解释一下。

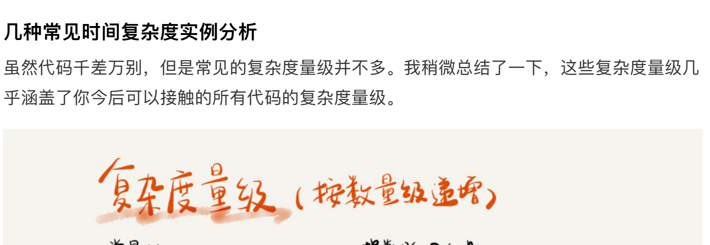
```
1 int cal(int n) {
2    int ret = 0;
3    int i = 1;
4    for (; i < n; ++i) {
5      ret = ret + f(i);
6    }
7  }
8
9  int f(int n) {
10  int sum = 0;
11  int i = 1;
12  for (; i < n; ++i) {
13    sum = sum + i;
14  }
15  return sum;
16 }
```

我们单独看 cal() 函数。假设 f() 只是一个普通的操作，那第 4～6 行的时间复杂度就是，T1(n) = O(n)。但 f() 函数本身不是一个简单的操作，它的时间复杂度是 T2(n) = O(n)，所以，整个 cal() 函数的时间复杂度就是，$T(n) = T1(n) * T2(n) = O(n*n) = O(n^2)$。

我刚刚讲了三种复杂度的分析技巧。不过，你并不用刻意去记忆。实际上，复杂度分析这个东西关键在于“熟练”。你只要多看案例，多分析，就能做到“无招胜有招”。

几种常见时间复杂度实例分析

虽然代码千差万别，但是常见的复杂度量级并不多。我稍微总结了一下，这些复杂度量级几乎涵盖了你今后可以接触的所有代码的复杂度量级。



对于刚罗列的复杂度量级，我们可以粗略地分为两类，**多项式量级**和**非多项式量级**。其中，非多项式量级只有两个：$O(2^n)$ 和 O(n!)。

当数据规模 n 越来越大时，非多项式量级算法的执行时间会急剧增加，求解问题的执行时间会无限增长。所以，非多项式时间复杂度的算法其实是非常低效的算法。因此，关于 NP 时间复杂度我就不展开讲了。我们主要来看几种常见的**多项式时间复杂度**。

1. O(1)

首先你必须明确一个概念，O(1) 只是常量级时间复杂度的一种表示方法，并不是指只执行了一行代码。比如这段代码，即便有 3 行，它的时间复杂度也是 O(1） ，而不是 O(3)。

```
1  int i = 8;
2  int j = 6;
3  int sum = i + j;
```

我稍微总结一下，只要代码的执行时间不随 n 的增大而增长，这样代码的时间复杂度我们都记作 O(1)。或者说，**一般情况下，只要算法中不存在循环语句、递归语句，即使有成千上万行的代码，其时间复杂度也是O(1)**。

2. O(logn)、O(nlogn)

对数阶时间复杂度非常常见，同时也是最难分析的一种时间复杂度。我通过一个例子来说明一下。

```
1  i=1;
2  while (i <= n)  {
3    i = i * 2;
4  }
```

根据我们前面讲的复杂度分析方法，第三行代码是循环执行次数最多的。所以，我们只要能计算出这行代码被执行了多少次，就能知道整段代码的时间复杂度。

从代码中可以看出，变量 i 的值从 1 开始取，每循环一次就乘以 2。当大于 n 时，循环结束。还记得我们高中学过的等比数列吗？实际上，变量 i 的取值就是一个等比数列。如果我把它一个一个列出来，就应该是这个样子的：

$$2^0 \quad 2^1 \quad 2^2 \cdots 2^k \cdots 2^x = n$$

所以，我们只要知道 x 值是多少，就知道这行代码执行的次数了。通过 $2^x=n$ 求解 x 这个问题我们想高中应该就学过了，我就不多说了。$x=\log_2 n$，所以，这段代码的时间复杂度就是 $O(\log_2 n)$。

现在，我把代码稍微改下，你再看看，这段代码的时间复杂度是多少？

```
1  i=1;
2  while (i <= n)  {
3    i = i * 3;
4  }
```

根据我刚刚讲的思路，很简单就能看出来，这段代码的时间复杂度为 $O(\log_3 n)$。

实际上，不管是以 2 为底、以 3 为底，还是以 10 为底，我们可以把所有对数阶的时间复杂度都记为 O(logn)。为什么呢？

我们知道，对数之间是可以互相转换的，$\log_3 n$ 就等于 $\log_3 2 * \log_2 n$，所以 $O(\log_3 n) = O(C * \log_2 n)$，其中 $C=\log_3 2$ 是一个常量。基于我们前面的一个理论：**在采用大 O 标记复杂度的时候，可以忽略系数，即 O(Cf(n)) = O(f(n))**。所以，$O(\log_2 n)$ 就等于 $O(\log_3 n)$。因此，在对数阶时间复杂度的表示方法里，我们忽略对数的“底”，统一表示为 O(logn)。

如果你理解了我前面讲的 O(logn)，那 O(nlogn) 就很容易理解了。还记得我们刚讲的乘法法则吗？如果一段代码的时间复杂度是 O(logn)，我们循环执行 n 遍，时间复杂度就是 O(nlogn) 了。而且，O(nlogn) 也是一种非常常见的算法时间复杂度。比如，归并排序、快速排序的时间复杂度都是 O(nlogn)。

3. O(m+n)、O(m*n)

我们再来讲一种跟前面都不一样的时间复杂度，代码的复杂度**由两个数据的规模**来决定。老规矩，先看代码！

```
1 int cal(int m, int n) {
2   int sum_1 = 0;
3   int i = 1;
4   for (; i < m; ++i) {
5     sum_1 = sum_1 + i;
6   }
7
8   int sum_2 = 0;
9   int j = 1;
10  for (; j < n; ++j) {
11    sum_2 = sum_2 + j;
12  }
13
14  return sum_1 + sum_2;
15 }
```

从代码中可以看出，m 和 n 是表示两个数据规模。我们无法事先评估 m 和 n 谁的量级大，所以我们在表示复杂度的时候，就不能简单地利用加法法则，省略掉其中一个。所以，上面代码的时间复杂度就是 O(m+n)。

针对这种情况，原来的加法法则就不正确了，我们需要将加法规则改为：T1(m) + T2(n) = O(f(m) + g(n))。但是乘法法则继续有效：T1(m)*T2(n) = O(f(m) * f(n))。

空间复杂度分析

前面，咱们花了很长时间讲大 O 表示法和时间复杂度分析，理解了前面讲的内容，空间复杂度分析方法学起来就非常简单了。

前面我讲过，时间复杂度的全称是**渐进时间复杂度**，**表示算法的执行时间与数据规模之间的增长关系**。类比一下，空间复杂度全称就是**渐进空间复杂度**（asymptotic space complexity），**表示算法的存储空间与数据规模之间的增长关系**。

我还是拿具体的例子来给你说明。（这段代码有点“傻”，一般没人会这么写，我这么写只是为了方便给你解释。）

```
1 void print(int n) {
2   int i = 0;
3   int[] a = new int[n];
4   for (i; i <n; ++i) {
5     a[i] = i * i;
6   }
7
8   for (i = n-1; i >= 0; --i) {
9     print out a[i]
10  }
11 }
```

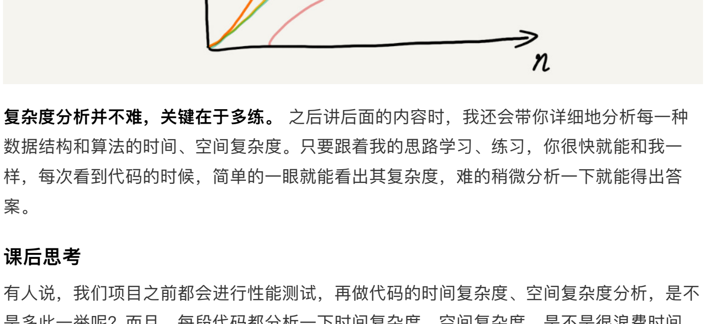
跟时间复杂度分析一样，我们可以看到，第 2 行代码中，我们申请了一个空间存储变量 i，但是它是常量阶的，跟数据规模 n 没有关系，所以我们可以忽略。第 3 行申请了一个大小为 n 的 int 类型数组，除此之外，剩下的代码都没有占用更多的空间，所以整段代码的空间复杂度就是 O(n)。

我们常见的空间复杂度就是 O(1)、O(n)、$O(n^2)$，像 O(logn)、O(nlogn) 这样的对数阶复杂度平时都用不到。而且，空间复杂度分析比时间复杂度分析要简单很多。所以，对于空间复杂度，掌握刚我说的这些内容已经足够了。

内容小结

基础复杂度分析的知识到此就讲完了，我们来总结一下。

复杂度也叫渐进复杂度，包括时间复杂度和空间复杂度，用来分析算法执行效率与数据规模之间的增长关系，可以粗略地表示，越高阶复杂度的算法，执行效率越低。常见的复杂度并不多，从低阶到高阶有：O(1)、O(logn)、O(n)、O(nlogn)、$O(n^2)$。等你学完整个专栏之后，你就会发现几乎所有的数据结构和算法的复杂度都跑不出这几个。



复杂度分析并不难，关键在于多练。 之后讲后面的内容时，我还会带你详细地分析每一种数据结构和算法的时间、空间复杂度。只要跟着我的思路学习、练习，你很快就能和我一样，每次看到代码的时候，简单的一眼就能看出其复杂度，难的稍微分析一下就能得出答案。

课后思考

有人说，我们项目之前都会进行性能测试，再做代码的时间复杂度、空间复杂度分析，是不是多此一举呢？而且，每段代码都分析一下时间复杂度、空间复杂度，是不是很浪费时间呢？你怎么看待这个问题呢？

欢迎留言和我分享，我会第一时间给你反馈。