02. 复杂度分析（上）

为什么需要复杂度分析？

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** 是算法的一个非常重要的考量指标。

而，算法代码的执行效率的衡量，需要用到 **时间、空间复杂度分析**。

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就基本上掌握了一半。

复杂度分析方法

事后统计法

- 概念：把代码跑一遍，通过统计、监控，得到算法执行的时间和占用内存大小的方法。
- 局限性（非常大）：
  - 测试结果非常依赖测试环境；
  - 测试结果受数据规模的影响很大。

大 O 复杂度 表示法

- 原因：需要一个不用具体的测试数据来测试，就可以粗略估计算法的执行效率的方法。
- 概念：大 O 复杂度 表示法，实际上并不具体表示代码真正的执行效率，而是表示代码执行效率随数据规模增长的变化趋势。通常会省略掉公式中的常量、低阶、系数，只需要记录一个最大阶的量级就可以了。
- 具体的看下面讲解。

复杂度分析

大O时间复杂度分析

算法的执行效率，粗略地讲，就是算法代码的执行时间。而如何在不运行代码的情况，用肉眼来得到一段代码的执行时间呢？

例子1：

这里有一段非常简单的代码，求 1，2，3 ... n 的累加和。

```
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、执行时间都不一样，但是，这里只是粗略估计。所以，可以假设每行代码执行的是几都是一样的，为 unit _ time.

在此基础上，第2、3行代码分别需要 1 个 unittime 的执行时间，第 4、5行代码都运行了 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.

可以看出来的是，所有代码的执行时间 T(n) 与 每行代码的执行次数正比。

例子2：

再来分析下面这段代码：

```
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  }
```

依旧假设每个语句的运行时间是 unit _ time.

第2、3、4 行代码，每行都需要 1 个 unittime 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 $2n^2 * unit\_time$ 的执行时间。所以，整段代码总的执行时间是：$T(n) = (2n^2 + 2n + 3) * unit\_time$.

尽管不知道 unit _ time 的具体值，但是通过这两段代码执行时间的推导过程，可以得到一个非常重要的规律：

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。

这样就可以把这个规律总结成一个公式，即 大O标记法：

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

参数说明：

- T（n）- 代码执行的总时间；
- n - 数据规模的大小；
- f(n) - 每行代码执行的次数总和；

- O - 表示代码的执行 T(n) 与 f(n) 成正比。

所以，第一个例子中的 $T(n) = O(2n+2)$，第二个例子中的$T(n) = O(2n^2+2n+3)$。

以上就是 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，也叫做渐进时间复杂度（Asymptotic Time Complexity),简称时间复杂度。

当 n 很大时，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，所以可以忽略。

所以，只需要记录一个最大量级就可以了，如果用大O法表示刚刚讲过的就可以记为：$T(n) = O(n)$ 和 $T(n) = O(n^2)$。

时间复杂度分析技巧

1. 只关注循环执行次数最多的一段代码

大O表示法只是表示一种变化趋势，通常会省略掉公式中的常量、低阶、系数，只需要记录一个最大阶的量级就可以了。

所以，在分析一个算法，一段代码的时间复杂度时，也只关注循环执行次数最多的那一段代码就可以了。即，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。

例子：

```
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。

第4、5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（仅含有一个数据规模）：

> - 总复杂度等于量级最大的那段代码的复杂度。

公式：

$$如果T_1(n) = O(f(n)), T_2(n) = O(g(n)), 那么T(n) = T_1(n) + T_2(n) = max(O(f(n)), O(g(n))) = O(max(f(n), g(n)).$$

例子：

```
int cal(int n) {
   int sum_1 = 0;
   int p = 1;
   for (; p < 100; ++p) {
```

```
        sum_1 = sum_1 + p;
      }

      int sum_2 = 0;
      int q = 1;
      for (; q < n; ++q) {
        sum_2 = sum_2 + q;
      }

      int sum_3 = 0;
      int i = 1;
      int j = 1;
      for (; i <= n; ++i) {
        j = 1;
        for (; j <= n; ++j) {
          sum_3 = sum_3 +  i * j;
        }
      }

      return sum_1 + sum_2 + sum_3;
    }
```

这个代码分为三部分，分别是求 sum_1、sum_2、sum_3.

先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个代码的复杂。

- 第一段：

第一段代码循环执行了100次，所以是一个常数量的执行时间，跟 n的规模无关，即为O(1).

> 注意：即便这段代码循环 10000次、100000次，只要一个已知的数，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的影响，但是回到时间复杂度的概念来说，它表示的一个算法执行效率与数据规模的变化趋势，所以不管常量的执行时间有多大，都可以忽略掉。因为它本身对增长趋势并没有影响。

- 第二段：

第二段代码循环执行了 2n 次，忽略掉次数,即为O(n)。

- 第三段：

第三段代码循环执行了 $2n + 2n^2$ 次，只关注最大量级并忽略系数，即为 $O(n^2)$.

- 总和：

这三段代码的时间复杂度，取其最大量级，整段代码时间复杂度就为$O(n^2)$.

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

公式：

如果$T_1(n)=O(f(n)), T_2(n)=O(g(n))$,那么$T(n)=T_1(n)*T_2(n)=O(f(n))*O(g(n))=O(f(n)*g(n))$.

- 例子：

```
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     }
```

但看 call() 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 $T_1(n)=O(n)$.

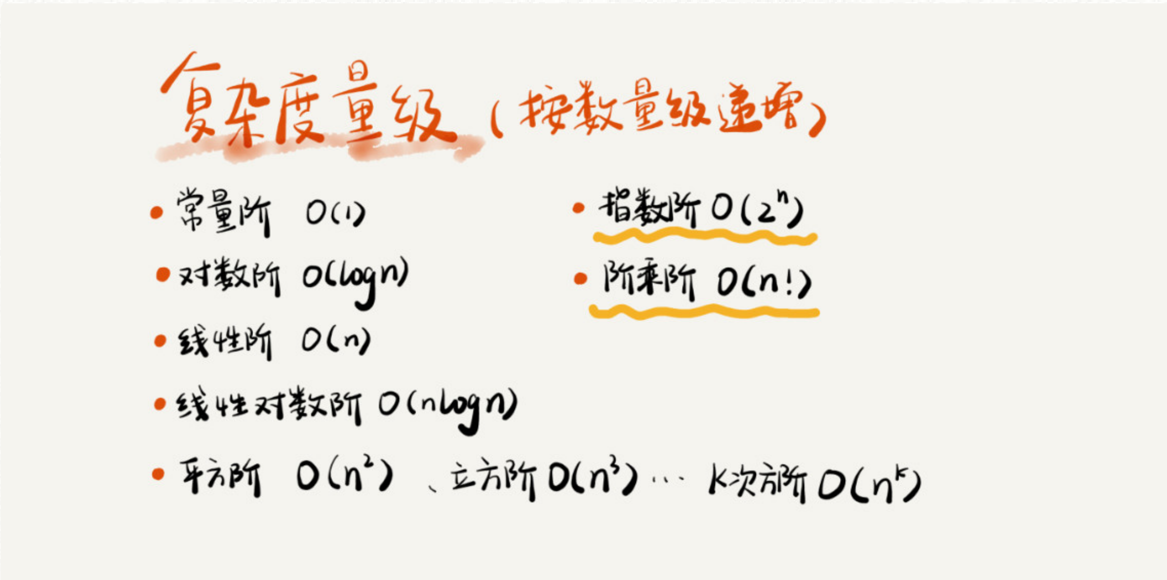
但是，f()函数本身不是一个简单的操作，它的时间复杂度是$T_2(n)=O(n)$.

所以，call（）函数的时间复杂度就是 $T(n)=T_1(n)*T_2(n)=O(n*n)=O(n^2)$.

几种常见时间复杂度实例分析

虽然代码千差万别，但常见的复杂度量级并不多。

总结为以下几种复杂度量级，这些复杂度量级几乎涵盖了可能接触到的所有代码的复杂度量级：



以上复杂度量级，大致可分为：多项式量级和非多项式量级。

非多项式量级，只有两个：$O(2^n)$ 和 $O(n!)$.

而把非多项式量级的算法问题叫作：NP（None-Deterministic Polynomial），非确定多项式问题。

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算法其实是非常低效的算法。

因此，关于NP的时间复杂度就不展开讲解，主要来看几种常见的多项式时间复杂度。

常见多项式时间复杂度

1. O (1)

首先必须明确一个概念，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).

```
int i = 8;
int j = 6;
int sum = i + j;
```

只要代码的执行时间不随 n 的增大而增大，这样代码的时间复杂度都记作 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。

通过以下例子来说明：

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

根据前面讲的复杂度分析法，第 3 行代码是循环执行次数最多的，所以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。

实际上，变量 i 的取值就是一个等比数列：

$$2^0，2^1，2^2,...,2^k,...,2^x=n$$

所以，只要知道 x 值是多少，就知道这行代码执行的次数了。

求解 方程，得 ，所以，这段代码时间复杂度就是 .

现在，把代码稍微改下，再看看，这段代码的时间复杂度是多少？

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

根据刚才思路，求解 方程，的，所以，这段代码时间复杂度就是 $O(log_3 n)$.

实际上，不管是以 2 为底、以 3 为底，还是以 10 为底，都可以把所有对数阶的时间复杂度都记为 $O(logn)$.

为什么呢？

> 对数之间是可以互相转换的，$O(log_3n)$ 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，忽略对数的"底"，统一表示为 $O(logn)$.**

3. O(m + n) 、O(m * n) --- 含多个数据规模，且无法事先评估谁的量级大

下面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。

```
int cal(int m, int n) {
  int sum_1 = 0;
  int i = 1;
  for (; i < m; ++i) {
    sum_1 = sum_1 + i;
  }

  int sum_2 = 0;
  int j = 1;
  for (; j < n; ++j) {
    sum_2 = sum_2 + j;
  }

  return sum_1 + sum_2;
}
```

从代码可以看出，**m 和 n 是表示两个数据规模，且无法事先评估 m 和 n 谁的量级大，所以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，需要将加法法则改为：**

$T_1(m) + T_2(n) = O(f(m) + g(n))$.

但是，**乘法法则继续有效**：

$T_1(m) * T_2(n) = O(f(m) * g(n))$.

空间复杂度分析

理解了时间复杂度分析，空间复杂度分析就非常简单了。

- 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.**

例子：

```
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，可以看到第 2 行代码中， 申请了一个空间存储变量 i，但是它是常量阶，跟数据规模 n 没有关系，所以可以忽略。

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).

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。

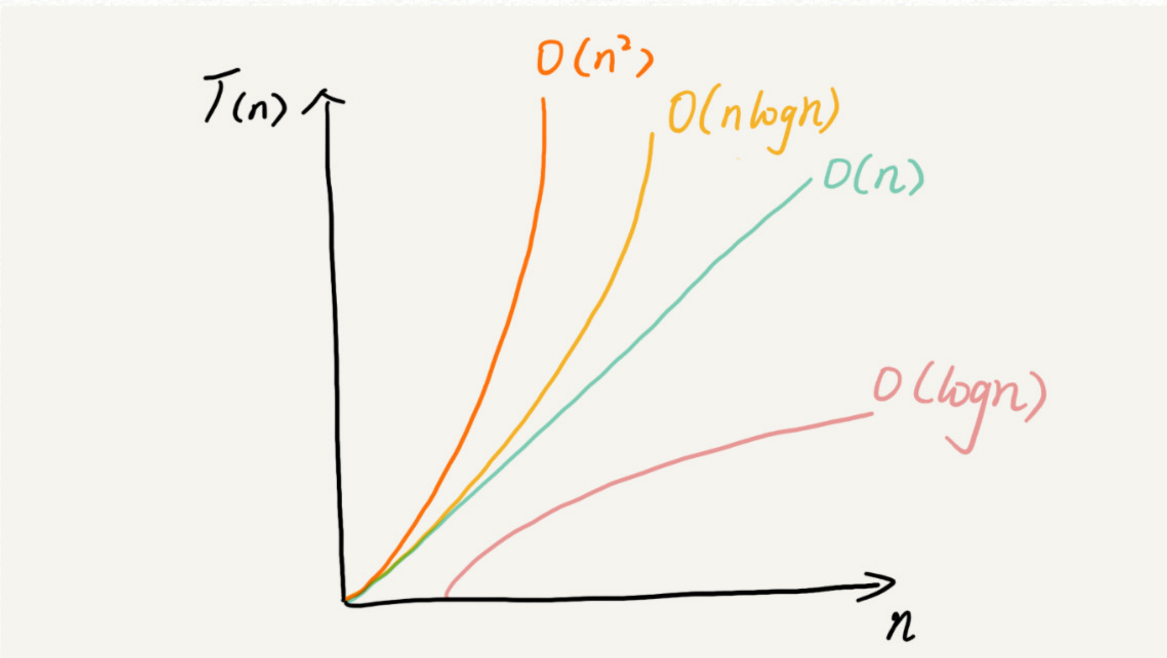
所以，对于空间复杂度，掌握刚说的这些内容就已经足够了.

算法执行效率 与 数据规模之间 的增长关系

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 ：

$O(1)$、$O(logn)$、$O(n)$、$O(nlogn)$、$O(n^2)$



注脚：

笔记时间：2021-02-22 二次总结 《算法与数据结构之美》王争专栏