15 定位问题才能更好地解决问题：开发前的复杂度分析与技术选型

在前面课时中，我们学习了数据结构和算法思维，这些知识和技巧，是解决问题、代码优化的基础。从本课时开始，我们将进入实战模块，从真正解决问题的角度来看看，如何将我们此前学到的知识灵活运用到实际工作中。

问题定位和技术选型

假设你现在面对一个实际的算法问题，则需要从以下两个方面进行思考。

首先，我们要明确目标。即用尽可能低的时间复杂度和空间复杂度，解决问题并写出代码； **接着，我们要定位问题。**目的是更高效地解决问题。这里定位问题包含很多内容。 例如：

- 这个问题是什么类型（排序、查找、最优化）的问题；
- 这个问题的复杂度下限是多少，即最低的时间复杂度可能是多少；
- 采用哪些数据结构或算法思维，能把这个问题解决。

为了方便你理解，下面我们来举一个例子，在一个包含 `n` 个元素的无序数组 `a` 中，输出其最大值 `max_val`。

这个问题比较简单。显然，要输出的最大值 `max_val`，也是原数组的元素之一。因此，这个问题的类型是，在数据中基于某个条件的查找问题。

关于查找问题，我们学习过二分查找，其复杂度是 `O(logn)`。但可惜的是，二分查找的条件是输入数据有序，这里并不满足。这就意味着，我们很难在 `O(logn)` 的复杂度下解决问题。

但是，继续分析你会发现，某一个数字元素的值会直接影响最终结果。这是因为，假设前 n-1 个数字的最大值是 5，但最后一个数字的值是否大于 5，会直接影响最后的结果。这就意味着，这个问题不把所有的输入数据全都过一遍，是无法得到正确答案的。要把所有数据全都过一遍，这就是 O(n) 的复杂度。

小结一下就是，因为该问题属于查找问题，所以考虑用 O(logn) 的二分查找。但因为数组无序，导致它并不适用。又因为必须把全部数据过一遍，因此考虑用 O(n) 的检索方法。这就是复杂度的下限。

当明确了复杂度的下限是 O(n) 后，你就能知道此时需要一层 for 循环去寻找最大值。那么循环的过程中，就可以实现动态维护一个最大值变量。空间复杂度是 O(1)，并不需要采用某些复杂的数据结构。这个问题我们在前面的课时 1 中写过的代码如下：

```
public void s1_3() {
        int a[] = { 1, 4, 3 };
        int max_val = -1;
        int max_inx = -1;
        for (int i = 0; i < a.length; i++) {
                if (a[i] > max_val) {
                        max_val = a[i];
                        max_inx = i;
                }
        }
        System.out.println(max_val);
}
```

通用解题的方法论

前面的例子只是一个简单的热身。在实际工作中，我们遇到的问题通常会更复杂多变。那么。面对这些问题是否有一些通用的解决方法呢？答案是有的。

面对一个未知问题时，你可以从复杂度入手。尝试去分析这个问题的时间复杂度上限是多少，也就是复杂度再高能高到哪里。这就是不计任何时间、空间损耗，采用暴力求解的方法去解题。然后分析这个问题的时间复杂度下限是多少，也就是时间复杂度再低能低到哪里。这就是你写代码的目标。

接着，尝试去定位问题。在分析出这两个问题之后，就需要去设计合理的数据结构和运用合适的算法思维，从暴力求解的方法去逼近写代码的目标了。 在这里需要先定位问题，这个问题的类型就决定了采用哪种算法思维。

最后，需要对数据操作进行分析。例如：在这个问题中，需要对数据进行哪些操作（增删查），数据之间是否需要保证顺序或逆序？当分析出这些操作的步骤、频次之后，就可以根据不同数据结构的特性，去合理选择你所应该使用的那几种数据结构了。

经过以上分析，我们对方法论进行提练，宏观上的步骤总结为以下 4 步：

1. **复杂度分析。**估算问题中复杂度的上限和下限。
2. **定位问题。**根据问题类型，确定采用何种算法思维。
3. **数据操作分析。**根据增、删、查和数据顺序关系去选择合适的数据结构，利用空间换取时间。
4. **编码实现。**

这套方法适用于绝大多数的问题，在实战中需要你灵活运用。

案例

梳理完方法论之后，我们回过头来再看一下以前的例子，看看采用方法论是如何分析题目并找到答案的。

例 1，在一个数组 `a = [1, 3, 4, 3, 4, 1, 3]` 中，找到出现次数最多的那个数字。如果并列存在多个，随机输出一个。

我们先来分析一下复杂度。假设我们采用最暴力的方法。利用双层循环的方式计算：

- 第一层循环，我们对数组中的每个元素进行遍历；

- 第二层循环，对于每个元素计算出现的次数，并且通过当前元素次数 time_tmp 和全局最大次数变量 time_max 的大小关系，持续保存出现次数最多的那个元素及其出现次数。

由于是双层循环，这段代码在时间方面的消耗就是 n*n 的复杂度，也就是 $O(n^2)$。这段代码我们在第 1 课时中的例子里讲过，这里就不再赘述了。

接着，我们思考一下这段代码最低的复杂度可能是多少？

不难发现，这个问题的复杂度最低低不过 O(n)。这是因为某个数字的数值是完全有可能影响最终结果。例如，a = [1, 3, 4, 3, 4, 1]，随机输出 1、3、4 都可以。如果 a 中增加一个元素变成，a = [1, 3, 4, 3, 4, 1, 3, 1]，则结果为 1。

由此可见，这个问题必须至少要对全部数据遍历一次，所以复杂度再低低不过 O(n)。

显然，这个问题属于在一个数组中，根据某个条件进行查找的问题。既然复杂度低不过 O(n)，我们也不用考虑采用二分查找了。此处是用不到任何算法思维。那么如何让 $O(n^2)$ 的复杂度降低为 O(n) 呢？

只有通过巧妙利用数据结构了。分析这个问题就可以发现，此时不需要关注数据顺序。因此，栈、队列等数据结构用到的可能性会很低。如果采用新的数据结构，增删操作肯定是少不了的。而原问题就是查找类型的问题，所以查找的动作一定是非常高频的。在我们学过的数据结构中，查找有优势，同时不需要考虑数据顺序的只有哈希表，因此可以很自然地想到用哈希表解决问题。

哈希表的结构是“key-value”的键值对，如何设计键和值呢？哈希表查找的 key，所以 key 一定存放的是被查找的内容，也就是原数组中的元素。数组元素有重复，但哈希表中 key 不能重复，因此只能用 value 来保存频次。

分析到这里，所有解决方案需要用到的关键因素就出来了，我们总结为以下 2 点：

1. 预期的时间复杂度是 O(n)，这就意味着编码采用一层的 for 循环，对原数组进行遍历。
2. 数据结构需要额外设计哈希表，其中 key 是数组的元素，value 是频次。这样可以支持 O(1) 时间复杂度的查找动作。

因此，这个问题的代码就是：

```
public void s2_4() {
        int a[] = { 1, 3, 4, 3, 4, 1, 3, 1 };
        Map<Integer, Integer> d = new HashMap<>();
        for (int i = 0; i < a.length; i++) {
                if (d.containsKey(a[i])) {
                        d.put(a[i], d.get(a[i]) + 1);
                } else {
                        d.put(a[i], 1);
                }
        }
        int val_max = -1;
        int time_max = 0;
        for (Integer key : d.keySet()) {
                if (d.get(key) > time_max) {
                        time_max = d.get(key);
                        val_max = key;
                }
        }
        System.out.println(val_max);
}
```

这个问题，我们在前面的课时中曾给出了答案。答案并不是最重要的，重要的是它背后的解题思路。这个思路可以运用在很多我们没有遇到过的复杂问题中。例如下面的问题。

例 2，这个问题是力扣的经典问题，two sums。给定一个整数数组 arr 和一个目标值 target，请你在该数组中找出加和等于目标值的两个整数，并返回它们在原数组中的下标。

你可以假设，原数组中没有重复元素，而且有且只有一组答案。但是，数组中的元素只能使用一次。例如，arr = [1, 2, 3, 4, 5, 6]，target = 4。因为，arr[0] + arr[2] = 1 + 3 = 4 = target，则输出 0，2。

首先，我们来分析一下复杂度。假设我们采用最暴力的方法，利用双层循环的方式计算，步骤如下：

- 第一层循环，我们对数组中的每个元素进行遍历；
- 第二层循环，对于第一层的元素与 target 的差值进行查找。

例如，第一层循环遍历到了 1，第二层循环就需要查找 target - arr[0] = 4 - 1 = 3 是否在数组中。由于是双层循环，这段代码在时间方面的消耗就是 n*n 的复杂度，也就是 $O(n^2)$。

接下来，我们看看下限。很显然，某个数字是否存在于原数组对结果是有影响的。因此，复杂度再低低不过 O(n)。

这里的问题是在数组中基于某个条件去查找数据的问题。然而可惜的是原数组并非有序，因此采用二分查找的可能性也会很低。那么如何把 $O(n^2)$ 的复杂度降低到 O(n) 呢？路径只剩下了数据结构。

在暴力的方法中，第二层循环的目的是查找 target - arr[i] 是否出现在数组中。很自然地就会联想到可能要使用哈希表。同时，这个例子中对于数据处理的顺序并不关心，栈或者队列使用的可能性也会很低。因此，不妨试试如何用哈希表去降低复杂度。

既然是要查找 target - arr[i] 是否出现过，因此哈希表的 key 自然就是 target - arr[i]。而 value 如何设计呢？这就要看一下结果了，最终要输出的是查找到的 arr[i] 和 target - arr[i] 在数组中的索引，因此 value 存放的必然是 index 的索引值。

基于上面的分析，我们就能找到解决方案，分析如下：

1. 预期的时间复杂度是 O(n)，这就意味着编码采用一层的 for 循环，对原数组进行遍历。
2. 数据结构需要额外设计哈希表，其中 key 是 target - arr[i]，value 是 index。这样可以支持 O(1) 时间复杂度的查找动作。

因此，代码如下：

```
private static int[] twoSum(int[] arr, int target) {
        Map<Integer, Integer> map = new HashMap<>();
```

```
        for (int i = 0; i < arr.length; i++) {
                map.put(arr[i], i);
        }
        for (int i = 0; i < arr.length; i++) {
                int complement = target - arr[i];
                if (map.containsKey(complement) && map.get(complement) != i) {
                        return new int[] { map.get(complement), i };
                }
        }
        return null;
}
```

在这段代码中我们采用了两个 for 循环，时间复杂度就是 O(n) + O(n) = O(n)。额外使用了 map，空间复杂度也是 O(n)。第一个 for 循环，把数组转为字典，存放的是“数值 -index”的键值对。第二个 for 循环，在字典中依次判断，target - arr[i] 是否出现过。如果它出现过，且不是它自己，则打印 target - arr[i] 和 arr[i] 的索引。

总结

在开发前，一定要对问题的复杂度进行分析，做好技术选型。这就是定位问题的过程。只有把这个过程做好，才能更好地解决问题。

通过本课时的学习，常用的分析问题的方法有以下 4 种：

1. **复杂度分析。**估算问题中复杂度的上限和下限。
2. **定位问题。**根据问题类型，确定采用何种算法思维。
3. **数据操作分析。**根据增、删、查和数据顺序关系去选择合适的数据结构，利用空间换取时间。
4. **编码实现。**

其中前 3 个步骤，分别对应于这个课程的模块 1 到模块 3，这也是算法开发的基础知识。有了这些知识，才能在实际问题中分析并拼装出解决方案。

练习题

最后，我们给出一个练习题。在这个课时案例 2 的 two sums 中，我们采用了两个 for 循环去实现。那么，能否只使用一个 for 循环完成结果的查找呢？

上一页 下一页