

## 例题分析一

LeetCode 第 56 题：给出一个区间的集合，请合并所有重叠的区间。

### 示例 1

输入:  $[[1,3], [2,6], [8,10], [15,18]]$

输出:  $[[1,6], [8,10], [15,18]]$

解释: 区间  $[1,3]$  和  $[2,6]$  重叠，将它们合并为  $[1,6]$ 。

### 示例 2

输入:  $[[1,4], [4,5]]$

输出:  $[[1,5]]$

解释: 区间  $[1,4]$  和  $[4,5]$  可被视为重叠区间。

## 解题思路：贪算法

在分析一些比较复杂的问题时，可以从比较简单的情况着手来寻找突破口，先来看看两个区间会出现多少种情况。

假设有区间  $a$  和  $b$ ，区间  $a$  的起始时间要早于  $b$  的起始时间。那么它们之间有如下的 3 种可能会出现的情况。

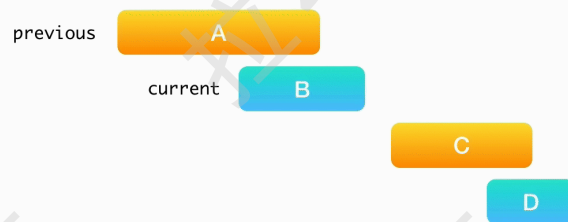


1. 情况一：两个区间没有任何重叠的部分，因此区间不会发生融合。
2. 情况二和三：区间有重叠。
  - a. 新区间的起始时间是 a 的起始时间，这个不变；
  - b. 新区间的终止时间是 a 的终止时间和 b 的终止时间的最大值，这个就是融合两个区间的最基本的思想。

给定了  $n$  个区间，如何有效地融合它们呢？以下是一种很直观也是非常有效的做法。

## 56. 合并区间

- ▶ 定义两个变量 `previous` 和 `current`，分别表示前一个区间和当前的区间
- 如果没有融合，当前区间就变成新的 `previous`，下一个区间成为新的 `current`
- 如果发生了融合，更新前一个区间的结束时间



1. 先将所有的区间按照起始时间的先后顺序排序，从头到尾扫描一遍
2. 定义两个变量 `previous` 和 `current`，分别表示前一个区间和当前的区间
  - a. 如果没有融合，那么当前区间就变成了新的前一个区间，下一个区间成为新的当前区间
  - b. 如果发生了融合，更新前一个区间的结束时间。

这个就是贪婪算法。

## 代码实现

```
int[][] merge(int[][] intervals) {  
    // 将所有区间按照起始时间的先后顺序排序  
    Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[0],  
        i2[0]));  
  
    // 定义一个 previous 变量，初始化为 null  
    int[] previous = null;
```

```

// 定义一个 result 变量，用来保存最终的区间结果
List<int[]> result = new ArrayList<>();

// 从头开始遍历给定的所有区间
for (int[] current : intervals) {
    // 如果这是第一个区间，或者当前区间和前一个区间没有重叠，那么
    // 将当前区间加入到结果中
    if (previous == null || current[0] > previous[1]) {
        result.add(previous = current);
    } else { // 否则，两个区间发生了重叠，更新前一个区间的结束时间
        prev[1] = Math.max(previous[1], current[1]);
    }
}

return result.toArray(new int[result.size()][2]);
}

```

### 算法分析

时间复杂度  $O(n\log(n))$ ，因为一开始要对数组进行排序。

空间复杂度为  $O(n)$ ，因为用了一个额外的 result 数组来保存结果。

注意：和区间相关的问题，有非常多的变化，融合区间可以说是最基本也是最常考的一个。

### 例题分析二

LeetCode 第 435 题：给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意：

1. 可以认为区间的终点总是大于它的起点。
2. 区间  $[1,2]$  和  $[2,3]$  的边界相互“接触”，但没有相互重叠。

### 示例 1

输入:  $[[1,2], [2,3], [3,4], [1,3]]$

输出: 1

**解释:** 移除  $[1,3]$  后，剩下的区间没有重叠。

### 示例 2

输入:  $[[1,2], [1,2], [1,2]]$

输出: 2

**解释:** 你需要移除两个  $[1,2]$  来使剩下的区间没有重叠。

### 示例 3

输入:  $[[1,2], [2,3]]$

输出: 0

**解释:** 你不需要移除任何区间，因为它们已经是无重叠的了。

**解题思路一：暴力法**

这道题是上一道题的一种变形，暴力法就是将各个区间按照起始时间的先后顺序排序，然后找出所有的组合，最后对每种组合分别判断各个区间有没有互相重叠。

### 算法分析

1. 排序需要  $O(n\log(n))$  的时间复杂度。
2. 找出所有组合，按照前一节课里提到的从一个字符串里找出所有子序列的组合个数的方法，取出  $n$  个区间，有  $Cnn$  种，算上空的集合，那么一共有  $Cn0 + Cn1 + Cn2 + \dots + Cnn = 2n$ 。
3. 对每种组合进行判断是否重叠， $k$  个区间，需要  $O(k)$  的时间复杂度。
4. 总体时间复杂度为  $Cn0 \times 0 + Cn1 \times 1 + Cn2 \times 2 + \dots + Cnk \times k + \dots + Cnn \times n = n \times 2n - 1$ 。

由于  $n \times 2n - 1$  已经远大于  $n\log(n)$ ，所以排序的时间复杂度就可以忽略不计，整体的时间复杂度就是  $O(n \times 2n)$ 。

建议：一定要记一些常见的时间复杂度计算公式，对于在面试中能准确快速地分析复杂度是非常有帮助的。

### 解题思路二：另一种暴力法

对于暴力法，还有另外的分析方法。用两个变量 `prev` 和 `curr` 分别表示前一个区间和当前区间。

1. 如果当前区间和前一个区间没有发生重叠，则尝试保留当前区间，表明此处不需要删除操作。
2. 题目要求最少的删除个数，只有在这样的情况下，才不需要做任何删除操作。
3. 在这种情况下，虽然两个区间没有重叠，但是也要考虑尝试删除当前区间的情况。
4. 对比哪种情况所需要删除的区间最少。

举例：有如下的几个区间 A、B、C，其中 A 是前一个区间，B 是当前区间，A 和 B 没有重叠。

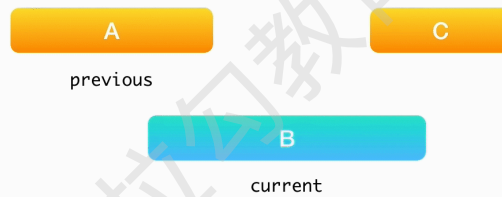


1. 如果只考虑保留 B 的情况，而不考虑把 B 删除的情况，那么就会错过一个答案，因为在这个情况下，把 B 删除，只剩下 A 和 C，它们互不重叠，也能得到最优的解。
2. 遇到 A 和 B 相互重叠的情况时，必须要考虑把 B 删除掉。

### 435. 无重叠区间

暴力法

▸ 方法二:

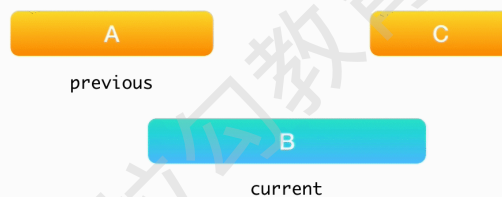


为什么不把 A 删除呢？因为如果把 A 删了，B 和 C 还是可能会重叠，则需要删除掉更多的区间，不满足题目要求。

### 435. 无重叠区间

暴力法

▸ 方法二:



### 代码实现

```
// 在主体函数里，先将区间按照起始时间的先后顺序排序，然后调用递归函数
int eraseOverlapIntervals(int[][] intervals) {
    Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[0], i2[0]));
```



```

        return eraseOverlapIntervals(-1, 0, intervals);
    }

    // 递归函数里，先检查是否已经处理完所有的区间，是，表明不需要删除操作，直接返回
    int eraseOverlapIntervals(int prev, int curr, int[][] intervals) {
        {
            if (curr == intervals.length) {
                return 0;
            }

            int taken = Integer.MAX_VALUE, nottaken;

            if (prev == -1 || intervals[prev][1] <= intervals[curr][0]) {
                // 只有当 prev, curr 没有发生重叠的时候，才可以选择保留当前的区间 curr
                taken = eraseOverlapIntervals(curr, curr + 1, intervals);
            }

            // 其他情况，可以考虑删除掉 curr 区间，看看删除了它之后会不会产生最好的结果
            nottaken = eraseOverlapIntervals(prev, curr + 1, intervals) + 1;

            return Math.min(taken, nottaken);
        }
    }

```

## 解题思路二：贪算法

### 按照起始时间排序

**举例：**有四个区间 A, B, C, D, A 跨度很大, B 和 C 重叠, C 和 D 重叠, 而 B 和 D 不重叠。

**解法：**要尽可能少得删除区间，那么当遇到了重叠的时候，应该把区间跨度大，即结束比较晚的那个区间删除。因为如果不删除它，它会和剩下的其他区间发生重叠的可能性非常大。

当发现 A 和 B 重叠，如果不删除 A，就得牺牲 B，C，D，而正确的答案是只需要删除 A 和 C 即可。

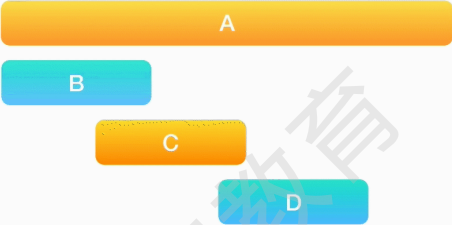
按照上述思想求解，实现过程如下。

1. A 和 B 重叠，由于 A 结束得比较晚，此时删除区间 A，保留区间 B。

435. 无重叠区间

贪婪法

方法一：按照起始时间排序



The diagram illustrates four intervals represented as horizontal bars. Interval A is the longest and highest, colored orange. Interval B is shorter than A, colored blue, and starts at the same left position as A but ends earlier. Interval C is colored orange, starts at the same position as B, but ends later than B. Interval D is the shortest, colored blue, and starts at the same position as C but ends earliest. This visualizes the greedy selection process where intervals are sorted by their end times.

2. B 和 C 重叠，由于 C 结束得晚，把区间 C 删除，保留区间 B。

### 435. 无重叠区间

贪婪法

▸ 方法一：按照起始时间排序

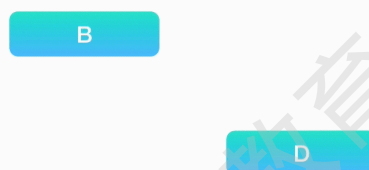


3. B 和 D 不重叠，结束，一共只删除了 2 个区间。

### 435. 无重叠区间

贪婪法

▸ 方法一：按照起始时间排序



### 代码实现

```
int eraseOverlapIntervals(int[][] intervals) {  
    if (intervals.length == 0) return 0;  
  
    // 将所有区间按照起始时间排序
```

```

        Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[0],
i2[0]));

        // 用一个变量 end 记录当前的最小结束时间点，以及一个 count 变量记录到目前为止删除掉了多少区间
        int end = intervals[0][1], count = 0;

        // 从第二个区间开始，判断一下当前区间和前一个区间的结束时间
        for (int i = 1; i < intervals.length; i++) {
            // 当前区间和前一个区间有重叠，即当前区间的起始时间小于上一个区间的结束时间，end 记录下两个结束时间的最小值，把结束时间晚的区间删除，计数加 1。
            if (intervals[i][0] < end) {
                end = Math.min(end, intervals[i][1]);
                count++;
            } else {
                end = intervals[i][1];
            }
        }

        // 如果没有发生重叠，根据贪算法，更新 end 变量为当前区间的结束时间
        return count;
    }
}

```

### 按照结束时间排序

**题目演变：**在给定的区间中，最多有多少个区间相互之间是没有重叠的？

思路：假如求出了最多有  $m$  个区间是互相之间没有重叠的，则最少需要将  $n - m$  个区间删除才行。即，删掉“害群之马”，则剩下的就不会互相冲突了。

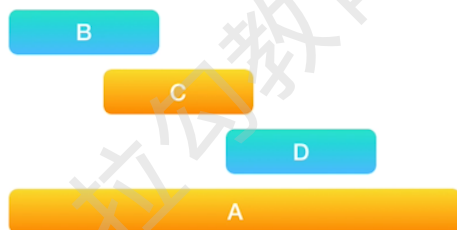
为什么按照结束时间排序会有助于我们统计出没有重叠的区间最大个数呢？举

例说明如下。

假设今天有很多活动要举行，每个活动都有固定的时间，选择哪些活动，才能使参加的活动最多，然后在时间上不会互相重叠呢？

如果我们按照活动的起始时间去挑选，某个活动虽然开始得早，但是很有可能会持续一整天，就没有时间去参加其他活动了。如果按照活动的结束时间选，先挑那些最早结束的，就会尽可能节省出更多的时间来参加更多的活动。

根据这个思路，这道题也可以按照结束时间排序处理，于是，区间的顺序就是 {B, C, D, A}。



实现：目标就是统计有多少个没有重叠的情况发生。若当前的区间和前一个区间没有重叠的时候，计数器加 1，同时，用当前的区间去和下一个区间比较。

### 代码实现

```
int eraseOverlapIntervals(int[][] intervals) {  
    if (intervals.length == 0) return 0;  
  
    // 按照结束时间将所有区间进行排序  
    Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[1],
```

```

        i2[1])));

        // 定义一个变量 end 用来记录当前的结束时间，count 变量用来记录有多少个没有重叠的区间
        int end = intervals[0][1], count = 1;

        // 从第二个区间开始遍历剩下的区间
        for (int i = 1; i < intervals.length; i++) {
            // 当前区间和前一个结束时间没有重叠，那么计数加 1，同时更新一下新的结束时间
            if (intervals[i][0] >= end) {
                end = intervals[i][1];
                count++;
            }
        }

        // 用总区间的个数减去没有重叠的区间个数，即为最少要删除掉的区间个数
        return intervals.length - count;
    }
}

```

关于区间的问题，LeetCode 上还有很多类似的题目，大家一定要去做做。

### 例题分析三

LeetCode 第 269 题，火星字典：现有一种使用字母的全新语言，这门语言的字母顺序与英语顺序不同。假设，您并不知道其中字母之间的先后顺序。但是，会收到词典中获得一个不为空的单词列表。因为是从词典中获得的，所以该单词列表内的单词已经按这门新语言的字母顺序进行了排序。您需要根据这个输入的列表，还原出此语言中已知的字母顺序。

#### 示例 1

输入:

```
[ "wrt", "wrf", "er", "ett", "rftt"]
```

输出: "wertf"

### 示例 2

输入:

[ "z", "x" ]

输出: "zx"

### 示例 3

输入:

[ "z", "x", "z" ]

输出: ""

解释: 此顺序是非法的, 因此返回 ""。

### 解题思路

首先, 确定字符串排序方法。

理解题意, 关键是搞清楚给定的输入字符串是怎么排序的?

举例: 假如我们有这些单词 bar, bat, algorithm, cook, cog, 那么按照字符顺序, 应该怎么排呢?

正确的排序应该是：algorithm bat bar cog cook。

解法：

- 逐位地比较两个相邻的字符串
- 第一个字母出现的顺序越早，排位越靠前
- 第一个字母相同时，比较第二字母，以此类推

注意：两个字符串某个相同的位置出现了不同，就立即能得出它们的顺序，无需继续比较字符串剩余字母。

### 求解示例 1

输入是：

```
wrt  
wrf  
er  
ett  
rftt
```

解法：

1. 比较以 w 开头的字符串，它们是 wrt 和 wrf，之所以 wrt 会排在 wrf 之前，是因为 t 比 f 在火星字典里出现的顺序要早。此时将这两个字母的关系表达为  $t \rightarrow f$ 。



### 269. 火星字典



wrt  
wrf

2. 比较 wrf 和 er，第一个字母开始不同，因此，得出 w 排在 e 之前，记为  $w \rightarrow e$ 。

### 269. 火星字典



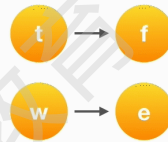
wrf  
er

t → f

3. 比较 er 和 ett，从第二个字母开始不一样，因此，得出 r 排在 t 之前，记为  $r \rightarrow t$ 。

269. 火星字典

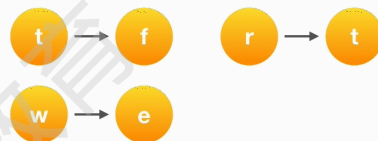
er  
ett



4. 比较 ett 和 rftt , 从第一个字母开始不一样 , 得出 e 排在 r 之前 , 记为  $e \rightarrow r$ 。

269. 火星字典

ett  
rftt



梳理上述关系 , 得  $t \rightarrow f, w \rightarrow e, r \rightarrow t, e \rightarrow r$

拓扑排序得到正确顺序：将每个字母看成是图里的顶点，它们之间的关系就好比是连接顶点与顶点的变，而且是有向边，所以这个图是一个有向图。最后对这个有向图进行拓扑排序，就可以得出最终的结果。



## 代码实现

包括两大步骤，第一步是根据输入构建一个有向图；第二步是对这个有向图进行拓扑排序。

```
// 基本情况处理，比如输入为空，或者输入的字符串只有一个
String alienOrder(String[] words) {
    if (words == null || words.length == 0)
        return null;
    if (words.length == 1) {
        return words[0];
    }

    // 构建有向图：定义一个邻接链表 adjList，也可以用邻接矩阵
    Map<Character, List<Character>> adjList = new HashMap<>();

    for (int i = 0; i < words.length - 1; i++) {
        String w1 = words[i], w2 = words[i + 1];
        int n1 = w1.length(), n2 = w2.length();

        boolean found = false;
        for (int j = 0; j < Math.max(w1.length(), w2.length()); j++) {
            Character c1 = j < n1 ? w1.charAt(j) : null;
```

```

        Character c2 = j < n2 ? w2.charAt(j)
: null;

        if (c1 != null && !adjList.containsKey(c1)) {
            adjList.put(c1, new ArrayList<Character>());
        }

        if (c2 != null && !adjList.containsKey(c
2)) {
            adjList.put(c2, new ArrayList<Character>());
        }

        if (c1 != null && c2 != null && c1
!= c2 && !found) {
            adjList.get(c1).add(c2);
            found = true;
        }
    }
}

Set<Character> visited = new HashSet<>();
Set<Character> loop = new HashSet<>();
Stack<Character> stack = new Stack<Character>();

for (Character key : adjList.keySet()) {
    if (!visited.contains(key)) {
        if (!topologicalSort(adjList, key, visited, loop, st
ack)) {
            return "";
        }
    }
}

StringBuilder sb = new StringBuilder();

while (!stack.isEmpty()) {
    sb.append(stack.pop());
}

return sb.toString();
}

// 将当前节点 u 加入到 visited 集合以及 loop 集合中。
boolean topologicalSort(Map<Character, List<Character>> adjList,

```

```

char u,
    Set<Character> visit
ed, Set<Character> loop, Stack<Character> stack) {
    visited.add(u);
    loop.add(u);

    if (adjList.containsKey(u)) {
        for (int i = 0; i < adjList.get(u).size(); i++) {
            char v = adjList.get(u).get(i);

            if (loop.contains(v))
                return false;

            if (!visited.contains(v)) {
                if (!topologicalSort(adjList, v, visited, loop,
stack)) {
                    return false;
                }
            }
        }
    }

    loop.remove(u);

    stack.push(u);

    return true;
}

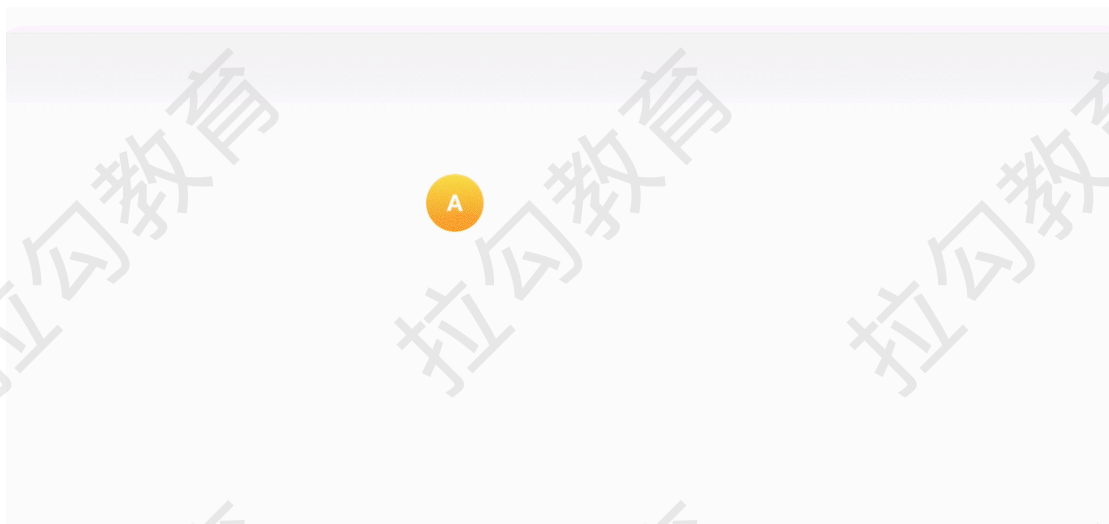
```

用深度优先的方法进行拓扑排序，一定要借用下面三者。

1. visited 集合，用来记录哪些顶点已经被访问过。
2. stack 堆栈，从某个顶点出发，访问完了所有其他顶点，最后才把当前的这个顶点加入到堆栈里。即，若要该点加入到 stack 里，必须先把跟它有联系的顶点都处理完。举例说明，如果我要学习课程 A，得先把课程 B，C 以及 D 都看完。

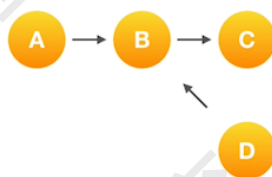
3. loop 集合，为了有效防止有向图里出现环的情况。举例说明如下。

假如我们有这么一个有向图。



- 从 A 开始对这个图进行深度优先的遍历，那么当访问到顶点 D 的时候，visited 集合以及 loop 集合都是 {A, B, C, D}。
- 当从 D 继续遍历到 B 的时候，发现 B 已经在 loop 集合里。
- 因此得出结论，在这一轮遍历中，出现了环。

为什么不能单单用 visited 集合来帮助判断呢？例如下面情况。



- 从 D 访问 B 的时候，如果判断因为 B 已经被访问过了，于是得出这里就有一个环，显然判断错误。
- 当每一轮访问结束后，都必须要把 loop 集合清空，才能把其他顶点也加入到堆栈里。
- 否则，当 D 遇到 B 的时候，也会认为这里有环出现，而提前终止程序，无法将它加入到堆栈中。

#### 例题分析四

LeetCode 第 772 题，基本计算器：实现一个基本的计算器来计算简单的表达式字符串。

说明：

- 表达式字符串可以包含左括号（和右括号），加号 + 和减号 -，非负整数和空格。

- 表达式字符串只包含非负整数， $+$   $-$   $*$   $/$  操作符，左括号（ $($ ，右括号 $)$ 和空格。整数除法需要向下截断。

#### 示例 1：

"1 + 1" = 2

"6-4 / 2" = 4

"2×(5+5×2)/3+(6/2+8)" = 21

"(2+6×3+5- (3×14/7+2)×5)+3" = -12

#### 解题思路一：只有加号

例题：若表达式里只有数字和加法符号，没有减法，也没有空格，并且输入的表达式一定合法，那么应该如何处理？例如：1+2+10。

解法：一旦遇到了数字就不断地相加。

#### 代码实现

```
// 转换，将字符串的字符放入到一个优先队列中
int calculate(String s) {
    Queue<Character> queue = new LinkedList<>();
    for (char c : s.toCharArray()) {
        queue.offer(c);
    }

    // 定义两个变量，num 用来表示当前的数字，sum 用来记录最后的和
    int num = 0, sum = 0;

    // 遍历优先队列，从队列中一个一个取出字符
    while (!queue.isEmpty()) {
        char c = queue.poll();
```



```

        // 如果当前字符是数字，那么就更新 num 变量，如果遇到
        // 到了加号，就把当前的 num 加入到 sum 里，num 清零
        if (Character.isDigit(c)) {
            num = 10 * num + c - '0';
        } else {
            sum += num;
            num = 0;
        }
    }

    sum += num; // 最后没有加号，再加一次

    return sum;
}

```

### 代码扩展一

如上，在返回 sum 之前，我们还进行了一次额外的操作：sum += num，就是为了要处理结束时的特殊情况。若在表达式的最后添加上一个“+”，也能实现同样的效果，代码实现如下。

```

int calculate(String s) {
    Queue<Character> queue = new LinkedList<>();
    for (char c : s.toCharArray()) {
        queue.offer(c);
    }
    queue.add('+'); // 在末尾添加一个加号

    while (!queue.isEmpty()) {
        ...
    }

    return sum;
}

```

如上，在优先队列的最后添加一个加号。

### 代码扩展二

若输入的时候允许空格，如何处理？代码实现如下。

```
int calculate(String s) {  
    ...  
    for (char c : s.toCharArray()) {  
        if (c != ' ') queue.offer(c);  
    }  
    ...  
}
```

如上，在添加到优先队列的时候，过滤到那些空格就好了。

### 解题思路二：引入减号

例题：若表达式支持减法，应该怎么处理？例如： $1 + 2 - 10$ 。

解法 1：借助两个 stack，一个 stack 专门用来放数字；一个 stack 专门用来放符号。

解法 2：将表达式看作  $1 + 2 + (-10)$ ，把  $-10$  看成一个整体，同时，利用一个变量 sign 来表示该数字前的符号，这样即可沿用解法。

解法 2 的具体操作如

下。

### 772. 基本计算器 III

+ 1 + 2 - 10 +

```
sign =  
num =  
sum =
```

#### 代码实现

```
int calculate(String s) {  
    Queue<Character> queue = new LinkedList<>();  
    for (char c : s.toCharArray()) {  
        if (c != ' ') queue.offer(c);  
    }  
    queue.add('+');  
  
    char sign = '+'; // 添加一个符号标志变量  
  
    int num = 0, sum = 0;  
  
    while (!queue.isEmpty()) {  
        char c = queue.poll();  
  
        if (Character.isDigit(c)) {  
            num = 10 * num + c - '0';  
        } else {  
            // 遇到了符号，表明我们要开始统计一下当前的结果了  
            if (sign == '+') { // 数字的符号是 +  
                sum += num;  
            } else if (sign == '-') { // 数字的符号是 -  
                sum -= num;  
            }  
            num = 0;  
            sign = c;  
        }  
    }  
}
```

```
}  
    return sum;  
}
```

### 解题思路三：引入乘除

例题：若引入乘法和除法，如何处理？举个例子： $1 + 2 \times 10$ 。

解法：要考虑符号的优先级问题，不能再简单得对 sum 进行单向的操作。当遇到乘号的时候： $sum = 1$ ， $num = 2$ ，而乘法的优先级比较高，得先处理  $2 \times 10$  才能加 1。对此，就把它暂时记录下来，具体操作如下。

#### 772. 基本计算器 III

+ 1 + 2 × 10 +

```
sign =  
num =  
stack =
```

### 代码实现

```
int calculate(String s) {  
    Queue<Character> queue = new LinkedList<>();  
    for (char c : s.toCharArray()) {  
        if (c != ' ') queue.offer(c);  
    }  
    queue.add('+');  
}
```

```

        char sign = '+';
        int num = 0;

        // 定义一个新的变量 stack，用来记录要被处理的数
        Stack<Integer> stack = new Stack<>();

        while (!queue.isEmpty()) {
            char c = queue.poll();

            if (Character.isDigit(c)) {
                num = 10 * num + c - '0';
            } else {
                if (sign == '+') {
                    stack.push(num); // 遇到加号，把当前的数压入到堆栈
中
                } else if (sign == '-') {
                    stack.push(-num); // 减号，把当前数的相反数压入到堆
栈中
                } else if (sign == '*') {
                    stack.push(stack.pop() * num); // 乘号，把前一个
数从堆栈中取出，然后和当前的数相乘，再放回堆栈
                } else if (sign == '/') {
                    stack.push(stack.pop() / num); // 除号，把前一个
数从堆栈中取出，然后除以当前的数，再把结果放回堆栈
                }

                num = 0;
                sign = c;
            }
        }

        int sum = 0;

        // 堆栈里存储的都是各个需要相加起来的结果，把它们加起来，返回总
和即可
        while (!stack.isEmpty()) {
            sum += stack.pop();
        }
        return sum;
    }

```

**解题思路四：引入小括号**

例题：如何支持小括号？

解法：小括号里的表达式优先计算。

1. 先利用上面的方法计算小括号里面的表达式。
2. 当遇到一个左括号的时候，可以递归地处理；当遇到了右括号，表明小括号里面的处理完毕，递归应该返回。

### 代码实现

```
// 在主函数里调用一个递归函数
int calculate(String s) {
    Queue<Character> queue = new LinkedList<>();
    for (char c : s.toCharArray()) {
        if (c != '(') queue.offer(c);
    }
    queue.offer('+');
    return calculate(queue);
}

int calculate(Queue<Character> queue) {
    char sign = '+';
    int num = 0;

    Stack<Integer> stack = new Stack<>();

    while (!queue.isEmpty()) {
        char c = queue.poll();

        if (Character.isDigit(c)) {
            num = 10 * num + c - '0';
        }

        // 遇到一个左括号，开始递归调用，求得括号里的计算结果，将它赋给当前的 num
        else if (c == '(') {
            num = calculate(queue);
        }
        else {

```

```
        if (sign == '+') {
            stack.push(num);
        } else if (sign == '-') {
            stack.push(-num);
        } else if (sign == '*') {
            stack.push(stack.pop() * num);
        } else if (sign == '/') {
            stack.push(stack.pop() / num);
        }

        num = 0;
        sign = c;

        // 遇到右括号，就可以结束循环，直接返回当前的总
和
        if (c == ')') {
            break;
        }
    }

    int sum = 0;
    while (!stack.isEmpty()) {
        sum += stack.pop();
    }
    return sum;
}
```