

# 我写了首诗，让你闭着眼睛也能写对二分搜索

 Stars 107k  B站 @labuladong 配套PDF和插件 下载 打卡挑战 报名 精品课程 查看



微信搜一搜

Q labuladong公众号

**通知：**数据结构精品课持续更新中，[详情见这里](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

牛客	LeetCode	力扣	难度
-	34. Find First and Last Position of Element in Sorted Array	34. 在排序数组中查找元素的第一个和最后一个位置	
-	704. Binary Search	704. 二分查找	

本文是前文 [二分搜索详解](#) 的修订版，添加了对二分搜索算法更详细的分析。

先给大家讲个笑话乐呵一下：

有一天阿东到图书馆借了  $N$  本书，出图书馆的时候，警报响了，于是保安把阿东拦下，要检查一下哪本书没有登记出借。阿东正准备把每一本书在报警器下过一下，以找出引发警报的书，但是保安露出不屑的眼神：你连二分查找都不会吗？于是保安把书分成两堆，让第一堆过一下报警器，报警器响；于是再把这堆书分成两堆……最终，检测了  $\log N$  次之后，保安成功的找到了那本引起警报的书，露出了得意和嘲讽的笑容。于是阿东背着剩下的书走了。

从此，图书馆丢了  $N - 1$  本书。

二分查找并不简单，Knuth 大佬（发明 KMP 算法的那位）都说二分查找：**思路很简单，细节是魔鬼**。很多人喜欢拿整型溢出的 bug 说事儿，但是二分查找真正的坑根本就不是那个细节问题，而是在于到底要给 `mid` 加一还是减一，`while` 里到底用 `<=` 还是 `<`。

你要是没有正确理解这些细节，写二分肯定就是玄学编程，有没有 bug 只能靠菩萨保佑（谁写谁知道）。我特意写了一首诗来歌颂该算法，概括本文的主要内容，建议保存（手动狗头）：

二分搜索升天词  
作者：labuladong

二分搜索不好记，左右边界让人迷。  
小于等于变小于，`mid` 加一又减一。  
就算这样还没完，`return` 应否再 -1？  
信心满满刷力扣，AC 比率二十一。  
我本将心向明月，奈何明月照沟渠！  
问君能有几多愁？恰似深情喂了狗。

labuladong 从天降，一同手撕算法题。  
赠君一法写二分，不用拜佛与念经。  
**管他左侧还右侧，搜索区间定乾坤。**

搜索一个元素时，搜索区间两端闭。  
`while` 条件带等号，否则需要打补丁。  
`if` 相等就返回，其他的事甭操心。  
`mid` 必须加减一，因为区间两端闭。  
`while` 结束就凉了，凄凄惨惨返 -1。

搜索左右边界时，搜索区间要阐明。  
左闭右开最常见，其余逻辑便自明：  
`while` 要用小于号，这样才能不漏掉。  
`if` 相等别返回，利用 `mid` 锁边界。  
`mid` 加一或减一？要看区间开或闭。  
`while` 结束不算完，因为你还没返回。  
索引可能出边界，`if` 检查保平安。

**左闭右开最常见，难道常见就合理？**  
labuladong 不信邪，偏要改成两端闭。  
搜索区间记于心，或开或闭有何异？  
二分搜索三变体，逻辑统一容易记。  
一套框架改两行，胜过千言和万语。

此等神人何处寻？全靠缘分不可期！  
**labuladong 公众号，开启算法新天地。**  
关注标星加分享，“下次一定”不可取。

本文就来探究几个最常用的二分查找场景：寻找一个数、寻找左侧边界、寻找右侧边界。而且，我们就是要深入细节，比如不等号是否应该带等号，`mid` 是否应该加一等等。分析这些细节的差异以及出现这些差异的原因，保证你能灵活准确地写出正确的二分查找算法。

另外再声明一下，对于二分搜索的每一个场景，本文还会探讨多种代码写法，目的是为了让你理解出现这些细微差异的本质原因，最起码你看到别人的代码时不会懵逼。实际上这些写法没有优劣之分，你喜欢哪种就用哪种好了。

## 零、二分查找框架

```
int binarySearch(int[] nums, int target) {
    int left = 0, right = ...;

    while(...) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            ...
        } else if (nums[mid] < target) {
            left = ...
        } else if (nums[mid] > target) {
            right = ...
        }
    }
    return ...;
}
```

**分析二分查找的一个技巧是：不要出现 `else`，而是把所有情况用 `else if` 写清楚，这样可以清楚地展现所有细节。**本文都会使用 `else if`，旨在讲清楚，读者理解后可自行简化。

其中 `...` 标记的部分，就是可能出现细节问题的地方，当你见到一个二分查找的代码时，首先注意这几个地方。后文用实例分析这些地方能有什么样的变化。

**另外提前说明一下，计算 `mid` 时需要防止溢出**，代码中 `left + (right - left) / 2` 就和 `(left + right) / 2` 的结果相同，但是有效防止了 `left` 和 `right` 太大，直接相加导致溢出的情况。

## 一、寻找一个数（基本的二分搜索）

这个场景是最简单的，可能也是大家最熟悉的，即搜索一个数，如果存在，返回其索引，否则返回 `-1`。

```

int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1; // 注意

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1; // 注意
        else if (nums[mid] > target)
            right = mid - 1; // 注意
    }
    return -1;
}

```

这段代码可以解决力扣第 704 题「二分查找」，但我们深入探讨一下其中的细节。

## 1、为什么 while 循环的条件中是 <=，而不是 <？

答：因为初始化 `right` 的赋值为 `nums.length - 1`，即最后一个元素的索引，而不是 `nums.length`。

这二者可能出现在不同功能的二分查找中，区别是：前者相当于两端都闭区间 `[left, right]`，后者相当于左闭右开区间 `[left, right)`，因为索引大小为 `nums.length` 是越界的。

我们这个算法中使用的是前者 `[left, right]` 两端都闭的区间。**这个区间其实就是每次进行搜索的区间。**

什么时候应该停止搜索呢？当然，找到了目标值的时候可以终止：

```

if(nums[mid] == target)
    return mid;

```

但如果没找到，就需要 while 循环终止，然后返回 -1。那 while 循环什么时候应该终止？**搜索区间为空的时候应该终止**，意味着你没得找了，就等于没找到嘛。

`while(left <= right)` 的终止条件是 `left == right + 1`，写成区间的形式就是

`[right + 1, right]`，或者带个具体的数字进去 `[3, 2]`，可见**这时候区间为空**，因为没有数字既大于等于 3 又小于等于 2 的吧。所以这时候 `while` 循环终止是正确的，直接返回 -1 即可。

`while(left < right)` 的终止条件是 `left == right`，写成区间的形式就是 `[right, right]`，或者带个具体的数字进去 `[2, 2]`，**这时候区间非空**，还有一个数 2，但此时 `while` 循环终止了。也就是说这区间 `[2, 2]` 被漏掉了，索引 2 没有被搜索，如果这时候直接返回 -1 就是错误的。

当然，如果你非要用 `while(left < right)` 也可以，我们已经知道了出错的原因，就打个补丁好了：

```
//...
while(left < right) {
    // ...
}
return nums[left] == target ? left : -1;
```

## 2、为什么 `left = mid + 1`，`right = mid - 1`？我看有的代码是 `right = mid` 或者 `left = mid`，没有这些加加减减，到底怎么回事，怎么判断？

答：这也是二分查找的一个难点，不过只要你能理解前面的内容，就能够很容易判断。

刚才明确了「搜索区间」这个概念，而且本算法的搜索区间是两端都闭的，即 `[left, right]`。那么当我们发现索引 `mid` 不是要找的 `target` 时，下一步应该去搜索哪里呢？

当然是去搜索区间 `[left, mid-1]` 或者区间 `[mid+1, right]` 对不对？**因为 `mid` 已经搜索过，应该从搜索区间中去除。**

## 3、此算法有什么缺陷？

答：至此，你应该已经掌握了该算法的所有细节，以及这样处理的原因。但是，这个算法存在局限性。

比如说给你有序数组 `nums = [1,2,2,2,3]`，`target` 为 2，此算法返回的索引是 2，没错。但是如果我想得到 `target` 的左侧边界，即索引 1，或者我想得到 `target` 的右侧边界，即索引 3，这样的话此算法是无法处理的。

这样的需求很常见，你也许会说，找到一个 `target`，然后向左或向右线性搜索不行吗？可以，但是不好，因为这样难以保证二分查找对数级的复杂度了。

我们后续的算法就来讨论这两种二分查找的算法。

## 二、寻找左侧边界的二分搜索

以下是最常见的代码形式，其中的标记是需要注意的细节：

```
int left_bound(int[] nums, int target) {  
    if (nums.length == 0) return -1;  
    int left = 0;  
    int right = nums.length; // 注意  
  
    while (left < right) { // 注意  
        int mid = left + (right - left) / 2;  
        if (nums[mid] == target) {  
            right = mid;  
        } else if (nums[mid] < target) {  
            left = mid + 1;  
        } else if (nums[mid] > target) {  
            right = mid; // 注意  
        }  
    }  
    return left;  
}
```

### 1、为什么 while 中是 `<` 而不是 `<=`？

答：用相同的方法分析，因为 `right = nums.length` 而不是 `nums.length - 1`。因此每次循环的「搜索区间」是 `[left, right)` 左闭右开。

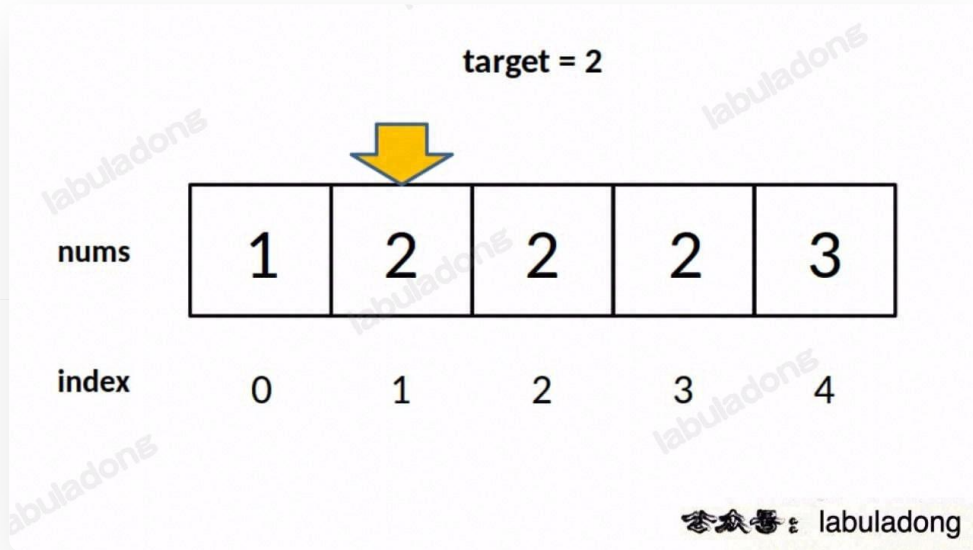
`while(left < right)` 终止的条件是 `left == right`，此时搜索区间 `[left, left)` 为空，所以可以正确终止。

PS：这里先要说一个搜索左右边界和上面这个算法的一个区别，也是很多读者问的：**刚才的 `right` 不是 `nums.length - 1` 吗，为啥这里非要写成 `nums.length` 使得「搜索区间」变成左闭右开呢？**

因为对于搜索左右侧边界的二分查找，这种写法比较普遍，我就拿这种写法举例了，保证你以后遇到这类代码可以理解。你非要用两端都闭的写法反而更简单，我会在后面写相关的代码，把三种二分搜索都用一种两端都闭的写法统一起来，你耐心往后看就行了。

## 2、为什么没有返回 -1 的操作？如果 `nums` 中不存在 `target` 这个值，怎么办？

答：因为要一步一步来，先理解一下这个「左侧边界」有什么特殊含义：



对于这个数组，算法会返回索引 1。

这个索引 1 的含义可以解读为「`nums` 中小于 2 的元素有 1 个」。

比如对于有序数组 `nums = [2, 3, 5, 7]`，`target = 1`，算法会返回 0，含义是：`nums` 中小于 1 的元素有 0 个。

再比如说 `nums = [2, 3, 5, 7]`，`target = 8`，算法会返回 4，含义是：`nums` 中小于 8 的元素有 4 个。

PS：对于 `target` 不存在 `nums` 中的情况，函数的返回值还可以有多种理解方式，详见 [随机权重算法](#) 中对二分搜索的运用。

综上所述，函数的返回值（即 `left` 变量的值）取值区间是闭区间 `[0, nums.length]`，所以我们简单添加两行代码就能在正确的时候 `return -1`：

```
while (left < right) {  
    //...  
}  
// target 比所有数都大
```

```
if (left == nums.length) return -1;
// 类似之前算法的处理方式
return nums[left] == target ? left : -1;
```

### 3、为什么 `left = mid + 1`, `right = mid` ? 和之前的算法不一样?

答：这个很好解释，因为我们的「搜索区间」是 `[left, right)` 左闭右开，所以当 `nums[mid]` 被检测之后，下一步应该去 `mid` 的左侧或者右侧区间搜索，即 `[left, mid)` 或 `[mid + 1, right)`。

### 4、为什么该算法能够搜索左侧边界?

答：关键在于对于 `nums[mid] == target` 这种情况的处理：

```
if (nums[mid] == target)
    right = mid;
```

可见，找到 `target` 时不要立即返回，而是缩小「搜索区间」的上界 `right`，在区间 `[left, mid)` 中继续搜索，即不断向左收缩，达到锁定左侧边界的目的。

### 5、为什么返回 `left` 而不是 `right` ?

答：都是一样的，因为 `while` 终止的条件是 `left == right`。

### 6、能不能想办法把 `right` 变成 `nums.length - 1`，也就是继续使用两边都闭的「搜索区间」？这样就可以和第一种二分搜索在某种程度上统一起来了。

答：当然可以，只要你明白了「搜索区间」这个概念，就能有效避免漏掉元素，随便你怎么改都行。下面我们严格根据逻辑来修改：

因为你非要让搜索区间两端都闭，所以 `right` 应该初始化为 `nums.length - 1`，`while` 的终止条件应该是 `left == right + 1`，也就是其中应该用 `<=`：

```
int left_bound(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
```

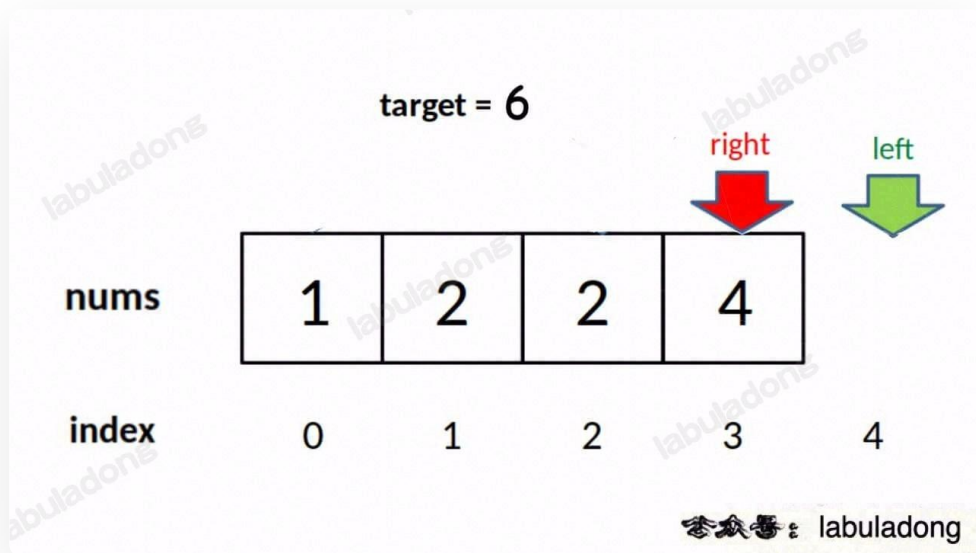


```
    // if else ...  
}
```

因为搜索区间是两端都闭的，且现在是搜索左侧边界，所以 `left` 和 `right` 的更新逻辑如下：

```
if (nums[mid] < target) {  
    // 搜索区间变为 [mid+1, right]  
    left = mid + 1;  
} else if (nums[mid] > target) {  
    // 搜索区间变为 [left, mid-1]  
    right = mid - 1;  
} else if (nums[mid] == target) {  
    // 收缩右侧边界  
    right = mid - 1;  
}
```

由于 `while` 的退出条件是 `left == right + 1`，所以当 `target` 比 `nums` 中所有元素都大时，会存在以下情况使得索引越界：



因此，最后返回结果的代码应该检查越界情况：

```
if (left >= nums.length || nums[left] != target)
```

```
    return -1;
return left;
```

至此，整个算法就写完了，完整代码如下：

```
int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    // 搜索区间为 [left, right]
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩右侧边界
            right = mid - 1;
        }
    }
    // 检查出界情况
    if (left >= nums.length || nums[left] != target) {💡
        return -1;
    }
    return left;
}
```

这样就和第一种二分搜索算法统一了，都是两端都闭的「搜索区间」，而且最后返回的也是 `left` 变量的值。只要把住二分搜索的逻辑，两种形式大家看自己喜欢哪种记哪种吧。

### 三、寻找右侧边界的二分查找

类似寻找左侧边界的算法，这里也会提供两种写法，还是先写常见的左闭右开的写法，只有两处和搜索左侧边界不同：

```
int right_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
```

```

int left = 0, right = nums.length;

while (left < right) {
    int mid = left + (right - left) / 2;
    if (nums[mid] == target) {
        left = mid + 1; // 注意
    } else if (nums[mid] < target) {
        left = mid + 1;
    } else if (nums[mid] > target) {
        right = mid;
    }
}
return left - 1; // 注意
}

```

## 1、为什么这个算法能够找到右侧边界？

答：类似地，关键点还是这里：

```

if (nums[mid] == target) {
    left = mid + 1;
}

```

当 `nums[mid] == target` 时，不要立即返回，而是增大「搜索区间」的左边界 `left`，使得区间不断向右靠拢，达到锁定右侧边界的目的。

## 2、为什么最后返回 `left - 1` 而不像左侧边界的函数，返回 `left`？而且我觉得这里既然是搜索右侧边界，应该返回 `right` 才对。

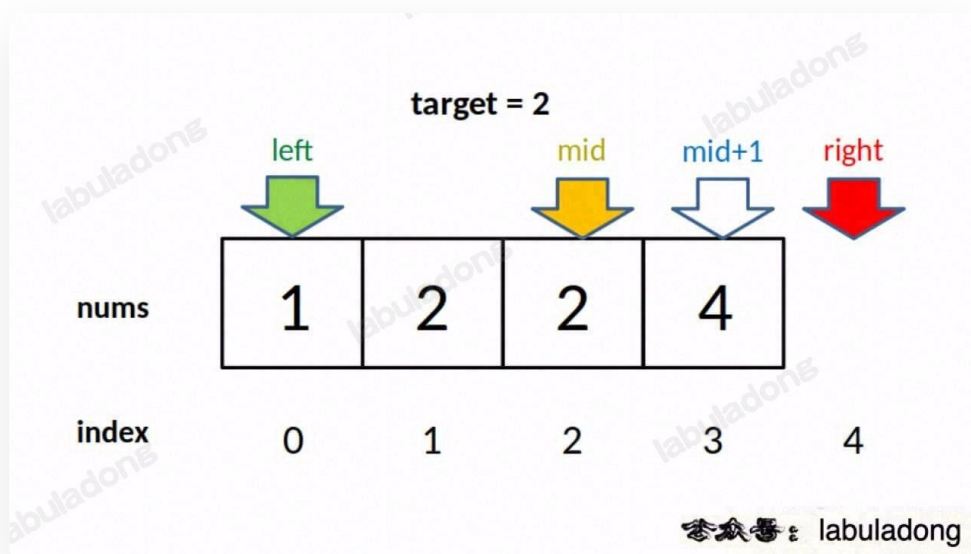
答：首先，while 循环的终止条件是 `left == right`，所以 `left` 和 `right` 是一样的，你非要体现右侧的特点，返回 `right - 1` 好了。

至于为什么要减一，这是搜索右侧边界的一个特殊点，关键在锁定右边界时的这个条件判断：

```

// 增大 left，锁定右侧边界
if (nums[mid] == target) {
    left = mid + 1;
    // 这样想: mid = left - 1
}

```



因为我们对 `left` 的更新必须是 `left = mid + 1`，就是说 `while` 循环结束时，`nums[left]` 一定不等于 `target` 了，而 `nums[left-1]` 可能是 `target`。

至于为什么 `left` 的更新必须是 `left = mid + 1`，当然是为了锁定右侧边界，就不再赘述。

### 3、为什么没有返回 -1 的操作？如果 `nums` 中不存在 `target` 这个值，怎么办？

答：类似之前的左侧边界搜索，因为 `while` 的终止条件是 `left == right`，就是说 `left` 的取值范围是 `[0, nums.length]`，所以可以添加两行代码，正确地返回 -1：

```
while (left < right) {  
    // ...  
}  
if (left == 0) return -1;  
return nums[left-1] == target ? (left-1) : -1;
```

### 4、是否也可以把这个算法的「搜索区间」也统一成两端都闭的形式呢？这样这三个写法就完全统一了，以后就可以闭着眼睛写出来了。

答：当然可以，类似搜索左侧边界的统一写法，其实只要改两个地方就行了：

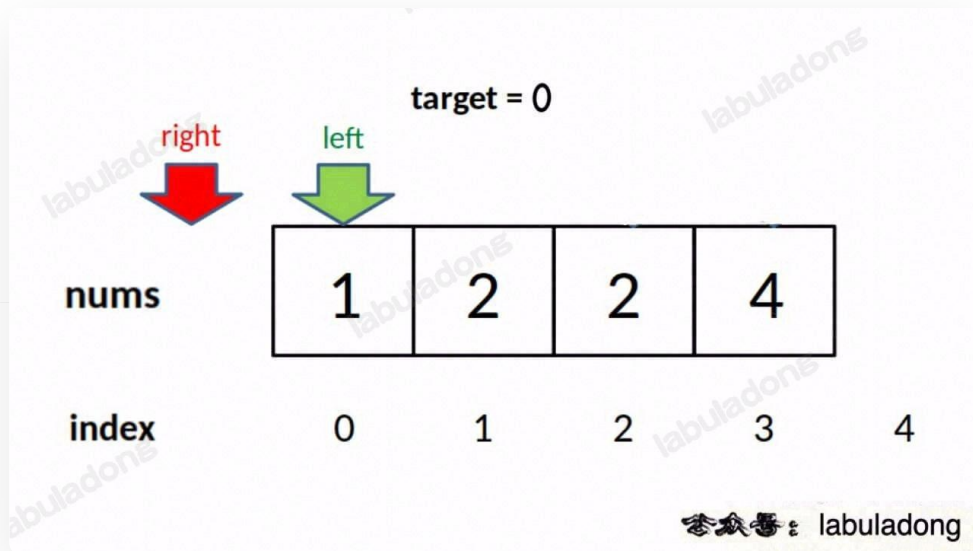
```
int right_bound(int[] nums, int target) {
```

```

int left = 0, right = nums.length - 1;
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (nums[mid] < target) {
        left = mid + 1;
    } else if (nums[mid] > target) {
        right = mid - 1;
    } else if (nums[mid] == target) {
        // 这里改成收缩左侧边界即可
        left = mid + 1;
    }
}
// 这里改为检查 right 越界的情况，见下图
if (right < 0 || nums[right] != target) {💡
    return -1;
}
return right;
}

```

当 `target` 比所有元素都小时，`right` 会被减到 -1，所以需要在最后防止越界：



至此，搜索右侧边界的二分查找的两种写法也完成了，其实将「搜索区间」统一成两端都闭反而更容易记忆，你说是吧？

## 四、逻辑统一

有了搜索左右边界的二分搜索，你可以去解决力扣第 34 题「[在排序数组中查找元素的第一个和最后一个位置](#)」，

接下来梳理一下这些细节差异的因果逻辑：

### 第一个，最基本的二分查找算法：

因为我们初始化 `right = nums.length - 1`  
所以决定了我们的「搜索区间」是 `[left, right]`  
所以决定了 `while (left <= right)`  
同时也决定了 `left = mid + 1` 和 `right = mid - 1`

因为我们只需找到一个 `target` 的索引即可  
所以当 `nums[mid] == target` 时可以立即返回

### 第二个，寻找左侧边界的二分查找：

因为我们初始化 `right = nums.length`  
所以决定了我们的「搜索区间」是 `[left, right)`  
所以决定了 `while (left < right)`  
同时也决定了 `left = mid + 1` 和 `right = mid`

因为我们需找到 `target` 的最左侧索引  
所以当 `nums[mid] == target` 时不要立即返回  
而要收紧右侧边界以锁定左侧边界

### 第三个，寻找右侧边界的二分查找：

因为我们初始化 `right = nums.length`  
所以决定了我们的「搜索区间」是 `[left, right)`  
所以决定了 `while (left < right)`  
同时也决定了 `left = mid + 1` 和 `right = mid`

因为我们需找到 `target` 的最右侧索引  
所以当 `nums[mid] == target` 时不要立即返回  
而要收紧左侧边界以锁定右侧边界

又因为收紧左侧边界时必须 `left = mid + 1`  
所以最后无论返回 `left` 还是 `right`，必须减一

对于寻找左右边界的二分搜索，常见的手法是使用左闭右开的「搜索区间」，**我们还根据逻辑将「搜索区间」全都统一成了两端都闭，便于记忆，只要修改两处即可变化出三种写法：**

```
int binary_search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while(left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 直接返回
            return mid;
        }
    }
    // 直接返回
    return -1;
}
```

```
int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回，锁定左侧边界
            right = mid - 1;
        }
    }
    // 最后要检查 left 越界的情况
    if (left >= nums.length || nums[left] != target) {💡
        return -1;
    }
    return left;
}
```

```
int right_bound(int[] nums, int target) {
```

```

int left = 0, right = nums.length - 1;
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (nums[mid] < target) {
        left = mid + 1;
    } else if (nums[mid] > target) {
        right = mid - 1;
    } else if (nums[mid] == target) {
        // 别返回，锁定右侧边界
        left = mid + 1;
    }
}
// 最后要检查 right 越界的情况
if (right < 0 || nums[right] != target) {💡
    return -1;
}
return right;
}

```

如果以上内容你都能理解，那么恭喜你，二分查找算法的细节不过如此。通过本文，你学会了：

- 1、分析二分查找代码时，不要出现 else，全部展开成 else if 方便理解。
- 2、注意「搜索区间」和 while 的终止条件，如果存在漏掉的元素，记得在最后检查。
- 3、如需定义左闭右开的「搜索区间」搜索左右边界，只要在 `nums[mid] == target` 时做修改即可，搜索右侧时需要减一。
- 4、如果将「搜索区间」全都统一成两端都闭，好记，只要稍改 `nums[mid] == target` 条件处的代码和返回的逻辑即可，推荐拿小本本记下，作为二分搜索模板。

最后我想说，以上二分搜索的框架属于「术」的范畴，如果上升到「道」的层面，**二分思维的精髓就是：通过已知信息尽可能多地收缩（折半）搜索空间**，从而增加穷举效率，快速找到目标。

理解本文能保证你写出正确的二分查找的代码，但实际题目中不会直接让你写二分代码，我会在[二分查找的变体](#)和[二分查找的运用](#)中进一步讲解如何把二分思维运用到更多算法题中。

---

## ► 引用本文的题目

---

## ► 引用本文的文章

---

-----