

# 线段树 (Segment Tree) 数据结构

黄京

Jul 06, 2025

在算法和数据结构的领域中，处理动态数组的区间查询（如求和、求最大值或最小值）是一个常见需求。朴素方法中，对数组进行区间查询需要遍历整个区间，时间复杂度为  $O(n)$ ；而单点更新只需  $O(1)$  时间。这种不对称性在动态数据场景下成为性能瓶颈，尤其当查询操作频繁时，整体效率急剧下降。线段树正是为解决这一问题而设计的平衡数据结构，它通过预处理构建树形结构，将区间查询和单点更新的时间复杂度均优化到  $O(\log n)$ 。线段树的核心价值在于高效处理区间操作，适用于区间求和、区间最值计算以及批量区间修改等场景，例如在实时数据监控或大规模数值分析中。

## 1 线段树的核心思想

线段树的核心思想基于分而治之策略，将大区间递归划分为不相交的子区间，形成一棵二叉树结构。这种划分充分利用了空间换时间的原则：在构建阶段预处理并存储每个子区间的计算结果，从而在查询时避免重复遍历。线段树的关键性质包括其作为完全二叉树的特性，通常用数组存储以提高效率；叶子节点直接对应原始数组元素，而非叶子节点则存储子区间的合并结果（如求和或最值）。例如，对于区间  $[l, r]$ ，其值由子区间  $[l, \text{mid}]$  和  $[\text{mid} + 1, r]$  推导而来，其中  $\text{mid} = l + \lfloor (r - l)/2 \rfloor$ ，确保划分的平衡性。

## 2 线段树的逻辑结构与存储

线段树的逻辑结构始于根节点，代表整个区间  $[0, n - 1]$ ；每个父节点  $[l, r]$  的左子节点覆盖  $[l, \text{mid}]$ ，右子节点覆盖  $[\text{mid} + 1, r]$ ，其中  $\text{mid}$  是中点值。这种递归划分确保所有子区间互不重叠。存储方式采用数组实现而非指针结构，以减少内存开销。数组大小需安全预留，通常为  $4n$ ，这是基于二叉树最坏情况的空间推导：一棵高度为  $h$  的完全二叉树最多有  $2^{h+1} - 1$  个节点，而  $h \approx \log_2 n$ ，因此  $4n$  足够覆盖所有节点。在 Python 中，初始化存储数组的代码如下：

```
tree = [0] * (4 * n) # 为线段树预留大小为 4*n 的数组
```

这段代码创建一个长度为  $4n$  的数组 `tree`，初始值设为 0。索引从 0 开始，根节点位于索引 0，左子节点通过  $2 \times \text{node} + 1$  计算，右子节点通过  $2 \times \text{node} + 2$  计算。这种索引技巧避免了指针操作，提升访问速度。

## 3 核心操作原理与实现

线段树的核心操作包括构建、查询和更新。构建操作通过递归实现：从根节点开始，将区间划分为左右子树，直到叶子节点存储原始数组值，然后回溯合并结果。以下是 Python 实现构建函数的代码：

```

1 def build_tree(arr, tree, node, start, end):
2     if start == end: # 叶子节点: 区间长度为 1
3         tree[node] = arr[start] # 直接存储数组元素值
4     else:
5         mid = (start + end) // 2 # 计算区间中点
6         build_tree(arr, tree, 2*node+1, start, mid) # 递归构建左子树
7         build_tree(arr, tree, 2*node+2, mid+1, end) # 递归构建右子树
8         tree[node] = tree[2*node+1] + tree[2*node+2] # 合并结果 (求和为例)

```

这段代码中，`arr` 是原始数组，`tree` 是存储树结构的数组，`node` 是当前节点索引，`start` 和 `end` 定义当前区间。当 `start == end` 时，处理叶子节点；否则，计算中点 `mid`，递归构建左右子树（左子树索引为  $2 \times node + 1$ ，右子树为  $2 \times node + 2$ ），最后合并子树结果到当前节点。查询操作基于区间关系处理：如果查询区间  $[q_l, q_r]$  完全包含当前节点区间  $[l, r]$ ，则直接返回节点值；若部分重叠，则递归查询左右子树；若不相交，返回中性值（如 0 用于求和）。单点更新类似，递归定位到叶子节点后修改值，并回溯更新父节点。区间更新可引入延迟传播优化，但基础实现中，我们优先聚焦单点操作。

## 4 关键实现细节与边界处理

实现线段树时，边界处理至关重要，以避免死循环或逻辑错误。区间划分使用公式  $mid = l + \lfloor (r - l)/2 \rfloor$  而非简单  $(l + r)//2$ ，防止整数溢出和死循环。查询合并逻辑需根据操作类型调整：区间求和时，结果为左子树和加右子树和；区间最值时，结果为  $\max(left\_max, right\_max)$  或  $\min(\cdot)$ 。索引技巧确保父子关系正确，根节点索引为 0，左子节点为  $2 \times node + 1$ ，右子节点为  $2 \times node + 2$ 。递归终止条件必须明确：当 `start == end` 时处理叶子节点。例如，在查询函数中，边界条件包括：

```

def query_tree(tree, node, start, end, ql, qr):
1    if qr < start or end < ql: # 查询区间与当前区间无重叠
2        return 0 # 返回中性值 (求和时为 0)
3    if ql <= start and end <= qr: # 当前区间完全包含在查询区间内
4        return tree[node] # 直接返回存储值
5    mid = (start + end) // 2
6    left_sum = query_tree(tree, 2*node+1, start, mid, ql, qr) # 查左子树
7    right_sum = query_tree(tree, 2*node+2, mid+1, end, ql, qr) # 查右子树
8    return left_sum + right_sum # 合并结果

```

这段代码处理三种情况：无重叠返回中性值；完全包含返回节点值；部分重叠则递归查询并合并。开闭区间处理需一致，通常使用闭区间  $[l, r]$  以避免混淆。

## 5 复杂度分析

线段树的复杂度分析揭示其效率优势。构建操作的时间复杂度为  $O(n)$ ，因为每个节点仅处理一次，总节点数约为  $2n - 1$ 。查询和单点更新的时间复杂度均为  $O(\log n)$ ，源于树高度为  $\lceil \log_2 n \rceil$ ，递归路径长度对数级。空间复杂度为  $O(n)$ ：原始数据占  $O(n)$ ，树存储数组大小为  $O(4n)$ ，但常数因子可忽略，整体线性。与树状数组

(Fenwick Tree) 对比时，线段树更通用：支持任意区间操作如最值查询；而树状数组仅优化前缀操作，代码更简洁但功能受限。例如，树状数组的区间求和需两个前缀查询，但无法直接处理区间最值。

## 6 实战代码实现 (Python 示例)

以下是完整的线段树 Python 类实现，支持区间求和和单点更新：

```
1 class SegmentTree:
2     def __init__(self, arr):
3         self.n = len(arr)
4         self.tree = [0] * (4 * self.n) # 初始化存储数组
5         self.arr = arr
6         self._build(0, 0, self.n-1) # 从根节点开始构建
7
8     def _build(self, node, start, end):
9         if start == end: # 叶子节点
10             self.tree[node] = self.arr[start] # 存储数组元素
11         else:
12             mid = (start + end) // 2
13             left_node = 2 * node + 1 # 左子节点索引
14             right_node = 2 * node + 2 # 右子节点索引
15             self._build(left_node, start, mid) # 构建左子树
16             self._build(right_node, mid+1, end) # 构建右子树
17             self.tree[node] = self.tree[left_node] + self.tree[right_node] # 合并求和
18
19     def query(self, ql, qr):
20         return self._query(0, 0, self.n-1, ql, qr) # 从根节点开始查询
21
22     def _query(self, node, start, end, ql, qr):
23         if qr < start or end < ql: # 无重叠
24             return 0
25         if ql <= start and end <= qr: # 完全包含
26             return self.tree[node]
27         mid = (start + end) // 2
28         left_sum = self._query(2*node+1, start, mid, ql, qr) # 查询左子树
29         right_sum = self._query(2*node+2, mid+1, end, ql, qr) # 查询右子树
30         return left_sum + right_sum # 返回合并结果
31
32     def update(self, index, value):
33         diff = value - self.arr[index] # 计算值变化量
```

```

35     self.arr[index] = value # 更新原始数组
36     self._update(0, 0, self.n-1, index, diff) # 从根节点开始更新

37 def _update(self, node, start, end, index, diff):
38     if start == end: # 到达叶子节点
39         self.tree[node] += diff # 更新节点值
40     else:
41         mid = (start + end) // 2
42         if index <= mid: # 目标索引在左子树
43             self._update(2*node+1, start, mid, index, diff)
44         else: # 目标索引在右子树
45             self._update(2*node+2, mid+1, end, index, diff)
46         self.tree[node] = self.tree[2*node+1] + self.tree[2*node+2] # 回溯更新父节点

```

这个类包含初始化构建 `__init__`、区间查询 `query` 和单点更新 `update` 方法。在 `_build` 方法中，递归划分区间并存储求和结果；`_query` 处理查询逻辑，根据区间重叠情况递归；`_update` 定位到叶子节点更新值，并回溯修正父节点。测试用例可验证正确性，例如：

```

arr = [1, 3, 5, 7, 9]
1 st = SegmentTree(arr)
2 print(st.query(1, 3)) # 输出: 3+5+7=15
3 st.update(2, 10) # 更新索引 2 的值从 5 到 10
4 print(st.query(1, 3)) # 输出: 3+10+7=20

```

## 7 经典应用场景

线段树在算法竞赛和工程中广泛应用。区间统计问题如 LeetCode 307「区域和检索 - 数组可修改」，直接使用线段树实现高效查询和更新。区间最值问题中，线段树可求解滑动窗口最大值，通过构建存储最大值的树结构，在  $O(\log n)$  时间响应查询。衍生算法包括扫描线算法，用于计算矩形面积并集；线段树处理事件点的区间覆盖，时间复杂度  $O(n \log n)$ 。动态区间问题如逆序对统计，也可结合线段树优化。这些场景凸显线段树在高效处理动态数据中的核心作用。

## 8 常见问题与优化方向

实现线段树时，易错点包括区间边界混淆（如使用开闭区间不一致）和递归栈溢出（对大数组可能引发递归深度限制）。解决方案是统一使用闭区间  $[l, r]$ ，并考虑迭代实现或尾递归优化。进阶优化方向有动态开点线段树，适用于稀疏数据，避免预分配大数组；通过懒标记仅在需要时创建节点，节省空间。离散化技术处理大范围数据，将原始值映射到紧凑索引，减少树规模。例如，坐标范围  $[1, 10^9]$  可离散化为  $[0, k - 1]$ ， $k$  为唯一值数量。线段树的核心价值在于高效处理动态区间操作，将查询和更新的时间复杂度平衡到  $O(\log n)$ 。学习路径建议从基础区间求和开始，逐步扩展到区间最值；进阶阶段引入延迟传播优化区间更新，最终探索持久化线段树支持历史版本查询。终极目标是理解分治思想在数据结构中的优雅体现：通过递归划分和结果合并，将复杂问题分解

为可管理的子问题。

## 9 附录

可视化工具如 VisuAlgo 提供线段树交互演示，帮助理解构建和查询过程。相关 LeetCode 练习题包括「307. 区域和检索 - 数组可修改」、「315. 计算右侧小于当前元素的个数」等。参考书籍推荐《算法导论》第 14 章，详细讨论区间树变体；论文如 Bentley 的「Decomposable Searching Problems」奠定理论基础。