c13n #33

c13n

2025 年 11 月 19 日

第 I 部

无锁数据结构的设计原理与实现

杨其臻

Sep 28, 2025

1 副标题：告别锁的阻塞，探索高性能并发数据结构的核心奥秘

在现代并发编程中，锁作为传统的同步机制，虽然简单易用，却常常成为系统性能的瓶颈。锁会导致线程串行化，引发上下文切换的开销，并在高并发场景下产生严重的竞争问题。更糟糕的是，锁可能带来死锁、活锁和优先级反转等致命缺陷，随着多核处理器的普及，这些问题的负面影响被放大，限制了系统的可伸缩性。相比之下，无锁数据结构承诺更高的并发度和可伸缩性，能够避免锁相关的经典问题，对于实时系统和低延迟应用具有关键价值。本文旨在帮助读者理解无锁数据结构的核心设计原理，掌握实现无锁结构的关键技术与挑战，并通过经典案例如无锁队列和无锁栈来剖析实现细节，同时探讨无锁编程的适用场景与潜在陷阱。

2 基石：无锁编程的核心概念

无锁编程的核心概念源于对并发控制的重新思考。学术上，无锁被定义为在系统范围的任意延迟或故障下，保证至少有一个线程能够继续执行。通俗地说，无锁编程不使用互斥锁，而是通过原子操作如比较并交换（CAS）来保证并发正确性。关键特性包括无锁、无等待和无阻塞；无锁保证系统整体进度，无等待则更强，确保每个线程的进度都有保障，而无阻塞是上述两者的统称。这些特性使得无锁编程在高性能计算中备受青睐。

硬件基石是支撑无锁编程的关键，原子操作和内存序构成了其基础。比较并交换（CAS）操作是无锁编程的瑞士军刀，它允许原子地比较一个内存位置的值，并在匹配时更新为新值。其他原子指令如取并加（Fetch-And-Add）和加载链接/存储条件（LL/SC）也在不同架构中发挥作用。内存模型和内存屏障则确保指令执行的正确顺序；指令重排可能导致意想不到的并发问题，因此需要内存屏障来强制内存访问顺序。在 C++ 中，std::memory_order 提供了不同级别的内存序，例如宽松（relaxed）、获取（acquire）、释放（release）、获取释放（acq_rel）和顺序一致（seq_cst），这些枚举值帮助开发者精确控制内存可见性。

3 挑战：无锁编程的“三座大山”

无锁编程虽然强大，却面临三大挑战：ABA 问题、内存回收问题以及复杂度与正确性问题。ABA 问题描述了一个值从 A 变为 B 又变回 A，而 CAS 操作无法感知中间变化，这在链表操作中尤为常见，例如节点被释放后重用可能导致数据损坏。解决方案包括标签指针，它利用指针的高位作为版本号来检测变化；风险指针，线程声明自己正在访问的指针以防止其被回收；以及引用计数，通过原子计数管理对象生命周期，但在无锁环境下实现较为复杂。

内存回收问题涉及当一个线程准备释放内存时，另一个线程可能仍持有该内存的指针并准备访问，这会导致悬空指针和未定义行为。解决方案有多种：风险指针通过线程本地存储来标记正在使用的指针；引用计数在无锁环境下需要原子操作来维护；基于时代的回收（Epoch Based Reclamation）适用于读多写少的场景，它将回收延迟到安全时期；基于静默状态的回收（Quiescent State Based Reclamation）常用于读-复制-更新（RCU）模式，它在线程进入静默状态时回收内存。复杂度与正确性问题体现在代码难以设计和验证，测试困难且竞态条件难以复现，同时对平台和编译器的依赖性强，这要求开发者在实现时格外

谨慎。

4　实战：从零实现一个无锁队列

在设计无锁队列时，我们通常选择基于链表的实现，因为它能动态扩展并避免固定容量的限制。队列结构包含头指针和尾指针，指向单链表的起始和结束节点。核心操作包括入队（Enqueue）和出队（Dequeue），这些操作通过原子指令确保并发安全。

数据结构定义是基础，我们使用节点结构来存储数据和下一个指针，队列结构则维护头尾指针。例如，在 C++ 中，我们可以定义节点为包含整型数据和节点指针的结构，队列类则封装头尾指针及其操作。

入队操作涉及三个关键步骤：首先创建新节点，然后循环使用 CAS 更新尾节点的 next 指针，最后处理“滞后尾”问题，即通过另一个 CAS 循环更新队列的 tail 指针。出队操作则包括读取 head、tail 和 next 指针，判断队列是否为空，循环 CAS 更新 head 指针，并处理出队数据的内存回收，这里可以集成之前讨论的内存回收方案，例如使用风险指针来安全释放内存。

以下是一个简化的 C++ 代码示例，展示无锁队列的核心部分。我们使用 std::atomic 来实现原子操作，并添加详细注释以解释关键步骤。

```
#include <atomic>

struct Node {
    int data;
    std::atomic<Node*> next;
    Node(int value) : data(value), next(nullptr) {}
};

class LockFreeQueue {
private:
    std::atomic<Node*> head;
    std::atomic<Node*> tail;
public:
    LockFreeQueue() {
        Node* dummy = new Node(0);
        head.store(dummy);
        tail.store(dummy);
    }

    void enqueue(int value) {
        Node* newNode = new Node(value);
        while (true) {
            Node* currentTail = tail.load(std::memory_order_acquire);
            Node* nextTail = currentTail->next.load(std::
```

```
            ↪ memory_order_acquire);
        if (currentTail == tail.load(std::memory_order_acquire)) {
            if (nextTail == nullptr) {
                if (currentTail->next.compare_exchange_weak(nextTail,
                    ↪ newNode, std::memory_order_release)) {
                    tail.compare_exchange_weak(currentTail, newNode,
                        ↪ std::memory_order_release);
                    return;
                }
            } else {
                tail.compare_exchange_weak(currentTail, nextTail, std::
                    ↪ memory_order_release);
            }
        }
    }
}

int dequeue() {
    while (true) {
        Node* currentHead = head.load(std::memory_order_acquire);
        Node* currentTail = tail.load(std::memory_order_acquire);
        Node* nextHead = currentHead->next.load(std::
            ↪ memory_order_acquire);
        if (currentHead == head.load(std::memory_order_acquire)) {
            if (currentHead == currentTail) {
                if (nextHead == nullptr) {
                    return -1; // 队列为空
                }
                tail.compare_exchange_weak(currentTail, nextHead, std::
                    ↪ memory_order_release);
            } else {
                int value = nextHead->data;
                if (head.compare_exchange_weak(currentHead, nextHead,
                    ↪ std::memory_order_release)) {
                    // 此处需处理内存回收，例如使用风险指针
                    delete currentHead;
                    return value;
                }
            }
        }
    }
}
```

```
};
```

在这段代码中，enqueue 函数通过循环 CAS 确保新节点被正确链接到队列尾部。首先，它加载当前尾指针和其 next 指针，然后检查尾指针是否未被其他线程修改。如果 next 指针为空，则尝试原子地将其设置为新节点，成功后更新尾指针。dequeue 函数类似，它加载头指针和尾指针，检查队列状态，如果队列非空，则原子地更新头指针并返回数据。内存回收部分在出队时删除旧头节点，但实际应用中需集成风险指针等机制以避免 use-after-free 错误。内存序参数如 std::memory_order_acquire 和 std::memory_order_release 确保操作的有序性，防止指令重排带来的问题。

另一种思路是基于数组的无锁环形缓冲区，它通过模运算操作下标实现循环访问。优点是内存连续、缓存友好且实现简单，但缺点在于容量固定，适用于生产者-消费者场景。相比之下，链表实现更灵活，但复杂度更高。

5　进阶：更多无锁数据结构掠影

除了无锁队列，无锁栈是理解无锁编程的理想起点，它通过 CAS 原子地更新栈顶指针来实现推送和弹出操作。无锁哈希表通常采用锁分段思想，将哈希桶划分为独立的无锁结构，如无锁链表，以减少竞争。全无锁哈希表实现更为复杂，但能提供更高的并发度。无锁链表是所有无锁结构的基础，支持完整的增删改查操作，但实现难度大，需要处理 ABA 问题和内存回收等挑战。

6　现实考量：何时使用以及如何正确使用

无锁编程并非银弹，它适用于高并发场景，当锁竞争成为主要瓶颈时，无锁结构能显著提升性能。在需要极低延迟和确定性响应的系统，如实时计算或金融交易中，无锁方案尤为关键。然而，在并发度低、业务逻辑复杂或团队经验不足的情况下，无锁实现可能得不偿失，反而引入难以调试的错误。

最佳实践包括优先使用成熟库如 Intel TBB、Boost.Lockfree 或 Java 的 JUC，以减少自行实现的風險。充分测试是必须的，使用线程检查工具如 TSAN 进行压力测试，以捕捉竞态条件。保持代码简单明了，避免过度优化，同时通过代码审查让多人参与，以发现潜在的并发问题。

无锁编程通过原子操作和 CAS 循环替代锁，以换取更好的可伸缩性，但带来了 ABA 问题、内存回收等新挑战。随着硬件发展，如硬件事务内存（HTM）的兴起，并发编程的未来将更加多元化。最终建议是，在性能瓶颈确实存在且由锁引起时，再谨慎考虑无锁方案，理解原理比盲目使用更为重要。

第 II 部

SQLite 索引的优化原理与实践

王思成

Sep 29, 2025

7 从 B-Tree 数据结构到查询计划分析，全面提升你的数据库性能优化能力

在数据库系统中，索引是提升查询性能的核心机制之一。本文将从基础原理出发，深入探讨 SQLite 索引的工作方式、优化策略及实践技巧，帮助您在实际项目中高效应用索引技术。想象一下，在一本没有目录的百科全书中查找一个特定词条，您可能需要逐页翻阅，整个过程既耗时又低效。索引在数据库中扮演着类似的「目录」角色，它通过额外的存储空间和写入开销，换取极高的数据检索效率。随着数据量的增长，全表扫描操作（如 `SELECT * FROM table WHERE ...`）的成本呈线性上升，这会导致数据库性能急剧下降。因此，理解索引的工作原理并正确使用它，成为解决性能瓶颈的关键。本文不仅会讲解如何创建索引，更会深入剖析索引生效的底层原因，并通过实用工具验证优化效果，让您真正掌握 SQLite 索引的精髓。

8 SQLite 索引的核心原理

索引本质上是一种高效查找的数据结构，它存储了数据库表中一列或多列值的副本，并按照特定方式组织，以支持快速数据访问。SQLite 采用 B-Tree 作为索引的基石数据结构，这是因为 B-Tree 非常适合磁盘存储，能够显著减少 I/O 操作次数。与普通二叉树相比，B-Tree 是一种平衡树结构，包含根节点、内部节点和叶子节点，所有叶子节点都位于同一层，这确保了查询效率的稳定性，其时间复杂度为 $O(\log n)$。在 B-Tree 上的查找过程从根节点开始，通过比较键值逐层向下遍历，最终定位到目标叶子节点。

索引的内部存储机制涉及两个关键部分：索引键值和指向数据行的指针。每个索引条目（即叶子节点）包含索引列的值以及对应的 Rowid 或主键。在 SQLite 中，如果表未使用 `WITHOUT ROWID` 选项，则指针为 `ROWID`；否则，指针为主键本身。这种设计使得索引能够快速关联到表中的完整数据行，从而实现高效查询。

9 SQLite 索引的类型与创建

单列索引是最基础的索引形式，通过 `CREATE INDEX idx_name ON table(column);` 命令即可创建。这里，`idx_name` 是用户定义的索引名称，`table` 为目标表名，`column` 为需要索引的列。该命令会在指定列上构建一个 B-Tree 结构，加速基于该列的等值或范围查询。多列复合索引（或称组合索引）则在多个列上创建，例如 `CREATE INDEX idx_name ON table(col1, col2, col3);`。复合索引的核心在于「最左前缀原则」，即查询条件必须包含索引的最左列才能有效利用索引。举例来说，如果存在索引 `(col1, col2, col3)`，那么 `WHERE col1 = ?` 或 `WHERE col1 = ? AND col2 = ?` 能够使用索引，但 `WHERE col2 = ?` 则无法利用索引，因为缺少最左列 `col1`。

唯一索引通过 `CREATE UNIQUE INDEX idx_name ON table(column);` 命令创建，它确保索引列的值唯一，常用于实现数据完整性约束。唯一索引与 `PRIMARY KEY` 或 `UNIQUE` 约束密切相关，后者在底层自动创建唯一索引。

表达式索引（或称函数索引）允许对列应用函数或表达式后建立索引，例如 `CREATE INDEX idx_upper_name ON users(UPPER(name));`。这种索引适用于忽略大小写的查询场景，

但需要注意，查询时必须使用与索引定义完全相同的表达式（如 WHERE UPPER(name) = ?），否则索引无法生效。

部分索引是一种高级优化技术，它只对表中满足特定条件的行建立索引，例如 CREATE INDEX idx_active_users ON users(status) WHERE status = 'active';。部分索引能大幅减小索引尺寸，提升查询和更新性能，特别适用于高频访问的数据子集，如活跃用户或已完成订单。

10 实践：索引优化策略与 EXPLAIN QUERY PLAN

要识别需要索引的查询，可以使用 EXPLAIN QUERY PLAN 命令分析 SQL 语句。例如，执行 EXPLAIN QUERY PLAN SELECT * FROM users WHERE name = 'John'; 会输出查询计划详情。解读结果时，SCAN TABLE 表示全表扫描，通常需要优化；SEARCH TABLE USING INDEX 表示使用了索引扫描，是理想状态；USE TEMP B-TREE 则暗示排序或分组操作可能带来较大开销。

索引应优先为 WHERE 子句和 JOIN 条件创建，因为这些是查询中最常见的过滤操作。此外，索引还能优化排序操作（ORDER BY）。如果索引的键顺序与 ORDER BY 子句一致，数据库可以避免昂贵的排序步骤。例如，复合索引 (col1, col2) 能直接支持 ORDER BY col1, col2 的查询。

覆盖索引是性能优化的高级技巧，它指索引包含了查询所需的所有字段，使得数据库无需回表查询原始数据。例如，如果存在索引 (a, b)，那么查询 SELECT a, b FROM table WHERE a = ? 就是一个覆盖索引查询，能显著减少随机 I/O，提升响应速度。

然而，索引并非没有代价。它需要占用额外磁盘空间，并在每次 INSERT、UPDATE、DELETE 操作时更新，这会降低写入性能。因此，索引的选择性至关重要：高选择性列（如用户 ID 或邮箱）适合建索引，而低选择性列（如性别或状态标志）则收益有限。总之，索引不是越多越好，应平衡读写性能，只为高频查询创建必要索引。

ANALYZE 命令在索引优化中扮演重要角色，它会收集表和索引的统计信息（如选择性），供 SQLite 查询规划器使用。当有多个索引可选时，规划器依赖这些统计信息做出最优决策。建议在大量数据增删后运行 ANALYZE，以确保统计信息的准确性。

SQLite 还支持特殊索引如 AUTOINDEX，它在查询包含 ORDER BY 或 GROUP BY 时自动创建临时索引。如果经常观察到此类现象，应考虑手动创建永久索引以提升性能。

基于以上讨论，我们总结出索引最佳实践清单：首先，基于查询需求创建索引，而非盲目基于表结构；其次，深入理解并应用复合索引的最左前缀原则；第三，尽可能利用覆盖索引避免回表操作；第四，对静态数据定期运行 ANALYZE 以优化查询规划；第五，善用 EXPLAIN QUERY PLAN 验证索引效果；最后，警惕索引维护成本，避免过度索引导致性能下降。

索引的核心价值在于通过空间换时间，利用 B-Tree 数据结构为数据访问建立「高速通道」。掌握索引原理后，结合 EXPLAIN QUERY PLAN 等工具在实践中不断调试，您将能显著提升 SQLite 数据库的性能。总之，精通索引是数据库优化不可或缺的一环，通过本文的讲解，希望您能在海量数据处理中游刃有余，构建出高效稳定的应用系统。

第 III 部

霍夫曼编码（Huffman Coding）算法

黄梓淳

Sep 30,

从信息论基础到手写代码，掌握这一经典无损压缩技术的核心

在数字世界中，数据压缩技术无处不在，从常见的 ZIP 或 RAR 文件到 MP3 音乐和 JPEG 图片，这些格式都依赖于高效的压缩算法来减少存储空间和传输带宽。数据压缩主要分为无损压缩和有损压缩两大类；无损压缩确保原始数据可以完全恢复，而有损压缩则通过牺牲部分信息来换取更高的压缩率。霍夫曼编码作为一种经典的无损压缩算法，由 David A. Huffman 在 1952 年提出，它结合了贪婪算法和最优前缀码的特性，至今仍是许多现代压缩算法如 DEFLATE 的基石。本文的目标不仅是解释霍夫曼编码的原理，还将通过代码实现带领读者从理论走向实践，真正掌握这一技术。

11 预备知识：信息论与编码的基石

定长编码如 ASCII 码为所有字符分配相同长度的编码，这在字符分布不均的文本中效率低下，因为它无法利用高频字符的优势。变长编码的引入解决了这一问题，其核心思想是为出现频率高的字符分配短码，为频率低的字符分配长码，从而降低整体编码长度。前缀码是实现变长编码的关键；它定义为任何一个字符的编码都不是另一个字符编码的前缀，这保证了编码的唯一可译性，解码时无需分隔符也不会产生歧义。举例来说，如果一个编码方案中 A 的编码为 0，B 的编码为 01，那么解码字符串 01 时会出现歧义，因为它可能代表 A 后跟 B 或单独一个 B；而前缀码如 A=0 和 B=10 则能避免这种问题，确保解码过程清晰无误。

12 霍夫曼编码的核心：霍夫曼树的构建

霍夫曼编码的核心在于构建一棵霍夫曼树，这棵树通过贪婪算法逐步合并节点来实现最优编码。首先，我们需要统计待编码数据中每个字符出现的频率，这可以通过遍历数据并记录每个字符的出现次数来完成。接着，将每个字符视为一个只有根节点的二叉树，节点的权值即为该字符的频率，所有这样的树构成一个初始森林。然后，进入循环合并阶段：当森林中树的数目大于一时，每次从中选出两个权值最小的根节点，创建一个新的内部节点，其权值为这两个节点权值之和，并将选出的节点作为新节点的左右孩子（通常约定权值小的在左，权值大的在右，但这不影响压缩效率）。将新树放回森林并移除原来的两棵树，重复此过程直到只剩下一棵树，这棵树就是霍夫曼树。为了可视化这一过程，我们可以以字符串 ABRACADABRA 为例：首先统计频率，A 出现 5 次，B 和 R 各 2 次，C 和 D 各 1 次；然后从森林中合并最小权值节点，例如先合并 C 和 D（权值各为 1），形成权值为 2 的新节点，再与 B 或 R 合并，依此类推，最终构建出完整的霍夫曼树。

13 从霍夫曼树生成编码表

一旦霍夫曼树构建完成，就可以从中生成编码表。从根节点出发，向左子树走标记为 0，向右子树走标记为 1，遍历每个叶子节点（即字符节点）的路径所组成的 0 和 1 序列，即为该字符的霍夫曼编码。例如，如果字符 A 的路径是左-左，那么它的编码就是 00。这种生成方式自动满足了前缀码的性质，因为所有字符都位于叶子节点，任何字符的编码路径都不会成为另一个字符路径的前缀，从而确保了解码的唯一性。通过深度优先遍历霍夫曼树，我们可以递归地记录路径，并在到达叶子节点时将路径存入字典，形成完整的编码表。

14　实战：手把手实现霍夫曼编码

在实战部分，我们将使用 Python 语言来实现霍夫曼编码的关键步骤。首先，设计一个节点类来表示霍夫曼树中的节点，每个节点包含字符、频率以及左右子节点的引用。

```
class Node:
    def __init__(self, char, freq):
        self.char = char # 字符，内部节点可为 None
        self.freq = freq # 频率
        self.left = None # 左孩子
        self.right = None # 右孩子
```

这段代码定义了一个节点类，其中 char 存储字符，对于内部节点，它可以为 None；freq 存储频率；left 和 right 分别指向左右子节点。这为构建霍夫曼树提供了基础数据结构。
接下来，实现核心算法。首先统计输入字符串中每个字符的频率，构建一个字符-频率字典。

```
def build_frequency_dict(data):
    freq_dict = {}
    for char in data:
        freq_dict[char] = freq_dict.get(char, 0) + 1
    return freq_dict
```

这段代码遍历输入字符串 data，使用字典 freq_dict 记录每个字符的出现次数。get 方法用于安全地获取并更新频率，如果字符不存在则初始化为 0。
然后，构建一个优先级队列（使用最小堆）来高效管理节点。我们将所有字符节点放入堆中，按频率排序。

```
import heapq

def build_huffman_tree(freq_dict):
    heap = []
    for char, freq in freq_dict.items():
        heapq.heappush(heap, (freq, Node(char, freq)))
    while len(heap) > 1:
        freq1, node1 = heapq.heappop(heap)
        freq2, node2 = heapq.heappop(heap)
        merged_node = Node(None, freq1 + freq2)
        merged_node.left = node1
        merged_node.right = node2
        heapq.heappush(heap, (merged_node.freq, merged_node))
    return heapq.heappop(heap)[1]
```

这段代码首先将每个字符节点压入最小堆，堆根据频率排序。然后循环合并：每次弹出两个最小频率的节点，创建一个新节点作为它们的父节点，权值为频率之和，并将新节点压回堆

中。循环直到堆中只剩一个节点，即霍夫曼树的根节点。这体现了贪婪算法的思想，总是合并当前最小的两个权重。

生成编码表时，通过深度优先遍历霍夫曼树，递归记录路径。

```
def build_code_table(root):
    code_table = {}
    def traverse(node, code):
        if node is None:
            return
        if node.char is not None:
            code_table[node.char] = code
        traverse(node.left, code + '0')
        traverse(node.right, code + '1')
    traverse(root, '')
    return code_table
```

这段代码定义了一个递归函数 traverse，从根节点开始，向左走添加 0，向右走添加 1。当遇到叶子节点（char 不为 None）时，将当前路径存入 code_table。这确保了每个字符的编码唯一且符合前缀码规则。

最后，编码数据：遍历原始字符串，将每个字符替换为其霍夫曼编码。

```
def encode_data(data, code_table):
    encoded = ''
    for char in data:
        encoded += code_table[char]
    return encoded
```

这段代码简单地将输入字符串中的每个字符映射到编码表中对应的二进制字符串，并拼接成最终的编码结果。

解码过程同样重要，它需要根据霍夫曼树逐位解析编码字符串。

```
def decode_data(encoded, root):
    decoded = ''
    current = root
    for bit in encoded:
        if bit == '0':
            current = current.left
        else:
            current = current.right
        if current.char is not None:
            decoded += current.char
            current = root
    return decoded
```

这段代码从根节点开始，根据编码字符串的每一位（0 或 1）遍历霍夫曼树：遇到 0 走向左

子树，遇到 1 走向右子树。当到达叶子节点时，输出对应字符并重置到根节点，继续解码剩余部分。这确保了编码字符串可以被准确还原为原始数据。

15 分析与展望

霍夫曼编码之所以是最优前缀码，是因为它通过贪婪算法总是合并当前最小的两个权重，从而最小化整体编码长度，这可以从信息论的角度证明，但本文不展开复杂数学推导。霍夫曼编码的优点包括无损压缩、最优前缀码特性以及原理简单易懂；然而，它也有一些缺点，例如需要预先统计整个数据的频率分布，这可能导致两次遍历数据（统计和编码），并且必须将编码表与压缩数据一起存储或传输，对于小文件来说，编码表的开销可能较大。此外，霍夫曼编码对数据变化敏感，如果频率分布发生变化，就需要重新构建树。在实际应用中，霍夫曼编码被广泛用于 GZIP、PKZIP、JPEG 和 MP3 等格式；同时，自适应霍夫曼编码作为变种，可以在编码过程中动态更新频率模型，无需提前统计，提高了灵活性。

回顾霍夫曼编码的核心思想，它通过字符频率决定码长，并利用二叉树生成前缀码，实现了高效的无损压缩。作为计算机科学中的经典算法，霍夫曼编码不仅在理论上具有启发性，还在实际应用中发挥着重要作用。鼓励读者动手实现代码，并尝试将其应用于简单的文件压缩任务，以加深对数据压缩技术的理解。通过本文的讲解和示例，希望读者能够真正掌握霍夫曼编码的精髓，并在实践中灵活运用。

第 IV 部

二分查找（Binary Search）算法

王思成

Oct 01, 2025

在计算机科学中，二分查找是一种高效且基础的搜索算法。它之所以重要，是因为它能在海量数据中快速定位目标。想象一下，如果你需要从一本百万词汇的字典中找到「Algorithm」这个词，逐页翻阅的线性查找方式可能需要数小时，而二分查找则通过不断对半分割搜索区间，在短短几秒内完成。这种效率源于其 O(log n) 的时间复杂度，意味着即使数据规模巨大，查找次数也仅以对数增长。二分查找不仅是面试中的高频考点，更是许多实际系统如数据库索引和路由表的核心组件。本文将从零开始，深入解析二分查找的思想、实现细节和常见陷阱，帮助读者彻底掌握这一算法。

二分查找的重要性体现在其惊人的效率和应用广泛性上。以一个引人入胜的场景为例，当你在有序数组中搜索一个元素时，线性查找需要逐个比较，而二分查找通过每次将搜索区间减半，迅速缩小范围。例如，在百万级数据中，线性查找可能需要百万次比较，而二分查找仅需约二十次。这种效率提升在现实世界中至关重要，例如在大型数据库或游戏逻辑中快速检索数据。本文的目标不仅是教会读者写对二分查找代码，更要深入理解其每个细节，包括循环条件、中间值计算和边界更新等经典难题，从而在实际应用中游刃有余。

16　算法思想与前提条件

二分查找的核心思想是分而治之，它将有序的搜索区间不断对半分割，通过比较中间元素与目标值，逐步缩小范围直至找到目标或确认不存在。然而，二分查找并非万能，它依赖于两个必要前提：数据集必须有序，无论是升序还是降序排列；同时，数据结构必须支持随机访问，例如数组可以在常数时间内通过索引访问任意元素，而链表则不适合标准二分查找。基本步骤包括确定初始搜索区间、计算中间位置、比较中间元素与目标值，并根据比较结果调整边界。具体来说，对于升序数组，首先设置左右指针为数组的起始和结束索引，然后循环计算中间索引，如果中间元素等于目标值则返回索引；如果中间元素小于目标值，则调整左指针到中间索引加一的位置；如果中间元素大于目标值，则调整右指针到中间索引减一的位置。重复这一过程直到找到目标或区间无效。

17　图解算法流程

由于本文避免使用图片，我们将通过文字详细描述二分查找的流程。假设有一个升序数组 [1, 3, 5, 7, 9, 11]，目标是查找数字 7。初始时，左指针指向索引 0，右指针指向索引 5，搜索区间为 [0, 5]。第一步计算中间索引，使用公式 mid = left + (right - left) // 2，得到 mid = 2，对应元素 5。比较 5 和 7，由于 5 小于 7，目标在右侧，因此调整左指针为 mid + 1 = 3。新区间为 [3, 5]，中间索引 mid = 4，对应元素 9。比较 9 和 7，由于 9 大于 7，目标在左侧，调整右指针为 mid - 1 = 3。新区间为 [3, 3]，中间索引 mid = 3，对应元素 7，与目标相等，返回索引 3。如果目标不存在，例如查找数字 4，过程类似，但最终左指针超过右指针，返回 -1 表示未找到。

18　关键实现细节与“坑点”剖析

二分查找的实现看似简单，但细节决定成败。首先，循环条件的选择至关重要。常见的有两种：while (left <= right) 和 while (left < right)。前者表示搜索区间为闭区间 [left, right]，当 left 等于 right 时，区间仍有一个元素需要检查，这是最推荐的方式，因为它不

易出错。后者表示左闭右开区间 [left, right)，当 left 等于 right 时区间为空，需要更细致的边界处理，容易导致遗漏或错误。其次，中间值计算时，直接使用 mid = (left + right) / 2 可能存在整数溢出风险，当 left 和 right 都很大时，它们的和可能超出整型最大值。解决方案是使用 mid = left + (right - left) // 2 或位运算 mid = (left + right) >> 1（在 Java 等语言中），这些方法避免了溢出问题。最后，边界更新必须排除已检查的元素，即 left = mid + 1 和 right = mid - 1，这是因为 arr[mid] 已经被比较过，如果不排除，可能导致死循环，尤其是在区间缩小到单个元素时。

19 标准实现代码（迭代版本）

以下是二分查找的迭代版本 Python 代码，我们将逐段解读其关键部分。

```
def binary_search(arr, target):
    """
    在升序数组 arr 中查找 target。
    找到则返回其索引，否则返回-1。
    """
    left, right = 0, len(arr) - 1 # 初始化搜索区间为闭区间 [0, n-1]

    while left <= right: # 当区间不为空时循环
        mid = left + (right - left) // 2 # 防止溢出
        # mid = (left + right) // 2 # 在 Python 中通常不会溢出，但好习惯是
            ↪ 使用上面的写法

        if arr[mid] == target:
            return mid # 找到目标，返回索引
        elif arr[mid] < target:
            left = mid + 1 # 目标在右侧，调整左边界
        else: # arr[mid] > target
            right = mid - 1 # 目标在左侧，调整右边界

    return -1 # 未找到目标
```

在这段代码中，首先初始化左指针 left 为 0，右指针 right 为数组长度减一，这定义了初始搜索区间为闭区间 [0, n-1]。循环条件使用 left <= right，确保在区间内所有元素都被检查。中间索引 mid 的计算采用 left + (right - left) // 2，这是一种防止整数溢出的安全方法，尽管在 Python 中整数不会溢出，但养成这种习惯有助于跨语言应用。在循环体内，通过比较 arr[mid] 与 target 决定下一步操作：如果相等，直接返回 mid；如果 arr[mid] 小于 target，说明目标在右侧，因此将 left 更新为 mid + 1，排除已检查的中间元素；如果 arr[mid] 大于 target，则将 right 更新为 mid - 1。如果循环结束仍未找到目标，返回 -1。

20 其他实现方式

除了迭代版本，二分查找还可以通过递归实现。递归版本将搜索过程分解为子问题，每次递归调用处理一半的区间。例如，在递归函数中，基础情况是区间为空或找到目标，否则根据中间值比较结果递归调用左半部分或右半部分。递归实现的空间复杂度为 O(log n)，因为递归调用栈的深度与数据规模的对数成正比，而迭代版本的空间复杂度为 O(1)，只使用固定变量。另一种常见实现是使用左闭右开区间 [left, right)，在这种方式下，初始右指针为数组长度，循环条件为 left < right，边界更新时 right 直接设为 mid，而不是 mid - 1。这种方式需要更仔细的初始化，但有时在特定场景下更简洁。
递归版本代码示例如下：

```
def binary_search_recursive(arr, target, left, right):
    if left > right:
        return -1
    mid = left + (right - left) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target, mid + 1, right)
    else:
        return binary_search_recursive(arr, target, left, mid - 1)
```

这里，递归函数接受数组、目标值和当前区间左右指针作为参数。如果左指针超过右指针，返回 -1 表示未找到；否则计算中间索引，比较后递归调用左或右半部分。这种实现虽然直观，但需要注意递归深度可能导致的栈溢出问题。

21 复杂度分析

二分查找的时间复杂度为 O(log n)，其中 n 是数据规模。这是因为每次循环或递归调用都将问题规模减半，最坏情况下需要执行 log ☒ (n) 次操作。例如，当 n=100 万时，log ☒ (n) 约等于 20，而线性查找可能需要 100 万次比较，凸显了二分查找的高效性。空间复杂度方面，迭代版本为 O(1)，因为它只使用常数级别的额外空间；递归版本为 O(log n)，由于递归调用栈的深度与 log n 成正比。用 LaTeX 公式表示，时间复杂度为 (O(\log n))，空间复杂度迭代版本为 (O(1))，递归版本为 (O(\log n))。

22 常见变体与应用场景

二分查找有多种变体，适用于不同场景。例如，在重复元素数组中寻找第一个等于目标的元素，需要调整边界更新策略，在找到目标后继续向左搜索；寻找最后一个等于目标的元素则向右搜索。另一种变体是寻找第一个大于等于目标的元素，常用于插入位置确定，例如在排序列表中插入新值。这些变体在数据库索引和范围查询中广泛应用，通过微调比较和边界逻辑，可以解决更复杂的问题。每个变体的核心在于理解搜索区间的定义和边界排除原则，确

保算法正确性。

二分查找的核心思想是通过分治策略在有序数据中高效搜索，其前提是数据有序且支持随机访问。实现时需注意循环条件推荐使用闭区间、中间值计算防溢出、边界更新排除已检查元素。关键口诀包括区间定义清晰、中间计算安全、边界更新彻底。通过实践，读者可以在力扣等平台练习相关题目，如二分查找和搜索插入位置，以巩固理解。掌握这些细节后，二分查找将不再令人畏惧，而是成为解决实际问题的有力工具。

23 思考题

为了进一步加深理解，请思考以下问题：如果数组是降序排列，代码需要如何修改？二分查找是否可以用在链表上，如果能，时间复杂度是多少？除了搜索，二分思想还能解决哪些问题，例如求平方根或在旋转数组中搜索？这些问题鼓励读者探索算法的灵活性和应用边界。

第 V 部

信号量（Semaphore）机制

杨岢瑞

Oct 02, 2025

在并发编程的世界中，多个线程或进程同时访问共享资源时，常常会引发数据不一致或资源冲突的问题。想象一个电影院售票场景：如果多个顾客同时尝试购买同一场次的最后一张票，而没有协调机制，可能会导致超售或数据错误。这种问题被称为竞态条件，其核心在于对有限资源的无序竞争。为了解决这类问题，计算机科学家 Edsger Dijkstra 提出了信号量机制，它作为一种经典的同步工具，能够有效管理资源分配并确保线程安全。本文将带领读者从理论概念入手，逐步深入，最终亲手实现一个基本的信号量，并将其应用于实际场景中，从而夯实并发编程的基础。

24 信号量是什么?—— 理论与概念

信号量的核心思想基于一个非负整数计数器，该计数器表示可用资源的数量。它通过两种原子操作来管理资源访问：P 操作（也称为 wait 或 acquire）和 V 操作（也称为 signal 或 release）。P 操作用于尝试获取资源，如果计数器值大于零，则将其减一并允许线程继续执行；如果计数器值为零，则线程被阻塞，直到其他线程执行 V 操作释放资源。V 操作则负责释放资源，将计数器值加一，并唤醒等待队列中的某个线程。这两种操作必须保证原子性，即执行过程中不可被中断，否则会导致同步失效。

信号量主要分为两种类型：二进制信号量和计数信号量。二进制信号量的计数器值只能为零或一，常用于实现互斥锁，确保同一时刻只有一个线程可以进入临界区。例如，在保护共享变量时，二进制信号量可以充当门卫角色。计数信号量的计数器值可以是任意非负整数，适用于控制多实例资源的访问，比如数据库连接池中有十个连接，则计数信号量初始值设为十，允许最多十个线程同时使用。信号量的行为规范还包括使用等待队列来管理被阻塞的线程，确保公平性和效率。原子性是信号量正确工作的基石，它依赖于底层硬件或操作系统的支持，防止在多线程环境下出现数据竞争。

25 从零开始实现一个信号量

为了深入理解信号量的内部机制，我们将使用 Java 语言实现一个简单的信号量类。首先，设计类结构，包括成员变量和方法。成员变量包括一个整型值 `value` 表示当前计数，以及一个等待队列 `waitQueue` 用于存储阻塞线程。方法包括构造函数 `Semaphore(int initialValue)`、`wait()` 方法实现 P 操作，以及 `signal()` 方法实现 V 操作。在实现中，我们将使用 `synchronized` 关键字来确保操作的原子性。

以下是 `wait()` 方法的实现代码：

```
public synchronized void wait() {
   value--;
   if (value < 0) {
      try {
         this.wait();
      } catch (InterruptedException e) {
         Thread.currentThread().interrupt();
      }
   }
}
```

在这段代码中，synchronized 关键字确保了对 value 的修改和判断操作是原子的，防止多线程同时执行导致的竞态条件。当线程调用 wait() 方法时，首先将 value 减一，然后检查是否小于零。如果 value 小于零，表示资源不足，当前线程通过 this.wait() 进入等待状态，直到被其他线程唤醒。异常处理部分负责处理线程中断情况，确保程序的健壮性。这里使用 notify() 而非 notifyAll()，是因为通常只需要唤醒一个等待线程，避免不必要的性能开销。

接下来是 signal() 方法的实现代码：

```
public synchronized void signal() {
    value++;
    if (value <= 0) {
        this.notify();
    }
}
```

在 signal() 方法中，同样使用 synchronized 保证原子性。首先将 value 加一，然后检查是否小于等于零。如果条件成立，说明有线程在等待队列中，此时调用 notify() 唤醒其中一个线程。这种实现方式简单高效，但需要注意的是，在复杂场景下，可能需要更精细的队列管理来避免饥饿问题。这种基于单个锁的实现虽然易于理解，但在高并发环境下可能存在性能瓶颈；实际操作系统中的信号量通常依赖硬件原子指令和调度器优化。

26　实战应用 —— 用我们实现的信号量解决经典问题

信号量在实际应用中非常广泛，我们以两个经典场景为例：实现互斥锁和解决生产者-消费者问题。首先，在实现互斥锁时，可以使用一个初始值为一的二进制信号量。线程在进入临界区前调用 wait() 方法获取锁，退出时调用 signal() 方法释放锁。例如，创建一个信号量实例 Semaphore mutex = new Semaphore(1);，线程代码中在临界区前后分别执行 mutex.wait() 和 mutex.signal()，确保同一时间只有一个线程访问共享资源。

另一个经典应用是生产者-消费者问题，其中多个生产者和消费者共享一个有限大小的缓冲区。我们需要三个信号量：emptySlots 表示空槽位数量，初始值为缓冲区大小 N；fullSlots 表示已填充槽位数量，初始值为零；mutex 作为二进制信号量，保护缓冲区的互斥访问。生产者线程在生产物品后，先等待 emptySlots 信号量确保有空位，然后获取 mutex 锁，将物品放入缓冲区，最后释放 mutex 并通知 fullSlots。消费者线程则相反，先等待 fullSlots 信号量确保有物品，然后获取 mutex 锁，取出物品后释放 mutex 并通知 emptySlots。

以下是生产者线程的代码示例：

```
void producer() {
    while (true) {
        Item item = produceItem();
        emptySlots.wait();
        mutex.wait();
```

```
        buffer.add(item);
        mutex.signal();
        fullSlots.signal();
    }
}
```

在这段代码中，emptySlots.wait() 确保缓冲区有空位时才生产，防止溢出；mutex.wait() 和 mutex.signal() 保护对缓冲区的互斥访问，避免数据竞争；fullSlots.signal() 通知消费者有新物品可用。消费者线程的代码类似，通过等待 fullSlots 和 emptySlots 信号量协调生产与消费的节奏。这种设计确保了缓冲区在满时不会生产，空时不会消费，同时保证了线程安全。

27 信号量的陷阱与现代替代方案

尽管信号量是强大的同步工具，但在使用过程中容易陷入一些陷阱。死锁是常见问题，例如当多个线程以不同顺序获取信号量时，可能导致互相等待而无法继续执行。优先级反转则发生在高优先级线程等待低优先级线程持有的信号量时，造成系统响应延迟。此外，编程错误可能导致遗忘唤醒，即本该唤醒的线程未被正确处理，进而引发线程饥饿。

在现代并发编程中，信号量有更成熟的实现和替代方案。例如，Java 标准库提供了 java.util.concurrent.Semaphore 类，它支持公平性和非公平性策略，并集成了更高级的功能。与互斥锁相比，信号量更通用，不仅用于互斥，还能用于同步；而互斥锁严格限制于临界区保护。条件变量则常与互斥锁配合使用，提供更灵活的等待和通知机制，但信号量通过计数器自带状态管理，更适用于资源计数场景。开发者应根据具体需求选择合适工具，并注意测试和调试以避免潜在问题。

信号量作为并发编程的基石，通过计数器模型和原子操作，有效解决了资源同步与互斥问题。从理论到实践，我们不仅理解了其核心概念，还亲手实现了简单的信号量类，并应用于互斥锁和生产者-消费者等经典场景。这种深入的学习方式有助于巩固操作系统和并发编程原理的知识。在实际项目中，建议使用标准库提供的信号量实现，同时注意规避死锁和优先级反转等陷阱。通过不断实践和优化，读者可以更好地掌握并发编程技能，构建高效可靠的系统。