深入理解并实现基本的跳表（Skip List）数据结构

叶家炜

May 13, 2025

—— 原理、实现与性能分析

在计算机科学中，数据结构的选择往往需要在时间与空间效率之间进行权衡。传统链表虽然插入和删除操作高效，但查询需要 O(n) 的时间复杂度；而平衡二叉搜索树虽能实现 O(\log n) 的查询效率，却需要复杂的旋转操作维持平衡。跳表（Skip List）作为一种概率型数据结构，通过多层索引机制实现了接近 O(\log n) 的查询性能，同时保持了实现的简洁性。

Redis 的有序集合（ZSET）和 LevelDB 的 MemTable 均采用跳表作为核心数据结构，这得益于其动态扩展性和高效的并发支持潜力。本文将深入解析跳表的原理，通过 Python 代码实现一个基础版本，并分析其性能特点。

1 一、跳表的基础知识

跳表的本质是多层链表的叠加。最底层为原始链表，存储所有数据节点；上层链表则作为索引层，通过跳跃式遍历加速查询。每个节点的层数由随机过程决定，高层节点稀疏分布，低层节点密集分布。

头节点（Head）作为各层链表的起点，不存储实际数据，仅提供遍历入口。尾节点（Tail）通常为 `None`，标识链表的结束。这种设计使得跳表的查询过程可以从高层快速缩小范围，逐步下沉到底层定位目标。

跳表的核心思想在于空间换时间。通过为部分节点建立多层索引，将单次查询的路径长度从 O(n) 缩减到 O(\log n)。随机层数生成策略（如“抛硬币”机制）避免了手动平衡的开销，使得插入操作的时间复杂度稳定在平均 O(\log n)。

2 二、跳表的核心操作原理

2.1 查询操作

查询操作的逻辑可概括为“从高层向底层逐级下沉”。以查找值 `target` 为例：

- 从最高层头节点出发，向右遍历直至当前节点的后继节点值大于 `target`。
- 下沉到下一层，重复上述过程直至到达底层。
- 最终检查底层节点的值是否等于 `target`。

这一过程的时间复杂度为 O(\log n)，因为每层索引的步长呈指数级增长。

2.2 插入操作

插入操作需完成三个关键步骤：

- 定位插入位置：类似查询过程，记录每层中最后一个小于待插入值的节点（称为前置节点）。
- 生成随机层数：通过随机函数决定新节点的层数，通常采用概率 p=0.5，使得第 i 层的节点数量约为第 i-1 层的一半。
- 更新指针：将新节点的各层指针指向对应前置节点的后继节点，并更新前置节点的指针。

随机层数的生成确保了索引分布的均匀性，避免手动维护平衡。

2.3 删除操作

删除操作首先定位待删除节点，随后逐层更新其前置节点的指针，跳过该节点。时间复杂度与插入操作相同，均为平均 O(\log n)。

3 三、跳表的代码实现（以 Python 为例）

3.1 数据结构定义

```
import random

class Node:
    def __init__(self, value, level):
        self.value = value
        self.forward = [None] * (level + 1) # 各层的前向指针

class SkipList:
    def __init__(self, max_level=16, p=0.5):
        self.max_level = max_level
        self.p = p
        self.head = Node(-float('inf'), max_level) # 头节点初始化为最小值
        self.current_level = 0 # 当前有效层数
```

Node 类的 forward 数组存储该节点在各层的后继指针。SkipList 类的 max_level 限制最大层数以防止内存过度消耗，p 控制层数生成概率。

3.2 随机层数生成

```
def random_level(self):
    level = 0
```

```
    while random.random() < self.p and level < self.max_level:
        level += 1
    return level
```

此方法通过循环抛“硬币”（随机数小于 p 的概率）决定层数。例如，当 p=0.5 时，生成第 i 层的概率为 1/2^i。

3.3 插入方法实现

```
def insert(self, value):
    update = [None] * (self.max_level + 1) # 记录各层的前置节点
    current = self.head

    # 从最高层开始查找插入位置
    for i in range(self.current_level, -1, -1):
        while current.forward[i] and current.forward[i].value < value:
            current = current.forward[i]
        update[i] = current

    # 生成新节点层数
    new_level = self.random_level()
    if new_level > self.current_level:
        for i in range(self.current_level + 1, new_level + 1):
            update[i] = self.head
        self.current_level = new_level

    # 创建新节点并更新指针
    new_node = Node(value, new_level)
    for i in range(new_level + 1):
        new_node.forward[i] = update[i].forward[i]
        update[i].forward[i] = new_node
```

update 数组保存了每层中最后一个小于待插入值的节点。插入新节点时，需从底层到新节点的最高层更新这些节点的指针。

3.4 查询方法实现

```
def search(self, value):
    current = self.head
    for i in range(self.current_level, -1, -1):
        while current.forward[i] and current.forward[i].value <= value:
            current = current.forward[i]
```

```
    return current.value == value
```

查询过程从最高层逐步下沉，最终在底层确认是否存在目标值。

4 四、跳表的性能分析

4.1 时间复杂度

跳表的查询、插入和删除操作的平均时间复杂度均为 O(\log n)。其证明依赖于概率论：假设每层索引的节点数以概率 p 递减，则遍历的层数约为 \log_{1/p} n。当 p=0.5 时，层数期望为 2，时间复杂度接近 O(\log n)。最坏情况下（所有节点集中在同一层），时间复杂度退化为 O(n)，但这种情况的概率极低。

4.2 空间复杂度

跳表的额外空间开销主要来自索引层。理论上，索引节点总数约为 n/(1-p)。当 p=0.5 时，空间复杂度为 O(n)，相比原始链表多消耗一倍内存。

4.3 与平衡树的对比

跳表在并发环境下更具优势，因为其插入和删除操作只需局部调整指针，无需全局锁。而红黑树等平衡树需要复杂的旋转操作，难以高效实现并发控制。

5 五、跳表的实际应用与优化

5.1 经典应用场景

Redis 使用跳表实现有序集合（ZSET），支持 O(\log n) 的成员查询和范围查询。LevelDB 的 MemTable 同样采用跳表，其内存中的有序键值存储依赖跳表的高效插入与查询。

5.2 优化方向

- 动态调整最大层数：根据数据规模自适应调整 `max_level`，避免内存浪费。
- 概率参数调优：p 值的选择影响时间与空间效率。p 越小，层数越高，查询越快，但空间消耗越大。
- 并发控制：通过无锁编程（如 CAS 操作）实现线程安全的跳表。

跳表以简单的实现获得了接近平衡树的性能，成为许多高性能系统的首选数据结构。其缺点在于空间开销和理论上的最坏情况，但在实际应用中，随机化设计使得最坏情况几乎不可能出现。
对于需要频繁插入、删除和范围查询的场景（如实时排行榜、数据库索引），跳表是一个理想的选择。进一步学习可参考 William Pugh 的原始论文《Skip Lists: A Probabilistic Alternative to Balanced Trees》，其中详细推导了跳表的数学性质。