

# LevelDB 中的跳表实现

抽奖



## 何为跳表

跳跃表 (skiplist)，简称「跳表」。是一种在链表基础上进行优化的数据结构，最早由 William Pugh 在论文《[Skip Lists: A Probabilistic Alternative to Balanced Trees](#)》中提出。

William Pugh 于 1989 在论文中将「跳表」定位为：一种能够替代**平衡树**的数据结构，比起使用**强制平衡算法**的各种平衡树，跳表采用一种**随机平衡**的策略。因此跳表拥有更为简单的算法实现，同时拥有与平衡树相媲美的时间复杂度( $\log n$  级别)和操作效率。

## 设计思想

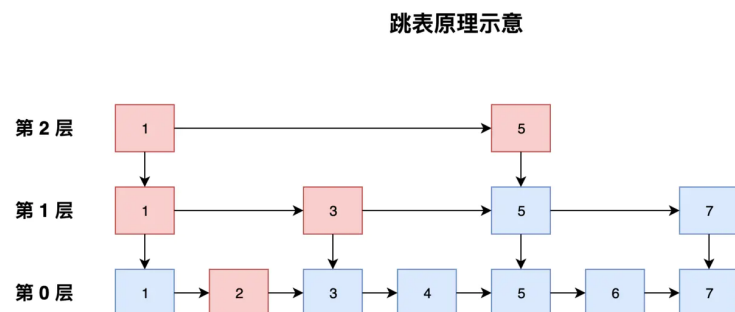
普通链表是一种**顺序查找**的数据结构。即在查找某个元素时需要依次遍历所有元素进行比较。且在元素有序的情况下也无法像数组那样利用**二分查找**，固元素查询时间复杂度为  $O(n)$ ，若需维护链表有序，元素插入的时间复杂度也同样需要  $O(n)$ 。

在一些数据量极大的场景下， $O(n)$  的时间复杂度仍然存在优化的空间。

**平衡二叉查找树** AVL 或者**红黑树**是经常被用来实现上述优化的数据结构。但这些平衡树需要在整个过程中保持树的平衡，这带来了一定的复杂度。

而跳表的核心思想类似于对有序链表中元素建立**索引**。整个跳表按照层次进行构建，底层为原始的有序链表，然后抽取链表中的关键元素到上一层作为索引，从刚构建的索引中可以继续抽取关键元素到新的一层，以此类推。

跳表原理结构如下图所示：



normal\_skip\_list.png

以查找元素 2 为例，查找将从第 2 层索引确定 1 ~ 5 范围，再到第 1 层索引进一步确定 1 ~ 3 范围，最后回到底层原始链表查找到元素 2。

## 性能分析

上文提到了「抽取每一层的关键元素到上一层作为索引」，但是随着数据量的增加每一层的结点也会越来越多，该如何选取结点让跳表的结点呈现我们所需的分布？

跳表采用「**投硬币**」的方式决定结点是否向上提升。

假设现在底层有序链表有  $n$  个结点且投硬币概率为 50%，则第一层**应该**<sup>[1]</sup>有  $n/2$  个索引结点，第二层有  $n/4$  个索引结点，第  $i$  层索引有  $n/2^i$  个结点，当  $n/2^i$  等于 2 时，意味着已经到了**最高层**。此时由  $n/2^i = 2$ ，可推导出  $i = \log_2 n$ 。

即投硬币概率为 50% 时，跳表层高为  $\log_2 n$ ，且由于每两个结点就有一个结点上升为一个索引结点。所以当从最上层向下搜索的过程中，每一个最多只会比较 3 个结点（常数级），所以整个搜索过程时间复杂度最终为  $\log_2 n$ 。

[1] 概率事件，并非一定具有准确的  $n/2$  结点。

将上述过程进一步扩展概率为  $p$ ，则时间复杂度为  $\log_{1/p} n$ 。其中每一层比较的次数不超过  $1/p$ 。

上述是为了方便理解而简化的概率推导过程，结论也建立在  $n$  足够大的前提下。实际推导过程要复杂很多，有兴趣的读者可以阅读论文原文：[《Skip Lists: A Probabilistic Alternative to Balanced Trees》](#)

## 实现

上文介绍了跳表的基本思想，其中为了方便理解和讲述，我们将索引结点单独绘制成一个结点。如果完全按照上文图示实现跳表，则跳表需要额外  $n$  个结点空间。但在实际实现时，无需额外结点只需使用指针指向相应结点即可，因此只是多出了  $n$  个指针而已。

即跳表实际实现的结构如下图所示：

skip\_list\_wiki.png

其中黄色格子为数据结点，白色格子为数据结点内的指针。

## LevelDB 中的跳表源码解析

我们以 LevelDB 中的跳表实现 [skiplist.h](#) 为例，分析跳表的具体实现

设计结点结构如下：

```
template <typename Key, class Comparator>
struct SkipList<Key, Comparator>::Node {
    explicit Node(const Key& k) : key(k) {}
    // 存储 key
    Key const key;

    // .....

private:
    // 下标表示结点的层次 level
    // 整个数组表示该结点在各层次的存储情况
    std::atomic<Node*> next_[1];
}
```

如下图所示：

data\_structure\_0.png

进一步理解如下图所示：

data\_structure\_1.png

其中上图 head\_ 内的 next\_ 数组存储着**指向各个索引层次第一个元素的指针**。

其它每个结点（如图中的结点 1）中的 next\_ 数组包含了如下信息：

### 结点在各个索引层中的下一个结点的指针

查询元素

元素查询主要逻辑集中在 FindGreaterOrEqual 这个函数，就以这个函数为例，体现元素查询过程：

```
// 搜索大于等于 key 的所有结点
template <typename Key, class Comparator>
typename SkipList<Key, Comparator>::Node*
SkipList<Key, Comparator>::FindGreaterOrEqual(const Key& key,
                                                Node** prev) const {
    Node* x = head_;
    // 获取当前结点的层高
    // 从最上层的索引层开始遍历
    int level = GetMaxHeight() - 1;
    while (true) {
        // 假设 next_ = [*3, *5, *6]
        // 表示该结点：
        // 在第 2 层的下一个索引结点为 6
        // 在第 1 层的下一个索引结点为 5
        // 在第 0 层的下一个结点为 3
        // 那么就可以直接通过 next_[level] 找到下一个索引结点
```

```

Node* next = x->Next(level);
if (KeyIsAfterNode(key, next)) { // key 是否在当前结点之后 (大小关系由比较器最
    // Keep searching in this list
    // 继续遍历搜索该层的剩余结点
    x = next;
} else { // key 是否在当前结点之后 (大小关系由比较器最终确认)
    // 记录结点到 prev 数组
    // prev 数组记录每个索引层次要插入 key 的位置
    if (prev != nullptr) prev[level] = x; prev
    if (level == 0) { // 遍历到 0 层, 遍历结束
        return next;
    } else {
        // Switch to next list
        // 进入下一层遍历
        level--;
    }
}
}
}
}

```

## 删除元素

LevelDB 业务层面无删除结点的需求, 见源码注解如下:

```

// (1) Allocated nodes are never deleted until the SkipList is
// destroyed. This is trivially guaranteed by the code since we
// never delete any skip list nodes.

```

## 插入元素

```

template <typename Key, class Comparator>
void SkipList<Key, Comparator>::Insert(const Key& key) {
    // TODO(opt): We can use a barrier-free variant of FindGreaterOrEqual()
    // here since Insert() is externally synchronized.
    Node* prev[kMaxHeight];
    // 获取所有大于等于 (比较器定义) key 的结点
    // prev 保存各个索引层要插入的前一个结点
    Node* x = FindGreaterOrEqual(key, prev);

    // Our data structure does not allow duplicate insertion
    // 不允许插入重复的元素
    // 那么为空, 表示没有 >= key 的结点。要么不等于列表中的所有 key, 表示没有重复元素
    assert(x == nullptr || !Equal(key, x->key));

    // 生成一个随机高度
    int height = RandomHeight();
    // 如果随机高度比当前最大高度大
    if (height > GetMaxHeight()) {
        // prev 下标从原先的最大 height 到最新的最大 height 之间初始化为 head_
        for (int i = GetMaxHeight(); i < height; i++) {

```

```

    prev[i] = head_;
}
// It is ok to mutate max_height_ without any synchronization
// with concurrent readers. A concurrent reader that observes
// the new value of max_height_ will see either the old value of
// new level pointers from head_ (nullptr), or a new value set in
// the loop below. In the former case the reader will
// immediately drop to the next level since nullptr sorts after all
// keys. In the latter case the reader will use the new node.
// 原子操作：保存最新的最大高度
max_height_.store(height, std::memory_order_relaxed);
}

// 创建一个新结点
x = NewNode(key, height);
for (int i = 0; i < height; i++) {
    // NoBarrier_SetNext() suffices since we will add a barrier when
    // we publish a pointer to "x" in prev[i].
    //
    // 插入新结点，即：
    // new_node->next = pre->next;
    // pre->next = new_node;
    x->NoBarrier_SetNext(i, prev[i]->NoBarrier_Next(i));
    prev[i]->SetNext(i, x);
}
}

```

## 并发处理

LevelDB 的跳表实现支持**单线程写、多线程读**，为了满足该特点，LevelDB 在更新和读取时需要注意 C++ `memory_order` 的设置。

在讲解 LevelDB 跳表中的 `memory_order` 之前需要先介绍相关的基础知识。

### 原子性

原子寓意着「**不可再分的最小单位**」，固计算机领域提及的**原子性操作**指的是那些「不可或不该再被切分（或中断）的操作」。

而关于原子性，我们应当具有一个基本的认知：**高级语言层面，单条语句并不能保证对应的操作具有原子性。**

在使用 C、C++、Java 等各种高级语言编写代码时，不少人会下意识的认为一条不可再分的单条语句具有原子性，例如常见 `i++`。

// 伪码

```
int i = 0;
```

```

void increase() {
    i++;
}

int main() {
    /* 创建两个线程，每个线程循环进行 100 次 increase */
    // 线程 1
    Thread thread1 = new Thread(
        run() {
            for (int i = 0; i < 100; i++) increase();
        }
    );
    // 线程 2
    Thread thread2 = new Thread(
        run() {
            for (int i = 0; i < 100; i++) increase();
        }
    );
}

```

如果 `i++` 是原子操作，则上述伪码中的 `i` 最终结果为 200。但实际上每次运行结果可能都不相同，且通常小于 200。

之所以出现这样的情况是因为 `i++` 在执行时通常还会继续划分为多条 CPU 指令。以 Java 为例，`i++` 编译将形成四条字节码指令，如下所示：

```

// Java 字节码指令
0: getstatic
1: iconst_1
2: iadd
3: putstatic

```

而上述四条指令的执行并不保证原子性，即执行过程可被打断。考虑如下 CPU 执行序列：

1. 线程 1 执行 `getstatic` 指令，获得 `i = 1`
2. CPU 切换到线程 2，也执行了 `getstatic` 指令，获得 `i = 1`。
3. CPU 切回线程 1 执行剩下指令，此时 `i = 2`
4. CPU 切到线程 2，由于步骤 2 读到的是 `i = 1`，固执行剩下指令最终只会得到 `i = 2`

以上四条指令是 Java 虚拟机中的字节码指令，字节码指令是 JVM 执行的指令。实际每条字节码指令还可以继续划分为更底层的机器指令。但字节码指令已经足够演示原子性的含义了

如果对底层 CPU 层面如何实现机器指令的原子操作<sup>[1]</sup>感兴趣，可查阅 [Spinlock](#)、[MESI protocol](#) 等资料。

[1] 一条 CPU 指令可能需要涉及到缓存、内存等多个单元的交互，而在多核 CPU 的场景下并会存在与高层次多线程类似的问题。固需要一些机制和策略才可实现机器指令的原子操作。

## 有序性

上述已经提到 CPU 的一条指令执行时，通常会有多个步骤，如取指 IF 即从主存储器中取出指令、ID 译码即翻译指令、EX 执行指令、存储器访问 MEM 取数、WB 写回。

即指令执行将经历：IF、ID、EX、MEM、WB 阶段。

现在考虑 CPU 在执行一条又一条指令时该如何完成上述步骤？最容易想到并是顺序串行，指令 1 依次完成上述五个步骤，完成之后，指令 2 再开始依次完成上述步骤。这种方式简单直接，但执行效率显然存在很大的优化空间。

思考一种流水线工作：

```
指令1  IF ID EX MEM WB
指令2      IF ID EX MEM WB
指令3          IF ID EX MEM WB
```

采用这种流水线的工作方式，将避免 CPU 、存储器中各个器件的空闲，从而充分利用每个器件，提升性能。

同时注意到由于每条指令执行的情况有所不同，指令执行的先后顺序将会影响到这条流水线的负载情况，而我们的目标则是让整个流水线满载紧凑的运行。

为此 CPU 又实现了「指令重排」技术，CPU 将有选择性的对部分指令进行重排来提高 CPU 执行的性能和效率。例如：

```
x = 100;    // #1
y = 200;    // #2
z = x + y;  // #3
```

虽然上述高级语言的语句会编译成多条机器指令，多条机器指令还会进行「指令重排」，#1 语句与 #2 语句完全有可能被 CPU 重新排序，所以程序实际运行时可能会先执行 `y = 200;` 然后再执行 `x = 100;`；

但另一方面，指令重排的前提是不会影响线程内程序的串行语义，CPU 在重排指令时必须保证线程内语义不变，例如：

```
x = 0; // #1
x = 1; // #2
y = x; // #3
```

上述的 `y` 一定会按照正常的串行逻辑被赋值为 1。

但不幸的是，CPU 只能保证线程内的串行语义。在多线程的视角下，「指令重排」造成的影响需要程序员自己关注。

```
// 公共资源
int x = 0;
int y = 0;
int z = 0;
```

```
Thread_1:          Thread_2:
x = 100;           while (y != 200);
y = 200;           print x
z = x + y;
```

如果 CPU 不进行「乱序优化」执行，那么  $y = 200$  时， $x$  已经被赋值为 100，此时线程 2 输出  $x = 200$ 。

但实际运行时，线程 1 可能先执行  $y = 200$ ，此时  $x$  还是初始值 0。线程 2 观察到  $y = 200$  后，退出循环，输出  $x = 0$ ；

C++ 中的 *atomic* 和 *memory\_order*

C++ 提供了 `std::atomic` 类模板，以保证操作原子性。同时也提供了内存顺序模型 *memory\_order* 指定内存访问，以便提供有序性和可见性。

其中 *memory\_order* 共有六种，如下表所示：

| <i>memory_order</i>         | 解释  |
|-----------------------------|---|
| <i>memory_order_relaxed</i> | 只保证原子操作的原子性，不提供有序性的保证   |
| <i>memory_order_consume</i> | 当前线程中依赖于当前加载的该值的读或写不能被重排到此加载前   |
| <i>memory_order_acquire</i> | 在其影响的内存位置进行获得操作：当前线程中读或写不能被重排到此加载前  |
| <i>memory_order_release</i> | 当前线程中的读或写不能被重排到此存储后   |
| <i>memory_order_acq_rel</i> | 带此内存顺序的读改写操作既是获得操作又是释放操作  |
| <i>memory_order_seq_cst</i> | 有此内存顺序的加载操作进行获得操作，存储操作进行释放操作，而读改写操作进行获得操作和释放操作，再加上存在一个单独全序，其中所有线程以同一顺序观测到所有修改 |

六种 *memory\_order* 可以组合出四种顺序：

#### 1. Relaxed ordering 宽松顺序

```
Thread1:
y.load(std::memory_order_relaxed);
```



Thread2:

```
y.store(h, std::memory_order_relaxed);
```

宽松顺序只保证原子变量的原子性（变量操作的机器指令不进行重排序），但无其他同步操作，不保证多线程的有序性。

#### 1. Release-Acquire ordering 释放获得顺序

```
std::atomic<std::string*> ptr;
```

```
int data;
```

```
void producer()
```

```
{
    std::string* p = new std::string("Hello"); // #1
    data = 42; // #2
    ptr.store(p, std::memory_order_release);
}
```

```
void consumer()
```

```
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))
        ;
    assert(*p2 == "Hello"); // 绝无问题 #3
    assert(data == 42); // 绝无问题 #4
}
```

```
int main()
```

```
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

如例子所示，store 使用 memory\_order\_release，load 使用 memory\_order\_acquire，CPU 将保证如下两点：

- store 之前的语句不允许被重排序到 store 之后（例子中的 #1 和 #2 语句一定在 store 之前执行）
- load 之后的语句不允许被重排序到 load 之前（例子中的 #3 和 #4 一定在 load 之后执行）

同时 CPU 将保证 store 之前的语句比 load 之后的语句「先行发生」，即先执行 #1、#2，然后执行 #3、#4。这实际上就意味着线程 1 中 store 之前的读写操作对线程 2 中 load 执行后是可见的。

**注意是所有操作都同步了，不管 #3 是否依赖了 #1 或 #2**

值得关注的是这种顺序模型在一些强顺序系统例如 x86、SPARC TSO、IBM 主框架上是自动进行的。但在另外一些系统如 ARM、

Power PC 等需要额外指令来保障。

### 3. Release-Consume ordering 释放消费顺序

理解了释放获得顺序顺序后，就非常容易理解释放消费顺序，因为两者十分类似。

```
std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p = new std::string("Hello"); // #1
    data = 42; // #2
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_consume)))
        ;
    assert(*p2 == "Hello"); // #3 绝无出错: *p2 从 ptr 携带依赖
    assert(data == 42); // #4 可能也可能不会出错: data 不从 ptr 携带依赖
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

**store 使用 memory\_order\_release, load 使用 memory\_order\_consume。**其效果与 Release-Acquire ordering 释放获得顺序类似，唯一不同的是**并不是所有操作都同步（不够高效），而是只对依赖操作进行同步，保证其有序性**上例就是 #3 一定发生在 #1 之后，因为这两个操作依赖于 ptr。但不会保证 #4 一定发生在 #2 之后（注意「释放获得顺序」可以保证这一点）。

### 4. Sequential consistency 序列一致顺序

理解上述几种顺序后，Sequential consistency 就很好理解了。

「释放获得顺序」是对某一个变量进行同步，Sequential consistency 序列一致顺序则是对所有变量的所有操作都进行同步。

**store 和 load 都使用 memory\_order\_seq\_cst**，可以理解对每个变量都进行 Release-Acquire 操作。所以这也是最慢的一

种顺序模型。

#### LevelDB 跳表的并发处理

在 LevelDB 的 `skiplist.h` 中, 涉及到了 `atomic` 和 `memory_order`, 我们结合上文的介绍来理解其中的实现逻辑。

首先对跳表的最高高度 `max_height_` 设置了 `atomic`, 并采用 `memory_order_relaxed` 进行读写:

```
// 确保在所有平台下以及内存对齐或非对齐情况下
// 对 max_height_ 的读写都是原子性的
std::atomic<int> max_height_;

// ....

// store 和 load 都采用了 memory_order_relaxed
// 即采用 Relaxed ordering 宽松顺序
// 即对多线程有序性不做保证
max_height_.store(height, std::memory_order_relaxed);

// ...
```

`max_height_.load(std::memory_order_relaxed);`  
`max_height_ 如同实现一个计数器 i++ 一样, 如果多线程读不是原子性的, 那么就会造成类似某个线程读到旧数据或不完整数据的局面。`

其次对跳表结点的索引结点也进行了 `atomic` 的处理, 如下所示:

```
std::atomic<Node*> next_[1];

// ...

// 插入结点时
next_[n].store(x, std::memory_order_release);

// ...
```

```
// 读取结点时
next_[n].load(std::memory_order_acquire);
```

从中可知, 对 `next_[n]` 使用了 `Release-Acquire ordering` 释放获得顺序, 其可以保证某个线程进行 `store` 后, 其他所有执行 `load` 的读线程都将读到 `store` 的最新数据。因为释放获得顺序保证了先 `store` 后 `load` 的执行顺序。

这也正是 LevelDB 的跳表支持多线程读的原因。

值得注意的是其中还实现了 `NoBarrier_SetNext` 和 `NoBarrier_Next`。这两个没有内存屏障的操作实际就是使用了

宽松顺序对 next\_[n] 进行读写。这种操作是线程不安全的，为什么需要这种操作？

```
void SkipList<Key, Comparator>::Insert(const Key& key) {  
    // ...  
  
    // 插入新结点  
    x = NewNode(key, height);  
    for (int i = 0; i < height; i++) {  
        // 这两句相当于：  
        // new_node->next = pre->next;  
        // pre->next = new_node;  
        x->NoBarrier_SetNext(i, prev[i]->NoBarrier_Next(i));  
        prev[i]->SetNext(i, x);  
    }  
}
```

在一个链表中插入一个结点的步骤实际就是：

```
new_node->next = pre->next;  
pre->next = new_node;  
而 new_node->next = pre->next; 这一步赋值不必立马对所有读线程可见，因为此时还未完全插入结点，并不影响读线程的读取。如下图所示：
```

concurrency.png

为什么要特意使用 NoBarrier\_SetNext ？因为宽松顺序效率更高，可以看到 LevelDB 的跳表实现为了性能已经优化到了如此变态的地步。

## 附录

### 源码注解

对 LevelDB 中跳表 skiplist.h 的源码实现做了详细注解，可见[源码注解](#)。

### 操作样例

1. 创建一个 SkipList，数据结构初始化如下图所示：

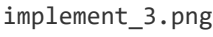
implement\_0.png

1. 新增一个结点，且设 key = 10，随机 height = 4，则数据结构如下图所示：

implement\_1.png

1. 继续新增一个结点，且设 key = 5，随机 height = 3，则数据结构如下图所示：

implement\_2.png

1. 继续新增一个结点, 且设  $\text{key} = 4$ , 随机  $\text{height} = 5$ , 则数据结构如下图所示:  


## 参考资料

汪

汪