zhuanlan.zhihu.com

# RocksDB——内联跳跃表

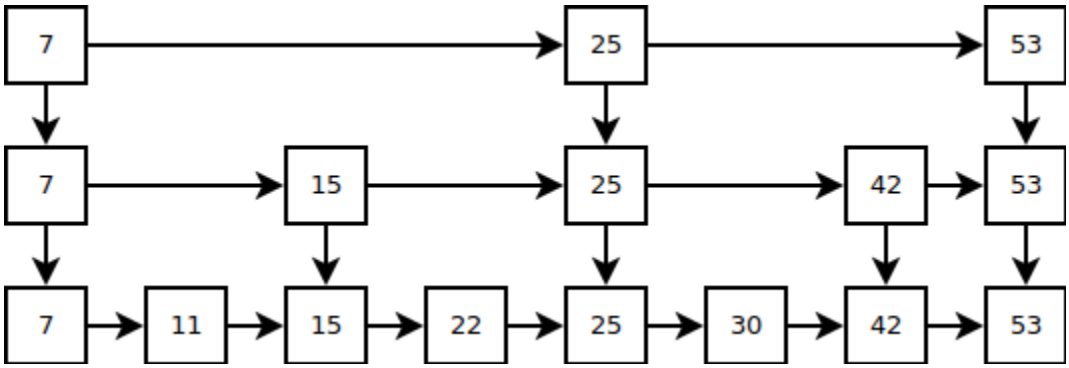
2 minutes

---

这篇文章介绍RocksDB中的memtable之一——内联跳跃表（Inline Skip List）。

> RocksDB是Facebook基于LevelDB研发的键值存储引擎，它不但对LevelDB的不足之处进行了优化（比如多线程压缩），而且引入了很多新特性（比如事务）。

如果你对LevelDB有所了解的话那应该对跳跃表不会陌生，跳跃表在LevelDB中用于存储内存数据，在LevelDB中被称作Memtable。




如果你并不熟悉跳跃表的话，这里有篇博客： Skip Lists: Done Right · Ticki's blog。这篇博客不仅介绍了跳跃表的工作原理而且介绍了一些优化策略。

我们首先介绍一下LevelDB和RocksDB中两种跳跃表的相同之处：

- 都支持一写多读
- 对顺序插入进行了优化，LevelDB中维护了一组prev_指针，RocksDB中使用了Splice在每一层维护了一对指针

内联跳跃表是对于LevelDB中的跳跃表的优化，原理很简单：

```
通过更紧凑的内存安排
1. 减少了内存的使用
2. 提供了更好的局部性
```

比较下两个跳跃表在节点所使用的数据结构差异。

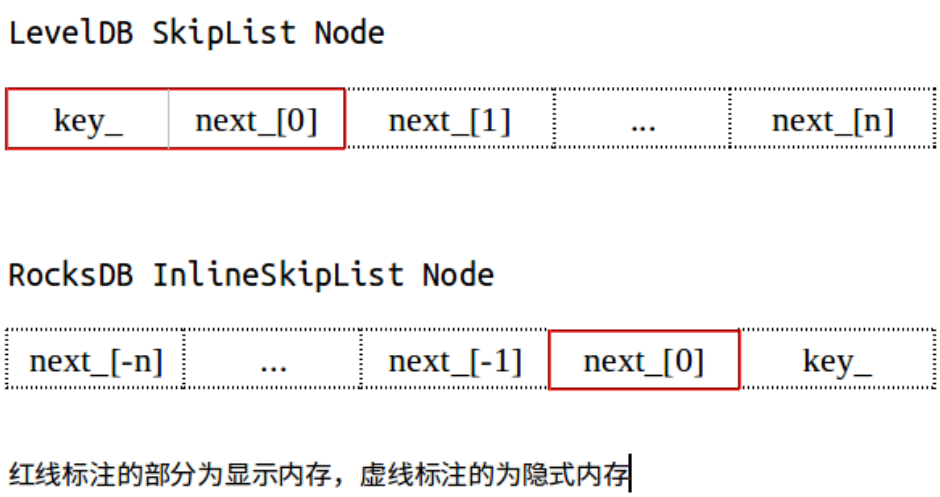
```
class Node {  // LevelDB
  const Key           key_;
  std::atomic<Node*> next_[1];
}


class Node {  // RocksDB
  std::atomic<Node*>  next_[1];
}
```

你可能会有以下疑问：

- 为什么指针只有一个，跳跃表的每个节点应该保存若干个节点指针来维护结构的完整性

- 内联跳跃表中的key去哪了

  这其实都和C/C++的语言特性有关，C/C++提供裸内存，所以你可以通过内存hack来对程序进行很多的优化，有些时候你看到的结构体的大小并不等于其真实大小，下面这张图可以帮助你很好理解这两个结构体的内存布局到底是怎样的。




在LevelDB中，key_可能是一个指针，但是在RocksDB中，key_存放实际值，它的内存在创建新节点时一起分配，因此节省了一个指针的内存（这么说不是特别准确，因为内联跳跃表的节点在分配时需要对齐，所以会浪费小于一个指针的内存，总的来说还是减少了内存的使用）。同时内联跳跃表将各层的next_指针移到了next_域的前面，并且通过next_[-n]这种方式来访问，这是一种C/C++中一种指针使用的小技巧。

其实内联跳跃表的数据结构还可以使用柔型数组，因为现在如果我们要访问key_，需要采用这种方法：

```
char *key = reinterpret_cast<char*>(&next_[1]);
```

改变后的结构体是这样的：

```
class Node {  // RocksDB
  std::atomic<Node*>  next_[1];
  char key_[0];
}
```

如果你仔细的话你会发现这两个结构体都没有height域，也就是说你无法知道next节点的数量，这是它们的另一个特点，那就是不完整，只有在具体调用的函数中你才能知道它们的next节点数量。其实这个特点也可以用来解释next_[-n]这样的存在，如果使用next_[n]的话我们需要height参数来知道具体的key_所在的位置。更过分的是RocksDB在new新节点时会把height写在next_[0]的前4个字节处，然后将这个节点插入的时候再读出来，这时候next_[0]又变成了一个指针。这种做法是不是很刺激？

以上就是RocksDB内联跳跃表在优化方面的介绍。感觉一般人这么写很容易被贴上“不规范编程”和“代码可读性差”的标签。源代码在RocksDB的memtable/skiplist.h和memtable/inlineskiplist.h这两个头文件中。