LevelDB 源码分析「二、基本数据结构续」

2019.07.29 SF-Zhou

本系列的上一篇介绍了 LevelDB 中的 Slice、Hash 和 LRUCache 的实现，这一篇将继续分析布隆过滤器、内存池和跳表。

4. 布隆过滤器 BloomFilter

在介绍布隆过滤器之前，先介绍下 LevelDB 中的过滤器策略 FilterPolicy。考虑一个场景：在 LevelDB 中查询某个指定 key = query 对应的 value，如果我们事先知道了所有的 key 里都找不到这个 query，那也就不需要进一步的读取磁盘、精确查找了，可以有效地减少磁盘访问数量。

FilterPolicy 就负责这件事情：它可以根据一组 key 创建一个小的过滤器 filter，并且可以将该过滤器和键值对存储在磁盘中，在查询时快速判断 query 是否在 filter 中。默认使用的 FilterPolicy 即为布隆过滤器。FilterPolicy 定义于 include/leveldb/filter_policy.h：

```
#include <string>

#include "leveldb/export.h"

namespace leveldb {

class Slice;
```

```
class LEVELDB_EXPORT FilterPolicy {
 public:
  virtual ~FilterPolicy();


  virtual const char* Name() const = 0;
  virtual void CreateFilter(const Slice* keys, int n,
                            std::string* dst) const = 0;
  virtual bool KeyMayMatch(const Slice& key, const Slice& filter) const = 0;
};

LEVELDB_EXPORT const FilterPolicy* NewBloomFilterPolicy(int bits_per_key);

}  // namespace leveldb
```

FilterPolicy 中， CreateFilter 负责创建 filter ， KeyMayMatch 负责判断 key 是否在 filter 中。注意这个 May ，即这里 Match 判断可能会出错，也允许会出错。对于布隆过滤器，如果 Key 在 filter 里，那么一定会 Match 上；反之如果不在，那么有小概率也会 Match 上，进而会多做一些磁盘访问，只要这个概率足够小也无伤大雅。这也刚好符合 KeyMayMatch 函数的需求，有兴趣可以看原代码中的英文注释。

暴露的接口除了 FilterPolicy 接口类，还有 NewBloomFilterPolicy 函数，其他代码中均使用 FilterPolicy 。这样的设计可以保证使用者可以自行定义策略类、方便地替换原有的布隆过滤器。这种设计也称之为策略模式，将策略单独设计为一个类或接口，不同的子类对应不同的策略方法。继续看布隆过滤器的实现 util/bloom.cc ：

```
#include "leveldb/filter_policy.h"
```

```
#include "leveldb/filter_policy.h"

#include "leveldb/slice.h"
#include "util/hash.h"

namespace leveldb {


namespace {
static uint32_t BloomHash(const Slice& key) {
  return Hash(key.data(), key.size(), 0xbc9f1d34);
}

class BloomFilterPolicy : public FilterPolicy {
 public:
  explicit BloomFilterPolicy(int bits_per_key) : bits_per_key_(bits_per_key) {
    // We intentionally round down to reduce probing cost a little bit
    k_ = static_cast<size_t>(bits_per_key * 0.69);  // 0.69 =~ ln(2)
    if (k_ < 1) k_ = 1;
    if (k_ > 30) k_ = 30;
  }

  const char* Name() const override { return "leveldb.BuiltinBloomFilter2"; }

  void CreateFilter(const Slice* keys, int n, std::string* dst) const override {
    // Compute bloom filter size (in both bits and bytes)
    size_t bits = n * bits_per_key_;

    // For small n, we can see a very high false positive rate.  Fix it
    // by enforcing a minimum bloom filter length.
    if (bits < 64) bits = 64;
```

```
  size_t bytes = (bits + 7) / 8;
  bits = bytes * 8;

  const size_t init_size = dst->size();
  dst->resize(init_size + bytes, 0);

  dst->push_back(static_cast<char>(k_));  // Remember # of probes in filter
  char* array = &(*dst)[init_size];
  for (int i = 0; i < n; i++) {
    // Use double-hashing to generate a sequence of hash values.
    // See analysis in [Kirsch,Mitzenmacher 2006].
    uint32_t h = BloomHash(keys[i]);
    const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
    for (size_t j = 0; j < k_; j++) {
      const uint32_t bitpos = h % bits;
      array[bitpos / 8] |= (1 << (bitpos % 8));
      h += delta;
    }
  }
}

bool KeyMayMatch(const Slice& key, const Slice& bloom_filter) const override {
  const size_t len = bloom_filter.size();
  if (len < 2) return false;

  const char* array = bloom_filter.data();
  const size_t bits = (len - 1) * 8;

  // Use the encoded k so that we can read filters generated by
  // bloom filters created using different parameters.
  const size_t k = array[len - 1];
```

```
    if (k > 30) {
      // Reserved for potentially new encodings for short bloom filters.
      // Consider it a match.
      return true;
    }


    uint32_t h = BloomHash(key);
    const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
    for (size_t j = 0; j < k; j++) {
      const uint32_t bitpos = h % bits;
      if ((array[bitpos / 8] & (1 << (bitpos % 8))) == 0) return false;
      h += delta;
    }
    return true;
  }

 private:
  size_t bits_per_key_;
  size_t k_;
};
}  // namespace

const FilterPolicy* NewBloomFilterPolicy(int bits_per_key) {
  return new BloomFilterPolicy(bits_per_key);
}

}  // namespace leveldb
```

BloomFilterPolicy 构造时需要提供 bits_per_key，后再根据 key 的数量 n 一起计算

出所需要的 bits 数 m。而代码中的 k_ 即为布隆过滤器中哈希函数的数目 k，这里 $k = m/n \ln 2$，详细介绍可以参考维基百科。

而后，依次计算 n 个 key 的 k 个哈希结果。这里使用了 Double Hash，即：

$$h(key, i) = h_1(key) + i \cdot h_2(key) \mod 2^{32}$$

Double Hash 一般用于开放寻址哈希的优化。这里直接取连续的 k 个哈希结果作为布隆过滤器需要的 k 个哈希函数结果，一切为了速度。这里的 h_1 为代码中的 BloomHash 函数；而 h_2 为代码中的 delta，也就是 $h_1(key)$ 循环移位 17 位的结果。Match 函数则为逆过程，依次检查各个 bit 是否被标记即可。

5. 内存池 Arena

LevelDB 中实现了一个简单的内存池组建 Arena，位于 util/arena.h：

```
#include <atomic>
#include <cassert>
#include <cstddef>
#include <cstdint>
#include <vector>

namespace leveldb {

class Arena {
 public:
  Arena();
```

```
  Arena(const Arena&) = delete;
  Arena& operator=(const Arena&) = delete;

  ~Arena();


  // Return a pointer to a newly allocated memory block of "bytes" bytes.
  char* Allocate(size_t bytes);

  // Allocate memory with the normal alignment guarantees provided by malloc.
  char* AllocateAligned(size_t bytes);

  // Returns an estimate of the total memory usage of data allocated
  // by the arena.
  size_t MemoryUsage() const {
    return memory_usage_.load(std::memory_order_relaxed);
  }

private:
  char* AllocateFallback(size_t bytes);
  char* AllocateNewBlock(size_t block_bytes);

  // Allocation state
  char* alloc_ptr_;
  size_t alloc_bytes_remaining_;

  // Array of new[] allocated memory blocks
  std::vector<char*> blocks_;

  // Total memory usage of the arena.
```

```
  std::atomic<size_t> memory_usage_;
};

inline char* Arena::Allocate(size_t bytes) {
  // The semantics of what to return are a bit messy if we allow

  // 0-byte allocations, so we disallow them here (we don't need
  // them for our internal use).
  assert(bytes > 0);
  if (bytes <= alloc_bytes_remaining_) {
    char* result = alloc_ptr_;
    alloc_ptr_ += bytes;
    alloc_bytes_remaining_ -= bytes;
    return result;
  }
  return AllocateFallback(bytes);
}

}  // namespace leveldb
```

Arena 提供三个接口， Allocate 、 AllocateAligned 和 MemoryUsage ， 分别实现申请指定大小内存、申请对齐的指定大小内存和查询内存使用。具体的函数实现在 util/arena.cc ：

```
#include "util/arena.h"

namespace leveldb {

static const int kBlockSize = 4096;
```

```
Arena::Arena()
    : alloc_ptr_(nullptr), alloc_bytes_remaining_(0), memory_usage_(0) {}

Arena::~Arena() {
  for (size_t i = 0; i < blocks_.size(); i++) {
    delete[] blocks_[i];
  }
}

char* Arena::AllocateFallback(size_t bytes) {
  if (bytes > kBlockSize / 4) {
    // Object is more than a quarter of our block size.  Allocate it separately
    // to avoid wasting too much space in leftover bytes.
    char* result = AllocateNewBlock(bytes);
    return result;
  }

  // We waste the remaining space in the current block.
  alloc_ptr_ = AllocateNewBlock(kBlockSize);
  alloc_bytes_remaining_ = kBlockSize;

  char* result = alloc_ptr_;
  alloc_ptr_ += bytes;
  alloc_bytes_remaining_ -= bytes;
  return result;
}

char* Arena::AllocateAligned(size_t bytes) {
  const int align = (sizeof(void*) > 8) ? sizeof(void*) : 8;
  static_assert((align & (align - 1)) == 0,
```

```
                "Pointer size should be a power of 2");
  size_t current_mod = reinterpret_cast<uintptr_t>(alloc_ptr_) & (align - 1);
  size_t slop = (current_mod == 0 ? 0 : align - current_mod);
  size_t needed = bytes + slop;
  char* result;

  if (needed <= alloc_bytes_remaining_) {
    result = alloc_ptr_ + slop;
    alloc_ptr_ += needed;
    alloc_bytes_remaining_ -= needed;
  } else {
    // AllocateFallback always returned aligned memory
    result = AllocateFallback(bytes);
  }
  assert((reinterpret_cast<uintptr_t>(result) & (align - 1)) == 0);
  return result;
}

char* Arena::AllocateNewBlock(size_t block_bytes) {
  char* result = new char[block_bytes];
  blocks_.push_back(result);
  memory_usage_.fetch_add(block_bytes + sizeof(char*),
                          std::memory_order_relaxed);
  return result;
}

}  // namespace leveldb
```

代码也很容易看懂。每次会申请一个大的 block，默认大小为 4KB。而后申请 bytes 长度的空间时，如果当前 block 的剩余大小足够分配，则返回分配的内存地址并更新余下的

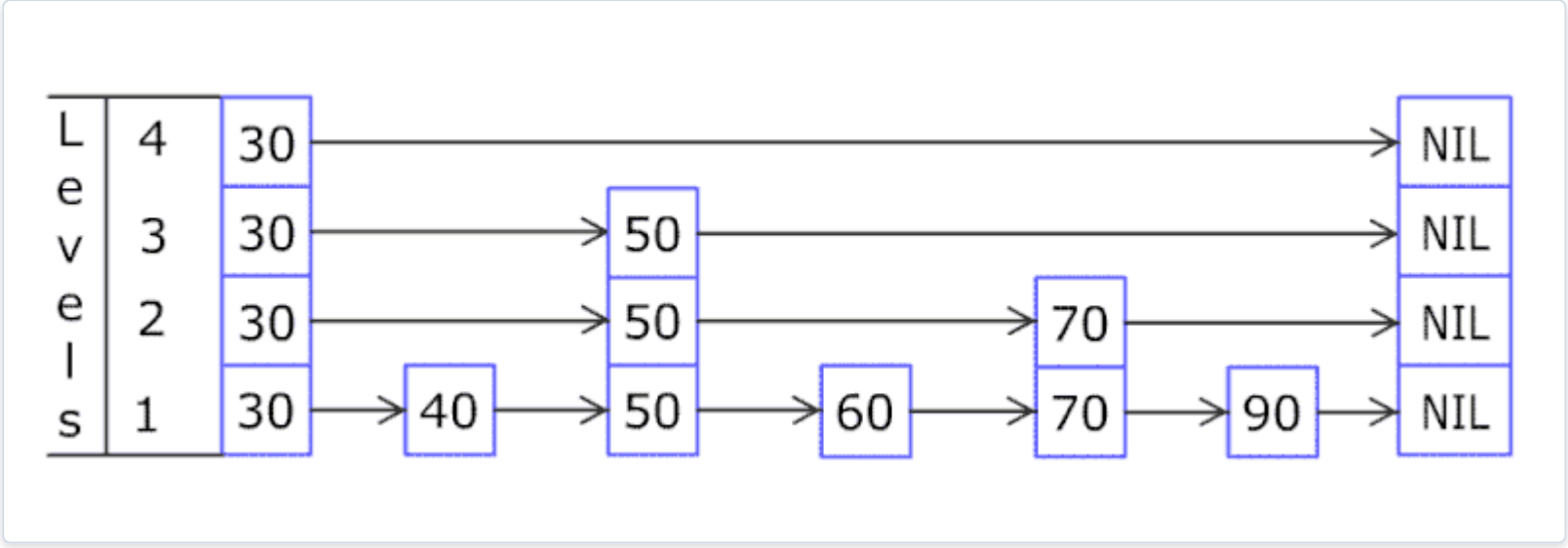
起始位置和大小；否则将会直接申请新的 block。析构时会删除所有 block。

当前空间不足时有一个优化，如果申请的空间大于 kBlockSize / 4 也就是 1KB 时，会直接申请对应长度的 block 返回，不更新当前剩余 block 的起始位置和大小，这样下次申请小空间时依然可以使用当前余下的空间；否则将放弃当前剩余空间，重新申请一块 4KB 的 block 再分配。

当频繁申请小内存时，内存池可以规避掉大部分的系统级申请。接下来的介绍的跳表、以及后续文章介绍的 MemTable 中均会使用到该内存池。

6. 跳表 SkipList

跳表是在传统链表上加入跳跃连接的有序链表。因为有序，所以可以根据顺序关系，快速跳过无关元素。查询和插入的平均复杂度均为 $\mathcal{O}(\log n)$。维基百科上有一副经典图：



跳表插入过程示意图 from 维基百科

LevelDB 中实现的跳表位于 db/skiplist.h，所有实现均在该头文件里。仔细看图，对照代码，就很容易理解了。

```
#include <atomic>
#include <cassert>
#include <cstdlib>

#include "util/arena.h"
#include "util/random.h"

namespace leveldb {

class Arena;

template <typename Key, class Comparator>
class SkipList {
 private:
  struct Node;

 public:
  // Create a new SkipList object that will use "cmp" for comparing keys,
  // and will allocate memory using "*arena".  Objects allocated in the arena
  // must remain allocated for the lifetime of the skiplist object.
  explicit SkipList(Comparator cmp, Arena* arena);

  SkipList(const SkipList&) = delete;
  SkipList& operator=(const SkipList&) = delete;
```

```
// Insert key into the list.
// REQUIRES: nothing that compares equal to key is currently in the list.
void Insert(const Key& key);

// Returns true iff an entry that compares equal to key is in the list.
bool Contains(const Key& key) const;

// Iteration over the contents of a skip list
class Iterator {
 public:
  // Initialize an iterator over the specified list.
  // The returned iterator is not valid.
  explicit Iterator(const SkipList* list);

  // Returns true iff the iterator is positioned at a valid node.
  bool Valid() const;

  // Returns the key at the current position.
  // REQUIRES: Valid()
  const Key& key() const;

  // Advances to the next position.
  // REQUIRES: Valid()
  void Next();

  // Advances to the previous position.
  // REQUIRES: Valid()
  void Prev();

  // Advance to the first entry with a key >= target
```

```
    void Seek(const Key& target);

    // Position at the first entry in list.
    // Final state of iterator is Valid() iff list is not empty.
    void SeekToFirst();


    // Position at the last entry in list.
    // Final state of iterator is Valid() iff list is not empty.
    void SeekToLast();

  private:
    const SkipList* list_;
    Node* node_;
    // Intentionally copyable
  };

private:
  enum { kMaxHeight = 12 };

  inline int GetMaxHeight() const {
    return max_height_.load(std::memory_order_relaxed);
  }

  Node* NewNode(const Key& key, int height);
  int RandomHeight();
  bool Equal(const Key& a, const Key& b) const { return (compare_(a, b) == 0); }

  // Return true if key is greater than the data stored in "n"
  bool KeyIsAfterNode(const Key& key, Node* n) const;
```

```
  // Return the earliest node that comes at or after key.
  // Return nullptr if there is no such node.
  //
  // If prev is non-null, fills prev[level] with pointer to previous
  // node at "level" for every level in [0..max_height_-1].
  Node* FindGreaterOrEqual(const Key& key, Node** prev) const;

  // Return the latest node with a key < key.
  // Return head_ if there is no such node.
  Node* FindLessThan(const Key& key) const;

  // Return the last node in the list.
  // Return head_ if list is empty.
  Node* FindLast() const;

  // Immutable after construction
  Comparator const compare_;
  Arena* const arena_;  // Arena used for allocations of nodes

  Node* const head_;

  // Modified only by Insert().  Read racily by readers, but stale
  // values are ok.
  std::atomic<int> max_height_;  // Height of the entire list

  // Read/written only by Insert().
  Random rnd_;
};

// Implementation details follow
```

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

  Key const key;


  // Accessors/mutators for links.  Wrapped in methods so we can
  // add the appropriate barriers as necessary.
  Node* Next(int n) {
    assert(n >= 0);
    // Use an 'acquire load' so that we observe a fully initialized
    // version of the returned Node.
    return next_[n].load(std::memory_order_acquire);
  }
  void SetNext(int n, Node* x) {
    assert(n >= 0);
    // Use a 'release store' so that anybody who reads through this
    // pointer observes a fully initialized version of the inserted node.
    next_[n].store(x, std::memory_order_release);
  }

  // No-barrier variants that can be safely used in a few locations.
  Node* NoBarrier_Next(int n) {
    assert(n >= 0);
    return next_[n].load(std::memory_order_relaxed);
  }
  void NoBarrier_SetNext(int n, Node* x) {
    assert(n >= 0);
    next_[n].store(x, std::memory_order_relaxed);
  }
```

```
 private:
  // Array of length equal to the node height.  next_[0] is lowest level link.
  std::atomic<Node*> next_[1];
};


template <typename Key, class Comparator>
typename SkipList<Key, Comparator>::Node* SkipList<Key, Comparator>::NewNode(
    const Key& key, int height) {
  char* const node_memory = arena_->AllocateAligned(
      sizeof(Node) + sizeof(std::atomic<Node*>) * (height - 1));
  return new (node_memory) Node(key);
}

template <typename Key, class Comparator>
inline SkipList<Key, Comparator>::Iterator::Iterator(const SkipList* list) {
  list_ = list;
  node_ = nullptr;
}

template <typename Key, class Comparator>
inline bool SkipList<Key, Comparator>::Iterator::Valid() const {
  return node_ != nullptr;
}

template <typename Key, class Comparator>
inline const Key& SkipList<Key, Comparator>::Iterator::key() const {
  assert(Valid());
  return node_->key;
}
```

```
template <typename Key, class Comparator>
inline void SkipList<Key, Comparator>::Iterator::Next() {
  assert(Valid());
  node_ = node_->Next(0);

}

template <typename Key, class Comparator>
inline void SkipList<Key, Comparator>::Iterator::Prev() {
  // Instead of using explicit "prev" links, we just search for the
  // last node that falls before key.
  assert(Valid());
  node_ = list_->FindLessThan(node_->key);
  if (node_ == list_->head_) {
    node_ = nullptr;
  }
}

template <typename Key, class Comparator>
inline void SkipList<Key, Comparator>::Iterator::Seek(const Key& target) {
  node_ = list_->FindGreaterOrEqual(target, nullptr);
}

template <typename Key, class Comparator>
inline void SkipList<Key, Comparator>::Iterator::SeekToFirst() {
  node_ = list_->head_->Next(0);
}

template <typename Key, class Comparator>
inline void SkipList<Key, Comparator>::Iterator::SeekToLast() {
```

```
  node_ = list_->FindLast();
  if (node_ == list_->head_) {
    node_ = nullptr;
  }
}


template <typename Key, class Comparator>
int SkipList<Key, Comparator>::RandomHeight() {
  // Increase height with probability 1 in kBranching
  static const unsigned int kBranching = 4;
  int height = 1;
  while (height < kMaxHeight && ((rnd_.Next() % kBranching) == 0)) {
    height++;
  }
  assert(height > 0);
  assert(height <= kMaxHeight);
  return height;
}

template <typename Key, class Comparator>
bool SkipList<Key, Comparator>::KeyIsAfterNode(const Key& key, Node* n) const {
  // null n is considered infinite
  return (n != nullptr) && (compare_(n->key, key) < 0);
}

template <typename Key, class Comparator>
typename SkipList<Key, Comparator>::Node*
SkipList<Key, Comparator>::FindGreaterOrEqual(const Key& key,
                                              Node** prev) const {
  Node* x = head_;
```

```
  int level = GetMaxHeight() - 1;
  while (true) {
    Node* next = x->Next(level);
    if (KeyIsAfterNode(key, next)) {
      // Keep searching in this list

      x = next;
    } else {
      if (prev != nullptr) prev[level] = x;
      if (level == 0) {
        return next;
      } else {
        // Switch to next list
        level--;
      }
    }
  }
}

template <typename Key, class Comparator>
typename SkipList<Key, Comparator>::Node*
SkipList<Key, Comparator>::FindLessThan(const Key& key) const {
  Node* x = head_;
  int level = GetMaxHeight() - 1;
  while (true) {
    assert(x == head_ || compare_(x->key, key) < 0);
    Node* next = x->Next(level);
    if (next == nullptr || compare_(next->key, key) >= 0) {
      if (level == 0) {
        return x;
      } else {
```

```
        // Switch to next list
        level--;
      }
    } else {
      x = next;

    }
  }
}

template <typename Key, class Comparator>
typename SkipList<Key, Comparator>::Node* SkipList<Key, Comparator>::FindLast()
    const {
  Node* x = head_;
  int level = GetMaxHeight() - 1;
  while (true) {
    Node* next = x->Next(level);
    if (next == nullptr) {
      if (level == 0) {
        return x;
      } else {
        // Switch to next list
        level--;
      }
    } else {
      x = next;
    }
  }
}

template <typename Key, class Comparator>
```

```
SkipList<Key, Comparator>::SkipList(Comparator cmp, Arena* arena)
    : compare_(cmp),
      arena_(arena),
      head_(NewNode(0 /* any key will do */, kMaxHeight)),
      max_height_(1),

      rnd_(0xdeadbeef) {
  for (int i = 0; i < kMaxHeight; i++) {
    head_->SetNext(i, nullptr);
  }
}

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];
  Node* x = FindGreaterOrEqual(key, prev);

  // Our data structure does not allow duplicate insertion
  assert(x == nullptr || !Equal(key, x->key));

  int height = RandomHeight();
  if (height > GetMaxHeight()) {
    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.
    max_height_.store(height, std::memory_order_relaxed);
  }


  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].
    x->NoBarrier_SetNext(i, prev[i]->NoBarrier_Next(i));
    prev[i]->SetNext(i, x);
  }
}

template <typename Key, class Comparator>
bool SkipList<Key, Comparator>::Contains(const Key& key) const {
  Node* x = FindGreaterOrEqual(key, nullptr);
  if (x != nullptr && Equal(key, x->key)) {
    return true;
  } else {
    return false;
  }
}

}  // namespace leveldb
```

References

1. "Bloom filter". *Wikipedia*

1. "Bloom filter", *Wikipedia*
2. "Strategy pattern", *Wikipedia*
3. "Double hashing", *Wikipedia*
4. "Skip List", *Wikipedia*

9 comments – *powered by giscus*

Oldest | Newest

SF-Zhou Aug 5, 2019 Owner

前两篇说的都是局部细节，没有过多的代码依赖，容易看懂。而真正困难的部分，是梳理清楚整体的流程和结构。这时自上而下就要容易很多，比如看看文档、架构图、流程图等，不至于在庞杂的代码细节中迷失。

↑ 1 ☺ 0 replies

zhengdixin Aug 5, 2019

M

↑ 1 ☺ 0 replies

SF-Zhou Aug 5, 2019 Owner

> **@zhengdixin**

> M

这里 M 不了吧

↑ 1 ☺ 0 replies

WingsGo Sep 9, 2019

> **@SF-Zhou**
> 前两篇说的都是局部细节，没有过多的代码依赖，容易看懂。而真正困难的部分，是梳理清楚整体的流程和结构。这时自上而下就要容易很多，比如看看文档、架构图、流程图等，不至于在庞杂的代码细节中迷失。

最近看大型项目的源码，对师兄这句梳理清楚整体的流程和结构，不至于在庞杂的代码细节中迷失深有体会。

↑ 1 ☺ 0 replies

ikiwixx Jun 17, 2020

5.(...当申请大于 1KB 或用完时则申请新的 block。....)这句话我感觉应该是，(用完时则申请新的block，如需要申请的block大于1kb，则根据申请内存大小分配，否则统一分配4kb)

↑ 1 ☺ 0 replies

 **SF-Zhou** Jun 17, 2020 Owner

> **@ikiwixx**
> 5.(...当申请大于 1KB 或用完时则申请新的 block。....)这句话我感觉应该是，(用完时则申请新的block，如需要申请的block大于1kb，则根据申请内存大小分配，否则统一分配4kb)

已更新了表述。

↑ 1 ☺ 0 replies

 **ericuni** Jun 28, 2020 edited

KeyMayMatch 这个函数接口的注释是

> This method must return true if the key was in the list of keys passed to CreateFilter(). This method may return true or false if the key was not on the list, but it should aim to return false with a high probability.

但是bloom filter 又是返回true 的时候, 元素不一定在其中, 返回false 的时候一定不在其中.
两个感觉是刚好相反的, 请问这里怎么理解?

↑ 1 ☺ 0 replies

 **SF-Zhou** Jun 28, 2020 Owner

> **@ericuni**

KeyMayMatch 这个函数接口的注释是

> This method must return true if the key was in the list of keys passed to CreateFilter().
> This method may return true or false if the key was not on the list, but it should aim to return false with a high probability.

但是bloom filter 又是返回true 的时候, 元素不一定在其中, 返回false 的时候一定不在其中.
两个感觉是刚好相反的, 请问这里怎么理解?

英文注释和你的中文表达意思是一致的，逆否命题：
"返回 `false` 的时候一定不在其中" == "在其中一定返回 `true` "

↑ 1 ☺ 0 replies

ericuni Jun 28, 2020

> **@ericuni**
> KeyMayMatch 这个函数接口的注释是
>
> > This method must return true if the key was in the list of keys passed to CreateFilter().
> > This method may return true or false if the key was not on the list, but it should aim to return false with a high probability.
>
> 但是bloom filter 又是返回true 的时候, 元素不一定在其中, 返回false 的时候一定不在其中.
> 两个感觉是刚好相反的, 请问这里怎么理解?
>
> 英文注释和你的中文表达意思是一致的，逆否命题：

> "返回 `false` 的时候一定不在其中" == "在其中一定返回 `true` "

懂了, 这里好绕啊

↑ 1 ☺ 0 replies

Write Preview Aa

Sign in to comment