华中科技大学

大数据存储系统与管理报告

姓　名：　翁哲宇

学　院：　计算机科学与技术学院

专　业：　计算机科学与技术

班　级：　CS2105

学　号：　U202115474

指导教师：　施展

分数	
教师签名	

2024 年 4 月 19 日

目 录

1 数据结构的设计

1.1 Hash_Generator

一个哈希函数生成器，可以自动生成指定类型的哈希函数，并且每次调用时生成的哈希函数都不同。代码如下：

```
template <class T>
class Hash_Generator {
public:
    std::function<int(T)> alloc() {
        int mask = std::hash<int>()(time(0));
        return [=](T a) { return std::hash<T>()(a) ^ mask; };
    };
};
```

通过 std::hash 来实现对指定的类型生成哈希，并且异或上一个每次都不同的 mask，从而做到每次都能生成一个不同的哈希函数。

1.2 CuckooMap

1.2.1 常量

CuckooMap 实现了动态扩容，而扩容所需要的数据是一系列实现定好的常量，定义如下：

```
//用于扩容的 size 备选值，数值来源于 STL
static const int size_num = 28;
static const unsigned long sizes[size_num] = {
    53,       97,       193,       389,       769,       1543,
    3079,     6151,     12289,     24593,     49157,     98317,
    196613,   393241,   786433,    1572869,   3145739,   6291469,
    12582917, 25165843, 50331653,  100663319, 201326611, 402653189,
    805306457, 16106122741, 3221225473ul, 4294967291ul};
```

1.2.2 成员变量

CuckooMap 类内有若干成员变量，其定义与功能如下：

```
//当前大小在 sizes 里的对应下标
int size_pos;
//当前大小
int _size;
//当前元素数量
int num;
//拥挤阈值
double num_limits;
//每个链表元素上限
int num_per_bucket;
// a 和 b 两个链表数组
std::vector<std::list<std::pair<T, P>>> a, b;
//两个基准哈希函数,用于生成实际调用的哈希函数
std::function<int(T)> Hash;
//两个面具，用于 rehash
int maska, maskb;
//冲突次数，指两桶均满的冲突
int conflicts;
// rehash 次数，如果多了说明当前容量不行
int rehashs;
//开启调试模式
bool debug;
```

1.2.3 迭代器

为了方便容器的遍历，以及便于返回查找结果，为 Cuckoo 实现了迭代器。迭代器的定如下：class iterator {

```
public:
    typedef typename std::list<std::pair<T, P>>::iterator Iter;
    //链表指针
    Iter ptr;
    //父实例
```

```
    CuckooMap<T, P> &outer;
    //当前在哪个数组
    int vec_num;
    //在当前数组的第几个链表
    int pos;

public:
    // 迭代器构造函数
    iterator(CuckooMap<T, P> &_outer, Iter p, int _vec_num, int _pos) :
        outer(_outer), ptr(p), vec_num(_vec_num), pos(_pos) {
    }
    // 指针解引用操作符重载
    std::pair<T, P> &operator*() {
        return *ptr;
    }
    //成员访问重载
    Iter operator->() {
        return ptr;
    }
    // 前缀自增操作符重载
    iterator &operator++() {
        /*内容过长，省略这部分*/
    }

    // 后缀自增操作符重载
    iterator operator++(int) {
        // todo
        iterator temp = *this;
        ++*this;
        return temp;
    }
    // 比较操作符重载
    bool operator!=(const iterator &other) const {
        return ptr != other.ptr;
    };
```

```
    // 比较操作符重载
    bool operator==(const iterator &other) const {
        return ptr == other.ptr;
    };
};
```

1.2.4 成员函数

为了方便 CuckooMap 的使用以及优化，定义了如下的成员函数：

```
//默认构造函数
    CuckooMap<T, P>()
    //指定哈希函数进行构造
    CuckooMap<T, P>(std::function<int(T)> _hash)
    //析构函数
    ~CuckooMap<T, P>()
    //实际调用的哈希函数 a
    int hasha(T key)
    //实际调用的哈希函数 b
    int hashb(T key)
    //清空
    void clear()
    //判断是否需要扩容
    bool need_enlarge()
    //扩容
    void enlarge()
    //修改阈值
    void change_limits(double _limits)
    //修改每个链表的元素上限
    void change_num_limits(int _nums)
    //插入键值对
    void insert(const T &key, const P &value)
    //插入 pair
    void insert(std::pair<T, P> p)
```

```
//重载下标运算符
P &operator[](T key)
//删除键值对,返回下一个元素的迭代器
iterator erase(iterator it)
//删除键值对,返回下一个元素的迭代器
iterator erase(const T &key)
//查找键值对
iterator find(const T &key)
//键值对计数(其实只有 0 或 1)
int count(const T &key)
//首
iterator begin()
//尾
iterator end()
//返回键值对数量
int size()
//输出哈希表信息
void debug_info()
//开启调试模式,会输出一些调试信息，仅开发时调试用。
void debug_on()
//是否需要重新哈希
bool need_rehash()
//重新哈希
void rehash()
//调整过长的链表，只在修改 nums_per_list 或 rehash 后使用
void shrink()
//是否为空
bool empty()
```

2 操作流程分析

取主要的几个函数来进行讲解

2.1 迭代器自增操作

从当前迭代器触发，先依次遍历当前 list 中的元素，如果没有，则寻找下一个 list，以此类推，直到遍历到最后一个 list 的 end。代码如下：

```
iterator &operator++() {
        ptr++;
        if (vec_num == 0) { //当前在 a 数组
            auto &vec = outer.a;
            if (ptr != vec[pos].end()) return *this;
            pos++;
            while (pos < vec.size()) {
                if (!vec[pos].empty()) break;
                pos++;
            }
            if (pos == vec.size()) {
                vec_num = 1;
                auto &Vec = outer.b;
                pos = 0;
                while (pos < Vec.size()) {
                    if (!Vec[pos].empty()) break;
                    pos++;
                }
                // b 数组走到了最后，返回 end
                if (pos == Vec.size()) {
                    ptr = Vec.back().end();
                    return *this;
                }
                //在 b 数组中找到了
                ptr = Vec[pos].begin();
```

```
            return *this;
        }
        //在 a 数组找到了
        ptr = vec[pos].begin();
        return *this;
    } else { //当前在 b 数组
        auto &vec = outer.b;
        if (ptr != vec[pos].end()) return *this;
        pos++;
        while (pos < vec.size()) {
            if (!vec[pos].empty()) break;
            pos++;
        }
        // b 数组走到了最后，返回 end
        if (pos == vec.size()) {
            ptr = vec.back().end();
            return *this;
        }
        //在 b 数组中找到了
        ptr = vec[pos].begin();
        return *this;
    }
    //理论上不会走到这
    return *this;
}
```

2.2 元素查找

对当前的 key 通过两个哈希函数算出它在两个 vector 中的位置，然后遍历链表查询是否有这个 key，有则返回对应的迭代器，无则返回 end 迭代器。代码如下：

```
//查找键值对
iterator find(const T &key) {
    int posa = hasha(key) % _size;
    int posb = hashb(key) % _size;
```

```
    if (debug) {
        printf("(%d,%d)\n", posa, a.size());
        std::cout << std::endl;
    }
    for (auto it = a[posa].begin(); it != a[posa].end(); it++)
        if (it->first == key) return iterator(*this, it, 0, posa);
    for (auto it = b[posb].begin(); it != b[posb].end(); it++)
        if (it->first == key) return iterator(*this, it, 1, posa);
    return end();
}
```

2.3 元素插入

首先判断当前的 key 是否在容器中，如果在的话直接利用迭代器对 value 进行修改。如果不在的话，首先判断两个候选 list 哪个元素数量少，将新元素插入到内容较少的 list 中。如果两个 list 均满的话就会启动替换策略替换掉其中的一个元素，然后对新的元素递归使用插入操作。代码如下：

```
//插入键值对
void insert(const T &key, const P &value) {
    if (need_rehash()) rehash();
    iterator it = find(key);
    if (it != end()) { //原本就在
        it->second = value;
        return;
    }
    //新增键值对
    if (need_enlarge()) enlarge();
    num++;
    int posa = hasha(key) % _size;
    int posb = hashb(key) % _size;
    if (debug) printf("insert key %d to pos(%d,%d)\n", key, posa, posb);
    //没满就加到较小的桶里
    if (a[posa].size() > b[posb].size()) {
        b[posb].emplace_back(key, value);
        //类似拥塞控制的指数下降，线性增长
```

```
            conflicts /= 2;
        } else if (a[posa].size() < num_per_bucket) {
            a[posa].emplace_back(key, value);
            conflicts /= 2;
        } else {
            //虽然弹出的元素如果也全满，直接调用插入有二分之一的概率选到同一个桶，但由于添加元素是从桶位加，弹出元素是从桶头弹，因此选中的不是同一个元素，与另选一个桶没有区别。
            num--;
            conflicts++;
            if (rand() % 2) {
                auto temp = a[posa].front();
                a[posa].pop_front();
                a[posa].emplace_back(key, value);
                insert(temp.first, temp.second);
            } else {
                auto temp = a[posb].front();
                a[posa].pop_front();
                a[posa].emplace_back(key, value);
                insert(temp.first, temp.second);
            }
        }
    }
```

2.4 元素删除

利用 find 找到当前元素，直接从链表中删除，然后返回下一个元素的迭代器，代码如下：

```
    //删除键值对,返回下一个元素的迭代器
    iterator erase(iterator it) {
        iterator nxt = it;
        nxt++;
        if (it.vec_num == 0) {
            it.outer.a[it.pos].erase(it.ptr);
```

```
    } else {
        it.outer.b[it.pos].erase(it.ptr);
    }
    num--;
    return nxt;
}
```

2.5 动态扩容

当元素数量满足一定条件时会触发动态扩容，增大 vector 的大小，从而能把元素映射到一个更宽广的范围内。考虑到每个元素在新的容量下改变位置并不会影响其它的元素，并且对同一个元素二次进行查找位置的操作也不会出错，因此可以直接使用原地的算法来进行元素的转移，而不需要将元素进行大量的拷贝后重新插入。同时因为原本的元素一定在[0,oldsize-1]这个区间内，所以遍历时不需要遍历[oldsize,_size-1]这个区间。

```
//扩容
void enlarge() {
    if (debug) printf("enter enlage\n");
    size_pos++;
    int oldsize = _size;
    _size = sizes[size_pos];
    a.resize(_size);
    b.resize(_size);
    //同一个链表内的元素在扩容后不一定在同一个链表内，因此不能整个链表转移
    for (int i = 0; i < oldsize; i++) {
        if (a[i].empty()) continue;
        auto it = a[i].begin();
        while (it != a[i].end()) {
            int posa = hasha(it->first) % _size;
            if (posa == i) {
                it++;
                continue;
            }
            a[posa].push_back(*it);
```

```
            it = a[i].erase(it);
        }
    }
    for (int i = 0; i < oldsize; i++) {
        if (b[i].empty()) continue;
        auto it = b[i].begin();
        while (it != b[i].end()) {
            int posb = hashb(it->first) % _size;
            if (posb == i) {
                it++;
                continue;
            }
            b[posb].push_back(*it);
            it = b[i].erase(it);
        }
    }
    debug_info();
}
```

2.6 重新哈希

对于检测到疑似死循环的情况，需要重新进行哈希。首先需要更换哈希函数。之后经过分析后可以发现，除了不需要扩容以外，重新哈希的一部分操作是与扩容一致的，因此可以直接调用扩容函数但不进行扩充来进行元素的重新定位。由于元素修改位置后，可能出现某个位置的元素超过元素数量上限的情况，这个时候就需要调用 shink 函数来对这些超出上限的元素重新进行插入。代码如下：

```
void rehash() {
    conflicts = 0;
    if (debug) printf("enter rehash\n");
    // rehashs 后依旧失败，选择扩容
    if (rehashs && size_pos != 27) {
        enlarge();
        return;
    }
```

```
    rehashs++;
    //容量已经是最大，再考虑重新哈希
    maska = std::hash<int>()(rand());
    maskb = std::hash<int>()(rand());
    //重新哈希的操作其实跟扩容一样，只是不需要扩容，因此代码可以复用
    size_pos--;
    enlarge();
    //重新 hash 后可能有些链表元素数量不满足 num_per_list
    shrink();
    //能跑到这里说明 shrink 运行成功了，新的哈希表成功保存了所有函数
    //否则 shink 运行过程中会再次递归触发 rehash
    rehashs = 0;
}
```

3 理论分析

Cuckoo 不存在误判的情况，因此 false positive 和 false negetive 均为 0。当 list 的大小上限比较小（比如 1），而插入的元素又量特别大以至于动态扩容已经扩到最大时，可能不存在一个合法的方案能够容纳所有的元素，这个时候会超出动态扩容的上限从而触发 panic。

4 性能测试

4.1 不同 list 容量下的性能：

```
insert 1000000 elements spent 1.90 sec with bucket size 4.
insert 1000000 elements spent 1.82 sec with bucket size 8.
insert 1000000 elements spent 1.71 sec with bucket size 16.
insert 1000000 elements spent 1.83 sec with bucket size 32.
insert 1000000 elements spent 2.19 sec with bucket size 64.
insert 1000000 elements spent 2.92 sec with bucket size 128.
```

可以看到速度呈先减小后增大的趋势。这是因为对于固定的阈值，桶的大小过小的话会频繁触发动态扩容，而大部分桶处于空的状态，造成时间和空间的浪费，同时由于桶的过小，所以触发替换的概率也会提高。

桶的大小过大的话，每个桶内的元素也会处于一个较多的状态，每次通过哈希找到对应的桶时，还需要花较多时间遍历链表才能找到对应的元素，浪费了时间。

因此，最佳的情况是桶的大小与阈值的乘积在 1.5 左右，也就是平均每个桶放一两个元素，因为这样既能充分地利用每个桶，又能防止元素在同一个桶内大量堆积。从测试结果也可以看到，当桶的大小为 16 时速度最快，也正是这个时候大小和阈值的乘积最接近 1.5。

4.2 与 std::unordered_map 进行对比

4.2.1 时间

使用 STL 标准库里的哈希表进行同样的一百万的元素的插入，结果如下：

```
unordered_map insert 1000000 elements spent 0.84 sec.
```

只比标准库慢了一倍，性能还是可以的。

4.2.2 空间

使用 gnu time 工具来进行峰值内存的统计，cuckoo 表现如下：

```
root@iZbp1hf0ex12aff1lmsq0kZ:~/bigdata-storage-experiment-assignment-2024/U202115474/cuckoo# /usr/bin/time -v ./cucko
o
start testing insert
start testing find
start testing count
start testing erase
All tests passed!
insert 1000000 elements spent 1.78 sec with bucket size 16.
        Command being timed: "./cuckoo"
        User time (seconds): 2.05
        System time (seconds): 0.05
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:02.11
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 61012
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 18848
        Voluntary context switches: 1
        Involuntary context switches: 48
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Unordered_map 表现如下：

```
root@iZbp1hf0ex12aff1lmsq0kZ:~/bigdata-storage-experiment-assignment-2024/U202115474/cuckoo# /usr/bin/time -v ./cucko
o
start testing insert
start testing find
start testing count
start testing erase
All tests passed!
unordered_map insert 1000000 elements spent 0.85 sec.
        Command being timed: "./cuckoo"
        User time (seconds): 1.04
        System time (seconds): 0.07
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.13
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 54076
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 16150
        Voluntary context switches: 1
        Involuntary context switches: 76
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

两者的内存消耗峰值并没有显著区别