华中科技大学

大数据存储与管理报告

姓　　名：　　郑卯杨

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　计算机 2105

学　　号：　　U202115478

指导教师：　　施展

分数	
教师签名	

2024 年 4 月 22 日

目　录

一　数据结构设计 1

1.1　CuckooHash 1

二 具体实现 2

1.1　Hash 2

1.2　查找键值对 2

1.3　查询 3

1.4　插入 3

1.5　删除 5

1.6　遍历 5

1.7　运算符重载 6

三 性能优化 7

1.1　参数对性能的影响 7

1.2　查询并行 8

一　数据结构设计

1.1 CuckooHash

CuckooHash 结构如图 1 所示。使用两桶多槽 Hash,bucket_size 指示桶的大小,slot_size 指示桶一格对应的槽位数,factor 指示扩容时新容量因扩大为原容量的多少倍，used 用于计数有多少槽位被使用。Bucket 对应的数据结构为 Vec<Vec<Option<(K, V)>>>, 使用 Option<(K,V)>对应一个可用槽位,None 即为空,Some((K,V))为非空。

```
pub struct Cuckoo<K, V> {
    bucket_size: usize,
    slot_size: usize,
    factor: usize,
    used: usize,
    max_loop_times: usize,
    bucket_one: Vec<Vec<Option<(K, V)>>>,
    bucket_two: Vec<Vec<Option<(K, V)>>>,
}
```

图 1 Cuckoo

为了获得较高性能,需要选取计算快速,分布均匀,冲突较小的 Hash 函数。经研究,选用 SipHash 和 FnvHash 函数。通过 type 定义关键字定义 Hasher 实现来隐藏具体的 Hasher 类型,方便替换 Hash 函数。Hash 选取如图 2 所示。

```
type HashImplOne = DefaultHasher;
type HashImplTwo = FnvHasher;
```

图 2 Hasher

二 具体实现

1.1 Hash

Hash 函数传入的 key 并非为 Cuckoo<K,V>对应的 K 类型,而是类型 Q,但是 K 可以借用为 Q。Q 作为实际使用的类型需要满足 Hash+Eq 特性,而?Sized 标记则表明 Q 可以为非定长类型。Hash 特型标识该类型如何计算 Hash。任何类型都可以默认借出为自身的引用。这样做是为了简化操作。以 String 为例,&String 可以借用为&str,因此可以简单地调用 h1("abc")否则需要 h1(&String::from("abc"));为了一次操作而构造一个具有堆内存的容器，开销过大。Hash 函数如图 2 所示

```rust
/// 默认使用std::DefaultHasher
fn h1<Q>(&self, k: &Q) -> usize
where
    K: Borrow<Q>,
    Q: Hash + Eq + ?Sized,
{
    let mut state: DefaultHasher = HashImplOne::default();
    k.hash(&mut state);
    (state.finish() % self.bucket_size as u64) as usize
}
/// 默认使用fnv::FnvHasher
fn h2<Q>(&self, k: &Q) -> usize
where
    K: Borrow<Q>,
    Q: Hash + Eq + ?Sized,
{
    let mut state: FnvHasher = HashImplTwo::default();
    k.hash(&mut state);
    (state.finish() % self.bucket_size as u64) as usize
}
```

图 2 Hash 函数

1.2 查找键值对

使用内部函数 index_of_key 查找键值对,返回值分别标识 在哪个桶,桶位置,槽位置。使用 Rust 迭代器以及模式匹配简化代码。如图 3 所示。

图 3　index_of_key

1.3 查询

查询是对 index_of_key 的简单封装,如图 4 所示

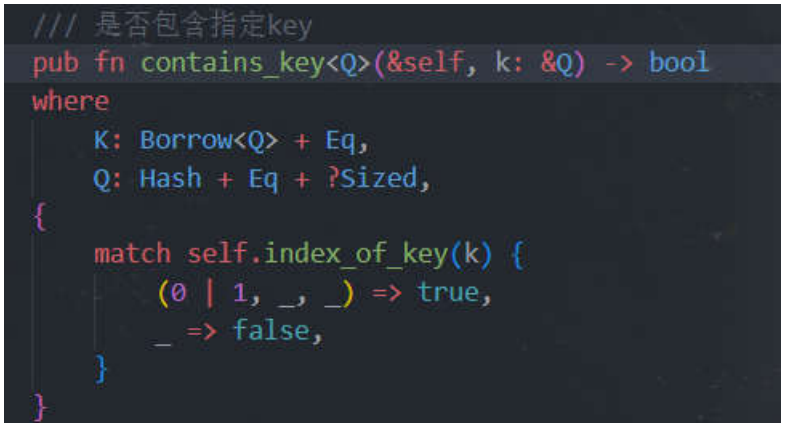


图 4 contains_key

1.4 插入

插入操作分为三步,检查是否有 k 对应的键值对,若有,原地修改;否则尝试插入,若有空位，直接插入；否则采用 kick 操作，若循环次数超过 MAX_LOOP 则需要扩容,扩容后再度插入。

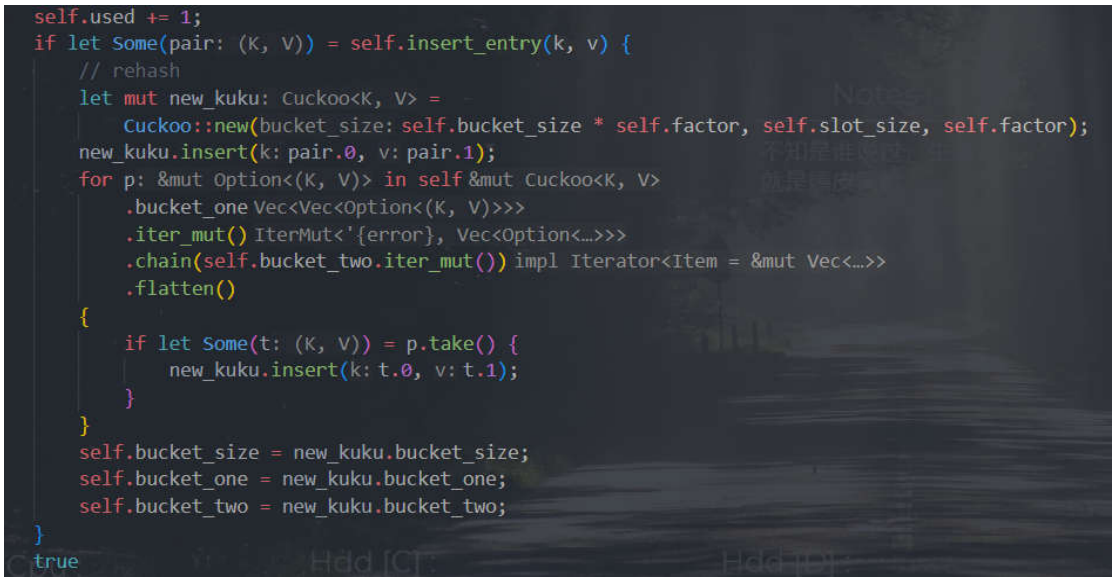
尝试插入操作如图 5 所示.插入成功返回 None,否则返回最后被 kick 的键值对。

```rust
let mut pair: (K, V) = (k, v);
for _ in 0usize..self.max_loop_times {
    let p1: usize = self.h1(&pair.0);
    if let Some((idx: usize, _)) = self.bucket_one[p1].Vec<Option<(K, V)>>
        .iter() Iter<'{error}, Option<(K, …)>>
        .enumerate() impl Iterator<Item = (usize, …)>
        .find(|(_, op: &&Option<(K, V)>)| op.is_none())
    {
        self.bucket_one[p1][idx] = Some(pair);
        return None;
    }
    let mut new_pair: Option<(K, V)> = Some(pair);
    swap(
        x: &mut new_pair,
        y: &mut self.bucket_one[p1][random::<usize>() % Cuckoo::<K, V>::SLOT_SIZE],
    );
    pair = new_pair.unwrap();

    let p2: usize = self.h2(&pair.0);
    if let Some((idx: usize, _)) = self.bucket_two[p2].Vec<Option<(K, V)>>
        .iter() Iter<'{error}, Option<(K, …)>>
        .enumerate() impl Iterator<Item = (usize, …)>
        .find(|(_, op: &&Option<(K, V)>)| op.is_none())
    {
        self.bucket_two[p2][idx] = Some(pair);
        return None;
    }
    let mut new_pair: Option<(K, V)> = Some(pair);
    swap(
        x: &mut new_pair,
        y: &mut self.bucket_two[p2][random::<usize>() % Cuckoo::<K, V>::SLOT_SIZE],
    );
    pair = new_pair.unwrap();
}
Some(pair)
```

图 5 try_insert

扩容操作如图 6 所示。Vec 原地扩容后也需要重新插入所有键值对，已有的键值对会被大量 kick，所以选用建立新 Cuckoo 然后插入所有键值。



```rust
self.used += 1;
if let Some(pair: (K, V)) = self.insert_entry(k, v) {
    // rehash
    let mut new_kuku: Cuckoo<K, V> =
        Cuckoo::new(bucket_size: self.bucket_size * self.factor, self.slot_size, self.factor);
    new_kuku.insert(k: pair.0, v: pair.1);
    for p: &mut Option<(K, V)> in self &mut Cuckoo<K, V>
        .bucket_one Vec<Vec<Option<(K, V)>>>
        .iter_mut() IterMut<'{error}, Vec<Option<…>>>
        .chain(self.bucket_two.iter_mut()) impl Iterator<Item = &mut Vec<…>>
        .flatten()
    {
        if let Some(t: (K, V)) = p.take() {
            new_kuku.insert(k: t.0, v: t.1);
        }
    }
    self.bucket_size = new_kuku.bucket_size;
    self.bucket_one = new_kuku.bucket_one;
    self.bucket_two = new_kuku.bucket_two;
}
true
```

图 6 扩容

1.5 删除

删除操作只需找出对应 k 对应的键值对,随后转移所有权即可，开销低。如图 7 所示

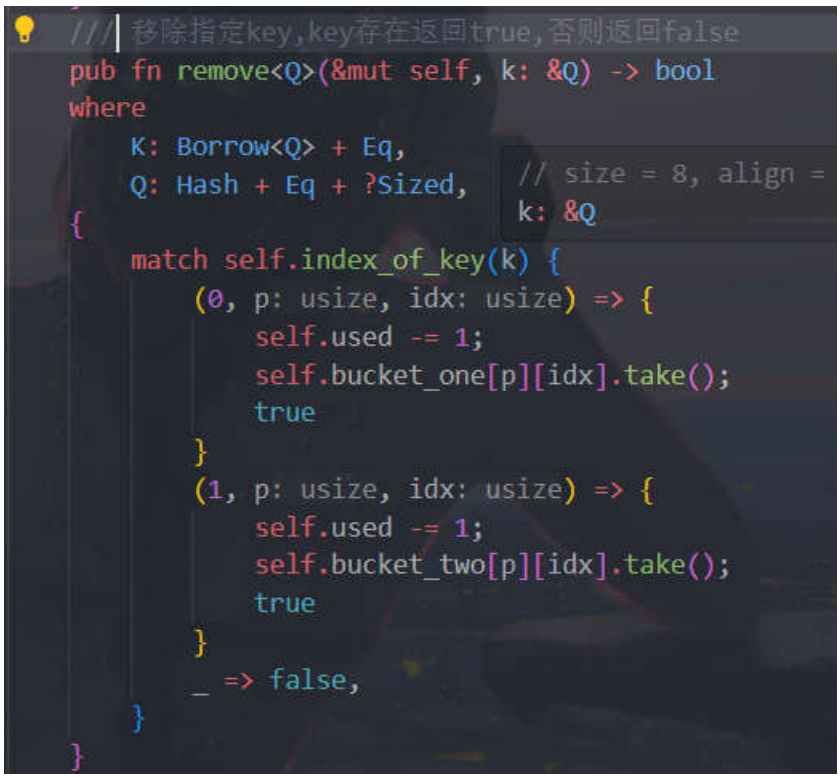


图 7 remove

1.6 遍历

对外提供了迭代器和可变迭代器来支持遍历 Cuckoo,因为内部使用 Vec 作为存储结构,而 Vec 实现了迭代器以及可变迭代器，只需将 Vec 的迭代器串联后展开即可。如图 8 所示。

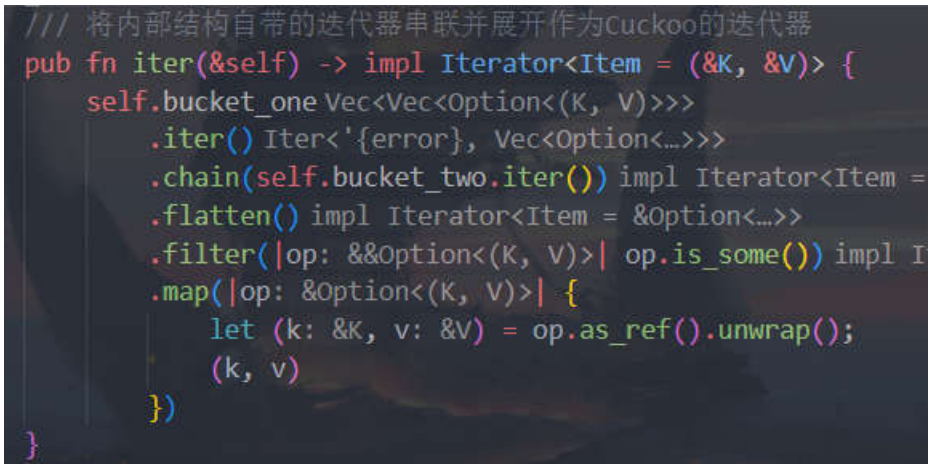


图 8 iter

1.7 运算符重载

重载中括号运算符,方便对 Cuckoo 的访问。如图 9 所示

```rust
/// 重载运算符[],不存在对应的key就Panic
impl<K, V, Q> Index<&Q> for Cuckoo<K, V>
where
    K: Eq + Hash + Borrow<Q>,
    Q: Eq + Hash + ?Sized,
{
    type Output = V;
    fn index(&self, k: &Q) -> &Self::Output {
        self.get(k).unwrap()
    }
}

impl<K, V, Q> IndexMut<&Q> for Cuckoo<K, V>
where
    K: Eq + Hash + Borrow<Q>,
    Q: Eq + Hash + ?Sized,
{
    fn index_mut(&mut self, k: &Q) -> &mut Self::Output {
        self.get_mut(k).unwrap()
    }
}
```

图 9 重载运算符[]

三 性能优化

1.1 参数对性能的影响

Cuckoo Hash 插入时最大的耗时来自于 指定位置被占用时需要进行的 kick 操作；Cuckoo 通过 MAX_LOOP_TIMES 和 FACTOR 来控制最大循环次数以及扩容的倍数，显然，缩小 MAX_LOOP_TIMES 并增大 FACTOR 可能带来性能提升。在 Cuckoo 整体空间利用率下降不大的情况下调参观察性能；测试结果如图 10 所示。

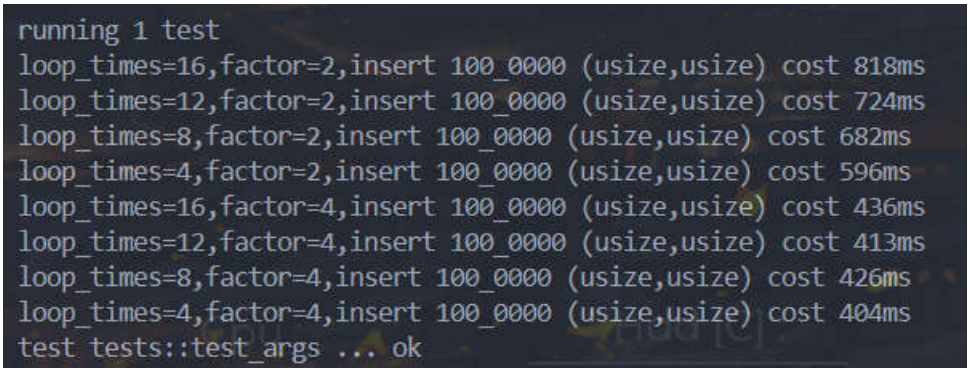


图 10 test_args

发现当扩容因子等于 2 时,减小 MAX_LOOP_TIMES 能显著提升插入性能，因为负载较高时，冲突频率大。当容量扩大 4 倍后，数据密度显著降低，扩容后插入数据冲突较少，对性能影响不大。

测试扩容因子增大后是否会严重影响空间利用率。发现当扩容因子大于 4 后空间利用率大幅降低，不予采用。但扩容因子在 2-4 时,提高扩容因子能获得较大的性能提升，而空间利用率几乎不会降低。使用 rust HashMap 作为参照组。采用两组典型参数探索。第一组 MAX_LOOP_TIMES 为 16,扩容因子为 2；第二组 MAX_LOOP_TIMES 为 8，扩容因子为 4.测试结果如图 11，图 12 所示。

图 11 第一组

```
Cuckoo insert 1000000 (k,v) cost 342ms
HashMap insert 1000000 (k,v) cost 69ms
Cuckoo Space utilization rate = 0.9537
Cuckoo remove 500028 (k,v) cost 72ms
HashMap remove 500028 (k,v) cost 31ms
Cuckoo Space utilization rate = 0.5618
Cuckoo query 499972 (k,v) cost 107ms
HashMap query 499972 (k,v) cost 29ms
Cuckoo get 499972 (k,v) cost 103ms
HashMap get 499972 (k,v) cost 0ms
```

图 12 第二组

1.2 查询并行

自己的收获，对实验的建议等。

最初的实现串行查询两个桶，优化后同时查询两个桶。代码如图 13，图 14 所示。

```rust
pub fn contains_key<Q>(&self, k: &Q) -> bool
where
    K: Borrow<Q> + Eq,
    Q: Hash + Eq + ?Sized,
{
    let p1: usize = self.h1(k);
    let p2: usize = self.h2(k);
    let ok1: bool = self.bucket_one[p1] Vec<Option<(K, V)>>
        .iter() Iter<'{error}, Option<(K, …)>>
        .any(|op: &Option<(K, V)>| op.is_some() && op.as_ref().unwrap().0.borrow() == k);
    let ok2: bool = self.bucket_two[p2] Vec<Option<(K, V)>>
        .iter() Iter<'{error}, Option<(K, …)>>
        .any(|op: &Option<(K, V)>| op.is_some() && op.as_ref().unwrap().0.borrow() == k);
    ok1 | ok2
}
```

图 13 串行查询

```rust
pub fn contains_key<Q>(&self, k: &Q) -> bool
where
    K: Borrow<Q> + Eq,
    Q: Hash + Eq + ?Sized,
{
    let p1: usize = self.h1(k);
    let p2: usize = self.h2(k);
    self.bucket_one[p1] Vec<Option<(K, V)>>
        .iter() Iter<'{error}, Option<(K, …)>>
        .zip(self.bucket_two[p2].iter()) impl Iterator<Item = (&Option<…>, …)>
        .any(|(o1: &Option<(K, V)>, o2: &Option<(K, V)>)| {
            (match o1 {
                Some((ref key: &K, _)) => key.borrow() == k,
                _ => false,
            }) || (match o2 {
                Some((ref key: &K, _)) => key.borrow() == k,
                _ => false,
            })
        })
}
```

图 14 并行查询

测试后发现无明显性能差距，可能是因为 Rust release 模式下优化度高