HashMap ⟶ HashTable 的 Map 接口实现

<key, value>

① null 可以作为 key, value
② 线程不安全

区别

null 不可以作为
线程安全

HashTable 几乎已经

X

若要线程安全

Concurrent HashMap

JDK 1.8 之前

HashMap

<key, value> → HashCode(key)

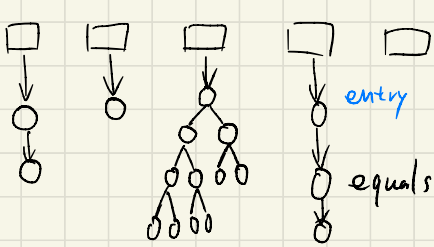
主体 数组
+
Hash冲突 链表

entry

equals true

不同的 key hash 值
有可能相等

JDK 1.8 之后



链表长度 > 8

红黑树

loadFactor (0, 1)

0 → 疏　数组利用率低

1 → 密　查找慢

合理值 (0.75f)

threshold = capacity * loadFactor

if (size >= threshold) → 扩容

size　<k, v> 的数量

capacity　桶的数量 — default = 16；第一次扩容 = 64　2^

---

1. 计算 <k, v> 中 key 对应桶的 Index

① 计算 h = key.hashCode( )

② 高位运算 h ^ (h >>> 16)

③ 取模运算 h & (n-1)

2. put 方法

① 判断 数组是否未初始化 → resize( )

② table[i] = null ✓ → 直接添加新结点；✗ ↓

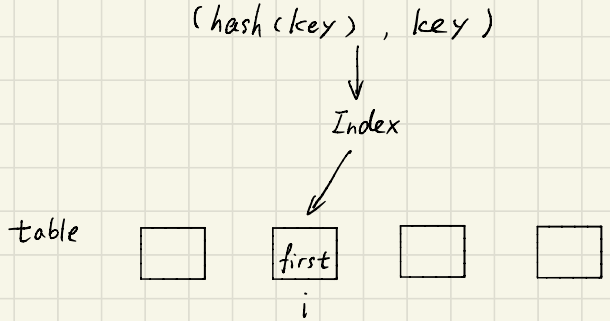
③ key 与 table[i] 首个元素一致 (hashCode( ) 和 equals( )) ✓ → 覆盖 value；✗ ↓

④ table[i] instanceof treeNode ✓ → 插入 <key, value>；✗ ↓

⑤ ListNode.size( ) >= 8 ✓ → 红黑树 → 插入 <key, value>；✗ → 链表 → 插入 <key, value> || 覆盖 value

⑥ <key, value>.size > threshold → resize( )

3、get 方法



① table[i] = null → null

② first.hash == hash
&&
first.key == key
&&
key != null
&&
key.equals(first.key)

③ treeNode → get

④ ListNode → 类似②

4、resize 方法

JDK 1.7

① 判断旧 Entry 数组是否达到 2^30 容量 → 改阈值为 2^31 - 1

② 初始化一个新的 Entry 数组

③ 数据转移 →

④ table 引用新 Entry 数组

⑤ 修改阈值

```
for (int i=0; i < src.length; i++)
 {
   Entry<K,V> e = src[i];
  do {
   Entry<K,V> next = e.next;
        e.next = table[index];
        table[i] = e;
        e = next;
 while (e != null)
```

0 □

1 □ → 3 → 7 → 5 → null

Key = 3, 7, 5

2 × 0.75] = 1

打算 <<<1

0 □

1 □ → 5 → null

2 □

3 □ → 7 → 3 → null

初始化

Map<Integer, String> map = new HashMap<>(10);

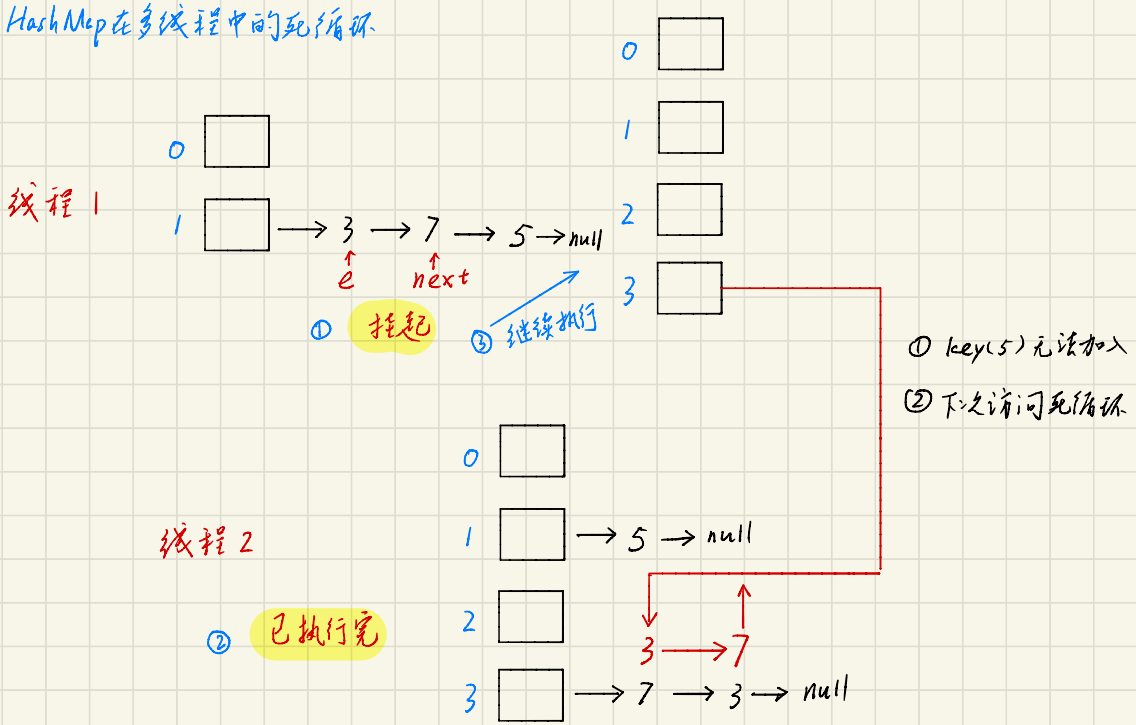
元素存储的 Index 是用 hash (mod) capacity 计算得到

↙ 消耗 >> 位运算

hash & 2^N-1

输入的 capacity 会被转化为最小的 2^

HashMap在多线程中的死循环



线程安全解决方案

- 单线程
  - 删除 → HashMap / Iterator（独立使用）
  - 增加 → HashMap
- 多线程 → Collections.sychronizedMap

多线程编程

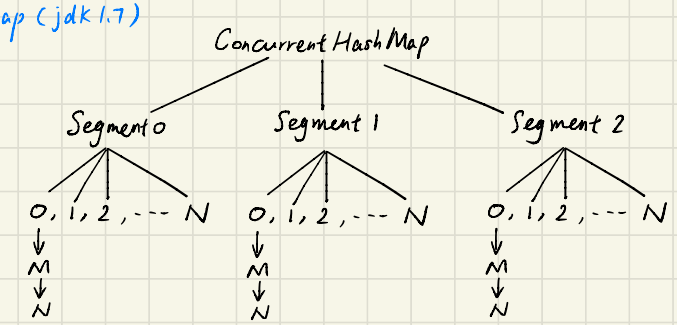
原子性（互斥性）
- 锁 lock 轻量锁（几行）
- 同步方法或同步代码块 sychronized 重量锁（代码块）

i++ 非原子操作，CPU级别的 CAS

可见性 volatile 关键字

顺序性
- volatile，lock，sychronized
- JVM的 happens-before，当两个操作可以通过该原则隐式推导出来

Concurrent Hash Map（jdk 1.7）



同步方式
- 读：获取 key 所在 Segment 时
  - lock
  - volatile
  - UNSAFE.getObjectVolatile
  - 开销太大
  - ConcurrentHashMap
- 写：获取 <key, value> 所在 Segment 的锁
  - lock ✗ 失败会挂起
  - 自旋锁 ✓

```
for (int i=0; i< retry)
{
  tryLock 操作
}
```

key 链表头修改重置

占用CPU资源较多，自旋次数超过阈值采用互斥锁

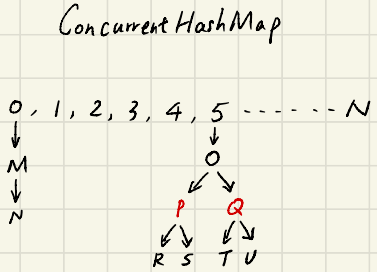
Size 操作

每个 Segment 计算 3 次 Size（比较 modCount）

↓ 有更新

加锁计算

Concurrent Hash Map (jdk 1.8)



同步就
- put
  - key对应的数组为 null ⟶ CAS设当前值
  - !null ⟶ sychronized 申请锁
- get
  - 数组被 volatile 修饰
  - 每个元素是 Node，而非 HashEntry (1.7)