HashMap

- 概念：以key-value键值对的形式存储数据的集合，允许key为null，非线程安全
- 数据结构：数组 + 链表 + 红黑树，数组初始容量16，加载因子0.75，链表树化长度为8，反树化长度为6

put流程

- 计算key的**哈希值**，与数组长度（n - 1) 进行**与运算**得到key在数组中的位置
- 如果数组索引位置没有值，则创建新的节点，放到数组对应下标
- 如果有值，且key相同，则将旧值覆盖
- 如果是树，则根据树添加元素的方式添加数据
- 如果是链表，则在链表尾部添加数据，当链表长度超过8，进行树化操作
- 集合元素数量+1，如果超过扩容阈值，则进行**扩容**操作

哈希算法

- 如果key为null，哈希值为0
- 将hashcode右移16位与自身进行异或计算

> 为什么高位和低位做异或运算?
>
> 答：让高位也参与hash寻址计算，降低哈希冲突的概率

```
// 若key为null，哈希值为0;
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

哈希寻址

- 将哈希值与数组长度-1进行与计算

> 按位与运算比取余算法效率高

```
i = (n - 1) & hash
```

自动扩容

当集合【容量 > 数组长度 * 扩容因子】进行扩容操作

- 创建一个数组容量为原来2倍的新数组

```
newCap = oldCap << 1
```

- 遍历旧数组元素，进行数据迁移

- 如果数组某个位置只有1个元素，不存在哈希冲突，`hash & (newCap - 1)`，得到元素在新数组的位置，直接拷贝过去

```
if (e.next == null)
    newTab[e.hash & (newCap - 1)] = e;
```

- 如果数组位置挂的是一颗树，则以树的方式进行扩容

  将树中的数据使用高低位链表进行拆分，如果拆分后链表长度小于6，将树转成链表；链表长度仍大于8，则将新链表转成树

  > 低位位置 = 原数组的索引位置
  >
  > 高位位置 = 旧数组索引 + 旧数组长度

- 如果数组位置挂的是一个链表，则以链表方式进行扩容，采用**高低位链表**进行辅助扩容

  > 什么是高低位链表?
  >
  > 采用 e.hash & oldCap 来判断是落在高位还是低位
  >
  > 低位位置 = 原数组的索引位置
  >
  > 高位位置 = 旧数组索引 + 旧数组长度

```
// 如果哈希值与旧数组容量进行与运算 = 0，则该节点用低位链表存储，否则用高位链表
if ((e.hash & oldCap) == 0) {
    if (loTail == null)
        loHead = e;
    else
        loTail.next = e;
    loTail = e;
} else {
    if (hiTail == null)
        hiHead = e;
    else
        hiTail.next = e;
    hiTail = e;
}
// 低位链表在数组中的位置和旧数组一致
newTab[j] = loHead;
// 高位链表在数组中的位置 = 旧数组的位置 + 长度
newTab[j + oldCap] = hiHead;
```

> 为什么使用红黑树，而不是平衡二叉树?
>
> - 红黑树是二叉查找树，左小右大，根据这个规则可以快速查找数据，时间复杂度为O(logn)
> - 普通的二叉查找树，有可能变成瘸子，不平衡，导致查询性能变成O(n)，线性查询
> - 红黑树和平衡二叉树相比，在于不追求绝对的平衡。平衡二叉树适合读多写少的场景，而HashMap读写操作都比较频繁，因此选择红黑树算是在读写性能上的一种折中

get流程

- 计算key的哈希值，并与数组长度-1进行与运算，得到数组的索引
- 如果数组位置元素与key相等，直接返回数组位置元素

- 如果数组位置节点是树节点，则遍历树查找
- 如果数组位置节点是链表节点，则遍历链表查找

remove流程

> 删除元素**不缩容**

- 和get流程一样，先查找元素
- 找到元素后，如果是树节点，将节点从树中删除
- 如果是链表节点，用链表删除的方式删除元素
- 数组长度减1

size流程

- HashMap维护一个size变量，统计集合中元素数量，当添加一个元素时size+1，删除一个元素时size-1
- 调用size()方法，直接返回size大小，即为集合中元素的数量

ConcurrentHashMap

- 概念：并发容器，并发环境下，可以安全的读取数据，key和value都不能为null
- 数据结构：数组 + 链表 + 红黑树，数组初始容量16，加载因子0.75，链表树化长度为8，反树化长度为6

变量

sizeCtl

- -1 ： 表示正在执行初始化
- 0 ： 表示
- > 0 ： 在初始化前表示的是数组容量，初始化或扩容后是下一次扩容的阈值
- resizeStamp<<16 + (1 + nThreads) ： 表示正在扩容，高位存储扩容邮戳，低位存储扩容线程数 + 1

put流程

采用分段锁 + 无锁的思想；基于 synchronized + cas

```
int hash = spread(key.hashCode());
for (Node<K,V>[] tab = table;;){
    if (tab == null || (n = tab.length) == 0){
        tab = initTable();
    }else if ((f = tabAt(tab, i = (n - 1) & hash)) == null){
        if (casTabAt(tab, i, null,new Node<K,V>(hash, key, value, null)))
           break;
    }else if ((fh = f.hash) == MOVED){
        tab = helpTransfer(tab, f);
    }else{
        synchronized (f) {
            //
```

```
        }
    }
}
```

- 计算key的哈希值，与 数组长度-1进行与运算，得到数组位置，通过 `U.getObjectVolatile` 获取数组位置数据
- 如果数组位置没有元素，新建节点，使用CAS方式设置数组位置的元素 `U.compareAndSwapObject`
- 如果数组中有元素，且元素正在执行扩容后的数据迁移，则当前put元素的线程，协助执行数据迁移操作
- 如果数组中有元素，数据未发生迁移，则使用 synchronized 锁住数组
  - 如果数组槽位哈希值大于0，表示是链表，遍历链表，查找元素，找到则覆盖value值，若找不到，则在链表尾部追加节点
  - 如果数组槽位哈希值小于0，且节点为树节点，遍历树，将节点添加到树中
  - 若链表数组长度大于8，则会将链表转成树
- 使用CountCell进行计数

简单概括：

- 通过hashcode**高低位异或**计算，并与Integer的最大值进行**与运算**，得到一个正数的哈希值

  再将哈希值与数组长度-1进行**与运算**计算哈希桶的位置
- 如果桶中没有数据，则尝试使用CAS写入，失败则自旋保证成功
- 若当前位置节点hash值为-1，表示正在扩容，当前线程帮助扩容
- 若都不满足，则使用synchronized锁住数组中的这个桶，然后添加元素
  - 判断当前节点是链表还是树
  - 若当前节点是红黑树节点，遍历树，查找匹配的key，找到则覆盖，找不到则加到树中；
  - 若是链表节点，则遍历链表插入，若链表长度大于8，则进行**树化操作。**
- 最后计算集合元素的大小，若超过了扩容阈值，则创建新数组，大小为原来的2倍。其他线程在执行put操作时，若发现节点正在扩容，则会帮助执行**扩容操作**

哈希算法

- 哈希值高16位与低16位异或，再和 2147483647 （int的最大值）进行与运算

  与 int 最大值进行与运算作用：消除负数。可能是ConcurrentHashMap内置了 `MOVED`、`TREEBIN`、`RESERVED` 三个负数哈希值，避免与之冲突

```
(h ^ (h >>> 16)) & HASH_BITS
```

```
static final int MOVED     = -1; // hash for forwarding nodes
static final int TREEBIN   = -2; // hash for roots of trees
static final int RESERVED  = -3; // hash for transient reservations
static final int HASH_BITS = 0x7fffffff; // usable bits of normal node hash
```

哈希寻址

与HashMap相同，哈希值与数组长度-1进行与运算

```
i = (n - 1) & hash
```

自动扩容

```
while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
                   (n = tab.length) < MAXIMUM_CAPACITY){
    int rs = resizeStamp(n);
    // 正在扩容
    if (sc < 0) {
        // 判断扩容是否结束，或者扩容线程数已达到上限；break退出循环
        if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                        sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                        transferIndex <= 0)
            break;
        // 扩容还未结束，且允许扩容线程加入，通过cas，将扩容线程数量+1，
        if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
            transfer(tab, nt);
    }else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                          (rs << RESIZE_STAMP_SHIFT) + 2)){
        // 未处于扩容状态，进入扩容状态，初始化netTab数组
        // (rs << RESIZE_STAMP_SHIFT) + 2 为首个扩容线程所设置的特定值
        transfer(tab, null);
    }
}
```

- 新增元素，在执行`addCount`方法后，去校验集合中的元素是否达到扩容阈值sizeCtl；扩容时 sizeCtl为负数
- 若sizeCtl小于0，表示正在执行扩容，再判断扩容是否结束，扩容线程是否到达上限；若到达上限，则直接退出while循环，否则，加入扩容大军，辅助扩容
- 若sizeCtl大于0，表示未处于扩容状态，则进入扩容状态，初始化nextTab数组

```
// 1.分配每个线程处理的桶数量
if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE){
    stride = MIN_TRANSFER_STRIDE;
}
if (nextTab == null) { // initiating
    // 2.创建新数组，大小为原来2倍的
    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
    nextTab = nt;
}
```

  - 计算每个线程处理的桶个数，至少处理16个桶
  - 创建一个新数组，数组容量为原来的2倍
  - 新建一个 ForwardingNode 对象，hash值为MOVED(-1)，表示集合正在扩容；主要有2个作用
    - 占位：用于标识数组中该桶位置的元素是否已经迁移完毕
    - 转发：扩容期间如果遇到查询操作，遇到转发节点，会把查询操作转发到新数组上，不会阻塞查询
  - 循环遍历线程各自负责需要处理的桶，处理完一个，继续处理下一个，直到全部处理完毕
  - 同样采用高低位链表帮助数据迁移

get流程

- 根据哈希算法计算key的哈希值，并和数组长度-1进行与运算，得出在数组中的位置
- 若数组位置key与获取的key相等，则返回key对应的value值
- 遍历数组节点所在位置，查找数据，找到返回value，找不到返回null

size计算

```
// 使用 baseCount 记录元素的个数，当插入数据或删除数据时，会通过 addCount 方法更新 baseCount
private transient volatile long baseCount;
// 计数单元，在并发量比较高时，两个线程同时使用 CAS 修改 baseCount 值，失败的线程会继续执行方法的逻辑
// 使用 CounterCell 计数
@sun.misc.Contended static final class CounterCell {
    volatile long value;
    CounterCell(long x) { value = x; }
}
```

```
CounterCell[] as; long b, s;
if ((as = counterCells) != null ||
    // 使用 CAS 更新 baseCount值
    !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
    CounterCell a; long v; int m;
    boolean uncontended = true;
    // CAS更新失败，则使用 CounterCell 计数，如果 CounterCell 为空，调用 fullAddCount 方法初始化
    if (as == null || (m = as.length - 1) < 0 ||
        (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
        !(uncontended =
          U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
        fullAddCount(x, uncontended);
        return;
    }
    if (check <= 1)
        return;
    s = sumCount();
}
```

通过`CAS`设置cellsBusy字段，只有设置成功的线程才能初始化`CounterCell`数组，实现如下：

```
else if (cellsBusy == 0 && counterCells == as &&
         U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
    boolean init = false;
    try {                           // Initialize table
        if (counterCells == as) {
            CounterCell[] rs = new CounterCell[2];
            rs[h & 1] = new CounterCell(x);
            counterCells = rs;
            init = true;
        }
    } finally {
```

```
            cellsBusy = 0;
        }
        if (init)
            break;
    }
```

如果通过`CAS`设置cellsBusy字段失败的话，则继续尝试通过`CAS`修改`baseCount`字段，如果修改`baseCount`字段成功的话，就退出循环，否则继续循环插入`CounterCell`对象；

```
else if (U.compareAndSwapLong(this, BASECOUNT, v = baseCount, v + x))
    break;
```

总结：

- 通过 baeCount 和 CounterCell数组来统计元素的个数
- 当添加或删除一个元素时，使用 CAS 更新 baseCount ，并发比较高时，有2个线程竞争 baseCount，有一个失败，那么它就会去通过 CounterCell来进行计数
- 在调用 size 的时候，累加 baseCount 和 CounterCell数组中的数量，就能得到元素的总个数

CounterCell 通过注解 @Contended，防止**伪共享**的发生