## Hash冲突

[**https://www.cnblogs.com/wuchaodzxx/p/7396599.html**](https://www.cnblogs.com/wuchaodzxx/p/7396599.html)

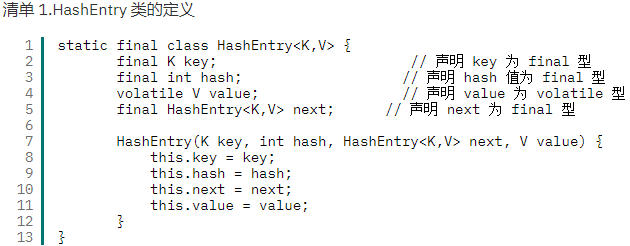
## ConcurrentHashMap（线程安全的Map）

ConcurrentHashMap不同于HashMap，**它既不允许key值为null，也不允许value值为null**

ConcurrentHashMap 类中包含两个静态内部类 HashEntry 和 Segment。HashEntry 用来封装映射表的键 / 值对；Segment 用来充当锁的角色，每个 Segment 对象守护整个散列映射表的若干个桶。每个桶是由若干个 HashEntry 对象链接起来的链表。一个 ConcurrentHashMap 实例中包含由若干个 Segment 对象组成的数组。

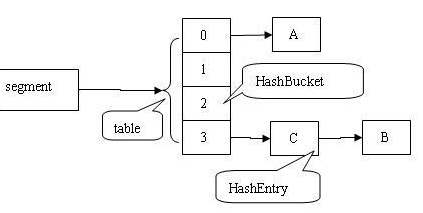
## HashEntry 类

HashEntry 用来封装散列映射表中的键值对。在 HashEntry 类中，key，hash 和 next 域都被声明为 final 型，value 域被声明为 volatile 型。



在散列时如果产生“碰撞”，将采用“分离链接法”来处理“碰撞”：把“碰撞”的 HashEntry 对象链接成一个链表。由于 HashEntry 的 next 域为 final 型，所以新节点只能在链表的表头处插入。

## Segment 类



Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。每个 Segment 对象用来守护其（成员对象 table 中）包含的若干个桶。

table 是一个由 HashEntry 对象组成的数组。table 数组的每一个数组成员就是散列映射表的一个桶。

**count 变量是一个计数器，**它表示每个 Segment 对象管理的 table 数组（若干个 HashEntry 组成的链表）包含的 HashEntry 对象的个数。每一个 Segment 对象都有一个 count 对象来表示本 Segment 中包含的 HashEntry 对象的总数。注意，之所以在每个 Segment 对象中包含一个计数器，而不是在 ConcurrentHashMap 中使用全局的计数器，是为了避免出现“热点域”而影响 ConcurrentHashMap 的并发性。

## JDK1.7中的实现

ConcurrentHashMap使用的是分段锁技术,将ConcurrentHashMap将锁一段一段的存储，然后给每一段数据配一把锁（segment），当一个线程占用一把锁（segment）访问其中一段数据的时候，其他段的数据也能被其它的线程访问，默认分配16个segment。默认比Hashtable效率提高16倍。

put实现

当执行put方法插入数据时，根据key的hash值，在Segment数组中找到相应的位置，如果相应位置的Segment还未初始化，则通过CAS进行赋值，接着执行Segment对象的put方法通过加锁机制插入数据

先采用不加锁的方式，连续计算元素的个数，最多计算3次：

1、如果前后两次计算结果相同，则说明计算出来的元素个数是准确的；

2、如果前后两次计算结果都不同，则给每个Segment进行加锁，再计算一次元素的个数；

## JDK1.8中的实现

ConcurrentHashMap取消了segment分段锁，而采用CAS和synchronized来保证并发安全。数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。

synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

size实现

1.8中使用一个volatile类型的变量baseCount记录元素的个数，当插入新数据或则删除数据时，会通过addCount()方法更新baseCount

put实现

当执行put方法插入数据时，根据key的hash值，在Node数组中找到相应的位置

1. 如果相应位置的Node还未初始化，则通过CAS插入相应的数据；
2. 如果相应位置的Node不为空，且当前该节点不处于移动状态，则对该节点加synchronized锁，如果该节点的hash不小于0，则遍历链表更新节点或插入新节点；
3. 如果该节点是TreeBin类型的节点，说明是红黑树结构，则通过putTreeVal方法往红黑树中插入节点；
4. 如果binCount不为0，说明put操作对数据产生了影响，如果当前链表的个数达到8个，则通过treeifyBin方法转化为红黑树，如果oldVal不为空，说明是一次更新操作，没有对元素个数产生影响，则直接返回旧值；
5. 如果插入的是一个新节点，则执行addCount()方法尝试更新元素个数baseCount；

## CAS

CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”

乐观锁是一种思想。CAS是这种思想的一种实现方式。

## 为什么HashMap是线程不安全的，实际会如何体现？

第一，如果多个线程同时使用put方法添加元素

假设正好存在两个put的key发生了碰撞(hash值一样)，那么根据HashMap的实现，这两个key会添加到数组的同一个位置，这样最终就会发生其中一个线程的put的数据被覆盖。

第二，如果多个线程同时检测到元素个数超过数组大小\*loadFactor

这样会发生多个线程同时对hash数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给table，也就是说其他线程的都会丢失，并且各自线程put的数据也丢失。且会引起死循环的错误。

## Hash冲突解决

1. [开放定址法](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_0)
   1. [线性探测再散列](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_0_1)
   2. [二次探测再散列](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_0_2)
   3. [伪随机探测再散列](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_0_3)
2. [再哈希法](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_1)
3. [链地址法](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_2)
4. [建立公共溢出区](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_3)
5. [优缺点](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_4)
   1. [开放散列（open hashing）/ 拉链法（针对桶链结构）](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_4_1)
   2. [封闭散列（closed hashing）/ 开放定址法](https://www.cnblogs.com/wuchaodzxx/p/7396599.html#H1_4_2)

# 开放定址法

这种方法也称再散列法，其基本思想是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。这种方法有一个通用的再散列函数形式：

Hi=（H（key）+di）% m   i=1，2，…，n

其中H（key）为哈希函数，m 为表长，di称为增量序列。增量序列的取值方式不同，相应的再散列方式也不同。主要有以下三种：

## 线性探测再散列

dii=1，2，3，…，m-1

这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

## 二次探测再散列

di=12，-12，22，-22，…，k2，-k2    ( k<=m/2 )

这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

## 伪随机探测再散列

di=伪随机数序列。

具体实现时，应建立一个伪随机数发生器，（如i=(i+p) % m），并给定一个随机数做起点。

例如，已知哈希表长度m=11，哈希函数为：H（key）= key  %  11，则H（47）=3，H（26）=4，H（60）=5，假设下一个关键字为69，则H（69）=3，与47冲突。

如果用线性探测再散列处理冲突，下一个哈希地址为H1=（3 + 1）% 11 = 4，仍然冲突，再找下一个哈希地址为H2=（3 + 2）% 11 = 5，还是冲突，继续找下一个哈希地址为H3=（3 + 3）% 11 = 6，此时不再冲突，将69填入5号单元。

如果用二次探测再散列处理冲突，下一个哈希地址为H1=（3 + 12）% 11 = 4，仍然冲突，再找下一个哈希地址为H2=（3 - 12）% 11 = 2，此时不再冲突，将69填入2号单元。

如果用伪随机探测再散列处理冲突，且伪随机数序列为：2，5，9，……..，则下一个哈希地址为H1=（3 + 2）% 11 = 5，仍然冲突，再找下一个哈希地址为H2=（3 + 5）% 11 = 8，此时不再冲突，将69填入8号单元。

# 再哈希法

这种方法是同时构造多个不同的哈希函数：

Hi=RH1（key）  i=1，2，…，k

当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

# 链地址法

这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

# 建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

# 优缺点

## 开放散列（open hashing）/ 拉链法（针对桶链结构）

1）优点： ①对于记录总数频繁可变的情况，处理的比较好（也就是避免了动态调整的开销） ②由于记录存储在结点中，而结点是动态分配，不会造成内存的浪费，所以尤其适合那种记录本身尺寸（size）很大的情况，因为此时指针的开销可以忽略不计了 ③删除记录时，比较方便，直接通过指针操作即可

2）缺点： ①存储的记录是随机分布在内存中的，这样在查询记录时，相比结构紧凑的数据类型（比如数组），哈希表的跳转访问会带来额外的时间开销 ②如果所有的 key-value 对是可以提前预知，并之后不会发生变化时（即不允许插入和删除），可以人为创建一个不会产生冲突的完美哈希函数（perfect hash function），此时封闭散列的性能将远高于开放散列 ③由于使用指针，记录不容易进行序列化（serialize）操作

## 封闭散列（closed hashing）/ 开放定址法

1）优点： ①记录更容易进行序列化（serialize）操作 ②如果记录总数可以预知，可以创建完美哈希函数，此时处理数据的效率是非常高的

2）缺点： ①存储记录的数目不能超过桶数组的长度，如果超过就需要扩容，而扩容会导致某次操作的时间成本飙升，这在实时或者交互式应用中可能会是一个严重的缺陷 ②使用探测序列，有可能其计算的时间成本过高，导致哈希表的处理性能降低 ③由于记录是存放在桶数组中的，而桶数组必然存在空槽，所以当记录本身尺寸（size）很大并且记录总数规模很大时，空槽占用的空间会导致明显的内存浪费 ④删除记录时，比较麻烦。比如需要删除记录a，记录b是在a之后插入桶数组的，但是和记录a有冲突，是通过探测序列再次跳转找到的地址，所以如果直接删除a，a的位置变为空槽，而空槽是查询记录失败的终止条件，这样会导致记录b在a的位置重新插入数据前不可见，所以不能直接删除a，而是设置删除标记。这就需要额外的空间和操作。