HashMap与ConcurrentHashMap解析

1.HashMap为什么是线程不安全的

- resize时链表可能出现回路，导致get操作时陷入死循环
- 在使用迭代器遍历时如果其他线程修改了map 会发生fail-fast，抛出ConcurrentModificationException异常

一.HashMap(1.7)

底层数据结构：其内部采用链表数组数据结构，HashMap里面实现一个静态内部类Entry, Entry其重要的属性有key , value, next以及hash值，用该Entry定义了一个链表的数组用来存放数据， 但为了避免链表过长，查询速度变为了线性，java1.8 引入了红黑树。当链表长度大于8时就采用红黑树结构。

hashMap的存取实现：

- **put:**如果key不为null的话，首先会根据key的hashCode计算得到散列桶坐标（数组下标），如果当前数组位置存在元素，则对该链表进行遍历，通过调用key.equals方法判断是否存在相同key，如果存在则修改内容。如果不存在则采用头插入的方式在链表头部插入一个新的节点存储这个新的映射关系。如果当前数组位置没有元素，则直接将元素放在该位置上。null key总是存放在Entry[]数组的第一个元素，而且仅保留一个null key。
- **get:** 也会根据key的hashCode的值定位到散列桶，并遍历存放在该桶中的链表。如果存在的该key则返回对应的value，如果不存在则返回null。

一些元素的定义：默认初始容量一般为16（数组长度），默认负载因子一般设为0.75，负载因子loadFactor定义为：散列表的实际元素数目(n)/ 散列表的容量(m)，扩容元素阈值为capacity * loadFactor

1.hashmap底层数组长度为什么设为2的n次方?

因为对于HashMap而言，数据分布需要均匀（最好每项都只有一个元素，这样就可以直接找到），不能太紧也不能太松，太紧会导致查询速度慢，太松则浪费

空间，这时我们就想到对数组长度进行取模运算，因此在定位元素在数组中位置时，indexFor方法，该方法仅有一条语句：h&(length - 1)，当length为2的n次方时，h&(length - 1)就相当于对length取模，而且速度比直接取模快得多，这是HashMap在速度上的一个优化。除此之外，它还可以均匀分布table数据和充分利用空间。 也就是说当length = 2^n时，不同的hash值发生碰撞的概率比较小，这样就会使得数据在table数组中分布较均匀，查询速度也较快。

2.jdk1.7与jdk1.8中hashmap的区别

- 1.8中引入了红黑树的结构，当链表长度大于8时，就采用红黑树的结构
- 1.8中在扩容时不需要重新计算hash值，因为元素所在索引位置要么是在原位置，要么在原位置+oldcap（原数组长度）（由元素hash值新增的那个bit是1还是0决定，即 e.hash & oldCap值是0还是1决定），并且1.7中rehash时，旧链表迁移新链表时，如果在新表索引位置相同的话，链表元素会倒置，1.8则不会
- 当hash不均匀时即数组中元素分布不均匀时，则1.8的性能要好于1.7，因为采用红黑树结构后，get和put操作的时间复杂度为o(logn)，而链表为O(n)

二.Concurrenthashmap(1.7)

数据结构：ConcurrentHashMap采取分段锁，每把锁锁住容器中的一个Segment,则多线程访问容器里不同Segment数据时线程就不会存在竞争，从而提高并发访问效率。CorrentHashMap内部是由Segment数组和HashEntry数组组成，而Segment的结构与HashMap一样，由一个链表数组构成，包含一个HashEntry数组。每个HashEntry是一个链表结构。 ConcurrentHashMap的元素数量增加导致ConrruentHashMap需要扩容时，ConcurrentHashMap不会增加Segment的数量，而只会增加Segment中链表数组的容量大小，这样的好处是扩容过程不需要对整个ConcurrentHashMap做rehash，而只需要对Segment里面的元素做一次rehash就可以了。

初始化时：Segment的数量，默认为16，一经指定不能再改变，每个segment中数组长度为2的n次方，最小为2，（设为2的n次方都是为了利用位运算加快

hash过程)

get:get操作实现简单高效，整个过程无需加锁，除非读到空值才会加锁重读，通过一次hash定位到Segment,然后再hash定位到HashEntry,遍历链表，取出对应value值，如果value值为null,则可能key，value正在put过程中，加锁重读保证取出的value是完整的，不为null,则直接返回

get操作不需加锁的原因是由于volatile修饰count值与value,由于所有修改操作在进行结构修改时（remove,put等）都会在最后一步写入count变量，这样就可以保证在get操作能够得到最新的结构更新；对于非结构更新，节点值改变，由于value用volatile修饰，这样也能保证读取到最新的值

put: 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

remove:remove 节点时由于HashEntry中除了value不是final的，其它值都是final的，这意味着不能从hash链的中间或尾部添加或删除节点，因为这需要修改next 引用值，所有的节点的修改只能从头部开始。如果从中间删除一个节点时就需要将要删除节点的前面所有节点整个复制一遍，最后一个节点指向要删除结点的下一个结点。

size: 每个Segment都有一个count变量，是一个volatile变量。当调用size方法时，首先先尝试2次通过不锁住segment的方式统计各个Segment的count值得总和，如果两次值不同则将锁住整个ConcurrentHashMap然后进行计算。

2.jdk1.8

数据结构：

1.8的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层采用数组+链表+红黑树的存储结构。对于个数超过8(默认值)的列表采用了红黑树结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能。

size: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,元素个数保存baseCount中，部分元素的变化个数保存在CounterCell数组中, 通过累加baseCount和CounterCell数组中的数量，即可得到元素的总个数

rehash后元素位置的相对顺序不会改变