24 经典并发容器，多线程面试必备。—深入解析ConcurrentHashMap

更新时间：2019-11-19 10:04:41

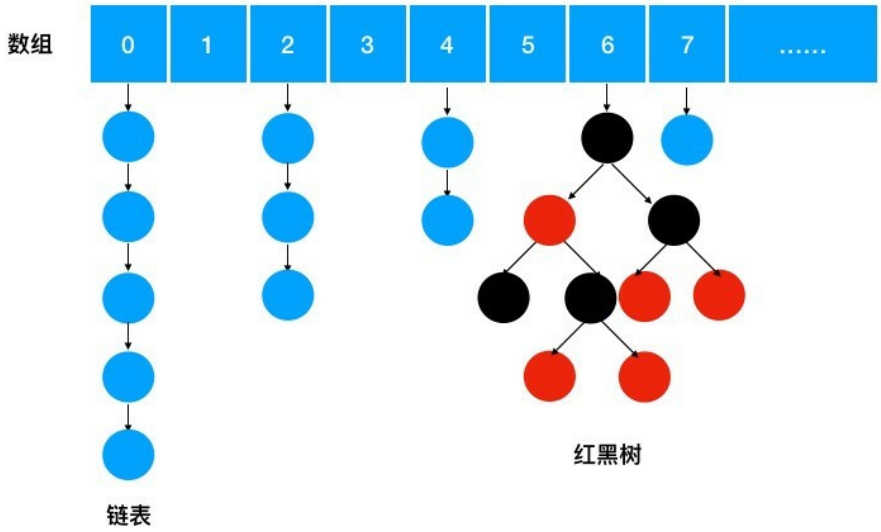
> 勤能补拙是良训，一分辛劳一分才。
>
> ——华罗庚

本小节我们将学习一个经典的并发容器 ConcurrentHashMap，它在技术面试中出现的频率相当之高，所以我们必须对它深入理解和掌握。

谈到 ConcurrentHashMap，就一定会想到 HashMap。HashMap 在我们的代码中使用频率更高，不需要考虑线程安全的地方，我们一般都会使用 HashMap。HashMap 的实现非常经典，如果你读过 HashMap 的源代码，那么对 ConcurrentHashMap 源代码的理解会相对轻松，因为两者采用的数据结构是类似的。不过即使没读过HashMap 源代码，也不影响本节的学习。

1、ConcurrentHashMap 原理概述

ConcurrentHashMap 是一个存储 key/value 对的容器，并且是线程安全的。我们先看 ConcurrentHashMap 的存储结构，如下图：



这是经典的数组加链表的形式。并且在链表长度过长时转化为红黑树存储（ Java 8 的优化），加快查找速度。

存储结构定义了容器的“形状”，那容器内的东西按照什么规则来放呢？换句话讲，某个 key 是按照什么逻辑放入容器的对应位置呢？

我们假设要存入的 key 为对象 x，这个过程如下：

1、通过对象 x 的 hashCode() 方法获取其 hashCode；

2、将 hashCode 映射到数组的某个位置上；

3、把该元素存储到该位置的链表中。

从容器取数的逻辑如下：

1、通过对象 x 的 hashCode() 方法获取其 hashCode；

2、将 hashCode 映射到数组的某个位置上；

3、遍历该位置的链表或者从红黑树中找到匹配的对象返回。

这个数组+链表的存储结构其实是一个哈希表。把对象 x 映射到数组的某个位置的函数，叫做 hash函数。这个函数的好坏决定元素在哈希表中分布是否均匀，如果元素都堆积在一个位置上，那么在取值时需要遍历很长的链表。但元素如果是均匀的分布在数组中，那么链表就会较短，通过哈希函数定位位置后，能够快速找到对应的元素。具体 ConcurrentHashMap 中的哈希函数如何实现我们后面会详细讲到。

扩容

我们大致了解了ConcurrentHashMap 的存储结构，那么我们思考一个问题，当数组中保存的链表越来越多，那么再存储进来的元素大概率会插入到现有的链表中，而不是使用数组中剩下的空位。这样会造成数组中保存的链表越来越长，由此导致哈希表查找速度下降，从 O(1) 慢慢趋近于链表的时间复杂度 O(n/2)，这显然违背了哈希表的初衷。所以 ConcurrentHashMap 会做一个操作，称为扩容。也就是把数组长度变大，增加更多的空位出来，最终目的就是预防链表过长，这样查找的时间复杂度才会趋向于 O(1)。扩容的操作并不会在数组没有空位时才进行，因为在桶位快满时，新保存元素更大的概率会命中已经使用的位置，那么可能最后几个桶位很难被使用，而链表却越来越长了。ConcurrentHashMap 会在更合适的时机进行扩容，通常是在数组中 75% 的位置被使用时。

另外 ConcurrentHashMap 还会有链表转红黑树的操作，以提高查找的速度，红黑树时间复杂度为 O(logn)，而链表是 O(n/2)，因此只在 O(logn)<O(n/2) 时才会进行转换，也就是以 8 作为分界点。

其实以上内容和 HashMap 类似，ConcurrentHashMap 此外提供了线程安全的保证，它主要是通过 CAS 和 Synchronized 关键字来实现，我们在源码分析中再详细来看。

我们做一下总结：

1、ConcurrentHashMap 采用数组+链表+红黑树的存储结构；

2、存入的Key值通过自己的 hashCode 映射到数组的相应位置；

3、ConcurrentHashMap 为保障查询效率，在特定的时候会对数据增加长度，这个操作叫做扩容；

4、当链表长度增加到 8 时，可能会触发链表转为红黑树（数组长度如果小于 64，优先扩容，具体看后面源码分析）。

接下来，我们的源码分析就从 ConcurrentHashMap 的构成、保存元素、哈希算法、扩容、查找数据这几个方面来进行。

2、ConcurrentHashMap 的构成

2.1 重要属性

我们来看看 ConcurrentHashMap 的几个重要属性

1、transient volatile Node<K,V>[] table

这个Node数组就是ConcurrentHashMap用来存储数据的哈希表。

2、private static final int DEFAULT_CAPACITY = 16;

这是默认的初始化哈希表数组大小

3、static final int TREEIFY_THRESHOLD = 8

转化为红黑树的链表长度阈值

4、static final int MOVED = -1

这个标识位用于识别扩容时正在转移数据

5、static final int HASH_BITS = 0x7fffffff;

计算哈希值时用到的参数，用来去除符号位

6、private transient volatile Node<K,V>[] nextTable;

数据转移时，新的哈希表数组

可能有些属性看完解释你还摸不到头脑。没关系，我们在后面源码分析时，具体使用的地方还会做相应讲解。

2.2 重要组成元素

Node

链表中的元素为Node对象。他是链表上的一个节点，内部存储了key、value值，以及他的下一个节点的引用。这样一系列的Node就串成一串，组成一个链表。

ForwardingNode

当进行扩容时，要把链表迁移到新的哈希表，在做这个操作时，会在把数组中的头节点替换为ForwardingNode对象。ForwardingNode中不保存key和value，只保存了扩容后哈希表（nextTable）的引用。此时查找相应node时，需要去nextTable中查找。

TreeBin

当链表转为红黑树后，数组中保存的引用为 TreeBin，TreeBin 内部不保存 key/value，他保存了 TreeNode的list以及红黑树 root。

TreeNode

红黑树的节点。

3、put 方法源码分析

put 方法用来把一个键值对存储到map中。代码如下：

```
public V put(K key, V value) {
    return putVal(key, value, false);
}
```

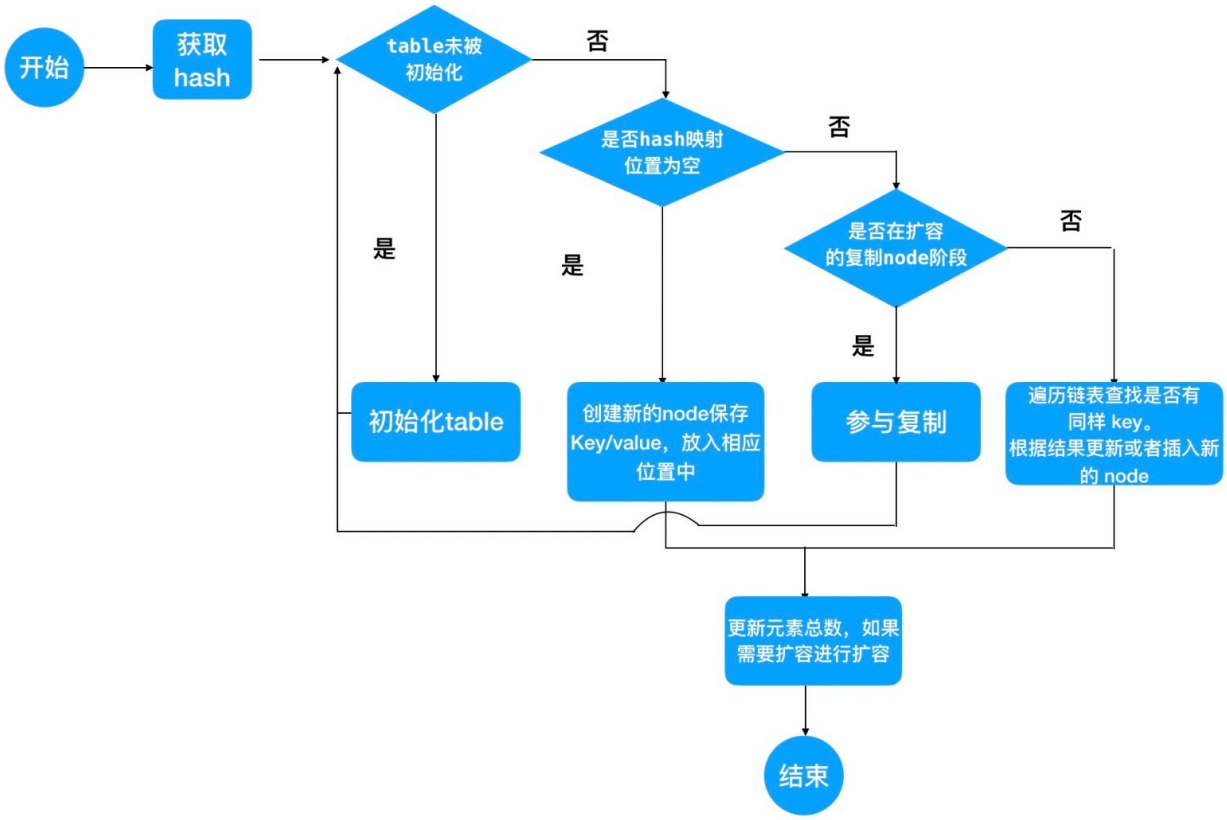
实际调用的是 putVal 方法，第三个参数传入 false，控制 key 存在时覆盖原来的值。

我们接下来看 putVal 的代码，代码比较多，我把解释直接放到代码中：

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
  //key和value不能为空
    if (key == null || value == null) throw new NullPointerException();
  //计算key的hash值，后面我们会看spread方法的实现
    int hash = spread(key.hashCode());
    int binCount = 0;
  //开始自旋，table属性采取懒加载，第一次put的时候进行初始化
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
      //如果table未被初始化，则初始化table
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
      //通过key的hash值映射table位置，如果该位置的值为空，那么生成新的node来存储该key、value，放入此位置
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null,
```

```
                    new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
            }
        //如果该位置节点元素的hash值为MOVED，也就是-1，代表正在做扩容的复制。那么该线程参与复制工作。
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);
        //下面分支处理table映射的位置已经存在node的情况
            else {
                V oldVal = null;
                synchronized (f) {
                    //再次确认该位置的值是否已经发生了变化
                    if (tabAt(tab, i) == f) {
                        //fh大于0，表示该位置存储的还是链表
                        if (fh >= 0) {
                            binCount = 1;
                            //遍历链表
                            for (Node<K,V> e = f;; ++binCount) {
                                K ek;
                                //如果存在一样hash值的node，那么根据onlyIfAbsent的值选择覆盖value或者不覆盖
                                if (e.hash == hash &&
                                    ((ek = e.key) == key ||
                                     (ek != null && key.equals(ek)))) {
                                    oldVal = e.val;
                                    if (!onlyIfAbsent)
                                        e.val = value;
                                    break;
                                }
                                Node<K,V> pred = e;
                                //如果找到最后一个元素，也没有找到相同hash的node，那么生成新的node存储key/value，作为尾节点放入链表。
                                if ((e = e.next) == null) {
                                    pred.next = new Node<K,V>(hash, key,
                                                              value, null);
                                    break;
                                }
                            }
                        }
                        //下面的逻辑处理链表已经转为红黑树时的key/value保存
                        else if (f instanceof TreeBin) {
                            Node<K,V> p;
                            binCount = 2;
                            if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                          value)) != null) {
                                oldVal = p.val;
                                if (!onlyIfAbsent)
                                    p.val = value;
                            }
                        }
                    }
                }
                //node保存完成后，判断链表长度是否已经超出阈值，则进行哈希表扩容或者将链表转化为红黑树
                if (binCount != 0) {
                    if (binCount >= TREEIFY_THRESHOLD)
                        treeifyBin(tab, i);
                    if (oldVal != null)
                        return oldVal;
                    break;
                }
            }
        }
        //计数，并且判断哈希表中使用的桶位是否超出阈值，超出的话进行扩容
        addCount(1L, binCount);
        return null;
    }
```

主线流程梳理如下图：



其实 put 的核心思想都在这里了。接下来我们分别看一下关键节点的方法源码。

4、spread 方法源码分析

哈希算法的逻辑，决定 ConcurrentHashMap 保存和读取速度。hash 算法是 hashmap 的核心算法，JDK 的实现十分巧妙，值得我们学习。

spreed 方法源代码如下：

```
static final int spread(int h) {
    return (h ^ (h >>> 16)) & HASH_BITS;
}
```

传入的参数h为 key 对象的 hashCode，spreed 方法对 hashCode 进行了加工。重新计算出 hash。我们先暂不分析这一行代码的逻辑，先继续往下看如何使用此 hash 值。

hash 值是用来映射该 key 值在哈希表中的位置。取出哈希表中该 hash 值对应位置的代码如下。

```
tabAt(tab, i = (n - 1) & hash)
```

我们先看这一行代码的逻辑，第一个参数为哈希表，第二个参数是哈希表中的数组下标。通过 (n - 1) & hash 计算下标。n 为数组长度，我们以默认大小 16 为例，那么 n-1 = 15，我们可以假设 hash 值为 100，那么 15 & 100为多少呢？& 把它左右数值转化为二进制，按位进行与操作，只有两个值都为 1 才为 1，有一个为 0 则为 0。那么我们把 15 和 100 转化为二进制来计算，java中 int 类型为 8 个字节，一共 32 个bit位。

n的值15转为二进制：

0000 0000 0000 0000 0000 0000 0000 1111

hash的值100转为二进制：

0000 0000 0000 0000 0000 0000 0110 0100。

计算结果：

0000 0000 0000 0000 0000 0000 0000 0100

对应的十进制值为 4

是不是已经看出点什么了？15的二进制高位都为0，低位都是1。那么经过&计算后，hash值100的高位全部被清零，低位则保持不变，并且一定是小于（n-1）的。也就是说经过如此计算，通过hash值得到的数组下标绝对不会越界。

这里我提出两个问题：

1、数组大小可以为 17，或者 18 吗？

2、如果为了保证不越界为什么不直接用 % 计算取余数？

3、为什么不直接用 key 的 hashCode，而是使用经 spreed 方法加工后的 hash 值？

这几个问题是面试经常会问到的相关问题。我们一个个来解答。

4.1 数组大小必须为 2 的 n 次方

第一个问题的答案是数组大小必须为 2 的 n 次方，也就是 16、32、64....不能为其他值。因为如果不是 2 的 n 次方，那么经过计算的数组下标会增大碰撞的几率，例如数组长度为 21，那么 n-1=20，对应的二进制为：

10100

那么hash值的二进制如果是　10000（十进制16）、10010（十进制18）、10001（十进制17），和10100做&计算后，都是10000，也就是都被映射到数组16这个下标上。这三个值会以链表的形式存储在数组16下标的位置。这显然不是我们想要的结果。

但如果数组长度n为2的n次方，2进制的数值为10，100，1000，10000……n-1后对应二进制为
1，11，111，1111……这样和hash值低位&后，会保留原来hash值的低位数值，那么只要hash值的低位不一样，就不会发生碰撞。

其实如果数组长度为 2 的 n 次方，那么 (n - 1) & hash 等价于 hash%n。那么为什么不直接用hash%n呢？这是因为按位的操作效率会更高，经过我本地测试，& 计算速度大概是 % 操作的 50 倍左右。

所以 JDK 为了性能，而使用这种巧妙的算法，在确保元素均匀分布的同时，还保证了效率。

4.2 为什么不直接用 key 的 hashCode？

本来我们要分析 spreed 方法的代码，但是现在看起来这个方法好像并没有什么用处，直接用 key 的 hashCode来定位哈希表的位置就可以了啊，为什么还要经过 spreed 方法的加工呢？其实说到底还是为了减少碰撞的概率。我们先看看 spreed 方法中的代码做了什么事情：

1、h ^ (h >>> 16)

h >>> 16 的意思是把 h 的二进制数值向右移动 16 位。我们知道整形为 32 位，那么右移 16 位后，就是把高 16 位移到了低 16 位。而高 16 位清0了。

^为异或操作，二进制按位比较，如果相同则为 0，不同则为 1。这行代码的意思就是把高低16位做异或。如果两个hashCode值的低16位相同，但是高位不同，经过如此计算，低16位会变得不一样了。为什么要把低位变得不一样呢？这是由于哈希表数组长度n会是偏小的数值，那么进行 (n - 1) & hash 运算时，一直使用的是hash较低位的值。那么即使hash值不同，但如果低位相当，也会发生碰撞。而进行h ^ (h >>> 16)加工后的hash值，让hashCode高位的值也参与了哈希运算，因此减少了碰撞的概率。

2、(h ^ (h >>> 16)) & HASH_BITS

我们再看完整的代码，为何高位移到低位和原来低位做异或操作后，还需要和HASH_BITS这个常量做 & 计算呢？HASH_BITS 这个常量的值为 0x7fffffff，转化为二进制为 0111 1111 1111 1111 1111 1111 1111 1111。这个操作后会把最高位转为 0，其实就是消除了符号位，得到的都是正数。这是因为负的 hashCode 在 ConcurrentHashMap 中有特殊的含义，因此我们需要得到一个正的 hashCode。

总结

通过以上分析我们已经清楚 ConcurrentHashMap 中是如何通过 Hash 值来映射数组位置的，这里面的算法设计确实十分的巧妙。可能平时我们编码用不到，但也要熟记于心，相信在面试中一定用得到。下面一节我们再来看看 table 初始化以及扩容的相关内容，put 方法的主要内容就是这些，下节最后再看一下 get 方法。

}

23 按需上锁—ReadWriteLock详解

25 不让我进门，我就在门口一直等！—BlockingQueue和ArrayBlockingQueue