# ConcurrentHashMap源码（jdk1.6.0\_13）

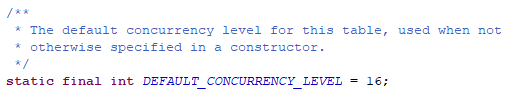
我们主要从两个部分来分析一下ConcurrentHashMap。第一部分ConcurrentHashMap的源码，第二部分ConcurrentHashMap是怎么实现并发的。

# ConcurrentHashMap源码

## 1.1常量

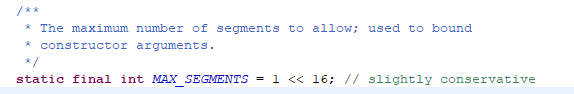
其中DEFAULT\_INITIAL\_CAPACITY，DEFAULT\_LOAD\_FACTOR，MAXIMUM\_CAPACITY，与hashmap相同，不做过多介绍。接下来看看新增的几个常量。

### 1.1.1默认并发级别DEFAULT\_CONCURRENCY\_LEVEL

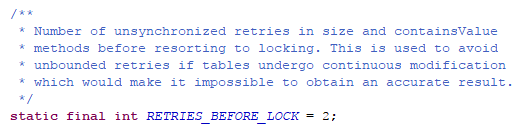


默认的并发级别，意味着同时可以有16个线程并发访问。

### 1.1.2允许的最大分段数MAX\_SEGMENTS



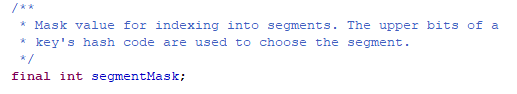
### 1.2.3采用lock之前重试次数



这个常量在size与containsValue方法中用到了，在讲size方法时在看这个常量。

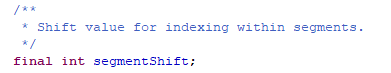
## 1.2属性

### 1.2.1分段掩码segmentMask



这个属性用来确定当前key在分段的索引位置。

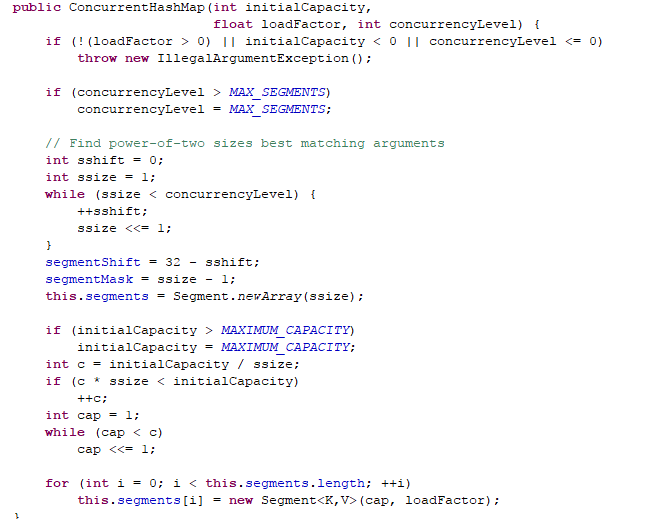
### 1.2.2移位



这个属性被用来移位操作。

## 1.3默认构造器



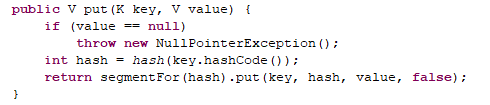


默认构造器调用了ConcurrentHashMap(int, float, int)这个构造器，在这个构造器中主要做了一些初始化操作，segmentShift，segmentMask与segments的初始化。

可以看到segments的长度始终为最接近的大于并发级别的2的幂数，segmentShift为32-幂数，segmentMask为segments长度-1。这样设计的目的在讲get方法时候在说。

## 1.4基本方法

### 1.4.1put



可以看到ConcurrentHashMap中key，value都不能为空。这里调用了segmentFor方法查找到key在哪个分段，并调用put方法将key-value存储到该分段中。我们来看看ConcurrentHashMap是怎样找到key对应的segment的。

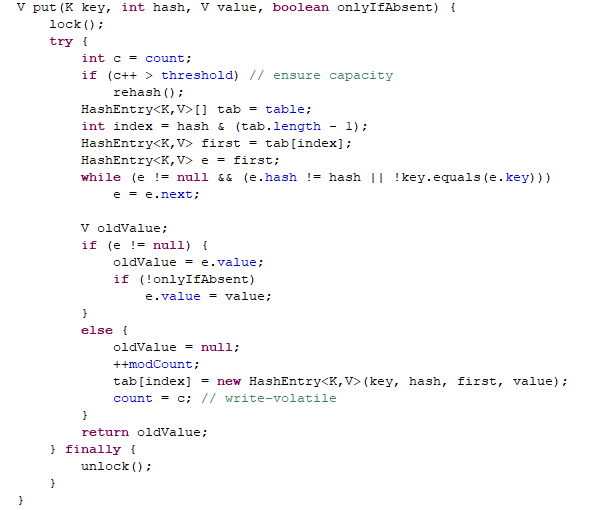


在上一节默认构造器中提到了segmentShift的值为32-logl（以2为底），l=segments.length，

这里使用32-是因为l为int类型，int类型是32位。segmentMask为l-1。这里hash取了高几位与上segmentMask，这样得到的结果范围为[0,l)，正好分布在segments数组中。例如：

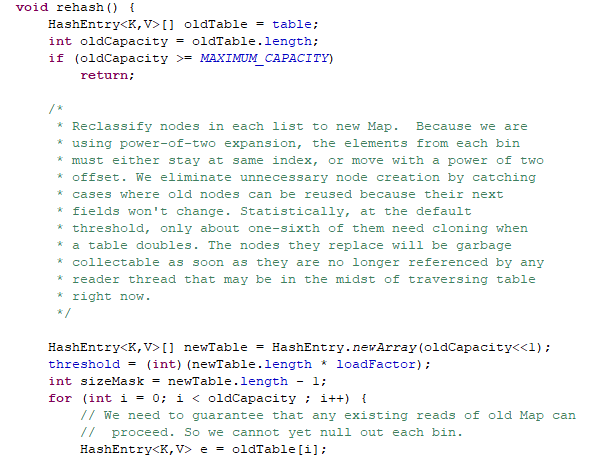
并发数取默认值16，l=16，segmentShift=28，segmentMask=1111，右移28位取了hash的高四位&1111=高四位。

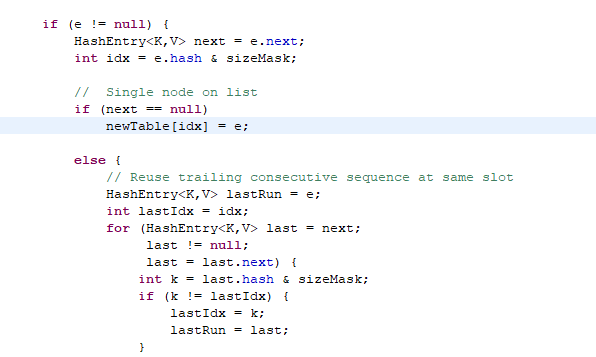
找到对应的segment调用put方法。

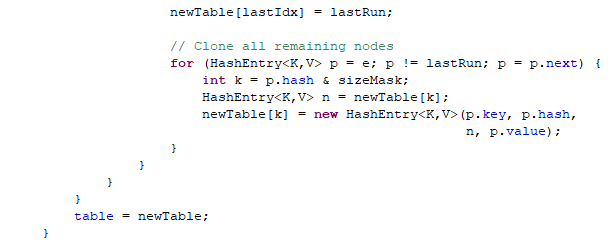


首先判断count++是否超出了容量，超出就rehash，下一节讲rehash方法。之后的操作就跟HashMap差不多了，遍历整个链表，找到就更新value，否则将key-value添加到链表中。注意onlyIfAbsent这个参数，当为true时，不替换value值，为false时，替换value值。

### 1.4.2rehash

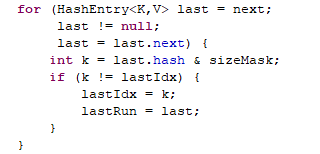




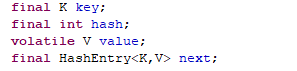


方法的说明中已经说的很清楚了，因为我们一直用2的次方来扩大数组，所以每个节点要么还在原来数组索引的位置，要么在原来索引位置加上一个2的次方。我们不必要移动没一个节点，只需要找到最后一个节点的id与它之后的节点相同的节点，然后将这个节点赋值给newTable[id]，最后遍历这个节点之前的节点，转移到新数组中。

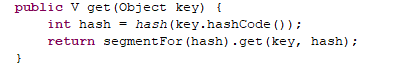
下面这段代码是用来找到最后一个节点的：



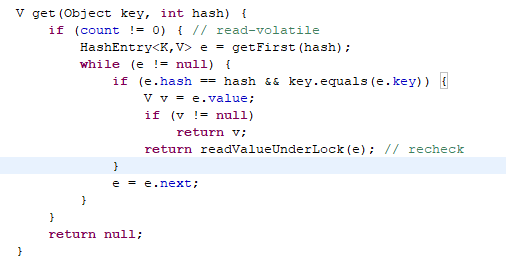
这里需要注意的是：在转移最后一个节点之前的节点时，都是重新创建一个新的节点，而不是更改next属性的指向，因为这里next属性被声明为final了，第二部分讲声明为final的原因。

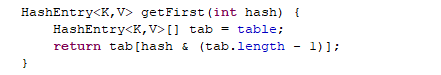


### 1.4.3get



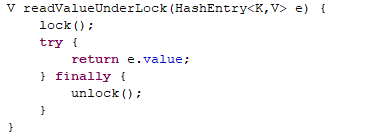
还是先获取key对应的segment然后调用Segment的get方法。





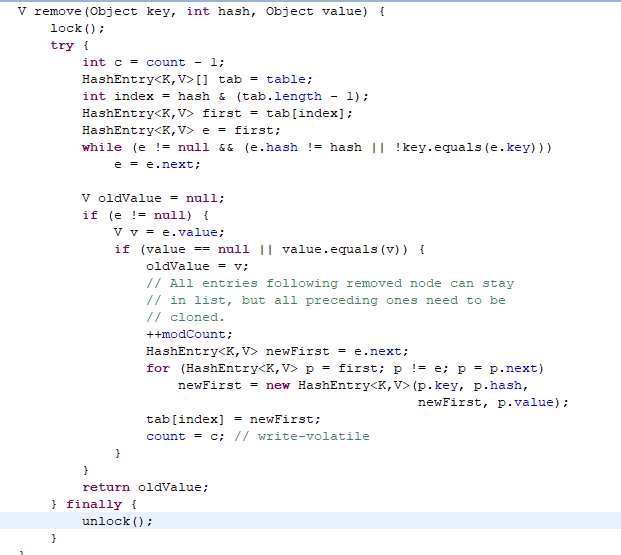
首先获取链表第一个节点，然后遍历整个链表，找到与key对应的value值，如果没有找到，返回空。

注意下面这段代码：

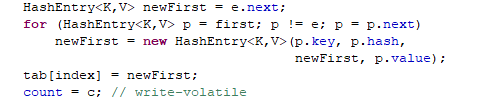


这里之所以会出现得到value为弄的情况，是因为当一个线程A正在put操作时，另一个线程B正好读到了这个key-value，但是线程A设置了key的值，value的值还没来得及设置，最终导致了线程B读到了value为弄的情况。这里使用了readValueUnderLock来重新读取value，其中使用了lock方法，获取了对象锁，两个线程不可能同时访问该对象。

### 1.4.4remove

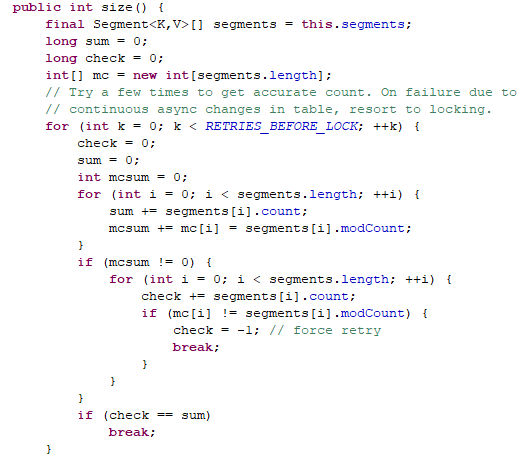


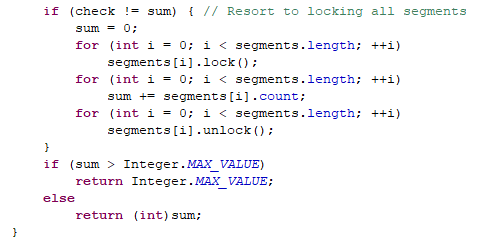
这个方法想对比较简单，这里主要讲述一下下面这段代码：



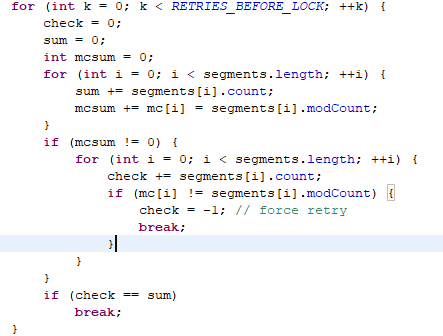
这里还是next属性的问题，因为next被声明为final了，所以只能遍历e之前的元素，然后将没个元素添加到链表的头部，e之前的元素顺序与原来相反，e之后的元素顺序没变。

### 1.4.5size



这个方法主要是两部分，第一部分是没有加锁统计，有两次机会，当两次统计机会都用完，仍然没有统计出结果时，执行第二部分的操作，加锁统计，为每一个segment加锁，统计完成之后释放锁。

先看一下第一部分：



这里统计了两次，sum为总数，check为第二次统计的总数，mcsum为第一次所有segment中modCount的总和，mc[]记录了每个segment的modCount的值。

这里统计了两次是防止有其他线程对ConcurrentHashMap做了结构上的修改，导致统计结果不准确，如果有线程对ConcurrentHashMap做了结构上的修改，则check=-1，跳出循环；如果两次统计结果相同，跳出循环，返回结果。

# ConcurrentHashMap并发

ConcurrentHashMap使用了分段锁的概念，ConcurrentHashMap内部默认设置为16个并发级别，也就是内部有16个分段。当对ConcurrentHashMap做结构上的修改的操作时，只是锁住了当前的分段，其他线程仍然可以操作其他的分段，相比HashTable锁住整个对象，减小了锁的粒度，大大的提高了并发效率。

## 2.1结构性方法

ConcurrentHashMap中结构性方法主要有get，remove，clear。这些方法都是通过加锁来保证并发的。这里不再赘述。

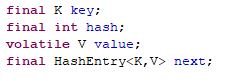
## 2.2get方法

先来看一下Segment这个内部类：



count属性被声明为volatile的，这个修饰符表示每次线程直接从count属性所在内存中直接读取值，也就是说其他线程对count属性的修改对当前线程一定是可见的。这个属性是保证get方法并发的一个重要点。

再来看一下HashEntry内部类：



这个内部类的属性key，hash与next属性都被声明为final。也就是表明HashEntry在创建之后key，hash，next都不能被修改，在添加元素时，只能添加到链表的头部，删除元素时，不能想常规链表删除节点那样操作。这个设计也是保证get方法并发的一个重要点。

当get访问元素时，如果有其他线程修改了ConcurrentHashMap的结构，HashEntry的设计保证了数据的一致性，原始的链表依然才在，才能保证get方法能一直遍历下去。