目录

[ConcurrentHashMap 1](#_Toc514407233)

[预备知识 2](#_Toc514407234)

[Hash 2](#_Toc514407235)

[位运算 2](#_Toc514407236)

[1.7中原理和实现 3](#_Toc514407237)

[ConcurrentHashMap中的数据结构 3](#_Toc514407238)

[初始化做了什么事？ 3](#_Toc514407239)

[在get和put操作中，是如何快速定位元素放在哪个位置的？ 5](#_Toc514407240)

[get（）方法 5](#_Toc514407241)

[put()方法 6](#_Toc514407242)

[扩容操作 6](#_Toc514407243)

[size方法 7](#_Toc514407244)

[弱一致性 7](#_Toc514407245)

[1.8 7](#_Toc514407246)

[更多的并发容器 7](#_Toc514407247)

[ConcurrentSkipListMap 和 ConcurrentSkipListSet 7](#_Toc514407248)

[跳表 8](#_Toc514407249)

[ConcurrentLinkedQueue 8](#_Toc514407250)

[写时复制容器 8](#_Toc514407251)

[阻塞队列 8](#_Toc514407252)

[概念、生产者消费者模式 8](#_Toc514407253)

[常用方法 8](#_Toc514407254)

[常用阻塞队列 8](#_Toc514407255)

[阻塞队列的实现原理 9](#_Toc514407256)

并发容器

# ConcurrentHashMap

Hashmap多线程会导致HashMap的Entry链表形成环形数据结构，一旦形成环形数据结构，Entry的next节点永远不为空，就会产生死循环获取Entry。

HashTable使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法，其他线程也访问HashTable的同步方法时，会进入阻塞或轮询状态。如线程1使用put进行元素添加，线程2不但不能使用put方法添加元素，也不能使用get方法来获取元素，所以竞争越激烈效率越低。

putIfAbsent() ：没有这个值则放入map，有这个值则返回key本来对应的值。

## 预备知识

### Hash

散列，哈希：把任意长度的输入通过一种算法（散列），变换成为固定长度的输出，这个输出值就是散列值。属于压缩映射，容易产生哈希冲突。Hash算法有直接取余法等。

产生哈希冲突时解决办法：开放寻址；2、再散列；3、链地址法（相同hash值的元素用链表串起来）。

ConcurrentHashMap在发生hash冲突时采用了链地址法。

md4,md5,sha-hash算法也属于hash算法，又称摘要算法。

### 位运算

**int类型的位**

**高位 低位**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1 | 0 | 1 | 0 | 0 | 0 |

2的0次方 = 1，2的1次方=2…….，以上表格代表数字 （2的5次方+2的3次方）= 40

由上面的表格可以看出，数字类型在数字渐渐变大时，是由低位慢慢向高位扩展的。

Java实际保存int型时 正数 第31位 =0 负数：第31位=1

常用位运算有：

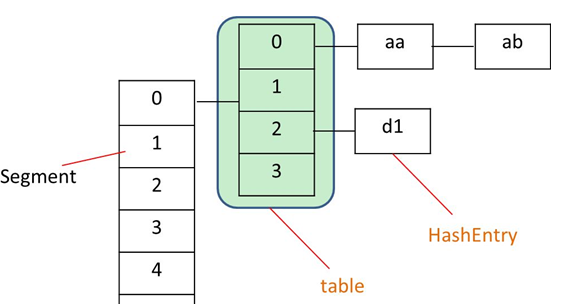
* 位与 & (1&1=1 1&0=0 0&0=0)
* 位或 | (1|1=1 1|0=1 0|0=0)
* 位非 ~ （ ~1=0 ~0=1）
* 位异或 ^ (1^1=0 1^0=1 0^0=0)
* <<有符号左移 >>有符号的右移 >>>无符号右移 例如：8 << 2 = 32 8>>2 = 2
* 取模的操作 a % (Math.pow(2,n)) 等价于 a&( Math.pow(2,n)-1)

**位运算适用**：权限控制，物品的属性非常多时的保存

## 1.7中原理和实现

### ConcurrentHashMap中的数据结构

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment实际继承自可重入锁（ReentrantLock），在ConcurrentHashMap里扮演锁的角色；HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，每个Segment里包含一个HashEntry数组，我们称之为table，每个HashEntry是一个链表结构的元素。



**面试常问：**

1. **ConcurrentHashMap实现原理是怎么样的或者问ConcurrentHashMap如何在保证高并发下线程安全的同时实现了性能提升？**

答：ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了**锁分离**技术。它使用了多个锁来控制对hash表的不同部分进行的修改。内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的hash table，只要多个修改操作发生在不同的段上，它们就可以并发进行。

### 初始化做了什么事？

初始化有三个参数

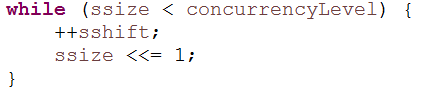
**initialCapacity**：初始容量大小 ，默认16。

**loadFactor,** 扩容因子，默认0.75，当一个Segment存储的元素数量大于initialCapacity\* loadFactor时，该Segment会进行一次扩容。

**concurrencyLevel** 并发度，默认16。并发度可以理解为程序运行时能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数，实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。如果并发度设置的过小，会带来严重的锁竞争问题；如果并发度设置的过大，原本位于同一个Segment内的访问会扩散到不同的Segment中，CPU cache命中率会下降，从而引起程序性能下降。

**构造方法中部分代码解惑：**

1、



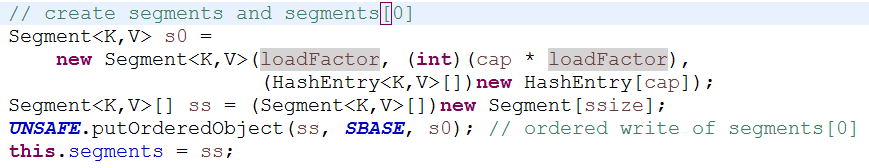
保证Segment数组的大小，一定为2的幂，例如用户设置并发度为17，则实际Segment数组大小则为32

2、



保证每个Segment中tabel数组的大小，一定为2的幂，初始化的三个参数取默认值时，table数组大小为2

3、



初始化Segment数组，并实际只填充Segment数组的第0个元素。

4、

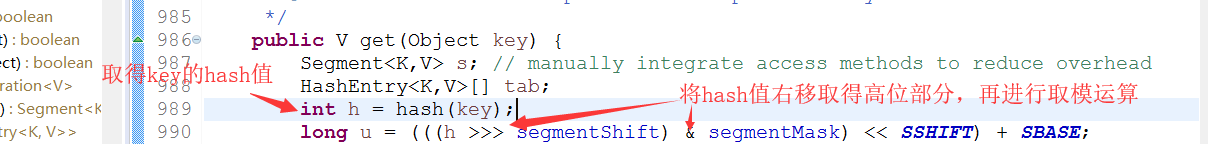


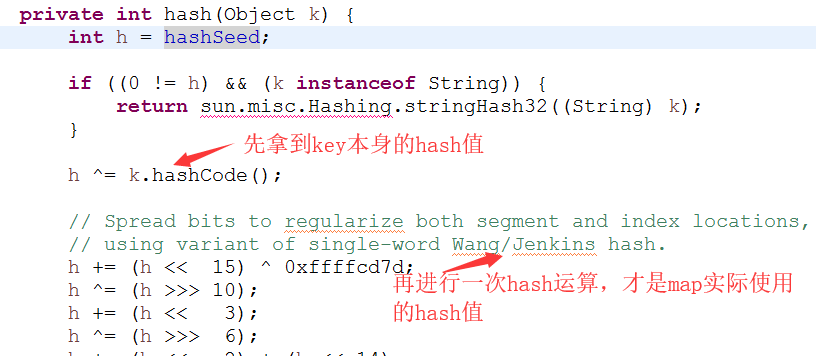
用于定位元素所在segment。segmentShift表示偏移位数，通过前面的int类型的位的描述我们可以得知，int类型的数字在变大的过程中，低位总是比高位先填满的，为保证元素在segment级别分布的尽量均匀，计算元素所在segment时，总是取hash值的高位进行计算。segmentMask作用就是为了利用位运算中取模的操作： a % (Math.pow(2,n)) 等价于 a&( Math.pow(2,n)-1)

### 在get和put操作中，是如何快速定位元素放在哪个位置的？

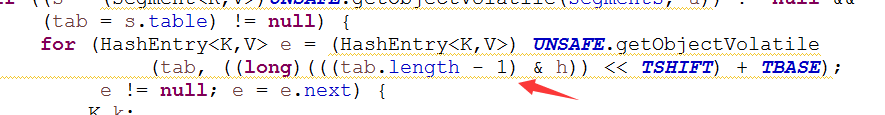
对于某个元素而言，一定是放在某个segment元素的某个table元素中的，所以在定位上，

**定位segment：**取得key的hashcode值进行一次再散列（通过Wang/Jenkins算法），拿到再散列值后，以再散列值的高位进行取模得到当前元素在哪个segment上。





定位table：同样是取得key的再散列值以后，用再散列值的全部和table的长度进行取模，得到当前元素在table的哪个元素上。

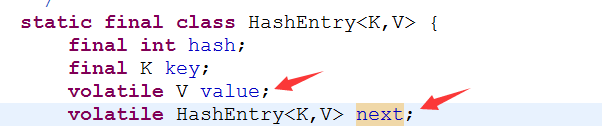


### get（）方法

定位segment和定位table后，依次扫描这个table元素下的的链表，要么找到元素，要么返回null。

**在高并发下的情况下如何保证取得的元素是最新的？**

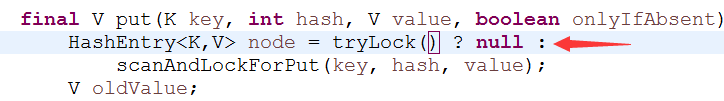
答：用于存储键值对数据的HashEntry，在设计上它的成员变量value等都是volatile类型的，这样就保证别的线程对value值的修改，get方法可以马上看到。



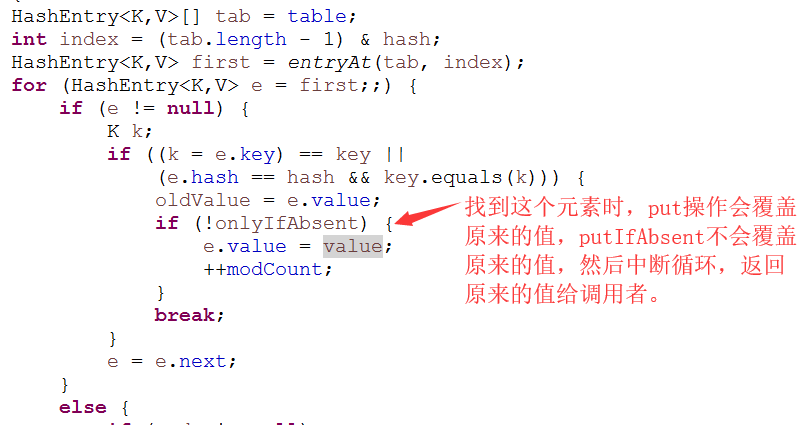
### put()方法

1、首先定位segment，当这个segment在map初始化后，还为null，由ensureSegment方法负责填充这个segment。

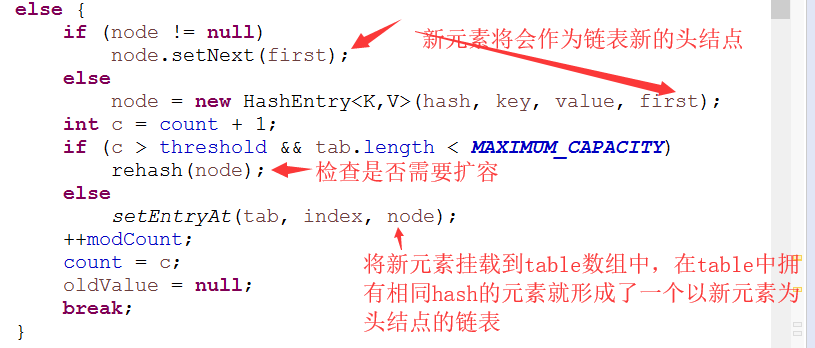
1. 对Segment 加锁



3、定位所在的table元素，并扫描table下的链表，**找到时：**

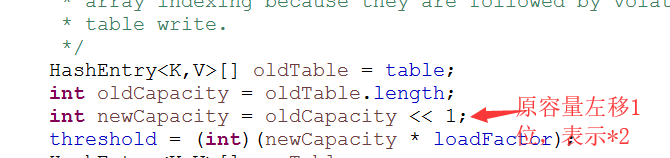


**没有找到时：**



### 扩容操作

Segment 不扩容，扩容下面的table数组，每次都是将数组翻倍



**带来的好处**

假设原来table长度为4，那么元素在table中的分布是这样的：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hash值 | 15 | 23 | 34 | 56 | 77 |
| 在table中下标 | 3 = 15%4 | 3 = 23 % 4 | 2 = 34%4 | 0 = 56%4 | 1 = 77 % 4 |

扩容后table长度变为8，那么元素在table中的分布变成：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hash值 | 56 |  | 34 |  |  | 77 |  | 15,23 |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

可以看见 hash值为34和56的下标保持不变，而15,23,77的下标都是在原来下标的基础上+4即可，可以快速定位和减少重排次数。

### size方法

size的时候进行两次不加锁的统计，两次一致直接返回结果，不一致，重新加锁再次统计

### 弱一致性

get方法和containsKey方法都是通过对链表遍历判断是否存在key相同的节点以及获得该节点的value。但由于遍历过程中其他线程可能对链表结构做了调整，因此get和containsKey返回的可能是过时的数据，这一点是ConcurrentHashMap在弱一致性上的体现。

## 1.8

# 更多的并发容器

## ConcurrentSkipListMap 和 ConcurrentSkipListSet

### 跳表

## ConcurrentLinkedQueue

## 写时复制容器

# 阻塞队列

## 概念、生产者消费者模式

## 常用方法

## 常用阻塞队列

·ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。

·LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。

·PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。

·DelayQueue：一个使用优先级队列实现的无界阻塞队列。

·SynchronousQueue：一个不存储元素的阻塞队列。

·LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

·LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

## 阻塞队列的实现原理