*摘要: 本文主要介绍ConcurrentHashMap以及ConcurrentHashMap的内部实现*

曾经在 [[高并发Java 五] JDK并发包1](http://my.oschina.net/hosee/blog/607677" \t "https://my.oschina.net/hosee/blog/_blank) 中提到过ConcurrentHashMap，只是简单的提到了下ConcurrentHashMap的优点，以及大概的实现原理。

而本文则重点介绍ConcurrentHashMap实现的细节。

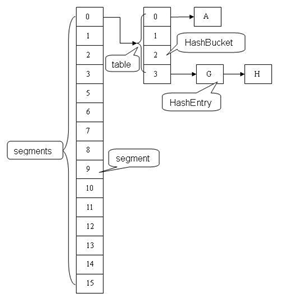
HashMap就不介绍了，具体请查看[JDK7与JDK8中HashMap的实现](http://my.oschina.net/hosee/blog/618953" \t "https://my.oschina.net/hosee/blog/_blank)

HashTable是一个线程安全的类，它使用synchronized来锁住整张Hash表来实现线程安全，即每次锁住整张表让线程独占。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。ConcurrentHashMap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Hashtable，它们有自己的锁。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。

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

## 1. 实现原理

ConcurrentHashMap使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。如下图是ConcurrentHashMap的内部结构图：



从图中可以看到，ConcurrentHashMap内部分为很多个Segment，每一个Segment拥有一把锁，然后每个Segment（继承ReentrantLock）

static final class Segment<K,V> extends ReentrantLock implements Serializable

Segment继承了ReentrantLock，表明每个segment都可以当做一个锁。（ReentrantLock前文已经提到，不了解的话就把当做synchronized的替代者吧）这样对每个segment中的数据需要同步操作的话都是使用每个segment容器对象自身的锁来实现。只有对全局需要改变时锁定的是所有的segment。

Segment下面包含很多个HashEntry列表数组。对于一个key，需要经过三次（为什么要hash三次下文会详细讲解）hash操作，才能最终定位这个元素的位置，这三次hash分别为：

1. 对于一个key，先进行一次hash操作，得到hash值h1，也即h1 = hash1(key)；
2. 将得到的h1的高几位进行第二次hash，得到hash值h2，也即h2 = hash2(h1高几位)，通过h2能够确定该元素的放在哪个Segment；
3. 将得到的h1进行第三次hash，得到hash值h3，也即h3 = hash3(h1)，通过h3能够确定该元素放置在哪个HashEntry。

ConcurrentHashMap中主要实体类就是三个：ConcurrentHashMap（整个Hash表）,Segment（桶），HashEntry（节点），对应上面的图可以看出之间的关系

/\*\*

\* The segments, each of which is a specialized hash table

\*/ final Segment<K,V>[] segments;

不变(Immutable)和易变(Volatile)ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁。如果使用传统的技术，如HashMap中的实现，如果允许可以在hash链的中间添加或删除元素，读操作不加锁将得到不一致的数据。ConcurrentHashMap实现技术是保证HashEntry几乎是不可变的。HashEntry代表每个hash链中的一个节点，其结构如下所示：

static final class HashEntry<K,V> {

final K key;

final int hash;

volatile V value;

volatile HashEntry<K,V> next;

}

在JDK 1.6中，HashEntry中的next指针也定义为final，并且每次插入将新添加节点作为链的头节点（同HashMap实现），而且每次删除一个节点时，会将删除节点之前的所有节点 拷贝一份组成一个新的链，而将当前节点的上一个节点的next指向当前节点的下一个节点，从而在删除以后 有两条链存在，因而可以保证即使在同一条链中，有一个线程在删除，而另一个线程在遍历，它们都能工作良好，因为遍历的线程能继续使用原有的链。因而这种实现是一种更加细粒度的happens-before关系，即如果遍历线程在删除线程结束后开始，则它能看到删除后的变化，如果它发生在删除线程正在执行中间，则它会使用原有的链，而不会等到删除线程结束后再执行，即看不到删除线程的影响。如果这不符合你的需求，还是乖乖的用Hashtable或HashMap的synchronized版本，Collections.synchronizedMap()做的包装。

而HashMap中的Entry只有key是final的

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

int hash;

**不变** **模式（immutable）**是多线程安全里最简单的一种保障方式。因为你拿他没有办法，想改变它也没有机会。  
不变模式主要通过final关键字来限定的。在JMM中final关键字还有特殊的语义。Final域使得确保初始化安全性（initialization safety）成为可能，初始化安全性让不可变形对象不需要同步就能自由地被访问和共享。

### 1.1 初始化

先看看ConcurrentHashMap的初始化做了哪些事情，构造函数的源码如下：

public ConcurrentHashMap(int initialCapacity,

float loadFactor, int concurrencyLevel) {

if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)

throw new IllegalArgumentException();

if (concurrencyLevel > MAX\_SEGMENTS)

concurrencyLevel = MAX\_SEGMENTS;

// Find power-of-two sizes best matching arguments

int sshift = 0;

int ssize = 1;

while (ssize < concurrencyLevel) {

++sshift;

ssize <<= 1;

}

this.segmentShift = 32 - sshift;

this.segmentMask = ssize - 1;

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

int c = initialCapacity / ssize;

if (c \* ssize < initialCapacity)

++c;

int cap = MIN\_SEGMENT\_TABLE\_CAPACITY;

while (cap < c)

cap <<= 1;

// create segments and segments[0]

Segment<K,V> s0 =

new Segment<K,V>(loadFactor, (int)(cap \* loadFactor),

(HashEntry<K,V>[])new HashEntry[cap]);

Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];

UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]

this.segments = ss;

}

传入的参数有initialCapacity，loadFactor，concurrencyLevel这三个。

* initialCapacity表示新创建的这个ConcurrentHashMap的初始容量，也就是上面的结构图中的Entry数量。默认值为static final int DEFAULT\_INITIAL\_CAPACITY = 16;
* loadFactor表示负载因子，就是当ConcurrentHashMap中的元素个数大于loadFactor \* 最大容量时就需要rehash，扩容。默认值为static final float DEFAULT\_LOAD\_FACTOR = 0.75f;
* concurrencyLevel表示并发级别，这个值用来确定Segment的个数，Segment的个数是大于等于concurrencyLevel的第一个2的n次方的数。比如，如果concurrencyLevel为12，13，14，15，16这些数，则Segment的数目为16(2的4次方)。默认值为static final int DEFAULT\_CONCURRENCY\_LEVEL = 16;。理想情况下ConcurrentHashMap的真正的并发访问量能够达到concurrencyLevel，因为有concurrencyLevel个Segment，假如有concurrencyLevel个线程需要访问Map，并且需要访问的数据都恰好分别落在不同的Segment中，则这些线程能够无竞争地自由访问（因为他们不需要竞争同一把锁），达到同时访问的效果。这也是为什么这个参数起名为“并发级别”的原因。

初始化的一些动作：

1. 验证参数的合法性，如果不合法，直接抛出异常。
2. concurrencyLevel也就是Segment的个数不能超过规定的最大Segment的个数，默认值为static final int MAX\_SEGMENTS = 1 << 16;，如果超过这个值，设置为这个值。
3. 然后使用循环找到大于等于concurrencyLevel的第一个2的n次方的数ssize，这个数就是Segment数组的大小，并记录一共向左按位移动的次数sshift，并令segmentShift = 32 - sshift，并且segmentMask的值等于ssize - 1，segmentMask的各个二进制位都为1，目的是之后可以通过key的hash值与这个值做&运算确定Segment的索引。
4. 检查给的容量值是否大于允许的最大容量值，如果大于该值，设置为该值。最大容量值为static final int MAXIMUM\_CAPACITY = 1 << 30;。
5. 然后计算每个Segment平均应该放置多少个元素，这个值c是向上取整的值。比如初始容量为15，Segment个数为4，则每个Segment平均需要放置4个元素。
6. 最后创建一个Segment实例，将其当做Segment数组的第一个元素。

### 1.2 put操作

put操作的源码如下：

public V put(K key, V value) {

Segment<K,V> s;

if (value == null)

throw new NullPointerException();

int hash = hash(key);

int j = (hash >>> segmentShift) & segmentMask;

if ((s = (Segment<K,V>)UNSAFE.getObject // nonvolatile; recheck

(segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment

s = ensureSegment(j);

return s.put(key, hash, value, false);

}

操作步骤如下：

1. 判断value是否为null，如果为null，直接抛出异常。
2. key通过一次hash运算得到一个hash值。(这个hash运算下文详说)
3. 将得到hash值向右按位移动segmentShift位，然后再与segmentMask做&运算得到segment的索引j。  
   在初始化的时候我们说过segmentShift的值等于32-sshift，例如concurrencyLevel等于16，则sshift等于4，则segmentShift为28。hash值是一个32位的整数，将其向右移动28位就变成这个样子：  
   0000 0000 0000 0000 0000 0000 0000 xxxx，然后再用这个值与segmentMask做&运算，也就是取最后四位的值。这个值确定Segment的索引。
4. 使用Unsafe的方式从Segment数组中获取该索引对应的Segment对象。
5. 向这个Segment对象中put值，这个put操作也基本是一样的步骤（通过&运算获取HashEntry的索引，然后set）。

final V put(K key, int hash, V value, boolean onlyIfAbsent) {

HashEntry<K,V> node = tryLock() ? null :

scanAndLockForPut(key, hash, value);

V oldValue;

try {

HashEntry<K,V>[] tab = table;

int index = (tab.length - 1) & hash;

HashEntry<K,V> first = entryAt(tab, index);

for (HashEntry<K,V> e = first;;) {

if (e != null) {

K k;

if ((k = e.key) == key ||

(e.hash == hash && key.equals(k))) {

oldValue = e.value;

if (!onlyIfAbsent) {

e.value = value;

++modCount;

}

break;

}

e = e.next;

}

else {

if (node != null)

node.setNext(first);

else

node = new HashEntry<K,V>(hash, key, value, first);

int c = count + 1;

if (c > threshold && tab.length < MAXIMUM\_CAPACITY)

rehash(node);

else

setEntryAt(tab, index, node);

++modCount;

count = c;

oldValue = null;

break;

}

}

} finally {

unlock();

}

return oldValue;

}

put操作是要加锁的。

### 1.3 get操作

get操作的源码如下：

public V get(Object key) {

Segment<K,V> s; // manually integrate access methods to reduce overhead

HashEntry<K,V>[] tab;

int h = hash(key);

long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;

if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&

(tab = s.table) != null) {

for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile

(tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE);

e != null; e = e.next) {

K k;

if ((k = e.key) == key || (e.hash == h && key.equals(k)))

return e.value;

}

}

return null;

}

操作步骤为：

1. 和put操作一样，先通过key进行两次hash确定应该去哪个Segment中取数据。
2. 使用Unsafe获取对应的Segment，然后再进行一次&运算得到HashEntry链表的位置，然后从链表头开始遍历整个链表（因为Hash可能会有碰撞，所以用一个链表保存），如果找到对应的key，则返回对应的value值，如果链表遍历完都没有找到对应的key，则说明Map中不包含该key，返回null。

值得注意的是，get操作是不需要加锁的（如果value为null，会调用readValueUnderLock，只有这个步骤会加锁），通过前面提到的volatile和final来确保数据安全。

### 1.4 size操作

size操作与put和get操作最大的区别在于，size操作需要遍历所有的Segment才能算出整个Map的大小，而put和get都只关心一个Segment。假设我们当前遍历的Segment为SA，那么在遍历SA过程中其他的Segment比如SB可能会被修改，于是这一次运算出来的size值可能并不是Map当前的真正大小。所以一个比较简单的办法就是计算Map大小的时候所有的Segment都Lock住，不能更新(包含put，remove等等)数据，计算完之后再Unlock。这是普通人能够想到的方案，但是牛逼的作者还有一个更好的Idea：先给3次机会，不lock所有的Segment，遍历所有Segment，累加各个Segment的大小得到整个Map的大小，如果某相邻的两次计算获取的所有Segment的更新的次数（每个Segment都有一个modCount变量，这个变量在Segment中的Entry被修改时会加一，通过这个值可以得到每个Segment的更新操作的次数）是一样的，说明计算过程中没有更新操作，则直接返回这个值。如果这三次不加锁的计算过程中Map的更新次数有变化，则之后的计算先对所有的Segment加锁，再遍历所有Segment计算Map大小，最后再解锁所有Segment。源代码如下：

public int size() {

// Try a few times to get accurate count. On failure due to

// continuous async changes in table, resort to locking.

final Segment<K,V>[] segments = this.segments;

int size;

boolean overflow; // true if size overflows 32 bits

long sum; // sum of modCounts

long last = 0L; // previous sum

int retries = -1; // first iteration isn't retry

try {

for (;;) {

if (retries++ == RETRIES\_BEFORE\_LOCK) {

for (int j = 0; j < segments.length; ++j)

ensureSegment(j).lock(); // force creation

}

sum = 0L;

size = 0;

overflow = false;

for (int j = 0; j < segments.length; ++j) {

Segment<K,V> seg = segmentAt(segments, j);

if (seg != null) {

sum += seg.modCount;

int c = seg.count;

if (c < 0 || (size += c) < 0)

overflow = true;

}

}

if (sum == last)

break;

last = sum;

}

} finally {

if (retries > RETRIES\_BEFORE\_LOCK) {

for (int j = 0; j < segments.length; ++j)

segmentAt(segments, j).unlock();

}

}

return overflow ? Integer.MAX\_VALUE : size;

}

举个例子：

一个Map有4个Segment，标记为S1，S2，S3，S4，现在我们要获取Map的size。计算过程是这样的：第一次计算，不对S1，S2，S3，S4加锁，遍历所有的Segment，假设每个Segment的大小分别为1，2，3，4，更新操作次数分别为：2，2，3，1，则这次计算可以得到Map的总大小为1+2+3+4=10，总共更新操作次数为2+2+3+1=8；第二次计算，不对S1,S2,S3,S4加锁，遍历所有Segment，假设这次每个Segment的大小变成了2，2，3，4，更新次数分别为3，2，3，1，因为两次计算得到的Map更新次数不一致(第一次是8，第二次是9)则可以断定这段时间Map数据被更新，则此时应该再试一次；第三次计算，不对S1，S2，S3，S4加锁，遍历所有Segment，假设每个Segment的更新操作次数还是为3，2，3，1，则因为第二次计算和第三次计算得到的Map的更新操作的次数是一致的，就能说明第二次计算和第三次计算这段时间内Map数据没有被更新，此时可以直接返回第三次计算得到的Map的大小。最坏的情况：第三次计算得到的数据更新次数和第二次也不一样，则只能先对所有Segment加锁再计算最后解锁。

### 1.5 containsValue操作

containsValue操作采用了和size操作一样的想法:

public boolean containsValue(Object value) {

// Same idea as size()

if (value == null)

throw new NullPointerException();

final Segment<K,V>[] segments = this.segments;

boolean found = false;

long last = 0;

int retries = -1;

try {

outer: for (;;) {

if (retries++ == RETRIES\_BEFORE\_LOCK) {

for (int j = 0; j < segments.length; ++j)

ensureSegment(j).lock(); // force creation

}

long hashSum = 0L;

int sum = 0;

for (int j = 0; j < segments.length; ++j) {

HashEntry<K,V>[] tab;

Segment<K,V> seg = segmentAt(segments, j);

if (seg != null && (tab = seg.table) != null) {

for (int i = 0 ; i < tab.length; i++) {

HashEntry<K,V> e;

for (e = entryAt(tab, i); e != null; e = e.next) {

V v = e.value;

if (v != null && value.equals(v)) {

found = true;

break outer;

}

}

}

sum += seg.modCount;

}

}

if (retries > 0 && sum == last)

break;

last = sum;

}

} finally {

if (retries > RETRIES\_BEFORE\_LOCK) {

for (int j = 0; j < segments.length; ++j)

segmentAt(segments, j).unlock();

}

}

return found;

}

## 2. 关于hash

看看hash的源代码：

private int hash(Object k) {

int h = hashSeed;

if ((0 != h) && (k instanceof String)) {

return sun.misc.Hashing.stringHash32((String) k);

}

h ^= k.hashCode();

// Spread bits to regularize both segment and index locations,

// using variant of single-word Wang/Jenkins hash.

h += (h << 15) ^ 0xffffcd7d;

h ^= (h >>> 10);

h += (h << 3);

h ^= (h >>> 6);

h += (h << 2) + (h << 14);

return h ^ (h >>> 16);

}

源码中的注释是这样的：

Applies a supplemental hash function to a given hashCode, which defends against poor quality hash functions. This is critical because ConcurrentHashMap uses power-of-two length hash tables, that otherwise encounter collisions for hashCodes that do not differ in lower or upper bits.

这里用到了Wang/Jenkins hash算法的变种，主要的目的是为了减少哈希冲突，使元素能够均匀的分布在不同的Segment上，从而提高容器的存取效率。假如哈希的质量差到极点，那么所有的元素都在一个Segment中，不仅存取元素缓慢，分段锁也会失去意义。

举个简单的例子：

System.out.println(Integer.parseInt("0001111", 2) & 15);System.out.println(Integer.parseInt("0011111", 2) & 15);System.out.println(Integer.parseInt("0111111", 2) & 15);System.out.println(Integer.parseInt("1111111", 2) & 15);

这些数字得到的hash值都是一样的，全是15，所以如果不进行第一次预hash，发生冲突的几率还是很大的，但是如果我们先把上例中的二进制数字使用hash()函数先进行一次预hash，得到的结果是这样的：

0100｜0111｜0110｜0111｜1101｜1010｜0100｜1110 1111｜0111｜0100｜0011｜0000｜0001｜1011｜1000 0111｜0111｜0110｜1001｜0100｜0110｜0011｜1110 1000｜0011｜0000｜0000｜1100｜1000｜0001｜1010

上面这个例子引用自:  [InfoQ](http://www.infoq.com/cn/articles/ConcurrentHashMap/" \t "https://my.oschina.net/hosee/blog/_blank)

可以看到每一位的数据都散开了，并且ConcurrentHashMap中是使用预hash值的高位参与运算的。比如之前说的先将hash值向右按位移动28位，再与15做&运算，得到的结果都别为：4，15，7，8，没有冲突！

## 3. 注意事项

* ConcurrentHashMap中的key和value值都不能为null，HashMap中key可以为null，HashTable中key不能为null。
* ConcurrentHashMap是线程安全的类并不能保证使用了ConcurrentHashMap的操作都是线程安全的！
* ConcurrentHashMap的get操作不需要加锁，put操作需要加锁

## Reference：

1. http://www.cnblogs.com/ITtangtang/p/3948786.html

2. http://qifuguang.me/2015/09/10/[Java%E5%B9%B6%E5%8F%91%E5%8C%85%E5%AD%A6%E4%B9%A0%E5%85%AB]%E6%B7%B1%E5%BA%A6%E5%89%96%E6%9E%90ConcurrentHashMap/

3. http://www.cnblogs.com/yydcdut/p/3959815.html