Java并发编程之ConcurrentHashMap

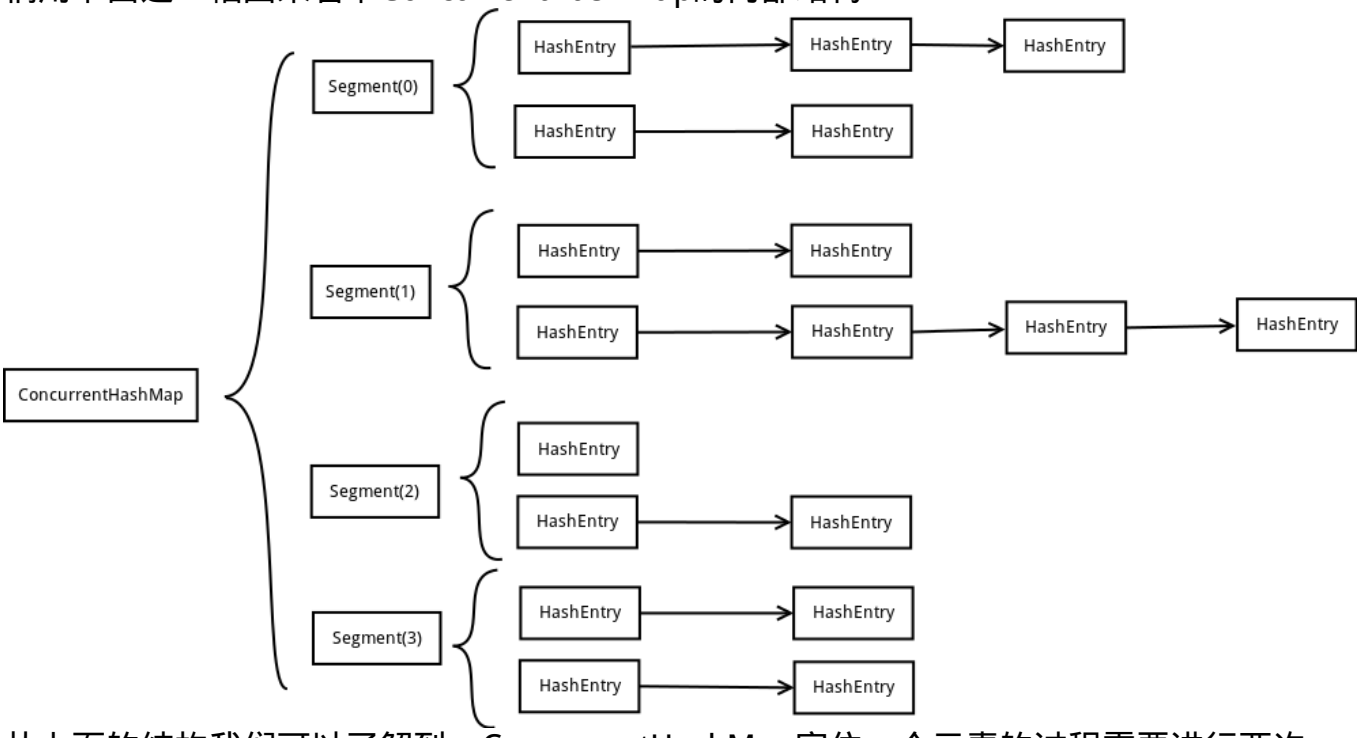
本篇文章已经发表在GoldenDoc上，欢迎大家访问GoldenDoc，在自己的博客这里仅仅做一次备份。

ConcurrentHashMap

ConcurrentHashMap是一个线程安全的Hash Table，它的主要功能是提供了一组和HashTable功能相同但是线程安全的方法。ConcurrentHashMap可以做到读取数据不加锁，并且其内部的结构可以让其在进行写操作的时候能够将锁的粒度保持地尽量地小，不用对整个ConcurrentHashMap加锁。

ConcurrentHashMap的内部结构

ConcurrentHashMap为了提高本身的并发能力，在内部采用了一个叫做Segment的结构，一个Segment其实就是一个类Hash Table的结构，Segment内部维护了一个链表数组，我们用下面这一幅图来看下ConcurrentHashMap的内部结构：



从上面的结构我们可以了解到，ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作，第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部，因此，这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长，但是带来的好处是写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作（刚好这些写操作都非常平均地分布在所有的Segment上），所以，通过这一种结构，ConcurrentHashMap的并发能力可以大大的提高。

Segment

我们再来具体了解一下Segment的数据结构：

```
static final class Segment<K,V> extends ReentrantLock imp
    transient volatile int count;
    transient int modCount;
    transient int threshold;
    transient volatile HashEntry<K,V>[] table;
    final float loadFactor;
}
```

详细解释一下Segment里面的成员变量的意义：

- count：Segment中元素的数量
- modCount：对table的大小造成影响的操作的数量（比如put或者remove操作）
- threshold：阈值，Segment里面元素的数量超过这个值依旧就会对Segment进行扩容
- table：链表数组，数组中的每一个元素代表了一个链表的头部
- loadFactor：负载因子，用于确定threshold

HashEntry

Segment中的元素是以HashEntry的形式存放在链表数组中的，看一下HashEntry的结构：

```
static final class HashEntry<K,V> {
    final K key;
    final int hash;
    volatile V value;
    final HashEntry<K,V> next;
}
```

可以看到HashEntry的一个特点，除了value以外，其他的几个变量都是final的，这样做是为了防止链表结构被破坏，出现ConcurrentModification的情况。

ConcurrentHashMap的初始化

下面我们来结合源代码来具体分析一下ConcurrentHashMap的实现，先看下初始化方法：

```
01  public ConcurrentHashMap(int initialCapacity,
02                           float loadFactor, int concurrenc
03      if (!(loadFactor > 0) || initialCapacity < 0 || concu
04          throw new IllegalArgumentException();
05
06      if (concurrencyLevel > MAX_SEGMENTS)
07          concurrencyLevel = MAX_SEGMENTS;
08
09      // Find power-of-two sizes best matching arguments
10      int sshift = 0;
11      int ssize = 1;
12      while (ssize < concurrencyLevel) {
13          ++sshift;
14          ssize <<= 1;
15      }
16      segmentShift = 32 - sshift;
17      segmentMask = ssize - 1;
18      this.segments = Segment.newArray(ssize);
19
20      if (initialCapacity > MAXIMUM_CAPACITY)
21          initialCapacity = MAXIMUM_CAPACITY;
22      int c = initialCapacity / ssize;
23      if (c * ssize < initialCapacity)
24          ++c;
25      int cap = 1;
26      while (cap < c)
27          cap <<= 1;
28
29      for (int i = 0; i < this.segments.length; ++i)
30          this.segments[i] = new Segment<K,V>(cap, loadFact
31  }
```

CurrentHashMap的初始化一共有三个参数，一个initialCapacity，表示初始的容量，一个loadFactor，表示负载参数，最后一个是concurrentLevel，代表ConcurrentHashMap内部的Segment的数量，ConcurrentLevel一经指定，不可改变，后续如果ConcurrentHashMap的元素数量增加导致ConrruentHashMap需要扩容，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。

整个ConcurrentHashMap的初始化方法还是非常简单的，先是根据concurrentLevel来new出Segment，这里Segment的数量是不大于concurrentLevel的最大的2的指数，就是说Segment的数量永远是2的指数个，这样的好处是方便采用移位操作来进行hash，加快hash的过程。接下来就是根据intialCapacity确定Segment的容量的大小，每一个Segment的容量大小也是2的指数，同样使为了加快hash的过程。

这边需要特别注意一下两个变量，分别是segmentShift和segmentMask，这两个变量在后面将会起到很大的作用，假设构造函数确定了Segment的数量是2的n次方，那么segmentShift就等于32减去n，而segmentMask就等于2的n次方减一。

ConcurrentHashMap的get操作

前面提到过ConcurrentHashMap的get操作是不用加锁的，我们这里看一下其实现：

```
1  public V get(Object key) {
2      int hash = hash(key.hashCode());
3      return segmentFor(hash).get(key, hash);
4  }
```

看第三行，segmentFor这个函数用于确定操作应该在哪一个segment中进行，几乎对ConcurrentHashMap的所有操作都需要用到这个函数，我们看下这个函数的实现：

```
1  final Segment<K,V> segmentFor(int hash) {
2      return segments[(hash >>> segmentShift) & segmentMask];
3  }
```

这个函数用了位操作来确定Segment，根据传入的hash值向右无符号右移segmentShift位，然后和segmentMask进行与操作，结合我们之前说的segmentShift和segmentMask的值，就可以得出以下结论：假设Segment的数量是2的n次方，根据元素的hash值的高n位就可以确定元素到底在哪一个Segment中。

在确定了需要在哪一个segment中进行操作以后，接下来的事情就是调用对应的Segment的get方法：

```
01  V get(Object key, int hash) {
02      if (count != 0) { // read-volatile
03          HashEntry<K,V> e = getFirst(hash);
04          while (e != null) {
05              if (e.hash == hash && key.equals(e.key)) {
06                  V v = e.value;
07                  if (v != null)
08                      return v;
09                  return readValueUnderLock(e); // recheck
10              }
11              e = e.next;
12          }
13      }
14      return null;
15  }
```

先看第二行代码，这里对count进行了一次判断，其中count表示Segment中元素的数量，我们可以来看一下count的定义：

```
1  transient volatile int count;
```

可以看到count是volatile的，实际上这里里面利用了volatile的语义：

> 对volatile字段的写入操作happens-before于每一个后续的同一个字段的读操作。

因为实际上put、remove等操作也会更新count的值，所以当竞争发生的时候，volatile的语义可以保证写操作在读操作之前，也就保证了写操作对后续的读操作都是可见的，这样后面get的后续操作就可以拿到完整的元素内容。

然后，在第三行，调用了getFirst()来取得链表的头部：

```
1  HashEntry<K,V> getFirst(int hash) {
2      HashEntry<K,V>[] tab = table;
3      return tab[hash & (tab.length - 1)];
4  }
```

同样，这里也是用位操作来确定链表的头部，hash值和HashTable的长度减一做与操作，最后的结果就是hash值的低n位，其中n是HashTable的长度以2为底的结果。

在确定了链表的头部以后，就可以对整个链表进行遍历，看第4行，取出key对应的value的值，如果拿出的value的值是null，则可能这个key，value对正在put的过程中，如果出现这种情况，那么就加锁来保证取出的value是完整的，如果不是null，则直接返回value。

ConcurrentHashMap的put操作

看完了get操作，再看下put操作，put操作的前面也是确定Segment的过程，这里不再赘述，直接看关键的segment的put方法：

```
01  V put(K key, int hash, V value, boolean onlyIfAbsent) {
02      lock();
03      try {
04          int c = count;
05          if (c++ > threshold) // ensure capacity
06              rehash();
07          HashEntry<K,V>[] tab = table;
08          int index = hash & (tab.length - 1);
09          HashEntry<K,V> first = tab[index];
10          HashEntry<K,V> e = first;
11          while (e != null && (e.hash != hash || !key.equal
12              e = e.next;
13
14          V oldValue;
15          if (e != null) {
16              oldValue = e.value;
17              if (!onlyIfAbsent)
18                  e.value = value;
19          }
20          else {
21              oldValue = null;
22              ++modCount;
23              tab[index] = new HashEntry<K,V>(key, hash, fi
24              count = c; // write-volatile
25          }
26          return oldValue;
27      } finally {
28          unlock();
29      }
30  }
```

首先对Segment的put操作是加锁完成的，然后在第五行，如果Segment中元素的数量超过了阈值（由构造函数中的loadFactor算出）这需要进行对Segment扩容，并且要进行

rehash，关于rehash的过程大家可以自己去了解，这里不详细讲了。

第8和第9行的操作就是getFirst的过程，确定链表头部的位置。

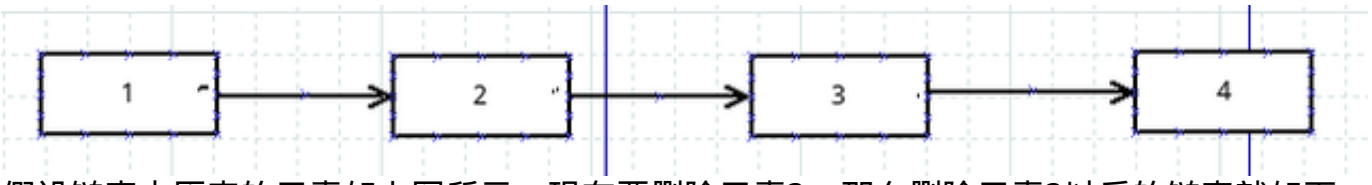
第11行这里的这个while循环是在链表中寻找和要put的元素相同key的元素，如果找到，就直接更新更新key的value，如果没有找到，则进入21行这里，生成一个新的HashEntry并且把它加到整个Segment的头部，然后再更新count的值。

ConcurrentHashMap的remove操作

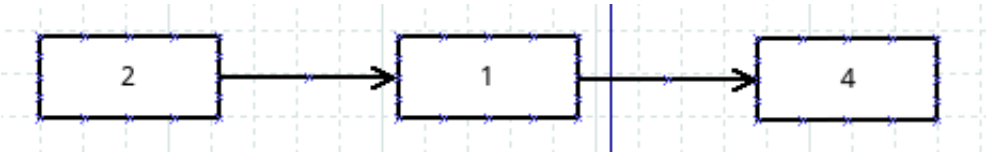
Remove操作的前面一部分和前面的get和put操作一样，都是定位Segment的过程，然后再调用Segment的remove方法：

```
01  V remove(Object key, int hash, Object value) {
02      lock();
03      try {
04          int c = count - 1;
05          HashEntry<K,V>[] tab = table;
06          int index = hash & (tab.length - 1);
07          HashEntry<K,V> first = tab[index];
08          HashEntry<K,V> e = first;
09          while (e != null && (e.hash != hash || !key.equal
10              e = e.next;
11
12          V oldValue = null;
13          if (e != null) {
14              V v = e.value;
15              if (value == null || value.equals(v)) {
16                  oldValue = v;
17                  // All entries following removed node can
18                  // in list, but all preceding ones need t
19                  // cloned.
20                  ++modCount;
21                  HashEntry<K,V> newFirst = e.next;
22                  for (HashEntry<K,V> p = first; p != e; p
23                      newFirst = new HashEntry<K,V>(p.key,
24                                                    newFirs
25                  tab[index] = newFirst;
26                  count = c; // write-volatile
27              }
28          }
29          return oldValue;
30      } finally {
31          unlock();
32      }
33  }
```

首先remove操作也是确定需要删除的元素的位置，不过这里删除元素的方法不是简单地把待删除元素的前面的一个元素的next指向后面一个就完事了，我们之前已经说过HashEntry中的next是final的，一经赋值以后就不可修改，在定位到待删除元素的位置以后，程序就将待删除元素前面的那一些元素全部复制一遍，然后再一个一个重新接到链表上去，看一下下面这一幅图来了解这个过程：



假设链表中原来的元素如上图所示，现在要删除元素3，那么删除元素3以后的链表就如下图所示：



ConcurrentHashMap的size操作

在前面的章节中，我们涉及到的操作都是在单个Segment中进行的，但是ConcurrentHashMap有一些操作是在多个Segment中进行，比如size操作，ConcurrentHashMap的size操作也采用了一种比较巧的方式，来尽量避免对所有的Segment都加锁。

前面我们提到了一个Segment中的有一个modCount变量，代表的是对Segment中元素的数量造成影响的操作的次数，这个值只增不减，size操作就是遍历了两次Segment，每次记录Segment的modCount值，然后将两次的modCount进行比较，如果相同，则表示期间没有发生过写入操作，就将原先遍历的结果返回，如果不相同，则把这个过程再重复做一次，如果再不相同，则就需要将所有的Segment都锁住，然后一个一个遍历了，具体的实现大家可以看ConcurrentHashMap的源码，这里就不贴了。

Concurrent, Java, 编程