深入分析ConcurrentHashMap

聊聊并发（四）深入分析ConcurrentHashMap

本文是作者原创，发表于InfoQ：http://www.infoq.com/cn/articles/ConcurrentHashMap

术语定义

术语	英文	解释
哈希算法	**hash algorithm**	是一种将任意内容的输入转换成相同长度输出的加密方式，其输出被称为哈希值。
哈希表	**hash table**	根据设定的哈希函数H(key)和处理冲突方法将一组关键字映象到一个有限的地址区间上，并以关键字在地址区间中的象作为记录在表中的存储位置，这种表称为哈希表或散列，所得存储位置称为哈希地址或散列地址。

线程不安全的HashMap

因为多线程环境下，使用Hashmap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。

如以下代码：

```
01  final HashMap<String, String> map = new HashMap<String, String>(2);
02
03          Thread t = new Thread(new Runnable() {
04
05              @Override
06
07              public void run() {
08
09                  for (int i = 0; i < 10000; i++) {
10
11                      new Thread(new Runnable() {
12
13                          @Override
14
15                          public void run() {
16
17                              map.put(UUID.randomUUID().toString(), "");
18
19                          }
20
21                      }, "ftf" + i).start();
22
```

```
23                  }
24
25              }
26
27          }, "ftf");
28
29          t.start();
30
31          t.join();
```

效率低下的HashTable容器

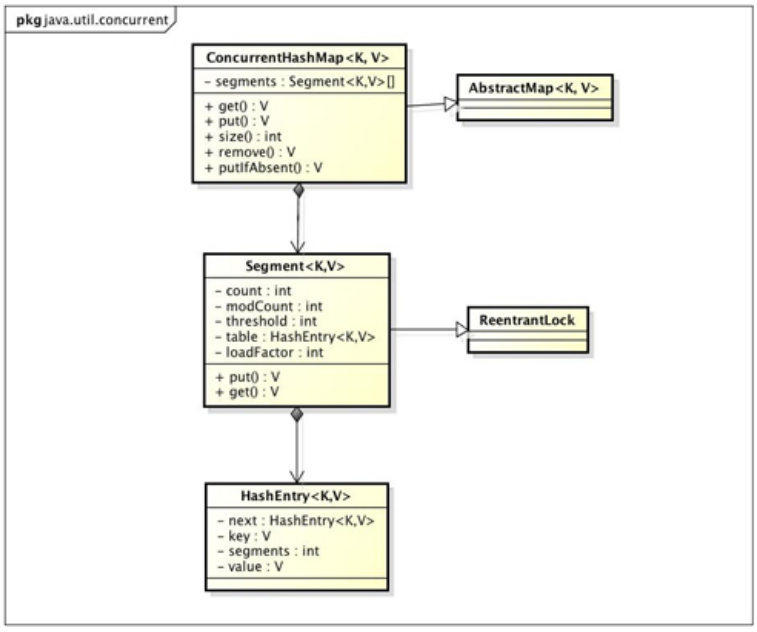
HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

ConcurrentHashMap的锁分段技术

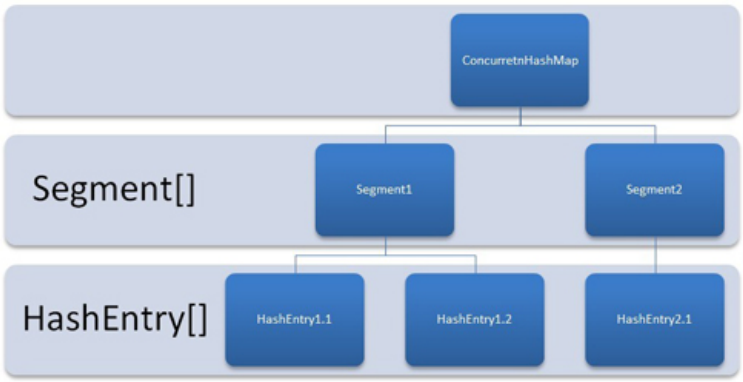
HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

ConcurrentHashMap的结构

我们通过ConcurrentHashMap的类图来分析ConcurrentHashMap的结构。



ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁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，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。



ConcurrentHashMap的初始化

ConcurrentHashMap初始化方法是通过initialCapacity，loadFactor, concurrencyLevel几个参数来初始化segments数组，段偏移量segmentShift，段掩码segmentMask和每个segment里的HashEntry数组。

初始化segments数组。让我们来看一下初始化segmentShift，segmentMask和segments数组的源代码。

```
01  if (concurrencyLevel > MAX_SEGMENTS)
02
03  concurrencyLevel = MAX_SEGMENTS;
04
05  // Find power-of-two sizes best matching arguments
06
07  int sshift = 0;
08
09  int ssize = 1;
10
11  while (ssize < concurrencyLevel) {
12
13  ++sshift;
14
15  ssize <<= 1;
16
17  }
18
19  segmentShift = 32 - sshift;
20
21  segmentMask = ssize - 1;
22
23  this.segments = Segment.newArray(ssize);
```

由上面的代码可知segments数组的长度ssize通过concurrencyLevel计算得出。为了能通过按位与的哈希算法来定位segments数组的索引，必须保证segments数组的长度是2的N次方（power-of-two size），所以必须计算出一个是大于或等于concurrencyLevel的最小的2的N次方值来作为segments数组的长度。假如concurrencyLevel等于14，15或16，ssize都会等于16，即容器里锁的个数也是16。注意

concurrencyLevel的最大大小是65535，意味着segments数组的长度最大为65536，对应的二进制是16位。

初始化segmentShift和segmentMask。这两个全局变量在定位segment时的哈希算法里需要使用，sshift等于ssize从1向左移位的次数，在默认情况下concurrencyLevel等于16，1需要向左移位移动4次，所以sshift等于4。segmentShift用于定位参与hash运算的位数，segmentShift等于32减sshift，所以等于28，这里之所以用32是因为ConcurrentHashMap里的hash()方法输出的最大数是32位的，后面的测试中我们可以看到这点。segmentMask是哈希运算的掩码，等于ssize减1，即15，掩码的二进制各个位的值都是1。因为ssize的最大长度是65536，所以segmentShift最大值是16，segmentMask最大值是65535，对应的二进制是16位，每个位都是1。

初始化每个Segment。输入参数initialCapacity是ConcurrentHashMap的初始化容量，loadfactor是每个**segment**的负载因子，在构造方法里需要通过这两个参数来初始化数组中的每个**segment**。

```
01    if (initialCapacity > MAXIMUM_CAPACITY)
02
03            initialCapacity = MAXIMUM_CAPACITY;
04
05        int c = initialCapacity / ssize;
06
07        if (c * ssize < initialCapacity)
08
09            ++c;
10
11        int cap = 1;
12
13        while (cap < c)
14
15            cap <<= 1;
16
17        for (int i = 0; i < this.segments.length; ++i)
18
19            this.segments[i] = new Segment<K,V>(cap, loadFactor);
```

上面代码中的变量cap就是segment里HashEntry数组的长度，它等于initialCapacity除以ssize的倍数c，如果c大于1，就会取大于等于c的2的N次方值，所以cap不是1，就是2的N次方。segment的容量threshold＝(int)cap*loadFactor，默认情况下initialCapacity等于16，loadfactor等于0.75，通过运算cap等于1，threshold等于零。

定位Segment

既然ConcurrentHashMap使用分段锁Segment来保护不同段的数据，那么在插入和获取元素的时候，必须先通过哈希算法定位到Segment。可以看到ConcurrentHashMap会首先使用Wang/Jenkins hash的变种算法对元素的hashCode进行一次再哈希。

```
1    private static int hash(int h) {
2
3    h += (h << 15) ^ 0xffffcd7d; h ^= (h >>> 10);
4
5    h += (h << 3); h ^= (h >>> 6);
6
7    h += (h << 2) + (h << 14); return h ^ (h >>> 16);
```

```
8
9 }
```

再哈希，其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。我做了一个测试，不通过再哈希而直接执行哈希计算。

```
1 System.out.println(Integer.parseInt("0001111", 2) & 15);
2
3 System.out.println(Integer.parseInt("0011111", 2) & 15);
4
5 System.out.println(Integer.parseInt("0111111", 2) & 15);
6
7 System.out.println(Integer.parseInt("1111111", 2) & 15);
```

计算后输出的哈希值全是15，通过这个例子可以发现如果不进行再哈希，哈希冲突会非常严重，因为只要低位一样，无论高位是什么数，其哈希值总是一样。我们再把上面的二进制数据进行再哈希后结果如下，为了方便阅读，不足32位的高位补了0，每隔四位用竖线分割下。

```
1 0100 | 0111 | 0110 | 0111 | 1101 | 1010 | 0100 | 1110
2
3 1111 | 0111 | 0100 | 0011 | 0000 | 0001 | 1011 | 1000
4
5 0111 | 0111 | 0110 | 1001 | 0100 | 0110 | 0011 | 1110
6
7 1000 | 0011 | 0000 | 0000 | 1100 | 1000 | 0001 | 1010
```

可以发现每一位的数据都散列开了，通过这种再哈希能让数字的每一位都能参加到哈希运算当中，从而减少哈希冲突。
ConcurrentHashMap通过以下哈希算法定位segment。

默认情况下segmentShift为28，segmentMask为15，再哈希后的数最大是32位二进制数据，向右无符号移动28位，意思是让高4位参与到hash运算中， (hash >>> segmentShift) & segmentMask的运算结果分别是4，15，7和8，可以看到hash值没有发生冲突。

```
1 final Segment<K,V> segmentFor(int hash) {
2
3         return segments[(hash >>> segmentShift) & segmentMask];
4
5     }
```

ConcurrentHashMap的get操作

Segment的get操作实现非常简单和高效。先经过一次再哈希，然后使用这个哈希值通过哈希运算定位到segment，再通过哈希算法定位到元素，代码如下：

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

```
6
7     }
```

get操作的高效之处在于整个get过程不需要加锁，除非读到的值是空的才会加锁重读，我们知道HashTable容器的get方法是需要加锁的，那么ConcurrentHashMap的get操作是如何做到不加锁的呢？原因是它的get方法里将要使用的共享变量都定义成volatile，如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。之所以不会读到过期的值，是根据java内存模型的happen before原则，对volatile字段的写入操作先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值，这是用volatile替换锁的经典应用场景。

```
1  transient volatile int count;
2
3  volatile V value;
```

在定位元素的代码里我们可以发现定位HashEntry和定位Segment的哈希算法虽然一样，都与数组的长度减去一相与，但是相与的值不一样，定位Segment使用的是元素的hashcode通过再哈希后得到的值的高位，而定位HashEntry直接使用的是再哈希后的值。其目的是避免两次哈希后的值一样，导致元素虽然在Segment里散列开了，但是却没有在HashEntry里散列开。

```
1  hash >>> segmentShift) & segmentMask//定位Segment所使用的hash算法
2
3  int index = hash & (tab.length - 1);// 定位HashEntry所使用的hash算法
```

ConcurrentHashMap的Put操作

由于put方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须得加锁。Put方法首先定位到Segment，然后在Segment里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置然后放在HashEntry数组里。

是否需要扩容。在插入元素前会先判断Segment里的HashEntry数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，Segment的扩容判断比HashMap更恰当，因为HashMap是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时HashMap就进行了一次无效的扩容。

如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。

ConcurrentHashMap的size操作

如果我们要统计整个ConcurrentHashMap里元素的大小，就必须统计所有Segment里元素的大小后求和。Segment里的全局变量count是一个volatile变量，那么在多线程场景下，我们是不是直接把所有Segment的count相加就可以得到整个ConcurrentHashMap大小了呢？不是的，虽然相加时可以获取每个Segment的count的最新值，但是拿到之后可能累加前使用的count发生了变化，那么统计结果就不准了。所以最安全的做法，是在统计size的时候把所有Segment的put，remove和clean方法全部锁住，但是这种做法显然非常低效。

因为在累加count操作过程中，之前累加过的count发生变化的几率非常小，所以ConcurrentHashMap的做法是先尝试2次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化，则再采用加锁的方式来统计所有Segment的大小。

那么ConcurrentHashMap是如何判断在统计的时候容器是否发生了变化呢？使用modCount变量，在put , remove和clean方法里操作元素前都会将变量modCount进行加1，那么在统计size前后比较modCount是否发生变化，从而得知容器的大小是否发生变化。

参考资料

1. JDK1.6源代码。
2. 《Java并发编程实践》
3. Java并发编程之ConcurrentHashMap