How ConcurrentHashMap Works Internally in Java

by Arun Pandey MVB · Sep. 06, 16 · Java Zone · Tutorial

Before talking in detail let us review a few concepts below:

ConcurrentHashMap: It allows concurrent access to the map. Part of the map called *Segment (internal data structure)* is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all. This data structure was introduced to improve performance.

Concurrency-Level: Defines the number which is an estimated number of concurrently updating threads. The implementation performs internal sizing to try to accommodate this many threads.

Load-Factor: It's a threshold, used to control resizing.

Initial Capacity: The implementation performs internal sizing to accommodate these many elements.

A ConcurrentHashMap is divided into number of segments, and the example which I am explaining here used default as 32 on initialization.

A ConcurrentHashMap has internal final class called Segment so we can say that ConcurrentHashMap is internally divided in segments of size 32, so at max 32 threads can work at a time. It means each thread can work on a each segment during high concurrency and atmost 32 threads can operate at max which simply maintains 32 locks to guard each bucket of the ConcurrentHashMap.

```
The definition of Segment is as below:

/** Inner Segment class plays a significant role **/
protected static final class Segment {
  protected int count;

  protected synchronized int getCount() {
    return this.count;
  }

  protected synchronized void synch() {}
}

/** Segment Array declaration **/
public final Segment[] segments = new Segment[32];
```

As we all know that Map is a kind of data structure which stores data in key-value pair which is array of inner class Entry, see as below:

```
 static class Entry implements Map.Entry {
   protected final Object key;
   protected volatile Object value;
   protected final int hash;
   protected final Entry next;

   Entry(int hash, Object key, Object value, Entry next) {

     this.value = value;
```

```
      this.hash = hash;
      this.key = key;
      this.next = next;
    }

    // Code goes here like getter/setter
  }
```

And ConcurrentHashMap class has an array defined as below of type Entry class:

```
protected transient Entry[] table;
```

This Entry array is getting initialized when we are creating an instance of ConcurrentHashMap, even using a default constructor called internally as below:

```
public ConcurrentHashMap(int initialCapacity, float loadFactor) {

  //Some code
  int cap = getCapacity();
  this.table = newTable(cap); // here this.table is Entry[] table
}

protected Entry[] newTable(int capacity) {
  this.threshold = ((int)(capacity * this.loadFactor / 32.0F) + 1);
  return new Entry[capacity];
}
```

Here, threshold is getting initialized for re-sizing purpose.

Inserting (Put) Element in ConcurrentHashMap:

Most important thing to understand the put method of ConcurrentHashMap, that how ConcurrentHashMap works when we are adding the element. As we know put method takes two arguments both of type Object as below:

```
put(Object key, Object value)
```

So it wil 1st calculate the hash of key as below:

```
int hashVal = hash(key);

static int hash(Object x) {
  int h = x.hashCode();
  return (h << 7) - h + (h >>> 9) + (h >>> 17);
}
```

After getting the hashVal we can decide the Segment as below:

```
Segment seg = segments[(hash & 0x1F)];     // segments is an array defined above
```

Since it's all about concurrency, we need synchronized block on the above Segment as below:

```
synchronized (seg) {
  // code to add

  int index = hash & table.length - 1; // hash we have calculated for key and table is Entry[] table
  Entry first = table[index];
  for (Entry e = first; e != null; e = e.next) {
    if ((e.hash == hash) && (eq(key, e.key))) { // if key already exist means updating the value
      Object oldValue = e.value;
      e.value = value;
      return oldValue;
    }
  }

  Entry newEntry = new Entry(hash, key, value, first); // new entry, i.e. this key not exist in map
  table[index] = newEntry; // Putting the Entry object at calculated Index
}
```

Size of ConcurrentHashMap

Now when we are asking for size() of the ConcurrentHashMap the size comes out as below:

```
for (int i = 0; i < this.segments.length; i++) {
  c += this.segments[i].getCount();         //here c is an integer initialized with zero
}
```

Getting (get) Element From ConcurrentHashMap

When we are getting an element from ConcurrentHashMap we are simply passing key and hash of key is getting calculated. The defintion goes something like as below:

```
public Object get(Object key){
  //some  code here

  int index = hash & table.length - 1;  //hash we have calculated for key and calculating index with he
  Entry first = table[index];          //table is Entry[] table
  for (Entry e = first; e != null; e = e.next) {
    if ((e.hash == hash) && (eq(key, e.key))) {
      Object value = e.value;
      if (value == null) {
        break;
      }
      return value;
    }
  }
  //some  code here
}
```

Note: No need to put any lock when getting the element from ConcurrentHashMap.

Removing Element From ConcurrentHashMap

Now question is how remove works with ConcurrentHashMap, so let us understand it. Remove basically takes one argument 'Key' as an argument or takes two argument 'Key' and 'Value' as below:

```
Object remove(Object key);

boolean remove(Object key, Object value);
```

Now let us understand how this works internally. The method remove (Object key) internally calls remove (Object key, Object value) where it passed 'null' as a value. Since we are going to remove an element from a Segment, we need a lock on the that Segment.

```
Object remove(Object key, Object value) {

  Segment seg = segments[(hash & 0x1F)]; //hash we have calculated for key

  synchronized (seg) {
    Entry[] tab = this.table; //table is Entry[] table
    int index = hash & tab.length - 1; //calculating index with help of hash
    Entry first = tab[index]; //Getting the Entry Object

    Entry e = first;
    while(true) {
      if ((e.hash == hash) && (eq(key, e.key))) {
        break;
      }
      e = e.next;
    }
    Object oldValue = e.value;
    Entry head = e.next;
    for (Entry p = first; p != e; p = p.next) {
      head = new Entry(p.hash, p.key, p.value, head);
    }
    table[index] = head;
    seg.count -= 1;
  }
  return oldValue;
}
```

Hope this will give you a clear understanding of the internal functionality of ConcurrentHashMap.

Topics: JAVA , CONCURRENTHASHMAP