COLLECTIONS, CORE JAVA, INTERVIEW, RELATED CONCEPTS

How hashmap works in java

OCTOBER 9, 2012 LOKESH121 COMMENTS

http://howtodoinjava.com/2012/10/09/how-hashmap-works-in-java/

Most of you will agree that HashMap is most favorite topic for discussion in interviews now-a-days. I have gone through several discussions with my colleagues time to time and it really helped. Now, I am continuing this discussion with you all.
I am assuming that if you are interested in internal working of HashMap, you already know basics of HashMap, so i am skipping that part. But if you are new to concept, follow official java docs.
Before moving forward, i will highly recommend reading my previous post : **Working with hashCode and equals methods in java**

Sections in this post

Single statement answer

If anybody asks me to describe "**How HashMap works?**", I simply answer: "**On principle of Hashing**". As simple as it is. Now before answering it, one must be very sure to know at least basics of Hashing. Right??

What is Hashing

Hashing in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/algorithm on its properties. A true Hashing function must follow this rule:
Hash function should return the same hash code each and every time, when function is applied on same or equal objects. In other words, two equal objects must produce same hash code consistently.
Note: All objects in java inherit a default implementation of hashCode() function defined in Object class. This function produce hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.
Read more here : Working with hashCode() and equals() methods in java

A little about Entry class

A map by definition is : **"An object that maps keys to values"**. Very easy.. right?
So, there must be some mechanism in HashMap to store this key value pair. Answer is YES.
HashMap has an inner class Entry, which looks like this:

```
static class Entry implements Map.Entry
{
        final K key;
        V value;
        Entry next;
        final int hash;
        ...//More code goes here
}
```

Surely Entry class has key and value mapping stored as attributes. Key has been marked as final and two more fields are there: next and hash. We will try to understand the need of these fields as we go forward.

What put() method actually does

Before going into put() method's implementation, it is very important to learn that instances of Entry class are stored in an array. HashMap class defines this variable as:

```
   /**
     * The table, resized as necessary. Length MUST Always be a
power of two.
     */
    transient Entry[] table;
```

Now look at code implementation of put() method:

```
1    /**
2         * Associates the specified value with the specified key in this map.
3         * map previously contained a mapping for the key, the old value is
4         * replaced.
5         *
6         * @param key
7         *            key with which the specified value is to be associated
8         * @param value
9         *            value to be associated with the specified key
10        * @return the previous value associated with <tt>key</tt>, or <tt>nul
11        *         if there was no mapping for <tt>key</tt>. (A <tt>null</tt>
12        *         can also indicate that the map previously associated
13        *         <tt>null</tt> with <tt>key</tt>.)
14        */
15       public V put(K key, V value) {
16           if (key == null)
17               return putForNullKey(value);
18           int hash = hash(key.hashCode());
19           int i = indexFor(hash, table.length);
20           for (Entry<k , V> e = table[i]; e != null; e = e.next) {
21               Object k;
22               if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
23                   V oldValue = e.value;
```

```
24                  e.value = value;
25                  e.recordAccess(this);
26                  return oldValue;
27              }
28          }
29
30          modCount++;
31          addEntry(hash, key, value, i);
32          return null;
33      }
```

Lets note down the steps one by one:

-- First of all, key object is checked for null. If key is null, value is stored in table[0] position. Because hash code for null is always 0.

-- Then on next step, a hash value is calculated using key's hash code by calling its hashCode() method. This hash value is used to calculate index in array for storing Entry object. JDK designers well assumed that there might be some poorly written hashCode() functions that can return very high or low hash code value. To solve this issue, they introduced another **hash()** function, and passed the object's hash code to this hash() function to bring hash value in range of array index size.
-- Now *indexFor(hash, table.length)* function is called to calculate exact index position for storing the Entry object.
-- Here comes the main part. Now, as we know that two unequal objects can have same hash code value, how two different objects will be stored in same array location [**called bucket**]. Answer is LinkedList. If you remember, Entry class had an attribute **"next"**. This attribute always points to next object in chain. This is exactly the behavior of LinkedList.
So, in case of collision, Entry objects are stored in LinkedList form. When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, Entry object is stored in this location.

If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current Entry object becomes next node in LinkedList. If next variable is not null, procedure is followed until next is evaluated as null.

What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over LinkedList on calculated index, HashMap calls equals method on key object for each Entry object. All these Entry objects in LinkedList will have similar hash code but equals() method will test for true equality. If **key.equals(k)** will be true then both keys are treated as same key object. This will cause the replacing of value object inside Entry object only. In this way, HashMap ensure the uniqueness of keys.

How get() methods works internally

Now we have got the idea, how key-value pairs are stored in HashMap. Next big question is : what happens when an object is passed in get method of HashMap? How the value object is determined?

Answer we already should know that the way key uniqueness is determined in put() method , same logic is applied in get() method also. The moment HashMap identify exact match for the key object passed as argument, it simply returns the value object stored in current Entry object.

If no match is found, get() method returns null.

Let have a look at code:

```
1    /**
2         * Returns the value to which the specified key is mapped, or {@code null}
3         * if this map contains no mapping for the key.
4         *
5         * <p>
6         * More formally, if this map contains a mapping from a key {@code k} to a
7         * value {@code v} such that {@code (key==null ? k==null :
8         * key.equals(k))}, then this method returns {@code v}; otherwise it returns
9         * {@code null}. (There can be at most one such mapping.)
10        *
11        * </p><p>
12        * A return value of {@code null} does not <i>necessarily</i> indicate that
13        * the map contains no mapping for the key; it's also possible that the map
14        * explicitly maps the key to {@code null}. The {@link #containsKey
15        * containsKey} operation may be used to distinguish these two cases.
16        *
17        * @see #put(Object, Object)
18        */
19       public V get(Object key) {
20           if (key == null)
21               return getForNullKey();
22           int hash = hash(key.hashCode());
23           for (Entry<k , V> e = table[indexFor(hash, table.length)]; e != null; e = e.n
24               Object k;
25               if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
26                   return e.value;
27           }
28           return null;
29       }
```

Above code is same as put() method till *if (e.hash == hash && ((k = e.key) == key || key.equals(k)))*, after this simply value object is returned.

Key Notes

1. Data structure to store Entry objects is an array named **table** of type Entry.
2. A particular index location in array is referred as bucket, because it can hold the first element of a LinkedList of Entry objects.

3. Key object’s hashCode() is required to calculate the index location of Entry object.
4. Key object’s equals() method is used to maintain uniqueness of Keys in map.
5. Value object’s hashCode() and equals() method are not used in HashMap’s get() and put() methods.
6. Hash code for null keys is always zero, and such Entry object is always stored in zero index in Entry[].

I hope, i have correctly communicated my thoughts by this article. If you find any difference or need any help in any point, please drop a comment.

Happy Learning !!