(/)

A Guide to LinkedHashMap in Java

Last modified: April 29, 2018

by baeldung (/author/baeldung/)

Java (/category/java/)

1. Overview

In this article, we are going to explore the internal implementation of *LinkedHashMap* class. *LinkedHashMap* is a common implementation of *Map* interface.

This particular implementation is a subclass of *HashMap* and therefore shares the core building blocks of the *HashMap* implementation (/java-hashmap). As a result, it's highly recommended to brush up on that before proceeding with this article.

2. *LinkedHashMap* vs *HashMap*

The *LinkedHashMap* class is very similar to *HashMap* in most aspects. However, the linked hash map is based on both hash table and linked list to enhance the functionality of hash map.

It maintains a doubly-linked list running through all its entries in addition to an underlying array of default size 16.

To maintain the order of elements, the linked hashmap modifies the *Map.Entry* class of *HashMap* by adding pointers to the next and previous entries:

```
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }
}
```

Notice that the *Entry* class simply adds two pointers; *before* and *after* which enable it to hook itself to the linked list. Aside from that, it uses the *Entry* class implementation of a the HashMap.

Finally, remember that this linked list defines the order of iteration, which by default is the order of insertion of elements (insertion-order).

3. Insertion-Order *LinkedHashMap*

Let's have a look at a linked hash map instance which orders its entries according to how they're inserted into the map. It also guarantees that this order will be maintained throughout the life cycle of the map:

```
@Test
public void givenLinkedHashMap_whenGetsOrderedKeyset_thenCorrect() {
    LinkedHashMap<Integer, String> map = new LinkedHashMap<>();
    map.put(1, null);
    map.put(2, null);
    map.put(3, null);
    map.put(4, null);
    map.put(5, null);

    Set<Integer> keys = map.keySet();
    Integer[] arr = keys.toArray(new Integer[0]);

    for (int i = 0; i < arr.length; i++) {
        assertEquals(new Integer(i + 1), arr[i]);
    }
}
```

Here, we're simply making a rudimentary, non-conclusive test on the ordering of entries in the linked hash map.

We can guarantee that this test will always pass as the insertion order will always be maintained. **We cannot make the same guarantee for a HashMap.**

This attribute can be of great advantage in an API that receives any map, makes a copy to manipulate and returns it to the calling code. If the client needs the returned map to be ordered the same way before calling the API, then a linked hashmap is the way to go.

Insertion order is not affected if a key is re-inserted into the map.

4. Access-Order *LinkedHashMap*

LinkedHashMap provides a special constructor which enables us to specify, among custom load factor (LF) and initial capacity, **a different ordering mechanism/strategy called access-order**:

```
LinkedHashMap<Integer, String> map = new LinkedHashMap<>(16, .75f, true);
```

The first parameter is the initial capacity, followed by the load factor and the **last param is the ordering mode**. So, by passing in *true*, we turned out access-order, whereas the default was insertion-order.

This mechanism ensures that the order of iteration of elements is the order in which the elements were last accessed, from least-recently accessed to most-recently accessed.

And so, building a Least Recently Used (LRU) cache is quite easy and practical with kind of a map. A successful *put* or *get* operation results in an access for the entry:

```
@Test
public void givenLinkedHashMap_whenAccessOrderWorks_thenCorrect() {
    LinkedHashMap<Integer, String> map
      = new LinkedHashMap<>(16, .75f, true);
    map.put(1, null);
    map.put(2, null);
    map.put(3, null);
    map.put(4, null);
    map.put(5, null);

    Set<Integer> keys = map.keySet();
    assertEquals("[1, 2, 3, 4, 5]", keys.toString());

    map.get(4);
    assertEquals("[1, 2, 3, 5, 4]", keys.toString());

    map.get(1);
    assertEquals("[2, 3, 5, 4, 1]", keys.toString());

    map.get(3);
    assertEquals("[2, 5, 4, 1, 3]", keys.toString());
}
```

Notice how **the order of elements in the key set is transformed as we perform access operations on the map**.

Simply put, any access operation on the map results in an order such that the element that was accessed would appear last if an iteration were to be carried out right away.

After the above examples, it should be obvious that a *putAll* operation generates one entry access for each of the mappings in the specified map.

Naturally, iteration over a view of the map does't affect the order of iteration of the backing map; **only explicit access operations on the map will affect the order**.

LinkedHashMap also provides a mechanism for maintaining a fixed number of mappings and to keep dropping off the oldest entries in case a new one needs to be added.

The *removeEldestEntry* method may be overridden to enforce this policy for removing stale mappings automatically.

To see this in practice, let us create our own linked hash map class, for the sole purpose of enforcing the removal of stale mappings by extending *LinkedHashMap*:

```
1  public class MyLinkedHashMap<K, V> extends LinkedHashMap<K, V> {
2
3      private static final int MAX_ENTRIES = 5;
4
5      public MyLinkedHashMap(
6        int initialCapacity, float loadFactor, boolean accessOrder) {
7          super(initialCapacity, loadFactor, accessOrder);
8      }
9
10     @Override
11     protected boolean removeEldestEntry(Map.Entry eldest) {
12         return size() > MAX_ENTRIES;
13     }
14
15 }
```

Our override above will allow the map to grow to a maximum size of 5 entries. When the size exceeds that, each new entry will be inserted at the cost of losing the eldest entry in the map i.e. the entry whose last-access time precedes all the other entries:

```
@Test
public void givenLinkedHashMap_whenRemovesEldestEntry_thenCorrect() {
    LinkedHashMap<Integer, String> map
      = new MyLinkedHashMap<>(16, .75f, true);
    map.put(1, null);
    map.put(2, null);
    map.put(3, null);
    map.put(4, null);
    map.put(5, null);
    Set<Integer> keys = map.keySet();
    assertEquals("[1, 2, 3, 4, 5]", keys.toString());

    map.put(6, null);
    assertEquals("[2, 3, 4, 5, 6]", keys.toString());

    map.put(7, null);
    assertEquals("[3, 4, 5, 6, 7]", keys.toString());

    map.put(8, null);
    assertEquals("[4, 5, 6, 7, 8]", keys.toString());
}
```

Notice how oldest entries at the start of the key set keep dropping off as we add new ones to the map.

5. Performance Considerations

Just like *HashMap*, *LinkedHashMap* performs the basic *Map* operations of add, remove and contains in constant-time, as long as the hash function is well-dimensioned. It also accepts a null key as well as null values.

However, this **constant-time performance of *LinkedHashMap* is likely to be a little worse than the constant-time of *HashMap*** due to the added overhead of maintaining a doubly-linked list.

Iteration over collection views of *LinkedHashMap* also takes linear time *O(n)* similar to that of *HashMap*. On the flip side, ***LinkedHashMap*'s linear time performance during iteration is better than *HashMap*'s linear time**.

This is because, for *LinkedHashMap*, *n* in *O(n)* is only the number of entries in the map regardless of the capacity. Whereas, for *HashMap*, *n* is capacity and the size summed up, *O(size+capacity).*

Load Factor and Initial Capacity are defined precisely as for *HashMap*. Note, however, that **the penalty for choosing an excessively high value for initial capacity is less severe for *LinkedHashMap* than for *HashMap***, as iteration times for this class are unaffected by capacity.

6. Concurrency

Just like *HashMap*, *LinkedHashMap* implementation is not synchronized. So if you are going to access it from multiple threads and at least one of these threads is likely to change it structurally, then it must be externally synchronized.

It's best to do this at creation:

```
1 Map m = Collections.synchronizedMap(new LinkedHashMap());
```

The difference with *HashMap* lies in what entails a structural modification. **In access-ordered linked hash maps, merely calling the *get* API results in a structural modification**. Alongside this, are operations like *put* and *remove*.

7. Conclusion

In this article, we have explored Java *LinkedHashMap* class as one of the foremost implementations of *Map* interface in terms of usage. We have also explored its internal workings in terms of the difference from *HashMap* which is its superclass.

Hopefully, after having read this post, you can make more informed and effective decisions as to which Map implementation to employ in your use case.

The full source code for all the examples used in this article can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/core-java-collections).

CATEGORIES

SPRING (/CATEGORY/SPRING/)
REST (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SECURITY (/CATEGORY/SECURITY-2/)

PERSISTENCE (/CATEGORY/PERSISTENCE/)

JACKSON (/CATEGORY/JACKSON/)

HTTPCLIENT (/CATEGORY/HTTP/)

KOTLIN (/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)

JACKSON JSON TUTORIAL (/JACKSON)

HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)

REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES/)

SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES/)

SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT/)

THE COURSES (HTTP://COURSES.BAELDUNG.COM)

CONSULTING WORK (/CONSULTING)

META BAELDUNG (HTTP://META.BAELDUNG.COM/)

THE FULL ARCHIVE (/FULL_ARCHIVE)

WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)

CONTACT (/CONTACT)

EDITORS (/EDITORS)

MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)