ConcurrentHashMap in Java

Difficulty Level : Easy Last Updated : 22 Mar, 2021

Prerequisites: ConcurrentMap

The **ConcurrentHashMap** class is introduced in JDK 1.5 belongs to **java.util.concurrent** package, which implements ConcurrentMap as well as to Serializable interface also. ConcurrentHashMap is an enhancement of HashMap as we know that while dealing with Threads in our application HashMap is not a good choice because performance-wise HashMap is not up to the mark.

Key points of ConcurrentHashMap:

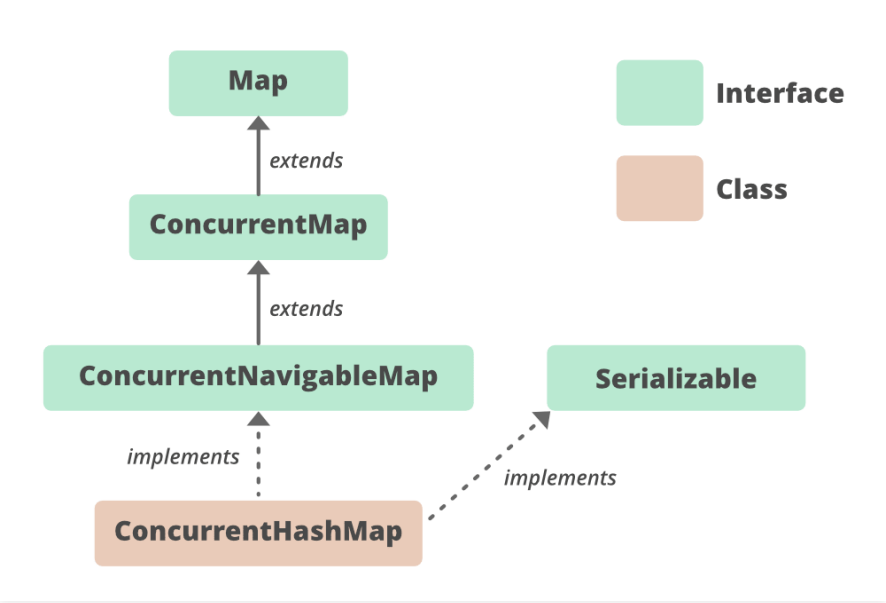
- The underlined data structure for ConcurrentHashMap is Hashtable.
- ConcurrentHashMap class is thread-safe i.e. multiple threads can operate on a single object without any complications.
- At a time any number of threads are applicable for a read operation without locking the ConcurrentHashMap object which is not there in HashMap.
- In ConcurrentHashMap, the Object is divided into a number of segments according to the concurrency level.
- The default concurrency-level of ConcurrentHashMap is 16.
- In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updated in the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is known as **Segment locking or bucket locking**. Hence at a time, 16 update operations can be performed by threads.
- Inserting null objects is not possible in ConcurrentHashMap as a key or value.

Declaration:

public class ConcurrentHashMap<K,V> extends AbstractMap<K,V> implements Concurre ntMap<K,V>, Serializable

Here, **K** is the key Object type and **V** is the value Object type.

The Hierarchy of ConcurrentHashMap



It implements **Serializable**, ConcurrentMap<K, V>, Map<K, V> interfaces and extends **AbstractMap<K, V>** class.

Constructors of ConcurrentHashMap

- **Concurrency-Level:** It is the number of threads concurrently updating the map. The implementation performs internal sizing to try to accommodate this many threads.
- **Load-Factor:** It's a threshold, used to control resizing.
- **Initial Capacity:** Accommodation of a certain number of elements initially provided by the implementation. if the capacity of this map is 10. It means that it can store 10 entries.

1. ConcurrentHashMap(): Creates a new, empty map with a default initial capacity (16), load factor (0.75) and concurrencyLevel (16).

```
ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>();
```

2. ConcurrentHashMap(int initialCapacity): Creates a new, empty map with the specified initial capacity, and with default load factor (0.75) and concurrencyLevel (16).

```
ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>(int initialCapacity);
```

3. ConcurrentHashMap(int initialCapacity, float loadFactor): Creates a new, empty map with the specified initial capacity and load factor and with the default concurrencyLevel (16).

ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>(int initialCapacity, float loadFactor);

4. ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel): Creates a new, empty map with the specified initial capacity, load factor, and concurrency level.

ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>(int initialCapacity, float loadFactor, int concurrencyLevel);

5. ConcurrentHashMap(Map m): Creates a new map with the same mappings as the given map.

ConcurrentHashMap<K, V> chm = new ConcurrentHashMap<>(Map m);

Example:

```
// Java program to demonstrate working of ConcurrentHashMap

import java.util.concurrent.*;

class ConcurrentHashMapDemo {

    public static void main(String[] args)
    {
        // create an instance of
        // ConcurrentHashMap
        ConcurrentHashMap<Integer, String> m
            = new ConcurrentHashMap<>();

        // Insert mappings using
        // put method
        m.put(100, "Hello");
        m.put(101, "Geeks");
        m.put(102, "Geeks");

        // Here we cant add Hello because 101 key
        // is already present in ConcurrentHashMap object
```

```
        m.putIfAbsent(101, "Hello");

        // We can remove entry because 101 key
        // is associated with For value
        m.remove(101, "Geeks");

        // Now we can add Hello
        m.putIfAbsent(103, "Hello");

        // We cant replace Hello with For
        m.replace(101, "Hello", "For");
        System.out.println(m);
    }
}
```

Output

```
{100=Hello, 102=Geeks, 103=Hello}
```

Basic Operations on ConcurrentHashMap

1. Adding Elements

To insert mappings into a ConcurrentHashMap, we can use put() or putAll() methods. The below example code explains these two methods.

```
// Java program to demonstrate adding
// elements to the ConcurrentHashMap

import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class AddingElementsToConcuurentHashMap {

    public static void main(String[] args)
    {
        // Creating ConcurrentHashMap
        ConcurrentHashMap<String, String> my_cmmap
            = new ConcurrentHashMap<String, String>();

        // Adding elements to the map
        // using put() method
        my_cmmap.put("1", "1");
        my_cmmap.put("2", "1");
```

```
        my_cmmap.put("3", "1");
        my_cmmap.put("4", "1");
        my_cmmap.put("5", "1");
        my_cmmap.put("6", "1");

        // Printing the map
        System.out.println("Mappings of my_cmmap : "
                           + my_cmmap);

        // create another concurrentHashMap
        ConcurrentHashMap<String, String> new_chm
            = new ConcurrentHashMap<>();

        // copy mappings from my_cmmap to new_chm
        new_chm.putAll(my_cmmap);

        // Displaying the new map
        System.out.println("New mappings are: " + new_chm);
    }
}
```

Output

```
Mappings of my_cmmap : {1=1, 2=1, 3=1, 4=1, 5=1, 6=1}
New mappings are: {1=1, 2=1, 3=1, 4=1, 5=1, 6=1}
```

2. Removing Elements

To remove a mapping, we can use remove(Object key) method of class ConcurrentHashmap. If the key does not exist in the map, then this function does nothing. To clear the entire map, we can use the clear() method.

```
// Java program to demonstrate removing
// elements from ConcurrentHashMap

import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class RemoveElementsFromConcurrentHashMap {

    public static void main(String[] args)
    {
        // Creating ConcurrentHashMap
```

```
        Map<String, String> my_cmmap
            = new ConcurrentHashMap<String, String>();

        // Adding elements to the map
        // using put() method
        my_cmmap.put("1", "1");
        my_cmmap.put("2", "1");
        my_cmmap.put("3", "1");
        my_cmmap.put("4", "1");
        my_cmmap.put("5", "1");
        my_cmmap.put("6", "1");

        // Printing the map
        System.out.println("Map: " + my_cmmap);
        System.out.println();

        // Removing the mapping
        // with existing key 6
        // using remove() method
        String valueRemoved = my_cmmap.remove("6");

        // Printing the map after remove()
        System.out.println(
            "After removing mapping with key 6:");
        System.out.println("Map: " + my_cmmap);
        System.out.println("Value removed: "
                           + valueRemoved);
        System.out.println();

        // Removing the mapping
        // with non-existing key 10
        // using remove() method
        valueRemoved = my_cmmap.remove("10");

        // Printing the map after remove()
        System.out.println(
            "After removing mapping with key 10:");
        System.out.println("Map: " + my_cmmap);
        System.out.println("Value removed: "
                           + valueRemoved);
        System.out.println();

        // Now clear the map using clear()
        my_cmmap.clear();

        // Print the clea Map
        System.out.println("Map after use of clear(): "
                           + my_cmmap);
    }
}
```

Output

```
Map: {1=1, 2=1, 3=1, 4=1, 5=1, 6=1}

After removing mapping with key 6:
Map: {1=1, 2=1, 3=1, 4=1, 5=1}
Value removed: 1

After removing mapping with key 10:
Map: {1=1, 2=1, 3=1, 4=1, 5=1}
Value removed: null

Map after use of clear(): {}
```

3. Accessing the Elements

We can access the elements of a ConcurrentHashMap using the get() method, the example of this is given below.

```
// Java Program Demonstrate accessing
// elements of ConcurrentHashMap

import java.util.concurrent.*;

class AccessingElementsOfConcurrentHashMap {

    public static void main(String[] args)
    {

        // create an instance of ConcurrentHashMap
        ConcurrentHashMap<Integer, String> chm
            = new ConcurrentHashMap<Integer, String>();

        // insert mappings using put method
        chm.put(100, "Geeks");
        chm.put(101, "for");
        chm.put(102, "Geeks");
        chm.put(103, "Contribute");

        // Displaying the HashMap
        System.out.println("The Mappings are: ");
        System.out.println(chm);

        // Display the value of 100
        System.out.println("The Value associated to "
```

```
                            + "100 is : " + chm.get(100));

        // Getting the value of 103
        System.out.println("The Value associated to "
                            + "103 is : " + chm.get(103));
    }
}
```

Output

```
The Mappings are:
{100=Geeks, 101=for, 102=Geeks, 103=Contribute}
The Value associated to 100 is : Geeks
The Value associated to 103 is : Contribute
```

4. Traversing

We can use the Iterator interface to traverse over any structure of the Collection Framework. Since Iterators work with one type of data we use Entry< ? , ? > to resolve the two separate types into a compatible format. Then using the next() method we print the elements of the ConcurrentHashMap.

```
// Java Program for traversing a
// ConcurrentHashMap
import java.util.*;
import java.util.concurrent.*;

public class TraversingConcurrentHashMap {

    public static void main(String[] args)
    {

        // create an instance of ConcurrentHashMap
        ConcurrentHashMap<Integer, String> chmap
            = new ConcurrentHashMap<Integer, String>();

        // Add elements using put()
        chmap.put(8, "Third");
        chmap.put(6, "Second");
        chmap.put(3, "First");
        chmap.put(11, "Fourth");
```

```
        // Create an Iterator over the
        // ConcurrentHashMap
        Iterator<ConcurrentHashMap.Entry<Integer, String> >
            itr = chmap.entrySet().iterator();

        // The hasNext() method is used to check if there is
        // a next element The next() method is used to
        // retrieve the next element
        while (itr.hasNext()) {
            ConcurrentHashMap.Entry<Integer, String> entry
                = itr.next();
            System.out.println("Key = " + entry.getKey()
                               + ", Value = "
                               + entry.getValue());
        }
    }
}
```

Output

```
Key = 3, Value = First
Key = 6, Value = Second
Key = 8, Value = Third
Key = 11, Value = Fourth
```

Methods of ConcurrentHashMap

- **K** – The type of the keys on the map.
- **V** – The type of values mapped in the map.

METHOD	DESCRIPTION
clear()	Removes all of the mappings from this map.
compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)	If the specified key is not already associated with a value, attempts to compute its value using the given mapping function and enters it into this map unless null.

METHOD	DESCRIPTION
computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	If the value for the specified key is present, attempts to compute a new mapping given the key and its current mapped value.
contains(Object value)	Tests if some keymaps into the specified value in this table.
containsKey(Object key)	Tests if the specified object is a key in this table.
containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
elements()	Returns an enumeration of the values in this table.
entrySet()	Returns a Set view of the mappings contained in this map.
equals(Object o)	Compares the specified object with this map for equality.
forEach(long parallelismThreshold, BiConsumer<? super K,? super V> action)	Performs the given action for each (key, value).
forEach(long parallelismThreshold, BiFunction<? super K,? super V,? extends U> transformer, Consumer<? super U> action)	Performs the given action for each non-null transformation of each (key, value).
forEachEntry(long parallelismThreshold, Consumer<? super Map.Entry<K,V>> action)	Performs the given action for each entry.
forEachEntry(long parallelismThreshold, Function<Map.Entry<K,V>,? extends U> transformer, Consumer<? super U> action)	Performs the given action for each non-null transformation of each entry.
forEachKey(long parallelismThreshold, Consumer<? super K> action)	Performs the given action for each key.
forEachKey(long parallelismThreshold, Function<? super K,? extends U> transformer, Consumer<? super U> action)	Performs the given action for each non-null transformation of each key.
forEachValue(long parallelismThreshold, Consumer<? super V> action)	Performs the given action for each value.

METHOD	DESCRIPTION
forEachValue(long parallelismThreshold, Function<? super V,? extends U> transformer, Consumer<? super U> action)	Performs the given action for each non-null transformation of each value.
get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
getOrDefault(Object key, V defaultValue)	Returns the value to which the specified key is mapped, or the given default value if this map contains no mapping for the key.
hashCode()	Returns the hash code value for this Map, i.e., the sum of, for each key-value pair in the map, key.hashCode() ^ value.hashCode().
keys()	Returns an enumeration of the keys in this table.
keySet()	Returns a Set view of the keys contained in this map.
keySet(V mappedValue)	Returns a Set view of the keys in this map, using the given common mapped value for any additions (i.e., Collection.add(E) and Collection.addAll(Collection)).
mappingCount()	Returns the number of mappings.
merge(K key, V value, BiFunction<? super V, ? super V,? extends V> remappingFunction)	If the specified key is not already associated with a (non-null) value, associates it with the given value.
newKeySet()	Creates a new Set backed by a ConcurrentHashMap from the given type to Boolean.TRUE.
newKeySet(int initialCapacity)	Creates a new Set backed by a ConcurrentHashMap from the given type to Boolean.TRUE.
put(K key, V value)	Maps the specified key to the specified value in this table.

METHOD	DESCRIPTION
putAll(Map<? extends K,? extends V> m)	Copies all of the mappings from the specified map to this one.
putIfAbsent(K key, V value)	If the specified key is not already associated with a value, associates it with the given value.
reduce(long parallelismThreshold, BiFunction<? super K,? super V,? extends U> transformer, BiFunction<? super U,? super U, ? extends U> reducer)	Returns the result of accumulating the given transformation of all (key, value) pairs using the given reducer to combine values, or null if none.
reduceEntries(long parallelismThreshold, BiFunction<Map.Entry<K,V>,Map.Entry<K, V>,? extends Map.Entry<K,V>> reducer)	Returns the result of accumulating all entries using the given reducer to combine values, or null if none.
reduceEntries(long parallelismThreshold, Function<Map.Entry<K,V>,? extends U> transformer, BiFunction<? super U,? super U, ? extends U> reducer)	Returns the result of accumulating the given transformation of all entries using the given reducer to combine values, or null if none.
reduceEntriesToDouble(long parallelismThreshold, ToDoubleFunction<Map.Entry<K,V>> transformer, double basis, DoubleBinaryOperator reducer)	Returns the result of accumulating the given transformation of all entries using the given reducer to combine values, and the given basis as an identity value.
reduceEntriesToInt(long parallelismThreshold, ToIntFunction<Map.Entry<K,V>> transformer, int basis, IntBinaryOperator reducer)	Returns the result of accumulating the given transformation of all entries using the given reducer to combine values, and the given basis as an identity value.
reduceEntriesToLong(long parallelismThreshold, ToLongFunction<Map.Entry<K,V>> transformer, long basis, LongBinaryOperator reducer)	Returns the result of accumulating the given transformation of all entries using the given reducer to combine values, and the given basis as an identity value.
reduceKeys(long parallelismThreshold, BiFunction<? super K,? super K,? extends K> reducer)	Returns the result of accumulating all keys using the given reducer to combine values, or null if none.

METHOD	DESCRIPTION
reduceKeys(long parallelismThreshold, Function<? super K,? extends U> transformer, BiFunction<? super U,? super U, ? extends U> reducer)	Returns the result of accumulating the given transformation of all keys using the given reducer to combine values, or null if none.
reduceKeysToDouble(long parallelismThreshold, ToDoubleFunction<? super K> transformer, double basis, DoubleBinaryOperator reducer)	Returns the result of accumulating the given transformation of all keys using the given reducer to combine values, and the given basis as an identity value.
reduceKeysToInt(long parallelismThreshold, ToIntFunction<? super K> transformer, int basis, IntBinaryOperator reducer)	Returns the result of accumulating the given transformation of all keys using the given reducer to combine values, and the given basis as an identity value.
reduceKeysToLong(long parallelismThreshold, ToLongFunction<? super K> transformer, long basis, LongBinaryOperator reducer)	Returns the result of accumulating the given transformation of all keys using the given reducer to combine values, and the given basis as an identity value.
reduceToDouble(long parallelismThreshold, ToDoubleBiFunction<? super K,? super V> transformer, double basis, DoubleBinaryOperator reducer)	Returns the result of accumulating the given transformation of all (key, value) pairs using the given reducer to combine values, and the given basis as an identity value.
reduceToInt(long parallelismThreshold, ToIntBiFunction<? super K,? super V> transformer, int basis, IntBinaryOperator reducer)	Returns the result of accumulating the given transformation of all (key, value) pairs using the given reducer to combine values, and the given basis as an identity value.
reduceToLong(long parallelismThreshold, ToLongBiFunction<? super K,? super V> transformer, long basis, LongBinaryOperator reducer)	Returns the result of accumulating the given transformation of all (key, value) pairs using the given reducer to combine values, and the given basis as an identity value.
reduceValues(long parallelismThreshold, BiFunction<? super V,? super V,? extends V> reducer)	Returns the result of accumulating all values using the given reducer to combine values, or null if none.
reduceValues(long parallelismThreshold, Function<? super V,? extends U> transformer, BiFunction<? super U,? super U,? extends U> reducer)	Returns the result of accumulating the given transformation of all values using the given reducer to combine values, or null if none.

METHOD	DESCRIPTION
reduceValuesToDouble(long parallelismThreshold, ToDoubleFunction<? super V> transformer, double basis, DoubleBinaryOperator reducer)	Returns the result of accumulating the given transformation of all values using the given reducer to combine values, and the given basis as an identity value.
reduceValuesToInt(long parallelismThreshold, ToIntFunction<? super V> transformer, int basis, IntBinaryOperator reducer)	Returns the result of accumulating the given transformation of all values using the given reducer to combine values, and the given basis as an identity value.
reduceValuesToLong(long parallelismThreshold, ToLongFunction<? super V> transformer, long basis, LongBinaryOperator reducer)	Returns the result of accumulating the given transformation of all values using the given reducer to combine values, and the given basis as an identity value.
remove(Object key)	Removes the key (and its corresponding value) from this map.
remove(Object key, Object value)	Removes the entry for a key only if currently mapped to a given value.
replace(K key, V value)	Replaces the entry for a key only if currently mapped to some value.
replace(K key, V oldValue, V newValue)	Replaces the entry for a key only if currently mapped to a given value.
search(long parallelismThreshold, BiFunction<? super K,? super V,? extends U> searchFunction)	Returns a non-null result from applying the given search function on each (key, value), or null if none.
searchEntries(long parallelismThreshold, Function<Map.Entry<K,V>,? extends U> searchFunction)	Returns a non-null result from applying the given search function on each entry, or null if none.
searchKeys(long parallelismThreshold, Function<? super K,? extends U> searchFunction)	Returns a non-null result from applying the given search function on each key, or null if none.
searchValues(long parallelismThreshold, Function<? super V,? extends U> searchFunction)	Returns a non-null result from applying the given search function on each value, or null if none.

METHOD	DESCRIPTION
toString()	Returns a string representation of this map.
values()	Returns a Collection view of the values contained in this map.

Methods declared in class java.util.AbstractMap

METHOD	DESCRIPTION
clone()	Returns a shallow copy of this AbstractMap instance: the keys and values themselves are not cloned.
isEmpty()	Returns true if this map contains no key-value mappings.
size()	Returns the number of key-value mappings in this map.

Methods declared in interface java.util.concurrent.ConcurrentMap

METHOD	DESCRIPTION
forEach(BiConsumer<? super K,? super V> action)	Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
replaceAll(BiFunction<? super K,? super V,? extends V> function)	Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

Must Read: Difference between HashMap and ConcurrentHashMap

ConcurrentHashMap vs Hashtable

HashTable

- **Hashtable** is an implementation of Map data structure
- This is a legacy class in which all methods are synchronized on Hashtable instances using the synchronized keyword.
- Thread-safe as it's method are synchronized

ConcurrentHashMap

- **ConcurrentHashMap** implements Map data structure and also provide thread safety like Hashtable.
- It works by dividing complete hashtable array into segments or portions and allowing parallel access to those segments.
- The locking is at a much finer granularity at a hashmap bucket level.
- Use **ConcurrentHashMap** when you need very high concurrency in your application.
- It is a thread-safe without synchronizing the whole map.
- Reads can happen very fast while the write is done with a lock on segment level or bucket level.
- There is no locking at the object level.
- ConcurrentHashMap doesn't throw a **ConcurrentModificationException** if one thread tries to modify it while another is iterating over it.
- ConcurrentHashMap does not allow NULL values, so the key can not be null in **ConcurrentHashMap**
- ConcurrentHashMap doesn't throw a **ConcurrentModificationException** if one thread tries to modify it, while another is iterating over it.

Properties	Hashtable	ConcurrentHashMap
Creation	Map ht = new Hashtable();	Map chm = new ConcurrentHashMap();
Is Null Key Allowed ?	No	No
Is Null Value Allowed ?	No	No (does not allow either null keys or values)
Is Thread Safe ?	Yes	Yes, Thread safety is ensured by having separate locks for separate buckets, resulting in better performance. Performance is further improved by providing read access concurrently without any blocking.
Performance	Slow due to synchronization overhead.	Faster than Hashtable. ConcurrentHashMap is a better choice when there are **more reads than writes**.

Properties	Hashtable	ConcurrentHashMap
Iterator	Hashtable uses enumerator to iterate the values of Hashtable object. Enumerations returned by the Hashtable keys and elements methods are not fail-fast.	Fail-safe iterator: Iterator provided by the ConcurrentHashMap is fail-safe, which means it will not throw **ConcurrentModificationException**.

Conclusion:

If a thread-safe highly-concurrent implementation is desired, then it is recommended to use **ConcurrentHashMap** in place of **Hashtable**.

Reference: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ConcurrentHashMap.html