Concurrent collections in java

Concurrent Collections are introduced from JDK 1.5 onwards to enhance the performance of multithreaded application.

These are threadsafe collection and available in java.util.concurrent sub package.

Limitation of Traditional Collection :
---
1) In the Collection framework most of the Collection classes are not thread-safe because those are non-synchronized like ArrayList, LinkedList, HashSet, HashMap is non-synchronized in nature, So If multiple threads will perform any operation on the collection object simultaneously then we will get some wrong data this is known as Data race or Race condition.

2) Some Collection classes are synchronized like Vector, Hashtable but performance wise these classes are slow in nature.

Collections class has provided static methods to make our List, Set and Map interface classes as a synchronized.

a) public static List synchronizedList(List list)
b) public static Set synchronizedSet(Set set)
c) public static Map synchronizedMap(Map map)

3) Traditional Collection works with fail fast iterator that means while iterating the element, if there is a change in structure then we will get java.util.ConcurrentModificationException, On the other hand concurrent collection works with fail safe iterator where even though there is a change in structure but we will not get ConcurrentModificationException.

```
import java.util.*;
public class Collection1
{
    public static void main(String args[])
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
                al.add(50);
        System.out.println("Arraylist Elements : "+al);
        Set s = new HashSet(al);
```

```
        System.out.println("Set Elements are: "+s);
    }
}
----------------------------------------------------------------//Collections.synchronizedList(List list);
import java.util.*;
public class Collection2
{
        public static void main(String[] args)
        {
                ArrayList<String> arl = new ArrayList<>();
                arl.add("Apple");
                arl.add("Orange");
                arl.add("Grapes");
                arl.add("Mango");
                arl.add("Guava");
                arl.add("Mango");

                List<String> syncCollection = Collections.synchronizedList(arl);

                List<String> upperList = new ArrayList<>(); //New List

                Runnable listOperations = () ->
                {
                        synchronized (syncCollection)
                        {
           syncCollection.forEach(str -> upperList.add(str.toUpperCase()));
         }
     };

           Thread t1 = new Thread(listOperations);
           t1.start();

    upperList.forEach(x -> System.out.println(x));
        }
}
------------------------------------------------------------
//Collections.synchronizedSet(Set set);
import java.util.*;
public class Collection3
{
   public static void main(String[] args)
                {
      Set<String> set = Collections.synchronizedSet(new HashSet<>());
```

```
        set.add("Apple");
                set.add("Orange");
                set.add("Grapes");
                set.add("Mango");
                set.add("Guava");
                set.add("Mango");
        System.out.println("Set after Synchronization :");
        synchronized (set)
                {
          Spliterator<String> itr = set.spliterator();
                  itr.forEachRemaining(str -> System.out.println(str));
         }
    }
}
-------------------------------------------------------------
//Collections.synchronizedMap(Map map);
import java.util.*;
public class Collection4
{
   public static void main(String[] args)
         {
        Map<String, String> map = new HashMap<String, String>();
        map.put("1", "Ravi");
        map.put("4", "Elina");
        map.put("3", "Aryan");
        Map<String, String> synmap = Collections.synchronizedMap(map);
        System.out.println("Synchronized map is :" + synmap);
    }
}
-------------------------------------------------------------
package com.ravi.concurrent;

import java.util.Iterator;
import java.util.Vector;

class Concurrent extends Thread
{
        private Vector<String> listOfFruits;

        public Concurrent(Vector<String> listOfFruits)
        {
                super();
                this.listOfFruits = listOfFruits;
```

```
	}


	@Override
	public void run()
	{
		try
		{
			Thread.sleep(2000);
		}
		catch(InterruptedException e)
		{

		}
		listOfFruits.add("POMOGRANATE");
	}



}

public class ConcurrentModificationDemo {

	public static void main(String[] args) throws InterruptedException
	{
		Vector<String> fruits = new Vector<>();
		fruits.add("Apple");
		fruits.add("Orange");
		fruits.add("Grapes");
		fruits.add("Mango");
		fruits.add("Guava");

		Concurrent crm = new Concurrent(fruits);
		crm.start();

		Iterator<String> itr = fruits.iterator();
		while(itr.hasNext())
		{
			System.out.println(itr.next());
			Thread.sleep(500);
		}
```

```
        }

}
```
Note :- In the above program we will get java.util.ConcurrentModificationException because Iterator is fail fast iterator.

Working with Concurrent collection classes :

CopyOnWriteArrayList in java :

---

public class CopyOnWriteArrayList implements List, Cloneable, Serializable, RanomAccess

A CopyOnWriteArrayList is similar to an ArrayList but it has some additional features like thread-safe. This class is existing in java.util.concurrent sub package.

ArrayList is not thread-safe. We can't use ArrayList in the multi-threaded environment because it creates a problem in ArrayList values (Data inconsistency).

*The CopyOnWriteArrayList is an enhanced version of ArrayList. If we are making any modifications(add, remove, etc.) in CopyOnWriteArrayList then JVM creates a new copy by use of Cloning.

The CopyOnWriteArrayList is costly, if we want to perform update operations so it is immutable object , because whenever we make any changes the JVM creates a cloned copy of the array and add/update element to new object.

It is a thread-safe version of ArrayList as well as here Iterator is fail safe iterator.

*CopyOnWriteArrayList is the best choice if we want to perform read operation frequently in multithreaded environment.

The CopyOnWriteArrayList is a replacement of a synchronized List, because it offers better concurrency.

Constructors of CopyOnWriteArrayList in java :

We have 3 constructors :

1) CopyOnWriteArrayList c = new CopyOnWriteArrayList();
It creates an empty list in memory. This constructor is useful when we want to create a list without any value.

2) CopyOnWriteArrayList c = new CopyOnWriteArrayList(Collection c);
Interconversion of collections.

3) CopyOnWriteArrayList c = new CopyOnWriteArrayList(Object[] obj) ;
It Creates a list that containing all the elements that is specified Array. This constructor is useful when we want to create a CopyOnWriteArrayList from Array.

Note : All the immutable objects are thread-safe because, On immutable objects if we perform any operation then another object will be created in a new memory location so at a time multiple threads can work.

All String Object, Wrapper classes object, Concurrent collection classes are immutable so by default Thread-Safe.

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListExample1
{
   public static void main(String[] args)
   {
      List<String> list = Arrays.asList("Apple", "Orange", "Mango","Kiwi", "Grapes");


      CopyOnWriteArrayList<String> copyOnWriteList = new
CopyOnWriteArrayList<String>(list);

      System.out.println("Without modification = "+copyOnWriteList);

                  //Iterator1
      Iterator<String> iterator1 = copyOnWriteList.iterator();

      //Add one element and verify list is updated
      copyOnWriteList.add("Guava");

      System.out.println("After modification = "+copyOnWriteList);

      //Iterator2
      Iterator<String> iterator2 = copyOnWriteList.iterator();
```

```
        System.out.println("Element from first Iterator:");
        iterator1.forEachRemaining(System.out::println);

        System.out.println("Element from Second Iterator:");
        iterator2.forEachRemaining(System.out::println);
    }
}
------------------------------------------------------------
import java.util.*;
import java.util.concurrent.*;
class ConcurrentModification extends Thread
{
        CopyOnWriteArrayList<String> al = null;
        public ConcurrentModification(CopyOnWriteArrayList<String> al)
        {
                this.al = al;
        }
        @Override
        public void run()
        {
                try
                {
                        Thread.sleep(1000);
                }
                catch (InterruptedException e)
                {
                }
           al.add("KIWI");
        }
}
public class CopyOnWriteArrayListExample2
{
        public static void main(String[] args) throws InterruptedException
        {
                CopyOnWriteArrayList<String> arl = new CopyOnWriteArrayList<>();
                arl.add("Apple");
                arl.add("Orange");
                arl.add("Grapes");
                arl.add("Mango");
                arl.add("Guava");
                ConcurrentModification cm = new ConcurrentModification(arl);
                cm.start();
```

```
                Iterator<String> itr = arl.iterator();
        while(itr.hasNext())
                {
                        String str = itr.next();
                        System.out.println(str);
                   Thread.sleep(500);
                }

             System.out.println(".....");

                Spliterator<String> spl =  arl.spliterator();
                spl.forEachRemaining(x -> System.out.println(x));
          }
}
```

CopyOnWriteArraySet :

---

public class CopyOnWriteArraySet extends AbstractSet implements Serializable

A CopyOnWriteArraySet is a thread-safe version of HashSet in Java and it works like CopyOnWriteArrayList in java.

The CopyOnWriteArraySet internally used CopyOnWriteArrayList to perform all type of operation.It means the CopyOnWriteArraySet internally creates an object of CopyOnWriteArrayList and perform operation on it.

Whenever we perform add, set, and remove operation on CopyOnWriteArraySet, it internally creates a new object of CopyOnWriteArrayList and copies all the data to the new object by eliminating duplicates so, when it is used in by multiple threads, it doesn't create a problem, but it is well suited if we have small size collection and want to perform only read operation by multiple threads.

The CopyOnWriteArraySet is the replacement of synchronizedSet and offers better concurrency.

It creates a new copy of the array every time iterator is created, so performance is slower than HashSet.

Constructors :

It has two constructors

1) CopyOnWriteArraySet set1 = new CopyOnWriteArraySet();

It will create an empty Set

2) CopyOnWriteArraySet set1 = new CopyOnWriteArraySet(Collection c); Interconversion of collection.

```
import java.util.*;
import java.util.concurrent.CopyOnWriteArraySet;
import java.util.concurrent.CopyOnWriteArraySet;

public class CopyOnWriteArraySetExample1
{
  public static void main(String[] args)
   {
        CopyOnWriteArraySet<String> set = new CopyOnWriteArraySet<>();

        set.add("Java");
        set.add("Python");
        set.add("C++");
                    set.add("Java");

        Iterator itr = set.iterator();


        // Adding a new element
        set.add("JavaScript");

        for (String language : set)
                    {
           System.out.println(language);
        }

                    System.out.println(".....");
                    while(itr.hasNext())
              {
                          System.out.println(itr.next());
              }
    }
}
```

```
import java.util.concurrent.CopyOnWriteArraySet;

public class CopyOnWriteArraySetExample2
```

```
{
   public static void main(String[] args)
   {
      CopyOnWriteArraySet<Integer> set = new CopyOnWriteArraySet<Integer>();
      set.add(1);
      set.add(2);
      set.add(3);
      set.add(4);
      set.add(5);

      System.out.println("Is element contains: "+set.contains(1));

      System.out.println("Is set empty: "+set.isEmpty());

      System.out.println("remove element from set: "+set.remove(3));

      System.out.println("Element from Set: "+ set);
   }
}
```

*** ConcurrentHashMap : [Bucket Level Locking]

---

public class ConcurrentHashMap<K,V> extends AbstractMap<K,V> implements java.util.concurrent.ConcurrentMap<K,V>, Serializable

Like HashMap, ConcurrentHashMap provides similar functionality except that it has internally maintained concurrency.

It is the concurrent version of the HashMap. It internally maintains a Hashtable that is divided into segments(Buckets).

The number of segments depends upon the level of concurrency required the Concurrent HashMap. By default, it divides into 16 segments and each Segment behaves independently. It doesn't lock the whole HashMap as done in Hashtables/synchronizedMap, it only locks the particular segment(Bucket) of HashMap. [Bucket level locking]

ConcurrentHashMap allows multiple threads can perform read/write operation without locking the ConcurrentHashMap object.

It does not allow null as a key or evan null as a value.

[Note :- TreeSet, TreeMap, Hashtable, PriroityQueue, ConcurrentHashMap , These 5 classes never containing null key or null element)

It contains 5 types of constructor :
---
1) ConcurrentHashMap chm1 = new ConcurrentHashMap();

2) ConcurrentHashMap chm2 = new ConcurrentHashMap(int initialCapacity);

3) ConcurrentHashMap chm3 = new ConcurrentHashMap(int initialCapacity, float loadFactor);

4) ConcurrentHashMap chm4 = new ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel);

5) ConcurrentHashMap chm5 = new ConcurrentHashMap(ConcurrentMap m);

19-09-2024

Internal Working of ConcurrentHashMap :

Like HashMap and Hashtable, the ConcurrentHashMap is also used Hashtable data structure. But it is using the segment locking strategy to handle the multiple threads.

A segment(bucket) is a portion of ConcurrentHashMap and ConcurrentHashMap uses a separate lock for each thread. Unlike Hashtable or synchronized HashMap, it doesn't synchronize the whole HashMap or Hashtable for one thread.

As we have seen in the internal implementation of the HashMap, the default size of HashMap is 16 and it means there are 16 buckets. The ConcurrentHashMap uses the same concept is used in ConcurrentHashMap. It uses the 16 separate locks for 16 buckets by default because the default concurrency level is 16. It means a ConcurrentHashMap can be used by 16 threads at same time. If one thread is reading from one bucket(Segment), then the second bucket doesn't affect it.

Why we need ConcurrentHashMap in java?

As we know Hashtable and HashMap works based on key-value pairs. But why we are introducing another Map? As we know HashMap is not thread safe, but we can make it thread-safe by using Collections.synchronizedMap() method and Hashtable is thread-safe by default.

But a synchronized HashMap or Hashtable is accessible only by one thread at a time because the object get the lock for the whole HashMap or Hashtable. Even multiple threads can't perform read operations at the same time. It is the main disadvantage of Synchronized HashMap or

Hashtable, which creates performance issues. So ConcurrentHashMap provides better performance than Synchronized HashMap or Hashtable.

```
//Converting HashMap to ConcurrentHashMap
import java.util.HashMap;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample1
{
  public static void main(String args[])
  {

     HashMap<Integer, String> hashMap = new HashMap<Integer, String>();
     hashMap.put(1, "Ravi");
     hashMap.put(2, "Ankit");
     hashMap.put(3, "Prashant");
          hashMap.put(4, "Pallavi");

     ConcurrentHashMap<Integer, String> concurrentHashMap = new
ConcurrentHashMap<>(hashMap);
     System.out.println("Object from ConcurrentHashMap: "+ concurrentHashMap);

  }

}
```

```
import java.util.Iterator;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample2
{
  public static void main(String args[])
  {
     // Creating ConcurrentHashMap
     Map<String, String> cityTemperatureMap = new ConcurrentHashMap<>();

      cityTemperatureMap.put("Delhi", "30");
      cityTemperatureMap.put("Mumbai", "32");
      cityTemperatureMap.put("Chennai", "35");
      cityTemperatureMap.put("Bangalore", "22" );

      Iterator<String> iterator = cityTemperatureMap.keySet().iterator();
```

```
        while (iterator.hasNext())
        {
        System.out.println(cityTemperatureMap.get(iterator.next()));
          // adding new value, it won't throw error
          cityTemperatureMap.put("Hyderabad", "28");
        }
    }
}
```