Concurrent Collection: Understanding CopyOnWriteArrayList

* Why CopyOnWriteArrayList ?

Basically, a `CopyOnWriteArrayList` is similar to an `ArrayList`, with some additional and more advanced thread-safe features.

You know, `ArrayList` is not thread-safe so it's not safe to use in multi-threaded applications. We can achieve thread-safe feature for an `ArrayList` by using a synchronized wrapper like this:

```
List< String > unsafeList = new ArrayList< >();

List< String > safeList = Collections.synchronizedList(unsafeList);

safeList.add("Boom");   // safe to use with multiple threads
```

However, this synchronized list has a limitation: all of its read and write methods (`add`, `set`, `remove`, `iterator`, etc) are synchronized on the list object itself. That means if a thread is executing `add()` method, it blocks other threads which want to get the `iterator` to access elements in the list, for example. Also, only one thread can iterate the list's elements at a time, which can be inefficient. That's quite rigid.

What if we want a more flexible collection which allows:

- Multiple threads executing read operations concurrently.
- One thread executing read operation and another executing write operation concurrently.
- Only one thread can execute write operation while other threads can execute read operations simultaneously.

The `CopyOnWriteArrayList` class is designed to enable such sequential write and concurrent reads features. For example, we can write a multi-threaded program that allows one thread to add elements to the list while other threads are traversing the list's elements at the same time, and no worry about `ConcurrentModificationException` as per the case of a synchronized list.

That's interesting. So how does the `CopyOnWriteArrayList` implement this concurrent feature?

* How does CopyOnWriteArrayList works?

The `CopyOnWriteArrayList` class uses a mechanism called ***copy-on-write*** which works like this: For every write operation (`add`, `set`, `remove`, etc) it makes a new copy of the

elements in the list. That means the read operations (`get`, `iterator`, `listIterator`, etc) work on a different copy.

In addition, a thread must acquire a separate lock before executing a write operation, and all write operations use this same lock so there's only one write operation can be executed by only one thread at a time. The read operations do not use any lock so multiple threads can execute multiple read operations simultaneously. And of course, read and write operations do not block each other.

The methods `iterator()` and `listIterator()` return an iterator object that holds different copy of the elements, hence the term ***snapshot iterator***. The snapshot iterator doesn't allow modifying the list while traversing, and it will not throw `ConcurrentModificationException` if the list is being modified by other thread during the traversal, and the read and write operations work on different copies of elements.

* When to Use CopyOnWriteArrayList?

Due to its special behaviors, `CopyOnWriteArrayList` is suitable for use in scenarios require sequential write and concurrent reads on a same collection. But you should take performance issue into consideration because the process of copying elements is costly for a list that has a large number of elements and many write operations.

Having said that, use `CopyOnWriteArrayList` only when the number of write operations is very small as compared to the read operations and the list contains a small number of elements.

In some cases, we can use `CopyOnWriteArrayList` as a thread-safe alternative to `ArrayList`, and to take advantages of its new methods `addIfAbsent()` and `addAllAbsent()`, which are explained below.

* Understanding CopyOnWriteArrayList API:

`CopyOnWriteArrayList` is a member of the Java Collection framework and is an implementation the `List` interface so it has all typical behaviors of a list. `CopyOnWriteArrayList` is considered as a thread-safe alternative to `ArrayList` with some differences:

- You can pass an array when creating a new `CopyOnWriteArrayList` object. The list holds a copy of this array, for example:

```
String[] fruits = {"Apple", "Banana", "Lemon", "Grape", "Mango"};

List< String > list = new CopyOnWriteArrayList< >(fruits);
```

- Though a list allows duplicate elements, you can add an element to the list if and only if it is not already in the list, by using the method `addIfAbsent(element)`. More importantly, this method is thread-safe which means it guarantees no other threads can add the same element at the same time. This method returns true if the element was added. For example:

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

list.add("Apple");

list.add("Banana");

if (list.addIfAbsent("Orange")) {

      System.out.println("Orange was added");

}
```

- Similarly, the method `addAllAbsent(Collection)` appends all elements in the specified collection that are not already contained in the list. And more importantly, this method is thread-safe. This method returns the number of elements were added. For example:

```
CopyOnWriteArrayList< String > list1 = new CopyOnWriteArrayList< >();

list1.add("Apple");

list1.add("Banana");

List< String > list2 = Arrays.asList("Lemon", "Banana");

int result = list1.addAllAbsent(list2);

System.out.println("Elements added: " + result);
```

This print `Elements added: 1` because the element `Banana` is already contained in the `list1`.

- The method `iterator()` returns a generic `Iterator` that holds a snapshot of the list. This iterator doesn't support the `remove()` method.

- The method `listIerator()` returns a generic `ListIterator` that holds a snapshot of the list. This iterator doesn't support the `remove()`, `set()` or `add()` method.

And as stated previously, the iterator will not throw `ConcurrentModificationException` if the list is being modified by another thread while the current thread is traversing the iterator, because a snapshot iterator holds a different copy of elements.

* CopyOnWriteArrayList Examples:

Let's see a couple of examples in action. The first one creates two threads:

- Thread Writer adds a number to `CopyOnWriteArrayList` for every 5 seconds.

- Thread Reader iterates the list repeatedly with a small delay (10 milliseconds) for every iteration.

That means the read operations outnumber the write ones, and here's the full source code of the program:

```
import java.util.*;

import java.util.concurrent.*;

public class CopyOnWriteArrayListTest {

      public static void main(String[] args) {

            List< Integer > list = new CopyOnWriteArrayList< >();

            list.add(1);

            list.add(2);

            list.add(3);

            list.add(4);

            list.add(5);

            new WriteThread("Writer", list).start();

            new ReadThread("Reader", list).start();

      }

}

class WriteThread extends Thread {
```

```
	private List< Integer > list;
	public WriteThread(String name, List< Integer > list) {
		this.list = list;
		super.setName(name);
	}
	public void run() {
		int count = 6;
		while (true) {
			try {
				Thread.sleep(5000);
			} catch (InterruptedException ex) {
				ex.printStackTrace();
			}
			list.add(count++);
			System.out.println(super.getName() + " done");
		}
	}
}
class ReadThread extends Thread {
	private List< Integer > list;
	public ReadThread(String name, List< Integer > list) {
		this.list = list;
		super.setName(name);
	}
	public void run() {
		while (true) {
```

```
                        String output = "\n" + super.getName() + ":";
                        Iterator< Integer > iterator = list.iterator();
                        while (iterator.hasNext()) {
                              Integer next = iterator.next();
                              output += " " + next;
                              // fake processing time
                              try {
                                    Thread.sleep(10);
                              } catch (InterruptedException ex) {
                                    ex.printStackTrace();
                              }
                        }
                        System.out.println(output);
                  }
            }
      }
```

Run this program and observe the result. You will see that the reader thread constantly prints out the elements in the list, whereas the writer thread slowly adds a new number to the list. This program runs forever until you press Ctrl + C to stop it.

Try to change the list implementation from `CopyOnWriteArrayList` to `ArrayList` like this:

```
List< Integer > list = new ArrayList< >();
```

Recompile and run the program again, you will see that the reader thread throws `ConcurrentModificationException` as soon as the writer thread adds a new element to the list. The reader thread die and only the writer thread alive.

The second example demonstrates how to use `CopyOnWriteArrayList` in event handling. Consider the following class:

```
public class GuiComponent {

	private List< ActionListener > listeners = new
CopyOnWriteArrayList< >();

	public void addActionListener(ActionListener listener) {

		listeners.add(listener);

	}

	public void removeActionListener(ActionListener listener) {

		listeners.remove(listener);

	}

	public void fireActionEvent() {

		for (ActionListener listener : listeners) {

			listener.actionPerformed(new ActionEvent(this,
"message"));

		}

	}

}
```

Suppose this class represents a GUI component which can receives events such as mouse click. The client code can register (subscribe) to receive notification when the event occurs via the following method:

```
public void addActionListener(ActionListener listener)
```

The components use a `CopyOnWriteArrayList` object to maintain all registered listeners:

```
private List< ActionListener > listeners = new CopyOnWriteArrayList<
>();
```

When an event occurs, the component notifies all of its listeners by iterating the list and invoke the action handler method on each listener, as shown in the `fireActionEvent()` method:

```
for (ActionListener listener : listeners) {
      listener.actionPerformed(new ActionEvent(this, "message"));
}
```

The `ActionListener` interface is defined as follows:

```
public interface ActionListener {
      public void actionPerformed(ActionEvent evt);
}
```

And the `ActionEvent` class is implemented as follows:

```
public class ActionEvent {
      Object source;
      Object data;
      public ActionEvent(Object source, Object data) {
            this.source = source;
            this.data = data;
      }
}
```

Look at the `GuiComponent` class, a `CopyOnWriteArrayList` is used because:

- It allows two threads can both read and write the list concurrently: one thread adds or removes a listener and the other thread notifies all listeners.

- The number of listeners is changed infrequently, whereas the number of times the listeners are notified more frequently (the read operations outnumber the write ones).

- It doesn't throw `ConcurrentModificationException` if a thread is adding/removing a listener while another thread is iterating the list of listeners.

Summary:

`CopyOnWriteArrayList` can be used as a thread-safe alternative to `ArrayList`, with additional methods `addIfAbsent()` and `addAllAbsent()` that append elements if they are not contained in the list. A `CopyOnWriteArrayList` makes a new copy of its elements for every write operation and its iterator holds a different copy (snapshot) so it enables sequential writes and concurrent reads: only one thread can execute write operation and multiple threads can execute read operations at the same time. And its iterator doesn't throw `ConcurrentModification`.

Concurrent Collection: Understanding CopyOnWriteArraySet

The second concurrent collection in the `java.util.concurrent` package I'd like to talk about is `CopyOnWriteArraySet`, which is similar to `CopyOnWriteArrayList`. Actually, a `CopyOnWriteArraySet` uses a `CopyOnWriteArrayList` internally for its operations. Thus it has the following behaviors:

- It's thread-safe, and can be used as a thread-safe alternative to `HashSet`.
- It allows sequential write and concurrent reads by multiple threads. Only one thread can execute write operations (`add` and `remove`), and multiple threads can execute read operations (`iterator`) at a time.
- Its iterator doesn't throw `ConcurrentModificationException` and doesn't support `remove` method.

`CopyOnWriteArraySet` is a Set so it doesn't allow duplicate elements. But unlike `HashSet`, its iterator returns elements in the order they were added.

Therefore, consider using a `CopyOnWriteArraySet` if you need a thread-safe collection that is similar to `CopyOnWriteArrayList` but no duplicate elements are allowed.

Also note that you should use `CopyOnWriteArraySet` only when the read operations outnumber the write operations and has a small number of elements, because the set creates a new copy of its elements for each write operation, which affects performance if the set has a large number of elements and the write operations are frequent.

Let's see an example that demonstrates how `CopyOnWriteArraySet` works. The following program e creates two threads:

- Thread Writer adds a number to `CopyOnWriteArraySet` for every 5 seconds.

- Thread Reader iterates the list repeatedly with a small delay (10 milliseconds) for every iteration.

Here's the full source code of the program:

```
import java.util.*;

import java.util.concurrent.*;

public class CopyOnWriteArraySetTest {

	public static void main(String[] args) {

		Set< Integer > set = new CopyOnWriteArraySet< >();

		set.add(1);

		set.add(2);

		set.add(3);

		set.add(4);

		set.add(5);

		new WriteThread("Writer", set).start();

		new ReadThread("Reader", set).start();

	}

}

class WriteThread extends Thread {

	private Set< Integer > set;

	public WriteThread(String name, Set< Integer > set) {

		this.set = set;

		super.setName(name);

	}

	public void run() {
```

```
            int count = 6;
            while (true) {
                  try {
                        Thread.sleep(5000);
                  } catch (InterruptedException ex) {
                        ex.printStackTrace();
                  }
                  set.add(count++);
                  System.out.println(super.getName() + " done");
            }
      }
}
class ReadThread extends Thread {
      private Set< Integer > set;
      public ReadThread(String name, Set< Integer > set) {
            this.set = set;
            super.setName(name);
      }
      public void run() {
            while (true) {
                  String output = "\n" + super.getName() + ":";
                  Iterator< Integer > iterator = set.iterator();
                  while (iterator.hasNext()) {
                        Integer next = iterator.next();
                        output += " " + next;
                        // fake processing time
```

```
                        try {
                              Thread.sleep(10);
                        } catch (InterruptedException ex) {
                              ex.printStackTrace();
                        }
                  }
                  System.out.println(output);
            }
      }
}
```

You can see that the read operations outnumber the write ones.

Run this program and observe the result. You will see that the reader thread constantly prints out the elements in the list, whereas the writer thread slowly adds a new number to the list. This program runs forever until you press Ctrl + C to stop it.

Try to change the list implementation from `CopyOnWriteArraySet` to `HashSet` like this:

```
Set< Integer > set = new HashSet< >();
```

Recompile and run the program again, you will see that the reader thread throws `ConcurrentModificationException` as soon as the writer thread adds a new element to the list. The reader thread die and only the writer thread alive.

Concurrent Collection: Understanding ConcurrentHashMap

The next concurrent collection in the `java.util.concurrent` package I'd like to help you get familiar with today is `ConcurrentHashMap` - a `Map` implementation like `HashMap`and `Hashtable`, with additional support for concurrency features:

- Unlike `Hastable` or `synchronizedMap` which locks the entire map exclusively to gain thread-safety feature, `ConcurrentHashMap` allows concurrent writer and reader threads. That means it allows some threads to modify the map and other threads to read values from the map at the same time, while `Hashtable` or `synchronizedMap`allows only one thread to work on the map at a time. More specifically, `ConcurrentHashMap` allows any number of concurrent

reader threads and a limited number of concurrent writer threads, and both reader and writer threads can operate on the map simultaneously.

+ Reader threads perform retrieval operations such as `get`, `containsKey`, `size`, `isEmpty`, and iterate over keys set of the map.

+ Writer threads perform update operations such as `put` and `remove`.

- Iterators returned by `ConcurrentHashMap` are weakly consistent, meaning that the iterator may not reflect latest update since it was constructed. An iterator should be used by only one thread and no `ConcurrentModificationException` will be thrown if the map is modified while the iterator is being used.

`ConcurrentHashMap` is an implementation of `ConcurrentMap` which is a subtype of the `Map` interface. A `ConcurrentMap` defines the following atomic operations:

- **`putIfAbsent(K key, V value)`**: associates the specified key to the specified value if the key is not already associated with a value. This method is performed atomically, meaning that no other threads can intervene in the middle of checking absence and association.

- **`remove(Object key, Object value)`**: removes the entry for a key only if currently mapped to some value. This method is performed atomically.

- **`replace(K key, V value)`**: replaces the entry for a key only if currently mapped to some value. This method is performed atomically.

- **`replace(K key, V oldValue, V newValue)`**: replaces the entry for a key only if currently mapped to a given value. This method is performed atomically.

Also note that the methods `size()` and `isEmpty()` may return an approximation instead of an exact count due to the concurrent nature of the map. `ConcurrentHashMap` does not allow `null` key and `null` value.

`ConcurrentHashMap` has such advanced concurrent capabilities because it uses a finer-grained locking mechanism. We don’t delve in to the details of the locking algorithm, but understand that the `ConcurrentHashMap` uses different locks to lock different parts of the map, which enables concurrent reads and updates.

* ConcurrentHashMap Example:

The following example demonstrates how `ConcurrentHashMap` is used in a multi-threaded context. The program creates two writer threads and 5 reader threads working on a shared instance of a `ConcurrentHashMap`.

The writer thread randomly modifies the map (put and remove). Here's the code:

```
public class WriterThread extends Thread {

	private ConcurrentMap< Integer, String > map;

	private Random random;

	private String name;

	public WriterThread(ConcurrentMap< Integer, String > map,

                                    String threadName, long randomSeed)
{

		this.map = map;

		this.random = new Random(randomSeed);

		this.name = threadName;

	}

	public void run() {

		while (true) {

			Integer key = random.nextInt(10);

			String value = name;

			if(map.putIfAbsent(key, value) == null) {

				long time = System.currentTimeMillis();

				String output = String.format("%d: %s has put
[%d => %s]",


time, name, key, value);

				System.out.println(output);

			}

			Integer keyToRemove = random.nextInt(10);
```

```
                    if (map.remove(keyToRemove, value)) {
                          long time = System.currentTimeMillis();
                          String output = String.format("%d: %s has
removed [%d => %s]",

time, name, keyToRemove, value);
                          System.out.println(output);
                    }
                    try {
                          Thread.sleep(500);
                    } catch (InterruptedException ex) {
                          ex.printStackTrace();
                    }
              }
        }
}
```

The reader thread iterates over each key-value pair in the map and prints it out. Here's the code:

```
public class ReaderThread extends Thread {
      private ConcurrentHashMap< Integer, String > map;
      private String name;
      public ReaderThread(ConcurrentHashMap< Integer, String > map,
String threadName) {
              this.map = map;
              this.name = threadName;
      }
      public void run() {
              while (true) {
```

```
                    ConcurrentHashMap.KeySetView< Integer, String >
keySetView = map.keySet();

                    Iterator< Integer > iterator = keySetView.iterator();

                    long time = System.currentTimeMillis();

                    String output = time + ": " + name + ": ";

                    while (iterator.hasNext()) {

                          Integer key = iterator.next();

                          String value = map.get(key);

                          output += key + "=>" + value + "; ";

                    }

                    System.out.println(output);

                    try {

                          Thread.sleep(300);

                    } catch (InterruptedException ex) {

                          ex.printStackTrace();

                    }

              }

      }

}
```

And the main program creates and starts 2 writer threads and 5 reader threads to work concurrently on a shared instance of a `ConcurrentHashMap`. Here's the code:

```
public class ConcurrentHashMapExamples {

      public static void main(String[] args) {

            ConcurrentHashMap< Integer, String > map = new
ConcurrentHashMap< >();

            new WriterThread(map, "Writer-1", 1).start();

            new WriterThread(map, "Writer-2", 2).start();
```

```
                for (int i = 1; i <= 5; i++) {

                      new ReaderThread(map, "Reader-" + i).start();

                }

          }

    }
```

This program runs forever because all threads run an infinite loop, so you need to press Ctrl + C to stop the program and observe the output. The reader threads let you know that the mp is constantly updated by the writer threads. Here's a screenshot captured when running the above program on Windows:

* Differences between ConcurrentHashMap and HashMap, Hashtable and synchronizedMap:

`HashMap` is a non-threadsafe `Map` which should not be used by multiple threads.

`Hashtable` is a thread-safe `Map` that allows only one thread to execute a read/update operation at a time.

`synchronizedMap` is a thread-safe wrapper on a `Map` implementation. It is generated by the `Collections.synchronizedMap(Map)` factory method. A `synchronizedMap`also allows only a single thread to work on the map at a time.

And `ConcurrentHashMap` is a thread-safe `Map` with greater flexibility and higher scalability as it uses a special locking mechanism that enables multiple threads to read/update the map concurrently.

Therefore, you can use `ConcurrentHashMap` to replace `HashMap`/`Hastable`/`synchronizedMap` for concurrency needs without locking the whole map.