Understand Java Collections and Thread Safety

Written by Nam Ha Minh
Last Updated on 17 June 2019 | Print Email

This Java tutorial helps you understand how the Java Collections Framework is designed for concurrency; how we should use collections in single-threaded applications versus in multi-threaded ones.

Topics about concurrency are often a little bit complicated and not easy to understand, so I will try my best to explain them as simple as possible. By reading to the end of this tutorial, you will be armed with deeper knowledge about the Java Collections Framework and I bet it will be definitely helpful for your daily Java coding.

1. Why are almost collection classes not thread-safe?

Do you notice that all the basic collection classes - `ArrayList`, `LinkedList`, `HashMap`, `HashSet`, `TreeMap`, `TreeSet`, etc - all are not synchronized? In fact, all collection classes

ept `Vector` and `Hashtable`) in the `java.util` package are not thread-safe. The two legacy collections are thread-safe: `Vector` and `Hashtable`. WHY?

Here's the reason: Synchronization can be very expensive!

You know, `Vector` and `Hashtable` are the two collections exist early in Java history, and they are designed for thread-safe from the start (if you have chance to look at their source code, you will see their methods are all synchronized!). However, they quickly expose poor performance in multi-threaded programs. As you may know, synchronization requires locks which always take time to monitor, and that reduces the performance.

That's why the new collections (`List`, `Set`, `Map`, etc) provide no concurrency control at all to provide maximum performance in single-threaded applications.

The following test program compares performance between `Vector` and `ArrayList` - the two similar collections (`Vector` is thread-safe and `ArrayList` is not):

```
import java.util.*;

/**
 * This test program compares performance of Vector versus ArrayList
 * @author www.codejava.net
 *
 */
public class CollectionsThreadSafeTest {

    public void testVector() {
        long startTime = System.currentTimeMillis();

        Vector<Integer> vector = new Vector<>();

        for (int i = 0; i < 10_000_000; i++) {
            vector.addElement(i);
        }

        long endTime = System.currentTimeMillis();

        long totalTime = endTime - startTime;

        System.out.println("Test Vector: " + totalTime + " ms");

    }

    public void testArrayList() {
        long startTime = System.currentTimeMillis();

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

        for (int i = 0; i < 10_000_000; i++) {
            list.add(i);
        }

        long endTime = System.currentTimeMillis();

        long totalTime = endTime - startTime;

        System.out.println("Test ArrayList: " + totalTime + " ms");

    }

    public static void main(String[] args) {
        CollectionsThreadSafeTest tester = new CollectionsThreadSafe

        tester.testVector();

        tester.testArrayList();

    }

}
```

This program performs the test by comparing the time needed to add ten millions of elements into each collection. And here's a result:

```
Test Vector: 9266 ms
Test ArrayList: 4588 ms
```

As you can see, with a fairly large number of elements, the `ArrayList` performs about twice faster than the `Vector`. Let run this program on your computer to experiment the results yourself.

2. Fail-Fast Iterators

When working with collections, you also need to understand this concurrency policy with regard to their iterators: ***Fail-fast iterators***.

Consider the following code snippet that iterates a list of Strings:

```
List<String> listNames = Arrays.asList("Tom", "Joe", "Bill", "Dave",

Iterator<String> iterator = listNames.iterator();

while (iterator.hasNext()) {
    String nextName = iterator.next();
    System.out.println(nextName);
}
```

Here, we use the collection's iterator to traverse through elements in the collection. Imagine the `listNames` is shared between two threads: the current thread that executes the iteration, and another thread. Now imagine the second thread is modifying the collection (adding or removing elements) while the first thread is still iterating over the elements. Can you guess what happens?

The iteration code in the first thread throws `ConcurrentModificationException` and fails immediately, hence the term '*fail-fast iterators*'.

Why does the iterator fail so fast? It's because iterating a collection while it is being modified by another thread is very dangerous: the collection may have more, less or no elements after the iterator has been obtained, so that leads to unexpected behavior and inconsistent result. And this should be avoided as early as possible, thus the iterator must throw an exception to stop the execution of the current thread.

The following test program mimics a situation that throws ConcurrentModificationException:

```
import java.util.*;

/**
 * This test program illustrates how a collection's iterator fails f
 * and throw ConcurrentModificationException
 * @author www.codejava.net
 *
 */
public class IteratorFailFastTest {

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

    public IteratorFailFastTest() {
        for (int i = 0; i < 10_000; i++) {
            list.add(i);
        }
    }

    public void runUpdateThread() {
        Thread thread1 = new Thread(new Runnable() {

            public void run() {
                for (int i = 10_000; i < 20_000; i++) {
                    list.add(i);
                }
            }
        });

        thread1.start();
    }


    public void runIteratorThread() {
        Thread thread2 = new Thread(new Runnable() {

            public void run() {
                ListIterator<Integer> iterator = list.listIterator()
                while (iterator.hasNext()) {
                    Integer number = iterator.next();
                    System.out.println(number);
                }
            }
        });

        thread2.start();
    }

    public static void main(String[] args) {
        IteratorFailFastTest tester = new IteratorFailFastTest();

        tester.runIteratorThread();
        tester.runUpdateThread();
    }
}
```

As you can see, the `thread1` is iterating the `list`, while the `thread2` adds more elements to the collection. This causes the `ConcurrentModificationException` to be thrown.

Note that the fail-fast behavior of collection's iterators intends to help find and diagnose bugs easily. We should not rely on it to handle `ConcurrentModificationException` in our programs, because the fail-fast behavior is not guaranteed. That means if this exception is thrown, the program should stop immediately, instead of continuing the execution.

Now you understand how ConcurrentModificationExceptionworks and it's better to avoid it.

Synchronized Wrappers

So far we've understood that the basic collection implementations are not thread-safe in order to provide maximum performance in single-threaded applications. What if we have to use collections in multi-threaded context?

Of course we should not use non-thread-safe collections in concurrent context, as doing so may lead to undesired behaviors and inconsistent results. We can use synchronized blocks manually to safeguard our code, however it's always wise to use thread-safe collections instead of writing synchronization code manually.

You already know that, the Java Collections Framework provides factory methods for creating thread-safe collections. These methods are in the following form:

```
Collections.synchronizedXXX(collection)
```

These factory methods wrap the specified collection and return a thread-safe implementation. Here, XXX can be `Collection`, `List`, `Map`, `Set`, `SortedMap` and `SortedSet` implementations. For example, the following code creates a thread-safe list collection:

```
1  List<String> safeList = Collections.synchronizedList(new ArrayList<>(
```

If we have an existing non-thread-safe collection, we can wrap it in a thread-safe collection like this:

```
1  Map<Integer, String> unsafeMap = new HashMap<>();
2  Map<Integer, String> safeMap = Collections.synchronizedMap(unsafeMap)
```

You see, these factory methods wrap the specified collection in an implementation having same interfaces as the wrapped collection but all the methods are synchronized to provide thread-safety, hence the term '*synchronized wrappers*'. Actually, the synchronized collection delegate all work to the wrapped collection.

NOTE:

When using the iterator of a synchronized collection, we should use synchronized block to safeguard the iteration code because the iterator itself is not thread-safe. Consider the following code:

```
1  List<String> safeList = Collections.synchronizedList(new ArrayList<>
2
3  // adds some elements to the list
4
5  Iterator<String> iterator = safeList.iterator();
6
7  while (iterator.hasNext()) {
8      String next = iterator.next();
9      System.out.println(next);
10 }
```

Although the `safeList` is thread-safe, its iterator is not, so we should manually add synchronized block like this:

```
synchronized (safeList) {
    while (iterator.hasNext()) {
        String next = iterator.next();
        System.out.println(next);
    }
}
```

Also note that the iterators of the synchronized collections are fail-fast.

Although synchronized wrappers can be safely used in multi-threaded applications, they have drawbacks as explained in the next section.

4. Concurrent Collections

A drawback of synchronized collections is that their synchronization mechanism uses the collection object itself as the lock object. That means when a thread is iterating over elements in a collection, all other collection's methods block, causing other threads having to wait. Other threads cannot perform other operations on the collection until the first thread release the lock. This causes overhead and reduces performance.

That's why Java 5 (and beyond) introduces ***concurrent collections*** that provide much better performance than synchronized wrappers. The concurrent collection classes are organized in the `java.util.concurrent` package. They are categorized into 3 groups based on their thread safety mechanisms.

* **The first group is *copy-on-write* collections:** this kind of thread-safe collections stores values in an immutable array; any change to the value of the collection results in a new array being created to reflect the new values. These collections are designed for situations in which read operations greatly predominate write operations. There are two implementations of this kind: `CopyOnWriteArrayList` and `CopyOnWriteArraySet`.

Note that copy-on-write collections have ***snapshot iterators*** which do not throw `ConcurrentModificationException`. Since these collections are backed by immutable arrays, an iterator can read the values in one of these arrays (but never modify them) without danger of them being changed by another thread.

* **The second group is *Compare-And-Swap* or *CAS* collections:** the collections in this group implement thread safety using an algorithm called Compare-And-Swap (CAS) which can be understood like this:

To perform calculation and update value of a variable, it makes a local copy of the variable and performs the calculation without getting exclusive access. When it is ready to update the variable, it compares the variable's value with its value at the start and, if they are the same, updates it with the new value.

If they are not the same, the variable must have been changed by another thread. In this situation, the CAS thread can try the whole computation again using the new value, or

up, or continue. Collections using CAS include `ConcurrentLinkedQueue` and `urrentSkipListMap`.

Note that the CAS collections have weakly consistent iterators, which reflect some but not necessarily all of the changes that have been made to their backing collection since they were created. Weakly consistent iterators do not throw ConcurrentModificationException.

*** The third group is concurrent collections using a special lock object (`java.util.concurrent.lock.Lock`):** This mechanism is more flexible than classical synchronization. This can be understood as following:

A lock has the same basic behavior as classical synchronization but a thread can also acquire it under special conditions: only if the lock is not currently held, or with a timeout, or if the thread is not interrupted.

Unlike synchronization code, in which an object lock is held while a code block or a method is executed, a `Lock` is held until its `unlock()` method is called. Some implementations make use of this mechanism to divide the collection into parts that can be separately locked, giving improved concurrency. For example, `LinkedBlockingQueue` has separate locks for the head and the tail ends of the queue, so that elements can be added and removed in parallel.

Other collections using this lock include `ConcurrentHashMap` and most of the implementations of `BlockingQueue`.

Collections in this group also have weakly consistent iterators and do not throw ConcurrentModificationException.

Let's summarize the most important points we've learned so far today:

- Most collections in the `java.util` package are not thread-safe in order to provide maximum performance in single-threaded applications. `Vector` and `Hashtable` are the only two legacy collections that are thread-safe.
- Synchronized collections can be created by using the `Collection` utility class' factory methods `synchronizedXXX(collection)`. They are thread-safe but poor at performance.
- Concurrent collections are improved for thread safety and performance with different synchronization mechanisms: copy-on-write, compare-and-swap and special lock. They are organized in `java.util.concurrent` package.

You can learn more about concurrent collections in my Java multi-threading/concurrency tutorials.

Other Java Collections Tutorials:

- Java Set Tutorial
- Java Map Tutorial
- Java List Tutorial and
- Java Queue Tutorial
- Understanding equals() and hashCode() in Java
- Understanding Object Ordering in Java with Comparable and Comparator
- 18 Java Collections and Generics Best Practices