ConcurrentHashMap

This lesson explains the working of the concurrent hash map and the other hash map data structures available in Java.

We'll cover the following

HashMaps and Concurrency#

`HashMap` is a commonly used data structure offering constant time access, however, it is not thread-safe. Consider the two methods `get()` and `put()` that get invoked by two different threads on an instance of `HashMap` in the following sequence:

```
1. Thread 1 invokes put() and inserts the key value pair ("myKe
y","item-1")

2. Thread 2 invokes get() but before get operation completes, th
e thread is context-switched

3. Thread 1 updates myKey with a new value say "item-2"

4. Thread 2 becomes active again but retrieves the stale key valu
e pair ("myKey","item-1")
```

The above scenario is naive and one may argue that if stale values are acceptable for an application then we may get away without using synchronization. However, that is not true. Consider the scenario when a `HashMap` reaches capacity and resizes. When a resize occurs the elements from the old structure are copied over to the new structure and a rehashing of the elements that hash to the same bucket and form a collision list may take place to reduce the size of the collision list. As one can imagine, there are several things that can go wrong when multiples threads operate on a map whilst the resize takes place. For instance, a reader thread might be iterating over the map when a rehashing occurs and a key that is present in the map is erroneously reported to be not present, since it was rehashed to a different bucket. When we throw into the mix all the other operation such as `remove` and `putIfAbsent`, it is easy to come up with sequences that demonstrate why `HashMap` is inherently thread unsafe. Compared to its synchronized cousins `HashTable` and `ConcurrentHashMap`, `HashMap` is lightweight and faster, but can only be used in single-threaded scenarios.

Using ConcurrentHashMaps#

`CocurrentHashMap` is a thread-safe class and multiple threads can operate on it in parallel without incurring any of the issues that a `HashMap` may suffer

it in parallel without incurring any of the issues that a `HashMap` may suffer from in a concurrent environment. For write operations the entire map is never locked rather only a segment of the map is locked. However, the retrieval or read operations generally don't involve locking at all. So in case of a read, the value set for a key by the most recently completed update operation is returned i.e. a completed update operation on a given key bears a *happens before* relationship with any (non-null) read operation. This does mean that a stale value may be returned if an update operation is in progress but not yet completed.

Since read operations can happen while update operations are on-going, any concurrent reads during the execution of aggregate operations such as `putAll()` or `clear()` may return insertion or removal of some of the entries respectively, when all or none are expected.

Another important detail is that `Iterators`, `SplitIterator`, or `Enumerations` for an instance of the `ConcurrentHashMap` represent the state or snapshot of the data structure at a point in time, specifically when they were created and don't throw the `ConcurrentModificationException` exception. Though the iterator itself is designed to be used by a single thread. See next lesson for details on `ConcurrentModificationException`.

Properties of ConcurrentHashMap#

Other notable properties of the `ConcurrentHashMap` class are listed below:

- `null` can't be inserted either as a key or a value.
- The `ConcurrentHashMap` shards its data into segments and the segments are locked individually when being written to. Each segment can be written independently of other segments allowing multiple threads to operate on the map object.
- The reads happen without locking for the majority of cases, thus making them synchronization-free and improving performance. However, note

that there are certain minority scenarios when reads have to go through synchronization.

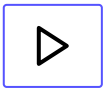
- In general, using keys that evaluate to the same hashCode will slow down the performance of any hash map.

Newbie Mistakes with ConcurrentHashMap#

One of the follies assumed when working with `ConcurrentHashMap` is to think that any accesses of and operations on the key/values within the data structure are somehow magically thread-safe. The map doesn't protect against external race conditions. Consider the program in the widget below that has two threads increment a key's value in a `ConcurrentHashMap` by a hundred times each. Run the program multiple times and see each run prints a different result.

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    public static void main( String args[] ) throws Exception {
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
        map.put("Biden", 0);

        ExecutorService es = Executors.newFixedThreadPool(5);

        // create a task to increment the vote count
        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++)
                    map.put("Biden", map.get("Biden") + 1);
            }
        };

        // submit the task twice
```

```
        Future future1 = es.submit(task);
        Future future2 = es.submit(task);

        // wait for the threads to finish
        future1.get();
```





We sort of cheated in the above program to make it fail. Consider the line:

```
map.put("Biden", map.get("Biden") + 1)
```

The above line is really three operations:

1. retrieval of the value
2. incrementing the value
3. updating the value

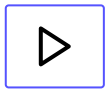
The right implementation should execute all the three steps together as a transaction or atomically to avoid synchronization issues. The takeaway is that a `ConcurrentHashMap` doesn't protect its constituents from race conditions but access to the data structure itself is thread safe.

Fixing with Atomic Integer#

One of the ways to fix the above program is to use instance of the `AtomicInteger` class as value. We'll invoke the `incrementAndGet()` method to register an increment on the value. The code is presented below:

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.atomic.AtomicInteger;

class Demonstration {

    public static void main( String args[] ) throws Exception {
```

```
        ConcurrentHashMap<String, AtomicInteger> map = new ConcurrentHashMap
        // create an atomic integer to keep the vote count
        AtomicInteger ai = new AtomicInteger(0);
        map.put("Biden", ai);

        ExecutorService es = Executors.newFixedThreadPool(5);

        // create a task to increment the vote count
        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++)
                    // We are ignoring the returned updated value from the
                    // function call
                    map.get("Biden").incrementAndGet();
            }
        };

```



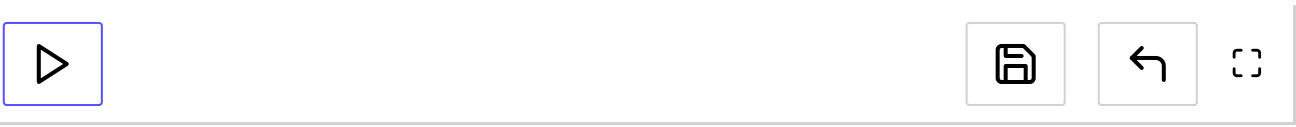
The count of the above program will always be 200 no matter how many times you run it. But this brings up another question: if we use `AtomicInteger` as value with the `HashMap` class would our program output the correct result? The answer is yes for this naive/simple program because the atomic integer itself is thread-safe so multiple threads attempting to increment it do so serially. However, the data structure i.e. the hash map itself is thread-unsafe and can exhibit concurrency bugs when multiple threads operate on it, traverse its keys or values, or when the map resizes. Remember, we have to think about concurrency both at the map level and at the key/value level.

Fixing with Custom Counter Class#

We could re-write the above program with a class that tracks the count and perform explicit synchronization ourselves instead of relying on the `AtomicInteger` class. Read the listing below:

```
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {

    // Class to keep track of vote count
    static class MyCounter {
        private int count = 0;

        void increment() {
            count++;
        }

        int getCount() {
            return count;
        }
    }

    public static void main( String args[] ) throws Exception {
        ConcurrentHashMap<String, MyCounter> map = new ConcurrentHashMap<>();
        map.put("Biden", new MyCounter());

        ExecutorService es = Executors.newFixedThreadPool(5);

        // create a task to increment the vote count
        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    MyCounter mc = map.get("Biden");

                    // explicit synchronization
                    synchronized (mc) {
                        mc.increment();
                    }
                }
            }
        };

        // submit the task twice
        Future future1 = es.submit(task);
        Future future2 = es.submit(task);

        // wait for the threads to finish
        future1.get();
        future2.get();
```

```
        // shutdown the executor service
        es.shutdown();


        System.out.println("votes for Biden = " + map.get("Biden").getCount());
    }
}
```



HashMap vs HashTable vs ConcurrentHashMap#

As a Java newbie when looking for an appropriate choice for a hash map you'll be confronted with the following options:

1. `HashMap`
2. `Hashtable`
3. `Collections.synchronizedMap(...)`
4. `ConcurrentHashMap`

As already discussed `HashMap` isn't thread-safe and if your scenario doesn't involve multiple threads, pick `HashMap` as it will provide the best performance. `Hashtable` is a synchronized map i.e. it can only be accessed by a single reader or writer thread at a time. `Hashtable` performs poorly in a highly concurrent environment. Iterators on a `Hashtable` observe *fail fast* behavior i.e. the iterators throw a `ConcurrentModificationException` if the `Hashtable` is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` method. `Hashtable` is part of Java's legacy code and was retrofitted. to implement the `Map` interface so that it could become part of the Java's collections framework.

`Collections.synchronizedMap(...)` creates an instance of the `SynchronizedMap` class (a private class in `Collections` class) backed by the passed-in map. The returned object synchronizes access to the backing map

passed in map. The returned object synchronizes access to the backing map allowing a single thread to operate on the map at a time. For example:

```
Map<String, Integer> map = Collections.synchronizedMap(new HashMap
<>());
```

The user must synchronize manually on the returned map object (not the backing map) when traversing any of the collection views using iterators, split iterators or streams.

A difference between `HashTable` and `Collections.synchronizedMap(...)` is that in case of `HashTable` all the methods are synchronized on the map object itself i.e. the `this` object. In case. of `Collections.synchronizedMap(...)` a separate object (not the backing map) is used for synchronization. But essentially both `HashTable` and `Collections.synchronizedMap(...)` have similar synchronization behavior. Note that a synchronized map backed by a `HashTable` simply adds a redundant synchronization layer and should never be used.

```
Collections.synchronizedMap(new Hashtable<>());
```

Finally, the `ConcurrentHashMap` uses sophisticated techniques to reduce the need for synchronization when multiple threads access the map in parallel. It is highly scalable and concurrent and can be iterated upon without requiring synchronization.

Property	HashMap	Hashtable	Collections.synchronizedMap(...)	ConcurrentHashMap
Null values/keys	yes	no	depends on backing map	no

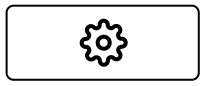
Property	HashMap	Hashtable	Collections.synchronizedMap(...)	ConcurrentHashMap
Thread-safe	no	yes	yes	yes
Lock Mechanism	not applicable	locks the entire map	locks the entire map	locks a segment of the map
Iterator	fail-fast	fail-fast	fail-fast	weakly consistent

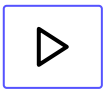
Performance#

We can run a simple and unsophisticated test to observe the performance of the different types of maps. The program in the widget below has 5 threads writing the same keys to a map with an initial capacity of 10. We time the run for each of the maps we discussed. The output demonstrates `ConcurrentHashMap` outperforms the other two maps and has a higher write throughput. Note, that this test is crude and doesn't take read performance into account but in general, the `ConcurrentHashMap` is the right choice in high concurrency environments.

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class Demonstration {
    public static void main( String args[] ) throws Exception {

        // start executor service
        ExecutorService es = Executors.newFixedThreadPool(5);

        performanceTest(new Hashtable<>(10), "Hashtable", es);
        performanceTest(Collections.synchronizedMap(new HashMap<>(10)), "Collection
        performanceTest(new ConcurrentHashMap<>(10), "Concurrent Hash Map", es);

        // shutdown the executor service
        es.shutdown();

    }

    static void performanceTest(Map<String, Integer> map, String mapName, ExecutorS
        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 1000000; i++)
                    map.put("key-" + i, i);
            }
        };

        long start = System.currentTimeMillis();

        Future future1 = es.submit(task);
        Future future2 = es.submit(task);
        Future future3 = es.submit(task);
        Future future4 = es.submit(task);
        Future future5 = es.submit(task);

        // wait for the threads to finish
        future1.get();
        future2.get();
        future3.get();
        future4.get();
        future5.get();

        long end = System.currentTimeMillis();

        System.out.println("Milliseconds taken using " + mapName + ": " + (end - st
```

```
    }
}
```






In some runs you may observe `ConcurrentHashMap` taking longer than other maps for the writes to complete. This is because the code executes in the cloud where execution environment (e.g. vCPUs, JVM version, compile-time optimization flags, memory etc) are beyond our control, however, if you ran the above program on a machine with multiple CPUs, the `ConcurrentHashMap` would outperform the other maps when executing majority of workloads.

Q Consider the following snippet:

```
Map map = Collections.synchronizedMap(new ConcurrentHash
Map<>());
```

Which of the following statements is true?

A) Snippet throws an exception because `ConcurrentHashMap` is already thread-safe.

B) `map` variable is an instance of the `ConcurrentHashMap` class.

C) `map` variable adds a layer of synchronization on the backing `ConcurrentHashMap`, effectively serializing access to the backing map one thread at a time and nullifying its ability to allow

multiple writer threads to operate on it at the same time.

Submit Answer

Reset Quiz

Interviewing soon? We've partnered with Hired so that companies apply to yo
utm_source=educative&utm_medium=lesson&utm_location=CA&utm_campa

Back

Concurrent Collections

Next

ConcurrentModificationException

Mark as Completed