How it works in java. ConcurrentHashMap.

Sergey Kuptsov · Follow
11 min read · Apr 5, 2017

202 2

The main principle of programming says not to reinvent the wheel. But sometimes in order to understand what is going on and how not to misuse instrument we need to do this. Today reinventing concurrent hash map.

At first we need 2 things. I start with 2 tests — one indicates that our concurrent map implementation has no data races(actually we need to check if our test correct also by testing incorrect implementation) and second one that we will use to test performance in terms of throughput under different profiles of load.

Let’s implement only some number of methods from Map interface:

```
public interface Map<K, V> {

    V put(K key, V value);

    V get(Object key);

    V remove(Object key);

    int size();
}
```

Thread-safety correctness test

It is practically impossible to write thread safety test exhaustively, you need to take into consideration all aspects defined in Chapter 17 of the JLS, more over it heavily depends on hardware memory models or JVM implementation.

For concurrency test let’s use one of stress testing frameworks such as jcstress that will heavy run your code trying to find data inconsistency. Although jcstress is still experimental it is better choice. Why it is hard to write your own concurrency test — look at Shipilev’s lecture.

Let’s start exploring jstress with jcstress-gradle-plugin. Full source code can be found in how-it-works-concurrent-map.

```
public class ConcurrentMapThreadSafetyTest {

    @State
    public static class MapState {
        final Map<String, Integer> map = new HashMap<>(3);

    }

    @JCStressTest
    @Description("Test race map get and put")
    @Outcome(id = "0, 1", expect = ACCEPTABLE, desc = "return 0L and
1L")
    @Outcome(expect = FORBIDDEN, desc = "Case violating atomicity.")
    public static class MapPutGetTest {

        @Actor
        public void actor1(MapState state, LongResult2 result) {
            state.map.put("A", 0);
            Integer r = state.map.get("A");
            result.r1 = (r == null ? -1 : r);
        }

        @Actor
        public void actor2(MapState state, LongResult2 result) {
            state.map.put("B", 1);
            Integer r = state.map.get("B");
            result.r2 = (r == null ? -1 : r);
        }
    }

    @JCStressTest
    @Description("Test race map check size")
    @Outcome(id = "2", expect = ACCEPTABLE, desc = "size of map = 2
")
    @Outcome(id = "1", expect = FORBIDDEN, desc = "size of map = 1 is
race")
    @Outcome(expect = FORBIDDEN, desc = "Case violating atomicity.")
    public static class MapSizeTest {

        @Actor
        public void actor1(MapState state) {
            state.map.put("A", 0);
        }

        @Actor
        public void actor2(MapState state) {
            state.map.put("B", 0);
        }

        @Arbiter
        public void arbiter(MapState state, IntResult1 result) {
            result.r1 = state.map.size();
        }
    }
}
```

In first test MapPutGetTest we have two threads executing concurrently methods actor1 and actor2 respectively, both of them put some value to map, and checking them back, if there is no data race, both threads must see setted values.

In second MapSizeTest we concurrently put some different keys to map and after all checking the size of map — if there is no data race — the result must be 2.

We must check if this test correct both on non-thread-safe HashMap — we must observe atomicity violation and thread-safe ConcurrentHashMap — we must not see alternative results.

1. Results with non-thread-safe HashMap

```
[FAILED]
ru.skuptsov.concurrent.map.test.ConcurrentMapTest.MapPutGetTest
  Observed state   Occurrences   Expectation  Interpretation
           -1, 1       293,867     FORBIDDEN  Case violating atomic
           0, -1       282,190     FORBIDDEN  Case violating atomic
            0, 1    28,013,763     ACCEPTABLE  return 0 and 1

[FAILED]
ru.skuptsov.concurrent.map.test.ConcurrentMapTest.MapSizeTest
  Observed state   Occurrences   Expectation  Interpretation
               1     1,434,783    FORBIDDEN   size of map = 1 race
               2    11,733,097    ACCEPTABLE  size of map = 2
```

In not thread-safe HashMap we see some statistical number of inconsistent result, all 2 test failed

2. Results with thread-safe ConcurrentHashMap

```
[OK] ru.skuptsov.concurrent.map.test.ConcurrentMapTest.MapPutGetTest
  Observed state   Occurrences   Expectation  Interpretation
            0, 1    20,195,000     ACCEPTABLE

[OK] ru.skuptsov.concurrent.map.test.ConcurrentMapTest.MapSizeTest
  Observed state   Occurrences   Expectation  Interpretation
               2    6,573,730    ACCEPTABLE  size of map = 2
```

ConcurrentHashMap passed test, at least we can admit than our test can detect some simple concurrency issues. Same results for Collection.synchronizedMap and HashTable.

First concurrent hash map attempt

The first naive approach is to simply synchronize every access to internal structures of map(array of buckets).

Actually we can write some concurrent wrapper over given Map provider — so does java.util.Collections#synchronizedMap, Hashtable and guava's SynchronizedMultimap.

```
public class GeneralMonitorSynchronizedHashMap<K, V> extends
BaseMap<K, V> implements Map<K, V>, IMap<K, V> {

    private final Map<K, V> provider;
    private final Object monitor;

    public SynchronizedHashMap(Map<K, V> provider) {
        this.provider = provider;
        monitor = this;

    }

    @Override
    public V put(K key, V value) {
        synchronized (monitor) {
            return provider.put(key, value);
        }
    }

    @Override
    public V get(Object key) {
        synchronized (monitor) {
            return provider.get(key);
        }
    }

    @Override
    public int size() {
        synchronized (monitor) {
            return provider.size();
        }
    }
}
```

Changes to non-volatile provider will be visible between threads, according to documentation:

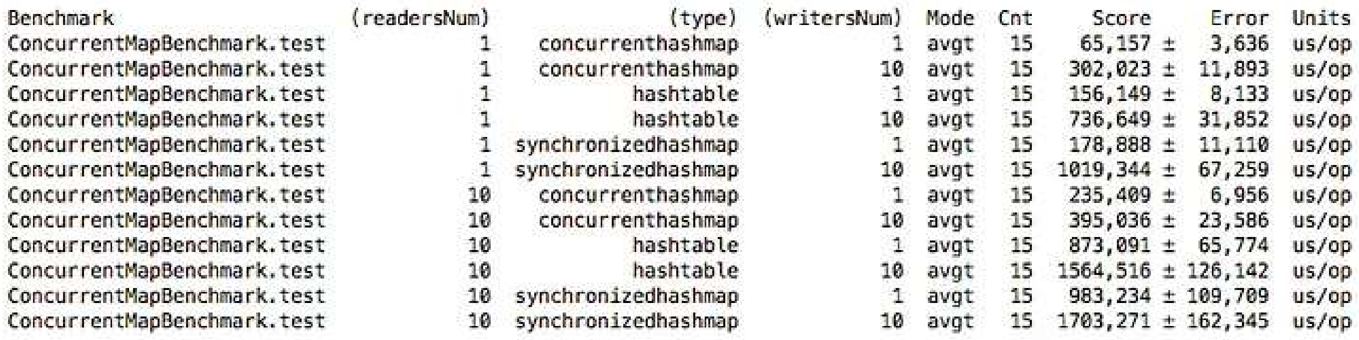
> *Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.*

Our SynchronizedHashMap passes concurrent tests but at what cost? in every method there can be only one thread at the same time even if we work with different keys, so in multithreaded load we must expect some performance penalty. Let’s measure it.

Performance benchmark test

For performance test we will use jmh. In performance benchmark w’ll do not test non-thread-safe implementations.

```
@State(Scope.Thread)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(MICROSECONDS)
public class ConcurrentMapBenchmark {
    private Map<Integer, Integer> map;

    @Param({"concurrenthashmap", "hashtable", "synchronizedhashmap"})
    private String type;

    @Param({"1", "10"})
    private Integer writersNum;

    @Param({"1", "10"})
    private Integer readersNum;

    private final static int NUM = 1000;

    @Setup
    public void setup() {
        switch (type) {
            case "hashtable":
                map = new Hashtable<>();
                break;
            case "concurrenthashmap":
                map = new ConcurrentHashMap<>();
                break;
            case "synchronizedhashmap":
                map = new SynchronizedHashMap<>(new HashMap<>());
                break;
        }
    }

    @Benchmark
    public void test(Blackhole bh) throws ExecutionException,
InterruptedException {

        List<CompletableFuture> futures = new ArrayList<>();

        for (int i = 0; i < writersNum; i++) {
            futures.add(CompletableFuture.runAsync(() -> {
                for (int j = 0; j < NUM; j++) {
                    map.put(j, j);
                }
            }));
        }

        for (int i = 0; i < readersNum; i++) {
            futures.add(CompletableFuture.runAsync(() -> {
                for (int j = 0; j < NUM; j++) {
                    bh.consume(map.get(j));
                }
            }));
        }

        CompletableFuture.allOf(futures.toArray(new
CompletableFuture[1])).get();
    }
}
```



```
Benchmark                   (readersNum)              (type)  (writersNum)  Mode  Cnt      Score     Error  Units
ConcurrentMapBenchmark.test            1   concurrenthashmap             1  avgt   15     65,157 ±   3,636  us/op
ConcurrentMapBenchmark.test            1   concurrenthashmap            10  avgt   15    302,023 ±  11,893  us/op
ConcurrentMapBenchmark.test            1           hashtable             1  avgt   15    156,149 ±   8,133  us/op
ConcurrentMapBenchmark.test            1           hashtable            10  avgt   15    736,649 ±  31,852  us/op
ConcurrentMapBenchmark.test            1 synchronizedhashmap             1  avgt   15    178,888 ±  11,110  us/op
ConcurrentMapBenchmark.test            1 synchronizedhashmap            10  avgt   15   1019,344 ±  67,259  us/op
ConcurrentMapBenchmark.test           10   concurrenthashmap             1  avgt   15    235,409 ±   6,956  us/op
ConcurrentMapBenchmark.test           10   concurrenthashmap            10  avgt   15    395,036 ±  23,586  us/op
ConcurrentMapBenchmark.test           10           hashtable             1  avgt   15    873,091 ±  65,774  us/op
ConcurrentMapBenchmark.test           10           hashtable            10  avgt   15   1564,516 ± 126,142  us/op
ConcurrentMapBenchmark.test           10 synchronizedhashmap             1  avgt   15    983,234 ± 109,709  us/op
ConcurrentMapBenchmark.test           10 synchronizedhashmap            10  avgt   15   1703,271 ± 162,345  us/op
```

We checked that performance of our SynchronizedHashMap is pretty much the same at java's HashTable and it is 2 times worse than ConcurrentHashMap. Let's try to increase it.

Lock striping concurrent hash map attempt

The first improvement can come from the idea that instead of blocking access to the whole map it's better block thread access only to the same bucket where bucket array index = key.hashCode() % array.length. This technique is called lock striping or fine-grained synchronization, more other techniques here «The Art of Multiprocessor Programming».

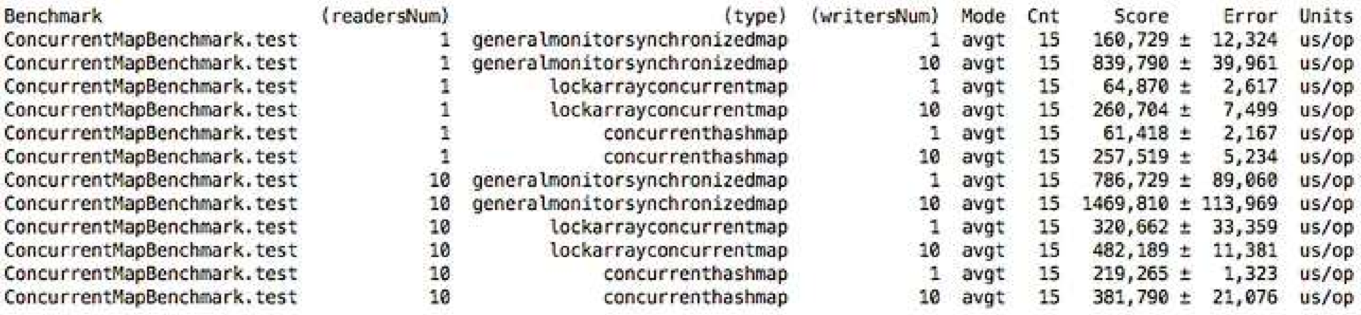
For array of buckets we need an array of locks, at start the size of locks array must be equal to internal array size — it is important because we don't want situation where 2 locks are responsible for one array slot.

For simplicity we will design not resizable Map — that means that we cannot extend initial capacity(if N >> initialCapacity we will loose O(1) map gurantee. Also we do not need loadFactor) — resizable concurrent map is separate big topic.

```
public class LockStripingArrayConcurrentHashMap<K, V> extends
BaseMap<K, V> implements Map<K, V> {

    private final AtomicInteger count = new AtomicInteger(0);
    private final Node<K, V>[] buckets;
    private final Object[] locks;

    @SuppressWarnings({"rawtypes", "unchecked"})
    public LockStripingArrayConcurrentHashMap(int capacity) {
        locks = new Object[capacity];
        for (int i = 0; i < locks.length; i++) {
            locks[i] = new Object();
        }

        buckets = (Node<K, V>[]) new Node[capacity];
    }

    @Override
    public int size() {
        return count.get();
    }

    @Override
    public V get(Object key) {
        if (key == null) throw new IllegalArgumentException();
        int hash = hash(key);
        synchronized (getLockFor(hash)) {
            Node<K, V> node = buckets[getBucketIndex(hash)];

            while (node != null) {
                if (isKeyEquals(key, hash, node)) {
                    return node.value;
                }

                node = node.next;
            }

            return null;
        }
    }

    @Override
    public V put(K key, V value) {
        if (key == null || value == null) throw new
IllegalArgumentException();
        int hash = hash(key);
        synchronized (getLockFor(hash)) {
            int bucketIndex = getBucketIndex(hash);
            Node<K, V> node = buckets[bucketIndex];

            if (node == null) {
                buckets[bucketIndex] = new Node<>(hash, key, value,
null);
                count.incrementAndGet();
                return null;
            } else {
                Node<K, V> prevNode = node;
                while (node != null) {
                    if (isKeyEquals(key, hash, node)) {
                        V prevValue = node.value;
                        node.value = value;

                        return prevValue;
                    }

                    prevNode = node;
                    node = node.next;
                }

                prevNode.next = new Node<>(hash, key, value, null);
                count.incrementAndGet();
                return null;
            }          ...
        }
    }

    private boolean isKeyEquals(Object key, int hash, Node<K, V>
node) {
        return node.hash == hash &&
                node.key == key ||
                (node.key != null && node.key.equals(key));
    }

    private int hash(Object key) {
        return key.hashCode();
    }

    private int getBucketIndex(int hash) {
        return hash % buckets.length;
    }

    private Object getLockFor(int hash) {
        return locks[hash % locks.length];
    }

    private static class Node<K, V> {
        final int hash;
        K key;
        V value;
        Node<K, V> next;

        Node(int hash, K key, V value, Node<K, V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }
}
```

It is important that all out class fields are final cause only final and static fields guarantee safe publication through constructor.

Source code can be found here. Benchmark results:



```
Benchmark                  (readersNum)                       (type)  (writersNum)  Mode  Cnt     Score     Error  Units
ConcurrentMapBenchmark.test           1  generalmonitorsynchronizedmap             1  avgt   15   160,729 ±  12,324  us/op
ConcurrentMapBenchmark.test           1  generalmonitorsynchronizedmap            10  avgt   15   839,790 ±  39,961  us/op
ConcurrentMapBenchmark.test           1         lockarrayconcurrentmap             1  avgt   15    64,870 ±   2,617  us/op
ConcurrentMapBenchmark.test           1         lockarrayconcurrentmap            10  avgt   15   260,704 ±   7,499  us/op
ConcurrentMapBenchmark.test           1              concurrenthashmap             1  avgt   15    61,418 ±   2,167  us/op
ConcurrentMapBenchmark.test           1              concurrenthashmap            10  avgt   15   257,519 ±   5,234  us/op
ConcurrentMapBenchmark.test          10  generalmonitorsynchronizedmap             1  avgt   15   786,729 ±  89,060  us/op
ConcurrentMapBenchmark.test          10  generalmonitorsynchronizedmap            10  avgt   15  1469,810 ± 113,969  us/op
ConcurrentMapBenchmark.test          10         lockarrayconcurrentmap             1  avgt   15   320,662 ±  33,359  us/op
ConcurrentMapBenchmark.test          10         lockarrayconcurrentmap            10  avgt   15   482,189 ±  11,381  us/op
ConcurrentMapBenchmark.test          10              concurrenthashmap             1  avgt   15   219,265 ±   1,323  us/op
ConcurrentMapBenchmark.test          10              concurrenthashmap            10  avgt   15   381,790 ±  21,076  us/op
```

We can see that fine-grained synchronized implementation is better than our overall lock. Results for when there are one reader and one writer comparing with concurrent hash map are practically the same but when number of threads increases — the difference is bigger, especially where there are a lot of readers.

Lock free concurrent hash map attempt

Frankly speaking, synchronizing is not a parrallel programming technique cause it sets up threads in serial queue to wait for another thread to complete. And additional system cost of synchronization context switching increasing as high as number of waiting threads grows but all we want is to make small number of instructions to change map's key value.

Let's rule some requirments to new hash map implementation that will improve our realization. And requirements are:

1. If we 2 threads that work with different keys(write or read) we do not want any kind of synchronization between them(cause word tearing is impossible in java — and access to two diff array fields is safe)
2. If multiple threads work on the same key(write and read) we do not want cache interleave(more about cache structure) and need safe happens-before guarantees for access between threads otherwise one thread might not see changed value by other thread. But we do not want to block read thread and wait for write thread to complete .
3. We do not want to block multiple readers to the same key if there is no one writer among them.

Let's concetrate on item 2 and 3. Actually we can make map read operation fully lock-free if we can make (1) volatile read array of buckets and then traverse inside bucket by linked list with (2) volatile read of next Node and value of Node itself which.

For (2) we can just mark Node's next and value fields volatile.

For the (1) there is no such thing as volatile array, even if it is declared as volatile, it not provides volatile semantics when reading or writing elements, concurrent accessing the k-th element of the array requires an explicit volatile read, volatile is only link to array. We can use AtomicReferenceArray for this purpose but it accepts only Object[] arrays. As alternative we can use Unsafe for volatile array read and lock-free write. The same technique is used in AtomicReferenceArray and ConcurrentHashMap.

```
@SuppressWarnings("unchecked")
// read array value by index
private <K, V> Node<K, V> volatileGetNode(int i) {
    return (Node<K, V>) U.getObjectVolatile(buckets, ((long) i <<
ASHIFT) + ABASE);
}

// cas set array value by index
private <K, V> boolean compareAndSwapNode(int i, Node<K, V>
expectedNode, Node<K, V> setNode) {
    return U.compareAndSwapObject(buckets, ((long) i << ASHIFT) +
ABASE, expectedNode, setNode);
}

private static final sun.misc.Unsafe U;
// Node[] header shift
private static final long ABASE;
// Node.class size shift
private static final int ASHIFT;

static {
    try {
```

```
// get unsafe by reflection - it is illegal to use not in java lib
Constructor<Unsafe> unsafeConstructor =
Unsafe.class.getDeclaredConstructor();
        unsafeConstructor.setAccessible(true);
        U = unsafeConstructor.newInstance();
    } catch (NoSuchMethodException | InstantiationException |
InvocationTargetException | IllegalAccessException e) {
        throw new RuntimeException(e);
    }

    Class<?> ak = Node[].class;

    ABASE = U.arrayBaseOffset(ak);
    int scale = U.arrayIndexScale(ak);
    ASHIFT = 31 - Integer.numberOfLeadingZeros(scale);
}
```

In volatileGetNode we can now read values safely with memory barier.

Let’s now write lock-free map V get(Object key) method:

```
public V get(Object key) {
    if (key == null) throw new IllegalArgumentException();
    int hash = hash(key);
    Node<K, V> node;

    // volatile read of bucket head at hash index
    if ((node = volatileGetNode(getBucketIndex(hash))) != null) {
        // check first node
        if (isKeyEquals(key, hash, node)) {
            return node.value;
        }

        // walk through the rest to find target node
        while ((node = node.next) != null) {
            if (isKeyEquals(key, hash, node))
                return node.value;
        }
    }

    return null;
}
```

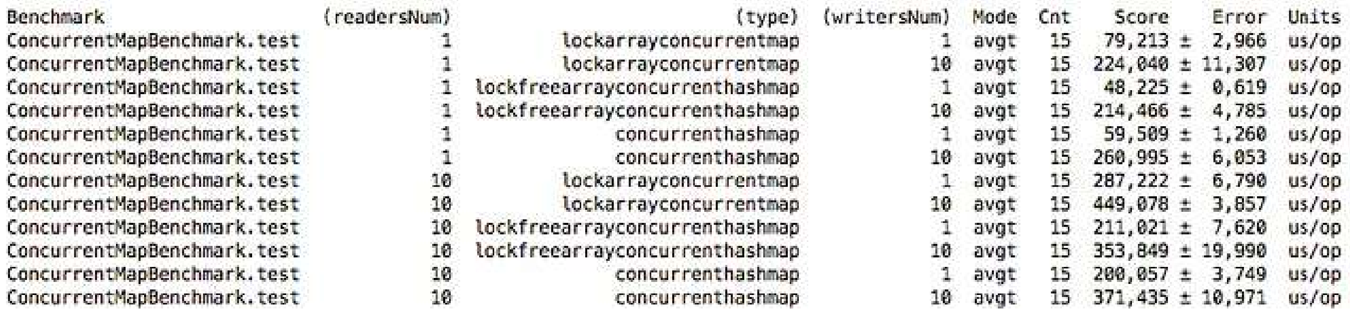
We can check if we are really lock-free, according to this.

In our first attempt we had big memory overhead with locks pool array — actually we can use the same fine-grained with additional memory — just lock on first node if it is exists. If it does not exists — we cannot block and need some lock-free method to set head of buckets — we already mentioned it above — method compareAndSwapNode

```
@Override
public V put(K key, V value) {
    if (key == null || value == null) throw new
IllegalArgumentException();
    int hash = hash(key);
    // no resize in this implementation - so the index will not
change
    int bucketIndex = getBucketIndex(hash);

    // cas loop trying not to miss
    while (true) {
        Node<K, V> node;
        // if bucket is empty try to set new head with cas
        if ((node = volatileGetNode(bucketIndex)) == null) {
            if (compareAndSwapNode(bucketIndex, null,
                    new Node<>(hash, key, value, null))) {
                // if we succeed to set head - then break and return
null

                count.increment();
                break;
            }
        } else {
            // head is not null - try to find place to insert or
update under lock
            synchronized (node) {
                // check if node have not been changed since we got
it
                // otherwise let's go to another loop iteration
                if (volatileGetNode(bucketIndex) == node) {
                    V prevValue = null;
                    Node<K, V> n = node;
                    while (true) {
                        ... simply walk through list under lock and
update or insert value...
                    }

                    return prevValue;
                }
            }
        }
    }

    return null;
}
```

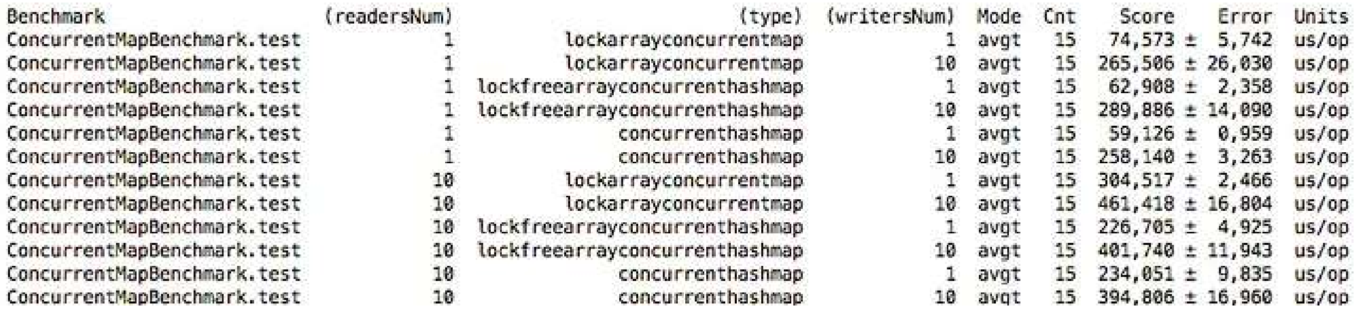
Full source code here.

Let’s benchmark it



```
Benchmark                     (readersNum)                           (type)  (writersNum)  Mode  Cnt    Score    Error  Units
ConcurrentMapBenchmark.test              1          lockarrayconcurrentmap             1  avgt   15   79,213 ±  2,966  us/op
ConcurrentMapBenchmark.test              1          lockarrayconcurrentmap            10  avgt   15  224,040 ± 11,307  us/op
ConcurrentMapBenchmark.test              1  lockfreearrayconcurrenthashmap             1  avgt   15   48,225 ±  0,619  us/op
ConcurrentMapBenchmark.test              1  lockfreearrayconcurrenthashmap            10  avgt   15  214,466 ±  4,785  us/op
ConcurrentMapBenchmark.test              1               concurrenthashmap             1  avgt   15   59,509 ±  1,260  us/op
ConcurrentMapBenchmark.test              1               concurrenthashmap            10  avgt   15  260,995 ±  6,053  us/op
ConcurrentMapBenchmark.test             10          lockarrayconcurrentmap             1  avgt   15  287,222 ±  6,790  us/op
ConcurrentMapBenchmark.test             10          lockarrayconcurrentmap            10  avgt   15  449,078 ±  3,857  us/op
ConcurrentMapBenchmark.test             10  lockfreearrayconcurrenthashmap             1  avgt   15  211,021 ±  7,620  us/op
ConcurrentMapBenchmark.test             10  lockfreearrayconcurrenthashmap            10  avgt   15  353,849 ± 19,990  us/op
ConcurrentMapBenchmark.test             10               concurrenthashmap             1  avgt   15  200,057 ±  3,749  us/op
ConcurrentMapBenchmark.test             10               concurrenthashmap            10  avgt   15  371,435 ± 10,971  us/op
```

We are even better than ConcurrentHashMap in some cases — but it is not a fair competition —cause ConcurrentHashMap do lazy table initialization during load and at least one resize cause resize number threshold = initialCapacity * loadFactor. If we run test again with initialCapacity != N inserted elements (=N/6)— results are slightly different.



```
Benchmark                     (readersNum)                           (type)  (writersNum)  Mode  Cnt    Score    Error  Units
ConcurrentMapBenchmark.test              1          lockarrayconcurrentmap             1  avgt   15   74,573 ±  5,742  us/op
ConcurrentMapBenchmark.test              1          lockarrayconcurrentmap            10  avgt   15  265,506 ± 26,030  us/op
ConcurrentMapBenchmark.test              1  lockfreearrayconcurrenthashmap             1  avgt   15   62,908 ±  2,358  us/op
ConcurrentMapBenchmark.test              1  lockfreearrayconcurrenthashmap            10  avgt   15  289,886 ± 14,090  us/op
ConcurrentMapBenchmark.test              1               concurrenthashmap             1  avgt   15   59,126 ±  0,959  us/op
ConcurrentMapBenchmark.test              1               concurrenthashmap            10  avgt   15  258,140 ±  3,263  us/op
ConcurrentMapBenchmark.test             10          lockarrayconcurrentmap             1  avgt   15  304,517 ±  2,466  us/op
ConcurrentMapBenchmark.test             10          lockarrayconcurrentmap            10  avgt   15  461,418 ± 16,804  us/op
ConcurrentMapBenchmark.test             10  lockfreearrayconcurrenthashmap             1  avgt   15  226,705 ±  4,925  us/op
ConcurrentMapBenchmark.test             10  lockfreearrayconcurrenthashmap            10  avgt   15  401,740 ± 11,943  us/op
ConcurrentMapBenchmark.test             10               concurrenthashmap             1  avgt   15  234,051 ±  9,835  us/op
ConcurrentMapBenchmark.test             10               concurrenthashmap            10  avgt   15  394,806 ± 16,960  us/op
```

This is because in ConcurrentHashMap we made resize during test and get element by key spends less time walking through bucket linked list.

Frankly speaking what we received is not a completely non-locking data structure — so does ConcurrentHashMap — but all we need is just to have a lock-free linked list — but with resizing and concurrent modification it is not so easy task — read here.

Original java 8 ConcurrentHashMap has a number of small improvements we did not mentioned sush as

1. Lazy table initialization that minimizes footprint until first use
2. Concurrent resizing array of buckets
3. Element count is maintained using a specialization of LongAdder which is one is well under high contention
4. Special types of bucket nodes(since 1.8) — TreeBins if the length of bucket list grows more than TREEIFY_THRESHOLD = 8 — bin become balanced tree with worst case key search (O(log(Nbucket_size)))

Needless to say that implementation of ConcurrentHashMap in java 1.8 was significally changed since 1.7. In 1.7 it was an idea of Segments where number of segment equals concurrencyLevel. Java 8 represents with a single array.

Java Concurrency Threads

Written by Sergey Kuptsov

117 Followers · 20 Following

Follow

Responses (2)

What are your thoughts?

Respond

Ilya Kharlamov
over 6 years ago

return locks[hash % locks.length];

this is obviously incorrect

since hash can be negative this will return a negative index

should be

return locks[hash & (locks.length-1)];

1 Reply

Mohan Radhakrishnan
over 5 years ago

•••

“Special types of bucket nodes(since 1.8) — TreeBins if the length of bucket list grows more than TREEIFY_THRESHOLD = 8 — bin become balanced tree with worst case key search (O(log(Nbucket_size)))”

Are to referring to a red-black tree or something……

Read More

Reply

More from Sergey Kuptsov

Sergey Kuptsov

How it works in java. CompletableFuture.

The main principle of programming says not to reinvent the wheel. But sometimes in orde…

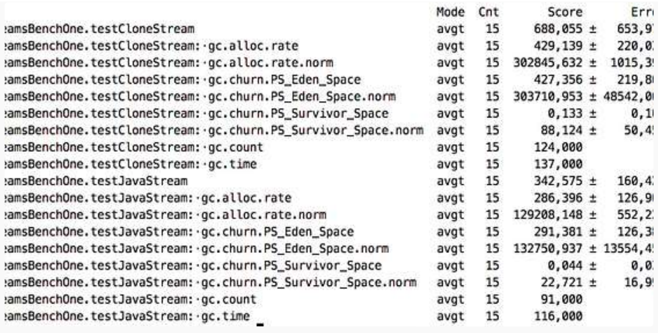
Nov 18, 2017 •••

Sergey Kuptsov

How it works in java. Thread Pool.

The main principle of programming says not to reinvent the wheel. But sometimes in orde…

Mar 4, 2017 •••



```
                                                        Mode  Cnt        Score      Err
amsBenchOne.testCloneStream                             avgt   15      688,055 ±   653,9
amsBenchOne.testCloneStream:·gc.alloc.rate              avgt   15      429,139 ±   220,0
amsBenchOne.testCloneStream:·gc.alloc.rate.norm         avgt   15   302845,632 ±  1015,3
amsBenchOne.testCloneStream:·gc.churn.PS_Eden_Space     avgt   15      427,356 ±   219,8
amsBenchOne.testCloneStream:·gc.churn.PS_Eden_Space.norm avgt  15   303710,953 ± 48542,0
amsBenchOne.testCloneStream:·gc.churn.PS_Survivor_Space avgt   15        0,133 ±     0,1
amsBenchOne.testCloneStream:·gc.churn.PS_Survivor_Space.norm avgt 15    88,124 ±    50,4
amsBenchOne.testCloneStream:·gc.count                   avgt   15      124,000
amsBenchOne.testCloneStream:·gc.time                    avgt   15      137,000
amsBenchOne.testJavaStream                              avgt   15      342,575 ±   160,4
amsBenchOne.testJavaStream:·gc.alloc.rate               avgt   15      286,396 ±   126,9
amsBenchOne.testJavaStream:·gc.alloc.rate.norm          avgt   15   129208,148 ±   552,2
amsBenchOne.testJavaStream:·gc.churn.PS_Eden_Space      avgt   15      291,381 ±   126,3
amsBenchOne.testJavaStream:·gc.churn.PS_Eden_Space.norm avgt   15   132750,937 ± 13554,4
amsBenchOne.testJavaStream:·gc.churn.PS_Survivor_Space  avgt   15        0,044 ±     0,0
amsBenchOne.testJavaStream:·gc.churn.PS_Survivor_Space.norm avgt 15     22,721 ±    16,9
amsBenchOne.testJavaStream:·gc.count                    avgt   15       91,000
amsBenchOne.testJavaStream:·gc.time                     avgt   15      116,000
```

Sergey Kuptsov

How it works in java. Streams.

The main principle of programming says not to reinvent the wheel. But sometimes in orde...

Feb 25, 2018 •••

Sergey Kuptsov

GraphQL vs REST

In web service with 2 separate teams—front and back— we have a communication...

Jan 9, 2019 •••

See all from Sergey Kuptsov

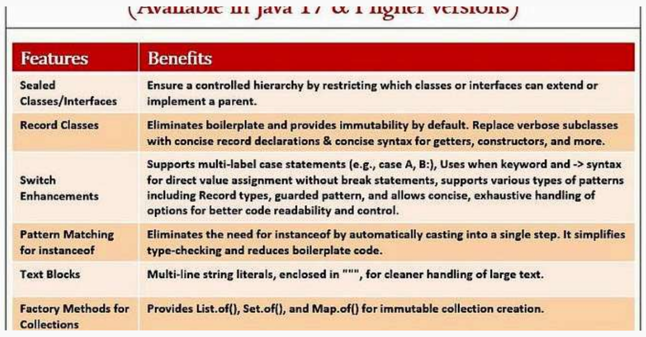
Recommended from Medium

In Stackademic by Abhinav

Is Java Really Dead? The Surprising Truth Behind Its Undefeated...

The world of programming is ever-evolving, with new languages, frameworks, and...

✦ Dec 20, 2024 •••



(Available in java 17 & Higher versions)

Features	Benefits
Sealed Classes/Interfaces	Ensure a controlled hierarchy by restricting which classes or interfaces can extend or implement a parent.
Record Classes	Eliminates boilerplate and provides immutability by default. Replace verbose subclasses with concise record declarations & concise syntax for getters, constructors, and more.
Switch Enhancements	Supports multi-label case statements (e.g., case A, B:), Uses when keyword and -> syntax for direct value assignment without break statements, supports various types of patterns including Record types, guarded pattern, and allows concise, exhaustive handling of options for better code readability and control.
Pattern Matching for instanceof	Eliminates the need for instanceof by automatically casting into a single step. It simplifies type-checking and reduces boilerplate code.
Text Blocks	Multi-line string literals, enclosed in """, for cleaner handling of large text.
Factory Methods for Collections	Provides List.of(), Set.of(), and Map.of() for immutable collection creation.

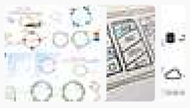
javatechonline blog

Java Interview Coding Problems Using Java 17 vs. Java 8 Features

In this post, we will explore some important Java Coding Interview questions with their...

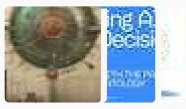
✦ Dec 8, 2024 •••

Lists



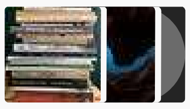
General Coding Knowledge

20 stories · 1849 saves



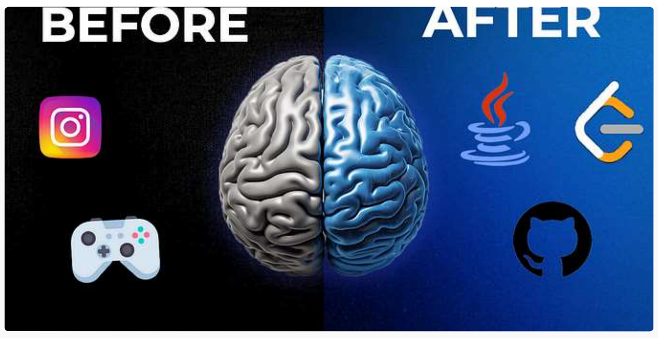
data science and AI

40 stories · 310 saves



Staff picks

793 stories · 1553 saves



In AlgoMaster.io by Ashish Pratap Singh

How I Tricked My Brain to Be Addicted to Coding

The Dopamine Hack

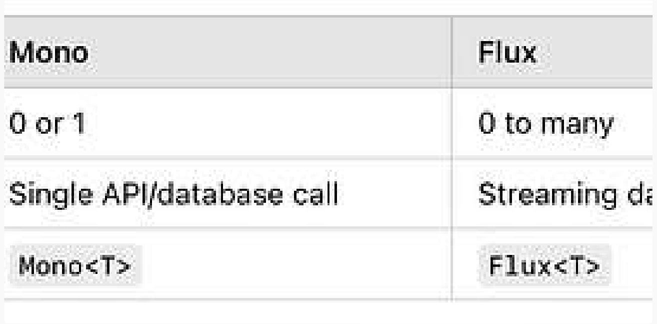
Nov 15, 2024

Full Stack Developer

Spring doesn't recommend @Autowired anymore???

There are 3 ways we can inject dependencies and Field Injection is not recommended...

Jul 19, 2024



Mono	Flux
0 or 1	0 to many
Single API/database call	Streaming da
`Mono<T>`	`Flux<T>`

Shishir Kumar

Reactive Programming in Java: Mono and Flux with Spring Boot

Introduction

Sep 12, 2024

Engineering Digest

Multithreading in Java

CPU

Aug 2, 2024

See more recommendations