Java Best Practices – Queue battle and the Linked ConcurrentHashMap

javacodegeeks.com Byron Kiourtzoglou September 21st, 2010 view original

Continuing our series of articles concerning proposed practices while working with the Java programming language, we are going to perform a performance comparison between four popular Queue implementation classes with relevant semantics. To make things more realistic we are going to test against a multi–threading environment so as to discuss and demonstrate how to utilize ArrayBlockingQueue, ConcurrentLinkedQueue, LinkedBlockingQueue and/or LinkedList for high performance applications.

Last but not least we are going to provide our own implementation of a ConcurrentHashMap. The ConcurrentLinkedHashMap implementation differs from ConcurrentHashMap in that it maintains a doubly–linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map. The ConcurrentLinkedHashMap benefits from the combined characteristics of ConcurrentHashMap and LinkedHashMap achieving performance just slightly below that of ConcurrentHashMap due to the added expense of maintaining the linked list.

All discussed topics are based on use cases derived from the development of mission critical, ultra high performance production systems for the telecommunication industry.

Prior reading each section of this article it is highly recommended that you consult the relevant Java API documentation for detailed information and code samples.

All tests are performed against a Sony Vaio with the following characteristics :

- System : openSUSE 11.1 (x86_64)
- Processor (CPU) : Intel(R) Core(TM)2 Duo CPU T6670 @ 2.20GHz

- Processor Speed : 1,200.00 MHz
- Total memory (RAM) : 2.8 GB
- Java : OpenJDK 1.6.0_0 64-Bit

The following test configuration is applied :

- Concurrent worker Threads : 50
- Test repeats per worker Thread : 100
- Overall test runs : 100

ArrayBlockingQueue vs ConcurrentLinkedQueue vs LinkedBlockingQueue vs LinkedList

One of the most common tasks a Java developer has to implement is storing and retrieving objects from Collections. The Java programming language provides a handful of Collection implementation classes with both overlapping and unique characteristics. Since Java 1.5 the Queue implementation classes became the de–facto standard for holding elements prior processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Nevertheless working with Queue implementation classes, especially in a multi–threading environment, can by tricky. The majority of them provide concurrent access by default but concurrency can be treated in a blocking or non – blocking manner. A BlockingQueue implementation class supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

The case scenarios that will be discussed here are having multiple Threads inserting, retracting and iterating through elements of ConcurrentLinkedQueue and LinkedList Queue implementation classes and ArrayBlockingQueue and LinkedBlockingQueue BlockingQueue implementation classes. We are going to demonstrate how to properly utilize the aforementioned collection implementation classes in a multi–threading environment and provide relevant performance comparison charts so as to show which one performs better in every test case.

To make a fare comparison we will assume that no NULL elements are allowed and that the size of every queue is limited where it is applicable. Thus the BlockingQueue implementation classes of our test group

will be initialized with a maximum size of 5000 elements – remember that we will be using 50 worker Threads performing 100 test repeats each. Furthermore since LinkedList is the only Queue implementation class of our test group that does not provide concurrent access by default, concurrency for LinkedList will be achieved using a synchronized block to access the list. At this point we must pinpoint that the Java documentation for the LinkedList Collection implementation class proposes the use of *Collections.synchronizedList* static method in order to maintain concurrent access to the list. This method provides a "wrapped" synchronized instance of the designated Collection implementation class as shown below :

```
List syncList = Collections.synchronizedList(new LinkedList());
```

This approach is appropriate when you want to use the specific implementation class as a List rather than a Queue. To be able to use the "queue" functionality of the specific Collection implementation class you must utilize it as is, or cast it to a Queue interface.

Test case #1 – Adding elements in a queue

For the first test case we are going to have multiple Threads adding String elements in each Queue implementation class. In order to maintain uniqueness among String elements, we will construct them as shown below :

- A static first part e.g. "helloWorld"
- The worker Thread id, remember that we have 50 worker Threads running concurrently
- The worker Thread test repeat number, remember that each worker thread performs 100 test repeats for each test run

For every test run each worker Thread will insert 100 String elements, as shown below :

- For the first test repeat
- - Worker Thread #1 will insert the String element : "helloWorld-1-1"
  - Worker Thread #2 will insert the String element : "helloWorld-2-1"
  - Worker Thread #3 will insert the String element : "helloWorld-3-1"

- etc …

- For the second test repeat
- - Worker Thread #1 will insert the String element : “helloWorld-1-2”
  - Worker Thread #2 will insert the String element : “helloWorld-2-2”
  - Worker Thread #3 will insert the String element : “helloWorld-3-2”
  - etc …
- etc …

At the end of each test run every Queue implementation class will be populated with 5000 distinct String elements. For adding elements we will be utilizing the “put()” operation for the BlockingQueue implementation classes and the “offer()” operation for the Queue implementation classes as shown below :

- arrayBlockingQueue.put(“helloWorld-” + id + “-” + count);
- linkedBlockingQueue.put(“helloWorld-” + id + “-” + count);
- concurrentLinkedQueue.offer(“helloWorld-” + id + “-” + count);
- synchronized(linkedList) {
  linkedList.offer(“helloWorld-” + id + “-” + count);
  }

Below we present a performance comparison chart between the four aforementioned Queue implementation classes

The horizontal axis represents the number of test runs and the vertical axis the average transactions per second (TPS) for each test run. Thus higher values are better. As you can see all Queue implementation classes performed almost identically when adding elements to them. ArrayBlockingQueue and ConcurrentLinkedQueue performed slightly better compared to LinkedList and LinkedBlockingQueue. The latter was the worst performer scoring 625000 TPS on average.

Test case #2 – Removing elements from a queue

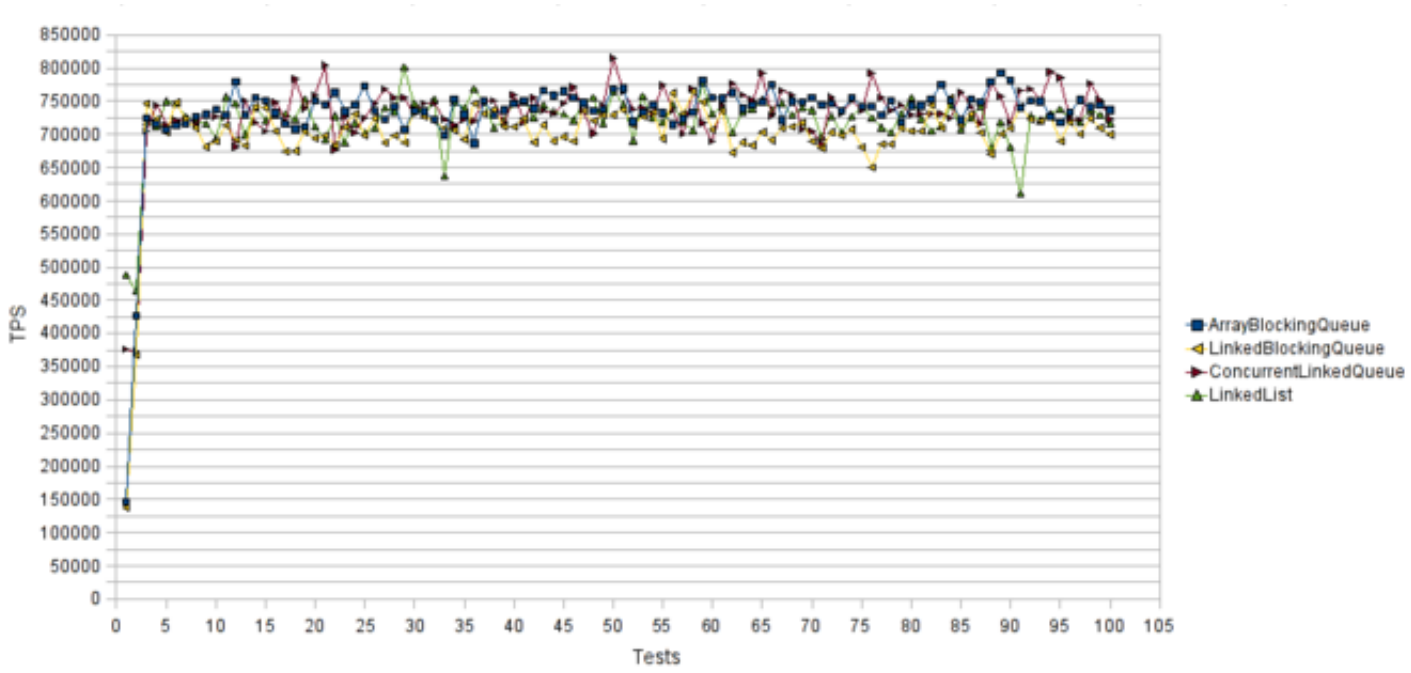
For the second test case we are going to have multiple Threads removing String elements from each Queue

implementation class. All queue implementation classes will be pre–populated with the String elements from the previous test case. Every Thread will remove a single element from each Queue implementation class until the Queue is empty.

For removing elements we will be utilizing the “take()” operation for the BlockingQueue implementation classes and the “poll()” operation for the Queue implementation classes as shown below :

- arrayBlockingQueue.take();
- linkedBlockingQueue.take();
- concurrentLinkedQueue.poll();
- synchronized(linkedList) {
  linkedList.poll();
  }

Below we present a performance comparison chart between the four aforementioned Queue implementation classes



The horizontal axis represents the number of test runs and the vertical axis the average transactions per second (TPS) for each test run. Thus higher values are better. Again all Queue implementation classes performed almost identically when removing String elements from them. ArrayBlockingQueue and

ConcurrentLinkedQueue performed slightly better compared to LinkedList and LinkedBlockingQueue. The latter was the worst performance scoring 710000 TPS on average.

Test case #3 – Iterators

For the third test case we are going to have multiple worker Threads iterating over the elements of each Queue implementation class. Every worker Thread will be using the Queue's "iterator()" operation to retrieve a reference to an Iterator instance and iterate through all the available Queue elements using the Iterator's "next()" operation. All Queue implementation classes will be pre–populated with the String values from the first test case. Below is the performance comparison chart for the aforementioned test case.

The horizontal axis represents the number of test runs and the vertical axis the average transactions per second (TPS) for each test run. Thus higher values are better. Both ArrayBlockingQueue and LinkedBlockingQueue implementation classes performed poorly compared to the ConcurrentLinkedQueue and LinkedList implementation classes. LinkedBlockingQueue scored 35 TPS on average, while ArrayBlockingQueue scored 81 TPS on average. On the other hand LinkedList outperformed ConcurrentLinkedQueue, resulting in 15000 TPS on average.

Test case #4 – Adding and removing elements

For our final test case we are going to implement the combination of test case #1 and test case #2 scenarios. In other words we are going to implement a producer – consumer test case. One group of worker Threads is going to insert String elements to every Queue implementation class, whereas another group of worker Threads is going to retract the String elements from them. Every Thread from the "inserting elements" group is going to insert just one element, whereas every Thread from the "retracting elements" group is going to retract just one element. Thus we are going to concurrently insert and retract 5000 unique String elements from every Queue implementation class in total.

To properly simulate the aforementioned test case we must start all worker Threads that retract elements prior starting the worker Threads that insert elements. In order for the worker Threads of the "retracting elements" group to be able to retract a single element they must wait and retry if the relevant Queue is empty. BlockingQueue implementation classes provide waiting functionality by default but Queue implementation classes do not. Thus for removing elements we will be utilizing the "take()" operation for the BlockingQueue implementation classes and the "poll()" operation for the Queue implementation classes

as shown below :

- arrayBlockingQueue.take();
- linkedBlockingQueue.take();
- while(result == null)
  result = concurrentLinkedQueue.poll();
- while(result == null)
  synchronized(linkedList) {
  result = linkedList.poll();
  }

As you can see we have implemented the bare minimum – a while loop – in order for our ConcurrentLinkedQueue and LinkedList consumers to be able to perform reties when retracting from an empty Queue. Of course you can experiment and implement a more sophisticated approach to the matter. Nevertheless keep in mind that the aforementioned along with any other artificial implementation is NOT a proposed solution and should be avoided in favor to the BlockingQueue "take()" operation as will be shown in the following performance comparison charts.

Below is the performance comparison chart for the addition part of the aforementioned test case.

Following is the performance comparison chart for the retraction part of the aforementioned test case.

The horizontal axis represent the number of test runs and the vertical axis the average transactions per second (TPS) for each test run. Thus higher values are better. As expected ArrayBlockingQueue and LinkedBlockingQueue outperformed both LinkedList and ConcurrentLinkedQueue. BlockingQueue implementations are designed to be used primarily for producer–consumer queues. Their blocking behavior grands them unparalleled performance and efficiency in this kind of scenarios, especially when the number of producer and consumer Threads is relatively high. In fact, according to relevant performance comparisons, the relative gain in performance between Queue implementation classes and BlockingQueue implementation classes increases in favor of the latter as the number of producer and consumer Threads rises.

As you can see from the provided performance results LinkedBlockingQueue achieved the best combined

(adding and removing elements) performance results and should be your number one candidate for implementing producer – consumer schenarios.

ConcurrentLinkedHashMap

Our ConcurrentLinkedHashMap implementation is a tweaked version of the ConcurrentHashMap implementation originally coded by **Doug Lea** and found on OpenJDK 1.6.0_0. We present a concurrent hash map and linked list implementation of the ConcurrentMap interface, with predictable iteration order. This implementation differs from ConcurrentHashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). Note that insertion order is not affected if a key is re-inserted into the map.

This implementation spares its clients from the unspecified, generally chaotic ordering provided by ConcurrentHashMap and Hashtable, without incurring the increased cost associated with TreeMap. It can be used to produce a copy of a map that has the same order as the original, regardless of the original map's implementation:

```
void foo(Map m) {
Map copy = new ConcurrentLinkedHashMap(m);
…
}
```

This technique is particularly useful if a module takes a map on input, copies it, and later returns results whose order is determined by that of the copy. (Clients generally appreciate having things returned in the same order they were presented.)

A special "ConcurrentLinkedHashMap(int,float,int,boolean)" constructor is provided to create a concurrent linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (access-order). This kind of map is well-suited to building LRU caches. Invoking the put or get method results in an access to the corresponding entry (assuming it exists after the invocation completes). The "putAll()" method generates one entry access for each mapping in the specified map, in the order that key-value mappings are provided by the specified map's entry set iterator. No other methods generate entry accesses. In particular, operations on collection-views do not affect the order of

iteration of the backing map.

The “removeEldestEntry(Map.Entry)” method may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map.

Performance is likely to be just slightly below that of ComcurrentHashMap, due to the added expense of maintaining the linked list, with one exception: Iteration over the collection-views of a ConcurrentLinkedHashMap requires time proportional to the size of the map, regardless of its capacity. Iteration over a ConcurrentHashMap is likely to be more expensive, requiring time proportional to its capacity.

You can download the latest version of the ConcurrentLinkedHashMap source and binary code from here

You can download the source code of all the “tester” classes that we used to conduct our performance comparison from here, here and here

Happy coding

Justin