ABOUT BYRON KIOURTZOGLOU

Byron is a master software engineer working in the IT and Telecom domains. He is an applications developer in a wide variety of applications/services. He is currently acting as the team leader and technical architect for a proprietary service creation and integration platform for both the IT and Telecom industries in addition to a in-house big data real-time analytics solution. He is always fascinated by SOA, middleware services and mobile development. Byron is co-founder and Executive Editor at Java Code Geeks.

Java Best Practices – Vector vs ArrayList vs HashSet

Posted by: Byron Kiourtzoglou in Core Java August 16th, 2010 5 Comments 1206 Views

Continuing our series of articles concerning proposed practices while working with the Java programming language, we are going to perform a performance comparison between the three probably most used Collection implementation classes. To make things more realistic we are going to test against a multi–threading environment so as to discuss and demonstrate how to utilize Vector, ArrayList and/or HashSet for high performance applications.

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

Vector vs ArrayList vs HashSet

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. Vector, ArrayList and HashSet are probably the most frequently used.

Nevertheless working with Collection implementation classes, especially in a multi–threading environment, can by tricky. The majority of them does not provide synchronized access by default. As a consequence, altering the internal structure of a Collection (inserting or retracting elements) in a concurrent manner will certainly lead to errors.

The case scenario that will be discussed here is having multiple Threads inserting, retracting and iterating through elements of every Collection implementation class mentioned above. 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 we do not mind the ordering of elements in the Collections. Furthermore since Vector is the only Collection implementation class of our test group that provides synchronized access by default, synchronization for ArrayList and HashSet Collection implementation classes will be achieved using *Collections.synchronizedList* and

Collections.synchronizedSet static methods. These methods provide a "wrapped" synchronized instance of a designated Collection implementation class as shown below :

- List syncList = Collections.synchronizedList(new ArrayList());
- Set syncSet = Collections.synchronizedSet(new HashSet());

Test case #1 – Adding elements in a collection

For the first test case we are going to have multiple Threads adding String elements in each Collection 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 :

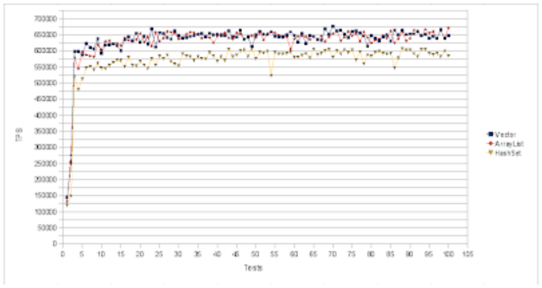
- For the first test repeat
  - Worker Thread #1 will insert the String element : "helloWorld-1-1"
  - Worker Thread #2 will insert the String element : "helloWorld-2-1"
  - Worker Thread #3 will insert the String element : "helloWorld-3-1" etc …

- 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 Collection implementation class will be populated with 5000 distinct String elements.

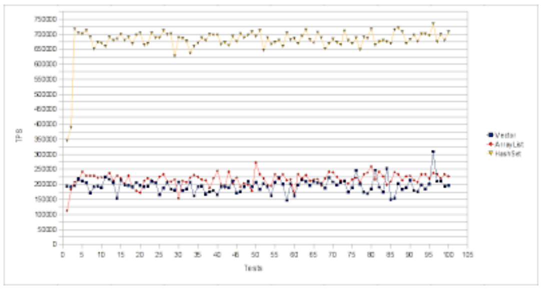
Below we present a performance comparison chart between the three aforementioned Collection implementation classes



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 Vector and ArrayList Collection implementation classes performed almost identically when adding elements to them. On the other hand the HashSet Collection implementation class presented a slightly inferior performance mainly due to the more complex internal structure and the hash generation mechanism.

Test case #2 – Removing elements from a collection

For the second test case we are going to have multiple Threads removing String elements from each Collection implementation class. All collection implementation classes will be pre–populated with the String elements from the previous test case. For removing elements we will be utilizing a shared Iterator instance among all worker Threads for each Collection implementation class. Synchronized access to the Iterator instance will also be implemented. Every worker Thread will be removing the next available element of the Collection implementation class, issuing the "next()" and "remove()" Iterator operations (to avoid ConcurrentModificationException). 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. Again both Vector and ArrayList Collection implementation classes performed almost identically when removing String elements from them. On the other hand the HashSet Collection implementation class outperformed Vector and ArrayList by far, resulting in 678000 TPS on average.

At this point we must pinpoint that by using the "remove(0)" method of Vector and ArrayList Collection implementation classes to remove String elements, we have achieved slightly better performance results compared to utilizing a synchronized shared Iterator instance. The reason that we have demonstrated the synchronized shared Iterator instance "next()" and "remove()" operations approach is to maintain a fare comparison between the three Collection implementation classes.

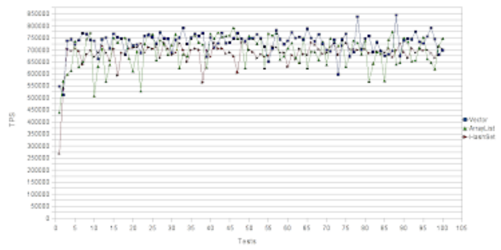
IMPORTANT NOTICE

Our test case scenario dictates that we remove the first element from each Collection implementation class. Nevertheless we must pinpoint that this applies as the worst case scenario for Vector and ArrayList Collection implementation classes. As one of our readers, James Watson, successfully commented on a relevant post from TheServerSide

"*The reason is that on ArrayList and Vecrtor, the remove() method will result in a System.arraycopy() call if any element except the last element is removed (the last element being the element with index: size – 1). Removing the first element means the entire rest of the array is copied which is an O(n) operation. Since the test removes all the elements in the List, the full test becomes O(n^2) (slow.)*

HashSet remove does not do any such array copies so it's remove is O(1) or constant time. For the full test it then is O(n) (fast.). If the tests were rewritten to remove the last element from the ArrayList and Vector, you would likely similar performance to the HashSet."

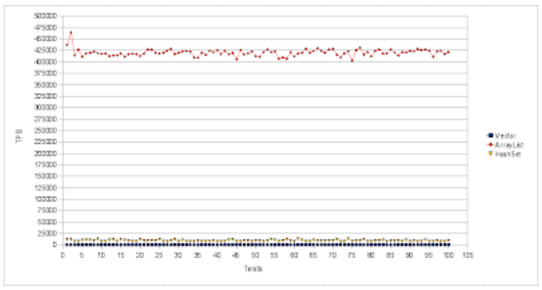
For that reason we will conduct this test again, removing the last element from Vector and ArrayList Collection implementation classes since we presume that the order of elements in the Collections is of no importance. The performance results are show below.



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 expected all Collection implementation classes performed almost identically when removing String elements from them.

Test case #3 – Iterators

For the third test case we are going to have multiple worker Threads iterating over the elements of each Collection implementation class. Every worker Thread will be using the Collection "iterator()" operation to retrieve a reference to an Iterator instance and iterate through all the available Collection elements using the Iterator "next()" operation. All Collection 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 Vector and HashSet Collection implementation classes performed poorly compared to the ArrayList Collection implementation class. Vector scored 68 TPS on average, while HashSet scored 9200 TPS on average. On the other hand the ArrayList outperformed Vector and HashSet by far, resulting in 421000 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. One group of worker Threads is going to insert String elements to every Collection implementation class, whereas another group of worker Threads is going to retract the String elements from them.

In the combined (adding and retracting elements) operation the synchronized shared Iterator instance approach for removing elements cannot be used. Adding and removing elements concurrently from a single Collection eliminates the use of a shared Iterator instance due to the fact that the internal structure of the Collection is constantly changing. For Vector and ArrayList Collection implementation classes the aforementioned limitation can be bypassed by using the "remove(0)" operation, which retracts the first element from a Collection internal storage. Unfortunately the HashSet Collection implementation class does not provide such functionality. Our proposed way for achieving maximum performance results for the combined operation using the HashSet Collection implementation class is the following :

- Use two distinct HashSets, one for adding elements and the other for retracting
- Implement a "controller" Thread that will swap the contents of the aforementioned HashSet classes when the "retracting" HashSet is empty. The "controller" Thread can be implemented as a regular TimerTask that can check the contents of the "retracting" HashSet at regular time intervals and perform the swap if needed
- A shared Iterator instance for the "retracting" HashSet should be created upon swapping the two HashSets
- All worker Threads that retract elements should wait for the "retracting" HashSet to be filled with elements and be notified after the swap

What follows is a code snippet displaying the proposed implementation :

Global Declarations :

```
Set<string> s = new HashSet<string>(); // The HashSet for retracting elements
Iterator<string> sIt = s.iterator(); // The shared Iterator for retracting elements
Set<string> sAdd = new HashSet<string>(); // The HashSet for adding new elements
Boolean read = Boolean.FALSE; // Helper Object for external synchronization and wait - notify
functionality when retracting elements from the “s” HashSet
Boolean write = Boolean.FALSE; // Helper Object for external synchronization when writing
elements to the “sAdd” HashSet
```

Code for adding elements to the “sAdd” HashSet :

```
synchronized(write) {
 sAdd.add(“helloWorld” + "-" + threadId + "-" + count);
}
```

Code for retracting elements from the “s” HashSet :

```
while (true) {
 synchronized (read) {
  try {
   sIt.next();
   sIt.remove();
   break;
  } catch (NoSuchElementException e) {
   read.wait();
  }
 }
}
```

The “controller” class code :

```
public class Controller  extends TimerTask {

 public void run() {
  try {
   performSwap();
  } catch (Exception ex) {
   ex.printStackTrace();
  }
 }

 private void performSwap() throws Exception {
  synchronized(read) {
   if(s.isEmpty()) {
    synchronized(write) {
     if(!sAdd.isEmpty()) {
      Set<string> tmpSet;
      tmpSet = s;
      s = sAdd;
      sAdd = tmpSet;
      sIt = s.iterator();
      read.notifyAll();
     }
    }
   }
  }
 }

}
```

Finally, the code to schedule the “controller” TimerTask

```
Timer timer = new Timer();
timer.scheduleAtFixedRate(new Controller(), 0, 1);
```

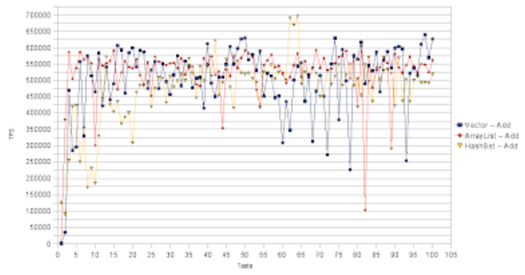
We should start the “controller” task prior starting the worker Threads that write to and read from the relevant HashSets.

Keep in mind that for the Vector and ArrayList Collection implementation classes we have used the “remove(0)” operation to retract elements. Below are the code snippets for the aforementioned Collection implementation classes :

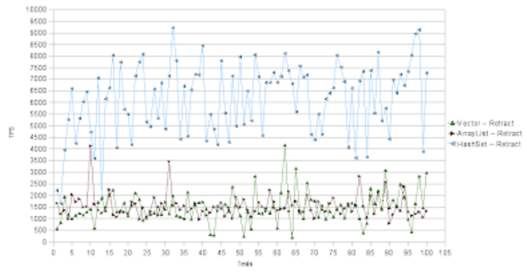
```
while (true) {
 try {
  vector.remove(0);
  break;
 } catch (ArrayIndexOutOfBoundsException e) {
 }
}

while (true) {
 try {
  syncList.remove(0);
  break;
 } catch (IndexOutOfBoundsException e) {
 }
}
```

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 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 Vector and ArrayList Collection implementation classes performed almost identically when adding and retracting elements from them. On the other hand for the element addition test case our proposed implementation, with the "adding" and "retracting" HashSet pair, performed slightly inferiorly compared to Vector and ArrayList implementations. Nevertheless for the element retraction test case our "adding" and "retracting" HashSet pair implementation outperformed both Vector and ArrayList implementations by far scoring 6000 TPS on average.

Happy coding

Justin

Related Articles :

- Java Best Practices – DateFormat in a Multithreading Environment
- Java Best Practices – High performance Serialization
- Java Best Practices – String performance and Exact String Matching
- Java Best Practices – Queue battle and the Linked ConcurrentHashMap
- Java Best Practices – Char to Byte and Byte to Char conversions

Related Snippets :

- ArrayList size example
- Insert all elements of Collection to specific ArrayList index
- Check for element existence in HashSet example
- HashSet Iterator example
- Java Map Example
- Add element to specified index of Vector example

Tagged with: ARRAYLIST HASHSET JAVA BEST PRACTICES VECTOR

(***0*** *rating,* ***0*** *votes*)

You need to be a registered member to rate this. 5 Comments 1206 Views Tweet it!

I agree to the Terms and Privacy Policy

Sign up

LIKE A CODE GEEKS

5

Join the discussion...

5 0 0 4

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Subscribe

newest oldest most voted

Tilmann Kuhn
Guest

Hey Justin, this is great work and very helpful! The only pity is that you did not include the other three Collections LinkedList, LinkedHashSet and TreeSet in your statistics. Could you add them or provide the tests source code so one can reproduce your findings?

+ 0 − Reply 6 years ago

Sridhar.Goranti
Guest

Interesting comparison.This is very helpful to everyone. thank you – Sridhar.Goranti

+ 0 − Reply 5 years ago

KH Nayef Reza
Guest

A really nice one. Really helpful. But still the question remains, should list and set be compared in this way, or should there be other things that need be considered to make these two comparable.

+ 2 − Reply 5 years ago

JK
Guest

ArraList Vs Vector !! Most Frequently asked question in java interviews

http://jeet-software.blogspot.in/2014/11/arraylist-vs-vector-in-java.html

+ 0 − Reply 3 years ago

sean
Guest

this is very helpful!

+ 0 − Reply 22 hours ago

KNOWLEDGE BASE

HALL OF FAME

ABOUT JAVA CODE GEEKS

- Courses
- Examples
- Minibooks
- Resources
- Tutorials

PARTNERS

- Mkyong

THE CODE GEEKS NETWORK

- .NET Code Geeks
- Java Code Geeks
- System Code Geeks
- Web Code Geeks

- “Android Full Application Tutorial” series
- 11 Online Learning websites that you should check out
- Advantages and Disadvantages of Cloud Computing – Cloud computing pros and cons
- Android Google Maps Tutorial
- Android JSON Parsing with Gson Tutorial
- Android Location Based Services Application – GPS location
- Android Quick Preferences Tutorial
- Difference between Comparator and Comparable in Java
- GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial
- Java Best Practices – Vector vs ArrayList vs HashSet

JCGs (Java Code Geeks) is an independent online community focused on creat ultimate Java to Java developers resource center; targeted at the technical ar technical team lead (senior developer), project manager and junior developer JCGs serve the Java, SOA, Agile and Telecom communities with daily news wri domain experts, articles, tutorials, reviews, announcements, code snippets an source projects.

DISCLAIMER