Java object allocation overhead

Asked 16 years, 3 months ago Modified 10 years, 7 months ago

Viewed 2k times

8

I am writing an immutable DOM tree in Java, to simplify access from multiple threads.*

However, it does need to support inserts and updates as fast as possible. And since it is immutable, if I make a change to a node on the N'th level of the tree, I need to allocate at least N new nodes in order to return the new tree.

My question is, would it be dramatically faster to pre-allocate nodes rather than create new ones every time the tree is modified? It would be fairly easy to do - keep a pool of several hundred unused nodes, and pull one out of the pool rather than create one whenever it was required for a modify operation. I can replenish the node pool when there's nothing else going on. (in case it isn't obvious, execution time is going to be much more at a premium in this application than heap space is)

Is it worthwhile to do this? Any other tips on speeding it up?

Alternatively, does anyone know if an immutable DOM library already? I searched, but couldn't find anything.

*Note: For those of you who aren't familiar with the concept of immutability, it basically means that on any operation to an object that changes it, the method returns a copy of the object with the changes in place, rather than the changed object. Thus, if another thread is still reading the object it will continue to happily operate on the "old" version, unaware that changes have been made, rather than crashing horribly. See http://www.javapractices.com/topic/TopicAction.do?Id=29

java xml dom concurrency

Share

Improve this question

Follow

edited May 23, 2014 at 9:26

Baby

5,092 ● 3 ● 32 ● 52

asked Sep 3, 2008 at 19:37

levand

8,480 ● 3 ● 42 ● 55

6 Answers

Sorted by: Highest score (default)

12

These days, object creation is pretty dang fast, and the concept of object pooling is kind of obsolete (at least in general; connection pooling is of course still valid).

Avoid premature optimization. Create your nodes when you need them when doing your copies, and then see if that becomes prohibitively slow. If so, then look into some

techniques to speed it up. But unless you already know that what you've got isn't fast enough, I wouldn't go introducing all the complexity you're going to need to get pooling going.

Share Improve this answer

Follow

answered Sep 3, 2008 at 19:47

jodonnell

50.4k ● 11 ● 64 ● 67

> +1 for the right answer. Looking up an object in a pool (especially in a thread-safe way!) is nowadays usually slower than object creation. Also worth noting that various libraries (and also Clojure) effectively do this for all their immutable data structures. So it's a pretty tried and tested approach.
> – mikera Jun 17, 2010 at 13:05

3

I hate to give a non-answer, but I think the only definitive way to answer a performance question like this might be for you to code both approaches, benchmark the two, and compare the results.

Share Improve this answer

Follow

answered Sep 3, 2008 at 19:45

matt b

140k ● 66 ● 284 ● 350

1

> I'm not sure if you can avoid explicitly synchronizing certain methods in order to make sure everything is thread-safe.

One specific case you need to synchronize one side or the other of making a newly created node available to other threads as otherwise you risk the VM/CPU re-ordering the writes of the fields past the write of the reference to the shared node, exposing a party constructed object.

> Try to think in a higher level. You have an IMMUTABLE tree (that is basically a set of nodes pointing to its children). You want to insert a node in it. Then, there's no way out: you have to create a new WHOLE tree.

If you choose to implement the tree as a set of nodes pointing to the children, then you would have to create new nodes along the path of the changed node to the root. The others have the same value as before, and normally are shared. So you need to create a partial new tree, which usually would mean (depth of edited node) parent nodes.

If you can cope with a less direct implementation, you should be able to get away with only creating parts of nodes, using techniques similar to those described in Purely Functional Data Structures to either reduce the average cost of the creation, or you can by-pass it using semi-functional approaches (such as creating an iterator which wraps an existing iterator, but returns the new node instead of the old, together with a mechanism to repair such patches in the structure as time goes on). An XPath

style api might be better than a DOM api in that case - it might you decouple the nodes from the tree a bit more, and treat the mutated tree more intelligently.

Share Improve this answer

Follow

answered Sep 4, 2008 at 0:11

Pete Kirkham

49.3k ● 5 ● 94 ● 173

0

I'm a little confused about what you're trying to do in the first place. You want all of the nodes to be immutable AND you want to pool them? Aren't these 2 ideas mutually exclusive? When you pull an object out of the pool, won't you have to invoke a setter to link up the children?

I think that using immutable nodes is probably not going to give you the kind of thread-safety you need in the first place. What happens if 1 thread is iterating over the nodes (a search or something), while another thread is adding/removing nodes? Won't the results of the search be invalid? I'm not sure if you can avoid explicitly synchronizing certain methods in order to make sure everything is thread-safe.

Share Improve this answer

Follow

answered Sep 3, 2008 at 19:59

Tim Frey

9,931 ● 9 ● 45 ● 61

@Outlaw Programmer

0

> When you pull an object out of the pool, won't you have to invoke a setter to link up the children?

Each node needn't be immutable internally to the package, only to the outward-facing interface. `node.addChild()` would be an immutable function with public visibility and return a Document, wheras `node.addChildInternal()` would be be a normal, mutable function with package visibility. But since it is internal to the package, it can only be called as a descendent of `addChild()` and the structure as a whole is guarenteed to be thread safe (provided I synchronize access to the object pool). Do you see a flaw in this...? If so, please tell me!

> I think that using immutable nodes is probably not going to give you the kind of thread-safety you need in the first place. What happens if 1 thread is iterating over the nodes (a search or something), while another thread is adding/removing nodes?

The tree as a whole will be immutable. Say I have Thread1 and Thread2, and tree dom1. Thread1 starts a read operation on dom1, while, concurrently, Thread2 starts a write operation on dom1. However, all the changes Thread2 makes will actually be made to a new object, dom2, and dom1 will be immutable. It is true that

the values read by Thread1 will be (a few microseconds) out of date, but it won't crash on an IndexOutOfBounds or NullPointer exception or something like it would if it was reading a mutable object that was being written to. Then, Thread2 can fire an event containing dom2 to Thread1 so that it can do its read again and update its results, if necessary.

Edit: clarified

Share Improve this answer

Follow

edited May 23, 2017 at 10:27

Community Bot

1 ● 1

answered Sep 3, 2008 at 20:25

levand

8,480 ● 3 ● 42 ● 55

0

I think @Outlaw has a point. The structure of the DOM tree resides in the nodes itself, having a node pointing to its children. To modify the structure of a tree you have to modify the node, so you can't have it pooled, you have to create a new one.

Try to think in a higher level. You have an IMMUTABLE tree (that is basically a set of nodes pointing to its children). You want to insert a node in it. Then, there's no way out: you have to create a new WHOLE tree.

Yes, the immutable tree is thread-safe, but it will impact performance. Object creation may be fast, but not faster

then NO object creation. :)

Share Improve this answer

Follow

answered Sep 3, 2008 at 20:42

Marcio Aguiar
14.5k ● 6 ● 40 ● 42

1 It's not true that you need to create a whole new tree - in fact you only need to create a copy of the ancestors of the new node. See "persistent data structures". – mikera Jun 17, 2010 at 13:02