Does the Java VM move objects in memory, and if so - how?

Asked 16 years, 3 months ago Modified 16 years, 2 months ago

Viewed 4k times

15

Does the Java virtual machine ever move objects in memory, and if so, how does it handle updating references to the moved object?

I ask because I'm exploring an idea of storing objects in a distributed fashion (ie. across multiple servers), but I need the ability to move objects between servers for efficiency reasons. Objects need to be able to contain pointers to each-other, even to objects on remote servers. I'm trying to think of the best way to update references to moved objects.

My two ideas so far are:

1. Maintain a reference indirection somewhere that doesn't move for the lifetime of the object, which we update if the object moves. But - how are these indirections managed?
2. Keep a list of reverse-references with each object, so we know what has to be updated if the object is moved. Of course, this creates a performance overhead.

I'd be interested in feedback on these approaches, and any suggestions for alternative approaches.

java memory-management garbage-collection jvm

Share

Improve this question

Follow

edited Oct 17, 2008 at 13:17

Dan Dyer

54.4k ● 19 ● 135 ● 166

asked Sep 18, 2008 at 0:13

sanity

35.7k ● 43 ● 140 ● 228

6 Answers

Sorted by: Highest score (default)

11

In reference to the comment above about walking the heap.

Different GC's do it different ways.

Typically copying collectors when they walk the heap, they don't walk all of the objects in the heap. Rather they walk the LIVE objects in the heap. The implication is that if it's reachable from the "root" object, the object is live.

So, at this stage is has to touch all of the live objects anyway, as it copies them from the old heap to the new heap. Once the copy of the live objects is done, all that remains in the old heap are either objects already copied,

or garbage. At that point the old heap can be discarded completely.

The two primary benefits of this kind of collector are that it compacts the heap during the copy phase, and that it only copies living objects. This is important to many systems because with this kind of collector, object allocation is dirt cheap, literally little more than incrementing a heap pointer. When GC happens, none of the "dead" objects are copied, so they don't slow the collector down. It also turns out in dynamic systems that there's a lot more little, temporary garbage, than there is long standing garbage.

Also, by walking the live object graph, you can see how the GC can "know" about every object, and keep track of them for any address adjustment purposes performed during the copy.

This is not the forum to talk deeply about GC mechanics, as it's a non-trivial problem, but that's the basics of how a copying collector works.

A generational copying GC will put "older" objects in different heaps, and those end up being collected less often than "newer" heaps. The theory is that the long lasting objects get promoted to older generations and get collected less and less, improving overall GC performance.

answered Sep 18, 2008 at 1:43

Share Improve this answer

Follow

Will Hartung

118k ● 20 ● 133 ● 207

3

The keyword you're after is "compacting garbage collector". JVMs are permitted to use one, meaning that objects can be relocated. Consult your JVM's manual to find out whether yours does, and to see whether there are any command-line options which affect it.

The conceptually simplest way to explain compaction is to assume that the garbage collector freezes all threads, relocates the object, searches heap and stack for all references to that object, and updates them with the new address. Actually it's more complex than that, since for performance reasons you don't want to perform a full sweep with threads stalled, so an incremental garbage collector will do work in preparation for compaction whenever it can.

If you're interested in indirect references, you could start by researching weak and soft references in Java, and also the remote references used by various RPC systems.

Share Improve this answer

Follow

answered Sep 18, 2008 at 0:40

Steve Jessop

279k ● 40 ● 469 ● 709

Interesting, I will investigate that. But, isn't it extremely inefficient to search the entire heap for references to an object? – sanity Sep 18, 2008 at 0:47

Yes and no. In a sense that's what the "mark" phase has to do anyway, in order to find out whether or not an object it's planning to destroy is referenced. So hundreds of man-years of work have gone into optimising incremental garbage collectors, to avoid doing it all in one go. – Steve Jessop Sep 18, 2008 at 0:54

3

I'd be curious to know more about your requirements. As another answer suggests, Terracotta may be exactly what you are looking for.

There is a subtle difference however between what Terracotta provides, and what you are asking for, thus my inquiry.

The difference is that as far as you are concerned, Terracotta does not provide "remote" references to objects - in fact the whole "remote" notion of RMI, JMS, etc. is entirely absent when using Terracotta.

Rather, in Terracotta, all objects reside in large virtual heap. Threads, whether on Node 1, or Node 2, Node 3, Node 4, etc all have access to any object in the virtual heap.

There's no special programming to learn, or special APIs, objects in the "virtual" heap have exactly the same behavior as objects in the local heap.

In short, what Terracotta provides is a programming model for multiple JVMs that operates exactly the same as a the programming model for a single JVM. Threads in

separate nodes simply behave like threads in a single node - object mutations, synchronized, wait, notify all behave exactly the same across nodes as as across threads - there's no difference.

Furthermore, unlike any solution to come before it, object references are maintained across nodes - meaning you can use ==. It's all a part of maintaining the Java Memory Model across the cluster which is the fundamental requirement to make "regular" Java (e.g. POJOs, synchronized, wait/notify) work (none of that works if you don't / can't preserve object identity across the cluster).

So the question comes back to you to further refine your requiements - for what purpose do you need "remote" pointers?

Share Improve this answer

Follow

answered Sep 30, 2008 at 6:06

Taylor Gautier

4,944 ● 7 ● 32 ● 24

2

(Practically) Any garbage collected system has to move objects around in memory to pack them more densely and avoid fragmentation problems.

What you are looking at is a very large and complex subject. I'd suggest you read up on existing remote object style API's: .NET remoting and going further back technologies like CORBA

Any solution for tracking the references will be complicated by having to deal with all the failure modes that exist in distributed systems. The JVM doesn't have to worry about suddenly finding it can't see half of its heap because a network switch glitched.

When you drill into the design I think a lot of it will come down to how you want to handle different failure cases.

Response to comments:

Your question talks about storing objects in a distributed fashion, which is exactly what .NET remoting and CORBA address. Admittedly neither technology supports migration of these objects (AFAIK). But they both deal extensively with the concepts of object identity which is a critical part of any distributed object system: how do different parts of the system know which objects they are talking about.

I am not overly familiar with the details of the Java garbage collector, and I'm sure the Java and .NET garbage collectors have a lot of complexity in them to achieve maximum performance with minimum impact on the application.

However, the basic idea for garbage collection is:

- The VM stops all threads from running managed code
- It performs a reachability analysis from the set of known 'roots': static variables, local variables on all

the threads. For each object it finds it follows all references within the object.

- Any object not identified by the reachability analysis is garbage.
- Objects that are still alive can then be moved down in memory to pack them densely. This means that any references to these objects also have to be updated with the new address. By controlling when a garbage collect can occur the VM is able to guarantee that there are no object references 'in-the-air' (ie. being held in a machine register) that would cause a problem.
- Once the process is complete the VM starts the threads executing again.

As a refinement of this process the VM can perform generational garbage collection, where separate heaps are maintained based on the 'age' of an object. Objects start in heap 0 and if they survive several GCs then the migrate to heap 1 and eventually to heap 2 (and so on - .NET supports 3 generations only though). The advantage of this is that the GC can run heap 0 collections very frequently, and not have to worry about doing the work to prove the long lived objects (which have ended up in heap 2) are still alive (which they almost certainly are).

There are other refinements to support concurrent garbage collection, and details around threads that are

actually executing unmanaged code when the GC is scheduled that add a lot more complexity to this area.

Share Improve this answer Follow

edited Sep 18, 2008 at 1:04

answered Sep 18, 2008 at 0:27

Rob Walker
47.4k ● 15 ● 100 ● 137

I honestly can't see the relevance of .NET remoting and CORBA to this. Can you provide more specific pointers? – sanity Sep 18, 2008 at 0:33

Also, really what I'm looking for is an explanation of *how* garbage collected systems move objects around. – sanity Sep 18, 2008 at 0:38

@Sanity - Rob Walker is recommending those technologies as a means to better understand their strategies on why and how they do the things they do. – mP. May 7, 2009 at 11:24

1

sounds like you are looking for a distributed cache, something like terracotta or oracle's java objece cache (formerly tangersol).

Share Improve this answer Follow

answered Sep 18, 2008 at 3:24

james
699 ● 4 ● 3

I think the product you are refering to is called "Coherence" and the company was Tangosol – Tnilsson Sep 18, 2008 at 8:40

0

If you are willing to go that deep down, you can take a look to JBoss Cache architecture docs and grab some of its source code as reference.

This is not exactly what you described, but it works very similar.

Here's the link.

http://www.jboss.org/jbosscache/

I hope this helps.

Share Improve this answer

Follow

answered Sep 30, 2008 at 16:23

OscarRyz

199k ● 119 ● 396 ● 573