How does the default hashCode() work?

Jan 30, 2017

In which scratching the surface of `hashCode()` leads to a speleology trip through the JVM source reaching object layout, biased locking, and surprising performance implications of relying on the default `hashCode()`.

Abundant thanks to Gil Tene and Duarte Nunes reviewing drafts of this article and their very valuable insights, suggestions and edits. Any remaining errors are my own.

A trivial mystery

Last week at work I submitted a trivial change to a class, an implementation of `toString()` so logs would be meaningful. To my surprise, the change caused a ~5% coverage drop in the class. I knew that all new code was covered by existing unit tests so, what could be wrong? Comparing coverage reports a sharper colleague noticed that the implementation of `hashCode()` was covered before the change but not after. Of course, that made sense: the default `toString()` calls `hashCode()`:

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

After overriding `toString()`, our custom `hashCode()` was no longer being called. We were missing a test.

Everyone knew the default `toString()` but..

What is the default implementation of `hashCode()`?

The value returned by the default implementation of `hashCode()` is called **identity hash code** so I will use this term from now on to distinguish it from the hash provided by overriden implementations of `hashCode()`. FYI: even if a class overrides `hashCode()`, you can always get the identity hash code of an object `o` by calling `System.identityHashCode(o)`.

Common wisdom is that the identity hash code uses the integer representation of the memory address. That's also what the J2SE JavaDocs for Object.hashCode() imply:

```
... is typically implemented by converting the internal address of
the object into an integer, but this implementation technique is not
required by the Java™ programming language.
```

Still, this seems problematic as the method contract requires that:

```
Whenever it is invoked on the same object more than once during an
execution of a Java application, the hashCode method must consistently
return the same integer.
```

Given that the JVM will relocate objects (e.g. during garbage collection cycles due to promotion or compaction), after we calculate an object's identity hash we must be able to retain it in a way that survives object relocation.

A possibility could be to take the current memory position of the object on the first call to `hashCode()`, and save it somewhere along with the object, like the object's header. That way, if the object is moved to a different memory location, it would carry the original hash with it. A caveat of this method is that it won't prevent two objects from having the same identity hash, but that's allowed by the spec.

The best confirmation would be to to look at the source. Unfortunately, the default `java.lang.Object::hashCode()` is a native function:

```
public native int hashCode();
```

Helmets on.

Will the real `hashCode()` please stand up

Note that the identity `hashCode()` implementation **is dependant on the JVM**. Since I will only look at OpenJDK sources, you should assume this specific implementation whenever I talk about the JVM. All links refer to changeset 5820:87ee5ee27509 of the Hotspot tree, I assume that most of it will also be applicable to Oracle's JVM, but things could (in fact, are) different in others (more about this later.)

OpenJDK defines entry points for `hashCode()` at `src/share/vm/prims/jvm.h` and `src/share/vm/prims/jvm.cpp`. The latter has:

```
e(JNIEnv* env, jobject handle))
);
lassic virtual machine; return 0 if object is NULL
: ObjectSynchronizer::FastHashCode (THREAD, JNIHandles::resolve_non_null(handle)) ;
```

ObjectSynchronizer::FastHashCode() is also called from `identity_hash_value_for`, which is used from a few other call sites (e.g.: `System.identityHashCode()`)

```
708 intptr_t ObjectSynchronizer::identity_hash_value_for(Handle obj) {
709   return FastHashCode (Thread::current(), obj()) ;
710 }
```

One might naively expect ObjectSynchronizer::FastHashCode() to do something like:

```
if (obj.hash() == 0) {
    obj.set_hash(generate_new_hash());
```

```
}
return obj.hash();
```

But it turns out to be a hundred line function that seems to be far more complicated. At least we can spot a couple of if-not-exists-generate blocks like:

```
685   mark = monitor->header();
...
687   hash = mark->hash();
688   if (hash == 0) {
689     hash = get_next_hash(Self, obj);
...
701   }
...
703   return hash;
```

Which seems to confirm our hypothesis. Let's ignore that `monitor` for now, and be satisfied that it gives us the object header. It is kept at `mark`, a pointer to an instance of markOop, which represents the mark word that belongs in the low bits of the object header. So, tries to get a hash inside the mark word. If it's not there, it's generated using `get_next_hash`, saved, and returned.

The actual identity hash generation

As we saw, this happens at get_next_hash. This function offers six methods based on the value of some `hashCode` variable.

```
0. A randomly generated number.
1. A function of memory address of the object.
2. A hardcoded 1 (used for sensitivity testing.)
3. A sequence.
4. The memory address of the object, cast to int.
5. Thread state combined with xorshift (https://en.wikipedia.org/wiki/Xorshift)
```

So what's the default method? OpenJDK 8 seems to default on 5 according to globals.hpp:

```
1127   product(intx, hashCode, 5,                                                \
1128           "(Unstable) select hashCode generation algorithm")                \
```

OpenJDK 9 keeps the same default. Looking at previous versions, both OpenJDK 7 and OpenJDK 6 use the first method, a random number generator.

So, unless I'm looking at the wrong place **the default hashCode implementation in OpenJDK has nothing to do with the memory address**, at least since version 6.

Object headers and synchronization

Let's go back a couple of points that we left unexamined. First, ObjectSynchronizer::FastHashCode() seems overly complex, needing over 100 lines to perform what we though was a trivial get-or-generate operation. Second, who is this `monitor` and why does it have our object's header?

The structure of the mark word is a good place to start making progress. In OpenJDK, it looks like this

```
30 // The markOop describes the header of an object.
31 //
32 // Note that the mark is not a real oop but just a word.
33 // It is placed in the oop hierarchy for historical reasons.
34 //
35 // Bit-format of an object header (most significant first, big endian layout bel
36 //
37 //  32 bits:
38 //  --------
39 //             hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal obje
40 //             JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased obje
41 //             size:32 ------------------------------------------>| (CMS free bl
42 //             PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promote
43 //
```

```
44 //  64 bits:
45 //  --------
46 //  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal obje
47 //  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased obje
48 //  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promote
49 //  size:64 ----------------------------------------------------->| (CMS free bl
50 //
51 //  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && no
52 //  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && bi
53 //  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CM
54 //  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CM
```

The format is slightly different on 32 and 64 bits. The latter has two variants depending on whether Compressed Object Pointers are enabled. Both Oracle and OpenJDK 8 **do** by default.

Object headers may thus relate to a free block or an actual object, in which case there are multiple possible states. In the simplest, (“normal object”) the identity hash is stored directly in the low addresses of the header.

But in other states, we find a pointer to a `JavaThread` or a `PromotedObject`. The plot thickens: if we put the identity hash in a “normal object”, will someone take it away? Where? If the object is biased, where can we get/set the hash? What is a biased object?

Let’s try to answer those questions.

Biased locking

Biased objects appear as a result of Biased Locking. A (patented!) feature enabled by default from HotSpot 6 that tries to alleviate the cost of locking objects. Such operations are expensive because their implementation often relies on atomic CPU instructions (CAS) in order to safely handle lock/unlock requests on the object from different threads. It was observed that in most applications, the majority of objects are only ever locked by one thread so paying the cost of the atomic operation was often a waste. To avoid it, JVMs with biased locking allow threads to try and “bias” an object towards themselves. While

an object is biased, the lucky thread can lock/unlock the object without atomic instructions. As long as there are no threads contending for the same object, we'll gain performance.

The `biased_lock` bit in the header indicates whether an object is biased by the thread pointed at by `JavaThread*`. The `lock` bits indicate whether the object is locked.

Precisely because OpenJDK's implementation of biased locking requires writing a pointer in the mark word, it also needs to relocate the real mark word (which contains the identity hash.)

This could explain the additional complexity in `FastHashCode`. The header not only holds the identity hash code, but also locking state (like the pointer to the lock's owner thread). So we need to consider all cases and find where the identity hash resides.

Let's go read `FastHashCode`. The first thing we find is:

```
601 intptr_t ObjectSynchronizer::FastHashCode (Thread * Self, oop obj) {
602   if (UseBiasedLocking) {
610     if (obj->mark()->has_bias_pattern()) {
          ...
617       BiasedLocking::revoke_and_rebias(hobj, false, JavaThread::current());
          ...
619       assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now
620     }
621   }
```

Wait. It just **revoked existing biases, and disabled biased locking on the object** (the `false` means "don't attempt rebias"). A few lines down, this is indeed an invariant:

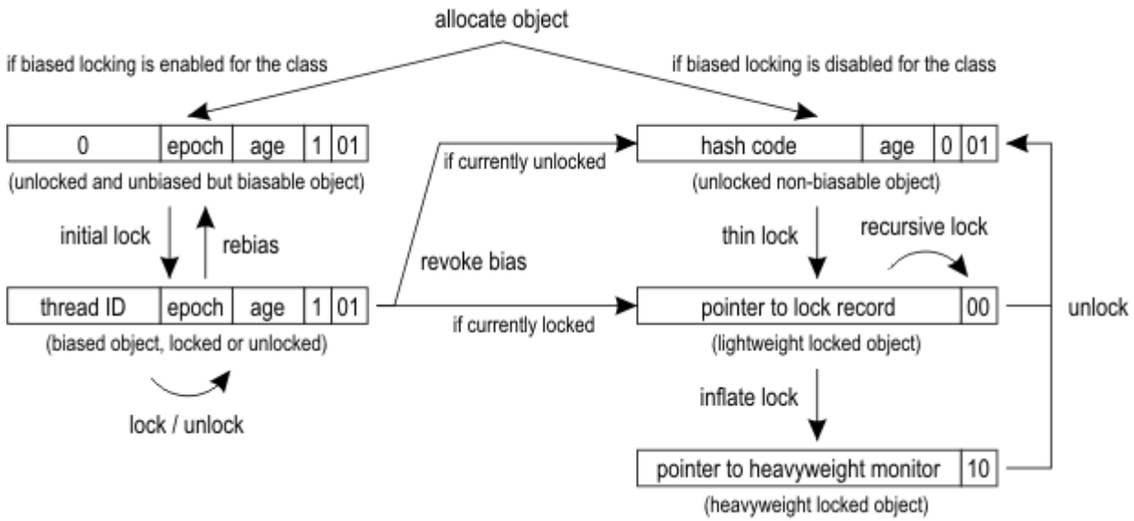
```
637   // object should remain ineligible for biased locking
638   assert (!mark->has_bias_pattern(), "invariant") ;
```

If I'm reading correctly, this means that **simply asking for the identity hash code of an object will disable biased locking**, which in turn forces any attempt to lock the object to use expensive atomic instructions. Even if there is only one thread.

Oh boy.

Why does keeping biased locking state conflict with keeping the identity hash code?

To answer this question we must understand which are the possible locations of the mark word (that contains the identity hash) depending on the lock state of the object. The transitions are illustrated in this diagram from the HotSpot Wiki:



My (fallible) reasoning is the following.

For the 4 states at the top of the diagram, the OpenJDK will be able to use “thin” lock representations. In the simplest case (no locks) this means having the identity hash and other data directly in the object’s space for the mark word:

```
46 //  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal obje
```

in more complex cases, it needs that space to keep a pointer to the “lock record”. The mark word will thus be “displaced” and put somewhere else.

While we have only one thread trying to lock the object, that pointer will actually refer to a memory location in the thread’s own stack. Which is twice good: it’s fast (no contention or coordination to access that memory location), and it suffices for the thread to identify that it owns the lock (because the memory location points to its own stack.)

But this won’t work in all cases. If we have contended objects (e.g. objects used on synchronized statements that many threads traverse) we will need a more complex structure that fits not only a copy of the object’s header (again, “displaced”), but also a list of waiters. A similar need for a list of waiters appears if a thread executes `object.wait()`.

This richer data structure is the ObjectMonitor, which is referred to as a the “heavyweight” monitor in the diagram. The value left in the object’s header doesn’t point to a “displaced mark word” anymore, but to an actual object (the monitor). Accessing the identity hash code will now require “inflating the monitor”: chasing a pointer to an object and reading/mutating whichever field contains the displaced mark word. Which is more expensive and requires coordination.

`FastHashCode` does have work to do.

Lines L640 to L680 deal with finding the header and checking for a cached identity hash. I believe these are a fast path that probe for cases that don’t need to inflate the monitor.

From L682 it needs to bite the bullet:

```
682   // Inflate the monitor to set hash code
683   monitor = ObjectSynchronizer::inflate(Self, obj);

684   // Load displaced header and check it has hash code
685   mark = monitor->header();
...
687   hash = mark->hash();
```

At this point, if the id. hash is there (`hash != 0`), the JVM can return. Otherwise we'll get one from `get_next_hash` and safely store it in the displaced header kept by the `ObjectMonitor` .

This seems to offer a reasonable explanation to why calling `hashCode()` on an object of a class that doesn't override the default implementation makes the object ineligible for biased locking:

- In order to keep the identity hash of an object consistent after relocation we need to store the hash in the object's header.
- Threads asking for the identity hash may not even care about locking the object, but in practise they will be sharing data structures used by the locking mechanism. This is a complex beast in itself that might be not only mutating, but also **moving** (displacing) the header contents.
- Biased locking helped perform lock/unlock operations without atomic operations, and this was effective as long as only one thread locked the object because we could keep the lock state in the mark word. I'm not 100% sure here, but I understand that since other threads may ask for the identity hash, **even if there is a single thread interested in the lock**, the header word will be contended and require atomic operations to be handled correctly. Which defeats the whole point of biased locking.

Recap

- The default `hashCode()` implementation (identity hash code) **has nothing to do with the object's memory address**, at least in OpenJDK. In versions 6 and 7 it is a randomly generated number. In 8 and, for now, 9, it is a number based on the thread state. Here is a test that yields the same conclusion.
  - Proving that "implementation-dependent" warns are not aesthetic: Azul's Zing **does** generate the identity hash from the object's memory address.
- In HotSpot, the result of the identity hash generation is generated once, and cached in the **mark word** of the object's header.
  - Zing uses a different solution to keep it consistent despite object relocations, in which they delay storing the id. hash until the object relocates. At that point, it's stored in a "pre-header"
- In HotSpot, calling the default `hashCode()` , or `System.identityHashCode()` will make the object ineligible for biased locking.
  - This implies that **if you are synchronizing on objects that have no contention, you'd better override the default `hashCode()` implementation** or you'll miss out on JVM optimizations.

- It is possible to disable biased locking in HotSpot, on a per-object basis.
  - This can be very useful. I've seen applications very heavy on contended producer/consumer queues where biased locking was causing more trouble than benefit, so we disabled the feature completely. Turns out, we could've done this only on specific objects/classes simply by calling System.identityHashCode() on them.
- ~~I have found no HotSpot flag that allows changing the default generator, so experimenting with other options might need to compile from source~~.
  - Admittedly, I didn't look much. Michael Rasmussen kindly pointed out that **-XX:hashCode=2** can be used to change the default. Thanks!

Benchmarks

I wrote a simple JMH harness to verify those conclusions.

The benchmark (source) does something equivalent to this:

```
object.hashCode();
while(true) {
    synchronized(object) {
        counter++;
    }
}
```

One configuration (`withIdHash`) synchronizes on an object that uses the identity hash, so we expect that biased locking will be disabled as soon as `hashCode()` is invoked. A second configuration (`withoutIdHash`) implements a custom hash code so biased locking should not be disabled. Each configuration is ran first with one thread, then with two threads (these have the suffix "Contended".)

By the way, we must enable `-XX:BiasedLockingStartupDelay=0` as otherwise the JVM will take 4s to trigger the optimisation distorting the results.

The first execution:

```
Benchmark                                          Mode  Cnt        Score       Error
BiasedLockingBenchmark.withIdHash                 thrpt  100    35168,021 ±   230,252  o
BiasedLockingBenchmark.withoutIdHash              thrpt  100   173742,468 ± 4364,491  o
BiasedLockingBenchmark.withIdHashContended        thrpt  100    22478,109 ± 1650,649  o
BiasedLockingBenchmark.withoutIdHashContended     thrpt  100    20061,973 ±  786,021  o
```

We can see that the using a custom hash code makes the lock/unlock loop work 4x faster than the one using the identity hash code (which disables biased locking.) When two threads contend for the lock, biased locking is disabled anyway so there is no significative difference between both hash methods.

A second run disables biased locking (`-XX:-UseBiasedLocking`) in all configurations.

```
Benchmark                                          Mode  Cnt      Score       Error   U
BiasedLockingBenchmark.withIdHash                 thrpt  100  37374,774 ±  204,795  op
BiasedLockingBenchmark.withoutIdHash              thrpt  100  36961,826 ±  214,083  op
BiasedLockingBenchmark.withIdHashContended        thrpt  100  18349,906 ± 1246,372  op
BiasedLockingBenchmark.withoutIdHashContended     thrpt  100  18262,290 ± 1371,588  op
```

The hash method no longer has any impact and `withoutIdHash` loses its advantage.

(All benchmarks were ran on a 2,7 GHz Intel Core i5.)

References

Whatever is not wild speculation and my weak reasoning trying to make sense of the JVM sources, comes from stitching together various sources about layout, biased locking, etc. The main ones are below:

- https://blogs.oracle.com/dave/entry/biased_locking_in_hotspot
- http://fuseyism.com/openjdk/cvmi/java2vm.xhtml
- http://www.dcs.gla.ac.uk/~jsinger/pdfs/sicsa_openjdk/OpenJDKArchitecture.pdf
- https://www.infoq.com/articles/Introduction-to-HotSpot

- http://blog.takipi.com/5-things-you-didnt-know-about-synchronization-in-java-and-scala/#comment-1006598967
- http://www.azulsystems.com/blog/cliff/2010-01-09-biased-locking
- https://dzone.com/articles/why-should-you-care-about-equals-and-hashcode
- https://wiki.openjdk.java.net/display/HotSpot/Synchronization
- https://mechanical-sympathy.blogspot.com.es/2011/11/biased-locking-osr-and-benchmarking-fun.html:

This site is built with Vim and Jekyll.