Object.hashCode() algorithm

Ask Question

I'm looking for the algorithm of **Object.hashCode()**.

This code is native in **Object.java**.

Is this because

(a) the code is in assembly-- never was in Java or any other HLL at all

or

(b) it simply isn't disclosed

?

In either case, I am looking to get hold of the algorithm (pseudo-code or some detailed explanation) of "how **hashCode()** is calculated"-- what are the params going into its calculation and the calculation itself?

Please note: It's the **hashCode()** of **Object** i'm looking for-- not another like that of **String** or **hashMap/table**.

//==========

the new Java docs-- jdk 8 now saying

```
"The value returned by hashCode() is the object's hash code, which is the
object's memory address in hexadecimal."
```

java hashcode pseudocode

edited Aug 26 '14 at 19:33

asked Jul 31 '13 at 17:54

Roam

2,012 5 26 49

3 I believe that it is native because the hashCode of Object is a memory address identifier, not actually a hash of the fields of the object. – Trevor Freeman Jul 31 '13 at 17:56

Someone needs to update the docs you are quoting. Because it is completely wrong. – Björn Lindqvist May 1 '16 at 22:48

4 Answers

Native `hashCode` method implementation depends on the `JVM` . By default in HotSpot it returns random number, you can check it in the source code (function `get_next_hash`)

edited Oct 24 '14 at 18:09

answered Jul 31 '13 at 18:04

nkukhar

1,675 2 14 33

Thx for the interesting link. – Roam Jul 31 '13 at 18:10

Despite the Javadoc, the algo only may use the address as an input. This means that even though new objects use the same address in eden space they won't have the same hashCode.

There is a number of algos it might be using and not all use the address.

Note: the hashCode() is 31-bit.

BTW You can set it with `Unsafe.putInt(object, 1, value)` on Hotspot.

```
Set<Integer> ints = new LinkedHashSet<>();
int negative = 0, nonneg = 0;
for (int i = 0; i < 100; i++) {
    System.gc();
```

```
    for (int j = 0; j < 100; j++) {
        int h = new Object().hashCode();
        ints.add(h);
        if (h < 0) negative++;
        else nonneg++;
    }
}
System.out.println("unique: " + ints.size() + " negative: " + negative + " non-
neg: " + nonneg);
```

prints

```
unique: 10000 negative: 0 non-neg: 10000
```

Using Unsafe

```
Field theUnsafe = Unsafe.class.getDeclaredField("theUnsafe");
theUnsafe.setAccessible(true);
Unsafe unsafe = (Unsafe) theUnsafe.get(null);

Object o = new Object();
System.out.println("From header " + Integer.toHexString(unsafe.getInt(o, 1L)));
// sets the hashCode lazily
System.out.println("o.hashCode()  " + Integer.toHexString(o.hashCode()));
// it's here now.
System.out.println("after hashCode() From header " +
Integer.toHexString(unsafe.getInt(o, 1L)));
unsafe.putInt(o, 1L, 0x12345678);
System.out.println("after change o.hashCode()  " +
Integer.toHexString(o.hashCode()));
```

prints

```
From header 0
o.hashCode()  2260e277
after hashCode() From header 2260e277
after change o.hashCode()  12345678
```

edited Jul 31 '13 at 18:08

answered Jul 31 '13 at 18:02

Peter Lawrey

421k 54 522 890

1 "hashCode() using only the address of the object" would be consistent w/the specs at docjar.com/docs/api/java/lang/Object.html#hashCode. but then youre saying "even though new objects use the same address, they won't have the same hashCode". how then is this tie broken? some other memory location than the

object's reference address (like that of some member, the ending address/amount of memory it uses, some stuff on the timestamp...?) – Roam Jul 31 '13 at 18:15

although meets "same hashCode for same object at every call of hashCode() within one execution", solely the address of the object for this calculation doesn`t feel so right-- too consistent with memory space, not that strong on randomness of hashcode. – Roam Jul 31 '13 at 18:18

1 it would be an immutable "handle" identifier that would remain constant for the lifetime of the object. Most likely VM implementation dependent and will likely not have any relationship to the memory address, because due to GC operations all objects in java are relocatable in physical memory. – peterk Aug 28 '14 at 16:33

@peterk I believe in a very early version, objects could not be moved and indeed the address was part of the hash. (before Java 1.2) – Peter Lawrey Aug 30 '14 at 7:06

1 likely in the past as it was easy to implement, but surely not the case today or proper. The only thing appropriate to assume is that it is invariant during the lifespan of the object and unique amongst all objects in the VM instance. – peterk Sep 1 '14 at 3:39

`hashCode` is a native method, which means that a system library is called internally. This is because of the reason that hashcode internally will try to generate a number depending on the object memory location. This code is machine dependent and probably written in C.

But if you are really interested to see the native code, then follow this:

http://hg.openjdk.java.net/jdk7/jdk7-gate/jdk/file/e947a98ea3c1/src/share/native/java/

answered Jul 31 '13 at 17:59

Juned Ahsan

52.5k 7 66 100

Well, thats the thing-- is object memory location the only param to go into hashCode() calculation? how's it done-- on the lower-end bits of the address maybe? – Roam Jul 31 '13 at 18:21

It's because it relies on low-level details that aren't exposed to Java code. Some basic parts of the standard library (like `java.lang.Object`) must be implemented in native code.

As an aside, you can find at least one interesting article that does into more detail about the HotSpot implementation.

answered Jul 31 '13 at 17:57

Jeremy Roman

13.8k 1 31 37