先看一个最简单的打印

System.out.println(**new** Object());

会输出该类的全限定类名和一串字符串：

java.lang.Object@6659c656

​@符号后面的是什么？是 hashcode 还是对象的内存地址？还是其他的什么值？  
​

其实@后面的只是对象的 hashcode 值，16进制展示的 hashcode 而已，来验证一下：

Object o = **new** Object();

**int** hashcode = o.hashCode();

*// toString*

System.out.println(o);

*// hashcode 十六进制*

System.out.println(Integer.toHexString(hashcode));

*// hashcode*

System.out.println(hashcode);

*// 这个方法，也是获取对象的 hashcode；不过和 Object.hashcode 不同的是，该方法会无视重写的hashcode*

System.out.println(System.identityHashCode(o));

输出结果：

java.lang.Object@6659c656

6659c656

1717159510

1717159510

那对象的 hashcode 到底是怎么生成的呢？真的就是内存地址吗？  
​

**本文内容基于 JAVA 8 HotSpot**

hashCode 的生成逻辑

JVM 里生成 hashCode 的逻辑并没有那么简单，它提供了好几种策略，每种策略的生成结果都不同。  
​

来看一下 openjdk 源码里生成 hashCode 的[核心方法](https://github.com/openjdk/jdk/blob/7ba83041b1d65545833655293d0976dfd1ffdea8/hotspot/src/share/vm/runtime/synchronizer.cpp#L557)：

**static** **inline** **intptr\_t** **get\_next\_hash**(Thread \* Self, oop obj) {

**intptr\_t** value = 0 ;

**if** (hashCode == 0) {

*// This form uses an unguarded global Park-Miller RNG,*

*// so it's possible for two threads to race and generate the same RNG.*

*// On MP system we'll have lots of RW access to a global, so the*

*// mechanism induces lots of coherency traffic.*

value = os::random() ;

} **else**

**if** (hashCode == 1) {

*// This variation has the property of being stable (idempotent)*

*// between STW operations. This can be useful in some of the 1-0*

*// synchronization schemes.*

**intptr\_t** addrBits = intptr\_t(obj) >> 3 ;

value = addrBits ^ (addrBits >> 5) ^ GVars.stwRandom ;

} **else**

**if** (hashCode == 2) {

value = 1 ; *// for sensitivity testing*

} **else**

**if** (hashCode == 3) {

value = ++GVars.hcSequence ;

} **else**

**if** (hashCode == 4) {

value = intptr\_t(obj) ;

} **else** {

*// Marsaglia's xor-shift scheme with thread-specific state*

*// This is probably the best overall implementation -- we'll*

*// likely make this the default in future releases.*

**unsigned** t = Self->\_hashStateX ;

t ^= (t << 11) ;

Self->\_hashStateX = Self->\_hashStateY ;

Self->\_hashStateY = Self->\_hashStateZ ;

Self->\_hashStateZ = Self->\_hashStateW ;

**unsigned** v = Self->\_hashStateW ;

v = (v ^ (v >> 19)) ^ (t ^ (t >> 8)) ;

Self->\_hashStateW = v ;

value = v ;

}

value &= markOopDesc::hash\_mask;

**if** (value == 0) value = 0xBAD ;

assert (value != markOopDesc::no\_hash, "invariant") ;

TEVENT (hashCode: GENERATE) ;

**return** value;

}

从源码里可以发现，生成策略是由一个 hashCode 的全局变量控制的，默认为5；而这个变量的定义在另一个[头文件](https://github.com/openjdk/jdk/blob/7ba83041b1d65545833655293d0976dfd1ffdea8/hotspot/src/share/vm/runtime/globals.hpp#L1069)里：

product(intx, hashCode, 5,

"(Unstable) select hashCode generation algorithm" )

源码里很清楚了……**（非稳定）选择 hashCode 生成的算法**，而且这里的定义，是可以由 jvm 启动参数来控制的，先来确认下默认值：

java -XX:+PrintFlagsFinal -version | grep hashCode

intx hashCode = 5 {product}

openjdk version "1.8.0\_282"

OpenJDK Runtime **Environment** (AdoptOpenJDK)(build 1.8.0\_282-b08)

OpenJDK 64-Bit Server **VM** (AdoptOpenJDK)(build 25.282-b08, mixed mode)

**所以我们可以通过 jvm 的启动参数来配置不同的 hashcode 生成算法，测试不同算法下的生成结果：**

-XX:hashCode=N

现在来看看，每种 hashcode 生成算法的不同表现。

第 0 种算法

**if** (hashCode == 0) {

*// This form uses an unguarded global Park-Miller RNG,*

*// so it's possible for two threads to race and generate the same RNG.*

*// On MP system we'll have lots of RW access to a global, so the*

*// mechanism induces lots of coherency traffic.*

value = os::random();

}

这种生成算法，使用的一种**Park-Miller RNG**的随机数生成策略。不过需要注意的是……这个随机算法在高并发的时候会出现自旋等待

第 1 种算法

**if** (hashCode == 1) {

*// This variation has the property of being stable (idempotent)*

*// between STW operations. This can be useful in some of the 1-0*

*// synchronization schemes.*

**intptr\_t** addrBits = intptr\_t(obj) >> 3 ;

value = addrBits ^ (addrBits >> 5) ^ GVars.stwRandom ;

}

这个算法，真的是对象的内存地址了，直接获取对象的 intptr\_t 类型指针

第 2 种算法

**if** (hashCode == 2) {

value = 1 ; *// for sensitivity testing*

}

这个就不用解释了……固定返回 1，应该是用于内部的测试场景。  
​

有兴趣的同学，可以试试-XX:hashCode=2来开启这个算法，看看 hashCode 结果是不是都变成 1 了。

第 3 种算法

**if** (hashCode == 3) {

value = ++GVars.hcSequence ;

}

这个算法也很简单，自增嘛，所有对象的 hashCode 都使用这一个自增变量。来试试效果：

System.out.println(**new** Object());

System.out.println(**new** Object());

System.out.println(**new** Object());

System.out.println(**new** Object());

System.out.println(**new** Object());

System.out.println(**new** Object());

*//output*

java.lang.Object@144

java.lang.Object@145

java.lang.Object@146

java.lang.Object@147

java.lang.Object@148

java.lang.Object@149

果然是自增的……有点意思

第 4 种算法

**if** (hashCode == 4) {

value = intptr\_t(obj) ;

}

这里和第 1 种算法其实区别不大，都是返回对象地址，只是第 1 种算法是一个变体。

第 5 种算法

最后一种，**也是默认的生成算法**，hashCode 配置不等于 0/1/2/3/4 时使用该算法：

**else** {

*// Marsaglia's xor-shift scheme with thread-specific state*

*// This is probably the best overall implementation -- we'll*

*// likely make this the default in future releases.*

**unsigned** t = Self->\_hashStateX ;

t ^= (t << 11) ;

Self->\_hashStateX = Self->\_hashStateY ;

Self->\_hashStateY = Self->\_hashStateZ ;

Self->\_hashStateZ = Self->\_hashStateW ;

**unsigned** v = Self->\_hashStateW ;

v = (v ^ (v >> 19)) ^ (t ^ (t >> 8)) ;

Self->\_hashStateW = v ;

value = v ;

}

这里是通过当前状态值进行异或（XOR）运算得到的一个 hash 值，相比前面的自增算法和随机算法来说效率更高，但重复率应该也会相对增高，不过 hashCode 重复又有什么关系呢……  
​

本来 jvm 就不保证这个值一定不重复，像 HashMap 里的链地址法就是解决 hash 冲突用的