HashCode的作用原理和实例解析

# HashCode定义

（1）HashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，HashCode是用来在散列存储结构中确定对象的存储地址的；

（2）如果两个对象相同， equals方法一定返回true，并且这两个对象的HashCode一定相同；除非重写了方法

（3）如果对象的equals方法被重写，那么对象的HashCode也尽量重写，并且产生HashCode使用的对象，一定要和equals方法中使用的一致，否则就会违反上面提到的第2点；

（4）两个对象的HashCode相同，并不一定表示两个对象就相同，也就是equals方法不一定返回true，只能够说明这两个对象在散列存储结构中，如Hashtable，他们存放在同一个篮子里。

# HashCode作用

Java中的集合（Collection）有两类，一类是List，再有一类是Set。前者集合内的元素是有序的，元素可以重复；后者元素无序，但元素不可重复。 equals方法可用于保证元素不重复，但是，如果每增加一个元素就检查一次，如果集合中现在已经有1000个元素，那么第1001个元素加入集合时，就要调用1000次equals方法。这显然会大大降低效率。

于是，Java采用了哈希表的原理。

哈希算法也称为散列算法，是将数据依特定算法直接指定到一个地址上。

这样一来，当集合要添加新的元素时，先调用这个元素的HashCode方法，就一下子能定位到它应该放置的物理位置上。

（1）如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了；

（2）如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了；

（3）不相同的话，也就是发生了Hash key相同导致冲突的情况，那么就在这个Hash key的地方产生一个链表，将所有产生相同HashCode的对象放到这个单链表上去，串在一起（很少出现）。这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。 （下面1、的实例就为这里的测试实例）

## 1、如何理解HashCode的作用：

从Object角度看，JVM每new一个Object，它都会将这个Object丢到一个Hash表中去，这样的话，下次做Object的比较或者取这个对象的时候（读取过程），它会根据对象的HashCode再从Hash表中取这个对象。这样做的目的是提高取对象的效率。若HashCode相同再去调用equal。

## 2、HashCode是用于查找使用的，而equals是用于比较两个对象的是否相等的。

（1）例如内存中有这样的位置 ：

0  1  2  3  4  5  6  7

而我有个类，这个类有个字段叫ID，我要把这个类存放在以上8个位置之一，如果不用HashCode而任意存放，那么当查找时就需要到这八个位置里挨个去找，或者用二分法一类的算法。

但如果用HashCode那就会使效率提高很多。  定义我们的HashCode为ID％8，比如我们的ID为9，9除8的余数为1，那么我们就把该类存在1这个位置，如果ID是13，求得的余数是5，那么我们就把该类放在5这个位置。依此类推。

（2）但是如果两个类有相同的HashCode，例如9除以8和17除以8的余数都是1，也就是说，我们先通过 HashCode来判断两个类是否存放某个桶里，但这个桶里可能有很多类，比如hashtable，那么我们就需要再通过 equals 在这个桶里找到我们要的类。

# 1、实例重写HashCode方法

|  |
| --- |
| **public** **class** HashTest {  **private** **int** i;    **public** **int** getI() {  **return** i;  }    **public** **void** setI(**int** i) {  **this**.i = i;  }    @Override  **public** **int** hashCode() {  **return** i % 10;  }    /\*\*  \* 对象的内存地址与hashcode有关系，但并不是hashcode值相等，就是代表内存地址相同，这种想法是幼稚的  \* 比如hashtable中hashcode值相等，  \* 但是存了很多的对象，这表明对象的== 肯定不相等，Ojbect逆向推理，equals不相等，==肯定不相等  \*  \*/    **public** **final** **static** **void** main(String[] args) {  HashTest a = **new** HashTest();  HashTest b = **new** HashTest();    System.*out*.println(a.hashCode() == b.hashCode()); //true 人为制造hashcode值相同  System.*out*.println(a==b); //false //== 比较对象的相等比较对象引用地址是否相等。还要要比较对象内容是否相等  System.*out*.println(a.equals(b)); //false 不同的对象 object中 == 和euqals是一样的  a.setI(1);  b.setI(1);  Set<HashTest> set = **new** HashSet<HashTest>();  set.add(a);  set.add(b);  //没有 equels 重写的情况  System.*out*.println(a.hashCode() == b.hashCode()); //true hashcode相同    System.*out*.println(a.equals(b)); //false 不同的对象 ，创建出来的是地址就不同了    //2 这个时候会发想存入了两个值 set中存放是根据hashcode值存放，如果hashcode值相同，  //再比较equals值，如果equals值也相同，则产生一个单链表放进去  System.*out*.println(set.size());    } |

# 2、重写equels方法

|  |
| --- |
| **public** **class** HashTest {  **private** **int** i;    **public** **int** getI() {  **return** i;  }    **public** **void** setI(**int** i) {  **this**.i = i;  }    @Override  **public** **boolean** equals(Object object) {  **if** (object == **null**) {  **return** **false**;  }  **if** (object == **this**) {  **return** **true**;  }  **if** (!(object **instanceof** HashTest)) {  **return** **false**;  }  HashTest other = (HashTest) object;  **if** (other.getI() == **this**.getI()) {  **return** **true**;  }  **return** **false**;  }    @Override  **public** **int** hashCode() {  **return** i % 10;  }  **public** **final** **static** **void** main(String[] args) {  HashTest a = **new** HashTest();  HashTest b = **new** HashTest();    System.*out*.println(a.hashCode() == b.hashCode());  System.*out*.println(a==b);  System.*out*.println(a.equals(b));  a.setI(1);  b.setI(1);  Set<HashTest> set = **new** HashSet<HashTest>();  set.add(a);  set.add(b);    System.*out*.println(a.hashCode() == b.hashCode());  System.*out*.println(a.equals(b));    System.*out*.println(set.size());    }  }  } |

## 3、如果咩有重新equals 和hashcode方法，则set中也会放入2个值