[**浅谈Java中的hashcode方法**](http://www.cnblogs.com/szlbm/p/5806226.html)

　浅谈Java中的hashcode方法

　　哈希表这个数据结构想必大多数人都不陌生，而且在很多地方都会利用到hash表来提高查找效率。在Java的Object类中有一个方法:

|  |  |
| --- | --- |
| 1 | public native int hashCode(); |

　　根据这个方法的声明可知，该方法返回一个int类型的数值，并且是本地方法，因此在Object类中并没有给出具体的实现。

　　为何Object类需要这样一个方法？它有什么作用呢？今天我们就来具体探讨一下hashCode方法。

**一.hashCode方法的作用**

　　对于包含容器类型的程序设计语言来说，基本上都会涉及到hashCode。在Java中也一样，hashCode方法的主要作用是为了配合基于散列的集合一起正常运行，这样的散列集合包括HashSet、HashMap以及HashTable。

　　为什么这么说呢？考虑一种情况，当向集合中插入对象时，如何判别在集合中是否已经存在该对象了？（注意：集合中不允许重复的元素存在）

　　也许大多数人都会想到调用equals方法来逐个进行比较，这个方法确实可行。但是如果集合中已经存在一万条数据或者更多的数据，如果采用 equals方法去逐一比较，效率必然是一个问题。此时hashCode方法的作用就体现出来了，当集合要添加新的对象时，先调用这个对象的 hashCode方法，得到对应的hashcode值，实际上在HashMap的具体实现中会用一个table保存已经存进去的对象的hashcode 值，如果table中没有该hashcode值，它就可以直接存进去，不用再进行任何比较了；如果存在该hashcode值， 就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址，所以这里存在一个冲突解决的问题，这样一来实际调用 equals方法的次数就大大降低了，说通俗一点：Java中的hashCode方法就是根据一定的规则将与对象相关的信息（比如对象的存储地址，对象的 字段等）映射成一个数值，这个数值称作为散列值。下面这段代码是java.util.HashMap的中put方法的具体实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public V put(K key, V value) {          if (key == null)              return putForNullKey(value);          int hash = hash(key.hashCode());          int i = indexFor(hash, table.length);          for (Entry<K,V> e = table[i]; e != null; e = e.next) {              Object k;              if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {                  V oldValue = e.value;                  e.value = value;                  e.recordAccess(this);                  return oldValue;              }          }            modCount++;          addEntry(hash, key, value, i);          return null;      } |

　　put方法是用来向HashMap中添加新的元素，从put方法的具体实现可知，会先调用hashCode方法得到该元素的hashCode 值，然后查看table中是否存在该hashCode值，如果存在则调用equals方法重新确定是否存在该元素，如果存在，则更新value值，否则将 新的元素添加到HashMap中。从这里可以看出，hashCode方法的存在是为了减少equals方法的调用次数，从而提高程序效率。

　　如果对于hash表这个数据结构的朋友不清楚，可以参考这几篇博文;

<http://www.cnblogs.com/jiewei915/archive/2010/08/09/1796042.html>

<http://www.cnblogs.com/dolphin0520/archive/2012/09/28/2700000.html>

<http://www.java3z.com/cwbwebhome/article/article8/83560.html?id=4649>

　　有些朋友误以为默认情况下，hashCode返回的就是对象的存储地址，事实上这种看法是不全面的，确实有些JVM在实现时是直接返回对象的存储地址，但是大多时候并不是这样，只能说可能存储地址有一定关联。下面是HotSpot JVM中生成hash散列值的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | 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;  } |

　　该实现位于hotspot/src/share/vm/runtime/synchronizer.cpp文件下。

　　因此有人会说，可以直接根据hashcode值判断两个对象是否相等吗？肯定是 不可以的，因为不同的对象可能会生成相同的hashcode值。虽然不能根据hashcode值判断两个对象是否相等，但是可以直接根据hashcode 值判断两个对象不等，如果两个对象的hashcode值不等，则必定是两个不同的对象。如果要判断两个对象是否真正相等，必须通过equals方法。

　　也就是说对于两个对象，如果调用equals方法得到的结果为true，则两个对象的hashcode值必定相等；

　　如果equals方法得到的结果为false，则两个对象的hashcode值不一定不同；

　　如果两个对象的hashcode值不等，则equals方法得到的结果必定为false；

　　如果两个对象的hashcode值相等，则equals方法得到的结果未知。

**二.equals方法和hashCode方法**

　　在有些情况下，程序设计者在设计一个类的时候为需要重写equals方法，比如String类，但是千万要注意，在重写equals方法的同时，必须重写hashCode方法。为什么这么说呢？

　　下面看一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | package com.cxh.test1;    import java.util.HashMap;  import java.util.HashSet;  import java.util.Set;      class People{      private String name;      private int age;        public People(String name,int age) {          this.name = name;          this.age = age;      }        public void setAge(int age){          this.age = age;      }        @Override      public boolean equals(Object obj) {          // TODO Auto-generated method stub          return this.name.equals(((People)obj).name) && this.age== ((People)obj).age;      }  }    public class Main {        public static void main(String[] args) {            People p1 = new People("Jack", 12);          System.out.println(p1.hashCode());            HashMap<People, Integer> hashMap = new HashMap<People, Integer>();          hashMap.put(p1, 1);            System.out.println(hashMap.get(new People("Jack", 12)));      }  } |

　　在这里我只重写了equals方法，也就说如果两个People对象，如果它的姓名和年龄相等，则认为是同一个人。

　　这段代码本来的意愿是想这段代码输出结果为“1”，但是事实上它输出的是“null”。为什么呢？原因就在于重写equals方法的同时忘记重写hashCode方法。

　　虽然通过重写equals方法使得逻辑上姓名和年龄相同的两个对象被判定为相等的对象（跟String类类似），但是要知道默认情况 下，hashCode方法是将对象的存储地址进行映射。那么上述代码的输出结果为“null”就不足为奇了。原因很简单，p1指向的对象和

　　System.out.println(hashMap.get(new People("Jack", 12)));这句中的new People("Jack", 12)生成的是两个对象，它们的存储地址肯定不同。下面是HashMap的get方法的具体实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public V get(Object key) {          if (key == null)              return getForNullKey();          int hash = hash(key.hashCode());          for (Entry<K,V> e = table[indexFor(hash, table.length)];               e != null;               e = e.next) {              Object k;              if (e.hash == hash && ((k = e.key) == key || key.equals(k)))                  return e.value;          }          return null;      } |

　　所以在hashmap进行get操作时，因为得到的hashcdoe值不同（注意，上述代码也许在某些情况下会得到相同的hashcode值， 不过这种概率比较小，因为虽然两个对象的存储地址不同也有可能得到相同的hashcode值），所以导致在get方法中for循环不会执行，直接返回 null。

　　因此如果想上述代码输出结果为“1”，很简单，只需要重写hashCode方法，让equals方法和hashCode方法始终在逻辑上保持一致性。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | package com.cxh.test1;    import java.util.HashMap;  import java.util.HashSet;  import java.util.Set;      class People{      private String name;      private int age;        public People(String name,int age) {          this.name = name;          this.age = age;      }        public void setAge(int age){          this.age = age;      }        @Override      public int hashCode() {          // TODO Auto-generated method stub          return name.hashCode()\*37+age;      }        @Override      public boolean equals(Object obj) {          // TODO Auto-generated method stub          return this.name.equals(((People)obj).name) && this.age== ((People)obj).age;      }  }    public class Main {        public static void main(String[] args) {            People p1 = new People("Jack", 12);          System.out.println(p1.hashCode());            HashMap<People, Integer> hashMap = new HashMap<People, Integer>();          hashMap.put(p1, 1);            System.out.println(hashMap.get(new People("Jack", 12)));      }  } |

　　这样一来的话，输出结果就为“1”了。

　　下面这段话摘自Effective Java一书：

* 在程序执行期间，只要equals方法的比较操作用到的信息没有被修改，那么对这同一个对象调用多次，hashCode方法必须始终如一地返回同一个整数。
* 如果两个对象根据equals方法比较是相等的，那么调用两个对象的hashCode方法必须返回相同的整数结果。
* 如果两个对象根据equals方法比较是不等的，则hashCode方法不一定得返回不同的整数。

　　对于第二条和第三条很好理解，但是第一条，很多时候就会忽略。在《Java编程思想》一书中的P495页也有同第一条类似的一段话：

　　“设计hashCode()时最重要的因素就是：无论何时，对同一个对象调用 hashCode()都应该产生同样的值。如果在讲一个对象用put()添加进HashMap时产生一个hashCdoe值，而用get()取出时却产生 了另一个hashCode值，那么就无法获取该对象了。所以如果你的hashCode方法依赖于对象中易变的数据，用户就要当心了，因为此数据发生变化 时，hashCode()方法就会生成一个不同的散列码”。

　　下面举个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | package com.cxh.test1;    import java.util.HashMap;  import java.util.HashSet;  import java.util.Set;      class People{      private String name;      private int age;        public People(String name,int age) {          this.name = name;          this.age = age;      }        public void setAge(int age){          this.age = age;      }        @Override      public int hashCode() {          // TODO Auto-generated method stub          return name.hashCode()\*37+age;      }        @Override      public boolean equals(Object obj) {          // TODO Auto-generated method stub          return this.name.equals(((People)obj).name) && this.age== ((People)obj).age;      }  }    public class Main {        public static void main(String[] args) {            People p1 = new People("Jack", 12);          System.out.println(p1.hashCode());            HashMap<People, Integer> hashMap = new HashMap<People, Integer>();          hashMap.put(p1, 1);            p1.setAge(13);            System.out.println(hashMap.get(p1));      }  } |

　　这段代码输出的结果为“null”，想必其中的原因大家应该都清楚了。

　　因此，在设计hashCode方法和equals方法的时候，如果对象中的数据易变，则最好在equals方法和hashCode方法中不要依赖于该字段。

　　以上属个人理解，如有不正之处，欢迎批评指正。

分类: [java基础](http://www.cnblogs.com/szlbm/category/830568.html)