# equals()方法和hashCode()方法详解

## hashCode介绍

### 什么是hashCode

hashCode是jdk根据对象的地址或者字符串或者数字算出来的int类型的数值，也就是哈希码，哈希码并不是完全唯一的，它是一种算法，让同一个类的对象按照自己不同的特征尽量的有不同的哈希码，但不表示不同的对象哈希码完全不同。

在Java中，哈希码代表对象的特征。

例如：

String str1 = “aa”,

str1.hashCode= 3104

String str2 = “bb”,

str2.hashCode= 3106

String str3 = “aa”,

str3.hashCode= 3104

根据HashCode由此可得出

str1!=str2,

str1==str3

原文：<https://blog.csdn.net/qq_21163061/article/details/73606523>

### 1.2 hashCode()方法的作用

这个可以理解为为什么要用hashCode。

那么我们就说说为什么使用 hashcode ，hashCode 存在的第一重要的原因就是在 HashMap(HashSet 其实就是HashMap) 中使用（其实Object 类的 hashCode 方法注释已经说明了 ），我知道，HashMap 之所以速度快，因为他使用的是散列表，根据 key 的 hashcode 值生成数组下标（通过内存地址直接查找，没有任何判断），时间复杂度完美情况下可以达到 n1（和数组相同，但是比数组用着爽多了，但是需要多出很多内存，相当于以空间换时间）。

原文：<https://blog.csdn.net/qq_38182963/article/details/78940047>

### 1.3为什么大部分 hashcode 方法使用 31

如果有使用 eclipse 的同学肯定知道，该工具默认生成的 hashCode 方法实现也和 String 类型差不多。都是使用的 31 ，那么有没有想过：为什么要使用 31 呢？

在名著 《Effective Java》第 42 页就有对 hashCode 为什么采用 31 做了说明：

之所以使用 31， 是因为他是一个奇素数。如果乘数是偶数，并且乘法溢出的话，信息就会丢失，因为与2相乘等价于移位运算（低位补0）。使用素数的好处并不很明显，但是习惯上使用素数来计算散列结果。 31 有个很好的性能，即用移位和减法来代替乘法，可以得到更好的性能： 31 \* i == (i << 5） - i， 现代的 VM 可以自动完成这种优化。这个公式可以很简单的推导出来。

这个问题在 SO 上也有讨论： https://stackoverflow.com/questions/299304/why-does-javas-hashcode-in-string-use-31-as-a-multiplier）

可以看到，使用 31 最主要的还是为了性能。当然用 63 也可以。但是 63 的溢出风险就更大了。那么15 呢？仔细想想也可以。

在《Effective Java》也说道：编写这种散列函数是个研究课题，最好留给数学家和理论方面的计算机科学家来完成。我们此次最重要的是知道了为什么使用31。

原文：<https://blog.csdn.net/qq_38182963/article/details/78940047>

### 1.4 hashCode()方法重写的规则或要求

java.lang.Object.hashCode() 是三条约定是

1、多次运行 hashCode()，其值必须总是一致的（前提：1、 equals() 中用到的信息没发生变化 2、在同一次 execution 中）

2、obj1.equals(obj2) == true，则必须  obj1.hashCode() == obj1.hashCode() 总是 true

3、obj1.equals(obj2) == false，则 obj1.hashCode() == obj1.hashCode()  最好 false

        这是因为 HashMap.containsKey()，HashMap.put() 时

        a：由于 hash 不同，则直接就不尝试了（好。这样效率高啊）

        b：“两把刷子程序员” 把 hash 弄成相同的（equals()不同，hashCode()相同），还得向下尝试 equals() （不好）

原文：<https://blog.csdn.net/iceman1952/article/details/7358716>

## equals()方法

### 2.1 equals()方法的本质

超类Object中有这个equals()方法，该方法主要用于比较两个对象是否相等。该方法的源码如下：

public boolean equals(Object obj) {

return (this == obj);

}

      我们知道所有的对象都拥有标识(内存地址)和状态(数据)，同时“==”比较两个对象的的内存地址，所以说使用Object的equals()方法是比较两个对象的内存地址是否相等，即若object1.equals(object2)为true，则表示equals1和equals2实际上是引用同一个对象。

### 2.2 为什么要重写equals()方法和hashCode方法？

回答这个问题之前，我们先要知道不被重写（原生）的hashCode和equals是什么样的？

#### 2.2.1 原生equals()方法和hashCode()方法

（1）不被重写（原生）的hashCode值是根据内存地址换算出来的一个值。

（2）不被重写（原生）的equals方法是严格判断一个对象是否相等的方法（object1 == object2）。

#### 2.2.2为什么需要重写equals和hashCode方法？

##### 2.2.2.1 为什么要重写equals()方法？

默认equals在比较两个对象时，是看他们是否指向同一个地址的。

但有时，希望两个对象只要是某些属性相同就认为他们的equals为true或者只要值相等（如new Integer(2)）就认为他们的equals为true。比如：

Student s1 = new Student(1,"name"); Student s2 = new Student(1,"name");

如果不重写equals的话，他们是不相同的，因为引用s1和s2指向的堆内存地址显然不一样，所以我们要重些equals，判断只要他们的id和名字相同equals就为true，在一些集合里有时也这样用，集合里的contain也是用equals来比较。

这个时候，我们就要重写equals()方法，修改默认的判断逻辑。

##### 2.2.2.2 重写equals()方法遵循的原则

一般我们在设计一个类时，需要重写父类的equals方法，在重写这个方法时，需要按照以下几个规则设计：

1、自反性：对任意引用值X，x.equals(x)的返回值一定为true.

2、对称性：对于任何引用值x,y,当且仅当y.equals(x)返回值为true时，x.equals(y)的返回值一定为true;

3、传递性：如果x.equals(y)=true, y.equals(z)=true,则x.equals(z)=true

4、一致性：如果参与比较的对象没任何改变，则对象比较的结果也不应该有任何改变

5、非空性：任何非空的引用值X，x.equals(null)的返回值一定为false。

对于上面几个规则，我们在使用的过程中最好遵守，否则会出现意想不到的错误。

在java中进行比较，我们需要根据比较的类型来选择合适的比较方式：

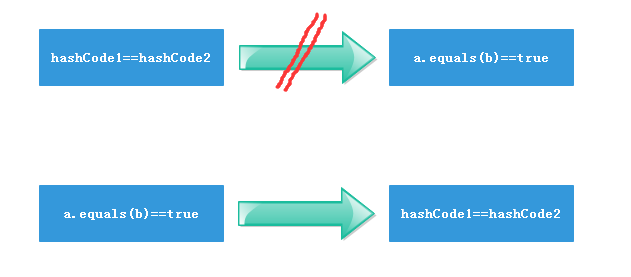
1. 对象域，使用equals方法 。
2. 类型安全的枚举，使用equals或== 。
3. 可能为null的对象域 : 使用 == 和 equals 。
4. 数组域 : 使用 Arrays.equals 。
5. 除float和double外的原始数据类型 : 使用 == 。
6. float类型: 使用Float.foatToIntBits转换成int类型，然后使用==。
7. double类型: 使用Double.doubleToLongBit转换成long类型，然后使用==。

##### 2.2.2.3 为什么重写equals()方法的同时一定要重写hashCode()方法？

首先，我们来看这两个方法之间的关系。

1. hashCode作用主要在于增加数据在哈希家族中的查询速度。
2. 如果hashCode相等，他们在哈希结构中存储位置相等，但是不是同一个对象！换句话说hashCode相等，调用equals不一定相等。
3. 如果equals相等，那么他们的存储位置当然相等，所以hashCode一定也是相等的。
4. 根据2、3两条，重写equals方法就一定要重写hashCode方法。

用图来表示他们虹之间的关系如下：



如果重写equals()而不重写hashCode会有什么结果呢？

看下面的例子：



我们从1处存进去一个以MapTest为键、以”hello”为值的键值对，结果在2处取出来，再打印，结果是null。为什么呢？因为第一次的new MapTest(s)和第二次的new MapTest(s)的对象的内存地址是不同的，用他们的内存地址作为key取值，得到的hashCode也是不同的，自然第二次就取不到结果。所以尽管equals()为true，hashCode不同，得到的结果也是不同的。

所以，如果一个类重写了equals方法，但没有重写hashCode方法，将会直接违法了第2条规定，这样的话，如果我们通过映射表(Map接口)操作相关对象时，就无法达到我们预期想要的效果。

因此，重写equals()必须重写hashCode，以保证当以“同一个对象”（两个对象相等）为键的时候，存放的位置也相同。

### 2.3 equals()方法和==的区别和联系

通过上面对hashCode()和equals()方法的分析，可以得到以下信息：

超类Object中的equals()方法其实就是用==来做判断的，此时，二者没有区别，都是比较的引用，即对象的地址。

当重写equals()方法后，equals()就变成了某种意义上的“相等”，比如：属性值相等则两个对象相等。所以，在这种情况下，equals()的意义就与==不同了。==仍然是比较引用，而equals()则变成了比较对象中的值是否相等，这个值指的是某个、某些或全部属性的值都相等，具体情况根据重写的equals()方法来定。