Java 基础提升篇：equals()与 hashCode()方法详解

概述

java.lang.Object 类中有两个非常重要的方法：

```
public boolean equals(Object obj)
public int hashCode()
```

Object 类是类继承结构的基础，所以是每一个类的父类。所有的对象，包括数组，都实现了在 Object 类中定义的方法。

equals()方法详解

equals()方法是用来判断其他的对象是否和该对象相等。
equals()方法在 object 类中定义如下：

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

很明显是对两个对象的地址值进行的比较（即比较引用是否相同）。但是我们知道，String 、Math、Integer、Double 等这些封装类在使用 equals()方法时，已经覆盖了 object 类的 equals()方法。
比如在 String 类中如下：

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = count;
        if (n == anotherString.count) {
            char v1[] = value;
```

```
            char v2[] = anotherString.value;
            int i = offset;
            int j = anotherString.offset;
            while (n- != 0) {
                if (v1[i++] != v2[j++])
                    return false;
            }
            return true;
        }
    }
    return false;
}
```

很明显，这是进行的内容比较，而已经不再是地址的比较。依次类推 Math、Integer、Double 等这些类都是重写了 equals()方法的，从而进行的是内容的比较。当然，基本类型是进行值的比较。

它的性质有：

- **自反性（reflexive）**。对于任意不为 null 的引用值 x，x.equals(x)一定是 true。
- **对称性（symmetric）**。对于任意不为 null 的引用值 x 和 y，当且仅当 x.equals(y)是 true 时，y.equals(x)也是 true。
- **传递性（transitive）**。对于任意不为 null 的引用值 x、y 和 z，如果 x.equals(y)是 true，同时 y.equals(z)是 true，那么 x.equals(z)一定是 true。
- **一致性（consistent）**。对于任意不为 null 的引用值 x 和 y，如果用于 equals 比较的对象信息没有被修改的话，多次调用时 x.equals(y)要么一致地返回 true 要么一致地返回 false。
- 对于任意不为 null 的引用值 x，x.equals(null)返回 false。

对于 Object 类来说，equals()方法在对象上实现的是差别可能性最大的等价关系，即，对于任意非 null 的引用值 x 和 y，当且仅当 x 和 y 引用的是同一个对象，该方法才会返回 true。

需要注意的是当 equals()方法被 override 时，hashCode()也要被 override。按照一般 hashCode()方法的实现来说，相等的对象，它们的 hashCode 一定相等。

hashcode() 方法详解

hashCode()方法给对象返回一个 hashcode 值。这个方法被用于 hashtables，例如 HashMap。

它的性质是：

- 在一个 Java 应用的执行期间，如果一个对象提供给 equals 做比较的信息没有被修改的话，该对象多次调用 hashCode()方法，该方法必须始终如一返回同一个 integer。
- 如果两个对象根据 equals(Object)方法是相等的，那么调用二者各自的 hashCode()方法必须产生同一个 integer 结果。
- 并不要求根据 equals(java.lang.Object)方法不相等的两个对象，调用二者各自的 hashCode()方法必须产生不同的 integer 结果。然而，程序员应该意识到对于不同的对象产生不同的 integer 结果，有可能会提高 hashtable 的性能。

大量的实践表明，由 Object 类定义的 hashCode()方法对于不同的对象返回不同的 integer。在 object 类中，hashCode 定义如下：

```
public native int hashCode();
```

说明是一个本地方法，它的实现是根据本地机器相关的。当然我们可以在自己写的类中覆盖 hashcode()方法，比如 String、Integer、Double 等这些类都是覆盖了 hashcode()方法的。例如在 String 类中定义的 hashcode()方法如下：

```
public int hashCode() {
    int h = hash;
    if (h == 0) {
        int off = offset;
        char val[] = value;
        int len = count;
        for (int i = 0; i < len; i++) {
            h = 31 * h + val[off++];
        }
        hash = h;
    }
    return h;
}
```

解释一下这个程序（String 的 API 中写到）：s[0]*31^(n-1) + s[1]*31^(n-2) + … + s[n-1]

使用 int 算法，这里 s[i] 是字符串的第 i 个字符，n 是字符串的长度，^ 表示求幂（空字符串的哈希码为 0）。

想要弄明白 hashCode 的作用，必须要先知道 Java 中的集合。

总的来说，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。这里就引出一个问题：要想保证元素不重复，可两个元素是否重复应该依据什么来判断呢？

这就是 Object.equals 方法了。但是，如果每增加一个元素就检查一次，那么当元素很多时，后添加到集合中的元素比较的次数就非常多了。也就是说，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，它就要调用 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（Hash）实际上是个人名，由于他提出一哈希算法的概念，所以就以他的名字命名了。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，初学者可以简单理解，hashCode 方法实际上返回的就是对象存储的物理地址（实际可能并不是）。

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。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；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，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，几乎只需要一两次。

简而言之，在集合查找时，hashcode 能大大降低对象比较次数，提高查找效率！

Java 对象的 eqauls 方法和 hashCode 方法是这样规定的：

1) **相等（相同）的对象必须具有相等的哈希码（或者散列码）。**
2) **如果两个对象的 hashCode 相同，它们并不一定相同。**

以下是 Object 对象 API 关于 equal 方法和 hashCode 方法的说明：

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.
- 以上 API 说明是对之前 2 点的官方详细说明

关于第一点，相等（相同）的对象必须具有相等的哈希码（或者散列码），为什么？

想象一下，假如两个 Java 对象 A 和 B，A 和 B 相等（eqauls 结果为 true），但 A 和 B 的哈希码不同，则 A 和 B 存入 HashMap 时的哈希码计算得到的 HashMap 内部数组位置索引可能不同，那么 A 和 B 很有可能允许同时存入 HashMap，显然相等/相同的元素是不允许同时存入 HashMap，HashMap 不允许存放重复元素。

关于第二点，两个对象的 hashCode 相同，它们并不一定相同

也就是说，不同对象的 hashCode 可能相同；假如两个 Java 对象 A 和 B，A 和 B 不相等（eqauls 结果为 false），但 A 和 B 的哈希码相等，将 A 和 B 都存入 HashMap 时会发生哈希冲突，也就是 A 和 B 存放在 HashMap 内部数组的位置索引相同，这时 HashMap 会在该位置建立一个链接表，将 A 和 B 串起来放在该位置，显然，该情况不违反 HashMap 的使用原则，是允许的。当然，哈希冲突越少越好，尽量采用好的哈希算法以避免哈希冲突。

所以，Java 对于 eqauls 方法和 hashCode 方法是这样规定的：

1) 如果两个对象相同，那么它们的 hashCode 值一定要相同；

如果两个对象的 hashCode 相同，它们并不一定相同（这里说的对象相同指的是用 eqauls 方法比较）。

2) 如不按要求去做了，会发现相同的对象可以出现在 Set 集合中，同时，增加新元素的效率会大大下降。

3) equals()相等的两个对象，hashcode()一定相等；equals()不相等的两个对象，却并不能证明他们的 hashcode()不相等。

4) 换句话说，equals()方法不相等的两个对象，hashcode()有可能相等（我的理解是由于哈希码在生成的时候产生冲突造成的）。反过来，hashcode()不等，一定能推出 equals()也不等；hashcode()相等，equals()可能相等，也可能不等。

在 object 类中，hashcode()方法是本地方法，返回的是对象的地址值，而 object 类中的 equals()方法比较的也是两个对象的地址值，如果 equals()相等，说明两个对象地址值也相等，当然 hashcode()也就相等了；在 String 类中，equals()返回的是两个对象内容的比较，当两个对象内容相等时，Hashcode()方法根据 String 类的重写代码的分析，也可知道 hashcode()返回结果也会相等。以此类推，可以知道 Integer、Double 等封装类中经过重写的 equals()和 hashcode()方法也同样适合于这个原则。当然没有经过重写的类，在继承了 object 类的 equals()和 hashcode()方法后，也会遵守这个原则。

Hashset、Hashmap、Hashtable 与 hashcode()和 equals()的密切关系

Hashset 是继承 Set 接口，Set 接口又实现 Collection 接口，这是层次关系。那么 Hashset、Hashmap、Hashtable 中的存储操作是根据什么原理来存取对象的呢？

下面以 HashSet 为例进行分析，我们都知道：在 hashset 中不允许出现重复对象，元素的位置也是不确定的。在 hashset 中又是怎样判定元素是否重复的呢？在 java 的集合中，判断两个对象是否相等的规则是：

1) 判断两个对象的 hashCode 是否相等

如果不相等，认为两个对象也不相等，完毕

如果相等，转入 2)

（这一点只是为了提高存储效率而要求的，其实理论上没有也可以，但如果没有，实际使用时效率会大大降低，所以我们这里将其做为必需的。）

2) 判断两个对象用 equals 运算是否相等

如果不相等，认为两个对象也不相等

如果相等，认为两个对象相等（equals()是判断两个对象是否相等的关键）

为什么是两条准则，难道用第一条不行吗？不行，因为前面已经说了，hashcode()相等时，equals()方法也可能不等，所以必须用第 2 条准则进行限制，才能保证加入的为非重复

元素。

例 1：

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
public class HashSetTest {
    public static void main(String args[]) {
        String s1 = new String("aaa");
        String s2 = new String("aaa");
        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
        System.out.println(s1.hashCode());
        System.out.println(s2.hashCode());
        Set hashset = new HashSet();
        hashset.add(s1);
        hashset.add(s2);
        Iterator it = hashset.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

运行结果：

```
false
true
96321
96321
aaa
```

这是因为 String 类已经重写了 equals()方法和 hashcode()方法，所以 hashset 认为它们是相等的对象，进行了重复添加。

例 2：

```
import java.util.HashSet;
import java.util.Iterator;
public class HashSetTest {
    public static void main(String[] args) {
        HashSet hs = new HashSet();
```

```
        hs.add(new Student(1, "zhangsan"));
        hs.add(new Student(2, "lisi"));
        hs.add(new Student(3, "wangwu"));
        hs.add(new Student(1, "zhangsan"));
        Iterator it = hs.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
class Student {
    int num;
    String name;
    Student(int num, String name) {
        this.num = num;
        this.name = name;
    }
    public String toString() {
        return num + ":" + name;
    }
}
```

运行结果：

```
1:zhangsan
3:wangwu
2:lisi
1:zhangsan
```

为什么 hashset 添加了相等的元素呢，这是不是和 hashset 的原则违背了呢？回答是：没有。因为在根据 hashcode()对两次建立的 new Student(1,“zhangsan”)对象进行比较时，生成的是不同的哈希码值，所以 hashset 把他当作不同的对象对待了，当然此时的 equals()方法返回的值也不等。

为什么会生成不同的哈希码值呢？上面我们在比较 s1 和 s2 的时候不是生成了同样的哈希码吗？原因就在于我们自己写的 Student 类并没有重写自己的 hashcode()和 equals()方法，所以在比较时，是继承的 object 类中的 hashcode()方法，而 object 类中的 hashcode()方法是一个本地方法，比较的是对象的地址（引用地址），使用 new 方法创建对象，两次生成的当然是不同的对象了，造成的结果就是两个对象的 hashcode()返回的值不一样，所以 Hashset 会把它们当作不同的对象对待。

怎么解决这个问题呢？答案是：在 Student 类中重写 hashcode()和 equals()方法。

```
class Student {
    int num;
    String name;
    Student(int num, String name) {
        this.num = num;
        this.name = name;
    }
    public int hashCode() {
        return num * name.hashCode();
    }
    public boolean equals(Object o) {
        Student s = (Student) o;
        return num == s.num && name.equals(s.name);
    }
    public String toString() {
        return num + ":" + name;
    }
}
```

运行结果：

```
1:zhangsan
3:wangwu
2:lisi
```

可以看到重复元素的问题已经消除，根据重写的方法，即便两次调用了 new Student(1,"zhangsan")，我们在获得对象的哈希码时，根据重写的方法 hashcode()，获得的哈希码肯定是一样的，当然根据 equals()方法我们也可判断是相同的，所以在向 hashset 集合中添加时把它们当作重复元素看待了。

重写 equals()和 hashcode()小结

1. 重点是 equals，重写 hashCode 只是技术要求（为了提高效率）
2. 为什么要重写 equals 呢？因为在 java 的集合框架中，是通过 equals 来判断两个对象是否相等的
3. 在 hibernate 中，经常使用 set 集合来保存相关对象，而 set 集合是不允许重复的。在向 HashSet 集合中添加元素时，其实只要重写 equals()这一条也可以。但当 hashset 中元素比较多时，或者是重写的 equals()方法比较复杂时，我们只用 equals()方法进行比较判断，效率也会非常低，所以引入了 hashCode()这个方法，只是为了提高效率，且这是

非常有必要的。

比如可以这样写：

```
public int hashCode(){
   return 1; //等价于 hashcode 无效
}
```

这样做的效果就是在比较哈希码的时候不能进行判断，因为每个对象返回的哈希码都是1，每次都必须要经过比较 equals()方法后才能进行判断是否重复，这当然会引起效率的大大降低。