 Java equals方法实现总结

作者: Denlee

原文链接: <http://blog.csdn.net/denlee/archive/2008/10/15/3081722.aspx>

    所有的对象都有标识(内存中的地址)和状态(对象的数据)。'=='运算符比较两个对象的地址，Object类的equals方法的默认实现也是按照内存地址比较对象是否相等，因此如果 object1.equals(object2)为true，表明object1变量和object2变量实际上引用同一个对象。  
    有些时候，默认的equals方法的实现就可以满足要求；但是很多时候我们需要比较对象的数据，特别是由数据库记录映射的对象。  
    Java规范建议equals方法遵循以下几个特性:  
1) 自反性：对于任何非空引用 x, x.equals(Object) 将返回 true;   
2) 对称性：对于任何引用 x 和 y，当且仅当 y.equals(x)返回 true 时，x.equals(y)返回 true;   
3) 传递性：对于任何引用 x、y 和 z，如果x.equals(y) 返回true 并且 y.equals(z)也返回true，那么 x.equals(z) 也应该返回 true;   
4) 一致性：如果 x 和 y 引用的对象没有改变，那么 x.equals(y)的重复调用应该返回同一结果；   
5) 对任何非空引用 x， x.equals(null)应该返回false。  
    在实现equals方法时，应该根据实例域的类型进行不同的比较：  
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==

举例:

1. //equals方法的参数保持为Object类型，否则在动态绑定调用机制会调用超类的equals方法
2. **public** **boolean** equals(Object o) {
3. //先检查是否是自比较
4. **if** ( **this** == o ) **return** **true**;
5. **if** ( o == **null** || o.getClass() != **this**.getClass() ) **return** **false**;
6. //转换参数的类型
7. MyObject that = (MyObject)o;
8. //完成所需域的比较
9. **return**  ...;
10. }

    尽管在参考资料[2]中提到，如果有以下2方面的原因使得我们可以使用instanceof代替getClass()：   
    1) 如果需要匹配超类; Java的多态性, 可以将一个子类的实例赋值给一个超类的实例实例  
    2) "null instanceof [type]"总是返回false，隐含了aThat == null检查(参见 Effective Java by Joshua Bloch.)   
    也就是说用if ( !(aThat instanceof MyObject) ) return false; 来代替if ( aThat == null || aThat.getClass() != this.getClass() ) return false;  
    但是在实际应用时应该加以小心，比如MyObject是超类，而MySubObject是子类，看下面的代码：

1. **public** **class** MyObject
2. {
3. **public** **int** x;
4. **public** MyObject(**int** x){**this**.x = x;}
5. **public** **boolean** equals(Object o)
6. {
7. **if**(**this** == o)**return** **true**;
8. **if**(!(o **instanceof** MyObject))**return** **false**;
9. **return** ((MyObject)o).x == x;
10. }
11. }
12. **public** **class** MySubObject **extends** MyObject
13. {
14. **public** **int** x;
15. **public** String text;
16. **public** MySubObject(**int** x, String text)
17. {
18. **super**(x);
19. **this**.x = x;
20. **this**.text = text;
21. }
22. **public** **boolean** equals(Object o)
23. {
24. **if**(**this** == o)**return** **true**;
25. **if**(!(o **instanceof** MySubObject))**return** **false**;
26. MySubObject obj = (MySubObject)o;
27. **return** (x == obj.x && text.equals(obj.text));
28. }
29. }

测试代码如下：

1. MyObject o1 = **new** MyObject(1);
2. MyObject o2 = **new** MySubObject(1, "MySubObject's instance");
3. System.out.println(o1.equals(o2));
4. System.out.println(o2.equals(o1));

    尽管我们都知道这2个测试都应该输出false，因为o2实际上是子类的实例的引用，而子类除了比较int数据成员外，还需要比较String数据成员，而超类仅需比较int数据成员；但是结果发现第1个测试输出true，而第二个测试输出false。因为第一个测试调用的是超类的equals方法，并且子类的实例o2匹配了超类，因此返回true。

    由此可见，在实现equals方法时最好还是使用getClass()进行比较完成实例到类的匹配更为可靠。

    每当覆盖 equals() 时，也应该覆盖 hashCode()，以便将类的实例插到散列表中。hashCode() 应返回一个整数值，并遵循的绝对原则是，它必须返回：  
1) 同一对象的相同值。   
2) 相等对象的相等值。   
    每个对象都有一个默认的散列码，就是对象的地址。覆盖hashCode方法的返回值，一般来说，可以返回一个域的hashcode或多个域的hashcode与常量相乘后求和，也可以是将实例域转换为String并组合它们，然后返回结果String的散列码。

    个人认为，至于是不是要覆盖hashCode要具体问题具体分析，如果在比较相等时需要比较hashcode则要覆盖，否则无需覆盖；比如集合类有contains方法，contains方法在实现时如果也比较了hashcode，则一定要覆盖hashCode方法；另一个例子是，如果将类的实例作为key放到散列表中，则应该覆盖hashCode方法。当然，同时覆盖hashCode是一个好的习惯。

参考资料:   
1. Implementing equals, <http://www.javapractices.com/topic/TopicAction.do?Id=17>   
2. Item 6 ：按照常规去覆盖equals方法, <http://www.douban.com/subject/discussion/1106608/>   
3. Java编程介绍--增强对象--覆盖equals(), <http://www.java3z.com/cwbwebhome/article/article4/del/j-intjava/j-intjava-10-10.html>   
4. JAVA指导:深入equals方法, <http://www.examda.com/JAVA/Instructs/060522/154004211.html>   
5. JAVA2核心技术第1卷：基础知识(第7版)