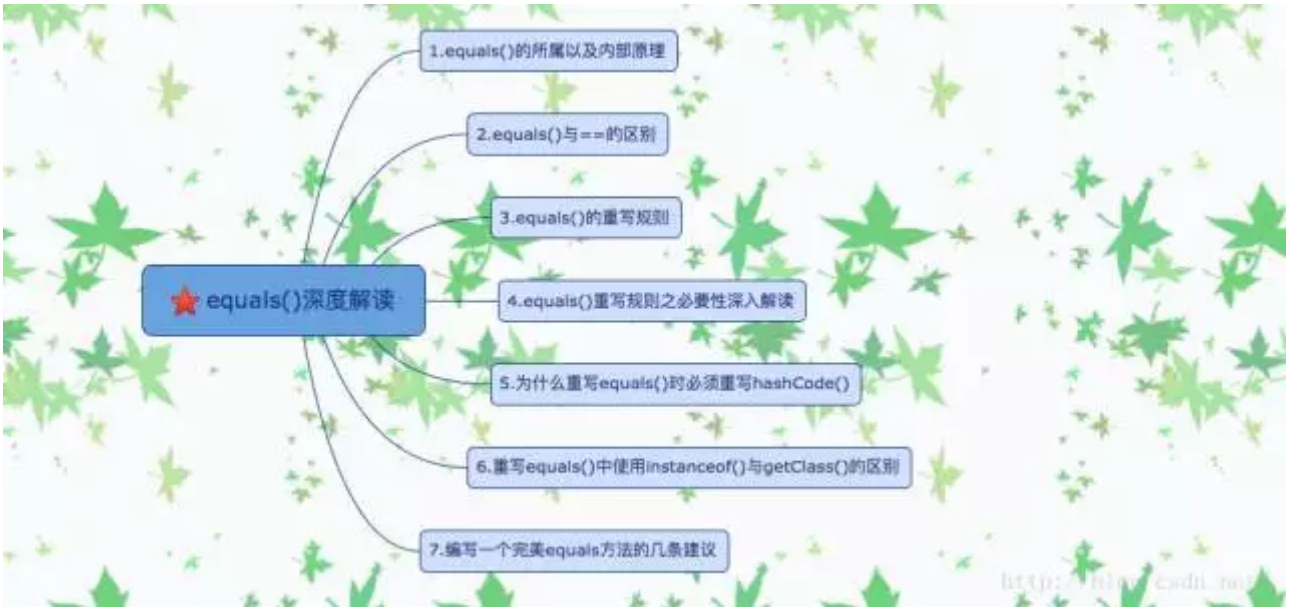
重写equals()时为什么也得重写hashCode()之深度解读equals方法与hashCode方法渊源（上）

2016-08-12 安卓应用频道

(点击上方公众号，可快速关注)

> 来源：伯乐在线专栏作者 - shine_zejian
> 链接：http://android.jobbole.com/84152/
> 点击 → 了解如何加入专栏作者

今天这篇文章我们打算来深度解读一下equal方法以及其关联方法hashCode()，我们准备从以下几点入手分析：



1.equals()的所属以及内部原理（即Object中equals方法的实现原理）

说起equals方法，我们都知道是超类Object中的一个基本方法，用于检测一个对象是否与另外一个对象相等。而在Object类中这个方法实际上是判断两个对象是否具有相同的引用，如果有，它们就一定相等。其源码如下：

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

实际上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 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.equals()与‘==’的区别

或许这是我们面试时更容易碰到的问题”equals方法与‘==’运算符有什么区别？“，并且常常我们都会胸有成竹地回答：“equals比较的是对象的内容，而‘==’比较的是对象的地址。”。但是从前面我们可以知道equals方法在Object中的实现也是间接使用了‘==’运算符进行比较的，所以从严格意义上来说，我们前面的回答并不完全正确。我们先来看一段代码并运行再来讨论这个问题。

```
package com.zejian.test;
public class Car {
    private int batch;
    public Car(int batch) {
        this.batch = batch;
    }
    public static void main(String[] args) {
        Car c1 = new Car(1);
        Car c2 = new Car(1);
        System.out.println(c1.equals(c2));
        System.out.println(c1 == c2);
    }
}
```

运行结果：

```
false

false
```

分析：对于‘==’运算符比较两个Car对象，返回了false，这点我们很容易明白，毕竟它们比较的是内存地址，而c1与c2是两个不同的对象，所以c1与c2的内存地址自然也不一样。现在的问题是，我们希望生产的两辆的批次（batch）相同的情况下就认为这两辆车相等，但是运行的结果是尽管c1与c2的批次相同，但equals的结果却反回了false。当然对于equals返回了false，我们也是心知肚明的，因为equal来自Object超类，访问修饰符为public，而我们并没有重写equal方法，故调用的必然是Object超类的原始方equals方法，根据前面分析我们也知道该原始equal方法内部实现使用的是’==’运算符，所以返回了false。因此为了达到我们的期望值，我们必须重写Car的equal方法，让其比较的是对象的批次（即对象的内容），而不是比较内存地址，于是修改如下：

```
@Override
    public boolean equals(Object obj) {
        if (obj instanceof Car) {
            Car c = (Car) obj;
            return batch == c.batch;
```

```
        }
        return false;
    }
```

使用instanceof来判断引用obj所指向的对象的类型，如果obj是Car类对象，就可以将其强制转为Car对象，然后比较两辆Car的批次，相等返回true，否则返回false。当然如果obj不是 Car对象，自然也得返回false。我们再次运行：

```
true

false
```

嗯，达到我们预期的结果了。因为前面的面试题我们应该这样回答更佳
总结：默认情况下也就是从超类Object继承而来的equals方法与‘==’是完全等价的，比较的都是对象的内存地址，但我们可以重写equals方法，使其按照我们的需求的方式进行比较，如String类重写了equals方法，使其比较的是字符的序列，而不再是内存地址。

3.equals()的重写规则

前面我们已经知道如何去重写equals方法来实现我们自己的需求了，但是我们在重写equals方法时，还是需要注意如下几点规则的。

- 自反性。对于任何非null的引用值x，x.equals(x)应返回true。
- 对称性。对于任何非null的引用值x与y，当且仅当：y.equals(x)返回true时，x.equals(y)才返回true。
- 传递性。对于任何非null的引用值x、y与z，如果y.equals(x)返回true，y.equals(z)返回true，那么x.equals(z)也应返回true。
- 一致性。对于任何非null的引用值x与y，假设对象上equals比较中的信息没有被修改，则多次调用x.equals(y)始终返回true或者始终返回false。
- 对于任何非空引用值x，x.equal(null)应返回false。

当然在通常情况下，如果只是进行同一个类两个对象的相等比较，一般都可以满足以上5点要求，下面我们来看前面写的一个例子。

```
package com.zejian.test;
public class Car {
    private int batch;
    public Car(int batch) {
        this.batch = batch;
    }
    public static void main(String[] args) {
```

```
        Car c1 = new Car(1);
        Car c2 = new Car(1);
        Car c3 = new Car(1);
        System.out.println("自反性->c1.equals(c1)：" + c1.equals(c1));
        System.out.println("对称性：");
        System.out.println(c1.equals(c2));
        System.out.println(c2.equals(c1));
        System.out.println("传递性：");
        System.out.println(c1.equals(c2));
        System.out.println(c2.equals(c3));
        System.out.println(c1.equals(c3));
        System.out.println("一致性：");
        for (int i = 0; i < 50; i++) {
            if (c1.equals(c2) != c1.equals(c2)) {
                System.out.println("equals方法没有遵守一致性！");
                break;
            }
        }
        System.out.println("equals方法遵守一致性！");
        System.out.println("与null比较：");
        System.out.println(c1.equals(null));
    }
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Car) {
            Car c = (Car) obj;
            return batch == c.batch;
        }
        return false;
    }
}
```

运行结果：

```
自反性->c1.equals(c1)：true

对称性：

true

true
```

传递性：

true

true

true

一致性：

equals方法遵守一致性！

与null比较：

false

由运行结果我们可以看出equals方法在同一个类的两个对象间的比较还是相当容易理解的。但是如果是子类与父类混合比较，那么情况就不太简单了。下面我们来看看另一个例子，首先，我们先创建一个新类BigCar，继承于Car,然后进行子类与父类间的比较。

```
package com.zejian.test;
public class BigCar extends Car {
    int count;
    public BigCar(int batch, int count) {
        super(batch);
        this.count = count;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof BigCar) {
            BigCar bc = (BigCar) obj;
            return super.equals(bc) && count == bc.count;
        }
        return false;
    }
    public static void main(String[] args) {
        Car c = new Car(1);
        BigCar bc = new BigCar(1, 20);
        System.out.println(c.equals(bc));
        System.out.println(bc.equals(c));
```

```
    }
}
```

运行结果：

```
true

false
```

对于这样的结果，自然是我们意料之中的啦。因为BigCar类型肯定是属于Car类型，所以c.equals(bc)肯定为true，对于bc.equals(c)返回false，是因为Car类型并不一定是BigCar类型（Car类还可以有其他子类）。嗯，确实是这样。但如果有这样一个需求，只要BigCar和Car的生产批次一样，我们就认为它们两个是相当的，在这样一种需求的情况下，父类（Car）与子类（BigCar）的混合比较就不符合equals方法对称性特性了。很明显一个返回true，一个返回了false，根据对称性的特性，此时两次比较都应该返回true才对。那么该如何修改才能符合对称性呢？其实造成不符合对称性特性的原因很明显，那就是因为Car类型并不一定是BigCar类型（Car类还可以有其他子类），在这样的情况下(Car instanceof BigCar)永远返回false，因此，我们不应该直接返回false，而应该继续使用父类的equals方法进行比较才行（因为我们的需求是批次相同，两个对象就相等，父类equals方法比较的就是batch是否相同）。因此BigCar的equals方法应该做如下修改：

```
@Override
public boolean equals(Object obj) {
    if (obj instanceof BigCar) {
        BigCar bc = (BigCar) obj;
        return super.equals(bc) && count == bc.count;
    }
    return super.equals(obj);
}
```

这样运行的结果就都为true了。但是到这里问题并没有结束，虽然符合了对称性，却还没符合传递性，实例如下：

```
package com.zejian.test;
public class BigCar extends Car {
    int count;
    public BigCar(int batch, int count) {
        super(batch);
        this.count = count;
    }
    @Override
```

```
    public boolean equals(Object obj) {
        if (obj instanceof BigCar) {
            BigCar bc = (BigCar) obj;
            return super.equals(bc) && count == bc.count;
        }
        return super.equals(obj);
    }
    public static void main(String[] args) {
        Car c = new Car(1);
        BigCar bc = new BigCar(1, 20);
        BigCar bc2 = new BigCar(1, 22);
        System.out.println(bc.equals(c));
        System.out.println(c.equals(bc2));
        System.out.println(bc.equals(bc2));
    }
}
```

运行结果：

```
true

true

false
```

bc，bc2，c的批次都是相同的，按我们之前的需求应该是相等，而且也应该符合equals的传递性才对。但是事实上运行结果却不是这样，违背了传递性。出现这种情况根本原因在于：

- 父类与子类进行混合比较。
- 子类中声明了新变量，并且在子类equals方法使用了新增的成员变量作为判断对象是否相等的条件。

只要满足上面两个条件，equals方法的传递性便失效了。而且目前并没有直接的方法可以解决这个问题。因此我们在重写equals方法时这一点需要特别注意。虽然没有直接的解决方法，但是间接的解决方案还说有滴，那就是通过组合的方式来代替继承,还有一点要注意的是组合的方式并非真正意义上的解决问题（只是让它们间的比较都返回了false，从而不违背传递性，然而并没有实现我们上面batch相同对象就相等的需求），而是让equals方法满足各种特性的前提下，让代码看起来更加合情合理，代码如下：

```
package com.zejian.test;
public class Combination4BigCar {
```

```
    private Car c;
    private int count;
    public Combination4BigCar(int batch, int count) {
        c = new Car(batch);
        this.count = count;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Combination4BigCar) {
            Combination4BigCar bc = (Combination4BigCar) obj;
            return c.equals(bc.c) && count == bc.count;
        }
        return false;
    }
}
```

从代码来看即使batch相同，Combination4BigCar类的对象与Car类的对象间的比较也永远都是false，但是这样看起来也就合情合理了，毕竟Combination4BigCar也不是Car的子类，因此equals方法也就没必要提供任何对Car的比较支持，同时也不会违背了equals方法的传递性。

4.equals()的重写规则之必要性深入解读

前面我们一再强调了equals方法重写必须遵守的规则，接下来我们就是分析一个反面的例子，看看不遵守这些规则到底会造成什么样的后果。

```
package com.zejian.test;
import java.util.ArrayList;
import java.util.List;
/** * 反面例子 * @author zejian */
public class AbnormalResult {
    public static void main(String[] args) {
        List<A> list = new ArrayList<A>();
        A a = new A();
        B b = new B();
        list.add(a);
        System.out.println("list.contains(a)->" + list.contains(a));
        System.out.println("list.contains(b)->" + list.contains(b));
        list.clear();
        list.add(b);
        System.out.println("list.contains(a)->" + list.contains(a));
        System.out.println("list.contains(b)->" + list.contains(b));
```

```
    }
    static class A {
        @Override
        public boolean equals(Object obj) {
            return obj instanceof A;
        }
    }
    static class B extends A {
        @Override
        public boolean equals(Object obj) {
            return obj instanceof B;
        }
    }
}
```

上面的代码，我们声明了 A,B两个类，注意必须是static，否则无法被main调用。B类继承A，两个类都重写了equals方法，但是根据我们前面的分析，这样重写是没有遵守对称性原则的，我们先来看看运行结果：

```
list.contains(a)->true

list.contains(b)->false

list.contains(a)->true

list.contains(b)->true
```

19行和24行的输出没什么好说的，将a，b分别加入list中，list中自然会含有a，b。但是为什么20行和23行结果会不一样呢？我们先来看看contains方法内部实现

```
@Override
public boolean contains(Object o) {
    return indexOf(o) != -1;
}
```

进入indexof方法

```
@Override
ublic int indexOf(Object o) {
E[] a = this.a;
if (o == null) {
```

```
    for (int i = 0; i < a.length; i++)
        if (a[i] == null)
            return i;
} else {
    for (int i = 0; i < a.length; i++)
        if (o.equals(a[i]))
            return i;
}
return -1;
```

可以看出最终调用的是对象的equals方法，所以当调用20行代码list.contains(b)时，实际上调用了

b.equals(a[i]),a[i]是集合中的元素集合中的类型而且为A类型(只添加了a对象)，虽然B继承了A,但此时

```
a[i] instanceof B
```

结果为false，equals方法也就会返回false；而当调用23行代码list.contains(a)时，实际上调用了a.equal(a[i]),其中a[i]是集合中的元素而且为B类型(只添加了b对象)，由于B类型肯定是A类型（B继承了A），所以

```
a[i] instanceof A
```

结果为true，equals方法也就会返回true，这就是整个过程。但很明显结果是有问题的，因为我们的 list的泛型是A,而B又继承了A，此时无论加入了a还是b，都属于同种类型，所以无论是contains(a),还是contains(b)都应该返回true才算正常。而最终却出现上面的结果，这就是因为重写equals方法时没遵守对称性原则导致的结果，如果没遵守传递性也同样会造成上述的结果。当然这里的解决方法也比较简单，我们只要将B类的equals方法修改一下就可以了。

```
static class B extends A{
    @Override
    public boolean equals(Object obj) {
        if(obj instanceof B){
            return true;
        }
        return super.equals(obj);
    }
}
```

到此，我们也应该明白了重写equals必须遵守几点原则的重要性了。当然这里不止是list，只要是java集合类或者java类库中的其他方法，重写equals不遵守5点原则的话，都可能出现意想不到的结果。

接下文

阅读原文