

## 08 | 判等问题：程序里如何确定你就是你？

2020-03-26 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)



讲述：王少泽

时长 20:30 大小 18.79M



你好，我是朱晔。今天，我来和你聊聊程序里的判等问题。

你可能会说，判等不就是一行代码的事情吗，有什么好说的。但，这一行代码如果处理不当，不仅会出现 Bug，还可能会引起内存泄露等问题。涉及判等的 Bug，即使是使用 `==` 这种错误的判等方式，也不是所有时候都会出问题。所以类似的判等问题不太容易发现，可能会被隐藏很久。

今天，我就 `equals`、`compareTo` 和 Java 的数值缓存、字符串驻留等问题展开讨论，你可以理解其原理，彻底消除业务代码中的相关 Bug。



### 注意 `equals` 和 `==` 的区别

在业务代码中，我们通常使用 equals 或 == 进行判等操作。equals 是方法而 == 是操作符，它们的使用是有区别的：

对基本类型，比如 int、long，进行判等，只能使用 ==，比较的是直接值。因为基本类型的值就是其数值。

对引用类型，比如 Integer、Long 和 String，进行判等，需要使用 equals 进行内容判等。因为引用类型的直接值是指针，使用 == 的话，比较的是指针，也就是两个对象在内存中的地址，即比较它们是不是同一个对象，而不是比较对象的内容。

这就引出了我们必须要知道的第一个结论：**比较值的内容，除了基本类型只能使用 == 外，其他类型都需要使用 equals。**

在开篇我提到了，即使使用 == 对 Integer 或 String 进行判等，有些时候也能得到正确结果。这又是为什么呢？

我们用下面的测试用例深入研究下：


使用 == 对两个值为 127 的直接赋值的 Integer 对象判等；

使用 == 对两个值为 128 的直接赋值的 Integer 对象判等；

使用 == 对一个值为 127 的直接赋值的 Integer 和另一个通过 new Integer 声明的值为 127 的对象判等；

使用 == 对两个通过 new Integer 声明的值为 127 的对象判等；

使用 == 对一个值为 128 的直接赋值的 Integer 对象和另一个值为 128 的 int 基本类型判等。

 复制代码

```
1 Integer a = 127; //Integer.valueOf(127)
2 Integer b = 127; //Integer.valueOf(127)
3 log.info("\nInteger a = 127;\n" +
4         "Integer b = 127;\n" +
5         "a == b ? {}", a == b);    // true
6
7 Integer c = 128; //Integer.valueOf(128)
8 Integer d = 128; //Integer.valueOf(128)
9 log.info("\nInteger c = 128;\n" +
10         "Integer d = 128;\n" +
```

```

11         "c == d ? {}", c == d);    //false
12
13 Integer e = 127; //Integer.valueOf(127)
14 Integer f = new Integer(127); //new instance
15 log.info("\nInteger e = 127;\n" +
16         "Integer f = new Integer(127);\n" +
17         "e == f ? {}", e == f);    //false
18
19 Integer g = new Integer(127); //new instance
20 Integer h = new Integer(127); //new instance
21 log.info("\nInteger g = new Integer(127);\n" +
22         "Integer h = new Integer(127);\n" +
23         "g == h ? {}", g == h);    //false
24
25 Integer i = 128; //unbox
26 int j = 128;
27 log.info("\nInteger i = 128;\n" +
28         "int j = 128;\n" +
29         "i == j ? {}", i == j); //true

```

通过运行结果可以看到，虽然看起来永远是在对 127 和 127、128 和 128 判等，但 == 却没有永远给我们 true 的答复。原因是什么呢？

第一个案例中，编译器会把 Integer a = 127 转换为 Integer.valueOf(127)。查看源码可以发现，这个**转换在内部其实做了缓存，使得两个 Integer 指向同一个对象**，所以 == 返回 true。

 复制代码

```

1 public static Integer valueOf(int i) {
2     if (i >= IntegerCache.low && i <= IntegerCache.high)
3         return IntegerCache.cache[i + (-IntegerCache.low)];
4     return new Integer(i);
5 }

```

第二个案例中，之所以同样的代码 128 就返回 false 的原因是，默认情况下会缓存[-128, 127]的数值，而 128 处于这个区间之外。设置 JVM 参数加上 -XX:AutoBoxCacheMax=1000 再试试，是不是就返回 true 了呢？

 复制代码

```

1 private static class IntegerCache {
2     static final int low = -128;
3     static final int high;

```

```

4
5
6     static {
7         // high value may be configured by property
8         int h = 127;
9         String integerCacheHighPropValue =
10             sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
11         if (integerCacheHighPropValue != null) {
12             try {
13                 int i = parseInt(integerCacheHighPropValue);
14                 i = Math.max(i, 127);
15                 // Maximum array size is Integer.MAX_VALUE
16                 h = Math.min(i, Integer.MAX_VALUE - (-low) - 1);
17             } catch (NumberFormatException nfe) {
18                 // If the property cannot be parsed into an int, ignore it.
19             }
20         }
21         high = h;
22
23
24         cache = new Integer[(high - low) + 1];
25         int j = low;
26         for(int k = 0; k < cache.length; k++)
27             cache[k] = new Integer(j++);
28
29
30         // range [-128, 127] must be interned (JLS7 5.1.7)
31         assert IntegerCache.high >= 127;
32     }
33 }

```

第三和第四个案例中，New 出来的 Integer 始终是不走缓存的新对象。比较两个新对象，或者比较一个新对象和一个来自缓存的对象，结果肯定不是相同的对象，因此返回 false。

第五个案例中，我们把装箱的 Integer 和基本类型 int 比较，前者会先拆箱再比较，比较的肯定是数值而不是引用，因此返回 true。

看到这里，对于 Integer 什么时候是相同对象什么时候是不同对象，就很清楚了吧。但知道这些其实意义不大，因为在大多数时候，我们并不关心 Integer 对象是否是同一个，**只需要记得比较 Integer 的值请使用 equals，而不是 ==**（对于基本类型 int 的比较当然只能使用 ==）。

其实，我们应该都知道这个原则，只是有的时候特别容易忽略。以我之前遇到过的一个生产事故为例，有这么一个枚举定义了订单状态和对于状态的描述：

[复制代码](#)

```
1 enum StatusEnum {
2     CREATED(1000, "已创建"),
3     PAID(1001, "已支付"),
4     DELIVERED(1002, "已送到"),
5     FINISHED(1003, "已完成");
6
7     private final Integer status; //注意这里的Integer
8     private final String desc;
9
10    StatusEnum(Integer status, String desc) {
11        this.status = status;
12        this.desc = desc;
13    }
14 }
```

在业务代码中，开发同学使用了 == 对枚举和入参 OrderQuery 中的 status 属性进行判等：

[复制代码](#)

```
1 @Data
2 public class OrderQuery {
3     private Integer status;
4     private String name;
5 }
6
7 @PostMapping("enumcompare")
8 public void enumcompare(@RequestBody OrderQuery orderQuery){
9     StatusEnum statusEnum = StatusEnum.DELIVERED;
10    log.info("orderQuery:{} statusEnum:{} result:{}", orderQuery, statusEnum, :
11 }
```

**因为枚举和入参 OrderQuery 中的 status 都是包装类型，所以通过 == 判等肯定是有问题的。只是这个问题比较隐晦，究其原因在于：**

只看枚举的定义 CREATED(1000, “已创建” ), 容易让人误解 status 值是基本类型；

因为有 Integer 缓存机制的存在，所以使用 == 判等并不是所有情况下都有问题。在这次事故中，订单状态的值从 100 开始增长，程序一开始不出问题，直到订单状态超过 127 后才出现 Bug。


在了解清楚为什么 Integer 使用 == 判等有时候也有效的原因之后，我们再来看看为什么 String 也有这个问题。我们使用几个用例来测试下：

对两个直接声明的值都为 1 的 String 使用 == 判等；

对两个 new 出来的值都为 2 的 String 使用 == 判等；

对两个 new 出来的值都为 3 的 String 先进行 intern 操作，再使用 == 判等；

对两个 new 出来的值都为 4 的 String 通过 equals 判等。

 复制代码

```
1 String a = "1";
2 String b = "1";
3 log.info("\nString a = \"1\";\n" +
4         "String b = \"1\";\n" +
5         "a == b ? {}", a == b); //true
6
7 String c = new String("2");
8 String d = new String("2");
9 log.info("\nString c = new String(\"2\");\n" +
10        "String d = new String(\"2\");\n" +
11        "c == d ? {}", c == d); //false
12
13 String e = new String("3").intern();
14 String f = new String("3").intern();
15 log.info("\nString e = new String(\"3\").intern();\n" +
16        "String f = new String(\"3\").intern();\n" +
17        "e == f ? {}", e == f); //true
18
19 String g = new String("4");
20 String h = new String("4");
21 log.info("\nString g = new String(\"4\");\n" +
22        "String h = new String(\"4\");\n" +
23        "g == h ? {}", g.equals(h)); //true
```

在分析这个结果之前，我先和你说说 Java 的字符串常量池机制。首先要明确的是其设计初衷是节省内存。当代码中出现双引号形式创建字符串对象时，JVM 会先对这个字符串进行检查，如果字符串常量池中存在相同内容的字符串对象的引用，则将这个引用返回；否则，创建新的字符串对象，然后将这个引用放入字符串常量池，并返回该引用。这种机制，就是字符串驻留或池化。

再回到刚才的例子，再来分析一下运行结果：

第一个案例返回 true，因为 Java 的字符串驻留机制，直接使用双引号声明出来的两个 String 对象指向常量池中的相同字符串。

第二个案例，new 出来的两个 String 是不同对象，引用当然不同，所以得到 false 的结果。

第三个案例，使用 String 提供的 intern 方法也会走常量池机制，所以同样能得到 true。

第四个案例，通过 equals 对值内容判等，是正确的处理方式，当然会得到 true。

**虽然使用 new 声明的字符串调用 intern 方法，也可以让字符串进行驻留，但在业务代码中滥用 intern，可能会产生性能问题。**

写代码测试一下，通过循环把 1 到 1000 万之间的数字以字符串形式 intern 后，存入一个 List：

 复制代码

```
1 List<String> list = new ArrayList<>();
2
3 @GetMapping("internperformance")
4 public int internperformance(@RequestParam(value = "size", defaultValue = "10000000") int size) {
5     // -XX:+PrintStringTableStatistics
6     // -XX:StringTableSize=100000000
7     long begin = System.currentTimeMillis();
8     list = IntStream.rangeClosed(1, size)
9         .mapToObj(i -> String.valueOf(i).intern())
10        .collect(Collectors.toList());
11    log.info("size:{} took:{}", size, System.currentTimeMillis() - begin);
12    return list.size();
13 }
```

在启动程序时设置 JVM 参数 -XX:+PrintStringTableStatistic，程序退出时可以打印出字符串常量表的统计信息。调用接口后关闭程序，输出如下：

 复制代码

```
1 [11:01:57.770] [http-nio-45678-exec-2] [INFO] [.t.c.e.d.IntAndStringEqualCont
2 StringTable statistics:
3 Number of buckets      :      60013 =    480104 bytes, avg    8.000
4 Number of entries      :   10030230 =  240725520 bytes, avg   24.000
5 Number of literals     :   10030230 =  563005568 bytes, avg   56.131
6 Total footprint        :              =  804211192 bytes
```




```
7 Average bucket size      : 167.134
8 Variance of bucket size : 55.808
9 Std. dev. of bucket size: 7.471
10 Maximum bucket size     : 198
```

可以看到，1000 万次 intern 操作耗时居然超过了 44 秒。

其实，原因在于字符串常量池是一个固定容量的 Map。如果容量太小（Number of buckets=60013）、字符串太多（1000 万个字符串），那么每一个桶中的字符串数量会非常多，所以搜索起来就很慢。输出结果中的 Average bucket size=167，代表了 Map 中桶的平均长度是 167。

解决方式是，设置 JVM 参数 -XX:StringTableSize，指定更多的桶。设置 -XX:StringTableSize=10000000 后，重启应用：

 复制代码

```
1 [11:09:04.475] [http-nio-45678-exec-1] [INFO ] [.t.c.e.d.IntAndStringEqualCont
2 StringTable statistics:
3 Number of buckets      : 10000000 = 80000000 bytes, avg 8.000
4 Number of entries      : 10030156 = 240723744 bytes, avg 24.000
5 Number of literals     : 10030156 = 562999472 bytes, avg 56.131
6 Total footprint       :           = 883723216 bytes
7 Average bucket size    : 1.003
8 Variance of bucket size: 1.587
9 Std. dev. of bucket size: 1.260
10 Maximum bucket size   : 10
```

可以看到，1000 万次调用耗时只有 5.5 秒，Average bucket size 降到了 1，效果明显。

好了，是时候给出第二原则了：**没事别轻易用 intern，如果要用一定要注意控制驻留的字符串的数量，并留意常量表的各项指标。**

## 实现一个 equals 没有这么简单

如果看过 Object 类源码，你可能就知道，equals 的实现其实是比较对象引用：

 复制代码

```
1 public boolean equals(Object obj) {
```



```
2     return (this == obj);
3 }
```

之所以 Integer 或 String 能通过 equals 实现内容判等，是因为它们都重写了这个方法。比如，String 的 equals 的实现：

 复制代码

```
1 public boolean equals(Object anObject) {
2     if (this == anObject) {
3         return true;
4     }
5     if (anObject instanceof String) {
6         String anotherString = (String)anObject;
7         int n = value.length;
8         if (n == anotherString.value.length) {
9             char v1[] = value;
10            char v2[] = anotherString.value;
11            int i = 0;
12            while (n-- != 0) {
13                if (v1[i] != v2[i])
14                    return false;
15                i++;
16            }
17            return true;
18        }
19    }
20    return false;
21 }
```

对于自定义类型，如果不重写 equals 的话，默认就是使用 Object 基类的按引用的比较方式。我们写一个自定义类测试一下。


假设有这样一个描述点的类 Point，有 x、y 和描述三个属性：

 复制代码

```
1 class Point {
2     private int x;
3     private int y;
4     private final String desc;
5
6     public Point(int x, int y, String desc) {
7         this.x = x;
8         this.y = y;
```

```
9         this.desc = desc;
10     }
11 }
```

定义三个点 p1、p2 和 p3，其中 p1 和 p2 的描述属性不同，p1 和 p3 的三个属性完全相同，并写一段代码测试一下默认行为：

 复制代码

```
1 Point p1 = new Point(1, 2, "a");
2 Point p2 = new Point(1, 2, "b");
3 Point p3 = new Point(1, 2, "a");
4 log.info("p1.equals(p2) ? {}", p1.equals(p2));
5 log.info("p1.equals(p3) ? {}", p1.equals(p3));
```

通过 equals 方法比较 p1 和 p2、p1 和 p3 均得到 false，原因正如刚才所说，我们并没有为 Point 类实现自定义的 equals 方法，Object 超类中的 equals 默认使用 == 判等，比较的是对象的引用。

我们期望的逻辑是，只要 x 和 y 这 2 个属性一致就代表是同一个点，所以写出了如下的改进代码，重写 equals 方法，把参数中的 Object 转换为 Point 比较其 x 和 y 属性：

 复制代码


```
1 class PointWrong {
2     private int x;
3     private int y;
4     private final String desc;
5
6     public PointWrong(int x, int y, String desc) {
7         this.x = x;
8         this.y = y;
9         this.desc = desc;
10    }
11
12    @Override
13    public boolean equals(Object o) {
14        PointWrong that = (PointWrong) o;
15        return x == that.x && y == that.y;
16    }
17 }
```

为测试改进后的 Point 是否可以满足需求，我们定义了三个用例：

比较一个 Point 对象和 null；


比较一个 Object 对象和一个 Point 对象；

比较两个 x 和 y 属性值相同的 Point 对象。

 复制代码

```
1 PointWrong p1 = new PointWrong(1, 2, "a");
2 try {
3     log.info("p1.equals(null) ? {}", p1.equals(null));
4 } catch (Exception ex) {
5     log.error(ex.getMessage());
6 }
7
8 Object o = new Object();
9 try {
10    log.info("p1.equals(expression) ? {}", p1.equals(o));
11 } catch (Exception ex) {
12    log.error(ex.getMessage());
13 }
14
15 PointWrong p2 = new PointWrong(1, 2, "b");
16 log.info("p1.equals(p2) ? {}", p1.equals(p2));
```

通过日志中的结果可以看到，第一次比较出现了空指针异常，第二次比较出现了类型转换异常，第三次比较符合预期输出了 true。

 复制代码

```
1 [17:54:39.120] [http-nio-45678-exec-1] [ERROR] [t.c.e.demo1.EqualityMethodCont
2 [17:54:39.120] [http-nio-45678-exec-1] [ERROR] [t.c.e.demo1.EqualityMethodCont
3 [17:54:39.120] [http-nio-45678-exec-1] [INFO ] [t.c.e.demo1.EqualityMethodCont
```

**通过这些失效的用例，我们大概可以总结出实现一个更好的 equals 应该注意的点：**


考虑到性能，可以先进行指针判等，如果对象是同一个那么直接返回 true；

需要对另一方进行判空，空对象和自身进行比较，结果一定是 false；

需要判断两个对象的类型，如果类型都不同，那么直接返回 false；

确保类型相同的情况下再进行类型强制转换，然后逐一判断所有字段。

修复和改进后的 equals 方法如下：


 复制代码

```
1 @Override
2 public boolean equals(Object o) {
3     if (this == o) return true;
4     if (o == null || getClass() != o.getClass()) return false;
5     PointRight that = (PointRight) o;
6     return x == that.x && y == that.y;
7 }
```

改进后的 equals 看起来完美了，但还没完。我们继续往下看。

## hashCode 和 equals 要配对实现

我们来试试下面这个用例，定义两个 x 和 y 属性值完全一致的 Point 对象 p1 和 p2，把 p1 加入 HashSet，然后判断这个 Set 中是否存在 p2：

 复制代码

```
1 PointWrong p1 = new PointWrong(1, 2, "a");
2 PointWrong p2 = new PointWrong(1, 2, "b");
3
4 HashSet<PointWrong> points = new HashSet<>();
5 points.add(p1);
6 log.info("points.contains(p2) ? {}", points.contains(p2));
```

按照改进后的 equals 方法，这 2 个对象可以认为是同一个，Set 中已经存在了 p1 就应该包含 p2，但结果却是 false。

出现这个 Bug 的原因是，散列表需要使用 hashCode 来定位元素放到哪个桶。如果自定义对象没有实现自定义的 hashCode 方法，就会使用 Object 超类的默认实现，**得到的两个 hashCode 是不同的，导致无法满足需求。**

要自定义 hashCode，我们可以直接使用 Objects.hash 方法来实现，改进后的 Point 类如下：

[复制代码](#)

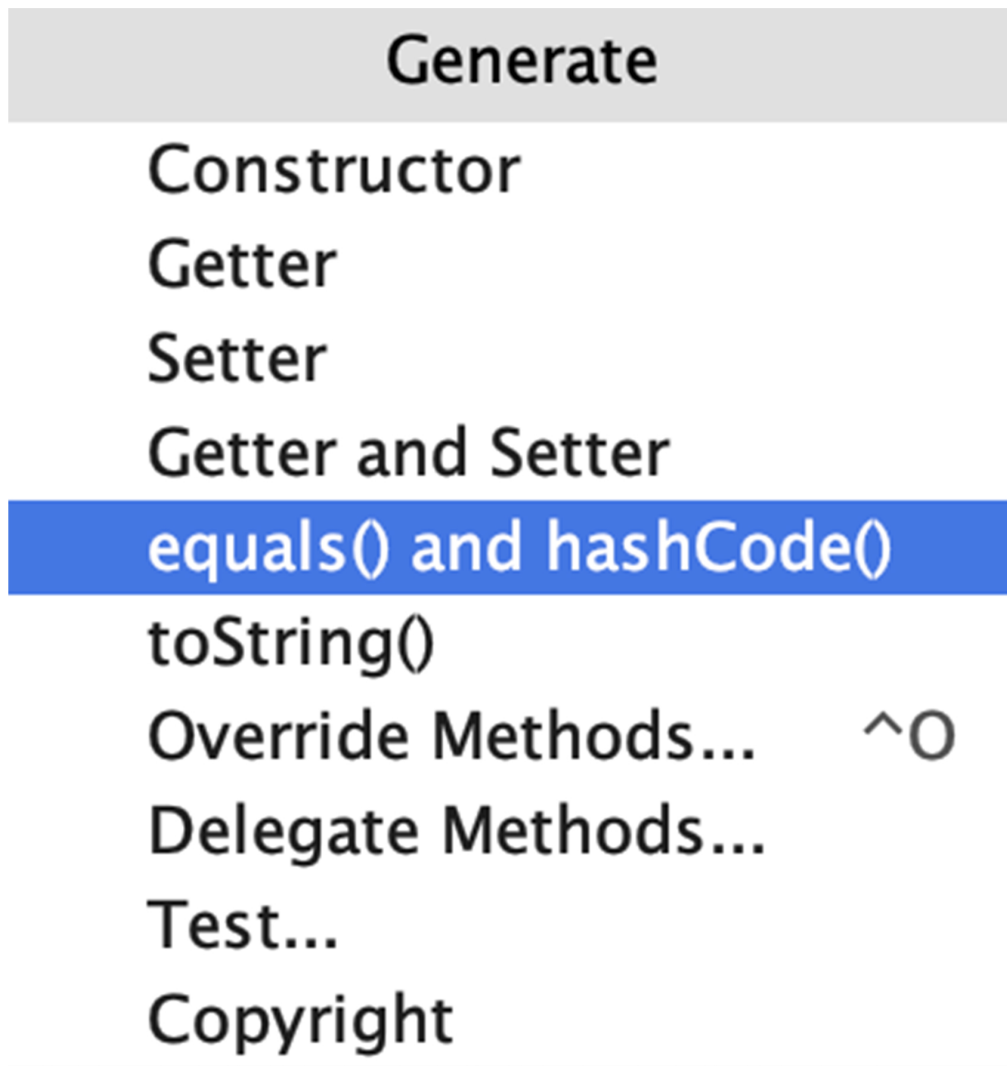
```
1 class PointRight {
2     private final int x;
3     private final int y;
4     private final String desc;
5     ...
6     @Override
7     public boolean equals(Object o) {
8         ...
9     }
10
11    @Override
12    public int hashCode() {
13        return Objects.hash(x, y);
14    }
15 }
```

改进 equals 和 hashCode 后，再测试下之前的四个用例，结果全部符合预期。

[复制代码](#)

```
1 [18:25:23.091] [http-nio-45678-exec-4] [INFO ] [t.c.e.demo1.EqualityMethodCont
2 [18:25:23.093] [http-nio-45678-exec-4] [INFO ] [t.c.e.demo1.EqualityMethodCont
3 [18:25:23.094] [http-nio-45678-exec-4] [INFO ] [t.c.e.demo1.EqualityMethodCont
4 [18:25:23.094] [http-nio-45678-exec-4] [INFO ] [t.c.e.demo1.EqualityMethodCont
```

看到这里，你可能会觉得自己实现 equals 和 hashCode 很麻烦，实现 equals 有很多注意点而且代码量很大。不过，实现这两个方法也有简单的方式，一是后面要讲到的 Lombok 方法，二是使用 IDE 的代码生成功能。IDEA 的类代码快捷生成菜单支持的功能如下：



## 注意 compareTo 和 equals 的逻辑一致性

除了自定义类型需要确保 equals 和 hashCode 要逻辑一致外，还有一个更容易被忽略的问题，即 compareTo 同样需要和 equals 确保逻辑一致性。

我之前遇到过这么一个问题，代码里本来使用了 ArrayList 的 indexOf 方法进行元素搜索，但是一位好心的开发同学觉得逐一比较的时间复杂度是  $O(n)$ ，效率太低了，于是改为了排序后通过 Collections.binarySearch 方法进行搜索，实现了  $O(\log n)$  的时间复杂度。没想到，这么一改却出现了 Bug。

我们来重现下这个问题。首先，定义一个 Student 类，有 id 和 name 两个属性，并实现了一个 Comparable 接口来返回两个 id 的值：

```

1  @Data
2  @AllArgsConstructor
3  class Student implements Comparable<Student>{
4      private int id;
5      private String name;
6
7      @Override
8      public int compareTo(Student other) {
9          int result = Integer.compare(other.id, id);
10         if (result==0)
11             log.info("this {} == other {}", this, other);
12         return result;
13     }
14 }

```

然后，写一段测试代码分别通过 `indexOf` 方法和 `Collections.binarySearch` 方法进行搜索。列表中我们存放了两个学生，第一个学生 `id` 是 1 叫 `zhang`，第二个学生 `id` 是 2 叫 `wang`，搜索这个列表是否存在一个 `id` 是 2 叫 `li` 的学生：

```

1  @GetMapping("wrong")
2  public void wrong(){
3
4      List<Student> list = new ArrayList<>();
5      list.add(new Student(1, "zhang"));
6      list.add(new Student(2, "wang"));
7      Student student = new Student(2, "li");
8
9      log.info("ArrayList.indexOf");
10     int index1 = list.indexOf(student);
11     Collections.sort(list);
12     log.info("Collections.binarySearch");
13     int index2 = Collections.binarySearch(list, student);
14
15     log.info("index1 = " + index1);
16     log.info("index2 = " + index2);
17 }

```

代码输出的日志如下：

```

1  [18:46:50.226] [http-nio-45678-exec-1] [INFO ] [t.c.equals.demo2.CompareToCont
2  [18:46:50.226] [http-nio-45678-exec-1] [INFO ] [t.c.equals.demo2.CompareToCont

```



```
3 [18:46:50.227] [http-nio-45678-exec-1] [INFO ] [t.c.equals.demo2.CompareToCont
4 [18:46:50.227] [http-nio-45678-exec-1] [INFO ] [t.c.equals.demo2.CompareToCont
5 [18:46:50.227] [http-nio-45678-exec-1] [INFO ] [t.c.equals.demo2.CompareToCont
```

我们注意到如下几点：

**binarySearch 方法内部调用了元素的 compareTo 方法进行比较；**

indexOf 的结果没问题，列表中搜索不到 id 为 2、name 是 li 的学生；

binarySearch 返回了索引 1，代表搜索到的结果是 id 为 2，name 是 wang 的学生。

修复方式很简单，确保 compareTo 的比较逻辑和 equals 的实现一致即可。重新实现一下 Student 类，通过 Comparator.comparing 这个便捷的方法来实现两个字段的比较：

 复制代码

```
1 @Data
2 @AllArgsConstructor
3 class StudentRight implements Comparable<StudentRight>{
4     private int id;
5     private String name;
6
7     @Override
8     public int compareTo(StudentRight other) {
9         return Comparator.comparing(StudentRight::getName)
10             .thenComparingInt(StudentRight::getId)
11             .compare(this, other);
12     }
13 }
```

其实，这个问题容易被忽略的原因在于两方面：

一是，我们使用了 Lombok 的 @Data 标记了 Student，@Data 注解（详见[这里](#)）其实包含了 @EqualsAndHashCode 注解（详见[这里](#)）的作用，也就是默认情况下使用类型所有的字段（不包括 static 和 transient 字段）参与到 equals 和 hashCode 方法的实现中。因为这两个方法的实现不是我们自己实现的，所以容易忽略其逻辑。

二是，compareTo 方法需要返回数值，作为排序的依据，容易让人使用数值类型的字段随意实现。


我再强调下，对于自定义的类型，如果要实现 Comparable，请记得 equals、hashCode、compareTo 三者逻辑一致。

## 小心 Lombok 生成代码的“坑”

Lombok 的 @Data 注解会帮我们实现 equals 和 hashCode 方法，但是有继承关系时，Lombok 自动生成的方法可能就不是我们期望的了。

我们先来研究一下其实现：定义一个 Person 类型，包含姓名和身份证两个字段：

```
1 @Data
2 class Person {
3     private String name;
4     private String identity;
5
6     public Person(String name, String identity) {
7         this.name = name;
8         this.identity = identity;
9     }
10 }
```

 复制代码

对于身份证相同、姓名不同的两个 Person 对象：

```
1 Person person1 = new Person("zhuye", "001");
2 Person person2 = new Person("Joseph", "001");
3 log.info("person1.equals(person2) ? {}", person1.equals(person2));
```


 复制代码

使用 equals 判等会得到 false。如果你希望只要身份证一致就认为是同一个人的话，可以使用 @EqualsAndHashCode.Exclude 注解来修饰 name 字段，从 equals 和 hashCode 的实现中排除 name 字段：

```
1 @EqualsAndHashCode.Exclude
2 private String name;
```

 复制代码

修改后得到 true。打开编译后的代码可以看到，Lombok 为 Person 生成的 equals 方法的实现，确实只包含了 identity 属性：

 复制代码


```
1 public boolean equals(final Object o) {
2     if (o == this) {
3         return true;
4     } else if (!(o instanceof LombokEqualsController.Person)) {
5         return false;
6     } else {
7         LombokEqualsController.Person other = (LombokEqualsController.Person)
8         if (!other.canEqual(this)) {
9             return false;
10        } else {
11            Object this$identity = this.getIdentity();
12            Object other$identity = other.getIdentity();
13            if (this$identity == null) {
14                if (other$identity != null) {
15                    return false;
16                }
17            } else if (!this$identity.equals(other$identity)) {
18                return false;
19            }
20
21            return true;
22        }
23    }
24 }
```

但到这里还没完，如果类型之间有继承，Lombok 会怎么处理子类的 equals 和 hashCode 呢？我们来测试一下，写一个 Employee 类继承 Person，并新定义一个公司属性：

 复制代码

```
1 @Data
2 class Employee extends Person {
3
4     private String company;
5     public Employee(String name, String identity, String company) {
6         super(name, identity);
7         this.company = company;
8     }
9 }
```


在如下的测试代码中，声明两个 Employee 实例，它们具有相同的公司名称，但姓名和身份证均不同：

 复制代码

```
1 Employee employee1 = new Employee("zhuye", "001", "bkjk.com");
2 Employee employee2 = new Employee("Joseph", "002", "bkjk.com");
3 log.info("employee1.equals(employee2) ? {}", employee1.equals(employee2));
```

很遗憾，结果是 true，显然是没有考虑父类的属性，而认为这两个员工是同一人，**说明 @EqualsAndHashCode 默认实现没有使用父类属性。**

为解决这个问题，我们可以手动设置 callSuper 开关为 true，来覆盖这种默认行为：

 复制代码

```
1 @Data
2 @EqualsAndHashCode(callSuper = true)
3 class Employee extends Person {
```

修改后的代码，实现了同时以子类的属性 company 加上父类中的属性 identity，作为 equals 和 hashCode 方法的实现条件（实现上其实是调用了父类的 equals 和 hashCode）。

## 重点回顾

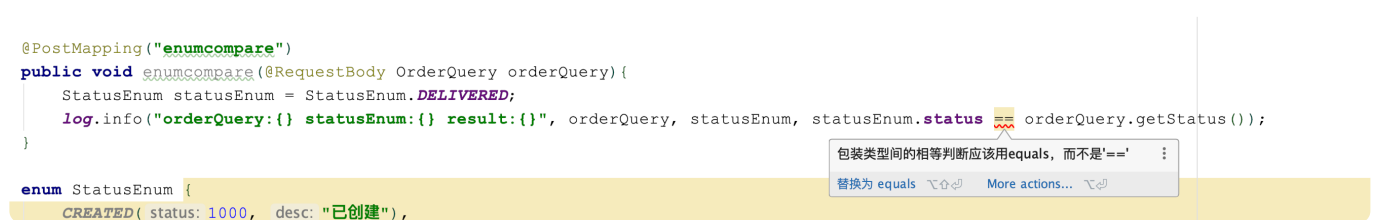
现在，我们来回顾下对象判等和比较的重点内容吧。

首先，我们要注意 equals 和 == 的区别。业务代码中进行内容的比较，针对基本类型只能使用 ==，针对 Integer、String 在内的引用类型，需要使用 equals。Integer 和 String 的坑在于，使用 == 判等有时也能获得正确结果。

其次，对于自定义类型，如果类型需要参与判等，那么务必同时实现 equals 和 hashCode 方法，并确保逻辑一致。如果希望快速实现 equals、hashCode 方法，我们可以借助 IDE 的代码生成功能，或使用 Lombok 来生成。如果类型也要参与比较，那么 compareTo 方法的逻辑同样需要和 equals、hashCode 方法一致。

最后，Lombok 的 `@EqualsAndHashCode` 注解实现 `equals` 和 `hashCode` 的时候，默认使用类型所有非 `static`、非 `transient` 的字段，且不考虑父类。如果希望改变这种默认行为，可以使用 `@EqualsAndHashCode.Exclude` 排除一些字段，并设置 `callSuper = true` 来让子类的 `equals` 和 `hashCode` 调用父类的相应方法。

在比较枚举值和 POJO 参数值的例子中，我们还可以注意到，使用 `==` 来判断两个包装类型的低级错误，确实容易被忽略。所以，**我建议你在 IDE 中安装阿里巴巴的 Java 规约插件**（详见 [这里](#)），来及时提示我们这类低级错误：



今天用到的代码，我都放在了 GitHub 上，你可以点击 [这个链接](#) 查看。

## 思考与讨论

1. 在实现 `equals` 时，我是先通过 `getClass` 方法判断两个对象的类型，你可能会想到还可以使用 `instanceof` 来判断。你能说说这两种实现方式的区别吗？
2. 在第三节的例子中，我演示了可以通过 `HashSet` 的 `contains` 方法判断元素是否在 `HashSet` 中，同样是 `Set` 的 `TreeSet` 其 `contains` 方法和 `HashSet` 有什么区别吗？

有关对象判等、比较，你还遇到过其他坑吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

## 进入朱晔老师「读者群」带你攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | 数据库索引：索引并不是万能药

下一篇 09 | 数值计算：注意精度、舍入和溢出问题


### 精选留言 (15)




 写留言



Darren

2020-03-26

稍微补充一点，可能因为篇幅的原因，老师没提到，关于equals其实还有一个大坑，equals比较的对象除了所谓的相等外，还有一个非常重要的因素，就是该对象的类加载器也必须是同一个，不然equals返回的肯定是false；之前遇到过一个坑：重启后，两个对象相等，结果是true，但是修改了某些东西后，热加载（不用重启即可生效）后，再次执行equals，返回就是false，因为热加载使用的类加载器和程序正常启动的类加载器不同。关于类...  
展开 

作者回复:   

这位同学作为本课课代表 

 13

 36



2020-03-26

2. HashSet 底层是HashMap。TreeSet底层是TreeMap

HashSet就是使用HashMap调用equals，判断两对象的HashCode是否相等。

TreeSet因为是一个树形结构，则需要考虑数的左右。则需要通过compareTo计算正负值，看最后能否找到compareTo为0的值，找到则返回true。

...

展开

作者回复: ㊗



5



Sun

2020-03-26

老师的课程，真的是干货，每天凌晨更新完看一遍，早上上班前在看一遍，感受都不一样，期待出更多干货，共同进步

展开

作者回复: 设计篇和安全篇还会有更丰富的内容，紧跟脚步，细细品味



3



Z\_CHP

2020-03-26

老师的每一篇文章都是满满的干货呀，手动点赞

作者回复: 觉得好可以多转发分享



3



2020-03-26

问题1:

```
Father father = new Father();
```

```
Son son = new Son();
```

```
System.out.println(son.getClass() == father.getClass());
```

```
System.out.println(son instanceof Father);...
```

展开





3

**东方奇骥**

2020-03-26

看到这节，说起Lombok，老师觉得Lombok 适合用于生产环境吗？之前一直都是自己业余练习使用，但是工作中项目都还是没有使用。

展开 ∨

作者回复: 只要你理解它各种注解会生成怎样的代码，就没问题



3

**Huodefa\_0426**

2020-03-26

老师，文中你说：在启动程序时设置 JVM 参数 `-XX:+PrintStringTableStatistics`，程序退出时可以打印出字符串常量表的统计信息。调用接口后关闭程序，输出如下。我设置了关闭程序怎么没看见输出的信息，是输出在控制台还是在日志文件中？如果是文件 是哪个文件？

展开 ∨

作者回复: 控制台，确保参数生效了



2

**pedro**

2020-03-26

1楼的回答已经趋于完美，我也翻了一下 JDK 源码，HashSet 的本质是 HashMap，会通过 hash 函数来比较值，TreeSet 的本质是 TreeMap 会通过 compareTo 比较。至于类加载器的问题，我想这个不好复现，有没有楼下的小伙伴补充一下的。

展开 ∨

作者回复: 🙏



2

**失火的夏天**

2020-03-28

hashset和treeSet从根本上来说，没什么关系，只是有个N代以前的祖宗了，哈哈，一个玩hash，一个玩comparator。一个底层是散列表，一个底层是红黑树。

展开 ▾



1



yihang

2020-03-28

另外对于 intern 也有它的用武之处，据说 twitter 使用它减少重复地址（字符串）大量节约了内存



1



yihang

2020-03-28

补充一点，浮点数的 == 比较也有坑，跟浮点数小数精度有关

作者回复: 是的，这个下篇提到了



1



EchosEcho

2020-03-29

getClass需要具体一种类型才能做比较，instanceof可以在子类 and 父类间实现equals方法



努力奋斗的Pisces

2020-03-29

1. instanceof 涉及到继承的子类是都属于父类的判断
2. TreeSet 是 TreeMap 的实现，使用了 compareTo 来判断是否包含

展开 ▾



2020-03-27

`Collections.sort(list);`

也调用了 compareTo 吧，所以返回下标 index2 是不是应该等于 0?

展开 ▾



Geek\_3b1096

2020-03-27

上班前看一遍 +1

展开 ✓

