

【软件构造】第三章第五节 **ADT**和**OOP**中的等价性

第三章第五节 **ADT**和**OOP**中的等价性

在很多场景下，需要判定两个对象是否“相等”，例如：判断某个Collection 中是否包含特定元素。

==和equals()有和区别？如何为自定义 ADT正确实现equals()？

OutLine

- 等价性equals() 和 ==
- equals()的判断方法
 - 自反、传递、对称性
- hashCode()
- 不可变类型的等价性
- 可变类型的等价性
 - 观察等价性
 - 行为等价性

Notes

等价性equals() 和 ==

- 和很多其他语言一样，Java有两种判断相等的操作—— == 和 equals () 。
- ==是引用等价性；而equals () 是对象等价性。
 - == 比较的是索引。更准确的说，它测试的是指向相等（referential equality）。如果两个索引指向同一块存储区域，那它们就是==的。对于我们之前提到过的快照图来说，==就意味着它们的箭头指向同一个对象。
 - equals () 操作比较的是对象的内容，换句话说，它测试的是对象值相等（object equality）。e在每一个ADT中，quals操作必须合理定义。

Java中的数据类型，可分为两类：

- 基本数据类型，也称原始数据类型。byte,short,char,int,long,float,double,boolean
 - 他们之间的比较，应用双等号（==）,比较的是他们的值。
- 复合数据类型(类)
 - 当他们用（==）进行比较的时候，比较的是他们在内存中的存放地址，所以，除非是同一个new出来的对象，他们的比较后的结果为true，否则比较后结果为false。
 - JAVA当中所有的类都是继承于Object这个基类的，在Object中的基类中定义了一个equals的方法，这个方法的初始行为是比较对象的内存地址，但在一些类库当中这个方法被覆盖掉了，如String,Integer,Date在这些类当中equals有其自身的实现，而不再是比较类在堆内存中的存放地址了。
 - 对于复合数据类型之间进行equals比较，在没有覆写equals方法的情况下，他们之间的比较还是基于他们在内存中的存放位置的地址值的，因为Object的equals方法也是用双等号（==）进行比较的，所以比较后的结果跟双等号（==）的结果相同。

[关于equals\(\)与== 欢迎阅读 海子的博客](#)

equals()的判断方法

严格来说，我们可以从三个角度定义相等：

- 抽象函数：回忆一下抽象函数（ $AF: R \rightarrow A$ ），它将具体的表示数据映射到了抽象的值。如果 $AF(a)=AF(b)$ ，我们就说a和b相等。
- 等价关系：等价是指对于关系 $E \subseteq T \times T$ ，它满足：
 - 自反性: $x.equals(x)$ 必须返回true
 - 对称性: $x.equals(y)$ 与 $y.equals(x)$ 的返回值必须相等。
 - 传递性: $x.equals(y)$ 为true， $y.equals(z)$ 也为true，那么 $x.equals(z)$ 必须为true。

以上两种角度/定义实际上是一样的，通过等价关系我们可以构建一个抽象函数（译者注：就是一个封闭的二元关系运算）；而抽象函数也能推出一个等价关系。

- 从使用者/外部的角度去观察：我们说两个对象相等，当且仅当使用者无法观察到它们之间有不同，即每一个观察总会都会得到相同的结果。例如对于两个集合对象 $\{1,2\}$ 和 $\{2,1\}$ ，我们就无法观察到不同：
 - $|\{1,2\}| = 2, |\{2,1\}| = 2$
 - $1 \in \{1,2\}$ is true, $1 \in \{2,1\}$ is true
 - $2 \in \{1,2\}$ is true, $2 \in \{2,1\}$ is true
 - $3 \in \{1,2\}$ is false, $3 \in \{2,1\}$ is false
 - ...

hashCode()方法

- 对于不可变类型：
 - equals() 应该比较抽象值是否相等。这和 equals() 比较行为相等性是一样的。
 - hashCode() 应该将抽象值映射为整数。
 - 所以不可变类型应该同时覆盖 equals() 和 hashCode()。
- 对于可变类型：
 - equals() 应该比较索引，就像 == 一样。同样的，这也是比较行为相等性。
 - hashCode() 应该将索引映射为整数。
 - 所以可变类型不应该将 equals() 和 hashCode() 覆盖，而是直接继承 Object 中的方法。Java 没有为大多数聚合类遵守这一规定，这也许会导致上面看到的隐秘bug。
- equals与hashCode两个方法均属于Object对象，equals根据我们的需要重写，用来判断是否是同一个内容或同一个对象，具体是判断什么，怎么判断得看怎么重写，默认的equals是比较地址。
- hashCode方法返回一个int的哈希码，同样可以重写来自定义获取哈希码的方法。
- equals判定为相同，hashCode一定相同。equals判定为不同，hashCode不一定不同。
- hashCode必须为两个被该equals方法视为相等的对象产生相同的结果。
- 与equals()方法类似，hashCode()方法可以被重写。JDK中对hashCode()方法的作用，以及实现时的注意事项做了说明：
 - hashCode()在哈希表中起作用，如java.util.HashMap。
 - 如果对象在equals()中使用的信息都没有改变，那么hashCode()值始终不变。
 - 如果两个对象使用equals()方法判断为相等，则hashCode()方法也应该相等。
 - 如果两个对象使用equals()方法判断为不相等，则不要求hashCode()也必须不相等；但是开发人员应该认识到，不相等的对象产生不相同的hashCode可以提高哈希表的性能。

不可变类型的等价性

首先来看Object中实现的缺省equals()：



```
public class Object {  
    ...  
    public boolean equals(Object that) {  
        return this == that;  
    }  
}
```



在Object中实现的缺省equals() 是在判断引用等价性。这通常不是程序员所期望的，因此需要重写，下面是一个栗子：



```
public class Duration {  
    ...  
    // Problematic definition of equals()  
    public boolean equals(Duration that) {  
        return this.getLength() == that.getLength();  
    }  
}
```




尝试如下客户端代码，可得到

```
Duration d1 = new Duration (1, 2);  
Duration d2 = new Duration (1, 2);  
Object o2 = d2;  
d1.equals(d2) → true  
d1.equals(o2) → false
```

基于以上结果进行以下解释：

- 即使d2和o2最终参照相同的对象在内存中，对他们来说你仍然得到不同的结果。
- 事实证明，该方法Duration已经超载equals()，因为方法签名与Object's 不相同。我们实际上有两种**equals()**方法：隐式equals(Object)继承Object，和新的equals(Duration)。
- 如果我们通过一个Object参考，那么d1.equals(o2)我们最终会调用equals(Object)实现。
- 如果我们通过Duration参考，如在d1.equals(d2)，我们最终调用equals(Duration)版本。
- 即使发生这种情况o2，d2两者都会在运行时指向同一个对象！平等已经变得不一致。

我们需要注释 @Override，重写超类中的方法，因此，这里实施正确的 equals() 方法：



```
@Override
```

```
public boolean equals (Object thatObject) {  
    if (!(thatObject instanceof Duration)) return false;  
    Duration thatDuration = (Duration) thatObject;  
    return this.getLength() == thatDuration.getLength();  
}
```



再次执行客户端代码，可得到：

```
Duration d1 = new Duration(1, 2);  
Duration d2 = new Duration(1, 2);  
Object o2 = d2;  
d1.equals(d2) → true  
d1.equals(o2) → true
```

可变类型的等价性

回忆之前我们对于相等的定义，即它们不能被使用者观察出来不同。而对于可变对象来说，它们多了一种新的可能：通过在观察前调用改造者，我们可以改变其内部的状态，从而观察出不同的结果。

- 所以我们重新定义两种相等：
 - 观察等价性：两个索引在不改变各自对象状态的前提下不能被区分。即通过只调用`observer`，`producer`和`creator`的方法，它测试的是这两个索引在当前程序状态下“看起来”相等。
 - 行为等价性：两个索引在任何代码的情况下都不能被区分，即使有一个对象调用了改造者。它测试的是两个对象是否会在未来所有的状态下“行为”相等。
- 对于不可变对象，观察相等和行为相等是完全等价的，因为它们没有改造者改变对象内部的状态。
- 对于可变对象，**Java**通常实现的是观察相等。例如两个不同的 `List` 对象包含相同的序列元素，那么`equals()` 操作就会返回真。

在有些时候，观察等价性可能导致bug，甚至可能破坏RI。

假设我们做了一个`List`，然后把它放到`Set`：

```
List<String> list = new ArrayList<>();  
list.add("a");  
  
Set<List<String>> set = new HashSet<List<String>>();  
set.add(list);
```

我们可以检查该集合是否包含我们放入其中的列表，并且它会：

```
set.contains(list) → true
```

但是如果修改这个存入的列表：

```
list.add("goodbye");
```

它似乎就不在集合中了！

```
set.contains(list) → false!
```

事实上，更糟糕的是：当我们（用迭代器）循环遍历这个集合时，我们依然会发现集合存在，但是contains() 还是说它不存在！

```
for (List<String> l : set) {
    set.contains(l) → false!
}
```

如果一个集合的迭代器和contains() 都互相冲突的时候，显然这个集合已经被破坏了。

发生了什么？我们知道 List<String> 是一个可变对象，而在Java对可变对象的实现中，改造操作通常都会影响 equals() 和 hashCode() 的结果。所以列表第一次放入 HashSet 的时候，它是存储在这时 hashCode() 对应的索引位置。但是后来列表发生了改变，计算 hashCode() 会得到不一样的结果，但是 HashSet 对此并不知道，所以我们调用contains 时候就会找不到列表。

当 equals() 和 hashCode() 被改动影响的时候，我们就破坏了哈希表利用对象作为键的不变量。

下面是 java.util.Set 规格说明中的一段话：

1 注意：当可变对象作为集合的元素时要特别小心。如果对象内容改变后会影响相等比较而且对象是集合的元素，那么集合的行为是不确定。

我们应该从这个例子中吸取教训，对可变类型，实现行为等价性即可，也就是说，只有指向同样内存空间的objects，才是相等的。所以对可变类型来说，无需重写这两个函数，直接继承 Object对象的两个方法即可。如果一定要判断两个可变对象看起来是否一致，最好定义一个新的方法。