

Java 基础提升篇：深入剖析 Java 中的装箱和拆箱

一、什么是装箱？什么是拆箱？

我们知道 Java 为每种基本数据类型都提供了对应的包装器类型，至于为什么会为每种基本数据类型提供包装器类型在此不进行阐述，有兴趣的朋友可以查阅相关资料。在 Java SE5 之前，如果要生成一个数值为 10 的 Integer 对象，必须这样进行：

```
Integer i = new Integer(10);
```

而在从 Java SE5 开始就提供了自动装箱的特性，如果要生成一个数值为 10 的 Integer 对象，只需要这样就可以了：

```
Integer i = 10;
```

这个过程中会自动根据数值创建对应的 Integer 对象，这就是装箱。

那什么是拆箱呢？顾名思义，跟装箱对应，就是自动将包装器类型转换为基本数据类型：

```
Integer i = 10; //装箱  
int n = i; //拆箱
```

简单一点说，**装箱就是自动将基本数据类型转换为包装器类型；拆箱就是自动将包装器类型转换为基本数据类型。**

下表是基本数据类型对应的包装器类型：

int (4 字节)	Integer
byte (1 字节)	Byte
short (2 字节)	Short
long (8 字节)	Long
float (4 字节)	Float
double (8 字节)	Double
char (2 字节)	Character

boolean (未定)	Boolean
--------------	---------


二、装箱和拆箱是如何实现的

上一小节了解装箱的基本概念之后，这一小节来了解一下装箱和拆箱是如何实现的。我们就以 Integer 类为例，下面看一段代码：

```
public class Main {  
    public static void main(String[] args) {  
        Integer i = 10;  
        int n = i;  
    }  
}
```

反编译 class 文件之后得到如下内容：

```
E:\Workspace\Test\bin\com\cxh\test1>javap -c Main  
Compiled from "Main.java"  
public class com.cxh.test1.Main extends java.lang.Object{  
    public com.cxh.test1.Main();  
    Code:  
    0:   aload_0  
    1:   invokespecial   #8; //Method java/lang/Object."<init>":()V  
    4:   return  
  
    public static void main(java.lang.String[]);  
    Code:  
    0:   bipush 10  
    2:   invokestatic    #16; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;  
    5:   astore_1  
    6:   aload_1  
    7:   invokevirtual   #22; //Method java/lang/Integer.intValue:()I  
   10:   istore_2  
   11:   return  
}
```



从反编译得到的字节码内容可以看出，在装箱的时候自动调用的是 Integer 的 valueOf(int)方法。而在拆箱的时候自动调用的是 Integer 的 intValue 方法。

其他的也类似，比如 Double、Character，不相信的朋友可以自己手动尝试一下。因此可以用一句话总结装箱和拆箱的实现过程：

装箱过程是通过调用包装器的 valueOf 方法实现的，而拆箱过程是通过调用包装器的 xxxValue 方法实现的。（xxx 代表对应的基本数据类型）。

三、面试中相关的问题

虽然大多数人对装箱和拆箱的概念都清楚，但是在面试和笔试中遇到了与装箱和拆箱的问题却不一定会答得上来。下面列举一些常见的与装箱/拆箱有关的面试题。

1. 下面这段代码的输出结果是什么？

```
public class Main {  
    public static void main(String[] args) {  
        Integer i1 = 100;  
        Integer i2 = 100;  
        Integer i3 = 200;  
        Integer i4 = 200;  
        System.out.println(i1==i2);  
        System.out.println(i3==i4);  
    }  
}
```

也许有些朋友会说都会输出 false，或者也有朋友会说都会输出 true。但是事实上输出结果是：

```
true  
false
```

为什么会出现这样的结果？输出结果表明 i1 和 i2 指向的是同一个对象，而 i3 和 i4 指向的是不同的对象。此时只需一看源码便知究竟，下面这段代码是 Integer 的 valueOf 方法的具体实现：

```
public static Integer valueOf(int i) {  
    if(i >= -128 && i <= IntegerCache.high)  
        return IntegerCache.cache[i + 128];  
    else  
        return new Integer(i);  
}
```

而其中 IntegerCache 类的实现为：

```
private static class IntegerCache {  
    static final int high;  
    static final Integer cache[];  
    static {  
        final int low = -128;
```

```
// high value may be configured by property
int h = 127;
if (integerCacheHighPropValue != null) {
    // Use Long.decode here to avoid invoking methods that
    // require Integer's autoboxing cache to be initialized
    int i = Long.decode(integerCacheHighPropValue).intValue();
    i = Math.max(i, 127);
    // Maximum array size is Integer.MAX_VALUE
    h = Math.min(i, Integer.MAX_VALUE - -low);
}
high = h;
cache = new Integer[(high - low) + 1];
int j = low;
for(int k = 0; k < cache.length; k++)
    cache[k] = new Integer(j++);
}
private IntegerCache() {}
}
```

从这 2 段代码可以看出，在通过 `valueOf` 方法创建 `Integer` 对象的时候，如果数值在 `[-128,127]` 之间，便返回指向 `IntegerCache.cache` 中已经存在的对象的引用；否则创建一个新的 `Integer` 对象。

上面的代码中 `i1` 和 `i2` 的数值为 100，因此会直接从 `cache` 中取已经存在的对象，所以 `i1` 和 `i2` 指向的是同一个对象，而 `i3` 和 `i4` 则是分别指向不同的对象。

2. 下面这段代码的输出结果是什么？

```
public class Main {
    public static void main(String[] args) {
        Double i1 = 100.0;
        Double i2 = 100.0;
        Double i3 = 200.0;
        Double i4 = 200.0;
        System.out.println(i1==i2);
        System.out.println(i3==i4);
    }
}
```

也许有的朋友会认为跟上面一道题目的输出结果相同，但是事实上却不是。实际输出结果为：

```
false  
false
```

至于具体为什么，读者可以去查看 Double 类的 valueOf 的实现。

```
public static Double valueOf(double d) {  
    return new Double(d);  
}
```

在这里只解释一下为什么 Double 类的 valueOf 方法会采用与 Integer 类的 valueOf 方法不同的实现。很简单：在某个范围内的整型数值的个数是有限的，而浮点数却不是。

注意，Integer、Short、Byte、Character、Long 这几个类的 valueOf 方法的实现是类似的。

Double、Float 的 valueOf 方法的实现是类似的。

3. 下面这段代码输出结果是什么：

```
public class Main {  
    public static void main(String[] args) {  
        Boolean i1 = false;  
        Boolean i2 = false;  
        Boolean i3 = true;  
        Boolean i4 = true;  
        System.out.println(i1==i2);  
        System.out.println(i3==i4);  
    }  
}
```

输出结果是：

```
true  
true
```

至于为什么是这个结果，同样地，看了 Boolean 类的源码也会一目了然。下面是 Boolean 的 valueOf 方法的具体实现：

```
public static Boolean valueOf(boolean b) {  
    return (b ? TRUE : FALSE);  
}
```

而其中的 TRUE 和 FALSE 又是什么呢？在 Boolean 中定义了 2 个静态成员属性：

```
public static final Boolean TRUE = new Boolean(true);  
  
/**  
 * The Boolean object corresponding to the primitive
```

```
* value <code>false</code>.  
*/  
  
public static final Boolean FALSE = new Boolean(false);
```

至此，大家应该明白了为何上面输出的结果都是 true 了。

4. 谈谈 Integer i = new Integer(xxx)和 Integer i =xxx; 这两种方式的区别。

当然，这个题目属于比较宽泛类型的。但是要点一定要答上，我总结一下主要有以下这两点区别：

- 1) 第一种方式不会触发自动装箱的过程；而第二种方式会触发；
- 2) 在执行效率和资源占用上的区别。第二种方式的执行效率和资源占用在一般性情况下要优于第一种情况（注意这并不是绝对的）。

5. 下面程序的输出结果是什么？

```
public class Main {  
    public static void main(String[] args) {  
        Integer a = 1;  
        Integer b = 2;  
        Integer c = 3;  
        Integer d = 3;  
        Integer e = 321;  
        Integer f = 321;  
        Long g = 3L;  
        Long h = 2L;  
        System.out.println(c==d);  
        System.out.println(e==f);  
        System.out.println(c==(a+b));  
        System.out.println(c.equals(a+b));  
        System.out.println(g==(a+b));  
        System.out.println(g.equals(a+b));  
        System.out.println(g.equals(a+h));  
    }  
}
```

先别看输出结果，读者自己想一下这段代码的输出结果是什么。这里面需要注意的是：

当 “==” 运算符的两个操作数都是包装器类型的引用，则是比较指向的是否是同一个对象，而如果其中有一个操作数是表达式（即包含算术运算）则比较的是数值（即会触发自动拆箱的过程）。另外，对于包装器类型，equals 方法并不会进行类型转换。明白了这 2 点之后，上面的输出结果便一目了然：

```
true
false
true
true
true
false
true
```

第一个和第二个输出结果没有什么疑问。第三句由于 `a+b` 包含了算术运算，因此会触发自动拆箱过程（会调用 `intValue` 方法），因此它们比较的是数值是否相等。而对于 `c.equals(a+b)` 会先触发自动拆箱过程，再触发自动装箱过程，也就是说 `a+b`，会先各自调用 `intValue` 方法，得到了加法运算后的数值之后，便调用 `Integer.valueOf` 方法，再进行 `equals` 比较。同理对于后面的也是这样，不过要注意倒数第二个和最后一个输出的结果（如果数值是 `int` 类型的，装箱过程调用的是 `Integer.valueOf`；如果是 `long` 类型的，装箱调用的 `Long.valueOf` 方法）。