

本文由 [简悦 SimpRead](#) 转码，原文地址 [www.imooc.com](http://www.imooc.com)

《手册》第 7 页有一段关于包装对象之间值的比较问题的规约：

【强制】所有整型包装类对象之间值的比较，全部使用 equals 方法比较。

说明：对于 Integer var = ? 在 -128 至 127 范围内的赋值，Integer 对象是在 IntegerCache.cache 产生，会复用已有对象，这个区间内的 Integer 值可以直接使用 == 进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals 方法进行判断。

这条建议非常值得大家关注，而且该问题在 Java 面试中十分常见。

我们还需要思考以下几个问题：

- 如果不看《手册》，我们如何知道 Integer var = ? 会缓存 -128 到 127 之间的赋值？
- 为什么会缓存这个范围的赋值？
- 我们如何学习和分析类似的问题？

我们先看下面的示例代码，并思考该段代码的输出结果：

```
public class IntTest {  
    public static void main(String[] args) {  
        Integer a = 100, b = 100, c = 150, d = 150;  
        System.out.println(a == b);  
        System.out.println(c == d);  
    }  
}
```

通过运行代码可以得到答案，程序输出的结果分别为：true, false。

那么为什么答案是这样？

结合《手册》的描述很多人可能会颇有自信地回答：因为缓存了 -128 到 127 之间的数值，就没有然后了。

那么为什么会缓存这一段区间的数值？缓存的区间可以修改吗？其它的包装类型有没有类似缓存？

what? 咋还有这么多问题？这谁知道啊！

莫急，且看下面的分析。

## 2.1 源码分析法

首先我们可以通过源码对该问题进行分析。

我们知道，Integer var = ? 形式声明变量，会通过 java.lang.Integer#valueOf(int) 来构造 Integer 对象。

很多人可能会说：“你咋能知道这个呢”？

如果不信大家可以打断点，运行程序后会调到这里，总该信了吧？（后面还会再作解释）。

我们先看该函数源码：

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

通过源码可以看出，如果用 `Integer.valueOf(int)` 来创建整数对象，参数大于等于整数缓存的最小值（`IntegerCache.low`）并小于等于整数缓存的最大值（`IntegerCache.high`），会直接从缓存数组（`java.lang.Integer.IntegerCache#cache`）中提取整数对象；否则会 `new` 一个整数对象。

**那么这里的缓存最大和最小值分别是多少呢？**

从上述注释中我们可以看出，最小值是 -128, 最大值是 127。

**那么为什么会缓存这一段区间的整数对象呢？**

通过注释我们可以得知：**如果不要必须新建一个整型对象，缓存最常用的值（提前构造缓存范围内的整型对象），会更省空间，速度也更快。**

这给我们一个非常重要的启发：

如果想减少内存占用，提高程序运行的效率，可以将常用的对象提前缓存起来，需要时直接从缓存中提取。

那么我们再思考下一个问题：**`Integer` 缓存的区间可以修改吗？**

通过上述源码和注释我们还无法回答这个问题，接下来，我们继续看 `java.lang.Integer.IntegerCache` 的源码：

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache;
    static {
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");

        }
    }
}
```

通过 `IntegerCache` 代码和注释我们可以看到，最小值是固定值 -128，最大值并不是固定值，缓存的最大值是通过虚拟机参数 `-XX:AutoBoxCacheMax=<size>` 或 `Djava.lang.Integer.IntegerCache.high=<value>` 来设置的，未指定则为 127。

因此可以通过修改这两个参数其中之一，让缓存的最大值大于等于 150。

如果作出这种修改，示例的输出结果便会是：`true,true`。

**学到这里是不是发现，对此问题的理解和最初的想法有些不同呢？**

这段注释也解答了为什么要缓存这个范围的数据：

**是为了自动装箱时可以复用这些对象，这也是 JLS 的要求。**

If the value `p` being boxed is an integer literal of type `int` between `-128` and `127` inclusive ([§3.10.1](#)), or the boolean literal `true` or `false` ([§3.10.3](#)), or a character literal between `'\u0000'` and `'\u007f'` inclusive ([§3.10.4](#)), then let `a` and `b` be the results of any two boxing conversions of `p`. It is always the case that `a == b`.

在 -128 到 127（含）之间的 `int` 类型的值，或者 `boolean` 类型的 `true` 或 `false`，以及范围在 `'\u0000'` 和 `'\u007f'`（含）之间的 `char` 类型的数值 `p`，自动包装成 `a` 和 `b` 两个对象时，可以使用 `a == b` 判断 `a` 和 `b` 的值是否相等。

## 2.2 反汇编法

那么究竟 `Integer var = ?` 形式声明变量，是不是通过 `java.lang.Integer.valueOf(int)` 来构造 `Integer` 对象呢？总不能都是猜测 `N` 个可能的函数，然后断点调试吧？

**如果遇到其它类似的问题，没人告诉我底层调用了哪个方法，该怎么办？** 囧...

这类问题有个杀手锏，可以通过对编译后的 `class` 文件进行反汇编来查看。

首先编译源代码：`javac IntTest.java`

然后需要对代码进行反汇编，执行：`javap -c IntTest`

如果了解 `javap` 的用法，直接输入 `javap -help` 查看用法提示（很多命令行工具都支持 `-help` 或 `--help` 给出用法提示）。

```
用法: javap <options> <classes>
其中，可能的选项包括：
  -help --help -?      输出此用法消息
  -version             版本信息
  -v -verbose          输出附加信息
  -l                  输出行号和本地变量表
  -public              仅显示公共类和成员
  -protected          显示受保护的/公共类和成员
  -package             显示程序包/受保护的/公共类
                      和成员（默认）
  -p -private          显示所有类和成员
  -c                   对代码进行反汇编
  -s                   输出内部类型签名
  -sysinfo              显示正在处理的类的
                      系统信息（路径，大小，日期，MD5 散列）
  -constants           显示最终常量
  -classpath <path>    指定查找用户类文件的位置
  -cp <path>           指定查找用户类文件的位置
  -bootclasspath <path> 覆盖引导类文件的位置
```

反编译后，我们得到以下代码：

```
Compiled from "IntTest.java"
public class com.chujianyun.common.int_test.IntTest {
    public com.chujianyun.common.int_test.IntTest();
        Code:
            0: aload_0
            1: invokespecial #1
            4: return

    public static void main(java.lang.String[]);
        Code:
            0: bipush        100
            2: invokestatic  #2
```

```

5: astore_1
6: bipush      100
8: invokestatic #2
11: astore_2
12: sipush      150
15: invokestatic #2
18: astore_3
19: sipush      150
22: invokestatic #2
25: astore      4
27: getstatic   #3
30: aload_1
31: aload_2
32: if_acmpne   39
35: iconst_1
36: goto        40
39: iconst_0
40: invokevirtual #4
43: getstatic   #3
46: aload_3
47: aload       4
49: if_acmpne   56
52: iconst_1
53: goto        57
56: iconst_0
57: invokevirtual #4
60: return
}

```

可以明确得 "看到" 这四个 `Integer.valueOf = ?` 形式声明的变量的确是通过 `java.lang.Integer.valueOf(int)` 来构造 `Integer` 对象的。

接下来对汇编后的代码进行详细分析，如果看不懂可略过：

偏移为 0 的指令为： `bipush 100` ，其含义是将单字节整型常量 100 推入操作数栈的栈顶；

偏移为 2 的指令为： `invokestatic #2 // Method java/lang/Integer.valueOf: (I)Ljava/lang/Integer;` 表示调用一个 `static` 函数，即 `java.lang.Integer.valueOf(int)` ；

偏移为 5 的指令为： `astore_1` ，其含义是从操作数栈中弹出对象引用，然后将其存到第 1 个局部变量 Slot 中；

偏移 6 到 25 的指令和上面类似；

偏移为 30 的指令为 `aload_1` ，其含义是从第 1 个局部变量 Slot 取出对象引用（即 a），并将其压入栈；

偏移为 31 的指令为 `aload_2` ，其含义是从第 2 个局部变量 Slot 取出对象引用（即 b），并将其压入栈；

偏移为 32 的指令为 `if_acmpn` ，该指令为条件跳转指令，`if_` 后以 a 开头表示对象的引用比较。

由于该指令有以下特性：

- `if_acmpeq` 比较栈两个引用类型数值，相等则跳转
- `if_acmpne` 比较栈两个引用类型数值，不相等则跳转

由于 Integer 的缓存问题，所以 a 和 b 引用指向同一个地址，因此此条件不成立（成立则跳转到偏移为 39 的指令处），执行偏移为 35 的指令。

偏移为 35 的指令：iconst\_1，其含义为将常量 1 压栈（Java 虚拟机中 boolean 类型的运算类型为 int，其中 true 用 1 表示，详见 2.11.1 数据类型和 Java 虚拟机。

然后执行偏移为 36 的 goto 指令，跳转到偏移为 40 的指令。

偏移为 40 的指令：invokevirtual #4 // Method java/io/PrintStream.println:(Z)V。

可知参数描述符为 Z，返回值描述符为 V。

根据 4.3.2 字段描述符，可知 FieldType 的字符为 Z 表示 boolean 类型，值为 true 或 false。

根据 4.3.3 字段描述符，可知返回值为 void。

因此可以知，最终调用了 java.io.PrintStream#println(boolean) 函数打印栈顶常量即 true。

然后比较执行偏移 43 到 57 之间的指令，比较 c 和 d，打印 false。

执行偏移为 60 的指令，即 retrain，程序结束。

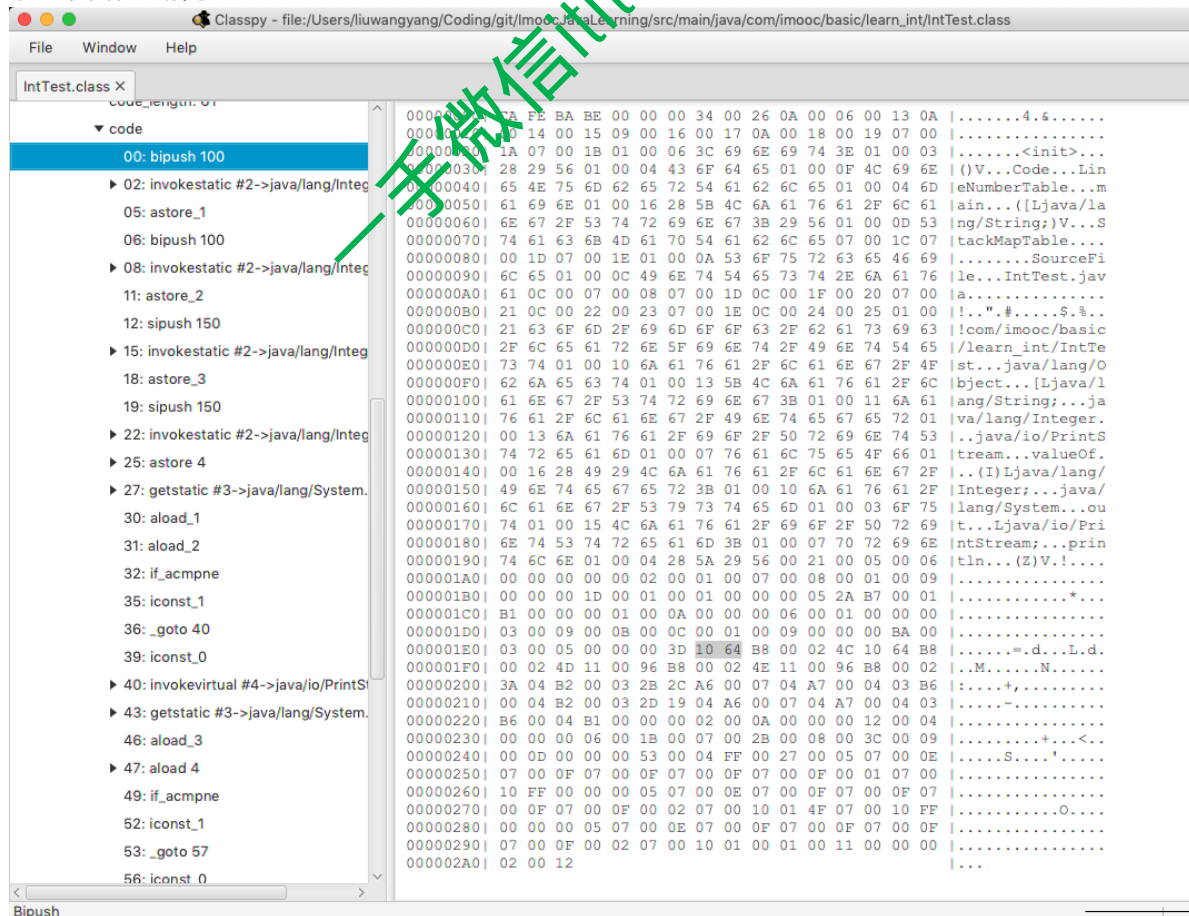
可能有些朋友会对反汇编的代码有些抵触和恐惧，这都是非常正常的现象。

我们分析和研究问题的时候，看懂核心逻辑即可，不要纠结于细节，而失去了重点。

一回生两回熟，随着遇到的例子越来越多，遇到类似的问题时，会喜欢用 javap 来分析和解决问题。

如果想深入学习 java 反汇编，强烈建议结合官方的 JVM 或其中文版：《Java 虚拟机规范》这本书进行拓展学习。

如果大家不喜欢命令行的方式进行 Java 的反汇编，这里推荐一个简单易用的可视化工具：classpy，大家可以自行了解学习。



我们学习的目的之一就是要学会举一反三。因此我们对 Long 也进行类似的研究，探究两者之间有何异同。

## 3.1 源码分析

类似的，我们接下来分析 `java.lang.Long#valueOf(long)` 的源码：

```
public static Long valueOf(long l) {
    final int offset = 128;
    if (l >= -128 && l <= 127) {
        return LongCache.cache[(int)l + offset];
    }
    return new Long(l);
}
```

发现该函数的写法和 `Integer.valueOf(int)` 非常相似。

我们同样也看到，`Long` 也用到了缓存。使用 `java.lang.Long#valueOf(long)` 构造 `Long` 对象时，值在 `[-128, 127]` 之间的 `Long` 对象直接从缓存对象数组中提取。

而且注释同样也提到了：**缓存的目的是为了提高性能。**

但是通过注释我们发现这么一段提示：

Note that unlike the `{@linkplain Integer#valueOf(int)}` corresponding method in the `{@code Integer}` class, this method is *not* required to cache values within a particular range.

注意：和 `Integer.valueOf(int)` 不同的是，此方法并没有被要求缓存特定范围的值。

这也正是上面源码中缓存范围判断的注释为何用 `// will cache` 的原因（可以对比一下上面 `Integer` 的缓存的注释）。

因此我们可知，虽然此处采用了缓存，但应该不是 JLS 的要求。

那么 `Long` 类型的缓存是如何构造的呢？

我们查看缓存数组的构造：

```
private static class LongCache {
    private LongCache(){}

    static final Long cache[] = new Long[-(-128) + 127 + 1];

    static {
        for(int i = 0; i < cache.length; i++)
            cache[i] = new Long(i - 128);
    }
}
```

可以看到，它是在静态代码块中填充缓存数组的。

## 3.2 反编译

同样地我们也编写一个示例片段：

```

public class LongTest {

    public static void main(String[] args) {
        Long a = -128L, b = -128L, c = 150L, d = 150L;
        System.out.println(a == b);
        System.out.println(c == d);
    }
}

```

编译源代码: `javac LongTest.java`

对编译后的类文件进行反汇编: `javap -c LongTest`

得到下面反编译的代码:

```

public class com.imooc.basic.learn_int.LongTest {
    public com.imooc.basic.learn_int.LongTest();
    Code:
        0: aload_0
        1: invokespecial #1
        4: return

    public static void main(java.lang.String[]);
    Code:
        0: ldc2_w        #2
        3: invokestatic  #4
        6: astore_1
        7: ldc2_w        #2
       10: invokestatic  #4
       13: astore_2
       14: ldc2_w        #5
       17: invokestatic  #4
       20: astore_3
       21: ldc2_w        #5
       24: invokestatic  #4
       27: astore        4
       29: getstatic     #7
       32: aload_1
       33: aload_2
       34: if_acmpne     41
       37: iconst_1
       38: goto          42
       41: iconst_0
       42: invokevirtual #8
       45: getstatic     #7
       48: aload_3
       49: aload         4
       51: if_acmpne     58
       54: iconst_1
       55: goto          59
       58: iconst_0
       59: invokevirtual #8
       62: return
}

```



我们从上述代码中发现 `Long var = ?` 的确是通过 `java.lang.Long#valueOf(long)` 来构造对象的。

本小节通过源码分析法、阅读 JLS 和 JVMs、使用反汇编法，对 `Integer` 和 `Long` 缓存的目的和实现方式问题进行了深入分析。

让大家能够通过更丰富的手段来学习知识和分析问题，通过对缓存目的的思考来学到更通用和本质的东西。

本节使用的几种手段将是我们未来常用的方法，也是工作进阶的必备技能和一个程序员专业程度的体现，希望大家未来能够多动手实践。

下一节我们将介绍 Java 序列化相关问题，包括序列化的定义，序列化常见的方案，序列化的坑点等。

**第 1 题：**请大家根据今天的研究分析过程，对下面的一个示例代码进行分析。

```
public class CharacterTest {
    public static void main(String[] args) {
        Character a = 126, b = 126, c = 128, d = 128;
        System.out.println(a == b);
        System.out.println(c == d);
    }
}
```

**第 2 题：**结合今天的讲解，请自行对 `Character`、`Short`、`Boolean` 的缓存问题进行分析，并比较它们的异同。

```
}
```