

二

07 int和Integer有什么区别？-极客时间

Java 虽然号称是面向对象的语言，但是原始数据类型仍然是重要的组成元素，所以在面试中，经常考察原始数据类型和包装类等 Java 语言特性。

今天我要问你的问题是，**int 和 Integer 有什么区别？谈谈 Integer 的值缓存范围。**

典型回答

int 是我们常说的整形数字，是 Java 的 8 个原始数据类型（Primitive Types，boolean、byte、short、char、int、float、double、long）之一。**Java 语言虽然号称一切都是对象，但原始数据类型是例外。**

Integer 是 int 对应的包装类，它有一个 int 类型的字段存储数据，并且提供了基本操作，比如数学运算、int 和字符串之间转换等。在 Java 5 中，引入了自动装箱和自动拆箱功能（boxing/unboxing），Java 可以根据上下文，自动进行转换，极大地简化了相关编程。

关于 Integer 的值缓存，这涉及 Java 5 中另一个改进。构建 Integer 对象的传统方式是直接调用构造器，直接 new 一个对象。但是根据实践，我们发现大部分数据操作都是集中在有限的、较小的数值范围，因而，在 Java 5 中新增了静态工厂方法 valueOf，在调用它的时候会利用一个缓存机制，带来了明显的性能改进。按照 Javadoc，**这个值默认缓存是 -128 到 127 之间。**

考点分析

今天这个问题涵盖了 Java 里的两个基础要素：原始数据类型、包装类。谈到这里，就可以非常自然地扩展到自动装箱、自动拆箱机制，进而考察封装类的一些设计和实践。坦白说，理解基本原理和用法已经足够日常工作需求了，但是要落实到具体场景，还是有很多问题需要仔细思考才能确定。

面试官可以结合其他方面，来考察面试者的掌握程度和思考逻辑，比如：

- 我在专栏第 1 讲中介绍的 Java 使用的不同阶段：编译阶段、运行时，自动装箱 / 自动

拆箱是发生在什么阶段？

- 我在前面提到使用静态工厂方法 `valueOf` 会使用到缓存机制，那么自动装箱的时候，缓存机制起作用吗？
- 为什么我们需要原始数据类型，Java 的对象似乎也很高效，应用中具体会产生哪些差异？
- 阅读过 `Integer` 源码吗？分析下类或某些方法的设计要点。

似乎有太多内容可以探讨，我们一起来分析一下。

知识扩展

1. 理解自动装箱、拆箱

自动装箱实际上算是一种**语法糖**。什么是语法糖？可以简单理解为 Java 平台为我们自动进行了一些转换，保证不同的写法在运行时等价，它们发生在编译阶段，也就是生成的字节码是一致的。

像前面提到的整数，`javac` 替我们自动把装箱转换为 `Integer.valueOf()`，把拆箱替换为 `Integer.intValue()`，这似乎这也顺道回答了另一个问题，既然调用的是 `Integer.valueOf`，自然能够得到缓存的好处啊。

如何程序化的验证上面的结论呢？

你可以写一段简单的程序包含下面两句代码，然后反编译一下。当然，这是一种从表现倒推的方法，大多数情况下，我们还是直接参考规范文档会更加可靠，毕竟软件承诺的是遵循规范，而不是保持当前行为。

```
Integer integer = 1;
int unboxing = integer ++;
```

反编译输出：

```
1: invokestatic #2          // Method
  java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
8: invokevirtual #3          // Method
  java/lang/Integer.intValue:()I
```

这种缓存机制并不是只有 `Integer` 才有，同样存在于其他的一些包装类，比如：

- Boolean，缓存了 true/false 对应实例，确切说，只会返回两个常量实例 Boolean.TRUE/FALSE。
- Short，同样是缓存了 -128 到 127 之间的数值。
- Byte，数值有限，所以全部都被缓存。
- Character，缓存范围'\u0000' 到 '\u007F'。

自动装箱 / 自动拆箱似乎很酷，在编程实践中，有什么需要注意的吗？

原则上，**建议避免无意中的装箱、拆箱行为**，尤其是在性能敏感的场所，创建 10 万个 Java 对象和 10 万个整数的开销可不是一个数量级的，不管是内存使用还是处理速度，光是对象头的空间占用就已经是数量级的差距了。

我们其实可以把这个观点展开，使用原始数据类型、数组甚至本地代码实现等，在性能极度敏感的场景往往具有比较大的优势，用其替换掉包装类、动态数组（如 ArrayList）等可以作为性能优化的备选项。一些追求极致性能的产品或者类库，会极力避免创建过多对象。当然，在大多数产品代码里，并没有必要这么做，还是以开发效率优先。以我们会经常使用的计数器实现为例，下面是一个常见的线程安全计数器实现。

```
class Counter {  
    private final AtomicLong counter = new AtomicLong();  
    public void increase() {  
        counter.incrementAndGet();  
    }  
}
```

如果利用原始数据类型，可以将其修改为

```
class CompactCounter {  
    private volatile long counter;  
    private static final AtomicLongFieldUpdater<CompactCounter> updater = AtomicLongFieldUpdater.  
        of(CompactCounter.class);  
    public void increase() {  
        updater.incrementAndGet(this);  
    }  
}
```

2. 源码分析

考察是否阅读过、是否理解 JDK 源代码可能是部分面试官的关注点，这并不完全是一种苛刻要求，阅读并实践高质量代码也是程序员成长的必经之路，下面我来分析下 Integer 的源码。

整体看一下 Integer 的职责，它主要包括各种基础的常量，比如最大值、最小值、位数等；前面提到的各种静态工厂方法 `valueOf()`；获取环境变量数值的方法；各种转换方法，比如转换为不同进制的字符串，如 8 进制，或者反过来的解析方法等。我们进一步来看一些有意思的地方。

首先，继续深挖缓存，Integer 的缓存范围虽然默认是 -128 到 127，但是在特别的应用场景，比如我们明确知道应用会频繁使用更大的数值，这时候应该怎么办呢？

缓存上限值实际是可以根据需要调整的，JVM 提供了参数设置：

```
-XX:AutoBoxCacheMax=N
```

这些实现，都体现在 `java.lang.Integer` 源码之中，并实现在 `IntegerCache` 的静态初始化块里。

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];
    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue = VM.getSavedProperty("
        ...
        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }
    ...
}
```

第二，我们在分析字符串的设计实现时，提到过字符串是不可变的，保证了基本的信息安全和并发编程中的线程安全。如果你去看包装类里存储数值的成员变量“`value`”，你会发现，不管是 Integer 还 Boolean 等，都被声明为“`private final`”，所以，它们同样是不可变类型！

这种设计是可以理解的，或者说是必须的选择。想象一下这个应用场景，比如 Integer 提供了 `getInteger()` 方法，用于方便地读取系统属性，我们可以用属性来设置服务器某个服务的端口，如果我可以轻易地把获取到的 Integer 对象改变为其他数值，这会带来产品可靠性方面的严重问题。

第三，Integer 等包装类，定义了类似 SIZE 或者 BYTES 这样的常量，这反映了什么样的设计考虑呢？如果你使用过其他语言，比如 C、C++，类似整数的位数，其实是不确定的，可能在不同的平台，比如 32 位或者 64 位平台，存在非常大的不同。那么，在 32 位 JDK 或者 64 位 JDK 里，数据位数会有不同吗？或者说，这个问题可以扩展为，我使用 32 位 JDK 开发编译的程序，运行在 64 位 JDK 上，需要做什么特别的移植工作吗？

其实，这种移植对于 Java 来说相对要简单些，因为原始数据类型是不存在差异的，这些明确定义在[Java 语言规范](#)里面，不管是 32 位还是 64 位环境，开发者无需担心数据的位数差异。

对于应用移植，虽然存在一些底层实现的差异，比如 64 位 HotSpot JVM 里的对象要比 32 位 HotSpot JVM 大（具体区别取决于不同 JVM 实现的选择），但是总体来说，并没有行为差异，应用移植还是可以做到宣称的“一次书写，到处执行”，应用开发者更多需要考虑的是容量、能力等方面的差异。

3. 原始类型线程安全

前面提到了线程安全设计，你有没有想过，原始数据类型操作是不是线程安全的呢？

这里可能存在着不同层面的问题：

- 原始数据类型的变量，显然要使用并发相关手段，才能保证线程安全，这些我会在专栏后面的并发主题详细介绍。如果有线程安全的计算需要，建议考虑使用类似 `AtomicInteger`、`AtomicLong` 这样的线程安全类。
- 特别的是，部分比较宽的数据类型，比如 `float`、`double`，甚至不能保证更新操作的原子性，可能出现程序读取到只更新了一半数据位的数值！

4. Java 原始数据类型和引用类型局限性

前面我谈了非常多的技术细节，最后再从 Java 平台发展的角度来看，原始数据类型、对象的局限性和演进。

对于 Java 应用开发者，设计复杂而灵活的类型系统似乎已经习以为常了。但是坦白说，毕竟这种类型系统的设计是源于很多年前的技术决定，现在已经逐渐暴露出了一些副作用，例如：

- 原始数据类型和 Java 泛型并不能配合使用

这是因为 Java 的泛型某种程度上可以算作伪泛型，它完全是一种编译期的技巧，Java 编译器会自动将类型转换为对应的特定类型，这就决定了使用泛型，必须保证相应类型可以转换为 `Object`。

- 无法高效地表达数据，也不便于表达复杂的数据结构，比如 `vector` 和 `tuple`

我们知道 Java 的对象都是引用类型，如果是一个原始数据类型数组，它在内存里是一段连续的内存，而对象数组则不然，数据存储的是引用，对象往往是分散地存储在堆的不同位置。这种设计虽然带来了极大灵活性，但是也导致了数据操作的低效，尤其是无法充分利用

现代 CPU 缓存机制。

Java 为对象内建了各种多态、线程安全等方面的支持，但这不是所有场合的需求，尤其是数据处理重要性日益提高，更加高密度的值类型是非常现实的需求。

针对这些方面的增强，目前正在 OpenJDK 领域紧锣密鼓地进行开发，有兴趣的话你可以关注相关工程：<http://openjdk.java.net/projects/valhalla/>。

今天，我梳理了原始数据类型及其包装类，从源码级别分析了缓存机制等设计和实现细节，并且针对构建极致性能的场景，分析了一些可以借鉴的实践。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？留一道思考题给你，前面提到了从空间角度，Java 对象要比原始数据类型开销大的多。你知道对象的内存结构是什么样的吗？比如，对象头的结构。如何计算或者获取某个 Java 对象的大小？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。

[上一页](#)

[下一页](#)