第7讲 | int和Integer有什么区别?

2018-05-19 杨晓峰

第7讲 | int和Integer有什么区别?

杨晓峰

00:00 / 11:04

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合，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扩展开，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。以我们经常会使用到的计数器实现为例，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.newUpdater(CompactCounter.class, "counter");
    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("java.lang.Integer.IntegerCache.high");
            ...
            // 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编译期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。

cookie。

2018-05-19

对象由三部分组成，对象头，对象实例，对齐填充。
其中对象头一般是十六个字节，包括两部分，第一部分有哈希码，锁状态标志，线程持有的锁，偏向线程id，gc分代年龄等。第二部分是类型指针，也就是对象指向它的类元数据指针，可以理解，对象指向它的类。
对象实例就是对象存储的真正有效信息，也是程序中定义各种类型的字段包括父类继承的和子类定义的，这部分的存储顺序会被虚拟机和代码中定义的顺序影响（这里问一下，这个被虚拟机影响是不是就是重排序？？如果是的话，我知道的volatile定义的变量不会被重排序应该就是这里不会受虚拟机影响吧？？）。
第三部分对齐填充只是一个类似占位符的作用，因为内存的使用都会被填充为八字节的倍数。

还是个初学者。以上是我了解，不知道有没有错，希望老师能告知。

Kyle

2018-05-19

节选自《深入理解JAVA虚拟机》：
在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

HotSpot虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，这部分数据的长度在32位和64位的虚拟机（未开启压缩指针）中分别为32bit和64bit，官方称它为"Mark Word"。

对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。并不是所有的虚拟机实现都必须在对象数据上保留类型指针，换句话说，查找对象的元数据信息并不一定要经过对象本身，这点将在2.3.3节讨论。另外，如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通Java对象的元数据信息确定Java对象的大小，但是从数组的元数据中却无法确定数组的大小。

接下来的实例数据部分是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。无论是从父类继承下来的，还是在子类中定义的，都需要记录起来。

第三部分对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。由于HotSpot VM的自动内存管理系统要求对象起始地址必须是8字节的整数倍，换句话说，就是对象的大小必须是8字节的整数倍。

公号-Java大后端

2018-05-19

1 int和Integer

JDK1.5引入了自动装箱与自动拆箱功能，Java可根据上下文，实现int/Integer,double/Double,boolean/Boolean等基本类型与相应对象之间的自动转换，为开发过程带来极大便利。

最常用的是通过new方法构建Integer对象。但是，基于大部分数据操作都是集中在有限的、较小的数值范围，在JDK1.5 中新增了静态工厂方法 valueOf，其背后实现是将int值为-128 到 127 之间的Integer对象进行缓存，在调用时候直接从缓存中获取，进而提升构建对象的性能，也就是说使用该方法后，如果两个对象的int值相同且落在缓存值范围内，那么这个两个对象就是同一个对象；当值较小且频繁使用时，推荐优先使用整型池方法（时间与空间性能俱佳）。

2 注意事项

[1] 基本类型均具有取值范围，在大数*大数的时候，有可能会出现越界的情况。
[2] 基本类型转换时，使用声明的方式。例：long result= 1234567890 * 24 * 365；结果值一定不会是你所期望的那个值，因为1234567890 * 24已经超过了int的范围，如果修改为：long result= 1234567890L * 24 * 365；就正常了。
[3] 慎用基本类型处理货币存储。如采用double常会带来差距，常采用BigDecimal、整型（如果要精确表示分，可将值扩大100倍转化为整型）解决该问题。
[4] 优先使用基本类型。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合，
[5] 如果有线程安全的计算需要，建议考虑使用类型AtomicInteger、AtomicLong 这样的线程安全类。部分比较宽的基本数据类型，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。

kursk.ye

2018-06-13

这篇文章写得比较零散，整体思路没有串起来，其实我觉得可以从这么一条线索理解这个问题。原始数据类型和 Java 泛型并不能配合使用，也就是Primitive Types 和Generic 不能混用，于是JAVA就设计了这个auto-boxing/unboxing机制，实际上就是primitive value 与 object之间的隐式转换机制，否则要是没有这个机制，开发者就必须每次手动显示转换，那多麻烦是不是？但是primitive value 与 object各自有各自的优势，primitive value在内存中存的是值，所以找到primitive value的内存位置，就可以获得值；不像object存的是reference，找到object的内存位置，还要根据reference找下一个内存空间，要产生更多的IO，所以计算性能比primitive value差，但是object具备generic的能力，更抽象，解决业务问题编程效率高。于是JAVA设计者的初衷估计是这样的：如果开发者要做计算，就应该使用primitive value如果开发者要处理业务问题，就应该使用object，采用Generic机制；反正JAVA有auto-boxing/unboxing机制，对开发者来讲也不需要注意什么。然后为了弥补object计算能力的不足，还设计了static valueOf()方法提供缓存机制，算是一个弥补。

行者

2018-05-20

1. Mark Word:标记位 4字节，类似轻量级锁标记位，偏向锁标记位等。
2. Class对象指针:4字节，指向对象对应class对象的内存地址。
3. 对象实际数据:对象所有成员变量。
4. 对齐:对齐填充字节，按照8个字节填充。

Integer占用内存大小，4+4+4+4=16字节。

作者回复

2018-05-22

不错，如果是64位不用压缩指针，对象头会变大，还可能有对齐开销

麦田

2018-05-19

周末了是不是没人看文章了

George

2018-05-25

计算对象大小可通过dump内存之后用memory analyze分析

作者回复

2018-05-25

嗯，也可以利用：
jol，jmap，或者instrument api（Java agent）等等

George

2018-05-25

java内存结构
对象头：
markword：用于存储对象自身的运行时数据，如哈希码、GC分代年龄、锁状态标志、线程持有的锁等。这部分数据长度在32位机器和64位机器虚拟机中分别为4字节和8字节；
lass指针：即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象属于哪个类的实例；
length：如果是java数组，对象头必须有一块用于记录数组长度的数据，用4个字节来int来记录数组长度；
实例数据
实例数据是对象真正存储的有效信息，也是程序代码中定义的各种类型的字段内容。无论是从父类继承下来还是在子类中定义的数据，都需要记录下来
堆积填充
对于hotspot迅疾的自动内存管理系统要求对象的起始地址必须为8字节的整数倍，这就要求当部位8字节的整数倍时，就需要填充数据对其填充。原因是访问未对齐的内存，处理器需要做两次内存访问，而对齐的内存访问仅需一次访问

Miaozhe

2018-05-21

杨老师，问个问题，如果使用原始类型int定义一个变量在-128和127之间，如int c = 64;会放入Integer 常量缓存吗(IntegerCache)？编译器是怎么操作的?

作者回复

2018-05-23

不需要，不是对象

两只🐱

2018-05-19

原始数据类型貌似反射也不行。

Gerald

2018-05-29

为什么我感觉都这么难啊🙈

作者回复

2018-05-29

感谢反馈，具体哪个方面，我可以调整一下，尽量照顾不同基础的朋友

ZC叶��

2018-05-22

想问下 自动装箱和自动拆箱是指类型转换吗？

作者回复

2018-05-23

这个...似乎也算，如果你的"转换"是conversion，不是casting

---

jutsu

2018-05-20

老师的讲解让我想起了科比主导的 细节栏目

步 * 亮

2018-05-19

缓存用得很巧秒，值得借鉴

---

hansc

2018-05-19

垃圾回收分带年龄，hashcode值，锁标记，请问对象逃过垃圾回收的次数记录到哪里呢？

---

feifei

2018-07-03

JAVA的内存结构分为3部分：
1，对象头 有两部分,markWord和Class对象指针，
markwork包括存储对象自身的运行时数据， 如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳，
2，实例数据
3，对齐填充

获取一个JAVA对象的大小，可以将一个对象进行序列化为二进制的Byte，便可以查看大小，
Integer value = 10;

```
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(value);
// // 读出当前对象的二进制流信息
System.out.println(bos.size());
```

遗忘明天

2018-06-22

long的赋值也不是原子操作吗？

遗忘明天

2018-06-22

long的赋值也不是原子操作吗？

Darren

2018-06-16

老师，原始数据类型的包装类是对象吗？

作者回复

2018-06-16

类是类，实例化后才是对象

Darren

2018-06-16

老师，反编译输出怎么理解的，看不懂语法

作者回复

2018-06-17

具体哪一段，是文章中片段的invokestatic之类吗？如果是的话，最准确的可以参考java虚拟机规范，查询相应之类；大多数情况下可以搜索相关分析文章，理解难道会小些

---

tracer

2018-06-12

integer获取环境变量数值的方法，这个具体是指哪个方法？

作者回复

2018-06-12

getInteger，建议看看文档

不瘦十斤不换名字

2018-05-24

为啥大家都在讨论对象的组成部分��

梁作斌

2018-05-24

不是原子操作的基本类型是 float 、double？为啥不是 long、double？

作者回复

2018-05-25

那是举例，不是定义

Slug

2018-05-19

感谢老师放假还在写文章，学到很多，钱花的很值。

Hua

2018-05-19

希望老师多写一些文章这样我就不用看源码了。

极客时间