



Java 数据类型常见面试题总结

这篇文章绝对干货！文章涉及到的概念经常会被面试官拿来考察求职者的 Java 基础。


本篇采用大家比较喜欢的面试官问答的形式来展开。

基本数据类型

 面试官：Java 中有哪 8 种基本数据类型？

 我：Java 中有 8 种基本数据类型，分别为：

- 1. 6 种数字类型：`byte`、`short`、`int`、`long`、`float`、`double`
- 2. 1 种字符类型：`char`
- 3. 1 种布尔型：`boolean`。

 面试官：它们的默认值和占用的空间大小知道不？

 我：这 8 种基本数据类型的默认值以及所占空间的大小如下：


基本类型	位数	字节	默认值
<code>int</code>	32	4	0
<code>short</code>	16	2	0
<code>long</code>	64	8	0L
<code>byte</code>	8	1	0
<code>char</code>	16	2	'u0000'
<code>float</code>	32	4	0f
<code>double</code>	64	8	0d
<code>boolean</code>	1		false

另外，对于 `boolean`，官方文档未明确定义，它依赖于 JVM 厂商的具体实现。逻辑上理解是占用 1 位，但是实际中会考虑计算机高效存储因素。

注意：

- 1. Java 里使用 `long` 类型的数据一定要在数值后面加上 L，否则将作为整型解析：
- 2. `char a = 'h'` char:单引号，`String a = "hello"` :双引号

包装类型

 面试官：说说这 8 种基本数据类型对应的包装类型。

 我：这八种基本类型都有对应的包装类分别为：`Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`、`Character`、`Boolean`

 面试官：那基本类型和包装类型有啥区别不？

 我：

- **用途**：除了定义一些常量和局部变量之外，我们在其他地方比如方法参数、对象属性中很少会使用基本类型来定义变量。并且，包装类型可用于泛型，而基本类型不可以。
- **存储方式**：基本数据类型的局部变量存放在 Java 虚拟机栈中的局部变量表中，基本数据类型的成员变量（未被 `static` 修饰）存放在 Java 虚拟机的堆中。包装类型属于对象类型，我们知道几乎所有对象实例都存在于堆中。
- **占用空间**：相比于包装类型（对象类型），基本数据类型占用的空间往往非常小。
- **默认值**：成员变量包装类型不赋值就是 `null`，而基本类型有默认值且不是 `null`。
- **比较方式**：对于基本数据类型来说，`==` 比较的是值。对于包装数据类型来说，`==` 比较的是对象的内存地址。所有整型包装类对象之间值的比较，全部使用 `equals()` 方法。

为什么说几乎所有对象实例都存在于堆中呢？这是因为 HotSpot 虚拟机引入了 JIT 优化之后，会对对象进行逃逸分析，如果发现某一个对象并没有逃逸到方法外部，那么就可能通过标量替换来实现栈上分配，而避免堆上分配内存


⚠ 注意：基本数据类型存放在栈中是一个常见的误区！基本数据类型的成员变量如果没有被 `static` 修饰的话（不建议这么使用，应该要使用基本数据类型对应的包装类型），就存放在堆中。


▼

Java |

```
1  class BasicTypeVar{
2      private int x;
3  }
```

包装类型的常量池技术

 面试官：包装类型的常量池技术了解么？

 我：Java 基本类型的包装类的大部分都实现了常量池技术。

`Byte`、`Short`、`Integer`、`Long` 这 4 种包装类默认创建了数值 `[-128, 127]` 的相应类型的缓存数据，`Character` 创建了数值在 `[0,127]`范围的缓存数据，`Boolean` 直接返回 `True` Or `False`。

Integer 缓存源码:

```
1  /**
2   * 利用 IntegerCache 类实现了对指定范围内整数对象的缓存。
3   * 默认情况下，这个缓存范围是 -128 到 127（包含边界值）。
4   */
5   public static Integer valueOf(int i) {
6       if (i >= IntegerCache.low && i <= IntegerCache.high)
7           return IntegerCache.cache[i + (-IntegerCache.low)];
8       return new Integer(i);
9   }
10  private static class IntegerCache {
11      static final int low = -128;
12      static final int high;
13      static final Integer cache[];
14  }
```

Character 缓存源码:

```
1  public static Character valueOf(char c) {
2      if (c <= 127) { // must cache
3          return CharacterCache.cache[(int)c];
4      }
5      return new Character(c);
6  }
7  private static class CharacterCache {
8      private CharacterCache(){}
9
10     static final Character cache[] = new Character[127 + 1];
11     static {
12         for (int i = 0; i < cache.length; i++)
13             cache[i] = new Character((char)i);
14     }
15 }
```

Boolean 缓存源码:

```
1  public static Boolean valueOf(boolean b) {
2      return (b ? TRUE : FALSE);
3  }
```

如果超出对应范围仍然会去创建新的对象，缓存的范围区间的大小只是在性能和资源之间的权衡。

两种浮点数类型的包装类 `Float`，`Double` 并没有实现常量池技术。

```
1  Integer i1 = 33;
2  Integer i2 = 33;
3  System.out.println(i1 == i2); // 输出 true
4  Float i11 = 333f;
5  Float i22 = 333f;
6  System.out.println(i11 == i22); // 输出 false
7  Double i3 = 1.2;
8  Double i4 = 1.2;
9  System.out.println(i3 == i4); // 输出 false
```

下面我们来看一下问题。下面的代码的输出结果是 `true` 还是 `false` 呢？

```
1  Integer i1 = 40;
2  Integer i2 = new Integer(40);
3  System.out.println(i1==i2);
```

`Integer i1=40` 这一行代码会发生装箱，也就是说这行代码等价于 `Integer i1=Integer.valueOf(40)`。因此，`i1` 直接使用的是常量池中的对象。而 `Integer i2 = new Integer(40)` 会直接创建新的对象。

因此，答案是 `false`。你答对了吗？

记住：所有整型包装类对象之间值的比较，全部使用 `equals` 方法比较。

7. 【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` 之间的赋值，`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。

为什么要有包装类型？

面试官：为什么要有包装类型？

我：

Java 本身就是一门 OOP（面向对象编程）语言，对象可以说是 Java 的灵魂。

除了定义一些常量和局部变量之外，我们在其他地方比如方法参数、对象属性中很少会使用基本类型来定义变量。

为什么呢？

我举个例子，假如你有一个对象中的属性使用了基本类型，那这个属性就必然存在默认值了。这个逻辑不正确的！因为很多业务场景下，对象的某些属性没有赋值，我就希望它的值为 `null`。你给我默认赋值，不是帮倒忙么？

另外，像泛型参数不能是基本类型。因为基本类型不是 `Object` 子类，应该用基本类型对应的包装类型代替。我们直接拿 JDK 中线程的代码举例。

Java 中的集合在定义类型的时候不能使用基本类型的。比如：

```
1 public class HashMap<K,V> extends AbstractMap<K,V>
2 # implements Map<K,V>, Cloneable, Serializable {
3 }
4
5 Map<Integer, Set<String>> map = new HashMap<>();
```

自动拆装箱

什么是自动拆装箱？原理？

面试官：什么是自动拆装箱？原理解么？

我：

基本类型和包装类型之间的互转。举例：

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

上面这两行代码对应的字节码为：

```
1 L1
2 LINENUMBER 8 L1
3 ALOAD 0
4 BIPUSH 10
5 INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;
6 PUTFIELD AutoBoxTest.i : Ljava/lang/Integer;
7 L2
8 LINENUMBER 9 L2
9 ALOAD 0
10 ALOAD 0
11 GETFIELD AutoBoxTest.i : Ljava/lang/Integer;
12 INVOKEVIRTUAL java/lang/Integer.intValue ()I
13 PUTFIELD AutoBoxTest.n : I
14 RETURN
```

从字节码中，我们发现装箱其实就是调用了包装类的 `valueOf()` 方法，拆箱其实就是调用了 `xxxValue()` 方法。

因此，

- `Integer i = 10` 等价于 `Integer i = Integer.valueOf(10)`
- `int n = i` 等价于 `int n = i.intValue()`;

自动拆箱引发的 NPE 问题

面试官：自动拆箱可能会引发 NPE 问题，遇到过类似的场景么？

我：

案例 1

在《阿里巴巴开发手册》上就有这样一条规定。

12.关于基本数据类型与包装数据类型的使用标准如下：

- 1) 【强制】所有的 POJO 类属性必须使用包装数据类型。
- 2) 【强制】RPC 方法的返回值和参数必须使用包装数据类型。
- 3) 【推荐】所有的局部变量使用基本数据类型。

说明：POJO 类属性没有初值是提醒使用者在需要使用时，必须自己显式地进行赋值，任何 NPE 问题，或者入库检查，都由使用者来保证。

正例：数据库的查询结果可能是 null，因为自动拆箱，用基本数据类型接收有 NPE 风险。

反例：某业务的交易报表上显示成交总额涨跌情况，即正负 x%，x 为基本数据类型，调用的 RPC 服务，调用不成功时，返回的是默认值，页面显示为 0%，这是不合理的，应该显示成中划线-。所以包装数据类型的 null 值，能够表示额外的信息，如：远程调用失败，异常退出。

我们从上图可以看到，有一条是这样说的：“数据库的查询结果可能是 null，因为自动拆箱，用基本数据类型接收有 NPE 风险”。

我们来模拟一个实际的案例：

```
1 public class AutoBoxTest {
2     @Test
3     void should_Throw_NullPointerException(){
4         long id = getNum();
5     }
6     public Long getNum(){
7         return null;
8     }
9 }
```

运行代码之后，果然出现了 NPE 的问题。

为什么会这样呢? 我们对 AutoBoxTest.class 进行反编译查看其字节码（我更推荐使用 IDEA 插件 jclasslib 来查看类的字节码）。

```
1 javap -c AutoBoxTest.class
```

反编译后得到的 should_Throw_NullPointerException() 方法的字节码如下：

```
1 0 aload_0
2 1 invokevirtual #2 <AutoBoxTest.getNum>
3 4 invokevirtual #3 <java/lang/Long.longValue>
4 7 lstore_1
5 8 return
```

我们可以发现自动拆箱 Long -> long 的过程，不过是调用了 longValue() 方法罢了！

```
1 public long longValue() {
2     return value;
3 }
```

也就是说下面两行的代码实际是等价的：

```
1 long id = getNum();
2 long id = getNum().longValue();
```

因为，getNum() 返回的值为 null，一个 null 值调用方法，当然会有 NPE 的问题了。

案例 2

通过上面的分析之后，我来考了一个不论是平时开发还是面试中都会经常碰到的一个问题：“三目运算符使用不当会导致诡异的 NPE 异常”。

请你回答下面的代码会有 NPE 问题出现吗？如果有 NPE 问题出现的话，原因是什么呢？你会怎么分析呢？

```
1 public class Main {
2     public static void main(String[] args) {
3         Integer i = null;
4         Boolean flag = false;
5         System.out.println(flag ? 0 : i);
6     }
7 }
```

答案是会有 NPE 问题出现的。

我们还是通过查看其字节码来搞懂背后的原理（这里借助了 IDEA 插件 jclasslib 来查看类字节码）。

```
0 aconst_null
1 astore_1
2 iconst_0
3 invokestatic #2 <java/lang/Boolean.valueOf>
6 astore_2
7 getstatic #3 <java/lang/System.out>
10 aload_2
11 invokevirtual #4 <java/lang/Boolean.booleanValue>
14 ifeq 21 (+7)
17 iconst_0
18 goto 25 (+7)
21 aload_1
22 invokevirtual #5 <java/lang/Integer.intValue>
25 invokevirtual #6 <java/io/PrintStream.println>
28 return
```

从字节码中可以看出，22 行的位置发生了拆箱操作。

详细解释下就是：flag ? 0 : i 这行代码中，0 是基本数据类型 int，返回数据的时候 i 会被强制拆箱成 int 类型，由于 i 的值是 null，因此就抛出了 NPE 异常。

```
1 Integer i = null;
2 Boolean flag = false;
3 System.out.println(flag ? 0 : i);
```

如果，我们把代码中 flag 变量的值修改为 true 的话，就不会存在 NPE 问题了，因为会直接返回 0，不会进行拆箱操作。

我们在实际项目中应该避免这样的写法，正确 ☒ 修改之后的代码如下：

```
1 Integer i = null;
2 Boolean flag = false;
3 System.out.println(flag ? new Integer(0) : i); // 两者类型一致就不会有拆箱导致的 NPE 问题了
```

这个问题也在《阿里巴巴开发手册》中被提到过。

前言
目录
一、编程规范
(一) 命名风格
(二) 常量定义
(三) 代码格式
(四) OOP规范
(五) 日期时间
(六) 集合处理
(七) 并发处理
(八) 控制语句
(九) 注释规范
(十) 前后端规范
(十一) 其他
二、异常日志
三、单元测试
四、安全规范
五、MySQL数据库
六、工程结构
七、设计规则
附录1：版本历史
附录2：专有名词解释
附录3：错误码列表

3. 【强制】在 if/else/for/while/do 语句中必须使用大括号。

说明：即使只有一行代码，也禁止不采用大括号的编码方式：if (condition) statements;

4. 【强制】三目运算符 condition? 表达式 1：表达式 2 中，高度注意表达式 1 和 2 在类型对齐时，可能抛出因自动拆箱导致的 NPE 异常。

说明：以下两种场景会触发类型对齐的拆箱操作：

1) 表达式 1 或表达式 2 的值只要有一个是原始类型。

2) 表达式 1 或表达式 2 的值的类型不一致，会强制拆箱升级成表示范围更大的那个类型。

反例：

```
Integer a = 1;
Integer b = 2;
Integer c = null;
Boolean flag = false;
// a*b 的结果是 int 类型，那么 c 会强制拆箱成 int 类型，抛出 NPE 异常
Integer result = (flag ? a*b : c);
```