

55 | 享元模式（下）：剖析享元模式在Java Integer、String中的应用

王争 · 设计模式之美



上一节课，我们通过棋牌游戏和文本编辑器这样两个实际的例子，学习了享元模式的原理、实现以及应用场景。用一句话总结一下，享元模式中的“享元”指被共享的单元。享元模式通过复用对象，以达到节省内存的目的。


今天，我再用一节课的时间带你剖析一下，享元模式在 Java Integer、String 中的应用。如果你不熟悉 Java 编程语言，那也不用担心看不懂，因为今天的内容主要还是介绍设计思路，跟语言本身关系不大。

话不多说，让我们正式开始今天的学习吧！

享元模式在 Java Integer 中的应用

我们先来看下面这样一段代码。你可以先思考下，这段代码会输出什么样的结果。

```
1 Integer i1 = 56;
```

 复制代码

```
2 Integer i2 = 56;
3 Integer i3 = 129;
4 Integer i4 = 129;
5 System.out.println(i1 == i2);
6 System.out.println(i3 == i4);
```

如果不熟悉 Java 语言，你可能会觉得，i1 和 i2 值都是 56，i3 和 i4 值都是 129，i1 跟 i2 值相等，i3 跟 i4 值相等，所以输出结果应该是两个 true。这样的分析是不对的，主要还是因为你对 Java 语法不熟悉。要正确地分析上面的代码，我们需要弄清楚下面两个问题：

如何判定两个 Java 对象是否相等（也就代码中的“==”操作符的含义）？

什么是自动装箱（Autoboxing）和自动拆箱（Unboxing）？

在 [🔗 加餐一](#) 中，我们讲到，Java 为基本数据类型提供了对应的包装器类型。具体如下所示：

基本数据类型	对应的包装器类型
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
short	Short
byte	Byte
char	Character



所谓的自动装箱，就是自动将基本数据类型转换为包装器类型。所谓的自动拆箱，也就是自动将包装器类型转化为基本数据类型。具体的代码示例如下所示：

```
1 Integer i = 56; //自动装箱
2 int j = i; //自动拆箱
```

[复制代码](#)


数值 56 是基本数据类型 int，当赋值给包装器类型（Integer）变量的时候，触发自动装箱操作，创建一个 Integer 类型的对象，并且赋值给变量 i。其底层相当于执行了下面这条语句：

[复制代码](#)

```
1 Integer i = 59; 底层执行了: Integer i = Integer.valueOf(59);
```


反过来，当把包装器类型的变量 `i`，赋值给基本数据类型变量 `j` 的时候，触发自动拆箱操作，将 `i` 中的数据取出，赋值给 `j`。其底层相当于执行了下面这条语句：

```
1 int j = i; 底层执行了: int j = i.intValue();
```

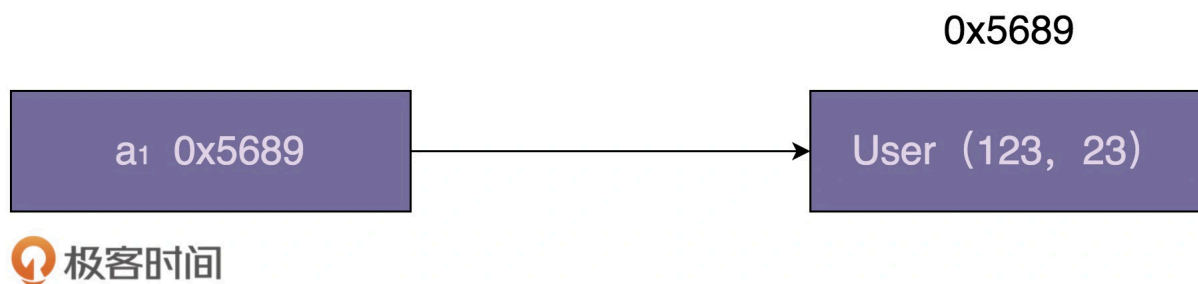
 复制代码

弄清楚了自动装箱和自动拆箱，我们再来看，如何判定两个对象是否相等？不过，在此之前，我们先要搞清楚，Java 对象在内存中是如何存储的。我们通过下面这个例子来说明一下。

```
1 User a = new User(123, 23); // id=123, age=23
```


 复制代码

针对这条语句，我画了一张内存存储结构图，如下所示。`a` 存储的值是 `User` 对象的内存地址，在图中就表现为 `a` 指向 `User` 对象。



当我们通过“`==`”来判定两个对象是否相等的时候，实际上是在判断两个局部变量存储的地址是否相同，换句话说，是在判断两个局部变量是否指向相同的对象。


了解了 Java 的这几个语法之后，我们重新看一下开头的那段代码。

 复制代码

```
1 Integer i1 = 56;
2 Integer i2 = 56;
3 Integer i3 = 129;
4 Integer i4 = 129;
5 System.out.println(i1 == i2);
6 System.out.println(i3 == i4);
```


前 4 行赋值语句都会触发自动装箱操作，也就是会创建 Integer 对象并且赋值给 i1、i2、i3、i4 这四个变量。根据刚刚的讲解，i1、i2 尽管存储的数值相同，都是 56，但是指向不同的 Integer 对象，所以通过“==”来判定是否相同的时候，会返回 false。同理，i3==i4 判定语句也会返回 false。

不过，上面的分析还是不对，答案并非是两个 false，而是一个 true，一个 false。看到这里，你可能会比较纳闷了。实际上，这正是因为 Integer 用到了享元模式来复用对象，才导致了这样的运行结果。当我们通过自动装箱，也就是调用 valueOf() 来创建 Integer 对象的时候，如果要创建的 Integer 对象的值在 -128 到 127 之间，会从 IntegerCache 类中直接返回，否则才调用 new 方法创建。看代码更加清晰一些，Integer 类的 valueOf() 函数的具体代码如下所示：

 复制代码

```
1 public static Integer valueOf(int i) {
2     if (i >= IntegerCache.low && i <= IntegerCache.high)
3         return IntegerCache.cache[i + (-IntegerCache.low)];
4     return new Integer(i);
5 }
```

实际上，这里的 IntegerCache 相当于，我们上一节课中讲的生成享元对象的工厂类，只不过名字不叫 xxxFactory 而已。我们来看它的具体代码实现。这个类是 Integer 的内部类，你也可以自行查看 JDK 源码。

 复制代码

```
1 /**
2  * Cache to support the object identity semantics of autoboxing for values between
3  * -128 and 127 (inclusive) as required by JLS.
4  *
5  * The cache is initialized on first usage. The size of the cache
```

```

6  * may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} option.
7  * During VM initialization, java.lang.Integer.IntegerCache.high property
8  * may be set and saved in the private system properties in the
9  * sun.misc.VM class.
10 */
11 private static class IntegerCache {
12     static final int low = -128;
13     static final int high;
14     static final Integer cache[];
15
16     static {
17         // high value may be configured by property
18         int h = 127;
19         String integerCacheHighPropValue =
20             sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
21         if (integerCacheHighPropValue != null) {
22             try {
23                 int i = parseInt(integerCacheHighPropValue);
24                 i = Math.max(i, 127);
25                 // Maximum array size is Integer.MAX_VALUE
26                 h = Math.min(i, Integer.MAX_VALUE - (-low) - 1);
27             } catch (NumberFormatException nfe) {
28                 // If the property cannot be parsed into an int, ignore it.
29             }
30         }
31         high = h;
32
33         cache = new Integer[(high - low) + 1];
34         int j = low;
35         for(int k = 0; k < cache.length; k++)
36             cache[k] = new Integer(j++);
37
38         // range [-128, 127] must be interned (JLS7 5.1.7)
39         assert IntegerCache.high >= 127;
40     }
41
42     private IntegerCache() {}
43 }


```

为什么 IntegerCache 只缓存 -128 到 127 之间的整型值呢？

在 IntegerCache 的代码实现中，当这个类被加载的时候，缓存的享元对象会被集中一次性创建好。毕竟整型值太多了，我们不可能在 IntegerCache 类中预先创建好所有的整型值，这样

既占用太多内存，也使得加载 IntegerCache 类的时间过长。所以，我们只能选择缓存对于大部分应用来说最常用的整型值，也就是一个字节的大小（-128 到 127 之间的数据）。


实际上，JDK 也提供了方法来让我们可以自定义缓存的最大值，有下面两种方式。如果你通过分析应用的 JVM 内存占用情况，发现 -128 到 255 之间的数据占用的内存比较多，你就可以用如下方式，将缓存的最大值从 127 调整到 255。不过，这里注意一下，JDK 并没有提供设置最小值的方法。

 复制代码

```
1 //方法一：
2 -Djava.lang.Integer.IntegerCache.high=255
3 //方法二：
4 -XX:AutoBoxCacheMax=255
```

现在，让我们再回到最开始的问题，因为 56 处于 -128 和 127 之间，i1 和 i2 会指向相同的享元对象，所以 i1==i2 返回 true。而 129 大于 127，并不会被缓存，每次都会创建一个全新的对象，也就是说，i3 和 i4 指向不同的 Integer 对象，所以 i3==i4 返回 false。

实际上，除了 Integer 类型之外，其他包装器类型，比如 Long、Short、Byte 等，也都利用了享元模式来缓存 -128 到 127 之间的数据。比如，Long 类型对应的 LongCache 享元工厂类及 valueOf() 函数代码如下所示：

 复制代码

```
1 private static class LongCache {
2     private LongCache(){}
3
4     static final Long cache[] = new Long[-(-128) + 127 + 1];
5
6     static {
7         for(int i = 0; i < cache.length; i++)
8             cache[i] = new Long(i - 128);
9     }
10 }
11
12 public static Long valueOf(long l) {
13     final int offset = 128;
14     if (l >= -128 && l <= 127) { // will cache
```

```
15         return LongCache.cache[(int)l + offset];
16     }
17     return new Long(l);
18 }
```

在我们平时的开发中，对于下面这样三种创建整型对象的方式，我们优先使用后两种。


 复制代码

```
1 Integer a = new Integer(123);
2 Integer a = 123;
3 Integer a = Integer.valueOf(123);
```

第一种创建方式并不会使用到 IntegerCache，而后面两种创建方法可以利用 IntegerCache 缓存，返回共享的对象，以达到节省内存的目的。举一个极端一点的例子，假设程序需要创建 1 万个 -128 到 127 之间的 Integer 对象。使用第一种创建方式，我们需要分配 1 万个 Integer 对象的内存空间；使用后两种创建方式，我们最多只需要分配 256 个 Integer 对象的内存空间。

享元模式在 Java String 中的应用

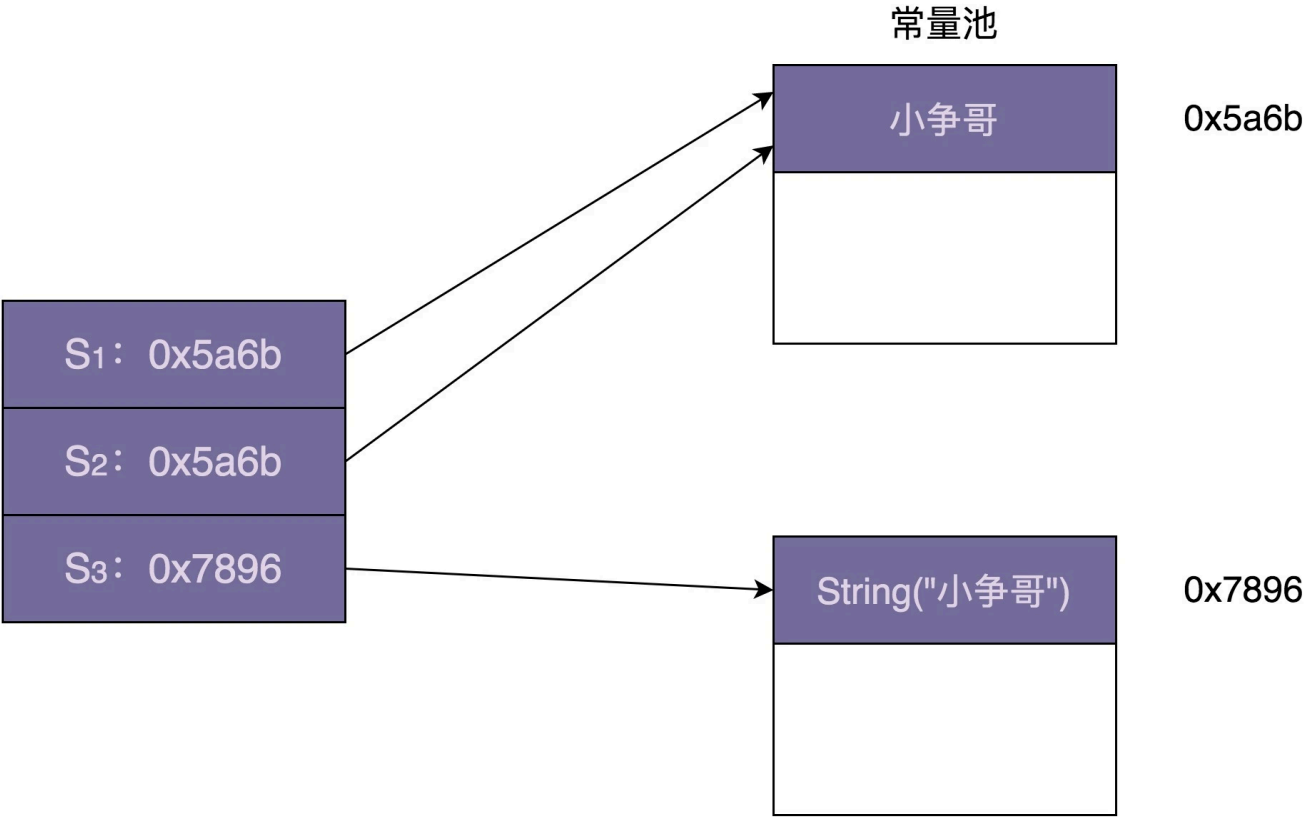
刚刚我们讲了享元模式在 Java Integer 类中的应用，现在，我们再来看下，享元模式在 Java String 类中的应用。同样，我们还是先来看一段代码，你觉得这段代码输出的结果是什么呢？

 复制代码

```
1 String s1 = "小争哥";
2 String s2 = "小争哥";
3 String s3 = new String("小争哥");
4
5 System.out.println(s1 == s2);
6 System.out.println(s1 == s3);
```

上面代码的运行结果是：一个 true，一个 false。跟 Integer 类的设计思路相似，String 类利用享元模式来复用相同的字符串常量（也就是代码中的“小争哥”）。JVM 会专门开辟一块存

储区来存储字符串常量，这块存储区叫作“字符串常量池”。上面代码对应的内存存储结构如下所示：



不过，String 类的享元模式的设计，跟 Integer 类稍微有些不同。Integer 类中要共享的对象，是在类加载的时候，就集中一次性创建好的。但是，对于字符串来说，我们没法事先知道要共享哪些字符串常量，所以没办法事先创建好，只能在某个字符串常量第一次被用到的时候，存储到常量池中，当之后再用到的时候，直接引用常量池中已经存在的即可，就不需要再重新创建了。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

在 Java Integer 的实现中，-128 到 127 之间的整型对象会被事先创建好，缓存在 IntegerCache 类中。当我们使用自动装箱或者 valueOf() 来创建这个数值区间的整型对象

时，会复用 IntegerCache 类事先创建好的对象。这里的 IntegerCache 类就是享元工厂类，事先创建好的整型对象就是享元对象。

在 Java String 类的实现中，JVM 开辟一块存储区专门存储字符串常量，这块存储区叫作字符串常量池，类似于 Integer 中的 IntegerCache。不过，跟 IntegerCache 不同的是，它并非事先创建好需要共享的对象，而是在程序的运行期间，根据需要来创建和缓存字符串常量。

除此之外，这里我再补充强调一下。

实际上，享元模式对 JVM 的垃圾回收并不友好。因为享元工厂类一直保存了对享元对象的引用，这就导致享元对象在没有任何代码使用的情况下，也不会被 JVM 垃圾回收机制自动回收掉。因此，在某些情况下，如果对象的生命周期很短，也不会被密集使用，利用享元模式反倒可能会浪费更多的内存。所以，除非经过线上验证，利用享元模式真的可以大大节省内存，否则，就不要过度使用这个模式，为了一点点内存的节省而引入一个复杂的设计模式，得不偿失啊。

课堂讨论

IntegerCache 只能缓存事先指定好的整型对象，那我们是否可以借鉴 String 的设计思路，不事先指定需要缓存哪些整型对象，而是在程序的运行过程中，当用到某个整型对象的时候，创建好放置到 IntegerCache，下次再被用到的时候，直接从 IntegerCache 中返回呢？

如果可以这么做，请你按照这个思路重新实现一下 IntegerCache 类，并且能够做到在某个对象没有任何代码使用的时候，能被 JVM 垃圾回收机制回收掉。

欢迎留言和我分享你的想法，如果有收获，欢迎你把这篇文章分享给你的朋友。

AI智能总结

Java中的享元模式在Integer和String类中的应用是非常重要的。通过对Java中的自动装箱和自动拆箱的理解，我们可以看到Integer类利用享元模式来缓存-128到127之间的整型值，以节省内存空间。这种机制使得在创建整型对象时，可以复用已存在的对象，而不是每次都创建新的对象，从而提高了内存利用率。String类也利用享元模式来复用相同的字符串常量，通过JVM开辟的字符串常量池来存储字符串常量，从而节省内存空间。然而，享元模式对JVM的垃圾回收并不友好，因为享元工厂类一直保存了对享元对象的引用，导致对象即使没有任何代码使用的情况下也不会被JVM垃圾回收机制自动回收掉。因此，在某些情况下，利用享元模式可能会浪费更多的内存。除非经过线上验证，利用享元模式真的可以大大节省内存，否则，就不要过度使用这个模式。文章还提出了一个思路，即在程序的运行过程中，当用到某个整型对象的时候，创建好放置到

IntegerCache，下次再被用到的时候，直接从IntegerCache中返回，并且能够做到在某个对象没有任何代码使用的时候，能被JVM垃圾回收机制回收掉。这篇文章通过具体的代码示例和内存存储结构图的解释，帮助读者理解了Java中享元模式的应用，适合Java开发者快速了解享元模式在Integer和String中的应用，以及如何在实际开发中充分利用这一特性来提升性能。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (48)

最新 精选



一丁、乙

2020-11-18

享元--->复用，线程池等。通过复用对象，以达到节省内存的目的

1.懒加载，dubble check

2.weak reference持有享元对象

作者回复: 嗯嗯



1



张三丰

2020-07-31

为什么说垃圾回收的时候如果保存了对象的"引用"就不友好，垃圾回收的依据不是只看这个对象还有没有被"使用"吗？

作者回复: 有引用，就是在被使用啊



Liam

2020-03-09

享元池用weak reference持有享元对象

共 7 条评论 >



75



小晏子

2020-03-09

如果IntegerCache不事先指定缓存哪些整形对象，那么每次用到的时候去new一个，这样会稍微影响一些效率，尤其在某些情况下如果常用到-128~127之间的数，可能会不停的new/delete，不过这个性能问题在大部分时候影响不是很大，所以按照string的设计思路也是可行的，按照这个思路设计IntegerCache类的话，如下

```
private static class IntegerCache {

    public static final WeakHashMap<Integer, WeakReference<Integer>> cache =
        new WeakHashMap<Integer, WeakReference<Integer>>(); //也可以提前分配容量

    private IntegerCache(){}
}

public static Integer valueOf(int i) {
    final WeakReference<Integer> cached = IntegerCache.cache.get(i);
    if (cached != null) {
        final Integer value = cached.get(i);
        if (value != null) {
            return value;
        }
    }
    WeakReference<Integer> val = new WeakReference<Integer>(i);
    IntegerCache.cache.put(i, val);
    return val.get();
}
```

共 7 条评论 >

👍 51



辣么大

2020-03-11

谢谢各位的讨论，今天学到了软引用，弱引用，和WeakHashMap。内存吃紧的时候可以考虑使用WeakHashMap。

<https://www.baeldung.com/java-weakhashmap>

<https://www.baeldung.com/java-soft-references>

<https://www.baeldung.com/java-weak-reference>

共 7 条评论 >

👍 47



李小四

2020-03-17

设计模式_55:

作业

原来还有个WeakHashMap，学习了。

感想

自己尝试了写了一个，然后分别测试了10,000次、100,000次，1,000,000次创建，value从1-100，100-200，10000-10100，发现不管哪个场景，总是JVM的Integer时间更短，我写的要3倍左右的时间，不禁感叹，Java二十几年了，大部分的优化应该都做了，不要期望自己花20分钟能改出超过JVM的性能。



👍 30



3Spiders

2020-03-09

课后题。因为整型对象长度固定，且内容固定，可以直接申请一块连续的内存地址，可以加快访问，节省内存？而String类不行。

共 1 条评论 >

👍 25



Geek_41d472

2020-03-10

我勒个擦，这好像是我碰到的两道面试题，包装和拆箱这道题简直就是个坑，有踩坑的举个手



👍 13



webmin

2020-03-09

抛砖引玉实现了一个有限范围的缓存（-128~2048383(127 * 127 * 127)）

```
public class IntegerCache {
    private static final int bucketSize = 127;
    private static final int level1Max = bucketSize * bucketSize;
    private static final int max = bucketSize * bucketSize * bucketSize;
    private static final WeakHashMap<Integer, WeakHashMap<Integer, WeakHashMap<Integer, WeakReference<Integer>>>> CACHE = new WeakHashMap<>();

    public static Integer intern(int integer) {
        if (integer <= 127) {
            return integer;
        }
    }
}
```

```

    }

    if (integer > max) {
        return integer;
    }

    synchronized (CACHE) {
        Integer l1 = 0;
        int tmp = integer;
        if(integer >= level1Max){
            l1 = integer / level1Max;
            integer -= level1Max;
        }
        Integer l2 = integer / bucketSize;
        Integer mod = integer % bucketSize;
        WeakHashMap<Integer, WeakHashMap<Integer,WeakReference<Integer>>> level1 = CACHE.computeIfAbsent(l1, val -> new WeakHashMap<>());
        WeakHashMap<Integer,WeakReference<Integer>> level2 = level1.computeIfAbsent(l2, val -> new WeakHashMap<>());
        WeakReference<Integer> cache = level2.computeIfAbsent(mod, val -> new WeakReference<>(tmp));
        Integer val = cache.get();
        if (val == null) {
            val = integer;
            level2.put(mod, new WeakReference<>(val));
        }
        return val;
    }

}

public static int integersInCache() {
    synchronized (CACHE) {
        int sum = CACHE.size();
        for (Integer key : CACHE.keySet()) {
            WeakHashMap<Integer, WeakHashMap<Integer,WeakReference<Integer>>> tmp = CACHE.get(key);
            sum += tmp.size();
        }
    }
}

```

```
        for(Integer l2Key : tmp.keySet()) {  
            sum += tmp.get(l2Key).size();  
        }  
    }  
    return sum;  
}  
}
```

共 1 条评论 >

 10



Eden Ma

2020-03-09

突然理解OC中NSString等也用到了享元设计模式.

共 2 条评论 >

 9