

28 | Immutability模式：如何利用不变性解决并发问题？

2019-05-02 王宝令

Java并发编程实战

[进入课程 >](#)



讲述：王宝令

时长 07:56 大小 7.27M



我们曾经说过，“多个线程同时读写同一共享变量存在并发问题”，这里的必要条件之一是读写，如果只有读，而没有写，是没有并发问题的。


解决并发问题，其实最简单的办法就是让共享变量只有读操作，而没有写操作。这个办法如此重要，以至于被上升到了一种解决并发问题的设计模式：**不变性 (Immutability) 模式**。所谓**不变性**，简单来讲，就是对象一旦被创建之后，状态就不再发生变化。换句话说，就是变量一旦被赋值，就不允许修改了（没有写操作）；没有修改操作，也就是保持了不变性。

快速实现具备不可变性的类

实现一个具备不可变性的类，还是挺简单的。**将一个类所有的属性都设置成 final 的，并且只允许存在只读方法，那么这个类基本上就具备不可变性了。**更严格的做法是**这个类本身也是 final 的**，也就是不允许继承。因为子类可以覆盖父类的方法，有可能改变不可变性，所以推荐你在实际工作中，使用这种更严格的做法。

Java SDK 里很多类都具备不可变性，只是由于它们的使用太简单，最后反而被忽略了。例如经常用到的 String 和 Long、Integer、Double 等基础类型的包装类都具备不可变性，这些对象的线程安全性都是靠不可变性来保证的。如果你仔细翻看这些类的声明、属性和方法，你会发现它们都严格遵守不可变类的三点要求：**类和属性都是 final 的，所有方法均是只读的。**

看到这里你可能会疑惑，Java 的 String 方法也有类似字符替换操作，怎么能说所有方法都是只读的呢？我们结合 String 的源代码来解释一下这个问题，下面的示例代码源自 Java 1.8 SDK，我略做了修改，仅保留了关键属性 value[] 和 replace() 方法，你会发现：String 这个类以及它的属性 value[] 都是 final 的；而 replace() 方法的实现，就的确没有修改 value[]，而是将替换后的字符串作为返回值返回了。

 复制代码

```
1 public final class String {
2     private final char value[];
3     // 字符替换
4     String replace(char oldChar,
5         char newChar) {
6         // 无需替换，直接返回 this
7         if (oldChar == newChar){
8             return this;
9         }
10
11         int len = value.length;
12         int i = -1;
13         /* avoid getfield opcode */
14         char[] val = value;
15         // 定位到需要替换的字符位置
16         while (++i < len) {
17             if (val[i] == oldChar) {
18                 break;
19             }
20         }
21         // 未找到 oldChar，无需替换
22         if (i >= len) {
23             return this;
24         }
25         // 创建一个 buf[]，这是关键
```

```
26     // 用来保存替换后的字符串
27     char buf[] = new char[len];
28     for (int j = 0; j < i; j++) {
29         buf[j] = val[j];
30     }
31     while (i < len) {
32         char c = val[i];
33         buf[i] = (c == oldChar) ?
34             newChar : c;
35         i++;
36     }
37     // 创建一个新的字符串返回
38     // 原字符串不会发生任何变化
39     return new String(buf, true);
40 }
41 }
```

通过分析 String 的实现，你可能已经发现了，如果具备不可变性的类，需要提供类似修改的功能，具体该怎么操作呢？做法很简单，那就是**创建一个新的不可变对象**，这是与可变对象的一个重要区别，可变对象往往是修改自己的属性。

所有的修改操作都创建一个新的不可变对象，你可能会会有这种担心：是不是创建的对象太多了，有点太浪费内存呢？是的，这样做的确有些浪费，那如何解决呢？

利用享元模式避免创建重复对象


如果你熟悉面向对象相关的设计模式，相信你一定能想到**享元模式（Flyweight Pattern）**。利用享元模式可以减少创建对象的数量，从而减少内存占用。Java 语言里面 Long、Integer、Short、Byte 等这些基本数据类型的包装类都用到了享元模式。

下面我们就以 Long 这个类作为例子，看看它是如何利用享元模式来优化对象的创建的。

享元模式本质上其实就是一个**对象池**，利用享元模式创建对象的逻辑也很简单：创建之前，首先去对象池里看看是不是存在；如果已经存在，就利用对象池里的对象；如果不存在，就会新建一个对象，并且把这个新建出来的对象放进对象池里。


Long 这个类并没有照搬享元模式，Long 内部维护了一个静态的对象池，仅缓存了 [-128,127] 之间的数字，这个对象池在 JVM 启动的时候就创建好了，而且这个对象池一直都不会变化，也就是说它是静态的。之所以采用这样的设计，是因为 Long 这个对象的状态

共有 2^{64} 种，实在太多，不宜全部缓存，而 $[-128,127]$ 之间的数字利用率最高。下面的示例代码出自 Java 1.8，`valueOf()` 方法就用到了 `LongCache` 这个缓存，你可以结合着来加深理解。

 复制代码

```
1 Long valueOf(long l) {
2     final int offset = 128;
3     // [-128,127] 直接的数字做了缓存
4     if (l >= -128 && l <= 127) {
5         return LongCache
6             .cache[(int)l + offset];
7     }
8     return new Long(l);
9 }
10 // 缓存，等价于对象池
11 // 仅缓存 [-128,127] 直接的数字
12 static class LongCache {
13     static final Long cache[]
14         = new Long[-(-128) + 127 + 1];
15
16     static {
17         for(int i=0; i<cache.length; i++)
18             cache[i] = new Long(i-128);
19     }
20 }
```

前面我们在[《13 | 理论基础模块热点问题答疑》](#)中提到“Integer 和 String 类型的对象不适合做锁”，其实基本上所有的基础类型的包装类都不适合做锁，因为它们内部用到了享元模式，这会导致看上去私有的锁，其实是共有的。例如在下面代码中，本意是 A 用锁 `al`，B 用锁 `bl`，各自管理各自的，互不影响。但实际上 `al` 和 `bl` 是一个对象，结果 A 和 B 共用的是一把锁。

 复制代码

```
1 class A {
2     Long al=Long.valueOf(1);
3     public void setAX(){
4         synchronized (al) {
5             // 省略代码无数
6         }
7     }
8 }
9 class B {
10     Long bl=Long.valueOf(1);
```


```
11 public void setBY(){
12     synchronized (b1) {
13         // 省略代码无数
14     }
15 }
16 }
```

使用 Immutability 模式的注意事项

在使用 Immutability 模式的时候，需要注意以下两点：

1. 对象的所有属性都是 final 的，并不能保证不可变性；
2. 不可变对象也需要正确发布。

在 Java 语言中，final 修饰的属性一旦被赋值，就不可再修改，但是如果属性的类型是普通对象，那么这个普通对象的属性是可以被修改的。例如下面的代码中，Bar 的属性 foo 虽然是 final 的，依然可以通过 setAge() 方法来设置 foo 的属性 age。所以，**在使用 Immutability 模式的时候一定要确认保持不变性的边界在哪里，是否要求属性对象也具备不可变性。**

 复制代码

```
1 class Foo{
2     int age=0;
3     int name="abc";
4 }
5 final class Bar {
6     final Foo foo;
7     void setAge(int a){
8         foo.age=a;
9     }
10 }
```

下面我们再看看如何正确地发布不可变对象。不可变对象虽然是线程安全的，但是并不意味着引用这些不可变对象的对象就是线程安全的。例如在下面的代码中，Foo 具备不可变性，线程安全，但是类 Bar 并不是线程安全的，类 Bar 中持有对 Foo 的引用 foo，对 foo 这个引用的修改在多线程中并不能保证可见性和原子性。

```
1 //Foo 线程安全
2 final class Foo{
3     final int age=0;
4     final int name="abc";
5 }
6 //Bar 线程不安全
7 class Bar {
8     Foo foo;
9     void setFoo(Foo f){
10         this.foo=f;
11     }
12 }
```

如果你的程序仅仅需要 foo 保持可见性，无需保证原子性，那么可以将 foo 声明为 volatile 变量，这样就能保证可见性。如果你的程序需要保证原子性，那么可以通过原子类来实现。下面的示例代码是合理库存的原子化实现，你应该很熟悉了，其中就是用原子类解决了不可变对象引用的原子性问题。

```
1 public class SafewM {
2     class WMRRange{
3         final int upper;
4         final int lower;
5         WMRRange(int upper,int lower){
6             // 省略构造函数实现
7         }
8     }
9     final AtomicReference<WMRRange>
10     rf = new AtomicReference<>(<
11         new WMRRange(0,0)
12     );
13     // 设置库存上限
14     void setUpper(int v){
15         while(true){
16             WMRRange or = rf.get();
17             // 检查参数合法性
18             if(v < or.lower){
19                 throw new IllegalArgumentException();
20             }
21             WMRRange nr = new
22                 WMRRange(v, or.lower);
23             if(rf.compareAndSet(or, nr)){
24                 return;
25             }
26         }
27     }
28 }
```



```
27     }
28 }
```


总结

利用 Immutability 模式解决并发问题，也许你觉得有点陌生，其实你天天都在享受它的战果。Java 语言里面的 String 和 Long、Integer、Double 等基础类型的包装类都具备不可变性，这些对象的线程安全性都是靠不可变性来保证的。Immutability 模式是最简单的解决并发问题的方法，建议当你试图解决一个并发问题时，可以首先尝试一下 Immutability 模式，看是否能够快速解决。

具备不变性的对象，只有一种状态，这个状态由对象内部所有的不变属性共同决定。其实还有一种更简单的不变性对象，那就是**无状态**。无状态对象内部没有属性，只有方法。除了无状态的对象，你可能还听说过无状态的服务、无状态的协议等等。无状态有很多好处，最核心的一点就是性能。在多线程领域，无状态对象没有线程安全问题，无需同步处理，自然性能很好；在分布式领域，无状态意味着可以无限地水平扩展，所以分布式领域里面性能的瓶颈一定不是出在无状态的服务节点上。

课后思考

下面的示例代码中，Account 的属性是 final 的，并且只有 get 方法，那这个类是不是具备不可变性呢？

 复制代码

```
1 public final class Account{
2     private final
3         StringBuffer user;
4     public Account(String user){
5         this.user =
6             new StringBuffer(user);
7     }
8
9     public StringBuffer getUser(){
10         return this.user;
11     }
12     public String toString(){
13         return "user"+user;
14     }
15 }
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。



Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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。

上一篇 27 | 并发工具类模块热点问题答疑

下一篇 29 | Copy-on-Write模式：不是延时策略的COW

精选留言 (21)

写留言



Jialin

2019-05-02

18

根据文章内容,一个类具备不可变属性需要满足"类和属性都必须是 final 的,所有方法均是只读的",类的属性如果是引用型,该属性对应的类也需要满足不可变类的条件,且不能提供修改该属性的方法,

Account类的唯一属性user是final的,提供的方法是可读的,user的类型是StringBuffer,StringBuffer也是final的,这样看来,Account类是不可变性的,但是去看...

展开



张天屹

2019-05-05

👍 4

具不具备不可变性看怎么界定边界了，类本身是具备的，StringBuffer的引用不可变。但是因为StringBuffer是一个对象，持有非final的char数组，所以底层数组是可变的。但是StringBuffer是并发安全的，因为方法加锁synchronized



摇山樵客™

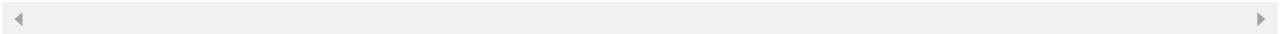
2019-05-05

👍 4

这段代码应该是线程安全的，但它不是不可变模式。StringBuffer只是字段引用不可变，值是可以调用StringBuffer的方法改变的，这个需要改成把字段改成String这样的不可变对象来解决。

展开 ▾

作者回复: 👍



对象正在输...

2019-05-05

👍 2

不可变类的三个要求：类和属性都是 final 的，所有方法均是只读的
这里的StringBuffer传进来的只是个引用，调用方可以修改，所以这个类不具备不可变性。



Hour

2019-06-01

👍 1

//Foo 线程安全

```
final class Foo{  
    final int age=0;  
    final int name="abc";  
}...
```

展开 ▾



炎炎

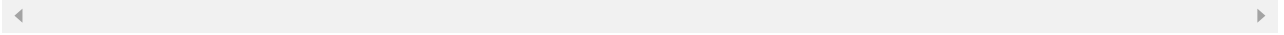
2019-05-24

👍 1

这个专栏一直看到这儿，真的很棒，课后问题也很好，让我对并发编程有了一个整体的了解，之前看书一直看不懂，老师带着梳理一遍，看书也容易多了，非常感谢老师，希望老师再出专栏

展开 ∨

作者回复: 感谢一路相伴 😊



rayjun

2019-05-05

👍 1

不是不可变的，user 逃逸了

展开 ∨



陈华应

2019-05-03

👍 1

不具备，stringbuffer本身线程不安全

展开 ∨



xuery

2019-05-31

👍

不是，通过getUser拿到StringBuffer类型的user后，还是可以通过append改变字符串



炎炎

2019-05-24

👍

想请教老师一个问题，Long里面的内部类为什么不用final修饰，这样这个内部类不就可以被继承修改了么？怎么保证它的不可变性呢？

// 缓存，等价于对象池

// 仅缓存 [-128,127] 直接的数字

static class LongCache {...

展开 ∨



Zach_

2019-05-13

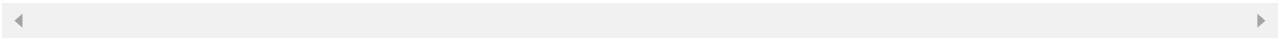
👍

final StringBuffer user;

StringBuffer 是 引用 类型，当我们说它final StringBuffer user 不可变时，实际上说的是它user指向堆内存的地址不可变，但堆内存的user对象，通过sub append 方法实际是可变的.....

展开 ▾

作者回复: 📬



易儿易

2019-05-07



思考题：不是不可变类，用下边的代码可以进行验证！（返回的对象自身提供了修改方法）

```
public final class Test {  
    public static void main(String[] args) {  
        Account a = new Account("小A");...
```

展开 ▾



肖魁

2019-05-05



虽然没有对外提供修改user的方法，但是提供了get方法返回user可以修改

展开 ▾



松花皮蛋me

2019-05-03



Stringbuffer虽然逃出来了，但是没有引用其他对象，另外它本身也是线程安全的，所以具有不可变性



老醋

2019-05-03



我的理解是：

不具有不可变性，因为get方法返回的是user对象的引用，不是一个拷贝，所以可以改变Account类的user对象。

展开 ▾



张三

2019-05-03



打卡。

展开 ▾



QQ怪

2019-05-02



不具备不可变性,原因是stringbuffer类存在更改user对象方法

展开 ▾



发条橙子 ...

2019-05-02



老师五一节日快乐。

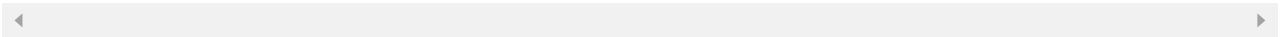
思考题：

不可变类的三要素：类、属性、方法都是不可变的。思考题这个类虽然是final，属性也是final并且没有修改的方法，但是stringbuffer这个属性的内容是可变的，所以应该...

展开 ▾

作者回复: 感谢感谢 😊

你的问题有点笼统，jdk也不是没有bug, sync的锁是记在对象头里的



MiracleCM

2019-05-02



我认为不是线程安全的 因为得到的stringbuffer 提供了方法改变user。请老师指点。



木木匠

2019-05-02



Account 的属性是 final 的，而且只有get方法，从这里考虑，确实觉得这个类不可变，但是这个类的引用指向user对象，这个对象的结构没有说明，那么就有可能这个对象里面的属性可变，所以就会导致虽然Account 不可变，但是User属性是可变的。所以这个类不具有不可变性。

