

28 | Immutability模式：如何利用不变性解决并发问题？

2019-05-02 王宝令



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，以至于被上升到了的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。

```
public final class String {
    private final char value[];

    // 字符替换

    String replace(char oldChar,
        char newChar) {
        //无需替换，直接返回this
        if (oldChar == newChar){
            return this;
        }

        int len = value.length;
        int i = -1;
        /* avoid getfield opcode */
        char[] val = value;
        //定位到需要替换的字符位置
        while (++i < len) {
            if (val[i] == oldChar) {
                break;
            }
        }
        //未找到oldChar，无需替换
        if (i >= len) {
            return this;
        }
        //创建一个buf[]，这是关键
        //用来保存替换后的字符串
        char buf[] = new char[len];
        for (int j = 0; j < i; j++) {
            buf[j] = val[j];
        }
        while (i < len) {
            char c = val[i];
```

```
    buf[i] = (c == oldChar) ?  
        newChar : c;  
    i++;  
}  
//创建一个新的字符串返回  
//原字符串不会发生任何变化  
return new String(buf, true);  
}  
}
```

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。

```

Long valueOf(long l) {
    final int offset = 128;
    // [-128,127]直接的数字做了缓存
    if (l >= -128 && l <= 127) {
        return LongCache
            .cache[(int)l + offset];
    }
    return new Long(l);
}

//缓存，等价于对象池
//仅缓存[-128,127]直接的数字
static class LongCache {
    static final Long cache[]
        = new Long[-(-128) + 127 + 1];

    static {
        for(int i=0; i<cache.length; i++)
            cache[i] = new Long(i-128);
    }
}

```

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。

```
class A {  
    Long al=Long.valueOf(1);  
    public void setAX(){  
        synchronized (al) {  
            //省略代码无数  
        }  
    }  
}  
  
class B {  
    Long bl=Long.valueOf(1);  
    public void setBY(){  
        synchronized (bl) {  
            //省略代码无数  
        }  
    }  
}
```

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。

```

class Foo{
    int age=0;
    int name="abc";
}
final class Bar {
    final Foo foo;
    void setAge(int a){
        foo.age=a;
    }
}

```

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。

```

//Foo线程安全
final class Foo{
    final int age=0;
    final int name="abc";
}
//Bar线程不安全
class Bar {
    Foo foo;
    void setFoo(Foo f){
        this.foo=f;
    }
}

```

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。

```

public class SafeWM {
    class WMRange{
        final int upper;
        final int lower;
        WMRange(int upper,int lower){
            //省略构造函数实现
        }
    }
    final AtomicReference<WMRange>
        rf = new AtomicReference<> (
            new WMRange(0,0)
        );
    // 设置库存上限
    void setUpper(int v){
        while(true){
            WMRange or = rf.get();
            // 检查参数合法性
            if(v < or.lower){
                throw new IllegalArgumentException();
            }
            WMRange nr = new
                WMRange(v, or.lower);
            if(rf.compareAndSet(or, nr)){
                return;
            }
        }
    }
}

```

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？

```
public final class Account{  
    private final  
        StringBuffer user;  
    public Account(String user){  
        this.user =  
            new StringBuffer(user);  
    }  
  
    public StringBuffer getUser(){  
        return this.user;  
    }  
    public String toString(){  
        return "user"+user;  
    }  
}
```

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。

精选留言



Jialin

👍 20

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**StringBuffer**的源码,你会发现**StringBuffer**类的属性**value**是可变的<**String**类中的**value**定义:`private final char value[]`;**StringBuffer**类中的**value**定义:`char[] value`>,并且提供了**append(Object object)**和**setCharAt(int index, char ch)**修改**value**.所以,**Account**类不具备不可变性

2019-05-02



摇山樵客™

👍 5

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。

2019-05-05

作者回复

👍

2019-05-20



张天屹

👍 4

具不具备不可变性看怎么界定边界了,类本身是具备的,**StrnigBuffer**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**

2019-05-05



对象正在输入...

👍 3

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。

2019-05-05



Hour

👍 1

//Foo 线程安全

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

//Bar 线程不安全

```
class Bar {  
    Foo foo;  
    void setFoo(Foo f){  
        this.foo=f;  
    }  
}
```

老师好，对**foo**的引用和修改在多线程环境中并不能保证原子性和可见性，这句话怎么理解，能用具体的例子说明一下吗？

2019-06-01



炎炎

👍 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

2019-05-24

作者回复

感谢一路相伴🙏

2019-05-24



rayjun

👍 1

不是不可变的，**user** 逃逸了

2019-05-05



陈华应

👍 1

不具备，**stringbuffer**本身线程不安全

2019-05-03



发条橙子。

👍 1

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，所以应该没有满足三要素中的属性不可变，应该不属于不可变类。

另外老师我有个问题想问下，我看jdk一些源码里，也用了对象做锁。例如我有个变量`final ConcurrentHashMap cache`，有些方法中会对`cache`变量`put`新的值，但是还有用这个对象做`synchronized(cache)`对象锁，这种做法对么？如果对话的话，是因为管程只判断对象的首地址没有改变的原因么，希望老师指点一下

2019-05-02

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

2019-05-20



Jxin

👍 0

我们web开发的service层就是一种不可变模式的写法。所以没有并发问题。

2019-06-15



嗨喽

👍 0

上面得SafeWM类代码会不会有ABA问题呢，老师

2019-06-13

作者回复

版本号会一直增加，所以不会有aba问题

2019-06-13



xuery

👍 0

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

2019-05-31



炎炎

👍 0

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 {
```

```
    static final Long cache[]
```

```
    = new Long[(-128) + 127 + 1];
```

```
static {
for(int i=0; i<cache.length; i++)
cache[i] = new Long(i-128);
}
}
```

2019-05-24



Zach_

👍 0

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.....

2019-05-13

作者回复

👍

2019-05-13



易儿易

👍 0

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");
```

```
System.out.println(a.getUser());
```

```
a.getUser().append("小B");
```

```
System.out.println(a.getUser());
}
}
```

2019-05-07



肖魁

👍 0

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

2019-05-05



松花皮蛋me

👍 0

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

2019-05-03



老醋

👍 0

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。

2019-05-03



张三
打卡。

👍 0

2019-05-03



QQ怪

不具备不可变性,原因是`stringbuffer`类存在更改`user`对象方法

👍 0

2019-05-02