**Immutability模式：如何利用不变性解决并发问题？**[](" \l "immutability" \o "Permanent link)

* 快速实现具备不可变性的类
* 利用享元模式避免创建重复对象
* 使用Immutability模式的注意事项
* 总结

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恭喜那个变量只有读操作，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，简单来说，就是对象一旦被创建之后，状态就不再发生变化。** 欢聚话说，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**[](" \l "_2" \o "Permanent link)

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，并且只允许存在只读方法，那么这个类基本上就具备了不可变性了**。更严格的做法就是 **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，所以推荐你再实际工作中，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**){**

**if(**newChar **==** oldChar**){return** **this;}**

**int** len **=** value**.**length**;**

**int** i **=** **-**1**;**

**char[]** val **=** value**;**

*//定位到需要替换的字符串*

**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**，这时与可变对象的重要区别，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**[](" \l "_3" \o "Permanent link)

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，就会新创建一个对象，并且把这个新创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的64次方中，实在是太多，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，

*//利用享元模式就能减少对象的创建*

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*

*//这个对象在JVM启动的时候就创建好了*

**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**);**

**}**

**}**

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本类型都不适合做锁，因为他们内部都用到了享元模式，这会导致看上去私有的锁，其实是共有的。例如下面的代码中，本意是A用到锁a1，B用到锁b1，各种管理各自的，互不影响。但实际上a1和b1是一个对象，结果A和B共用的是一把锁。

**public** **class** **ATest** **{**

Long a1 **=** Long**.**valueOf**(**1**);**

**public** **void** **setAX()** **throws** InterruptedException **{**

*//synchronized (ATest.class){*

**synchronized** **(**a1**){**

**for(int** i**=**0**;**i**<**100**;**i**++){**

Thread**.**sleep**(**1000**);**

System**.**out**.**println**(**"a: "**+**i**++);**

**}**

**}**

**}**

**}**

**public** **class** **BTest** **{**

Long b1 **=** Long**.**valueOf**(**1**);**

**public** **void** **setAX()** **throws** InterruptedException **{**

*//synchronized (BTest.class){*

**synchronized** **(**b1**){**

**for(int** i**=**0**;**i**<**100**;**i**++){**

Thread**.**sleep**(**1000**);**

System**.**out**.**println**(**"b: "**+**i**++);**

**}**

**}**

**}**

**}**

**public** **class** **FlyWeight** **{**

**public** **static** **void** **main(**String**[]** args**)** **throws** InterruptedException **{**

ATest aTest **=** **new** ATest**();**

BTest bTest **=** **new** BTest**();**

**new** Thread**(new** Runnable**()** **{**

@Override

**public** **void** **run()** **{**

**try** **{**

aTest**.**setAX**();**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

**}**

**}).**start**();**

**new** Thread**(new** Runnable**()** **{**

@Override

**public** **void** **run()** **{**

**try** **{**

bTest**.**setAX**();**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

**}**

**}).**start**();**

**}**

**}**

a: 0

a: 2

a: 4

a: 6

a: 8

结果是ab顺序执行的，因为共用的是一把锁。

**使用Immutability模式的注意事项**[](" \l "immutability_1" \o "Permanent link)

在使用 Immutability 模式的时候，需要注意以下两点：

* 对象的所有属性都是 final 的，并不能保证不可变性；
* 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，就不可以再修改，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**[](" \l "_4" \o "Permanent link)

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**;**

**}**

**}**

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。

StingBuffer 是 引用 类型， 当我们说它final StingBuffer 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……