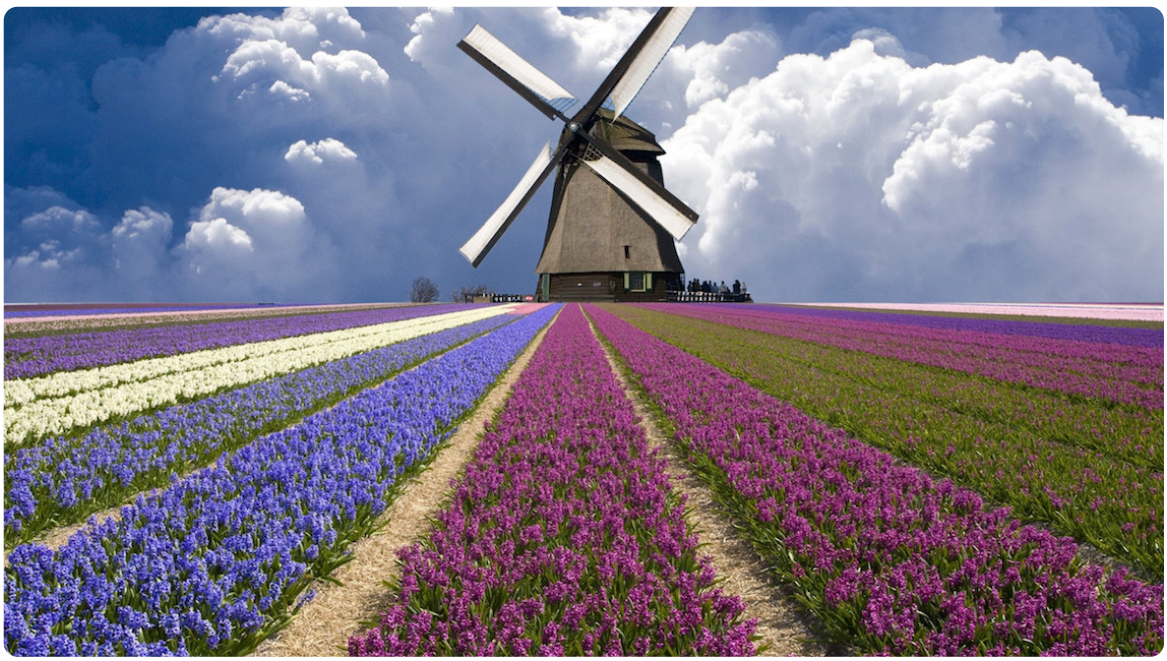
12 | 如何用面向对象思想写好并发程序?

王宝令 2019-03-26



00:00 10:10

讲述：王宝令 大小：9.32M

在工作中，我发现很多同学在设计之初都是直接按照单线程的思路来写程序的，而忽略了本应该重视的并发问题；等上线后的某天，突然发现诡异的 Bug，再历经千辛万苦终于定位到问题所在，却发现对于如何解决已经没有了思路。

关于这个问题，我觉得咱们今天很有必要好好聊聊“如何用面向对象思想写好并发程序”这个话题。

面向对象思想与并发编程有关系吗？本来是没关系的，它们分属两个不同的领域，但是在 Java 语言里，这两个领域被无情地融合在一起了，好在融合的效果还是不错的：**在 Java 语言里，面向对象思想能够让并发编程变得更简单。**

那如何才能用面向对象思想写好并发程序呢？结合我自己的工作经验来看，我觉得你可以从封装共享变量、识别共享变量间的约束条件和制定并发访问策略这三个方面下手。

一、封装共享变量

并发程序，我们关注的一个核心问题，不过是解决多线程同时访问共享变量的问题。在《03 | 互斥锁（上）：解决原子性问题》中，我们类比过球场门票的管理，现实世界里门票管理的一个核心问题是：所有观众只能通过规定的入口进入，否则检票就形同虚设。在编程世界这个问题也很

重要，编程领域里面对于共享变量的访问路径就类似于球场的入口，必须严格控制。好在有了面向对象思想，对共享变量的访问路径可以轻松把控。

面向对象思想里面有一个很重要的特性是**封装**，封装的通俗解释就是**将属性和实现细节封装在对象内部**，外界对象**只能通过**目标对象提供的**公共方法来间接访问**这些内部属性，这和门票管理模型匹配度相当的高，球场里的座位就是对象属性，球场入口就是对象的公共方法。我们把共享变量作为对象的属性，那对于共享变量的访问路径就是对象的公共方法，所有入口都要安排检票程序就相当于我们前面提到的并发访问策略。

利用面向对象思想写并发程序的思路，其实就这么简单：**将共享变量作为对象属性封装在内部，对所有公共方法制定并发访问策略**。就拿很多统计程序都要用到计数器来说，下面的计数器程序共享变量只有一个，就是 value，我们把它作为 Counter 类的属性，并且将两个公共方法 get() 和 addOne() 声明为同步方法，这样 Counter 类就成为一个线程安全的类了。

复制代码

```
public class Counter {
  private long value;
  synchronized long get(){
    return value;
  }
  synchronized long addOne(){
    return ++value;
  }
}

```

当然，实际工作中，很多的场景都不会像计数器这么简单，经常要面临的情况往往是有很多的共享变量，例如，信用卡账户有卡号、姓名、身份证、信用额度、已出账单、未出账单等很多共享变量。这么多的共享变量，如果每一个都考虑它的并发安全问题，那我们就累死了。但其实仔细观察，你会发现，很多共享变量的值是不会变的，例如信用卡账户的卡号、姓名、身份证。**对于这些不会发生变化的共享变量，建议你用 final 关键字来修饰。**这样既能避免并发问题，也能很明了地表明你的设计意图，让后面接手你程序的兄弟知道，你已经考虑过这些共享变量的并发安全问题了。

二、识别共享变量间的约束条件

识别共享变量间的约束条件非常重要。因为**这些约束条件，决定了并发访问策略**。例如，库存管理里面有个合理库存的概念，库存量不能太高，也不能太低，它有一个上限和一个下限。关于这些约束条件，我们可以用下面的程序来模拟一下。在类 SafeWM 中，声明了两个成员变量 upper 和 lower，分别代表库存上限和库存下限，这两个变量用了 AtomicLong 这个原子类，原子类是线程安全的，所以这两个成员变量的 set 方法就不需要同步了。

复制代码

```
public class SafeWM {
  // 库存上限
  private final AtomicLong upper =
        new AtomicLong(0);
  // 库存下限
  private final AtomicLong lower =
```

```
      new AtomicLong(0);
  // 设置库存上限
  void setUpper(long v){
    upper.set(v);
  }
  // 设置库存下限
  void setLower(long v){
    lower.set(v);
  }
  // 省略其他业务代码
}

```

虽说上面的代码是没有问题的，但是忽视了一个约束条件，就是**库存下限要小于库存上限**，这个约束条件能够直接加到上面的 set 方法上吗？我们先直接加一下看看效果（如下面代码所示）。我们在 setUpper() 和 setLower() 中增加了参数校验，这乍看上去好像是对的，但其实存在并发问题，问题在于存在竞态条件。这里我顺便插一句，其实当你看到代码里出现 if 语句的时候，就应该立刻意识到可能存在竞态条件。

我们假设库存的下限和上限分别是 (2,10)，线程 A 调用 setUpper(5) 将上限设置为 5，线程 B 调用 setLower(7) 将下限设置为 7，如果线程 A 和线程 B 完全同时执行，你会发现线程 A 能够通过参数校验，因为这个时候，下限还没有被线程 B 设置，还是 2，而 5>2；线程 B 也能够通过参数校验，因为这个时候，上限还没有被线程 A 设置，还是 10，而 7<10。当线程 A 和线程 B 都通过参数校验后，就把库存的下限和上限设置成 (7, 5) 了，显然此时的结果是不符合**库存下限要小于库存上限**这个约束条件的。

复制代码

```
public class SafeWM {
  // 库存上限
  private final AtomicLong upper =
        new AtomicLong(0);
  // 库存下限
  private final AtomicLong lower =
        new AtomicLong(0);
  // 设置库存上限
  void setUpper(long v){
    // 检查参数合法性
    if (v < lower.get()) {
      throw new IllegalArgumentException();
    }
    upper.set(v);
  }
  // 设置库存下限
  void setLower(long v){
    // 检查参数合法性
    if (v > upper.get()) {
      throw new IllegalArgumentException();
    }
    lower.set(v);
  }
  // 省略其他业务代码
}

```

在没有识别出**库存下限要小于库存上限**这个约束条件之前，我们制定的并发访问策略是利用原子类，但是这个策略，完全不能保证**库存下限要小于库存上限**这个约束条件。所以说，在设计阶段，我们**一定要识别出所有共享变量之间的约束条件，如果约束条件识别不足，很可能导致制定的并发访问策略南辕北辙**。

共享变量之间的约束条件，反映在代码里，基本上都会有 if 语句，所以，一定要特别注意竞态条件。

三、制定并发访问策略

制定并发访问策略，是一个非常复杂的事情。应该说整个专栏都是在尝试搞定它。不过从方案上来看，无外乎就是以下“三件事”。

1. 避免共享：避免共享的技术主要是利于线程本地存储以及为每个任务分配独立的线程。
2. 不变模式：这个在 Java 领域应用的很少，但在其他领域却有着广泛的应用，例如 Actor 模式、CSP 模式以及函数式编程的基础都是不变模式。
3. 管程及其他同步工具：Java 领域万能的解决方案是管程，但是对于很多特定场景，使用 Java 并发包提供的读写锁、并发容器等同步工具会更好。

接下来在咱们专栏的第二模块我会仔细讲解 Java 并发工具类以及他们的应用场景，在第三模块我还会讲解并发编程的设计模式，这些都是和制定并发访问策略有关的。

除了这些方案之外，还有一些宏观的原则需要你了解。这些宏观原则，有助于你写出“健壮”的并发程序。这些原则主要有以下三条。

1. 优先使用成熟的工具类：Java SDK 并发包里提供了丰富的工具类，基本上能满足你日常的需要，建议你熟悉它们，用好它们，而不是自己再“发明轮子”，毕竟并发工具类不是随随便便就能发明成功的。
2. 迫不得已时才使用低级的同步原语：低级的同步原语主要指的是 synchronized、Lock、Semaphore 等，这些虽然感觉简单，但实际上并没那么简单，一定要小心使用。
3. 避免过早优化：安全第一，并发程序首先要保证安全，出现性能瓶颈后再优化。在设计期和开发期，很多人经常会情不自禁地预估性能的瓶颈，并对此实施优化，但残酷的现实却是：性能瓶颈不是你想预估就能预估的。

总结

利用面向对象思想编写并发程序，一个关键点就是利用面向对象里的封装特性，由于篇幅原因，这里我只做了简单介绍，详细的你可以借助相关资料定向学习。而对共享变量进行封装，要避免“逸出”，所谓“逸出”简单讲就是共享变量逃逸到对象的外面，比如在《02 | Java 内存模型：看 Java 如何解决可见性和有序性问题》那一篇我们已经讲过构造函数里的 this“逸出”。这些都是必须要避免的。

这是我们专栏并发理论基础的最后一部分内容，这一部分内容主要是让你对并发编程有一个全面的认识，让你了解并发编程里的各种概念，以及它们之间的关系，当然终极目标是让你知道遇到并发问题该怎么思考。这部分的内容还是有点烧脑的，但专栏后面几个模块的内容都是具体的实践部分，相对来说就容易多了。我们一起坚持吧！

课后思考

本期示例代码中，类 SafeWM 不满足库存下限要小于库存上限这个约束条件，那你来试试修改一下，让它能够在并发条件下满足库存下限要小于库存上限这个约束条件。

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。

延伸阅读

关于这部分的内容，如果你觉得还不“过瘾”，这里我再给你推荐一本书吧——《Java 并发编程实战》，这本书的第三章《对象的共享》、第四章《对象的组合》全面地介绍了如何构建线程安全的对象，你可以拿来深入地学习。

猜你喜欢

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表 0/2000字 提交留言

精选留言(16)

无庸

compareAndSet吧

```
void setUpper（long v）{
  upper.compareAndSet(upper.longValue
(),v);
}
```

2 2019-03-26

抽离の♡

老师、讲的真好！

1 2019-03-26

京

一早起来，就把文章看完了，期待老师后面更精彩的内容

1 2019-03-26

undifined

要保证变量间的约束条件，就必须保证判断和赋值是一个原子操作，可以通过给 upper 和 lower 同时加锁，也可以通过 AtomicLong 提供的方法进行操作

```
// 设置库存上限
void setUpper(long v) {
  // 检查参数合法性
  upper.getAndUpdate(u -> {
    if (v < lower.get()) {
      throw new IllegalArgumentException();
    } else {
      return v;
    }
  });
}

// 设置库存下限
void setLower(long v) {
  // 检查参数合法性
  lower.getAndUpdate(u -> {
    if (v > upper.get()) {
      throw new IllegalArgumentException();
    } else {
      return v;
    }
  });
}
```

老师这样理解对吗

2019-03-26

Zach_

比起 用同步synchronized 用锁Lock 屏蔽可见性使用volatile ， 使用 cas操作好像更简洁！

2019-03-26

木木匠

思考题:对于两个互相比较的变量来说，赋值的时候只能加锁来控制。但是这也会带来性能问题，不过可以采用读锁和写锁来优化，申请写锁了就互斥，读锁可以并发访问，这样性能相对粗粒度的锁来说会高

点。

2019-03-26

Vickygu

是不是可以设置一个静态锁，来锁住这两个资源：public class SafeWM {

```
 private final AtomicLong upper = new AtomicLong(0);
 private final AtomicLong lower =new AtomicLong(0);
 //静态锁
 private static Object lock = new Object();


 void setUpper(long v){
   sychronized(lock){
      upper.set(v);
   }
 }

 void setLower(long v){
   sychronized(lock){
     lower.set(v);
     }
 }

}
```

2019-03-26

Kid☺

1.修改变量声明，利用transient关键字修改上下限变量；
2.用synchronize修饰设置方法；
老师，这两种方法是否可行?

2019-03-26

轻歌赋

第一种方案加锁
第二种方案，两个变量只能通过一个方法set，如果只修改其中一个，则另一个为Null。
两个属性额外定义一个类，属性final。当修改的时候只能额外创建新的类，修改引用。
对于并发替换类的引用问题，可以通过原子引用，返回boolean判断。具体不成功之后的策略根据实际情况来。
希望老师同学能够给出宝贵的意见评价一下方案的优劣，谢谢!

2019-03-26

handylin

可以说说java对象之间是怎么考虑共享的吗， 必须要一个线程对应一个对象吗?

2019-03-26

高源

王老师我能想到的就是将变量v 前加上volatile,原子性控制解决

2019-03-26

IamNigel

竞态条件要实现同步

2019-03-26

张建磊

文中问题可以把if条件和set方法放入synchronized（class）中，这样会不会影响性能。

2019-03-26

WL

老师如果像以下这么写有什么问题没?

```
@Data
public class SafeWM {

    private final AtomicLong upper = new AtomicLong(0);

    private final AtomicLong lower = new AtomicLong(0);

    public synchronized void setUpper(long v){

        synchronized (this.lower){
            if (this.getLower().get() > v) {
                throw new IllegalArgumentException();
            }
        }
        upper.set(v);
    }

    public synchronized void setLower(long v) {

        synchronized (this.upper) {
            if (this.getUpper().get() < v) {
                throw new IllegalArgumentException();
            }
            lower.set(v);
        }
    }
}
```

2019-03-26

刘章周

简单粗暴方法：在两个if语句外层加上synchronize(this){...}同步块。

其他的方法没想到。

2019-03-26

邋遢的流浪剑客

思考题可以用等待唤醒机制实现

2019-03-26