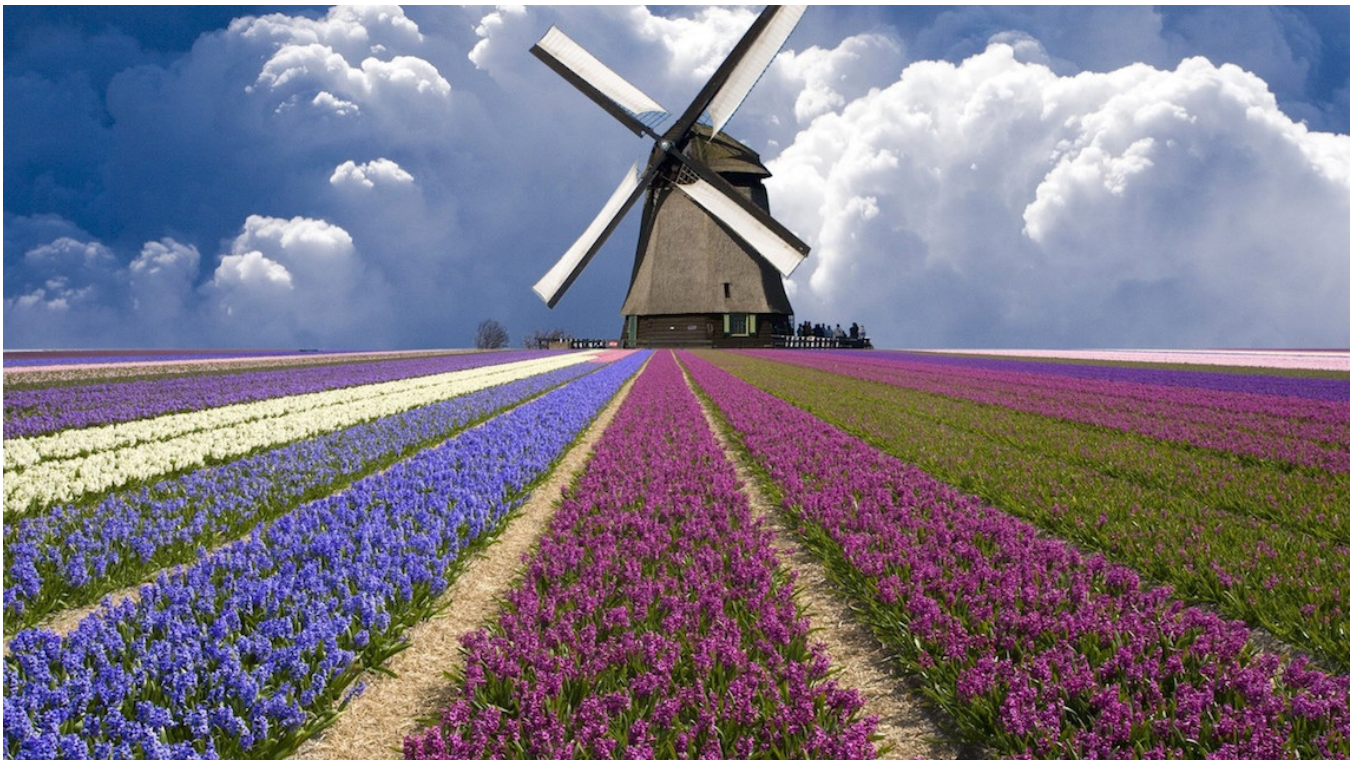
12 | 如何用面向对象思想写好并发程序？

2019-03-26 王宝令



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。

```
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。

```
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。

```
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。

精选留言

海鸿 13

我看有些人评论用volatie，volatie只能保证可见性，但是保证不了原子性，所以得加锁保证互斥。

老师我这样理解对吗？

2019-03-26

作者回复

相当正确！

2019-03-27

QQ怪 7

必须加锁啊，synchronized (this)就行了，最简单加锁吧，volatile只能保证内存可见性，并不能保证原子性

2019-03-27

作者回复

[]

2019-03-27

magict4 7

1. setUpper() 跟 setLower() 都加上 "synchronized" 关键字。不要太在意性能，老师都说了，避免过早优化。
2. 如果性能有问题，可以把 lower 跟 upper 两个变量封装到一个类中，例如

```
public class Boundary {
private final lower;
private final upper;

public Boundary(long lower, long upper) {
if(lower >= upper) {
// throw exception
}
this.lower = lower;
this.upper = upper;
}
}
```

移除 SafeVM 的 setUpper() 跟 setLower() 方法，并增入 setBoundary(Boundary boundary) 方法。

2019-03-26

作者回复

□□

2019-03-27

Cc 3

又想到一种，既然两个变量要同时锁定，那就把两个变量封装成一个，然后使用cas操作。这样行不行，另外老师帮我看看volatile是不是有多余的地方

```
.....
volatile AtomicReference<Inventory> inventory = new AtomicReference<>();

static class Inventory {
private volatile long upper = 0;

private volatile long lower = 0;
}

void setUpper(long v) {
long low;
Inventory oldObj;
Inventory newObj;
do {
```

```
oldObj = inventory.get();
if (v >= (low = oldObj.lower)) {
throw new IllegalArgumentException();
}
newObj = new Inventory();
newObj.lower = low;
newObj.upper = v;

} while (inventory.compareAndSet(oldObj, newObj));
}

void setLower(long v) {
long upp;
Inventory oldObj;
Inventory newObj;
do {
oldObj = inventory.get();
if (v <= (upp = oldObj.upper)) {
throw new IllegalArgumentException();
}
newObj = new Inventory();
newObj.lower = v;
newObj.upper = upp;

} while (inventory.compareAndSet(oldObj, newObj));
}
```

2019-03-29

作者回复

我觉得这个没有问题，volatile 换成 final会更好

2019-03-30

抽离の

3

老师、讲的真好！

2019-03-26

无庸

3

compareAndSet吧

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

}

2019-03-26

作者回复

不满足合理库存的约束条件

2019-03-26

Boomkeeper 2

我就不明白了，使用了synchronized，为啥还用voliate，他的确是保证可见性，但是并不能保证原子性，一般他的应用场景应该是不依赖之前的结果而改变数据，累加的场景明显不适合

2019-04-09

悟空 2

访问时使用syncchronize对类加锁。保证变量访问的互斥

2019-03-26

作者回复

对象加锁就可以了

2019-03-26

Lemon 2

使用 Condition

```
public class SafeWM {

// 库存上限
private final AtomicLong upper =
new AtomicLong(10);
// 库存下限
private final AtomicLong lower =
new AtomicLong(2);

private ReentrantLock lock = new ReentrantLock();
private Condition c1 = lock.newCondition();
private Condition c2 = lock.newCondition();

// 设置库存上限
void setUpper(long v) {
try {
lock.lock();
// 检查参数合法性
while (v < lower.get()) {
c1.await();
}
upper.set(v);
c2.signal();
} catch (Exception e) {
```

```
e.printStackTrace();
} finally {
lock.unlock();
}

}

// 设置库存下限
void setLower(long v) {

try {
lock.lock();
// 检查参数合法性
while (v > upper.get()) {
c2.await();
}
lower.set(v);
c1.signal();
} catch (Exception e) {
e.printStackTrace();
} finally {
lock.unlock();
}
}

}
```

2019-03-26

作者回复

错误的设置会导致永远等待，太危险了

2019-03-26

Zm

2

Volatile修饰变量

2019-03-26

京

2

一早起来，就把文章看完了，期待老师后面更精彩的内容

2019-03-26

逆水行舟

1

那本书，有些晦涩，但是是必读的。

2019-03-31

作者回复

太晦涩了

2019-03-31

Tomcat

1

老师，这样写有什么问题吗，总感觉哪里怪怪的。

```
public class DBPush {
private volatile static DBPush dbPush = null;

private DBPush() {
}

public static DBPush getInStance() {
if (dbPush == null) {
synchronized (DBPush.class) {
if (dbPush == null) {
dbPush = new DBPush();
}
}
}
return dbPush;
}
}
```

2019-03-26

作者回复

没问题

2019-03-26

undifined

1

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 lowe r 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

作者回复

我感觉不可以，还是有竞态条件，你可以在return前面增加sleep看看

2019-03-27

minggushen

0

getset方法使用一把锁就好，

2019-05-16

null

0

synchronized this(无需 volatile)
@蓝天白云看...
@非礼勿言-...

synchronized this + volatile：
@靖远小和尚
@陈华应

看了老师对他们的回复，感觉一下又迷糊了。

假设成员变量 upper、lower 只有 setUpper(...) 和 setLower(...) 方法对其更新，只要保证这两个 set 方法互斥就可以了，即 synchronized setUpper(...) 和 synchronized setLower(...)。而成员变量在此也可以从 atomic 换成原始数据类型。
如理解有误，还望老师指正，谢谢！！

2019-05-12

作者回复

你说的这个条件下，有synch保证互斥就没必要用其他手段了，如果还有读操作，而读操作没用synch，最好用同时用volatile。总之能用内存模型的规则推断出来可见性就可以

2019-05-13

皮蛋

0

方法内使用对象加可以解决竞态条件，但不就是用了低级的原语处理了吗，还有其他更好的方式吗？

2019-04-15

作者回复

可以用无锁算法，不过都很复杂，建议还是先用锁

2019-04-15

缪文@有赞

0

作为后端服务开发者，我们开发的是服务接口，实际上很多并发问题都不会遇到，比如这里提到的设置库存上下线，要考虑上限要大于下限，都是在数据库层才考虑的，java这一层dubbo为每个请求分配不同的线程的

2019-04-03

ZOU志伟

0

利用之前管程的管程知识也可以解决这个约束条件问题

2019-04-03

蓝天白云看大海

0

有些评论里说，加syncronized和volatile，但我觉得syncronized已经可以保证可见性啊，为啥还要加volatile？

2019-03-31

作者回复

如果用syncronized，就没必要加volatile

2019-03-31