

## 07-安全性、活跃性以及性能问题

通过前面六篇文章，我们开启了一个简单的并发旅程，相信现在你对并发编程需要注意的问题已经有了更深入的理解，这是一个很大的进步，正所谓只有发现问题，才能解决问题。但是前面六篇文章的知识点可能还是有点分散，所以是时候将其总结一下了。

并发编程中我们需要注意的问题有很多，很庆幸前人已经帮我们总结过了，主要有三个方面，分别是：**安全性问题、活跃性问题和性能问题**。下面我就来一一介绍这些问题。

### 安全性问题

相信你一定听说过类似这样的描述：这个方法不是线程安全的，这个类不是线程安全的，等等。

那什么是线程安全呢？其实本质上就是正确性，而正确性的含义就是**程序按照我们期望的执行**，不要让我们感到意外。在[第一篇《可见性、原子性和有序性问题：并发编程Bug的源头》](#)中，我们已经见识过很多诡异的Bug，都是出乎我们预料的，它们都没有按照我们**期望**的执行。

那如何才能写出线程安全的程序呢？[第一篇文章](#)中已经介绍了并发Bug的三个主要源头：原子性问题、可见性问题和有序性问题。也就是说，理论上线程安全的程序，就要避免出现原子性问题、可见性问题和有序性问题。

那是不是所有的代码都需要认真分析一遍是否存在这三个问题呢？当然不是，其实只有一种情况需要：**存在共享数据并且该数据会发生变化，通俗地讲就是有多个线程会同时读写同一数据**。那如果能够做到不共享数据或者数据状态不发生变化，不就保证线程的安全性了嘛。有不少技术方案都是基于这个理论的，例如线程本地存储（Thread Local Storage，TLS）、不变模式等等，后面我会详细介绍相关的技术方案是如何在Java语言中实现的。

但是，现实生活中，**必须共享会发生变化的数据**，这样的应用场景还是很多的。

当多个线程同时访问同一数据，并且至少有一个线程会写这个数据的时候，如果我们不采取防护措施，那么就会导致并发Bug，对此还有一个专业的术语，叫做**数据竞争**（Data Race）。比如，前面[第一篇文章](#)里有个add10K()的方法，当多个线程调用时候就会发生**数据竞争**，如下所示。

```
public class Test {
    private long count = 0;
    void add10K() {
        int idx = 0;
        while(idx++ < 10000) {
            count += 1;
        }
    }
}
```

那是不是在访问数据的地方，我们加个锁保护一下就能解决所有的并发问题了呢？显然没有这么简单。例如，对于上面示例，我们稍作修改，增加两个被 synchronized 修饰的get()和set()方法，add10K()方法里面通过get()和set()方法来访问value变量，修改后的代码如下所示。对于修改后的代码，所有访问共享变量value的地方，我们都增加了互斥锁，此时是不存在数据竞争的。但很显然修改后的add10K()方法并不是线

程安全的。

```
public class Test {
    private long count = 0;
    synchronized long get(){
        return count;
    }
    synchronized void set(long v){
        count = v;
    }
    void add10K() {
        int idx = 0;
        while(idx++ < 10000) {
            set(get()+1)
        }
    }
}
```

假设count=0，当两个线程同时执行get()方法时，get()方法会返回相同的值0，两个线程执行get()+1操作，结果都是1，之后两个线程再将结果1写入了内存。你本来期望的是2，而结果却是1。

这种问题，有个官方的称呼，叫**竞态条件**（Race Condition）。所谓**竞态条件**，指的是程序的执行结果依赖**线程执行的顺序**。例如上面的例子，如果两个线程完全同时执行，那么结果是1；如果两个线程是前后执行，那么结果就是2。在并发环境里，线程的执行顺序是不确定的，如果程序存在竞态条件问题，那就意味着程序执行的结果是不确定的，而执行结果不确定这可是个大Bug。

下面再结合一个例子来说明下**竞态条件**，就是前面文章中提到的转账操作。转账操作里面有个判断条件——转出金额不能大于账户余额，但在并发环境里面，如果不加控制，当多个线程同时对一个账号执行转出操作时，就有可能出现超额转出问题。假设账户A有余额200，线程1和线程2都要从账户A转出150，在下面的代码里，有可能线程1和线程2同时执行到第6行，这样线程1和线程2都会发现转出金额150小于账户余额200，于是就会发生超额转出的情况。

```
class Account {
    private int balance;
    // 转账
    void transfer(
        Account target, int amt){
        if (this.balance > amt) {
            this.balance -= amt;
            target.balance += amt;
        }
    }
}
```

所以你也可以按照下面这样来理解**竞态条件**。在并发场景中，程序的执行依赖于某个状态变量，也就是类似于下面这样：

```
if (状态变量 满足 执行条件) {  
    执行操作  
}
```

当某个线程发现状态变量满足执行条件后，开始执行操作；可是就在这个线程执行操作的时候，其他线程同时修改了状态变量，导致状态变量不满足执行条件了。当然很多场景下，这个条件不是显式的，例如前面addOne的例子中，set(get()+1)这个复合操作，其实就隐式依赖get()的结果。

那面对数据竞争和竞态条件问题，又该如何保证线程的安全性呢？其实这两类问题，都可以用**互斥**这个技术方案，而实现**互斥**的方案有很多，CPU提供了相关的互斥指令，操作系统、编程语言也会提供相关的API。从逻辑上来看，我们可以统一归为：**锁**。前面几章我们也粗略地介绍了如何使用锁，相信你已经胸中有丘壑了，这里就不再赘述了，你可以结合前面的文章温故知新。

## 活跃性问题

所谓活跃性问题，指的是某个操作无法执行下去。我们常见的“死锁”就是一种典型的活跃性问题，当然除了死锁外，还有两种情况，分别是“活锁”和“饥饿”。

通过前面的学习你已经知道，发生“死锁”后线程会互相等待，而且会一直等待下去，在技术上的表现形式是线程永久地“阻塞”了。

但有时线程虽然没有发生阻塞，但仍然会存在执行不下去的情况，这就是所谓的“活锁”。可以类比现实世界里的例子，路人甲从左手边出门，路人乙从右手边进门，两人为了不相撞，互相谦让，路人甲让路走右手边，路人乙也让路走左手边，结果是两人又相撞了。这种情况，基本上谦让几次就解决了，因为人会交流啊。可是如果这种情况发生在编程世界了，就有可能一直没完没了地“谦让”下去，成为没有发生阻塞但依然执行不下去的“活锁”。

解决“活锁”的方案很简单，谦让时，尝试等待一个随机的时间就可以了。例如上面的那个例子，路人甲走左手边发现前面有人，并不是立刻换到右手边，而是等待一个随机的时间后，再换到右手边；同样，路人乙也不是立刻切换路线，也是等待一个随机的时间再切换。由于路人甲和路人乙等待的时间是随机的，所以同时相撞后再次相撞的概率就很低了。“等待一个随机时间”的方案虽然很简单，却非常有效，Raft这样知名的分布式一致性算法中也用到了它。

那“饥饿”该怎么去理解呢？所谓“饥饿”指的是线程因无法访问所需资源而无法执行下去的情况。“不患寡，而患不均”，如果线程优先级“不均”，在CPU繁忙的情况下，优先级低的线程得到执行的机会很小，就可能发生线程“饥饿”；持有锁的线程，如果执行的时间过长，也可能导致“饥饿”问题。

解决“饥饿”问题的方案很简单，有三种方案：一是保证资源充足，二是公平地分配资源，三就是避免持有锁的线程长时间执行。这三个方案中，方案一和方案三的适用场景比较有限，因为很多场景下，资源的稀缺性是没办法解决的，持有锁的线程执行的时间也很难缩短。倒是方案二的适用场景相对来说更多一些。

那如何公平地分配资源呢？在并发编程里，主要是使用公平锁。所谓公平锁，是一种先来后到的方案，线程的等待是有顺序的，排在等待队列前面的线程会优先获得资源。

## 性能问题

使用“锁”要非常小心，但是如果小心过度，也可能出“性能问题”。“锁”的过度使用可能导致串行化的范围过大，这样就不能够发挥多线程的优势了，而我们之所以使用多线程搞并发程序，为的就是提升性能。

所以我们要尽量减少串行，那串行对性能的影响是怎么样的呢？假设串行百分比是5%，我们用多核多线程相比单核单线程能提速多少呢？

有个阿姆达尔（Amdahl）定律，代表了处理器并行运算之后效率提升的能力，它正好可以解决这个问题，具体公式如下：

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

公式里的n可以理解为CPU的核数，p可以理解为并行百分比，那（1-p）就是串行百分比了，也就是我们假设的5%。我们再假设CPU的核数（也就是n）无穷大，那加速比S的极限就是20。也就是说，如果我们的串行率是5%，那么我们无论采用什么技术，最高也就只能提高20倍的性能。

所以使用锁的时候一定要关注对性能的影响。那怎么才能避免锁带来的性能问题呢？这个问题很复杂，Java SDK并发包里之所以有那么多东西，有很大一部分原因就是提升在某个特定领域的性能。

不过从方案层面，我们可以这样来解决这个问题。

第一，既然使用锁会带来性能问题，那最好的方案自然就是使用无锁的算法和数据结构了。在这方面有很多相关的技术，例如线程本地存储(Thread Local Storage, TLS)、写入时复制(Copy-on-write)、乐观锁等；Java并发包里面的原子类也是一种无锁的数据结构；Disruptor则是一个无锁的内存队列，性能都非常好……

第二，减少锁持有的时间。互斥锁本质上是将并行的程序串行化，所以要增加并行度，一定要减少持有锁的时间。这个方案具体的实现技术也有很多，例如使用细粒度的锁，一个典型的例子就是Java并发包里的ConcurrentHashMap，它使用了所谓分段锁的技术（这个技术后面我们会详细介绍）；还可以使用读写锁，也就是读是无锁的，只有写的时候才会互斥。

性能方面的度量指标有很多，我觉得有三个指标非常重要，就是：吞吐量、延迟和并发量。

1. 吞吐量：指的是单位时间内能处理的请求数量。吞吐量越高，说明性能越好。
2. 延迟：指的是从发出请求到收到响应的时间。延迟越小，说明性能越好。
3. 并发量：指的是能同时处理的请求数量，一般来说随着并发量的增加、延迟也会增加。所以延迟这个指标，一般都会是基于并发量来说的。例如并发量是1000的时候，延迟是50毫秒。

## 总结

并发编程是一个复杂的技术领域，微观上涉及到原子性问题、可见性问题和有序性问题，宏观则表现为安全性、活跃性以及性能问题。

我们在设计并发程序的时候，主要是从宏观出发，也就是要重点关注它的安全性、活跃性以及性能。安全性方面要注意数据竞争和竞态条件，活跃性方面需要注意死锁、活锁、饥饿等问题，性能方面我们虽然介绍了两个方案，但是遇到具体问题，你还是要具体分析，根据特定的场景选择合适的数据结构和算法。

要解决问题，首先要把问题分析清楚。同样，要写好并发程序，首先要了解并发程序相关的问题，经过这7章的内容，相信你一定对并发程序相关的问题有了深入的理解，同时对并发程序也一定心存敬畏，因为一不

小心就出问题了。不过这恰恰也是一个很好的开始，因为你已经学会了分析并发问题，然后解决并发问题也就不远了。

## 课后思考

Java语言提供的Vector是一个线程安全的容器，有同学写了下面的代码，你看看是否存在并发问题呢？

```
void addIfNotExist(Vector v,
    Object o){
    if(!v.contains(o)) {
        v.add(o);
    }
}
```

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。

## 猜你喜欢



## 精选留言：

- 峰 2019-03-14 07:05:19  
vector是线程安全，指的是它方法单独执行的时候没有并发正确性问题，并不代表把它的操作组合在一起问木有，而这个程序显然有老师讲的竞态条件问题。 [28赞]

作者回复2019-03-14 19:02:26



- 易水南风 2019-03-15 14:15:16  
add10k的例子不明白，因为两个方法都已经加上锁了，同一个test对象应该不可能两个线程同时执行吧？ [9赞]

作者回复2019-03-15 20:10:25

同时执行，指的是同时被调用。被锁串行后，还是有问题

- 虎虎♥ 2019-03-14 17:46:46  
老师讲的太好了。我没有并发的编程经验，但是可以看懂每一篇文章，也可以正确回答每节课后的习题。我觉得这次跟对了人，觉得很有希望跟着老师学好并发。

但是，这样跟着学完课程就能学好并发编程吗？老师可以给些建议吗？除了跟着课程，我还需要做些什么来巩固战果？老师能不能给加餐一篇学习方法，谢谢！

本节课总结：

安全性：

数据竞争：多个线程同时访问一个数据，并且至少有一个线程会写这个数据。

竞态条件：程序的执行结果依赖程序执行的顺序。

也可以按照以下方式理解竞态条件：程序的执行依赖于某个状态变量，在判断满足条件的时候执行，但是在执行时其他变量同时修改了状态变量。

```
if (状态变量 满足 执行条件) {
```

```
    执行操作
```

```
}
```

问题：数据竞争一定会导致程序存在竞态条件吗？有没有什么相关性？

活跃性：

死锁：破坏造成死锁的条件，1,使用等待-通知机制的Allocator; 2主动释放占有的资源；3,按顺序获取资源。

活锁：虽然没有发生阻塞，但仍会存在执行不下去的情况。我感觉像进入了某种怪圈。解决办法，等待随机的时间，例如Raft算法中重新选举leader。

饥饿：我想到了没有引入时间片概念时，cpu处理作业。如果遇到长作业，会导致短作业饥饿。如果优先处理短作业，则会饿死长作业。长作业就可以类比持有锁的时间过长，而时间片可以让cpu资源公平地分配给各个作业。当然，如果有无穷多的cpu，就可以让每个作业得以执行，就不存在饥饿了。

性能：

核心就是在保证安全性和活跃性的前提下，根据实际情况，尽量降低锁的粒度。即尽量减少持有锁的时间。JDK的并发包里，有很多特定场景针对并发性能的设计。还有很多无锁化的设计，例如MVCC，TLS，COW等，可以根据不同的场景选用不同的数据结构或设计。

最后，在程序设计时，要从宏观出发，也就是关注安全性，活跃性和性能。遇到问题的时候，可以从微观去分析，让看似诡异的bug无所遁形。 [9赞]

作者回复2019-03-14 22:36:46

能看懂说明基本功很扎实啊。你的建议我会考虑的。

● 亮亮 2019-03-14 08:30:02

```
void addIfNotExist(Vector v,
Object o){
    synchronized(v) {
        if(!v.contains(o)) {
            v.add(o);
        }
    }
}
```

这样不知道对不对 [7赞]

作者回复2019-03-15 00:35:14

对的

● 刘章周 2019-03-14 07:38:48

contains和add之间不是原子操作，有可能重复添加。 [6赞]

- ken 2019-03-14 08:26:24

实例不是线程安全的，Vector容器虽然是安全的单这个安全的原子性范围紧紧是每个成员方法。当需要调用多个方法来完成一个操作时Vector容器的原子性就适用了需要收到控制原子性，可以通过在方法上加synchronize保证安全性原子性。 [5赞]

作者回复2019-03-15 00:34:36

方法上加还不行

- Demter 2019-03-14 21:29:20

老师说两个线程同时访问get()，所以可能返回1.但是两个线程不可能同时访问get(),get（）上面有互斥锁啊，所以这个不是很懂啊 [4赞]

作者回复2019-03-14 22:24:46

同时访问，被串行化后，一先一后，结果两个线程都得到1

- hanmshashou 2019-03-14 08:31:18

ConcurrentHashMap 1.8后没有分段锁 syn + cas [3赞]

作者回复2019-03-15 00:31:32

是这样，高手！

- iron\_man 2019-03-16 15:28:39

关于活锁，看了老师举的例子还是不太明白。

死锁是多个线程互相持有彼此需要的资源，形成依赖循环。

活锁是多个线程类似死锁的情况下，同时释放掉自己已经获取的资源，然后同时获取另外一种资源，又形成依赖循环，导致都不能执行下去？不知道总结的对不对，老师可否点评一下？ [2赞]

作者回复2019-03-16 17:37:36

总结的对。就是同时放弃，然后又重试竞争，最后死循环在里面了。

- 飘呀飘的小叶子 2019-03-14 09:47:53

Vector实现线程安全是通过给主要的写方法加了synchronized，类似contains这样的读方法并没有synchronized，该题的问题就出在不是线程安全的contains方法，两个线程如果同时执行到if(!v.contains(o)) 是可以都通过的，这时就会执行两次add方法，重复添加。也就是老师说的竞态条件。 [2赞]

作者回复2019-03-14 21:32:28

👍

- 探索无止境 2019-03-14 07:26:01

吞吐量和并发量从文中描述的概念上来看，总觉得很像，具体该怎么区分？期待指点！ [2赞]

作者回复2019-03-14 22:28:13

对于一台webserver，吞吐量一般指的是server每秒钟能处理多少请求；并发量指的是有多少个客户端同时访问。

- 寒铁 2019-04-03 22:50:42

add10K() 如果用synchronized修饰 应该就没有问题了吧？ get和set是synchronized不能保证调用get和set之间的没有其他线程进入get和set，所以这是导致出错的根本原因。 [1赞]

作者回复2019-04-04 08:35:03

👍

- 你只是看起来很努力 2019-03-19 14:47:30

```
void addIfNotExist(Vector v,
Object o){
synchronized(v) {
if(!v.contains(o)) {
v.add(o);
}
}
}
```

老师关于亮亮这个改动我有个问题：如果两个线程读到的是一个满的vector，那么线程1先加锁执行，这时候会进行扩容，vector的地址就改变了，线程2再来执行的时候，它之前读取到的vector地址是已经释放掉的，那么程序不会出问题吗？ [1赞]

作者回复2019-03-20 08:52:03

vector的地址不会变，只是个指针而已

- 李林杰 2019-03-17 22:42:46

add10K例子中，set,get都是同一把锁，而且执行规则是set方法拿到锁之后，get方法再次获取该锁，不存在两个线程同时执行get方法啊，请老师解答下 [1赞]

作者回复2019-03-18 11:59:27

指的是方法被同时调用，不是先拿set的锁，是先拿get的锁。先计算参数，后调用方法

- 陈华应 2019-03-17 14:00:11

老师这里说被串行化还是1，是不是可见性问题？先执行的线程的count最新值并没有对后一个执行的可见啊 [1赞]

作者回复2019-03-17 15:02:50

执行count=1，压根就没有读操作，哪里来的可见性问题？

- 王玉坤 2019-03-16 21:58:36

老师，add10K()那块不是很懂，就算两个线程get()方法都读到0，他们在s调set()方法时因为是同步方法，总会一前一后的，根据happens-before原则，前面修改的值应该对后面可见，为什么这个地方会出错呢？ [1赞]

作者回复2019-03-17 10:58:10

两个线程同时执行set(1){count=1}，即便有同步，写到内存里的值也是1

- 果然如此 2019-03-15 09:19:34

问题是线程安全的。

线程锁从两个方面考虑，一是颗粒度，二是被锁的对象。

假设把锁加在addIfNotExist方法上虽然颗粒度达到了，但是多线程被锁的对象可能不是同一个，所以还要调整锁定的对象。 [1赞]

- 不靠谱的琴谱 2019-03-15 06:40:35

```
void addIfNotExist(Vector v,
Object o){
sync ( o ) {
if(!v.contains(o)) {
v.add(o);
}
```



```
}  
}  
}
```

思考题会有一个文中提到的竞态问题，虽然vector是线程安全的集合。然后这里如果保证添加同一个对象的原子性可不可以像我上面这样使用添加的对象当锁，降低锁的粒度

[1赞]

作者回复2019-03-15 12:08:04

我觉得锁v会更好

- 空知 2019-03-14 19:28:22

老师，转账这个例子如果不加措施，因为可见性，两个线程彼此看不见对方，初始值都>150 然后都减150变成50，或者因为任务切换 切换到另一个线程减了150也是50

会存在一个线程减到50，另一个线程从50开始减嘛？也就是一个减操作结束后刷到内存，另一个做减法的时候又从内存读值 [1赞]

作者回复2019-03-14 21:28:16

也有这个可能

- 我是卖报小行家 2019-03-14 09:34:24

不知道老师后序会不会介绍synchronized，lock，AQS一些底层原理，面试经常问道 [1赞]

作者回复2019-03-14 19:58:11

不会介绍，推荐你看看并发编程的艺术，那里面讲了很多这个