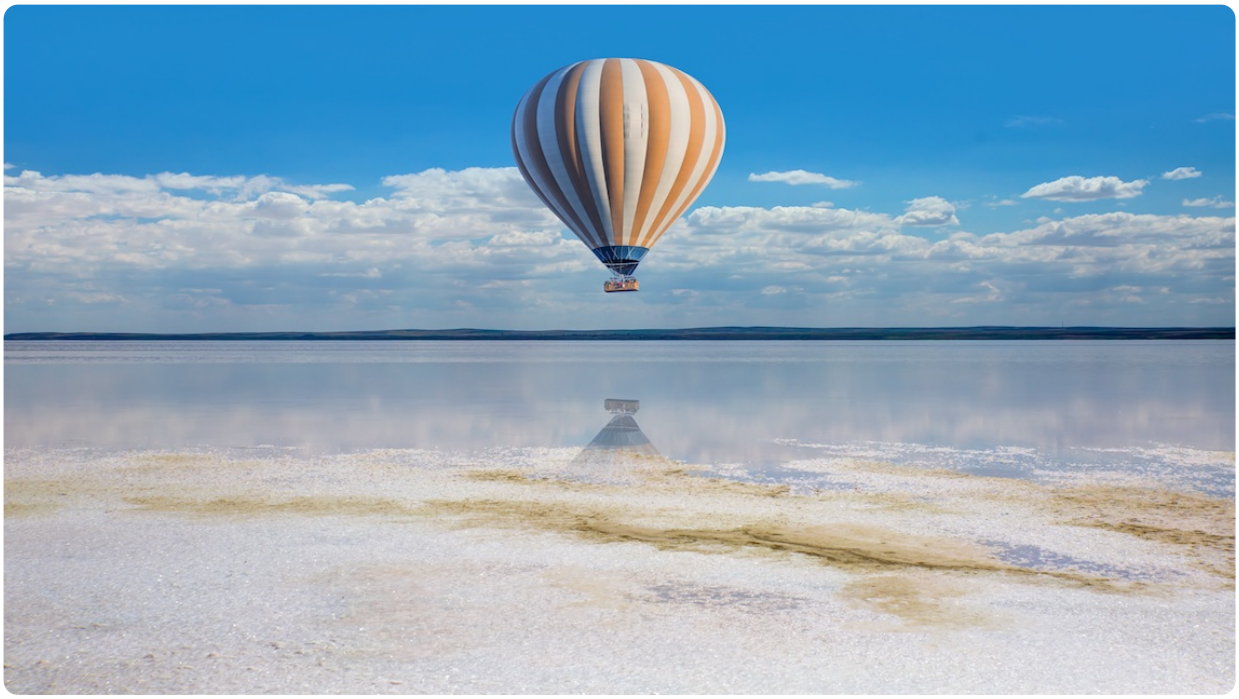
07 | 安全性、活跃性以及性能问题

王宝令 2019-03-14



00:00 13:01

讲述：王宝令　　大小：11.94M

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

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，不就能够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。

复制代码

```
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。

复制代码

```
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。

复制代码

```
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：

复制代码

```
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

其他线程间的修改了状态变量，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，就有可能会一直没完没了地“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

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要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?

复制代码

```
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

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

精选留言(19)

刘章周

contains和add之间不是原子操作，有可能重复添加。

2 2019-03-14

峰

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。

2 2019-03-14

亮亮

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

这样不知道对不对

1 2019-03-14

探索无止境

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！

1 2019-03-14

姜戈

Vector的contains操作与add操作缺乏原子性，存在判定不包含此对象时，同时重复加入同一对象的可能！

1 2019-03-14

新世界

vector的contain等读相关的操作是没有加synchronized的，这个竞态条件非线程安全

2019-03-14

raviup

rayjun

存在并发问题，vector 本身的线程安全并不能保证这 contains 和 add 的两个操作是线程安全的

2019-03-14

---

hanmshashou

ConcurrentHashMap 1.8后没有分段锁 syn + cas

2019-03-14

ken

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。

2019-03-14

冰激凌的眼泪

忘了哪里说的，check和set操作，是个典型的需要原子操作来保证线程安全的操作组合

2019-03-14

轻歌赋

存在线程安全问题，因为判断和add是分别线程安全的但是两个操作并不是原子的，会出现判断后被抢占CPU的情况。

请问老师，如何计算串行百分比呢?

2019-03-14

IamNigel

vector对象是共享的话会有并发问题

2019-03-14

gyl-coder

第二个代码样例中的get方法应该返回的是count吧 return long——> return count

2019-03-14

> 作者回复: 多谢多谢！是个bug

探索无止境

老师你好，多核CPU最多开启多少个线程是比较合适的？比如4核的CPU，这里面有什么分析的依据吗?

2019-03-14

> 作者回复: 后面有一章专门讲

小马

contains方法并不是synchronized的

2019-03-14

捞鱼的搬砖奇

另外想到SpringFrameWork中各种模板类都是线程安全的，模板类访问数据不同的持久化技术要绑定不同的会话资源，这些资源本身不是线程安全的。多线程环境下使用synchronized会降低并发,正是使用了ThreadLocal决绝了不用线程同步情况下解决了线程安全的问题。

2019-03-14

捞鱼的搬砖奇

自己思考了下，当对象可以表示某个状态，且这些状态可能会被修改，这样才有并发问题。如果对象带有状态，但只是只读，就不会有并发安全带问题。

2019-03-14

李皮皮皮皮皮

课后思考题：有并发问题，两个线程同时执行到第3行的判断，这时容器无元素o，判断通过。元素o会被添加两次☹

2019-03-14

西西弗与卡夫卡

有并发问题。两个线程先后判断出!contains某个值后，会先后都add该值。要synchronized整个addIfNotExist方法才行

2019-03-14