

## 13 | 理论基础模块热点问题答疑

2019-03-28 王宝令



讲述：王宝令

时长 12:24 大小 11.37M



到这里，专栏的第一模块——并发编程的理论基础，我们已经讲解完了，总共 12 篇，不算少，但“跳出来，看全景”你会发现这 12 篇的内容基本上是一个“串行的故事”。所以，在学习过程中，建议你从一个个单一的知识和技术中“跳出来”，看全局，搭建自己的并发编程知识体系。

为了便于你更好地学习和理解，下面我会先将这些知识点再简单地为你“串”一下，咱们一起复习下；然后就每篇文章的课后思考题、留言区的热门评论，我也集中总结和回复一下。

**那这个“串行的故事”是怎样的呢？**

起源是一个硬件的核心矛盾：CPU 与内存、I/O 的速度差异，系统软件（操作系统、编译器）在解决这个核心矛盾的同时，引入了可见性、原子性和有序性问题，这三个问题就是很多并发程序的 Bug 之源。这，就是01的内容。

那如何解决这三个问题呢？Java 语言自然有招儿，它提供了 **Java 内存模型和互斥锁方案**。所以，在02我们介绍了 Java 内存模型，以应对可见性和有序性问题；那另一个原子性问题该如何解决？多方考量用好互斥锁才是关键，这就是03和04的内容。

虽说互斥锁是解决并发问题的核心工具，但它也可能会带来死锁问题，所以05就介绍了死锁的产生原因以及解决方案；同时还引出一个线程间协作的问题，这也就引出了06这篇文章的内容，介绍线程间的协作机制：等待 - 通知。

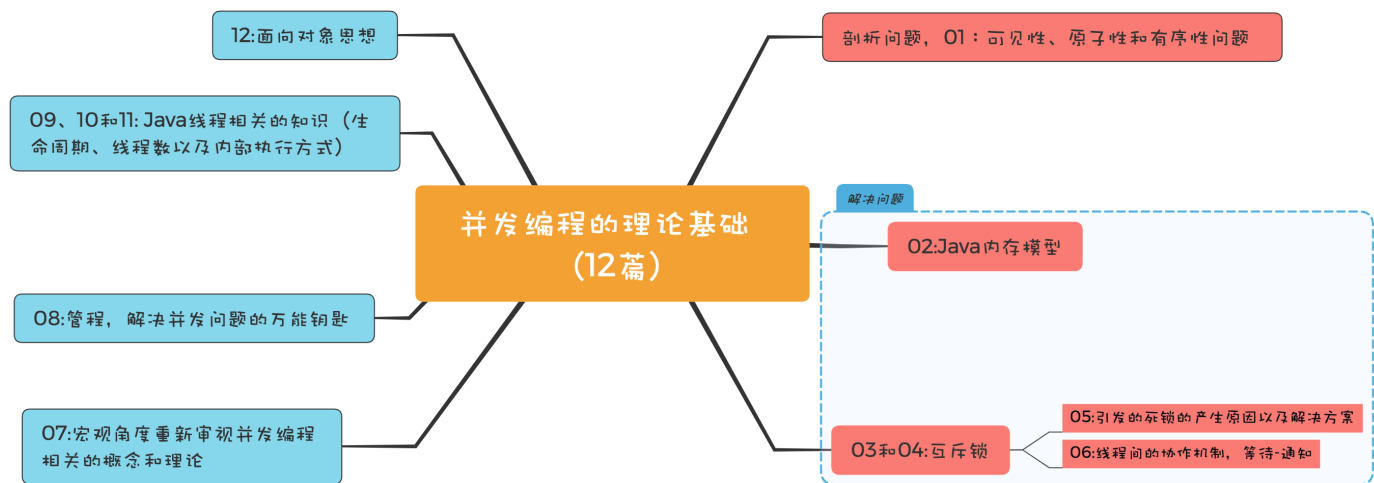
你应该也看出来了，前六篇文章，我们更多地是站在微观的角度看待并发问题。而07则是换一个角度，站在宏观的角度重新审视并发编程相关的概念和理论，同时也是对前六篇文章的查漏补缺。

08介绍的管程，是 Java 并发编程技术的基础，是解决并发问题的万能钥匙。并发编程里两大核心问题——互斥和同步，都是可以由管程来解决的。所以，学好管程，就相当于掌握了一把并发编程的万能钥匙。

至此，并发编程相关的问题，理论上你都应该能找到问题所在，并能给出理论上的解决方案了。

而后在09、10和11我们又介绍了线程相关的知识，毕竟 Java 并发编程是要靠多线程来实现的，所以有针对性地学习这部分知识也是很有必要的，包括线程的生命周期、如何计算合适的线程数以及线程内部是如何执行的。

最后，在12我们还介绍了如何用面向对象思想写好并发程序，因为在 Java 语言里，面向对象思想能够让并发编程变得更简单。



经过这样一个简要的总结，相信你此时对于并发编程相关的概念、理论、产生的背景以及它们背后的关系已经都有了一个相对全面的认识。至于更深刻的认识和应用体验，还是需要你“钻进去，看本质”，加深对技术本身的认识，拓展知识深度和广度。


另外，在每篇文章的最后，我都附上了一个思考题，这些思考题虽然大部分都很简单，但是隐藏的问题却很容易让人忽略，从而不经意间就引发了 Bug；再加上留言区的一些热门评论，所以我想着**将这些隐藏的问题或者易混淆的问题，做一个总结也是很有必要的。**

## 1. 用锁的最佳实践

例如，在[《03 | 互斥锁（上）：解决原子性问题》](#)和[《04 | 互斥锁（下）：如何用一把锁保护多个资源？》](#)这两篇文章中，我们的思考题都是关于如何创建正确的锁，而思考题里的做法都是错误的。

[03](#)的思考题的示例代码如下，`synchronized (new Object())` 这行代码很多同学已经分析出来了，每次调用方法 `get()`、`addOne()` 都创建了不同的锁，相当于无锁。这里需要你再次加深一下记忆，**“一个合理的受保护资源与锁之间的关联关系应该是 N:1”**。只有共享一把锁才能起到互斥的作用。

另外，很多同学也提到，JVM 开启逃逸分析之后，`synchronized (new Object())` 这行代码在实际执行的时候会被优化掉，也就是说在真实执行的时候，这行代码压根就不存在。不过无论你是否懂“逃逸分析”都不影响你学好并发编程，如果你对“逃逸分析”感兴趣，可以参考一些 JVM 相关的资料。

 复制代码


```
1 class SafeCalc {
2     long value = 0L;
3     long get() {
4         synchronized (new Object()) {
5             return value;
6         }
7     }
8     void addOne() {
9         synchronized (new Object()) {
10             value += 1;
11         }
12     }
13 }
```

04的思考题转换成代码，是下面这个样子。它的核心问题有两点：一个是锁有可能会变化，另一个是 Integer 和 String 类型的对象不适合做锁。如果锁发生变化，就意味着失去了互斥功能。Integer 和 String 类型的对象在 JVM 里面是可能被重用的，除此之外，JVM 里可能被重用的对象还有 Boolean，那重用意味着什么呢？意味着你的锁可能被其他代码使用，如果其他代码 synchronized(你的锁)，而且不释放，那你的程序就永远拿不到锁，这是隐藏的风险。

 复制代码

```
1 class Account {
2     // 账户余额
3     private Integer balance;
4     // 账户密码
5     private String password;
6     // 取款
7     void withdraw(Integer amt) {
8         synchronized(balance) {
9             if (this.balance > amt){
10                 this.balance -= amt;
11             }
12         }
13     }
14     // 更改密码
15     void updatePassword(String pw){
16         synchronized(password) {
17             this.password = pw;
18         }
19     }
20 }
```

通过这两个反例，我们可以总结出这样一个基本的原则：**锁，应是私有的、不可变的、不可重用的。**我们经常看到别人家的锁，都长成下面示例代码这样，这种写法貌不惊人，却能避免各种意想不到的坑，这个其实就是最佳实践。最佳实践这方面的资料推荐你看《Java 安全编码标准》这本书，研读里面的每一条规则都会让你受益匪浅。

 复制代码

```
1 // 普通对象锁
2 private final Object
3     lock = new Object();
4 // 静态对象锁
5 private static final Object
6     lock = new Object();
```

## 2. 锁的性能要看场景

[《05 | 一不小心就死锁了，怎么办？》](#)的思考题是比较`while(!actr.apply(this, target))`；这个方法和`synchronized(Account.class)`的性能哪个更好。

这个要看具体的应用场景，不同应用场景它们的性能表现是不同的。在这个思考题里面，如果转账操作非常费时，那么前者的性能优势就显示出来了，因为前者允许 A->B、C->D 这种转账业务的并行。不同的并发场景用不同的方案，这是并发编程里面的一项基本原则；没有通吃的技术和方案，因为每种技术和方案都是优缺点和适用场景的。

## 3. 竞态条件需要格外关注

[《07 | 安全性、活跃性以及性能问题》](#)里的思考题是一种典型的竞态条件问题（如下所示）。竞态条件问题非常容易被忽略，`contains()` 和 `add()` 方法虽然都是线程安全的，但是组合在一起却不是线程安全的。所以你的程序里如果存在类似的组合操作，一定要小心。

 复制代码

```
1 void addIfNotExist(Vector v,  
2     Object o){  
3     if(!v.contains(o)) {  
4         v.add(o);  
5     }  
6 }
```

这道思考题的解决方法，可以参考 [《12 | 如何用面向对象思想写好并发程序？》](#)，你需要将共享变量 `v` 封装在对象的内部，而后控制并发访问的路径，这样就能有效防止对 `Vector v` 变量的滥用，从而导致并发问题。你可以参考下面的示例代码来加深理解。


 复制代码

```
1 class SafeVector{  
2     private Vector v;  
3     // 所有公共方法增加同步控制  
4     synchronized  
5     void addIfNotExist(Object o){  
6         if(!v.contains(o)) {  
7             v.add(o);  
8         }  
9     }  
10 }
```




## 4. 方法调用是先计算参数

不过，还有同学对[07](#)文中所举的例子有疑议，认为`set(get()+1)`；这条语句是进入 `set()` 方法之后才执行 `get()` 方法，其实并不是这样的。**方法的调用，是先计算参数，然后将参数压入调用栈之后才会执行方法体**，方法调用的过程在[11](#)这篇文章中我们已经做了详细的介绍，你可以再次重温一下。

 复制代码


```
1 while(idx++ < 10000) {  
2     set(get()+1);  
3 }
```

先计算参数这个事情也是容易被忽视的细节。例如，下面写日志的代码，如果日志级别设置为 INFO，虽然这行代码不会写日志，但是会计算 `"The var1: " + var1 + ", var2:" + var2` 的值，因为方法调用前会先计算参数。

 复制代码

```
1 logger.debug("The var1: " +  
2     var1 + ", var2:" + var2);
```

更好地写法应该是下面这样，**这种写法仅仅是讲参数压栈，而没有参数的计算。使用{}占位符是写日志的一个良好习惯。**


 复制代码

```
1 logger.debug("The var1: {}, var2:{}",  
2     var1, var2);
```

## 5. InterruptedException 异常处理需小心

[《09 | Java 线程（上）：Java 线程的生命周期》](#)的思考题主要是希望你能够注意 `InterruptedException` 的处理方式。当你调用 Java 对象的 `wait()` 方法或者线程的 `sleep()` 方法时，需要捕获并处理 `InterruptedException` 异常，在思考题里面（如下所示），本意是


通过 `isInterrupted()` 检查线程是否被中断了，如果中断了就退出 `while` 循环。当其他线程通过调用 `th.interrupt()` 来中断 `th` 线程时，会设置 `th` 线程的中断标志位，从而使 `th.isInterrupted()` 返回 `true`，这样就能退出 `while` 循环了。

 复制代码

```
1 Thread th = Thread.currentThread();
2 while(true) {
3     if(th.isInterrupted()) {
4         break;
5     }
6     // 省略业务代码无数
7     try {
8         Thread.sleep(100);
9     }catch (InterruptedException e){
10         e.printStackTrace();
11     }
12 }
```

这看上去一点问题没有，实际上却是几乎起不了作用。原因是这段代码在执行的时候，大部分时间都是阻塞在 `sleep(100)` 上，当其他线程通过调用 `th.interrupt()` 来中断 `th` 线程时，大概率地会触发 `InterruptedException` 异常，**在触发 `InterruptedException` 异常的同时，JVM 会同时把线程的中断标志位清除**，所以这个时候 `th.isInterrupted()` 返回的是 `false`。

正确的处理方式应该是捕获异常之后重新设置中断标志位，也就是下面这样：

 复制代码

```
1 try {
2     Thread.sleep(100);
3 }catch(InterruptedException e){
4     // 重新设置中断标志位
5     th.interrupt();
6 }
```

## 6. 理论值 or 经验值

《10 | Java 线程（中）：创建多少线程才是合适的？》的思考题是：经验值为“最佳线程 =  $2 * \text{CPU 的核数} + 1$ ”，是否合理？

从理论上讲，这个经验值一定是靠不住的。但是经验值对于很多“I/O 耗时 / CPU 耗时”不太容易确定的系统来说，却是一个很好到初始值。

我们曾讲到最佳线程数最终还是靠压测来确定的，实际工作中大家面临的系统，“I/O 耗时 / CPU 耗时”往往都大于 1，所以基本上都是在这个**初始值的基础上增加**。增加的过程中，应关注线程数是如何影响吞吐量和延迟的。一般来讲，随着线程数的增加，吞吐量会增加，延迟也会缓慢增加；但是当线程数增加到一定程度，吞吐量就会开始下降，延迟会迅速增加。这个时候基本上就是线程能够设置的最大值了。

实际工作中，不同的 I/O 模型对最佳线程数的影响非常大，例如大名鼎鼎的 Nginx 用的是非阻塞 I/O，采用的是多进程单线程结构，Nginx 本来是一个 I/O 密集型系统，但是最佳进程数设置的却是 CPU 的核数，完全参考的是 CPU 密集型的算法。所以，理论我们还是要活学活用。

## 总结

这个模块，内容主要聚焦在并发编程相关的理论上，但是思考题则是聚焦在细节上，我们经常说细节决定成败，在并发编程领域尤其如此。理论主要用来给我们提供解决问题的思路和方法，但在具体实践的时候，还必须重点关注每一个细节，哪怕有一个细节没有处理好，都会导致并发问题。这方面推荐你认真阅读《Java 安全编码标准》这本书，如果你英文足够好，也可以参考[这份文档](#)。

最后总结一句，学好理论有思路，关注细节定成败。

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

### 玩转 Spring 全家桶

一站通关 Spring、Spring Boot 与 Spring Cloud

戳此试读



丁雪丰  
平安壹钱包高级架构师  
《Spring Boot 实战》  
《Spring 攻略》译者



上一篇 12 | 如何用面向对象思想写好并发程序?

## 精选留言 (7)

写留言



**binary**

2019-03-28

6

这个专栏内容值得反复阅读!



**iamNigel**

2019-03-28

3

Integer string Boolean的可重用没太明白，希望老师讲解下



**Jialin**

2019-03-28

1

建议iamNigel同学去搜索下Integer String Boolean相关的知识，Integer会缓存-128 ~ 127这个范围内的数值，String对象同样会缓存字符串常量到字符串常量池，可供重复使用，所以不能用来用作锁对象，网上有相关的知识讲解和面试问题  
老师讲解的非常不错，单看每一节，觉得自己已略一二，学完这节才发现要自己的知识点要串起来，整体了解并发



**皮卡皮卡丘**

2019-03-28

1

看下源码就知道了，Integer里有个内部类，会缓存一定范围的整数



**wypsma11**

2019-03-28

1

看来要重读前12章啦



**程序员人生**

2019-03-28

1



张建磊

2019-03-28



王老师好，在第11讲中，new出的对象放入堆，局部变量放入栈帧。那么new出的线程会放到哪里？麻烦老师这块能否展开讲一下，谢谢😊