

03 | 互斥锁（上）：解决原子性问题

2019-03-05 王宝令



讲述：王宝令

时长 12:56 大小 11.85M



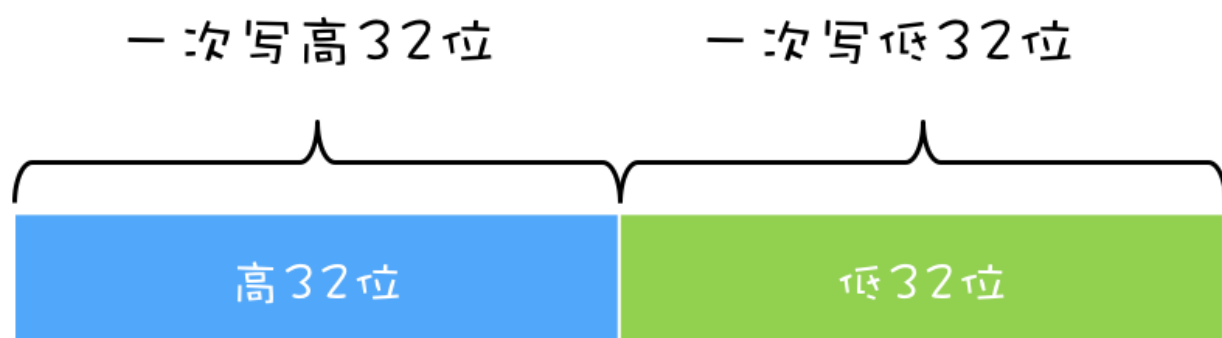
在[第一篇文章](#)中提到，一个或者多个操作在 CPU 执行的过程中不被中断的特性，称为“原子性”。理解这个特性有助于你分析并发编程 Bug 出现的原因，例如利用它可以分析出 long 型变量在 32 位机器上读写可能出现的诡异 Bug，明明已经把变量成功写入内存，重新读出来却不是自己写入的。

那原子性问题到底该如何解决呢？

你已经知道，**原子性问题的源头是线程切换**，如果能够禁用线程切换那不就能解决这个问题了吗？而操作系统做线程切换是依赖 CPU 中断的，所以禁止 CPU 发生中断就能够禁止线程切换。

在早期单核 CPU 时代，这个方案的确是可行的，而且也有很多应用案例，**但是并不适合多核场景**。这里我们以 32 位 CPU 上执行 long 型变量的写操作为例来说明这个问题，long

型变量是 64 位，在 32 位 CPU 上执行写操作会被拆分成两次写操作（写高 32 位和写低 32 位，如下图所示）。



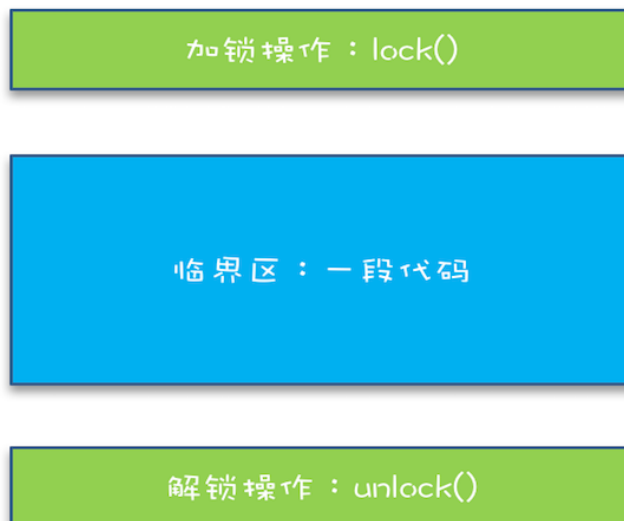
在单核 CPU 场景下，同一时刻只有一个线程执行，禁止 CPU 中断，意味着操作系统不会重新调度线程，也就是禁止了线程切换，获得 CPU 使用权的线程就可以不间断地执行，所以两次写操作一定是：要么都被执行，要么都没有被执行，具有原子性。

但是在多核场景下，同一时刻，有可能有两个线程同时在执行，一个线程执行在 CPU-1 上，一个线程执行在 CPU-2 上，此时禁止 CPU 中断，只能保证 CPU 上的线程连续执行，并不能保证同一时刻只有一个线程执行，如果这两个线程同时写 long 型变量高 32 位的话，那就有可能出现我们开头提及的诡异 Bug 了。

“同一时刻只有一个线程执行” 这个条件非常重要，我们称之为**互斥**。如果我们能够保证对共享变量的修改是互斥的，那么，无论是单核 CPU 还是多核 CPU，就都能保证原子性了。

简易锁模型

当谈到互斥，相信聪明的你一定想到了那个杀手级解决方案：锁。同时大脑中还会出现以下模型：



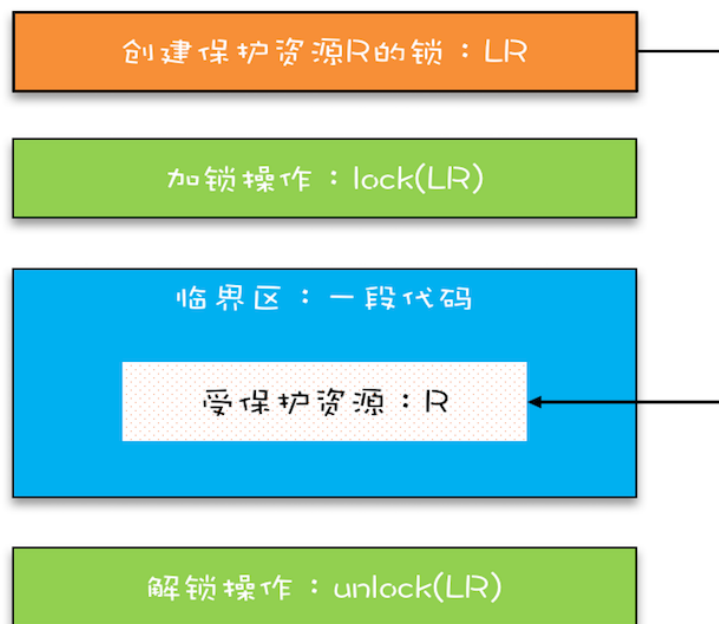
简易锁模型

我们把一段需要互斥执行的代码称为**临界区**。线程在进入临界区之前，首先尝试加锁 `lock()`，如果成功，则进入临界区，此时我们称这个线程持有锁；否则呢就等待，直到持有锁的线程解锁；持有锁的线程执行完临界区的代码后，执行解锁 `unlock()`。

这个过程非常像办公室里高峰期抢占坑位，每个人都是进坑锁门（加锁），出坑开门（解锁），如厕这个事就是**临界区**。很长时间里，我也是这么理解的。这样理解本身没有问题，但却很容易让我们忽视两个非常非常重要的点：我们锁的是什么？我们保护的又是什么？

改进后的锁模型

我们知道在现实世界里，锁和锁要保护的资源是有对应关系的，比如你用你家的锁保护你家的东西，我用我家的锁保护我家的东西。在并发编程世界里，锁和资源也应该有这个关系，但这个关系在我们上面的模型中是没有体现的，所以我们需要完善一下我们的模型。



改进后的锁模型

首先，我们要把临界区要保护的资源标注出来，如图中临界区里增加了一个元素：受保护的资源 R；其次，我们要保护资源 R 就得为它创建一把锁 LR；最后，针对这把锁 LR，我们还需在进出临界区时添上加锁操作和解锁操作。另外，在锁 LR 和受保护资源之间，我特地用一条线做了关联，这个关联关系非常重要。很多并发 Bug 的出现都是因为把它忽略了，然后就出现了类似锁自家门来保护他家资产的事情，这样的 Bug 非常不好诊断，因为潜意识里我们认为已经正确加锁了。

Java 语言提供的锁技术：synchronized

锁是一种通用的技术方案，Java 语言提供的 synchronized 关键字，就是锁的一种实现。synchronized 关键字可以用来修饰方法，也可以用来修饰代码块，它的使用示例基本上都是下面这个样子：

 复制代码

```
1 class X {
2     // 修饰非静态方法
3     synchronized void foo() {
4         // 临界区
5     }
6     // 修饰静态方法
7     synchronized static void bar() {
8         // 临界区
9     }
10    // 修饰代码块
```

```
11  Object obj = new Object();
12  void baz() {
13      synchronized(obj) {
14          // 临界区
15      }
16  }
17 }
```

看完之后你可能会觉得有点奇怪，这个和我们上面提到的模型有点对不上号啊，加锁 `lock()` 和解锁 `unlock()` 在哪里呢？其实这两个操作都是有的，只是这两个操作是被 Java 默默加上的，Java 编译器会在 `synchronized` 修饰的方法或代码块前后自动加上加锁 `lock()` 和解锁 `unlock()`，这样做的好处就是加锁 `lock()` 和解锁 `unlock()` 一定是成对出现的，毕竟忘记解锁 `unlock()` 可是个致命的 Bug（意味着其他线程只能死等下去了）。

那 `synchronized` 里的加锁 `lock()` 和解锁 `unlock()` 锁定的对象在哪里呢？上面的代码我们看到只有修饰代码块的时候，锁定了一个 `obj` 对象，那修饰方法的时候锁定的是什么呢？这个也是 Java 的一条隐式规则：

当修饰静态方法的时候，锁定的是当前类的 `Class` 对象，在上面的例子中就是 `Class X`；
当修饰非静态方法的时候，锁定的是当前实例对象 `this`。

对于上面的例子，`synchronized` 修饰静态方法相当于：

 复制代码

```
1  class X {
2      // 修饰静态方法
3      synchronized(X.class) static void bar() {
4          // 临界区
5      }
6  }
```

修饰非静态方法，相当于：


 复制代码

```
1  class X {
2      // 修饰非静态方法
3      synchronized(this) void foo() {
```

```
4    // 临界区
5 }
6 }
```

用 synchronized 解决 count+=1 问题

相信你一定记得我们前面文章中提到过的 `count+=1` 存在的并发问题，现在我们可以尝试用 `synchronized` 来小试牛刀一把，代码如下所示。SafeCalc 这个类有两个方法：一个是 `get()` 方法，用来获得 `value` 的值；另一个是 `addOne()` 方法，用来给 `value` 加 1，并且 `addOne()` 方法我们用 `synchronized` 修饰。那么我们使用的这两个方法有没有并发问题呢？

 复制代码

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


我们先来看看 `addOne()` 方法，首先可以肯定，被 `synchronized` 修饰后，无论是单核 CPU 还是多核 CPU，只有一个线程能够执行 `addOne()` 方法，所以一定能保证原子操作，那是否有可见性问题呢？要回答这问题，就要重温一下[上一篇文章](#)中提到的**管程中锁的规则**。

管程中锁的规则：对一个锁的解锁 Happens-Before 于后续对这个锁的加锁。

管程，就是我们这里的 `synchronized`（至于为什么叫管程，我们后面介绍），我们知道 `synchronized` 修饰的临界区是互斥的，也就是说同一时刻只有一个线程执行临界区的代码；而所谓“对一个锁解锁 Happens-Before 后续对这个锁的加锁”，指的是前一个线程的解锁操作对后一个线程的加锁操作可见，综合 Happens-Before 的传递性原则，我们就能得出前一个线程在临界区修改的共享变量（该操作在解锁之前），对后续进入临界区（该操作在加锁之后）的线程是可见的。

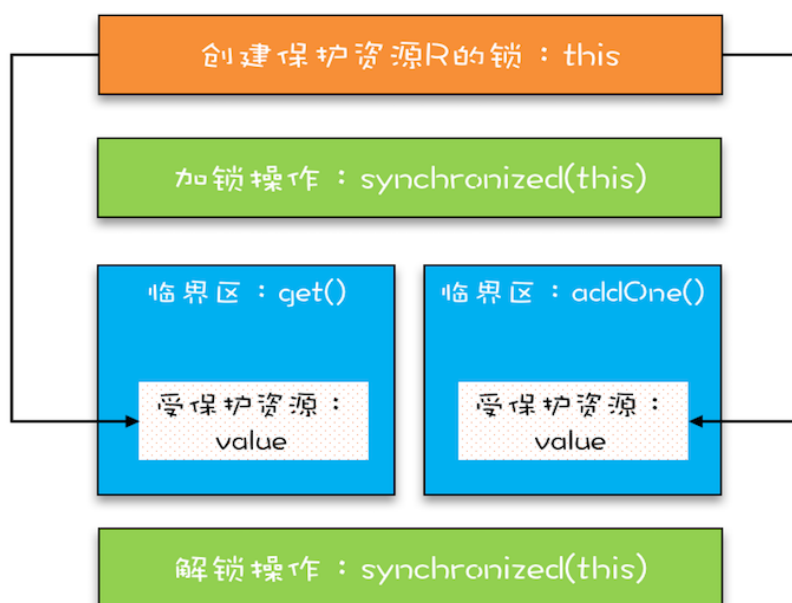
按照这个规则，如果多个线程同时执行 `addOne()` 方法，可见性是可以保证的，也就是说如果有 1000 个线程执行 `addOne()` 方法，最终结果一定是 `value` 的值增加了 1000。看到这个结果，我们长出一口气，问题终于解决了。

但也许，你一不小心就忽视了 `get()` 方法。执行 `addOne()` 方法后，`value` 的值对 `get()` 方法是可见的吗？这个可见性是无法保证的。管程中锁的规则，是只保证后续对这个锁的加锁的可见性，而 `get()` 方法并没有加锁操作，所以可见性无法保证。那如何解决呢？很简单，就是 `get()` 方法也 `synchronized` 一下，完整的代码如下所示。

 复制代码

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

上面的代码转换为我们提到的锁模型，就是下面图示这个样子。`get()` 方法和 `addOne()` 方法都需要访问 `value` 这个受保护的资源，这个资源用 `this` 这把锁来保护。线程要进入临界区 `get()` 和 `addOne()`，必须先获得 `this` 这把锁，这样 `get()` 和 `addOne()` 也是互斥的。



保护临界区 get() 和 addOne() 的示意图

这个模型更像现实世界里面球赛门票的管理，一个座位只允许一个人使用，这个座位就是“受保护资源”，球场的入口就是 Java 类里的方法，而门票就是用来保护资源的“锁”，Java 里的检票工作是由 synchronized 解决的。

锁和受保护资源的关系

我们前面提到，受保护资源和锁之间的关联关系非常重要，他们的关系是怎样的呢？一个合理的关系是：**受保护资源和锁之间的关联关系是 N:1 的关系**。还拿前面球赛门票的管理来类比，就是一个座位，我们只能用一张票来保护，如果多发了重复的票，那就要打架了。现实世界里，我们可以用多把锁来保护同一个资源，**但在并发领域是不行的，并发领域的锁和现实世界的锁不是完全匹配的**。不过倒是可以用同一把锁来保护多个资源，这个对应到现实世界就是我们所谓的“包场”了。

上面那个例子我稍作改动，把 value 改成静态变量，把 addOne() 方法改成静态方法，此时 get() 方法和 addOne() 方法是否存在并发问题呢？

复制代码

```
1 class SafeCalc {
2     static long value = 0L;
3     synchronized long get() {
4         return value;
5     }
```

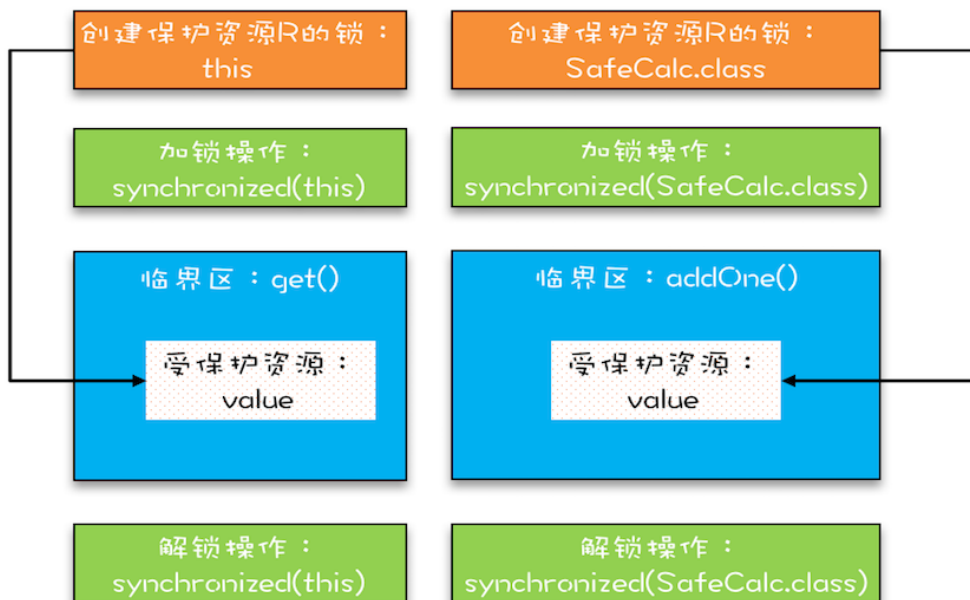


```

6   synchronized static void addOne() {
7       value += 1;
8   }
9 }

```

如果你仔细观察，就会发现改动后的代码是用两个锁保护一个资源。这个受保护的资源就是静态变量 `value`，两个锁分别是 `this` 和 `SafeCalc.class`。我们可以用下面这幅图来形象描述这个关系。由于临界区 `get()` 和 `addOne()` 是用两个锁保护的，因此这两个临界区没有互斥关系，临界区 `addOne()` 对 `value` 的修改对临界区 `get()` 也没有可见性保证，这就导致并发问题了。



两把锁保护一个资源的示意图


总结

互斥锁，在并发领域的知名度极高，只要有了并发问题，大家首先容易想到的就是加锁，因为大家都知道，加锁能够保证执行临界区代码的互斥性。这样理解虽然正确，但是却不能够指导你真正用好互斥锁。临界区的代码是操作受保护资源的路径，类似于球场的入口，入口一定要检票，也就是要加锁，但不是随便一把锁都能有效。所以必须深入分析锁定的对象和受保护资源的关系，综合考虑受保护资源的访问路径，多方面考量才能用好互斥锁。

synchronized 是 Java 在语言层面提供的互斥原语，其实 Java 里面还有很多其他类型的锁，但作为互斥锁，原理都是相通的：锁，一定有一个要锁定的对象，至于这个锁定的对象要保护的资源以及在哪里加锁 / 解锁，就属于设计层面的事情了。

课后思考

下面的代码用 synchronized 修饰代码块来尝试解决并发问题，你觉得这个使用方式正确吗？有哪些问题呢？能解决可见性和原子性问题吗？

 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 }
```

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。

Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得转载

上一篇 02 | Java内存模型：看Java如何解决可见性和有序性问题

精选留言 (42)

 写留言



nonohony

2019-03-05

 7

加锁本质就是在锁对象的对象头中写入当前线程id，但是new object每次在内存中都是新对象，所以加锁无效。



探索无止境

2019-03-05

 5

不能，**因为new了，所以不是同一把锁**。老师您好，我对那 synchronized的理解是这样，它并不能改变CPU时间片切换的特点，只是当其他线程要访问这个资源时，发现锁还未释放，所以只能在外面等待，不知道理解是否正确

展开 ∨



大南瓜

2019-03-05

👍 4

沙发，并不能，不是同一把锁

展开 ▾



zyl

2019-03-05

👍 3

sync锁的对象monitor指针指向一个ObjectMonitor对象，所有线程加入他的entrylist里面，去cas抢锁，更改state加1拿锁，执行完代码，释放锁state减1，和aqs机制差不多，只是所有线程不阻塞，cas抢锁，没有队列，属于非公平锁。

wait的时候，线程进waitset休眠，等待notify唤醒

展开 ▾



王大王

2019-03-05

👍 2

Get方法加锁不是为了解决原子性问题，这个读操作本身就是原子性的，是为了实现不能线程间addone方法的操作结果对get方法可见，那么value变量加volatile也可以实现同样效果吗？



小黄

2019-03-05

👍 1

明显getOne 和 addOne 每次加锁在不同资源上，并没有形成互斥

展开 ▾



黄朋飞

2019-03-05

👍 1

老师你好，能解释下get方法不加锁为什么可见性没法保证吗？没看懂

展开 ▾



Junzi

2019-03-05

👍

new Object()使每次线程访问该对象方法都会生成一个新的锁，即每次的锁都不一样，不能保证原子性和可见性



张三

2019-03-05



很有收获，相同的锁才互斥，并不是随意加一个锁就行了。

展开 ▾



[] Resolu...

2019-03-05



锁定不同的资源，导致无法解决可见性的问题，并且无法形成互斥关系，无法保证原子性问题

展开 ▾



刘志兵

2019-03-05



老实，静态方法锁住的对象是静态类对象，那如果静态方法中有多个静态方法，给其中一个加上synchronize而且被一个线程锁住的话，是不是其他线程执行其他方法也不行



冰激凌的眼...

2019-03-05



锁value是否有效果？

展开 ▾



空空空空

2019-03-05



回答问题：我觉得这样解决不了并发问题。该题目中每次锁的对象都是new出来的不同的对象，相当于每个线程获得的锁都是不一样的，并不会互斥。老师这样理解正确吗



DebugCat

2019-03-05



每个线程每次执行这两个方法时，每次都是对不同的对象（每次重新new）加锁（对象头中写入锁状态），并没有形成互斥锁。这样其实每次调用不同线程均可进入，无法保证可见性和原子性。

展开 ▾



影子传说
2019-03-05



老师上面用java内置锁保证count++操作能用volatile代替吗？volatile可以保证多线程可见，在结合happen before规则似乎也可以的。请老师指导下



Geek_961ee...
2019-03-05



想问下作者，类锁和对象实例锁之间有什么联系？

展开 ∨



Smile
2019-03-05



互斥性关注两个点：

1. 锁是什么对象
2. 锁住的内存是什么，临界区的内容

同一个锁只能一个线程占有，当一个线程抢占到锁时，其他线程只能等到释放后才可以抢到。...

展开 ∨



王大王
2019-03-05



管程锁规则和连续性规则一结合，可以实现对同一个锁保护的资源间实现可见性，真是太奥妙了



ChallengeN...
2019-03-05



synchronized的加锁解锁，具体是怎么实现的，没有讲

展开 ∨



波波
2019-03-05



这一节锁讲的有新意，很少有书籍特地讲锁和锁定资源的关系，赞一个

展开 ∨

