

03 | 互斥锁（上）：解决原子性问题

2019-03-05 王宝令

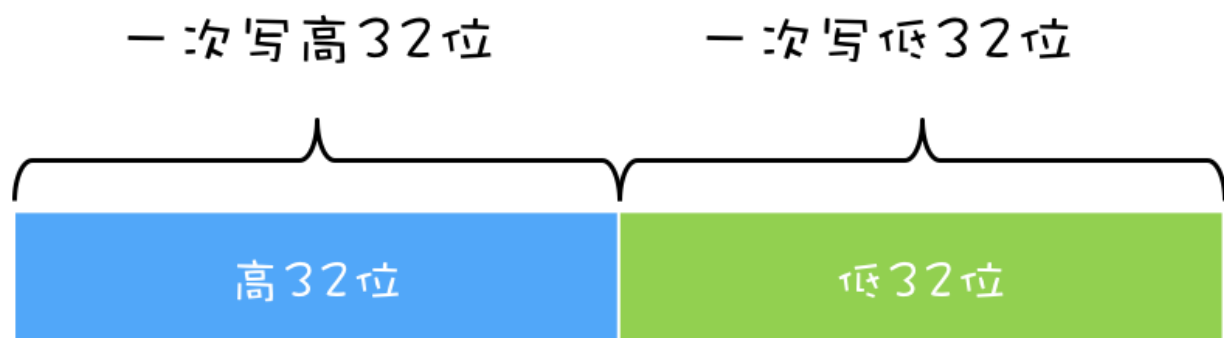


在[第一篇文章](#)中提到，一个或者多个操作在CPU执行的过程中不被中断的特性，称为“原子性”。理解这个特性有助于你分析并发编程Bug出现的原因，例如利用它可以分析出long型变量在32位机器上读写可能出现的诡异Bug，明明已经把变量成功写入内存，重新读出来却不是自己写入的。

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）。



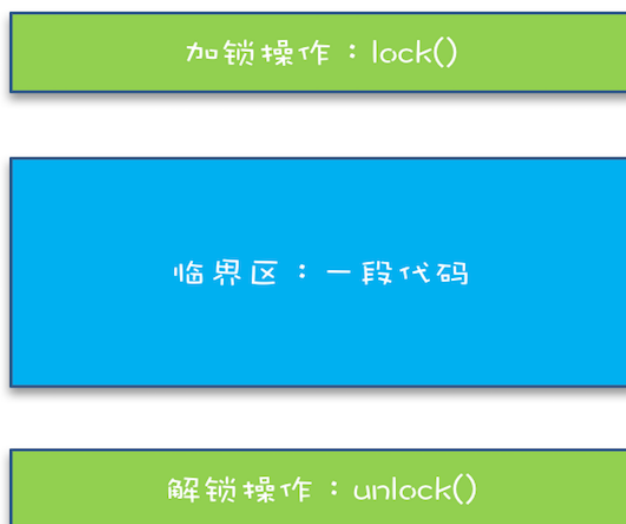
在单核CPU场景下，同一时刻只有一个线程执行，禁止CPU中断，意味着操作系统不会重新调度线程，也就是禁止了线程切换，获得CPU使用权的线程就可以不间断地执行，所以两次写操作一定是：要么都被执行，要么都没有被执行，具有原子性。

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：



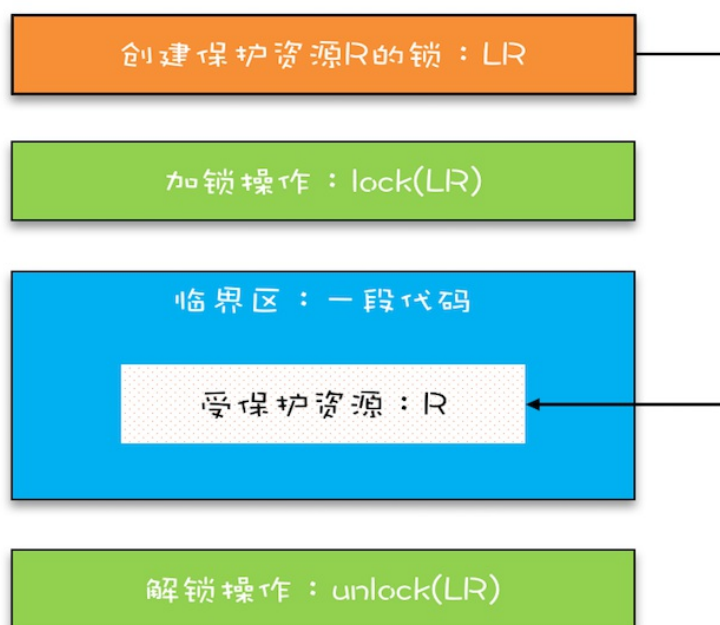
简易锁模型

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：

```
class X{
    // 修饰非静态方法
    synchronized void foo() {
        // 临界区
    }
    // 修饰静态方法
    synchronized static void bar() {
        // 临界区
    }
    // 修饰代码块
    Object obj = new Object();
    void baz() {
        synchronized(obj) {
            // 临界区
        }
    }
}
```

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：

```
class X{
    // 修饰静态方法
    synchronized(X.class) static void bar() {
        // 临界区
    }
}
```

修饰非静态方法，相当于：

```
class X{
    // 修饰非静态方法
    synchronized(this) void foo() {
        // 临界区
    }
}
```

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？

```
class SafeCalc {
    long value = 0L;
    long get() {
        return value;
    }
    synchronized void addOne() {
        value += 1;
    }
}
```

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。

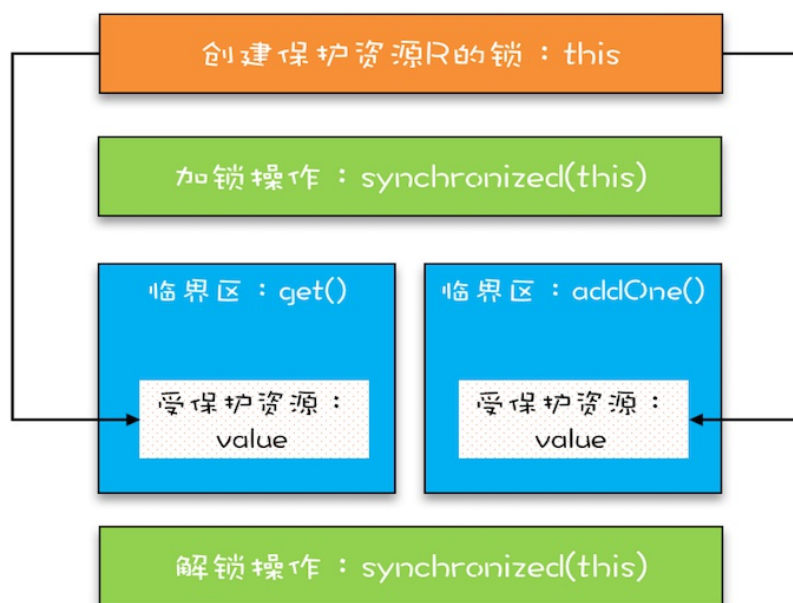
管程，就是我们这里的**synchronized**（至于为什么叫管程，我们后面介绍），我们知道**synchronized**修饰的临界区是互斥的，也就是说同一时刻只有一个线程执行临界区的代码；而所谓“对一个锁解锁 **Happens-Before** 后续对这个锁的加锁”，指的是前一个线程的解锁操作对后一个线程的加锁操作可见，综合**Happens-Before**的传递性原则，我们就能得出前一个线程在临界区修改的共享变量（该操作在解锁之前），对后续进入临界区（该操作在加锁之后）的线程是可见的。

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？这个可见性是没法保证的。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，所以可见性没法保证。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。

```
class SafeCalc {
    long value = 0L;
    synchronized long get() {
        return value;
    }
    synchronized void addOne() {
        value += 1;
    }
}
```

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。

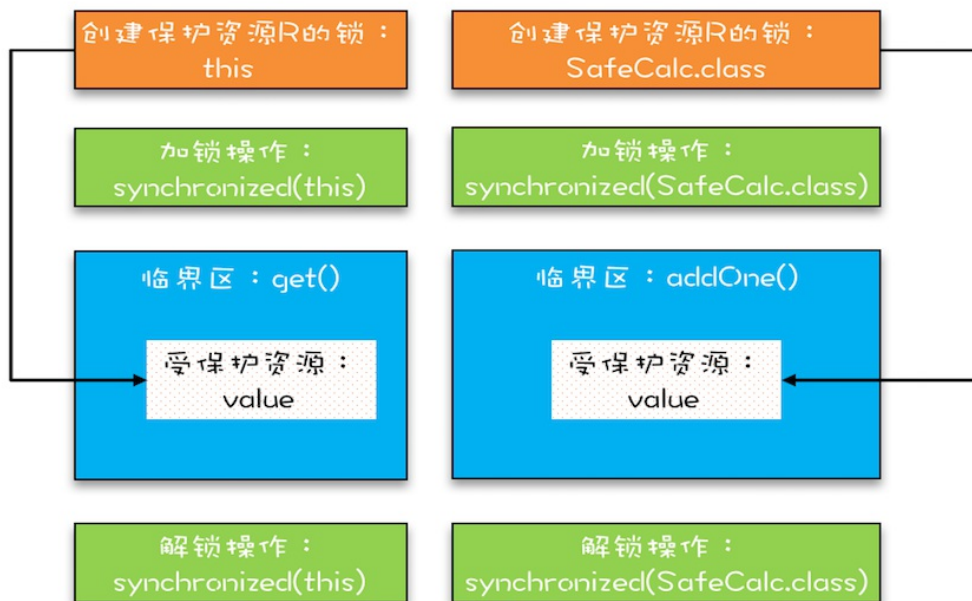
上面那个例子我稍作改动，把value改成静态变量，把addOne()方法改成静态方法，此时get()方法和addOne()方法是否存在并发问题呢？

```

class SafeCalc {
    static long value = 0L;
    synchronized long get() {
        return value;
    }
    synchronized static void addOne() {
        value += 1;
    }
}

```

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。

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？

```
class SafeCalc {  
    long value = 0L;  
    long get() {  
        synchronized (new Object()) {  
            return value;  
        }  
    }  
    void addOne() {  
        synchronized (new Object()) {  
            value += 1;  
        }  
    }  
}
```

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」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言



nonohony

👍 150

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。

2019-03-05

作者回复

synchronized的实现都知道了，厉害！

2019-03-05



zyl

👍 41

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

2019-03-05

作者回复

sync的优化都知道了，厉害啊

2019-03-05



w1sl1y

👍 39

经过JVM逃逸分析的优化后，这个sync代码直接会被优化掉，所以在运行时该代码块是无锁的

2019-03-05

| 作者回复

厉害

2019-03-06



探索无止境

👍 29

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

2019-03-05

| 作者回复

理解正确！

2019-03-05



老杨同志

👍 22

两把不同的锁，不能保护临界资源。而且这种new出来只在一个地方使用的对象，其它线程不能对它解锁，这个锁会被编译器优化掉。和没有**synchronized**代码块效果是相同的

2019-03-05

| 作者回复

实在是太厉害了！！！！

2019-03-05



王大王

👍 20

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？

2019-03-05

| 作者回复

是的，并发包里的原子类都是靠它实现的

2019-03-05



石头剪刀布

👍 10

老师说：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。不能用两把锁锁定同一个资源吗？

如下代码：

```
public class X {  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
    private int value = 0;
```

```
    private void addOne() {  
        synchronized (lock1) {  
            synchronized (lock2) {  
                value += 1;  
            }  
        }  
    }  
}
```

```
}  
}
```

```
private int get() {  
    synchronized (lock1) {  
        synchronized (lock2) {  
            return value;  
        }  
    }  
}
```

虽然说这样做没有实际意义，但是也不会导致死锁或者其他不好的结果吧？请老师指导，谢谢。

2019-03-08

作者回复

你这么优秀，我该怎么指导呢？你这不是用lock1 保护 lock2，lock2保护value吗？很符合我们的原则。我怎么没想到呢？

2019-03-09



sbweil

👍 8

最后的思考题: 多把锁保护同一个资源，就像一个厕所坑位，有N多门可以进去，没有丝毫保护效果，管理员一看，还不如把门都撤了，弄成开放式(编译器代码优化)。

2019-03-24



大南瓜

👍 8

沙发，并不能，不是同一把锁

2019-03-05

作者回复

为快点赞

2019-03-05



小和尚笨南北

👍 6

不正确

使用锁保护资源时，对资源的所有操作应该使用同一个锁，这样才能起到保护的作用。

课后题中每个线程对资源的操作都是用的是各自的锁，不存在互斥和竞争的情况。

这就相当于有一个房间，每个人过来都安装一个门，每个人都有自己门的钥匙，大家都可以随意出入这个房间。

由于每个线程都可以随时进入方法，所以存在原子性问题；

但是因为每次都有加锁和解锁的操作，unlock操作会使其他缓存的变量失效，需要重新从主内存中加载变量的值，所以可以解决可见性问题。

如有错误，请老师指正。

2019-03-05

| 作者回复

比喻很生动

2019-03-06



落落彩虹

👍 4

老师的文章我都要看几遍.评论区也不敢放过.

评论区有些demo, 注意关于join的hb原则; 注意system.out.println对可见性的影响, 该方法内部加锁了.

还有个问题, 如果我不用join, 而是sleep足够长时间以确保线程跑完了, 也能保证可见性.因为线程结束了他的本地工作空间该释放了, 数据要强制刷回内存了.....这也可以认为是join的hb原则吗? 我看网上说到hb都会把join的那个原则说成是线程终止规则.

2019-03-10

| 作者回复

感谢不离不弃啊

测试的时候经常用sleep, 实际项目还是用join吧。这个我感觉不能认为是join原则。规范里确实没有。

2019-03-10



侯大虎

👍 3

老师, 有个小问题 class锁锁的是该类的所有实例, 和this不应该是同一把锁吗(this不就是这个类的实例吗)?

2019-03-30

| 作者回复

没有包含关系, 就像公交卡和单次票一样, 都能坐车

2019-03-31



别皱眉

👍 3

老师, 我觉得get方法有必要用加锁来保证可见性的另一个理由如下:

```
class SafeCalc {  
    long value = 0L;  
  
    synchronized long get() {  
        return value;  
    }  
  
    synchronized void add(int i) {  
        // 业务代码....假如这里比较耗时  
        value += i;  
    }  
}
```

假如线程A执行add方法 当方法还没执行完

线程B执行get方法

如果get方法没有加锁 因为此时A正在修改这个数据 B获取的数据不是最新的

您看我说的对吗？还是说具体场景有不同的需求，有些还是允许这点延迟的？

本人大三，请前辈多指教☺谢谢

2019-03-13

作者回复

我觉得你这个才是正道，并发问题小心还躲不过呢，哪里敢冒险啊！没想到还有学生看这个专栏，有前途☺

2019-03-13



陈华

👍 3

我理解get方法不需要加synchronized关键字，也可以保证可见性。

因为 对 value的写有被 synchronized 修饰，addOne（）方法结束后，会强制其他CPU缓存失效，从新从内存读取最新值！

```
class SafeCalc {
    long value = 0L;
    long get() {
        return value;
    }
    synchronized void addOne() {
        value += 1;
    }
}
```

2019-03-07

作者回复

你说的对，从实现上看是这样。但是hb没有这样的要求

2019-03-07



churchchen

👍 3

```
class SafeCalc {
    static long value = 0L;
    synchronized long get() {
        return value;
    }
    synchronized static void addOne() {
        value += 1;
    }
}
```


get()方法的可见性不太理解为什么不能保证

2019-03-06

| 作者回复

get和addone锁的是一个对象，结合上一期的hb规则再想想

2019-03-07



ChallengeND

👍 3

synchronized的加锁解锁，具体是怎么实现的，没有讲

2019-03-05

| 作者回复

有兴趣的自己找资料看吧

2019-03-05



Monster!

👍 2

```
private volatile Long count = 0L;
private void add(){
    int add = 0;
    while (add++<10000){
        count ++;
    }
}

public static Long calc() throws InterruptedException {
    ThreadCountTest threadCountTest = new ThreadCountTest();
    Thread thread1 = new Thread(()->{
        threadCountTest.add();
    });
    Thread thread2 = new Thread(() -> {
        threadCountTest.add();
    });
    //开启两个线程
    thread1.start();
    thread2.start();
    //等待两个线程结束
    /*thread1.join();
    thread2.join();*/

    return threadCountTest.count;
}
```

王老师，请教一下，根据上一节的思考题，如果要想让线程A对共享资源的修改对线程B可见其中一种方法就是在共享变量上加volatile关键字，如上述代码在注释掉两个方法的join()时得出的结果为0，放开两个join()方法后得出的结果又是随机的，这个问题出在哪里呢？

2019-03-28

| 作者回复

那两个线程还没来得及跑，主线程执行到`return threadCountTest.count;`了

2019-03-28



别皱眉

👍 2

相信很多人跟我一样会碰到这个问题,评论里也看到有人在问，内容有点长，辛苦老师帮忙大家分析下了 哈哈

```
public class A implements Runnable {
    public Integer b = 1;

    @Override
    public void run() {
        System.out.println("A is begin!");
        while (true) {
            System.out.println("a");
            // System.out.println(b);
            if (b.equals(2))
                break;
        }

        System.out.println("A is finish!");
    }

    public static void main(String[] args) {
        A a = new A();
        //线程A
        new Thread(a).start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        a.b = 2;
    }
}
```

我们知道这个程序会出现可见性问题。

但是在`while`内加上`System.out.println(b)`后 当主线程修改`b`的值后 线程A居然能够取得最新值 可见性问题得到解决

`System.out.println(b)`的实现如下

```
public void println(String x) {
    synchronized (this) {
        print(x);
        newLine();
    }
}
```

```
}  
}
```

Doug Lea大神的Concurrent Programming in Java一书中有这样一个片段来描述synchronized这个关键字：

这里英文就不放出来了 字数超过两千.....

这篇文章也有提及<https://www.jianshu.com/p/3c06ffbf0d52>

简单翻译一下：从本质上来说，当线程释放一个锁时会强制性的将工作内存中之前所有的写操作都刷新到主内存中去，而获取一个锁则会强制性的加载可访问到的值到线程工作内存中来。虽然锁操作只对同步方法和同步代码块这一块起到作用，但是影响的却是线程执行操作所使用的的所有字段。

也就是说当调用System.out.println("a")时当前线程的缓存会被重新刷新过，所以才能够读到这个值最新值

然后问题来了

问题1:

首先上面的说法不知道是不是真的是这样。

然后我在下面加了System.out.println(b) 结果打印出来的是旧值，但是下面的b.equals(2)却能通过 这里没弄明白 我觉得应该是编译器进行了优化?因为现在大三能力不够，还没学会看class文件 没法验证

问题2:

网上找了一些文章

有些人的说法是：打印是IO操作，而IO操作会引起线程的切换，线程切换会导致线程原本的缓存失效，从而也会读取到修改后的值。

我尝试着将打印换成File file = new File("D://1.txt");这句代码，程序也能够正常的结束。当然，在这里也可以尝试将打印替换成synchronized(A.class){ }这句空同步代码块，发现程序也能够正常结束。

这里有个问题就是 线程切换时会把之前操作的相关数据保存到内存里，切换回来后会内存里的数据重新加载到寄存器里吗，这样说的话 就算切换也是获取不到修改后的值的,不知道是什么做到能够读到这个修改后的值的？

问题3:

是不是

线程执行过程中，操作系统会随机性的把缓存刷到内存

线程结束后一定会把缓存里的数据刷到内存

在评论里好多大神 能学到好多东西

2019-03-17

作者回复

1. `println`的代码里锁的`this`指的是你的控制台，这个锁跟你的代码没关系，而且`println`里也没有写操作，所以`println`不会导致强刷缓存。

我觉得是因为`println`产生了IO，IO相对CPU来说，太慢，所以这个期间大概率的会把缓存的值写入内存。也有可能这个线程被调度到了其他的CPU上，压根没有缓存，所以只能从内存取数。你调用`sleep`，效果应该也差不多。

2. 线程切换显然不足以保证可见性，保证的可见性只能靠hb规则。

3. 线程结束后，不一定会强刷缓存。否则Join的规则就没必要了

并发问题本来就是小概率的事件，尤其有了IO操作之后，概率就更低了。

2019-03-17



毛祥

2

线程每次调用方法锁的都是新`new`的一个对象。令哥讲解得透彻，让我这个菜鸟一看`code`就知道答案。此外，留言板潜伏一樽樽大神，有种豁然开朗的感觉。

2019-03-07

作者回复

你也会成为一尊大神的

2019-03-07



hxy

2

老师请问`synchronized`修饰的临界区中，如果不是同一把锁，能保证共享变量的可见性吗？

```
private final static int cnt = 10000;
```

```
private static int tmp = 0;
```

```
public static void main(String[] args) {
```

```
    Object lock1 = new Object();
```

```
    Object lock2 = new Object();
```

```
    Thread thread1 = new Thread(() -> {
```

```
        synchronized (lock1) {
```

```
            for (int i = 0; i < cnt; i++) {
```

```
                tmp++;
```

```
            }
```

```
        }
```

```
});  
Thread thread2 = new Thread() -> {  
    Watch.sleep(10L);  
    synchronized (lock2) {  
        System.out.println("tmp is: " + tmp);  
    }  
});
```

```
thread2.start();  
thread1.start();  
}
```

我这样试了一下，**thread2**最后读的值是正确的，感觉有一点奇怪

2019-03-06

作者回复

这种简单情况，实际上出bug的概率还真是很低。但是低并不意味着正确。

也不用奇怪，我们所说的都是可能。锁两个对象，编译器官方不保证可见性，私下里也许能保证。我们不能依赖于私下的方案。

2019-03-06