

# 并发编程带来的安全性挑战之同步锁

如果多个线程在做同一件事情的时候。

- 原子性 Synchronized, AtomicXXX、Lock,
- 可见性 Synchronized, volatile
- 有序性 Synchronized, volatile

## 原子性问题

在下面的案例中，演示了两个线程分别去调用 `demo.incr` 方法来对 `i` 这个变量进行叠加，预期结果应该是20000，但是实际结果却是小于等于20000的值。

```
public class Demo {  
    int i = 0;  
    public void incr(){  
        i++;  
    }  
    public static void main(String[] args) {  
        Demo demo = new Demo();  
        Thread[] threads=new Thread[2];  
        for (int j = 0;j<2;j++) {  
            threads[j]=new Thread(() -> { // 创建两个线程  
                for (int k=0;k<10000;k++) { // 每个线程跑10000次  
                    demo.incr();  
                }  
            });  
            threads[j].start();  
        }  
        try {  
            threads[0].join();  
            threads[1].join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println(demo.i);  
    }  
}
```

## 问题的原因

这个就是典型的线程安全问题中原子性问题的体现。那什么是原子性呢？

在上面这段代码中，`count++`是属于Java高级语言中的编程指令，而这些指令最终可能会有多条CPU指令来组成，而`count++`最终会生成3条指令，通过 `javap -v xxx.class` 查看字节码指令如下。

```

public incr()v
L0
LINENUMBER 13 L0
ALOAD 0
DUP
GETFIELD com/gupaoedu/pb/Demo.i : I    // 访问变量i
ICONST_1
IADD
结果放入操作数栈
PUTFIELD com/gupaoedu/pb/Demo.i : I    // 访问类字段（类变量），复制给Demo.i这个变量

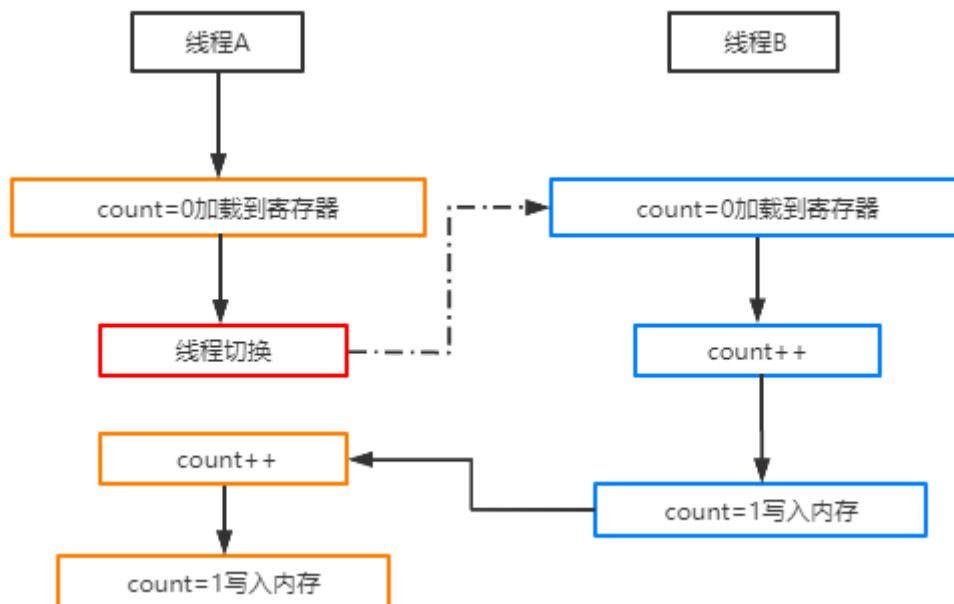
```

这三个操作，如果要满足原子性，那么就需要保证某个线程在执行这个指令时，不允许其他线程干扰，然后实际上，确实会存在这个问题。

## 图解问题本质

前面我们说过，一个CPU核心在同一时刻只能执行一个线程，如果线程数量远远大于CPU核心数，就会发生线程的切换，这个切换动作可以发生在任何一条CPU指令执行完之前。

对于 `i++` 这三个cpu指令来说，如果线程A在执行指令1之后，做了线程切换，假设切换到线程B，线程B同样执行CPU指令，执行的顺序如下图所示。就会导致最终的结果是1，而不是2.



这就是在多线程环境下，存在的原子性问题，那么，怎么解决这个问题呢？

大家认真观察上面这个图，表面上是多个线程对于同一个变量的操作，实际上是`count++`这行代码，它不是原子的。所以才导致在多线程环境下出现这样一个问题。

也就是说，我们只需要保证，`count++`这个指令在运行期间，在同一时刻只能由一个线程来访问，就可以解决问题。这就需要引出到今天的课程内容，同步锁Synchronized

## Synchronized的基本应用

`synchronized`有三种方式来加锁，不同的修饰类型，代表锁的控制粒度：

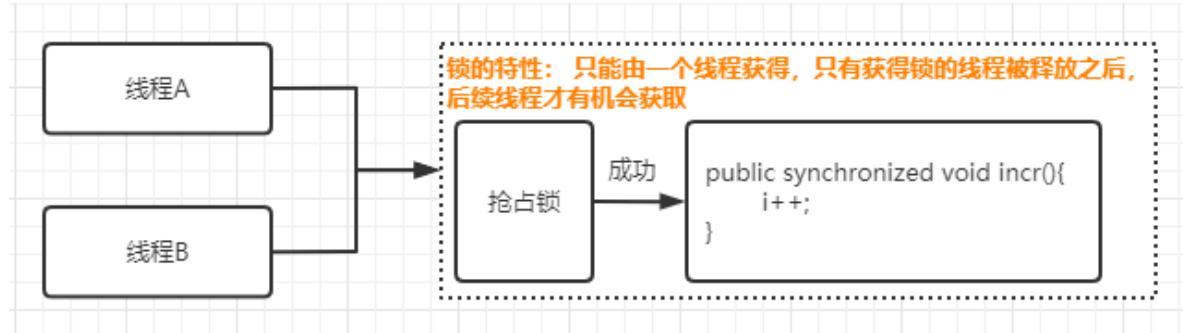
1. 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁

2. 静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
3. 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

## 锁的实现模型理解

Synchronized到底帮我们做了什么，为什么能够解决原子性呢？

在没有加锁之前，多个线程去调用incr()方法时，没有任何限制，都是可以同时拿到这个i的值进行++操作，但是当加了Synchronized锁之后，线程A和B就由并行执行变成了串行执行。



## Synchronized的原理

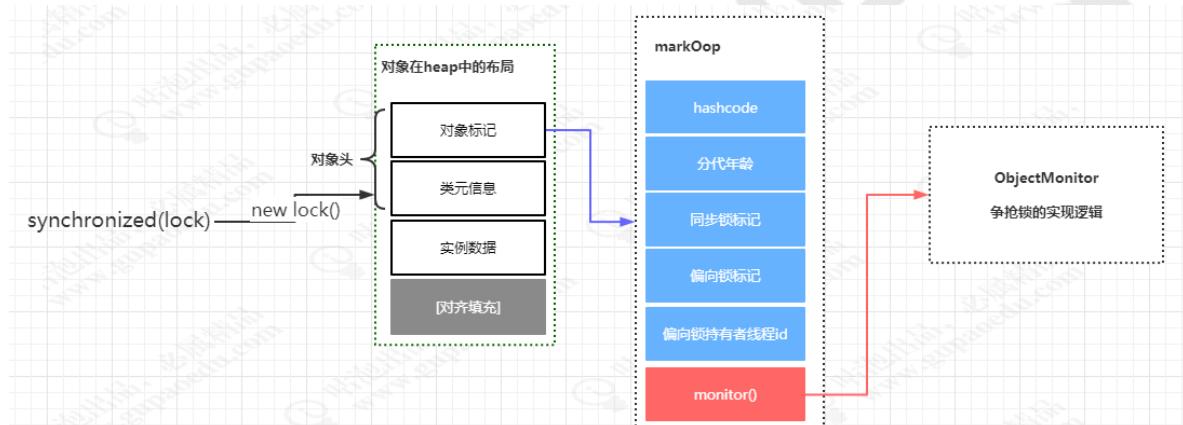
Synchronized是如何实现锁的，以及锁的信息是存储在哪里？就拿上面分析的图来说，线程A抢到锁了，线程B怎么知道当前锁被抢占了，这个地方一定会有一个标记来实现，而且这个标记一定是存储在某个地方。

## Markword对象头

这就要引出Markword对象头这个概念了，它是对象头的意思，简单理解，就是一个对象，在JVM内存中的布局或者存储的形式。

jdk8u: markOop.hpp

在Hotspot虚拟机中，对象在内存中的存储布局，可以分为三个区域:**对象头(Header)**、**实例数据(Instance Data)**、**对齐填充(Padding)**。



- mark-word: 对象标记字段占4个字节，用于存储一些列的标记位，比如：哈希值、轻量级锁的标记位，偏向锁标记位、分代年龄等。
- Klass Pointer: Class对象的类型指针，Jdk1.8默认开启指针压缩后为4字节，关闭指针压缩（-XX:-UseCompressedOops）后，长度为8字节。其指向的位置是对象对应的Class对象（其对应的元数据对象）的内存地址。

- 对象实际数据：包括对象的所有成员变量，大小由各个成员变量决定，比如：byte占1个字节8比特位、int占4个字节32比特位。
- 对齐：最后这段空间补全并非必须，仅仅为了起到占位符的作用。由于HotSpot虚拟机的内存管理系统要求对象起始地址必须是8字节的整数倍，所以对象头正好是8字节的倍数。因此当对象实例数据部分没有对齐的话，就需要通过对齐填充来补全。

## 通过ClassLayout打印对象头

为了让大家更加直观的看到对象的存储和实现，我们可以使用JOL查看对象的内存布局。

- 添加jol依赖

```
<dependency>
    <groupId>org.openjdk.jol</groupId>
    <artifactId>jol-core</artifactId>
    <version>0.9</version>
</dependency>
```

- 编写测试代码，在不加锁的情况下，对象头的信息打印

```
public class Demo {
    Object o=new Object();
    public static void main(String[] args) {
        Demo demo=new Demo(); //o这个对象，在内存中是如何存储和布局的。
        System.out.println(ClassLayout.parseInstance(demo).toPrintable());
    }
}
```

- 输出内容如下

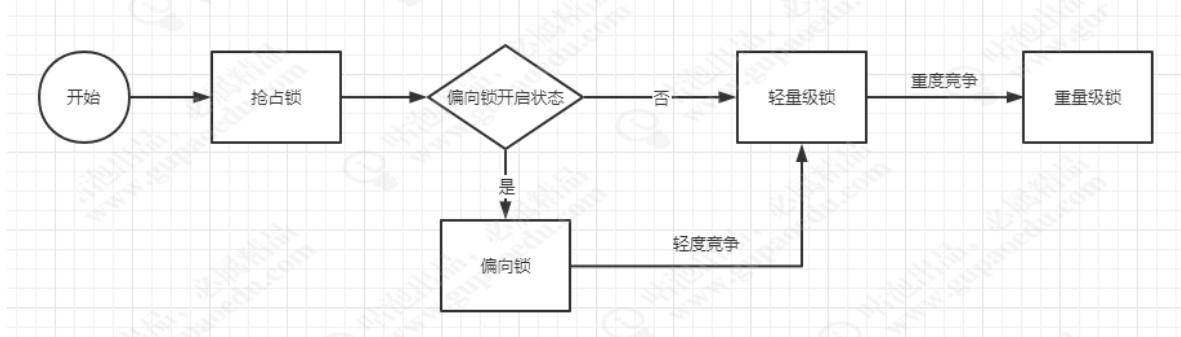
```
com.gupaoedu.pb.Demo object internals:
  OFFSET  SIZE            TYPE DESCRIPTION
  VALUE
      0     4          (object header)
  01 00 00 00 (00000001 00000000 00000000 00000000) (1)
      4     4          (object header)
  00 00 00 00 (00000000 00000000 00000000 00000000) (0)
      8     4          (object header)
  05 c1 00 f8 (00000101 11000001 00000000 11111000) (-134168315)
     12     4  java.lang.Object Demo.o
(Object)
  Instance size: 16 bytes
  Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

## 关于Synchronized锁的升级

Jdk1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

锁主要存在四中状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。

这么设计的目的，其实是为了减少重量级锁带来的性能开销，尽可能的在无锁状态下解决线程并发问题，其中偏向锁和轻量级锁的底层实现是基于自旋锁，它相对于重量级锁来说，算是一种无锁的实现。



- 默认情况下是偏向锁是开启状态，偏向的线程ID是0，偏向一个Anonymous BiasedLock
- 如果有线程去抢占锁，那么这个时候线程会先去抢占偏向锁，也就是把markword的线程ID改为当前抢占锁的线程ID的过程
- 如果有线程竞争，这个时候会撤销偏向锁，升级到轻量级锁，线程在自己的线程栈帧中会创建一个LockRecord，用CAS操作把markword设置为指向自己这个线程的LR的指针，设置成功后表示抢到锁。
- 如果竞争加剧，比如有线程超过10次自旋（-XX:PreBlockSpin参数配置），或者自旋线程数超过CPU核心数的一半，在1.6之后，加入了自适应自旋Adaptive Self Spinning。JVM会根据上次竞争的情况来自动控制自旋的时间。

升级到重量级锁，向操作系统申请资源，Linux Mutex，然后线程被挂起进入到等待队列。

## 轻量级锁的获取及原理

接下来，我们通过下面的例子来演示一下，通过加锁之后继续打印对象布局信息，来关注对象头里面的变化。

```
public class Demo {  
    Object o=new Object();  
    public static void main(String[] args) {  
        Demo demo=new Demo(); //o这个对象，在内存中是如何存储和布局的。  
        System.out.println(ClassLayout.parseInstance(demo).toPrintable());  
        synchronized (demo){  
            System.out.println(ClassLayout.parseInstance(demo).toPrintable());  
        }  
    }  
}
```

得到的对象布局信息如下

```
// 在未加锁之前，对象头中的第一个字节最后三位为 [001]，其中最后两位 [01] 表示无锁，第一位[0]也  
表示无锁  
com.gupaoedu.pb.Demo object internals:  
OFFSET  SIZE          TYPE DESCRIPTION          VALUE  
      0   4          (object header)          01 00  
00 00 (00000001 00000000 00000000 00000000) (1)  
      4   4          (object header)          00 00  
00 00 (00000000 00000000 00000000 00000000) (0)  
      8   4          (object header)          05 c1  
00 f8 (00000101 11000001 00000000 11111000) (-134168315)  
     12   4  java.lang.Object Demo.o  
(object)
```

```

Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
// 下面部分是加锁之后的对象布局变化
// 其中在前4个字节中，第一个字节最后三位都是[000]，后两位00表示轻量级锁，第一位为[0]，表示当前不是偏向锁状态。
com.gupaoedu.pb.Demo object internals:
OFFSET  SIZE          TYPE DESCRIPTION           VALUE
      0    4          (object header)          d8 f0
d5 02 (11011[000] 11110000 11010101 00000010) (47575256)
      4    4          (object header)          00 00
00 00 (00000000 00000000 00000000 00000000) (0)
      8    4          (object header)          05 c1
00 f8 (00000101 11000001 00000000 11111000) (-134168315)
     12    4  java.lang.Object Demo.o
(Object)
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

Process finished with exit code 0

```

这里很多同学会有疑惑，老师，你不是说锁的升级是基于线程竞争情况，来实现从偏向锁到轻量级锁再到重量级锁的升级的吗？可是为什么这里明明没有竞争，它的锁的标记是轻量级锁呢？

## 偏向锁的获取及原理

默认情况下，偏向锁的开启是有个延迟，默认是4秒。为什么这么设计呢？

因为JVM虚拟机自己有一些默认启动的线程，这些线程里面有很多的Synchronized代码，这些Synchronized代码启动的时候就会触发竞争，如果使用偏向锁，就会造成偏向锁不断的进行锁的升级和撤销，效率较低。

通过下面这个JVM参数可以将延迟设置为0.

-XX:BiasedLockingStartupDelay=0

再次运行下面的代码。

```

public class Demo {
    Object o=new Object();
    public static void main(String[] args) {
        Demo demo=new Demo(); //o这个对象，在内存中是如何存储和布局的。
        System.out.println(classLayout.parseInstance(demo).toPrintable());
        synchronized (demo){
            System.out.println(classLayout.parseInstance(demo).toPrintable());
        }
    }
}

```

得到如下的对象布局，可以看到对象头中的高位第一个字节最后三位数为[101]，表示当前为偏向锁状态。

这里的第一个对象和第二个对象的锁状态都是101，是因为偏向锁打开状态下，默认会有配置匿名的对象获得偏向锁。

| com.gupaoedu.pb.Demo object internals: |      |                                  |
|--|------|----------------------------------|
| OFFSET                                 | SIZE | TYPE DESCRIPTION           VALUE |
|  |      |                                  |

```

        0      4          (object header)          05 00
00 00 (00000101 00000000 00000000 00000000) (5)
        4      4          (object header)          00 00
00 00 (00000000 00000000 00000000 00000000) (0)
        8      4          (object header)          05 c1
00 f8 (00000101 11000001 00000000 11111000) (-134168315)
        12     4  java.lang.Object Demo.o
(object)
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

com.gupaoedu.pb.Demo object internals:
  OFFSET  SIZE   TYPE DESCRIPTION          VALUE
    0      4          (object header)          05 30
  4a  03 (00000101 00110000 01001010 00000011) (55193605)
        4      4          (object header)          00 00
00 00 (00000000 00000000 00000000 00000000) (0)
        8      4          (object header)          05 c1
00 f8 (00000101 11000001 00000000 11111000) (-134168315)
        12     4  java.lang.Object Demo.o
(object)
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

```

## 重量级锁的获取

在竞争比较激烈的情况下，线程一直无法获得锁的时候，就会升级到重量级锁。

仔细观察下面的案例，通过两个线程来模拟竞争的场景。

```

public static void main(String[] args) {
    Demo testDemo = new Demo();
    Thread t1 = new Thread(() -> {
        synchronized (testDemo){
            System.out.println("t1 lock ing");

            System.out.println(classLayout.parseInstance(testDemo).toPrintable());
        }
    });
    t1.start();
    synchronized (testDemo){
        System.out.println("main lock ing");
        System.out.println(classLayout.parseInstance(testDemo).toPrintable());
    }
}

```

从结果可以看出，在竞争的情况下锁的标记为 [010]，其中所标记 [10] 表示重量级锁

```

com.gupaoedu.pb.Demo object internals:
  OFFSET  SIZE   TYPE DESCRIPTION          VALUE
    0      4          (object header)          8a 20 5e 26
(10001010 00100000 01011110 00100110) (643702922)
        4      4          (object header)          00 00 00 00
(00000000 00000000 00000000 00000000) (0)
        8      4          (object header)          05 c1 00 f8
(00000101 11000001 00000000 11111000) (-134168315)

```

```

12      4      (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

t1 lock ing
com.gupaoedu.pb.Demo object internals:
OFFSET  SIZE   TYPE DESCRIPTION                                VALUE
0        4     (object header)                               8a 20 5e 26
(10001010 00100000 01011110 00100110) (643702922)
4        4     (object header)                               00 00 00 00
(00000000 00000000 00000000 00000000) (0)
8        4     (object header)                               05 c1 00 f8
(00000101 11000001 00000000 11111000) (-134168315)
12      4      (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

```

## CAS

CAS这个在Synchronized底层用得非常多，它的全称有两种

- Compare and swap
- Compare and exchange

就是比较并交换的意思。它可以保证在多线程环境下对于一个变量修改的原子性。

CAS的原理很简单，包含三个值当前内存值(V)、预期原来的值(E)以及期待更新的值(N)。

