线程安全I

2024年4月3日 11:27

基本使用回顾

```
public class TicketDemo {

    static Object lock = new Object();
    int ticketNum = 10;
    public void getTicket() {
        synchronized (lock){
            if (ticketNum <= 0) {
                return;
            }
            System.out.println(Thread.currentThread().getName() + "抢到一张票,剩余:" + ticketNum);
            // 非原子性操作
            ticketNum--;
        }
    }

    public static void main(String[] args) {
        TicketDemo ticketDemo = new TicketDemo();
        for (int i = 0; i < 20; i++) {
            new Thread(() -> {
                ticketDemo.getTicket();
            }).start();
        }
    }

}
```

Synchronized【对象锁】采用互斥的方式让同一时刻至多只有一个线程能持有【对象锁】，其它线程再想获取这个【对象锁】时就会阻塞住

Monitor

```
public class SyncTest {

    static final Object lock = new Object();
    static int counter = 0;
    public static void main(String[] args) {
        synchronized (lock) {
            counter++;
        }
    }
}
```

javap -v xx.class 查看class字节码信息

class反汇编

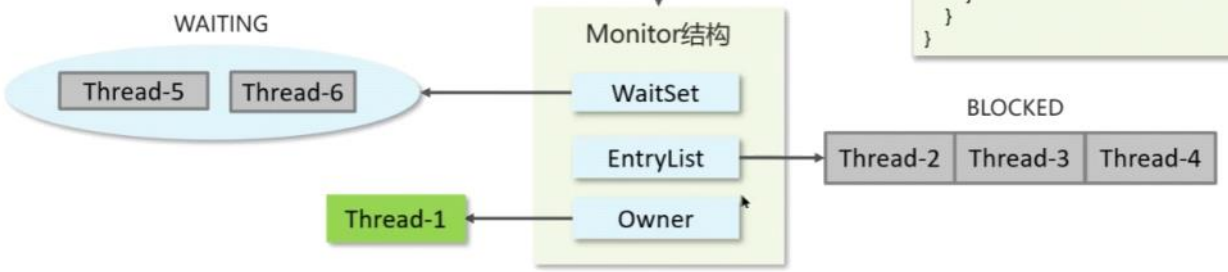
```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=3, args_size=1
       0: getstatic     #2                  // Field lock:Ljava/lang/Object;
       3: dup
       4: astore_1
       5: monitorenter  上锁（对象锁）
       6: getstatic     #3                  // Field counter:I
       9: iconst_1
      10: iadd
      11: putstatic     #3                  // Field counter:I
      14: aload_1
      15: monitorexit   解锁（对象锁）
      16: goto          24
      19: astore_2
      20: aload_1
      21: monitorexit   解锁（对象锁）
      22: aload_2
      23: athrow
      24: return
```

Monitor

Monitor 被翻译为监视器，是由jvm提供，c++语言实现

```
public class SyncTest {

    static final Object lock = new Object();
    static int counter = 0;
    public static void main(String[] args) {
        synchronized (lock) {
            counter++;
        }
    }
}
```



- Owner：存储当前获取锁的线程的，只能有一个线程可以获取
- EntryList：关联没有抢到锁的线程，处于Blocked状态的线程
- WaitSet：关联调用了wait方法的线程，处于Waiting状态的线程

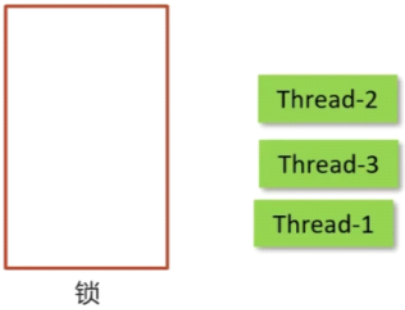
synchronized关键字的底层原理

- Synchronized【对象锁】采用互斥的方式让同一时刻至多只有一个线程能持有【对象锁】
- 它的底层由monitor实现的，monitor是jvm级别的对象（C++实现），线程获得锁需要使用对象（锁）关联monitor
- 在monitor内部有三个属性，分别是owner、entrylist、waitset
- 其中owner是关联的获得锁的线程，并且只能关联一个线程；entrylist关联的是处于阻塞状态的线程；waitset关联的是处于Waiting状态的线程

synchronized关键字的底层原理-进阶

Monitor实现的锁属于重量级锁，你了解过锁升级吗?

- Monitor实现的锁属于重量级锁，里面涉及到了用户态和内核态的切换、进程的上下文切换，成本较高，性能比较低。
- 在JDK 1.6引入了两种新型锁机制：偏向锁和轻量级锁，它们的引入是为了解决在没有多线程竞争或基本没有竞争的场景下因使用传统锁机制带来的性能开销问题。



对象的内存结构

在HotSpot虚拟机中，对象在内存中存储的布局可分为3块区域：对象头（Header）、实例数据（Instance Data）和对齐填充

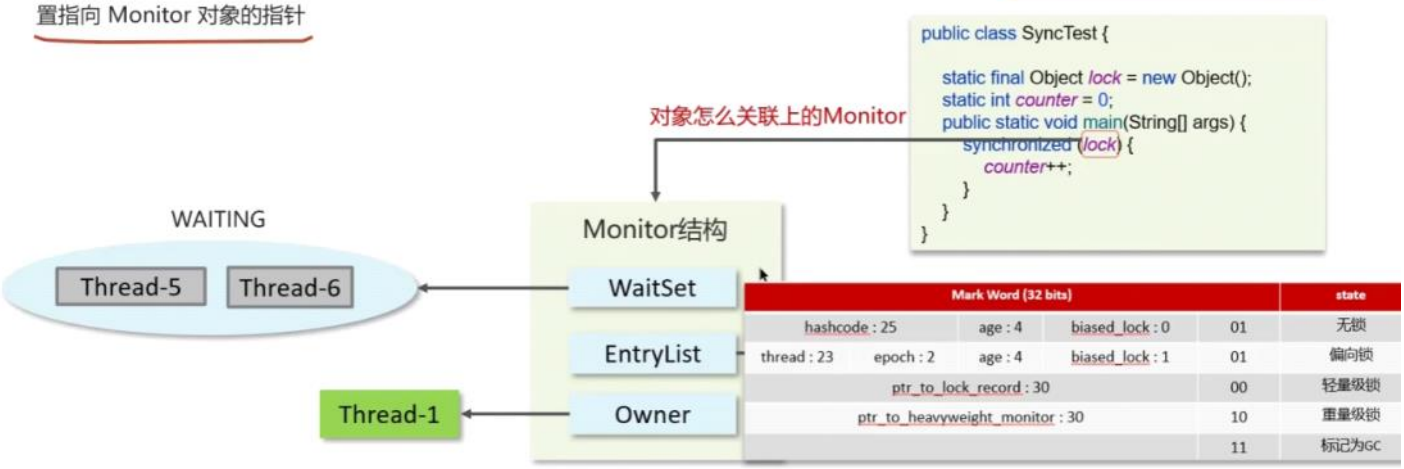
MarkWord

Mark Word (32 bits)				state
hashcode : 25	age : 4	biased_lock : 0	01	无锁
thread : 23	epoch : 2	age : 4	biased_lock : 1	01	偏向锁
ptr_to_lock_record : 30			00	轻量级锁
ptr_to_heavyweight_monitor : 30			10	重量级锁
			11	标记为GC

- hashcode：25位的对象标识Hash码　　lock标识，占2位
- age：对象分代年龄占4位
- biased_lock：偏向锁标识，占1位，0表示没有开始偏向锁，1表示开启了偏向锁
- thread：持有偏向锁的线程ID，占23位
- epoch：偏向时间戳，占2位
- ptr_to_lock_record：轻量级锁状态下，指向栈中锁记录的指针，占30位
- ptr_to_heavyweight_monitor：重量级锁状态下，指向对象监视器Monitor的指针，占30位

Monitor重量级锁

每个 Java 对象都可以关联一个 Monitor 对象，如果使用 synchronized 给对象上锁（重量级）之后，该对象头的Mark Word 中就被设置指向 Monitor 对象的指针



轻量级锁

在很多的情况下，在Java程序运行时，同步块中的代码都是不存在竞争的，不同的线程交替的执行同步块中的代码。这种情况下，用重量级锁是没必要的。因此JVM引入了轻量级锁的概念。

```
static final Object obj = new Object();
public static void method1() {
   synchronized( obj ) {
      // 同步块 A
      method2();
   }
}
public static void method2() {
   synchronized( obj ) {
   // 同步块 B
   }
}
```

同一个线程锁重入

轻量级锁

加锁流程

1.在线程栈中创建一个Lock Record，将其obj字段指向锁对象。

2.通过CAS指令将Lock Record的地址存储在对象头的mark word中，如果对象处于无锁状态则修改成功，代表该线程获得了轻量级锁。

3.如果是当前线程已经持有该锁了，代表这是一次锁重入。设置Lock Record第一部分为null，起到了一个重入计数器的作用。

4.如果CAS修改失败，说明发生了竞争，需要膨胀为重量级锁。

解锁过程

1.遍历线程栈,找到所有obj字段等于当前锁对象的Lock Record。

2.如果Lock Record的Mark Word为null，代表这是一次重入，将obj设置为null后continue。

3.如果Lock Record的 Mark Word不为null，则利用CAS指令将对象头的mark word恢复成为无锁状态。如果失败则膨胀为重量级锁。

偏向锁

轻量级锁在没有竞争时（就自己这个线程），每次重入仍然需要执行 CAS 操作。

Java 6 中引入了偏向锁来做进一步优化：只有第一次使用 CAS 将线程 ID 设置到对象的 Mark Word 头，之后发现这个线程 ID 是自己的就表示没有竞争，不用重新 CAS。以后只要不发生竞争，这个对象就归该线程所有

```
static final Object obj = new Object();
public static void m1 () {
    synchronized (obj) {
        // 同步块 A
        m2();
    }
}
public static void m2 () {
    synchronized (obj) {
        // 同步块 B
        m3();
    }
}
public static void m3 () {
    synchronized (obj) {

    }
}
```

Monitor实现的锁属于重量级锁，你了解过锁升级吗?

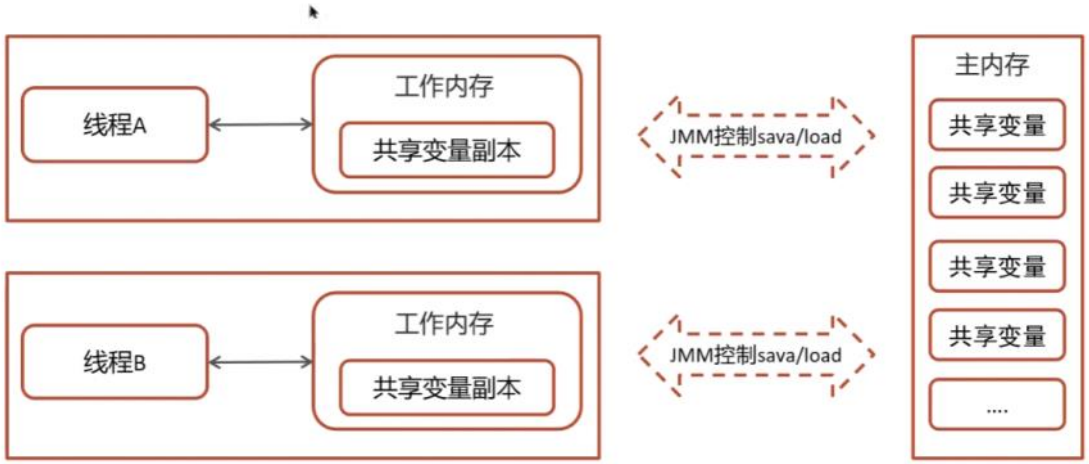
Java中的synchronized有偏向锁、轻量级锁、重量级锁三种形式，分别对应了锁只被一个线程持有、不同线程交替持有锁、多线程竞争锁三种情况。

	描述
重量级锁	底层使用的Monitor实现，里面涉及到了用户态和内核态的切换、进程的上下文切换，成本较高，性能比较低。
轻量级锁	线程加锁的时间是错开的（也就是没有竞争），可以使用轻量级锁来优化。轻量级修改了对象头的锁标志，相对重量级锁性能提升很多。每次修改都是CAS操作，保证原子性
偏向锁	一段很长的时间内都只被一个线程使用锁，可以使用了偏向锁，在第一次获得锁时，会有一个CAS操作，之后该线程再获取锁，只需要判断mark word中是否是自己的线程id即可，而不是开销相对较大的CAS命令

一旦锁发生了竞争，都会升级为重量级锁

Java 内存模型

JMM(Java Memory Model)Java内存模型，定义了共享内存中多线程程序读写操作的行为规范，通过这些规则来规范对内存的读写操作从而保证指令的正确性



你谈谈 JMM（Java内存模型）

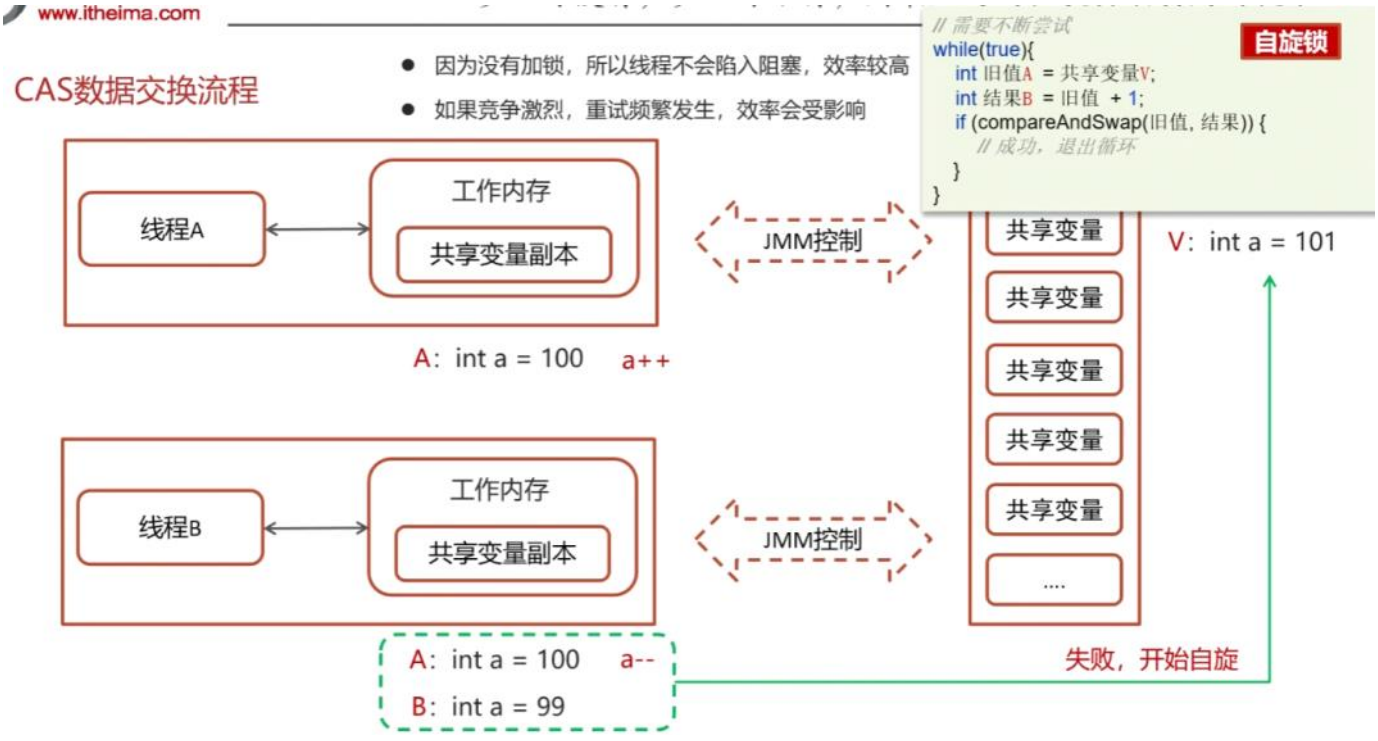
- JMM(Java Memory Model)Java内存模型，定义了共享内存中多线程程序读写操作的行为规范，通过这些规则来规范对内存的读写操作从而保证指令的正确性
- JMM把内存分为两块，一块是私有线程的工作区域（工作内存），一块是所有线程的共享区域（主内存）
- 线程跟线程之间是相互隔离，线程跟线程交互需要通过主内存

CAS

CAS的全称是： Compare And Swap(比较再交换)，它体现的一种乐观锁的思想，在无锁情况下保证线程操作共享数据的原子性。

在JUC（java.util.concurrent）包下实现的很多类都用到了CAS操作

- AbstractQueuedSynchronizer（AQS框架）
- AtomicXXX类



CAS 底层实现

CAS 底层依赖于一个 Unsafe 类来直接调用操作系统底层的 CAS 指令

ReentrantLock中的一段CAS代码

```
protected final boolean compareAndSetState(int expect, int update) {
    return STATE.compareAndSet(this, expect, update);
}
```

当前值　　期望的值　　更新后的值

```
// 需要不断尝试
while(true){
    int 旧值A = 共享变量V;
    int 结果B = 旧值 + 1;
    if (compareAndSwap(旧值, 结果)) {
        // 成功，退出循环
    }
}
```

乐观锁和悲观锁

- CAS 是基于乐观锁的思想：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
- synchronized 是基于悲观锁的思想：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。

```
// 需要不断尝试
while(true){
    int 旧值A = 共享变量V;
    int 结果B = 旧值 + 1;
    if (compareAndSwap(旧值, 结果)) {
        // 成功，退出循环
    }
}
```

CAS 你知道吗?

- CAS的全称是： Compare And Swap(比较再交换);它体现的一种乐观锁的思想，在无锁状态下保证线程操作数据的原子性。
- CAS使用到的地方很多：AQS框架、AtomicXXX类
- 在操作共享变量的时候使用的自旋锁，效率上更高一些
- CAS的底层是调用的Unsafe类中的方法，都是操作系统提供的，其他语言实现

乐观锁和悲观锁的区别

- CAS 是基于乐观锁的思想：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
- synchronized 是基于悲观锁的思想：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。

请谈谈你对 volatile 的理解

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

① 保证线程间的可见性

② 禁止进行指令重排序

保证线程间的可见性

用 volatile 修饰共享变量，能够防止编译器等优化发生，让一个线程对共享变量的修改对另一个线程可见

```
static boolean stop = false;
public static void main(String[] args) {
    new Thread(() -> {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        stop = true;
        System.out.println(Thread.currentThread().getName()+"：  modify stop to true...");
    },"t1").start();

    new Thread(() -> {
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()+"：  "+stop);
    },"t2").start();

    new Thread(() -> {
        int i = 0;
        while (!stop) {
            i++;
        }
        System.out.println("stopped... c:"+ i);
    },"t3").start();
}
```

请谈谈你对 volatile 的理解

问题分析：主要是因为在JVM虚拟机中有一个JIT（即时编译器）给代码做了优化。

```
while (!stop) {
    i++;
}
```

优化

```
while (true) {
    i++;
}
```

解决方案一：在程序运行的时候加入vm参数-Xint表示禁用即时编译器，不推荐，得不偿失（其他程序还要使用）

解决方案二：在修饰stop变量的时候加上volatile,当前告诉 jit，不要对 volatile 修饰的变量做优化

volatile禁止指令重排序

用 volatile 修饰共享变量会在读、写共享变量时加入不同的屏障，阻止其他读写操作越过屏障，从而达到阻止重排序的效果

```
int x;
int y;

@Actor
public void actor1() {
    x = 1;
    y = 1;
}

@Actor
public void actor2(II_Result r) {
    r.r1 = y;
    r.r2 = x;
}
```

情况一：先执行actor2获取结果 ⟹ 0,0

情况二：先执行actor1中的第一行代码，然后执行actor2获取结果 ⟹ 0,1

情况三：先执行actor1中所有代码，然后执行actor2获取结果 ⟹ 1,1

情况四：先执行actor1中第二行代码，然后执行actor2获取结果 ⟹ 1,0

已经发生了指令重排序

注解@Actor保证方法内的代码在同一个线程下执行

volatile禁止指令重排序

在变量上添加volatile，禁止指令重排序，则可以解决问题

```
int x;
volatile int y;

@Actor
public void actor1() {
    x = 1;
    y = 1;
}

@Actor
public void actor2(II_Result r) {
    r.r1 = y;
    r.r2 = x;
}
```

volatile禁止指令重排序

在变量上添加volatile，禁止指令重排序，则可以解决问题

```
volatile int x;
int y;

@Actor
public void actor1() {
    x = 1;
    y = 1;
}

@Actor
public void actor2(II_Result r) {
    r.r1 = y;
    r.r2 = x;
}
```

volatile使用技巧：

- 写变量让volatile修饰的变量的在代码最后位置
- 读变量让volatile修饰的变量的在代码最开始位置

1. 请谈谈你对 volatile 的理解

①保证线程间的可见性

用 volatile 修饰共享变量，能够防止编译器等优化发生，让一个线程对共享变量的修改对另一个线程可见

② 禁止进行指令重排序

指令重排：用 volatile 修饰共享变量会在读、写共享变量时加入不同的屏障，阻止其他读写操作越过屏障，从而达到阻止重排序的效果