链滴

Java 并发编程（二）线程同步

作者：wlgzs-sjl

原文链接：https://ld246.com/article/1574332216314

来源网站：链滴

synchronized关键字

1、对某个对象加锁

```
public class T {

    private int count = 10;
    private final Object lock = new Object();

    public void m() {
        synchronized (lock) { // 任何线程要执行下面的代码，都必须先拿到lock锁，锁信息记录在堆存对象中的，不是在栈引用中
            // 如果lock已经被锁定，其他线程再进入时，就会进行阻塞等待
            // 所以 synchronized 是互斥锁
            count--;
            System.out.println(Thread.currentThread().getName() + " count = " + count);
        }
        // 当代码块执行完毕后，锁就会被释放，然后被其他线程获取
    }
}
```

2、每次使用锁都要newObject，比较麻烦，可以使用this代替objec锁

```
public class T {

    private int count = 10;

    public void m() {
        synchronized (this) { // 任何线程要执行下面的代码，必须先拿到this锁
```

```
            // synchronized 锁定的不是代码块，而是 this 对象
            count--;
            System.out.println(Thread.currentThread().getName() + " count = " + count);
        }
    }
}
```

3、若整个方法内所有代码都被synchronized修饰,则可以使synchronized关键字修饰整个方法.

```
public class T {

    private int count = 10;

    public synchronized void m() { // 等同于 synchronized (this) {
        count--;
        System.out.println(Thread.currentThread().getName() + " count = " + count);
    }

}
```

4、若synchronized关键字锁定静态方法,等价于锁定T.class对象

```
public class T {

    private static int count = 10;

    public static synchronized void m() { // 等同于 synchronized (package.T.class) {
        count--;
        System.out.println(Thread.currentThread().getName() + " count = " + count);
    }

    public static void mm(){
        synchronized (T.class){//这里不可以写this
            count--;
            System.out.println(Thread.currentThread().getName() + " count = " + count);
        }
    }
}
```

synchronized关键字的使用

使用synchronized关键字修饰代码块,保证synchronized代码块内作的原子性

```
public class T implements Runnable{

    private int count = 10;

    @Override
    public /*synchronized*/ void run() {
        count--;
```

```
        System.out.println(Thread.currentThread().getName() + " count = " + count);
    }

    public static void main(String[] args) {
        T t = new T();
        for (int i = 0; i < 5; i++) {
            new Thread(t,"THREAD").start();
        }
    }
}
```

不加synchronized关键字,程序输出如下: 因为不保证原子性,每个线程在执行自减操作和输出操作之间可能被其它线程打断.

```
Thread-0 count = 7
Thread-4 count = 5
Thread-3 count = 6
Thread-2 count = 7
Thread-1 count = 7
```

加上synchronized关键字,程序输出如下:

```
Thread-0 count = 9
Thread-4 count = 8
Thread-3 count = 7
Thread-2 count = 6
Thread-1 count = 5
```

同步方法和非同步方法可以同时调用

```
public class T {

    public synchronized void m1() {
        System.out.println(Thread.currentThread().getName() + " m1 start");
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " m1 end");
    }

    public void m2() {
        System.out.println(Thread.currentThread().getName() + " m2 start");
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " m2 end");
    }

    public static void main(String[] args) {
        T t = new T();
```

```
        new Thread(t::m1).start();
        new Thread(t::m2).start();
    }
}
```

程序输出如下：

```
Thread-0 m1 start
Thread-1 m2 start
Thread-1 m2 end
Thread-0 m1 end
```

对业务写方法加锁，而对业务读方法不加锁，容易出现脏读问题

因为在执行写的过程中，读操作没有加锁，所以读会读取到写未改完的脏数据。所以需要给读写都加锁

```
public class Account {

    /**
     * 银行账户名称
     */
    String name;
    /**
     * 银行账余额
     */
    double balance;

    public synchronized void set(String name, double balance) {
        this.name = name;
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.balance = balance;
    }

    public /*synchronized*/ double getBalance() {
        return this.balance;
    }

    public static void main(String[] args) {
        Account a = new Account();
        new Thread(() -> a.set("张三", 100.0)).start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(a.getBalance()); // 0.0  加锁后100.0

        try {
```

```
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(a.getBalance()); // 100.0  加锁后100.0
    }
}
```

synchronized 是可重入锁

即一个同步方法可以调用另外一个同步方法，一个线程已经拥有某个对象的锁，再次申请时仍然会得该对象的锁

```
public class T {

    synchronized void m1() {
        System.out.println("m1 start ");
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        m2();
    }

    synchronized void m2() {
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(" m2"); // 这句话会打印，调用m2时，不会发生死锁
    }
}
```

子类调用父类的同步方法，也是可重入的

```
public class T {

    synchronized void m() {
        System.out.println("m start ");
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("m end ");
    }

    public static void main(String[] args) {
        new TT().m();
    }
```

```
}

class TT extends T {
    @Override
    synchronized void m() {
        System.out.println("child m start ");
        super.m();
        System.out.println("child m end ");
    }
}
```

程序运行结果如下：

```
child m start
m start
m end
child m end
```

synchronized 代码块中，如果发生异常，锁会被释放

在并发处理过程中，有异常要多加小心，不然可能发生数据不一致的情况。

比如，在一个web app处理过程中，多个servlet线程共同访问同一资源，这时如果异常处理不合适，一个线程抛出异常，其他线程就会进入同步代码区，有可能访问到异常产生的数据。

因此要非常小心处理同步业务逻辑中的异常。

```
public class T {

    int count = 0;

    synchronized void m() {
        System.out.println(Thread.currentThread().getName() + " start");
        while (true) {
            count++;
            System.out.println(Thread.currentThread().getName() + " count=" + count);
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            if (count == 5) {  // 当count == 5 时，synchronized代码块会抛出异常
                int i = 1 / 0;
            }
        }
    }

    public static void main(String[] args) {
        T t = new T();
        Runnable r = new Runnable() {
            @Override
            public void run() {
                t.m();
            }
```

```
        };
        new Thread(r, "t1").start(); // 执行到第5秒时，抛出 ArithmeticException
        // 如果抛出异常后，t2 会继续执行，就代表t2拿到了锁，即t1在抛出异常后释放了锁

        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        new Thread(r, "t2").start();
    }

}
```

程序运行结果如下：

```
t1 start
t1 count=1
t1 count=2
t1 count=3
t1 count=4
t1 count=5
t2 start
Exception in thread "t1" java.lang.ArithmeticException: / by zero
t2 count=6
    at c_011.T.m(T.java:29)
    at c_011.T$1.run(T.java:39)
    at java.lang.Thread.run(Thread.java:748)
t2 count=7
t2 count=8
t2 count=9
t2 count=10
```

synchronized锁住的是堆中o对象的实例,而不是o对象的引用,因为synchronized是针对堆中o对象的实例上进行计数

1. 若在程序运行过程中,引用o指向对象的属性发生改变,锁状态不变.
2. 若在程序运行过程中,引用o指向的对象发生改变,则锁状态改变,原本抢到的锁作废,线程会去抢新锁.

因此实际编程中常将锁对象的引用用final修饰,保证其指向的锁对象不发生改变.(final修饰引用时,该引所指向的属性可以改变,但该引用不能再指向其他对象)

```
public class T {

    Object o = new Object();

    // 该方法锁住的o对象引用没有被设为final
    void m() {
        synchronized (o) {
            while (true) {
                System.out.println(Thread.currentThread().getName() + "正在运行");
                try {
```

```
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public static void main(String[] args) {
    T t = new T();
    new Thread(t::m, "线程1").start();

    // 在这里让程序睡一会儿,保证两个线程得到的o对象不同
    try {
        TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    Thread thread2 = new Thread(t::m, "线程2");

    // 改变锁引用,使得线程2也有机会运行,否则一直都是线程1运行
    t.o = new Object();
    thread2.start();
}
}
```

程序输出如下,看到主线程睡了3秒之后,线程1和线程2交替运行,他们各自抢到了不同的锁.

```
线程1正在运行
线程1正在运行
线程1正在运行
线程2正在运行
线程1正在运行
线程2正在运行
线程1正在运行
线程2正在运行
线程1正在运行
线程2正在运行
...
```

不要以字符串常量作为锁定对象: 因为字符串常量池的存在,两个不同字符串引用可能指向同一字符串对象

```
public class T {

    // 两个字符串常量,作为两同步方法的锁
    String s1 = "Hello";
    String s2 = "Hello";

    // 同步m1方法以s1为锁
    void m1() {
```

```
        synchronized (s1) {
            while (true) {
                System.out.println(Thread.currentThread().getName() + ":m1 is running");
            }
        }
    }

    // 同步m2方法以s2为锁
    void m2() {
        synchronized (s2) {
            while (true) {
                System.out.println(Thread.currentThread().getName() + ":m1 is running");
            }
        }
    }

    public static void main(String[] args) {
        T t = new T();

        // 输出两个锁的哈希码
        System.out.println(t.s1.hashCode());
        System.out.println(t.s2.hashCode());

        new Thread(t::m1, "线程1").start();
        new Thread(t::m2, "线程2").start();
    }
}
```

程序执行结果如下,我们发现两个字符串常量指向的是同一对象,且有一个线程永远得不到锁. 若我们的序与某个库使用了同一个字符串对象作为锁,就会出现难以发现的bug.

```
69609650
69609650
线程1:m1 is running
线程1:m1 is running
线程1:m1 is running
线程1:m1 is running
线程1:m1 is running
线程1:m1 is running
```

synchronized方法和非synchronized方法可以同时执行,因为非synchronized方法不需要抢这把锁

volatile关键字

volatile关键字向编译器声明该变量是易变的,每次对volatile关键字的修改会通知给所有相关进程.

1. 要理解volatile关键字的作用,要先理解Java内存模型JMM

● 在JMM中,所有对象以及信息都存放在主内存中(包含堆,栈),而每个线程在CPU中都有自己的独立空间存储了需要用到的变量的副本.

● 线程对共享变量的操作,都会先在自己CPU中的工作内存中进行,然后再同步给主内存.若不加volatile键字修饰,每个线程都有可能从自己CPU中的工作内存读取内存;而加以volatile关键字修饰后,每个线程

该变量进行修改后都会马上通知给所有进程.

```
public class T {

    /*volatile*/ boolean running = true; // 若无volatile关键字修饰,则变量running难以在每个线程间共享,对running变量的修改自然不能终止线程

    // 可以通过将running变量设为false来终止m()方法
    void m() {
        System.out.println("m start");
        while (running) {
            // 死循环
        }
        System.out.println("m end");
    }

    public static void main(String[] args) {
        T t = new T();
        new Thread(t::m, "t1").start();
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 将running变量设为false,观察线程是否被终止
        t.running = false;
    }
}
```

我们发现,若不对running变量加以volatile修饰,则对running变量的修改不能终止子线程,说明在主线中对running的修改对子线程不可见.

有趣的是,若在while死循环体中加入一些语句之后,可见性问题可能会消失,这是因为加入语句后,CPU可能会出现空闲,并同步主内存中的内容到工作内存,但这是不确定的,因此在这种情况下还是尽量要加上olatile

2. volatile只能保证可见性,但不能保证原子性. volatile不能解决多个线程同时修改一个变量带来的线安全问题.

```
public class T {

    volatile int count = 0;
    /*AtomicInteger count = new AtomicInteger(0);*/

    /*synchronized*/ void m() {
        for (int i = 0; i < 10000; i++) {
            count++;
            /*count.incrementAndGet();*/
        }
    }

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

```
        // 创建一个10个线程的list，执行任务皆是 m方法
        T t = new T();
        List<Thread> threads = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            threads.add(new Thread(t::m, "t-" + i));
        }

        // 启动这10个线程
        threads.forEach(Thread::start);

        // join 到主线程，防止主线程先行结束
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // 10个线程，每个线程执行10000次，结果应为 100000
        System.out.println(t.count);
    }

}
```

运行该程序,我们发现最终变量t.count并非如我们所预计的那样为100000,而是小于100000(当然,若掉volatile修饰,最终t.count会更小).这说明volatile并不能保证对变量操作的原子性.

要保证多线程操作同一变量的原子性,有如下两种方法:

1. 在方法上加synchronized修饰,synchronized既保证可见性,又保证原子性.但synchronized效率最低.
2. 使用AtomicInteger代替int类型(AtomicXXX类可以用来替代基本数据类型,其支持一些原子操作).

综上所述,volatile保证对被修饰变量的修改对于其他相关线程是可见的,即保证了可见性;但volatile并能解决多个线程同时修改同一变量带来的线程安全问题,即不能保证原子性. 因此,只有在满足以下两个件的情况下volatile才能保证解决线程的安全问题:

1. 运算结果并不依赖变量的当前值,或者能够确保只有单一的线程修改变量的值。
2. 变量不需要与其他状态变量共同参与不变约束