- 前言
- synchronized的三种应用方式
  - synchronized作用于实例方法
  - synchronized作用于静态方法
  - synchronized同步代码块
- synchronized禁止指令重排分析
- synchronized的可重入性
- 总结
- 学习交流

> 主要讲解synchronized的应用方式和内存语义。

前言

看这篇文章前，建议大家先看我前面的文章《Java并发编程系列1-基础知识》，否则里面的相关知识看不懂，特别是并发编程相关的可见性、有序性，以及内存模型JMM等。

在Java中，关键字synchronized可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块(主要是对方法或者代码块中存在共享数据的操作)，同时我们还应该注意到synchronized另外一个重要的作用，synchronized可保证一个线程的变化(主要是共享数据的变化)被其他线程所看到（保证可见性，完全可以替代Volatile功能）。

synchronized的三种应用方式

synchronized关键字最主要有以下3种应用方式，下面分别介绍：

- 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；
- 修饰静态方法，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；
- 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。

synchronized作用于实例方法

所谓的实例对象锁就是用synchronized修饰实例对象中的实例方法，注意是实例方法不包括静态方法，如下：

```
public class AccountingSync implements Runnable {
    //共享资源(临界资源)
    static int i = 0;
    // synchronized 修饰实例方法
    public synchronized void increase() {
        i ++;
    }
    @Override
    public void run() {
        for(int j=0;j<1000000;j++){
            increase();
        }
    }
    public static void main(String args[]) throws InterruptedException {
        AccountingSync instance = new AccountingSync();
        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("static, i output:" + i);
    }
}
/**
 * 输出结果:
 * static, i output:2000000
 */
```

如果在函数increase()前不加synchronized，因为i++不具备原子性，所以最终结果会小于2000000，具体分析可以参考文章《Java并发编程系列2-volatile》。下面这点非常重要：

> 一个对象只有一把锁，当一个线程获取了该对象的锁之后，其他线程无法获取该对象的锁，所以无法访问该对象的其他synchronized实例方法，但是其他线程还是可以访问该实例对象的其他非synchronized方法。

但是一个线程 A 需要访问实例对象 obj1 的 synchronized 方法 f1(当前对象锁是obj1)，另一个线程 B 需要访问实例对象 obj2 的 synchronized 方法 f2(当前对象锁是obj2)，这样是允许的：

```
public class AccountingSyncBad implements Runnable {
    //共享资源(临界资源)
    static int i = 0;
    // synchronized 修饰实例方法
    public synchronized void increase() {
        i ++;
    }
    @Override
    public void run() {
        for(int j=0;j<1000000;j++){
            increase();
        }
    }
    public static void main(String args[]) throws InterruptedException {
        // new 两个AccountingSync新实例
        Thread t1 = new Thread(new AccountingSyncBad());
        Thread t2 = new Thread(new AccountingSyncBad());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("static, i output:" + i);
    }
}
/**
 * 输出结果:
 * static, i output:1224617
 */
```

上述代码与前面不同的是我们同时创建了两个新实例AccountingSyncBad，然后启动两个不同的线程对共享变量i进行操作，但很遗憾操作结果是1224617而不是期望结果2000000，因为上述代码犯了严重的错误，虽然我们使用synchronized修饰了increase方法，但却new了两个不同的实例对象，这也就意味着存在着两个不同的实例对象锁，因此t1和t2都会进入各自的对象锁，也就是说t1和t2线程使用的是不同的锁，因此线程安全是无法保证的。

> 每个对象都有一个对象锁，不同的对象，他们的锁不会互相影响。

解决这种困境的的方式是将synchronized作用于静态的increase方法，这样的话，对象锁就当前类对象，由于无论创建多少个实例对象，但对于的类对象拥有只有一个，所有在这样的情况下对象锁就是唯一的。下面我们看看如何使用将synchronized作用于静态的increase方法。

synchronized作用于静态方法

> 当synchronized作用于静态方法时，其锁就是当前类的class锁，不属于某个对象。
>
> 当前类class锁被获取，不影响对象锁的获取，两者互不影响。

由于静态成员不专属于任何一个实例对象，是类成员，因此通过class对象锁可以控制静态成员的并发操作。需要注意的是如果一个线程A调用一个实例对象的非static synchronized方法，而线程B需要调用这个实例对象所属类的静态synchronized方法，不会发生互斥现象，因为访问静态synchronized方法占用的锁是当前类的class对象，而访问非静态synchronized方法占用的锁是当前实例对象锁，看如下代码：

```
public class AccountingSyncClass implements Runnable {
    static int i = 0;
    /**
     * 作用于静态方法,锁是当前class对象,也就是
     * AccountingSyncClass类对应的class对象
     */
    public static synchronized void increase() {
        i++;
    }
    // 非静态,访问时锁不一样不会发生互斥
    public synchronized void increase4Obj() {
        i++;
    }
    @Override
    public void run() {
        for(int j=0;j<1000000;j++){
            increase();
        }
    }
    public static void main(String[] args) throws InterruptedException {
        //new新实例
        Thread t1=new Thread(new AccountingSyncClass());
        //new新实例
        Thread t2=new Thread(new AccountingSyncClass());
        //启动线程
        t1.start();t2.start();
        t1.join();t2.join();
        System.out.println(i);
    }
}
/**
 * 输出结果:
 * 2000000
 */
```

由于synchronized关键字修饰的是静态increase方法，与修饰实例方法不同的是，其锁对象是当前类的class对象。注意代码中的increase4Obj方法是实例方法，其对象锁是当前实例对象，如果别的线程调用该方法，将不会产生互斥现象，毕竟锁对象不同，但我们应该意识到这种情况下可能会发现线程安全问题(操作了共享静态变量i)。

synchronized同步代码块

在某些情况下，我们编写的方法体可能比较大，同时存在一些比较耗时的操作，而需要同步的代码又只有一小部分，如果直接对整个方法进行同步操作，可能会得不偿失，此时我们可以使用同步代码块的方式对需要同步的代码进行包裹，这样就无需对整个方法进行同步操作了，同步代码块的使用示例如下：

```
public class AccountingSync2 implements Runnable {
    static AccountingSync2 instance = new AccountingSync2(); // 饿汉单例模式
    static int i=0;
    @Override
    public void run() {
        //省略其他耗时操作....
        //使用同步代码块对变量i进行同步操作,锁对象为instance
        synchronized(instance){
            for(int j=0;j<1000000;j++){
                i++;
            }
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread t1=new Thread(instance);
        Thread t2=new Thread(instance);
        t1.start();t2.start();
        t1.join();t2.join();
        System.out.println(i);
    }
}
/**
 * 输出结果:
 * 2000000
 */
```

从代码看出，将synchronized作用于一个给定的实例对象instance，即当前实例对象就是锁对象，每次当线程进入synchronized包裹的代码块时就会要求当前线程持有instance实例对象锁，如果当前有其他线程正持有该对象锁，那么新到的线程就必须等待，这样也就保证了每次只有一个线程执行i++;操作。当然除了instance作为对象外，我们还可以使用this对象(代表当前实例)或者当前类的class对象作为锁，如下代码：

```
//this,当前实例对象锁
synchronized(this){
    for(int j=0;j<1000000;j++){
        i++;
    }
}
//class对象锁
synchronized(AccountingSync.class){
    for(int j=0;j<1000000;j++){
        i++;
    }
}
```

synchronized禁止指令重排分析

> 指令重排的情况，可以参考文章《Java并发编程系列1-基础知识》

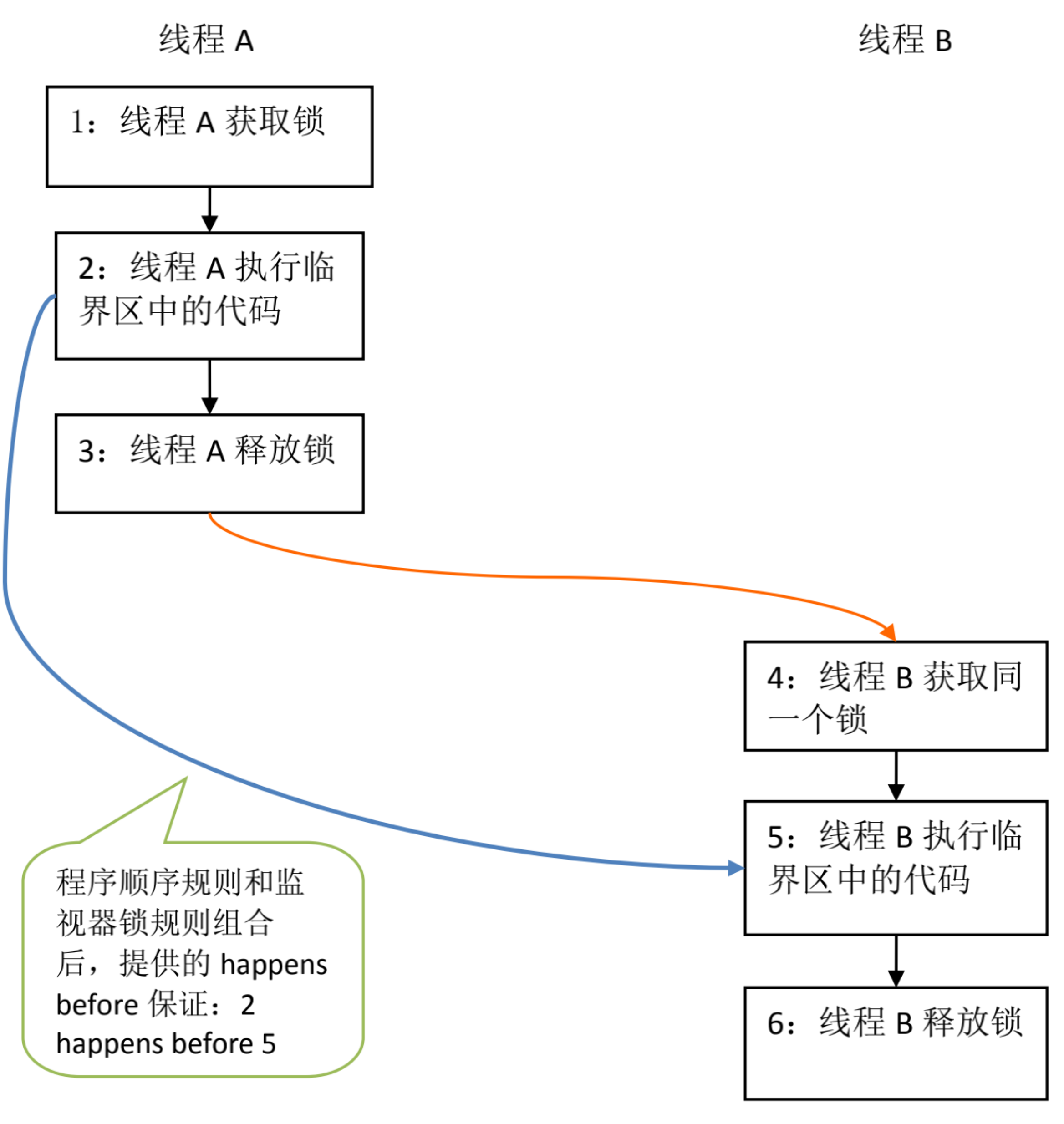
我们先看如下代码：

```
class MonitorExample {
    int a = 0;
    public synchronized void writer() {  //1
        a++;                             //2
    }                                    //3
    public synchronized void reader() {  //4
        int i = a;                       //5
        //……
    }                                    //6
}
```

假设线程A执行writer()方法，随后线程B执行reader()方法。根据happens before规则，这个过程包含的happens before关系可以分为两类：

- 根据程序次序规则，1 happens before 2, 2 happens before 3; 4 happens before 5, 5 happens before 6。
- 根据监视器锁规则，3 happens before 4。
- 根据happens before的传递性，2 happens before 5。

上述happens before 关系的图形化表现形式如下：



> 在上图中，每一个箭头链接的两个节点，代表了一个happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示监视器锁规则；蓝色箭头表示组合这些规则后提供的happens before保证。

上图表示在线程A释放了锁之后，随后线程B获取同一个锁。在上图中，2 happens before 5。因此，线程A在释放锁之前所有可见的共享变量，在线程B获取同一个锁之后，将立刻变得对B线程可见。

synchronized的可重入性

从互斥锁的设计上来说，当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁，请求将会成功。

synchronized就是可重入锁，因此一个线程调用synchronized方法的同时，在其方法体内部调用该对象另一个synchronized方法是允许的，如下：

```
public class AccountingSync implements Runnable{
    static AccountingSync instance=new AccountingSync();
    static int i=0;
    static int j=0;
    @Override
    public void run() {
        for(int j=0;j<1000000;j++){
            //this,当前实例对象锁
            synchronized(this){
                i++;
                increase();//synchronized的可重入性
            }
        }
    }
    public synchronized void increase(){
        j++;
    }
    public static void main(String[] args) throws InterruptedException {
        Thread t1=new Thread(instance);
        Thread t2=new Thread(instance);
        t1.start();t2.start();
        t1.join();t2.join();
        System.out.println(i);
    }
}
```

当前实例对象锁后进入synchronized代码块执行同步代码，并在代码块中调用了当前实例对象的另外一个synchronized方法，再次请求当前实例锁时，将被允许。需要特别注意另外一种情况，当子类继承父类时，子类也是可以通过可重入锁调用父类的同步方法。注意由于synchronized是基于monitor实现的，因此每次重入，monitor中的计数器仍会加1。

总结

该文章给大家讲解了synchronized的三种应用方式，指令重排情况分析，以及synchronized的可重入性，通过该文章，基本可以掌握synchronized的使用姿势，以及可能会遇到的坑。关于"线程中断与synchronized"的相关知识，因为篇幅原因就不写了，大家可以到网上查一下相关资料，进一步学习。

参考资料：
《深入理解Java内存模型》
《Java并发编程实战》

学习交流

可以扫下面二维码，关注「楼仔」公众号。

长按二维码，回复 **「加群」**，欢迎一起学习交流哈~~ 👏👏👏

> 尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。