



链滴

# Synchronized 关键字解析

作者: [sirwsl](#)

原文链接: <https://ld246.com/article/1596457398995>

来源网站: 链滴

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)



## 总结

JVM会通过monitor来进行加锁、解锁，从而保证有且仅有一个线程能够执行指定的代码，从而保证程的安全，同时还具有可重入和不可中断的性质。

## 简介

### 同步方法：

同步方法支持一种简单的策略来防止线程干扰和内存一致性错误：如果一个对象对多个线程可见，则对象变量的所有读取或写入都是通过同步方法完成的。

**synchronized**保证在同一时刻有且仅有一个线程在执行该段代码，从而保证并发的安全。

### synchronized地位

Synchronized 是java的关键字，被java语言原生支持，是最基本的互斥同步手段。

## 为什么学习

接下来我们来看一段代码：

```
public class test1 implements Runnable{
    //创建本类实例 ps：需要添加static关键字
    static test1 instance = new test1();

    static int count = 0;
    public static void main(String[] args) throws InterruptedException {
```

```

//两个线程共用test1实例的方法
Thread t1 = new Thread(instance);
Thread t2 = new Thread(instance);
//启动线程
t1.start();
t2.start();

//为了在两个线程执行完毕之后执行sout, 直到线程执行完毕才执行sout
//调用join方法, 让线程进入等待状态
t1.join();
t2.join();

System.out.println(count);
//计算结果不符合预期
}

//对象锁: synchronized 添加在普通方法 public synchronized void run(){}
//对象锁: 同步代码块 synchronized(this){}
//类锁: .class synchronized(test.class){}
//类锁: static形式 run(){method();} public static synchronized void method(){}
@Override
public void run() {
    for (int i = 0; i < 100000; i++) count++;
}
}
}

```

运行结果:

```

"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
140833

```

从代码和结果中可以看出, 虽然两个线程各执行100000次, 但是结果却没有200000, 那是为什么呢?

两个线程同时对count++操作, 其中包含的动作为:

1. 读取count
2. 执行count+1操作
3. 讲count的值写入内存中

而为什么会出现值比真实值小呢? 我们可以设想一下:

当t1在对count执行第二步+1操作时候 (count=11), 但是还没有写入内存中的时候, t2进来了 (取count=10), 这时候t1虽然执行+1操作 (count=11) 了, 但是t2读取到的仍然是之前的值 (count=10)。

出现上述情况我们称之为线程不安全。我们设想一下, 这是有关钱的计算, 是不是就会很严重了。

## 用法

**synchronized**关键字用法包括对象锁和类锁。其中对象锁包括方法锁和同步代码块; 类锁包括synchronized修饰静态方法和锁为Class对象。

# 对象锁

包括方法锁（默认锁对象为this当前实例对象）和同步代码块锁（自己指定锁对象）

## 代码块形式

手动指定锁对象

```
public class test2 implements Runnable{
    static test2 instance = new test2();
    Object lock1 = new Object();//锁对象
    Object lock2 = new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);
        t1.start();
        t2.start();

        //只要线程有存活就执行空的while循环
        while(t1.isAlive() || t2.isAlive()){

            System.out.println("end");

        }

        @Override
        public void run() {
            //synchronized保证串行执行，有且仅有一个线程执行
            /*synchronized (this){
                System.out.println("对象锁:" + Thread.currentThread().getName());//获取线程名字-> Thread.currentThread()获取当前线程
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("结束: " + Thread.currentThread().getName());
            }*/

            //当业务逻辑复杂时候需要自己定义锁对象
            synchronized (lock1){
                System.out.println("lock1锁:" + Thread.currentThread().getName());//获取线程名字-> Thread.currentThread()获取当前线程
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("lock1结束: " + Thread.currentThread().getName());
            }

            synchronized (lock2){
```

```
        System.out.println("lock2锁:" + Thread.currentThread().getName()); //获取线程名字->Thr
ad.currentThread()获取当前线程
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("lock2结束: " + Thread.currentThread().getName());
    }
}
}
```

## 方法锁形式

synchronized修饰的普通方法，锁对象默认为this

```
public class test3 implements Runnable{
    static test3 instance = new test3();

    public static void main (String[] args){
        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);

        t1.start();
        t2.start();

        while (t1.isAlive()|| t2.isAlive()){

            System.out.println("end");
        }

        @Override
        public void run() {
            method();
        }

        public synchronized void method(){
            System.out.println("对象锁方法修饰形式: " + Thread.currentThread().getName());

            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("对象锁修饰方法结束: " + Thread.currentThread().getName());
        }
    }
}
```

## 类锁

## synchronized修饰静态的方法或指定锁为Class对象

java类可能会有很多对象，但只有一个class对象。由于只有一个，不同的实例之间会有互斥，只能一个线程在同一时间访问被类锁修饰的方法。**所谓的类锁其实就是Class对象锁而已，（概念性存在帮助我们理解实例方法与静态方法区别）**

**效果：类锁只能在同一时间被一个对象拥有，其他对象竞争就会出现阻塞情况**

PS：即使是不同的Runnable实例，这个线程所对应的实例只有一个。（对象锁：不同的实例创建出，都可以一块运行，但是类锁只能一个运行）

## synchronized加在static方法上

```
public class test4 implements Runnable{
    static test4 instance1 = new test4();
    static test4 instance2 = new test4();

    public static void main (String[] args){
        Thread t1 = new Thread(instance1);
        Thread t2 = new Thread(instance2);

        t1.start();
        t2.start();

        while (t1.isAlive()|| t2.isAlive()){

            System.out.println("end");
        }

        @Override
        public void run() {
            method();
        }

        public static synchronized void method(){
            System.out.println("类锁static形式： "+Thread.currentThread().getName());

            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("类锁static结束： "+Thread.currentThread().getName());
        }
    }
}
```

## synchronized(\*.class)代码块

```
public class test5 implements Runnable{
    static test5 instance1 = new test5();
```

```

static test5 instance2 = new test5();

public static void main (String[] args){
    Thread t1 = new Thread(instance1);
    Thread t2 = new Thread(instance2);

    t1.start();
    t2.start();

    while (t1.isAlive()|| t2.isAlive()){

        System.out.println("end");
    }

    @Override
    public void run() {
        method();
    }

    public void method() {
        synchronized (test5.class){
            System.out.println("类锁.class形式: "+Thread.currentThread().getName());

            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("类锁.class结束: "+Thread.currentThread().getName());
        }
    }
}

```

---

## 性质

**可重入不可中断**

### 可重入

可重入也叫做递归锁。指的是同一线程的外层函数获得锁之后，内层函数可以直接再次获得锁。

**优点：**避免死锁、提升封装性

**粒度：**线程而非调用

```
public class test6 {
```

```

boolean flag = true;

public static void main(String[] args){
    test6 test = new test6();
    test.methond();

    test.methond1();
}
//一个方法是可重入的
private synchronized void methond() {
    System.out.println("进入methond,flag:" + flag);
    if(flag){
        flag = false;
        methond();
    }
}
//可重入不要求同一个方法
private synchronized void methond1() {
    System.out.println("进入methond1");
    methond2();
}

private synchronized void methond2() {
    System.out.println("进入methond2");
}

public synchronized void dosomething(){
    System.out.println("父类方法");
}
}

//可重入不要求同一个类
class testdo extends test6 {
    @Override
    public synchronized void dosomething() {
        System.out.println("子类方法");
        super.dosomething();
    }
    public static void main(String[] args){
        testdo t = new testdo();
        t.dosomething();
    }
}

```

## 不可中断

一旦这个锁已经被其他线程获得。如果当前线程还想获得，那么只能选择等待或者阻塞，直到别的线程释放这个锁。如果永远不释放锁，那么只能永远等待下去。

PS：相比之下，Lock可以拥有中断得能力，如果等待时间太长可以选择中断现在已经获得锁得线程行或者退出。



# 原理

加锁、可重入、可见性原理

## 加锁、释放锁得原理

现象：

每个类的实例对应一把锁，而每一个synchronized方法都必须获得调用该方法实例的锁，才能够执行否则进入阻塞状态，而方法一旦执行就独占这把锁，直到该方法返回或者抛出异常才释放锁，其他程序才能够获得。

原理：

每个java对象都可以用作一个实现同步的锁，这个锁被称为内置锁或者监视器锁，线程在进入同步代码之前会自动获得该锁，并且在退出时候会自动释放锁。

## monitorenter与monitorexit

通过反编译后可以看见两个关键字。

monitorenter(加锁，锁计数+1) monitorexit(解锁，锁计数-1)，类似PV操作。通常可以看见monitorexit数量多于monitorenter，因为在进入时候需要加锁，但是在退出时候可能是执行结束退出或者是常退出等。

**monitorenter:**

每个对象都与一个monitorenter相关联，而一个monitorenter的lock锁只能在同一时间被一个线程获得，一个线程在尝试获得于monitorenter想关联的所有权的时候，如果monitorenter的计数器为0，则线程会立刻获得锁，将计数器+1；如果获得monitorenter所有权后发生重入，则计数器依次+1。

如果monitorenter被其他线程所持有，则当前线程只能处于阻塞状态，直到monitorenter计数器为0，才能获得锁。

**monitorexit:**

前提是已经获得锁的所有权，在释放的过程中依次对计数器-1，直到计数器变为0，则线程不再拥有monitorenter的所有权（解锁），如果计数器不为0，说明之前为可重入的，则继续持有锁。

## 可重入原理：加锁次数计数器

一个线程拿到一把锁之后如果还想再次进入由这把锁控制的方法，它可直接进入，通过利用加锁次数计数器来实现。

每个对象自动含有一把锁，而JVM负责跟踪对象被加锁的次数，当线程第一次给对象加锁的时候，计数器变为1。每当这个相同的线程在此对象上再次获得锁的时候，计数器依次加一，每当任务离开时候，数递减。当计数为0的时候，锁被完全释放。

## 保证可见性的原理：内存模型

在java线程模型中：当线程A要与线程B进行通信时候，线程A会将本地内存A中自己修改过的变量放到主内存中，线程B通过到主内存中去读取。整个过程由JMM (java内存模型) 控制，从而提高可见性的保证。

内存模型中synchronized实现：

一个代码块或者方法被synchronized修饰，那么它在进入代码块得到锁之后，将从主内存中获取锁定对象的数据，在执行结束后，被锁住对象所做的任何修改，在锁被释放之前都要从线程内存写回到主存中，从而保证每次执行都是可靠的。

---

## Synchronized 的缺陷

### 效率低、不够灵活、无法预判是否成功获取锁

#### ● 效率低：

锁的释放情况少：当前线程只有在执行结束或者异常抛出才会释放锁，否则都不会。如果在等待IO或其他操作而阻塞了，那么其他线程则不能获得锁。

试图获得锁时不能设定超时：没有获得锁的线程只能一直等待锁释放，例如对于Lock，则可以设置超，超过时间则不再等待。

不能中断一个正在试图获得锁的线程：synchronized加锁后不能中断，但是对于Lock能够中断

#### ● 不够灵活：

加锁和释放锁的时机单一，因为每个锁仅能锁住由一个单一条件（某个对象），直到自己解开，但这很多时候是不够的，例如读写锁则在读的时候不加锁，在写的时候加锁，这样则更加灵活

#### ● 无法直到是否成功获取到锁：

如果使用synchronized没有办法提前知道他能否成功获取到锁，而造成不便，但是lock中就可尝试去取，如果获取到则对应一些逻辑，如果失败了则对应另一些逻辑

---

## 多线程访问同步方法的情况：

#### ● 两个线程同时访问一个对象的同步方法

会依次执行，锁生效

#### ● 两个线程同时访问两个对象的同步方法

他们之间不受干扰，因为锁对象不是同一个

#### ● 两个线程同时访问synchronized修饰的静态方法

会依次执行锁生效

#### ● 同时访问同步方法与非同步方法

非同步方法不受到影响，synchronized只是作用与修饰的方法中

#### ● 访问同一个对象的不同的普通同步方法

由于是synchronized拿到的是同一个this，所以他们没办法同时运行，出现串行执行情况

#### ● 同时访问静态的synchronized和非静态的synchronized方法

static synchronized锁住的对象是class对象，没有static修饰的非静态方法锁住的是这个对象实例本身his。由于锁住的是不同对象，所以会同时运行

#### ● 方法抛出异常后会释放锁吗？

在抛出异常后lock不会释放锁，需要在try...catch..final的final中或其他方法去释放

但是synchronized修饰的方法中抛出异常，java虚拟机中会自动释放锁

- 当在执行synchronized修饰的方法中去调用没有被synchronized修饰的方法是线程安全吗？不是线程安全，因为没有被synchronized修饰的方法，可以被多个线程同时访问。

总结：

**一把锁只能同时被一个线程获取，没有拿到锁的线程必须等待**

**每个实例都对应有自己的一把锁，不同实例之间互不影响；** 例如：锁对象是.class以及synchronized修饰的static方法时候，所有的对象共用同一把类锁

**无论是方法正常执行完毕或者方法抛出异常，都会释放锁**

## 常见问题

**synchronized使用注意点：**

- 锁对象不能为空：我们指定一个对象作为我们的锁对象，它必须是一个实例对象（new）或者使用他方法创建好的而不能是一个空对象，因为这些锁的信息是保存在对象头中的。
- 作用域不易过大：作用域过大后影响程序执行速度
- 避免死锁

**如何选择Lock和synchronized关键字：**

- 如果可以则不适用Lock也不适用synchronized，而是选择使用java.util.concurrent中的类。在使这些类的时候更加方便，不需要自己去做同步工作
- 如果synchronized在程序中适用，则优先适用synchronized关键字，这样可以减少要编写的代码从而减少出错的概率
- 如果特别需要Lock独有特性的时候才选择Lock

**多线程访问同步方法的各种具体情况（前8种）**

**多个线程等待同一个synchronized锁的时候，jvm如何选择下一个获取锁的是哪个线程？**

锁调度机制。对于synchronized内置锁，不同版本的JVM处理方式不同，blocked和running都有几。

**synchronized使得同时只有一个线程可以执行，性能较差，有什么办法可以提升性能？**

优化适用范围，或者转化为读写锁

**想要更加灵活的控制锁的获取和释放**

可以自己实现Lock接口