**Synchronized三种用法**

0.1562018.04.14 13:22:43字数 1844阅读 5697

首先我们了解到Java中的线程同步锁可以是任意对象。  
这里我们介绍synchronized的三种应用方式：

1.作用于实例方法，当前实例加锁，进入同步代码前要获得当前实例的锁；

2.作用于静态方法，当前类加锁，进去同步代码前要获得当前类对象的锁；

3.作用于代码块，这需要指定加锁的对象，对所给的指定对象加锁，进入同步代码前要获得指定对象的锁。

这三种应用方式接下来分别介绍

synchronized修饰实例方法（普通方法）

  使用时，作用范围为整个函数，这里所谓的实例锁就是调用该实例方法（不包括静态方法）的对象。不多BB，上代码：  
【demo1】

public class SyncTest implements Runnable{

//共享资源变量

int count = 0;

@Override

public synchronized void run() {

for (int i = 0; i < 5; i++) {

increaseCount();

System.out.println(Thread.currentThread().getName()+":"+count++);

}

}

public static void main(String[] args) throws InterruptedException {

SyncTest syncTest1 = new SyncTest();

// SyncTest syncTest2 = new SyncTest();

Thread thread1 = new Thread(syncTest1,"thread1");

Thread thread2 = new Thread(syncTest1, "thread2");

thread1.start();

thread2.start();

}

}

/\*\*

\* 输出结果

thread1:0

thread1:1

thread1:2

thread1:3

thread1:4

thread2:5

thread2:6

thread2:7

thread2:8

thread2:9

\*/

  代码中开启了两个线程去操作一个变量（共享变量）,count++是先读取值，再写回一个新值。我们想一下，如果第一个线程执行这一过程中，第二个线程拿到写回之前的count值，做count++操作，那么这就造成了线程不安全。所以这里在run方法加上synchronized，获取一个对象锁，代码中的实例锁就是syncTest1了。  
  同时我们从输出结果看出：当一个线程正在访问一个对象synchronized实例方法时，别的线程是访问不了的。一个对象一把锁说的就是这个，当线程获取了该对象的锁后，其他线程无法获取该对象的锁，当然就访问不了该对象的synchronized方法，但是！但是！但是！可以访问该对象的其他未被synchronized修饰的方法。  
  如果是一个线程 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，因为两个实例对象锁并不同相同，此时如果两个线程操作数据并非共享的，线程安全是有保障的，遗憾的是如果两个线程操作的是共享数据，那么线程安全就有可能无法保证了。我们把上面代码中的main方法中的注释放开，表达这一线程不安全的现象  
【demo2】

public class SyncTest implements Runnable{

//共享资源变量

int count = 0;

@Override

public synchronized void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName()+":"+count++);

}

}

public static void main(String[] args) throws InterruptedException {

SyncTest syncTest1 = new SyncTest();

SyncTest syncTest2 = new SyncTest();

Thread thread1 = new Thread(syncTest1,"thread1");

Thread thread2 = new Thread(syncTest2, "thread2");

thread1.start();

thread2.start();

}

/\*\*

\* 输出结果

thread1:0

thread2:0

thread1:1

thread2:1

thread1:2

thread2:2

thread1:3

thread2:3

thread1:4

thread2:4

\*/

}

  我们从输出结果来看，两个线程可能同时拿到共享变量去做count++操作。上述操作中虽然我们的run方法还是使用synchronized修饰，但是我们new了两个实例。这就意味存在了两个不同的实例锁，thread1和thread2分别进入了syncTest1和syncTest2的实例锁，当然保证不了线程安全。但是我们也有解决方案啦：如果synchronized修饰的是静态方法呢？下面我们再介绍修饰静态方法。

synchronized修饰静态方法

  我们知道静态方法是不属于当前实例的，而是属性类的，那么这个锁就是类的class对象锁，上述问题引刃而解，请看代码：  
【demo3】

public class SyncTest implements Runnable {

//共享资源变量

static int count = 0;

@Override

public synchronized void run() {

increaseCount();

}

private synchronized static void increaseCount() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + ":" + count++);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void main(String[] args) throws InterruptedException {

SyncTest syncTest1 = new SyncTest();

SyncTest syncTest2 = new SyncTest();

Thread thread1 = new Thread(syncTest1, "thread1");

Thread thread2 = new Thread(syncTest2, "thread2");

thread1.start();

thread2.start();

}

/\*\*

\* 输出结果

thread1:0

thread1:1

thread1:2

thread1:3

thread1:4

thread2:5

thread2:6

thread2:7

thread2:8

thread2:9

\*/

}

  瞧瞧输出结果，问题解决了没？同样是new了两个不同实例，却保持了线程同步。那是我们synchronizd修饰的是静态方法，run方法中调用这个静态方法，再说一次 静态方法不属于当前实例，而是属于类。所以这个方案其实是用的一个把锁，而这个锁就是这个类的class对象锁。  
  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，是允许的，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(结合demo2，demo3)。

synchronized修饰代码块

  首先这个使用时的场景是：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。所以他的作用范围为synchronizd（obj）{}的这个大括号中  
【demo4】

public class SyncTest implements Runnable {

//共享资源变量

static int count = 0;

private byte[] mBytes = new byte[0];

@Override

public synchronized void run() {

increaseCount();

}

private void increaseCount() {

//假设省略了其他操作的代码。

//……………………

synchronized (this) {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + ":" + count++);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public static void main(String[] args) throws InterruptedException {

SyncTest syncTest1 = new SyncTest();

SyncTest syncTest2 = new SyncTest();

Thread thread1 = new Thread(syncTest1, "thread1");

Thread thread2 = new Thread(syncTest2, "thread2");

thread1.start();

thread2.start();

}

/\*\*

\* 输出结果

thread1:0

thread2:0

thread1:1

thread2:2

thread2:4

thread1:3

thread2:5

thread1:5

thread2:7

thread1:6

\*/

}

  从输出结果看出，这个demo并没有保证线程安全，因为我们指定锁为this，指的就是调用这个方法的实例对象。这里我们new了两个不同的实例对象syncTest1，syncTest2，所以有两个锁，thread1与thread2分别进入自己传入的对象锁的线程执行increaseCount方法，做成线程不安全。如果把demo4的成员变量注释放开，并将mBytes传入synchronized后面的括号中，也是线程不安全的结果。这里之所以加上mBytes这个对象是为了说明synchronized后面的括号中是可以指定任意对象充当锁的，而零长度的byte数组对象创建起来将比任何对象都经济。当然，如果要使用这个经济实惠的锁并保证线程安全，那就不能new出多个不同实例对象出来啦。如果你非要想new两个不同对象出来，又想保证线程同步的话，那么synchronized后面的括号中可以填入SyncTest.class，表示这个类对象作为锁，自然就能保证线程同步啦。使用方法为：

synchronized(xxxx.class){

//todo

}

总结

1. 修饰普通方法 一个对象中的加锁方法只允许一个线程访问。但要注意这种情况下锁的是访问该方法的实例对象， 如果多个线程不同对象访问该方法，则无法保证同步。
2. 修饰静态方法 由于静态方法是类方法， 所以这种情况下锁的是包含这个方法的类，也就是类对象；这样如果多个线程不同对象访问该静态方法，也是可以保证同步的。
3. 修饰代码块 其中普通代码块 如Synchronized（obj） 这里的obj 可以为类中的一个属性、也可以是当前的对象，它的同步效果和修饰普通方法一样；Synchronized方法 （obj.class）静态代码块它的同步效果和修饰静态方法类似。