Java并发编程：synchronized

2016/03/13 | 分类： [基础技术](http://www.importnew.com/cat/basic) | [0 条评论](http://www.importnew.com/18523.html#respond) | 标签： [SYNCHRONIZED](http://www.importnew.com/tag/synchronized), [并发编程](http://www.importnew.com/tag/%e5%b9%b6%e5%8f%91%e7%bc%96%e7%a8%8b)

分享到：**8**

原文出处： [海子](http://www.cnblogs.com/dolphin0520/p/3923737.html)

虽然多线程编程极大地提高了效率，但是也会带来一定的隐患。比如说两个线程同时往一个数据库表中插入不重复的数据，就可能会导致数据库中插入了相同的数据。今天我们就来一起讨论下线程安全问题，以及Java中提供了什么机制来解决线程安全问题。

以下是本文的目录大纲：

一.什么时候会出现线程安全问题？

二.如何解决线程安全问题？

三.synchronized同步方法或者同步块

若有不正之处，请多多谅解并欢迎批评指正。

一.什么时候会出现线程安全问题？

在单线程中不会出现线程安全问题，而在多线程编程中，有可能会出现同时访问同一个资源的情况，这种资源可以是各种类型的的资源：一个变量、一个对象、一个文件、一个数据库表等，而当多个线程同时访问同一个资源的时候，就会存在一个问题：

由于每个线程执行的过程是不可控的，所以很可能导致最终的结果与实际上的愿望相违背或者直接导致程序出错。

举个简单的例子：

现在有两个线程分别从网络上读取数据，然后插入一张数据库表中，要求不能插入重复的数据。

那么必然在插入数据的过程中存在两个操作：

1）检查数据库中是否存在该条数据；

2）如果存在，则不插入；如果不存在，则插入到数据库中。

假如两个线程分别用thread-1和thread-2表示，某一时刻，thread-1和thread-2都读取到了数据X，那么可能会发生这种情况：

thread-1去检查数据库中是否存在数据X，然后thread-2也接着去检查数据库中是否存在数据X。

结果两个线程检查的结果都是数据库中不存在数据X，那么两个线程都分别将数据X插入数据库表当中。

这个就是线程安全问题，即多个线程同时访问一个资源时，会导致程序运行结果并不是想看到的结果。

这里面，这个资源被称为：临界资源（也有称为共享资源）。

也就是说，当多个线程同时访问临界资源（一个对象，对象中的属性，一个文件，一个数据库等）时，就可能会产生线程安全问题。

不过，当多个线程执行一个方法，方法内部的局部变量并不是临界资源，因为方法是在栈上执行的，而Java栈是线程私有的，因此不会产生线程安全问题。

二.如何解决线程安全问题？

那么一般来说，是如何解决线程安全问题的呢？

基本上所有的并发模式在解决线程安全问题时，都采用“序列化访问临界资源”的方案，即在同一时刻，只能有一个线程访问临界资源，也称作同步互斥访问。

通常来说，是在访问临界资源的代码前面加上一个锁，当访问完临界资源后释放锁，让其他线程继续访问。

在Java中，提供了两种方式来实现同步互斥访问：synchronized和Lock。

本文主要讲述synchronized的使用方法，Lock的使用方法在下一篇博文中讲述。

三.synchronized同步方法或者同步块

在了解synchronized关键字的使用方法之前，我们先来看一个概念：互斥锁，顾名思义：能到达到互斥访问目的的锁。

举个简单的例子：如果对临界资源加上互斥锁，当一个线程在访问该临界资源时，其他线程便只能等待。

在Java中，每一个对象都拥有一个锁标记（monitor），也称为监视器，多线程同时访问某个对象时，线程只有获取了该对象的锁才能访问。

在Java中，可以使用synchronized关键字来标记一个方法或者代码块，当某个线程调用该对象的synchronized方法或者访问synchronized代码块时，这个线程便获得了该对象的锁，其他线程暂时无法访问这个方法，只有等待这个方法执行完毕或者代码块执行完毕，这个线程才会释放该对象的锁，其他线程才能执行这个方法或者代码块。

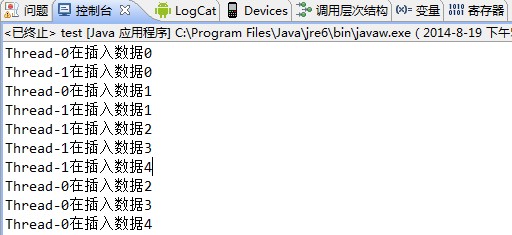
下面通过几个简单的例子来说明synchronized关键字的使用：

**1.synchronized方法**

下面这段代码中两个线程分别调用insertData对象插入数据：

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class Test {        public static void main(String[] args)  {          final InsertData insertData = new InsertData();            new Thread() {              public void run() {                  insertData.insert(Thread.currentThread());              };          }.start();            new Thread() {              public void run() {                  insertData.insert(Thread.currentThread());              };          }.start();      }  }    class InsertData {      private ArrayList<Integer> arrayList = new ArrayList<Integer>();        public void insert(Thread thread){          for(int i=0;i<5;i++){              System.out.println(thread.getName()+"在插入数据"+i);              arrayList.add(i);          }      }  } | |  |

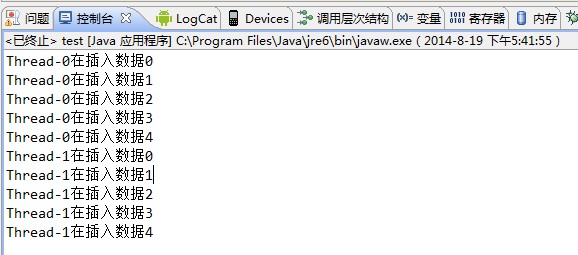
此时程序的输出结果为：



说明两个线程在同时执行insert方法。

而如果在insert方法前面加上关键字synchronized的话，运行结果为：

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10 | class InsertData {      private ArrayList<Integer> arrayList = new ArrayList<Integer>();        public synchronized void insert(Thread thread){          for(int i=0;i<5;i++){              System.out.println(thread.getName()+"在插入数据"+i);              arrayList.add(i);          }      }  } | |  |



从上输出结果说明，Thread-1插入数据是等Thread-0插入完数据之后才进行的。说明Thread-0和Thread-1是顺序执行insert方法的。

这就是synchronized方法。

不过有几点需要注意：

1）当一个线程正在访问一个对象的synchronized方法，那么其他线程不能访问该对象的其他synchronized方法。这个原因很简单，因为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方法。

2）当一个线程正在访问一个对象的synchronized方法，那么其他线程能访问该对象的非synchronized方法。这个原因很简单，访问非synchronized方法不需要获得该对象的锁，假如一个方法没用synchronized关键字修饰，说明它不会使用到临界资源，那么其他线程是可以访问这个方法的，

3）如果一个线程A需要访问对象object1的synchronized方法fun1，另外一个线程B需要访问对象object2的synchronized方法fun1，即使object1和object2是同一类型），也不会产生线程安全问题，因为他们访问的是不同的对象，所以不存在互斥问题。

**2.synchronized代码块**

synchronized代码块类似于以下这种形式：

|  |  |
| --- | --- |
| 1  2  3 | synchronized(synObject) {        } |

当在某个线程中执行这段代码块，该线程会获取对象synObject的锁，从而使得其他线程无法同时访问该代码块。

synObject可以是this，代表获取当前对象的锁，也可以是类中的一个属性，代表获取该属性的锁。

比如上面的insert方法可以改成以下两种形式：

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12 | class InsertData {      private ArrayList<Integer> arrayList = new ArrayList<Integer>();        public void insert(Thread thread){          synchronized (this) {              for(int i=0;i<100;i++){                  System.out.println(thread.getName()+"在插入数据"+i);                  arrayList.add(i);              }          }      }  } | |  |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class InsertData {      private ArrayList<Integer> arrayList = new ArrayList<Integer>();      private Object object = new Object();        public void insert(Thread thread){          synchronized (object) {              for(int i=0;i<100;i++){                  System.out.println(thread.getName()+"在插入数据"+i);                  arrayList.add(i);              }          }      }  } | |

从上面可以看出，synchronized代码块使用起来比synchronized方法要灵活得多。因为也许一个方法中只有一部分代码只需要同步，如果此时对整个方法用synchronized进行同步，会影响程序执行效率。而使用synchronized代码块就可以避免这个问题，synchronized代码块可以实现只对需要同步的地方进行同步。

另外，每个类也会有一个锁，它可以用来控制对static数据成员的并发访问。

并且如果一个线程执行一个对象的非static synchronized方法，另外一个线程需要执行这个对象所属类的static synchronized方法，此时不会发生互斥现象，因为访问static synchronized方法占用的是类锁，而访问非static synchronized方法占用的是对象锁，所以不存在互斥现象。

看下面这段代码就明白了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class Test {        public static void main(String[] args)  {          final InsertData insertData = new InsertData();          new Thread(){              @Override              public void run() {                  insertData.insert();              }          }.start();          new Thread(){              @Override              public void run() {                  insertData.insert1();              }          }.start();      }  }    class InsertData {      public synchronized void insert(){          System.out.println("执行insert");          try {              Thread.sleep(5000);          } catch (InterruptedException e) {              e.printStackTrace();          }          System.out.println("执行insert完毕");      }        public synchronized static void insert1() {          System.out.println("执行insert1");          System.out.println("执行insert1完毕");      }  } |

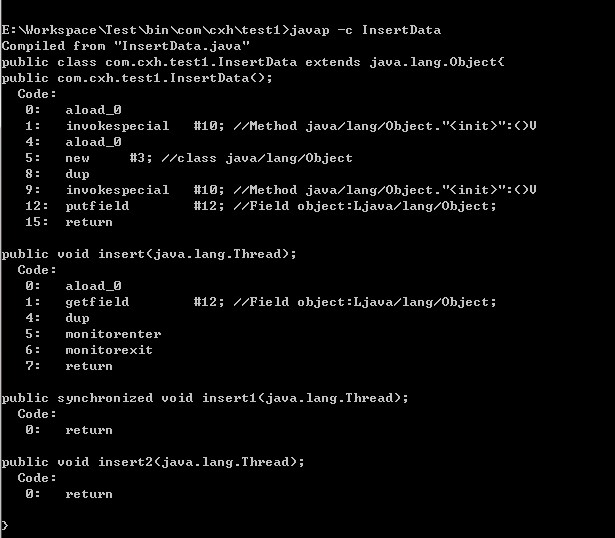
执行结果;



第一个线程里面执行的是insert方法，不会导致第二个线程执行insert1方法发生阻塞现象。

下面我们看一下synchronized关键字到底做了什么事情，我们来反编译它的字节码看一下，下面这段代码反编译后的字节码为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class InsertData {      private Object object = new Object();        public void insert(Thread thread){          synchronized (object) {            }      }        public synchronized void insert1(Thread thread){        }        public void insert2(Thread thread){        }  } |



从反编译获得的字节码可以看出，synchronized代码块实际上多了monitorenter和monitorexit两条指令。monitorenter指令执行时会让对象的锁计数加1，而monitorexit指令执行时会让对象的锁计数减1，其实这个与操作系统里面的PV操作很像，操作系统里面的PV操作就是用来控制多个线程对临界资源的访问。对于synchronized方法，执行中的线程识别该方法的 method\_info 结构是否有 ACC\_SYNCHRONIZED 标记设置，然后它自动获取对象的锁，调用方法，最后释放锁。如果有异常发生，线程自动释放锁。

有一点要注意：对于synchronized方法或者synchronized代码块，当出现异常时，JVM会自动释放当前线程占用的锁，因此不会由于异常导致出现死锁现象。