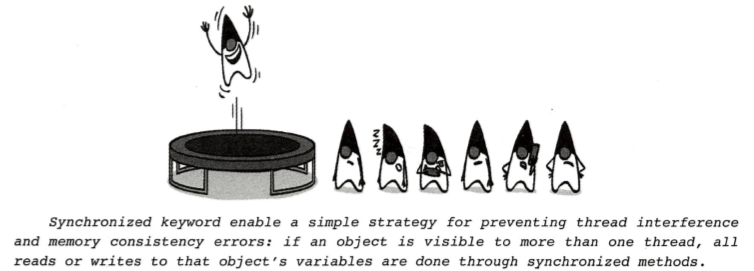
# Synchronized关键字的初识和深入

什么是synchronized？

下面是jdk官网堆synchronized关键字比较权威的解释



上述的意思为：synchronized关键字可以实现一个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是可见的，那么对该对象的所有读或者写都将通过同步的方式来进行，具体表现如下：

* Synchronized关键字提供了一种锁的机制，能够确保共享变量的互斥访问，从而防止数据不一致问题的出现
* Synchronized关键字包括monitor enter和monitor exit两个JVM指令，它能够保证在任何时候任何线程执行到monitor enter成功之前都必须从主内存中获取数据，而不是从缓存中，在monitor exit运行成功后，共享变量被更新后的值必须刷入主内存
* Synchronized的指令严格遵守java happens-before规则，一个monito exit指令之前必定要有一个monitor enter

Synchronized关键字的用法

Synchronized关键字可以用于对代码块或方法进行修饰，而不能用于对class以及变量进行修饰

1. **同步方法**

示例代码如下

|  |
| --- |
| public synchronized void sync()  {  …  …  }  public synchronized static void staticsync()  {  …  …  } |

**2. 同步代码块的语法示例**

|  |
| --- |
| private final Object MUTEX=new Object();  public void sync()  {  synchronied(MUTEX)  {  …  …  }  } |

用synchronized关键字改写叫号程序

package com.lagoon.test;  
  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/7 9:59  
 \** ***@Description*** *synchronized关键字改写的叫号程序  
 \*/*public class TicketWindowRunnable implements Runnable{  
  
 private int index=1;  
 private final static int *MAX*=500;  
  
 private final static Object *MUTEX*=new Object();  
  
 @Override  
 public void run() {  
 synchronized (*MUTEX*){  
 while (index<*MAX*){  
  
 System.*out*.println(Thread.*currentThread*()+"的号码是"+(index++));  
 try {  
 TimeUnit.*SECONDS*.sleep(1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 }  
  
 public static void main(String[] args) {  
 final TicketWindowRunnable task=new TicketWindowRunnable();  
 Thread windowThread1=new Thread(task,"一号窗口");  
 Thread windowThread2=new Thread(task,"二号窗口");  
 Thread windowThread3=new Thread(task,"三号窗口");  
 Thread windowThread4=new Thread(task,"四号窗口");  
  
 windowThread1.start();  
 windowThread2.start();  
 windowThread3.start();  
 windowThread4.start();  
 }  
}

深入synchronized关键字

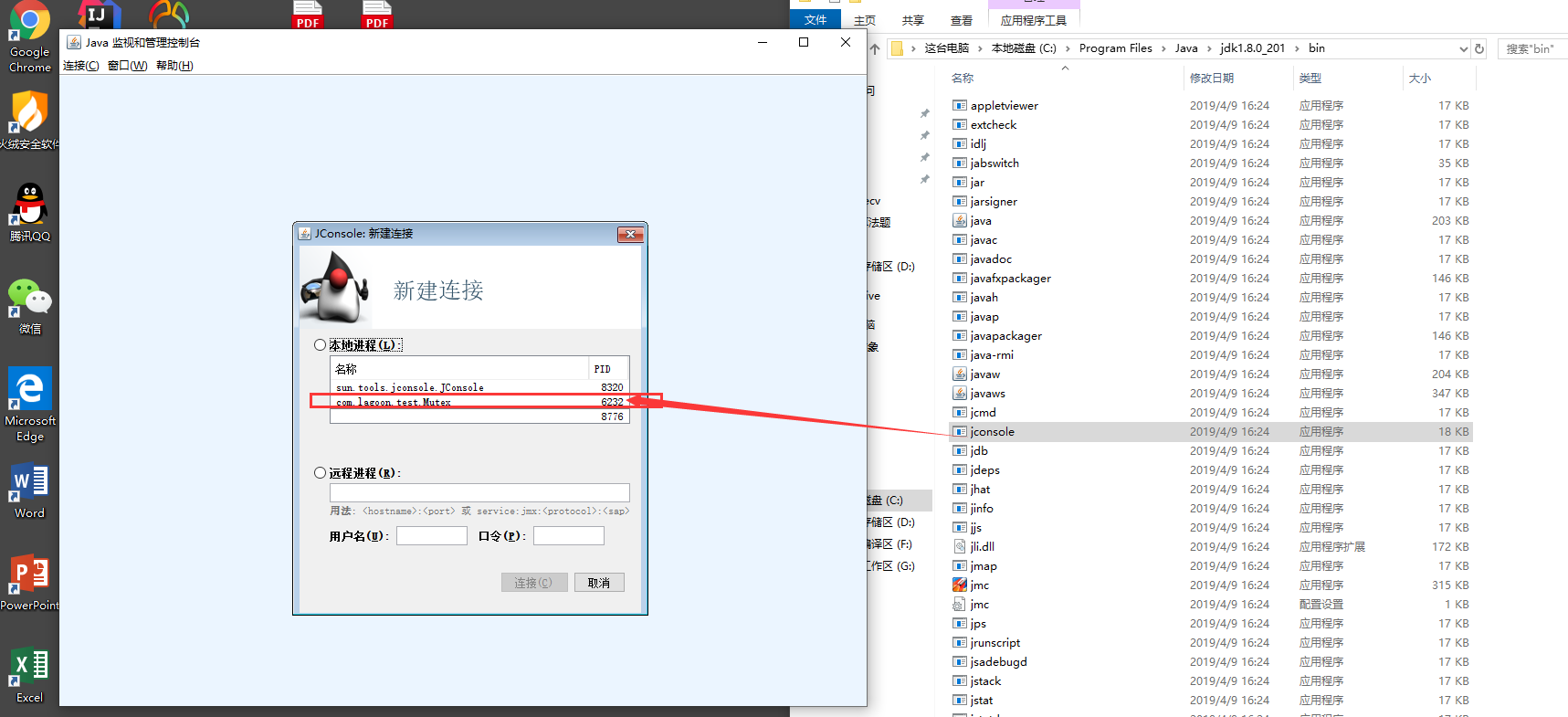
线程堆栈分析

Synchronized关键字提供了一种互斥机制，也就是说在同一时刻，只能有一个线程访问同步资源，很多资料，书籍将synchronized（mutex）称为锁，其实这种说法是不严谨的，准确的说应该是某线程获取了与mutex关联的monitor锁

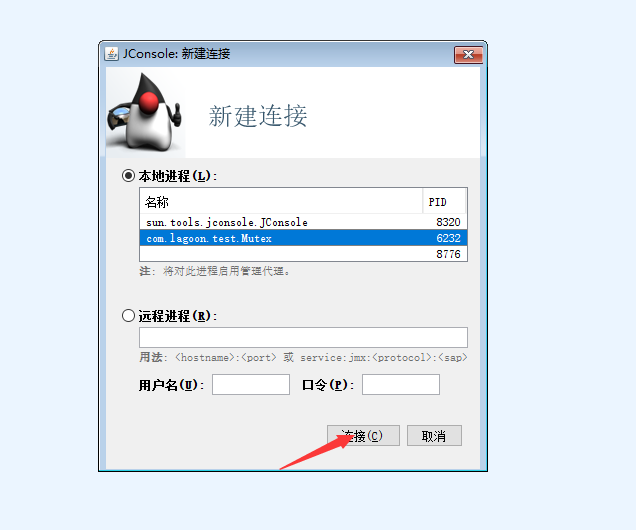
Demo：

package com.lagoon.test;  
  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/7 10:25  
 \** ***@Description*** *synchronized关键字线程堆栈分析  
 \*/*public class Mutex {  
   
 private final static Object *MUTEX*=new Object();  
   
 public void accessResource(){  
 synchronized (*MUTEX*){  
 try {  
 TimeUnit.*MINUTES*.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 final Mutex mutex=new Mutex();  
 for (int i=0;i<5;i++){  
 new Thread(mutex::accessResource).start();  
 }  
 }  
}

上面的代码中定义了一个方法accessResource，并且使用同步代码块的方式堆accessResource进行了同步，同时定义5个线程调用accessResource方法，由于同步代码块的互斥性，只能有一个线程获取了mutex monitor的锁，其他线程只能进入阻塞状态，等待获取mutex monitor锁的线程对其进行释放，运行上面的程序，然后打开JConsole工具监控

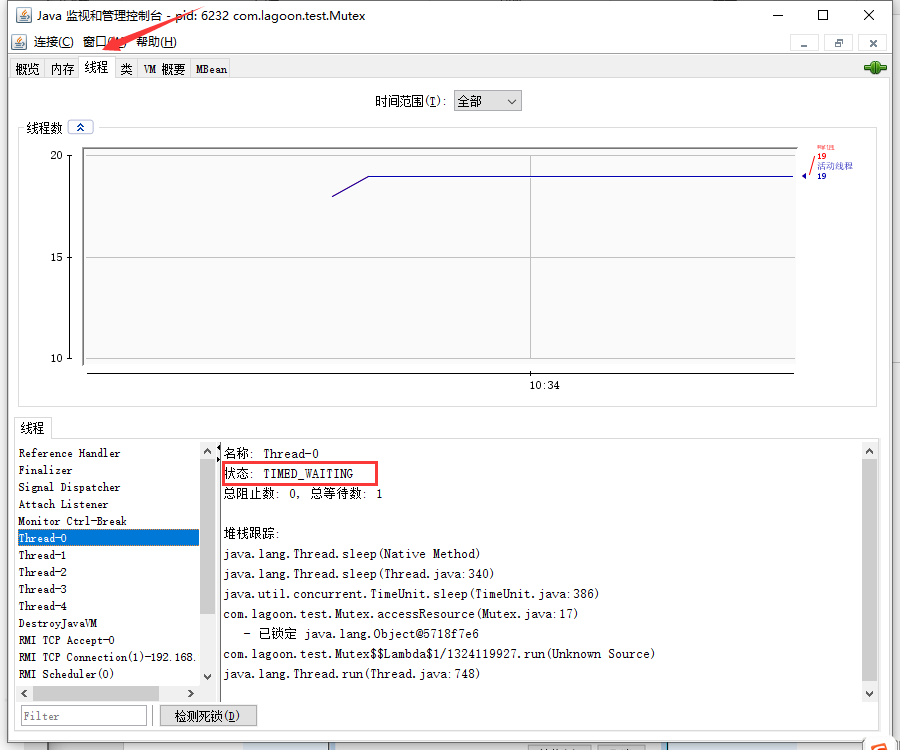


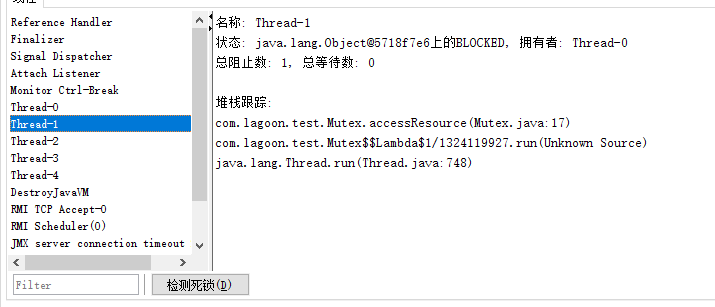
链接本地进程



切换至线程tab

会发现创建的五个线程只有线程0号处于等待执行状态，其他都处于阻塞状态





…

JVM指令分析

使用JDK命令对程序进行反编译

如何在idea中快捷查看反汇编？

参考

<https://blog.csdn.net/weixin_40739833/article/details/81301577>

反汇编结果如下：

|  |
| --- |
| "C:\Program Files\Java\jdk1.8.0\_201\bin\javap.exe" -c Mutex.class  Compiled from "Mutex.java"  public class com.lagoon.test.Mutex {  public com.lagoon.test.Mutex();  Code:  0: aload\_0  1: invokespecial #1 // Method java/lang/Object."<init>":()V  4: return  public void accessResource();  Code:  0: getstatic #2 // Field MUTEX:Ljava/lang/Object;  3: dup  4: astore\_1  5: monitorenter  6: getstatic #3 // Field java/util/concurrent/TimeUnit.MINUTES:Ljava/util/concurrent/TimeUnit;  9: ldc2\_w #4 // long 10l  12: invokevirtual #6 // Method java/util/concurrent/TimeUnit.sleep:(J)V  15: goto 23  18: astore\_2  19: aload\_2  20: invokevirtual #8 // Method java/lang/InterruptedException.printStackTrace:()V  23: aload\_1  24: monitorexit  25: goto 33  28: astore\_3  29: aload\_1  30: monitorexit  31: aload\_3  32: athrow  33: return  Exception table:  from to target type  6 15 18 Class java/lang/InterruptedException  6 25 28 any  28 31 28 any  public static void main(java.lang.String[]);  Code:  0: new #9 // class com/lagoon/test/Mutex  3: dup  4: invokespecial #10 // Method "<init>":()V  7: astore\_1  8: iconst\_0  9: istore\_2  10: iload\_2  11: iconst\_5  12: if\_icmpge 42  15: new #11 // class java/lang/Thread  18: dup  19: aload\_1  20: dup  21: invokevirtual #12 // Method java/lang/Object.getClass:()Ljava/lang/Class;  24: pop  25: invokedynamic #13, 0 // InvokeDynamic #0:run:(Lcom/lagoon/test/Mutex;)Ljava/lang/Runnable;  30: invokespecial #14 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V  33: invokevirtual #15 // Method java/lang/Thread.start:()V  36: iinc 2, 1  39: goto 10  42: return  static {};  Code:  0: new #16 // class java/lang/Object  3: dup  4: invokespecial #1 // Method java/lang/Object."<init>":()V  7: putstatic #2 // Field MUTEX:Ljava/lang/Object;  10: return  }  Process finished with exit code 0 |

发现monitor enter和monitor exit都是成对出现的（有些时候会出现一个monitor enter多个monitor exit但是每一个monitor exit之前必定有对应的monitor enter）

选取其中的片段进行分析：

|  |
| --- |
| public void accessResource();  Code:  0: getstatic #2 // Field MUTEX:Ljava/lang/Object;  3: dup  4: astore\_1  5: monitorenter  6: getstatic #3 // Field java/util/concurrent/TimeUnit.MINUTES:Ljava/util/concurrent/TimeUnit;  9: ldc2\_w #4 // long 10l  12: invokevirtual #6 // Method java/util/concurrent/TimeUnit.sleep:(J)V  15: goto 23  18: astore\_2  19: aload\_2  20: invokevirtual #8 // Method java/lang/InterruptedException.printStackTrace:()V  23: aload\_1  24: monitorexit  25: goto 33  28: astore\_3  29: aload\_1  30: monitorexit  31: aload\_3  32: athrow  33: return |

从上至下高亮分析：

1. 获取到MUTEX引用，然后执行
2. Monitorenter JVM指令，休眠结束后goto至23行的位置
3. 23行 astore\_<n>存储引用本地变量表；aload\_<n>从本地变量表加载引用；getstatic从class中获得静态属性
4. 最后一行，执行monitor exit JVM指令

Monitorenter与Monitorexit

Monitorenter

每个对象都与一个monitor相关联，一个monitor的lock锁只能被一个线程在同一时间获得，在一个线程尝试获得与对象monitor的所有权时会发生如下几件事情：

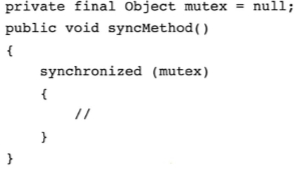
* 如果monitor的计数器为0，则意味着该monitor的lock锁还没有被获得，某个线程获得之后将立即对该计数器加一，从此该线程就是这个monitor的所有者了
* 如果一个已经拥有该monitor所有权的线程重入，则会导致monitor计数器再次累加。
* 如果monitor已经被其他线程所拥有，则其他线程尝试获取该monitor的所有权时，会被陷入阻塞状态直到monitor计数器变为0，才能再次尝试获取堆monitor的所有权

Monitorexit

释放队monitor的所有权想要释放对某个对象关联的monitor的所有权的前提是，曾经获得了所有权。释放monitor所有权的过程比较简单，就是将monitor的计数器减一，如果计数器为0，那就意味着该线程不再拥有对该monitor的所有权，通俗的将就是解锁。与此同时被该monitor block的线程将再次尝试获得对该monitor的所有权

使用synchronized关键字需要注意的问题

* 与monitor关联的对象不能为空



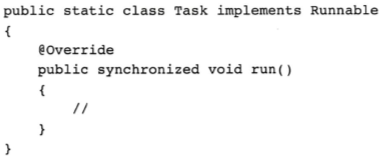
Mutex为null

每一个对象和一个monitor关联，对象为null，monitor无从谈起

* Synchronized作用域太大

由于synchronized关键字存在排他性，也就是说所有的线程必须串行地经过synchronized保护的共享区域，如果synchronized作用域越大，则代表着其效率越低，甚至还会丧失并发的优势

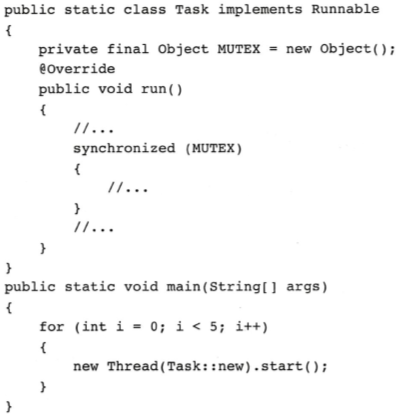
例如



上面的代码对整个线程而定执行逻辑单元都进行了synchronized同步，从而丧失了并发的能力，synchronized关键字应该尽可能地只作用于共享资源（数据）的读写作用域

* 不同的monitor企图锁相同的方法

例如



上面的代码构造了五个线程，同时也构造了五个Runnable实例，Runnable作为线程逻辑执行单元传递给Thread，但是上面的例子，synchronized根本互斥不了与之对应的作用域，线程之间进行monitor lock的争抢只能发生在与monitor关联的同一个引用上，上面的代码每一个线程争抢的monitor关联引用都是彼此独立的，因此不可能起到互斥的作用

* 多个锁的交叉导致死锁

多个锁的交叉很容易引起线程出现死锁的情况，程序并没有任何错误输出，但就是不工作，例如：

