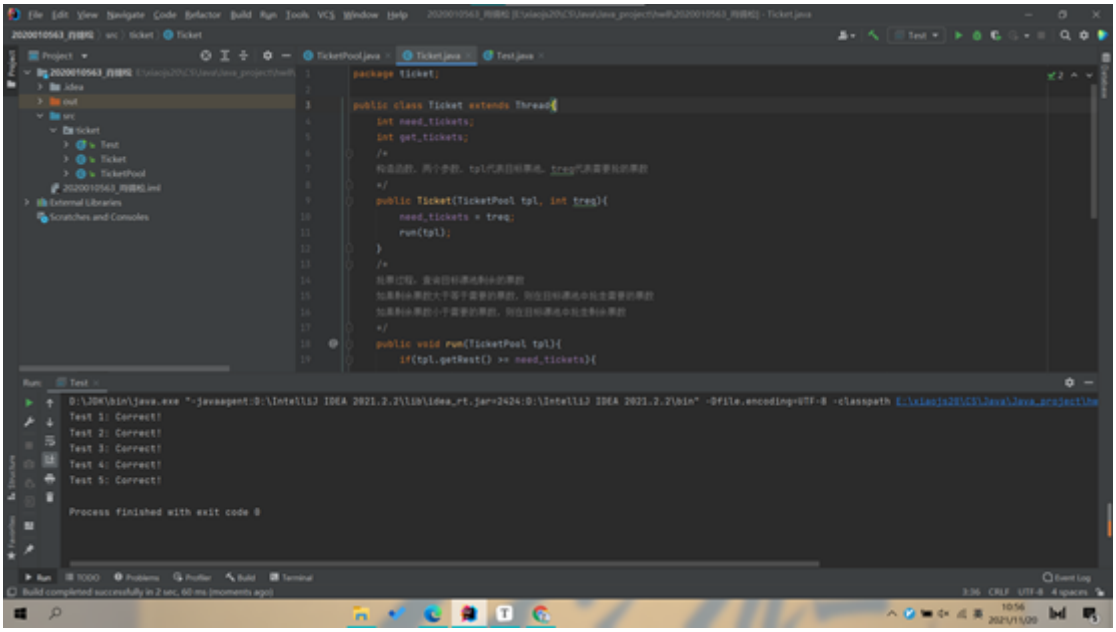
HW8

运行结果



附加选做题：Lock

synchronized 是java中一种常见的解决资源访问冲突的方法，但其使用方式比较单一，功能上具有局限性。java中提供了一些其他机制解决资源访问冲突和线程同步问题，调研java中 java.util.concurrent.locks 包下的Lock接口，与 synchronized 关键字对比，在文档中简述其用法和作用。

区别

1. lock是一个接口，而synchronized是java的一个关键字。
2. synchronized在发生异常时会自动释放占有的锁，因此不会出现死锁；
   而lock发生异常时，不会主动释放占有的锁，必须手动来释放锁，可能引起死锁的发生。

synchronized实现原理

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的class对象
- 同步方法块，锁是括号里面的对象
  当一个线程访问同步代码块时，它首先是需要得到锁，当退出或者抛出异常时必须要释放锁

synchronized使用方法

一 修饰方法

Synchronized修饰一个方法很简单，就是在方法的前面加synchronized，synchronized修饰方法和修饰一个代码块类似，只是作用范围不一样，修饰代码块是大括号括起来的范围，而修饰方法范围是整个函数。

方法一

```
public synchronized void method()
{
   // todo
}
```

方法二

```
public void method()
{
   synchronized(this) {
      // todo
   }
}
```

写法一修饰的是一个**方法**，写法二修饰的是一个**代码块**，但写法一与写法二是**等价**的，都是锁定了整个方法时的内容。

synchronized关键字**不能继承。**
虽然可以使用synchronized来定义方法，但**synchronized并不属于方法定义的一部分**，因此，synchronized关键字不能被继承。如果在父类中的某个方法使用了synchronized关键字，而在子类中覆盖了这个方法，在子类中的这个方法默认情况下并不是同步的，而必须**显式**地在子类的这个方法中加上synchronized关键字才可以。当然，还可以在子类方法中调用父类中相应的方法，这样虽然子类中的方法不是同步的，但子类调用了父类的同步方法，因此，子类的方法也就相当于同步了。这两种方式的例子代码如下：
在子类方法中加上synchronized关键字

```
class Parent {
   public synchronized void method() { }
}
class Child extends Parent {
   public synchronized void method() { }
}
```

在子类方法中调用父类的同步方法

```
class Parent {
   public synchronized void method() {   }
}
class Child extends Parent {
   public void method() { super.method();   }
}
```

1. 在定义接口方法时不能使用synchronized关键字。
2. 构造方法不能使用synchronized关键字，但可以使用synchronized代码块来进行同步。

二 修饰一个代码块

1）一个线程访问一个对象中的synchronized(this)同步代码块时，其他试图访问该对象的线程将被阻塞

注意下面两个程序的区别

```
class SyncThread implements Runnable {
       private static int count;

       public SyncThread() {
          count = 0;
       }

       public void run() {
          synchronized(this) {
             for (int i = 0; i < 5; i++) {
                try {
                   System.out.println(Thread.currentThread().getName() + ":"
+ (count++));
                   Thread.sleep(100);
                } catch (InterruptedException e) {
                   e.printStackTrace();
                }
             }
          }
       }

       public int getCount() {
          return count;
       }
}

public class Demo00 {
    public static void main(String args[]){
       //test01
       //SyncThread s1 = new SyncThread();
       //SyncThread s2 = new SyncThread();
       //Thread t1 = new Thread(s1);
       //Thread t2 = new Thread(s2);
       //test02
        SyncThread s = new SyncThread();
        Thread t1 = new Thread(s);
        Thread t2 = new Thread(s);

        t1.start();
        t2.start();
    }
}
```

test01的运行结果

```
Thread-0:1
Thread-1:0
Thread-1:2
Thread-0:3
Thread-1:4
Thread-0:5
Thread-1:6
Thread-0:7
Thread-1:8
Thread-0:9
```

test02的运行结果

```
Thread-0:0
Thread-0:1
Thread-0:2
Thread-0:3
Thread-0:4
Thread-1:5
Thread-1:6
Thread-1:7
Thread-1:8
Thread-1:9
```

当两个并发线程(thread1和thread2)访问同一个对象(syncThread)中的synchronized代码块时，在同一时刻只能有一个线程得到执行，另一个线程受阻塞，必须等待当前线程执行完这个代码块以后才能执行该代码块。Thread1和thread2是**互斥**的，因为在执行synchronized代码块时会锁定当前的对象，只有执行完该代码块才能释放该对象锁，下一个线程才能执行并锁定该对象。

为什么上面的例子中thread1和thread2同时在执行。这是因为**synchronized只锁定对象**，每个对象只有一个锁（lock）与之相关联。

```
1  class Counter implements Runnable{
2     private int count;
3
4     public Counter() {
5        count = 0;
6     }
7
8     public void countAdd() {
9        synchronized(this) {
10          for (int i = 0; i < 5; i ++) {
11             try {
12                System.out.println(Thread.currentThread().getName() + ":" +
   (count++));
13                Thread.sleep(100);
14             } catch (InterruptedException e) {
15                e.printStackTrace();
16             }
17          }
18       }
19    }
20
```

```
    //非synchronized代码块，未对count进行读写操作，所以可以不用synchronized
    public void printCount() {
        for (int i = 0; i < 5; i ++) {
            try {
                System.out.println(Thread.currentThread().getName() + " count:"
+ count);
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void run() {
        String threadName = Thread.currentThread().getName();
        if (threadName.equals("A")) {
            countAdd();
        } else if (threadName.equals("B")) {
            printCount();
        }
    }
}

public class Demo00{
    public static void main(String args[]){
        Counter counter = new Counter();
        Thread thread1 = new Thread(counter, "A");
        Thread thread2 = new Thread(counter, "B");
        thread1.start();
        thread2.start();
    }
}
```

```
A:0
B count:1
B count:1
A:1
B count:2
A:2
B count:3
A:3
B count:4
A:4
```

可以看见B线程的调用是非synchronized,并不影响A线程对synchronized部分的调用。
从上面的结果中可以看出**一个线程访问一个对象的synchronized代码块时，别的线程可以访问该对象的非synchronized代码块而不受阻塞。**

3）指定要给某个对象加锁

```
/**
 * 银行账户类
 */
```

```
class Account {
   String name;
   float amount;

   public Account(String name, float amount) {
      this.name = name;
      this.amount = amount;
   }
   //存钱
   public  void deposit(float amt) {
      amount += amt;
      try {
         Thread.sleep(100);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
   //取钱
   public  void withdraw(float amt) {
      amount -= amt;
      try {
         Thread.sleep(100);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }

   public float getBalance() {
      return amount;
   }
}

/**
 * 账户操作类
 */
class AccountOperator implements Runnable{
   private Account account;
   public AccountOperator(Account account) {
      this.account = account;
   }

   public void run() {
      synchronized (account) {
         account.deposit(500);
         account.withdraw(500);
         System.out.println(Thread.currentThread().getName() + ":" +
account.getBalance());
      }
   }
}

public class Demo00{

    //public static final Object signal = new Object(); // 线程间通信变量
    //将account改为Demo00.signal也能实现线程同步
    public static void main(String args[]){
        Account account = new Account("zhang san", 10000.0f);
        AccountOperator accountOperator = new AccountOperator(account);
```

```
61
62          final int THREAD_NUM = 5;
63          Thread threads[] = new Thread[THREAD_NUM];
64          for (int i = 0; i < THREAD_NUM; i ++) {
65             threads[i] = new Thread(accountOperator, "Thread" + i);
66             threads[i].start();
67          }
68      }
69  }
```

运行结果

```
Thread0:10000.0
Thread4:10000.0
Thread3:10000.0
Thread2:10000.0
Thread1:10000.0
```

在AccountOperator 类中的run方法里，我们用synchronized 给account对象加了锁。这时，当一个线程访问account对象时，其他试图访问account对象的线程将会阻塞，直到该线程访问account对象结束。也就是说谁拿到那个锁谁就可以运行它所控制的那段代码。
当有一个明确的对象作为锁时，就可以用类似下面这样的方式写程序。

```
1  public void method3(SomeObject obj)
2  {
3     //obj 锁定的对象
4     synchronized(obj)
5     {
6        // todo
7     }
8  }
```

当没有明确的对象作为锁，只是想让一段代码同步时，可以创建一个特殊的对象来充当锁：

```
1  class Test implements Runnable
2  {
3     private byte[] lock = new byte[0];  // 特殊的instance变量
4     public void method()
5     {
6        synchronized(lock) {
7           // todo 同步代码块
8        }
9     }
10
11    public void run(){
12
13    }
14 }
```

本例中去掉注释中的signal可以看到同样的运行结果

三 修饰一个静态的方法

Synchronized也可修饰一个静态方法，用法如下：

```
public synchronized static void method() {
   // todo
}
```

静态方法是属于类的而不属于对象的。同样的，**synchronized修饰的静态方法锁定的是这个类的所有对象。**

```
/**
 * 同步线程
 */
class SyncThread implements Runnable {
   private static int count;

   public SyncThread() {
      count = 0;
   }

   public synchronized static void method() {
      for (int i = 0; i < 5; i ++) {
         try {
            System.out.println(Thread.currentThread().getName() + ":" +
(count++));
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

   public synchronized void run() {
      method();
   }
}

public class Demo00{

    public static void main(String args[]){
        SyncThread syncThread1 = new SyncThread();
        SyncThread syncThread2 = new SyncThread();
        Thread thread1 = new Thread(syncThread1, "SyncThread1");
        Thread thread2 = new Thread(syncThread2, "SyncThread2");
        thread1.start();
        thread2.start();
    }
}
```

syncThread1和syncThread2是SyncThread的两个对象，但在thread1和thread2并发执行时却保持了线程同步。这是因为run中调用了静态方法method，而静态方法是属于类的，所以syncThread1和syncThread2相当于用了同一把锁。

四 修饰一个类

Synchronized还可作用于一个类，用法如下：

```
class ClassName {
   public void method() {
      synchronized(ClassName.class) {
         // todo
      }
   }
}
```

```
/**
 * 同步线程
 */
class SyncThread implements Runnable {
   private static int count;

   public SyncThread() {
      count = 0;
   }

   public static void method() {
      synchronized(SyncThread.class) {
         for (int i = 0; i < 5; i ++) {
            try {
               System.out.println(Thread.currentThread().getName() + ":" +
(count++));
               Thread.sleep(100);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
         }
      }
   }

   public synchronized void run() {
      method();
   }
}
```

本例的的给class加锁和上例的给静态方法加锁是一样的，所有对象公用一把锁。

总结

A. 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。
B. 每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。
C. 实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

但是采用synchronized关键字来实现同步的话，就会导致一个问题：

如果多个线程都只是进行读操作，所以当一个线程在进行读操作时，其他线程只能等待无法进行读操作。因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过Lock就可以办到。另外，通过Lock可以知道线程有没有成功获取到锁。这个是synchronized无法办到的。

总结一下，也就是说Lock提供了比synchronized更多的功能。但是要注意以下几点：

1. Lock不是Java语言内置的，synchronized是Java语言的关键字，因此是内置特性。Lock是一个类，通过这个类可以实现同步访问；
2. Lock和synchronized有一点非常大的不同，采用synchronized不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而Lock则必须要用户去**手动释放锁**，如果没有主动释放锁，就有可能导致出现**死锁现象**。

lock用法

方法介绍

```
// 获取锁
void lock()

// 如果当前线程未被中断，则获取锁，可以响应中断
void lockInterruptibly()

// 返回绑定到此 Lock 实例的新 Condition 实例
Condition newCondition()

// 仅在调用时锁为空闲状态才获取该锁，可以响应中断
boolean tryLock()

// 如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁
boolean tryLock(long time, TimeUnit unit)

// 释放锁
void unlock()
```

1.普通用法

```
public static void testlock() {
        Lock lock = new ReentrantLock();
        Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            // TODO Auto-generated method stub
            lock.lock();
            try {
                Thread.sleep(1000);
                System.out.println("goon");
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }

    });

```

```
    t.start();
    System.out.println("start");
    lock.lock();
    System.out.println("over");
    lock.unlock();
}
```

```
start
goon
over
```

2.trylock

```
public static void testtry() {
        Lock lock = new ReentrantLock();
        Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            // TODO Auto-generated method stub
            lock.lock();
            System.out.println("get");
            try {
                Thread.sleep(1000);

            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } finally {
                lock.unlock();
                System.out.println("release");
            }
        }
    });

    Thread t1 = new Thread(new Runnable() {

        @Override
        public void run() {
            // TODO Auto-generated method stub

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            while (true) {
                if (lock.tryLock()) {
                        System.out.println("get success");
                        lock.unlock();
                        break;
                }else {
                    System.out.println("get fail ... ");
                    try {
                        Thread.sleep(100);
```

```
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            }
        }

    });

    t.start();
    t1.start();

}
```

```
get
get fail ...
get fail ...
get fail ...
get fail ...
get fail ...
get fail ...
get fail ...
get fail ...
get fail ...
release
get success
```

3.interruptlock

```
public static void testinterrupt() {
        Lock lock = new ReentrantLock();
        Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            // TODO Auto-generated method stub
            lock.lock();
            try {
                Thread.sleep(10000);
                System.out.println("goon ...");
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }

    });

    Thread t1 = new Thread(new Runnable() {

        @Override
        public void run() {
            // TODO Auto-generated method stub
            try {
```

```
                lock.lockInterruptibly();
                System.out.println("get ...");
                lock.unlock();
            } catch (InterruptedException e1) {
                // TODO Auto-generated catch block
                //e1.printStackTrace();

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

        }

    });

    t.start();
    t1.start();

    try {
        Thread.sleep(5000);
    }catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    System.out.println("to interrupt ");
    t1.interrupt();
}
```

```
to interrupt
interrupt ...
goon ...
```

4.delay

```
public static void testdelay() {
        Lock lock = new ReentrantLock();
        Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            // TODO Auto-generated method stub
            lock.lock();
            try {
                Thread.sleep(10000);
                System.out.println("goon ...");
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }

    });

    Thread t1 = new Thread(new Runnable() {

```

```
        @Override
        public void run() {
            // TODO Auto-generated method stub
            try {
                if(lock.tryLock(5, TimeUnit.SECONDS)) {
                    System.out.println("get ...");
                    lock.unlock();
                }else {
                    System.out.println("have not get  ...");
                }
            } catch (InterruptedException e1) {
                // TODO Auto-generated catch block
                //e1.printStackTrace();

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

        }

    });

    t.start();
    t1.start();
}
```

```
have not get  ...
goon ...
```