HW8

运行结果

```
Discrete Berlane Code Belace Data Nam (John VC) Mindow (Bell) 2000000001, NBMC (Consequence Code Data Name (Code Data Name (Co
```

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
// todo
// 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() {
  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 {
 2
           private static int count;
3
4
           public SyncThread() {
 5
              count = 0;
 6
           }
 7
           public void run() {
8
9
              synchronized(this) {
10
                  for (int i = 0; i < 5; i++) {
11
                     try {
                        System.out.println(Thread.currentThread().getName() + ":"
12
    + (count++));
13
                        Thread.sleep(100);
14
                     } catch (InterruptedException e) {
                        e.printStackTrace();
15
16
                     }
17
                  }
18
              }
19
           }
20
21
           public int getCount() {
22
              return count;
23
           }
24
    }
25
26
    public class Demo00 {
27
        public static void main(String args[]){
28
           //test01
29
           //SyncThread s1 = new SyncThread();
30
           //SyncThread s2 = new SyncThread();
31
           //Thread t1 = new Thread(s1);
32
           //Thread t2 = new Thread(s2);
33
           //test02
34
            SyncThread s = new SyncThread();
35
            Thread t1 = new Thread(s);
36
            Thread t2 = new Thread(s);
37
38
            t1.start();
39
            t2.start();
40
        }
41 }
```

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。

```
class Counter implements Runnable{
 1
 2
       private int count;
 4
       public Counter() {
 5
          count = 0;
 6
       }
 7
       public void countAdd() {
 8
9
          synchronized(this) {
              for (int i = 0; i < 5; i ++) {
10
11
                    System.out.println(Thread.currentThread().getName() + ":" +
12
    (count++));
13
                    Thread.sleep(100);
                 } catch (InterruptedException e) {
14
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
21
22
       public void printCount() {
23
          for (int i = 0; i < 5; i ++) {
24
             try {
25
                System.out.println(Thread.currentThread().getName() + " count:"
    + count);
26
                Thread.sleep(100);
27
             } catch (InterruptedException e) {
28
                e.printStackTrace();
29
30
          }
31
       }
32
33
       public void run() {
34
          String threadName = Thread.currentThread().getName();
35
          if (threadName.equals("A")) {
36
             countAdd();
37
          } else if (threadName.equals("B")) {
38
             printCount();
39
40
       }
41
    }
42
    public class Demo00{
43
44
        public static void main(String args[]){
45
            Counter counter = new Counter();
            Thread thread1 = new Thread(counter, "A");
46
            Thread thread2 = new Thread(counter, "B");
47
            thread1.start();
48
49
            thread2.start();
50
        }
51
   }
```

```
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

```
1 /**
2 * 银行账户类
3 */
```

```
4
    class Account {
 5
       String name;
 6
       float amount;
 7
 8
       public Account(String name, float amount) {
 9
          this.name = name;
10
          this.amount = amount;
11
       }
12
       //存钱
13
       public void deposit(float amt) {
14
          amount += amt;
15
          try {
16
             Thread.sleep(100);
          } catch (InterruptedException e) {
17
18
             e.printStackTrace();
19
          }
20
       }
21
       //取钱
       public void withdraw(float amt) {
22
23
          amount -= amt;
24
          try {
25
             Thread.sleep(100);
26
          } catch (InterruptedException e) {
27
             e.printStackTrace();
28
          }
29
       }
30
       public float getBalance() {
31
32
          return amount;
33
       }
34
    }
35
    /**
36
     * 账户操作类
37
38
39
    class AccountOperator implements Runnable{
40
       private Account account;
41
       public AccountOperator(Account account) {
42
          this.account = account;
43
       }
44
45
       public void run() {
46
          synchronized (account) {
47
             account.deposit(500);
48
             account.withdraw(500);
             System.out.println(Thread.currentThread().getName() + ":" +
49
    account.getBalance());
50
          }
51
       }
52
    }
53
54
    public class Demo00{
55
        //public static final Object signal = new Object(); // 线程间通信变量
56
57
        //将account改为Demo00.signal也能实现线程同步
58
        public static void main(String args[]){
59
            Account account = new Account("zhang san", 10000.0f);
60
            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:

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

```
/**
 1
    * 同步线程
 2
     */
 3
    class SyncThread implements Runnable {
 4
 5
       private static int count;
 6
 7
       public SyncThread() {
 8
          count = 0;
 9
10
       public synchronized static void method() {
11
12
          for (int i = 0; i < 5; i ++) {
13
                System.out.println(Thread.currentThread().getName() + ":" +
14
    (count++));
                Thread.sleep(100);
15
16
             } catch (InterruptedException e) {
17
                e.printStackTrace();
18
             }
19
          }
       }
20
21
22
       public synchronized void run() {
23
          method();
24
       }
25
    }
26
27
    public class Demo00{
28
        public static void main(String args[]){
29
30
            SyncThread syncThread1 = new SyncThread();
31
            SyncThread syncThread();
            Thread thread1 = new Thread(syncThread1, "SyncThread1");
32
            Thread thread2 = new Thread(syncThread2, "SyncThread2");
33
34
            thread1.start();
35
            thread2.start();
36
        }
37 }
```

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
    }
}
```

```
/**
1
2
     * 同步线程
     */
 3
    class SyncThread implements Runnable {
 5
       private static int count;
 6
 7
       public SyncThread() {
8
          count = 0;
9
10
       public static void method() {
11
12
          synchronized(SyncThread.class) {
13
             for (int i = 0; i < 5; i ++) {
14
                try {
15
                   System.out.println(Thread.currentThread().getName() + ":" +
    (count++));
                   Thread.sleep(100);
16
17
                } catch (InterruptedException e) {
                   e.printStackTrace();
18
19
             }
20
21
          }
22
       }
23
       public synchronized void run() {
24
25
          method();
26
       }
27
   }
```

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

```
1 // 获取锁
2
   void lock()
   // 如果当前线程未被中断,则获取锁,可以响应中断
4
5
   void lockInterruptibly()
6
7
   // 返回绑定到此 Lock 实例的新 Condition 实例
8
   Condition newCondition()
9
10
   // 仅在调用时锁为空闲状态才获取该锁,可以响应中断
11
   boolean tryLock()
12
13
   // 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)
14
15
16
   // 释放锁
17
   void unlock()
```

1.普通用法

```
public static void testlock() {
 1
 2
            Lock lock = new ReentrantLock();
 3
            Thread t = new Thread(new Runnable() {
4
            @override
             public void run() {
 6
                 // TODO Auto-generated method stub
 7
                 lock.lock();
                try {
8
9
                     Thread.sleep(1000);
                     System.out.println("goon");
10
11
                 } catch (InterruptedException e) {
12
                     // TODO Auto-generated catch block
13
                     e.printStackTrace();
14
                } finally {
                     lock.unlock();
15
16
                 }
17
            }
18
19
        });
20
```

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

```
1 start
2 goon
3 over
```

2.trylock

```
1
    public static void testtry() {
 2
            Lock lock = new ReentrantLock();
 3
            Thread t = new Thread(new Runnable() {
 4
            @override
 5
            public void run() {
                 // TODO Auto-generated method stub
 6
 7
                 lock.lock();
 8
                 System.out.println("get");
 9
                 try {
10
                     Thread.sleep(1000);
11
12
                 } catch (InterruptedException e) {
13
                     // TODO Auto-generated catch block
14
                     e.printStackTrace();
15
                 } finally {
16
                     lock.unlock();
17
                     System.out.println("release");
18
                 }
19
            }
20
        });
21
22
        Thread t1 = new Thread(new Runnable() {
23
24
            @override
25
             public void run() {
26
                 // TODO Auto-generated method stub
27
                 try {
28
29
                     Thread.sleep(100);
30
                 } catch (InterruptedException e) {
31
                     // TODO Auto-generated catch block
32
                     e.printStackTrace();
33
                 }
34
35
                 while (true) {
36
                     if (lock.tryLock()) {
37
                             System.out.println("get success");
38
                             lock.unlock();
                             break;
39
40
                     }else {
                         System.out.println("get fail ... ");
41
42
                         try {
43
                             Thread.sleep(100);
```

```
44
                         } catch (InterruptedException e) {
45
                              // TODO Auto-generated catch block
46
                              e.printStackTrace();
                         }
47
48
                     }
49
                }
            }
50
51
52
        });
53
54
        t.start();
55
        t1.start();
56
57
   }
```

```
1
    get
 2
    get fail ...
 3
    get fail ...
    get fail ...
 5
    get fail ...
 6
    get fail ...
    get fail ...
 8
    get fail ...
9
    get fail ...
10
    get fail ...
11
    release
12
    get success
```

3.interruptlock

```
public static void testinterrupt() {
1
 2
            Lock lock = new ReentrantLock();
 3
            Thread t = new Thread(new Runnable() {
4
            @override
 5
            public void run() {
6
                // TODO Auto-generated method stub
 7
                lock.lock();
8
                try {
9
                     Thread.sleep(10000);
                     System.out.println("goon ...");
10
11
                } catch (InterruptedException e) {
12
                     // TODO Auto-generated catch block
13
                     e.printStackTrace();
                } finally {
14
15
                     lock.unlock();
16
                }
17
            }
18
19
        });
20
21
        Thread t1 = new Thread(new Runnable() {
22
23
            @override
            public void run() {
24
25
                // TODO Auto-generated method stub
26
                try {
```

```
27
                     lock.lockInterruptibly();
28
                     System.out.println("get ...");
29
                     lock.unlock();
30
                 } catch (InterruptedException e1) {
31
                     // TODO Auto-generated catch block
32
                     //e1.printStackTrace();
33
                     System.out.println("interrupt ... ");
34
35
                 }
36
37
            }
38
39
        });
40
41
        t.start();
42
        t1.start();
43
44
        try {
            Thread.sleep(5000);
45
46
        }catch (InterruptedException e) {
            // TODO Auto-generated catch block
47
             e.printStackTrace();
48
49
        }
50
51
        System.out.println("to interrupt ");
        t1.interrupt();
52
53
    }
```

```
1 to interrupt
2 interrupt ...
3 goon ...
```

4.delay

```
public static void testdelay() {
 1
            Lock lock = new ReentrantLock();
 2
 3
            Thread t = new Thread(new Runnable() {
            @override
 4
            public void run() {
 5
 6
                 // TODO Auto-generated method stub
 7
                 lock.lock();
 8
                try {
 9
                     Thread.sleep(10000);
10
                     System.out.println("goon ...");
11
                 } catch (InterruptedException e) {
12
                     // TODO Auto-generated catch block
13
                     e.printStackTrace();
14
                 } finally {
15
                     lock.unlock();
                 }
16
17
            }
18
19
        });
20
21
        Thread t1 = new Thread(new Runnable() {
22
```

```
23
            @override
24
            public void run() {
                // TODO Auto-generated method stub
25
                try {
26
                    if(lock.tryLock(5, TimeUnit.SECONDS)) {
27
28
                        System.out.println("get ...");
29
                        lock.unlock();
30
                    }else {
                        System.out.println("have not get ...");
31
32
33
                } catch (InterruptedException e1) {
34
                    // TODO Auto-generated catch block
35
                    //e1.printStackTrace();
36
37
                    System.out.println("interrupt ... ");
                }
38
39
40
            }
41
        });
42
43
44
        t.start();
45
        t1.start();
46 }
```

```
1 have not get ...
2 goon ...
```