线程安全问题

一，什么是线程安全问题

多个线程访问一个对象，不考虑这些线程在运行环境下的调度和交替运行，也不需要进行额外的同步，或者调用方进行其他的协调操作，调用这个对象的行为都可以获得正确的结果。那么这个对象是线程安全的。

二，synchronized的使用

- 该关键字主要包含了两个特征：

  互斥性：保证在同一时刻，只有一个线程可以执行synchronized修饰的方法和代码块。

  可见性：保证了线程工作内存中的变量与公共内存中的变量同步，使得公共内存中的变量被其他线程读取都是最新的结果。

- **使用synchronized关键字**

  该关键字主要对类对象，类方法，和对Object类型的字段加锁。

- 对类方法进行加锁(对象锁)

  当多个线程调用同一个对象实例的方法时，synchronized会对该对象实例进行加锁，使得每一时刻保证只有一个线程在使用这个对象实例。如果不加锁的话，那么多个线程将会一起使用这个对象实例。但是其他线程可以访问非synchronized的方法，因为那些方法不需要获取对象的锁。

```
import java.util.concurrent.ExecutionException;

public class T1 {
    public static void main(String[] args) {
        Test1 x = new Test1();
        new Thread("Thread1") {
            @Override
            public void run() {
                x.T();
            }
        }.start();
        new Thread("Thread2") {
            @Override
            public void run() {
                x.T();
            }
        }.start();
    }

}

class Test1 {
    public synchronized void T() {
        for (int i = 1; i <= 5; i++)
        {
```

```
                System.out.println(Thread.currentThread().getName() + "调用T方法" + i +
"次");
            }
        }
}
```

- 对Object对象加锁(对代码块加锁)

  对于这种方式，它的锁粒度是要小于上面的那种方式的，因为这种方式只是锁了一个代码块，不会锁住整个方法。

```
import java.util.concurrent.ExecutionException;

public class _T2 {
    public static void main(String[] args) {
        Test2 x = new Test2();
        new Thread("Thread1") {
            @Override
            public void run() {
                x.T();
            }
        }.start();
        new Thread("Thread2") {
            @Override
            public void run() {
                x.T();
            }
        }.start();
    }

}

class Test2 {
    final Object obj = new Object();
    public void T() {
        System.out.println(Thread.currentThread().getName()+"使用");
        synchronized (this) {
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getName() + "调用了T方法"
+ i + "次");
            }
        }
    }
}
```

  运行结果：

  Thread1使用 Thread1调用了T方法1次 Thread1调用了T方法2次 Thread2使用(这个就是由于synchronized锁的不是整个方法，其他进程是可以进入到该方法中，但是不可以进入到方法块内) Thread1调用了T方法3次 Thread1调用了T方法4次 Thread1调用了T方法5次 Thread2调用了T方法1次 Thread2调用了T方法2次 Thread2调用了T方法3次 Thread2调用了T方法4次 Thread2调用了T方法5次

- 对类对象进行加锁(对那些用static修饰的方法加锁)

每个类都有class锁，它是对于那些synchronized static 修饰而言的，当一个线程调用非static的方法时，与调用static方法的线程之间并不会产生互斥关系，因为一个用的是对象锁，一个用的是class锁，两者之间并没有互斥的关系。

```
public class _T3{
    public static void main(String[] args){

        Test3 x=new Test3();

        new Thread("Thread1"){
            @Override
            public void run() {
                Test3.T1();
            }
        }.start();
        new Thread("Thread2"){
            @Override
            public void run() {
                x.t();
            }
        }.start();
        new Thread("Thread3"){
          @Override
          public void run() {
              Test3.T2();
          }
        }.start();

    }
}
class Test3{
    public synchronized static void T1(){
        for(int i=1;i<=100;i++){
            System.out.println(Thread.currentThread().getName()+"调用了T1方
法"+i+"次");
        }
    }
    public synchronized static void T2(){
        for(int i=1;i<=2;i++){
            System.out.println(Thread.currentThread().getName()+"调用了T2方
法"+i+"次");
        }
    }
    public void t(){
        for(int i=1;i<=100;i++){
            System.out.println(Thread.currentThread().getName()+"调用了t方
法"+i+"次");
        }
    }
}
```

结果中可以看出只有当Thread1完成之后，Thread3才开始执行。而Thread1并不会影响Thread2的执行，因为Thread2使用的是对象锁。

三，可重入性

该性质就是当一个线程获取某个锁的时候，如果该锁已经被该线程所获取到，那么该线程也是可以继续获取这个锁的。

四,并发编程中的三个概念

- **原子性**：对于一个操作或者多个操作，要么全部都做并且在操作的执行过程中不被外界因素所破坏，要么全部都不执行。
- **有序性**：程序执行的顺序按照代码的先后顺序执行。(主要是针对指令重排序问题而言的，虽然重排序对于单个线程而言，不会出现什么问题，但是对于多个线程而言就不满足了)
- **可见性**：是指多个线程访问同一个变量的时候，当其中的一个线程对该变量进行了修改之后，其他线程能够及时的看到该变量的变化。

五，volatile关键字

- 该关键字保证了有序性(禁止了指令的重排序)，可见性(及时的将新的结果写入到公共内存当中)，但是没有保证原子性，也就是说，在使用volatitle关键字的时候，一定要保证操作的原子性。
- volatile和synchronized的区别

  volatile再某些情况下的性能是要由于synchrnoized的，但是volatile是无法替代synchrnoized，原因是volatile无法保证操作的原子性，因此就没有办法保证线程的安全。

  volatile使用情景：1，对变量的写操作不依赖于当前值

  2，该变量没有包含在具有其他变量的不变式中

六，Lock

1，引入Lock的原因如下

- 当一个线程由于某种原因进入了阻塞状态，如果该线程阻塞的时间过长的话，其他的线程就要在这一直等，这极大的降低了程序的执行的效率，使用Lock中的tryLock(long time,TimeUnit unit)就可以是那些等待的线程不一直等下去。
- 读文件的时候，如果使用synchrnoized，那么任何一个时刻都只能有一个线程去读文件，但是当多个线程只是进行读操作的时候，那么这种方式的效率就有点低，需要一种支持多个线程都只进行读操作的时候，线程之间不会出现冲突的现象。Lock中的ReentrantReadWriteLock。
- 可以使用Lock得知线程有没有成功获取到锁，但是synchronized是无法办到的。

Lock是java.util.concurrent.locks包下的接口，它的优点具有是更加灵活，更加广泛，粒度更细的锁操作。

两者之间的一些比较：

- synchronized是java的关键字，是基于JVM层面上实现的。并且会自动释放锁，不用用户自己手动去释放锁。
- Lock是一个java接口，是基于JDK层面上实现的，通过这个接口可以实现同步访问。并且Lock必须要用户手动去释放锁。

2，Lock接口中函数的介绍

- lock()函数：是获取锁的一种方式，一般配合try和catch使用，在finally中将锁释放掉，防止死锁的发生。

```
Lock lock = new Lock();
lock.lock()
try{
        //处理事务语句
} catch(Exception e){

} finally{
    object.unlock();
}
```

- tryLock和tryLock(long time,TimeUnit unit)

  tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，否则返回false，tryLock(long time,TimeUnit unit)方法是和tryLock方法差不多，但是这个方法在拿不到锁的情况下会等待一段时间，然后再返回true或者false。

```
Lock lock = ...;
if(lock.tryLock()) {
     try{
         //处理任务
     }catch(Exception ex){

     }finally{
         lock.unlock();   //释放锁
     }
}else {
    //如果不能获取锁，则直接做其他事情
}
```

3,ReentrantLock

- 可重入锁，继承了Lock接口

  使用(Lock获取锁)：

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class _T4 {
    private Lock lock = new ReentrantLock();
    public static void main(String[] args) {
        _T4 x = new _T4();
        new Thread("Thread1") {
            public void run() {
                x.T1();
            }
        }.start();
        new Thread("Thread2"){
            @Override
            public void run() {
                x.T1();
            }
        }.start();
```

```
    }
    private void T1() {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "执行了T1方法");
            try {
                Thread.currentThread().sleep(1000);
            } catch (Exception e) {
                // TODO: handle exception
            }
        } catch (Exception e) {

        } finally {
            lock.unlock();
            System.out.println(Thread.currentThread().getName() + "释放了对象锁");
        }
    }
}
```

使用tryLock方式来获取锁。

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class _T5{
    private Lock lock=new ReentrantLock();
    public static void main(String[] args){
        _T5 x=new _T5();
        new Thread("Thread1"){
            @Override
            public void run() {
                x.T1();
            }
        }.start();
        new Thread("Thread2"){
            @Override
            public void run() {
                x.T1();
            }
        }.start();
    }

    public void T1(){
        if(lock.tryLock()){
            try{
                System.out.println(Thread.currentThread().getName()+"调用T1方法");
                try{
                    Thread.currentThread().sleep(1000);
                } catch (Exception e){

                }
            } catch (Exception e){
```

```
            } finally{
                lock.unlock();
                System.out.println(Thread.currentThread().getName()+"释放锁资源");
            }
        } else{
            System.out.println(Thread.currentThread().getName()+"获取锁失败");
        }
    }
}
```

结果为：

Thread1调用T1方法 Thread2获取锁失败 Thread1释放锁资源

这是预料之中的，由于Thread1线程占用了x对象的锁，并且还让Thread1线程占用x对象锁一秒钟，所以Thread2线程在获取x对象锁的时候是无法获取的，因此输出获取锁失败。