正文

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。

synchronized同步方法

synchronized是Java语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。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。

synchronized的使用

- synchronized代码块，被修饰的代码成为同步语句块，其作用的范围是调用这个代码块的对象，我们在用synchronized关键字的时候，能缩小代码段的范围就尽量缩小，能在代码段上加同步就不要再整个方法上加同步。这叫减小锁的粒度，使代码更大程度的并发。
- synchronized方法，被修饰的方法成为同步方法，其作用范围是整个方法，作用对象是调用这个方法的对象。
- synchronized静态方法，修饰一个static静态方法，其作用范围是整个静态方法，作用对象是这个类的所有对象。
- synchronized类，其作用范围是Synchronized后面括号括起来的部分synchronized(className.class)，作用的对象是这个类的所有对象。

- synchronized()，()中是锁住的对象， synchronized(this)锁住的只是对象本身，同一个类的不同对象调用的synchronized方法并不会被锁住，而synchronized(className.class)实现了全局锁的功能，所有这个类的对象调用这个方法都受到锁的影响，此外()中还可以添加一个具体的对象，实现给具体对象加锁。

```
synchronized (object) {
//在同步代码块中对对象进行操作
}
```

synchronized注意事项

- 当两个并发线程访问同一个对象中的`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。两个线程间是互斥的，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。
- 当一个线程访问`object`的一个`synchronized(this)`同步代码块时，另一个线程仍然可以访问该`object`中的非`synchronized(this)`同步代码块。(两个线程使用的是同一个对象)
- 当一个线程访问`object`的一个`synchronized(this)`同步代码块时，其他线程对`object`中所有其它`synchronized(this)`同步代码块的访问将被阻塞(同上，两个线程使用的是同一个对象)。

下面通过代码来实现：

1）当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程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。

```
package ths;

public class Thread1 implements Runnable {
     public void run() {
          synchronized(this) {
               for (int i = 0; i < 5; i++) {
                    System.out.println(Thread.currentThread().getName() + "
synchronized loop " + i);
               }
          }
     }
     public static void main(String[] args) {
          Thread1 t1 = new Thread1();
          Thread ta = new Thread(t1, "A");
          Thread tb = new Thread(t1, "B");
          ta.start();
          tb.start();
     }
}
```

输出结果：

```
A synchronized loop 0
A synchronized loop 1
A synchronized loop 2
A synchronized loop 3
A synchronized loop 4
B synchronized loop 0
B synchronized loop 1
B synchronized loop 2
B synchronized loop 3
B synchronized loop 4
```

2）然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。

```
package ths;

public class Thread2 {
     public void m4t1() {
          synchronized(this) {
               int i = 5;
               while( i-- > 0) {
                    System.out.println(Thread.currentThread().getName() + "
: " + i);
                    try {
                         Thread.sleep(500);
                    } catch (InterruptedException ie) {
                    }
               }
          }
     }
     public void m4t2() {
          int i = 5;
          while( i-- > 0) {
               System.out.println(Thread.currentThread().getName() + " : "
+ i);
               try {
                    Thread.sleep(500);
               } catch (InterruptedException ie) {
               }
          }
     }
     public static void main(String[] args) {
          final Thread2 myt2 = new Thread2();
          Thread t1 = new Thread(  new Runnable() {  public void run() {
 myt2.m4t1();  }  }, "t1"  );
          Thread t2 = new Thread(  new Runnable() {  public void run() {
myt2.m4t2();   }  }, "t2"  );
          t1.start();
          t2.start();
     }
}
```

输出结果：

```
t1 : 4
t2 : 4
t1 : 3
t2 : 3
t1 : 2
t2 : 2
t1 : 1
t2 : 1
t1 : 0
t2 : 0
```

3)尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞。

```
//修改Thread2.m4t2()方法:
     public void m4t2() {
          synchronized(this) {
               int i = 5;
               while( i-- > 0) {
                    System.out.println(Thread.currentThread().getName() + "
: " + i);
                    try {
                         Thread.sleep(500);
                    } catch (InterruptedException ie) {
                    }
               }
          }

     }
```

输出结果:

```
t1 : 4
t1 : 3
t1 : 2
t1 : 1
t1 : 0
t2 : 4
t2 : 3
t2 : 2
t2 : 1
t2 : 0
```

4)第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。

```
 //修改Thread2.m4t2()方法如下：

     public synchronized void m4t2() {
          int i = 5;
          while( i-- > 0) {
               System.out.println(Thread.currentThread().getName() + " : "
+ i);
               try {
                    Thread.sleep(500);
               } catch (InterruptedException ie) {
               }
          }
     }
```

输出结果：

```
t1 : 4
t1 : 3
t1 : 2
t1 : 1
t1 : 0
t2 : 4
t2 : 3
t2 : 2
t2 : 1
t2 : 0
```

5)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。 代码如下：

```
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");
    }
}
```

输出结果：

```
执行insert
执行insert1
执行insert1完毕
执行insert完毕
```

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。

面试题

当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B? 答：不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的synchronized修饰符要求执行方法时要获得对象的锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（**注意不是等待池哦**）中等待对象的锁。

synchronized关键字的用法? 答：synchronized关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问，可以用synchronized(对象) { ... }定义同步代码块，或者在声明方法时将synchronized作为方法的修饰符。

简述synchronized 和java.util.concurrent.locks.Lock的异同? 答：Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；主要不同点：Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally 块中释放（这是释放外部资源的最好的地方）