带你一次搞定 Java 多线程(V)

4 线程间的通信

4.1 等待/通知机制

4.1.3 interrupt()方法会中断 wait()

当线程处于 wait()等待状态时，调用线程对象的 interrupt()方法会中断线程的等待状态，会产生 InterruptedException 异常

```
package com.wkcto.wait;

/**
 * Interrupt()会中断线程的 wait()等待
 * 北京动力节点老崔
 */
public class Test05 {
    public static void main(String[] args) throws InterruptedException {
        SubThread t = new SubThread();
        t.start();

        Thread.sleep(2000);        //主线程睡眠 2 秒，确保子线程处于 Wait 等待状态
        t.interrupt();
    }

    private  static final  Object LOCK = new Object();        //定义常量作为锁对象

    static  class  SubThread extends  Thread{
        @Override
        public void run() {
            synchronized (LOCK){
```

```
                try {
                    System.out.println("begin wait...");
                    LOCK.wait();
                    System.out.println("end wait..");
                } catch (InterruptedException e) {
                    System.out.println("wait 等待被中断了****");
                }
            }
        }
    }
}
```

4.1.4 notify()与 notifyAll()

notify()一次只能唤醒一个线程,如果有多个等待的线程,只能随机唤醒其中的某一个; 想要唤醒所有等待线程,需要调用 notifyAll().

```
package com.wkcto.wait;

/**
 * notify()与 notifyAll()
 * 北京动力节点老崔
 */
public class Test06 {
    public static void main(String[] args) throws InterruptedException {
        Object lock = new Object();          //定义一个对象作为子线程的锁对象
        SubThread t1 = new SubThread(lock);
        SubThread t2 = new SubThread(lock);
        SubThread t3 = new SubThread(lock);
        t1.setName("t1");
        t2.setName("t2");
        t3.setName("t3");
        t1.start();
```

```
            t2.start();
            t3.start();

            Thread.sleep(2000);
            //调用 notify()唤醒  子线程
            synchronized (lock){
//                  lock.notify();          //调用一次 notify()只能唤醒其中的一个线程,其他等待的线程依然处于等待状态,对于处于等待状态的线程来说,错过了通知信号,这种现象也称为信号丢失
                    lock.notifyAll();           //唤醒所有的线程
            }
        }

    static   class   SubThread extends   Thread{
            private Object lock;            //定义实例变量作为锁对象
            public SubThread(Object lock) {
                this.lock = lock;
            }

            @Override
            public void run() {
                synchronized (lock){
                    try {
                        System.out.println(Thread.currentThread().getName()   +   "   --   begin wait...");
                        lock.wait();
                        System.out.println(   Thread.currentThread().getName()   +   "   --   end wait...");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
}
```

4.1.5 wait(long)的使用

wait(long)带有 long 类型参数的 wait()等待,如果在参数指定的时间内没有被唤醒,超时后会自动唤醒.

```
package com.wkcto.wait;

/**
 * wait(long)
 * 北京动力节点老崔
 */
public class Test07 {
    public static void main(String[] args) {
        final Object obj = new Object();
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized ( obj ){
                    try {
                        System.out.println("thread begin wait");
                        obj.wait(5000);              //如果 5000 毫秒内没有被唤醒 ,会自动
唤醒
                        System.out.println("end wait....");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
        t.start();
    }
}
```

4.1.6 通知过早

线程 wait()等待后,可以调用 notify()唤醒线程, 如果 notify()唤醒的过早,在等待之前就调用了 notify()可能会打乱程序正常的运行逻辑.

```
package com.wkcto.wait;

/**
 * notify()通知过早

 * 北京动力节点老崔
 */
public class Test08 {
    public static void main(String[] args) {
        final   Object Lock = new Object();         //定义对象作为锁对象
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (Lock){
                    try {
                        System.out.println("begin wait");
                        Lock.wait();
                        System.out.println("wait end...");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (Lock){
                    System.out.println("begin notify");
                    Lock.notify();;
                    System.out.println("end nofity");
                }
            }
```

```
        });

        //如果先开启 t1,再开启 t2 线程,大多数情况下, t1 先等待,t1 再把 t1 唤醒
//          t1.start();
//          t2.start();

        //如果先开启 t2 通知线程,再开启 t1 等待线程,可能会出现 t1 线程等待没有收到通知的情况,
        t2.start();
        t1.start();

    }
}
```

```
package com.wkcto.wait;

/**
 * notify()通知过早，就不让线程等待了
 * 北京动力节点老崔
 */
public class Test09 {
    static boolean isFirst = true;          //定义静态变量作为是否第一个运行的线程标志
    public static void main(String[] args) {
        final    Object Lock = new Object();          //定义对象作为锁对象

        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (Lock){
                    while ( isFirst ) {              //当线程是第一个开启的线程就等待
                        try {
                            System.out.println("begin wait");
                            Lock.wait();
                            System.out.println("wait end...");
```

```
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (Lock){
                    System.out.println("begin notify");
                    Lock.notify();;
                    System.out.println("end nofity");

                    isFirst = false;            //通知后,就把第一个线程标志修改为 false

                }
            }
        });

        //如果先开启 t1,再开启 t2 线程,大多数情况下, t1 先等待,t1 再把 t1 唤醒

//          t1.start();
//          t2.start();


        //如果先开启 t2 通知线程,再开启 t1 等待线程,可能会出现 t1 线程等待没有收到通知的情况,

        t2.start();
        t1.start();


        //实际上,调用 start()就是告诉线程调度器,当前线程准备就绪,线程调度器在什么时候开启这个线程不确定,即调用 start()方法的顺序,并不一定就是线程实际开启的顺序.

        //在当前示例中,t1 等待后让 t2 线程唤醒 ， 如果 t2 线程先唤醒了,就不让 t1 线程等待了
```

```
    }
}
```

4.1.7 wait 等待条件发生了变化

在使用 wait/nofity 模式时,注意 wait 条件发生了变化,也可能会造成逻辑的混乱

```
package com.wkcto.wait;

import java.util.ArrayList;
import java.util.List;

/**
 * wait 条件发生变化
 *  *  定义一个集合
 *  *  定义一个线程向集合中添加数据,添加完数据后通知另外的线程从集合中取数据
 *  *  定义一个线程从集合中取数据,如果集合中没有数据就等待
 *  * 北京动力节点老崔
 */
public class Test10 {
    public static void main(String[] args) {
        //定义添加数据的线程对象
        ThreadAdd threadAdd = new ThreadAdd();
        //定义取数据的线程对象
        ThreadSubtract threadSubtract = new ThreadSubtract();
        threadSubtract.setName("subtract 1 ");
        //测试一: 先开启添加数据的线程,再开启一个取数据的线程,大多数情况下会正常
```

取数据

```
//            threadAdd.start();
//            threadSubtract.start();
```

//测试二: 先开启取数据的线程,再开启添加数据的线程, 取数据的线程会先等待,等到添加数据之后 ,再取数据

```
//            threadSubtract.start();
//            threadAdd.start();


        //测试三: 开启两个 取数据的线程,再开启添加数据的线程
        ThreadSubtract threadSubtract2 = new ThreadSubtract();
        threadSubtract2.setName("subtract 2 ");
        threadSubtract.start();
        threadSubtract2.start();
        threadAdd.start();
        /*
            某一次执行结果如下:
                subtract 1    begin wait....
                subtract 2  从集合中取了 data 后,集合中数据的数量:0
                subtract 1    end wait..
                Exception in thread "subtract 1 " java.lang.IndexOutOfBoundsException:
            分析可能的执行顺序:
                threadSubtract 线程先启动, 取数据时,集合中没有数据,wait()等待
                threadAdd 线程获得 CPU 执行权, 添加数据 , 把 threadSubtract 线程唤醒,
                threadSubtract2 线程开启后获得 CPU 执行权, 正常取数据
                threadSubtract 线程获得 CPU 执行权, 打印 end wait..., 然后再执行 list.remove(0) 取数据时,现在 list 集合中已经没有数据了, 这时会产生 java.lang.IndexOutOfBoundsException 异常
            出现异常的原因是: 向 list 集合中添加了一个数据,remove()了两次
```

```
            如何解决?

                当等待的线程被唤醒后，再判断一次集合中是否有数据可取. 即需要把
sutract()方法中的 if 判断改为 while
            */
    }

    //1)定义 List 集合
    static List list = new ArrayList<>();

    //2)定义方法从集合中取数据
    public static void subtract(){
        synchronized (list) {
//                  if (list.size() == 0) {
                while (list.size() == 0) {
                    try {
                        System.out.println(Thread.currentThread().getName()  +  "  begin
wait....");
                        list.wait();            //等待
                        System.out.println(Thread.currentThread().getName() + " end wait..");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                Object data = list.remove(0);    //从集合中取出一个数据
                System.out.println( Thread.currentThread().getName() + "从集合中取了" + data +
"后,集合中数据的数量:" + list.size());
            }
    }

    //3)定义方法向集合中添加数据后,通知等待的线程取数据
    public static void add(){
        synchronized (list){
            list.add("data");
```

```
                System.out.println( Thread.currentThread().getName() + "存储了一个数据");
                list.notifyAll();
            }
        }

    //4)定义线程类调用 add()取数据的方法
    static class ThreadAdd extends    Thread{
        @Override
        public void run() {
            add();
        }
    }
    //定义线程类调用 subtract()方法
    static class ThreadSubtract extends    Thread{
        @Override
        public void run() {
            subtract();
        }
    }
}
```

4.1.8 生产者消费者模式

在 Java 中,负责产生数据的模块是生产者,负责使用数据的模块是消费者. 生产者消费者解决数据的平衡问题,即先有数据然后才能使用,没有数据时,消费者需要等待

1 生产-消费:操作值

```
package com.wkcto.producerdata;

/**
```

```
 * 定义一个操作数据的类
 * 北京动力节点老崔
 */
public class ValueOP {
    private String value = "";

    //定义方法修改 value 字段的值
    public void setValue(){
        synchronized ( this ){
            //如果 value 值不是""空串就等待
            while ( !value.equalsIgnoreCase("")){
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            //如果 value 字段值是容串，就设置 value 字段的值
            String value = System.currentTimeMillis() + " - " + System.nanoTime();
            System.out.println("set 设置的值是: " + value);
            this.value = value;
//                this.notify();          //在多生产者多消费者环境中,notify()不能保证是生产者唤醒消费者,如果生产者唤醒的还是生产者可能会出现假死的情况
            this.notifyAll();
        }
    }

    //定义方法读取字段值
    public void getValue(){
        synchronized (this){
            //如果 value 是空串就等待
            while ( value.equalsIgnoreCase("")){
```

```
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            //不是空串,读取  字段值

            System.out.println("get 的值是: " + this.value);

            this.value = "";
            this.notifyAll();
        }
    }
}
```

```
package com.wkcto.producerdata;

/**
 * 定义线程类模拟生产者

 * 北京动力节点老崔

 */
public class ProducerThread extends Thread {

    //生产者生产数据就是调用 ValueOP 类的 setValue 方法给 value 字段赋值

    private ValueOP obj;

    public ProducerThread(ValueOP obj) {
        this.obj = obj;
    }

    @Override
    public void run() {
        while (true){
            obj.setValue();
        }
    }
}
```

```
package com.wkcto.producerdata;
```

```
/**
 * 定义线程类模拟消费者
 * 北京动力节点老崔
 */
public class ConsumerThread extends Thread {
    //消费者使用数据，就是使用 ValueOP 类的 value 字段值
    private ValueOP obj;

    public ConsumerThread(ValueOP obj) {
        this.obj = obj;
    }

    @Override
    public void run() {
        while (true){
            obj.getValue();
        }
    }
}
```

```
package com.wkcto.producerdata;

/**
 * 测试多生产,多消费的情况
 * 北京动力节点老崔
 */
public class Test2 {
    public static void main(String[] args) {
        ValueOP valueOP = new ValueOP();

        ProducerThread p1 = new ProducerThread(valueOP);
        ProducerThread p2 = new ProducerThread(valueOP);
        ProducerThread p3 = new ProducerThread(valueOP);
        ConsumerThread c1 = new ConsumerThread(valueOP);
        ConsumerThread c2 = new ConsumerThread(valueOP);
        ConsumerThread c3 = new ConsumerThread(valueOP);
```

```
            p1.start();
            p2.start();
            p3.start();
            c1.start();
            c2.start();
            c3.start();
        }
}
```

2 操作栈

使生产者把数据存储到 List 集合中，消费者从 List 集合中取数据，使用 List 集合模拟栈.

```
package com.wkcto.produerstack;

import java.util.ArrayList;
import java.util.List;

/**
 * 模拟栈
 * 北京动力节点老崔
 */
public class MyStack {
    private List list = new ArrayList();            //定义集合模拟栈
    private static final    int MAX = 3;          //集合的最大容量

    //定义方法模拟入栈
    public synchronized void push(){
        //当栈中的数据已满 就等待
```

```
            while ( list.size() >= MAX ){
                System.out.println(Thread.currentThread().getName() + " begin    wait....");
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            String data = "data--" + Math.random();

            System.out.println( Thread.currentThread().getName() + "添加了数据: " + data);

            list.add(data);

//              this.notify();                  //当多个生产者多个消费者时,使用 notify()可能会出现

假死的情况

            this.notifyAll();
        }

        //定义方法模拟出栈

        public synchronized void pop(){

            //如果没有数据就等待

            while ( list.size() == 0 ){
                try {
                    System.out.println(Thread.currentThread().getName() + " begin    wait....");
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            System.out.println( Thread.currentThread().getName() + "出栈数据:" + list.remove(0) );

            this.notifyAll();
        }
}
```

```
package com.wkcto.produerstack;

/**
```

```
 * 生产者线程

 * 北京动力节点老崔

 */
public class ProduerThread extends    Thread {
    private    MyStack stack;

    public ProduerThread(MyStack stack) {
        this.stack = stack;
    }

    @Override
    public void run() {
        while (true){
            stack.push();
        }
    }
}
```

```
package com.wkcto.produerstack;

/**

 * 消费线程

 * 北京动力节点老崔

 */
public class ConsumerThread extends    Thread {
    private    MyStack stack;

    public ConsumerThread(MyStack stack) {
        this.stack = stack;
    }

    @Override
    public void run() {
        while (true){
            stack.pop();
        }
    }
}
```

```
package com.wkcto.produerstack;


/**
 * 测试多生产多消费的情况

 * 北京动力节点老崔

 */
public class Test02 {
	public static void main(String[] args) {
		MyStack stack = new MyStack();

		ProduerThread    p = new ProduerThread(stack);
		ProduerThread    p2 = new ProduerThread(stack);
		ProduerThread    p3 = new ProduerThread(stack);
		ConsumerThread c1 = new ConsumerThread(stack);
		ConsumerThread c2 = new ConsumerThread(stack);
		ConsumerThread c3 = new ConsumerThread(stack);

		p.setName("生产者 1 号");

		p2.setName("生产者 2 号");

		p3.setName("生产者 3 号");

		c1.setName("消费者 1 号");

		c2.setName("消费者 2 号");

		c3.setName("消费者 3 号");


		p.start();
		p2.start();
		p3.start();
		c1.start();
		c2.start();
		c3.start();

	}
}
```