java并发总结三

一.实现线程间通信的几种方法

1. Object.wait(),notifyAll

```
package concurrent.Produce_Consumer;

import java.util.Vector;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ProducerConsumerSolutionDemoOne {

    public static void main(String[] args) {
        Vector sharedQueue = new Vector();
        int size = 4;
        Thread prodThread = new Thread(new ProducerThread(sharedQueue, size),
"Producer");
        Thread consThread = new Thread(new ConsumerThread(sharedQueue, size),
"Consumer");

        prodThread.start();
        consThread.start();
    }


}

class ProducerThread implements Runnable {

    private final Vector queue;
    private final int SIZE;

    public ProducerThread(Vector queue, int size) {
        this.queue = queue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        for (int i = 0; i < 7; i++) {
            System.out.println(" Produced: " + i);
            try {
                produce(i);
            } catch (InterruptedException e) {
                Logger.getLogger(ProducerThread.class.getName()).log(Level.SEVERE,
null, e);
```

```
            }
        }
    }

    public void produce(int i) throws InterruptedException {
        // wait if queue is full
        while (queue.size() == SIZE) {
            synchronized (queue) {
                System.out.println("queue is full " +
Thread.currentThread().getName() + " is waiting,size: " + queue.size());
                queue.wait();
            }
        }
        // notify consumer take element
        synchronized (queue) {
            queue.add(i);
            queue.notifyAll();
        }

    }
}

class ConsumerThread implements Runnable {

    private final Vector queue;
    private final int SIZE;

    public ConsumerThread(Vector queue, int size) {
        this.queue = queue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        while (true) {
            try {
                System.out.println("Consumed: " + consume());
                TimeUnit.MILLISECONDS.sleep(50);
            } catch (InterruptedException e) {
                Logger.getLogger(ConsumerThread.class.getName()).log(Level.SEVERE,
null, e);
            }
        }
    }

    public int consume() throws InterruptedException {
        //wait if queue isEmpty
        while (queue.isEmpty()) {
            synchronized (queue) {
                System.out.println("Queue is empty " +
Thread.currentThread().getName()
                        + " is waiting , size: " + queue.size());
```

```
                queue.wait();
            }
        }
        synchronized (queue) {
            queue.notifyAll();
            return (Integer) queue.remove(0);
        }
    }
}
/*
Queue is empty Consumer is waiting , size: 0
 Produced: 0
 Produced: 1
Consumed: 0
 Produced: 2
 Produced: 3
 Produced: 4
 Produced: 5
queue is full Producer is waiting,size: 4
Consumed: 1
 Produced: 6
queue is full Producer is waiting,size: 4
Consumed: 2
Consumed: 3
Consumed: 4
Consumed: 5
Consumed: 6
Queue is empty Consumer is waiting , size: 0

*/
```

2. BlockingQueue

```
public class ProducerConsumerSolution {

    public static void main(String[] args) {
        BlockingQueue<Integer> sharedQ = new LinkedBlockingQueue<Integer>();

        Producer p = new Producer(sharedQ);
        Consumer c = new Consumer(sharedQ);

        p.start();
        c.start();
    }
}

class Producer extends Thread {
    private BlockingQueue<Integer> sharedQueue;

    public Producer(BlockingQueue<Integer> aQueue) {
        super("PRODUCER");
        this.sharedQueue = aQueue;
    }
```

```
    public void run() {
        // no synchronization needed
        for (int i = 0; i < 10; i++) {
            try {
                System.out.println(getName() + " produced " + i);
                sharedQueue.put(i);
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
    }
}

class Consumer extends Thread {
    private BlockingQueue<Integer> sharedQueue;

    public Consumer(BlockingQueue<Integer> aQueue) {
        super("CONSUMER");
        this.sharedQueue = aQueue;
    }

    public void run() {
        try {
            while (true) {
                Integer item = sharedQueue.take();
                System.out.println(getName() + " consumed " + item);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
/*
Output
PRODUCER produced 0
CONSUMER consumed 0
PRODUCER produced 1
CONSUMER consumed 1
PRODUCER produced 2
CONSUMER consumed 2
PRODUCER produced 3
CONSUMER consumed 3
PRODUCER produced 4
CONSUMER consumed 4
PRODUCER produced 5
CONSUMER consumed 5
PRODUCER produced 6
CONSUMER consumed 6
PRODUCER produced 7
CONSUMER consumed 7
```

```
PRODUCER produced 8
CONSUMER consumed 8
PRODUCER produced 9
CONSUMER consumed 9
*/
```

3. Semaphore

- Semaphore是一个计数信号量。
- 从概念上将，Semaphore包含一组许可证。
- 如果有需要的话，每个acquire()方法都会阻塞，直到获取一个可用的许可证。
- 每个release()方法都会释放持有许可证的线程，并且归还Semaphore一个可用的许可证。
- 然而，实际上并没有真实的许可证对象供线程使用，Semaphore只是对可用的数量进行管理维护。

```
package concurrent.Produce_Consumer;

import java.util.Vector;
import java.util.concurrent.Semaphore;

public class ProducerConsumerSolutionDemoTwo {
    public static void main(String[] args) {
        Semaphore notFull = new Semaphore(10);
        Semaphore notEmpty = new Semaphore(0);
        Vector queue = new Vector();
        Producer producer = new Producer("生产者线程", notFull, notEmpty, queue);
        Consumer consumer = new Consumer("消费者线程", notFull, notEmpty, queue);

        producer.start();
        consumer.start();
    }
}

class Producer extends Thread {

    private Semaphore notFull;
    private Semaphore notEmpty;
    //    private Semaphore mutex;
    private Vector queue;
    private final int SIZE = 4;

    //,Semaphore mutex
    public Producer(String name, Semaphore notFull, Semaphore notEmpty, Vector queue) {
        super(name);
        this.notFull = notFull;
        this.notEmpty = notEmpty;
//        this.mutex = mutex;
        this.queue = queue;
    }

    @Override
```

```
    public void run() {
        for (int i = 0; i < 7; i++) {
            try {
                // 非满阻塞
                log(" not full is waiting for permit");
                notFull.acquire();
                log(" acquired a permit");
                log(" add value! ");
//                    mutex.acquire();
                queue.add(i);
                notEmpty.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }

    private void log(String msg) {
        System.out.println(Thread.currentThread().getName() + "  " + msg);
    }

}


class Consumer extends Thread {

    private Semaphore notFull;
    private Semaphore notEmpty;
    //    private Semaphore mutex;
    private Vector queue;
    private final int SIZE = 4;

    //,Semaphore mutex
    public Consumer(String name, Semaphore notFull, Semaphore notEmpty, Vector queue) {
        super(name);
        this.notFull = notFull;
        this.notEmpty = notEmpty;
//        this.mutex = mutex;
        this.queue = queue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 7; i++) {
            try {
                // 非满阻塞
                log(" not empty is waiting for permit");
                notEmpty.acquire();
                log(" acquired a permit");
                log(" getValue! ");
//                    mutex.acquire();
                log(queue.get(i) + "");
```

```
                notFull.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }

    private void log(String msg) {
        System.out.println(Thread.currentThread().getName() + " 消费 " + msg);
    }

}
```