重入锁

ReentrantLock 限时性：判断某个线程**在一定的时间内**能否获取锁，通过 tryLock 方法来实现

tryLock(long time,TimeUnit unit)

time 指时间数值

unit 时间单位

```
package com.southwind.demo;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

public class Test {
    public static void main(String[] args) {
        TimeLock timeLock = new TimeLock();
        /**
         * A 拿到锁，执行业务代码，休眠 5 秒钟
         * B 尝试拿锁，需要在 3 秒钟之内拿到锁
         */
        new Thread(()->{
            timeLock.lock();
        },"A").start();
        new Thread(()->{
            timeLock.lock();
        },"B").start();
    }
}

class TimeLock{

    private ReentrantLock reentrantLock = new ReentrantLock();

    public void lock(){
        /**
         * 尝试在3S内获取锁
         */
        try {
            if(reentrantLock.tryLock(3, TimeUnit.SECONDS)){
                System.out.println(Thread.currentThread().getName()+" get
lock");
                TimeUnit.SECONDS.sleep(5);
            }else{
                System.out.println(Thread.currentThread().getName()+" not
lock");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            if(reentrantLock.isHeldByCurrentThread()){
                reentrantLock.unlock();
            }
        }
    }
}
```

生产者消费者模式

在一个生产环境中，生产者和消费者在同一时间段内共享同一块缓冲区，生产者负责向缓冲区添加数据，消费者负责从缓冲区取出数据。

汉堡类

```
package com.southwind.demo2;

public class Hamburger {
    private int id;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @Override
    public String toString() {
        return "Hamburger{" +
                "id=" + id +
                '}';
    }
}
```

容器类

```
package com.southwind.demo2;

public class Container {
    public Hamburger[] array = new Hamburger[6];
    public int index = 0;
    /**
     * 向容器中添加汉堡
     */
    public synchronized void push(Hamburger hamburger){
        while(index == array.length){
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.notify();
        array[index] = hamburger;
        index++;
        System.out.println("生产类一个汉堡"+hamburger);
    }
    /**
     * 从容器中取出汉堡
     */
    public synchronized Hamburger pop(){
        while(index == 0){
            //当前线程暂停
            //让正在访问当前资源的线程暂停
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //唤醒之前暂停的线程
        this.notify();
        index--;
        System.out.println("消费了一个汉堡"+array[index]);
        return array[index];
    }
}
```

生产者

```
package com.southwind.demo2;

import java.util.concurrent.TimeUnit;

/**
 * 生产者
 */
public class Producer {
    private Container container;
    public Producer(Container container){
        this.container = container;
    }
    public void product(){
        for (int i = 0; i < 30; i++) {
            Hamburger hamburger = new Hamburger(i);
            this.container.push(hamburger);
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

消费者

```
package com.southwind.demo2;

import java.util.concurrent.TimeUnit;

public class Consumer {
    private Container container;

    public Consumer(Container container) {
        this.container = container;
    }

    public void consum(){
        for (int i = 0; i < 30; i++) {
            this.container.pop();
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

测试类

```
package com.southwind.demo2;

public class Test {
    public static void main(String[] args) {
        Container container = new Container();
        Producer producer = new Producer(container);
        Consumer consumer = new Consumer(container);
        new Thread(()->{
            producer.product();
        }).start();
        new Thread(()->{
            producer.product();
        }).start();
        new Thread(()->{
            consumer.consum();
        }).start();
        new Thread(()->{
            consumer.consum();
        }).start();
        new Thread(()->{
            consumer.consum();
        }).start();
    }
}
```

多线程并发卖票

一场球赛的球票分 3 个窗口出售，共 15 张票，用多线程并发来模拟 3 个窗口的售票情况

```
package com.southwind.demo3;

import java.util.concurrent.TimeUnit;

public class Ticket {
    //剩余球票
    private int surpluCount = 15;
    //已售出球票
    private int outCount = 0;

    public synchronized void sale(){
        while(surpluCount > 0){
            try {
                TimeUnit.MILLISECONDS.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if(surpluCount == 0){
                return;
            }
            surpluCount--;
            outCount++;
            if(surpluCount == 0){
                System.out.println(Thread.currentThread().getName()+"售出
第"+outCount+"张票，球票已售罄");
            }else{
                System.out.println(Thread.currentThread().getName()+"售出
第"+outCount+"张票，剩余"+surpluCount+"张票");
```

```
            }
        }
    }
}
```

```java
package com.southwind.demo3;

public class Test {
    public static void main(String[] args) {
        Ticket ticket = new Ticket();
        new Thread(()->{
            ticket.sale();
        },"A").start();

        new Thread(()->{
            ticket.sale();
        },"B").start();

        new Thread(()->{
            ticket.sale();
        },"C").start();
    }
}
```