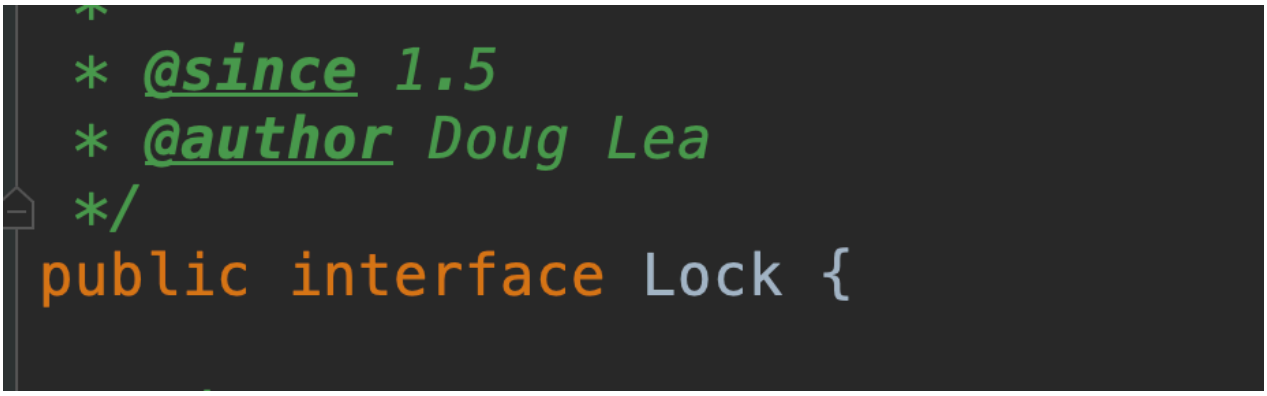
Lock

JUC 提供的一种锁机制，功能和 synchronized 类似，是对 synchronized 的升级，它是一个接口。



它的常用实现类是 ReentrantLock。

synchronized 是通过 JVM 实现锁机制，ReentrantLock 是通过 JDK 实现锁机制。

synchronized 是一个关键字，ReentrantLock 是一个类。

重入锁：可以给同一个资源添加多把锁。

synchronized 是线程执行完毕之后自动释放锁，ReentrantLock 需要手动解锁。

用 synchronized 实现卖票

```
package com.southwind.demo;

import java.util.concurrent.TimeUnit;

public class Test {
    public static void main(String[] args) {
        Ticket ticket = new Ticket();
        new Thread(()->{
            for (int i = 0; i < 40; i++) {
                ticket.sale();
            }
        },"A").start();
        new Thread(()->{
            for (int i = 0; i < 40; i++) {
                ticket.sale();
            }
        },"B").start();
    }
}

class Ticket{
    private Integer saleNum = 0;
    private Integer lastNum = 30;
```

```
    public synchronized void sale(){
        if(lastNum > 0){
            saleNum++;
            lastNum--;
            try {
                TimeUnit.MILLISECONDS.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()+"卖出了
第"+saleNum+"张票，剩余"+lastNum+"张票");
        }
    }

}
```

用 Lock 完成卖票

```
package com.southwind.demo;

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

public class Test2 {
    public static void main(String[] args) {
        Ticket2 ticket = new Ticket2();
        new Thread(()->{
            for (int i = 0; i < 40; i++) {
                ticket.sale();
            }
        },"A").start();
        new Thread(()->{
            for (int i = 0; i < 40; i++) {
                ticket.sale();
            }
        },"B").start();
    }
}

class Ticket2{
    private Integer saleNum = 0;
    private Integer lastNum = 30;
    private Lock lock = new ReentrantLock();

    public void sale(){

        lock.lock();
```

```
            lock.lock();
            if(lastNum > 0){
                saleNum++;
                lastNum--;
                try {
                    TimeUnit.MILLISECONDS.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()+"卖出了
第"+saleNum+"张票，剩余"+lastNum+"张票");
            }
            lock.unlock();
            lock.unlock();
        }
}
```

> synchronized 和 lock 的区别

1、synchronized 自动上锁，自动释放锁，Lock 手动上锁，手动释放锁。

2、synchronized 无法判断是否获取到了锁，Lock 可以判断是否拿到了锁。

3、synchronized 拿不到锁就会一直等待，Lock 不一定会一直等待。

4、synchronized 是 Java 关键字，Lock 是接口。

5、synchronized 是非公平锁，Lock 可以设置是否为公平锁。

公平锁：很公平，排队，当锁没有被占用时，当前线程需要判断队列中是否有其他等待线程。

非公平锁：不公平，插队，当锁没有被占用时，当前线程可以直接占用，而不需要判断当前队列中是否有等待线程。

实际开发中推荐使用 Lock 的方式。

ReentrantLock 具备限时性的特点，可以判断某个线程在一定的时间段内能否获取到锁，使用 tryLock 方法，返回值是 boolean 类型，true 表示可以获取到锁，false 表示无法获取到锁。

```
package com.southwind.demo2;

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

public class Test {
    public static void main(String[] args) {
        TimeLock timeLock = new TimeLock();
        new Thread(()->{
            timeLock.getLock();
        },"A").start();
        new Thread(()->{
```

```
            timeLock.getLock();
        },"B").start();
    }
}

class TimeLock{
    private ReentrantLock lock = new ReentrantLock();

    public void getLock(){
        try {
            if(lock.tryLock(3, TimeUnit.SECONDS)){
                System.out.println(Thread.currentThread().getName()+"拿到了锁");
                TimeUnit.SECONDS.sleep(5);
            }else{
                System.out.println(Thread.currentThread().getName()+"拿不到锁");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            if(lock.isHeldByCurrentThread()){
                lock.unlock();
            }
        }
    }

}
```

生产者消费者模式

synchronized

```
package com.southwind.demo3;

import java.util.concurrent.TimeUnit;

public class Test {
    public static void main(String[] args) {
        Data data = new Data();
        new Thread(()->{
            for (int i = 0; i < 30; i++) {
                data.increment();
            }
        },"A").start();
        new Thread(()->{
            for (int i = 0; i < 30; i++) {
                data.decrement();
            }
        },"B").start();
    }
```

```
}

class Data{
    private Integer num = 0;

    public synchronized void increment(){
        while(num!=0){
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        num++;
        this.notify();
        System.out.println(Thread.currentThread().getName()+"生产了汉堡"+num);
    }

    public synchronized void decrement(){
        while(num == 0){
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        num--;
        this.notify();
        System.out.println(Thread.currentThread().getName()+"消费了汉堡"+num);
    }
}
```

必须使用 while 判断，不能用 if，因为 if 会存在线程虚假唤醒的问题，虚假唤醒就是一些 wait 方法会在除了 notify 的其他情况被唤醒，不是真正的唤醒，使用 while 完成多重检测，避免这一问题。

Lock

```
package com.southwind.demo4;

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

public class Test {
    public static void main(String[] args) {
        Data data = new Data();
        new Thread(()->{
            for (int i = 0; i < 30; i++) {
                data.increment();
```

```
            }
        },"A").start();
        new Thread(()->{
            for (int i = 0; i < 30; i++) {
                data.decrement();
            }
        },"B").start();
    }
}

class Data{
    private Integer num = 0;

    private Lock lock = new ReentrantLock();

    private Condition condition = lock.newCondition();

    public void increment(){
        lock.lock();
        while(num!=0){
            try {
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        num++;
        condition.signal();
        System.out.println(Thread.currentThread().getName()+"生产了汉堡"+num);
        lock.unlock();
    }

    public synchronized void decrement(){
        lock.lock();
        while(num == 0){
            try {
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        num--;
        condition.signal();
        System.out.println(Thread.currentThread().getName()+"消费了汉堡"+num);
        lock.unlock();
    }
}
```

使用 Lock 锁，就不能通过 wait 和 notify 来暂停线程和唤醒线程，而应该使用 Condition 的 await 和 signal 来暂停和唤醒线程。

ConcurrentModificationException

并发访问异常

```
package com.southwind.demo5;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class Test {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                //写
                list.add("a");
                //读
                System.out.println(list);
            },String.valueOf(i)).start();
        }
    }
}
```

如何解决?

1、Vector

```
public synchronized boolean add(E e) {
    modCount++;
    add(e, elementData, elementCount);
    return true;
}
```

```
public boolean add(E e) {
    modCount++;
    add(e, elementData, size);
    return true;
}
```

2、Collections.synchronizedList

3、JUC：CopyOnWriteArrayList

```
package com.southwind.demo5;

import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.TimeUnit;

public class Test2 {
    public static void main(String[] args) {
        List<String> list = new CopyOnWriteArrayList<>();
        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                try {
                    TimeUnit.MILLISECONDS.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                list.add("a");
                System.out.println(list);
            }).start();
        }
    }
}
```

CopyOnWrite 写时复制，当我们往一个容器添加元素的时候，不是直接给容器添加，而是先将当前容器复制一份，向新的容器中添加数据，添加完成之后，再将原容器的引用指向新的容器。

```
*/
public boolean add(E e) {
    synchronized (lock) {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, newLength: len +
        newElements[len] = e;
        setArray(newElements);
        return true;
    }
}
```

Set

```
package com.southwind.demo5;

import java.util.List;
import java.util.Set;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.CopyOnWriteArraySet;
import java.util.concurrent.TimeUnit;

public class Test2 {
    public static void main(String[] args) {
        Set<String> set = new CopyOnWriteArraySet<>();
        for (int i = 0; i < 10; i++) {
            final int temp = i;
            new Thread(()->{
                try {
                    TimeUnit.MILLISECONDS.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                set.add(String.valueOf(temp)+"a");
                System.out.println(set);
            })).start();
        }
    }
}
```

Map

```
package com.southwind.demo5;

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.CopyOnWriteArraySet;
import java.util.concurrent.TimeUnit;

public class Test2 {
    public static void main(String[] args) {
        Map<String,String> map = new ConcurrentHashMap<>();
        for (int i = 0; i < 10; i++) {
```

```
            final int temp = i;
            new Thread(()->{
                try {
                    TimeUnit.MILLISECONDS.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

 map.put(UUID.randomUUID().toString().substring(0,3),UUID.randomUUID().toStrin
g().substring(0,2));
                System.out.println(map);
            })).start();
        }
    }
}
```