* 学习目标

  * 能够理解多线程安全问题

    * 多个线程访问成员变量|静态变量，出现数据安全问题

    * 售票的例子

  * 能够掌握解决多线程安全问题常见的方式

    * 加锁机制

      * synchronized:同步代码块，同步方法

        * 可以加把锁：任意类型对象,this

        * 同步方法:静态方法，锁：类型的class对象

      * ReentrantLock(重入锁)

        * Lock lock=new ReentrantLock();

        * lock.lock();

        * lock.unLock();

  * 能够掌握ReentrantLock锁的使用

      * 公平锁功能

          * Lock lock=new ReentrantLock(true);

          * 非公平锁

      * 可以中断死锁

          * lockInterruptBy()

      * tryLock(1,TimeUtils.SECONDS)

       * Condition-await,singal,singalALl(wait,notify,notfiyAll)

  * 能够掌握多线程的生产者和消费者模式

      * 用一个缓冲区（集合，队列）解耦生成者和消费者

      * sychronized-->wait，notify,notifyALl-->List

      * ReentrantLock-lock,unlock,Condition-await-singal-singalAll

---

  * 回顾

* 多线程
  * 客户端，服务端
  * 并发，并行
  * 程序，进程，线程
  * Java内置多线程支持
  * 线程创建和调度
    new Thread(){run}.start();
    new Thread(new Runnable(){run}).start();
    lamda表达式
    new Thread(()->{...}).start();
    run 和 start的区别
* 线程生命周期
    * 新建状态(new 线程 到start之前)
    * 就绪状态(start())
    * 运行状态（run）yield()
    * 阻塞状态（sleep，IO，join，同步锁,wait,...）
    * 死亡状态(run 跑完)
* Thread 与 Runnable的区别
    * Thread 线程而Runnbale不是线程
    * Java是单继承
* Thread 常见的方法
    * setName,getName,setProirity,getProirity
    * 守护线程：setDaemon(true)
        * jvm-用户线程--调用main方法
        * jvm-守护线程- gc
    * interrupt:中断线程
        * Thread.sleep();
    * join:等待线程去死。t.join(2000)

* 图片下载线程，图片显示线程

* wait，notify

* Object 的方法

* 要在同步方法或者同步代码块里执行（需要一把锁）

* yield

* 线程启动和停止

* 停止正在运行的线程或者阻塞的线程

* 共享变量

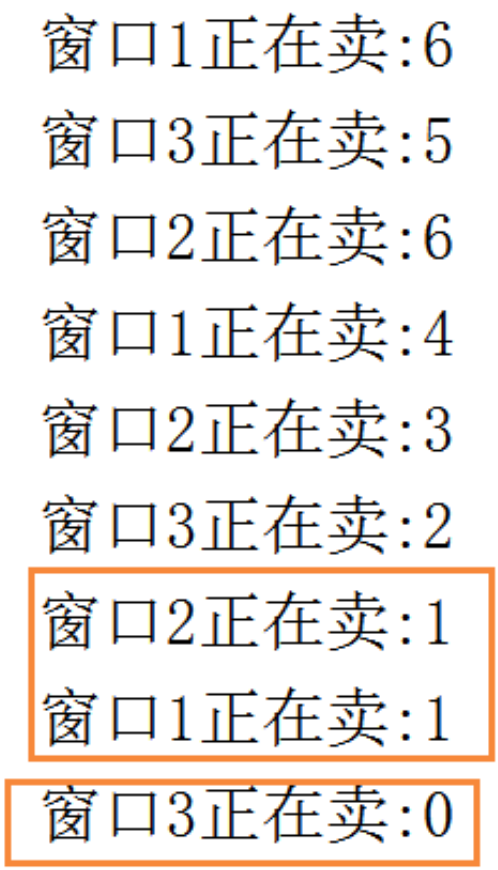
* interrupt+ 共享变量

* 能够理解多线程安全问题

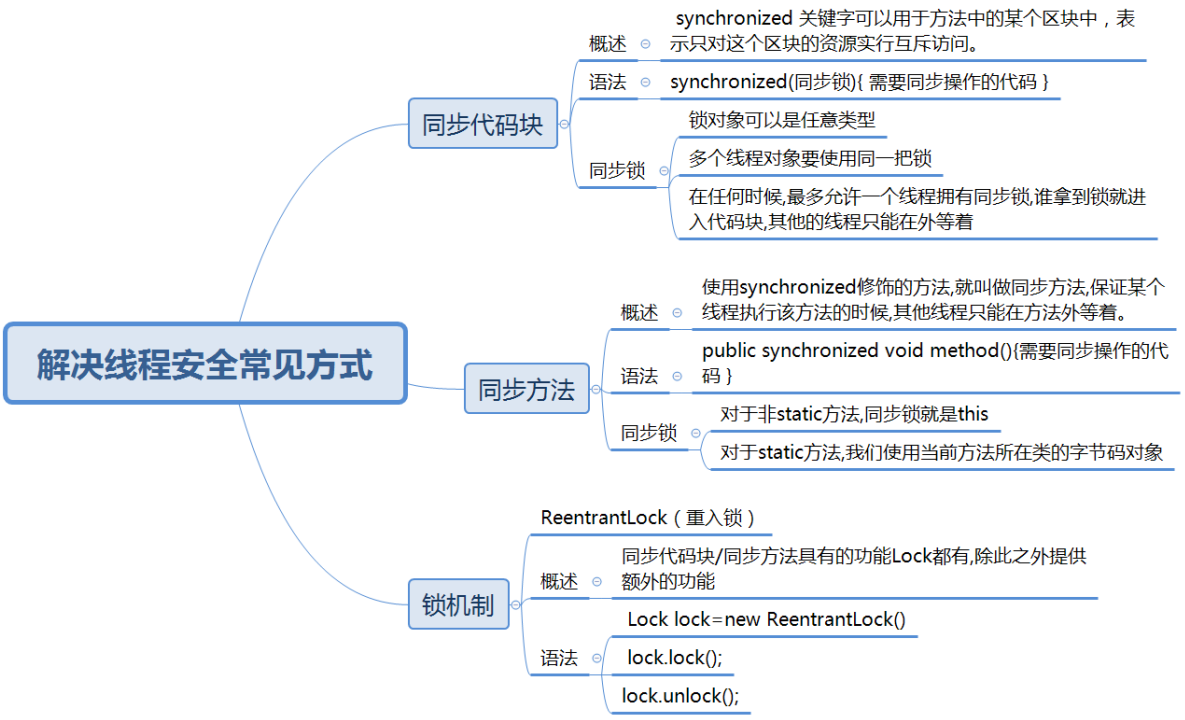
* 当多个线程同时操作一个可共享的资源变量，可能会导致数据不准确（线程不安全）

```
* 案例：电影院售票
  * 实现多个窗口同时卖“哪吒之魔童降世”这场电影票(多个窗口一起卖这100张票)
public class Ticket implements Runnable {
    private int ticket=100;
    @Override
    public void run() {
        while (true){
            if(ticket>0){
                //出票操作
                //使用sleep模拟一下出票时间
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                String name=Thread.currentThread().getName();
                System.out.println(name+"正在卖:"+ticket--);
            }
        }
    }
}
public class Main {
    public static void main(String[] args) {
```

```
        Ticket ticket=new Ticket();
        Thread t1=new Thread(ticket,"窗口1");
        Thread t2=new Thread(ticket,"窗口2");
        Thread t3=new Thread(ticket,"窗口3");
        t1.start();
        t2.start();
        t3.start();
    }
}
* 发现程序出现了两个问题：
  * 相同的票数,比如1这张票被卖了两回。
  * 不存在的票，比如0票与-1票，是不存在的。
* 这种问题，几个窗口(线程)票数不同步了，这种问题称为线程不安全。
```



* 能够掌握解决多线程安全问题常见的方式

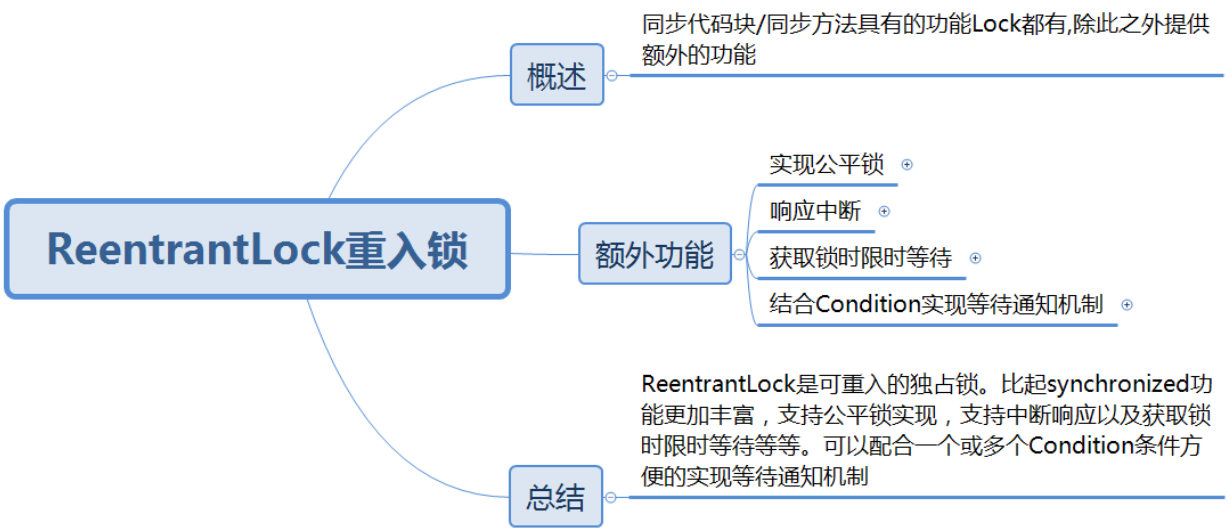



```
* 同步代码块
public class Ticket2 implements Runnable {
    private int ticket=100;
    @Override
    public void run() {
        while (true){
            synchronized (this) {
                    if (ticket > 0) {
                    //出票操作
                    //使用sleep模拟一下出票时间
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    String name = Thread.currentThread().getName();
                    System.out.println(name + "正在卖:" + ticket--);
                }
            }
        }
    }
}
* 同步方法
public class Ticket3 implements Runnable {
```

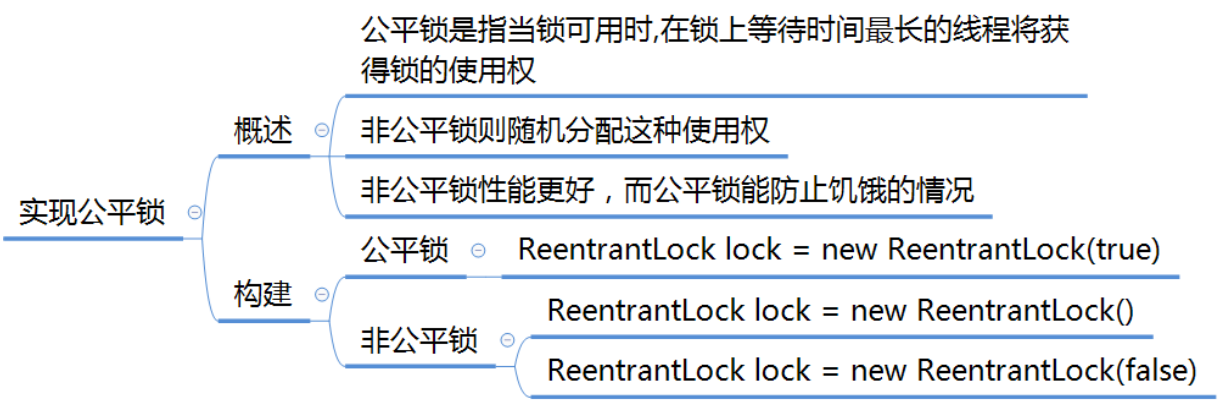
```
    private int ticket=100;
    @Override
    public  void run() {
        while (true) {
            sellTicket();
        }
    }

    public synchronized void sellTicket(){
            if (ticket > 0) {
                //出票操作
                //使用sleep模拟一下出票时间
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                String name = Thread.currentThread().getName();
                System.out.println(name + "正在卖:" + ticket--);
            }
    }
}

* 锁机制
public class Ticket4 implements Runnable {
    private int ticket=100;
    private Lock lock=new ReentrantLock();
    @Override
    public  void run() {
        sellTicket();
    }

    public  void sellTicket(){
        while (true){
            lock.lock();
            if (ticket > 0) {
                //出票操作
                //使用sleep模拟一下出票时间
                try {
                    Thread.sleep(100);
```

```
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                String name = Thread.currentThread().getName();
                System.out.println(name + "正在卖:" + ticket--);
            }
            lock.unlock();
        }
    }
}

```

* 能够掌握ReentrantLock锁的使用



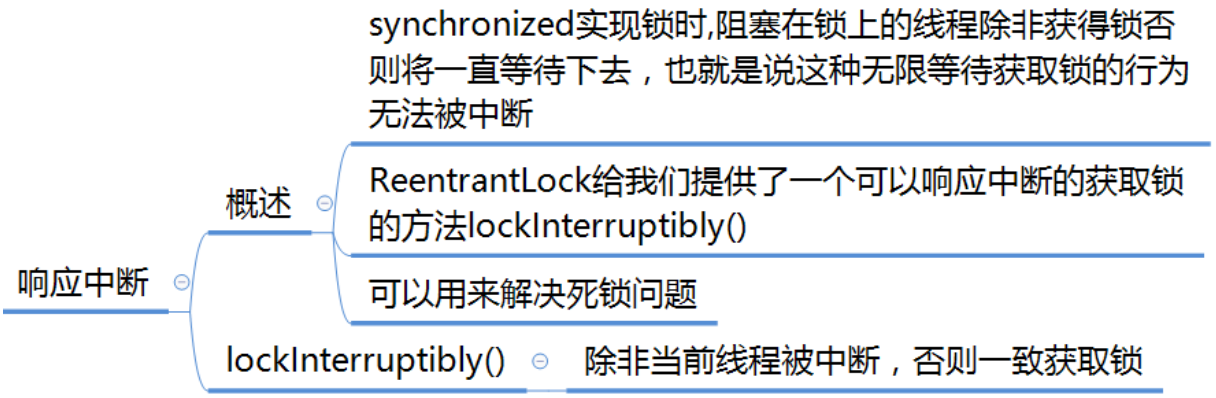
* 实现公平锁

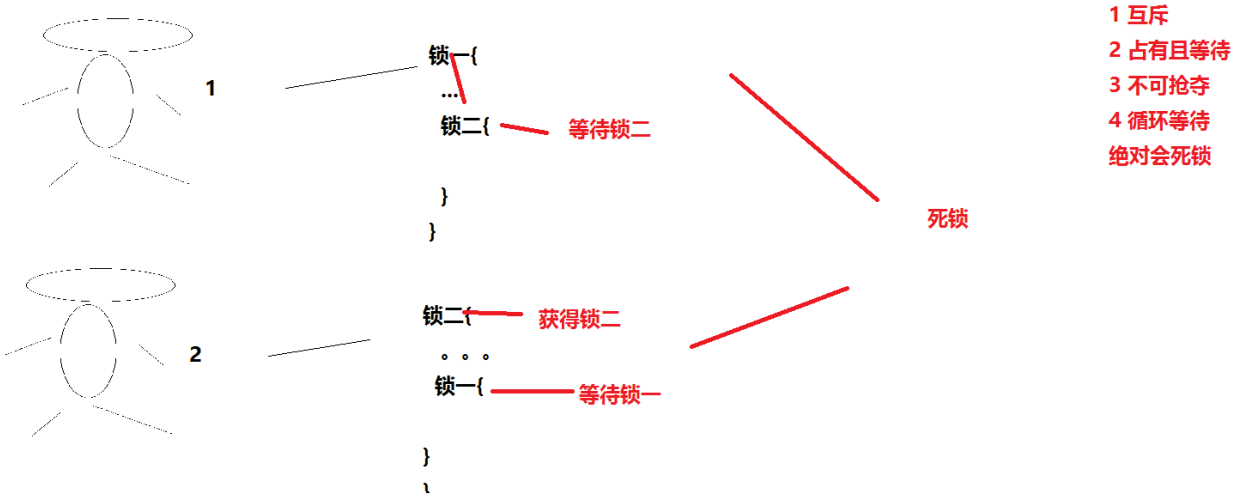


```
* 公平锁实现例子
public class Main {
    private static Lock lock=new ReentrantLock(true);
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            new Thread(new Task(),"t"+i).start();
        }
    }

    static class Task implements Runnable{

        @Override
        public void run() {

            for (int i = 0; i < 2; i++) {
                lock.lock();
                System.out.println(Thread.currentThread().getName()+"被执行了...
                lock.unlock();
            }
        }
    }
}
t0被执行了...
t1被执行了...
t2被执行了...
t3被执行了...
t4被执行了...
t0被执行了...
t1被执行了...
t2被执行了...
t3被执行了...
t4被执行了...
* 非公平锁
private static Lock lock=new ReentrantLock(false);
结果:
t0被执行了...
t3被执行了...
t3被执行了...
t0被执行了...
t4被执行了...
```

```
t4被执行了...
t1被执行了...
t1被执行了...
t2被执行了...
t2被执行了...

* 温馨提醒：
 * 大部分情况下我们使用非公平锁，因为其性能比公平锁好很多。
 * 但是公平锁能够避免线程饥饿，某些情况下也很有用
```

* 响应中断





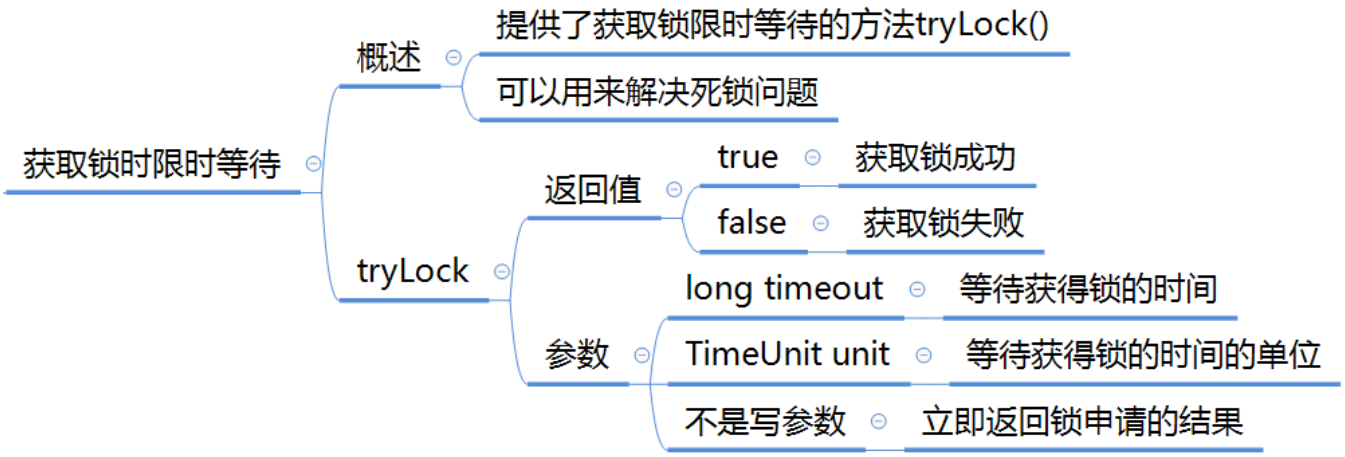

```
* 死锁产生的4个必要条件
```

```
 * 互斥：某种资源一次只允许一个进程访问，即该资源一旦分配给某个进程，
   其他进程就不能再访问，直到该进程访问结束。
 * 占有且等待：一个进程本身占有资源（一种或多种），同时还有资源未得到满足，
   正在等待其他进程释放该资源。
 * 不可抢占：别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来
 * 循环等待：存在一个进程链，使得每个进程都占有下一个进程所需的至少一种资源。
 * 当以上四个条件均满足，必然会造成死锁，发生死锁的进程无法进行下去，
   它们所持有的资源也无法释放。

* synchronized死锁案例
public class Main {
    private static Object lock1=new Object();
    private static Object lock2=new Object();
    public static void main(String[] args) {
        Thread t1=new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (lock1){
                    try {
                        System.out.println(Thread.currentThread().getName()+"获
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    synchronized (lock2){
                        System.out.println(Thread.currentThread().getName()+"获
                    }
                }
            }
        },"t1"){};
        Thread t2=new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (lock2){
                    try {
                        System.out.println(Thread.currentThread().getName()+"获
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
```

```
                    synchronized (lock1){
                        System.out.println(Thread.currentThread().getName()+"获
                    }
                }
            }
        },"t2"){};
        t1.start();
        t2.start();
    }
}
* 结果:
 t2获得了lock2锁
 t1获得了lock1锁
 死锁

* ReentrantLock死锁案例
public class Main {
    private static ReentrantLock lock1=new ReentrantLock();
    private static ReentrantLock lock2=new ReentrantLock();
    public static void main(String[] args) {
        Thread t1=new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    lock1.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName()+"获得了
                    Thread.sleep(500);
                    lock2.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName()+"获得了
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    lock1.unlock();
                    lock2.unlock();
                }
            }
        },"t1"){};
        Thread t2=new Thread(new Runnable() {
            @Override
            public void run() {
```

```
                try {
                    lock2.lockInterruptibly();
                        System.out.println(Thread.currentThread().getName()+"获
                        Thread.sleep(500);
                    lock1.lockInterruptibly();
                        System.out.println(Thread.currentThread().getName()+"获
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    lock1.unlock();
                    lock2.unlock();
                }
            }
        },"t2"){};
        t1.start();
        t2.start();
        // 中断某个线程
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                t1.interrupt();
            }
        }).start();
    }
}
* 可以中断某个线程，解决死锁问题
```

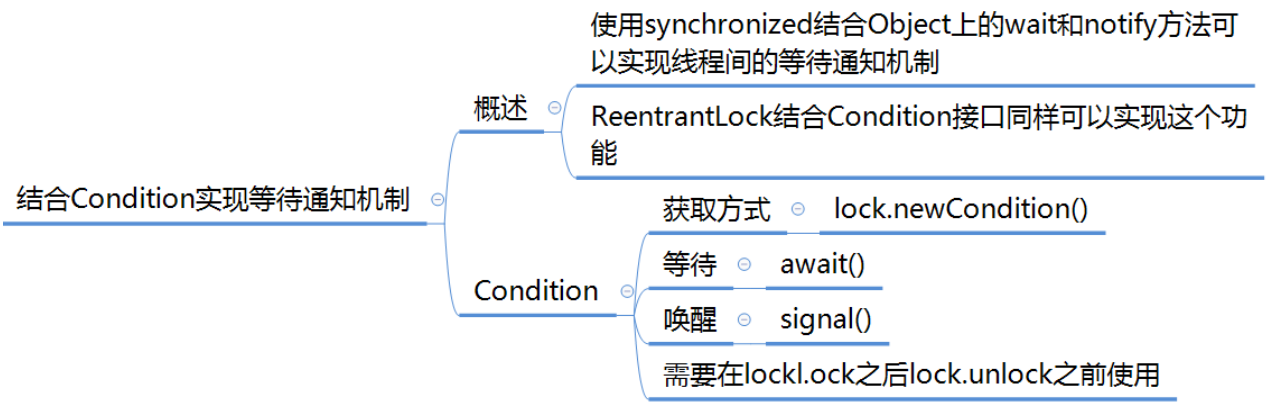
* 获取锁时限时等待




```
* 死锁案例
public class Main {
    private static ReentrantLock lock1=new ReentrantLock();
    private static ReentrantLock lock2=new ReentrantLock();
    public static void main(String[] args) {
        Thread t1=new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    if(lock1.tryLock()){
                        System.out.println(Thread.currentThread().getName()+"获
                        Thread.sleep(500);
                        if(lock2.tryLock()){
                            System.out.println(Thread.currentThread().getName()
                        }
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    if(lock1.isLocked()){
                        lock1.unlock();
                    }
                    if(lock2.isLocked()){
                        lock2.unlock();
                    }
                }
            }
        },"t1"){};
        Thread t2=new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                try {
                    if(lock2.tryLock()){
                        System.out.println(Thread.currentThread().getName()+"获
                        Thread.sleep(500);
                        if(lock1.tryLock()){
                            System.out.println(Thread.currentThread().getName()
                        }
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    if(lock1.isLocked()){
                        lock1.unlock();
                    }
                    if(lock2.isLocked()){
                        lock2.unlock();
                    }
                }
            }
        },"t2"){};
        t1.start();
        t2.start();
    }
}

* 结果
 * 不会出现死锁

```

* 结合Condition实现等待通知机制




```
* ReentrantLock的Condition实现等待通知的例子
public class Main {
    public static boolean isFinish=false;
    private static ReentrantLock lock=new ReentrantLock();
    private static Condition condition = lock.newCondition();
    public static void main(String[] args) {
        // 下载图片的线程
        Thread downThread=new Thread(new Runnable() {
            @Override
            public void run() {
                // 模拟开始下载
                System.out.println("download:开始下载图片...");
                for (int i = 1; i <= 100; i++) {
                    System.out.println("download:已完成" + i + "%");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("download:图片下载完毕");
                isFinish = true;
                lock.lock();
                condition.signal();
                lock.unlock();
                System.out.println("download:开始下载视频...");
                for (int i = 1; i <= 100; i++) {
                    System.out.println("download:已完成" + i + "%");
                    try {
```

```
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
        downThread.start();
        // 显示图片的线程
        Thread showImgThread=new Thread(new Runnable() {
            @Override
            public void run() {
                // 等待图片下载完
                try {
                    lock.lock();
                    condition.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    lock.unlock();
                }
                if(!isFinish){
                    throw new RuntimeException("show:图片还没有下载完");
                }
                System.err.println("show:图片显示完成！");
            }
        });
        showImgThread.start();
    }
}

```

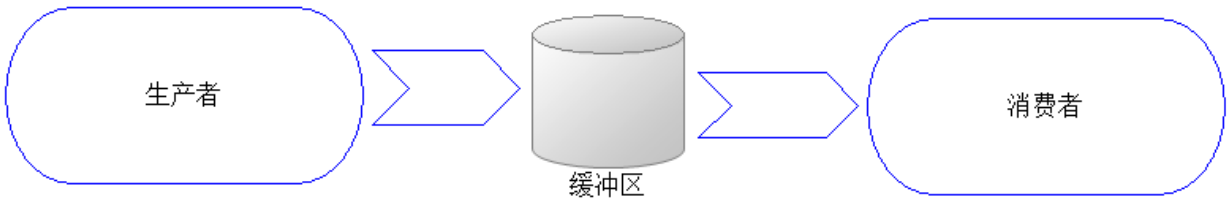
* 能够掌握多线程的生产者和消费者模式

* 生产者和消费者模式的概述

* 在实际的软件开发过程中，经常会碰到如下场景：某个模块负责产生数据，这些数据由另一个模块来负责处理 产生数据的模块，就形象地称为生产者；而处理数据的模块，就称为消费者

* 生产者和消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消

费者彼此并不直接通信，而是通过阻塞队列进行通信，所以生产者生产完数据后不用等待消费者进行处理，而是直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列中获取数据，阻塞队列就相当于一个缓冲区，平衡生产者和消费者的处理能力。




```
* 一条生产者线程，一条消费者线程
 * 通过synchronized+wait+notify实现
 * 生产者
public class Producer1 implements Runnable {
    private List list;
    public Producer1(List list){
        this.list=list;
    }
    @Override
    public void run() {
        // 一直在生产
        Random r=new Random();
        while (true){
            synchronized (list){
                if(list.size()>0){
                    // 假如集合有数据，等待消费者，消费完
                    try {
                        list.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
                // 没有数据，就生产
                list.add(r.nextInt(50));
                System.out.println(Thread.currentThread().getName()+"线程生产了:
                list.notify();// 通知消费者
            }
        }
    }
}
 * 消费者
public class Consumer1 implements Runnable {
    private List list;
    public Consumer1(List list){
        this.list=list;
    }
    @Override
    public void run() {
        while (true){
            synchronized (list) {
                if (list.size() <= 0) {
                    // 假如集合中没有数据，等待生产者生产
                    try {
                        list.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 集合中有数据了，可消费了
                System.err.println(Thread.currentThread().getName()+"线程消费了:
                // 通知生产者，我消费完了
                list.notify();
            }
        }

    }
```

```
}
 * Main
public class Main {
    public static void main(String[] args) {
        List<Integer> list=new ArrayList<Integer>();
        // 生产者
        new Thread(new Producer1(list),"生产者"){}.start();
        // 消费者
        new Thread(new Consumer1(list),"消费者"){}.start();
    }
}

* 多条生产者线程，多条消费者线程
 * 通过synchronized+wait+notifyAll实现
  * 生产者
    * if(list.size()>0) 改成 while(list.size()>0)
    * list.notify(); 改成 list.notifyAll();
  * 消费者
    * if (list.size() <= 0) 改成 while (list.size() <= 0)
    * list.notify(); 改成 list.notifyAll();
  * Main
    public class Main {
    public static void main(String[] args) {
        List<Integer> list=new ArrayList<Integer>();
        for (int i = 0; i < 5; i++) {
            // 生产者
            new Thread(new Producer1(list),"生产者"+i){}.start();
            // 消费者
            new Thread(new Consumer1(list),"消费者"+i){}.start();
        }
    }
}
* 结果:
生产者0线程生产了：40
消费者4线程消费了：40
生产者3线程生产了：22
消费者1线程消费了：22
生产者1线程生产了：31
消费者2线程消费了：31
生产者2线程生产了：19
```

```
消费者0线程消费了：19
生产者2线程生产了：17
消费者2线程消费了：17

* 一条生产者线程，一条消费者线程
 * 通过ReentrantLock实现
 * 生产者
public class Producer1 implements Runnable {
    private List list;
    private Lock mLock;
    private Condition mCondition;
    public Producer1(List list, Lock lock, Condition condition){
        this.list=list;
        this.mLock=lock;
        this.mCondition=condition;
    }
    @Override
    public void run() {
        // 一直在生产
        Random r=new Random();
        while (true){
            mLock.lock();
                if (list.size()>0){
                    // 假如集合有数据，等待消费者，消费完
                    try {
                        mCondition.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 没有数据，就生产
                list.add(r.nextInt(50));
                System.out.println(Thread.currentThread().getName()+"线程生产了:
                mCondition.signal();// 通知所有消费者
            mLock.unlock();
```

```
        }
    }
}
  * 消费者
public class Consumer1 implements Runnable {
    private List list;
    private Lock mLock;
    private Condition mCondition;
    public Consumer1(List list, Lock lock, Condition condition){
        this.list=list;
        this.mLock=lock;
        this.mCondition=condition;
    }
    @Override
    public void run() {
        while (true){
                mLock.lock();
                if (list.size() <= 0) {
                    // 假如集合中没有数据，等待生产者生产
                    try {
                        mCondition.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 集合中有数据了，可消费了
                System.err.println(Thread.currentThread().getName()+"线程消费了:
                // 通知生产者，我消费完了
                mCondition.signal();
                mLock.unlock();
        }

    }
}
 * Main
```

```
public class Main {
    public static void main(String[] args) {
        List<Integer> list=new ArrayList<Integer>();
        ReentrantLock lock=new ReentrantLock();
        Condition condition = lock.newCondition();
        // 生产者
        new Thread(new Producer1(list,lock,condition),"生产者"){}.start();
        // 消费者
        new Thread(new Consumer1(list,lock,condition),"消费者"){}.start();

    }
}
 * 结果
生产者线程生产了：21
消费者线程消费了：21
生产者线程生产了：23
消费者线程消费了：23
生产者线程生产了：24
消费者线程消费了：24
生产者线程生产了：24
消费者线程消费了：24
生产者线程生产了：21
消费者线程消费了：21
生产者线程生产了：0
消费者线程消费了：0
生产者线程生产了：47
消费者线程消费了：47
生产者线程生产了：28
消费者线程消费了：28
生产者线程生产了：31
消费者线程消费了：31
生产者线程生产了：28
消费者线程消费了：28

* 多条生产者线程，多条消费者线程
 * 通过ReentrantLock实现
  * 生产者
    * if(list.size()>0) 改成 while(list.size()>0)
    *  mCondition.signal(); 改成  mCondition.signalAll();
  * 消费者
```

```
227     * if (list.size() <= 0) 改成 while (list.size() <= 0)
228     * mCondition.signal(); 改成  mCondition.signalAll();
229   * Main
230 public class Main {
231     public static void main(String[] args) {
232         List<Integer> list=new ArrayList<Integer>();
233         ReentrantLock lock=new ReentrantLock();
234         Condition condition = lock.newCondition();
235         for (int i = 0; i < 5; i++) {
236             // 生产者
237             new Thread(new Producer1(list,lock,condition),"生产者"+i){}.start()
238             // 消费者
239             new Thread(new Consumer1(list,lock,condition),"消费者"+i){}.start()
240         }
241     }
242 }
```