

## \* 学习目标

### \* 能够掌握多线程之并发工具类

- \* CountDownLatch:计数器，减，countDown

- \* CyclicBarrier:计数器,加

- \* Semaphore:信号灯，限定线程数量

- \* LockSupport : park , unpark

### \* 能够理解Java内存模型

- \* Java Memory Model:JMM

- \* 内存访问跨平台，变量访问规则，共享变量

- \* 原子性，可见性，有序性

- \* 主内存，工作内存

- \* 8操作：主lock-主read-工load-工use-工assign-工store-主write-主unlock

- \* 8个规则：主变量-->工不改不回主->一线lock->lock清工-->unlock-->unlock--readload,storewrite-->assign

### \* 能够理解指令重排序

- \* 优化性能（编译器和CPU）

- \* 单线程：as-if-serial,数据依赖

- \* 多线程下：禁用指令重排序volatile

### \* 能够理解多线程的内存可见性

- \* volatile.加锁，原子类

### \* 能够理解线程安全性CAS

- \* CAS: Compare And Swap

- \* 内存地址V里面值，预期值A，需要更新值B

- \* 确保线程安全：当V里面值和预期值想，就V里面值更新B

- \* atomic: CAS

- \* 能够掌握BlockingQueue相关阻塞队列
  - \* BlockingQueue:阻塞，生产者和消费者模式
    - \* put,take
  - \* ArrayBlockingQueue: 有边界
  - \* ...

---

## \* 回顾

- \* 线程安全的问题：多个线程同时去访问一个共有变量，可能会出现线程安全问题

- \* 卖电影票

## \* 加锁

- \* synchronized(lock)

- \* 代码块，方法

- \* ReentrantLock

- \* lock.lock(),lock.unlock()

## \* ReentrantLock 额外的功能

- \* 实现公平锁，性能较差

ReentrantLock lock=new ReentrantLock(true);

- \* 可中断，解决死锁问题

- \* 死锁：互斥，占有且等待，不可抢夺，循环等待

- \* lockInterruptBy()

- \* tryLock,解决死锁问题

- \* Condition,await,singal,singalAll\

## \* 生成者和消费者模式

生产者--->缓存区--->消费者

- \* 加锁

- \* 等待和唤醒机制

- \* synchronized+wait,notify,notifyAll

- \* ReentrantLock-lock,unlock,condition-await,singal,singalAll

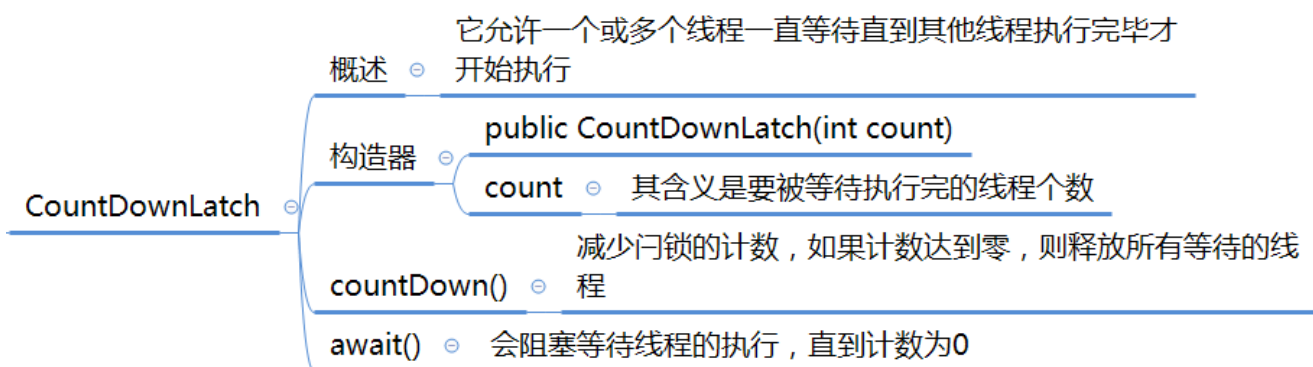
```
1 }
```

\* 能够掌握多线程之并发工具类

\* JDK1.5引入了concurrent并发包，常见的并发工具类



\* CountDownLatch



1 \* 需求：多个线程同时工作，然后其中几个可以随意并发执行，  
2 但有一个线程需要等其他线程工作结束后，才能开始。  
3 例如：开启多个线程分块下载一个电影，每个线程只下载固定的一截，  
4 最后由另外一个线程来拼接所有的分段  
5

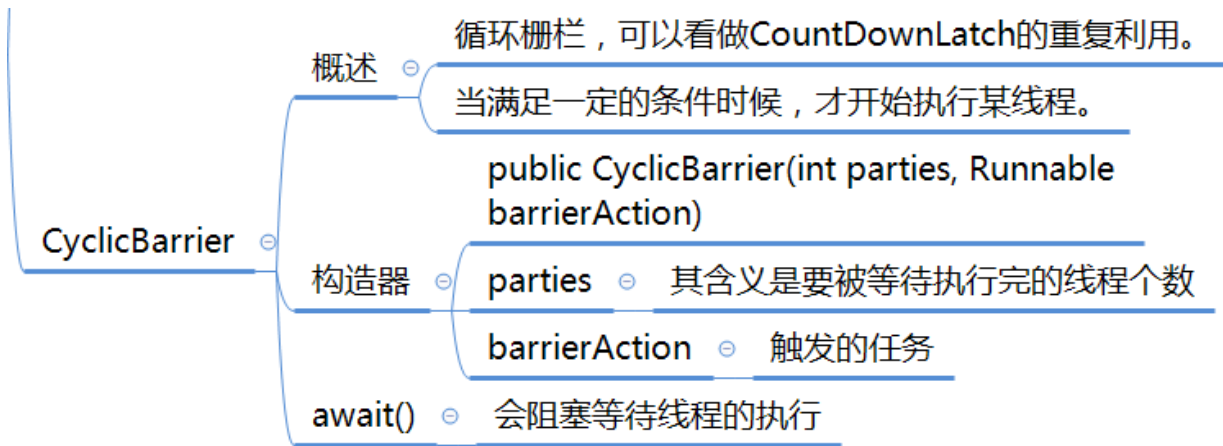
```

6 * 代码
7 public class Main {
8     // 3个要被等待执行完的线程个数
9     private static CountDownLatch latch=new CountDownLatch(3);
10    public static void main(String[] args) {
11        new Thread(new Runnable() {
12            @Override
13            public void run() {
14                System.out.println(Thread.currentThread().getName()+"线程开始下载
15            try {
16                latch.await();
17            } catch (InterruptedException e) {
18                e.printStackTrace();
19            }
20            System.out.println(Thread.currentThread().getName()+"线程下载完
21            System.out.println(Thread.currentThread().getName()+"线程合并文件
22        }
23    }, "download").start();
24    for (int i = 0; i < 3; i++) {
25        new Thread(new Runnable() {
26            @Override
27            public void run() {
28                // 模拟线程下载
29                System.out.println(Thread.currentThread().getName()+"线程开
30            try {
31                Thread.sleep(3000);
32            } catch (InterruptedException e) {
33                e.printStackTrace();
34            }
35            System.out.println(Thread.currentThread().getName()+"线程下
36            latch.countDown();
37        }
38    }, "t"+i).start();
39    }
40 }
41 }
42
43 * 结果
44 download线程开始下载...
45 t0线程开始下载...

```

```
46 t1线程开始下载...
47 t2线程开始下载...
48 t1线程下载完成...
49 t0线程下载完成...
50 t2线程下载完成...
51 download线程下载完成...
52 download线程合并文件
```

## \* CyclicBarrier



```
1 * 需求：开启多个线程分块下载一个电影，每个线程只下载固定的一截，
2 最后所有线程下载完成了，触发方法来拼接所有的分段
3 * 代码
4 public class Main {
5     public static void main(String[] args) {
6         // 3个要被等待执行完的线程个数
7         CyclicBarrier cyclicBarrier=new CyclicBarrier(3, new Runnable() {
8             @Override
9             public void run() {
10                 System.out.println(Thread.currentThread().getName()+"线程合并文件
11             }
12         });
13
14         for (int i = 0; i < 3; i++) {
15             new Thread(new Runnable() {
16                 @Override
17                 public void run() {
18                     // 模拟线程下载
19                     System.out.println(Thread.currentThread().getName()+"线程开
```

```

20         try {
21             Thread.sleep(3000);
22         } catch (InterruptedException e) {
23             e.printStackTrace();
24         }
25         System.out.println(Thread.currentThread().getName()+"线程下载完成...");
26         // 等待所有 parties已经在这个障碍上调用了 await
27         try {
28             cyclicBarrier.await();
29         } catch (InterruptedException e) {
30             e.printStackTrace();
31         } catch (BrokenBarrierException e) {
32             e.printStackTrace();
33         }
34     }
35     }, "t"+i).start();
36 }
37 }
38 }
39 * 结果
40 t0线程开始下载...
41 t1线程开始下载...
42 t2线程开始下载...
43 t0线程下载完成...
44 t2线程下载完成...
45 t1线程下载完成...
46 t2线程合并文件
47

```

\* Semaphore



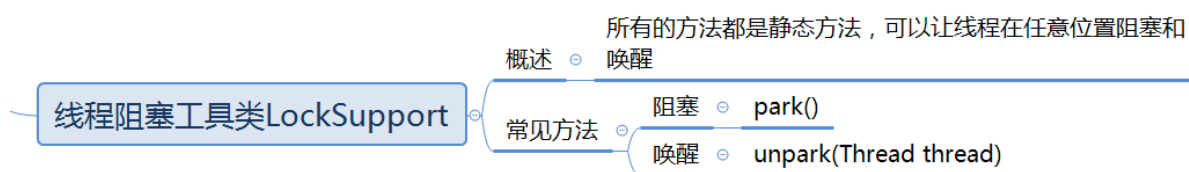
```
1 * 案例:
2 * 停车场上有5个停车位，一个停车位一次只能停一辆车，
3 一旦所有停车位在使用，那么后面的来的车就需要等待，直到有一辆车开出某个停车位
4 后面来的车才能进来
5 * 代码实现
6 public class Main {
7     public static void main(String[] args) {
8         // 5个停车位
9         Semaphore semaphore=new Semaphore(5);
10        // 10辆车近来
11        for (int i = 0; i < 10; i++) {
12            new Thread(new Runnable() {
13                @Override
14                public void run() {
15                    // 获取（许可证）车位
16                    try {
17                        semaphore.acquire();
18                        System.out.println(Thread.currentThread().getName()+"线
19                    } catch (InterruptedException e) {
20                        e.printStackTrace();
21                    }
22                    //模拟停车的时间
```

```

23         try {
24             Thread.sleep(5000);
25         } catch (InterruptedException e) {
26             e.printStackTrace();
27         }
28         // 释放许可证：车开走
29         semaphore.release();
30         System.out.println(Thread.currentThread().getName()+"线程释放");
31     }
32     }, "t"+i).start();
33 }
34 }
35 }
36

```

## \* LockSupport



```

1 * 案例
2 public class Main {
3     public static boolean isFinish=false;
4     public static void main(String[] args) {
5         // 显示图片的线程
6         Thread showImgThread=new Thread(new Runnable() {
7             @Override
8             public void run() {
9                 // 等待图片下载完
10                LockSupport.park();
11                if(!isFinish){
12                    throw new RuntimeException("show: 图片还没有下载完");
13                }
14                System.err.println("show: 图片显示完成! ");
15            }
16        });

```



```

17 // 下载图片的线程
18 Thread downThread=new Thread(new Runnable() {
19     @Override
20     public void run() {
21         // 模拟开始下载
22         System.out.println("download:开始下载图片...");
23         for (int i = 1; i <= 100; i++) {
24             System.out.println("download:已完成" + i + "%");
25             try {
26                 Thread.sleep(100);
27             } catch (InterruptedException e) {
28                 e.printStackTrace();
29             }
30         }
31         System.out.println("download:图片下载完毕");
32         isFinish = true;
33         LockSupport.unpark(showImgThread);
34         System.out.println("download:开始下载视频...");
35         for (int i = 1; i <= 100; i++) {
36             System.out.println("download:已完成" + i + "%");
37             try {
38                 Thread.sleep(100);
39             } catch (InterruptedException e) {
40                 e.printStackTrace();
41             }
42         }
43     }
44 });
45 downThread.start();
46 showImgThread.start();
47
48 }
49 }

```

\* 能够理解Java内存模型

\* Java内存模型概述

## JMM : Java Memory Model

用来屏蔽掉java程序在各种不同的硬件和操作系统对内存的访问的差异，这样就可以实现java程序在各种不同的平台上都能达到内存访问的一致性

### 概述

主要目标是定义程序中变量的访问规则

在虚拟机中将变量存储到主内存或者将变量从主内存取出这样的底层细

#### 共享变量

变量是指实例字段，静态字段，构成数组对象的元素

不包括局部变量和方法参数

Java内存模型围绕着并发过程中如何处理原子性、可见性和有序性这三个特征来设计的

## \* 工作内存与主内存交互

### \* 概述

### 工作内存与主内存交互

主内存 java虚拟机规定所有的变量都必须在主内存中产生

工作内存 java虚拟机中每个线程都有自己的工作内存，该内存是线程私有的

关系 所有变量都存储在主内存中，而每条线程有自己的工作内存，工作内存保存了共享变量的副本

虚拟机规定 线程对主内存变量的修改必须在线程的工作内存中进行，不能直接读写主内存中的变量

不同的线程之间也不能相互访问对方的工作内存

如果线程之间需要传递变量的值，必须通过主内存来作为中介进行传递

工作内存与主内存交互

### \* 交互

### 工作内存与主内存交互

#### 概述

一个变量从主内存传输到工作内存中，

把修改后的变量从工作内存同步回主内存

#### 原子性

#### 有序性

指令重排序

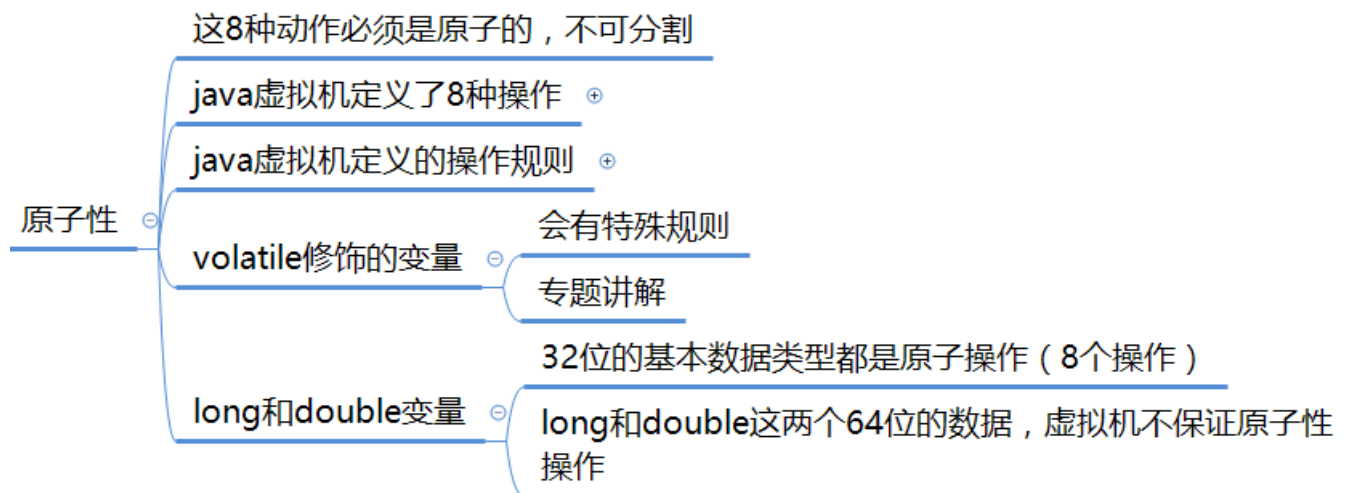
专门讲解

#### 可见性

专门讲解

### \* 原子性

## \* 概述



## \* 操作



## \* 规则

变量只能在主内存中产生

不允许一个线程回写没有修改的变量到主内存

一个变量同一时刻只允许一条线程进行lock操作

对一个变量进行lock操作，那么将会清空工作内存中此变量的值

对一个变量执行unlock之前，必须先把变量同步回主内存中

不允许对未lock的变量进行unlock操作

不允许read和load, store和write操作之一单独出现

不允许一个线程丢弃它最近的assign操作

## java虚拟机定义的操作规则

### \* 画图讲解

概述：核心点

JMM Java Memory Model

定义了规范，JVM具体实现

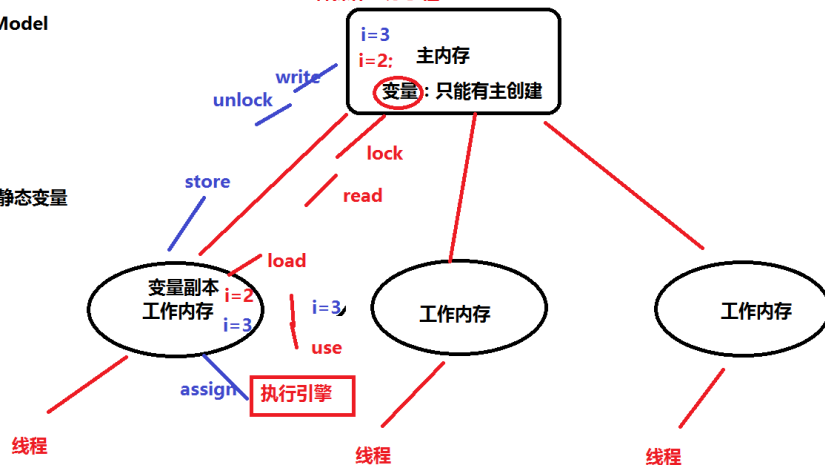
内存访问跨平台

变量规则

变量：共享变量 成员变量，静态变量

原子性，顺序性，可见性

8种操作：原子性



规则

\* 变量1：主生变

\* 变量2：工变没改不回主

\* lock1：一线lock

\* lock2：lock清工

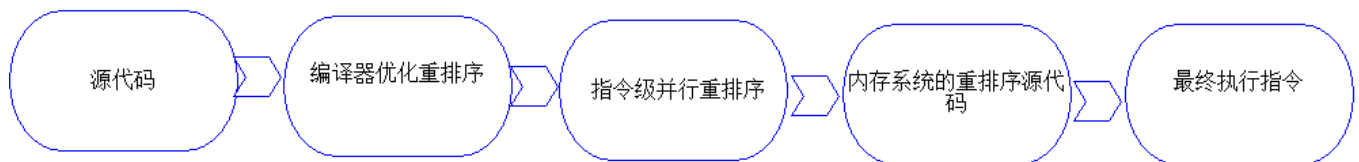
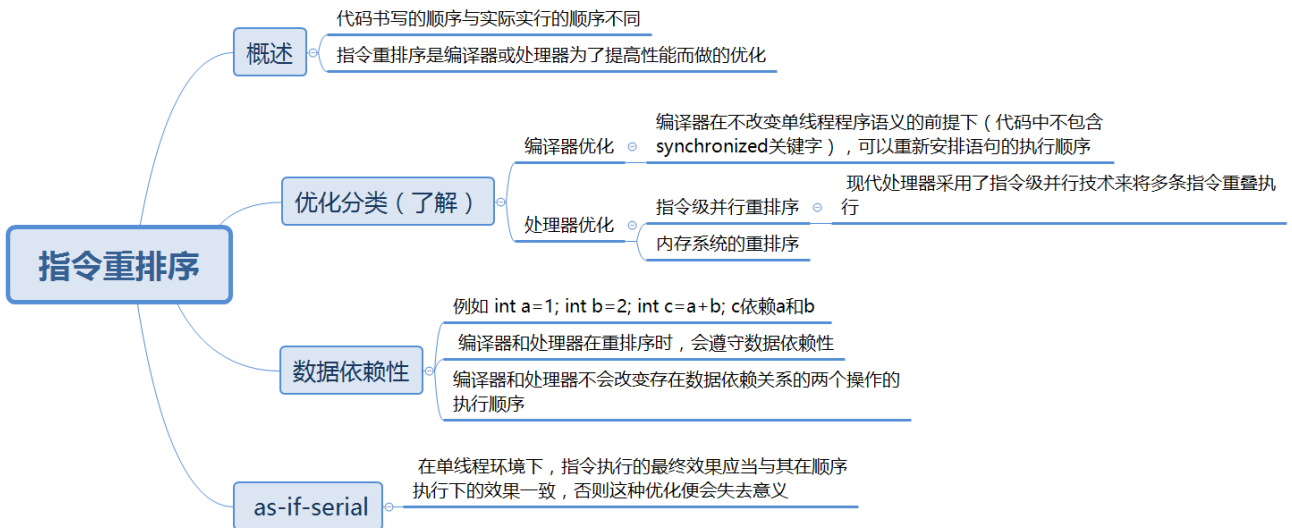
\* unlock1：unlock前同主

\* unlock2：未lock不unlock

\* read和load，store和write

\* assign：不丢assign

### \* 能够理解指令重排序



```
1 * 案例一
2 * 在单线程的情况
3 public static void main(String[] args) {
4     int a=1; // 1
5     int b=2; // 2
6     int c=a+b; // 3
7 }
8 结果:
9 * 按程序顺序执行结果: 1 - 2 - 3 ---> c=3
10 * 重排序后的执行结果: 2 - 1 - 3 ---> c=3
11
12 * 温馨提醒:
13 * 数据依赖性: 例如c依赖a和b
14 * 编译器和处理器在重排序时，会遵守数据依赖性，
15 * 编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序
16 * as-if-serial:
17 * 在单线程环境下，指令执行的最终效果应当与其在顺序执行下的效果一致，
18 否则这种优化便会失去意义
19
20 * 案例二
```

```

21  * 在多线程情况
22  public class Main {
23      public static int a;
24      public static boolean flag;
25      public static void main(String[] args) throws InterruptedException {
26          while (true){
27              Thread t1= new Thread(new Runnable() {
28                  @Override
29                  public void run() {
30                      a=1; // 1
31                      flag=true; // 2
32                  }
33              });
34              Thread t2= new Thread(new Runnable() {
35                  @Override
36                  public void run() {
37                      if (flag){ // 3
38                          if(a==0){ // 4
39                              System.err.println("bingo");
40                          }
41                      }
42                  }
43              });
44              t1.start();
45              t2.start();
46              t1.join();
47              t2.join();
48              a=0;
49              flag=false;
50          }
51      }
52  }
53 }
54
55 * 按程序顺序执行结果：1 - 2 -3
56 * 重排序后的执行结果：2 - 3 - 4
57 * 解决方案
58 * 1 禁止指令重排序
59     public volatile static int a;
60     public volatile static boolean flag;

```

\* 结果：等待一段时间

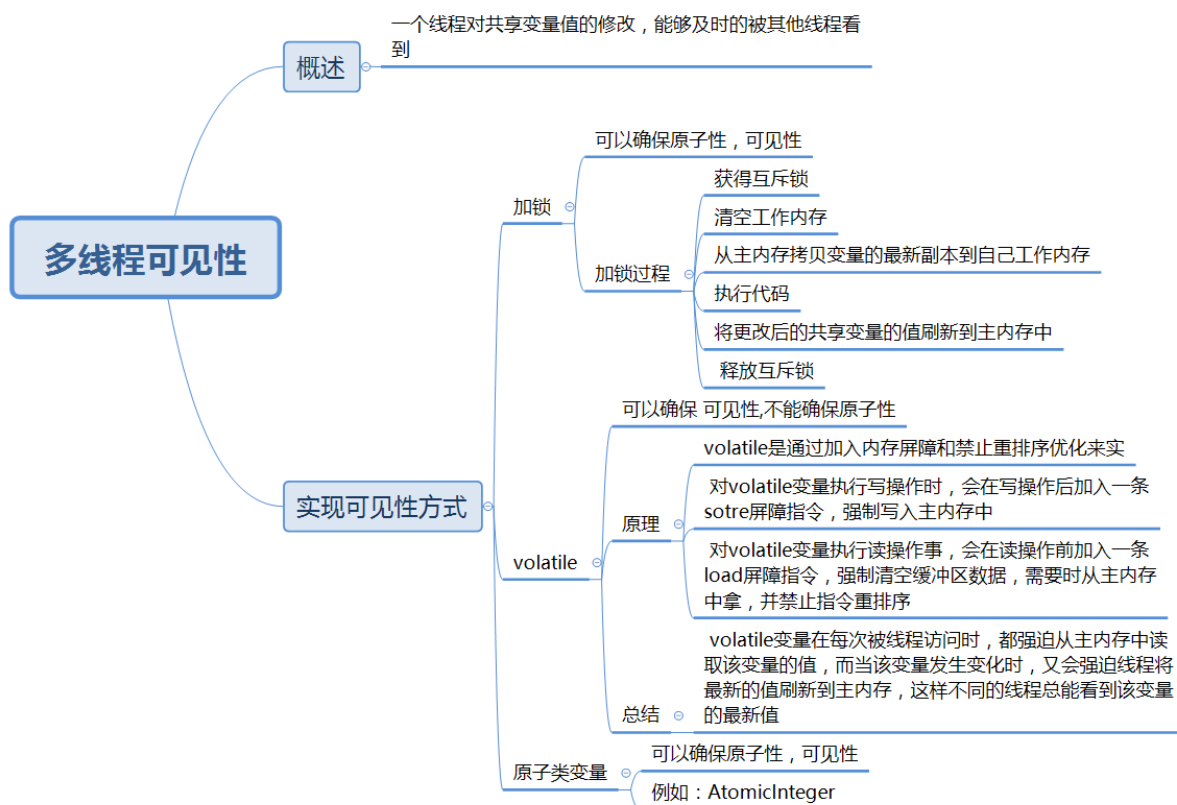
"C:\Program Files\Java\jdk1.8.0\_181\bin\java.exe" ...

bingo

bingo

bingo

\* 能够理解多线程的内存可见性



1 案例

```

2 public class Main {
3     public static void main(String[] args) {
4         Game game=new Game();
5         game.start();
6         try {

```

```

7         Thread.sleep(1000);
8     } catch (InterruptedException e) {
9         e.printStackTrace();
10    }
11    game.command=-1;
12    System.out.println("main over...");
13 }
14 static class Game extends Thread{
15     public int command;
16     @Override
17     public void run() {
18         System.out.println("游戏开始...");
19         while (true){
20             if(command== -1){
21                 break;
22             }
23         }
24         System.out.println("游戏结束");
25     }
26 }
27 }
28
29 * 结果
30 游戏开始...
31 main over...
32 * 发现：在main线程修改变量，无法更新到game线程，线程不可见性
33 * 解决方案
34 * volatile
35     * public volatile int command;
36 * 加锁
37 public class Main3 {
38     public static void main(String[] args) {
39         Game game=new Game();
40         game.start();
41         try {
42             Thread.sleep(1000);
43         } catch (InterruptedException e) {
44             e.printStackTrace();
45         }
46         game.setCommand(-1);

```



```
47     System.out.println("main over...");
48 }
49 static class Game extends Thread{
50     private int command;
51     public synchronized void setCommand(int command){
52         this.command=command;
53     }
54     public synchronized int getCommand(){
55         return command;
56     }
57     @Override
58     public void run() {
59         System.out.println("游戏开始...");
60         while (true){
61             if(getCommand()==-1){
62                 break;
63             }
64         }
65         System.out.println("游戏结束");
66     }
67 }
68 }
69 * 原子性变量
70 public class Main4 {
71     public static void main(String[] args) {
72         Game game=new Game();
73         game.start();
74         try {
75             Thread.sleep(1000);
76         } catch (InterruptedException e) {
77             e.printStackTrace();
78         }
79         game.command.set(-1);
80         System.out.println("main over...");
81     }
82     static class Game extends Thread{
83         public AtomicInteger command;
84         public Game(){
85             command=new AtomicInteger(0);
86         }
```

```

87      @Override
88      public void run() {
89          System.out.println("游戏开始...");
90          while (true){
91              if(command.get()==-1){
92                  break;
93              }
94          }
95          System.out.println("游戏结束");
96      }
97  }
98 }
99

```

## \* 能够理解线程安全性CAS



## \* 画图解释CAS机制

### 银行转账案例

A1 : 100 -- 转 20

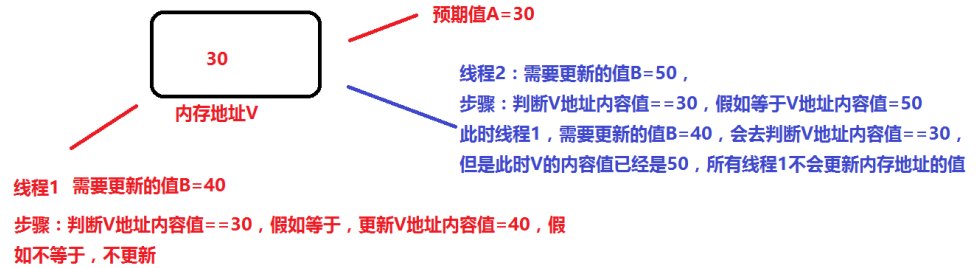
V=100,A=100,B=80

V=70

A2 : 100 -- 转30

V=100,A=100,B=70

V=70



```
1 * 案例
2 * num++ 不是原子性操作
3   num=num+1
4 package com.lg.test7;
5
6 /**
7  * @author xiaozhao
8  */
9 public class Main6 {
10     public static int num;
11     public static void main(String[] args) {
12         for (int i = 0; i < 100; i++) {
13             new Thread(new Runnable() {
14                 @Override
15                 public void run() {
16                     add();
17                 }
18             }).start();
19         }
20     }
21     public static void add(){
22         num++;
23         System.out.println("num = " + num);
24     }
25 }
26 * 结果：可能会出现重复数字
27
28 * 解决方案：原子性
```

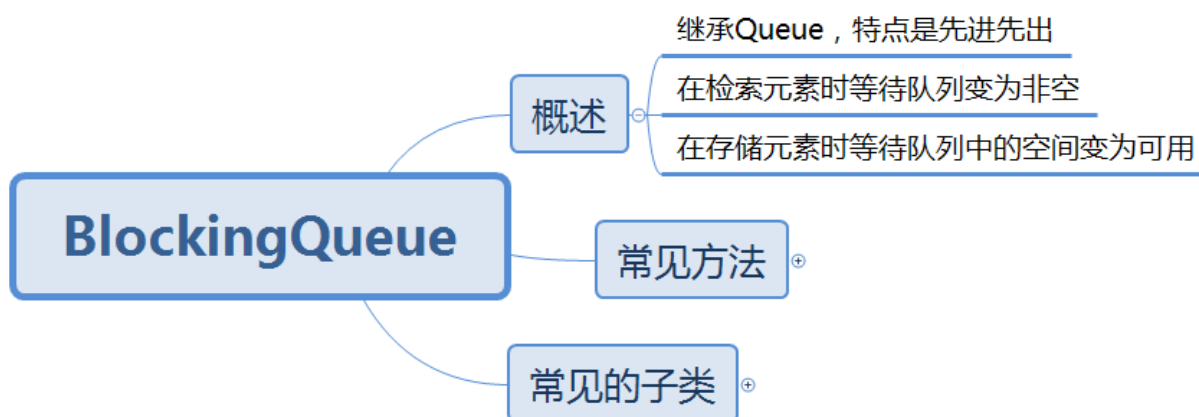
```

29 * 加锁
30 * 用原子类
31 public class Main {
32     public static AtomicInteger num=new AtomicInteger(0);
33     public static void main(String[] args) {
34         for (int i = 0; i < 100; i++) {
35             new Thread(new Runnable() {
36                 @Override
37                 public void run() {
38                     add();
39                 }
40             }).start();
41         }
42     }
43     public static void add(){
44         int value=num.incrementAndGet();
45         System.out.println("value = " + value);
46     }
47 }
48

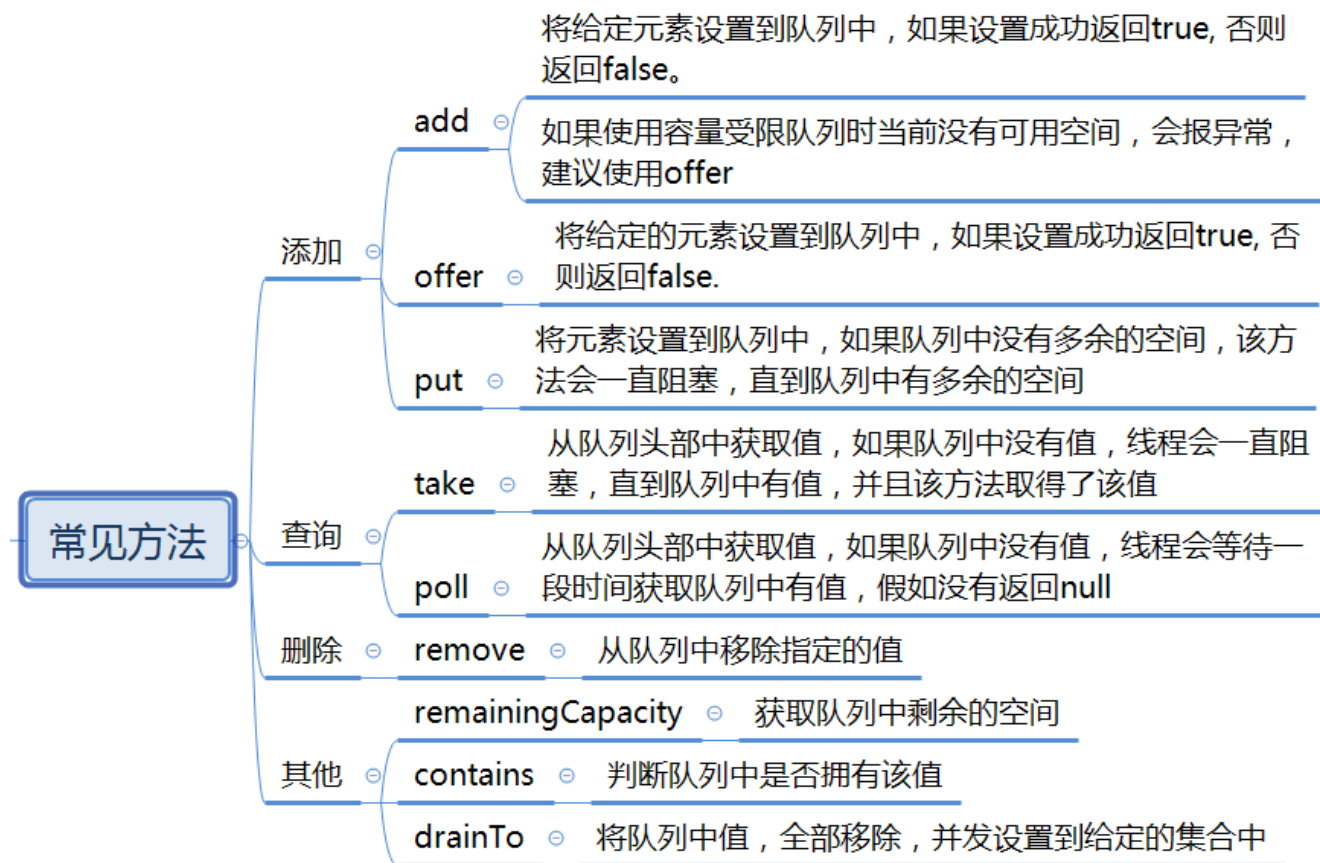
```

\* 能够掌握BlockingQueue相关阻塞队列

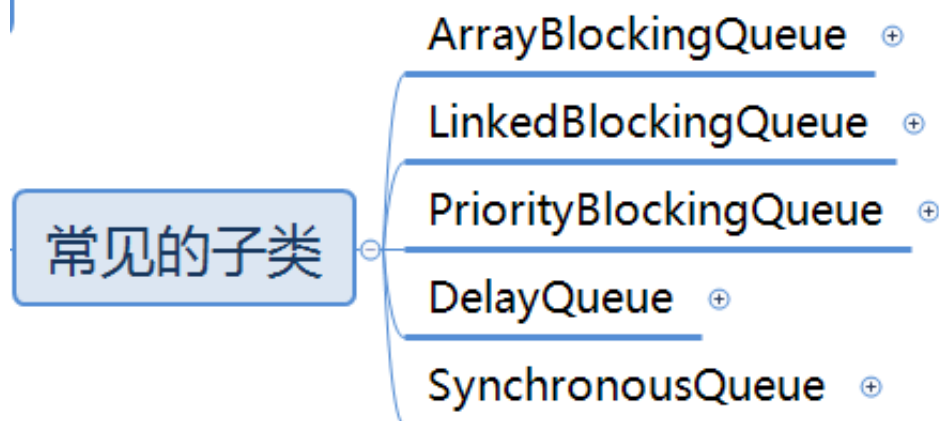
\* 概述



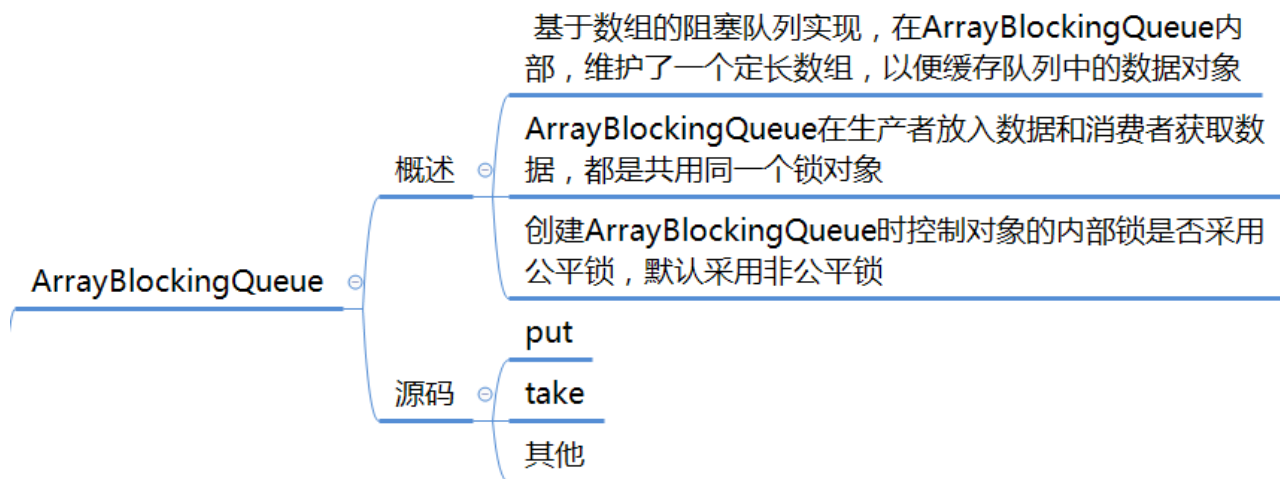
\* 常见方法



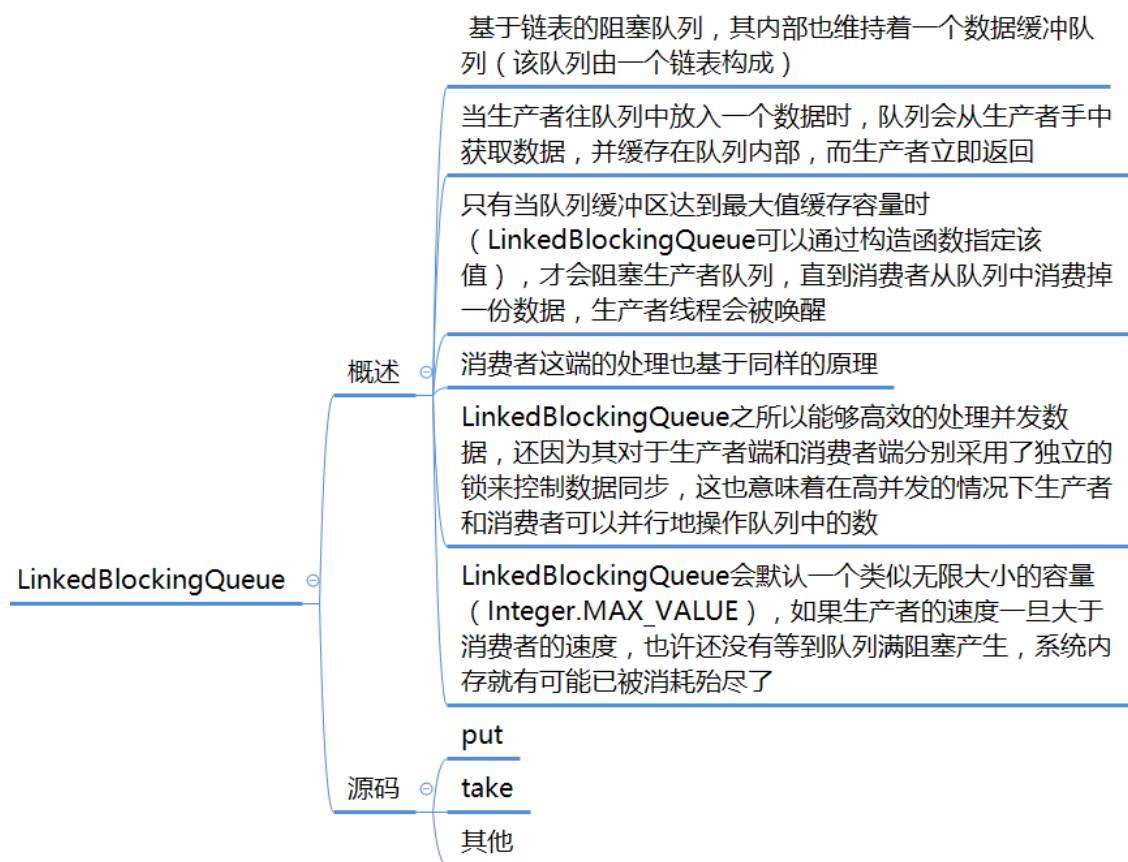
\* 常见的子类



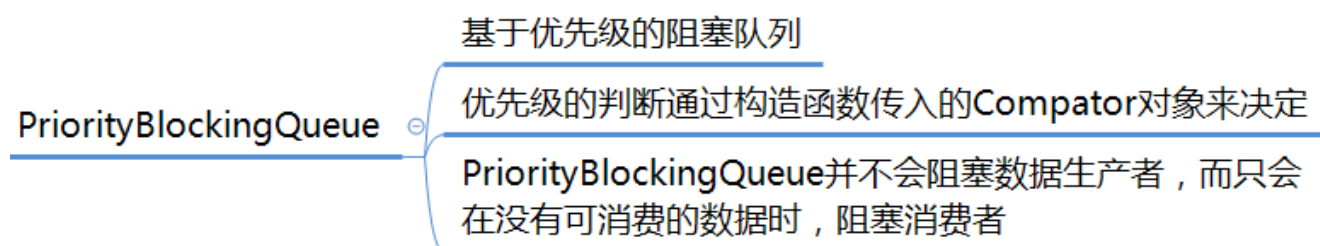
\* ArrayBlockingQueue



### \* LinkedBlockingQueue



### \* PriorityBlockingQueue



### \* DelayQueue

DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素

DelayQueue

DelayQueue是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞

\* SynchronousQueue

SynchronousQueue

无缓冲的等待队列

并不是真正的队列

是一种管理直接在生产者和消费者之间交流的机制

```
1 * 案例: ArrayBlockingQueue
2 * 生产者
3 public class Producer1 implements Runnable {
4     private BlockingQueue mQueue;
5     public Producer1(BlockingQueue queue){
6         this.mQueue=queue;
7     }
8     @Override
9     public void run() {
10         // 一直在生产
11         Random r=new Random();
12         while (true){
13             // 没有数据，就生产
14             while (mQueue.size()<=0) {
15                 try {
16                     Thread.sleep(2000);
17                     int num = r.nextInt(50);
18                     mQueue.put(num);
19                     System.out.println(Thread.currentThread().getName() + "线程
20 } catch (InterruptedException e) {
21     e.printStackTrace();
22 }
23 }
24 }
```

```

25     }
26 }
27 * 消费者
28 public class Consumer1 implements Runnable {
29     private BlockingQueue mQueue;
30     public Consumer1(BlockingQueue queue){
31         this.mQueue=queue;
32     }
33     @Override
34     public void run() {
35         while (true){
36             while (mQueue.size() > 0) {
37                 try {
38                     Thread.sleep(1000);
39                     System.err.println(Thread.currentThread().getName()+"线
40                 } catch (InterruptedException e) {
41                     e.printStackTrace();
42                 }
43             }
44         }
45     }
46 }
47 * Main
48 public class Main {
49     public static void main(String[] args) {
50         BlockingQueue blockingQueue=new ArrayBlockingQueue(16);
51         for (int i = 0; i < 5; i++) {
52             // 生产者
53             new Thread(new Producer1(blockingQueue),"生产者"+i){}.start();
54             // 消费者
55             new Thread(new Consumer1(blockingQueue),"消费者"+i){}.start();
56         }
57     }
58 }
59 }
60
61 * 案例二: LinkedBlockedQueue
62 * ArrayBlockingQueue 替换成LinkedBlockedQueue
63
64 * 案例三: PriorityBlockingQueue

```



```

65 @Data
66 @AllArgsConstructor
67 public class Task implements Comparable<Task> {
68     private int id;
69     private String name;
70     @Override
71     public int compareTo(Task o) {
72         return this.getId()-o.id ;
73     }
74 }
75
76 public class Main {
77     public static void main(String[] args) throws InterruptedException {
78         BlockingQueue<Task> queue=new PriorityBlockingQueue<Task>();
79         Task task1=new Task(1,"xiaohei");
80         Task task3=new Task(3,"xiaobai");
81         Task task2=new Task(2,"xiaoming");
82         queue.put(task1);
83         queue.put(task2);
84         queue.put(task3);
85
86         for (Task task : queue) {
87             System.out.println(task);
88         }
89         List list=new ArrayList();
90         queue.drainTo(list);
91         queue.take();
92         System.out.println("main over");
93     }
94 }
95
96 * 案例网吧上网 :DelayQueue
97 * 实现 Delayed接口
98 @Data
99 public class Customer implements Delayed {
100     private int id;
101     private String name;
102     private long endTime;
103
104     @Override

```

```

105     public long getDelay(TimeUnit unit) {
106         return this.endTime-System.currentTimeMillis();
107     }
108
109     public Customer(int id, String name, long endTime) {
110         this.id = id;
111         this.name = name;
112         // 加上当前时间
113         this.endTime = endTime+System.currentTimeMillis();
114     }
115
116     @Override
117     public int compareTo(Delayed o) {
118         return (int) (this.endTime-((Customer) o).endTime);
119     }
120 }
121
122 * DownTask
123 public class DownTask implements Runnable {
124     private BlockingQueue<Customer> mQueue;
125     public DownTask(BlockingQueue<Customer> blockingQueue){
126         this.mQueue=blockingQueue;
127     }
128     @Override
129     public void run() {
130         while (true) {
131             try {
132                 Customer customer = mQueue.take();
133                 System.out.println("编号: " + customer.getId() + " 姓名: " + cus
134             } catch (InterruptedException e) {
135                 e.printStackTrace();
136             }
137         }
138     }
139
140 * Main
141 public class Main {
142     public static void main(String[] args) {
143         BlockingQueue<Customer> queue=new DelayQueue<Customer>();
144         Customer c1=new Customer(1001,"刘备",1000*5);

```

```

145     Customer c2=new Customer(1002,"关羽",1000*10);
146     Customer c3=new Customer(1003,"张飞",1000*20);
147     queue.add(c1);
148     queue.add(c2);
149     queue.add(c3);
150     new Thread(new DownTask(queue)).start();
151     SimpleDateFormat sdf =new SimpleDateFormat("yyyy-mm-dd hh:mm:ss");
152     while (true){
153         try {
154             Thread.sleep(1000);
155         } catch (InterruptedException e) {
156             e.printStackTrace();
157         }
158         System.out.println(sdf.format(new Date()));
159     }
160 }
161 }
162
163 * 案例: SynchronousQueue
164 * 生产者
165 public class Producer1 implements Runnable {
166     private BlockingQueue mQueue;
167     public Producer1(BlockingQueue queue){
168         this.mQueue=queue;
169     }
170     @Override
171     public void run() {
172         // 一直在生产
173         Random r=new Random();
174         while (true){
175             try {
176                 Thread.sleep(2000);
177                 int num = r.nextInt(50);
178                 mQueue.put(num);
179                 System.out.println(Thread.currentThread().getName() + "线程
180             } catch (InterruptedException e) {
181                 e.printStackTrace();
182             }
183
184         }

```

```
185     }
186 }
187
188 * 消费者
189 public class Consumer1 implements Runnable {
190     private BlockingQueue mQueue;
191     public Consumer1(BlockingQueue queue){
192         this.mQueue=queue;
193     }
194     @Override
195     public void run() {
196         while (true){
197             try {
198                 Thread.sleep(1000);
199                 System.err.println(Thread.currentThread().getName()+"线
200             } catch (InterruptedException e) {
201                 e.printStackTrace();
202             }
203         }
204     }
205 }
206
207 * Main
208 public class Main {
209     public static void main(String[] args) {
210         BlockingQueue blockingQueue=new SynchronousQueue();
211         // 生产者
212         new Thread(new Producer1(blockingQueue),"生产者").start();
213         // 消费者
214         new Thread(new Consumer1(blockingQueue),"消费者").start();
215     }
216 }
217
```

