1、单例模式

单例模式能保证某个类在程序中只存在唯一一份实例, 而不会创建出多个实例.

1. 饿汉模式

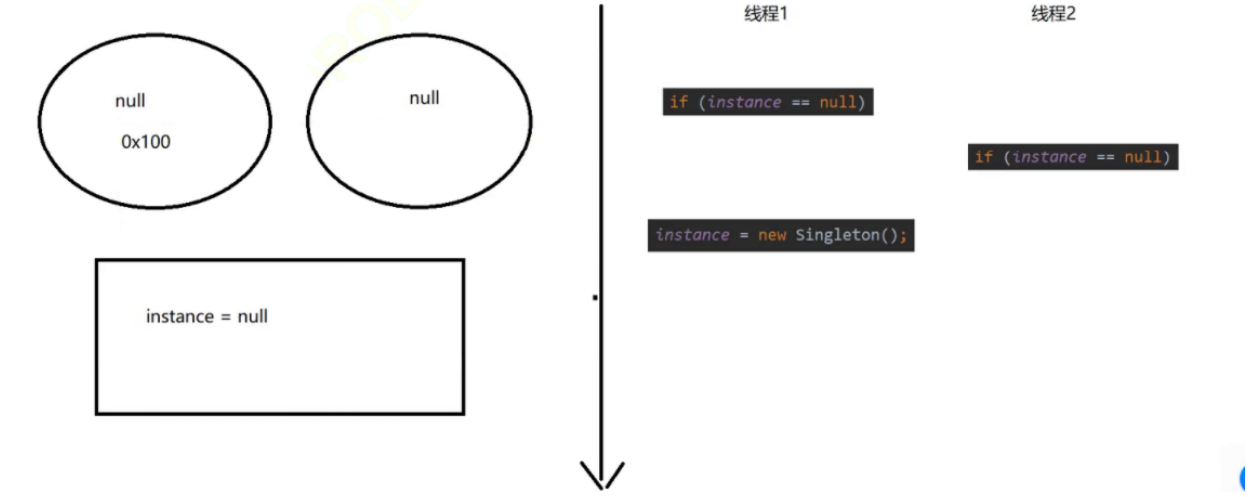
static 在类加载阶段就把实例创建出来

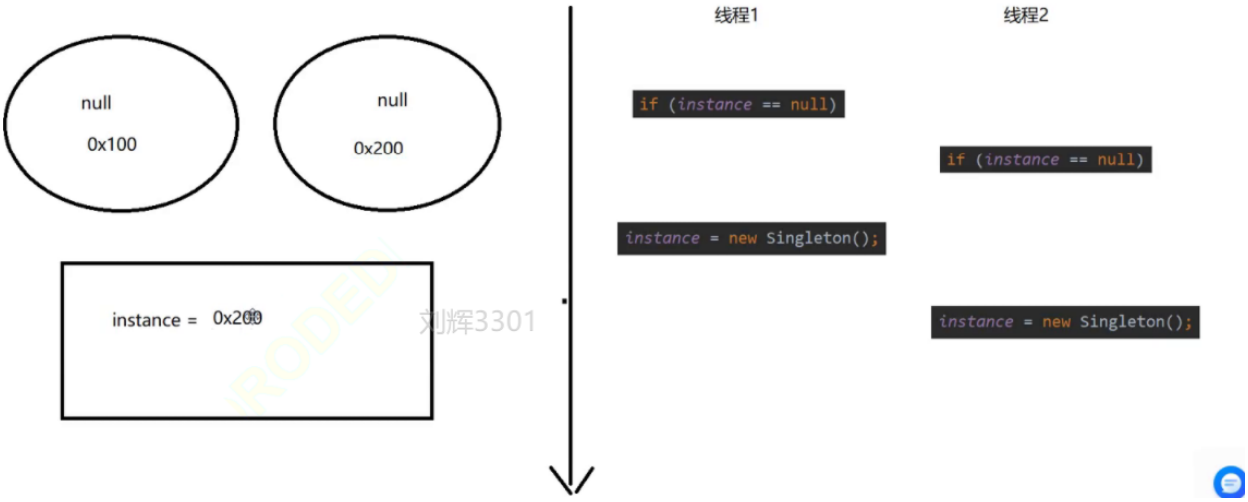
```
static class Singlrton{
 // 把构造方法设为 private ,防止在类外面调用构造方法,
 // 也就禁止了调用者在其他地方创建实例的机会
 private Singleton{
 }
 private static Singleton instance = new Singleton();
 public static Singleton getInstance(){
 return instance;
 }
}

// 此处认为是线程安全的! 当前是多个线程读, 不涉及到修改!
```

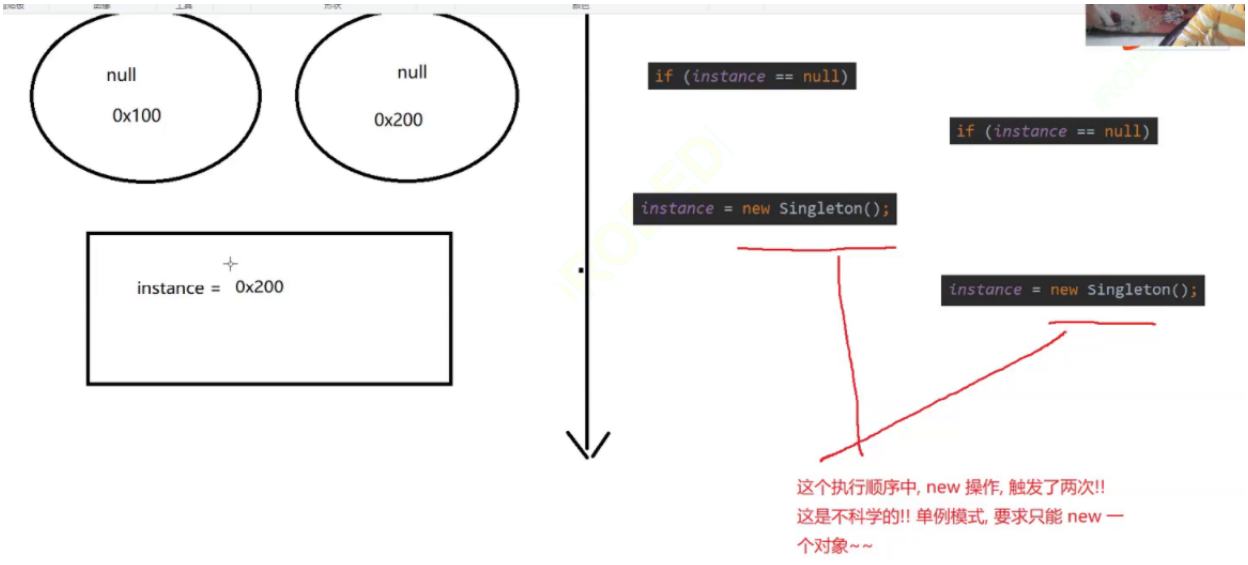
2. 懒汉模式—单线程版

通过 getInstance 方法类获取到实例。类加载的时候不创建实例. 第一次使用的时候才创建实例.

```
static class Singlrton{
 // 把构造方法设为 private ,防止在类外面调用构造方法,
 // 也就禁止了调用者在其他地方创建实例的机会
 private Singleton{
 }
 private static Singleton instance = null;
 public static Singleton getInstance(){
 if(instance == null){
 instance = new Singleton();
  }
  return instance;
  }
}
```







3. 懒汉模式—多线程版

上面的懒汉模式的实现是线程不安全的.

- 线程安全问题发生在首次创建实例时. 如果在多个线程中同时调用 getInstance 方法, 就可能导致创建出多个实例.
- 一旦实例已经创建好了, 后面再多线程环境调用 getInstance 就不再有线程安全问题了(不再修改 instance 了)

典型的错误写法:

```
if(instance == null){
 synchronized (Singleton.class){
 instance = new Singleton();
 }
}
```

这是一种典型的错误写法,此处的线程不安全,主要是因为 if 操作和 = 操作不是原子的,要想解决这里的线程安全问题,主要是要把这两个操作变成原子的.需要使用synchronized 把他们给包上.

科学的写法:

```
public static Singleton getInstance() {
 synchronized (Singleton.class){
 if(instance == null){
 instance = new Singleton();
 }
 }
 return instance;
}
```

注意:如果这里这么加锁,确实解决了线程安全问题,但是引入的新的问题! getInstance是首次调用的时候才涉及到线程安全问题,后续调用的时候就不涉及了 (后续没有修改了!)按照上面加锁的写法,不仅仅是首次调用,包括后续的调用,也会涉及到加锁操作.后续本来没有线程安全问题,不需要加锁,再尝试加锁,就是多此一举了.加锁操作本来就是一个开销比较大的操作!

改进:

思路:首次调用的时候就加锁,后续调用的时候就不加锁。

```
class Singleton {
 private static volatile Singleton instance = null;
 private Singleton() {}
```

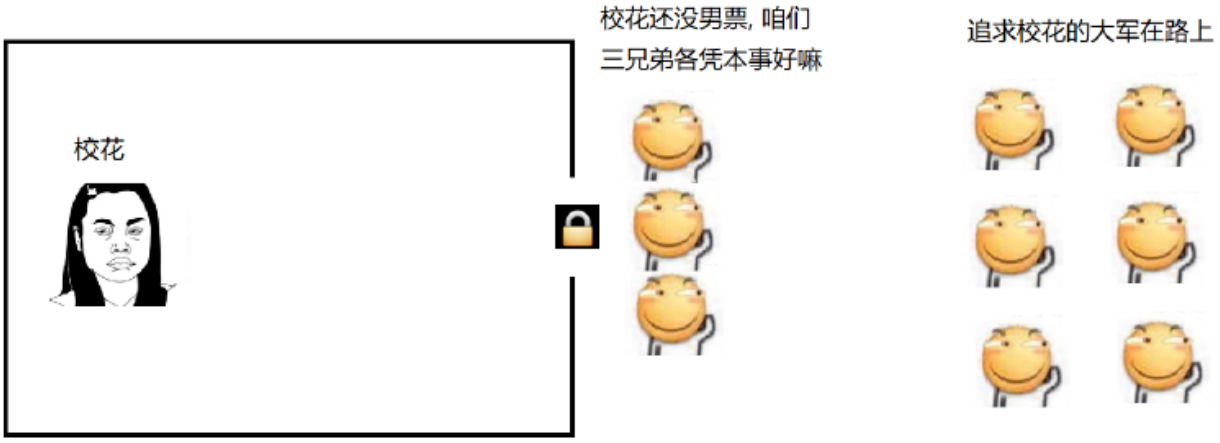
```
4   public static Singleton getInstance() {
5   if(instance == null){ // 区分当前是首次调用还是后续调用,决定是否加
锁
6   synchronized (Singleton.class){
7   if(instance == null){// 第一批调用的这些线程,其中谁是能获取到锁的幸
运儿
8   instance = new Singleton();
9   }
10   }
11   }
12   return instance;
13   }
14  }
```

上述代码中: if(instance == null) 这个条件在 synchronized 内外各出现了一次,如果没有 synchronized ,这样的代码是毫无意义的,但是有了 synchronized 之后,这个代码就很关键

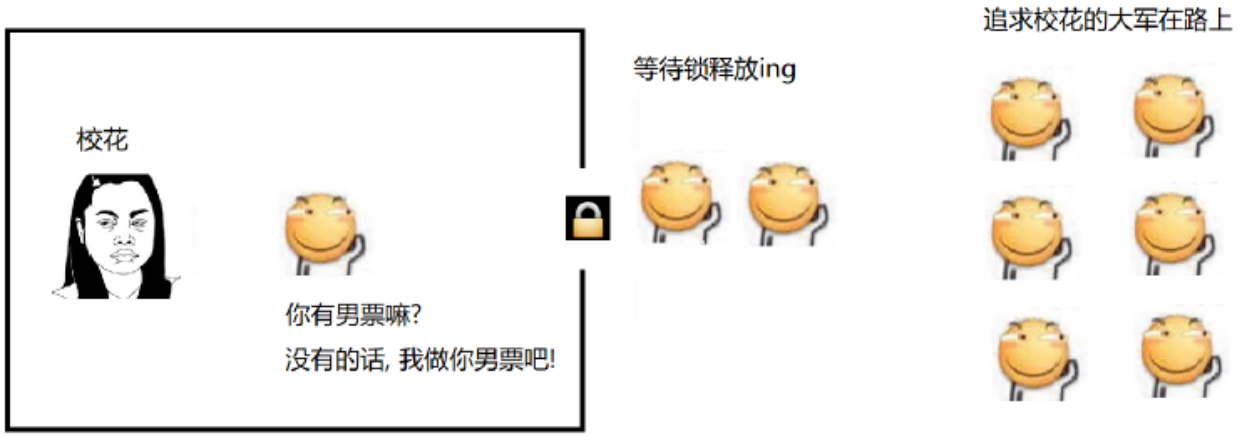
这里面还涉及到一个很重要的问题!当首批线程通过第一层if,进入到锁阶段,并创建好锁对象之后,这个时候就相当于把内存中 instance 的值修改成非 null 了,后续批次的线程,通过第一层if的时候,也要判定 instance 的值,但是这个判定不一定是从内存中读取的数据,也有可能是从寄存器中读取的数据。

理解双重 if 判定 / volatile:

1) 有三个线程, 开始执行 getInstance , 通过外层的 if (instance == null) 知道了实例还没有创建的消息. 于是开始竞争同一把锁.

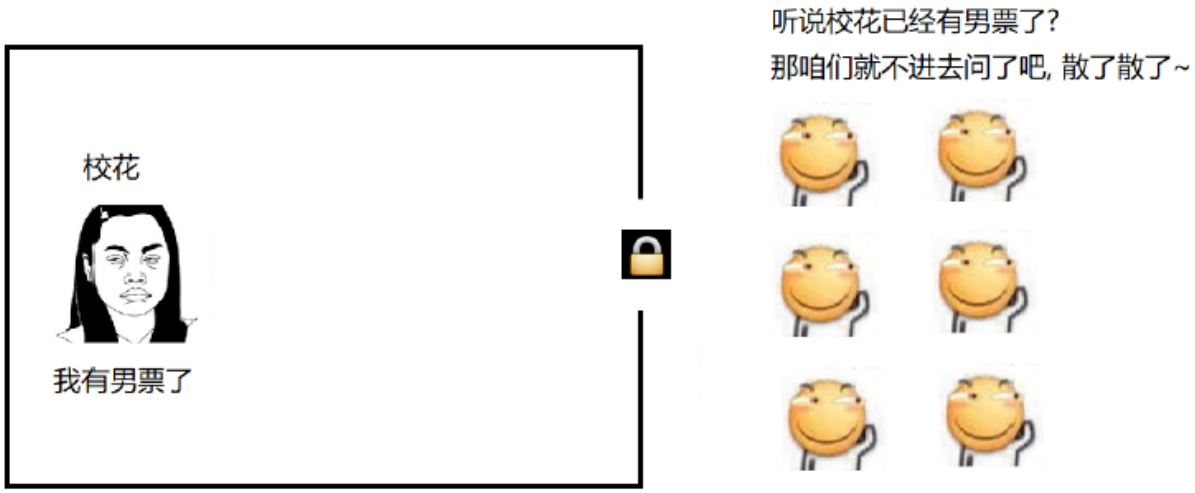


2) 其中线程1 率先获取到锁, 此时线程1 通过里层的 if (instance == null) 进一步确认实例是否已经创建. 如果没创建, 就把这个实例创建出来.



3) 当线程1 释放锁之后, 线程2 和 线程3 也拿到锁, 也通过里层的 if (instance == null) 来确认实例是否已经创建, 发现实例已经创建出来了, 就不再创建了.

4) 后续的线程, 不必加锁, 直接就通过外层 if (instance == null) 就知道实例已经创建了, 从而不再尝试获取锁了. 降低了开销



2、阻塞式队列

1. 阻塞队列是什么

阻塞队列是一种特殊的队列。也遵守"先进先出"的原则。

阻塞队列能是一种线程安全的数据结构, 并且具有以下特性:

- 当队列满的时候, 继续入队列就会阻塞, 直到有其他线程从队列中取走元素.
- 当队列空的时候, 继续出队列也会阻塞, 直到有其他线程往队列中插入元素.

2. 标准库中的阻塞队列

在 Java 标准库中内置了阻塞队列. 如果我们需要在一些程序中使用阻塞队列, 直接使用标准库中的即可.

i. BlockingQueue 是一个接口. 真正实现的类是 LinkedBlockingQueue.

ii. put 方法用于阻塞式的入队列, take 用于阻塞式的出队列.

iii. BlockingQueue 也有 offer, poll, peek 等方法, 但是这些方法不带有阻塞特性

```
1  public class ThreadDemo16 {
2   public static void main(String[] args) throws InterruptedExcept
 ion {
3   // 此处可以看到 LinkBlockingQueue 内部是基于链表来实现的
4   BlockingDeque<String> queue = new LinkedBlockingDeque<>();
5   // put 带有阻塞功能, 但是 offer 不带有,使用阻塞队列一般都是使用 put
6   queue.put("hello");
7   String elem = queue.take();
8   System.out.println(elem); // hello
9   // 代码执行到这里就不走了,一直等到有其它线程给
10   // 这个队列加入新的元素为止
11   elem = queue.take();
12   System.out.println(elem);
13   }
14  }
```

3. 生产者消费者模型

生产者消费者模式就是通过一个容器来解决生产者和消费者的强耦合问题。

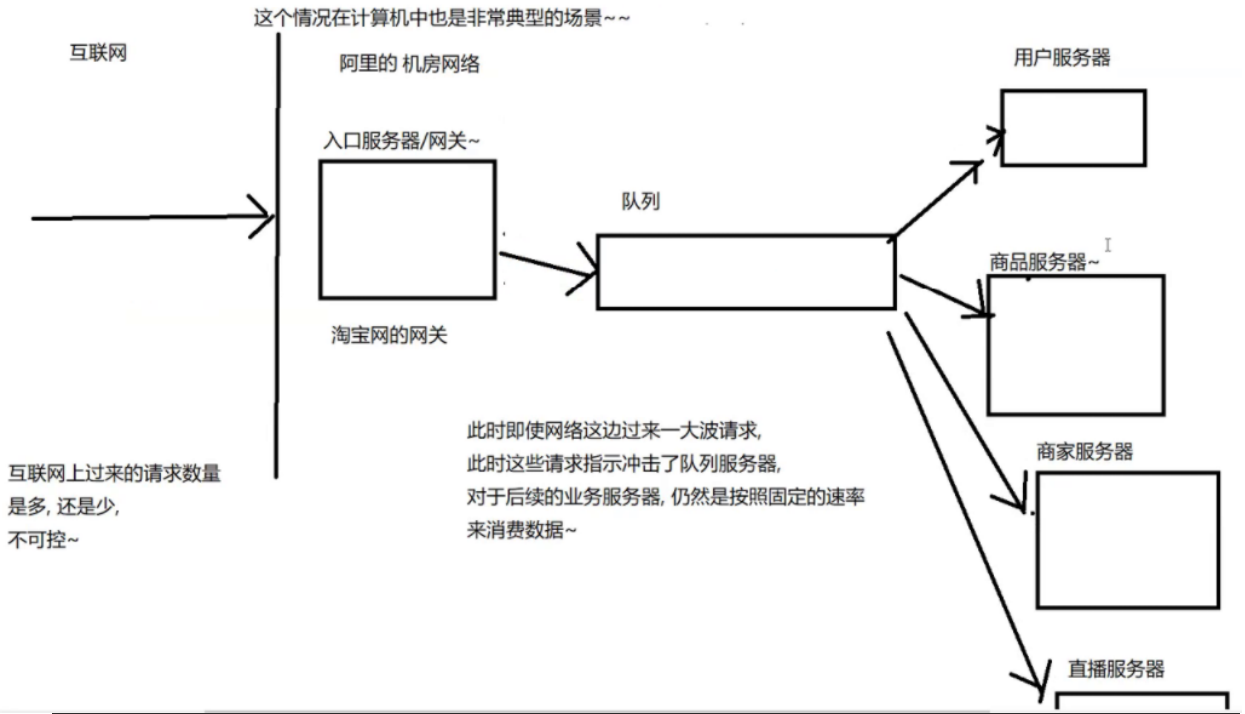
生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取.

生产者消费者模型是服务器开发中非常常用及非常有用的一种编程手段。

最大的用途：

1.解耦合

2.削峰填谷




```java
public class ThreadExer1 {
 public static void main(String[] args) {
 // 先创建一个 BlockingQueue 队列作为交易场所
 BlockingQueue<Integer> queue = new LinkedBlockingQueue<>();

 // 创建一个线程作为消费者
 Thread customer = new Thread(){
 @Override
 public void run() {
  while(true){
  // 获取队首元素
  try {
  Integer value = queue.take();
```

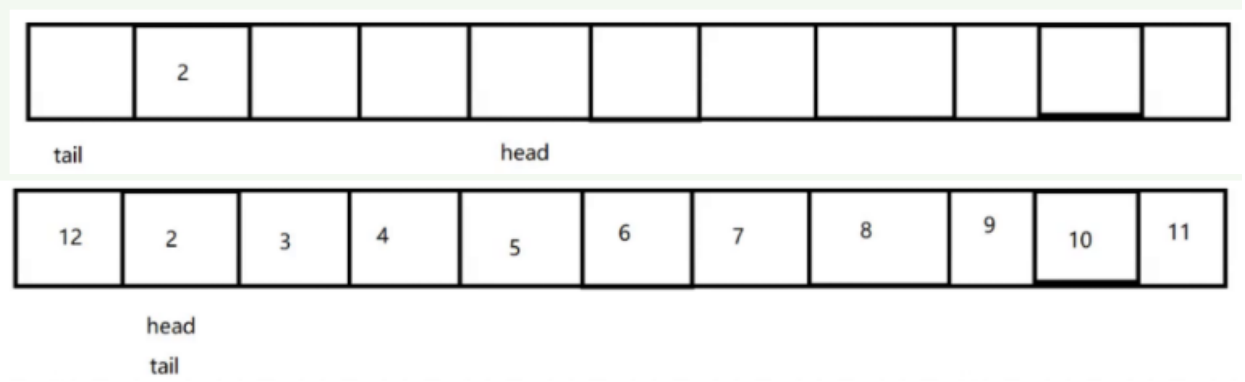
```
14  System.out.println("消费的模型: " + value);
15  } catch (InterruptedException e) {
16  e.printStackTrace();
17  }
18  }
19  }
20  };
21  customer.start();
22
23  // 创建一个线程作为生产者
24  Thread producer = new Thread(){
25  @Override
26  public void run() {
27  for(int i = 0; i < 10; i ++){
28  System.out.println("生产了元素: " + i);
29  try {
30  queue.put(i);
31  Thread.sleep(1000);
32  } catch (InterruptedException e) {
33  e.printStackTrace();
34  }
35  }
36  }
37  };
38  producer.start();
39
40  try {
41  customer.join();
42  producer.join();
43  } catch (InterruptedException e) {
44  e.printStackTrace();
45  }
46  }
47 }
```

4. 阻塞队列的实现

借助BlockingQueue 写一个简单的生产者消费者模型的代码

1. 通过 "循环队列" 的方式来实现.

数组实现队列,其实就是"循环队列".



[head,tail) 表示队列中的有效元素区间。

入队列，就是把新的元素放到 tail 上,并且 tail ++;

出队列， 就是把队首元素取出来,并且 head ++;

最重要的问题,如何区分队列是队列空?

如果由于一直插入元素,导致 head 和 tail 重合,此时队列就满了。

如果由于一直删除元素,导致 head 和 tail 重合,此时队列就空了。

区分的方案:

1. 浪费一个格子

head 和 tail 重合表示为空，tail + 1 和 head 重合；表示为满。

2. 专门给定一个变量 size，记录当前元素的个数。

入队列，就 size ++；出队列，就 size --；size 为0的时候就是空，size 为数组最大长度，就是满。

2. 使用 synchronized 进行加锁控制.

3. put 插入元素的时候, 判定如果队列满了, 就进行 wait. (注意, 要在循环中进行 wait. 被唤醒时不一定队列就不满了, 因为同时可能是唤醒了多个线程).

4. take 取出元素的时候, 判定如果队列为空, 就进行 wait. (也是循环 wait)

```
1 public class ThreadExer2 {
2  static class BlockingQueue{
```

```
// 1000相当于队列的最大容量，不考虑扩容
private int[] items = new int[1000];
private int head = 0;
private int tail = 0;
private int size = 0;
Object locker = new Object();

// put 用来入队列
public void put(int item) throws InterruptedException {
synchronized (locker) {
// 此处最好使用 while.
// 否则 notifyAll 的时候，该线程从 wait 中被唤醒，
// 但是紧接着并未抢占到锁. 当锁被抢占的时候，可能又已经队列满了
// 就只能继续等待
while(size == items.length){
// 队列已满，对于阻塞队列来说就要阻塞，进行等待
locker.wait();
}

items[tail] = item;
tail ++;

// 当队列到达末尾，就要回到起始位置
if(tail >= items.length){
tail = 0;
}

size ++;
// 此处的 notify 用来唤醒 take 中的 wait
locker.notify();
}
}

// take 用来出队列
public int take() throws InterruptedException {
```

```
38  int ret = 0;
39  synchronized (locker) {
40  while(size == 0){
41  // 对于阻塞队列来说,如果队列为空,再尝试取元素,就要阻塞
42  locker.wait();
43  }
44
45  ret = items[head];
46  head ++;
47
48  if(head >= items.length){
49  head = 0;
50  }
51  size --;
52
53  // 此处的 notify 用来唤醒 put 中的 wait
54  locker.notify();
55  }
56  return ret;
57  }
58  }
59
60  public static void main(String[] args) {
61  BlockingQueue queue = new BlockingQueue();
62
63  // 创建一个线程作为消费者
64  Thread customer = new Thread(){
65  @Override
66  public void run() {
67  for(int i = 1;i <= 10; i ++){
68
69  try {
70  int value = queue.take();
71  System.out.println("消费的模型: " + value);
72  } catch (InterruptedException e) {
```

```
73  e.printStackTrace();
74  }
75  }
76  }
77  };
78  customer.start();
79
80  // 创建一个线程作为生产者
81  Thread producer = new Thread(){
82
83  @Override
84  public void run() {
85  for(int i = 1;i <= 10; i ++){
86  System.out.println("生产的模型: " + i );
87  try {
88  queue.put(i);
89  Thread.sleep(1000);
90  } catch (InterruptedException e) {
91  e.printStackTrace();
92  }
93  }
94  }
95  };
96  producer.start();
97
98  try {
99  customer.join();
100  producer.join();
101  } catch (InterruptedException e) {
102  e.printStackTrace();
103  }
104  }
105 }
```

put 和 take 都有可能出现阻塞的情况（wait），由于这两个代码中的条件是对立的，因此这两边的wait 不会同时被触发。

put来唤醒 take 的阻塞，put 操作就破坏了 take 的阻塞条件，

take 来唤醒 put 的阻塞，take 操作也破坏了 put 的阻塞条件。

3、定时器

1. 定时器是什么?

定时器也是软件开发中的一个重要组件. 类似于一个 "闹钟". 达到一个设定的时间之后, 就执行某个指定好的代码.

应用场景:

1.网络编程:

浏览器访问某个网站,网卡了,浏览器就会转圈圈(阻塞等待),这个等待不是无限等待,等到一定时间之后,就显示访问超时。

2.前端开发:

很多网站上有一些动画效果，也是通过定时器，比如每隔30ms，把页面往下滚动几个像素。

主要作用：使用定时器，让滞留时间太久的要求直接被删除

2. 标准库中的定时器

1. 标准库中提供了一个 Timer 类. Timer 类的核心方法为 schedule .

2. schedule 包含两个参数. 第一个参数指定即将要执行的任务代码, 第二个参数指定多长时间之后执行 (单位为毫秒).

```
1  public class ThreadDemo19 {
2   public static void main(String[] args) {
3   System.out.println("代码开始执行!");
4   Timer timer = new Timer();
5   timer.schedule(new TimerTask() {
6   @Override
7   public void run() {
8   System.out.println("触发定时器!");
9   }
10   },10000);
11   }
12  }
```

3、实现定时器

1. 定时器的构成

1. 一个带优先级的阻塞队列

为啥要带优先级

因为阻塞队列中的任务都有各自的执行时刻（delay）。最先执行的任务一定是 delay 最小的。使用带优先级的队列就可以高效的把这个 delay 最小的任务找出来。

2. 队列中的每个元素是一个 Task 对象

3. Task 中带有一个时间属性，队首元素就是即将到点执行的任务

4. 同时有一个 worker 线程一直扫描队首元素，看队首元素是否需要执行

2. 代码的实现

1. Timer 类提供的核心接口为 schedule, 用于注册一个任务, 并指定这个任务多长时间后执行.

```
public class Timer {
 public void schedule(Runnable command, long after) {
 // TODO
 }
}
```

2. Task 类用于描述一个任务(作为 Timer 的内部类). 里面包含一个 Runnable 对象和一个 time(毫秒时间戳)

这个对象需要放到 优先队列 中. 因此需要实现 Comparable 接口.

```
// 使用这个类来描述一个任务
static class Task implements Comparable<Task>{
 // command 代表这个任务是啥
 private Runnable command;
 // time 代表这个任务什么时候到时间
 // 这里的 time 使用 ms 级的时间戳来表示
 private long time;

 // 约定参数 time 是一个时间差
 // 希望 this.time 保存一个绝对的时间(毫秒级时间戳)
 public Task(Runnable command, long time) {
```

```
    this.command = command;
    this.time = System.currentTimeMillis() + time;
    }

    public void run(){
    command.run();
    }


    @Override
    public int compareTo(Task o) {
    return (int)(this.time - o.time );
    }
}
```

3. Timer 实例中, 通过 PriorityBlockingQueue 来组织若干个 Task 对象.

通过 schedule 来往队列中插入一个个 Task 对象.

```
public void secedule(Runnable command,long delay){
    Task task = new Task(command,delay);
    queue.put(task);

    // 每次插入新的任务都要唤醒扫描线程,让扫描线程能够重新计算 wait 的时间,保证新的任务不会错过
    synchronized (locker) {
    locker.notify();
    }
}
```

4.Timer 类中存在一个 worker 线程, 一直不停的扫描队首元素, 看看是否能执行这个任务.

```
class Timer {
    // ... 前面的代码不变
    public Timer() {
    // 启动 worker 线程
    Worker worker = new Worker();
```

```
worker.start();
}
class Worker extends Thread{
@Override
 public void run() {
 while (true) {
 try {
 Task task = queue.take();
 long curTime = System.currentTimeMillis();
 if (task.time > curTime) {
 // 时间还没到，就把任务再塞回去
 queue.put(task);
 } else {
 // 时间到了，可以执行任务
 task.run();
 }
 } catch (InterruptedException e) {
 e.printStackTrace();
 break;
 }
 }
 }
 }
}
```

但是当前这个代码中存在一个严重的问题, 就是 while (true) 转的太快了, 造成了无意义的 CPU 浪费.

比如第一个任务设定的是 1 min 之后执行某个逻辑. 但是这里的 while (true) 会导致每秒钟访问队首元素几万次. 而当前距离任务执行的时间还有很久呢.

5. 引入一个 locker 对象, 借助该对象的 wait / notify 来解决 while (true) 的忙等问题.

```
class Timer {
 // 存在的意义是避免 worker 线程出现忙等的情况
 private Object locker = new Object();
```

```
}
```

修改 Worker 的 run 方法, 引入 wait, 等待一定的时间.

```
public Timer(){
 // 来创建一个扫描线程,用来判定当前的任务,看看是不是已经到时间执行了
 Thread worker = new Thread(){
 @Override
 public void run() {
 while(true){
 try {
 // 取出队列的首元素,判定时间是不是到了
 Task task = queue.take();
  long curtime = System.currentTimeMillis();
  if(task.time > curtime){
  // 时间还没到
  // 前面的 take 操作会把首元素删除,
  // 但是此时队首元素的任务还没有执行,不能删持,于是需要重新插入回队列
  queue.put(task);
  // 指定等待时间 wait
  synchronized (locker) {
  locker.wait(task.time - curtime);
  }
  }else{
  // 时间到了,可以执行任务
  task.run();
  }
  } catch (InterruptedException e) {
  e.printStackTrace();
  // 如果出现了 interrupt 方法,就退出线程
  break;
  }
  }
  }
  };
  worker.start();
  }
```

```
}
```

修改 Timer 的 schedule 方法, 每次有新任务到来的时候唤醒一下 worker 线程. (因为新插入的任务可能是需要马上执行的).

```
public void secedule(Runnable command,long delay){
 Task task = new Task(command,delay);
 queue.put(task);
 // 每次插入新的任务都要唤醒扫描线程,让扫描线程能够重新计算 wait 的时间,保证新的任务不会错过
 synchronized (locker) {
 locker.notify();
 }
}
```

完整代码:

```
/**
* 定时器的构成:
* 一个带优先级的阻塞队列
* 队列中的每个元素是一个 Task 对象.
* Task 中带有一个时间属性, 队首元素就是即将
* 同时有一个 worker 线程一直扫描队首元素, 看队首元素是否需要执行
*/
public class ThreadExer3 {
 // 使用这个类来描述一个任务
 static class Task implements Comparable<Task>{
 // command 代表这个任务是啥
 private Runnable command;
 // time 代表这个任务什么时候到时间
 // 这里的 time 使用 ms 级的时间戳来表示
 private long time;

 // 约定参数 time 是一个时间差
 // 希望 this.time 保存一个绝对的时间(毫秒级时间戳)
 public Task(Runnable command, long time) {
 this.command = command;
 this.time = System.currentTimeMillis() + time;
 }
```

```
23
24  public void run(){
25  command.run();
26  }
27
28
29  @Override
30  public int compareTo(Task o) {
31  return (int)(this.time - o.time );
32  }
33  }
34
35  static class Timer{
36  // 使用这个带优先级的阻塞队列来组织这些任务
37  private PriorityBlockingQueue<Task> queue = new PriorityBlocki
ngQueue<Task>();
38
39  // 使用这个 Locker 对象
40  private Object locker = new Object();
41
42  public void secedule(Runnable command,long delay){
43  Task task = new Task(command,delay);
44  queue.put(task);
45
46  // 每次插入新的任务都要唤醒扫描线程,让扫描线程能够重新计算 wait 的时
间,保证新的任务不会错过
47  synchronized (locker) {
48  locker.notify();
49  }
50  }
51
52  public Timer(){
53  // 来创建一个扫描线程,用来判定当前的任务,看看是不是已经到时间执行了
54  Thread worker = new Thread(){
55  @Override
56  public void run() {
```

```
57  while(true){
58  try {
59  // 取出队列的首元素,判定时间是不是到了
60  Task task = queue.take();
61  long curtime = System.currentTimeMillis();
62  if(task.time > curtime){
63  // 时间还没到
64  // 前面的 take 操作会把首元素删除,
65  // 但是此时队首元素的任务还没有执行,不能删持,于是需要重新插入回队列
66  queue.put(task);
67  // 根据时间差进行一个等待
68  synchronized (locker) {
69  locker.wait(task.time - curtime);
70  }
71  }else{
72  // 时间到了
73  task.run();
74  }
75  } catch (InterruptedException e) {
76  e.printStackTrace();
77  // 如果出现了 interrupt 方法,就退出线程
78  break;
79  }
80  }
81  }
82  };
83  worker.start();
84  }
85  }
86
87  public static void main(String[] args) {
88  System.out.println("程序启动");
89  Timer timer = new Timer();
90  timer.secedule(new Runnable() {
91  @Override
```

```
92   public void run() {
93   System.out.println("hello world");
94   }
95   },3000);
96   }
97  }
```

4、线程池

1、线程池的引入

多进程的引入就是为了解决并发编程的问题，但是进程比较重量（创建和销毁，开销比较大）。因此引入了线程，线程比进程轻量的多。但是，如果在某些场景中，需要频繁的创建和销毁线程，此时，线程的创建和销毁，也就无法忽视了。

为了解决这样的问题：

1.引入携程

2,引入线程池

2、线程池的好处

使用线程池的时候，不是说使用的时候才创建，而是提前创建好，放到一个“池子里”（和字符串常量池，类似的东西）。

当我们使用线程的时候，就直接从池中取一个线程过来，当我们不使用这个线程的时候，就把这个线程放回到池中（此时的操作会比创建销毁线程效率更高）。

如果是创建、销毁线程，设计到用户态和内核态的切换（切换到内核态，然后创建出对应的PCB），如果不是真的创建销毁线程，而只是放回到池中，就相当于全在用户态可以解决这个事情。

在用户态操作，就会更高效，切换到内核态之后，操作就会很低效。

3、标准库中的线程池

面试题：

ThreadPoolExecutor 的构造方法的参数都是什么意思?

```
1 public ThreadPoolExecutor(int corePoolSize,
2  int maximumPoolSize,
3  long keepAliveTime,
4  TimeUnit unit,
```

```
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory)
```

1.corePoolSize - 要保留在池中的线程数，即使它们处于空闲状态，除非设置了 allowCoreThreadTimeOut

2.maximumPoolSize - 池中允许的最大线程数

3.keepAliveTime - 当线程数大于核心时，这是多余的空闲线程在终止之前等待新任务的最大时间。

4.unit - keepAliveTime参数的时间单位

5.workQueue - 在执行任务之前用于保存任务的队列。 这个队列只会保存execute方法提交的Runnable任务。

6.threadFactory - 执行程序创建新线程时使用的工厂

由于ThreadPoolExecutor 使用起来比较复杂,标准库中又提供了一组其他的类 ,相当于对ThreadPoolExecutor 又进行了一次封装。

标准库中提供了一个 Executors 类，这个类相当于一个“工厂类 ‘。通过这个类提供的一组工厂方法，可以创建不同风格的线程池实例。

- 使用 Executors.newFixedThreadPool(10) 能创建出固定包含 10 个线程的线程池.
- 返回值类型为 ExecutorService
- 通过 ExecutorService.submit 可以注册一个任务到线程池中.

```
ExecutorService pool = Executors.newFixedThreadPool(10);
 pool.submit(new Runnable() {
 @Override
 public void run() {
 System.out.println("hello");
 }
});
```

Executors创建线程的几种方式：

1.newFixedThreadPool: 创建固定线程数的线程池（完全没有临时工的版本）

2.newCachedThreadPool: 创建出一个数量可变的线程池（完全没有正式员工，全是临时工）

3.newSingleThreadExecutor: 创建只包含单个线程的线程池（只在特定场景下使用）.

4.newScheduledThreadPool: 设定延迟时间后执行命令，或者定期执行命令. 是进阶版的定时器(Timer)

Executors 本质上是 ThreadPoolExecutor 类的封装, ThreadPoolExecutor 提供了更多的可选参数, 可以进一步细化线程池行为的设定.

4、实现线程池

- 核心操作为 submit, 将任务加入线程池中
- 使用 Worker 类描述一个工作线程. 使用 Runnable 描述一个任务.
- 使用一个 BlockingQueue 组织所有的任务
- 每个 worker 线程要做的事情: 不停的从 BlockingQueue 中取任务并执行.
- 指定一下线程池中的最大线程数 maxWorkerCount; 当当前线程数超过这个最大值时, 就不再新线程了.

```
1  public class ThreadExer4 {
2   static class Worker extends Thread{
3   private BlockingQueue<Runnable> queue = null;
4   public Worker(BlockingQueue queue){
5   this.queue = queue;
6   }
7
8   // 工作线程的具体安排
9   // 需要从阻塞队列取出任务
10   @Override
11   public void run() {
12   try {
13   while (!Thread.interrupted()) {
14   Runnable command = queue.take();
15   command.run();
16   }
17   } catch (InterruptedException e) {
18   e.printStackTrace();
```

```
19  }
20  }
21  }
22
23  static class ThreadPool{
24  // 包含一个阻塞队列,用来组织任务
25  BlockingQueue<Runnable> queue = new LinkedBlockingDeque<>();
26
27  // List 用来存放当前的工作线程
28  List<Thread> workers = new ArrayList<>();
29
30  private static final int MAX_WORKERS_COUNT = 10;
31
32  // 通过这个方法,将线程加入到线程池中
33  // sumbit 不仅可以把把线程加入到阻塞队列中,同时还可以负责创建线程
34  public void sumbit(Runnable command) throws InterruptedExcepti
on {
35  if(workers.size() < MAX_WORKERS_COUNT){
36  // 如果当前工作线程的数目小于线程数目的上限,就创建出新的线程
37  // 工作线程见专门来创建一个类来完成
38  // worker 内部要能够取到队列的内容,就需要把这个队列实例通过 worker
的构造方法,传过去
39  Worker worker = new Worker(queue);
40  worker.start();
41  workers.add(worker);
42  }
43  // 将任务添加到任务列中
44  queue.put(command);
45  }
46  }
47
48  public static void main(String[] args) throws InterruptedExcep
tion {
49  ThreadPool pool = new ThreadPool();
50  pool.sumbit(new Runnable() {
51  @Override
```

```
52  public void run() {
53  for(int i = 1; i <= 20; i ++){
54  System.out.println(i + ": hello");
55  }
56  }
57  });
58  Thread.sleep(5000);
59  }
60 }
```