

# 多线程基础

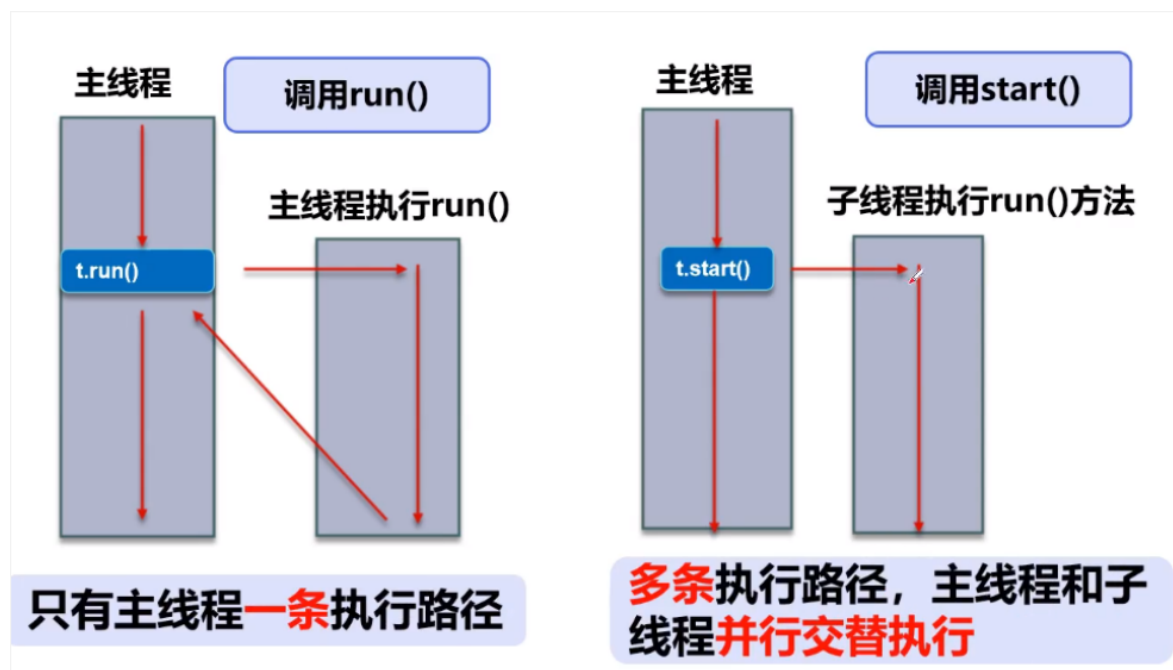
## 线程简介

Java.Thread

### 什么是多任务

多任务看起来是多个任务都在做，但是其实同一个时间都在做一件事情，只不过切换的非常快

### 普通方法和多线程



### 程序，进程，线程

操作系统中运行的程序都是进程

一个进程中存在多个线程

### Process与Thread

- 程序是指令和程序的有序集合，其本身没有任何运行的含义，是一个静态的概念
- 进程则是执行程序的一次执行过程，他是一个动态的概念，是系统资源分配的单位
- 一个进程中存在多个线程，至少有一个线程。
- 线程是CPU调度和执行的单位

很多多线程是模拟出来的，真正的多线程是指有多个cpu，如服务器。

如果是模拟出来的多线程，在同一个cpu下面，在同一个时间点，cpu只可以执行一个代码，因为切换的很快，所以就有同时执行的错觉

- 线程就是独立的执行路径
- 在程序运行时，即使没有自己创建线程，后台也会有多个线程,如主线程, gc线程
- main()称之为主线程,为系统的入口，用于执行整个程序
- 在一个进程中，如果开辟了多个线程，线程的运行由调度器安排调度,调度器是与操作系统紧密相关的，先后顺序是不能人为的干预的
- **对同一份资源操作时，会存在资源抢夺的问题，需要加入并发控制**
- **线程会带来额外的开销**，如cpu调度时间，并发控制开销
- **每个线程在自己的工作内存交互，内存控制不当会造成数据不一致**

## 线程实现

### 三种方法

三种：Thread, Runnable, Callable

1. 继承 Thread 类，重写 run 方法并开启它
2. 实现 Runnable 接口
3. 实现 Callable 接口（重要）

### Thread

```
1  /*
2  * 1. 继承Thread
3  * 2. 重写Run
4  * 3. 调用start
5  * */
6  public class ThreadDemo1 extends Thread{
7      @Override
8      public void run() {
9          for (int i = 0; i < 200; i++) {
10             System.out.println("thread的run方法执行了...");
11         }
12     }
13
14
15     //主线程执行
16     public static void main(String[] args) {
17
18         ThreadDemo1 thread = new ThreadDemo1();
19
20         thread.start();
21
22         for (int i = 0; i < 1000; i++) {
23             System.out.println("主线程执行");
```

```
24     }
25
26     }
27 }
```

线程开启之后不一定立刻执行，由CPU调度

## 实现Runnable

```
1  /*
2  * 1. 实现Runnable
3  * 2. 重写run方法
4  * 3. 执行线程需要丢入Runnable接口的实现类
5  * 4. 调用start
6  * */
7  public class RunnableDemo1 implements Runnable{
8      @Override
9      public void run() {
10         for (int i = 0; i < 200; i++) {
11             System.out.println("thread的run方法执行了...");
12         }
13     }
14
15
16     //主线程执行
17     public static void main(String[] args) {
18
19         RunnableDemo1 runnable = new RunnableDemo1();
20
21         Thread thread = new Thread(runnable);
22
23         thread.start();
24
25         for (int i = 0; i < 1000; i++) {
26             System.out.println("main方法执行...");
27         }
28     }
29 }
30 }
```

对比实现Runnable和继承Thread，推荐实现Runnable，因为Java是单继承

## 实现Callable

```
1  /*
2  * 1. 实现Callable接口，需要返回值类型
3  * 2. 重写call方法，需要抛出异常
4  * 3. 创建目标对象
5  * 4. 创建执行任务
6  * 5. 提交执行
7  * 6. 获取结果
```

```

8      * 7. 关闭服务
9      */
10     public class CallableDemo1 implements Callable<Boolean> { //这里的泛型是Boolean
11
12         private String name;
13
14         public CallableDemo1(String name) {
15             this.name = name;
16         }
17
18         @Override
19         public Boolean call() throws Exception {
20             System.out.println("call方法执行: "+name);
21             return true; //永远返回true
22         }
23
24
25
26
27         public static void main(String[] args) {
28
29             //新建了三个callable
30             CallableDemo1 callable1 = new CallableDemo1("callable1");
31             CallableDemo1 callable2 = new CallableDemo1("callable2");
32             CallableDemo1 callable3 = new CallableDemo1("callable3");
33
34             // 创建执行服务, 创建了一个线程池, 里面有3个线程
35             ExecutorService service = Executors.newFixedThreadPool(3);
36
37             //把三个callable提交执行
38             Future<Boolean> submit1 = service.submit(callable1);
39             Future<Boolean> submit2 = service.submit(callable2);
40             Future<Boolean> submit3 = service.submit(callable3);
41
42             //获得结果
43             try {
44                 Boolean result1 = submit1.get();
45                 Boolean result2 = submit2.get();
46                 Boolean result3 = submit3.get();
47             } catch (InterruptedException e) {
48                 e.printStackTrace();
49             } catch (ExecutionException e) {
50                 e.printStackTrace();
51             }
52
53             //关闭服务
54             service.shutdown();
55         }
56
57     }

```

## 静态代理

我们以结婚为例子，作为静态代理的例子

```

1  package com.bean.proxy;
2
3  //代理对象和真实对象都要实现一个接口
4  interface Marry{
5      void HappyMarry();
6  }
7
8  //真实对象
9  class Man implements Marry{
10
11      @Override
12      public void HappyMarry() {
13          System.out.println("男方结婚");
14      }
15  }
16
17  //代理对象，婚庆公司
18  class WeddingCompany implements Marry{
19
20      //代理的目标，真实对象
21      private Marry target;
22
23      public WeddingCompany(Marry target) {
24          this.target = target;
25      }
26
27      @Override
28      public void HappyMarry() {
29          before();
30          //实现真实对象的代理
31          target.HappyMarry();
32          after();
33      }
34
35      private void before(){
36          System.out.println("结婚之前布置现场");
37      }
38
39      private void after(){
40          System.out.println("结婚之后收钱");
41      }
42  }
43
44  public class StaticProxy {
45      public static void main(String[] args) {
46          Man man = new Man();
47          WeddingCompany weddingCompany = new WeddingCompany(man);
48          weddingCompany.HappyMarry();
49      }
50
51  }

```

1. 真实对象和代理对象都要实现同一个接口
2. 代理对象要代理真实角色

好处：

1. 代理对象可以做真实对象做不了的
2. 真实对象专注于自己的事情

## Lambda

---

### 为什么要使用lambda

- 避免匿名内部类定义过多
- 可以让代码觉得简洁
- 去掉了没有意义的代码

### 函数式接口

假如一个接口只包含一个抽象方法，那么就叫做函数式接口

函数式接口可以实现Lambda

```
1 public class LambdaDemo1{
2
3     public static void main(String[] args) {
4         ILike like = ()->System.out.println("输出");
5     }
6 }
7
8 interface ILike{
9     void show();
10 }
```

```
1 package com.bean.lambda;
2
3 public class LambdaDemo1{
4
5     public static void main(String[] args) {
6         ILike like = ()->{
7             System.out.println("输出");
8             System.out.println("另一句输出");
9         };
10    }
11 }
12
13 interface ILike{
14     void show();
15 }
```

```

1  public class LambdaDemo1{
2
3      public static void main(String[] args) {
4          ILike like = i->System.out.println("输出"+i);
5      }
6  }
7
8  interface ILike{
9      void show(int i);
10 }

```

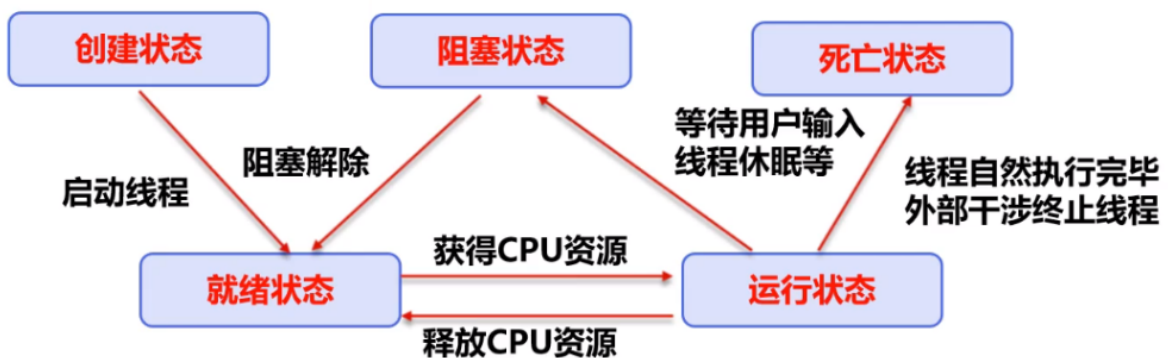
```

1  public class LambdaDemo1{
2
3      public static void main(String[] args) {
4          ILike like = (i,s)->System.out.println("输出"+i+s);
5      }
6  }
7
8  interface ILike{
9      void show(int i,String s);
10 }

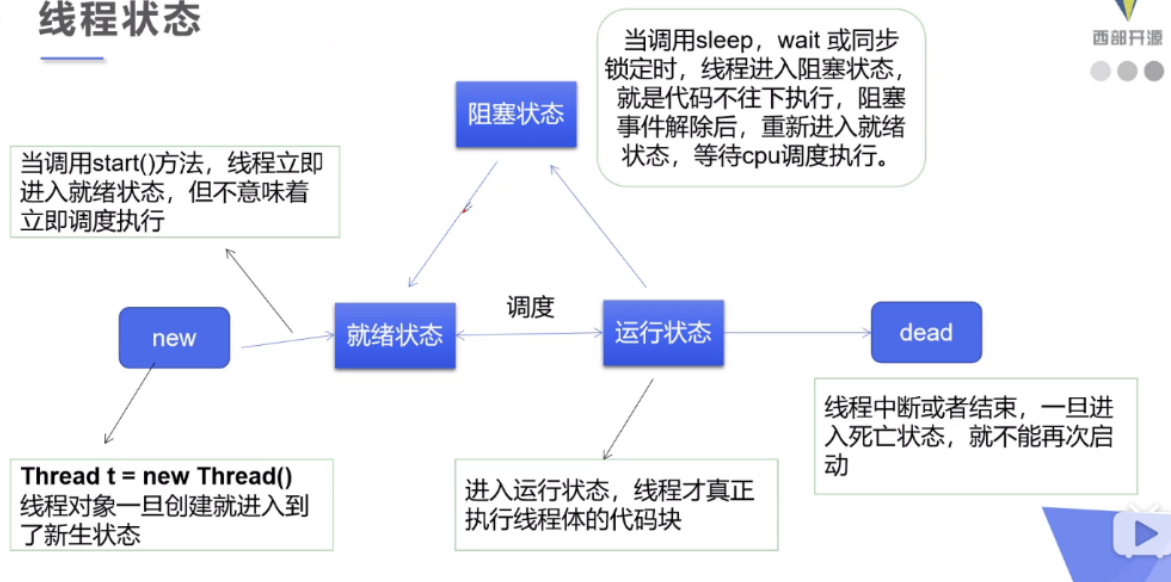
```

## 线程状态

### 线程的五大状态



## 线程状态



## 线程方法

方法	说明
<code>setPriority(int newPriority)</code>	更改线程的优先级
<code>static void sleep(long millis)</code>	在指定的毫秒数内让当前正在执行的线程休眠
<code>void join()</code>	等待该线程终止
<code>static void yield()</code>	中断线程，不推荐使用 推荐使用一个 <code>boolean flag</code> ，让线程自己停下
<code>boolean isAlive()</code>	测试线程是否存活

## 线程停止

```

1  public class StopDemo implements Runnable {
2
3      // 设置一个标志位
4      boolean flag = true;
5
6      @Override
7      public void run() {
8          int i = 0;
9          while (flag){
10             System.out.println(i+ " ");
11             i++;
12         }
13     }
14
15     public void stop(){
16         flag = false;
17     }
18

```



```

19     public static void main(String[] args) {
20         StopDemo stop = new StopDemo();
21
22         new Thread(stop).start();
23
24         for (int i = 0; i < 1000000; i++) {
25             if (i==900000){
26                 stop.stop();
27             }
28         }
29     }
30 }
31 }

```

## 线程休眠

```

1     public class SleepDemo implements Runnable{
2         @Override
3         public void run() {
4             for (int i = 0; i < 10; i++) {
5                 System.out.println(i+"");
6                 if (i== 5){
7                     try {
8                         Thread.sleep(3000);//线程休眠
9                     } catch (InterruptedException e) {
10                        e.printStackTrace();
11                    }
12                }
13            }
14        }
15
16        public static void main(String[] args) {
17            SleepDemo sleep = new SleepDemo();
18            new Thread(sleep).start();
19        }
20    }

```

线程休眠可以模拟网络延时，放大问题

每一个对象都有一个锁，但是sleep不会释放锁

## 线程礼让

让当前执行的线程停止，但是不阻塞

礼让不一定成功，看cpu

```

1     public class YiledDemo {
2         public static void main(String[] args) {
3             MyYield yield = new MyYield();
4             //第二个参数：给线程起名字
5             new Thread(yield,"A").start();

```

```

6         new Thread(yield, "B").start();
7     }
8 }
9
10 class MyYield implements Runnable{
11
12     @Override
13     public void run() {
14         //Thread.currentThread().getName(): 获取线程名字
15         System.out.println(Thread.currentThread().getName()+"线程开始执行");
16         Thread.yield();//礼让
17         System.out.println(Thread.currentThread().getName()+"线程停止执行");
18     }
19 }

```

## 线程强制执行

通俗讲就是插队

```

1
2 public class JoinDemo implements Runnable {
3     @Override
4     public void run() {
5         for (int i = 0; i < 10; i++) {
6             System.out.println("线程vip插队");
7         }
8     }
9
10    public static void main(String[] args) {
11        JoinDemo join = new JoinDemo();
12        Thread thread = new Thread(join);
13        thread.start();
14
15        for (int i = 0; i < 50; i++) {
16            if (i==20){
17                try {
18                    thread.join();//等待thread插队
19                } catch (InterruptedException e) {
20                    e.printStackTrace();
21                }
22            }
23            System.out.println("main"+i);
24        }
25    }
26 }

```

注意，Join可以保证vip这个线程先走完

## 线程状态观测

Thread.State

- Thread.State.New：尚未启动

- Thread.State.Runable：在JVM中执行
- Thread.State.Block：阻塞
- Thread.State.Waiting：等待另一个线程执行动作
- Thread.State.Timed\_Waiting：等待另一个线程指定动作达到指定的事件
- Thread.State.Terminated：退出的线程

死亡的线程不能再启动了，所以线程只能启动一次

```
1 Thread.State state = thread.getState();
```

## 线程的优先级

JAVA提供一个线程调度器来监控程序中启动后进入就绪状态的所有线程，线程调度器按照优先级决定应该调用哪个线程来执行

**线程的优先级用数字来显示，范围从1~10**

- Thread.MIN\_PRIORITY=1
- Thread.MAX\_PRIORITY=10
- Thread.NORM\_PRIORITY=5

**以下方式改变线程或者获取线程的优先级**

- getPriority()
- setPriority(int xxx)

主线程是默认优先级

```
1 //测试线程优先级
2 public class PriorityDemo1 extends Thread{
3
4     public static void main(String[] args) {
5         //主线程是默认优先级
6         System.out.println(Thread.currentThread().getName()+"-->"+Thread.currentThread().getPriority()); //main-->5
7
8         MyPriority priority = new MyPriority();
9
10        Thread thread1 = new Thread(priority);
11        Thread thread2 = new Thread(priority);
12        Thread thread3 = new Thread(priority);
13        Thread thread4 = new Thread(priority);
14        Thread thread5 = new Thread(priority);
15        Thread thread6 = new Thread(priority);
16        Thread thread7 = new Thread(priority);
17
18
19        thread1.start(); //Thread-0-->5
20
```

```

21     thread2.setPriority(1);        //Thread-1-->1
22     thread2.start();
23
24     thread3.setPriority(4);        //Thread-2-->4
25     thread3.start();
26
27     thread4.setPriority(Thread.MAX_PRIORITY);    //Thread-3-->10
28     thread4.start();
29
30     /*
31     main-->5
32     Thread-3-->10
33     Thread-0-->5
34     Thread-2-->4
35     Thread-1-->1
36     */
37
38 }
39
40 }
41
42 class MyPriority implements Runnable {
43
44     @Override
45     public void run() {
46         System.out.println(Thread.currentThread().getName()+"--
47         >"+Thread.currentThread().getPriority());
48     }
49 }

```

注意了，其实并不是优先级高的时候就一定先跑，只不过是提高了先跑的概率  
cpu也有可能先调用优先级低的，这种情况叫做性能倒置，不过一般不会出现

而且要注意，要先设置优先级，然后在启动

## 守护线程

线程分为 **用户线程** 和 **守护线程**

- 虚拟机必须保证用户线程执行完毕
- 虚拟机不用等待守护线程执行完毕
- 守护线程如：后台记录操作日志，监控内存，垃圾回收等等

```

1     public class GuardDemo {
2         public static void main(String[] args) {
3             Guard guard = new Guard();
4
5             User user = new User();
6
7
8             Thread userThread = new Thread(user);    //默认都是用户线程
9
10            Thread guardThread = new Thread(guard); //默认都是用户线程

```

```

11         guardThread.setDaemon(true);    //设置为守护线程，默认都是用户线程
12
13         guardThread.start();
14         userThread.start();
15     }
16 }
17
18 class Guard implements Runnable{
19
20     @Override
21     public void run() {
22         while (true){
23             System.out.println("守护线程");
24         }
25     }
26 }
27
28
29 class User implements Runnable{
30
31     @Override
32     public void run() {
33         for (int i = 0; i < 10; i++) {
34             System.out.println("用户线程");
35         }
36     }
37 }

```

## 线程同步

- 并发：多个线程操作同一个资源

比如10000去抢一张票，两个银行同时去取钱  
对于并发问题，我们需要用到对象的等待池去排队

- 队列和锁

形成队列之后，我们也要保证同一个资源在同一时间只有一个线程可以访问，这个时候我们就需要用到锁，来保证资源的访问

我们在之前在讲解 **sleep** 说过，每个对象都拥有一把锁，而sleep不会释放锁

也就是说，解决并发问题安全性需要两步：

1. 多个线程形成队列
2. 资源上锁

在访问的时候假如锁机制：**synchronized**，当一个线程获得对象的排它锁，独占资源，其余线程必须等待，使用之后释放锁即可

存在以下问题：

- 一个线程持有锁会导致其他所有需要此锁的线程挂起，效率变低了
- 多线程竞争下，加锁和释放锁会导致比较多的上下文切换和调度延时，引起性能问题
- 如果一个优先级高的线程等待一个线程优先级低的线程释放锁，那么会导致优先级倒置，引起性能问题

## 不安全例子

- 买票

```
1 //线程不安全
2 public class UnSafeDemo {
3
4     public static void main(String[] args) {
5         BuyTicket buyTicket = new BuyTicket();
6
7         new Thread(buyTicket, "A").start();
8         new Thread(buyTicket, "B").start();
9         new Thread(buyTicket, "C").start();
10    }
11 }
12
13 class BuyTicket implements Runnable{
14
15     private int ticket = 10; //10张票
16     private boolean flag = true;
17     @Override
18     public void run() {
19         //买票
20         while (flag){
21             buy();
22         }
23     }
24
25     private void buy(){
26         if (ticket <= 0){
27             flag = false;
28             return;
29         }
30
31         try {
32             Thread.sleep(100); //模拟延时，增大问题发生率
33         } catch (InterruptedException e) {
34             e.printStackTrace();
35         }
36         //买票
37         System.out.println(Thread.currentThread().getName()+"-->"+ticket--);
38     }
39 }
```

每个线程都有自己的工作内存开销，内存控制不当会导致数据不一致

在这个例子里，有可能导致抢到了同一张票的情况，甚至可能会有抢到了负数的情况

- 银行取钱

```
1 //两个人去银行取钱
2 public class UnSafeDemo2 {
3     public static void main(String[] args) {
4         Account account = new Account(100, "基金");
5
6         Drawing man = new Drawing(account, 50, "男方");
7         Drawing women = new Drawing(account, 100, "女方");
8
9         man.start();
10        women.start();
11        /*
12        基金余额为: 0
13        基金余额为: -50
14        女方手里的钱: 100
15        男方手里的钱: 50
16        */
17    }
18 }
19
20 //账户
21 class Account{
22     int money; //余额
23     String name;//卡号
24
25     public Account(int money, String name) {
26         this.money = money;
27         this.name = name;
28     }
29 }
30
31 //银行, 模拟取款
32 class Drawing extends Thread{
33     Account account;//账户
34     int drawingMoney;//取了多少钱
35     int nowMoney; //现在手里有多少钱
36
37     public Drawing(Account account,int drawingMoney,String name){
38         super(name);//给Thread起名字
39         this.account = account;
40         this.drawingMoney = drawingMoney;
41     }
42
43     @Override
44     public void run() {
45         //判断有钱么
46         if (account.money-drawingMoney<0){
47             System.out.println(Thread.currentThread().getName()+"：钱不够");
48             return;
49         }
50
51         //模拟延时
52         try {
53             Thread.sleep(1000);
54         } catch (InterruptedException e) {
```

```

55         e.printStackTrace();
56     }
57
58     //卡内余额
59     account.money = account.money-drawingMoney;
60     //手中的钱
61     nowMoney = nowMoney+drawingMoney;
62
63     System.out.println(account.name+"余额为: "+account.money);
64     System.out.println(this.getName()+"手里的钱: "+nowMoney);
65 }
66 }

```

- 集合

```

1 //线程不安全的集合
2 public class UnSafeList {
3     public static void main(String[] args) {
4         List<String> lists = new ArrayList<>();
5
6         for (int i = 0; i < 10000; i++) {
7             new Thread()->{
8                 lists.add(Thread.currentThread().getName());
9             }.start();
10        }
11        try {
12            Thread.sleep(3000);
13        } catch (InterruptedException e) {
14            e.printStackTrace();
15        }
16
17        System.out.println(lists.size()); //9997
18    }
19 }

```

集合是不安全的

集合不安全的原因在于：线程的内存都是各自的，我们会发现每一个线程都有自己的内存，但是线程操作的都是一个资源

对于每一个线程来说，所看到的资源是全部的，但是程序不知道还会有其他线程来操作资源

所以就会出现这种问题

归根结底的问题就是：**线程在自己的工作内存进行交互**

## 同步方法和同步代码块

由于我们可以通过 `private` 关键字来保证数据对象只能被方法访问，所以我们只需要对方法提出一套机制，这套机制就是 **synchronized** 关键字，它包括两种用法：

### 同步方法

类似：`public synchronized void method(int args){}`，只需要在方法上加上 **synchronized** 即可



synchronized方法访问控制对 **对象** 的访问

### 每一个对象对应一把锁

每一个 **synchronized** 方法都必须获得该方法的对象的锁才可以执行，否则线程就会被阻塞

方法一旦执行，就独占该锁，直到方法返回才释放锁，后面被阻塞的线程才可以获得这个锁，继续执行

缺陷：若将一个大的方法声明为 **synchronized**，将会影响效率

缺陷：

我们知道，只读代码其实并不需要加锁，但是修改代码需要加锁

如果我们使用同步方法，那么只读代码也会被加上锁，影响效率

我们对上面不安全的例子进行修改，比如修改买票的例子：

```
1  public class UnSafeDemo {
2
3      public static void main(String[] args) {
4          BuyTicket buyTicket = new BuyTicket();
5
6          new Thread(buyTicket, "A").start();
7          new Thread(buyTicket, "B").start();
8          new Thread(buyTicket, "C").start();
9      }
10 }
11
12 class BuyTicket implements Runnable{
13
14     private int ticket = 10;
15     private boolean flag = true;
16     @Override
17     public void run() {
18         while (flag){
19             buy();
20         }
21     }
22
23     private synchronized void buy(){ //我们只需要在修改的方法上加上同步标签，将这个方法改为同步方法
24         if (ticket <= 0){
25             flag = false;
26             return;
27         }
28
29         try {
30             Thread.sleep(100);
31         } catch (InterruptedException e) {
32             e.printStackTrace();
33         }
34
35         System.out.println(Thread.currentThread().getName()+"-->"+ticket--);
36     }
37 }
```

加上了同步代码块之后，实现了

1. 排队
2. 锁

这样实现了同步代码块的锁定

但是我们注意，这个锁的意思是锁定了一个对象，因为每一个对象都有一把锁

我们加上了同步标签锁定的是这一个对象，但是假如拥有了多个对象，这个同步方法就没用了

看下面的例子：

```
1  public class UnSafeDemo2 {
2      public static void main(String[] args) {
3          Account account = new Account(100, "基金");
4
5          Drawing man = new Drawing(account, 50, "男方");
6          Drawing women = new Drawing(account, 100, "女方");
7
8          man.start();
9          women.start();
10     }
11 }
12
13
14 class Account{
15     int money;
16     String name;
17
18     public Account(int money, String name) {
19         this.money = money;
20         this.name = name;
21     }
22 }
23
24
25 class Drawing extends Thread{
26     Account account;
27     int drawingMoney;
28     int nowMoney;
29
30     public Drawing(Account account, int drawingMoney, String name){
31         super(name);
32         this.account = account;
33         this.drawingMoney = drawingMoney;
34     }
35
36     @Override
37     public synchronized void run() {
38
39         if (account.money - drawingMoney < 0){
40             System.out.println(Thread.currentThread().getName() + ": 钱不够");
41             return;
42         }
43
44     }
```

```

45         try {
46             Thread.sleep(1000);
47         } catch (InterruptedException e) {
48             e.printStackTrace();
49         }
50
51
52         account.money = account.money-drawingMoney;
53
54         nowMoney = nowMoney+drawingMoney;
55
56         System.out.println(account.name+"余额为: "+account.money);
57         System.out.println(this.getName()+"手里的钱: "+nowMoney);
58     }
59 }

```

如果你去运行这个代码，你会发现：

这个同步方法好像没有用

分析：

我们锁定的是Run方法，而run方法对应的对象是Drawing对象，所以我们操作的锁其实是 **Drawing** 的锁

但是我们增删改的对象并不是 **Drawing** 对象，而是Account对象

所以同步方法没用了

## 同步代码块

对于上面的问题，我们可以使用同步代码块来进行实现同步

同步代码块： `synchronized(obj){}`

其中obj称为同步监视器

- obj可以为任何对象，但是推荐使用 **共享资源** 作为同步监视器
- 同步方法中无需指定同步监视器，因为同步方法的同步监视器就是 **this** 这个对象本身，或者是 **class**

### • 同步监视器的执行过程

1. 第一个线程访问，锁定同步监视器，执行其中代码
2. 第二个线程访问，发现同步监视器被锁定，无法访问
3. 第一个线程访问完毕，解锁同步监视器
4. 第二个线程访问，锁定同步监视器，执行其中代码

例子，还是上面那个银行取钱的案例

```

1     public class UnSafeDemo2 {
2         public static void main(String[] args) {
3             Account account = new Account(100,"基金");
4
5             Drawing man = new Drawing(account,50,"男方");

```

```

6      Drawing women = new Drawing(account,100,"女方");
7
8      man.start();
9      women.start();
10     }
11 }
12
13
14 class Account{
15     int money;
16     String name;
17
18     public Account(int money, String name) {
19         this.money = money;
20         this.name = name;
21     }
22 }
23
24
25 class Drawing extends Thread{
26     Account account;
27     int drawingMoney;
28     int nowMoney;
29
30     public Drawing(Account account,int drawingMoney,String name){
31         super(name);
32         this.account = account;
33         this.drawingMoney = drawingMoney;
34     }
35
36     @Override
37     public void run() {
38
39         synchronized (account){
40             if (account.money-drawingMoney<0){
41                 System.out.println(Thread.currentThread().getName()+"： 钱不够");
42                 return;
43             }
44
45
46             try {
47                 Thread.sleep(1000);
48             } catch (InterruptedException e) {
49                 e.printStackTrace();
50             }
51
52
53             account.money = account.money-drawingMoney;
54
55             nowMoney = nowMoney+drawingMoney;
56
57             System.out.println(account.name+"余额为: "+account.money);
58             System.out.println(this.getName()+"手里的钱: "+nowMoney);
59         }
60     }
61 }

```

成功了

我们锁定的是 **公共资源**，这里的公共资源其实就是account

或者可以这样认为：我们进行增删改的对象是account，但是我们执行的方法是run，run对应的对象不是增删改所对应的对象，所以使用同步代码块，锁住公共资源

## CopyOnWriteArrayList

首先看下面一段代码：

```
1  import java.util.concurrent.CopyOnWriteArrayList;
2
3  public class TestJUC {
4
5      public static void main(String[] args) {
6          CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<String>();
7
8          for (int i = 0; i < 10000; i++) {
9              new Thread()->{
10                  list.add(Thread.currentThread().getName());
11              }.start();
12          }
13          try {
14              Thread.sleep(3000);
15          } catch (InterruptedException e) {
16              e.printStackTrace();
17          }
18          System.out.println(list.size());
19      }
20  }
```

运行完成之后发现：这个是线程安全的

点进源码看一下：

```
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    private static final long serialVersionUID = 8673264195747942595L;

    /** The lock protecting all mutators */
    final transient ReentrantLock lock = new ReentrantLock(); 死锁

    /** The array, accessed only via getArray/setArray. */
    private transient volatile Object[] array; JUC里面的关键字
```

## 死锁

线程只有获取完整的资源才可以运行，但是现在多个线程平分了一些共享资源

线程占有着自己的资源，并且都等待着其他的线程释放另外的资源，因为资源不完全，这些线程全部都不能执行

这种情况叫做 **死锁**

多个线程互相抱着对方需要的资源，然后形成僵持

某一个同步块同时拥有 **两个以上对象的锁** 时，就可能发生死锁现象

```
1  public class DeadLock {
2      public static void main(String[] args) {
3          Makeup girl1 = new Makeup(0,"girlA");
4          Makeup girl2 = new Makeup(1,"girlB");
5
6
7          girl1.start();
8          girl2.start();
9      }
10
11 }
12
13
14
15 //口红
16 class Lipstick{
17
18 }
19
20 //镜子
21 class Mirror{
22
23 }
24
25 class Makeup extends Thread{
26
27     //使用static来保证只有一份
28     static Lipstick lipstick = new Lipstick();
29     static Mirror mirror = new Mirror();
30
31     int choice;//选择
32     String girlName;
33
34     Makeup(int choice,String girlName){
35         this.choice = choice;
36         this.girlName =girlName;
37     }
38
39
40     @Override
41     public void run() {
42         //化妆
43         try {
```

```

44         makeup();
45     } catch (InterruptedException e) {
46         e.printStackTrace();
47     }
48 }
49
50 //化妆需要互相持有对方的锁，就是需要拿到对方的资源
51 private void makeup() throws InterruptedException {
52     if (choice==0){
53         synchronized (lipstick){//获得口红的锁
54             System.out.println(girlName+"获得口红的锁");
55             Thread.sleep(1000);
56             synchronized (mirror){//1秒钟之后想要获得镜子的锁
57                 System.out.println(girlName+"获得镜子的锁");
58             }
59         }
60     }else {
61         synchronized (mirror){//获得镜子的锁
62             System.out.println(girlName+"获得镜子的锁");
63             Thread.sleep(2000);
64             synchronized (lipstick){//2秒钟之后想要获得口红的锁
65                 System.out.println(girlName+"获得口红的锁");
66             }
67         }
68     }
69 }
70 }

```

这样的话，程序卡住了，因为这样会导致死锁

这是因为：

- A和B都想要对方的锁
- A和B都没有释放锁的途径

所以会产生死锁

我们知道，A和B想要对方的锁没有关系，但是 **一定要让A和B有释放锁的途径**，否则只是期待资源而不释放资源，早晚会产生死锁的现象

我们这么改：

```

1     public class DeadLock {
2         public static void main(String[] args) {
3             Makeup girl1 = new Makeup(0,"girlA");
4             Makeup girl2 = new Makeup(1,"girlB");
5
6
7             girl1.start();
8             girl2.start();
9         }
10    }
11
12
13
14
15    //口红

```

```

16  class Lipstick{
17
18  }
19
20  //镜子
21  class Mirror{
22
23  }
24
25  class Makeup extends Thread{
26
27      //使用static来保证只有一份
28      static Lipstick lipstick = new Lipstick();
29      static Mirror mirror = new Mirror();
30
31      int choice;//选择
32      String girlName;
33
34      Makeup(int choice,String girlName){
35          this.choice = choice;
36          this.girlName =girlName;
37      }
38
39
40      @Override
41      public void run() {
42          //化妆
43          try {
44              makeup();
45          } catch (InterruptedException e) {
46              e.printStackTrace();
47          }
48      }
49
50      //化妆需要互相持有对方的锁，就是需要拿到对方的资源
51      private void makeup() throws InterruptedException {
52          if (choice==0){
53              synchronized (lipstick){//获得口红的锁
54                  System.out.println(girlName+"获得口红的锁");
55                  Thread.sleep(1000);
56              }
57              synchronized (mirror){//1秒钟之后想要获得镜子的锁
58                  System.out.println(girlName+"获得镜子的锁");
59              }
60          }else {
61              synchronized (mirror){//获得镜子的锁
62                  System.out.println(girlName+"获得镜子的锁");
63                  Thread.sleep(2000);
64              }
65              synchronized (lipstick){//2秒钟之后想要获得口红的锁
66                  System.out.println(girlName+"获得口红的锁");
67              }
68          }
69      }
70  }

```



这个办法就是把里面的同步代码块放到了外面，这样虽然A和B还是期望对方的锁，但是A和B都有释放锁的途径

得不到锁，在代码执行完之后还是会释放锁

产生死锁的四个必要条件：

- 互斥条件：一个资源每次只能被一个进程使用
- 请求与保持：一个进程因为请求资源而被阻塞时，对已经获得的资源保持不放
- 不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺
- 循环等待条件：若干进程形成一种头尾相接的循环等待资源关系

这四个条件只要剥夺一条，就会破开死锁的局面

而往往比较容易破开的条件就是请求与保持条件

## Lock

从 **JDK5.0** 开始，JAVA提供了一个更加强大的线程同步机制：通过显式定义同步锁对象来实现同步。Lock对象充当同步锁

**java.util.concurrent.locks.Lock** 接口 是控制多个线程对共享资源访问的工具

锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应该先获得Lock对象

**ReentrantLock** 类实现了Lock（**可重用锁**），它拥有与 **synchronized** 相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是 **ReentrantLock**，可以显示加锁，释放锁

```
1  public class LockDemo {
2      public static void main(String[] args) {
3          TestLock testLock = new TestLock();
4
5          new Thread(testLock).start();
6          new Thread(testLock).start();
7          new Thread(testLock).start();
8      }
9  }
10
11
12  class TestLock implements Runnable{
13
14      //定义Lock锁
15      private final ReentrantLock lock = new ReentrantLock();
16
17      int ticket = 10;
18  }
```

```

19     @Override
20     public void run() {
21         while (true){
22             lock.lock();//加锁
23
24             try {
25                 if (ticket>0){
26                     try {
27                         Thread.sleep(1000);
28                     } catch (InterruptedException e) {
29                         e.printStackTrace();
30                     }
31                     System.out.println(ticket--);
32                 }else {
33                     break;
34                 }
35             } finally {
36                 lock.unlock();
37             }
38         }
39     }
40 }

```

## Synchronized与Lock对比

### Lock

1. 是显示锁（手动开启和关闭锁，不要忘记关闭）
2. 只有代码块锁
3. JVM将花费较少的时间来调度线程，性能更好，并且有更好的扩展性（子类更多）

### Synchronized

1. 是隐式锁，出了作用域自动释放
2. 有代码块锁和方法锁

优先级：Lock>同步代码块>同步方法块

## 线程通信

## 生产者和消费者模式

生产者和消费者并不是一个模式，而是一个问题，不是23中设计模式之一

生产者是一个线程

消费者是一个线程

生产者生产产品放到仓库，消费者从仓库中取走消费

仓库中只可以存放一个产品

如果仓库中没有产品，则生产者将产品放入仓库，直到仓库中的产品被消费者取走为止才可以继续生产

如果仓库中存在产品，消费者可以从仓库取走产品，如果仓库中没有产品则一直等待

这是一个线程同步问题

生产者和消费者共享同一个资源，消费者和生产者共享同一个资源，而且这两者之间相互依赖，互为条件

在生产者和消费者问题中，仅有 `synchronized` 是不够的

- `synchronized` 可以阻止并发更新同一个资源，实现同步
- 但是不可以用来实现不同线程之间的通信

JAVA提供了几个方法来解决线程中的通信问题

方法名	作用
<code>wait()</code>	表示线程会一直等待，直到其他线程通知， <code>sleep</code> 不会释放锁，但是 <code>wait()</code> 会释放锁
<code>wait(long timeout)</code>	指定等待的毫秒数
<code>notify()</code>	唤醒一个处于等待状态的线程
<code>notifyAll()</code>	唤醒同一个对象上所有调用 <code>wait()</code> 方法的线程，优先级高的线程优先调度

注意：都是使用 `Object` 类的方法，都 **只能在同步方法或者同步代码块中使用**

否则会抛出异常 `IllegalMonitorStateException`

解决方式

1. 做一个缓冲区，生产者将生产好的数据放入缓冲区，消费者从缓冲区拿取数据（管程法）
  2. 做一个标志 `flag`（信号灯法）
-

## 管程法

```
1 //生产者，消费者，产品，缓冲区，
2 public class Tube {
3     public static void main(String[] args) {
4
5         Synchro synchro = new Synchro();
6
7         new Productor(synchro).start();
8         new Consumer(synchro).start();
9     }
10 }
11
12 //生产者
13 class Productor extends Thread{
14
15     Synchro container = new Synchro();
16
17     Productor(Synchro container){
18         this.container = container;
19     }
20
21     @Override
22     public void run() {
23         //生产
24         for (int i = 0; i < 20; i++) {
25             container.push(new Chiken(i));
26             System.out.println("生产了第"+i+"只鸡");
27         }
28     }
29 }
30
31 //消费者
32 class Consumer extends Thread{
33
34     Synchro container = new Synchro();
35
36     Consumer(Synchro container){
37         this.container = container;
38     }
39
40     @Override
41     public void run() {
42         for (int i = 0; i < 20; i++) {
43             container.pop();
44             System.out.println("消费了第"+i+"只鸡");
45         }
46     }
47 }
48
49 //产品
50 class Chiken{
51     int id;//产品编号
52
53     public Chiken(int id) {
54         this.id = id;
55     }
56 }
```

```

56     }
57
58
59     //缓冲区
60     class Synchro{
61         //容器大小
62         Chicken[] chickens = new Chicken[10];
63
64         int count = 0; //容器计数器
65
66         //生产者放入产品
67         public synchronized void push(Chicken chicken){
68             //假如容器满了，那么等待消费者消费
69             if (count==chickens.length){
70                 try {
71                     this.wait();
72                 } catch (InterruptedException e) {
73                     e.printStackTrace();
74                 }
75             }
76             //假如容器没有满，那么就需要丢入产品
77             chickens[count] = chicken;
78             count++;
79             //通知消费者消费
80             this.notifyAll();
81         }
82
83
84         //消费者消费产品
85         public synchronized Chicken pop(){
86
87             //假如没有鸡，等待生产者生产
88             if (count==0){
89                 try {
90                     this.wait();
91                 } catch (InterruptedException e) {
92                     e.printStackTrace();
93                 }
94             }
95
96             //假如可以消费
97             count--;
98             Chicken chicken = chickens[count];
99
100             //通知生产者生产
101             this.notifyAll();
102
103             return chicken;
104         }
105     }

```

生产者只管生产

消费者只管消费

而缓冲区内协调生产和消费

## 信号灯法

```
1 //信号灯法一般使用标志位
2 public class Flag {
3     public static void main(String[] args) {
4         TV tv = new TV();
5         new Player(tv).start();
6         new Watcher(tv).start();
7     }
8 }
9
10
11 //生产者
12 class Player extends Thread{
13     TV tv;
14
15     public Player(TV tv){
16         this.tv = tv;
17     }
18
19     @Override
20     public void run() {
21         for (int i = 0; i < 20; i++) {
22             if (i%2==0){
23                 this.tv.play("节目");
24             }else {
25                 this.tv.play("广告");
26             }
27         }
28     }
29 }
30
31 //消费者
32 class Watcher extends Thread{
33     TV tv;
34
35     public Watcher(TV tv){
36         this.tv = tv;
37     }
38
39     @Override
40     public void run() {
41         for (int i = 0; i < 20; i++) {
42             tv.watch();//观看
43         }
44     }
45 }
46
47 //产品
48 class TV{
49     //演员表演，观众等待
50     //观众观看，演员等待
51
52     //表演的节目
53     String voice;
54
55     //标志位
```

```

56     boolean flag = true;
57
58     //表演
59     public synchronized void play(String voice){
60         if (!flag){
61             try {
62                 this.wait();
63             } catch (InterruptedException e) {
64                 e.printStackTrace();
65             }
66         }
67         System.out.println("演员表演了..." + voice);
68         //通知观众观看
69         this.notifyAll(); //通知唤醒
70         this.voice = voice;
71         this.flag = !this.flag;
72     }
73
74     //观看
75     public synchronized void watch(){
76         if (flag){
77             try {
78                 this.wait();
79             } catch (InterruptedException e) {
80                 e.printStackTrace();
81             }
82         }
83         System.out.println("观看了: " + voice);
84
85         this.notifyAll(); //唤醒演员
86         this.flag = !this.flag;
87     }
88 }

```

演员只管表演

观众只管看

TV作为标志位来协调

## 线程池

我们之前经常创建和销毁线程，对性能影响很大

假如我们提前创建好多个线程，放入线程池中，用的时候直接用，用完之后放回线程池

好处：

- 减少了创建新线程的事件，提高响应速度
- 重复利用线程池中的线程，降低资源消耗
- 便于线程管理

- `corePoolSize` : 核心池的大小
- `maximumPoolSize` : 最大线程数
- `keepAliveTime` : 线程没有任务时最多保持多长时间会终止

JDK5提供了真正关于线程池相关的API: `ExecutorService` , `Executors`

- `ExecutorService` : 真正的线程池接口, 常见子类 `ThreadPoolExecutor`
  - `void execute(Runnable command)` : 执行任务/命令, 没有返回值, 一般用来执行 `Runnable`
  - `<T>Future<T> submit(Callable<T> task)` : 执行任务, 有返回值, 一般用来执行 `Callable`
  - `void shutdown()` : 关闭线程池

`Executors` : 工具类, 线程池的工厂类, 用于创建并返回不同类型的线程池

```
1 //测试线程池
2 public class Pool {
3     public static void main(String[] args) {
4         //创建服务, 创建线程池, 10个线程
5         ExecutorService service = Executors.newFixedThreadPool(10);
6
7         //放入Runnable里面的实现类
8         service.execute(new MyThread());
9         service.execute(new MyThread());
10        service.execute(new MyThread());
11        service.execute(new MyThread());
12
13        //关闭链接
14        service.shutdown();
15    }
16
17 }
18
19 class MyThread implements Runnable{
20
21     @Override
22     public void run() {
23         System.out.println(Thread.currentThread().getName());
24     }
25 }
```

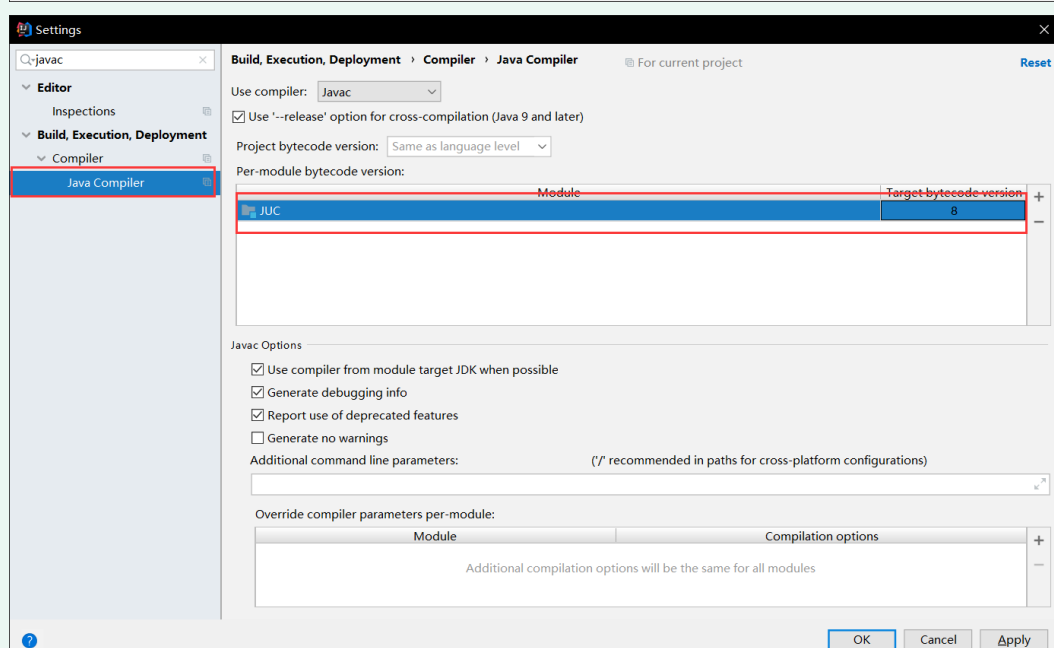
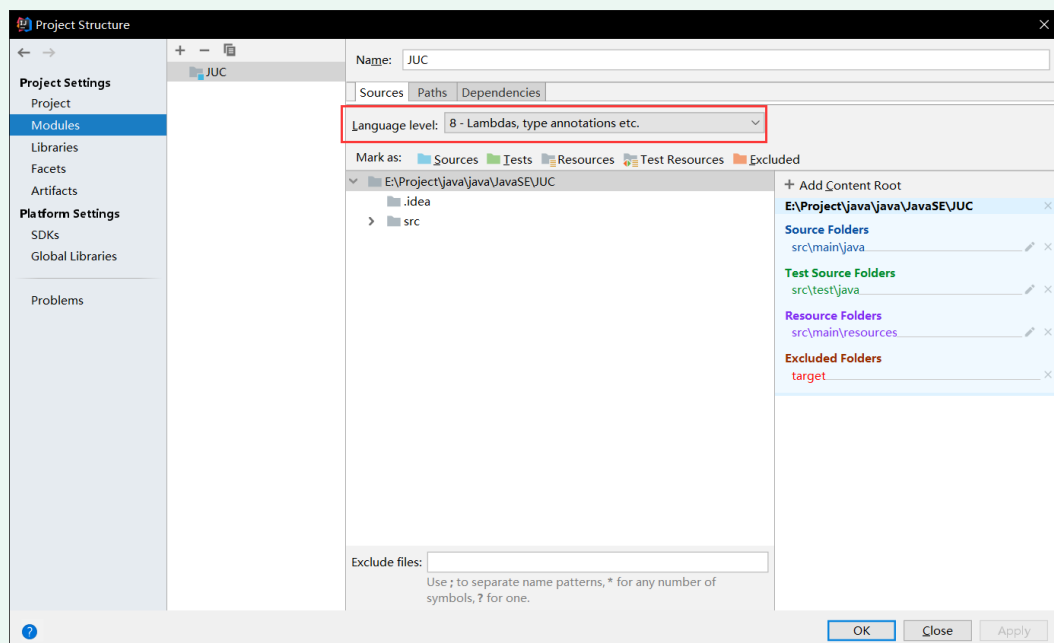
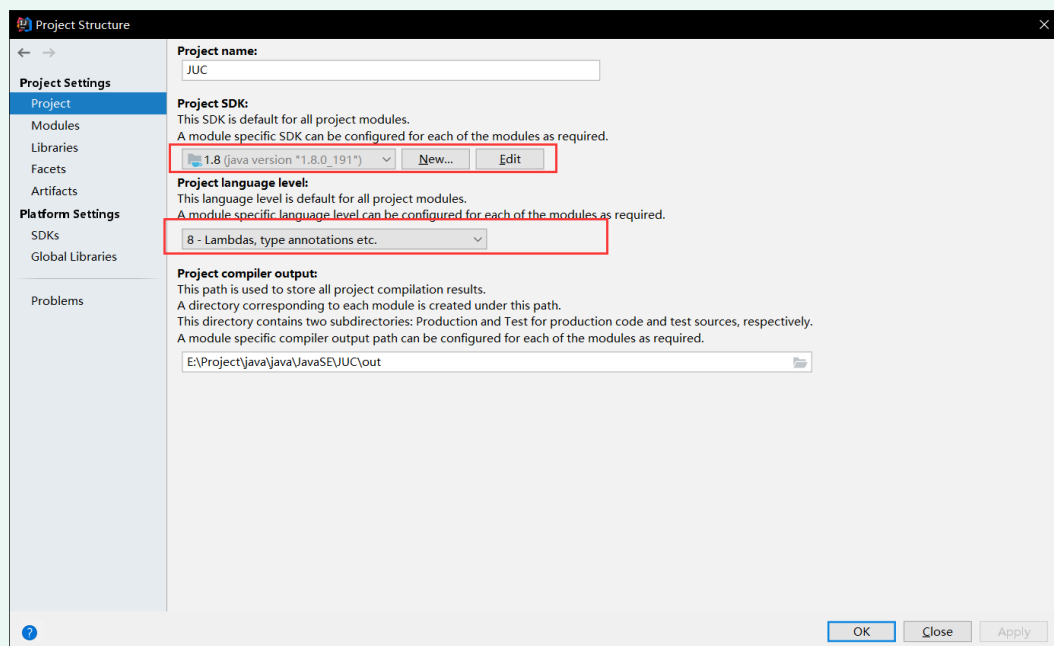
## JUC

### 环境准备

1. 打开IDEA, 新建一个maven项目



## 2. 查看



# 什么是JUC

`java.util.concurrent` 简称JUC，也就是下面的三个包

<code>java.util.concurrent</code>	
<code>java.util.concurrent.atomic</code>	原子性
<code>java.util.concurrent.locks</code>	Lock锁
<code>java.util.function</code>	函数式接口等

JUC

## 为什么要学习JUC

1. JUC面试高频问，所以一定要掌握
2. 之前使用 `Thread`，非常普通的线程类  
`Runnable`，没有返回类型，效率相比于Callable相对较低
3. 之前也学过Lock锁，但是没有深入的系统地学习JUC

面试的时候：单例模式，排序算法，生产者和消费者，死锁

## 线程和进程

一句话说明线程和进程？

1个进程至少包含1个线程

JAVA默认有几个线程？

2个：Main+GC垃圾回收

JAVA真的可以开启线程吗？

JAVA不可以开启线程，如果我们调用底层，我们会发现：

```
1 public synchronized void start() {
2     /**
3      * This method is not invoked for the main method thread or "system"
4      * group threads created/set up by the VM. Any new functionality added
```

```

5      * to this method in the future may have to also be added to the VM.
6      *
7      * A zero status value corresponds to state "NEW".
8      */
9      if (threadStatus != 0)
10         throw new IllegalArgumentException();
11
12     /* Notify the group that this thread is about to be started
13      * so that it can be added to the group's list of threads
14      * and the group's unstarted count can be decremented. */
15     group.add(this);
16
17     boolean started = false;
18     try {
19         start0();
20         started = true;
21     } finally {
22         try {
23             if (!started) {
24                 group.threadStartFailed(this);
25             }
26         } catch (Throwable ignore) {
27             /* do nothing. If start0 threw a Throwable then
28              it will be passed up the call stack */
29         }
30     }
31 }

```

这个start最后调用了一个很诡异的方法 `start0()`，这个其实是调用的本地方法，所以JAVA是没有资格直接操作硬件的

## 并发和并行

- 并发：多个线程操作一个资源
  - CPU一核，但是要模拟出来多线程的操作，只能通过快速的交替来模拟这样的效果
  - 一种假象
- 并行：多个人一起行走
  - 多核CPU下面，多个线程可以同时执行

并发编程的本质：**充分利用CPU的资源**

```

1  public class JUC {
2      public static void main(String[] args) {
3          //获取cpu的核数
4          System.out.println(Runtime.getRuntime().availableProcessors());
5      }
6  }

```

## 线程有几个状态?

```
1  public enum State {
2
3      NEW,          //新生
4
5      RUNNABLE,     //运行
6
7      BLOCKED,      //阻塞
8
9      WAITING,      //等待，永远等待
10
11     TIMED_WAITING, //超时等待，过时不候
12
13     TERMINATED;   //终止
14 }
```

6个

## wait和sleep的区别?

1. 来自不同的类
  - `wait` 来自Object
  - `sleep` 来自Thread
2. 锁的释放
  - `wait` 会释放锁
  - `sleep` 不会释放锁
3. 使用范围
  - `wait` : 必须要在同步代码块中
  - `sleep` : 可以在任何地方使用
4. 是否需要捕获异常
  - `wait` 不用(中断异常不算，只要是线程都会有中断异常 `InterruptedException`)
  - `sleep` 需要捕获异常

## Lock锁

我们先聊传统: `synchronized`

```

1 public class SaleTicketDemo {
2     public static void main(String[] args) {
3         new Thread(new MyThread()).start();
4     }
5 }
6
7 class MyThread implements Runnable{
8
9     @Override
10    public void run() {
11
12    }
13 }

```

以前我们就是这么用的，但是今天开始要推翻

真正的多线程开发中

**线程就是一个单独的资源类，没有任何附属的操作**

```

1 public class SaleTicketDemo {
2     public static void main(String[] args) {
3         //多线程
4
5         //并发：多个线程操作同一个资源类
6         Ticket ticket = new Ticket();
7
8         new Thread()->{
9             for (int i = 0; i < 40; i++) {
10                 ticket.sale();
11             }
12         }, "A").start();
13
14         new Thread()->{
15             for (int i = 0; i < 40; i++) {
16                 ticket.sale();
17             }
18         }, "B").start();
19
20         new Thread()->{
21             for (int i = 0; i < 40; i++) {
22                 ticket.sale();
23             }
24         }, "C").start();
25
26     }
27 }
28
29 //这就是资源类，对这个类进行操作叫做真正的OOP编程
30 //假如在这里集成一个Runnable，这不叫面向对象了，就变成了一个线程类了，而且耦合性高
31 class Ticket{
32     //属性和方法
33     //属性
34     private int number = 50;
35     //方法，使用同步方法：synchronized

```

```

36     public synchronized void sale(){
37         if (number>0){
38             System.out.println(Thread.currentThread().getName() + "-->" + (number--) + ", 剩
余" + number);
39         }
40     }
41 }

```

上面是传统方式简单回顾

## 下面要用JUC

`java.util.concurrent.locks`

我们看到有三个接口，其中有一个Lock接口

- `Interface Lock`，三个实现类
  - `ReentrantLock`：可重用锁（常用）
  - `ReadLock`：读锁
  - `WriteLock`：写锁

我们查看 `ReentrantLock`，我们发现这样一段源码

```

1     public ReentrantLock() {
2         sync = new NonfairSync(); //new一个新的非公平锁
3     }
4
5     public ReentrantLock(boolean fair) { //ReentrantLock的构造参数
6         sync = fair ? new FairSync() : new NonfairSync();
7     }
8     //假如为真，则为公平锁，否则为非公平锁

```

## 公平锁和非公平锁

- 公平锁：十分公平，必须先来后到
- 非公平锁：十分不公平，可以插队（默认）

默认为非公平锁，这是因为：

假如有两个线程，A执行3秒，B执行3小时，B先来的

假如为公平锁，那么A就要等3个小时

我们使用一下Lock锁，来举一个例子：

```

1     import java.util.concurrent.locks.Lock;
2     import java.util.concurrent.locks.ReentrantLock;
3

```

```

4 public class SaleTicketDemo2 {
5     public static void main(String[] args) {
6
7         Ticket ticket = new Ticket();
8
9         new Thread()->{ for (int i = 0; i < 40; i++) ticket.sale(); }, "A").start();
10
11        new Thread()->{ for (int i = 0; i < 40; i++) ticket.sale(); }, "B").start();
12
13        new Thread()->{ for (int i = 0; i < 40; i++) ticket.sale(); }, "C").start();
14
15    }
16 }
17
18 //Lock
19 class Ticket2{
20
21     private int number = 50;
22
23     Lock lock = new ReentrantLock();
24
25     public void sale(){
26         lock.lock();
27
28         try {
29             if (number>0){
30                 System.out.println(Thread.currentThread().getName() + "-->" + (number--) +
31 "， 剩余" + number);
32             }
33         } finally {
34             lock.unlock();//在finally里面解锁
35         }
36     }
37 }

```

## Lock和Synchronized的区别

1. `Synchronized` 是内置的JAVA关键字，而 `Lock` 是一个Java类
2. `Synchronized` 无法判断获取锁的状态， `Lock` 可以判断是否获取到了锁
3. `Synchronized` 会自动释放锁，而 `Lock` 必须手动释放。不释放会导致死锁
4. `Synchronized` 线程1（获得锁）， 线程2（死死地等待）  
`Lock` 就不一定会等待下去：`lock.tryLock()` 尝试获取锁，获取不到就算了
5. `Synchronized` 可重入锁，不可以中断的，非公平锁  
`Lock` ，可重入锁，可以判断锁是否中断，可以自己设置为公平锁或者非公平锁
6. `Synchronized` 适合锁少量的代码同步问题， `Lock` 适合大量的同步代码

## 生产者和消费者问题

## 老版方式

### 线程同步

老版方式: `Synchronized`

```
1  /*
2  线程之间的通信问题：生产者和消费者问题
3  线程交替执行      A      B      操作同一个变量，num
4  A num+1
5  B num-1
6  */
7  public class A {
8      public static void main(String[] args) {
9          Data data = new Data();
10
11         new Thread()->{
12             for (int i = 0; i < 10; i++) {
13                 try {
14                     data.increment();
15                 } catch (InterruptedException e) {
16                     e.printStackTrace();
17                 }
18             }
19         }, "A").start();
20
21         new Thread()->{
22             for (int i = 0; i < 10; i++) {
23                 try {
24                     data.decrement();
25                 } catch (InterruptedException e) {
26                     e.printStackTrace();
27                 }
28             }
29         }, "B").start();
30
31         /*A=>1    B=>0    A=>1    B=>0    A=>1    B=>0
32         A=>1    B=>0    A=>1    B=>0    A=>1    B=>0
33         A=>1    B=>0    A=>1    B=>0    A=>1    B=>0
34         A=>1    B=>0*/
35
36     }
37 }
38
39 //等待，业务，通知：判断是否要进行等待，如果不等待就干活，干完活就通知另一方
40 class Data{//数字，资源类
41
42     private int number = 0;
43
44     //+1，对应生产者
45     public synchronized void increment() throws InterruptedException {
46         if (number!=0){
47             //等待
48             this.wait();
```



```

49     }
50     number++;
51     System.out.println(Thread.currentThread().getName() + "=>" + number);
52     //通知其他线程
53     this.notifyAll();
54 }
55
56 // -1, 对应消费者
57 public synchronized void decrement() throws InterruptedException {
58     if (number==0){
59         //等待
60         this.wait();
61     }
62     number--;
63     System.out.println(Thread.currentThread().getName() + "=>" + number);
64     //通知其他线程
65     this.notifyAll();
66 }
67 }

```

## 虚假唤醒

问题来了：现在只有两个线程，但是我们加到了四个线程，甚至八个线程，结果呢？

我们让A和C加，B和D减

```

1  public class A {
2      public static void main(String[] args) {
3          Data data = new Data();
4
5          new Thread()->{
6              for (int i = 0; i < 5; i++) {
7                  try {
8                      data.increment();
9                  } catch (InterruptedException e) {
10                     e.printStackTrace();
11                 }
12             }
13         }, "A").start();
14
15         new Thread()->{
16             for (int i = 0; i < 5; i++) {
17                 try {
18                     data.decrement();
19                 } catch (InterruptedException e) {
20                     e.printStackTrace();
21                 }
22             }
23         }, "B").start();
24
25         new Thread()->{
26             for (int i = 0; i < 5; i++) {
27                 try {
28                     data.increment();

```

```

29         } catch (InterruptedException e) {
30             e.printStackTrace();
31         }
32     }
33     }, "C").start();
34
35     new Thread()->{
36         for (int i = 0; i < 5; i++) {
37             try {
38                 data.decrement();
39             } catch (InterruptedException e) {
40                 e.printStackTrace();
41             }
42         }
43     }, "D").start();
44
45     /*
46         A=>1    B=>0    C=>1    A=>2    C=>3
47         B=>2    B=>1    B=>0    C=>1    A=>2
48         C=>3    B=>2    C=>3    A=>4    D=>3
49         D=>2    D=>1    D=>0    A=>1    D=>0
50     */
51
52 }
53 }
54
55 //等待，业务，通知：判断是否要进行等待，如果不等待就干活，干完活就通知另一方
56 class Data{//数字，资源类
57
58     private int number = 0;
59
60     //+1，对应生产者
61     public synchronized void increment() throws InterruptedException {
62         if (number!=0){
63             //等待
64             this.wait();
65         }
66         number++;
67         System.out.println(Thread.currentThread().getName() + "=>" + number);
68         //通知其他线程
69         this.notifyAll();
70     }
71
72     //-1，对应消费者
73     public synchronized void decrement() throws InterruptedException {
74         if (number==0){
75             //等待
76             this.wait();
77         }
78         number--;
79         System.out.println(Thread.currentThread().getName() + "=>" + number);
80         //通知其他线程
81         this.notifyAll();
82     }
83 }

```

结果来了，同步凉了

那么为什么会出现这种问题呢？因为用了if判断，有的时候就会出现这种问题，这种问题叫做虚假唤醒

查看jdk文档：[java.lang-->Object-->wait/notify](#)

线程也可以唤醒，而不会被通知，中断或超时，即所谓的“虚假唤醒”，虽然这在实践中很少会发生，但应用程序必须通过测试应该使线程被唤醒的条件来防范，并且如果条件不满足则继续等待。换句话说，等待应该总是出现在循环中，就像这样：

```
synchronized (obj) {  
    while (<condition does not hold>)  
        obj.wait(timeout);  
    ... // Perform action appropriate to condition  
}
```

注意点，这里要防止虚假唤醒

虚假唤醒就是当一个条件满足时，很多线程都被唤醒了，但是只有部分是有用的唤醒，其余的都是无用功

解决方案：条件判断的时候，把 **if** 改为 **while**

```
1  public class A {  
2      public static void main(String[] args) {  
3          Data data = new Data();  
4  
5          new Thread()->{  
6              for (int i = 0; i < 5; i++) {  
7                  try {  
8                      data.increment();  
9                  } catch (InterruptedException e) {  
10                     e.printStackTrace();  
11                 }  
12             }  
13         }, "A").start();  
14  
15         new Thread()->{  
16             for (int i = 0; i < 5; i++) {  
17                 try {  
18                     data.decrement();  
19                 } catch (InterruptedException e) {  
20                     e.printStackTrace();  
21                 }  
22             }  
23         }, "B").start();  
24  
25         new Thread()->{  
26             for (int i = 0; i < 5; i++) {  
27                 try {  
28                     data.increment();  
29                 } catch (InterruptedException e) {  
30                     e.printStackTrace();  
31                 }  
32             }  
33         }, "C").start();  
34  
35         new Thread()->{  
36             for (int i = 0; i < 5; i++) {  
37                 try {  
38                     data.decrement();  
39                 } catch (InterruptedException e) {  
40                     e.printStackTrace();  
41                 }  
42             }  
43         }, "D").start();  
44     }  
45 }
```

```

39         } catch (InterruptedException e) {
40             e.printStackTrace();
41         }
42     }
43     }, "D").start();
44
45
46     }
47 }
48
49
50 class Data{
51
52     private int number = 0;
53
54     public synchronized void increment() throws InterruptedException {
55         while (number!=0){ //这里为了防止虚假唤醒，改为使用while
56
57             this.wait();
58         }
59         number++;
60         System.out.println(Thread.currentThread().getName() + "=>" + number);
61
62         this.notifyAll();
63     }
64
65
66     public synchronized void decrement() throws InterruptedException {
67         while (number==0){ //这里为了防止虚假唤醒，改为使用while
68
69             this.wait();
70         }
71         number--;
72         System.out.println(Thread.currentThread().getName() + "=>" + number);
73
74         this.notifyAll();
75     }
76 }

```

## JUC方式

### 新老三剑客对应

传统三剑客: Synchronized, wait, notify

其中 synchronized 被 Lock 替换了，那么根据逻辑来讲，其余两个也是有替换的

java.util.concurrent.locks

- Condition

- Lock
- ReadWriteLock

那么 **Condition** 就是配套的三剑客之一，对应老版的 **wait** 和 **notify**

**Condition**因素出**Object**监视器方法（**wait**，**notify**和**notifyAll**）成不同的对象，以得到具有多个等待集的每个对象，通过将它们与使用任意的组合的效果Lock个实现。**Lock**替换**synchronized**方法和语句的使用，**Condition**取代了对象监视器方法的使用。

从官方文档我们可以看出来，**Condition**替代了对象的监视器的方法，原来我们使用**wait**和**notify**，现在就要使用**Condition**

```
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock(); try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }

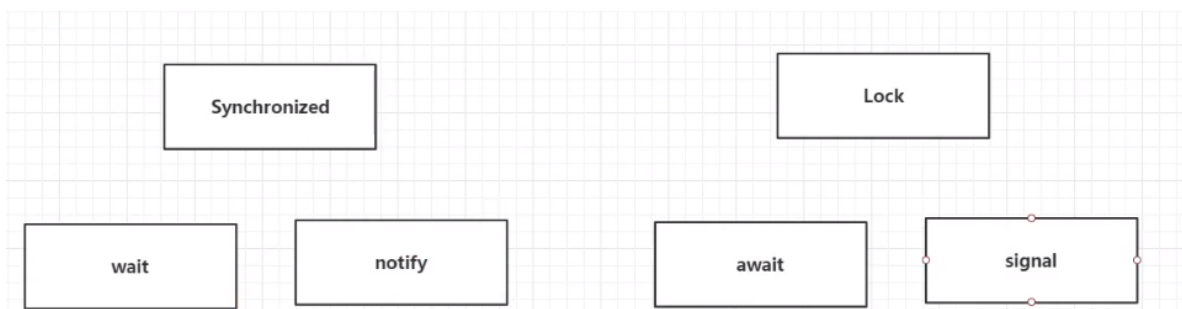
    public Object take() throws InterruptedException {
        lock.lock(); try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally { lock.unlock(); }
    }
}
```

首先new一个监视器对象

等待使用await

通知使用signal

那么对应关系来了：



代码实现一下：

```
1 public class B {
2     public static void main(String[] args) {
3         Data2 data = new Data2();
4
5         new Thread()->{
6             for (int i = 0; i < 5; i++) {
7                 try {
8                     data.increment();
9                 } catch (InterruptedException e) {
10                     e.printStackTrace();
11                 }
12             }
13         }, "A").start();
14
15         new Thread()->{
16             for (int i = 0; i < 5; i++) {
17                 try {
18                     data.decrement();
19                 } catch (InterruptedException e) {
20                     e.printStackTrace();
21                 }
22             }
23         }, "B").start();
24
25         new Thread()->{
26             for (int i = 0; i < 5; i++) {
27                 try {
28                     data.increment();
29                 } catch (InterruptedException e) {
30                     e.printStackTrace();
31                 }
32             }
33         }, "C").start();
34
35         new Thread()->{
36             for (int i = 0; i < 5; i++) {
37                 try {
38                     data.decrement();
39                 } catch (InterruptedException e) {
40                     e.printStackTrace();
41                 }
42             }
43         }, "D").start();
44     }
45 }
46
47 class Data2{
48
49     private int number = 0;
50
51     Lock lock = new ReentrantLock();
52
53     Condition condition = lock.newCondition();//取代了wait和notify
54
55     public void increment() throws InterruptedException {
56         lock.lock();
```

```

57         try {
58             while (number!=0){
59                 //等待
60                 condition.await();
61             }
62             number++;
63             System.out.println(Thread.currentThread().getName() + ">=" + number);
64
65             //通知
66             condition.signalAll();
67         } finally {
68             lock.unlock();
69         }
70     }
71
72
73     public void decrement() throws InterruptedException {
74
75         lock.lock();
76         try {
77             while (number==0){
78                 //等待
79                 condition.await();
80             }
81             number--;
82             System.out.println(Thread.currentThread().getName() + ">=" + number);
83             //通知
84             condition.signalAll();
85         } finally {
86             lock.unlock();
87         }
88     }
89 }

```

上面这个可以了，但是好像没有什么区别

那么这个好像和原来的技术没什么区别，那么我为什么要用新技术？

## 精准通知，有序执行

Condition 新技术可以让线程有序的执行，精准的通知和唤醒线程

```

1  import java.util.concurrent.locks.Condition;
2  import java.util.concurrent.locks.Lock;
3  import java.util.concurrent.locks.ReentrantLock;
4
5  public class C {
6      public static void main(String[] args) {
7          Data3 data = new Data3();
8          new Thread()->{ for (int i = 0; i < 3; i++) data.printA(); }, "A").start();
9          new Thread()->{for (int i = 0; i < 3; i++) data.printB();}, "B").start();
10         new Thread()->{for (int i = 0; i < 3; i++) data.printC();}, "C").start();
11         /*

```

```

12         A=>A    B=>B    C=>C
13         A=>A    B=>B    C=>C
14         A=>A    B=>B    C=>C
15     */
16 }
17 }
18
19 //资源类使用Lock锁，要求A执行完调用B，B执行完调用C，C执行完调用A
20 class Data3{
21     private Lock lock = new ReentrantLock();
22
23     //因为一个同步监视器只能监视一个线程，所以我们使用三个监视器，然后通过监视器来判断我们来唤醒什么
24     private Condition condition1 = lock.newCondition();
25     private Condition condition2 = lock.newCondition();
26     private Condition condition3 = lock.newCondition();
27
28     //我们让number为1的时候执行A，number为2执行B，number为3执行C
29     private int number = 1;
30
31     public void printA(){
32         lock.lock();
33         try {
34             //判断是否等待
35             while (number!=1){
36                 //等待
37                 condition1.await();
38             }
39             //执行
40             System.out.println(Thread.currentThread().getName()+"=>A");
41             //通知，唤醒B
42             number=2;
43             condition2.signal();
44         } catch (InterruptedException e) {
45             e.printStackTrace();
46         } finally {
47             lock.unlock();
48         }
49     }
50
51     public void printB(){
52         lock.lock();
53         try {
54             //判断是否等待
55             while (number!=2){
56                 //等待
57                 condition2.await();
58             }
59             //执行
60             System.out.println(Thread.currentThread().getName()+"=>B");
61             //通知，唤醒C
62             number=3;
63             condition3.signal();
64         } catch (InterruptedException e) {
65             e.printStackTrace();
66         } finally {
67             lock.unlock();
68         }
69     }

```



```

70
71     public void printC(){
72         lock.lock();
73         try {
74             //判断是否等待
75             while (number!=3){
76                 //等待
77                 condition3.await();
78             }
79             //执行
80             System.out.println(Thread.currentThread().getName()+"=>C");
81             //通知，唤醒B
82             number=1;
83             condition1.signal();
84         } catch (InterruptedException e) {
85             e.printStackTrace();
86         } finally {
87             lock.unlock();
88         }
89     }
90 }

```

## 八锁现象彻底理解锁

如何判断锁的谁？，什么是锁？

**锁只会锁两个东西：对象，Class模板**

先发短信还是先打电话？

八锁其实是八个问题，我们以先发短信还是先打电话作为这个问题，在不同的情况下分析八次

1. 标准情况下，两个线程先打印发短信还是先打印打电话

```

1     import java.util.concurrent.TimeUnit;
2
3     public class Lock1 {
4         public static void main(String[] args) {
5             Phone phone = new Phone();
6
7             new Thread(()->{
8                 phone.sendMessage();
9             }, "A").start();
10
11         try {
12             //公司中使用这个工具类来实现休眠，这里休眠一秒
13             TimeUnit.SECONDS.sleep(1);
14         } catch (InterruptedException e) {
15             e.printStackTrace();

```

```

16         }
17
18         new Thread()->{
19             phone.call();
20         }.start();
21
22     }
23 }
24
25
26 class Phone{
27
28     public synchronized void sendMessage(){
29         System.out.println("发短信");
30     }
31
32     public synchronized void call(){
33         System.out.println("打电话");
34     }
35
36
37 }

```

这里应该是和先发短信，然后再打电话

原因不是在于谁先调用的，而是在于谁先获得的 **锁**

**这里的锁是锁住的对象**，而两者都是同一个对象的方法

所以为了同步，谁先获得锁，谁就先执行

中间加了那一秒的延迟其实是个陷阱

## 2. 发短信延迟4秒，谁先执行？

```

1  import java.util.concurrent.TimeUnit;
2
3  public class Lock2 {
4      public static void main(String[] args) {
5          Phone2 phone = new Phone2();
6
7          new Thread()->{
8              phone.sendMessage();
9          }.start();
10
11
12          new Thread()->{
13              phone.call();
14          }.start();
15
16      }
17  }
18
19
20 class Phone2{

```

```

21
22     public synchronized void sendMessage(){
23         try {
24             TimeUnit.SECONDS.sleep(4);
25         } catch (InterruptedException e) {
26             e.printStackTrace();
27         }
28         System.out.println("发短信");
29     }
30
31     public synchronized void call(){
32         System.out.println("打电话");
33     }
34
35
36 }

```

还是先发短信，然后再打电话

这个问题和第一个问题没有本质区别，都是考验 **锁** 的问题

**锁是锁的对象**，所以两者的锁都是同一把

谁先获得锁，谁就先执行

中间停顿的4秒是陷阱

### 3. 改变一个为普通方法之后，谁先执行？

```

1  import java.util.concurrent.TimeUnit;
2
3  public class Lock3 {
4      public static void main(String[] args) {
5          Phone3 phone = new Phone3();
6
7          new Thread()->{
8              phone.sendMessage();
9          }, "A").start();
10
11
12         new Thread()->{
13             phone.call();
14         }, "B").start();
15     }
16 }
17
18
19
20 class Phone3{
21
22     public synchronized void sendMessage(){
23         try {
24             TimeUnit.SECONDS.sleep(4);
25         } catch (InterruptedException e) {
26             e.printStackTrace();

```

```

27     }
28     System.out.println("发短信");
29 }
30
31 public void call(){
32     System.out.println("打电话");
33 }
34 }

```

这个答案是先打电话

**因为打电话不是一个同步方法，所以不用获取锁**

这次的延迟不是一个陷阱

#### 4. 两个对象，两个方法，谁先执行

```

1  public class Lock4 {
2      public static void main(String[] args) {
3          Phone4 phone1 = new Phone4();
4          Phone4 phone2 = new Phone4();
5
6          new Thread()->{
7              phone1.sendMessage();
8          }, "A").start();
9
10
11         new Thread()->{
12             phone2.call();
13         }, "B").start();
14
15     }
16 }
17
18
19 class Phone4{
20
21     public synchronized void sendMessage(){
22         try {
23             TimeUnit.SECONDS.sleep(4);
24         } catch (InterruptedException e) {
25             e.printStackTrace();
26         }
27         System.out.println("发短信");
28     }
29
30     public synchronized void call(){
31         System.out.println("打电话");
32     }
33
34
35 }

```

这里应该是先打电话

本质上的问题还是锁

**这个锁是锁的对象**，所以两个对象的锁并不是同一把

但是因为发短信延迟了四秒钟，所以先打电话

#### 5. 一个对象，两个静态同步方法，谁先执行？

```
1  import java.util.concurrent.TimeUnit;
2
3  public class Lock5 {
4      public static void main(String[] args) {
5          Phone5 phone = new Phone5();
6
7          new Thread()->{
8              phone.sendMessage();
9          }, "A").start();
10
11
12         new Thread()->{
13             phone.call();
14         }, "B").start();
15     }
16 }
17
18
19
20 class Phone5{
21
22     public static synchronized void sendMessage(){
23         try {
24             TimeUnit.SECONDS.sleep(4);
25         } catch (InterruptedException e) {
26             e.printStackTrace();
27         }
28         System.out.println("发短信");
29     }
30
31     public static synchronized void call(){
32         System.out.println("打电话");
33     }
34
35
36 }
```

这个题的答案应该是先执行发短信

但是如果是按照之前根据同一把锁，锁的对象来回答的话，那么答案不正确

因为加入了 `static` 关键字，表明了这是一个静态加载，说明了类一加载就有了

**所以这个锁的是Class模板，就是 `Phone5.class`**，而不是锁的对象

打电话和发短信虽然争抢的是同一把锁，但是他们争抢的是Class的锁而不是对象的锁

## 6. 两个对象，两个静态同步方法，谁先执行？

```
1  import java.util.concurrent.TimeUnit;
2
3  public class Lock6 {
4      public static void main(String[] args) {
5          Phone6 phone1 = new Phone6();
6          Phone6 phone2 = new Phone6();
7
8          new Thread()->{
9              phone1.sendMessage();
10             }, "A").start();
11
12
13         new Thread()->{
14             phone2.call();
15             }, "B").start();
16
17     }
18 }
19
20
21 class Phone6{
22
23     public static synchronized void sendMessage(){
24         try {
25             TimeUnit.SECONDS.sleep(4);
26         } catch (InterruptedException e) {
27             e.printStackTrace();
28         }
29         System.out.println("发短信");
30     }
31
32     public static synchronized void call(){
33         System.out.println("打电话");
34     }
35
36
37 }
```

还是先发短信再打电话

**因为他们两个争抢的是Class模板的锁，而不是对象的锁**

所以有几个对象都无所谓

谁先抢到Class的锁，谁就先执行

## 7. 一个静态同步方法，一个普通同步方法是，一个对象，谁先执行？

```
1  import java.util.concurrent.TimeUnit;
2
3  public class Lock8 {
```

```

4      public static void main(String[] args) {
5          Phone7 phone = new Phone7();
6
7          new Thread()->{
8              phone.sendMessage();
9          }, "A").start();
10
11
12         new Thread()->{
13             phone.call();
14         }, "B").start();
15
16     }
17 }
18
19 class Phone7{
20
21     public static synchronized void sendMessage(){
22         try {
23             TimeUnit.SECONDS.sleep(4);
24         } catch (InterruptedException e) {
25             e.printStackTrace();
26         }
27         System.out.println("发短信");
28     }
29
30     public synchronized void call(){
31         System.out.println("打电话");
32     }
33
34
35 }

```

应当是先打电话，在发短信

因为这个问题他们争抢的不是一把锁

发短信抢的是Class模板锁，打电话争抢的是对象锁

又因为发短信延迟4秒执行

所以先执行打电话

8. 两个对象，一个静态同步，一个普通同步，谁先执行？

```

1      import java.util.concurrent.TimeUnit;
2
3      public class Lock8 {
4          public static void main(String[] args) {
5              Phone7 phone1 = new Phone7();
6              Phone7 phone2 = new Phone7();
7
8              new Thread()->{

```

```

9         phone1.sendMessage();
10    }, "A").start();
11
12
13    new Thread()->{
14        phone2.call();
15    }, "B").start();
16
17    }
18 }
19
20
21 class Phone8{
22
23     public static synchronized void sendMessage(){
24         try {
25             TimeUnit.SECONDS.sleep(4);
26         } catch (InterruptedException e) {
27             e.printStackTrace();
28         }
29         System.out.println("发短信");
30     }
31
32     public synchronized void call(){
33         System.out.println("打电话");
34     }
35
36
37 }

```

先打电话，然后发短信  
 因为两个争抢的不是一把锁  
 又因为发短信慢四秒  
 所以先打电话

## 集合类不安全

### List不安全的解决

一个普通的List集合



```

1 public class ListTest {
2     public static void main(String[] args) {
3         List<String> list = new ArrayList<>();
4
5         //报错了: java.util.ConcurrentModificationException
6         for (int i = 1; i <= 10; i++) {
7             new Thread()->{
8                 list.add(UUID.randomUUID().toString().substring(0,5));
9                 System.out.println(list);
10            },String.valueOf(i)).start();
11        }
12    }
13 }

```

这个是不安全的，结果就是报错了，报错：`java.util.ConcurrentModificationException`  
并发修改异常

并发下ArrayList是不安全的

## 解决方案（普通层面）

1. 使用 `Vector` : `List<String> list = new Vector<>();`

但是 `Vector` 虽然可行，但是面试不会给高分的，因为Vector比 `ArrayList` 出现的时间早  
那么为什么JDK要出现一个线程不安全的 `ArrayList` 来替代Vector呢？  
肯定比Vector更加高效，那么现在我回退了版本使用旧技术，简直是自寻死路

2. 通过工具类转换: `Collections`

```

1 public class ListTest {
2     public static void main(String[] args) {
3         List<String> list = Collections.synchronizedList(new ArrayList<>());
4
5         //报错了: java.util.ConcurrentModificationException
6         for (int i = 1; i <= 10; i++) {
7             new Thread()->{
8                 list.add(UUID.randomUUID().toString().substring(0,5));
9                 System.out.println(list);
10            },String.valueOf(i)).start();
11        }
12    }
13 }

```

既然 `ArrayList` 不安全，那么我们让它变得安全不就行了么？  
集合的老大 `Collections`，可以帮助我们解决这个问题，进行同步  
转变为 `Synchronized`  
但是其实这个答案和Vector并没有区别，因为Vector也是使用了 `Synchronized`

## 解决方案 (JUC)

打开jdk文档，翻到 `java.util.concurrent` 包下，找到对应的class: `CopyOnWriteArrayList`

这个就是JUC下面的解决方案

```
1 import java.util.concurrent.CopyOnWriteArrayList;
2
3 public class ListTest {
4     public static void main(String[] args) {
5         List<String> list = new CopyOnWriteArrayList<>();
6
7         //报错了: java.util.ConcurrentModificationException
8         for (int i = 1; i <= 10; i++) {
9             new Thread(()->{
10                 list.add(UUID.randomUUID().toString().substring(0,5));
11                 System.out.println(list);
12             },String.valueOf(i)).start();
13         }
14     }
15 }
```

**CopyOnWrite**：写入时复制，简称 **COW**，是计算机程序设计领域的一种优化策略

有多个线程调用的时候，比如调用list，list是唯一的，在读取的时候是固定的，但是写入的时候不能让他们同时写，因为A写完之后可能B就把A给覆盖了，那么现在就有：在写入的时候复制一份，写完之后交给调用者，这样就在写入的时候避免覆盖，造成数据问题

这里涉及到一个读写分离的思想

走底层，发现

```
1 private transient volatile Object[] array;
```

这里又看不懂了，但是下面会讲

那么 `CopyOnWriteArrayList` 比 `Vector` 牛逼在哪里

我们找一下 `Vector` 的源码

```
1 public synchronized boolean add(E e) {
2     modCount++;
3     ensureCapacityHelper(elementCount + 1);
4     elementData[elementCount++] = e;
5     return true;
6 }
```

我们不可避免的发现了Vector使用了 `synchronized`

这在意料之中

我们再查看 `CopyOnWriteArrayList`

```

1      public boolean add(E e) {
2          final ReentrantLock lock = this.lock;
3          lock.lock();
4          try {
5              Object[] elements = getArray();
6              int len = elements.length;
7              Object[] newElements = Arrays.copyOf(elements, len + 1);    //拿过来的时候复制一份
8              newElements[len] = e;
9              setArray(newElements);    //还回去
10             return true;
11         } finally {
12             lock.unlock();
13         }
14     }

```

我们看到它使用了Lock锁

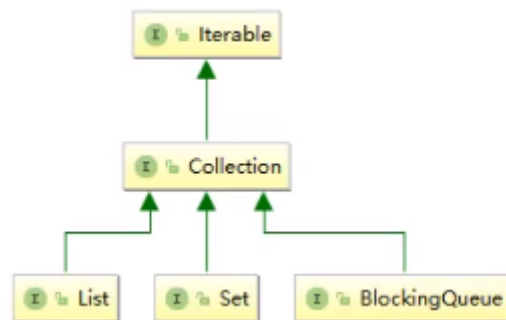
其中在拿过来的时候去复制一份

还回去的时候交给他

牛逼在就在这里

---

## Set不安全的解决



首先做一个铺垫：List, Set, BlockingQueue

我们知道有List, Set, 但是 **BlockingQueue(阻塞队列)** 是第一次听, 和List,Set同级  
先知道有这个东西就好了

---

Set其实和List没有区别

```

1 public class SetTest {
2     public static void main(String[] args) {
3         Set<String> set = new HashSet<>();
4
5         for (int i = 1; i <= 10; i++) {
6             new Thread()->{
7                 set.add(UUID.randomUUID().toString().substring(0,5));
8                 set.forEach(System.out::println);
9             }.start();
10        }
11    }
12 }

```

Exception in thread "Thread-3" java.util.ConcurrentModificationException

并发修改异常

## 解决方案 (类似List)

1. Set没有类似于Vector这样顶替的方式，所以直接上工具类

```

1 public class SetTest {
2     public static void main(String[] args) {
3         // Set<String> set = new HashSet<>();
4         Set<String> set = Collections.synchronizedSet(new HashSet<>());
5
6         for (int i = 1; i <= 10; i++) {
7             new Thread()->{
8                 set.add(UUID.randomUUID().toString().substring(0,5));
9                 set.forEach(System.out::println);
10            }.start();
11        }
12    }
13 }

```

通过工具类转为 Synchronized

2. JUC的解决方法

```

1 public class SetTest {
2     public static void main(String[] args) {
3         // Set<String> set = new HashSet<>();
4         // Set<String> set = Collections.synchronizedSet(new HashSet<>());
5         Set<String> set = new CopyOnWriteArraySet();
6
7         for (int i = 1; i <= 10; i++) {
8             new Thread()->{
9                 set.add(UUID.randomUUID().toString().substring(0,5));
10                set.forEach(System.out::println);
11            }.start();
12        }
13    }

```

## Set的底层是什么

查看HashSet的第层：

```
1 public HashSet() {  
2     map = new HashMap<>();  
3 }
```

清楚了吧，HashSet的第层其实是new了一个HashMap

但是还没完

```
1 public boolean add(E e) {  
2     return map.put(e, PRESENT) == null;  
3 }
```

这次看清楚了吧，HashSet的add方法其实就是 `hashmap` 的put进去的值的key，也就是键而这个 `PRESENT`

```
1 private static final Object PRESENT = new Object();
```

随便来了一个Object

**所以HashSet的本质就是HashMap的key，因为key是无法重复的**

所以JDK官方也是够坑爹的

## HashMap不安全的解决

首先有两个问题：

```
1 //map是这样用的吗？默认等价于什么？  
2 Map<String, String> map = new HashMap<>();
```

1. 工作中不用HashMap
2. 默认等价于 `new HashMap(16, 0.75);`

看到这里我们需要先看一看源码

```

1 public HashMap(int initialCapacity, float loadFactor) {
2     if (initialCapacity < 0)
3         throw new IllegalArgumentException("Illegal initial capacity: " +
4                                           initialCapacity);
5     if (initialCapacity > MAXIMUM_CAPACITY)
6         initialCapacity = MAXIMUM_CAPACITY;
7     if (loadFactor <= 0 || Float.isNaN(loadFactor))
8         throw new IllegalArgumentException("Illegal load factor: " +
9                                           loadFactor);
10    this.loadFactor = loadFactor;
11    this.threshold = tableSizeFor(initialCapacity);
12 }

```

```

1 public HashMap(int initialCapacity) {
2     this(initialCapacity, DEFAULT_LOAD_FACTOR);
3 }

```

```

1 public HashMap() {
2     this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
3 }

```

这三个重载其中有两个重载的变量需要重视：

`loadFactor`：加载因子，默认 `加载因子0.75`

`initialCapacity`：初始容量，默认 `16`，这个16是位运算的16

```

1 static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

```

重点是这个不安全

```

1 public class MapTest {
2     public static void main(String[] args) {
3         //map是这样用的吗？默认等价于什么？
4         Map<String, String> map = new HashMap<>();
5         for (int i = 0; i < 10; i++) {
6             new Thread()->{
7                 map.put(Thread.currentThread().getName(),
8                     UUID.randomUUID().toString().substring(0,5));
9                 System.out.println(map);
10            },String.valueOf(i)).start();
11        }
12    }

```

`java.util.ConcurrentModificationException`

并发修改异常

## 解决方案

### 1. Collections

```
1 public class MapTest {
2     public static void main(String[] args) {
3         //map是这样用的吗? 默认等价于什么?
4         Map<String, String> map = Collections.synchronizedMap(new HashMap<>());
5         for (int i = 0; i < 10; i++) {
6             new Thread()->{
7                 map.put(Thread.currentThread().getName(),
8                     UUID.randomUUID().toString().substring(0,5));
9                 System.out.println(map);
10            },String.valueOf(i)).start();
11        }
12    }
13 }
```

### 2. JUC中没有CopyOnWritexxx, 注意, 名字变了, 叫做 ConcurrentHashMap

ConcurrentHashMap 就是这个  
ConcurrentHashMap.KeySetView  
ConcurrentLinkedDeque 队列  
ConcurrentLinkedQueue  
ConcurrentSkipListMap ListMap和  
ConcurrentSkipListSet ListSet  
CopyOnWriteArrayList 普通的  
CopyOnWriteArraySet  
CountDownLatch  
CountedCompleter  
CyclicBarrier  
DelayQueue

```
1 public class MapTest {
2     public static void main(String[] args) {
3         //map是这样用的吗? 默认等价于什么?
4         Map<String, String> map = new ConcurrentHashMap();
5         for (int i = 0; i < 10; i++) {
6             new Thread()->{
7                 map.put(Thread.currentThread().getName(),
8                     UUID.randomUUID().toString().substring(0,5));
9                 System.out.println(map);
10            },String.valueOf(i)).start();
11        }
12    }
13 }
```

对于 ConcurrentHashMap, 看源码

```

1     public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {
2         if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)
3             //假如负载因子>0或者初始化的长度<0或者并发级别<=0, 那么就抛出非法异常
4             throw new IllegalArgumentException();
5         if (initialCapacity < concurrencyLevel)    //假如初始化长度<并发级别
6             initialCapacity = concurrencyLevel;    // 令初始化长度=并发级别
7         long size = (long)(1.0 + (long)initialCapacity / loadFactor);
8         int cap = (size >= (long)MAXIMUM_CAPACITY) ?
9             MAXIMUM_CAPACITY : tableSizeFor((int)size);
10        this.sizeCtl = cap;
11    }

```

## Callable

### 什么是Callable

`java.util.concurrent .Interface Callable<V>`

- `Callable` 接口类似于 `Runnable`

然而, `Runnable` 不返回结果, 也不能抛出被检查的异常

1. `Callable` 可以有返回值
2. `Callable` 可以抛出异常
3. `Callable` 方法不同, `Thread`中继承的叫做 `run()`, `Callable`中继承的叫做 `call()`

```

1     @FunctionalInterface
2     public interface Callable<V> {
3         V call() throws Exception;
4     }

```

泛型就是返回值

现在有一个问题:

`Thread`没法直接启动`Callable`

因为`Thread`只能直接和`Runnable`挂上钩, 但是没法和`Callable`挂上钩, 从源码中可以看出

```

1     public Thread(Runnable target) {
2         init(null, target, "Thread-" + nextThreadNum(), 0);
3     }

```

那么我们就知道了, 因为`Callable`没法直接和`Thread`挂上钩, 但是如果我们想办法, 把`Callable`和`Runnable`挂上钩, 那么不就间接的实现目的了么?



java也是这么想的

`java.lang.Runnable` 中有

All Known Subinterfaces:

`RunnableFuture <V>`, `RunnableScheduledFuture <V>` 父类的接口

所有已知实现类:

`AsyncBoxView.ChildState`, `ForkJoinWorkerThread`, `FutureTask`, `RenderableImageProducer`, `SwingWorker`, `Thread`, `TimerTask`

我们查看FutureTask这个Runnable的实现类

#### Constructor and Description

**FutureTask(Callable<V> callable)**

创建一个 `FutureTask`，它将在运行时执行给定的 `Callable`。

**FutureTask(Runnable runnable, V result)**

创建一个 `FutureTask`，将在运行时执行给定的 `Runnable`，并安排 `get`将在成功完成后返回给定的结果。

我们看到了，构造方法可以和Runnable挂上关系，也可以和Callable挂上关系

所以结果清晰明了：

Callable勾搭上了FutureTask，而FutureTask是Runnable的实现类，而Runnable和Thread有关系

这就完美的实现了Callable<————>Thread

```
new Thread(()->{new FutureTask<V>(Callable)}).start()
```

所以Thread就可以启动Callable了

```
1  import java.util.concurrent.Callable;
2  import java.util.concurrent.ExecutionException;
3  import java.util.concurrent.FutureTask;
4
5  public class CallableTest {
6      public static void main(String[] args) throws ExecutionException, InterruptedException {
7          FutureTask futureTask = new FutureTask(new MyThread());
8          new Thread(futureTask, "A").start();
9
10         Integer i = (Integer)futureTask.get();
11
12         System.out.println(i);
13     }
14 }
15
16 class MyThread implements Callable<Integer> {
17     @Override
18     public Integer call() throws Exception {
19         System.out.println("call()");
20         return 123123;
21     }
22 }
```

注意点：

`futureTask.get()` 这个方法可能会造成阻塞，结果可能会阻塞

所以一般我们有两种解决方案：

1. 放到最后一行
2. 异步通信

注意点，假如我们启用两个线程：

```
1 public class CallableTest {
2     public static void main(String[] args) throws ExecutionException,
        InterruptedException {
3         FutureTask futureTask = new FutureTask(new MyThread());
4         new Thread(futureTask, "A").start();
5         new Thread(futureTask, "B").start();
6
7         Integer i = (Integer)futureTask.get();
8
9         System.out.println(i);
10    }
11 }
12
13 class MyThread implements Callable<Integer> {
14     @Override
15     public Integer call() throws Exception {
16         System.out.println("call()");
17         return 123123;
18     }
19 }
```

结果却是：

```
call()
123123
```

这个说明结果有缓存，这是一个小坑

## 常用的辅助类

`java.util.concurrent`

### CountDownLatch

减法计数器，两个方法

- `countDownLatch.countDown()`：每当有一个线程执行一次，减法计数器减一（计数）
- `countDownLatch.await()`：等待计数器归零，才能执行这行代码之后的代码（拦截器）

```
1 import java.util.concurrent.CountDownLatch;
```

```

2
3 public class CountdownLatchDemo {
4     public static void main(String[] args) {
5         //假设我们设置计数器为6
6         CountdownLatch countDownLatch = new CountdownLatch(6);
7
8         for (int i = 0; i < 6; i++) {
9             final int temp = i;
10            new Thread()->{
11                System.out.println("这是第"+temp+"个");    //这里要注意了，因为lambda最终是new了一个接口，所以不能直接操作i
12                countDownLatch.countDown();//计数器减一
13            },String.valueOf(i)).start();
14        }
15        try {
16            countDownLatch.await(); //这里是等待计数器归零之后才可以执行后面的代码
17        } catch (InterruptedException e) {
18            e.printStackTrace();
19        }
20        System.out.println("计数器归零");
21    }
22 }

```

## CyclicBarrier

加法计数器，两个方法

两个构造函数，其中一个构造函数的作用是指定好计数器的个数，并且指定在达到计数器的个数的时候执行的线程

- `cyclicBarrier.await()`：等待线程执行完毕之后执行后面的方法

```

1 import java.util.concurrent.BrokenBarrierException;
2 import java.util.concurrent.CyclicBarrier;
3
4 public class CyclicBarrierDemo {
5     public static void main(String[] args) {
6         CyclicBarrier cyclicBarrier = new CyclicBarrier(7,()->{
7             System.out.println("执行后来的线程");//当七个线程都执行的时候，那么就开始执行这个线程
8         });
9
10        for (int i = 0; i < 7; i++) {
11            int temp = i+1;
12            new Thread()->{
13                System.out.println("这里是第"+temp+"个线程");
14                try {
15                    cyclicBarrier.await();//等待7个线程
16                } catch (InterruptedException e) {
17                    e.printStackTrace();
18                } catch (BrokenBarrierException e) {
19                    e.printStackTrace();
20                }
21            }).start();
22        }
23    }
24 }

```

假如它要传递多余七个线程，但是其实达不到这个数量，那么程序就会卡在这里

## Semaphore

信号量，让线程可以等待，让线程在有限的情况下有秩序的执行，限流可以使用  
就像抢车位，假如有6辆车，三个车位，那就需要抢车位

- `acquire()`：得到，假设得不到那么等待信号量被释放位置
- `release()`：释放当前的信号量，唤醒等待的线程

```
1  import java.util.concurrent.Semaphore;
2  import java.util.concurrent.TimeUnit;
3
4  public class SemaphoreDemo {
5      public static void main(String[] args) {
6          //信号量，这里可以理解为车位
7          Semaphore semaphore = new Semaphore(3);
8
9          //有6个线程需要争抢
10         for (int i = 1; i <= 6; i++) {
11             new Thread()->{
12                 try {
13                     semaphore.acquire();//得到
14                     System.out.println(Thread.currentThread().getName()+"得到");
15                     TimeUnit.SECONDS.sleep(2);//模拟线程操作，休息两秒钟
16                 } catch (InterruptedException e) {
17                     e.printStackTrace();
18                 } finally {
19                     semaphore.release();//释放
20                 }
21             },String.valueOf(i)).start();
22         }
23     }
24 }
```

## ReadWriteLock

读写锁， `java.util.concurrent`

为了提高工作的效率，读写锁规定：读取可以由多个线程读取，写则只能有一个线程去写

- `writeLock()`：写锁，也称独占锁
  - `lock()`
  - `unlock()`
- `readLock()`：读锁，也称共享锁

- o lock()
- o unlock()

```
1  import java.util.HashMap;
2  import java.util.Map;
3  import java.util.concurrent.locks.ReadWriteLock;
4  import java.util.concurrent.locks.ReentrantReadWriteLock;
5
6  public class ReadWriteLockDemo {
7      public static void main(String[] args) {
8
9          MyCache cache = new MyCache();
10
11         for (int i = 0; i < 5; i++) {
12             final int temp = i+1;
13             new Thread()->{
14                 cache.put(temp+"", temp+"");
15             },String.valueOf(temp)).start();
16         }
17
18         for (int i = 0; i < 5; i++) {
19             final int temp = i+1;
20             new Thread()->{
21                 cache.get(temp+"");
22             },String.valueOf(temp)).start();
23         }
24     }
25 }
26
27
28 //自定义缓存
29 class MyCache{
30     private volatile Map<String,Object> map = new HashMap<>();
31
32     //这个是读写锁，可以实现更加细粒度的操作，就是读取可以多线程操作，写入只能一个线程
33     private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
34
35     //写
36     public void put(String key,Object value){
37         readWriteLock.writeLock().lock();//加了一把写锁，更加细粒度的划分了读锁和写锁
38         try {
39             System.out.println(Thread.currentThread().getName()+"写入"+key);
40             map.put(key,value);
41             System.out.println(Thread.currentThread().getName()+"写入ok");
42         } catch (Exception e) {
43             e.printStackTrace();
44         } finally {
45             readWriteLock.writeLock().unlock();//解锁
46         }
47     }
48
49     //读
50     public void get(String key){
51         readWriteLock.readLock().lock();    //读取锁
52         try {
53             System.out.println(Thread.currentThread().getName()+"读取"+key);
54             Object o = map.get(key);
```

```

55         System.out.println(Thread.currentThread().getName()+"读取ok");
56     } catch (Exception e) {
57         e.printStackTrace();
58     } finally {
59         readWriteLock.readLock().unlock();//写入锁
60     }
61 }
62 }

```

## 阻塞队列

BlockingQueue: `java.util.concurrent`

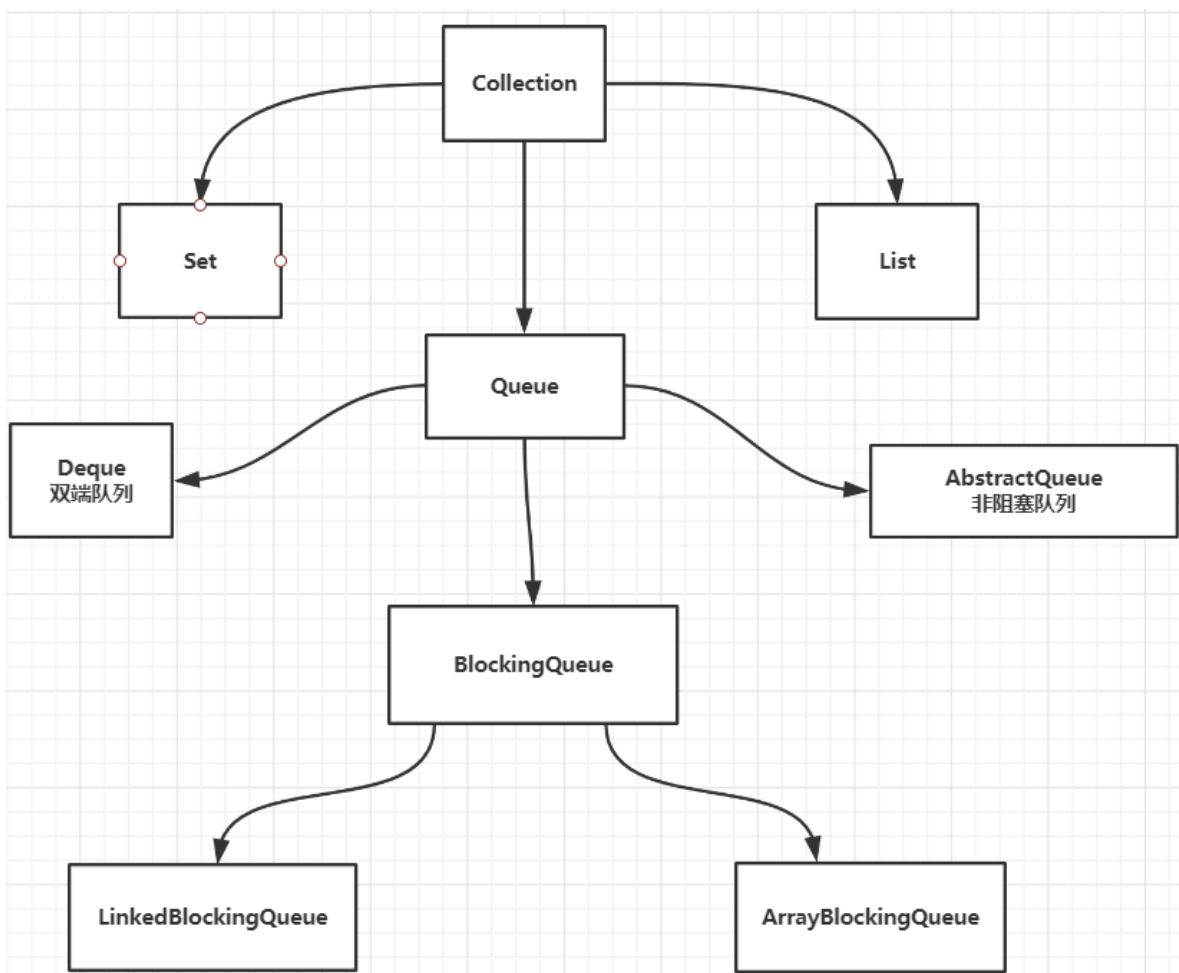
阻塞队列由两个词构成

- 阻塞
- 队列

队列分为：写，读

如果队列是满的，要写东西必须阻塞等待取出

如果队列是空的，要取东西必须阻塞等待生产



阻塞队列并不是新东西，他也是Collection的子类

阻塞队列属于队列家族，属于庞大分支的一小部分

我们什么时候使用BlockingQueue

线程池，多线程并发处理

学会使用队列

添加，移除

四组API

- 1. 抛出异常
- 2. 不抛出异常
- 3. 阻塞等待
- 4. 超时等待

方式	有返回值，抛出异常	有返回值，不抛出异常	阻塞等待	超时等待
添加	add	offer	put	offer（重载）
移除	remove	poll	take	poll（重载）
判断队列首	element	peek	null	null

- 抛出异常

```
1  import java.util.concurrent.ArrayBlockingQueue;
2
3  public class BlockQueueDemo {
4      public static void main(String[] args) {
5          test1();
6          test2();
7      }
8
9      //抛出异常
10     public static void test1(){
11
12         //队列的大小设为3
13         ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue(3);
14         System.out.println(blockingQueue.add("A")); //true
15         System.out.println(blockingQueue.add("B")); //true
16         System.out.println(blockingQueue.add("C")); //true
17
18         //查看队首
19         System.out.println(blockingQueue.element()); //A
20
21         //因为队列大小为3，所以添加第四个，这里我们需要的API是抛出异常的API：
22         java.lang.IllegalStateException: Queue full
23         System.out.println(blockingQueue.add("D"));
24     }
25
26     //抛出异常
```

```

27     public static void test2(){
28
29         //队列的大小设为3
30         ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue(3);
31
32         //直接取出元素，因为什么也没有，所以不能取出，这里我们使用的是抛出异常的API:
        java.util.NoSuchElementException
33         System.out.println(blockingQueue.remove());
34     }
35 }

```

- 有返回值，不抛出异常

```

1     public class BlockQueueDemo {
2         public static void main(String[] args) {
3             test3();
4             test4();
5         }
6
7         //返回值
8         public static void test3(){
9
10            //队列的大小设为3
11            ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue(3);
12
13
14            System.out.println(blockingQueue.offer("A")); //true
15            System.out.println(blockingQueue.offer("B")); //true
16            System.out.println(blockingQueue.offer("C")); //true
17
18            System.out.println(blockingQueue.peek()); //查看队首: A
19
20            //不抛出异常，有返回值
21            System.out.println(blockingQueue.offer("D")); //false
22        }
23
24        //有返回值
25        public static void test4(){
26
27            //队列的大小设为3
28            ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue(3);
29
30            System.out.println(blockingQueue.poll()); //null
31        }
32    }

```

- 一直阻塞等待（put进去没有返回值）

```

1     public class BlockQueueDemo {
2         public static void main(String[] args) throws InterruptedException {
3             // test5();
4             test6();
5         }
6
7         //阻塞（一直等待）
8         public static void test5() throws InterruptedException {
9
10            //队列的大小设为3

```



```

11     ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue(3);
12
13
14     blockingQueue.put("A");
15     blockingQueue.put("B");
16     blockingQueue.put("C");
17
18     //一直等待
19     blockingQueue.put("D");
20 }
21
22 //阻塞（一直等待）
23 public static void test6() throws InterruptedException {
24
25     //队列的大小设为3
26     ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue(3);
27
28     System.out.println(blockingQueue.take()); //一直等待
29 }
30 }

```

- 超时退出

```

1 public class BlockQueueDemo {
2     public static void main(String[] args) throws InterruptedException {
3         test7();
4         test8();
5     }
6
7     //阻塞（超时等待）
8     public static void test7() throws InterruptedException {
9
10        //队列的大小设为3
11        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue(3);
12
13
14        blockingQueue.offer("A");
15        blockingQueue.offer("B");
16        blockingQueue.offer("C");
17
18        //超时等待
19        blockingQueue.offer("D", 2, TimeUnit.SECONDS);
20    }
21
22    //阻塞（超时等待）
23    public static void test8() throws InterruptedException {
24
25        //队列的大小设为3
26        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue(3);
27
28        System.out.println(blockingQueue.poll(2, TimeUnit.SECONDS)); //null
29    }
30 }
31

```

SynchronousQueue: `java.util.concurrent`

只能存储一个元素，进去一个元素必须等待取出来之后才能往里面存另一个元素

所以同步队列和其他的阻塞队列是不一样的

```
1  import java.util.concurrent.BlockingQueue;
2  import java.util.concurrent.SynchronousQueue;
3  import java.util.concurrent.TimeUnit;
4
5  public class SynchronousQueueDemo {
6      public static void main(String[] args) {
7          //同步队列
8          BlockingQueue<String> synchronousQueue = new SynchronousQueue<String>();
9
10         new Thread(()->{
11             try {
12                 System.out.println(Thread.currentThread().getName()+"-->put 1");
13                 synchronousQueue.put("1");
14
15                 System.out.println(Thread.currentThread().getName()+"-->put 2");
16                 synchronousQueue.put("2");
17
18                 System.out.println(Thread.currentThread().getName()+"-->put 3");
19                 synchronousQueue.put("3");
20             } catch (InterruptedException e) {
21                 e.printStackTrace();
22             }
23         }).start();
24
25         new Thread(()->{
26             try {
27                 //为了保证上面的插入完了，这里等待三秒
28                 TimeUnit.SECONDS.sleep(3);
29                 System.out.println(Thread.currentThread().getName()+"-->get 1");
30                 synchronousQueue.take();
31
32                 TimeUnit.SECONDS.sleep(3);
33                 System.out.println(Thread.currentThread().getName()+"-->get 2");
34                 synchronousQueue.take();
35
36                 TimeUnit.SECONDS.sleep(3);
37                 System.out.println(Thread.currentThread().getName()+"-->get 3");
38                 synchronousQueue.take();
39             } catch (InterruptedException e) {
40                 e.printStackTrace();
41             }
42         }).start();
43     }
44 }
45 /*
46 Thread-0-->put 1
47 Thread-1-->get 1
48 Thread-0-->put 2
49 Thread-1-->get 2
50 Thread-0-->put 3
51 Thread-1-->get 3
```

## 线程池

### 线程池的东西：三大方法，七大参数，四种拒绝策略

池化技术：事先准备好一些资源，有人要用，就来我这里拿，用完之后还给我

程序的运行，本质：占用系统的资源！

优化资源的使用！=> 池化技术

线程池、连接池、内存池、对象池等都是池

线程池的好处：

1. 降低资源的消耗
2. 提高响应的速度
3. 方便管理

### 线程复用，可以控制最大并发数，管理线程

4. 【强制】线程池不允许使用 `Executors` 去创建，而是通过 `ThreadPoolExecutor` 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：`Executors` 返回的线程池对象的弊端如下：

- 1) `FixedThreadPool` 和 `SingleThreadPool`：
 

约为21亿  
 允许的请求队列长度为 `Integer.MAX_VALUE`，可能会堆积大量的请求，从而导致 OOM。
- 2) `CachedThreadPool` 和 `ScheduledThreadPool`：
 

允许的创建线程数量为 `Integer.MAX_VALUE`，可能会创建大量的线程，从而导致 OOM。



## 三大方法

```
1  import java.util.concurrent.Executors;
2
3  public class PoolDemo {
4      public static void main(String[] args) {
5          Executors.newSingleThreadExecutor();//线程池只有单个线程
6          Executors.newFixedThreadPool(5);//创建一个固定的线程池大小
7          Executors.newCachedThreadPool();//可伸缩的线程池，遇强则强
8      }
9  }
```

使用线程池的使用案例

```
1  import java.util.concurrent.ExecutorService;
2  import java.util.concurrent.Executors;
```

```

3
4 public class PoolDemo {
5     public static void main(String[] args) {
6         //      ExecutorService threadPool = Executors.newSingleThreadExecutor();//线程池只有单个线程
7         //      ExecutorService threadPool = Executors.newFixedThreadPool(5);//创建一个固定的线程池
        大小
8         ExecutorService threadPool = Executors.newCachedThreadPool();//可伸缩的线程池，遇强则强
9
10        //new Thread(()->{}).start();这个是以前我们创建的线程，但是从今天开始，要使用线程池来创建
11        //execute()里面也是一个Runnable
12        try {
13            for (int i = 0; i < 10; i++)    threadPool.execute(()->{
14                >System.out.println(Thread.currentThread().getName()+"    ok");
15            } catch (Exception e) {
16                e.printStackTrace();
17            } finally {
18                threadPool.shutdown();//注意，最后要关闭线程池
19            }
20        }
21    }
22 }

```

从上面的例子中分析：

第一个单个线程中，10个语句只有一条线程执行

第二个线程中，10个语句最多有第5条线程执行，也就是说最多到了5条线程同时执行

第三个线程中，10个语句最多有第10条线程执行，也就是说10条线程同时执行了，所以只要你CPU可以撑住，想多少就多少

## 七大参数

源码分析：

```

1 public static ExecutorService newSingleThreadExecutor() {
2     return new FinalizableDelegatedExecutorService
3         (new ThreadPoolExecutor(1, 1,
4             0L, TimeUnit.MILLISECONDS,
5             new LinkedBlockingQueue<Runnable>()));
6 }
7
8 public static ExecutorService newFixedThreadPool(int nThreads) {
9     return new ThreadPoolExecutor(nThreads, nThreads,
10        0L, TimeUnit.MILLISECONDS,
11        new LinkedBlockingQueue<Runnable>());
12 }
13
14 public static ExecutorService newCachedThreadPool() {
15     return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
16        60L, TimeUnit.SECONDS,
17        new SynchronousQueue<Runnable>());
18 }

```

观察这三个线程，我们发现他们三个本质都是调用的 `new ThreadPoolExecutor`

## 七大参数：

```
1      public ThreadPoolExecutor(int corePoolSize, //核心线程池大小
2                                int maximumPoolSize, //最大核心线程池大小
3                                long keepAliveTime, //没有人调用则超时释放
4                                TimeUnit unit, //超时单位
5                                BlockingQueue<Runnable> workQueue, //阻塞队列
6                                ThreadFactory threadFactory, //线程工厂，创建线程的，一般不用动
7                                RejectedExecutionHandler handler) { //拒绝策略
8
9      if (corePoolSize < 0 ||
10         maximumPoolSize <= 0 ||
11         maximumPoolSize < corePoolSize ||
12         keepAliveTime < 0)
13         throw new IllegalArgumentException();
14     if (workQueue == null || threadFactory == null || handler == null)
15         throw new NullPointerException();
16     this.acc = System.getSecurityManager() == null ?
17         null :
18         AccessController.getContext();
19     this.corePoolSize = corePoolSize;
20     this.maximumPoolSize = maximumPoolSize;
21     this.workQueue = workQueue;
22     this.keepAliveTime = unit.toNanos(keepAliveTime);
23     this.threadFactory = threadFactory;
24     this.handler = handler;
25 }
```

我们找到了这样的源码，这里有七个参数

看完了这个我们再去分析一下上面的三个线程池

```
1      public static ExecutorService newSingleThreadExecutor() {
2          return new FinalizableDelegatedExecutorService
3              (new ThreadPoolExecutor(1, 1, //核心线程为1，共有1个线程
4                                      0L, TimeUnit.MILLISECONDS,
5                                      new LinkedBlockingQueue<Runnable>()));
6      }
7
8      public static ExecutorService newFixedThreadPool(int nThreads) {
9          return new ThreadPoolExecutor(nThreads, nThreads, //核心线程和线程总数都由外界传过来，比
10                                     如我们刚才的5
11                                     0L, TimeUnit.MILLISECONDS,
12                                     new LinkedBlockingQueue<Runnable>());
13      }
14
15     public static ExecutorService newCachedThreadPool() {
16         return new ThreadPoolExecutor(0, Integer.MAX_VALUE, //核心线程为0，但是最大线程可以有
17                                     Integer.MAX_VALUE: 21亿
18                                     60L, TimeUnit.SECONDS,
19                                     new SynchronousQueue<Runnable>());
20     }
```

如果我们使用缓存机制的线程池，电脑一定会爆出OOM

这样我们就知道了为什么阿里巴巴强制不允许使用 `Executors` 来创建，而是通过 `ThreadPoolExecutor`

因为可能会因为内存过大导致溢出

4. 【强制】线程池不允许使用 `Executors` 去创建，而是通过 `ThreadPoolExecutor` 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：`Executors` 返回的线程池对象的弊端如下：

- 1) `FixedThreadPool` 和 `SingleThreadPool`:  
允许的请求队列长度为 `Integer.MAX_VALUE`，可能会堆积大量的请求，从而导致 OOM。  
→ 约为21亿
- 2) `CachedThreadPool` 和 `ScheduledThreadPool`:  
允许的创建线程数量为 `Integer.MAX_VALUE`，可能会创建大量的线程，从而导致 OOM。

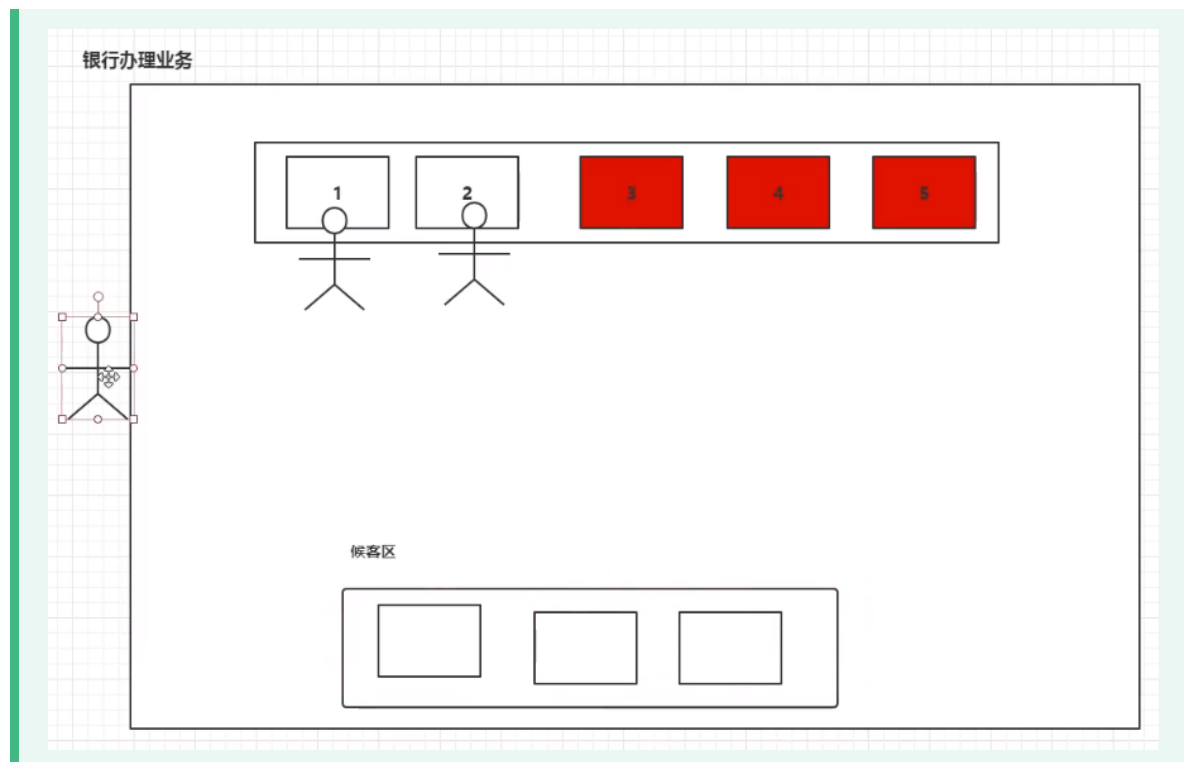
JVM  
↑

## 线程池的形象解释

在生活中，我们会遇到银行排队的情况

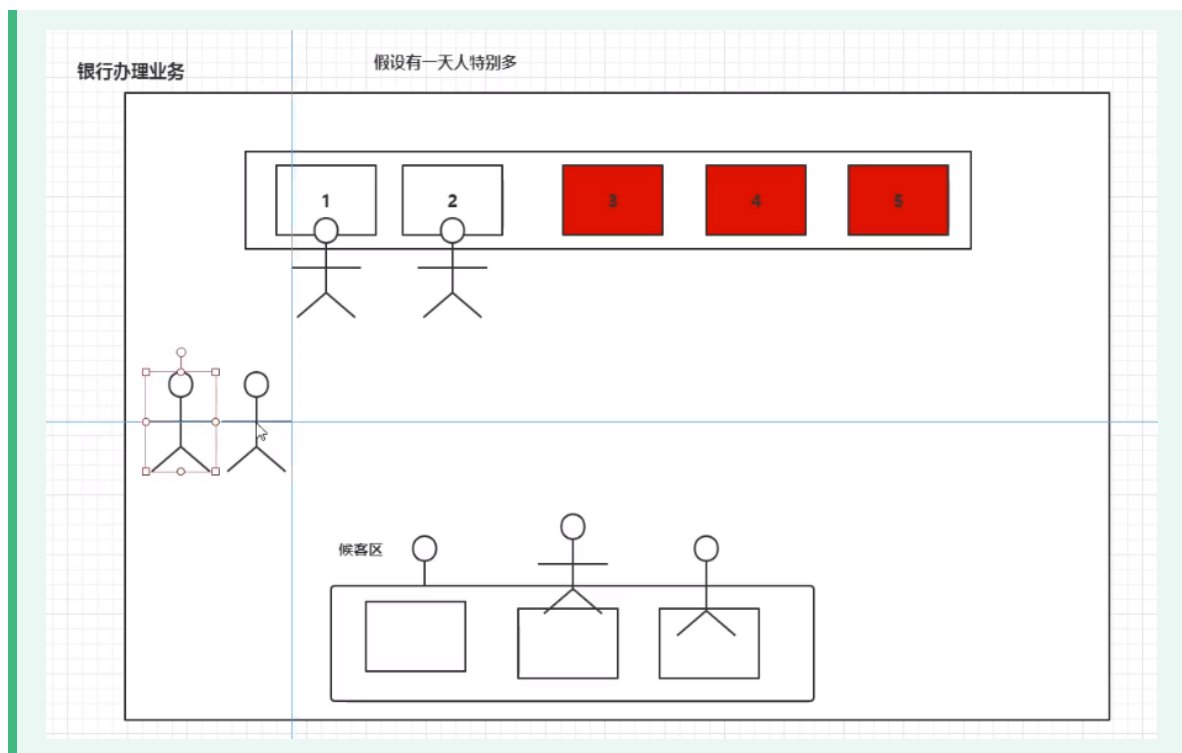
假设现在银行中有5个窗口，候客区中有三个座位

平常人不多的时候银行窗口并不是全都开着，比如只开着两个窗口，剩下三个窗口



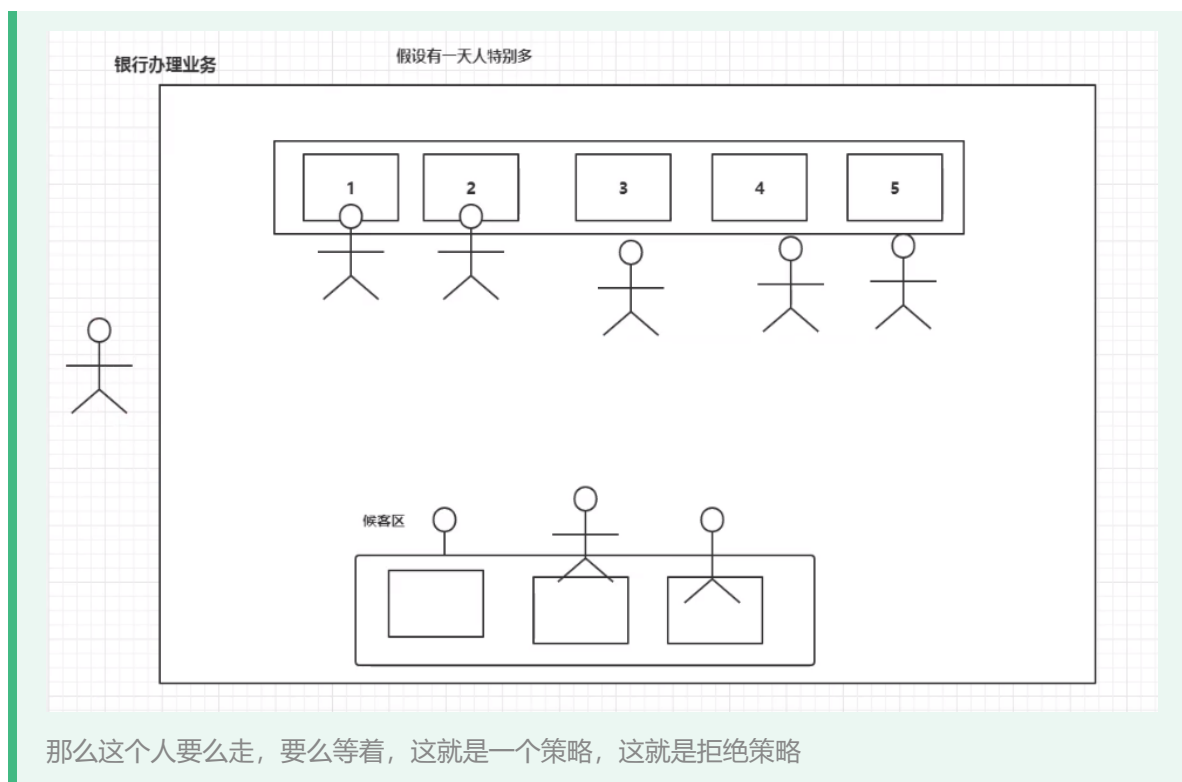
如上图所示，假如现在开着的窗口已经满了，又来了一个人，那么就只能到候客区中坐下等待银行办理

但是假如银行中开着的窗口满了，而候客区也满了，但是还在进入



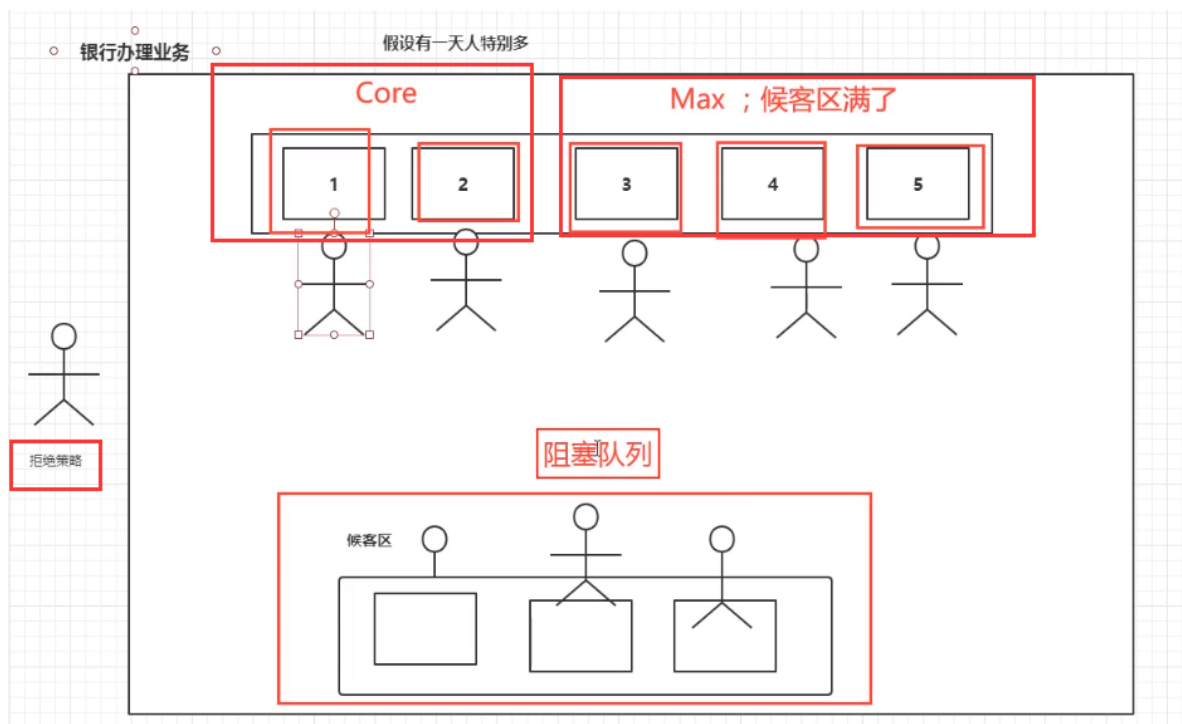
这个时候另外三个候选窗口就要开启了

但是假如全部的窗口都满了，候客区也满了，又进来一个人



那么这个人要么走，要么等着，这就是一个策略，这就是拒绝策略

综上，银行全部的窗口就是总线程，一直开着的窗口就是核心线程，候客区就是阻塞队列，全部的位置满了之后对再进来的人的处理策略就是拒绝策略



处理到了最后，所有人，或者只剩下两个主线程的还没有都处理完了，等待了一段时间之后，还没有更多的人来处理，那么剩下的3，4，5这三个银行端口就关闭，等待下一次开启

这个部分就叫做超时等待

## 四种拒绝策略

名称	拒绝策略
AbortPolicy	抛出异常
CallerRunsPolicy	回去执行，比如main线程到了新线程，那么就回到main线程执行
DiscardOldestPolicy	尝试去和最早的线程竞争，竞争失败任务则不执行
DiscardPolicy	不会抛出异常，但是任务不执行

## 不使用Executors创建线程池的方法

```
1 import java.util.concurrent.*;
2
3 public class PoolDemo {
4     public static void main(String[] args) {
5         ExecutorService threadPool
6             = new ThreadPoolExecutor(
7             2, //核心线程池
8             5, //最大线程池
9             3, //超时等待时间
```



```

10         TimeUnit.SECONDS, //时间单位
11         new LinkedBlockingDeque<>(3), //阻塞队列
12         Executors.defaultThreadFactory(), //默认的创建工厂，一般不改变
13         new ThreadPoolExecutor.AbortPolicy() //拒绝策略，抛出异常
14     );
15 }
16 }

```

**线程<=2**：只使用核心线程

**2<线程<=5**：使用核心线程+阻塞队列，2个线程运行，3个线程等待

**5<线程<=8**：使用全部线程+阻塞队列，5个线程运行，3个线程等待，最大承载

**线程>8**：使用全部线程+阻塞队列+抛出异常，5个线程运行，3个线程等待，其余线程执行拒绝策略

## 小结和扩展

调优：线程池的最大大小如何去设置

### 最大线程到底该如何定义

- CPU密集型
- IO密集型

#### CPU密集型的写法：

我们一开始的时候说到的并发和并行中，说到了cpu的核数，假如是12核的cpu，就意味着并发量为12  
12条线程同时执行

那么定义为CPU密集型的时候，这个时候我们将最大线程定义为12，就可以确保电脑的利用率达到最好

我们可以使用：`Runtime.getRuntime().availableProcessors()` 获取CPU的核数

#### IO密集型的写法：

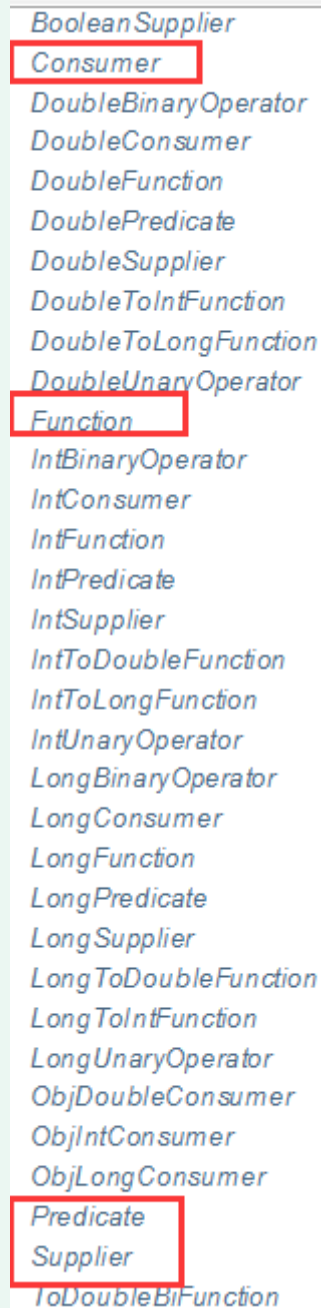
一个程序有15个大型任务，IO十分占用资源，那么我们至少留15个线程执行任务，在这种情况下只要比它大就可以了

一般情况下设置为IO任务的两倍。这里是30

## 四大函数式接口

### 函数式接口

四大函数式接口（基本）：`java.util.function`



BooleanSupplier  
Consumer  
DoubleBinaryOperator  
DoubleConsumer  
DoubleFunction  
DoublePredicate  
DoubleSupplier  
DoubleToIntFunction  
DoubleToLongFunction  
DoubleUnaryOperator  
Function  
IntBinaryOperator  
IntConsumer  
IntFunction  
IntPredicate  
IntSupplier  
IntToDoubleFunction  
IntToLongFunction  
IntUnaryOperator  
LongBinaryOperator  
LongConsumer  
LongFunction  
LongPredicate  
LongSupplier  
LongToDoubleFunction  
LongToIntFunction  
LongUnaryOperator  
ObjDoubleConsumer  
ObjIntConsumer  
ObjLongConsumer  
Predicate  
Supplier  
ToDoubleBiFunction

其余的都是组合式接口

`@FunctionalInterface` : 函数式接口

只有一个方法的接口，比如Runnable，比如 `foreach()` 的参数就是一个消费者类型的函数式接口

```
1 public void forEach(Consumer<? super E> action) {}
```

```
1 @FunctionalInterface
2 public interface Consumer<T> {}
```

函数式接口在java中超级多，在新版本的第层大量应用

- Function：函数型接口

```

1  @FunctionalInterface
2  public interface Function<T, R> {
3      R apply(T t);
4
5      default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
6          Objects.requireNonNull(before);
7          return (V v) -> apply(before.apply(v));
8      }
9
10     default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
11         Objects.requireNonNull(after);
12         return (T t) -> after.apply(apply(t));
13     }
14
15     static <T> Function<T, T> identity() {
16         return t -> t;
17     }
18 }

```

使用:

```

1      Function<String, String> function = new Function<String, String>() {
2          @Override
3          public String apply(String s) {
4              return s;
5          }
6      };
7      //简化后: Function<String, String> function = str-> {return str;};

```

函数式接口: 有一个输入参数, 有一个输出

- Predicate: 断定型接口

```

1  @FunctionalInterface
2  public interface Predicate<T> {
3
4      boolean test(T t);
5
6      default Predicate<T> and(Predicate<? super T> other) {
7          Objects.requireNonNull(other);
8          return (t) -> test(t) && other.test(t);
9      }
10
11     default Predicate<T> negate() {
12         return (t) -> !test(t);
13     }
14
15     default Predicate<T> or(Predicate<? super T> other) {
16         Objects.requireNonNull(other);
17         return (t) -> test(t) || other.test(t);
18     }
19
20     static <T> Predicate<T> isEqual(Object targetRef) {
21         return (null == targetRef)
22             ? Objects::isNull
23             : object -> targetRef.equals(object);

```

```
24     }
25 }
```

使用:

```
1     Predicate<String> predicate = new Predicate<String>() {
2         @Override
3         public boolean test(String s) {
4             return s.isEmpty();
5         }
6     };
7 //简化后: Predicate<String> predicate = s -> {return s.isEmpty();};
```

断定型接口，有一个输入参数，返回值只能是布尔值

- Consumer：消费型接口

```
1     @FunctionalInterface
2     public interface Consumer<T> {
3
4         void accept(T t);
5
6         default Consumer<T> andThen(Consumer<? super T> after) {
7             Objects.requireNonNull(after);
8             return (T t) -> { accept(t); after.accept(t); };
9         }
10 }
```

使用:

```
1     Consumer<String> consumer = new Consumer<String>() {
2         @Override
3         public void accept(String s) {
4             System.out.println(s);
5         }
6     };
7 //Consumer<String> consumer = s-> System.out.println(s);
```

消费型接口：只有输入，没有返回值

- Supplier：供给型接口

```
1     @FunctionalInterface
2     public interface Supplier<T> {
3
4         T get();
5     }
```

使用:

```

1      Supplier<String> supplier = new Supplier<String>() {
2          @Override
3          public String get() {
4              return "str";
5          }
6      };
7      //Supplier<String> supplier =()->{return "str";};

```

供给型接口：只有输出，没有输入

## Stream流计算

### 什么是Stream流计算

存储+计算

`java.util.Stream`

我们的存储使用集合，MySQL等等

计算现在交给流来做

```

1      @Data
2      @AllArgsConstructor
3      @NoArgsConstructor
4      public class User {
5          private int id;
6          private String name;
7          private int age;
8      }

```

```

1      /*
2      * 题目要求：一分钟之内完成此题，而且只能用一行代码实现！
3      * 现在有5个用户，筛选：
4      * 1. ID必须为偶数
5      * 2. 年龄必须大于23岁
6      * 3. 用户名字转为大写
7      * 4. 用户名字倒着排序
8      * 5. 只输出一个用户
9      */
10     public class Test {
11         public static void main(String[] args) {
12             User u1 = new User(1, "a", 21);
13             User u2 = new User(2, "b", 22);
14             User u3 = new User(3, "c", 23);
15             User u4 = new User(4, "d", 24);
16             User u5 = new User(5, "e", 25);
17
18             List<User> list = Arrays.asList(u1, u2, u3, u4, u5);
19         }
20     }

```

最后我们输出的应当是D

## 使用流进行计算

查看jdk: `java.util.stream`

- `stream()` : 将集合转换为流
- `filter()` : 过滤, 传入断定型接口, 假如为true则保留, 为false过滤掉
- `map()` : 转换, 传入函数型接口, 可以自由地对参数进行转换
- `sorted()` : 排序, 默认为正序, 可传入函数式接口使用 `compare()` 进行倒序排序
- `limit()` : 分页
- `foreach()` : 可以传入消费型函数式接口, 一般用于遍历

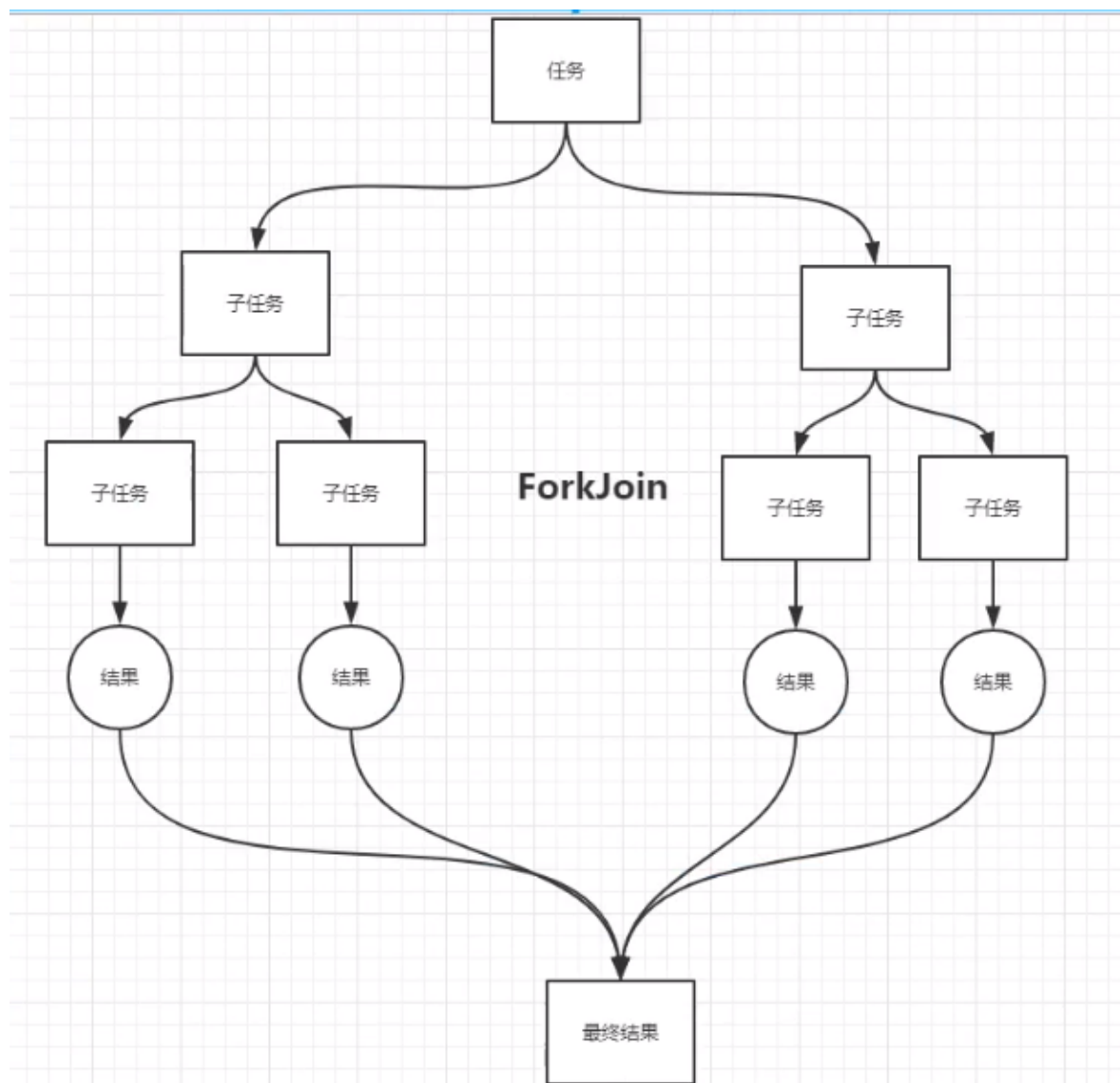
```
1  /*
2  * 题目要求: 一分钟之内完成此题, 而且只能用一行代码实现!
3  * 现在有5个用户, 筛选:
4  * 1. ID必须为偶数
5  * 2. 年龄必须大于23岁
6  * 3. 用户名字转为大写
7  * 4. 用户名字字母倒着排序
8  * 5. 只输出一个用户
9  */
10 public class Test {
11     public static void main(String[] args) {
12         User u1 = new User(1, "a", 21);
13         User u2 = new User(2, "b", 22);
14         User u3 = new User(3, "c", 23);
15         User u4 = new User(4, "d", 24);
16         User u5 = new User(5, "e", 25);
17
18         List<User> list = Arrays.asList(u1, u2, u3, u4, u5);
19         list.stream()
20             .filter(u->{return u.getId()%2==0;})    //选取偶数
21             .filter(u->{return u.getAge()>23;})    //年龄必须大于23岁
22             .map(u->{return u.getName().toUpperCase();})    //用户名转大写
23             .sorted((uu1,uu2)->{return uu2.compareTo(uu1);})    //倒序排序
24             .forEach(System.out::println);
25     }
26 }
```

## ForkJoin

### 什么是ForkJoin

中文名字叫做分支合并

ForkJoin在JDK7的时候出现, 并行执行任务, 提高效率



什么时候使用ForkJoin

**大数据量的时候**

ForkJoin特点：工作窃取

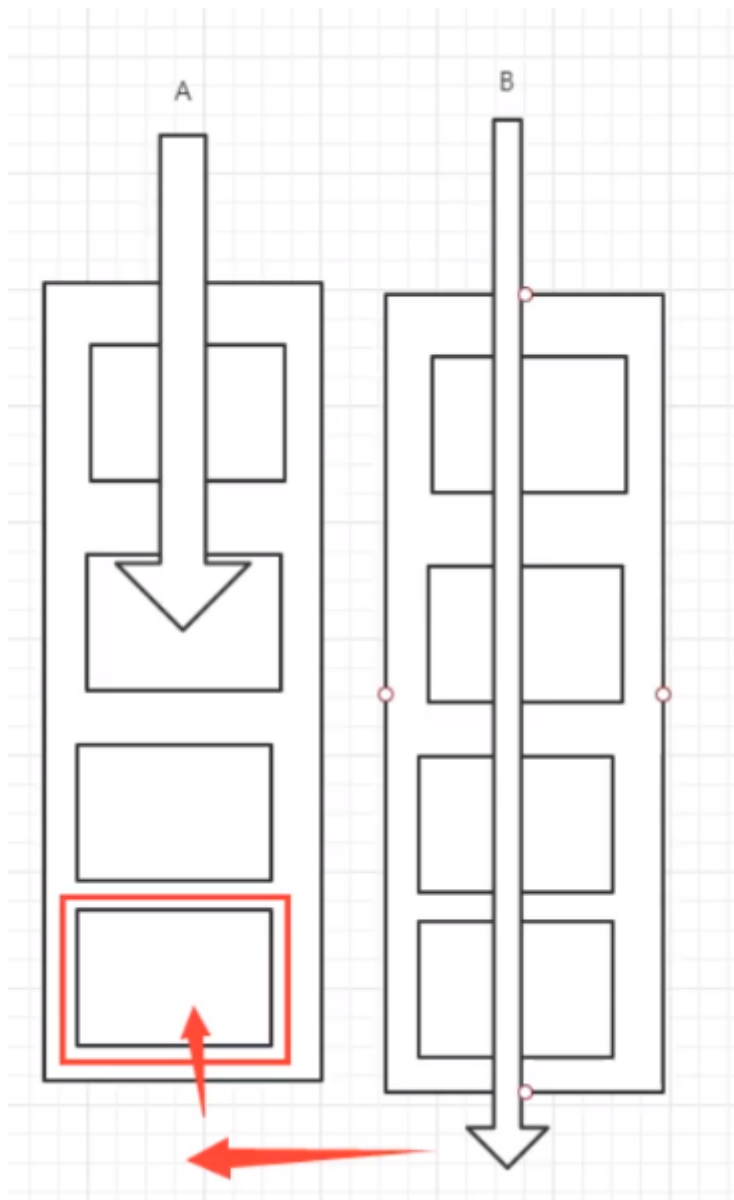
**A, B维护着双端队列**

假如A线程有4个任务，B线程也有4个任务

A线程还没执行完人物，但是B线程已经执行完任务了

这个时候B线程会 **从另一头** 把A线程的人物拿过来执行，不让B线程去等待

这就叫 **工作窃取**



## ForkJoin的操作

`java.util.concurrent`

ForkJoin分为两步操作：Fork, Join

1. ForkJoinPool执行
2. 计算任务forkjointask执行: `forkjoinPool.execute(ForkJoinTask task)`

execute: 同步提交

submit: 异步提交

ForkJoinTask

- RecursiveAction: 递归事件, 没有返回值
- RecursiveTask: 递归任务, 有返回值



```
compact1, compact2, compact3
java.util.concurrent
```

## Class ForkJoinTask<V>

```
java.lang.Object
java.util.concurrent.ForkJoinTask<V>
```

All Implemented Interfaces:

Serializable , Future <V>

已知直接子类:

递归任务, 无返回值

递归任务, 有返回值

CountedCompleter

RecursiveAction

RecursiveTask

### 3. 计算类继承 ForkJoinTask

- ForkJoinTask

```
1  import java.util.concurrent.RecursiveTask;
2
3  public class ForkJoinDemo extends RecursiveTask<Long> {
4
5      private Long start; //开始
6      private Long end;   //结束
7
8      private Long temp = 10000L; //临界值, 高于临界值使用forkjoin, 小于临界值用普通方法
9
10     public ForkJoinDemo(Long start, Long end) {
11         this.start = start;
12         this.end = end;
13     }
14
15     //计算
16     @Override
17     protected Long compute() {
18
19         if ((end-start)<temp){
20             Long sum = 0L;
21             for (Long i = start; i <= end; i++) {
22                 sum+=i;
23             }
24             return sum;
25         }else { //forkjoin
26             long middle = (start + end) / 2; //中间值, 使用中间值将这个任务分为两个小任务
27
28             ForkJoinDemo task1 = new ForkJoinDemo(start, middle); //前半段的任务
29             task1.fork(); //把任务压入队列
30
31             ForkJoinDemo task2 = new ForkJoinDemo(middle+1, end); //后半段的任务, 注意这里是
middle+1
32             task2.fork(); //把任务压入队列
33
34             return task1.join() + task2.join(); //获得结果
35
36         }
37     }
38 }
```

- Test

```
1  import java.util.concurrent.ExecutionException;
2  import java.util.concurrent.ForkJoinPool;
3  import java.util.concurrent.ForkJoinTask;
4  import java.util.stream.LongStream;
5
6  public class Test {
7      public static void main(String[] args) throws ExecutionException, InterruptedException {
8          //      test1();//sum=500000000500000000时间: 8296
9          //      test2();//sum=500000000500000000, 时间: 6227, 这里效率还可以更高, 他可以把临界值调整一下,
           我们这里使用的是1000L, 但是如果修改之后还可以更快
10         test3();//sum=500000000500000000, 时间: 543, 使用stream流还是牛逼
11     }
12
13     //拿3000块钱的程序员
14     public static void test1(){
15         long start = System.currentTimeMillis();
16
17         Long sum = 0L;
18         for (Long i = 1L; i <= 10_0000_0000L; i++) {
19             sum+=i;
20         }
21         long end = System.currentTimeMillis();
22         System.out.println("sum="+sum+", 时间: "+(end-start));
23     }
24
25     //拿6000块钱的程序员: forkjoin
26     public static void test2() throws ExecutionException, InterruptedException {
27         long start = System.currentTimeMillis();
28
29         ForkJoinPool forkJoinPool = new ForkJoinPool();
30         ForkJoinTask<Long> task = new ForkJoinDemo(1L, 10_0000_0000L);
31
32         //      forkJoinPool.execute(task);//执行任务, 没有返回值
33         ForkJoinTask<Long> submit = forkJoinPool.submit(task);//提交任务, 有返回值
34
35         Long sum = submit.get();//需要阻塞等待
36
37         long end = System.currentTimeMillis();
38         System.out.println("sum="+sum+", 时间: "+(end-start));
39     }
40
41
42
43     //拿9000块钱的程序员: stream
44     public static void test3() throws ExecutionException, InterruptedException {
45         long start = System.currentTimeMillis();
46
47         //range()和rangeClose()的选择区间: range左右不包含, rangeClose左开右闭
48         long sum = LongStream.rangeClosed(0L, 10_0000_0000L).parallel().reduce(0,
           Long::sum);
49
50         long end = System.currentTimeMillis();
51         System.out.println("sum="+sum+", 时间: "+(end-start));
52     }
53 }
```

从上面那个案例中可以看出，真要计算还是要是用stream流，但是forkjoin也是要会用

而且我们也知道了流还有LongStream，那么其他的DoubleStream，IntStream等等，没有区别

## 异步回调

Future：设计的初衷就是对未来的某个事件进行建模

java.util.concurrent

异步回调说白了就三个：

- 异步执行
- 成功回调
- 失败回调

Future的实现类：CompletableFuture

```
1  import java.util.concurrent.CompletableFuture;
2  import java.util.concurrent.ExecutionException;
3  import java.util.concurrent.TimeUnit;
4
5  //异步调用
6  public class Demo01 {
7      public static void main(String[] args) throws ExecutionException, InterruptedException {
8
9
10     //      resultNull();
11     resultNotNull();
12
13     }
14
15     public static void resultNull() throws ExecutionException, InterruptedException {
16         //发起一个请求
17
18         //没有返回值的异步回调，注意这里的Void
19         CompletableFuture<Void> completedFuture = CompletableFuture.runAsync(()->{
20             try {
21                 TimeUnit.SECONDS.sleep(2); //这里模拟结果执行
22             } catch (InterruptedException e) {
23                 e.printStackTrace();
24             }
25             System.out.println(Thread.currentThread().getName());
26         });
27         completedFuture.get(); //他要等待completedFuture的结果执行完毕才会得到结果，所以获取执行结果会
           阻塞
28         /*
29         上面的这个例子可以理解一下，我们可以把任务放到completedFuture里面去执行
```

```

30         然后等到我们需要结果的时候只需要执行completedFuture.get();就可以获得相应的结果
31         */
32     }
33
34
35     public static void resultNotNull() throws ExecutionException, InterruptedException {
36         //有返回值的异步回调
37         CompletableFuture<Integer> completableFuture = CompletableFuture.supplyAsync(()->{
38             return 1024;
39         });
40
41
42         Integer result = completableFuture
43             .whenComplete((t, u) -> { //编译成功
44                 System.out.println("t=>" + t); //正常的返回结果，这里是1024
45                 System.out.println("u=>" + u); //错误信息
46             })
47             .exceptionally((e) -> { //编译失败
48                 System.out.println(e.getMessage());
49                 e.printStackTrace();
50                 return 233;
51             }).get(); // .get()得到结果
52
53         System.out.println(result);
54     }
55 }

```

## JMM

请你谈谈 Volatile的理解

Volatile是JAVA虚拟机提供**轻量级的同步机制**

1. 保证可见性
2. **不保证** 原子性
3. 禁止指令重排

可见性--》JMM

### 什么是JMM

JMM：Java内存模型，这个是不存在的东西，是一种概念，约定

### 关于JMM的同步约定

1. 线程解锁前，必须把共享变量 **立刻** 刷新到主存

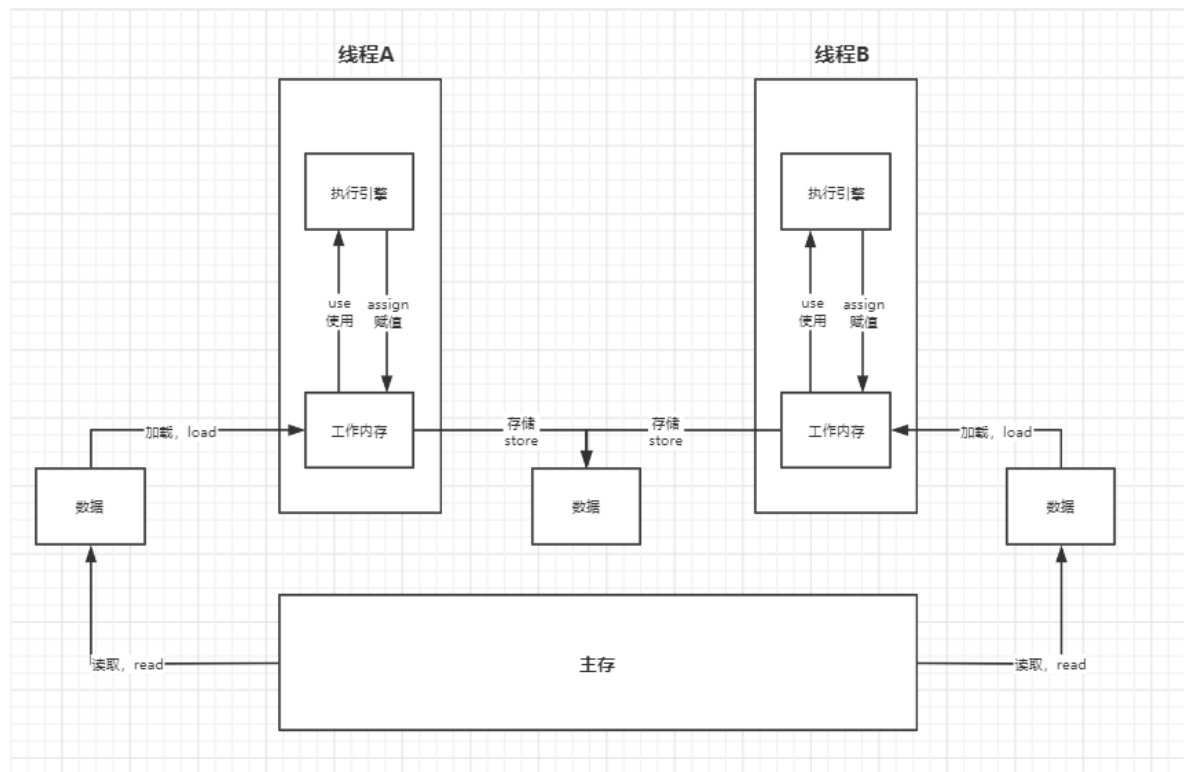
我们知道线程的操作是：将主内存的共享变量复制一份到线程自己的工作内存中

那么线程在解锁之前必须 **立刻** 把自己工作内存中的值刷新到主存中

2. 线程加锁前：必须读取主存中的最新值到工作内存中
3. 加锁和解锁必须是同一把锁

线程中分为工作内存和主内存

线程工作时会将内存中的数据刷到工作内存中，工作完成之后会将数据再次刷回主内存中



内存交互操作有8种，虚拟机实现必须保证每一个操作都是原子的，不可在分的

对于double和long类型的变量来说，load、store、read和write操作在某些平台上允许例外

- lock (锁定)：作用于主内存的变量，把一个变量标识为线程独占状态
- unlock (解锁)：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
- read (读取)：作用于主内存变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用
- load (载入)：作用于工作内存的变量，它把read操作从主存中变量放入工作内存中
- use (使用)：作用于工作内存中的变量，它把工作内存中的变量传输给执行引擎，每当虚拟机遇到一个需要使用到变量的值，就会使用到这个指令
- assign (赋值)：作用于工作内存中的变量，它把一个从执行引擎中接受到的值放入工作内存的变量副本中
- store (存储)：作用于主内存中的变量，它把一个从工作内存中一个变量的值传送到主内存中，以便后续的write使用
- write (写入)：作用于主内存中的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中

JMM对这八种指令的使用，制定了如下规则：

- 不允许read和load、store和write操作之一单独出现。即使用了read必须load，使用了store必须write

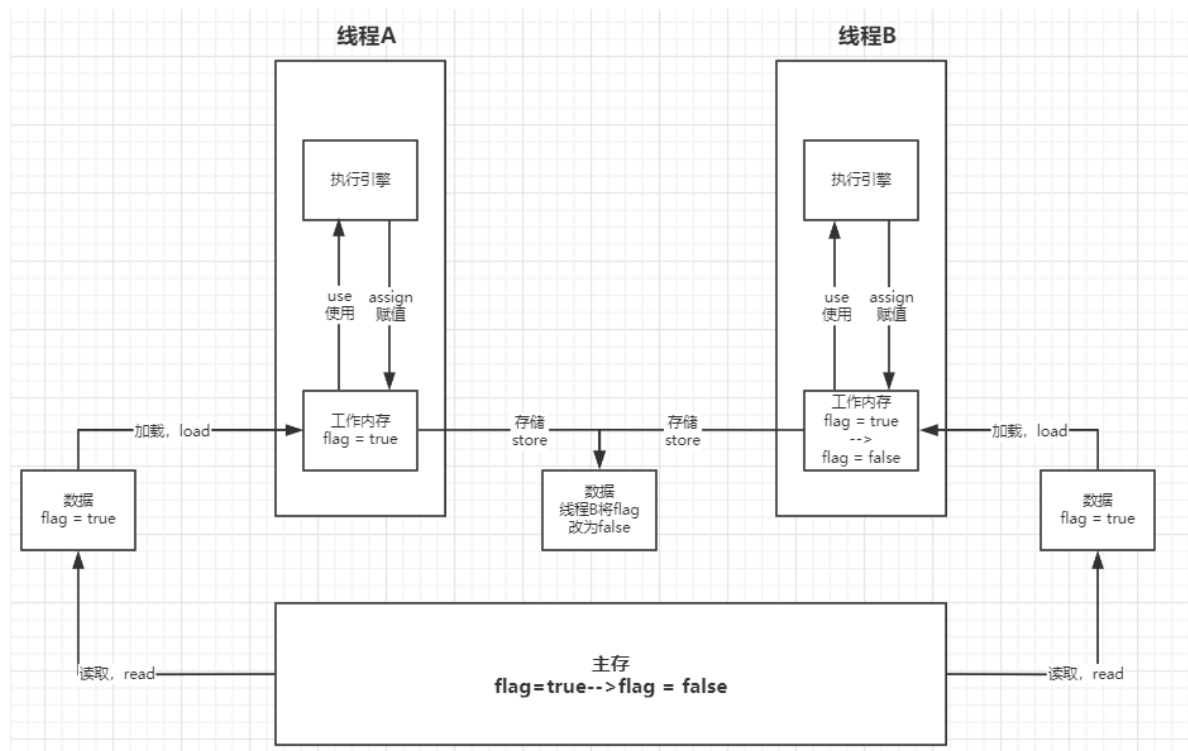
- 不允许线程丢弃他最近的assign操作，即工作变量的数据改变了之后，必须告知主存
- 不允许一个线程将没有assign的数据从工作内存同步回主内存
- 一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。就是变量实施use、store操作之前，必须经过assign和load操作
- 一个变量同一时间只有一个线程能对其进行lock。多次lock后，必须执行相同次数的unlock才能解锁
- 如果对一个变量进行lock操作，会清空所有工作内存中此变量的值，在执行引擎使用这个变量前，必须重新load或assign操作初始化变量的值
- 如果一个变量没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的变量
- 对一个变量进行unlock操作之前，必须把此变量同步回主内存

## Volatile

### 问题

那么现在有一个问题：

线程B修改了值，但是线程A不能及时可见



```

1  import java.util.concurrent.TimeUnit;
2
3  public class JMMDemo {
4
5      private static boolean flag = true;
6
7      public static void main(String[] args) {
8
9          new Thread(()->{
10              while (flag){
11              }
12          }).start();
13
14      try {

```

```

15         TimeUnit.SECONDS.sleep(2);
16     } catch (InterruptedException e) {
17         e.printStackTrace();
18     }
19
20     flag = false;
21     System.out.println(flag);
22 }
23 }

```

如果按照正常的逻辑，这里是应该是当flag = true的时候一直进入while，但是将flag = false修改之后结束循环

但是事实却是当flag输出为False的时候程序还没有结束

## Volatiles的三大特点

### 保证可见性

对上面的代码进行改造，假如volatile关键字

```

1  import java.util.concurrent.TimeUnit;
2
3  public class volatileDemo {
4
5      private volatile static boolean flag = true;
6
7      public static void main(String[] args) {
8
9          new Thread()->{
10              while (flag){
11              }
12          }.start();
13
14          try {
15              TimeUnit.SECONDS.sleep(2);
16          } catch (InterruptedException e) {
17              e.printStackTrace();
18          }
19
20          flag = false;
21          System.out.println(flag);
22      }
23  }

```

当控制台打印出：flag的时候，程序就已经停止

### 不保证原子性

原子性：操作不可分割，要么一起完成，要么不完成

```

1  //不保证原子性
2  public class volatileDemo2 {

```

```

3
4     private volatile static int num = 0;
5
6     public static void add(){
7         num++;
8     }
9
10    public static void main(String[] args) {
11        for (int i = 1; i <= 100; i++) {
12            new Thread()->{
13                for (int i1 = 0; i1 < 100; i1++) {
14                    add();
15                }
16            }.start();
17        }
18
19
20        while (Thread.activeCount()>2){//假如除了main线程和gc线程之外还有其他的线程
21            Thread.yield();//那么就让main线程去礼让其他的线程，让他们跑完
22        }
23
24        System.out.println(num);
25    }
26 }
27 //这次是9907

```

我们可以看到了，volatile不能保证原子操作

我们通过: `javap -c volatileDemo2.class` 来将class文件反编译，查询到:

```

public static void add();
Code:
  0: getstatic      #2                // Field num:I
  3: iconst_1
  4: iadd
  5: putstatic      #2                // Field num:I
  8: return

```

1. 获得这个值
2. 执行+1
3. 写会这个值

### 那么不加lock或者synchronized，如何保证原子性操作？

volatile不可以保证原子操作，那么可以使用lock锁或者synchronized保证原子操作，但是我们要说一个更厉害的:

在 `java.util.concurrent.atomic` 中，都是原子操作，其中有一个 `AtomicInteger`

那么我们对这个程序进行更改

```

1     import java.util.concurrent.atomic.AtomicInteger;

```



```

2
3 //不保证原子性
4 public class volatileDemo2 {
5
6 //    private volatile static int num = 0;
7     private volatile static AtomicInteger num = new AtomicInteger();
8
9     public static void add(){
10         num.getAndIncrement();//执行+1操作
11     }
12
13     public static void main(String[] args) {
14         for (int i = 1; i <= 100; i++) {
15             new Thread()->{
16                 for (int i1 = 0; i1 < 100; i1++) {
17                     add();
18                 }
19             }.start();
20         }
21
22
23         while (Thread.activeCount()>2){//假如除了main线程和gc线程之外还有其他的线程
24             Thread.yield();//那么就让main线程去礼让其他的线程，让他们跑完
25         }
26
27         System.out.println(num);
28     }
29 }

```

不论怎么测试，这个就变成了10000，不再变化，说明这才是保证的原子操作

那么为什么 `atomic` 这么厉害可以保证原子操作？

因为这些类的底层都和操作系统挂钩，在内存中修改值

## 禁止指令重排

指令重排：计算机并不是按照你写的程序进行执行的

从源代码到执行之间的可能性：编译器优化的重排，指令并行可能会重排，内存系统可能会重排

案例一：

比如：我写了一个

```

1 int x = 1; //第1步
2 int y = 2; //第2步
3 x = x + 5; //第3步
4 y = x * x; //第4步

```

我们期望的是 1234，但是在计算机中可能会被重新排列为2134，1324

但是不会为4123

因为不管是2134还是1324都不会对数据产生影响，但是4123会对数据产生影响

**处理器在考虑指令重排的时候会考虑数据之间的相互依赖**

案例二：

但是在多线程下看一个可能造成影响的结果：有  $a \times y$ ，这四个默认值都是0

线程A	线程B
$x = a$	$y = b$
$b = 1$	$a = 2$

正常的结果是： $x = 0$ ， $y = 0$

但是指令重排之后可能会造成线程A中和线程B中的顺序进行颠倒

线程A	线程B
$b = 1$	$a = 2$
$x = a$	$y = b$

重排之后的诡异结果： $x = 2$ ， $y = 1$

平时的时候是不会出现这种问题的，但是在理论上这个问题是存在的，所以即使这个问题没有出现我们也要预防可能发生的结果

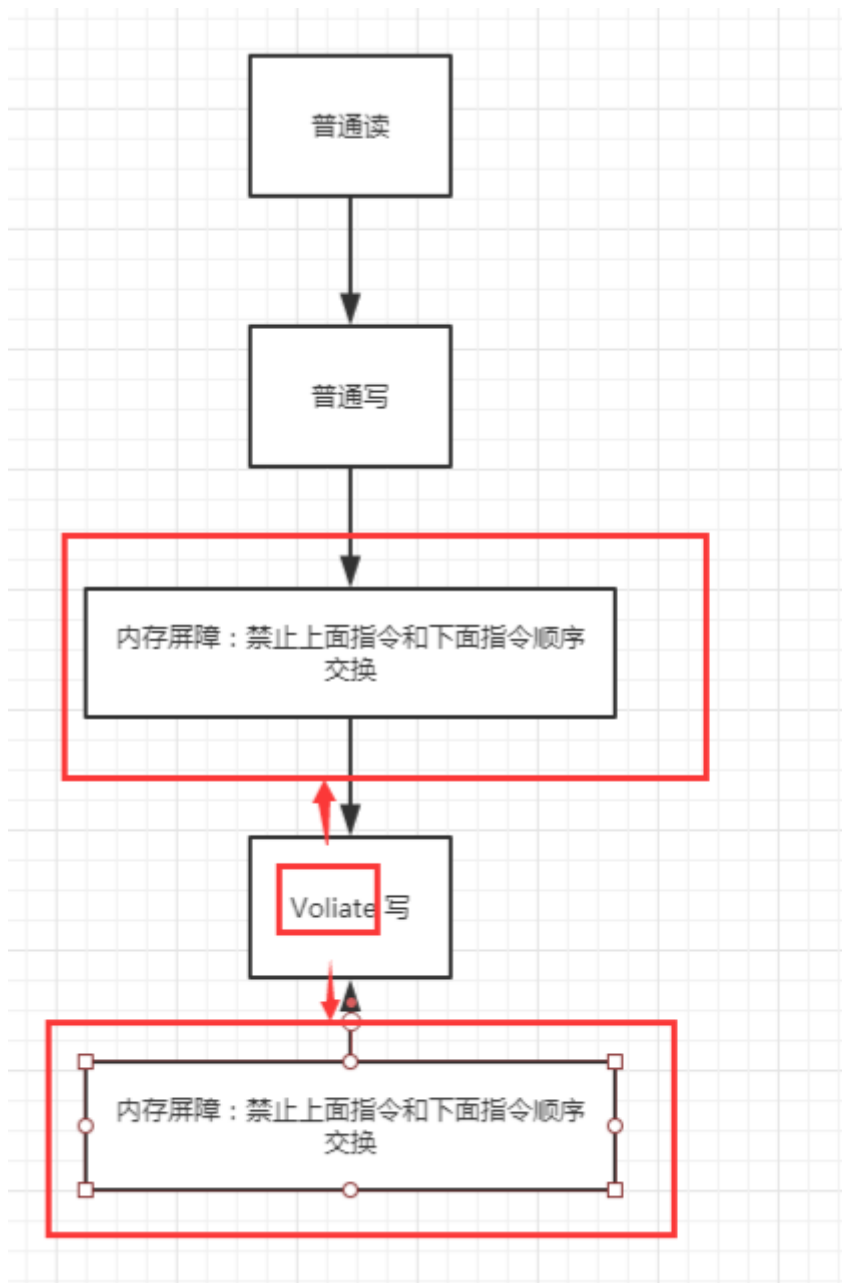
volatile可以避免指令重排的依仗：

在计算机中存在着一个叫做 **内存屏障** 的东西

内存屏障的作用：可以保证特定的执行顺序

可以保证某些变量的内存可见性（volatile就是利用这些特性实现了可见性）

volatile就是在他之前和之后都加上了内存屏障



## 单例模式

### 饿汉式单例

```
1 //饿汉式单例
2 public class Hungry {
3
4
5     private byte[] data1 = new byte[1024*1024];
6     private byte[] data2 = new byte[1024*1024];
7     private byte[] data3 = new byte[1024*1024];
8     private byte[] data4 = new byte[1024*1024];
9
10    private Hungry() {
11    }
12
13    //一开始就创建了，可能会浪费空间
```

```

14     private final static Hungry HUNGRY = new Hungry();
15
16     public static Hungry getInstance(){
17         return HUNGRY;
18     }
19 }

```

单例模式首先要进行构造器私有

## 懒汉式单例

```

1 //DCL 懒汉式单例
2 public class Lazy {
3
4     public Lazy() {
5     }
6
7     private static Lazy lazy;
8
9     public static Lazy getInstance(){
10         if (lazy!=null){
11             lazy = new Lazy();
12         }
13         return lazy;
14     }
15 }

```

饿汉式单例虽然浪费资源，但是在多线程的情况下貌似是没有问题的，但是懒汉式单例在多线程下可能就出现了问题

```

1 //DCL 懒汉式单例
2 public class LazyMan {
3
4     private LazyMan() {
5         System.out.println(Thread.currentThread().getName()+" ok");
6     }
7
8     private static LazyMan lazyMan;
9
10    public static LazyMan getInstance(){
11        if (lazyMan ==null){
12            lazyMan = new LazyMan();
13        }
14        return lazyMan;
15    }
16
17    public static void main(String[] args) {
18        for (int i = 0; i < 10; i++) {
19            new Thread(()->{
20                LazyMan.getInstance();
21            }).start();
22        }
23    }
24 }

```

```
25 //Thread-2 ok
26 //Thread-1 ok
27 //Thread-0 ok
```

很明显，多线程下单例是有问题的，构造方法执行了三次，他们拿到的并不是一个，单例模式失败了

我们可以使用锁来解决多线程的一个夺取问题

```
1 public static LazyMan getInstance(){
2     if (lazyMan ==null){
3         synchronized (LazyMan.class){
4             if (lazyMan==null){
5                 lazyMan = new LazyMan();
6             }
7         }
8     }
9     return lazyMan;
10 }
```

使用 **双重检测锁模式** 来检测DCL懒汉式单例，我们再次运行，发现没有问题了，从始至终都只有一个 `Thread-0 ok`，说明构造方法只执行了一次，单例模式执行Ok

但是这样其实并不保险，因为 `lazyMan = new LazyMan()` 它不是一个原子性操作，至少要执行三步：

1. 分配内存空间
2. 执行构造方法初始化对象
3. 把对象指向空间

这个时候也有可能出现指令重排的现象，比如改为 `132`。

对于多线程模式下者就完蛋了，因为可能A线程执行了13，然后B线程来了，发现 `lazyMan!=null`，这个时候直接 `return lazyMan`，但是事实上还没有完成构造，这就完蛋了

所以为了防止指令重排，必须加上 **volatile**

```
1 private LazyMan() {
2     System.out.println(Thread.currentThread().getName()+" ok");
3 }
4
5 private volatile static LazyMan lazyMan;
6
7 public static LazyMan getInstance(){
8     if (lazyMan ==null){
9         synchronized (LazyMan.class){
10             if (lazyMan==null){
11                 lazyMan = new LazyMan();
12             }
13         }
14     }
15     return lazyMan;
16 }
```

## 静态内部类

静态内部类实现

```
1 //静态内部类实现
2 public class Holder {
3     private Holder(){}
4
5     public static Holder getInstance(){
6         return InnerClass.HOLDER;
7     }
8
9     public static class InnerClass{
10         private static final Holder HOLDER = new Holder();
11     }
12 }
```

静态内部类看起来虽然不错，但是其实并没有什么卵用

## 利用反射破解单例模式

单例模式其实都是不安全的，比如我们现在要破解DCL懒汉式单例

```
1 import java.lang.reflect.Constructor;
2
3 //懒汉式单例
4 public class LazyMan {
5
6     private LazyMan() {}
7
8     private volatile static LazyMan lazyMan;
9
10    public static LazyMan getInstance(){
11        if (lazyMan ==null){
12            synchronized (LazyMan.class){
13                if (lazyMan==null){
14                    lazyMan = new LazyMan();
15                }
16            }
17        }
18        return lazyMan;
19    }
20
21    public static void main(String[] args) throws Exception {
22        LazyMan lazyMan1 = LazyMan.getInstance();//普通方法获得
23
24        //利用反射破坏单例获得
25        Constructor<LazyMan> declaredConstructor =
26            LazyMan.class.getDeclaredConstructor(null);
27        declaredConstructor.setAccessible(true);
28        LazyMan lazyMan2 = declaredConstructor.newInstance();
29    }
30 }
```

```

28
29         System.out.println(lazyMan1); //com.bean.single.LazyMan@74a14482
30         System.out.println(lazyMan2); //com.bean.single.LazyMan@1540e19d
31
32     }
33 }

```

现在看的很清楚了，他们两个不一样，单例被破坏了

解决单例的破坏问题：加锁

```

1 //懒汉式单例
2 public class LazyMan {
3
4     private LazyMan() {
5         synchronized (LazyMan.class){
6             if (lazyMan!=null){
7                 throw new RuntimeException("不要使用反射破坏");
8             }
9         }
10    }
11
12    private volatile static LazyMan lazyMan;
13
14    public static LazyMan getInstance(){
15        if (lazyMan ==null){
16            synchronized (LazyMan.class){
17                if (lazyMan==null){
18                    lazyMan = new LazyMan();
19                }
20            }
21        }
22        return lazyMan;
23    }
24
25    public static void main(String[] args) throws Exception {
26        LazyMan lazyMan1 = LazyMan.getInstance(); //普通方法获得
27
28        //利用反射破坏单例获得
29        Constructor<LazyMan> declaredConstructor =
30        LazyMan.class.getDeclaredConstructor(null);
31        declaredConstructor.setAccessible(true);
32        LazyMan lazyMan2 = declaredConstructor.newInstance();
33
34        System.out.println(lazyMan1);
35        System.out.println(lazyMan2);
36    }
37 }

```

我们直接在类模板上加了一个锁，普通情况下已经有了 `lazyMan` 的时候不会执行构造方法，但是现在还是执行了

说明有人在用反射破坏

执行之后发现抛出了异常，没有问题 `java.lang.RuntimeException: 不要使用反射破坏`

现在成为了三重检测

虽然这样检测，其实还是可以破坏的

```
1 LazyMan lazyMan1 = LazyMan.getInstance();
```

我们之前看到的第一个对象是用这种普通方法去创建的，所以会执行构造方法

但是现在我不用这个方法，我直接不创建这个对象，直接使用方法，那单例模式就又被破坏了

```
1 public static void main(String[] args) throws Exception {
2     // LazyMan lazyMan1 = LazyMan.getInstance(); //普通方法获得
3
4     //利用反射破坏单例获得
5     Constructor<LazyMan> declaredConstructor =
6     LazyMan.class.getDeclaredConstructor(null);
7     declaredConstructor.setAccessible(true);
8     LazyMan lazyMan1 = declaredConstructor.newInstance(); //现在直接不创建对象，直接使用这个方法，那单例完蛋了
9     LazyMan lazyMan2 = declaredConstructor.newInstance();
10
11     System.out.println(lazyMan1); //com.bean.single.LazyMan@74a14482
12     System.out.println(lazyMan2); //com.bean.single.LazyMan@1540e19d
13
14 }
```

即便是这样，还是有办法来解决的：

使用红绿灯办法，设置一个flag变量，在不知道这个变量的情况下，反射还是不能破坏

```
1 //懒汉式单例
2 public class LazyMan {
3
4     private static boolean flag = false;
5
6     private LazyMan() {
7         synchronized (LazyMan.class) {
8             if (flag == false) {
9                 flag = true;
10            } else {
11                throw new RuntimeException("不要使用反射破坏");
12            }
13        }
14    }
15
16    private volatile static LazyMan lazyMan;
17
18    public static LazyMan getInstance() {
19        if (lazyMan == null) {
```



```

20         synchronized (LazyMan.class){
21             if (lazyMan==null){
22                 lazyMan = new LazyMan();
23             }
24         }
25     }
26     return lazyMan;
27 }
28
29 public static void main(String[] args) throws Exception {
30     // LazyMan lazyMan1 = LazyMan.getInstance();//普通方法获得
31
32     //利用反射破坏单例获得
33     Constructor<LazyMan> declaredConstructor =
LazyMan.class.getDeclaredConstructor(null);
34     declaredConstructor.setAccessible(true);
35
36     LazyMan lazyMan1 = declaredConstructor.newInstance();
37     LazyMan lazyMan2 = declaredConstructor.newInstance();
38
39     //java.lang.RuntimeException: 不要使用反射破坏
40     System.out.println(lazyMan1);
41     System.out.println(lazyMan2);
42
43 }
44 }

```

不管是不是要创建对象，构造方法总是要走的，只要一走构造方法，flag就会变为true，然后就可以防止反射被破坏

但是这样依旧不安全，我们可以使用反编译的方式查看这个变量。

```

1     public static void main(String[] args) throws Exception {
2
3         Constructor<LazyMan> declaredConstructor =
LazyMan.class.getDeclaredConstructor(null);
4         declaredConstructor.setAccessible(true);
5
6         LazyMan lazyMan1 = declaredConstructor.newInstance();
7
8
9         Field flag = LazyMan.class.getDeclaredField("flag");
10        flag.setAccessible(true);
11        flag.set(lazyMan1, false);
12
13        LazyMan lazyMan2 = declaredConstructor.newInstance();
14
15
16        System.out.println(lazyMan1);
17        System.out.println(lazyMan2);
18
19    }

```

## 单例又被破坏了

这样还是可以防止的：我们可以使用枚举防止单例被破坏，反射是不能破坏枚举的

```
1 public enum EnumSingle {
2     INSTANCE;
3
4     public EnumSingle getInstance(){
5         return INSTANCE;
6     }
7 }
```

我们通过查看它的class文件可以发现它是一个无参构造

```
1 public enum EnumSingle {
2     INSTANCE;
3
4     private EnumSingle() {
5     }
6
7     public EnumSingle getInstance() {
8         return INSTANCE;
9     }
10 }
```

```
1 class Test{
2     public static void main(String[] args) throws Exception {
3         EnumSingle instance = EnumSingle.INSTANCE;
4
5         Constructor<EnumSingle> declaredConstructor =
6 EnumSingle.class.getDeclaredConstructor(null);
7         declaredConstructor.setAccessible(true);
8         EnumSingle enumSingle = declaredConstructor.newInstance();
9     }
10 }
```

```
java.lang.NoSuchMethodException: com.been.single.EnumSingle.<init>()
```

报错了，他说没有这个构造方法

问题来了，明明有这个构造方法，却说没有，这个说明class文件欺骗了我们，通过反编译 `javap -c xxx.class` 来查看

```
1 C:\Users\Administrator\Desktop\开发编程\ juc\target\classes\com\kuang\single>javap -p EnumSingle.class
1 Compiled from "EnumSingle.java"
1 public final class com.kuang.single.EnumSingle extends java.lang.Enum<com.kuang.single.EnumSingle> {
1     public static final com.kuang.single.EnumSingle INSTANCE;
1     private static final com.kuang.single.EnumSingle[] $VALUES;
1     public static com.kuang.single.EnumSingle[] values();
1     public static com.kuang.single.EnumSingle valueOf(java.lang.String);
1     private com.kuang.single.EnumSingle();
1     public com.kuang.single.EnumSingle getInstance();
1     static {};
```

发现还是空参方法，说明这个也骗了我们

那么使用专业的工具 `jad.exe`，反编译成java文件，查看出来：

```
EnumSingle.java
11
12     public static EnumSingle[] values()
13     {
14         return (EnumSingle[])$VALUES.clone();
15     }
16
17     public static EnumSingle valueOf(String name)
18     {
19         return (EnumSingle)Enum.valueOf(com/kuang/single/EnumSingle, name);
20     }
21
22     private EnumSingle(String s, int i)
23     {
24         super(s, i);
25     }
26
27     public EnumSingle getInstance()
28     {
29         return INSTANCE;
30     }
31
32     public static final EnumSingle INSTANCE;
33     private static final EnumSingle $VALUES[];
34
35     static
36     {
37         INSTANCE = new EnumSingle("INSTANCE", 0);
38         $VALUES = (new EnumSingle[] {
39             INSTANCE
40         });
41     }
42 }
43
```

构造方法变了，不是空参的构造方法

那就好办了，继续破解

```
1  class Test{
2      public static void main(String[] args) throws Exception {
3          EnumSingle instance = EnumSingle.INSTANCE;
4
5          Constructor<EnumSingle> declaredConstructor =
6          EnumSingle.class.getDeclaredConstructor(String.class,int.class);
7          declaredConstructor.setAccessible(true);
8          EnumSingle enumSingle = declaredConstructor.newInstance();
9      }
10 }
```

执行，发现错误变了： `Cannot reflectively create enum objects`

反射不能破坏枚举对象

到此为止，交锋结束了

枚举确实好使

## CAS

什么是CAS

**CAS：比较当前内存中的值和主内存中的值，如果这个值是期望的，那么则执行操作，如果不是则一直循环**

```
1  import java.util.concurrent.atomic.AtomicInteger;
```

```

2
3 //CAS: compare and set-->比较并交换
4 //CAS是CPU的并发原语
5 public class CASDemo {
6
7     public static void main(String[] args) {
8         //之前的AtomicInteger其实就是使用了CAS
9         AtomicInteger atomicInteger = new AtomicInteger(2020);
10
11         //期望, 更新
12         //public final boolean compareAndSet(int expect,int update)
13         //如果我期望的值达到了, 那么就更新, 否则就不更新
14         System.out.println(atomicInteger.compareAndSet(2020, 2021));
15         System.out.println(atomicInteger.get()); //2021
16
17         //再次修改就会修改失败, 因为只有2020才能够修改为2021, 但是上面已经修改为了2021
18         System.out.println(atomicInteger.compareAndSet(2020, 2021));
19     }
20 }

```

## Unsafe类

之前我们使用过AtomicInteger的+1操作, 现在来看一下底层

```

1     public final int getAndIncrement() {
2         return unsafe.getAndAddInt(this, valueOffset, 1);
3     }
4
5
6     private static final Unsafe unsafe = Unsafe.getUnsafe();
7     private static final long valueOffset;
8
9     static {
10         try {
11             valueOffset = unsafe.objectFieldOffset //获得内存地址偏移值
12                 (AtomicInteger.class.getDeclaredField("value"));
13         } catch (Exception ex) { throw new Error(ex); }
14     }
15
16     private volatile int value; //避免指令重排

```

unsafe类:

- Java不可以操作内存
- Java可以调用c++
- c++可以操作内存

所以这个类是Java的后门, 可以通过这个类来操作内存

```

    public final int getAndIncrement() {
        return unsafe.getAndAddInt(0: this, valueOffset, i: 1);
    }

    public final int getAndAddInt(Object var1, long var2, int var4) {
        int var5;
        do {
            var5 = this.getIntVolatile(var1, var2);
        } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));

        return var5;
    }
}

```

getAndAddInt: 当前对象, 当前地址偏移值, 要增加的值

所以 `var5` 就是获取当前对象的地址偏移值, 也就是获取内存地址中的值

```
compareAndSwapInt(var1, var2, var5, var5 + var4)
```

如果当前的地址偏移值还是我期望的那个地址偏移值, 那么就让他+var4

如果 `var1+var2` 还是期望的那个地址偏移值 `var5`, 那么就让 `var5+var4`

所以这是一个内存操作, 效率非常高

`atomicInteger.getAndIncrement()` 所以这是一个标准的 **自旋锁**

CAS的优缺点

优点: 自带原子性

缺点:

1. 循环会耗时, 但是比java强多了
2. 一次性只能保证一个共享变量的原子性
3. ABA问题

CAS: ABA问题 (狸猫换太子)

现在有一个资源A = 1, 两条线程

线程1期望A=1, 要通过CAS修改为2

线程2也期望A=3, 要通过CAS修改为3

那么现在: 线程2因为操作比较快, 将A修改为了3, 然后又修改回了1

这个过程, 线程1是不知情的, 它拿到了资源A, 那么这个时候A = 1

这个线程1以为A还是之前那个A，但是其实已经被动过手脚了

这就是ABA问题

## 测试

```
1  import java.util.concurrent.atomic.AtomicInteger;
2
3  //CAS: compare and set-->比较并交换
4  //CAS是CPU的并发原语
5  public class CASDemo {
6
7      public static void main(String[] args) {
8
9          AtomicInteger atomicInteger = new AtomicInteger(2020);
10
11
12         /*=====捣乱的线程=====*/
13         System.out.println(atomicInteger.compareAndSet(2020, 2021));
14         System.out.println(atomicInteger.get());
15         System.out.println(atomicInteger.compareAndSet(2021, 2020));
16         System.out.println(atomicInteger.get());
17
18         /*=====期望的线程=====*/
19         System.out.println(atomicInteger.compareAndSet(2020, 6666));
20         System.out.println(atomicInteger.get());
21
22     }
23 }
24 }
```

对于我们写的SQL，只要写上一把乐观锁，就可以解决这个问题

## 原子引用解决ABA

带版本号原子操作

`java.util.concurrent.atomic.AtomicReference`：原子引用

```
1
2  //CAS: compare and set-->比较并交换
3  //CAS是CPU的并发原语
4  public class CASDemo {
5
6      public static void main(String[] args) {
7
8          //      AtomicInteger atomicInteger = new AtomicInteger(2020);
9
10         //替换AtomicInteger，使用带有版本号的AtomicStampedReference，有两个参数：期望值，版本号
```

```

11      /*
12          注意，这里的Integer，Integer使用了对象缓存机制，默认范围是-128~127，推荐使用静态工厂方法
valueOf获取对象实例，而不是new
13          因为valueOf使用缓存，new一定会创建新的对象分配新的内存空间
14          因为一开始的默认值是2020，超出了Integer的范围，所以出现了错误
15
16          而且所有相同类型的包装类之间的比较要全部使用equals，这是因为在-128~127之间是从缓存中产生的，
在这个区间中会复用已有对象，可以使用==
17          但是超过了这个区间就会在堆上产生，并不会复用已有的对象，那么指向的并不是一个，所以会出现问题
18
19          这是一个大坑
20
21          所以泛型如果是一个包装类，就要注意对象的引用问题
22      */
23      AtomicStampedReference<Integer> atomicStampedReference = new
AtomicStampedReference<>(1,1);
24
25      new Thread()->{
26
27          System.out.println("A1 ==>" + atomicStampedReference.getStamp()); //获取版本号
28          try {
29              TimeUnit.SECONDS.sleep(1);
30          } catch (InterruptedException e) {
31              e.printStackTrace();
32          }
33          //期望的值为2020，修改的值为2022。修改值并执行版本号+1
34          System.out.println(atomicStampedReference.compareAndSet(1, 2,
atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1));
35
36          System.out.println("A2 ==>" + atomicStampedReference.getStamp()); //获取版本号
37
38          //再把值改回去
39          System.out.println(atomicStampedReference.compareAndSet(2, 1,
atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1));
40
41          System.out.println("A3 ==>" + atomicStampedReference.getStamp()); //获取版本号
42      }, "A").start();
43
44      new Thread()->{
45          System.out.println("B1 ==>" + atomicStampedReference.getStamp()); //获取版本号
46          try {
47              TimeUnit.SECONDS.sleep(2);
48          } catch (InterruptedException e) {
49              e.printStackTrace();
50          }
51
52          System.out.println(atomicStampedReference.compareAndSet(1, 6,
atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1));
53
54          System.out.println("B2 ==>" + atomicStampedReference.getStamp());
55      }, "B").start();
56
57
58  }
59
60
61  }

```

# 锁

## 公平锁，非公平锁

公平锁：非常公平，一个队列不能插队

非公平锁：可以插队

默认都是非公平

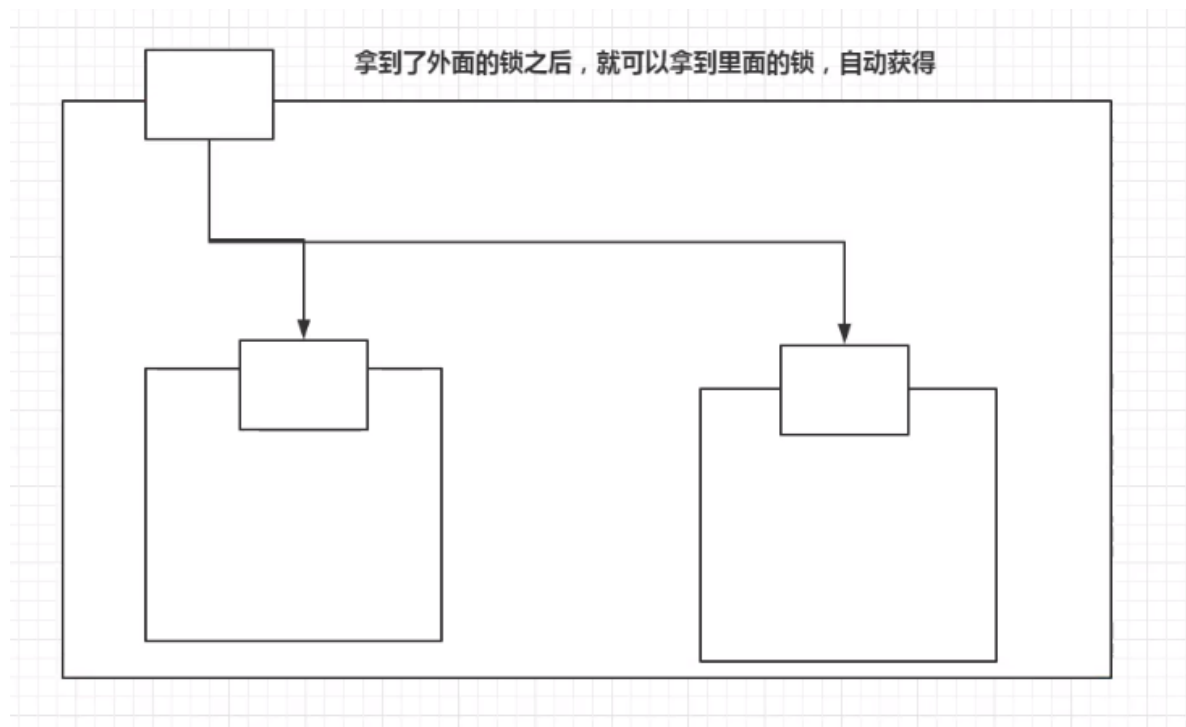
```
public ReentrantLock() {
    sync = new NonfairSync();
}

public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

在前面的Lock锁就已经讲过了

## 可重入锁

所有的锁都是可重入锁，也叫递归锁



```
1 //可重入锁
2 public class Demo01 {
3     public static void main(String[] args) {
4
5         Phone phone = new Phone();
6
7         new Thread(()->{
8             phone.sms();
9             }, "A").start();
```



```

10
11         new Thread()->{
12             phone.sms();
13         }, "B").start();
14     }
15 }
16
17 class Phone{
18     public synchronized void sms(){
19         System.out.println(Thread.currentThread().getName()+"sms");
20         call();//这里也有锁
21     }
22
23     public synchronized void call(){
24         System.out.println(Thread.currentThread().getName()+"call");
25     }
26 }
27 /*
28 Asms
29 Acall
30 Bsms
31 Bcall
32 */

```

在上面我们可以知道，无论执行多少次，A都会先执行完，然后释放锁

这里A获得的是两个锁：sms方法和call方法的锁

这就是可重入锁，A获取了sms的锁之后自动获得call的锁

使用Lock锁也是一样的结果，本质上是一样的

```

1  import java.util.concurrent.locks.Lock;
2  import java.util.concurrent.locks.ReentrantLock;
3
4  //可重入锁
5  public class Demo02 {
6      public static void main(String[] args) {
7
8          Phone phone = new Phone();
9
10         new Thread()->{
11             phone.sms();
12         }, "A").start();
13
14         new Thread()->{
15             phone.sms();
16         }, "B").start();
17     }
18 }
19
20 class Phone2{
21     Lock lock = new ReentrantLock();
22     public void sms(){
23         lock.lock();//加锁
24         try {
25             System.out.println(Thread.currentThread().getName()+"sms");
26             call();//这里也有锁

```

```

27         } catch (Exception e) {
28             e.printStackTrace();
29         } finally {
30             lock.unlock();//解锁
31         }
32     }
33
34     public synchronized void call(){
35         lock.lock();    //加锁，注意这里的锁和上面的锁不是同一把锁
36         try {
37             System.out.println(Thread.currentThread().getName()+"call");
38         } catch (Exception e) {
39             e.printStackTrace();
40         } finally {
41             lock.unlock();//解锁，注意这里的锁和上面的锁不是同一把锁
42         }
43     }
44 }

```

## 自旋锁

**spinlock**，自旋锁，这种锁会不断循环，直到成功

```

i6     public native int getLoadAverage(double[] var1, int var2);
i7
i8     public final int getAndAddInt(Object var1, long var2, int var4) {
i9         int var5;
i10        do {
i11            var5 = this.getIntVolatile(var1, var2);
i12        } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
i13
i14        return var5;
i15    }
i16
i17     public final long getAndAddLong(Object var1, long var2, long var4) {

```

```

1 //自旋锁
2 public class SpinlockDemo {
3     AtomicReference<Thread> atomicReference = new AtomicReference<>();
4
5     //加锁
6     public void myLock(){
7         Thread thread = Thread.currentThread();//获取当前的线程
8
9         System.out.println(Thread.currentThread().getName()+" ==> mylock");
10
11         //自旋锁，使用CAS，我们的期望值为null，然后设为thread
12         //这个CAS操作返回肯定是false，所以我们对他进行取反，让它一直循环
13         while (!atomicReference.compareAndSet(null, thread)){
14
15         }
16     }
17
18     //解锁
19     public void myUnLock(){
20         Thread thread = Thread.currentThread();//获取当前的线程
21
22         System.out.println(Thread.currentThread().getName()+"==>myUnLock");
23
24         //加锁自旋，解锁就不需要了

```

```

25         //我们期望值为thread当前线程，把它置为空
26         //以达到lock锁while循环解锁的条件
27         atomicReference.compareAndSet(thread,null);
28     }
29 }

```

```

1  import java.util.concurrent.TimeUnit;
2
3
4  public class TestSpinLock {
5      public static void main(String[] args) {
6          //第层使用了自旋锁，使用CAS操作
7          SpinlockDemo lock = new SpinlockDemo();
8
9          new Thread()->{
10              lock.myLock();
11              try {
12                  TimeUnit.SECONDS.sleep(5); //休息一下
13              } catch (InterruptedException e) {
14                  e.printStackTrace();
15              } finally {
16                  lock.myUnLock();
17              }
18
19          }, "T1").start();
20
21          //我们休息一秒钟，保证t1可以先获取到锁
22          try {
23              TimeUnit.SECONDS.sleep(1);
24          } catch (InterruptedException e) {
25              e.printStackTrace();
26          }
27
28          new Thread()->{
29              lock.myLock();
30              try {
31                  TimeUnit.SECONDS.sleep(1); //t2也休息一下
32              } catch (InterruptedException e) {
33                  e.printStackTrace();
34              } finally {
35                  lock.myUnLock();
36              }
37          }, "T2").start();
38
39      }
40  }
41  /*
42  T1 ==> mylock
43  T2 ==> mylock
44  T1==>myUnLock
45  T2==>myUnLock
46  */

```

这样答案很明显了

只有当T1释放了锁之后，T2才有资格释放锁

情况是这样的：

t1首先执行了mylock的输出，然后使用了CAS

然后t2执行了mylock的输出，但是因为CAS正在修改t1，所以t2一直在等待

然后t1执行了unlock

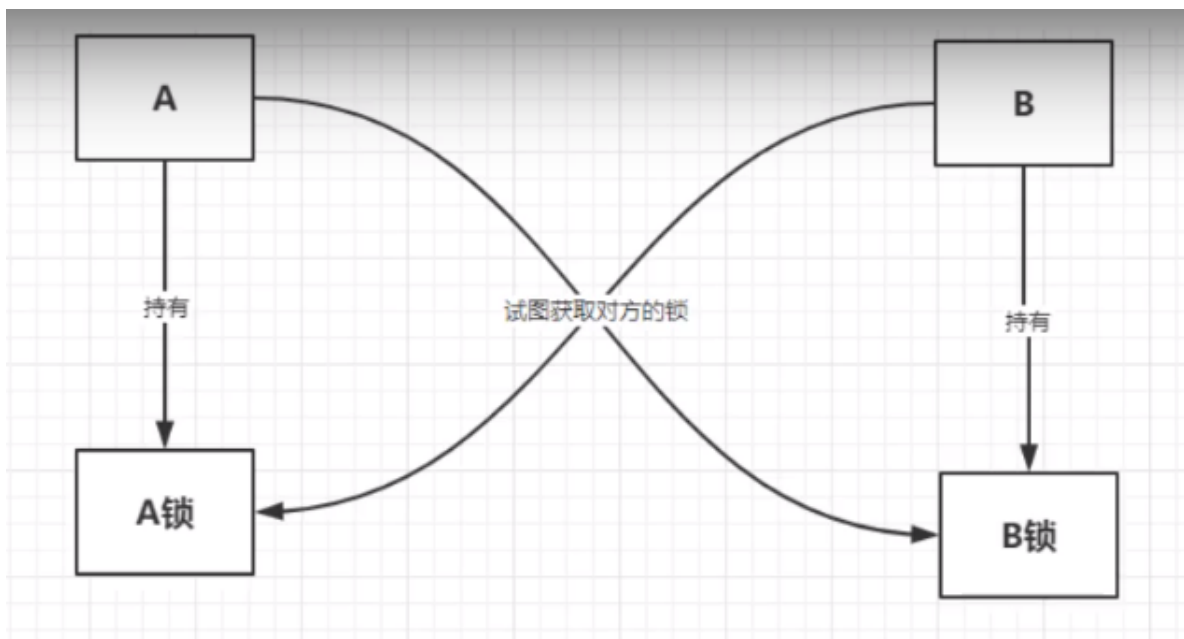
t2使用了CAS，然后执行了unlock

如果不信，可以把lock输出放到while之后，然后再执行一遍看看

## 死锁

### 什么是死锁

死锁就是两个线程互相抢对方的锁



### 怎么排除死锁

解决问题

1. 使用 `jps -l` 定位进程号

```
C:\Users\Administrator\Desktop\并发编程\juc>jps -l
10048
1140 org.jetbrains.jps.cmdline.Launcher
11444 com.kuang.lock.DeadLockDemo
9400 org.jetbrains.idea.maven.server.RemoteMavenServer
7884 sun.tools.jps.Jps
```

2. 使用 `jstack 进程号` 找到死锁问题

```
Found one Java-level deadlock:
=====
"T2":
  waiting to lock monitor 0x0000000018590f28 (object 0x00000000d5b86a90, a java.lang.String),
  which is held by "T1"
"T1":
  waiting to lock monitor 0x00000000185932e8 (object 0x00000000d5b86ac8, a java.lang.String),
  which is held by "T2"

Java stack information for the threads listed above:
=====
"T2":
  at com.kuang.lock.MyThread.run(DeadLockDemo.java:42)
    - waiting to lock <0x00000000d5b86a90> (a java.lang.String)
    - locked <0x00000000d5b86ac8> (a java.lang.String)
    at java.lang.Thread.run(Thread.java:748)
"T1":
  at com.kuang.lock.MyThread.run(DeadLockDemo.java:42)
    - waiting to lock <0x00000000d5b86ac8> (a java.lang.String)
    - locked <0x00000000d5b86a90> (a java.lang.String)
    at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

面试或者工作中，出现问题可以查看：日志，堆栈信息