多线程与并发编程

多线程介绍

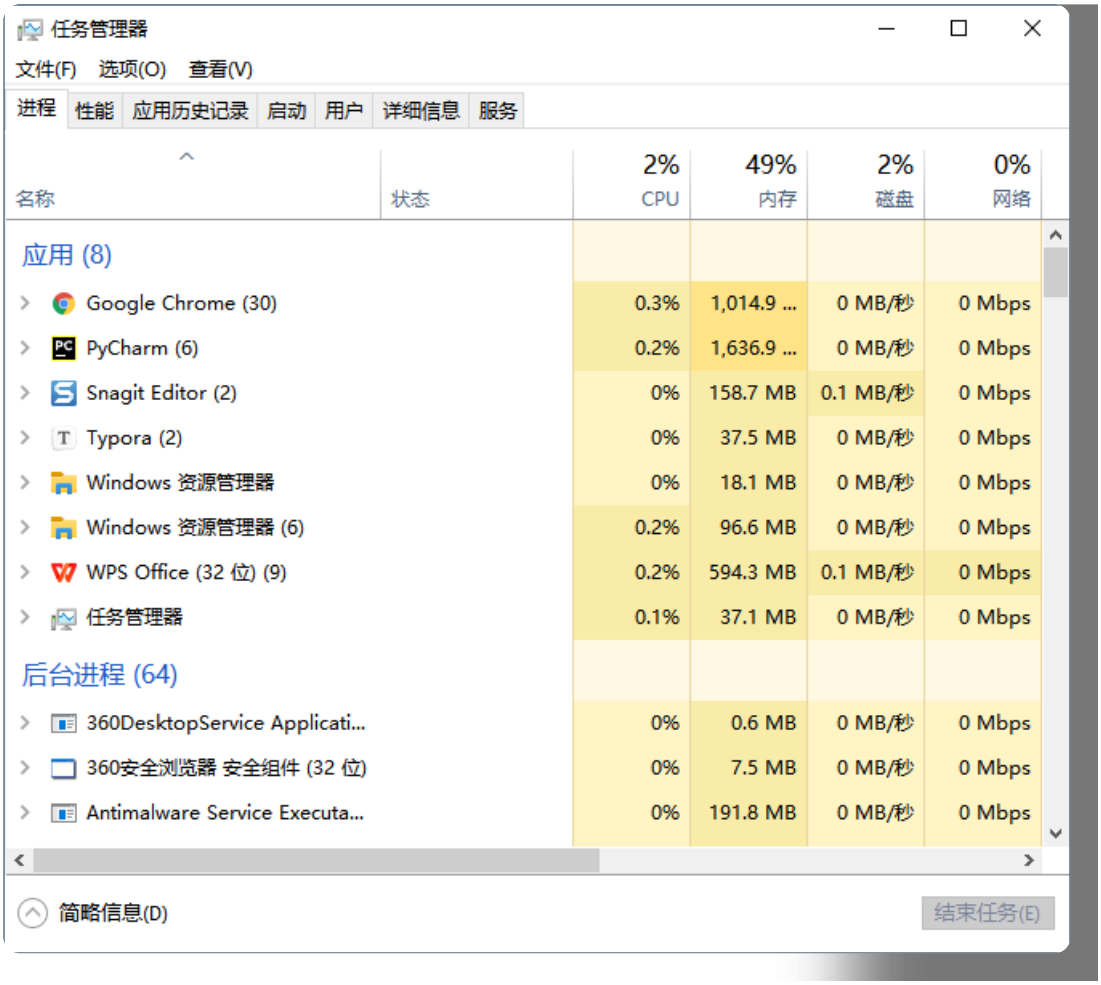
什么是程序?

程序（Program）是一个静态的概念，一般对应于操作系统中的一个可执行文件。

什么是进程?

执行中的程序叫做进程(Process)，是一个动态的概念。其实进程就是一个在内存中独立运行的程序空间 。

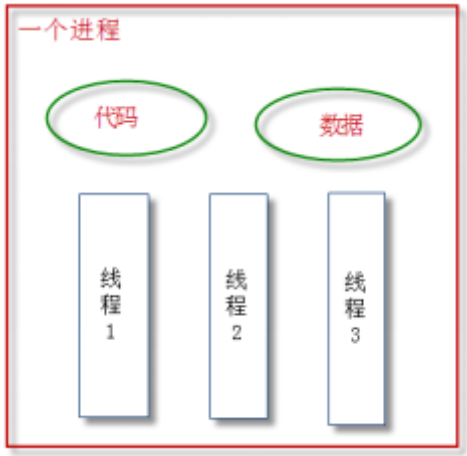
> 现代操作系统比如Mac OS X，Linux，Windows等，都是支持“多任务”的操作系统，叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用逛淘宝，一边在听音乐，一边在用微信聊天，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。



什么是线程?

线程（Thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。

> 有些进程还不止同时干一件事，比如微信，它可以同时进行打字聊天，视频聊天，朋友圈等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。



实时效果反馈

1. 如下对线程描述，错误的说法是：

A 一个进程可以产生多个线程；

B 多个线程可以共享同一进程中的资源；

C 线程的启动或消亡时消耗资源非常少；

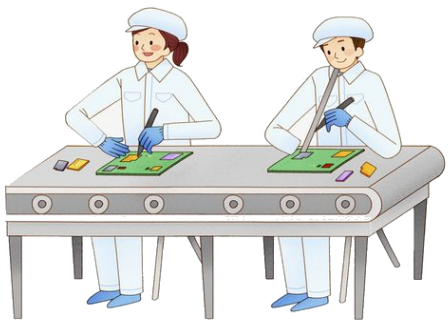
D 进程是需要运行在线程内的；

答案

1=>D

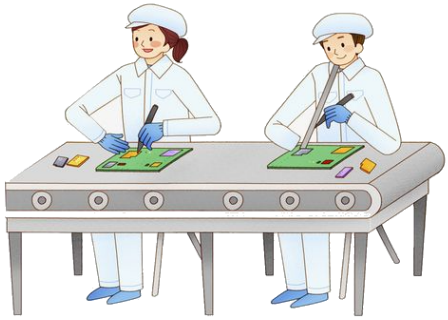
进程、线程的区别

一个故事说明进程、线程的关系



进程A， 两个线程

生产线A，两个工人



进程B， 两个线程

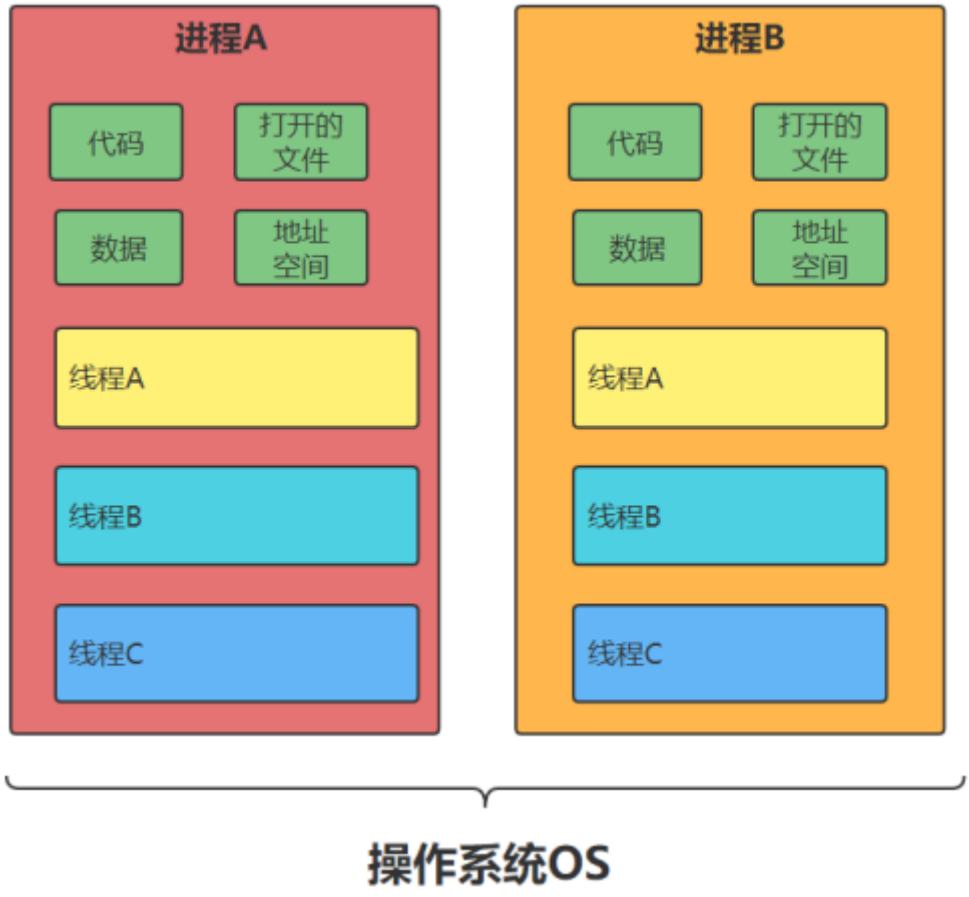
生产线B，两个工人

乔布斯想开工厂生产手机，费劲力气，制作一条生产线，这个生产线上有很多的器件以及材料。**一条生产线就是一个进程。**

只有生产线是不够的，所以找五个工人来进行生产，这个工人能够利用这些材料最终一步步的将手机做出来，**这五个工人就是五个线程。**

为了提高生产率，有两种办法：

1. 一条生产线上多招些工人，一起来做手机，这样效率是成倍增长，即单进程多线程方式
2. 多条生产线，每个生产线上多个工人，即多进程多线程



1. 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位；
2. 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线；
3. 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段、数据集、堆等)及一些进程级的资源(如打开文件和信号)，某进程内的线程在其它进程不可见；

4 调度和切换：线程上下文切换比进程上下文切换要快得多。

实时效果反馈

1. 如下进程、线程相关的概念，错误的说法是：

A 进程：拥有自己独立的堆和栈，既不共享堆，也不共享栈；

B 进程：进程由操作系统调度；

C 线程：被包含在进程之中，是进程中的实际运作单位。

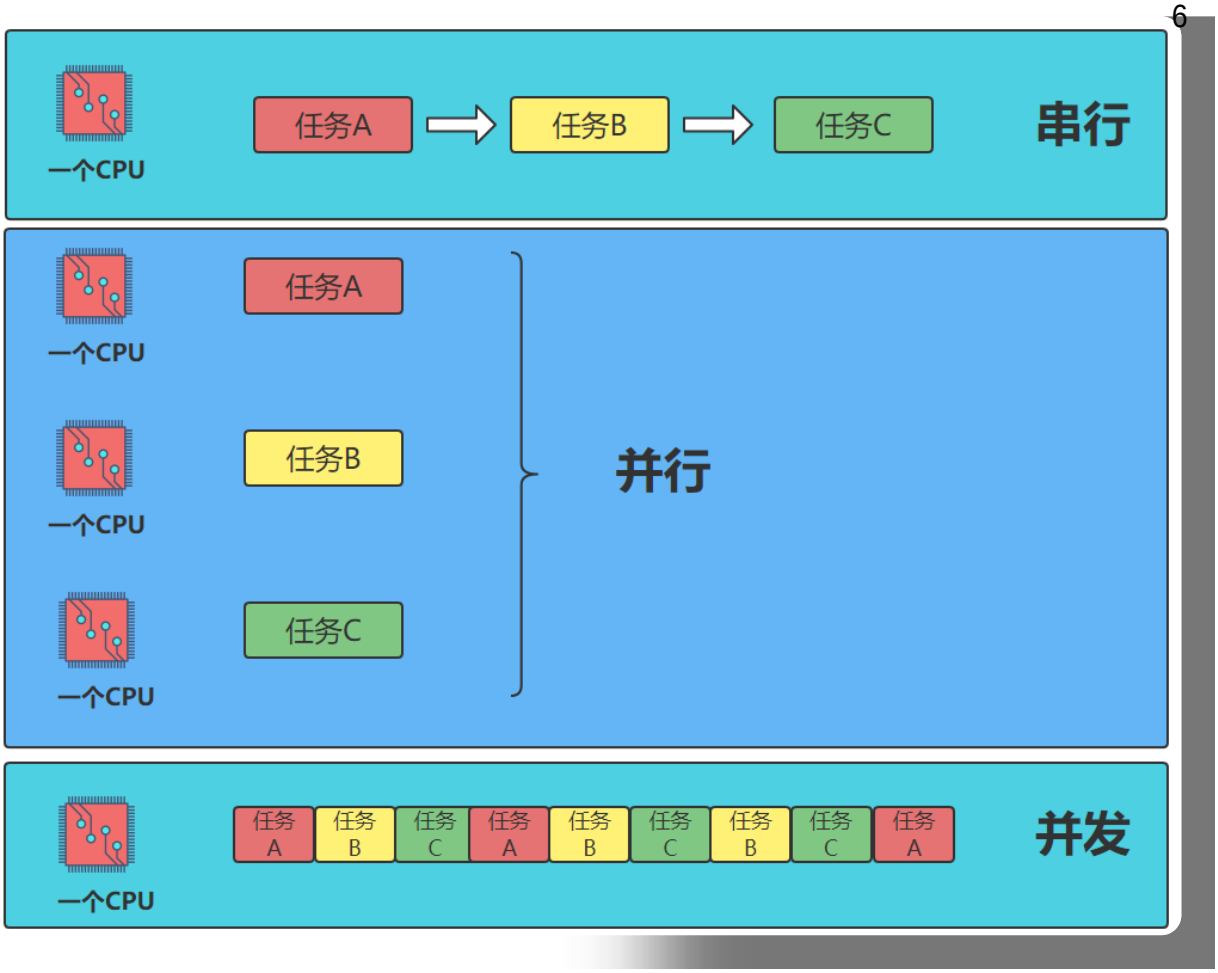
D 线程：是程序执行的最小单位，同一进程内的线程不共享内存空间；

答案

1=>D

什么是并发

并发是指在一段时间内同时做多个事情。当有多个线程在运行时,如果只有一个CPU,这种情况下计算机操作系统会采用并发技术实现并发运行，具体做法是采用“ 时间片轮询算法”，在一个时间段的线程代码运行时，其它线程处于就绪状。这种方式我们称之为并发。(Concurrent)。



1. 串行(serial)：一个CPU上，按顺序完成多个任务
2. 并行(parallelism)：指的是任务数小于等于cpu核数，即任务真的是一起执行的
3. 并发(concurrency)：一个CPU采用时间片管理方式，交替的处理多个任务。一般是是任务数多余cpu核数，通过操作系统的各种任务调度算法，实现用多个任务"一起"执行（实际上总有一些任务不在执行，因为切换任务的速度相当快，看上去一起执行而已）

实时效果反馈

1. 如下并发编程相关的概念，错误的说法是：

A 并行：每个CPU执行一个任务

B 并发：一个CPU交替执行多个任务

C 串行：一个CPU上，按顺序完成多个任务

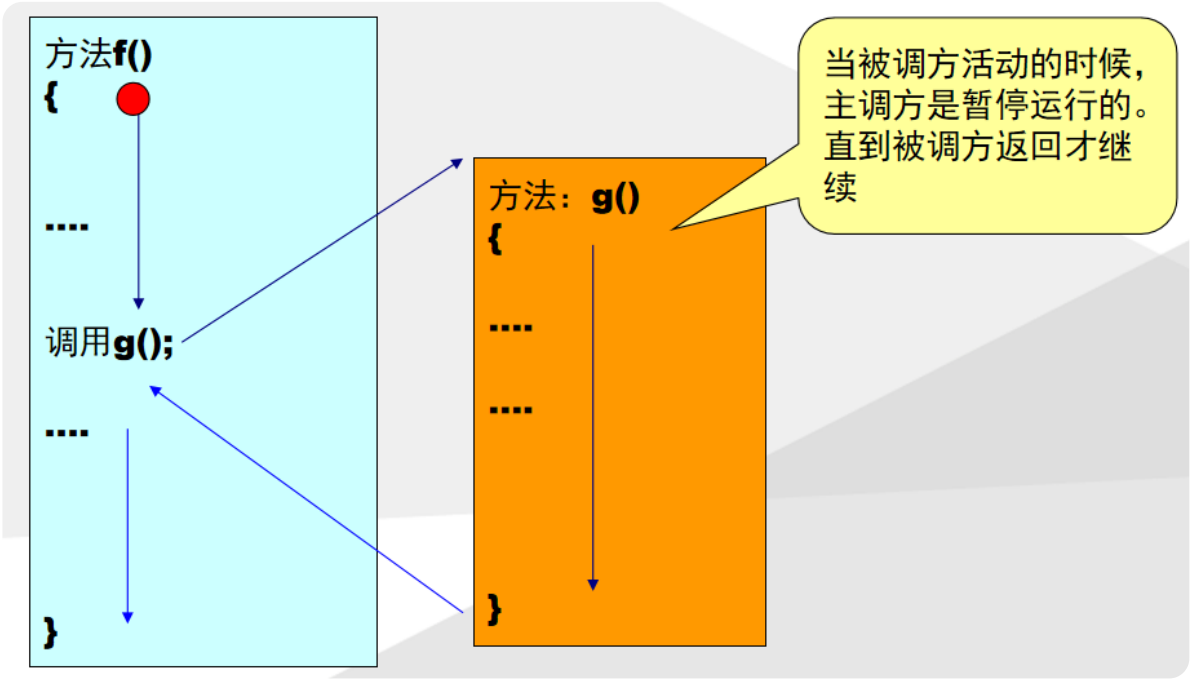
D 串行：一个任务，在多个CPU上执行

答案

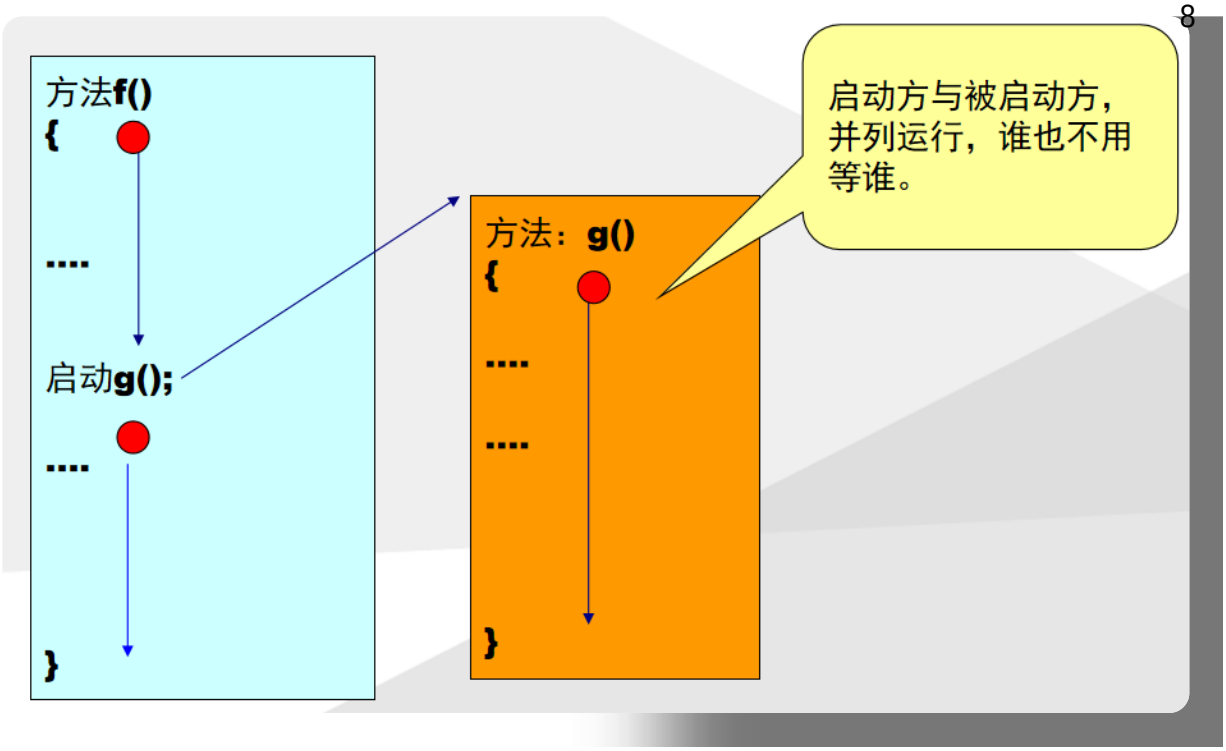
1=>D

线程的执行特点

方法的执行特点



线程的执行特点



实时效果反馈

1. 如下对线程的执行特点描述正确的是?

A 启动方会等待被启动方的执行;

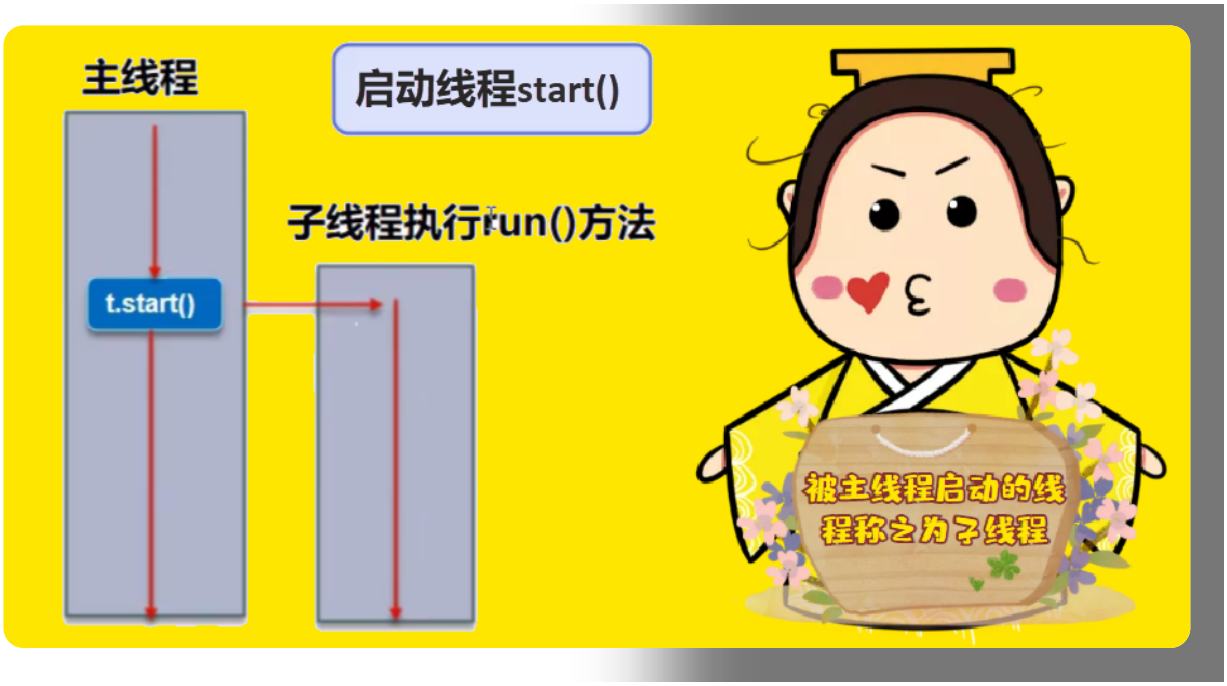
B 被启动方会等待启动方的执行;

C 启动方与被启动方并列执行;

答案

1=>C

什么是主线程以及子线程



主线程

当Java程序启动时，一个线程会立刻运行，该线程通常叫做程序的主线程（main thread），即main方法对应的线程，它是程序开始时就执行的。

Java应用程序会有一个main方法，是作为某个类的方法出现的。当程序启动时，该方法就会第一个自动的得到执行，并成为程序的主线程。也就是说，main方法是一个应用的入口，也代表了这个应用的主线程。JVM在执行main方法时,main方法会进入到栈内存,JVM会通过操作系统开辟一条main方法通向cpu的执行路径,cpu就可以通过这个路径来执行main方法,而这个路径有一个名字,叫main(主)线程

主线程的特点

它是产生其他子线程的线程。

它不一定是最后完成执行的线程，子线程可能在它结束之后还在运行。

子线程

在主线程中创建并启动的线程，一般称之为子线程。

实时效果反馈

1. java中，主线程方法名为?

A main;

B run;

C start;

D mainThread;

答案

1=>A

线程的创建

通过继承Thread类实现多线程

继承Thread类

继承Thread类实现多线程的步骤：

1. 在Java中负责实现线程功能的类是java.lang.Thread 类。

   > **此种方式的缺点：**如果我们的类已经继承了一个类（如小程序必须继承自 Applet 类），则无法再继承 Thread 类。

2. 可以通过创建 Thread的实例来创建新的线程。
3. 每个线程都是通过某个特定的Thread对象所对应的方法run( )来完成其操作的，方法run( )称为线程体。
4. 通过调用Thread类的start()方法来启动一个线程。

通过继承Thread类实现多线程

```
public class TestThread extends Thread {//自定义类继承Thread类
    //run()方法里是线程体
    public void run() {
        for (int i = 0; i < 10; i++) {
```

```
5
            System.out.println(this.getName() + ":" +
        i);//getName()方法是返回线程名称
6           }
7       }
8
9       public static void main(String[] args) {
10          TestThread thread1 = new
        TestThread();//创建线程对象
11          thread1.start();//启动线程
12          TestThread thread2 = new
        TestThread();
13          thread2.start();
14      }
15  }
```

实时效果反馈

1. 在多线程中，可以通过继承类实现多线程?

A Object;

B Arrays;

C Collections

D Thread;

答案

1=>D

通过Runnable接口实现多线程

在开发中，我们应用更多的是通过Runnable接口实现多线程。这种方式克服了继承Thread类的缺点，即在实现Runnable接口的同时还可以继承某个类。

从源码角度看，Thread类也是实现了Runnable接口。Runnable接口的源码如下：

```
public interface Runnable {
      void run();
}
```

两种方式比较看，实现Runnable接口的方式要通用一些。

通过Runnable接口实现多线程

```
public class TestThread2 implements Runnable
{//自定义类实现Runnable接口；
    //run()方法里是线程体；
    public void run() {
        for (int i = 0; i < 10; i++) {

System.out.println(Thread.currentThread().ge
tName() + ":" + i);
        }
    }
    public static void main(String[] args) {
        //创建线程对象，把实现了Runnable接口的对象
作为参数传入；
        Thread thread1 = new Thread(new
TestThread2());
        thread1.start();//启动线程；
        Thread thread2 = new Thread(new
TestThread2());
        thread2.start();
    }
}
```

实时效果反馈

1. 在多线程中，可以通过接口实现多线程?

A Comparable;

B Comparator;

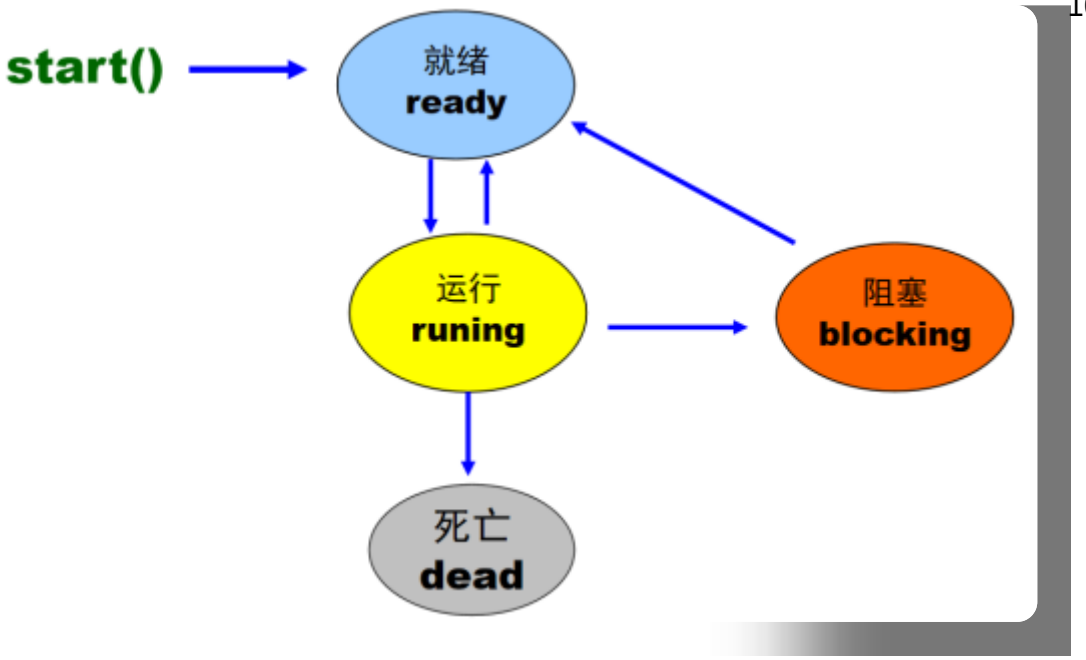
C Collection

D Runnable;

答案

1=>D

线程的执行流程



实时效果反馈

1. 如下对线程执行流程描述错误的是?

A 线程启动后会进入就绪状态;

B 当线程执行的时间片到达后会重新进入就绪状态;

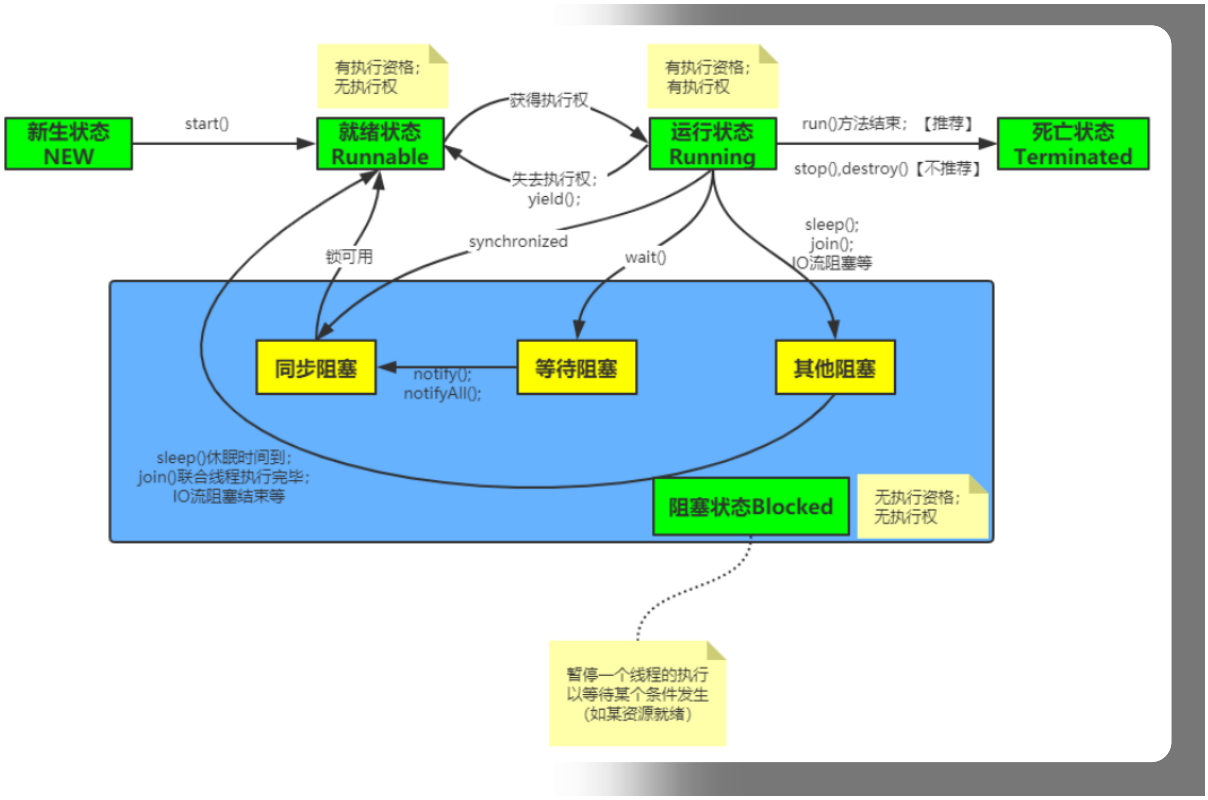
C 线程只有运行时才有可能出现阻塞状态;

D 当线程解除阻塞后会立即被运行;

答案

1=>D

线程状态和生命周期



一个线程对象在它的生命周期内，需要经历5个状态。

1. 新生状态(New)

   用new关键字建立一个线程对象后，该线程对象就处于新生状态。处于新生状态的线程有自己的内存空间，通过调用start方法进入就绪状态。

2. 就绪状态(Runnable)

   处于就绪状态的线程已经具备了运行条件，但是还没有被分配到CPU，处于“线程就绪队列”，等待系统为其分配CPU。就绪状态并不是执行状态，当系统选定一个等待执行的Thread对象后，它就会进入执行状态。一旦获得CPU，线程就进入运行状态并自动调用自己的run方法。有4种原因会导致线程进入就绪状态：

   1. 新建线程：调用start()方法，进入就绪状态；
   2. 阻塞线程：阻塞解除，进入就绪状态；
   3. 运行线程：调用yield()方法，直接进入就绪状态；
   4. 运行线程：JVM将CPU资源从本线程切换到其他线程。

3. 运行状态(Running)

在运行状态的线程执行自己run方法中的代码，直到调用其他方法而终止或等待某资源而阻塞或完成任务而死亡。如果在给定的时间片内没有执行结束，就会被系统给换下来回到就绪状态。也可能由于某些“导致阻塞的事件”而进入阻塞状态。

4. 阻塞状态(Blocked)
   阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪）。

   > 有4种原因会导致阻塞：
   >
   > 1. 执行sleep(int millsecond)方法，使当前线程休眠，进入阻塞状态。当指定的时间到了后，线程进入就绪状态。
   > 2. 执行wait()方法，使当前线程进入阻塞状态。当使用nofity()方法唤醒这个线程后，它进入就绪状态。
   > 3. 线程运行时，某个操作进入阻塞状态，比如执行IO流操作(read()/write()方法本身就是阻塞的方法)。只有当引起该操作阻塞的原因消失后，线程进入就绪状态。
   > 4. join()线程联合: 当某个线程等待另一个线程执行结束后，才能继续执行时，使用join()方法。

5. 死亡状态(Terminated)
   死亡状态是线程生命周期中的最后一个阶段。线程死亡的原因有两个。一个是正常运行的线程完成了它run()方法内的全部工作； 另一个是线程被强制终止，如通过执行stop()或destroy()方法来终止一个线程（注：stop()/destroy()方法已经被JDK废弃，不推荐使用）。
   当一个线程进入死亡状态以后，就不能再回到其它状态了。

实时效果反馈

1. 如下对于线程生命周期错误的说法是：

A 刚创建的线程是新生状态；

B 被start方法启动后进入运行状态；

C 对于时间片执行完毕的线程会进入就绪状态。

D 当线程的run方法执行完毕后线程处于死亡状态

答案

1=>B

线程的使用

终止线程的典型方式

终止线程我们一般不使用JDK提供的stop()/destroy()方法(它们本身也被JDK废弃了)。通常的做法是提供一个boolean型的终止变量，当这个变量置为false，则终止线程的运行。

终止线程的典型方法

```
1 public class StopThread implements Runnable
  {
2     private boolean flag = true;
3     @Override
```

```
4       public void run() {
5
        System.out.println(Thread.currentThread().getName()+" 线程开始");
6               int i= 0;
7               while(flag){
8
        System.out.println(Thread.currentThread().getName()+" "+i++);
9                   try {
10                      Thread.sleep(1000);
11                  } catch (InterruptedException e) {
12                      e.printStackTrace();
13                  }
14              }
15
        System.out.println(Thread.currentThread().getName()+" 线程结束");
16      }
17      public void stop(){
18          this.flag = false;
19      }
20
21      public static void main(String[] args)throws Exception {
22          System.out.println("主线程开始");
23          StopThread st = new StopThread();
24          Thread t1 = new Thread(st);
25          t1.start();
26          System.in.read();
27          st.stop();
```

```
28            System.out.println("主线程结束");
29        }
30    }
```

线程休眠

sleep()方法：可以让正在运行的线程进入阻塞状态，直到休眠时间满了，进入就绪状态。sleep方法的参数为休眠的毫秒数。

```
1  public class SleepThread implements Runnable
   {
2      @Override
3      public void run() {
4
    System.out.println(Thread.currentThread().g
   etName()+" 线程开始");
```

```
            for(int i=0;i<20;i++){

 System.out.println(Thread.currentThread().getName()+" "+i);
                try {
                    //线程休眠1秒
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

 System.out.println(Thread.currentThread().getName()+" 线程结束");
    }

    public static void main(String[] args) {
        System.out.println("主线程开始");
        Thread t = new Thread(new SleepThread());
        t.start();
        System.out.println("主线程结束");
    }
}
```

实时效果反馈

1. Thread类中的sleep方法参数的单位是?

A 毫秒;

B 秒；

C 分钟；

D 小时；

答案

1=>A

线程让步

yield()让当前正在运行的线程回到就绪状态，以允许具有相同优先级的其他线程获得运行的机会。因此，使用yield()的目的是让具有相同优先级的线程之间能够适当的轮换执行。但是，实际中无法保证yield()达到让步的目的，因为，让步的线程可能被线程调度程序再次选中。

使用yield方法时要注意的几点：

- yield是一个静态的方法。
- 调用yield后，yield告诉当前线程把运行机会交给具有相同优先级的线程。

- yield不能保证，当前线程迅速从运行状态切换到就绪状态。
- yield只能是将当前线程从运行状态转换到就绪状态，而不能是等待或者阻塞状态。

```
public class TestyieldThread implements Runnable {
    @Override
    public void run() {
        for(int i=0;i<30;i++){
            if("Thread-0".equals(Thread.currentThread().getName())){
                if(i == 0){
                    Thread.yield();
                }
            }

 System.out.println(Thread.currentThread().getName()+" "+i);
        }
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new TestyieldThread());
        Thread t2 = new Thread(new TestyieldThread());
        t1.start();
        t2.start();
    }
}
```

实时效果反馈

1. 如下对yield方法描述错误的是?

A yield是一个静态的方法;

B yield方法可以将当前线程运行状态转换为就绪状态;

C yield方法可以将当前线程运行状态转换为阻塞状态;

D yield方法把运行机会交给具有相同优先级的线程;

答案

1=>C

线程联合

当前线程邀请调用方法的线程优先执行，在调用方法的线程执行结束之前，当前线程不能再次执行。线程A在运行期间，可以调用线程B的join()方法，让线程B和线程A联合。这样，线程A就必须等待线程B执行完毕后，才能继续执行。

join方法的使用

join()方法就是指调用该方法的线程在执行完run()方法后，再执行join方法后面的代码，即将两个线程合并，用于实现同步控制。

```
class A implements Runnable{
    private Thread b;
    public A(Thread b){
        this.b = b;
    }

    @Override
    public void run() {
        for(int i=0;i<10;i++){

 System.out.println(Thread.currentThread().getName()+"  A  "+i);
            if(i == 5){
                try {
                    this.b.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            try {
                Thread.sleep(1000);
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class B implements Runnable{
    @Override
    public void run() {
        for(int i=0;i<20;i++){

 System.out.println(Thread.currentThread().getName()+" B "+i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class TestJoinThread {
    public static void main(String[] args) {
        Thread t1 = new Thread(new B());
        Thread t = new Thread(new A(t1));

        t.start();
        t1.start();
```

```
48          for(int i=0;i<10;i++){
49
 System.out.println(Thread.currentThread().getName()+" "+i);
50              if(i ==2){
51                  try {
52                      t.join();
53                  } catch (InterruptedException e) {
54                      e.printStackTrace();
55                  }
56              }
57              try {
58                  Thread.sleep(1000);
59              } catch (InterruptedException e) {
60                  e.printStackTrace();
61              }
62          }
63      }
64  }
```

实时效果反馈

1. 如下对线程联合描述错误的是?

A 当两个线程联合后，线程的执行方式为串行化执行;

B 当两个线程联合后，线程的执行方式为并行化执行;

C 通过Thread类中的join方法可以实现线程的联合;

D 在线程联合时，联合线程会等待被联合的线程执行完毕后在执行；

答案

1=>B

线程联合案例

需求：

实现爸爸让儿子买烟。

```
/**
 * 儿子买烟线程
 */
class SonThread implements Runnable{

    @Override
    public void run() {
        System.out.println("儿子出门买烟");
        System.out.println("儿子买烟需要10分钟");
        for(int i=0;i<10;i++){
            System.out.println("第"+i+"分钟");
            try {
                Thread.sleep(1000);
```

```
            } catch (InterruptedException e)
{
                e.printStackTrace();
            }
        }
        System.out.println("儿子买烟回来了");
    }
}

/**
 * 爸爸抽烟线程
 */
class FatherThread implements Runnable{

    @Override
    public void run() {
        System.out.println("爸爸想抽烟，发现烟抽
完了");
        System.out.println("爸爸让儿子去买一包红
塔山");
        Thread t = new Thread(new
SonThread());
        t.start();
        System.out.println("等待儿子买烟回来");
        try {
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.out.println("爸爸出门找儿
子");
            System.exit(1);
        }
```

```
41          System.out.println("爸爸高兴的接过烟，并把零钱给了儿子");
42      }
43  }
44
45  public class TestJoinDemo {
46      public static void main(String[] args) {
47          System.out.println("爸爸和儿子买烟的故事");
48          Thread t = new Thread(new FatherThread());
49          t.start();
50      }
51  }
```

Thread类中的其他常用方法

获取当前线程名称

方式一

this.getName()获取线程名称，该方法适用于继承Thread实现多线程方式。

```
1  class GetName1 extends Thread{
2      @Override
3      public void run() {
4          System.out.println(this.getName());
5      }
6  }
```

方式二

Thread.currentThread().getName()获取线程名称，该方法适用于实现Runnable接口实现多线程方式。

```
class GetName2 implements Runnable{

    @Override
    public void run() {

 System.out.println(Thread.currentThread().getName());
    }
}
```

设置线程的名称

方式一

通过构造方法设置线程名称。

```
class SetName1 extends Thread{
    public SetName1(String name){
        super(name);
    }
    @Override
    public void run() {
        System.out.println(this.getName());
    }
}

public class SetNameThread {
    public static void main(String[] args) {
```

```
13          SetName1 setName1 = new SetName1("SetName1");
14          setName1.start();
15      }
16  }
```

方式二

通过setName()方法设置线程名称。

```
1  class SetName2 implements Runnable{
2
3      @Override
4      public void run() {
5
 System.out.println(Thread.currentThread().getName());
6      }
7  }
8  public class SetNameThread {
9      public static void main(String[] args) {
10         Thread thread = new Thread(new SetName2());
11         thread.setName("SetName2");
12         thread.start();
13     }
14 }
```

判断线程是否存活

isAlive()方法： 判断当前的线程是否处于活动状态。

活动状态是指线程已经启动且尚未终止，线程处于正在运行或准备开始运行的状态，就认为线程是存活的。

```
class Alive implements  Runnable{

    @Override
    public void run() {
        for(int i=0;i<4;i++){

 System.out.println(Thread.currentThread().getName()+" "+i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
}

public class TestAliveThread {
    public static void main(String[] args) {
        Thread thread = new Thread(new Alive());
        thread.setName("Alive");
        thread.start();

 System.out.println(thread.getName()+" "+thread.isAlive());
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

 System.out.println(thread.getName()+" "+thread.isAlive());
    }
}
```

线程的优先级

什么是线程的优先级

每一个线程都是有优先级的，我们可以为每个线程定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程的优先级用数字表示，范围从1到10，一个线程的缺省优先级是5。

Java 的线程优先级调度会委托给操作系统去处理，所以与具体的操作系统优先级有关，如非特别需要，一般无需设置线程优先级。

> **注意**
>
> 线程的优先级，不是说哪个线程优先执行，如果设置某个线程的优先级高。那就是有可能被执行的概率高。并不是优先执行。

实时效果反馈

1. 如下对线程优先级描述错误的是?

A 线程的优先级用数字表示;

B 高优先级的线程不能保证会在低优先级的线程前执行;

C 高优先级的线程会绝对优先执行;

D 一个线程的缺省优先级是5；

答案

1=>C

线程优先级的使用

使用下列方法获得或设置线程对象的优先级。

- int getPriority();
- void setPriority(int newPriority);

> **注意**：优先级低只是意味着获得调度的概率低。并不是绝对先调用优先级高的线程后调用优先级低的线程。

```
class Priority implements Runnable{
    private int num = 0;
    private  boolean flag = true;
    @Override
    public void run() {
        while(this.flag){

 System.out.println(Thread.currentThread().getName()+" "+num++);
        }
    }
    public void stop(){
```

```
11          this.flag = false;
12      }
13  }
14
15  public class PriorityThread {
16      public static void main(String[]
    args)throws Exception {
17          Priority p1 = new Priority();
18          Priority p2 = new Priority();
19          Thread t1 = new Thread(p1,"线程1");
20          Thread t2 = new Thread(p2,"线程2");
21
     System.out.println(t1.getPriority());
22          //Thread.MAX_PRIORITY = 10
23          t1.setPriority(Thread.MAX_PRIORITY);
24          //Thread.MAX_PRIORITY = 1
25          t2.setPriority(Thread.MIN_PRIORITY);
26          t1.start();
27          t2.start();
28          Thread.sleep(1000);
29          p1.stop();
30          p2.stop();
31      }
32  }
```

守护线程

什么是守护线程

在Java中有两类线程：

- User Thread(用户线程)：就是应用程序里的自定义线程。
- Daemon Thread(守护线程)：比如垃圾回收线程，就是最典型的守护线程。

守护线程（即Daemon Thread），是一个服务线程，准确地来说就是服务其他的线程，这是它的作用，而其他的线程只有一种，那就是用户线程。

守护线程特点：

守护线程会随着用户线程死亡而死亡。

守护线程与用户线程的区别：

用户线程，不随着主线程的死亡而死亡。用户线程只有两种情况会死掉，1在run中异常终止。2正常把run执行完毕，线程死亡。

守护线程，随着用户线程的死亡而死亡，当用户线程死亡守护线程也会随之死亡。

实时效果反馈

1. 如下对守护线程描述错误的是?

A 守护线程就是辅助线程;

B 守护线程不会随着用户线程死亡而死亡;

C 守护线程会随着用户线程死亡而死亡;

D 在Java多线程中Daemon Thread表示为守护线程;

答案

1=>B

守护线程的使用

```
/**
 * 守护线程类
 */
class Daemon implements  Runnable{

    @Override
    public void run() {
        for(int i=0;i<20;i++){
```

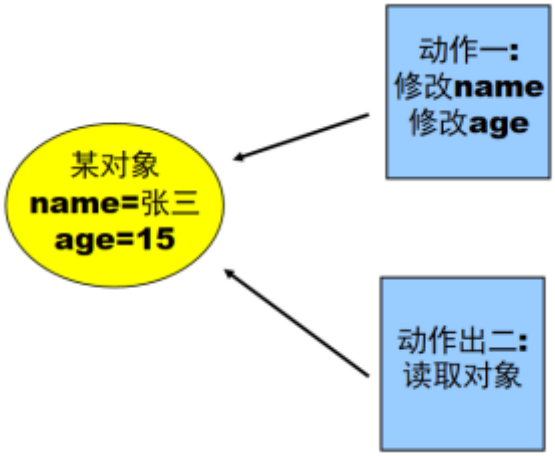
```
9
 System.out.println(Thread.currentThread().getName()+" "+i);
10              try {
11                  Thread.sleep(2000);
12              } catch (InterruptedException e) {
13                  e.printStackTrace();
14              }
15          }
16      }
17  }
18
19  class UsersThread implements Runnable{
20
21      @Override
22      public void run() {
23          Thread t = new Thread(new Daemon(),"Daemon");
24          //将该线程设置为守护线程
25          t.setDaemon(true);
26          t.start();
27          for(int i=0;i<5;i++){
28
 System.out.println(Thread.currentThread().getName()+" "+i);
29              try {
30                  Thread.sleep(500);
31              } catch (InterruptedException e) {
32                  e.printStackTrace();
33              }
```

```
34          }
35      }
36  }
37  public class DaemonThread {
38      public static void main(String[]
    args)throws Exception {
39          Thread t = new Thread(new
    UsersThread(),"UsersThread");
40          t.start();
41          Thread.sleep(1000);
42          System.out.println("主线程结束");
43      }
44  }
```

线程同步

什么是线程同步

线程冲突现象



同步问题的提出

现实生活中，我们会遇到“同一个资源，多个人都想使用”的问题。比如：教室里，只有一台电脑，多个人都想使用。天然的解决办法就是，在电脑旁边，大家排队。前一人使用完后，后一人再使用。

线程同步的概念

处理多线程问题时，多个线程访问同一个对象，并且某些线程还想修改这个对象。 这时候，我们就需要用到“线程同步”。 线程同步其实就是一种等待机制，多个需要同时访问此对象的线程进入这个对象的等待池形成队列，等待前面的线程使用完毕后，下一个线程再使用。

实时效果反馈

1. 如下对线程同步描述错误的是?

A 线程同步是为了解决线程冲突问题;

B 线程同步其实就是一种等待机制;

C 线程同步是将并行化变为串行化;

D 线程同步是将串行化变为并行化;

答案

1=>D

线程冲突案例演示

我们以银行取款经典案例来演示线程冲突现象。

银行取钱的基本流程基本上可以分为如下几个步骤。

（1）用户输入账户、密码，系统判断用户的账户、密码是否匹配。

（2）用户输入取款金额

（3）系统判断账户余额是否大于或等于取款金额

（4）如果余额大于或等于取款金额，则取钱成功；如果余额小于取款金额，则取钱失败。

```
/**
 * 账户类
 */
class Account{
    //账号
    private String accountNo;
    //账户的余额
    private double balance;

    public Account() {
    }

    public Account(String accountNo, double balance) {
        this.accountNo = accountNo;
        this.balance = balance;
    }

    public String getAccountNo() {
```

```
19          return accountNo;
20      }
21
22      public void setAccountNo(String
accountNo) {
23          this.accountNo = accountNo;
24      }
25
26      public double getBalance() {
27          return balance;
28      }
29
30      public void setBalance(double balance) {
31          this.balance = balance;
32      }
33  }
34  /**
35   * 取款线程
36   */
37  class DrawThread implements Runnable{
38      //账户对象
39      private Account account;
40      //取款金额
41      private double drawMoney;
42      public DrawThread(Account account,double
drawMoney){
43          this.account = account;
44          this.drawMoney = drawMoney;
45      }
46
47      /**
48       * 取款线程
```

```
49          */
50         @Override
51         public void run() {
52             //判断当前账户余额是否大于或等于取款金额
53             if(this.account.getBalance() >=
this.drawMoney){
54
 System.out.println(Thread.currentThread().g
etName()+" 取钱成功！吐出钞
票："+this.drawMoney);
55                 try {
56                     Thread.sleep(1000);
57                 } catch (InterruptedException e)
{
58                     e.printStackTrace();
59                 }
60                 //更新账户余额
61
 this.account.setBalance(this.account.getBal
ance()- this.drawMoney);
62                 System.out.println("\t 余额
为："+this.account.getBalance());
63             }else{
64
 System.out.println(Thread.currentThread().g
etName()+" 取钱失败，余额不足");
65             }
66         }
67 }
68
69 public class TestDrawMoneyThread {
70     public static void main(String[] args) {
```

```
71          Account account = new
    Account("1234",1000);
72          new Thread(new
    DrawThread(account,800),"老公").start();
73          new Thread(new
    DrawThread(account,800),"老婆").start();
74      }
75  }
```

实现线程同步

由于同一进程的多个线程共享同一块存储空间，在带来方便的同时，也带来了访问冲突的问题。Java语言提供了专门机制以解决这种冲突，有效避免了同一个数据对象被多个线程同时访问造成的这种问题。这套机制就是synchronized关键字。

synchronized语法结构：

```
synchronized(锁对象){
        同步代码
    }
```

synchronized关键字使用时需要考虑的问题：

- 需要对那部分的代码在执行时具有线程互斥的能力(线程互斥：并行变串行)。
- 需要对哪些线程中的代码具有互斥能力(通过synchronized锁对象来决定)。

它包括两种用法：

synchronized 方法和 synchronized 块。

- synchronized 方法
  通过在方法声明中加入 synchronized关键字来声明，语法如下：

```
public  synchronized  void accessVal(int newVal);
```

  synchronized 在方法声明时使用：放在访问控制符(public)之前或之后。这时同一个对象下synchronized方法在多线程中执行时，该方法是同步的，即一次只能有一个线程进入该方法，其他线程要想在此时调用该方法，只能排队等候，当前线程(就是在synchronized方法内部的线程)执行完该方法后，别的线程才能进入。

- synchronized块
  synchronized 方法的缺陷：若将一个大的方法声明为 synchronized 将会大大影响效率。
  Java 为我们提供了更好的解决办法，那就是 synchronized 块。块可以让我们精确地控制到具体的“成员变量”，缩小同步的范围，提高效率。

实时效果反馈

1. 如下关于synchronized的概念错误的说法是?

A synchronized是Java中的关键字;

B synchronized是用于实现线程同步的;

C synchronized可以出现在方法上的任何位置;

D synchronized的锁只能是对象类型;

答案

1=>C

修改线程冲突案例演示

```
/**
 * 账户类
 */
class Account{
    //账号
    private String accountNO;
    //账户余额
    private double balance;

    public Account() {
    }

    public Account(String accountNO, double balance) {
        this.accountNO = accountNO;
        this.balance = balance;
    }

    public String getAccountNO() {
        return accountNO;
    }

    public void setAccountNO(String accountNO) {
        this.accountNO = accountNO;
```

```
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }
}

/**
 * 取款线程
 */
class DrawThread implements Runnable{
    //账户对象
    private Account account;
    //取款金额
    private double drawMoney;
    public DrawThread(){

    }
    public DrawThread(Account account,double drawMoney){
        this.account = account;
        this.drawMoney = drawMoney;
    }

    /**
     * 取款线程体
     */
    @Override
```

```
55      public void run() {
56          synchronized (this.account){
57              //判断当前账户余额是否大于或等于取款金额
58              if(this.account.getBalance() >= this.drawMoney){
59
 System.out.println(Thread.currentThread().getName()+" 取钱成功！突出钞票"+this.drawMoney);
60                  try {
61                      Thread.sleep(1000);
62                  } catch (InterruptedException e) {
63                      e.printStackTrace();
64                  }
65                  //更新账户余额
66
 this.account.setBalance(this.account.getBalance() - this.drawMoney);
67                  System.out.println("\t 余额为："+this.account.getBalance());
68              }else{
69
 System.out.println(Thread.currentThread().getName()+" 取钱失败，余额不足");
70              }
71
72          }
73
74      }
75  }
76
```

```
77 public class TestDrawMoneyThread {
78     public static void main(String[] args) {
79         Account account = new
   Account("1234",1000);
80         new Thread(new
   DrawThread(account,800),"老公").start();
81         new Thread(new
   DrawThread(account,800),"老婆").start();
82     }
83 }
```

线程同步的使用

使用this作为线程对象锁

语法结构：

```
1 synchronized(this){
2         //同步代码
3     }
```

或

```
public  synchronized  void accessVal(int newVal){

//同步代码

}
```

```
/**
 * 定义程序员类
 */
class Programmer{
    private String name;
    public Programmer(String name){
        this.name = name;
    }
    /**
     * 打开电脑
     */
    synchronized  public  void computer(){
            try {
                System.out.println(this.name + " 接通电源");
                Thread.sleep(500);
                System.out.println(this.name + " 按开机按键");
                Thread.sleep(500);
                System.out.println(this.name + " 系统启动中");
                Thread.sleep(500);
```

```
20                  System.out.println(this.name
+ " 系统启动成功");
21              } catch (InterruptedException e)
{
22                  e.printStackTrace();
23              }
24      }
25      /**
26       * 编码
27       */
28      synchronized public void coding(){
29              try {
30                  System.out.println(this.name
+ " 双击Idea");
31                  Thread.sleep(500);
32                  System.out.println(this.name
+ " Idea启动完毕");
33                  Thread.sleep(500);
34                  System.out.println(this.name
+ " 开开心心的写代码");
35              } catch (InterruptedException e)
{
36                  e.printStackTrace();
37              }
38          }
39  }
40
41  /**
42   * 打开电脑的工作线程
43   */
44  class Working1 extends Thread{
45      private  Programmer p;
```

```
46      public Working1(Programmer p){
47          this.p = p;
48      }
49      @Override
50      public void run() {
51          this.p.computer();
52      }
53  }
54
55  /**
56   * 编写代码的工作线程
57   */
58  class Working2 extends Thread{
59      private  Programmer p;
60      public Working2(Programmer p){
61          this.p = p;
62      }
63      @Override
64      public void run() {
65          this.p.coding();
66      }
67  }
68  public class TestSyncThread {
69      public static void main(String[] args) {
70          Programmer p = new Programmer("张
    三");
71          new Working1(p).start();
72          new Working2(p).start();
73      }
74  }
```

使用字符串作为线程对象锁

语法结构：

```
synchronized(“字符串”){
        //同步代码
    }
```

```
/**
 * 定义程序员类
 */
class Programmer{
    private String name;
    public Programmer(String name){
        this.name = name;
    }
    /**
     * 打开电脑
```

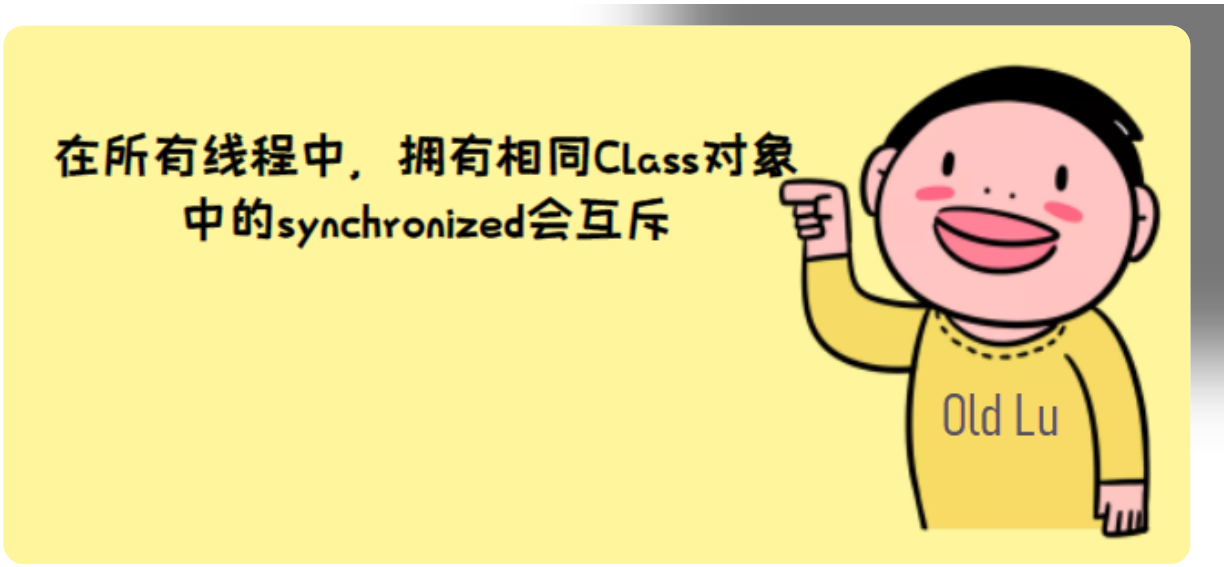
```
11      */
12     synchronized  public  void computer(){
13             try {
14 
 System.out.println(this.name + " 接通电源");
15                 Thread.sleep(500);
16 
 System.out.println(this.name + " 按开机按
键");
17                 Thread.sleep(500);
18 
 System.out.println(this.name + " 系统启动
中");
19                 Thread.sleep(500);
20 
 System.out.println(this.name + " 系统启动成
功");
21             } catch (InterruptedException
e) {
22                 e.printStackTrace();
23             }
24     }
25     /**
26      * 编码
27      */
28     synchronized public void coding(){
29             try {
30 
 System.out.println(this.name + " 双击
Idea");
31                 Thread.sleep(500);
```

```
32
 System.out.println(this.name + " Idea启动完毕");
33                  Thread.sleep(500);
34
 System.out.println(this.name + " 开开心心的写代码");
35              } catch (InterruptedException e) {
36                  e.printStackTrace();
37              }
38          }
39      /**
40       * 去卫生间
41       */
42      public void wc(){
43          synchronized ("suibian") {
44              try {
45
 System.out.println(this.name + " 打开卫生间门");
46                  Thread.sleep(500);
47
 System.out.println(this.name + " 开始排泄");
48                  Thread.sleep(500);
49
 System.out.println(this.name + " 冲水");
50                  Thread.sleep(500);
51
 System.out.println(this.name + " 离开卫生间");
```

```
52              } catch (InterruptedException
e) {
53                  e.printStackTrace();
54              }
55          }
56      }
57  }
58
59  /**
60   * 打开电脑的工作线程
61   */
62  class Working1 extends Thread{
63      private  Programmer p;
64      public Working1(Programmer p){
65          this.p = p;
66      }
67      @Override
68      public void run() {
69          this.p.computer();
70      }
71  }
72
73  /**
74   * 编写代码的工作线程
75   */
76  class Working2 extends Thread{
77      private  Programmer p;
78      public Working2(Programmer p){
79          this.p = p;
80      }
81      @Override
82      public void run() {
```

```
83          this.p.coding();
84      }
85  }
86
87  /**
88   * 去卫生间的线程
89   */
90  class WC extends Thread{
91      private  Programmer p;
92      public WC(Programmer p){
93          this.p = p;
94      }
95      @Override
96      public void run() {
97          this.p.wc();
98      }
99  }
100 public class TestSyncThread {
101     public static void main(String[] args)
    {
102         Programmer p = new Programmer("张
    三");
103         Programmer p1 = new Programmer("李
    四");
104         Programmer p2 = new Programmer("王
    五");
105         new WC(p).start();
106         new WC(p1).start();
107         new WC(p2).start();
108     }
109 }
```

使用Class作为线程对象锁



语法结构：

```
synchronized(XX.class){
        //同步代码
    }
```

或

```
synchronized public static void accessVal()
```

```
/**
 * 定义销售员工类
 */
class Sale{
    private String name;
    public Sale(String name){
```

```
7           this.name = name;
8       }
9       /**
10       * 领取奖金
11       */
12      synchronized  public static void
money(){
13              try {
14
 System.out.println(Thread.currentThread().
getName() + " 被领导表扬");
15                  Thread.sleep(500);
16
 System.out.println(Thread.currentThread().
getName() + " 拿钱");
17                  Thread.sleep(500);
18
 System.out.println(Thread.currentThread().
getName() + " 对公司表示感谢");
19                  Thread.sleep(500);
20
 System.out.println(Thread.currentThread().
getName() + " 开开心心的拿钱走人");
21              } catch (InterruptedException
e) {
22                  e.printStackTrace();
23              }
24          }
25  }
26  class Programmer{
27      private String name;
28      public Programmer(String name){
```

```
29          this.name = name;
30      }
31      /**
32       * 打开电脑
33       */
34      synchronized  public  void computer(){
35              try {
36
 System.out.println(this.name + " 接通电源");
37                  Thread.sleep(500);
38
 System.out.println(this.name + " 按开机按
键");
39                  Thread.sleep(500);
40
 System.out.println(this.name + " 系统启动
中");
41                  Thread.sleep(500);
42
 System.out.println(this.name + " 系统启动成
功");
43              } catch (InterruptedException
e) {
44                  e.printStackTrace();
45              }
46      }
47      /**
48       * 编码
49       */
50      synchronized public void coding(){
51              try {
```

```
52
 System.out.println(this.name + " 双击
Idea");
53                  Thread.sleep(500);
54
 System.out.println(this.name + " Idea启动完
毕");
55                  Thread.sleep(500);
56
 System.out.println(this.name + " 开开心心的
写代码");
57              } catch (InterruptedException
e) {
58                  e.printStackTrace();
59              }
60          }
61      /**
62       * 去卫生间
63       */
64      public void wc(){
65          synchronized ("suibian") {
66              try {
67
 System.out.println(this.name + " 打开卫生间
门");
68                  Thread.sleep(500);
69
 System.out.println(this.name + " 开始排泄");
70                  Thread.sleep(500);
71
 System.out.println(this.name + " 冲水");
72                  Thread.sleep(500);
```

```
73
 System.out.println(this.name + " 离开卫生间");
74              } catch (InterruptedException e) {
75                  e.printStackTrace();
76              }
77          }
78      }
79      /**
80       * 领取奖金
81       */
82      public void money(){
83          synchronized (Programmer.class) {
84              try {
85
 System.out.println(this.name + " 被领导表扬");
86                  Thread.sleep(500);
87
 System.out.println(this.name + " 拿钱");
88                  Thread.sleep(500);
89
 System.out.println(this.name + " 对公司表示感谢");
90                  Thread.sleep(500);
91
 System.out.println(this.name + " 开开心心的拿钱走人");
92              } catch (InterruptedException e) {
93                  e.printStackTrace();
```

```
94                  }
95              }
96          }
97      }
98
99      /**
100      * 打开电脑的工作线程
101      */
102     class Working1 extends Thread{
103         private  Programmer p;
104         public Working1(Programmer p){
105             this.p = p;
106         }
107         @Override
108         public void run() {
109             this.p.computer();
110         }
111     }
112
113     /**
114      * 编写代码的工作线程
115      */
116     class Working2 extends Thread{
117         private  Programmer p;
118         public Working2(Programmer p){
119             this.p = p;
120         }
121         @Override
122         public void run() {
123             this.p.coding();
124         }
125     }
```

```
126
127 /**
128  * 去卫生间的线程
129  */
130 class WC extends Thread{
131     private  Programmer p;
132     public WC(Programmer p){
133         this.p = p;
134     }
135     @Override
136     public void run() {
137         this.p.wc();
138     }
139 }
140
141 /**
142  * 程序员领取奖金
143  */
144 class ProgrammerMoney extends Thread{
145     private  Programmer p;
146     public ProgrammerMoney(Programmer p){
147         this.p = p;
148     }
149     @Override
150     public void run() {
151         this.p.money();
152     }
153 }
154
155 /**
156  * 销售部门领取奖金
157  */
```

```
158 class SaleMoney extends  Thread{
159     private  Sale p;
160     public SaleMoneyThread(Sale p){
161         this.p = p;
162     }
163     @Override
164     public void run() {
165         this.p.money();
166     }
167 }
168 
169 public class TestSyncThread {
170     public static void main(String[] args)
    {
171         /*  Programmer p = new Programmer("张
    三");
172         Programmer p1 = new Programmer("李
    四");
173         new ProgrammerMoney(p).start();
174         new ProgrammerMoney(p1).start();*/
175 
176         Sale s = new Sale("张晓丽");
177         Sale s1 = new Sale("王晓红");
178         new SaleMoney(s).start();
179         new SaleMoney(s1).start();
180     }
181 }
```

使用自定义对象作为线程对象锁

语法结构：

```
synchronized(自定义对象){
        //同步代码
}
```

```
/**
 * 定义销售员工类
 */
class Sale{
    private String name;
    public Sale(String name){
        this.name = name;
    }
    /**
     * 领取奖金
     */
    synchronized  public static void money(){
            try {
```

```
14
                System.out.println(Thread.currentThread().
getName() + " 被领导表扬");
15                  Thread.sleep(500);
16
                System.out.println(Thread.currentThread().
getName() + " 拿钱");
17                  Thread.sleep(500);
18
                System.out.println(Thread.currentThread().
getName() + " 对公司表示感谢");
19                  Thread.sleep(500);
20
                System.out.println(Thread.currentThread().
getName() + " 开开心心的拿钱走人");
21              } catch (InterruptedException
e) {
22                  e.printStackTrace();
23              }
24          }
25  }
26  class Programmer{
27      private String name;
28      public Programmer(String name){
29          this.name = name;
30      }
31      /**
32       * 打开电脑
33       */
34      synchronized  public  void computer(){
35              try {
```

```
36
 System.out.println(this.name + " 接通电源");
37                 Thread.sleep(500);
38
 System.out.println(this.name + " 按开机按
键");
39                 Thread.sleep(500);
40
 System.out.println(this.name + " 系统启动
中");
41                 Thread.sleep(500);
42
 System.out.println(this.name + " 系统启动成
功");
43             } catch (InterruptedException
e) {
44                 e.printStackTrace();
45             }
46     }
47     /**
48      * 编码
49      */
50     synchronized public void coding(){
51             try {
52
 System.out.println(this.name + " 双击
Idea");
53                 Thread.sleep(500);
54
 System.out.println(this.name + " Idea启动完
毕");
55                 Thread.sleep(500);
```

```
56
 System.out.println(this.name + " 开开心心的写代码");
57              } catch (InterruptedException e) {
58                  e.printStackTrace();
59              }
60          }
61      /**
62       * 去卫生间
63       */
64      public void wc(){
65          synchronized ("suibian") {
66              try {
67
 System.out.println(this.name + " 打开卫生间门");
68                  Thread.sleep(500);
69
 System.out.println(this.name + " 开始排泄");
70                  Thread.sleep(500);
71
 System.out.println(this.name + " 冲水");
72                  Thread.sleep(500);
73
 System.out.println(this.name + " 离开卫生间");
74              } catch (InterruptedException e) {
75                  e.printStackTrace();
76              }
77          }
```

```
78      }
79      /**
80       * 领取奖金
81       */
82      public void money(){
83          synchronized (Programmer.class) {
84              try {
85
 System.out.println(this.name + " 被领导表
扬");
86                  Thread.sleep(500);
87
 System.out.println(this.name + " 拿钱");
88                  Thread.sleep(500);
89
 System.out.println(this.name + " 对公司表示
感谢");
90                  Thread.sleep(500);
91
 System.out.println(this.name + " 开开心心的
拿钱走人");
92              } catch (InterruptedException
e) {
93                  e.printStackTrace();
94              }
95          }
96      }
97  }
98  class Manager{
99      private String name;
100     public Manager(String name){
101         this.name = name;
```

```
102     }
103     public String getName(){
104         return this.name;
105     }
106     /**
107      * 敬酒
108      */
109     public void cheers(String mName,String
eName){
110             try {
111                 System.out.println(mName +
" 来到 " + eName + " 面前");
112                 Thread.sleep(500);
113                 System.out.println(eName +
" 拿起酒杯");
114                 Thread.sleep(500);
115                 System.out.println(mName +
" 和 " + eName + " 干杯");
116             } catch (InterruptedException
e) {
117                 e.printStackTrace();
118             }
119     }
120 }
121 /**
122  * 打开电脑的工作线程
123  */
124 class Working1 extends Thread{
125     private  Programmer p;
126     public Working1(Programmer p){
127         this.p = p;
128     }
```

```
129     @Override
130     public void run() {
131         this.p.computer();
132     }
133 }
134 
135 /**
136  * 编写代码的工作线程
137  */
138 class Working2 extends Thread{
139     private  Programmer p;
140     public Working2(Programmer p){
141         this.p = p;
142     }
143     @Override
144     public void run() {
145         this.p.coding();
146     }
147 }
148 
149 /**
150  * 去卫生间的线程
151  */
152 class WC extends Thread{
153     private  Programmer p;
154     public WC(Programmer p){
155         this.p = p;
156     }
157     @Override
158     public void run() {
159         this.p.wc();
160     }
```

```
161 }
162
163 /**
164  * 程序员领取奖金
165  */
166 class ProgrammerMoney extends Thread{
167     private  Programmer p;
168     public ProgrammerMoney(Programmer p){
169         this.p = p;
170     }
171     @Override
172     public void run() {
173         this.p.money();
174     }
175 }
176
177 /**
178  * 销售部门领取奖金
179  */
180 class SaleMoneyThread extends  Thread{
181     private  Sale p;
182     public SaleMoneyThread(Sale p){
183         this.p = p;
184     }
185     @Override
186     public void run() {
187         this.p.money();
188     }
189 }
190
191 /**
192  * 敬酒线程类
```

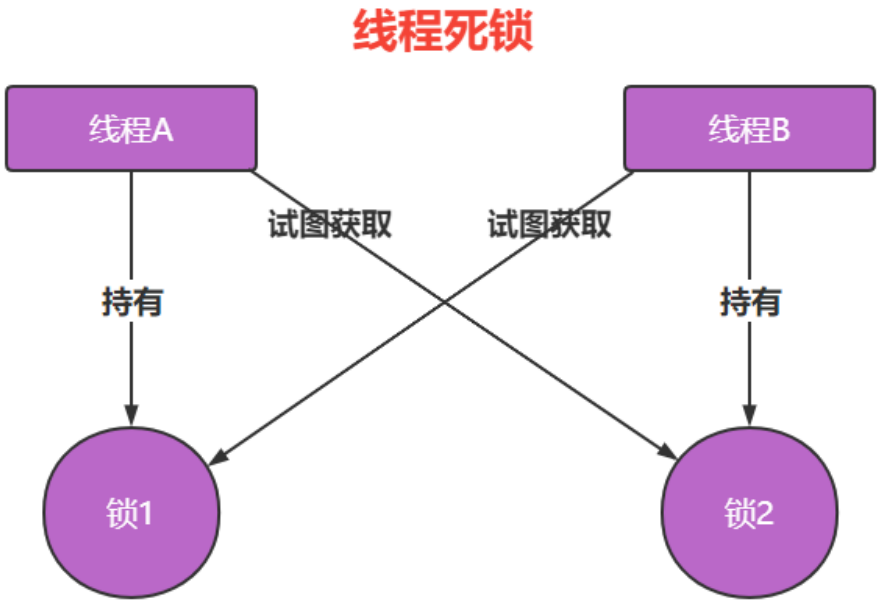
```
 */
class CheersThread extends Thread{
    private Manager manager;
    private String name;
    public CheersThread(String name,Manager manager){
        this.name = name;
        this.manager = manager;
    }
    @Override
    public void run() {
        synchronized (this.manager) {
         this.manager.cheers(this.manager.getName(), name);
        }
    }
}

public class TestSyncThread {
    public static void main(String[] args) {
        Manager manager = new Manager("张三丰");
        new CheersThread("张三",manager).start();
        new CheersThread("李四",manager).start();

    }
}
```

死锁及解决方案

死锁的概念



“死锁”指的是：

多个线程各自占有一些共享资源，并且互相等待其他线程占有的资源才能进行，而导致两个或者多个线程都在等待对方释放资源，都停止执行的情形。

> 某一个同步块需要同时拥有“两个以上对象的锁”时，就可能会发生“死锁”的问题。比如，“化妆线程”需要同时拥有“镜子对象”、“口红对象”才能运行同步块。那么，实际运行时，“小丫的化妆线程”拥有了“镜子对象”，“大丫的化妆线程”拥有了“口红对象”，都在互相等待对方释放资源，才能化妆。这样，两个线程就形成了互相等待，无法继续运行的“死锁状态”。

死锁案例演示

```
/**
 * 口红类
 */
class Lipstick{

}

/**
 * 镜子类
 */
class Mirror{

}

/**
 * 化妆线程类
 */
class Makeup extends Thread{
    private int flag; //flag=0:拿着口红。flag!=0:拿着镜子
```

```
20      private String girlName;
21      static Lipstick lipstick = new
Lipstick();
22      static Mirror mirror = new Mirror();
23
24      public Makeup(int flag,String girlName){
25          this.flag  = flag;
26          this.girlName = girlName;
27      }
28
29      @Override
30      public void run() {
31          this.doMakeup();
32      }
33      /**
34       * 开始化妆
35       */
36      public void doMakeup(){
37          if(flag == 0){
38              synchronized (lipstick){
39
 System.out.println(this.girlName+" 拿着口
红");
40                  try {
41                      Thread.sleep(1000);
42                  } catch
(InterruptedException e) {
43                      e.printStackTrace();
44                  }
45                  synchronized (mirror){
```

```
46
 System.out.println(this.girlName+" 拿着镜子");
47                  }
48              }
49          }else{
50              synchronized (mirror){
51
 System.out.println(this.girlName+" 拿着镜子");
52                  try {
53                      Thread.sleep(2000);
54                  } catch (InterruptedException e) {
55                      e.printStackTrace();
56                  }
57                  synchronized (lipstick){
58
 System.out.println(this.girlName+" 拿着口红");
59                  }
60              }
61          }
62      }
63  }
64
65  public class DeadLockThread {
66      public static void main(String[] args) {
67          new Makeup(0,"大丫").start();
68          new Makeup(1,"小丫").start();
69      }
70  }
```

死锁问题的解决

死锁是由于“同步块需要同时持有多个对象锁造成”的，要解决这个问题，思路很简单，就是：同一个代码块，不要同时持有两个对象锁。

```
/**
 * 口红类
 */
class Lipstick{

}

/**
 * 镜子类
 */
class Mirror{

}

/**
 * 化妆线程类
 */
class Makeup extends Thread{
    private int flag; //flag=0:拿着口红。flag!=0:拿着镜子
    private String girlName;
    static Lipstick lipstick = new Lipstick();
    static Mirror mirror = new Mirror();
```

```
23
24      public void setFlag(int flag) {
25          this.flag = flag;
26      }
27
28      public void setGirlName(String girlName)
{
29          this.girlName = girlName;
30      }
31
32      @Override
33      public void run() {
34          this.doMakeup();
35      }
36      /**
37       * 开始化妆
38       */
39      public void doMakeup(){
40          if(flag == 0){
41              synchronized (lipstick){
42
 System.out.println(this.girlName+" 拿着口
红");
43                  try {
44                      Thread.sleep(1000);
45                  } catch
(InterruptedException e) {
46                      e.printStackTrace();
47                  }
48              }
49              synchronized (mirror){
```

```
50
 System.out.println(this.girlName+" 拿着镜
子");
51              }
52          }else{
53              synchronized (mirror){
54
 System.out.println(this.girlName+" 拿着镜
子");
55                  try {
56                      Thread.sleep(2000);
57                  } catch
(InterruptedException e) {
58                      e.printStackTrace();
59                  }
60              }
61              synchronized (lipstick){
62
 System.out.println(this.girlName+" 拿着口
红");
63              }
64          }
65      }
66  }
67
68  public class DeadLockThread {
69      public static void main(String[] args) {
70          Makeup makeup = new Makeup();
71          makeup.setFlag(0);
72          makeup.setGirlName("大丫");
73          Makeup makeup1 = new Makeup();
74          makeup1.setFlag(1);
```

```
75          makeup1.setGirlName("小丫");
76          makeup.start();
77          makeup1.start();
78      }
79  }
```

死锁问题的解决

死锁是由于“同步块需要同时持有多个对象锁造成”的，要解决这个问题，思路很简单，就是：同一个代码块，不要同时持有两个对象锁。

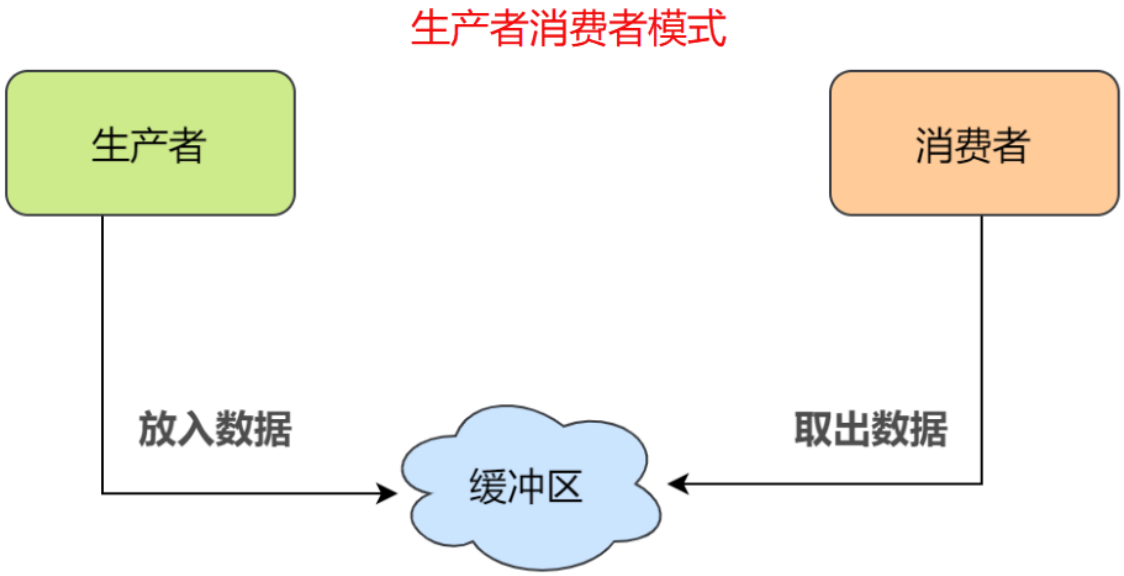
```
1  /**
2   * 口红类
3   */
4  class Lipstick{
5
6  }
7
8  /**
9   * 镜子类
10  */
11 class Mirror{
12
13 }
14
15 /**
16  * 化妆线程类
17  */
18 class Makeup extends Thread{
19     private int flag; //flag = 0 :拿着口红, flag != 0 :拿着镜子
```

```
20      private String girlName;
21      static Lipstick lipstick = new
Lipstick();
22      static Mirror mirror = new Mirror();
23
24      public Makeup(int flag,String girlName){
25          this.flag = flag;
26          this.girlName = girlName;
27      }
28
29      @Override
30      public void run() {
31          this.doMakeup();
32      }
33
34      /**
35       * 开始化妆
36       */
37      public void doMakeup(){
38
39          if(this.flag == 0){
40              synchronized (lipstick){
41
 System.out.println(this.girlName+" 拿着口
红");
42                  try {
43                      Thread.sleep(1000);
44                  } catch
(InterruptedException e) {
45                      e.printStackTrace();
46                  }
47              }
```

```
48              synchronized (mirror){
49
 System.out.println(this.girlName+" 拿着镜
子");
50              }
51          }else{
52              synchronized (mirror){
53
 System.out.println(this.girlName+" 拿着镜
子");
54                  try {
55                      Thread.sleep(2000);
56                  } catch
(InterruptedException e) {
57                      e.printStackTrace();
58                  }
59
60              }
61
62              synchronized (lipstick){
63
 System.out.println(this.girlName+" 拿着口
红");
64              }
65          }
66
67      }
68  }
69
70
71  public class DeadLockThread {
72      public static void main(String[] args) {
```

```
73              new Makeup(0,"小丫").start();
74              new Makeup(1,"大丫").start();
75          }
76      }
```

线程并发协作(生产者/消费者模式)



多线程环境下，我们经常需要多个线程的并发和协作。这个时候，就需要了解一个重要的多线程并发协作模型“生产者/消费者模式”。

角色介绍

- **什么是生产者?**
  生产者指的是负责生产数据的模块（这里模块可能是：方法、对象、线程、进程）。
- **什么是消费者?**
  消费者指的是负责处理数据的模块（这里模块可能是：方法、对象、线程、进程）。
- **什么是缓冲区?**

消费者不能直接使用生产者的数据，它们之间有个“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”拿要处理的数据。

缓冲区是实现并发的核心，缓冲区的设置有两个好处：

1. 实现线程的并发协作
   有了缓冲区以后，生产者线程只需要往缓冲区里面放置数据，而不需要管消费者消费的情况；同样，消费者只需要从缓冲区拿数据处理即可，也不需要管生产者生产的情况。 这样，就从逻辑上实现了“生产者线程”和“消费者线程”的分离，解除了生产者与消费者之间的耦合。

2. 解决忙闲不均，提高效率
   生产者生产数据慢时，缓冲区仍有数据，不影响消费者消费；消费者处理数据慢时，生产者仍然可以继续往缓冲区里面放置数据。

实现生产者与消费者模式

创建缓冲区

```
/**
 * 定义馒头类
 */
class ManTou{
    private int id;
    public ManTou(int id){
        this.id = id;
    }
```

```
9       public int getId(){
10          return this.id;
11      }
12  }
13
14  /**
15   * 定义缓冲区类
16   */
17  class SyncStack{
18      //定义存放馒头的盒子
19      private ManTou[] mt = new ManTou[10];
20      //定义操作盒子的索引
21      private int index;
22
23      /**
24       * 放馒头
25       */
26      public synchronized void push(ManTou
    manTou){
27          //判断盒子是否已满
28          while(this.index == this.mt.length){
29              try {
30                  /**
31                   * 语法：wait(),该方法必须要在
    synchronized块中调用。
32                   * wait执行后，线程会将持有的对象
    锁释放，并进入阻塞状态，
33                   * 其他需要该对象锁的线程就可以继
    续运行了。
34                   */
35                  this.wait();
```

```
36              } catch (InterruptedException e)
{
37                  e.printStackTrace();
38              }
39          }
40          //唤醒取馒头的线程
41          /**
42           * 语法：该方法必须要在synchronized块中调
用。
43           * 该方法会唤醒处于等待状态队列中的一个线
程。
44           */
45          this.notify();
46          this.mt[this.index] = manTou;
47          this.index++;
48      }
49      /**
50       * 取馒头
51       */
52      public synchronized ManTou pop(){
53          while(this.index == 0){
54              try {
55                  /**
56                   * 语法：wait(),该方法必须要在
synchronized块中调用。
57                   * wait执行后，线程会将持有的对象
锁释放，并进入阻塞状态，
58                   * 其他需要该对象锁的线程就可以继
续运行了。
59                   */
60                  this.wait();
```

```
61                 } catch (InterruptedException e)
{
62                     e.printStackTrace();
63                 }
64             }
65             this.notify();
66             this.index--;
67             return this.mt[this.index];
68         }
69 }
70
71 public class TestProduceThread {
72     public static void main(String[] args) {
73
74     }
75 }
```

创建生产者消费者线程

```
1  /**
2   * 定义馒头类
3   */
4  class ManTou{
5      private int id;
6      public ManTou(int id){
7          this.id = id;
8      }
9      public int getId(){
10         return this.id;
11     }
```

```
12 }
13
14 /**
15  * 定义缓冲区类
16  */
17 class SyncStack{
18     //定义存放馒头的盒子
19     private ManTou[] mt = new ManTou[10];
20     //定义操作盒子的索引
21     private int index;
22
23     /**
24      * 放馒头
25      */
26     public synchronized void push(ManTou
manTou){
27         //判断盒子是否已满
28         while(this.index == this.mt.length)
{
29             try {
30                 /**
31                  * 语法：wait(),该方法必须要在
synchronized块中调用。
32                  * wait执行后，线程会将持有的对
象锁释放，并进入阻塞状态，
33                  * 其他需要该对象锁的线程就可以继
续运行了。
34                  */
35                 this.wait();
36             } catch (InterruptedException
e) {
37                 e.printStackTrace();
```

```
38                 }
39             }
40             //唤醒取馒头的线程
41             /**
42              * 语法：该方法必须要在synchronized块中调用。
43              * 该方法会唤醒处于等待状态队列中的一个线程。
44              */
45             this.notify();
46             this.mt[this.index] = manTou;
47             this.index++;
48         }
49         /**
50          * 取馒头
51          */
52         public synchronized ManTou pop(){
53             while(this.index == 0){
54                 try {
55                     /**
56                      * 语法：wait(),该方法必须要在synchronized块中调用。
57                      * wait执行后，线程会将持有的对象锁释放，并进入阻塞状态，
58                      * 其他需要该对象锁的线程就可以继续运行了。
59                      */
60                     this.wait();
61                 } catch (InterruptedException e) {
62                     e.printStackTrace();
63                 }
```

```
64          }
65          this.notify();
66          this.index--;
67          return this.mt[this.index];
68      }
69  }
70
71  /**
72   * 定义生产者线程类
73   */
74  class ShengChan extends Thread{
75      private SyncStack ss;
76      public ShengChan(SyncStack ss){
77          this.ss = ss;
78      }
79      @Override
80      public void run() {
81          for(int i=0;i<10;i++){
82              System.out.println("生产馒
    头: "+i);
83              ManTou manTou = new ManTou(i);
84              this.ss.push(manTou);
85          }
86      }
87  }
88
89  /**
90   * 定义消费者线程类
91   */
92  class XiaoFei extends Thread{
93      private SyncStack ss;
94      public XiaoFei(SyncStack ss){
```

```
95          this.ss = ss;
96      }
97      @Override
98      public void run() {
99          for(int i=0;i<10;i++){
100             ManTou manTou = this.ss.pop();
101              System.out.println("消费馒
    头: "+i);
102         }
103     }
104 }
105 public class ProduceThread {
106     public static void main(String[] args)
    {
107         SyncStack ss = new SyncStack();
108         new ShengChan(ss).start();
109         new XiaoFei(ss).start();
110     }
111 }
```

线程并发协作总结

线程并发协作（也叫线程通信）

生产者消费者模式：

1. 生产者和消费者共享同一个资源，并且生产者和消费者之间相互依赖，互为条件。
2. 对于生产者，没有生产产品之前，消费者要进入等待状态。而生产了产品之后，又需要马上通知消费者消费。
3. 对于消费者，在消费之后，要通知生产者已经消费结束，需要继续生产新产品以供消费。

4. 在生产者消费者问题中，仅有synchronized是不够的。synchronized可阻止并发更新同一个共享资源，实现了同步但是synchronized不能用来实现不同线程之间的消息传递（通信）。
5. 那线程是通过哪些方法来进行消息传递（通信）的呢？见如下总结：

方法名	作 用
final void wait()	表示线程一直等待，直到得到其它线程通知
void wait(long timeout)	线程等待指定毫秒参数的时间
final void wait(long timeout,int nanos)	线程等待指定毫秒、微秒的时间
final void notify()	唤醒一个处于等待状态的线程
final void notifyAll()	唤醒同一个对象上所有调用wait()方法的线程，优先级别高的线程优先运行

6. 以上方法均是java.lang.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。

> **OldLu建议**
>
> 在实际开发中，尤其是“架构设计”中，会大量使用这个模式。对于初学者了解即可，如果晋升到中高级开发人员，这就是必须掌握的内容。