

# day18 多线程

---

- 学习目标
  - 线程概念
  - Java实现多线程程序一
  - Thread类的方法
  - Java实现多线程程序二
  - 线程安全问题
  - 同步synchronized使用
  - 锁对象的选择
  - 死锁案例
  - 生产者与消费者
  - JDK5特性JUC
  - 单例模式
  - 关键字volatile
  - 线程池
  - ConcurrentHashMap

## 1. 线程的基本概念

---

### 1.1 进程

任何的软件存储在磁盘中,运行软件的时候,OS使用IO技术,将磁盘中的软件的文件加载到内存,程序在能运行。

**进程的概念：** 应用程序(typerpa,word,IDEA)运行的时候进入到内存,程序在内存中占用的内存空间(进程)。

### 1.2 线程

线程(Thread): 在内存和CPU之间,建立一条连接通路,CPU可以到内存中取出数据进行计算,这个连接的通路,就是线程。

一个内存资源：一个独立的进程,进程中可以开启多个线程 (多条通路)

并发: 同一个时刻多个线程同时操作了同一个数据

并行: 同一个时刻多个线程同时执行不同的程序

## 2. Java实现线程程序

---

今天之前的所有程序都有一个共性：main启动之后,一条线走到底 (单线程)

### 2.1 java.lang.Thread类

一切都是对象,线程也是对象,Thread类是线程对象的描述类

- 实现线程程序的步骤：
  - 定义类继承Thread
  - 子类重写方法run
  - 创建子类对象

- 调用子类对象的方法start()启动线程

```

1  //- 定义类继承Thread
2  //- 子类重写方法run
3  public class SubThread extends Thread {
4      public void run(){
5          for(int x = 0 ; x < 50 ;x++)
6              System.out.println("run..." +x);
7      }
8  }

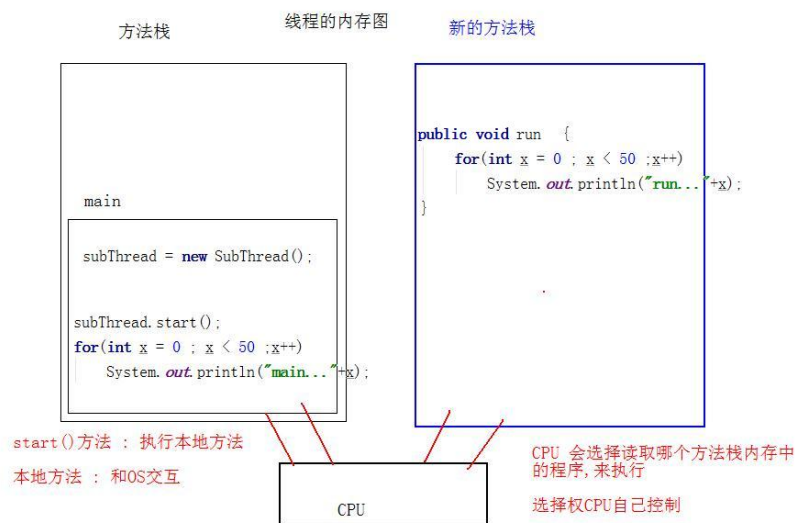
```

```

1  public static void main(String[] args) {
2      //创建线程程序
3      SubThread subThread = new SubThread();
4      //调用子类对象的方法start()启动线程
5      //启动线程,JVM调用方法run
6      subThread.start();
7      for(int x = 0 ; x < 50 ;x++)
8          System.out.println("main..." +x);
9  }

```

## 2.2 线程的内存图



## 2.3 Thread类方法

- Thread类的方法 getName()返回线程的名字,返回值是String类型

```

1  public class ThreadName extends Thread {
2      public void run (){
3          System.out.println("线程名字:: " + super.getName());
4      }
5  }

```

```

1 public static void main(String[] args) {
2     ThreadName threadName = new ThreadName();
3     //threadName.setName("旺财");
4     threadName.start();
5
6     ThreadName threadName1 = new ThreadName();
7     //threadName1.setName("小强");
8     threadName1.start();
9 }

```

- Thread类静态方法 : Thread.currentThread()
  - 静态调用,作用是放回当前的线程对象
  - "当前", 当今皇上. 本地主机

```

1 //获取当前线程对象,拿到运行main方法的线程对象
2 Thread thread = Thread.currentThread();
3 System.out.println("name::"+thread.getName());

```

- Thread类的方法 join()
  - 解释,执行join()方法的线程,他不结束,其它线程运行不了

```

1 public static void main(String[] args) throws InterruptedException {
2     JoinThread t0 = new JoinThread();
3     JoinThread t1 = new JoinThread();
4
5     t0.start();
6     t0.join();
7     t1.start();
8 }

```

- Thread类的方法 static yield()
  - 线程让步,线程把执行权让出

```

1 public void run() {
2     for(int x = 0 ; x < 50 ;x++){
3         Thread.yield();
4         System.out.println(Thread.currentThread().getName()+"x.." +x);
5     }
6 }

```

## 3. Java实现线程程序

### 3.1 java.lang.Runnable接口

- 实现线程程序的步骤 :
  - 定义类实现接口
  - 重写接口的抽象方法run()
  - 创建Thread类对象
    - Thread类构造方法中,传递Runnable接口的实现类对象

- 调用Thread对象方法start()启动线程

```
1  //- 定义类实现接口
2  //- 重写接口的抽象方法run()
3  public class SubRunnable implements Runnable{
4      @Override
5      public void run() {
6          for(int x = 0 ; x < 50 ;x++){
7              System.out.println(Thread.currentThread().getName()+"x.." +x);
8          }
9      }
10 }
11
```

```
1      public static void main(String[] args) {
2          //创建接口实现类对象
3          Runnable r = new SubRunnable();
4          //创建Thread对象,构造方法传递接口实现类
5          Thread t0 = new Thread(r);
6          t0.start();
7
8          for(int x = 0 ; x < 50 ;x++){
9              System.out.println(Thread.currentThread().getName()+"x.." +x);
10         }
11     }
```

## 3.2 实现接口的好处

接口实现好处是设计上的分离效果：线程要执行的任务和线程对象本身是分离的。

继承Thread重写方法run(): Thread是线程对象,run()是线程要执行的任务

实现Runnable接口：方法run在实现类,和线程无关,创建Thread类传递接口的实现类对象,线程的任务和Thread没有联系, 解开耦合性

## 4. 线程安全

出现线程安全的问题需要一个前提：多个线程同时操作同一个资源

线程执行调用方法run,同一个资源是堆内存的

### 4.1 售票例子

火车票的票源是固定的,购买渠道在火车站买,n多个窗口

```
1
2  /**
3   * 票源对象,需要多个线程同时操作
4   */
5  public class Ticket implements Runnable {
6
7      //定义票源
8      private int tickets = 100;
9  }
```

```

10     @Override
11     public void run() {
12         while (true) {
13             if (tickets > 0) {
14                 try {
15                     Thread.sleep(10); //线程休眠, 暂停执行
16                 } catch (Exception ex) {}
17                 System.out.println(Thread.currentThread().getName() + " 出售第"
+ tickets + "张");
18                 tickets--;
19             } else
20                 break;;
21         }
22     }
23 }

```

```

1  public static void main(String[] args) {
2      Ticket ticket = new Ticket();
3      //创建3个窗口, 3个线程
4      Thread t0 = new Thread(ticket);
5      Thread t1 = new Thread(ticket);
6      Thread t2 = new Thread(ticket);
7
8      t0.start();
9      t1.start();
10     t2.start();
11 }

```

解决线程的安全问题：当一个线程没有完成全部操作的时候, 其它线程不能操作

```

private int tickets = 1 ;

public void run() {
    while (true) {
        if (tickets > 0) {
            try {
                Thread.sleep(10); //线程休眠, 暂停执行
            } catch (Exception ex) {}
            System.out.println(Thread.currentThread().getName() + " 出售第" + tickets + "张");
            tickets--;
        } else
            break;;
    }
}

```



线程的安全问题：多线程操作一个资源, 有可能出现安全问题  
出现的原因：当一个线程还没有操作完成, 另一个线程就开始操作了

## 4.2 同步代码块

同步代码块可以解决线程安全问题：格式 synchronized关键字

```

1  synchronized(任意对象){
2      //线程操作的共享资源
3  }

```

任意对象：在同步中这个对象称为对象锁, 简称锁, 官方的稳定称为 对象监视器

同步代码块,如何保证线程的安全性.

- 同步代码块的执行原理: 关键点就是对象锁
  - 线程执行到同步,判断锁是否存在
    - 如果锁存在,获取到锁,进入到同步中执行
    - 执行完毕,线程出去同步代码块,讲锁对象归还
  - 线程执行到同步,判断锁所否存在
    - 如果锁不存在,线程只能在同步代码块这里等待,锁的到来

```
private int tickets = 1 ;
public void run() {
    while (true) {
        synchronized (object) {
            if (tickets > 0) {
                try {
                    Thread.sleep(20); //线程休眠, 暂停执行
                } catch (Exception ex) {}
                System.out.println(Thread.currentThread().getName() + " 出售第" + tickets + "张");
                tickets--;
            }
        }
    }
}
```

线程执行到同步的时候: 线程判断同步中的锁还有没有  
线程获取这把锁,带着锁进入到同步的代码块中  
执行同步中的程序,当线程出去同步代码块,锁要归还

使用同步: 线程要先判断锁,然后获取锁,出去同步要释放锁,增加了许多步骤,因此线程安全运行速度慢. 牺牲性能,不能牺牲数据安全

## 4.3 同步方法

当一个方法中,所有代码都是线程操作的共享内容,可以在方法的定义上添加同步的关键字 `synchronized`, 同步的方法,或者称为同步的函数.

- 同步方法中有对象锁吗, `this`对象
- 静态同步方法中有对象锁吗,锁对象是本类.class属性. 这个属性表示这个类的class文件的对象.

```
1  @Override
2  public void run() {
3      while (true)
4          sale();
5  }
6
7  private static synchronized void sale(){
8      // synchronized (Ticket.class) {
9      if (tickets > 0) {
10         try {
11             Thread.sleep(20); //线程休眠, 暂停执行
12         } catch (Exception ex) {}
13     }
14     System.out.println(Thread.currentThread().getName() + " 出售第" + tickets
15 + "张");
16     tickets--;
17 }
18 }
```

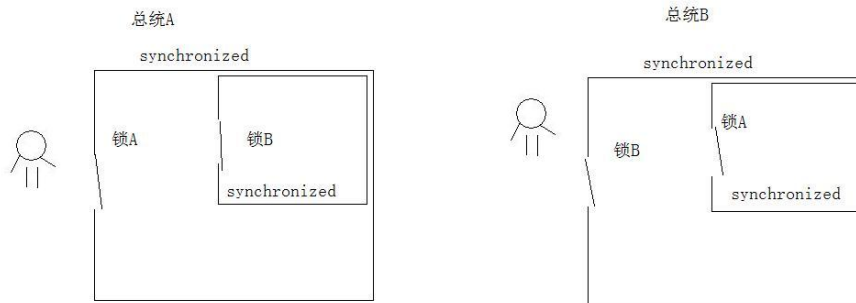
## 5. 死锁

死锁程序：多个线程同时争夺同一个锁资源,出现程序的假死现象.

面试点：考察开发人员是否充分理解同步代码的执行原理

同步代码块：线程判断锁,获取锁,释放锁,不出代码,锁不释放

完成死锁的案例：同步代码块的嵌套



- 死锁代码

```
1  /**
2   * 实现死锁程序
3   */
4  public class ThreadDeadLock implements Runnable{
5
6      private boolean flag ;
7
8      public ThreadDeadLock(boolean flag){
9          this.flag = flag;
10     }
11
12     @Override
13     public void run() {
14         while (true){
15             //同步代码块的嵌套
16             if (flag){
17                 //先进入A锁同步
18                 synchronized (LockA.lockA){
19                     System.out.println("线程获取A锁");
20                     //在进入另一个同步B锁
21                     synchronized (LockB.lockB){
22                         System.out.println("线程获取B锁");
23                     }
24                 }
25             }else {
26                 //先进入B锁同步
27                 synchronized (LockB.lockB){
28                     System.out.println("线程获取B锁");
29                     //再进入另一个同步锁A锁
```

```

30         synchronized (LockA.lockA){
31             System.out.println("线程获取A锁");
32         }
33     }
34 }
35 }
36 }
37 }

```

```

1 public class LockA {
2     public static LockA lockA = new LockA();
3 }

```

```

1 public class LockB {
2     public static LockB lockB = new LockB();
3 }

```

```

1 public static void main(String[] args) {
2     ThreadDeadLock threadDeadLock = new ThreadDeadLock(true);
3     ThreadDeadLock threadDeadLock2 = new ThreadDeadLock(false);
4
5     new Thread(threadDeadLock).start();
6     new Thread(threadDeadLock2).start();
7 }

```

## 6. JDK5新特性Lock锁

JDK5新的特性 : java.util.concurrent.locks包. 定义了接口Lock.

Lock接口替代了synchronized,可以更加灵活

- Lock接口的方法
  - void lock() 获取锁
  - void unlock()释放锁
- Lock接口的实现类ReentrantLock

```

1 /**
2  * 优化为juc包的接口Lock
3  */
4 public class Ticket implements Runnable {
5
6     //定义票源
7     private int tickets = 100;
8     //获取Lock接口的实现类对象
9     private Lock lock = new ReentrantLock();
10
11     @Override
12     public void run() {
13         while (true)
14             sale();
15     }
16
17     private void sale(){

```



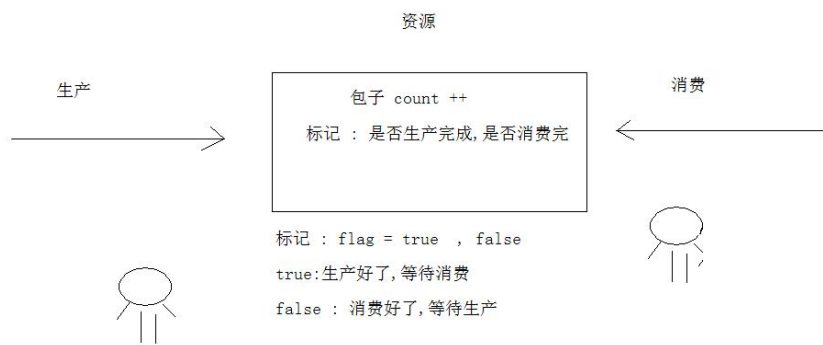
```

18         //获取锁
19         lock.lock();
20         if (tickets > 0) {
21             try {
22                 Thread.sleep(20); //线程休眠, 暂停执行
23             } catch (Exception ex) {
24             }
25             System.out.println(Thread.currentThread().getName() + " 出售第" +
tickets + "张");
26             tickets--;
27         }
28         //释放锁
29         lock.unlock();
30     }
31 }

```

## 7. 生产者与消费者

创建2个线程,一个线程表示生产者,另一个线程表示消费者



```

1  /**
2   * 定义资源对象
3   *   成员 : 产生商品的计数器
4   *           标志位
5   */
6  public class Resource {
7      int count ;
8      boolean flag ;
9  }
10

```

```

1  /**
2   * 生产者线程
3   *   资源对象中的变量++
4   */
5  public class Produce implements Runnable{
6
7      private Resource r ;
8

```

```

9      public Produce(Resource r) {
10         this.r = r;
11     }
12
13     @Override
14     public void run() {
15         while (true){
16             synchronized (r) {
17                 //判断标志位,是否允许生产
18                 //flag是true,生产完成,等待消费
19                 if (r.flag )
20                     //无限等待
21                     try{ r.wait();
22                         }catch (Exception ex){}
23                 r.count++;
24                 System.out.println("生产第" + r.count + "个");
25                 //修改标志位,已经生产了,需要消费
26                 r.flag = true;
27                 //唤醒消费者线程
28                 r.notify();
29             }
30         }
31     }
32 }

```

```

1  /**
2   * 消费者线程
3   * 资源对象中的变量输出打印
4   */
5  public class Customer implements Runnable{
6      private Resource r ;
7
8      public Customer(Resource r) {
9          this.r = r;
10     }
11
12     @Override
13     public void run() {
14         while (true){
15             synchronized (r) {
16                 //是否要消费,判断标志位 ,允许消费才能执行
17                 if (!r.flag )
18                     //消费完成,不能再次消费,等待生产
19                     try{r.wait();}catch (Exception ex){}
20                 System.out.println("消费第" + r.count);
21                 //消费完成后,修改标志位,变成已经消费
22                 r.flag = false;
23                 //唤醒生产线程
24                 r.notify();
25             }
26         }
27     }
28 }

```

```

1 public static void main(String[] args) {
2     Resource r = new Resource();
3     //接口实现类,生产的,消费的
4     Produce produce = new Produce(r);
5     Customer customer = new Customer(r);
6     //创建线程
7     new Thread(produce).start();
8     new Thread(customer).start();
9 }

```

- 线程通信的方法 wait() notify()
  - 方法的调用必须写在同步中
  - 调用者必须是作为锁的对象
  - wait(),notify()为什么要定义在Object类
    - 同步中的锁,是任意对象,任何类都继承Object
- 案例改为方法实现

```

1  /**
2   * 定义资源对象
3   * 成员 : 产生商品的计数器
4   *      标志位
5   */
6  public class Resource {
7      private int count ;
8      private boolean flag ;
9
10     //消费者调用
11     public synchronized void getCount() {
12         //flag是false,消费完成,等待生产
13         if (!flag)
14             //无限等待
15             try{this.wait();}catch (Exception ex){}
16         System.out.println("消费第"+count);
17         //修改标志位,为消费完成
18         flag = false;
19         //唤醒对方线程
20         this.notify();
21     }
22     //生产者调用
23     public synchronized void setCount() {
24         //flag是true,生产完成,等待消费
25         if (flag)
26             //无限等待
27             try{this.wait();}catch (Exception ex){}
28         count++;
29         System.out.println("生产第"+count+"个");
30         //修改标志位,为生产完成
31         flag = true;
32         //唤醒对方线程
33         this.notify();
34     }
35 }

```

```
1  /**
2   * 消费者线程
3   *    资源对象中的变量输出打印
4   */
5  public class Customer implements Runnable{
6      private Resource r ;
7
8      public Customer(Resource r) {
9          this.r = r;
10     }
11
12     @Override
13     public void run() {
14         while (true) {
15             r.getCount();
16         }
17     }
18 }
```

```
1  /**
2   * 生产者线程
3   *    资源对象中的变量++
4   */
5  public class Produce implements Runnable{
6
7      private Resource r ;
8
9      public Produce(Resource r) {
10         this.r = r;
11     }
12
13     @Override
14     public void run() {
15         while (true) {
16             r.setCount();
17         }
18     }
19 }
```