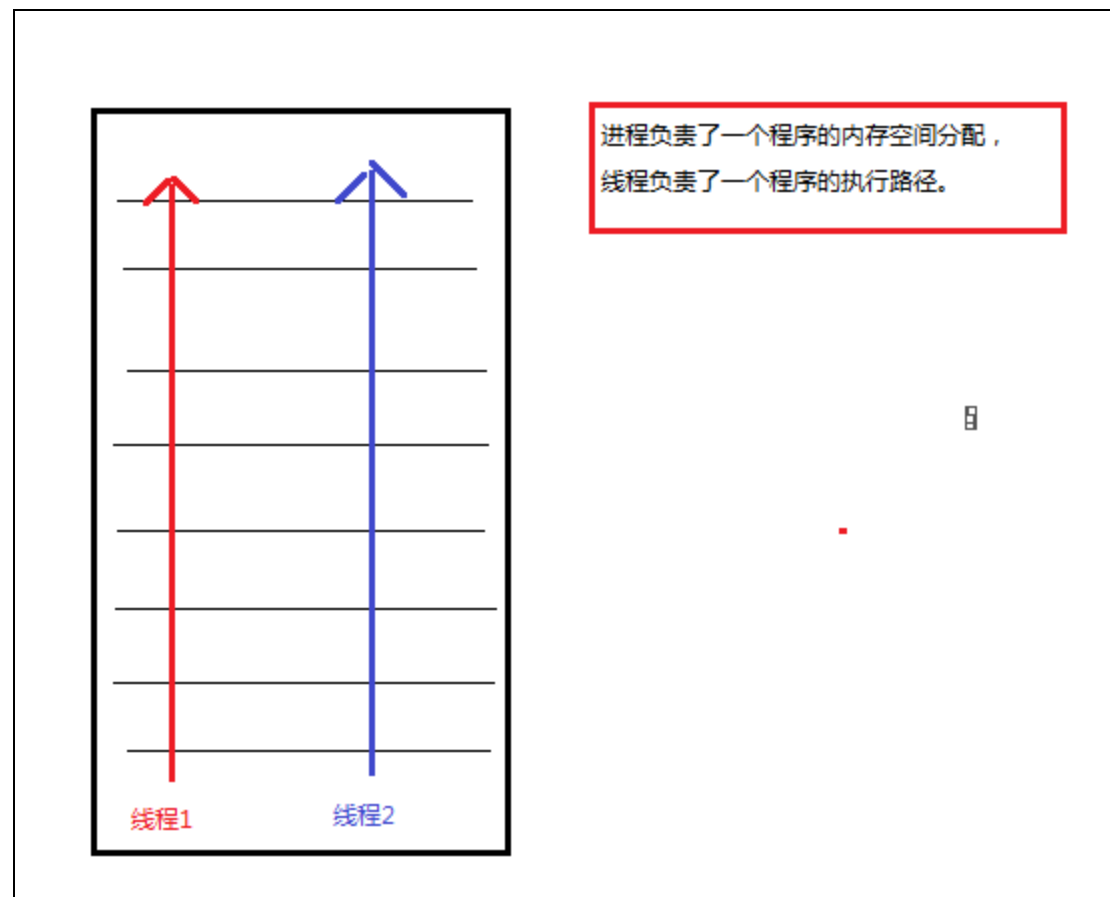


1 线程的概述

进程：正在运行的程序，负责了这个程序的内存空间分配，代表了内存中的执行区域。

线程：就是在一个进程中负责一个执行路径。

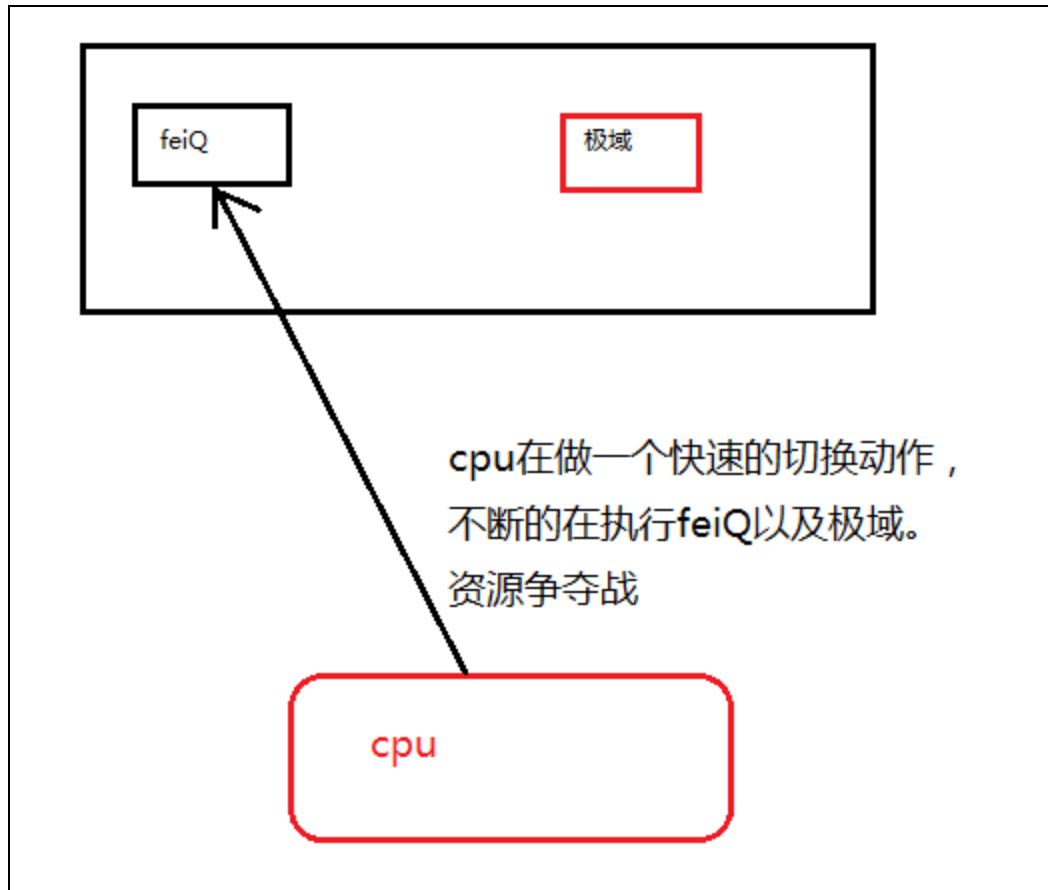
多线程：就是在一个进程中多个执行路径同时执行。



图上的一键优化与垃圾清除同时在运行，在一个进程中同时在执行了多个任务。

假象：

电脑上的程序同时在运行。“多任务”操作系统能同时运行多个进程（程序）——但实际是由于 CPU 分时机制的作用，使每个进程都能循环获得自己的 CPU 时间片。但由于轮换速度非常快，使得所有程序好象是在“同时”运行一样。



多线程的好处:

1. 解决了一个进程里面可以同时运行多个任务（执行路径）。
2. 提供资源的利用率，而不是提供效率。

多线程的弊端:

1. 降低了一个进程里面的线程的执行频率。
2. 对线程进行管理要求额外的 CPU 开销。线程的使用会给系统带来上下文切换的额外负担。
3. 公有变量的同时读或写。当多个线程需要对公有变量进行写操作时,后一个线程往往会修改掉前一个线程存放的数据，发生线程安全问题。
4. 线程的死锁。即较长时间的等待或资源竞争以及死锁等多线程症状。

2 创建线程的方式

2.1 创建线程的方式一

1. 继承 Thread 类

```

class Demo1 extends Thread{

    public Demo1(String name){
        super(name);
    }

    public void print(){
        for(int i = 0 ; i<10 ; i++){
            System.out.println(this.getName()+" : "+i);
        }
    }

    public static void main(String[] args) {
        Demo1 demo1 = new Demo1("张三"); //创建线程一
        Demo1 demo2 = new Demo1("李四"); //创建线程二
        demo1.print();
        demo2.print();
    }
}

```

getName()是获取线程的名字。

执行后的效果：

```

张三 : 0
张三 : 1
张三 : 2
张三 : 3
张三 : 4
张三 : 5
张三 : 6
张三 : 7
张三 : 8
张三 : 9
李四 : 0
李四 : 1
李四 : 2
李四 : 3
李四 : 4
李四 : 5
李四 : 6
李四 : 7
李四 : 8
李四 : 9

```

问题：先按照顺序运行完了张三，然后接着再按照顺序运行完李四，我们想要的效果是张三和李四做资源的争夺战，也就是先是张三然后李四，没有顺序的执行。这就证明多线程没有起到效果。

2. 需要复写 run 方法，把要执行的任务放在 run 方法中。

```

class Demo1 extends Thread{

    @Override
    public void run() {
        print(); //该程序的主要任务就是运行print()方法，那么在run方法中调用print方法即可
    }

    public Demo1(String name){
        super(name);
    }

    public void print(){
        for(int i = 0 ; i<10 ; i++){
            System.out.println(this.getName()+" : "+i);
        }
    }

    public static void main(String[] args) {
        Demo1 demo1 = new Demo1("张三"); //创建线程一
        Demo1 demo2 = new Demo1("李四"); //创建线程二
        demo1.run();
        demo2.run();
    }
}

```

运行效果:

```

张三 : 0
张三 : 1
张三 : 2
张三 : 3
张三 : 4
张三 : 5
张三 : 6
张三 : 7
张三 : 8
张三 : 9
李四 : 0
李四 : 1
李四 : 2
李四 : 3
李四 : 4
李四 : 5
李四 : 6
李四 : 7
李四 : 8
李四 : 9

```

问题：先按照顺序运行完了张三，然后接着再按照顺序运行完李四，我们想要的效果是张三和李四做资源的争夺战，也就是先是张三然后李四，没有顺序的执行。这就证明多线程没有起到效果。

3. 调用 start()方法启动线程

```

class Demo1 extends Thread{

    @Override
    public void run() {
        print(); //该程序的主要任务就是运行print()方法，那么在run方法中调用print方法即可
    }

    public Demo1(String name){
        super(name);
    }

    public void print(){
        for(int i = 0 ; i<10 ; i++){
            try {
                this.sleep(100); //让该运行的线程睡眠100毫秒
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(this.getName()+" : "+i);
        }
    }

    public static void main(String[] args) {
        Demo1 demo1 = new Demo1("张三"); //创建线程一
        Demo1 demo2 = new Demo1("李四"); //创建线程二
        demo1.start();
        demo2.start();
    }
}

```

效果:

```

李四 : 0
张三 : 0
李四 : 1
张三 : 1
李四 : 2
张三 : 2
李四 : 3
张三 : 3
李四 : 4
张三 : 4
李四 : 5
张三 : 5
李四 : 6
张三 : 6
李四 : 7
张三 : 7
李四 : 8
张三 : 8
李四 : 9
张三 : 9

```

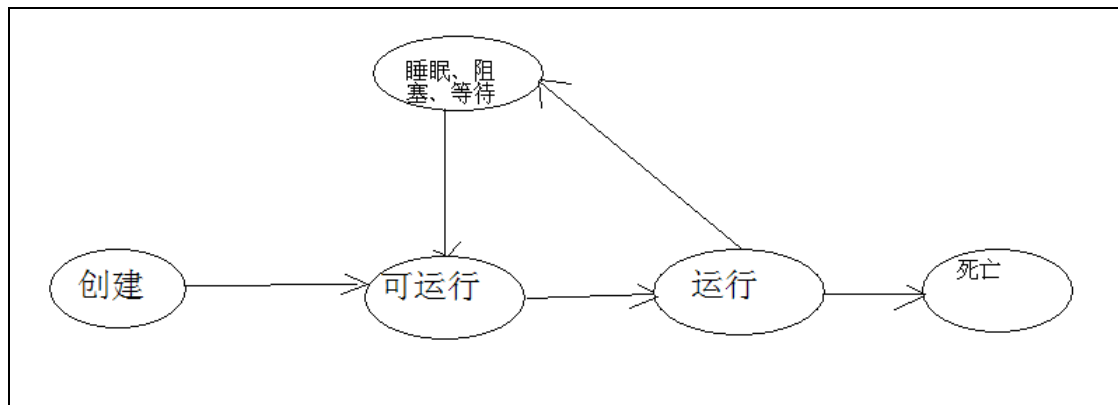
达到了我们预期的效果。

线程的使用细节:

1. 线程的启动使用父类的 start()方法
2. 如果线程对象直接调用 run(), 那么 JVN 不会当作线程来运行, 会认为是普通的方法调用。

3. 线程的启动只能由一次，否则抛出异常
4. 可以直接创建 `Thread` 类的对象并启动该线程，但是如果没有重写 `run()`，什么也不执行。
5. 匿名内部类的线程实现方式

2.2 线程的状态



创建：新创建了一个线程对象。

可运行：线程对象创建后，其他线程调用了该对象的 `start()` 方法。该状态的线程位于可运行线程池中，变得可运行，等待获取 `cpu` 的执行权。

运行：就绪状态的线程获取了 `CPU` 执行权，执行程序代码。

阻临时塞：阻塞状态是线程因为某种原因放弃 `CPU` 使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

死亡：线程执行完它的任务时。

2.3 常见线程的方法

<code>Thread(String name)</code>	初始化线程的名字
<code>getName()</code>	返回线程的名字
<code>setName(String name)</code>	设置线程对象名
<code>sleep()</code>	线程睡眠指定的毫秒数。
<code>getPriority()</code>	返回当前线程对象的优先级 默认线程的优先级是 5
<code>setPriority(int newPriority)</code>	设置线程的优先级 虽然设置了线程的优先级，但是具体的实现取决于底层的操作系统的实现（最大的优先级是 10，最小的 1，默认是 5）。
<code>currentThread()</code>	返回 <code>CPU</code> 正在执行的线程的对象

```

class ThreadDemo1 extends Thread
{
    public ThreadDemo1() {

    }

    public ThreadDemo1( String name ) {
        super( name );
    }

    public void run() {
        int i = 0;
        while(i < 30) {
            i++;
            System.out.println( this.getName() + " " + " : i = " + i );
            System.out.println( Thread.currentThread().getName() + " " + " :
i = " + i );
            System.out.println( Thread.currentThread() == this );
            System.out.println( "getId() " + " " + " : id = " + super.getId() );
            System.out.println( "getPriority() " + " " + " : Priority = " +
super.getPriority() );
        }
    }
}

class Demo3
{
    public static void main(String[] args)
    {
        ThreadDemo1 th1 = new ThreadDemo1("线程1");
        ThreadDemo1 th2 = new ThreadDemo1("线程2");
        // 设置线程名
        th1.setName( "th1" );
        th2.setName( "th2" );
        // 设置线程优先级 1 ~ 10
        th1.setPriority( 10 );
        th2.setPriority( 7 );
        // 查看SUN定义的线程优先级范围
        System.out.println("max : " + Thread.MAX_PRIORITY );
        System.out.println("min : " + Thread.MIN_PRIORITY );
        System.out.println("nor : " + Thread.NORM_PRIORITY );
        th1.start();
        th2.start();
        System.out.println("Hello World!");
    }
}

```

练习：模拟卖票

```
class SaleTickets extends Thread
{
    int tickets = 100;

    public void run() {
        while(tickets>0) {
            System.out.println("卖到了第"+tickets+"张票");
            tickets--;
        }
    }
}
class Demo2
{
    public static void main(String[] args)
    {
        SaleTickets thread1 = new SaleTickets();
        SaleTickets thread2 = new SaleTickets();
        SaleTickets thread3 = new SaleTickets();
        SaleTickets thread4 = new SaleTickets();
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
    }
}
```

存在问题：这时候启动了四个线程，那么 tickets 是一个成员变量，也就是在一个线程对象中都维护了属于自己的 tickets 属性，那么就总共存在了四份。

解决方案一： tickets 使用 static 修饰，使每个线程对象都是共享一份属性。


```
class SaleTickets extends Thread
{
    static int tickets = 100;

    public void run() {
        while(tickets>0) {
            System.out.println("卖到了第"+tickets+"张票");
            tickets--;
        }
    }
}
class Demo2
{
    public static void main(String[] args)
    {
        SaleTickets thread1 = new SaleTickets();
        SaleTickets thread2 = new SaleTickets();
        SaleTickets thread3 = new SaleTickets();
        SaleTickets thread4 = new SaleTickets();
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
    }
}
```

解决方案 2：编写一个类实现 Runnable 接口。

2.4 创建线程的方式二

创建线程的第二种方式.使用 Runnable 接口.

该类中的代码就是对线程要执行的任务的定义.

- 1: 定义了实现 Runnable 接口
 - 2: 重写 Runnable 接口中的 run 方法, 就是将线程运行的代码放入在 run 方法中
 - 3: 通过 Thread 类建立线程对象
 - 4: 将 Runnable 接口的子类对象作为实际参数, 传递给 Thread 类构造方法
 - 5: 调用 Thread 类的 start 方法开启线程, 并调用 Runnable 接口子类 run 方法
- 为什么要将 Runnable 接口的子类对象传递给 Thread 的构造函数, 因为自定义的 run 方法所属对象是 Runnable 接口的子类对象, 所以要让线程去执行指定对象的 run 方法

```
package cn.itcast.gz.runnable;

public class Demo1 {

    public static void main(String[] args) {

        MyRun my = new MyRun();
        Thread t1 = new Thread(my);
        t1.start();

        for (int i = 0; i < 200; i++) {
            System.out.println("main:" + i);
        }

    }

}

class MyRun implements Runnable {

    public void run() {

        for (int i = 0; i < 200; i++) {
            System.err.println("MyRun:" + i);
        }

    }

}
```

理解 Runnable:

Thread 类可以理解为一个工人, 而 Runnable 的实现类的对象就是这个工人的工作 (通过构造方法传递). Runnable 接口中只有一个方法 run 方法, 该方法中定义的事会被新线程执行的代码. 当我们把 Runnable 的子类对象传递给 Thread 的构造时, 实际上就是让给 Thread 取得 run 方法, 就是给了 Thread 一项任务.

买票例子使用 Runnable 接口实现

在上面的代码中故意照成线程执行完后, 执行 Thread.sleep(100), 以让 cpu 让给别的线程, 该方法会出现非运行时异常需要处理, 这里必须进行 try{}catch() {}, 因为子类不能比父类抛出更多的异常, 接口定义中没有异常, 实现类也不能抛出异常。

运行发现票号出现了负数, 显示了同一张票被卖了 4 次的情况。

出现了同样的问题。如何解决？

```

class MyTicket implements Runnable {
    int tickets = 100;
    public void run() {
        while (true) {
            if (tickets > 0) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + "
窗口@销售: "
                                + tickets + "号票");
                tickets--;
            } else {
                System.out.println("票已卖完。。。");
                break;
            }
        }
    }
}

public class Demo6 {
    public static void main(String[] args) {
        MyTicket mt = new MyTicket();
        Thread t1 = new Thread(mt);
        Thread t2 = new Thread(mt);
        Thread t3 = new Thread(mt);
        Thread t4 = new Thread(mt);
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}

```

3 锁对象

什么是锁对象？

每个 java 对象都有一个锁对象.而且只有一把钥匙.

如何创建锁对象:

可以使用 this 关键字作为锁对象,也可以使用所在类的字节码文件对应的 Class 对象作为锁对象

1. 类名.class

2. 对象.getClass()

Java 中的每个对象都有一个内置锁,只有当对象具有同步方法代码时,内置锁才会起作用,当进入一个同步的非静态方法时,就会自动获得与类的当前实例(this)相关的锁,该类的代码就是正在执行的代码。获得一个对象的锁也成为获取锁、锁定对象也可以称之为监视器来指我们正在获取的锁对象。

因为一个对象只有一个锁,所有如果一个线程获得了这个锁,其他线程就不能获得了,直到这个线程释放(或者返回)锁。也就是说在锁释放之前,任何其他线程都不能进入同步代码(不可以进入该对象的任何同步方法)。释放锁指的是持有该锁的线程退出同步方法,此时,其他线程可以进入该对象上的同步方法。

1: 只能同步方法(代码块),不能同步变量或者类

2: 每个对象只有一个锁

3: 不必同步类中的所有方法,类可以同时具有同步方法和非同步方法

4: 如果两个线程要执行一个类中的一个同步方法,并且他们使用的是了类的同一个实例(对象)来调用方法,那么一次只有一个线程能够执行该方法,另一个线程需要等待,直到第一个线程完成方法调用,总结就是:一个线程获得了对象的锁,其他线程不可以进入该对象的同步方法。

5: 如果类同时具有同步方法和非同步方法,那么多个线程仍然可以访问该类的非同步方法。同步会影响性能(甚至死锁),优先考虑同步代码块。

6: 如果线程进入 sleep() 睡眠状态,该线程会继续持有锁,不会释放。

4 死锁



经典的“哲学家就餐问题”,5个哲学家吃中餐,坐在圆桌子旁。每人有5根筷子(不是5双),每两个人中间放一根,哲学家时而思考,时而进餐。每个人都需要一双筷子才能吃到东西,吃完后将筷子放回原处继续思考,如果每个人都立刻抓住自己左边的筷子,然后等待右边的筷子空出来,同时又不放下已经拿到的筷子,这样每个人都无法得到1双筷子,无法吃饭都会饿死,这种情况就会产生死锁:每个人都拥有其他人需要的资源,同时又等待其他人拥有的资源,并且每个人在获得所有需要的资源之前都不会放弃已经拥有的资源。

当多个线程完成功能需要同时获取多个共享资源的时候可能会导致死锁。

1: 两个任务以相反的顺序申请两个锁，死锁就可能出现

2: 线程 T1 获得锁 L1，线程 T2 获得锁 L2，然后 T1 申请获得锁 L2，同时 T2 申请获得锁 L1，此时两个线程将要永久阻塞，死锁出现

如果一个类可能发生死锁，那么并不意味着每次都会发生死锁，只是表示有可能。要避免程序中出现死锁。

例如，某个程序需要访问两个文件，当进程中的两个线程分别各锁住了一个文件，那它们都在等待对方解锁另一个文件，而这永远不会发生。

3: 要避免死锁

```

public class DeadLock {
    public static void main(String[] args) {
        new Thread(new Runnable() { // 创建线程，代表中国人
            public void run() {
                synchronized ("刀叉") { // 中国人拿到了刀叉

System.out.println(Thread.currentThread().getName()
                    + ": 你不给我筷子，我就不给你刀叉");

                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                synchronized ("筷子") {
                    System.out.println(Thread.currentThread()
                        .getName() + ": 给你刀叉");
                }
            }
        }, "中国人").start();

        new Thread(new Runnable() { // 美国人
            public void run() {
                synchronized ("筷子") { // 美国人拿到了筷子

System.out.println(Thread.currentThread().getName()
                    + ": 你先给我刀叉，我再给你筷子");

                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                synchronized ("刀叉") {
                    System.out.println(Thread.currentThread()
                        .getName() + ": 好吧，把筷子给你.");
                }
            }
        }, "美国人").start();
    }
}

```

5 线程的通讯

线程间通信其实就是多个线程在操作同一个资源，但操作动作不同

生产者消费者

如果有多个生产者和消费者，一定要使用 `while` 循环判断标记，然后在使用 `notifyAll` 唤醒，否则容易只用 `notify` 容易出现只唤醒本方线程情况，导致程序中的所有线程都在等待。例如：有一个数据存储空间，划分为两个部分，一部分存储人的姓名，一部分存储性别，我们开启一个线程，不停地想其中存储姓名和性别（生产者），开启另一个线程从数据存储空间中取出数据（消费者）。

由于是多线程的，就需要考虑，假如生产者刚向数据存储空间中添加了一个人名，还没有来得及添加性别，`cpu` 就切换到了消费者的线程，消费者就会将这个人的姓名和上一个人的性别进行了输出。

还有一种情况是生产者生产了若干次数据，消费者才开始取数据，或者消费者取出数据后，没有等到消费者放入新的数据，消费者又重复的取出自己已经去过的数据。

```
public class Demo10 {  
    public static void main(String[] args) {  
        Person p = new Person();  
        Producer pro = new Producer(p);  
        Consumer con = new Consumer(p);  
        Thread t1 = new Thread(pro, "生产者");  
        Thread t2 = new Thread(con, "消费者");  
        t1.start();  
        t2.start();  
    }  
}  
  
// 使用Person作为数据存储空间  
class Person {  
    String name;  
    String gender;  
}  
  
// 生产者  
class Producer implements Runnable {  
    Person p;  
  
    public Producer() {  
    }  
  
    public Producer(Person p) {  
        this.p = p;  
    }  
}
```

```

@Override
public void run() {
    int i = 0;
    while (true) {
        if (i % 2 == 0) {
            p.name = "jack";
            p.gender = "man";
        } else {
            p.name = "小丽";
            p.gender = "女";
        }
        i++;
    }
}

// 消费者
class Consumer implements Runnable {
    Person p;

    public Consumer() {

    }

    public Consumer(Person p) {
        this.p = p;
    }

    @Override
    public void run() {

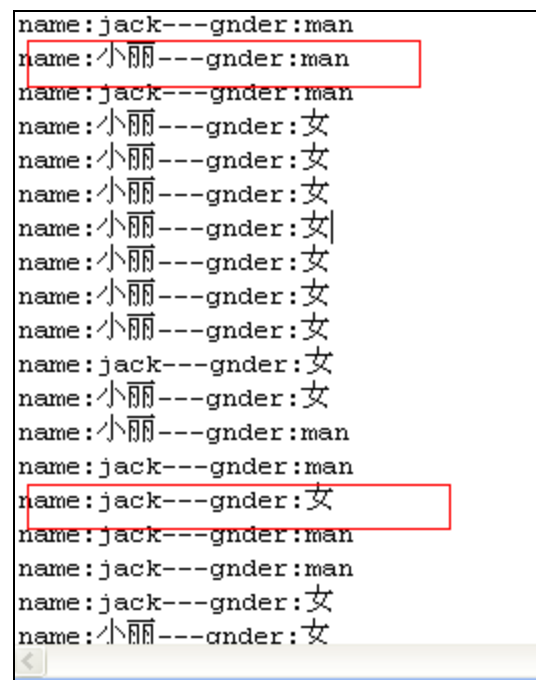
        while (true) {
            System.out.println("name:" + p.name + "---gender:" + p.gender);
        }
    }
}

```

在上述代码中，Producer 和 Consumer 类的内部都维护了一个 Person 类型的 p 成员变量，通过构造函数进行赋值，在 man 方法中创建了一个 Person 对象，将其同时传递给 Producer 和 Consumer 对象，所以 Producer 和 Consumer 访问的是同一个 Person

对象。并启动了两个线程。

输出：



```
name:jack---gnder:man
name:小丽---gnder:man
name:jack---gnder:man
name:小丽---gnder:女
name:小丽---gnder:女
name:小丽---gnder:女
name:小丽---gnder:女|
name:小丽---gnder:女
name:小丽---gnder:女
name:小丽---gnder:女
name:jack---gnder:女
name:小丽---gnder:女
name:小丽---gnder:man
name:jack---gnder:man
name:jack---gnder:女
name:jack---gnder:man
name:jack---gnder:man
name:jack---gnder:女
name:小丽---cnder:女
```

显然屏幕输出了小丽 man 这样的结果是出现了线程安全问题。所以需要使用 synchronized 来解决该问题。

```
package cn.itcast.gz.runnable;

public class Demo10 {
    public static void main(String[] args) {
        Person p = new Person();
        Producer pro = new Producer(p);
        Consumer con = new Consumer(p);
        Thread t1 = new Thread(pro, "生产者");
        Thread t2 = new Thread(con, "消费者");
        t1.start();
        t2.start();
    }
}

// 使用Person作为数据存储空间
class Person {
    String name;
    String gender;
}

// 生产者
```

```
class Producer implements Runnable {
    Person p;

    public Producer() {

    }

    public Producer(Person p) {
        this.p = p;
    }

    @Override
    public void run() {
        int i = 0;
        while (true) {
            synchronized (p) {
                if (i % 2 == 0) {
                    p.name = "jack";
                    p.gender = "man";
                } else {
                    p.name = "小丽";
                    p.gender = "女";
                }
                i++;
            }

        }

    }
}

// 消费者
class Consumer implements Runnable {
    Person p;

    public Consumer() {

    }

    public Consumer(Person p) {
        this.p = p;
    }
}
```

```

@Override
public void run() {

    while (true) {
        synchronized (p) {
            System.out.println("name:" + p.name + "---gender:" +
p.gender);
        }

    }
}
}

```

编译运行：屏幕没有再输出 jack -女 或者小丽- man 这种情况了。说明我们解决了线程同步问题，但是仔细观察，生产者生产了若干次数据，消费者才开始取数据，或者消费者取出数据后，没有等到生产者放入新的数据，消费者又重复的取出自己已经去过的数据。这个问题依然存在。

升级：在 Person 类中添加两个方法，set 和 read 方法并设置为 synchronized 的，让生产者和消费者调用这两个方法。

```

public class Demo10 {
    public static void main(String[] args) {
        Person p = new Person();
        Producer pro = new Producer(p);
        Consumer con = new Consumer(p);
        Thread t1 = new Thread(pro, "生产者");
        Thread t2 = new Thread(con, "消费者");
        t1.start();
        t2.start();
    }
}

// 使用Person作为数据存储空间
class Person {
    String name;
    String gender;

    public synchronized void set(String name, String gender) {
        this.name = name;
        this.gender = gender;
    }

    public synchronized void read() {

```

```
        System.out.println("name:" + this.name + "----gender:" +
this.gender);
    }

}

// 生产者
class Producer implements Runnable {
    Person p;

    public Producer() {

    }

    public Producer(Person p) {
        this.p = p;
    }

    @Override
    public void run() {
        int i = 0;
        while (true) {

            if (i % 2 == 0) {
                p.set("jack", "man");
            } else {
                p.set("小丽", "女");
            }
            i++;

        }

    }

}

// 消费者
class Consumer implements Runnable {
    Person p;

    public Consumer() {

    }

}
```

```

public Consumer(Person p) {
    this.p = p;
}

@Override
public void run() {

    while (true) {
        p.read();

    }
}
}

```

需求：我们需要生产者生产一次，消费者就消费一次。然后这样有序的循环。这就需要使用线程间的通信了。Java 通过 Object 类的 wait, notify, notifyAll 这几个方法实现线程间的通信。

1.1.1. 等待唤醒机制

wait: 告诉当前线程放弃执行权，并放弃监视器（锁）并进入阻塞状态，直到其他线程持有获得执行权，并持有了相同的监视器（锁）并调用 notify 为止。

notify: 唤醒持有同一个监视器（锁）中调用 wait 的第一个线程，例如，餐馆有空位置后，等候就餐最久的顾客最先入座。注意：被唤醒的线程是进入了可运行状态。等待 cpu 执行权。

notifyAll: 唤醒持有同一监视器中调用 wait 的所有的线程。

如何解决生产者和消费者的问题？

可以通过设置一个标记，表示数据的（存储空间的状态）例如，当消费者读取了（消费了一次）一次数据之后可以将标记改为 false，当生产者生产了一个数据，将标记改为 true。也就是只有标记为 true 的时候，消费者才能取走数据，标记为 false 时候生产者才生产数据。

代码实现：

```

package cn.itcast.gz.runnable;

public class Demo10 {
    public static void main(String[] args) {
        Person p = new Person();
        Producer pro = new Producer(p);
        Consumer con = new Consumer(p);
        Thread t1 = new Thread(pro, "生产者");
    }
}

```

```

        Thread t2 = new Thread(con, "消费者");
        t1.start();
        t2.start();
    }
}

// 使用Person作为数据存储空间
class Person {
    String name;
    String gender;
    boolean flag = false;

    public synchronized void set(String name, String gender) {
        if (flag) {
            try {
                wait();
            } catch (InterruptedException e) {

                e.printStackTrace();
            }
        }
        this.name = name;
        this.gender = gender;
        flag = true;
        notify();
    }

    public synchronized void read() {
        if (!flag) {
            try {
                wait();
            } catch (InterruptedException e) {

                e.printStackTrace();
            }
        }
        System.out.println("name:" + this.name + "----gender:" +
this.gender);
        flag = false;
        notify();
    }
}

```

```
// 生产者
class Producer implements Runnable {
    Person p;

    public Producer() {

    }

    public Producer(Person p) {
        this.p = p;
    }

    @Override
    public void run() {
        int i = 0;
        while (true) {

            if (i % 2 == 0) {
                p.set("jack", "man");
            } else {
                p.set("小丽", "女");
            }
            i++;

        }

    }

}
```

```
// 消费者
class Consumer implements Runnable {
    Person p;

    public Consumer() {

    }

    public Consumer(Person p) {
        this.p = p;
    }

    @Override
    public void run() {
```

```
        while (true) {  
            p.read();  
        }  
    }  
}
```

线程间通信其实就是多个线程在操作同一个资源，但操作动作不同，wait，notify（），notifyAll（）都使用在同步中，因为要对持有监视器（锁）的线程操作，所以要使用在同步中，因为只有同步才具有锁。

为什么这些方法定义在 Object 类中

因为这些方法在操作线程时，都必须标识他们所操作线程持有的锁，只有同一个锁上的被等待线程，可以被统一锁上 notify 唤醒，不可以对不同锁中的线程进行唤醒，就是等待和唤醒必须是同一个锁。而锁由于可以使任意对象，所以可以被任意对象调用的方法定义在 Object 类中

wait() 和 sleep() 有什么区别？

wait(): 释放资源，释放锁。是 Object 的方法

sleep(): 释放资源，不释放锁。是 Thread 的方法

定义了 notify 为什么还要定义 notifyAll，因为只用 notify 容易出现只唤醒本方线程情况，导致程序中的所有线程都在等待。

2. 线程生命周期

任何事物都是生命周期，线程也是，

1. 正常终止 当线程的 run() 执行完毕，线程死亡。
2. 使用标记停止线程

注意：Stop 方法已过时，就不能再使用这个方法。

如何使用标记停止线程停止线程。

开启多线程运行，运行代码通常是循环结构，只要控制住循环，就可以让 run 方法结束，线程就结束。

```
class StopThread implements Runnable {  
    public boolean tag = true;  
    @Override  
    public void run() {  
        int i = 0;  
  
        while (tag) {  
            i++;  
        }  
    }  
}
```



```

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

public class Demo8 {
    public static void main(String[] args) {
        StopThread st = new StopThread();
        Thread th = new Thread(st, "线程1");
        th.start();

        for (int i = 0; i < 100; i++) {
            if (i == 50) {
                System.out.println("main i:" + i);
                st.tag = false;
            }
        }
    }
}

```

上述案例中定义了一个计数器 i，用来控制 main 方法（主线程）的循环打印次数，在 i 到 50 这段时间内，两个线程交替执行，当计数器变为 50，程序将标记改为 false，也就是终止了线程 1 的 while 循环，run 方法结束，线程 1 也随之结束。注意：当计数器 i 变为 50 的，将标记改为 false 的时候，cpu 不一定马上回到线程 1，所以线程 1 并不会马上终止。

3. 后台线程

后台线程：就是隐藏起来一直在默默运行的线程，直到进程结束。

实现：

```
setDaemon(boolean on)
```

特点：

当所有的非后台线程结束时，程序也就终止了同时还会杀死进程中的所有后台线程，也就是说，只要有非后台线程还在运行，程序就不会终止，执行 main 方法的主线程就是一个非后台线程。

必须在启动线程之前（调用 start 方法之前）调用 setDaemon（true）方法，才可以把该线程设置为后台线程。

一旦 main（）执行完毕，那么程序就会终止，JVM 也就退出了。

可以使用 isDaemon（）测试该线程是否为后台线程（守护线程）。

该案例：开启了一个 qq 检测升级的后台线程，通过 while 真循环进行不停检测，当计数器变为 100 的时候，表示检测完毕，提示是否更新，线程同时结束。

为了验证，当非后台线程结束时，后台线程是否终止，故意让该后台线程睡眠一会。发现只要 main 线程执行完毕，后台线程也就随之消亡了。

```

class QQUpdate implements Runnable {
    int i = 0;

```

```

@Override
public void run() {
    while (true) {

        System.out.println(Thread.currentThread().getName() + " 检测是否有可用更新");
        i++;
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {

            e.printStackTrace();
        }
        if (i == 100) {
            System.out.println("有可用更新，是否升级？");
            break;
        }
    }
}

public class Demo9 {
    public static void main(String[] args) {
        QQUpdate qq = new QQUpdate();
        Thread th = new Thread(qq, "qqupdate");
        th.setDaemon(true);
        th.start();

        System.out.println(th.isDaemon());
        System.out.println("hello world");
    }
}

```

Thread 的 join 方法

当 A 线程执行到了 B 线程 Join 方法时 A 就会等待，等 B 线程都执行完 A 才会执行，Join 可以用来临时加入线程执行

本案例，启动了一个 JoinThread 线程，main（主线程）进行 for 循环，当计数器为 50 时，让 JoinThread，通过 join 方法，加入到主线程中，发现只有 JoinThread 线程执行完，主线程才会执行完毕。

可以刻意让 JoinThread 线程 sleep，如果 JoinThread 没有调用 join 方法，那么肯定是主线程执行完毕，但是由于 JoinThread 线程加入到了 main 线程，必须等 JoinThread 执行完毕主线程才能继续执行。

```

class JoinThread implements Runnable {

    @Override

```

```

    public void run() {
        int i = 0;
        while (i < 300) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "
i:" + i);
            i++;
        }
    }
}

public class Demo10 {
    public static void main(String[] args) throws InterruptedException
    {
        JoinThread jt = new JoinThread();
        Thread th = new Thread(jt, "one");
        th.start();
        int i = 0;
        while (i < 200) {
            if (i == 100) {
                th.join();
            }
            System.err.println(Thread.currentThread().getName() + "
i:" + i);
            i++;
        }
    }
}

```

上述程序用到了 Thread 类中的 join 方法，即 th.join 语句，作用是将 th 对应的线程合并到调用 th.join 语句的线程中，main 方法的线程中计数器到达 100 之前，main 线程和 one 线程是交替执行的。在 main 线程中的计数器到达 100 后，只有 one 线程执行，也就是 one 线程此时被加进了 main 线程中，one 线程不执行完，main 线程会一直等待带参数的 join 方法是指定合并时间，有纳秒和毫秒级别。