多线程技术

主要内容

多线程介绍

线程的创建

线程的使用

线程的优先级

守护线程

线程同步

线程并发协作

学习目标

知识点	要求
多线程介绍	了解
线程的创建	掌握
线程的使用	掌握
线程的优先级	掌握
守护线程	掌握
线程同步	掌握
线程并发协作	掌握

一、 多线程介绍

1 多线程的基本概念

1.1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，比

如：我们要启动酷狗听音乐，则需要执行酷狗对应的的可执行程序。当我们双击酷狗的可执行程序后操作系统会将该程序加载到内存中，开始执行该程序，于是产生了“进程”。

1.2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 。如正在运行的写字板程序就是一个进程。

- 进程是程序的一次动态执行过程， 占用特定的地址空间。
- 每个进程由 3 部分组成：cpu、data、code。每个进程都是独立的，保有自己的 cpu 时间，代码和数据，即便用同一份程序产生好几个进程，它们之间还是拥有自己的这 3 样东西，这样的缺点是：浪费内存，cpu 的负担较重。
- 多任务(Multitasking)操作系统将 CPU 时间动态地划分给每个进程，操作系统同时执行多个进程，每个进程独立运行。以进程的观点来看，它会以为自己独占 CPU 的使用权。
- 进程的查看
  - Windows 系统: Ctrl+Alt+Del，启动任务管理器即可查看所有进程。

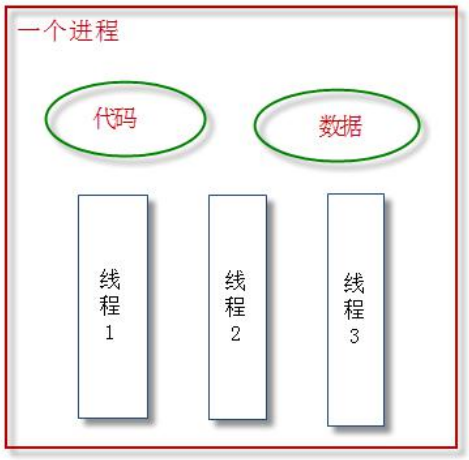
1.3线程

一个进程可以产生多个线程。同多个进程可以共享操作系统的某些资源一样，同一进程的多个线程也可以共享此进程的某些资源（比如：代码、数据），所以线程又被称为轻量级进程(lightweight process)。

- ❑ 一个进程内部的一个执行单元，它是程序中的一个单一的顺序控制流程。
- ❑ 一个进程可拥有多个并行的(concurrent)线程。
- ❑ 一个进程中的多个线程共享相同的内存单元/内存地址空间，可以访问相同的变量和对象，而且它们从同一堆中分配对象并进行通信、数据交换和同步操

作。

- 由于线程间的通信是在同一地址空间上进行的，所以不需要额外的通信机制，这就使得通信更简便而且信息传递的速度也更快。
- 线程的启动、中断、消亡，消耗的资源非常少。



2 线程和进程的区别

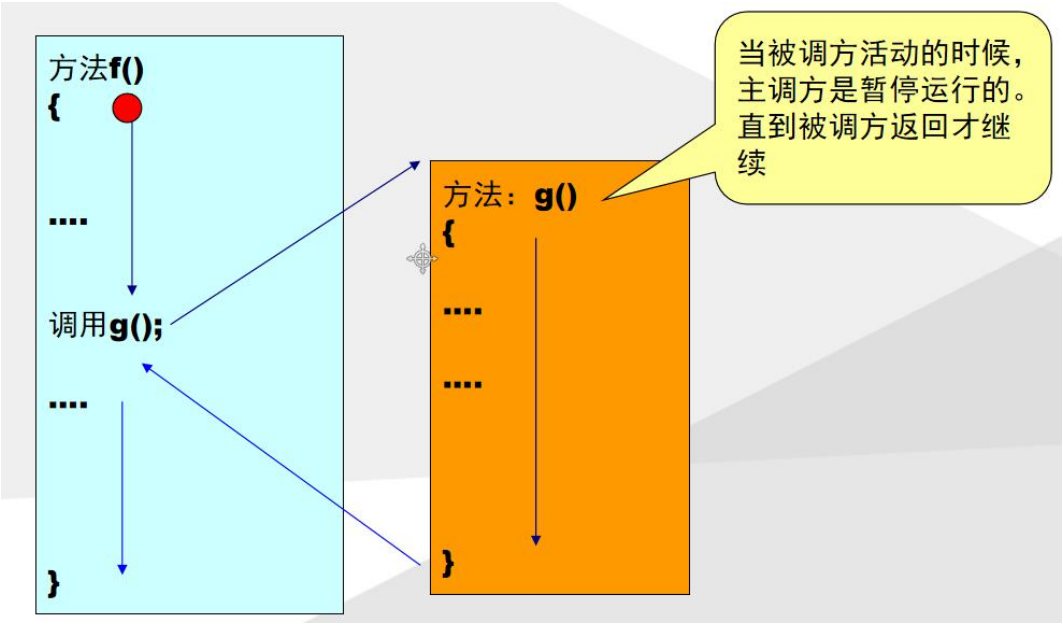
- 线程在进程中运行的。
- 一个进程可以包含多个线程。
- 不同进程间数据很难共享，而同一进程下不同线程间数据很易共享。
- 进程要比线程消耗更多的计算机资源。
- 进程间不会相互影响，因为它们的空间是完全隔离的。而进程中的一个线程挂掉将导致整个进程挂掉。
- 进程使用的内存地址可以上锁，即一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。
- 一个进程如果只有一个线程则可以被看作单线程的，如果一个进程内拥有多个线程，进程的执行过程不是一条线（线程）的，而是多条线（线程）共同完成的。

3 什么是并发

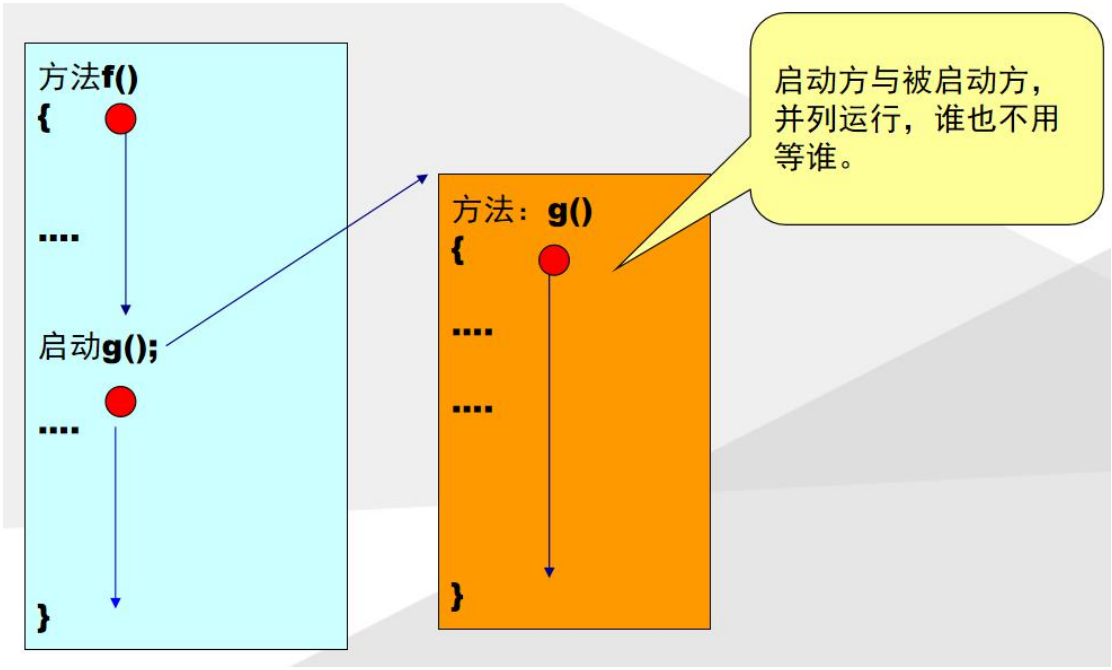
并发是指在一段时间内同时做多个事情。当有多个线程在运行时,如果只有一个 CPU,这种情况下计算机操作系统会采用并发技术实现并发运行，具体做法是采用“ 时间片轮询算法”，在一个时间段的线程代码运行时，其它线程处于就绪状。这种方式我们称之为并发(Concurrent)。

4 线程的执行特点

4.1方法的执行特点



4.2线程的执行特点



5 什么是主线程以及子线程

5.1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。

5.2子线程

在主线程中创建并启动的线程，一般称之为子线程。

二、 线程的创建

在 Java 中使用多线程非常简单，我们先学习如何创建线程，然后再结合案例深入剖析线程的特性。

1 通过继承 Thread 类实现多线程

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

继承 Thread 类实现多线程的步骤：

- 继承 Thread 类定义线程类。
- 重写 Thread 类中的 run( )方法。run( )方法也称为线程体。
- 实例化线程类并通过 start()方法启动线程。

```
public class TestThread extends Thread {
    public TestThread(){
        System.out.println(this.getName());
    }

    /**
     * 线程的线程体
     */
    @Override
    public void run() {

        System.out.println(this.getName()+"线程开始");

        for(int i=0;i<20;i++){
            System.out.println(this.getName()+" "+i);
```

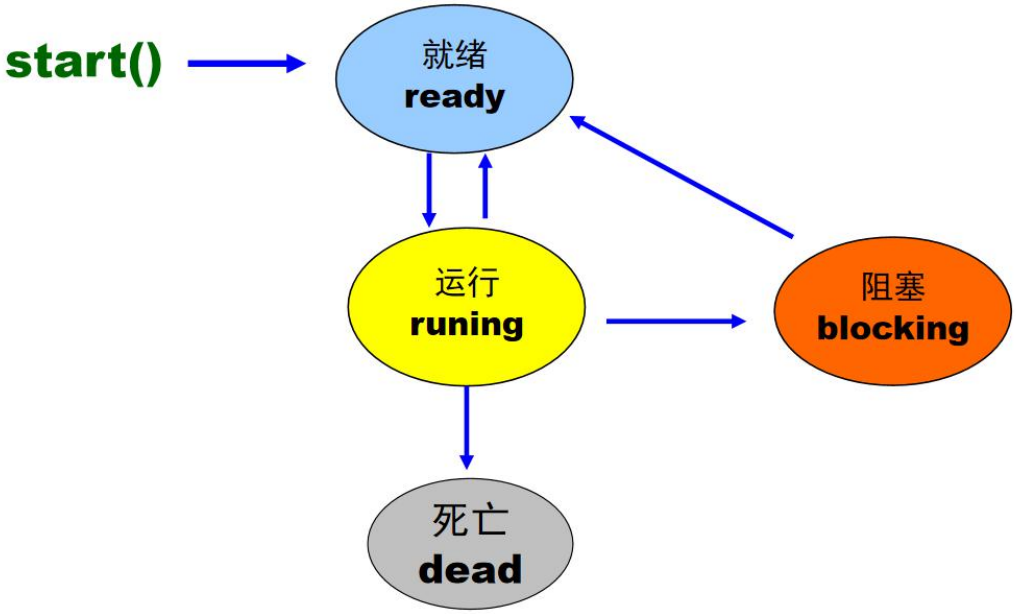
```
        }

        System.out.println(this.getName()+"线程结束");

    }

    public static void main(String[] args) {

        System.out.println("主线程开始");

        TestThread t1 = new TestThread();

        //启动线程

        t1.start();
        TestThread t2 = new TestThread();

        //启动线程

        t2.start();

        System.out.println("主线程结束");

    }
}
```

2 通过实现 Runnable 接口实现多线程

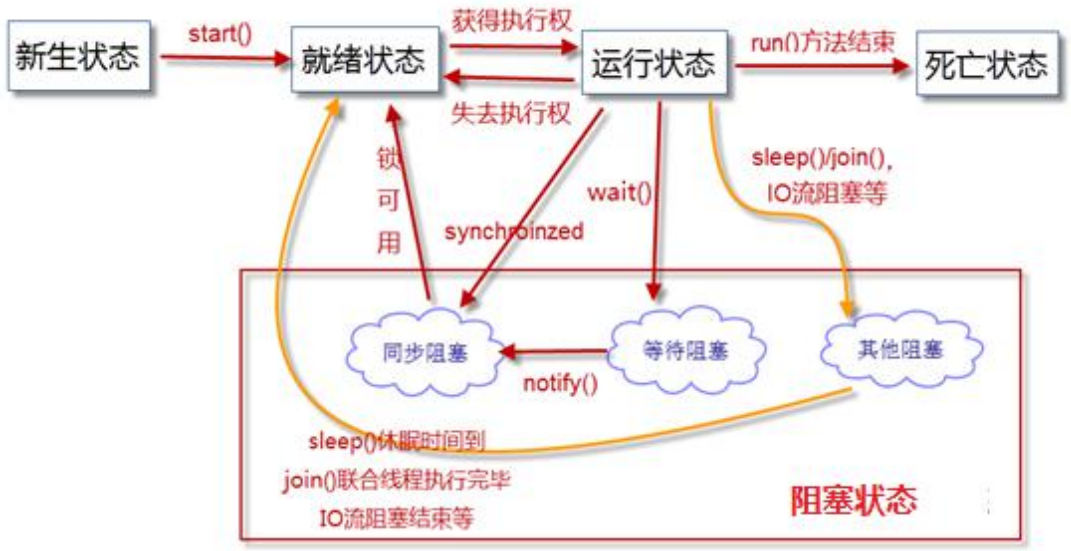
```
public class TestThread2 implements Runnable {
    public TestThread2(){
        System.out.println(Thread.currentThread().getName());
    }
    /**

     * 当前线程的线程体方法

     */
    @Override
    public void run() {

        System.out.println(Thread.currentThread().getName()+" 线程开

始");

        for(int i=0;i<20;i++){
            System.out.println(Thread.currentThread().getName()+"
"+i);
        }
```

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

    public static void main(String[] args) {
        System.out.println("主线程开始");
        TestThread2 testThread2 = new TestThread2();
        Thread t1 = new Thread(testThread2);
        t1.start();

        Thread t2 = new Thread(new TestThread2());
        t2.start();
        System.out.println("主线程结束");
    }
}
```

3 线程的执行流程



4 线程的生命周期



一个线程对象在它的生命周期内，需要经历 5 个状态。

- **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。

- **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 中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。

- **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。

- **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。

- **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。

三、 线程的使用

1 终止线程

如果我们想在一个线程中终止另一个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，当这个变量值为 false 时，则终止线程的运行。

```
public class StopThread implements Runnable {
    private boolean flag = true;
    @Override
    public void run() {

        System.out.println(Thread.currentThread().getName()+" 线程开
始");

            int i= 0;
            while(flag){

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

        System.out.println(Thread.currentThread().getName()+" 线程结
束");
```

```
    }
    public void stop(){
        this.flag = false;
    }

    public static void main(String[] args)throws Exception {

        System.out.println("主线程开始");

        StopThread st = new StopThread();
        Thread t1 = new Thread(st);
        t1.start();
        System.in.read();
        st.stop();

        System.out.println("主线程结束");

    }
}
```

2 暂停当前线程执行 sleep/yield

暂停线程执行常用的方法有 sleep()和 yield()方法，这两个方法的区别是：

- 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。
- yield()方法：可以让正在运行的线程直接进入就绪状态，让出 CPU 的使用权。

2.1 sleep 方法的使用

```
public class SleepThread implements Runnable {
    @Override
    public void run() {

        System.out.println(Thread.currentThread().getName()+" 线程开始");

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

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

```
                try {
                    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");
    }
}
```

2.2 yield 方法的使用

yield()方法的作用：暂停当前正在执行的线程，并执行其他线程。

yield()让当前正在运行的线程回到可运行状态，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 YieldThread 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 YieldThread());
        Thread t2 = new Thread(new YieldThread());
        t1.start();
        t2.start();
    }
}
```

3 线程的联合

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。

3.1join 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{

```

```
    @Override
    public void run() {
        for(int i=0;i<10;i++){
            System.out.println(Thread.currentThread().getName()+"
"+i);
            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()+"
"+i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class JoinThread {
    public static void main(String[] args) {
        Thread t = new Thread(new A());
        Thread t1 = new Thread(new B());
        t.start();
        t1.start();
        for(int i=0;i<10;i++){
            System.out.println(Thread.currentThread().getName()+"
"+i);
            if(i ==2 ){
                try {
                    t.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
```

```
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

3.2线程联合案例

```
/**
 * 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);
        }

        System.out.println("爸爸高兴的接过烟，并把零钱给了儿子");

    }
}

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

        System.out.println("爸爸和儿子买烟的故事");

        Thread t = new Thread(new FatherThread());
        t.start();
    }
}
```

4 Thread 类中的其他常用方法

4.1获取当前线程名称

4.1.1 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

```
class GetName1 extends Thread{
    @Override
    public void run() {
        System.out.println(this.getName());
    }
}
```

4.1.2 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());
    }
}
```

4.2设置线程的名称

4.2.1 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) {
        SetName1 setName1 = new SetName1("SetName1");
        setName1.start();
```

```
    }
}
```

4.2.2 方式二

通过 setName()方法设置线程名称。

```
class SetName2 implements Runnable{

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}
public class SetNameThread {
    public static void main(String[] args) {
        /*SetName1 setName1 = new SetName1();
        setName1.setName("SetName1");
        setName1.start();*/
        Thread thread = new Thread(new SetName2());
        thread.setName("SetName2");
        thread.start();
    }
}
```

4.3判断当前线程是否存活

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() {
        System.out.println(Thread.currentThread().isAlive()+" 2");
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }
}
public class AliveThread {
    public static void main(String[] args) {
        Thread thread = new Thread(new Alive());
        System.out.println(thread.isAlive()+" 1");
        thread.start();
        System.out.println(thread.isAlive()+" 3");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(thread.isAlive()+" 4");
    }
}
```

四、 线程的优先级

1 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。

2 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);

```
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(){
        this.flag = false;
    }
}

public class PriorityThread {
    public static void main(String[] args)throws Exception {
        Priority p1 = new Priority();
        Priority p2 = new Priority();

        Thread t1 = new Thread(p1,"线程 1");

        Thread t2 = new Thread(p2,"线程 2");

        System.out.println(t1.getPriority());
        //Thread.MAX_PRIORITY = 10
        t1.setPriority(Thread.MAX_PRIORITY);
        //Thread.MAX_PRIORITY = 1
        t2.setPriority(Thread.MIN_PRIORITY);
        t1.start();
        t2.start();
        Thread.sleep(1000);
        p1.stop();
        p2.stop();
    }
}
```

五、 守护线程

1 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。

2 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。

3 守护线程的使用

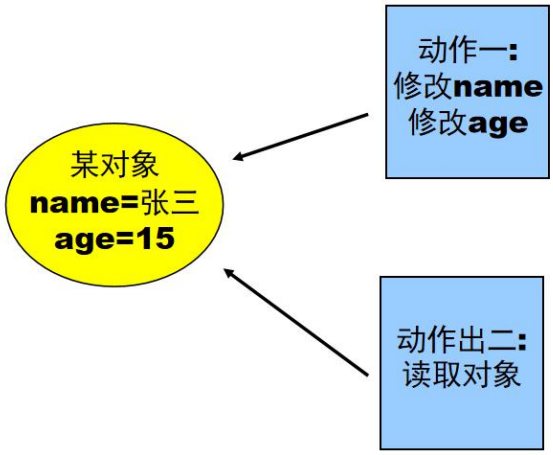
```
/**
 * 守护线程类
 */
class Daemon implements  Runnable{

    @Override
    public void run() {
        for(int i=0;i<20;i++){
            System.out.println(Thread.currentThread().getName()+"
```

```
"+i);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class UsersThread implements Runnable{

    @Override
    public void run() {
        Thread t = new Thread(new Daemon(),"Daemon");

        //将该线程设置为守护线程

        t.setDaemon(true);
        t.start();
        for(int i=0;i<5;i++){
            System.out.println(Thread.currentThread().getName()+"
"+i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class DaemonThread {
    public static void main(String[] args)throws Exception {
        Thread t = new Thread(new UsersThread(),"UsersThread");
        t.start();
        Thread.sleep(1000);

        System.out.println("主线程结束");

    }
}
```

六、 线程同步

1 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。

2 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取款金额

（4）如果余额大于取款金额，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() {
        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 extends Thread{
    //账户对象
    private Account account;
    //取款金额
    private double drawMoney;
    public DrawThread(String name,Account account,double drawMoney){
        super(name);
        this.account = account;
        this.drawMoney = drawMoney;
    }

    /**
     * 取款线程
     */
    @Override
    public void run() {
        //判断当前账户余额是否大于或等于取款金额
        if(this.account.getBalance() >= this.drawMoney){
            System.out.println(this.getName()+" 取钱成功！吐出钞票：
"+this.drawMoney);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
            //更新账户余额
            this.account.setBalance(this.account.getBalance()-
this.drawMoney);
            System.out.println("\t 余额为：
"+this.account.getBalance());
        }else{
            System.out.println(this.getName()+" 取钱失败，余额不足");
        }
    }
}
public class DrawMoneyThread {
    public static void main(String[] args) {
        Account account = new Account("1234",1000);
        new DrawThread("老公",account,800).start();
        new DrawThread("老婆",account,800).start();
    }
}
```

3 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：放在范围操作符(public)之后,返回类型声明(void)之前。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。

4 修改线程冲突案例演示

```
/**
```

```
 * 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() {
        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 extends Thread{
    //账户对象
```

```
    private Account account;

    //取款金额

    private double drawMoney;
    public DrawThread(String name,Account account,double drawMoney){
        super(name);
        this.account = account;
        this.drawMoney = drawMoney;
    }

    /**

     * 取款线程

     */
    @Override
    public void run() {
        synchronized (this.account) {

            //判断当前账户余额是否大于或等于取款金额

            if (this.account.getBalance() >= this.drawMoney) {

                System.out.println(this.getName() + " 取钱成功！吐出钞票：
" + this.drawMoney);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                //更新账户余额

                this.account.setBalance(this.account.getBalance() -
this.drawMoney);

                System.out.println("\t 余额为：" +
this.account.getBalance());
            } else {

                System.out.println(this.getName() + " 取钱失败，余额不足
");
            }
        }
    }
}
```

```
public class DrawMoneyThread {
    public static void main(String[] args) {
        Account account = new Account("1234",1000);

        new DrawThread("老公",account,800).start();

        new DrawThread("老婆",account,800).start();

    }
}
```

5 线程同步的使用

5.1使用 this 作为线程对象锁

在不同线程中，相同对象中的 synchronized 会互斥。

语法结构：

synchronized(this){

//同步代码

}

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);
            System.out.println(this.name + " 系统启动成功");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    /**
    * 编码
    */
    synchronized public void coding(){
        try {
            System.out.println(this.name + " 双击 Idea");
            Thread.sleep(500);
            System.out.println(this.name + " Idea 启动完毕");
            Thread.sleep(500);
            System.out.println(this.name + " 开开心心的写代码");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

/**
 * 打开电脑的工作线程
```

```
 */
class Working1 extends Thread{
    private  Programmer p;
    public Working1(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.computer();
    }
}

/**
 * 编写代码的工作线程
 */
class Working2 extends Thread{
    private  Programmer p;
    public Working2(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.coding();
    }
}
public class TestSyncThread {
    public static void main(String[] args) {

        Programmer p = new Programmer("张三");
        new Working1(p).start();
        new Working2(p).start();
    }
}
```

5.2使用字符串作为线程对象锁

所有线程在执行 synchronized 时都会同步。

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
     */
    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);
                System.out.println(this.name + " 系统启动成功");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
    }
    /**
     * 编码
     */
    synchronized public void coding(){
            try {
                System.out.println(this.name + " 双击 Idea");
                Thread.sleep(500);
```

```
                System.out.println(this.name + " Idea 启动完毕");
                Thread.sleep(500);
                System.out.println(this.name + " 开开心心的写代码");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    /**
     * 去卫生间
     */
    public void wc(){
        synchronized ("suibian") {
            try {
                System.out.println(this.name + " 打开卫生间门");
                Thread.sleep(500);
                System.out.println(this.name + " 开始排泄");
                Thread.sleep(500);
                System.out.println(this.name + " 冲水");
                Thread.sleep(500);
                System.out.println(this.name + " 离开卫生间");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

/**
 * 打开电脑的工作线程
 */
class Working1 extends Thread{
    private  Programmer p;
    public Working1(Programmer p){
        this.p = p;
    }
```

```
    @Override
    public void run() {
        this.p.computer();
    }
}

/**
 * 编写代码的工作线程
 */
class Working2 extends Thread{
    private  Programmer p;
    public Working2(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.coding();
    }
}

/**
 * 去卫生间的线程
 */
class WC extends Thread{
    private  Programmer p;
    public WC(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.wc();
    }
}
public class TestSyncThread {
    public static void main(String[] args) {

        /*Programmer p = new Programmer("张三");

        new Working1(p).start();
        new Working2(p).start();*/

        Programmer p = new Programmer("张三");
```

```
        Programmer p1 = new Programmer("李四");

        Programmer p2 = new Programmer("王五");
        new WC(p).start();
        new WC(p1).start();
        new WC(p2).start();
    }
}
```

5.3使用 Class 作为线程对象锁

在不同线程中，拥有相同 Class 对象中的 synchronized 会互斥。

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){
        this.name = name;
    }
    /**
     * 领取奖金
     */
    synchronized  public static void money(){
            try {
                System.out.println(Thread.currentThread().getName() +
```

```
" 被领导表扬");
                Thread.sleep(500);
                System.out.println(Thread.currentThread().getName() +
" 拿钱");
                Thread.sleep(500);
                System.out.println(Thread.currentThread().getName() +
" 对公司表示感谢");
                Thread.sleep(500);
                System.out.println(Thread.currentThread().getName() +
" 开开心心的拿钱走人");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
}
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);
                System.out.println(this.name + " 系统启动成功");
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
        }
    }
    /**
     * 编码
     */
    synchronized public void coding(){
        try {
            System.out.println(this.name + " 双击 Idea");
            Thread.sleep(500);
            System.out.println(this.name + " Idea 启动完毕");
            Thread.sleep(500);
            System.out.println(this.name + " 开开心心的写代码");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    /**
     * 去卫生间
     */
    public void wc(){
        synchronized ("suibian") {
            try {
                System.out.println(this.name + " 打开卫生间门");
                Thread.sleep(500);
                System.out.println(this.name + " 开始排泄");
                Thread.sleep(500);
                System.out.println(this.name + " 冲水");
                Thread.sleep(500);
                System.out.println(this.name + " 离开卫生间");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
    /**
     * 领取奖金
     */
    public void money(){
        synchronized (Programmer.class) {
            try {
                System.out.println(this.name + " 被领导表扬");
                Thread.sleep(500);
                System.out.println(this.name + " 拿钱");
                Thread.sleep(500);
                System.out.println(this.name + " 对公司表示感谢");
                Thread.sleep(500);
                System.out.println(this.name + " 开开心心的拿钱走人");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

/**
 * 打开电脑的工作线程
 */
class Working1 extends Thread{
    private  Programmer p;
    public Working1(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.computer();
    }
}

/**
 * 编写代码的工作线程
```

```
 */
class Working2 extends Thread{
    private  Programmer p;
    public Working2(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.coding();
    }
}

/**
 * 去卫生间的线程
 */
class WC extends Thread{
    private  Programmer p;
    public WC(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.wc();
    }
}

/**
 * 程序员领取奖金
 */
class ProgrammerMoney extends Thread{
    private  Programmer p;
    public ProgrammerMoney(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.money();
    }
}

/**
```

```
 * 销售部门领取奖金
 */
class SaleMoneyThread extends  Thread{
    private  Sale p;
    public SaleMoneyThread(Sale p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.money();
    }
}

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

        /*Programmer p = new Programmer("张三");

        new Working1(p).start();
        new Working2(p).start();*/

        /*Programmer p = new Programmer("张三");

        Programmer p1 = new Programmer("李四");

        Programmer p2 = new Programmer("王五");

        new WC(p).start();
        new WC(p1).start();
        new WC(p2).start();*/

        /*Programmer p = new Programmer("张三");

        Programmer p1 = new Programmer("李四");

        new ProgrammerMoney(p).start();
        new ProgrammerMoney(p1).start();*/

        Sale sale = new Sale("王晓丽");

        Sale sale1 = new Sale("张丽丽");

        new SaleMoneyThread(sale).start();
        new SaleMoneyThread(sale1).start();

        /*Programmer p1 = new Programmer("李四");
```

```
        new ProgrammerMoney(p1).start();
        Sale sale1 = new Sale("张丽丽");
        new SaleMoneyThread(sale1).start();*/

    }
}
```

5.4使用自定义对象作为线程对象锁

在不同线程中，拥有相同自定义对象中的 synchronized 会互斥。

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 {
                System.out.println(Thread.currentThread().getName() +
" 被领导表扬");
                Thread.sleep(500);
                System.out.println(Thread.currentThread().getName() +
```

```
" 拿钱");
                Thread.sleep(500);
                System.out.println(Thread.currentThread().getName() +
" 对公司表示感谢");
                Thread.sleep(500);
                System.out.println(Thread.currentThread().getName() +
" 开开心心的拿钱走人");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
}
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);
                System.out.println(this.name + " 系统启动成功");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
    }
    /**
```

```
     * 编码
     */
    synchronized public void coding(){
            try {
                System.out.println(this.name + " 双击 Idea");
                Thread.sleep(500);
                System.out.println(this.name + " Idea 启动完毕");
                Thread.sleep(500);
                System.out.println(this.name + " 开开心心的写代码");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    /**
     * 去卫生间
     */
    public void wc(){
        synchronized ("suibian") {
            try {
                System.out.println(this.name + " 打开卫生间门");
                Thread.sleep(500);
                System.out.println(this.name + " 开始排泄");
                Thread.sleep(500);
                System.out.println(this.name + " 冲水");
                Thread.sleep(500);
                System.out.println(this.name + " 离开卫生间");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    /**
     * 领取奖金
```

```
     */
    public void money(){
        synchronized (Programmer.class) {
            try {
                System.out.println(this.name + " 被领导表扬");
                Thread.sleep(500);
                System.out.println(this.name + " 拿钱");
                Thread.sleep(500);
                System.out.println(this.name + " 对公司表示感谢");
                Thread.sleep(500);
                System.out.println(this.name + " 开开心心的拿钱走人");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
class Manager{
    private String name;
    public Manager(String name){
        this.name = name;
    }
    public String getName(){
        return this.name;
    }
    /**
     * 敬酒
     */
    public void cheers(String mName,String eName){
            try {
                System.out.println(mName + " 来到 " + eName + " 面前");
                Thread.sleep(500);
                System.out.println(eName + " 拿起酒杯");
                Thread.sleep(500);
                System.out.println(mName + " 和 " + eName + " 干杯");
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
    }
}
/**
 * 打开电脑的工作线程
 */
class Working1 extends Thread{
    private  Programmer p;
    public Working1(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.computer();
    }
}

/**
 * 编写代码的工作线程
 */
class Working2 extends Thread{
    private  Programmer p;
    public Working2(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.coding();
    }
}

/**
 * 去卫生间的线程
 */
class WC extends Thread{
    private  Programmer p;
    public WC(Programmer p){
        this.p = p;
```

```
    }
    @Override
    public void run() {
        this.p.wc();
    }
}

/**
 * 程序员领取奖金
 */
class ProgrammerMoney extends Thread{
    private  Programmer p;
    public ProgrammerMoney(Programmer p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.money();
    }
}

/**
 * 销售部门领取奖金
 */
class SaleMoneyThread extends  Thread{
    private  Sale p;
    public SaleMoneyThread(Sale p){
        this.p = p;
    }
    @Override
    public void run() {
        this.p.money();
    }
}

/**
 * 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) {

        /*Programmer p = new Programmer("张三");

        new Working1(p).start();
        new Working2(p).start();*/

        /*Programmer p = new Programmer("张三");

        Programmer p1 = new Programmer("李四");

        Programmer p2 = new Programmer("王五");

        new WC(p).start();
        new WC(p1).start();
        new WC(p2).start();*/

        /*Programmer p = new Programmer("张三");

        Programmer p1 = new Programmer("李四");

        new ProgrammerMoney(p).start();
        new ProgrammerMoney(p1).start();*/

       /* Sale sale = new Sale("王晓丽");

        Sale sale1 = new Sale("张丽丽");

        new SaleMoneyThread(sale).start();
        new SaleMoneyThread(sale1).start();*/

        /*Programmer p1 = new Programmer("李四");

        new ProgrammerMoney(p1).start();
```

```
        Sale sale1 = new Sale("张丽丽");
        new SaleMoneyThread(sale1).start();*/
        Manager manager = new Manager("张三丰");
        new CheersThread("张三",manager).start();
        new CheersThread("李四",manager).start();

    }
}
```

6 死锁及解决方案

6.1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。

因此， 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”。

6.2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

    private String girlName;
    static Lipstick lipstick = new Lipstick();
    static Mirror mirror = new Mirror();

    public void setFlag(int flag) {
        this.flag = flag;
    }

    public void setGirlName(String girlName) {
        this.girlName = girlName;
    }

    @Override
    public void run() {
        this.doMakeup();
    }
    /**
     * 开始化妆
     */
    public void doMakeup(){
        if(flag == 0){
            synchronized (lipstick){

                System.out.println(this.girlName+" 拿着口红");
```

```
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (mirror){

                    System.out.println(this.girlName+" 拿着镜子");

                }
            }
        }else{
            synchronized (mirror){

                System.out.println(this.girlName+" 拿着镜子");

                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (lipstick){

                    System.out.println(this.girlName+" 拿着口红");

                }
            }
        }
    }
}

public class DeadLockThread {
    public static void main(String[] args) {
        Makeup makeup = new Makeup();
        makeup.setFlag(0);

        makeup.setGirlName("大丫");

        Makeup makeup1 = new Makeup();
        makeup1.setFlag(1);

        makeup1.setGirlName("小丫");

        makeup.start();
        makeup1.start();
    }
}
```

6.3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();

    public void setFlag(int flag) {
        this.flag = flag;
    }

    public void setGirlName(String girlName) {
        this.girlName = girlName;
    }

    @Override
    public void run() {
```

```
        this.doMakeup();
    }
    /**
     * 开始化妆
     */
    public void doMakeup(){
        if(flag == 0){
            synchronized (lipstick){
                System.out.println(this.girlName+" 拿着口红");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            synchronized (mirror){
                System.out.println(this.girlName+" 拿着镜子");
            }
        }else{
            synchronized (mirror){
                System.out.println(this.girlName+" 拿着镜子");
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            synchronized (lipstick){
                System.out.println(this.girlName+" 拿着口红");
            }
        }
    }
}

public class DeadLockThread {
    public static void main(String[] args) {
        Makeup makeup = new Makeup();
        makeup.setFlag(0);
```

```
        makeup.setGirlName("大丫");

        Makeup makeup1 = new Makeup();
        makeup1.setFlag(1);

        makeup1.setGirlName("小丫");

        makeup.start();
        makeup1.start();
    }
}
```

七、 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”。

1 角色介绍

1.1什么是生产者？

生产者指的是负责生产数据的模块。

1.2什么是消费者？

消费者指的是负责处理数据的模块。

1.3什么是缓冲区？

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：

- **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。

- **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 。

2 实现生产者与消费者模式

2.1创建缓冲区

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

```
/**
 * 定义缓冲区类
 */
class SyncStack{
    //定义存放馒头的盒子
    private ManTou[] mt = new ManTou[10];
    //定义操作盒子的索引
    private int index;

    /**
     * 放馒头
     */
    public synchronized void push(ManTou manTou){
        //判断盒子是否已满
        while(this.index == this.mt.length){
            try {
                /**
                 * 语法：wait(),该方法必须要在 synchronized 块中调用。
                 * wait 执行后，线程会将持有的对象锁释放，并进入阻塞状态，
                 * 其他需要该对象锁的线程就可以继续运行了。
                 */
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //唤醒取馒头的线程
        /**
         * 语法：该方法必须要在 synchronized 块中调用。
         * 该方法会唤醒处于等待状态队列中的一个线程。
         */
        this.notify();
```

```
        this.mt[this.index] = manTou;
        this.index++;
    }
    /**
     * 取馒头
     */
    public synchronized ManTou pop(){
        while(this.index == 0){
            try {
                /**
                 * 语法：wait(),该方法必须要在 synchronized 块中调用。
                 * wait 执行后，线程会将持有的对象锁释放，并进入阻塞状态，
                 * 其他需要该对象锁的线程就可以继续运行了。
                 */
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.notify();
        this.index--;
        return this.mt[this.index];
    }
}
```

2.2创建生产者消费者线程

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

```
    }
}

/**
 * 定义缓冲区类
 */
class SyncStack{
    //定义存放馒头的盒子
    private ManTou[] mt = new ManTou[10];
    //定义操作盒子的索引
    private int index;

    /**
     * 放馒头
     */
    public synchronized void push(ManTou manTou){
        //判断盒子是否已满
        while(this.index == this.mt.length){
            try {
                /**
                 * 语法：wait(),该方法必须要在 synchronized 块中调用。
                 * wait 执行后，线程会将持有的对象锁释放，并进入阻塞状态，
                 * 其他需要该对象锁的线程就可以继续运行了。
                 */
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //唤醒取馒头的线程
        /**
         * 语法：该方法必须要在 synchronized 块中调用。
```

```
        * 该方法会唤醒处于等待状态队列中的一个线程。
        */
        this.notify();
        this.mt[this.index] = manTou;
        this.index++;
    }
    /**
     * 取馒头
     */
    public synchronized ManTou pop(){
        while(this.index == 0){
            try {
                /**
                 * 语法：wait(),该方法必须要在 synchronized 块中调用。
                 * wait 执行后，线程会将持有的对象锁释放，并进入阻塞状态，
                 * 其他需要该对象锁的线程就可以继续运行了。
                 */
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.notify();
        this.index--;
        return this.mt[this.index];
    }
}

/**
 * 定义生产者线程类
 */
class ShengChan extends Thread{
    private SyncStack ss;
    public ShengChan(SyncStack ss){
        this.ss = ss;
    }
    @Override
```

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

          System.out.println("生产馒头："+i);

          ManTou manTou = new ManTou(i);
          this.ss.push(manTou);
       }
    }
}

/**
 * 定义消费者线程类
 */
class XiaoFei extends Thread{
    private SyncStack ss;
    public XiaoFei(SyncStack ss){
        this.ss = ss;
    }
    @Override
    public void run() {
        for(int i=0;i<10;i++){
           ManTou manTou = this.ss.pop();

            System.out.println("消费馒头："+i);

        }
    }
}
public class ProduceThread {
    public static void main(String[] args) {
        SyncStack ss = new SyncStack();
        new ShengChan(ss).start();
        new XiaoFei(ss).start();
    }
}
```

3 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。