程序中的并发
多线程程序设计

主讲老师：申雪萍

主要内容

- **进程的概念**
- **线程的概念**
- 线程的调度
- 创建和启动线程
  - Thread线程类、
  - Runnable接口
- 多个线程的同步
- 线程之间的通信

线程的互斥锁和线程同步 (synchronized)

背景

- 在多线程环境中，两个或者两个以上的线程访问一个共享对象的情况是很普遍的
- 比如某公司在银行里有一个帐号，其在北京和上海的营业部都可以使用，那么如果北京的营业部在存款的时候，上海的营业部正好在取款，会发生什么事呢？

背景

- 线程安全是并发编程中的重要关注点，造成线程安全问题的主要诱因有两点：
  - 一是存在共享数据(也称临界资源)
  - 二是存在多条线程共同操作共享数据

共享数据的线程安全问题怎样处理？

- 自然而然的想法就是每一个线程依次去读写这个共享变量，这样就不会有任何数据安全的问题，因为每个线程所操作的都是当前最新的版本数据。
- Java关键字synchronized就具有使每个线程依次排队操作共享变量的功能。

示例代码:ThreadDemo.java

```
public class ThreadDemo{
    public static void main(String[] args){
    TestThread tt=new TestThread();
    new Thread(tt).start();
    new Thread(tt).start();
    new Thread(tt).start();
    new Thread(tt).start();
    }
}
class TestThread implements Runnable
    int tickets =100;
    public void run(){
       while(true){
         if(tickets>0){
        try{Thread.sleep(10);}
            catch(Exception e){}
 System.out.println(Thread.currentThread().getName()+"is sailing ticket"+tickets--);
              }
       }
    }
}
```

运行结果

```
Thread-4is sailing ticket21
Thread-1is sailing ticket20
Thread-2is sailing ticket19
Thread-3is sailing ticket18
Thread-4is sailing ticket17
Thread-1is sailing ticket16
Thread-2is sailing ticket15
Thread-3is sailing ticket14
Thread-4is sailing ticket13
Thread-1is sailing ticket12
Thread-2is sailing ticket11
Thread-3is sailing ticket10
Thread-4is sailing ticket9
Thread-1is sailing ticket8
Thread-2is sailing ticket7
Thread-3is sailing ticket6
Thread-4is sailing ticket5
Thread-1is sailing ticket4
Thread-2is sailing ticket3
Thread-3is sailing ticket2
Thread-4is sailing ticket1
Thread-1is sailing ticket0
Thread-2is sailing ticket-1
Thread-3is sailing ticket-2
```

线程的互斥锁

- 同步是一种保证共享资源完整性的手段；
- 具体作法是在代表原子操作的程序代码前加上synchronized标记，这样的代码被称为同步代码块。
- Java中用关键字synchronized为共享资源加锁，在任何一个时刻只能有一个线程能取得加锁对象的钥匙，对加锁对象进行操作。从而达到了对共享资源访问时的保护。

线程的互斥锁

- synchronized关键字可以使用在：
  - ①1. 一个成员方法上
  - ②2. 一个静态方法上
  - ③3. 一个语句块上

示例代码：ThreadDemo1.java

```
public class ThreadDemo1{
    public static void main(String[] args){
        TestThread tt=new TestThread();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
    }
}
class TestThread implements Runnable{
        int tickets =100;
        String str=new String("hello");
        public void run(){
//  String str=new String("hello");  //这样不可以，因为str每次是一个新的对象所以要使用全局的str
                while(true){
                   synchronized(str){
                        if(tickets>0){
                                try{Thread.sleep(10);}
                                catch(Exception e){}
        System.out.println(Thread.currentThread().getName()+"is sailing ticket"+tickets--);
                        }
                   }
                }
        }
}
```

运行结果

```
Thread-3is sailing ticket24
Thread-4is sailing ticket23
Thread-1is sailing ticket22
Thread-2is sailing ticket21
Thread-3is sailing ticket20
Thread-4is sailing ticket19
Thread-1is sailing ticket18
Thread-2is sailing ticket17
Thread-3is sailing ticket16
Thread-4is sailing ticket15
Thread-1is sailing ticket14
Thread-2is sailing ticket13
Thread-3is sailing ticket12
Thread-4is sailing ticket11
Thread-1is sailing ticket10
Thread-2is sailing ticket9
Thread-3is sailing ticket8
Thread-4is sailing ticket7
Thread-1is sailing ticket6
Thread-2is sailing ticket5
Thread-3is sailing ticket4
Thread-4is sailing ticket3
Thread-1is sailing ticket2
Thread-2is sailing ticket1
```

示例代码：ThreadDemo2.java

```
public class ThreadDemo2{
    public static void main(String[] args){
        TestThread tt=new TestThread();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
    }
}
class TestThread implements Runnable{
        int tickets =100;
        public void run(){
                while(true){
                        sale();
                }
        }
        public synchronized void sale(){
                        if(tickets>0){
                                try{Thread.sleep(10);}
                                catch(Exception e){}
System.out.println(Thread.currentThread().getName()+"is sailing ticket"+tickets--);
                        }
        }
}
```

运行结果

```
Thread-3is sailing ticket24
Thread-4is sailing ticket23
Thread-1is sailing ticket22
Thread-2is sailing ticket21
Thread-3is sailing ticket20
Thread-4is sailing ticket19
Thread-1is sailing ticket18
Thread-2is sailing ticket17
Thread-3is sailing ticket16
Thread-4is sailing ticket15
Thread-1is sailing ticket14
Thread-2is sailing ticket13
Thread-3is sailing ticket12
Thread-4is sailing ticket11
Thread-1is sailing ticket10
Thread-2is sailing ticket9
Thread-3is sailing ticket8
Thread-4is sailing ticket7
Thread-1is sailing ticket6
Thread-2is sailing ticket5
Thread-3is sailing ticket4
Thread-4is sailing ticket3
Thread-1is sailing ticket2
Thread-2is sailing ticket1
```

关键字synchronized的使用方法(1)

- public synchronized void write();
  - 由synchronized关键字锁定的是一个对象,而不是一个方法,
  - 一个线程调用某个对象的synchronized方法,就锁定了这个对象，直到它从这个方法中退出,才释放了这个锁定。因此，一旦一个对象被某个线程锁定了,它的所有synchronized方法就都不允许别的线程调用了。
  - 如果一个线程调用该方法，则方法体中访问的对象都被锁定，不允许其他线程访问，以解决共享资源的访问冲突问题。

关键字synchronized的使用方法(2)

- public static synchronized int getValue();
- 一个线程调用一个类的static synchronized方法,就锁定了这个类对象,也就控制了所有static synchronized方法。即类中所有的静态同步方法在每个时刻只有一个可以执行

关键字synchronized的使用方法(3)

- synchronized (obj) {… … }
- 一个线程进入由synchronized (obj) {…}修饰的程序块，就锁定了由obj指定的那个对象。

注意事项

① 一个对象在某一时刻只能被一个线程锁定；

② 一个类可以有多个对象，不同的对象可以被不同的线程锁定

③ 一个线程可以锁定多个对象

Java关键字synchronized的用法

分类	具体分类	被锁的对象	伪代码

方法	实例方法	类的实例对象	//实例方法，锁住的是该类的实例对象
			public synchronized void method() {
			.....
			}

	静态方法	类对象	//静态方法，锁住的是类对象
			public static synchronized void method1(){
			.....
			}

代码块	实例对象	类的实例对象	//同步代码块，锁住的是该类的实例对象
			synchronized (this){
			.....
			}

	class对象	类对象	//同步代码块，锁住的是该类的类对象
			synchronized (SynchronizedDemo.class){
			.....
			}

	任意实例对象Object	实例对象Object	//同步代码块，锁住的是配置的实例对象
			//String对象作为锁
			String lock = "";
			synchronized (lock){
			.....
			}

误区

- 关键字synchronized可以作为Java方法修饰符，也可以作为Java方法内的语句。
- 被它修饰的代码部分往往被描述为临界区。这使很多人认为，由于代码被synchronized保护着，因此同一时刻只能有一个线程访问它。但这种观点是错误的。

澄清

- 对于Java类中的方法，关键字synchronized其实并不锁定该方法或该方法的部分代码，它只是锁定对象。
- synchronized方法或synchronized区段内的代码在同一时刻下可有多个线程执行，只要是对不同的对象调用该方法就可以。

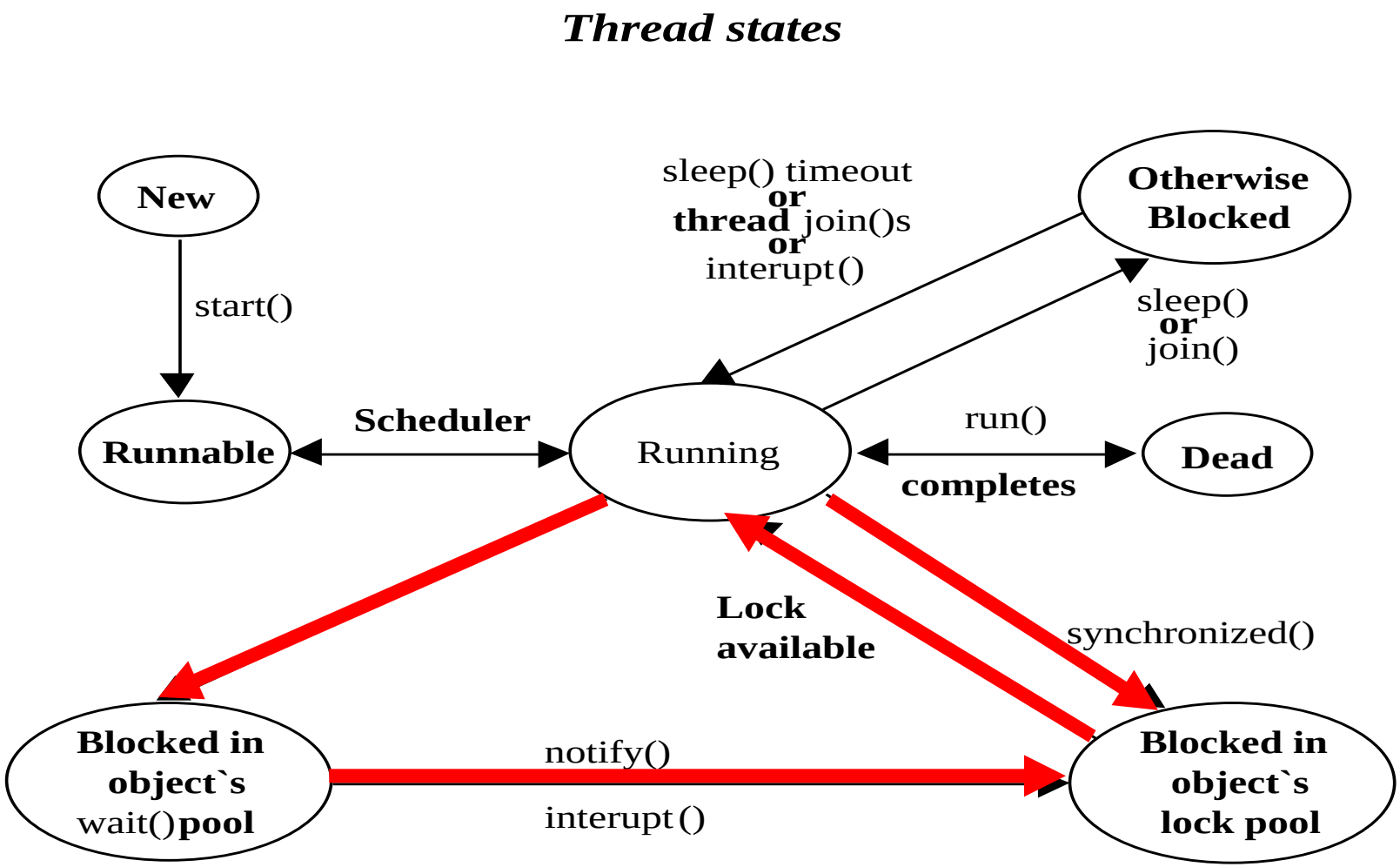
线程通信

- 不同的线程执行不同的任务，如果这些任务有某种联系，线程之间必须能够通信，协调完成工作。
- 例如生产者和消费者共同操作堆栈，当堆栈为空时，消费者无法取出产品，应该先通知生产者向堆栈中加入产品。当堆栈已满时，生产者无法继续加入产品，应该先通知消费者从堆栈中取出产品。

线程通信

- java.lang.Object类中提供了两个用于线程通信的方法：
  - wait()：执行该方法的线程释放对象的锁，Java虚拟机把该线程放到该对象的等待池中。该线程等待其他线程将它唤醒。
  - notify()：执行该方法的线程唤醒在对象的等待池中等待的一个线程。Java虚拟机从对象的等待池中随机的选择一个线程，把它转到对象的锁池中。

线程状态


Thread states
New
start()
Runnable
Scheduler
Running
sleep() timeout
or
thread join()s
or
interupt()
Otherwise
Blocked
sleep()
or
join()
run()
completes
Dead
Lock
available
synchronized()
Blocked in
object`s
wait()pool
notify()
interupt()
Blocked in
object`s
lock pool

线程状态说明

- 一般来说，每个共享对象的互斥锁存在两个队列，一个是锁等待队列，另一个是锁申请队列，锁申请队列中的第一个线程可以对该共享对象进行操作，而锁等待队列中的线程在某些情况下将移入到锁申请队列。

wait()、notify()和notifyAll()方法

- wait,nofity,notifyAll必须在已经持有锁的情况下执行,所以它们只能出现在synchronized作用的范围内，也就是出现在synchronized修饰的方法中。
- wait的作用:释放已持有的锁,进入等待队列.
- notify的作用:随机唤醒wait队列中的一个线程并把它移入锁申请队列
- notifyAll的作用:唤醒wait队列中的所有的线程并把它们移入锁申请队列.

wait()、notify()和notifyAll()方法

- wait,notify和notifyAll都是与同步相关联的方法,只有在synchronized方法中才可以用
- 一个“在同步方法中等待” 的线程自己是没有办法恢复运行的,只能等另一个也进入了该对象的synchronized方法的线程调用notify和notifyAll后才有可能恢复运行

例如：

如果银行帐户里的金额比要转帐的金额少，就等待从别的地方存入足够的金额后再继续处理。

这种情况也被称为“在同步方法中等待”

```
public synchronized void get(){
    … …
    if(account.amount<amount){
        …
        wait();
    }
    … …
}
```

方法wait()与sleep() 方法对比

① 方法wait()与sleep() 方法一样，都能使线程等待而停止运行

② 其区别在于sleep()方法不会释放对象的锁，而wait()方法进入等待时，可以释放对象的锁，因而别的线程能对这些加锁的对象进行操作。所以，wait,notify和notifyAll都是与同步相关联的方法,只有在synchronized方法中才可以用。

③ 在不同步的方法或代码中则使用sleep()方法使线程暂时停止运行。

示例代码

```
class Account{
        String name;
        long id;
        private double sum,saveMoney,getMoney;
        public Account(long id,String name,double sum){
                this.id=id;
                this.name=name;
                this.sum=sum;
        }
        public synchronized double get(double d){
                while(sum<50000){
                        try{
                                this.wait();
                        }
                        catch(InterruptedException e){

                        }
                }
                this.notify();
                getMoney=d;
                sum-=getMoney;
                return getMoney;
        }
```

示例代码

```
        public synchronized double save(double d){
                while(sum>=125000){
                        try{
                                this.wait();
                        }
                        catch(InterruptedException e){

                        }
                }
                this.notify();
                saveMoney=d;
                sum+=saveMoney;
                return saveMoney;
        }
        public double getSum(){
                return sum;
        }
        public String toString(){
                return "账号："+id+"；姓名"+name+"；存款余额"+sum;
        }
}
```

示例代码

```
class GetMoney implements Runnable{
        Account account1;
        public GetMoney(Account a){
                account1=a;
        }
        public void run(){
                for(int i=0;i<4;i++){
System.out.print("取钱: "+account1.get(50000)+"\t账户余额
sum="+account1.getSum()+"\n");
                        try{
                                Thread.sleep(500);
                        }
                        catch(InterruptedException e){}
                }
        }
}
```

示例代码

```
class SaveMoney implements Runnable{
        Account account1;
        public SaveMoney(Account a){
                account1=a;
        }
        public void run(){
                for(int i=0;i<8;i++){
System.out.print("存钱: "+account1.save(25000)+"\t账户余额 : sum="+account1.getSum()+"\n");
                        try{
                                Thread.sleep(50);
                        }
                        catch(InterruptedException e){

                        }
                }
        }
}
```

案例：对同一个账户存钱，取钱并行

```
public class Test{
        public static void main(String[] args){
                //建立一个账户，存款为0
                Account ac=new Account(123456789,"Tom",0.0);
                System.out.println(ac.toString()+": 存取款操作如下");
                Thread p=new Thread(new GetMoney(ac));//////////////////
                Thread c=new Thread(new SaveMoney(ac));//////////////////
                p.start();
                c.start();
        }
}
```

```
账号：123456789：姓名Tom；存款余额0.0：存取款操作如下
存钱：25000.0        账户余额：sum=25000.0
存钱：25000.0        账户余额：sum=0.0
取钱：50000.0        账户余额： sum=0.0
存钱：25000.0        账户余额：sum=25000.0
存钱：25000.0        账户余额：sum=50000.0
存钱：25000.0        账户余额：sum=75000.0
存钱：25000.0        账户余额：sum=100000.0
存钱：25000.0        账户余额：sum=125000.0
取钱：50000.0        账户余额： sum=75000.0
存钱：25000.0        账户余额：sum=100000.0
存钱：25000.0        账户余额：sum=125000.0
取钱：50000.0        账户余额： sum=100000.0
存钱：25000.0        账户余额：sum=100000.0
存钱：25000.0        账户余额：sum=125000.0
取钱：50000.0        账户余额： sum=100000.0
```

案例：PV操作

- 多个生产者生产产品，多个消费者消费产品，产品要提供一个含有标识的id属性，另外需要在生产或消费时打印产品的详细内容；
- 店员一次最多只能持有10份产品，如果生产者生产的产品多于10份，则会让生产者线程等待；
- 当店员没有产品消费时，消费线程等待；

分析

- 多个生产者线程和多个消费者线程都需要面向店员，生产者线程生产的产品交给店员消费，消费者线程从店员那里获得产品消费；
- 店员在操作产品时，需要做到互斥；
- 产品类：Products
- 店员类：Clerk
- 生产者线程：Producer
- 消费者线程：Consumer

```
public class Products {
    int id;
   public Products() {
        super();
   }
   public Products(int id) {
        super();
        this.id = id;
   }
   public String toString() {
        return "Products [id=" + id + "]";
   }
}
```

```
public class Clerk {
    int index = 0;// 默认为0个产品
    Products[] pro = new Products[10];
    // 生产者生产出来的产品交给店员
    public synchronized void addProduct(Products pd) {
        while (index == pro.length) {
            try {
                this.wait();// 产品已满，请稍后再生产
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.notify();// 通知等待区的消费者可以取产品了
        pro[index] = pd;
        index++;
    }
```

```
    // 消费者从店员处取产品
    public synchronized Products getProduct() {
        while (index == 0) {
            try {
                this.wait(); // 缺货，请稍后再取。
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.notify();// 通知等待区的生产者可以生产产品了
        index--;
        return pro[index];
    }
}
```

```java
public class Producer implements Runnable {
    private Clerk clerk;
    public Producer() {
        super();
    }
    public Producer(Clerk clerk) {
        super();
        this.clerk = clerk;
    }
    public void run() {
        System.out.println("生产者开始生产产品");
        for (int i = 0; i < 15; i++) {
            // 注意此处的循环次数一定要大于pro数组的长度（10）
            Products pd = new Products(i);
            clerk.addProduct(pd);// 生产产品
            System.out.println("生产了：" + pd);
            try { // 睡眠时间用随机产生的值来设置
                Thread.sleep((int) (Math.random() * 10) * 100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
public class Consumer implements Runnable {
    private Clerk clerk;
    public Consumer() {
        super();
    }
    public Consumer(Clerk clerk) {
        super();
        this.clerk = clerk;
    }
    public void run() {
        System.out.println("消费者开始取走产品");
        for (int i = 0; i < 15; i++) {
            // 注意此处的循环次数一定要大于pro数组的长度（10）
            // 取产品
            Products pd = clerk.getProduct();
            System.out.println("消费了：" + pd);
            try { // 睡眠时间用随机产生的值来设置
                Thread.sleep((int) (Math.random() * 10) * 100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class ProducerAndConsumerTest {
    public static void main(String[] args) {
        Clerk clerk = new Clerk();
        Thread producerThread = new Thread(new Producer(clerk));
        Thread consumerThread = new Thread(new Consumer(clerk));
        Thread producerThread1 = new Thread(new Producer(clerk));
        Thread consumerThread2 = new Thread(new Consumer(clerk));
        producerThread.start();
        consumerThread.start();
        producerThread1.start();
        consumerThread2.start();
    }
}
```

```
消费了：Products [id=4]
生产了：Products [id=3]
消费了：Products [id=3]
消费了：Products [id=2]
消费了：Products [id=1]
生产了：Products [id=5]
消费了：Products [id=5]
生产了：Products [id=4]
消费了：Products [id=4]
```

案例

- 在数据结构上（stack）实现生产者和消费者问题
- PV.java

小结

- 线程之间交换信息称为线程通信，可以认为，wait()和notify()方法是java采用的一种简单的线程间通信机制，利用它们，彼此间只传送一个信号。
- 线程间要传送的数据较多时，必须使用其它的像共享主存、管道流等通信方式。

  ① 主存读/写通信:Java输入输出流

  ② 管道流通信:

管道流通信

- 管道用来把一个程序输出连接到另一程序输入。java.io中提供了类PipedInputStream和PipedOutputStream作为管道的输入/输出部件。
  - 构造方法中，对于管道输入/输出流来说，对应管道输出/输入流作为参数以进行连接；
  - 管道输入/输出流还提供了方法connect()以进行相应的连接。

示例代码

- **TestPipedStream.java**

```
import java.io.*;
import java.util.*;

public class TestPipedStream {
        public static void main(String[] args) {
                try {
                        PipedOutputStream out = new PipedOutputStream();
                        PipedInputStream in = new PipedInputStream();
                        out.connect(in);
                        ThreadOut to = new ThreadOut(out);
                        ThreadIn ti = new ThreadIn(in);
                        to.start();
                        ti.start();
                } catch (IOException e) {
                        System.out.println(e);
                }
        }
}
```

```
class ThreadOut extends Thread {
        private int i1 = 12345;
        private boolean b1 = true;
        private char c1 = 'a';
        private String s1 = "Java 程序设计";
        private DataOutputStream dos;

        public ThreadOut(PipedOutputStream out) {
                dos = new DataOutputStream(out);
        }

        public void run() {
                System.out.println("输出管道发送以数据");
                System.out.println("\t整数：" + i1 + "\t逻辑值：" + b1 + "\t字符：" + c1
                                + "\t字符串：" + s1);
                try {
                        dos.writeInt(i1);
                        dos.writeBoolean(b1);
                        dos.writeChar(c1);
                        dos.writeUTF(s1);
                        dos.close();
                } catch (IOException e) {
                }
        }
}
```

```
class ThreadIn extends Thread {
        private DataInputStream dis;

        public ThreadIn(PipedInputStream in) {
                dis = new DataInputStream(in);
        }

        public void run() {
                try {
                        int i2 = dis.readInt();
                        boolean b2 = dis.readBoolean();
                        char c2 = dis.readChar();
                        String s2 = dis.readUTF();
                        System.out.println("这是输入管道接收到的输出管道发送的数据：");
                        System.out.println("\t整数：" + i2 + "\t逻辑值：" + b2 + "\t字符：" + c2
                                        + "\t字符串" + s2);
                        dis.close();
                } catch (IOException e) {
                        System.out.println(e);
                }
        }
}
```

死锁（1）

- 在多线程竞争使用多资源的程序中，有可能出现死锁的情况。死锁不是由于资源短缺造成的，而是由于程序设计不合理而造成程序中所有的线程都陷入等待态或阻塞态。
- 这种情况发生在一个线程等待另一个线程所持有的锁，而那个线程又在等待第一个线程持有的锁的时候。

死锁（2）

- 每个线程都不能继续运行，除非另一线程运行完同步程序块。
- 因为哪个线程都不能继续运行，所以哪个线程都无法运行完同步程序块。
- 另一种情况是在wait,notify结构中,如果所有的线程都wait了,就不会有线程来notify它们,因此要特别注意;

后台线程

- 后台线程是指为其他线程提供服务的线程，也称为守护线程（Daemon Threads ）。
- 守护线程是一类特殊的线程，它和普通线程的区别在于它并不是应用程序的核心部分，当一个应用程序的所有非守护线程终止运行时，即使仍然有守护线程在运行，应用程序也将终止.
- 反之，只要有一个非守护线程在运行，应用程序就不会终止。
- 主线程默认情况下是前台线程，由前台线程创建的线程默认情况下也是前台线程。

守护线程(Daemon Threads)

- 可以通过调用方法 isDaemon() 来判断一个线程是否是守护线
- 程，也可以调用方法 setDaemon() 来将一个线程设为守护线程。
  - *Garbage collector is a daemon thread*

守护线程(Daemon Threads)

```
public class Machine extends Thread {
    private int a;
    private static int count;
    public void start() {
        super.start();
        // 匿名线程类
        Thread deamon = new Thread() {
            public void run() {
                while (true) { // 无限循环
                    reset();
                    try {
                        sleep(50);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        };
        deamon.setDaemon(true);
        deamon.start();
    }
```

```
    public void reset() {
        a = 0;
    }
    public void run() {
        while (true) {
            System.out.println(getName() + ":" + a++);
            if (count++ == 100)
                break;
            yield();
        }
    }
    public static void main(String args[]) throws Exception {
        Machine machine = new Machine();
        machine.start();
    }
}
```

定时器Timer

- 在JDK的java.util包中提供了一个实用类Timer，它能够定时执行特定的任务。
- TimerTask类表示定时器执行的一项任务。
- Timer类的schedule(TimerTask task, long delay, long period)方法用来设置定时器需要定时执行的任务。task参数表示任务，delay参数表示延迟执行的时间，以毫秒为单位，period参数表示每次执行任务的间隔时间，以毫秒为单位。

定时器

- timer.schedule(task,10,50);
- 以上代码表明定时器将在10毫秒以后开始执行task任务（即执行TimerTask实例的run()方法），以后每隔50毫秒重复执行一次task任务。

定时器

```
import java.util.Timer;
import java.util.TimerTask;
public class Machine1 extends Thread {
    private int a;
    public void start() {
        super.start();
        // 把与Timer关联的线程设为后台线程
        Timer timer = new Timer(true);
        // 匿名类
        TimerTask task = new TimerTask() {
            public void run() {
                reset();
            }
        };
        // 设置定时任务
        timer.schedule(task, 10, 50);
    }
```

```
    public void reset() {
        a = 0;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println(getName() + ":" + a++);
            yield();
        }
    }

    public static void main(String args[]) throws Exception {
        Machine1 machine = new Machine1();
        machine.start();
    }
}
```

思考：多线程问题

- 对多线程程序本身来说，它会对系统产生以下影响：
  - 线程需要占用内存。
  - 线程过多，会消耗大量CPU时间来跟踪线程。
  - 必须考虑多线程同时访问共享资源的问题，如果没有协调好，就会产生令人意想不到的问题，例如可怕的死锁和资源竞争。
  - 因为同一个任务的所有线程都共享相同的地址空间，并共享任务的全局变量，所以程序也必须考虑多线程同时访问全局变量的问题。

小结（1）

- 可以采用 Thread 类的子类或实现 Runnable 接口，实现多线程
- 两个关键性操作：
  - 定义用户线程的操作，即定义用户线程的run( )方法；
  - 在适当的时候建立用户线程的实例。

小结（2）

- 继承 Thread 类
  - 用户创建自己的 Thread 类的子类，并在子类中重新定义自己的 run( )方法，run( )方法中包含了用户线程的操作。
  - 在需要建立自己的线程时，只需创建一个已定义好的Thread 子类对象就可以了。

小结（3）

- 实现 Runnable 接口
  - 编写实现 Runnable 接口的线程类，重写 Runnable 接口中的 run( )方法。
  - 在实现 Runnable 接口的同时还可以继承其它类。

小结（4）

- Java中用关键字synchronized为共享资源加锁，在任何一个时刻只能有一个线程能取得加锁对象的钥匙，对加锁对象进行操作。从而达到了对共享资源访问时的保护。
- synchronized关键字可以使用在：
  1. 一个成员方法上
  2. 一个静态方法上
  3. 一个语句块上

小结（5）

- 线程间传递信号：
  - wait,notify和notifyAll都是与同步相关联的方法,只有在synchronized方法中才可以用
- 主存读/写通信
- 管道流通信