Day06

新单词

- **synchronized**-同步
-

多线程

多线程并发安全问题

概念

当多个线程并发操作同一临界资源,由于线程切换时机不确定,导致操作临界资源的顺序出现混乱严重时可能导致系统瘫痪.
临界资源:操作该资源的全过程同时只能被单个线程完成.

例

当beans为1时，若两个线程同时调用getBean方法，t1线程先进行if判断，此时beans不为0，于是执行if后面的操作准备获取beans的值并对其进行--操作，但是还没有执行这句话发生了线程切换，那么t2线程也进行if判断，由于beans不为0，t2线程也执行if后面的操作获取beans的值并对其进行--操作，这会导致两个线程最终将beans的值从-减为了-1.导致后续操作出现死循环。

这就是由于线程切换不确定导致执行顺序出现了混乱，也就是所谓的并发安全问题

```
package thread;

/**
 * 多线程并发安全问题
 * 当多个线程并发操作同一临界资源，由于线程切换的时机不确定，导致操作顺序出现
 * 混乱，严重时可能导致系统瘫痪。
 * 临界资源:同时只能被单一线程访问操作过程的资源。
 */
public class SyncDemo {
    public static void main(String[] args) {
        Table table = new Table();
        Thread t1 = new Thread(){
            public void run(){
                while(true){
                    int bean = table.getBean();
                    Thread.yield();
                    System.out.println(getName()+":"+bean);
                }
            }
        };
        Thread t2 = new Thread(){
            public void run(){
                while(true){
                    int bean = table.getBean();
```

```
                    /*
                        static void yield()
                        线程提供的这个静态方法作用是让执行该方法的线程
                        主动放弃本次时间片。
                        这里使用它的目的是模拟执行到这里CPU没有时间了，发生
                        线程切换，来看并发安全问题的产生。
                     */
                    Thread.yield();
                    System.out.println(getName()+":"+bean);
                }
            }
        };
        t1.start();
        t2.start();
    }
}

class Table{
    private int beans = 20;//桌子上有20个豆子

    public int getBean(){
        if(beans==0){
            throw new RuntimeException("没有豆子了!");
        }
        Thread.yield();
        return beans--;
    }
}
```

synchronized关键字

解决并发安全问题的本质就是将多个线程并发(同时)操作改为同步(排队)操作来解决。

synchronized有两种使用方式

- 在方法上修饰,此时该方法变为一个同步方法
- 同步块,可以更准确的锁定需要排队的代码片段

同步方法

当一个方法使用synchronized修饰后,这个方法称为"同步方法",即:多个线程不能同时 在方法内部执行.只能有先后顺序的一个一个进行. 将并发操作同一临界资源的过程改为同步执行就可以有效的解决并发安全问题.

```
package thread;

/**
 * 多线程并发安全问题
 * 当多个线程并发操作同一临界资源，由于线程切换的时机不确定，导致操作顺序出现
 * 混乱，严重时可能导致系统瘫痪。
 * 临界资源:同时只能被单一线程访问操作过程的资源。
 */
public class SyncDemo {
    public static void main(String[] args) {
```

```
        Table table = new Table();
        Thread t1 = new Thread(){
            public void run(){
                while(true){
                    int bean = table.getBean();
                    Thread.yield();
                    System.out.println(getName()+":"+bean);
                }
            }
        };
        Thread t2 = new Thread(){
            public void run(){
                while(true){
                    int bean = table.getBean();
                    /*
                        static void yield()
                        线程提供的这个静态方法作用是让执行该方法的线程
                        主动放弃本次时间片。
                        这里使用它的目的是模拟执行到这里CPU没有时间了，发生
                        线程切换，来看并发安全问题的产生。
                     */
                    Thread.yield();
                    System.out.println(getName()+":"+bean);
                }
            }
        };
        t1.start();
        t2.start();
    }
}

class Table{
    private int beans = 20;//桌子上有20个豆子

    /**
     * 当一个方法使用synchronized修饰后，这个方法称为同步方法，多个线程不能
     * 同时执行该方法。
     * 将多个线程并发操作临界资源的过程改为同步操作就可以有效的解决多线程并发
     * 安全问题。
     * 相当于让多个线程从原来的抢着操作改为排队操作。
     */
    public synchronized int getBean(){
        if(beans==0){
            throw new RuntimeException("没有豆子了!");
        }
        Thread.yield();
        return beans--;
    }
}
```

同步块

有效的缩小同步范围可以在保证并发安全的前提下尽可能的提高并发效率.同步块可以更准确的控制需要多个线程排队执行的代码片段.

语法:

```
synchronized(同步监视器对象){
    需要多线程同步执行的代码片段
}
```

同步监视器对象即上锁的对象,要想保证同步块中的代码被多个线程同步运行,则要求多个线程看到的同步监视器对象是同一个.

```
package thread;

/**
 * 有效的缩小同步范围可以在保证并发安全的前提下尽可能提高并发效率。
 *
 * 同步块
 * 语法:
 * synchronized(同步监视器对象){
 *      需要多个线程同步执行的代码片段
 * }
 * 同步块可以更准确的锁定需要多个线程同步执行的代码片段来有效缩小排队范围。
 */
public class SyncDemo2 {
    public static void main(String[] args) {
        Shop shop = new Shop();
        Thread t1 = new Thread(){
            public void run(){
                shop.buy();
            }
        };
        Thread t2 = new Thread(){
            public void run(){
                shop.buy();
            }
        };
        t1.start();
        t2.start();
    }
}

class Shop{
    public void buy(){
        /*
            在方法上使用synchronized，那么同步监视器对象就是this。
         */
//    public synchronized void buy(){
        Thread t = Thread.currentThread();//获取运行该方法的线程
        try {
            System.out.println(t.getName()+":正在挑衣服...");
            Thread.sleep(5000);
            /*
```

```
                使用同步块需要指定同步监视器对象，即:上锁的对象
                这个对象可以是java中任何引用类型的实例，只要保证多个需要排队
                执行该同步块中代码的线程看到的该对象是"同一个"即可
             */
            synchronized (this) {
//            synchronized (new Object()) {//没有效果!
                System.out.println(t.getName() + ":正在试衣服...");
                Thread.sleep(5000);
            }

            System.out.println(t.getName()+":结账离开");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

在静态方法上使用synchronized

当在静态方法上使用synchronized后,该方法是一个同步方法.由于静态方法所属类,所以一定具有同步效果.

静态方法使用的同步监视器对象为当前类的类对象(Class的实例).

注:类对象会在后期反射知识点介绍.

```
package thread;

/**
 * 静态方法上如果使用synchronized，则该方法一定具有同步效果。
 */
public class SyncDemo3 {
    public static void main(String[] args) {
        Thread t1 = new Thread(){
            public void run(){
                Boo.dosome();
            }
        };
        Thread t2 = new Thread(){
            public void run(){
                Boo.dosome();
            }
        };
        t1.start();
        t2.start();
    }
}
class Boo{
    /**
     * synchronized在静态方法上使用是，指定的同步监视器对象为当前类的类对象。
     * 即:Class实例。
     * 在JVM中，每个被加载的类都有且只有一个Class的实例与之对应，后面讲反射
     * 知识点的时候会介绍类对象。
```

```
     */
    public synchronized static void dosome(){
            try {
                Thread t = Thread.currentThread();
                System.out.println(t.getName() + ":正在执行dosome方法...");
                Thread.sleep(5000);
                System.out.println(t.getName() + ":执行dosome方法完毕!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

静态方法中使用同步块时,指定的锁对象通常也是当前类的类对象

```
package thread;

public class SyncDemo3 {
    public static void main(String[] args) {
//        new Thread(()->Foo.dosome()).start();
//        new Thread(Foo::dosome).start();

        Foo f1 = new Foo();
        Foo f2 = new Foo();
        new Thread(()->f1.dosome()).start();
        new Thread(()->f2.dosome()).start();
    }
}

class Foo{
//    public synchronized static void dosome(){
    public static void dosome(){
        /*
            在静态方法中使用同步块时，同步监视器对象还是使用当前类的类对象
            获取类对象的方式:类名.class
            例如获取Foo的类对象就是:Foo.class
         */
        synchronized (Foo.class) {
            try {
                Thread t = Thread.currentThread();
                System.out.println(t.getName() + ":正在执行dosome方法");
                Thread.sleep(5000);
                System.out.println(t.getName() + ":执行dosome方法完毕");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

互斥锁

当多个线程执行不同的代码片段,但是这些代码片段之间不能同时运行时就要设置为互斥的.

使用synchronized锁定多个代码片段,并且指定的同步监视器是同一个时,这些代码片段之间就是互斥的.

```
package thread;

/**
 * 互斥锁
 * 当使用synchronized锁定多个不同的代码片段，并且指定的同步监视器对象相同时，
 * 这些代码片段之间就是互斥的，即:多个线程不能同时访问这些方法。
 */
public class SyncDemo4 {
    public static void main(String[] args) {
        Foo foo = new Foo();
        Thread t1 = new Thread(){
            public void run(){
                foo.methodA();
            }
        };
        Thread t2 = new Thread(){
            public void run(){
                foo.methodB();
            }
        };
        t1.start();
        t2.start();
    }
}
class Foo{
    public synchronized void methodA(){
        Thread t = Thread.currentThread();
        try {
            System.out.println(t.getName()+":正在执行A方法...");
            Thread.sleep(5000);
            System.out.println(t.getName()+":执行A方法完毕!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public synchronized void methodB(){
        Thread t = Thread.currentThread();
        try {
            System.out.println(t.getName()+":正在执行B方法...");
            Thread.sleep(5000);
            System.out.println(t.getName()+":执行B方法完毕!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

总结

守护线程与普通线程的区别:守护线程是通过普通线程调用setDaemon(true)设置而来的

主要区别体现在当java进程中所有的普通线程都结束时进程会结束，在结束前会杀死所有还在运行的守护线程。

重点:多线程并发安全问题

- 什么是多线程并发安全问题:

  当多个线程并发操作同一临界资源，由于线程切换时机不确定，导致执行顺序出现混乱。

  解决办法:

  将并发操作改为同步操作就可有效的解决多线程并发安全问题
- 同步与异步的概念:同步和异步都是说的多线程的执行方式。

  多线程各自执行各自的就是异步执行，而多线程执行出现了先后顺序进行就是同步执行
- synchronized的两种用法

  1.直接在方法上声明，此时该方法称为同步方法，同步方法同时只能被一个线程执行

  2.同步块，推荐使用。同步块可以更准确的控制需要同步执行的代码片段。

  有效的缩小同步范围可以在保证并发安全的前提下提高并发效率
- 同步监视器对象的选取:

  对于同步的成员方法而言，同步监视器对象不可指定，只能是this

  对于同步的静态方法而言，同步监视器对象也不可指定，只能是类对象

  对于同步块而言，需要自行指定同步监视器对象，选取原则:

  1.必须是引用类型

  2.多个需要同步执行该同步块的线程看到的该对象必须是**同一个**
- 互斥性

  当使用多个synchronized修饰了多个代码片段，并且指定的同步监视器都是同一个对象时，这些代码片段就是互斥的，多个线程不能同时在这些代码片段上执行。

聊天室(续)

实现服务端发送消息给客户端

在服务端通过Socket获取输出流,客户端获取输入流,实现服务端将消息发送给客户端.

这里让服务端直接将客户端发送过来的消息再回复给客户端来进行测试.

服务端代码:

```
package socket;

import java.io.*;
import java.net.ServerSocket;
import java.nio.charset.StandardCharsets;
import java.net.Socket;
```

```java
/**
 * 聊天室服务端
 */
public class Server {
    /**
     * 运行在服务端的ServerSocket主要完成两个工作:
     * 1:向服务端操作系统申请服务端口，客户端就是通过这个端口与ServerSocket建立链接
     * 2:监听端口，一旦一个客户端建立链接，会立即返回一个Socket。通过这个Socket
     *   就可以和该客户端交互了
     *
     * 我们可以把ServerSocket想象成某客服的"总机"。用户打电话到总机，总机分配一个
     * 电话使得服务端与你沟通。
     */
    private ServerSocket serverSocket;

    /**
     * 服务端构造方法，用来初始化
     */
    public Server(){
        try {
            System.out.println("正在启动服务端...");
            /*
                实例化ServerSocket时要指定服务端口，该端口不能与操作系统其他
                应用程序占用的端口相同，否则会抛出异常:
                java.net.BindException:address already in use

                端口是一个数字，取值范围:0-65535之间。
                6000之前的的端口不要使用，密集绑定系统应用和流行应用程序。
             */
            serverSocket = new ServerSocket(8088);
            System.out.println("服务端启动完毕!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 服务端开始工作的方法
     */
    public void start(){
        try {
            while(true) {
                System.out.println("等待客户端链接...");
                /*
                    ServerSocket提供了接受客户端链接的方法:
                    Socket accept()
                    这个方法是一个阻塞方法，调用后方法"卡住"，此时开始等待客户端
                    的链接，直到一个客户端链接，此时该方法会立即返回一个Socket实例
                    通过这个Socket就可以与客户端进行交互了。

                    可以理解为此操作是接电话，电话没响时就一直等。
                 */
                Socket socket = serverSocket.accept();
                System.out.println("一个客户端链接了！");
                //启动一个线程与该客户端交互
```

```
                ClientHandler clientHandler = new ClientHandler(socket);
                Thread t = new Thread(clientHandler);
                t.start();

            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }

    /**
     * 定义线程任务
     * 目的是让一个线程完成与特定客户端的交互工作
     */
    private class ClientHandler implements Runnable{
        private Socket socket;
        private String host;//记录客户端的IP地址信息

        public ClientHandler(Socket socket){
            this.socket = socket;
            //通过socket获取远端计算机地址信息
            host = socket.getInetAddress().getHostAddress();
        }
        public void run(){
            try{
                 /*
                    Socket提供的方法:
                    InputStream getInputStream()
                    获取的字节输入流读取的是对方计算机发送过来的字节
                 */
                InputStream in = socket.getInputStream();
                InputStreamReader isr = new InputStreamReader(in,
StandardCharsets.UTF_8);
                BufferedReader br = new BufferedReader(isr);

                OutputStream out = socket.getOutputStream();
                OutputStreamWriter osw = new
OutputStreamWriter(out,StandardCharsets.UTF_8);
                BufferedWriter bw = new BufferedWriter(osw);
                PrintWriter pw = new PrintWriter(bw,true);


                String message = null;
                while ((message = br.readLine()) != null) {
                    System.out.println(host + "说:" + message);
                    //将消息回复给客户端
                    pw.println(host + "说:" + message);
                }
            }catch(IOException e){
                e.printStackTrace();
```

```
            }
        }
    }


}
```

客户端代码:

```
package socket;

import java.io.*;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

/**
 * 聊天室客户端
 */
public class Client {
    /*
        java.net.Socket 套接字
        Socket封装了TCP协议的通讯细节，我们通过它可以与远端计算机建立链接，
        并通过它获取两个流(一个输入，一个输出)，然后对两个流的数据读写完成
        与远端计算机的数据交互工作。
        我们可以把Socket想象成是一个电话，电话有一个听筒(输入流)，一个麦克
        风(输出流)，通过它们就可以与对方交流了。
     */
    private Socket socket;

    /**
     * 构造方法，用来初始化客户端
     */
    public Client(){
        try {
            System.out.println("正在链接服务端...");
            /*
                实例化Socket时要传入两个参数
                参数1:服务端的地址信息
                      可以是IP地址，如果链接本机可以写"localhost"
                参数2:服务端开启的服务端口
                我们通过IP找到网络上的服务端计算机，通过端口链接运行在该机器上
                的服务端应用程序。
                实例化的过程就是链接的过程，如果链接失败会抛出异常:
                java.net.ConnectException: Connection refused: connect
             */
            socket = new Socket("localhost",8088);
            System.out.println("与服务端建立链接!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 客户端开始工作的方法
```

```
     */
    public void start(){
        try {
            /*
                Socket提供了一个方法:
                OutputStream getOutputStream()
                该方法获取的字节输出流写出的字节会通过网络发送给对方计算机。
             */
            //低级流，将字节通过网络发送给对方
            OutputStream out = socket.getOutputStream();
            //高级流，负责衔接字节流与字符流，并将写出的字符按指定字符集转字节
            OutputStreamWriter osw = new
OutputStreamWriter(out,StandardCharsets.UTF_8);
            //高级流，负责块写文本数据加速
            BufferedWriter bw = new BufferedWriter(osw);
            //高级流，负责按行写出字符串，自动行刷新
            PrintWriter pw = new PrintWriter(bw,true);

            //通过socket获取输入流读取服务端发送过来的消息
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new
InputStreamReader(in,StandardCharsets.UTF_8);
            BufferedReader br = new BufferedReader(isr);


            Scanner scanner = new Scanner(System.in);
            while(true) {
                String line = scanner.nextLine();
                if("exit".equalsIgnoreCase(line)){
                    break;
                }
                pw.println(line);

                line = br.readLine();
                System.out.println(line);
            }

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                /*
                    通讯完毕后调用socket的close方法。
                    该方法会给对方发送断开信号。
                 */
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        Client client = new Client();
        client.start();
```

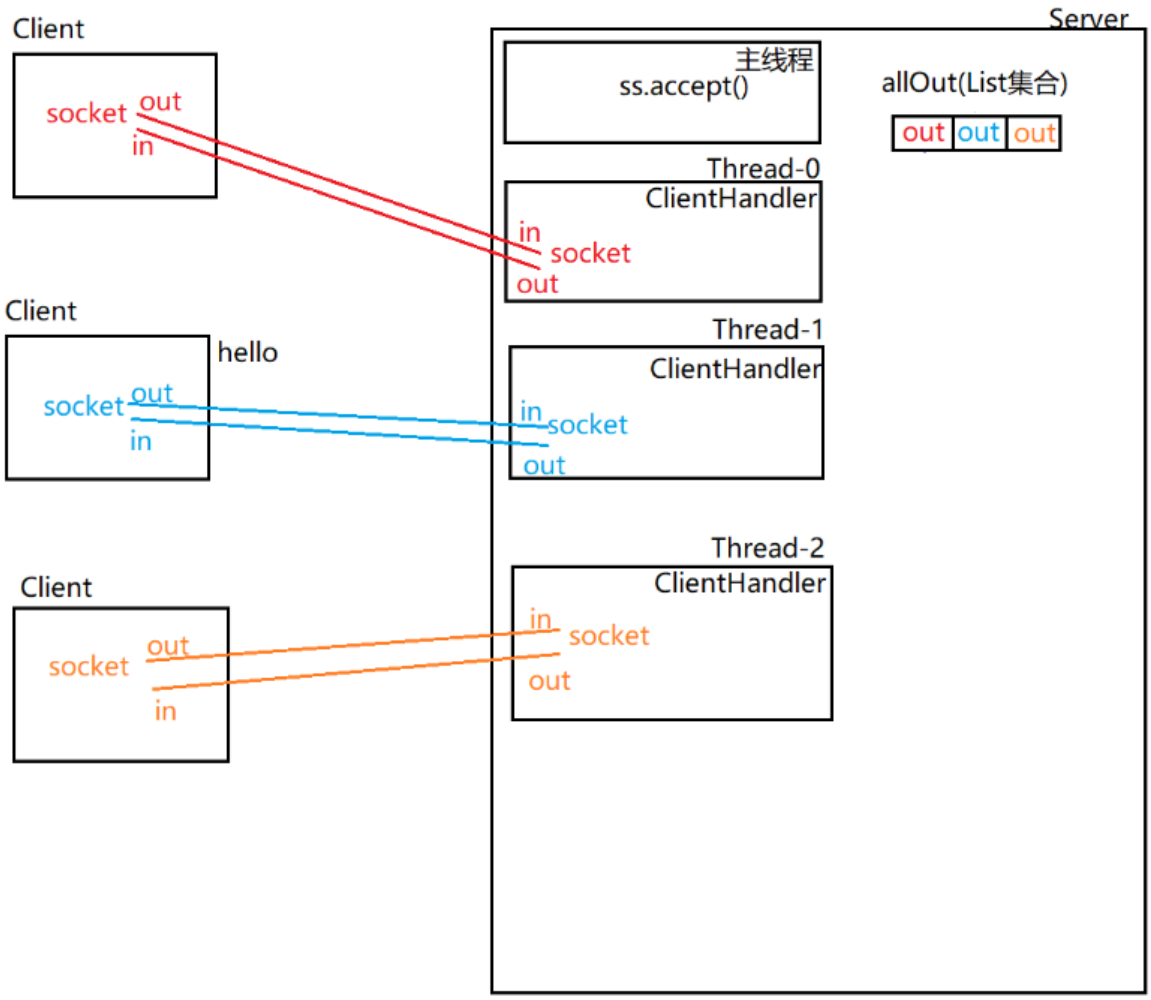
```
    }
}
```

服务端转发消息给所有客户端

当一个客户端发送一个消息后,服务端收到后如何转发给所有客户端.

问题:例如红色的线程一收到客户端消息后如何获取到橙色的线程二中的输出流?得不到就无法将消息转发给橙色的客户端(进一步延伸就是无法转发给所有其他客户端)

解决:内部类可以访问外部类的成员,因此在Server类上定义一个集合allOut可以被所有内部类ClientHandler实例访问.从而将这些ClientHandler实例之间想互访的数据存放在这个集合中达到共享数据的目的.对此只需要将所有ClientHandler中的输出流都存入到集合allOut中就可以达到互访输出流转发消息的目的了.



服务端代码:

```
package socket;

import java.io.*;
import java.net.ServerSocket;
import java.nio.charset.StandardCharsets;
import java.net.Socket;
import java.util.List;
import java.util.ArrayList;

/**
 * 聊天室服务端
```

```
 */
public class Server {
    /**
     * 运行在服务端的ServerSocket主要完成两个工作:
     * 1:向服务端操作系统申请服务端口，客户端就是通过这个端口与ServerSocket建立链接
     * 2:监听端口，一旦一个客户端建立链接，会立即返回一个Socket。通过这个Socket
     *   就可以和该客户端交互了
     *
     * 我们可以把ServerSocket想象成某客服的"总机"。用户打电话到总机，总机分配一个
     * 电话使得服务端与你沟通。
     */
    private ServerSocket serverSocket;
    /*
        存放所有客户端输出流，用于广播消息
     */
    private List<PrintWriter> allOut = new ArrayList();

    /**
     * 服务端构造方法，用来初始化
     */
    public Server(){
        try {
            System.out.println("正在启动服务端...");
            /*
                实例化ServerSocket时要指定服务端口，该端口不能与操作系统其他
                应用程序占用的端口相同，否则会抛出异常:
                java.net.BindException:address already in use

                端口是一个数字，取值范围:0-65535之间。
                6000之前的的端口不要使用，密集绑定系统应用和流行应用程序。
             */
            serverSocket = new ServerSocket(8088);
            System.out.println("服务端启动完毕!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 服务端开始工作的方法
     */
    public void start(){
        try {
            while(true) {
                System.out.println("等待客户端链接...");
                /*
                    ServerSocket提供了接受客户端链接的方法:
                    Socket accept()
                    这个方法是一个阻塞方法，调用后方法"卡住"，此时开始等待客户端
                    的链接，直到一个客户端链接，此时该方法会立即返回一个Socket实例
                    通过这个Socket就可以与客户端进行交互了。

                    可以理解为此操作是接电话，电话没响时就一直等。
                 */
                Socket socket = serverSocket.accept();
```

```
                System.out.println("一个客户端链接了！");
                //启动一个线程与该客户端交互
                ClientHandler clientHandler = new ClientHandler(socket);
                Thread t = new Thread(clientHandler);
                t.start();

            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }

    /**
     * 定义线程任务
     * 目的是让一个线程完成与特定客户端的交互工作
     */
    private class ClientHandler implements Runnable{
        private Socket socket;
        private String host;//记录客户端的IP地址信息

        public ClientHandler(Socket socket){
            this.socket = socket;
            //通过socket获取远端计算机地址信息
            host = socket.getInetAddress().getHostAddress();
        }
        public void run(){
            try{
                 /*
                    Socket提供的方法:
                    InputStream getInputStream()
                    获取的字节输入流读取的是对方计算机发送过来的字节
                 */
                InputStream in = socket.getInputStream();
                InputStreamReader isr = new InputStreamReader(in,
StandardCharsets.UTF_8);
                BufferedReader br = new BufferedReader(isr);

                OutputStream out = socket.getOutputStream();
                OutputStreamWriter osw = new
OutputStreamWriter(out,StandardCharsets.UTF_8);
                BufferedWriter bw = new BufferedWriter(osw);
                PrintWriter pw = new PrintWriter(bw,true);

                //将该输出流存入allOut中
                //1对allOut数组扩容
                allOut.add(pw);


                String message = null;
                while ((message = br.readLine()) != null) {
```

```
                    System.out.println(host + "说:" + message);
                    //将消息回复给所有客户端
                    for(PrintWriter o : allOut) {
                        o.println(host + "说:" + message);
                    }
                }
            }catch(IOException e){
                e.printStackTrace();
            }
        }
    }


}
```

客户端解决收发消息的冲突问题

由于客户端start方法中循环进行的操作顺序是先通过控制台输入一句话后将其发送给服务端,然后再读取服务端发送回来的一句话.这导致如果客户端不输入内容就无法收到服务端发送过来的其他信息(其他客户端的聊天内容).因此要将客户端中接收消息的工作移动到一个单独的线程上执行,才能保证收发消息互不打扰.

客户端代码:

```
package socket;

import java.io.*;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

/**
 * 聊天室客户端
 */
public class Client {
    /*
        java.net.Socket 套接字
        Socket封装了TCP协议的通讯细节，我们通过它可以与远端计算机建立链接，
        并通过它获取两个流(一个输入，一个输出)，然后对两个流的数据读写完成
        与远端计算机的数据交互工作。
        我们可以把Socket想象成是一个电话，电话有一个听筒(输入流)，一个麦克
        风(输出流)，通过它们就可以与对方交流了。
     */
    private Socket socket;

    /**
     * 构造方法，用来初始化客户端
     */
    public Client(){
        try {
            System.out.println("正在链接服务端...");
            /*
                实例化Socket时要传入两个参数
                参数1:服务端的地址信息
                      可以是IP地址，如果链接本机可以写"localhost"
```

```
                参数2:服务端开启的服务端口
                我们通过IP找到网络上的服务端计算机，通过端口链接运行在该机器上
                的服务端应用程序。
                实例化的过程就是链接的过程，如果链接失败会抛出异常:
                java.net.ConnectException: Connection refused: connect
             */
            socket = new Socket("localhost",8088);
            System.out.println("与服务端建立链接!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 客户端开始工作的方法
     */
    public void start(){
        try {
            //启动读取服务端发送过来消息的线程
            ServerHandler handler = new ServerHandler();
            Thread t = new Thread(handler);
            t.setDaemon(true);
            t.start();


            /*
                Socket提供了一个方法:
                OutputStream getOutputStream()
                该方法获取的字节输出流写出的字节会通过网络发送给对方计算机。
             */
            //低级流，将字节通过网络发送给对方
            OutputStream out = socket.getOutputStream();
            //高级流，负责衔接字节流与字符流，并将写出的字符按指定字符集转字节
            OutputStreamWriter osw = new
OutputStreamWriter(out,StandardCharsets.UTF_8);
            //高级流，负责块写文本数据加速
            BufferedWriter bw = new BufferedWriter(osw);
            //高级流，负责按行写出字符串，自动行刷新
            PrintWriter pw = new PrintWriter(bw,true);


            Scanner scanner = new Scanner(System.in);
            while(true) {
                String line = scanner.nextLine();
                if("exit".equalsIgnoreCase(line)){
                    break;
                }
                pw.println(line);
            }

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                /*
```

```
                    通讯完毕后调用socket的close方法。
                    该方法会给对方发送断开信号。
                 */
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        Client client = new Client();
        client.start();
    }

    /**
     * 该线程负责接收服务端发送过来的消息
     */
    private class ServerHandler implements Runnable{
        public void run(){
            //通过socket获取输入流读取服务端发送过来的消息
            try {
                InputStream in = socket.getInputStream();
                InputStreamReader isr = new
InputStreamReader(in,StandardCharsets.UTF_8);
                BufferedReader br = new BufferedReader(isr);

                String line;
                //循环读取服务端发送过来的每一行字符串
                while((line = br.readLine())!=null){
                    System.out.println(line);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

服务端解决多线程并发安全问题

为了让能叫消息转发给所有客户端，我们 在Server上添加了一个集合类型的属性allOut,并且共所有线程ClientHandler使用，这时对集合的操作要考虑并发安全问题，还要考虑对集合的不同操作之间的互斥问题。因此，对allOut集合的添加元素，删除元素和遍历操作要进行互斥。

最终代码:

```
package socket;

import java.io.*;
import java.net.ServerSocket;
import java.nio.charset.StandardCharsets;
import java.net.Socket;

/**
```

```
 * 聊天室服务端
 */
public class Server {
    /**
     * 运行在服务端的ServerSocket主要完成两个工作:
     * 1:向服务端操作系统申请服务端口，客户端就是通过这个端口与ServerSocket建立链接
     * 2:监听端口，一旦一个客户端建立链接，会立即返回一个Socket。通过这个Socket
     *   就可以和该客户端交互了
     *
     * 我们可以把ServerSocket想象成某客服的"总机"。用户打电话到总机，总机分配一个
     * 电话使得服务端与你沟通。
     */
    private ServerSocket serverSocket;
    /*
        存放所有客户端输出流，用于广播消息
     */
    private PrintWriter[] allOut = {};

    /**
     * 服务端构造方法，用来初始化
     */
    public Server(){
        try {
            System.out.println("正在启动服务端...");
            /*
                实例化ServerSocket时要指定服务端口，该端口不能与操作系统其他
                应用程序占用的端口相同，否则会抛出异常:
                java.net.BindException:address already in use

                端口是一个数字，取值范围:0-65535之间。
                6000之前的的端口不要使用，密集绑定系统应用和流行应用程序。
             */
            serverSocket = new ServerSocket(8088);
            System.out.println("服务端启动完毕!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 服务端开始工作的方法
     */
    public void start(){
        try {
            while(true) {
                System.out.println("等待客户端链接...");
                /*
                    ServerSocket提供了接受客户端链接的方法:
                    Socket accept()
                    这个方法是一个阻塞方法，调用后方法"卡住"，此时开始等待客户端
                    的链接，直到一个客户端链接，此时该方法会立即返回一个Socket实例
                    通过这个Socket就可以与客户端进行交互了。

                    可以理解为此操作是接电话，电话没响时就一直等。
                 */
```

```
                Socket socket = serverSocket.accept();
                System.out.println("一个客户端链接了！");
                //启动一个线程与该客户端交互
                ClientHandler clientHandler = new ClientHandler(socket);
                Thread t = new Thread(clientHandler);
                t.start();

            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }

    /**
     * 定义线程任务
     * 目的是让一个线程完成与特定客户端的交互工作
     */
    private class ClientHandler implements Runnable{
        private Socket socket;
        private String host;//记录客户端的IP地址信息

        public ClientHandler(Socket socket){
            this.socket = socket;
            //通过socket获取远端计算机地址信息
            host = socket.getInetAddress().getHostAddress();
        }
        public void run(){
            PrintWriter pw = null;
            try{
                 /*
                    Socket提供的方法:
                    InputStream getInputStream()
                    获取的字节输入流读取的是对方计算机发送过来的字节
                 */
                InputStream in = socket.getInputStream();
                InputStreamReader isr = new InputStreamReader(in,
StandardCharsets.UTF_8);
                BufferedReader br = new BufferedReader(isr);

                OutputStream out = socket.getOutputStream();
                OutputStreamWriter osw = new
OutputStreamWriter(out,StandardCharsets.UTF_8);
                BufferedWriter bw = new BufferedWriter(osw);
                pw = new PrintWriter(bw,true);

                //将该输出流存入allOut中
                synchronized (allOut) {
                   allOut.add(pw);
                }
                //通知所有客户端该用户上线了
```

```
                System.out.println(host + "上线了,当前在线人数:"+allOut.length);


                String message = null;
                while ((message = br.readLine()) != null) {
                    System.out.println(host + "说:" + message);
                    //将消息回复给所有客户端
                    synchronized (allOut) {
                    for (PrintWriter o : allOut) {
                        allOut[i].println(host + "说:" + message);
                    }
            }
                }
            }catch(IOException e){
                e.printStackTrace();
            }finally{
                //处理客户端断开链接的操作
                //将当前客户端的输出流从allOut中删除
                synchronized (allOut) {
                    allOut.remove(pw);
                }
                System.out.println(host+"下线了，当前在线人数:"+allOut.length);
                try {
                    socket.close();//与客户端断开链接
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```