

day09

实现客户端循环发消息给服务端

客户端代码:

```
package socket;

import java.io.*;
import java.net.Socket;
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,"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();
}

```

```
}  
}
```

服务端代码:

```
package socket;  
  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.InputStreamReader;  
import java.net.ServerSocket;  
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;

/**
 * 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 {
        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
         的连接，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！");

        /**
         * 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, "UTF-8");
    BufferedReader br = new
BufferedReader(isr);

    String message = null;
    while((message =
br.readLine())!=null) {
        System.out.println("客户端
说:" + message);
    }
} catch (IOException e) {
    e.printStackTrace();
}
}

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


需要注意的几个点:

1:当客户端不再与服务端通讯时, 需要调用 `socket.close()` 断开链接, 此时会发送断开链接的信号给服务端。这时服务端的 `br.readLine()` 方法会返回 `null`, 表示客户端断开了链接。

2:当客户端链接后不输入信息发送给服务端时, 服务端的 `br.readLine()` 方法是出于阻塞状态的, 直到读取了一行来自客户端发送的字符串。

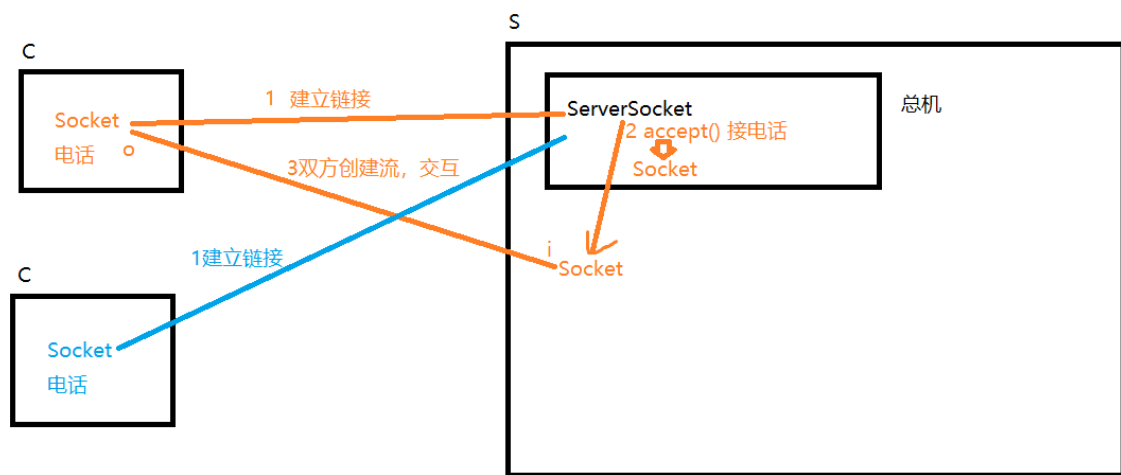
多客户端链接

之前只有第一个连接的客户端可以与服务端说话。

原因:

服务端只调用过一次 `accept` 方法, 因此只有第一个客户端链接时服务端接受了链接并返回了 `Socket`, 此时可以与其交互。

而第二个客户端建立链接时, 由于服务端没有再次调用 `accept`, 因此无法与其交互。



```
package socket;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
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;
```

```
/**
```

* 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  
                的连接，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！");
        /*
        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, "UTF-8");
        BufferedReader br = new
BufferedReader(isr);

        String message = null;
        while ((message =
br.readLine()) != null) {
            System.out.println("客户
端说：" + message);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

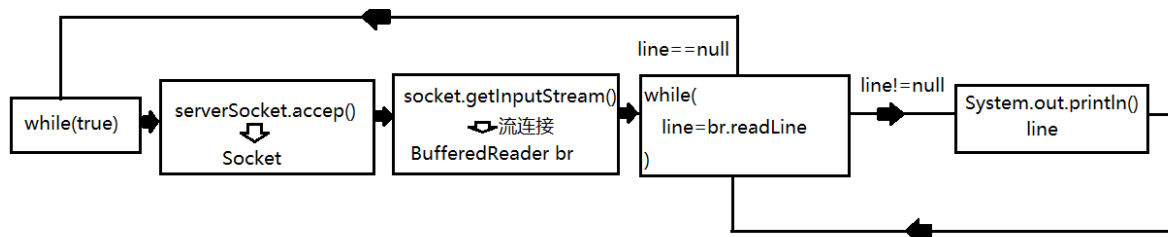
    public static void main(String[] args)
    {
        Server server = new Server();
        server.start();
    }
}

```

添加循环操作后，发现依然无法实现。

原因在于：

外层的while循环里面嵌套了一个内层循环(循环读取客户端发送消息)，而循环执行机制决定了里层循环不结束，外层循环则无法进入第二次操作。



多线程

线程:一个顺序的单一的程序执行流程就是一个线程。代码一句一句的有先后顺序的执行。

多线程:多个单一顺序执行的流程并发运行。造成"感官上同时运行"的效果。

并发:

多个线程实际运行是走走停停的。线程调度程序会将CPU运行时间划分为若干个时间片段并

尽可能均匀的分配给每个线程，拿到时间片的线程被CPU执行这段时间。当超时后线程调度

程序会再次分配一个时间片段给一个线程使得CPU执行它。如此反复。由于CPU执行时间在

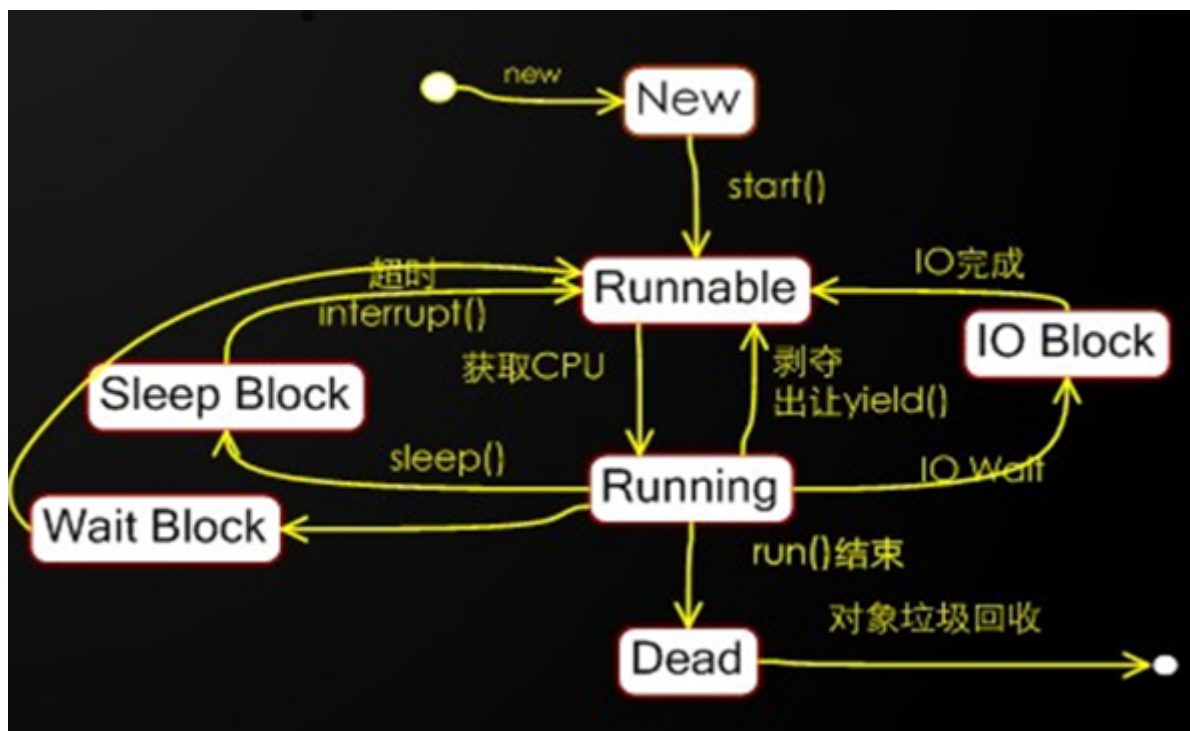
纳秒级别，我们感觉不到切换线程运行的过程。所以微观上走走停停，宏观上感觉一起运行

的现象成为并发运行!

用途:

- 当出现多个代码片段执行顺序有冲突时，希望它们各干各的时就应当放在不同线程上"同时"运行
- 一个线程可以运行，但是多个线程可以更快时，可以使用多线程运行

线程的生命周期图



创建线程有两种方式

方式一:继承Thread并重写run方法

定义一个线程类，重写run方法，在其中定义线程要执行的任务(希望和其他线程并发执行的任务)。

注:启动该线程要调用该线程的start方法，而不是run方法!!!

```
package thread;
```

```
/**
```

```
 * 多线程
```

```
 * 线程: 程序中一个单一的顺序执行流程
```

```
 * 多线程: 多个单一顺序执行流程"同时"执行
```

```
 *
```

```
 * 多线程改变了代码的执行方式，从原来的单一顺序执行流程变为多个执行流程"同时"执行。
```


- * 可以让多个代码片段的执行互不打扰。
- *
- * 线程之间是并发执行的，并非真正意义上的同时运行。
- * 常见线程有两种方式：
- * 1: 继承**Thread**并重写**run**方法
- *
- */

```
public class ThreadDemo1 {
    public static void main(String[] args)
{
```

```
    //创建两个线程
```

```
    Thread t1 = new MyThread1();
```

```
    Thread t2 = new MyThread2();
```

```
    /*
```

启动线程,注意:不要调用**run**方法!!

线程调用完**start**方法后会纳入到系统的线程调度器程序中被统一管理。

线程调度器会分配时间片段给线程，使得CPU执行该线程这段时间，用完后

线程调度器会再分配一个时间片段给一个线程，如此反复，使得多个线程

都有机会执行一会，做到走走停停，并发运行。

线程第一次被分配到时间后会执行它的**run**方法开始工作。

```
    */
```

```
    t1.start();
```

```
    t2.start();
```

```
}
```

```

}
/**
 * 第一种创建线程的优点：
 * 结构简单，利于匿名内部类形式创建。
 *
 * 缺点：
 * 1: 由于java是单继承的，这会导致继承了Thread就无法再继承其他类去复用方法
 * 2: 定义线程的同时重写了run方法，这等于将线程的任务定义在了这个线程中导致
 *     线程只能干这件事。重(chong)用性很低。
 */
class MyThread1 extends Thread{
    public void run(){
        for (int i=0;i<1000;i++){
            System.out.println("hello姐~");
        }
    }
}

class MyThread2 extends Thread{
    public void run(){
        for (int i=0;i<1000;i++){
            System.out.println("来了~老弟!");
        }
    }
}

```

第一种创建线程的方式

优点:

在于结构简单，便于匿名内部类形式创建。

缺点:

- 1:直接继承线程，会导致不能在继承其他类去复用方法，这在实际开发中是非常不便的。
- 2:定义线程的同时重写了run方法，会导致线程与线程任务绑定在了一起，不利于线程的重用。

方式二:实现Runnable接口单独定义线程任务

```
package thread;

/**
 * 第二种创建线程的方式
 * 实现Runnable接口单独定义线程任务
 */
public class ThreadDemo2 {
    public static void main(String[] args)
    {
        //实例化任务
        Runnable r1 = new MyRunnable1();
        Runnable r2 = new MyRunnable2();
        //创建线程并指派任务
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        t1.start();
        t2.start();
    }
}
```

```

class MyRunnable1 implements Runnable{
    public void run() {
        for (int i=0;i<1000;i++){
            System.out.println("你是谁啊?");
        }
    }
}

class MyRunnable2 implements Runnable{
    public void run() {
        for (int i=0;i<1000;i++){
            System.out.println("开门!查水表的!");
        }
    }
}

```

匿名内部类形式的线程创建

```

package thread;

/**
 * 使用匿名内部类完成线程的两种创建
 */
public class ThreadDemo3 {
    public static void main(String[] args)
    {
        Thread t1 = new Thread(){
            public void run(){
                for(int i=0;i<1000;i++){

```

```

        System.out.println("你是
谁啊?");
    }
}
};

//        Runnable r2 = new Runnable() {
//            public void run() {
//                for(int i=0;i<1000;i++){
//                    System.out.println("我
是查水表的!");
//                }
//            }
//        };
//Runnable可以使用lambda表达式创建
Runnable r2 = ()->{
    for(int i=0;i<1000;i++){
        System.out.println("我是
查水表的!");
    }
};

Thread t2 = new Thread(r2);

t1.start();
t2.start();
}
}

```

java中的代码都是靠线程运行的，执行main方法的线程称为"主线程"。

线程提供了一个方法:

- static Thread currentThread()

该方法可以获取运行这个方法的线程

```
package thread;
```

```
/**
```

```
 * java中所有的代码都是靠线程执行的，main方法也不例外。JVM启动后会创建一条线程来执行main
```

```
 * 方法，该线程的名字叫做"main"，所以通常称它为"主线程"。
```

```
 * 我们自己定义的线程在不指定名字的情况下系统会分配一个名字，格式为"thread-x"(x是一个数)。
```

```
 *
```

```
 * Thread提供了一个静态方法：
```

```
 * static Thread currentThread()
```

```
 * 获取执行该方法的线程。
```

```
 *
```

```
 */
```

```
public class CurrentThreadDemo {
```

```
    public static void main(String[] args)
{
```

```
    /*
```

后期会学习到一个很重要的

API:ThreadLocal,它可以使得我们在一个线程上跨越多个

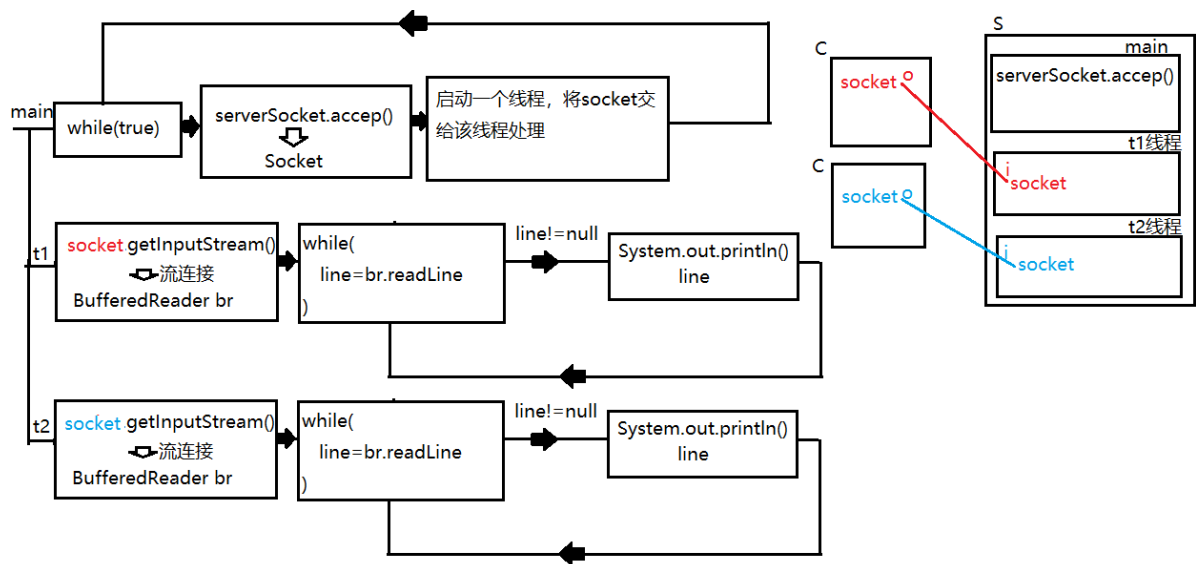
方法时共享数据使用，其内部要用到
`currentThread`方法来辨别线程。

如spring的事物控制就是靠
`ThreadLocal`实现的。

```
        */  
        Thread main =  
Thread.currentThread(); //获取执行main方法的线程(主线程)  
        System.out.println("线程:"+main);  
  
        dosome(); //主线程执行dosome方法  
    }  
    public static void dosome(){  
        Thread t =  
Thread.currentThread(); //获取执行dosome方法的线程  
        System.out.println("执行dosome方法的线程是:"+t);  
    }  
}
```

使用多线程实现多客户端连接服务端

流程图



服务端代码改造:

```
package socket;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
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;
```

```
/**
```

* 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
的连接，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;
    public ClientHandler(Socket socket)
{
        this.socket = socket;
    }
    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, "UTF-8");
            BufferedReader br = new
BufferedReader(isr);

            String message = null;
            while ((message =
br.readLine()) != null) {
                System.out.println("客户
端说：" + message);
            }
        }catch(IOException e){

```

```

        e.printStackTrace();
    }
}
}
}
}

```

线程API

获取线程相关信息的方法

```

package thread;

/**
 * 获取线程相关信息的一组方法
 */
public class ThreadInfoDemo {
    public static void main(String[] args)
    {
        Thread main =
Thread.currentThread();//获取主线程

        String name = main.getName();//获取
线程的名字
        System.out.println("名字:"+name);
    }
}

```

```
        long id = main.getId();//获取该线程的  
        唯一标识  
        System.out.println("id:"+id);  
  
        int priority =  
main.getPriority();//获取该线程的优先级  
        System.out.println("优先  
级:"+priority);  
  
        boolean isAlive = main.isAlive();//  
该线程是否活着  
        System.out.println("是否活  
着:"+isAlive);  
  
        boolean isDaemon =  
main.isDaemon();//是否为守护线程  
        System.out.println("是否为守护线  
程:"+isDaemon);  
  
        boolean isInterrupted =  
main.isInterrupted();//是否被中断了  
        System.out.println("是否被中断  
了:"+isInterrupted);  
  
    }  
}
```

线程优先级

线程start后会纳入到线程调度器中统一管理,线程只能被动的被分配时间片并发运行,而无法主动索取时间片.线程调度器尽可能均匀的将时间片分配给每个线程.

线程有10个优先级,使用整数1-10表示

- 1为最小优先级,10为最高优先级.5为默认值
- 调整线程的优先级可以最大程度的干涉获取时间片的几率.优先级越高的线程获取时间片的次数越多,反之则越少.

```
package thread;

public class PriorityDemo {
    public static void main(String[] args)
    {
        Thread max = new Thread(){
            public void run(){
                for(int i=0;i<10000;i++){

System.out.println("max");
                }
            }
        };
        Thread min = new Thread(){
            public void run(){
                for(int i=0;i<10000;i++){

System.out.println("min");
                }
            }
        };
    }
}
```

```

        }
    }
};

Thread norm = new Thread(){
    public void run(){
        for(int i=0;i<10000;i++){

System.out.println("nor");
        }
    }
};

min.setPriority(Thread.MIN_PRIORITY);

max.setPriority(Thread.MAX_PRIORITY);
    min.start();
    norm.start();
    max.start();
}
}

```

sleep阻塞

线程提供了一个静态方法:

- static void sleep(long ms)
- 使运行该方法的线程进入阻塞状态指定的毫秒,超时后线程会自动回到RUNNABLE状态等待再次获取时间片并发运行.


```
package thread;

public class SleepDemo {
    public static void main(String[] args)
    {
        System.out.println("程序开始了!");
        try {
            Thread.sleep(5000); //主线程阻塞5
秒钟
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("程序结束了!");
    }
}
```