网络编程

主要内容

网络编程基本概念

Java 网络编程中的常用类

TCP 通信实现

UDP 通信实现

章节总结

学习目标

知识点	要求
网络编程基本概念	了解
Java 网络编程中的常用类	掌握
TCP 通信实现	掌握
UDP 通信实现	掌握
章节总结	掌握

一、 网络编程基本概念

1 计算机网络

计算机网络是指将地理位置不同的具有独立功能的多台计算机及其外部设备，通过通信线路连接起来，在网络操作系统，网络管理软件及网络通信协议的管理和协调下，实现资源共享和信息传递的计算机系统。

从其中我们可以提取到以下内容：

1. 计算机网络的作用：资源共享和信息传递。
2. 计算机网络的组成：

a) 计算机硬件：计算机（大中小型服务器，台式机、笔记本等）、外部设备（路由器、交换机等）、通信线路（双绞线、光纤等）。

b) 计算机软件 :网络操作系统(Windows 2000 Server/Advance Server、Unix、Linux 等）、网络管理软件（WorkWin、SugarNMS 等）、网络通信协议（如 TCP/IP 协议栈等）。

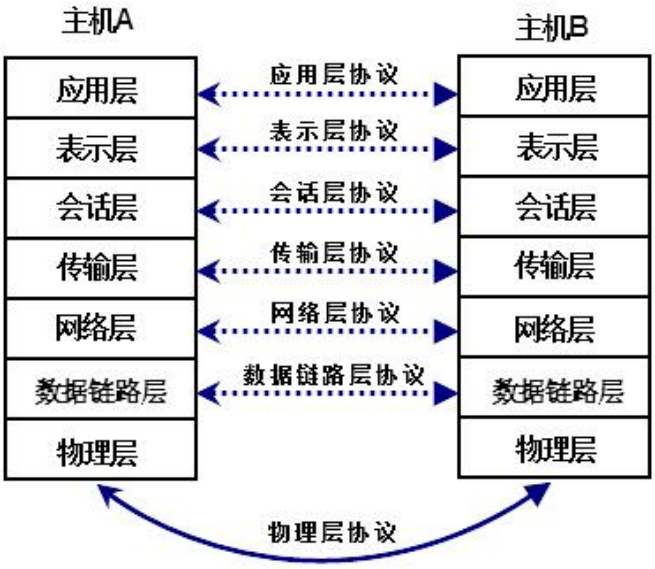
2 网络通信协议

❑ 网络通信协议

通过计算机网络可以实现不同计算机之间的连接与通信 ,但是计算机网络中实现通信必须有一些约定即通信协议，对速率、传输代码、代码结构、传输控制步骤、出错控制等制定标准。就像两个人想要顺利沟通就必须使用同一种语言一样，如果一个人只懂英语而另外一个人只懂中文，这样就会造成没有共同语言而无法沟通。

国际标准化组织(ISO，即 International Organization for Standardization)定义了网络通信协议的基本框架，被称为 OSI（Open System Interconnect，即开放系统互联）模型。要制定通讯规则，内容会很多，比如要考虑 A 电脑如何找到 B 电脑，A 电脑在发送信息给 B 电脑时是否需要 B 电脑进行反馈，A 电脑传送给 B 电脑的数据格式又是怎样的？内容太多太杂，所以 OSI 模型将这些通讯标准进行层次划分，每一层次解决一个类别的问题，这样就使得标准的制定没那么复杂。OSI 模型制定的七层标准模型，分别是：应用层，表示层，会话层，传输层，网络层，数据链路层，物理层。

OSI 七层协议模型：



虽然国际标准化组织制定了这样一个网络通信协议的模型，但是实际上互联网通讯使用最多的网络通信协议是 TCP/IP 网络通信协议。

TCP/IP 是一个协议族，也是按照层次划分，共四层：应用层，传输层，互连网络层，网络接口层（物理+数据链路层）。

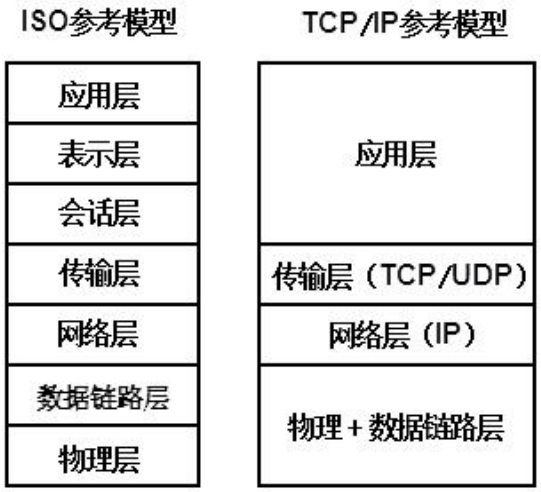
那么 TCP/IP 协议和 OSI 模型有什么区别呢？OSI 网络通信协议模型，是一个参考模型，而 TCP/IP 协议是事实上的标准。TCP/IP 协议参考了 OSI 模型，但是并没有严格按照 OSI 规定的七层标准去划分，而只划分了四层，这样会更简单点，当划分太多层次时，你很难区分某个协议是属于哪个层次的。TCP/IP 协议和 OSI 模型也并不冲突，TCP/IP 协议中的应用层协议，就对应于 OSI 中的应用层，表示层，会话层。就像以前有工业部和信息产业部，现在实行大部制后只有工业和信息化部一个部门，但是这个部门还是要做以前两个部门一样多的事情，本质上没有多大的差别。TCP/IP 中有两个重要的协议，传输层的 TCP 协议和互连网络层的 IP 协议，因此就拿这两个协议做代表，来命名整个协议族了，再说 TCP/IP 协议时，是指整个协议族。

- **网络协议的分层**

由于网络结点之间联系很复杂，在制定协议时，把复杂成份分解成一些简单的成份，再将它们复合起来。最常用的复合方式是层次方式，即同层间可以通信、上一层可以调用下一层，而与再下一层不发生关系。

把用户应用程序作为最高层，把物理通信线路作为最低层，将其间的协议处理分为若干层，规定每层处理的任务，也规定每层的接口标准。

ISO 模型与 TCP/IP 模型的对应关系。



3 数据封装与解封

由于用户传输的数据一般都比较大，有的可以达到 MB 字节，一次性发送出去十分困难，于是就需要把数据分成许多片段，再按照一定的次序发送出去。这个过程就需要对数据进行封装。

数据封装（Data Encapsulation）是指将协议数据单元（PDU）封装在一组协议头和协议尾中的过程。在 OSI 七层参考模型中，每层主要负责与其它机器上的对等层进行通信。该过程是在协议数据单元（PDU）中实现的，其中每层的 PDU 一般由本层的协议头、协议尾和数据封装构成。

1.数据发送处理过程

（1）应用层将数据交给传输层，传输层添加上 TCP 的控制信息（称为 TCP 头部），这个数据单元称为段（Segment），加入控制信息的过程称为封装。然后，将段交给网络层。

（2）网络层接收到段，再添加上 IP 头部，这个数据单元称为包（Packet）。然后，将包交给数据链路层。

（3） 数据链路层接收到包，再添加上 MAC 头部和尾部，这个数据单元称为帧（Frame）。然后，将帧交给物理层。

（4）物理层将接收到的数据转化为比特流，然后在网线中传送。

2.数据接收处理过程

（1）物理层接收到比特流，经过处理后将数据交给数据链路层。

（2）数据链路层将接收到的数据转化为数据帧，再除去 MAC 头部和尾部，这个除去控制信息的过程称为解封，然后将包交给网络层。

（3）网络层接收到包，再除去 IP 头部，然后将段交给传输层。

（4）传输层接收到段，再除去 TCP 头部，然后将数据交给应用层。

从以上传输过程中，可以总结出以下规则：

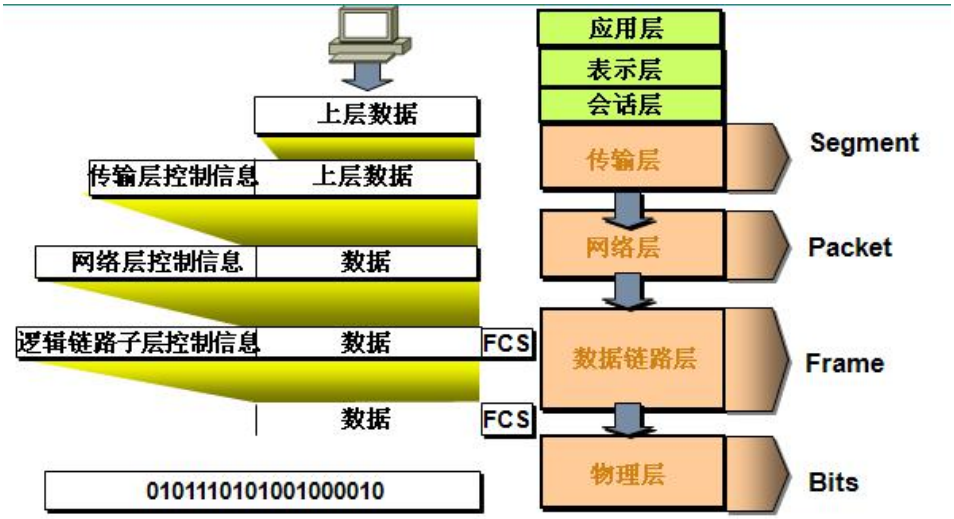
（1）发送方数据处理的方式是从高层到底层，逐层进行数据封装。

（2）接收方数据处理的方式是从底层到高层，逐层进行数据解封装。

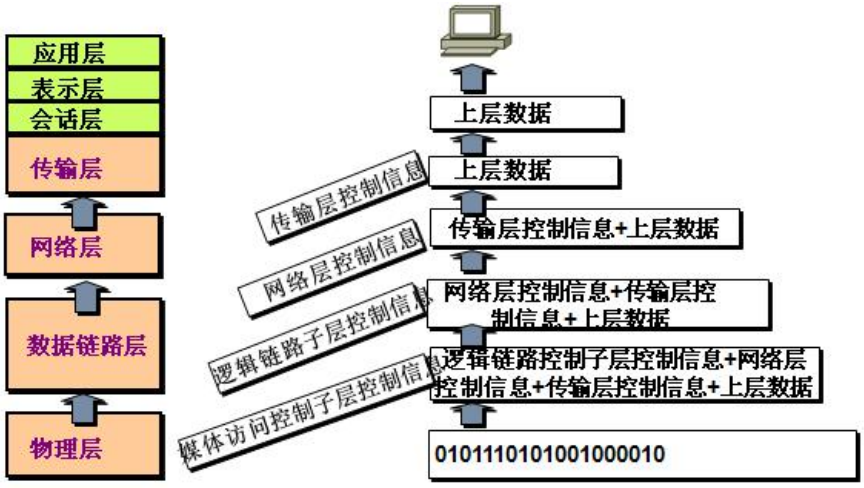
接收方的每一层只把对该层有意义的数据拿走，或者说每一层只能处理发送方同等层的数据，然后把其余的部分传递给上一层，这就是对等层通信的概念。

数据封装与解封：

数据封装



数据解封



4 IP 地址与端口

❑ IP 地址

用来标识网络中的一个通信实体的地址。通信实体可以是计算机、路由器等。 比如互联网的每个服务器都要有自己的 IP 地址，而每个局域网的计算机要通信也要配置 IP 地址。

路由器是连接两个或多个网络的网络设备。

目前主流使用的 IP 地址是 IPV4，但是随着网络规模的不断扩大，IPV4 面临着枯竭的危险，所以推出了 IPV6。

IPV4：32 位地址，并以 8 位为一个单位，分成四部分，以点分十进制表示，如 192.168.0.1。

IPV6：128 位（16 个字节）写成 8 个 16 位的无符号整数，每个整数用四个十六进制位表示，每个数之间用冒号（：）分开，如：3ffe:3201:1401:1280:c8ff:fe4d:db39:1984

注意事项

- 127.0.0.1 本机地址
- 192.168.0.0--192.168.255.255 为私有地址，属于非注册地址，专门为组织机构内部使用。

- **端口**

IP 地址用来标识一台计算机，但是一台计算机上可能提供多种网络应用程序，如何来区分这些不同的程序呢？这就要用到端口。

端口是虚拟的概念，并不是说在主机上真的有若干个端口。通过端口，可以在一个主机上运行多个网络应用程序。端口的表示是一个16位的二进制整数，对应十进制的0-65535。

Oracle、MySQL、Tomcat、QQ、msn、迅雷、电驴、360 等网络程序都有自己的端口。

总结

- IP 地址好比每个人的地址（门牌号），端口好比是房间号。必须同时指定 IP 地址和端口号才能够正确的发送数据。

5 URL

在 www 上，每一信息资源都有统一且唯一的地址，该地址就叫 URL（Uniform Resource Locator），它是 www 的统一资源定位符。URL 由 4 部分组成：协议 、存放资源的主机域名、资源文件名和端口号。如果未指定该端口号，则使用协议默认的端口。例如 http 协议的默认端口为 80。 在浏览器中访问网页时，地址栏显示的地址就是 URL。

在 java.net 包中提供了 URL 类，该类封装了大量复杂的涉及从远程站点获取信息的细节。

6 Socket

我们开发的网络应用程序位于应用层，TCP 和 UDP 属于传输层协议，在应用层如何使用传输层的服务呢？在应用层和传输层之间，则是使用套接字 Socket 来进行分离。

套接字就像是传输层为应用层开的一个小口，应用程序通过这个小口向远程发送数据，或者接收远程发来的数据；而这个小口以内，也就是数据进入这个口之后，或者数据从这个口出来之前，是不知道也不需要知道的，也不会关心它如何传输，这属于网络其它层次工作。

Socket 实际是传输层供给应用层的编程接口。Socket 就是应用层与传输层之间的桥梁。使用 Socket 编程可以开发客户机和服务器应用程序，可以在本地网络上进行通信，也可通过 Internet 在全球范围内通信。

7 TCP 协议和 UDP 协议

❑ 联系和区别

TCP 协议和 UDP 协议是传输层的两种协议。Socket 是传输层供给应用层的编程接口，所以 Socket 编程就分为 TCP 编程和 UDP 编程两类。

在网络通讯中，TCP 方式就类似于拨打电话，使用该种方式进行网络通讯时，需要建立专门的虚拟连接，然后进行可靠的数据传输，如果数据发送失败，则客户端会自动重发该数据。而 UDP 方式就类似于发送短信，使用这种方式进行网络通讯时，不需要建立专门的虚拟连接，传输也不是很可靠，如果发送失败则客户端无法获得。

这两种传输方式都在实际的网络编程中使用，重要的数据一般使用 TCP 方式进行数据传输，而大量的非核心数据则可以通过 UDP 方式进行传递，在一些程序中甚至结合使用这两种方式进行数据传递。

由于 TCP 需要建立专用的虚拟连接以及确认传输是否正确，所以使用 TCP 方式的速度稍微慢一些，而且传输时产生的数据量要比 UDP 稍微大一些。

总结

- ❑ TCP 是面向连接的，传输数据安全，稳定，效率相对较低。
- ❑ UDP 是面向无连接的，传输数据不安全，效率较高。

❑ TCP 协议

TCP(Transfer Control Protocol)是面向连接的，所谓面向连接，就是当计算机双方通信时必需经过先建立连接，然后传送数据，最后拆除连接三个过程。

TCP 在建立连接时又分三步走：

- 第一步，是请求端（客户端）发送一个包含 SYN 即同步（Synchronize）标志的 TCP 报文，SYN 同步报文会指明客户端使用的端口以及 TCP 连接的初始序号。
- 第二步，服务器在收到客户端的 SYN 报文后，将返回一个 SYN+ACK 的报文，表示客户端的请求被接受，同时 TCP 序号被加一，ACK 即确认（Acknowledgement）。
- 第三步，客户端也返回一个确认报文 ACK 给服务器端，同样 TCP 序列号被加一，到此一个 TCP 连接完成。然后才开始通信的第二步：数据处理。
- 这就是所说的 TCP 的三次握手（Three-way Handshake）。

❑ **UDP 协议**

基于 TCP 协议可以建立稳定连接的点对点的通信。这种通信方式实时、快速、安全性高，但是很占用系统的资源。

在网络传输方式上，还有另一种基于 UDP 协议的通信方式，称为数据报通信方式。在这种方式中，每个数据发送单元被统一封装成数据报包的方式，发送方将数据报包发送到网络中，数据报包在网络中去寻找它的目的地。

二、 Java 网络编程中的常用类

Java 为了跨平台，在网络应用通信时是不允许直接调用操作系统接口的，而是由 java.net 包来提供网络功能。下面我们来介绍几个 java.net 包中的常用的类。

1 InetAddress 的使用

作用：封装计算机的 IP 地址和域名。

特点：这个类没有构造方法。如果要得到对象，只能通过 getLocalHost()、getByName()

等静态方法创建对象。

1.1获取本机信息

获取本机信息需要使用 getLocalHost 方法创建 InetAddress 对象。getLocalHost()方法返回一个 InetAddress 对象，这个对象包含了本机的 IP 地址，计算机名等信息。

```
public class InetTest {
    public static void main(String[] args)throws Exception {
        //实例化 InetAddress 对象
        InetAddress inetAddress = InetAddress.getLocalHost();
        //返回当前计算机的 IP 地址
        System.out.println(inetAddress.getHostAddress());
        //返回当前计算机名
        System.out.println(inetAddress.getHostName());
    }
}
```

1.2根据域名获取计算机的信息

根据域名获取计算机信息时需要使用 getByName(“域名”)方法创建 InetAddress 对象。

```
public class InetTest2 {
    public static void main(String[] args)throws Exception {
        InetAddress inetAddress =
InetAddress.getByName("www.baidu.com");
        System.out.println(inetAddress.getHostAddress());
        System.out.println(inetAddress.getHostName());
    }
}
```

1.3根据 IP 获取计算机的信息

根据 IP 地址获取计算机信息时需要使用 getByName(“IP”)方法创建 InetAddress

对象。

```
public class InetTest3 {
    public static void main(String[] args)throws  Exception {
        InetAddress inetAddress =
InetAddress.getByName("39.156.66.14");
        System.out.println(inetAddress.getHostAddress());
        System.out.println(inetAddress.getHostName());
    }
}
```

2 InetSocketAddress 的使用

作用:包含 IP 和端口信息,常用于 Socket 通信。此类实现 IP 套接字地址(IP 地址 + 端口号),不依赖任何协议。

InetSocketAddress 相比较 InetAddress 多了一个端口号，端口的作用：一台拥有 IP 地址的主机可以提供许多服务，比如 Web 服务、FTP 服务、SMTP 服务等，这些服务完全可以通过 1 个 IP 地址来实现。

那么，主机是怎样区分不同的网络服务呢？显然不能只靠 IP 地址，因为 IP 地址与网络服务的关系是一对多的关系。实际上是通过“IP 地址+端口号”来区分不同的服务的

```
public class InetSocketTest {
    public static void main(String[] args) {
        InetSocketAddress inetSocketAddress = new
InetSocketAddress("www.baidu.com",80);

System.out.println(inetSocketAddress.getAddress().getHostAddress(
));
        System.out.println(inetSocketAddress.getHostName());
    }
}
```

3 URL 类的使用

IP 地址标识了 Internet 上唯一的计算机,而 URL 则标识了这些计算机上的资源。 URL

代表一个统一资源定位符，它是指向互联网“资源”的指针。资源可以是简单的文件或目录，也可以是对更为复杂的对象的引用，例如对数据库或搜索引擎的查询。

为了方便程序员编程，JDK 中提供了 URL 类，该类的全名是 java.net.URL，有了这样一个类，就可以使用它的各种方法来对 URL 对象进行分割、合并等处理。

```
public class UrlTest {
    public static void main(String[] args)throws Exception {
        URL url = new
URL("http://www.baidu.com/s?ie=utf-8&f=3&rsv_bp=1&rsv_idx=1&tn=ba
idu&wd=itbz");

        System.out.println("获取与此 URL 相关联协议的默认端口：
"+url.getDefaultPort());

        System.out.println("访问资源："+url.getFile());

        System.out.println("主机名"+url.getHost());

        System.out.println("访问资源路径："+url.getPath());

        System.out.println("协议："+url.getProtocol());

        System.out.println("参数部分："+url.getQuery());
    }
}
```

三、 TCP 通信的实现

1 TCP 通信介绍

前边我们提到 TCP 协议是面向连接的协议，在通信时客户端与服务器端必须建立连接。在网络通信中，第一次主动发起通信的程序被称作客户端(Client)程序，简称客户端，而在第一次通信中等待连接的程序被称作服务器端(Server)程序，简称服务器。一旦通信建立，则客户端和服务器端完全一样，没有本质的区别。

- **“请求-响应”模式：**
  - Socket 类：发送 TCP 消息。
  - ServerSocket 类：创建服务器。

套接字 Socket 是一种进程间的数据交换机制。这些进程既可以在同一机器上，也可以在通过网络连接的不同机器上。换句话说，套接字起到通信端点的作用。

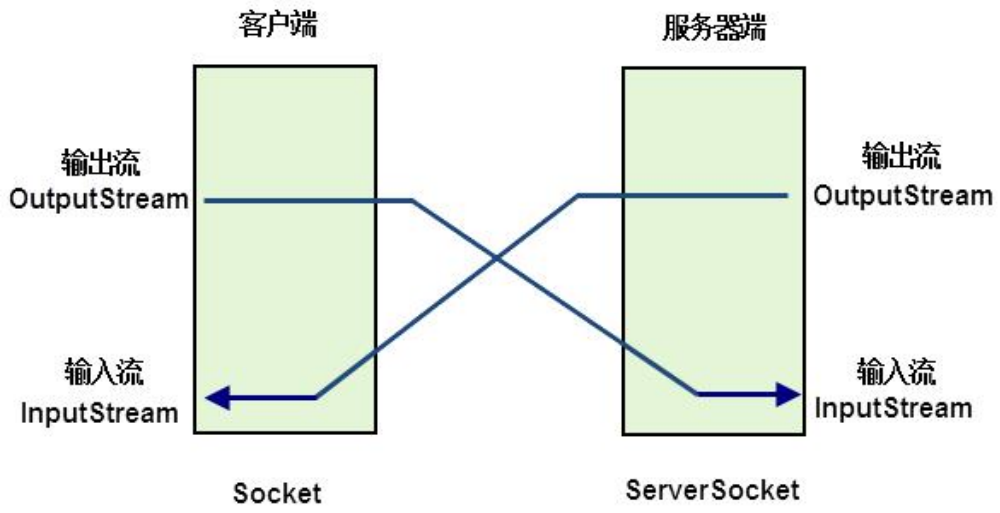
单个套接字是一个端点，而一对套接字则构成一个双向通信信道，使非关联进程可以在本地或通过网络进行数据交换。一旦建立套接字连接，数据即可在相同或不同的系统中双向或单向发送，直到其中一个端点关闭连接。套接字与主机地址和端口地址相关联。主机地址就是客户端或服务器程序所在的主机的 IP 地址。端口地址是指客户端或服务器程序使用的主机的通信端口。

在客户端和服务器中，分别创建独立的 Socket，并通过 Socket 的属性，将两个 Socket 进行连接，这样，客户端和服务器通过套接字所建立的连接使用输入输出流进行通信。

TCP/IP 套接字是最可靠的双向流协议，使用 TCP/IP 可以发送任意数量的数据。

实际上，套接字只是计算机上已编号的端口。如果发送方和接收方计算机确定好端口，他们就可以通信了。

客户端与服务器端的通信关系图：



- **TCP/IP 通信连接的简单过程：**

位于 A 计算机上的 TCP/IP 软件向 B 计算机发送包含端口号的消息 ,B 计算机的 TCP/IP 软件接收该消息，并进行检查，查看是否有它知道的程序正在该端口上接收消息。如果有，他就将该消息交给这个程序。

要使程序有效地运行，就必须有一个客户端和一个服务器。

- **通过 Socket 的编程顺序：**

1. 创建服务器 ServerSocket，在创建时，定义 ServerSocket 的监听端口（在这个端口接收客户端发来的消息）。
2. ServerSocket 调用 accept()方法，使之处于阻塞状态。
3. 创建客户端 Socket，并设置服务器的 IP 及端口。
4. 客户端发出连接请求，建立连接。
5. 分别取得服务器和客户端 Socket 的 InputStream 和 OutputStream。
6. 利用 Socket 和 ServerSocket 进行数据传输。
7. 关闭流及 Socket。

2 入门案例

2.1创建服务端

```
public class BasicSocketServer {
    public static void main(String[] args) {
        Socket socket = null;
        BufferedReader br = null;
        try {
            ServerSocket serverSocket = new ServerSocket(8888);

            System.out.println("服务器启动等待监听。。。。");

            //开启端口的监听

           socket =  serverSocket.accept();

            //读取客户端发送的消息

            br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            System.out.println(br.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            if(br != null){
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (socket !=null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

2.2创建客户端

```
public class BasicSocketClient {
    public static void main(String[] args) {
        Socket socket = null;
        PrintWriter pw= null;
        try {
            //创建 Socket 对象，两个参数：1，服务端的 IP 地址，2，服务端所监听
的端口
            socket = new Socket("127.0.0.1",8888);
            pw = new PrintWriter(socket.getOutputStream());
            pw.println("服务端 您好！");
            pw.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            if(pw != null){
                pw.close();
            }
            if(socket !=null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

3 TCP 单向通信

单向通信是指通信双方中，一方固定为发送端，一方则固定为接收端。

3.1创建服务端

```
public class OneWaySocketServer {
    public static void main(String[] args) {
        Socket socket = null;
```

```
        BufferedReader br = null;
        PrintWriter pw = null;
        try{
            ServerSocket serverSocket = new ServerSocket(8888);

            System.out.println("服务端启动，开始监听。。。。。");

            socket = serverSocket.accept();

            System.out.println("连接成功！");

            //读取客户端发送的消息

            br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

            //向客户端写回消息的流对象

            pw = new PrintWriter(socket.getOutputStream());
            while(true){
                String str = br.readLine();

                System.out.println("客户端说："+str);

                if("exit".equals(str)){
                    break;
                }
                pw.println(str);
                pw.flush();
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(br != null){
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(pw != null){
                pw.close();
            }
            if(socket != null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
```

```
                }
            }
        }
    }
}
```

3.2创建客户端

```
public class OneWaySocketClient {
    public static void main(String[] args) {
        Socket socket = null;
        Scanner scanner = null;
        PrintWriter pw = null;
        BufferedReader br = null;
        try{
            socket = new Socket("127.0.0.1",8888);
            //创建键盘输入对象
            scanner = new Scanner(System.in);
            //创建向服务端输出消息的流对象
            pw = new PrintWriter(socket.getOutputStream());
            //创建读取服务端返回消息的流对象
            br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            while(true){
                //通过键盘输入获取需要向服务端发送的消息
                String str = scanner.nextLine();
                //将消息发送到服务端
                pw.println(str);
                pw.flush();
                if("exit".equals(str)){
                    break;
                }
                //读取服务端返回的消息
                String serverInput = br.readLine();
                System.out.println("服务端返回的："+serverInput);
            }
```

```
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(scanner != null){
                scanner.close();
            }
            if(br != null){
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(pw != null){
                pw.close();
            }
            if(socket != null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

4 TCP 双向通信

双向通信是指通信双方中，任何一方都可为发送端，任何一方都可为接收端。

4.1创建服务端

```
public class TwoWaySocketServer {
    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        Socket socket =  null;
        BufferedReader br = null;
        Scanner scanner = null;
        PrintWriter pw = null;
        try{
            serverSocket  = new ServerSocket(8888);
```

```
            System.out.println("服务端启动！监听端口 8888。。。。");
            socket = serverSocket.accept();
            //创建从客户端读取消息的流对象
            br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            //创建键盘输入对象
            scanner = new Scanner(System.in);
            //创建向客户端发送消息的输出流对象
            pw  = new PrintWriter(socket.getOutputStream());
            while(true){
                //读取客户端发送的消息
                String str = br.readLine();
                System.out.println("客户端说："+str);
                String keyInput = scanner.nextLine();
                //发送到客户端
                pw.println(keyInput);
                pw.flush();
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(serverSocket != null){
                try {
                    serverSocket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(socket != null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(br != null){
                try {
```

```
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(scanner != null){
            scanner.close();
        }
        if(pw != null){
            pw.close();
        }
    }
}
```

4.2创建客户端

```
public class TwoWaySocketClient {
    public static void main(String[] args) {
        Socket socket = null;
        Scanner scanner = null;
        PrintWriter pw = null;
        BufferedReader br = null;
        try{
            socket = new Socket("127.0.0.1",8888);

            //键盘输入对象

            scanner = new Scanner(System.in);

            //创建向服务端发送消息的流对象

            pw = new PrintWriter(socket.getOutputStream());

            //创建读取服务端发送消息的流对象

            br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            while(true){
                String keyInput = scanner.nextLine();
                pw.println(keyInput);
                pw.flush();
                String input = br.readLine();

                System.out.println("服务端说："+input);

            }
```

```
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(socket!= null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(pw != null){
                pw.close();
            }
            if(br != null){
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(scanner != null){
                scanner.close();
            }
        }
    }
}
```

5 创建点对点的聊天应用

5.1创建服务端

```
/**
 * 发送消息线程
 */
class Send extends Thread{
    private Socket socket;
    public Send(Socket socket){
        this.socket = socket;
    }
    @Override
    public void run() {
        this.sendMsg();
```

```
    }
    /**
     * 发送消息
     */
    private void sendMsg(){
        Scanner scanner = null;
        PrintWriter pw = null;
        try{
            //创建 Scanner 对象
            scanner = new Scanner(System.in);
            //创建向对方输出消息的流对象
            pw = new PrintWriter(this.socket.getOutputStream());
            while(true){
                String msg = scanner.nextLine();
                pw.println(msg);
                pw.flush();
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally {
            if(scanner != null ){
                scanner.close();
            }
            if(pw != null){
                pw.close();
            }
            if(this.socket != null){
                try {
                    this.socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

/**
 * 接收消息的线程
 */
```

```
class Receive extends Thread{
    private Socket socket;
    public Receive(Socket socket){
        this.socket = socket;
    }
    @Override
    public void run() {
       this.receiveMsg();

    }
    /**
     * 用于接收对方消息的方法
     */
    private void receiveMsg(){
        BufferedReader br = null;
        try{
            //创建用于接收对方发送消息的流对象
            br = new BufferedReader(new
InputStreamReader(this.socket.getInputStream()));
            while(true){
                String msg = br.readLine();

                System.out.println("他说："+msg);

            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(br != null){
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
public class ChatSocketServer {
    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        try{
            serverSocket = new ServerSocket(8888);
```

```
            System.out.println("服务端启动，等待连接。。。。。");
            Socket socket = serverSocket.accept();
            System.out.println("连接成功！");
            new Send(socket).start();
            new Receive(socket).start();
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(serverSocket != null){
                try {
                    serverSocket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

5.2创建客户端

```
/**
 * 用于发送消息的线程类
 */
class ClientSend extends Thread{
    private Socket socket;
    public ClientSend(Socket socket){
        this.socket = socket;
    }
    @Override
    public void run() {
        this.sendMsg();
    }
    /**
     * 发送消息
     */
    private void sendMsg(){
        Scanner scanner = null;
```

```
        PrintWriter pw = null;
        try{
            //创建 Scanner 对象
            scanner = new Scanner(System.in);
            //创建向对方输出消息的流对象
            pw = new PrintWriter(this.socket.getOutputStream());
            while(true){
                String msg = scanner.nextLine();
                pw.println(msg);
                pw.flush();
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally {
            if(scanner != null ){
                scanner.close();
            }
            if(pw != null){
                pw.close();
            }
            if(this.socket != null){
                try {
                    this.socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
/**
 * 用于接收消息的线程类
 */
class ClientReceive extends Thread{
    private Socket socket;
    public ClientReceive(Socket socket){
        this.socket = socket;
    }
    @Override
    public void run() {
        this.receiveMsg();
```

```
    }
    /**
     * 用于接收对方消息的方法
     */
    private void receiveMsg(){
        BufferedReader br = null;
        try{
            //创建用于接收对方发送消息的流对象
            br = new BufferedReader(new
InputStreamReader(this.socket.getInputStream()));
            while(true){
                String msg = br.readLine();
                System.out.println("他说："+msg);
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(br != null){
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
public class ChatSocketClient {
    public static void main(String[] args) {
        try {
            Socket socket = new Socket("127.0.0.1", 8888);
            System.out.println("连接成功！");
            new ClientSend(socket).start();
            new ClientReceive(socket).start();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

5.3优化点对点聊天应用

```
/**
 * 发送消息线程
 */
class Send1 extends Thread{
    private Socket socket;
    private Scanner scanner;
    public Send1(Socket socket,Scanner scanner){
        this.socket = socket;
        this.scanner = scanner;
    }
    @Override
    public void run() {
        this.sendMsg();
    }
    /**
     * 发送消息
     */
    private void sendMsg(){

        PrintWriter pw = null;
        try{
            //创建向对方输出消息的流对象
            pw = new PrintWriter(this.socket.getOutputStream());
            while(true){
                String msg = scanner.nextLine();
                pw.println(msg);
                pw.flush();
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally {
            if(scanner != null ){
                scanner.close();
            }
            if(pw != null){
                pw.close();
            }
```

```
            if(this.socket != null){
                try {
                    this.socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

/**
 * 接收消息的线程
 */
class Receive1 extends Thread{
    private Socket socket;
    public Receive1(Socket socket){
        this.socket = socket;
    }
    @Override
    public void run() {
        this.receiveMsg();
    }
    /**
     * 用于接收对方消息的方法
     */
    private void receiveMsg(){
        BufferedReader br = null;
        try{
            //创建用于接收对方发送消息的流对象
            br = new BufferedReader(new
InputStreamReader(this.socket.getInputStream()));
            while(true){
                String msg = br.readLine();
                System.out.println("他说："+msg);
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(br != null){
```

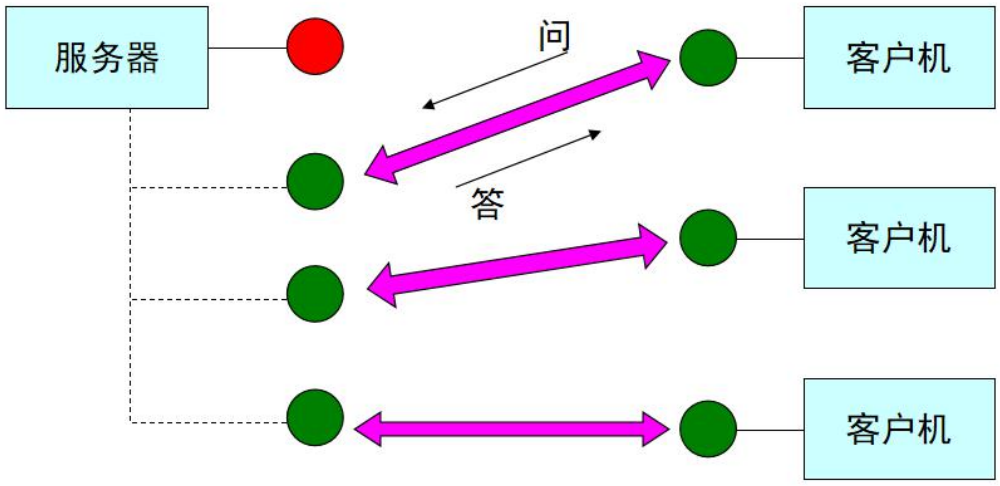
```
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(this.socket != null){
                try {
                    this.socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
public class GoodTCP {
    public static void main(String[] args) {
        Scanner scanner = null;
        ServerSocket serverSocket = null;
        Socket socket = null;
        try{
            scanner = new Scanner(System.in);

            System.out.println("请输入：server,<port> 或者：
<ip>,<port>");
            String str = scanner.nextLine();
            String[] arr = str.split(",");
            if("server".equals(arr[0])){

                //启动服务端

                System.out.println("TCP Server Listen at
"+arr[1]+" .....");
                serverSocket = new
ServerSocket(Integer.parseInt(arr[1]));
                socket = serverSocket.accept();

                System.out.println("连接成功！");


            }else{

                //启动客户端

              socket = new Socket(arr[0],Integer.parseInt(arr[1]));
```

```
                System.out.println("连接成功！");
            }
            //启动发送消息的线程
            new Send1(socket,scanner).start();
            //启动接收消息的线程
            new Receive1(socket).start();
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(serverSocket != null){
                try {
                    serverSocket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

6 一对多应用

6.1一对多应用设计

各 socket 对间独立问答，互相间不需要传递信息



6.2一对多问答型服务器

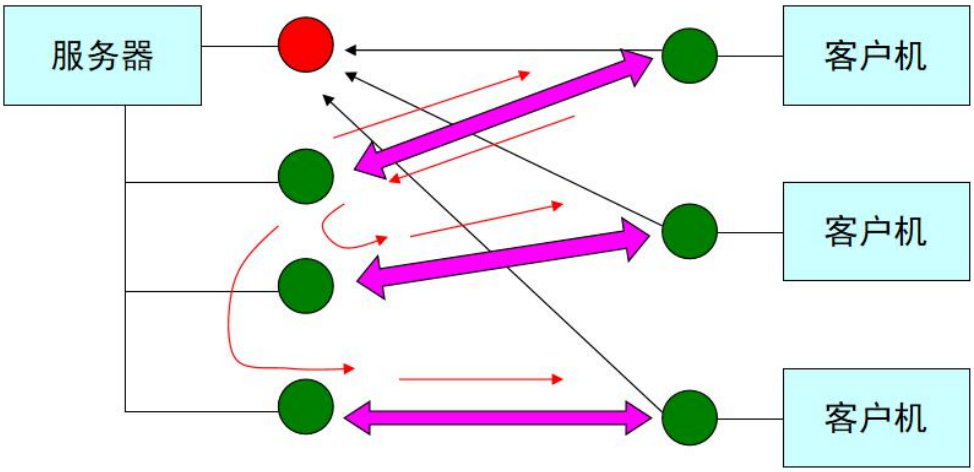
```
/**
 * 定义消息处理线程类
 */
class Msg extends Thread{
    private Socket socket;
    public Msg(Socket socket){
        this.socket = socket;
    }
    @Override
    public void run() {
       this.msg();
    }
    private void msg(){
        BufferedReader br =  null;
        PrintWriter pw = null;
        try{
            br = new BufferedReader(new
InputStreamReader(this.socket.getInputStream()));
            pw = new PrintWriter(this.socket.getOutputStream());
            while(true){
                pw.println(br.readLine()+" [ok]");
                pw.flush();
            }
        }catch(Exception e){

            System.out.println(this.socket.getInetAddress()+" 断线了！
");

            e.printStackTrace();
        }finally{
            if(br != null){
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(pw != null){
                pw.close();
            }
        }
    }
```

```
}
public class EchoServer {
    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        try{
            serverSocket = new ServerSocket(8888);
            while(true){
                Socket socket = serverSocket.accept();
                new Msg(socket).start();
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(serverSocket != null){
                try {
                    serverSocket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

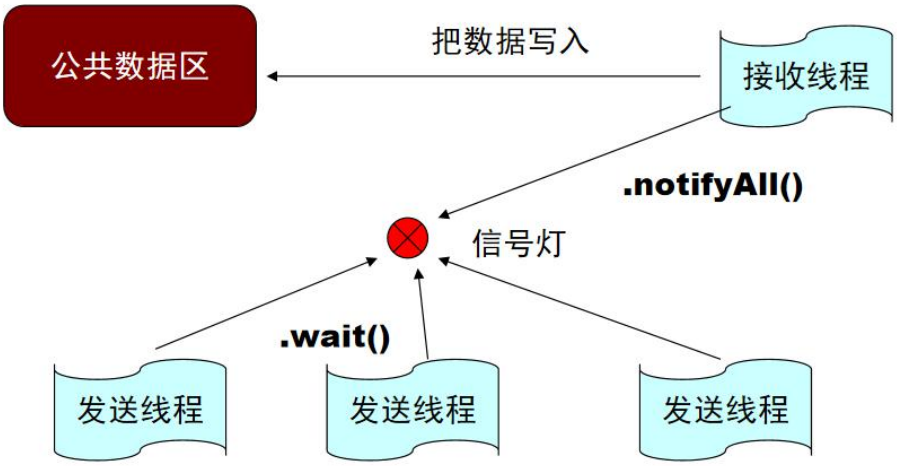
6.3 一对多聊天服务器

6.3.1 服务器设计

6.3.1.1 服务器的连接设计



6.3.1.2 服务器的线程设计



6.4创建一对多聊天服务应用

```
/**
 * 接收客户端消息的线程类
 */
class ChatReceive extends Thread{
    private Socket socket;
    public ChatReceive(Socket socket){
        this.socket =socket;
    }
    @Override
    public void run() {
       this.receiveMsg();
    }
    /**
     * 实现接收客户端发送的消息
     */
    private void receiveMsg(){
        BufferedReader br = null;
        try{
            br = new BufferedReader(new
InputStreamReader(this.socket.getInputStream()));
            while(true){
                String msg = br.readLine();
                synchronized ("abc"){
                    //把读取到的数据写入公共数据区

ChatRoomServer.buf="["+this.socket.getInetAddress()+"] "+msg;
                    //唤醒发送消息的线程对象。
                    "abc".notifyAll();
                }
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(br != null){
                try {
                    br.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(this.socket != null){
                try {
                    this.socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
/**
 * 向客户端发送消息的线程类
 */
class ChatSend extends Thread{
    private Socket socket;
    public ChatSend(Socket socket){
        this.socket = socket;
    }
    @Override
    public void run() {
       this.sendMsg();
    }
    /**
     * 将公共数据区的消息发送给客户端
     */
    private void sendMsg(){
        PrintWriter pw = null;
        try{
            pw = new PrintWriter(this.socket.getOutputStream());
            while(true){
                synchronized ("abc"){
                    //让发送消息的线程处于等待状态
                    "abc".wait();
                    //将公共数据区中的消息发送给客户端
                    pw.println(ChatRoomServer.buf);
                    pw.flush();
```

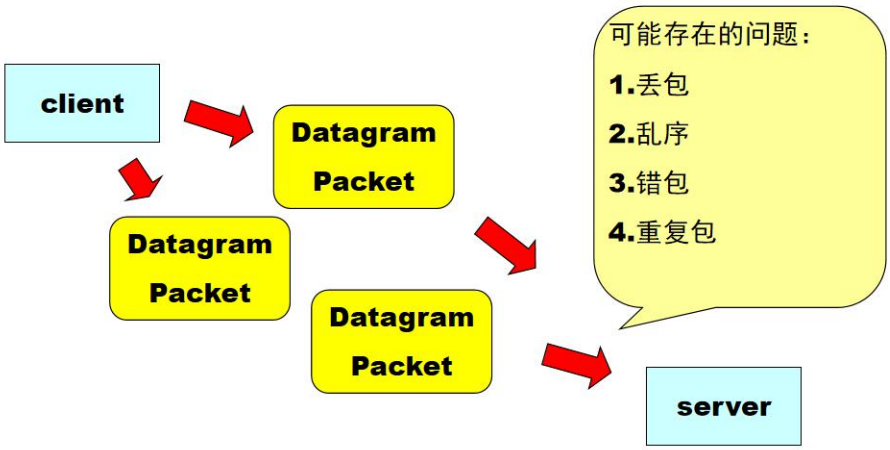
```
                }
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(pw != null){
                pw.close();
            }
            if(this.socket != null){
                try {
                    this.socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
public class ChatRoomServer {

    //定义公共数据区
    public static String buf;
    public static void main(String[] args) {
        System.out.println("Chat Server Version 1.0");
        System.out.println("Listen at 8888.....");
        ServerSocket serverSocket =null;
        try{
            serverSocket = new ServerSocket(8888);
            while(true){
                Socket socket = serverSocket.accept();

                System.out.println("连接到：
"+socket.getInetAddress());
                new ChatReceive(socket).start();
                new ChatSend(socket).start();
            }
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(serverSocket != null){
                try {
                    serverSocket.close();
                } catch (IOException e) {
                    e.printStackTrace();
```

```
                }
            }
        }
    }
}
```

四、 UDP 通信的实现

1 UDP 通信介绍

UDP 协议与之前讲到的 TCP 协议不同，是面向无连接的，双方不需要建立连接便可通信。UDP 通信所发送的数据需要进行封包操作（使用 DatagramPacket 类），然后才能接收或发送（使用 DatagramSocket 类）。



- **DatagramPacket：数据容器（封包）的作用**

  此类表示数据报包。 数据报包用来实现将发送的数据进行封包处理的。

  常用方法：

  - DatagramPacket(byte[] buf, int length) ：构造数据报包，用来指定长度为 length 的数据包。
  - DatagramPacket(byte[] buf, int length, InetAddress address, int port) ：构造数据报包，用来将长度为 length 的包发送到指定主机上的指定端口号。

- getAddress() ：获取发送或接收方计算机的 IP 地址，此数据报将要发往该机器或者是从该机器接收到的。
- getData() ：获取发送或接收的数据。
- setData(byte[] buf) ：设置发送的数据。

❑ **DatagramSocket：用于发送或接收数据报包**

当客户端要向服务端发送数据时，需要在客户端产生一个 DatagramSocket 对象，在服务端产生一个 DatagramSocket 对象。客户端的 DatagramSocket 将 DatagramPacket 发送到网络上，然后被服务端的 DatagramSocket 接收。

常用方法：

- DatagramSocket(int port) ：创建数据报套接字并将其绑定到本地主机上的指定端口。
- send(DatagramPacket p) ：从此套接字发送数据报包。
- receive(DatagramPacket p) ：从此套接字接收数据报包。
- close() ：关闭此数据报套接字。

❑ **UDP 通信编程基本步骤：**

1. 创建服务器端的 DatagramSocket，创建时，定义服务器端的监听端口。
2. 创建客户端的 DatagramSocket，创建时，定义客户端的监听端口。
3. 在服务器端定义 DatagramPacket 对象，封装待接收的数据包。
4. 客户端将数据报包发送出去。
5. 服务器端接收数据报包。

1 入门案例

1.1 创建服务端

```
public class UDPTest {
    public static void main(String[] args) {
        DatagramSocket datagramSocket = null;
        try{
            //创建服务端接收数据的 DatagramSocket 对象
            datagramSocket = new DatagramSocket(9999);
            //创建数据缓存区
            byte[] b = new byte[1024];
            //创建数据报包对象
            DatagramPacket dp =new DatagramPacket(b,b.length);
            //等待接收客户端所发送的数据
            datagramSocket.receive(dp);
            String str = new String(dp.getData(),0,dp.getLength());
            System.out.println(str);
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(datagramSocket != null){
                datagramSocket.close();
            }
        }
    }
}
```

1.2 创建客户端

```
public class UDPClient {
    public static void main(String[] args) {
        DatagramSocket ds =null;
        try {
            //消息需要进行类型转换，转换成字节数据类型。
```

```
            byte[] b = "北京尚学堂".getBytes();


            //创建数据报包装对象 DatagramPacket
            DatagramPacket dp = new DatagramPacket(b, b.length, new
InetSocketAddress("127.0.0.1", 9999));
            //创建数据发送对象 DatagramSocket,需要指定消息的发送端口
            ds = new DatagramSocket(8888);
            //发送消息
            ds.send(dp);
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(ds != null){
                ds.close();
            }
        }
    }
}
```

2 传递基本数据类型

2.1创建服务端

```
public class BasicTypeServer {
    public static void main(String[] args) {
        DatagramSocket datagramSocket = null;
        DataInputStream ds = null;
        try{
         datagramSocket = new DatagramSocket(9999);
         byte[] buf = new byte[1024];
         DatagramPacket datagramPacket = new
DatagramPacket(buf,buf.length);
         datagramSocket.receive(datagramPacket);
         //通过基本数据输入流对象获取传递的数据
            ds = new DataInputStream(new
ByteArrayInputStream(datagramPacket.getData()));
            System.out.println(ds.readLong());
```

```
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(datagramSocket != null){
                datagramSocket.close();
            }
            if(ds != null){
                try {
                    ds.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

2.2创建客户端

```
public class BasicTypeClient {
    public static void main(String[] args) {
        long n = 2000L;
        ByteArrayOutputStream bos = null;
        DataOutputStream dos = null;
        DatagramSocket datagramSocket = null;
        try{
          bos = new ByteArrayOutputStream();
          dos = new DataOutputStream(bos);
          dos.writeLong(n);

          //将基本数据类型数据转换成字节数组类型

            byte[] arr = bos.toByteArray();
            DatagramPacket dp = new DatagramPacket(arr,arr.length,new
InetSocketAddress("127.0.0.1",9999));
            datagramSocket = new DatagramSocket(9000);
            datagramSocket.send(dp);
        }catch (Exception e){
            e.printStackTrace();
        }finally{
            if(bos != null ){
                try {
                    bos.close();
                } catch (IOException e) {
```

```
                e.printStackTrace();
            }
        }
        if(dos != null){
            try {
                dos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(datagramSocket != null){
            datagramSocket.close();
        }
    }
}
```

3 传递自定义对象类型

3.1创建 Person 类

```
/**
 * 当该对象需要在网络上传输时，一定要实现 Serializable 接口
 */
public class Person implements Serializable {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
```

```
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

3.2创建服务端

```
public class ObjectTypeServer {
    public static void main(String[] args) {
        DatagramSocket datagramSocket = null;
        ObjectInputStream objectInputStream = null;
        try{
        datagramSocket = new DatagramSocket(9999);
        byte[] b = new byte[1024];
            DatagramPacket dp = new DatagramPacket(b,b.length);
            datagramSocket.receive(dp);

            //对接收的内容做类型转换

            objectInputStream = new ObjectInputStream(new
ByteArrayInputStream(dp.getData()));
            System.out.println(objectInputStream.readObject());
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(datagramSocket != null){
                datagramSocket.close();
            }
            if(objectInputStream != null){
                try {
                    objectInputStream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

```
}
```

3.3创建客户端

```
public class ObjectTypeClient {
    public static void main(String[] args) {
        DatagramSocket datagramSocket = null;
        ByteArrayOutputStream bos = null;
        ObjectOutputStream oos = null;
        try{
            Person p = new Person();
            p.setName("Oldlu");
            p.setAge(18);

            bos = new ByteArrayOutputStream();
            oos = new ObjectOutputStream(bos);
            oos.writeObject(p);
            byte[] arr = bos.toByteArray();

            DatagramPacket dp = new DatagramPacket(arr,arr.length,new
InetSocketAddress("127.0.0.1",9999));
            datagramSocket = new DatagramSocket(8888);
            datagramSocket.send(dp);
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(datagramSocket != null ){
                datagramSocket.close();
            }
            if(bos != null){
                try {
                    bos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(oos != null){
                try {
                    oos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
```

```
        }
    }
}
```

五、 本章总结

- 端口是虚拟的概念，并不是说在主机上真的有若干个端口。
- 在 www 上，每一信息资源都有统一且唯一的地址，该地址就叫 URL（Uniform Resource Locator），它是 www 的统一资源定位符。
- TCP 与 UDP 的区别
  - TCP 是面向连接的，传输数据安全，稳定，效率相对较低。
  - UDP 是面向无连接的，传输数据不安全，效率较高。
- Socket 通信是一种基于 TCP 协议，建立稳定连接的点对点的通信。
- 网络编程是由 java.net 包来提供网络功能。
  - InetAddress：封装计算机的 IP 地址和 DNS（没有端口信息！）。
  - InetSocketAddress：包含 IP 和端口，常用于 Socket 通信。
  - URL：以使用它的各种方法来对 URL 对象进行分割、合并等处理。
- 基于 TCP 协议的 Socket 编程和通信
  - “请求-响应”模式：
    - Socket 类：发送 TCP 消息。
    - ServerSocket 类：创建服务器。
- UDP 通讯的实现
  - DatagramSocket：用于发送或接收数据报包。
  - 常用方法：send()、receive()、 close()。
- DatagramPacket：数据容器（封包）的作用

- 常用方法：构造方法、getAddrress(获取发送或接收方计算机的 IP 地址)、getData(获取发送或接收的数据)、setData(设置发送的数据)。