网络编程基本概念

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：
   - 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）。
   - 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）。

实时效果反馈

1.如下哪个选项不是计算机网络中的组成部分?

A 计算机设备、网络设备、通信线路;

B 网络操作系统、网络管理软件;

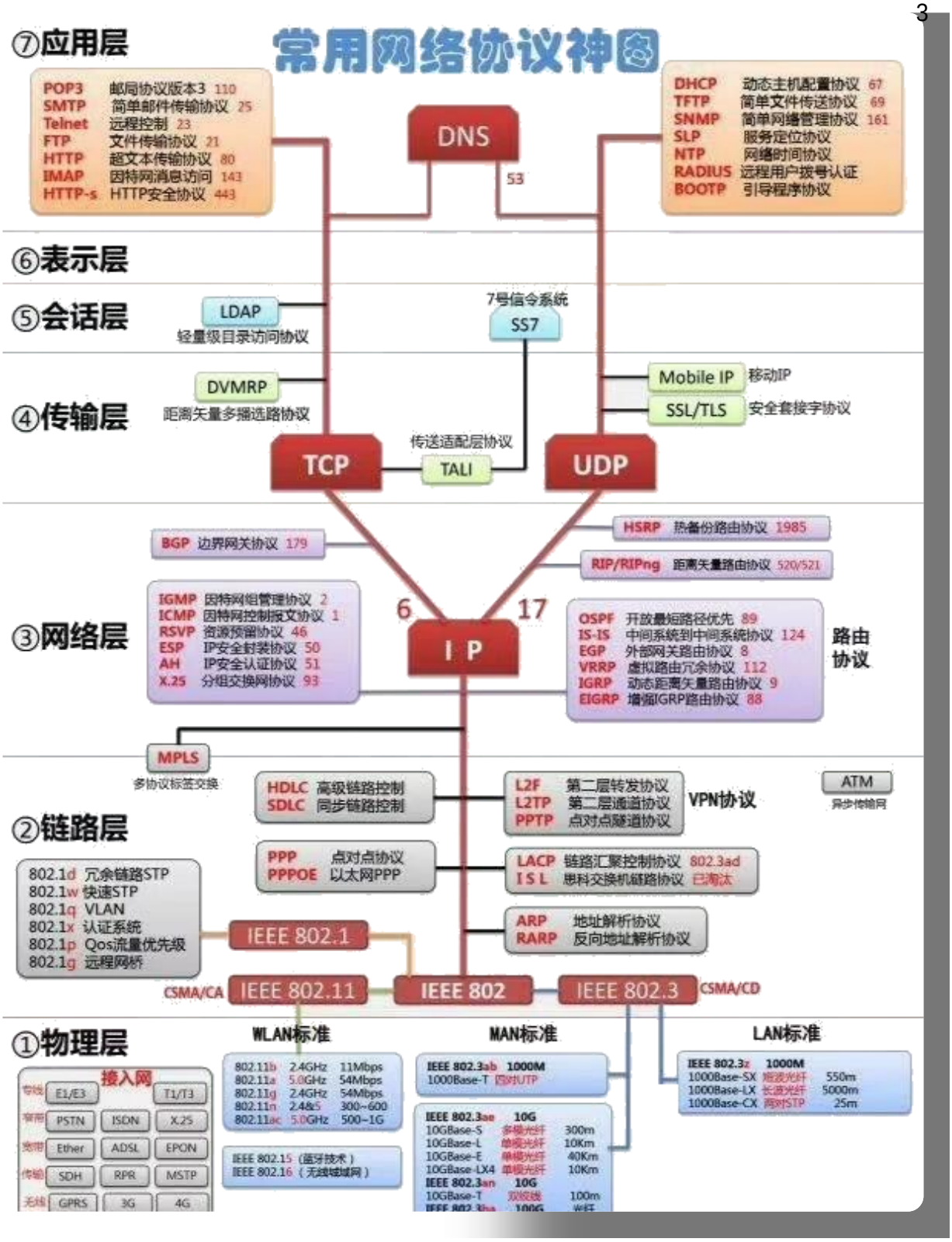
C 网络通信协议;

D 聊天软件;

答案

1=>D

网络通信协议



什么是网络通信协议

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。

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。

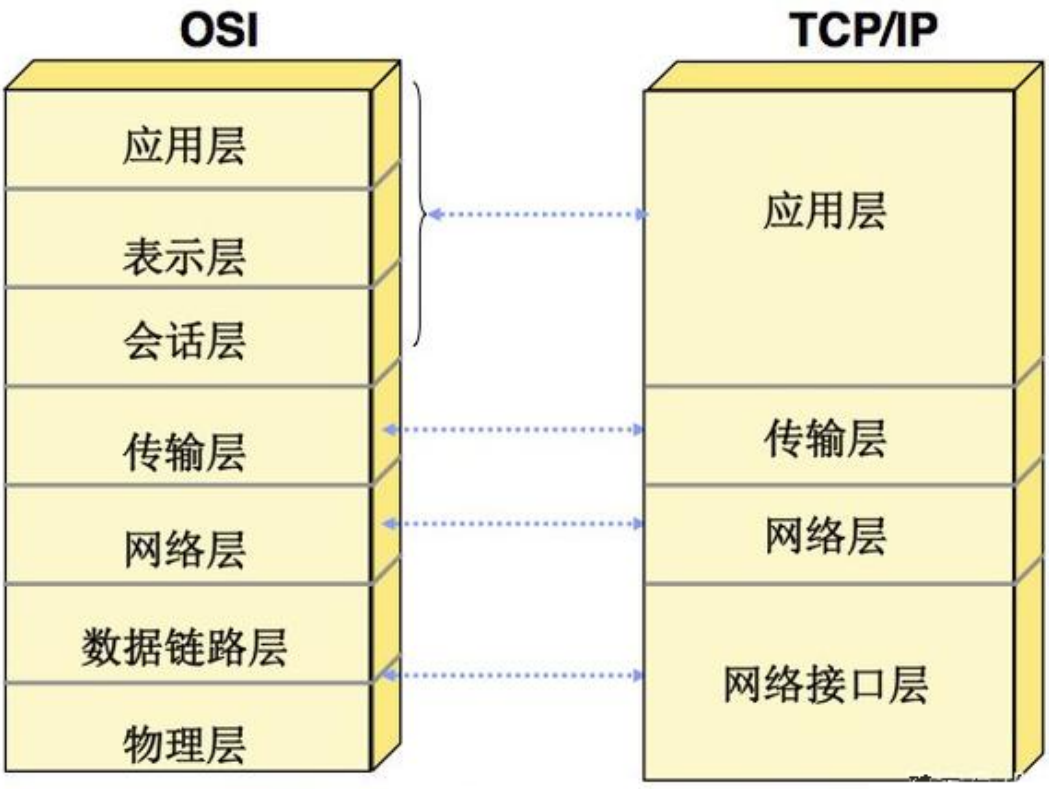
OSI七层协议模型：

网络协议的分层

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 模型，也是按照层次划分，共四层：应用层，传输层，网络层，网络接口层（物理+数据链路层）。

OSI模型与TCP/IP模型的对应关系：



实时效果反馈

1.如下哪个选项不是TCP/IP 模型中的分层。

A 应用层;

B 传输层;

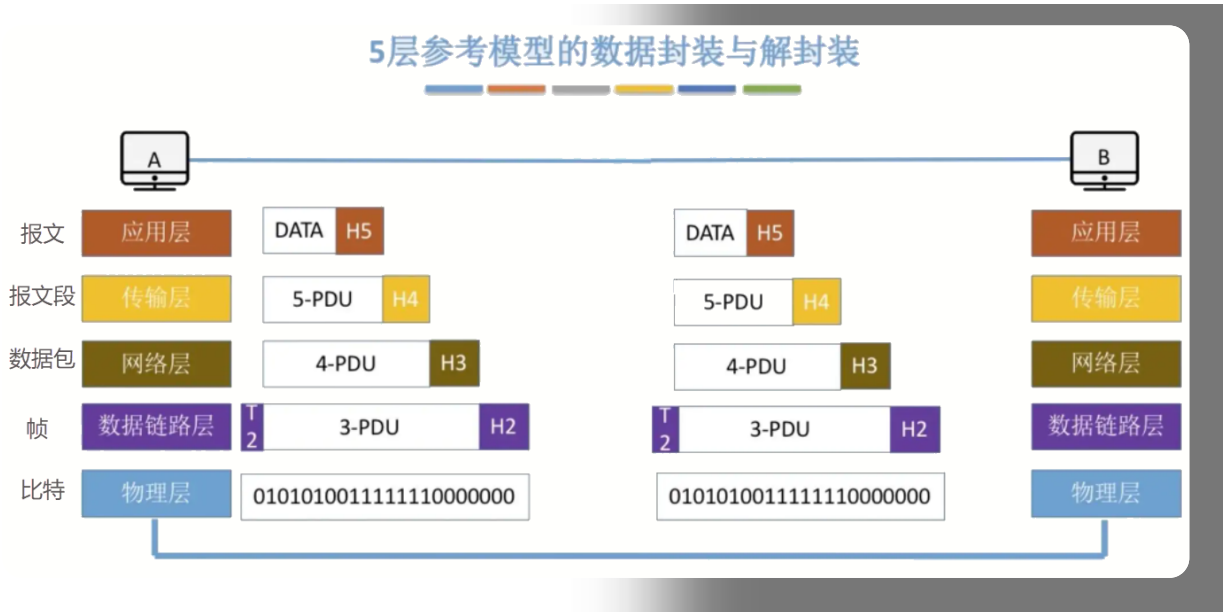
C 网络层;

D 物理层;

答案

1=>D

数据封装与解封



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。

- 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。

- 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。

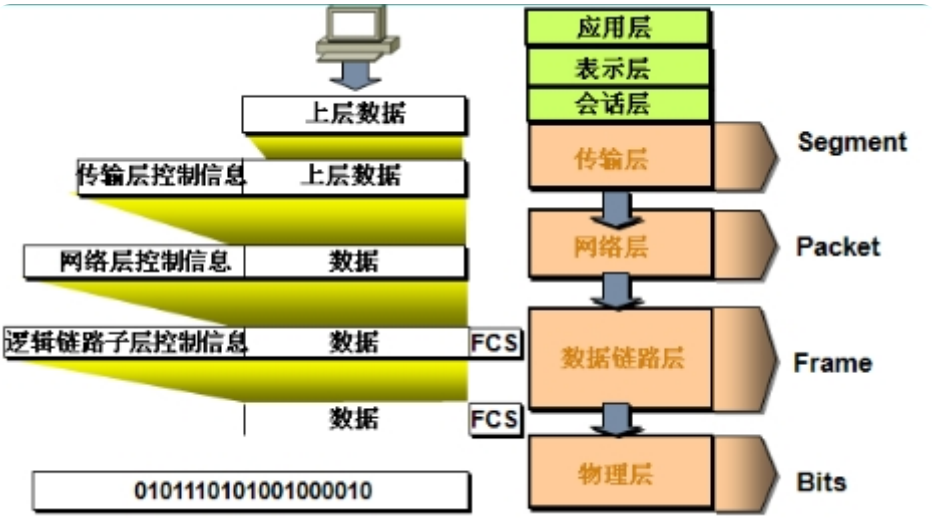
从以上传输过程中，可以总结出以下规则：

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。

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

实时效果反馈

1.在网络通信中，对数据封装步骤描述错误的是?

A 应用层将数据交给传输层;

B 传输层将数据交给网络链路层;

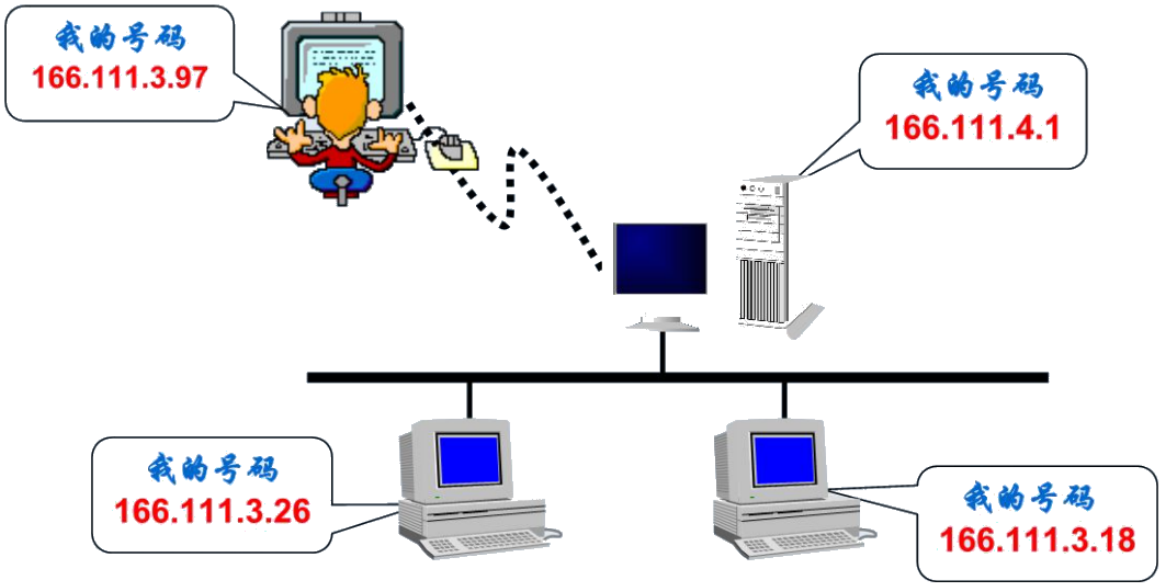
C 网络链路层将数据交给物理层;

D 物理层将数据交给网络链路层;

答案

1=>D

IP地址



IP是计算机的地址。

相当于：你家的地址，或者你手机的号码

IP地址

IP是Internet Protocol Address，即"互联网协议地址"。

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。

IP地址分类：

类别	最大网络数	IP地址范围	单个网段最大主机数	私有IP地址范围
A	126(2^7-2)	1.0.0.1-127.255.255.254	16777214	10.0.0.0-10.255.255.255
B	16384(2^14)	128.0.0.1-191.255.255.254	65534	172.16.0.0-172.31.255.255
C	2097152(2^21)	192.0.0.1-223.255.255.254	254	192.168.0.0-192.168.255.255

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。

IPV6，万物互联啦

地球每平方米**1000**个地址。

你的电饭煲都可以给个**IP**了！

> IPV4，采用32位地址长度，只有大约43亿个地址，它只有4段数字，每一段最大不超过255。随着互联网的发展，IP地址不够用了，在2019年11月25日IPv4位地址分配完毕。
>
> IPv6采用128位地址长度，几乎可以不受限制地提供地址。按保守方法估算IPv6实际可分配的地址，整个地球的每平方米面积上仍可分配1000多个地址。

IP地址实际上是一个32位整数（称为IPv4），以字符串表示的IP地址如`192.168.0.1`实际上是把32位整数按8位分组后的数字表示，目的是便于阅读。

IPv6地址实际上是一个128位整数，它是目前使用的IPv4的升级版，以字符串表示类似于`2001:0db8:85a3:0042:1000:8a2e:0370:7334`

公有地址

公有地址（Public address）由Inter NIC（Internet Network Information Center互联网信息中心）负责。这些IP地址分配给注册并向Inter NIC提出申请的组织机构。通过它直接访问互联网。

私有地址

私有地址（Private address）属于非注册地址，专门为组织机构内部使用。

以下列出留用的内部私有地址

A类 10.0.0.0--10.255.255.255

B类 172.16.0.0--172.31.255.255

C类 192.168.0.0--192.168.255.255

> **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

实时效果反馈

1.如下关于IP的说法，错误的是。

A IP用来识别计算机通信地址的。

B 192.168.0.0--192.168.255.255为私有地址，专门为组织机构内部使用

C 127.0.0.1 本机地址。

D IP用来识别计算机中进行通信的不同应用程序。

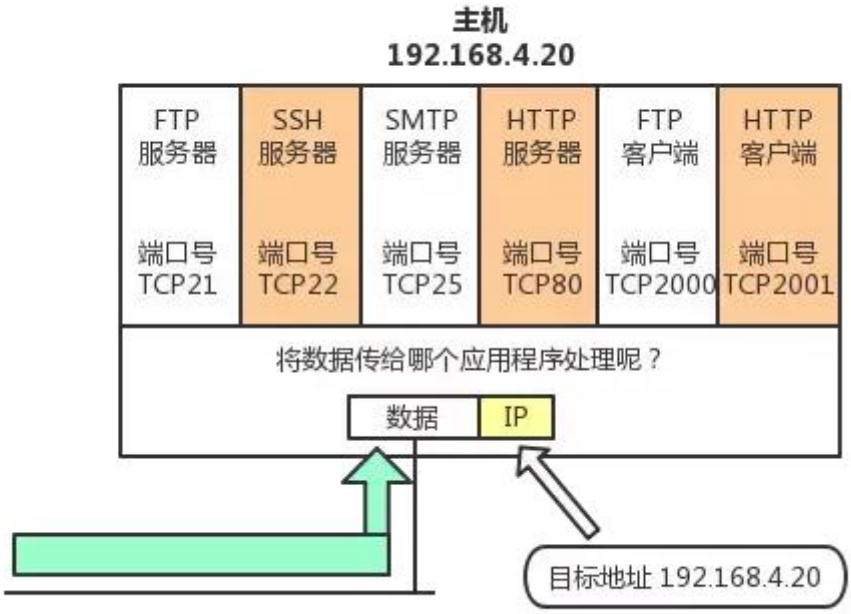
答案

1=>D

端口port

端口号用来识别计算机中进行通信的应用程序。因此，它也被称为程序地址。

一台计算机上同时可以运行多个程序。传输层协议正是利用这些端口号识别本机中正在进行通信的应用程序，并准确地进行数据传输。



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。
- IP地址好比为电话号码，而端口号就好比为分机号。

端口分配

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。

操作系统中一共提供了0~65535可用端口范围。

按端口号分类：

公认端口（Well Known Ports）：从0到1023，它们紧密绑定（binding）于一些服务。通常这些端口的通讯明确表明了某种服务的协议。例如：80端口实际上总是HTTP通讯。

注册端口（Registered Ports）：从1024到65535。它们松散地绑定于一些服务。也就是说有许多服务绑定于这些端口，这些端口同样用于许多其它目的。例如：许多系统处理动态端口从1024左右开始。

实时效果反馈

1.如下关于端口的说法，正确的是?

A 端口号用来识别计算机通信地址的。

B 端口号用来识别计算机中进行通信的不同应用程序。

C 端口号用来识别计算机中网卡。

D 端口号用来识别路由器。

答案

1=>B

URL

URL作用：

URL（Uniform Resource Locator），是互联网的统一资源定位符。用于识别互联网中的信息资源。通过URL我们可以访问文件、数据库、图像、新闻等。

在互联网上，每一信息资源都有统一且唯一的地址，该地址就叫URL，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。

实时效果反馈

1.如下关于URL说法，错误的是?

A URL全称为统一资源定位符；

B URL由4部分组成；

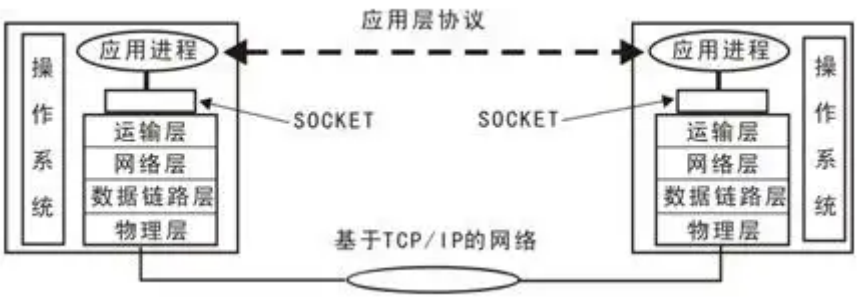
C URL用来识别计算机中网卡。

D 在java.net包中提供了对URL操作的类。

答案

1=>C

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。

实时效果反馈

1.如下关于Socket说法，错误的是?

A Socket是网络层供给传输层的编程接口；

B Socket是传输层供给应用层的编程接口；

C 使用Socket编程可以开发客户机和服务器应用程序；

D Socket应用层与传输层之间的桥梁；

答案

1=>A

TCP协议和UDP协议

TCP协议

TCP（Transmission Control Protocol，传输控制协议）。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。

TCP在建立连接时又分三步走：

n 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。

n 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）。

n 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。

n 这就是所说的TCP的三次握手（Three-way Handshake）。

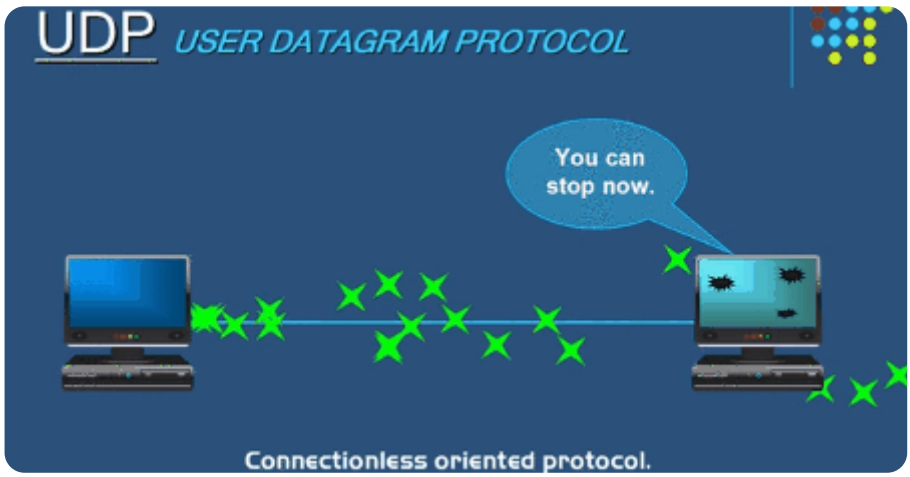
UDP协议

UDP（User Data Protocol，用户数据报协议）

UDP是一个非连接的协议，传输数据之前源端和终端不建立连接，当它想传送时就简单地去抓取来自应用程序的数据，并尽可能快地把它扔到网络上。 在发送端，UDP传送数据的速度仅仅是受应用程序生成数据的速度、 计算机的能力和传输带宽的限制； 在接收端，UDP把每个消息段放在队列中，应用程序每次从队列中读一个消息段。

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。

UDP 因为没有拥塞控制，一直会以恒定的速度发送数据。即使网络条件不好，也不会对发送速率进行调整。这样实现的弊端就是在网络条件不好的情况下可能会导致丢包，但是优点也很明显，在某些实时性要求高的场景（比如电话会议）就需要使用 UDP 而不是 TCP



TCP和UDP区别

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。

	UDP	TCP
是否连接	无连接	面向连接
是否可靠	不可靠传输，不使用流量控制和拥塞控制	可靠传输，使用流量控制和拥塞控制
连接对象个数	支持一对一，一对多，多对一和多对多交互通信	只能是一对一通信
传输方式	面向报文	面向字节流
首部开销	首部开销小，仅8字节	首部最小20字节，最大60字节
适用场景	适用于实时应用（IP电话、视频会议、直播等）	适用于要求可靠传输的应用，例如文件传输

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。

实时效果反馈

1.如下关于TCP和UDP协议，说法错误的是。

A TCP是面向连接的，传输数据安全，稳定，效率相对较低

B UDP是面向无连接的，传输数据不安全，效率较高。

C UDP适用于实时应用（IP电话、视频会议、直播等）

D UDP适用于要求可靠传输的应用，例如文件传输

答案

1=>D

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。

InetAddress的使用

作用：封装计算机的IP地址和DNS（没有端口信息）

> 注：DNS是Domain Name System，域名系统。

> **特点：**

> 这个类没有构造方法。如果要得到对象，只能通过静态方法：getLocalHost()、getByName()、 getAllByName()、getAddress()、getHostName()

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。

```
1  public class InetTest {
2      public static void main(String[]
   args)throws Exception {
3          //实例化InetAddress对象
4          InetAddress inetAddress =
   InetAddress.getLocalHost();
5          //返回当前计算机的IP地址
6
    System.out.println(inetAddress.getHostAddre
   ss());
7          //返回当前计算机名
8
    System.out.println(inetAddress.getHostName(
   ));
9      }
10 }
```

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());
    }
}
```

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("14.215.177.38");

 System.out.println(inetAddress.getHostAddress());

 System.out.println(inetAddress.getHostName());
    }
}
```

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());
    }
}
```

URL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("https://www.itbaizhan.com/search.html?kw=java");
        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());
    }
}
```

通过URL实现最简单的网络爬虫

```
public class UrlTest2{
    public static void main(String[] args)throws Exception {
            URL url = new URL("https://www.itbaizhan.com/");
            try (BufferedReader br = new BufferedReader(new InputStreamReader(url.openStream()))) {
```

```
                StringBuilder sb = new StringBuilder();
                String temp;
            /*
             * 这样就可以将网络内容下载到本地机器。
             * 然后进行数据分析，建立索引。这也是搜索引擎的第一步。
             */
                while ((temp = br.readLine()) != null) {
                    sb.append(temp);
                }
                System.out.println(sb);

            } catch (Exception e) {
                e.printStackTrace();
            }
    }
}
```

TCP通信的实现和项目案例

TCP通信实现原理

前边我们提到TCP协议是面向的连接的，在通信时客户端与服务器端必须建立连接。在网络通讯中，第一次主动发起通讯的程序被称作客户端(Client)程序，简称客户端，而在第一次通讯中等待连接的程序被称作服务器端(Server)程序，简称服务器。一旦通讯建立，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。

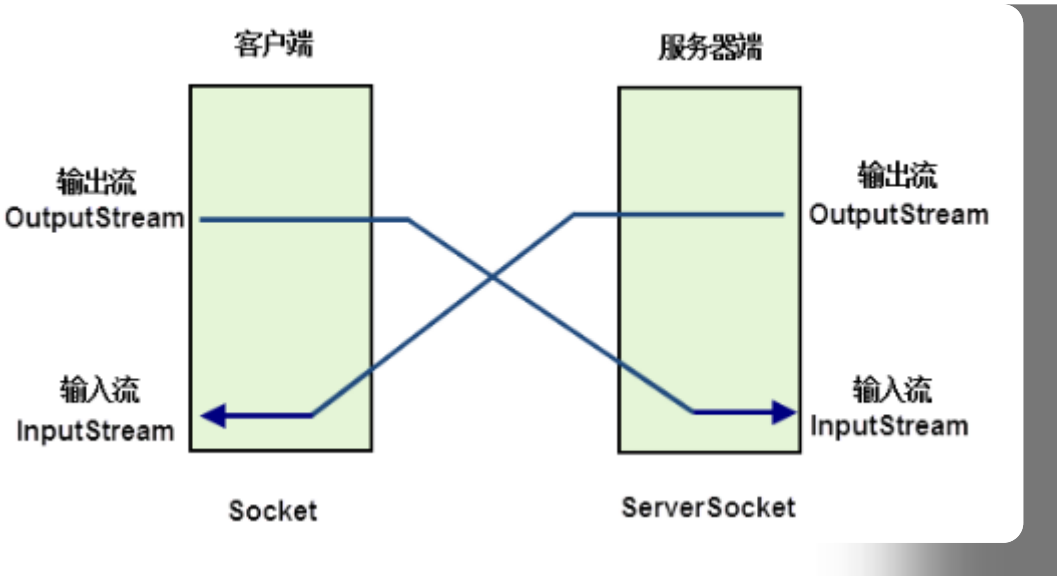
套接字Socket是一种进程间的数据交换机制。这些进程既可以在同一机器上，也可以在通过网络连接的不同机器上。换句话说，套接字起到通信端点的作用。单个套接字是一个端点，而一对套接字则构成一个双向通信信道，使非关联进程可以在本地或通过网络进行数据交换。一旦建立套接字连接，数据即可在相同或不同的系统中双向或单向发送，直到其中一个端点关闭连接。套接字与主机地址和端口地址相关联。主机地址就是客户端或服务器程序所在的主机的IP地址。端口地址是指客户端或服务器程序使用的主机的通信端口。

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。

实时效果反馈

1.如下对TCP连接模式描述正确的是?

A 基于请求模式；

B 基于响应模式；

C 基于请求与响应模式；

D 基于广播模式；

答案

1=>C

TCP通信入门案例

创建服务端

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

        System.out.println("服务器启动等待监听。。。。");
        //创建ServerSocket
        try(ServerSocket ss =new ServerSocket(8888);
            //监听8888端口，此时线程会处于阻塞状态。
            Socket socket = ss.accept();
            //连接成功后会得到与客户端对应的Socket对象，并解除线程阻塞。
```

```
            //通过客户端对应的Socket对象中的输入流对象，获取客户端发送过来的消息。
            BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()))
        ){


 System.out.println(br.readLine());
        }catch(Exception e){
            e.printStackTrace();
            System.out.println("服务器启动失败。。。。");
        }
    }
}
```

创建客户端

```
public class BasicSocketClient {
    public static void main(String[] args) {
        //创建Socket对象
        try(Socket socket =new Socket("127.0.01",8888);
            //创建向服务端发送消息的输出流对象。
            PrintWriter pw = new PrintWriter(socket.getOutputStream())){
            pw.println("服务端，您好！");
            pw.flush();
        }catch(Exception e){
```

```
            e.printStackTrace();
        }
    }
}

```

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。

创建服务端

```
public class OneWaySocketServer {
    public static void main(String[] args) {
        System.out.println("服务端启动，开始监听。。。。。");
        try(ServerSocket serverSocket = new ServerSocket(8888);
            //监听8888端口，获与取客户端对应的Socket对象
            Socket socket = serverSocket.accept();
            //通过与客户端对应的Socket对象获取输入流对象
            BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            //通过与客户端对应的Socket对象获取输出流对象
```

```
            PrintWriter pw = new PrintWriter(socket.getOutputStream())){
            System.out.println("连接成功！");
            while(true){
                //读取客户端发送的消息
                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();
            System.out.println("服务端启动失败。。。。。");
        }
    }
}

```

创建客户端

```
public class OneWaySocketClient {
    public static void main(String[] args) {
        //获取与服务端对应的Socket对象
        try(Socket socket = new Socket("127.0.0.1",8888);
```

```
            //创建键盘输入对象
            Scanner scanner = new Scanner(System.in);
            //通过与服务端对应的Socket对象获取输出流对象
            PrintWriter pw = new PrintWriter(socket.getOutputStream());
            //通过与服务端对应的Socket对象获取输入流对象
            BufferedReader 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();
        }
    }
}
```

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。

创建服务端

```
public class TwoWaySocketServer {
    public static void main(String[] args) {
        System.out.println("服务端启动！监听端口8888。。。。");
        try(ServerSocket serverSocket  = new ServerSocket(8888);
            Socket socket = serverSocket.accept();
            //创建键盘输入对象
            Scanner scanner = new Scanner(System.in);
            //通过与客户端对应的Socket对象获取输入流对象
```

```
            BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            //通过与客户单对应的Socket对象获取输出流对象
            PrintWriter 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();
        }
    }
}
```

创建客户端

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

```
        try(Socket socket = new Socket("127.0.0.1", 8888);
            //创建键盘输入对象
            Scanner  scanner = new Scanner(System.in);
            //通过与服务端对应的Socket对象获取输入流对象
            BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            //通过与服务端对应的Socket对象获取输出流对象
            PrintWriter pw = new PrintWriter(socket.getOutputStream());){

            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();
        }
    }
}
```

创建点对点的聊天应用

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(){
        //创建Scanner对象
        try(Scanner scanner = new Scanner(System.in);
            //创建向对方输出消息的流对象
            PrintWriter pw = new PrintWriter(this.socket.getOutputStream());)
        {

            while(true){
                String msg = scanner.nextLine();
                pw.println(msg);
```

```
                pw.flush();
            }
        }catch(Exception 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(){
        //创建用于接收对方发送消息的流对象
        try(BufferedReader  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();
        }
    }
}
public class ChatSocketServer {
    public static void main(String[] args) {

        try(ServerSocket 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();
        }
    }
}

```

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(){
        //创建Scanner对象
        try(Scanner scanner = new Scanner(System.in);
            //创建向对方输出消息的流对象
            PrintWriter pw = new PrintWriter(this.socket.getOutputStream());)
        {

            while(true){
                String msg = scanner.nextLine();
                pw.println(msg);
                pw.flush();
            }
        }catch(Exception e){
            e.printStackTrace();
```

```
29          }
30      }
31  }
32  /**
33   * 用于接收消息的线程类
34   */
35  class ClientReceive extends Thread{
36      private Socket socket;
37      public ClientReceive(Socket socket){
38          this.socket = socket;
39      }
40      @Override
41      public void run() {
42          this.receiveMsg();
43      }
44      /**
45       * 用于接收对方消息的方法
46       */
47      private void receiveMsg(){
48          //创建用于接收对方发送消息的流对象
49          try(BufferedReader br = new
    BufferedReader(new
    InputStreamReader(this.socket.getInputStream
    ()));){
50
51              while(true){
52                  String msg = br.readLine();
53                  System.out.println("他
    说: "+msg);
54              }
55          }catch(Exception e){
56              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();
        }
    }
}
```

优化点对点聊天应用

```
/**
 * 发送消息线程
 */
class Send extends Thread{
    private Socket socket;
    private Scanner scanner;
    public Send(Socket socket,Scanner
scanner){
        this.socket = socket;
        this.scanner = scanner;
```

```
    }
    @Override
    public void run() {
        this.sendMsg();
    }
    /**
     * 发送消息
     */
    private void sendMsg(){

        //创建向对方输出消息的流对象
        try(PrintWriter pw = new
PrintWriter(this.socket.getOutputStream()))
{

            while(true){
                String msg =
scanner.nextLine();
                pw.println(msg);
                pw.flush();
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}

/**
 * 接收消息的线程
 */
class Receive extends Thread{
    private Socket socket;
```

```
39      public Receive(Socket socket){
40          this.socket = socket;
41      }
42      @Override
43      public void run() {
44          this.receiveMsg();
45      }
46      /**
47       * 用于接收对方消息的方法
48       */
49      private void receiveMsg(){
50          //创建用于接收对方发送消息的流对象
51          try(BufferedReader br = new
BufferedReader(new
InputStreamReader(this.socket.getInputStrea
m()))){
52
53              while(true){
54                  String msg = br.readLine();
55                  System.out.println("他
说: "+msg);
56              }
57          }catch(Exception e){
58              e.printStackTrace();
59          }
60      }
61  }
62  public class GoodTCP {
63      public static void main(String[] args)
{
64          Scanner scanner = null;
65          ServerSocket serverSocket = null;
```

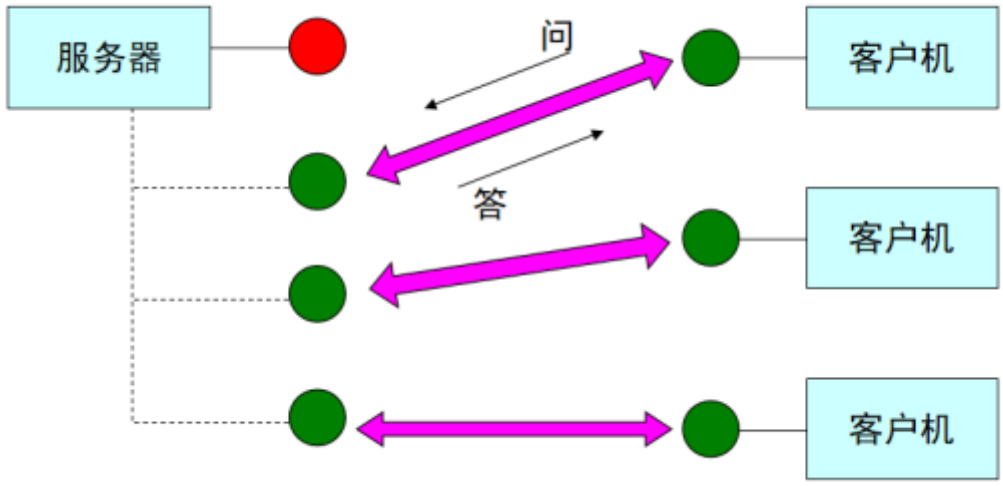
```
66          Socket socket = null;
67          try{
68              scanner = new
Scanner(System.in);
69              System.out.println("请输入:
server,<port> 或者: <ip>,<port>");
70              String str =
scanner.nextLine();
71              String[] arr = str.split(",");
72              if("server".equals(arr[0])){
73                  //启动服务端
74                  System.out.println("TCP
Server Listen at "+arr[1]+" .....");
75                  serverSocket = new
ServerSocket(Integer.parseInt(arr[1]));
76                  socket =
serverSocket.accept();
77                  System.out.println("连接成
功! ");
78
79              }else{
80                  //启动客户端
81                  socket = new
Socket(arr[0],Integer.parseInt(arr[1]));
82                  System.out.println("连接成
功! ");
83              }
84              //启动发送消息的线程
85              new
Send(socket,scanner).start();
86              //启动接收消息的线程
87              new Receive(socket).start();
```

```
88          }catch(Exception e){
89              e.printStackTrace();
90          }finally{
91              if(serverSocket != null){
92                  try {
93                      serverSocket.close();
94                  } catch (IOException e) {
95                      e.printStackTrace();
96                  }
97              }
98          }
99      }
100 }
```

一对多应用

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。



一对多应答型服务器

```
/**
 * 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();
    }

    /**
     * 将从客户端读取到的消息写回给客户端
     */
    private void msg(){
        try(BufferedReader br = new
BufferedReader(new
InputStreamReader(this.socket.getInputStream
()));
            PrintWriter pw = new
PrintWriter(this.socket.getOutputStream())){

            while(true){
                pw.println(br.readLine()+"
[ok]");
```

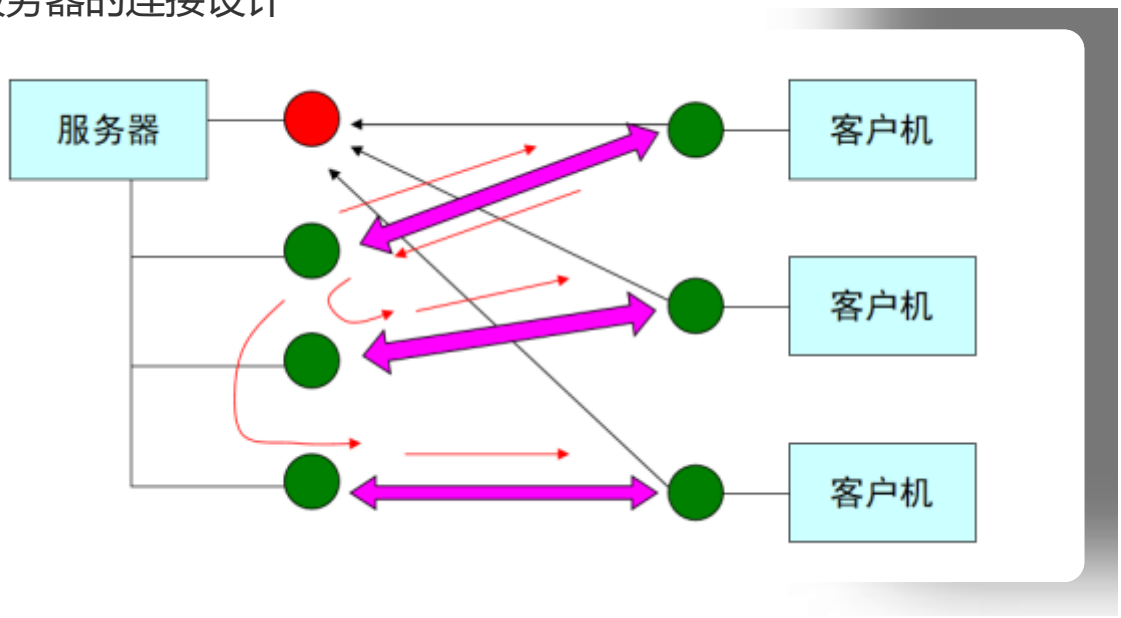
```
                pw.flush();
            }
        }catch(Exception e){
            e.printStackTrace();

 System.out.println(this.socket.getInetAddress()+" 断线了！");
        }
    }
}
public class EchoServer {
    public static void main(String[] args) {
        try(ServerSocket serverSocket = new ServerSocket(8888)){
            //等待多客户端连接
            while(true){
                Socket socket = serverSocket.accept();
                new Msg(socket).start();
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

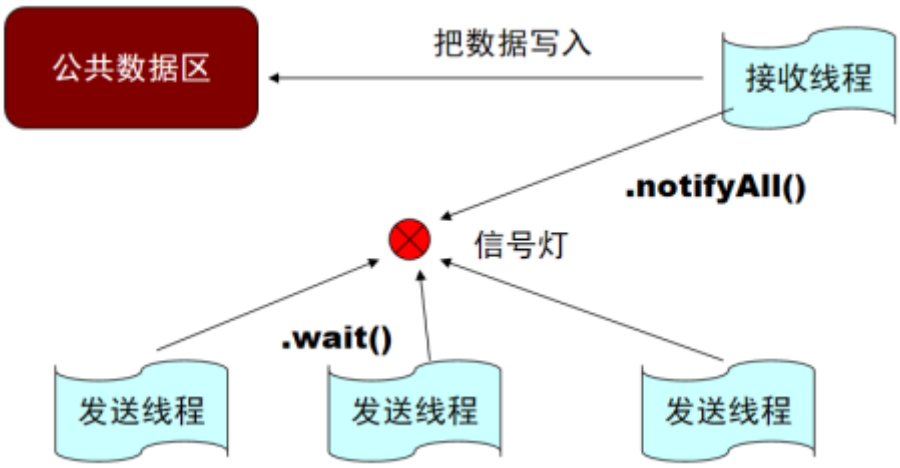
一对多聊天服务器

服务器设计

1. 服务器的连接设计



2. 服务器的线程设计



创建一对多聊天服务应用

```
1 /**
2  * 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(){

        try(BufferedReader 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){
```

```
30              e.printStackTrace();
31          }
32      }
33  }
34  /**
35   * 向客户端发送消息的线程类
36   */
37  class ChatSend extends Thread{
38      private Socket socket;
39      public ChatSend(Socket socket){
40          this.socket = socket;
41      }
42      @Override
43      public void run() {
44          this.sendMsg();
45      }
46      /**
47       * 将公共数据区的消息发送给客户端
48       */
49      private void sendMsg(){
50
51          try(PrintWriter  pw = new
    PrintWriter(this.socket.getOutputStream())){
52
53              while(true){
54                  synchronized ("abc"){
55                      //让发送消息的线程处于等待状
    态
56                      "abc".wait();
57                      //将公共数据区中的消息发送给
    客户端
```

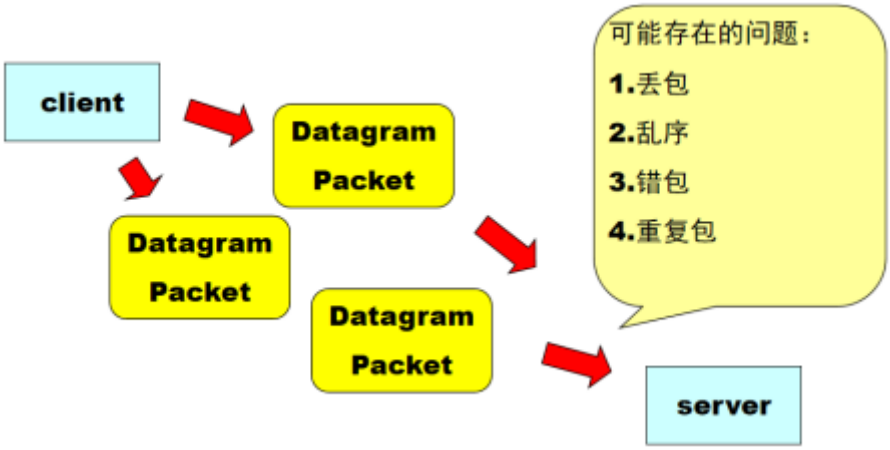
```
58
 pw.println(ChatRoomServer.buf);
59                      pw.flush();
60                  }
61              }
62          }catch(Exception e){
63              e.printStackTrace();
64          }
65      }
66  }
67  public class ChatRoomServer {
68      //定义公共数据区
69      public static String buf;
70      public static void main(String[] args) {
71          System.out.println("Chat Server
    Version 1.0");
72          System.out.println("Listen at
    8888.....");
73          try(ServerSocket serverSocket = new
    ServerSocket(8888)){
74
75              while(true){
76                  Socket socket =
    serverSocket.accept();
77                  System.out.println("连接
    到："+socket.getInetAddress());
78                  new
    ChatReceive(socket).start();
79                  new
    ChatSend(socket).start();
80              }
81          }catch(Exception e){
```

```
82                  e.printStackTrace();
83              }
84          }
85      }
```

UDP通信的实现和项目案例

UDP通信实现原理

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。 数据报包用来实现封包的功能。

常用方法：

方法名	使用说明
DatagramPacket(byte[] buf, int length)	构造数据报包，用来接收长度为 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

当服务器要向客户端发送数据时，需要在服务器端产生一个DatagramSocket对象，在客户端产生一个DatagramSocket对象。服务器端的DatagramSocket将DatagramPacket发送到网络上，然后被客户端的DatagramSocket接收。

DatagramSocket有两种常用的构造函数。一种是无需任何参数的，常用于客户端；另一种需要指定端口，常用于服务器端。如下所示：

1. DatagramSocket()：构造数据报套接字并将其绑定到本地主机上任何可用的端口。
2. 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。

常用方法：

方法名	使用说明
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. 创建客户端的DatagramSocket，创建时，定义客户端的监听端口。

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

UDP通信入门案例

创建服务端

```
public class UDPServer {
    public static void main(String[] args) {
        //创建服务端接收数据的DatagramSocket对象
        try(DatagramSocket 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();
        }
    }
}
```

创建客户端

```
1  public class UDPClient {
2      public static void main(String[] args) {
3          //创建数据发送对象 DatagramSocket,需要指定消息的发送端口
4          try(DatagramSocket ds = new DatagramSocket(8888)) {
5
6              //消息需要进行类型转换，转换成字节数据类型。
7              byte[] b = "百战程序员".getBytes();
8
9              //创建数据报包装对象DatagramPacket
10             DatagramPacket dp = new DatagramPacket(b, b.length, new InetSocketAddress("127.0.0.1", 9999));
11
12             //发送消息
13             ds.send(dp);
14         }catch(Exception e){
15             e.printStackTrace();
16         }
17     }
18 }
```

传递基本数据类型

创建服务端

```
public class BasicTypeUDPServer {
    public static void main(String[] args) {
        try(DatagramSocket datagramSocket = new DatagramSocket(9999)){
            byte[] buf = new byte[1024];
            DatagramPacket dp = new DatagramPacket(buf,buf.length);

            datagramSocket.receive(dp);

            //实现数据类型转换
            try(DataInputStream dis = new DataInputStream(new ByteArrayInputStream(dp.getData()))){
                //通过基本数据数据流对象获取传递的数据

 System.out.println(dis.readLong());
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

创建客户端

```
public class BasicTypeClient {
    public static void main(String[] args) {
        long n = 2000L;
        try(DatagramSocket datagramSocket = new DatagramSocket(9000);
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            DataOutputStream 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.send(dp);
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

传递自定义对象类型

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 +
                '}';
    }
}
```

创建服务端

```
public class ObjectTypeServer {
    public static void main(String[] args) {
        try(DatagramSocket 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
            try(ObjectInputStream objectInputStream = new ObjectInputStream(new ByteArrayInputStream(dp.getData()))){

             System.out.println(objectInputStream.readObject());
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

创建客户端

```
public class ObjectTypeClient {
    public static void main(String[] args) {
        try(DatagramSocket datagramSocket = new DatagramSocket(8888);
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(bos)){

            Person p = new Person();
            p.setName("Oldlu");
            p.setAge(18);


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

            DatagramPacket dp = new DatagramPacket(arr,arr.length,new InetSocketAddress("127.0.0.1",9999));

            datagramSocket.send(dp);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

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)。