南京大学
NANJING UNIVERSITY

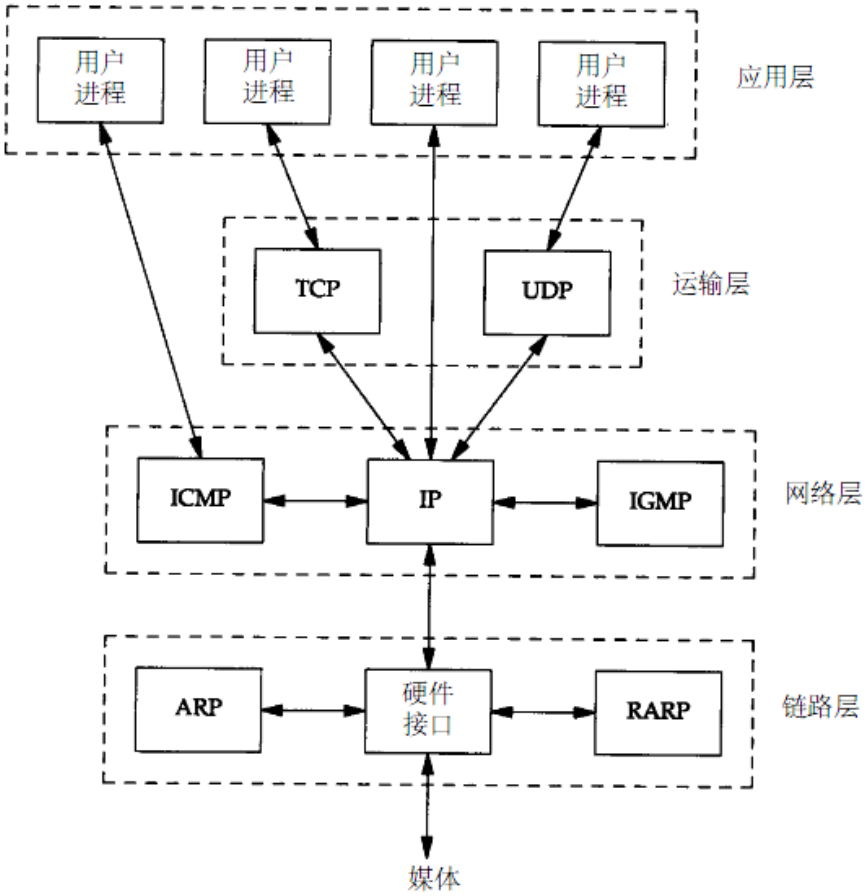
# Socket 编程


# 总览

- Socket 简介
- Socket 常用函数
- Java Socket API
  - 套接字地址的表示
  - 域名解析
  - 套接字的表示
  - 使用套接字传输数据

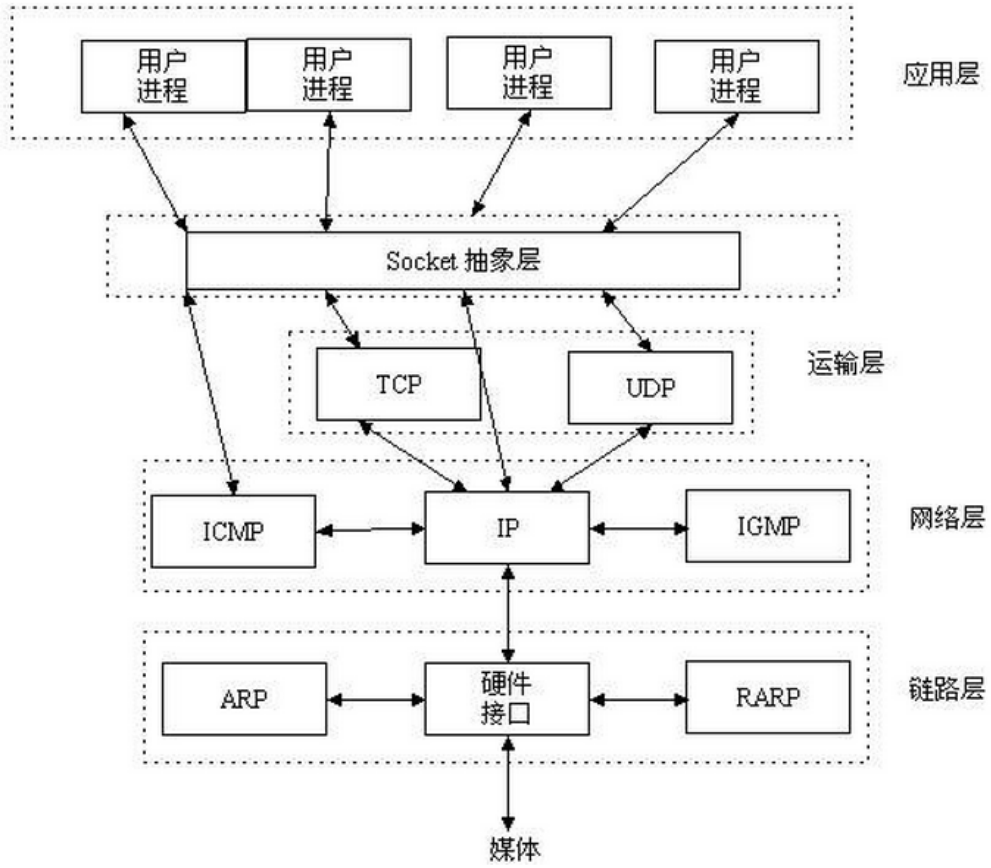

# Socket 简介

# 简介




图片来源于《tcp/ip协议详解卷一》

# 简介




图片来源于《tcp/ip协议详解卷一》

# 简介

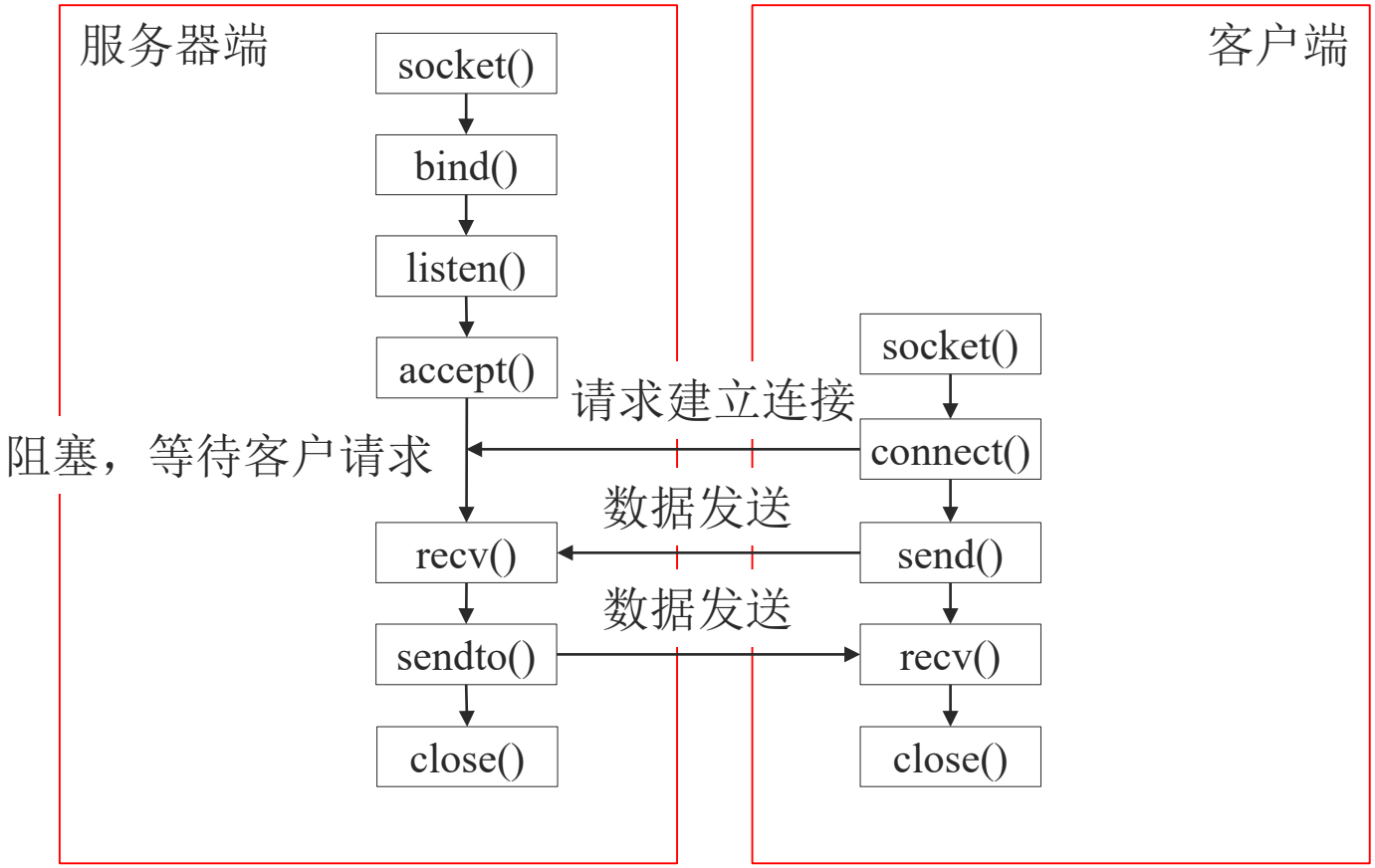
A major issue is how to refer to processes in a foreign HOST. Each HOST has some internal naming scheme, but these various schemes often are incompatible. Since it is not practical to impose a common internal process naming scheme, an intermediate name space was created with a separate portion of the name space given to each HOST. It is left to each HOST to map internal process identifiers into its name space. **The elements of the name space are called sockets. A socket forms one end of a connection, and a connection is fully specified by a pair of sockets.**

---IETF RFC33（1970）

# 简介

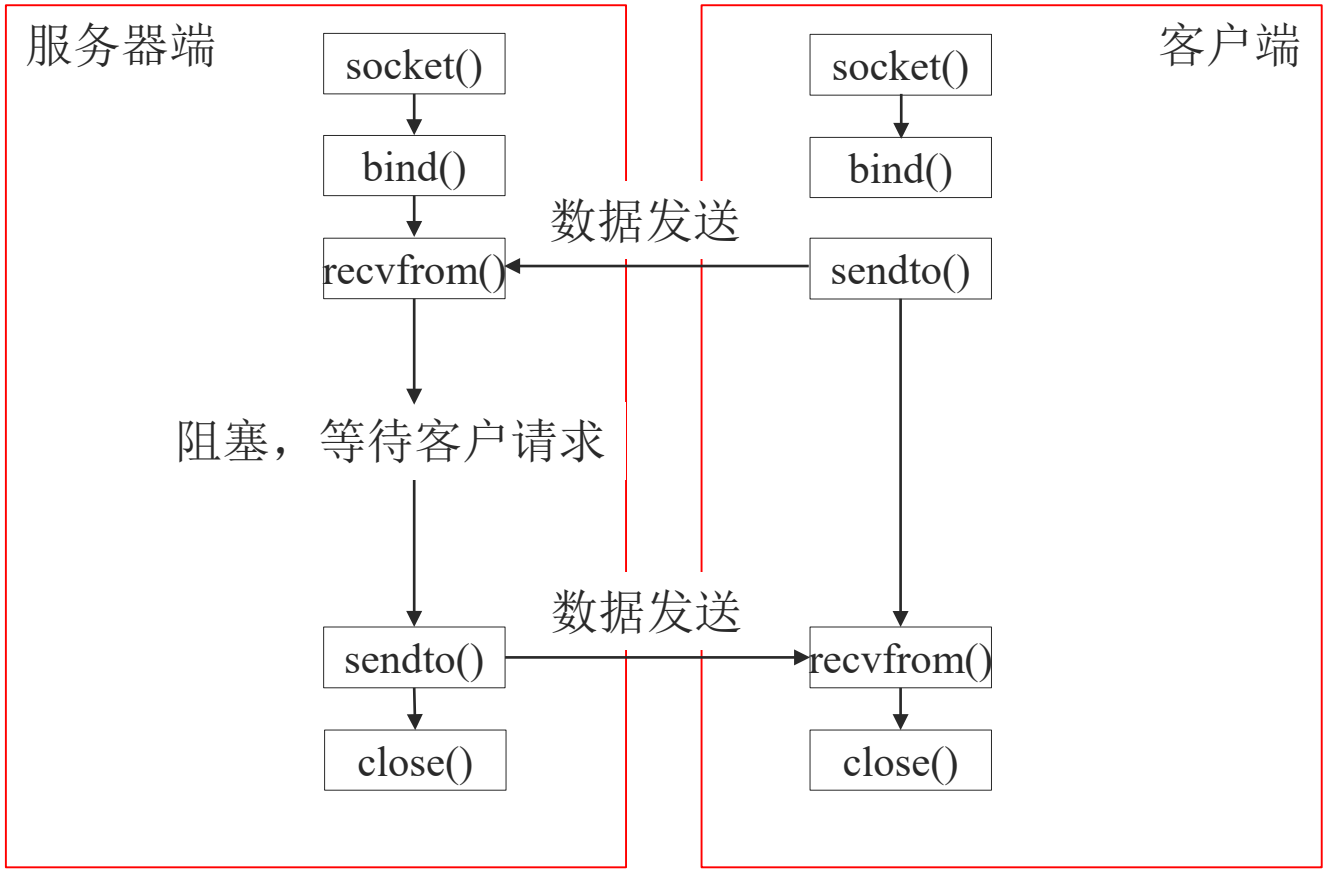
- 三类 Socket
  - 流套接字：
    - 主要用于 TCP 协议
    - 提供了双向的、有序的、无重复的、无记录边界的数据传输服务
  - 数据报套接字：
    - 主要用于 UDP 协议
    - 提供了双向的、无序的、有重复的、有记录边界的数据传输服务
  - 原始套接字：
    - 主要用于访问底层协议，如 IP、ICMP 和 IGMP 等协议
    - 原始套接字可保存 IP 包中的完整 IP 头部

- 面向连接的客户/服务器时序图



# 简介

- 无连接的客户/服务器时序图



# Socket 常用函数

# 常用函数 – socket()

- **int socket(int protofamily, int so_type, int protocol);**
  - protofamily – 指协议族，常见值有：
    - AF_INET：指定so_pcb中的地址要采用ipv4地址类型
    - AF_INET6：指定so_pcb中的地址要采用ipv6地址类型
    - AF_LOCAL/AF_UNIX:指定so_pcb中的地址要使用绝对路径名
  - so_type – 指定socket类型，常见值有：
    - SOCK_STREAM：基于TCP，数据传输比较有保障
    - SOCK_DGRAM：基于UDP，专门用于局域网
  - protocol – 指定具体的协议，常见值有：
    - IPPROTO_TCP：TCP协议
    - IPPROTO_UDP：UDP协议
    - 0：若指定为0，表示由内核根据so_type指定默认的通信协议

**注意：并不是上面的type和protocol可以随意组合，如SOCK_STREAM不可以和IPPROTO_UDP组合**

# 常用函数 – bind()

- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - sockfd

    即socket描述字，它是通过socket()函数创建了，唯一标识一个socket; bind()函数就是将给这个描述字绑定一个名字
  - addr

    一个const struct sockaddr *指针，指向要绑定给sockfd的协议地址。这个地址结构根据地址创建socket时的地址协议族的不同而不同
  - addrlen

    地址的长度

# 常用函数 – listen()、connect()

- `int listen(int sockfd, int backlog);`
- `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`

- 如果作为一个服务器，在调用socket()、bind()之后就会调用listen()来监听这个socket，如果客户端这时调用connect()发出连接请求，服务器端就会接收到这个请求。

- listen函数的第一个参数即为要监听的socket描述字，第二个参数为相应socket可以排队的最大连接个数。socket()函数创建的socket默认是一个主动类型的，listen函数将socket变为被动类型的，等待客户的连接请求。

- connect函数的第一个参数即为客户端的socket描述字，第二参数为服务器的socket地址，第三个参数为socket地址的长度。客户端通过调用connect函数来建立与TCP服务器的连接。

# 常用函数 – accept()

- `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`

- accept函数的第一个参数为服务器的socket描述字，第二个参数为指向struct sockaddr *的指针，用于返回客户端的协议地址，第三个参数为协议地址的长度。如果accpet成功，那么其返回值是由内核自动生成的一个全新的描述字，代表与返回客户的TCP连接。

- TCP服务器端依次调用socket()、bind()、listen()之后，就会监听指定的socket地址了。TCP客户端依次调用socket()、connect()之后就想TCP服务器发送了一个连接请求。TCP服务器监听到这个请求之后，就会调用accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络I/O操作了，即类同于普通文件的读写I/O操作。

# 常用函数 – read()、write()等

- 网络I/O操作有下面几组：
  - read()/write()
  - recv()/send()
  - readv()/writev()
  - recvmsg()/sendmsg()
  - recvfrom()/sendto()
- 负责相应的数据读写操作

# Linux的阻塞与非阻塞IO

- Linux文件系统：
  - Linux系统有“一切皆文件”的思想，在调用socket()创建一个套接字后，会获得一个表示该套接字的文件描述符fd
  - 文件描述符是一个非负整数，对应进程内的文件描述表中的指针，是进程内文件的唯一索引
- int fcntl(int fd, int cmd, ... /* arg */ );
  - fcntl是一个用于改变文件属性的系统调用，该系统调用的一个重要功能为改变文件的阻塞或非阻塞特性
  - 创建的套接字默认是阻塞模式
  - 以下语句将fd对应的文件或套接字设为非阻塞模式：

```
int flag = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_GETFL, flag|O_NONBLOCK)
```

# Linux的阻塞与非阻塞IO（续）

- ssize_t read(int fd, void *buf, size_t count);
  - read系统调用从fd对应的文件中，**尝试**读取count字节的数据，写入buf指针指向的缓冲区
  - read系统调用的返回值：
    - >0:返回值代表成功读取的字节数，这个值可能小于count但不会大于count
    - 0:代表遇到了文件尾EOF，如果是普通文件代表该文件被读完，如果是套接字，一般说明对方close了该套接字，该行为会向对方发送一个EOF表示结束
    - -1:代表发生错误，错误码被设置到errno中，errno是Linux的一个全局变量，随时可以调用
  - 当一个套接字连接建立，但对方没发送任何数据时：
    - 阻塞模式的read，会长时间阻塞线程，该线程无法进行别的工作，直到有数据可读，或对方关闭套接字获得EOF
    - 非阻塞模式的read，会在无数据可读时返回-1，errno被设为EAGAIN
  - 非阻塞模式的表现与特性被广泛用于多路复用
  - 其余常见的errno包括：EFAULT代表写入buf访问到非法内存，EINTR代表read被其他信号中端，代表各类的错误EINVAL、EIO等


# 思考

- 如何使用read调用，刚好读取指定字节长度的数据？

# 阻塞模式示例

```
ssize_t ReadNBytes(int sockfd, void *buf, size_t len, int flags) {
    size_t nleft = len;     // nleft代表离读取到len字节的目标还剩多少字节
    ssize_t nread;
    auto *bufp = static_cast<unsigned char *>(buf);
    while (nleft > 0) {
        if ((nread = read(sockfd, bufp, nleft, flags)) < 0)
            return -1;  // 阻塞模式下，返回-1对应出错
        else if (nread == 0)
           break;
        nleft -= nread;
        bufp += nread;
    }
    return (len - nleft);
}
```

# 非阻塞模式示例

```
ssize_t ReadNBytes(int sockfd, void *buf, size_t len, int flags) {
    size_t nleft = len;     // nleft代表离读取到len字节的目标还剩多少字节
    ssize_t nread;
    auto *bufp = static_cast<unsigned char *>(buf);
    while (nleft > 0) {
        if ((nread = read(sockfd, bufp, nleft, flags)) < 0) {
            if (errno == EAGAIN)
                continue;
            else
                return -1;
        } // 非阻塞模式下，返回值为-1时，若EAGAIN，说明是无数据可读，但不进入阻塞，因此重新尝试读
        else if (nread == 0)
           break;
        nleft -= nread;
        bufp += nread;
    }
    return (len - nleft);
}
```

# 常用函数 – close()

- **`int close(int fd)`**

- 在服务器与客户端建立连接之后，会进行一些读写操作，完成了读写操作就要关闭相应的socket描述字，好比操作完打开的文件要调用fclose关闭打开的文件。

**注意：close操作只是使相应socket描述字的引用计数-1，只有当引用计数为0的时候，才会触发TCP客户端向服务器发送终止连接请求。**

# Java Socket API

# Java Socket API

- Java Socket相关类

| 套接字地址域名解析 | InetAddress |
| --- | --- |
| TCP套接字 | Socket<br>ServerSocket |
| 套接字传输数据 | InputStream<br>OutputStream |
| UDP套接字 | DatagramSocket<br>DatagramPacket |

# 套接字地址/域名解析

- Java使用InetAddress表示IP地址
  - 定义于java.net包下
  - 既可以表示IPv4的地址，也可以表示IPv6的地址
- 在Java的Socket API中，往往将一个InetAddress对象和一个端口号一起使用作为套接字地址
- 通过InetAddress的静态方法getByName可以将一个IP地址或域名转换为InetAddress对象
  - 当对应域名不存在或无法解析时，会抛出java.net.UnknownHostException异常，需要对其进行处理

# 套接字地址/域名解析

```
import java.net.*;

class Lookup {
  public static void main(String[] args) {
    try {
      InetAddress a = InetAddress.getByName(args[0]);
      System.out.println(args[0] + ":" + a.getHostAddress());
    } catch (UnknownHostException e) {
      System.out.println("No address found for " + args[0]);
    }
  }
}
```

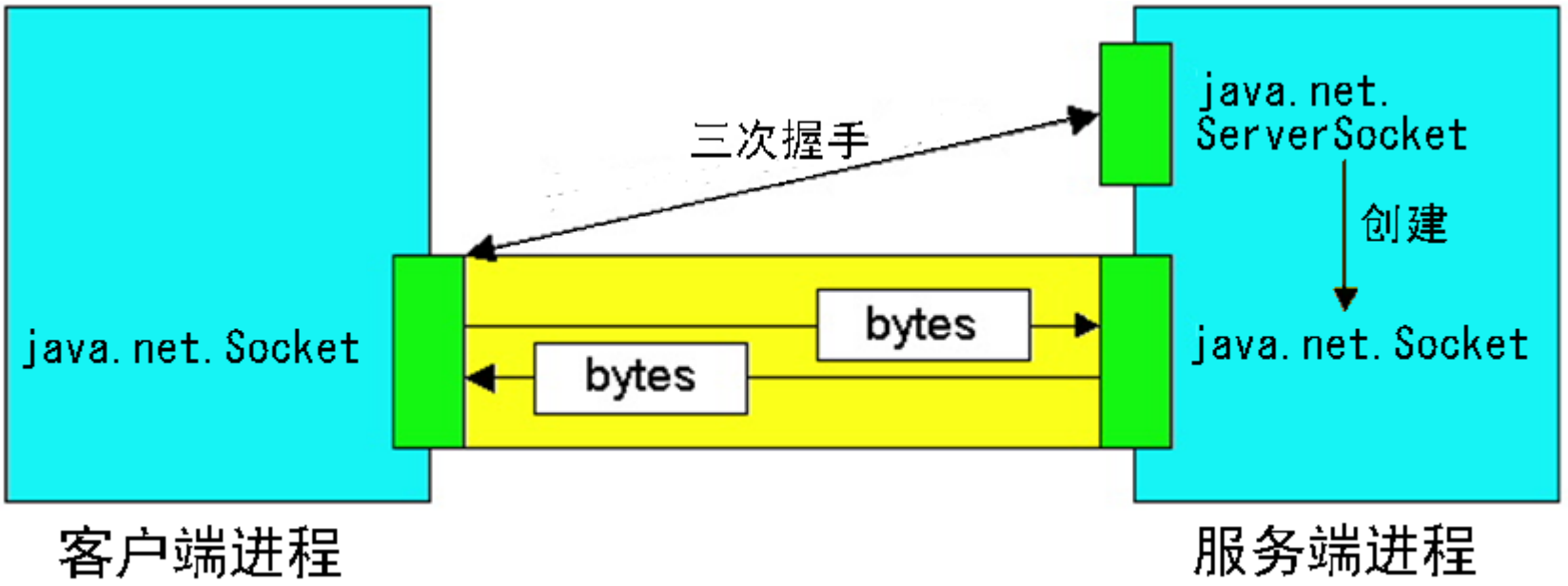
```
> java Lookup software.nju.edu.cn
software.nju.edu.cn:219.219.120.45
> java Lookup 127.0.0.1
127.0.0.1:127.0.0.1
```

# Java Socket API

- Java Socket相关类

| | |
|---|---|
| 套接字地址域名解析 | InetAddress |
| **TCP套接字** | **Socket**<br>**ServerSocket** |
| 套接字传输数据 | InputStream<br>OutputStream |
| UDP套接字 | DatagramSocket<br>DatagramPacket |

# TCP套接字


三次握手
java.net.
ServerSocket
创建
bytes
bytes
java.net.Socket
java.net.Socket
客户端进程
服务端进程

# TCP套接字 – ServerSocket类

- ServerSocket类表示服务端创建的等待客户端来连接的TCP套接字（称为被动套接字）
- 绑定ServerSocket到本地端口并监听
  - new ServerSocket(int port) // backlog=50
  - new ServerSocket(int port, int backlog)
  - new ServerSocket(int port, int backlog, InetAddress bindAddr)
  - 当绑定失败（譬如端口被占用时）均抛出java.io.IOException
- 通过accept调用获取一个完成三次握手的TCP连接
  - Socket accept()
  - 创建时的backlog参数表示允许完成三次握手但没被accept调用获取到的TCP连接个数，超出backlog的连接将会被拒绝

# TCP套接字 –Socket类

- Socket类表示一个建立好的TCP连接，它可以由服务端通过主动连接创建，也可以被ServerSocket通过accept调用创建（称为主动套接字）
- 客户端主动连接
  - new Socket(InetAddress addr, int port)
  - 当目标机器不可达、连接被重置或拒绝等情况时，会抛出java.io.IOException异常
- 与远程机器通信
  - InputStream getInputStream()—从远程机器读数据
  - OutputStream getOutputStream()—向远程机器写数据
  - void close()—关闭该套接字连接

# Java Socket API

- Java Socket相关类

| | |
|---|---|
| 套接字地址域名解析 | InetAddress |
| TCP套接字 | Socket<br>ServerSocket |
| **套接字传输数据** | **InputStream**<br>**OutputStream** |
| UDP套接字 | DatagramSocket<br>DatagramPacket |

# 读取数据

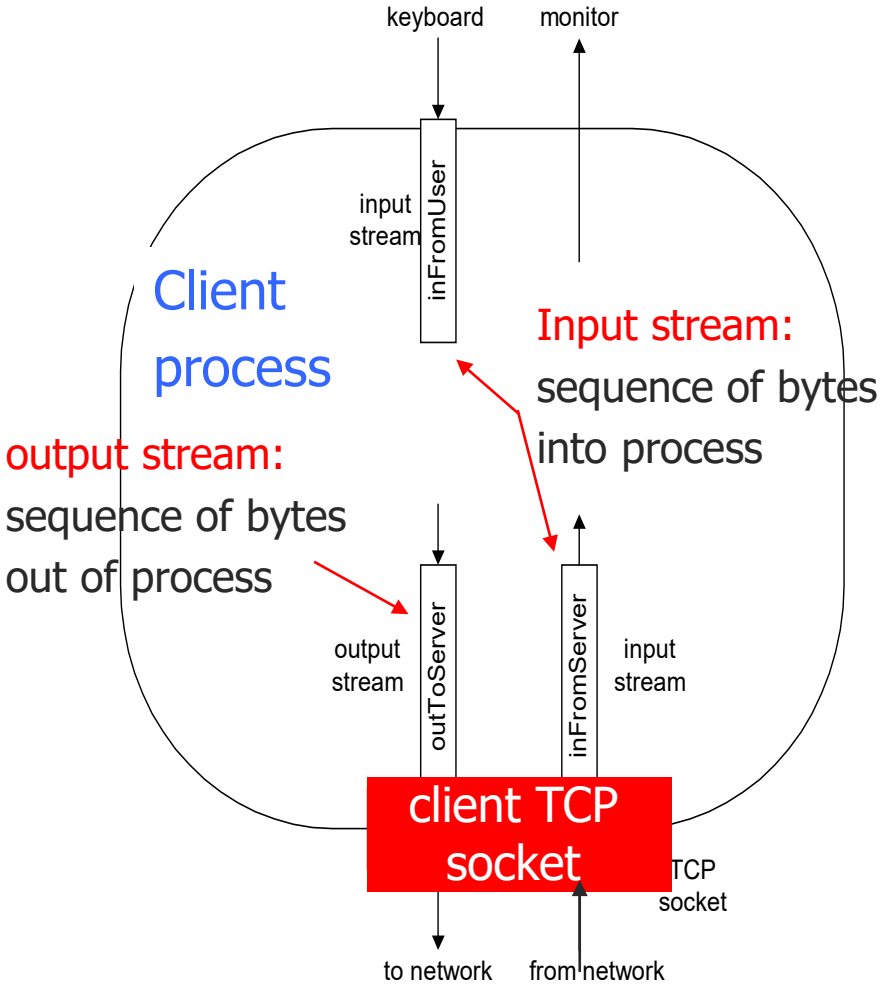
- InputStream类为Java提供的输入流抽象
  - 定义于java.io包下
  - 输入流有流终止和传输错误两种情况，而一般而言前者不抛出异常，后者抛出java.io.IOException
- 通过read方法进行读取
  - abstract int read(): 从输入流读取单个字节，当读取成功时，返回0-255的整数，当流终止时，返回-1
  - int read(byte[] b): 从输入流读取最多b.length个字节，并返回读取到b的元素的个数，当流终止时，返回-1
  - int read(byte[] b, int off, int len): 从输入流读取最多len个字节到b中以b[off]开头的存储空间中，当流终止时，返回-1
- 通过实现InputStream类，可以实现新的输入源，如FileInputStream，ByteBufferInputStream等
- 通过包装InputStream类，可以扩充和简化InputStream的API，如DataInputStream，BufferedInputStream等

# 写出数据

- OutputStream类为Java提供的输出流抽象
  - 定义于java.io包下
  - 由于数据流的流终止由调用者决定，因此只存在传输错误一种情况，会抛出java.io.IOException异常
- 通过write方法进行写出
  - abstract void write(int b)——写出单个字节到输出流，只有低8位有效
  - void write(byte[] b)——将b中b.length个字节写出到输出流
  - void write(byte[] b, int off, int len)——将b中b[off]开头的len个字节写出到输出流
- 通过close方法关闭/终止输出流
  - 子类通过重写close方法决定close时的行为
- 通过实现OutputStream类，可以实现新的输出源，如FileOutputStream，ByteBufferOutputStream等
- 通过包装OutputStream类，可以扩充和简化OutputStream的API，如DataOutputStream，BufferedOutputStream等

# TCP套接字编程示例

- 客户端从标准输入流读取一行字符串，并将其写出到服务端
- 服务端读取客户端的输入数据，将其转换为大写，并传回给客户端
- 客户端从服务端读取数据，并将转换后的字符串输出到标准输出流，回显给用户



# TCPClient.java

```
import java.io.*;
import java.net.*;

class TCPClient {
    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
        BufferedReader inFromServer = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));

        sentence = inFromUser.readLine();
        outToServer.writeBytes(sentence + '\n');

        modifiedSentence = inFromServer.readLine();
        System.out.println("FROM SERVER: " + modifiedSentence);

        clientSocket.close();
    }
}
```

# TCPServer.java

```
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(6789);
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient = new BufferedReader(
                new InputStreamReader(connectionSocket.getInputStream()));
            DataOutputStream outToClient = new DataOutputStream(
                connectionSocket.getOutputStream());

            clientSentence = inFromClient.readLine();

            capitalizedSentence = clientSentence.toUpperCase() + '\n';

            outToClient.writeBytes(capitalizedSentence);
        }
    }
}
```

# Java Socket API

- Java Socket相关类

| | |
|---|---|
| 套接字地址域名解析 | InetAddress |
| TCP套接字 | Socket<br>ServerSocket |
| 套接字传输数据 | InputStream<br>OutputStream |
| **UDP套接字** | **DatagramSocket**<br>**DatagramPacket** |

# UDP套接字

- 回顾UDP的特点
  - 通过套接字为网络中的其他设备提供服务，即知道目标机器IP地址和端口即可向目标机器的对应进程发送UDP报文
  - 基于分组交换进行工作
  - 无需目标机器无需提供被动套接字和三次握手建立连接
  - 如果分组能到达目标机器，不保证分组的到达次序
  - 分组可能会遗失和被丢弃
- 回顾传输层UDP报文的格式
  - 源端口，目标端口
  - 载荷长度
  - 校验和

# DatagramSocket类

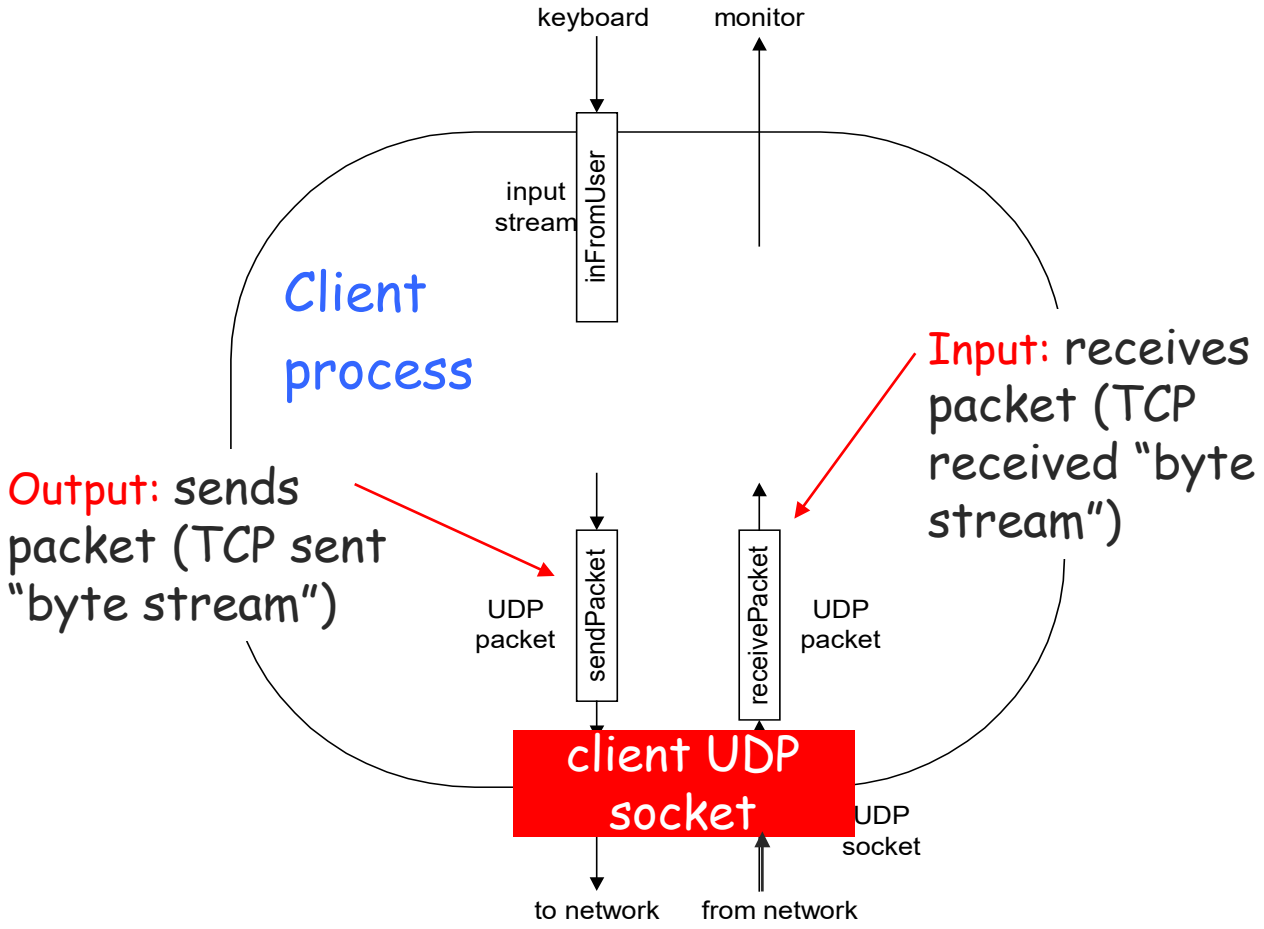
- DatagramSocket类表示一个UDP套接字
- 绑定UDP套接字到本地端口
  - new DatagramSocket()——绑定套接字到任意端口
  - new DatagramSocket(int port)——绑定套接字到指定端口
  - new DatagramSocket(int port, InetAddress addr)——绑定套接字到指定IP地址和端口
  - 当绑定失败时，均会抛出java.net.SocketException异常
  - 通过close()关闭当前UDP套接字，释放其占用的端口
- 发送和接收数据报文
  - send(DatagramPacket p)——发送UDP报文
  - receive(DatagramPacket p)——接收UDP报文，会一直阻塞接收到报文，或者在设置了超时时间后，抛出java.net.SocketTimeoutException
  - setSoTimeout(int timeout)——设置接收UDP报文的超时时间

# DatagramPacket类

- DatagramPacket类封装了一个由UDP套接字传输的数据包
  - 包含一个缓冲区用于存放UDP报文数据
  - 包含了目标机器的IP地址和端口，当DatagramSocket的connect方法被调用后，DatagramPacket中的目标IP和端口将被忽略，直到DatagramSocket的disconnect方法被调用

- 创建DatagramPacket的载荷
  - new DatagramPacket(byte[] buf, int len)——创建一个用于接收的DatagramPacket类，超出len的UDP数据将被截断
  - new DatagramPacket(byte[] buf, int len)——创建一个用于发送的DatagramPacket类

# UDP套接字编程示例


keyboard
monitor
input stream
inFromUser
Client process
Input: receives packet (TCP received "byte stream")
Output: sends packet (TCP sent "byte stream")
UDP packet
sendPacket
receivePacket
UDP packet
client UDP socket
UDP socket
to network
from network

# UDPClient.java

```
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("hostname");

        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];

        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
        clientSocket.send(sendPacket);

        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);

        String modifiedSentence = new String(receivePacket.getData());
        System.out.println("FROM SERVER:" + modifiedSentence);
        clientSocket.close();
    }
}
```

# UDPServer.java

```
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);

        byte[] receiveData = new byte[1024];
        byte[] sendData  = new byte[1024];

        while(true) {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);

            String sentence = new String(receivePacket.getData());
            InetAddress IPAddress = receivePacket.getAddress();
            int port = receivePacket.getPort();

            String capitalizedSentence = sentence.toUpperCase();
            sendData = capitalizedSentence.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
            serverSocket.send(sendPacket);
        }
    }
}
```