

Socket编程API

网络程序设计接口主要类型

理解应用编程接口API

几种典型的应用编程接口

Socket编程-Socket API概述

Socket API

如何标识socket？

对内的Socket抽象

socket地址结构

Socket编程-Socket API函数

Socket API函数（以WinSock为例）

WSAStartup

WSACleanup

socket

Socket面向TCP/IP的服务类型

closesocket（linux版为close）

bind（服务器）

listen（服务器）

connect（客户端，tcp/udp差异）

accept（服务器，tcp）

send, sendto（服务器客户机都用）

recv, recvfrom（服务器客户机都用）

setsockopt, getsockopt（不具体讲了）

Socket API函数小结

win独占

多平台通用

关于网络字节顺序

网络应用的Socket API(TCP)调用基本流程

Socket编程-客户端软件设计

解析服务器IP地址

解析服务器（熟知）端口号

解析协议号

TCP客户端软件流程

UDP客户端软件流程

客户端软件的实现- connectsock()

客户端软件的实现-UDP客户端

客户端软件的实现-TCP客户端

客户端软件的实现-异常处理

例1：访问DAYTIME服务的客户端（TCP）

例2：访问DAYTIME服务的客户端（UDP）

Socket编程-服务器软件设计

4种类型基本服务器

循环无连接服务器基本流程

数据发送

获取客户端点地址

循环面向连接服务器基本流程

并发无连接服务器基本流程

并发面向连接服务器基本流程

服务器的实现

服务器的实现-passivesock()

服务器的实现-passiveUDP()

服务器的实现-passiveTCP()
例1：无连接循环DAYTIME服务器
例2：面向连接并发DAYTIME服务器

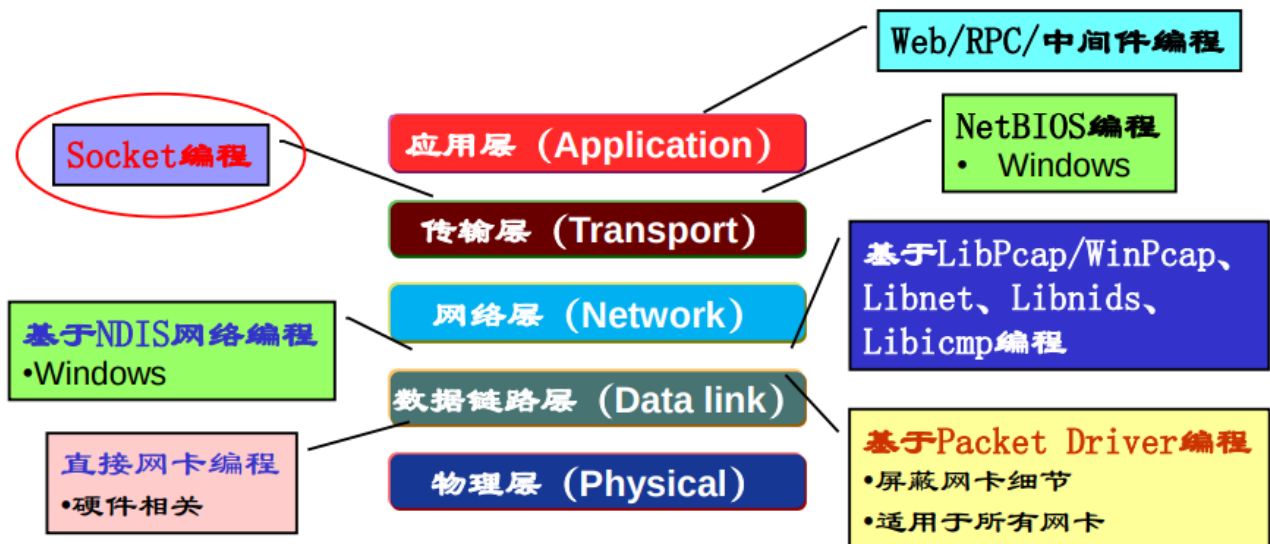
Socket编程API

网络程序设计接口主要类型

开发网络应用程序关联的API类型。

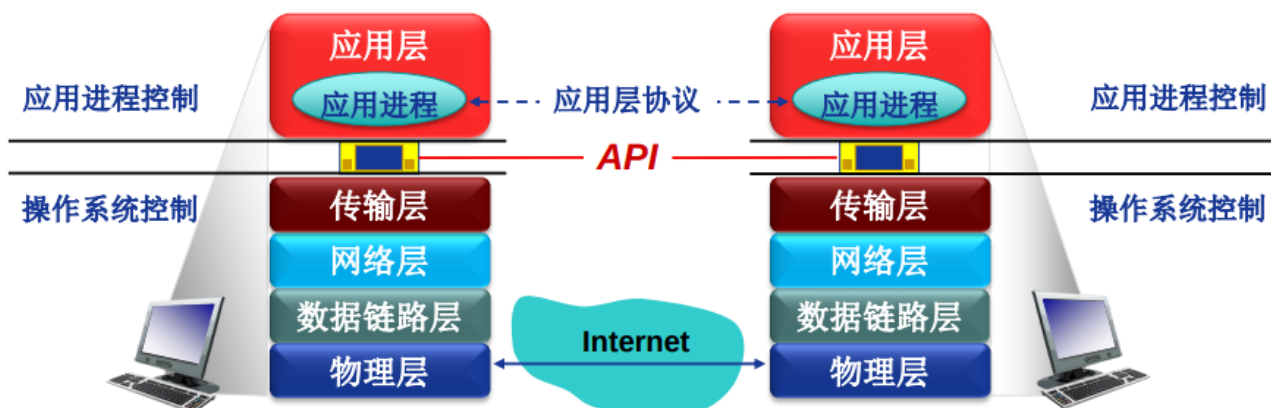
按5层结构观察，除物理层外，所有层次，包括应用层本身，都能提供网络程序设计的api。

- 直接面向网卡编程-大部分不需要也难以掌握。 **网卡是数据链路层**
- 网卡之上的，数据链路层的编程，屏蔽网卡细节，适用所有网卡。
- 特定操作系统的开发api。
- 基于库的。
- socket：应用层、传输层之间的。 **属于传输层**



理解应用编程接口API

应用编程接口 API (Application Programming Interface)



应用编程接口API:就是应用进程的控制权和操作系统的控制权进行转换的一个系统调用接口。

- 应用层协议组构应用进程之间的逻辑连接。
- API通常是从传输层开始封装。

几种典型的应用编程接口

- Berkeley UNIX 操作系统定义了一种 API，称为**套接字接口**(socket interface)，简称套接字 (socket)。
- 微软公司在其操作系统中采用了套接字接口 API，形成了一个稍有不同的 API，并称之为Windows Socket Interface，**WINSOCK**。
- AT&T (美国电话电报公司) 为其 UNIX 系统 V 定义了一种 API，简称为 **TLI** (Transport Layer Interface)。
 - **UNIX**，一种计算机**操作系统**，具有多任务、多用户的特征。于1969年，在美国**AT&T**公司的**贝尔实验室**开发**类UNIX** (UNIX-like)

Socket编程-Socket API概述

Socket API

抽象通信机制。是一种门面模式，为应用层封装传输层协议，为应用层提供抽象链路。

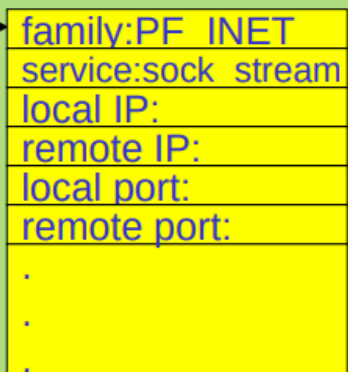
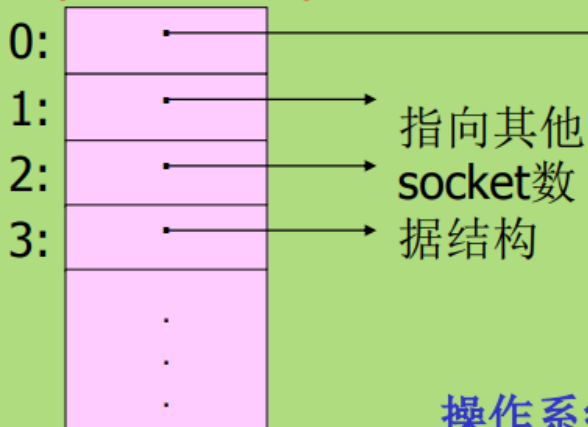
- 最初设计
 - 面向BSD UNIX-Berkley
 - 面向TCP/IP协议栈接口
- 目前
 - Internet网络应用最典型的API接口，事实上的工业标准
 - 绝大多数操作系统都支持
- 通信模型

- **由进程或者操作系统创建**

- 对外

- IP地址+端口号 (16位整数端口号, 0-65535, 即16位二进制无符号数)

- 类似于文件的抽象，像文件一样管理socket
- 当应用进程创建套接字时，**操作系统分配一个数据结构存储该套接字相关信息**
- 进程调用api通知操作系统创建套接字，该函数由操作系统返回套接字描述符给进程。
- 都是通过该socket描述符来引用、访问套接字。
- 每一个进程都管理一个socket描述符表，管理其创建的socket，这个表类似一个结构体指针数组，每个指针指向一个socket数据结构。
- **由系统使用来提供api。**



socket地址结构

❖ 已定义结构 `sockaddr_in`:

```
struct sockaddr_in
{
    u_char sin_len;           /*地址长度           */
    u_char sin_family;       /*地址族(TCP/IP: AF_INET) */
    u_short sin_port;        /*端口号           */
    struct in_addr sin_addr;  /*IP地址           */
    char sin_zero[8];        /*未用(置0)        */
}
```

- **sin : socket internet**
- IP地址、本地端口号这两个必需。
- socket提供多协议支持，不仅仅是TCP/IP
 - 地址族：表述所使用的传输层协议
 - AF_INET：TCP/IP使用的地址族
 - 只需知道，windows下tcpip要用的地址族是AF_INET就够了
 - sin_zero是为了让sockaddr与sockaddr_in两个数据结构保持大小相同而保留的空字节。
- 使用TCP/IP协议簇的网络应用程序声明端点地址变量时，使用结构sockaddr_in

sockaddr与sockaddr_in：

注释中标明了属性的含义及其字节大小，这两个结构体一样大，都是16个字节，而且都有family属性，不同的是：

sockaddr用其余14个字节来表示sa_data，而sockaddr_in把14个字节拆分成sin_port, sin_addr和sin_zero分别表示端口、ip地址。sin_zero用来填充字节使sockaddr_in和sockaddr保持一样大小。

sockaddr和sockaddr_in包含的数据都是一样的，但他们在使用上有区别：

程序员不应操作sockaddr，sockaddr是给操作系统用的

程序员应使用sockaddr_in来表示地址，sockaddr_in区分了地址和端口，使用更方便。

思考：为何这里要维护地址长度？

Socket编程-Socket API函数

Socket API函数（以WinSock为例）

winsock实现机制是动态连接库，所以要初始化、释放动态连接库，使用api开始和结束要分别调用

- WSASStartup（初始化Windows Sockets API）
- WSACleanup（释放所使用的WindowsSockets DLL）

- **wsa表示Windows Sockets API**

WSAStartup

使用Socket的应用程序在使用Socket之前**必须首先**调用WSAStartup函数

```
1 | int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSADATA);
```

- WORD wVersionRequested
 - 指明程序请求使用的WinSock版本，其中高位字节指明副版本、低位字节指明主版本
 - 十六进制整数，例如0x102表示2.1版（**WORD表示双字节**）
- LPWSADATA lpWSADATA
 - 返回实际的WinSock的版本信息，指向WSADATA结构的指针
- 例：使用2.1版本的WinSock的程序代码段

```
1 | int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSADATA);
2 | wVersionRequested = MAKEWORD( 2, 1 );
3 | err = WSAStartup( wVersionRequested, &wsaData );
```

WSACleanup

应用程序在完成对请求的Socket库的使用，最后要调用WSACleanup函数。

```
1 | int WSACleanup (void);
```

- 解除与Socket动态库的绑定。
- 释放Socket库所占用的系统资源。

下面的不带wsa的api是多系统通用的，上面的wsa api是win专用的

socket

创建套接字

```
1 | sd = socket(protofamily, type, proto);
```

- sd
 - 操作系统返回的套接字描述符，应用层使用该描述符操作、引用套接字。
- protofamily协议族（说明面向哪个协议族）
 - protofamily = PF_INET（TCP/IP）
- **type套接字类型**（每种协议族不同，下面的例子是）
 - type = SOCK_STREAM, SOCK_DGRAM or SOCK_RAW（TCP/IP）
- proto(协议号，访问的是哪种协议):0表示缺省，可以使用对应数字表示所选协议族和套接字类型的支持的协议号
- 例：创建一个流套接字的代码段

```

1 struct protoent *p;
2 p=getprotobyname("tcp");
3 SOCKET sd=socket(PF_INET, SOCK_STREAM, p->p_proto);

```

思考：为何这里用PF——INET？

Socket面向TCP/IP的服务类型

stream（传输流，使用tcp）：可靠、面向连接、字节流传输、点对点（一个连接只能连接两点）

dgram（datagram数据报，使用udp）：不可靠、无连接、数据报传输

raw（raw：生的，这里指不加传输层处理的原始套接字）

closesocket（linux版为close）

关闭一个描述符为sd的套接字。

```

1 int closesocket(SOCKET sd);

```

- 如果多个**进程**共享一个套接字，有计数。
 - 调用closesocket将把调用进程的描述符表中的引用删除，然后该socket的数据结构的reference counting减1，减至0才释放该socket数据结构空间。
- 一个进程中的多**线程**对一个套接字的使用无计数
 - 如果进程中的一个线程调用closesocket将一个套接字关闭，则该进程的描述符表中无该socket数据结构的引用，该进程中的其他线程也将不能访问该套接字。
 - **描述符表是一个进程一个，而不是一个线程一个！**
- 返回值
 - 0：成功
 - SOCKET_ERROR：失败

bind（服务器）

绑定（填写）套接字的本地端点地址（IP地址+**16进制**端口号）

```

1 int bind(sd, localaddr, addrlen);

```

- 参数:
 - 套接字描述符（sd）
 - 端点地址（localaddr）
 - 结构sockaddr_in
- 客户程序一般不必调用bind函数，因为这个工作一般是由操作系统来完成的。
- 服务器端需要调用
 - 绑定设置熟知端口号：80 25 等等
 - IP地址？服务器运行主机的ip地址可以吗？

- 绑定问题：服务器应该绑定哪个地址？
- 解决方案
 - 地址通配符：INADDR_ANY，把ip地址附成该值，表示主机上任意一个有效ip地址都是可以来访问该socket。
 - 服务器应该绑定INADDR_ANY：端口号，而是saddrln不具体IP：端口号
 - INADDR_ANY一般是0.0.0.0

思考：为什么要加地址长度？localaddr里面已经包含长度

listen（服务器）

置流套接字处于监听状态，listen只用于服务器端，仅面向tcp连接的流类型。

```
1 | int listen(sd, queue_size);
```

- 设置连接**请求缓存队列**大小（queue_size）
- 返回值：
 - 0：成功
 - SOCKET_ERROR：失败

connect（客户端，tcp/udp差异）

连接建立不等同数据请求发送，还需要send

```
1 | connect(sd, saddr, saddrln);
```

- 客户程序调用connect函数来使本地套接字（sd）与特定计算机的特定端口（saddr）的套接字（服务）进行连接
- 仅用于客户端
- 可用于TCP客户端也可以用于UDP客户端
 - 用于TCP客户端：建立TCP连接
 - 用于UDP客户端：只是简单的本地指定了服务器端点地址，在之后发送数据报和接受数据时使用该地址。

accept（服务器，tcp）

```
1 | newsock = accept(sd, caddr, caddrln);
```

- 服务程序调用accept函数从处于监听状态的流套接字sd的客户连接请求队列中取出排在**最前的一个**客户请求，并且**创建一个新的套接字**来与客户套接字创建连接通道。
- 注意这里接**受的是建立连接的请求，而不是对数据的请求**，后者由recv, recvfrom负责，数据由send, sendto发送。
- **仅用于TCP套接字**
- **仅用于服务器**

- 利用新创建的套接字
- 使用新套接字（newsock）与客户通信，why？
 - tcp是点对点的，socket2socket的，单对单的，如果不这么做，就不能并发的提供服务。这里把接受服务和提供服务的socket区分开了。

send, sendto（服务器客户机都用）

```
1 send(sd, *buf, len, flags);
2 sendto(sd, *buf, len, flags, destaddr, addrlen);
```

- send函数TCP套接字（客户与服务器）或调用了connect函数的UDP客户端套接字
- sendto函数用于UDP服务器端套接字与未调用connect函数的UDP客户端套接字

recv, recvfrom（服务器客户机都用）

```
1 recv(sd, *buffer, len, flags);
2 recvfrom(sd, *buf, len, flags, senderaddr, saddrlen);
```

- recv函数从TCP连接的另一端接收数据，或者从调用了connect函数的UDP客户端套接字接收服务器发来的数据。
- recvfrom函数用于从UDP服务器端套接字与未调用connect函数的UDP客户端套接字接收对端数据。

setsockopt, getsockopt（不具体讲了）

```
1 int setsockopt(int sd, int level, int optname, *optval, int optlen);
2 int getsockopt(int sd, int level, int optname, *optval, socklen_t *optlen);
```

- setsockopt()函数用来设置套接字sd的选项参数
- getsockopt()函数用于获取任意类型、任意状态套接口的选项当前值，并把结果存入optval

Socket API函数小结

win独占

- □ WSASStartup: 初始化socket库(仅对WinSock)
- □ WSACleanup: 清楚/终止socket库的使用 (仅对WinSock)

多平台通用

按照服务器、客户机、tcp、udp去理解

- □ socket: 创建套接字
- □ connect: “连接”远端服务器 (仅用于客户端)
- □ closesocket: 释放/关闭套接字
- □ bind: 绑定套接字的本地IP地址和端口号（通常客户端不需要）
- □ listen: 置服务器端TCP套接字为监听模式，并设置队列大小 (仅用于服务器端TCP套接字)
- □ accept: 接受/提取一个连接请求，创建新套接字，通过新套接 (仅用于服务器端的TCP套接字)
- □ recv: 接收数据（用于TCP套接字或连接模式的客户端UDP套接字）

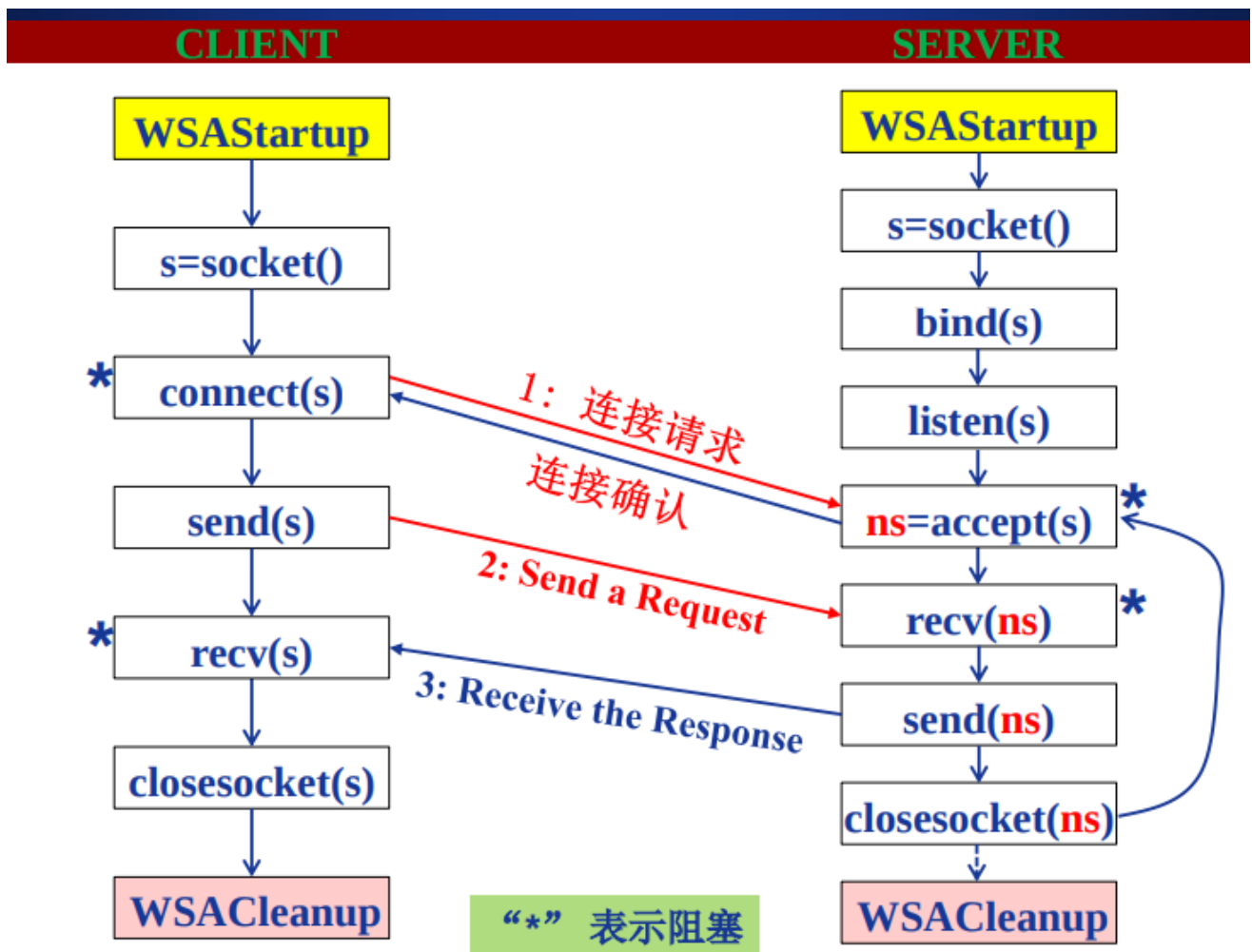
- recvfrom: 接收数据报（用于非连接模式的UDP套接字）
- send: 发送数据（用于TCP套接字或连接模式的客户端UDP套接字）
- sendto: 发送数据报（用于非连接模式的UDP套接字）
- setsockopt: 设置套接字选项参数（用的时候查）
- getsockopt: 获取套接字选项参数（用的时候查）

关于网络字节顺序

osi7层模型才有表示层来兼容字节顺序，5层中是没有的，需要协议辅助完成该功能。

- TCP/IP定义了标准的用于协议头中的二进制整数表示：网络字节顺序（network byte order）
- 某些Socket API函数的参数需要存储为网络字节顺序而不是本地字节顺序（如IP地址、端口号等）。
- 可以实现本地字节顺序与网络字节顺序间转换的函数
 - htons: 本地字节顺序 → 网络字节顺序(16bits) (host2net s)
 - ntohs: 网络字节顺序 → 本地字节顺序(16bits)
 - htonl: 本地字节顺序 → 网络字节顺序(32bits)
 - ntohl: 网络字节顺序 → 本地字节顺序(32bits)

网络应用的Socket API(TCP)调用基本流程



注意，这张图并不完整，注意区分tcp udp

listen并不阻塞，listen只是开启listen状态。

recv、accept是真正对连接的反馈需要循环开启，也是后续的操作的前提，所以要阻塞accept、recv。

阻塞过程，表示当函数未成功则一直等待。

左右两边都有2个阻塞函数。

ns : newsocket

注意两边close的区别，服务器只是关闭了ns。

Socket编程-客户端软件设计

socket服务器按上述流程要：选择服务器ip、端口号，网络字节转化，选择协议。

具体实现上要求：4位ip/域名：服务名转化为32位ip：端口号，协议名转化为服务号

解析服务器IP地址

- 客户端可能使用域名（如：study.163.com）或IP地址（如：123.58.180.121）标识服务器
- IP协议需要使用32位二进制IP地址
- 需要将域名或IP地址转换为32位IP地址
 - 函数inet_addr() 实现点分十进制IP地址到32位IP地址转换
 - 如果正确执行将返回一个无符号长整数型数。如果传入的字符串不是一个合法的IP地址，将返回INADDR_NONE。
 - 已经是网络字节顺序
 - 函数gethostbyname() 实现域名到32位IP地址转换
 - 返回一个指向结构hostent 的指针,该结构中包含32位ip地址
 - 已经是网络字节顺序

解析服务器（熟知）端口号

- 客户端可能不使用端口号而是使用服务名（如HTTP）标识服务器端口
- 需要将服务名转换为标准的熟知端口号
- 函数getservbyname()
 - 返回一个指向结构servent的指针，该结构包含端口号

解析协议号

- socket使用协议号来标识协议
- 客户端可能使用协议名（如：TCP）指定协议，需要将协议名转换为协议号（如：6）
- 函数getprotobyname() 实现协议名到协议号的转换
 - 返回一个指向结构protoent的指针

TCP客户端软件流程

1. 确定服务器IP地址与端口号
2. 创建套接字
3. 分配本地端点地址（IP地址+端口号）（显示的创建socket的同时，系统自动完成本地端点地址分配）
4. 连接服务器（套接字）
5. 遵循应用层协议进行通信
6. 关闭/释放连接

UDP客户端软件流程

1. 确定服务器IP地址与端口号
2. 创建套接字
3. 分配本地端点地址（IP地址+端口号）
4. 指定服务器端点地址，构造UDP数据报并发送（注意，这里并没有通知服务器，而是直接发送）
5. 遵循应用层协议进行通信
6. 关闭/释放套接字

客户端软件的实现- connectsock()

设计一个connectsock过程封装底层代码，这部分代码在udp和tcp中都可能用到。

```
1  /* consock.cpp - connectsock */
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <winsock.h>
6  #ifndef INADDR_NONE
7  #define INADDR_NONE 0xffffffff
8  #endif /* INADDR_NONE */
9  void errexit(const char *, ...);
10 /*-----
11  * connectsock - allocate & connect a socket using TCP or UDP
12  *-----
13  */
14
15 //transport指的是所用协议名称
16 SOCKET connectsock(const char *host, const char *service, const char
17 *transport )
18 {
19     struct hostent *phe; /* pointer to host information entry */
20     struct servent *pse; /* pointer to service information entry */
21     struct protoent *ppe; /* pointer to protocol information entry */
22     struct sockaddr_in sin; /* an Internet endpoint address */
23     int s, type; /* socket descriptor and socket type */
24     memset(&sin, 0, sizeof(sin));
25     sin.sin_family = AF_INET;
26
27
28
```

```

29  /* Map service name to port number */
30  if ( pse = getservbyname(service, transport) )
31  sin.sin_port = pse->s_port;
32  else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
33  errexit("can't get \"%s\" service entry\n", service);
34  /* Map host name to IP address, allowing for dotted decimal */
35  if ( phe = gethostbyname(host) )
36  memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
37  else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
38  errexit("can't get \"%s\" host entry\n", host);
39  /* Map protocol name to protocol number */
40  if ( (ppe = getprotobyname(transport)) == 0 )
41  errexit("can't get \"%s\" protocol entry\n", transport);
42
43
44
45  /* Use protocol to choose a socket type */
46  if (strcmp(transport, "udp") == 0)
47  type = SOCK_DGRAM;
48  else
49  type = SOCK_STREAM;
50  /* Allocate a socket */
51  s = socket(PF_INET, type, ppe->p_proto);
52  if (s == INVALID_SOCKET)
53  errexit("can't create socket: %d\n", GetLastError());
54  /* Connect the socket */
55  if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) == SOCKET_ERROR)
56  errexit("can't connect to %s.%s: %d\n", host, service,
57  GetLastError());
58  return s;
59  }

```

客户端软件的实现-UDP客户端

设计connectUDP过程用于创建连接模式客户端UDP套接字

```

1  /* conUDP.cpp - connectUDP */
2  #include <winsock.h>
3  SOCKET connectsock(const char *, const char *, const char *);
4  /*-----
5   * connectUDP - connect to a specified UDP service
6   * on a specified host
7   *-----
8   */
9  SOCKET connectUDP(const char *host, const char *service )
10 {
11 return connectsock(host, service, "udp");
12 }

```

客户端软件的实现-TCP客户端

设计connectTCP过程，用于创建客户端TCP套接字

```
1  /* conTCP.cpp - connectTCP */
2  #include <winsock.h>
3  SOCKET connectsock(const char *, const char *, const char *);
4  /*-----
5   * connectTCP - connect to a specified TCP service
6   * on a specified host
7   *-----
8   */
9  SOCKET connectTCP(const char *host, const char *service )
10 {
11     return connectsock( host, service, "tcp");
12 }
```

客户端软件的实现-异常处理

```
1  /* errexit.cpp - errexit */
2  #include <stdarg.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <winsock.h>
6  /*-----
7   * errexit - print an error message and exit
8   *-----
9   */
10 /*VARARGS1*/
11 void errexit(const char *format, ...)
12 { va_list args;
13   va_start(args, format);
14   vfprintf(stderr, format, args);
15   va_end(args);
16   WSACleanup();
17   exit(1);}
```

例1：访问DAYTIME服务的客户端（TCP）

□ DAYTIME服务 □ 获取日期和时间 □ 双协议服务（TCP、UDP），端口号13 □ TCP版利用TCP连接请求触发服务（**不需要发送任何数据，只需要发送连接建立请求**） □ UDP版需要**客户端发送一个数据报**

```
1  /* TCPdtc.cpp - main, TCPdaytime */
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <winsock.h>
5  void TCPdaytime(const char *, const char *);
6  void errexit(const char *, ...);
7  SOCKET connectTCP(const char *, const char *);
8  #define LINELEN 128
9  #define WSVERS MAKEWORD(2, 0)
10 /*-----
```

```

11  * main - TCP client for DAYTIME service
12  *-----
13  */
14  int main(int argc, char *argv[])
15  {
16  char *host = "localhost"; /* host to use if none supplied */
17  char *service = "daytime"; /* default service port */
18  WSADATA wsadata;
19  switch (argc) {
20  case 1:
21  host = "localhost";
22  break;
23  case 3:
24  service = argv[2];
25  /* FALL THROUGH */
26  case 2:
27  host = argv[1];
28  break;
29  default:
30  fprintf(stderr, "usage: TCPdaytime [host [port]]\n");
31  exit(1);
32  }
33  if (WSAStartup(WSEVER, &wsadata) != 0)
34  errexit("WSAStartup failed\n");
35  TCPdaytime(host, service);
36  WSACleanup();
37  return 0; /* exit */
38  }
39  /*-----
40  * TCPdaytime - invoke Daytime on specified host and print results
41  *-----
42  */
43  void TCPdaytime(const char *host, const char *service)
44  {
45  char buf[LINELEN+1]; /* buffer for one line of text */
46  SOCKET s; /* socket descriptor */
47  int cc; /* recv character count */
48  s = connectTCP(host, service);
49  cc = recv(s, buf, LINELEN, 0);
50  while( cc != SOCKET_ERROR && cc > 0)
51  {
52  buf[cc] = '\0'; /* ensure null-termination */
53  (void) fputs(buf, stdout);
54  cc = recv(s, buf, LINELEN, 0);
55  }
56  closesocket(s);
57  }

```

例2：访问DAYTIME服务的客户端（UDP）

```

1  /* UDPdtc.cpp - main, UDPdaytime */
2  #include <stdlib.h>

```

```

3  #include <stdio.h>
4  #include <winsock.h>
5  void UDPdaytime(const char *, const char *);
6  void errexit(const char *, ...);
7  SOCKET connectUDP(const char *, const char *);
8  #define LINELEN 128
9  #define WSVERS MAKEWORD(2, 0)
10 #define MSG "what daytime is it?\n"
11 /*-----
12  * main - UDP client for DAYTIME service
13  *-----
14  */
15 int main(int argc, char *argv[])
16 {
17     char *host = "localhost"; /* host to use if none supplied */
18     char *service = "daytime"; /* default service port */
19     WSADATA wsadata;
20     switch (argc) {
21     case 1:
22         host = "localhost";
23         break;
24     case 3:
25         service = argv[2];
26         /* FALL THROUGH */
27     case 2:
28         host = argv[1];
29         break;
30     default:
31         fprintf(stderr, "usage: UDPdaytime [host [port]]\n");
32         exit(1);
33     }
34     if (WSAStartup(WSVERS, &wsadata) != 0)
35         errexit("WSAStartup failed\n");
36     UDPdaytime(host, service);
37     WSACleanup();
38     return 0; /* exit */
39 }
40 /*-----
41  * UDPdaytime - invoke Daytime on specified host and print results
42  *-----
43  */
44 void UDPdaytime(const char *host, const char *service)
45 {
46     char buf[LINELEN+1]; /* buffer for one line of text */
47     SOCKET s; /* socket descriptor */
48     int n; /* recv character count */
49     s = connectUDP(host, service);
50     (void) send(s, MSG, strlen(MSG), 0);
51     /* Read the daytime */
52     n = recv(s, buf, LINELEN, 0);
53     if (n == SOCKET_ERROR)
54         errexit("recv failed: recv() error %d\n", GetLastError());
55     else

```



```

56 {
57     buf[cc] = '\0'; /* ensure null-termination */
58     (void) fputs(buf, stdout);
59 }
60 closesocket(s);
61 return 0; /* exit */
62 }

```

Socket编程-服务器软件设计

4种类型基本服务器

循环：同时处理一个用户的请求

并发：同时处理多个用户请求

无连接：基于udp

面向连接：基于tcp

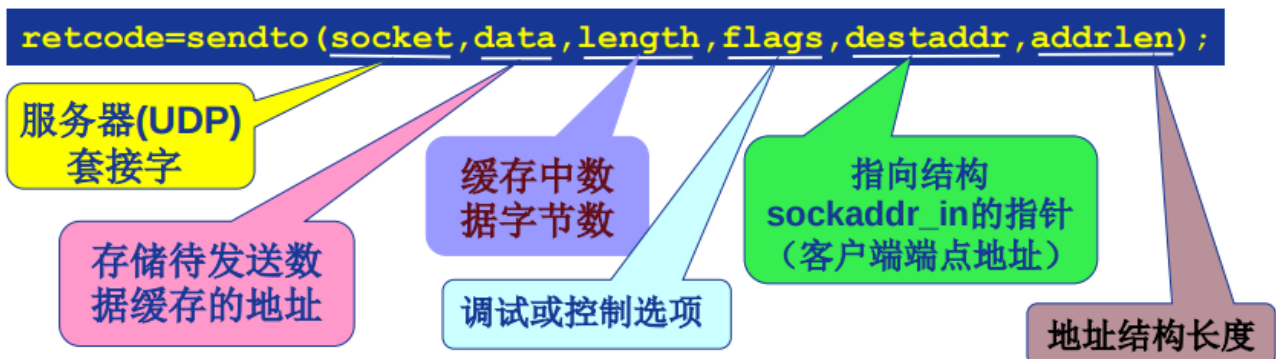
1. 循环无连接(Iterative connectionless)服务器
2. 循环面向连接(Iterative connection-oriented)服务器
3. 并发无连接(Concurrent connectionless)服务器
4. 并发面向连接(Concurrent connection-oriented)服务器

循环无连接服务器基本流程

1. 创建套接字
2. 绑定端点地址 (**INADDR_ANY+端口号**)
3. 反复接收来自客户端的请求
4. 遵循应用层协议，构造响应报文，发送给客户

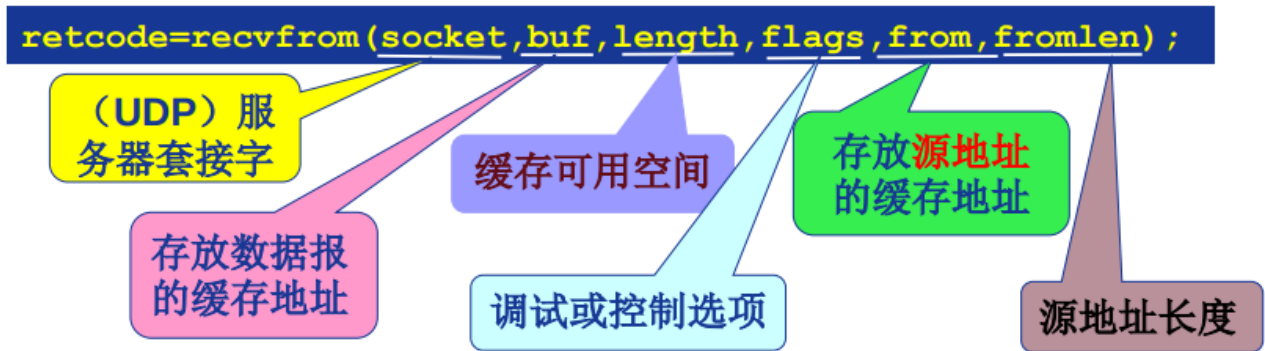
数据发送

□ 服务器端不能使用connect()函数，connect()是客户端专用 □ 无连接服务器使用sendto()函数发送数据报



获取客户端点地址

□ 调用recvfrom()函数接收数据时，自动提取客户进程端点地址



循环面向连接服务器基本流程

1. 创建（主）套接字，并绑定熟知端口号；
2. 设置（主）套接字为被动监听模式，准备用于服务器；
3. 调用`accept()`函数接收下一个连接请求（通过主套接字），创建新套接字用于与该客户建立连接；（主套接字要用与接受连接申请）
4. 遵循应用层协议，反复接收客户请求，构造并发送响应(通过新套接字)；
5. 完成为特定客户服务后，关闭与该客户之间的连接，返回步骤3.

并发无连接服务器基本流程

这里123指的是第一步第二步

主线程1: 创建套接字，并绑定熟知端口号；主线程2: 反复调用`recvfrom()`函数，接收下一个客户请求并创建新线程处理（为了并发）该客户响应；子线程1: 接收一个特定请求；子线程2: 依据应用层协议构造响应报文，并调用`sendto()`发送；子线程3: 退出(一个子线程处理一个请求后即终止)。

并发面向连接服务器基本流程

这里123指的是第一步第二步

主线程1: 创建（主）套接字，并绑定熟知端口号；

主线程2: 设置（主）套接字为被动监听模式，准备用于服务器；

主线程3: 反复调用`accept()`函数接收下一个连接请求（通过主套接字），并创建一个新子线程处理该客户响应；

子线程1: 接收一个客户的服务请求（通过新创建的套接字）；

子线程2: 遵循应用层协议与特定客户进行交互；

子线程3: 关闭/释放连接并退出（线程终止）。

服务器的实现

实现的一种设计，可以不这么设计。

□ 设计一个底层过程隐藏底层代码：passivesock() □ 两个高层过程分别用于创建服务器端UDP套接字和TCP套接字（调用passivesock()函数）：□ passiveUDP() □ passiveTCP()

服务器的实现-passivesock()

```
1  /* passsock.cpp - passivesock */
2  #include <stdlib.h>
3  #include <string.h>
4  #include <winsock.h>
5  void errexit(const char *, ...);
6  /*-----
7   * passivesock - allocate & bind a server socket using TCP or UDP
8   *-----
9   */
10 SOCKET passivesock(const char *service, const char *transport, int qlen)
11 {
12     struct servent *pse; /* pointer to service information entry */
13     struct protoent *ppe; /* pointer to protocol information entry */
14     struct sockaddr_in sin; /* an Internet endpoint address */
15     SOCKET s; /* socket descriptor */
16     int type; /* socket type (SOCK_STREAM, SOCK_DGRAM) */
17     memset(&sin, 0, sizeof(sin));
18     sin.sin_family = AF_INET;
19     sin.sin_addr.s_addr = INADDR_ANY;
20     /* Map service name to port number */
21     if ( pse = getservbyname(service, transport) )
22         sin.sin_port = (u_short)pse->s_port;
23     else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
24         errexit("can't get \"%s\" service entry\n", service);
25     /* Map protocol name to protocol number */
26     if ( (ppe = getprotobyname(transport)) == 0 )
27         errexit("can't get \"%s\" protocol entry\n", transport);
28     /* Use protocol to choose a socket type */
29     if (strcmp(transport, "udp") == 0)
30         type = SOCK_DGRAM;
31     else
32         type = SOCK_STREAM;
33     /* Allocate a socket */
34     s = socket(PF_INET, type, ppe->p_proto);
35     if (s == INVALID_SOCKET)
36         errexit("can't create socket: %d\n", GetLastError());
37     /* Bind the socket */
38     if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) == SOCKET_ERROR)
39         errexit("can't bind to %s port: %d\n", service,
40             GetLastError());
41     if (type == SOCK_STREAM && listen(s, qlen) == SOCKET_ERROR)
42         errexit("can't listen on %s port: %d\n", service,
43             GetLastError());
44     return s;}
45
46
```

服务器的实现-passiveUDP()

```
1  /* passUDP.cpp - passiveUDP */
2  #include <winsock.h>
3  SOCKET passivesock(const char *, const char *, int);
4  /*-----
   --
5  * passiveUDP - create a passive socket for use in a UDP server
6  *-----
   --
7  */
8  SOCKET passiveUDP(const char *service)
9  {
10 return passivesock(service, "udp", 0);
11 }
```

服务器的实现-passiveTCP()

```
1  /* passTCP.cpp - passiveTCP */
2  #include <winsock.h>
3  SOCKET passivesock(const char *, const char *, int);
4  /*-----
   --
5  * passiveTCP - create a passive socket for use in a TCP server
6  *-----
   -
7  */
8  SOCKET passiveTCP(const char *service, int qlen)
9  {
10 return passivesock(service, "tcp", qlen);
```

例1：无连接循环DAYTIME服务器

```
1  /* UDPdtd.cpp - main, UDPdaytimed */
2  #include <stdlib.h>
3  #include <winsock.h>
4  #include <time.h>
5  void errexit(const char *, ...);
6  SOCKET passiveUDP(const char *);
7  #define WSVERS MAKEWORD(2, 0)
8  /*-----
9  * main - Iterative UDP server for DAYTIME service
10 *-----
11 */
12 void main(int argc, char *argv[])
13 {
14 struct sockaddr_in fsin; /* the from address of a client */
```

```

15 char *service = "daytime"; /* service name or port number */
16 SOCKET sock; /* socket */
17 int alen; /* from-address length */
18 char *pts; /* pointer to time string */
19 time_t now; /* current time */
20 WSADATA wsadata;
21 switch (argc)
22 {
23     case 1:
24         break;
25     case 2:
26         service = argv[1];
27         break;
28     default:
29         errexit("usage: UDPdaytimed [port]\n");
30 }
31 if (WSAStartup(WSVERS, &wsadata) != 0)
32     errexit("WSAStartup failed\n");
33 sock = passiveUDP(service);
34 while (1)
35 {
36     alen = sizeof(struct sockaddr);
37     if (recvfrom(sock, buf, sizeof(buf), 0,
38         (struct sockaddr *)&fsin, &alen) == SOCKET_ERROR)
39         errexit("recvfrom: error %d\n", GetLastError());
40     (void) time(&now);
41     pts = ctime(&now);
42     (void) sendto(sock, pts, strlen(pts), 0,
43         (struct sockaddr *)&fsin, sizeof(fsin));
44 }
45 return 1; /* not reached */
46 }

```

例2：面向连接并发DAYTIME服务器

```

1  /* TCPdtd.cpp - main, TCPdaytimed */
2  #include <stdlib.h>
3  #include <winsock.h>
4  #include <process.h>
5  #include <time.h>
6  void errexit(const char *, ...);
7  void TCPdaytimed(SOCKET);
8  SOCKET passiveTCP(const char *, int);
9  #define QLEN 5
10 #define WSVERS MAKEWORD(2, 0)
11 /*-----
12  * main - Concurrent TCP server for DAYTIME service
13  *-----
14  */
15 void main(int argc, char *argv[])
16 {
17     struct sockaddr_in fsin; /* the from address of a client */

```

```

18 char *service = "daytime"; /* service name or port number*/
19 SOCKET msock, ssock; /* master & slave sockets */
20 int alen; /* from-address length */
21 WSADATA wsadata;
22 switch (argc) {
23 case1:
24 break;
25 case2:
26 service = argv[1];
27 break;
28 default:
29 errexit("usage: TCPdaytimed [port]\n");
30 }
31 if (WSAStartup(WSAVERS, &wsadata) != 0)
32 errexit("WSAStartup failed\n");
33 msock = passiveTCP(service, QLEN);
34 while (1) {
35 alen = sizeof(struct sockaddr);
36 ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
37 if (ssock == INVALID_SOCKET)
38 errexit("accept failed: error number %d\n",
39 GetLastError());
40 if (_beginthread((void (*)(void *)) TCPdaytimed, 0,
41 (void *)ssock) < 0) {
42 errexit("_beginthread: %s\n", strerror(errno));
43 }
44 }
45 return 1; /* not reached */
46 }
47 /*-----
48 * TCPdaytimed - do TCP DAYTIME protocol
49 *-----
50 */
51 void TCPdaytimed(SOCKET fd)
52 {
53 char * pts; /* pointer to time string */
54 time_t now; /* current time */
55 (void) time(&now);
56 pts = ctime(&now);
57 (void) send(fd, pts, strlen(pts), 0);
58 (void) closesocket(fd);
59 }

```