## 第12课 讲稿（Linux网络编程-1）

1. **TCP/IP 协议概述**
2. **OSI 参考模型及TCP/IP 参考模型**

OSI（开放式系统互联）协议参考模型，是基于国际标准化组织（ISO）的建议发展起来的，从上到下共分为7 层：应用层、表示层、会话层、传输层、网络层、数据链路层及物理层。这个7 层的协议模型虽然规定得非常细致和完善，但在实际中却得不到广泛的应用，其重要的原因之一就在于它过于复杂。但它仍是此后很多协议模型的基础，这种分层架构的思想在很多领域都得到了广泛的应用。

与此相区别的TCP/IP 协议模型从一开始就遵循简单明确的设计思路，它将TCP/IP 的7 层协议模型简化为4 层，从而更有利于实现和使用。TCP/IP 的协议参考模型和OSI 协议参考模型的对应关系如图1所示。

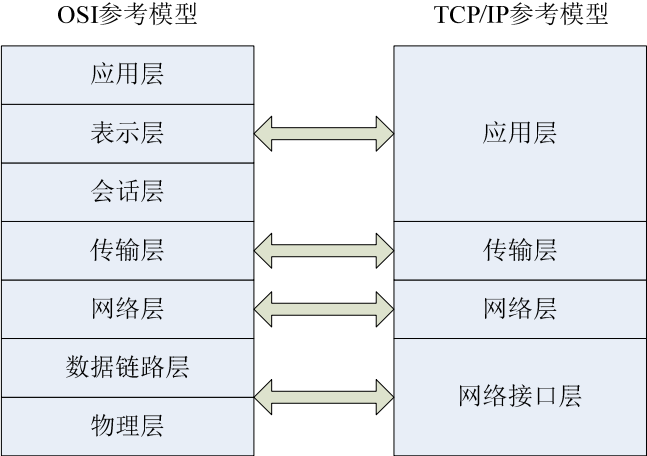


图1 OSI模型和TCP/IP参考模型对应关系

下面分别对 TCP/IP 的4 层模型进行简要介绍。

* 网络接口层：负责将二进制流转换为数据帧，并进行数据帧的发送和接收。要注意的是数据帧是独立的网络信息传输单元。
* 网络层：负责将数据帧封装成 IP 数据包，并运行必要的路由算法。
* 传输层：负责端对端之间的通信会话连接与建立。传输协议的选择根据数据传输方式而定。
* 应用层：负责应用程序的网络访问，这里通过端口号来识别各个不同的进程。

1. **TCP/IP 协议族**

虽然 TCP/IP 名称只包含了两个协议，但实际上，TCP/IP 是一个庞大的协议族，它包括了各个层次上的众多协议，图2列举了各层中一些重要的协议，并给出了各个协议在不同层次中所处的位置，如下所示。

* ARP：用于获得同一物理网络中的硬件主机地址。
* MPLS：多协议标签协议，是很有发展前景的下一代网络协议。
* IP：负责在主机和网络之间寻址和路由数据包。
* ICMP：用于发送有关数据包的传送错误的协议。
* IGMP：被IP 主机用来向本地多路广播路由器报告主机组成员的协议。
* TCP：为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况。并适用于要求得到响应的应用程序。
* UDP：提供了无连接通信，且不对传送包进行可靠性保证。适合于一次传输少量数据，可靠性则由应用层来负责。

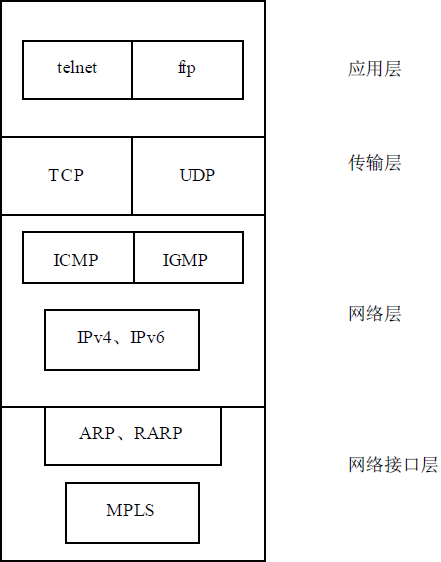


图2 TCP/IP协议族

1. **TCP 和UDP**

在此主要介绍在网络编程中涉及的传输层TCP 和UDP 协议。

1. **TCP**
2. 概述

同其他任何协议栈一样，TCP 向相邻的高层提供服务。因为TCP 的上一层就是应用层，因此，**TCP 数据传输实现了从一个应用程序到另一个应用程序的数据传递**。应用程序通过编程调用TCP 并使用TCP 服务，提供需要发送的数据，用来区分接收数据应用的目的地址和端口号。

通常应用程序通过打开一个socket来使用TCP 服务，TCP 管理到其他socket 的数据传递。可以说，**通过IP 的源/目的可以惟一地区分网络中两个设备的连接**，**通过socket 的源/目的可以惟一地区分网络中两个应用程序的连接**。

1. 三次握手协议。

TCP 对话通过三次握手来进行初始化。三次握手的目的是使数据段的发送和接收同步，告诉其他主机其一次可接收的数据量，并建立虚连接。

下面描述了这三次握手的简单过程。

* 初始化主机通过一个同步标志置位的数据段发出会话请求。
* 接收主机通过发回具有以下项目的数据段表示回复：同步标志置位、即将发送的数据段的起始字节的顺序号、应答并带有将收到的下一个数据段的字节顺序号。
* 请求主机再回送一个数据段，并带有确认顺序号和确认号。

图3就是这个流程的简单示意图。

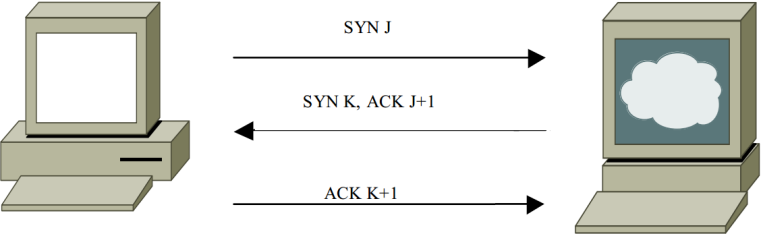


图3 TCP三次握手协议

TCP 实体所采用的基本协议是滑动窗口协议。当发送方传送一个数据报时，它将启动计时器。当该数据报到达目的地后，接收方的TCP 实体往回发送一个数据报，其中包含有一个确认序号，它表示希望收到的下一个数据包的顺序号。如果发送方的定时器在确认信息到达之前超时，那么发送方会重发该数据包。

1. TCP 数据包头。

图4给出了TCP数据报头的格式。



图4 TCP数据报头的格式

TCP 数据包头的含义如下所示。

* 源端口、目的端口：16 位长。标识出远端和本地的端口号。
* 序号：32 位长。标识发送的数据报的顺序。
* 确认号：32 位长。希望收到的下一个数据包的序列号。
* TCP 头长：4 位长。表明TCP 头中包含多少个32 位字。
* 6位未用。
* URG：紧急指针 ，设置为1时有效。紧急方式是向对方发送紧急数据的一种方式，表示数据需要优先处理。
* ACK：ACK 位置1 表明确认号是合法的。如果ACK 为0，那么数据报不包含确认信息，确认字段被省略。
* PSH：表示是带有PUSH 标志的数据。接收方因此请求数据包一到便将其送往应用程序而不必等到缓冲区装满时才传送。
* RST：用于复位由于主机崩溃或其他原因而出现的错误连接。还可以用于拒绝非法的数据包或拒绝连接请求。
* SYN：用于建立连接。
* FIN：用于释放连接。
* 窗口大小：16 位长。窗口大小字段表示在确认了字节之后还可以发送多少个字节。
* 校验和：16 位长。是为了确保高可靠性而设置的。它校验头部、数据和伪TCP 头部之和。
* 可选项：0 个或多个32 位字。包括最大TCP 载荷，滑动窗口比例以及选择重发数据包等选项。

1. **UDP**
2. 概述。

UDP 即用户数据报协议，它是一种无连接协议，因此不需要像TCP 那样通过三次握手来建立一个连接。

同时，一个UDP 应用可同时作为应用的客户或服务器方。由于UDP 协议并不需要建立一个明确的连接，因此建立UDP 应用要比建立TCP 应用简单得多。

UDP 协议从问世至今已经被使用了很多年，虽然其最初的光彩已经被一些类似协议所掩盖，但是在网络质量越来越高的今天，UDP 的应用得到了大大的增强。它比TCP 协议更为高效，也能更好地解决实时性的问题。如今，包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都使用UDP 协议。

1. UDP 数据报头。

UDP 数据报头如图5所示。

* 源地址、目的地址：16 位长。标识出远端和本地的端口号。
* 数据报的长度是指包括报头和数据部分在内的总的字节数。因为报头的长度是固定的，所以该域主要用来计算可变长度的数据部分（又称为数据负载）。



图5 UDP 数据报头

1. **协议的选择**

协议的选择应该考虑到以下 3 个方面。

1. 对数据可靠性的要求。

对数据要求高可靠性的应用需选择 TCP 协议，如验证、密码字段的传送都是不允许出错的。而对数据的可靠性要求不那么高的应用可选择UDP 传送。

1. 应用的实时性。

TCP 协议在传送过程中要使用三次握手、重传确认等手段来保证数据传输的可靠性。使用TCP 协议会有较大的时延，因此不适合对实时性要求较高的应用，如VOIP、视频监控等。相反，UDP 协议则在这些应用中能发挥很好的作用。

1. 网络的可靠性。

由于 TCP 协议的提出主要是解决网络的可靠性问题，它通过各种机制来减少错误发生的概率。因此，在网络状况不是很好的情况下需选用TCP 协议（如在广域网等情况），但是若在网络状况很好的情况下（如局域网等）就不需要再采用TCP 协议，而建议选择UDP 协议来减少网络负荷。

1. **网络基础编程**
2. **socket概述**
3. **socket定义**

在Linux中的网络编程是通过socket接口来进行的。Socket也称为套接字，**它是一种特殊的I/O接口，也是一种文件描述符**。socket是一种常用的进程之间通信机制，通过它不仅能实现**本地机器上的进程之间的通信**，而且通过网络能够在**不同机器上的进程之间进行通信**。

每一个socket都用一个半相关描述{协议、本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议、本地地址、本地端口、远程地址、远程端口}来表示。socket也有一个类似于打开文件的函数调用，该函数返回一个整型的socket描述符，随后的连接建立、数据传输等操作都是通过socket来实现的。

1. **socket 类型**

常见的socket有如下3种类型：

1. 流式socket（SOCK\_STREAM）。

流式套接字提供可靠的、面向连接的通信流；它使用TCP协议，从而保证了数据传输的正确性和顺序性。

1. 数据报socket（SOCK\_DGRAM）。

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证是可靠、无差错的。它使用数据报协议UDP。

1. 原始socket。

原始套接字允许对底层协议如IP或ICMP进行直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

1. **地址及顺序处理**
2. **地址结构相关处理**
3. 数据结构介绍。

下面首先介绍两个重要的数据类型：sockaddr和sockaddr\_in，这两个结构类型都是用来保存socket信息的，如下所示：

|  |
| --- |
| **struct** **sockaddr**  {  unsigned short sa\_family; /\*地址族\*/  char sa\_data[14]; /\*14 字节的协议地址，包含该socket 的IP 地址和端口号。\*/  };  **struct** **sockaddr\_in**  {  short int sa\_family; /\*地址族\*/  unsigned short int sin\_port; /\*端口号\*/  struct in\_addr sin\_addr; /\*IP 地址\*/  unsigned char sin\_zero[8]; /\*填充0以保持与struct sockaddr 同样大小\*/  }; |

这两个数据类型是等效的，可以相互转化，通常sockaddr\_in数据类型使用更为方便。在建立sockaddr或sockaddr\_in后，就可以对该socket进行适当的操作了。

1. 结构字段。

表1列出了该结构sa\_family字段可选的常见值。

|  |  |
| --- | --- |
| 表1 sa\_family字段可选的常见值 | |
| 结构定义头文件 | #include <netinet/in.h> |
| sa\_family | AF\_INET：IPv4协议，在sockaddr和sockaddr\_in中只能取这个值，要使用AF\_INET6，需要使用sockaddr\_in6。 |
| AF\_INET6：IPv6协议 |
| AF\_LOCAL：UNIX域协议 |
| AF\_LINK：链路地址协议 |
| AF\_KEY：密钥套接字（socket） |

sockaddr\_in其他字段的含义非常清楚，具体的设置涉及其他函数，在后面会有详细的讲解。

1. **数据存储优先顺序**
2. 函数说明。

计算机数据存储有两种字节优先顺序：高位字节优先（称为大端模式）和低位字节优先（称为小端模式）。Internet上数据以大端模式传输，PC机通常采用小端模式，因此在有些情况下，需要对这两个字节存储优先顺序进行相互转化。这里用到了4个函数：htons()、ntohs()、htonl()和ntohl()，分别实现网络字节序和主机字节序的转化，这里的h代表host，n代表network，s代表short，l代表long。通常16位的IP端口号用s代表，而32位的IP地址用l来代表。

1. 函数格式说明。

表2列出了这4个函数的语法格式。

|  |  |
| --- | --- |
| 表2 htons等函数语法要点 | |
| 所需头文件 | #include <netinet/in.h> |
| 函数原型 | uint16\_t htons(unit16\_t host16bit)  uint32\_t htonl(unit32\_t host32bit)  uint16\_t ntohs(unit16\_t net16bit)  uint32\_t ntohs(unit32\_t net32bit) |
| 函数传入值 | host16bit：主机字节序的16位数据 |
| host32bit：主机字节序的32位数据 |
| net16bit：网络字节序的16位数据 |
| net32bit：网络字节序的32位数据 |
| 函数返回值 | 成功：返回要转换的字节序 |
| 出错： -1 |

|  |  |
| --- | --- |
| 注意 | 调用该函数只是使其得到相应的字节序，用户不需清楚该系统的主机字节序和网络字节序是否真正相等。如果是相同不需要转换的话，该系统的这些函数会定义成空宏。 |

1. **地址格式转化**
2. 函数说明。

通常用户在表达地址时采用的是点分十进制表示的数值（或者是以冒号分开的十进制 IPv6 地址），而在通常使用的socket 编程中所使用的则是二进制值，这就需要将这两个数值进行转换。这里主要介绍能同时兼容IPv4和IPv6的函数inet\_pton()和inet\_ntop()。

inet\_pton()函数是将点分十进制地址映射为二进制地址，而inet\_ntop()是将二进制地址映射为点分十进制地址。

1. 函数格式。

表3列出了inet\_pton 函数的语法要点。

|  |  |  |
| --- | --- | --- |
| 表3 inet\_pton函数语法要点 | | |
| 所需头文件 | #include <arpa/inet.h> | |
| 函数原型 | int inet\_pton(int family, const char \*src, void \*dst) | |
| 函数传入值 | family | AF\_INET： IPv4 协议 |
| AF\_INET6： IPv6 协议 |
| src：来源地址 | |
| dst：转化后的地址 | |
| 函数返回值 | 成功： 0 | |
| 出错： -1 | |

表4列出了inet\_ntop 函数的语法要点。

|  |  |  |
| --- | --- | --- |
| 表4 inet\_ntop函数语法要点 | | |
| 所需头文件 | #include <arpa/inet.h> | |
| 函数原型 | int inet\_ntop(int family, const void \*src, char \*dst, size\_t len) | |
| 函数传入值 | family | AF\_INET： IPv4 协议 |
| AF\_INET6： IPv6 协议 |
| src：来源地址 | |
| dst：转化后的地址 | |
| len：缓存区dst的大小，避免溢出，如果缓存区太小无法存储地址的值，则返回一个空[指针](http://baike.baidu.com/view/159417.htm)，并将errno置为ENOSPC。 | |
| 函数返回值 | 成功： 0 | |
| 出错： -1 | |

1. **名字地址转化**
2. 函数说明。

通常，人们在使用过程中都不愿意记忆冗长的IP地址，尤其到IPv6时，地址长度多达128位，那时就更加不可能一次次记忆那么长的IP 地址了。因此，使用主机名将会是很好的选择。在Linux中，同样有一些函数可以实现主机名和地址的转化，最为常见的有gethostbyname()、gethostbyaddr()和getaddrinfo()等，它们都可以实现IPv4和IPv6的地址和主机名之间的转化。其中gethostbyname()是将主机名转化为IP地址，gethostbyaddr()则是逆操作，是将IP地址转化为主机名，另外getaddrinfo()还能实现自动识别IPv4地址和IPv6地址。

gethostbyname()和gethostbyaddr()都涉及一个hostent的结构体，如下所示：

|  |
| --- |
| **struct hostent**  {  char \*h\_name;/\*正式主机名\*/  char \*\*h\_aliases;/\*主机别名\*/  int h\_addrtype;/\*地址类型\*/  int h\_length;/\*地址字节长度\*/  char \*\*h\_addr\_list;/\*指向IPv4或IPv6的地址指针数组\*/  };  #define h\_addr h\_addr\_list[0] |

调用gethostbyname()函数或gethostbyaddr()函数后就能返回hostent结构体的相关信息。

getaddrinfo()函数涉及一个addrinfo的结构体，如下所示：

|  |
| --- |
| **struct addrinfo**  {  int ai\_flags;/\*AI\_PASSIVE, AI\_CANONNAME;\*/  int ai\_family;/\*地址族\*/  int ai\_socktype;/\*socket 类型\*/  int ai\_protocol;/\*协议类型\*/  size\_t ai\_addrlen;/\*地址字节长度\*/  char \*ai\_canonname;/\*主机名\*/  struct sockaddr \*ai\_addr;/\*socket 结构体\*/  struct addrinfo \*ai\_next;/\*下一个指针链表\*/  } |

表5列举了addrinfo结构体常见的选项值。

|  |  |
| --- | --- |
| 表5 addrinfo 结构体常见选项值 | |
| 所需头文件 | #include <netdb.h> |
| ai\_flags | AI\_PASSIVE：该套接口是被动的，用于bind，主要用于服务器 |
| AI\_CANONNAME：将正规主机名返回给addrinfo结构体中的ai\_canonname成员 |
| ai\_family | AF\_INET： IPv4 协议 |
| AF\_INET6： IPv6 协议 |
| AF\_UNSPEC： IPv4 或 IPv6 均可 |
| ai\_socktype | SOCK\_STREAM：字节流套接字 socket（ TCP） |
| SOCK\_DGRAM：数据报套接字 socket（ UDP） |
| ai\_protocol | IPPROTO\_IP： IP 协议 |
| IPPROTO\_IPV4： IPv4 协议 |
| IPPROTO\_IPV6： IPv6 协议 |
| IPPROTO\_UDP： UDP |
| IPPROTO\_TCP： TCP |

可见，相对于hostent 结构体，addrinfo 结构体包含更多的信息。

1. 函数格式。

表6列出了gethostbyname()函数的语法要点。

|  |  |
| --- | --- |
| 表6 gethostbyname 函数语法要点 | |
| 所需头文件 | #include <netdb.h> |
| 函数原型 | struct hostent \*gethostbyname(const char \*hostname) |
| 函数传入值 | hostname：主机名 |
| 函数返回值 | 成功： hostent 类型指针 |
| 出错： -1 |

调用该函数时可以首先对 hostent 结构体中的h\_addrtype 和h\_length 进行设置，若为IPv4 可设置为AF\_INET 和4；若为IPv6 可设置为AF\_INET6 和16；若不设置则默认为IPv4 地址类型。

表7列出了getaddrinfo()函数的语法要点。

|  |  |
| --- | --- |
| 表7 getaddrinfo()函数语法要点 | |
| 所需头文件 | #include <netdb.h> |
| 函数原型 | int getaddrinfo(const char \*node, const char \*service, const struct addrinfo \*hints, struct addrinfo \*\*result) |
| 函数传入值 | node：IP地址或者主机名 |
| service：十进制的端口号或已定义的服务名称，如ftp、http等 |
| hints：服务线索 |
| result：返回结果 |
| 函数返回值 | 成功： 0 |
| 出错： -1 |

在调用之前，首先要对 hints 服务线索进行设置，它也是addrinfo类型的结构体。

1. 使用实例。

下面的实例给出了getaddrinfo函数用法的示例：

|  |
| --- |
| **/\* getaddrinfo.c \*/**  #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <string.h>  #include <netdb.h>  #include <sys/types.h>  #include <netinet/in.h>  #include <sys/socket.h>  int main(int argc, char \* argv[])  {  struct addrinfo hints, \*res = NULL;  int rc;  memset(&hints, 0, sizeof(hints));  /\*设置addrinfo 结构体中各参数 \*/  **hints.ai\_flags = AI\_CANONNAME;**  **hints.ai\_family = AF\_UNSPEC;**  **hints.ai\_socktype = SOCK\_DGRAM;**  **hints.ai\_protocol = IPPROTO\_UDP;**  /\*调用getaddinfo 函数\*/  if(argc<2)  sprintf(buf,"localhost");  else  sprintf(buf,argv[1]);  rc = getaddrinfo(buf, NULL, &hints, &res);  if (rc != 0)  {  perror("getaddrinfo");  exit(1);  }  else  {  printf("Host name is %s\n", res->ai\_canonname);  }  exit(0);  } |

该程序可以将主机名hostname对应的IP地址存入res结构体。

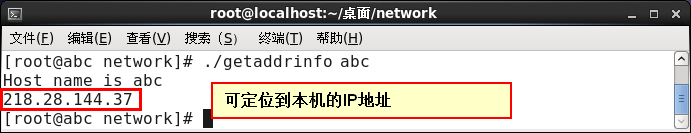
若在程序中添加如下代码，可以把主机名对应的IP地址打印输出。

|  |
| --- |
| struct addrinfo \*cur;  struct sockaddr\_in \*addr;  char ipbuf[16];  for (cur = res; cur != NULL; cur = cur->ai\_next)  {  addr = (struct sockaddr\_in \*)cur->ai\_addr;  inet\_ntop(AF\_INET,&addr->sin\_addr, ipbuf, 16);  printf("%s\n", ipbuf); } |

【测试1】将虚拟机的网络连接方式设置为“桥接”，并给Linux设置IP地址（可以自动获取），编译运行程序。



【测试2】用hostname命令把Linux的主机名修改为“abc”，把程序中“rc = getaddrinfo("localhost", NULL, &hints, &res)”改为“rc = getaddrinfo("abc", NULL, &hints, &res)”，再次运行程序，通过argv[1]把修改后的主机名传递给程序。



【注意】主机名“localhost”时，对应的IP是本地环回路由127.0.0.1；主机名为hostname的值时，对应的IP是eth0的IP地址。

1. **socket 基础编程**
2. 函数说明。

socket 编程的基本函数有socket()、bind()、listen()、accept()、send()、sendto()、recv()以及recvfrom()等，其中根据客户端还是服务端，或者根据使用TCP协议还是UDP协议，这些函数的调用流程都有所区别，这里先对每个函数进行说明，再给出各种情况下使用的流程图。

* socket()：该函数用于建立一个socket连接，可指定socket类型等信息。在建立了socket 连接之后，可对sockaddr或sockaddr\_in结构进行初始化，以保存所建立的socket地址信息。
* bind()：该函数是用于将本地IP地址绑定到端口号，主要用于TCP的连接，而在UDP的连接中则无必要。
* listen()：在服务端程序成功建立套接字和与地址进行绑定之后，还需要准备在该套接字上接收新的连接请求，此时调用listen()函数来创建一个等待队列，在其中存放未处理的客户端连接请求。
* accept()：服务端程序调用listen()函数创建等待队列之后，调用accept()函数等待并接收客户端的连接请求。它通常从由listen()所创建的等待队列中取出第一个未处理的连接请求，创建一个新的套接字，并返回指向该套接字的文件描述符
* connect()：该函数在TCP中是用于bind()之后的client端，用于与服务器端建立连接。
* send()和recv()：这两个函数分别用于发送和接收数据，可以用在TCP中，也可以用在UDP中。
* sendto()和recvfrom()：这两个函数的作用与send()和recv()函数类似，也可以用在TCP和UDP中。当用在TCP时，后面的几个与地址有关参数不起作用，函数作用等同于send()和recv()；当用在UDP时，可以用在之前没有使用connect()的情况下，这两个函数可以自动寻找指定地址并进行连接。

服务器端和客户端使用TCP和UDP协议的流程分别如图6和图7所示。

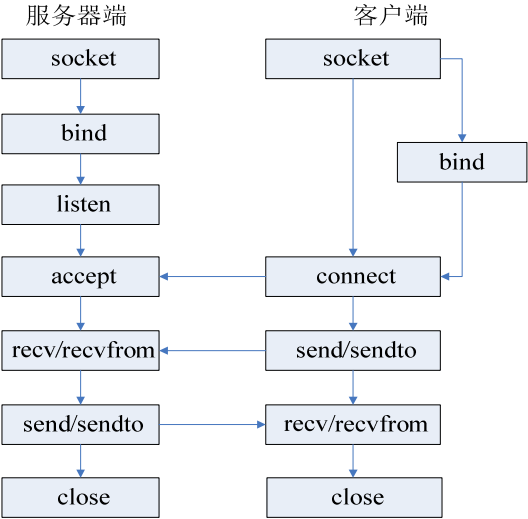
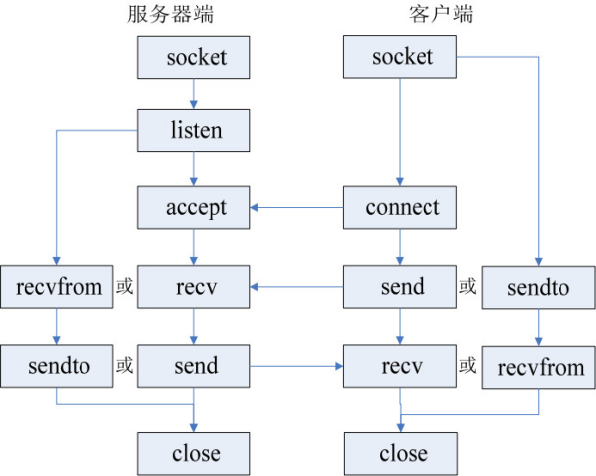
 

图6 使用TCP协议socket编程流程图 图7 使用UDP协议socket编程流程图

1. 函数格式。

表8列出了socket()函数的语法要点。

|  |  |  |
| --- | --- | --- |
| 表8 socket()函数语法要点 | | |
| 所需头文件 | #include <sys/socket.h> | |
| 函数原型 | int socket(int family, int type, int protocol) | |
| 函数传入值 | family：  协议族 | AF\_INET： IPv4 协议 |
| AF\_INET6： IPv6 协议 |
| AF\_LOCAL： UNIX 域协议 |
| AF\_ROUTE：路由套接字 |
| AF\_KEY：密钥套接字 |
| type：  套接字类型 | SOCK\_STREAM：字节流套接字（TCP） |
| SOCK\_DGRAM：数据报套接字（UDP） |
| SOCK\_RAW：原始套接字 |
| protocol： 常用的协议有IPPROTO\_TCP、IPPROTO\_UDP等，分别对应于TCP传输和UDP传输，如果不想指定，可以直接设置为0 | |
| 函数返回值 | 成功：非负套接字描述符 | |
| 出错： -1 | |

表9列出了bind()函数的语法要点。

|  |  |
| --- | --- |
| 表9 bind()函数语法要点 | |
| 所需头文件 | #include <sys/socket.h> |
| 函数原型 | int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen) |
| 函数传入值 | sockfd：套接字描述符 |
| my\_addr：本地地址 |
| addrlen：地址长度 |
| 函数返回值 | 成功： 0 |
| 出错： -1 |

端口号和地址在my\_addr中给出，若不指定地址，则内核随意分配一个临时端口给该应用程序。

表10列出了listen()函数的语法要点。

|  |  |
| --- | --- |
| 表10 listen()函数语法要点 | |
| 所需头文件 | #include <sys/socket.h> |
| 函数原型 | int listen(int sockfd, int backlog) |
| 函数传入值 | socktd：套接字描述符 |
| backlog：请求队列中允许的最大请求数，大多数系统缺省值为5 |
| 函数返回值 | 成功： 0 |
| 出错： -1 |

表11列出了accept()函数的语法要点。

|  |  |
| --- | --- |
| 表11 accept()函数语法要点 | |
| 所需头文件 | #include <sys/socket.h> |
| 函数原型 | int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen) |
| 函数传入值 | socktd：服务器端套接字描述符 |
| addr：返回参数，返回客户端地址 |
| addrlen：返回参数，返回客户端地址长度 |
| 函数返回值 | 成功： 返回客户端的socket |
| 出错： -1 |

表12列出了connect()函数的语法要点。

|  |  |
| --- | --- |
| 表12 connect()函数语法要点 | |
| 所需头文件 | #include <sys/socket.h> |
| 函数原型 | int connect(int sockfd, struct sockaddr \*serv\_addr, int addrlen) |
| 函数传入值 | socktd：套接字描述符 |
| serv\_addr：服务器端地址 |
| addrlen：地址长度 |
| 函数返回值 | 成功： 0 |
| 出错： -1 |

表13列出了send()函数的语法要点。

|  |  |
| --- | --- |
| 表13 send()函数语法要点 | |
| 所需头文件 | #include <sys/socket.h> |
| 函数原型 | int send(int sockfd, const void \*msg, int len, int flags) |
| 函数传入值 | socktd：套接字描述符 |
| msg：指向要发送数据的指针 |
| len：数据长度 |
| flags：一般为 0 |
| 函数返回值 | 成功：发送的字节数 |
| 出错： -1 |

表14列出了recv()函数的语法要点。

|  |  |
| --- | --- |
| 表14 recv()函数语法要点 | |
| 所需头文件 | #include <sys/socket.h> |
| 函数原型 | int recv(int sockfd, void \*buf, int len, int flags) |
| 函数传入值 | socktd：套接字描述符 |
| buf：存放接收数据的缓冲区 |
| len：数据长度 |
| flags：一般为 0 |
| 函数返回值 | 成功：接收的字节数 |
| 出错： -1 |

表15列出了sendto()函数的语法要点。

|  |  |
| --- | --- |
| 表15 sendto()函数语法要点 | |
| 所需头文件 | #include <sys/socket.h> |
| 函数原型 | int sendto(int sockfd, const void \*msg, int len, unsigned int flags, const struct sockaddr \*to, int tolen) |
| 函数传入值 | socktd：套接字描述符 |
| msg：指向要发送数据的指针 |
| len：数据长度 |
| flags：一般为 0 |
| to：目地机的 IP 地址和端口号信息 |
| tolen：地址长度 |
| 函数返回值 | 成功：发送的字节数 |
| 出错： -1 |

表16列出了recvfrom()函数的语法要点。

|  |  |
| --- | --- |
| 表16 recvfrom()函数语法要点 | |
| 所需头文件 | #include <sys/socket.h> |
| 函数原型 | int recvfrom(int sockfd, void \*buf, int len, unsigned int flags, struct sockaddr \*from, int \*fromlen) |
| 函数传入值 | socktd：套接字描述符 |
| buf：存放接收数据的缓冲区 |
| len：数据长度 |
| flags：一般为 0 |
| from：源主机的 IP 地址和端口号信息 |
| tolen：地址长度 |
| 函数返回值 | 成功：接收的字节数 |
| 出错： -1 |

1. 使用实例。

该实例分为客户端和服务器端两部分，其中服务器端首先建立起socket，然后与本地端口进行绑定，接着就监听客户端，接收从客户端的连接请求并建立与它的连接，接下来，接收客户端发送的消息。客户端则在建立socket之后调用connect()函数来建立连接。

服务端的代码如下所示：

|  |
| --- |
| **/\*server.c\*/**  #include <sys/types.h>  #include <sys/socket.h>  #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <string.h>  #include <unistd.h>  #include <netinet/in.h>  #define PORT 4321  #define BUFFER\_SIZE 1024  #define MAX\_QUE\_CONN\_NM 5  int main()  {  struct sockaddr\_in server\_sockaddr,client\_sockaddr;  int sin\_size,recvbytes;  int sockfd, client\_fd;  char buf[BUFFER\_SIZE];  /\*建立socket 连接\*/  if ((sockfd = **socket(AF\_INET,SOCK\_STREAM,0))=**= -1)  {  perror("socket");  exit(1);  }  printf("Socket id = %d\n",sockfd);  /\*设置sockaddr\_in 结构体中相关参数\*/  **server\_sockaddr.sin\_family = AF\_INET;**  **server\_sockaddr.sin\_port = htons(PORT);**  **server\_sockaddr.sin\_addr.s\_addr = INADDR\_ANY;**  **memset(&(server\_sockaddr.sin\_zero),0,8);**  int i = 1;  /\* 允许重复使用本地地址与套接字进行绑定 \*/  **setsockopt (sockfd, SOL\_SOCKET, SO\_REUSEADDR, &i, sizeof(i));**  /\*绑定函数bind()\*/  if **(bind(sockfd, (struct sockaddr \*)&server\_sockaddr,**  **sizeof(struct sockaddr))** == -1)  {  perror("bind");  exit(1);  }  printf("Bind success!\n");  /\*调用listen()函数，创建未处理请求的队列\*/  if **(listen(sockfd, MAX\_QUE\_CONN\_NM)** == -1)  {  perror("listen");  exit(1);  }  printf("Listening....\n");  /\*调用accept()函数，等待客户端的连接\*/  if ((client\_fd = **accept(sockfd,(struct sockaddr \*)&client\_sockaddr, &sin\_size))** == -1)  {  perror("accept");  exit(1);  }  /\*调用recv()函数接收客户端的请求\*/  memset(buf , 0, sizeof(buf));  if **((recvbytes = recv(client\_fd, buf, BUFFER\_SIZE, 0))** == -1)  {  perror("recv");  exit(1);  }  printf("Received a message: %s\n", buf);  close(sockfd);  exit(0);  } |

客户端的代码如下所示：

|  |
| --- |
| **/\*client.c\*/**  #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <string.h>  #include <netdb.h>  #include <sys/types.h>  #include <netinet/in.h>  #include <sys/socket.h>  #define PORT 4321  #define BUFFER\_SIZE 1024  int main(int argc, char \*argv[])  {  int sockfd,sendbytes;  char buf[BUFFER\_SIZE];  struct hostent \*host;  struct sockaddr\_in serv\_addr;  if(argc < 3)  {  printf("USAGE: ./client Hostname(or ip address) Text\n");  exit(1);  }  /\*地址解析函数\*/  if ((host = gethostbyname(argv[1])) == NULL)  {  perror("gethostbyname");  exit(1);  }  memset(buf, 0, sizeof(buf));  sprintf(buf, "%s", argv[2]);  /\*创建socket\*/  if **((sockfd = socket(AF\_INET, SOCK\_STREAM, 0))** == -1)  {  perror("socket");  exit(1);  }  /\*设置sockaddr\_in 结构体中相关参数\*/  **serv\_addr.sin\_family = AF\_INET;**  **serv\_addr.sin\_port = htons(PORT);**  **serv\_addr.sin\_addr = \*((struct in\_addr \*)host->h\_addr);**  **memset(&(serv\_addr.sin\_zero),0,8);**  /\*调用connect 函数主动发起对服务器端的连接\*/  if**(connect(sockfd,(struct sockaddr \*)&serv\_addr,**  **sizeof(struct sockaddr))==** -1)  {  perror("connect");  exit(1);  }  /\*发送消息给服务器端\*/  if ((sendbytes = **send(sockfd, buf, strlen(buf), 0))** == -1)  {  perror("send");  exit(1);  }  close(sockfd);  exit(0);  } |

在运行时需要先启动服务器端，再启动客户端。这里可以把服务器端下载到开发板上，客户端在宿主机上

运行，然后配置双方的IP 地址，在确保双方可以通信（如使用ping 命令验证）的情况下运行该程序即可。

|  |
| --- |
| $ ./server  Socket id = 3  Bind success!  Listening....  Received a message: Hello,Server!  $ ./client localhost(或者输入IP 地址) Hello,Server! |