**[本地socket unix domain socket](http://blog.csdn.net/bingqingsuimeng/article/details/8470029)**

**Unix domain socket** 或者 **IPC socket**是一种终端，可以使同一台[操作系统](http://zh.wikipedia.org/wiki/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \o "操作系统)上的两个或多个[进程](http://zh.wikipedia.org/wiki/%E8%BF%9B%E7%A8%8B)进行数据通信。与[管道](http://zh.wikipedia.org/wiki/%E7%AE%A1%E9%81%93_(Unix))相比，Unix domain sockets 既可以使用[字节流](http://zh.wikipedia.org/wiki/%E5%AD%97%E7%AF%80%E6%B5%81" \o "字节流)和数据队列，而管道通信则只能通过[字节流](http://zh.wikipedia.org/wiki/%E5%AD%97%E7%AF%80%E6%B5%81)。Unix domain sockets的接口和Internet socket很像，但它不使用网络底层协议来通信。Unix domain socket 的功能是[POSIX](http://zh.wikipedia.org/wiki/POSIX" \o "POSIX)操作系统里的一种组件。

Unix domain sockets 使用系统文件的地址来作为自己的身份。它可以被系统进程引用。所以两个进程可以同时打开一个Unix domain sockets来进行通信。不过这种通信方式是发生在系统内核里而不会在网络里传播。

socket API原本是为网络通讯设计的，但后来在socket的框架上发展出一种IPC机制，就是UNIXDomain Socket。虽然网络socket也可用于同一台主机的进程间通讯（通过loopback地址127.0.0.1），但是UNIX Domain Socket用于IPC更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。这是因为，IPC机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。UNIX Domain Socket也提供面向流和面向数据包两种API接口，类似于TCP和UDP，但是面向消息的UNIX Domain Socket也是可靠的，消息既不会丢失也不会顺序错乱。

UNIX Domain Socket是全双工的，API接口语义丰富，相比其它IPC机制有明显的优越性，目前已成为使用最广泛的IPC机制，比如X Window服务器和GUI程序之间就是通过UNIX Domain Socket通讯的。

使用UNIX Domain Socket的过程和网络socket十分相似，也要先调用socket()创建一个socket文件描述符，address family指定为AF\_UNIX，type可以选择SOCK\_DGRAM或SOCK\_STREAM，protocol参数仍然指定为0即可。

UNIX Domain Socket与网络socket编程最明显的不同在于地址格式不同，用结构体sockaddr\_un表示，网络编程的socket地址是IP地址加端口号，而UNIX Domain Socket的地址是一个socket类型的文件在文件系统中的路径，这个socket文件由bind()调用创建，如果调用bind()时该文件已存在，则bind()错误返回。

今天我们介绍如何编写Linux下的TCP程序，关于UDP程序可以参考这里：

<http://blog.csdn.net/htttw/article/details/7519971>

**本文绝大部分是参考《Linux程序设计(第4版)》的第15章套接字**

**服务器端的步骤如下：**

**1. socket**：      建立一个socket

**2. bind**：          将这个socket绑定在某个文件上（AF\_UNIX）或某个端口上（AF\_INET），我们会分别介绍这两种。

**3. listen**：        开始监听

**4. accept**：      如果监听到客户端连接，则调用accept接收这个连接并同时新建一个socket来和客户进行通信

**5. read/write**：读取或发送数据到客户端

**6. close**：        通信完成后关闭socket

**客户端的步骤如下：**

**1. socket**：      建立一个socket

**2. connect**：   主动连接服务器端的某个文件（AF\_UNIX）或某个端口（AF\_INET）

**3. read/write**：如果服务器同意连接（accept），则读取或发送数据到服务器端

**4. close**：        通信完成后关闭socket

上面是整个流程，我们先给出一个例子，具体分析会在之后给出。**例子实现的功能是客户端发送一个字符到服务器，服务器将这个字符+1后送回客户端，客户端再把它打印出来**：

**Makefile：**

**[plain]** [view plaincopy](http://blog.csdn.net/htttw/article/details/7519964)

1. all: tcp\_client.c tcp\_server.c
2. gcc -g -Wall -o tcp\_client tcp\_client.c
3. gcc -g -Wall -o tcp\_server tcp\_server.c
5. clean:
6. rm -rf \*.o tcp\_client tcp\_server

**tcp\_server.c：**

**[cpp]** [view plaincopy](http://blog.csdn.net/htttw/article/details/7519964)

1. #include <sys/types.h>
2. #include <sys/socket.h>
3. #include <sys/un.h>
4. #include <unistd.h>
5. #include <stdlib.h>
6. #include <stdio.h>
8. **int** main()
9. {
10. /\* delete the socket file \*/
11. unlink("server\_socket");
13. /\* create a socket \*/
14. **int** server\_sockfd = socket(AF\_UNIX, SOCK\_STREAM, 0);
16. **struct** sockaddr\_un server\_addr;
17. server\_addr.sun\_family = AF\_UNIX;
18. strcpy(server\_addr.sun\_path, "server\_socket");
20. /\* bind with the local file \*/
21. bind(server\_sockfd, (**struct** sockaddr \*)&server\_addr, **sizeof**(server\_addr));
23. /\* listen \*/
24. listen(server\_sockfd, 5);
26. **char** ch;
27. **int** client\_sockfd;
28. **struct** sockaddr\_un client\_addr;
29. socklen\_t len = **sizeof**(client\_addr);
30. **while**(1)
31. {
32. printf("server waiting:\n");
34. /\* accept a connection \*/
35. client\_sockfd = accept(server\_sockfd, (**struct** sockaddr \*)&client\_addr, &len);
37. /\* exchange data \*/
38. read(client\_sockfd, &ch, 1);
39. printf("get char from client: %c\n", ch);
40. ++ch;
41. write(client\_sockfd, &ch, 1);
43. /\* close the socket \*/
44. close(client\_sockfd);
45. }
47. **return** 0;
48. }

**tcp\_client.c：**

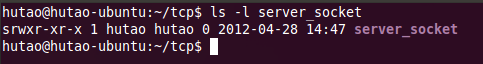
**[cpp]** [view plaincopy](http://blog.csdn.net/htttw/article/details/7519964)

1. #include <sys/types.h>
2. #include <sys/socket.h>
3. #include <sys/un.h>
4. #include <unistd.h>
5. #include <stdlib.h>
6. #include <stdio.h>
8. **int** main()
9. {
10. /\* create a socket \*/
11. **int** sockfd = socket(AF\_UNIX, SOCK\_STREAM, 0);
13. **struct** sockaddr\_un address;
14. address.sun\_family = AF\_UNIX;
15. strcpy(address.sun\_path, "server\_socket");
17. /\* connect to the server \*/
18. **int** result = connect(sockfd, (**struct** sockaddr \*)&address, **sizeof**(address));
19. **if**(result == -1)
20. {
21. perror("connect failed: ");
22. exit(1);
23. }
25. /\* exchange data \*/
26. **char** ch = 'A';
27. write(sockfd, &ch, 1);
28. read(sockfd, &ch, 1);
29. printf("get char from server: %c\n", ch);
31. /\* close the socket \*/
32. close(sockfd);
34. **return** 0;
35. }

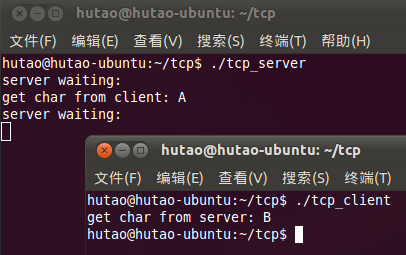
如果我们首先运行tcp\_client，会提示没有这个文件：

http://my.csdn.net/uploads/201204/28/1335595740_7456.png

因为我们是**以AF\_UNIX方式进行通信的，这种方式是通过文件来将服务器和客户端连接起来的，因此我们应该先运行tcp\_server，创建这个文件，默认情况下，这个文件会创建在当前目录下，并且第一个s表示它是一个socket文件**：



程序运行的结果如下图：



下面我们详细讲解：

**1.**

我们调用socket函数创建一个socket：

**int socket(int domain, int type, int protocol)**

**domain：**指定socket所属的域，常用的是AF\_UNIX或AF\_INET

AF\_UNIX表示以文件方式创建socket，AF\_INET表示以端口方式创建socket（我们会在后面详细讲解AF\_INET）

**type：**指定socket的类型，可以是SOCK\_STREAM或SOCK\_DGRAM

**SOCK\_STREAM**表示创建一个有序的，可靠的，面向连接的socket，因此如果我们要使用**TCP**，就应该指定为SOCK\_STREAM

**SOCK\_DGRAM**表示创建一个不可靠的，无连接的socket，因此如果我们要使用**UDP**，就应该指定为SOCK\_DGRAM

**protocol：**指定socket的协议类型，我们一般指定为0表示由第一第二两个参数自动选择。

**socket()函数返回新创建的socket，出错则返回-1**

**2.**

地址格式：

**常用的有两种socket域：AF\_UNIX或AF\_INET，因此就有两种地址格式：sockaddr\_un和sockaddr\_in，分别定义如下：**

**[cpp]** [view plaincopy](http://blog.csdn.net/htttw/article/details/7519964)

1. **struct** sockaddr\_un
2. {
3. sa\_family\_t sun\_family;  /\* AF\_UNIX \*/
4. **char** sun\_path[];         /\* pathname \*/
5. }

8. **struct** sockaddr\_in
9. {
10. **short** **int** sin\_family;          /\* AF\_INET \*/
11. unsigned **short** **int** sin\_port;   /\* port number \*/
12. **struct** in\_addr sin\_addr;       /\* internet address \*/
13. }

**其中in\_addr正是用来描述一个ip地址的：**

**[cpp]** [view plaincopy](http://blog.csdn.net/htttw/article/details/7519964)

1. **struct** in\_addr
2. {
3. unsigned **long** **int** s\_addr;
4. }

从上面的定义我们可以看出，**sun\_path存放socket的本地文件名，sin\_addr存放socket的ip地址，sin\_port存放socket的端口号**。

**3.**

创建完一个socket后，我们需要使用bind将其绑定：

**int bind(int socket, const struct sockaddr \* address, size\_t address\_len)**

**如果我们使用AF\_UNIX来创建socket，相应的地址格式是sockaddr\_un，而如果我们使用AF\_INET来创建socket，相应的地址格式是sockaddr\_in，因此我们需要将其强制转换为sockaddr这一通用的地址格式类型，而sockaddr\_un中的sun\_family和sockaddr\_in中的sin\_family分别说明了它的地址格式类型，因此bind()函数就知道它的真实的地址格式。第三个参数address\_len则指明了真实的地址格式的长度。**

**bind()函数正确返回0，出错返回-1**

**4.**

接下来我们需要开始监听了：

**int listen(int socket, int backlog)**

backlog：等待连接的最大个数，如果超过了这个数值，则后续的请求连接将被拒绝

**listen()函数正确返回0，出错返回-1**

**5.**

接受连接：

**int accept(int socket, struct sockaddr \* address, size\_t \* address\_len)**

同样，第二个参数也是一个通用地址格式类型，这意味着我们需要进行强制类型转化

**这里需要注意的是，address是一个传出参数，它保存着接受连接的客户端的地址，如果我们不需要，将address置为NULL即可。**

**address\_len：我们期望的地址结构的长度，注意，这是一个传入和传出参数，传入时指定我们期望的地址结构的长度，如果多于这个值，则会被截断，而当accept()函数返回时，address\_len会被设置为客户端连接的地址结构的实际长度。**

另外如果没有客户端连接时，accept()函数会阻塞

**accept()函数成功时返回新创建的socket描述符，出错时返回-1**

**6.**

客户端通过connect()函数与服务器连接：

**int connect(int socket, const struct sockaddr \* address, size\_t address\_len)**

对于第二个参数，我们同样需要强制类型转换

address\_len指明了地址结构的长度

**connect()函数成功时返回0，出错时返回-1**

**7.**

**双方都建立连接后，就可以使用常规的read/write函数来传递数据了**

**8.**

通信完成后，我们需要关闭socket：

**int close(int fd)**

close是一个通用函数（和read，write一样），不仅可以关闭文件描述符，还可以关闭socket描述符