

Socket 编程指南

说明一下先：次教程要求读者需要具有网络方面的基础只是，起码知道什么事 TCP/UDP 和数据包等等的内容啦^_^!! 基于本教程，本人也写了一本电子版的，有兴趣的同学联系我啦！！不过还是觉得 PDF 最好了 ^_^! QQ: 330974900

正文

1.什么是 socket?

相信同学朋友们经常听到人们谈论着“socket”，或许你还不知道它的确切含义。现在让我告诉你：它是使用 标准 Unix 文件描述符 (file descriptor) 和其它程序通讯的方式。什么？你也许听到一些 Unix 高手(hacker)这样说过：“呀，Unix 中的一切就是文件！”那个家伙也许正在说到一个事实：Unix 程序在执行任何形式的 I/O 的时候，程序是在读或者写一个文件描述符。一个文件描述符只是一个和打开的文件相关联的整数。但是(注意后面的话)，这个文件可能是一个网络连接，FIFO，管道，终端，磁盘上的文件或者什么其它的东西。Unix 中所有的东西就是文件！所以，你想和 Internet 上别的程序通讯的时候，你将要使用到文件描述符。你必须理解刚才的话。现在你脑海中或许冒出这样的念头：“那么我从哪里得到网络通讯的文件描述符呢？”，这个问题无论如何我都要回答：你利用系统调用 socket()，它返回套接字描述符 (socket descriptor)，然后你再通过它来进行 send() 和 recv() 调用。“但是...”，你可能有很大的疑惑，“如果它是个文件描述符，那么为什么不用一般调用 read()和 write()来进行套接字通讯？”简单的答案是：“你可以使用！”。详细的答案是：“你可以，但是使用 send()和 recv()让你更好的控制数据传输。”存在这样一个情况：在我们的世界上，有很多种套接字。有 DARPA Internet 地址 (Internet 套接字)，本地节点的路径名 (Unix 套接字)，CCITT X.25 地址 (你可以将 X.25 套接字完全忽略)。也许在你的 Unix 机器上还有其它的。我们在这里只讲第一种：Internet 套接字。

Internet 套接字的两种类型

什么意思？有两种类型的 Internet 套接字？是的。不，我在撒谎。其实还有很多，但是我可不想吓着你。我们这里只讲两种。除了这些，我打算另外介绍的 "Raw Sockets" 也是非常强大的，很值得查阅。

那么这两种类型是什么呢？一种是 "Stream Sockets"（流格式），另外一种是 "Datagram Sockets"（数据包格式）。我们以后谈到它们的时候也会用到 "SOCK_STREAM" 和

"SOCK_DGRAM"。数据报套接字有时也叫“无连接套接字”(如果你确实要连接的时候可以用 connect()。) 流式套接字是可靠的双向通讯的数据流。如果你向套接字按顺序输出“1, 2”, 那么它们将按顺序“1, 2”到达另一边。它们是无错误的传递的, 有自己的错误控制, 在此不讨论。

有什么在使用流式套接字? 你可能听说过 telnet, 不是吗? 它就使用流式套接字。你需要你所输入的字符按顺序到达, 不是吗? 同样, WWW 浏览器使用的 HTTP 协议也使用它们来下载页面。实际上, 当你通过端口 80 telnet 到一个 WWW 站点, 然后输入 “GET pagename” 的时候, 你也可以得到 HTML 的内容。为什么流式套接字可以达到高质量的数据传输? 这是因为它使用了“传输控制协议 (The Transmission Control Protocol)”, 也叫 “TCP” (请参考 RFC-793 获得详细资料。) TCP 控制你的数据按顺序到达并且没有错误。你也许听到 “TCP” 是因为听到过 “TCP/IP”。这里的 IP 是指“Internet 协议”(请参考 RFC-791。) IP 只是处理 Internet 路由而已。

那么数据报套接字呢? 为什么它叫无连接呢? 为什么它是不可靠的呢? 有这样的一些事实: 如果你发送一个数据报, 它可能会到达, 它可能次序颠倒了。如果它到达, 那么在这个包的内部是无错误的。数据报也使用 IP 作路由, 但是它不使用 TCP。它使用“用户数据报协议 (User Datagram Protocol)”, 也叫 “UDP” (请参考 RFC-768。)

为什么它们是无连接的呢? 主要是因为它并不象流式套接字那样维持一个连接。你只要建立一个包, 构造一个有目标信息的 IP 头, 然后发出去。无需连接。它们通常使用于传输包-包信息。简单的应用程序有: tftp, bootp 等等。

你也许会想: “假如数据丢失了这些程序如何正常工作?” 我的朋友, 每个程序在 UDP 上有自己的协议。例如, tftp 协议每发出的一个被接受到包, 收到者必须发回一个包来说“我收到了!” (一个“命令正确应答”也叫“ACK”包)。如果在一定时间内(例如 5 秒), 发送方没有收到应答, 它将重新发送, 直到得到 ACK。这一 ACK 过程在实现 SOCK_DGRAM 应用程序的时候非常重要。

网络理论

既然我刚才提到了协议层, 那么现在是讨论网络究竟如何工作和一些 关于 SOCK_DGRAM 包是如何建立的例子。当然, 你也可以跳过这一段, 如果你认为已经熟悉的话。

现在是学习数据封装 (Data Encapsulation) 的时候了! 它非常非常重要。它重要性重要到

你在网络课程学（图 1：数据封装）习中无论如何也得也得掌握它。主要 的内容是：一个包，先是被第一个协议(在这里是 **TFTP**)在它的报头（也许 是报尾）包装(“封装”)，然后，整个数据(包括 **TFTP** 头)被另外一个协议 (在这里是 **UDP**)封装，然后下一个(**IP**)，一直重复下去，直到硬件(物理) 层(这里是以太网)。

当另外一台机器接收到包，硬件先剥去以太网头，内核剥去 **IP** 和 **UDP** 头，**TFTP** 程序再剥去 **TFTP** 头，最后得到数据。现在我们终于讲到声名狼藉的网络分层模型 (**Layered Network Model**)。这种网络模型在描述网络系统上相对其它模型有很多优点。例如， 你可以写一个套接字程序而不用关心数据的物理传输(串行口，以太网，连 接单元接口 (**AUI**) 还是其它介质)，因为底层的程序会为你处理它们。实际 的网络硬件和拓扑对于程序员来说是透明的。

不说其它废话了，我现在列出整个层次模型。如果你要参加网络考试，可一定要记住：

应用层 (**Application**)

表示层 (**Presentation**)

会话层 (**Session**)

传输层(**Transport**)

网络层(**Network**)

数据链路层(**Data Link**)

物理层(**Physical**)

物理层是硬件(串口，以太网等等)。应用层是和硬件层相隔最远的--它 是用户和网络交互的地方。

这个模型如此通用，如果你想，你可以把它作为修车指南。把它对应 到 **Unix**，结果是：

应用层(**Application Layer**) (**telnet**, **ftp**,等等)

传输层(**Host-to-Host Transport Layer**) (**TCP**, **UDP**)

Internet 层(**Internet Layer**) (**IP** 和路由)

网络访问层 (**Network Access Layer**) (网络层，数据链路层和物理层)

现在，你可能看到这些层次如何协调来封装原始的数据了。

看看建立一个简单的数据包有多少工作？对于流式套接字你要作的是 **send()** 发 送数据。

对于数据报式套接字，你按照你选择的方式封装数据然后使用 **sendto()**。内核将为你建立

传输层和 Internet 层，硬件完成网络访问层。这就是现代科技。

结构体

终于谈到编程了。在这章，我将谈到被套接字用到的各种数据类型。因为它们中的一些内容很重要了。

首先是简单的一个：**socket** 描述符。它是下面的类型：

int

仅仅是一个常见的 **int**。

从现在起，事情变得不可思议了，而你所需做的就是继续看下去。注意这样的事实：有两种字节排列顺序：重要的字节（有时叫 "octet"，即八位位组）在前面，或者不重要的字节在前面。前一种叫“网络字节顺序 (Network Byte Order)”。有些机器在内部是按照这个顺序储存数据，而另外一些则不然。当我说某数据必须按照 NBO 顺序，那么你要调用函数(例如 **htons()**)来将它从本机字节顺序 (Host Byte Order) 转换过来。如果我没有提到 NBO，那么就让它保持本机字节顺序。

我的第一个结构(在这个技术手册 TM 中)--**struct sockaddr**。这个结构为许多类型的套接字储存套接字地址信息：

```
struct sockaddr {  
    unsigned short sa_family; /* 地址家族, AF_xxx */  
    char sa_data[14]; /*14 字节协议地址*/  
};
```

sa_family 能够是各种各样的类型，但是在这篇文章中都是 "AF_INET"。 **sa_data** 包含套接字中的目标地址和端口信息。这好像有点不明智。

为了处理 **struct sockaddr**，程序员创造了一个并列的结构：**struct sockaddr_in** ("in" 代表 "Internet"。)

```
struct sockaddr_in {  
    short int sin_family; /* 通信类型 */  
    unsigned short int sin_port; /* 端口 */  
    struct in_addr sin_addr; /* Internet 地址 */  
    unsigned char sin_zero[8]; /* 与 sockaddr 结构的长度相同*/  
};
```

用这个数据结构可以轻松处理套接字地址的基本元素。注意 `sin_zero` (它被加入到这个结构, 并且长度和 `struct sockaddr` 一样) 应该使用函数 `bzero()` 或 `memset()` 来全部置零。同时, 这一重要的字节, 一个指向 `sockaddr_in` 结构体的指针也可以被指向结构体 `sockaddr` 并且代替它。这样的话即使 `socket()` 想要的是 `struct sockaddr *`, 你仍然可以使用 `struct sockaddr_in`, 并且在最后转换。同时, 注意 `sin_family` 和 `struct sockaddr` 中的 `sa_family` 一致并能够设置为 `"AF_INET"`。最后, `sin_port` 和 `sin_addr` 必须是网络字节顺序 (Network Byte Order)!

你也许会反对道: "但是, 怎么让整个数据结构 `struct in_addr sin_addr` 按照网络字节顺序呢?" 要知道这个问题的答案, 我们就要仔细的看一看这个数据结构: `struct in_addr`, 有这样一个联合 (unions):

```
/* Internet 地址 (一个与历史有关的结构) */
```

```
    struct in_addr {
        unsigned long s_addr;
    };
```

它曾经是个最坏的联合, 但是现在那些日子过去了。如果你声明 `"ina"` 是数据结构 `struct sockaddr_in` 的实例, 那么 `"ina.sin_addr.s_addr"` 就储存 4 字节的 IP 地址(使用网络字节顺序)。如果你不幸的系统使用的还是恐怖的联合 `struct in_addr`, 你还是可以放心 4 字节的 IP 地址并且和上面我说的一样(这是因为使用了`"#define"`。)

本机转换

我们现在到了新的章节。我们曾经讲了很多网络到本机字节顺序的转换, 现在可以实践了!

你能够转换两种类型: `short` (两个字节)和 `long` (四个字节)。这个函数对于变量类型 `unsigned` 也适用。假设你想将 `short` 从本机字节顺序转换为网络字节顺序。用 `"h"` 表示 "本机 (host)", 接着是 `"to"`, 然后用 `"n"` 表示 "网络 (network)", 最后用 `"s"` 表示 `"short"`: `h-to-n-s`, 或者 `htons()` ("Host to Network Short")。

太简单了...

如果不是太傻的话, 你一定想到了由`"n"`, `"h"`, `"s"`, 和 `"l"`形成的正确组合, 例如这里肯定没有 `stohl()` ("Short to Long Host") 函数, 不仅在这里没有, 所有场合都没有。但是这里有: `htons()`--"Host to Network Short"

htonl()--"Host to Network Long"

ntohs()--"Network to Host Short"

ntohl()--"Network to Host Long"

现在，你可能想你已经知道它们了。你也可能想：“如果我想改变 char 的顺序要怎么办呢？”但是你也也许马上就想到，“用不着考虑的”。你也也许 会想到：我的 68000 机器已经使用了网络字节顺序，我没有必要去调用 htonl() 转换 IP 地址。你可能是对的，但是当你移植你的程序到别的机器 上的时候，你的程序将失败。可移植性！这里是 Unix 世界！记住：在你 将数据放到网络上的时候，确信它们是网络字节顺序的。

最后一点：为什么在数据结构 struct sockaddr_in 中， sin_addr 和 sin_port 需要转换为网络字节顺序，而 sin_family 需不需要呢？答案是： sin_addr 和 sin_port 分别封装在包的 IP 和 UDP 层。因此，它们必须要是网络字节顺序。但是 sin_family 域只是被内核 (kernel) 使用来决定在数 据结构中包含什么类型的地址，所以它必须是本机字节顺序。同时， sin_family 没有发送到网络上，它们可以是本机字节顺序。

IP 地址和如何处理它们

现在我们很幸运，因为我们有很多的函数来方便地操作 IP 地址。没有 必要用手工计算它们，也没有必要用"<<"操作来储存成长整字型。 首先，假设你已经有了一个 sockaddr_in 结构体 ina，你有一个 IP 地址"132.241.5.10"要储存在其中，你就要用到函数 inet_addr()，将 IP 地址从 点数格式转换成无符号长整型。使用方法如下：

```
ina.sin_addr.s_addr = inet_addr("132.241.5.10");
```

注意，inet_addr()返回的地址已经是网络字节格式，所以你无需再调用 函数 htonl()。

我们现在发现上面的代码片断不是十分完整的，因为它没有错误检查。 显而易见，当 inet_addr()发生错误时返回-1。记住这些二进制数字(无符号数)-1 仅仅和 IP 地址 255.255.255.255 相符合！这可是广播地址！大错特错！记住要先进行错误检查。

好了，现在你可以将 IP 地址转换成长整型了。有没有其相反的方法呢？ 它可以将一个 in_addr 结构体输出成点数格式？这样的话，你就要用到函数 inet_ntoa()("ntoa"的含义是 "network to ascii")，就像这样：

```
printf("%s",inet_ntoa(ina.sin_addr));
```

它将输出 IP 地址。需要注意的是 inet_ntoa()将结构体 in-addr 作为一 个参数，不是长整形。同样需要注意的是它返回的是一个指向一个字符的 指针。它是一个由 inet_ntoa()控制的静

态的固定的指针，所以每次调用 `inet_ntoa()`，它就将覆盖上次调用时所得的 IP 地址。例如：

```
char *a1, *a2;

.

a1 = inet_ntoa(ina1.sin_addr); /* 这是 198.92.129.1 */
a2 = inet_ntoa(ina2.sin_addr); /* 这是 132.241.5.10 */

printf("address 1: %s\n", a1);
printf("address 2: %s\n", a2);
```

输出如下：

```
address 1: 132.241.5.10
address 2: 132.241.5.10
```

假如你需要保存这个 IP 地址，使用 `strcpy()` 函数来指向你自己的字符 指针。

上面就是关于这个主题的介绍。稍后，你将学习将一个类 似 "wintehouse.gov" 的字符串转 换成它所对应的 IP 地址(查阅域名服务,稍 后)。

socket() 函数

我想我不能再不提这个了。下面我将讨论一下 `socket()` 系统调用。

下面是详细介绍：

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

但是它们的参数是什么？首先，`domain` 应该设置成 "AF_INET"，就 象上面的数据结构 `struct sockaddr_in` 中一样。然后，参数 `type` 告诉内核 是 `SOCK_STREAM` 类型还是 `SOCK_DGRAM` 类型。最后，把 `protocol` 设置为 "0"。(注意：有很多种 `domain`、`type`， 我不可能一一列出了，请看 `socket()` 的 man 帮助。当然，还有一个"更好"的方式去得到 `protocol`。同 时请查阅 `getprotobyname()` 的 man 帮助。)

`socket()` 只是返回你以后在系统调用种可能用到的 `socket` 描述符，或 者在错误的时候返 回 -1。全局变量 `errno` 中将储存返回的错误值。(请参考 `perror()` 的 man 帮助。)

bind() 函数

一旦你有一个套接字，你可能要将套接字和机器上的一定的端口关联起来。(如果你想

用 `listen()` 来侦听一定端口的数据，这是必要一步--MUD 告诉你用命令 "telnet x.y.z 6969"。)如果你只想用 `connect()`，那么这个步骤没有必要。但是无论如何，请继续读下去。

这里是系统调用 `bind()` 的大概：

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

`sockfd` 是调用 `socket` 返回的文件描述符。`my_addr` 是指向数据结构 `struct sockaddr` 的指针，它保存你的地址(即端口和 IP 地址) 信息。`addrlen` 设置为 `sizeof(struct sockaddr)`。简单得很不是吗？再看看例子：

```
#include <string.h>
```

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
#define MYPORT 3490
```

```
main()
```

```
{
```

```
    int sockfd;
```

```
    struct sockaddr_in my_addr;
```

```
    sockfd = socket(AF_INET, SOCK_STREAM, 0); /*需要错误检查 */
```

```
    my_addr.sin_family = AF_INET; /* host byte order */
```

```
        my_addr.sin_port = htons(MYPORT); /* short, network byte order */
```

```
        my_addr.sin_addr.s_addr = inet_addr("132.241.5.10");
```

```
        bzero(&(my_addr.sin_zero), /* zero the rest of the struct */
```

```
/* don't forget your error checking for bind(): */
```

```
        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
```

```
    .
```

这里也有要注意的几件事情。`my_addr.sin_port` 是网络字节顺序，

`my_addr.sin_addr.s_addr` 也是的。另外要注意到的事情是因系统的不同， 包含的头文件也不尽相同，请查阅本地的 `man` 帮助文件。

在 `bind()` 主题中最后要说的话是，在处理自己的 IP 地址和/或端口的时候，有些工作是

可以自动处理的。

```
my_addr.sin_port = 0; /* 随机选择一个没有使用的端口 */
```

```
    my_addr.sin_addr.s_addr = INADDR_ANY; /* 使用自己的 IP 地址 */
```

通过将 0 赋给 `my_addr.sin_port`，你告诉 `bind()` 自己选择合适的端口。同样，将

`my_addr.sin_addr.s_addr` 设置为 `INADDR_ANY`，你告诉它自动填上它所运行的机器的 IP 地址。

如果你一向小心谨慎，那么你可能注意到我没有将 `INADDR_ANY` 转换为网络字节顺序！这是因为我知道内部的东西：`INADDR_ANY` 实际上就是 0！即使你改变字节的顺序，0 依然是 0。但是完美主义者说应该处处一致，`INADDR_ANY` 或许是 12 呢？你的代码就不能工作了，那么就看看下面的代码：

```
my_addr.sin_port = htons(0); /* 随机选择一个没有使用的端口 */
```

```
my_addr.sin_addr.s_addr = htonl(INADDR_ANY); /* 使用自己的 IP 地址 */
```

你或许不相信，上面的代码可以随便移植。我只是想指出，既然你所遇到的程序不会都运行使用 `htonl` 的 `INADDR_ANY`。

`bind()` 在错误的时候依然是返回 -1，并且设置全局错误变量 `errno`。

在你调用 `bind()` 的时候，你要小心的另一件事情是：不要采用小于 1024 的端口号。所有小于 1024 的端口号都被系统保留！你可以选择从 1024 到 65535 的端口(如果它们没有被别的程序使用的话)。

你要注意的另外一件小事是：有时候你根本不需要调用它。如果你使用 `connect()` 来和远程机器进行通讯，你不需要关心你的本地端口号(就象你在使用 `telnet` 的时候)，你只要简单的调用 `connect()` 就可以了，它会检查套接字是否绑定端口，如果没有，它会自己绑定一个没有使用的本地端口。

`connect()` 程序

现在我们假设你是个 `telnet` 程序。你的用户命令你得到套接字的文件描述符。你听从命令调用了 `socket()`。下一步，你的用户告诉你通过端口 23(标准 `telnet` 端口)连接到 "132.241.5.10"。你该怎么做呢？幸运的是，你正在阅读 `connect()`--如何连接到远程主机这一章。你可不想让你的用户失望。

`connect()` 系统调用是这样的：

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

`sockfd` 是系统调用 `socket()` 返回的套接字文件描述符。`serv_addr` 是保存着目的地端口和 IP 地址的数据结构 `struct sockaddr`。`addrlen` 设置为 `sizeof(struct sockaddr)`。

想知道得更多吗？让我们来看个例子：

```
#include <string.h>
```

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
#define DEST_IP "132.241.5.10"
```

```
    #define DEST_PORT 23
```

```
main()
```

```
{
```

```
int sockfd;
```

```
struct sockaddr_in dest_addr; /* 目的地址*/
```

```
sockfd = socket(AF_INET, SOCK_STREAM, 0); /* 错误检查 */
```

```
dest_addr.sin_family = AF_INET; /* host byte order */
```

```
dest_addr.sin_port = htons(DEST_PORT); /* short, network byte order */
```

```
dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
```

```
bzero(&(dest_addr.sin_zero),; /* zero the rest of the struct */
```

```
/* don't forget to error check the connect()! */
```

```
connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
```

```
.
```

再一次，你应该检查 `connect()` 的返回值--它在错误的时候返回-1，并设置全局错误变量 `errno`。

同时，你可能看到，我没有调用 `bind()`。因为我不在乎本地的端口号。我只关心我要去那。内核将为我选择一个合适的端口号，而我们所连接的地方也自动地获得这些信息。一切都不用担心。

`listen()`函数

是换换内容的时候了。假如你不希望与远程的一个地址相连，或者说，仅仅是将它踢

开，那你就需要等待接入请求并且用各种方法处理它们。处理过程分两步：首先，你听--listen()，然后，你接受--accept() (请看下面的内容)。

除了要一点解释外，系统调用 listen 也相当简单。

```
int listen(int sockfd, int backlog);
```

sockfd 是调用 socket() 返回的套接字文件描述符。backlog 是在进入队列中允许的连接数目。什么意思呢？进入的连接是在队列中一直等待直到你接受 (accept() 请看下面的文章)连接。它们的数目限制于队列的允许。大多数系统的允许数目是 20，你也可以设置为 5 到 10。

和别的函数一样，在发生错误的时候返回-1，并设置全局错误变量 errno。

你可能想象到了，在你调用 listen() 前你或者要调用 bind() 或者让内核随便选择一个端口。如果你想侦听进入的连接，那么系统调用的顺序可能是这样的：

```
socket();
```

```
    bind();
```

```
listen();
```

```
    /* accept() 应该在这 */
```

因为它相当的明了，我将在这里不给出例子了。(在 accept() 那一章的代码将更加完全。)

真正麻烦的部分在 accept()。

accept()函数

准备好了，系统调用 accept() 会有点古怪的地方的！你可以想象发生这样的事情：有人从很远的地方通过一个你在侦听 (listen()) 的端口连接 (connect()) 到你的机器。它的连接将加入到等待接受 (accept()) 的队列中。你调用 accept() 告诉它你有空闲的连接。它将返回一个新的套接字文件描述符！这样你就有两个套接字了，原来的一个还在侦听你的那个端口，新的在准备发送 (send()) 和接收 (recv()) 数据。这就是这个过程！

函数是这样定义的：

```
#include <sys/socket.h>
```

```
int accept(int sockfd, void *addr, int *addrlen);
```

sockfd 相当简单，是和 listen() 中一样的套接字描述符。addr 是个指向局部的数据结构 sockaddr_in 的指针。这是要求接入的信息所要去的地方（你可以测定那个地址在那个端口呼叫你）。在它的地址传递给 accept 之前，addrlen 是个局部的整形变量，设置为

sizeof(struct sockaddr_in)。 accept 将不会将多余的字节给 addr。如果你放入的少些，那么它会通过改

变 addrlen 的值反映出来。

同样，在错误时返回-1，并设置全局错误变量 errno。

现在是你应该熟悉的代码片段。

```
#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#define MYPORT 3490 /*用户接入端口*/

#define BACKLOG 10 /* 多少等待连接控制*/

main()
{
    int sockfd, new_fd; /* listen on sock_fd, new connection on new_fd */

    struct sockaddr_in my_addr; /* 地址信息 */

    struct sockaddr_in their_addr; /* connector's address information */

    int sin_size;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); /* 错误检查*/

    my_addr.sin_family = AF_INET; /* host byte order */

    my_addr.sin_port = htons(MYPORT); /* short, network byte order */

    my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */

    bzero(&(my_addr.sin_zero), /* zero the rest of the struct */

    /* don't forget your error checking for these calls: */

    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

    listen(sockfd, BACKLOG);

    sin_size = sizeof(struct sockaddr_in);

    new_fd = accept(sockfd, &their_addr, &sin_size);

    .
```

注意，在系统调用 send() 和 recv() 中你应该使用新的套接字描述符 new_fd。如果你只想让一个连接进来，那么你可以使用 close() 去关闭原来的文件描述符 sockfd 来避免同一个端口更多的连接。

send() and recv()函数

这两个函数用于流式套接字或者数据报套接字的通讯。如果你喜欢使用无连接的数据报套接字，你应该看一看下面关于 `sendto()` 和 `recvfrom()` 的章节。

`send()` 是这样的：

```
int send(int sockfd, const void *msg, int len, int flags);
```

`sockfd` 是你想发送数据的套接字描述符(或者是调用 `socket()` 或者是 `accept()` 返回的。) `msg` 是指向你发送的数据的指针。`len` 是数据的长度。把 `flags` 设置为 0 就可以了。(详细的资料请看 `send()` 的 man page)。

这里是一些可能的例子：

```
char *msg = "Beej was here!";

int len, bytes_sent;

.

len = strlen(msg);

bytes_sent = send(sockfd, msg, len, 0);

.
```

`send()` 返回实际发送的数据的字节数--它可能小于你要求发送的数目！注意，有时候你告诉它要发送一堆数据可是它不能处理成功。它只是发送它可能发送的数据，然后希望你能发送其它的数据。记住，如果 `send()` 返回的数据和 `len` 不匹配，你就应该发送其它的数据。但是这里也有个好消息：如果你要发送的包很小(小于大约 1K)，它可能处理让数据一次发送完。最后要说得就是，它在错误的时候返回-1，并设置 `errno`。

`recv()` 函数很相似：

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

`sockfd` 是要读的套接字描述符。`buf` 是要读的信息的缓冲。`len` 是缓冲的最大长度。`flags` 可以设置为 0。(请参考 `recv()` 的 man page。) `recv()` 返回实际读入缓冲的数据的字节数。或者在错误的时候返回-1，同时设置 `errno`。

很简单，不是吗？你现在可以在流式套接字上发送数据和接收数据了。你现在是 Unix 网络程序员了！

sendto() 和 recvfrom()函数

“这很不错啊”，你说，“但是你还没有讲无连接数据报套接字呢？”没问题，现在我们开始这个内容。

既然数据报套接字不是连接到远程主机的，那么在我们发送一个包之前需要什么信息呢？

不错，是目标地址！看看下面的：

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,  
           const struct sockaddr *to, int tolen);
```

你已经看到了，除了另外的两个信息外，其余的和函数 `send()` 是一样的。 `to` 是个指向数据结构 `struct sockaddr` 的指针，它包含了目的地的 IP 地址和端口信息。`tolen` 可以简单地设置为 `sizeof(struct sockaddr)`。和函数 `send()` 类似，`sendto()` 返回实际发送的字节数(它也可能小于你想要发送的字节数！)，或者在错误的时候返回 `-1`。

相似的还有函数 `recv()` 和 `recvfrom()`。`recvfrom()` 的定义是这样的：

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,  
             struct sockaddr *from, int  
             *fromlen);
```

又一次，除了两个增加的参数外，这个函数和 `recv()` 也是一样的。`from` 是一个指向局部数据结构 `struct sockaddr` 的指针，它的内容是源机器的 IP 地址和端口信息。`fromlen` 是个 `int` 型的局部指针，它的初始值为 `sizeof(struct sockaddr)`。函数调用返回后，`fromlen` 保存着实际储存在 `from` 中的地址的长度。

`recvfrom()` 返回收到的字节长度，或者在发生错误后返回 `-1`。

记住，如果你用 `connect()` 连接一个数据报套接字，你可以简单的调用 `send()` 和 `recv()` 来满足你的要求。这个时候依然是数据报套接字，依然使用 UDP，系统套接字接口会为你自动加上了目标和源的信息。

`close()`和 `shutdown()`函数

你已经整天都在发送 (`send()`) 和接收 (`recv()`) 数据了，现在你准备关闭你的套接字描述符了。这很简单，你可以使用一般的 Unix 文件描述符的 `close()` 函数：

```
close(sockfd);
```

它将防止套接字上更多的数据的读写。任何在另一端读写套接字的企图都将返回错误信息。如果你想在如何关闭套接字上有多一点的控制，你可以使用函数 `shutdown()`。它允许你将一定方向上的通讯或者双向的通讯(就象 `close()` 一样)关闭，你可以使用：

```
int shutdown(int sockfd, int how);
```

`sockfd` 是你想要关闭的套接字文件描述符。`how` 的值是下面的其中之一：

0 – 不允许接受

1 – 不允许发送

2 – 不允许发送和接受(和 `close()` 一样)

`shutdown()` 成功时返回 0，失败时返回 -1(同时设置 `errno`。)如果在无连接的数据报套接字中使用 `shutdown()`，那么只不过是让 `send()` 和 `recv()` 不能使用(记住你在数据报套接字中使用了 `connect` 后 是可以使用它们的)。

`getpeername()`函数

这个函数太简单了。

它太简单了，以至我都不想单列一章。但是我还是这样做了。函数 `getpeername()` 告诉你在连接的流式套接字上谁在另外一边。函数是这样的：

```
#include <sys/socket.h>
```

```
int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

`sockfd` 是连接的流式套接字的描述符。`addr` 是一个指向结构 `struct sockaddr` (或者是 `struct sockaddr_in`) 的指针，它保存着连接的另一边的信息。`addrlen` 是一个 `int` 型的指针，它初始化为 `sizeof(struct sockaddr)`。函数在错误的时候返回 -1，设置相应的 `errno`。一旦你获得它们的地址，你可以使用 `inet_ntoa()` 或者 `gethostbyaddr()` 来打印或者获得更多的信息。但是你不能得到它的帐号。(如果它运行着愚蠢的守护进程，这是可能的，但是它的讨论已经超出了本文的范围，请参考 RFC-1413 以获得更多的信息。)

`gethostname()`函数

甚至比 `getpeername()` 还简单的函数是 `gethostname()`。它返回你程序所运行的机器的主机名字。然后你可以使用 `gethostbyname()` 以获得你的机器的 IP 地址。

下面是定义：

```
#include <unistd.h>
```

```
int gethostname(char *hostname, size_t size);
```

参数很简单：`hostname` 是一个字符数组指针，它将在函数返回时保存主机名。`size` 是 `hostname` 数组的字节长度。

函数调用成功时返回 0，失败时返回 -1，并设置 `errno`。

域名服务 (DNS)

如果你不知道 DNS 的意思, 那么我告诉你, 它代表域名服务(Domain Name Service)。它主要的功能是: 你给它一个容易记忆的某站点的地址, 它给你 IP 地址(然后你就可以使用 `bind()`, `connect()`, `sendto()` 或者其它 函数) 。当一个人输入:

```
$ telnet whitehouse.gov
```

telnet 能知道它将连接 (`connect()`) 到 "198.137.240.100"。

但是这是如何工作的呢? 你可以调用函数 `gethostbyname()`:

```
#include <netdb.h>
```

```
struct hostent *gethostbyname(const char *name);
```

很明白的是, 它返回一个指向 `struct hostent` 的指针。这个数据结构 是这样的:

```
struct hostent {  
    char *h_name;  
    char **h_aliases;  
    int h_addrtype;  
    int h_length;  
    char **h_addr_list;  
};  
  
#define h_addr h_addr_list[0]
```

这里是这个数据结构的详细资料:

`struct hostent`:

`h_name` – 地址的正式名称。

`h_aliases` – 空字节-地址的预备名称的指针。

`h_addrtype` –地址类型; 通常是 `AF_INET`。

`h_length` – 地址的比特长度。

`h_addr_list` – 零字节-主机网络地址指针。网络字节顺序。

`h_addr` - `h_addr_list` 中的第一地址。

`gethostbyname()` 成功时返回一个指向结构体 `hostent` 的指针, 或者 是个空 (`NULL`) 指针。(但是和以前不同, 不设置 `errno`, `h_errno` 设置错误信息。请看下面的 `herror()`。)

但是如何使用呢? 有时候(我们可以从电脑手册中发现), 向读者灌输 信息是不够的。这

个函数可不象它看上去那么难用。

这里是个例子 1:

```
#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <netdb.h>

#include <sys/types.h>

#include <netinet/in.h>

int main(int argc, char *argv[])

{

    struct hostent *h;

    if (argc != 2) { /* 检查命令行 sockfd, new_fd; /* listen on sock_fd, new connection on

new_fd */

        struct sockaddr_in my_addr; /* 地址信息 */

        struct sockaddr_in their_addr; /* connector's address information */

        int sin_size;

        sockfd = socket(AF_INET, SOCK_STREAM, 0); /* 错误检查*/

        my_addr.sin_family = AF_INET; /* host byte order */

        my_addr.sin_port = htons(MYPORT); /* short, network byte order */

        my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */

        bzero(&(my_addr.sin_zero), /* zero the rest of the struct */

/* don't forget your error checking for these calls: */

        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

        listen(sockfd, BACKLOG);

        sin_size = sizeof(struct sockaddr_in);

        new_fd = accept(sockfd, &their_addr, &sin_size);

        .
```

注意，在系统调用 `send()` 和 `recv()` 中你应该使用新的套接字描述符 `new_fd`。如果你只想让一个连接进来，那么你可以使用 `close()` 去关闭原来的文件描述符 `sockfd` 来避免同一个端口更多的连接。

例子 2:

```
#include <stdio.h>
```

```
    #include <stdlib.h>
```

```
    #include <errno.h>
```

```
    #include <netdb.h>
```

```
    #include <sys/types.h>
```

```
    #include <netinet/in.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    struct hostent *h;
```

```
if (argc != 2) { /* 检查命令行 :
```

```
#include <stdio.h>
```

```
    #include <stdlib.h>
```

```
    #include <errno.h>
```

```
    #include <string.h>
```

```
    #include <sys/types.h>
```

```
    #include <netinet/in.h>
```

```
    #include <sys/socket.h>
```

```
    #include <sys/wait.h>
```

```
#define MYPORT 4950 /* the port users will be sending to */
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    int sockfd;
```

```
    struct sockaddr_in their_addr; /* connector's address information */
```

```
    struct hostent *he;
```

```
    int numbytes;
```

```
if (argc != 3) {
```

```
    fprintf(stderr, "usage: talker hostname message\n");
```

```

        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) { /* get the host info */
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET; /* host byte order */
    their_addr.sin_port = htons(MYPORT); /* short, network byte order */
    /*
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(their_addr.sin_zero), /* zero the rest of the struct */
    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0, \
        (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1) {
        perror("sendto");
        exit(1);
    }

    printf("sent %d bytes to\n", numbytes, inet_ntoa(their_addr.sin_addr));
    close(sockfd);
    return 0;
}

```

这就是所有的了。在一台机器上运行 **listener**，然后在另外一台机器上 运行 **talker**。观察它们的通讯！

除了一些我在上面提到的数据套接字连接的小细节外，对于数据套接字，我还得说一些，当一个讲话者呼叫 **connect()**函数时并指定接受者的地址时，从这点可以看出，讲话者只能

向 `connect()`函数指定的地址发送和接受信息。因此，你不需要使用 `sendto()`和 `recvfrom()`，你完全可以用 `send()` 和 `recv()`代替。

阻塞

阻塞，你也许早就听说了。“阻塞”是 “sleep” 的科技行话。你可能注意到前面运行的 `listener` 程序，它在那里不停地运行，等待数据包的到来。实际在运行的是它调用 `recvfrom()`，然后没有数据，因此 `recvfrom()` 说“阻塞 (block)”，直到数据的到来。很多函数都利用阻塞。`accept()` 阻塞，所有的 `recv*()` 函数阻塞。它们之所以能这样做是因为它们被允许这样做。当你第一次调用 `socket()` 建立套接字描述符的时候，内核就将它设置为阻塞。如果你不想套接字阻塞，你就要调用函数 `fcntl()`：

```
#include <unistd.h>
```

```
    #include <fcntl.h>
```

```
    .
```

```
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
    fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

```
    .
```

通过设置套接字为非阻塞，你能够有效地“询问”套接字以获得信息。如果你尝试从一个非阻塞的套接字读信息并且没有任何数据，它不允许阻塞--它将返回 `-1` 并将 `errno` 设置为 `EWOULDBLOCK`。

但是一般说来，这种询问不是个好主意。如果你让你的程序在忙等状态查询套接字的数据，你将浪费大量的 CPU 时间。更好的解决之道是用下一章讲的 `select()` 去查询是否有数据要读进来。

`select()`--多路同步 I/O

虽然这个函数有点奇怪，但是它很有用。假设这样的情况：你是个服务器，你一边在不停地从连接上读数据，一边在侦听连接上的信息。没问题，你可能会说，不就是一个 `accept()` 和两个 `recv()` 吗？这么容易吗，朋友？如果你在调用 `accept()` 的时候阻塞呢？你怎么能够同时接受 `recv()` 数据？“用非阻塞的套接字啊！”不行！你不想耗尽所有的 CPU 吧？那么，该如何是好？

`select()` 让你可以同时监视多个套接字。如果你想知道的话，那么它就会告诉你哪个套接

字准备读，哪个又准备写，哪个套接字又发生了例外 (exception)。

闲话少说，下面是 `select()`：

```
#include <sys/time.h>
```

```
    #include <sys/types.h>
```

```
    #include <unistd.h>
```

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);
```

这个函数监视一系列文件描述符，特别是 `readfds`、`writefds` 和 `exceptfds`。如果你想知道你是否能够从标准输入和套接字描述符 `sockfd` 读入数据，你只要将文件描述符 0 和 `sockfd` 加入到集合 `readfds` 中。参数 `numfds` 应该等于最高的文件描述符的值加 1。在这个例子中，你应该设置该值为 `sockfd+1`。因为它一定大于标准输入的文件描述符 (0)。当函数 `select()` 返回的时候，`readfds` 的值修改为反映你选择的哪个文件描述符可以读。你可以用下面讲到的宏 `FD_ISSET()` 来测试。在我们继续下去之前，让我来讲讲如何对这些集合进行操作。每个集合类型都是 `fd_set`。下面有一些宏来对这个类型进行操作：

`FD_ZERO(fd_set *set)` – 清除一个文件描述符集合

`FD_SET(int fd, fd_set *set)` - 添加 `fd` 到集合

`FD_CLR(int fd, fd_set *set)` – 从集合中移去 `fd`

`FD_ISSET(int fd, fd_set *set)` – 测试 `fd` 是否在集合中

最后，是有点古怪的数据结构 `struct timeval`。有时你可不想永远等待别人发送数据过来。也许什么事情都没有发生的时候你也想每隔 96 秒在终端上打印字符串 "Still Going...". 这个数据结构允许你设定一个时间，如果时间到了，而 `select()` 还没有找到一个准备好的文件描述符，它将返回让你继续处理。

数据结构 `struct timeval` 是这样的：

```
struct timeval {
    int tv_sec; /* seconds */
    int tv_usec; /* microseconds */
};
```

只要将 `tv_sec` 设置为你要等待的秒数，将 `tv_usec` 设置为你要等待的微秒数就可以了。是的，是微秒而不是毫秒。1,000 微秒等于 1 毫秒，1,000 毫秒等于 1 秒。也就是说，1 秒等于 1,000,000 微秒。为什么用符号 "usec" 呢？字母 "u" 很象希腊字母 μ ，而 μ 表

示 "微" 的意思。当然，函数 返回的时候 `timeout` 可能是剩余的时间，之所以是可能，是因为它依赖于 你的 `Unix` 操作系统。

哈！我们现在有一个微秒级的定时器！别计算了，标准的 `Unix` 系统 的时间片是 100 毫秒，所以无论如何设置你的数据结构 `struct timeval`，你都要等待那么长的时间。

还有一些有趣的事情：如果你设置数据结构 `struct timeval` 中的数据为 0，`select()` 将立即超时，这样就可以有效地轮询集合中的所有文件描述符。如果你将参数 `timeout` 赋值为 `NULL`，那么将永远不会发生超时，即 一直等到第一个文件描述符就绪。最后，如果你不是很关心等待多长时间，那么就把它赋为 `NULL` 吧。

下面的代码演示了在标准输入上等待 2.5 秒：

```
#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

#define STDIN 0 /* file descriptor for standard input */

main()
{
    struct timeval tv;

    fd_set readfds;

    tv.tv_sec = 2;

    tv.tv_usec = 500000;

    FD_ZERO(&readfds);

    FD_SET(STDIN, &readfds);

    /* don't care about writefds and exceptfds: */

    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");

    else
        printf("Timed out.\n");

}
```

如果你是在一个 `line buffered` 终端上，那么你敲的键应该是回车 (`RETURN`)，否则无论如何它都会超时。

现在，你可能回认为这就是在数据报套接字上等待数据的方式--你是对的：它可能是。有些 Unix 系统可以按这种方式，而另外一些则不能。你在尝试以前可能要先看看本系统的 man page 了。

最后一件关于 `select()` 的事情：如果你有一个正在侦听 (`listen()`) 的套接字，你可以通过将该套接字的文件描述符加入到 `readfds` 集合中来看是否有新的连接。

OK!!教程到这里基本上写完了！相信大家对 socket 这个东东了解好多了吧？！哈哈，那就太好了。

后记：

本来不怎么想写这个基础教程的，我也不是开发这个领域的软件。可是既然看了(其实有段时间无聊才看的啦!)，也想着以后可能有师弟师妹做这方面的开发，那就写一下,呵呵...这些内容上网络课的时候一般都不会讲的（除了会有时涉及一下..），所以要想了解的话只有自己去拓展了，而且本教程的内容也符合实际应用，所以觉得很有必要写一下。我知道网上也有类似教程，但是觉得还是要写，因为网上的可能会太专业了就难以接受了。。所以还是用学生的语言来解释对于学生来说容易接受点，，嘻嘻!!所以希望有兴趣的同学朋友都看看啦!!

Made By Rock.Z
ROZ Studio
6.18.2009

