**Linux 网络与系统编程和并发处理--------详解**

**文戈 钟鸿文 著**

**I, of course, believe I am simply enthusiastic!**

——**Randal E. Bryant**

[内容提要 7](#_Toc16626902)

[前言 8](#_Toc16626903)

[第一章 网络编程基础 8](#_Toc16626904)

[1.1 帮助的使用和基本配置 8](#_Toc16626905)

[1.1.1 man手册的使用 8](#_Toc16626906)

[1.1.2 服务器--客户机模式 9](#_Toc16626907)

[1.1.3 gcc编译工具 10](#_Toc16626908)

[1.1.4 环境配置 11](#_Toc16626909)

[1.2 套接字的数据类型 16](#_Toc16626910)

[1.3 套接字和字节序 17](#_Toc16626911)

[1.3.1 TCP/IP模型 17](#_Toc16626912)

[1.3.2 套接字 19](#_Toc16626913)

[1.3.3 服务器-客户机模式下的TCP/IP通信 20](#_Toc16626914)

[1.3.4字节序 21](#_Toc16626915)

[1.3.5 字节序转换函数hton和ntoh族 22](#_Toc16626916)

[1.3.6 结构体置零函数memset 24](#_Toc16626917)

[1.4 带点地址和二进制地址 26](#_Toc16626918)

[1.4.1 IP地址的结构体 26](#_Toc16626919)

[1.4.2 inet\_aton，\*inet\_ntoa和inet\_addr函数 27](#_Toc16626920)

[1.4.3 inet\_pton和inet\_ntop 35](#_Toc16626921)

[1.5 服务器的网络编程函数 38](#_Toc16626922)

[1.5.1 struct sockaddr类型的地址作用 38](#_Toc16626923)

[1.5.2 socket和bind函数 39](#_Toc16626924)

[1.5.3 listen和accept函数 40](#_Toc16626925)

[1.6 客户机通信的函数 41](#_Toc16626926)

[1.6.1 客户机连接函数connect 41](#_Toc16626927)

[1.6.2 send、recv和close函数 41](#_Toc16626928)

[1.7 发送固定消息的服务器 45](#_Toc16626929)

[1.7.1 服务器端实现 45](#_Toc16626930)

[1.7.2 客户端实现 47](#_Toc16626931)

[1.8 代码的改进 50](#_Toc16626932)

[1.8.1 服务器端代码改进 50](#_Toc16626933)

[1.8.2 客户机端代码改进 52](#_Toc16626934)

[1.8.3 errno和标准错误处理函数 54](#_Toc16626935)

[1.9 使用write和read函数写的文件复制程序 62](#_Toc16626936)

[1.9.1 open文件打开和读写函数read、write 62](#_Toc16626937)

[1.9.2 read和write完成文件的复制 65](#_Toc16626938)

[1.10 域名和IP地址 68](#_Toc16626939)

[1.10.1 使用域名获得IP地址gethostbyname 70](#_Toc16626940)

[1.10.2 使用IP地址获得域名gethostbyaddr 73](#_Toc16626941)

[1.11 小结 76](#_Toc16626942)

[第二章 TCP和UDP下的编程 76](#_Toc16626943)

[2.1 接收固定消息的服务器 77](#_Toc16626944)

[2.1.1 需要做的事情 77](#_Toc16626945)

[2.1.2 服务器代码 77](#_Toc16626946)

[2.1.3客户机代码 79](#_Toc16626947)

[2.1.4 运行测试 81](#_Toc16626948)

[2.2 TCP的数据边界 82](#_Toc16626949)

[2.2.1 TCP数据边界 82](#_Toc16626950)

[2.2.2 TCP数据边界的程序验证 82](#_Toc16626951)

[2.3 反射服务器 87](#_Toc16626952)

[2.3.1反射服务器代码 87](#_Toc16626953)

[2.3.2 客户端代码初步 90](#_Toc16626954)

[2.3.3 改进后的客户机代码 92](#_Toc16626955)

[2.4 网络上文件的传送 95](#_Toc16626956)

[2.5 套接字选项 95](#_Toc16626957)

[2.5.1 常见的SOL\_SOCKET选项 96](#_Toc16626958)

[2.5.2 setsockopt和getsockopt 97](#_Toc16626959)

[2.6 对1.8节的改进 100](#_Toc16626960)

[2.7 TCP的连接和断开 104](#_Toc16626961)

[2.7.1 tcpdump使用初步 104](#_Toc16626962)

[2.7.2 TCP的连接：三次握手 107](#_Toc16626963)

[2.7.3 TCP数据的传输 109](#_Toc16626964)

[2.7.4 TCP的断开：四次分手 111](#_Toc16626965)

[2.8 UDP的编程-消息的收发 112](#_Toc16626966)

[2.8.1 UDP服务器 114](#_Toc16626967)

[2.8.2 UDP客户机 116](#_Toc16626968)

[2.8.3 问题分析 119](#_Toc16626969)

[2.9 UDP的数据传输 119](#_Toc16626970)

[2.10 UDP的数据边界 120](#_Toc16626971)

[2.11 UDP下使用connect函数 123](#_Toc16626972)

[2.12 广播-文件传输 126](#_Toc16626973)

[2.12.1 feof函数 127](#_Toc16626974)

[2.12.1 文件传输的发送端 129](#_Toc16626975)

[2.12.2 文件传输的接收端 131](#_Toc16626976)

[2.12.4 测试 132](#_Toc16626977)

[2.13 广播-UDP消息 133](#_Toc16626978)

[2.13.1 消息发送端 133](#_Toc16626979)

[2.13.2 消息接收端 135](#_Toc16626980)

[2.14 小结 137](#_Toc16626981)

[第三章 标准IO和文件描述符 137](#_Toc16626982)

[3.1 标准IO 137](#_Toc16626983)

[3.1.1 系统I/O和标准I/O 138](#_Toc16626984)

[3.1.2 fopen和fclose 139](#_Toc16626985)

[3.1.3 fread、fwrite和shutdown 140](#_Toc16626986)

[3.2 半关闭下网络文件的传输 142](#_Toc16626987)

[3.2.1 发送文件的服务器 142](#_Toc16626988)

[3.2.2 接收文件的客户机 145](#_Toc16626989)

[3.3 文件描述符和流的转换 147](#_Toc16626990)

[3.3.1 fdopen函数 147](#_Toc16626991)

[3.3.2 fclose和fflush函数 149](#_Toc16626992)

[3.3.3 fileno函数 150](#_Toc16626993)

[3.4 标准IO：发送固定消息的服务器 152](#_Toc16626994)

[3.4.1 标准IO：发送固定消息的服务器的服务器端代码 152](#_Toc16626995)

[3.4.2 标准IO：发送固定消息的服务器客户机端代码 154](#_Toc16626996)

[3.5 标准IO：反射服务器 156](#_Toc16626997)

[3.5.1 标准IO：反射服务器的服务器代码 156](#_Toc16626998)

[3.5.2 标准IO：反射服务器的客户机代码 158](#_Toc16626999)

[3.6流的半关闭文件和描述符的复制 160](#_Toc16627000)

[3.6.1 标准IO：失败的半关闭模型 161](#_Toc16627001)

[3.6.2 dup、dup2和fcntl函数 165](#_Toc16627002)

[3.6.3 标准IO：成功的半关闭模型 167](#_Toc16627003)

[3.7 小结 172](#_Toc16627004)

[第四章 进程 172](#_Toc16627005)

[4.1 什么是进程 172](#_Toc16627006)

[4.1.1 进程也是文件 172](#_Toc16627007)

[4.1.2 查看进程的状态ps 173](#_Toc16627008)

[4.1.3 进程的前后台切换bg、fg 175](#_Toc16627009)

[4.2 进程的产生 176](#_Toc16627010)

[4.2.1 进程函数fork 176](#_Toc16627011)

[4.2.2 为什么要用进程 177](#_Toc16627012)

[4.2.3 孤儿进程和僵尸进程 177](#_Toc16627013)

[4.3 进程和子进程 177](#_Toc16627014)

[4.4 父子进程的数据处理 184](#_Toc16627015)

[4.5 wait函数 192](#_Toc16627016)

[4.6 信号 196](#_Toc16627017)

[4.6.1标准信号 196](#_Toc16627018)

[4.6.2 信号值 197](#_Toc16627019)

[4.7 信号注册函数signal 199](#_Toc16627020)

[4.7.1 signal函数原型 199](#_Toc16627021)

[4.7.2 signal函数分析 200](#_Toc16627022)

[4.7.3 errno变量 201](#_Toc16627023)

[4.7.4 kill-杀死进程 201](#_Toc16627024)

[4.7.5 signal函数的使用 202](#_Toc16627025)

[4.7.6 小结 206](#_Toc16627026)

[4.8 信号注册函数sigaction 207](#_Toc16627027)

[4.8.1 sigaction函数原型 207](#_Toc16627028)

[4.8.2 sigaction函数的使用 208](#_Toc16627029)

[4.9 系统中断调用的处理 210](#_Toc16627030)

[4.10 进程：TCP反射服务器 214](#_Toc16627031)

[4.10.1 进程-TCP服务器端 214](#_Toc16627032)

[4.10.2 进程-TCP客户机端 219](#_Toc16627033)

[4.10.3 进程-TCP客户机的半关闭讨论 222](#_Toc16627034)

[4.11 listen的第二个参数 223](#_Toc16627035)

[4.12 进程和管道pipe 225](#_Toc16627036)

[4.13 进程、管道和质数求解 230](#_Toc16627037)

[4.13.1 质数求解-常规方法 230](#_Toc16627038)

[4.13.2 多进程质数求解-失败的模型 232](#_Toc16627039)

[4.13.3 多进程质数求解-成功的模型 235](#_Toc16627040)

[4.14 管道-监听反射内容 238](#_Toc16627041)

[4.15 小结 245](#_Toc16627042)

[第五章 线程 246](#_Toc16627043)

[5.1 进程和线程 246](#_Toc16627044)

[5.1.1、进程及其处理 246](#_Toc16627045)

[5.1.2、线程 246](#_Toc16627046)

[5.1.3、线程的编译 247](#_Toc16627047)

[5.1.4、线程的结束 248](#_Toc16627048)

[5.2 线程的建立和销毁 252](#_Toc16627049)

[5.2.1 创建线程 252](#_Toc16627050)

[5.2.2 线程的终结函数 257](#_Toc16627051)

[5.2.3 线程终结的不同 259](#_Toc16627052)

[5.3 一个进程下最多的线程数 267](#_Toc16627053)

[5.4 函数的可重入问题 269](#_Toc16627054)

[5.5 线程的竞争，应该怎么做？ 272](#_Toc16627055)

[5.5.1 竞争的表现 272](#_Toc16627056)

[5.5.2 竞争的原因 276](#_Toc16627057)

[5.5.3 竞争的解决 280](#_Toc16627058)

[5.5.4 可移植的解决方案 282](#_Toc16627059)

[5.6 互斥锁 284](#_Toc16627060)

[5.7 信号量 292](#_Toc16627061)

[5.8 互斥锁和信号量的使用 304](#_Toc16627062)

[5.9 线程-计算质数的个数 310](#_Toc16627063)

[5.9.1 常规方法 310](#_Toc16627064)

[5.9.2 失败的模型 311](#_Toc16627065)

[5.9.3 修改后的模型 313](#_Toc16627066)

[5.9.4 最后的模型 317](#_Toc16627067)

[5.10 进程和线程的效率比较 322](#_Toc16627068)

[5.11 线程-聊天室的服务器端 327](#_Toc16627069)

[5.11.1 反射服务器和聊天室分析 327](#_Toc16627070)

[5.11.2 数组元素的删除 328](#_Toc16627071)

[5.11.3 线程-聊天室服务器 329](#_Toc16627072)

[5.12 线程-聊天室的客户机端 333](#_Toc16627073)

[5.12.1 sprintf函数 333](#_Toc16627074)

[5.12.2 exit和return 334](#_Toc16627075)

[5.12.3 -1的指针类型 335](#_Toc16627076)

[5.12.2 线程的客户机代码 336](#_Toc16627077)

[5.13 线程-聊天室的监控 339](#_Toc16627078)

[5.13.1 服务器端 339](#_Toc16627079)

[5.13.2 运行测试 342](#_Toc16627080)

[5.13 线程的编程模式 343](#_Toc16627081)

[5.13.1 每个客户一个线程 343](#_Toc16627082)

[5.13.2 每个线程各自accept 343](#_Toc16627083)

[5.14 小结 343](#_Toc16627084)

[第六章 select和IO复用 344](#_Toc16627085)

[6.1 IO复用 344](#_Toc16627086)

[6.1.1 什么是IO复用 344](#_Toc16627087)

[6.1.2 什么时候使用IO复用？ 344](#_Toc16627088)

[6.2 如何实现IO复用？ 345](#_Toc16627089)

[6.2.1 IO复用函数select 345](#_Toc16627090)

[6.2.2 select函数说明 346](#_Toc16627091)

[6.2.3 select函数的运行 350](#_Toc16627092)

[6.3 IO复用的服务器 351](#_Toc16627093)

[6.3.1 过程分析 352](#_Toc16627094)

[6.3.2 服务器代码 352](#_Toc16627095)

[6.3.3 代码说明 356](#_Toc16627096)

[6.4 IO复用的客户机 356](#_Toc16627097)

[6.4.1 IO复用的客户机模型1.0版 356](#_Toc16627098)

[6.4.2 版本1.0的说明 361](#_Toc16627099)

[6.4.3 版本1.0和2.0的测试 361](#_Toc16627100)

[6.4.3 版本2.0的说明 362](#_Toc16627101)

[6.5 select实现TCP与UDP混合的服务器 362](#_Toc16627102)

[6.5.1 过程分析 362](#_Toc16627103)

[6.5.2 混合的tcp和udp实现 363](#_Toc16627104)

[6.5.3 代码说明 366](#_Toc16627105)

[6.6 小结 368](#_Toc16627106)

[第七章 epoll 368](#_Toc16627107)

[7.1 epoll是什么 368](#_Toc16627108)

[7.1.1 epoll和select 369](#_Toc16627109)

[7.1.2 水平触发和边缘触发 370](#_Toc16627110)

[7.1.3 IO阻塞与非阻塞 370](#_Toc16627111)

[7.1.4 read函数非阻塞的例子 371](#_Toc16627112)

[7.1.5 非阻塞式IO 373](#_Toc16627113)

[7.2 EPOLL的基本函数 373](#_Toc16627114)

[7.2.1 epoll的使用 373](#_Toc16627115)

[7.2.2 epoll\_create() 374](#_Toc16627116)

[7.2.3 epoll\_ctl() 375](#_Toc16627117)

[7.2.4 epoll\_wait() 377](#_Toc16627118)

[7.3 epoll的服务器 378](#_Toc16627119)

[7.3.1 水平触发模式服务器 378](#_Toc16627120)

[7.3.2 服务器运行测试 382](#_Toc16627121)

[7.3.3 边缘触发模式服务器 383](#_Toc16627122)

[7.4 小结 387](#_Toc16627123)

[第八章 Web服务器 388](#_Toc16627124)

[8.1 http和html协议 388](#_Toc16627125)

[8.1.1 http模型介绍 388](#_Toc16627126)

[8.1.2获得HTTP的协议头 388](#_Toc16627127)

[8.1.3执行过程 389](#_Toc16627128)

[8.1.4 html协议 392](#_Toc16627129)

[8.2 Web服务器的设计思路和测试方法 394](#_Toc16627130)

[8.2.1 设计思路 394](#_Toc16627131)

[8.2.2 测试工作 394](#_Toc16627132)

[8.3 需要使用的函数 396](#_Toc16627133)

[8.3.1 strstr函数 396](#_Toc16627134)

[8.3.2 strtok函数 397](#_Toc16627135)

[8.4 Web服务器的实现 399](#_Toc16627136)

[8.4.1 服务器代码 399](#_Toc16627137)

[8.4.2 代码说明 404](#_Toc16627138)

[8.5 libcurl：客户端下载实现 405](#_Toc16627139)

[8.5.1 libcurl是什么？ 405](#_Toc16627140)

[8.5.2 使用yum下载安装 405](#_Toc16627141)

[8.5.3 CURL、CURLoption和CURLcode类型 408](#_Toc16627142)

[8.5.4 curl\_global\_init和curl\_easy\_init函数 408](#_Toc16627143)

[8.5.5 curl\_easy\_cleanup和curl\_easy\_setopt函数 409](#_Toc16627144)

[8.5.6 curl\_easy\_perform函数 410](#_Toc16627145)

[8.5.7 实现代码及测试 411](#_Toc16627146)

[8.6 小结 412](#_Toc16627147)

[附录A TCP和UDP使用的模块函数说明 413](#_Toc16627148)

[附录B 代码说明 414](#_Toc16627149)

[1、长代码的续行问题 414](#_Toc16627150)

[2、错误函数 414](#_Toc16627151)

[3、本书代码 414](#_Toc16627152)

[附录C 二级指针的含义 415](#_Toc16627153)

[附录D 参考文献 419](#_Toc16627154)

# 内容提要

本书不是一本系统编程手册，也不是一本教材，不过它因为有以下的特征，所以也适合用作教材。

1、内容具有连贯性；

2、内容从例题入手，保持从简单到深入；

3、代码讲解详细，适合教学和自学；

从网络的基本函数开始，围绕TCP和UDP的项目展开。

# 前言

能把复杂问题讲得简单的老师就是好老师。这是我读大学的时候我的老师告诉我的。

学习并发编程技术，最好的方法就是编程，这里有两个意思，一个是使用C语言，因为它处理的是原始的数据；第二个意思是使用网络编程技术。

网络编程又必须和系统编程相联系。所以我们就有了一个线条：C语言—网络编程—系统编程—并发处理方法。

在多数情况下，我们的编程理念是平面的，也就是说只需要考虑程序代码的逻辑关系，代码从头走到最后，如果没有问题，那么程序的运行就不会有问题。但是在并发编程中，我们的编程理念就编程立体的方式了，我们不仅要关注程序进行的逻辑关系，还要关注程序进行的时间安排。在线程编程中，我们会看到大量的莫名其妙的结果，如果不理解并发造成的竞争，就无法处理这些问题。事实上，现代大量的代码BUG很多都是并发造成的，最有影响的事件可能就是美国的火星探路者航天器由于线程编程的问题造成的故障。可见，即使是那么尖端的科技人员也会出现这样的问题。最主要的是由于并发原因造成的错误隐藏很深，它并不是马上就显示出来，有的时候在你的系统里运行正常，但是在其它的系统里运行就不正常，而且这种不正常显示出来的结果没有重复的特征，导致了发现BUG的难度很大。

全书共八章。

第一章，先介绍man手册的用法，因为全书都是围绕man手册给出的说明而进行的。然后详细介绍了网络编程的数据类型，逐步讲解各个接口函数的含义和使用，并实现了TCP编程的初步架构。

第二章，讲述TCP和UDP在迭代方式下的实现，实现TCP下的反射模式，通过分析套接字的选项，进一步完善TCP的模型，使用对比的方式讲解UDP方式和TCP方式的不同。通过使用tcpdump工具观察数据的流向来分析TCP的连接和断开特点以及UDP的数据传输特点。

第三章，按照Linux的口号“一切皆文件”的说法，本章使用文件方式来实现前两章的项目，对其中使用到的函数也进行了详细的举例和说明。

第四章，开始并发编程设计的方法之一：进程。从分析孤儿进程和僵尸进程产生的原因开始，讲述了信号和信号函数，处理僵尸进程的方法，pipe管道完成进程之间的通信等，完成了并发的反射服务器，然后讲解了求解质数的进程方法，并把这种方法应用到监听服务器内容上。

第五章，讲述线程技术。并发设计之二：线程。讲述了线程的各种处理函数，竞争的原因和处理方法，并使用“质数个数求解”来演示竞争的过程，同步的方法——讲述了信号量和互斥锁的原理和使用环境，然后通过反射服务器实现聊天室，并且实现了监控聊天室的方法。

第六章，讲述并发技术的另一种方法——IO复用技术。讲述了IO复用的基本原理和建立方法，然后使用select实现了一个TCP和UDP混合的项目实例。

第七章，讲述新版的IO复用技术——epoll方法。分析了epoll和select的不同之处，对非阻塞模式和边缘触发进行了讨论，最后实现了epoll复用的反射服务器。

第八章，综合实例，讲述了在Web下的编程，包括两个方面：Web服务器的简单设计方法和使用第三方的库函数libcurl编程实现网络下载。这里是前面几章内容的综合实践，同时也讲述了如何去下载使用第三方库实现新的功能，这样做有助于提高效率。

# 第一章 网络编程基础

## 1.1 帮助的使用和基本配置

### 1.1.1 man手册的使用

#### 1. man手册章节

标准的man手册主要分为8个章节，分别为：

man 1 -------用户在shell下可以执行的命令；

man 2 -------系统内核课调用的函数和工具等；

man 3 -------一些常用的函数和库，大多是C函数库；

man 4 ------- 设备文件和特殊文件，通常在/dev下的文件；

man 5 ------- 配置文件及格式；

man 6 ------- 游戏；

man 7 ------- 杂项和一些约定，比如协议等等，你可以看到TCP、IP等的协议；

man 8 ------- 管理类命令；

man 9 ------- 跟内核有关的文件。

使用最多的参数是“-a”参数，见下面的使用内容。

**2、man命令的使用**

例题1.1 使用man手册查看printf和ip。

使用如下的命令，你可以进入printf的命令帮助。注意“1”的前后有空格。下面都是这样。

[root@localhost ~]# man 1 printf

使用如下的命令，你可以进入printf的函数库中的函数帮助。

[root@localhost ~]# man 3 printf

如果你想查看所有的有关printf的内容，那么可以使用如下的命令：

[root@localhost ~]# man printf -a

-a：的意思是all，意即显示所有的printf的条目。

使用下面的命令，会进入ip的第7章内容，在这里你可以看到socket函数、IP的地址格式、IP的socket选项等的内容。

[root@localhost ~]# man 7 ip

**说明：**

如果使用命令：man -help 或者：man man

你可以看到更多的关于man的参数。不过，我们使用最多的是参数“-a”，目的就是查找同名的命令或者函数（比如printf）的用法，找到自己需要的内容。

在我们的编程中会大量的使用man来查看有关函数的使用，事实上，网络和系统编程中需要使用大量的系统函数，我们不可能都记住这些函数的参数和使用方法，所以需要经常的使用man命令查看使用的方法。

使用man来查看一个函数，需要注意以下三点：

* 函数的参数；
* 函数的返回值；
* 函数的其他情况，比如有的时候需要关注errno的值，用来分析判断函数的异常。

### 1.1.2 服务器--客户机模式

又叫客户机/服务器系统 Client/server system 简称C/S系统。

在这种模式下，服务器和客户机各司其职。服务器总是被动的等待连接，客户机总是主动的发起连接，并请求服务。

服务器提供各种的服务，比如http，FTP，视频数据等等，客户机根据自己的需要去向服务器请求对应的服务。这样的模型简化了任务处理，各做各的事情。我们要开发的TCP/IP的各种项目就是基于这样的模型进行的。

### 1.1.3 gcc编译工具

**1、GCC的参数：**

GCC的初衷是为GNU操作系统专门编写的一款编译器。

使用下面的命令，可以看到其复杂的参数：

[root@localhost ~]# man gcc

或者，直接使用gcc的帮助：

[root@localhost ~]# gcc --help

虽然有很多参数，但是对于我们而言，只需要使用几个参数即可。

(1)、只使用gcc，不带参数，那么生成的可执行文件是a.out，所以，在Linux的这个环境下，a.out是被改写最多的文件。受此影响，有的时候为了省事，我们的程序文件名就是：a.c，只使用不带参数的gcc编译，这样造成的问题是你可能忘记了再次编译而使用之前的a.out，导致结果会莫名其妙。所以在课程讲述的时候，我们还是尽量的给代码起一个好记的名字。

[root@localhost ~]# gcc server.c

(2)、使用参数“-o”，生成指定的可执行文件。

[root@localhost ~]# gcc -o server.c server

意思就是：把源文件server.c 编译为名字叫做server的可执行文件，而不是a.out了。

(3)、-Wall

输出所有的gcc的警告信息。

(4)、-w

关闭警告信息。比如，输入字符串如果使用gets函数，编译的时候会有警告，“警告”不是错误，所以编译会继续进行，如果你不想看到这个警告就可以使用这个参数。不过，不建议你使用这个参数，在大多数情况下，警告信息总是有用的。

(5)、-std=c99

gcc默认是使用c89的标准，如果使用这个参数的话，就使用c99的标准。

(6)、-l

装入编译时候需要链接的库。

比如，我们在编译含有线程的代码a.c的时候，需要输入下面的命令：

gcc a.c -lpthread

-lpthread的含义就是装入线程的库。

又比如，我们在编译含有数学公式的文件，例如cos，sin等，需要使用下面的编译命令：

gcc a.c -lm

-lm的含义就是装入数学（math）的库函数。

2、程序的调试

当一个辛苦写的代码无法通过编译，或者运行的结果有问题，怎么去调试？

有的书上或者网络上会介绍很多调试工具。不过，我不建议你使用，因为去熟悉一个工具也是要花时间的，所以我的建议就是使用printf，使用printf可以输出字符(%c)，整数(%d)，小数(%f)和地址(%p)。每一次的输出就形成一个“断点”，这样可以去检查程序运行的情况。

### 1.1.4 环境配置

首先配置linuc系统，建立两个IP地址。

在Vm虚拟机的右下角，如图1-1，选择“网络适配器”下的“设置”，如图1-2，使用桥接模式，那么虚拟机和主机就处于同一个网段，虚拟机也可以上网浏览网页。图1-3和图1-4的作用是增加虚拟机的网卡并设置网卡的模式。

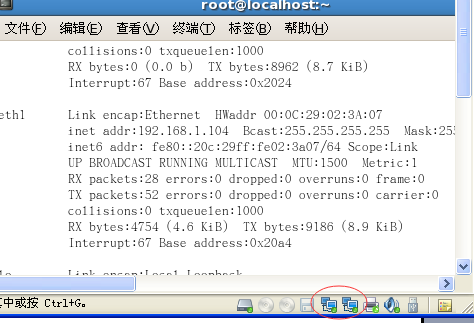


图1-1 选择网卡设置

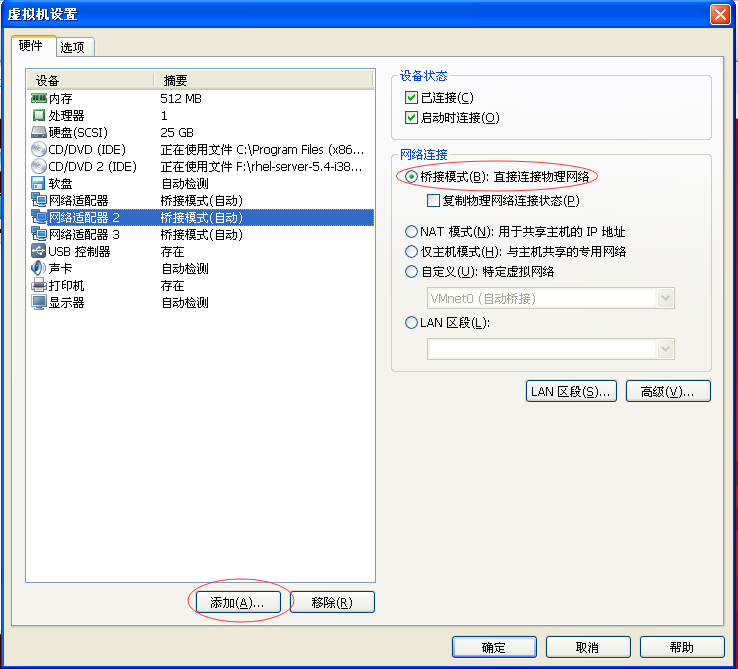


图1-2 设置网卡的模式

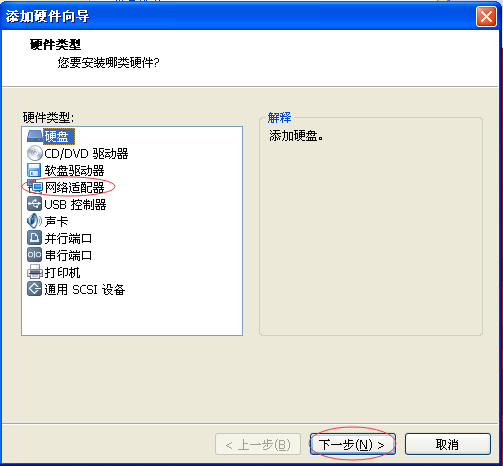


图1-3 增加网卡



图1-4 增加的网卡也需要设置为桥接模式

注意选择桥接模式，然后输入命令：reboot。重启系统，也可以使用网卡重启命令，见下面的说明。

进入linux的终端，输入命令：ifconfig，就应该能看到有eth0和eth1两个网卡。

现在对这两个网卡进行IP地址的配置。

输入命令“cd”，进入目录：

cd /etc/sysconfig/network-scripts/

应该能够看到：ifcfg-eth0和ifcfg-eth1两个文件，如图3，就是两个网卡，因为在linux里面，“一切皆文件”。

  
图1-5 查看网卡配置文件

我们打开ifcfg-eth0文件，修改后的内容如下：

[root@localhost ~]# vim /etc/sysconfig/network-scripts/ifcfg-eth0

DEVICE=eth0:0 //设备名称

BOOTPROTO=static //静态的IP地址，也可以使用dhcp

BROADCAST=10.10.33.255 //广播地址

IPADDR=192.168.1.10 //ip地址

NETMASK=255.255.255.0 //子网掩码

ONBOOT=yes //不要修改，启动时运行

TYPE=Ethernet //不要修改

HWADDR=00：0c：29：02：3a：fd //MAC地址

我们打开ifcfg-eth1文件，进行同样的修改，只需把

IPADDR=192.168.1.10

修改为：

IPADDR=192.168.1.20

这样就有了两个ip地址：192.168.1.10和192.168.1.20

修改完成之后，还没有生效，需要输入下面的命令，启动网卡的设置：

[root@localhost ~]# sevice network restart

然后，系统开始启动网卡设置，一切都显示“OK”，说明配置成功。

需要说明的是，你的电脑网卡的配置内容可能和我的不一样，但是DEVICE，BOOTPROTO，HWADDR，ONBOOT等这几项是有的。

执行ifconfig查看是否有了两个网卡？下面的图1-6是我的电脑的ifconfig显示：

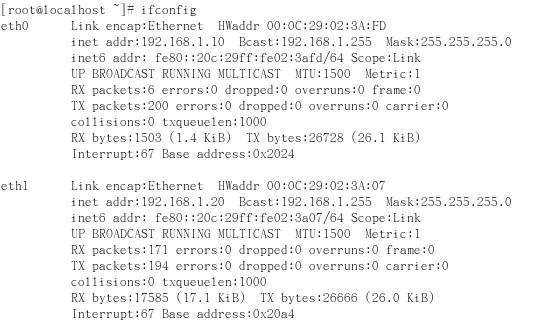
****

图1-6 使用ifconfig查看网卡配置

互相ping一下，看能否ping通， 现在模拟两个ip地址通信的基础就已经搭好，下面开始编程。实现TCP/IP的通信。

## 1.2 套接字的数据类型

套接字常用的数据类型，见有关的头文件。

例如：在终端中输入下面的命令：

[root@localhost ~]# cat /usr/include/sys/types.h

可以找到如下的定义：

......

**typedef char int8\_t;**

**typedef short int int16\_t;**

**typedef int int32\_t;**

......

8位整数（int类型）和char类型在0-128以内是通用的。还记得吗？

仔细一看，其实就是对整型数据的又一次定义，俗称：给类型起别名。

你要是有兴趣的话，就去下面的头文件里看，可以发现一大堆typedef定义。

所以，我们的网络编程会有很多头文件，多使用手册，你就会知道头文件在什么时候使用。如果少些一个头文件的话，有的时候编译没有问题，运行却会有莫名其妙的问题，后面我们会看到这样的例子。

部分数据类型和头文件详细如下表：

|  |  |  |
| --- | --- | --- |
| 数据类型 | 说明 | 头文件 |
| int8\_t | 带符号的8位整数 | sys/types.h |
| uint8\_t | 无符号8位整数 |
| int16\_t | 带符号16位整数 |
| uint16\_t | 无符号16位整数 |
| uint32\_t | 无符号32位整数 |
| int32\_t | 带符号32位整数 |
| sa\_family\_t | 套接字地址结构的地址族 | sys/socket.h |
| socklen\_t | 套接字地址结构长度，uint32\_t类型 |
| in\_addr\_t | IPV4地址,uint32\_t类型 | netinet/in.h |
| in\_port\_t | TCP或UDP端口，uint16\_t |

表1-1 部分数据类型及所在的头文件

所以，上面的表大都是8位、16位或者32位整数的另外的声明（或者叫定义），目的只是为了在TCP/IP编程中使用起来更加直接，容易理解而已。

所以，你在编程中直接使用int类型也不会有太大的问题，只要整数不要超过规定的最大值和最小值。

千万不要被这么多数据类型吓倒了，记住，他们就是不同整数的不同写法而已。

## 1.3 套接字和字节序

### 1.3.1 TCP/IP模型

TCP/IP是什么意思呢？TCP表示Transmission Control Protocol就是传输控协议，IP表示Internet Protocol就是网际协议。

经典的TCP/IP参考模型从上至下分为4个层次：应用层、传输层、网络层和网络接口层，其中网络接口层是由设备驱动和硬件接口组成，见表1-2。与OSI模型不同的是在TCP/IP参考模型中，根据实际情况将OSI参考模型的会话层、表示层和应用层合并到TCP/IP模型的应用层中；同时，将OSI参考模型中的数据链路层和物理层合并为设备驱动和硬件接口层。

从实际应用看，OSI七层模型用处不大，由于TCP/IP的广泛使用，四层模型被广泛的使用。

表1-2的过程可以从下面的描述来理解。

封包，从上往下看。你发出的数据从应用层开始，比如你就输入了“hello wenzi”，从上往下逐次把改成的数据和协议封装在下层的协议中，最后在物理层通过网线或者无线信号发出；

解包，从下往上看。接收端从最底层开始，逐次网上解包，到最后，回到应用层，用户看到的就是“hello wenzi”，用户根本不知道在数据传输的过程中发生了什么。

通过封包和解包的过程，我们看到这样做的好处是：每一层只需解决本层所关心的事情，并留出上层或者下层的接口，以便调用。其实，这个也是C语言函数的做法，对于C语言的函数而言，我们只关心函数的参数和返回值（有的函数是没有返回值的），函数的参数就是供传递给函数使用的，函数的结果（返回值）可以提供给其他函数使用。

|  |  |
| --- | --- |
| **OSI七层模型** | **TCP/IP四层模型** |
| 应用层 | 应用层 |
| 表示层 |
| 会话层 |
| 传输层 | TCP ，UDP传输层 |
| 网络层 | IPV4， IPV6网络层 |
| 数据链路层 | 设备驱动和硬件接口层 |
| 物理层 |

表1-2 OSI模型TCP/IP模型

TCP位于传输层，IP位于网络层，所以IP地址指的就是网络的IP地址。从下往上看，数据到达网络层后，送给传输层。

什么是文件描述符呢？

在Linux中强调：一切皆文件，所以文件描述符就是描述一个文件的整数，最直接获得这个整数的方法就是使用open函数，例如：下面的函数使用：

fd=open(argv[1], O\_WRONLY|O\_CREAT);

上面的函数告诉我们，打开一个文件，这个文件是命令行的第一个参数（第零个参数是命令本身），打开文件的方式是只写（O\_WRONLY），并且如果这个文件不存在的话就创建（O\_CREAT），如果open函数执行成功，那么fd就是一个文件描述符，它和这个文件及其打开的属性相关联。这个文件描述符总是当前系统可以使用的文件描述符的最小的那个。其中，0，1，2是系统定义使用的文件描述符。所以，如果没有其他的文件描述符的话，这个fd就是3，但是这个3和“int x=3”是不同的，因为前者关联的一个文件。

什么是套接字？

我们使用手机作为类比，见表1-3。

服务器端：

|  |  |
| --- | --- |
| 行为 | 对应的网络编程 |
| 买一部手机 | 建立套接字（socket） |
| 去移动电信部门申请一个号码，比如：99999999 | 给socket分配协议，IP地址，端口号 |
| 手机绑定了一个号码 | 绑定协议（bind） |
| 然后就坐下来，等手机铃响起 | 监听（listen） |
| 手机铃响起，接听手机 | 接受连接（accept） |

表1-3 套接字和网络通信

客户端：

拨打手机电话号码，就是主动拨打某人的电话------->对应网络编程的连接（connect）。

### 1.3.2 套接字

1. 什么是套接字：是网络交换信息的协议，也就是源设备和目标设备的IP地址和端口的组合，有了IP地址和端口信息，那么信息才能在TCP/IP协议下进行传输，套接字，有的书也翻译为插口，套接字函数也被翻译为插口函数。
2. 套接字地址结构
3. 通用套接字地址结构：

**struct sockaddr //套接字地址结构**

**{**

**sa\_family\_ sa\_family; //协议族**

**char sa\_data[14]; //协议族数据**

**}**

其中：

**typdef unsigned short sa\_family;**

1. Linux的地址结构：

在Linux中，采用下面的地址数据结构：

见：man 7 ip

**struct sockaddr\_in**

**{**

**sa\_family\_t sin\_family; /\* address family: AF\_INET \*/**

**u\_int16\_t sin\_port; /\* port in network byte order \*/**

**struct in\_addr sin\_addr; /\* internet address \*/**

**};**

**/\* Internet address. \*/**

**struct in\_addr**

**{**

**u\_int32\_t s\_addr; /\* address in network byte order \*/**

**};**

**说明：**

1. sin\_port和s\_addr都是网络字节序，见上面的注释。

2、使用通用套接字地址struct sockaddr的目的是为了使用不同格式的地址，这些地址都被强制的转换为通用的套接字地址，在bind，accept，connect等套接字函数中就是这样处理的。

使用如下命令，可以看到这些结构的定义，包括IPV4以及IPV6下的定义，你还可以看到常用的端口的定义（Standard well-known ports.），比如FTP是21。端口号和他提供的服务是对应的。比如80号端口就提供web服务。

[root@localhost ~]# cat /usr/include/netinet/in.h //1

u8: 无符号8位整数；

u16：无符号16位整数；

大量的定义信息，位置在：

sys/types.h

sys/sockets.h

netinet/in.h

等的头文件里面。你可以使用//1的方法去看，建议使用cat命令，不建议使用vim编辑器，原因在于担心你的一个不小心修改了这些头文件！

所以，在创建了套接字之后，你的任务：

首先，是把这个结构体清零，因为C语言下的局部变量没有初始化或者赋值的话，变量的内容是随机的，清零需要使用函数memset，当然，如果你不嫌麻烦的话，也可以把结构体的成员一个一个的清零！

memset函数在哪个头文件里面呢？请使用man memset查看，当你使用一个函数的时候，不知道它在哪个头文件里面，就可以使用man去查看，仅凭这点，我觉得linux编程就很方便。

然后，就是填空！把上面的三个结构体成员填空，之后就做下面的事情。

### 1.3.3 服务器-客户机模式下的TCP/IP通信

在TCP套接字的编程中，服务器端实现的步骤如下：

1. 使用socket（）函数创建套接字；
2. 分配地址和端口；（前面说过的）

3、把创建的套接字绑定到指定的地址结构，bind（）；

4、listen（）函数设置套接字为监听模式，使服务器进入被动打开的状态；

5、接受客户端的连接请求，建立连接，accept（）；

6、接受、应答客户端的数据请求，read（），write（）；

7、完成后，终止连接，close（）。

在客户端实现的步骤如下：

1、使用socket（）函数创建套接字；

2、使用connect（）函数建立一个与TCP服务器的连接；

3、发送数据请求，接受服务器的数据应答，read（），write（）；

4、完成后，终止连接，close（）。

这些函数，我们之后会逐个讲解，现在要处理一个问题：字节序。

不同的系统，整数在内存里面的存放顺序是不一样的，有的是“倒着放”，有的是“正着放”，于是就有了小端字节序和大端字节序的问题，但是，网络上面的传输数据统一规定为“大端字节序”。

### 1.3.4 字节序

数据在电脑主机里面的存放和在网络上的传输有那么一点不一样，所以需要统一规范。

就是电脑主机的字节存放，以及数据在网络的传输的匹配问题。

* 字节序的问题是由于CPU对整数在内存中的存放方式造成的。多于一个字节的数据类型在内存中的存放顺序叫主机字节序。最常见的字节序有两种，小端字节序和大端字节序。
* 小端字节序：内存的起始地址放低序字节。
* 大端字节序：内存的起始地址放高序字节。

例如：字节0x1234，小端字节序为：3412；大端字节序为：1234。

34 12（小） 12 34（大）

-------------------------------->

低地址----->高地址

* 网络的字节序标准规定为大端字节序，不同平台上会对主机字节序进行转化后再进行传送，到主机后再转化为主机字节序，数据的传输就不会产生传输造成的问题了。同一个数据在不同的平台上可以使用网络字节序的转换函数来实现。

对于0x123456这样的数据，我们可以使用union类型进行字节的数组查看，我们的数组一般总是先输出类似a[0]的数据，这个a[0]就是地址的起始地址。

这个不用去死记硬背，看下面的代码就明白了。

例题1.2、使用union类型来查看我的电脑的字节序的例题。

union类型的数据和struct类型不同，union是共用内存。

//代码：example1\_2.c

1 #include “stdio.h”

2 typedef union {

3 int a; //int类型为4个字节

4 char b[4];

5 }to; //定义to为union类型

6

7 int main()

8 {

9 to i;

10 i.a=0x12345678;

11 printf(“%x “,i.b[0]); //起始地址放78，所以是小端字节序

12 printf(“%x “,i.b[1]);

13 printf(“%x “,i.b[2]);

14 printf(“%x “,i.b[3]);

15 }

运行结果是：

[root@localhost ~]# vim example1\_2.c

[root@localhost ~]# gcc example1\_2.c

[root@localhost ~]# ./a.out

78 56 34 12 [root@localhost ~]#

总结：

对于主机，有的主机是大端字节序，有的主机是小端字节序；

对于网络，只能是大端字节序。

### 1.3.5 字节序转换函数hton和ntoh族

网络字节序采用大端字节序，所以小端字节序需要转换，当然，如果你的主机系统是大端字节序就不需要转换，然而，当软件移植到别的系统的时候就会出现问题，所以，建议都转换一下。

于是，发出信息数据的时候，需要主机到网络的字节序列转换；接收信息数据的时候，需要网络到主机的字节序列转换。是这样吗？

答：没有这么麻烦，我们只需要在分配socket的端口的时候才需要这么做，也就是说，在向sockaddr\_in结构体变量填充地址信息的时候，才需要这么做，其它的事情，系统会自动完成。

[root@localhost ~]# man htonl

得到下面的结果：

**函数位置头文件：arpa/inet.h**

**uint32\_t htonl(uint32\_t hostlong);**

**uint16\_t htons(uint16\_t hostshort);**

**uint32\_t ntohl(uint32\_t netlong);**

**uint16\_t ntohs(uint16\_t netshort);**

**用处：**主机需要和网络通信的时候，要用到这些函数。

**说明：**

h：host，表示主机；

n：net，表示网络；

htons: 主机序列转网络序列，s表示short int，2个字节，一般用于端口的转换；

ntohl ：网络序列转主机序列，l表示long int，4个字节，一般用于地址的转换；

剩下的两个函数的含义也是一样的。

**例题1.3、**下面的例子是对16位数值和32位数值进行字节序转换，把主机字节序转换为网络字节序，最后输出**。**

//代码：example1\_3.c

1 #include <stdio.h>

2 #include <arpa/inet.h> //htons

3 #define P printf

4

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

6 {

7 unsigned short int host\_port=0x1234; //1

8 unsigned short int net\_port; //2

9

10 int host\_addr=0x12345678;

11 int net\_addr;

12

13 net\_port=htons(host\_port);

14 net\_addr=htonl(host\_addr);

15

16 P("主机序列端口：%x\n",host\_port);

17 P("网络序列端口：%x\n",net\_port);

18 P("主机序列地址：%x\n",host\_addr);

19 P("网络序列地址：%x\n",net\_addr);

20 }

运行后的结果如下：

[root@localhost ~]# vim example1\_3.c

[root@localhost ~]# gcc example1\_3.c

[root@localhost ~]# ./a.out

主机序列端口：1234

网络序列端口：3412

主机序列地址：12345678

网络序列地址：78563412

[root@localhost ~]#

**说明：**

//1和//2：使用int类型也不会出现问题，因为这个时候是数据是0x1234，正好是两个字节。我们的linux系统是32位的。unsigned short int是2字节。请大家使用sizeof验证 。

运行一下，验证结果。我的电脑是intel芯片，验证结果如上，所以是小端字节序，如果你的电脑是MAC（苹果电脑），那么结果可能会不一样，据说，苹果电脑是大端字节序。

### 1.3.6 结构体置零函数memset

使用man命令查看memser函数的位置、含义和使用方法，今后都是这么做。

**[root@localhost ~]# man memset**

**#include <string.h>**

**void \*memset(void \*s, int c, size\_t n);**

**DESCRIPTION**

**The memset() function fills the first n bytes of the memory area**

**pointed to by s with the constant byte c.**

**RETURN VALUE**

**The memset() function returns a pointer to the memory area s.**

意思是：给内存的前n个字节填充整数c。

这里需要留意的是void，如果一个函数是void类型，意思就是没有返回值的函数，而一个指针是void类型的，就是说这个指针没有类型，就是万能类型的指针，可以使用强制转换的方法转化为任意类型的指针。

**参数：**

s：需要初始化的地址；

c：初始化的值；

n：需要初始化的结构的长度；

**返回值：**

返回s的指针。

**例题1.4、使用memset置零结构体。**

代码如下：

//代码：example1\_4.c

1 #include <stdio.h>

2 #include <string.h> //memset函数

3

4 typedef struct stu

5 {

6 char name[20];

7 int age;

8 }STU; //定义了一个结构体的类型STU

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

10 {

11 STU a;

12 printf("a.name=%s\t\ta.age=%d\n",a.name,a.age); //1

13

14 strcpy(a.name,"张山");

15 a.age=22;

16 printf("a.name=%s\t\ta.age=%d\n",a.name,a.age); //2

17

18 memset(&a,0,sizeof(a));

19 printf("a.name=%s\t\ta.age=%d\n",a.name,a.age); //3

20 }

**说明：**

//1：a变量没有初始化的值，所以结果是不确定的；

//2：经过赋值后，变量a有了确定的值；

//3：使用memset函数，a变量的值被清零了。

**问题：**

memset(&a,0,sizeof(a));

修改为：

memset(&a,67,sizeof(a));

结果如下：

a.name=CCCCCCCCCCCCCCCCCCCCCCCC a.age=1128481603

[root@localhost ~]#

为什么？

**答：**结构体里面填充的是ASCII码67，也就是大写字母c，“0”是NULL，所以，一般memset里面是写零。

编译运行，结果如下：

[root@localhost ~]# vim example1\_4.c

[root@localhost ~]# gcc example1\_4.c

[root@localhost ~]# ./a.out

a.name=Y�%�a.age=10544576

a.name=张山 a.age=22

a.name= a.age=0

[root@localhost ~]#

于是，使用一个socket，还需要处理地址，参看socket的结构体，地址是整数类型，可是我们使用的是点分结构的字符串类型，所以还有任务要继续，而且，我还需要传输数据，所以后面还会讲解数据处理的函数，比如read和write。

## 1.4 带点地址和二进制地址

### 1.4.1 IP地址的结构体

现在我们明白了套接字和字节序。

现在讲带点的IP地址和二进制IP地址及其转换。

由inet\_xxx类型的函数完成，我们这里先讨论inet\_pton和inet\_ntop两个函数。

用处：当需要按照可以看懂的方式显示IP地址的时候，或者需要让计算机来理解IP地址的时候，就要用到这些函数。

还记得socket下地址的定义吗：

在1.3.2节中，我们提到了下面的地址数据结构：

一级结构体：

struct sockaddr\_in

{

sa\_family\_t sin\_family; /\* address family: AF\_INET \*/

u\_int16\_t sin\_port; /\* port in network byte order \*/

struct in\_addr sin\_addr; /\* internet address \*/

};

**其中：**

**二级结构体：**

struct in\_addr //IP地址结构

{

u32 s\_addr; //32位IP地址，网络字节序

}

**注意了：sin\_addr地址类型是32位的整数！但是我们经常使用的地址是类似这样的：192.168.1.2**

这个是字符串。

**所以，我们应该明白了，我们看到的地址是字符串的模样，可是计算机网络处理的地址是整数的，于是又需要来进行转换了。**

### 1.4.2 inet\_aton，\*inet\_ntoa和inet\_addr函数

常见的函数：

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

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

**#include <arpa/inet.h>**

**int inet\_aton(const char \*cp, struct in\_addr \*inp); //1：字符串转换为整数**

**char \*inet\_ntoa(struct in\_addr in); //2：整数转换为字符串**

**in\_addr\_t inet\_addr(const char \*cp);**  **//3：字符串转换为整数**

**函数字母说明：**

a：字符串

n：整数（二进制）

**所以，对于一个懒人而言，记住1和2的互转函数即可，反正都是inet\_开头的函数。第3个函数的参数比第一个似乎简单。**

下面分别介绍这三个函数。

1、**int inet\_aton(const char \*cp, struct in\_addr \*inp)**

**参数：**

char \*cp,地址的字符串形式；

struct in\_addr \*inp，地址的十进制形式保存在这个指针里面；结果是网络字节序（大端字节序）。

**返回值：**

执行成功，返回1，失败，返回0；

**使用方法：**

1、char \*cp：说明这个参数是字符串，需要把字符串的数组名放进来；

2、struct in\_addr \*inp：说明inp参数是一个in\_addr的结构体类型，需要把这个结构体类型的变量地址放进来；

记忆方法：

aton：a表示数组array,n表示int，意味着字符串转换为整数。

**例题1.5、测试inet\_aton函数。**

先看看使用一级结构体，sockaddr\_in，地址在二级结构体sin\_addr里面。

代码如下：

**//代码：example1\_5.c**

1 #include <stdio.h>

2 #include <arpa/inet.h>

3 #include <stdlib.h> //exit

4 #define P printf

5

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

7 {

8 char \*addr="1.2.3.4";

9 struct sockaddr\_in addr\_inet;

10

11 if(!inet\_aton(addr,&addr\_inet.sin\_addr))

12 {

13 P("地址转换错误。");

14 exit(1);

15 }

16 else

17 P("字符串转换为整数地址是：%x\n",addr\_inet.sin\_addr.s\_addr);

18 }

**说明：**

第9行：addr\_inet是sockaddr\_in类型的结构体，地址在另一个结构体sin\_addr里面；

第11行：注意if语句里面的“非”运算，转换不成功，函数的值是0，经过非运算后变成1，于是输出错误信息；

第17行：正如第9行的解释，最后的地址是在二级结构体里面，所以是：addr\_inet.sin\_addr.s\_addr；

**代码运行结果如下：**

[root@localhost ~]# vim example1\_5.c

[root@localhost ~]# gcc example1\_5.c

[root@localhost ~]# ./a.out

字符串转换为整数地址是：4030201

[root@localhost ~]#

**注意：输出结果显示是网络字节序。**

那么，我可以使用二级结构体吗？也就是使用结构体in\_addr？

例题1.6 对例题1.5部分行进行了注释。

**代码如下：**

**//代码：example1\_6.c**

1 #include <stdio.h>

2 #include <arpa/inet.h>

3 #include <stdlib.h> //exit

4 #define P printf

5

6 //int inet\_aton(const char \*cp, struct in\_addr \*inp)

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

8 {

9 char \*addr="1.2.3.4";

10 //struct sockaddr\_in addr\_inet;

11 struct in\_addr addr\_inet;

12 //if(!inet\_aton(addr,&addr\_inet.sin\_addr))

13 if(!inet\_aton(addr,&addr\_inet))

14 {

15 P("地址转换错误。");

16 exit(1);

17 }

18 else

19 //P("字符串转换为整数地址是：%x\n",addr\_inet.sin\_addr.s\_addr);

20 P("字符串转换为整数地址是：%x\n",addr\_inet.s\_addr);

21 }

**运行结果：**

[root@localhost ~]# vim example1\_6.c

[root@localhost ~]# gcc example1\_6.c

[root@localhost ~]# ./a.out

字符串转换为整数地址是：4030201

[root@localhost ~]#

可见，结果是一样的。

**注意，第10和11行，第12和13行，第19和20行代码之间的不同。**

**2、char \*inet\_ntoa(struct in\_addr in);**

**参数：**

in：参数是struct in\_addr类型；

**返回值：**

这个是指针函数，我们在C语言的函数和指针里面讲过，指针函数比较危险，因为它返回的是地址，除非指针变量使用的是静态变量或者全局变量，如果使用局部变量的指针在函数使用完之后会释放，这样的话，这个指针就不是原来的地址了。

**所以，使用完这个函数，你的第一任务就是把这个指针函数的内容马上保存。这样比较安全。**

例题1.7 使用inet\_ntoa函数。

**代码如下：**

**//代码：example1\_7.c**

**1 #include <stdio.h>**

**2 #include <arpa/inet.h> //**

**3 #include <string.h> //string函数**

**4 #define P printf**

**5**

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

**7 {**

**8 //不可以按照下面的整数来写地址，因为函数的参数不是int**

**9 //int addr1=0x1020304;**

**10**

**11 struct sockaddr\_in addr1,addr2;**

**12 char \*p;**

**13 char p\_str[30]; //赋值备用**

**14 //下面两行代码，没有使用字节序列转换函数，所以输出的地址是倒着的**

**15 //addr1.sin\_addr.s\_addr=0x1020304;**

**16 //addr2.sin\_addr.s\_addr=0xc0a80101; //192.168.1.1的十六进制**

**17**

**18 addr1.sin\_addr.s\_addr=htonl(0x1020304);**

**19 addr2.sin\_addr.s\_addr=htonl(0xc0a80101); //192.168.1.1的十六进制**

**20**

**21 p=inet\_ntoa(addr1.sin\_addr);**

**22 strcpy(p\_str,p);**

**23 P("整数转化为字符串是：%s\n",p\_str);**

**24**

**25 p=inet\_ntoa(addr2.sin\_addr);**

**26 strcpy(p\_str,p);**

**27 P("整数转化为字符串是：%s\n",p\_str);**

**28 }**

**说明：**

第26行代码也可以省略，因为在使用inet\_ntoa函数后，没有覆盖p的内容，如果这样，第27行就修改如下，直接使用p指针的内容：

27 P("整数转化为字符串是：%s\n",p);

**编译运行：**

[root@localhost ~]# vim example1\_7.c

[root@localhost ~]# gcc example1\_7.c

[root@localhost ~]# ./a.out

整数转化为字符串是：1.2.3.4

整数转化为字符串是：192.168.1.1

[root@localhost ~]#

**变化一下，直接使用二级结构体，可以吗？**

**答：可以，代码如下：**

**1 #include <stdio.h>**

**2 #include <arpa/inet.h> //**

**3 #include <string.h> //string函数**

**4 #define P printf**

**5**

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

**7 {**

**8 //struct sockaddr\_in addr1,addr2;**

**9 struct in\_addr addr1,addr2;**

**10 char \*p;**

**11 char p\_str[30]; //赋值备用**

**12**

**13 //addr1.sin\_addr.s\_addr=htonl(0x1020304);**

**14 //addr2.sin\_addr.s\_addr=htonl(0xc0a80101); //192.168.1.1的十六进制**

**15**

**16 addr1.s\_addr=htonl(0x1020304);**

**17 addr2.s\_addr=htonl(0xc0a80101);**

**18 p=inet\_ntoa(addr1);**

**19 strcpy(p\_str,p);**

**20 P("整数转化为字符串是：%s\n",p\_str);**

**21**

**22 p=inet\_ntoa(addr2);**

**23 strcpy(p\_str,p);**

**24 P("整数转化为字符串是：%s\n",p\_str);**

**25 }**

**运行结果是一样的，请参照上面的那个函数的分析。**

**3、in\_addr\_t inet\_addr(const char \*cp);**

**参数：**

地址的字符串；

**返回值：**

成功：返回32位大端字节序整数；

失败：返回INADDR\_NONE;

我知道，大家喜欢这个函数，因为简单啊。

不过，有的资料认为应该抛弃这个函数，因为广播地址是255.255.255.255，其中，255是-1的补码，这个地址是有效的，但是函数失败的返回值INADDR\_NONE也被定义为-1，这样会产生二义性问题。建议不再使用。

**例题1.8、**测试inet\_addr函数。

**代码如下：**

//代码：example1\_8.c

**1 #include <stdio.h>**

**2 #include <arpa/inet.h> //**

**3 #include <stdlib.h> //exit**

**4 #define P printf**

**5**

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

**7 {**

**8 char \*addr1="1.2.3.4";**

**9 char \*addr2="256.1.2.3"; //错误的地址**

**10 int addr;**

**11**

**12 addr=inet\_addr(addr1);**

**13 if(addr==INADDR\_NONE)**

**14 {**

**15 P("地址转换错误。\n");**

**16 exit(1);**

**17 }**

**18 else**

**19 P("字符串转换为整数地址是：%x\n",addr);**

**20**

**21 addr=inet\_addr(addr2);**

**22 if(addr==INADDR\_NONE)**

**23 {**

**24 P("地址转换错误。\n");**

**25 exit(1);**

**26 }**

**27 else**

**28 P("字符串转换为整数地址是：%x\n",addr);**

**29 }**

**编译运行：**

[root@localhost ~]# vim example1\_8.c

[root@localhost ~]# gcc example1\_8.c

[root@localhost ~]# ./a.out

字符串转换为整数地址是：4030201

地址转换错误。

[root@localhost ~]#

### 1.4.3 inet\_pton和inet\_ntop

下面的两个函数同时适用于IPV4和IPV6，所以也必须知道。

见：

**man inet\_pton**

**1、inet\_pton函数：字符串类型的地址转换为二进制类型。**

**#include “sys/types.h”**

**#include “sys/socket.h”**

**#include “arpa/inet.h”**

**int inet\_pton(int af, const char \*src, void \*dst )**

**参数：**

af：协议族，AF\_INET是IPV4，AF\_INET6 是IPV6。

src：点分地址字符串；

dst：套接字地址结构中的二进制地址，一般使用十六进制；

**返回值：**

返回0，说明不是合法的IP地址；

返回-1，说明af不支持指定的协议族；

返回正值，说明转换成功；

**2、inet\_ntop函数：把二进制的IP地址转换为字符串类型。**

**#include “sys/types.h”**

**#include “sys/socket.h”**

**#include “arpa/inet.h”**

**const char \*inet\_ntop(int af, const void \*src, char \*dst, socklen\_t len )**

**参数：**

len：为了避免溢出而设置的

**返回值：**

返回一个指向dst的指针。如果错误，返回NULL。

**函数名中的p和n分别代表表达（presentation）和数值（numeric）。**

例题1.9、下面的例子说明了IP地址在二进制和字符串之间的转换

**//代码：example1\_9.c**

**1 #include <sys/types.h>**

**2 #include <sys/socket.h>**

**3 #include <arpa/inet.h>**

**4 #include <stdio.h>**

**5 #include <string.h>**

**6 #define ADDRLEN 16**

**7**

**8 int main()**

**9 {**

**10 struct in\_addr ip;**

**11 char ipstr[]="192.168.1.1";**

**12 char addr[ADDRLEN];**

**13 const char \*str=NULL;**

**14 int t;**

**15**

**16 //192.168.1.1**转化为二进制

**17 t=inet\_pton(AF\_INET,ipstr,&ip); //IPV4**

**18 if(t>0)**

**19 {**

**20 printf("inet\_pton:%s**转化为二进制是**：0x%x\n ",ipstr,ip.s\_addr);**

**21 }**

**22 //192.168.1.1**转化为字符串

**23 str=(const char \*)inet\_ntop(AF\_INET,(void \*)&ip,(char \*)addr,ADDRLEN);**

**24 if(str)**

**25 {**

**26 printf("inet\_ntop:0x%x**转化为字符串为：**%s\n",ip.s\_addr,str);**

**27 }**

**28 return 0;**

**29 }**

编译运行：

[root@localhost ~]# vim example1\_9.c

[root@localhost ~]# gcc example1\_9.c

[root@localhost ~]# ./a.out

inet\_pton:192.168.1.1转化为二进制是：0x101a8c0

inet\_ntop:0x101a8c0转化为字符串为：192.168.1.1

用Windows自带的计算器，可以知道192的十六进制是：c0，168是a8，其他的依此类推。

**问题：**

1、以上显示的是小端字节序，如何转化为大端字节序，就是下面的显示：

inet\_pton:192.168.1.1转化为二进制是：0xc0a80101

inet\_ntop:0xc0a80101转化为字符串为：192.168.1.1

答：程序的第20行修改为：

printf("inet\_pton:%s转化为二进制是：0x%x\n ",ipstr,htonl(ip.s\_addr));

第26行修改为：

printf("inet\_ntop:0x%x转化为字符串为：%s\n",htonl(ip.s\_addr),str);

2、

程序的第六行代码：

6 #define ADDRLEN 16

为什么要定义为16？

答：一个满秩IP地址（满秩就是说每一个点的数字部分都是三个数字组成）类似如下：

192.168.123.123

数一下们这个IP地址正好是15位，最后加上尾零，正好是16位。

## 1.5 服务器的网络编程函数

### 1.5.1 struct sockaddr类型的地址作用

man ip结果如下：

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

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

**#include <netinet/ip.h> /\* superset of previous \*/**

**tcp\_socket = socket(PF\_INET, SOCK\_STREAM, 0);**

**udp\_socket = socket(PF\_INET, SOCK\_DGRAM, 0);**

**raw\_socket = socket(PF\_INET, SOCK\_RAW, protocol);**

**struct sockaddr\_in {**

**sa\_family\_t sin\_family; /\* address family: AF\_INET \*/**

**u\_int16\_t sin\_port; /\* port in network byte order \*/**

**struct in\_addr sin\_addr; /\* internet address \*/**

**};**

**/\* Internet address. \*/**

**struct in\_addr {**

**u\_int32\_t s\_addr; /\* address in network byte order \*/**

**};**

而且，IP地址和端口都是以网络字节序保存。

这个地址的结构体，我们1.3.2节和1.4.1节都仔细的讨论过。

现在讨论通用套接字结构体，使用下面的命令可以看到这个通用套接字结构体，

[root@localhost ~]# man bind

**The sockaddr structure is defined as something like:**

**//sockaddr结构体被定义成下面的模样：**

**struct sockaddr {**

**sa\_family\_t sa\_family;**

**char sa\_data[14];**

**}**

那个时候，这个结构体还没有什么用处，现在却有用了。

TCP/IP的通信依靠下面的这些函数，例如：bind，accept，connect等函数，在这些函数里面，你会看到这些函数的参数里面有struct sockaddr的类型，这个类型就是通用套接字，那么，sockaddr\_in类型的结构体就要被强制转换为sockaddr的类型，即使你使用的是IPV6的结构体：sockaddr\_in6，也要求强制转换为sockaddr的类型，这个就是通用的含义。

如果我们省略了强制转换(struct sockaddr \*)，那么在编译的时候会产生一个警告，明确的告诉你存在一个不兼容的指针类型传递给函数。

所以，在下面的函数中，参数是struct sockaddr类型的结构体，你就不会奇怪了，它的目的就是把不同的结构体地址转换为相同的参数类型，供系统使用。

“一切皆文件”，这个是linux的核心口号。文件描述符，就是一个整数，但是它包含了其它的内容。

系统特定的文件描述符：

0：标准输入

1：标准输出

2：标准错误

数字3以后的文件描述符由其它的程序产生。

### 1.5.2 socket和bind函数

**1、socket()函数**

**#include “sys/types.h”**

**#include “sys/socket.h”**

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

**参数：**

domain:确定通信的特征,包括地址格式.

AF\_INET IPV4因特网域

AF\_INET6 IPV6因特网域

AF\_PACKET

AF\_UNIX / AF\_LOCAL UNIX域

AF\_UNSPEC 未指定,可以是任何域

type :确定套接字类型,进一步确定通信特征

SOCK\_DGRAM 长度固定,无连接的不可靠的报文传递(默认为UDP)

SOCK\_RAW IP协议的数据报接口 (原如套接字,底层套接字)

SOCK\_SEQPACKET长度固定 有序 可靠的面向连接报文传递

SOCK\_STREAM 有序 可靠 双向的面向连接的字节流 (默认协议TCP/IP)

domain：PF\_INET，IPV4 Internet协议， AF\_INET和 PF\_INET在头文件中是一样的；

type： SOCK\_STREAM是字节流套接字，适用于TCP通信，SOCK\_DGRAM是数据包套接字，适用于UDP协议的通信。

protocol：通常设置为0，因为一般一种协议只有一种类型。

**返回值：**成功就返回一个表示这个套接字的描述符，失败就返回-1。

**2、bind（）函数**

**#include “sys/type.h”**

**#include “sys/socket.h”**

**int bind(int sockfd, const struct sockaddr \*my\_addr, sock\_len addrlen);**

**参数：**

sockfd: 是socket()函数创建的套接字；

my\_addr：是结构体类型的指针；因为需要将结构体里面的服务器IP地址、端口等信息和my\_addr绑定在一起；

addrlen：是my\_addr的长度。可以设置为：sizeof(struct sockaddr)。

**返回值：**0表示成功，-1表示绑定失败，并设置errno。

这个函数的作用在于：把struct sockaddr类型的my\_addr的地址，端口等信息分配给第一个参数（套接字）sockfd，于是这个套接字sockfd就拥有的IP地址和端口号。

所以，在使用bind函数之前，需要确定服务器的协议，IP地址和端口，也就是要把sockaddr\_in结构体的成员赋值（或者叫完成结构体的成员填空）。

### 1.5.3 listen和accept函数

**3、listen ()函数**

**#include “sys/socket.h”**

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

**参数：**

sockfd：socket参数的文件描述符，说明是socked在监听；

backlog：侦听队列，一次性的侦听，设置为1，循环侦听，可以设置为整型，如果超出了这个长度，会产生错误，代码为：ECONNREFUSED。

**返回值：**成功，返回0；失败，返回-1，并设置errno；

关于参数backlog，man listen和man 7 tcp里面有详细的说明。

listen是一个进程，当队列满的时候，会逐个处理队列里面的连接，一旦队列为空，那么这个进程就处于休眠状态，发现有连接进入队列就激活。

一般设置为5，只是表示处理的队列是5个连接，不代表只能处理5个连接，因为他是活动的，随时可以被激活。

这个backlog参数在4.11节还要讨论。

**4、accept ()函数**

**#include “sys/types.h”**

**#include “sys/socket.h”**

**int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);**

**参数：**

sockfd：和前面的意义一样；

addr：客户机的地址信息；

addr\_len：addr的长度，可以使用：sizeof(struct sockaddr\_in)来获得。

**返回值：**是新连接的客户端套接字的文件描述符。如果失败，返回-1，并设置errno，其中，errno=EINTR ，就表示系统调用被中断了，在进程一章中会讨论这个问题。

所以，服务器端有两个套接字，旧的socket负责listen，新的套接字是accept产生的，负责发送、接受客户机的信息。

服务器函数总结：

socket

（memset）--->填空

bind

listen

accept

服务器要先运行。目的就是要被动的接收客户机的连接。

## 1.6 客户机通信的函数

### 1.6.1 客户机连接函数connect

**#include “sys/types.h”**

**#include “sys/socket.h”**

**int connect(int sockfd, struct sockaddr \*addr, int addrlen);**

**参数：**

sockfd：和前面的一样，为套接字文件描述符；

addr：包含要连接的服务器的IP地址等信息；

addrlen：指明addr类型的长度；

**返回值：**成功时返回0；失败返回-1，并设置errno.

connect发起三次握手。在建立连接的时候总会有一方先发送数据，那么谁调用了connect谁就是先发送数据的一方，accept是确认connect的函数。如此理解connect三个参数是容易了，我必需指定数据发送的地址，同时也必需指定数据从哪里发送，这正好是connect的前两个参数，而第三个参数是为第二个参数服务的。

### 1.6.2 send、recv和close函数

服务器和客户机使用各自的套接字描述符进行读/写的操作，所以可以使用write和read函数读/写数据。

**1、write（）函数**

**2、read（）函数**

这两个函数在1.9节会详细讨论，这里大家先知道这两个函数。网络编程和系统编程是紧密联系在一起的，所以，我们会讨论很多系统编程的方法。

**3、send()函数**

**#include “sys/types.h”**

**#include “sys/socket.h”**

**ssize\_t send(int sockfd, const void \*buf, size\_t len,int flags);**

**参数**：

前面3个参数和write（）一样，其中flags是传输控制标志，一般为0，表示常规操作。

**返回值：**

如果超过就返回写出的字节数，失败就返回-1.

**4、recv()函数**

**#include “sys/types.h”**

**#include “sys/socket.h”**

**ssize\_t recv(int sockfd, const void \*buf, size\_t len,int flags);**

**参数：**

前面3个参数和read（）一样，其中flags是传输控制标志，一般为0，表示常规操作。

**返回值：**如果成功就返回写出的字节数，失败就返回-1.

**5、close()函数**

**#include “unistd.h”**

**int close(int sockfd);**

**参数：**

sockfd：要关闭的套接字；

**返回值：**如果成功就返回0，失败返回-1。

实现同一个任务，可能有很多的函数，例如，bzero和memset都可以清空结构体，为了不至于引起麻烦，我们就只使用一个函数。

另外，TCP/IP的缓冲，是套接字的缓冲，建立套接字后就建立的缓冲，那么这个缓冲有多大呢？后面，我们使用套接字选项就可以知道这个套接字缓冲的大小。write和read函数里面的缓冲，是属于进程缓冲，是用户自己定义的缓冲大小，所以我们也把他们叫“数组”。

**例题1.10、测试read和write的使用。**

代码如下：

//代码：example1\_10.c

1 #include <stdio.h>

2

3 int main()

4 {

5 char a[10];

6 int len;

7 len=read(0,a,10);

8 write(1,a,len);

9 printf("len=%d\n",len);

10 }

代码不长，请大家自己编译运行。

**说明：**

1、第7行，read(0,a,10)中的“0”表示标准输入（就是键盘），接收10个字符；

2、第8行，write(1,a,len)中的“1”表示标准输出（就是屏幕），从数组a里面取出len个字符输出到屏幕；

3、可以使用printf代替第8行的write吗？

**答：**不可以，因为因为printf的%s是输出字符串，而这里没有尾零“\0”，所以，数组a不是字符串。

如果增加一句尾零，就可以使用printf的%s来输出字符串了，见下面的代码第9行：

**例题1.11、**read和printf的匹配。

代码如下：

//代码：example1-11.c

1 #include <stdio.h>

2

3 int main()

4 {

5 char a[10];

6 int len;

7 len=read(0,a,10);

8 //write(1,a,len);

9 a[len]=0;

10 printf("len=%d,a=%s\n",len,a);

11 }

请大家自己测试结果。

编译运行：

①、输入的字符数小于10：

[root@localhost ~]# vim example1\_11.c

[root@localhost ~]# gcc example1\_11.c

[root@localhost ~]# ./a.out

12345678

12345678

len=9

[root@localhost ~]#

最后显示，输入的字符数是9，是因为包含了换行符，所以，len=9,会在另起一行输出。

②、输入的字符数等于10：

[root@localhost ~]# ./a.out

1234567890

1234567890len=10

[root@localhost ~]#

[root@localhost ~]#

正好输出了10个字符，len=10是紧跟着输出了，而换行符在之后才输出，所以会有两个系统的提示符：“[root@localhost ~]# ”出现。

③、输入的字符数大于10：

你可以自己做测试，会发现多余10的字符会在shell里面作为命令，当然，结果是：“command not found”，原因参见②。

## 1.7 发送固定消息的服务器

在这节里，我们实现服务器端向客户机发送固定的信息，先要理解几个问题。

问题：

1、头文件是怎么来的？

答：使用man手册查看对应的函数。

2、大量的函数前面讲过，是什么意思？

答：前面讲过的那些socket地址、端口及其通信函数，在这里开始使用。

服务器设置固定信息，例如："这里是服务器发出的消息。"，然后等待客户机连接。客户机连接服务器后，就会收到服务器发来的这个信息。

### 1.7.1 服务器端实现

参见1.3.3，在那里我们提到了服务器端编程的步骤，请按照那里的步骤来理解下面的代码，客户机端的代码也是如此。

例题1.12、服务器发送消息，客户机接收，这里是服务器端的代码。

服务器代码如下：

**//代码：server1\_1.c**

1 #include <stdio.h>

2 #include <string.h> //memset

3 #include <stdlib.h> //exit

4 #include <sys/socket.h>

5 #include <sys/types.h>

6 #include <arpa/inet.h> //htonl

7

8 void error\_msg (char \*message); //把出错部分独立出来，免得重复写代码

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

10 {

11 int serv\_sock,clnt\_sock;

12

13 struct sockaddr\_in serv\_addr;

14 struct sockaddr\_in clnt\_addr;

15 int clnt\_addr\_size;

16

17 char message[]="这里是服务器发出的消息。";

18 //检查参数是否正确

19 if(argc!=2)

20 {

21 printf("用法：%s 端口号\n",argv[0]);

22 return 0;

23 }

24 //创建套接字

25 serv\_sock=socket(PF\_INET,SOCK\_STREAM,0);

26 if(serv\_sock==-1)

27 error\_msg("socket创建失败！");

28 //初始化地址结构体

29 memset(&serv\_addr,0,sizeof(serv\_addr));

30 serv\_addr.sin\_family=AF\_INET; //地址结构体填空

31 serv\_addr.sin\_addr.s\_addr=htonl(INADDR\_ANY); //同上

32 serv\_addr.sin\_port=htons(atoi(argv[1])); //同上

33 //绑定地址信息到套接字

34 if(bind(serv\_sock,(struct sockaddr\*)&serv\_addr,sizeof(serv\_addr))==-1)

35 error\_msg("bind函数绑定失败！");

36 //监听连接

37 if(listen(serv\_sock,5)==-1) //队列监听5个用户

38 error\_msg("listen函数失败。");

39 //获得客户机地址的长度，因为accept的参数需要这个长度

40 clnt\_addr\_size=sizeof(clnt\_addr);

41 clnt\_sock=accept(serv\_sock,(struct sockaddr\*)&clnt\_addr,&clnt\_addr\_size);//接受连接

42 //clnt\_sock=accept(serv\_sock,NULL,NULL);

43 if(clnt\_sock==-1) //连接失败

44 error\_msg("accept函数失败。");

45 //连接成功，就往客户机套接字写入数据

46 write(clnt\_sock,message,sizeof(message));

47 close(serv\_sock); //这两句是关闭套接字

48 close(clnt\_sock);

49 return 0;

50 }

51

52 void error\_msg(char \*message)

53 {

54 puts(message);

55 exit (0);

56 }

编译如下：

[root@localhost ~]# gcc server\_1.c -o server\_1

**说明：**

1、第25行代码，产生套接字，是流类型套接字（STREAM）；

2、第34行代码，把服务器端的地址、端口等信息绑定到套接字；

3、第37行代码：

37 if(listen(serv\_sock,5)==-1)

这里设置为5，意义不大，因为下面的代码并没有一个循环来处理监听队列，所以，可以设置为1。关于listen函数的第二个参数的含义，见4.11节。

4、第42行代码：服务器对客户机的地址信息没有兴趣，所以后面两个参数可以设置为空。clnt\_addr里面是客户机的IP地址，在2.1.2节我们会看到这个地址。

这里message是字符串常量，所以没有指定缓冲大小。如果不能确定字符串的长度，就需要指定数组大小。

5、请仔细阅读程序的注释，因为以后的工作大都如此，不会再写注释了。

### 1.7.2 客户端实现

参见1.3.3节，在那里我们提到了实现客户机的步骤。

例题1.13、客户机端代码的实现，主动发起连接。

**//代码：client\_1.c**

1 #include <stdio.h>

2 #include <string.h> //memset

3 #include <stdlib.h> //exit

4 #include <sys/socket.h>

5 #include <sys/types.h>

6 #include <arpa/inet.h> //htonl

7 #define SIZE 60 //注释1

8

9 void error\_msg(char \*message);

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

11 {

12 int sock;

13 struct sockaddr\_in serv\_addr;

14 char message[SIZE];

15 int str\_len;

16

17 if(argc!=3)

18 {

19 printf("格式：%s 服务器IP地址 服务器的端口\n",argv[0]);

20 return 0;

21 }

22

23 sock=socket(AF\_INET,SOCK\_STREAM,0);

24 if(sock==-1)

25 error\_msg("socket函数失败。");

26 //初始化套接字

27 memset(&serv\_addr,0,sizeof(serv\_addr));

28 serv\_addr.sin\_family=AF\_INET;

29 serv\_addr.sin\_addr.s\_addr=inet\_addr(argv[1]);

30 serv\_addr.sin\_port=htons(atoi(argv[2]));

31 //发起连接的失败处理

32 if(connect(sock,(struct sockaddr\*)&serv\_addr,sizeof(serv\_addr))==-1)

33 error\_msg("connect连接失败！");

34

35 str\_len=read(sock,message,SIZE); //注释2

36 if(str\_len==-1)

37 error\_msg("read读取服务器数据失败！");

38 //把读到的数据字符串输出处理

39 printf("连接服务器，读取信息是：%s\n",message);

40 close(sock);

41 return 0;

42 }

43

44 void error\_msg(char \*message)

45 {

46 puts(message);

47 exit (0);

48 }

**说明：**

1、第23行代码产生套接字，但是没有绑定地址，端口等信息到套接字？

2、第32行代码connect函数的sock没有绑定地址、端口等信息，系统会自动分配地址和端口给客户机。很明显的，serv\_addr是服务器的IP地址，因为套接字的初始化的时候已经赋值了。

3、请仔细阅读客户机端的代码注释。

当我给别人打电话的时候，需要拨通别人的电话号码，可是有谁会在这个时候去思考自己的电话号码呢？

编译如下：

[root@localhost ~]# gcc client\_1.c -o client\_1

运行：

先运行服务器端：

[root@localhost ~]# ./server\_1 9999

//没有信息，等待连接

然后运行客户端：

[root@localhost ~]# ./client\_1 192.168.1.108 9999

连接服务器，读取信息是：这里是服务器发出的消息。

**注释说明：**

注释1：数组缓冲大小为60，汉字是UTF-8编码，一个汉字长度是3个字节，汉字的标点符号也是3个字节。

注释2：SIZE也可以使用sizeof(message)。一次读入SIZE长度的数据到数组message里面。

**注意：**

1、我这里服务器有好几个网卡，所以客户机连接的时候可以指定任何的服务器网卡地址。服务器代码的第31行的INADDR\_ANY，含义就是任意的服务器地址。

如果我的服务器的IP地址是：192.168.1.112

那么可以把这句修改为：

**31 serv\_addr.sin\_addr.s\_addr=inet\_addr("192.168.1.112");**

2、listen函数，虽然可以监听5个客户队列，但是，因为没有使用循环迭代，所以，先到先得，完成后就退出；

3、服务器端运行完毕之后，再次马上运行的时候会报错，出现如下提示：

**[root@localhost ~]# ./server\_1 9999**

**bind函数绑定失败！**

这个错误是TCP协议的等待延时（time-wait）造成的，不是你的错误，只需要等待一会就可以再次运行，或者把端口号9999修改成其他的数字，例如9998等就可以了，后面我们会讲套接字选项的时候会处理这个问题。

这个时候，还有一个检查方法，那就是再打开第三个终端，输入下面的命令（假设地址是：192.168.1.112，端口是9999）：

netstat -an | grep 9999

结果显示如下：

[root@localhost ~]# netstat -an | grep 9999

tcp 0 0 192.168.1.112:9999 192.168.1.112:47479 TIME\_WAIT

[root@localhost ~]#

可见，确实是发生了time\_wait事件。

所以，我们首先真的需要使用错误函数，发现问题所在；其次，可以考虑使用一些网络的工具去检测你的猜想。

netstat -an | grep 9999的含义如下：

netstat：网络状态检查的命令；

-an：以数字的方式显示所有的端口；

grep 9999：过滤的作用，只显示端口号是9999的端口。

4、套接字，要么是自动分配地址（例如connect），要么需要人工分配（例如bind）；

5、服务器有两个套接字，一个叫监听套接字（listen），是一直都存在的，一个叫已连接套接字（accept产生），连接结束就需要关闭，以准备下一个连接。

## 1.8 代码的改进

针对服务器端和客户端，首先都是需要初始化套接字，这个叫代码的冗余，因为只要你写网络编程的代码，这个就是一定要写的，所以，我们可以把这部分独立处理写成一个函数，供以后去调用，这是C语言编程的常见做法。

### 1.8.1 服务器端代码改进

例题1.14、只发送消息的服务器代码的改进，把程序的功能函数模块化，这样就可以多次调用。

**//服务器端代码：server\_2.c**

1 #include <stdio.h>

2 #include <string.h>

3 #include <stdlib.h>

4 #include <sys/socket.h>

5 #include <sys/types.h>

6 #include <arpa/inet.h>

7

8 void error\_msg(char \*message);

9 int bind\_1 (char \*port);

10

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

12 {

13 int serv\_sock,clnt\_sock;

14

15 struct sockaddr\_in clnt\_addr;

16 int clnt\_addr\_size;

17

18 char message[]="这里是服务器发出的消息。";

19

20 if(argc!=2)

21 {

22 printf("用法：%s 端口号\n",argv[0]);

23 return 0;

24 }

25

26 serv\_sock=bind\_1(\*(argv+1));

27 if(listen(serv\_sock,5)==-1)

28 error\_msg("listen函数失败。");

29

30 clnt\_addr\_size=sizeof(clnt\_addr);

31 clnt\_sock=accept(serv\_sock,(struct sockaddr\*)&clnt\_addr,&clnt\_addr\_size);

32 //clnt\_sock=accept(serv\_sock,NULL,NULL);

33 if(clnt\_sock==-1)

34 error\_msg("accept函数失败。");

35

36 write(clnt\_sock,message,sizeof(message));

37 close(serv\_sock);

38 close(clnt\_sock);

39 return 0;

40 }

41

42 void error\_msg(char \*message)

43 {

44 puts(message);

45 exit (0);

46 }

47

48 int bind\_1(char \*port)

49 {

50 int sock;

51 struct sockaddr\_in addr;

52 sock=socket(AF\_INET,SOCK\_STREAM,0);

53 memset(&addr,0,sizeof(addr));

54 addr.sin\_family=AF\_INET;

55 addr.sin\_addr.s\_addr=htonl(INADDR\_ANY);

56 addr.sin\_port=htons(atoi(port));

57

58 if(bind(sock,(struct sockaddr\*)&addr,sizeof(addr))==-1)

59 error\_msg("bind函数绑定失败！");

60 return sock;

61 }

**说明：**

1. 第9行，函数原型，参数是命令行提供的端口号，返回值是套接字描述符；
2. 第27行，调用bind\_1函数，获得套接字，供下面的程序使用；
3. 第49-62行，完成创建套接字和套接字的初始化，然后进行bind函数的绑定，绑定成功就返回套接字描述符，这个时候的套接字就已经包含IP地址和端口号了。

但是第53行，忽略了socket失败的情况，所以53行代码实际上应该写成三行：

sock=socket(AF\_INET,SOCK\_STREAM,0);

if(sock==-1)

error\_msg(“socket函数失败。”);

好在我们的程序正常运行！不过还是建议增加socket失败的判断情况，即if语句。

编译运行：

[root@localhost ~]# vim server\_2.c

[root@localhost ~]# gcc server\_2.c -o server\_2

[root@localhost ~]# ./server\_2 9999 //等待状态，一旦连接就完成任务

[root@localhost ~]#

### 1.8.2 客户机端代码改进

例题1.15、使用函数修改客户机端的代码。

**//代码：client1\_2.c**

1 #include <stdio.h>

2 #include <string.h>

3 #include <stdlib.h>

4 #include <sys/socket.h>

5 #include <sys/types.h>

6 #include <arpa/inet.h>

7 #define SIZE 60

8

9 void error\_msg(char \*message);

10 int connect\_1 (char \*addr,char \*port);

11

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

13 {

14 int sock;

15 struct sockaddr\_in serv\_addr;

16 char message[SIZE];

17 int str\_len;

18

19 if(argc!=3)

20 {

21 printf("格式：%s 服务器IP地址 服务器的端口\n",argv[0]);

22 return 0;

23 }

24

25 sock=connect\_1(\*(argv+1),\*(argv+2));

26 str\_len=read(sock,message,SIZE);

27 if(str\_len==-1)

28 error\_msg("read读取服务器数据失败！");

29

30 printf("连接服务器，读取信息是：%s\n",message);

31 close(sock);

32 return 0;

33 }

34

35 void error\_msg(char \*message)

36 {

37 puts(message);

38 exit (0);

39 }

40

41 int connect\_1(char \*addr,char \*port)

42 {

43 struct sockaddr\_in serv\_addr;

44 int sock;

45 sock=socket(AF\_INET,SOCK\_STREAM,0);

46 memset(&serv\_addr,0,sizeof(serv\_addr));

47 serv\_addr.sin\_family=AF\_INET;

48 serv\_addr.sin\_addr.s\_addr=inet\_addr(addr);

49 serv\_addr.sin\_port=htons(atoi(port));

50

51 if(connect(sock,(struct sockaddr\*)&serv\_addr,sizeof(serv\_addr))==-1)

52 error\_msg("connect连接失败！");

53 return sock;

54 }

说明：

1. 客户端发起连接，起作用的是connect函数；
2. 第10行，connect\_1函数原型；
3. 第41行到54行，作用是建立套接字、完成初始化并完成连接，函数参数是IP地址和端口，都是字符串类型，所以在第41行实现函数的时候需要转换。这里connect函数使用了socket创建的套接字，最后return这个套接字，所以，sock并不是connect函数的结果，这点请大家务必注意。
4. 这样做main函数的结构显得简单、清晰。

编译运行：

[root@localhost ~]# vim client1\_2.c

[root@localhost ~]# gcc client1\_2.c -o client1\_2

[root@localhost ~]# ./client1\_2 192.168.1.108 9999

连接服务器，读取信息是：这里是服务器发出的消息。

[root@localhost ~]#

但是，服务器端运行完毕之后，再次马上运行的时候会报错，出现如下提示：

[root@localhost ~]# ./server1\_2 9999

bind函数绑定失败

这个问题仍然没有解决。

### 1.8.3 errno、perror和strerror函数

**首先来看perror函数：**

**[root@localhost ~]# man perror**

**SYNOPSIS**

**#include <stdio.h>**

**void perror(const char \*s);**

**参数：**

s：自己定义的供错误提示的字符串。

**返回值：**

无。

马上我们就给出实例，告诉你具体的使用方法。

什么是错误号errno呢？我们使用man手册查看一下。

[root@localhost ~]# man errno

SYNOPSIS

#include <errno.h>

.....

.....

下面是错误号：（部分）

E2BIG Argument list too long (POSIX.1)

EACCES Permission denied (POSIX.1)

EADDRINUSE

Address already in use (POSIX.1)

EADDRNOTAVAIL

Address not available (POSIX.1)

..............

.............

手册给出了对应的宏和解释，我们可以自己编程来查看错误号及其含义。

例题1.16、编程查看系统定义的错误号。

**//代码：example1\_16.c**

1 #include <stdio.h>

2 #include <errno.h>

3 int main()

4 {

5 int i;

6 for(i=0;i<199;i++)

7 printf("errno:%d=%s\n",i,strerror(i));

8 }

编译，运行，得到结果如下：

[root@localhost ~]# vim example1\_16.c

[root@localhost ~]# gcc example1\_16.c

[root@localhost ~]# ./a.out

errno:0=Success

errno:1=Operation not permitted

errno:2=No such file or directory

errno:3=No such process

errno:4=Interrupted system call

errno:5=Input/output error

errno:6=No such device or address

errno:7=Argument list too long

errno:8=Exec format error

errno:9=Bad file descriptor

errno:10=No child processes

errno:11=Resource temporarily unavailable

errno:12=Cannot allocate memory

errno:13=Permission denied

errno:14=Bad address

errno:15=Block device required

errno:16=Device or resource busy

errno:17=File exists

errno:18=Invalid cross-device link

errno:19=No such device

errno:20=Not a directory

errno:21=Is a directory

errno:22=Invalid argument

errno:23=Too many open files in system

errno:24=Too many open files

errno:25=Inappropriate ioctl for device

errno:26=Text file busy

errno:27=File too large

errno:28=No space left on device

errno:29=Illegal seek

errno:30=Read-only file system

errno:31=Too many links

errno:32=Broken pipe

errno:33=Numerical argument out of domain

errno:34=Numerical result out of range

errno:35=Resource deadlock avoided

errno:36=File name too long

errno:37=No locks available

errno:38=Function not implemented

errno:39=Directory not empty

errno:40=Too many levels of symbolic links

errno:41=Unknown error 41

errno:42=No message of desired type

errno:43=Identifier removed

errno:44=Channel number out of range

errno:45=Level 2 not synchronized

errno:46=Level 3 halted

errno:47=Level 3 reset

errno:48=Link number out of range

errno:49=Protocol driver not attached

errno:50=No CSI structure available

errno:51=Level 2 halted

errno:52=Invalid exchange

errno:53=Invalid request descriptor

errno:54=Exchange full

errno:55=No anode

errno:56=Invalid request code

errno:57=Invalid slot

errno:58=Unknown error 58

errno:59=Bad font file format

errno:60=Device not a stream

errno:61=No data available

errno:62=Timer expired

errno:63=Out of streams resources

errno:64=Machine is not on the network

errno:65=Package not installed

errno:66=Object is remote

errno:67=Link has been severed

errno:68=Advertise error

errno:69=Srmount error

errno:70=Communication error on send

errno:71=Protocol error

errno:72=Multihop attempted

errno:73=RFS specific error

errno:74=Bad message

errno:75=Value too large for defined data type

errno:76=Name not unique on network

errno:77=File descriptor in bad state

errno:78=Remote address changed

errno:79=Can not access a needed shared library

errno:80=Accessing a corrupted shared library

errno:81=.lib section in a.out corrupted

errno:82=Attempting to link in too many shared libraries

errno:83=Cannot exec a shared library directly

errno:84=Invalid or incomplete multibyte or wide character

errno:85=Interrupted system call should be restarted

errno:86=Streams pipe error

errno:87=Too many users

errno:88=Socket operation on non-socket

errno:89=Destination address required

errno:90=Message too long

errno:91=Protocol wrong type for socket

errno:92=Protocol not available

errno:93=Protocol not supported

errno:94=Socket type not supported

errno:95=Operation not supported

errno:96=Protocol family not supported

errno:97=Address family not supported by protocol

errno:98=Address already in use

errno:99=Cannot assign requested address

errno:100=Network is down

errno:101=Network is unreachable

errno:102=Network dropped connection on reset

errno:103=Software caused connection abort

errno:104=Connection reset by peer

errno:105=No buffer space available

errno:106=Transport endpoint is already connected

errno:107=Transport endpoint is not connected

errno:108=Cannot send after transport endpoint shutdown

errno:109=Too many references: cannot splice

errno:110=Connection timed out

errno:111=Connection refused

errno:112=Host is down

errno:113=No route to host

errno:114=Operation already in progress

errno:115=Operation now in progress

errno:116=Stale NFS file handle

errno:117=Structure needs cleaning

errno:118=Not a XENIX named type file

errno:119=No XENIX semaphores available

errno:120=Is a named type file

errno:121=Remote I/O error

errno:122=Disk quota exceeded

errno:123=No medium found

errno:124=Wrong medium type

errno:125=Operation canceled

errno:126=Required key not available

errno:127=Key has expired

errno:128=Key has been revoked

errno:129=Key was rejected by service

errno:130=Owner died

errno:131=State not recoverable

errno:132=Unknown error 132

在我的系统中，132之后都没有定义。

程序中使用了strerror函数，使用man来查看strerror的用法：

**SYNOPSIS**

**#include <string.h>**

**char \*strerror(int errnum);**

**char \*strerror\_r(int errnum, char \*buf, size\_t buflen); //说明**

**参数：**

1、errnum：strerror函数的参数是一个上面的错误代码，映射的是对应的错误信息；

2、perror函数的参数是一个字符串，执行的时候先输出这个字符串，然后是冒号，空格，接着是对应于error值的错误信息。

**返回值：**

返回错误号对应的错误信息，也可能有系统没有定义的错误信息。

**说明：**

**char \*strerror\_r(int errnum, char \*buf, size\_t buflen);**

是strerror的加强版，是线程安全函数，在5.4节中还会讲到。

错误号0表示success，意即成功，所以，我们的C程序的最后一句一般是：

return 0；

这样做是有道理的。

见下面的例子。

例题1.17、关于perror和strerror函数使用的例题。

**//代码：example1\_17.c**

1 #include <stdio.h>

2 #include <errno.h>

3 #include <string.h>

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

5 {

6 fprintf(stderr,"%s\n",strerror(EACCES));

7 errno=3;

8 //errno=4;

9 perror(argv[0]);

10 }

运行结果：

[root@localhost ~]# vim example1\_17.c

[root@localhost ~]# gcc example1\_17.c

[root@localhost ~]# ./a.out

Permission denied //1

./a.out: No such process //2

[root@localhost ~]#

**说明：**

1、第6行，EACCES宏定义对应的错误信息是：Permission denied (POSIX.1)（参见前面的说明），于是"//1"就是结果；

2、第7行，先输出参数argv[0],就是命令本身，然后是冒号和空格，error=3，对应的错误信息是： No such process，于是就输出了“//2”的结果；

3、结论：错误代码可以使用数字或者其对应的宏。

**总结：**

错误提示，可以使用标准错误信息，也可以使用自己预设的信息，比较而言，自己设置的出错信息比较灵活。

我们的错误函数error\_msg也可以改成如下的样子：

void error\_msg(char \*message)

{

perror(message);

exit(0);

}

错误信息由系统给出。

## 1.9 使用write和read函数写的文件复制程序

数组计数：从零开始计数，C语言的数组是采用这样的计数方法，C++、java等也是采用这样的计数方法。

自然计数：从1开始计数，我们日常生活就是这样的。

write和read函数不仅可以使用在系统编程，也使用在网络信息传输，是“万能的数据传输函数”，不过，万能的函数就意味着在某些方面就无法实现一些特定的功能，所以，针对socket，也有一些特定的发送、接收函数，例如recv，recvmsg等，我们专注于万能函数，对于其他的几个函数请大家自己测试，其实明白了read和write的使用，其它的几个输入输出函数就不难理解了。

### 1.9.1 read、write和open函数

**1、read函数**

**#include <unistd.h>**

**ssize\_t read(int fd, void \*buf, size\_t count);**

**读作：**

从fd文件描述符每次读取count字节到buf缓冲区（数组）。

**参数：**

fd：read的文件描述符对象，

buf：读出的数据保存在这里；

count：读出的字节数。

**返回值：**

成功：读取的字节数。

失败：返回-1，并且设置errno。

man手册的描述如下：

[root@localhost ~]# man read

DESCRIPTION

read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

If count is zero, read() returns zero and has no other results. If count is greater than SSIZE\_MAX, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

**2、write函数**

**#include <unistd.h>**

**ssize\_t write(int fd, const void \*buf, size\_t count);**

**读作：**

每次从缓冲区buf读取count字节，写入到fd文件描述符。

**返回值：**

成功：写入的字节数。

失败：返回-1，并设置errno。

man手册的描述如下：

[root@localhost ~]# man write

DESCRIPTION

write() writes up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. POSIX requires that a read() which can be proved to occur after a write() has returned returns the new data. Note that not all file systems are POSIX conforming.

RETURN VALUE

On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately. If count is zero and the file descriptor refers to a regular file, 0 may be returned, or an error could be detected. For a special file, the results are not portable.

**3、open**

**SYNOPSIS**

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

**#include <sys/stat.h>**

**#include <fcntl.h>**

**int open(const char \*pathname, int flags); //1**

**int open(const char \*pathname, int flags, mode\_t mode); //2**

**参数：**

pathname：打开文件的名字；

flags：文件访问方式选项，有以下三个值，以下的三个值只能选择一个：

O\_RDONLY：只读打开；

O\_WRONLY：只写打开；

O\_RDWR：读写打开。

也就是说，在打开文件时候就必须指定打开的方式，而且这三种方式还可以和下面的常量进行“或”运算，man手册中有很多这样的常量，这里只列出几个，方便大家查询手册的时候对比：

O\_APPEND：总在文件尾部追加数据；

O\_CREAT：如果文件不存在就创建它；

O\_TRUNC：截断已有的文件，使其长度为0；

O\_NONBLOCK：以非阻塞方式打开。见6.1.3节。

………..

……….

mode：仅仅当创建新的文件时候才使用这个参数，说明了新建文件的权限。

**返回值：**

成功：返回文件描述符；

失败：返回-1，并设置errno；

可见//1和//2函数名字一样，区别就在第三个参数，如果打开创建的文件不存在，就需要参数：mode。

在下一节可以看到这几个函数的用法的例子。

### 1.9.2 read和write完成文件的复制

例题1.18、流和文件描述符的使用。

**//代码：example1\_18.c**

1 #include <stdio.h>

2 #include <string.h>

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

4 {

5 char buf[30]="hello world!";

6 fprintf(stderr,"%s\n",buf);

7 write(1,buf,sizeof(buf));

8 puts("");

9 write(1,buf,strlen(buf));

10 puts("");

11 }

运行结果：

[root@localhost ~]# vim example1\_18.c

[root@localhost ~]# gcc example1\_18.c

[root@localhost ~]# ./a.out

hello world!

hello world!

hello world!

[root@localhost ~]#

例题1.19、编写一个程序，实现文件的复制功能。

格式为：命令 源文件 目标文件

回顾一下两个操作系统的文件复制命令：

Linux下的命令是：cp

Windows下的命令是：copy

我们自己编写的命令，经过编译后名字叫：mycopy。

**//代码：mycopy.c**

1 #include <stdio.h>

2 #include <stdlib.h> //exit

3 #include <sys/types.h>

4 #include <sys/stat.h>

5 #include <fcntl.h>

6

7 #define SIZE 1024

8

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

10 {

11 int source\_fd,target\_fd,count;

12 char buf[SIZE];

13

14 if(argc!=3)

15 {

16 puts("格式为：命令 源文件 目标文件");

17 return 0;

18 }

19

20 if((source\_fd=open(argv[1],O\_RDONLY))==-1)

21 {

22 fprintf(stderr,"%s\n",strerror(2)); //采用标准错误信息

23 exit(1);

24 }

25 if((target\_fd=open(argv[2],O\_WRONLY|O\_CREAT,S\_IRWXU))==-1) //见说明

26 {

27 printf("无法建立文件，请检查磁盘剩余空间。\n"); //使用自己编写的错误信息

28 exit(1);

29 }

30

31 while((count=read(source\_fd,buf,SIZE))>0)

32 {

33 if(write(target\_fd,buf,count)!=count) //如果写入的数据和读出的数据长度不一样，就报错

34 {

35 printf("文件无法写入！");

36 exit(1);

37 }

38 }

39 close(source\_fd);

40 close(target\_fd);

41 }

说明：

第25行代码：

重要的是这几个参数：O\_WRONLY|O\_CREAT,S\_IRWXU

含义是：以只写方式打开文件，文件名是main函数的第2个参数（argv[2]，数组计数法的第二个，是自然计数法的第三个），如果文件不存在的话就创建，所以需要参数O\_CREAT,然后设置文件权限：文件属主有读，写，执行的权限。

编译，运行：

[root@localhost ~]# gcc mycp.c -o mycp //1

[root@localhost ~]# ./mycp abcd /home/aaa //2

No such file or directory

[root@localhost ~]# ./mycp a.txt /home/aaa //3

[root@localhost ~]# cat /home/aaa //4

12345678900

[root@localhost ~]#

**说明：**

//1：编译源程序文件名为mycp，编译中使用了参数“-o”，目的就是为了生成mycopy的可执行文件，而不是毫无生气的a.out；

//2：把不存在的文件abcd进行复制，显示的就是系统的错误提示；其他的提示都是我自己设置的错误提示。

//3：文件a.txt已经存在，把a.txt复制到/home目录，文件名是aaa；

//4：显示aa的内容，所以，复制成功。

总结：

write和read：使用的第一个参数是文件描述符，返回读或者写的字节数。

不使用close，在什么情况下会出现错误？

**答**：在进程正常结束的情况下，不会有任何问题，因为进程结束会自动的关闭所有打开的文件描述符。但是，如果进程没有正常的结束，那么可能会造成数据的丢失。

Linux 和Windows对文件的命名不太一样，Linux除了“/”之外的字符都可以是文件名的组成部分，还有一个主要特点是，Windows规定只有扩展名是：exe、com和bat的才是可以执行的文件，而Linux下无论说明扩展名都是可以执行的，只需要你赋予它可以执行的权限即可，当然，可以执行不意味着一定能执行，例如，一个文本文件执行起来是没有什么结果的。

## 1.10 域名和IP地址

在shell下查看主机名：

[root@localhost ~]# hostname

localhost.localdomain

[root@localhost ~]#

什么是域名？

答：互联网通过域名服务器来完成域名到IP地址的转换，叫做DNS服务器。一般情况下，域名不会发生变更，而IP地址可能会发生变更，我们通过www.163.com可以访问网易的网页，通过其IP地址可能就不能访问，原因可能是避免IP地址的攻击。

例如，我们Windows下使用ping可以找到网易的IP地址：

方法，进入windows的命令模式：CMD

C:\Users\Administrator>ping www.163.com

正在 Ping www.163.com.lxdns.com [122.70.142.172] 具有 32 字节的数据:

来自 122.70.142.172 的回复: 字节=32 时间=60ms TTL=49

来自 122.70.142.172 的回复: 字节=32 时间=51ms TTL=50

来自 122.70.142.172 的回复: 字节=32 时间=59ms TTL=49

来自 122.70.142.172 的回复: 字节=32 时间=55ms TTL=50

122.70.142.172 的 Ping 统计信息:

数据包: 已发送 = 4，已接收 = 4，丢失 = 0 (0% 丢失)，

往返行程的估计时间(以毫秒为单位):

最短 = 51ms，最长 = 60ms，平均 = 56ms

C:\Users\Administrator>

我们可以知道网易的IP地址是：

122.70.142.172

以上操作在Linux终端下也可以进行，请大家测试。

所以，我们更多的是使用域名来负责通信。

通过域名查找IP地址，叫正向解析，是DNS服务器来完成的；

通过IP地址查找域名，叫反向解析，ISP网络服务商来做的事情，所以，一般情况下是不可以。

但是我们可以编辑文件：/etc/hosts，把IP地址和域名写进这个文本文件里面，不过这样看到的信息就不是原来真实的了。

所以，仅供我们实验使用。

**[root@localhost ~]# man gethostbyname**

**#include <netdb.h>**

**extern int h\_errno;**

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

**#include <sys/socket.h> /\* for AF\_INET \*/**

**struct hostent \***

**gethostbyaddr(const void \*addr, int len, int type);**

**struct hostent {**

**char \*h\_name; /\* official name of host \*/**

**char \*\*h\_aliases; /\* alias list \*/**

**int h\_addrtype; /\* host address type \*/**

**int h\_length; /\* length of address \*/**

**char \*\*h\_addr\_list; /\* list of addresses \*/**

**}**

**#define h\_addr h\_addr\_list[0] /\* for backward compatibility \*/**

这里有两个函数，一个是使用域名获得IP地址，另一个恰恰相反，使用IP地址获得域名。

hostent是host entry的缩写，该结构记录主机的信息，包括主机名、别名、地址类型、地址长度和地址列表。之所以主机的地址是一个列表的形式，原因是当一个主机有多个网络接口时，自然有多个地址。

### 1.10.1 使用域名获得IP地址gethostbyname

函数：struct hostent \*gethostbyname(const char \*name);

怎么理解？

我们说过，看一个函数是什么意思，主要看两点：函数的返回值和函数的参数。

**1、返回值：**

是struct hostent类型的结构体，man gethostbyname说明如下：

RETURN VALUE

The gethostbyname() and gethostbyaddr() functions return the hostent

structure or a NULL pointer if an error occurs. On error, the h\_errno

variable holds an error number. When non-NULL, the return value may

point at static data, see the notes below.

如果失败，返回NULL；

**2、参数：**

字符串，就是IP地址的点分形式；

**3、结构体成员说明：**

（1）、

char \*h\_name;

字符串，里面是官方域名，就是域名的主页，不过，有的公司的域名并没有使用官方域名；

（2）、char \*\*h\_aliases;

字符类型二级指针，或者说是指针数组，那么这个数组里面就是若干个字符串，这多个字符串就是多个域名，因为一个IP地址可以绑定多个域名；

（3）、int h\_addrtype;

地址族类型，IPV4是AF\_INET，也支持IPV6；

（4）、int h\_length;

IP地址长度，IPV4地址是4个字节，所以是4，IPV6是16个字节。

（5）、 char \*\*h\_addr\_list;

又是一个字符类型的二级指针，IP地址列表，因为一个域名可以有多个IP地址，这样做可以利用多个服务器进行负载均衡。

例题1.19、使用函数来获得网易的IP地址。

首先，你必须可以上网。

其次，代码如下：

**//代码：example1\_19.c**

1 #include <stdio.h>

2 #include <netdb.h> //gethostbyname

3 int main(int argc,char \*\*argv) //使用二级指针，也可以使用指针数组

4 {

5 int i;

6 struct hostent \*host; //0

7 if(argc!=2)

8 {

9 printf("格式：%s 域名\n",\*(argv+1));

10 return 0;

11 }

12

13 host=gethostbyname(\*(argv+1)); //参数是域名，字符串

14 if(!host) //函数执行失败，返回NULL

15 {

16 fputs("gethostbyname执行失败",stderr); //错误通过标准错误输出

17 return 0;

18 }

19 //依次输出结构体的信息

20 printf("官方域名是：%s\n",host->h\_name); //1

21 for(i=0;host->h\_aliases[i];i++) //2

22 {

23 printf("其他域名 %d:%s\n",i+1,host->h\_aliases[i]);

24 }

25 printf("地址类型是：%s\n",\

26 (host->h\_addrtype==AF\_INET)?"AF\_INET":"AF\_INET6"); //3

27

28 for(i=0;host->h\_addr\_list[i];i++)

29 {

30 printf("其他IP地址%d: %s\n",\

31 i+1,inet\_ntoa(\*(struct in\_addr\*)host->h\_addr\_list[i]));//4

32 }

33 }

**说明：**

1、大家可能觉得头文件少了点吧？没有包含inet\_ntoa的头文件啊？但是可以编译就说明没有问题，原因在于头文件：netdb.h已经包含了其他的头文件。但是还是建议写上这个头文件，有的时候编译会给出莫名其妙的警告。

2、其他的说明：注意host是指针，所以要使用指向运算符“->”。

//0：这里需要定义为struct hostent的指针，因为函数gethostbyname返回的是指针；

//1：host->h\_name，取出h\_name里面的字符串域名；

//2：注意for循环里面的循环条件，host->h\_aliases[i]，h\_aliases是指针数组，里面是字符串，所以这个条件的含义是：依次取这个数组的字符串，循环结束的条件就是没有字符串！

//3：这个是问号冒号表达式，因为地址族协议不是IPV4就是IPV6;

(host->h\_addrtype==AF\_INET)?"AF\_INET":"AF\_INET6"

的意思是：host->h\_addrtype里面是IPV4吗？如果是，那么值就是IPV4，否则就是IPV6；

//4：

第25和26行，是一行代码，采用了分行书写符号“\”，或者叫续行符，一段代码太长了，分成两行书写。

第30和31行也是一行代码。C语言的续行问题，见附录B的说明。

运行结果如下，我们测试了三个域名：网易，搜狐，腾讯，你也可以做其它的测试：

[root@localhost ~]# vim example1\_19.c

[root@localhost ~]# gcc example1\_19.c

[root@localhost ~]# ./a.out www.163.com //网易

官方域名是：www.163.com.lxdns.com

其他域名 1:www.163.com

地址类型是：AF\_INET

其他IP地址1: 183.239.68.87

[root@localhost ~]# ./a.out www.sohu.com //搜狐

官方域名是：fgzyd.a.sohu.com

其他域名 1:www.sohu.com

其他域名 2:gs.a.sohu.com

地址类型是：AF\_INET

其他IP地址1: 183.240.112.10

[root@localhost ~]# ./a.out www.qq.com //腾讯

官方域名是：https.qq.com

其他域名 1:www.qq.com

地址类型是：AF\_INET

其他IP地址1: 121.51.142.21

[root@localhost ~]#

### 1.10.2 使用IP地址获得域名gethostbyaddr

**#include <sys/socket.h> /\* for AF\_INET \*/**

**struct hostent \***

**gethostbyaddr(const void \*addr, int len, int type);**

**参数：**

\*addr：char类型的指针，注意前面的void意味着是无类型，也就是万能指针；

len：地址信息的字节数，4就是IPV4，6就是IPV6；

type：AF\_INET，为IPV4；AF\_INET6，位IPV6;

问题就来了，这个addr是整数地址还是点分地址呢？还是查找man手册吧。

man gethostbyaddr

里面关于gethostbyaddr有专门的描述：

The gethostbyaddr() function returns a structure of type hostent for

the given host address addr of length len and address type type. Valid

address types are AF\_INET and AF\_INET6. The host address argument is a

pointer to a struct of a type depending on the address type, for exam-

ple a struct in\_addr \* (probably obtained via a call to inet\_addr())

for address type AF\_INET.

大概意思如下：

gethostbyaddr（）函数返回hostent类型的结构给定的主机地址addr长度为len和地址类型类型。

有效地址类型是AF\_INET和AF\_INET6。 主机地址参数是a指向类型结构的指针，取决于地址类型，用于检查一个结构in\_addr \*（可能是通过调用inet\_addr（）获得的）对于地址类型AF\_INET。

**返回值：**

hostent结构体类型的指针，那么很多内容就可以去这个结构体里面找，失败：返回NULL。

准备工作：

如前面所述，我对/etc/hosts文件进行了增加，行号是vim增加的，实现的时候不能写：

1 # Do not remove the following line, or various programs

2 # that require network functionality will fail.

3 127.0.0.1 localhost.localdomain localhost

4 ::1 localhost6.localdomain6 localhost6

5

6 183.239.68.87 www.163.com, www.qq.com //1

//1的内容就是新增的，表示网易和腾讯都是同一个IP地址（杜撰的，只做测试）。

例题1.20、通过IP地址获得域名。

代码如下：

**//代码：example1\_20.c**

1 #include <stdio.h>

2 #include <string.h> //memset

3 #include <netdb.h>

4 int main(int argc,char \*\*argv) //使用二级指针，也可以使用指针数组

5 {

6 int i;

7 struct hostent \*host;

8 struct sockaddr\_in addr; //socket的地址结构

9

10 if(argc!=2)

11 {

12 printf("格式：%s IP地址\n",\*(argv+1));

13 return 0;

14 }

15

16 memset(&addr,0,sizeof(addr)); //结构体的地址置零

17 addr.sin\_addr.s\_addr=inet\_addr(\*(argv+1)); //点分地址转整数，见说明1

18 host=gethostbyaddr((struct sockaddr\_in \*)&addr.sin\_addr,4,AF\_INET); //说明2

19 //host=gethostbyaddr((char \*)&addr.sin\_addr,4,AF\_INET); //说明2

20

21 if(!host) //执行函数gethostbyaddr失败

22 {

23 puts("gethostbyaddr失败！");

24 return 0;

25 }

26

27 printf("官方域名：%s\n",host->h\_name);

28

29 for(i=0;host->h\_aliases[i];i++)

30 printf("其他域名%d: %s\n",i+1,host->h\_aliases[i]);

31

32 printf("地址类型：%s\n",(host->h\_addrtype==AF\_INET)?"AF\_INET":"AF\_INET6");

33

34 for(i=0;host->h\_addr\_list[i];i++)

35 printf("IP地址%d: %s\n",\

36 i+1,inet\_ntoa(\*(struct in\_addr \*)host->h\_addr\_list[i])); //说明3

37 }

说明：

//1：地址是结构体类型，具体说，地址在sockaddr\_in里面的结构体成员sin\_addr，但sin\_addr也是一个结构体，在in\_addr里面，最后的成员是s\_addr，所以有“=”左边的写法，输入的地址是一个字符串\*(argv+1)，但是网络需要地址的整数形式，于是就使用了函数inet\_addr完成转换；

//2：gethostbyaddr函数的第一个参数是void类型的指针，可以强制转换为其他类型，对sin\_addr使用struct sockaddr\_in类型的指针完成强制转换。也可以使用下面的方法完成转换。

18 host=gethostbyaddr((struct in\_addr \*)&addr.sin\_addr.s\_addr,4,AF\_INET); //说明2

但是，第19行代码，虽然也可以正常运行，但是使用的是char类型的强制转换，仅仅是把地址的字符串作为参数，也是可行的。

//3：第35,36行代码使用了续行符号“\”，所以他们是一行代码，inet\_ntoa函数，参数是struct in\_addr in类型的数据，所以需要把host->h\_addr\_list[i]里面的IP地址字符串转化为struct in\_addr类型的地址，然后使用“\*”取出里面的地址值。

运行结果：

[root@localhost ~]# vim example1\_20.c

[root@localhost ~]# gcc example1\_20.c

[root@localhost ~]# ./a.out 183.239.68.87

官方域名：www.163.com,

其他域名1: www.qq.com

地址类型：AF\_INET

IP地址1: 183.239.68.87

[root@localhost ~]#

[root@localhost ~]# ./a.out 127.0.0.1

官方域名：localhost.localdomain

其他域名1: localhost

地址类型：AF\_INET

IP地址1: 127.0.0.1

[root@localhost ~]#

127.0.0.1就是主机的名字。

作为服务器的话，主机名一般是域名。

## 1.11 小结

1. 网络和系统编程需要大量的使用系统的函数接口（API），所以我们必须学会使用man命令来查看这些函数接口的使用；
2. 编程环境和网络环境决定了你的代码测试的结果，测试本身是一个不完全归纳方法。这里不出问题并不意味着其他地方也不出问题。有条件的话，请大家一定要在多台机器组成的网络下测试，发现BUG，找出解决方法；
3. IP地址的结构体，一个是通用的地址结构体，一个是以太网的地址结构体，在网络编程的函数参数中需要转换为通用的地址结构；
4. 网络和系统编程需要调用大量的系统函数（API），而且还要使用if语句来判断这些函数是否运行成功，如果不成功，还要分析其原因，作为教学，有的时候我们会省去那些if语句，目的是为了突出整个程序的架构，实际使用中，你不能省略这些if语句，否则出现了问题，你不会发现问题在什么地方。

比如，如果我们省略服务器代码（1.8.1节）的函数bind\_1里面的if语句：

59 if(bind(sock,(struct sockaddr\*)&addr,sizeof(addr))==-1)

60 error\_msg("bind函数绑定失败！");

修改为：

59 bind(sock,(struct sockaddr\*)&addr,sizeof(addr));

我们的程序不再给出错误提示，程序编译运行也不会有问题。

但是，正如我们前面分析的那样，程序正常运行后就会退出，你再次运行的时候就不会出现 "bind函数绑定失败！" 的提示。但是，程序就是不能正常的运行，你也不知道问题出在什么地方。