## Webbench

/\*

webbench：

简单的网站压力测试工具

通过创建子进程模拟客户端，在一定时间内向目标网站重复发送请求，并计算返回的数据

\*/

/\*

\* (C) Radim Kolar 1997-2004

\* This is free software, see GNU Public License version 2 for

\* details.

\*

\* Simple forking WWW Server benchmark:

\*

\* Usage:

\* webbench --help

\*

\* Return codes:

\* 0 - sucess

\* 1 - benchmark failed (server is not on-line)

\* 2 - bad param

\* 3 - internal error, fork failed

\*

如何使用webbench ；把源代码编译成webbench后，在shell中使用如下

的命令行。webbench -c 100 -t 60 http：//www。baidu。com/（注意末尾的‘/’不能少。）

\*/

/\* 这个socket.c文件是自己写的\*/

#include "socket.c"

#include <unistd.h>

#include <sys/param.h>

#include <rpc/types.h>

#include <getopt.h>

#include <strings.h>

#include <time.h>

#include <signal.h>

/\* values \*/

//控制测试的时长：经过benchtime时间，通过发送SIGALARM信号给子进程，子进程调用函数alarm\_handler，将timerexpired置为1，timerexpired这个变量是子进程停止测试的红绿灯。

volatile int timerexpired = 0; //volatile的作用是作为指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值。

/\*

测试的结果，其实用一个结构体表示更完善。

子进程把他们的各自的speed，failed，bytes写到管道文件里，各个子进程测试完成以后，父进程把管道里面的类似struct『int speed；int failed；bytes 』【】的数组的所有的元素进行汇总，计算出最后的speed，failed，bytes。

\*/

int speed = 0;

int failed = 0;

int bytes = 0;

int http10 = 1; /\* 0 - http/0.9, 1 - http/1.0, 2 - http/1.1 \*/

/\*

method: GET, HEAD, OPTIONS, TRACE以下这些宏是从命令行参数里获得用户输入时候使用， 下面这些变量大部分是用来获取用户输入的。在getopt\_long里使用

\*/

#define METHOD\_GET 0

#define METHOD\_HEAD 1

#define METHOD\_OPTIONS 2

#define METHOD\_TRACE 3

#define PROGRAM\_VERSION "1.5"

int method = METHOD\_GET;

int clients = 1; //需要模拟的客户端的个数，即需要创建的子进程数

int force = 0; //代表是否接收服务器的应答，0表示接受

int force\_reload = 0; //默认不重新发送请求

int proxyport = 80; //默认端口为80

char \*proxyhost = NULL; //代理服务器

int benchtime = 30; //模拟客户端的运行时间

int mypipe[2]; //父子进程通信的管道

//主机地址的字符串

char host[MAXHOSTNAMELEN]; //MAXHOSTNAMELEN是宏，大小为64

#define REQUEST\_SIZE 2048

/\*

这个是发送给服务器的http请求的字符串，是在后面用各种拼凑字符串的函数strcat之类的拼接出来的。

同时这个字符串也是会被write（sock，request，sizeof（request））的函数发送给服务器的。

\*/

char request[REQUEST\_SIZE];

/\*

struct option{

//参数的全名

const char \*name;

//选项是否需要参数: no\_argument(0)/required\_argument(1)/optional\_argument(2)这些都是宏。

int has\_arg;

//如果flag为NULL，getopt\_long返回结构体中val的值；如果flag不为空指针NULL，getopt\_long返回0，flag指针所指对象的值val，如果没有发现长选项，则flag所指的值不变。

int \*flag;

//发现长选项后的返回值，一般为短选项字符常量，或者是flag不为NULL时，载入flag的值

int val;

}

\*/

static const struct option long\_options[] =

{

//用户如果输入--force选项，则把第三个参数force这个全局变量的值赋值为1。这样程序后面就可以根据用户的选择做处理了

{"force",no\_argument,&force,1},

{"reload",no\_argument,&force\_reload,1},

{"time",required\_argument,NULL,'t'},

{"help",no\_argument,NULL,'?'},

{"http09",no\_argument,NULL,'9'},

{"http10",no\_argument,NULL,'1'},

{"http11",no\_argument,NULL,'2'},

{"get",no\_argument,&method,METHOD\_GET},

{"head",no\_argument,&method,METHOD\_HEAD},

{"options",no\_argument,&method,METHOD\_OPTIONS},

{"trace",no\_argument,&method,METHOD\_TRACE},

{"version",no\_argument,NULL,'V'},

{"proxy",required\_argument,NULL,'p'},

{"clients",required\_argument,NULL,'c'},

{NULL,0,NULL,0}

};

/\* prototypes \*/

//测试核心函数，在bench函数内调用

static void benchcore(const char\* host, const int port, const char \*request);

//测试函数，在获取用户的各种输入以后调用。

static int bench(void);

//构造http请求报文

static void build\_request(const char \*url);

//下面的函数是注册成sigalrm信号的处理函数。

static void alarm\_handler(int signal)

{

timerexpired = 1;

}

//以下为用户进行了错误的输入选项和参数后提示用户的。

static void usage(void)

{

//打印到st derr，会在屏幕上显示出来

fprintf(stderr,

"webbench [option]... URL\n"

" -f|--force Don't wait for reply from server.\n"

" -r|--reload Send reload request - Pragma: no-cache.\n"

" -t|--time <sec> Run benchmark for <sec> seconds. Default 30.\n"

" -p|--proxy <server:port> Use proxy server for request.\n"

" -c|--clients <n> Run <n> HTTP clients at once. Default one.\n"

" -9|--http09 Use HTTP/0.9 style requests.\n"

" -1|--http10 Use HTTP/1.0 protocol.\n"

" -2|--http11 Use HTTP/1.1 protocol.\n"

" --get Use GET request method.\n"

" --head Use HEAD request method.\n"

" --options Use OPTIONS request method.\n"

" --trace Use TRACE request method.\n"

" -?|-h|--help This information.\n"

" -V|--version Display program version.\n"

);

};

//以上是函数声明，以下是主函数

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

{

int opt = 0;

int options\_index = 0;

//tmp为处理代理服务器时用到的变量

char \*tmp = NULL;

//程序没有除命令以外的参数，则打印说明并退出

if (argc == 1)

{

usage();

return 2;

}

/\*\*

这个getlong-opt函数是属于c库的一个函数，等于是方便我们获取用户的输入，并把argv数组里的参数做整理的，注意函数调用完后会重排argv里面的元素。

函数名：getlong\_opt

注意这个函数可以从短选项和长选项里面分别去获取参数 ，用户输入一个-f为短选项，--force为长选项

参数：int argc

char \* const argv[]

const char\* optstring:所有定义的短选项字符,如果短选项字符后加‘：’表示必须跟参数

const struct option\* longopts:长选项结构体

int\* longindex:当前找到的参数在longopts中的下标值

返回值：如果为短选项，返回短选项的选项名

如果为长选项，返回长选项结构体定义的返回值

如果分析选项时遇到没有定义的选项，则返回？

如果已经分析完所有选项，返回-1

备注：两个全局变量（optarg/optind

optarg:当前处理选项的参数值

optind:下一个被处理的参数在argv中的下标值，当处理完所有选项，optint指向第一个非选项参数

\*\*/

//这个一定得自己去找资料去了解，还有optind与optarg，参考资料如下，有两个比较长的讲解，第一个比较难一些，第二个比较简单，篇幅都比较长慢慢看

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*资料一\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

getopt被用来解析命令行选项参数。

#include <unistd.h>

extern char \*optarg; //选项的参数指针

extern int optind, //下一次调用getopt的时，从optind存储的位置处重新开始检查选项。

extern int opterr, //当opterr=0时，getopt不向stderr输出错误信息。

extern int optopt; //当命令行选项字符不包括在optstring中或者选项缺少必要的参数时，该选项存储在optopt 中，getopt返回'？’、

int getopt(int argc, char \* const argv[], const char \*optstring);

调用一次，返回一个选项。 在命令行选项参数再也检查不到optstring中包含的选项时，返回－1，同时optind储存第一个不包含选项的命令行参数。

首先说一下什么是选项，什么是参数。

1.单个字符，表示选项，

2.单个字符后接一个冒号：表示该选项后必须跟一个参数。参数紧跟在选项后或者以空格隔开。该参数的指针赋给optarg。

3 单个字符后跟两个冒号，表示该选项后必须跟一个参数。参数必须紧跟在选项后不能以空格隔开。该参数的指针赋给optarg。（这个特性是GNU的扩张）。

例如gcc -g -o test test.c ，其中g和o表示选项，test为选项o的参数。

上面是getopt()函数的基本含义，大家懂得了这些之后，我们一个例子加深一下理解。

例如我们这样调用getopt(argc, argv, "ab:c:de::");

从上面我们可以知道，选项a，d没有参数，选项b,c有一个参数，选项e有有一个参数且必须紧跟在选项后不能以空格隔开。getopt首先扫描argv[1]到argv[argc-1]，并将选项及参数依次放到argv数组的最左边，非选项参数依次放到argv的最后边。

执行程序为:

0 1 2 3 4 5 6 7 8 9

$ ./test file1 -a -b -c code -d file2 -e file3

扫描过程中，optind是下一个选项的索引, 非选项参数将跳过，同时optind增1。optind初始值为1。当扫描argv[1]时，为非选项参数，跳过，optind=2;扫描到-a选项时， 下一个将要扫描的选项是-b,则optind更改为3；扫描到-b选项时，后面有参数（会认为-c为选项b的参数），optind=5，扫描到code非选项跳过optind=6；扫描到-d选项，后面没有参数，optind=7；扫描到file2非选项跳过optind=8；扫描到-e后面本来应该有参数，optind=9但是有空格所以e的参数为空。

扫描结束后，getopt会将argv数组修改成下面的形式

0 1 2 3 4 5 6 7 8 9

$ ./test -a -b -c -d -e file1 code file2 file3

同时，optind会指向非选项的第一个参数，如上面，optind将指向file1

代码如下：

#include <unistd.h>

#include <stdio.h>

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

{

int aflag=0, bflag=0, cflag=0;

int ch;

printf("optind:%d，opterr：%dn",optind,opterr);

printf("--------------------------n");

while ((ch = getopt(argc, argv, "ab:c:de::")) != -1)

{

printf("optind: %d,argc:%d,argv[%d]:%sn", optind,argc,optind,argv[optind]);

switch (ch) {

case 'a':

printf("HAVE option: -ann");

break;

case 'b':

printf("HAVE option: -bn");

printf("The argument of -b is %snn", optarg);

break;

case 'c':

printf("HAVE option: -cn");

printf("The argument of -c is %snn", optarg);

break;

case 'd':

printf("HAVE option: -dn");

break;

case 'e':

printf("HAVE option: -en");

printf("The argument of -e is %snn", optarg);

break;

case '?':

printf("Unknown option: %cn",(char)optopt);

break;

}

}

printf("----------------------------n");

printf("optind=%d,argv[%d]=%sn",optind,optind,argv[optind]);

}

执行结果：

shiqi@wjl-desktop:~/code$ vim getopt.c

shiqi@wjl-desktop:~/code$ gcc getopt.c -o g

shiqi@wjl-desktop:~/code$ ./g file1 -a -b -c code -d file2 -e file3

optind:1，opterr：1

--------------------------

optind: 3,argc:10,argv[3]:-b

HAVE option: -a

optind: 5,argc:10,argv[5]:code

HAVE option: -b

The argument of -b is -c

optind: 7,argc:10,argv[7]:file2

HAVE option: -d

optind: 9,argc:10,argv[9]:file3

HAVE option: -e

The argument of -e is (null)

----------------------------

optind=6,argv[6]=file1 //while循环执行完后，optind=6

opt函数会一边扫描参数获得值，扫完所有的参数以后会把AR g的数组的元素的顺序重新整理。形成符合opt的一个顺序。这样肯定是为了方便处理参数。optind一路往下都不是参数和选项的东西。

转自：https://www.cnblogs.com/xhg940420/p/7016574.html

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*资料一结束\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*以下是webbench的源代码\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//把用户输入的webbench -c 100 -t 60 http：//www。baidu。com/这种命令的参数整理出来。

while ((opt = getopt\_long(argc, argv, "912Vfrt:p:c:?h", long\_options, &options\_index)) != EOF)

{

switch (opt)

{

case 0:

break;

case 'f':

force = 1;

break;

case 'r':

force\_reload = 1;

break;

case '9':

http10 = 0;

break;

case '1':

http10 = 1;

break;

case '2':

http10 = 2;

break;

case 'V':

printf(PROGRAM\_VERSION"\n");

exit(0);

case 't':

benchtime = atoi(optarg);

break;

case 'p':

/\* proxy server parsing server:port \*/

/\*

strrchr(const char \* s ,int c):查找字符在字符串中最后一次出现的位置，返回该字符的地址，

比如strrchr(cabbbab，a)返回的位置是ca这个a所在的位置。

\*/

//用户的 输入是IP地址加冒号加端口号，所以通过查冒号来确定字符串把ip地址和端口号分开的位置 127.0.0.1：80

tmp = strrchr(optarg, ':');

//用proxyhost去获取用户输入的ip地址和端口的字符串，optarg会随着while循环而变化

proxyhost = optarg;

//搜索”：“失败了，没找到，那是用户的输入错误了，退出

if (tmp == NULL)

{

break;

}

//冒号在最前面：80，没有写入IP地址，比如用户输入：127.0.0.1：80

if (tmp == optarg)

{

fprintf(stderr, "Error in option --proxy %s: Missing hostname.\n", optarg);

return 2;

}

//冒号后面的端口数据不足

//比如：8,至少要两位

if (tmp == optarg + strlen(optarg) - 1)

{

fprintf(stderr, "Error in option --proxy %s Port number is missing.\n", optarg);

return 2;

}

/\*

\*tmp='/0' //'\0'为字符串的结束

将代理服务器分为两个字符串，一个是IP抵制，一个是端口，例如：127.0.0.1:80，这里就将:改成'\0',proxyhost = "127.0.0.1"，proxyhost原来是”127.0.0.1::80”

这种作者想到的细节，我们去推敲的话就非常浪费时间。特别是没有注释的情况下。所以我觉得大型的项目，比如说内核还不如自己去写。我过去三四年的时间都花在推敲人家的代码上面，太苦了。而且也没有什么成果。

\*/

\*tmp = '\0';

//tmp+1是获得端口字符串地址，atoi把字符串80转变成数字，并存在proxyport里面。

proxyport = atoi(tmp + 1);

break;

case ':':

case 'h':

case '?':

usage();

return 2;

break;

case 'c':

clients = atoi(optarg);

break;

}

}

/\*

这是一个webbench命令行webbench -p host:port -c 5 -t 60 http://www.baidu.com/，最后这个百度网址http://www.baidu.com/不是任何参数和选项，会被getopt—long函数排在最后一个，并用optind指向他。

这个比较复杂一些说到底是设计getopt\_long设计的规则，参看下面的资料

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*资料二\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

转自：https://blog.csdn.net/chaoyue1216/article/details/7329788

刚开始接触 一些处理命令行参数的操作，开始不太明白，用例子测试了一下，感觉比以前明了多了。

命令行参数有长参数如version, 还有短参数 如 v, 那么用这两个都可以。程序处理的时候，会首先把长参数转换成对应的短参数，如会把version转成v, 再进行 v 对应的操作就可以了。

命令行参数的选项，有的需要参数，有的不需要参数，或者有的参数是可选的，那么怎么区分呢？

首先，对这些选项，如何组织起来？ 是以字符串的形式组织起来了。如我有一个程序，有两个选项，-a, -b, 我输入的时候是 ./a.out -a -b, 那么中间会处理成这种 ab这种字符串的形式，这个字符串就是所有的命令行的输入选项。区别是否有参数就在于此。如果某个选项必须有参数，则这一选项后有一个参数，如果参数是可选的，则其后面有两个冒号。如

-a 是没有参数的， -b 后面必须有参数， -c 后面是否有参数是可选的。则短的命令行选项是： ab:c::

下面我们通过一个简单的例子看一下。

#include <stdio.h>

#include <unistd.h>

#include <getopt.h>

char \*l\_opt\_arg;

char\* const short\_options = "myl:";

//char\* const short\_options = "nbl:";

struct option long\_options[] = {

{ "name", 0, NULL, 'm' }, //长选项对应的短选项参数， 第二个0表示选项后面无参数， 1为有参数，2为可选

{ "yourname", 0, NULL, 'y' },

{ "love", 1, NULL, 'l' },

{ 0, 0, 0, 0},

};

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

{

int c, i;

printf("before:\n");

for (i=1; i<argc; i++)

printf("arg:%d : %s\n", i, argv[i]);

printf("\n");

while((c = getopt\_long (argc, argv, short\_options, long\_options, NULL)) != -1)

{

// printf("optind:%d, %c \n", optind, c);

switch (c)

{

case 'm':

printf("My name is A.\n");

break;

case 'y':

printf("His name is B.\n");

break;

case 'l':

l\_opt\_arg = optarg;

printf("Our love is %s!\n", l\_opt\_arg);

break;

}

}

printf("optind:%d\n", optind);

printf("after:\n");

for (i=1; i<argc; i++)

printf("arg:%d : %s\n", i, argv[i]);

return 0;

}

注意，此程序可接收的的选项有三个， 一个是m ,不带参数， y 不带参数， l 要求有参数。

那如果-m 不带参数，如果我写了参数，会怎么样呢？下面看测试

在调用 getopt\_long 以后， optind 的值随之变化 。在while循环后，我们再把开始的命令行参数打印出来，看一下有什么不同。

把上面的代码命名为：　　getopt\_long.c

编译，可执行文件为 a.out

$ gcc getopt\_long.c

$ ./a.out -m -y

before:

arg:1 : -m

arg:2 : -y

My name is A.

His name is B.

optind:3

after:

arg:1 : -m

arg:2 : -y

$ ./a.out -m -y -l banana

before:

arg:1 : -m

arg:2 : -y

arg:3 : -l

arg:4 : banana

My name is A.

His name is B.

Our love is banana!

optind:5

after:

arg:1 : -m

arg:2 : -y

arg:3 : -l

arg:4 : banana

$./a.out -m lisi -y zhangsan -l banana aaa

before:

arg:1 : -m

arg:2 : lisi

arg:3 : -y

arg:4 : zhangsan

arg:5 : -l

arg:6 : banana

arg:7 : aaa

My name is A.

His name is B.

Our love is banana!

optind:5

after:

arg:1 : -m

arg:2 : -y

arg:3 : -l

arg:4 : banana

arg:5 : lisi

arg:6 : zhangsan

arg:7 : aaa

注意 argv 里面值的顺序已经和原来不一样了，对命令行的参数重新组织了一下顺序，也就是不认识的命令行参数，都放在了argv的最后，其中 optind 指向了这些没有被解释的参数的第一个。

optind有作用吧！如果你想输出哪些命令行参数没有被识别，可以打印出来 for (i=optind; i<argc; i++) printf("%s\n", argv[i]); 即可

附：如果是长参数，则使用 --, 如 --help, 因为 -help时，(选项不需要参数的情况) 会把它当成 四个选项， -h -e -l -p. 所以使用长参数时，要用两个 横线

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*资料结束。\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* 这里有一点一定要注意.。optind虽然为argv数组下标，正常情况下最大值是argc-1,但是getopt-long这个函数，会自动的在没有任何多余参数的情况下，要optind等于argc，所有才有下面这一句代码，去判断用户是不是有输入入网址。比如用户这样输入webbench -t 10 -c 20

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*以下为webbench代码\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (optind == argc)

{

fprintf(stderr, "webbench: Missing URL!\n");

usage();

return 2;

}

//防止用户将clients，benchtime设置为0

if (clients == 0) clients = 1;

if (benchtime == 0) benchtime = 30;//默认运行时间30

//输出版权信息

fprintf(stderr, "Webbench - Simple Web Benchmark "PROGRAM\_VERSION"\n"

"Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.\n"

);

/\*

现在getopt\_long执行完while循环后这个argv【optind】指向那个百度的网址的字符串，

所以就用这个字符串来基础上构建HTTP的request的消息。

构造好会放到全局变量request的字符串数组里

\*/

build\_request(argv[optind]);//构造请求消息体，参数为url

/\* print bench info \*///根据用户输入的method方法来，决定打印什么信息到界面

printf("\nBenchmarking: ");

switch (method)

{

case METHOD\_GET:

default:

printf("GET");

break;

case METHOD\_OPTIONS:

printf("OPTIONS");

break;

case METHOD\_HEAD:

printf("HEAD");

break;

case METHOD\_TRACE:

printf("TRACE");

break;

}

//这里会显示出百度网址

printf(" %s", argv[optind]);

switch (http10)

{

case 0:

printf(" (using HTTP/0.9)");

break;

case 1:

printf(" (using HTTP/1.0)");

break;

case 2:

printf(" (using HTTP/1.1)");

break;

}

printf("\n");

if (clients == 1) printf("1 client");

else

printf("%d clients", clients);

printf(", running %d sec", benchtime);

if (force) printf(", early socket close");

if (proxyhost != NULL) printf(", via proxy server %s:%d", proxyhost, proxyport);

if (force\_reload) printf(", forcing reload");

printf(".\n");

//执行测试工作，大部分的内容，核心内容都在bench这个函数

return bench();

}

/\*构造请求消息体\*/

void build\_request(const char \*url)//形参实际使用的函数参数是，optind指向的argv数组元素

{

//临时的字符串数组，组装request请求的时候用。

char tmp[10];

int i;

/\*

bzero函数的作用是将s指针指向的地址的n个字节清零，包括'\0'。

bzero的头文件：#include <string.h>

\*/

bzero(host, MAXHOSTNAMELEN);

bzero(request, REQUEST\_SIZE);

if (force\_reload && proxyhost != NULL && http10 < 1) http10 = 1; //缓存和代理都是http1.0后才有的

if (method == METHOD\_HEAD && http10 < 1) http10 = 1; //HEAD请求是http1.0后才有

if (method == METHOD\_OPTIONS && http10 < 2) http10 = 2; //OPTIONS和TRACE都是http1.1才有

if (method == METHOD\_TRACE && http10 < 2) http10 = 2;

//把get这些消息关键字放在request字符串的最前面

switch (method)

{

default:

case METHOD\_GET:

strcpy(request, "GET");

break;

case METHOD\_HEAD:

strcpy(request, "HEAD");

break;

case METHOD\_OPTIONS:

strcpy(request, "OPTIONS");

break;

case METHOD\_TRACE:

strcpy(request, "TRACE");

break;

}

/\*

前面用strc py拷贝，因为request里面是空的。

现在request里面有东西了，所以只能拼接了。

strcat是字符串连接函数，可以直接连接。

以下代码request加个空格request的内容由get变成get+“ ”

\*/

strcat(request, " ");

/\*

那这里strchr是查找字符，若有则返回地址，否则返回NULL。

url是optind指向的字符串，是这个函数的参数。

下面代码表示字符串里没有关键的地址符号

\*/

if (NULL == strstr(url, "://"))

{

fprintf(stderr, "\n%s: is not a valid URL.\n", url);

exit(2);

}

/\*

下面代码表示字符串太长了，也就是argv【optind】太长了，也就是用户输入的网址太长了，

所以出错。超出了程序预设缓冲区的大小吧

防止缓存区溢出是一种安全措施，见linux c编程实战P309

\*/

if (strlen(url) > 1500)

{

fprintf(stderr, "URL is too long.\n");

exit(2);

}

/\*

这个proxyhost（代理host）是可以为NULL的。

因为只有使用了-p选项后，才可以在之前给proxy赋值，不然，其一直为NULL

如果不是使用代理，那么就不能使用http之外的协议

\*/

/\*

strncasecmp函数

头文件：#include <string.h>

　　函数定义：int strncasecmp(const char \*s1, const char \*s2, size\_t n)

　　函数说明：strncasecmp()用来比较参数s1和s2字符串前n个字符，比较时会自动忽略大小写的差异

　　返回值 ：若参数s1和s2字符串相同则返回0 s1若大于s2则返回大于0的值 s1若小于s2则返回小于0的值

\*/

if (proxyhost == NULL)

if (0 != strncasecmp("http://", url, 7))

{

fprintf(stderr, "\nOnly HTTP protocol is directly supported, set --proxy for others.\n");

exit(2);

}

/\* protocol/host delimiter \*/

/\*

strstr（str1，str2）查找str2在str1中的位置,str2在str1的位置

如http://www.baidu.com

-url得到协议长度 即http的长度

+3结果为域名之前的长度 即http://

\*/

i = strstr(url, "://") - url + 3;

/\*可以在代码里面加上这句话把i打印出来， printf("%d\n，i）\*/

/\*计算出i来主要是为了分割字符串。

1.

那么这里的url+i就是从://后面开始的字符串

那这里strchr是查找字符，若有则返回地址，否则返回NULL。

下面这段代码是查找http：//www.baiu.com/是否有打最后的字符串。

URL语法错误——hostname没有以 / 结尾

2.我看到别人上面特别提醒，一定不能忘记了 / 这个，因为从这个程序来看，如果没有这个 / 是会退出程序的

3.不管是不是代理，这里都得是一定的URL要 / 在最后的

\*/

//这里用url+i的意思就是找http/后面的/

if (strchr(url + i, '/') == NULL)

{

fprintf(stderr, "\nInvalid URL syntax - hostname don't ends with '/'.\n");

exit(2);

}

//没有使用代理服务器

if (proxyhost == NULL)

{

/\* get port from hostname \*/

/\* 有端口数据 http://www.baidu.com:3300/ \*/

//index是查找字符串中第一个出现的字符，并返回该字符的地址，本质上和strchr()函数是一样的

//注意这里url+l是移动字符串指针，是在对网址进行一个操作，取子集。

if (index(url + i, ':') != NULL //这个判断是用户有输入端口

&& index(url + i, ':') < index(url + i, '/') //（这个判断是用户输入的端口的冒号在结束的|号之前)

{

//host和端口是分开拷贝的，下面这句话是拷贝host的。拷贝的是 http://www.baidu.com:3300/中的 www.baidu.com

strncpy(host, url + i, strchr(url + i, ':') - url - i)；

bzero(tmp, 10);

/\*

将这句话分解

strncpy(tmp, index(url + i, ':') + 1这个地址是：3300，这个冒号的地址+1 也就是3的位置，从这个位置来拷贝端口 ,

以下是计算拷贝多少 strchr(url + i, '/')这个是3300/最后这个/的地址 - index(url + i, ':')这个是：3300的冒号的地址 - 1)，这样就是要拷贝的地址长度了。;

\*/

strncpy(tmp, index(url + i, ':') + 1, strchr(url + i, '/') - index(url + i, ':') - 1); //复制端口号

//这个代码是用来测试tmp的值的，调试用的 printf("tmp=%s\n",tmp);

//因为如果URL: http://www.baidu.com:/ 这个样子tmp就是NULL了

proxyport = atoi(tmp);

//当用户没有输入端口号的时候，tmp为0，这样proxyport也为0，所以就默认使用80的端口。

if (proxyport == 0) proxyport = 80;

}

else

{

/\*

下面的代码是获取当用户只输入地址没有端口的情况htt：//www.baidu.com，

这样直接把www.baidu.com的域名字符串拷贝到host字符串，

host字符串是放到socket那个函数去使用的，区别于requset字符串，不要混淆。

在两个 / 之间就是完整的ip地址，在函数开始，就将host清空了，所以这里的复制不用担心，例如： http://www.baidu.com/

strcspn(a,b)函数，会返回a中从开头连续不含b字符串的个数注意这里的url+i是指向 http://的末尾的。

\*/

strncpy(host, url + i, strcspn(url + i, "/"));

}

//可以用这个代码 printf("Host=%s\n",host)看看host的值;

//注释request + strlen(request)表示前面已经写好的字符串跳过，不要被篡改

strcat(request + strlen(request), url + i + strcspn(url + i, "/"));

}

//使用代理服务器

else

{

// printf("ProxyHost=%s\nProxyPort=%d\n",proxyhost,proxyport);

/\*

strcat(a,b)函数是会将b接在a的后面，不会覆盖原来字符串a的内容

\*/

strcat(request, url); //使用了代理服务器后，其请求行必须是完整的UIR

}

//填写http协议版本到请求行

if (http10 == 1)

strcat(request, " HTTP/1.0");

else if (http10 == 2)

strcat(request, " HTTP/1.1");

strcat(request, "\r\n");

//请求报头

if (http10 > 0)

strcat(request, "User-Agent: WebBench "PROGRAM\_VERSION"\r\n");

if (proxyhost == NULL && http10 > 0)

{

strcat(request, "Host: ");

strcat(request, host); //填写域名

strcat(request, "\r\n");

}

if (force\_reload && proxyhost != NULL)

{

strcat(request, "Pragma: no-cache\r\n");

}

if (http10 > 1)

strcat(request, "Connection: close\r\n");

/\* add empty line at end \*/

if (http10 > 0) strcat(request, "\r\n");

// printf("Req=%s\n",request);

}

/\* vraci system rc error kod \*/

/\*

父进程创建clients个子进程，父子进程通过pipe通信

\*/

static int bench(void)

{

int i, j, k;

pid\_t pid = 0;

//这里的f就是去打开对应管道文件的。

FILE \*f;

/\* check avaibility of target server \*/

//这里的proxyport不单单是proxyhost的端口，也是真实ip的端口

i = Socket(proxyhost == NULL ? host : proxyhost, proxyport);

if (i < 0)

{

fprintf(stderr, "\nConnect to server failed. Aborting benchmark.\n");

return 1;

}

//为什么这里一来就要将文件描述符i给关闭？

//因为这里只是测试一下服务器能否连接，失败就退出，否则关闭，不关的话就会浪费socket资源

close(i);

/\* create pipe \*/

if (pipe(mypipe))

{

perror("pipe failed.");

return 3;

}

/\* not needed, since we have

() in childrens \*/

/\* wait 4 next system clock tick \*/

/\*time函数获取当前的时间，以下这个注释的语句是在当前的时间里，父亲进程不执行的，让出执行时间的。sched——yield就是这个意思。

cas=time(NULL);

while(time(NULL)==cas)

sched\_yield();

\*/

/\* fork childs \*/

for (i = 0; i < clients; i++)

{

pid = fork(); //fork函数创建子进程，并返回子进程id，子进程中pid等于0

if (pid <= (pid\_t)0)

{

/\* child process or error\*/

sleep(1); /\* make childs faster \*/

break;

}

}

if (pid < (pid\_t)0)

{

fprintf(stderr, "problems forking worker no. %d\n", i);

perror("fork failed.");

return 3;

}

if (pid == (pid\_t)0)

{

/\* I am a child 根据proxyhost的值来判断是否使用了代理\*/

if (proxyhost == NULL)

benchcore(host, proxyport, request);

else

benchcore(proxyhost, proxyport, request);

/\* write results to pipe注意这里是子进程打开pipe的写端，写speed，failed，byte三个值进去 \*/

f = fdopen(mypipe[1], "w");

if (f == NULL)

{

perror("open pipe for writing failed.");

return 3;

}

/\* fprintf(stderr,"Child - %d %d\n",speed,failed); \*/

/\*

speed，bytes，fail在benchcore里随着连接和接受数据作更改的。

这里是测试完成之后，每个子进程往管道里写数据，方便父进程读出数据来汇总。

管道是通过这三个元素的结构体数组来传递数据的。

\*/

fprintf(f, "%d %d %d\n", speed, failed, bytes);

fclose(f);

return 0;

}//子进程的代码到这里结束了

else

{

//注意子进程测试结束之后才会往管道写数据，所以父亲进程只需要一直去往管道里读取数据就可以了。

/\*I am a parent\*/

f = fdopen(mypipe[0], "r");

if (f == NULL)

{

perror("open pipe for reading failed.");

return 3;

}

//不设置读的缓冲

//不使用库函数的任何缓冲机制。但是不知道为什么要讲这句话

setvbuf(f, NULL, \_IONBF, 0);

//初始化最后要输出的父亲进程的speed，failed，bytes，其实可以命名上作f—speed更好。这样更容易区别于子进程的speed，更容易理解。

speed = 0;

failed = 0;

bytes = 0;

while (1)

{

//从管到你读数据，一次读三个。读完之后数据就会消失掉

pid = fscanf(f, "%d %d %d", &i, &j, &k);

if (pid < 2)

{

fprintf(stderr, "Some of our childrens died.\n");

break;

}

//ijk是用来做临时变量存储speed，fail byte的值的。

speed += i;

failed += j;

bytes += k;

/\* fprintf(stderr,"\*Knock\* %d %d read=%d\n",speed,failed,pid); \*/

//有多少个子进程就读取多少次

if (--clients == 0) break;

}

fclose(f);

//最后把结果转化的显示出来

printf("\nSpeed=%d pages/min, %d bytes/sec.\nRequests: %d susceed, %d failed.\n",

(int)((speed + failed) / (benchtime / 60.0f)),

(int)(bytes / (float)benchtime),

speed,

failed);

}

return i;

}

/\*

子进程通过socket向被测服务器发送请求，该过程会改变speed、failed、bytes的值，在执行完benchcore以后，这些值会通过管道汇总。

\*/

void benchcore(const char \*host, const int port, const char \*req)

{

int rlen;

char buf[1500];

// s是sock 。i用来算从网站读取到的数据的

int s, i;

struct sigaction sa;

/\* setup alarm signal handler \*/

sa.sa\_handler = alarm\_handler;

sa.sa\_flags = 0; //sa\_flags包含了许多标志位，等于0表示清空所有标志

if (sigaction(SIGALRM, &sa, NULL))

exit(3);

alarm(benchtime);

rlen = strlen(req);

nexttry:

while (1)

{

//时间到后，会跳转到alarm\_handler()函数，将timerexpired设置为 1

if (timerexpired)

{

//为什么此时会要failed--?因为这次的连接失败是时间到了造成的。

if (failed > 0)

{

/\* fprintf(stderr,"Correcting failed by signal\n"); \*/

failed--;

}

return;

}

//Socket函数看后面的定义

s = Socket(host, port);

if (s < 0)

{

failed++; //连接失败，failed++

continue;

}

//下面write的函数s这个参数是sock，等于是往sock里写东西

if (rlen != write(s, req, rlen))

{

failed++; //向服务器写入数据失败，failed++

close(s);

continue;

}

if (http10 == 0)

if (shutdown(s, 1))

{

failed++; //1为标准输出，这里关闭了客户端的输出，即中止写入服务器数据操作，成功为0，失败为-1

close(s);

continue;

}

/\*force==0代表读取服务器发回来的数据\*/

if (force == 0)

{

/\* read all available data from socket \*/

while (1)

{

if (timerexpired) break;

i = read(s, buf, 1500);

/\* fprintf(stderr,"%d\n",i); \*/

/\*读取服务器发送过来的数据失败，failed++\*/

if (i < 0)

{

failed++;

close(s);

//重新连接并测试

goto nexttry;

}

else if (i == 0) break;

else

bytes += i;

}

}

//sock无法正常关闭，失败加1

if (close(s))

{

failed++;

continue;

}

speed++;

}

}

// webbench.cpp : 此文件包含 "socket.c" 函数。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

module: socket.c

program: popclient

SCCS ID: @(#)socket.c 1.5 4/1/94

programmer: Virginia Tech Computing Center

compiler: DEC RISC C compiler (Ultrix 4.1)

environment: DEC Ultrix 4.3

description: UNIX sockets code.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <sys/types.h>

#include <sys/socket.h>

#include <fcntl.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <sys/time.h>

#include <string.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <stdarg.h>

/\*\*\*\*\*\*\*\*\*\*\*

功能：通过地址和端口建立网络连接

host:网络地址

clientPort:端口

Return：建立的socket连接。

如果返回-1，表示建立连接失败

\*\*\*\*\*\*\*\*\*\*\*\*/

int Socket(const char \*host, int clientPort)

{

int sock;

unsigned long inaddr;

struct sockaddr\_in ad;

struct hostent \*hp;

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

ad.sin\_family = AF\_INET; //AF（Address Family） internet家族AF\_INET

inaddr = inet\_addr(host);/\*此inet\_addr函数将点分的十进制的IP转为无符号长整形，host可能是www.baidu.com也可能是IP地址数字的格式，所以下面的if判断。

\*/

if (inaddr != INADDR\_NONE)

// when输入为正确的IP地址时候

memcpy(&ad.sin\_addr, &inaddr, sizeof(inaddr));

else

//如果host是域名，输入的是网址，结果就是错误

{

hp = gethostbyname(host);//用域名获取IP

if (hp == NULL)

return -1;

memcpy(&ad.sin\_addr, hp->h\_addr, hp->h\_length);

}

ad.sin\_port = htons(clientPort); //将主机的无符号短整形数转换成网络字节顺序

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

if (sock < 0)

return sock;

//连接,其中sockaddr\_in和sockaddr二者长度一样，都是16个字节，即占用的内存大小是一致的，因此可以互相转化。二者是并列结构，指向sockaddr\_in结构的指针也可以指向sockaddr

if (connect(sock, (struct sockaddr \*)&ad, sizeof(ad)) < 0)

return -1;

return sock;

}