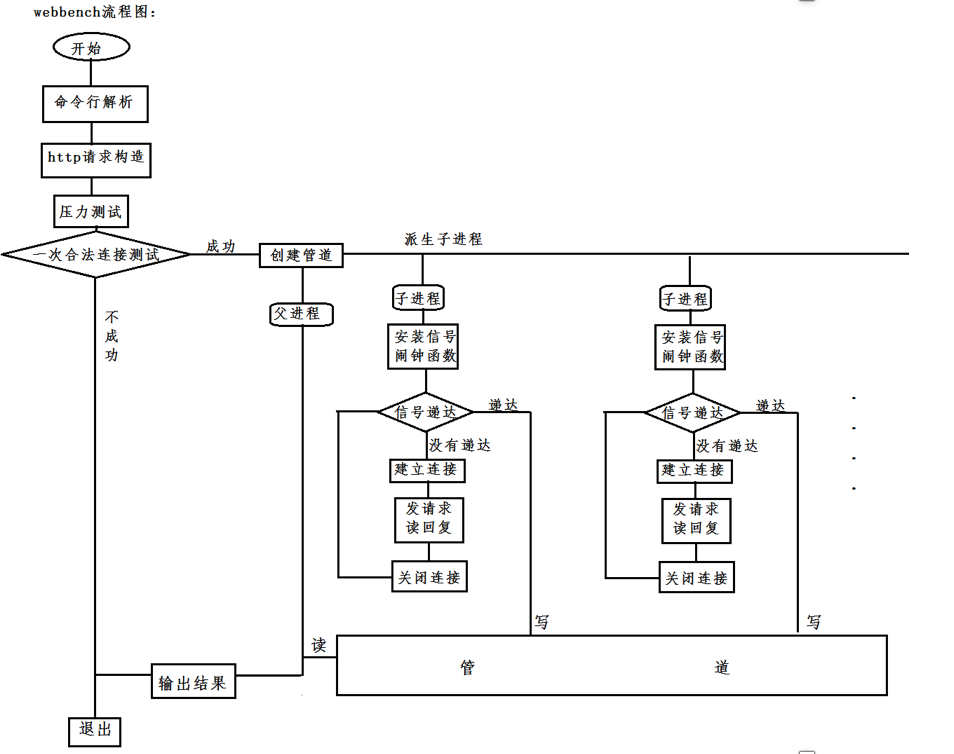
webbench:其为linux上一款web性能压力测试工具，它最多可以模拟3万个并发连接数来测试服务器压力，其原理为fork多个子进程，每个子进程都循环做web访问测试，子进程将访问的结果通过管道告诉父进程，父进程做最终结果统计。

其主要原理如下图：



其代码实现中主要运用4个函数：getopt\_long()系统命令行解析函数,build\_request()函数，bench()函数，benchcore()函数。

流程如下：

webbench.c:

全局变量选项参数：都有默认值

1.命令行参数解析，参数构造（getopt\_long()函数实现）（若用户没有传

参设定则为其默认值）；

2.根据1得到的一些参数，进行http请求构造，调用build\_request()函数：

(1)构造第一行：

请求方法：method–>request[]

url：判断url合法性,合法：

无代理服务器：

将其网络地址–>host[],端口号–>proxyport;

url资源路径–>request[]

有代理服务器：

url直接填入–>request[];

http协议版本：http0.9 http1.0 http1.1

(2)请求报头：Name:Value–>request[]

(3)填入空行–>request[]

构造完成

3.进行压力测试，调用bench()函数：

(1)进行一次连接合法性测试；

(2)建立管道通信；

(3)按照客户端数目派生子进程；

(4)每个子进程调用benchcore()函数进行连接请求：

benchcore函数：

创建自定义信号捕捉函数，建立benchtime时间闹钟：

(1)调用Socket()，得到连接套接字；

(2)向网络中写请求；

(3)http0.9协议处理(关闭连接写一半)；

(4)读取服务器响应消息；

(5)关闭连接；

在以上过程中统计各自进程的faild(失败次数)、bytes(服务器回应字节 数)、speed(连接成功次数)；

(5)benchcore函数调用完成，每个进程向管道中写入faild、bytes、speed；

(6)父进程打开管道读取每个进程写入的信息，进行统计计算，输出压力测试结果。

Socket.c:

1.建立套接字：调用socket();

2.建立连接：调用connect();

返回套接字：sock.

源码如下：

webbench.c

#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>

//统计的压力测试最终结果表示

volatile int timerexpired = 0;

int speed = 0;

int failed = 0;

int bytes = 0;

//http请求方法

#define METHOD\_GET 0

#define METHOD\_HEAD 1

#define METHOD\_OPTIONS 2

#define METHOD\_TRACE 3

#define PROGRAM\_VERSION "1.5"

// 默认设置:一般需用户自己传入命令行参数设置

int method = METHOD\_GET; //默认请求方法为GET方式

int clients = 1; //默认只模拟一个客户端

int force = 0; //默认需要等待服务器响应

int force\_reload = 0; //失败时重新请求

int proxyport = 80; //默认访问服务器端口为80

char \*proxyhost = NULL; //默认无代理服务器

int benchtime = 30; //默认模拟请求时间为30s

// globals 版本号

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

int mypipe[2]; //管道用于父子进程通信

char host[MAXHOSTNAMELEN]; //存储服务器网络地址

#define REQUEST\_SIZE 2048

char request[REQUEST\_SIZE]; //存放http请求报文信息数组

//函数声明

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

static int bench(void);

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

// 用法与各参数详细含义

static void usage(void)

{

fprintf(stderr,

"webbench [option]... URL\n" //用法

" -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"

);

};

//结构体数组：每一个元素格式为：{长选项，选项后是否带有参数，int\*指针（为NULL），对应短选项或

static const struct option long\_options[]= //不为NULL，将第四个参数值给第三个参数}

{

{"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}

};

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

{

int opt = 0;

int options\_index = 0;

char \*tmp = NULL;

//一、检验命令行参数

//1.不带选项时直接输出用法help信息

if(argc == 1)

{

usage();

return 2;

}

//2.带选项时则解析命令行参数并根据传入选项进行相关设置

// getopt\_long 为命令行解析的库函数，根据argc来寻找(argv,"912Vfrt:p:c:?h")这两个字符串匹配的选项，

//如果是短选项，则直接返回这个选项给opt，

//如果是长选项，则到option long\_options[]结构体数组中寻找匹配其长选项，返回其对应的短选项给opt，

//若其第三个参数不为NULL，将第四个参数值给第三个参数，并且返回0给opt

//此函数自带全局变量：

//optarg：指向选项后的参数：-t 100,指向100；

//optind: 当前访问到的argv索引值

//opterr: 其值非0时，代表有无效选项，缺少参数，输出错误信息

//optopt: 发现无效选项时，函数返回“? / :”,将其值设为无效选项字符

while((opt = getopt\_long(argc,argv,"912Vfrt:p:c:?h",long\_options/\*结构体数组指针\*/,&options\_index)) != EOF )

{

switch(opt) //根据返回值判断用户传入的参数进行相关设置

{

case 0 : break;

case 'f': force = 1;break; //force=1代表不等待服务器响应

case 'r': force\_reload = 1;break; //发送重新加载请求

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); //设置用户传入的运行时间

break;

case 'c':

clients = atoi(optarg); //设置创建的客户端数

break;

case 'p':

//使用代理服务器，设置其代理网络号和端口号：格式：-p server:port

tmp = strrchr(optarg,':'); //查找“：”在optarg中最后一次出现的位置

proxyhost = optarg; //设置网络号

if(tmp == NULL) //没有：号，没有端口号

{

break;

}

if(tmp == optarg) //端口号在首位置，错误：缺失主机名

{

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

return 2;

}

if(tmp == optarg + strlen(optarg)-1) //：号在末位，缺少端口号

{

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

return 2;

}

\*tmp = '\0'; //将：号置为“\0”

proxyport = atoi(tmp+1); //设置新的端口号

break;

case ':':

case 'h':

case '?': usage();return 2;break;

}

}

//getopt\_long函数将选项解析完成后，读到url不会在读取，此时argv[optind]指向url

//optind 被 getopt\_long设置为命令行参数中未读取的下一个元素下标值

if(optind == argc) //若相等即没有输入URL

{

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

usage();

return 2;

}

//若客户端选项后参数设为0，则更改

if(clients == 0)

clients = 1;

if(benchtime == 0)

benchtime = 60;

//输出webbench版本相关信息

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

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

);

//二、构造HTTP请求到request数组

build\_request(argv[optind]); //传入URL

//http请求构造成功后

//以下输出提示信息

printf("\nBenchmarking: "); //测压开始

switch(method) //用的请求方法

{

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;

}

printf(" %s",argv[optind]); //访问的url

switch(http10) //http协议版本号

{

case 0: printf(" (using HTTP/0.9)");break;

case 2: printf(" (using HTTP/1.1)");break;

}

printf("\n");

//模拟连接客户端数目

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

else

printf("%d clients",clients);

//连接测试的时间

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

if(force)

printf(", early socket close");

//输出代理服务器的信息

if(proxyhost != NULL)

printf(", via proxy server %s:%d",proxyhost,proxyport);

if(force\_reload)

printf(", forcing reload");

printf(".\n");

//开始压力测试，返回 bench 函数执行结果

return bench();

}

//二、构造HTTP请求到request数组

void build\_request(const char \*url)

{

char tmp[10];

int i;

//初始化

bzero(host,MAXHOSTNAMELEN);

bzero(request,REQUEST\_SIZE);

//判断应该使用的 HTTP 协议

if(force\_reload && proxyhost != NULL && http10 < 1)

http10 = 1;

if(method == METHOD\_HEAD && http10 < 1)

http10 = 1;

if(method == METHOD\_OPTIONS && http10 < 2)

http10 = 2;

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

http10 = 2;

//1.填写http请求第一行

//填写请求方法method

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;

}

strcat(request," ");

//URL 合法性判断

//若没有"://"则不合法

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

{

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

exit(2);

}

//若url过长非法

if(strlen(url)>1500)

{

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

exit(2);

}

if(proxyhost == NULL) //若无代理服务器

{

if(0 != strncasecmp("http://",url,7)) //忽略大小写比较

{

//只支持 HTTP 地址

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

exit(2);

}

}

//找到主机名开始的地方:如：http://baidu.com:80/

i = strstr(url,"://")-url+3; //i==7

// 必须以 / 结束

if(strchr(url+i,'/')==NULL) //在字符串中寻找“/” ，找不到则非法URL

{

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

exit(2);

}

if(proxyhost == NULL) //若无代理服务器

{

// 得到端口号从主机名

if(index(url+i,':') != NULL && index(url+i,':') < index(url+i,'/')) //若带有端口号，index函数与strchr相似

{

//设置网络号

strncpy(host,url+i,strchr(url+i,':')-url-i); //如将baidu.com拷贝到host数组里即网络地址

//初始化

bzero(tmp,10);

strncpy(tmp,index(url+i,':')+1,strchr(url+i,'/')-index(url+i,':')-1); //将端口号拷贝到tmp数组中

//设置端口

proxyport = atoi(tmp);

if(proxyport==0)

proxyport=80;

}

else //没有端口号，直接拷贝域名到host数组中

{

strncpy(host,url+i,strcspn(url+i,"/")); //strcspn找url+i到“/”之间的字符个数

}

//将资源路径填入请求行里

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

}

else //若有代理服务器

{

strcat(request,url); //直接填入URL到请求行中

}

//填入http版本号到请求行中

if(http10 == 1)

strcat(request," HTTP/1.0");

else if (http10==2)

strcat(request," HTTP/1.1");

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

//2.填请求报头：NAME:VALUE

if(http10 > 0)

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

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

{

strcat(request,"Host: ");

strcat(request,host);

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");

//3.填入空行

if(http10>0)

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

//构造完成

}

static int bench(void) //父进程做的工作

{

int i,j,k;

pid\_t pid = 0;

FILE \*f;

//建立网络连接 ：先测试一次，服务器是否可以正常连接成功

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

if(i < 0)

{

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

return 1;

}

close(i); //测试成功，一次连接完成关闭

//建立管道通信

if(pipe(mypipe))

{

perror("pipe failed.");

return 3;

}

//派生子进程进行压力测试 ：传入多少个客户端则建立多少个子进程进行连接

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

{

pid = fork();

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

{

sleep(1);

break; //使子进程立刻跳出循环，要不就子进程继续 fork 了

}

}

//子进程创建失败

if( pid < (pid\_t)0)

{

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

perror("fork failed.");

return 3;

}

//子进程执行

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

{

//子进程发出实际请求

if(proxyhost == NULL)

benchcore(host,proxyport,request);

else

benchcore(proxyhost,proxyport,request);

// 打开管道写:连接请求状态的信息

f = fdopen(mypipe[1],"w"); //将文件描述符转换为文件指针

if(f == NULL)

{

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

return 3;

}

//写入f文件中此进程在一定时间中请求成功的次数，失败的次数，读取服务器回复的总字节数

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

fclose(f);

return 0;

}

else {

//父进程打开管道读

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

if(f == NULL)

{

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

return 3;

}

setvbuf(f,NULL,\_IONBF,0); //设置f的缓冲区为无缓冲区

speed = 0; //连接成功总次数

failed = 0; //失败请求数

bytes = 0; //传输字节数

while(1)

{

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

if(pid<2)

{

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

break;

}

speed += i; //连接成功总次数

failed += j; //连接失败总次数

bytes += k; //传输总字节数

//子进程是否读取完

if(--clients == 0)

break;

}

fclose(f);

//统计结果计算

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

(int)((speed+failed)/(benchtime/60.0f)), //总连接次数/总时间=每分钟请求连接次数

(int)(bytes/(float)benchtime), //每秒传输字节数

speed, //连接成功次数

failed); //连接失败次数

}

return i;

}

//信号处理函数

static void alarm\_handler(int signal)

{

timerexpired = 1;

}

//子进程处理发起请求

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

{

int rlen;

char buf[1500];

int s,i;

struct sigaction sa;

//安装信号

sa.sa\_handler = alarm\_handler;

sa.sa\_flags = 0;

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

exit(3);

//设置闹钟函数

alarm(benchtime);

rlen = strlen(req);

nexttry:

while(1){

//收到信号则使 timerexpired = 1

if(timerexpired)

{

if(failed > 0)

{

failed--;

}

return;

}

//建立 socket连接, 进行 HTTP 请求

s = Socket(host,port);

if(s < 0)

{

failed++; //连接失败则++

continue;

}

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

{

failed++; //写失败++

close(s);

continue;

}

//HTTP 0.9 的处理

if(http10==0)

// 如果关闭不成功

if(shutdown(s,1)) //关闭连接写一半

{

failed++;

close(s);

continue;

}

// -f 选项时未设置时等待读取服务器回复

if(force == 0)

{

while(1)

{

if(timerexpired)

break;

i = read(s,buf,1500);

if(i<0)

{

failed++; //读失败++

close(s);

goto nexttry;

}

else

if(i == 0) //读完退出

break;

else

bytes+=i; //统计服务器回复的字节数

}

}

if(close(s))

{

failed++; //关闭失败++

continue;

}

speed++; //成功连接一次++一次

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131
* 132
* 133
* 134
* 135
* 136
* 137
* 138
* 139
* 140
* 141
* 142
* 143
* 144
* 145
* 146
* 147
* 148
* 149
* 150
* 151
* 152
* 153
* 154
* 155
* 156
* 157
* 158
* 159
* 160
* 161
* 162
* 163
* 164
* 165
* 166
* 167
* 168
* 169
* 170
* 171
* 172
* 173
* 174
* 175
* 176
* 177
* 178
* 179
* 180
* 181
* 182
* 183
* 184
* 185
* 186
* 187
* 188
* 189
* 190
* 191
* 192
* 193
* 194
* 195
* 196
* 197
* 198
* 199
* 200
* 201
* 202
* 203
* 204
* 205
* 206
* 207
* 208
* 209
* 210
* 211
* 212
* 213
* 214
* 215
* 216
* 217
* 218
* 219
* 220
* 221
* 222
* 223
* 224
* 225
* 226
* 227
* 228
* 229
* 230
* 231
* 232
* 233
* 234
* 235
* 236
* 237
* 238
* 239
* 240
* 241
* 242
* 243
* 244
* 245
* 246
* 247
* 248
* 249
* 250
* 251
* 252
* 253
* 254
* 255
* 256
* 257
* 258
* 259
* 260
* 261
* 262
* 263
* 264
* 265
* 266
* 267
* 268
* 269
* 270
* 271
* 272
* 273
* 274
* 275
* 276
* 277
* 278
* 279
* 280
* 281
* 282
* 283
* 284
* 285
* 286
* 287
* 288
* 289
* 290
* 291
* 292
* 293
* 294
* 295
* 296
* 297
* 298
* 299
* 300
* 301
* 302
* 303
* 304
* 305
* 306
* 307
* 308
* 309
* 310
* 311
* 312
* 313
* 314
* 315
* 316
* 317
* 318
* 319
* 320
* 321
* 322
* 323
* 324
* 325
* 326
* 327
* 328
* 329
* 330
* 331
* 332
* 333
* 334
* 335
* 336
* 337
* 338
* 339
* 340
* 341
* 342
* 343
* 344
* 345
* 346
* 347
* 348
* 349
* 350
* 351
* 352
* 353
* 354
* 355
* 356
* 357
* 358
* 359
* 360
* 361
* 362
* 363
* 364
* 365
* 366
* 367
* 368
* 369
* 370
* 371
* 372
* 373
* 374
* 375
* 376
* 377
* 378
* 379
* 380
* 381
* 382
* 383
* 384
* 385
* 386
* 387
* 388
* 389
* 390
* 391
* 392
* 393
* 394
* 395
* 396
* 397
* 398
* 399
* 400
* 401
* 402
* 403
* 404
* 405
* 406
* 407
* 408
* 409
* 410
* 411
* 412
* 413
* 414
* 415
* 416
* 417
* 418
* 419
* 420
* 421
* 422
* 423
* 424
* 425
* 426
* 427
* 428
* 429
* 430
* 431
* 432
* 433
* 434
* 435
* 436
* 437
* 438
* 439
* 440
* 441
* 442
* 443
* 444
* 445
* 446
* 447
* 448
* 449
* 450
* 451
* 452
* 453
* 454
* 455
* 456
* 457
* 458
* 459
* 460
* 461
* 462
* 463
* 464
* 465
* 466
* 467
* 468
* 469
* 470
* 471
* 472
* 473
* 474
* 475
* 476
* 477
* 478
* 479
* 480
* 481
* 482
* 483
* 484
* 485
* 486
* 487
* 488
* 489
* 490
* 491
* 492
* 493
* 494
* 495
* 496
* 497
* 498
* 499
* 500
* 501
* 502
* 503
* 504
* 505
* 506
* 507
* 508
* 509
* 510
* 511
* 512
* 513
* 514
* 515
* 516
* 517
* 518
* 519
* 520
* 521
* 522
* 523
* 524
* 525
* 526
* 527
* 528
* 529
* 530
* 531
* 532
* 533
* 534
* 535
* 536
* 537
* 538
* 539
* 540
* 541
* 542
* 543
* 544
* 545
* 546
* 547
* 548
* 549
* 550
* 551
* 552
* 553
* 554
* 555
* 556
* 557
* 558
* 559
* 560
* 561
* 562
* 563
* 564
* 565
* 566
* 567
* 568
* 569
* 570
* 571
* 572
* 573
* 574
* 575
* 576
* 577

Socket.c

#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>

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

{

int sock;

unsigned long inaddr;

struct sockaddr\_in ad;

struct hostent \*hp;

// 初始化地址

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

ad.sin\_family = AF\_INET;

// 尝试把主机名转化为数字

inaddr = inet\_addr(host);

if (inaddr != INADDR\_NONE)

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

else

{

// 取得 ip 地址

hp = gethostbyname(host); //得到主机的二进制Ip给hp->h\_addr

if (hp == NULL)

return -1;

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

}

ad.sin\_port = htons(clientPort);

// 建立 socket

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

if (sock < 0)

return sock;

// 建立链接

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

return -1;

return sock;

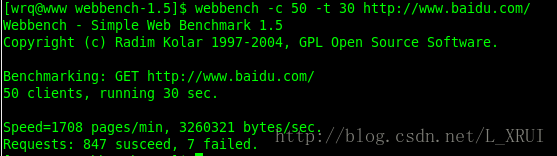
}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48

webbench参数含义如下：



在centos 6.5 测试使用结果如下：



即以上模拟50个客户端在30秒期间并发请求百度，结果如下：

每分钟平均有1708次请求连接，服务器每秒传输字节为3260321，在30秒期间请求连接成功为847次，失败7次。

其中webbench的优点为：

1.部署简单，适用于小型网站压力测试，（最多可模拟3万并发）；

2.它具有静态页面测试能力也支持动态页面（ASP,PHP,JAVA,CGI）进行测试能力；

3.支持对含有SSL的安全网站如电子商务网站进行动态或静态性能测试；

缺点为：

1.不适合中大型网站测试；

2.其并发采用多进程实现并非线程，长时间其会大量占用内存与CPU，所以一般长时间的压力测试不推荐使用webbench.