

网站压力测试工具 Webbench 源码解析

By Hov

目 录

1 前言.....	3
2 整体架构.....	3
2.1 Webbench 程序结构.....	3
2.2 Webbench 程序流程.....	4
3 函数分析.....	5
3.1 main()函数.....	5
3.1.1 函数功能.....	5
3.1.2 main()函数源码及注释:	6
3.2 bench()函数	8
3.2.1 函数功能.....	8
3.2.2 bench()函数源码及注释:	8
3.3 benchcore()函数	10
3.3.1 函数功能.....	10
3.3.2 benchcore()函数源码及注释:	11
3.4 build_request()函数	12
3.4.1 函数功能.....	12
3.4.2 build_request()函数源码及注释:	13
3.5 alarm_handler()函数.....	15
3.5.1 函数功能.....	15
3.5.2 alarm_handler()函数源码及注释:	15
3.6 Socket()函数	15
3.6.1 函数功能.....	15
3.6.2 Socket()函数源码及注释:	15
3.7 usage()函数.....	16
3.7.1 函数功能.....	16
3.7.2 usage()函数源码及注释:	16
3.8 库函数.....	17
3.8.1 getopt_long ()	17
3.8.2 pipe ()	17
3.8.3 fork ()	18
3.8.4 fdopen ()	18
3.8.5 fclose ()	19
3.8.6 setvbuf ()	19
3.8.7 sigaction ()	19
3.8.8 inet_addr ()	19
3.8.9 gethostbyname ()	19
3.8.10 htons ()	20
4 软件演示.....	20
5 结语.....	22

1 前言

Webbench 是一款由 Lionbridge 公司开发的在 LINUX 下使用的简单高效的网站压力测试工具，是一款采用 C 语言开发的开源软件。它可以测试相同硬件上不同服务的性能以及不同硬件上同一个服务的运行状况。Webbench 的标准测试可以向我们展示服务器的两项内容：每秒钟相应请求数和每秒钟传输数据量。它不但能测试静态页面，还能对动态页面（ASP,PHP,JAVA,CGI）进行测试，同时支持对含有 SSL 的安全网站（例如电子商务网站）进行静态或动态的性能测试。值得注意的是，Webbench 程序的完整源代码仅仅只有 600 行左右，被认为是学习 LINUX 下 C 编程的经典程序。

2 整体架构

2.1 Webbench 程序结构

Webbench 程序源代码由两个文件构成，分别是 webbench.c 文件和 socket.c 文件。程序的主要功能实现代码大部分在 webbench.c 文件中， socket.c 文件中只定义了一个函数 Socket（），用于创建 socket 连接。

Webbench 程序文件结构如图 1 所示。

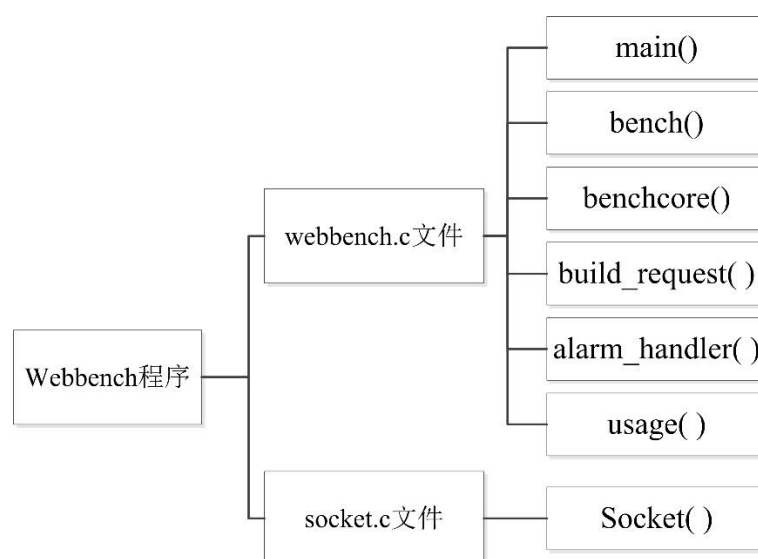


图 1 Webbench 程序结构

webbench.c 文件中包含了程序的主要代码，在该文件中有几个主要函数，尤其重要的是 `bench()` 和 `benchcore()` 两个函数。文件中几个主要的函数功能说明如下：

- (1) `bench()` 函数：用于创建管道并派发子进程，调用测试函数；
- (2) `benchcore()` 函数：HTTP 请求测试函数；
- (3) `build_request()` 函数：封装 HTTP 请求信息；
- (4) `usage()` 函数：显示使用说明。

Webbench 程序函数之间调用关系如图 2 所示。

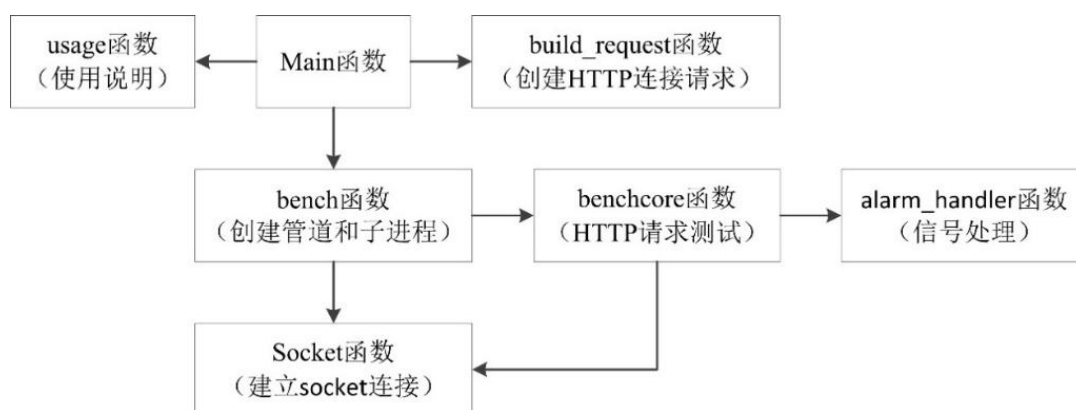


图 2 webbench 程序函数关系图

2.2 Webbench 程序流程

Webbench 程序通过派生若干个子进程模拟多个客户端同时访问某个 URL 地址，以此进行网站压力测试。大体流程如下：

(1) 解析命令行输入参数。通过参数获取测试并发数、测试时间、目标 URL 地址等数据；

(2) 封装 HTTP 请求信息。根据获取的 URL 地址，封装一个完整的 HTTP 请求信息，为后续测试工作做准备；

(3) 准备工作就绪，开始进行网站压力测试。`main()` 函数最后调用 `bench()` 函数，创建管道用于父子进程间的数据传输，并根据设定的并发数派生出相应数量的子进程。每个子进程调用 `benchcore()` 函数进行 HTTP 请求测试，并

将测试结果写入管道。最终父进程从管道读取子进程写入的数据，直到读取完所有的子进程的测试结果，将最终测试结果输出到屏幕上。

程序具体流程如图 3 所示。

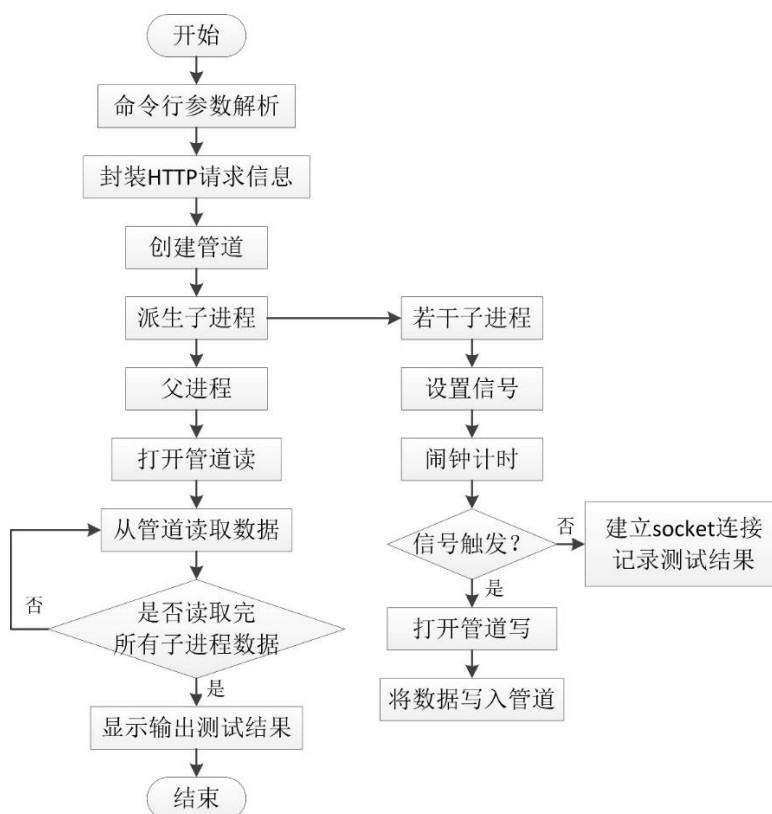


图 3 程序流程图

3 函数分析

3.1 main()函数

3.1.1 函数功能

main()函数，程序执行起点。main()函数刚开始执行时主要是完成准备工作，而程序真正的测试过程是当main()函数执行到最后一句时才开始的，也就是对bench()函数的调用。

main()函数首先解析命令行输入参数，接着调用build_request()函数封装HTTP请求信息，当准备工作就绪后，才调用bench()函数开始进行测试，最终将所有测试结果收集并输出到屏幕。

3.1.2 main()函数源码及注释:

```
int main(int argc, char *argv[])
{
    int opt=0;
    int options_index=0;
    char *tmp=NULL;
    if(argc==1)    //命令行无参数
    {
        usage();    //调用 usage()函数: 使用说明
        return 2;
    }
    //解析命令行参数
    while((opt=getopt_long(argc,argv,"912Vftr:p:c:?h",long_options,&options_index
))!=EOF )
    {
        switch(opt) //相应命令行参数
        {
            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);    //输出版本号
            case 't': benchtime=atoi(optarg);break;    //测试时间
            case 'p':
                /* proxy server parsing server:port */
                tmp=strchr(optarg,':');
                proxyhost=optarg;
                if(tmp==NULL)
                {
                    break;
                }
                if(tmp==optarg)
                {
                    fprintf(stderr,"Error in option --proxy %s: Missing
hostname.\n",optarg);
                    return 2;
                }
                if(tmp==optarg+strlen(optarg)-1)
                {
```

```

        fprintf(stderr,"Error in option --proxy %s Port number is
missing.\n",optarg);
        return 2;
    }
    *tmp='\0';
    proxyport=atoi(tmp+1);break;    //端口号
case ':':
case 'h':
case '?': usage();return 2;break;
case 'c': clients=atoi(optarg);break;    //并发数
    }
}
if(optind==argc) {    // optind: 命令行参数中未读取的下一个元素下标
    fprintf(stderr,"webbench: Missing URL!\n");
    usage();
    return 2;
}
//并发数和请求时间不能为 0
if(clients==0) clients=1;
if(benchmark==0) benchmark=60;
fprintf(stderr,"Webbench - Simple Web Benchmark "PROGRAM_VERSION"\n"
"Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.\n"
);
build_request(argv[optind]); //调用 build_request 函数: 封装 HTTP 请求信息
//输出信息
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;
}
printf(" %s",argv[optind]);
switch(http10)
{
    case 0: printf(" (using HTTP/0.9)");break;

```

```

        case 2: printf(" (using HTTP/1.1)");break;
    }
    printf("\n");
    if(clients==1) printf("1 client");
    else
        printf("%d clients",clients);           //并发数
    printf(", running %d sec", benchtime);       //测试时间
    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");
    return bench();           //开始测试
}

```

3.2 bench()函数

3.2.1 函数功能

bench()函数用于创建管道并派生子进程，并调用 HTTP 请求测试函数开始进行测试，最终将所有测试结果输出到屏幕。

bench()函数首先调用 Socket ()函数建立一次 socket 连接，测试 URL 地址是否可以正常访问。然后调用 pipe ()函数创建管道，用于父子进程间的数据传输。接着根据所设定的并发数派生出相应数目的子进程，每个子进程调用 benchcore()函数进行 HTTP 请求测试，并将测试结果写入前面所建立的管道。最终，父进程从管道读取数据，直到读取完所有子进程数据，在屏幕上输出最终测试结果。

3.2.2 bench()函数源码及注释：

```

static int bench(void)
{
    int i,j,k;
    pid_t pid=0;
    FILE *f;
    //调用 Socket 函数创建 socket 连接，测试地址是否可以正常访问
    i=Socket(proxyhost==NULL?host:proxyhost,proxyport);
    if(i<0) {           //错误信息
        fprintf(stderr,"\nConnect to server failed. Aborting benchmark.\n");
    }
}

```



```

        return 1;
    }
    close(i);
    if(pipe(mypipe))    //创建管道：用于子进程向父进程回报数据
    {
        perror("pipe failed.");
        return 3;
    }
    for(i=0;i<clients;i++)
    {
        pid=fork();    //根据 clients 数目派生子进程
        if(pid <= (pid_t) 0)
        {
            /* child process or error*/
            sleep(1); /* make childs faster */
            break;    //注意：子进程要跳出循环，防止子进程派生子子进程
        }
    }
    if( pid< (pid_t) 0)    //错误信息
    {
        fprintf(stderr,"problems forking worker no. %d\n",i);
        perror("fork failed.");
        return 3;
    }
    if(pid== (pid_t) 0)    //子进程
    {
        /* I am a child */
        if(proxyhost==NULL)    //判断是否使用代理服务器
            benchcore(host,proxyport,request);    //测试 HTTP 请求
        else
            benchcore(proxyhost,proxyport,request);
        f=fdopen(mypipe[1],"w");    //子进程打开管道写
        if(f==NULL)    //错误信息
        {
            perror("open pipe for writing failed.");
            return 3;
        }
        fprintf(f,"%d %d %d\n",speed,failed,bytes);    //子进程往管道写数据
        fclose(f);
        return 0;
    }
    else{    //父进程

```

```

f=fdopen(mypipe[0],"r");           //父进程打开管道读
if(f==NULL)    //错误信息
{
    perror("open pipe for reading failed.");
    return 3;
}
setvbuf(f,NULL,_IONBF,0);
speed=0;           //传输速度
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);    //父进程从管道读数据
    if(pid<2)
    {
        fprintf(stderr,"Some of our childrens died.\n");
        break;
    }
    speed+=i;    //传输速度计数
    failed+=j;    //失败请求数计数
    bytes+=k;    //传输字节数计数
    //判断是否读取完所有子进程数据，读取完则退出循环
    if(--clients==0) break;
}
fclose(f);
//在屏幕上输出测试结果
printf("\nSpeed=%d pages/min, %d bytes/sec.\nRequests: %d succeed, %d
failed.\n",
    (int)((speed+failed)/(benchtime/60.0f)),
    (int)(bytes/(float)benchtime),
    speed,
    failed);
}
return i;
}

```

3.3 benchcore()函数

3.3.1 函数功能

benchcore()函数用于测试 HTTP 请求。该函数加载了信号处理函数

alarm_handler ()，通过 alarm()函数计时，计时完毕将产生 SIGALRM 信号，调用信号处理函数 alarm_handler ()，将变量 timerexpired 置为 1。最终，通过判断 timerexpired 的值来决定是否退出测试函数，从而结束测试工作。

3.3.2 benchcore()函数源码及注释：

```
void benchcore(const char *host,const int port,const char *req)
{
    int rlen;
    char buf[1500];                //存放服务器响应请求所返回数据
    int s,i;
    struct sigaction sa;
    sa.sa_handler=alarm_handler;   //信号处理函数
    sa.sa_flags=0;
    //计时结束将产生 SIGALRM 信号，调用信号处理函数
    if(sigaction(SIGALRM,&sa,NULL))
        exit(3);

    alarm(benchmark);              //设置闹钟函数：计时开始
    rlen=strlen(req);
    nexttry:while(1)               //带 go-to 语句的 while 循环
    {
        //计时结束将产生信号，信号处理函数将 timerexpired 置 1，结束测试
        if(timerexpired)
        {
            if(failed>0)
            {
                failed--;
            }
            return;
        }
        s=Socket(host,port);        //建立 socket 连接，获取 socket 描述符
        if(s<0) { failed++;continue;} //连接失败，failed 加 1
        if(rlen!=write(s,req,rlen)) { failed++;close(s);continue;} //发出请求
        if(http10==0)               //针对 HTTP0.9 进行特殊处理
        if(shutdown(s,1)) { failed++;close(s);continue;}
        if(force==0)                //是否等待服务器应答，-f 选项为不等待
        {
            /* read all available data from socket */
            while(1)
            {
```

```

        if(timerexpired) break;    //判断是否超时
        i=read(s,buf,1500);        //读取服务器应答数据，存放于数组 buf
        if(i<0)                    //读取数据失败
        {
            failed++;
            close(s);
            goto nexttry;          //重新开始循环
        }
        else                      //读取数据成功
            if(i==0) break;
            else
                bytes+=i;          //读取字节数增加
    }
}
if(close(s)) { failed++;continue;}
speed++;                        //HTTP 测试成功，speed 加 1
}
}

```

3.4 build_request() 函数

3.4.1 函数功能

build_request() 函数用于封装 HTTP 请求信息。该函数根据命令行输入的参数以及 URL 地址来确定 HTTP 请求内容，封装成一条完整的 HTTP 请求信息。一个典型的 HTTP 请求信息如下：

```

GET / HTTP/1.1
User-Agent: WebBench 1.5
Host: www.baidu.com
Pragma: no-cache
Connection: close

```

值得注意的是，在该函数中调用了大量的字符串处理函数。比如，strcpy, strstr, strncasecmp, strlen, strchr, strncpy, strcat 等等，最终将一条完整的 HTTP 请求信息保存在数组 request 中。

3.4.2 build_request() 函数源码及注释:

```
void build_request(const char *url)
{
    char tmp[10];
    int i;
    //请求地址和请求连接初始化清零
    bzero(host,MAXHOSTNAMELEN);
    bzero(request,REQUEST_SIZE);
    //设置 HTTP 协议版本
    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;
    //设置 HTTP 请求方法
    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," ");
    //判断 URL 地址是否合法
    if(NULL==strstr(url,"://")) //判断 url 是否包含 “: //”
    {
        //输出错误信息
        fprintf(stderr, "\n%s: is not a valid URL.\n",url);
        exit(2);
    }
    if(strlen(url)>1500) //判断 url 是否太长
    {
        fprintf(stderr,"URL is too long.\n");
        exit(2);
    }
    if(proxyhost==NULL) //判断代理服务器是否为空
        if (0!=strncasecmp("http://",url,7)) //判断前 7 个字符串是否为“http://”
        {
            fprintf(stderr, "\nOnly HTTP protocol is directly supported, set --proxy for others.\n");
            exit(2);
        }
    }
```

```

    }
    i=strstr(url,"://")-url+3;           //指向 http://后第一个位置，即主机名
    if(strchr(url+i,'/')==NULL) {       //判断 URL 是否以 “/” 结尾
        fprintf(stderr,"\nInvalid URL syntax - hostname don't ends with '\/.n");
        exit(2);
    }
    if(proxyhost==NULL)
    {
        //获取端口号
        if(index(url+i,':')!=NULL && index(url+i,':')<index(url+i,'/'))
        {
            strncpy(host,url+i,strchr(url+i,':')-url-i);    //取出主机地址
            bzero(tmp,10);
            //取出端口号
            strncpy(tmp,index(url+i,':')+1,strchr(url+i,'/')->index(url+i,':')-1);
            proxyport=atoi(tmp);                          //类型转换
            if(proxyport==0) proxyport=80;
        } else
        {
            strncpy(host,url+i,strcspn(url+i,"/"));
        }
        strcat(request+strlen(request),url+i+strcspn(url+i,"/"));
    } else
    {
        strcat(request,url);
    }
    //开始封装 HTTP 请求内容
    if(http10==1)                                     //版本号
        strcat(request," HTTP/1.0");
    else if (http10==2)
        strcat(request," HTTP/1.1");
    strcat(request,"\r\n");                          //\r\n: 回车换行
    if(http10>0)                                     // User-Agent:
        strcat(request,"User-Agent: WebBench \"PROGRAM_VERSION\"\r\n");
    if(proxyhost==NULL && http10>0)                  //Host:
    {
        strcat(request,"Host: ");
        strcat(request,host);                        //URL 地址
        strcat(request,"\r\n");
    }
    if(force_reload && proxyhost!=NULL)             //Pragma:
    {

```

```

        strcat(request,"Pragma: no-cache\r\n");
    }
    if(http10>1)                                // Connection:
        strcat(request,"Connection: close\r\n");
    /* add empty line at end */
    if(http10>0) strcat(request,"\r\n");
}

```

3.5 alarm_handler()函数

3.5.1 函数功能

alarm_handler()函数是一个信号处理函数。子进程在进行 HTTP 请求测试时，规定了相应的测试时间。当计时结束后将产生一个特定的信号，从而调用信号处理函数进行处理。信号处理函数 alarm_handler()的功能是将变量 timerexpired 的值置为 1，以便程序判断计时是否结束，从而结束测试。

3.5.2 alarm_handler()函数源码及注释：

```

static void alarm_handler(int signal)
{
    timerexpired=1;    //timerexpired 置 1
}

```

3.6 Socket()函数

3.6.1 函数功能

Socket()函数用于创建 socket 连接并返回 socket 描述符。

3.6.2 Socket()函数源码及注释：

```

int Socket(const char *host, int clientPort)
{
    int sock;
    unsigned long inaddr;
    struct sockaddr_in ad;
    struct hostent *hp;
    memset(&ad, 0, sizeof(ad));    //初始化地址
}

```

```

ad.sin_family = AF_INET;
inaddr = inet_addr(host);           //将点分十进制的 IP 地址转为无符号长整形
if (inaddr != INADDR_NONE)         //判断是否为合法 IP 地址
    memcpy(&ad.sin_addr, &inaddr, sizeof(inaddr));
else                                //如果是域名
{
    hp = gethostbyname(host);       //通过域名获取 IP 地址
    if (hp == NULL)
        return -1;
    memcpy(&ad.sin_addr, hp->h_addr, hp->h_length);
}
ad.sin_port = htons(clientPort);
sock = socket(AF_INET, SOCK_STREAM, 0); //获取 socket 描述符
if (sock < 0)
    return sock;
if (connect(sock, (struct sockaddr *)&ad, sizeof(ad)) < 0) //建立连接
    return -1;
return sock;
}

```

3.7 usage()函数

3.7.1 函数功能

usage()函数用于显示使用说明。当命令行参数输入有误时会调用此函数，在屏幕显示程序的使用说明。

3.7.2 usage()函数源码及注释：

```

static void usage(void)
{
    //显示使用说明
    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"
);
};

```

3.8 库函数

在 Webbench 程序中，调用了不少库函数。这些函数平时比较少见，但实现起来简单高效。通过对库函数的调用，可以大大地提高程序开发的效率。因此，掌握库函数的使用方法极为重要。下面着重挑选部分源程序中涉及到的库函数进行介绍。

3.8.1 getopt_long ()

(1) 函数原型: `int getopt_long(int argc, char * const argv[], const char *optstring, const struct option *longopts, int *longindex);`

(2) 函数说明:

`getopt` 用来解析命令行选项参数, `getopt_long` 支持长选项的命令行解析。

参数说明如下:

`argc`、`argv[]`: 通常直接从 `main()` 的两个参数传递而来;

`optstring`: 由该命令要处理的各个选项组成的字符串。选项后面带有冒号时, 表示该选项是一个带参数的选项;

`longopts`: 构造体 `struct option` 数组;

`longindex`: 输出参数, 函数 `getopt_long ()` 返回时, 该参数的值是 `struct option` 数组的索引。

3.8.2 pipe ()

(1) 函数原型: `int pipe(int fd[2]);`

(2) 函数说明:

`pipe()`会建立管道, 并将文件描述符由参数 `filedes` 数组返回。

`filedes[0]`为管道里的读取端,

`filedes[1]`则为管道的写入端。

3.8.3 `fork()`

(1) 函数原型: `pid_t fork(void);`

(2) 函数说明:

一个现有进程可以调用 `fork` 函数创建一个新进程。由 `fork` 创建的新进程被称为子进程。`fork` 函数被调用一次但返回两次。两次返回的唯一区别是子进程中返回 0 值而父进程中返回子进程 ID。

子进程是父进程的副本, 它将获得父进程数据空间、堆、栈等资源的副本。注意, 子进程持有的是上述存储空间的“副本”, 这意味着父子进程间不共享这些存储空间。

UNIX 将复制父进程的地址空间内容给子进程, 因此, 子进程有了独立的地址空间。在不同的 UNIX (Like)系统下, 我们无法确定 `fork` 之后是子进程先运行还是父进程先运行, 这依赖于系统的实现。所以在移植代码的时候我们不应该对此作出任何的假设。

3.8.4 `fdopen()`

(1) 函数原型: `FILE * fdopen(int fildes,const char * mode);`

(2) 函数说明:

`fdopen` 取一个现存的文件描述符 (我们可能从 `open`, `dup`, `dup2`, `fcntl`, `pipe`, `socket`, `socketpair` 或 `accept` 函数得到此文件描述符), 并使一个标准的 I/O 流与该描述符相结合。

此函数常用于由创建管道和网络通信通道函数获得的描述符。因为这些特殊类型的文件不能用标准 I/O `fopen` 函数打开, 首先必须先调用设备专用函数以获得一个文件描述符, 然后用 `fdopen` 使一个标准 I/O 流与该描述符相结合。

3.8.5 fclose ()

(1) 函数原型: `int fclose(FILE *fp);`

(2) 函数说明:

`fclose()`用来关闭先前 `fopen()`打开的文件. 此动作会让缓冲区内的数据写入文件中, 并释放系统所提供的文件资源

3.8.6 setvbuf ()

(1) 函数原型: `int setvbuf(FILE *stream, char *buf, int type, unsigned size);`

(2) 函数说明:

函数 `setvbuf()`用来设定文件流的缓冲区。实际意义在于用户打开一个文件后, 可以建立自己的文件缓冲区, 而不必使用 `fopen()`函数打开文件时设定的默认缓冲区。这样就可以让用户自己来控制缓冲区, 包括改变缓冲区大小、定时刷新缓冲区、改变缓冲区类型、删除流中默认的缓冲区、为不带缓冲区的流开辟缓冲区等。

3.8.7 sigaction ()

(1) 函数原型: `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`

(2) 函数说明:

`sigaction()`会依参数 `signum` 指定的信号编号来设置该信号的处理函数。参数 `signum` 可以指定 `SIGKILL` 和 `SIGSTOP` 以外的所有信号。

3.8.8 inet_addr ()

(1) 函数原型: `in_addr_t inet_addr(const char *cp);`

(2) 函数说明:

若无错误发生, `inet_addr()`返回一个无符号长整型数, 其中以适当字节顺序存放 Internet 地址。如果传入的字符串不是一个合法的 Internet 地址, 那么 `inet_addr()`返回 `INADDR_NONE`。

3.8.9 gethostbyname ()

(1) 函数原型: `struct hostent *gethostbyname(const char *name);`

(2) 函数说明:

返回对应于给定主机名的包含主机名字和地址信息的 `hostent` 结构指针。如果函数调用失败，将返回 `NULL`。

3.8.10 `htons()`

(1) 函数原型: `u_short PASCAL FAR htons(u_short hostshort);`

(2) 函数说明:

将一个无符号短整型的主机数值转换为网络字节顺序。即是将整型变量从主机字节顺序转变成网络字节顺序，整数在地址空间存储方式变为：高位字节存放在内存的低地址处。

网络字节顺序是 `TCP/IP` 中规定好的一种数据表示格式，它与具体的 `CPU` 类型、操作系统等无关，从而可以保证数据在不同主机之间传输时能够被正确解释，网络字节顺序采用 `big-endian` 排序方式。

4 软件演示

下面举例演示对 www.baidu.com 进行压力测试。

(1) 下载安装 Webbench 程序

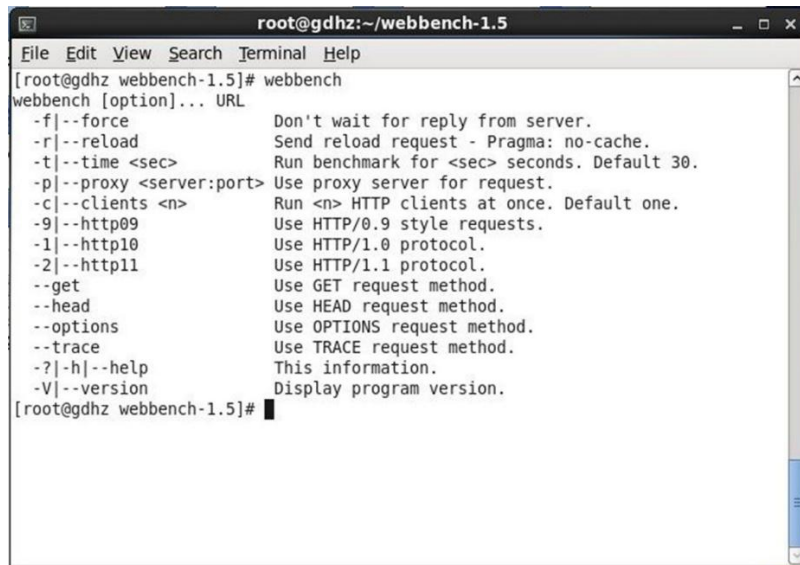
Webbench 下载地址: <http://home.tiscali.cz/~cz210552/webbench.html>

下载完毕，解压安装，过程如图 4 所示:

图 4 Webbench 安装

(2) 使用说明

输入 `webbench`，程序提示使用说明。



```
root@gdhz:~/webbench-1.5
File Edit View Search Terminal Help
[root@gdhz webbench-1.5]# webbench
webbench [option]... URL
  -f|--force           Don't wait for reply from server.
  -r|--reload          Send reload request - Pragma: no-cache.
  -t|--time <sec>     Run benchmark for <sec> seconds. Default 30.
  -p|--proxy <server:port> Use proxy server for request.
  -c|--clients <n>    Run <n> HTTP clients at once. Default one.
  -9|--http09          Use HTTP/0.9 style requests.
  -1|--http10          Use HTTP/1.0 protocol.
  -2|--http11          Use HTTP/1.1 protocol.
  --get               Use GET request method.
  --head              Use HEAD request method.
  --options            Use OPTIONS request method.
  --trace             Use TRACE request method.
  -?|-h|--help        This information.
  -V|--version        Display program version.
[root@gdhz webbench-1.5]#
```

图 5 Webbench 使用说明

(3) 压力测试

测试 URL 地址: www.baidu.com

并发数: 100

测试时间: 30s



```
root@gdhz:~/webbench-1.5
File Edit View Search Terminal Help
[root@gdhz webbench-1.5]# webbench -c 100 -t 30 http://www.baidu.com/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://www.baidu.com/
100 clients, running 30 sec.
```

图 6 Webbench 对 www.baidu.com 进行压力测试

(4) 测试结果

测试结果表明：网站速度为 208 页/分钟，494632 字节/秒；请求成功个数 104 个，请求失败个数 0 个。

```
root@gdhz:~/webbench-1.5
File Edit View Search Terminal Help
[root@gdhz webbench-1.5]# webbench -c 100 -t 30 http://www.baidu.com/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://www.baidu.com/
100 clients, running 30 sec.

Speed=208 pages/min, 494632 bytes/sec.
Requests: 104 succeed, 0 failed.
[root@gdhz webbench-1.5]#
```

图 7 测试结果

5 结语

在运维工作中，压力测试是一项很重要的工作。比如在一个网站上线之前，能承受多大访问量以及在访问量特别大的情况下性能怎样，这些数据指标好坏将会直接影响用户体验。但是，在压力测试中存在一个共性，那就是压力测试的结果与实际负载结果不会完全相同，就算压力测试工作做的再好，也不能保证 100% 和线上性能指标相同。面对这些问题，我们只能尽量去模拟。所以，压力测试非常有必要，有了这些数据，才能对维护的平台做到心中有数。

Webbench 程序是在 LINUX 环境下使用的网站压力测试工具。它可以通过若个子进程模拟多个客户端同时访问某个 URL 地址，从而测试网站性能。Webbench 最多可以模拟 3 万个并发连接去测试网站的负载能力。在本文中，通过对 Webbench 程序源码的深入解析，使我进一步学习了 LINUX 下 C 编程的思想和技巧。