第 1 章 Web 服务器开发基础

1.1 Web 服务器简介

Web 服务器是通过 HTTP 协议将客户端请求的文件发送到客户端的软件系统。其主要功能是读取 Web 网页，并将 Web 网页发送到客户端的浏览器中。Web 服务器主要包括两种类型：静态 Web 服务器和动态 Web 服务器。静态 Web 服务器不负责代码脚本的执行，只是将 Web 文件发送到客户端，例如 Apache、Nginx 和 IIS 等 Web 服务器；动态 Web 服务器，需运行客户请求的代码脚本，并将运行结果发送到客户端。动态 Web 服务器一般也被称为应用服务器。例如，Tomcat 应用服务器负责 JSP 代码脚本的解析和运行；Zend 应用服务器服务 PHP 代码脚本的解析和运行。在目前企业应用架构中，经常将静态服务器与动态服务器混合，以支持灵活的企业应用。例如，Apache、Nginx 和 IIS 等服务器可以通过配置相关的模块，与应用服务结合来达到即能完成静态页面传输，也能完成动态页面解析运行的功能。

由于本实验教程主要以实现静态 Web 服务器为目标，在文章以后的内容中出现的“Web 服务器”特指“静态 Web 服务器”。Web 服务器主要包括 Web 文件存储，客户请求路径解析，Web 文件读取和 Web 文件传输四个部分。用户浏览器与 Web 服务器具体的交流流程如下图 1-1 所示。

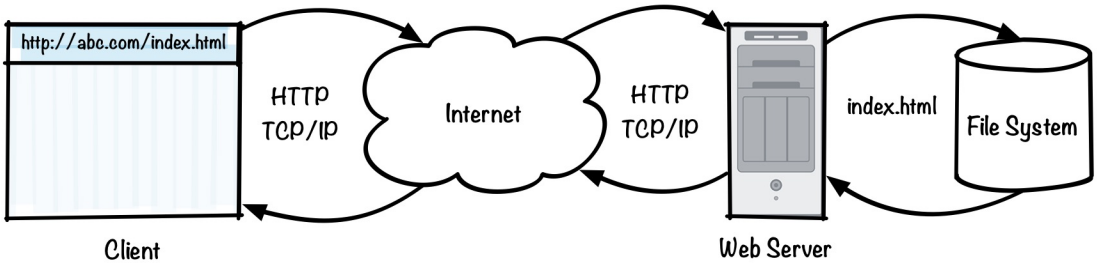

图 1-1 用户请求 Web 网页过程图

用户在浏览器中输入URL链接地址 “http://abc.com/index.html” ，浏览器将根据此URL封装成 HTTP 请求消息，并通过 TCP/IP 协议将此请求消息发送到指定地址的 Web 服务器中。Web 服务器接收到此 HTTP 请求消息后，首先进行解析并从中获得到用户需要 index.html 文件的信息，然后在文件系统中查找此文件并读取此文件内容，然后将此文件内容封装成 HTTP 消息返回给用户浏览器。浏览器在接收到返回消息后，对里面的 html 内容进行解析，并展示到浏览器界面中。

1.2 TCP 与 HTTP 协议

1.2.1 TCP/IP 协议族简介

OSI 将计算机网络体系分为七层：物理层、数据链路层、网络层、传输层、会话层、表示层和应用层[1]。

[1] OSI Model---https://zh.wikipedia.org/wiki/OSI 模型

在 Internet 网络中使用的是五层网络模型：物理层、网络接口层、网络层、传输层和应用层。物理层对应网络的基本硬件；网络接口层中的协议定义了网络中传输的帧格式；网络层中协议定义了信息包的格式以及这些信息包在网络中的转发机制；传输层中协议用于网络中两个终端之间的信息传输；应用层中协议指定了具体应用中的信息格式。

TCP/IP 协议族是支撑 Internet 网络的主要协议组，其中应用层包括 DNS、FTP、HTTP、IMAP、LDAP、RTP、SSH、Telnet、TLS/SSL 等协议；传输层包括 TCP、UDP、RSVP、SCTP 等协议；网络层包括 IP(IPv4,IPv6)、ICMP、ECN、IGMP 等协议；网络接口层包括 ARP、PPP、Ethernet、DSL、ISDN、FDDI 等协议。具体包含协议情况详见 Wikipedia 中的 TCP/IP 词条。

TCP/IP 协议族以传输层中的 TCP(Transmission Control Protocol)和互联网层中的 IP(Internet Protocol)来命名，足以说明这两个协议的重要性。其中，TCP 是一种面向连接的、可靠的、基于字节流的传输协议；IP 定义了寻址方法和数据包的封装结构。

1.2.2 HTTP 协议

HTTP(Hypertext Transfer Protocol)协议，是应用层协议，主要负责超文本的交互与传输。而超文本是结构化的文档，其中使用超链接来关联不同站点上的文件。例如，在网站 A 上网页 w1 上可以通过点击一个超链接，来打开另一个网页 w2，这个网页 w2 可以来自于网站 A 也可以来自于其它网站。而 HTTP 协议能够保证这些网页之间的无缝链接，把点击链接的相应网页或其它文件发送到用户的浏览器中。

HTTP 协议是请求响应式协议，即客户端向服务器发出请求消息，服务器根据请求消息方法和自身状态来做成响应，并把响应消息发送给客户端。HTTP 协议消息使用 ASCII 编码，其 1.1 版本具体格式按增强巴斯特范式(Augmented BNF)定义为如下[2]。

```
HTTP-message    = Request | Response
Request         = Request-Line
                  *(( general-header
                   | request-header
                   | entity-header ) CRLF)
                  CRLF
                  [ message-body ]

Response        = Status-Line
                 *(( general-header
                   | response-header
                   | entity-header ) CRLF)
                 CRLF
                 [ message-body ]
Request-Line    = Method SP Request-URI SP HTTP-Version CRLF
Method          =      "OPTIONS"
                     | "GET"
                     | "HEAD"
                     | "POST"
                     | "PUT"
                     | "DELETE"
                     | "TRACE"
                     | "CONNECT"
                     | extension-method
extension-method = token
Request-URI     = "*" | absoluteURI | abs_path | authority

general-header =      Cache-Control
                    | Connection
```

[2] https://www.ietf.org/rfc/rfc2616.txt

```
                              | Date
                              | Pragma
                              | Trailer
                              | Transfer-Encoding
                              | Upgrade
                              | Via
                              | Warning

request-header =                Accept
                              | Accept-Charset
                              | Accept-Encoding
                              | Accept-Language
                              | Authorization
                              | Expect
                              | From
                              | Host
                              | If-Match
entity-header  =                Allow
                              | Content-Encoding
                              | Content-Language
                              | Content-Length
                              | Content-Location
                              | Content-MD5
                              | Content-Range
                              | Content-Type
                              | Expires
                              | Last-Modified
                              | extension-header

Status-Line     = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

Status-Code     =       "100"       ; Continue
                      | "101"       ; Switching Protocols
                      | "200"   ; OK
                      | "201"   ; Created
                      | "202"   ; Accepted
                      | "203"   ; Non-Authoritative Information
                      | "204"   ; No Content
                      | "205"   ; Reset Content
                      | "206"   ; Partial Content
                      | "300"   ; Multiple Choices
                       | "301"   ; Moved Permanently
                      | "302"   ; Found
                      | "303"   ; See Other
                      | "304"   ; Not Modified
                      | "305"   ; Use Proxy
                      | "307"   ; Temporary Redirect
                      | "400"   ; Bad Request
                      | "401"   ; Unauthorized
                      | "402"   ; Payment Required
                      | "403"   ; Forbidden
                      | "404"   ; Not Found
                      | "405"   ; Method Not Allowed
                      | "406"   ; Not Acceptable
                      | "407"   ; Proxy Authentication Required
                      | "408"   ; Request Time-out
                      | "409"   ; Conflict
                      | "410"   ; Gone
                      | "411"   ; Length Required
                      | "412"   ; Precondition Failed
                     | "413"   ; Request Entity Too Large
                     | "414"   ; Request-URI Too Large
                     | "415"   ; Unsupported Media Type
```

```
                        | "416"   ; Requested range not satisfiable
                        | "417"   ; Expectation Failed
                        | "500"   ; Internal Server Error
                        | "501"   ; Not Implemented
                        | "502"   ; Bad Gateway
                        | "503"   ; Service Unavailable
                        | "504"   ; Gateway Time-out
                        | "505"   ; HTTP Version not supported
                        | extension-code
response-header = Accept-Ranges
                | Age
                | ETag
                | Location
                | Proxy-Authenticate
```

根据上面的协议格式描述，一个 Request-Line 可以表示“GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1”。在 RFC2616 协议中，HTTP 为区分客户端发出的请求消息方法，在 HTTP/1.1 中将请求消息分为 GET、POST、HEAD、OPTIONS、PUT、DELETE、TRACE 和 CONNECT 八种方法。

- GET 方法表示要请求获取特定的资源，比如 html 网页、jpg 图像文件等。GET 类型请求消息仅表示获取数据，并不影响数据（增加、删除、修改等）。
- HEAD 方法与 GET 获得的响应一致，但要求响应消息中只包含 head，不包含 body。这个方法在获取元信息时非常有效。例如要获取一个文件的大小、日期等信息，并不需要服务器的响应信息中包含这个文件，而只是将这些元信息封装到响应消息的 head 中即可。
- POST 方法请求服务器接收封装在请求消息中的数据实体，并将其作为新的附属资源粘贴到指定 URI 的 Web 资源中。封装在 POST 请求消息中的数据可以是邮件列表、Web 页面中需要提交的数据、公告板中的一条消息等内容。
- PUT 方法请求服务器将请求消息中的数据实体存储指定 URI 中。如果 URI 指向已经存在的资源，则将此数据实体替代已经存在的资源；如果 URI 指向的资源不存在，则将此数据实体表示为此 URI 指向的资源。
- DELETE 方法请求删除指定的资源
- TRACE 方法请求中间服务器将自身消息及对请求消息的改变添加到请求消息中，从而使得客户端能够追踪请求消息的路由过程及消息变化情况。
- OPTIONS 方法请求服务器返回其能够支持的 HTTP 请求方法。
- CONNECT 方法将请求连接转换到透明的 TCP/IP 通道，这样做的目的是为了便于加密的 HTTPS 通过非加密的 HTTP 代理。

有关以上方法的详细说明，请见 RFC7231 和 RFC5789. 本书实验将主要关注 GET 和 HEAD 类型的请求消息处理。

请求消息的格式由下面四部分组成。

- 请求行：其用来表明请求消息类型和请求资源的 URI。例如，GET /web/index.html 表示请求获取服务器管理的虚拟路径下 web 目录中的 index.html 网页。
- 请求头域列表：在列表内部，每个请求头域描述请求消息中的一个参数及其值，其中参数表示此请求头域的名称。具体值格式为 parameter：value。例如，Accept: text/plain 是 Accept 头域，其值 text/plain 表示响应消息中的内容格式类型为 text/plain。
- 一个空行(/r/n)

- 消息体（可选）

在请求行和请求头域每行必须以符合“<CR><LF>”结尾。在空行中只有符号“<CR><LF>”，不能出现空格。在 HTTP/1.1 协议中，除了 Host 头域外，其它所有请求头域都是可选的。

与请求消息相对应的是服务器给客户端的响应消息。响应消息由下面四部分组成。

- 响应状态行：其内部包含状态码和原因内容。例如 “HTTP/1.1 200 OK”表示客户端请求成功。
- 响应头域：给出响应参数信息。例如 “Content-Type:text/html”表示响应消息体的数据格式为“text/html”。
- 一个空行（/r/n）。
- 消息体：存放响应消息具体数据。

例如，一个请求消息实例如下所示。

```
GET /index.html HTTP/1.1
Host: www.example.com
```

服务器给出的响应消息为：

```
HTTP/1.1 200 OK
Date: Mon, 08 May 2017 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Sun, 08 Jan 2017 12:21:50 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
<head>
  <title>An Example Page</title>
</head>
<body>
  Hello World, this is a very simple HTML document.
</body>
</Html>
```

1. 3 Socket 编程

Socket 是操作系统中实现 TCP/IP 等通信协议的 API 接口。通过调用 Socket 能够实现多台计算机之间的消息传递。Socket 分为客户端和服务端两种状态。Socket 服务端状态主要用于服务器的开发。在单进程单线程的 TCP 服务器模型中，socket 接口调用顺序和状态变化如下面代码“Server Code”所示。首先初始化自身，并绑定一个侦听端口；然后设置为侦听状态，并阻塞当前运行线程；一旦有客户端的连接请求，将与客户端建立一个新的连接通道，并在这个通道中通过读、写接口与客户端进行通信；如果处理完与客户端的通信，就可以将这个通道关闭；然后继续阻塞当前线程，直到有新的客户端进行连接请求。具体过程如下所示。

在 linux 系统中，涉及到 TCP/IP 传输的主要有以下接口。

- socket()函数负责初始化一个用于通信的 socket 描述符。其操作语义类似于使用 C 语言中的函数 fopen，其打开一个文件并其返回一个文件描述符，通过此描述符，

能够对文件进行读写。因此通过此函数返回的 socket 描述符，能够进行通信信息的读取和写入。

其具体函数接口如下：

int socket(int protofamily,int type,int protocol)

返回值为此操作此 socket 的描述符。

参数 protofamily 表示所使用网络地址协议，使用 AF_INET, AF_INET6、AF_LOCAL 等数值来分别表示 IPv4、IPv6、文件路径等类型通信地址。例如当使用 AF_INET 作为此函数参数时，则在通信时需要指定 32 位的 IPv4 地址和端口号，如 127.0.0.1:8080。

参数 type 指定 socket 类型，常用的类型有 SOCK_STREAM、SOCK_DGRAM 和 SOCK_RAW。SOCK_STREAM 是面向连接的可靠的双向数据流通信，发送的数据按顺序到达，一般应用在 TCP 协议的消息传递；SOCK_DGRAM 是面向无连接的非可靠数据通信，一般应用在 UDP 协议；SOCK_RAW 是指直接向网络硬件直接发送或接收原始的数据报文，socket 通过此项设置给予上层调用程序，自己设计数据报文格式和解析报文的能力。

参数 protocol 表示 socket 使用传输协议，其数值有 IPPROTO_TCP、IPPROTO_UDP、IPPROTO_STCP、IPPROTO_TIPC 等，分别应用 TCP 传输协议、UDP 传输协议、STCP 传输协议、TIPC 传输协议。

例如： int clientsock_fd=socket(AF_INET,SOCK_STREAM,0)

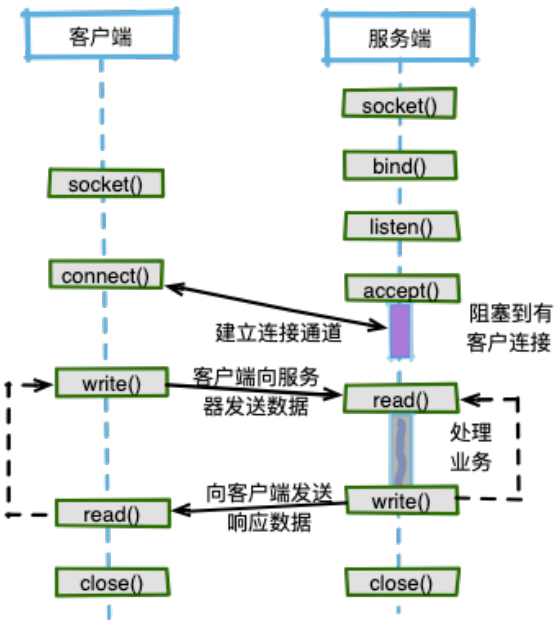

图 1-2 socket 客户端服务器通信流程

- bind 函数负责将 socket 描述符与指定地址绑定。bind 函数是服务端调用的函数，用来绑定具体的侦听端口号。因为客户端会自动创建连接和端口号，因此在客户端并不需要 bind 函数。其具体函数格式如下：

int bind(int sockfd, const struct sockaddr * addr, socklen_t addrlen)

其中参数 sockfd 为 socket 描述符（由 socket 函数产生）；addr 为地址指针，指向要为 sockfd 绑定的地址。地址数据结构要与创建 socket 描述符时参数 protofamily 一致。例如，如果 protofamily 参数值为 AF_INET，则 addr 指向一个 IPv4 的地址结构 sockadd_in；如果 protofamily 参数值为 AF_INET6，则 addr 指向一个 IPv6 地址结构 sockaddr_in6;如果 protofamily 参数值为 AF_LOCAL，则 addr 指向一个路径结构 sockadr_un。 参数 addrlen 为地址长度。

- listen()函数主要用于服务端，使得服务器能够侦听来自于指定 socket 描述符下的消息。在调用完此函数后，指定的 socket 将变为侦听状态，用于等待用户的连接请求。其具体函数格式如下：

  int listen(int sockfd, int backlog)

  其参数 sockfd 表示 socket 描述符；backlog 表示此 socket 可以接受排队的连接最大个数。
- connect()函数表示为指定的 socket 文件描述符与服务器端的地址建立连接。此函数用于客户端，使得客户端能够向服务器发起连接。其具体函数格式如下：

  int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

  其参数 sockfd 表示一个已经通过 socket 函数创建的 socket 描述符；addr 为服务端的地址；addrlen 为地址长度。
- accept()函数表示使得处于侦听状态下的 socket 能够接受连接请求，同时此函数会阻塞当前线程，直到有客户端与此 socket 建立连接。其具体函数格式如下：

  int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)

  其参数 sockfd 表示处于侦听状态下的 socket 描述符；addr 用于返回客户端的地址；addrlen 为客户端地址的长度。

  返回值为服务端与客户端新建立的通信通道，即新建立一个 socket 描述符，用于与客户端通信。为什么会新建立一个 socket 描述符呢？这是因为服务器处在侦听状态下的 socket 只负责接收客户端的连接请求，一旦受到请求信号，accept 函数将新建立一个 socket 与客户端 socket 进行通信。这样能够使得服务器能够与多个客户端同时保持通信通道（一个客户端，服务器就有一个 socket 与其对应）。
- read/write 函数，把 socket 描述符当作文件描述符，读写调用与文件操作函数一样，负责在 socket 中读取或写入信息，来实现消息的发生和接收。除此之外，socket 接口函数中还包括 Recv/send 函数、sendto/recvfrom 函数和 sendmsg/recvmsg 函数。
- close 函数，负责关闭指定的 socket，并释放资源。

具体的 socket 中函数与如何支持各个协议，以及各个协议的实现细节，请查询相关书籍材料。在本实验中，将主要关注 TCP/IP 协议基础上的应用层协议 HTTP 的实现。下面例子是 nweb()项目中的代码。其中客户端代码通过 socket 向指定服务器发出了一个 HTTP 协议消息，其目的是请求一个网页 helloworld.html；然后服务器在 socket 端口中，读取 HTTP

协议消息，然后读取客户端指定的网页内容，并将此内容写入与客户端建立的 socket 中；客户端在接收到此网页信息后，将消息打印到控制台，并关闭此 socket。

在 TCP 客户端，socket 接口调用顺序和状态变化，如下面代码所示。其首先初始化自身，向服务器发送请求并建立连接通道；然后通过读、写接口与服务器进行通信；当通信完毕后，关闭这个连接通道。

```
/* Client Code*/
/* The following main code from https://github.com/ankushagarwal/nweb, but they are modified slightly
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* IP address and port number */
#define PORT          8181          // 定义端口号，一般情况下 Web 服务器的端口号为 80
#define IP_ADDRESS "192.168.0.8"      //定义服务端的 IP 地址
/* Request a html file base on HTTP */
char *httprequestMsg = "GET /helloworld.html HTTP/1.0 \r\n\r\n" ;   //定义了 HTTP 请求消息，即请求
                                                                    helloworld.html 文件

#define BUFSIZE 8196

void pexit(char * msg)
{
 perror(msg);
 exit(1);
}
void main()
{
 int i,sockfd;
 char buffer[BUFSIZE];
 static struct sockaddr_in serv_addr;

 printf("client trying to connect to %s and port %d\n",IP_ADDRESS,PORT);
 if((sockfd = socket(AF_INET, SOCK_STREAM,0)) <0) //创建客户端 socket
        pexit("socket() error");

 serv_addr.sin_family = AF_INET;    //设置 Socket 为 IPv4 模式
 serv_addr.sin_addr.s_addr = inet_addr(IP_ADDRESS);// 设置连接服务器的 IP 地址
 serv_addr.sin_port = htons(PORT);   // 设置连接服务器的端口号

 /* 连接指定的服务器*/
 if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <0)
        pexit("connect() error");

 /* 当连接成功后，通过 socket 连接通道向服务器端发送请求消息 */
 printf("Send bytes=%d %s\n",strlen(httprequestMsg), httprequestMsg);
 write(sockfd, httprequestMsg, strlen(httprequestMsg));

 /* 通过 socket 连接通道，读取服务器的响应消息；即 helloworld.html 文件内容；如果是在 Web 浏览
器中，web 浏览器将根据得到的文件内容，进行 Web 页面渲染*/
 while( (i=read(sockfd,buffer,BUFSIZE)) > 0)
        write(1,buffer,i);
 /*close the socket*/
```

```
        close(sockfd);
    }
```

TCP 服务端代码如下面代码所示。其中，数据结构 extensions 主要用来存放 nweb 服务器能够支持的文件类型；logger 函数主要用来给客户端返回一下服务器内部状态的响应消息（响应代码为 403 的 Forbidden 消息和响应代码为 404 的 NOT FOUND 消息），并将相关内容写入日志文件中；web 函数首先从 socket 中读取并解析 HTTP 消息，然后读取指定的文件内容，并合成 HTTP 的响应消息，最后将响应消息写入指定 socket。

在 TCP 服务端主函数流程中，首先对参数 argc 和 argv 进行判断和内容识别，其主要作用是从 argv 参数列表中获得端口号和网页存取路径。例如执行命令 nweb 8181 /home/newdir，在参数列表 argv[1]中保存‘8181’字符串；argv[2]中保存‘/home/newdir’字符串。然后创建侦听 socket，并通过 bind 函数将此 socket 绑定到指定端口(通过参数结构 sockaddr_in 实现，使用 listen 函数设置此 socket 为侦听状态，并最终阻塞在 accept 函数位置。直到有客户端与服务端建立连接，这个函数将返回与客户端建立连接的 socket 描述符。根据此 socket 描述符，使用 web 函数对用户的请求做出响应，并记录信息到日志文件。

```
    /*Server Code*/
    /* webserver.c*/
    /*The following main code from https://github.com/ankushagarwal/nweb*, but they are modified slightly*/

    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>
    #include <string.h>
    #include <fcntl.h>
    #include <signal.h>
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #define VERSION 23
    #define BUFSIZE 8096
    #define ERROR      42
    #define LOG        44
    #define FORBIDDEN 403
    #define NOTFOUND  404

    #ifndef SIGCLD
    #   define SIGCLD SIGCHLD
    #endif

    struct {
      char *ext;
      char *filetype;
    } extensions [] = {
      {"gif", "image/gif" },
      {"jpg", "image/jpg" },
      {"jpeg","image/jpeg"},
      {"png", "image/png" },
      {"ico", "image/ico" },
      {"zip", "image/zip" },
```

```
        {"gz",  "image/gz"  },
        {"tar", "image/tar" },
        {"htm", "text/html" },
        {"html","text/html" },
        {0,0} };

    /* 日志函数，将运行过程中的提示信息记录到 webserver.log 文件中*/
    void logger(int type, char *s1, char *s2, int socket_fd)
    {
      int fd ;
      char logbuffer[BUFSIZE*2];
      /*根据消息类型，将消息放入 logbuffer 缓存，或直接将消息通过 socket 通道返回给客户端*/
      switch (type) {
      case ERROR: (void)sprintf(logbuffer,"ERROR: %s:%s Errno=%d exiting pid=%d",s1, s2, errno,getpid());
        break;
      case FORBIDDEN:
        (void)write(socket_fd, "HTTP/1.1 403 Forbidden\nContent-Length: 185\nConnection:
close\nContent-Type: text/html\n\n<html><head>\n<title>403
Forbidden</title>\n</head><body>\n<h1>Forbidden</h1>\n The requested URL, file type or operation
is not allowed on this simple static file webserver.\n</body></html>\n",271);
        (void)sprintf(logbuffer,"FORBIDDEN: %s:%s",s1, s2);
        break;
      case NOTFOUND:
          (void)write(socket_fd, "HTTP/1.1 404 Not Found\nContent-Length: 136\nConnection:
close\nContent-Type: text/html\n\n<html><head>\n<title>404 Not
Found</title>\n</head><body>\n<h1>Not Found</h1>\nThe requested URL was not found on this
server.\n</body></html>\n",224);
        (void)sprintf(logbuffer,"NOT FOUND: %s:%s",s1, s2);
        break;
      case LOG: (void)sprintf(logbuffer," INFO: %s:%s:%d",s1, s2,socket_fd); break;
      }
      /* 将 logbuffer 缓存中的消息存入 webserver.log 文件*/
      if((fd = open("webserver.log", O_CREAT| O_WRONLY | O_APPEND,0644)) >= 0) {
        (void)write(fd,logbuffer,strlen(logbuffer));
        (void)write(fd,"\n",1);
        (void)close(fd);
      }
    }

    /* 此函数完成了 WebServer 主要功能，它首先解析客户端发送的消息，然后从中获取客户端请求的文
件名，然后根据文件名从本地将此文件读入缓存，并生成相应的 HTTP 响应消息；最后通过服务器与客户
端的 socket 通道向客户端返回 HTTP 响应消息*/

    void web(int fd, int hit)
    {
      int j, file_fd, buflen;
      long i, ret, len;
      char * fstr;
      static char buffer[BUFSIZE+1]; /* 设置静态缓冲区 */

      ret =read(fd,buffer,BUFSIZE);   /* 从连接通道中读取客户端的请求消息 */
      if(ret == 0 || ret == -1) {  //如果读取客户端消息失败，则向客户端发送 HTTP 失败响应信息
        logger(FORBIDDEN,"failed to read browser request","",fd);
      }
      if(ret > 0 && ret < BUFSIZE)  /* 设置有效字符串，即将字符串尾部表示为 0 */
        buffer[ret]=0;
      else buffer[0]=0;
      for(i=0;i<ret;i++)  /* 移除消息字符串中的 “CF” 和 “LF” 字符*/
        if(buffer[i] == '\r' || buffer[i] == '\n')
          buffer[i]='*';
      logger(LOG,"request",buffer,hit);
      /*判断客户端 HTTP 请求消息是否为 GET 类型，如果不是则给出相应的响应消息*/
      if( strncmp(buffer,"GET ",4) && strncmp(buffer,"get ",4) ) {
```

```
            logger(FORBIDDEN,"Only simple GET operation supported",buffer,fd);
          }
          for(i=4;i<BUFSIZE;i++) { /* null terminate after the second space to ignore extra stuff */
            if(buffer[i] == ' ') { /* string is "GET URL " +lots of other stuff */
              buffer[i] = 0;
              break;
            }
          }
          for(j=0;j<i-1;j++)    /* 在消息中检测路径，不允许路径中出现 ".."  */
            if(buffer[j] == '.' && buffer[j+1] == '.') {
              logger(FORBIDDEN,"Parent directory (..) path names not supported",buffer,fd);
            }
          if( !strncmp(&buffer[0],"GET /\0",6) || !strncmp(&buffer[0],"get /\0",6) )
            /* 如果请求消息中没有包含有效的文件名，则使用默认的文件名 index.html */
            (void)strcpy(buffer,"GET /index.html");

          /* 根据预定义在 extensions 中的文件类型，检查请求的文件类型是否本服务器支持 */
          buflen=strlen(buffer);
          fstr = (char *)0;
          for(i=0;extensions[i].ext != 0;i++) {
            len = strlen(extensions[i].ext);
            if( !strncmp(&buffer[buflen-len], extensions[i].ext, len)) {
              fstr =extensions[i].filetype;
              break;
            }
          }
          if(fstr == 0) logger(FORBIDDEN,"file extension type not supported",buffer,fd);

          if(( file_fd = open(&buffer[5],O_RDONLY)) == -1) {  /* 打开指定的文件名*/
            logger(NOTFOUND, "failed to open file",&buffer[5],fd);
          }
          logger(LOG,"SEND",&buffer[5],hit);
          len = (long)lseek(file_fd, (off_t)0, SEEK_END); /* 通过 lseek 获取文件长度*/
           (void)lseek(file_fd, (off_t)0, SEEK_SET); /* 将文件指针移到文件首位置*/
            (void)sprintf(buffer,"HTTP/1.1 200 OK\nServer: nweb/%d.0\nContent-Length: %ld\nConnection:
close\nContent-Type: %s\n\n", VERSION, len, fstr); /* Header + a blank line */
          logger(LOG,"Header",buffer,hit);
          (void)write(fd,buffer,strlen(buffer));

          /* 不停地从文件里读取文件内容，并通过 socket 通道向客户端返回文件内容*/
          while (  (ret = read(file_fd, buffer, BUFSIZE)) > 0 ) {
            (void)write(fd,buffer,ret);
          }
          sleep(1);  /* sleep 的作用是防止消息未发出，已经将此 socket 通道关闭*/
          close(fd);
        }

        int main(int argc, char **argv)
        {
          int i, port, listenfd, socketfd, hit;
          socklen_t length;
          static struct sockaddr_in cli_addr; /* static = initialised to zeros */
          static struct sockaddr_in serv_addr; /* static = initialised to zeros */

           /*解析命令参数*/
          if( argc < 3  || argc > 3 || !strcmp(argv[1], "-?") ) {
            (void)printf("hint: nweb Port-Number Top-Directory\t\tversion %d\n\n"
          "\tnweb is a small and very safe mini web server\n"
          "\tnweb only servers out file/web pages with extensions named below\n"
          "\t and only from the named directory or its sub-directories.\n"
          "\tThere is no fancy features = safe and secure.\n\n"
          "\tExample:webserver 8181 /home/nwebdir &\n\n"
          "\tOnly Supports:", VERSION);
```

```
    for(i=0;extensions[i].ext != 0;i++)
      (void)printf(" %s",extensions[i].ext);

    (void)printf("\n\tNot Supported: URLs including \"..\", Java, Javascript, CGI\n"
  "\tNot Supported: directories / /etc /bin /lib /tmp /usr /dev /sbin \n"
  "\tNo warranty given or implied\n\tNigel Griffiths nag@uk.ibm.com\n"   );
    exit(0);
  }
  if( !strncmp(argv[2],"/"     ,2 ) || !strncmp(argv[2],"/etc", 5 ) ||
      !strncmp(argv[2],"/bin",5 ) || !strncmp(argv[2],"/lib", 5 ) ||
      !strncmp(argv[2],"/tmp",5 ) || !strncmp(argv[2],"/usr", 5 ) ||
      !strncmp(argv[2],"/dev",5 ) || !strncmp(argv[2],"/sbin",6) ){
    (void)printf("ERROR: Bad top directory %s, see nweb -?\n",argv[2]);
    exit(3);
  }
  if(chdir(argv[2]) == -1){
    (void)printf("ERROR: Can't Change to directory %s\n",argv[2]);
    exit(4);
  }

  /* 建立服务端侦听 socket*/
  if((listenfd = socket(AF_INET, SOCK_STREAM,0)) <0)
    logger(ERROR, "system call","socket",0);
  port = atoi(argv[1]);
  if(port < 0 || port >60000)
    logger(ERROR,"Invalid port number (try 1->60000)",argv[1],0);
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  serv_addr.sin_port = htons(port);
  if(bind(listenfd, (struct sockaddr *)&serv_addr,sizeof(serv_addr)) <0)
    logger(ERROR,"system call","bind",0);
  if( listen(listenfd,64) <0)
    logger(ERROR,"system call","listen",0);
  for(hit=1; ;hit++) {
    length = sizeof(cli_addr);
    if((socketfd = accept(listenfd, (struct sockaddr *)&cli_addr, &length)) < 0)
      logger(ERROR,"system call","accept",0);
      web(socketfd,hit); /* never returns */
  }
}
```

1.4 开发环境与测试环境

本书的开发环境，即代码的编写、编译和调试分别使用 vim、gcc/g++和 gdb 来完成。对于 VIM 程序相关命令的使用方法请查阅相关文献。编写源代码除了使用 VIM 外，还可以使用 Emacs、sublime text 和其它文本编辑器工具。另外，本书开发环境还将使用 make 对项目工程中的众多代码文件进行集中编译。

测试环境包含两方面内容，一方面是性能统计、测试工具 vmstat、iostat、iotop、netstat、perf 和 http_load；另一方面是 Web 服务器运行逻辑正确性测试工具 – Web 浏览器，例如 Chrome，Firefox 或 IE。

以上这些工具将被部署在不同的位置，具体如图 1-3 所示。http_load 和 Web 浏览器将被部署到客户端，分别用来测试 Web 服务器的功能和性能。Vim、make、gcc/g++、GDB、vmstat、iostat、iotop、netstat 和 perf 部署在服务器端，分别用来编写、编译、调试服务端程序，并进行 CPU、内存、磁盘、网络性能统计和应用程序的性能分析。

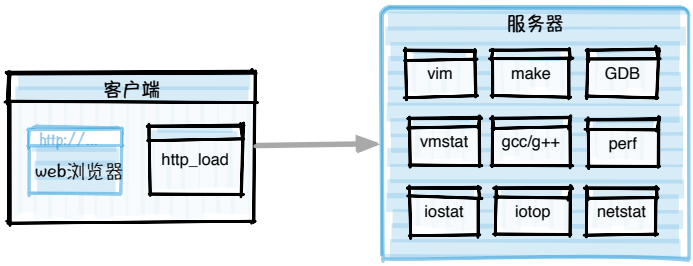

图 1-3 开发和测试环境工具部署视图

1.4.1 GCC

GCC（GNU Complier Collection)是 GNU 项目下的一个编译系统，用以支持各种程序的编译。本文将主要关注与 C 语言相关的常用编译参数选项。GCC 程序在编译程序时包含预处理（Pre-Processing）、编译(Compling)、汇编（Assembling）和链接（Linking）四个阶段。每个阶段对应不同内容信息的输出。

- **预处理阶段**

该阶段执行 C 语言代码中的预处理及宏指令。根据#include 指令，在文件的相应位置插入引入的文件；根据#define 指令，将代码中相应宏替换为定义的字符串。该阶段可以使用 gcc 命令中的 “-E” 参数选用来完成。 例如，

```
gcc -E client.c -o client.i
```

将对 client.c 文件进行预处理，并将预处理结果保存为 client.i 文件。打开 client.i 文件，如下面代码所示，将会发现在源文件中#include 指令的位置插入了相关头文件的内容，并且 main 函数中的宏被替换为具体的定义数值。

```
#577 "/usr/include/sys/socket.h" 3 4
struct msghdr {
 void *msg_name;
 socklen_t msg_namelen;
 struct iovec *msg_iov;
 int msg_iovlen;
 void *msg_control;
 socklen_t msg_controllen;
 int msg_flags;
};
# 577 "/usr/include/sys/socket.h" 3 4
struct cmsghdr {
 socklen_t cmsg_len;
 int cmsg_level;
 int cmsg_type;

};
# 668 "/usr/include/sys/socket.h" 3 4
struct sf_hdtr {
 struct iovec *headers;
 int hdr_cnt;
 struct iovec *trailers;
 int trl_cnt;
};
int accept(int, struct sockaddr * restrict, socklen_t * restrict)
   __asm("_" "accept" );
int bind(int, const struct sockaddr *, socklen_t) __asm("_" "bind" );
int connect(int, const struct sockaddr *, socklen_t) __asm("_" "connect" );
```

```
int getpeername(int, struct sockaddr * restrict, socklen_t * restrict)
  __asm("_" "getpeername" );
int getsockname(int, struct sockaddr * restrict, socklen_t * restrict)
  __asm("_" "getsockname" );
int getsockopt(int, int, int, void * restrict, socklen_t * restrict);
int listen(int, int) __asm("_" "listen" );
ssize_t recv(int, void *, size_t, int) __asm("_" "recv" );
ssize_t recvfrom(int, void *, size_t, int, struct sockaddr * restrict,
  socklen_t * restrict) __asm("_" "recvfrom" );
ssize_t recvmsg(int, struct msghdr *, int) __asm("_" "recvmsg" );
ssize_t send(int, const void *, size_t, int) __asm("_" "send" );
ssize_t sendmsg(int, const struct msghdr *, int) __asm("_" "sendmsg" );
ssize_t sendto(int, const void *, size_t,
  int, const struct sockaddr *, socklen_t) __asm("_" "sendto" );
int setsockopt(int, int, int, const void *, socklen_t);
int shutdown(int, int);
int sockatmark(int) __attribute__((availability(macosx,introduced=10.5)));
int socket(int, int, int);
int socketpair(int, int, int, int *) __asm("_" "socketpair" );
int sendfile(int, int, off_t, off_t *, struct sf_hdtr *, int);
……
……
main()
{
int i,sockfd;
char buffer[8196];
static struct sockaddr_in serv_addr;

 printf("client trying to connect to %s and port %d\n","192.168.0.8",8181);
 if((sockfd = socket(2, 1,0)) <0)
  pexit("socket() error");

 serv_addr.sin_family = 2;
 serv_addr.sin_addr.s_addr = inet_addr("192.168.0.8");
 serv_addr.sin_port  =  ((__uint16_t)(__builtin_constant_p(8181)  ?  ((__uint16_t)((((__uint16_t)(8181)  &
0xff00) >> 8) | (((__uint16_t)(8181) & 0x00ff) << 8))) : _OSSwapInt16(8181)));


 if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <0)
  pexit("connect() error");


 printf("Send bytes=%d %s\n",strlen(httprequestMsg), httprequestMsg);
 write(sockfd, httprequestMsg, strlen(httprequestMsg));


 while( (i=read(sockfd,buffer,8196)) > 0)
  write(1,buffer,i);

        close(sockfd)
}
```

- **编译阶段**

在此阶段，gcc 将检查代码的语法规范，并将 C 语言代码编译成汇编代码。该阶段可以使用 gcc 命令中的 “-S” 参数选用来完成。例如，

gcc –S client.i –o client.s

将生成 client.c 的汇编代码文件 client.s。当然也可以直接使用 “gcc -S client.c -o client.s” 来完成汇编代码的生成，这时将包括预处理和汇编两个阶段。具体 client.s 汇编代码片段如下所示。

```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 12
    .globl      _pexit
    .p2align    4, 0x90
_pexit:                                 ## @pexit
    .cfi_startproc
## BB#0:
    pushq       %rbp
Ltmp0:
    .cfi_def_cfa_offset 16
Ltmp1:
    .cfi_offset %rbp, -16
    movq        %rsp, %rbp
Ltmp2:
    .cfi_def_cfa_register %rbp
    subq $16, %rsp
    movq        %rdi, -8(%rbp)
    movq        -8(%rbp), %rdi
    callq _perror
    movl $1, %edi
    callq _exit
    .cfi_endproc

    .globl      _main
    .p2align    4, 0x90
_main:                                  ## @main
    .cfi_startproc
## BB#0:
    pushq       %rbp
Ltmp3:
    .cfi_def_cfa_offset 16
Ltmp4:
    .cfi_offset %rbp, -16
    movq        %rsp, %rbp
Ltmp5:
….
```

- **汇编阶段**

在此阶段，gcc 将汇编代码转换为二进制目标代码。该阶段使用 gcc 命令中的 “-c” 参数选用来完成。例如，

gcc –c client.s –o client.o

同样也可以使用“gcc -c client.c -o client.o” 连续执行预处理、编译和汇编三个阶段的处理。在使用“gcc -g -o client.o client.c”后，可以使用“objdump -S client.o”命令来查看 C 源代码及其对应汇编代码的混合输出，其显示效果如下面代码片段所示。

```
…
; printf("client trying to connect to %s and port %d\n",IP_ADDRESS,PORT);
100000ccc: b0 00        movb    $0, %al
100000cce: e8 b9 01 00 00  callq 441
100000cd3:bf 02 00 00 00   movl $2, %edi
100000cd8:be 01 00 00 00   movl $1, %esi
100000cdd:31 d2         xorl    %edx, %edx
; if((sockfd = socket(AF_INET, SOCK_STREAM,0)) <0) //create a client socket
100000cdf: 89 85 e4 df ff ff        movl %eax, -8220(%rbp)
```

```
100000ce5: e8 ae 01 00 00   callq 430
100000cea: 89 85 e8 df ff ff        movl %eax, -8216(%rbp)
100000cf0: 83 f8 00     cmpl $0, %eax
100000cf3: 0f 8d 0c 00 00 00        jge    12 <_main+0x65>
100000cf9: 48 8d 3d 85 02 00 00         leaq  645(%rip), %rdi
; pexit("socket() error");
100000d00:      e8 7b ff ff ff      callq  -133 <_pexit>
100000d05:      48 8d 3d 6d 02 00 00        leaq  621(%rip), %rdi
; serv_addr.sin_family = AF_INET;    //Set the socket with IPv4
100000d0c: c6 05 66 03 00 00 02         movb        $2, 870(%rip)
; serv_addr.sin_addr.s_addr = inet_addr(IP_ADDRESS);// set ip address
100000d13:      e8 68 01 00 00   callq 360
100000d18:      48 8d 3d 59 03 00 00        leaq  857(%rip), %rdi
100000d1f: ba 10 00 00 00   movl $16, %edx
100000d24:      89 05 52 03 00 00       movl %eax, 850(%rip)
...
```

- **链接阶段**

在此阶段，gcc 使用链接器 ld，将多个二进制目标文件和库文件链接在一起，以生成可执行格式的文件。例如，

gcc client.o -o client。

同样也可以使用“gcc client.c -o client”将上述四个阶段一起执行，并生成可执行程序。

除了涉及以上编译阶段的参数指令外，还有一些参数选项比较常用。

-include file

引入某个头函数文件，例如命令 “gcc client.c -include /usr/include/example.h”，在编译 client 文件时需要使用 example.h 的头文件。

-Idir

gcc 在遇到源代码中“#include file.h”时，将在当前文件目录查找 file.h 头文件，如果没有找到，将到缺省目录中进行查找。在此命令指定目录后，gcc 将首先在指定目录进行头文件查找，如没有找到在按上述查找顺序进行查找。

-llibrary

指定 gcc 在链接阶段所使用的库文件。

-Ldir

指定 gcc 链接阶段库文件所在路径。

例如，gcc -o webserver webserver.o -L. -ldisplay，将 webserver.o 文件与库 libdisplay.so 链接在一起，并生成可执行程序 webserver.

-g

在编译过程中产生调试信息，这些信息可供 gdb 等调试器使用。

-static

用于 gcc 生成静态库文件

-shared

用于 gcc 生成动态库文件

-fPIC

表示生成与位置无关的代码。

例如，“gcc -shared -fPIC display.c -o libdisplay.so” 将生成 libdisplay.so 动态链接库。

-std

表示 gcc 支持的 C 语言标准，其取值有 C89，C99，gnu99 等，以表示其支持的 C 语言版本标准。例如，“gcc -std=C99 client.c -o client”，将 client.c 文件视为使用 C 语言 1999 年版本标准进行编写。

-pedantic

当 gcc 在编译时，将不符合相关语言标准的源代码进行标注，并产生相应的警告。

-Wall

让 gcc 产生尽可能多的警告信息。

-Werror

让 gcc 将警告信息看为程序的语法错误。使用此编译选项，将使得 gcc 停止在出现警告的位置。

-O0 -O1 -O2 -O3 表示编译器生成优化代码的程度。 其中-O0 表示没有优化；-O1 为缺省值，尽量采用一些优化算法降低代码大小和提高代码执行速度；-O2 会牺牲部分编译速度，除具有-O1 所有的优化外，还会采用支持目标配置的优化算法来提高代码执行速度；-O3 除具有-O2 所有优化的选项外，还利用 CPU 内部结构采用很多向量化优化算法，其产生的代码运行速度最快。

1.4.2 构建 makefile

Makefile 是一个包含命令集的文件，此文件中的命令能够被 make 程序所解析并执行，以完成大型程序的编译、部署等工作。可以想象一下在一个大型工程项目里面有成千上万个代码文件，而这些代码文件被放置在不同的目录里面。如果要将这些文件按照工程项目要求生成不同类型的可执行程序，那么该怎么按照这些要求来编译、链接和生成这些程序代码呢？上个世纪 70 年代贝尔实验室的 Stuart Feldman 在 unix 系统上创建了 make 工具来完成上述任务要求。

编写能够被 make 程序解析执行的 makefile 文件，需要掌握其编写规则。其具体编写规则如下。

```
target: prerequisites
        command1
        command2
         …
        commandn
```

其中，target 表示命令执行的目标。其可以是生成的目标文件，也可以是一个标签；prerequisites 表示完成 target 目标所需要的前提条件，前提条件可以是文件或标签；command1，comomand2，… ，commandn 表示要完成目标所需要执行的 shell 命令。

此规则可以被解析为要实现目标 target，需要先执行前提条件，如果前提条件已经被执行，则完成 command 中指定的命令。target 和 prerequisites 使得多个规则之间形成了偏序关系。make 程序总能够知道先执行哪个 target 和后执行哪个 target。例如下面的 makefile 文件，完成对拥有 3 个头文件和 2 个 C 文件的项目编译和程序生成。该 makefile 文件主要生成了 webserver.o libdisplay.so 和 webserver 三个文件。其中 webserver.o 是编译后的目标二进制文件；libdisplay.so 是静态库文件；webserver 是可执行程序。

```
webserver: webserver.o libdisplay.so
      gcc -g -o webserver webserver.o -L. -ldisplay

webserver.o: webserver.c webserver.h display.h counter.h
gcc -g -c webserver webserver.c
libdisplay.so: display.c display.h
gcc -g -shared -fPIC display.c -o libdisplay.so

clean:
  rm webserver webserver.o libdisplay.so
```

1.4.3 调试代码 GDB

GDB 是 GNU 项目下的调试器，其能够调试 Ada、C、C++、Objective-C、Pascal 等许多程序语言编写的程序。GDB 是 linux 平台下被广为使用的调试器，具有跟踪程序运行、断点调试、动态修改程序数据等特点。GDB 要进行指定程序调试的前提是该程序在使用 GCC 编译时使用参数 g。

本节将介绍其常用的命令，更为详细的命令参数请查阅其使用手册。需要注意的是，在 GDB 命令使用中为调试方便，很多命令有缩写方式。

- GDB 启动

GDB 调试指定程序包含以下三种启动方式：直接命令启动、恢复程序执行现场、调试指定运行程序。

“gdb program” 表示使用 gdb 启动一个指定程序，其中 program 表示此程序名。

“gdb program core”表示要恢复指定程序运行的现场，其中 core 表示程序非法执行后 core dump 产生的文件。此命令常用于分析程序运行崩溃的原因。如果让操作系统能够产生 core dump 需要使用“ulimit -c unlimited”命令来解除操作系统对生成 core 文件的限制。

“gdb program PID”表示跟踪调试目前正在运行的程序，其中 PID 表示此程序运行的进程标识符。该命令可以让 gdb 关联到正在运行的程序，并调试它。

- list 命令

当 GDB 启动后，在调试环境中可以使用 list 命令来查看程序文件的源代码，其缩写命令为“l”。list 命令后可以跟指定的代码行和指定的函数名。例如，“list 80” 或 “l 80” 表示列出代码行 80 位置处的源代码；而“list main”或“l main”表示列出 main 函数附近的源代码。

- break 命令

break 命令用来设置程序运行断点，其缩写命令为“b”。break 命令可以对指定函数、指定文件代码行、指定内存地址来设置断点。

“b function” 表示在指定函数入口设置断点。

“b linenum” 表示在指定代码行设置断点。

“b +offset” 或 “b - offset”表示在当前代码行后面或前面 offset 行来设置断点。

“b filename:function”表示在指定文件中的函数设置断点。

“b filename:linenum” 表示在指定文件中的代码行设置断点。

“b *address” 表示在程序运行的指定内存地址设置断点。

Break 命令还支持设置条件断点，其命令格式为“b … if condition” ，其中“…”表示为上述 break 命令参数，condition 为断点条件。例如，“b client.c:web if hit==1” 表示在参数变量 hit 等于 1 时位置为 client.c 文件中 web 函数的断点有效。

断点设置成功后会为此断点返回一个断点号，断点号与断点一一对应，可以作为其它断点相关命令的参数来使用。

- 断点操作命令

在通过 break 命令设置断点后，可以使用 info、clear、delete、disable 和 enable 等命令来操作断点。

“Info break” 命令可以查看目前设置的所有断点信息。

“clear”命令可以清除 break 命令设置的断点。例如“clear”清除所有 break 设置的断点；“clear linenum”清除指定行的断点；“clear filename:linenum”清除指定文件中代码行上的断点；“clear filename:function”清除指定文件中函数上的断点。

“delete breakpoints” 或 “delete range” 可以清除指定断点号的断点，其中 breakpoints 表示指定的断点号，如果没有指定断点号，则清楚所有的断点；range 表示断点号范围，例如 “clear 2-6”命令表示清除断点号 2-6 的断点。

“disable breakpoints”或“disable range”将使得指定断点号的断点失效，例如“disable 2-6”将使得断点号 2-6 的断点失效。与 clear 和 delete 命令相比，disable 命令并不删除断点，其可以通过 enable 命令来恢复失效的断点。

“enable breakpoints”或“enable range” 可以恢复失效的断点。如果想让断点在执行一次后马上失效，则可以使用“enable breakpoints once”命令。

- **watch 命令**

如果想让某个变量值发生变化后中断当前程序运行，可以使用 watch 相关命令。

“watch expr” 命令为变量 expr 设置一个观察点。一旦这个变量值发生变化，将中断当前程序运行。

“rwatch expr” 当 expr 变量被读取时中断当前程序。

“awatch expr” 当 expr 变量被读或被写时中断当前程序。

而对于观察点操作命令与断点操作命令相同，如下面的代码图所示。

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483c6 in main at test.c:5
        breakpoint already hit 1 time
4   hw watchpoint  keep y   x
        breakpoint already hit 1 time
(gdb) disable 4
```

- **运行程序命令**

如果想让被调试的程序从头开始执行可以使用 run 命令，其缩写为“r”。

如果程序执行时需要运行参数，可以使用“set args”设置程序启动运行的参数。例如，set args 8181 “/home/nwebdir”。

在程序被中断后，如果想让程序继续运行，可以使用 continue 命令，其缩写“c”。此命令将使得程序运行到下一个断点或直到观察点变量发生变化。

- **单步运行命令**

如果想让程序单步执行，则可以使用 step 命令，其缩写为“s”。当程序单步运行到函数时，将进入函数的内部。

“next”同样使程序单步执行，其缩写为“n”。但是当程序运行到函数时，使用此命令并不会让调试器进入函数，而是执行该函数，并跳到下一行。

“finish”命令将继续运行程序，直到当前函数运行完毕并返回。

“si”和“ni”命令与“s”和“n”类似，只不过它们作用于汇编指令上。

- **backtrace 命令**

用于打印当前函数调用栈的所有信息，其缩写为“bt”。

- **帧命令**

“frame”命令用于查看当前栈层的信息，其缩写为“f”。

“frame n”将程序运行栈切换到 n。其中 n 是栈中层次编号，0 表示栈顶。

- **查看运行数据**

“print expr”命令用于显示指定变量 expr 的数值，其缩写为“p”。

“print file::variable”或“print function::variable”可以用来显示指定文件或函数中的变量值。

“print address@len” 用来显示数组内指定长度内的数值，其 address 表示数组的首地址，len 表示要显示数据项的格式。例如，“print a@4”用来显示数组 a 中 4 个数据项下的数值。

“print $regiester”用来显示寄存器中数值，例如“print $pc”显示当前 pc 寄存器中数值。

在输出变量值时，可以指定输出变量的格式参数。其中，

x 表示按十六进制显示变量；

d 表示按十进制显示变量；

u 表示按十六进制格式显示无符号整型；

o 表示按八进制显示变量；

t 表示按二进制格式显示变量；

a 表示打印一个内存地址；

c 按字符格式显示变量;

f 按浮点数格式显示变量。

例如，“p/x k” 将把 k 变量数值按 16 进制输出。

- **display 命令**

“display expr”用来自动显示变量的数值。当每次程序中断或单步跟踪调试时，会自动显示变量 expr 中的数值。

- **查看内存命令**

“x address”用来查看指定内存中的数据。

- **退出命令**

“quit”命令使得 gdb 退出。

- **GDB 进程多线程调试**

GDB7.0 以上版本支持多进程调试，但需要通过设置 fork 模式参数来启动对多个进程的调试。fork 模式的参数通过两个命令来体现：set follow-fork-mode [parent|child] 和 set detach-on-fork [on|off]。 其中 follow-fork-mode 表示跟踪 fork 进程状态，如果设置为 parent，GDB 则跟踪调试父进程；如果设置为 child，GDB 则跟踪调试子进程；detach-on-fork 表示在 fork 后是否与不跟踪的进程脱离，如果设置为 on，则脱离不跟踪的进程，如果设置为 off，则不脱离不跟踪的进程。这两个参数的不同设置组合具有以下的意义。

表 1-1 GDB 多进程跟踪参数设置表

follow-fork-mode	detach-on-fork	含义
parent	on	只调试主进程（GDB 默认）
Child	on	只调试子进程
Parent	off	同时调试两个进程，gdb 跟主进程，子进程阻塞在 fork 位置
Child	off	同时调试两个进程，gdb 跟子进程，主进程阻塞在 fork 位置

“info inferiors”命令来查看正在调试的进程。

“Inferior infno”命令来切换调试的进程。

“add-inferior [-copies n][-exec executable]”命令来添加新的调试进程, 其中“-copies n”表示启动 n 份进程，“-exec executable”表示要启动进程的程序文件名。

“detach inferior inno”命令终止对指定进程的跟踪，其中 inno 为 gdb 中的进程标识号。

“kill inferior inno”命令关闭指定的进程

“Info threads”命令用来查看当前进程的线程态

“thread threadno”命令用来切换调试线程。

以下面代码为例，来说明如何使用 GDB 来调试多进程多线程程序。

```
#多进程多线程代码
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void childprocess();
void threadfunc();

int main(){
 pid_t pid=fork();
    if (pid == 0){
       childprocess();
    }
    else{
       pid_t parentpid=getpid();
       printf("Parent Id is %d\n", parentpid);
       printf("Child Id is %d \n", pid);
    }

}

void childprocess(){
 pid_t pid=getpid();
 pthread_t pt;
 int status=pthread_create(&pt,NULL, (void *)threadfunc,NULL);
 if (status!=0)
 {
       printf("Cannot create a new thread\n");
 }
 pthread_t tid=pthread_self();
 printf("Current process id is %d , current thread id is %ld \n", pid, tid);
 sleep(10000);
}

void threadfunc(){
 pid_t pid=getpid();
 pthread_t tid=pthread_self();
 printf("Current process id is %d , current thread id is %ld \n", pid, tid);
 sleep(10000);
}
```

调试过程如下所示。

```
gcc -g -o multiprocessthreads multiprocessthreads.c -lpthread    #编译上面的多进程和多线程程序
gdb multiprocessthreads    #启动 GDB
(gdb) set follow-fork-mode parent #  设置同时调试父子进程，gdb 跟主进程
(gdb) set detach-on-fork off
(gdb) b 10     #设置在代码行 10 处的断点
Breakpoint 1 at 0x4007c5: file multiprocessthreads1.c, line 10.
(gdb) b childprocess #设置函数断点
Breakpoint 2 at 0x400819: file multiprocessthreads1.c, line 23.
(gdb) b threadfunc   #设置函数断点
Breakpoint 3 at 0x400884: file multiprocessthreads1.c, line 36.
(gdb) r   #从头开始运行
Starting program: /root/book-examples/multiprocessthreads1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

```
Breakpoint 1, main () at multiprocessthreads1.c:10
10          pid_t pid=fork();       #停止在断点
(gdb) info inferiors       #查看进程信息，从下面信息看到目前仅有一个主进程
  Num   Description          Executable
* 1     process 20527       /root/book-examples/multiprocessthreads1
(gdb) n          #单步运行
[New process 20533]...
11          if (pid == 0){
(gdb) info inferiors # 查看进程信息，可以看到目前已经启动了两个进程，并且当前跟踪进行为父进程。
  Num   Description          Executable
  2     process 20533       /root/book-examples/multiprocessthreads1
* 1     process 20527       /root/book-examples/multiprocessthreads1
(gdb) n          #单步运行
15            pid_t parentpid=getpid();
(gdb) inferior 2     #转到子进程 2 进行跟踪调试
[Switching to inferior 2 [process 20533] (/root/book-examples/multiprocessthreads1)]
[Switching to thread 2 (Thread 0x7ffff7fca740 (LWP 20533))]
...
10          pid_t pid=fork();
Value returned is $1 = 0
(gdb) c        #在子进程中继续运行，并在 childprocess 断点处停止。
Continuing.
Breakpoint 2, childprocess () at multiprocessthreads1.c:23
23            pid_t pid=getpid();
(gdb) info threads #查看目前线程个数，下面两个线程分别为已经创建的两个进程中的线程。
  Id    Target Id          Frame
* 2     Thread 0x7ffff7fca740 (LWP 20533) "multiprocessthr" childprocess () at multiprocessthreads1.c:23
  1     Thread 0x7ffff7fca740 (LWP 20527) "multiprocessthr" main () at multiprocessthreads1.c:15
...
(gdb) n   #当执行 pthread_create 后，将创新新的线程，并且新创建的线程为 3
[New Thread 0x7ffff77f6700 (LWP 20550)]
[Switching to Thread 0x7ffff77f6700 (LWP 20550)]

Breakpoint 3, threadfunc () at multiprocessthreads1.c:36
36            pid_t pid=getpid();

(gdb) info threads   #查看目前所有线程，其中标号为“*”的表示目前正在跟踪的线程
  Id    Target Id          Frame
* 3     Thread 0x7ffff77f6700 (LWP 20550)··· threadfunc () at ultiprocessthreads1.c:36
  2     Thread 0x7ffff7fca740 (LWP 20533) ···   childprocess () at multiprocessthreads1.c:25
  1     Thread 0x7ffff7fca740 (LWP 20527) ··· main () at multiprocessthreads1.c:15
(gdb) n #运行线程 3 中的代码
Current process id is 20533 , current thread id is 140737353918272
37            pthread_t tid=pthread_self();
(gdb) thread 2 #跟踪线程 2.
…
```

1.4.4 服务性能测试工具

1. **http_load**

http_load 能够对 web 服务器进行性能压力测试。可以按照官方网站（http://acme.com/software/http_load/）的说明来进行安装。其主要参数如下。

“-parallel num” 表示并发客户端的数量。

“-fetches num” 表示所有客户端总共访问次数。

“-rate num” 表示每秒访问频率。

“-seconds num”表示总共访问时间，以秒为单位。

“urls” 为保存访问网页链接的文件。在文件内部保存要访问的页面链接地址。其文件格式如下所示。

```
http://127.0.0.1:8088/index.html
http://127.0.0.1:8088/example.html
...
```

例如运行测试命令“http_load -parallel 5 -fetches 50 -seconds 20 urls”，表示启动同时5 个客户端，并在 20 秒时间内共抓取 50 个网页。其运行结果，如下所示。

```
20 fetches, 5 max parallel, 5880 bytes, in 20.0022 seconds
294 mean bytes/connection
0.999891 fetches/sec, 293.968 bytes/sec
msecs/connect: 107.569 mean, 1017.36 max, 3.426 min
msecs/first-response: 4141.73 mean, 5013.75 max, 5.283 min
HTTP response codes:
  code 200 -- 20
```

执行的结果反映了如下的信息。

第一行 “20 fetches, 5 max parallel, 5880 bytes, in 20.0022 seconds”表明在 20.0022 秒时间内，最大启动了 5 个客户端，共完成 20 次抓取，总共传输了 5880 字节。可以看出在20 秒内没有完成 50 次网页的抓取工作。

第二行 “294 mean bytes/connection” 表示每次连接平均传输的数据量。

第三行“0.999891 fetches/sec, 293.968 bytes/sec”表示每秒平均完成多少次网页传输，以及每秒传输的数据量。其中 fetches/sec 为常用的性能指标参数 QPT（每秒响应数量）。

第四行“msecs/connect: 107.569 mean, 1017.36 max, 3.426 min” 表示建立请求连接的平均时间、最大时间和最小时间（单位为毫秒）。其中 `msecs/connect` 为常用的性能指标参数（客户端与服务端建立连接的平均时间）。

第五行“msecs/first-response: 4141.73 mean, 5013.75 max, 5.283 min”。表示每个连接（客户端）从发出 htpp 请求消息到开始接受服务器第一个响应消息的平均时间、最大时间和最小时间。这里统计的时间是第四行参数已经建立好连接基础之上的发送请求消息到接受响应消息之间的时间，可以看为是服务器与客户端建立连接后，响应客户请求网页的时间。

第六行 “HTTP response codes: code 200 -- 20” 表示响应代码为 200 的有 20 个。

通过观察上面的参数数据，能够知道 web 服务器所支持的并发访问量及响应时间、web 所支持的并发访问量和单位时间网络传输数据量等信息。通过这些信息，可以对 web 服务器性能做出分析。例如，上面的测试中，可以看出每秒才完成一个网页的数据传输，而传输的数据量仅为约 294 字节，并且“msecs/first-response: 4141.73 mean”数值较大，可以看出每个连接都等待了很长时间才得到服务器的响应信息。

2. Perf

Perf (Performance Event) 是 Linux 内核的性能分析工具。它基于事件采样原理，以固定频率采集样本，分析这些样本在事件或函数中的数量，进而统计出各个函数所消耗时间。Perf 能够分析服务器系统的性能热点，找到 Web 服务器代码中存在的问题。Perf 主要包含以下五种工具集：

- Perf list 命令

  Perf list 用来查看 perf 所支持的事件，这些事件包括软件事件和硬件事件。

```
# perf list

List of pre-defined events (to be used in -e):
```

```
  cpu-cycles OR cycles                                [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend     [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend       [Hardware event]
  instructions                                        [Hardware event]
  cache-references                                    [Hardware event]
  cache-misses                                        [Hardware event]
  branch-instructions OR branches                     [Hardware event]
  branch-misses                                       [Hardware event]
  bus-cycles                                          [Hardware event]

  cpu-clock                                           [Software event]
  task-clock                                          [Software event]
  page-faults OR faults                               [Software event]
  minor-faults                                        [Software event]
  major-faults                                        [Software event]
  context-switches OR cs                              [Software event]
  cpu-migrations OR migrations                        [Software event]
  alignment-faults                                    [Software event]
  emulation-faults                                    [Software event]

  L1-dcache-loads                                     [Hardware cache event]
  L1-dcache-load-misses                               [Hardware cache event]
  L1-dcache-stores                                    [Hardware cache event]
  L1-dcache-store-misses                              [Hardware cache event]
  L1-dcache-prefetches                                [Hardware cache event]
  ...
```

参数 e 用来指定监控的事件，具体参数使用格式如下：

-e <event>: [u | k | h | G | H]

event 为要监控事件的名称; [u | k | h | G | H] 表示监控时间的位置, u 表示用户空间，k 表示内核空间，h 表示 hypervisor，G 表示在 KVM guests，H 表示不再 KVM guests。

例如，perf -e cycles 监控 perf 来监控 cup 运行指令次数的事件

- perf stat 或 perf top 命令

perf stat 用于分析统计程序运行的总体性能情况。例如针对如下代码，执行编译命令“gcc -o perf-test -g -pg perf-test.c”。

```
//perf-test.c
#include "stdio.h"

void test() {
 int i,j;
 for (int i = 0; i < 1000000; i++){
        j=i;
 }
}

int main(void){
 test();
}
```

然后运行命令“perf stat ./perf-test”，执行结果如下。其中，

task-clock 表示 CPU 利用率；

context-switches 表示进程上下文交换次数；

cpu-migrations 表示运行指令迁移 CPU 的次数（从一个 CPU 移动到另一个 CPU）；

page-faults 表示“缺页”次数；

cycles 表示 CPU 逻辑时钟运行周期次数；
instructions 表示运行的机器指令数量；
branches 表示分支数量（代码中的跳转指令会产生分支）；
branches-misses 表示分支预测失败的次数。

```
Performance counter stats for './perf-test':

          6.323455      task-clock (msec)         #    0.924 CPUs utilized
                 0      context-switches          #    0.000 K/sec
                 0      cpu-migrations            #    0.000 K/sec
                49      page-faults               #    0.008 M/sec
         7,820,657      cycles                    #    1.237 GHz
<not supported>      stalled-cycles-frontend
<not supported>      stalled-cycles-backend
         5,552,990      instructions              #    0.71  insns per cycle
         1,109,974      branches                  #  175.533 M/sec
             6,413      branch-misses             #    0.58% of all branches

       0.006842042 seconds time elapsed
```

从上面的结果中，很容易知道该程序是计算密集型任务，其 CPU 利用率为 0.924。

下面使用 perf stat 分析 1.3 节中 socket 服务端代码（webserver.c）运行过程。首先，在服务端执行“gcc -o single-process-server -g -pg webserver.c”，生成程序 single-process-server；然后启动对给程序的分析，执行命令“perf stat ./single-process-server 8088 web”。在客户端，运行 http_load（执行命令“http_load -parallel 5 -fetches 50 -seconds 20 urls”）向 single-process-server 发送请求消息。当 http_load 运行结束后，在服务端按“ctrl+c”结束 perf，这时 perf 打印的结果如下所示。从结果中可以看出，single-process-server 是 I/O 密集型任务，因为 CPU 的利用率近似为 0。

```
Performance counter stats for './single-process-server 8088 web':

          4.752166      task-clock (msec)         #    0.000 CPUs utilized
                28      context-switches          #    0.006 M/sec
                 1      cpu-migrations            #    0.210 K/sec
                56      page-faults               #    0.012 M/sec
         5,676,956      cycles                    #    1.195 GHz
<not supported>      stalled-cycles-frontend
<not supported>      stalled-cycles-backend
         2,875,555      instructions              #    0.51  insns per cycle
           549,947      branches                  #  115.726 M/sec
            44,259      branch-misses             #    8.05% of all branches

      38.428969783 seconds time elapsed
```

perf top 类似 top 命令，能够定时刷新显示系统消耗过高的事件。例如，执行命令“perf top -e cycles”将能监控到系统内消耗 cycles（CPU 资源）较多的代码。

```
Samples: 4K of event 'cycles', Event count (approx.): 173567283
Overhead  Shared Object                     Symbol
  11.68%  [kernel]                        [k] igb_rd32
   3.06%  dockerd                         [.] runtime.scanobject
```

```
    1.85%  dockerd                  [.] runtime.greyobject
    1.24%  virtuoso-t               [.] 0x00000000004bc7ec
    1.19%  dockerd                  [.] runtime.heapBitsForObject
    1.17%  [kernel]                 [k] native_write_msr_safe
    1.16%  [kernel]                 [k] menu_select
    1.15%  perf                     [.] 0x0000000000080b77
    1.02%  [kernel]                 [k] int_sqrt
    0.88%  [kernel]                 [k] entry_SYSCALL_64
    0.88%  [kernel]                 [k] delay_tsc
    0.87%  perf                     [.] 0x000000000008b804
    0.85%  [kernel]                 [k] _raw_spin_lock_irqsave
```

• perf record 命令

perf record 命令能够记录指定事件在各个函数运行中出现的次数比例，并将相关信息保存到本地目录下 perf.data 文件中。例如，执行命令“perf record -e cpu-clock -g ./perf-test”后，perf 将统计 perf-test 各个函数所消耗 CPU 时间。

• perf report 命令

perf report 命令读取 perf record 创建的数据文件，并给出热点分析。例如，通过命令“perf report”，将上面的 perf-test 统计结果进行如下显示。

```
Samples: 27  of event 'cpu-clock', Event count (approx.): 6750000
     Children      Self  Command      Shared Object       Symbol
+    88.89%     0.00%    perf-test    perf-test           [.] main
-    88.89%    88.89%    perf-test    perf-test           [.] test
        __libc_start_main
        main
        test
+    88.89%     0.00%    perf-test    libc-2.19.so        [.] __libc_start_main
+     7.41%     3.70%    perf-test    [kernel.kallsyms]   [k] __do_page_fault
+     7.41%     0.00%    perf-test    [kernel.kallsyms]   [k] do_page_fault
+     7.41%     0.00%    perf-test    [kernel.kallsyms]   [k] page_fault
+     3.70%     0.00%    perf-test    [kernel.kallsyms]   [k] filemap_map_pages
+     3.70%     0.00%    perf-test    [kernel.kallsyms]   [k] unmap_page_range
+     3.70%     0.00%    perf-test    [kernel.kallsyms]   [k] handle_pte_fault
+     3.70%     0.00%    perf-test    [kernel.kallsyms]   [k] unmap_single_vma
```

上面结果可以看到 CPU 时间大多数消耗在 test 函数。通过选择 test，并选择 annotate test 项，显示如下。可以看到 perf-test 代码大量时间浪费在跳转指令 “jle 16”。

```
test  /home/csqlu/book-example/perf-test
           |                  for (int i = 0; i < 1000000; i++)
           |          movl     $0x0,-0x8(%rbp)
           |        ↓ jmp      20
           |                  {
           |                          j=i;
    12.50  | 16:  ┌──→mov      -0x8(%rbp),%eax
           |      |    mov      %eax,-0x4(%rbp)
           |      | #include "stdio.h"
           |      |
           |      | void test() {
           |      |              int i,j;
           |      |              for (int i = 0; i < 1000000; i++)
           |      |    addl     $0x1,-0x8(%rbp)
    12.50  | 20:  |    cmpl     $0xf423f,-0x8(%rbp)
    75.00  |      └────jle      16
           |                  {
           |                          j=i;
           |                  }
```

```
        |     }
        |        nop
        |        leaveq
        |     ← retq
```

- perf timechart 命令

  perf timechart 将使用图形的方式来展现程序在系统中的运行情况。例如，在服务端执行“perf timechart record ./single-process-server 8088 web”命令，在客户端执行命令“http_load -parallel 5 -fetches 50 -seconds 20 urls”后，在服务端按“ctrl+c”终止 perf timechart 运行。然后运行 perf timechart 命令，将输出 output.svg 文件。打开此文件，如下图所示。

  通过观察此图，很容易发现 CPU 利用率并不高，并且 single-process-server 进程绝大部分时间处在 sleep 或 I/O 操作的阻塞中，这也印证了上面通过 perf stat 命令得到的结论，即 single-process-server 是 I/O 密集型任务。

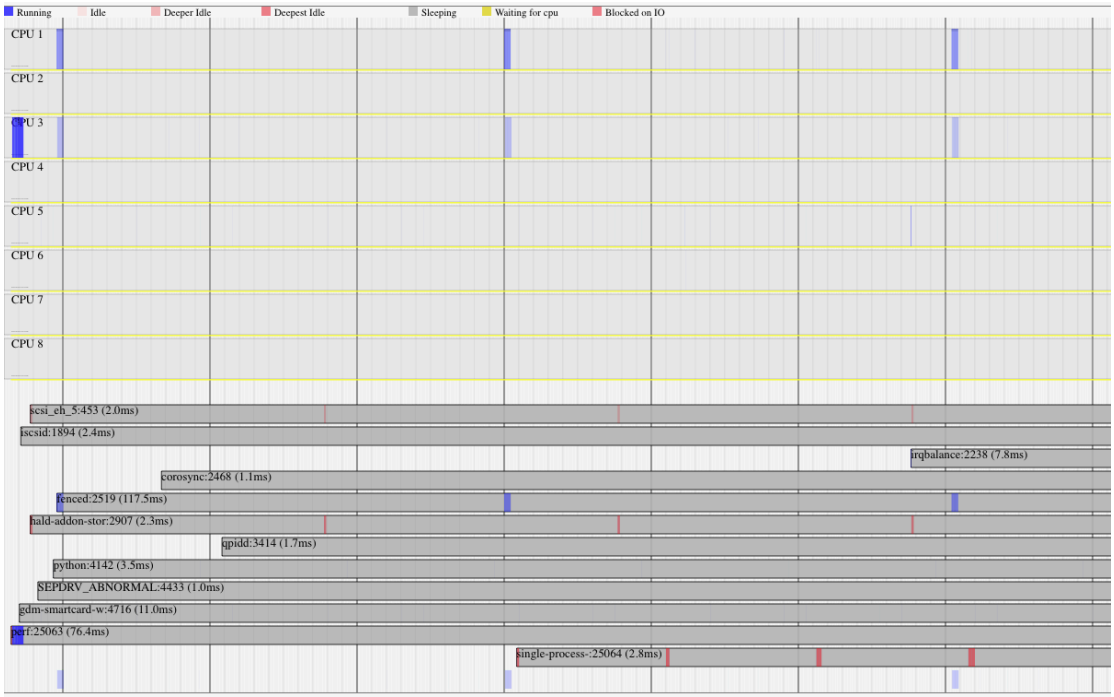

图 1-4 single-process-server 的 timechart 分析

perf 除了上述命令外，还有 perf sched，perf lock 和 perf kmem 等命令，它们分别统计调度器、锁和内核内存 slab 的使用情况。具体使用方法，请参考 perf Examples[3]。

3. vmstat

vmstat 命令能够以指定时间间隔显示系统中 CPU、内存、虚拟内存及 I/O 的使用情况。常用的命令格式为：vmstat [interval [count]]。其中 delay 为统计数据的时间间隔，count 为统计次数。

例如命令 “vmstat 2 10”将以每 2 秒的时间间隔统计 10 次系统状态，显示如下。各列的含义如下。

```
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b      swpd     free     buff    cache   si   so    bi    bo   in   cs us sy id wa st
 0  0         0 10271748 416908 4803120    0    0     4     2   16    9  0  0 100  0  0
 0  0         0 10271468 416908 4803124    0    0     0     0  139  323  0  0 100  0  0
 0  0         0 10271716 416908 4803124    0    0     0    44  110  104  0  0 100  0  0
```

[3] http://www.brendangregg.com/perf.html

```
 0  0      0 10271972 416908 4803124    0    0     0     0  137  320  0  0 100  0  0
 0  0      0 10271964 416912 4803124    0    0     0    38  115  103  0  0 100  0  0
 0  0      0 10272048 416912 4803124    0    0     0     0  128  317  0  0 100  0  0
 0  0      0 10272184 416912 4803124    0    0     0     0  133  323  0  0 100  0  0
 0  0      0 10272184 416916 4803124    0    0     0    38   99   96  0  0 100  0  0
 0  0      0 10272076 416916 4803124    0    0     0     0  140  328  0  0 100  0  0
 0  0      0 10272200 416916 4803128    0    0     0    46  147  334  0  0 100  0  0
```

r 表示运行任务数量，如果此数值远大于 CPU 数量，则表示系统的 CPU 运算很繁忙；**b** 表示阻塞进程的数量；**swpd** 表示虚拟内存已经使用的大小，如果此数值大于 0，表示物理内存可能已经不足；**free** 表示空闲的物理内存大小；**buff** 表示系统缓冲区大小，用来存储目录里面有什么文件及文件相关信息；**cache** 表示文件缓存区大小，用来存储文件中的一部分内容；**si** 为每秒从磁盘中读入虚拟内存的数据量；**so** 为每秒虚拟内存写入磁盘的数据量；**bi** 表示 I/O 块设备（block input）每秒接收的数据量（对应向 I/O 设备写数据）；**bo** 表示 I/O 块设备（block output）每秒发送的数据量（对应从 I/O 设备读数据）；**in** 表示每秒 CPU 中断次数；**cs** 表示上下文交换次数，进程、线程间切换需要上下文交换，系统调用也需要上下文交换；**us** 表示用户进程所占用 CPU 的时间百分比；**sy** 表示系统进程所占用 CPU 的时间百分比；**id** 表示 CPU 空闲时间的百分比；**wa** 表示 I/O 等待时间百分比；**st** 表示从虚拟机偷的时间。

4. iostat **和** iotop

iostat 命令主要用于统计磁盘活动相关情况，除此之外也能统计 CPU 的使用情况。例如，命令“iostat -k 2 10”执行结果如下所示。其中 tps 表示该设备每秒 I/O 请求次数, kB_read/s 表示每秒从磁盘中读多少字节，kB_wrtn/s 表示每秒向磁盘写多少字节。

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.07    0.00    0.24    0.02    0.00   99.67

Device:            tps   kB_read/s   kB_wrtn/s   kB_read   kB_wrtn
sdb               0.17       12.81        1.15   3112262    279640
sda               1.45        6.66       13.93   1619022   3383780

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.00    0.00    0.06    0.00    0.00   99.94

Device:            tps   kB_read/s   kB_wrtn/s   kB_read   kB_wrtn
sdb               0.00        0.00        0.00         0         0
sda               0.00        0.00        0.00         0         0

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.06    0.00    0.31    0.00    0.00   99.63

Device:            tps   kB_read/s   kB_wrtn/s   kB_read   kB_wrtn
sdb               0.00        0.00        0.00         0         0
sda               1.00        0.00       24.00         0        48

...
```

iotop 命令以动态刷新的形式不停显示最新的系统 I/Os 使用情况，具体情况如下所示。

```
Total DISK READ: 0.00 B/s | Total DISK WRITE: 0.00 B/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
17120 be/4  root        0.00 B/s    3.79 K/s  0.00 %  0.00 % ./single-process-server 8088 web
    1 be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 % init
    2 be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [kthreadd]
    3 rt/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [migration/0]
    4 be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [ksoftirqd/0]
    5 rt/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [migration/0]
```

```
 6 rt/4    root    0.00 B/s    0.00 B/s    0.00 %    0.00 % [watchdog/0]
 7 rt/4    root    0.00 B/s    0.00 B/s    0.00 %    0.00 % [migration/1]
 8 rt/4    root    0.00 B/s    0.00 B/s    0.00 %    0.00 % [migration/1]
 9 be/4    root    0.00 B/s    0.00 B/s    0.00 %    0.00 % [ksoftirqd/1]
10 rt/4    root    0.00 B/s    0.00 B/s    0.00 %    0.00 % [watchdog/1]
11 rt/4    root    0.00 B/s    0.00 B/s    0.00 %    0.00 % [migration/2]
12 rt/4    root    0.00 B/s    0.00 B/s    0.00 %    0.00 % [migration/2]
```

5. netstat

netstat 命令能够查看系统网络接口的使用情况。其与 watch 命令结合，能够以动态视图的方式，来实时观察系统的各个网络接口的发送和接受消息的情况。例如，执行命令“watch -n 1 -d netstat -antop”将以每秒刷新屏幕，并将各个接口的统计信息输出到窗口，具体如下所示。其中，Recv-Q 是此端口的接受缓冲区中数据的字节数，这些数据并未被用户取走；Send-Q 指的是此端口的发送缓冲区中数据的字节数，这些数据被用户程序存放在此缓存区，但还未被发送出去。如果 Recv-Q 的数字过大，则说明接收消息的程序出现了拥塞/阻塞，不能即时从接收缓冲区提取数据。同理，Send-Q 的数字过大，则说明网络本身出现拥塞，不能向网络另一端发送数据。

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address     Foreign Address       State       PID/Program name     Timer
tcp        0      0       0.0.0.0:111           0.0.0.0:*           LISTEN      2260/rpcbind        off (0.00/0/0)
tcp        0      0       0.0.0.0:80            0.0.0.0:*           LISTEN      3391/nginx          off (0.00/0/0)
tcp        0      0       0.0.0.0:8084          0.0.0.0:*           LISTEN      4142/python         off (0.00/0/0)
tcp        0      0     0.0.0.0:8088            0.0.0.0:*           LISTEN      30263/./single-proc off
tcp        0      0    0.0.0.0:39272            0.0.0.0:*           LISTEN      2163/stap-serverd off (0.0/0/0)
tcp       80      0   10.3.40.47:8088    10.120.53.172:56902    ESTABLISHED -                   off (0.00/0/0)
tcp       79      0 10.3.40.47:8088      10.120.53.172:56904    ESTABLISHED -                   off (0.00/0/0)
tcp       79      0 10.3.40.47:8088      10.120.53.172:56905    ESTABLISHED -                   off (0.00/0/0)
tcp        0      0   10.3.40.47:22      10.120.53.172:54936    ESTABLISHED 26813/sshd      keepalive
...
```

1.4.5 性能指标

Berkeley 大学的研究人员给出了计算机系统中不同设备和组件的在 2019 年的数据传输速度[4]，具体如下表 1-2 所示。

在进行系统设计和性能分析时，可以参考这个数据表来发现影响系统运行性能的关键点。例如，对于一个 I/O 读写频繁的程序，经过测试发现系统通过磁盘读写文件的速度在 5MB/秒，远远低于下面表格中的磁盘传输数据速度项。因此可以认为影响目前系统性能的一个因素为文件读写。通过进一步分析，发现这个系统在进行文件读写过程中出现了大量的阻塞操作，从而导致文件读写速度较慢。

表 1-2 计算机系统各个器件的数据传输速度表

系统器件数据传输	时间
L1 cache reference 读取 CPU 的一级缓存	1 ns
Branch mispredict(转移、分支预测)	3 ns
L2 cache reference 读取 CPU 的二级缓存	4 ns
Mutex lock/unlock 互斥锁\解锁	17 ns
Main memory reference 读取内存数据	100 ns

[4] https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Compress 1K bytes with Zippy 1k 字节压缩	2,000 ns
Send 2,000 bytes over commodity network	62 ns
Read 4K randomly from SSD 在 SSD 中随机读 4K 字节	16,000ns
Read 1 MB sequentially from SSD 在 SSD 中顺序读 1MB	62,000ns
Read 1 MB sequentially from memory 从内存顺序读取 1MB	4,000 ns
Round trip within same datacenter 从一个数据中心往返一次 ping	500,000 ns
Disk seek 磁盘寻道时间	3,000,000 ns
Read 1 MB sequentially from disk 从磁盘里面读出 1MB	947,000 ns
Send packet CA->Netherlands->CA 一个包的一次远程访问	150,000,000 ns

1.5 实验 1 Web 服务器初步实现

题目 1：创建 makefile 文件，将 1.3 节中的 webserver.c 代码进行编译为 webserver 可执行程序。

题目 2：在指定目录内准备好 html 文件，以及这些文件中链接的图片。例如，在 /home/web 目录下有 index.html 文件和图像文件 favicon.ico、example.jpg，其代码如下所示。

```
<html>
  <head>
    <link rel="shortcut icon" href="favicon.ico"  type="image/x-icon"/>
       <title>The example web</title>
  </head>
  <body>
       <H1>webserver test page</H1>
       <p>
      Not pretty but it should prove that webserver works :-)
    <p>
       <IMG SRC="example.jpg">
  </body>
</html>
```

首先启动 webserver 程序， 例如，“webserver 8088 /home/web”命令将服务器的侦听端口设置为 8088，检索文件的根路径为“/home/web”。然后，在浏览器中输入“http://127.0.0.1:8088/index.html”，观察浏览器中是否能够正常显示网页。在目录中查找 webserver.log 文件，将其打开观察日志信息。

请解释为什么在浏览器中仅请求一次网页，而实际上 webserver 接收了很多次从浏览器发出的文件请求？

请查阅相关文献，说明浏览器请求网页文件时，为加快 html 网页显示的速度，都采用了什么样的技术？

题目 3：修改 webserver.c 文件中 logger 函数源代码，使得日志文件中每行信息的起始部分均有时间信息，以表示这行信息被写入的时间。

题目 4：在浏览器中多次快速点击刷新按钮后，为什么浏览器要隔很长一段时间才开始显示页面？请结合日志文件中的信息来分析具体原因。

题目 5：使用 http_load 工具对此 webserver 程序进行性能测试，并记录其返回的各种参数数据。同时在服务器端，使用 vmstat、iostat 和 iotop 等工具收集 webserver 运行时系统的各种数据，并对 webserver 进行分析，结合它的代码说明其对系统所带来的各种消耗。

题目 6：在 server.c 中增加相关计时函数，分析一下程序的哪个部分最耗时？使用 perf 工具来跟踪 webserver 程序，根据其运行报告进行程序性能分析，请指出 webserver 中比较耗费时间的函数有哪些？

题目 7：根据题目 5 和题目 6 的结论，能否指出 webserver 性能低下的原因？并给出相应的解决方法？