第三章 基于 I/O 多路复用的并发 Web 服务器实现

3.1 Web 服务器源代码组织结构

第二章中已经介绍了 Web 服务器的工作原理和基本知识并且分析了实现它所用到的关键技术，接下来将要讨论这样一个基于 I/O 多路复用的并发 Web 服务器的源代码组织结构和关键技术的代码实现。Web 服务器的目录组织结构主要分为三部分，包括/bin、/src 和主目录下的 C 语言源代码。

bin 目录下包含的是各种 CGI 程序的 C 源代码和编译后的可执行程序。

src 目录下包含各种静态资源，例如 TXT 纯文本文件、PDF 格式的文本文件、PNG 格式和 JPEG 格式的图片、MPEG 格式的视频文件以及 HTML 文件等供 Web 服务器操作的诸多类型的静态资源。

./主目录下实现基于 I/O 多路复用并发 Web 服务器的 C 语言源代码主要分为三部分，一部分是增加自写系统调用错误报告的系统调用函数封装：

err_handle_wra.c	用于系统调用中打印对应的系统调用错误信息
sock_intf_wra.c	增加错误报告对系统实现的套接字接口函数再封装
unixIO_wra.c	增加错误报告对系统实现的低级应用接口 UnixI/O 函数再封装
mem_map_wra.c	增加错误报告对系统实现的用户级内存映射函数再封装
pro_ctrl_war.c	增加错误报告对系统实现的进程控制函数再封装

第二部分则是基于系统实现的系统调用进行改进的代码实现：

CS_helper_funcs.c	对系统实现的套接字接口进行组合使用更加合理有效建立连接
rIO_pkg.c	基于系统实现的 UnixI/O 构建健壮的 I/O 包--Robust I/O

第三部分是实现 Web 服务器基本功能和扩展功能的代码实现：

service.c	Web 服务器实现 HTTP 请求处理等基本功能的源代码
concurrent.c	Web 服务器实现基于 I/O 多路复用的抗并发能力的源代码
main.c	启动 Web 服务器的源代码

此外还有一些重要的源代码：

general.h	Web 服务器涉及的所有结构体和函数声明头文件
general.a	整合所有其他源代码文件的静态连接库文件，配合头文件使用

在编译 Web 服务器主目录下的源代码时，如果使用 GCC 编译器，首先要使用`gcc -c 源代码文件名列表`命令将除了 mian.c 之外的 err_handle_wra.c、sock_intf_wra.c 等源文件编译成.o 文件即对象文件；然后使用`ar crs general.a *.o`命令将编译好的所有对象文件链接成一个.a 文件即静态链接库文件。最后配合头文件 general.h 使用`gcc -o main main.c general.a`命令链接静态链接库形成单一的一个.out 可执行程序 main。这样一个过程就完成了这个小巧但是功能齐全的基于 I/O 多路复用的并发 Web 服务器的编译工作。

3.2 网络通信与资源操作基础实现

在这一节中主要介绍 Web 服务器实现网络通信与资源操作的低级应用接口和系统调用支持以及对他们的一些改进内容。其中主要包含对套接字接口的封装与组合改进、Unix I/O 的封装和基于此的 Robust I/O 实现以及用户级内存映射的函数封装。除了这三个比较重要的技术点，错误处理和进程控制这些比较应用广泛的系统调用代码实现，在本节中不再作详细的介绍。

3.2.1 套接字接口

在第二章的 2.1.1 节中已经详细介绍了作为 TCP/IP 协议系统软件实现 API 的套接字接口。在服务器的网络通信中只需要用到如图 3.1 所示，用于创建套接字描述符的 Socket 函数、检验套接字地址是否被使用的 Setsockopt 函数、绑定套接字描述符和套接字地址的 Bind 函数、将套接字描述符转换为监听描述符的 Listen 函数和等待客户端连接请求的 Accept 函数。这些函数在 sys/socket.h 中都有良好的实现，本文在此处只是在此基础上增加了调用是的错误处理，当调用出错是报告错误的函数调用信息。

```
int Socket(int domain, int type, int protocol);
void Setsockopt(int s, int level, int optname, const void *optval, int optlen);
void Bind(int sockfd, struct sockaddr *my_addr, int addrlen);
void Listen(int s, int backlog);
int Accept(int s, struct sockaddr *addr, socklen_t *addrlen);
                                                                    // ./general.h
```

图 3.1 套接字接口封装

根据 2.1.1 节中的套接字接口网络应用示例可知，在服务端进程完成一个监听套接字的创建需要经过套接字描述符创建、套接字描述绑定套接字地址和套接字描述符转换为一个监听套接字三个步骤。这个过程分别需要调用 socket、bind 和 listen 三个函数，如果任意一个函数调用出错都将导致服务端进程的监听套接字创建失败。为了避免这一情况，本文将一个监听套接字的创建过程视为一个元操作，任意函数调用错误都无法正确创建监听描述符，并需要从起点重新开始。这一方案的代码实现如图 3.2 中的 open_listenfd 函数实现。

```
1  int open_listenfd(int port)
2  {
3      int listenfd, optval=1;
4      struct sockaddr_in serveraddr;
5
6      /* 创建socket描述符 */
7      if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
8          return -1;
9      if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval ,
   sizeof(int)) < 0)
10         return -1;
11
12     /* 为套接字地址serveraddr各属性赋值 */
13     bzero((char *) &serveraddr, sizeof(serveraddr));
14     serveraddr.sin_family = AF_INET;
15     serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
16     serveraddr.sin_port = htons((unsigned short)port);
17
18     /* 将socket描述符绑定到套接字地址 */
19     if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
20         return -1;
21
22     /* 将socket描述符转换为监听描述符 */
23     if (listen(listenfd, LISTENQ) < 0)
24         return -1;
25
26     return listenfd;
27 }                                                    // ./CS_helper_funcs.c
```

图 3.2 open_listenfd 函数创建监听描述符

如图 3.2 所示，在第 7 行、第 9 行、第 19 行和第 23 行中分别调用创建套接字描述符的 Socket 函数、检验套接字地址是否被使用的 Setsockopt 函数、绑定套接字描述符和套接字地

址的 Bind 函数和将套接字描述符转换为监听描述符的 Listen 函数完成一个监听套接字 listenfd 的创建，且一旦任意一个调用出错都将返回-1 表示监听套接字创建失败。注意到图中第 13 到 16 行是对套接字地址赋值的一个过程，而一个套接字地址在系统中实现为如图 3.3 所示的结构体，sin_family 表示网络通信所使用的协议簇、sin_port 表示临时分配的端口好、sin_addr 表示主机的 IP 地址。

```
struct sockaddr_in{
    uint16_t sin_family;          //协议簇
    uint16_t sin_port;            //端口号
    struct in_addr sin_adrr;      //IP地址
    unsigned char sin_zero[8];    //结构体大小
};
```

图 3. 3 sockaddr_in 套接字地址结构体

3.2.2 系统级 I/O 和内存映射

和套接字接口封装一样，在本节将 Linux 内核引出的这组低级应用接口 Unix I/O 进行封装。如图 3.4 所示，主要封装了用于打开或创建文件的 Open 函数、用于读写文件的 Read 和 Write 函数、用于关闭文件的 Close 函数、用于读取文件元信息的 Stat 函数以及在提供动态内容服务中调用 CGI 程序时需要用到的 Dup2 函数。C 语言标准库中也实现了这些低级接口，和套接字接口封装的处理方法一样，只是在调用这些标准库实现的函数时进行的是否调用成功的判断，并在调用失败是打印错误信息。

```
int Open(const char *pathname, int flags, mode_t mode);
ssize_t Read(int fd, void *buf, size_t count);
ssize_t Write(int fd, const void *buf, size_t count);
void Close(int fd);
void Stat(const char *filename, struct stat *buf);
int Dup2(int fd1, int fd2);
                                                    // ./general.h
```

图 3. 4 Unix I/O 输入输出函数封装

本文在前面已经提到，在网络应用的环境中使用基础的 Unix I/O 存在一定的缺陷，并且提出了使用健壮的 I/O 包，Robust I/O 的解决方案。基于 Unix I/O 的封装，我们在 rIO_pkg.c 源文件中实现了不带缓冲的输入输出函数 rio_readn 和 rio_writen，也实现了带缓冲的输入函数 rio_readnb 和 rio_readlineb。本文的 Web 服务器主要使用带缓冲输入函数 rio_readlineb 和不带缓冲的输出函数 rio_wirten 完成 I/O 请求的处理，所以下文将主要介绍这连个函数的实现。

```
#define RIO_BUFSIZE 8192
typedef struct {
    int rio_fd;                // 缓冲区描述符
    int rio_cnt;               // 缓冲区内未读字节
    char *rio_bufptr;          // 下一个未读字节
    char rio_buf[RIO_BUFSIZE]; // 缓冲区
} rio_t;
                                                    // ./general.h
```

图 3. 5 Robust I/O 读缓冲区 rio_t 结构体

采用 Robust I/O 可以让服务器更加高效地在从网络读写数据或者读写数据到网络上的过程完成从文件读写文本行或二进制数据的工作。其中，起到关键作用之一的就是应用级读缓冲区 rio_t 的存在，它将文件内容缓存在缓存区中避免了高频反复的访问文件操作。如图 3.5 所示，一个 rio_t 包含缓冲区描述符 rio_fd、表示缓冲区中未读字节数的 rio_cnt、标记下

一未读字节的 rio_bufptr 和表示缓冲区的 rio_buf 四个属性，他们完成的标识了一个读缓冲区的各种信息。而图 3.6 中的 rio_readinitb 函数就是用来初始化这一读缓存区的，如第 3 到 5 行所示，分别初始化 rio_cnt 为 0 并让 rio_bufptf 指向缓冲区的开头。

```
1  void rio_readinitb(rio_t *rp, int fd)
2  {
3      rp->rio_fd = fd;
4      rp->rio_cnt = 0;
5      rp->rio_bufptr = rp->rio_buf;
6  }                                                    // ./rIO_pkg.c
```

图 3. 6 rio_readinitb 函数

在 Web 服务器中最多使用的就是读取 HTTP 请求和进行静态资源的读取工作，这些内容中很多涉及的是文本行的读取，而 rio_readlineb 函数就是为这一需求诞生的，它从内部读缓冲区 rio_t 中拷贝一个文本行，当读缓存区为空时则需要调用 read 函数把它填充满。基于 Linux 内核 read 函数的带缓存版本的 rio_read 函数，rio_readlineb 函数最多调用 maxlen-1 次 rio_read 函数完成文本行读取。rio_readlineb 函数的实现如图 3.7 所示，第 7 行到第 12 行是在数据可读的情况下调用 rio_read 函数完成数据读取，而第 13 行到 20 行则是分别处理数据为空、数据不可读以及读取操作出错的情况。

```
1  ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
2  {
3      int n, rc;
4      char c, *bufp = usrbuf;
5
6      for (n = 1; n < maxlen; n++) {
7          if ((rc = rio_read(rp, &c, 1)) == 1) {
8              *bufp++ = c;
9              if (c == '\n') {
10                 n++;
11                 break;
12             }
13         } else if (rc == 0) {
14             if (n == 1)
15                 return 0;
16             else
17                 break;
18         } else{
19             return -1;
20         }
21     }
22     *bufp = 0;
23     return n-1;
24 }                                                    // ./rIO_pkg.c
```

图 3. 7 rio_readlineb 函数

Robust I/O 包实现读取操作的方式有很多，但是写入的方式就只有 rio_writen 函数了，它的任务是将 usrbuf 位置的 n 个字节传送到已连接描述符 fd。和 Robust I/O 其他函数实现一样，如图 3.8 所示，rio_writen 函数的实现也是基于 Linux 内核引出的写入接口 write 函数。第 7 到 13 行就是通过循环地调用 write 函数来完成写入操作。在第 9 行中设置了一个中断处理，当该线程在执行写入操作时被中断，函数就会将 nwritten 参数置为 0 来手动重启写入操作，这种处理方式提高了 Web 服务器在执行读写操作是的可靠性也一定程度上提到了 Web 服务器运行的稳定性。

```c
ssize_t rio_writen(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nwritten;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nwritten = write(fd, bufp, nleft)) <= 0) {
            if (errno == EINTR)
                nwritten = 0;
            else
                return -1;
        }
        nleft -= nwritten;
        bufp += nwritten;
    }
    return n;
}                                                    // ./rIO_pkg.c
```

图 3.8 rio_writen 函数

在第二章介绍过内存映射在本文的 Web 服务器构建中也发挥着重要作用，例如静态内容服务中需要用内存映射来构建响应主体提高资源处理效率；在使用 CGI 程序实现动态内容服务时，创建新子进程运行外部程序需要用到内存映射从父进程获取资源，在调用 execve 函数加载并执行 CGI 程序时也需要使用内存映射为其创建虚拟内存空间。内存映射 Linux 也有实现，在 sys/mman.h 库中实现了用于要求内核创建虚拟内存区域的 mmap 函数和删除虚拟内存区域的 munmap 函数。和上述处理方法一样，本文在调用这对函数的时候加入了是否调用成功的判断，当调用出错是打印错误信息提醒用户错误位置。

```c
void *Mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
void Munmap(void *start, size_t length);
                                                     // ./general.h
```

图 3.9 用户级内存映射函数封装

3.3 Web 服务实现

有了这些由 Linux 内核实现的网络通信和资源操作基础实现与改进，使得实现一个能够提供静态内容和动态内容的 Web 服务器成为可能。本节就是在前面介绍的套接字接口、Unix I/O 和内存映射，以及未介绍的进程控制的 Linux 基础实现与改进的基础上，根据 HTTP 协议的工作原理让 Web 服务器能够提供 Web 服务。在本节中主要涉及三部分内容，分别是作为 Web 服务基础的 HTTP 事务处理实现、Web 服务器基本功能静态内容服务和重要扩展基于 CGI 标准的动态内容服务。

3.3.1 HTTP 请求处理

HTTP 协议是请求响应模型的，而一个 HTTP 事务也是由 HTTP 请求和 HTTP 响应组成，在 2.2.2 节已经详细介绍了 HTTP 请求和 HTTP 响应的构成以及每个组成部分的含义。作为服务端进程要能够接收并分析 HTTP 请求，同时能够构建完整的 HTTP 响应。HTTP 响应是在提供具体服务时将资源操作结果作为响应主体发送给客户端的，所以这部分工作在提供动静态内容服务时完成。在本节中主要介绍 Web 服务器对客户端发来的 HTTP 请求读取并分析的过程。在处理 HTTP 请求的读取和分析两个过程中，请求内容读取由图 3.10 所示的 read_request_header 函数完成，第 6 行到第 14 行函数在不断循环读取请求行直到由回车和换行符`\r\n`构成的空行请求报头终止标志结束读取。请求内容分析则是由图 3.11 所示的

```c
int read_request_header(rio_t *rp, char *method)
{
    char buf[MAXLINE];
    int len = 0;

    do{
        Rio_readlineb(rp, buf, MAXLINE);
        printf("%s", buf);

        if(strcasecmp(method, "POST") == 0 && strncasecmp(buf, "Content-Length:", 15)
== 0)
        {
            sscanf(buf, "Content-Length: %d", &len);
        }
    }while(strcmp(buf, "\r\n"));
    return len;
}                                                                    // ./service.c
```

图 3. 10 read_request_header 函数

parsing_uri 函数完成的，Web 服务器使用该函数来解析请求行中的 uri，并根据 uri 所指定的资源位置来确定相应的服务类型。在本文的文件目录组织结构中./src/目录下存放的是静态资源而./bin/目录下则存放着提供动态内容服务的 CGI 程序。在第 5 行到第 12 行中，根据 uri 没有指向 bin 目录初步判断为静态内容服务，并将文件请求指向路径 src 目录，同时设置默认的静态资源访问为 index.html 文件。在第 16 行到第 27 行则是将请求判断为请求动态内容服务后，截取 uri 中的参数序列保存在 cgiargs 中，并通过 filename 传送请求的资源路径。

```c
int parsing_uri(char *uri, char *filename, char *cgiargs)
{
    char *ptr;
    /* 静态内容服务 */
    if (!strstr(uri, "bin")){
        strcpy(cgiargs, ""); // 清除CGI参数字符串
        strcpy(filename, "./src"); // 将URI转换为一个指向静态内容资源的相对路径
        strcat(filename, uri);
        if(uri[strlen(uri)-1] == '/'){
            strcat(filename, "index.html"); // 默认请求index.html文件
        }
        return 1;
    }else{
    /* 动态内容服务 */
        /* 截取所有CGI参数 */
        ptr = index(uri, '?');
        if (ptr) {
            strcpy(cgiargs, ptr+1);
            *ptr = '\0';
        }else{
            strcpy(cgiargs, "");
        }

        /* 将URI剩余部分转换为对应资源的相对路径 */
        strcpy(filename, ".");
        strcat(filename, uri);
        return 0;
    }
}                                                                    // ./service.c
```

图 3. 11 parsing_uri 函数

整体的 HTTP 请求处理过程是在图 3.12 所示的 handle_http_tans 函数中完成的。首先在第 8 行到第 10 行和第 16 到 17 行中完成请求行的读取，然后在第 19 行调用 parsing_uri 函数解析请求行中的 uri 确定目标文件的路径和服务类型，最后在第 25 行到 42 行完成根据不同请求方法和目标文件路径的动静态内容服务提供。在这个处理的过程中也需要检查请求内

```
1  void handle_http_trans(int fd)
2  {
3      char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE],
   filename[MAXLINE], cgiargs[MAXLINE];
4      int static_req;
5      struct stat sbuf; //文件描述信息表项
6      rio_t rio;        //Robust I/O缓冲区描述
7      /* 读请求报头 */
8      Rio_readinitb(&rio, fd);
9      Rio_readlineb(&rio, buf, MAXLINE);
10     sscanf(buf , "%s %s %s", method, uri, version);
11     if (!(strcasecmp(method, "GET") == 0 || strcasecmp(method, "HEAD") == 0 ||
   strcasecmp(method, "POST") == 0))
12     {
13         report_error(fd, method, "501", "Not Implemented", "This WebSever doesn't
   implement this method"); //请求方法错误报告
14         return;
15     }
16     int param_len = read_request_header(&rio, method);
17     Rio_readnb(&rio, buf, param_len);
18     /* 解析请求报头和请求行中的URI */
19     static_req = parsing_uri(uri, filename, cgiargs);
20     if (stat(filename, &sbuf) < 0) {
21         report_error(fd, filename, "404", "Not found", "This WebServer couldn't find
   this file"); // 文件不存在报错
22         return;
23     }
24     /* 静态内容服务 */
25     if(static_req){
26         if(!(S_ISREG(sbuf.st_mode)) || !(S_IRUSR & sbuf.st_mode)){
27             report_error(fd, filename, "403", "Forbidden", "This WebServer couldn't
   read the file"); // 文件无读取权限报错
28             return ;
29         }
30         static_content_service(fd, filename, sbuf.st_size, method);
31     }else{
32     /* 动态内容服务 */
33         if(!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode)){
34             report_error(fd, filename, "403", "Forbidden", "This WebServer couldn't
   run the CGI program"); // 无法执行动态服务程序报错
35             return ;
36         }
37         if(strcasecmp(method, "GET") == 0 || strcasecmp(method, "HEAD") == 0)
38         {
39             dynamic_content_service(fd, filename, cgiargs, method);
40         }else{
41             dynamic_content_service(fd, filename, buf, method);
42         }
43     }
44 }                                                          // ./service.c
```

图 3. 12 handle_http_trans 函数

容中各种信息的合法性。例如，在第 11 行到 15 行检查请求方法的合法性，本文实现的 Web 服务器目前只支持 GET、POST 和 HEAD 三种请求方法；在第 20 行到 23 行检查目标文件路径的合法性，如果目标文件不存在则返回 404 状态的“未发现”HTTP 响应；在第 26 行到第 29 行和第 33 行到第 36 行则是对文件访问权限的检验，在第二章介绍过 S_IRUSR 和 S_IXUSR 表示文件的可读权限和可执行权限，如果服务器进程没有目标文件的读写或执行权限则返回 403 状态的“禁止”HTTP 响应。

3.3.2 静态内容服务

如果 handle_http_trans 函数在处理 HTTP 请求过程中调用的是 static_contetn_service 函数，那么 Web 客户端请求的是静态内容服务。作为提供 Web 服务的函数其功能是为客户端发送一个对应的 HTTP 响应，如图 3.13 所示 static_contetn_service 函数完成了一个 HTTP 响应的构建和发送。在第 8 行到第 13 行完成的是一个响应报头和响应行的构建，它包含了表示服务器名称的 Server 字段、表示 HTTP 响应主体中内容的 MIME 类型信息的 Content-Type 字段和表示响应主体字节大小的 Content-Length 字段等其他信息。然后在第 14 行使用 Rio_writen 函数将响应报头发送出去。而在第 23 行到第 27 行完成的是一个响应主体的构建与发送，在第 23 行中根据 filename 文件路径打开对应的目标文件并在 24 行使用内存映射的方法将文件内容映射到一个虚拟内存空间，在第 26 行完成目标请求文件的发送。

```
1  void static_content_service(int fd, char *filename, int filesize, char *method)
2  {
3      int srcfd;
4      char *srcp;
5      char filetype[MAXLINE], buf[MAXLINE];
6
7      /* 发送响应行和响应报头 */
8      get_filetype(filename, filetype);
9      sprintf(buf, "HTTP/1.0 200 OK!\r\n");
10     sprintf(buf, "%s Server: Wanghaihua's Web Server\r\n", buf);
11     sprintf(buf, "%s Connection: close!\r\n", buf);
12     sprintf(buf, "%s Content-length: %d\r\n", buf, filesize);
13     sprintf(buf, "%s Content-type: %s\r\n\r\n", buf, filetype); //响应报头以一个空行标识结
   尾
14     Rio_writen(fd, buf, strlen(buf));
15     printf("Response headers:\n");
16     printf("%s", buf);
17
18     /* 判断是否是HEAD方法 */
19     if (strcasecmp(method, "HEAD") == 0)
20         return;
21
22     /* 发送包含一个本地文件内容的响应主体给客户端 */
23     srcfd = Open(filename, O_RDONLY, 0);
24     srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0); // 将被请求文件映射到一个
   虚拟内存空间
25     Close(srcfd);
26     Rio_writen(fd, srcp, filesize);
27     Munmap(srcp, filesize); //关闭内存映射
28 }                                                          // ./service.c
```

图 3.13 static_content_service 函数

前面提到在构建 HTTP 响应报头时需要发送响应主体中响应内容的 MIME 类型信息，在第二章中已经介绍过响应内容 MIME 类型信息对浏览器解析对应内容起着关键的作用，所以一个能够提供多媒体静态资源的 Web 服务器是不能忽略这一点的。在图 3.13 中的第 8 行调用了 get_filetype 函数来获取对应文件的 MIME 类型信息，而它的实现如图 3.14 所示，

通过目标文件的后缀，将文件类型转化为 MIME 类型所表示的内容，第 3 行到第 12 行分别表示了 HTML 文件、GIF 图像、JPG 和 PNG 图像以及 MPG 视频等类型文件转化为 MIME 类型的情况。

```
1  void get_filetype(char *filename, char *filetype)
2  {
3      if (strstr(filename, ".html"))
4          strcpy(filetype, "text/html");  //HTML
5      else if (strstr(filename, ".gif"))
6          strcpy(filetype, "image/gif");  //GIF
7      else if (strstr(filename, ".jpg"))
8          strcpy(filetype, "image/jpeg"); //JPG
9      else if (strstr(filename, ".png"))
10         strcpy(filetype, "image/png");  //PNG
11     else if (strstr(filename, ".mpeg"))
12         strcpy(filetype, "video/mpeg"); //MPG
13     else
14         strcpy(filetype, "text/plain");
15 }                                                    // ./service.c
```

图 3. 14 get_filetype 函数

3.3.3 动态内容服务

如果 handle_http_trans 函数在处理 HTTP 请求过程中调用的是 dynamic_contetn_service 函数，那么 Web 客户端请求的是动态内容服务。和 static_content_sericve 函数一样，dynamic_contetn_service 也需要为客户端发送一个对应的 HTTP 响应，但是与前者不同的是如图 3.15 所示它部分响应报头和响应主体的形成不是在该主进程中完成的。在第 7 行到第 10 行中，它只完成了响应行和表示服务器名称的响应报头 Server 字段的构建与发送。在第 13 行到第 19 行动态内容服务函数派生一个子进程来运行 CGI 程序，第 15 行和第 16 行分别根据 HTTP 请求处理得到的 CGI 参数序列 cgiargs 和 HTTP 请求方法 method 为 CGI 定义的环境变量 QUERY_STRING 和 REQUEST_METHOD 赋值。第 17 行则是在第 18 行调用 Execve 函数根据 filename 加载并执行 CGI 程序前将输出重定向到已连接描述符。

```
1  void dynamic_content_service(int fd, char *filename, char *cgiargs, char *method)
2  {
3      char buf[MAXLINE];
4      char *emptylist[] = {NULL};
5
6      /* 发送响应行 */
7      sprintf(buf, "HTTP/1.0 200 OK!\r\n");
8      Rio_writen(fd, buf, strlen(buf));
9      sprintf(buf, "Server: Wanghaihua's Web Server\r\n");
10     Rio_writen(fd, buf, strlen(buf));
11
12     /* 派生子进程执行CGI程序 */
13     if( Fork() == 0 ){
14         /* 设置CGI环境变量 */
15         setenv("QUERY_STRING",cgiargs, 1);
16         setenv("REQUEST_METHOD", method, 1);
17         Dup2(fd, STDOUT_FILENO);           //将输出重定向到已连接的文件描述符
18         Execve(filename, emptylist, __environ); // 加载并运行CGI程序
19     }
20     Wait(NULL);  //等待子进程结束并回收资源
21 }                                                    // ./service.c
```

图 3. 15 dynamic_content_service 函数

那么动态内容服务的响应主体和另外一些响应报头是怎么构建的呢？dynamic_contetn_service 将这个任务交给了在它子进程中运行的 CGI 程序。如图 3.16 所示，CGI 程序在第 7 行到第 13 行调用 getenv 函数分别通过 CGI 环境变量 QUERY_STRING 和 REQUEST_METHOD 获得了 HTTP 请求中的参数以及 HTTP 请求方法。第 14 行到第 18 行 CGI 程序在根据程序执行结果构建响应主体，而在第 20 行和第 21 行 CGI 程序构建了主进程中没有构建的 Content-Length 和 Content-Type 响应报头字段。在完成了所有需要的响应内容的构建之后 CGI 程序直接根据主进程中重定向的已连接描述符将这些内容发送出去。

```
1  int main(void) {
2      char *buf, *p;
3      char arg1[MAXLINE], arg2[MAXLINE], content[MAXLINE];
4      int n1=0, n2=0;
5      char *method;
6      /* 根据CGI环境变量读取参数 */
7      if ((buf = getenv("QUERY_STRING")) != NULL) {
8          p = strchr(buf, '&');
9          *p = '\0';
10         sscanf(buf, "first = %d", &n1);
11         sscanf(p+1, "second = %d", &n2);
12     }
13     method = getenv("REQUEST_METHOD");
14     /* 创建HTTP响应主体 */
15     sprintf(content, "Welcome to WangHaihua's WebServer: ");
16     sprintf(content, "%sThese's the dynamic content service based on CGI\r\n<p>",
   content);
17     sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>", content, n1, n2, n1 +
   n2);
18     sprintf(content, "%sThanks for visiting!\r\n", content);
19     /* 形成HTTP响应 */
20     printf("Content-length: %d\r\n", (int)strlen(content));
21     printf("Content-type: text/html\r\n\r\n");
22     if(strcasecmp(method, "HEAD") !=0 )
23         printf("%s", content);
24     fflush(stdout);
25     exit(0);
26 }                                                          // ./bin/adder.c
```

图 3.16 加法 CGI 程序 adder.c

3.4 基于 I/O 多路复用的并发实现

至此，Web 服务器已经实现提供 Web 服务的功能，它不仅可以为 Web 客户端提供静态内容服务还可以为其提供动态内容服务。接下来就是使用 I/O 多路复用技术让这个 Web 服务器具备抗并发能力了，在 2.4.2 节中已经详细介绍了使用 select 函数和一组宏 FD_ZERO、

```
1  typedef struct{
2      int maxfd;                  // Select的最大文件描述符
3      fd_set read_set;            // 活动客户读的描述符
4      fd_set ready_set;           // 读就绪描述符设置
5      int nready;                 // 读就绪描述符的个数
6      int maxi;                   // clientfd数组的最大索引
7      int clientfd[FD_SETSIZE];   // 活动客户端描述符集合
8      rio_t clientrio[FD_SETSIZE];// 读缓冲区
9  } pool;                                                 // ./general.h
```

图 3.17 已连接描述符池 pool 结构体

FD_SET、FD_ISSET 管理描述符集合来实现 I/O 多路复用技术的过程。为了管理访问 Web 服务器的多个客户端，需要声明一个如图 3.17 所示的 pool 结构体来维护连接服务器的多个已连接描述符，pool 结构体的各个属性及其含义已经由图中的注释说明。图 3.18 的 init_pool 函数是用来初始化已连接描述符池 pool 结构体的编程实现，第 5 行到第 7 行是用-1 表示客户端槽位可用将所有槽位初始化，第 10 到 12 行则是将 Web 服务器的监听描述符 listenfd 加入到 select 读集合中。

```
1  void init_pool(int listenfd, pool *pl)
2  {
3      /* 初始化客户端可连接槽位 用-1表示可用 */
4      int i;
5      pl -> maxi = -1;
6      for(i = 0; i < FD_SETSIZE; i++)
7          pl->clientfd[i] = -1;
8
9      /* 监听描述符listenfd作为select读集合中唯一描述符 */
10     pl->maxfd = listenfd;
11     FD_ZERO(&pl->read_set);
12     FD_SET(listenfd, &pl->read_set);
13 }                                                    // ./concurrent.c
```

图 3.18 init_pool 函数

每当由新的客户端访问 Web 服务器，都需要将它加入到客户端池 pool 中便于维护和并发控制，图 3.19 是将一个新的客户端添加到 pool 中的编程实现。第 6 行的循环中是在为新的客户端找空槽位，如果找到空槽位在第 10 行和第 11 行将新客户端的套接字描述符加入到客户端队列中并为其创建一个读缓冲区。在第 12 行中将这个新的客户端的描述符添加到

```
1  void add_client(int connfd, pool *pl)
2  {
3      int i;
4      pl->nready--;
5      /* 寻找可用的活动客户端槽位 */
6      for(i = 0; i < FD_SETSIZE; i++)
7      {
8          if(pl->clientfd[i] < 0){
9              /* 添加一个已经建立连接的套接字描述符到活动客户端池中 */
10             pl->clientfd[i] = connfd;
11             Rio_readinitb(&pl->clientrio[i], connfd);
12             FD_SET(connfd, &pl->read_set);
13
14             /* 修改栈顶的描述符指向 */
15             if(connfd > pl->maxfd)
16                 pl->maxfd = connfd;
17             if(i > pl->maxi)
18                 pl->maxi = i;
19             break;
20         }
21     }
22
23     /* 找不到空槽位 */
24     if(i == FD_SETSIZE){
25         app_error("add_client error: Too many clients");
26     }
27 }                                                    // ./concurrent.c
```

图 3.19 add_client 函数

select 读集合中，并在第 15 行到 18 行中根据新客户端的描述符修改 pool 的最大描述符 maxfd 和客户端队列最大索引 maxi。当然，如果客户端池已经满载，新的客户端将找不到空的槽位，在第 24 行到第 26 行调用定义在 err_handle_wra.c 中 app_error 函数报告添加客户端失败的错误报告。

图 3.20 中实现的 check_client 函数是用来回送来自每个准备好的已连接描述符的一个文本行。在第 7 行的循环语句就是在扫描客户端队列中准备好的已连接描述符，一旦该描述符就绪就开始第 13 行到第 18 行的请求处理和响应内容回送工作，在第 15 行调用 handle_http_trans 函数处理 Web 客户端的 HTTP 请求，并将 HTTP 请求的处理结果作为 HTTP 响应主体回送给客户端。当客户端请求完成或者客户端在另一端断开了连接，Web 服务器也将关闭这一端的连接，并在第 17 行和第 18 行中完成从客户端池 pool 中清除这一客户端的工作。

```
1  void check_clients(pool *pl)
2  {
3      int i, connfd, n;
4      char buf[MAXLINE];
5      rio_t rio;
6
7      for(i = 0; (i <= pl->maxi) && (pl->nready > 0); i++){
8          connfd = pl->clientfd[i];
9          rio = pl->clientrio[i];
10
11         /* 描述符读就绪，就开始回送文本行 */
12         if((connfd > 0) && (FD_ISSET(connfd, &pl->ready_set))){
13             pl->nready--;
14             /* 处理客户端请求 */
15             handle_http_trans(connfd);
16             Close(connfd);
17             FD_CLR(connfd, &pl->read_set);
18             pl->clientfd[i] = -1;
19         }
20     }
21 }                                                    // ./concurrent.c
```

图 3. 20 check_clients 函数

最后来到了基于 I/O 多路复用的并发 Web 服务器的启动函数如图 3.21 中实现的 main 函数。在第 8 行到第 12 行是在根据命令行参数确定 Web 服务器的临时端口号，在第 8 行的判断语句中检查命令行参数是否输入了临时端口号，如果没有将报告错误并终止进程，如果检查到足够的命令行参数，在第 12 行中将命令行的第一个参数作为 Web 服务器的临时端口号。第 14 行则是使用前面实现的套接字接口辅助函数 Open_listenfd 根据临时端口号创建一个 Web 服务器的监听套接字，第 15 行根据这一监听套接字使用 init_pool 函数初始化客户端池，然后就进入了 Web 服务器的无限迭代循环中。在循环中，第 19 行 Web 服务器进程首先要更新准备好的已连接描述符可读集合，然后在第 20 行调用 select 函数检测输入事件。在本文中 Web 服务器使用环境中，select 函数需要检测两种输入事件，一种是新的客户端发送来的连接请求，另一种则是已添加到客户端池 pool 中客户端的已连接描述符准备好了可以进行读取操作。如果是一个新的客户端发送来的连接请求，Web 服务器就要执行第 21 行到第 24 行的代码，当监听套接字描述符准备好可读时，调用 Accept 函数等待客户端的连接并调用 add_client 函数将新的客户端加入到客户端池中。最后在 check_clients 函数中完成客户端请求的处理并将每个并将每个准备好可读的已连接描述符中的内容回送出去。

```c
int main(int argc, const char *argv[])
{
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    static pool pool;

    /* 检查命令行参数，并将第1个参数作为端口号port */
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(1);
    }
    port = atoi(argv[1]);

    listenfd = Open_listenfd(port); //将输入port与服务器套接字地址建立联系，形成监听套接字
    init_pool(listenfd, &pool);

    /*无限服务器循环，不断地接收连接请求*/
    while (1) {
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.read_set, NULL, NULL, NULL);
        if(FD_ISSET(listenfd, &pool.ready_set)){
            clientlen = sizeof(clientaddr);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);//使用Accept()函数等待客户端连接
            add_client(connfd, &pool);
        }
        check_clients(&pool);
    }
}                                                                    // ./main.c
```

图 3.21 Web 服务器启动入口 main 函数

3.5 本章小结

本章介绍了基于 I/O 多路复用的并发 Web 服务器的实现。在第一节中从整体上介绍了 Web 服务器源代码的组织结构，说明了各个文件目录中包含文件的内容。在后续的三节中采用层层递进的方式介绍了 Web 服务器的实现过程，从网络通信和资源操作等基础内容的实现与改进开始；然后介绍了基于这些基础内容的 Web 服务器提供 Web 服务的实现；在最后讲解了怎么使用 select 函数实现 I/O 多路复用技术并以此完成 Web 服务器抗并发能力的扩展。