**多体系结构web服务器**

|  |  |
| --- | --- |
| **小 组 成 员** | |
| **202021090xx3** | **陈xx** |
| **202022090xx3** | **肖xx** |

目录

[**1.** **课题分析** 1](#_Toc58094282)

[**1.1.** **课题要求** 1](#_Toc58094283)

[**1.2.** **课题目标** 1](#_Toc58094284)

[**1.3.** **开发环境** 1](#_Toc58094285)

[**2.** **课题背景** 2](#_Toc58094286)

[**2.1.** **Web服务器** 2](#_Toc58094287)

[**2.2.** **基本原理** 2](#_Toc58094288)

[**3.** **课题设计** 5](#_Toc58094289)

[**3.1.** **总体设计** 5](#_Toc58094290)

[**3.2.** **主模块设计** 5](#_Toc58094291)

[**3.3.** **响应体设计** 7](#_Toc58094292)

[**3.4.** **网络处理模式设计** 8](#_Toc58094293)

[**3.4.1** **迭代结构** 8](#_Toc58094294)

[**3.4.2** **多进程结构** 9](#_Toc58094295)

[**3.4.3** **多线程结构** 11](#_Toc58094296)

[**3.4.4** **I/O复用结构** 12](#_Toc58094297)

[**3.4.5** **线程池结构** 14](#_Toc58094298)

1. **课题分析**
   1. **课题要求**

用迭代、多进程、多线程、I/O复用、线程池等多种技术设计并实现web服务器。

* 1. **课题目标**
* 实现迭代式web服务器
* 实现多进程式web服务器
* 实现多线程式web服务器
* 实现I/O复用式web服务器
* 实现线程池式web服务器
* 测试各个服务器的性能
  1. **开发环境**

根据网络程序的特点，分析其编程环境，最终确定了开发的环境:

* 操作系统：[Ubuntu16.04.2 + Windows 10(VM 虚拟机](https://blog.csdn.net/q1007729991/article/details/69224317))
* 处理器：Intel 酷睿i5 9400
* 内存：2.00G
* 开发语言：C/C++

1. **课题背景**
   1. **Web服务器**

Web服务器也称为WWW（World Wide Web）服务器，主要功能是提供网上信息浏览服务。由于用户在通过Web浏览器访问信息资源的过程中，无需再关心一些技术性的细节，而且界面非常友好，因而Web在Internet上一推出就受到了热烈的欢迎，走红全球，并迅速得到了爆炸性的发展。

目前Web服务器的服务标准已逐渐形成规范，实现技术也日趋成熟。由于Web服务器要同时为多个客户提供服务，就必须使用某种架构或技术来支持这种多任务的服务，包括使用多进程、多线程、线程池等等。

* 1. **基本原理**

Web服务器的工作原理一般可分成如下四个步骤：建立连接、请求过程、应答过程以及关闭连接。

建立连接：客户机通过TCP/IP协议建立到服务器的TCP连接；请求过程：客户端向服务器发送HTTP协议请求包，请求服务器里的资源文档；应答过程：服务器向客户机发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理得到的数据返回给客户端，由客户端解释HTML文档，在客户端屏幕上渲染图形结果；关闭连接：客户机与服务器断开。其流程如图所示：

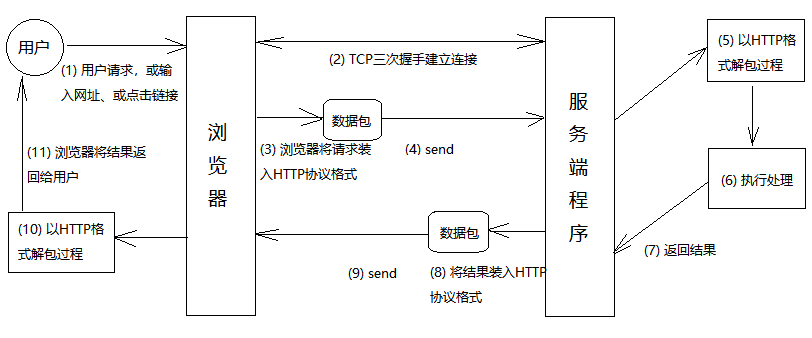


图2.1 web服务器工作原理

其底层TCP的socket通信流程大致为：

* **服务端**

1）socket----创建socket对象。

2）bind----绑定本机ip+port。

3）listen----监听来电，若在监听到来电，则建立起连接。

4）accept----再创建一个socket对象给其收发消息。原因是现实中服务端都是面对多个客户端，那么为了区分各个客户端，则每个客户端都需再分配一个socket对象进行收发消息。

5）read、write----就是收发消息了。

* **客户端**

1）socket----创建socket对象。

2）connect----根据服务端ip+port，发起连接请求。

3）write、read----建立连接后，就可发收消息了。

其流程如图所示：

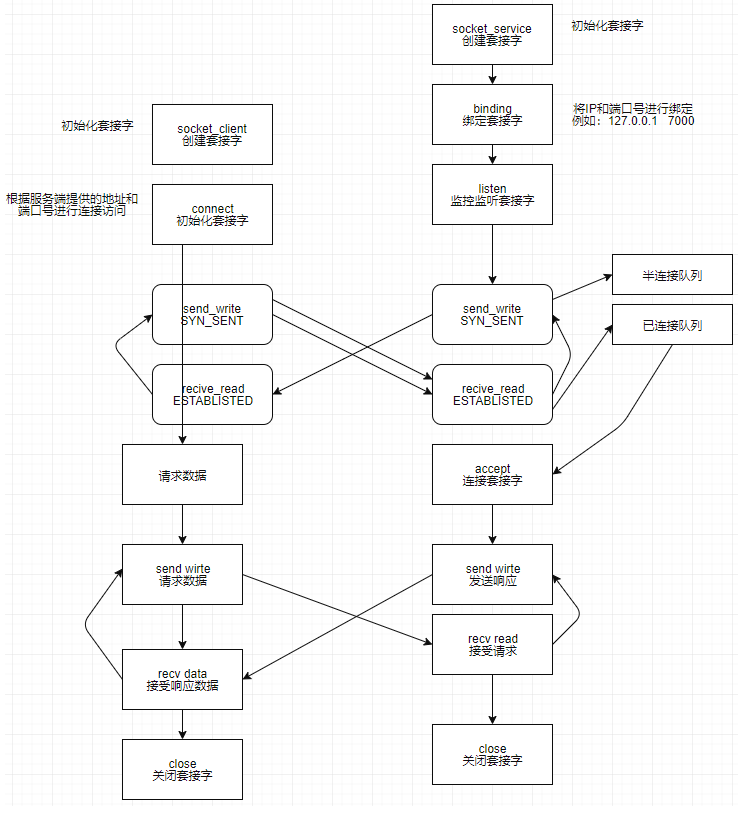


图2.2 TCP socket执行流程

1. **课题设计**
   1. **总体设计**

经分析，需要使用各种技术设计web服务器，对于不同的技术设计服务器，提取重复的地方：创建listen套接字并开始监听，将这个模块用一个文件实现。

另外，各种体系结构都需要对Http请求响应，提取出响应模块，此处需要包含主响应函数，以及接收到各种不同类型请求的响应函数。

最后，针对各种体系结构的web服务器，对其独特的部分，采用相应的技术分别实现，最后在main函数中分别调用各自的主函数即可。其系统模块图3.1所示：

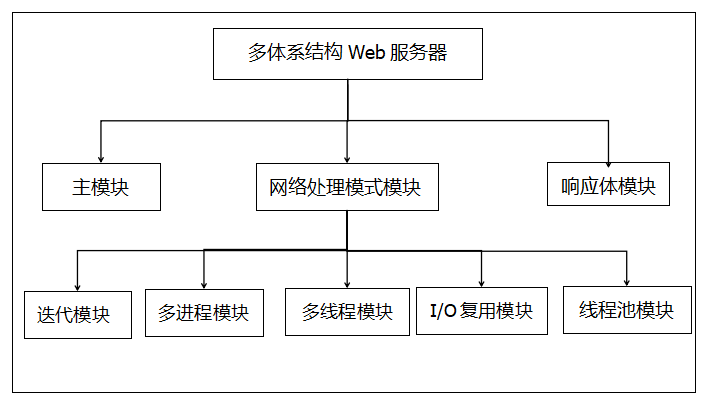


图3.1系统模块图

* 1. **主模块设计**

各main函数存在于main.cpp中。如图3.2所示，main函数首先创建listen套接字并开启监听，然后调用各个体系结构的loop函数，以达到用各个结构实现web服务器的目的。若选择好了要使用某种结构来实现web服务器，那么需要在main函数中包含该种结构实现的.h文件，并且在编译时将该种结构实现的.c文件与main.cpp和response.cpp一起编译。最终生成对应结构的web\_server可执行文件。

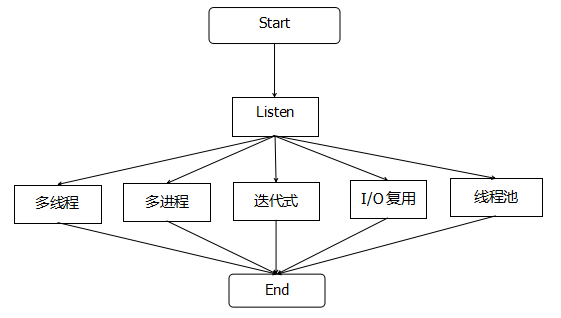


图3.2 主模块

核心代码如下：

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

{

int sock, fd;

sock = make\_server\_socket(5001);

if ( sock == -1 ) exit(2);

loop(sock);

return 0;

}

* 1. **响应体设计**

response函数存在于response.cpp中，response.cpp负责处理客户端请求（GET操作），所有工作都封装在response函数中，其他模块只需调用该函数即可处理请求。如图3.3所示，response函数首先打开通过参数传入的连接套接字，然后读取套接字中的内容，获取http请求，然后创建一个子进程进行响应，父进程则等待子进程结束。

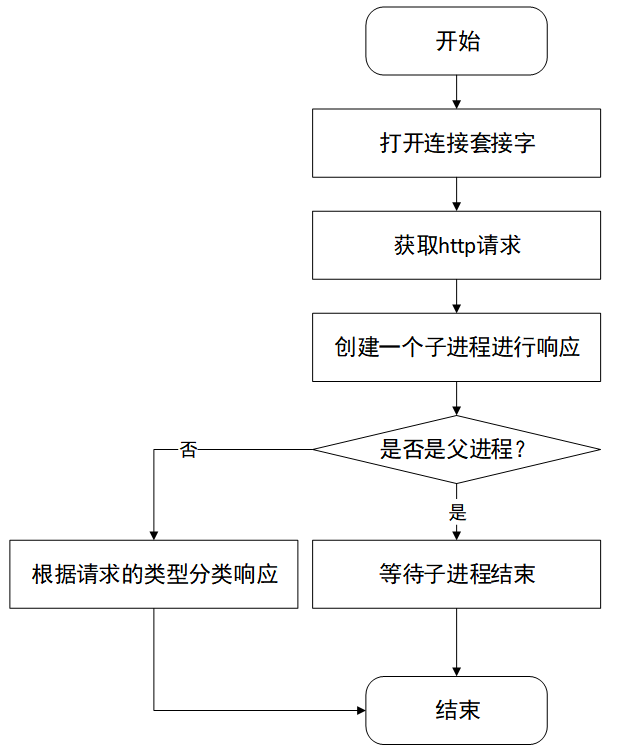


图3.3 响应函数流程图

核心代码如下：

void response(int fd)

{

char request[BUFSIZ];

pid\_t fpid;

int status;

FILE\* fpin;

if((fpin = fdopen(fd, "r" )) == NULL){

printf("Error while open fd : %d\n", fd, errno);

return;

}

/\* 读取 \*/

fgets(request,BUFSIZ,fpin);

printf("got a call: request = %s", request);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

read\_til\_crnl(fpin);

/\*等待应答 \*/

fpid = process\_rq(request, fd);

fclose(fpin);

waitpid(fpid, &status, 0);

}

* 1. **网络处理模式设计**

根据各个服务器体系结构的特点，分别设计。

* + 1. **迭代结构**

迭代式服务器在同一时刻只能响应一个客户端的请求，服务器逐个进行处理，处理完一个连接后再处理下一个连接，属于串行处理方式。迭代式服务器通过在单线程内设置循环控制实现对多个客户端请求的逐一响应，一般在循环执行的服务器对预期的负载能提供足够的反应速度时常使用这种类型的服务器。其处理逻辑如图3.4所示。

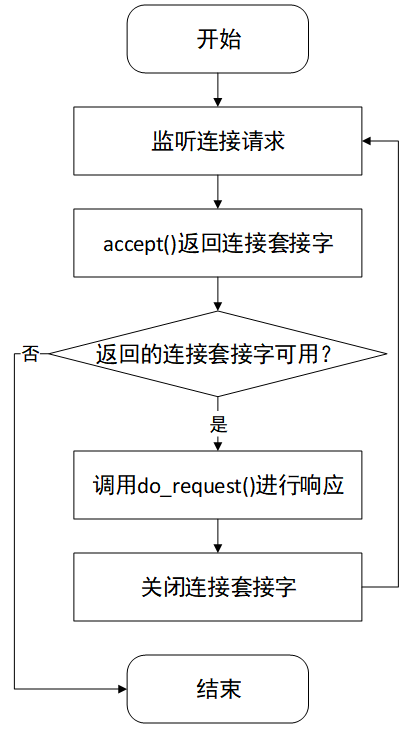


图3.4迭代结构处理流程图

核心代码如下:

while(1){

/\*接收请求 \*/

int fd = accept( socket, NULL, NULL );

if(fd == -1)

break;

response(fd);

close(fd);

}

* + 1. **多进程结构**

多进程并发服务器，即在父进程中接收客户端连接请求，然后创建一个新的子进程来为每个新客户端提供服务。多进程结构比迭代结构改进的点是通过创建子进程对客户端的连接进行响应，在响应过程中父进程不会被阻塞，而是会继续监听新的客户端发来的连接请求。其处理逻辑如图3.5所示。

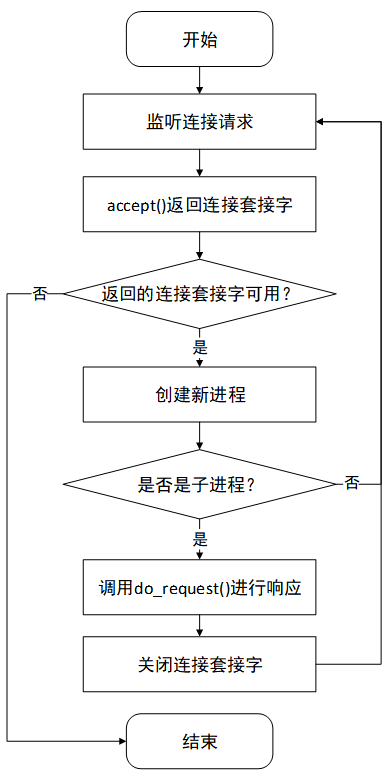


图3.5 多进程结构处理流程图

核心代码如下：

while(1){

int fd = accept( socket, NULL, NULL );

if(fd == -1)

break;

signal(SIGCHLD, SIG\_IGN);

pid\_t fpid = fork();

if(fpid < 0)

break;

else if(fpid == 0) {

printf("This is child process\n");

close(socket);

response(fd);

close(fd);

break;

}else{

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Created child process %d\n", fpid);

close(fd);

}}

* + 1. **多线程结构**

多线程结构在某些方面和多进程结构相似，每个进程开始生命周期时都是单一线程，这个线程称为主线程。在某一时刻，主线程创建一个子线程，从这时开始，两个线程并发运行。子线程执行完毕后会退出。其处理逻辑如图3.6所示。

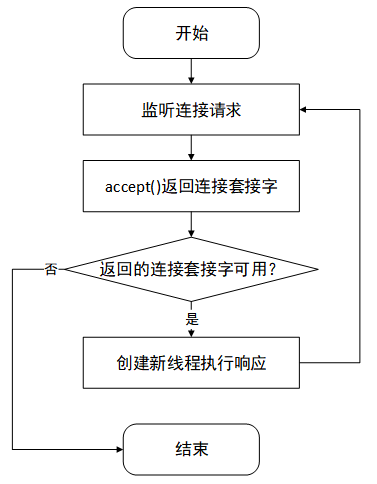


图3.6 多线程结构处理流程图

其核心代码如下：

int \*connfd;

pthread\_t t\_id;

while(1){

connfd = (int\*)malloc(sizeof(int));

/\* 接收连接请求 \*/

\*connfd = accept( socket, NULL, NULL );

if(\*connfd == -1)

break;

pthread\_create(&t\_id, NULL, thread, (void\*)connfd);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Created child thread %lu\n", t\_id);

}

* + 1. **I/O复用结构**

阻塞是指调用结果返回之前，当前线程会被挂起，一直处于等待消息通知，不能够执行其他业务；非阻塞是指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

* 阻塞式I/O

当用户进程调用了recvfrom等阻塞方法时，内核进入IO的第1个阶段：准备数据（内核需要等待足够的数据再拷贝）这个过程需要等待，用户进程会被阻塞，等内核将数据准备好，然后拷贝到用户地址空间，内核返回结果，用户进程才从阻塞态进入就绪态。

* 非阻塞式I/O

当用户进程发出read操作时，如果kernel中的数据还没有准备好，那么它并不会block用户进程，而是立刻返回一个error。用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作一旦kernel中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。非阻塞IO模式下用户进程需要不断地询问内核的数据准备好了没有，会一直占用cpu资源，非常的不实用。

主要使用linux的libevent库，通过event机制实现I/O复用。

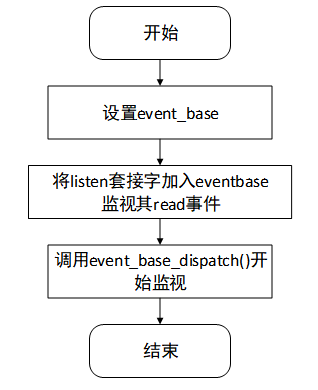


图3.7 I/O复用结构

如图3.7所示，在I/O复用结构的loop函数中，借助于libevent库强大的功能，需要做的事情只有将listen套接字加入event\_base，设定监视其读取事件（就是连接请求），然后开始循环监视即可。

其核心代码如下：

struct event listen\_ev;

base = event\_base\_new();

event\_set(&listen\_ev, socket, EV\_READ|EV\_PERSIST, on\_accept, NULL);

event\_base\_set(base, &listen\_ev);

event\_add(&listen\_ev, NULL);

event\_base\_dispatch(base);

* + 1. **线程池结构**

线程池:一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的[堆栈](https://baike.baidu.com/item/%E5%A0%86%E6%A0%88/1682032" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B%E6%B1%A0/_blank)大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在[托管代码](https://baike.baidu.com/item/%E6%89%98%E7%AE%A1%E4%BB%A3%E7%A0%81/2886980)中空闲（如正在等待某个事件）,则线程池将插入另一个[辅助线程](https://baike.baidu.com/item/%E8%BE%85%E5%8A%A9%E7%BA%BF%E7%A8%8B/4746601" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B%E6%B1%A0/_blank)来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。其原理如图3.8所示。

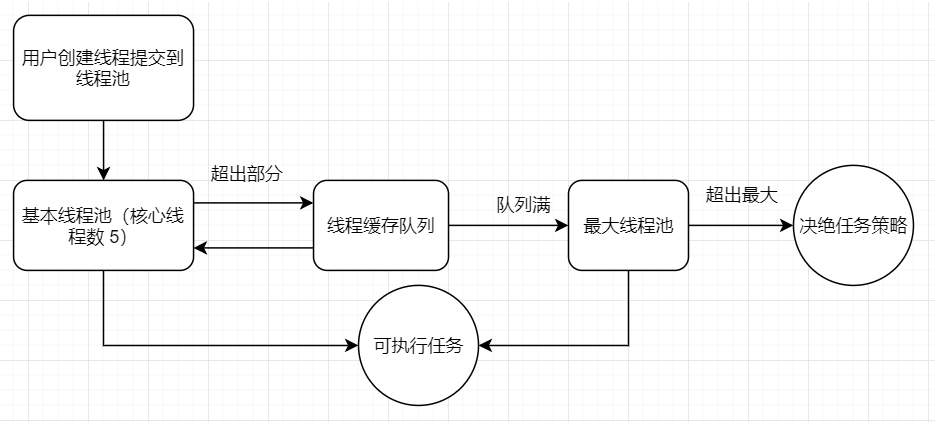


图3.8 线程池原理

根据其原理，设计为主线程进行accept，用信号量同步实现共享资源互斥。首先创建固定数量的线程，主线程将连接套接字放进缓冲区中，其它线程再从缓冲区中取出连接套接字并处理。其逻辑如图3.9所示。

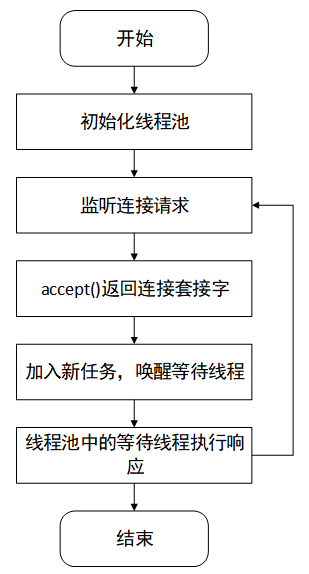


图3.9 线程池结构执行图

其核心代码如下：

pool\_init(5);

while(1){

/\* take a call and buffer it \*/

int fd = accept( socket, NULL, NULL );

if(fd == -1)

break;

pool\_add\_worker(fd);

}