**100071011 Computer Networks 2023-2024-2**

**Project-2**

**Building a CGI-Support Multi-Threaded Web Server**

**Specification**

|  |  |
| --- | --- |
| **学号 (Student ID)** |  |
| **姓名 (Name)** |  |
| **班号 (Class No.)** |  |
| **授课教师 (Instructor)** |  |

**School of Computer**

**Beijing Institute of Technology**

**Jun 16, 2024**

目录

[一、 Requirement Analysis 2](#_Toc171200902)

[1. 项目需求 2](#_Toc171200903)

[2. 基础知识 2](#_Toc171200904)

[2.1 Web服务器 2](#_Toc171200905)

[2.2 静态和动态页面 2](#_Toc171200906)

[2.3 URL 3](#_Toc171200907)

[2.4 CGI 4](#_Toc171200908)

[2.5 HTTP和TCP 4](#_Toc171200909)

[3. 主要问题 6](#_Toc171200910)

[二、 Design and Implementation 6](#_Toc171200911)

[1. 主要过程 6](#_Toc171200912)

[2. 文件组织： 7](#_Toc171200913)

[3. 线程池 7](#_Toc171200914)

[4. 主线程工作 8](#_Toc171200915)

[5. 解析HTTP请求 9](#_Toc171200916)

[6. 处理GET请求 11](#_Toc171200917)

[7. 处理CGI请求 12](#_Toc171200918)

[8. 返回响应给客户端 14](#_Toc171200919)

[9. CGI程序的编写 14](#_Toc171200920)

[三、 Development 15](#_Toc171200921)

[1. 开发环境 15](#_Toc171200922)

[2. 项目结构 16](#_Toc171200923)

[3. 总体流程 18](#_Toc171200924)

[四、 System Deployment, Startup, and Use 20](#_Toc171200925)

[五、 System Test 23](#_Toc171200926)

[1. GET请求默认页面 23](#_Toc171200927)

[2. GET请求指定页面 24](#_Toc171200928)

[3. 计算器CGI程序 26](#_Toc171200929)

[4. 数据库查询CGI程序 28](#_Toc171200930)

[5. 404页面 31](#_Toc171200931)

[6. 日志文件 32](#_Toc171200932)

[六、 Performance and Analysis 33](#_Toc171200933)

[1. 线程数为1 34](#_Toc171200934)

[2. 线程数为5 35](#_Toc171200935)

[3. 线程数为10 35](#_Toc171200936)

[4. 线程数为15 36](#_Toc171200937)

[5. 线程数为20 37](#_Toc171200938)

[6. 线程数为25 37](#_Toc171200939)

[七、 Summary or Conclusions 38](#_Toc171200940)

[八、 References 39](#_Toc171200941)

[九、 Comments 39](#_Toc171200942)

1. Requirement Analysis
   1. 项目需求

* 编写多线程的Web服务器，支持并行处理同时多个服务请求。
* Web服务器支持CGI，从而支持静态和动态页面。
* Web服务器必须可以处理GET、POST和HEAD类型的客户端请求。
* Web服务器应当返回正确的状态码，包括200、400、403和404。当返回错误码时，响应体要包含一个简单的错误页面。
* Web服务器需要有日志记录，每一行表示一个请求，每一行包括客户端的IP地址、客户端计算机表示、访问者的登录ID、日期和时间、请求方式、请求文件的位置和名称、HTTP状态码、请求文件大小、发出此请求的网址。
  1. 基础知识
     1. Web服务器

Web服务器通常被实现为多线程应用，以方便利用硬件资源和并行处理多个请求。

* + 1. 静态和动态页面

静态页面是指内容在服务器端生成后，以HTML、CSS、JS形式直接发送给客户端（如Web浏览器）的页面，而不执行任何附加过程。这些页面的内容在发布到服务器后通常不会改变，除非手动更新文件并重新上传到服务器。

动态页面是指内容在客户端请求时由服务器端动态生成的页面。这些页面通常包含服务器端脚本（如PHP、Python、Node.js等），这些脚本根据用户请求、数据库内容或其他数据源来生成HTML代码，并将其发送给客户端。这比静态页面需要更多的加载时间，且不利于SEO。

* + 1. URL

URL（Uniform Resource Locator，统一资源定位符）是互联网上资源的唯一标识符，通过URL可以定位和访问互联网上的各种资源。URL一般由多个部分组成，包括协议、主机名（或IP地址）、端口号（可选）、路径、查询参数（可选）和片段标识符（可选）。具体格式如下：

* 协议：指定了访问资源所使用的协议，常见的有HTTP、HTTPS、FTP等。例如，http:// 表示使用HTTP协议。
* 主机名/IP地址：存放资源的服务器的域名系统（DNS）主机名或IP地址。例如，www.example.com。
* 端口号：可选部分，用于指定访问服务器上的特定服务。如果省略，则使用方案的默认端口，如HTTP的默认端口为80。
* 路径：由零或多个“/”符号隔开的字符串，一般用来表示主机上的一个目录或文件地址。例如，/path/to/resource。
* 查询参数：可选部分，用于给动态网页（如使用CGI、PHP等技术制作的网页）传递参数，多个参数用“&”符号隔开，每个参数的名和值用“=”符号隔开。例如，?key1=value1&key2=value2。
* 片段标识符：可选部分，用于指定网络资源中的片段。例如，#section1。
  + 1. CGI

通用网关接口（CGI，Common Gateway Interface）是Web服务器与程序之间动态生成数据的一种标准接口协议。它允许Web浏览器发送请求到Web服务器，并将用户输入与服务器端执行的程序进行交互。

CGI是Web服务器运行时外部程序的规范，通过CGI编写的程序可以扩展服务器功能。CGI应用程序能与浏览器进行交互，还可通过数据API与数据库服务器等外部数据源进行通信，从数据库服务器中获取数据，并格式化为HTML文档后发送给浏览器，或者将从浏览器获得的数据放到数据库中。

具有CGI文件扩展名的文件是通用网关接口脚本文件。这些脚本文件经常出现在服务器的“cig-bin”目录中。

CGI程序的所有输出必须在前面加一个MIME类型的头，这是HTTP头。而且大多数时候CGI程序的输出是HTML，其他情况可能输出图片或其他非HTML内容。

* + 1. HTTP和TCP

HTTP是一种应用层协议，设计用来在客户端和服务器之间传输超文本和其他类型的数据。HTTP是建立在TCP之上的，意味着HTTP使用TCP作为它的传输层协议。HTTP定义了客户端如何向服务器请求网页（或其他资源），以及服务器如何响应这些请求。主要特点如下：

* 无状态性：HTTP协议本身是无状态的，即服务器不保留任何两次请求之间客户端的信息。但可以通过使用Cookie或Session等技术来实现状态的保持。
* 请求/响应模型：HTTP使用请求/响应模型。客户端发起一个请求到服务器，服务器处理请求并返回响应给客户端。
* 支持多种数据格式：HTTP可以传输多种类型的数据，包括HTML、XML、JSON、纯文本、图片等。
* 灵活性和可扩展性：HTTP支持各种方法（如GET、POST、PUT、DELETE等）和头部信息，可以灵活地处理不同类型的请求和响应。

TCP是一种面向连接的、可靠的、基于字节流的传输层通信协议。它位于IP（Internet Protocol，互联网协议）之上，提供端到端的可靠传输服务。TCP通过确认和重传机制来确保数据的正确传输，并且提供了流量控制和拥塞控制等功能。主要特点如下：

* 面向连接：TCP通信前需要建立连接，数据传输完成后需要断开连接。这种连接是全双工的，即数据可以在两个方向上同时传输。
* 可靠性：TCP通过序列号、确认应答、超时重传等机制确保数据的可靠性传输。
* 流量控制：TCP连接的每一方都有固定大小的缓冲空间，TCP的流量控制机制允许TCP连接的发送方根据接收方的接收能力来控制数据的发送量，从而避免接收方因来不及处理而丢失数据。
* 拥塞控制：当网络拥塞时，TCP能够减少数据的发送，以缓解网络拥塞。
  1. 主要问题
* 多线程和线程池
* Socket编程
* CGI
* 输入输出重定向
* HTTP请求和响应格式

1. Design and Implementation
   1. 主要过程

Web服务器在“8888”端口上创建socket，主线程侦听该端口，然后主线程进入无限循环中。该循环中做的工作如下：

* 当接收到客户端的连接请求时，应该创建一个新线程，然后把socket传递给该线程。
* 如果达到了最大连接数，则缓存到队列中。
* 如果缓存队列满，则抛弃该连接请求。

使用线程池，每个线程都会被阻塞，直到有HTTP请求需要它处理。因此，如果工作线程比活动请求多，那么一些线程将被阻塞，等待新的HTTP请求到达；如果请求数超过工作线程，那么超过的请求需要缓存，直到有一个空闲的线程。

* 1. 文件组织：
* Web服务器根目录：/webroot
* CGI程序：/webroot/cgi-bin/
* 日志文件：/webroot/log/
* 静态页面：/webroot/
* 默认页面：/webroot/index.html
* 404页面：/webroot/404.html
  1. 线程池

这里对Java中提供的线程池进行了简单的封装。这里的拒绝策略可以改为替换。

public class ThreadPool {

private final ExecutorService executorService;

public ThreadPool(int maxPoolSize, int queueSize) {

// corePoolSize表示线程池预设线程数；拒绝策略表示超过缓存队列时直接丢弃

this.executorService = new ThreadPoolExecutor(Integer.min(10, maxPoolSize / 2), maxPoolSize, 60L, TimeUnit.SECONDS, new ArrayBlockingQueue<>(queueSize), new ThreadPoolExecutor.DiscardPolicy());

}

public void execute(Runnable task) {

this.executorService.execute(task);

}

public void shutdown() {

this.executorService.shutdown();

}

}

* 1. 主线程工作

主线程的工作为接收客户端的连接请求，然后将socket传递给线程池的线程。

public class WebServer {  
 private static ServerSocket *serverSocket*;  
 private static ThreadPool *threadPool*;  
  
 public static void start() {  
 try {  
 WebServer.*serverSocket* = new ServerSocket(Config.*port*);  
 WebServer.*threadPool* = new ThreadPool(Config.*maxConnectionNum*, Config.*maxCachedConnectionNum*);  
 System.*out*.println("[info] 服务器已启动");  
 while (true) {  
 Socket clientSocket = *serverSocket*.accept();  
 *threadPool*.execute(new WorkThread(clientSocket));  
 }  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

* 1. 解析HTTP请求

该方法位于WorkThread类中，其主要工作是通过“\r\n”来截取首行、请求头、空行，然后如果请求头中包含“Content-Length”字段，则截取请求体。

// 处理客户端发送过来的消息，处理成功返回true，失败返回false  
private boolean requestHandle(InputStream in) {  
 List<String> firstLineAndHeaders = new ArrayList<>();  
 byte[] body = new byte[0];  
 boolean encounterBlankLine = false;  
 try {  
 int lineIndex = 0;  
 int lineLength = 0;  
 while (!encounterBlankLine) {  
 int readLength = in.available();  
 byte[] temp = in.readNBytes(readLength);  
 for (int i = 0; i < readLength; ) {  
 if (temp[i] == (byte) '\r' && temp[i + 1] == (byte) '\n') { // 遇到了换行  
 lineLength = i - lineIndex;  
 if (lineLength == 0) { // 遇到了空行  
 body = Arrays.*copyOfRange*(temp, Integer.*min*(i + 2, readLength), readLength);  
 encounterBlankLine = true;  
 break;  
 } else {  
 firstLineAndHeaders.add(new String(temp, lineIndex, lineLength, StandardCharsets.*UTF\_8*));  
 }  
 lineIndex = i + 2;  
 i += 2;  
 } else {  
 i += 1;  
 }  
 }  
 }  
 } catch (IOException e) {  
 return false;  
 }  
  
 if (firstLineAndHeaders.isEmpty()) {  
 return false;  
 }  
 // 读取第一行  
 String[] firstLineComp = firstLineAndHeaders.getFirst().split(" ");  
 if (firstLineComp.length != 3) {  
 return false;  
 }  
 switch (firstLineComp[0]) {  
 case "GET":  
 this.requestMethod = REQUEST\_METHOD.*GET*;  
 break;  
 case "POST":  
 this.requestMethod = REQUEST\_METHOD.*POST*;  
 break;  
 case "HEAD":  
 this.requestMethod = REQUEST\_METHOD.*HEAD*;  
 break;  
 default:  
 return false;  
 }  
 // 去除片段标识符，并分开请求地址和参数  
 String[] urlComp = firstLineComp[1].split("#")[0].split("\\?");  
 this.requestUrl = urlComp[0];  
 this.requestParams = new HashMap<>();  
 // 解析请求参数  
 if (urlComp.length == 2) {  
 String[] params = urlComp[1].split("&");  
 for (String param : params) {  
 String[] keyAndValue = param.split("=");  
 requestParams.put(keyAndValue[0], keyAndValue.length == 2 ? keyAndValue[1] : "");  
 }  
 }  
  
 // 读取请求头  
 this.requestHeaders = new HashMap<>();  
 for (int i = 1; i < firstLineAndHeaders.size(); i++) {  
 String[] headerComp = firstLineAndHeaders.get(i).split(": ");  
 if (headerComp.length != 2) {  
 return false;  
 }  
 requestHeaders.put(headerComp[0], headerComp[1]);  
 }  
  
 // 读取请求体  
 if (requestHeaders.containsKey("Content-Length")) {  
 int counter = Integer.*parseInt*(requestHeaders.get("Content-Length"));  
 try {  
 this.requestBody = new byte[counter];  
 // 将array1复制到requestBody数组  
 System.*arraycopy*(body, 0, this.requestBody, 0, body.length);  
 // 将array2复制到requestBody数组的后面  
 System.*arraycopy*(in.readNBytes(counter - body.length), 0, this.requestBody, body.length, counter - body.length);  
 } catch (IOException e) {  
 return false;  
 }  
 }  
 return true;  
}

* 1. 处理GET请求

如果请求的URL为“/”则进行默认页面替换，否则检查被请求文件是否存在，如果不存在则返回404页面，否则返回被请求的文件，注意要设置响应头“Content-Type”为正确值，从而使浏览器正确解析返回的内容。

// GET请求处理  
private void getHandle() {  
 STATUS\_CODE code = STATUS\_CODE.*CODE\_200*; // 状态码  
 Map<String, String> header = new HashMap<>(); // 请求头  
 byte[] body; // 请求体  
  
 // 默认页面替换  
 String path = Objects.*equals*(requestUrl, "/") ? Path.*of*(Config.*rootPath*, Config.*defaultPage*).toString() : Path.*of*(Config.*rootPath*, requestUrl).toString();  
  
 File file = new File(path);  
 // 文件不存在则返回404页面  
 if (!(file.exists() && file.isFile() && file.canRead())) {  
 respond404Page();  
 return;  
 }  
  
 body = Utils.*fromFileToString*(path);  
 try {  
 // 获取文件类型，正确标识MIME类型  
 header.put("Content-Type", Files.*probeContentType*(Path.*of*(path)));  
 } catch (IOException e) {  
 System.*out*.println("[ERROR] 解析文件类型失败");  
 }  
 responseHandle(code, header, body);  
}

* 1. 处理CGI请求

首先检查被请求的程序是否存在，如果不存在则返回404页面，否则通过Java提供的ProcessBuilder执行python脚本。要设置环境变量“REQUEST\_METHOD”。如果请求方式为POST，要设置环境变量“Content-Length”和“Content-Type”，并将请求体写入进程的输出流中。注意这里要使用flush()方法确保内容被完全写入；如果请求方式为GET，要设置环境变量“QUERY\_STRING”。最后读取进程的输入流，并写入到服务器与客户端连接的Socket中。

private void cgiHandle() {  
 File file = new File(Path.*of*(Config.*rootPath*, requestUrl).toString());  
 // 文件不存在则返回404页面  
 if (!(file.exists() && file.isFile() && file.canRead())) {  
 respond404Page();  
 return;  
 }  
  
 ProcessBuilder processBuilder = new ProcessBuilder("python", Path.*of*(Config.*rootPath*, requestUrl).toString());  
 switch (requestMethod) {  
 case *GET* -> {  
 processBuilder.environment().put("REQUEST\_METHOD", "GET");  
 }  
 case *POST* -> {  
 processBuilder.environment().put("REQUEST\_METHOD", "POST");  
 if (requestHeaders.containsKey("Content-Length")) {  
 processBuilder.environment().put("Content-Length", requestHeaders.get("Content-Length"));  
 }  
 if (requestHeaders.containsKey("Content-Type")) {  
 processBuilder.environment().put("Content-Type", requestHeaders.get("Content-Type"));  
 }  
 }  
 case *HEAD* -> {  
 processBuilder.environment().put("REQUEST\_METHOD", "HEAD");  
 }  
 }  
 StringBuilder query = new StringBuilder();  
 int counter = requestParams.size();  
 for (Map.Entry<String, String> entry : requestParams.entrySet()) {  
 query.append(String.*format*("%s=%s", entry.getKey(), entry.getValue()));  
 if (counter > 1) {  
 query.append("&");  
 counter--;  
 }  
 }  
 processBuilder.environment().put("QUERY\_STRING", query.toString());  
  
 try {  
 Process process = processBuilder.start();  
 if (requestMethod == REQUEST\_METHOD.*POST* && requestBody != null) {  
 process.getOutputStream().write(requestBody);  
 process.getOutputStream().flush();  
 }  
 out.write(String.*format*("HTTP/1.0 %d %s\r\n", 200, "OK").getBytes(StandardCharsets.*UTF\_8*));  
 // 重定向进程的输出到服务器的输出  
 out.write(process.getInputStream().readAllBytes());  
 process.waitFor();  
 log(STATUS\_CODE.*CODE\_200*);  
 } catch (IOException | InterruptedException e) {  
 System.*out*.println("[ERROR] CGI程序运行错误");  
 }  
}

* 1. 返回响应给客户端

为了方便代码重用，这里对响应进行封装，方法签名如下：

// 响应处理  
private void responseHandle(STATUS\_CODE statusCode, Map<String, String> headers, byte[] body);

* 1. CGI程序的编写

这里使用Python语言进行了CGI程序的编写。每个CGI程序的流程类似：通过环境变量“REQUEST\_METHOD”来对POST请求和GET请求进行不同处理，通过print()方法向服务器发送内容。

sys.stdout = io.TextIOWrapper(sys.stdout.buffer, *encoding*="utf-8")

if os.environ.get("REQUEST\_METHOD") == "GET":

    print("Content-Type: text/html;charset=utf-8'")

    print()

    content = template.replace("{{}}", "无")

    print(content)

    sys.stdout.flush()

elif os.environ.get("REQUEST\_METHOD") == "POST":

    print("Content-Type: text/html;charset=UTF-8")

    print()

    content\_length = os.environ.get("Content-Length")

    request\_content = sys.stdin.read(int(content\_length))

    param\_map = {}

    params = request\_content.split("&")

    for param in params:

        param\_comp = param.split("=")

        param\_map[param\_comp[0]] = param\_comp[1] if len(param\_comp) > 1 else ""

    replace\_str = "{} {} {} = {}"

    op = param\_map["op"]

    num1 = param\_map["num1"]

    num2 = param\_map["num2"]

    res = None

    try:

        num1 = float(num1)

        num2 = float(num2)

        if op == "add":

            res = num1 + num2

            op = "+"

        elif op == "sub":

            res = num1 - num2

            op = "-"

        elif op == "mul":

            res = num1 \* num2

            op = "\*"

        elif op == "div":

            res = num1 / num2

            op = "/"

        content = template.replace("{{}}", replace\_str.format(num1, op, num2, res))

    except ValueError:

        content = template.replace(

            "{{}}",

            replace\_str.format(num1, op, num2, "ERROR"),

        )

    print(content)

    sys.stdout.flush()

1. Development
   1. 开发环境

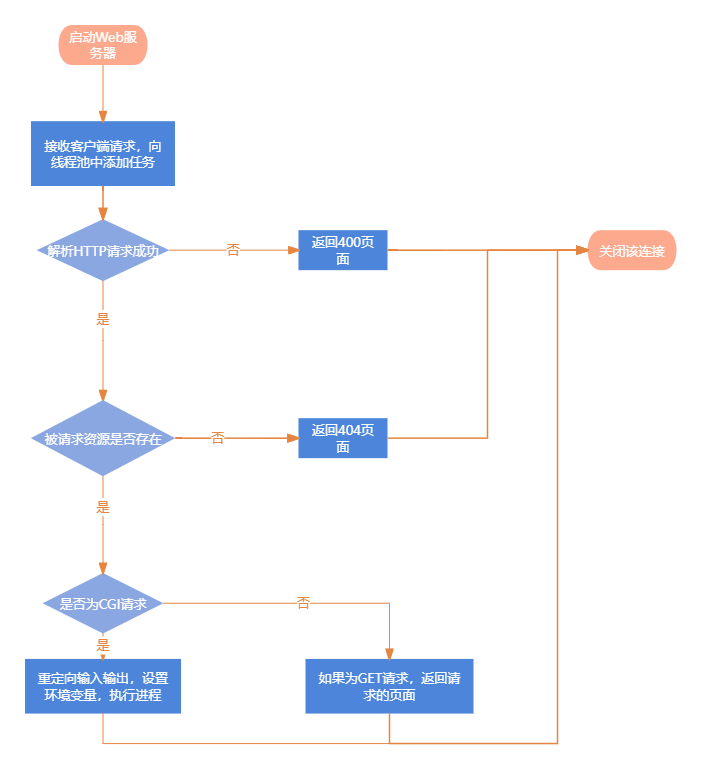
* 操作系统：Windows 11
* Web服务器编程语言：Java
* CGI程序编程语言：Python
* JDK：JDK 21
* 数据库：MySQL
* IDE：IntelliJ IDEA 2024.1.4 (Ultimate Edition)、VS Code 1.90.2
  1. 项目结构

本项目采用了面向对象的编写方式，具有清晰的项目结构和功能分类：

图形用户界面

描述已自动生成

* WebServer：接收客户端的Socket请求，并向线程池中添加任务。
* ThreadPool：线程池的简单封装。
* WorkThread：工作线程，项目中的主要类，负责处理客户端发过来的每个请求。
* Config：配置类，存放一些配置信息和常量。
* Utils：工具类，提供日志、文件转字节数组等方法。
* Main：入口类。
  1. 总体流程



启动Web服务器是整个流程的开始，服务器启动并等待客户端的连接和请求。一旦服务器启动并运行，它将开始监听来自客户端的HTTP请求。服务器接收到请求后，会将请求放入一个线程池中。线程池用于并发处理多个请求，以提高服务器处理请求的效率和吞吐量。线程池中的线程将负责解析接收到的HTTP请求。这一步是检查请求格式、参数和头信息等是否符合HTTP标准。

如果请求不符合格式则会返回400页（通常表示请求错误或不符合服务器要求）。如果请求的资源不存在，则服务器将返回404页（表示未找到资源）。

如果是CGI请求，服务器将执行相应的CGI脚本或程序，并需要重定向输入输出流（以便CGI程序可以读取输入并写入输出）和设置环境变量。如果请求是GET请求，服务器将确定请求的页面或资源，并准备将其发送给客户端。最后，服务器将请求的页面或资源发送给客户端，完成整个请求处理流程。

1. System Deployment, Startup, and Use

在编写完代码后，为了方便使用，修改配置类中的服务器根目录rootPath为“./webroot”。设置工件的主类，然后构建jar包。并将jar包和webroot目录放在同一目录下。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 应用程序

描述已自动生成

在命令行中输入：java -jar ./web-server.jar，然后Web服务器就成功启动，启动后控制台输出如下：

文本

AI 生成的内容可能不正确。

在浏览器中输入网址：<http://127.0.0.1:8888/>，可以看到首页正常显示，效果如下：

图形用户界面, 网站

AI 生成的内容可能不正确。

1. System Test

单元测试和集成测试在编程代码过程中已经完成，这里不再进行赘述。接下来展现系统测试效果。

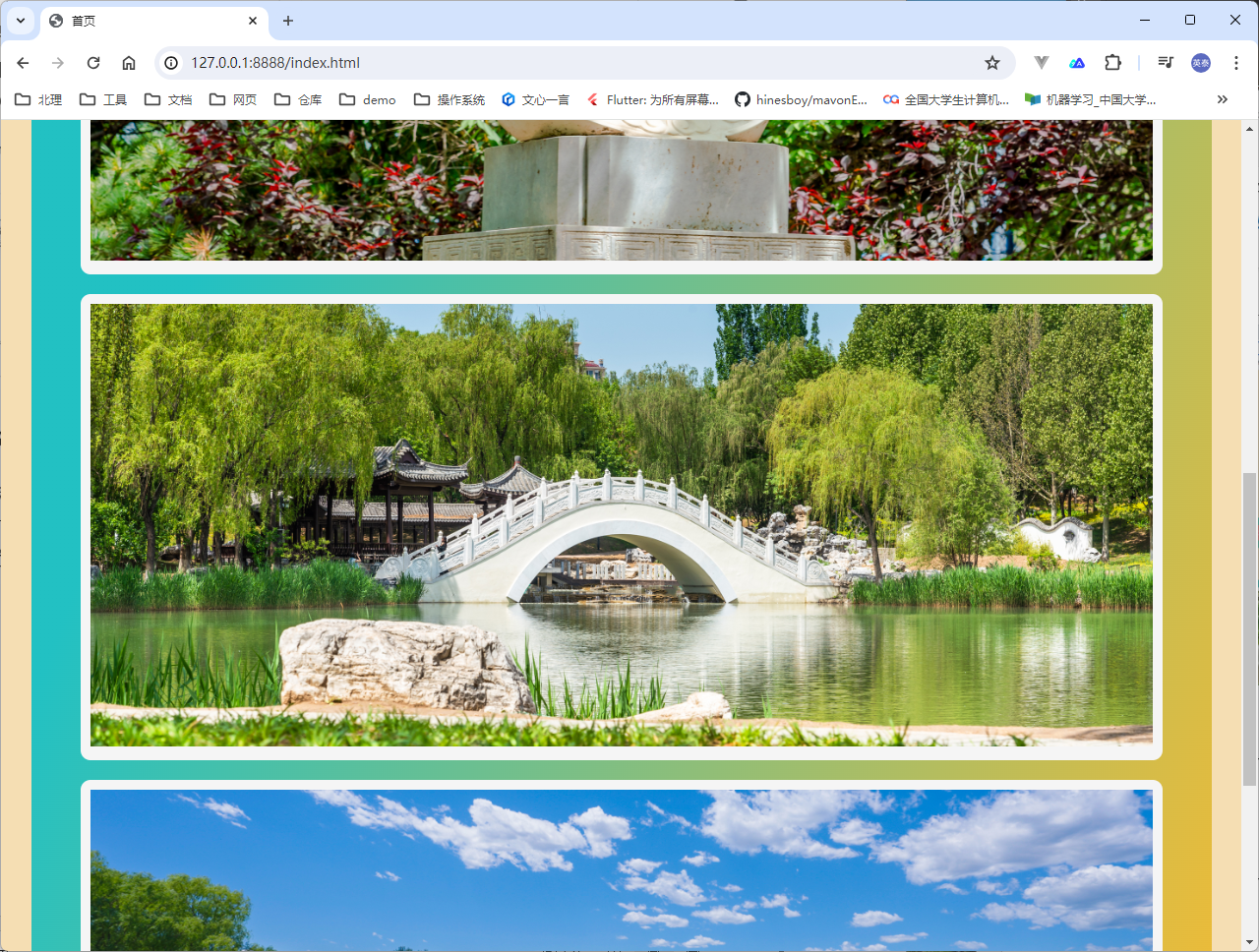
* 1. GET请求默认页面

在浏览器中输入网址：<http://127.0.0.1:8888/>，浏览器显示如下，可以看到该页面其实是index.html。



* 1. GET请求指定页面

在浏览器中输入网址：http://127.0.0.1:8888/index.html，浏览器显示如下，可以看到该服务器不仅可以处理HTML文本文件，对于图片文件传输也同样支持。



在浏览器中输入网址：<http://127.0.0.1:8888/page1.html>，浏览器显示如下：

图形用户界面, 图示

描述已自动生成

* 1. 计算器CGI程序

在浏览器中输入网址：<http://127.0.0.1:8888/cgi-bin/calculator.py>，浏览器显示如下：

图形用户界面, 应用程序

描述已自动生成

输入数字进行计算，浏览器显示如下：

图形用户界面, 应用程序

描述已自动生成

对于不正确的输入也进行了处理，增强了程序健壮性，效果如下：

图形用户界面, 应用程序

描述已自动生成

* 1. 数据库查询CGI程序

在浏览器中输入网址：<http://127.0.0.1:8888/cgi-bin/calculator.py>，浏览器显示如下：

图形用户界面, 应用程序

描述已自动生成

输入学号“112021xxxx”进行查询，浏览器显示如下：

图形用户界面, 应用程序

AI 生成的内容可能不正确。

对于非法输入和数据库中不存在的学号也进行了处理，增强了程序的健壮性。

图形用户界面, 应用程序

描述已自动生成

图形用户界面, 应用程序

描述已自动生成

* 1. 404页面

在服务器处理过程中，对请求不存在的文件和不支持的请求方式等特殊情况也进行了相应的处理，效果如下：

图形用户界面

描述已自动生成

* 1. 日志文件

在log目录下查看Web服务器产生的日志文件，结果如下：

图形用户界面, 文本, 应用程序, Word

AI 生成的内容可能不正确。

1. Performance and Analysis

这里使用Apifox进行性能测试，并发用户数设置为100，并设置测试时间为1min。在实际测试中，我发现由于缓存队列的存在，不同进程数并不会产生明显影响，所以这里设置线程池的缓存队列长度为1。

图形用户界面, 应用程序, Teams

描述已自动生成

* 1. 图形用户界面, 图表, 折线图

     描述已自动生成线程数为1
  2. 线程数为5

图表

描述已自动生成

* 1. 线程数为10

图形用户界面, 应用程序

描述已自动生成

* 1. 线程数为15

图形用户界面, 图表

描述已自动生成

* 1. 线程数为20

图表

描述已自动生成

* 1. 线程数为25

图形用户界面, 图表, 应用程序, 折线图

描述已自动生成

可以看到，对于几乎没有缓存的线程池，同时访问的用户数较多时，越低的线程数量则请求失败率越高。

线程数越高，每秒接口请求数越高。

但是在本次测试情况下，不同线程数对平均响应时间的影响不太清晰。

1. Summary or Conclusions

本次实验原理很简单，但是在具体实现过程中，我遇到了很多问题。具体如下：

* 请求体读取错误：在最先实现的代码中，我会遇到读请求体时程序就阻塞的问题，后来明白由于我使用了BufferedReader的readLine方法，它会缓存一部分数据，如果没有读到整行数据就会阻塞。所以我换为了InputStream的read方法直接读取，在读请求体时，会根据Content-Length字段来读取相应长度的数据。这样就不会因为多读数据造成程序阻塞。
* 图片传输：为了统一文本和图片文件传输，直接使用byte[]类型而不是String来存储数据。
* 文字编码错误：由于我的网页中包含中文，所以需要设置CGI程序的输出编码为UTF-8，“sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding="utf-8")”这行代码很重要。
* 写入没有及时刷新：要使用flush()方法保证输出及时写入到输出流中，这点在Web服务器和CGI程序中同样重要。

本程序的主要特点是代码编写较为规范，细节考虑较为全面，没有使用现成的http库，完成从零开始自己实现了一个较为简单的Web服务器。

1. References

* CGI详解（原理，配置及访问）：<https://blog.csdn.net/redparrot2008/article/details/132714474>
* HTTP请求格式及响应格式：<https://blog.csdn.net/weixin_44816919/article/details/107108737>
* java线程池详解及五种线程池方法详解：<https://blog.csdn.net/lixiangchibang/article/details/110456200>
* Java中的ProcessBuilder类详细解析：https://www.jb51.net/program/314331hvh.htm

1. Comments

本次实验较为简单，但是在实现过程中我也遇到了很多问题，如socket的阻塞、缓存区的刷新、文字编码等。这次实验确实让我对HTTP作为应用层协议有了更好的理解，而且也让我明白了CGI这个之前没有接触过的概念。