第8课 HTTP请求处理与响应

课程目标

1. 理解HTTP协议的基本结构
2. 学习如何解析HTTP请求（包括GET和POST请求）
3. 实践如何根据请求类型返回相应的响应

Http基本知识

HTTP（HyperText Transfer Protocol **超文本传输协议**）是一种基于TCP/IP协议的应用层通信协议，主要用于万维网（WWW）中客户端与服务器之间的通信。它是互联网上数据交换的基础，允许用户从WWW服务器传输超文本文件到本地浏览器。**HTTP的主要功能是定义了客户端如何向服务器请求网页内容以及服务器如何响应这些请求。**

HTTP协议规定的内容：（理解）

1. 请求/响应模型： HTTP采用客户端-服务器模式，客户端（如Web浏览器）发送HTTP请求给服务器（如Web服务器），服务器处理请求并返回HTTP响应。
2. 请求方法： 规定了多种请求方法，如GET、POST、PUT、DELETE等，每种方法对应不同的操作语义。例如，GET用于获取资源，POST用于提交数据创建新资源或更新现有资源。
3. 状态码： 服务器在响应中会包含一个三位数字的状态码，如200表示成功，404表示未找到资源，500表示服务器内部错误等，用以告知客户端请求的处理结果。
4. 消息头： 请求和响应中都包含了多个消息头字段，如Host、Content-Type、Cookie、Authorization等，用于描述请求属性、响应内容类型、客户端和服务端的附加信息等。
5. 消息体： 请求和响应可以携带一个可选的消息体，通常承载着要发送的数据或响应的具体内容，如HTML文档、JSON数据、图片文件等。

HTTP协议特点：

1. **请求/响应模型： HTTP采用客户端/服务器模式工作**，客户端（如Web浏览器）发起HTTP请求给服务器（如Web服务器），服务器接收并处理请求后返回HTTP响应。
2. **无状态： 默认情况下，HTTP协议是无状态的**，这意味着服务器不会保留两次请求之间的任何上下文信息。为了实现状态管理，可以使用Cookie、Session等技术。
    a. 顺便讲一下Cookie和Session，建议理解

Cookie

是一种**在客户端（通常是Web浏览器）存储用户信息的机制，服务器通过HTTP响应头Set-Cookie来设置Cookie。浏览器收到后将这些信息存储在本地，并在后续请求中自动附带到HTTP请求头Cookie中发送给服务器**。Cookie可以存储用户身份验证凭据、个性化设置、购物车内容等少量数据。

特点：

a. 客户端存储：Cookie是保存在客户端浏览器上的文本文件，容易被用户查看和修改。

b. 有限容量：单个Cookie大小限制通常为4KB左右，且每个域名下的Cookie数量也有限制。

c. 安全性较低：由于存储在客户端，敏感信息不宜直接存储在Cookie中，需加密处理或使用HttpOnly属性防止JavaScript读取以提高安全性。

d. 生命周期可配置：Cookie有生命周期，可以通过Expires或Max-Age属性指定过期时间，否则默认为会话级Cookie（浏览器关闭时删除）。

Cookie的好处：

a. 持久化用户状态：Cookie允许在客户端存储一些小量且非敏感的数据，如用户的语言偏好、主题设置等，使得用户在下次访问时能保持一致的体验。

b. 简化登录过程：通过将身份验证凭证（通常为加密过的Token）保存在Cookie中，实现“记住我”功能，使用户无需每次访问都重新登录。

c. 追踪分析：网站和第三方服务可以使用Cookie进行匿名或基于同意的用户行为追踪和分析，以优化用户体验和服务质量。

Session 是一种服务器端技术，用于维护用户状态。当用户访问网站并进行登录等操作时，服务器创建一个与该用户相关的Session对象并在内存或数据库中存储用户的会话信息。服务器会给客户端分配一个唯一的Session ID，并将其作为Cookie或URL重写的方式传递给客户端。客户端在后续请求中携带Session ID，服务器通过识别ID获取对应的Session数据，实现用户状态跟踪。

特点：

a. 服务器端存储：Session数据存储在服务器端，相比Cookie更安全，不易被篡改。

b. 数据量无严格限制：Session可以在服务器端存储较大量的数据，受限于服务器资源。

c. 生命周期管理：服务器可以根据需要管理Session的生命周期，如设定固定的有效期或者检测用户活动情况自动延长有效期。

d. 跨域共享问题：由于Session依赖于服务器，在分布式服务器架构下可能需要额外的技术手段（如Session复制、集中式Session存储）来实现跨多个服务器节点的Session共享。

Session的好处：

a. 安全存储：**由于Session数据存储在服务器端，相比Cookie更安全**，不易被窃取或篡改。可以放心地在其中**存储用户的登录状态和其他敏感信息。**

b. 较大容量：不受限于单个Cookie大小限制，Session可以存储更多类型和更大规模的数据。

c. 灵活管理：服务器可以根据需要管理和控制Session的生命周期，例如自动清除长时间无操作的Session，从而保护系统资源并提升安全性。

d. 跨域共享：虽然分布式架构下需要额外处理，但理论上Session可以通过设计支持跨多个服务器节点共享，实现在集群环境下的无缝会话跟踪。

总结来说，**Cookie和Session都是为了维持用户状态而在客户端和服务端之间建立的一种关联机制**，但它们在存储位置、安全性、数据量以及管理方式上存在显著差异。在实际应用中，二者常结合使用，Cookie用于传输Session ID，而Session则用来存储真正敏感和大量用户会话数据。

3. 媒体类型丰富： HTTP不仅可以传输HTML文档，还可以传输图片、视频、音频等各种格式的数据，通过Content-Type头字段来标识资源的MIME类型。

   a. MIME类型（Multipurpose Internet Mail Extensions）是一种标准，用于描述互联网上传输的数据内容的格式和类型。

   - `text/html`：表示资源是HTML文档。
   - `image/jpeg`：表示资源是JPEG格式的图片。
   - `application/json`：表示资源是以JSON格式编码的数据。
   - `video/mp4`：表示资源是一个MP4格式的视频文件。

4. 灵活的方法： HTTP定义了多种请求方法，包括GET、POST、PUT、DELETE、HEAD、OPTIONS等，每种方法都有特定的语义和用途。

   GET

   - 语义：获取资源。请求指定URI标识的资源。
   - 用途：从服务器检索数据，通常用于查询操作或加载网页内容。请求参数可以放在URL查询字符串中，请求体为空。

   POST

   - 语义：向指定URI发送数据，请求服务器处理这些数据（例如存储、更新等）并可能生成新的资源。
   - 用途：主要用于表单提交、创建新资源或者执行非幂等操作（即多次执行会产生不同结果的操作）。请求参数可以在请求体中以多种格式（如表单数据、JSON等）传输。

   PUT

   - 语义：替换目标资源的所有当前表示。如果资源不存在，则会被创建（取决于服务器实现）。
   - 用途：完整更新已存在的资源。请求体包含了整个资源的内容，客户端需提供资源的全部信息。

   DELETE

   - 语义：删除指定的资源。

- 用途：请求服务器删除指定URI标识的资源。该方法不包含请求体，仅通过URI确定要删除的对象。

HEAD

- 语义：与GET方法类似，但只返回响应头信息，不包括响应体内容。
- 用途：用于获取资源的元信息（如Content-Type、Last-Modified等），而不必传输整个实体主体，常用于检查资源是否存在、验证缓存是否有效等场景。

OPTIONS

- 语义：用于获取指定资源的HTTP请求方法列表，也可以用来探测服务器性能和功能。
- 用途：预检请求，询问服务器某个URI支持哪些HTTP方法，以及服务器对特定资源提供的选项信息。跨域资源共享（CORS）中的preflight request就是使用此方法进行通信权限确认。

5. 可扩展性： HTTP协议头部允许自定义扩展字段，以适应不断发展的网络应用需求，例如Cache-Control用于控制缓存策略，Authorization用于认证授权等。
6. 版本迭代： 目前广泛使用的HTTP版本主要有HTTP/1.1和HTTP/2，而HTTP/3也正在逐步推广中。不同版本在性能、连接管理等方面有所改进。

基本HTTP交互流程：

- 客户端构造一个HTTP请求报文，其中包括请求行（Method, URL, Version）、请求头部（Header）、空行及可选的消息体。
- 服务器收到请求后，解析请求报文，根据请求方法和URL找到对应的资源，并生成包含状态码、响应头部、空行及可选的消息体在内的HTTP响应报文。
- 客户端接收到响应报文后，解析并根据响应的内容类型展示相应的数据（如渲染HTML页面、下载文件等）。

HTTP 1.0 和 HTTP 2.0 是超文本传输协议（Hypertext Transfer Protocol）的两个主要版本，它们在功能、性能和优化方面存在显著差异。

不同版本的HTTP协议

HTTP 1.0：

- 发布于1996年，是HTTP协议的第一个正式版本。
- 连接管理：**每个请求都会创建一个新的TCP连接，完成请求后立即关闭连接，即非持久连接。这导致了“请求-响应-关闭”的模式，对资源密集型网站来说效率低下，因为建立TCP连接本身就需要时间（三次握手）。**
- 管道化：虽然HTTP 1.0引入了管道化（pipelining），允许在一个连接中发送多个请求而不必等待响应，但该特性并未被广泛支持，而且对于乱序响应的问题处理起来较复杂。
- 缓存机制：提供了基础的缓存控制指令，如 `Expires` 和 `Pragma`，但是不如后续版本完善。

- 头部压缩：不支持头部压缩，因此头部信息会占用较大的网络带宽。

HTTP 1.1：

- 发布于1999年，作为HTTP 1.0的升级版，它改进了许多不足之处，**现在大部分应用实际上使用的是HTTP 1.1而非1.0。(最多不是2.0是因为有的古老的服务器懒得更新)**
- 持久连接：**默认采用持续连接（Keep-Alive），一个TCP连接可以服务于多个请求，减少连接建立与释放的成本。**
- 管道化：虽然理论上支持，但在实际部署中由于上述问题，其效果有限。
- 缓存机制：新增了许多缓存控制相关的头部字段，如`Cache-Control`、`ETag`和`If-Modified-Since`等，提高了缓存的有效性和准确性。
- 分块传输编码：允许服务器将大对象分割成较小的数据块进行传输，客户端可以在接收到部分数据后就开始显示内容。

HTTP/2：

- 发布于2015年，相比HTTP 1.x有了重大改进，旨在解决旧版HTTP中的性能瓶颈。
- **多路复用：通过单一TCP连接并行发送多个请求和响应，解决了队头阻塞问题，大幅提升了页面加载速度。**
- 二进制分帧：将HTTP消息分解为更小的独立帧，以实现更高效的传输和优先级排序。
- 头部压缩：使用HPACK算法对头部进行压缩，极大地减少了冗余的头部信息，降低了网络延迟。
- 服务器推送：服务器能够预测客户端可能需要的资源，并主动将其推送给客户端，进一步提升性能。

总的来说，HTTP 1.0到HTTP/2的演进过程反映了互联网技术不断发展以及**用户对网页加载速度需求的提高**。HTTP/2针对HTTP 1.x的主要痛点进行了优化，显著改善了Web性能。

HTTP和HTTPS

HTTP（HyperText Transfer Protocol）和HTTPS（Hypertext Transfer Protocol Secure）都是用于在互联网上传输数据的应用层协议，但它们之间存在显著区别：

1. 安全传输：
   - HTTP：HTTP是一种无状态、明文传输的协议，不提供任何加密措施。这意味着通过HTTP传输的数据是未加密的，容易被监听、篡改或伪造。
   - HTTPS：HTTPS是在HTTP的基础上添加了SSL/TLS协议层，对传输数据进行加密，从而保证通信过程中的安全性。它可以保护用户信息如账号密码、交易数据等不被第三方窃取。
2. 身份验证：
   - HTTP：HTTP协议本身无法验证服务器的身份，任何人都可以架设一个声称是某个网站的服务器，这可能导致中间人攻击或者钓鱼网站欺诈。

- HTTPS：使用了SSL/TLS证书，可以实现服务器的身份验证。浏览器会检查服务器提供的证书是否由受信任的CA机构颁发，并确认其域名与证书上的域名一致，从而确保用户访问的是真实可靠的服务器。

3. 信任标志：
   - HTTP：在地址栏显示为"http://example.com"，没有安全锁图标或“不安全”提示。
   - HTTPS：在地址栏显示为"https://example.com"，通常有一个小绿锁图标或绿色的文字标识，表示连接安全。现代浏览器还会明确标记出不安全的HTTPS连接。
4. 性能影响：
   - HTTP：由于不需要进行加密解密操作，**理论上HTTP在数据传输速度上比HTTPS稍快一些**。
   - HTTPS：虽然增加了加密和认证环节，导致一定的性能开销，但随着硬件计算能力提升和技术优化，这个差距正在逐渐缩小。而且很多现代浏览器支持HTTP/2、HTTP/3以及TLS 1.3等新标准，在一定程度上弥补了HTTPS的性能损失。

综上所述，**HTTPS提供了更高级别的数据加密和服务器身份验证功能**，是现代Web应用和服务推荐使用的传输协议，尤其是在涉及敏感信息交换的情况下。

HTTP请求

HTTP请求结构

HTTP请求是客户端向服务器发送的消息，用于请求资源或提交数据。每个HTTP请求包含以下部分：

1. 请求行（Request Line）：请求行包含了HTTP方法、URI（统一资源标识符）和HTTP版本。它的格式通常如下：

```
HTTP方法 URI HTTP版本
```

1. 例如，以下请求行表示客户端请求获取 `/index.html` 资源，使用HTTP 1.1协议：

```
GET /index.html HTTP/1.1
```

   - HTTP方法（Method）：指定了客户端的请求类型，常见的方法包括GET、POST、PUT、DELETE等。
   - URI（Uniform Resource Identifier）：标识了所请求的资源的位置和名称。
   - HTTP版本：指定了所使用的HTTP协议版本，例如HTTP/1.1。

1. 请求头（Request Headers）：请求头包含了附加的信息，用于描述客户端的环境和请求体的信息。请求头以一行一行的键值对形式出现，每行由字段名和字段值组成，中间使用冒号（:）分隔。以下是一些常见的请求头示例：

```
Content-Type: application/json
User-Agent: Mihoyo/5.0
```

   - `Content-Type` 字段指定了请求体的内容类型，这对于服务器解析请求体非常重要，例如，它可以表示请求体是JSON格式数据。
   - `User-Agent` 字段通常包含了客户端的用户代理信息，用于标识请求的来源，例如浏览器类型和版本。

1. 请求体（Request Body，可选）：请求体仅在一些HTTP方法中使用，主要用于传输数据。通常，GET请求没有请求体，而POST请求则可能包含表单数据、JSON数据或文件内容等。请求体的格式和内容取决于请求头中的 `Content-Type` 字段。

GET与POST请求

- GET请求：
  - 用于请求服务器上的资源。
  - 参数通常附加在URI后，如 `/search?q=` yuanshen，表示请求搜索关键词为"yuanshen"的结果。

```
GET /search?q=yuanshen HTTP/1.1
Host: www.example.com
User-Agent: Mihoyo/5.0
```

HTTP方法是GET，表示请求获取资源。

URI是 `/search?q=openai` ，包含了参数 `q` ，其值为 `openai` ，表示搜索关键词为"openai"的结果。

Host字段指定了服务器的主机名。

User-Agent字段表示请求来自Mihoyo浏览器。

- POST请求：
  - 用于向服务器提交数据，如表单提交。
  - 数据放置在请求体中，不出现在URI中。

```
POST /submit-form HTTP/1.1
Host: www.example.com
User-Agent: Mihoyo/5.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 21

username=johndoe&password=12345

```

HTTP方法是POST，表示请求提交数据到服务器。

URI是 `/submit-form` ，不包含参数。

Host字段指定了服务器的主机名。

User-Agent字段表示请求来自Mihoyo浏览器。

Content-Type字段指定了请求体的内容类型为 `application/x-www-form-urlencoded` ，表示数据以表单形式编码。

Content-Length字段指定了请求体的长度为21个字节。

请求体中包含了表单数据，其中 `username` 字段的值为 `johndoe` ， `password` 字段的值为 `12345` 。

请求解析

解析HTTP请求的目的是获取请求中的关键信息，如方法、URI和请求体内容。

- 解析方法和URI：
  - 从请求行中提取HTTP方法（GET、POST等）和请求的URI。
  - 例如，请求行 `GET /index.html HTTP/1.1` 中，方法为GET，URI为 `/index.html` 。
- 处理POST请求体中的数据：
  - 如果请求方法是POST，从请求体中提取提交的数据。
  - 例如，表单提交的POST请求中，请求体可能包含键值对形式的数据。

实例：解析HTTP请求

假设接收到的HTTP请求内容为：

```
GET /index.html HTTP/1.1
Host: www.example.comUser-Agent: Mihoyo/5.0
```

- 解析请求：

- 请求行为 `GET /index.html HTTP/1.1`，表示这是一个GET请求，请求的资源为 `/index.html`。
- 请求头包含 `Host: www.example.com` 和 `User-Agent: Mihoyo/5.0`，分别表示请求的服务器地址和客户端信息。
- 由于是GET请求，因此没有请求体。

HTTP响应

HTTP响应结构

1. 状态行（Status Line）

状态行是HTTP响应的第一部分，包含了HTTP版本、状态码和状态消息。其格式如下：

```
HTTP版本 状态码 状态消息
```

例如：

```
HTTP/1.1 200 OK
```

- HTTP版本：标识了服务器返回响应时所使用的HTTP协议版本。
- 状态码：三位数字代码，指示请求处理的结果。如 `200` 表示成功，`404` 表示未找到资源，`500` 表示服务器内部错误等。
- 状态消息：对状态码的简短描述，如"OK"、"Not Found"等。

2. 响应头（Response Headers）

响应头提供了关于响应的附加信息，以键值对的形式列出。示例：

```
Content-Type: text/html; charset=UTF-8
Cache-Control: max-age=3600
Server: Apache/2.4.41
```

- Content-Type：指定了响应体的内容类型及编码。
- Cache-Control：指导客户端如何缓存响应内容。
- Server：标明处理请求的服务器及其软件版本。

3. 响应体（Response Body，可选）

响应体包含了服务器向客户端发送的实际数据，根据响应头中的Content-Type字段解析。响应体可以是HTML文档、JSON数据、图片或其他任何类型的数据。

GET与POST响应示例

GET响应示例

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mihoyo/5.0
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 1024

<!DOCTYPE html...
```

在上述例子中，服务器成功返回了GET请求的资源，并通过响应体返回了一个HTML页面。

POST响应示例

```
POST /submit-form HTTP/1.1
Host: www.example.com
User-Agent: Mihoyo/5.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 21
username=johndoe&password=12345

HTTP/1.1 201 Created
Location: /profile/johndoe
Content-Type: application/json
Content-Length: 56

{"status": "success", "message": "Form submitted"}
```

在这个例子中，服务器接收到并成功处理POST请求后，返回一个201状态码，表示新资源已创建，并通过响应体提供JSON格式的操作结果。

响应解析

- 解析状态码和状态消息：从状态行中提取相关信息，了解请求执行的成功与否。

- 处理响应体：根据响应头中的Content-Type字段确定响应体格式，并从中获取具体数据进行进一步处理。例如，如果是JSON格式，则将其解析为JSON对象。

代码逻辑

实操内容

1. 解析HTTP请求
    - 从接收到的HTTP请求中解析出请求方法和URI。
    - 对于POST，解析请求体。
2. 处理GET和POST请求
    - GET：提取URI参数，进行处理。
    - POST：获取请求体数据。
3. 生成并返回响应
    - 根据处理结果，生成HTTP响应。
    - 设置响应头和响应体，返回客户端。

实战示例：简单HTTP服务器

1. 服务器初始化
    - 创建TCP socket，绑定端口，监听。
2. 解析HTTP请求
    - 接收客户端HTTP请求。
    - 使用 `parseHttpRequest` 函数解析请求方法和URI。
3. 处理请求
    - 根据URI调用相应处理函数。
    - 针对GET/POST执行逻辑。
4. 发送响应
    - 构建HTTP响应，发送回客户端。

代码分析

```cpp
// 解析HTTP请求
std::pair<std::string, std::string> parseHttpRequest(const std::string& request) {
    // 找到第一个空格，确定HTTP方法的结束位置
    size_t method_end = request.find(" ");
    // 提取HTTP方法（如GET、POST）
    std::string method = request.substr(0, method_end);

    // 找到第二个空格，确定URI的结束位置
    size_t uri_end = request.find(" ", method_end + 1);
    // 提取URI（统一资源标识符）
    std::string uri = request.substr(method_end + 1, uri_end - method_end - 1);

    // 返回解析出的HTTP方法和URI
    return {method, uri};
}

// 处理HTTP请求
std::string handleHttpRequest(const std::string& method, const std::string& uri, const std::string& body) {
    // 检查GET请求和URI是否在路由表中
    if (method == "GET" && get_routes.count(uri) > 0) {
        // 根据URI调用相应的处理函数
        return get_routes[uri](body);
    }
    // 检查POST请求和URI是否在路由表中
    else if (method == "POST" && post_routes.count(uri) > 0) {
        // 根据URI调用相应的处理函数
        return post_routes[uri](body);
    }
    // 如果请求方法和URI不匹配任何路由，则返回404错误
    else {
        return "404 Not Found";
    }
}

```

- `parseHttpRequest`：解析出HTTP请求的方法和URI。
- `handleHttpRequest`：根据解析出的方法和URI，调用对应的处理函数。

新增后的代码变成了

```cpp
#include <iostream>
#include <map>
#include <functional>
#include <string>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>

#define PORT 8080

// 请求处理函数类型定义
using RequestHandler = std::function<std::string(const std::string&)>;

// 分别为GET和POST请求设置路由表
std::map<std::string, RequestHandler> get_routes;
std::map<std::string, RequestHandler> post_routes;

// 初始化路由表
void setupRoutes() {
    // GET请求处理
    get_routes["/"] = [](const std::string& request) {
        return "Hello, World!";
    };
    get_routes["/register"] = [](const std::string& request) {
        // TODO: 实现用户注册逻辑
        return "Please use POST to register";
    };
    get_routes["/login"] = [](const std::string& request) {
        // TODO: 实现用户登录逻辑
        return "Please use POST to login";
    };

    // POST请求处理
    post_routes["/register"] = [](const std::string& request) {
        // TODO: 实现用户注册逻辑
        return "Register Success!";
    };
    post_routes["/login"] = [](const std::string& request) {
        // TODO: 实现用户登录逻辑
        return "Login Success!";
    };

    // TODO: 添加其他路径和处理函数
}

```

```cpp
// 第六课新增，解析HTTP请求
std::pair<std::string, std::string> parseHttpRequest(const std::string& request) {
    // 找到第一个空格，确定HTTP方法的结束位置
    size_t method_end = request.find(" ");
    // 提取HTTP方法（如GET、POST）
    std::string method = request.substr(0, method_end);

    // 找到第二个空格，确定URI的结束位置
    size_t uri_end = request.find(" ", method_end + 1);
    // 提取URI（统一资源标识符）
    std::string uri = request.substr(method_end + 1, uri_end - method_end - 1);

    // 返回解析出的HTTP方法和URI
    return {method, uri};
}

// 处理HTTP请求
std::string handleHttpRequest(const std::string& method, const std::string& uri, const std::string& body) {
    // 检查GET请求和URI是否在路由表中
    if (method == "GET" && get_routes.count(uri) > 0) {
        // 根据URI调用相应的处理函数
        return get_routes[uri](body);
    }
    // 检查POST请求和URI是否在路由表中
    else if (method == "POST" && post_routes.count(uri) > 0) {
        // 根据URI调用相应的处理函数
        return post_routes[uri](body);
    }
    // 如果请求方法和URI不匹配任何路由，则返回404错误
    else {
        return "404 Not Found";
    }
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // 创建socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    // 定义地址
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
```

```
    address.sin_port = htons(PORT);

    // 绑定socket
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));

    // 监听请求
    listen(server_fd, 3);

    // 设置路由
    setupRoutes();

    while (true) {
        // 接受连接
        new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen);

        // 读取请求
        char buffer[1024] = {0};
        read(new_socket, buffer, 1024);
        std::string request(buffer);

        // 解析请求
        auto [method, uri] = parseHttpRequest(request);

        // 处理请求
        std::string response_body = handleHttpRequest(method, uri, request);

        // 发送响应
        std::string response = "HTTP/1.1 200 OK\nContent-Type: text/plain\n\n"
+ response_body;
        send(new_socket, response.c_str(), response.size(), 0);

        // 关闭连接
        close(new_socket);
    }

    return 0;
}

```


课后练习

1. **代码实践**：尝试在本地运行提供的服务器代码，理解其工作流程。

2. **扩展功能**：尝试添加新的路由和处理函数，比如处理不同的GET请求或POST请求。

HTTP 相关的互联网面试常考内容

问题1：HTTP 和 HTTPS 的区别是什么？

回答：

- **传输层安全性**：HTTP 是明文传输，不安全；HTTPS 使用 SSL/TLS 协议，对数据进行加密传输，确保安全性。

- **端口号**：HTTP 默认使用端口 80；HTTPS 默认使用端口 443。

- **证书要求**：HTTPS 需要申请数字证书，确保服务器的身份可信。

问题2：HTTP 的请求方法有哪些？分别有什么作用？

回答：

- **GET**：获取资源，无副作用。

- **POST**：提交数据，可能修改服务器状态。

- **PUT**：更新或创建资源。

- **DELETE**：删除资源。

- **HEAD**：获取响应头，不返回响应体。

- **OPTIONS**：查询服务器支持的请求方法。

- **PATCH**：部分更新资源。

问题3：HTTP 的状态码 301 和 302 有什么区别？

回答：

- **301 Moved Permanently**：永久重定向，表示资源已被永久移动，以后应使用新的 URL 访问。
- **302 Found**：临时重定向，表示资源暂时被移动，未来可能还会恢复。

客户端应根据具体状态码，决定是否更新缓存的资源地址。

问题4：什么是 HTTP 的持久连接？有什么好处？

回答：

- **持久连接（Keep-Alive）**：在一次 TCP 连接中，可以传输多个 HTTP 请求和响应，减少了连接的建立和关闭次数。
- **好处**：
  - 减少了 TCP 连接的开销，提高了传输效率。
  - 降低了网络拥塞，提高了页面加载速度。

问题5：如何理解 HTTP 的无状态性？如何在服务端维护用户状态？

回答：

- **无状态性**：HTTP 协议本身不保留客户端的状态，每次请求都是独立的。
- **维护用户状态的方法**：
  - **Cookie**：在客户端存储状态信息，每次请求时发送给服务器。
  - **Session**：在服务器端存储用户状态，通过 Session ID 关联。
  - **Token**：使用令牌（如 JWT）在客户端和服务器之间传递用户身份信息。

问题6：什么是跨域请求？如何解决跨域问题？

回答：

- **跨域请求**：出于安全考虑，浏览器限制了从一个域名的网页向另一个域名发送请求的行为，即同源策略。

- **解决方法**：

  - **CORS（跨域资源共享）**：服务器设置响应头 `Access-Control-Allow-Origin`，允许跨域访问。

  - **JSONP**：通过 `<script>` 标签加载跨域的 JavaScript 回调函数。

  - **反向代理**：通过服务器代理请求，避免跨域。

问题7：HTTP/1.1 与 HTTP/2 有哪些区别？

回答：

- **二进制分帧**：HTTP/2 使用二进制格式传输数据，而 HTTP/1.1 使用文本格式。

- **多路复用**：HTTP/2 可以在一个 TCP 连接中并行发送多个请求和响应，消除了队头阻塞。

- **头部压缩**：使用 HPACK 算法压缩头部信息，减少了带宽占用。

- **服务器推送**：服务器可以主动向客户端推送资源，提高页面加载速度。

问题8：什么是缓存？HTTP 缓存头有哪些？

回答：

- **缓存**：为了提高资源的加载速度和减少服务器压力，浏览器会将资源缓存到本地，在一定条件下直接使用缓存的资源。
- **HTTP 缓存头**：
  - **Expires**：过期时间，HTTP/1.0 使用。
  - **Cache-Control**：缓存控制，HTTP/1.1 使用，包括 `max-age`、`no-cache`、`no-store` 等指令。
  - **Last-Modified / If-Modified-Since**：基于资源的最后修改时间验证缓存。
  - **ETag / If-None-Match**：基于资源的唯一标识验证缓存。

问题9：Cookie 和 Session 的区别是什么？

回答：

- **存储位置**：
  - **Cookie**：存储在客户端浏览器。
  - **Session**：存储在服务器端。

- **安全性**：
  - **Cookie**：容易被篡改和劫持，安全性较低。
  - **Session**：相对安全，但需要通过 Session ID 关联。

- **性能**：
  - **Cookie**：每次请求都会发送，可能增加带宽消耗。
  - **Session**：保存在服务器，消耗服务器资源。

问题10：如何理解 RESTful 风格的 API 设计？

回答：

- **RESTful** 是一种基于 HTTP 协议的 API 设计风格，遵循以下原则：
  - **资源定位**：通过 URL 定位资源。
  - **统一接口**：使用标准的 HTTP 方法（GET、POST、PUT、DELETE）操作资源。
  - **无状态性**：服务器不保留客户端状态，每次请求都包含必要的信息。
  - **数据格式**：使用 JSON、XML 等通用格式传输数据。

问题11：什么是状态码 403 和 401？它们有什么区别？

回答：

- **401 Unauthorized**：表示请求未通过身份验证，需要提供有效的认证信息（如登录）。
- **403 Forbidden**：表示服务器理解请求，但拒绝执行，可能是权限不足或资源被禁用。

问题12：如何使用 HTTP 实现文件的断点续传？

回答：

- **使用范围请求（Range Request）**：
  - 客户端在请求头中加入 `Range` 字段，指定请求的字节范围。
  - 服务器返回状态码 **206 Partial Content**，并在响应头中包含 `Content-Range` 字段。

示例请求：

```
GET /file.zip HTTP/1.1
Host: www.example.com
Range: bytes=500-999
```

问题13：如何在 HTTP 中实现长连接和短连接？

回答：

- **短连接**：每次请求-响应后，立即关闭 TCP 连接。HTTP/1.0 默认使用短连接。

- **长连接**：在请求头中加入 `Connection: keep-alive`，保持 TCP 连接，进行多次请求。HTTP/1.1 默认使用长连接。

问题14：什么是 MIME 类型？为什么需要它？

回答：

- **MIME 类型（Multipurpose Internet Mail Extensions）**：用于描述文件或网络传输内容的类型。

- **作用**：

- 告诉客户端如何处理接收到的数据（如显示文本、渲染图片、播放视频）。
- 在请求头和响应头中使用 `Content-Type` 字段指定 MIME 类型。

问题15：简述一下 HTTP 报文的传输过程。

回答：

1. **客户端构建请求报文**：包含请求行、请求头和可选的请求体。
2. **通过 TCP 连接发送请求**：使用套接字传输数据。
3. **服务器接收并解析请求**：根据请求信息处理相应的业务逻辑。
4. **服务器构建响应报文**：包含状态行、响应头和可选的响应体。
5. **通过 TCP 连接发送响应**：将数据返回给客户端。
6. **客户端接收并解析响应**：根据状态码和内容进行处理，如渲染页面。
