

# PART 01

## 开发环境准备

ENVIRONMENT PREPARATION

# 环境配置

## 系统环境

Linux / macOS 或 Windows

Windows用户推荐在WSL下开发，可以和OS课程使用相同环境

实验在Ubuntu22.04+进行了测试，使用更低版本/其他发行版时部分操作可能略有差异



# 环境配置

## 开发工具

### [推荐] Git

推荐使用Git进行版本管理（含独立完成），务必避免用微信等IM同步代码

注意：代码必须以**私有仓库**的形式托管在GitHub等代码托管平台上

## 编译环境

使用以下命令确认编译环境是否配置正确，如出现报错，请根据文档进行配置

```
gcc --version  
g++ --version
```

你也可以根据个人喜好，选择用cmake/make等，但需要在Readme里注明编译方法

# 辅助工具

## GLog

std::cout不是线程安全的，实验多线程环境下可能产生交错混乱的输出，而GLog不受影响  
GLog同时可以提供日志分级、条件记录、格式化、调用栈信息输出等功能



```
#include <iostream>
#include <glog/logging.h> // For glog
class GlogWrapper{ // 封装Glog
public:
    GlogWrapper(char* program) {
        google::InitGoogleLogging(program);
        FLAGS_log_dir="PATH/prefix_"; //设置log文件保存路径及前缀
        FLAGS_alsologtostderr = true; //设置日志消息除了日志文件之外是否去标准输出
        FLAGS_colorlogtostderr = true; //设置记录到标准输出的颜色消息（如果终端支持）
        FLAGS_stop_logging_if_full_disk = true; //设置是否在磁盘已满时避免日志记录到磁盘
        // FLAGS_stderrthreshold=google::WARNING; //指定仅输出特定级别或以上的日志
        google::InstallFailureSignalHandler();
    }
    ~GlogWrapper() { google::ShutdownGoogleLogging(); }
};
```

# 辅助工具

## GLog



```
int main(int argc, char* argv[]) {
    // 初始化 glog
    auto glog = GlogWrapper(argv[0]);
    // 记录不同级别的日志
    LOG(INFO) << "This is an info message.";
    LOG(WARNING) << "This is a warning message.";
    LOG(ERROR) << "This is an error message.";
    // 相当于LOG(ERROR) + return -1;
    LOG(FATAL) << "This is a fatal message. The program will terminate after this message.";
    // 条件日志
    int x = 2;
    LOG_IF(INFO, x % 2 == 0) << "x is even.";
    LOG_IF(INFO, x % 2 != 0) << "x is odd.";
    // 每n次循环记录一条日志
    for (int i = 0; i < 5; ++i) { LOG_EVERY_N(INFO, x) << "Log every " << x << " iter, current: " << i; }
    // 触发段错误，演示崩溃处理效果
    int *p = nullptr;
    p[0] = 0;
}
```

# 辅助工具

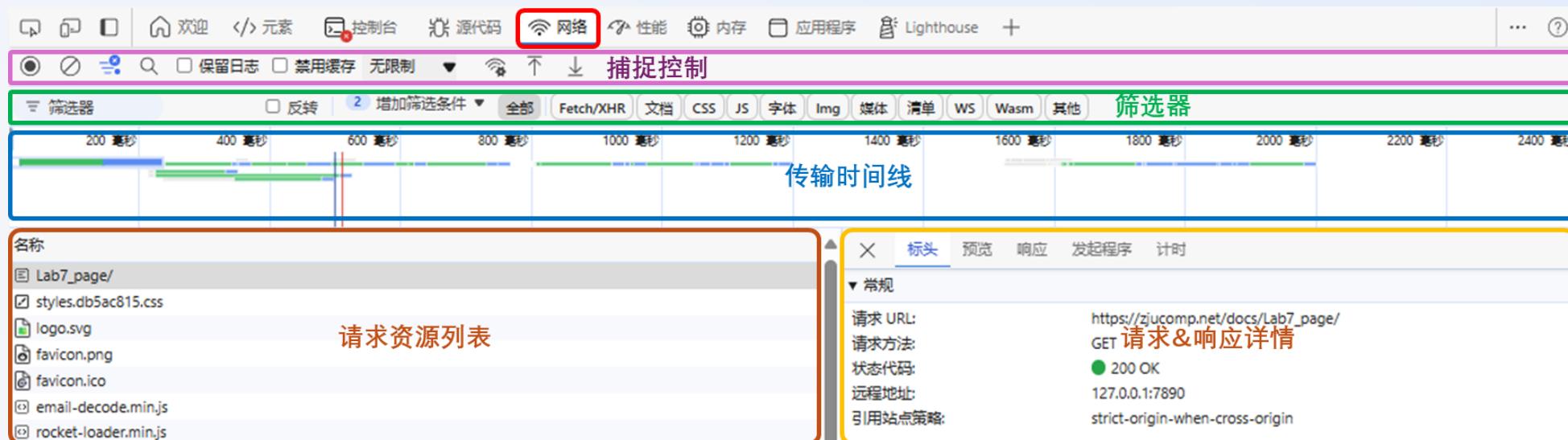
## Firefox浏览器

如果端口转发存在问题，你可能需要在WSL内启动浏览器进行测试

## 浏览器 - 开发人员工具 - 网络

主要用于观察网页加载时浏览器和Web服务器间发生的交互

如果你的列表中显示信息较少，可以右键顶部字段（如“名称”），并勾选想要查看的字段



# 辅助工具

## Wireshark

### Lab7

被测程序运行在哪个系统内，就在哪里运行Wireshark

运行在Windows：在Windows下运行Wireshark，并进行抓包

运行在WSL：在WSL下运行Wireshark，并进行抓包

**[WSL内安装Wireshark]** <https://zjucomp.net/docs/Wireshark/install/>

实验文档-Wireshark安装与使用-Wireshark安装-Linux

**[无法正确解析HTTP协议]** <https://zjucomp.net/docs/Wireshark/qa>

实验文档-Wireshark安装与使用-使用答疑Q&A

### Lab8

WSL正确启用localhost转发后，也可在Windows运行Wireshark并进行抓包

# 测试框架

## 运行系统环境

Lab7 / 8 localhost转发正确运行时，在WSL/Windows下运行测试框架均可

## 运行环境配置

Python 3

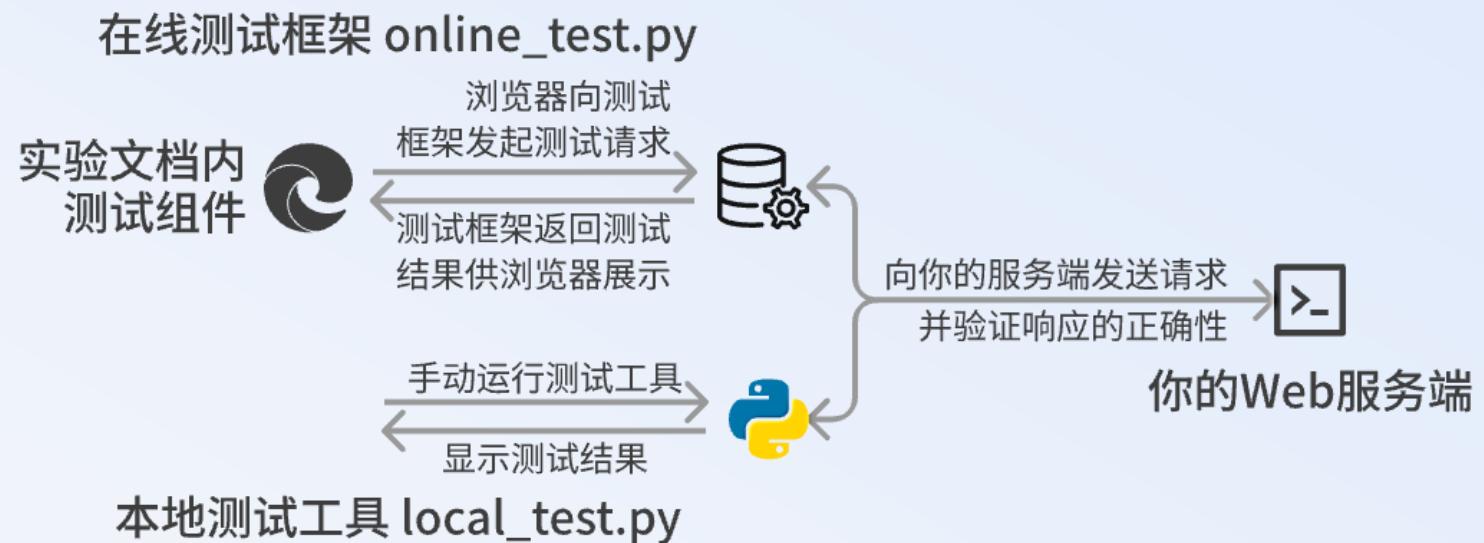
推荐Python3.10+

使用Uv等环境管理器安装  
均可

## 安装依赖包

参见requirement.txt

## 测试流程



# PART 02

## 实验原理

EXPERIMENT PRINCIPLE

# Lab7 核心动机



TCP是面向字节流的协议，那为什么TCP还有数据包、协议头？

TCP本来就是报文传输的，它只是**不对用户承诺这一点**！这意味着：

- 不保证将提交给TCP协议栈的报文**一次性完整**传输到对端
- 不保证将连续提交给TCP协议栈的若干报文**彼此分离地**、按提交次数分包传输

由于TCP只对用户承诺面向字节流的有序可靠传输，使用TCP协议时我们不可避免会需要将接收的字节流转换回一个个数据包，因此需要解决一系列的问题

# Lab7 核心动机

TCP是面向字节流的协议，那为什么TCP还有数据包、协议头？

TCP本来就是报文传输的，它只是**不对用户承诺这一点**！这意味着：

- 不保证将提交给TCP协议栈的报文**一次性地完整**传输到对端
- 不保证将连续提交给TCP协议栈的若干报文**彼此分离地**、按提交次数分包传输

由于TCP只对用户承诺面向字节流的有序可靠传输，使用TCP协议时我们不可避免会需要将接收的字节流转换回一个个数据包，因此需要解决一系列的问题

- 如何将字节流拆分对应到每个数据包？——数据包定界
- 如何从C++数据结构与字节流相互转化？——序列化/反序列化、字段定界

在此基础上，网络还是**并发的、异步的**，这意味着我们无法预料消息会以怎样的时间、频率传来，因此我们还需要解决：

- 服务端（面向多个客户端）/客户端（用户交互/接收信息）的多线程实现

## 实验8

# WEB服务器实现

主讲：王信博



# 网页加载过程



The screenshot shows a web-based experiment interface. At the top, there's a navigation bar with links like '计算机网络课程实验', '实验指导', '课程解析', '延伸阅读', '测试框架', '文档仓库', and search functions. The main content area has a title 'Lab8 实现一个轻量级的WEB服务器'. Below it, a section titled '实验资源' lists several files: 'Lab8 实验报告模板 Markdown版本' (25.7KB), 'Lab8 实验报告模板 Word版本' (60.0KB), '实验测试素材' (44.2KB), and '实验测试框架' (112KB). To the right, a sidebar contains a numbered list of topics related to HTTP and its implementation.

- 0 实验目的与意义
- 1 初识HTTP——概念浅析
- 2 让我们说HTTP 🎙——HTTP协议格式解析
- 2.1 HTTP请求
- 2.2 HTTP响应
- 3 什么事要办？——HTTP方法解析
- 4 事情怎样办？——HTTP头字段解析
- 5 东西怎么找&送？——资源访问处理与响应
- 5.1 URI解析与映射
- 5.2 状态码处理
- 6 具体功能实现
- 6.1 文件资源访问

**DNS解析**: 浏览器通过DNS解析得到相应服务器IP地址

**建立连接**: 浏览器根据IP, 通过TCP/IP (HTTP/3改用QUIC) 协议与服务器建立连接

**发送请求**: 浏览器构建HTTP请求报文, 通过已建立的连接发送给服务器

**处理请求**: 服务器根据请求内容执行相应业务逻辑 (如查询数据库等) 并返回响应

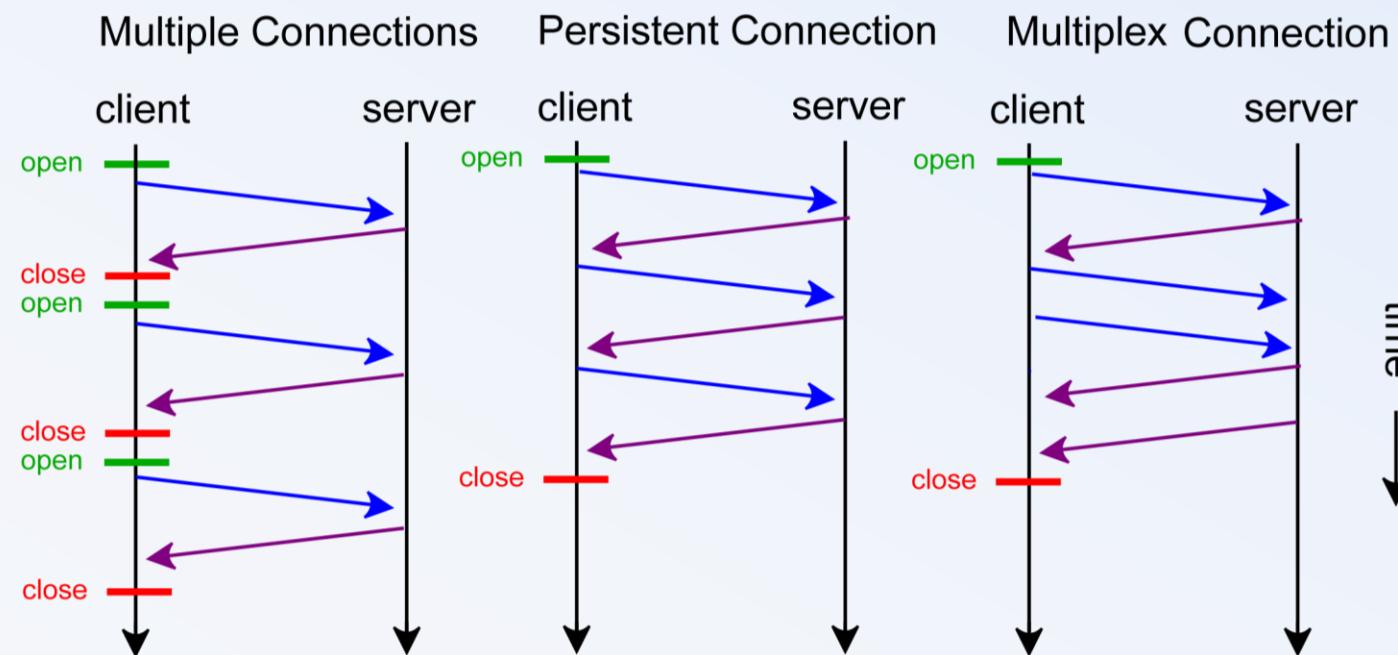
**关闭连接**: 完成请求后, 关闭连接 (时机与HTTP版本相关)

# 网页加载过程

**HTTP/1.0** 每次请求都需要重新建立TCP连接

**HTTP/1.1** 通过持久连接和请求管道化技术，允许在一个TCP连接上发送多对请求-响应，减少了连接建立的时间开销

**HTTP/2** 在此基础上引入了多路复用技术，允许多个请求和响应在同一连接上任意交错传输，不需要等待前一对响应传输完成



● ● ●

```

<!DOCTYPE html>
<html>
<head>
  <title>Test</title>
  <link rel="stylesheet" href=".../antd.min.css">
  <style>
    ...
  </style>
</head>
<body>
  <div class="form-container">
    
    <h1>This is a test</h1>
  </div>
</body>
</html>

```

# HTTP协议格式

请求行	请求方法 <b>URI</b> HTTP版本	状态行	HTTP版本 状态码 状态码描述
请求头	key: value key: value key: value ...	响应头	key: value key: value key: value ...
空行	空行	空行	空行
请求正文	请求相关的信息和数据	响应正文	HTML/图片/JSON等类型数据

# HTTP请求方法



属性	GET	POST	OPTIONS	HEAD	PUT	DELETE	PATCH
请求体	可选	通常	可选	可选	√	可选	√
响应体	√	√	√	✗	√	√	√
安全性	√	✗	√	√	✗	✗	✗
幂等性	√	✗	√	√	√	√	✗
可缓存	√	有时	✗	√	✗	✗	✗

- **安全性**: 相应方法只表示获取资源信息的意图，不包含任何请求副作用的意图
- **幂等性**: 无错误等意外情况时，多次相同请求无副作用/和单次请求的相同
- **缓存**: 允许将响应结果存储在客户端/其他中间节点（如CDN），以便后续直接使用缓存



# RESTful or POSTful?

## 表述性状态传递RESTful (Representational State Transfer)

关键特性：客户-服务器模式，无状态通信，缓存机制，统一接口，分层系统，[可选]按需代码  
每个URL表示一个特定的资源，HTTP方法表示对资源的动作，比如：

**GET** /users            获取用户列表

**GET** /users/xbwang    获取特定用户信息

**POST** /users            创建新用户

Content-Type: application/json

```
{  
  "name": "ZhangSan",  
  "email": "zhsan@zju.edu.cn",  
  "passwd": "meiyoumima"  
}
```

**DELETE** /users/ZhangSan    删除张三

**PATCH** /users/ZhangSan    部分修改

Content-Type: application/json  
{ "email": "zhsix@zju.edu.cn" }

**PUT** /users/ZhangSan    完整修改

Content-Type: application/json

```
{  
  "name": "ZhangSan",  
  "email": "zhsix@zju.edu.cn",  
  "passwd": "xindemima"  
}
```



# RESTful or POSTful?

**POSTful** (不存在这种规范) 全部请求使用POST方法，一些RPC框架会这么实现

原因：运营商/企业安全团队对非GET/POST乱处理，影响正常业务进行；

中间网关/运营商/客户端预期外的GET请求缓存；QueryString解析不兼容

**POST** /getUserList 获取用户列表

**POST** /getUser 获取特定用户信息

Content-Type: application/json

{ "name": "xbWang" }

**POST** /createUser 创建新用户

Content-Type: application/json

{

    "name": "ZhangSan",  
    "email": "zhsan@zju.edu.cn",  
    "passwd": "meiyoumima"

}

**POST** /deleteUser 删除张三

Content-Type: application/json

{ "name": "xbWang" }

**POST** /modifyUser 完整修改

Content-Type: application/json

{

    "name": "ZhangSan",  
    "email": "zhsix@zju.edu.cn",  
    "passwd": "xindemima"

}

# HTTP请求方法

属性	GET	POST	OPTIONS	HEAD	PUT	DELETE	PATCH
请求体	可选	通常	可选	可选	√	可选	√
响应体	√	√	√	✗	√	√	√
安全性	√	✗	√	√	✗	✗	✗
幂等性	√	✗	√	√	√	√	✗
可缓存	√	有时	✗	√	✗	✗	✗

世界破破烂烂，  
我们缝缝补补.....

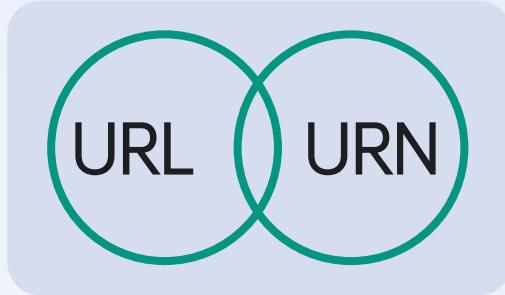
没有完美的解决方案，根据实际情况做最适合的选择

实验中，我们需要区分GET/POST方法，执行相应的功能逻辑并返回响应

# HTTP协议格式

请求行	请求方法 <b>URI</b> HTTP版本	状态行	HTTP版本 状态码 状态码描述
请求头	key: value key: value key: value ...	响应头	key: value key: value key: value ...
空行	空行	空行	空行
请求正文	请求相关的信息和数据	响应正文	HTML/图片/JSON等类型数据

# 统一资源标识符URI



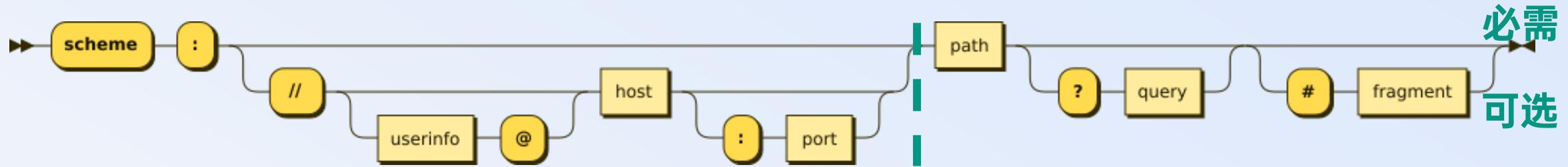
## 统一资源定位符URL (Uniform Resource Locator)

最常见的URI形式，提供了访问特定资源的路径和方法，帮助计算机确定资源是什么，同时说明如何找到并获取这个资源

## 统一资源名称URN (Uniform Resource Name)

URN更关注资源的身份标识而非物理位置，旨在提供一种持久不变的名字空间，即使资源的实际位置发生变化，其URN仍然保持不变

# 统一资源定位符URL



```
https : // zjucomp.net /docs/Lab8_page # 21-HTTP请求  
http : // 10.214.0.253 : 80 /...asid.php ? stu=5029&cid=105...  
mailto : // ZhangSan @ zju.edu.cn
```

scheme	指定访问资源时使用的协议类型，如HTTP、HTTPS、FTP等
host	资源所在的主机名或IP地址
port	指定主机上的端口号，如未指定，可能使用scheme默认端口
path	资源在服务器上的具体位置，如path/to/file
query	用于传递给服务器的查询参数，通常以键值对的形式出现
fragment	用于指示页面内部的一个特定部分或元素，通常用于页面内的导航

# URL映射

文件描述	文件路径	映射后URL
带有图片的首页HTML文件	/html/test.html	/index.html
去掉图片的首页HTML文件	/html/noimg.html	/index_noimg.html
纯文本文件	/txt/test.txt	/info/server
浙大校标图片文件	/img/logo.jpg	/assets/logo.jpg



1,383,329

Total Requests

253,937

Threats Mitigated

18.36%

Requests Mitigated as Threats

**安全性：**避免常见的路径遍历攻击，保证无法通过构造URI确定内部文件资源情况

**可维护性：**内部文件路径发生变更时，不需要修改外部URL，反之亦然

**灵活性：**可以实现多对多映射关系，同一URI可以根据地区等信息映射到不同资源

**性能优化：**可以通过合适的路径映射设计，实现对资源缓存策略的高效管理

# HTTP协议格式

请求行	请求方法 <b>URI</b> HTTP版本	状态行	HTTP版本 状态码 状态码描述
请求头	key: value key: value key: value ...	响应头	key: value key: value key: value ...
空行	空行	空行	空行
请求正文	请求相关的信息和数据	响应正文	HTML/图片/JSON等类型数据

# HTTP头字段

表达请求/响应的属性/需求等信息

**头字段** 由键值对key:value组成，字段间使用CRLF分隔(\r\n)

**Content-Length: 127**

**字段名称** 不区分大小写，任意大小写的形式都是合法的

**Content-Length**      **content-length**      **c0nteNt-lEnGTh**

**字段值名称** 区分大小写

**Cookie: cf\_clearance=tKTgM...S4QIw**      ≠      **Cookie: cf\_clearance=tktgm...d4qiw**

一些头字段在请求头/响应头均可使用，而一些仅能在请求/响应头中使用

# HTTP头字段

实验中，需要支持Content-Length和Content-Type字段的处理

## Content-Length

有正文时通常必须添加该字段，指明资源主体的大小，以字节为单位，对于接收方预估下载时间和分配缓冲区大小非常有用

## Content-Type

有正文时通常必须添加该字段，用于指定资源的MIME类型，帮助接收方正确解析资源

类型	字段	描述
文本	text/plain	纯文本文件
文本	text/html	HTML文档
文本	text/css	CSS样式表
应用程序	application/json	JSON数据

类型	字段	描述
图像	image/jpeg	JPEG图像
图像	image/png	PNG图像
视频	video/mp4	MP4视频文件
音频	audio/mpeg	MP3音频文件

# PART 03

## 开发步骤

DEVELOPMENT STEPS

# HTTP协议格式

请求行	请求方法   URI   HTTP版本	状态行	HTTP版本   状态码   状态码描述
请求头	key: value key: value key: value ...	响应头	key: value key: value key: value ...
空行	空行	空行	空行
请求正文	请求相关的信息和数据	响应正文	HTML/图片/JSON等类型数据

# 请求解析&响应组装

```
● ● ●  
  
class HTTPRequest {  
public:  
    HTTPRequest(std::string& is); // Ctor - Parse input string & construct  
    // Optional - Reload [] for easier header fields access ?  
    const std::string& operator[](const std::string& key) const { }  
    // Getters - Retrive info from object ...  
private:  
    // Necessary data elements  
};  
class HTTPResponse {  
public:  
    HTTPResponse(const std::string& version, int code, const std::string& reasonPhrase);  
    std::string serialize() const; // Serialize to bytestream for send  
    // Optional - Reload [] for easier header fields access & modification ?  
    std::string& operator[](const std::string& key) { }  
    // Setters - modify response ...  
private:  
    // Necessary data elements  
};
```

# 请求处理



```
void getHandler(info, response) { /* modify response obj based on info */ }
...
// retrive request and dispatch tasks
void connectionHandler(int socket) {
    while (!shouldExit) {
        // receive & parse full HTTP request
        HTTPRequest request(receivedMsg);
        HTTPResponse response; // handle request according to method, construct response
        if (request.getMethod() == "GET") getHandler(info, response);
        // construct response
        std::string byteStream = response.serialize();
        // return response & close the connection
    }
}
```

# 请求处理

## 接收&解析请求

- GET请求    映射URI→相应文件  
                  映射**存在** →返回200    映射**不存在**→返回404
- POST请求    检查URI是否为/dopost  
                  **是** 状态码200  
                  账号密码**正确**→返回成功信息    账号密码**错误**→返回失败信息  
                  **否** 状态码404

# 测试与检验

## 仅作为实现中特定阶段功能的验证

测试1 Hello World Web服务器

测试2 HTTP请求结构解析

测试3 服务器的WebEcho功能

测试4 资源URI映射

※完成每一阶段新功能的实现后  
上一阶段测试不通过是正常现象

## 需要能够通过测试，按照实验报告要求填写报告

测试5 资源访问

※完成全部功能实现后  
该部分的测试均需通过

测试6 用户登录测试

测试7 多线程访问测试

# PART 04

## 延伸內容

EXTENDED CONTENT

# HTTP协议，不止网页加载



什么触发了请求？



**DNS解析**: 浏览器通过DNS解析得到相应服务器IP地址

**建立连接**: 浏览器根据IP通过TCP/IP (HTTP/3改用QUIC) 协议与服务器建立连接

**发送请求**: 浏览器构建HTTP请求报文，通过已建立的连接发送给服务器

**处理请求**: 服务器根据请求内容执行相应业务逻辑 (如查询数据库等) 并返回响应

**关闭连接**: 完成请求后，关闭连接 (时机与HTTP版本相关)



Axios向127.0.0.1:5000/test/lab8/resource-retrieve发送POST请求，正文为含被测服务器等信息的JSON

Flask根据定义的路由处理请求，将其分发至对应处理函数，执行预定义的操作，并返回指定响应码/数据

前端解析响应数据，得到测试结果，向用户进行展示

# 单页应用SPA



一种网络应用程序或网站的模型，与动态页面通过**动态重写**当前页面来与用户交互，而不是传统的从服务器重新加载整个新页面

这种方法避免了页面之间切换打断用户体验，使应用程序更像一个桌面应用程序

## 工作原理

SPA中所有必要的代码（HTML、JavaScript 和 CSS）都在单个页面的加载过程中完成，随后JavaScript脚本开始接管页面的渲染

这之后，用户和页面交互时，点击链接/提交表单等操作不会加载新页面，而是通过AJAX请求向服务器请求数据，再用JavaScript动态局部更新页面

SPA通常前后端分离，两者独立开发/部署/扩展

## 前端常用框架

React.js、Vue.js、Angular等



## 后端常用框架

SpringBoot[Java]、Django[Python]、Flask[Python]、Node.js[JavaScript]

## 前后端开发最佳实践

- **前端**（或后端）确定接口文档初稿
- **前后端** 评审、完善接口文档，设计接口用例
- **前端** 使用根据接口文档生成的 Mock 数据开发
- **后端** 使用接口用例 调试开发中接口，只要所有接口用例调试通过，接口就开发完成
- **后端** 每次调试完一个功能就保存为一个接口用例
- **测试人员** 直接使用接口用例测试接口
- **测试人员** 所有接口开发完成后，进行多接口集成测试，完整测试整个接口调用流程
- **前后端** 都开发完，前端从Mock 数据切换到正式数据进行联调

# 参考文献与声明

- [1] HTTP - Wikipedia. <https://en.wikipedia.org/wiki/HTTP>
- [2] HTTP messages – HTTP | MDN. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages>
- [3] URL - Wikipedia. <https://en.wikipedia.org/wiki/URL>
- [4] Uniform Resource Name - Wikipedia. [https://en.wikipedia.org/wiki/Uniform\\_Resource\\_Name](https://en.wikipedia.org/wiki/Uniform_Resource_Name)
- [5] MIME types – HTTP | MDN. [https://developer.mozilla.org/en-US/docs/Web/HTTP/MIME\\_types](https://developer.mozilla.org/en-US/docs/Web/HTTP/MIME_types)
- [6] HTTP response status codes – HTTP | MDN. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>
- [7] HTTP headers – HTTP | MDN. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>
- [8] 最佳实践|Apifox帮助文档. <https://apifox.com/help/overview/best-practice>

本课件内部分内容来源于上述参考文献中维基百科页面，原文由多位贡献者编写，具体可访问相应链接查看完整编辑历史与贡献者列表。

根据知识共享署名-相同方式共享4.0国际许可协议 (CC BY-SA 4.0)，本课件也采用相同的许可协议发布。有关此许可协议的详细信息，请参阅 Creative Commons Attribution-ShareAlike 4.0 International License. <https://creativecommons.org/licenses/by-sa/4.0/>

