5-1 加餐 - HTTP cookie 与 session

一种关于登录的场景演示 - B 站登录和未登录

- 问题：B 站是如何认识我这个登录用户的？
- 问题：HTTP 是无状态，无连接的，怎么能够记住我？

引入 HTTP Cookie

定义

HTTP Cookie（也称为 Web Cookie、浏览器 Cookie 或简称 Cookie）是服务器发送到用户浏览器并保存在浏览器上的一小块数据，它会在浏览器之后向同一服务器再次发起请求时被携带并发送到服务器上。通常，它用于告知服务端两个请求是否来自同一浏览器，如保持用户的登录状态、记录用户偏好等。

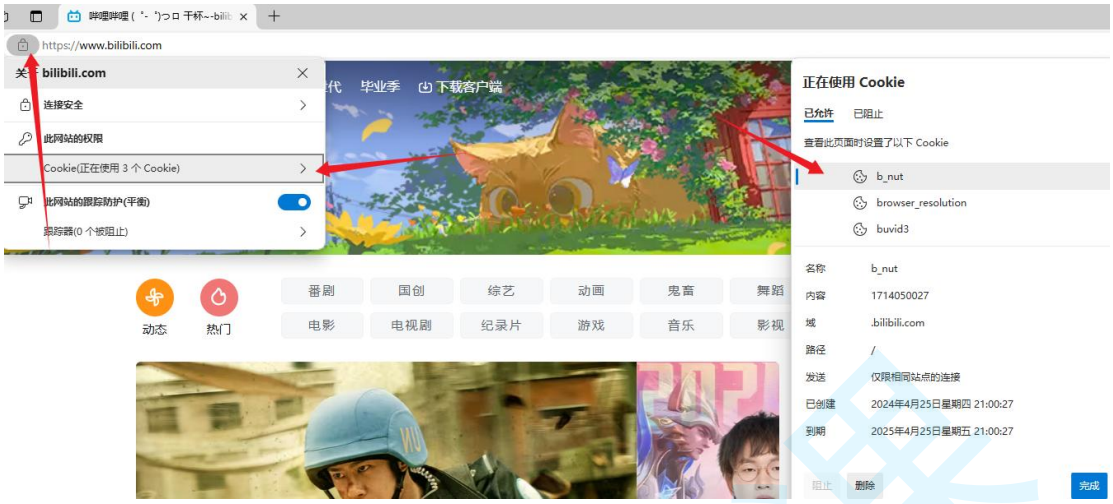
工作原理

- 当用户第一次访问网站时，服务器会在响应的 HTTP 头中设置 `Set-Cookie` 字段，用于发送 Cookie 到用户的浏览器。
- 浏览器在接收到 Cookie 后，会将其保存在本地（通常是按照域名进行存储）。
- 在之后的请求中，浏览器会自动在 `HTTP` 请求头中携带 `Cookie` 字段，将之前保存的 Cookie 信息发送给服务器。

分类

- 会话 `Cookie（Session Cookie）`：在浏览器关闭时失效。
- 持久 `Cookie（Persistent Cookie）`：带有明确的过期日期或持续时间，可以跨多个浏览器会话存在。
- 如果 `cookie` 是一个持久性的 `cookie`，那么它其实就是浏览器相关的，特定目录下的一个文件。但直接查看这些文件可能会看到乱码或无法读取的内容，因为 `cookie` 文件通常以二进制或 `sqlite` 格式存储。一般我们查看，直接在浏览器对应的选项中直接查看即可。

类似于下面这种方式：

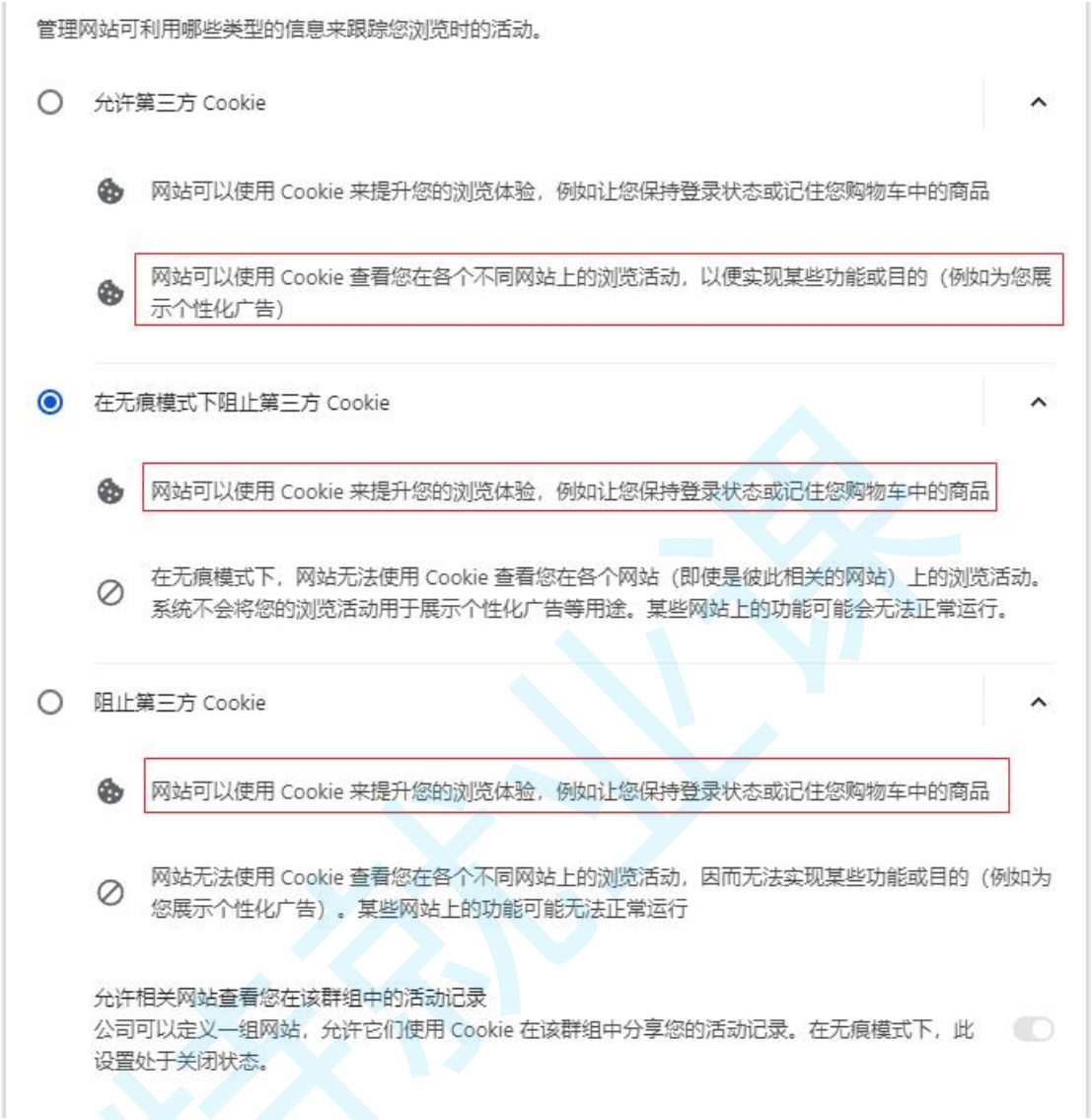


安全性

- 由于`Cookie`是存储在客户端的，因此存在被篡改或窃取的风险。

用途

- 用户认证和会话管理(最重要)
- 跟踪用户行为
- 缓存用户偏好等
- 比如在 chrome 浏览器下，可以直接访问：chrome://settings/cookies



认识 cookie

- HTTP 存在一个报头选项：Set-Cookie，可以用来进行给浏览器设置 Cookie 值。
- 在 HTTP 响应头中添加，客户端（如浏览器）获取并自行设置并保存 Cookie。

基本格式

```
C++
Set-Cookie: <name>=<value>
其中 <name> 是 Cookie 的名称，<value> 是 Cookie 的值。
```

完整的 Set-Cookie 示例

```C++
Set-Cookie: username=peter; expires=Thu, 18 Dec 2024 12:00:00 UTC; path=/; domain=.example.com; secure; HttpOnly
```

时间格式必须遵守 **RFC 1123** 标准，具体格式样例：`Tue, 01 Jan 2030 12:34:56 GMT` 或者 `UTC`(推荐)。

关于时间解释

- **Tue**: 星期二（星期几的缩写）
- **,**: 逗号分隔符
- **01**: 日期（两位数表示）
- **Jan**: 一月（月份的缩写）
- **2030**: 年份（四位数）
- **12:34:56**: 时间（小时、分钟、秒）
- **GMT**: 格林威治标准时间（时区缩写）

GMT vs UTC --- 以下内容，均来自文心一言，了解即可

GMT（格林威治标准时间）和 UTC（协调世界时）是两个不同的时间标准，但它们在大多数情况下非常接近，常常被混淆。以下是两者的简单解释和区别：

1. GMT（格林威治标准时间）：
   - GMT 是格林威治标准时间的缩写，它是以英国伦敦的格林威治区为基准的世界时间标准。
   - GMT 不受夏令时或其他因素的影响，通常用于航海、航空、科学、天文等领域。
   - GMT 的计算方式是基于地球的自转和公转。
2. UTC（协调世界时）：
   - UTC 全称为“协调世界时”，是国际电信联盟(ITU)制定和维护的标准时间。
   - UTC 的计算方式是基于原子钟，而不是地球的自转，因此它比 GMT 更准确。据称，世界上最精确的原子钟 50 亿年才会误差 1 秒。
   - UTC 是现在用的时间标准，多数全球性的网络和软件系统将其作为标准时间。

GMT 和 UTC 的英文全称以及相关信息如下：

1. GMT（格林尼治标准时间）
    - 英文全称：Greenwich Mean Time
    - GMT 是指位于英国伦敦郊区的皇家格林尼治天文台的标准时间，因为本初子午线被定义为通过那里的经线。理论上来说，格林尼治标准时间的正午是指当太阳横穿格林尼治子午线时的时间。
    - 但值得注意的是，地球的自转是有些不规则的，且正在缓慢减速。因此，格林尼治时间已经不再被作为标准时间使用。
2. UTC（协调世界时）
    - 英文全称：Coordinated Universal Time
    - UTC 是最主要的世界时间标准，其以原子时秒长为基础，在时刻上尽量接近于格林尼治标准时间。
    - UTC 被广泛使用在计算机网络、航空航天等领域，因为它提供了非常准确和可靠的时间参考。

总结来说，GMT 和 UTC 都曾是或现在是国际上重要的时间标准，但由于地球自转的不规则性和原子钟的精确性，**UTC 已经成为了全球性的标准时间**，而 GMT 则更多被用作历史和地理上的参考。

区别：

- 计算方式：GMT 基于地球的自转和公转，而 UTC 基于原子钟。
- 准确度：由于 UTC 基于原子钟，它比基于地球自转的 GMT 更加精确。

在实际使用中，GMT 和 UTC 之间的差别通常很小，大多数情况下可以互换使用。但在需要高精度时间计量的场合，如科学研究、网络通信等，UTC 是更为准确的选择。

关于其他可选属性的解释

- `expires=<date>`[要验证]：设置 Cookie 的过期日期/时间。如果未指定此属性，则 Cookie 默认为会话 Cookie，即当浏览器关闭时过期。
- `path=<some_path>`[要验证]：限制 Cookie 发送到服务器的哪些路径。默认为设置它的路径。
- `domain=<domain_name>`[了解即可]：指定哪些主机可以接受该 Cookie。默认为设置它的主机。
- `secure`[了解即可]：仅当使用 HTTPS 协议时才发送 Cookie。这有助于防止 Cookie 在不安全的 HTTP 连接中被截获。

- `HttpOnly`[了解即可]：标记 Cookie 为 HttpOnly，意味着该 Cookie 不能被客户端脚本（如 JavaScript）访问。这有助于防止跨站脚本攻击（XSS）。

以下是对 `Set-Cookie` 头部字段的简洁介绍

属性	值	描述
`username`	**`peter`**	**这是 Cookie 的名称和值，标识用户名为"peter"。**
`expires`	**`Thu, 18 Dec 2024 12:00:00 UTC`**	**指定 Cookie 的过期时间。在这个例子中，Cookie 将在 2024 年 12 月 18 日 12:00:00 UTC 后过期。**
`path`	**`/`**	**定义 Cookie 的作用范围。这里设置为根路径`/`，意味着 Cookie 对`.example.com`域名下的所有路径都可用。**
`domain`	`.example.com`	指定哪些域名可以接收这个 Cookie。点前缀（`.`）表示包括所有子域名。
`secure`	-	指示 Cookie 只能通过 HTTPS 协议发送，不能通过 HTTP 协议发送，增加安全性。
`HttpOnly`	-	阻止客户端脚本（如 JavaScript）访问此 Cookie，有助于防止跨站脚本攻击（XSS）。

注意事项

- 每个 Cookie 属性都以分号（`;`）和空格（ ）分隔。
- 名称和值之间使用等号（`=`）分隔。
- 如果 Cookie 的名称或值包含特殊字符（如空格、分号、逗号等），则需要进行 URL 编码。

Cookie 的生命周期

- 如果设置了 `expires` 属性，则 `Cookie` 将在指定的日期/时间后过期。
- 如果没有设置 `expires` 属性，则 `Cookie` 默认为会话 Cookie，即当浏览器关闭时过期。

安全性考虑[了解即可]

- 使用 `secure` 标志可以确保 Cookie 仅在 HTTPS 连接上发送，从而提高安全性。
- 使用 `HttpOnly` 标志可以防止客户端脚本（如 JavaScript）访问 Cookie，从而防止 XSS 攻击。
- 通过合理设置 Set-Cookie 的格式和属性，可以确保 Cookie 的安全性、有效性和可访问性，从而满足 Web 应用程序的需求。

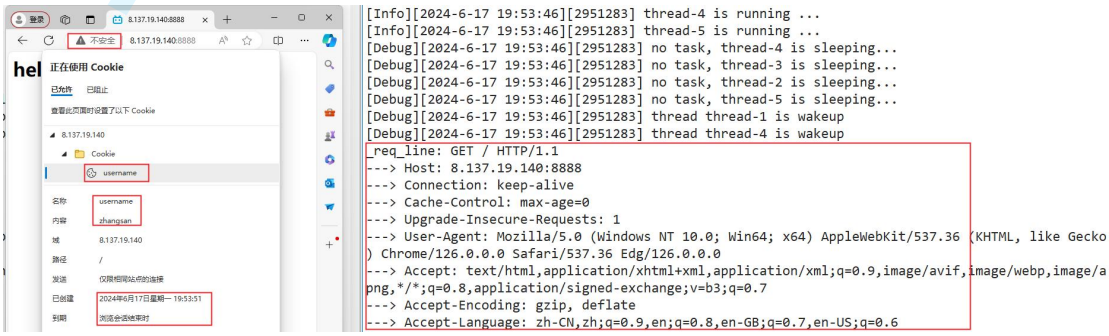
实验测试 cookie

- 测试 `cookie` 的代码：https://gitee.com/whb-helloworld/linux-plus-meal/tree/master/http-cookie-session/cookie
- chrome 浏览器查看 cookie 不方便，我们推荐使用 windows 自带浏览器：

- 代码文件结构

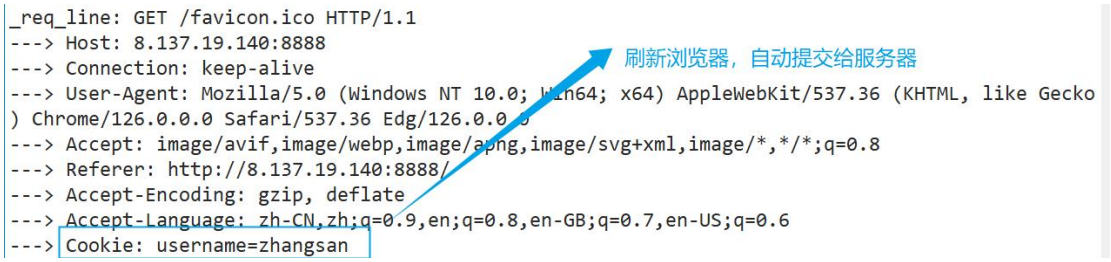
```
C++
Comm.hpp  HttpProtocol.hpp  InetAddr.hpp  LockGuard.hpp  Log.hpp
Main.cc  Makefile  Socket.hpp  TcpServer.hpp  Thread.hpp
ThreadPool.hpp
```

测试 `cookie` 写入到浏览器



测试自动提交

- 刷新浏览器，刚刚写入的 cookie 会自动被提交给服务器端



测试写入过期时间

- 这里要由我们自己形成 UTC 统一标准时间，下面是对应的 C++样例代码,以供参考

```
C++
    std::string GetMonthName(int month)
    {
        std::vector<std::string> months = {"Jan", "Feb", "Mar",
"Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
        return months[month];
    }
    std::string GetWeekDayName(int day)
    {
        std::vector<std::string> weekdays = {"Sun", "Mon", "Tue",
"Wed", "Thu", "Fri", "Sat"};
        return weekdays[day];
    }
    std::string ExpireTimeUseRfc1123(int t) // 秒级别的未来 UTC 时间
    {
        time_t timeout = time(nullptr) + t;
        // 这个地方有坑哦
        struct tm *tm = gmtime(&timeout); // 这里不能用 localtime，
因为 localtime 是默认带了时区的. gmtime 获取的就是 UTC 统一时间
        char timebuffer[1024];
        //时间格式如: expires=Thu, 18 Dec 2024 12:00:00 UTC
        snprintf(timebuffer, sizeof(timebuffer),
"%s, %02d %s %d %02d:%02d:%02d UTC",
            GetWeekDayName(tm->tm_wday).c_str(),
            tm->tm_mday,
            GetMonthName(tm->tm_mon).c_str(),
            tm->tm_year+1900,
            tm->tm_hour,
```

```
            tm->tm_min,
            tm->tm_sec
        );
        return timebuffer;
    }
```

测试路径 path

提交到非/a/b 路径下

- 比如：http://8.137.19.140:8888/a/x
- 比如：http://8.137.19.140:8888/
- 比如：http://8.137.19.140:8888/x/y

```
_req_line: GET /a/x HTTP/1.1
---> Host: 8.137.19.140:8888
---> Connection: keep-alive
---> Upgrade-Insecure-Requests: 1
---> User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko
) Chrome/126.0.0.0 Safari/537.36 Edg/126.0.0.0
---> Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a
png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
---> Accept-Encoding: gzip, deflate
---> Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6
```

没有任何cookie信息提交!

提交到/a/x

提交到/a/b 路径下

```
_req_line: GET /a/b HTTP/1.1
---> Host: 8.137.19.140:8888
---> Connection: keep-alive
---> Upgrade-Insecure-Requests: 1
---> User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko
) Chrome/126.0.0.0 Safari/537.36 Edg/126.0.0.0
---> Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a
png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
---> Accept-Encoding: gzip, deflate
---> Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6
---> Cookie: username=zhangsan
```

提交到/a/b路径下，Cookie被浏览器自动提交

附上部分核心代码

```C++
class Http
{
private:
    std::string GetMonthName(int month)
    {
        std::vector<std::string> months = {"Jan", "Feb", "Mar",
"Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
        return months[month];
    }
    std::string GetWeekDayName(int day)
    {
        std::vector<std::string> weekdays = {"Sun", "Mon", "Tue",
"Wed", "Thu", "Fri", "Sat"};
        return weekdays[day];
    }
    std::string ExpireTimeUseRfc1123(int t) // 秒级别的未来 UTC 时间
    {
        time_t timeout = time(nullptr) + t;
        struct tm *tm = gmtime(&timeout); // 这里不能用 localtime，
因为 localtime 是默认带了时区的. gmtime 获取的就是 UTC 统一时间
        char timebuffer[1024];
        //时间格式如: expires=Thu, 18 Dec 2024 12:00:00 UTC
        snprintf(timebuffer, sizeof(timebuffer),
"%s, %02d %s %d %02d:%02d:%02d UTC",
            GetWeekDayName(tm->tm_wday).c_str(),
```

```
            tm->tm_mday,
            GetMonthName(tm->tm_mon).c_str(),
            tm->tm_year+1900,
            tm->tm_hour,
            tm->tm_min,
            tm->tm_sec
        );
        return timebuffer;
    }
public:
    Http(uint16_t port)
    {
        _tsvr = std::make_unique<TcpServer>(port,
std::bind(&Http::HandlerHttp, this, std::placeholders::_1));
        _tsvr->Init();
    }
    std::string ProveCookieWrite() // 证明 cookie 能被写入浏览器
    {
        return "Set-Cookie: username=zhangsan;";
    }
    std::string ProveCookieTimeOut()
    {
        return "Set-Cookie: username=zhangsan; expires=" +
ExpireTimeUseRfc1123(60) + ";"; // 让 cookie 1min 后过期
    }
    std::string ProvePath()
    {
        return "Set-Cookie: username=zhangsan; path=/a/b;";
    }
    std::string HandlerHttp(std::string request)
    {
        HttpRequest req;
        req.Deserialize(request);
        req.DebugHttp();
        lg.LogMessage(Debug, "%s\n",
ExpireTimeUseRfc1123(60).c_str());
        HttpResponse resp;
        resp.SetCode(200);
        resp.SetDesc("OK");
        resp.AddHeader("Content-Type: text/html");
        //resp.AddHeader(ProveCookieWrite()); //测试 cookie 被写入与
自动提交
        //resp.AddHeader(ProveCookieTimeOut()); //测试过期时间的写入
        resp.AddHeader(ProvePath()); // 测试路径
```

```
        resp.AddContent("<html><h1>helloworld</h1></html>");
        return resp.Serialize();
    }
    void Run()
    {
        _tsvr->Start();
    }
    ~Http()
    {}
private:
    std::unique_ptr<TcpServer> _tsvr;
};
```

- 通过注释绿色区域，即可完成测试

单独使用 Cookie，有什么问题?

- 我们写入的是测试数据，如果写入的是用户的私密数据呢？比如，用户名密码，浏览痕迹等。
- 本质问题在于这些用户私密数据在浏览器(用户端)保存，非常容易被人盗取，更重要的是，除了被盗取，还有就是用户私密数据也就泄漏了。

引入 HTTP Session

定义

HTTP Session 是服务器用来跟踪用户与服务器交互期间用户状态的机制。由于 HTTP 协议是无状态的（每个请求都是独立的），因此服务器需要通过 Session 来记住用户的信息。

工作原理

当用户首次访问网站时，服务器会为用户创建一个唯一的 `Session ID`，并通过 `Cookie` 将其发送到客户端。

客户端在之后的请求中会携带这个 `Session ID`，服务器通过 `Session ID` 来识别用户，从而获取用户的会话信息。

服务器通常会将 `Session` 信息存储在内存、数据库或缓存中。

安全性：

与 `Cookie` 相似，由于 `Session ID` 是在客户端和服务器之间传递的，因此也存在被窃取的风险。

但是一般虽然 `Cookie` 被盗取了，但是用户只泄漏了一个 `Session ID`，私密信息暂时没有被泄露的风险

Session ID 便于服务端进行客户端有效性的管理，比如异地登录。

可以通过 `HTTPS` 和设置合适的 `Cookie` 属性（如 `HttpOnly` 和 `Secure`）来增强安全性。

超时和失效：

Session 可以设置超时时间，当超过这个时间后，Session 会自动失效。

服务器也可以主动使 Session 失效，例如当用户登出时。

用途：

用户认证和会话管理

存储用户的临时数据（如购物车内容）

实现分布式系统的会话共享（通过将会话数据存储在共享数据库或缓存中）

模拟 session 行为

测试 session 代码链接：https://gitee.com/whb-helloworld/linux-plus-meal/tree/master/http-cookie-session/session

代码文件结构

```
C++
Comm.hpp  HttpProtocol.hpp  InetAddr.hpp  LockGuard.hpp  Log.hpp
Main.cc  Makefile  Session.hpp  Socket.hpp  TcpServer.hpp
Thread.hpp  ThreadPool.hpp
```

部分核心代码：

Session.hpp

```
C++
#pragma once
```

```
#include <iostream>
#include <string>
#include <memory>
#include <ctime>
#include <unistd.h>
#include <unordered_map>

// 用来进行测试说明
class Session
{
public:
    Session(const std::string &username, const std::string
&status)
    :_username(username), _status(status)
    {
        _create_time = time(nullptr); // 获取时间戳就行了，后面实际需
要，就转化就转换一下
    }
    ~Session()
    {}
public:
    std::string _username;
    std::string _status;
    uint64_t _create_time;
    //当然还可以再加任何其他信息，看你的需求
};

using session_ptr = std::shared_ptr<Session>;

class SessionManager
{
public:
    SessionManager()
    {
        srand(time(nullptr) ^ getpid());
    }
    std::string AddSession(session_ptr s)
    {
        uint32_t randomid = rand() + time(nullptr); // 随机数+时间
戳，实际有形成 sessionid 的库，比如 boost uuid 库，或者其他第三方库等
        std::string sessionid = std::to_string(randomid);
        _sessions.insert(std::make_pair(sessionid, s));
        return sessionid;
    }
```

```
    session_ptr GetSession(const std::string sessionid)
    {
        if(_sessions.find(sessionid) == _sessions.end()) return
nullptr;
        return _sessions[sessionid];
    }
    ~SessionManager()
    {}
private:
    std::unordered_map<std::string, session_ptr> _sessions;
};
```

HttpProtocol.hpp

```
C++
#pragma once

#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <memory>
#include <ctime>
#include <functional>
#include "TcpServer.hpp"
#include "Session.hpp" // 引入 session

const std::string HttpSep = "\r\n";
// 可以配置的
const std::string homepage = "index.html";
const std::string wwwroot = "./wwwroot";

class HttpRequest
{
public:
    HttpRequest() : _req_blank(HttpSep), _path(wwwroot)
    {
    }
    bool GetLine(std::string &str, std::string *line)
    {
        auto pos = str.find(HttpSep);
        if (pos == std::string::npos)
            return false;
        *line = str.substr(0, pos); // \r\n
```

```
        str.erase(0, pos + HttpSep.size());
        return true;
    }
    void Parse()
    {
        // 解析出来 url
        std::stringstream ss(_req_line);
        ss >> _method >> _url >> _http_version;

        // 查找 cookie
        std::string prefix = "Cookie: ";
        for (auto &line : _req_header)
        {
            std::string cookie;
            if (strncmp(line.c_str(), prefix.c_str(),
prefix.size()) == 0) // 找到了
            {
                cookie = line.substr(prefix.size()); // 截取
"Cookie: "之后的就行了
                _cookies.emplace_back(cookie);
                break;
            }
        }

        // 查找 sessionid
        prefix = "sessionid=";
        for (const auto &cookie : _cookies)
        {
            if (strncmp(cookie.c_str(), prefix.c_str(),
prefix.size()) == 0)
            {
                _sessionid = cookie.substr(prefix.size()); // 截取
"sessionid="之后的就行了
                // std::cout << "_sessionid: " << _sessionid <<
std::endl;
            }
        }
    }
    std::string Url()
    {
        return _url;
    }
    std::string SessionId()
```

```
    {
        return _sessionid;
    }
    bool Deserialize(std::string &request)
    {
        std::string line;
        bool ok = GetLine(request, &line);
        if (!ok)
            return false;
        _req_line = line;

        while (true)
        {
            bool ok = GetLine(request, &line);
            if (ok && line.empty())
            {
                _req_content = request;
                break;
            }
            else if (ok && !line.empty())
            {
                _req_header.push_back(line);
            }
            else
            {
                break;
            }
        }

        return true;
    }
    void DebugHttp()
    {
        std::cout << "_req_line: " << _req_line << std::endl;
        for (auto &line : _req_header)
        {
            std::cout << "---> " << line << std::endl;
        }
    }
    ~HttpRequest()
    {
    }

private:
```

```
    // http 报文自动
    std::string _req_line; // method url http_version
    std::vector<std::string> _req_header;
    std::string _req_blank;
    std::string _req_content;

    // 解析之后的内容
    std::string _method;
    std::string _url; // /   /dira/dirb/x.html
/dira/dirb/XX?usrname=100&&password=1234 /dira/dirb
    std::string _http_version;
    std::string _path;                  // "./wwwroot"
    std::string _suffix;                // 请求资源的后缀
    std::vector<std::string> _cookies; // 其实 cookie 可以有多个，因为 Set-Cookie 可以被写多条，测试，一条够了。
    std::string _sessionid;             // 请求携带的 sessionid，仅仅用来测试
};

const std::string BlankSep = " ";
const std::string LineSep = "\r\n";

class HttpResponse
{
public:
    HttpResponse() : _http_version("HTTP/1.0"), _status_code(200),
_status_code_desc("OK"), _resp_blank(LineSep)
    {
    }
    void SetCode(int code)
    {
        _status_code = code;
    }
    void SetDesc(const std::string &desc)
    {
        _status_code_desc = desc;
    }
    void MakeStatusLine()
    {
        _status_line = _http_version + BlankSep +
std::to_string(_status_code) + BlankSep + _status_code_desc +
LineSep;
    }
    void AddHeader(const std::string &header)
```

```
    {
        _resp_header.push_back(header + LineSep);
    }
    void AddContent(const std::string &content)
    {
        _resp_content = content;
    }
    std::string Serialize()
    {
        MakeStatusLine();
        std::string response_str = _status_line;
        for (auto &header : _resp_header)
        {
            response_str += header;
        }
        response_str += _resp_blank;
        response_str += _resp_content;

        return response_str;
    }
    ~HttpResponse() {}

private:
    std::string _status_line;
    std::vector<std::string> _resp_header;
    std::string _resp_blank;
    std::string _resp_content; // body

    // httpversion StatusCode StatusCodeDesc
    std::string _http_version;
    int _status_code;
    std::string _status_code_desc;
};

class Http
{
private:
    std::string GetMonthName(int month)
    {
        std::vector<std::string> months = {"Jan", "Feb", "Mar",
"Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
        return months[month];
    }
    std::string GetWeekDayName(int day)
```

```
    {
        std::vector<std::string> weekdays = {"Sun", "Mon", "Tue",
"Wed", "Thu", "Fri", "Sat"};
        return weekdays[day];
    }
    std::string ExpireTimeUseRfc1123(int t) // 秒级别的未来 UTC 时间
    {
        time_t timeout = time(nullptr) + t;
        struct tm *tm = gmtime(&timeout); // 这里不能用 localtime，
因为 localtime 是默认带了时区的. gmtime 获取的就是 UTC 统一时间
        char timebuffer[1024];
        // 时间格式如: expires=Thu, 18 Dec 2024 12:00:00 UTC
        snprintf(timebuffer, sizeof(timebuffer),
"%s, %02d %s %d %02d:%02d:%02d UTC",
                 GetWeekDayName(tm->tm_wday).c_str(),
                 tm->tm_mday,
                 GetMonthName(tm->tm_mon).c_str(),
                 tm->tm_year + 1900,
                 tm->tm_hour,
                 tm->tm_min,
                 tm->tm_sec);
        return timebuffer;
    }

public:
    Http(uint16_t port)
    {
        _tsvr = std::make_unique<TcpServer>(port,
std::bind(&Http::HandlerHttp, this, std::placeholders::_1));
        _tsvr->Init();

        _session_manager = std::make_unique<SessionManager>();
    }
    std::string ProveCookieWrite() // 证明 cookie 能被写入浏览器
    {
        return "Set-Cookie: username=zhangsan;";
    }
    std::string ProveCookieTimeOut()
    {
        return "Set-Cookie: username=zhangsan; expires=" +
ExpireTimeUseRfc1123(60) + ";"; // 让 cookie 1min 后过期
    }
    std::string ProvePath()
    {
```

```
        return "Set-Cookie: username=zhangsan; path=/a/b;";
    }
    std::string ProveSession(const std::string &session_id)
    {
        return "Set-Cookie: sessionid=" + session_id + ";";
    }
    std::string HandlerHttp(std::string request)
    {
        HttpRequest req;
        HttpResponse resp;

        req.Deserialize(request);
        req.Parse();
        // req.DebugHttp();
        // std::cout << req.Url() << std::endl;

        // 下面的代码就用来测试，如果你想更优雅，可以回调出去处理
        static int number = 0;
        if (req.Url() == "/login") // 用/login path 向指定浏览器写入
sessionid，并在服务器维护对应的 session 对象
        {
            std::string sessionid = req.SessionId();
            if (sessionid.empty()) // 说明历史没有登陆过
            {
                std::string user = "user-" +
std::to_string(number++);
                session_ptr s = std::make_shared<Session>(user,
"logined");
                std::string sessionid = _session_manager-
>AddSession(s);
                lg.LogMessage(Debug, "%s 被添加，sessionid
是: %s\n", user.c_str(), sessionid.c_str());
                resp.AddHeader(ProveSession(sessionid));
            }
        }
        else
        {
            // 当浏览器在本站点任何路径中活跃，都会自动提交 sessionid，
我们就能知道谁活跃了.
            std::string sessionid = req.SessionId();
            if (!sessionid.empty())
            {
                session_ptr s = _session_manager-
```

```
>GetSession(sessionid);
                // 这个地方有坑，一定要判断服务器端 session 对象是否存在，因为可能测试的时候
                // 浏览器还有历史 sessionid，但是服务器重启之后，session 对象没有了.
                if(s != nullptr)
                    lg.LogMessage(Debug, "%s 正在活跃.\n", s->_username.c_str());
                else
                    lg.LogMessage(Debug, "cookie : %s 已经过期，需要清理\n", sessionid.c_str());
            }
        }

        resp.SetCode(200);
        resp.SetDesc("OK");
        resp.AddHeader("Content-Type: text/html");
        // resp.AddHeader(ProveCookieWrite()); //测试 cookie 被写入与自动提交
        // resp.AddHeader(ProveCookieTimeOut()); //测试过期时间的写入
        //  resp.AddHeader(ProvePath()); // 测试路径
        resp.AddContent("<html><h1>helloworld</h1></html>");
        return resp.Serialize();
    }
    void Run()
    {
        _tsvr->Start();
    }
    ~Http()
    {
    }

private:
    std::unique_ptr<TcpServer> _tsvr;
    std::unique_ptr<SessionManager> _session_manager;
};
```

- 绿色部分是新增的变化部分

实验测试 session:

- 准备两个浏览器： Google Chrome 和 Microsoft Edge(windows 自带的)

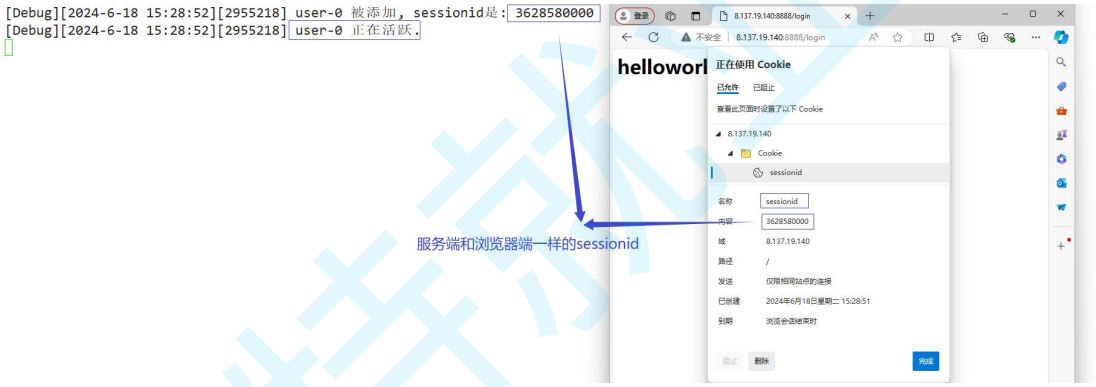
1. **删除浏览器中指定的服务器上的所有的 cookie**
   - 如果历史上没有做过测试，就不删了
   - chrome 的 cookie 有些特殊，实验不出来，尝试打印 chrome 浏览器发过来的 http 请求，观察 cookie 部分，你就能知道为什么要删除历史 cookie。

Microsoft Edge

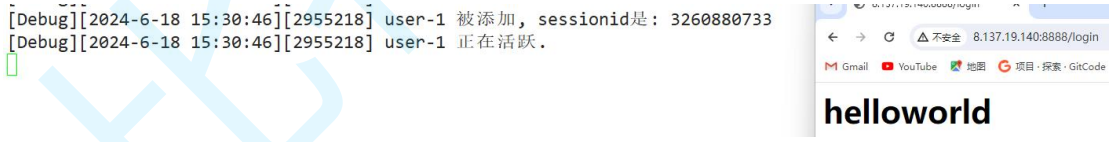
Google Chrome

2. **访问/login, 模拟登录**

Microsoft Edge



Google Chrome



3. **两个浏览器访问任意的站点资源**

```
[Debug][2024-6-18 15:30:46][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:32:31][2955218] user-0 正在活跃.
[Debug][2024-6-18 15:32:31][2955218] user-0 正在活跃.
[Debug][2024-6-18 15:32:44][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:32:45][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:32:48][2955218] user-0 正在活跃.
[Debug][2024-6-18 15:32:48][2955218] user-0 正在活跃.
[Debug][2024-6-18 15:32:49][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:32:49][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:32:49][2955218] user-0 正在活跃.
[Debug][2024-6-18 15:32:49][2955218] user-0 正在活跃.
[Debug][2024-6-18 15:32:50][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:32:50][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:32:52][2955218] user-0 正在活跃.
[Debug][2024-6-18 15:32:52][2955218] user-0 正在活跃.
[Debug][2024-6-18 15:32:52][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:32:52][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:33:16][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:33:16][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:33:17][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:33:17][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:33:17][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:33:17][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:33:17][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:33:17][2955218] user-1 正在活跃.
[Debug][2024-6-18 15:33:17][2955218] user-1 正在活跃.
```

- 服务器端已经能识别是哪一个浏览器了

总结：

HTTP Cookie 和 Session 都是用于在 Web 应用中跟踪用户状态的机制。Cookie 是存储在客户端的，而 Session 是存储在服务器端的。它们各有优缺点，通常在实际应用中会结合使用，以达到最佳的用户体验和安全性。

附录：

- `favicon.ico` 是一个网站图标，通常显示在浏览器的标签页上、地址栏旁边或收藏夹中。这个图标的文件名 `favicon` 是 "favorite icon" 的缩写，而 `.ico` 是图标的文件格式。
- 浏览器在发起请求的时候，也会为了获取图标而专门构建 http 请求，我们不管它。