

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。

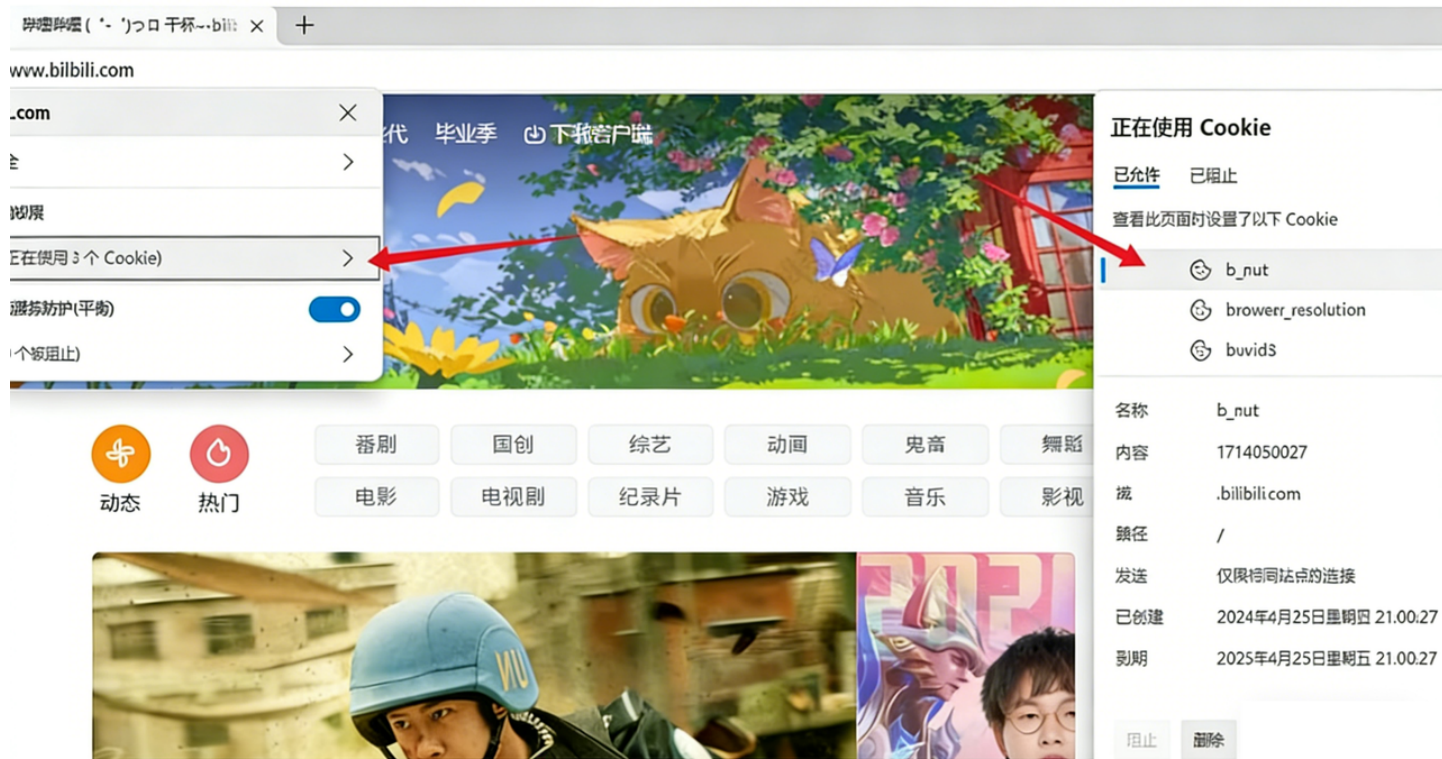
工作原理

- 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`


管理网站可利用哪些类型的信息来跟踪您浏览时的活动。


允许第三方 Cookie

 网站可以使用 Cookie 来提升您的浏览体验，例如让您保持登录状态或记住您购物车中的商品


 网站可以使用 Cookie 查看您在各个不同网站上的浏览活动，以便实现某些功能或目的（例如为您展示个性化广告）


在无痕模式下阻止第三方 Cookie

 网站可以使用 Cookie 来提升您的浏览体验，例如让您保持登录状态或记住您购物车中的商品

 在无痕模式下，网站无法使用 Cookie 查看您在各个网站（即使是彼此相关的网站）上的浏览活动。系统不会将您的浏览活动用于展示个性化广告等用途。某些网站上的功能可能会无法正常运行。

阻止第三方 Cookie

 网站可以使用 Cookie 来提升您的浏览体验，例如让您保持登录状态或记住您购物车中的商品

 网站无法使用 Cookie 查看您在各个不同网站上的浏览活动，因而无法实现某些功能或目的（例如为您展示个性化广告）。某些网站上的功能可能无法正常运行

允许相关网站查看您在该群组中的活动记录

公司可以定义一组网站，允许它们使用 Cookie 在该群组中分享您的活动记录，在无痕模式下，此设置处于关闭状态。



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

代码块

- 1 Set-Cookie: <name>=<value>
- 2 其中 <name> 是 Cookie 的名称，<value> 是 Cookie 的值。

完整的 Set-Cookie 示例

代码块

```
1 Set-Cookie: username=peter; expires=Thu, 18 Dec 2024 12:00:00 UTC; path=/;
2 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

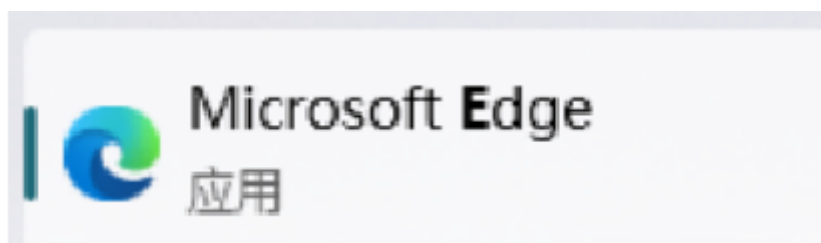
- 如果设置了 expires 属性，则 Cookie 将在指定的日期/时间后过期。
- 如果没有设置 expires 属性，则 Cookie 默认为会话 Cookie，即当浏览器关闭时过期。

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：

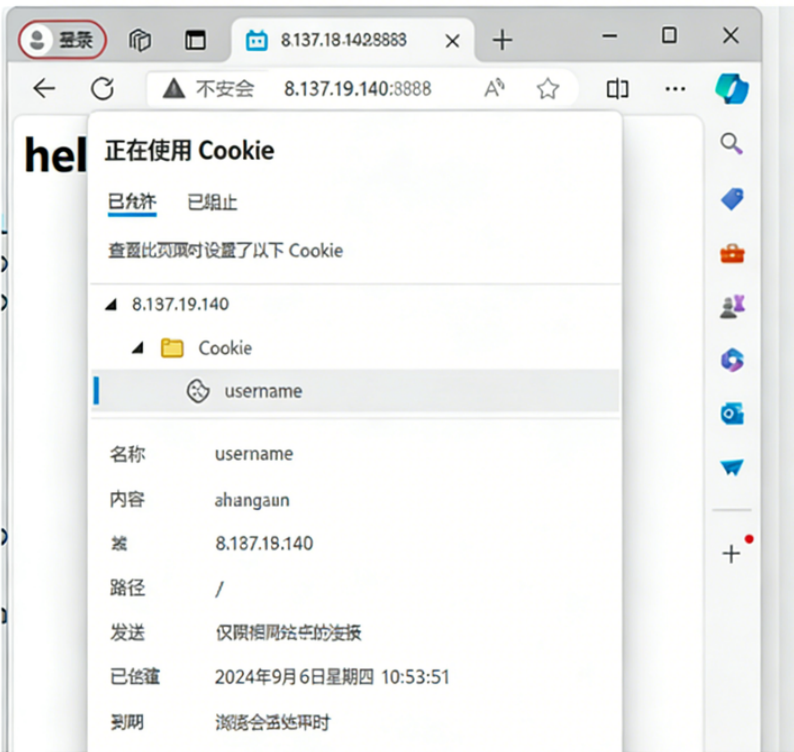


- 代码文件结构

代码块

```
1  Comm.hpp HttpProtocol.hpp InetAddr.hpp LockGuard.hpp Log.hpp Main.cc
2  Makefile Socket.hpp TcpServer.hpp Thread.hpp ThreadPool.hpp
```

测试 `cookie` 写入到浏览器



```
[Info][2024-6-17 19:53:46][2951283] ..
[Info][2024-6-17 19:53:46] is running ..
[Debug][2024-6-17 19:53:46][2951283]
[hebug][2024-6-17 19:53:46][2951283]
[Debug][2024-6-17 19:53:46][2951283]
[Debug][2024-6-17 19:53:46] is ready
[hebug][2024-6-17 19:53:46] is ahead
[Debug][2024-6-17 19:53:46] is running
_req_line: GET / HTTP/1.1
---> Host: 8.137.19.140:8888
---> Connection: keep-alive
---> Cache-Control: max-age=0
---> 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/537.35 Edg/126.0.0.0
---> Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/png,*/*;q=0.8,application/siamge,ex
---> Accept-Encoding: gzip, deflate
---> Accept-Language: zh-CN,zh;q=0.8
```

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
- 刷新浏览器，刚刚写入的cookie会自动被提交给服务器端

```
_req_line: GET /favicon.ico HTTP/1.1
---> Host: 8.137.19.140:8888
---> Connection: keep-alive
---> 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: image/avif,image/webp,image/png,image/svg+xml,image/*,*/*;q=0.8
---> Referer: http://8.137.19.140:8888/
---> 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
```

刷新浏览器，自动提交给服务器

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

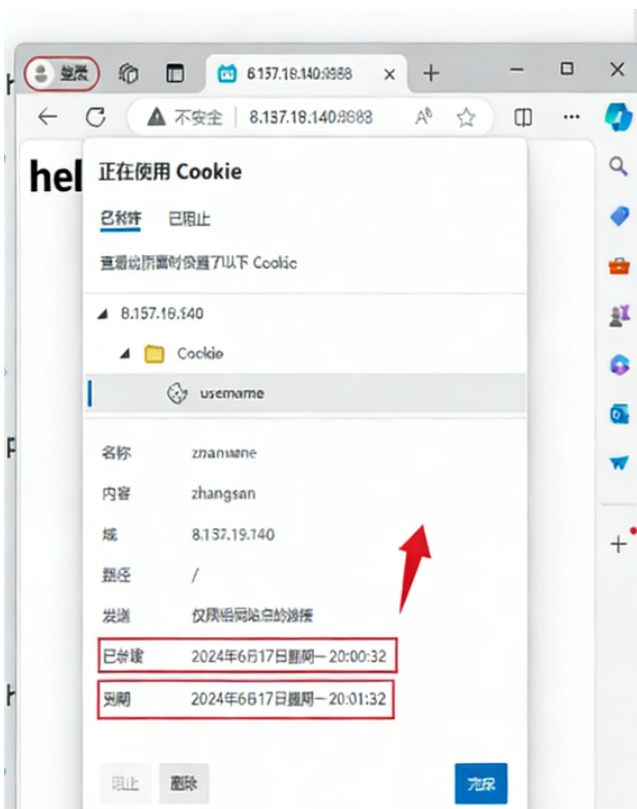
代码块

```
1 std::string GetMonthName(int month)
2 {
3     std::vector<std::string> months = {"Jan", "Feb", "Mar", "Apr", "May",
4                                         "Jun", "Jul", "Aug", "Sep", "Oct",
5                                         "Nov", "Dec"};
6     return months[month];
7 }
8 std::string GetWeekDayName(int day)
```

```

8  {
9      std::vector<std::string> weekdays = {"Sun", "Mon", "Tue", "Wed",
10                                         "Thu", "Fri", "Sat"};
11      return weekdays[day];
12  }
13  std::string ExpireTimeUseRfc1123(int t) // 秒级别的未来UTC时间
14  {
15      time_t timeout = time(nullptr) + t;
16      // 这个地方有坑哦
17      struct tm *tm = gmtime(&timeout); // 这里不能用localtime, 因为localtime
18      是默认带了时区的.gmtime获取的就是UTC统一时间 char timebuffer[1024];
19      // 时间格式如: expires=Thu, 18 Dec 2024 12:00:00 UTC
20      snprintf(timebuffer, sizeof(timebuffer), "%s, %02d %s %d
21                                         % 02d : % 02d : % 02d UTC ",
22
23      GetWeekDayName(tm->tm_wday)
24
25      .c_str(),
26      tm->tm_mday,
27      GetMonthName(tm->tm_mon).c_str(),
28      tm->tm_year + 1900,
29      tm->tm_hour,
30      tm->tm_min,
31      tm->tm_sec);
32      return timebuffer;
33  }

```



```

[Info][2024-6-17 20:0:27][2951376] thread-5 is running
[Debug][2024-6-17 20:0:27][2951376] thread thread-1 is
[Debug][2024-6-17 20:0:27][2951376] no task, thread-1
[Debug][2024-6-17 20:0:27][2951376] thread thread-5 is
_req_line: GET / HTTP/1.1
--> Host: 8.127.19.140:8888
--> Connection: keep-alive
--> Cache-Control: max-age=0
--> Upgrade-Insecure-Requests: 1
--> User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64;
) Chrome/126.0.0.0 Safart/537.36 Edg/126.0.0.0
--> Accept: text/html,application/xhtml+xml,application/javascript;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
[Debug][2024-6-17 20:0:32][2951376] Mon, 17 Jun 2024
[Debug][2024-6-17 20:0:32][2951376] no task, thread-5
_req_line: GET /favicon.ico HTTP/1.1
--> Host: 8.127.19.140:8888
--> Connection: keep-alive
--> User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64;
) Chrome/126.0.0.0 Safart/537.36 Edg/126.0.0.0

```

测试路径path

The image shows a browser window with a cookie notification dialog. The browser's address bar shows the URL '8.137.19.140:8888'. The notification dialog is titled '正在使用 Cookie' and shows a list of cookies. One cookie is selected and its details are shown below:

| | |
|-----|------------------------|
| 名称 | username |
| 内容 | zhangsan |
| 域 | 8.137.19.140 |
| 路径 | /a/b |
| 发送 | 仅限相同站点的连接 |
| 已创建 | 2024年6月17日星期一 20:03:51 |
| 到期 | 浏览会话结束时 |

At the bottom of the dialog, there are three buttons: '阻止', '删除', and '完成'. The '路径' field in the cookie details is highlighted with a red box.

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/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/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

代码块

```
1 class Http
2 {
3 private:
4     std::string GetMonthName(int month)
5     {
6         std::vector<std::string> months = {"Jan", "Feb", "Mar", "Apr", "May",
7                                             "Jun", "Jul", "Aug", "Sep", "Oct",
8                                             "Nov", "Dec"};
9         return months[month];
10    }
11    std::string GetWeekDayName(int day)
12    {
13        std::vector<std::string> weekdays = {"Sun", "Mon", "Tue", "Wed",
14                                              "Thu", "Fri", "Sat"};
15        return weekdays[day];
16    }
17    std::string ExpireTimeUseRfc1123(int t) // 秒级别的未来UTC时间
```

```

17     {
18         time_t timeout = time(nullptr) + t;
19         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
20         char timebuffer[1024];
21         // 时间格式如: expires=Thu, 18 Dec 2024 12:00:00 UTC
22         snprintf(timebuffer, sizeof(timebuffer), "%s, %02d %s %d
23                                     % 02d : % 02d : % 02d UTC
",
24
25         GetWeekDayName(tm->tm_wday)
26
27         .c_str(),
28             tm->tm_mday,
29             GetMonthName(tm->tm_mon).c_str(),
30             tm->tm_year + 1900,
31             tm->tm_hour,
32             tm->tm_min,
33             tm->tm_sec);
34         return timebuffer;
35     }
36
37 public:
38     Http(uint16_t port)
39     {
40         _tsvr = std::make_unique<TcpServer>(port,
41             std::bind(&Http::HandlerHttp,
42                 this, std::placeholders::_1));
43         _tsvr->Init();
44     }
45     std::string ProveCookieWrite() // 证明cookie能被写入浏览器
46     {
47         return "Set-Cookie: username=zhangsan;";
48     }
49     std::string ProveCookieTimeOut()
50     {
51         return "Set-Cookie: username=zhangsan; expires=" +
52             ExpireTimeUseRfc1123(60) + ";"; // 让cookie 1min后过期
53     }
54     std::string ProvePath()
55     {
56         return "Set-Cookie: username=zhangsan; path=/a/b;";
57     }
58     std::string HandlerHttp(std::string request)
59     {
60         HttpRequest req;
61         req.Deserialize(request);

```

```

59     req.DebugHttp();
60     lg.LogMessage(Debug, "%s\n", ExpireTimeUseRfc1123(60).c_str());
61     HttpResponse resp;
62     resp.SetCode(200);
63     resp.SetDesc("OK");
64     resp.AddHeader("Content-Type: text/html");
65     // resp.AddHeader(ProveCookieWrite()); //测试cookie被写入与自动提交
66     // resp.AddHeader(ProveCookieTimeOut()); //测试过期时间的写入
67     resp.AddHeader(ProvePath()); // 测试路径
68     resp.AddContent("<html><h1>helloworld</h1></html>");
69     return resp.Serialize();
70 }
71 void Run()
72 {
73     _tsvr->Start();
74 }
75 ~Http()
76 {
77 }
78
79 private:
80     std::unique_ptr<TcpServer> _tsvr;
81 };

```

- 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

代码块

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

部分核心代码：

Session.hpp

代码块

```
1 #pragma once
2 #include <iostream>
3 #include <string>
```

```

4  #include <memory>
5  #include <ctime>
6  #include <unistd.h>
7  #include <unordered_map>
8  // 用来进行测试说明
9  class Session
10 {
11 public:
12     Session(const std::string &username, const std::string &status)
13         : _username(username), _status(status)
14     {
15         _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
16     }
17     ~Session()
18     {
19     }
20
21 public:
22     std::string _username;
23     std::string _status;
24     uint64_t _create_time;
25     // 当然还可以再加任何其他信息, 看你的需求
26 };
27 using session_ptr = std::shared_ptr<Session>;
28 class SessionManager
29 {
30 public:
31     SessionManager()
32     {
33         srand(time(nullptr) ^ getpid());
34     }
35     std::string AddSession(session_ptr s)
36     {
37         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
38         std::string sessionid = std::to_string(randomid);
39         _sessions.insert(std::make_pair(sessionid, s));
40         return sessionid;
41     }
42     session_ptr GetSession(const std::string sessionid)
43     {
44         if (_sessions.find(sessionid) == _sessions.end())
45             return nullptr;
46         return _sessions[sessionid];
47     }
48     ~SessionManager()

```

```

49     {
50     }
51
52     private:
53         std::unordered_map<std::string, session_ptr> _sessions;
54     };

```

HttpProtocol.hpp

代码块

```

1  #pragma once
2  #include <iostream>
3  #include <string>
4  #include <sstream>
5  #include <vector>
6  #include <memory>
7  #include <ctime>
8  #include <functional>
9  #include "TcpServer.hpp"
10 #include "Session.hpp" // 引入session
11 const std::string HttpSep = "\r\n";
12 // 可以配置的
13 const std::string homepage = "index.html";
14 const std::string wwwroot = "./wwwroot";
15 class HttpRequest
16 {
17 public:
18     HttpRequest() : _req_blank(HttpSep), _path(wwwroot)
19     {
20     }
21     bool GetLine(std::string &str, std::string *line)
22     {
23         auto pos = str.find(HttpSep);
24         if (pos == std::string::npos)
25             return false;
26         *line = str.substr(0, pos); // \r\n
27         str.erase(0, pos + HttpSep.size());
28         return true;
29     }
30     void Parse()
31     {
32         // 解析出来url
33         std::stringstream ss(_req_line);
34         ss >> _method >> _url >> _http_version;
35         // 查找cookie

```

```

36     std::string prefix = "Cookie: ";
37     for (auto &line : _req_header)
38     {
39         std::string cookie;
40         if (strncmp(line.c_str(), prefix.c_str(), prefix.size()) == 0) //
找到了
41             {
42                 cookie = line.substr(prefix.size()); // 截取"Cookie: "之后的就行
了
43                 _cookies.emplace_back(cookie);
44                 break;
45             }
46     }
47     // 查找sessionid
48     prefix = "sessionid=";
49     for (const auto &cookie : _cookies)
50     {
51         if (strncmp(cookie.c_str(), prefix.c_str(), prefix.size()) == 0)
52         {
53             _sessionid = cookie.substr(prefix.size()); // 截取"sessionid="之
后的就行了
54             // std::cout << "_sessionid: " << _sessionid << std::endl;
55         }
56     }
57 }
58 std::string Url()
59 {
60     return _url;
61 }
62 std::string SessionId()
63 {
64     return _sessionid;
65 }
66 bool Deserialize(std::string &request)
67 {
68     std::string line;
69     bool ok = GetLine(request, &line);
70     if (!ok)
71         return false;
72     _req_line = line;
73     while (true)
74     {
75         bool ok = GetLine(request, &line);
76         if (ok && line.empty())
77         {
78             _req_content = request;
79             break;

```

```

80         }
81         else if (ok && !line.empty())
82         {
83             _req_header.push_back(line);
84         }
85         else
86         {
87             break;
88         }
89     }
90     return true;
91 }
92 void DebugHttp()
93 {
94     std::cout << "_req_line: " << _req_line << std::endl;
95     for (auto &line : _req_header)
96     {
97         std::cout << "---> " << line << std::endl;
98     }
99 }
100 ~HttpRequest()
101 {
102 }
103
104 private:
105     // http报文自动
106     std::string _req_line; // method url http_version
107     std::vector<std::string> _req_header;
108     std::string _req_blank;
109     std::string _req_content;
110     // 解析之后的内容
111     std::string _method;
112     std::string _url; // / /dira/dirb/x.html /dira/dirb/XX?
113     username = 100 &&password = 1234 / dira / dirb std::string _http_version;
114     std::string _path; // "./wwwroot"
115     std::string _suffix; // 请求资源的后缀
116     std::vector<std::string> _cookies; // 其实cookie可以有多个, 因为Set-Cookie可
117     以被写多条, 测试, 一条够了。
118     std::string _sessionid; // 请求携带的sessionid, 仅仅用来测试
119 };
120 const std::string BlankSep = " ";
121 const std::string LineSep = "\r\n";
122 class HttpResponse
123 {
124 public:
125     HttpResponse() : _http_version("HTTP/1.0"), _status_code(200),
126                     _status_code_desc("OK"), _resp_blank(LineSep)

```

```

127     {
128     }
129     void SetCode(int code)
130     {
131         _status_code = code;
132     }
133     void SetDesc(const std::string &desc)
134     {
135         _status_code_desc = desc;
136     }
137     void MakeStatusLine()
138     {
139         _status_line = _http_version + BlankSep +
140             std::to_string(_status_code) + BlankSep +
141             _status_code_desc + LineSep;
142     }
143     void AddHeader(const std::string &header)
144     {
145         _resp_header.push_back(header + LineSep);
146     }
147     void AddContent(const std::string &content)
148     {
149         _resp_content = content;
150     }
151     std::string Serialize()
152     {
153         MakeStatusLine();
154         std::string response_str = _status_line;
155         for (auto &header : _resp_header)
156         {
157             response_str += header;
158         }
159         response_str += _resp_blank;
160         response_str += _resp_content;
161         return response_str;
162     }
163     ~HttpResponse() {}
164 private:
165     std::string _status_line;
166     std::vector<std::string> _resp_header;
167     std::string _resp_blank;
168     std::string _resp_content; // body
169     // httpversion StatusCode StatusCodeDesc
170     std::string _http_version;
171     int _status_code;
172     std::string _status_code_desc;

```

```

173 };
174 class Http
175 {
176 private:
177     std::string GetMonthName(int month)
178     {
179         std::vector<std::string> months = {"Jan", "Feb", "Mar", "Apr", "May",
180                                           "Jun", "Jul", "Aug", "Sep", "Oct",
181                                           "Nov", "Dec"};
182         return months[month];
183     }
184     std::string GetWeekDayName(int day)
185     {
186         std::vector<std::string> weekdays = {"Sun", "Mon", "Tue", "Wed",
187                                              "Thu", "Fri", "Sat"};
188         return weekdays[day];
189     }
190     std::string ExpireTimeUseRfc1123(int t) // 秒级别的未来UTC时间
191     {
192         time_t timeout = time(nullptr) + t;
193         struct tm *tm = gmtime(&timeout); // 这里不能用localtime, 因为localtime是
194         // 默认带了时区的。gmtime获取的就是UTC统一时间
195         char timebuffer[1024];
196         // 时间格式如: expires=Thu, 18 Dec 2024 12:00:00 UTC
197         // %s, %02d %s %d
198         // % 02d : % 02d : % 02d UTC
199         sprintf(timebuffer, "%s, %02d %s %d",
200               tm->tm_wday, tm->tm_mday,
201               GetMonthName(tm->tm_mon).c_str(),
202               tm->tm_year + 1900,
203               tm->tm_hour,
204               tm->tm_min,
205               tm->tm_sec);
206         return timebuffer;
207     }
208 public:
209     Http(uint16_t port)
210     {
211         _tsvr = std::make_unique<TcpServer>(port,
212                                             std::bind(&Http::HandlerHttp,
213                                             this, std::placeholders::_1));
214         _tsvr->Init();

```

```

214     _session_manager = std::make_unique<SessionManager>();
215 }
216 std::string ProveCookieWrite() // 证明cookie能被写入浏览器
217 {
218     return "Set-Cookie: username=zhangsan;";
219 }
220 std::string ProveCookieTimeOut()
221 {
222     return "Set-Cookie: username=zhangsan; expires=" +
223           ExpireTimeUseRfc1123(60) + ";"; // 让cookie 1min后过期
224 }
225 std::string ProvePath()
226 {
227     return "Set-Cookie: username=zhangsan; path=/a/b;";
228 }
229 std::string ProveSession(const std::string &session_id)
230 {
231     return "Set-Cookie: sessionid=" + session_id + ";";
232 }
233 std::string HandlerHttp(std::string request)
234 {
235     HttpRequest req;
236     HttpResponse resp;
237     req.Deserialize(request);
238     req.Parse();
239     // req.DebugHttp();
240     // std::cout << req.Url() << std::endl;
241     // 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
242     static int number = 0;
243     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
244     {
245         std::string sessionid = req.SessionId();
246         if (sessionid.empty()) // 说明历史没有登陆过
247         {
248             std::string user = "user-" + std::to_string(number++);
249             session_ptr s = std::make_shared<Session>(user, "logged");
250             std::string sessionid = _session_manager->AddSession(s);
251             lg.LogMessage(Debug, "%s 被添加, sessionid是: %s\n",
252                           user.c_str(), sessionid.c_str());
253             resp.AddHeader(ProveSession(sessionid));
254         }
255     }
256     else
257     {
258         // 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。

```

```

259         std::string sessionid = req.SessionId();
260         if (!sessionid.empty())
261         {
262             session_ptr s = _session_manager->GetSession(sessionid);
263             // 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
264             // 浏览器还有历史sessionid，但是服务器重启之后，session对象没有了。
265             if (s != nullptr)
266                 lg.LogMessage(Debug, "%s 正在活跃.\n", s ->
_username.c_str());
267             else
268                 lg.LogMessage(Debug, "cookie : %s 已经过期，需要清理\n",
sessionid.c_str());
270         }
271     }
272     resp.SetCode(200);
273     resp.SetDesc("OK");
274     resp.AddHeader("Content-Type: text/html");
275     // resp.AddHeader(ProveCookieWrite()); //测试cookie被写入与自动提交
276     // resp.AddHeader(ProveCookieTimeOut()); //测试过期时间的写入
277     // resp.AddHeader(ProvePath()); // 测试路径
278     resp.AddContent("<html><h1>helloworld</h1></html>");
279     return resp.Serialize();
280 }
281 void Run()
282 {
283     _tsvr->Start();
284 }
285 ~Http()
286 {
287 }
288
289 private:
290     std::unique_ptr<TcpServer> _tsvr;
291     std::unique_ptr<SessionManager> _session_manager;
292 };

```

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。

hel

正在使用 Cookie

已允许 已阻止

查看此页面时设置了以下 Cookie

▲ 8.137.19.140

▲ 📁 Cookie

🍪 sessionid

| | |
|-----|------------------------|
| 名称 | sessionid |
| 内容 | 2550656811 |
| 域 | 8.137.19.140 |
| 路径 | / |
| 发送 | 仅限相同站点的连接 |
| 已创建 | 2024年6月18日星期二 15:21:15 |
| 到期 | 浏览会话结束时 |

阻止

删除

完成



2. 访问/login, 模拟登录

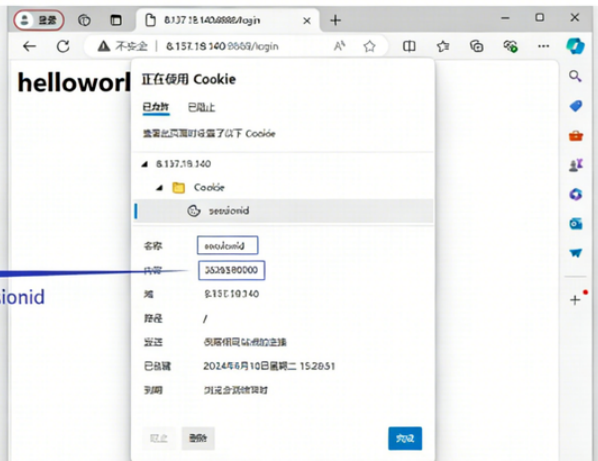
Microsoft Edge

Microsoft Edge

```
[Debug][2024-6-18 15:28:52][2955218] user-0 被添加, sessionId是: 3628580000
```

```
[Debug][2024-6-18 15:28:52][2955218] user-0 正在活跃
```

服务端和浏览器端一样的sessionId



Google Chrome

```
[Debug][2024-6-18 15:30:46][2955218] user-1 被添加, sessionId是: 3260880733
```

```
[Debug][2024-6-18 15:30:46][2955218] user-1 正在活跃
```



3. 两个浏览器访问任意的站点资源

