前后端交互

1、身份验证机制

1、什么身份认证：

确认用户身份、身份验证、身份鉴权

2、为什么要进行身份认证

确认用户是其声称的身份。需要提供信息才能进行身份认证。

3、不同模式下的身份认证

服务器端渲染：使用 ==Session认证机制==

前后端分类：使用 ==JWT认证机制==

1.1 Session认证机制

1.1.1、HTTP协议的无状态性

客户端的每次HTTP请求都是独立的，多个HTTP请求之间无关联。

1.1.2、如何使HTTP请求之间建立关联关系?

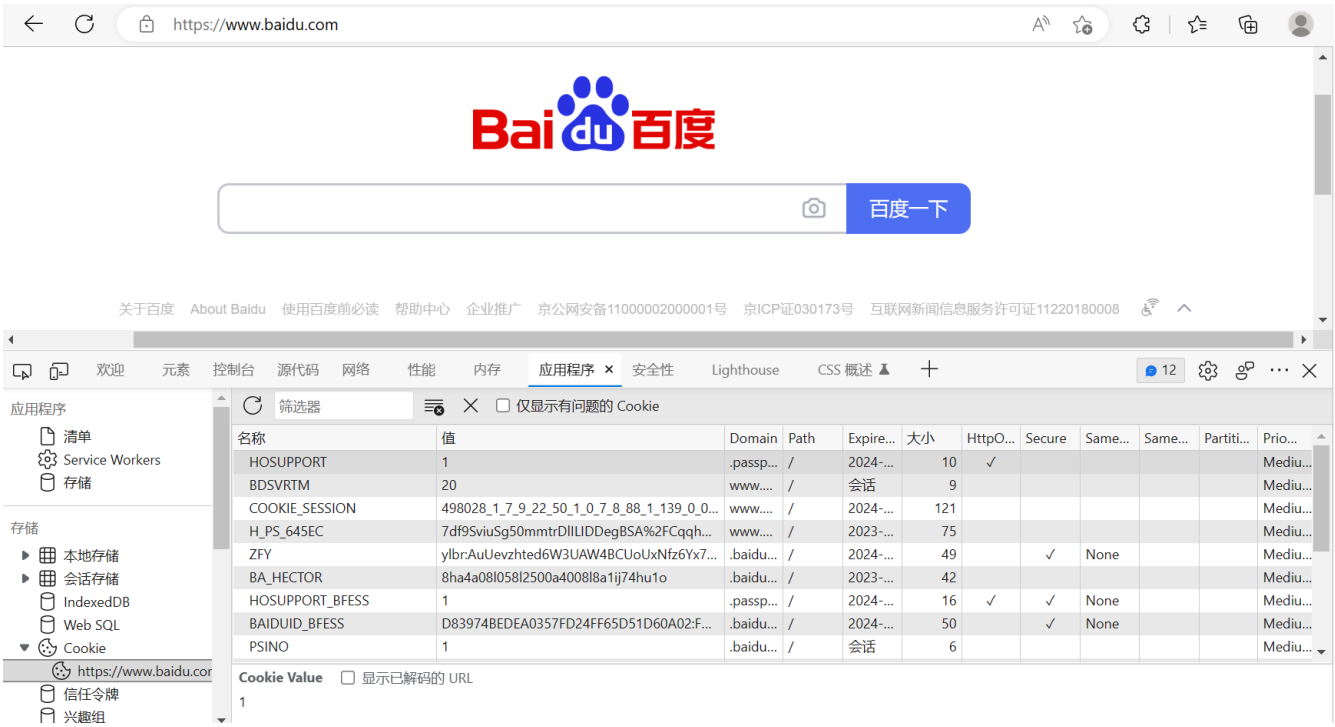
用户登录时给服务器提供身份信息，服务器保存用户的身份信息。身份信息就是==Cookie==--Web中的身份认证方式。

1.1.3、Cookie（不支持跨域认证）

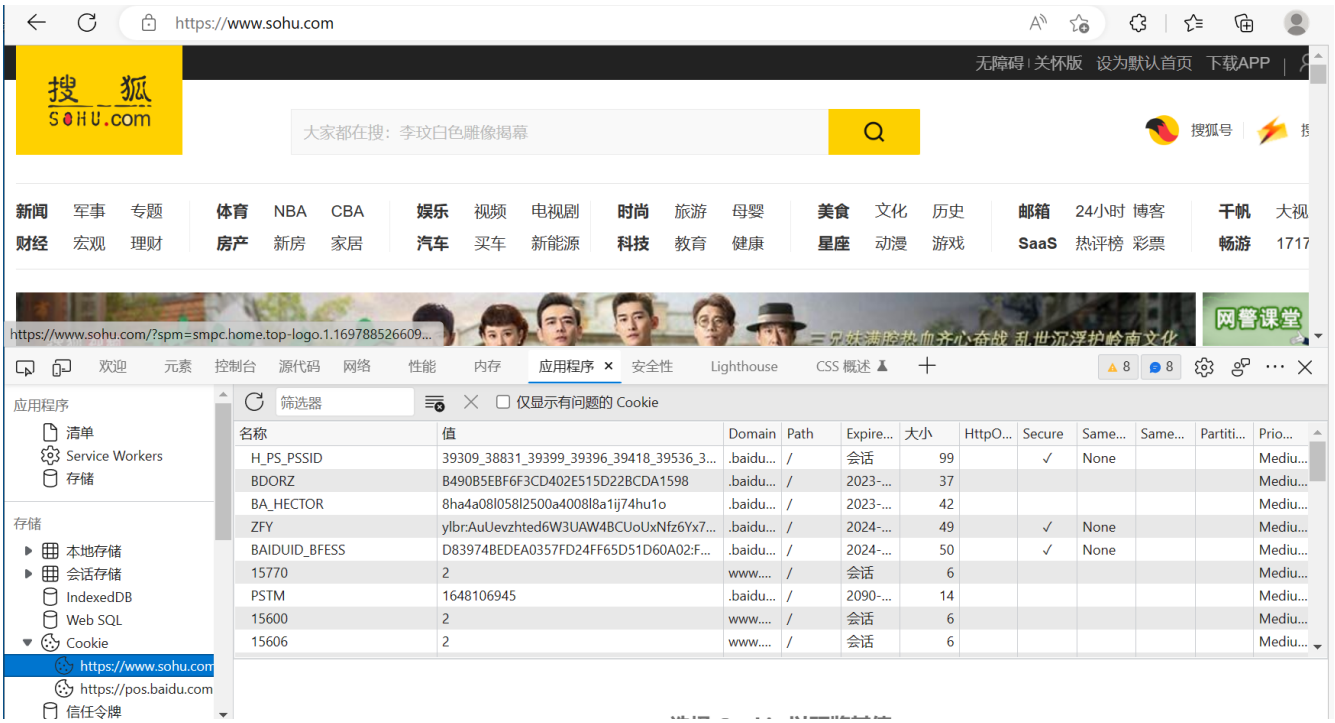
Cookie是存储在浏览器中的不超过4KB的==字符串==。由一个名称(Name)、一个值(Value)和几个可选属性组成（控制Cookie的有效期、安全性、使用范围）。

1）cookie的格式

在浏览器中浏览www.baidu.com，然后打开浏览器的调试面板，点到应用程序(application)。在该选项下，左侧的存储部分有Cookie，点Cookie左侧的按钮，就列出存储在浏览器中的不同网站的Cookie。选中baidu网站的Cookie，在右侧会列出和baidu网站交互过程中保存的所有Cookie。



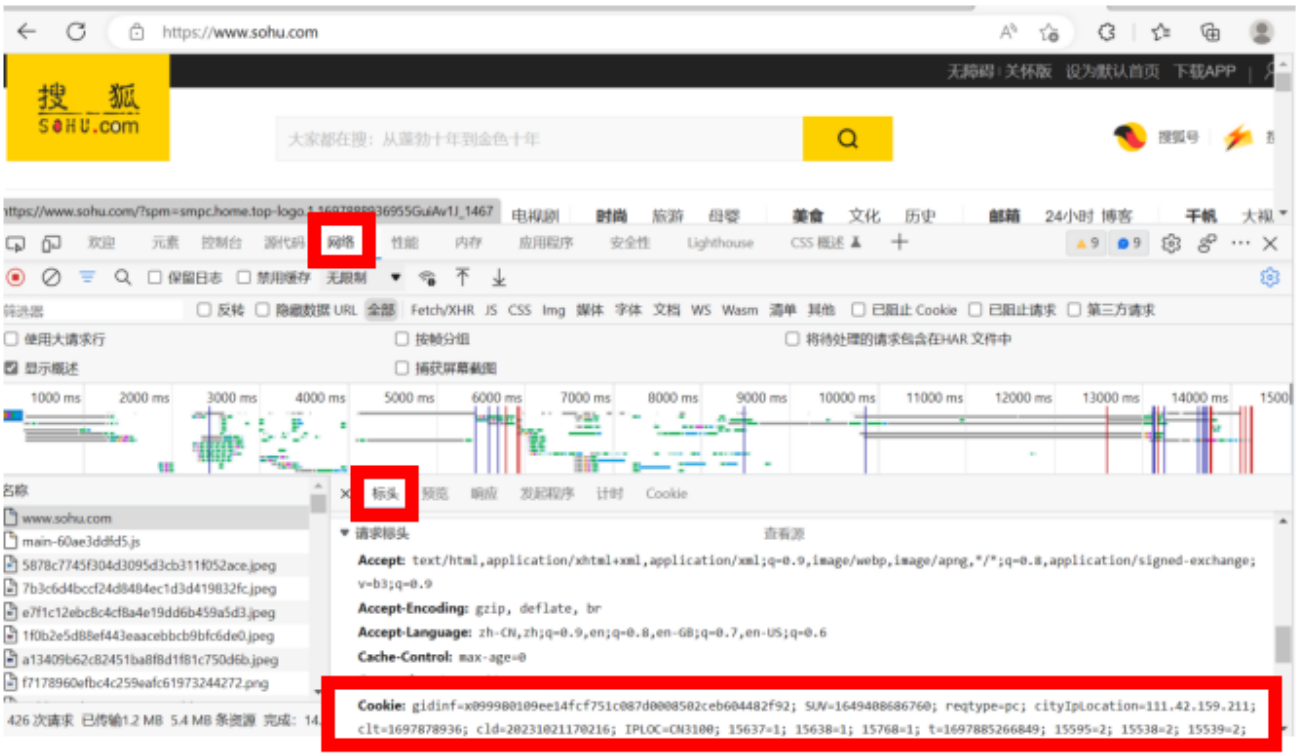
再访问sohu网站，查看保存的Cookie，会增加sohu网站保存的Cookie。



各网站保存各自的Cookie。每次通过浏览器访问某网站时，会自动把当前域名下的所有Cookie一同发送到该网站。

2）Cookie特性：自动发送、域名独立、过期时限、4KB长度

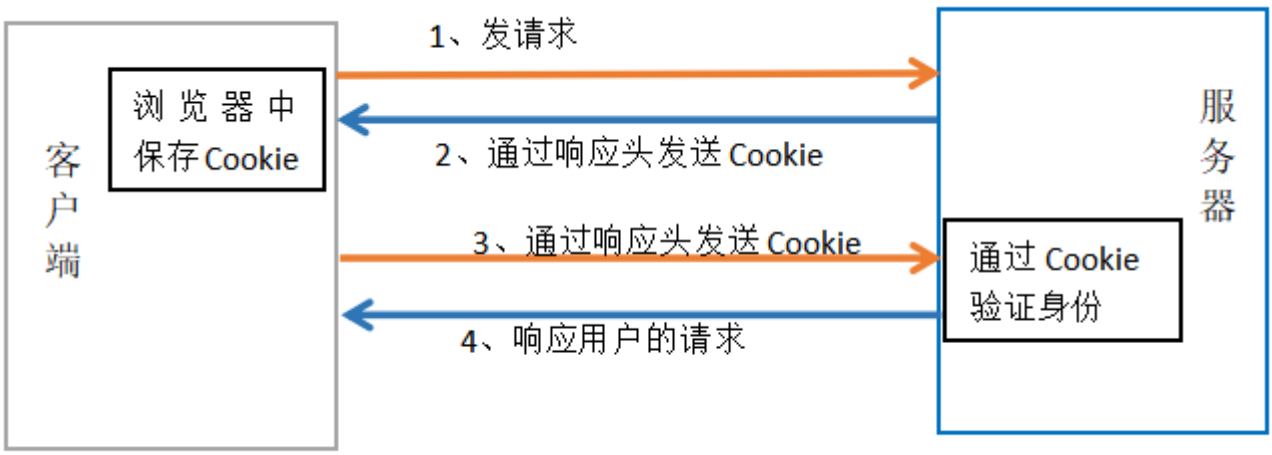
3) 在调试面板的network选项下，查看浏览器发送的Cookie(找到网站www.sohu.com，右侧选择标头，查看下面的请求头，请求头里面包含Cookie)



1.1.4、Cookie在身份认证中的作用

用户通过浏览器第一次请求某网站时，网站通过响应头的形式，向客户端发送一个身份认证的Cookie，客户端会自动将Cookie保存在浏览器中。

后面再访问该服务器时，浏览器会自动将身份认证相关Cookie通过请求头的形式发送给服务器，服务器就能通过Cookie验证用户的身份。



1.1.5、Cookie不安全

Cookie存储在浏览器中，很容易被伪造。不要把密码等重要信息存储在Cookie中。

1.1.6 提高身份认证的安全性

服务器会给客户端发送Cookie，用户通过向服务器验证Cookie来验证身份。

1.1.7 Session的工作原理

浏览器端将用户名和密码发送到服务器端，服务器进行验证：验证通过后，把用户的信息保存在服务器的内存中，生成对应的Cookie字符串。服务器把生成的Cookie返回给客户端，客户端保存Cookie，再次向该服务器发请求时，通过请求头把该域名下的可用Cookie发送给服务器。服务器通过内存中的Cookie进行验证，验证通过后响应浏览器的请求。

1.2 在Express中使用Session认证机制

1、在命令行下安装express-session模块

```
npm install express-session
```

2、在程序中配置express-session模块

```
//导入session中间件
const session = require('express-session')
//配置session中间件
app.use(session({
    secret: 'session',   //加密的秘钥，秘钥自己设置
    resave: false,  //固定写法
    saveUninitialized: true  //固定写法
}))
```

3、初始化服务器程序，代码如下

```
// 导入 express 模块
const express = require('express')
// 创建 express 的服务器实例
const app = express()

// TODO_01：请配置 Session 中间件
const session = require('express-session')
app.use(
  session({
    secret: '%￥#@! ',
    resave: false,
    saveUninitialized: true,
  })
)

// 托管静态页面
app.use(express.static('./pages'))
// 解析 POST 提交过来的表单数据，下面这行代码不写不能解析客户端发送的请求体
app.use(express.urlencoded({ extended: false }))

// 调用 app.listen 方法，指定端口号并启动web服务器
app.listen(80, function () {
  console.log('Express server running at http://127.0.0.1:80')
})
```

4、向session中保存浏览器发送的信息

express-session这个中间件配置成功后，可通过req.session访问和使用session对象，保存用户的相关信息。

```
// 登录的 API 接口,在登录请求中保存用户登录信息
app.post('/api/login', (req, res) => {
  // 判断用户提交的登录信息是否正确
    //用户名或密码不正确，返回登录失败信息，return结束
  if (req.body.username !== 'admin' || req.body.password !== '000000') {
    return res.send({ status: 1, msg: '登录失败' })
  }

  // TODO_02：请将登录成功后的用户信息，保存到 Session 中
  // 注意：只有成功配置了 express-session 这个中间件之后，才能够通过 req 点出来 session 这个属性
  // 用户的信息，在req.session里添加user保存浏览器发送的用户信息
  req.session.user = req.body
  // 用户的登录状态，在req.session里添加islogin保存是否登录状态
  req.session.islogin = true

  res.send({ status: 0, msg: '登录成功' })
})
```

5、从session中取数据

```
// 获取用户姓名的接口
app.get('/api/username', (req, res) => {
  // TODO_03：请从 Session 中获取用户的名称，响应给客户端
  if (!req.session.islogin) {
    return res.send({ status: 1, msg: 'fail' })
  }
  res.send({
    status: 0,
    msg: 'success',
    username: req.session.user.username,
  })
})
```

6、清空session

调用req.session.destroy()函数，即可清空服务器保存的session信息。

```
// 退出登录的接口
app.post('/api/logout', (req, res) => {
  // TODO_04：清空 Session 信息
  req.session.destroy()
  res.send({
    status: 0,
    msg: '退出登录成功',
  })
})
```

7、浏览器、服务器通过session完成会话过程

浏览器端：

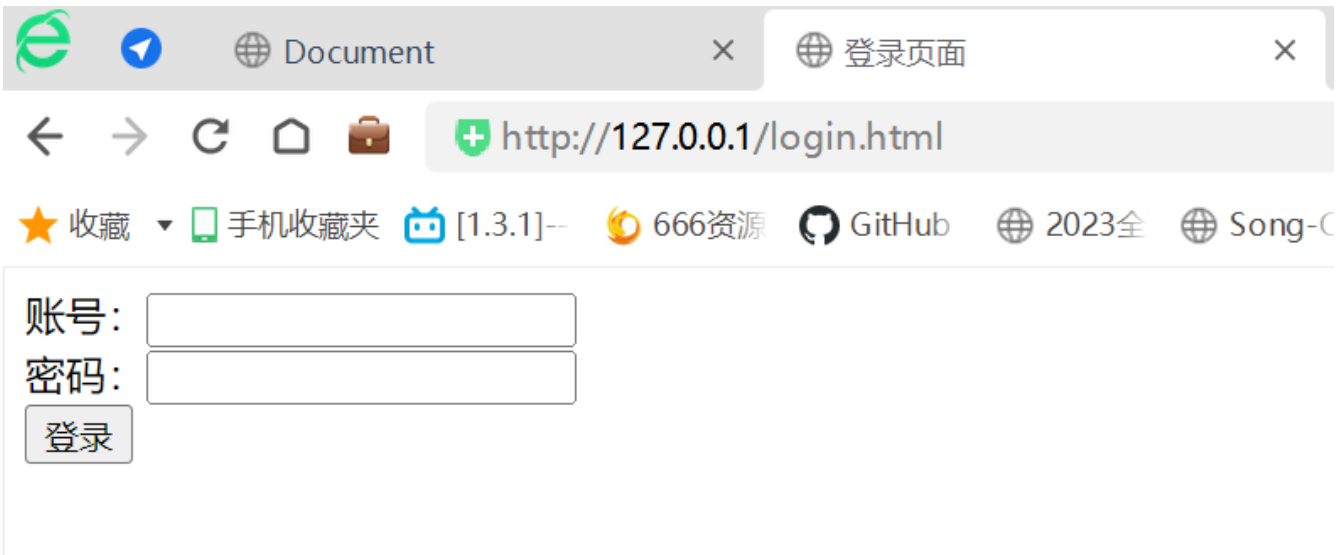
浏览器端登录程序在pages目录中。

index.html是主页面，进入index.html后自动向服务器的/api/username地址发送请求，判断 是否已登录。如果未登录，则进入login.html页面进行登录。点退出按钮则退出登录，服务器端情况session信息。

在login页面中进行登录请求，向/api/login地址发送登录请求。登录成功则进入index.html页面。登录失败，会弹出alert窗口告知登录失败。

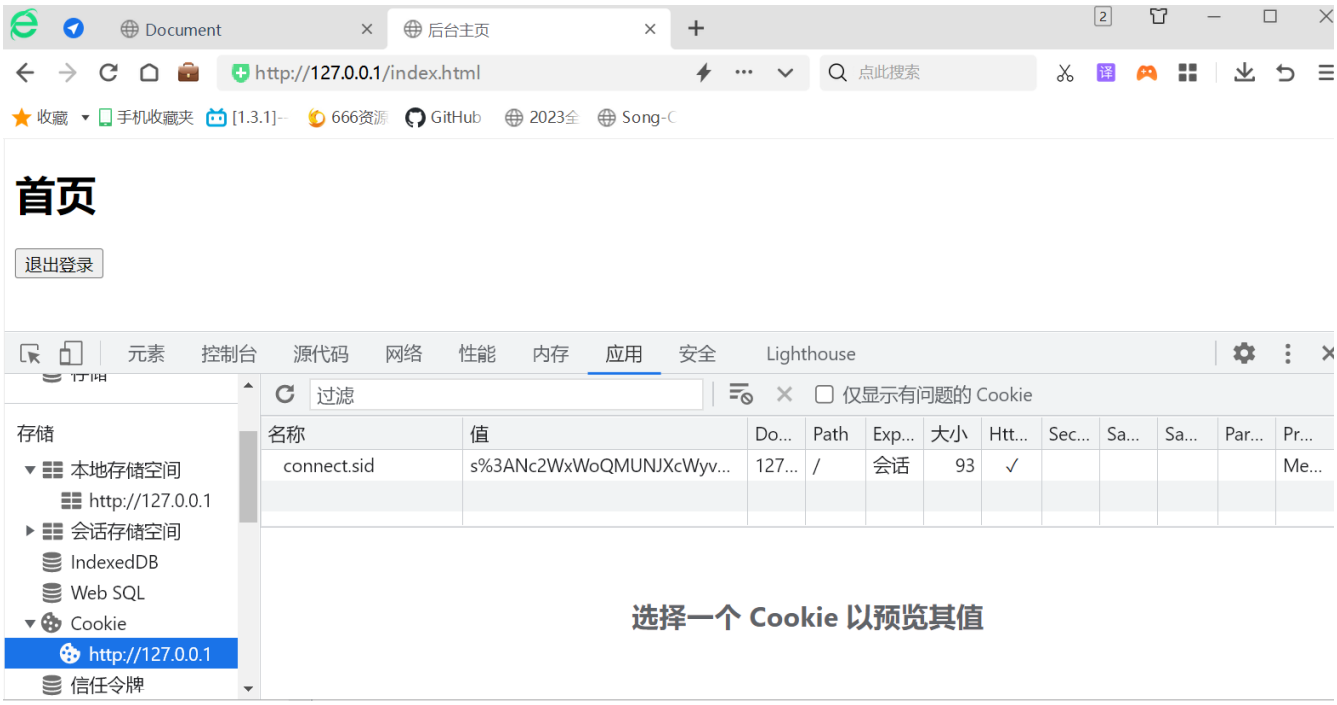
服务器端启动程序app.js后，通过浏览器访问127.0.0.1/，获得下面页面：

点击确定按钮进入登录页面：



输入用户名admin，输入密码000000则登录成功。（==可以在服务器端修改代码，验证是否和数据库中保存的用户名密码一致，一致则允许登录，不一致则不允许登录。==）

点击确定后进入index.html页面，为已登录状态。打开浏览器调试面板，查看cookie信息：



点击退出登录后，服务器端会删除会话信息。再访问127.0.0.1/需要重新登录。