网络编程 http https http2 websocket

复习

- node.js核心API：fs, buffer, global, process, path, event, http
- 仿写简版express，实现简单路由功能

课堂目标

- 掌握HTTP协议
- 掌握http服务使用
- 掌握前后端通信技术ajax、websocket等
- 能解决常见web问题：跨域、session
- 实战一个爬虫程序
- 利用多种方式实现实时聊天程序
- 了解https
- 了解http2

HTTP协议

- http协议详解
- 创建接口，http-server.js

```
const http = require("http");
const fs = require("fs");

http
  .createServer((req, res) => {
    const { method, url } = req;
    if (method == "GET" && url == "/") {
      fs.readFile("./index.html", (err, data) => {
        res.setHeader("Content-Type", "text/html");
        res.end(data);
      });
```

```
    } else if (method == "GET" && url == "/users") {
      res.setHeader("Content-Type", "application/json");
      res.end(JSON.stringify([{ name: "tom", age: 20 }]));
    }
  })
  .listen(3000);
```

- 请求接口，index.html

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    axios
        .get("/users")
        .then(res => res.data)
        .then(users => console.log(users));
</script>
```

- 跨域：浏览器同源策略引起的接口调用问题

```
// 1.创建http-server-2.js，使用端口3001
server.listen(3001);
// 2.index.html中请求位于3000服务器的接口
axios.get("http://localhost:3000/users")
```

  - 浏览器抛出跨域错误

```
Access to XMLHttpRequest at 'http://localhos (index):1
t:3000/users' from origin 'http://localhost:3001' has
been blocked by CORS policy: No 'Access-Control-Allow-
Origin' header is present on the requested resource.
▶Uncaught (in promise) Error: Network     spread.js:25
Error
    at e.exports (spread.js:25)
    at XMLHttpRequest.l.onerror (spread.js:25)
```

  - 常用解决方案：

    1. JSONP(JSON with Padding)，前端+后端方案，绕过跨域

    > 前端构造script标签请求指定URL（由script标签发出的GET请求不受同源策略限制），服务器返回一个函数执行语句，该函数名称通常由查询参callback的值决定，函数的参数为服务器返回的json数据。该函数在前端执行后即可获取数据。

    2. 代理服务器

    > 请求同源服务器，通过该服务器转发请求至目标服务器，得到结果再转发给前端。
    >
    > 前端开发中测试服务器的代理功能就是采用的该解决方案，但是最终发布上线时如果web应用和接口服务器不在一起仍会跨域。

    3. CORS(Cross Origin Resource Share) - 跨域资源共享，后端方案，解决跨域

原理：cors是w3c规范，真正意义上解决跨域问题。它需要服务器对请求进行检查并对响应头做相应处理，从而允许跨域请求。

具体实现：

- 响应简单请求: 动词为get/post/head，没有自定义请求头，Content-Type是application/x-www-form-urlencoded，multipart/form-data或text/plain之一，通过添加以下响应头解决：

```
res.setHeader('Access-Control-Allow-Origin', 'http://localhost:3001')
```

- 响应preflight请求，需要响应浏览器发出的options请求（预检请求），并根据情况设置响应头：

```
else if (method == "OPTIONS" && url == "/users") {
  res.writeHead(200, {
    "Access-Control-Allow-Origin": "http://localhost:3001",
    "Access-Control-Allow-Headers": "X-Token,Content-Type",
    "Access-Control-Allow-Methods": "PUT"
  });
  res.end();
}
```

> 该案例中可以通过添加自定义的x-token请求头使请求变为preflight请求
>
> ```
> // index.html
> axios.get("http://localhost:3000/users", {headers:{'X-Token':'jilei'}})
> ```
>
> 则服务器需要允许x-token，若请求为post，还传递了参数：
>
> ```
> // index.html
> axios.post("http://localhost:3000/users", {foo:'bar'}, {headers:{'X-Token':'jilei'}})
> // http-server.js
> else if ((method == "GET" || method == "POST") && url == "/users") {}
> ```
>
> 则服务器还需要允许content-type请求头

- 如果要携带cookie信息，则请求变为credential请求：

```
// 预检options中和/users接口中均需添加
res.setHeader('Access-Control-Allow-Credentials', 'true');
```

实战一个爬虫

原理：服务端模拟客户端发送请求到目标服务器获取页面内容并解析，获取其中关注部分的数据。

```
const originRequest = require("request");
const cheerio = require("cheerio");
const iconv = require("iconv-lite");

function request(url, callback) {
  const options = {
    url: url,
    encoding: null
  };
  originRequest(url, callback);
}

for (let i = 100553; i < 100563; i++) {
  const url = `https://www.dy2018.com/i/${i}.html`;
  request(url, function(err, res, body) {
    const html = iconv.decode(body, "gb2312");
    const $ = cheerio.load(html);
    console.log($(".title_all h1").text());
  });
}
```

实现一个实时聊天程序

- Socket实现

原理：Net模块提供一个异步API能够创建基于流的TCP服务器，客户端与服务器建立连接后，服务器可以获得一个全双工Socket对象，服务器可以保存Socket对象列表，在接收某客户端消息时，推送给其他客户端。

```
const net = require('net')
const chatServer = net.createServer(), clientList = []
chatServer.on('connection', function (client) {
    client.write('Hi!\n');
    clientList.push(client)
    client.on('data', function (data) {
        clientList.forEach(v => {
            v.write(data)
        })
    })
})
chatServer.listen(9000)
```

通过Telnet连接服务器

```
telnet localhost 9000
```

- Http实现

原理：客户端通过ajax方式发送数据给http服务器，服务器缓存消息，其他客户端通过轮询方式查询最新数据并更新列表。

```
<html>
<head>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
</head>

<body>
    <div id="app">
        <input v-model="message">
        <button v-on:click="send">发送</button>
        <button v-on:click="clear">清空</button>
        <div v-for="item in list">{{item}}</div>
    </div>

    <script>
        const host = 'http://localhost:3000'
        var app = new Vue({
            el: '#app',
            data: {
                list: [],
                message: 'Hello Vue!'
            },
            methods: {
                send: async function () {
                    let res = await axios.post(host + '/send', {
                        message: this.message
                    })
                    this.list = res.data
                },
                clear: async function () {
                    let res = await axios.post(host + '/clear')
                    this.list = res.data
                }
            },
            mounted: function () {
                setInterval(async () => {
                    const res = await axios.get(host + '/list')
                    this.list = res.data
                }, 1000);
            }
        });
    </script>
</body>
</html>
```

```
const express = require('express')
const app = express()
const bodyParser = require('body-parser');
const path = require('path')
```

```
app.use(bodyParser.json());

const list = ['ccc', 'ddd']

app.get('/chat', (req,res)=>{
    res.sendFile(path.resolve('./chat.html'))
})

app.get('/list', (req, res) => {
    res.end(JSON.stringify(list))
})

app.post('/send', (req, res) => {
    list.push(req.body.message)
    res.end(JSON.stringify(list))
})

app.post('/clear', (req, res) => {
    list.length = 0
    res.end(JSON.stringify(list))
})

app.listen(3000);
```

- Socket.IO实现

```
// src/im/index.js
const io = require('socket.io')(server)
io.on('connection', (socket) => {
    console.log('io connection ..')
    socket.on('chat', (msg) => {
    });
});

 app.post('/send', (req, res) => {
    list.push(req.body.message)

    // // SocketIO 增加
    io.emit('chat', list)

    res.end(JSON.stringify(list))
})
```

```
// src/im/index.html
mounted: function () {
    // http轮训
    // setInterval(async () => {
    //     const res = await axios.get(host + '/list')
    //     this.list = res.data
```

```
    // }, 1000);

    // websocket方式
    const socket = io(host)
    socket.on('chat', list => {
        this.list = list
    });
}
```

> Socket.IO库特点：
>
> - 源于HTML5标准
> - 支持优雅降级
>   - WebSocket
>   - WebSocket over FLash
>   - XHR Polling
>   - XHR Multipart Streaming
>   - Forever Iframe
>   - JSONP Polling

Https（安全课再讲）

- 创建证书

```
# 创建私钥
openssl genrsa -out privatekey.pem 1024
# 创建证书签名请求
openssl req -new -key privatekey.pem -out certrequest.csr
# 获取证书，线上证书需要经过证书授证中心签名的文件；下面只创建一个学习使用证书
openssl x509 -req -in certrequest.csr -signkey privatekey.pem -out certificate.pem
# 创建pfx文件
openssl pkcs12 -export -in certificate.pem -inkey privatekey.pem -out
certificate.pfx
```

Http2（优化的时候讲）

- 多路复用 - 雪碧图、多域名CDN、接口合并
  - 官方演示 - https://http2.akamai.com/demo
  - 多路复用允许同时通过单一的 HTTP/2 连接发起多重的请求-响应消息；而HTTP/1.1协议中，浏览器客户端在同一时间，针对同一域名下的请求有一定数量限制。超过限制数目的请求会被阻塞**
- 首部压缩
  - http/1.x 的 header 由于 cookie 和 user agent很容易膨胀，而且每次都要重复发送。http/2使用 encoder 来减少需要传输的 header 大小，通讯双方各自 cache一份 header fields 表，既避免了重复 header 的传输，又减小了需要传输的大小。高效的压缩算法可以很大的压缩 header，减少发送包的数量从而降低延迟
- 服务端推送

- 在 HTTP/2 中，服务器可以对客户端的一个请求发送多个响应。举个例子，如果一个请求请求的是 index.html，服务器很可能会同时响应index.html、logo.jpg 以及 css 和 js 文件，因为它知道客户端会用到这些东西。这相当于在一个 HTML 文档内集合了所有的资源