浏览器的通信能力

用户代理

浏览器可以代替用户完成http请求，代替用户解析响应结果，所以我们称之为：

用户代理 user agent

在网络层面，对于前端开发者，必须要知道浏览器拥有的两大核心能力：

- 自动发出请求的能力
- 自动解析响应的能力

自动发出请求的能力

当一些事情发生的时候，浏览器会代替用户自动发出http请求，常见的包括：

1. **用户在地址栏输入了一个url地址，并按下了回车**

   浏览器会自动解析URL，并发出一个 `GET` 请求，同时抛弃当前页面。

2. **当用户点击了页面中的a元素**

   浏览器会拿到a元素的href地址，并发出一个 `GET` 请求，同时抛弃当前页面。

3. **当用户点击了提交按钮** `<button type="submit">...</button>`

   浏览器会获取按钮所在的 `<form>` 元素，拿到它的 `action` 属性地址，同时拿到它 `method` 属性值，然后把表单中的数据组织到请求体中，发出 **`指定方法`** 的请求，同时抛弃当前页面。

   > 这种方式的提交现在越来越少见了

4. **当解析HTML时遇到了** `<link> <img> <script> <video> <audio>` **等元素**

浏览器会拿到对应的地址，发出 `GET` 请求

5. **当用户点击了刷新**

   浏览器会拿到当前页面的地址，以及当前页面的请求方法，重新发一次请求，同时抛弃当前页面。

> 浏览器在发出请求时，会自动附带一些请求头

重点来了

从古至今，浏览器都有一个约定：

当发送GET请求时，浏览器不会附带请求体

这个约定深刻的影响着后续的前后端各种应用，现在，几乎所有人都在潜意识中认同了这一点，无论是前端开发人员还是后端开发人员。

由于前后端程序的默认行为，逐步造成了GET和POST的各种差异：

1. 浏览器在发送 GET 请求时，不会附带请求体
2. GET 请求的传递信息量有限，适合传递少量数据；POST 请求的传递信息量是没有限制的，适合传输大量数据。
3. GET 请求只能传递 ASCII 数据，遇到非 ASCII 数据需要进行编码；POST 请求没有限制
4. 大部分 GET 请求传递的数据都附带在 path 参数中，能够通过分享地址完整的重现页面，但同时也暴露了数据，若有敏感数据传递，不应该使用 GET 请求，至少不应该放到 path 中
5. POST 不会被保存到浏览器的历史记录中
6. 刷新页面时，若当前的页面是通过 POST 请求得到的，则浏览器会提示用户是否重新提交。若是 GET 请求得到的页面则没有提示。

自动解析响应的能力

浏览器不仅能发送请求，还能够针对服务器的各种响应结果做出不同的自动处理

常见的处理有：

1. **识别响应码**

浏览器能够自动识别响应码，当出现一些特殊的响应码时浏览器会自动完成处理，比如 `301、302`

2. **根据响应结果做不同的处理**

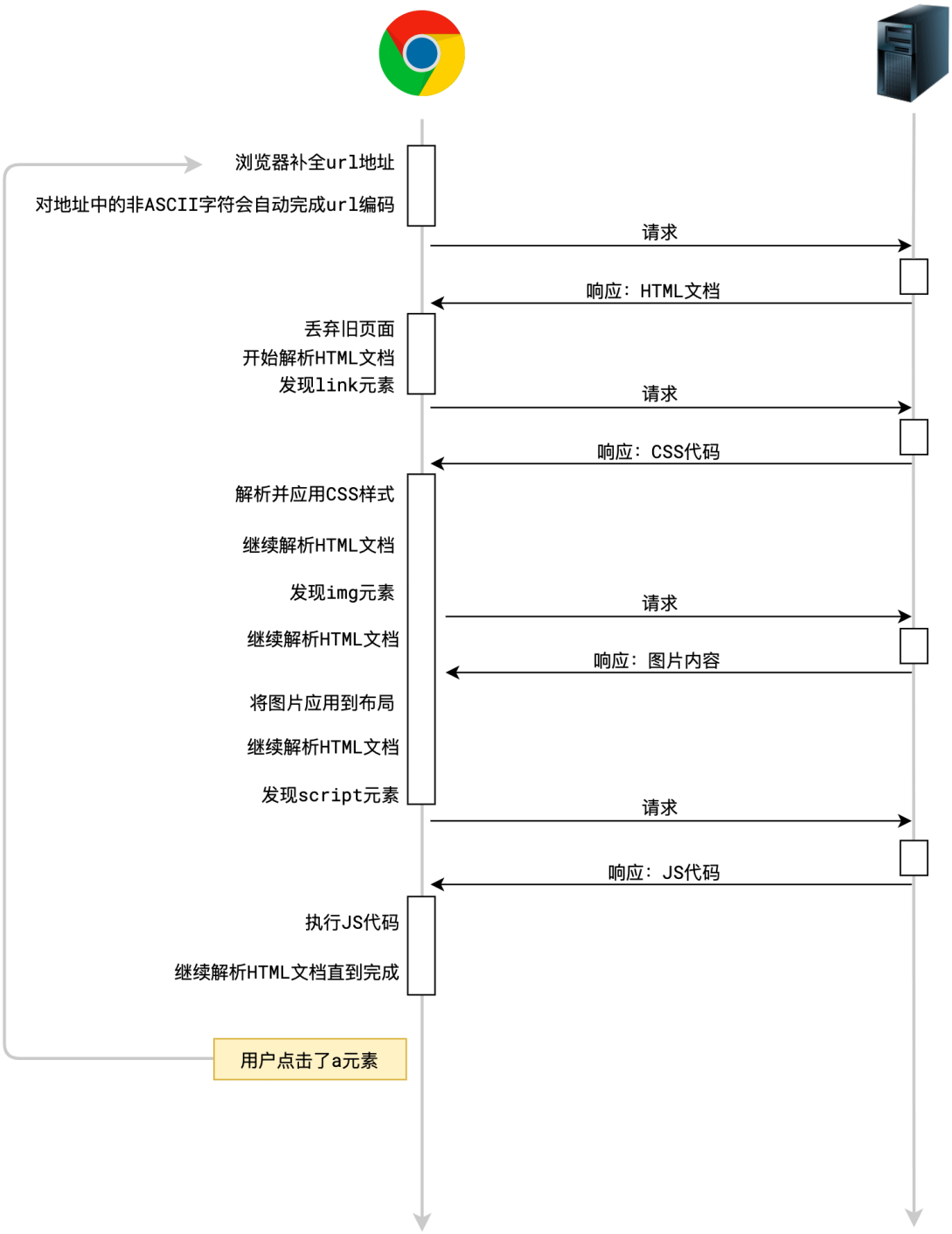
   浏览器能够自动分析响应头中的 `Content-Type` ，根据不同的值进行不同处理，比如：

   - `text/plain` ： 普通的纯文本，浏览器通常会将响应体原封不动的显示到页面上
   - `text/html` ： html文档，浏览器通常会将响应体作为页面进行渲染
   - `text/javascript` 或 `application/javascript` ： js代码，浏览器通常会使用JS执行引擎将它解析执行
   - `text/css` ： css代码，浏览器会将它视为样式
   - `image/jpeg` ： 浏览器会将它视为jpg图片
   - `application/octet-stream` ： 二进制数据，会触发浏览器下载功能
   - `attachment` ： 附件，会触发下载功能

     该值和其他值不同，应放到 `Content-Disposition` 头中。

基本流程

> 访问： https://oss.duyiedu.com/test/index.html



开发接口

浏览器本身就具备网络通信的能力，但在早期，浏览器并没有把这个能力开放给JS。

最早是微软在IE浏览器中把这一能力向JS开放，让JS可以在代码中实现发送请求，这项技术在2005年被正式命名为AJAX（**A**synchronous **J**avascript **A**nd **X**ML）

IE使用了一套API来完成请求的发送，这套API主要依靠一个构造函数完成。该构造函数的名称为 `XMLHttpRequest`，简称为 `XHR`，所以这套API又称之为 `XHR API`

由于 `XHR API` 有着诸多缺陷，在HTML5和ES6发布之后，产生了一套更完善的API来发送请求。这套API主要使用的是一个叫做 `fetch` 的函数，因此这套API又称之为 `Fetch API`

无论是 `XHR` 还是 `Fetch`，它们都是实现ajax的技术手段，只是API不同。

XHR API

```
var xhr = new XMLHttpRequest(); //创建发送请求的对象
xhr.onreadystatechange = function(){ //当请求状态发生改变时运行的函数
    // xhr.readyState:  一个数字，用于判断请求到了哪个阶段
    // 0: 刚刚创建好了请求对象，但还未配置请求（未调用open方法）
    // 1: open方法已被调用
    // 2: send方法已被调用
    // 3: 正在接收服务器的响应消息体
    // 4: 服务器响应的所有内容均已接收完毕

    // xhr.responseText:  获取服务器响应的消息体文本

    // xhr.getResponseHeader("Content-Type") 获取响应头Content-Type
}
xhr.open("请求方法", "url地址"); //配置请求
```

```
xhr.setRequestHeader("Content-Type", "application/json"); //设置请求头
xhr.send("请求体内容"); //构建请求体，发送到服务器，如果没有请求体，传递null
```

Fetch API

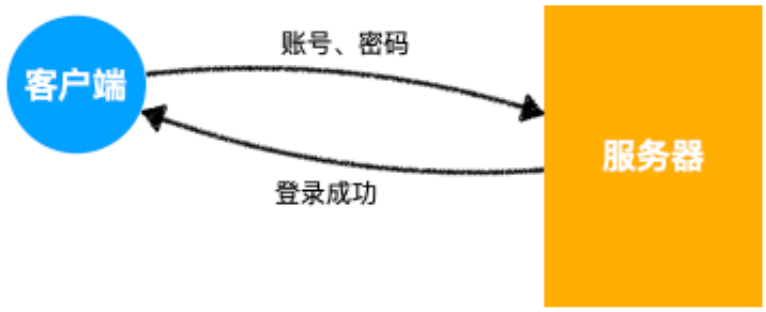
```
const resp = await fetch('url地址', { // 请求配置对象，可省略，省略则所有配置为默认值
  method: '请求方法', // 默认为GET
  headers: { // 请求头配置
    'Content-Type': 'application/json',
    'a': 'abc'
  },
  body: '请求体内容' // 请求体
}); // fetch会返回一个Promise，该Promise会在接收完响应头后变为fulfilled

resp.headers; // 获取响应头对象
resp.status; // 获取响应状态码，例如200
resp.statusText; // 获取响应状态码文本，例如OK
resp.json(); // 用json的格式解析即将到来的响应体，返回Promise，解析完成后得到一个对象
resp.text(); // 用纯文本的格式解析即将到来的响应体，返回Promise，解析完成后得到一个字符串
```

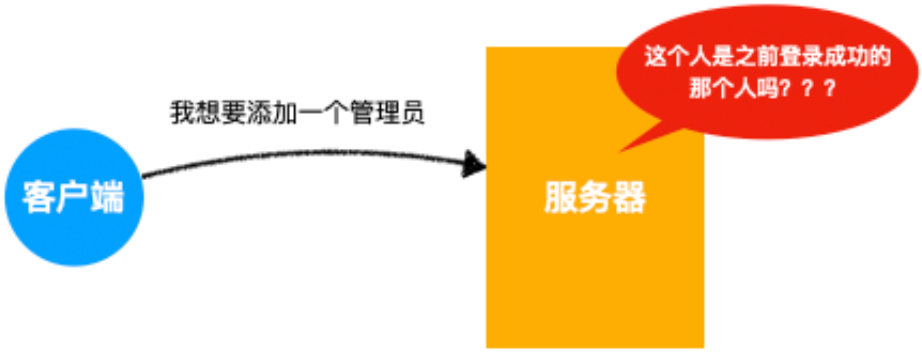
实战

token，令牌

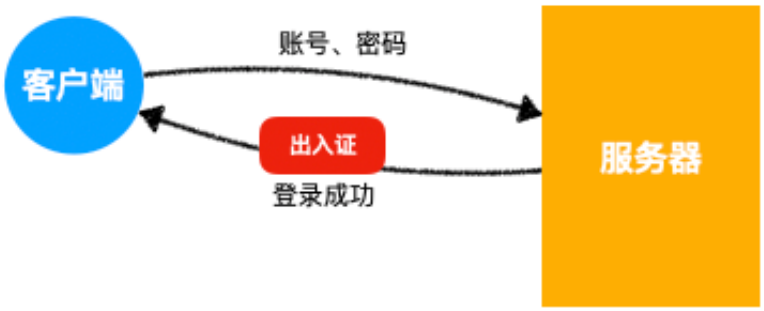
常出现在登录场景。

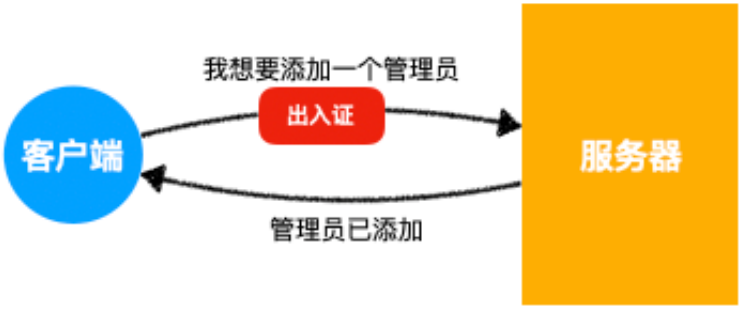
由于HTTP协议的特点，每次「请求-响应」都是独立的，这就会导致身份信息丢失的问题





这个问题可以使用token令牌解决





在使用AJAX时可以按照这样一种通用模式处理：

1. 在处理响应结果时，只要服务器给我的响应头中包含了`token`，就将其保存在`localStorage`中
2. 在请求时，只要`localStorage`中有`token`，就将其代入到响应头发送到服务器。

封装

封装AJAX请求方法，提供一个`request`函数：

```
request(method, url, [options]): Promise
```

- `method`，string
- `url`，string
- `options`，可选参数，object，请求配置，和fetch的配置相同
- `return`，返回一个Promise，Promise完成后会得到一个对象

```
{
  headers: { ... } // 响应头,
  data: { ... } // 响应体
}
```

`request` 函数能够自动处理 `token`

同时，它提供了 `request.get(url, [options])` 和 `request.post(url, data, [options])` 的简约调用方式。

完成该函数

思考

不要写代码，思考一个问题：

你开发了一个网站，你能否在你的网站中写入一段JS脚本，获取淘宝网站保存的 `localStorage` 的值?

如果你觉得应该进行限制，那么使用怎样的限制规则是合理的?