JS中的BOM

认识BOM

> BOM（Browser Object Model，浏览器对象模型）是一组描述浏览器及其组件的对象、方法和属性。BOM 主要处理浏览器窗口和与浏览器窗口相关的操作。与 DOM（文档对象模型）处理 HTML 和 XML 文档的结构不同，BOM 提供了与浏览器窗口进行交互的能力。
>
> BOM 的核心对象是 `window` 对象，它代表了浏览器的一个实例。当浏览器加载一个页面时，它会创建一个对应的 `window` 对象，该对象包含了许多与浏览器交互的方法和属性。在浏览器中，`window` 对象同时充当全局对象，也就是说，所有全局变量和函数都是 `window` 对象的属性和方法。
>
> BOM 的主要组成部分包括：
>
> 1. **window 对象**：代表浏览器窗口，提供了与浏览器窗口进行交互的方法和属性。一些常用的方法包括 `alert()`、`confirm()`、`prompt()` 等。
> 2. **location 对象**：提供了当前 URL 的信息以及一些方法，可以用于解析和操作 URL。例如，`location.href` 获取当前页面的 URL，`location.reload()` 用于重新加载当前页面。
> 3. **navigator 对象**：提供了浏览器和操作系统的信息。可以通过 `navigator.userAgent` 获取浏览器的用户代理字符串，以识别不同的浏览器。
> 4. **screen 对象**：提供了设备屏幕的信息，如屏幕的宽度、高度、可用宽度、可用高度等。这有助于为不同尺寸和分辨率的屏幕优化网页布局。
> 5. **history 对象**：提供了浏览器历史记录的信息以及用于导航的方法。例如，可以使用 `history.back()` 和 `history.forward()` 分别实现后退和前进功能。
> 6. **document 对象**：虽然 `document` 对象本身属于 DOM，但它在 BOM 中作为 `window` 对象的一个属性存在。`document` 对象表示当前加载的文档，包含了整个文档的内容和结构。
>
> 通过使用 BOM，开发者可以实现页面跳转、弹窗、获取设备信息、控制浏览器历史记录等与浏览器相关的操作，从而提高用户体验。

- **BOM：浏览器对象模型（Browser Object Model）**
  - 简称 **BOM**，由**浏览器提供的用于处理文档（document）之外的所有内容的其他对象**
  - 比如**navigator、location、history**等对象
- JavaScript有一个非常重要的运行环境就是浏览器
  - 而且浏览器本身又作为一个应用程序需要对其本身进行操作
  - 所以通常浏览器会有对应的对象模型（BOM，Browser Object Model）
  - 我们可以将BOM看成是连接JavaScript脚本与浏览器窗口的桥梁
- BOM主要包括一下的对象模型：
  - window：包括全局属性、方法，控制浏览器窗口相关的属性、方法
  - location：浏览器连接到的对象的位置（URL）
  - history：操作浏览器的历史
  - navigator：用户代理（浏览器）的状态和标识（很少用到）

- screen：屏幕窗口信息（很少用到）

window对象

- **window对象在浏览器中可以从两个视角来看待：**
  - 视角一：全局对象。

  ✓ 我们知道ECMAScript其实是有一个全局对象的，这个全局对象在**Node中是global**

  ✓ 在浏览器中就是**window对象**
  - **视角二：浏览器窗口对象。**

  ✓ 作为**浏览器窗口时，提供了对浏览器操作的相关的API**
- **当然，这两个视角存在大量重叠的地方，所以不需要刻意去区分它们：**
  - 事实上对于**浏览器和Node中全局对象名称不一样的情况**，目前已经指定了对应的标准，称之为**globalThis**，并且大多数现代浏览器都支持它
  - 放在**window对象**上的所有属性都可以被访问
  - 使用**var定义的变量会被添加到window对象中**
  - window默认给我们提供了全局的函数和类：**setTimeout、Math、Date、Object**等

window对象的作用

- **事实上window对象上肩负的重担是非常大的：**
  - 第一：**包含大量的属性**，localStorage、console、location、history、screenX、scrollX等等（大概60+个属性）
  - 第二：**包含大量的方法**，alert、close、scrollTo、open等等（大概40+个方法）
  - 第三：**包含大量的事件**，focus、blur、load、hashchange等等（大概30+个事件）
- **那么这些大量的属性、方法、事件在哪里查看呢?**
  - MDN文档：https://developer.mozilla.org/zh-CN/docs/Web/API/Window
- **查看MDN文档时，我们会发现有很多不同的符号，这里我解释一下是什么意思：**
  - **删除符号**：表示这个API已经废弃，不推荐继续使用了
  - **点踩符号**：表示这个API不属于W3C规范，某些浏览器有实现（所以兼容性的问题）
  - **实验符号**：该API是实验性特性，以后可能会修改，并且存在兼容性问题

window常见的属性

`window` 对象是浏览器窗口的代表，它提供了许多与浏览器窗口交互的属性和方法。以下是一些常见的 `window` 属性：

1. **window.document**：`document` 对象表示当前加载的 HTML 文档。通过 `window.document` 可以访问和操作文档内容、结构和样式。
2. **window.location**：`location` 对象包含了当前页面的 URL 信息。可以通过它获取和设置 URL，以及实现页面跳转、刷新等操作。

3. **window.navigator**：`navigator` 对象提供了关于浏览器和操作系统的信息。例如，可以通过 `window.navigator.userAgent` 获取浏览器的用户代理字符串。
4. **window.history**：`history` 对象表示浏览器的历史记录。可以通过它实现浏览器的前进、后退等操作。
5. **window.screen**：`screen` 对象提供了设备屏幕的信息，包括屏幕的宽度、高度、颜色深度等。这有助于为不同尺寸和分辨率的屏幕优化网页布局。
6. **window.innerWidth 和 window.innerHeight**：这两个属性分别表示浏览器窗口的视口宽度和高度，包括滚动条（如果存在的话）。
7. **window.outerWidth 和 window.outerHeight**：这两个属性表示浏览器窗口的外部尺寸，包括窗口边框、工具栏和滚动条。
8. **window.pageXOffset 和 window.pageYOffset**：这两个属性表示页面在水平和垂直方向上的滚动偏移量，即滚动条的位置。
9. **window.localStorage 和 window.sessionStorage**：这两个属性分别表示本地存储和会话存储对象，用于在浏览器中存储键值对数据。
10. **window.console**：`console` 对象用于输出调试信息。例如，通过 `window.console.log()` 可以在浏览器的控制台输出调试信息。

window常见的方法

`window` 对象提供了许多与浏览器窗口交互的方法。以下是一些常见的 `window` 方法：

1. **window.alert()**：显示带有一段消息和一个确认按钮的警告框。例如：`window.alert('Hello, world!')`。
2. **window.confirm()**：显示带有一段消息和两个按钮（确定和取消）的确认框。返回一个布尔值，表示用户是否点击了确定按钮。例如：`if (window.confirm('Are you sure?')) { /* 用户点击了确定 */ }`。
3. **window.prompt()**：显示带有一段消息、一个输入框和两个按钮（确定和取消）的对话框。返回用户输入的值，或在用户点击取消按钮时返回 `null`。例如：`const input = window.prompt('Please enter your name')`。
4. **window.open()**：在新窗口或新选项卡中打开指定的 URL。返回一个表示新窗口的 `window` 对象。例如：`const newWindow = window.open('https://www.example.com')`。
5. **window.close()**：关闭当前窗口或由 `window.open()` 打开的窗口。
6. **window.setTimeout()**：设定一个定时器，在指定的毫秒数之后执行指定的函数。返回一个定时器 ID，可以用于取消定时器。例如：`const timerId = window.setTimeout(() => { /* 执行代码 */ }, 1000)`。
7. **window.clearTimeout()**：取消由 `window.setTimeout()` 设置的定时器。例如：`window.clearTimeout(timerId)`。
8. **window.setInterval()**：设定一个重复定时器，在指定的毫秒数间隔内反复执行指定的函数。返回一个定时器 ID，可以用于取消定时器。例如：`const intervalId = window.setInterval(() => { /* 执行代码 */ }, 1000)`。
9. **window.clearInterval()**：取消由 `window.setInterval()` 设置的定时器。例如：`window.clearInterval(intervalId)`。

10. **window.requestAnimationFrame()**：告诉浏览器在下一次重绘前执行指定的回调函数。这对于动画和游戏开发非常有用，因为浏览器可以自动优化帧率。例如：`window.requestAnimationFrame(() => { /* 更新动画 */ })`。
11. **window.scrollTo()**：将页面滚动到指定的坐标（水平和垂直）。例如：`window.scrollTo(0, 500)`。
12. **window.scrollBy()**：将页面相对于当前位置滚动指定的距离（水平和垂直）。例如：`window.scrollBy(0, 100)`。

这些方法只是 `window` 对象众多方法中的一部分，但它们是开发过程中最常用的。通过这些方法，可以实现许多与浏览器窗口相关的功能，提高用户体验。

window常见的事件

`window` 对象支持许多事件，这些事件在不同的时间点或用户与浏览器窗口交互时触发。以下是一些常见的 `window` 事件及其简要描述：

`window` 对象支持许多事件，这些事件在不同的时间点或用户与浏览器窗口交互时触发。以下是一些常见的 `window` 事件及其简要描述：

1. **load**：当整个页面（包括所有依赖资源，如图片、样式表等）已完成加载时触发。常用于初始化页面内容或执行其他一次性任务。例如：

```
codewindow.addEventListener('load', () => {
  // 页面已加载，执行初始化操作
});
```

2. **unload**：当页面即将被卸载（例如，用户关闭标签页或导航到其他页面）时触发。用于执行清理操作，如取消网络请求、释放资源等。例如：

```
codewindow.addEventListener('unload', () => {
  // 页面即将卸载，执行清理操作
});
```

3. **beforeunload**：当页面即将被卸载时触发，可以在事件处理程序中返回一个字符串来提示用户是否确定离开页面。例如：

```
codewindow.addEventListener('beforeunload', (event) => {
  event.preventDefault();
  event.returnValue = 'Are you sure you want to leave?';
});
```

4. **resize**：当浏览器窗口大小发生变化时触发。常用于响应式设计或调整页面布局。例如：

```
codewindow.addEventListener('resize', () => {
  // 窗口大小发生变化，执行相应操作
});
```

5. **scroll**：当页面发生滚动时触发。可以用于监测滚动位置，实现懒加载、无限滚动等功能。例如：

```
codewindow.addEventListener('scroll', () => {
  // 页面滚动，执行相应操作
});
```

6. **popstate**：当浏览器历史发生变化（例如，用户点击后退按钮）时触发。常用于实现单页面应用（SPA）的导航功能。例如：

```
codewindow.addEventListener('popstate', (event) => {
  // 浏览器历史发生变化，执行相应操作
});
```

7. **focus** 和 **blur**：当浏览器窗口获得或失去焦点时触发。例如：

```
codewindow.addEventListener('focus', () => {
  // 窗口获得焦点，执行相应操作
});

window.addEventListener('blur', () => {
  // 窗口失去焦点，执行相应操作
});
```

8. **error**：当在页面上发生 JavaScript 错误时触发。可以用于捕获错误并向服务器报告。例如：

```
codewindow.addEventListener('error', (event) => {
  // 发生错误，执行相应操作
});
```

9. **message**：当其他窗口或 Web Worker 向当前窗口发送消息时触发。常用于实现跨域通信或与 Web Worker 交互。例如：

```
window.addEventListener('message', (event) => {
  // 收到消息，执行相应操作
});
```

10. **storage**：当本地存储（`localStorage` 或 `sessionStorage`）发生变化时触发。可以用于监控存储数据的变更，实现多个窗口之间的数据同步等功能。例如：

```
window.addEventListener('storage', (event) => {
  // 存储数据发生变化，执行相应操作
});
```

11. **online** 和 **offline**：当浏览器网络状态发生变化（在线或离线）时触发。可以用于根据网络状态调整应用的行为。例如：

```
window.addEventListener('online', () => {
  // 网络在线，执行相应操作
});

window.addEventListener('offline', () => {
  // 网络离线，执行相应操作
});
```

12. **DOMContentLoaded**：当页面的 DOM 结构已加载完成，但可能包括的样式、图像和子框架尚未完成加载时触发。这个事件在 `load` 事件之前触发，通常用于尽早执行某些操作。例如：

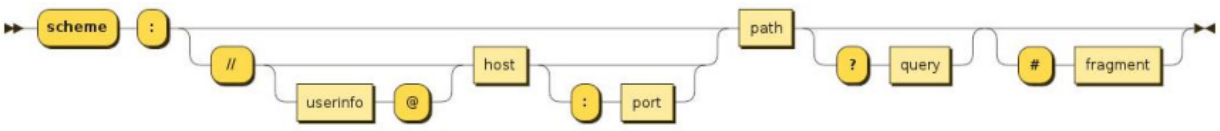
```
window.addEventListener('DOMContentLoaded', () => {
  // DOMContentLoaded，执行相应操作
});
```

location对象常见的属性

- **location对象用于表示window上当前链接到的URL信息。**
- **常见的属性有哪些呢?**
  - **href**: 当前window对应的超链接URL, 整个URL
  - **protocol**: 当前的协议
  - **host**: 主机地址
  - **hostname**: 主机地址(不带端口)
  - **port:** 端口
  - **pathname**: 路径
  - **search**: 查询字符串
  - **hash**: 哈希值
  - **username**：URL中的username（很多浏览器已经禁用）
  - **password**：URL中的password（很多浏览器已经禁用）

Location对象常见的方法

- **我们会发现location其实是URL的一个抽象实现：**



- **location有如下常用的方法：**
  - **assign**：赋值一个新的URL，并且跳转到该URL中
  - **replace**：打开一个新的URL，并且跳转到该URL中（不同的是不会在浏览记录中留下之前的记录）
  - **reload**：重新加载页面，可以传入一个Boolean类型

> 以一个用户登录验证的场景为例，当用户登录成功后，我们可能需要跳转到一个新的页面，例如用户的个人中心。同时，我们也希望阻止用户通过后退按钮回到登录页面。这时候，我们可以使用 `location` 对象的 `replace` 方法实现这个功能：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>登录示例</title>
</head>
<body>
    <form id="loginForm">
        <label for="username">用户名：</label>
        <input type="text" id="username" name="username" required>
        <br>
        <label for="password">密码：</label>
        <input type="password" id="password" name="password" required>
        <br>
        <button type="submit">登录</button>
    </form>
    <script>
        const loginForm = document.getElementById('loginForm');

        loginForm.addEventListener('submit', (event) => {
            event.preventDefault();
            const username = document.getElementById('username').value;
            const password = document.getElementById('password').value;

            // 假设验证成功
            if (username === 'user' && password === 'password') {
                // 使用 location.replace 方法跳转到用户中心页面
                location.replace('user_center.html');
            } else {
                alert('用户名或密码错误，请重试！');
            }
        });
    </script>
</body>
</html>
```

URLSearchParams

- **URLSearchParams 定义了一些实用的方法来处理 URL 的查询字符串**
    - 可以将一个字符串转化成**URLSearchParams**类型
    - 也可以将一个**URLSearchParams**类型转成字符串

```
var urlsearch = new URLSearchParams("name=why&age=18&height=1.88")
console.log(urlsearch.get("name")) // why
console.log(urlsearch.toString()) // name=why&age=18&height=1.88
```

- **URLSearchParams常见的方法有如下：**
  - **get**：获取搜索参数的值
  - **set**：设置一个搜索参数和值
  - **append**：追加一个搜索参数和值
  - **has**：判断是否有某个搜索参数
  - https://developer.mozilla.org/zh-CN/docs/Web/API/URLSearchParams
- **中文会使用encodeURIComponent和decodeURIComponent进行编码和解码**

```
decodeURIComponent('%E5%B9%BF%E5%B7%9E%E5%B8%82')
'广州市'
```

history对象常见属性和方法

> `history` 对象表示浏览器的历史记录堆栈。它提供了与浏览器历史记录进行交互的方法和属性。

- **history对象允许我们访问浏览器曾经的会话历史记录**
- 有两个属性：
  - length：返回当前 `history` 对象中的历史记录条目数量（包括当前页面）。
  - state：当前保留的状态值
- **有五个方法：**
  - **back()**：使浏览器加载历史记录中的前一个页面。等同于点击浏览器的后退按钮或调用 `history.go(-1)`
  - **forward()**：使浏览器加载历史记录中的下一个页面。等同于点击浏览器的前进按钮或调用 `history.go(1)`
  - **go()**：使浏览器加载历史记录中相对于当前页面偏移 `n` 个位置的页面。传入负数表示后退，正数表示前进。例如，`history.go(-2)` 表示后退两个页面，`history.go(3)` 表示前进三个页面
  - `pushState(stateObj, title, url)`：将给定的 `stateObj` 和 `url` 添加到历史记录堆栈中。该方法并不会导致页面刷新，而仅仅改变浏览器地址栏中的 URL。`title` 参数目前在大多数浏览器中并不起作用，可以传入空字符串。当用户在历史记录中导航时，可以使用 `popstate` 事件获取 `stateObj`。
  - `replaceState(stateObj, title, url)`：与 `pushState()` 类似，但 `replaceState()` 会用给定的 `stateObj` 和 `url` 替换当前历史记录条目，而不是添加一个新的条目。
- **history和hash目前是vue、react等框架实现路由的底层原理**
- **前端路由的核心：修改了URL，但是页面不刷新**
  - 前端路由的核心概念是在不触发页面刷新的情况下，修改URL并根据URL的变化来更新页面的内容。这可以通过以下两种技术实现：

1. **HTML5 History API**：使用 `history.pushState()` 和 `history.replaceState()` 方法修改浏览器的地址栏URL，同时不引发页面刷新。同时，监听 `popstate` 事件来捕获用户在历史记录中进行的导航操作。
2. **哈希路由（Hash-based Routing）**：通过修改URL中的哈希部分（`#`后的部分），实现对页面内容的更新。在这种方法中，我们监听 `hashchange` 事件，当URL的哈希部分发生变化时，根据新的哈希值来更新页面的内容。由于哈希值的变化不会引发页面刷新，因此这种方法在HTML5 History API之前就已经被广泛应用。

实现前端路由的原因：

1. **用户体验**：传统的服务器端路由在每次请求新页面时都会触发页面刷新。页面刷新会导致整个页面重新加载、渲染，这会造成用户体验的不连贯。前端路由可以避免页面刷新，使页面内容的更新更加平滑、快速，从而提高用户体验。
2. **性能优化**：使用前端路由，只需要加载和渲染发生变化的页面部分，而不是整个页面。这可以显著减少请求服务器的次数、降低网络传输的成本和浏览器的渲染负担，从而提高应用的性能。
3. **减轻服务器负担**：前端路由允许将更多的逻辑和资源放在客户端，从而减轻服务器端的负担。这样可以降低服务器的负担，提高应用的整体性能。
4. **前后端职责分离**：前端路由使得前端和后端可以更好地分离职责。前端负责渲染和用户交互，后端负责提供API和数据。这样的架构使得前端和后端可以独立开发和部署，提高了开发效率。

总之，前端路由通过修改URL但不触发页面刷新来实现对页面内容的更新，这有助于提高用户体验、优化性能、减轻服务器负担并实现前后端职责分离。

> 一个 `pushState()` 和 `popstate` 事件的示例：
>
> 在这个示例中，当用户点击按钮时，我们使用 `pushState()` 方法添加一个新的历史记录条目。我们将当前时间戳作为 `stateObj` 传入，并将 URL 中的查询参数设置为时间戳。当用户在历史记录中导航时（例如点击浏览器的后退/前进按钮），我们监听 `popstate` 事件，从 `event.state` 获取之前传入的时间戳，并在控制台上输出导航到的历史记录的时间戳。
>
> 在上述示例中，我们使用 `pushState()` 方法在浏览器的历史记录中添加新的条目，这将允许用户通过点击浏览器的后退/前进按钮来导航到这些记录。当用户导航时，我们监听 `popstate` 事件以捕获这些导航操作，并从 `event.state` 获取之前存储的时间戳。通过这种方式，我们可以在用户导航历史记录时跟踪和响应这些操作。
>
> 需要注意的是，使用 `pushState()` 和 `replaceState()` 修改历史记录并不会引起页面刷新，它们只会改变地址栏中的 URL。如果需要在导航历史记录时执行某些操作（例如更新页面内容），则需要使用 `popstate` 事件来捕获这些操作，并在事件处理程序中执行相应的操作。
>
> 总之，`history` 对象提供了一组属性和方法，使我们能够与浏览器的历史记录进行交互。这些方法包括用于导航历史记录（例如 `back()`、`forward()` 和 `go()`）的方法，以及用于修改历史记录（例如 `pushState()` 和 `replaceState()`）的方法。通过监听 `popstate` 事件，我们可以在用户导航历史记录时执行相应的操作。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>History 示例</title>
</head>
<body>
    <button id="pushStateBtn">添加历史记录</button>
    <script>
        const pushStateBtn = document.getElementById('pushStateBtn');

        pushStateBtn.addEventListener('click', () => {
            const timestamp = new Date().getTime();
            history.pushState({ timestamp }, '', `?t=${timestamp}`);
            console.log('添加历史记录：', timestamp);
        });

        window.addEventListener('popstate', (event) => {
            if (event.state) {
                console.log('导航到历史记录：', event.state.timestamp);
            }
        });
    </script>
</body>
</html>
```

navigator对象（很少使用）

- **navigator 对象表示用户代理的状态和标识等信息。**

属性/方法	说　明
locks	返回暴露 Web Locks API 的 LockManager 对象
mediaCapabilities	返回暴露 Media Capabilities API 的 MediaCapabilities 对象
mediaDevices	返回可用的媒体设备
maxTouchPoints	返回设备触摸屏支持的最大触点数
mimeTypes	返回浏览器中注册的 MIME 类型数组
onLine	返回布尔值，表示浏览器是否联网
oscpu	返回浏览器运行设备的操作系统和（或）CPU
permissions	返回暴露 Permissions API 的 Permissions 对象
platform	返回浏览器运行的系统平台
plugins	返回浏览器安装的插件数组。在 IE 中，这个数组包含页面中所有<embed>元素
product	返回产品名称（通常是"Gecko"）
productSub	返回产品的额外信息（通常是 Gecko 的版本）
registerProtocolHandler()	将一个网站注册为特定协议的处理程序
requestMediaKeySystemAccess()	返回一个期约，解决为 MediaKeySystemAccess 对象
sendBeacon()	异步传输一些小数据
serviceWorker	返回用来与 ServiceWorker 实例交互的 ServiceWorkerContainer
share()	返回当前平台的原生共享机制
storage	返回暴露 Storage API 的 StorageManager 对象
userAgent	返回浏览器的用户代理字符串
vendor	返回浏览器的厂商名称
vendorSub	返回浏览器厂商的更多信息
vibrate()	触发设备振动
webdriver	返回浏览器当前是否被自动化程序控制

screen对象（很少使用）

- **screen主要记录的是浏览器窗口外面的客户端显示器的信息：**
  - 比如屏幕的逻辑像素 screen.width、screen.height

属　性	说　明
availHeight	屏幕像素高度减去系统组件高度（只读）
availLeft	没有被系统组件占用的屏幕的最左侧像素（只读）
availTop	没有被系统组件占用的屏幕的最顶端像素（只读）
availWidth	屏幕像素宽度减去系统组件宽度（只读）
colorDepth	表示屏幕颜色的位数；多数系统是 32（只读）
height	屏幕像素高度
left	当前屏幕左边的像素距离
pixelDepth	屏幕的位深（只读）
top	当前屏幕顶端的像素距离
width	屏幕像素宽度
orientation	返回 Screen Orientation API 中屏幕的朝向

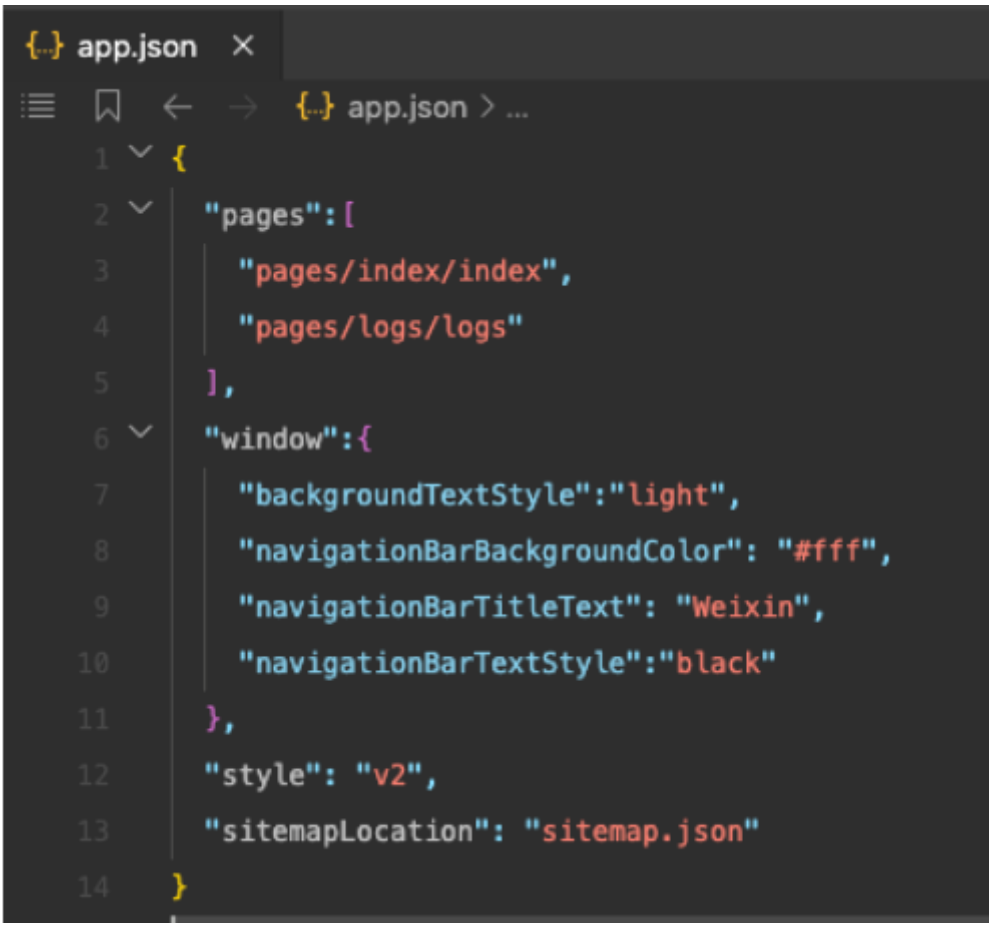
JSON的由来

> JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，它易于阅读和编写，同时也易于机器解析和生成。JSON由来可以追溯到JavaScript语言的发展过程。

> 在1990年代中期，JavaScript诞生于网景公司（Netscape），最初用作浏览器端的脚本语言。随着Web技术的发展，人们开始寻找一种轻量级、简单且易于解析的数据交换格式，以替代当时普遍使用的XML。尽管XML功能强大，但它较为繁琐，需要额外的解析工作，这在低性能的设备或者带宽有限的网络环境中可能会导致问题。
>
> JSON的提出源于JavaScript语言中的对象字面量表示法。Douglas Crockford发现了JavaScript对象字面量的潜力，认为它可以成为一种简洁、易于阅读和解析的数据交换格式。于是，他在2001年正式提出了JSON标准，并在2002年发布了JSON.org网站，详细介绍了JSON的规范和用法。
>
> JSON采用了完全独立于语言的文本格式，虽然源于JavaScript，但现已被许多编程语言所支持。由于其简洁性和易于解析，JSON在Web开发中得到了广泛应用，成为了当今最流行的数据交换格式之一。

- **在目前的开发中，JSON是一种非常重要的`数据格式`，它并不是`编程语言`，而是一种可以在服务器和客户端之间传输的数据格式。**
- **JSON的全称是JavaScript Object Notation（JavaScript对象符号）：**
  - JSON是由**Douglas Crockford构想和设计的一种轻量级资料交换格式，算是JavaScript的一个子集**
  - 但是虽然**JSON被提出来的时候是主要应用JavaScript**中，但是目前已经**独立于编程语言**，**可以在各个编程语言**中使用
  - 很多编程语言都实现了**将JSON转成对应模型的方式**
- **其他的传输格式：**
  - **XML**：在早期的网络传输中主要是使用XML来进行数据交换的，但是这种格式在解析、传输等各方面都弱于JSON，所以目前已经很少在被使用了
  - **Protobuf**：另外一个在网络传输中目前已经越来越多使用的传输格式是protobuf，但是直到2021年的3.x版本才支持JavaScript，所以目前在前端使用的较少
- **目前JSON被使用的场景也越来越多：**
  - 网络数据的传输JSON数据
  - 项目的某些配置文件
  - 非关系型数据库（NoSQL）将json作为存储格式

小程序的app.json



JSON基本语法

> JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，它基于JavaScript的对象字面量语法，但独立于任何编程语言。以下是JSON的基本语法：

1. **数据结构**：JSON有两种基本数据结构，分别是对象（Object）和数组（Array）。对象表示为一组键值对，而数组表示为一组有序的值。
2. **对象（Object）**：对象在JSON中表示为一组键值对，使用大括号`{}`括起来。键值对之间用逗号`,`分隔，键和值之间用冒号`:`分隔。键必须是字符串，而值可以是字符串、数字、布尔值、null、对象或数组。键必须用双引号`"`括起来。 示例：

```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "courses": ["math", "history"],
  "address": {
    "city": "New York",
    "country": "USA"
  }
}
```

3. **数组（Array）**：数组在JSON中表示为一组有序的值，使用中括号`[]`括起来。数组元素之间用逗号`,`分隔。数组中的值可以是字符串、数字、布尔值、null、对象或数组。 示例：

```
[  {    "name": "John Doe",    "age": 30  },  {    "name": "Jane Doe",    "age": 28  }]
```

4. **值（Value）**：JSON支持以下数据类型作为值：

> 1. 简单值：数字（Number）、字符串（String，不支持单引号）、布尔类型（Boolean）、null类型；
> 2. 对象值：由key、value组成，key是字符串类型，并且必须添加双引号，值可以是简单值、对象值、数组值
> 3. 数组值：数组的值可以是简单值、对象值、数组值

   - 字符串（String）：必须用双引号`"`括起来。
   - 数字（Number）：可以是整数或浮点数，不需要引号括起来。
   - 布尔值（Boolean）：可以是`true`或`false`，不需要引号括起来。
   - `null`：表示空值或不存在的值，不需要引号括起来。
   - 对象（Object）：使用大括号`{}`表示。
   - 数组（Array）：使用中括号`[]`表示。
5. **JSON要求**：JSON要求键名（Key）必须使用双引号`"`括起来，而值（Value）的表示则取决于其数据类型。此外，JSON的最外层必须是一个对象或数组。
6. **注释**：JSON规范本身不支持注释，但在某些实现或扩展中，可以使用`//`或`/* */`来添加注释。然而，在跨语言或跨平台场景下，添加注释可能会导致解析错误，所以通常建议避免在JSON中使用注释。

JSON序列化

- **某些情况下我们希望将JavaScript中的复杂类型转化成JSON格式的字符串，这样方便对其进行处理：**
  - 比如我们希望将一个对象保存到localStorage中
  - 但是如果我们直接存放一个对象，这个对象会被转化成 [object Object] 格式的字符串，并不是我们想要的结果

```
const obj = {
  name: "why",
  age: 18,
  friend: {
    name: "kobe"
  },
  hobbies: ["篮球", "足球", "乒乓球"]
}
```

Key	Value
info	[object Object]

JSON序列化方法

- **在ES5中引用了JSON全局对象，该对象有两个常用的方法：**
  - **stringify方法**：将JavaScript类型转成对应的JSON字符串
  - **parse方法**：解析JSON字符串，转回对应的JavaScript类型
- **那么上面的代码我们可以通过如下的方法来使用：**

1. `JSON.parse()`：这个方法用于将JSON格式的字符串转换成JavaScript对象。当你从服务器接收到JSON格式的数据时，通常需要使用这个方法将数据解析成JavaScript对象，以便在代码中进行处理。

   语法：

   ```
   JSON.parse(jsonString[, reviver]);
   ```

   参数：

   - `jsonString`：一个JSON格式的字符串。
   - `reviver`（可选）：一个可选的函数，用于在解析过程中转换或过滤对象的属性。函数接收两个参数：键名（key）和键值（value），并返回一个新的键值。如果返回`undefined`，则属性将被从结果中删除。

   示例：

   ```
   const jsonString = '{"name":"John", "age":30, "city":"New York"}';
   const jsonObj = JSON.parse(jsonString);
   console.log(jsonObj.name); // 输出 "John"
   ```

2. `JSON.stringify()`：这个方法用于将JavaScript对象或值转换成JSON格式的字符串。当你需要将JavaScript对象发送到服务器时，通常需要使用这个方法将对象序列化成JSON格式的字符串。

   语法：

   ```
   JSON.stringify(value[, replacer[, space]]);
   ```

   参数：

   - `value`：要转换成JSON字符串的JavaScript值或对象。
   - `replacer`（可选）：一个可选的函数或数组，用于在序列化过程中转换或过滤对象的属性。如果是函数，它接收两个参数：键名（key）和键值（value），并返回一个新的键值。如果返回`undefined`，则属性将被从结果中删除。如果是数组，数组中的字符串表示将被序列化的属性名。
   - `space`（可选）：一个可选的参数，用于控制输出的缩进。如果是数字，表示缩进的空格数（最大为10）；如果是字符串，表示缩进使用的字符（最多取前10个字符）。

   示例：

```
const jsonObj = {
  name: "John",
  age: 30,
  city: "New York"
};
const jsonString = JSON.stringify(jsonObj);
console.log(jsonString); // 输出 '{"name":"John","age":30,"city":"New York"}'
```

这两个方法是在处理JSON数据时的核心方法，分别用于将JSON字符串解析成JavaScript对象和将JavaScript对象序列化成JSON字符串。

Stringify的参数replace

- **JSON.stringify()** 方法将一个 JavaScript 对象或值转换为 JSON 字符串：
  - 如果指定了一个 **replacer 函数**，则可以**选择性地替换值**；
  - 如果**指定的 replacer 是数组**，则可**选择性地仅包含数组指定的属性**

Stringify的参数space

- 如果对象本身包含toJSON方法，那么会直接使用toJSON方法的结果

parse方法

- **JSON.parse() 方法用来解析JSON字符串，构造由字符串描述的JavaScript值或对象。**
  - 提供可选的 **reviver** 函数用以在返回之前对所得到的对象执行变换(操作)

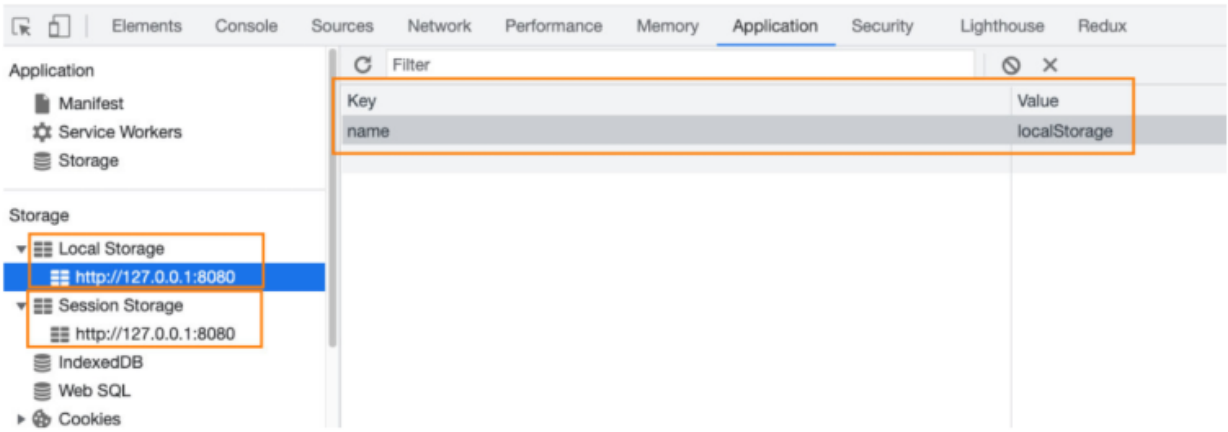
```
// 转回对象，并且转换某些值
const info2 = JSON.parse(objString, (key, value) => {
  if (key === "time") {
    return new Date(value)
  }
  return value
})
console.log(info2)
```

- JSON的方法可以帮我们实现对象的深拷贝：
  - 对象的拷贝、浅拷贝、深拷贝的概念放在JavaScript高级笔记中

认识Storage

- **WebStorage主要提供了一种机制，可以让浏览器提供一种比cookie更直观的key、value存储方式：**
  - **localStorage**：本地存储，提供的是一种**永久性的存储方法**，在关闭掉网页重新打开时，存储的内容依然保留
  - **sessionStorage**：会话存储，提供的是**本次会话的存储**，在关闭掉会话时，存储的内容会被清除

```
localStorage.setItem("name", "localStorage")
sessionStorage.setItem("name", "sessionStorage")
```



localStorage和sessionStorage的区别

- 我们会发现localStorage和sessionStorage看起来非常的相似。
- 那么它们有什么区别呢?
  - 验证一：关闭网页后重新打开，localStorage会保留，而sessionStorage会被删除
  - 验证二：在页面内实现跳转，localStorage会保留，sessionStorage也会保留
  - 验证三：在页面外实现跳转（打开新的网页），localStorage会保留，sessionStorage不会被保留

Storage常见的方法和属性

Web Storage API 提供了两种客户端存储数据的方式：`localStorage` 和 `sessionStorage`。这两种方式都具有类似的属性和方法。

1. localStorage：localStorage 用于永久性地存储数据在客户端，除非用户手动删除或者通过代码清除，否则数据会一直存在。
2. sessionStorage：sessionStorage 用于临时存储数据在客户端。当浏览器或者标签页关闭后，数据会被清除。

常见的方法和属性如下：

1. **setItem(key, value)**：将数据以键值对的形式存储。`key` 和 `value` 都需要是字符串类型。如果键名已存在，该方法会更新对应的值。

```
localStorage.setItem('name', 'John');
sessionStorage.setItem('age', '30');
```

1. **getItem(key)**：通过键名获取存储的数据。如果键名不存在，则返回 `null`。

```
const name = localStorage.getItem('name');
const age = sessionStorage.getItem('age');
```

1. **removeItem(key)**：通过键名删除存储的数据。

```
localStorage.removeItem('name');
sessionStorage.removeItem('age');
```

1. **clear()**：清除存储的所有数据。

```
localStorage.clear();
sessionStorage.clear();
```

1. **key(index)**：通过索引获取键名。如果索引不存在，则返回 `null`。

```
const firstKey = localStorage.key(0);
const secondKey = sessionStorage.key(1);
```

1. **length**：属性表示存储的键值对的数量。

```
const localStorageLength = localStorage.length;
const sessionStorageLength = sessionStorage.length;
```

这些方法和属性在 `localStorage` 和 `sessionStorage` 上都可以使用，它们提供了一种简便的方式在客户端存储和操作数据。但需要注意的是，Web Storage API 只适用于存储较小量的数据，因为浏览器对它们的存储空间有限制。