71-Dom-Bom-AdvancedJs-Typescript-Node

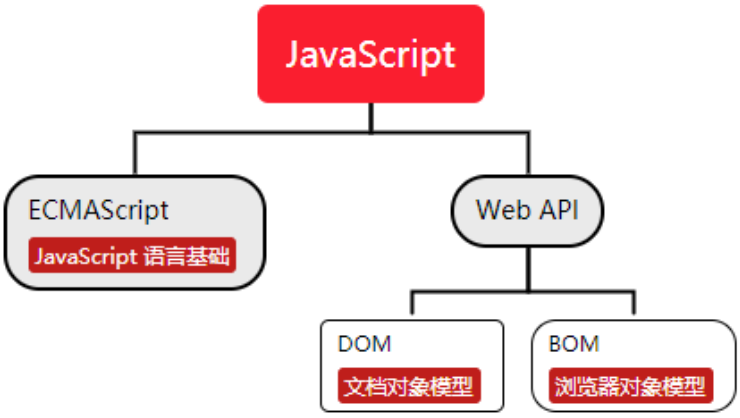
Dom & Bom

Reading

- https://www.yuque.com/fairy-era/xurq2q/qur8r6
- https://www.yuque.com/fairy-era/xurq2q/kcc9ge
- https://www.yuque.com/fairy-era/xurq2q/udsvag
- https://www.yuque.com/fairy-era/xurq2q/dll0xx
- https://www.yuque.com/fairy-era/xurq2q/vsx0tf

1.1 作用和分类

- 作用：Web API 就是使用 JavaScript 去操作 HTML 和浏览器。
- 分类：
    - DOM（文档对象模型）
    - BOM（浏览器对象模型）



许大仙

1.2 什么是 DOM？

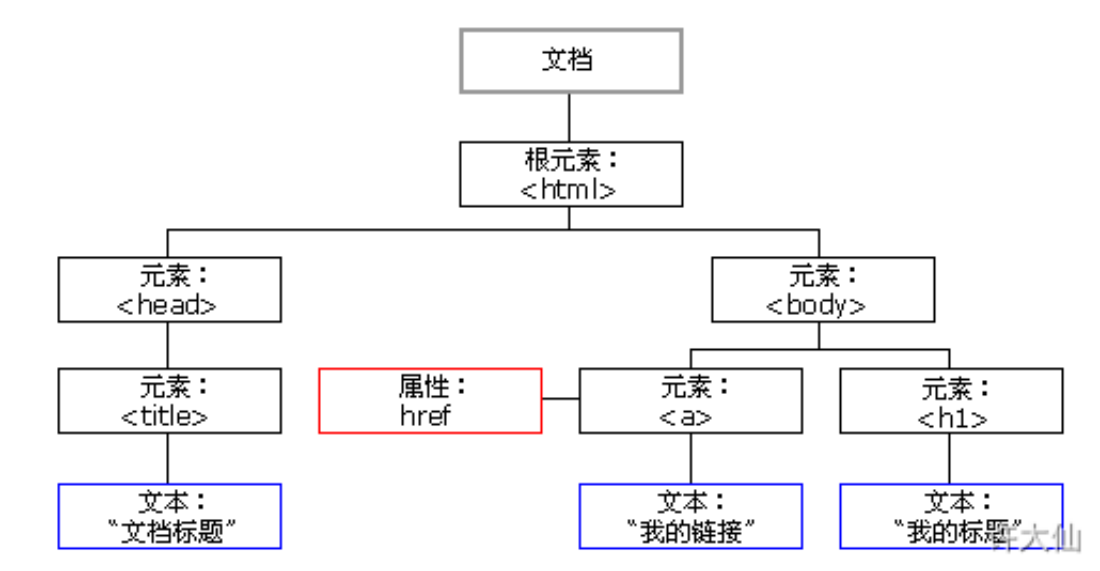
DOM（Document Object Model，文档对象模型）是用来呈现以及与任意 HTML 或 XML 文档交互的 API。换言之，DOM 是浏览器提供的一套专门用来 `操作网页内容` 的功能。

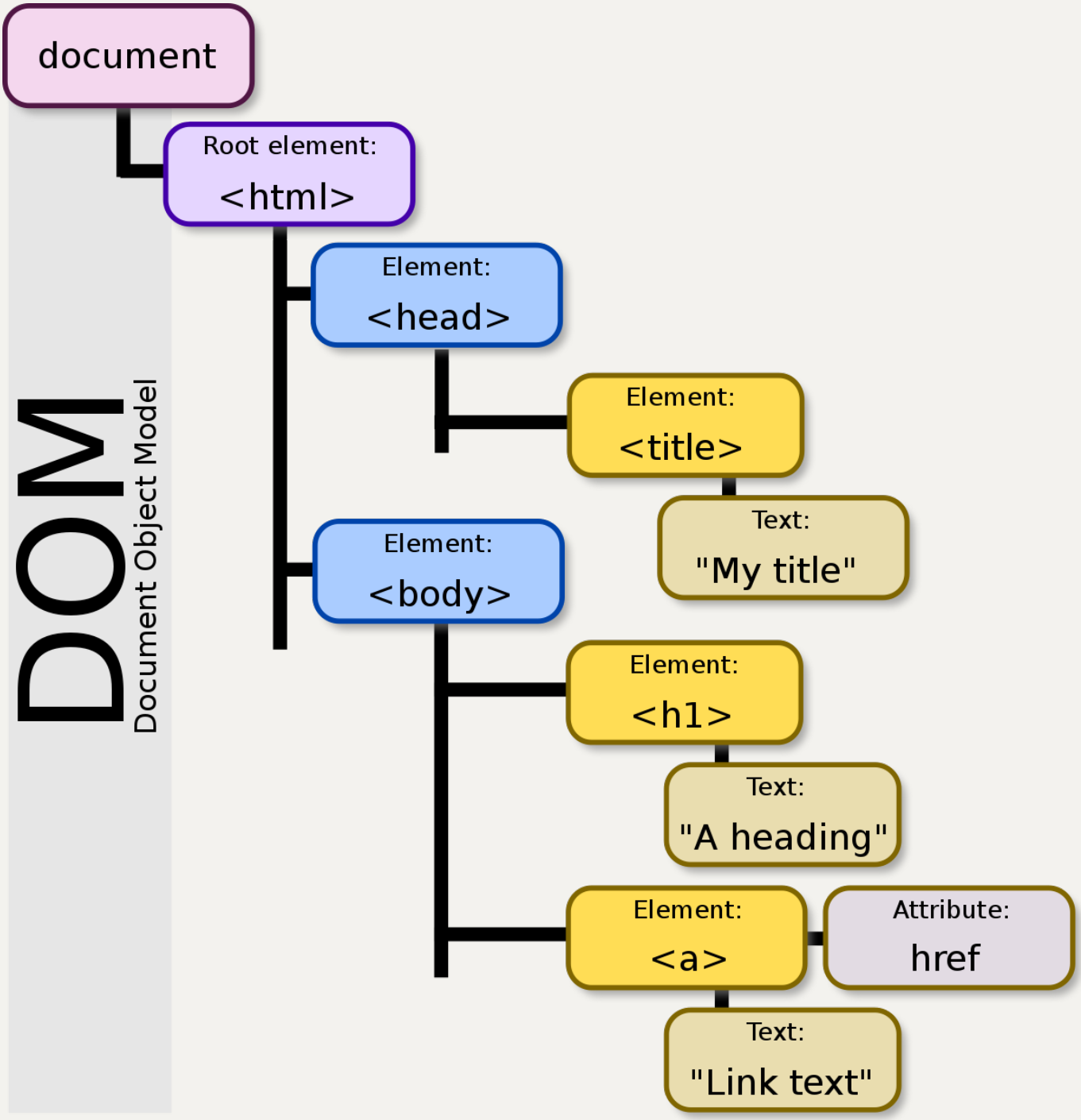
DOM 作用：开发网页内容特效和实现用户交互。

把html抽象成dom, 用Javascript可以操作dom，改变dom的内容，则改变了html的内容，此时html就不再是固定不变的内容了，网站就可以互动了

1.3 Dom Tree

- DOM 树是将 HTML 文档以树状结构直观的表现出来，我们称之为文档树或 DOM 树
- DOM 树是描述网页内容关系的名词
- DOM 树的作用是直观的体现了标签和标签之间的关系





1.4 Dom 对象

- DOM 对象：浏览器会根据 HTML 标签生成 JS 对象
    - 所有的标签属性都可以在这个对象上面找到
    - 修改这个对象的属性会自动映射到标签身上
- DOM 的核心思想：将网页内容当做 `对象` 来处理

- document 对象
    - 是 DOM 里面提供的一个 `对象`
    - 其提供的属性和方法都是用来 `访问和操作网页中的内容` ，
      如： `document.write()`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0"
name="viewport">
    <title>Title</title>
</head>
<body>
    <!-- 标签或元素 -->
    <button>点击</button>

    <script>

        /* btn 就是 DOM 对象 */
        let btn = document.querySelector('button');
        console.log(typeof btn); // object
        console.dir(btn); // object
        btn.innerHTML = '呵呵哒';
    </script>
</body>
</html>
```

3.0 设置修改dom 内容

div.innerText = '那还用说~';

对象.innerHTML 属性

4.1 设置、修改元素常见属性

img.src = './images/zxy.jpg';

div.style.backgroundColor = 'pink'

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
    <style>
        div {
            width: 200px;
            height: 200px;
            background-color: red;
        }
    </style>
</head>
<body>

    <div>heheda</div>

    <script>
        let div = document.querySelector('div');
        div.style.backgroundColor = 'pink'
    </script>
</body>
</html>
```

5.2 定时器 -- 间歇函数

```
let id = setInterval(函数,间隔函数);
```

- 作用：每隔一段时间调用这个函数。
- 间隔时间单位是毫秒。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        div {
            width: 200px;
            height: 200px;
            background-color: pink;
        }
    </style>
</head>

<body>

    <div>用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议用户协议</div>
    <button class="btn">我已经阅读用户协议(6)</button>

    <script>
        let btn = document.querySelector('button');
        // 刚开始的时候，将按钮禁止
        btn.disabled = true;
```

```
        // 定义变量，用来计数
        let times = 6;
        // 开启定时器
        let timer = setInterval((arg) => {
            if (num === 0) {
                clearInterval(timer);
                btn.innerHTML = '我同意该协议';
                btn.disabled = false;
            } else {
                btn.innerText = `我已经阅读用户协议(${times--})`;
            }
        }, 1000);

    </script>
</body>

</html>
```

1.1.1 什么是事件?

- https://www.yuque.com/fairy-era/xurq2q/kcc9ge

> *事件是在编程时系统内发生的 `动作` 或者发生的 `事情` ，如：用户在网页上 `单击` 了一个按钮。*

1.1.2 什么是事件监听?

> *就是让程序检测是否有事件产生，一旦有事件触发，就立即调用一个函数做出响应，也称为 `注册事件` 。*

```
元素.addEventListener('事件',要执行的函数);
```

1.1.3 事件监听的三要素

- 事件源：那个 DOM 元素被事件触发了，所以需要获取 DOM 元素。
- 事件：用什么方式触发，如：鼠标单击 click 、鼠标经过 mouseover 等。
- 事件调用的函数：要做的事情。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>

    <button>点我</button>

    <script>
        /* 获取元素 */
        let btn = document.querySelector('button');
        /* 注册事件 */
        btn.addEventListener('click', function () {
            alert('(*^▽^*)，点我了啦~');
        })
    </script>

</body>

</html>
```

Question: 上面和下面的代码中，那个是事件源，哪个是事件，哪个是事件调用的函数?

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>随机点名简单版</title>
    <style>
        * {
            margin: 0;
            padding: 0;
        }

        .box {
            width: 200px;
            height: 50px;
            margin: 50px auto;
        }

        .box .name {
            text-align: center;
            height: 50px;
            line-height: 50px;
            border: 1px solid pink;
        }

        .box button {
            transform: translateX(50%);
            width: 50%;
        }
    </style>
</head>

<body>
```

```
36
37      <div class="box">
38          <div class="name">开始点名了。。。</div>
39          <button class="btn">点击随机点名</button>
40      </div>
41
42      <script>
43          let arr = ['赵云', '黄忠', '关羽', '张飞', '马超', '刘备', '曹
    操'];
44
45          function getRandom(min, max) {
46              return Math.floor(Math.random() * (max - min + 1)) +
    min;
47          }
48
49          // 获取元素
50          let btn = document.querySelector('.btn');
51          let name = document.querySelector('.name');
52          // 注册事件
53          btn.addEventListener('click', function () {
54              let random = getRandom(0, arr.length - 1);
55              name.innerHTML = arr[random];
56              // 删除数组中的元素
57              arr.splice(random, 1);
58              // 如果数组里面没有元素，就需要禁用按钮
59              if (arr.length === 0) {
60                  btn.disabled = true;
61                  btn.innerHTML = '点名结束';
62              }
63          })
64      </script>
65
66  </body>
67
68  </html>
```

1.3 事件类型

- 鼠标触发：
    - 鼠标点击：click
    - 鼠标经过：mouseenter
    - 鼠标离开：mouseleave
- 表单获取光标：
    - 获取焦点：focus
    - 失去焦点：blur
- 键盘触发：
    - 键盘按下触发：keydown
    - 键盘抬起触发：keyup
- 表单输入触发：
    - 用户输入事件：input

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        * {
            margin: 0;
            padding: 0;
        }

        table {
            border-collapse: collapse;
            border-spacing: 0;
```

```
            border: 1px solid #c0c0c0;
            width: 500px;
            margin: 100px auto;
            text-align: center;
        }

        th {
            background-color: #09c;
            font: bold 16px "微软雅黑";
            color: #fff;
            height: 24px;
        }

        td {
            border: 1px solid #d0d0d0;
            color: #404060;
            padding: 10px;
        }

        .allCheck {
            width: 80px;
        }
    </style>
</head>

<body>
    <table>
        <tr>
            <th class="allCheck">
                <input type="checkbox" name="" id="checkAll">
<span class="all">全选</span>
            </th>
            <th>商品</th>
            <th>商家</th>
            <th>价格</th>
        </tr>
```

```
        <tr>
            <td>
                <input type="checkbox" name="check" class="ck">
            </td>
            <td>小米手机</td>
            <td>小米</td>
            <td>￥1999</td>
        </tr>
        <tr>
            <td>
                <input type="checkbox" name="check" class="ck">
            </td>
            <td>小米净水器</td>
            <td>小米</td>
            <td>￥4999</td>
        </tr>
        <tr>
            <td>
                <input type="checkbox" name="check" class="ck">
            </td>
            <td>小米电视</td>
            <td>小米</td>
            <td>￥5999</td>
        </tr>
    </table>

    <script>
        // 获取元素
        let checkAll = document.querySelector('#checkAll');
        let cks = document.querySelectorAll('.ck');
        let all = document.querySelector('.all');
        // 点全选按钮添加点击事件
        checkAll.addEventListener('click', function () {
            // 将全选按钮的点击状态赋值给下面每个按钮
            cks.forEach(ck => {
                ck.checked = this.checked;
```

```
            });
            // 如果全选按钮处于选中状态，就可以将文字改为取消
            if (this.checked) {
                all.innerHTML = '取消';
            } else {
                all.innerHTML = '全选';
            }
        });
        // 其余按钮添加点击事件，一旦有一个没有选中，全选取消
        for (let i = 0; i < cks.length; i++) {
            let ck = cks[i];
            // 给每个按钮添加点击事件
            ck.addEventListener('click', function () {
                // 只要点击任何一个小按钮，都需要遍历所有的小按钮
                for (let j = 0; j < cks.length; j++) {
                    let c = cks[j];
                    if (!c.checked) {// 如果有按钮没有被选中，则退出循环
                        checkAll.checked = false;
                        all.innerHTML = '全选';
                        return;
                    }
                }
                // 循环结束后，将全选状态设置为 true
                checkAll.checked = true;
                all.innerHTML = '取消';
            });
        }

    </script>

</body>

</html>;
```

第二章：高阶函数

- `高阶函数` 可以简单理解为函数的高级应用，JavaScript 中的函数可以被当做 `值` 来对待，基于这个特性实现函数的高级应用。
- `值` 就是 JavaScript 中的数据，如：数值、字符串、布尔、对象等
- 高级函数分类
    - 函数表达式
    - 回调函数

2.2 函数表达式

把函数赋给一个变量。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>函数表达式</title>
</head>

<body>

    <script>
        /* 高阶函数：函数的高级用法，将函数当值来使用 */
        let num = 10;
        // 函数表达式
        let fn = function (num = 0, num2 = 0) {
            return num1 + num2;
        };
        let result = fn(1, 2);
```

```
        console.log(result);
    </script>

</body>

</html>
```

2.3 回调函数

> *如果将函数 A 作为参数传递给函数 B 时，我们就称函数 A 为 `回调函数`*

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>

    <script>
        function fn(){
            console.log('我是回调函数');
        }

        // fn 传递给了 setInterval , fn 就是回调函数
        setInterval(fn, 500);
    </script>

</body>

```

```
</html>
```

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>

    <button>点我</button>

    <script>
        let btn = document.querySelector('button');
        btn.addEventListener('click', function () {
            // 此处的 function(){} 也是回调函数
        })
    </script>

</body>

</html>
```

第三章：环境对象（this）

- 环境变量指的是函数内部特殊的变量 `this` ，它代表着当前函数运行时所处的环境
- this 的作用：可以让我们的代码更加简洁
- 函数调用的方式不同，this 指向的对象也不同。换言之，谁调用，this 指向谁
- 如果直接调用函数，就相当于 window.函数，所以 this 指向 window

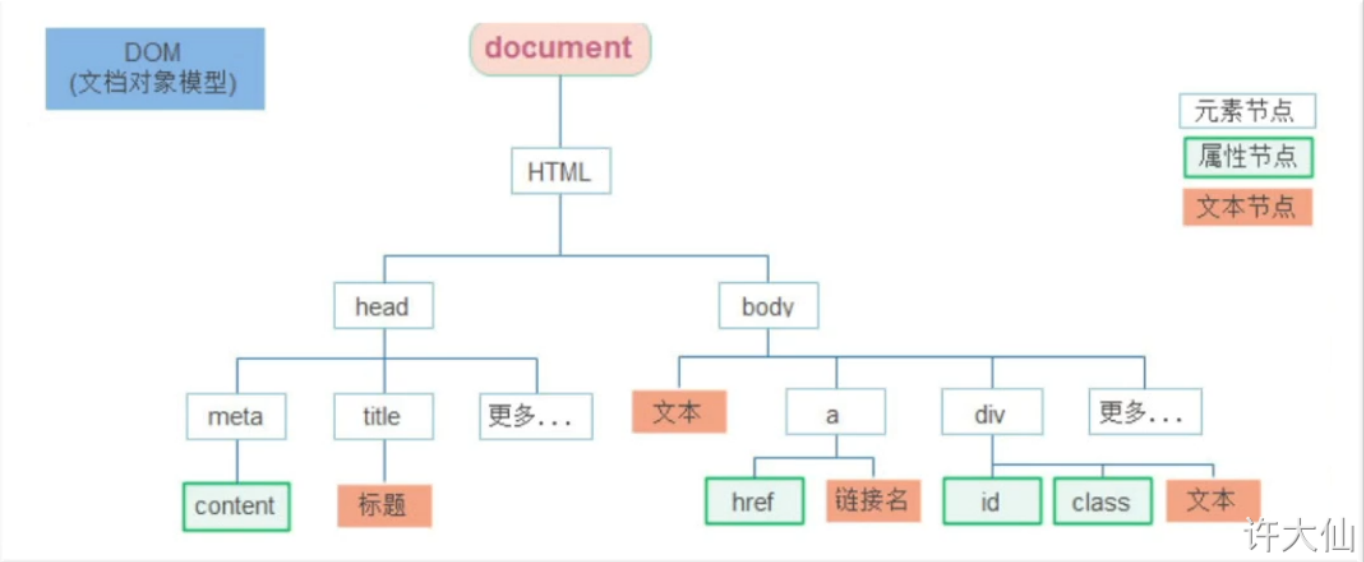
```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>

    <button>点我</button>

    <script>
        function fn() {
            console.log(this); // Window
        }

        fn();

        let btn = document.querySelector('button');
        btn.addEventListener('click', function () {
            console.log(this); // button
        })
    </script>

</body>

</html>
```

DOM--节点操作

- https://www.yuque.com/fairy-era/xurq2q/udsvag#50138368

1.1 DOM 节点

> *DOM 节点：DOM 树中的每一个内容都称之为节点*



- 1.2.1 查找父节点
- 1.2.2 查找子节点
- 1.2.3 查找兄弟节点
- 1.3.2 创建节点
- 1.3.3 追加节点
- 1.3.4 复制节点（克隆节点）
- **1.4 删除节点**

```
子元素.parentNode

父元素.children
```

```

元素.nextElementSibling
元素.previousSibling

// 创建一个新的元素节点
document.createElement('标签名');

父元素.appendChild(要插入的元素);
父元素.insertBefore(要插入的元素,在那个元素前面);

// 克隆一个已有的元素节点
元素.cloneNode(布尔值)

父元素.removeChild(要删除的元素)
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
    <style>
        li {
            margin: 20px auto;
        }
    </style>
</head>
<body>

    <ul>
        <li>我是孩子1</li>
        <li>我是孩子2</li>
        <li>我是孩子3</li>
```

```
        <li>我是孩子4</li>
        <li>我是孩子5</li>
    </ul>

    <button>点我</button>

    <script>
        let btn = document.querySelector('button');

        btn.addEventListener('click', function () {
            let ul = document.querySelector('ul');
            let lis = ul.children;
            for (let i = 0; i < lis.length; i++) {
                let li = lis[i];
                console.log(li.innerHTML);
                li.style.backgroundColor = 'pink';
            }
        });

    </script>

</body>
</html>
```

1.1 获取事件对象

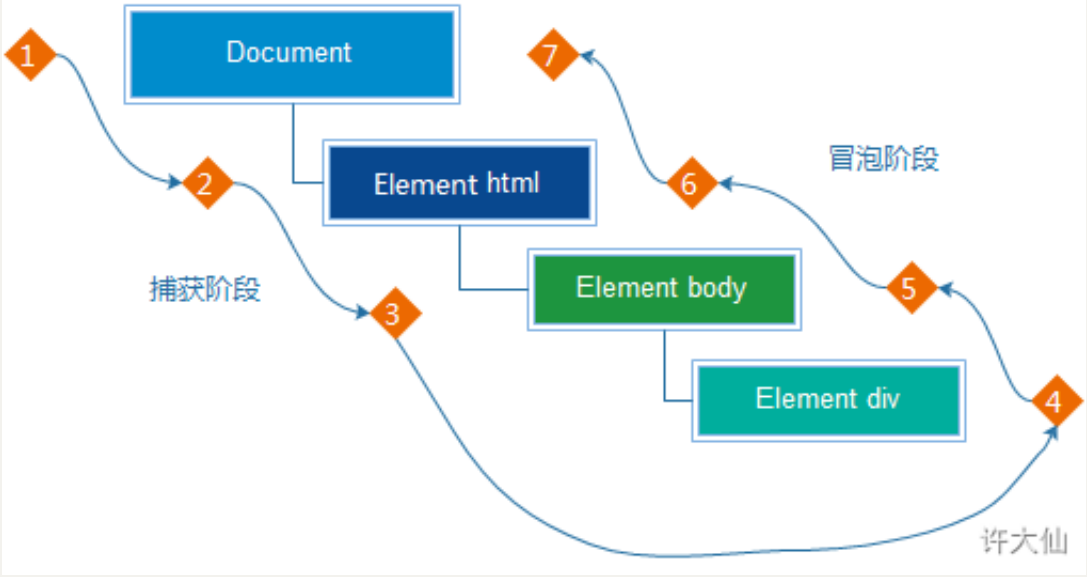
- 事件对象是什么?
    - 事件对象也是一个对象，这个对象里面有事件触发时的相关信息
    - 如：鼠标点击事件中，事件对象就保存了鼠标点在哪个位置等信息
- 获取事件对象
    - 在事件绑定的回调函数的第一个参数就是事件对象
    - 一般命名为 event 、e 等

```
元素.addEventListener('click',function(e){
    // e 就是事件对象
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>

    <button>点我</button>

    <script>
        let btn = document.querySelector('button');
        btn.addEventListener('click', function (event) {
            // event 就是事件对象
            console.log(event);
        });
    </script>

</body>
</html>
```

2.1 事件流和两个阶段说明

> *简单理解：当一个元素触发事件后，会依次向上调用所有父级元素的同名事件*



- 事件冒泡：当一个元素的事件被触发的时候，同样的事件将会在该元素的所有祖先元素中依次被触发，这一过程被称为事件冒泡。
- 默认情况下存在的就是事件冒泡

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
    <style>
        .father {
            position: relative;
            width: 500px;
            height: 500px;
            background-color: pink;
            margin: 100px auto;
        }

        .son {
            position: absolute;
            left: 50%;
```

```
            top: 50%;
            transform: translate(-50%, -50%);
            width: 250px;
            height: 250px;
            background-color: red;
        }
    </style>
</head>
<body>

    <div class="father">
        我是父亲
        <div class="son">我是儿子</div>
    </div>

    <script>

        /*
            点击儿子，依次会执行 儿子、父亲、爷爷的点击事件。
        */

        let son = document.querySelector('.son');
        let father = document.querySelector('.father');

        son.addEventListener('click', function () {
            alert('儿子');
        });

        father.addEventListener('click', function () {
            alert('父亲');
        });

        document.addEventListener('click', function () {
            alert('爷爷');
        });
    </script>
```

```
56
57 </body>
58 </html>
```

- 事件捕获：从 DOM 根元素开始去执行对应的事件（从外到里）
- 事件捕获需要写对应的代码才能看到效果
    - `元素.addEventListener(事件类型，事件处理函数，是否使用捕获机制)`
- 说明
    - addEventListener 的第三个参数传入 true ，表示开启事件捕获（实际开发中很少使用）
    - 如果传入 false 代表开启事件冒泡，默认就是 false

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta content="IE=edge" http-equiv="X-UA-Compatible">
6      <meta content="width=device-width, initial-scale=1.0"
   name="viewport">
7      <title>Title</title>
8      <style>
9          .father {
10             position: relative;
11             width: 500px;
12             height: 500px;
13             background-color: pink;
14             margin: 100px auto;
15         }
16
17         .son {
18             position: absolute;
19             left: 50%;
20             top: 50%;
21             transform: translate(-50%, -50%);
```

```
            width: 250px;
            height: 250px;
            background-color: red;
        }
    </style>
</head>
<body>

    <div class="father">
        我是父亲
        <div class="son">我是儿子</div>
    </div>

    <script>
        /*
            点击儿子，依次会执行 爷爷、父亲、儿子的点击事件。
        */

        let son = document.querySelector('.son');
        let father = document.querySelector('.father');

        son.addEventListener('click', function () {
            alert('儿子');
        }, true);

        father.addEventListener('click', function () {
            alert('父亲');
        }, true);

        document.addEventListener('click', function () {
            alert('爷爷');
        }, true);
    </script>

</body>
</html>
```

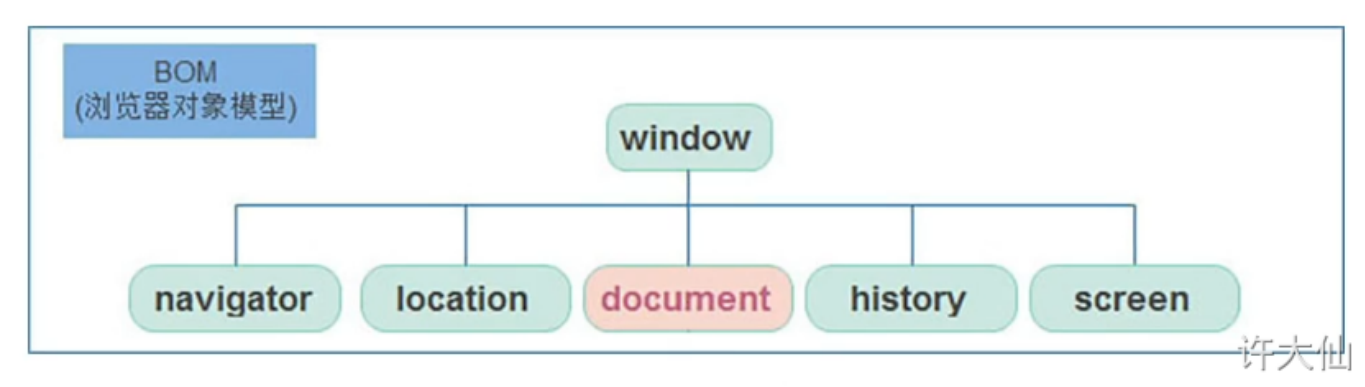
2.3 阻止事件流动

- 因为默认就有事件冒泡的存在，所以很容易导致子元素的事件影响到父元素的事件。
- 如果想将事件限制在当前元素内，就需要阻止事件流动，要想阻止事件流动必须先拿到事件对象。
- 语法
    - `事件对象.stopPropagation();`

Bom 操作浏览器

1.1 BOM（浏览器对象模型）

> *DOM（Browser Object Model）是浏览器对象模型*



- window 是浏览器内置的全局对象，我们学习的所有 WebAPI 的只是内容都是基于 window 对象实现的。
- window 对象下包含了 navigator 、location 、document 、history 、screen 5 个属性，即所谓的 BOM（浏览器对象模型）。
- document 是实现 DOM 的基础，它其实是依附于 window 的属性。

1.4 JS 执行机制

1.4.1 JS 是单线程

- JavaScript 语言的一大特点就是 `单线程` ，也就是说，`同一个时间只能做一件事情` 。这是因为 JavaScript 这门脚本语言诞生的使命所致 -- JavaScript 是为处理页面中用户的交互以及操作 DOM 而诞生的。如：我们对某个 DOM 元素进行添加和删除操作，不能同时进行，必须先进行添加，然后才能删除。
- 单线程就意味着，所有的任务需要排队，前一个任务结束，才会执行后一个任务。这样锁导致的问题就是：如果 JS 执行的时间过长，就会造成页面的渲染不连贯，会产生页面渲染加载阻塞的感觉。

1.4.2 同步和异步

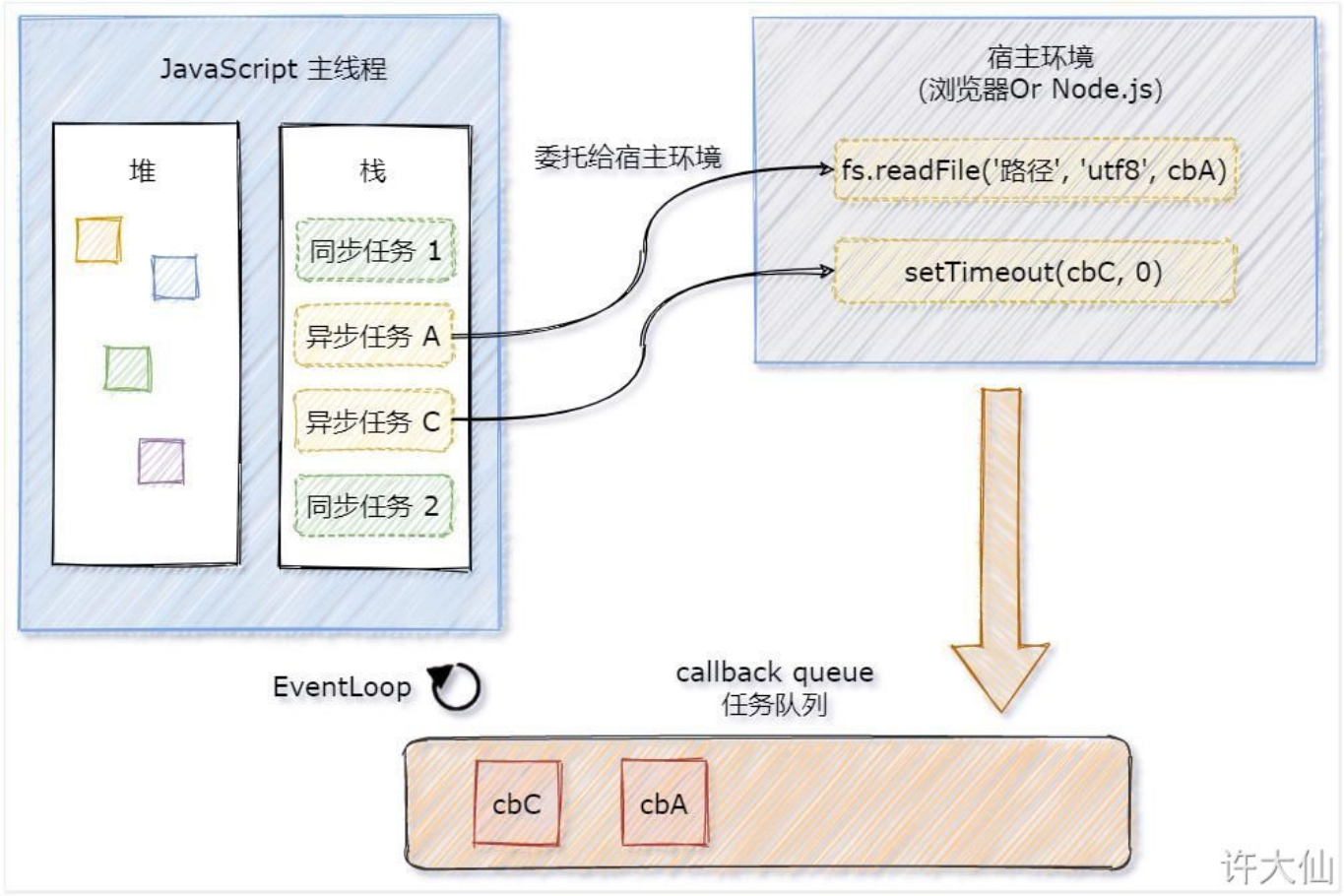
- 为了解决这个问题，利用多核 CPU 的计算能力，HTML5 提出了 Web Worker 标准，允许 JavaScript 脚本创建多个线程，于是，JS 中出现了 `同步` 和 `异步` 。
- 同步：同一个任务结束后再去执行后一个任务，程序的执行顺序和任务的排列顺序是一致的、同步的。如：做饭的同步做法就是先烧水，等水烧好之后（10分钟之后），再去切菜，炒菜。
- 异步：在做一件事情的时候，因为这件事情要花费很长时间，在做这件事情的同时，还可以做其他事情。如：做饭的异步做法就是在烧水的同时，利用这 10 分钟，去切菜，炒菜。

比如Walmart网站，打开时候有很多图片要加载，获取每一张图片都是I/O操作，如果获取完第一张图片，再去获取第二张图片，那么获得30张图片要很久。

- 同步任务：同步任务都是在主线程上执行，形成一个 `执行栈` 。
- 异步任务
    - JS 的异步任务是通过回调函数实现的
    - 异步任务的类型
        - 普通事件，如：click 、resize 等
        - 资源加载，如：load 、error 等
        - 定时器，如：setTimeout、setInterval 等

- 异步任务是添加到 任务队列 中
-

1.4.3 JS 的执行机制（Event Loop. 大重点）



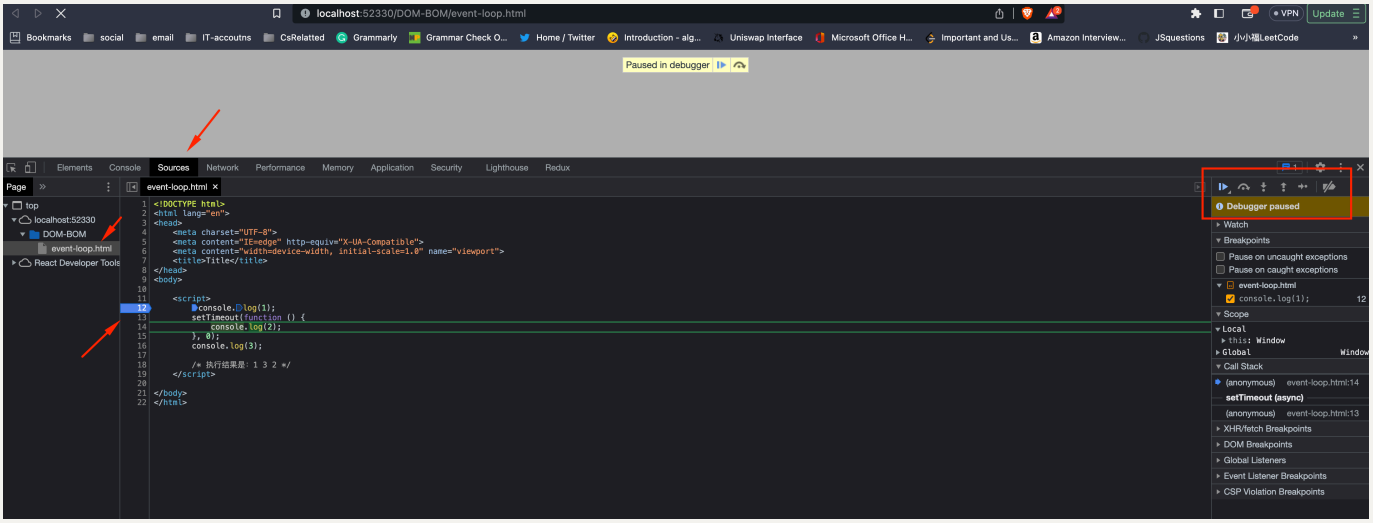
1. 先执行 执行栈中的同步任务
2. 异步任务放入任务队列中
3. 一旦执行栈中的所有同步任务执行完毕，系统就会按照次序读取 任务队列 中的异步任务，于是被读取的异步任务结束等待状态，进行执行栈，开始执行

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
```

```
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>

    <script>
        console.log(1);
        setTimeout(function () {
            console.log(2);
        }, 1000);
        console.log(3);

        /* 执行结果是: 1 3 2 */
    </script>

</body>
</html>
```

Question: 根据上图， 来逐步描述上述代码的过程。

How to debug



local storage

https://www.yuque.com/fairy-era/xurq2q/vsx0tf#cd05d52f

正则表达式

有能力的同学学会，基础弱的，知道是干什么的即可。面试时候可以说忘记语法了。

https://www.yuque.com/fairy-era/xurq2q/phcy4d#4658e6d1

Advanced JavaScript

Reading

- https://www.yuque.com/fairy-era/xurq2q/mmpkuv#b7b8e1d0

作用域、函数和解构赋值

Closure (闭包)

> *闭包是一种特殊的函数，使用闭包能够访问函数作用域中的变量。从代码形式上看闭包就是一个有返回值的函数*

Question: 正常情况下，当一个函数运行完毕，那么该函数内的变量会怎么样?

Answer: 会被清理掉

Question：如果我们想在该函数运行完之后，该变量依然继续用，该怎么做?

Answer: 把变量放到class里，当作全局变量，就像java的attributes一样

例如我们写一个计数器。

- Class版本。

```
class Counter {
  constructor() {
    this.count = 0;
  }

  increment() {
    this.count += 1;
    return this.count;
  }
}

const counter = new Counter();

console.log(counter.increment());  // 输出: 1
console.log(counter.increment());  // 输出: 2
console.log(counter.increment());  // 输出: 3
```

- 闭包版本

```
function createCounter() {
  let count = 0;

  // 注意返回的是函数
  return function() {
    count += 1;
    return count;
  };
}

const counter = createCounter();

console.log(counter());  // 输出: 1
console.log(counter());  // 输出: 2
```

```
15  console.log(counter());  // 输出: 3
```

Question：那么为什么还要使用闭包而不使用class呢?

Answer: JavaScript 是一种基于原型的编程语言，尽管在 ES6 中引入了类 (class) 这个概念，但它仍然是对原型的语法糖。例如，假设我们有一个函数，需要频繁地创建和销毁对象。使用类，我们需要为每个对象分配新的内存。但是使用闭包，我们可以在一个现有的函数作用域中存储数据，无需为每个对象创建新的作用域。

再举一个例子，论证访问函数之外的变量的必要性。

```
1  function fetchData(userId) {
2      const url = `https://api.example.com/user/${userId}`;
3      const requestTime = Date.now();
4
5      fetch(url)
6          .then(response => response.json())
7          .then(data => {
8              const responseTime = Date.now();
9              console.log(`Data for user ${userId}:`, data);
10             console.log(`Request sent at: ${requestTime}`);
11             console.log(`Response received at: ${responseTime}`);
12         });
13 }
14
15 fetchData(123);
```

Java Script 是一个non-blocking的语言。所以fetch.then().then()是一下子运行完的。

fetch的任务被扔到了，event loop,

然后.then(response => response.json())也被扔到event loop.

.then(data => {}) 也被扔到了event loop.

当前程序就运行完了。那么timestamp变量就会被销毁。

从第一个counter的例子中，closure返回的函数是会被call 才能真正执行。而这个例子是没有counter()的过程。原因是这里是Promise对象，promise对象会帮我们call

例子`let res = fetchData(123)`，这里`res`实际上是`fetchData`函数返回的Promise对象，而不是`then`方法里的回调函数。当你尝试调用`res(data)`时，JavaScript会报错，因为`res`是一个Promise对象，不是一个函数

总结，很难理解。closure的结论就是：

- 函数嵌套
- 内部函数引用了外部函数的变量。

1.9 变量提升（hoisting）

允许先使用变量，再declare变量。

> *注意：实际开发中推荐使用 let 关键字，并且需要先声明再访问变量。难道，你不觉得变量提升反人类吗?*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>

    <script>
        console.log(n); // undefined
```

```
        console.log(m); // undefined
        var n = 1;
        var m = 2;
        if (true) {
            n++;
        }
    </script>
</body>
</html>
```

2.2.1 参数默认值

```
// 许大仙 就是 name 的参数默认值
function fn(name='许大仙'){

}
```

- 如果参数没有自定义默认值，那么参数的默认值就是 undefined 。
- 调用函数的时候如果没有传入实参，当声明函数的时候，给形参设置了默认值，那么参数的默认值就会当做实参传入。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0"
name="viewport">
    <title>Title</title>
</head>
<body>

    <script>

```

```
        /* 参数默认值 */
        function fn(name = '呵呵哒', age = 19) {
            console.log(`我的名字是：${name}，年龄是：${age}`);
        }

        fn();
    </script>

</body>
</html>
```

2.2.2 动态参数

- `arguments` 是函数内部内置的伪数组变量，它包含了调用函数时传入的所有实参。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>

    <script>
        /* 动态参数：`arguments` 是函数内部内置的伪数组变量，它包含了调用函数时传入的所有实参。 */

        function fn() {
            console.log(arguments);
        }

```

```
        fn(1, 2, 3);
    </script>

</body>
</html>
```

2.2.3 剩余参数（rest 参数）

```
// ... args 表示剩余参数
function fn(...args){

}
```

> *意：*
>
> - *剩余参数在 Java 中被称为可变参数。*
> - *剩余参数必须置于函数形参列表的最后面。*
> - *剩余参数本质上就是数组，所以可以使用数组中的方法。*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>

    <script>
        /*
            剩余参数：声明在函数的形参列表中的最后位置，表示多个参数。
```

```
            剩余参数在 Java 中被称为可变参数。
            剩余参数本质上就是数组。
        */
        function fn(...arr) {
            // [1, 2]
            console.log(arr);
            arr.forEach(n => {
                console.log(n);
            });
        }

        fn(1, 2, 3,4,a,b);
    </script>

</body>
</html>
```

Question： 同样是..., 在哪里还见过么？

Spread

https://www.yuque.com/fairy-era/xurq2q/pddl6r#8f31146f

2.3 箭头函数

普通函数

```
let fn = function(){}
```

箭头函数

```
// 可以将箭头函数等价于普通的函数表达式理解
let fn = () => {}
```

> *注意：*
>
> - *箭头函数被称为 Lambda 表达式，在 Java 中是使用 `->`，在 JavaScript 中使用 `=>`。*
> - *因为 Java 是纯面向对象的语言，所以箭头函数更多的是被当做回调函数来处理的，而 JavaScript 中函数是一等公民，箭头函数既可以独立使用，也可以被当做回调函数来处理。*

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>

    <script>
        // let fn = function(){}

        // 箭头函数
        let fn = (a, b) => {
            return a + b;
        };

        let res = fn(1, 2);
        console.log(res);
    </script>

</body>
</html>
```

解构赋值

把数组中的值，赋给多个变量。类似于Java写法。注意第二种写法提供的语法糖，简化很多

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>

    <script>
        /* 解构赋值是一种快速为变量赋值的简洁语法，本质上仍然是为变量赋值，分为数组解构、对象解构两大类型。 */
        let arr = ['张飞', '刘备', '孙权', '关羽'];
        let uname1 = arr[0];
        let uname2 = arr[1];
        let uname3 = arr[2];
        let uname4 = arr[3];
        console.log(uname1, uname2, uname3, uname4);

        /* 解构赋值：解开数据的结构赋值给变量。 */
        let [u1, u2, u3, u4] = arr;
        console.log(u1, u2, u3, u4);

    </script>

</body>
</html>
```

Promise

https://www.yuque.com/fairy-era/xurq2q/axnrw1#a017ad71

我觉得讲的比较乱，需要很有耐心看几遍。多尝试理解例子。

- Promise 的理解
    - 抽象表达：Promise 是 JavaScript 进行异步编程的新方案（旧方案是纯回调函数，纯回调函数会产生回调地狱的问题）。
    - 具体表达：
        - 从语法上讲，Promise 是一个构造函数。
        - 从功能上讲，Promise 对象是用来封装一个异步操作并可以获取其结果。
- Promise 的状态
    - ① pending（初始化状态） 变为 fulfilled（成功状态）。
    - ② pending（初始化状态） 变为 rejected（失败状态）。
    - > *注意：*
      >
      > - *Promise 只有这两种状态，且一个 Promise 对象只能被改变一次。*
      > - *无论变为成功或失败，都会有一个结果数据。*
      > - *成功的结果数据一般称为 value ，失败的结果数据一般称为 reason （行业规范）。*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>
    <script>
```

```
        /*
        * 改变 Promise 实例的状态和指定回调函数谁先谁后?
        * ① 都有可能，正常情况下是先指定回调再改变状态，但是也可以先改变状态再指定回调。
        * ② 如何先改状态再指定回调? 只需要延迟一会再调用 then() 方法。
        * ③ Promise 实例什么时候能得到数据?
        *    如果先指定的回调，那么当状态发生改变的时候，回调函数就会调用，得到数据。
        *    如果先改变的状态，那么当指定回调的时候，回调函数就会调用，得到数据。
        */
        // 正常情况下是先指定回调再改变状态
        const p = new Promise((resolve, reject) => {
            // 函数体
            setTimeout(() => {
                // case1: resolve
                // case2: reject
                // case3: resolve + reject
                // case4: reject + resolve

                resolve(11);
                reject("培训太久了，疲惫了，not feel well");
            }, 1000);
        });
        p.then(value => {
            // 函数体
            console.log('成功了', value);
        }, reason => {
            // 函数体
            console.log('失败了', reason);
        });
    </script>
</body>
</html>

```

2.4.3 then 的链式调用

Then 中两个参数，一个resolve，一个是reject。 AKA,

```
promise.then(onFulfilled[, onRejected]);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>
    <script>
        /*
        *『问』Promise实例.then() 返回的是一个 `新 Promise 实例` ，它和值和状态由什么决定?
        *『答』:
        *   简单表达：由 then() 所指定的回调函数执行的结果决定的。
        *   详细表达:
        *    如果 then 所指定的回调返回的是 `非Promise 值 a` ，那么`新 Promise 实例`状态为成功（fulfilled），成功的 value 为 a。
        *    如果 then 所指定的回调返回的是一个 `Promise 实例 p` ，那么`新 Promise 实例`的状态和值都和 p 一致。
        *    如果 then 所指定的回调`抛出异常`，那么`新Promise实例`状态为 rejected ，reason 为抛出的异常。
        */
        const p = new Promise((resolve, reject) => {
            // 函数体
            setTimeout(() => {
```

```
                resolve('a');
            }, 1000);
        });
        const x = p.then(value => {
            // 函数体
            console.log('成功1', value); // 成功1 a
        }, reason => {
            // 函数体
            console.log('失败1', reason);
        });

        x.then(value => {
            // 函数体
            console.log('成功2', value); // 成功2 undefined
        }, reason => {
            // 函数体
            console.log('失败2', reason);
        });

    </script>
</body>
</html>
```

写成chain如下

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0"
name="viewport">
    <title>Title</title>
</head>
<body>
```

```
    <script>
        /*
        *『问』Promise实例.then() 返回的是一个 `新 Promise 实例` ，它和值和状态由什么决定?
        *『答』：
        * ① 简单表达：由 then() 所指定的回调函数执行的结果决定的。
        * ② 详细表达：
        *    如果 then 所指定的回调返回的是 `非Promise 值 a` ，那么`新 Promise 实例`状态为成功（fulfilled），成功的 value 为 a。
        *    如果 then 所指定的回调返回的是一个 `Promise 实例 p` ，那么`新 Promise 实例`的状态和值都和 p 一致。
        *    如果 then 所指定的回调`抛出异常`，那么`新Promise实例`状态为 rejected ，reason 为抛出的异常。
        */
        const p = new Promise((resolve, reject) => {
            // 函数体
            setTimeout(() => {
                resolve('a');
            }, 1000);
        });

        // 链式调用
        p.then(value => {
            // 函数体
            console.log('成功1', value); // 成功1 a
        }, reason => {
            // 函数体
            console.log('失败1', reason);
        }).then(value => {
            // 函数体
            console.log('成功2', value); // 成功2 undefined
        }, reason => {
            // 函数体
            console.log('失败2', reason);
        });

```

```
    </script>
</body>
</html>
```

2.4.7 Promise 异常穿透

- 当使用 Promise 实例的 then 链式调用时，可以在最后使用 catch 捕获一个事变的回调。当前面任何操作出现了问题，都会传递到最后失败的回调中进行处理。

```
p.then().then().then().catch()
```

这样的话，t hen里就不需要写两个箭头函数了（resolve, reject）,只需要写一个（resolve）， reject统一交给catch来处理。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>
    <script>
        // 异常穿透：当使用 Promise 实例的 then 链式调用时，可以在最后使用 catch 捕获一个事变的回调。当前面任何操作出现了问题，都会传递到最后失败的回调中进行处理。
        const p = new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve('用户数据');
            }, 1000);
        });
        p.then(value => {
```

```
            console.log('第一次请求成功', value);
            return new Promise((resolve, reject) => {
                setTimeout(() => {
                    reject('订单数据');
                }, 1000);
            });
        }).then(value => {
            console.log('第二次请求成功', value);
            return new Promise((resolve, reject) => {
                setTimeout(() => {
                    resolve('商品数据');
                }, 1000);
            });
        }).then(value => {
            console.log('第三次请求成功', value);
        }).catch(reason => {
            console.error(reason);
        });
    </script>
</body>
</html>
```

注意t hen里return的是Promise对象，正如Java中的completableFuture，中间对象都是CompletableFuture.
Java的stream，中间对象也都是Stream对象。

对比一下Java中的completableFuture

Promise then语法结构

- resolve and reject
    - `promise.then(onFulfilled[, onRejected]);`

- 如果你只对 Promise 的成功结果感兴趣，你可以只提供 `onFulfilled` 函数。这种情况下，你可能会使用 `catch()` 方法来处理任何可能的错误
    - `promise.then(onFulfilled).catch(onRejected);`
- 如果你只对 Promise 的失败结果感兴趣
    - `promise.then(null, onRejected);`
    - `promise.catch(onRejected);`

总的来说，`then()` 方法可以接受两个参数，但都不是必须的。你可以根据你的需求来决定是否提供这些参数。

Promise Vs CompletableFuture

例子，获取数据 -> 处理数据 -> 保存数据 -> log数据 -> handle error

JavaScript, Promise版本

```
function fetchData() {
    return fetch('https://api.example.com/data');
}

function processData(response) {
    return response.json();
}

function saveData(data) {
    // 这只是一个例子，假设我们有一个函数可以保存数据。
    return saveToDatabase(data);
}

function logData(data) {
    console.log(data);
    return data;
}
```

```
18
19 fetchData()
20     .then(processData)
21     .then(saveData)
22     .then(logData)
23     .catch(error => {
24         console.error('An error occurred:', error);
25     });
26
```

Java, CompletableFuture版本

```
1  import java.net.http.*;
2  import java.net.URI;
3  import java.util.concurrent.*;
4
5  public class Main {
6      private static final HttpClient httpClient =
   HttpClient.newHttpClient();
7
8      public static CompletableFuture<String> fetchData() {
9          HttpRequest request = HttpRequest.newBuilder()
10             .uri(URI.create("https://api.example.com/data"))
11             .build();
12
13         return httpClient.sendAsync(request,
   HttpResponse.BodyHandlers.ofString())
14             .thenApply(HttpResponse::body);
15     }
16
17     public static CompletableFuture<JsonNode> processData(String
   data) {
18         // 这只是一个例子，你需要用你选择的JSON处理库来处理JSON数据。
19         JsonNode jsonData = parseJson(data);
20         return CompletableFuture.completedFuture(jsonData);
```

```
    }

    public static CompletableFuture<Void> saveData(JsonNode data)
{
        // 这只是一个例子，假设我们有一个方法可以保存数据。
        return saveToDatabase(data);
    }

    public static void logData(JsonNode data) {
        System.out.println(data);
    }

    public static void main(String[] args) {
        fetchData()
            .thenCompose(Main::processData)
            .thenCompose(Main::saveData)
            .thenAccept(Main::logData)
            .exceptionally(error -> {
                System.err.println("An error occurred: " + error);
                return null;
            });
    }
}

```

Callback hell

Question，下方代码是在做什么事情

```
fetchData((error, data) => {
```

```
  if (error) {
    console.error(error);
  } else {
    processData(data, (error, processedData) => {
      if (error) {
        console.error(error);
      } else {
        saveData(processedData, (error, result) => {
          if (error) {
            console.error(error);
          } else {
            console.log(result); // Output: Data Processed Data fetched saved
          }
        });
      }
    });
  }
});
```

vs

```
fetchData()
  .then((data) => processData(data))
  .then((processedData) => saveData(processedData))
  .then((result) => console.log(result)) // Output: Data Processed Data fetched saved
  .catch((error) => console.error(error));
```

写法简单了很多

逻辑看起来也清晰了很多。

Async & Await

跟promise一样，都是为了让写异步代码的时候，像写同步代码一样。方便人的理解。
比如上面的.then().then().

.then().then().then() 这种类似于stream的chain,学会了用起来很爽。但是学习和适应成本有点高。
Async, Await是更接近普通人编码习惯的。
React框架中也更prefer async, await.

其实他俩还互补.比如不想干的三个异步的事情，就无法.then().then()了。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>Title</title>
</head>
<body>
    <script>
        const p1 = new Promise((resolve, reject) => {
            // 函数体
            setTimeout(() => {
                resolve(1);
            }, 1000);
        });

        const p2 = new Promise((resolve, reject) => {
            // 函数体
            setTimeout(() => {
                resolve(2);
            }, 2000);
        });
```

```
24
25          const p3 = new Promise((resolve, reject) => {
26              // 函数体
27              setTimeout(() => {
28                  resolve(3);
29              }, 3000);
30          });
31
32          // 可以使用立即执行函数简化，这种方式和 Promise 实例对象的 then 方法执行的结果是一样的。
33           (async () => {
34              try {
35                  const result1 = await p1;
36                  console.log(result1);
37                  const result2 = await p2;
38                  console.log(result2);
39                  const result3 = await p3;
40                  console.log(result3);
41              } catch (e) {
42                  console.error(e);
43              }
44          })();
45
46          console.log("done");
47      </script>
48  </body>
49  </html>
50
```

3.3 async 和 await 的规则

- async 修饰的函数：
    - 函数的返回值是 Promise 对象
    - Promise 实例的结果由 async 函数执行的返回值决定

- await 表达式：
    - await 右侧的表达式一般为 Promise 实例对象，但是也可以是其他的值
    - 如果表达式是 Promise 对象，await 的返回值是 Promise 成功的值
    - 如果表达式是其他值，直接将此值作为 await 的返回值

> *注意：*
>
> - *await 必须写在 async 函数中，但是 async 函数中可以没有 await （ES7 规定的）。*
> - *如果 await 的 Promise 实例失败了，就会抛出异常，需要通过 try...catch 进行捕获。*

Axios

https://www.yuque.com/fairy-era/xurq2q/spe3t5

> *Axios 是一个基于 promise 网络请求库*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta content="IE=edge" http-equiv="X-UA-Compatible">
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <title>完整版 GET 请求携带参数</title>
    <script src="./js/axios.min.js"></script>
</head>
<body>

    <button id="btn">获取人员信息</button>

    <script>
        // 获取按钮
```

```
        const btn = document.querySelector('#btn');
        // 点击
        btn.addEventListener('click', function () {
            /*
            * ① axios 调用的返回值是 Promise 实例。
            * ② 成功的值叫做 response ，失败的值叫做 error 。
            * ③ axios 成功的值是一个 axios 封装的 response 对象，服务器返回的真正数据在 response 对象的 data 属性中。
            */
            axios({
                // 请求方式
                method: 'GET',
                url: 'http://localhost:5000/person',
                params: {
                    id: '1',
                }
            }).then(response => {
                console.log(response);
                const {data} = response;
                console.log(data);
            }).catch(error => {
                console.error(error);
            });

        });
    </script>
</body>
</html>
```

fetch vs axios

`fetch`和`axios`都是在浏览器中进行HTTP请求的常见库。以下是它们的一些关键差异：

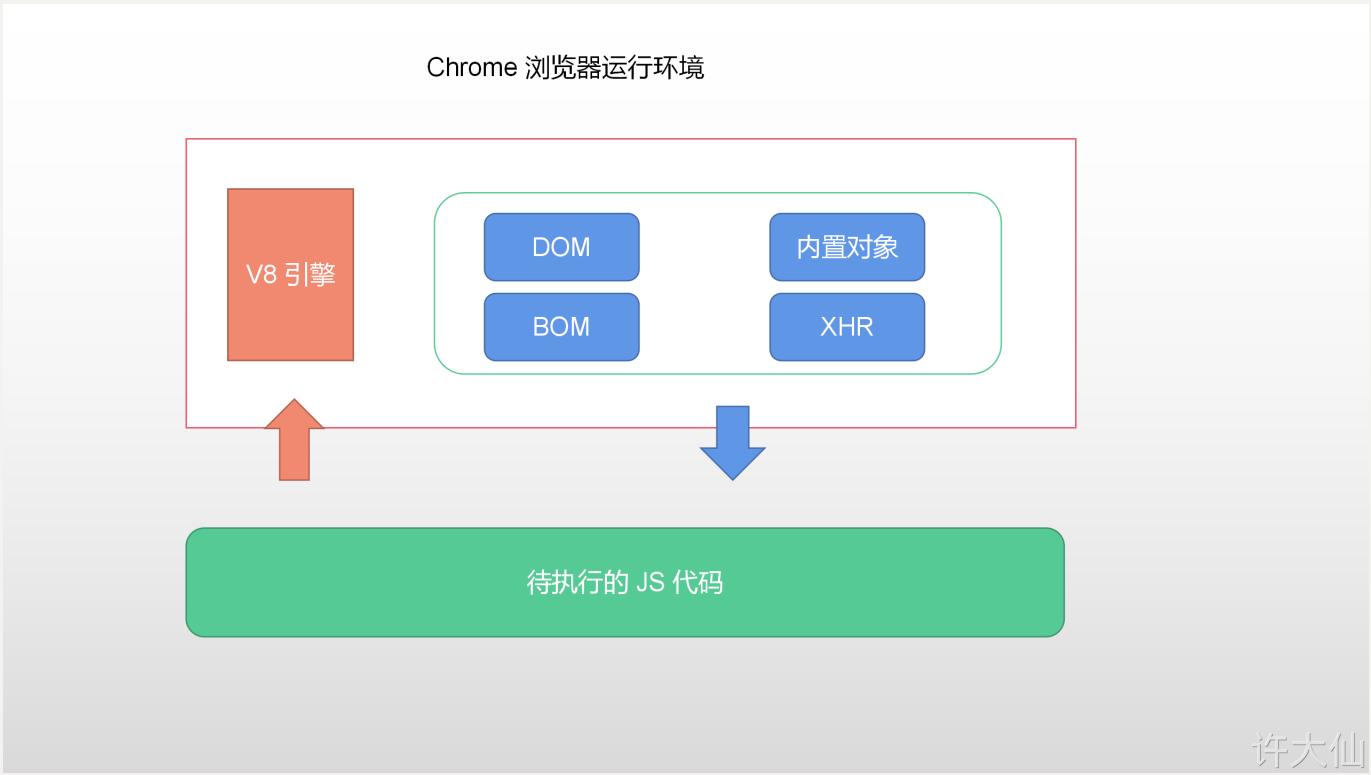
1. **默认错误处理**：`fetch`在默认情况下不会将HTTP错误状态（如404或500）视为错误，而只有在网络错误时才会reject。这意味着你必须手动检查`response.ok`。相比之下，`axios`会在HTTP错误状态下自动reject，这可能使错误处理更加直接。
2. **浏览器兼容性**：`fetch`是较新的API，可能在一些旧的或者不常用的浏览器中无法使用，可能需要使用polyfill（例如`isomorphic-fetch`或`fetch-everywhere`）来进行兼容性处理。相反，`axios`的兼容性更好，可以在更多的环境中使用。
3. **超时控制**：`fetch` API并没有提供请求超时的功能，因此如果一个请求需要在某个特定的时间内完成，这就需要使用其他的方法来实现。相比之下，`axios`允许你设置`timeout`属性来控制请求的超时时间。
4. **请求取消**：`fetch` API没有内置的请求取消功能，但可以通过`AbortController`接口来实现。`axios`提供了一种比较简单的取消请求的方式。
5. **自动转换 JSON**：`axios`会自动将请求和响应数据转换为 JSON，而`fetch`需要手动调用`response.json()`方法来进行转换。
6. **防御XSRF**：`axios`具有一些内置的防御XSRF的功能，这对于某些应用来说可能很有用。

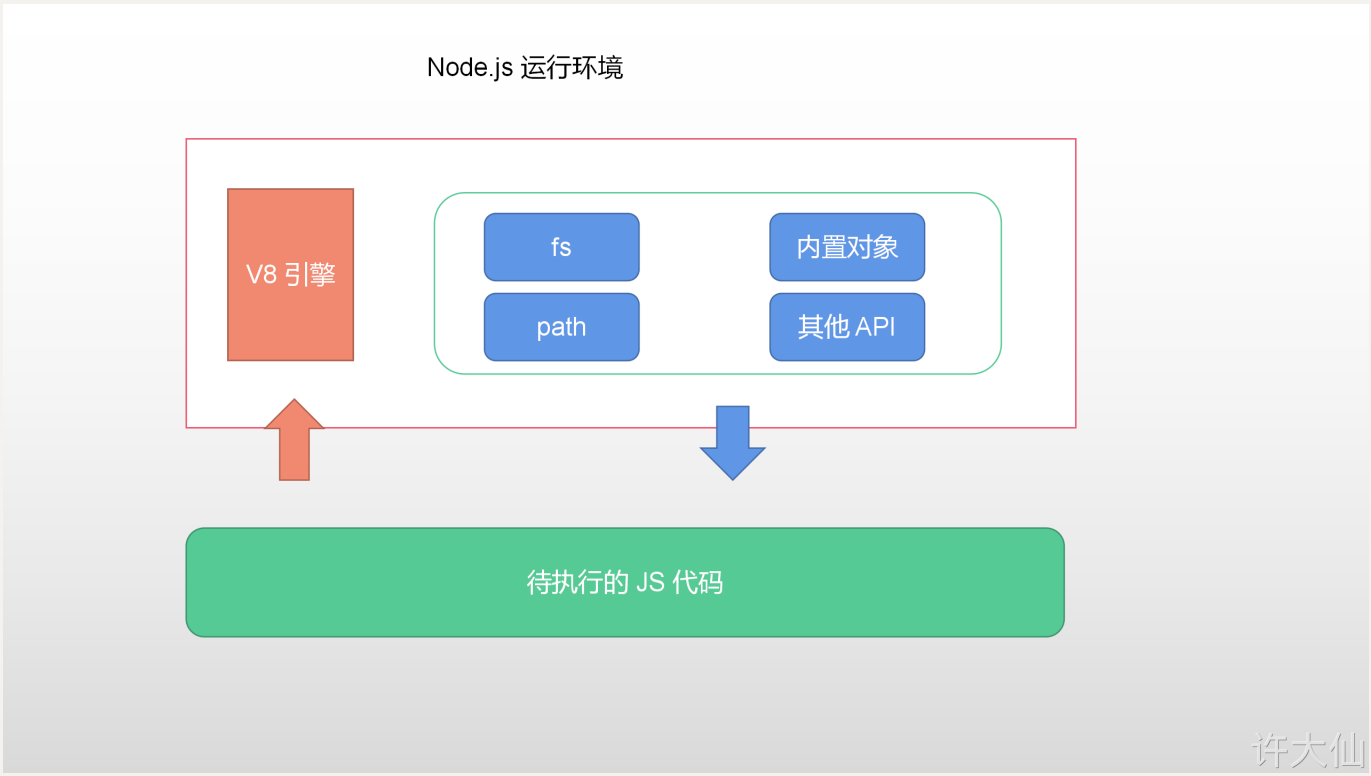
两者都是优秀的库，选择哪一个主要取决于你的项目需求。如果你需要更多的功能，例如请求取消、超时和自动的 JSON 转换，那么`axios`可能是一个更好的选择。如果你想要一个更轻量级或者更"原生"的解决方案，那么`fetch`可能是一个更好的选择。

Node

https://www.yuque.com/fairy-era/xurq2q/nzkhuc

1.3 什么是 Node.js ?





javascript是操作浏览器的。如第一张图有dom，bom

Node.js提供了操作文件fs， path以及其它API，此时Java script就可以脱离浏览器了。也就可以做后端了

两者都是基于V8引擎

Node的Express框架，可以用于开发we b应用。可以粗略的认为类似springbok。

TypeScript

https://www.yuque.com/fairy-era/xurq2q/wvg7zy

JavaScript是弱数据类型语言。所以容易有bug。

> *TypeScript = `Type` + JavaScript（在 JS 基础上，为 JS 添加了`类型支持`）*

1.2 TypeScript 为什么要为 JavaScript 添加类型支持?

- 背景：JS 的类型系统存在`"先天缺陷"`，JS 代码中绝大部分错误是`类型`错误（UncaughtTypeError）。
- 问题：增加了找 Bug，改 Bug 的时间，严重影响了开发效率。

- 从编程语言的动静来区分，TypeScript 是属于静态类型的编程语言，JavaScript 是属于动态类型的编程语言。
- 静态类型的编程语言是在`编译期`做类型检查；动态类型的编程语言是在`执行期`做类型检查。
- 静态语言需要先编译再执行。

- 对于 JavaScript 来说：需要等到代码真正去`执行`的时候才能`发现错误`（晚）。
- 对于 TypeScript 来说：在代码`编译`的时候（代码执行前）就可以`发现错误`（早）。

this

https://www.yuque.com/fairy-era/xurq2q/htzfof#ef030a82

JQuery

jQuery 是一款非常流行的 JavaScript 库，主要用于简化 HTML 文档遍历、事件处理、动画设计和 Ajax 交互等任务。它的目标是 "write less, do more"，也就是说，开发者可以通过较少的代码完成更多的功能。

特性如下：

1. **简洁明了的语法**：可以很方便地选择 DOM 元素、绑定事件、创建动画以及处理和发起 Ajax 请求等。
2. **跨浏览器兼容性**：它能够兼容所有主流的浏览器，如 Chrome、Firefox、IE、Safari 等。
3. **强大的选择器**：jQuery 继承了强大的 CSS3 选择器，可以方便地选取 DOM 元素。
4. **链式操作**：通过jQuery的链式操作，你可以在同一个元素上执行许多操作，而无需多次单独选取元素。
5. **丰富的 Ajax**：它提供了丰富的 Ajax 功能，可以更方便地进行异步请求。
6. **DOM 操作接口**：插入、删除、移动、复制、创建等操作。
7. **可扩展**：jQuery 允许开发者通过插件形式对其进行扩展，可以更好地满足特定需求。

然而，随着现代浏览器对于 JavaScript 标准的支持越来越好，以及 Vue、React、Angular 等现代 JavaScript 框架的出现，jQuery 的使用已经不如以前普遍。许多功能现在可以使用原生 JavaScript 或者现代框架实现，而无需依赖 jQuery。

```
// 确保文档已经完全加载，然后执行函数
$(document).ready(function() {

  // 选择 id 为 'myButton' 的元素，并给它绑定一个点击事件
  $('#myButton').click(function() {
    // 当按钮被点击时，选择 id 为 'myDiv' 的元素，并修改其文本内容
    $('#myDiv').text('Hello, jQuery!');
  });

});

```

Ajax

Call 后端API，达到前后端交互的效果。