HTML 分类

1. script 标签中的 defer 和 async

- **原理**：`defer` 和 `async` 是 `<script>` 标签用于控制脚本加载与执行顺序的属性。`defer` 使脚本在文档解析完成后、`DOMContentLoaded` 事件触发前按顺序执行；`async` 让脚本异步加载，加载完成后立即执行，不保证执行顺序。
- **代码案例**：

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <!-- defer 属性 -->
    <script defer src="defer.js"></script>
    <!-- async 属性 -->
    <script async src="async.js"></script>
</head>

<body>
    <h1>Hello, World!</h1>
</body>

</html>
```

- **案例应用**：若脚本有依赖关系且需在文档解析完后执行，用 `defer`；若脚本无依赖且不影响页面渲染，用 `async`。例如在加载第三方统计脚本时可用 `async`，加载依赖于文档结构的脚本时用 `defer`。

2. 解析 html 标签时哪些是异步的？哪些是同步的?

- **原理**：HTML 解析时，`<script>` 标签默认同步加载执行，会阻塞文档解析。加 `defer` 或 `async` 属性的 `<script>` 异步。`<link rel="stylesheet">` 同步加载，不阻塞文档解析，但阻塞 `DOMContentLoaded` 事件。`<img>` 异步加载，不影响文档解析。
- **代码案例**：

```
<!DOCTYPE html>
```

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <!-- 同步加载样式 -->
    <link rel="stylesheet" href="styles.css">
    <!-- 默认同步脚本 -->
    <script src="sync.js"></script>
    <!-- 异步脚本 -->
    <script async src="async.js"></script>
</head>

<body>
    <!-- 异步加载图片 -->
    <img src="example.jpg" alt="Example">
</body>

</html>
```

- **案例应用**：合理安排标签的同步异步加载，可优化页面性能。如将不影响页面渲染的脚本设为异步，避免阻塞文档解析；将样式文件放在头部同步加载，确保页面样式正常显示。

CSS 分类

1. 了解重排和重绘吗?

- **原理**：重排（回流）是指当 DOM 的变化影响了元素的布局信息（元素的宽高、边距、位置等），浏览器需要重新计算元素在视口内的位置和大小。重绘是指当一个元素的外观发生改变，但没有影响到布局信息时，浏览器将新样式绘制到屏幕上。重排的代价比重绘大，因为重排可能会触发后续一系列元素的重排和重绘。
- **代码案例**：

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
```

```
    <style>
        #box {
            width: 100px;
            height: 100px;
            background-color: red;
        }
    </style>
</head>

<body>
    <div id="box"></div>
    <button onclick="changeWidth()">Change Width</button>
    <button onclick="changeColor()">Change Color</button>
    <script>
        function changeWidth() {
            const box = document.getElementById('box');
            box.style.width = '200px'; // 触发重排和重绘
        }

        function changeColor() {
            const box = document.getElementById('box');
            box.style.backgroundColor = 'blue'; // 触发重绘
        }
    </script>
</body>

</html>
```

- **案例应用**：在开发中，应尽量减少重排和重绘的次数。比如批量修改 DOM 样式时，可先将元素从文档流中移除，修改完后再添加回去；避免频繁读取会触发重排的属性，如 `offsetWidth`、`scrollTop` 等。

2. CSS 和 JS 动画会导致重排吗?

- **原理**：CSS 动画和 JS 动画是否导致重排取决于动画改变的属性。如果改变的是影响布局信息的属性（如宽度、高度、边距等），则会触发重排；如果改变的是只影响外观的属性（如颜色、透明度等），则只会触发重绘。一般来说，CSS 的 `transform` 和 `opacity` 属性在动画中不会触发重排，性能较好。
- **代码案例**：

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

<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <style>
        #box {
            width: 100px;
            height: 100px;
            background-color: red;
            transition: all 1s;
        }

        .move {
            transform: translateX(100px); /* 不会触发重排 */
        }

        .resize {
            width: 200px; /* 会触发重排 */
        }
    </style>
</head>

<body>
    <div id="box"></div>
    <button onclick="moveBox()">Move Box</button>
    <button onclick="resizeBox()">Resize Box</button>
    <script>
        function moveBox() {
            const box = document.getElementById('box');
            box.classList.add('move');
        }

        function resizeBox() {
            const box = document.getElementById('box');
            box.classList.add('resize');
        }
    </script>
</body>

</html>
```

- **案例应用**：在实现动画效果时，优先使用 `transform` 和 `opacity` 属性来创建动画，以提高性能。例如制作元素的移动、旋转、缩放等动画时，使用 `transform` 属性；制作淡入淡出效果时，使用 `opacity` 属性。

JavaScript 分类

1. event loop

- **原理**：Event Loop（事件循环）是 JavaScript 的执行机制，用于处理异步操作。JavaScript 是单线程的，为了处理异步任务（如定时器、网络请求等），引入了事件循环机制。它的核心包括调用栈（Call Stack）、任务队列（Task Queue，分为宏任务队列和微任务队列）。当调用栈中的同步任务执行完后，事件循环会不断从任务队列中取出任务放入调用栈执行。
- **代码案例**：

```
console.log('Start');

setTimeout(() => {
    console.log('Timeout');
}, 0);

Promise.resolve().then(() => {
    console.log('Promise');
});

console.log('End');
```

- **代码细节**：首先，`console.log('Start')` 是同步任务，直接在调用栈执行并输出。`setTimeout` 是宏任务，会在 0 毫秒后将回调函数放入宏任务队列。`Promise.resolve().then()` 是微任务，会将回调函数放入微任务队列。`console.log('End')` 也是同步任务，接着执行。当调用栈为空时，事件循环会先处理微任务队列中的任务，所以输出 `Promise`，然后处理宏任务队列中的任务，输出 `Timeout`。
- **案例应用**：在处理异步操作时，如网络请求、定时器等，事件循环机制确保了代码的执行顺序和异步任务的处理。例如在前端开发中，使用 `fetch` 进行网络请求，请求完成后的回调函数会通过事件循环机制在合适的时机执行。

2. 事件循环代码输出

```
async function async1() {
    console.log('async1 start');
    await async2();
    console.log('async1 end');
}

async function async2() {
    console.log('async2');
}
```

```
console.log('script start');

setTimeout(() => {
    console.log('setTimeout');
}, 0);

async1();

new Promise((resolve) => {
    console.log('promise1');
    resolve();
}).then(() => {
    console.log('promise2');
});

console.log('script end');
```

- **原理**：同样基于事件循环机制，分析代码中同步任务、微任务和宏任务的执行顺序。`async/await` 本质上是基于 `Promise` 的语法糖，`await` 会暂停当前函数的执行，等待后面的 `Promise` 完成。
- **代码细节**：
  i. 首先执行同步任务，输出 `script start`。
  ii. `setTimeout` 是宏任务，将回调函数放入宏任务队列。
  iii. 调用 `async1` 函数，输出 `async1 start`，接着调用 `async2` 函数，输出 `async2`。`await async2()` 暂停 `async1` 函数的执行。
  iv. 执行 `new Promise`，输出 `promise1`，`resolve()` 后将 `then` 回调放入微任务队列。
  v. 输出 `script end`，此时调用栈为空。
  vi. 处理微任务队列，先执行 `async1` 函数中 `await` 后面的代码，输出 `async1 end`，再执行 `Promise` 的 `then` 回调，输出 `promise2`。
  vii. 处理宏任务队列，输出 `setTimeout`。
- **案例应用**：理解事件循环代码输出顺序有助于调试和优化涉及异步操作的代码，避免出现意外的执行顺序问题。

3. usestate 和 useref 区别?

- **原理**：`useState` 和 `useRef` 是 React Hooks 中的两个钩子。`useState` 用于在函数组件中添加状态，当状态更新时，组件会重新渲染。`useRef` 创建一个可变的对象，它的值在组件的整个生命周期内保持不变，并且修改它不会触发组件重新渲染。

- **代码案例**：

```
import React, { useState, useRef } from 'react';

function Example() {
    const [count, setCount] = useState(0);
    const refCount = useRef(0);

    const incrementState = () => {
        setCount(count + 1);
    };

    const incrementRef = () => {
        refCount.current++;
        console.log('Ref count:', refCount.current);
    };

    return (
        <div>
            <p>State count: {count}</p>
            <button onClick={incrementState}>Increment State</button>
            <button onClick={incrementRef}>Increment Ref</button>
        </div>
    );
}

export default Example;
```

- **代码细节**：`useState` 返回一个数组，第一个元素是状态值，第二个元素是更新状态的函数。每次调用 `setCount` 时，组件会重新渲染。`useRef` 返回一个对象，通过 `current` 属性访问和修改其值。
- **案例应用**：`useState` 适用于需要触发组件重新渲染的状态管理，如表单输入值、计数器等。`useRef` 适用于保存不需要触发重新渲染的值，如 DOM 节点引用、定时器 ID 等。

4. 平时是怎么发异步请求的

- **原理**：在前端开发中，常见的发送异步请求的方式有 XMLHttpRequest、`fetch` API 和第三方库如 `axios`。这些方式都是基于浏览器的网络请求能力，通过异步操作获取服务器数据，避免阻塞主线程。
- **代码案例（使用 `fetch`）**：

```
fetch('https://api.example.com/data')
```

```
  .then(response => {
        if (!response.ok) {
            throw new Error('Network response was not ok');
        }
        return response.json();
    })
  .then(data => {
        console.log('Data:', data);
    })
  .catch(error => {
        console.error('Error:', error);
    });
```

- **代码细节**：`fetch` 函数返回一个 `Promise`，它接收一个 URL 作为参数。第一个 `then` 方法处理响应对象，检查响应状态是否正常，若正常则将响应数据解析为 JSON。第二个 `then` 方法处理解析后的数据。`catch` 方法捕获请求过程中出现的错误。
- **案例应用**：在开发中，当需要从服务器获取数据时，如获取用户信息、商品列表等，可使用上述方法发送异步请求。

5. 事件循环你是怎么用的

- **原理**：事件循环主要用于处理异步任务，合理利用事件循环机制可以优化代码性能和实现复杂的异步逻辑。例如，将一些耗时的操作放在异步任务中，避免阻塞主线程。
- **代码案例**：

```
function longRunningTask() {
    let sum = 0;
    for (let i = 0; i < 1000000; i++) {
        sum += i;
    }
    return sum;
}

// 使用 setTimeout 将耗时任务放入宏任务队列
setTimeout(() => {
    const result = longRunningTask();
    console.log('Task result:', result);
}, 0);

console.log('Main thread continues...');
```

- **代码细节**：`longRunningTask` 是一个耗时的计算任务。使用 `setTimeout` 将其放

入宏任务队列，主线程可以继续执行后续代码，输出 `Main thread continues...`。当调用栈为空时，事件循环会从宏任务队列中取出 `longRunningTask` 并执行。

- **案例应用**：在前端开发中，对于一些复杂的计算、大数据处理等耗时操作，可利用事件循环将其异步化，避免页面卡顿。

6. 虚拟 DOM 的理解

- **原理**：虚拟 DOM（Virtual DOM）是一种轻量级的 JavaScript 对象，它是真实 DOM 的抽象表示。当组件状态发生变化时，React 等框架会先计算虚拟 DOM 的差异（Diffing 算法），然后将差异批量更新到真实 DOM 上，这样可以减少直接操作真实 DOM 的次数，提高性能。
- **代码案例（简单模拟虚拟 DOM 更新）**：

```
// 虚拟 DOM 节点类
class VNode {
    constructor(tag, props, children) {
        this.tag = tag;
        this.props = props;
        this.children = children;
    }
}

// 创建虚拟 DOM
const oldVNode = new VNode('div', { id: 'old' }, ['Old content']);
const newVNode = new VNode('div', { id: 'new' }, ['New content']);

// 简单的 Diff 算法和更新函数
function diff(oldVNode, newVNode) {
    if (oldVNode.tag!== newVNode.tag || oldVNode.props.id!==
newVNode.props.id) {
        // 这里简单模拟更新操作
        console.log('Update DOM');
    }
}

diff(oldVNode, newVNode);
```

- **代码细节**：定义了一个 `VNode` 类来表示虚拟 DOM 节点。创建了旧的和新的虚拟 DOM 节点，通过 `diff` 函数比较它们的差异，若有差异则进行更新操作。
- **案例应用**：在 React、Vue 等前端框架中广泛使用虚拟 DOM 来优化 DOM 操作。例如在一个列表组件中，当列表数据更新时，框架会通过虚拟 DOM 计算差异，只更新需要更新的部分，而不是重新渲染整个列表。

7. 你提到了 Vue3，说说它有哪些改进

- **原理**：Vue3 在性能、开发体验、代码组织等方面进行了诸多改进。采用了 Proxy 实现响应式系统，提高了响应式的性能和功能；引入了组合式 API，使代码更易于复用和维护；重构了虚拟 DOM 实现，优化了渲染性能。
- **代码案例（组合式 API 示例）**：

```
<template>
    <div>
        <p>{{ count }}</p>
        <button @click="increment">Increment</button>
    </div>
</template>

<script setup>
import { ref } from 'vue';

const count = ref(0);

const increment = () => {
    count.value++;
};
</script>
```

- **代码细节**：使用 `script setup` 语法和组合式 API，`ref` 函数创建响应式数据 `count`，通过 `count.value` 访问和修改其值。点击按钮时调用 `increment` 函数更新 `count` 的值。
- **案例应用**：组合式 API 使得代码逻辑可以按照功能进行分组，提高了代码的可读性和可维护性。例如在大型项目中，不同的功能模块可以独立封装和复用。

8. vue 双向绑定原理

- **原理**：Vue 的双向绑定是通过数据劫持结合发布 - 订阅模式实现的。在 Vue2 中，使用 `Object.defineProperty()` 对数据对象的属性进行劫持，当属性值发生变化时，触发 `setter` 方法通知所有订阅者更新视图；在 Vue3 中，使用 `Proxy` 对象进行数据劫持。

- **代码案例（Vue2 简单实现）**：

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

<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Vue2 Two - way Binding</title>
</head>

<body>
    <input type="text" id="input">
    <p id="output"></p>
    <script>
        const data = {
            message: ''
        };

        // 数据劫持
        Object.defineProperty(data, 'message', {
            get() {
                return this._message;
            },
            set(newValue) {
                this._message = newValue;
                document.getElementById('output').textContent =
newValue;
            }
        });

        const input = document.getElementById('input');
        input.addEventListener('input', function () {
            data.message = this.value;
        });

        // 初始化输出
        document.getElementById('output').textContent = data.message;
    </script>
</body>

</html>
```

- **代码细节**：使用 `Object.defineProperty` 对 `data` 对象的 `message` 属性进行劫持，当 `message` 属性被赋值时，`setter` 方法会被触发，更新页面上的 `p` 元素内容。同时，给输入框添加 `input` 事件监听器，当输入框内容变化时，更新 `data.message` 的值，从而实现双向绑定。
- **案例应用**：在表单输入场景中，如用户登录、注册表单，通过双向绑定可以方便地将用户输入的数据同步到数据对象中，同时将数据对象的变化反映到页面上。

9. 在什么阶段进行 DOM 操作

- **原理**：在前端开发中，不同的框架和场景有不同的适合进行 DOM 操作的阶段。在原生 JavaScript 中，通常在 `DOMContentLoaded` 事件触发后进行 DOM 操作，确保文档已经解析完成。在 Vue 中，可以在 `mounted` 生命周期钩子中进行 DOM 操作，此时组件已经挂载到 DOM 上。在 React 中，可以使用 `useEffect` 钩子在组件渲染后进行 DOM 操作。
- **代码案例（Vue 示例）**：

```
<template>
    <div>
        <p ref="myParagraph">This is a paragraph.</p>
    </div>
</template>

<script>
export default {
    mounted() {
        const paragraph = this.$refs.myParagraph;
        paragraph.style.color ='red';
    }
};
</script>
```

- **代码细节**：在 Vue 组件的 `mounted` 钩子中，通过 `this.$refs` 获取到 `p` 元素的引用，然后修改其样式。
- **案例应用**：当需要对页面上的元素进行样式修改、添加事件监听器等操作时，选择合适的阶段进行 DOM 操作可以避免出现元素未加载完成的错误。

10. 防抖节流的作用

- **原理**：防抖（Debounce）和节流（Throttle）是两种优化高频触发事件的技术。防抖是指在一定时间内，只有最后一次触发事件才会执行回调函数；节流是指在一定时间内，只执行一次回调函数。
- **代码案例（防抖示例）**：

```
function debounce(func, delay) {
    let timer = null;
    return function () {
        const context = this;
        const args = arguments;
        if (timer) {
```

```
            clearTimeout(timer);
        }
        timer = setTimeout(() => {
            func.apply(context, args);
        }, delay);
    };
}

function search() {
    console.log('Searching...');
}

const debouncedSearch = debounce(search, 300);

const input = document.getElementById('search-input');
input.addEventListener('input', debouncedSearch);
```

- **代码案例（节流示例）**：

```
function throttle(func, delay) {
    let timer = null;
    return function () {
        if (!timer) {
            const context = this;
            const args = arguments;
            func.apply(context, args);
            timer = setTimeout(() => {
                timer = null;
            }, delay);
        }
    };
}

function scrollHandler() {
    console.log('Scrolling...');
}

const throttledScroll = throttle(scrollHandler, 500);

window.addEventListener('scroll', throttledScroll);
```

- **代码细节**：防抖函数通过 `setTimeout` 和 `clearTimeout` 来控制回调函数的执行，每次触发事件时都会清除之前的定时器，重新计时。节流函数通过一个定时器来控制回调函数的执行频率，在定时器存在时不执行回调函数。
- **案例应用**：防抖适用于搜索框输入联想、窗口大小调整等场景，避免频繁触发事件

导致性能问题。节流适用于滚动加载、按钮点击等场景，控制事件的触发频率。

11. js 写一下保留两位小数

- **原理**：在 JavaScript 中，可以使用 `toFixed()` 方法或数学计算来实现保留两位小数的功能。`toFixed()` 方法会将数字转换为字符串，并按照指定的小数位数进行四舍五入。
- **代码案例（使用 `toFixed()` 方法）**：

```
const num = 3.14159;
const result = num.toFixed(2);
console.log(result); // 输出: "3.14"
```

- **代码案例（数学计算方法）**：

```
const num = 3.14159;
const result = Math.round(num * 100) / 100;
console.log(result); // 输出: 3.14
```

- **代码细节**：`toFixed(2)` 直接将数字转换为保留两位小数的字符串。数学计算方法先将数字乘以 100，然后使用 `Math.round()` 进行四舍五入，最后再除以 100 得到保留两位小数的结果。
- **案例应用**：在处理货币金额、统计数据等需要精确到小数点后两位的场景中使用。

12. 合并区间

- **原理**：合并区间问题通常是给定一个包含多个区间的数组，需要将重叠的区间合并为一个区间。可以先对区间数组按照区间的起始位置进行排序，然后遍历数组，合并重叠的区间。
- **代码案例**：

```
function merge(intervals) {
    if (intervals.length <= 1) return intervals;
    intervals.sort((a, b) => a[0] - b[0]);
    const result = [intervals[0]];
    for (let i = 1; i < intervals.length; i++) {
        const current = intervals[i];
        const last = result[result.length - 1];
        if (current[0] <= last[1]) {
            last[1] = Math.max(last[1], current[1]);
        } else {
            result.push(current);
```

```
            }
        }
        return result;
}

const intervals = [[1, 3], [2, 6], [8, 10], [15, 18]];
const mergedIntervals = merge(intervals);
console.log(mergedIntervals); // 输出: [[1, 6], [8, 10], [15, 18]]
```

- **代码细节**：首先对区间数组进行排序，确保区间按起始位置从小到大排列。然后遍历数组，比较当前区间和结果数组中最后一个区间是否重叠，如果重叠则合并，否则将当前区间添加到结果数组中。
- **案例应用**：在日程安排、时间管理等场景中，合并重叠的时间区间可以更清晰地展示可用时间。

13. 把常见的数据结构的特点说一下

- **原理**：常见的数据结构包括数组、链表、栈、队列、树、图等，每种数据结构都有其独特的特点和适用场景。
- **代码案例（简单示例）**：

```
// 数组
const array = [1, 2, 3, 4, 5];
// 链表节点类
class ListNode {
    constructor(val) {
        this.val = val;
        this.next = null;
    }
}
const node1 = new ListNode(1);
const node2 = new ListNode(2);
node1.next = node2;
// 栈
class Stack {
    constructor() {
        this.items = [];
    }
    push(item) {
        this.items.push(item);
    }
    pop() {
        return this.items.pop();
    }
```

```
}
const stack = new Stack();
stack.push(1);
stack.push(2);
console.log(stack.pop()); // 输出: 2
// 队列
class Queue {
    constructor() {
        this.items = [];
    }
    enqueue(item) {
        this.items.push(item);
    }
    dequeue() {
        return this.items.shift();
    }
}
const queue = new Queue();
queue.enqueue(1);
queue.enqueue(2);
console.log(queue.dequeue()); // 输出: 1
```

- **代码细节**：数组是一种线性数据结构，支持随机访问，通过下标可以快速访问元素。链表是由节点组成的线性数据结构，每个节点包含数据和指向下一个节点的指针，插入和删除操作效率高。栈是一种后进先出（LIFO）的数据结构，只能在栈顶进行插入和删除操作。队列是一种先进先出（FIFO）的数据结构，在队尾插入元素，在队头删除元素。
- **案例应用**：数组适用于需要随机访问元素的场景，如排序算法。链表适用于频繁插入和删除元素的场景，如实现栈和队列。栈适用于函数调用栈、浏览器历史记录等场景。队列适用于任务调度、消息队列等场景。

14. 了解哪些设计模式

- **原理**：设计模式是指在软件开发过程中，针对反复出现的问题所总结归纳出的通用解决方案。常见的设计模式包括单例模式、工厂模式、观察者模式、装饰器模式等。
- **代码案例（单例模式示例）**：

```
class Singleton {
    constructor() {
        if (!Singleton.instance) {
            this.data = [];
            Singleton.instance = this;
        }
```

```
        return Singleton.instance;
    }
    addItem(item) {
        this.data.push(item);
    }
    getData() {
        return this.data;
    }
}

const instance1 = new Singleton();
const instance2 = new Singleton();
instance1.addItem('Item 1');
console.log(instance2.getData()); // 输出: ['Item 1']
```

- **代码细节**：单例模式确保一个类只有一个实例，并提供一个全局访问点。在 `Singleton` 类的构造函数中，检查是否已经存在实例，如果不存在则创建一个新实例，否则返回已有的实例。
- **案例应用**：单例模式适用于需要确保只有一个实例的场景，如数据库连接池、日志记录器等。工厂模式适用于根据不同的条件创建不同类型的对象。观察者模式适用于对象之间的一对多依赖关系，当一个对象的状态发生变化时，所有依赖它的对象都会得到通知。

15. 说一下观察者模式，手写观察者模式

- **原理**：观察者模式定义了一种一对多的依赖关系，当一个对象（主题）的状态发生变化时，所有依赖它的对象（观察者）都会得到通知并自动更新。
- **代码案例**：

```
class Subject {
    constructor() {
        this.observers = [];
    }
    addObserver(observer) {
        this.observers.push(observer);
    }
    removeObserver(observer) {
        const index = this.observers.indexOf(observer);
        if (index!== -1) {
            this.observers.splice(index, 1);
        }
    }
    notify() {
        this.observers.forEach(observer => observer.update());
```

```
    }
    setState(newState) {
        this.state = newState;
        this.notify();
    }
}

class Observer {
    constructor(subject) {
        this.subject = subject;
        this.subject.addObserver(this);
    }
    update() {
        console.log(`Observer updated with state:
${this.subject.state}`);
    }
}

const subject = new Subject();
const observer1 = new Observer(subject);
const observer2 = new Observer(subject);

subject.setState('New state');
```

- **代码细节**：`Subject` 类表示主题，包含一个观察者数组，提供添加、移除观察者和通知观察者的方法。`Observer` 类表示观察者，在构造函数中注册到主题上，并实现 `update` 方法用于接收主题的通知。
- **案例应用**：在前端开发中，观察者模式常用于实现事件系统、状态管理库等。例如，当一个组件的状态发生变化时，通知其他依赖该组件的组件进行更新。

16. 【代码】实现一个 call 改变 this 指向

- **原理**：`call` 方法用于调用一个函数，并指定该函数内部的 `this` 值。可以通过将函数挂载到指定的对象上，然后调用该对象的方法来实现 `this` 指向的改变。
- **代码案例**：

```
Function.prototype.myCall = function (context = window, ...args) {
    const fnSymbol = Symbol('fn');
    context[fnSymbol] = this;
    const result = context[fnSymbol](...args);
    delete context[fnSymbol];
    return result;
};
```

```
function greet(message) {
    console.log(`${message}, ${this.name}`);
}

const person = { name: 'John' };
greet.myCall(person, 'Hello'); // 输出: Hello, John
```

- **代码细节**：在 `Function.prototype` 上添加 `myCall` 方法，将调用 `myCall` 的函数挂载到 `context` 对象上，使用 `Symbol` 作为属性名避免冲突。调用该属性并传入参数，最后删除该属性并返回结果。
- **案例应用**：在需要动态改变函数 `this` 指向的场景中使用，如在回调函数中使用特定对象的方法。

17. 【代码】实现一个 promise.all

- **原理**：`Promise.all` 方法接收一个可迭代对象（通常是数组），该数组中的每个元素都是一个 `Promise` 对象，返回一个新的 `Promise`。当所有输入的 `Promise` 都成功完成时，新的 `Promise` 会以一个包含所有结果的数组进行 `resolve`；如果其中任何一个 `Promise` 被 `reject`，新的 `Promise` 会立即以该错误进行 `reject`。
- **代码案例**：

```
function promiseAll(promises) {
    return new Promise((resolve, reject) => {
        if (!Array.isArray(promises)) {
            return reject(new TypeError('Expected an array'));
        }
        const results = [];
        let completedCount = 0;
        if (promises.length === 0) {
            return resolve(results);
        }
        promises.forEach((promise, index) => {
            Promise.resolve(promise).then(result => {
                results[index] = result;
                completedCount++;
                if (completedCount === promises.length) {
                    resolve(results);
                }
            }).catch(error => {
                reject(error);
            });
        });
    });
```

```
}

const promise1 = Promise.resolve(1);
const promise2 = Promise.resolve(2);
const promise3 = Promise.resolve(3);

promiseAll([promise1, promise2, promise3]).then(results => {
    console.log(results); // 输出: [1, 2, 3]
}).catch(error => {
    console.error(error);
});
```

- **代码细节**：首先检查传入的 `promises` 是否为数组，若不是则 `reject` 。创建一个结果数组和一个计数器，遍历 `promises` 数组，使用 `Promise.resolve` 确保每个元素都是 `Promise` 对象。当一个 `Promise` 成功完成时，将结果存入结果数组，计数器加 1，当计数器等于数组长度时， `resolve` 结果数组。若有一个 `Promise` 被 `reject` ，则立即 `reject` 整个 `Promise` 。
- **案例应用**：在需要同时处理多个异步任务，并且需要等待所有任务都完成后再进行下一步操作的场景中使用，如批量请求数据。

18. 【代码】算法题（实现数组元素偏移）

- **原理**：实现数组元素偏移可以通过将数组的后 `k` 个元素移动到数组的前面。可以使用三次反转数组的方法来实现，先反转整个数组，再反转前 `k` 个元素，最后反转剩下的元素。
- **代码案例**：

```
function rotate(nums, k) {
    const n = nums.length;
    k = k % n;
    reverse(nums, 0, n - 1);
    reverse(nums, 0, k - 1);
    reverse(nums, k, n - 1);
    return nums;
}

function reverse(nums, start, end) {
    while (start < end) {
        [nums[start], nums[end]] = [nums[end], nums[start]];
        start++;
        end--;
    }

```javascript
```
```

```
function rotate(nums, k) {
    const n = nums.length;
    k = k % n;
    reverse(nums, 0, n - 1);
    reverse(nums, 0, k - 1);
    reverse(nums, k, n - 1);
    return nums;
}

function reverse(nums, start, end) {
    while (start < end) {
        [nums[start], nums[end]] = [nums[end], nums[start]];
        start++;
        end--;
    }
}

const nums = [1, 2, 3, 4, 5, 6, 7];
const k = 3;
const rotatedNums = rotate(nums, k);
console.log(rotatedNums); // 输出: [5, 6, 7, 1, 2, 3, 4]
```

- **代码细节**：
  - 首先计算 `k` 对数组长度 `n` 取模，避免 `k` 大于数组长度的情况。
  - 第一次调用 `reverse` 函数反转整个数组。
  - 第二次调用 `reverse` 函数反转数组的前 `k` 个元素。
  - 第三次调用 `reverse` 函数反转数组中剩余的元素。
- **案例应用**：在处理循环数组、周期性数据等场景中会用到，比如实现一个时钟表盘上数字的循环滚动效果。

19. 【代码】 数组拉平（flatten）

- **原理**：数组拉平是将嵌套数组转换为一维数组。可以使用递归的方式遍历数组，若元素是数组则继续递归处理，若元素是基本类型则添加到结果数组中。
- **代码案例**：

```
function flatten(arr) {
    const result = [];
    for (let i = 0; i < arr.length; i++) {
        if (Array.isArray(arr[i])) {
            result.push(...flatten(arr[i]));
        } else {
            result.push(arr[i]);
```

```
        }
    }
    return result;
}

const nestedArray = [1, [2, [3, 4], 5], 6];
const flattenedArray = flatten(nestedArray);
console.log(flattenedArray); // 输出: [1, 2, 3, 4, 5, 6]
```

- **代码细节**：
  - 创建一个空数组 `result` 用于存储拉平后的元素。
  - 遍历输入数组 `arr`，若元素是数组，则递归调用 `flatten` 函数并将结果展开添加到 `result` 中；若元素不是数组，则直接添加到 `result` 中。
- **案例应用**：在处理树形结构数据、多维矩阵转换等场景中会用到，比如将一个多级菜单的数据结构转换为一维列表。

20. 节流和防抖的使用场景?

- **原理**：如前面所述，防抖是在一定时间内只有最后一次触发事件才执行回调，节流是在一定时间内只执行一次回调。
- **使用场景分析**：
  - **防抖**：
    - **搜索框输入联想**：当用户在搜索框输入内容时，不需要每次输入一个字符就发起请求，而是等用户停止输入一段时间后再发起请求，减少不必要的请求次数。
    - **窗口大小调整**：当窗口大小改变时，不需要每次大小变化都触发重新布局等操作，等用户停止调整一段时间后再进行处理。
  - **节流**：
    - **滚动加载**：在页面滚动时，不需要每次滚动都去检查是否需要加载更多数据，而是每隔一段时间检查一次，避免频繁触发加载操作。
    - **按钮点击**：对于一些防止用户重复点击的按钮，比如提交表单按钮，使用节流可以确保在一定时间内只响应一次点击事件。

21. 为什么有 hooks？hooks 是干什么用的?

- **原理**：在 React 中，Hooks 是 React 16.8 引入的新特性，主要是为了解决类组件的一些问题，如代码复用困难、逻辑复杂时难以拆分等。Hooks 可以让你在不编写 `class` 的情况下使用 `state` 以及其他 React 特性。
- **代码案例（`useState` 示例）**：

```
import React, { useState } from'react';

function Counter() {
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div>
    );
}

export default Counter;
```

- **代码细节**：`useState` 是一个 Hook，它返回一个数组，第一个元素是状态值，第二个元素是更新状态的函数。每次调用 `setCount` 时，组件会重新渲染并更新 `count` 的显示值。
- **案例应用**：Hooks 使得函数组件可以有状态，并且可以将复杂的逻辑拆分成多个小的 Hook，提高代码的复用性和可维护性。例如在多个组件中复用表单验证逻辑、数据获取逻辑等。

22. 常见的 hooks?

- **原理**：React 提供了多个内置的 Hooks，还有很多社区自定义的 Hooks，常见的内置 Hooks 可以帮助开发者在函数组件中使用不同的 React 特性。
- **常见 Hooks 介绍及代码案例**：
  - **`useState`**：用于在函数组件中添加状态。

```
import React, { useState } from'react';

function Example() {
    const [count, setCount] = useState(0);
    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count +
1)}>Increment</button>
        </div>
    );
}
```

- **`useEffect`**：用于处理副作用，如数据获取、订阅、DOM 操作等。

```
import React, { useState, useEffect } from'react';

function DataFetching() {
    const [data, setData] = useState(null);
    useEffect(() => {
        fetch('https://api.example.com/data')
          .then(response => response.json())
          .then(data => setData(data));
    }, []);
    return (
        <div>
            {data? <p>{data.message}</p> : <p>Loading...</p>}
        </div>
    );
}
```

- **`useContext`**：用于在组件之间共享数据，避免层层传递 `props`。

```
import React, { createContext, useContext } from'react';

const ThemeContext = createContext();

function ThemeProvider({ children }) {
    const theme = 'dark';
    return (
        <ThemeContext.Provider value={theme}>
            {children}
        </ThemeContext.Provider>
    );
}

function ChildComponent() {
    const theme = useContext(ThemeContext);
    return <p>Current theme: {theme}</p>;
}

function App() {
    return (
        <ThemeProvider>
            <ChildComponent />
        </ThemeProvider>
    );
```

```
}
```

- **案例应用**：不同的 Hooks 适用于不同的场景，`useState` 用于状态管理，`useEffect` 用于处理异步操作和副作用，`useContext` 用于组件间数据共享。

23. 原生的 hooks?

- **原理**：原生的 Hooks 指的是 React 官方提供的内置 Hooks，它们是 React 核心库的一部分，可以直接在 React 项目中使用。
- **常见原生 Hooks 总结**：
  - `useState`：管理组件的状态，允许函数组件有状态变化。
  - `useEffect`：处理副作用，如数据获取、订阅、DOM 操作等。在组件渲染后或依赖项变化时执行回调函数。
  - `useContext`：获取上下文对象，用于在组件树中共享数据。
  - `useReducer`：类似于 Redux 的 `reducer` 模式，用于管理复杂的状态逻辑。
  - `useCallback`：返回一个记忆化的回调函数，用于优化性能，避免不必要的函数重新创建。
  - `useMemo`：返回一个记忆化的值，用于优化性能，避免不必要的计算。
  - `useRef`：创建一个可变的引用对象，通常用于保存 DOM 节点或在组件的整个生命周期内保持值不变。
  - `useImperativeHandle`：用于自定义使用 `ref` 时暴露给父组件的实例值。
  - `useLayoutEffect`：与 `useEffect` 类似，但会在 DOM 更新后同步执行，常用于需要测量 DOM 布局的场景。
  - `useDebugValue`：用于在 React DevTools 中显示自定义 Hook 的调试信息。
- **案例应用**：在不同的开发场景中选择合适的原生 Hooks 可以提高开发效率和组件性能，例如在性能敏感的组件中使用 `useCallback` 和 `useMemo` 进行优化。

24. useContext 如何进行组件之间的传输?

- **原理**：`useContext` 结合 `createContext` 和 `Context.Provider` 实现组件间的数据传输。`createContext` 创建一个上下文对象，`Context.Provider` 用于提供数据，`useContext` 用于在组件中获取上下文对象中的数据。
- **代码案例**：

```
import React, { createContext, useContext } from'react';

// 创建上下文对象
const UserContext = createContext();
```

```jsx
// 提供数据的组件
function UserProvider({ children }) {
    const user = { name: 'John', age: 30 };
    return (
        <UserContext.Provider value={user}>
            {children}
        </UserContext.Provider>
    );
}

// 接收数据的组件
function DisplayUser() {
    const user = useContext(UserContext);
    return (
        <div>
            <p>Name: {user.name}</p>
            <p>Age: {user.age}</p>
        </div>
    );
}

function App() {
    return (
        <UserProvider>
            <DisplayUser />
        </UserProvider>
    );
}

export default App;
```

- **代码细节**：
  - 使用 `createContext` 创建 `UserContext`。
  - `UserProvider` 组件使用 `UserContext.Provider` 包裹子组件，并通过 `value` 属性提供数据。
  - `DisplayUser` 组件使用 `useContext(UserContext)` 获取上下文对象中的数据并显示。
- **案例应用**：在多层嵌套的组件结构中，当多个组件需要访问相同的数据时，使用 `useContext` 可以避免通过 `props` 层层传递数据，提高代码的简洁性和可维护性。

25. hash 模式和 history 模式有什么区别?

- **原理**：在前端路由中，hash 模式和 history 模式是两种不同的实现方式，主要区别在于 URL 的表现形式和浏览器历史记录的管理。
- **区别分析**：
  - **URL 表现形式**：
    - **hash 模式**：URL 中使用 `#` 符号来分隔路径，例如 `http://example.com/#/home`。`#` 后面的内容不会发送到服务器，浏览器只会根据 `#` 后面的路径进行前端路由匹配。
    - **history 模式**：URL 看起来像正常的路径，例如 `http://example.com/home`。它使用 HTML5 的 `History API` 来管理浏览器历史记录，路径会发送到服务器。
  - **浏览器历史记录管理**：
    - **hash 模式**：每次 `#` 后面的路径变化都会添加一条新的历史记录，通过浏览器的前进后退按钮可以在这些历史记录之间切换。
    - **history 模式**：使用 `pushState` 和 `replaceState` 方法来操作浏览器历史记录，也可以实现前进后退功能，但需要服务器端的支持，因为服务器需要对不同的路径返回相同的 HTML 文件。
- **代码案例（Vue Router 示例）**：

```
import Vue from 'vue';
import VueRouter from 'vue-router';
import Home from './views/Home.vue';
import About from './views/About.vue';

Vue.use(VueRouter);

const routes = [
    {
        path: '/home',
        name: 'Home',
        component: Home
    },
    {
        path: '/about',
        name: 'About',
        component: About
    }
];

// hash 模式
const routerHash = new VueRouter({
    mode: 'hash',
    routes
```

```
});

// history 模式
const routerHistory = new VueRouter({
    mode: 'history',
    routes
});
```

- **案例应用**：hash 模式适合简单的单页应用，不需要服务器端额外配置；history 模式适合对 URL 美观性有要求的应用，但需要服务器端进行相应的配置，如在 Node.js 中使用 Express 框架时需要配置中间件来处理不同路径的请求。

26. 实现 promise any 方法 和 once 方法?

- `Promise.any` **实现**：
  - **原理**：`Promise.any` 接收一个可迭代对象，返回一个新的 `Promise`。当可迭代对象中的任何一个 `Promise` 成功完成时，新的 `Promise` 会以该 `Promise` 的结果进行 `resolve`；如果所有 `Promise` 都被 `reject`，则新的 `Promise` 会以一个包含所有错误的 `AggregateError` 进行 `reject`。
  - **代码案例**：

```
Promise.myAny = function (promises) {
    return new Promise((resolve, reject) => {
        const errors = [];
        let rejectedCount = 0;
        promises.forEach((promise, index) => {
            Promise.resolve(promise).then(result => {
                resolve(result);
            }).catch(error => {
                errors[index] = error;
                rejectedCount++;
                if (rejectedCount === promises.length) {
                    reject(new AggregateError(errors, 'All promises
were rejected'));
                }
            });
        });
    });
};

const promise1 = Promise.reject(new Error('Error 1'));
const promise2 = Promise.resolve('Success');
const promise3 = Promise.reject(new Error('Error 2'));
```

```
Promise.myAny([promise1, promise2, promise3]).then(result => {
    console.log(result); // 输出: Success
}).catch(error => {
    console.error(error);
});
```

```
- **代码细节**:
    - 遍历 `promises` 数组，使用 `Promise.resolve` 确保每个元素都是 `Promise` 对象。
    - 当有一个 `Promise` 成功时，立即 `resolve` 该结果。
    - 当所有 `Promise` 都被 `reject` 时，`reject` 一个包含所有错误的 `AggregateError`。
```

- **`once` 方法实现**:
    - **原理**: `once` 方法用于确保一个函数只执行一次。可以通过闭包来实现，在函数内部记录函数是否已经执行过。
    - **代码案例**:

```
function once(func) {
    let hasBeenCalled = false;
    let result;
    return function (...args) {
        if (!hasBeenCalled) {
            result = func.apply(this, args);
            hasBeenCalled = true;
        }
        return result;
    };
}

function greet() {
    console.log('Hello!');
}

const onceGreet = once(greet);
onceGreet(); // 输出: Hello!
onceGreet(); // 无输出
```

```
- **代码细节**:
    - 使用闭包变量 `hasBeenCalled` 记录函数是否已经执行过。
    - 第一次调用时执行函数并记录结果，将 `hasBeenCalled` 设为 `true`，后续调用直接返回第一次的结果。
```

- **案例应用**: `Promise.any` 适用于多个异步任务，只要有一个成功就可以继续后续

操作的场景；`once` 方法适用于只需要执行一次的初始化操作等场景。

浏览器分类

1. http 和 https 的区别?

- **原理**：`HTTP`（超文本传输协议）是明文传输协议，数据在传输过程中容易被截取和篡改，存在安全风险。`HTTPS`（超文本传输安全协议）是在 `HTTP` 的基础上加入了 `SSL/TLS` 协议进行加密，保证了数据传输的安全性。
- **区别分析**：
  - **安全性**：
    - `HTTP`：数据以明文形式传输，容易被中间人窃取和篡改，如用户的账号密码、敏感信息等可能会泄露。
    - `HTTPS`：使用 `SSL/TLS` 协议对数据进行加密，即使数据被截取，攻击者也无法获取其中的敏感信息。
  - **端口号**：
    - `HTTP`：默认使用端口号 80。
    - `HTTPS`：默认使用端口号 443。
  - **证书**：
    - `HTTP`：不需要证书。
    - `HTTPS`：需要向认证机构申请 SSL 证书，用于验证服务器的身份和加密数据传输。
- **案例应用**：在涉及用户敏感信息交互的场景，如网上银行、电子商务平台等，必须使用 `HTTPS` 协议来保障用户数据的安全。而对于一些对安全性要求不高的静态页面，如新闻资讯网站的部分页面，可使用 `HTTP` 协议。

2. 开发过程中有没有用过抓包或者代理工具？类似 charles?

- **原理**：抓包工具（如 Charles、Fiddler 等）和代理工具可以拦截、监控和修改网络请求和响应。其原理是在客户端和服务器之间充当中间代理，捕获经过的数据，方便开发者分析请求和响应的详细信息，如请求头、请求体、响应状态码、响应内容等。
- **使用场景及案例**：
  - **接口调试**：在开发前后端分离的项目时，通过抓包工具可以查看前端发送的请求是否符合预期，后端返回的数据格式和内容是否正确。例如，前端发送一个登录请求，通过抓包工具可以查看请求的 `URL`、请求方法、请求参数以及后端返回的状态码和响应数据，帮助定位接口调用过程中出现的问题。

- **性能分析**：分析请求和响应的时间，找出性能瓶颈。比如，查看某个页面加载时各个请求的耗时情况，判断是哪个请求导致页面加载缓慢，进而进行优化。
- **数据篡改**：在测试过程中，可以使用抓包工具修改请求或响应的数据，模拟不同的场景进行测试。例如，修改请求中的参数，测试后端接口对异常数据的处理能力。

3. url 链接每一部分代表什么含义

- **原理**：一个完整的 `URL`（统一资源定位符）由多个部分组成，每个部分都有特定的含义，用于定位互联网上的资源。
- **示例及各部分含义**：
  - 示例 `URL`：`https://www.example.com:8080/path/to/resource?param1=value1&param2=value2#section`
  - **协议（Protocol）**：`https` 表示使用的是超文本传输安全协议，规定了客户端和服务器之间通信的规则和方式。
  - **域名（Domain Name）**：`www.example.com` 是网站的域名，用于标识服务器的位置。域名会通过 DNS（域名系统）解析为对应的 IP 地址。
  - **端口号（Port）**：`8080` 是服务器监听的端口号，用于区分不同的服务。默认情况下，`HTTP` 使用 80 端口，`HTTPS` 使用 443 端口。
  - **路径（Path）**：`/path/to/resource` 表示服务器上资源的具体位置，类似于文件系统中的文件路径。
  - **查询参数（Query Parameters）**：`param1=value1&param2=value2` 是传递给服务器的额外参数，用于向服务器传递数据或进行条件筛选等操作。
  - **片段标识符（Fragment Identifier）**：`#section` 用于定位页面内的特定部分，当页面加载完成后，浏览器会自动滚动到该部分。

4. 跨域与同源

- **原理**：同源策略是浏览器的一个重要安全机制，它规定浏览器只允许访问同源（协议、域名、端口都相同）的资源。如果一个请求的源（协议、域名、端口）与当前页面的源不同，就会产生跨域问题。
- **同源示例及非同源示例**：
  - **同源**：
    - 页面 `https://www.example.com/page1.html` 和请求 `https://www.example.com/api/data` 是同源的，因为协议都是 `https`，域名都是 `www.example.com`，端口默认都是 443。
  - **非同源（跨域）**：
    - 页面 `https://www.example.com/page1.html` 和请求

`http://anotherdomain.com/api/data` 存在跨域问题，因为协议不同（一个是 `https`，一个是 `http`），域名也不同。

- **跨域解决方案**：
  - **JSONP（JSON with Padding）**：利用 `<script>` 标签的 `src` 属性不受同源策略限制的特点，通过动态创建 `<script>` 标签来实现跨域数据请求。
  - **CORS（跨域资源共享）**：是一种现代的跨域解决方案，服务器端设置响应头（如 `Access-Control-Allow-Origin`）来允许特定源的请求访问资源。
  - **代理服务器**：在同源的服务器上设置代理，将客户端的请求转发到目标服务器，再将响应返回给客户端。

5. 输入网址到展示页面的整个过程?

- **原理**：当在浏览器中输入网址并回车后，会经历一系列复杂的过程，最终将页面展示给用户。
- **详细过程**：
  i. **DNS 解析**：浏览器首先会检查本地的 DNS 缓存，如果没有找到对应的 IP 地址，会向 DNS 服务器发送请求，将域名解析为对应的 IP 地址。例如，输入 `www.example.com`，通过 DNS 解析得到其对应的 IP 地址。
  ii. **TCP 连接**：浏览器使用解析得到的 IP 地址和端口号（默认 `HTTP` 为 80，`HTTPS` 为 443）与服务器建立 TCP 连接，通过三次握手确保连接的可靠性。
  iii. **HTTP 请求**：浏览器向服务器发送 HTTP 请求，包括请求行（如 `GET /path/to/resource HTTP/1.1`）、请求头（包含一些附加信息，如 `User - Agent`、`Cookie` 等）和请求体（如果有）。
  iv. **服务器处理请求**：服务器接收到请求后，根据请求的内容进行相应的处理，如查询数据库、调用后端接口等。
  v. **HTTP 响应**：服务器处理完请求后，返回 HTTP 响应，包括响应行（如 `HTTP/1.1 200 OK`）、响应头（包含一些信息，如 `Content - Type`、`Set - Cookie` 等）和响应体（页面的 HTML、CSS、JavaScript 等资源）。
  vi. **浏览器解析渲染页面**：
    - **解析 HTML**：浏览器解析 HTML 文件，构建 DOM 树。
    - **解析 CSS**：解析 CSS 文件，构建 CSSOM 树。
    - **合并渲染树**：将 DOM 树和 CSSOM 树合并为渲染树。
    - **布局（Layout）**：计算渲染树中每个节点的位置和大小。
    - **绘制（Painting）**：将渲染树中的节点绘制到屏幕上。
  vii. **TCP 连接关闭**：页面渲染完成后，浏览器和服务器之间的 TCP 连接通过四次挥手关闭。

6. http 链接创建的过程?

- **原理**：`HTTP` 是基于 `TCP` 协议的应用层协议，`HTTP` 链接的创建依赖于 `TCP` 连接的建立。
- **详细过程**：
  i. **客户端发起连接请求**：客户端（浏览器）根据输入的 `URL` 解析出服务器的 IP 地址和端口号，然后向服务器发送 `TCP` 连接请求（SYN 包），该包包含客户端的初始序列号等信息。
  ii. **服务器响应连接请求**：服务器接收到客户端的 SYN 包后，向客户端发送一个 SYN + ACK 包，表示同意建立连接，并包含服务器的初始序列号和对客户端序列号的确认信息。
  iii. **客户端确认连接**：客户端接收到服务器的 SYN + ACK 包后，向服务器发送一个 ACK 包，表示确认连接。至此，`TCP` 连接建立完成。
  iv. **发送 HTTP 请求**：在 `TCP` 连接建立后，客户端向服务器发送 `HTTP` 请求，包括请求行、请求头和请求体（如果有）。
  v. **服务器处理请求并返回响应**：服务器接收到 `HTTP` 请求后，进行相应的处理，然后返回 `HTTP` 响应，包括响应行、响应头和响应体。
  vi. **关闭连接**：当请求和响应完成后，客户端和服务器可以选择关闭 `TCP` 连接，通过四次挥手的过程释放连接资源。

7. 长链接和短链接的区别是什么?

- **原理**：长链接和短链接是指客户端和服务器之间连接的保持方式，主要区别在于连接的持续时间和使用场景。
- **区别分析**：
  - **连接持续时间**：
    - **短链接**：每次请求时建立连接，请求完成后立即关闭连接。例如，在传统的 `HTTP 1.0` 协议中，每个请求都需要重新建立 `TCP` 连接，请求结束后连接关闭。
    - **长链接**：在一次连接建立后，会保持一段时间不关闭，客户端和服务器可以在这个连接上多次发送请求和响应。例如，在 `HTTP 1.1` 及以后的版本中，默认支持长连接（`Connection: keep - alive`）。
  - **性能开销**：
    - **短链接**：每次建立和关闭连接都需要一定的时间和资源开销，如 `TCP` 连接的三次握手和四次挥手过程。在频繁请求的场景下，会增加性能开销。
    - **长链接**：避免了频繁建立和关闭连接的开销，提高了性能。但长连接会占用服务器的资源，如果长时间不使用，也需要进行管理和清理。
  - **使用场景**：

- **短链接**：适用于请求频率较低、对连接实时性要求不高的场景，如静态页面的请求。
- **长链接**：适用于需要频繁交互、对实时性要求较高的场景，如即时通讯、游戏服务器等。

8. 加密的具体过程 、ssl 加密的细节 、 http2.0

- **SSL/TLS 加密过程**：
  - **客户端发起握手请求**：客户端向服务器发送 `ClientHello` 消息，包含客户端支持的 SSL/TLS 版本、加密算法列表、随机数等信息。
  - **服务器响应握手请求**：服务器收到 `ClientHello` 消息后，发送 `ServerHello` 消息，选择一个 SSL/TLS 版本和加密算法，并发送服务器的证书和另一个随机数。
  - **客户端验证证书**：客户端验证服务器证书的有效性，包括证书的颁发机构、有效期等。如果证书有效，客户端会生成一个会话密钥，使用服务器证书中的公钥进行加密后发送给服务器。
  - **服务器解密会话密钥**：服务器使用自己的私钥解密客户端发送的会话密钥。
  - **建立安全连接**：客户端和服务器使用会话密钥进行对称加密通信，后续的请求和响应都使用该密钥进行加密和解密。
- **HTTP/2.0 特性**：
  - **二进制分帧**：`HTTP/2.0` 将请求和响应数据分割成二进制帧，提高了传输效率。
  - **多路复用**：允许在一个连接上同时发送多个请求和响应，避免了 `HTTP 1.x` 中的队头阻塞问题。
  - **头部压缩**：使用 `HPACK` 算法对请求和响应的头部进行压缩，减少了头部数据的传输量。
  - **服务器推送**：服务器可以主动向客户端推送资源，提前将客户端可能需要的资源发送给客户端，提高页面加载速度。

9. JSONP 和 Cors 为什么可以解决同源问题?

- **JSONP（JSON with Padding）**：
  - **原理**：`JSONP` 利用了 `<script>` 标签的 `src` 属性不受同源策略限制的特点。前端页面动态创建 `<script>` 标签，将请求的 `URL` 作为 `src` 属性的值，并在 `URL` 中添加一个回调函数名作为参数。服务器收到请求后，将数据包装在回调函数中返回给客户端。客户端的 `<script>` 标签加载该响应后，会执行回调函数，从而获取到服务器返回的数据。
  - **解决同源问题的原因**：由于 `<script>` 标签的 `src` 属性可以跨域请求资源，

所以通过这种方式绕过了浏览器的同源策略限制。

- **CORS（跨域资源共享）**：
  - **原理**：`CORS` 是一种现代的跨域解决方案，基于 `HTTP` 协议，通过服务器端设置响应头来允许特定源的请求访问资源。当客户端发起跨域请求时，浏览器会自动在请求头中添加 `Origin` 字段，标识请求的源。服务器收到请求后，检查 `Origin` 字段，如果该源在允许的范围内，会在响应头中添加相应的跨域允许信息（如 `Access - Control - Allow - Origin`），浏览器根据这些信息判断是否允许该跨域请求。
  - **解决同源问题的原因**：通过服务器端的配置，明确允许哪些源可以访问资源，从而在保证安全的前提下解决了跨域问题。

10. 浏览器发请求时有个 options 请求，做什么的（跨域的预检请求）

- **原理**：当浏览器发起跨域请求时，如果请求是复杂请求（如使用 `PUT`、`DELETE` 等非简单请求方法，或者包含自定义请求头、`Content - Type` 不是简单类型等），浏览器会先发送一个 `OPTIONS` 请求，称为预检请求。
- **预检请求的作用**：
  - **检查服务器支持情况**：通过 `OPTIONS` 请求，浏览器询问服务器是否支持当前请求的方法、请求头和源。服务器会在响应头中返回允许的请求方法（如 `Access - Control - Allow - Methods`）、允许的请求头（如 `Access - Control - Allow - Headers`）和允许的源（如 `Access - Control - Allow - Origin`）等信息。
  - **避免不必要的跨域请求**：如果服务器不支持当前请求的方法、请求头或源，浏览器就不会发送真正的请求，从而避免了不必要的跨域请求和潜在的安全风险。

11. 浏览器的组成

- **原理**：现代浏览器主要由多个组件组成，每个组件负责不同的功能，协同工作以实现网页的加载和渲染等功能。
- **主要组件**：
  - **用户界面**：包括地址栏、书签栏、标签页等，用于用户与浏览器进行交互。
  - **浏览器引擎**：负责协调和管理其他组件的工作，如解析 HTML、CSS 等资源，将渲染结果传递给渲染引擎。
  - **渲染引擎**：负责解析 HTML 和 CSS，构建 DOM 树和 CSSOM 树，合并为渲染树，进行布局和绘制，将页面内容显示在屏幕上。不同的浏览器使用不同的渲染引擎，如 Chrome 使用 Blink 引擎，Firefox 使用 Gecko 引擎。
  - **网络模块**：负责处理网络请求和响应，包括 DNS 解析、TCP 连接建立、HTTP

请求发送和响应接收等。
    - **JavaScript 引擎**：负责执行 JavaScript 代码，如 Chrome 的 V8 引擎。它将 JavaScript 代码编译为机器码，提高执行效率。
    - **数据存储模块**：负责管理浏览器的本地存储，如 `localStorage`、`sessionStorage`、`IndexedDB` 等，用于保存用户的浏览数据、缓存等。
    - **安全模块**：负责保障浏览器的安全，如防止跨站脚本攻击（XSS）、跨站请求伪造（CSRF）等，对网页的内容和请求进行安全检查。

12. DOM BOM 和 BOM 的 API

- **原理**：`DOM`（文档对象模型）和 `BOM`（浏览器对象模型）是浏览器编程中的重要概念。
    - **DOM**：是 HTML 文档的树形结构表示，将 HTML 文档解析为一个由节点组成的树，每个节点可以是元素节点、文本节点等。通过 `DOM`，可以使用 JavaScript 动态地操作 HTML 文档的内容、结构和样式。
    - **BOM**：是浏览器窗口的对象模型，提供了与浏览器窗口相关的操作接口，如窗口大小调整、导航、弹出对话框等。
- **常见 API**：
    - **DOM API**：
        - `document.getElementById()`：通过元素的 `id` 属性获取元素节点。
        - `document.createElement()`：创建一个新的元素节点。
        - `element.appendChild()`：将一个节点添加为另一个节点的子节点。
        - `element.style`：用于操作元素的样式。
    - **BOM API**：
        - `window.alert()`：弹出一个警告对话框。
        - `window.location`：用于获取和设置当前页面的 `URL`。
        - `window.history`：用于操作浏览器的历史记录，如前进、后退等。
        - `window.innerWidth` 和 `window.innerHeight`：获取浏览器窗口的内部宽度和高度。

13. websocket 通信原理

- **原理**：`WebSocket` 是一种在单个 `TCP` 连接上进行全双工通信的协议，它允许浏览器和服务器之间进行实时通信。
- **通信过程**：

i. **握手阶段**：客户端向服务器发送一个 `HTTP` 请求，请求头中包含 `Upgrade: websocket` 和 `Connection: Upgrade` 等信息，表示要升级为 `WebSocket` 连接。服务器收到请求后，如果支持 `WebSocket` 协议，会返回一个状态码为 101 的响应，表示同意升级连接。

ii. **建立连接**：握手成功后，`TCP` 连接保持打开，客户端和服务器可以在这个连接上进行双向通信。

iii. **数据传输**：客户端和服务器可以随时向对方发送数据，数据以帧的形式传输。`WebSocket` 协议定义了不同类型的帧，如文本帧、二进制帧等。

iv. **关闭连接**：当一方需要关闭连接时，会发送一个关闭帧给对方。对方收到关闭帧后，会发送一个确认关闭帧，然后双方关闭 TCP 连接。

- **代码案例（使用 Node.js 和浏览器示例）**：
  - **服务器端（使用 Node.js 的 ws 库）**：

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', function connection(ws) {
    ws.on('message', function incoming(message) {
        console.log('received: %s', message);
        ws.send('Server received: ' + message);
    });
});
```

```
- **客户端（浏览器端）**:
```

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>WebSocket Example</title>
</head>

<body>
    <button id="sendButton">Send Message</button>
    <script>
        const socket = new WebSocket('ws://localhost:8080');
```

```
        socket.onopen = function () {
            console.log('Connected to the server');
        };

        socket.onmessage = function (event) {
            console.log('Received from server: ', event.data);
        };

        socket.onclose = function () {
            console.log('Disconnected from the server');
        };

        const sendButton = document.getElementById('sendButton');
        sendButton.addEventListener('click', function () {
            socket.send('Hello, server!');
        });
    </script>
</body>

</html>
```

- **代码细节**：
  - 服务器端使用 `ws` 库创建一个 WebSocket 服务器，监听 8080 端口。当有客户端连接时，监听客户端发送的消息，并将消息原样返回给客户端。
  - 客户端使用 `WebSocket` 构造函数创建一个 WebSocket 实例，连接到服务器。监听 `open`、`message` 和 `close` 事件，分别处理连接建立、接收消息和连接关闭的情况。点击按钮时，向服务器发送消息。
- **案例应用**：WebSocket 适用于需要实时通信的场景，如在线聊天、实时数据展示（股票行情、实时监控等）、多人游戏等。

14. http 状态码

- **原理**：HTTP 状态码是服务器返回给客户端的三位数字代码，用于表示 HTTP 请求的结果。不同的状态码代表不同的含义，客户端可以根据状态码来判断请求的处理情况。
- **常见状态码分类及示例**：
  - **1xx（信息性状态码）**：表示临时响应，通常用于表示请求已接收，继续处理。例如，`100 Continue` 表示客户端可以继续发送请求的其余部分。
  - **2xx（成功状态码）**：表示请求成功。
    - `200 OK`：表示请求成功，服务器已经处理了请求并返回了相应的结果。
    - `201 Created`：表示请求成功，并且在服务器上创建了一个新的资源。
  - **3xx（重定向状态码）**：表示需要客户端采取进一步的操作才能完成请求，通

常用于重定向。
    - `301 Moved Permanently`：表示请求的资源已永久移动到新的 URL。
    - `302 Found`：表示请求的资源临时移动到新的 URL。
  - **4xx（客户端错误状态码）**：表示客户端的请求有错误，无法被服务器处理。
    - `400 Bad Request`：表示客户端的请求语法错误，不能被服务器识别。
    - `401 Unauthorized`：表示请求需要身份验证，客户端未提供有效的身份信息。
    - `403 Forbidden`：表示服务器理解请求客户端的请求，但是拒绝执行此请求。
    - `404 Not Found`：表示请求的资源不存在。
  - **5xx（服务器错误状态码）**：表示服务器在处理请求时发生了错误。
    - `500 Internal Server Error`：表示服务器内部发生了错误，无法完成请求。
    - `503 Service Unavailable`：表示服务器暂时无法处理请求，通常是由于服务器过载或维护中。
- **案例应用**：在开发和调试过程中，通过查看 HTTP 状态码可以快速定位问题。例如，当出现 `404` 状态码时，需要检查请求的 URL 是否正确；当出现 `500` 状态码时，需要检查服务器端的代码逻辑是否存在错误。

工程化分类

1. webpack 的作用，说说你对他的理解

- **原理**：Webpack 是一个模块打包工具，它可以将各种类型的模块（如 JavaScript、CSS、图片等）打包成一个或多个文件，主要用于优化前端项目的资源管理和性能。
- **主要作用**：
  - **模块打包**：Webpack 可以处理项目中的各种模块依赖关系，将所有的模块打包成一个或多个文件，减少浏览器的请求次数，提高页面加载速度。例如，一个项目中有多个 JavaScript 文件和 CSS 文件，Webpack 可以将它们打包成一个或几个文件。
  - **资源处理**：通过使用不同的 loader 和 plugin，Webpack 可以处理各种类型的资源，如将 Sass 或 Less 转换为 CSS，压缩图片，处理字体文件等。
  - **代码分割**：Webpack 支持代码分割，可以将项目代码分割成多个小块，实现按需加载。例如，将不同页面的代码分割成不同的文件，当用户访问某个页面时，只加载该页面所需的代码。
- **代码案例（简单配置示例）**：

```
const path = require('path');

module.exports = {
    entry: './src/index.js',
    output: {
        path: path.resolve(__dirname, 'dist'),
        filename: 'bundle.js'
    },
    module: {
        rules: [
            {
                test: /\.css$/,
                use: ['style-loader', 'css-loader']
            }
        ]
    }
};
```

- **代码细节**：
  - `entry`：指定项目的入口文件，Webpack 会从该文件开始分析模块依赖关系。
  - `output`：指定打包后的文件输出路径和文件名。
  - `module.rules`：配置不同类型文件的 loader，例如对于 `.css` 文件，使用 `style-loader` 和 `css-loader` 进行处理。
- **案例应用**：在大型前端项目中，Webpack 可以帮助开发者更好地管理项目的资源和依赖关系，提高开发效率和项目性能。例如，在 React 或 Vue 项目中，通常会使用 Webpack 进行打包和构建。

2. 你提到了前端工程化，说说你对它的理解

- **原理**：前端工程化是指将软件开发的工程方法应用于前端开发，通过一系列工具、流程和规范来提高前端开发的效率、质量和可维护性。
- **主要方面**：
  - **模块化开发**：将项目拆分成多个小的模块，每个模块具有独立的功能和职责，提高代码的复用性和可维护性。例如，使用 ES6 的模块语法（`import` 和 `export`）或 CommonJS 规范进行模块开发。
  - **自动化构建**：使用构建工具（如 Webpack、Gulp 等）自动完成代码的编译、打包、压缩、合并等任务，减少人工操作，提高开发效率。
  - **代码规范**：制定统一的代码规范和风格指南，如 ESLint、Prettier 等工具可以帮助开发者检查和修复代码中的语法错误和风格问题，保证代码的一致性。
  - **版本控制**：使用版本控制系统（如 Git）来管理项目的代码，方便团队协作和

代码的回溯、分支管理等。
    - **测试和部署**：建立自动化测试流程，使用测试框架（如 Jest、Mocha 等）对代码进行单元测试、集成测试等，确保代码的质量。同时，实现自动化部署，将代码部署到生产环境。
- **案例应用**：在一个多人协作的前端项目中，通过前端工程化的方法，可以规范开发流程，提高团队协作效率，减少代码冲突和错误，同时提高项目的可维护性和性能。

3. vite 和 webpack 的区别是什么呢

- **原理**：Vite 和 Webpack 都是前端构建工具，但它们的实现原理和适用场景有所不同。
- **区别分析**：
    - **启动速度**：
        - **Vite**：基于原生 ES 模块，在开发环境下采用按需加载的方式，不需要像 Webpack 那样对整个项目进行打包，因此启动速度非常快。当修改代码时，Vite 可以实现快速的热更新，只更新修改的模块。
        - **Webpack**：在启动时需要对整个项目进行打包，分析模块依赖关系，因此启动速度相对较慢，尤其是在大型项目中。
    - **构建方式**：
        - **Vite**：开发环境下利用浏览器的原生 ES 模块支持，直接在浏览器中加载模块。生产环境下使用 Rollup 进行打包，将代码打包成适合生产环境的文件。
        - **Webpack**：使用自己的打包机制，通过 loader 和 plugin 处理各种类型的模块，将所有模块打包成一个或多个文件。
    - **配置复杂度**：
        - **Vite**：配置相对简单，默认提供了很多常用的配置，对于小型项目可以快速上手。例如，只需要配置入口文件和输出路径等基本信息。
        - **Webpack**：配置相对复杂，需要根据项目的需求配置各种 loader 和 plugin，对于初学者来说有一定的学习成本。
    - **适用场景**：
        - **Vite**：适用于小型项目和快速迭代的项目，能够提供快速的开发体验。例如，一些简单的静态网站、原型项目等。
        - **Webpack**：适用于大型项目和复杂的项目，具有强大的功能和丰富的插件生态系统。例如，企业级的 Web 应用、大型电商网站等。

4. vite 项目怎么部署到线上

- **原理**：将 Vite 项目部署到线上需要完成项目的构建、服务器配置和文件上传等步骤，确保项目在生产环境中能够正常运行。
- **部署步骤**：
  i. **项目构建**：在本地项目根目录下运行 `npm run build` 命令，Vite 会根据配置文件将项目打包成生产环境可用的文件，通常会生成一个 `dist` 目录，包含打包后的 HTML、CSS、JavaScript 等文件。
  ii. **服务器选择**：选择合适的服务器来部署项目，常见的服务器有 Nginx、Apache 等。可以选择云服务器（如阿里云、腾讯云等）或自己搭建服务器。
  iii. **服务器配置**：
    - **Nginx 配置示例**：

```
server {
    listen 80;
    server_name yourdomain.com;

    root /path/to/your/dist;
    index index.html;

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

```
- 上述配置中，`listen` 指定监听的端口，`server_name` 指定域名，`root` 指定项目打包后的文件路径，`location` 配置用于处理路由，确保单页应用的路由正常工作。
4. **文件上传**：将本地 `dist` 目录下的所有文件上传到服务器的指定目录。可以使用 FTP、SFTP 等工具进行文件上传。
5. **域名配置**：如果使用自定义域名，需要将域名解析到服务器的 IP 地址。在域名管理平台上添加相应的 DNS 记录。
```

- **案例应用**：将一个使用 Vite 构建的 React 或 Vue 单页应用部署到线上，让用户可以通过域名访问该应用。

手写算法题分类

1. 手撕 lc1026 二叉树最大差值

- **题目描述**：给定二叉树的根节点 `root`，找出存在于不同节点 `A` 和 `B` 之间的最大值 `V`，其中 `V = |A.val - B.val|`，且 `A` 是 `B` 的祖先。
- **原理**：可以使用深度优先搜索（DFS）遍历二叉树，在遍历过程中记录从根节点到当前节点路径上的最大值和最小值，然后计算当前节点值与最大值、最小值的差

值，更新最大差值。

- **代码案例**：

```
class TreeNode {
    constructor(val = 0, left = null, right = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

function maxAncestorDiff(root) {
    let maxDiff = 0;

    function dfs(node, minVal, maxVal) {
        if (!node) return;

        // 更新最大差值
        maxDiff = Math.max(maxDiff, Math.abs(node.val - minVal),
Math.abs(node.val - maxVal));

        // 更新路径上的最大值和最小值
        minVal = Math.min(minVal, node.val);
        maxVal = Math.max(maxVal, node.val);

        // 递归遍历左子树和右子树
        dfs(node.left, minVal, maxVal);
        dfs(node.right, minVal, maxVal);
    }

    dfs(root, root.val, root.val);
    return maxDiff;
}

// 构建二叉树示例
const root = new TreeNode(8);
root.left = new TreeNode(3);
root.right = new TreeNode(10);
root.left.left = new TreeNode(1);
root.left.right = new TreeNode(6);
root.left.right.left = new TreeNode(4);
root.left.right.right = new TreeNode(7);
root.right.right = new TreeNode(14);
root.right.right.left = new TreeNode(13);

const result = maxAncestorDiff(root);
```

```
console.log(result);
```

- **代码细节**：
  - 定义 `TreeNode` 类表示二叉树节点。
  - `maxAncestorDiff` 函数中，使用 `dfs` 函数进行深度优先搜索。`dfs` 函数接收当前节点、路径上的最小值和最大值作为参数。
  - 在 `dfs` 函数中，计算当前节点值与最小值、最大值的差值，更新最大差值。然后更新路径上的最小值和最大值，并递归遍历左子树和右子树。
- **复杂度分析**：
  - **时间复杂度**：$O(n)$，其中 n 是二叉树的节点数，需要遍历每个节点一次。
  - **空间复杂度**：$O(h)$，其中 h 是二叉树的高度，递归调用栈的深度为树的高度。

2. 【代码】实现数组元素偏移

前面在 JavaScript 分类中已经详细介绍过，这里不再赘述。

3. 【代码】 数组拉平（flatten）

前面在 JavaScript 分类中已经详细介绍过，这里不再赘述。

4. 合并区间

前面在 JavaScript 分类中已经详细介绍过，这里不再赘述。

5. 实现一个 call 改变 this 指向

前面在 JavaScript 分类中已经详细介绍过，这里不再赘述。

6. 实现一个 promise.all

前面在 JavaScript 分类中已经详细介绍过，这里不再赘述。

7. 实现 promise any 方法 和 once 方法

前面在 JavaScript 分类中已经详细介绍过，这里不再赘述。

8. 长文章加载优化相关算法实现（预加载、分页并发处理）

预加载实现思路及代码示例

- **原理**：预加载是在用户当前操作之前提前加载可能会用到的资源，以提高后续操作的响应速度。对于长文章，可提前加载后续页面的内容。这里我们假设文章内容存储在一个数组中，每个元素代表一页的内容。
- **代码案例**：

```
// 模拟文章数据
const articlePages = Array.from({ length: 100 }, (_, i) => `Page ${i +
1} content`);

// 预加载函数
function preloadPages(currentPage, preloadCount) {
    const preloadedPages = [];
    for (let i = 1; i <= preloadCount; i++) {
        const nextPageIndex = currentPage + i;
        if (nextPageIndex < articlePages.length) {
            preloadedPages.push(articlePages[nextPageIndex]);
        }
    }
    return preloadedPages;
}

// 当前页面为第 1 页，预加载 3 页
const currentPage = 1;
const preloadCount = 3;
const preloaded = preloadPages(currentPage - 1, preloadCount);
console.log('Preloaded pages:', preloaded);
```

- **代码细节**：
  - `articlePages` 数组模拟了文章的所有页面内容。
  - `preloadPages` 函数接收当前页面索引和预加载的页面数量作为参数。
  - 通过循环，从当前页面的下一页开始，将指定数量的页面内容添加到 `preloadedPages` 数组中，若超出文章总页数则停止。
- **案例应用**：在长文章阅读场景中，当用户阅读到当前页面时，提前加载后续几页的内容，当用户翻页时可以立即显示，减少等待时间。

分页并发处理实现思路及代码示例

- **原理**：分页并发处理是指同时加载多个页面的数据，提高数据加载的效率。可以使用 `Promise.all` 来实现多个页面数据的并发加载。
- **代码案例**：

```
// 模拟异步加载页面数据
```

```
function loadPage(pageIndex) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve(articlePages[pageIndex]);
        }, Math.random() * 1000); // 模拟不同的加载时间
    });
}

// 分页并发加载函数
async function loadPagesConcurrently(pageIndices) {
    const promises = pageIndices.map(index => loadPage(index));
    return Promise.all(promises);
}

// 并发加载第 2、3、4 页
const pageIndicesToLoad = [1, 2, 3];
loadPagesConcurrently(pageIndicesToLoad).then(pages => {
    console.log('Concurrently loaded pages:', pages);
});
```

- **代码细节**：
  - `loadPage` 函数模拟异步加载页面数据，返回一个 `Promise`，在随机时间后解析为对应页面的内容。
  - `loadPagesConcurrently` 函数接收一个页面索引数组，将每个索引对应的 `loadPage` 函数调用封装成 `Promise`，并使用 `Promise.all` 并发执行这些 `Promise`。
- **案例应用**：在长文章列表展示中，需要同时加载多个页面的数据来填充列表，使用分页并发处理可以加快数据加载速度，提升用户体验。

9. 大文件断点续传实现（计算并记录切片的哈希值）

- **原理**：大文件断点续传的核心是将大文件分割成多个切片，分别上传这些切片，并记录每个切片的哈希值。在上传过程中，如果出现中断，下次上传时可以根据已上传切片的哈希值判断哪些切片需要重新上传。
- **代码案例**：

```
// 模拟文件
const largeFile = new Blob(['a'.repeat(1024 * 1024 * 10)], { type:
'text/plain' }); // 10MB 文件
const sliceSize = 1024 * 1024; // 每个切片 1MB
const slices = [];
const hashMap = new Map();
```

```
// 分割文件并计算哈希值
async function splitFileAndCalculateHash() {
    for (let i = 0; i < largeFile.size; i += sliceSize) {
        const slice = largeFile.slice(i, i + sliceSize);
        slices.push(slice);
        const hash = await calculateHash(slice);
        hashMap.set(i, hash);
    }
    return hashMap;
}

// 计算文件切片的哈希值
async function calculateHash(slice) {
    const buffer = await slice.arrayBuffer();
    const hashBuffer = await crypto.subtle.digest('SHA-256', buffer);
    const hashArray = Array.from(new Uint8Array(hashBuffer));
    const hashHex = hashArray.map(b => b.toString(16).padStart(2,
'0')).join('');
    return hashHex;
}

// 模拟上传切片
async function uploadSlices() {
    const uploadedHashes = new Map(); // 模拟已上传切片的哈希值记录
    for (let i = 0; i < slices.length; i++) {
        const start = i * sliceSize;
        const hash = hashMap.get(start);
        if (!uploadedHashes.has(hash)) {
            // 模拟上传操作
            await new Promise(resolve => setTimeout(resolve,
Math.random() * 1000));
            uploadedHashes.set(hash, true);
            console.log(`Uploaded slice starting at ${start} with hash
${hash}`);
        }
    }
}

splitFileAndCalculateHash().then(() => {
    uploadSlices();
});
```

- **代码细节**：
  - `splitFileAndCalculateHash` 函数将大文件分割成多个切片，并调用 `calculateHash` 函数计算每个切片的哈希值，将哈希值存储在 `hashMap` 中。

    - `calculateHash` 函数使用 `crypto.subtle.digest` 方法计算切片的 SHA - 256 哈希值。
    - `uploadSlices` 函数模拟上传切片的过程，通过 `uploadedHashes` 记录已上传切片的哈希值，避免重复上传。
- **案例应用**：在上传大文件（如视频、大型文档）时，网络不稳定或其他原因可能导致上传中断，使用断点续传可以从上次中断的位置继续上传，节省时间和带宽。

10. 前端文件分片（input 接受文件，拿到文件指针，对它进行分片处理，计算哈希值）

- **原理**：前端通过 `<input type="file">` 元素让用户选择文件，获取文件对象后，根据指定的切片大小对文件进行分割，同时计算每个切片的哈希值。
- **代码案例**：

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>File Slicing</title>
</head>

<body>
    <input type="file" id="fileInput">
    <button id="sliceButton">Slice File</button>
    <script>
        const fileInput = document.getElementById('fileInput');
        const sliceButton = document.getElementById('sliceButton');

        sliceButton.addEventListener('click', async () => {
            const file = fileInput.files[0];
            if (!file) {
                alert('Please select a file.');
                return;
            }
            const sliceSize = 1024 * 1024; // 每个切片 1MB
            const slices = [];
            const hashMap = new Map();

            for (let i = 0; i < file.size; i += sliceSize) {
                const slice = file.slice(i, i + sliceSize);
                slices.push(slice);
                const hash = await calculateHash(slice);
```

```
                hashMap.set(i, hash);
                console.log(`Slice starting at ${i} with hash
${hash}`);
            }
        });

        async function calculateHash(slice) {
            const buffer = await slice.arrayBuffer();
            const hashBuffer = await crypto.subtle.digest('SHA-256',
buffer);
            const hashArray = Array.from(new Uint8Array(hashBuffer));
            const hashHex = hashArray.map(b =>
b.toString(16).padStart(2, '0')).join('');
            return hashHex;
        }
    </script>
</body>

</html>
```

- **代码细节**：
  - 通过 `<input type="file">` 元素让用户选择文件，点击按钮时获取文件对象。
  - 按照指定的切片大小对文件进行分割，调用 `calculateHash` 函数计算每个切片的哈希值，并将哈希值存储在 `hashMap` 中。
  - `calculateHash` 函数使用 `crypto.subtle.digest` 方法计算切片的 SHA - 256 哈希值。
- **案例应用**：在需要上传大文件的场景中，前端先对文件进行分片处理并计算哈希值，然后将切片和哈希值信息发送到服务器，服务器可以根据哈希值验证切片的完整性。

11. 数组去重算法

利用 Set 数据结构实现去重

- **原理**：ES6 引入的 `Set` 对象是一种无序且唯一的数据结构，它不允许存储重复的值。因此，可以将数组转换为 `Set`，再将 `Set` 转换回数组，从而实现数组去重。
- **代码案例**：

```
function uniqueArray(arr) {
    return [...new Set(arr)];
}
```

```
const arr = [1, 2, 2, 3, 4, 4, 5];
const uniqueArr = uniqueArray(arr);
console.log(uniqueArr); // 输出: [1, 2, 3, 4, 5]
```

- **代码细节**：
  - `new Set(arr)`：将数组 `arr` 转换为 `Set` 对象，在转换过程中会自动去除重复的值。
  - `[...new Set(arr)]`：使用扩展运算符将 `Set` 对象展开到一个新数组中。
- **复杂度分析**：
  - **时间复杂度**：$O(n)$，其中 n 是数组的长度。因为 `Set` 的插入和查找操作的平均时间复杂度都是 $O(1)$，所以整个去重操作的时间复杂度为 $O(n)$。
  - **空间复杂度**：$O(n)$，因为需要额外的 `Set` 对象来存储数组中的元素。

利用双重循环实现去重

- **原理**：通过双重循环遍历数组，外层循环遍历每个元素，内层循环检查该元素是否已经在前面出现过，如果出现过则跳过，否则将其保留。
- **代码案例**：

```
function uniqueArrayLoop(arr) {
    const result = [];
    for (let i = 0; i < arr.length; i++) {
        let isDuplicate = false;
        for (let j = 0; j < result.length; j++) {
            if (arr[i] === result[j]) {
                isDuplicate = true;
                break;
            }
        }
        if (!isDuplicate) {
            result.push(arr[i]);
        }
    }
    return result;
}

const arrLoop = [1, 2, 2, 3, 4, 4, 5];
const uniqueArrLoop = uniqueArrayLoop(arrLoop);
console.log(uniqueArrLoop); // 输出: [1, 2, 3, 4, 5]
```

- **代码细节**：
  - 外层循环遍历数组 `arr` 中的每个元素。
  - 内层循环遍历结果数组 `result`，检查当前元素是否已经存在于 `result`

中。
    - 如果不存在，则将该元素添加到 `result` 中。
- **复杂度分析**：
    - **时间复杂度**：$O(n^2)$，因为需要使用双重循环遍历数组。
    - **空间复杂度**：$O(n)$，主要用于存储去重后的结果数组。

12. 字符串反转算法

利用数组的 reverse 方法实现字符串反转

- **原理**：将字符串转换为字符数组，使用数组的 `reverse` 方法反转数组元素的顺序，再将反转后的数组转换回字符串。
- **代码案例**：

```
function reverseString(str) {
    return str.split('').reverse().join('');
}

const str = 'hello';
const reversedStr = reverseString(str);
console.log(reversedStr); // 输出: 'olleh'
```

- **代码细节**：
    - `str.split('')`：将字符串 `str` 转换为字符数组。
    - `.reverse()`：调用数组的 `reverse` 方法反转数组元素的顺序。
    - `.join('')`：将反转后的数组转换回字符串。
- **复杂度分析**：
    - **时间复杂度**：$O(n)$，其中 n 是字符串的长度。因为 `split`、`reverse` 和 `join` 操作的时间复杂度都是 $O(n)$。
    - **空间复杂度**：$O(n)$，主要用于存储字符数组。

利用循环实现字符串反转

- **原理**：通过循环从字符串的末尾开始，依次将每个字符添加到一个新字符串中，从而实现字符串反转。
- **代码案例**：

```
function reverseStringLoop(str) {
    let reversed = '';
    for (let i = str.length - 1; i >= 0; i--) {
```

```
        reversed += str[i];
    }
    return reversed;
}

const strLoop = 'hello';
const reversedStrLoop = reverseStringLoop(strLoop);
console.log(reversedStrLoop); // 输出: 'olleh'
```

- **代码细节**：
  - 初始化一个空字符串 `reversed` 用于存储反转后的字符串。
  - 从字符串的最后一个字符开始，依次将字符添加到 `reversed` 中。
- **复杂度分析**：
  - **时间复杂度**：$O(n)$，其中 n 是字符串的长度。因为需要遍历字符串一次。
  - **空间复杂度**：$O(n)$，主要用于存储反转后的字符串。

13. 斐波那契数列生成算法

递归实现

- **原理**：斐波那契数列的定义是：$F(0) = 0$，$F(1) = 1$，$F(n) = F(n - 1) + F(n - 2)$（$n \geq 2$）。可以使用递归的方式根据这个定义来生成斐波那契数列。
- **代码案例**：

```
function fibonacciRecursive(n) {
    if (n === 0) return 0;
    if (n === 1) return 1;
    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
}

const n = 6;
const resultRecursive = fibonacciRecursive(n);
console.log(resultRecursive); // 输出: 8
```

- **代码细节**：
  - 当 n 为 0 或 1 时，直接返回 n。
  - 当 n 大于 1 时，递归调用 `fibonacciRecursive` 函数计算 $F(n - 1)$ 和 $F(n - 2)$ 的值，并将它们相加。
- **复杂度分析**：
  - **时间复杂度**：$O(2^n)$，因为递归调用会产生大量的重复计算，时间复杂度呈指数级增长。

- **空间复杂度**：$O(n)$，主要是递归调用栈的深度。

迭代实现

- **原理**：使用迭代的方式，从 $F(0)$ 和 $F(1)$ 开始，依次计算后续的斐波那契数，避免了递归带来的重复计算问题。
- **代码案例**：

```
function fibonacciIterative(n) {
    if (n === 0) return 0;
    if (n === 1) return 1;
    let a = 0, b = 1;
    for (let i = 2; i <= n; i++) {
        let temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}

const nIterative = 6;
const resultIterative = fibonacciIterative(nIterative);
console.log(resultIterative); // 输出: 8
```

- **代码细节**：
  - 初始化两个变量 `a` 和 `b` 分别为 0 和 1。
  - 通过循环从 2 到 n，依次计算下一个斐波那契数，并更新 `a` 和 `b` 的值。
- **复杂度分析**：
  - **时间复杂度**：$O(n)$，因为只需要遍历一次从 2 到 n 的所有数。
  - **空间复杂度**：$O(1)$，只使用了常数级的额外空间。

啦啦啦

1. Vue的diff算法优化（Vue2和3之间做了什么优化）

原理分析

- **Vue2 的 Diff 算法**：Vue2 采用的是基于虚拟 DOM 的双指针比较算法，使用 `snabbdom` 库实现。它会对新旧虚拟 DOM 树进行深度优先遍历，通过比较新旧节

点的 key、tag、data 等属性来判断节点是否需要更新。在更新子节点时，会采用双指针法，对新旧子节点数组进行比较，找到可复用的节点并进行移动、插入或删除操作。但这种算法在处理大规模列表更新时，由于需要频繁地进行全量比较，性能开销较大。

- **Vue3 的 Diff 算法优化**：Vue3 对 Diff 算法进行了多方面的优化。首先引入了静态标记（PatchFlag），对于模板中的静态节点会标记为静态节点，在 Diff 过程中直接跳过这些节点的比较，大大减少了比较的工作量。其次，在处理动态节点时，采用了快速 Diff 算法，结合最长递增子序列算法，能更高效地找到可复用的节点，减少节点的移动操作，提高了更新效率。

应用场景

- **Vue2**：适用于小型项目或者对性能要求不是特别高的项目，因为其 Diff 算法虽然在处理复杂更新时性能有限，但对于简单的页面更新已经足够。
- **Vue3**：更适合大型项目和需要频繁进行数据更新的场景，如实时数据展示、交互式应用等，其优化后的 Diff 算法能显著提升性能。

代码案例

以下是一个简单的 Vue2 和 Vue3 列表更新的示例，通过对比可以感受两者的差异，但由于 Diff 算法是底层实现，代码中不会直接体现其细节。

Vue2 示例：

```
<template>
  <div>
    <ul>
      <li v-for="item in list" :key="item.id">{{ item.name }}</li>
    </ul>
    <button @click="updateList">Update List</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      list: [
        { id: 1, name: 'Item 1' },
        { id: 2, name: 'Item 2' },
        { id: 3, name: 'Item 3' }
      ]
    };
```

```
  },
  methods: {
    updateList() {
      this.list = [
        { id: 3, name: 'Item 3' },
        { id: 1, name: 'Item 1' },
        { id: 2, name: 'Item 2' }
      ];
    }
  }
};
</script>
```

Vue3 示例：

```
<template>
  <div>
    <ul>
      <li v-for="item in list" :key="item.id">{{ item.name }}</li>
    </ul>
    <button @click="updateList">Update List</button>
  </div>
</template>

<script setup>
import { ref } from 'vue';

const list = ref([
  { id: 1, name: 'Item 1' },
  { id: 2, name: 'Item 2' },
  { id: 3, name: 'Item 3' }
]);

const updateList = () => {
  list.value = [
    { id: 3, name: 'Item 3' },
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' }
  ];
};
</script>
```

解析

在上述代码中，Vue2 和 Vue3 的模板和功能基本相同，都是展示一个列表并提供一个按

钮用于更新列表。但在底层，Vue3 的 Diff 算法会更高效地处理列表的更新，尤其是当列表元素顺序发生变化时，能更快地完成节点的移动和更新操作。

2. 进程之间如何进行通信？说一下应用场景？

原理分析

进程间通信（IPC，Inter - Process Communication）是指在不同进程之间传播或交换信息的机制。常见的进程间通信方式有以下几种：

- **管道（Pipe）**：管道是一种半双工的通信方式，数据只能在一个方向上流动，通常用于父子进程之间的通信。管道本质上是一个内核缓冲区，写入管道的数据会被内核缓存，读取进程可以从管道中读取数据。
- **消息队列（Message Queue）**：消息队列是一种消息的链表，存放在内核中并由消息队列标识符标识。消息队列允许一个或多个进程向它写入或读取消息，进程可以按照一定的规则从消息队列中获取消息，而不需要与其他进程保持同步。
- **共享内存（Shared Memory）**：共享内存是指多个进程可以访问同一块物理内存区域，这是最快的一种 IPC 方式。进程可以直接对共享内存进行读写操作，避免了数据的复制，但需要使用同步机制（如信号量）来避免竞态条件。
- **信号量（Semaphore）**：信号量是一种计数器，用于控制对共享资源的访问。它可以用来实现进程之间的同步和互斥，通过 P（等待）和 V（释放）操作来改变信号量的值。
- **套接字（Socket）**：套接字可以实现不同主机之间的进程通信，也可以用于同一主机上的进程通信。它提供了一种网络编程接口，通过网络协议进行数据传输。

应用场景

- **管道**：常用于 shell 脚本中，例如将一个命令的输出作为另一个命令的输入，如 `ls | grep test`。
- **消息队列**：适用于需要异步通信的场景，如任务调度系统，生产者进程将任务消息放入消息队列，消费者进程从队列中取出任务进行处理。
- **共享内存**：在需要频繁交换大量数据的场景中非常有用，如图形处理、数据库管理系统等，多个进程可以同时访问共享内存中的数据。
- **信号量**：用于解决进程间的同步和互斥问题，如多个进程同时访问一个共享资源时，使用信号量来保证资源的互斥访问。
- **套接字**：广泛应用于网络编程，如 Web 服务器与客户端之间的通信、即时通讯软件等。

代码案例（Python 中使用管道进行进程间通信）

```python
import os

# 创建管道
r, w = os.pipe()

# 创建子进程
pid = os.fork()

if pid == 0:
    # 子进程关闭写端
    os.close(w)
    # 从管道读取数据
    r = os.fdopen(r)
    print(f"Child process received: {r.read()}")
    r.close()
else:
    # 父进程关闭读端
    os.close(r)
    w = os.fdopen(w, 'w')
    # 向管道写入数据
    w.write("Hello from parent process!")
    w.close()
    # 等待子进程结束
    os.wait()
```

解析

在上述代码中，首先使用 `os.pipe()` 创建了一个管道，返回两个文件描述符 `r` 和 `w` 分别用于读取和写入。然后使用 `os.fork()` 创建了一个子进程。父进程关闭读端，向管道写入数据；子进程关闭写端，从管道读取数据。通过这种方式实现了父子进程之间的通信。

3. 前端性能优化

原理分析

前端性能优化的核心目标是提高网页的加载速度和响应速度，提升用户体验。主要从以下几个方面进行优化：

- **压缩代码**：对 HTML、CSS、JavaScript 代码进行压缩，去除不必要的空格、注释和换行符，减少文件大小，从而加快文件的下载速度。
- **合并文件**：将多个 CSS 文件和 JavaScript 文件合并成一个文件，减少浏览器的请求次数，因为每次请求都有一定的开销。

- **图片优化**：选择合适的图片格式（如 JPEG、PNG、WebP），对图片进行压缩，降低图片的分辨率和质量，同时可以采用图片懒加载技术，只在图片进入可视区域时才加载，减少初始加载的资源量。
- **缓存机制**：利用浏览器的缓存机制，设置合理的缓存策略，如使用 `Cache - Control`、`Expires` 等 HTTP 头信息，让浏览器缓存静态资源，避免重复下载。
- **CDN（内容分发网络）**：使用 CDN 来分发静态资源，CDN 节点分布在全球各地，可以将资源缓存到离用户最近的节点，减少数据传输的距离，提高资源的加载速度。
- **优化 DOM 操作**：减少 DOM 操作的次数，因为 DOM 操作会触发浏览器的重排和重绘，影响性能。可以使用文档片段（DocumentFragment）来批量操作 DOM，减少重排和重绘的次数。

应用场景

- **电商网站**：用户在浏览商品列表和详情页时，希望页面能够快速加载，因此需要进行前端性能优化，提高用户的购物体验。
- **新闻资讯网站**：大量的图片和文章内容需要快速加载，以满足用户获取信息的需求，性能优化可以提高用户的留存率。
- **在线游戏**：需要实时响应用户的操作，优化前端性能可以减少游戏的卡顿现象，提高游戏的流畅度。

代码案例（使用 Gulp 进行代码压缩和合并）

```
const gulp = require('gulp');
const uglify = require('gulp-uglify');
const concat = require('gulp-concat');
const cssnano = require('gulp-cssnano');

// 压缩和合并 JavaScript 文件
gulp.task('scripts', function () {
  return gulp.src('src/js/*.js')
    .pipe(concat('all.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist/js'));
});

// 压缩和合并 CSS 文件
gulp.task('styles', function () {
  return gulp.src('src/css/*.css')
    .pipe(concat('all.css'))
    .pipe(cssnano())
    .pipe(gulp.dest('dist/css'));
```

```
});

// 默认任务
gulp.task('default', gulp.parallel('scripts', 'styles'));
```

解析

上述代码使用 Gulp 构建工具，通过 `gulp-uglify` 插件对 JavaScript 文件进行压缩，`gulp-concat` 插件将多个 JavaScript 文件合并成一个文件，`gulp-cssnano` 插件对 CSS 文件进行压缩和合并。最后定义了一个默认任务，同时执行 JavaScript 和 CSS 的处理任务。

4. Vue 路由的实现原理

原理分析

Vue 路由（Vue Router）是 Vue.js 官方的路由管理器，它的实现基于单页面应用（SPA）的原理。主要通过监听浏览器的 URL 变化，根据配置的路由规则匹配相应的组件，并将组件渲染到指定的路由出口（`<router - view>`）中。

Vue Router 有两种模式：Hash 模式和 History 模式。

- **Hash 模式**：URL 中使用 `#` 符号来分隔路径，如 `http://example.com/#/home`。`#` 后面的内容不会发送到服务器，浏览器只会根据 `#` 后面的路径进行前端路由匹配。当 `#` 后面的路径发生变化时，会触发 `hashchange` 事件，Vue Router 会根据新的路径匹配相应的组件进行渲染。
- **History 模式**：使用 HTML5 的 `History API` 来管理浏览器的历史记录，URL 看起来像正常的路径，如 `http://example.com/home`。当用户点击链接或调用 `history.pushState` 等方法时，Vue Router 会根据新的路径匹配相应的组件进行渲染。同时，需要服务器端进行相应的配置，确保所有的请求都返回同一个 HTML 文件。

应用场景

- **单页面应用（SPA）**：如企业官网、后台管理系统等，通过 Vue Router 可以实现页面的切换和导航，给用户带来流畅的交互体验。
- **多页面应用（MPA）**：在一些复杂的多页面应用中，也可以使用 Vue Router 来管理局部的路由，实现局部页面的动态加载和切换。

代码案例

```
<template>
  <div id="app">
    <router-link to="/">Home</router-link>
    <router-link to="/about">About</router-link>
    <router-view></router-view>
  </div>
</template>

<script>
import Vue from 'vue';
import VueRouter from 'vue-router';
import Home from './views/Home.vue';
import About from './views/About.vue';

Vue.use(VueRouter);

const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home
  },
  {
    path: '/about',
    name: 'About',
    component: About
  }
];

const router = new VueRouter({
  mode: 'hash',
  routes
});

new Vue({
  router,
  render: h => h('div', { id: 'app' })
}).$mount('#app');
</script>
```

解析

上述代码定义了一个简单的 Vue 应用，使用 Vue Router 实现了两个路由：`/` 和 `/about`。通过 `<router-link>` 组件创建导航链接，点击链接时会改变 URL 的路径。`<router-view>` 是路由出口，匹配到的组件会渲染到这个位置。在 `main.js`

中，首先引入 `VueRouter` 并使用 `Vue.use(VueRouter)` 安装插件。然后定义路由规则数组 `routes`，每个规则包含路径 `path`、路由名称 `name` 和对应的组件 `component`。接着创建 `VueRouter` 实例，指定路由模式为 `hash`，并将路由规则传入。最后将路由实例挂载到 Vue 实例上。

5. HTTP 状态码

原理分析

HTTP 状态码是服务器返回给客户端的三位数字代码，用于表示 HTTP 请求的结果。状态码的第一个数字定义了响应的类别，后两个数字则提供了更具体的信息。常见的状态码类别如下：

- **1xx（信息性状态码）**：表示临时响应，通常用于表示请求已接收，继续处理。
- **2xx（成功状态码）**：表示请求成功。
- **3xx（重定向状态码）**：表示需要客户端采取进一步的操作才能完成请求，通常用于重定向。
- **4xx（客户端错误状态码）**：表示客户端的请求有错误，无法被服务器处理。
- **5xx（服务器错误状态码）**：表示服务器在处理请求时发生了错误。

应用场景

- **200 OK**：最常见的成功状态码，当客户端请求资源成功时，服务器返回此状态码。例如，浏览器请求一个 HTML 页面，服务器成功返回页面内容时，状态码为 200。
- **301 Moved Permanently**：表示请求的资源已永久移动到新的 URL。常用于网站域名变更或页面结构调整时，告诉搜索引擎和浏览器该页面已永久转移到新地址。
- **404 Not Found**：表示请求的资源不存在。当用户访问一个不存在的页面时，服务器会返回此状态码。
- **500 Internal Server Error**：表示服务器内部发生了错误，无法完成请求。可能是服务器代码出现异常、数据库连接失败等原因导致。

代码案例（使用 Node.js 实现简单的 HTTP 服务器并返回不同状态码）

```
const http = require('http');

const server = http.createServer((req, res) => {
    if (req.url === '/') {
        res.statusCode = 200;
```

```
        res.setHeader('Content-Type', 'text/plain');
        res.end('Welcome to the home page!');
    } else if (req.url === '/about') {
        res.statusCode = 301;
        res.setHeader('Location', '/new - about');
        res.end();
    } else if (req.url === '/new - about') {
        res.statusCode = 200;
        res.setHeader('Content-Type', 'text/plain');
        res.end('This is the new about page!');
    } else {
        res.statusCode = 404;
        res.setHeader('Content-Type', 'text/plain');
        res.end('Page not found!');
    }
});

server.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

解析

上述代码创建了一个简单的 HTTP 服务器，根据不同的请求 URL 返回不同的状态码和响应内容。当请求根路径 `/` 时，返回状态码 200 和欢迎信息；当请求 `/about` 时，返回状态码 301 并设置重定向到 `/new - about`；当请求 `/new - about` 时，返回状态码 200 和新的关于页面信息；当请求其他不存在的路径时，返回状态码 404 和页面未找到信息。

6. WebSocket 通信原理

原理分析

WebSocket 是一种在单个 TCP 连接上进行全双工通信的协议，它允许浏览器和服务器之间进行实时通信。其通信原理主要分为以下几个阶段：

- **握手阶段**：客户端向服务器发送一个 HTTP 请求，请求头中包含 `Upgrade: websocket` 和 `Connection: Upgrade` 等信息，表示要升级为 WebSocket 连接。服务器收到请求后，如果支持 WebSocket 协议，会返回一个状态码为 101 的响应，表示同意升级连接。
- **建立连接**：握手成功后，TCP 连接保持打开，客户端和服务器可以在这个连接上进行双向通信。

- **数据传输**：客户端和服务器可以随时向对方发送数据，数据以帧的形式传输。WebSocket 协议定义了不同类型的帧，如文本帧、二进制帧等。
- **关闭连接**：当一方需要关闭连接时，会发送一个关闭帧给对方。对方收到关闭帧后，会发送一个确认关闭帧，然后双方关闭 TCP 连接。

应用场景

- **实时聊天应用**：如在线客服、即时通讯软件等，通过 WebSocket 可以实现消息的实时发送和接收，提供流畅的聊天体验。
- **实时数据展示**：如股票行情、实时监控系统等，服务器可以实时将最新的数据推送给客户端，客户端及时更新页面显示。
- **多人游戏**：在多人在线游戏中，通过 WebSocket 可以实现玩家之间的实时交互，如同步游戏状态、传递玩家操作等。

代码案例（使用 Node.js 和浏览器示例）

- **服务器端（使用 Node.js 的 ws 库）**：

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', function connection(ws) {
    ws.on('message', function incoming(message) {
        console.log('received: %s', message);
        ws.send('Server received: ' + message);
    });
});
```

- **客户端（浏览器端）**：

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>WebSocket Example</title>
</head>

<body>
    <button id="sendButton">Send Message</button>
```

```
    <script>
        const socket = new WebSocket('ws://localhost:8080');

        socket.onopen = function () {
            console.log('Connected to the server');
        };

        socket.onmessage = function (event) {
            console.log('Received from server: ', event.data);
        };

        socket.onclose = function () {
            console.log('Disconnected from the server');
        };

        const sendButton = document.getElementById('sendButton');
        sendButton.addEventListener('click', function () {
            socket.send('Hello, server!');
        });
    </script>
</body>

</html>
```

解析

服务器端使用 `ws` 库创建一个 WebSocket 服务器，监听 8080 端口。当有客户端连接时，监听客户端发送的消息，并将消息原样返回给客户端。客户端使用 `WebSocket` 构造函数创建一个 WebSocket 实例，连接到服务器。监听 `open`、`message` 和 `close` 事件，分别处理连接建立、接收消息和连接关闭的情况。点击按钮时，向服务器发送消息。

7. DOM、BOM 和 BOM 的 API

原理分析

- **DOM（文档对象模型）**：是 HTML 文档的树形结构表示，将 HTML 文档解析为一个由节点组成的树，每个节点可以是元素节点、文本节点等。通过 DOM，JavaScript 可以动态地操作 HTML 文档的内容、结构和样式。
- **BOM（浏览器对象模型）**：是浏览器窗口的对象模型，提供了与浏览器窗口相关的操作接口，如窗口大小调整、导航、弹出对话框等。BOM 包含了一些全局对象，如 `window`、`location`、`history` 等。

应用场景

- **DOM**：在网页交互开发中广泛应用，如动态修改页面内容、添加或删除元素、处理用户事件等。例如，实现一个动态菜单，当用户点击菜单项时，通过 DOM 操作显示相应的子菜单。
- **BOM**：用于实现与浏览器窗口相关的功能，如页面跳转、弹出新窗口、获取浏览器信息等。例如，在页面中添加一个按钮，点击按钮时打开一个新的浏览器窗口。

代码案例

- **DOM 操作示例**：

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>DOM Example</title>
</head>

<body>
    <p id="myParagraph">This is a paragraph.</p>
    <button id="changeTextButton">Change Text</button>
    <script>
        const paragraph = document.getElementById('myParagraph');
        const button = document.getElementById('changeTextButton');

        button.addEventListener('click', function () {
            paragraph.textContent = 'The text has been changed!';
        });
    </script>
</body>

</html>
```

- **BOM 操作示例**：

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

<head>
    <meta charset="UTF-8">
```

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

<body>
    <button id="openNewWindowButton">Open New Window</button>
    <script>
        const openButton =
document.getElementById('openNewWindowButton');

        openButton.addEventListener('click', function () {
            window.open('https://www.example.com', '_blank');
        });
    </script>
</body>

</html>
```

解析

在 DOM 操作示例中，通过 `document.getElementById` 方法获取 `p` 元素和按钮元素，然后给按钮添加点击事件监听器，当点击按钮时，使用 `textContent` 属性修改 `p` 元素的文本内容。在 BOM 操作示例中，通过 `document.getElementById` 方法获取按钮元素，给按钮添加点击事件监听器，当点击按钮时，使用 `window.open` 方法打开一个新的浏览器窗口。

8. 浏览器的组成

原理分析

现代浏览器主要由多个组件组成，每个组件负责不同的功能，协同工作以实现网页的加载和渲染等功能。主要组件包括：

- **用户界面**：包括地址栏、书签栏、标签页等，用于用户与浏览器进行交互。
- **浏览器引擎**：负责协调和管理其他组件的工作，如解析 HTML、CSS 等资源，将渲染结果传递给渲染引擎。
- **渲染引擎**：负责解析 HTML 和 CSS，构建 DOM 树和 CSSOM 树，合并为渲染树，进行布局和绘制，将页面内容显示在屏幕上。不同的浏览器使用不同的渲染引擎，如 Chrome 使用 Blink 引擎，Firefox 使用 Gecko 引擎。
- **网络模块**：负责处理网络请求和响应，包括 DNS 解析、TCP 连接建立、HTTP 请求发送和响应接收等。

- **JavaScript 引擎**：负责执行 JavaScript 代码，如 Chrome 的 V8 引擎。它将 JavaScript 代码编译为机器码，提高执行效率。
- **数据存储模块**：负责管理浏览器的本地存储，如 `localStorage`、`sessionStorage`、`IndexedDB` 等，用于保存用户的浏览数据、缓存等。
- **安全模块**：负责保障浏览器的安全，如防止跨站脚本攻击（XSS）、跨站请求伪造（CSRF）等，对网页的内容和请求进行安全检查。

应用场景

- **用户界面**：方便用户输入 URL、管理书签、切换标签页等操作，提供良好的用户体验。
- **渲染引擎**：确保网页能够正确地解析和显示，处理不同的 HTML 和 CSS 标准。
- **网络模块**：实现网页资源的下载和更新，保证数据的传输稳定和高效。
- **JavaScript 引擎**：支持网页中的动态交互和复杂功能，如表单验证、动画效果等。
- **数据存储模块**：用于保存用户的登录状态、个性化设置、离线缓存等信息，提高用户的使用便利性。
- **安全模块**：保护用户的隐私和数据安全，防止恶意网站的攻击。

代码案例

虽然浏览器的组成是底层实现，无法直接通过代码展示，但可以通过一些 JavaScript 代码来演示部分功能，如使用 `localStorage` 进行数据存储：

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Browser Storage Example</title>
</head>

<body>
    <input type="text" id="inputText">
    <button id="saveButton">Save to Local Storage</button>
    <button id="loadButton">Load from Local Storage</button>
    <p id="output"></p>
    <script>
        const input = document.getElementById('inputText');
```

```
        const saveButton = document.getElementById('saveButton');
        const loadButton = document.getElementById('loadButton');
        const output = document.getElementById('output');

        saveButton.addEventListener('click', function () {
            const text = input.value;
            localStorage.setItem('savedText', text);
            output.textContent = 'Text saved to local storage.';
        });

        loadButton.addEventListener('click', function () {
            const savedText = localStorage.getItem('savedText');
            if (savedText) {
                output.textContent = 'Loaded text: ' + savedText;
            } else {
                output.textContent = 'No text found in local storage.';
            }
        });
    </script>
</body>

</html>
```

解析

此代码示例主要展示了浏览器数据存储模块中 `localStorage` 的使用。具体如下：

- 首先通过 `document.getElementById` 获取输入框、保存按钮、加载按钮和用于显示输出信息的段落元素。
- 给保存按钮添加点击事件监听器，当点击保存按钮时，获取输入框中的文本内容，使用 `localStorage.setItem` 方法将文本保存到 `localStorage` 中，并在输出段落显示保存成功的信息。
- 给加载按钮添加点击事件监听器，当点击加载按钮时，使用 `localStorage.getItem` 方法从 `localStorage` 中获取之前保存的文本。如果能获取到文本，则在输出段落显示加载的文本；如果没有找到保存的文本，则显示相应提示信息。这体现了浏览器数据存储模块对用户数据的存储和读取功能，方便在不同页面或会话间保留用户输入的信息。