2024前端高频—手写代码篇

1.js数组去重

在JavaScript中，有多种方法可以实现数组去重。以下是其中的几种常见方法：

1\. 使用 Set

Set 是 JavaScript 中一个内建的对象，它只允许存储唯一的值。利用这一特性，我们可以轻松实现数组去重。

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

const array = [1, 2, 3, 4, 4, 3, 2, 1]
const unique = uniqueArray(array)
console.log(unique)
```

2\. 使用 filter

你也可以使用数组的 `filter` 方法来实现去重。这个方法会遍历数组，并对每个元素执行一个测试函数，只有使测试函数返回 `true` 的元素才会被保留在新数组中。

```
function uniqueArray(arr) {
  return arr.filter((item, index) => arr.indexOf(item) === in
dex);
}

const array = [1, 2, 3, 4, 4, 3, 2, 1];
const unique = uniqueArray(array);
```

```
console.log(unique);
```

注意：这个方法在处理大数组时可能效率不高，因为它需要对每个元素都调用 `indexOf` 方法，而这个方法本身的时间复杂度就是 O(n)。

3\. 使用 reduce

`reduce` 方法也可以用来实现数组去重。这个方法会遍历数组，并将每个元素归并成一个单一的结果。你可以利用这个结果来创建一个新数组，只包含唯一的元素。

```
function uniqueArray(arr) {
  return arr.reduce((accumulator, current) => {
    if (!accumulator.includes(current)) {
      accumulator.push(current)
    }
    return accumulator
  }, [])
}

const array = [1, 2, 3, 4, 4, 3, 2, 1]
const unique = uniqueArray(array)
console.log(unique)
```

4\. 使用新特性（如 nullish coalescing 和可选链）

如果你在使用较新的JavaScript版本（ECMAScript 2020 或更高），你可以使用一些新特性来更简洁地实现数组去重。但这并不一定会提高性能，只是代码可能更简洁。

例如，结合 `Set` 和展开语法：

```
const array = [1, 2, 3, 4, 4, 3, 2, 1]
const unique = [...new Set(array)]
console.log(unique)
```

以上就是几种常见的在JavaScript中实现数组去重的方法。你可以根据自己的需求和喜好来选择使用哪一种。

2.js将数字每千分位用逗号隔开

在JavaScript中，你可以使用`toLocaleString`方法将数字格式化为带有千分位逗号的字符串。这个方法会根据你所在地区的本地化设置来格式化数字。

以下是一个简单的例子：

```
let num = 1234567.89
let formattedNum = num.toLocaleString()
console.log(formattedNum)
```

在这个例子中，`toLocaleString`方法会自动在数字的千分位处添加逗号。注意，输出的格式可能会因你的本地化设置而异。

如果你想要确保输出的格式不受本地化设置的影响，你可以为`toLocaleString`方法提供一个选项对象，明确指定你想要的语言环境（locale）。例如，如果你想要使用美国英语的格式（即使用逗号作为千分位分隔符，点作为小数点），你可以这样做：

```
let num = 1234567.89
let formattedNum = num.toLocaleString('en-US')
console.log(formattedNum)
```

在这个例子中，无论你的本地化设置是什么，输出的格式都会是美国英语的格式。

3.手写防抖节流

在JavaScript中，防抖（debounce）和节流（throttle）是两种常用的优化高频触发事件的策略。它们通常用于窗口大小调整、滚动、输入等场景，以减少事件处理函数的执行频率，提高性能。

防抖（Debounce）

防抖是指事件被触发后，n秒内函数只能执行一次，如果在这n秒内又被触发，则重新计算执行时间。

下面是一个简单的防抖函数实现：

```
function debounce(func, wait) {
  let timeout;
  return function() {
    const context = this;
    const args = arguments;
    clearTimeout(timeout);
    timeout = setTimeout(function() {
      func.apply(context, args);
    }, wait);
  };
}


const myEfficientFn = debounce(function() {

}, 250);

window.addEventListener('resize', myEfficientFn);
```

节流（Throttle）

节流是指连续触发事件，但在n秒内只执行一次函数。

下面是一个简单的节流函数实现：

```
function throttle(func, limit) {
  let inThrottle;
  return function() {
    const context = this;
    const args = arguments;
    if (!inThrottle) {
      func.apply(context, args);
      inThrottle = true;
```

```
      setTimeout(function() {
        return inThrottle = false;
      }, limit);
    }
  };
}


const myEfficientFn = throttle(function() {

}, 250);

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

注意事项

- 这两个函数都返回一个新的函数，这个新函数会包装传入的原始函数，并根据防抖或节流的逻辑来调用它。
- 防抖和节流的区别在于，防抖是在事件触发后等待一段时间再执行，而节流是确保事件触发后的一段时间内只执行一次。
- 这两个函数都可以接受任意数量的参数，并将它们传递给原始函数。
- 在实际应用中，你可能需要根据你的需求对这两个函数进行扩展或调整。

4.手写promise

在JavaScript中，`Promise` 是一个用于处理异步操作的对象。它代表了一个可能现在、将来或永远不会完成的异步操作及其结果值。下面是一个简化的 `Promise` 实现示例，只包括最基本的功能：

```
function MyPromise(executor) {
  this.status = 'pending'
  this.value = undefined
  this.reason = undefined
  this.onFulfilledCallbacks = []
```

```
  this.onRejectedCallbacks = []

  // 立即执行executor函数，传入resolve和reject函数
  const resolve = (value) => {
    if (this.status !== 'pending') return
    this.status = 'fulfilled'
    this.value = value
    this.onFulfilledCallbacks.forEach(callback => callback())
  }

  const reject = (reason) => {
    if (this.status !== 'pending') return
    this.status = 'rejected'
    this.reason = reason
    this.onRejectedCallbacks.forEach(callback => callback())
  }

  // 捕获executor函数中抛出的异常
  try {
    executor(resolve, reject)
  } catch (error) {
    reject(error)
  }
}

// then方法，用于指定Promise成功或失败时要执行的回调函数
MyPromise.prototype.then = function(onFulfilled, onRejected)
{
  const promise2 = new MyPromise((resolve, reject) => {
    // 处理Promise状态为fulfilled的情况
    if (this.status === 'fulfilled') {
      setTimeout(() => {
        try {
          const result = onFulfilled(this.value)
          resolve(result)
        } catch (error) {
```

```
          reject(error)
        }
      }, 0)
    }
    // 处理Promise状态为rejected的情况
    else if (this.status === 'rejected') {
      setTimeout(() => {
        try {
          const result = onRejected(this.reason)
          resolve(result)
        } catch (error) {
          reject(error)
        }
      }, 0)
    }
    // 如果Promise状态还为pending，则将其回调函数加入队列中
    else {
      this.onFulfilledCallbacks.push(() => {
        setTimeout(() => {
          try {
            const result = onFulfilled(this.value)
            resolve(result)
          } catch (error) {
            reject(error)
          }
        }, 0)
      })
      this.onRejectedCallbacks.push(() => {
        setTimeout(() => {
          try {
            const result = onRejected(this.reason)
            resolve(result)
          } catch (error) {
            reject(error)
          }
        }, 0)
```

```
      })
    }
  })
  return promise2
}

// 静态方法resolve，用于将现有值转换为Promise对象
MyPromise.resolve = function(value) {
  return new MyPromise(resolve => {
    resolve(value)
  })
}

// 静态方法reject，用于返回一个带有拒绝理由的Promise对象
MyPromise.reject = function(reason) {
  return new MyPromise((_, reject) => {
    reject(reason)
  })
}

// 使用示例
const promise = new MyPromise((resolve, reject) => {
  setTimeout(() => {
    resolve('成功!')
  }, 1000)
})

promise.then(
  value => console.log(value), // 输出 '成功!'
  reason => console.log(reason) // 不会执行
)
```

这个简化的 `MyPromise` 实现包括了以下基本功能：

- `executor` 函数接受 `resolve` 和 `reject` 两个参数，并立即执行。

- `resolve` 和 `reject` 函数用于改变 `Promise` 的状态，并触发相应的回调函数。
- `then` 方法用于指定 `Promise` 成功或失败时要执行的回调函数，并返回一个新的 `Promise` 对象。
- `MyPromise.resolve` 和 `MyPromise.reject` 是两个静态方法，用于创建已经解决或拒绝的 `Promise` 对象。

请注意，这个实现省略了很多特性，比如链式调用、错误冒泡、`catch` 方法、`finally` 方法、以及微任务队列的处理等。在真实的项目中，通常会使用原生的 `Promise` 对象，因为它已经由JavaScript引擎进行了优化，并且提供了完整的API支持。

5.手写深浅拷贝

在JavaScript中，浅拷贝和深拷贝是处理对象和数据结构时常见的概念。浅拷贝只会复制对象的顶层属性和值，如果属性值是对象或数组，那么它实际上只是复制了引用，而不是真正的对象。而深拷贝则会递归地复制对象的所有层级，确保所有的对象或数组都被真正复制，而不是仅复制引用。

下面是浅拷贝和深拷贝的手写实现：

浅拷贝（Shallow Copy）

浅拷贝可以通过扩展运算符（`...`）或者 `Object.assign()` 方法来实现。这里提供一个使用 `Object.assign()` 的示例：

```
function shallowCopy(obj) {
  if (typeof obj !== 'object' || obj === null) {
    return obj;
  }
  return Object.assign({}, obj);
}

const original = { a: 1, b: { c: 2 } };
const copied = shallowCopy(original);
console.log(copied);
```

深拷贝（Deep Copy）

深拷贝的实现则相对复杂一些，因为需要递归地处理对象的所有属性。下面是一个简单的深拷贝实现，只处理了对象和数组的情况：

```
function deepCopy(obj, hash = new WeakMap()) {
  if (typeof obj !== 'object' || obj === null) {
    return obj;
  }


  if (obj instanceof Date) {
    return new Date(obj);
  }
  if (obj instanceof RegExp) {
    return new RegExp(obj);
  }


  if (hash.has(obj)) {
    return hash.get(obj);
  }

  let newObj = Array.isArray(obj) ? [] : {};
  hash.set(obj, newObj);

  for (let key in obj) {
    if (obj.hasOwnProperty(key)) {
      newObj[key] = deepCopy(obj[key], hash);
    }
  }

  return newObj;
}

const original = { a: 1, b: { c: 2 } };
const copied = deepCopy(original);
console.log(copied);
```

```
console.log(original === copied);
console.log(original.b === copied.b);
```

这个深拷贝函数还使用了`WeakMap`来存储已经复制过的对象，以避免无限递归和循环引用的问题。如果对象中存在循环引用，即某个对象的属性直接或间接地引用了该对象本身，使用`WeakMap`可以有效地避免这个问题。

请注意，这个深拷贝函数并不完整，它没有处理函数、Symbol、Error等特殊对象，也没有处理原型链、getter/setter等情况。在实际应用中，你可能需要使用更健壮的库，如`lodash`的`_.cloneDeep()`方法，或者实现更完整的深拷贝逻辑。

6.手写 new 操作符

在 JavaScript 中，`new` 操作符用于创建一个用户自定义的对象类型的实例或具有构造函数的内置对象的实例。`new` 操作符执行以下步骤：

1. 创建一个新的空对象。
2. 将这个新对象的内部原型链接到构造函数的 `prototype` 对象。
3. 将这个新对象作为 `this` 上下文。
4. 如果该函数没有返回其他对象，那么返回 `this`。

下面是一个简单的手写 `new` 操作符的示例：

```
function myNew(constructor, ...args) {
  const obj = {};

  obj.__proto__ = constructor.prototype;

  const result = constructor.apply(obj, args);

  return result instanceof Object ? result : obj;
}
```

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name} and I'm ${this.
age} years old.`);
};


const person = myNew(Person, 'Alice', 30);

console.log(person.name);
console.log(person.age);
person.greet();
```

请注意，这个手写 `new` 操作符的示例中使用了 `__proto__` 属性来设置原型链。然而，`__proto__` 属性并不是标准的一部分，虽然大多数现代浏览器都支持它，但使用 `Object.create()` 或 `Object.setPrototypeOf()` 是更推荐的方式来设置对象的原型。

另外，这个示例没有处理一些边界情况，例如构造函数返回非对象值的情况（这在 `new` 操作符中是允许的）。在实际应用中，你可能需要更全面地处理这些情况。

最后，请记住，尽管手写 `new` 操作符有助于理解其背后的工作原理，但在实际编程中，你应该使用原生的 `new` 操作符，因为它已经被 JavaScript 引擎优化过，并且被广泛接受和使用。

7.函数柯里化的实现

函数柯里化（Currying）是一种在 JavaScript 中处理函数的技术，它可以将一个接受多个参数的函数转换成一系列使用一个参数的函数。这个转换后的函数链中的每一个函数都返回下一个函数，直到最后一个函数返回最终的结果。

以下是一个简单的 JavaScript 函数柯里化的实现：

```
function curry(fn) {
    if (typeof fn !== 'function') {
        throw new Error('curry() requires a function');
    }

    return function curried(...args) {
        if (args.length >= fn.length) {
            return fn.apply(this, args);
        } else {
            return function(...args2) {
                return curried.apply(this, args.concat(args
2));
            }
        }
    };
}


function sum(a, b, c) {
    return a + b + c;
}

const curriedSum = curry(sum);

console.log(curriedSum(1)(2)(3));
console.log(curriedSum(1, 2)(3));
```

在这个例子中，`curry` 函数接受一个函数 `fn` 作为参数，并返回一个新的函数 `curried`。`curried` 函数会检查是否已经提供了足够的参数来调用原始函数 `fn`。如果是，则直接调用 `fn` 并返回结果。否则，它返回一个新的函数，该函数接受剩余的参数，并将它们与已经收集的参数合并，然后递归调用 `curried`。这个过程会一直持续到收集到足够的参数为止。

注意，这个函数柯里化的实现假设了原始函数 `fn` 的参数数量是固定的。如果 `fn` 接受可变数量的参数，那么这个实现可能需要进行相应的调整。

8.实现AJAX请求，使用Promise封装AJAX请求

在JavaScript中，我们可以使用Promise来封装AJAX请求，使得异步操作更加易于管理和理解。以下是一个使用Promise封装AJAX请求的基本示例：

```
function ajaxRequest(url, method = 'GET', data = null) {
    return new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest();
        xhr.open(method, url, true);


        if (method === 'POST') {
            xhr.setRequestHeader('Content-Type', 'applicatio
n/x-www-form-urlencoded');
        }

        xhr.onload = function() {
            if (this.status === 200) {
                resolve(this.responseText);
            } else {
                reject(new Error(this.statusText));
            }
        };

        xhr.onerror = function() {
            reject(new Error('Network Error'));
        };

        if (data) {
            xhr.send(data);
        } else {
            xhr.send();
        }
    });
}
```

```
ajaxRequest('<https://api.example.com/data>', 'GET')
    .then(response => {
        console.log(response);
    })
    .catch(error => {
        console.error('Error:', error);
    });
```

在这个示例中，`ajaxRequest`函数接受一个URL、一个HTTP方法（默认为GET）和一个数据对象作为参数。它返回一个Promise对象，该对象在请求成功时解析为响应文本，或在请求失败时拒绝为错误对象。

你可以通过调用`.then()`方法来处理解析的值，通过调用`.catch()`方法来处理错误。这使得异步操作更加直观和易于管理。

注意：这个示例使用了原生的`XMLHttpRequest`对象来发送请求。在实际项目中，你可能会更倾向于使用更现代、更易于使用的库，如`fetch` API或`axios`，它们也提供了Promise支持。

9.交换a,b的值，不能用临时变量

在JavaScript中，如果你想要交换两个变量的值而不使用临时变量，你可以使用加法（如果它们是数字）或者解构赋值（对于任何类型）。这里有两种方法：

方法一：使用加法（仅限于数字）

```
let a = 5
let b = 10

a = a + b
b = a - b
a = a - b

console.log(a)
console.log(b)
```

方法二：使用解构赋值

解构赋值是ES6中引入的一种语法，它可以用来从数组或对象中提取数据，然后将其赋值给不同的变量。对于交换两个变量的值，解构赋值非常有用。

```
let a = 5
let b = 10

[a, b] = [b, a]

console.log(a)
console.log(b)
```

解构赋值在这里非常直观和简洁，因为它直接创建了新的变量绑定，然后将原来的`b`的值赋给`a`，原来的`a`的值赋给`b`。这是交换两个变量值而不使用临时变量的推荐方法。

10\. 实现数组元素求和

在JavaScript中，你可以使用多种方式来实现数组元素的求和。以下是其中的一些方法：

方法一：使用`for`循环

```
function sumArray(arr) {
    let sum = 0
    for (let i = 0
        sum += arr[i]
    }
    return sum
}

let arr = [1, 2, 3, 4, 5]
console.log(sumArray(arr))
```

方法二：使用`Array.prototype.reduce`方法

`reduce`方法是JavaScript数组的一个内置方法，用于将数组中的所有元素应用到一个函数上，并将其减少为单个输出值。

```
function sumArray(arr) {
    return arr.reduce((accumulator, currentValue) => accumula
tor + currentValue, 0);
}

let arr = [1, 2, 3, 4, 5];
console.log(sumArray(arr));
```

在上面的代码中，`reduce`方法接受一个回调函数和一个初始值（在这里是0）作为参数。回调函数接受两个参数：累加器（accumulator）和当前值（currentValue）。累加器的初始值是`reduce`方法的第二个参数，然后回调函数返回的新值会作为下一次调用的累加器的值。在每次调用中，累加器的值都会与数组中的下一个元素相加，最后返回总和。

这两种方法都可以有效地计算数组元素的和。在实际应用中，你可能会更倾向于使用`reduce`方法，因为它更简洁且更易于理解。

11\. 实现数组的扁平化

在JavaScript中，数组扁平化就是将一个嵌套数组转换为一个一维数组。这可以通过多种方法来实现，下面我将提供几种常见的方法：

方法一：使用递归

```
function flattenArray(arr) {
    let result = []
    for (let i = 0
        if (Array.isArray(arr[i])) {
            result = result.concat(flattenArray(arr[i]))
        } else {
            result.push(arr[i])
        }
    }
    return result
}
```

```
let nestedArray = [1, [2, [3, [4]], 5]]
console.log(flattenArray(nestedArray))
```

方法二：使用`reduce`方法

```
function flattenArray(arr) {
    return arr.reduce((acc, val) => Array.isArray(val) ? [...
acc, ...flattenArray(val)] : [...acc, val], []);
}

let nestedArray = [1, [2, [3, [4]], 5]];
console.log(flattenArray(nestedArray));
```

方法三：使用扩展运算符（ES6）

```
function flattenArray(arr) {
    while (arr.some(item => Array.isArray(item))) {
        arr = [].concat(...arr)
    }
    return arr
}

let nestedArray = [1, [2, [3, [4]], 5]]
console.log(flattenArray(nestedArray))
```

方法四：使用`flat`方法（ES10）

从ES10开始，JavaScript引入了`flat`方法，它可以直接用来扁平化数组。`flat`方法还可以接受一个可选的参数，表示要展开的嵌套层数。

```
let nestedArray = [1, [2, [3, [4]], 5]]
console.log(nestedArray.flat(Infinity))
```

在上面的代码中，`Infinity`用作`flat`方法的参数，意味着无论嵌套多少层，都会展开到一维数组。

在这些方法中，`flat`方法是最简洁和最容易理解的，但是需要注意的是，它可能在一些较旧的浏览器中不被支持。如果需要考虑兼容性，你可能需要使用其他方法，或者结合polyfill来提供对`flat`方法的支持。

12\. 实现 add(1)(2)(3)

在JavaScript中，你可以通过创建一个返回函数的函数来实现这种链式调用的模式。这种模式通常被称为柯里化（Currying）的一种形式，尽管在这个例子中，我们并没有减少参数的数量，而是延迟了它们的处理。

以下是一个简单的实现方式：

```
function add() {
    let sum = 0;

    function innerAdd(num) {
        sum += num;
        return innerAdd;
    }

    innerAdd.toString = function() {
        return sum;
    };

    return innerAdd;
}

console.log(add(1)(2)(3));
```

在这个例子中，`add` 函数返回了一个名为 `innerAdd` 的内部函数。`innerAdd` 函数接受一个数字参数，将其加到 `sum` 变量上，然后返回自己。因此，你可以连续调用 `add(1)(2)(3)`，每次调用都会将新的数字添加到 `sum` 中。

Power Human With Education

需要注意的是，由于 `innerAdd` 是一个函数，如果你直接打印 `add(1)(2)(3)`，它将输出函数本身而不是结果。为了解决这个问题，我们覆盖了 `innerAdd` 的 `toString` 方法，使其返回 `sum` 的值。这样，当你尝试打印 `add(1)(2)(3)` 时，实际上会调用 `innerAdd.toString()`，从而得到结果。

然而，这种覆盖 `toString` 的方法并不是最佳实践，因为它改变了 `innerAdd` 函数的预期行为。一个更好的方法是提供一个方法来获取结果，例如：

```
function add() {
    let sum = 0

    function innerAdd(num) {
        sum += num
        return innerAdd
    }

    innerAdd.getResult = function() {
        return sum
    }

    return innerAdd
}

let result = add(1)(2)(3)
console.log(result.getResult())
```

在这个版本中，我们添加了一个 `getResult` 方法来获取结果，而不是覆盖 `toString`。这样，`innerAdd` 函数的行为就更加清晰和可预测了。

13\. 实现类数组转化为数组

在JavaScript中，有时我们可能需要将类数组对象（例如，具有数字索引和`length`属性的对象）转换为真正的数组。有多种方法可以实现这一目的，以下是一些常见的方法：

方法一：使用`Array.prototype.slice.call()`

```
function toArray(arrayLike) {
    return Array.prototype.slice.call(arrayLike);
}

var arrayLike = {0: 'a', 1: 'b', 2: 'c', length: 3};
var realArray = toArray(arrayLike);
console.log(realArray);
```

方法二：使用扩展运算符（Spread Operator）

```
function toArray(arrayLike) {
    return [...arrayLike]
}

var arrayLike = {0: 'a', 1: 'b', 2: 'c', length: 3}
var realArray = toArray(arrayLike)
console.log(realArray)
```

扩展运算符是一种更简洁且现代的方法，但请注意，它可能在一些老旧的浏览器版本中不被支持。

方法三：使用`Array.from()`方法

```
function toArray(arrayLike) {
    return Array.from(arrayLike)
}

var arrayLike = {0: 'a', 1: 'b', 2: 'c', length: 3}
var realArray = toArray(arrayLike)
console.log(realArray)
```

`Array.from()`方法是一个静态方法，用于从一个类似数组或可迭代的对象创建一个新的数组实例。这种方法在ES6中引入，因此同样可能在一些老旧的浏览器版本中不被支持。

在选择使用哪种方法时，请考虑你的目标浏览器或环境对ES6的支持情况。如果你的代码需要在老旧的浏览器上运行，那么使用`Array.prototype.slice.call()`可能是一个更稳妥的选择。如果你的代码只在新版本的浏览器上运行，那么扩展运算符或`Array.from()`方法将是更简洁、更现代的解决方案。

14\. 将js对象转化为树形结构

将JS对象转换为树形结构通常涉及到递归遍历对象，并根据某种规则（如父子关系）构建树。以下是一个简单的示例，说明如何将一个包含id和parentId的扁平对象数组转换为树形结构。

假设你有以下对象数组：

```
const items = [
  { id: 1, name: 'Item 1', parentId: null },
  { id: 2, name: 'Item 1.1', parentId: 1 },
  { id: 3, name: 'Item 1.2', parentId: 1 },
  { id: 4, name: 'Item 2', parentId: null },
  { id: 5, name: 'Item 2.1', parentId: 4 },
  // ... 更多的项目
];
```

你可以使用以下函数将其转换为树形结构：

```
function buildTree(items, parentId = null) {
  let tree = []
  for (let i in items) {
    if (items[i].parentId == parentId) {
      const children = buildTree(items, items[i].id)
      if (children.length) {
        items[i].children = children
      }
      tree.push(items[i])
    }
  }
  return tree
```

```
}

// 使用上面的函数构建树
const tree = buildTree(items)
console.log(tree)
```

上面的`buildTree`函数会递归遍历`items`数组，查找所有具有指定`parentId`的项。对于找到的每个项，它又会递归地查找所有以该项的`id`为`parentId`的子项，并将这些子项作为`children`数组附加到该项上。最终，所有顶级项（即`parentId`为`null`的项）将被收集到`tree`数组中并返回。

输出将是一个树形结构的数组，如下所示：

```
[
  {
    id: 1,
    name: 'Item 1',
    parentId: null,
    children: [
      { id: 2, name: 'Item 1.1', parentId: 1 },
      { id: 3, name: 'Item 1.2', parentId: 1 }
    ]
  },
  {
    id: 4,
    name: 'Item 2',
    parentId: null,
    children: [
      { id: 5, name: 'Item 2.1', parentId: 4 }
    ]
  },
  // ... 更多的顶级项目及其子项目
]
```

请注意，这个简单的实现假设每个项只有一个父项，并且没有循环引用。如果数据结构更复杂或需要处理其他特殊情况，你可能需要调整或扩展这个函数。

15\. 红灯 3s 亮一次，绿灯 1s 亮一次，黄灯 2s 亮一次；如何让三个灯不断交替重复亮灯？

要实现三个灯（红灯、绿灯和黄灯）不断交替重复亮灯，你可以使用JavaScript的`setInterval`函数来设置每个灯的亮灯时间。以下是一个简单的示例，使用HTML和JavaScript来实现这个功能：

1. **HTML部分**：创建三个灯的元素，并给它们分别添加ID。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initia
l-scale=1.0">
    <title>交通灯模拟</title>
    <style>
        .light {
            width: 50px;
            height: 50px;
            margin: 10px;
        }
        #red {
            background-color: red;
        }
        #green {
            background-color: green;
        }
        #yellow {
            background-color: yellow;
        }
    </style>
</head>
<body>
```

```
    <div id="red" class="light"></div>
    <div id="green" class="light"></div>
    <div id="yellow" class="light"></div>

    <script src="traffic_lights.js"></script>
</body>
</html>
```

1. **JavaScript部分**（`traffic_lights.js`）：使用`setInterval`来设置每个灯的亮灯时间，并添加逻辑来交替显示它们。

```
let redLight = document.getElementById('red');
let greenLight = document.getElementById('green');
let yellowLight = document.getElementById('yellow');

let redInterval = setInterval(function() {
    redLight.style.opacity = 1;
    setTimeout(function() {
        redLight.style.opacity = 0;
    }, 3000);
}, 9000);

let greenInterval = setInterval(function() {
    setTimeout(function() {
        greenLight.style.opacity = 1;
        setTimeout(function() {
            greenLight.style.opacity = 0;
        }, 1000);
    }, 3000);
}, 9000);

let yellowInterval = setInterval(function() {
    setTimeout(function() {
        setTimeout(function() {
```

```
            yellowLight.style.opacity = 1;
            setTimeout(function() {
                yellowLight.style.opacity = 0;
            }, 2000);
        }, 4000);
    }, 0);
}, 9000);
```

注意：在这个示例中，我使用了`opacity`属性来模拟灯的亮灭。当`opacity`为1时，灯是亮的；当`opacity`为0时，灯是灭的。你可以根据需要调整这个逻辑，例如使用不同的背景颜色或添加动画效果。

此外，你可能还希望添加一些逻辑来处理用户交互或响应其他事件，但这已经超出了问题的基本范围。

16\. 用Promise实现图片的异步加载

在JavaScript中，我们可以使用Promise来实现图片的异步加载。Promise是一种表示异步操作可能完成（或失败）的对象。

以下是一个简单的例子，说明如何使用Promise来加载图片：

```
function loadImage(url) {
    return new Promise((resolve, reject) => {
        let img = new Image()
        img.onload = () => {
            resolve(img)
        }
        img.onerror = (error) => {
            reject(error)
        }
        img.src = url
    })
}

// 使用方式
loadImage('<https://example.com/path/to/image.jpg>')
```

```
    .then(img => {
        document.body.appendChild(img)
        console.log('图片加载成功')
    })
    .catch(error => {
        console.error('图片加载失败', error)
    })
```

在这个例子中，`loadImage`函数返回一个新的Promise。这个Promise在图片加载成功时通过`resolve`方法解决，并在加载失败时通过`reject`方法拒绝。

我们使用`new Image()`来创建一个新的Image对象，并设置其`onload`和`onerror`事件处理器。当图片加载成功时，`onload`事件处理器会被调用，我们使用`resolve`方法将图片对象传递给Promise的链式调用。如果图片加载失败，`onerror`事件处理器会被调用，我们使用`reject`方法将错误对象传递给Promise的链式调用。

然后，我们可以调用`loadImage`函数，并通过`.then()`和`.catch()`方法来处理Promise的结果。如果图片加载成功，`.then()`方法中的回调函数会被调用，并接收到加载成功的图片对象。如果图片加载失败，`.catch()`方法中的回调函数会被调用，并接收到错误对象。

17\. 实现发布-订阅模式

发布-订阅模式是一种消息传递模式，允许发送者（发布者）发送消息，而不直接将其发送给特定的接收者（订阅者）。同样，接收者可以订阅一个或多个消息，而无需知道发送者的身份。这种模式在事件驱动编程中非常常见，比如浏览器的DOM事件、Node.js的事件等。

下面是一个简单的JavaScript实现发布-订阅模式的例子：

```
class PubSub {
    constructor() {
        this.subscribers = {};
    }

    subscribe(event, callback) {
        if (!this.subscribers[event]) {
```

```
            this.subscribers[event] = [];
        }
        this.subscribers[event].push(callback);
    }


    unsubscribe(event, callback) {
        if (this.subscribers[event]) {
            this.subscribers[event] = this.subscribers[even
t].filter(subCallback => subCallback !== callback);
        }
    }


    publish(event, data) {
        if (this.subscribers[event]) {
            this.subscribers[event].forEach(callback => callb
ack(data));
        }
    }
}


const pubsub = new PubSub();


pubsub.subscribe('myEvent', data => console.log('Received dat
a:', data));


pubsub.publish('myEvent', 'Hello, world!');


pubsub.unsubscribe('myEvent', (data) => console.log('Received
data:', data));
```

```
pubsub.publish('myEvent', 'Hello again!');
```

在这个例子中，我们创建了一个`PubSub`类，它有三个方法：`subscribe`、`unsubscribe`和`publish`。`subscribe`方法用于订阅事件，它将回调函数存储在一个以事件名为键的对象中。`unsubscribe`方法用于取消订阅事件，它从存储的回调函数中移除指定的回调函数。`publish`方法用于发布事件，它遍历所有订阅了该事件的回调函数，并调用它们。

18\. 封装异步的fetch，使用async await方式来使用

在JavaScript中，封装异步的fetch请求，并使用async/await语法，可以使代码更加清晰和易于理解。下面是一个简单的封装示例：

```
async function fetchData(url, options = {}) {
    try {
        const response = await fetch(url, options);


        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.s
tatus}`);
        }


        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Fetch error:', error);
        throw error;
    }
}


async function fetchAndUseData() {
```

```
    try {
        const url = '<https://api.example.com/data>';
        const data = await fetchData(url);


        console.log(data);

    } catch (error) {
        console.error('Error fetching and using data:', erro
r);
    }
}


fetchAndUseData();
```

在这个示例中，`fetchData`函数是一个异步函数，它使用`await`关键字等待`fetch`请求的完成。一旦请求完成，它检查响应状态码，如果不在200-299范围内，则抛出错误。否则，它尝试解析响应为JSON格式的数据，并返回该数据。

`fetchAndUseData`函数也是异步的，它调用`fetchData`函数，并使用`await`等待其完成。一旦数据被成功获取，它就可以使用这些数据做进一步的处理。如果在获取或使用数据的过程中发生错误，它会捕获错误并打印到控制台。

注意，由于`fetchData`和`fetchAndUseData`都是异步函数，它们需要使用`await`来等待异步操作的结果，这要求调用它们的代码也必须是异步的（通常在一个`async`函数内部）。

此外，封装`fetch`请求的好处之一是你可以在其中添加额外的逻辑，比如添加请求头、处理不同的响应类型（不只是JSON）、添加重试逻辑、管理请求取消等。这样，你的代码会更加模块化和可维护。

19\. 实现双向数据绑定

双向数据绑定是一种设计模式，通常在前端框架（如Vue.js或Angular）中使用，它允许数据在视图（UI）和模型（数据源）之间自动同步。当模型数据发生变化时，视图会自动更新；同样地，当用户在视图中修改数据时，模型数据也会相应更新。

以下是一个使用原生JavaScript实现的简单双向数据绑定的例子。这个例子基于观察者模式（Observer Pattern）和发布-订阅模式（Publish-Subscribe Pattern）。

```
class Dep {
    constructor() {
        this.subs = [];
    }

    addSub(sub) {
        this.subs.push(sub);
    }

    notify() {
        this.subs.forEach(sub => sub.update());
    }
}

class Watcher {
    constructor(vm, exp, cb) {
        this.cb = cb;
        this.vm = vm;
        this.exp = exp;
        this.value = this.get();
    }

    get() {
        Dep.target = this;
        let value = this.vm[this.exp];
        Dep.target = null;
        return value;
    }
```

```
    update() {
        let newValue = this.vm[this.exp];
        if (newValue !== this.value) {
            this.value = newValue;
            this.cb(newValue);
        }
    }
}


class Vue {
    constructor(data) {
        this.data = data;
        Object.keys(data).forEach(key => {
            this[key] = this._proxyData(key);
        });
        this._initWatch();
    }


    _initWatch() {
        this._watchers = [];
        let updateComponent = () => {
            console.log('组件更新');
        };
        Object.keys(this.data).forEach(key => {
            new Watcher(this, key, updateComponent);
        });
    }


    _proxyData(key) {
        let self = this;
        return new Proxy(this.data[key], {
```

Power Human With Education

```
            get(target, prop) {
                if (Dep.target) {
                    let dep = target.__dep__ || (target.__dep
__ = new Dep());
                    dep.addSub(Dep.target);
                }
                return Reflect.get(target, prop);
            },
            set(target, prop, value) {
                let result = Reflect.set(target, prop, valu
e);
                let dep = target.__dep__;
                if (dep) {
                    dep.notify();
                }
                return result;
            }
        });
    }
}


let vm = new Vue({
    data: {
        message: 'Hello, Vue!'
    }
});


vm.$watch('message', (newVal, oldVal) => {
    console.log(`Message changed from ${oldVal} to ${newVal}
`);
});
```

Power Human With Education

```
vm.message = 'Hello, World!';
```

这个简单的双向数据绑定实现包含了三个主要部分：

1. **Dep类**：用于存储订阅者（Watcher实例），并在数据变化时通知它们更新。
2. **Watcher类**：观察数据变化，当数据变化时更新视图。
3. **Vue类**：模拟Vue实例，使用Proxy对数据进行代理，实现getter和setter的拦截，从而在数据读取和设置时添加或通知订阅者。

请注意，这个实现是非常基础和简化的，只是为了演示双向数据绑定的基本原理。在真实世界的应用中，框架如Vue.js和Angular会包含更多的优化和功能，如计算属性、虚拟DOM、组件系统、指令等。如果你想要一个成熟且功能丰富的双向数据绑定解决方案，建议使用这些现成的框架。

20\. 实现简单路由

一个简单的路由系统可以通过JavaScript对象和事件监听器来实现。下面是一个简单的基于浏览器的路由示例，使用URL的hash部分来模拟路由变化，并触发相应的事件或函数。

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
<title>Simple Router</title>
<script>

const routes = {
    '/': function() {
        document.getElementById('content').textContent = 'Hom
e Page';
    },
    '/about': function() {
        document.getElementById('content').textContent = 'Abo
```

```
ut Page';
    },
    '/contact': function() {
        document.getElementById('content').textContent = 'Con
tact Page';
    }

};


function initRoute() {
    const hash = window.location.hash.substr(1);
    if (routes[hash]) {
        routes[hash]();
    } else {
        routes['/']();
    }
}


window.addEventListener('hashchange', function() {
    initRoute();
});


window.onload = initRoute;
</script>
</head>
<body>

<nav>
    <a href="#/">Home</a> |
    <a href="#/about">About</a> |
    <a href="#/contact">Contact</a>
</nav>
```

Power Human With Education

```
<div id="content"></div>

</body>
</html>
```

在这个例子中，我们定义了一个`routes`对象，它包含了不同的路由路径和对应的处理函数。当URL的hash部分发生变化时（例如，用户点击了导航链接），`hashchange`事件会被触发，然后调用`initRoute`函数来根据新的hash值更新页面内容。

请注意，这个简单的路由系统并没有处理路由参数或复杂的路由模式。对于更复杂的路由需求，你可能需要使用现有的前端路由库，如`vue-router`（用于Vue.js应用）或`react-router`（用于React应用）。这些库提供了更强大和灵活的路由功能，包括嵌套路由、动态路由匹配、路由参数解析等。

21\. 实现斐波那契数列

在JavaScript中，你可以使用多种方法来实现斐波那契数列。以下是其中的一些方法：

1\. 递归方法

递归方法是最直接的实现方式，但效率较低，因为对于每个新的斐波那契数，它都会重新计算之前的数。

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

console.log(fibonacciRecursive(10));
```

2\. 迭代方法

迭代方法比递归方法更有效率，因为它避免了重复计算。

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

console.log(fibonacciIterative(10))
```

3\. 动态规划方法

动态规划方法也可以有效地计算斐波那契数列，尤其是当需要计算大量的斐波那契数时。

```
function fibonacciDynamicProgramming(n) {
    let dp = new Array(n + 1).fill(0)
    dp[0] = 0
    dp[1] = 1
    for (let i = 2
        dp[i] = dp[i - 1] + dp[i - 2]
    }
    return dp[n]
}

console.log(fibonacciDynamicProgramming(10))
```

4\. 使用Binet的公式

对于非常大的n值，可以使用Binet的公式来高效地计算斐波那契数。但是，由于JavaScript的浮点数精度问题，这种方法在n非常大时可能会产生不精确的结果。

```
function fibonacciBinet(n) {
    return Math.round((Math.pow(1.618033988749895, n) - Math.
pow(-0.618033988749895, n)) / Math.sqrt(5));
}

console.log(fibonacciBinet(10));
```

以上就是在JavaScript中实现斐波那契数列的几种方法。你可以根据你的具体需求选择合适的方法。

22\. 字符串出现的不重复最长长度

在JavaScript中，要找到字符串中出现的不重复字符的最长长度，可以使用滑动窗口（Sliding Window）技术。下面是一个使用滑动窗口算法实现的函数：

```
function lengthOfLongestSubstring(s) {
    let start = 0
    let maxLength = 0
    let seen = new Set()

    for (let end = 0
        // 如果当前字符已经在窗口内，则移动窗口的起始位置
        while (seen.has(s[end])) {
            seen.delete(s[start])
            start++
        }

        // 将当前字符添加到窗口内
        seen.add(s[end])

        // 更新最长不重复子串的长度
        maxLength = Math.max(maxLength, end - start + 1)
    }

    return maxLength
```

```
}

// 示例
const s = "abcabcbb"
console.log(lengthOfLongestSubstring(s))
```

在这个函数中，我们使用了一个 `Set` 数据结构 `seen` 来存储当前窗口内的字符。当遇到一个新的字符时，我们检查它是否已经在 `seen` 集合中。如果在，则说明这个字符在窗口内重复了，我们需要移动窗口的起始位置 `start`，并从 `seen` 中移除相应的字符，直到这个重复字符不再出现在窗口中。然后，我们将新字符添加到 `seen` 集合中，并更新最长不重复子串的长度。

最后，我们返回最长不重复子串的长度 `maxLength`。

在上面的示例中，字符串 `"abcabcbb"` 的最长不重复子串是 `"abc"`，因此函数返回 `3`。

23\. 使用 setTimeout 实现 setInterval

`setInterval` 和 `setTimeout` 都是 JavaScript 中用于执行定时任务的函数，但它们的行为有所不同。`setInterval` 会按照指定的间隔不断重复执行某个函数，而 `setTimeout` 则只会在指定的时间后执行一次函数。

虽然 `setInterval` 已经提供了我们需要的定时重复执行的功能，但有时我们可能出于某种原因（比如对性能的优化或者更精确的控制）想要使用 `setTimeout` 来模拟 `setInterval` 的行为。下面是一个示例：

```
function mySetInterval(callback, delay) {

    callback();


    const intervalId = setTimeout(() => {

        clearTimeout(intervalId);


        mySetInterval(callback, delay);
```

```
        callback();
    }, delay);
}

mySetInterval(() => console.log('Hello, world!'), 1000);
```

这个 `mySetInterval` 函数首先执行一次回调函数，然后设置一个 `setTimeout` 在指定的延迟后执行。在 `setTimeout` 的回调函数中，我们首先清除前一个 `setTimeout`（如果有的话），然后递归调用 `mySetInterval` 来设置下一个 `setTimeout`，最后执行回调函数。这样就实现了类似 `setInterval` 的功能。

需要注意的是，这种方法并不能完全替代 `setInterval`，因为 `setTimeout` 的延迟时间并不是精确的，它受到 JavaScript 事件循环和浏览器性能的影响。如果回调函数的执行时间超过了设定的延迟时间，那么 `setTimeout` 的实际触发时间就会比预期的要晚。这也是为什么我们在上面的代码中在每次调用 `setTimeout` 之前先清除前一个 `setTimeout` 的原因，以尽量避免这种累积的延迟。