JavaScript（进阶）

作用域

- 什么是作用域
  - 作用域规定了变量被访问的范围，离开了这个范围，变量就不能再被访问
- 作用域的分类
  - 局部作用域
    - 函数作用域：在函数内声明的变量在之后函数外面不能被访问
    - 块作用域:在大括号范围内定义的变量，在大括号外无法被访问
  - 全局作用域
    - 在script和js外部文件中声明的变量
    - 为window对象添加的属性也是全局的
    - 在函数中未使用关键字声明的也是全局作用的
- 作用域链
  - 作用域链的本质是一种查找机制

  -
    ```
    <script>
      function out(){
        let a=10
        function inp(){
          a=20
          console.log(a)
        }
        inp()
      }
      out()
    </script>
    ```
  - 这里是因为a都在两个函数中赋值了，如果inp函数没a，但是要打印，会往上面寻找，找到a
  - 子能访问父，父不能访问子
- 垃圾回收机制
  - js中内存分配和回收会自动完成，在不使用的时候会被垃圾回收机制回收
  - 生命周期
    - 内存分配
    - 内存使用
    - 内存回收
  - 全局变量一般不会回收，但是局部变量一般会回收
  - 内存泄露：内存在某种情况下，没有被回收
  - 算法说明
    - 在栈中存放的是简单数据类型，一般由操作系统自动回收，但是堆中存放的是复杂数据类型，一般由程序员分配释放，若不释放，垃圾回收机制会来
    - 分类
      - 引用计数法
      - 标记清除法
    - 引用计数法
      - 这个方法一般是来跟踪一个复杂数据类型，如果被引用，就会加一，不引用就减一，0的时候被回收

      -
        ```
        const arr = [1, 2, 3]
        arr = null
        ```

      -
        ```
        let obj1 = {}
        let obj2 = {}
        obj1.a = obj2
        obj2.a = obj1
        ```
      - 但是如果两个对象相互引用，就会内存泄漏
    - 标记清除法
      - 这个方法是从全局出发，根据各种关系去找，没找到的就去回收

      -
        ```
        let a = 10
        let b = 20
        let c = 30
        a += b
        console.log(a+b);
        ```
  - 闭包
    - 什么是闭包
      - 闭包就是将一个函数和一个引用捆绑在一起，用内层函数取访问外层函数的作用域
      - 闭包=内层函数+外层函数的变量

      -
        ```
        function outer() {
          let a = 10
          function fn() {
            a += 10
            console.log(a);
          }
          return fn
        }
        ```
    - 用法
      - 例图在上面
      - 调用：

        -
          ```
          const fun = outer()
          fun()
          ```
      - 解释

        -
          ```
          // outer()===fn===function fn(){}
          //const fun=funcion fn()=outer()
          ```
    - 应用
      - 可以来实现数据的私有，使得外部来访问内部的变量
    - 缺点
      - 可能会引起内存泄漏
  - 变量提升
    - 针对于var而言
      - 在执行之前，系统会把var声明的变量提到前面来，只声明，不赋值
    - 结果是undefined

      -
        ```
        console.log(num);
        var num = 10
        ```
    - 由于var的缘故，结果和这个相等

      -
        ```
        var num
        console.log(num);
        var num = 10
        ```

函数进阶

- 函数提升
  - 函数提升和变量提升一样，执行之前会把函数声明提升到当前作用域最前，并不提升调用

    -
      ```
      fn()
      function fn() {
        console.log(a);
      ```
    - 但是函数表达式不行

      -
        ```
        fn()
        const fn = function () { }
        ```
- 函数参数
  - 动态参数
    - 当不确定用户输进来几个参数，就引入了arguments

    -
      ```
      function fun() {
        for (let i = 0; i < arguments.length; i++)
          console.log(arguments[i]);
      }
      fun(1, 2, 3)
      ```
    - 注意这里的形参不要加东西
    - 这里的arguments是一个伪数组，没有pop,push
  - 剩余参数
    - 和上面的用法一样，不知道输入几个参数，就引入...

    -
      ```
      function fun(...arr) {
        for (let i = 0; i < arr.length; i++)
          console.log(arr[i]);
      }
      fun(1, 2, 3)
      ```
    - 这里的...arr要写在形参
    - 这里的...arguments是一个真数组，
    - 如果在...之前加几个参数，就会代替几个参数，如果变成a...arr结果就是2，3.这里的a就是1
  - 补充：展开运算符
    - 展开运算符能够将数组全部打印出来，不需要for循环
    - 和剩余参数类似，但是展开运算符是在函数外，剩余参数是在函数里面

    -
      ```
      const arr = [1, 2, 3, 5]
      console.log(...arr)
      ```
    - 作用
      - 求最值

        -
          ```
          const arr = [1, 2, 3, 5]
          console.log(...arr)
          console.log(Math.max(...arr))
          ```
        - 因为这里的数学表达式无法传入数组，只能传1,2,3,
      - 合并数组

        -
          ```
          const arr = [1, 2, 3, 5]
          console.log(...arr)
          // console.log(Math.max(...arr))
          const arr1 = [...arr, ...arr]
          console.log(...arr1)
          ```
        - 这里直接写入数组里面就行
- 箭头函数
  - 什么是箭头函数
    - 来代替函数表达式的一个简便写法
  - 基本语法

    -
      ```
      const fn1 = () => {
        console.log('111');
      }
      fn1()
      ```
    - 有点类似于lamda表达式

    -
      ```
      const fn2 = x => {
        console.log('222' + x);
      }
      fn2(2)
      ```
    - 当只有一个形参的时候，小括号可以省略

    -
      ```
      const fn3 = x => console.log('222' + x);
      fn3(3)
      ```
    - 当后面只有一个语句的时候，可以省略大括号

    -
      ```
      const fn4 = x => x * x
      console.log(fn4(2))
      ```
    - 当后面只有一行代码并且是renturn时，连return都可以省略

    -
      ```
      const fn6 = (uname) => ({ name: uname })
      console.log(fn6('刘'))
      ```
    - 可以直接返回一个对象
  - 箭头函数的参数
    - 箭头函数没有arguments,但是有...arr
    - this参数

      -
        ```
        function fun() {
          console.log(this)
        }
        fun()
        ```
      - 普通函数的this参数，输出是window

      -
        ```
        const obj = {
          name: 'zxl',
          getSum: function () {
            console.log(this)
          }
        }
        obj.getSum()
        ```
      - 对象中方法的this参数，输出是对象名

      -
        ```
        const fun1 = x => {
          console.log(this);
        }
        fun1()
        ```
      - 箭头函数的this参数，输出是window，但是这里的this参数不是window调用，而是本来没有this参数，他要往上去寻找，就找到了window

      -
        ```
        const fun1 = x => {
          console.log(this);
        }
        fun1()
        ```
      - 对象中的箭头函数的this参数，输出是window,这里网上找还是找不到，因为往上是对象，对象也是window调用

      -
        ```
        const obj1 = {
          name: 'ghy',
          getName: function () {
            const cur = () => {
              console.log(this);
            }
            cur()
          }
        }
        obj1.getName()
        ```
      - 对象中方法嵌套一个箭头函数的this参数，输出的就是对象名
  - 箭头函数的缺点
    - 对于事件回调函数，addeventlistener,里面使用箭头函数返回的是window,所以事件回调函数不建议使用箭头函数

解构赋值

- 数组解构
  - 什么是数组解构
    - 数组解构就是将数组元素快速批量的赋值给一系列变量的简介语法
  - 数组解构的语法

    -
      ```
      const arr = [1, 2, 3]
      const [a, b, c] = arr
      console.log(a)
      ```
  - 数组解构的简单应用

    -
      ```
      let d = 10
      let e = 20
        ;[d, e] = [e, d]
      console.log(d, e);
      ```
    - 要注意这里的[]前面一定要加;否则会报错，系统会认为和20连在一起了
  - 数组解构的特殊情况

    -
      ```
      const [a, b, c] = [1, 2]
      console.log(c);
      ```
    - 变量数<数组元素数量，这里c的值是undefined

    -
      ```
      const [a, b, ...c] = [1, 2, 3, 4]
      console.log(c);
      ```
    - 当变量数<数组元素数量，可以用剩余参数代替，这里的c是一个真数组

    -
      ```
      const [a = 0, b = 0] = [1]
      console.log(b);
      ```
    - 为了防止变量数<数组元素数量，可以先初始化，这里b的值是0

    -
      ```
      const [a, b, , d] = [1, 2, 3, 4]
      console.log(d);
      ```
    - 可以按需赋值，中间跳过，这里的d=4

    -
      ```
      const [a, b, [c, d]] = [1, 2, [3, 4]]
      console.log(c, d);
      ```
    - 可以支持多维数组，跟着写样式就行
- 对象解构
  - 什么是对象解构
    - 就是将对象里面的属性和方法快速批量的赋值给一系列变量的简洁操作
  - 对象解构的语法

    -
      ```
      const obj = {
        uname: 'pink',
        age: 18
      }
      const { uname, age } = obj
      console.log(age)
      ```
    - 要注意要变成{},并且里面的参数名要等于对象参数名

    -
      ```
      const obj = {
        uname: 'pink',
        age: 18
      }
      const { uname: name, age } = obj
      console.log(name)
      ```
    - 可以进行改名
  - 对象解构的特殊情况

    -
      ```
      const obj = [
        {
          uname: 'pink',
          age: 18
        }
      ]
      const [{ uname, age }] = obj
      console.log(age);
      ```
    - 可以进行数组对象解构

    -
      ```
      const obj = {
        uname: 'pink',
        son: {
          mothername: 'red'
        }
      }
      const { uname, son: { mothername } } = obj
      console.log(mothername);
      ```
    - 可以进行多级对象解构
- forEach方法遍历数组

  -
    ```
    const arr = [1, 2, 3]
    arr.forEach(item => { console.log(item); })
    const arr1 = arr.map(item1 => { return item1 + 5 })
    console.log(arr1);
    ```
  - 用法和map类似，但是不同在于这个没有返回值，map可以返回数组
- filter方法筛选数组

  -
    ```
    const arr = [1, 2, 3, 4]
    const arr1 = arr.filter(x => x * x)
    console.log(...arr1);
    ```
  - 可以筛选出想要的元素，而且以数组的形式返回，与箭头函数搭配更好

深入对象

- 创建对象的三种方法
  - 第一种就是常用的字面量，直接{}

  -
    ```
    const obj=new Object({uname:'pink'})
    ```
  - 第二种就是

  -
    ```
    function fun(uname) {
      this.uname = uname
    }
    const o1 = fun('pink')
    ```
  - 第三种就是创建一个构造函数并且调用
- 构造函数
  - 什么是构造函数
    - 构造函数是一种用来初始化对象的特殊函数
    - 构造函数的命名：大写开头
  - 怎么构造函数

    -
      ```
      function fun(uname) {
        this.uname = uname
      }
      const o1 = fun('pink')
      ```
  - 构造函数发生的过程
    - 首先创建一个新对象
    - 其次用this指向空对象
    - 然后执行构造函数进行添加新属性
    - 最后返回新对象
  - 实例成员
    - 实例成员是实例对象的属性和方法
  - 静态成员
    - 静态成员是构造函数的属性和方法
    - 静态成员只能由构造函数进行访问
    - 静态方法的this指向构造函数

内置构造函数

- 什么是内置构造函数
  - 因为简单数据类型有专门的构造函数，是包装类型
- Object
  - 三个静态方法

    -
      ```
      const obj = { uname: 'pink', age: 18 }
      console.log(Object.keys(obj));
      ```
    - Object.key(obj)用来获取对象的全部属性名，uname,age

    -
      ```
      const obj = { uname: 'pink', age: 18 }
      console.log(Object.values(obj));
      ```
    - values()用来获取所有属性值，'pink',18

    -
      ```
      const oo = {}
      Object.assign(oo, obj)
      console.log(oo);
      ```
    - assign(obj,obj1)用来拷贝对象
- Array
  - reduce方法
    - reduce方法是什么
      - reduce方法是处理数组累计处理的，常用于累加器
    - reduce方法的语法

      -
        ```
        const arr = [1, 4, 3]
        const sum = arr.reduce((prev, cur) => prev + cur)
        console.log(sum);
        const sumNew = arr.reduce((prev, cur) => prev + cur, 10)
        console.log(sumNew);
        ```
      - 这里的后面有一个起始值，可以加可以不加
    - reduce方法的执行过程
      - 如果没有起始值的话，按照数组第一个元素作为先前的元素，后面的作为后面的元素，累加的结果作为下一次的前者，一直循环，直到数组结束
      - 如果有起始值的话，起始值作为前者，数组的第一个元素作为后者，相加的结果再次作为前者，如上
    - reduce方法的注意
      - 对对象数组进行操作的时候，要给起始值加上0
  - find方法
    - 什么是find方法
      - find方法主要是进行查找的运算
    - find方法的语法

      -
        ```
        //find方法
        const arr = [
          {
            uname: '小米',
            price: 1888
          },
          {
            uname: '华为',
            price: 3999
          }
        ]
        const obj = arr.find(item => item.uname = '小米')
        console.log(obj.price);
        ```
      - 返回的是查找成功的第一个元素
  - every方法
    - 什么是every方法
      - every方法是检查是否数组里面的值都满足同一个函数规则
    - every的语法

      -
        ```
        const arr = [1, 2, 3, 4]
        const cur = arr.every(item => item >= 10)
        console.log(cur);
        ```
      - 返回的是true/false
  - from方法
    - 什么是from方法
      - from方法就是把伪数组转化为真数组
    - from方法的语法

      -
        ```
        Array.from(document.querySelector('ul li'))
        ```
      - 这里把document.querySelectorAll查找的伪数组转化为真数组，加了pop()等方法
- String
  - split方法
    - 什么是split方法
      - split方法是将字符串因为某个分隔符分隔开
    - split方法的语法

      -
        ```
        const str = 'Hel,lo'
        console.log(str.split(','));
        ```
      - 返回的是真数组，将，两边的分隔开
  - subString方法
    - 什么是subString方法
      - substring方法就是将字符串截取
    - substring的语法

      -
        ```
        const str1 = 'Hello'
        console.log(str1.substring(1, 3));
        ```
      - 如果后面只有一个参数，返回的就是从索引为n的字符往后全部截取，如果是两个，就是从n截取到m-1的字符，长度间检查一下就是m-n
  - startWith方法
    - 什么是startWith方法
      - startWith方法是检查字符串是否以某某字符开头
    - startwith的语法

      -
        ```
        const str2 = 'Hello'
        console.log(str2.startsWith('e', 1));
        ```
      - 返回的是true/false，如果后面没有参数，检查的就是全局是否是这个字符开头，如果有，就是从当前索引进行查找
  - includes方法
    - 什么是includes方法
      - includes就是来检查一个字符串是否包含在另一个字符串中
    - includes的语法

      -
        ```
        const str3 = 'Hello'
        console.log(str3.includes('ll', 3));
        ```
      - 返回的也是true/false，如果后面没有参数，就是全局查找，如果有的话，就是从当前索引进行查找
- Number
  - toFixed方法
    - 什么是toFixed方法
      - toFixed方法就是来保留几位小数的
    - toFixed方法的语法

      -
        ```
        const num = 100.966
        console.log(num.toFixed(2));
        ```
      - 返回的是保留几位小数后的值，2就是保留两位小数，并且会四舍五入

编程思想

- 面向过程
  - 面向过程是分析完成任务所需步骤，并且用函数一步一步实现（类似于蛋炒饭）
- 面向对象
  - 面向对象是把事务包装成类，然后分工合作（类似于盖浇饭）
- 两种思想区别
  - 面向对象的灵活性高，有封装的好处，但是性能不高
  - 面向对象性能高，适合硬件，操作系统，但是不易维护

构造函数

- 什么是构造函数
  - 构造函数是用来初始化对象的函数
- 构造函数的缺点
  - 会产生内存浪费，例如我要创建五个对象，要创建五个sing方法，用五个对象去调用这个方法，但是这五个对象因为不同，所以都要在开辟新的空间存放这个函数，会变成很多内存

原型

- 什么是原型
  - 原型是可以实现方法共享的
- 原型如何实现

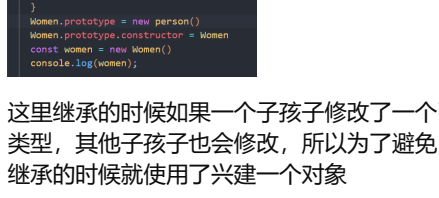
  -
    ```
    function Star(uname) {
      this.uname = uname
    }
    Star.prototype.Sing = function () {
      console.log('111');
    }
    const pink = new Star('pink')
    const red = new Star('red')
    console.log(pink.Sing === red.Sing)
    ```
  - 利用构造函数.prototype.变量来实现定义共享方法这时所打印的就是true，不会多开辟内存
- constructor属性

  -
    ```
    function person() {
      this.eays = 2
      this.head = 1
    }
    function Man() {

    }
    Man.prototype
    Man.prototype.constructor = Man
    ```
  - 原型是用来指向共享函数的，那原型怎么指向回构造函数呢，就要用到constructor来进行操作
- 对象原型
  - 每一个实例对象都会有一个属性__proto__来指向原型，因为我用对象来调用原型，那我是如何调用的呢，就是利用个属性
- 原型继承
  - 原型继承就是原型他其实也有__proto__方法用来指向上一个原型，他是继承下来，假如有一个对象的原型，他就是从Object的原型继承下来的

  -
    ```
    function person() {
      this.eays = 2
      this.head = 1
    }
    function Man() {

    }
    Man.prototype = new person()
    Man.prototype.constructor = Man
    Man.prototype.baby = function () {
      console.log('孩子');
    }
    const man = new Man()
    console.log(man);
    // console.log(man.head);
    function Women() {

    }
    Women.prototype = new person()
    Women.prototype.constructor = Women
    const women = new Women()
    console.log(women);
    ```
  - 这里继承的时候如果一个子孩子修改了一个引用类型，其他子孩子也会修改，所以为了避免，在继承的时候就使用了兴建一个对象
- 原型链
  - 
  - 类似于图

深浅拷贝

- 浅拷贝
  - 什么是浅拷贝
    - 浅拷贝就是把一个对象新建，来引用另一个对象
  - 浅拷贝如何实现

    -
      ```
      const obj = {
        arr: [1, 2, 3],
        obj1: {
          uanme: 'pink'
        }
      }
      const o = { ...obj }
      o.uname = 'red'
      console.log(o);
      console.log(obj);
      ```
    - 这里的...obj就是浅拷贝的第一种方法

    -
      ```
      const obj = {
        arr: [1, 2, 3],
        obj1: {
          uanme: 'pink'
        }
      }
      const o1 = Object.assign({}, obj)
      obj.obj1.uanme = 'red'
      console.log(obj);
      console.log(o1);
      ```
    - 这里用assign也是浅拷贝的一种方法
  - 浅拷贝的特点
    - 浅拷贝的特点在于是应用了地址过来，当修改原对象的引用数据时，原对象也会改变
- 深拷贝
  - 什么是深拷贝
    - 深拷贝就是原封不动的兴建一个对象和另一个对象一样，是开辟了新的空间拷贝过来
  - 深拷贝如何实现
    - 第一个方法四用lodash方法进行

    -
      ```
      const obj = {
        uname: 'pink'
      }
      const o = JSON.parse(JSON.stringify(obj))
      console.log(o);
      ```
    - 第二个是利用JSON方法进行

    -
      ```
      const obj = {
        uname: 'pink'
      }
      // const o = JSON.parse(JSON.stringify(obj))
      // console.log(o);
      function deepCopy(newobj, oldobj) {
        for (let k in oldobj) {
          if (oldobj instanceof Array) {
            newobj[k] = []
            deepCopy(newobj[k], oldobj[k])
          }
          else {
            newobj[k] = oldobj[k]
          }
        }
      }
      const o1 = {}
      deepCopy(o1, obj)
      obj.uname = 'red'
      console.log(o1);
      ```
    - 第三个方法是手写递归函数来实现这里面碰到了数组还要递归，我没写对象，对象也是
  - 深拷贝的特点
    - 深拷贝是源对象和新对象互不干预，谁修改对另一个没有影响

异常处理

- throw
  - 什么是throw
    - throw就是自己抛出一个异常
  - throw的语法

    -
      ```
      <script>
        function fun(a, b) {
          if (!a) {
            throw new Error('11')
          }
        }
        fun()
      ```
- try/catch
  - 什么是try/ctahch
    - try/ctach是一种电脑自动捕获异常的方法，通常还会有fianlly方法，表示不管有没有异常都要执行
  - try/catch如何实现

    -
      ```
      <script>
        try {
          const p = document.querySelector('.p')
          p.style.color = 'red'
        }
        catch (err) {
          console.log(err.message);
        }
        finally {
          alert('11')
        }
      ```
- debugger
  - 什么是debugger
    - debugger是手动写的断点，在调试的时候自动停在这，不用再设断点了

this深入

- 什么是this，this指向谁
  - 前面都说过了
- 改变this
  - 如何改变this
    - call()

      -
        ```
        const obj = {
          uname: 'pink'
        }
        function fun(a, b) {
          console.log(this);
          console.log(a + b);
        }
        fun.call(obj, 1, 1)
        ```
      - 这里用call方法来改变this的指向，后面的参数同时也是形参，本身就是调用的方法
    - apply()

      -
        ```
        const obj = {
          uname: 'pink'
        }
        function fun(a, b) {
          console.log(this);
          console.log(a + b);
        }
        fun.apply(obj, [1, 2])
        ```
      - 这里的apply方法也是改变this指向，不同的是后面的参数是以数组形式传递进去，而且也会直接调用

      -
        ```
        const arr=[1,2,3]
        console.log(Math.max(max.apply(Math,arr)));
        ```
      - 由于他是数组的形式，所以可以来求最大值
    - bind()

      -
        ```
        <script>
          const obj = {
            uname: 'pink'
          }
          function fun(a, b) {
            console.log(this);
            console.log(a + b);
          }
          fun.bind(obj, 1, 2)
        ```
      - bind方法也是改变this的指向，但是要注意的是后面的参数也是一个一个数，但是他不会立马执行，无法充当调用函数的方法

防抖

- 什么是防抖
  - 防抖类似于王者的回城，在一段时间内被打断，就要重新来，直到事件结束
- 如何实现防抖
  - 第一个利用lodash的_.debounce(fun,500),来进行操作

  -
    ```
    //手写防抖
    const div = document.querySelector('div')
    let i = 1
    function mouseMove() {
      div.innerHTML = i++
    }
    function debounce(fn, t) {
      let timer

      return function () {
        if (timer) clearTimeout(timer)
        timer = setTimeout(function () {
          fn()
        }, t)
      }
    }
    div.addEventListener('mousemove', debounce(mouseMove, 500))
    ```
  - 第二个就是自己写

节流

- 什么是节流
  - 节流类似于王者的冷却，只有等到时间到了才会，不然在怎么点都不行
- 如何实现防抖
  - 第一个利用lodash的_.throttle()方法

  -
    ```
    const div = document.querySelector('div')
    let i = 1
    function mouseMove() {
      div.innerHTML = i++
    }
    function debounce(fn, t) {
      let timer = null

      return function () {
        if (!timer) {
          timer = setTimeout(function () {
            fn()
            timer = null
          }, t)
        }
      }
    }
    div.addEventListener('mousemove', debounce(mouseMove, 500))
    ```
  - 第二个就是自己写