原生js

1. **函数调用this指向问题**

包括函数名()和匿名函数调用，this指向window

对象.方法名，this指向对象

New构造函数名，this指向构造函数

间接调用：利用call和apply来实现，this就是call和apply对应的第一个参数， 如果不传值或者第一个值为null,undefined时this指向window

1. Es6中箭头函数的this和定义时有关，和调用无关，调用就是函数的调用模式
2. Call apply bind

call和apply可以调用函数，改变this，实现继承和调用别的对象的方法

Call和apply的定义：调用方法，用一个对象替换掉另一个对象（this）

对象.call(新this对象，实参1，实参2，实参3…）

对象.apply(新this对象，[实参1，实参 2，实参3…]）

Call和apply的用法：

**间接调用函数，改变作用域的this值**

**劫持其他对象的方法**

var foo = {

name:"张三",

logName: function (){ console.log(this.name); }

}

var bar = { name:"李四" };

foo.logName.call(bar);//李四

实质是call改变了foo的this指向为bar,并调用该函数

**两个函数实现继承**

function Animal(name){

this.name=name;

this.showName = function(){ console.log(this.name); }

}

function Cat(name){ Animal.call(this, name); }

var cat = new Cat("Black Cat");

cat.showName(); //Black Cat

**为类数组（arguments和nodeList)添加数组方法push pop**

(function(){

Array.prototype.push.call(arguments, '王五');

console.log(arguments); //['张三', '李四', '王五'] })('张三', '李四')

**合并数组**

let arr1=[1,2,3];

let arr2=[4,5,6];

Array.prototype.push.apply(arr1,arr2); //将arr2合并到了arr1中

**求数组最大值**

Math.max.apply(null, arr)

**判断字符类型**

Object.prototype.toString.call({})

1. 函数的节流和防抖

**节流**：某个时间段内只执行一次

应用：滚动条，resize事件一段时间触发一次

let throttle = function(func, delay){

let timer = null;

return function(){

if(!timer){

timer = setTimeout(function(){

func.apply(this, arguments);

timer = null;

},delay);

}

};

};

function handle(){

console.log(Math.random());

}

window.addEventListener("scroll",throttle(handle, 1000)); //事件处理函数

**防抖**：处理函数截至后一段时间依次执行

应用：scroll resize事件触发完成后一段时间触发

function debounce(fn, wait){

var timeout = null;

return function(){

if(timeout!==null) clearTimeout(timeout); //如果多次触发将上次记录延迟清除掉

timeout=setTimeout(function(){

fn.apply(this, arguments);

timer=null;

},wait);

};

}

//处理函数

function handle(){

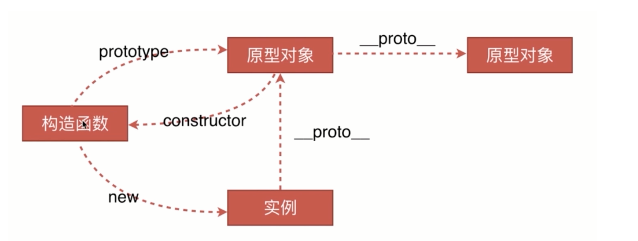
console.log(Math.random());

}

//滚动事件

window.addEventListener("onscroll",debounce(handle,1000));

1. 原型链：对象继承属性的一个链条



1. 创建实例的方法

字面量 let obj={‘name’:’张三’}

Object构造函数创建 let obj=new Object()

obj.name=’张三’

使用工厂模式创建对象

function createPerson(name){

var o=new Object(){

o.name=name;

};

return o;

}

var person1=createPerson(‘张三’);

使用构造函数创建对象

function person(name){

this.name=name;

}

var person1=new Person(‘张三’)

1. new运算符

创建一个新对象

this指向构造函数

构造函数有返回，会替换new出来的对象，如果没有就是new出来的对象

手动封装一个new运算符

var new2 = function ( func ) {

Var o = Object.create( func.prototype ); //创建对象

Var k = func.call( o ); //改变this指向，把结果付给k

If ( typeof k === ’object’) { //判断k的类型是不是对象

return k;

} else { return o; }

1. 继承的方式：js是一门弱类型动态语言，封装和继承是它的两大特性

**原型链继承**：将父类的实例作为子类的原型

//定义一个动物类 （父类）

function Animal(name) {

//属性

this.name = name || ‘Animal’ ;

//实例方法

this.sleep = function() {

console.log(this.name + ‘is sleeping’);

}

}

//原型方法

Animal.prototype.eat = function (food) {

console.log(this.name + ‘is sleeping’);

}

//子类

function Cat(){ }

Cat.prototype = new Animal();

Cat.prototype.name='cat'; //&emsp;Test Code

var cat = new Cat();

console.log(cat.name); //cat

console.log(cat.eat('fish')); //cat正在吃：fish undefined

console.log(cat.sleep()); //cat正在睡觉！

undefined console.log(cat instanceof Animal); //true

console.log(cat instanceof Cat); //true

优缺点：简单易于实现，但是要想为子类新增属性和方法，必须要在new Animal() 这 样的语句之后执行，无法实现多继承

**构造继承**：实质是利用call来改变Cat中的this指向

function Cat (name) {

Animal.call (this);

This.name = name || ’Tom’ ;

}

优缺点：可以实现多继承，不能继承原型属性/方法

**拷贝继承：**将父类的属性和方法拷贝一份到子类中

function Cat (name) {

var animal = new Animal ();

for( var p in animal) {

Cat.prototype [p] = animal [p];

}

Cat.prototype.name = name || ‘Tom’;

}

优缺点：支持多继承，但是效率低占用内存

**组合继承**：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实 例做为子类原型，实现函数复用

function Cat (name) {

Animal.call (this);

this.name = name || ‘Tom’ ;

}

Cat.prototype = new Animal ();

Cat.prototype.constructor = Cat;

**寄生组合继承：**

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

(function(){ // 创建一个没有实例方法的类

var Super = function(){};

Super.prototype = Animal.prototype; //将实例作为子类的原型

Cat.prototype = new Super(); })();

**ES6的extends继承：**ES6的继承机制是先创造父类的实例对象this（所以必须先调用 super方法），然后再用子类的构造函数修改this

//父类

class Person { //constructor是构造方法

constructor(skin, language) {

this.skin = skin;

this.language = language;

}

say() { console.log('我是父类') }

}

//子类

class Chinese extends Person {

constructor(skin, language, positon) {

//console.log(this);//报错

super(skin, language);

//super();相当于父类的构造函数

//console.log(this);调用super后得到了this，不报错，this指向子类，相当于调用了父类

.prototype.constructor.call(this)

this.positon = positon;

}

aboutMe() {

console.log(`${this.skin} ${this.language} ${this.positon}`); }

} //调用只能通过new的方法得到实例,再调用里面的方法

let obj = new Chinese('红色', '中文', '香港');

obj.aboutMe();

obj.say();

1. **高阶函数**：函数的参数是函数或返回函数

常见：map reduce filter sort

**柯里化**：只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数

fn(a, b, c, d) =>fn(a) (b) (c) (d)

let currying = function (fn) {

//args 获取第一个方法内的全部参数

var args = Array.prototype.slice.call (arguments, 1)

return function (){

//将后面方法里的全部参数和args进行合并

var newArgs = args.concat (Array.prototype.slice.call (arguments ))

//把合并后的参数通过apply作为fn的参数并执行

return fn.apply (this, newArgs)

}

}

**反柯里化**： obj.func (arg1, arg2) =>func (obj, arg1, arg2)

Function.prototype.uncurrying = function () {

Var that = this;

Return function (){

Return function.prototype.call.apply (that, arguments);

}

};

function sayHi () {

return "Hello " + this.value +" "+[].slice.call(arguments);

}

let sayHiuncurrying = sayHi.uncurrying ();

console.log( sayHiuncurrying ({value:'world'}, "hahaha"));

**偏函数**：指定部分参数来返回一个新的定制函数的形式

function foo (a,b,c) {

return a+b+c;

}

function func (a,b) {

return foo (a, b, 8);

}

1. **对象的声明方法**

**字面量**：var test2 = { x=123, y=456};

**构造函数**：var test1 = new Object ({ x=123, y=456 });

**new的作用**：

1. 创建了一个新对象 2. this指向构造函数
2. 构造函数有返回，会替换new出来的对象，没有就是new出来的对象

**内置方法**：Object.create (obj, descriptor), obj是对象，describe描述符属性(可选)

let test = Object.create({x:123,y:345});

三种方法的**优缺点**：

1. 功能：都能实现对象的声明，并能赋值和取值
2. 继承性：内置方法创建的对象继承到\_\_proto\_\_属性上
3. 隐藏属性：三种声明方法会默认为内部的每个成员（属性或方法）生成一些隐 藏属性，这些隐藏属性是可以读取和可配置的，属性分类见下：
4. 属性读取：Object.getOwnPropertyDescriptor() 或者 getOwnPropertyDescriptor()
5. 属性设置：Object.definePropertype 或 Object.defineProperties
6. **对象的属性**

**属性分类：**

1. 数据属性4个特性：configurable（可配置） enumerable(可枚举)

writable(可修改) value(属性值)

1. 访问器属性2个特性： get(获取) set(设置)
2. 内部属性：由JavaScript引擎内部使用的属性；不能直接访问,但是可以通过对象内 置方法间接访问,如:[[Prototype]]可以通过 Object.getPrototypeOf()访问; 内部属 性用[[ ]]包围表示,是一个抽象操作,没有对应字符串类型的属性名,如[[Prototype]].

**属性描述符**：将一个属性的所有特性编码成一个对象返回

描述符的属性有：数据属性和访问器属性

使用范围：作为方法Object.defineProperty, Object.getOwnPropertyDescriptor, Object.create的第二个参数

1. **Symbol** :一种数据类型，不能new, 因为symbol是一个原始类型，不是对象

**定义方法**：Symbol () ,可以传参var s1 = Symbol(); var s2=Symbol(); s1===s2 //false

**用法：**

1. 不能与其他类型的值进行运算
2. 作为属性名

let mySymbol = Symbol();

// 第一种写法

var a = {}; a[mySymbol] = 'Hello!';

// 第二种写法

var a = { [mySymbol]: 'Hello!' };

// 第三种写法

var a = {}; Object.defineProperty(a, mySymbol, { value: 'Hello!' });

// 以上写法都得到同样结果

a[mySymbol] // "Hello!"

1. 作为对象属性名时，不能用点运算符，可以用 [ ]

let a = {};

let name = Symbol();

1. name = 'lili';

a[name] = 'lucy';

console.log(a.name,a[name]);

1. 遍历不会被for…in、for…of和Object . Keys()、Object.getOwnPropertyNames()取到该 属性

**Symbol.for**: 在全局中搜索有没有以该参数作为名称的Symbol值，如果有，就返回这 个Symbol值否则就新建并返回一个以该字符串为名称的Symbol值

var s1 = Symbol.for(‘foo’);

var s2=Symbol.for(‘foo’);

s1===s2; //true

**Symbol.keyFor** : 返回一个已登记的Symbol类型值的key

var s1=Symbol.for(“foo”);

Symbol.keyFor(s1) //”foo”

var s2=Symbol(“foo”);

Symbol.keyFor(s2); //undefined

1. **遍历**

**一级对象遍历方法**

for…in 遍历对象自身的和可继承的可枚举属性（不含Symbol属性）

Object .keys(obj)

返回一个数组,包括对象自身的(不含继承)的所有可枚举属性(不含Symbol属性)

Object.getOwnPropertyNames(obj)

返回一个数组，包括对象自身的可枚举属性(不含Symbol属性)

Object.getOwnPropertySymbols(obj)

返回一个数组，包含对象自身的所有symbol属性

Reflect.ownKeys(obj)

返回一个数组,包含对象自身的所有(不枚举、可枚举、Symbol)属性

Reflect.enumerate(obj)

返回一个Iterator对象,遍历对象自身的和继承的所有可枚举属性(不含Symbol)

**总结:**1.只有Object.getOwnPropertySymbol(obj)和reflect.ownKeys(obj)可拿到Symbol属性

1. 只有reflect.ownKeys(obj)可以拿到不可枚举属性

**多级对象遍历**：递归实现

var treeNodes = [ {

id: 1,

name: '1',

children: [ {

id: 11,

name: '11',

children: [ {

id: 111,

name: '111',

children:[]

}, {

id: 112,

name: '112'

}

] }, {

id: 12,

name: '12',

children: []

} ],

users: []

}, ];

递归：

var parseTreeJson = function(treeNodes){

if (!treeNodes || !treeNodes.length) return;

for (var i = 0, len = treeNodes.length; i < len; i++) {

var childs = treeNodes[i].children;

console.log(treeNodes[i].id);

if(childs && childs.length > 0){

parseTreeJson(childs);

}

}

};

console.log('------------- 递归实现 ------------------');

parseTreeJson(treeNodes);

1. **深度拷贝**

**Object..assign:** 将源对象（source）的所有可枚举属性，复制到目标对象（target）

合并多个对象

var target = { a:1, b:1};

var source1 = { b:2, c:2};

var source2 ={ c:3};

Object.assign(target, source1, source2}

//这个是伪深度拷贝，只能拷贝第一层

**JSON.stringify** :将对象转化为字符串，而字符串是简单数据类型

**递归拷贝：**

function deepClone (source) {

const targetObj =source.constructor===Array?[]:{}; //判断复制的对象是数组还是对象

for (let keys in source){ //遍历目标

if (source.hasOwnProperty(keys)){

if(source[keys] && typeof source [keys] === ’object’){ //如果值是对象，就递归一下

source[ keys ] && typeof source [keys] === Array ? [] : {};

targetObj[keys] = deepClone( source [leys] );

} else { //如果不是，就直接赋值

targetObj [keys] = source [keys];

}

}

}

return targetObj;

}

1. **数据拦截**：利用对象内置方法，设置属性，进而改变对象的属性值

· **Object.defineProterty**

1. ES5出来的方法
2. 三个参数：对象（必填） 属性值（必填） 描述符（可选）
3. defineProterty的描述符属性

数据属性: value, writable, configurable, enumerable

访问器属性: get, set

注:不能同时设置value和writable,这两对属性是互斥的

1. **Proxy**
2. ES6出来的方法，实质上是对对象做了一个拦截，并提供了13个处理方法
3. 两个参数：对象和行为函数
4. **问题和优点**：reflect对象没有构造函数，可以监听数组索引赋值，改变数组长度的变 化，是直接监听对象的变化，不用深层遍历
5. **defineProterty和proxy的对比**

1.defineProterty是es5的标准,proxy是es6的标准;

2.proxy可以监听到数组索引赋值,改变数组长度的变化;

3.proxy是监听对象,不用深层遍历,defineProterty是监听属性;

4.利用defineProterty实现双向数据绑定(vue2.x采用的核心)

5.利用proxy实现双向数 据绑定(vue3.x会采用)

18、数组

**扁平化n维数组**

[1, [2, 3] ].flat(2) //[1,2, 3]

[1, [2, 3, [4, 5]].flat(3) //[1,2,3,4,5]

[1, [2, 3, [4, 5]]].toString() //'1, 2, 3, 4, 5'

[1 [2, 3, [4, 5[...]].flat(Infinity) //[1, 2, 3, 4...n]

Array.flat(n) 是ES10扁平数组的api, n表示维度，n值为Infinity时维度为无限大

function flatten(arr) {

while (arr.some(item =>Array.isArray(item))){

arr = [].concat(...arr);

}

return arr;

}

flatten ([1, [2, 3] ] //[1, 2, 3]

flatten ([1, [2, 3, [4, 5] ]) //[1, 2, 3, 4, 5]

实质是利用递归和数组合并方法 concat实现扁平

**去重**

Array.from( new Set ([1, 2, 3, 4, 4 ])) //[1, 2, 3, 4 ]

[...new Set ([1, 2, 3, 3, 4, 4 ])] //[1, 2, 3, 4 ]

set是ES6新出来的一种定义不重复数组的数据类型 Array.from是将类数组转换为数组

...是扩展运算符，将set里面的值转化为字符串

Array.prototype.distinct = nums => {

const map = {}

const result = []

for (const n of nums) {

if (!(n in map)) {

map[n] = 1

result.push(n)

}

}

return result

}

[1,2,3,3,4,4].distinct(); //[1,2,3,4]

**排序**

[1,2,3,4].sort ((a, b) => a - b ); // [1, 2,3,4],默认是升序

[1,2,3,4].sort ((a, b) => b - a ); // [4,3,2,1] 降序

sort是js内置的排序方法，参数为一个函数

**最大值**

Math.max(...[1,2,3,4]) //4

Math.max.apply(this,[1,2,3,4]) //4

[1,2,3,4].reduce( (prev, cur,curIndex,arr)=> {

return Math.max(prev,cur);

},0) //4

Math.max()是Math对象内置的方法，参数是字符串；reduce是ES5的数组api, 参数有 函数和默认初始值；函数有四个参数，pre(上一次返回值)，cur(当前值), curIndex(当前 值索引)，arr(当前数组)

**求和**

[1, 2, 3, 4].arr.reduce(function(pre, cur) {

Return prev + cur;

}, 0) //10

**合并**

[1, 2, 3, 4].concat ([5,6]) //[1,2,3,4,5,6]

[...[1, 2, 3, 4], ...[4, 5]] //[1,2,3,4,5,6]

let arrA = [1, 2],

arrB = [3, 4]

Array.prototype.push.apply(arrA, arrB)) //arrA值为[1,2,3,4]

**是否包含某值**

[1,2,3].includes(4) //false

[1,2,3].indexOf(4) //-1 如果存在换回索引

[1, 2, 3].find((item)=>item===3)) //3 如果数组中无值返回undefined

[1, 2, 3].findIndex((item)=>item===3)) //2 如果数组中无值返回-1

includes(), find(), findIndex() 是ES6的api

**类数组转化**

Array.prototype.slice.call(arguments) //arguments是类数组(伪数组) Array.prototype.slice.apply (arguments)

Array.from(arguments)

[...arguments]

类数组：表示有length属性，但是不具备数组的方法

call\apply: 是改变slice里面的this指向arguments,所以arguments也可调用数组的方法

Array.from是将类似数组或课迭代对象创建为数组

...是将类数组扩展为字符串，再定义为数组

**每一项设置值**

[ 1, 2, 3 ].fill(false) //[false, false, false] es6的方法

[ 1, 2, 3 ].map( ()=> 0)

每一项是否满足

[ 1, 2, 3 ].every(item => { return item>2 }) //false

every是ES5的api, 每一项满足则返回true

**有一项满足**

[ 1, 2, 3].some(item =>{ return item>2 }) //true

some是ES5的api, 有一项满足则返回true

**过滤数组**

[ 1, 2, 3].filter(item =>{ return item>2 }) //[3]

filter是ES5的api, 返回满足添加的项的数组

**数组和对象转化**

Object.keys({name:'张三',age:14}) //['name','age']

Object.values({name:'张三',age:14}) //['张三',14]

Object.entries({name:'张三',age:14}) //[[name,'张三'],[age,14]]

Object.fromEntries([name,'张三'],[age,14])

//ES10的api,Chrome不支持 , firebox输出{name:'张三',age:14}

**对象数组**

[{count:1},{count:2},{count:3}].reduce((p, e)=>p+(e.count), 0)

1. **栈：先进后出**

class Stack { constructor() { this.items = []; }

// 入栈

push(element) { this.items.push(element); }

// 出栈

pop() { return this.items.pop(); }

// 末位

get peek() { return this.items[this.items.length - 1]; }

// 是否为空栈

get isEmpty() { return !this.items.length; }

// 长度

get size() { return this.items.length; }

// 清空栈

clear() { this.items = []; } }

// 实例化一个栈

const stack = new Stack(); console.log(stack.isEmpty); // true

// 添加元素

stack.push(5); stack.push(8);

// 读取属性再添加

console.log(stack.peek); // 8

stack.push(11);

console.log(stack.size); // 3

console.log(stack.isEmpty); // false

1. 队列：先进先出

class Queue { constructor(items) { this.items = items || []; }

enqueue(element) { this.items.push(element); }

dequeue() { return this.items.shift(); }

front() { return this.items[0]; }

clear() { this.items = []; }

get size() { return this.items.length; }

get isEmpty() { return !this.items.length; }

print() { console.log(this.items.toString()); } }

const queue = new Queue();

console.log(queue.isEmpty); // true

queue.enqueue("John");

queue.enqueue("Jack");

queue.enqueue("Camila");

console.log(queue.size); // 3

console.log(queue.isEmpty); // false

queue.dequeue(); queue.dequeue();

1. 链表：存贮有序元素的集合

但是不同于数组，每个元素是一个存贮元素本身的节点和指向下一个元素的引用组 成的，要想访问链表中间的元素，需要从起点开始遍历找到所需元素

class Node {

constructor(element) {

this.element = element;

this.next = null;

}

}

// 链表

class LinkedList {

constructor() { this.head = null; this.length = 0; }

// 追加元素

append(element) {

const node = new Node(element);

let current = null;

if (this.head === null) {

this.head = node;

} else {

current = this.head;

while (current.next) {

current = current.next;

}

current.next = node;

}

this.length++;

}

// 任意位置插入元素

insert(position, element) {

if (position >= 0 && position <= this.length) {

const node = new Node(element);

let current = this.head;

let previous = null;

let index = 0;

if (position === 0) {

this.head = node;

} else {

while (index++ < position) {

previous = current;

current = current.next;

}

node.next = current;

previous.next = node;

}

this.length++;

return true;

}

return false;

}

// 移除指定位置元素

removeAt(position) {

// 检查越界值

if (position > -1 && position < length) {

let current = this.head;

let previous = null;

let index = 0;

if (position === 0) {

this.head = current.next;

} else {

while (index++ < position) {

previous = current;

current = current.next;

}

previous.next = current.next;

}

this.length--;

return current.element;

}

return null; }

// 寻找元素下标

findIndex(element) {

let current = this.head;

let index = -1;

while (current) {

if (element === current.element) {

return index + 1;

}

index++;

current = current.next;

}

return -1;

}

// 删除指定文档

remove(element) {

const index = this.findIndex(element);

return this.removeAt(index);

}

isEmpty() { return !this.length; }

size() { return this.length; }

// 转为字符串

toString() {

let current = this.head;

let string = "";

while (current) {

string += ` ${current.element}`;

current = current.next;

}

return string;

}

}

const linkedList = new LinkedList();

console.log(linkedList);

linkedList.append(2);

linkedList.append(6);

linkedList.append(24);

linkedList.append(152);

linkedList.insert(3, 18);

console.log(linkedList);

console.log(linkedList.findIndex(24));

1. 字典：类似对象。以key,value存贮值
2. 二叉树：每个节点最多有两个子树的树结构
3. 设计模式

**简单工厂模式**：又叫静态工厂方法，就是创建对象，并赋予属性和方法

应用：抽取类相同的属性和方法封装到对象上

**工厂方法模式**：对产品类的抽象使其创建业务主要负责用于创建多类产品的实例

应用：创建实例

**原型模式**：设置函数的原型属性

应用：实现继承

**单例模式：**只被允许被实例化一次的类

应用：提供一个命名空间

let singleCase = function(name) {

this.name = name;

};

singleCase.prototype.getName = function() {

return this.name;

}

//获取实例对象

let getInstance = (function() {

var instance = null;

return function(name) {

if(!instance) { //相当于一个一次性阀门，只能实例化一次

instance = new singleCase(name);

}

return instance;

}

}) ();

//测试单例模式的实例，所以one===two

let one = getInstance(“one”);

let two = getInstance(“two”);

**外观模式**：为子系统中的一组接口提供一个一致的界面

应用：简化复杂接口

**适配器模式**：将一个接口转换成客户端需要的接口而不需要去修改客户端代码，使得不 兼容的代码可以一起工作

作用：适配函数参数

**装饰者模式**：不改变原对象的基础上，给对象添加属性或方法

**桥接模式**：将抽象部分与它的实现部分分离，使他们都可以独立地变化

**模块方法模式**：定义一个模块，供以后传不同参数调用

**观察者模式**：解决类与对象，对象与对象之间的耦合

let Observer= (function(){

let \_message={};

return {

//注册接口,

//1.作用:将订阅者注册的消息推入到消息队列

//2.参数:所以要传两个参数,消息类型和处理动作,

//3.消息不存在重新创建,存在将消息推入到执行方法

regist:function(type,fn){

//如果消息不存在,创建

if(typeof \_message[type]==='undefined'){

\_message[type]=[fn];

}else{

//将消息推入到消息的执行动作

\_message[type].push(fn);

}

},

//发布信息接口

//1.作用:观察这发布消息将所有订阅的消息一次执行

//2.参数:消息类型和动作执行传递参数

//3.消息类型参数必须校验

fire:function(type,args){

//如果消息没有注册,则返回

if(!\_message[type]) return;

//定义消息信息

var events={

type:type, //消息类型

args:args||{} //消息携带数据

},

i=0,

len=\_message[type].length;

//遍历消息

for(;i<len;i++){

//依次执行注册消息

\_message[type][i].call(this,events);

}

},

//移除信息接口

//1.作用:将订阅者注销消息从消息队列清除

//2.参数:消息类型和执行的动作

//3.消息参数校验

remove:function(type,fn){

//如果消息动作队列存在

if(\_message[type] instanceof Array){

//从最后一个消息动作序遍历

var i=\_message[type].length-1; for(;i>=0;i--){

//如果存在该动作在消息队列中移除

\_message[type][i]===fn&&\_message[type].splice(i,1);

}

}

}

}

})()

//测试用例

//1.订阅消息

Observer.regist('test',function(e){ console.log(e.type,e.args.msg); })

//2.发布消息

Observer.fire('test',{msg:'传递参数1'});

Observer.fire('test',{msg:'传递参数2'});

Observer.fire('test',{msg:'传递参数3'});

**状态模式**：一个对象状态改变会导致行为变化

作用：解决复杂的if判断

**策略模式：**定义了一系列家族算法，并对每一种算法单独封装起来，让算法之间可 以相互替换，独立于使用算法的客户

**访问模式**：通过继承封装一些该数据类型不具备的属性

作用：让对象具备数组的操作方法

**中介者模式**：设置一个中间层，处理对象之间的交互

1. **HTTP**：HTTP是一个连接客户端、网关和服务器的一个协议

**特点**： 支持客户/服务器模式：可以连接客户端和服务器

简单快速：请求只需传送请求方法，路径和请求主体

灵活：传输数据类型灵活

无连接：请求结束立即断开

无状态：无法记住上一次请求

**解决无状态和无连接：**

无状态：HTTP协议本身无法解决这个状态，只有通过cookie和session将状态做 存贮，通常的场景是登陆状态保持

无连接：可以通过自身属性keep-alive

**请求过程**

HTTP(S) 请求地址 -->DNS解析 -->三次握手 -->发送请求 -->四次握手

**HTTPS:**

https是在http协议的基础上加了个SSL

主要包括：握手（凭证交换和验证）和记录协议（数据进行加密）

缓存：

**按协议分：协议层缓存和非http协议缓存**

协议层缓存：利用http协议头属性值设置

非协议层缓存：利用meta标签的http-equiv属性值Expires，set-cookie

**按缓存分：强缓存和协商缓存**

强缓存：利用 cache-control 和 expires 设置，直接返回一个过期时间，所 以在缓存期间不请求，If-modify-since；

协商缓存：响应头返回 etag 或 last-modified 的哈希值，第二次请求头 If-none-match 或 IF-modify-since 携带上次哈希值，一致则返回 304

1. **状态码**

**1XX（通知）**

**2XX（成功）**

200(成功)、201(服务器创建)、202(服务器接收未处理)、203(非授权信息)、

204(未返回内容)、205(重置内容)、206(部分内容)

**3XX（重定向）**

301(永久移动)、302(临时移动)、303(查看其他位置)、

304(未修改)、 305(使用代理)、307(临时重定向)

**4XX（客户端错误）**

400(错误请求)、401(未授权)、403(禁止)、404(未找到)、

405(方法禁用)、406(不接受)、407（需要代理授权）

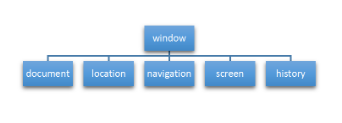
**5XX（服务器错误）**

500(服务器异常)、501（尚未实施）、502（错误网关）、

503（服务不可用）、 504（网关超时）、505（HTTP 版本不受支持）

1. DOM对象
2. BOM对象

BOM(Browser Object Mode)浏览器对象模型，是JavaScript的重要组成部分。它提供了一系列对象用于与浏览器窗口进行交互，这些对象通常称为BOM.



**window对象--BOM核心**

Window，窗口对象，它表示整个浏览器窗口，主要用来操作浏览器窗口。同时，window对象还是ECMAScript中的Global对象，因而所有全局变量和函数都是它的属性，且所有原生的构造函数及其他函数也都存在于它的命名空间之下。

弹框类的方法：

alert(‘提示信息’) confirm(‘确认信息’) prompt(‘弹出输入框’) open(“url地址”, “\_black或\_self”, ”新窗口大小”) close() 关闭当前网页

定时器，清除定时器

setTimeout(函数，时间) 只执行一次

clearTimeout(定时器名称)清除定时器,用于停止执行setTimeout()的函数代码。

setInterval(函数，时间) 无限执行

clearInterval() 方法用于停止setInterval()方法执行的函数代码

cookies用于存储web页面的用户信息

JavaScript可以使用document.cookie属性来创建、读取、及删除cookies.

document.cookie将以字符串的方式返回所有的cookies，

类型格式： cookie1=value; cookie2=value;

**document对象**：它是window对象的一个属性，可以用来处理页面文档。

Location对象：对象用于获得当前页面的地址（URL），并把浏览器重定向到新的页面。

原生JS复习完了，冲冲冲！