是时候撸一波 JS 基础啦,撸熟了  
本文不从传统的问答方式梳理,而是从知识维度梳理,以便形成知识网络;  
包括函数,数组,对象,数据结构,算法,设计模式和 http

**1. 函数**

**1.1函数的3种定义方法**

**1.1.1 函数声明**

//ES5

function getSum(){}

function (){}//匿名函数

//ES6

()=>{}//如果{}内容只有一行{}和return关键字可省,

复制代码

**1.1.2 函数表达式(函数字面量)**

//ES5

var sum=function(){}

//ES6

let sum=()=>{}//如果{}内容只有一行{}和return关键字可省,

复制代码

**1.1.3 构造函数**

const sum = new Function('a', 'b' , 'return a + b')

复制代码

**1.1.4 三种方法的对比**

1.函数声明有预解析,而且函数声明的优先级高于变量; 2.使用Function构造函数定义函数的方式是一个函数表达式,这种方式会导致解析两次代码，影响性能。第一次解析常规的JavaScript代码，第二次解析传入构造函数的字符串

**1.2.ES5中函数的4种调用**

在ES5中函数内容的this指向和调用方法有关

**1.2.1 函数调用模式**

包括函数名()和匿名函数调用,this指向window

function getSum() {

console.log(this) //这个属于函数名调用，this指向window

}

getSum()

(function() {

console.log(this) //匿名函数调用，this指向window

})()

var getSum=function() {

console.log(this) //实际上也是函数名调用，window

}

getSum()

复制代码

**1.2.2 方法调用**

对象.方法名(),this指向对象

var objList = {

name: 'methods',

getSum: function() {

console.log(this) //objList对象

}

}

objList.getSum()

复制代码

**1.2.3 构造器调用**

new 构造函数名(),this指向实例化的对象

function Person() {

console.log(this); //是构造函数调用，指向实例化的对象personOne

}

var personOne = new Person();

复制代码

**1.2.4 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

function foo() {

console.log(this);

}

foo.apply('我是apply改变的this值');//我是apply改变的this值

foo.call('我是call改变的this值');//我是call改变的this值

复制代码

**1.3 ES6中函数的调用**

箭头函数不可以当作构造函数使用，也就是不能用new命令实例化一个对象，否则会抛出一个错误 箭头函数的this是和定义时有关和调用无关 调用就是函数调用模式

(() => {

console.log(this)//window

})()

let arrowFun = () => {

console.log(this)//window

}

arrowFun()

let arrowObj = {

arrFun: function() {

(() => {

console.log(this)//this指向的是arrowObj对象

})()

}

}

arrowObj.arrFun();

复制代码

**1.4.call,apply和bind**

1.IE5之前不支持call和apply,bind是ES5出来的; 2.call和apply可以调用函数,改变this,实现继承和借用别的对象的方法;

**1.4.1 call和apply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.....])

**1.4.2 call和apply用法**

1.间接调用函数,改变作用域的this值 2.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

复制代码

3.两个函数实现继承

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

复制代码

4.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')

复制代码

5.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

复制代码

6.求数组最大值

Math.max.apply(null,arr)

复制代码

7.判断字符类型

Object.prototype.toString.call({})

复制代码

**1.4.3 bind**

bind是function的一个函数扩展方法，bind以后代码重新绑定了func内部的this指向,不会调用方法,不兼容IE8

var name = '李四'

var foo = {

name: "张三",

logName: function(age) {

console.log(this.name, age);

}

}

var fooNew = foo.logName;

var fooNewBind = foo.logName.bind(foo);

fooNew(10)//李四,10

fooNewBind(11)//张三,11 因为bind改变了fooNewBind里面的this指向

复制代码

**1.4.4 call,apply和bind原生实现**

call实现:

Function.prototype.newCall = function(context, ...parameter) {

if (typeof context === 'object' || typeof context === 'function') {

context = context || window

} else {

context = Object.create(null)

}

context.fn = this;

context.fn(...parameter);

delete context.fn;

}

let person = {

name: 'Abiel'

}

function sayHi(age,sex) {

console.log(this.name, age, sex);

}

sayHi.newCall (person, 25, '男'); // Abiel 25 男

复制代码

apply实现:

Function.prototype.newApply = function(context, parameter) {

if (typeof context === 'object' || typeof context === 'function') {

context = context || window

} else {

context = Object.create(null)

}

let fn = Symbol()

context[fn] = this

context[fn](parameter);

delete context[fn]

}

复制代码

bind实现:

Function.prototype.bind = function (context,...innerArgs) {

var me = this

return function (...finnalyArgs) {

return me.call(context,...innerArgs,...finnalyArgs)

}

}

let person = {

name: 'Abiel'

}

function sayHi(age,sex) {

console.log(this.name, age, sex);

}

let personSayHi = sayHi.bind(person, 25)

personSayHi('男')

复制代码

**1.4.5 三者异同**

同:都是改变this指向,都可接收参数 异:bind和call是接收单个参数,apply是接收数组

**1.5.函数的节流和防抖**

| **类型** | **概念** | **应用** |
| --- | --- | --- |
| 节流 | 事件触发后每隔一段时间触发一次,可触发多次 | scroll,resize事件一段时间触发多次 |
| 防抖 | 事件触发动作完成后一段时间触发一次 | scroll,resize事件触发完后一段时间触发 |

节流:

**1.5.1 节流**

let throttle = function(func, delay) {

let timer = null;

return ()=> {

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

复制代码

**1.5.2 防抖**

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(()=> {

fn.apply(this, arguments);

timeout = null;

}, wait);

};

}

// 处理函数

function handle() {

console.log(Math.random());

}

// 滚动事件

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

复制代码

**1.6.原型链**

**1.6.1 定义**

对象继承属性的一个链条

**1.6.2构造函数,实例与原型对象的关系**

var Person = function (name) { this.name = name; }//person是构造函数

var o3personTwo = new Person('personTwo')//personTwo是实例

复制代码

原型对象都有一个默认的constructor属性指向构造函数

**1.6.3 创建实例的方法**

1.字面量

let obj={'name':'张三'}

复制代码

2.Object构造函数创建

let Obj=new Object()

Obj.name='张三'

复制代码

3.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');

复制代码

4.使用构造函数创建对象

function Person(name){

this.name = name;

}

var person1 = new Person('张三');

复制代码

**1.6.4 new运算符**

1.创了一个新对象;  
2.this指向构造函数;  
3.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  
4.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 (k && typeof k === 'object') {//判断k的类型是不是对象

return k;&emsp;//是，返回k

} else {

return o;//不是返回返回构造函数的执行结果

}

}

复制代码

**1.6.5 对象的原型链**

**1.7 继承的方式**

JS是一门弱类型动态语言,封装和继承是他的两大特性

**1.7.1 原型链继承**

将父类的实例作为子类的原型 1.代码实现 定义父类:

// 定义一个动物类

function Animal (name) {

// 属性

this.name = name || 'Animal';

// 实例方法

this.sleep = function(){

console.log(this.name + '正在睡觉！');

}

}

// 原型方法

Animal.prototype.eat = function(food) {

console.log(this.name + '正在吃：' + food);

};

复制代码

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

复制代码

2.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

**1.7.2 构造继承**

实质是利用call来改变Cat中的this指向 1.代码实现 子类:

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

复制代码

2.优缺点 可以实现多继承,不能继承原型属性/方法

**1.7.3 实例继承**

为父类实例添加新特性，作为子类实例返回 1.代码实现 子类

function Cat(name){

var instance = new Animal();

instance.name = name || 'Tom';

return instance;

}

复制代码

2.优缺点 不限制调用方式,但不能实现多继承

**1.7.4 拷贝继承**

将父类的属性和方法拷贝一份到子类中 1.子类:

function Cat(name){

var animal = new Animal();

for(var p in animal){

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

}

Cat.prototype.name = name || 'Tom';

}

复制代码

2.优缺点 支持多继承,但是效率低占用内存

**1.7.5 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作为子类原型，实现函数复用 1.子类:

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

Cat.prototype = new Animal();

Cat.prototype.constructor = Cat;

复制代码

**1.7.6 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();

})();

复制代码

**1.7.7 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();

复制代码

**1.8.高阶函数**

**1.8.1定义**

函数的参数是函数或返回函数

**1.8.2 常见的高阶函数**

map,reduce,filter,sort

**1.8.3 柯里化**

1.定义: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)

复制代码

2.代码实现:

const currying = fn => {

const len = fn.length

return function curr (...args1) {

if (args1.length >= len) {

return fn(...args1)

}

return (...args2) => curr(...args1, ...args2)

}

}

复制代码

**1.8.4 反柯里化**

1.定义:

obj.func(arg1, arg2)=>func(obj, arg1, arg2)

复制代码

2.代码实现:

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"));

复制代码

**1.8.5偏函数**

1.定义:指定部分参数来返回一个新的定制函数的形式 2.例子:

function foo(a, b, c) {

return a + b + c;

}

function func(a, b) {

return foo(a,b,8);

}

复制代码

**2.对象**

**2.1.对象的声明方法**

**2.1.1 字面量**

var test2 = {x:123,y:345};

console.log(test2);//{x:123,y:345};

console.log(test2.x);//123

console.log(test2.\_\_proto\_\_.x);//undefined

console.log(test2.\_\_proto\_\_.x === test2.x);//false

复制代码

**2.1.2 构造函数**

var test1 = new Object({x:123,y:345});

console.log(test1);//{x:123,y:345}

console.log(test1.x);//123

console.log(test1.\_\_proto\_\_.x);//undefined

console.log(test1.\_\_proto\_\_.x === test1.x);//false

复制代码

new的作用: 1.创了一个新对象; 2.this指向构造函数; 3.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

**2.1.3 内置方法**

Obejct.create(obj,descriptor),obj是对象,describe描述符属性(可选)

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

console.log(test);//{}

console.log(test.x);//123

console.log(test.\_\_proto\_\_.x);//3

console.log(test.\_\_proto\_\_.x === test.x);//true

复制代码

**2.1.4 三种方法的优缺点**

1.功能:都能实现对象的声明,并能够赋值和取值  
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,属性分类见下面  
4.属性读取:Object.getOwnPropertyDescriptor()或getOwnPropertyDescriptor()  
5.属性设置:Object.definePropertype或Object.defineProperties

**2.2.对象的属性**

**2.2.1 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)

2.访问器属性2个特性: get(获取),set(设置)

3.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]].

**2.2.2 属性描述符**

1.定义:将一个属性的所有特性编码成一个对象返回 2.描述符的属性有:数据属性和访问器属性 3.使用范围: 作为方法Object.defineProperty, Object.getOwnPropertyDescriptor, Object.create的第二个参数,

**2.2.3 属性描述符的默认值**

1.访问对象存在的属性

| **特性名** | **默认值** |
| --- | --- |
| value | 对应属性值 |
| get | 对应属性值 |
| set | undefined |
| writable | true |
| enumerable | true |
| configurable | true |

所以通过上面三种声明方法已存在的属性都是有这些默认描述符 2.访问对象不存在的属性

| **特性名** | **默认值** |
| --- | --- |
| value | undefined |
| get | undefined |
| set | undefined |
| writable | false |
| enumerable | false |
| configurable | false |

**2.2.3 描述符属性的使用规则**

get,set与wriable,value是互斥的,如果有交集设置会报错

**2.2.4 属性定义**

1.定义属性的函数有两个:Object.defineProperty和Object.defineProperties.例如: Object.defineProperty(obj, propName, desc)

2.在引擎内部,会转换成这样的方法调用: obj.[[DefineOwnProperty]](propName, desc, true)

**2.2.5 属性赋值**

1.赋值运算符(=)就是在调用[[Put]].比如: obj.prop = v;

2.在引擎内部,会转换成这样的方法调用: obj.[[Put]]("prop", v, isStrictModeOn)

**2.2.6 判断对象的属性**

| **名称** | **含义** | **用法** |
| --- | --- | --- |
| in | 如果指定的属性在指定的对象或其原型链中，则in 运算符返回true | 'name' in test //true |
| hasOwnProperty() | 只判断自身属性 | test.hasOwnProperty('name') //true |
| .或[] | 对象或原型链上不存在该属性，则会返回undefined | test.name //"lei" test["name"] //"lei" |

**2.3.Symbol**

**2.3.1概念**

是一种数据类型; 不能new,因为Symbol是一个原始类型的值，不是对象。

**2.3.2 定义方法**

Symbol(),可以传参 var s1 = Symbol(); var s2 = Symbol(); s1 === s2 // false

// 有参数的情况

var s1 = Symbol("foo");

var s2 = Symbol("foo");

s1 === s2 // false

复制代码

**2.3.3 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!"

复制代码

3.作为对象属性名时，不能用点运算符,可以用[]

let a = {};

let name = Symbol();

a.name = 'lili';

a[name] = 'lucy';

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

复制代码

4.遍历不会被for...in、for...of和Object.keys()、Object.getOwnPropertyNames()取到该属性

**2.3.4 Symbol.for**

1.定义: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 2.举例:

var s1 = Symbol.for('foo');

var s2 = Symbol.for('foo');

s1 === s2 // true

复制代码

**2.3.5 Symbol.keyFor**

1.定义:返回一个已登记的Symbol类型值的key 2.举例:

var s1 = Symbol.for("foo");

Symbol.keyFor(s1) // "foo"

var s2 = Symbol("foo");

Symbol.keyFor(s2) // undefined

复制代码

**2.4.遍历**

**2.4.1 一级对象遍历方法**

| **方法** | **特性** |
| --- | --- |
| for ... in | 遍历对象自身的和继承的可枚举属性(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,包括对象自身的所有可枚举属性(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、可枚举和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属性) |

总结:1.只有Object.getOwnPropertySymbols(obj)和Reflect.ownKeys(obj)可以拿到Symbol属性 2.只有Reflect.ownKeys(obj)可以拿到不可枚举属性

**2.4.2 多级对象遍历**

数据模型:

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: []

},

];

复制代码

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);

复制代码

**2.5.深度拷贝**

**2.5.1 Object.assign**

1.定义:将源对象（source）的所有可枚举属性，复制到目标对象（target） 2.用法:

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);

复制代码

3.注意: 这个是伪深度拷贝,只能拷贝第一层

**2.5.2 JSON.stringify**

1.原理:是将对象转化为字符串,而字符串是简单数据类型

**2.5.3 递归拷贝**

function deepClone(source){

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

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

targetObj[keys] = source[keys].constructor === Array ? [] : {};

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

}else{ // 如果不是，就直接赋值

targetObj[keys] = source[keys];

}

}

}

return targetObj;

}

复制代码

**2.6.数据拦截**

定义:利用对象内置方法,设置属性,进而改变对象的属性值

**2.6.1 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

复制代码

4.拦截对象的两种情况:

let obj = {name:'',age:'',sex:'' },

defaultName = ["这是姓名默认值1","这是年龄默认值1","这是性别默认值1"];

Object.keys(obj).forEach(key => {

Object.defineProperty(obj, key, {

get() {

return defaultName;

},

set(value) {

defaultName = value;

}

});

});

console.log(obj.name);

console.log(obj.age);

console.log(obj.sex);

obj.name = "这是改变值1";

console.log(obj.name);

console.log(obj.age);

console.log(obj.sex);

let objOne={},defaultNameOne="这是默认值2";

Object.defineProperty(obj, 'name', {

get() {

return defaultNameOne;

},

set(value) {

defaultNameOne = value;

}

});

console.log(objOne.name);

objOne.name = "这是改变值2";

console.log(objOne.name);

复制代码

5.拦截数组变化的情况

let a={};

bValue=1;

Object.defineProperty(a,"b",{

set:function(value){

bValue=value;

console.log("setted");

},

get:function(){

return bValue;

}

});

a.b;//1

a.b=[];//setted

a.b=[1,2,3];//setted

a.b[1]=10;//无输出

a.b.push(4);//无输出

a.b.length=5;//无输出

a.b;//[1,10,3,4,undefined];

复制代码

结论:defineProperty无法检测数组索引赋值,改变数组长度的变化; 但是通过数组方法来操作可以检测到

多级嵌套对象监听

let info = {};

function observe(obj) {

if (!obj || typeof obj !== "object") {

return;

}

for (var i in obj) {

definePro(obj, i, obj[i]);

}

}

function definePro(obj, key, value) {

observe(value);

Object.defineProperty(obj, key, {

get: function() {

return value;

},

set: function(newval) {

console.log("检测变化", newval);

value = newval;

}

});

}

definePro(info, "friends", { name: "张三" });

info.friends.name = "李四";

复制代码

6.存在的问题

不能监听数组索引赋值和改变长度的变化

必须深层遍历嵌套的对象,因为defineProterty只能劫持对象的属性,因此我们需要对每个对象的每个属性进行遍历，如果属性值也是对象那么需要深度遍历,显然能劫持一个完整的对象是更好的选择

复制代码

**2.6.2 proxy**

1.ES6出来的方法,实质是对对象做了一个拦截,并提供了13个处理方法

2.两个参数:对象和行为函数

let handler = {

get(target, key, receiver) {

console.log("get", key);

return Reflect.get(target, key, receiver);

},

set(target, key, value, receiver) {

console.log("set", key, value);

return Reflect.set(target, key, value, receiver);

}

};

let proxy = new Proxy(obj, handler);

proxy.name = "李四";

proxy.age = 24;

复制代码

涉及到多级对象或者多级数组

//传递两个参数，一个是object, 一个是proxy的handler

//如果是不是嵌套的object，直接加上proxy返回，如果是嵌套的object，那么进入addSubProxy进行递归。

function toDeepProxy(object, handler) {

if (!isPureObject(object)) addSubProxy(object, handler);

return new Proxy(object, handler);

//这是一个递归函数，目的是遍历object的所有属性，如果不是pure object,那么就继续遍历object的属性的属性，如果是pure object那么就加上proxy

function addSubProxy(object, handler) {

for (let prop in object) {

if ( typeof object[prop] == 'object') {

if (!isPureObject(object[prop])) addSubProxy(object[prop], handler);

object[prop] = new Proxy(object[prop], handler);

}

}

object = new Proxy(object, handler)

}

//是不是一个pure object,意思就是object里面没有再嵌套object了

function isPureObject(object) {

if (typeof object!== 'object') {

return false;

} else {

for (let prop in object) {

if (typeof object[prop] == 'object') {

return false;

}

}

}

return true;

}

}

let object = {

name: {

first: {

four: 5,

second: {

third: 'ssss'

}

}

},

class: 5,

arr: [1, 2, {arr1:10}],

age: {

age1: 10

}

}

//这是一个嵌套了对象和数组的数组

let objectArr = [{name:{first:'ss'}, arr1:[1,2]}, 2, 3, 4, 5, 6]

//这是proxy的handler

let handler = {

get(target, property) {

console.log('get:' + property)

return Reflect.get(target, property);

},

set(target, property, value) {

console.log('set:' + property + '=' + value);

return Reflect.set(target, property, value);

}

}

//变成监听对象

object = toDeepProxy(object, handler);

objectArr = toDeepProxy(objectArr, handler);

//进行一系列操作

console.time('pro')

objectArr.length

objectArr[3];

objectArr[2]=10

objectArr[0].name.first = 'ss'

objectArr[0].arr1[0]

object.name.first.second.third = 'yyyyy'

object.class = 6;

object.name.first.four

object.arr[2].arr1

object.age.age1 = 20;

console.timeEnd('pro')

复制代码

3.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

**2.6.3 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;

3.利用defineProterty实现双向数据绑定(vue2.x采用的核心) 4.利用proxy实现双向数据绑定(vue3.x会采用)

**3.数组**

数组基本上考察数组方法多一点,所以这里就单纯介绍常见的场景数组的方法,还有很多场景后续补充;  
本文主要从应用来讲数组api的一些骚操作;  
如一行代码扁平化n维数组、数组去重、求数组最大值、数组求和、排序、对象和数组的转化等；  
上面这些应用场景你可以用一行代码实现？

**3.1 扁平化n维数组**

1.终极篇

[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]

复制代码

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

2.开始篇

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]

复制代码

实质是利用递归和数组合并方法concat实现扁平

**3.2 去重**

1.终极篇

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

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

复制代码

set是ES6新出来的一种一种定义不重复数组的数据类型 Array.from是将类数组转化为数组 ...是扩展运算符,将set里面的值转化为字符串 2.开始篇

Array.prototype.distinct = function() {

const map = {}

const result = []

for (const n of this) {

if (!(n in map)) {

map[n] = 1

result.push(n)

}

}

return result

}

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

复制代码

取新数组存值,循环两个数组值相比较

**3.3排序**

1.终极篇

[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

复制代码

sort是js内置的排序方法,参数为一个函数 2.开始篇 冒泡排序:

Array.prototype.bubleSort=function () {

let arr=this,

len = arr.length;

for (let outer = len; outer >= 2; outer--) {

for (let inner = 0; inner <= outer - 1; inner++) {

if (arr[inner] > arr[inner + 1]) {

//升序

[arr[inner], arr[inner + 1]] = [arr[inner + 1], arr[inner]];

console.log([arr[inner], arr[inner + 1]]);

}

}

}

return arr;

}

[1,2,3,4].bubleSort() //[1,2,3,4]

复制代码

选择排序

Array.prototype.selectSort=function () {

let arr=this,

len = arr.length;

for (let i = 0, len = arr.length; i < len; i++) {

for (let j = i, len = arr.length; j < len; j++) {

if (arr[i] > arr[j]) {

[arr[i], arr[j]] = [arr[j], arr[i]];

}

}

}

return arr;

}

[1,2,3,4].selectSort() //[1,2,3,4]

复制代码

**3.4最大值**

1.终极篇

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

复制代码

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(上一次的返回值),cur(当前值),curIndex(当前值索引),arr(当前数组)

2.开始篇 先排序再取值

**3.5求和**

1.终极篇

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

return prev + cur;

},0) //10

复制代码

2.开始篇

function sum(arr) {

var len = arr.length;

if(len == 0){

return 0;

} else if (len == 1){

return arr[0];

} else {

return arr[0] + sum(arr.slice(1));

}

}

sum([1,2,3,4]) //10

复制代码

利用slice截取改变数组,再利用递归求和

**3.6合并**

1.终极篇

[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]

复制代码

2.开始篇

let arr=[1,2,3,4];

[5,6].map(item=>{

arr.push(item)

})

//arr值为[1,2,3,4,5,6],注意不能直接return出来,return后只会返回[5,6]

复制代码

**3.7判断是否包含值**

1.终极篇

[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

复制代码

includes(),find(),findIndex()是ES6的api

2.开始篇

[1,2,3].some(item=>{

return item===3

}) //true 如果不包含返回false

复制代码

**3.8类数组转化**

1.终极篇

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

Array.prototype.slice.apply(arguments)

Array.from(arguments)

[...arguments]

复制代码

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是将类似数组或可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

2.开始篇

Array.prototype.slice = function(start,end){

var result = new Array();

start = start || 0;

end = end || this.length; //this指向调用的对象，当用了call后，能够改变this的指向，也就是指向传进来的对象，这是关键

for(var i = start; i < end; i++){

result.push(this[i]);

}

return result;

}

复制代码

**3.9每一项设置值**

1.终极篇

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

复制代码

fill是ES6的方法 2.开始篇

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

复制代码

**3.10每一项是否满足**

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

复制代码

every是ES5的api,每一项满足返回 true

**3.11有一项满足**

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

复制代码

some是ES5的api,有一项满足返回 true

**3.12.过滤数组**

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

复制代码

filter是ES5的api,返回满足添加的项的数组

**3.13对象和数组转化**

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}

复制代码

**3.14 对象数组**

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

复制代码

**4.数据结构篇**

数据结构是计算机存储、组织数据的方式,算法是系统描述解决问题的策略。了解基本的数据结构和算法可以提高代码的性能和质量。  
也是程序猿进阶的一个重要技能。  
手撸代码实现栈,队列,链表,字典,二叉树,动态规划和贪心算法

**4.1 栈**

栈的特点：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

复制代码

**4.2 队列**

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();

复制代码

**4.3 链表**

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引用组成  
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));

复制代码

**4.4 字典**

字典：类似对象，以key，value存贮值 class Dictionary { constructor() { this.items = {}; }

set(key, value) {

this.items[key] = value;

}

get(key) {

return this.items[key];

}

remove(key) {

delete this.items[key];

}

get keys() {

return Object.keys(this.items);

}

get values() {

/\*

也可以使用ES7中的values方法

return Object.values(this.items)

\*/

// 在这里我们通过循环生成一个数组并输出

return Object.keys(this.items).reduce((r, c, i) => {

r.push(this.items[c]);

return r;

}, []);

}

}

const dictionary = new Dictionary();

dictionary.set("Gandalf", "gandalf@email.com");

dictionary.set("John", "johnsnow@email.com");

dictionary.set("Tyrion", "tyrion@email.com");

console.log(dictionary);

console.log(dictionary.keys);

console.log(dictionary.values);

console.log(dictionary.items);

复制代码

**4.5 二叉树**

特点：每个节点最多有两个子树的树结构 class NodeTree { constructor(key) { this.key = key; this.left = null; this.right = null; } }

class BinarySearchTree {

constructor() {

this.root = null;

}

insert(key) {

const newNode = new NodeTree(key);

const insertNode = (node, newNode) => {

if (newNode.key < node.key) {

if (node.left === null) {

node.left = newNode;

} else {

insertNode(node.left, newNode);

}

} else {

if (node.right === null) {

node.right = newNode;

} else {

insertNode(node.right, newNode);

}

}

};

if (!this.root) {

this.root = newNode;

} else {

insertNode(this.root, newNode);

}

}

//访问树节点的三种方式:中序,先序,后序

inOrderTraverse(callback) {

const inOrderTraverseNode = (node, callback) => {

if (node !== null) {

inOrderTraverseNode(node.left, callback);

callback(node.key);

inOrderTraverseNode(node.right, callback);

}

};

inOrderTraverseNode(this.root, callback);

}

min(node) {

const minNode = node => {

return node ? (node.left ? minNode(node.left) : node) : null;

};

return minNode(node || this.root);

}

max(node) {

const maxNode = node => {

return node ? (node.right ? maxNode(node.right) : node) : null;

};

return maxNode(node || this.root);

}

}

const tree = new BinarySearchTree();

tree.insert(11);

tree.insert(7);

tree.insert(5);

tree.insert(3);

tree.insert(9);

tree.insert(8);

tree.insert(10);

tree.insert(13);

tree.insert(12);

tree.insert(14);

tree.inOrderTraverse(value => {

console.log(value);

});

console.log(tree.min());

console.log(tree.max());

复制代码

**5.算法篇**

**5.1 冒泡算法**

冒泡排序，选择排序，插入排序，此处不做赘述.

**5.2 斐波那契**

特点：第三项等于前面两项之和

function fibonacci(num) {

if (num === 1 || num === 2) {

return 1

}

return fibonacci(num - 1) + fibonacci(num - 2)

}

复制代码

**5.3 动态规划**

特点：通过全局规划,将大问题分割成小问题来取最优解  
案例：最少硬币找零  
美国有以下面额(硬币）：d1=1, d2=5, d3=10, d4=25  
如果要找36美分的零钱，我们可以用1个25美分、1个10美分和1个便士（ 1美分)

class MinCoinChange {

constructor(coins) {

this.coins = coins

this.cache = {}

}

makeChange(amount) {

if (!amount) return []

if (this.cache[amount]) return this.cache[amount]

let min = [], newMin, newAmount

this.coins.forEach(coin => {

newAmount = amount - coin

if (newAmount >= 0) {

newMin = this.makeChange(newAmount)

}

if (newAmount >= 0 &&

(newMin.length < min.length - 1 || !min.length) &&

(newMin.length || !newAmount)) {

min = [coin].concat(newMin)

}

})

return (this.cache[amount] = min)

}

}

const rninCoinChange = new MinCoinChange([1, 5, 10, 25])

console.log(rninCoinChange.makeChange(36))

// [1, 10, 25]

const minCoinChange2 = new MinCoinChange([1, 3, 4])

console.log(minCoinChange2.makeChange(6))

// [3, 3]

复制代码

**5.4 贪心算法**

特点：通过最优解来解决问题 用贪心算法来解决2.3中的案例

class MinCoinChange2 {

constructor(coins) {

this.coins = coins

}

makeChange(amount) {

const change = []

let total = 0

this.coins.sort((a, b) => a < b).forEach(coin => {

if ((total + coin) <= amount) {

change.push(coin)

total += coin

}

})

return change

}

}

const rninCoinChange2 = new MinCoinChange2 ( [ 1, 5, 10, 25])

console.log (rninCoinChange2. makeChange (36))

复制代码

**6 设计模式**

设计模式如果应用到项目中，可以实现代码的复用和解耦，提高代码质量。 本文主要介绍14种设计模式 写UI组件,封装框架必备

**6.1 简单工厂模式**

1.定义：又叫静态工厂方法，就是创建对象，并赋予属性和方法  
2.应用：抽取类相同的属性和方法封装到对象上  
3.代码：

let UserFactory = function (role) {

function User(opt) {

this.name = opt.name;

this.viewPage = opt.viewPage;

}

switch (role) {

case 'superAdmin':

return new User(superAdmin);

break;

case 'admin':

return new User(admin);

break;

case 'user':

return new User(user);

break;

default:

throw new Error('参数错误, 可选参数:superAdmin、admin、user')

}

}

//调用

let superAdmin = UserFactory('superAdmin');

let admin = UserFactory('admin')

let normalUser = UserFactory('user')

//最后得到角色,可以调用

复制代码

**6.2工厂方法模式**

1.定义：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  
2.应用:创建实例  
3.代码:

var Factory=function(type,content){

if(this instanceof Factory){

var s=new this[type](content);

return s;

}else{

return new Factory(type,content);

}

}

//工厂原型中设置创建类型数据对象的属性

Factory.prototype={

Java:function(content){

console.log('Java值为',content);

},

PHP:function(content){

console.log('PHP值为',content);

},

Python:function(content){

console.log('Python值为',content);

},

}

//测试用例

Factory('Python','我是Python');

复制代码

**6.3原型模式**

1.定义:设置函数的原型属性 2.应用:实现继承 3.代码:

function Animal (name) {

// 属性

this.name = name || 'Animal';

// 实例方法

this.sleep = function(){

console.log(this.name + '正在睡觉！');

}

}

// 原型方法

Animal.prototype.eat = function(food) {

console.log(this.name + '正在吃：' + food);

};

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

复制代码

**6.4单例模式**

1.定义:只允许被实例化依次的类 2.应用:提供一个命名空间 3.代码:

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;

}

})();

// 测试单体模式的实例,所以one===two

let one = getInstance("one");

let two = getInstance("two");

复制代码

**6.5外观模式**

1.定义:为子系统中的一组接口提供一个一致的界面 2.应用:简化复杂接口 3.代码: [外观模式](https://www.cnblogs.com/linda586586/p/4237093.html)

**6.6适配器模式**

1.定义: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 2.应用:适配函数参数 3.代码: [适配器模式](https://www.cnblogs.com/TomXu/archive/2012/04/11/2435452.html)

**6.7装饰者模式**

1.定义:不改变原对象的基础上,给对象添加属性或方法 2.代码

let decorator=function(input,fn){

//获取事件源

let input=document.getElementById(input);

//若事件源已经绑定事件

if(typeof input.onclick=='function'){

//缓存事件源原有的回调函数

let oldClickFn=input.onclick;

//为事件源定义新事件

input.onclick=function(){

//事件源原有回调函数

oldClickFn();

//执行事件源新增回调函数

fn();

}

}else{

//未绑定绑定

input.onclick=fn;

}

}

//测试用例

decorator('textInp',function(){

console.log('文本框执行啦');

})

decorator('btn',function(){

console.log('按钮执行啦');

})

复制代码

**6.8桥接模式**

1.定义:将抽象部分与它的实现部分分离，使它们都可以独立地变化 2.代码 [桥接模式](https://www.cnblogs.com/TomXu/archive/2012/04/19/2437321.html)

**6.9模块方法模式**

1.定义:定义一个模板,供以后传不同参数调用 2.代码: [模块方法模式](https://blog.csdn.net/namechenfl/article/details/80685741)

**6.10.观察者模式**

1.作用:解决类与对象,对象与对象之间的耦合 2.代码:

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'});

复制代码

**6.11状态模式**

1.定义:一个对象状态改变会导致行为变化 2.作用:解决复杂的if判断 3.代码 [状态模式](https://www.jianshu.com/p/ec9b9a1cd148)

**6.12策略模式**

1.定义: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 2.代码 [策略模式](https://www.cnblogs.com/Medeor/p/5001841.html)

**6.13.访问模式**

1.定义:通过继承封装一些该数据类型不具备的属性, 2.作用:让对象具备数组的操作方法 3.代码: [访问者模式](https://blog.csdn.net/itpinpai/article/details/51644922)

**6.14中介者模式**

1.定义:设置一个中间层,处理对象之间的交互 2.代码: [中介者模式](https://www.cnblogs.com/xiaohuochai/p/8042198.html)

**7. HTTP**

**1.1 什么是 HTTP**

HTTP 是一个连接客户端，网关和服务器的一个协议。

**7.2 特点**

支持客户/服务器模式：可以连接客户端和服务端；  
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。

**7.3 怎么解决无状态和无连接**

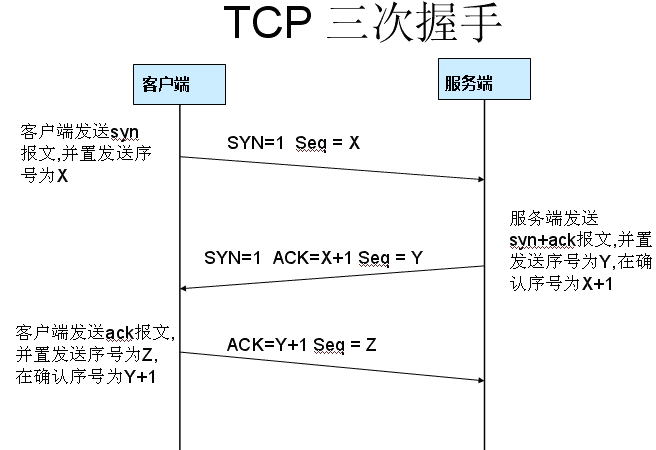
无状态：HTTP 协议本身无法解决这个状态，只有通过 cookie 和 session 将状态做贮存，常见的场景是登录状态保持；

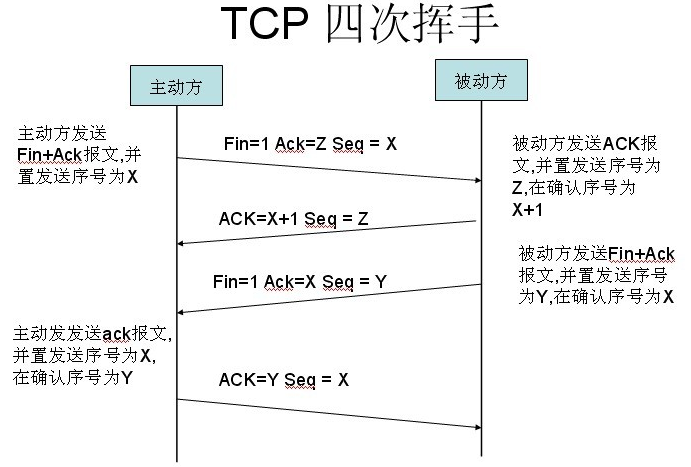
无连接：可以通过自身属性 Keep-Alive。

**7.4 请求过程**

HTTP(S) 请求地址 → DNS 解析 → 三次握手 → 发送请求 → 四次挥手

三次握手过程图片来源 CSDN）

在这里插入图片描述

1. 四次挥手过（图片来源 CSDN） 在这里插入图片描述

**7.5 HTTP 0.9~3.0 对比**

**7.5.1 HTTP 0.9**

只允许客户端发送 GET 这一种请求；  
且不支持请求头，协议只支持纯文本；  
无状态性，每个访问独立处理，完成断开；  
无状态码。

**7.5.2 HTTP 1.0**

有身份认证，三次握手； 请求与响应支持头域； 请求头内容；

| **属性名** | **含义** |
| --- | --- |
| Accept | 可接受的 MIME 类型 |
| Accept-Encoding | 数据可解码的格式 |
| Accept-Language | 可接受语言 |
| Connection | 值 keep-alive 是长连接 |
| Host | 主机和端口 |
| Pragma | 是否缓存,指定 no-cache 返回刷新 |
| Referer | 页面路由 |
| If-Modified-Since | 值为时间 |

响应头内容；

| **属性名** | **含义** |
| --- | --- |
| Connection | 值 keep-alive 是长连接 |
| Content-Type | 返回文档类型,常见的值有 text/plain,text/html,text/json |
| Date | 消息发送的时间 |
| Server | 服务器名字 |
| Last-Modified | 值为时间,s 返回的最后修改时间 |
| Expires | 缓存过期时间,b 和 s 时间做对比 |

注意

expires 是响应头内容，返回一个固定的时间,缺陷是时间到了服务器要重新设置;  
请求头中如果有 If-Modified-Since，服务器会将时间与 last-modified 对比，相同返回 304;  
响应对象以一个响应状态行开始;  
响应对象不只限于超文本;  
支持 GET、HEAD、POST 方法;  
有状态码;  
支持长连接（但默认还是使用短连接）、缓存机制以及身份认证。

**7.5.3 HTTP 1.1**

请求头增加 Cache-Control

| **属性名** | **含义** |
| --- | --- |
| Cache-Control | 在1.1 引入的方法,指定请求和响应遵循的缓存机制,值有:public(b 和 s 都缓存),private(b 缓存),no-cache(不缓存),no-store(不缓存),max-age(缓存时间,s 为单位),min-fresh(最小更新时间),max-age=3600 |
| If-None-Match | 上次请求响应头返回的 etag 值响应头增加 Cache-Control，表示所有的缓存机制是否可以缓存及哪种类型 etag 返回的哈希值,第二次请求头携带去和服务器值对比 |

注意

Cache-Control 的 max-age 返回是缓存的相对时间 Cache-Control 优先级比 expires 高 缺点：不能第一时间拿到最新修改文件

**7.5.4 HTTP 2.0**

采用二进制格式传输;  
多路复用，其实就是将请求数据分成帧乱序发送到 TCP 中。TCP 只能有一个 steam，所以还是会阻塞;  
报头压缩;  
服务器推送主动向 B 端发送静态资源，避免往返延迟。

**7.5.5 HTTP 3.0**

1.是基于 QUIC 协议，基于 UDP  
2.特点:  
自定义连接机制：TCP 以 IP/端口标识,变化重新连接握手，UDP 是一 64 位 ID 标识，是无连接；  
自定义重传机制：TCP 使用序号和应答传输，QUIC 是使用递增序号传输； 无阻塞的多路复用：同一条 QUIC 可以创建多个 steam。

**7.5.6 HTTPS**

1.https 是在 http 协议的基础上加了个 SSL；  
2.主要包括：握手(凭证交换和验证)和记录协议(数据进行加密)。

**7.5.7 缓存**

1.按协议分：协议层缓存和非 http 协议缓存：  
1.1协议层缓存：利用 http 协议头属性值设置；  
1.2非协议层缓存：利用 meta 标签的 http-equiv 属性值 Expires,set-cookie。

2.按缓存分：强缓存和协商缓存：  
2.1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；  
2.2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。

3.协商缓存对比： etag 优先级高于 last-modified；  
4.etag 精度高，last-modified 精度是 s，1s 内 etag 修改多少次都会被记录； last-modified 性能好，etag 要得到 hash 值。

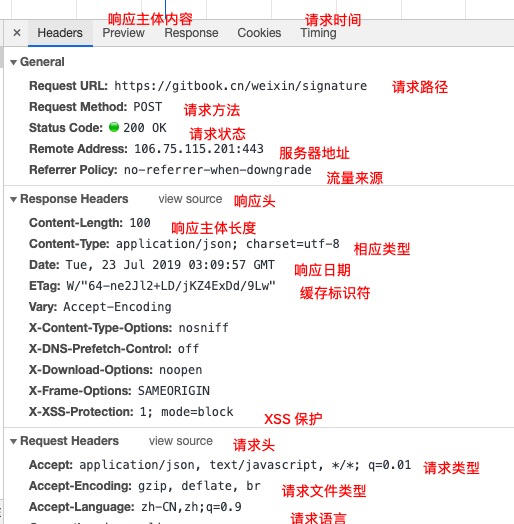
5.浏览器读取缓存流程： 会先判断强缓存；再判断协商缓存 etag(last-modified)是否存在；  
存在利用属性 If-None-match(If-Modified-since)携带值；  
请求服务器,服务器对比 etag(last-modified)，生效返回 304。

F5 刷新会忽略强缓存不会忽略协商缓存，ctrl+f5 都失效

**7.5.8 状态码**

| **序列** | **详情** |
| --- | --- |
| 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） |

**7.5.9 浏览器请求分析**



**7.5.10 总结**

协议

| **版本** | **内容** |
| --- | --- |
| http0.9 | 只允许客户端发送 GET 这一种请求;且不支持请求头,协议只支持纯文本;无状态性,每个访问独立处理,完成断开;无状态码 |
| http1.0 解决 0.9 的缺点,增加 If-modify-since(last-modify)和 expires 缓存属性 |  |
| http1.x | 增加 cache-control 和 If-none-match(etag)缓存属性 |
| http2.0 | 采用二进制格式传输;多路复用;报头压缩;服务器推送 |
| http3.0 | 采用 QUIC 协议,自定义连接机制;自定义重传机制;无阻塞的多路复用 |

缓存

| **类型** | **特性** |
| --- | --- |
| 强缓存 | 通过 If-modify-since(last-modify)、expires 和 cache-control 设置，属性值是时间，所以在时间内不用请求 |
| 协商缓存 | 通过 If-none-match(etag)设置，etag 属性是哈希值，所以要请求和服务器值对比 |

**8.总结**

这只是 JS 原生从初级到高级的梳理;  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。