前端复习笔记(初级)

instanceof hasOwnProperty 的用法

原型和原型链

1. 所有的引用类型（数组、对象、函数），都具有对象特性，即可自由扩展属性（除 null 外）
2. 所有的引用类型（数组、对象、函数），都有一个__proto__属性，属性值是一个普通的对象
3. 所有的函数，都有一个 prototype 属性，属性值也是一个普通的对象
4. 所有的引用类型（数组、对象、函数），__proto__属性指向它的构造函数的 prototype 属性

```JavaScript
var obj = {}; obj.a = 100;
var arr = []; arr.a = 100;
function fn () {};
fn.a = 100;

console.log(obj.__proto__);
console.log(arr.__proto__);
console.log(fn.__proto__);

console.log(fn.prototype);

// obj 的构造函数是 Object，因为 obj instanceOf Object 为 true
console.log(obj.__proto__ === Object.prototype) // true
```

5. （重要）当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么会它的__proto__（即它的构造函数的 prototype）中寻找
6. 原型的构造器指向构造函数

```JavaScript
function Star(name) {
  this.name = name;
}
let obj = new Star('小红');
console.log(Star.prototype.constructor === Star); // true
console.log(obj.__proto__.constructor === Star); // true
```

7. 如何继承父类

```JavaScript
function Father(name) {
  this.name = name;
}
Father.prototype.dance = function () {
  console.log('I am dancing');
};
function Son(name, age) {
  Father.call(this, name);
  this.age = age;
}
Son.prototype = new Father();
Son.prototype.sing = function () {
  console.log('I am singing');
```

```
};
let son = new Son('小红', 100);
console.log(Father.prototype) // {dance: ƒ, constructor: ƒ}
```

js 中的 new 做了些什么

js的new操作符到底做了什么?

做了什么? 1、创建了一个空的js对象（即{}） 2、将空对象的原型prototype指向构造函数的原型 3、将空对象作为构造函数的上下文（改变this指向） 4、对构造函数有返回值的判断 怎么实现? /* create函数要接受不定量...

zhuanlan.zhihu.com

(1) 创建一个新对象；
(2) 将构造函数的作用域赋给新对象（因此 this 就指向了这个新对象）；
(3) 执行构造函数中的代码（为这个新对象添加属性）；
(4) 对构造函数有返回值的判断，判断返回的是对象还是函数的返回。

JavaScript

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

let ss = new Obj('Allen', 24)
console.log(ss)

function myNew(Con, ...args) {
  // 创建一个新的空对象
  let obj = {};
  // 将这个空对象的__proto__指向构造函数的prototype
  // obj.__proto__ = Con.prototype;
  Object.setPrototypeOf(obj, Con.prototype);
  // 将this指向空对象
  let res = Con.apply(obj, args);
  // 对构造函数返回值做判断，然后返回对应的值
  return res instanceof Object ? res : obj;
}

ss = myNew(Obj, 'allen', 24)
console.log(ss)
```

this 的几种使用场景

> this 要在执行时才能确认，定义时无法确认

JavaScript

```
var obj = {
  name: 'A',
  fn: function() {
    console.log(this.name)
  }
}
obj.printName() // this => obj
obj.fn.call({ name: 'B' }) // this => { name: 'B' }
var fn1 = obj.fn
fn1() // this => window
```

1. 作为构造函数执行
2. 作为对象属性执行
3. 作为普通函数执行

4. call apply bind

```JavaScript
// 作为构造函数执行
function Foo(name) {
  this.name = name
}
var f = new Foo('allen')

// 作为对象属性执行
var obj = {
  name: 'A',
  fn: function() {
    console.log(this.name)
  }
}
obj.printName() // A

// 作为普通函数执行
function fn() {
  console.log(this)
}
fn() // Window

// call apply bind
function fn2(name, age) {
  console.log(name, age)
  console.log(this)
}
fn.call({ x: 100 }, 'zhangsan', 20) // zhangsan 20 {x: 100}
```

前端使用异步的场景

1. 定时任务：setTimeout，setInterval
2. 网络请求：ajax 请求，动态加载<img>标签
3. 事件绑定：addEventListener('event', fn)

DOM 对象的 property 和 attribute 的区别

property 是 js 对象的属性，attribute 是 html 标签的属性

```JavaScript
var pEl = document.createElement('p')
pEl.style.color = 'red'
console.log(pEl.style.color) // 'red'
console.log(pEl.getAttribute('style')) // 'color: red;'

console.log(pEl.nodeName) // P
console.log(pEl.getAttribute('nodeName')) // null
```

事件绑定

事件冒泡，事件冒泡的应用，子标签会动态加载的时候或者子标签只有少量不需要绑定事件，而同级需要绑定事件时，可以利用事件冒泡机制，在父标签上绑定个事件来进行处理

Ajax 跨域同源策略

jsonp，CORS，服务端跨域 设置 header，nginx 配置（支持跨域请求的标签 img，link，script）

Git 常用命令

存储

cookie 4kb 每次都会发送个服务端，所有不能太大

localStorage 5mb 浏览器共享

sessionStorage 5mb 标签不共享

模块化

https://juejin.cn/post/6844903576309858318

AMD（异步加载，require.js define 和 require）

CMD（动态加载的 AMD，sea.js）

CommonJS（node 的 module.exports 和 require）

ES6（import export）：

页面加载

dns 解析成 ip，浏览器发请求，服务端解析并返回数据，浏览器再渲染

页面渲染

css 应该放前面加载（先加载 css，再渲染 dom 才不会）

js 应该放在最后加载（script 在中间执行的话，会阻塞 dom 渲染，而且 js 可能会操作 dom，没渲染完成的话可能会报错）

window.onload 和 DOMContentLoaded

window.onload：页面的全部资源加载完才会执行，包括图片、视频等

DOMContentLoaded： DOM 渲染完即可执行，此时图片、视频可能还没有加载完

上线回滚

将生产文件替换为上一个版本的文件备份

性能优化

1、加载资源优化

静态资源压缩合并

静态资源缓存

使用 CDN

ssr 服务端渲染

2、渲染优化

css 放前面，js 放后面

图片懒加载，下拉加载

减少 dom 查询，对 dom 查询做缓存

减少 dom 操作，多个 dom 操作合并在一起执行

事件节流

尽早执行操作（DOMContentLoaded 就执行）

安全性

XSS（新增文章输入 script 标签，渲染时执行，解决办法：替换标签）

XSRF（邮件里面图片地址是付款链接，解决办法：增加验证流程，验证码，密码，短信验证码）