ES6

1、let 与 var

var 可重复声明，let 在同一作用域下不能重复声明。

var 的作用域可为全局作用域和函数作用域，let 的作用域可为全局作用域和块级作用域（即花括号{}，代表代码块）。

var 会进行预解析（先打印后声明也可打印出结果），let 不会进行预解析。

2、const

常量声明时必须赋值，此后不能再重新赋值，且作用于块级作用域，也不会被预解析

3、看文档：推荐 MDN，也可以看一下 ECMA 官方文档

4、解构赋值

```
let obj = {
    a: 1,
    b: 2
};
// let a = obj.a;
// let b = obj.b;
// console.log(a,b);
let {a,b} = obj;
console.log(a,b);
```

对象解构赋值：

在对象解构赋值中，变量名称必须一一对应（a 对应 a，b 对应 b），因为是根据属性进行赋值。

解构赋值时可以重命名，如将 name 重命名为 myname，即使用冒号隔开：

```
let obj = {name:"张三",age:20,obj:{}};
let {name:myname,age} = obj;
console.log(myname,age);
```

```
let arr = ["a","b","c"];
let [e,f] = arr;
console.log(e,f);
```

数组解构赋值：

在数组解构赋值中，变量名称不需要一一对应（e 对应 a，f 对应 b），因为是根据顺序进行赋值。

```
let str = "abc";
let [e,f] = str;
console.log(e,f);
```

字符串解构赋值：

在数组解构赋值中，变量名称不需要一一对应（e 对应 a，f 对应 b），因为是根据顺序进行赋值，与数组解构赋值类似。

5、展开运算符

```
let arr = [1,2,3,4];
let arr2 = ["a","b",...arr,"c","d"];
console.log(arr2);
```

数组展开运算符：

```
let obj = {
    a:1,
    b:2
};
let obj2 = {
    ...obj,
    c:3,
    d:4
};
console.log(obj2);
```

对象展开运算符：

展开运算符结合解构赋值可以存储解构赋值后剩余的参数：

```
let obj = {
    a:1,
    b:2
};
let obj2 = {
    ...obj,
    c:3,
    d:4
};
console.log(obj2);
// let {a,b,...c} = obj2;
// console.log(a,b,c);
```

传址问题：不建议将对象 *obj* 直接赋值给另外一个对象 *obj2*，因为修改 *obj2* 中的属性，*obj* 中的属性也会相应改变。但当应用展开运算符将对象 *obj*（即 *{...obj}* ）赋值给对象 *obj2*（那修改了 *obj2* 中的属性会不会影响这个新对象 *{...obj}* 中的相应属性？ ）后，修改 *obj2* 中的属性，则不会影响 *obj* 中的属性。因为 *{...obj}* 相当于一个新对象。

6、*Set*

```
// 构造函数 用来构建某一类型的对象 - 对象的实例化
let arr = [2,1,2,1,3,"a",4,4,5,1]
let s = new Set(arr);
// console.log(s.size);// size 数值的个数，==> length
//console.log(s.clear());//清空所有值
//console.log(s.delete("b")); //删除某一项
s.add(5).add(6).add(7);
//console.log(s.has("a"));
console.log(s);
```

可用于数组去重

Set 存储数组 *arr* 去重后的新数组

7、*Map*：映射

```
let arr = [
    ["a",1],
    ["b",2],
    ["c",3]
];
let s = new Map(arr);
console.log(s);
```

相当于构造了一个对象，属性 *a*（*key*）对应的值为 *1*（*value*），属性 *b*（*key*）对应的值为 *2*（*value*），属性 *c*（*key*）对应的值为 *3*（*value*）。但不能直接接收一个对象，因为对象不可迭代。

8、箭头函数

相当于函数。但与函数相比，箭头函数没有不定参（*arguments*）（即当不确定函数的形参有几个时，可直接将使用函数时传入的实参作为 *arguments* 的值）。若要在箭头函数中实现类似不定参的效果，可使用剩余（*rest*）参数，即展开符与参数名（...参数名）。

箭头函数没有 this，调用箭头函数的 this 时指向的是声明箭头函数时的作用域的 this。

9、冻结对象使之不可变

```
Object.freeze(obj);
```

10、ES6 兼容性不如 ES4、ES5，可用 Babeljs 将 ES6 语法转换为 ES4、ES5：

https://www.babeljs.cn/

11、${变量} 可以呈现变量的取值，也可以使用简单的三元运算符，但使用 if、else 等则不行

```
let a = 1234;
xxx.innerHTML = `<th>${a}</th>`;
${item.checked?'checked':''}
```

12、伪数组或类数组可用展开符转换为真正的数组：

```
let ageEles = document.querySelectorAll(".
age_sort a");
// 性别相关节点;
let genderEles = document.querySelectorAll
(".gender_show a");

// console.log(ageEles);
[...ageEles]
```

获取到的 ageEles 实际为 Nodelist，是伪数组，用展开符...可将其转换为真正的数组 Array。

13、sort 方法排序

从小到大：`.sort((a,b)=>a-b)`

从大到小：`.sort((a,b)=>b-a)`

14、需要用 this 就写 function，不需要用 this 就可以使用箭头语法（=>），比较简洁。

15、sort 会改变 data，所以需要用 map 预先映射一个临时 data，再改变这个临时 data，就不会改变原数据 data，filter 在 sort 后推出，因此 filter 自带预先映射临时 data 的功能，所以使用时不需要预先映射一个临时 data。

```
// 年龄数据的筛选
// map-->映射;
let sortAge = [
    data=>data.map(item=>item).sort((r1,r2)
    =>r1.age-r2.age),
    data=>data.map(item=>item).sort((r1,r2)
    =>r2.age-r1.age),
    data=>data.map(item=>item)
];

// 性别数据的筛选;
let genderFilter = [
    data=>data.filter(item=>item.gender==="
    男"),
    data=>data.filter(item=>item.gender==="
    女"),
    data=>data
];
```

16、Math.max.apply 巧妙用法

Math.max()的括号中不能直接取数组的最大值，即：

Math.max(1,2,3); → 这样可以

let arr = [1,2,3];

Math.max(arr); → 这样不行

此时可用：

Math.max.apply(null, arr); → 这样可以

17、创建对象

① 字面量

```
let obj = {
    id: xxx,
    name: xxx
}
```

②构造函数（约定俗成首字母大写；其 this 指向实例化对象）

```
let obj = new Object();
obj.id = xxx;
obj.name = xxx;
```

③属性方法放入原型中

```
let obj = Object.create({
    id: xxx,
    name: xxx
})
```

18、对象属性名称

①

```
let obj = {
    id: xxx,
    name: xxx
}
```

②

```
let str = 'id';
let obj = {
    [str]: xxx,
    name: xxx
}
```

19、new 运算符：执行函数，自动创建一个空对象，并将 this 与该空对象绑定（即 this 就是该空对象），同时，如果无返还 return，则隐式返还 this

20、持久化保存：前端离线存储、后端服务器存储

localstorage 以 json 存储 data，以后会讲

21、公有属性和私有属性

私有属性无法直接访问，但可以通过构造一个方法进行访问，如下图：

#为私有属性关键字，故#myname 即为私有属性。

consol.log(zhangsan.myname)无法得到该私有属性，但可以通过构造 getName 方法并执行该方法来获取该私有属性。

```
class Person{
    // 私有属性  _myname;
    #myname = "狗蛋";
    constructor(name){
        this.name = name;
        this.age = 20;
    }
    getName(){
        console.log(this.#myname);
    }
}
let zhangsan = new Person("张三");
console.log( zhangsan.myname );
zhangsan.getName();
```

22、不要过多地设置全局变量，容易造成变量污染。

23、类的静态属性

```
class Person{
    static instance = null;
    constructor(){
```

其中 static 的 instance 即为 class Person 的静态属性，属于类，不属于该类的任何实例化对象。

24、继承

```
// ES6 继承;
class Dad{
    constructor(name){
        this.name = name;
        this.age  = 50;
    }
}
class Son extends Dad{
    constructor(name){
        super(name);
    }
}
let zhangsan  = new Son("张三");
console.log(zhangsan.name);
```

当子类中有 constructor 的时候就必须要有 super，且 super 必须在子类中定义的属性之前。super 其实相当于调用父级的 constructor。

静态属性也可以在子类中继承。

25、合并空运算符

ES2020 中??可设置默认值

```
// ES2020; 合并空运算符;
function test(name ,age){
    // 参数默认值;
    // name = name || "默认名称";
    // age = age || 25;
    name = name ?? "默认名称";
    age = age ?? 25;
    console.log(name,age);
}
test("",0);
```

26、可选链式操作

?表示有该对象/方法/属性就打印，无则不操作，不会报错。

```
// 可选链式操作
let obj = {
    name:{
        age:20
    }

};
console.log(obj?.name?.age);
```

27、模块化

amd：*require.js*

cmd：*sea.js*

ESM：*COMMONJS* 规范　　*node.js*

export 可以导出多个，*export default* 只能导出一个

a.js 文件中代码：

index.html　a.js

ESM > a.js > ...

```
1  // 导出;
2  console.log("a.js");
3  // 导出多个;
4  export let a = 10;
5  export let c = 30;
6  let obj = {
7      name:"张三",
8      age:20
9  }
10 export {obj};
11 let b = 20;
12 // 导出一个
13 export default b;
```

在 *index.html* 中引入 *a.js*（如上所示）中的默认值（*myb* 即 *b*）、*a*、*c*、*obj*

```
<script type="module">
// ES module  ESM;  AMD :require.js ;CMD:sea.js;
// ESM / COMMONJS规范、nodejs 模块化;
// ES6模块化:  导入 import from    导出 export;
import myb,{a,c,obj} from './a.js';
console.log(myb,a,c,obj);
```

通过 as 起别名

```
import myb,{a as d,c,obj} from './a.js';
console.log(myb,d,c,obj);

// 通过as 起别名;
```

也可以不写具体需要引入的变量或对象等，而直接使用通配符*将所有返回值都引入，并将所有值都作为一个对象中的属性

```
import * as obj from './a.js';
```

在 script 里直接写 import 会自动导入该 js 文件，当将 import 写入 function 函数里时会报错，因为该 import 不支持按需导入（即触发一定条件才导入）

若需要按需导入，则可以在 function 中写以下 import 预函数

import 函数会返回一个 promise 对象，需要取 promise 对象的值则使用 then 方法

```
document.onclick = function(){
    // import b from './b.js';
    // 返还promise对象
    import("./b.js").then(res=>{
        console.log(res);
    })
}
```

可以将以上过程变成同步的写法:

```
// 按需导入;
document.onclick =async function(){
    // console.log("111");
    // import b from './b.js';
    // 返还promise对象
    // import("./b.js").then(res=>{
    //     console.log(res);
    // })
   let res =  await import("./b.js");
   console.log(res);

}
```

28、公共空间：原型

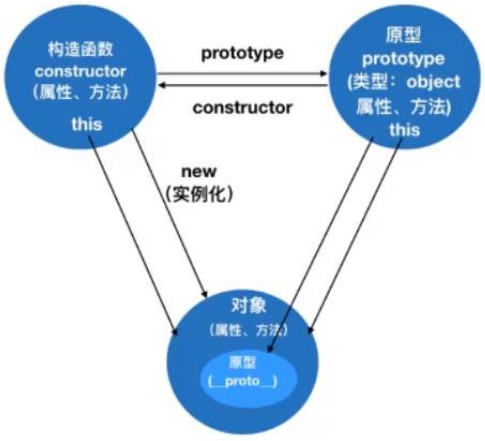
```
// 功能空间原型;
Person.prototype.hobby = function(){
    console.log("喜欢篮球");
}
Person.prototype.fn = function(){
    console.log("fn");
}

let zhangsan = new Person("张三");
// console.log(zhangsan);
console.log(zhangsan.__proto__===Person.prototype);
```

类.prototype 与 实例化对象.__proto__ 是一个东西

29、构造函数、原型、对象之间的关系

原型构造函数及对象关系



30、工厂模式、构造函数的区别

```
// 工厂模式
function Person(name){
    let obj = {};
    obj.name = name;
    obj.age = 20;
    obj.fn = function(){
        console.log("fn..");
    }
}
let zhangsan  = Person("张三");
```

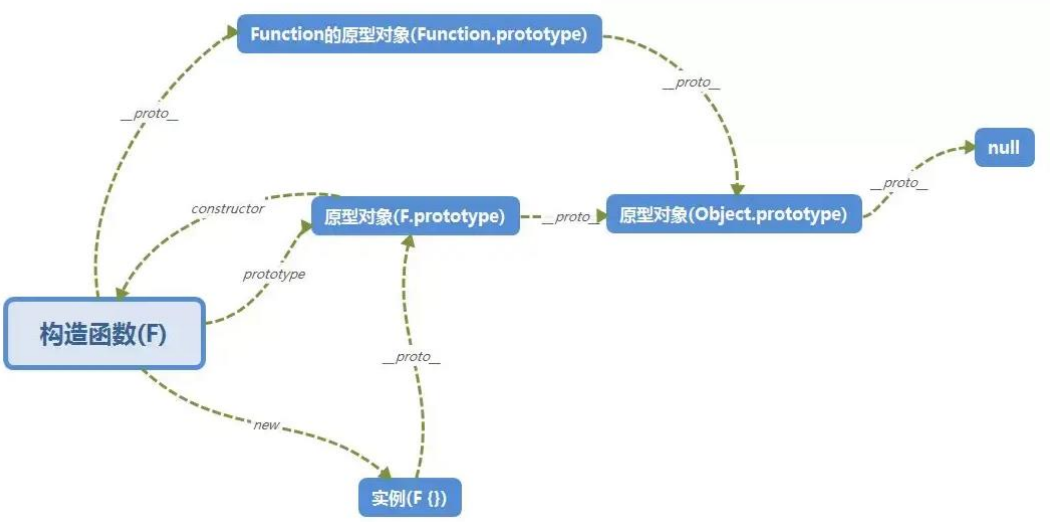
```
// 构造函数;
function Person(){
    this.name = name;
    this.age = 20;
}
Person.prototype.fn = function(){
    console.log("fn...");
}
let zhangsan  = new Person("张三");
```

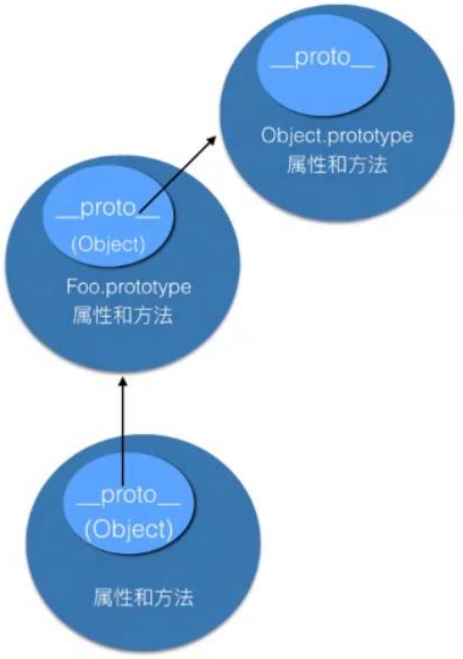
1）工厂模式无法解决对象识别的问题，即无法判断实例属于哪个对象，但构造函数模式则可以。

2）构造函数具有原型，相当于方法都存于公共空间中，而工厂模式则没有公共空间，相当于每实例化一个对象，工厂模式都要新建内存储存每个对象的方法，而构造函数则不需要新建内存存储。

31、原型链

https://www.jianshu.com/p/08c07a953fa0






```
// 构造函数
function Foo(name){
    this.name = name;
    this.age = 20;
    // this.test = "你好"
}
//原型对象;
Foo.prototype.fn = function () {
    console.log("f");
}
// Foo.prototype.test = "hello";
Object.prototype.test = "你好";
let newFoo = new Foo("张三");
console.log( newFoo.test);
```

按原型链查找 *newFoo.test*（就近原则）：构造函数的属性/方法（右上图中的 *this.test*）→构造函数的原型对象的属性/方法（*Foo.prototype.test*）→顶层对象的属性/方法（*Object.prototype.test*）

32、call、apply、bind

改变指向（*this*）：*foo* 原本指向（*this*）为 *window*，用 *foo.call*（*obj*）或 *foo.apply*（*obj*）后 *foo* 的指向（*this*）变为 *obj* 对象

```
function foo(name,age) {
    console.log(this,"姓名是"+name+"年龄是"+age);
}
// foo();
let obj = {
    name:"张三"
}
// foo.call(obj,"张三",20);
```

call 与 *apply* 的区别：参数写法不一样，*call* 从第二个参数开始依次传参即可，而 *apply* 则需

将参数变成数组并作为第二个参数传入，bind 则本身为一个函数，故需在第二括号中传入参数，如下图所示

```
// foo.call(obj,"张三",20);
// foo.apply(obj,["张三",20]);
foo.bind(obj)("张三",20);
```

33、继承

用 call、apply、bind 即可

```
// 继承;
function Dad(name,age) {
    this.name = name;
    this.age = age;
    this.money = "100000";
}


function Son(name,age) {
    // Dad.call(this,name,age);
    // Dad.apply(this,[name,age])
    Dad.bind(this)(name,age);
    this.sex = "男";
}

let zhangsann  = new Son("张三",20);
console.log( zhangsann.money);
```

34、传值和传址

在赋值（即简单的=）时，除简单数据类型（number、string、undefined、null 等）外，其他复杂数据类型都涉及传址问题，即新变量与旧变量共用同一个内存地址（相当于同一个人有两个名字）。

https://zhuanlan.zhihu.com/p/268685817

```
// 基本类型变量存的是值，复杂类型的变量存的是内存地址。
// 基本类型在赋值的时候拷贝值，复杂类型在赋值的时候只拷贝地址，不拷贝值。
let arrayA = [1]; // 基本类型（无地址值）：字符串（String）、数字(Number)、布尔(Boolean)、对空（Null）、未定义（Undefined）、Symbol(ES6 新增)
let arrayB = [{b:1}];// 复杂类型（有地址值）：对象(Object)、数组(Array)、函数(Function)
arrayA.forEach((itemA) => {
    itemA = 3// 只改变 arrayA.forEach 中的 itemA，而不会改变 arrayA（因为 arrayA.forEach 中的 itemA 为基本类型，因此无地址值，故本就不指向 arrayA）
});
arrayB.forEach((itemB) => {
    itemB = 3; // 只改变 arrayB.forEach 中的 itemB，而不会改变 arrayB（因为 arrayB.forEach 中的 itemB 由复杂类型变为基本类型，因此无地址值，故由指向 arrayB 变为不指向 arrayB）
    itemB = {b:3};// 只改变 arrayB.forEach 中的 itemB，而不会改变 arrayB（arrayB.forEach 中的 itemB 的地址值被修改，故由指向 arrayB 变为不指向 arrayB/指向{b:3}）
    itemB.b = 3;// 既改变 arrayB.forEach 中的 itemB，又改变 arrayB（arrayB.forEach 中的 itemB 的地址值没有被修改，依然指向 arrayB）
});
```

如果 array 是基本类型，那 forEach 中 item 的改变影响不了 array；

如果 array 是复杂类型，则 forEach 中的 item 的地址值默认与 array 相同：

forEach 修改了 item 的地址值（包括将 item 改为基本类型），改变不了 array；

forEach 没有修改 item 的地址值，只是修改 item 内部的某些属性，会改变 array。

```
// 基本类型变量存的是值，复杂类型的变量存的是内存地址。
// 基本类型在赋值的时候拷贝值，复杂类型在赋值的时候只拷贝地址，不拷贝值。
let arrayA = [1]; // 基本类型（无地址值）：字符串（String）、数字(Number)、布尔(Boolean)、对空（Null）、未定义（Undefined）、Symbol(ES6 新增)
let arrayB = [{b:1}];// 复杂类型（有地址值）：对象(Object)、数组(Array)、函数(Function)
arrayA.forEach((itemA) => {
    itemA = 3// 只改变 arrayA.forEach 中的 itemA，而不会改变 arrayA（因为 arrayA.forEach 中的 itemA 为基本类型，因此无地址值，故本就不指向 arrayA）
});
arrayB.forEach((itemB) => {
    itemB = 3; // 只改变 arrayB.forEach 中的 itemB，而不会改变 arrayB（因为 arrayB.forEach 中的 itemB 由复杂类型变为基本类型，因此无地址值，故由指向 arrayB 变为不指向 arrayB）
    itemB = {b:3};// 只改变 arrayB.forEach 中的 itemB，而不会改变 arrayB（arrayB.forEach 中的 itemB 的地址值被修改，故由指向 arrayB 变为不指向 arrayB/指向{b:3}）
    itemB.b = 3;// 既改变 arrayB.forEach 中的 itemB，又改变 arrayB（arrayB.forEach 中的 itemB 的地址值没有被修改，依然指向 arrayB）
});
```

如果 *array* 是基本类型，那 *forEach* 中 *item* 的改变影响不了 *array*；

如果 *array* 是复杂类型，则 *forEach* 中的 *item* 的地址值默认与 *array* 相同：

forEach 修改了 *item* 的地址值（包括将 *item* 改为基本类型），改变不了 *array*；

forEach 没有修改 *item* 的地址值，只是修改 *item* 内部的某些属性，会改变 *array*。

35、深拷贝

新变量与旧变量不共用同一个内存地址，而是新开辟一个内存地址，如下图所示。但通过这种方式创建的新对象只会保留简单的属性，而不会保留方法（*function*）等。

```
// 深拷贝;
let DadProto = {
    name:"张三",
    age:20
}
let SonProto = JSON.parse(JSON.stringify( DadProto));
```

封装深拷贝函数：*deepCopy(obj)*

Array.isArray(obj)?[]:{} 表示 判断 *obj* 是否是数组，若是数组则创建*[]*数组，若不是数组则创建*{}*对象

*hasOwnProperty()*方法会查找一个对象是否有某个属性，但是不会去查找它的原型链

```
function deepCopy(obj) {
    let newObj = Array.isArray(obj)?[]:{};
    for(let key in obj){
        if(obj.hasOwnProperty(key)){
            if(typeof obj[key] === "object"){
                newObj[key] = deepCopy(obj[key]) ;
            }else{
                newObj[key] = obj[key];
            }
        }
    }
    return newObj;
}
```

36、组合继承

https://www.cnblogs.com/sarahwang/p/9098044.html

在 33 的基础上，通过实例化对象的方式进行组合继承

```
let Link = function(){};
Link.prototype = Dad.prototype;
Son.prototype = new Link();
Son.prototype.constructor = Son;
```

37、extends 类继承

https://segmentfault.com/a/1190000010407445

```
export default class Luban extends Hero{
    constructor(){
        super("鲁班","./sources/heros/luban1.png",
        [new S1(),new S2(),new S3()],[new Skin1(),
        new Skin2()]);
        this.height = "1.2m";
```

estends 继承包括两步：①继承父类的原型属性（prototype）；②继承父类的对象属性

super()的实质就是 call 继承；把父类的对象方法继承给子类对象；这也是为什么在 es6 的继承时必须要加上 super()，因为不加的话无法继承父类的对象属性。

38、设计原则：单一原则；开闭原则，对内是封闭的，对外是扩展开放的；

39、设计模式：

- 单例模式：限制类的实例化次数只有一次，即一个类只能有一个实例，且可自行/主动实例化，并提供一个访问它的全局访问点。

  实现方式：创建一个变量，使其初始值为 null 或 undefined，进行类的实例化时，判断类实例对象是否存在或该变量是否为 null 或 undefined，若类实例对象不存在或该变量为 null 或 undefined，则创建类实例后返回该实例，若类实例对象存在或该变量不为 null 或 undefined，则返回已创建的类实例。（多次调用类实例生成方法，均返回同一个类实例对象。）

```
class Person{
    static instance = null;
    constructor(name){
        if(Person.instance){
            return Person.instance;
        }
        Person.instance = this;
        this.name = name;
    }
}
let newPerson = new Person("name1");
let newPerson2 = new Person("name2");
console.log(newPerson, newPerson2);
```

newPerson 和 newPerson2 的结果一样，都是↓

也可以如下，创造一个函数：

```
class Person{
    constructor(name){
        this.name = name;
    }
}
let instance;
function createInstance(...arg){
    if(!instance){
        instance = new Person(...arg);
    }
    return instance;
}
let newPerson = createInstance("name1");
let newPerson2 = createInstance("name2");
console.log(newPerson, newPerson2);
```

- 工厂模式：

```
//工厂模式
function Factory() {
    let obj = {};
    //加工
    obj.name = "张三";
    obj.age = 20;
    //出厂
    return obj;
}
```

- 装饰器：通过添加 @方法名 对一些对象进行装饰包装然后返回一个被包装过的对象，可以装饰的对象包括：类，属性，方法等。（@方法需要配置 webpack 环境，浏览器原生不支持）

```
@decorator
class A {}
// 等同于
class A {}
A = decorator(A) || A;
```

```
function Hurt() {
    console.log("造成了1000点伤害");
}

Function.prototype.Decorator = function(fn){
    this();
    fn();
}

class Hero{
    constructor() {
    }
    fire() {
        console.log("释放了技能");
    }
}

let newHero = new Hero();
newHero.fire.Decorator(Hurt);
```

、

- 观察者模式：一对多。观察者模式可以很好地实现 2 个模块之间的解耦。

```
const fn1 = function(){
console.log("fn1");
}
const fn2 = function(){
console.log("fn2");
}

let handles = {};
// 自定义事件
// 绑定eventName与fn
function addEvent(eventName,fn){
    if(typeof  handles[eventName] === "undefined"){
        handles[eventName] = [];
    }
    handles[eventName].push(fn);
}
// 触发eventName
function trigger(eventName){
    if(typeof handles[eventName]=== "undefined"){
        return ;
    }
    handles[eventName].forEach(fn=>{
        fn();
    });
}

// 将eventName与fn1、fn2绑定
addEvent("myevent",fn1);
addEvent("myevent",fn2);
// 触发eventName，因此会console.log出fn1和fn2（惰性执行/延迟执行）
trigger("myevent");
```

40、闭包：一个持有外部环境变量的函数就是闭包。（即函数能访问在函数外定义的变量，而在函数外无法访问函数内定义的变量。）

```
let a = 1
let b = function(){
    console.log(a)
}
```

41、assign：在各对象中的属性无重复的情况下可以合并各个对象，在各对象中的属性有重复的情况下会令后一个对象的值覆盖前一个对象的值。可用于合并配置。

```
this.opts = Object.assign({
    width: "30%",
    height: "250px",
    title: "测试标题",
    content: "测试内容",
    dragable: true, //是否可拖拽
    maskable: true, //是否有遮罩
    isCancel:false //是否有取消
},opts)
```

width 等为默认参数，*opts* 为用户设置的参数

42、addEventListener

```
// 绑定事件
this.dialogEle.querySelector(".k-dialog").
addEventListener("click",e=>{
    console.log(e.target);
})
```

e.target 即点击的对象

43、EventTarget

https://developer.mozilla.org/zh-CN/docs/Web/API/EventTarget/EventTarget

系统预定义的类。

▼ Constructor

EventTarget()

▼ 方法

addEventListener()

dispatchEvent

removeEventListener

*EventTarget.addEventListener()*方法可绑定事件。

EventTarget. dispatchEven 方法可触发事件，相当于 *trigger*。

https://developer.mozilla.org/zh-CN/docs/Web/Guide/Events/Creating_and_triggering_events

```
class MyEventTarget extends EventTarget {
    constructor(mySecret) {
      super();
      this._secret = mySecret;
    }

    get secret() { return this._secret; }
};

    let myEventTarget = new MyEventTarget(5);
    let value = myEventTarget.secret;  // == 5
    myEventTarget.addEventListener("foo", function(e) {
        this._secret = e.detail;
    });

    let event = new CustomEvent("foo", { detail: 7 });
    myEventTarget.dispatchEvent(event);
    let newValue = myEventTarget.secret; // == 7
```

myEventTarget 是 *EventTarget* 的扩展类

*myEventTarget(_secret=5)*是 *myEventTarget* 类的一个实例，该实例上可添加 *addEventListener*

方法，即绑定 foo 事件，function(e)即 foo 事件，通过 function(e)中的 e.detail 与 MyEventTarget 类中的 this._secret 的对应将值传给 this._secret。

※ 注：detail 不能更改成别的名字，是固定的。

44、扩展组件

①扩展 html 已有的组件：如 img 元素

```
class MyImg extends HTMLImageElement {
```

②扩展自己创建的组件

```
class InputDialog extends Dialog{
```

45、this 穿透：将函数作用域穿透至上层

46、class 里的非静态属性：使用 get xxx()

※ 无 get 而仅有 xxx()是 class 里的方法函数，有 get 则是设置属性，即 get xxx()等同于 this.xxx。

```
get title(){
    // 处理默认参数;
    return this.getAttribute("title") ?? '默认标题';
}
get content(){
    return this.getAttribute("content") ?? '默认内容';
}
```

47、jquery 思维方式与原生 js 类似，而不像 vue、react 是 mvvm 数据优先的方式

48、合并配置：assign（见 41 或下图方式四）、解构赋值与展开运算符（见下图方式二）、循环判断（见下图方式三）、空值合并运算符（即？？，当？？左侧的操作数为 null 或者 undefined 时，返回？？右侧操作数，否则返回？？左侧操作数）

```
constructor({width="30%",height="250px", title="测试标题",content="测试内容",dragable=true,mask
    super();
    //作业：合并配置，用老师之外的方法合并配置，外部有配置传入以 外部配置为准，如果没有配置传入就调用默
    // this.opts = opts;
    // let { width="30%",height="250px", title="测试标题",content="测试内容",dragable=true,mask
    // 方式一
    this.opts = {
        width,
        tittle,
        // ...
    }
    // 方式二
        let newobj =  {...obj1,...obj2}

    // 方式三
    for(let i in obj1){
        obj2[i] = obj1[i]
    }
    // 方式四
    this.opts = Object.assign({
        width: "30%",
        height: "250px",
        title: "测试标题",
        content: "测试内容",
        dragable: true, //是否可拖拽
        maskable: true, //是否有遮罩
        isCancel:false, //是否有取消
        success(){},
        cancel(){}
    },opts)
```

49、管道 pipe（nodejs）、组合 compose（react）：像链式操作

50、高阶函数：满足以下两个条件之一，函数 B 即可被称为高阶函数

①将函数 A 作为参数传入函数 B

②函数 B 将函数 A 当成返还参数（即 return 函数 A）

51、addEventListener 绑定 click 事件与直接 onclick 事件的区别

http://www.5imoban.net/jiaocheng/jquery/201809293429.html

①onclick 事件在同一时间只能指向唯一对象

②addEventListener 给一个事件注册多个 listener

③addEventListener 对任何 DOM 都是有效的，而 onclick 仅限于 HTML

④addEventListener 可以控制 listener 的触发阶段，（捕获/冒泡）。对于多个相同的事件处理器，不会重复触发，不需要手动使用 removeEventListener 清除

⑤IE9 使用 attachEvent 和 detachEvent

52、DOMContentLoaded 与 load

https://developer.mozilla.org/zh-CN/docs/Web/API/Window/DOMContentLoaded_event

DOMContentLoaded：当初始的 HTML 文档被完全加载和解析完成之后，DOMContentLoaded 事件被触发，而**无需等待样式表、图像和子框架的完全加载。**

load：仅用于检测一个完全加载的页面，**页面的 html、css、js、图片等资源都已经加载完之后才会触发 load 事件。**

53、类数组

```
// 情况三：对象
if(typeof arg.length === "undefined"){
    // 一个节点
    console.log("对象");

}else{
    // 多个节点;
    console.log("数组");
}
```

一个类数组对象：

具有：指向对象元素的数字索引下标以及 length 属性告诉我们对象的元素个数；

不具有：诸如 push、forEach 以及 indexOf 等数组对象具有的方法；

几个典型的类数组的例子是：

- DOM 方法 document.getElementsByClassName() 的返回结果（实际上许多 DOM 方法的返回值都是类数组）；
- 特殊变量 arguments 对象；
- input 的文件对象 FileList；

54、from()：用于通过拥有 length 属性的对象或可迭代的对象来返回一个数组。

55、链式操作：在函数中 return 了 this，即返还自身。

```
function Person(){
    this.age=20;//默认值为20;
}

//通过原型给 Person构造函数上添加一个设置年龄的方法
Person.prototype.setAge=function(num){
        this.age=num;
        return this;
}

//通过原型给 Person构造函数上添加一个获取年龄的方法
Person.prototype.getAge=function(){
        return this.age;
}

console.log(new Person().setAge(30).getAge());
```

56、*is* 属性：可指向自定义的继承了某种 *html* 元素的类

```
// Create a class for the element
class WordCount extends HTMLParagraphElement {
  constructor() {
    // Always call super first in constructor
    super();

    // Constructor contents ommitted for brevity
    ...

  }
}

// Define the new element
customElements.define('word-count', WordCount, { extends: 'p' });
```

```
<p is="word-count"></p>
```

57、自定义新的类（与已有的 *html* 元素类无关，即不是继承）

```
class MyCom extends HTMLElement{
    constructor(){
        super();
        // console.log(this);
        this.innerHTML = "<button>按钮</button>";
    }
}
customElements.define("my-com",MyCom);
```

58、*get* 定义属性（属性名为 *title*）

```
get title(){
    // 处理默认参数;
    return this.getAttribute("title") ?? '默认标题';
}
```

59、*shadow DOM*

60、实现链式操作：返还 *this*

61、正则表达式创建

①字面量创建

```
// 1.字面量创建
let str = "abc1231dfaf123213fda";
let reg = /\d+/g;
let res = str.match(reg);
console.log(res);
```

※ \d 不需要有转义字符：\（即 \d 即可）

※ 无法通过变量赋值的形式进行匹配：如下图，此时匹配依然只匹配“abc”这三个字符，而不匹配变量 abc 所代表的字符 “1231”

```
// 1.字面量创建
let str = "abc1231dfaf123213fda";
// let reg = /\d+/g;
let abc = "1231"
let reg = /abc/g;
let res = str.match(reg);
console.log(res);
```

②构造函数创建

```
// 2.构造函数创建
let str = "abc1231dfaf123213fda";
let reg = new RegExp("\\d+","g");
let res = str.match(reg);
console.log(res);
```

※ \d 需要有转义字符：\（即 \\d）

※ 可以通过变量赋值的形式进行匹配：如下图，可以匹配变量 abc 所代表的字符 “1231”

```
// 2.构造函数创建
let str = "abc1231dfaf123213fda";
// let reg = new RegExp("\\d+","g");
let abc = "1231";
let reg = new RegExp(abc,"g");
let res = str.match(reg);
console.log(res);
```

62、正则表达式方法

test 检查字符串中是否有对应字符，若有，返回 true，若没有，返回 false

exec 匹配字符串中是否有对应字符，若有，返回该对应字符及其 index 等集合的对象，且每条 exec 语句仅执行一次，均是在前条 exec 语句执行完的基础上（即从 reg.lastIndex 开始）继续执行。

63、字符串方法

split 切割

search 寻找，只返回第一个匹配的字符的 index，加了全局匹配 g 也只返回第一个

match 匹配，返回匹配的字符，可以全局匹配 g

replace 替换，将指定字符串 A 替换为指定字符串 B

64、元字符

字符相关

\w 数字、字母、下划线

\W 非数字、字母、下划线

\d 数字

\D 非数字

\s 空格

\S 非空格

. 非 \n（回车） \r（换行） \u2028（段落结束符） \u2029（行结束符）

数量相关

{} 次数

eg：{3} 出现 3 次

{1,4} 出现 1–4 次，即[1,4]闭区间

{1,} 出现 1–无限次，即[1,+∞）

? 0 个或 1 个，等同于 {0,1}

默认是贪婪匹配，要变成惰性匹配则加上 ? 即可

eg：/\d{2,4}/g 贪婪匹配会匹配 4 个数字，而不是 2 个或 3 个

/\d{2,4}?/g 惰性匹配会匹配 2 个数字

* 0 个或多个，等同于 {0,}

+ 一个或多个，即有就行，等同于 {1,}

位置相关

^ 开始的位置

```
let str = "abedef";
let reg =/^/g;
let res =  str.replace(reg,"*");→*abedef
```

```
let str = "abedef";
let reg =/^\w/g;
let res =  str.replace(reg,"*"); → *bedef
```

$ 结尾的位置，与^类似

\b 边界符，非\w 的都是边界（例如空格等）

\B 非边界

括号相关

() ①分组

```
// 查找  “{{}}”;{{ message }}
let reg = /\{\{\s*([^\{\}\s]+)
\s*\}\}/g;
let textContent = node.
textContent;
// console.log(textContent);
if( reg.test(textContent) ){
    let $1 = RegExp.$1;
```

用()在正则表达式里进行分组，RegExp.$1 即第一个分组（即第一组括号）中的内容。同理，RegExp.$2 即第二个分组（即第二组括号）中的内容，以此类推。

②提取值

```
let str = "2020-01-02";
let reg = /(\d{4})-(\d{2})-(\d{2})/;
console.log( str.match(reg) );
```

↓

```
(4) ["2020-01-02", "2020", "01",
"02", index: 0, input:
"2020-01-02", groups: undefined]
  0: "2020-01-02"
  1: "2020"
  2: "01"
  3: "02"
  index: 0
  input: "2020-01-02"
  groups: undefined
  length: 4
▶ __proto__: Array(0)
```

③替换

```
let str = "2020-01-02"; // 01/02/2020
let reg = /(\d{4})-(\d{2})-(\d{2})/;
let res =  str.replace(reg,"$2/$3/$1");
console.log(res)
```

或

```
let str = "2020-01-02"; // 01/02/2020
let reg = /(\d{4})-(\d{2})-(\d{2})/;
// let res =  str.replace(reg,"$2/$3/$1");
let res =  str.replace(reg,function(arg,year,
mouth,date){
    return mouth +"/" + date + "/" +year;
});
console.log(res)
```

↓

```
01/02/2020
```

④反向引用

```
let className = "news-container-nav"; //
news_container_nav
let reg = /\w{4}(-|_)\w{9}(\1)\w{3}/;
let res =  reg.test(className);
console.log(res);
```

若要保持连接符号一致（即要么全用 - ，要么全用 _ ），则需要后面的连接符号正则表达式部分与前面的连接符号正则表达式检测到的部分一致，因此可以用（\1）反向引用前面的（-|_）检测到的 -（图中例子为 - ），而不是（-|_）代表的 - 或 _ 。其中（\1）指的是匹配前面第一个（-|_）检测到的 - ，如果前面有好几个（-|_），而想要匹配到第二个（-|_）检测到的连接符，则使用（\2），以此类推。

[] 字符集合

[^0-9] 匹配非数字

65、匹配模式

g 全局匹配

i 忽略大小写

m 多行模式

s 让“.”可以匹配换行模式

u 匹配 unicode 编码

y 粘性模式

66、命名分组

```
    <meta
    http-equiv="X-UA-Compatible"
    content="ie=edge">
    <title>Document</title>
</head>
<body>

</body>
<script>
// 命名分组;
let str = "2020-01-06";
let reg = /(?<year>\d{4})-(?<month>\d
{2})-(?<day>\d{2})/;
console.log( str.match(reg));
```

```
9-命名分组及零宽断言.html:53
(4) ["2020-01-06", "2020",
"01", "06", index: 0, input:
"2020-01-06", groups: {…}]
  0: "2020-01-06"
  1: "2020"
  2: "01"
  3: "06"
  index: 0
  input: "2020-01-06"
  groups:
    year: "2020"
    month: "01"
    day: "06"
  length: 4
  __proto__: Array(0)
>
```

67、零宽断言

```
// let reg = /(?<year>\d{4})-(?
<month>\d{2})-(?<day>\d{2})/;
// console.log( str.match(reg));


// 零宽断言;
// 正向肯定零宽断言;
let str =
"iphone3iphone4iphone5iphonenumber";
// 肯定
let reg = /iphone(?=\d)/g;
let res = str.replace(reg,"苹果");
console.log(res);
```

```
苹果 9-命名分组及零宽断言.html:61
3苹果4苹果5iphonenumber
>
```

```
// let reg = /(?<year>\d{4})-(?
<month>\d{2})-(?<day>\d{2})/;
// console.log( str.match(reg));


// 零宽断言;
// 正向肯定零宽断言;
let str =
"iphone3iphone4iphone5iphonenumber";
// 肯定
// let reg = /iphone(?=\d)/g;
// 否定
let reg = /iphone(?!\d)/g;
let res = str.replace(reg,"苹果");
console.log(res);
```

```
ipho 9-命名分组及零宽断言.html:64
ne3iphone4iphone5苹果number
>
```

```
// // let reg = /iphone(?=\d)/g;
// // 否定
// let reg = /iphone(?!\d)/g;
// let res = str.replace(reg,"苹果");
// console.log(res);

// 负向零宽断言;
let str = '10px20px30pxipx';
// 肯定
let reg = /(?<=\d+)px/g;
let res =  str.replace(reg,"像素");
console.log(res);
```


```
// // let reg = /iphone(?=\d)/g;
// // 否定
// let reg = /iphone(?!\d)/g;
// let res = str.replace(reg,"苹果");
// console.log(res);

// 负向零宽断言;
let str = '10px20px30pxipx';
// 肯定
// let reg = /(?<=\d+)px/g;
// 否定
let reg = /(?<!\d+)px/g;
let res =  str.replace(reg,"像素");
console.log(res);
```

68、迭代

迭代语句

for...in：以原始插入的顺序迭代对象的可枚举属性

for...of：根据迭代对象的迭代器具体实现迭代对象数据

```
for(let attr in arr){
    console.log(attr);
}
for(let val of arr){
    console.log(val);
}
```

for in：循环数组的下标（*0*、*1*、*2*、*3*……）

for of：循环数组里的值（*arr[0]*、*arr[1]*、*arr[2]*、*arr[3]*……）

69、解决异步问题

①回调函数

https://segmentfault.com/a/1190000017935821

②自定义事件

```
// 2.自定义事件;
let myeventObj = new EventTarget();

function test(){
    setTimeout(() => {
        console.log("test");
        myeventObj.dispatchEvent(new CustomEvent("myevent"));
    }, 1000);
}
test();
myeventObj.addEventListener("myevent",function(){
    console.log(2222);
})
myeventObj.addEventListener("myevent",()=>{
    console.log(333);
})
```

③promise

https://www.qdtalk.com/2018/12/25/javascript-promise/

https://www.qdtalk.com/2018/12/26/javascript-promise1/

https://www.qdtalk.com/2018/12/29/javascript%E5%BC%82%E6%AD%A5%E4%B9%8Bpromise-resolve%E3%80%81promise-reject/

https://www.qdtalk.com/2019/01/01/promise/

```
//  三种状态: resolved(fulfilled)、
rejected、pending
// then的2个参数: onResoleved / onRejected;
// then: 三种返还值;

  let p1 = new Promise((resolve,reject)=>{
    // resolve("value11");
    reject("err");
  })
//   console.log(p1);
p1.then((res)=>{
    // onResoleved成功
    console.log("成功",res);
},err=>{
    // onRejected失败
    console.log("失败",err);
})

// then: 三种返还值;  1.没有任何返还  会自动返还
promise对象;  2.返还普通值, 会包装成promise对象
promisevalue值是 返还的值; 3.返还promise  就
会 直接把promise返还;
let res =  p1.then(res=>{
    return new Promise(resolve=>{
        resolve("success");
    })
})
```

```
Promise.all([p1,p2]).then(res=>{
    console.log(res);
})

race  是谁执行快获取谁的结果;
Promise.race([p1,p2]).then(res=>{
    console.log(res);
})
```

```
let res =  p1.then(res=>{
    console.log(res);
},err=>{
    console.log(err);
}).finally(()=>{
    console.log("完成了");
})
```

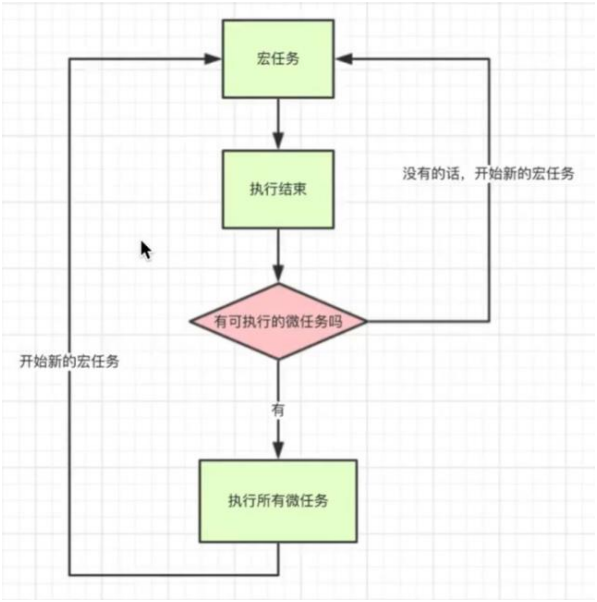
④async、await

https://www.qdtalk.com/2019/01/03/javascript-async-2/

https://www.qdtalk.com/2019/01/04/javascript-async-3/

```
const p1 = function () {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(111);
        }, 1000)
    })
}
const p2 = function () {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            // resolve(2222);
            reject("err");
        }, 2000)
    })
}
// 链式调用;
// p1().then(res=>{
//     console.log(res);
//     return p2();
// }).then(res=>{
//     console.log(res);
// }).cath(err=>{
//     console.log(err);
// })
async function asyncFn() {
    try {
        let res1 = await p1();
        console.log(res1);
        let res2 = await p2();
        console.log(res2);
    }catch(err){
        console.log(err);
    }
}
asyncFn();
```

70、微任务、宏任务：在一个宏任务中，微任务会先执行



Promise、MutationObserver、nodejs 里的 process.nextTick 为微任务
setTimeout 为宏任务

71、数据劫持

①defineProperty

如要劫持多个对象需要循环

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

②Proxy

比 defineProperty 更强大，性能更好，但可能不太兼容 ES6

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy

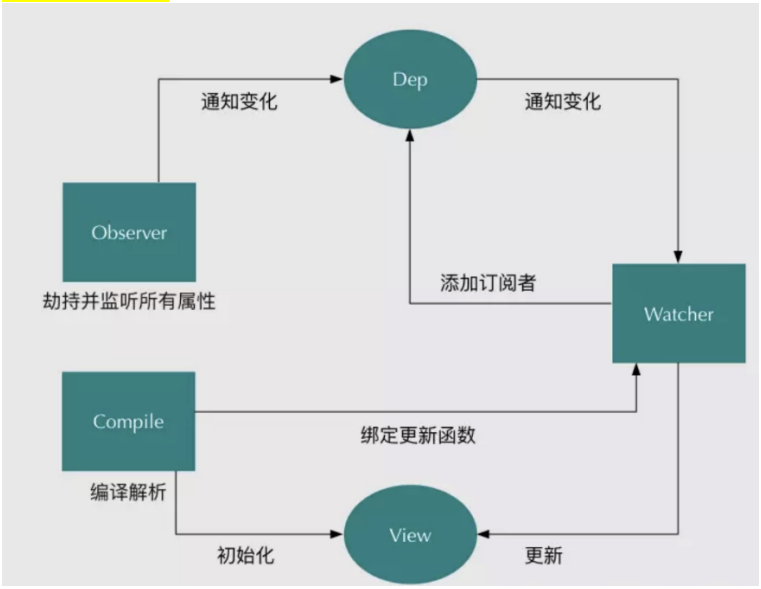
defineProperty 与 Proxy 的区别：https://www.jianshu.com/p/2df6dcddb0d7

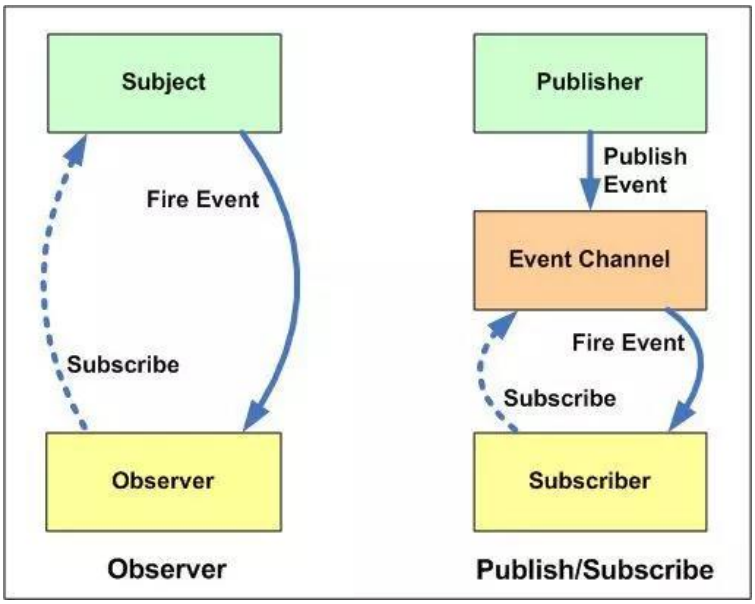
72、数据响应式、数据双向绑定（以 vue 的 MVVM 为例）

```
// 2.如何实现数据响应式？  --->数据劫持defineProperty--->触发二次编译 自定义事件（有关联）--->优化-->发布订阅;
```

https://juejin.cn/post/6844904183938678798

73、发布订阅





74、前端模块化：ESM、commonjs（nodejs）、AMD（require.js）、CMD（sea.js）

https://www.cnblogs.com/chenwenhao/p/12153332.html

在 NodeJS 之前，由于没有过于复杂的开发场景，前端是不存在模块化的，后端才有模块化。NodeJS 诞生之后，它使用 CommonJS 的模块化规范。从此，js 模块化开始快速发展。模块化的开发方式可以提供代码复用率，方便进行代码的管理。通常来说，一个文件就是一个模块，有自己的作用域，只向外暴露特定的变量和函数。目前流行的 js 模块化规范有 CommonJS、AMD、CMD 以及 ES6 的模块系统即 ESM。

75、字符串反转（ABC 变成 CBA）

```
msg.split("").reverse().join("")
```

76、forEach()、for in、for of

https://www.cnblogs.com/yangyangxxb/p/10036784.html

forEach()和 for of 最好用于遍历数组，for in 最好用于遍历对象。

77、三元表达式

(expr1) ? (expr2) : (expr3)

在 expr1 求值为 TRUE 时的值为 expr2，在 expr1 求值为 FALSE 时的值为 expr3。

78、cilck()和 onclick 的区别

https://www.feiniaomy.com/post/340.html