八. ES6

不改变原理基础上尽量简化代码

1.变量声明:

let/const: 代替 var/const 来声明变量和常量的

ES5 缺点:

a.因为传统的 var 和 const 会被声明提前，ES6

b.没有块级作用域: for if while 都不是作用域，其中的变量会被提前到外部，影响外部的代码。

ES6 优点:

a.不会被提前，不允许提前使用未声明的变量

b.让 for if while 等程序块，也变为作用域

原理: let 自动添加匿名函数自调来划分临时作用域，并将变量名前自动加_，与其他变量区分。

2.参数增强:

①默认参数(default): 即使用户不传入参数，参数也有备用的默认值代替

步骤: 定义函数时: function fun(形参,…,形参=默认值)

强调: 有默认值得参数，必须在形参列表的末尾

②剩余参数(rest): 代替 arguments 接收所有不确定个数的参数

arguments: 2 个问题:

a.不是数组类型，不能使用数组 API

b.只能获得全部参数，无法有选择的获得部分

语法: function fun(形参, …数组名)(定义时)

优点:数组名获得的是纯数组，且有选择的获得部分，数组可收集除前几个确定参数之外的剩余参数

```
function calc(ename,...bonus){
document.write(`${ename}的总工资是${bonus.reduce((prev,elem)=>prev+elem)}</br>`)}
calc("lilei",1200,1300,10000)
 // lilei 的总工资是 12500
```

③ 散播(spread): 代替 apply 专门用于打散数组为单个元素，再传入函数。

apply 的问题: 主要功能是替换 this，顺便可以打散数组为单个值。(不是主业)

步骤：调用函数时: fun(...数组)

```
console.log(Math.max(2,7,1,5)); //7
var arr=[2,7,1,5];
console.log(Math.max(arr));//null
console.log(Math.max.apply(null,arr)); //7
console.log(Math.max(...arr)); //7
```

3.箭头函数: 对回调函数或匿名函数自调的简写

步骤: 回调函数: function (形参,…){ … }

可简化为: (形参,…)=>{ … }

如果只有一个形参，可省略()
如果函数体中只有一句话，可省略{}
如果这仅有的一句话还是 return xxx，则必须省略 return

特点: 箭头函数内外的 this 是同一个/共通的

总结: 如果希望内外 this 相同时，应该简化，如果反而希望内外 this 不同时，不能简化!

```
var lilei={
    sname:"lilie",
    friends:["Tom","Jake","Jerry","Rose"],
    intr:function(){
        this.friends.forEach(
            (elem)=>{document.write(`${
                this.sname}认识${elem}</br>`)
            }//完整函数.bind(this)
        )
    }
}
lilei.intr();
```

4.解构: 从一个大的对象中抽取想要的部分成员，单独使用。释放到全局，方便调用。

三种:

① 数组解构: 从数组中抽取想要的元素出来，单独使用

步骤: 下标对下标:

var arr=[1,2,3];

↓

var [x,y,z]=arr

console.log(x,y,z)

结果: x=1, y=2, z=3

arr[0]//麻烦，且没有意义

② 对象解构：从对象中抽取想要的成员出来，单独使用

步骤： 属性名对属性名

```
var obj={x:1, y:2, z:3}
          ↓ ↓   ↓
      var {x:a, y:b, z:c}=obj;
obj.x   obj.y    obj.z //麻烦
```

结果: a=1, b=2, c=3

简写:属性名和变量名一致时，只写一个 var {x:x, y:y, z:z}=obj;

可简写为 var {x,y,z}=obj;

③ 参数解构：其实是对象解构在函数传

何时： 多个参数都可选时

步骤： 2 步:

a.定义时:将参数列表定义为对象语法:

//function fun({属性 1:形参 1,属性 2:形参 2,...})

function fun({形参 1,形参 2,...}){ ... }

b.调用时：将传入的参数放在一个对象中整体传入

```
fun({
   属性 1: 值 1, ...
})
```

执行: fun 将整个实参对象传给形参对象，形参对象通过解构，从实参对象中抽取对应的参数值。

如果找不到对应的，则形参值默认为 undefined

```
function ajax({url,type,data}){
    type=type||"get"
    document.write(`向服务器${url}发送${type}类型请求`)
    if(data) document.write(`携带数据${data}</br>`)
    else document.write(`无参数</br>`)
}
ajax({url:"localehost:3000/user/getUser",
      type:"post",
      data:"id=2"})
```

5.for of：最简化的遍历索引数组或类数组对象的方式：

总结：遍历索引数组:

①for(var i=0;i<arr.length;i++){

var elem=arr[i];

}

最灵活，万能

②arr.forEach((elem,i,arr)=>{ elem})

无法控制遍历顺序和步调

③for(var elem of arr){ ... }

of 会依次取出 arr 中每个元素值，保存在 of 前的变量中

无法获得位置 i

无法修改原数组中的值

for of 和 for in 比较

for of 专门遍历下标为数字的索引数组/类数组对象----数字下标

of 取得是元素值

for in 专门遍历下标为自定义名称的关联数组/对象----自定义下标

in 取得是属性名

6.class:

ES6 对整个面向对象语法的简化

① 对象直接量的简化

对象名和变量名相同的保留一个

方法去掉 “：function”

```
//对象直接量的简化
var sname="lilei",sage=11;
var lilei={
    sname,  //sname:sname
    sage,   //sage:sage
    intr(){ //:function(){
        console.log(`I'm ${
            this.sname},I'm ${sage}`)
    }}
console.log(lilei);
lilei.intr();
```

② 对创建一种类型的简化:

定义： 集中描述一类对象统一属性结构和行为的程序结构，今后只要创建一种类型，必须用 class

步骤：

a.用 class{}包裹原来的构造函数和原型对象方法

b.构造函数名提升为 class 名，构造函数要更名为 constructor

c.直接定义在 class 内的方法，默认保存在原型对象中，且不用加 Xxx.prototype 前缀和”=function”

```
class student{
  constructor(sname,sage){
  this.sname=sname;
  this.sage=sage;
  }
  intr(){console.log(`I'm ${this.sname},I'm ${sage}`)}
}
  var lilei=new student("lilei",18)
  console.log(lilei);
```

③继承: 2 步

a.class child extends father{}

不再需 Object.setPrototypeOf(...)

b.子类型构造函数中: super(参数值)

不需要传入 this，super 自动指向 extends 后的父类型，super()是调用父类型构造函数的意思

super 必须放在构造函数第一句

```
//准备: 先将Enemy和Plane两种类型改为class
class Enemy{
  constructor(fname,speed){
    this.fname=fname;
    this.speed=speed;
  }
  fly(){
    console.log(`${this.fname}以时速${this.speed}飞行`);
  }
}
//1.让子类型继承父类型: Plane extends Enemy
//不再需要Object.setPrototypeOf
class Plane extends Enemy{
  constructor(fname,speed,score){
    //2.用super(fname,speed)调用父类型构造
    //不再需要call，不再需要传this
    super(fname,speed);
    this.score=score;
  }
  getScore(){
    console.log(`击落${this.fname}得${this.score}分`);
  }
}

var f16=new Plane("F16",1000,5);
console.log(f16);
f16.fly();
f16.getScore();
```

④访问器属性: 依然要在构造函数内定义受保护的隐藏的数据属性

在 class 内:

get 访问器属性名(){ return this. ...}

set 访问器属性名(value){ 验证 value 并给 this.xxx 赋值}

```
class emp{
    constructor(eid,ename,age){
        this.eid=eid,
        this.ename=ename,
        Object.defineProperty(emp,"_age",{
            writable:true,
            enumerable:false
        }),
        this.age=age;
    }
    get age(){
        return this._age
    }
    set age(value){
        if(value>=18&&value<=65)this._age=value
        else throw Error("年龄超限")
    }
}
var lilei=new emp(1001,"lilei",18)
console.log(lilei)
```

⑤静态方法:

class 类名{

constructor(){}

方法(){}

static 静态方法 (){ }

}

```
class product{
   constructor(){};
    save(){
        console.log('保存商品')
    }
    static getbyid(){
    console.log('查找商品')
}
}
var p1=new product();
p1.save();
product.getbyid();
```

⑥Promise: 代替回调函数，实现多个异步调用，顺序执行(前一件事办完，再办)

错误做法：仅顺序编写

test1();

test2();

传统的做法: 利用回调函数:

```
function liang(callback){
    console.log("亮起跑...")
    setTimeout(function(){
        console.log('亮到达终点');
        callback();
    },6000)
}
function ran(callback){
    console.log("然起跑...")
    setTimeout(function(){
        console.log('然到达终点');
        callback();
    },4000)
}
function dong(){
    console.log("东起跑...")
    setTimeout(function(){
        console.log('东到达终点');
    },6000)
}
liang(function(){
    ran(
        function(){
            dong()
        }
    );
});
```

问题: 使用回调函数方式实现多个异步调用顺序执行会多级函数嵌套，导致回调地狱(callback hell)

根源: 所有回调函数规定，在调用函数前，就要提前传入到函数中

解决: 让回调函数在函数后传入

步骤:

前提: 不要在参数列表里传递回调函数了!

（1）定义函数支持 Promise

在原函数内，用 new Promise()包裹所有原代码

再在源代码外层套一层 function(){}

function()中必须接收 Promise 附赠的 open 开关

在当前函数异步任务调用后，自动打开开关 open();

```
function fun(){
    return new
Promise(function(open){
            异步任务
            异步任务执行完:open()
  })
  }
```

（2）将多个任务串联起来:

```
第一个函数().then(第二个函数).then(…)
```

强调: 中间的 then 中的函数，不要加()，因为不是立刻执行，且中间的函数必须支持 Promise

错误处理: 2 步:

① new Promise(function(open,err){
  //如果出错:
   err(“错误消息”)
  //通向最后的.catch()
 })

②在函数 1().then().then()…..catch(function(errMsg){ … })

无论中间哪个 then 出错，都会执行最后的 catch，并将 then 中 err(“错误消息”)传给 errMsg。

```
function liang(){return new Promise(
    function(open){
    console.log("亮起跑...")
    setTimeout(function(){
        console.log('亮到达终点');
        open();
    },6000)
})
}
function ran(){return new Promise(
    function(open){
    console.log("然起跑...")
    setTimeout(function(){
        console.log('然到达终点');
        open();
    },4000)
})
}
function dong(){
    console.log("东起跑...")
    setTimeout(function(){
        console.log('东到达终点');
    },2000)
}
liang().then(ran).then(dong);
```

DOM

一．什么是 DOM: Document Object Model

DOM:专门操作网页内容的 API 标准--W3C

优点：统一所有浏览器操作网页内容的 API

几乎所有浏览器 100%兼容 DOM API

包括: 增删改查+事件绑定

查找网页上的元素

修改元素

添加新元素

删除现有元素

事件绑定

使用：只要 js 操作网页内容，只能用 DOM API

二．DOM Tree

①定义：内存中保存一个网页中所有内容的树形结构

优点：因为网页中的内容，都是有明显的上下级包含关系的。

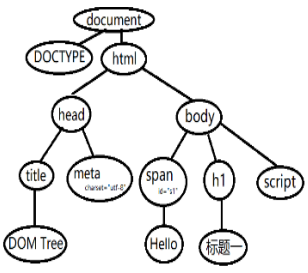
使用：只有理解了网页的树型结构，才能顺利找到要操作的 DOM 元素

步骤：

a.当浏览器读到一个 HTML 文件时，会自动在内存中创建一个唯一的树根节点：document

b.浏览器解析 HTML 中的内容，每读到一项内容，就创建一个节点对象，然后将节点对象添加 DOM 树上

```
.html
<!DOCTYPE HTML>
<html>
    <head>
        <title>DOM Tree</title>
        <meta charset="utf-8" />
    </head>
    <body>
        <span id="s1">Hello</span>
        <h1>标题一</h1>
        <script></script>
    </body>
</html>
```



②节点对象:

网页中每一项内容(元素，属性，文本)都是一个节点对象

节点对象的三个公共属性：(了解)

.nodeType: 判断该节点对象的类型，值为一个数字

包括: document 9 根节点

element 1 元素节点

attribute 2 属性节点

text 3 文本节点

问题: 只能判断类型，不能进一步判断元素名

③.nodeName: 判断节点的名称:

包括: document #document

Element 全大写的标签名

Attribute 获得属性名

Text #Text

.nodeValue: 获得节点的值

包括: document null

Element null

Attribute 属性值

Text 文本内容

三．查找: 四种

1. 不需要查找，就可直接获得的元素:

<html> document.documentElement

<head> document.head

<body> document.body

<form> document.forms[i/id]

2. 按节点间关系查找:

①**节点树**: 包含所有网页内容的最完整的树结构

两大类关系:

a.父子关系:

node.parentNode 获得 node 节点的父节点

node.childNodes 获得 node 下所有直接子节点的集合

node.firstChild 获得 node 下第一个直接子节点

node.lastChild 获得 node 下最后一个直接子节点

b.兄弟关系:

node.previousSibling 获得 node 节点的前一个兄弟

node.nextSibling 获得 node 节点的后一个兄弟

问题: 节点树会受到看不见的回车，换行，空字符的影响

程序员通常只关心元素节点

解决:元素树,仅包含元素节点的树结构

②元素树

好处:不包含看不见的空字符

强调: 元素树不是一棵新树，只是节点树的子集。

两大类关系:

a.父子关系:

node.parentElement 获得 node 节点的父元素

node.children 获得 node 下所有直接子元素的集合

node.firstElementChild 获得 node 下第一个直接子元素

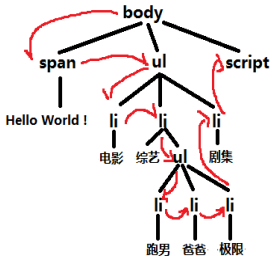
node.lastElementChild 获得 node 下最后一个直接子元素

b.兄弟关系:

node.previousElementSibling 获得 node 节点的前一个兄弟元素

node.nextElementSibling 获得 node 节点的后一个兄弟元素

```
<body>
  <span>Hello World !</span>
  <ul id="nav">
    <li>电影</li>
    <li>综艺
      <ul>
        <li>跑男</li>
        <li>爸爸</li>
        <li>极限</li>
    <li>剧集</li>
  <script></script>
```



③递归遍历: 指定父元素下所有后代元素，2 步:

a. 先定义函数，仅遍历指定父元素的直接子元素

b.对每个直接子元素，调用和父元素完全相同的操作

```
function getChildren(parent){
    var children=parent.children;
    for(var child of children){
        console.log(child.nodeName);
        getChildren(child)
    }
    }
window.onload=function(){
    getChildren(document.body)
}
```

问题: 必须先获得一个元素，才能按节点间关系查找

解决: 用 HTML 特征查找

3.按 HTML 特征查找: 4 种

id tag name class

何时使用：在没有获得任何元素的情况下，可作为首次查找之用

① 按 id 查找:

var elem=document.getElementById("id");

返回值: 返回一个元素对象，找不到，返回 null

强调: 只能用 document 调用

② 按标签名查找:

var elems=parent.getElementsByTagName(“tag”)

返回值: 返回所有符合条件的元素对象的集合

如果找不到，返回空集合: [].length=0

强调: a.可在任意父元素上调用

b.不仅查找直接子元素，且在所有后代中查找，返回多个元素组成的类数组对象

③ 按 name 属性查找:

var elems=document.getElementsByName(“name”)

使用： 专门查找表单中收集数据的表单元素

返回值: 多个表单元素的集合，如果找不到返回[]

如果明知道只找到一个元素: elems[0]

强调: 只能用 document 调用

返回多元素组成的类数组对象

④ 按 class 属性查找:

var elems=任意父元素.getElementsByClassName(“class”)

返回值: 多个元素的集合，如果找不到返回[]

强调: a.可在任意父元素上调用，控制查找范围

b.返回多元素组成的类数组对象

c.不仅查找直接子元素，且在所有后代中查找

d.如果一个元素有多个 class 饰，则只需要其中一个 class 就可找到该元素。

问题: 一次只能用一个条件查找，如果查找条件复杂，代码会很繁琐！