# ECMA6

## 变量

var ：

1. 可以重复声明
2. 无法限制修改
3. 没有块级作用域

let ：不能重复声明，变量—可以修改，有块级作用域

const ：不能重复声明，变量—不可以修改，有块级作用域

## 箭头函数

普通函数：

let **show** = function(){

**alert**('123');

}

箭头函数（去掉function再加上’=>’）：

let **show** = ()=>{

**alert**('123');

}

let array = [12, 5, 3, 5, 63];

array.**sort**(function (n1, n2) {

return n1 - n2;

});

array.**sort**((n1, n2) =>{

return n1 - n2;

});

**alert**(array);

1. 如果只有一个参数，（）可以省略

let **show** = function(a){

return a \* 2;

}

let **show** = a => a \* 2

1. 如果只有一个return，（）可以省略

let **show** = function(a, b){

return a \* b \* 2;

}

let **show** = (a, b) => a \* b \* 2;

## 函数的参数

1. 参数的扩展/数组展开

function **show**(a, b, ...args) {

**alert**(a); *//12*

**alert**(b); *//15*

**alert**(args); *//8,9,20,21,90*

}

**show**(12, 15, 8, 9, 20, 21, 90);

1. 收集剩余的参数

function **show**(a, b, ...args)

\*Rest Parameter必须是最后一个

1. 展示数组

let arr1 = [1, 2, 3];

let arr2 = [5, 6, 7];

let arr = [...arr1, ...arr2];

let arr = [1, 2, 3, 5, 6, 7];

\*展开后的效果，跟直接把数组的内容写这一样

function **show**(...args) {

**fun**(...args)

}

function **fn**(a, b) {

**alert**(a + b);

}

**show**(1, 2);

1. 默认参数

function **show**(a, b = 5, c = 12) {

**alert**(a, b, c);

}

**show**(99); *//99 5 12*

**show**(99, 10, 14); *//99, 10, 14*

## 解构赋值

1. 左右两边结构必须一样

let arr = [1, 2, 3];

a = arr[0];

b = arr[1];

c = arr[2];

let [a, b, c] = [1, 2, 3];

consloe.**log**(a, b, c);

let { a, b, c } = {

a: 5,

b: 10,

c: 15

}

let [{a, b}, [n1, n2, n3], num, str] = [{a:5, b:12}, [12, 5, 10], 8, "hello"];

**console**.**log**(a, b, n1, n2, n3, num, str); *//5, 12, 12, 5, 10, 8, 'hello'*

“拆分粒度”

let [json, arr, num, str] = [{a:5, b:12}, [12, 5, 10], 8, "hello"];

**console**.**log**(json, arr, num, str); *//Object {a:5, b:12} (3) [12, 5, 10] 8 "hello"*

1. 右边必须是个合乎语法的东西

let {a, b} = {12, 5};

**console**.**log**(a, b) *//编译错误，右侧既不是数组格式，又不是Json格式*

1. 声明和赋值不能分开（必须在一句话里完成）

let [a, b];

[a, b] = [5, 12]; *//编译错误*

## 数组

1. map 映射 一个对一个

[12, 58, 99, 86, 45, 91];

['不及格', '不及格', '及格', '及格', '不及格', '及格']

let arr = [12, 5, 8];

let result = arr.**map**((item) => item \* 2); *//[24, 10, 16]*

let score = [19, 85, 99, 25, 90];

let result = score.**map**((item) => item >= 60 ? '及格' : '不及格');

**alert**(result); *//['不及格', '及格', '及格', '及格', '不及格', '及格']*

1. reduce 汇总 一堆出来一个

算总数： [1, 2, 3] => 6

算平均数： [1, 2, 3] => 2

let arr = [12, 69, 180, 8763];

let result = arr.**reduce**(function(tmp, item, index){

return tmp + item;

})

1. filter 过滤器

let arr = [12, 5, 8, 99, 27, 36, 75];

let result = arr.**filter**((item) => item % 3 == 0);

**alert**(result); *//12, 99, 27, 36, 75;*

1. forEach 循环（迭代）

let arr = [12, 5, 8, 9];

arr.**forEach**((item, index) => {

**alert**(index + ":" + item);

})

## 字符串

1. 多了两个新方法
   1. startsWith

let str = "http://it.kaikeba.com"

if(str.**startsWith**('http://')){

**alert**("普通网址");

}else{

*//......*

}

* 1. endWith

let str = "1.txt"

if(str.**endWith**('.txt')){

**alert**("文本文件");

}else{

*//......*

}

1. 字符串模板：

let title = "标题";

let content = "内容"

let str2 = `

<div>

<h1>${title}</h1>

<p>${content}</p>

</div>

`

1. 直接把东西塞到字符串里面去 $(东西)
2. 可以折行 反单引号`

## 面向对象

ES5的语法：

function **User**(name, age){

this.name = name;

this.age = age;

}

**User**.prototype.**showName** = function(){

**alert**(this.name);

}

**User**.prototype.**showAge** = function(){

**alert**(this.age);

}

ES6的面向对象

class **User**{

constructor(name, age){

this.name = name;

this.age = age;

}

**showName**(){

**alert**(this.name);

}

**showAge**(){

**alert**(this.age);

}

}

1. class关键字、构造器和类分开了
2. class里面直接加方法

继承：

ES5的继承

function **VIPUser**(name, age, level){

User.**call**(this, name, age);

this.level = level;

}

**VIPUser**.prototype = new **User**();

**VIPUser**.prototype.constructor = VIPUser;

**VIPUser**.prototype.**showLevel** = function(){

**alert**(this.level);

}

var v1 = new **VIPUser**("Jiang", 20, 5);

ES6的继承

class **VIPUser** extends **User** {

constructor(name, age, level) {

super(name, age);

this.level = level;

}

**showLevel**() {

**alert**(this.level);

}

}

## 面向对象引用——React（依赖于ES6语法）

React：

1. 模块化、组件化——class
2. JSX（JSX == babel==browser.js）JS的扩展版，React强依赖于JSX

## JSON

1. JSON的标准写法：
2. 所有名字都必须用引号包起来

{ a: 12, b: 5 } ×

{ "a" : 12, "b": 5 } √

1. 只能用双引号

{ 'a': 12, 'b': 5 } ×

{ "a" : 12, "b": 5 } √

1. JSON对象

let json = { a: 10, b: 5 };

let str = 'https://www.baidu.com/path/user?data=' + **encodeURIComponent**(JSON.**stringify**(json));

let str = '{ "a" : 12, "b": 5 }';

**console**.**log**(JSON.**parse**(str));

1. 简写
2. 名字一样

let a = 12;

ler b = 5;

let json = {

a : a,

b : b

}

let json = {

a,

b

}

1. 方法

let json = {

a: 10,

**show**: function () {

**alert**(this.a);

}

**show**(){

**alert**(this.a) //去掉function和冒号

}

}

json.**show**();

## Promise—用同步的方式书写异步代码

1. 异步：操作之间没啥关系，同时进行多个操作（AJAX请求…）
2. 同步：同时只能做一件事
3. Promise到底怎么用

const promise = new **Promise**(function (resolve, reject) {

*// ... some code*

if (*/\* 异步操作成功 \*/*) {

**resolve**(value);

} else { //如果异步操作成功，则调用这个方法

**reject**(error);

} //如果异步操作失败，则调用这个方法

});

promise.**then**(function (value) {

*// success*

}, function (error) {

*// failure*

});

Promise实例生成以后，可以用then方法分别指定resolved状态和rejected状态的回调函数

resolve和reject参数都是方法，resolve对应者promise.then方法里的第一个参数，

reject对应者promise.then方法里的第二个参数

promise进行AJAX操作

let promise = new **Promise**(function (resovle, reject) {

*//异步代码，resovle——成功，reject——失败*

$.**ajax**({

url: 'data/arr.txt',

dataType: 'json',

**success**(arr) {

**resovle**(arr);

},

**error**() {

**reject**();

}

})

});

promise.**then**(function (arr) {

**console**.**log**('成功了');

**console**.**log**(arr);

}, function () {

**console**.**logg**('失败了');

})

**//首先需要引入Jquery文件，Jquery里面有AJAX方法**

**Promise**.**all**([

$.**ajax**({ url: 'data/arr.txt', dataType: 'json' }), //data目录下的arr文本文件

$.**ajax**({ url: 'data/json.txt', dataType: 'json' }),//整个$.ajax…是个Promise对象

$.**ajax**({ url: 'data/num.txt', dataType: 'json' }),

]).**then**(results => {

let [arr, json, num] = results;

**console**.**log**(arr, josn, num); *//()[...], Object{}, num*

**alert**('成功了');

}, err => {

**alert**('失败了')

})

### Promise.all()

Promise.all方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

const p = **Promise**.**all**([p1, p2, p3]);

Promise.all方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例

**Promise**.**all**([

p1, p2 *//p1, p2必须是Promise实例对象*

]).**then**(function (arr) { *//若p1的值为数组[1,2,3]，p2的值为json对象{"a":5, "b":10}*

let [res1, res2] = arr; *//则arr的值为[[1,2,3], {"a":5, "b":10}];————解构赋值操作*

**console**.**log**(res1, res2);

}, function () {

**alert**("至少一个失败了")

})

## Generator——生成器

1. 普通函数——一路到底
2. Generator函数——中间能停
3. Generator函数语法（不能用箭头函数形式）：

function \***show**(){ //function和函数名之间有个\*号

**console**.**log**('a');

yield; //yield有暂停的意思

**console**.**log**('b');

}

调用 Generator 函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象

let genObj = **show**();

genObj.**next**(); *//'a'*

genObj.**next**(); *//'b'*

每次调用next方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield表达式（或return语句）为止

1. yield：既可以传参，也可以返回

——传参：

function \***show**(){

**console**.**log**('a');

let i = yield;

**console**.**log**('b');

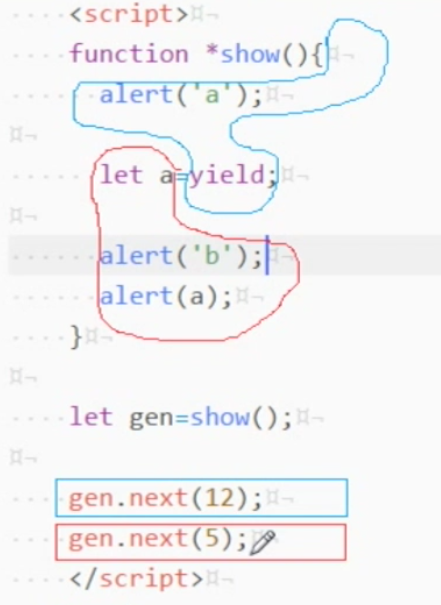
consloe.**log**(i); *// i = 5*

}

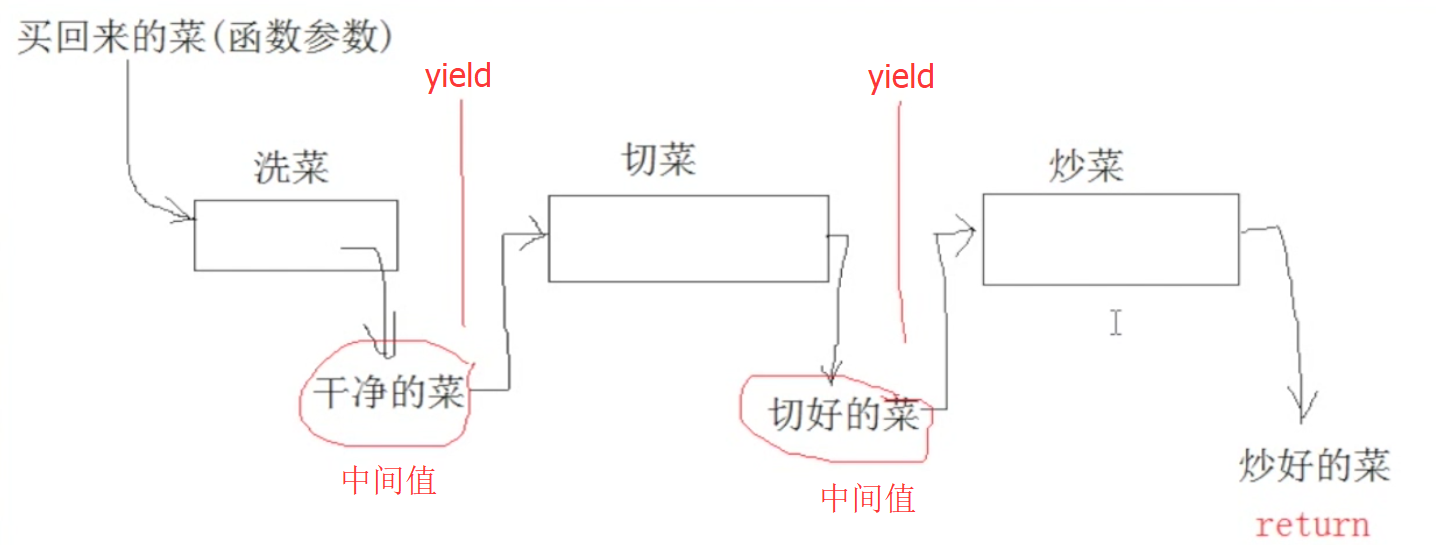
let genObj = **show**();

genObj.**next**(12); *//第一个next()没法给变量传参*

genObj.**next**(5);



——返回



function\* **show**() {

**console**.**log**('a');

yield 12;

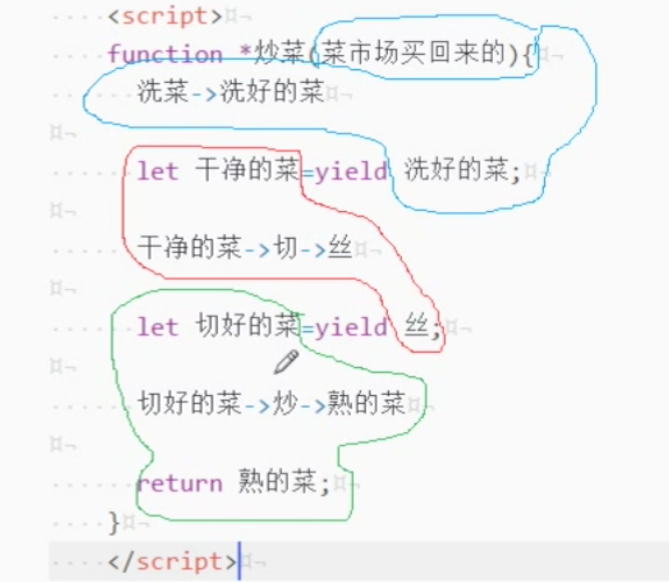
**console**.**log**('b');

}

let genObj = **show**();

let res1 = genObj.**next**(); *//{value:12, done:false}*

let res2 = genObj.**next**(); *//{value:undefined, done:false}，因为没return，所以value值为undefined*



1. generator实例（解决异步操作）

## 异步操作

1. 回调

$.**ajax**({

url: xxx,

dataType: 'json',

**success**(data1) {

$.**ajax**({

url: xxx,

dataType: 'json',

**success**(data2) {

$.**ajax**({

url: xxx,

dataType: 'json',

**success**(data3) {

**alert**('完事了')

},

**error**() {

**alert**('失败了')

}

})

},

**error**() {

**alert**('失败了')

}

})

},

**error**() {

**alert**('失败了')

}

})

1. Promise

**Promise**.**all**([

$.**ajax**({ url: xxx, dataType: 'json' }),

$.**ajax**({ url: xxx, dataType: 'json' }),

$.**ajax**({ url: xxx, dataType: 'json' })

]).**then**(results => {

**alert**('完事了')

}, err => {

**alert**('失败了')

})

1. Genrator

function\* **show**() {

let data1 = yield $.**ajax**({ url: xxx, dataType: 'json' });

let data2 = yield $.**ajax**({ url: xxx, dataType: 'json' });

let data3 = yield $.**ajax**({ url: xxx, dataType: 'json' });

*//完事了*

}

1. 带逻辑——Promise

**Promise**.**all**([

$.**ajax**({ url: 'getUserData', dataType: 'json' })

]).**then**(results => {

let userData = results[0];

if (userData.type == 'VIP') {

**Promise**.**all**([

$.**ajax**({ url: 'getVIPItems', dataType: 'json' })

]).**then**(results => {

let items = results[0];

*//生成列表...*

}, err => {

**alert**('失败了')

})

} else {

**Promise**.**all**([

$.**ajax**({ url: 'getItems', dataType: 'json' })

]).**then**(results => {

let items = results[0];

*//生成列表...*

}, err => {

**alert**('失败了')

})

}

}, err => {

**alert**('失败了')

})

1. 带逻辑——回调

$.**ajax**({url:'getUserData', dataType:'json', **success**(userData){

if(userData.type == 'VIP'){

$.**ajax**({url:'getUserData', dataType:'json', **success**(items){

*//生成列表，显示...*

}, **error**(){

**alert**('失败了')

}})

}else{

$.**ajax**({url:'gerItems', dataType:'json', **success**(items){

*//生成列表，显示...*

}, **error**(){

**alert**('失败了')

}})

}

}})

1. 带逻辑——Generation

function\* **show**() {

let userData = yield $.**ajax**({ url: 'getUserData', dataType: 'json' });

if(userData.type == 'VIP'){

let items = yield $.**ajax**({url :'getVIPItems', dataType:'json'});

}else{

let items = yield $.**ajax**({url :'getItems', dataType:'json'});

}

*//声称列表，显示...*

}

1. Promise——一次读一堆
2. Generation——逻辑性

## 撒大声地

## Sad