ES6

**一.let、const**

1.var声明变量，走预编译，变量会提升，let、const不会提升

2.let、const是一个块级作用域if(true){let a}

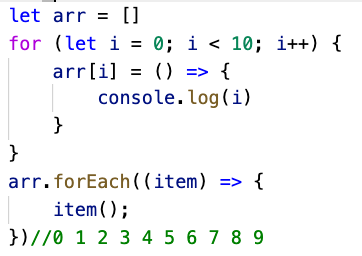
3.let、const不能同名重复声明

4.const除了以上三个，声明常量，一旦被声明，无法改变(只读)，对象的内部属性方法可以修改

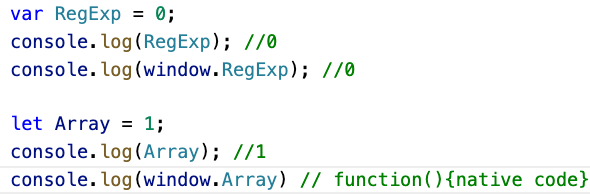
\*.local、169.254/16

**二.let作用**

1.for循环



2.不会污染全局变量



**三.模板字符串（反引号）**

l 

**四.函数**

1.带参数默认值的函数

**es5:**

function add(a, b) {

a = a || 10;

b = b || 10;

return a + b;

}

**es6:**

function add(a = 10, b = 10) {

return a + b;

}

2.默认的表达式也可以是一个函数

**es6:**

function add(a = 10, b = getVal(5)) {

return a + b;

}

function getVal(val) {

return val + 5;

}

3.剩余参数：由三个点紧跟着的巨明参数指定 ...args,解决了arguments的问题

**es6:**

function pick(obj, ...args) { // 解决了arguments的问题

console.log(args); // args就是一个存实参的数组

}

let book = {

title: "es6",

year: 2020

}

pick(book, title, year);

4.扩展运算符也是用...，将一个数组分割，并将各个项作为分离的参数

处理数组中的最大值

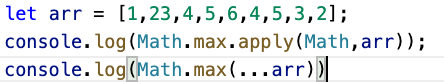
**es5:**

const arr = [212,343,5465,7765,34,65];

console.log(Math.max.apply(null,arr));

**es6:**

console.log(Math.max(...arr));



5.箭头函数=>来定义 取代了function (){} 等于 ()=>{}

**es5:**

function add(a = 10, b ) {

return a + b;

}

**es6:**

let add = (a = 10, b) => {

return a + b;

}

当只有一个参数的时候

let add = c => {

return c + 5;

}

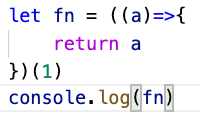
再简写

let add = c => (c + 5);

两个参数

let add = (c , d) =>(c + d);

6.立即执行函数



**五.箭头函数没有this指向**

箭头函数没有this绑定

es5中this指向：取决于调用该函数的上下文对象

**es5:**

let PageHandle = {

id: 123,

init: document.addEventListener("click", function (event) {

this.doSomething(event.type);

}.bind(this), false),//this指向document，只能通过bind去将this指向PageHandle

doSomething: function (type) {

console.log(`事件类型${type},当前id${this.id}`);

}

}

**es6:**箭头函数没有this指向，只能往上一个作用域找！

let PageHandle = {

id: 123,

init: document.addEventListener("click",(event)=>{

this.doSomething(event.type); //指向了init的作用域,PageHandle调用了init，所以指向了PageHandle

},false),

doSomething: function (type) {

console.log(`事件类型${type},当前id${this.id}`);

}

}

注意事项：1.使用了箭头函数就没有arguments了（因为都没作用域链了this都没了，arguments是this.arguments）

2,箭头函数不能作为构造函数，new无用了

**六.结构赋值**

对赋值运算符的一种扩展，针对**数组和对象**进行操作

优点：代码书写上简单易读

对象：

let node = {

type: "string",

name: "foo"

}

**es5:**

let type = node.type;

let name = node.name;

**es6:**

let { type, name } = node; //变量名必须是当前对象的属性名

let { type, ...reg } = node; //可以用剩余运算符，防止丢属性

数组：

let arr = [1, 2, 3];

let [a, b, c] = arr;

嵌套

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

**七.对象的扩展功能**

es6直接写入变量和函数，作为属性和方法

**es5:**

const name = 'tom', age = 20;

const Person = {

name: name,

age: age,

sayName: function () {

console.log(this.name);

}

}

**es6:**

const name = 'tom', age = 20;

const Person = {

name, // name:name, 名相同才可以

age,

sayName() {

console.log(this.name);

}

}

**八.扩展对象的方法**

**1.is()** 比较两个是否严格相等 不全是===

Object.is(NaN,NaN); //true

**2.assign()**用于对象的合并

Object.assign(target,obj1,obj2,...);

把后面所有的对象都合并成target,返回值就是一个新对象

Object.assign({},{a:1},{b:1});

如果合并的两个对象键相同，则取后面的值

**九.新的原始数据类型Symbol，表示独一无二的值 也就堆储存地址是不同的，用于定义对象的私有变量**

const name1 = Symbol('name');

const name2 = Symbol('name');

console.log(name1 === name2);//false

let s1 = Symbol('S1'); //这里是对值的类型进行定义命名（这里名前后是一样最好，方便演示这里不一样，应该都是s1）

let obj = {};

obj[s1] = "Tom";

查看对象obj如下:



取值：obj[s1] 只能这样取值

如果是Symbol定义的对象中的变量，取值时一定要用[变量名]，不能用.变量名

直接往对象里面写值，也不能忘记中括号

let obj = {

[s1]: "Tom";

};

如果是Symbol定义的，是无法遍历的，只能作为私有属性

let s1 =Symbol('S1');

let obj ={

[s1]:'tom'

};

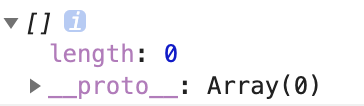
for(let key in obj){

console.log(key); //没有任何输出

}

调用keys方法查看属性：

Object.keys(obj); //是一个空数组

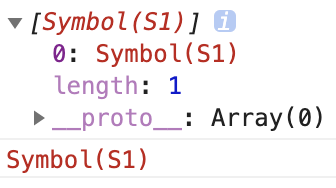


查找的方法：通过getOwnPropertySymbols

let a = Object.getOwnPropertySymbols(obj);

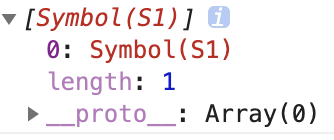
console.log(a);

console.log(a[0]);

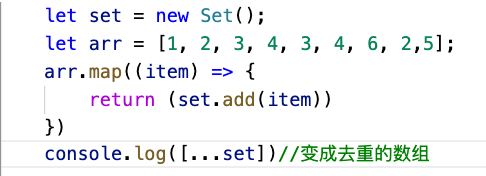


可以通过反射Reflect

console.log(Reflect.ownKeys(obj));



**十.Set数据类型，表示无重复值的有序列表**

****

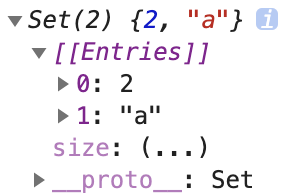
1.创建 let set =new Set();



2.添加元素add

set.add(2);

set.add("a");



3.删除delete

set.delete("a");

4.has方法校验某个值是否在这里集合中

5.foreach 没有意义

let set = new Set();

set.add(2);

set.add("a");

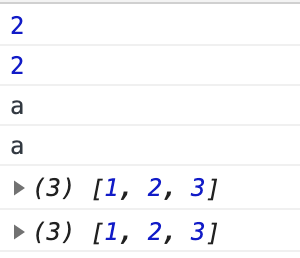
set.add([1, 2, 3]);

set.forEach((val, key) => {

console.log(val);//键为值

console.log(key);//值为值

})



6.set转化成数组

let set = new Set([2, 1, 2, 3, 4, 5, 4, 3]);

console.log(set);



变数组（扩展运算符）

let set = new Set([2, 1, 2, 3, 4, 5, 4, 3]);

let arr =[...set];

console.log(arr)



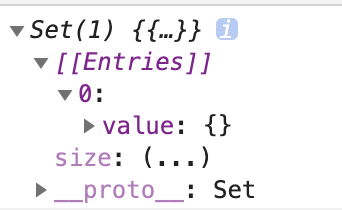
7.set中对象的引用无法被释放,用WeakSet解决

let set = new Set(), obj = {};

set.add(obj);

obj = null;

console.log(set);

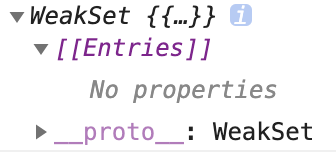


let set = new WeakSet(), obj = {};

set.add(obj);

obj = null;

console.log(set);



WeakSet的特点1.不能传入非对象类型的参数2.无法遍历出来3.没有foreach4.没有size属性（相对于Set少了许多方法属性的）

8.map循环，循环的结果写在return里，多用于react的列表循环输出

返回一个新数组

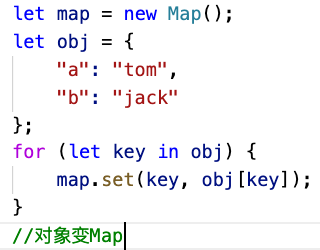
Array.map((item,index,array)=>{

})

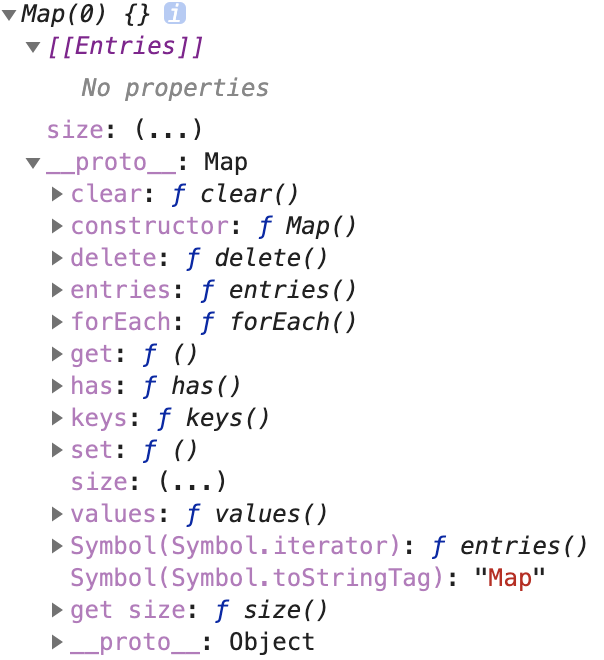
[1,2,3].map(parseInt)

新数组结果[1,NaN,NaN]

**十一.Map类型是键值对的有序列表，键和值是任意类**

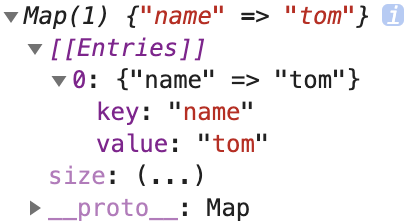
****

let map = new Map();



2.set方法设置键和值

map.set('name','tom');



3.get方法获取值

console.log(map.get('name'));

4.初始化，传入的是一个内部元素是键值对数组的数组

new Map([[1,true],[2,false]]);

5.WeakMap

**十二.数组的拓展方法**

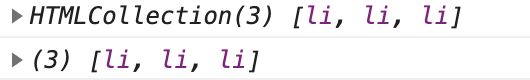
（1）from将类数组转化为数组

**es5:**

let arr = [ ].slice.call(arguments);

**es6:**

****

****

let arr = Array.from(arguments);

运用在<ul><li>

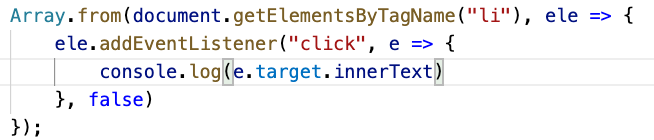
获取li的数组lis

Array.from(lis);

**也可以用扩展运算符**

**[...lis]**

还可以接受第二个参数，回调函数，用来遍历处理每个元素



（2）of将任意类型值，转化到数组

Array.of(1,2,3,4,5,6);

这样就成了一个数组

（3）copywithin复制替换

[1,2,3,4,5,8].copywithin(0,3);表示从3索引开始到结尾的值，给从0索引开始的值替换掉

结果就是[4,5,8,4,5,8]

（4）find和findIndex 找出第一个符合条件的成员

[1,2,3,-3,-5,0].find(n=>n<0)

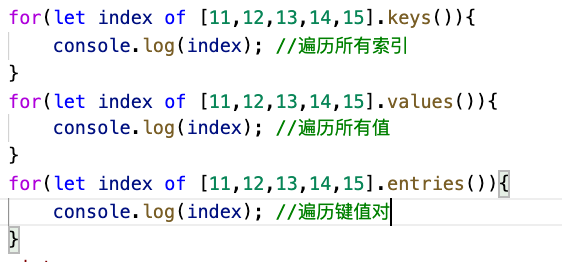
查找n<0的第一个符合条件的值返回

（5）属性entries() keys() values() 返回一个遍历器，可以使用for of循环

console.log([1,2,3,4,5].keys());



然后就可以遍历



（6）遍历器里的next方法，一次一次的遍历，调用一次，往下查阅一组数据

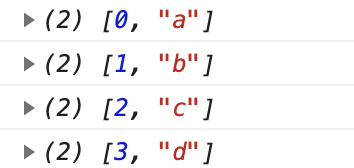
let it = ['a', 'b', 'c', 'd'].entries();

console.log(it.next().value);

console.log(it.next().value);

console.log(it.next().value);

console.log(it.next().value);



（7）includes 返回一个布尔值，表示某个数组是否包含给定的值

console.log([1,2,3,4].includes(1));//true 和indexOf一样，返回值不同，indexOf返回索引和-1

**十三.迭代器**

**有两个核心**

1.是一个接口，是各种数据结构能快速访问的接口，通过Symbol.iterator创建迭代器，用next调用

2.用于遍历数据结构的指针done

Iterator是一种新的遍历机制

属性里提供的方法

const items = ['bob','cdcd','fdf'];

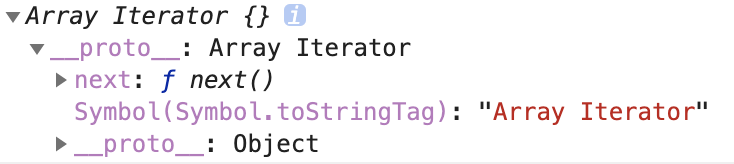
console.log(items);



console.log(items[Symbol.iterator]);

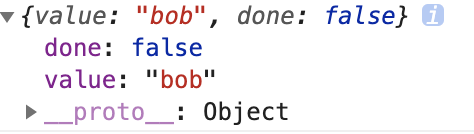


创建迭代器：**const ite = items[Symbol.iterator]();**



有唯一的方法next

console.log(ite.next());//遍历第一次



done表示遍历是否完成



**十四.生成器（与迭代器相呼应）**

generator函数，通过yield关键字将函数停留在当前位置，为改变执行流提供可能，同时为异步编程提供方案

与普通函数的区别：

1.function 后面函数名之前有一个\*

2.只能在函数内部使用yield，然函数停留当前位置

function\* fnc() {

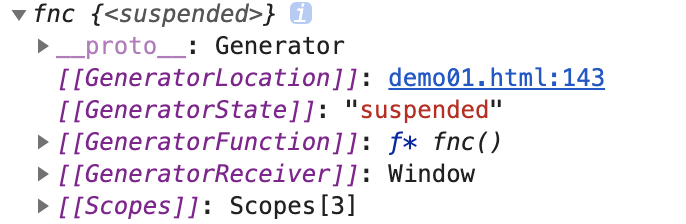
yield 2;

yield 3;

}

let o = fnc();//返回生成器对象，并不会调用函数内部，等next

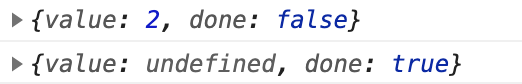
console.log(o);



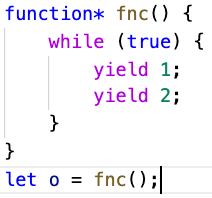
返回一个遍历器对象 可以调用next()

console.log(o.next());

console.log(o.next());



这样就能无限循环调用



总结：generator函数是分段执行，**yield语句是暂停执行，next是恢复执行**，一个next对应一个yield

function\* add() {

console.log('start');

let x = yield '2'; //next的参数传给x，x不是yield 2 的返回值，它是next()调用 恢复当前yield()执行传入的实参

console.log('one:' + x);

let y = yield '3';

console.log('two:' + y);

return x + y;

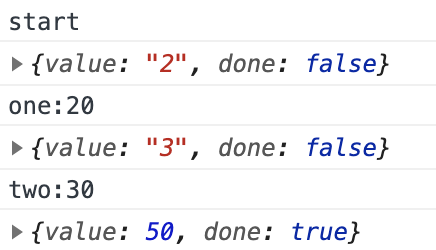
}

const fn = add();//生成一个生成器，函数内部不执行

console.log(fn.next());

console.log(fn.next(20));

console.log(fn.next(30));



**使用场景：为不具备Interator接口的对象提供遍历操作（数组有Interator接口）**

function\* objectEntries(obj) {

const propKeys = Object.keys(obj);//获取键的数组[name,age]

for (const propKey of propKeys) {

yield [propKey, obj[propKey]];

}

}

const obj = {

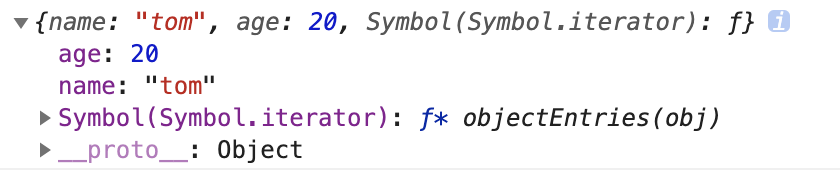
name: 'tom',

age: 20

}

obj[Symbol.iterator] = objectEntries;//给对象创建一个迭代器

console.log(obj);

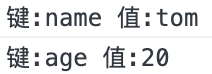


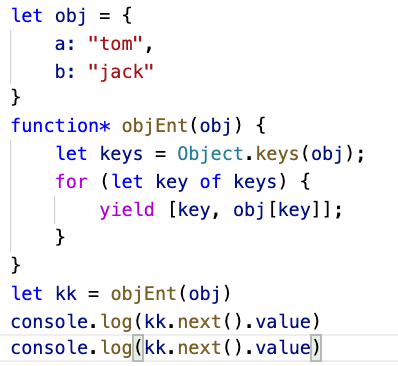
迭代器一旦生成就可以for of遍历

for (let [key, val] of objectEntries(obj)) {

console.log(`键:${key} 值:${val}`);

}





**十五.生成器generator的应用**

https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976

jQuery的Ajax请求

// 回调地狱

$.ajax({

url: 'https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976',

method: 'get',

success(res) {

console.log(res);

//继续发送请求

$.ajax({

url: '',

method: 'get',

success(res1) {

console.log(res1);

//继续发送请求...

}

});

}

});

解决回调地狱,ajax获取数据是需要时间的，防止未获取到数据就执行下一步操作

function\* main() {

let res = yield request('https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976');

console.log(res);

console.log('数据请求完成，接下去操作');

}

const ite = main();//生成了一个迭代器对象

ite.next();

function request(url) {

$.ajax({

url: url,

method: 'get',

success(res) {

ite.next(res);

}

});

}

加载图片时，1.先加载loading...页面，2.数据加载完成（异步操作），3.loading关闭

function loadUI() {

console.log('loading...界面打开');

}

function showData() {

setTimeout(() => {

console.log('数据加载完成');

}, 1000);

}

function hideUI() {

console.log('loading...页面关闭');

}

loadUI();

showData();

hideUI();



这里数据没加载完成就已经关闭loading页面了

function\* load() {

loadUI();

yield showData();

hideUI();

}

let itload = load();

itload.next();

function loadUI() {

console.log('loading...界面打开');

}

function showData() {

setTimeout(() => {

console.log('数据加载完成');

itload.next();

}, 1000);

}

function hideUI() {

console.log('loading...页面关闭');

}



**让异步代码同步化，部署ajax操作**

**同步就是：就是一件事一件事的执行。只有前一个任务执行完毕，才能执行后一个任务**

**十六.Promise对象（**在es6中有三种方法解决异步编程的问题**Generator生成器是一种，Promise是一种，async是一种）**

相当于是一个容器，保存着未来还未结束的事件（异步操作的结果）

各种异步操作都可以用同样的方法进行处理

特点：

1.对象的状态不受外界影响，**处理异步操作**三个状态 Pending（进行中） Resolved（已完成） Rejected（失败）

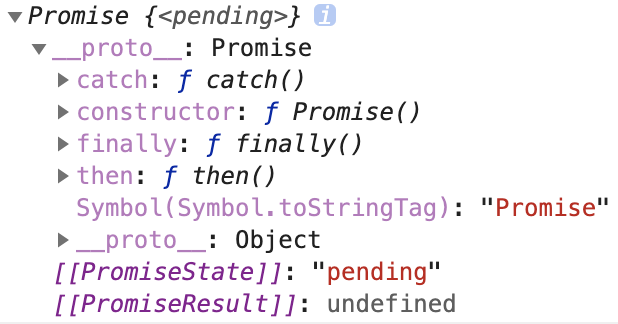
2.一旦状态改变，就不会再变，任何时候都可以获取到状态结果 要不是从Pending到Resolved，要不是从Pending到Rejected

let pro = new Promise(function(resolved,rejected){

//执行异步操作

});

console.log(pro)



then方法就是成功回调的方法

catch方法就是捕获错误

let pro = new Promise(function (resolved, rejected) {

//执行异步操作

//模拟后端获取的数据

let res = {

code: 200,

data: {

name: 'tom'

},

erro: '获取数据失败'

}

setTimeout(() => {

if (res.code == 200) {

resolved(res.data);//成功获取的数据返回出去

} else {

rejected(res.erro);//失败就把错误返回出去

}

}, 1000);

});

pro.then((val) => {//then方法可以接受两个回调函数，一个是获取数据成功，一个是获取数据失败

console.log(val)//获取数据成功，val就是成功数据

}, (err) => {

console.log(err);//获取数据失败，得到失败提示

});

封装一个ms毫秒以后获取数据的方法

function timeOut(ms) {

return new Promise((resolved, rejected) => {

setTimeout(() => {

resolved('获取成功');

}, ms);

});

}

timeOut(2000).then((val) => {

console.log(val);

});

**十七.Promise的应用**

**axios的getJSON方法**

let getJSON = function (url) {

return new Promise((resolved, rejected) => {

const xhr = new XMLHttpRequest();

xhr.open('GET', url);

xhr.onreadystatechange = handler;

xhr.responseType = 'json';

xhr.setRequestHeader('Accept', 'application/json');

xhr.send(null);

function handler() {

if (this.readyState == 4) {

if (this.status == 200) {

resolved(this.response)

} else {

rejected(new Error(this.statusText));

}

}

}

});

}

getJSON('https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976').then((data) => {

console.log(data);

}, (err) => {

console.log(err);

});

1.then方法 第一个参数必写是reslove的回调函数，第二个参数选写是reject的回调函数

then对象内部return 当前Promise对象

2.catch方法捕获失败

标准写法就是

getJSON('https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976').then((data) => {

console.log(data); //捕获成功

}).catch((err) => {

console.log(err); //捕获失败

});

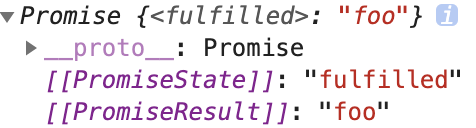
3.resolve/reject方法

能将现有任何对象直接转化成Promise对象

如将一个字符串转化成一个Promise对

let p = Promise.resolve('foo'); 等价于 let p = new Promise(resolve => resolve('foo'));

console.log(p);



状态是 fulfilled

值是foo

p.then((val) => {

console.log(val); // foo

});

4.all方法

同时执行多个异步

let promise1 = new Promise((resolved, rejected) => { })

let promise2 = new Promise((resolved, rejected) => { })

let promise3 = new Promise((resolved, rejected) => { })

let promise4 = Promise.all([promise1, promise2, promise3]);//传一个数组

promise4.then(() => {

//三个都成功才成功

}).catch(() => {

//如有一个失败，则失败

});

用于一些游戏类，素材比较多，等待图片，flash，静态文件都加载完，才能继续

5.race方法,用于请求超时时间设置，在超时后执行相应的操作

function requestImg(imgSrc) {

return new Promise((resolved, rejected) => {

const img = new Image();

img.src = imgSrc;

img.onload = function () {

resolved(img);

}

});

}

function timeOut() {

return new Promise((resolved, rejected) => {

setTimeout(() => {

rejected('图片请求超时');

}, 3000);

});

}

Promise.race([requestImg('http://img1.3png.com/b723851d7b05255ae975f1b4b565b6ba69f0.png'), timeOut()]).then((val) => {

//获取图片成功

console.log('图片获取成功');

document.body.appendChild(val);//这样就把图片放到页面上

}).catch((err) => {

//获取图片失败

console.log(err);

});

**race就是第一个成功了就不会往后走**

6.done/finally方法

都放在最后，调用不管请求成功还是失败都能调用

用于关闭服务器

**十八.async异步操作**

1.使异步操作更加方便

2.声明函数需要加关键字async

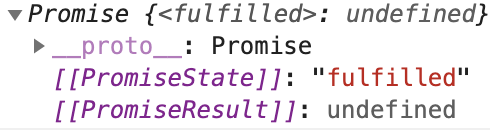
3.返回一个Promise对象

4.是Generator的语法糖

async function l() {

}

console.log(l());



**await命令后面跟Promise对象，不是就会转化成Promise对象，只能用在async函数中，只要有一个await后面的命令失败了，那就不往下执行**

async function l() {

let name = await 'tom,jack,bob';//将字符串转化成了Promise对象，获取name

let data = await name.split(','); //成功获取了name才会执行这一条

return data;

}

l().then((val) => {

console.log(val);//有许多异步await，但最后获取的数据一定是最终执行完return的数据

}).catch((err) => {

console.log(err);

});



1.解决了回调地狱

2.使得异步操作更加方便

**十九.class关键字解决了构造函数的陌生**

**es5:**

function Person(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.sayName = function () {

return this.name;

}

var person = Person('tom', 20);

**es6:**

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

sayName(){

return this.name;

}

}

当类实例化之后会自动调用constructor

可以通过Object.assign()方法一次性向类的原中添加多个方法

Object.assign(Person.prototype, {

sayName() {

return this.name;

},

sayAge() {

return this.age;

}

});

**二十.类的基础extends**

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

sayName(){

return this.name;

}

}

class Male extends Person {

constructor(name, age, sex) {

super(name, age);

this.sex = sex;

}

}

可以直接继承父类方法，也可以重写，也可以类的混合

**二十一.ES6的模块化实现**

功能：

1.export（抛出）和import（导入）组成

2.export用于规定模块的对外接口

3.import用于输入其他模块提供功能

4.一个模块就是一个独立文件

export const name = 'tom';//导出

export function sayName(){

return name;

}

或者export {name,sayName}//不建议

引入的script便签type必须是module

**<script type='module'>**

import {name,sayName} from './module/index.js'; //引入

console.log(name); //tom

</script>

默认抛出，整个js文件只能用一次

let obj ={

foo:'foo'

}

export default obj;

接收的时候

import obj, {name,sayName} from './module/index.js';