# 妙味课堂

## 0-ECMAScript 6简介

1. ECMAScript 6也称为ECMAScript2015，是2015年出版的ECMAScript新版本，是继ES5之后的一次主要改进。
2. ECMAScript 6特点：ES6增添了许多必要的特性，例如**模块**和**类**，**块级作用域**，**常量**与**变量**。
3. 支持程度：http://kangax.github.io/compat-table/es6/
4. 支持情况：node.js几乎完全支持ECMAScript6，标准浏览器支持部分语法。
5. 测试环境：如果在浏览器中使用es6部分语法可能无法使用，一部分是因为未实现，一部分需要特定的环境。

在chrome下使用es6为保证可以正常使用大部分语法，需要使用严格模式，即在js开始部分加上'use strict'。

在firefox下使用es6为保证可以正常使用大部分语法，需要指定测试版本，即在script标签的type属性中加上'application/ javascript;version=1.7'属性值。

## 1-变量声明(块级作用域)

1. Let基本特性

(1) **不允许重复声明**

let a=1;

let a=2; //报错:Identifier 'a' has already been declared

console.log(a);

(2)**没有预解析**(暂存死区)

console.log(b);// b is not defined

let b=2;

(3)**块级作用域**

{

let c=1;

let d=1;

console.log(c);

}

console.log(d);

注释:①块级作用域是相对于变量声明所在的**大括号{}**或者for循环所在**大括号{}**而言。②使用let声明的变量可以替代闭包和索引值(自定义属性)。

1. 定时器—let块级作用域替代闭包

for(var i=0;i<10;i++){

setTimeout(function(){

console.log(i);//打印十次10

})

}

注释：setInterval与setTimeout当不设置时间间隔时，以浏览器默认最小时间间隔（小于代码执行顺序）为准。

for(let j=0;j<10;j++){

setTimeout(function(){

console.log(j);//依次打印0-9

})

}

注释: setInterval与setTimeout的第三个参数为传入第一个函数参数的参数，第三个参数为可选参数。

function f(a){

alert(a);

}

setInterval(f, 1000, 2); //每一秒弹出一次2

// setTimeout(f, 1000, 2)//一秒过后弹出一次2

**注释:一对{}包括的区域为let 作用区域。**

for(var k=0;k<10;k++){

(function(k){

setTimeout(function(){

console.log(k);//依次打印0-9

})

})(k)

}

1. 选项卡-let块级作用域替代索引值

<script>

var tabs=document.getElementsByTagName('input');

var divs=document.getElementsByTagName('div');

for(var i=0;i<tabs.length;i++){

**tabs[i].index=i;**

tabs[i].onclick=function(){

for(var j=0;j<tabs.length;j++){

divs[j].className='';

tabs[j].className='';

}

this.className='active';

divs**[this.index].**className='show';

}

}

</script>

<script>

var tabs=document.getElementsByTagName('input');

var divs=document.getElementsByTagName('div');

for(**let** i=0;i<tabs.length;i++){

tabs[i].onclick=function(){

for(var j=0;j<tabs.length;j++){

divs[j].className='';

tabs[j].className='';

}

this.className='active';

**divs[i]**.className='show';

}

}

</script>

## 2-变量声明(常量)

1. 在定义之后值是固定不变的，即为常量。常量的值是不能修改的，但如果常量保存的是一个对象，那么对象的属性是可以被修改的。

<script>

const a=1;

a=2;

console.log(a);//弹出**1或者报错**

const b={name:'leo'};

b.name='jake';

console.log(b.name); //打印jake

</script>

注释: 常量不允许重新被赋值，但允许给常量添加属性或者修改属性值。

## 3-变量的解构赋值

1. 什么是解构赋值

ES6允许按照一定的模式，**从数组和对象中提取值，对变量进行赋值**，这被称为解构(Destructuring)。

1. 数组的解构赋值语法

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

**结构相同，按照顺序，一一赋值，允许嵌套**

<script>

var arr=[1,[2,3]]

var [a,[b,c]]=arr;

console.log(a,b,c)//打印1 2 3

</script>

1. 对象的解构赋值语法

var {foo,bar}={foo:”aaa”,bar:”bbb”};

**名字相同，顺序可变，依次赋值，允许嵌套**

<script>

var obj={

foo:function(){},

o:{},

arr:[],

str:'abc'

}

var {foo,str,arr,o}=obj;

console.log(foo,str,arr,o);//打印函数 字符串 数组 对象

</script>

1. 解构赋值的作用

函数只能返回一个值，如果要返回多个值，只能将他们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

1. 交换x、y的值

<script>

var x=1;

var y=2;

方法1:

// var n;

// n=x;

// x=y;

// y=n;

方法2:

**[x,y]=[2,1]**

console.log(x,y)

</script>

## 4-字符串扩展

1. ES6加强了对Unicode的支持，并且扩展了字符串对象。
2. 字符串方法
3. 字符串扩展方法:(0)获取长度 (1)字符转码点 (2)码点转字符 (3)复制字符串 (4)模板字符串 (5)判断包含关系 (6)判断起始字符 (7)判断结束字符
4. 实例

<script>

var str='miaov';

console.log(**str.length**);//5

var str2='吉';//注释：土加上口

console.log(str2.length);//2

console.log('倪梦威'**.codePointAt(0)**);//返回码点

console.log(String.**fromCodePoint(20522)**);//根据码点返回字符

console.log( str2.**repeat(3)**);//复制字符串

**//模板字符串**

var name='leo';

var age='39';

var str3=`你的名字是：**${name}**你的年龄是：**${age}**`;

console.log(str3);

//注释:①反引号;②${}包裹。

//字符串包含关系

var str4='miaov';

console.log(str4.**includes**('o'));

console.log(str4.**startsWith**('mi'))

console.log(str4.**endsWith**('vo'))

</script>

## 5-数值扩展

1. 数值扩展方法: (1)二进制转十进制 (2)八进制转十进制 (3)删除小数 (4)判断正、负、正零、负零 (5)平方和的平方根
2. 实例

<script>

var num1=**0b**11;//2进制

var num2=**0o**11;//8进制

console.log(num1,num2);//3 9

注释: console.log()方法中使用**逗号表示空格**。

console.log(Math.**trunc**(-3,67)) //去除小数点

console.log(Math.**sign**(-0))//判断正数、负数、正0还是负0

console.log(Math.**hypot**(3,4))//平方和的平方根

</script>

注释: truncate截短的,被删节的;sign符号,记号;

## 6-数组扩展

1. 数组扩展方法:
2. Array.from(类数组/字符串)—类数组/字符串转化为数组；
3. Array.of(参数)—参数转化为数组
4. arr.find(function(value,index){})—寻找第一个符合条件的**元素**
5. arr.findIndex(function(value,index){})—寻找第一个符合条件的**元素的下标**
6. arr.fill(填充内容，[起始下标]，[结束顺序])

注释: arr.fill()方法的返回值===新数组，原数组发生改变。

2. 实例

<script>

var divs=document.getElementsByTagName('div');

console.log(divs);//类数组

var eles1=**[].slice.call(divs);**//类数组转化为数组

console.log(eles1);

var eles2=**Array.from(divs)**;//类数组转化为数组的新方法

console.log(eles2);

var str='moaiv';

console.log(**Array.from(str)**);//字符串转化为数组

console.log(Array.of(1,[1,2],{key:'3'}));

//参数转化为数组[1,[1,2],{key:'3'}]

console.log(Array.of(3));//含义为**数组为[3]**

console.log(new Array(3)) ;//含义为**数组的长度为3**

console.log(new Array(1,2,3))//将参数转化为数组[1,2,3]

注释:new Array()与Array.of()的区别在于当参数只有一个时，前者表示数组长度，后者表示只有一个元素的数组的值。

//arr.find();遍历数组的每一个元素，返回第一个符合条件的数组元素，如果都不符合条件，则返回**undefined**。

var arr=[1,2,3,4,5];

var n=arr.find(function(value,index){

**console.log(arguments);**

//arguments代表**实参的集合**

return value>3;

})

console.log(n);

//findIndex();遍历数组的每一个元素，返回第一个符合条件的数组元素的下标，如果都不符合条件，则返回**-1**。

var arr=[1,2,3,4,5];

var m=arr.findIndex(function(value,index){

return value>2;

}

)

console.log(m);

//arr.fill();填充数据

//参数：填充的内容，起始下标，结束顺序

var arr=[1,2,3,4,5];

arr.fill(6,2,4);

console.log(arr);

//arr.fill(6)含义为数组元素全部替换为6，fill(6,2)的含义为从下标为2的数组元素开始全部替换为6。

//fill方法的适用场景

arr.length=10;

console.log(arr.fill(10))

</script>

1. for of遍历接口

for of可以遍历所有遍历接口（数组、字符串）不可遍历对象。

1. 遍历value值
2. 遍历key值
3. 遍历key值和value值
4. 数组推导

注释: P11使用Object.keys(obj)遍历对象的key值。P16部署遍历接口

(ymbol.iterator)使for of可以遍历对象。

1. 实例

<script>

var arr=[1,2,3,4,5];

for(**var value of arr**){

console.log(value);//依次打印1,2,3,4,5

};

var str='miaov';

for(**var value of str**){

console.log(value);//依次打印m,i,a,o,v

};

var obj={

a:1,

b:2,

c:3

};

for(var value of obj){

console.log(value);//报错

};

for(**var key of arr.keys()**){

console.log(key);//依次打印key值 0-4

}

// 注释:arr.keys()表示数组key值的接口

for(**var [key,value] of arr.entries()**){

console.log(key,value);//依次打印key value值

}

1. 数组推导

现有数组生成新数组

var arr2=**[for(value of arr)value\*2]**;

//注意：value不可声明

console.log(arr2);

var arr3=**[for(value of arr)if(value>2)if(value<5) value]**;

console.log(arr3)

</script>

## 7-对象扩展

1. 属性和方法的简洁表示法

ES6允许直接写入变量和函数，作为**对象的属性和方法**。

<script>

function fn(x,y){

x++;

y++;

// return {x:x,y:y} 对象属性的完整表示法

return {x,y} //对象属性的简洁表示法

}

console.log(fn(1,2))

</script>

<script>

var obj={

name:'momo',

// showName:function(){

// return this.name;

// }对象方法的完整表示法

**showName(){**

**return this.name;**

**}**//对象方法的简洁表示法

};

console.log(obj.showName())

</script>

1. 属性名表达式

ES6允许字面量定义对象时，用表达式作为对象的属性名，即把表达式放在方括号内。

<script>

var sex='男';

var person={

name:'momo',

**[sex]**:false,

['getname'](){

return this.name;

}

}

console.log(person.getname())

console.log(person['男'])

</script>

1. **(1)Object.is()**

用于对比参数是否严格相等，相对于===，可以正确比较-0和0还有NaN。

<script>

console.log(0===-0);//true

**console.log(Object.is(0,-0))**//false

**console.log(Object.is(NaN,NaN))**//true

</script>

1. **Object.assign(target,[source1...sourceN])**

将source对象的可枚举属性赋值到target对象上。

<script>

var obj1={};

var obj2={

name:'zz'

}

var obj3={

name:'cc',

age:20

}

**Object.assign(obj1,obj2,obj3);**

console.log(obj1);

</script>

注意: 如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

1. Object.getPrototypeOf(object)

用来获取一个对象的prototype对象

<script>

var Cat=function(name){

this.name=name;

};

Cat.prototype.showName=function(){

return this.name;

};

var c1=new Cat('momo');

console.log(**Object.getPrototypeOf(c1)**);

</script>

1. **Object.setPrototypeOf(object,prototype)**

用来设置一个对象的prototype对象

<script>

var Cat=function(name){

this.name=name;

};

Cat.prototype.showName=function(){

return this.name;

};

var Person=function(){};

Person.prototype.dadoudou=function(){};

var c1=new Cat('momo');

**Object.setPrototypeOf(c1,Person.prototype);**

console.log(Object.getPrototypeOf(c1));

</script>

1. \_\_proto\_\_

用来读取或设置当前对象的prototype对象

例如：var obj={

\_\_proto\_\_:super

}

**(6)Proxy（疑问）**

Proxy用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”(meta programming)，即对编程语言进行编程。

参数：(1)要代理的目标对象 (2)设置对象

<script>

var obj={

a:1,

b:2

};

var p1=new Proxy(obj,{

get(obj,attr){//当属性被访问时触发

console.log(obj,attr);

return 10;

},

set(obj,attr,value){//当属性值修改时触发

console.log(obj,attr,value);

}

});

console.log(p1.a);

p1.b=5;

</script>

1. **Object.observe(obj,observe)**

用于检测对象的变化，一旦发生变化就会调用回调函数。

参数：(1) 需要监控的对象 (2) 回调函数(回调函数接收一个数组参数)

触发事件：

add:添加属性

update：属性值的变化

delete：删除属性

SetPrototype：设置原型

reconfigue：属性的attributes对象发生变化

1. 实例

<script>

var obj={a:1,b:2};

**Object.observe(obj,function(a){**

console.log(a);//打印出一个对象

})

**obj.a=3;**//对象属性发生变化

</script>

注释:添加属性、属性值变化、删除属性、设置原型、attribute对象变化时都会触发事件函数。

<script>

var obj={a:1,b:2};

**Object.observe(obj,function(a){**

console.log(a);

})

**console.log(obj.a)**//访问对象的属性，并未触发事件

</script>

注释: 访问对象的属性并未触发事件函数。

1. Object.unobserve(obj,observe) 取消监测

<script>

var obj={a:1,b:2};

function fn(a){

console.log(a);//不再打印

};

Object.observe(obj,fn);

Object.unobserve(obj,fn);

obj.a=3;

</script>

注释: 添加属性在取消监测之后，不再触发事件函数。

1. **Object.keys(obj)** 返回所有类型的键名

<script>

var obj={a:1,b:2};

console.log(Object.keys(obj))//打印['a','b']

</script>

注释: P7使用for of遍历数组和字符串接口。

## 8-函数扩展

1. 函数参数的默认值

<script>

function fn(a,b=2){

return{a,b};

};

console.log(fn(1));

</script>

注意: 定义默认值的参数必须是尾参数，因为定义默认值之后该参数可以忽略。

1. 参数省略

用于获取函数的多余参数

语法: function fn(a,b,...变量名){}

<script>

function fn(a,b,c,...rest){

console.log(rest);//打印省略的形参对应的实参4,5,6,7

console.log(arguments);//打印所有实参1,2,3,4,5,6,7

}

fn(1,2,3,4,5,6,7)

</script>

注意: rest后边不能再有其他参数，否则会报错。

1. 扩展运算符...

扩展运算符是三个点。它好比是rest参数的逆运算，将一个数组转化为用逗号分隔的参数序列。该运算符主要用于函数的调用。

<script>

var arr=[1,2,3,4,5,6,7];

console.log(**Math.max.apply(null,arr)**);//打印7

console.log(M**ath.max(...arr)**);//打印7

var str='miaov';

var arr=**[...str]**;

console.log(arr); //打印['m','i','a','o','v']

</script>

注释: 扩展运算符可将set和map数据类型转化为数组。

1. **箭头函数**——用来作为回调函数使用

语法: 函数名=(形参)=>返回值

<script>

var f1=(a)=>a+1;

console.log(f1(1));

var f2=(a,b)=>a+b;

console.log(f2(1,2))

console.log([2,3,13].sort((a,b)=>a-b))

</script>

注意: ①函数体内的this指向函数定义时所在的对象，而不是使用时所在的对象。②不可以当做构造函数，不可以使用new命令，否则报错。③该函数体内不存在arguments。

## 9-数据结构(set)

1. Set([array])

ES6提供了新的数据结构Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。

Set是一个构造函数，可以**传入一个数组初始化默认值**。

<script>

var set=**new Set([1,2,3,3,4])**;

console.log(set);//打印出1,2,3,4

</script>

1. **set.size**

set的实例的成员个数(数组的长度)

<script>

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

console.log(set);//打印1,2,3,4

console.log(**set.size**);//打印4

</script>

1. **set.add(value)**

为set的实例添加值

<script>

var set=new Set([2,3,5,3,5]);

**set.add(7);**

console.log(set);//打印2,3,5,7

</script>

1. **set.delete(value)**

删除set实例的值

<script>

var set=new Set([2,3,5,3,5]);

set.add(7);

**set.delete(7);**

console.log(set);//打印2,3,5

</script>

注释: 添加set.add()和删除set.delete()方法的参数为值，而不是下标。

1. **set.has(value)**

判断传入的参数是否为set的成员

<script>

var set=new Set([2,3,5,3,5]);

console.log(set.has(6));//false

</script>

1. **set.clear()**

清除set中所有成员

<script>

var set=new Set([2,3,5,3,5]);

set.clear();

console.log(set);//{}

</script>

1. set数据类型转化为数组

<script>

var set=new Set([2,3,5,3,5]);

console.log(**[...set]**);//[2,3,5]

</script>

注释:通过扩展运算符将set转化为数组。

1. WeakSet(list)

WeakSet结构与Set类似，也是**不重复**的值的集合。

WeakSet是一个构造函数，可以传入一个**类数组**初始化默认值。

WeakSet的成员只能是**对象**，而不能是其他类型的值。

WeakSet中的对象都是弱引用，即垃圾回收机制不考虑

WeakSet对该对象的引用

WeakSet是不可遍历的

1. WeakSet的方法

ws.add(value) 为Weakset的实例添加值

ws.delete(value) 删除WeakSet实例的值

ws.has(value) 判断传入的参数是否为set的成员

## 10-数据结构(map)

1. ES6提供了map数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当做键。

Map()也可以接受一个**数组作为参数**。该数组的成员是一个个表示键值对的数组。

例如: [[key1,value1],[key2,value2]]

<script>

var obj={'say':'leo'};//obj可以作为map对象的键。

var map=**new Map([['age',20],['age',25],[obj,23]])**;

console.log(map);

</script>

注释:map数据结构的键名不可以重复，否则后者覆盖前者。

1. **map.size()**

返回成员总数

<script>

var obj={'say':'leo'};

var map=new Map([['age',20],['age',25],[obj,23]]);

console.log(**map.size**);//打印2

</script>

1. **map.get(key)**

获取value值

<script>

var map=new Map([['age',25],['name','jake']]);

console.log(**map.get('age')**);//打印25

</script>

1. **map.set(key,value)**

添加键值对

<script>

var map=new Map([['age',25],['name','jake']]);

**map.set('sex','男')**;

console.log(map.size);//打印3

</script>

1. **map.has(key)**

查找key值是否存在

<script>

var map=new Map([['age',25],['name','jake']]);

console.log(**map.has('age')**);//打印true

</script>

1. **map.delete(key)**

删除对应键值对

<script>

var map=new Map([['age',25],['name','jake']]);

**map.delete('age');**

console.log(map.has('age'));//打印false

</script>

1. **map.clear()**

清除所有成员

<script>

var map=new Map([['age',25],['name','jake']]);

**map.clear();**

console.log(map.size);//打印0

</script>

1. **map.keys()**

遍历map的key值

<script>

var map=new Map([['age',25],['name','jake']]);

for(**var key of map.keys()**){

console.log(key);//依次打印age，name

}

</script>

1. **map.values()**

遍历map的value值

<script>

var map=new Map([['age',25],['name','jake']]);

for(**var value of map.values()**){

console.log(value);//依次打印25，jake

}

</script>

1. **map.entries()**

遍历map的key值和value值

<script>

var map=new Map([['age',25],['name','jake']]);

for(**var [key,value] of map.entries()**){

console.log(key,value);//依次打印age 25,name jake

}

</script>

1. **map.forEach(callback)**

遍历(与数组的forEach一样)

第三个参数代表callback内的this指向(当前map对象)

<script>

var map=new Map([['age',25],['name','jake']]);

map.forEach(function(value,key,map){

console.log(arguments);

});

</script>

1. Map数据类型转化为数组

<script>

var map=new Map([['age',25],['name','jake']]);

console.log(**[...map]**);//打印数组

</script>

注释:通过扩展运算符将map转化为数组。

1. WeakMap()

WeakMap结构与map结构基本类似，唯一的区别是它只接受对象做为键名(null)除外，不接受原始类型的值作为键名，而且键名所指向的对象，不计入垃圾回收机制。

wm.get(key); 通过key获取value

wm.set(key,value) 为weakmap的实例添加新的键值对

wm.has(key) 查找key值是否存在

wm.delete(key) 删除对应键值对

## 11-遍历接口

1. Symbol

ES6引入了一种新的原始数据类型Symbol，表示独一无二的ID。它通过Symbol函数生成。

<script>

var s1=Symbol('test');

var s2=Symbol('test');

console.log(s1==s2); //false

console.log(typeof s1); //symbol

</script>

1. Lterator

遍历接口部署：一个对象如果要有lterator接口，必须部署一个@@iterator方法，该方法部署在一个键名为Symbol.iterator的属性上，对应的键值是一个函数，该函数放回一个**遍历器对象**。

1. Symbol.iterator

一个预定义好的、类型为Symbol的特殊值

1. 接口部署语法

object[Symbol.iterator]=function(){

**return {next(){return {value:value,done:boolean}}}**

}

1. 实例

<script>

var obj={a:1,b:2,c:3}

Object.prototype[**Symbol.iterator**]=function(){

var keys=Object.keys(this);

var index=0;

var self=this;

return {

next(){

**if(index<keys.length){**

return {

**value:self[keys[index++]],done:false}**

}else{

return {

**value:undefined,done:true**

};

}

}

}

}

for(var value of obj){

console.log(value);//打印1,2,3

}

</script>

## 12-Generator函数

1. 什么是Generator

所谓Generator，可以把它理解为一个函数的内部状态的遍历器，每调用一次，函数的内部状态发生一次改变。ES6引入Generator函数，作用就是可以完全控制函数的内部状态的变化，依次遍历这些状态。

在形式上，Generator是一个普通函数，但是有两个特征。一是function命令与函数名之间有一个星号；二是函数体内部使用yield语句，定义遍历器的每个成员，即不同内部状态（yield语句在英语里的意思就是“产出”）。

<script>

function\* fn(){

yield 1;

yield 2;

}

var f=fn();

console.log(f.next());

//返回一个遍历接口{value: 1, done: false}

console.log(f.next());//{value: 1, done: false}

console.log(f.next());//{value: undefined, done: true}

</script>

## 13-Promise对象

1. 什么是promise

ES6的Promise对象是一个构造函数，用来生成Promise实例。下面是Promise对象的基本用法。

所谓Promise对象，就是代表了**未来某个将要发生的事件**（通常是一个异步操作）。它的好处在于，有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。

注释: **Promise对象将一个或多个函数执行作为另一个函数执行的触发条件**。

1. 语法

Promise

**new Promise(function(resolve,reject){})**

**返回Promise实例对象**。

then

**promise.then(resolve,reject)**

返回promise。

注释: ①new Promise()对象中参数为一个**自执行函数**。②该自执行函数有两个函数参数。③resolve与resolve对应，reject与reject对应，前者函数执行是后者同名函数执行的触发条件。

1. catch 捕获前一个回调函数抛出的错误
2. 实例

<script>

var p1=new Promise(function(**resolve,reject**){

setTimeout(function(){

**resolve();**

},400)

})

p1.then(function(){

console.log('成功'); //打印“成功” console.log(b);

},function(){

console.log('失败');

}).catch(function(e){

console.log(e);

//ReferenceError: b is not defined(…)

})

</script>

1. Promise.all

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

参数要求拥有iterator接口，并且每一项都是promise实例

var p = Promise.all([p1,p2,p3])

p的状态由p1、p2、p3决定，分成两种情况。

(1)只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

(2)只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

<script>

var p1 = new Promise(function(resolve,reject){

setTimeout(function(){

resolve();

console.log('p1')

}, 20)

});

var p2 = new Promise(function(resolve,reject){

setTimeout(function(){

reject();

console.log('p2')

}, 400)

});

var p3 = new Promise(function(resolve,reject){

setTimeout(function(){

resolve();

console.log('p3')

}, 1000)

});

var p4 = Promise.all([p1,p2,p3]);

p4.then(function(){

console.log('成功');

}, function(){

console.log('失败');

}) //依次打印p1、p2、失败、p3

</script>

1. Promise.race

与Promise.all方法类似将多个promise包装成一个新的promise实例

但是其中有一项的状态发生改变新的实例的状态就会随着改变。

var p = Promise.race([p1,p2,p3])

p的状态由p1、p2、p3决定，分成两种情况。

(1)只要p1、p2、p3中一个的状态变成fulfilled，p的状态就变成fulfilled。

(2)只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

1. async函数**？？**

只要函数名之前加上async关键字，就表明该函数内部有异步操作。该异步操作应该返回一个Promise对象，前面用await关键字注明。当函数执行的时候，一旦遇到await就会先返回，等到触发的异步操作完成，再接着执行函数体内后面的语句。

例如：

async function fn(){

let data = await ajax ();

return data;

}

## 14-class类

1. 构造函数模拟类

<script>

function Cat(name){

this.name=name;

}

Cat.prototype.getName=function(){return this.name;}

var c1 = new Cat('momo');

console.log(c1.getName());//momo

</script>

1. Class类的书写方式

<script>

**class Cat{**

**constructor(name){**

this.name=name;

**}**

**getName(){**

return this.name;

**}**

}

var c1 = new Cat('momo');

console.log(c1.getName());//momo

</script>

注释: constructor表示构造函数，getName表示方法名。

1. extends继承的书写方式

<script>

class Cat{

constructor(name){this.name=name;}

getName(){return this.name;}

};

**class Momo extends Cat**{

constructor(name,age){**super(name);** this.age=age;}

showAge(){return this.age;}

}

var b1=new Momo('猫',20);

console.log(b1.getName(),b1.showAge());//猫 20

</script>

注释: 在constructor方法内，super指代父类的constructor方法；在其他方法内，super指代父类的同名方法。

## 15-module模块

1. 什么是module

es6所提供的模块化

1. export命令

export命令用于用户自定义模块，规定对外接口

语法: export var name = 'leo';

export {name1,name2}

1. import命令

import命令用于输入其他模块提供的接口

语法: import {nam1,name2} from '文件路径';

输入的名称必须与输出的相同

1. as关键字

为输入的变量换一个新的名字

import { name as n } from '路径';

1. 整体输入模块

**import \* as 变量名 from '路径';**

module 变量名 from '路径';

输入的模块定义在变量名上

1. export default

输出匿名函数

语法： export default function(){};

在输入的时候可以使用任何名字指向该匿名函数

例如： import name from '路径'；

1. 模块的继承

export \* from '模块路径';

输出模块中所有方法和属性

export { a as b} from '模块路径';

将模块中的a变量转为b输出

# ECMAScript 6 入门

作者：阮一峰

<http://es6.ruanyifeng.com/>

## 0-前言

1. 本书覆盖 ES6/ES7 与 ES5 的所有不同之处，适合已经掌握 ES5 的读者，用来了解这门语言的最新发展；也可当作参考手册，查寻新增的语法点。

## 1-ECMAScript 6简介

ECMAScript 6.0（以下简称ES6）是JavaScript语言的下一代标准，已经在2015年6月正式发布了。它的目标，是使得JavaScript语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

### 1.1 ECMAScript和JavaScript的关系

1996年11月，JavaScript的创造者Netscape公司，决定将JavaScript提交给国际标准化组织ECMA，希望这种语言能够成为国际标准。次年，ECMA发布262号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为ECMAScript，这个版本就是1.0版。

因此，ECMAScript和JavaScript的关系是，前者是后者的规格，后者是前者的一种实现（另外的ECMAScript方言还有Jscript和ActionScript）。

注释：JScript是由微软公司开发的活动脚本语言，是微软对ECMAScript规范的实现。ActionScript（简称AS）是由Macromedia（现已被Adobe收购）为其Flash产品开发的脚本语言。

### 1.2 ES6与ECMAScript 2015的关系

2011年，ECMAScript 5.1版发布后，就开始制定6.0版了。因此，”ES6”这个词的原意，就是指JavaScript语言的下一个版本。

ES6既是一个历史名词，也是一个泛指，含义是5.1版以后的JavaScript的下一代标准，涵盖了ES2015、ES2016、ES2017等等，而ES2015则是正式名称，特指该年发布的正式版本的语言标准。

注释:ES6为ECMAScript 5.1版本之后的下一**代**版本；ES2015为2015年6月发布的第一**个**ES6版本。

### 1.3 语法提案的批准流程

任何人都可以向TC39标准委员会提案。一种新的语法从提案到变成正式标准，需要经历五个阶段。每个阶段的变动都需要由TC39委员会批准。

### 1.4 ECMAScript的历史

ES6从开始制定到最后发布，整整用了15年。

|  |  |  |  |
| --- | --- | --- | --- |
| ECMAScript 1.0 | 1997年 | 发布 | 初期版本 |
| ECMAScript 2.0 | 1998年6月 | 发布 |
| ECMAScript 3.0 | 1999年12月 | 发布 | **通行标准** |
| ECMAScript 4.0 | 2000年 | 未发布 | 争论激烈 |
| ECMAScript 4.0 | 2007年10月 | 未发布 |
| ECMAScript 4.0 | 2008年7月 | 未发布 |
| ECMAScript 3.1/5 | 2008年7月 | 发布 | Harmony |
| ECMAScript 5.0 | 2009年12月 | 发布 | 新版本 |
| ECMAScript 6 | 2015年6月 | 发布 |

### 1.5 部署进度

随着时间的推移，各大浏览器的最新版本对ES6的支持度已经越来越高了，ES6的大部分特性都实现了。

Node.js是JavaScript语言的服务器运行环境，对ES6的支持度比浏览器更高。通过Node，可以体验更多ES6的特性。

### 1.6 Babel转码器

Babel是一个广泛使用的ES6转码器，可以将ES6代码转为ES5代码，从而在现有环境执行。

Babel提供一个[REPL在线编译器](https://babeljs.io/repl/)，可以在线将ES6代码转为ES5代码。转换后的代码，可以直接作为ES5代码插入网页运行。

<https://babeljs.io/repl/#?babili=false&evaluate=true&lineWrap=false&presets=es2015%2Creact%2Cstage-2&code=>

### 1.7 Traceur转码器

Google公司的Traceur转码器，也可以将ES6代码转为ES5代码。

## 2-let和const命令

### 2.1 let命令

1. 基本用法

(1)ES6新增了let命令，用来声明变量。它的用法类似于var，但是所声明的变量，只在let命令所在的代码块内有效。

{  
 var a = 1;  
 let b = 2;  
 }  
 console.log(a); //1  
 console.log(b); //ReferenceError: b is not defined.

注释：①在ES6之前，JavaScript并没有块级作用域的概念，单独使用{}没有意义。{}代码块只在if条件判断语句、for循环语句以及function函数语句中使用。②ES6中的let引入了块级作用域的概念，使{}代码块有了单独使用的意义。

(2)for循环的计数器，就很合适使用let命令。

示例1:

for(let i = 0; i < 10; i++){}  
 console.log(i); //ReferenceError: i is not defined

上面代码中，计数器i只在for循环体内有效，在循环体外引用就会报错。

var a = [];  
 for(var i = 0; i < 10; i++) {  
 a[i] = function () {  
 console.log(i);  
 };  
 }  
 a[6](); //10

上面代码中，变量i是var声明的，在全局范围内都有效。所以每一次循环，新的i值都会覆盖旧值，导致最后输出的是最后一轮的i的值。

如果使用let，声明的变量仅在块级作用域内有效，最后输出的是6。

var a = [];  
 for(let i = 0; i < 10; i++) {  
 a[i] = function () {  
 console.log(i);  
 };  
 }  
 a[6](); //6

上面代码中，变量i是let声明的，当前的i只在本轮循环有效，所以每一次循环的i其实都是一个新的变量，所以最后输出的是6。

注释:在for循环的块级作用域{}中，每一轮循环都会生成新的块级作用域。不同循环中使用let声明的同一变量名位于不同的块级作用域中，并不会发生相互覆盖。

1. 不存在变量提升

let不像var那样会发生“变量提升”现象。所以，变量一定要在声明后使用，否则报错。

console.log(foo); //undefined  
 console.log(bar); //ReferenceError: bar is not defined  
 var foo = 2;  
 let bar = 2;

注释:与var声明的变量不同，let声明的变量不存在预解析。

1. 暂时性死区

(1)只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

var tmp = 123;  
 {  
 tem = 'abc'; //tem is not defined  
 let tem;  
 }

ES6明确规定，如果区块中存在let和const命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。凡是在声明之前就使用这些变量，就会报错。

总之，在代码块内，使用let命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”（temporal dead zone，简称TDZ）。

if(true) {  
 //temporal dead zone(TDZ开始)  
 tem = 'abc'; //ReferenceError: tem is not defined  
 console.log(tem); //ReferenceError: tem is not defined  
 let tem; //TDZ结束  
 console.log(tem); //undefined  
 tem ='def';  
 console.log(tem); //def  
}

上面代码中，在let命令声明变量tmp之前，都属于变量tmp的“死区”。

注意:①在作用域链中，当前作用域声明的变量优先级高于外层作用域声明的变量。在块级作用域中，let和const声明的变量拥有唯一使用特权，此特权不仅仅在let和const声明变量之后有效，而是在整个块级作用域中都有效。②若某一个变量在块级作用域中使用let或const声明，则从该块级作用域的起始位置到该变量的声明位置之间，该变量都是不可用的。该区域属于该变量的暂存性死区，该区域中**任何形式的使用**该变量都会报错。

1. “暂时性死区”也意味着typeof不再是一个百分之百安全的操作。

{  
 typeof x; //ReferenceError: x is not defined  
 let x;  
 }

上面代码中，变量x使用let命令声明，所以在声明之前，都属于x的“死区”，只要用到该变量就会报错。

作为比较，如果一个变量根本没有被声明，使用typeof反而不会报错。

{  
 console.log(typeof y); //undefined  
 }

在没有let之前，typeof运算符是百分之百安全的，永远不会报错。现在这一点不成立了。这样的设计是为了让大家养成良好的编程习惯，变量一定要在声明之后使用，否则就报错。

1. 有些“死区”比较隐蔽，不太容易发现。

function bar(x = y, y = 2) {  
 return [x, y];  
 }  
 bar(); //ReferenceError: y is not defined

上面代码中，调用bar函数之所以报错（某些实现可能不报错），是因为参数x默认值等于另一个参数y，而此时y还没有声明，属于“死区”。如果y的默认值是x，就不会报错，因为此时x已经声明了。

function bar(y = 2, x = y) {  
 return [x, y];  
 }  
 bar(); //[2,2]

注释:①若一个变量未声明，则该变量的使用区域对该变量来说为死区。若一个变量在块级作用域中使用let和const声明，该块级作用域的起始位置到该变量的声明位置之间的区域为该变量的死区。

1. ES6规定暂时性死区和let、const语句不出现变量提升，主要是为了减少运行时错误，防止在变量声明前就使用这个变量，从而导致意料之外的行为。

总之，暂时性死区的本质就是，只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

1. 不允许重复声明

(1)let不允许在相同的**块级**作用域内，重复声明同一个变量。

function demo() {  
 var a = 1;  
 let a = 10; //报错  
 }

function demo(){  
 let a = 10;  
 let a = 1; //报错  
 }

注释:①声明变量有三种方式:var let const。②在同一**块级**作用域中重复声明同一变量不报错只有一种情况，即每次都使用var声明该变量。③以下五种情况的重复声明同一变量都会报错，即var与let、var与const、let与let、let与const、const与const。

注意:作用域链有两个，一个针对函数嵌套，一个针对块级作用域的嵌套。

1. 不能在函数内部重新声明参数。

function func(arg) {  
 let arg; //报错  
 }

function func(arg) {  
 {  
 let arg; //不报错  
 }  
 }

注释:①函数参数等同于在函数体的起始位置var的变量。②块级作用域可以相互嵌套，存在嵌套关系的块级作用域之间可以声明同一变量。

**总结:let声明变量——块级作用域内有效、变量不提升、存在暂时性死区、不可重复声明。**

**const声明变量——块级作用域内有效、变量不提升、存在暂时性死区、不可重复声明、值不能改变、声明立即初始化。**

### 2.2 块级作用域

1. 为什么需要块级作用域？
2. 避免内层变量可能会覆盖外层变量

var tmp = new Date();  
 function f() {  
 console.log(tmp);  
 **if(false) {** var tmp = 'Hello world';  
 }  
 }  
 f(); //undefined

上面代码中，函数f执行后，输出结果为undefined，原因在于变量提升，导致内层的tmp变量覆盖了外层的tmp变量。

注释:变量提升以函数作用域为基本单位，而不以块级作用域为基本单位。f函数内部所有使用var声明的变量(包括tmp)在函数执行前都会被预解析，即使tem变量在块级作用域中。

1. 避免用来计数的循环变量泄露为全局变量

var s = 'hello';  
for(var i = 0; i<s.length; i++){  
 console.log(s[i]);  
}  
console.log(i); //5

上面代码中，变量i只用来控制循环，但是循环结束后，它并没有消失，泄露成了全局变量。

注释:for循环用来计数的循环变量，很合适使用let声明。

1. ES6的块级作用域
2. let实际上为JavaScript新增了块级作用域。

function f1() {

let n = 5;

if (true) {

let n = 10;

}

console.log(n); // 5

}

上面的函数有两个代码块，都声明了变量n，运行后输出5。这表示外层代码块不受内层代码块的影响。如果使用var定义变量n，最后输出的值就是10。

1. **ES6允许块级作用域的任意嵌套**。

{{{{{let insane = 'Hello World'}}}}};

上面代码使用了一个五层的块级作用域。

1. 外层作用域无法读取内层作用域的变量。

{{{{

{let insane = 'Hello World'}

console.log(insane); // 报错}}}};

注释:内层作用域可以读取外层作用域的变量。

1. 内层作用域可以定义外层作用域的同名变量。

{{{{

let insane = 'Hello World';

{let insane = 'Hello World'}

}}}};

1. 块级作用域的出现，实际上使得获得广泛应用的立即执行匿名函数(IIFE)不再必要了。

// IIFE写法

(function () {

var tmp = ...;

...}());

// 块级作用域写法{

let tmp = ...;

...}

1. 块级作用域与函数声明

函数能不能在块级作用域之中声明，是一个相当令人混淆的问题。

|  |  |  |
| --- | --- | --- |
| 对比 | 除函数作用域之外的其他块级作用域之中声明函数 | 若允许，相当于var声明  还是let声明 |
| ES5规范 | 不允许 |  |
| ES5浏览器实现 | 允许 | var声明(变量提前) |
| ES5浏览器严格模式 | 不允许 |  |
| ES6规范 | 允许 | let声明(块级有效) |
| ES6浏览器实现 | 允许 | var声明(变量提前) |
| 注释:ES5浏览器的变量提前≠ES6浏览器的变量提前  例如:对于if(false){function f(){}} 块级作用域中定义函数的变量提前  ES5浏览器中: f = function(){}; ES6浏览器中: f = undefined;  对于if(true){function f(){}} 块级作用域中定义函数的变量提前  ES5浏览器中: f = function(){};ES6浏览器中: f = function(){}; | | |

1. ES5规定，函数只能在顶层作用域和函数作用域之中声明，不能在块级作用域声明。

// 情况一

if (true) {

function f() {}}

// 情况二

try {

function f() {}

} catch(e) {

}

上面代码的两种函数声明，根据ES5的规定都是非法的。

但是，浏览器没有遵守这个规定，还是支持在块级作用域之中声明函数，因此上面两种情况实际都能运行，不会报错。不过，“严格模式”下还是会报错。

// ES5严格模式

'use strict';

if (true) {

function f() {}

}

// 报错

注释:在ES5标准中，函数不能够在除函数作用域之外的其他块级作用域中声明。

ES6引入了块级作用域，明确允许在块级作用域之中声明函数。

// ES6严格模式

'use strict';

if (true) {

function f() {}

}

// 不报错

并且ES6规定，块级作用域之中，函数声明语句的行为类似于let，在块级作用域之外不可引用。

function f() { console.log('I am outside!'); }

(function () {

if (false) {

// 重复声明一次函数f

function f() { console.log('I am inside!'); }

}

f();

}());

上面代码在ES5中运行，会得到“I am inside!”，因为在if内声明的函数f会被提升到函数头部，实际运行的代码如下。

// ES5版本

function f() { console.log('I am outside!'); }

(function () {

function f() { console.log('I am inside!'); }

if (false) {

}

f();

}());

ES6的运行结果就完全不一样了，会得到“I am outside!”。因为块级作用域内声明的函数类似于let，对作用域之外没有影响，实际运行的代码如下。

// ES6版本

function f() { console.log('I am outside!'); }

(function () {

f();

}());

很显然，这种行为差异会对老代码产生很大影响。为了减轻因此产生的不兼容问题，ES6在附录B里面规定，浏览器的实现可以不遵守上面的规定，有自己的行为方式。

- 允许在块级作用域内声明函数。

- 函数声明类似于var，即会提升到全局作用域或函数作用域的头部。

- 同时，函数声明还会提升到所在的块级作用域的头部。

注意，上面三条规则只对ES6的浏览器实现有效，其他环境的实现不用遵守，还是将块级作用域的函数声明当作let处理。

前面那段代码，在Chrome环境下运行会报错。

// ES6的浏览器环境

function f() { console.log('I am outside!'); }

(function () {

if (false) {

// 重复声明一次函数f

function f() { console.log('I am inside!'); }

}

f();

}());

// Uncaught TypeError: f is not a function

上面的代码报错，是因为实际运行的是下面的代码。

// ES6的浏览器环境

function f() { console.log('I am outside!'); }

(function () {

var f = undefined;

if (false) {

function f() { console.log('I am inside!'); }

}

f();

}());

// Uncaught TypeError: f is not a function

1. 考虑到环境导致的行为差异太大，应该避免在块级作用域内声明函数。如果确实需要，也应该写成函数表达式，而不是函数声明语句。

// 函数声明语句

{

let a = 'secret';

function f() {

return a;

}

}

// 函数表达式

{

let a = 'secret';

let f = function () {

return a;

};

}

注释:使用函数表达式声明变量，即使变量提前，也只是var f = undefined。避免了ES5浏览器和ES6浏览器的差异性。

1. 另外，还有一个需要注意的地方。ES6的块级作用域允许声明函数的规则，只在使用大括号的情况下成立，如果没有使用大括号，就会报错。

// 不报错

'use strict';

if (true) {

function f() {}

}

// 报错

'use strict';

if (true)

function f() {}

### 2.3 const命令

1. const声明一个只读的常量。一旦声明，常量的值就不能改变。

const PI = 3.1415;

PI // 3.1415

PI = 3;

// TypeError: Assignment to constant variable.

上面代码表明改变常量的值会报错。

1. const声明的变量不得改变值，这意味着，const一旦声明变量，就必须立即初始化，不能留到以后赋值。

const foo;

// SyntaxError: Missing initializer in const declaration

上面代码表示，对于const来说，只声明不赋值，就会报错。

1. const的作用域与let命令相同：只在声明所在的块级作用域内有效。

if (true) {

const MAX = 5;

}

MAX // Uncaught ReferenceError: MAX is not defined

1. const命令声明的常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用。

if (true) {

console.log(MAX); // ReferenceError

const MAX = 5;

}

上面代码在常量MAX声明之前就调用，结果报错。

1. const声明的常量，也与let一样不可重复声明。

var message = "Hello!";

let age = 25;

// 以下两行都会报错

const message = "Goodbye!";

const age = 30;

1. 对于复合类型的变量，变量名不指向数据，而是指向数据所在的地址。const命令只是保证变量名指向的地址不变，并不保证该地址的数据不变，所以将一个对象声明为常量必须非常小心。

const foo = {};

foo.prop = 123;

foo.prop

// 123

foo = {}; // TypeError: "foo" is read-only

上面代码中，常量foo储存的是一个地址，这个地址指向一个对象。不可变的只是这个地址，即不能把foo指向另一个地址，但对象本身是可变的，所以依然可以为其添加新属性。

下面是另一个例子。

const a = [];

a.push('Hello'); // 可执行

a.length = 0; // 可执行

a = ['Dave']; // 报错

上面代码中，常量a是一个数组，这个数组本身是可写的，但是如果将另一个数组赋值给a，就会报错。

1. 如果真的想将对象冻结，应该使用Object.freeze方法。

const foo = **Object.freeze({});**

// 常规模式时，下面一行不起作用；

// 严格模式时，该行会报错

foo.prop = 123;

上面代码中，常量foo指向一个冻结的对象，所以添加新属性不起作用，严格模式时还会报错。

除了将对象本身冻结，对象的属性也应该冻结。下面是一个将对象彻底冻结的函数。

var constantize = (obj) => {

Object.freeze(obj);

Object.keys(obj).forEach( (key, index) => {

if ( typeof obj[key] === 'object' ) {

constantize( obj[key] );

}

});};

注释:ECMAScript5 Object的新属性方法

**Object.keys()**——获取所有的可枚举的属性，返回一个数组。

var obj = { 0 : "a", 1 : "b", 2 : "c"};

console.log(Object.keys(obj)); // console: ["0", "1", "2"]

1. ES5只有两种声明变量的方法：var命令和function命令。ES6除了添加let和const命令，后面章节还会提到，另外两种声明变量的方法：import命令和class命令。所以，ES6一共有6种声明变量的方法。

### 2.4 顶层对象的属性

1. 顶层对象，在浏览器环境指的是window对象，在Node指的是global对象。ES5之中，顶层对象的属性与全局变量是等价的。

window.a = 1;

a // 1

a = 2;

window.a // 2

上面代码中，顶层对象的属性赋值与全局变量的赋值，是同一件事。

顶层对象的属性与全局变量挂钩，被认为是JavaScript语言最大的设计败笔之一。

这样的设计带来了几个很大的问题，首先是没法在编译时就报出变量未声明的错误，只有运行时才能知道（因为全局变量可能是顶层对象的属性创造的，而属性的创造是动态的）；其次，程序员很容易不知不觉地就创建了全局变量（比如打字出错）；最后，顶层对象的属性是到处可以读写的，这非常不利于模块化编程。

另一方面，window对象有实体含义，指的是浏览器的窗口对象，顶层对象是一个有实体含义的对象，也是不合适的。

1. ES6为了改变这一点，一方面规定，为了保持兼容性，var命令和function命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，let命令、const命令、class命令声明的全局变量，不属于顶层对象的属性。也就是说，从ES6开始，全局变量将逐步与顶层对象的属性脱钩。

var a = 1;

// 如果在Node的REPL环境，可以写成global.a

// 或者采用通用方法，写成this.a

window.a // 1

let b = 1;

window.b // undefined

上面代码中，全局变量a由var命令声明，所以它是顶层对象的属性；全局变量b由let命令声明，所以它不是顶层对象的属性，返回undefined。

### 2.5 顶层对象

1. ES5的顶层对象，本身也是一个问题，因为它在各种实现里面是不统一的。

- 浏览器里面，顶层对象是window，但Node和Web Worker没有window。

- 浏览器和Web Worker里面，self也指向顶层对象，但是Node没有self。

- Node里面，顶层对象是global，但其他环境都不支持。

1. 同一段代码为了能够在各种环境都能取到顶层对象，现在一般是使用this变量，但是有局限性。

- 全局环境中，this会返回顶层对象。但是，Node模块和ES6模块中，this返回的是当前模块。

- 函数里面的this，如果函数不是作为对象的方法运行，而是单纯作为函数运行，this会指向顶层对象。但是，严格模式下，这时this会返回undefined。

- 不管是严格模式，还是普通模式，**new Function('return this')()，总是会返回全局对象**。但是，如果浏览器用了CSP（Content Security Policy，内容安全政策），那么eval、new Function这些方法都可能无法使用。

综上所述，很难找到一种方法，可以在所有情况下，都取到顶层对象。下面是两种勉强可以使用的方法。

// 方法一

(typeof window !== 'undefined'

? window

: (typeof process === 'object' &&

typeof require === 'function' &&

typeof global === 'object')

? global

: this);

// 方法二

var getGlobal = function () {

if (typeof self !== 'undefined') { return self; }

if (typeof window !== 'undefined') { return window; }

if (typeof global !== 'undefined') { return global; }

throw new Error('unable to locate global object');

};

注意: new Error的这种写法。

举例:try {

throw new Error('test');

} catch(e) {

alert(e);

}

1. 现在有一个提案，在语言标准的层面，引入global作为顶层对象。也就是说，在所有环境下，global都是存在的，都可以从它拿到顶层对象。

垫片库system.global模拟了这个提案，可以在所有环境拿到global。

// CommonJS的写法

require('system.global/shim')();

// ES6模块的写法

import shim from 'system.global/shim'; shim();

上面代码可以保证各种环境里面，global对象都是存在的。

// CommonJS的写法

var global = require('system.global')();

// ES6模块的写法

import getGlobal from 'system.global';

const global = getGlobal();

上面代码将顶层对象放入变量global。

## 3-变量的解构赋值

### 3.1 数组的解构赋值

1. 基本用法

ES6允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

1. 从数组中提取值，按照对应位置，对变量赋值。本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。

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

let [foo, [[bar], baz]] = [1, [[2], 3]];

let [ , , third] = ["foo", "bar", "baz"];

let [x, , y] = [1, 2, 3];

let [head, ...tail] = [1, 2, 3, 4];

let [x, y, ...z] = ['a'];

### 3.2 对象的解构赋值

### 3.3 字符串的解构赋值

### 3.4 数值和布尔值的解构赋值

### 3.5 函数参数的解构赋值

### 3.6 圆括号问题

### 3.7 用途

## 4-字符串的扩展

## 5-正则的扩展

## 6-数值的扩展

## 7-数组的扩展

## 8-函数的扩展

## 9-对象的扩展

## 10-Symbol

## 11-Proxy和Reflect

## 12-Set和Map数据结构

## 13-Interator和for...of循环

## 14-Generator函数

## 15-Promise对象

### 15.1 Promise的含义

1. Promise是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6将其写进了语言标准，统一了用法，原生提供了Promise对象。
2. 所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise是一个对象，从它可以获取异步操作的消息。Promise提供统一的API，各种异步操作都可以用同样的方法进行处理。

注释:Promise是一个容器，保存着异步操作的结果。

1. Promise对象有以下两个特点。

(1)对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：Pending（进行中）、Resolved（已完成，又称Fulfilled）和Rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

(2)一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从Pending变为Resolved和从Pending变为Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

1. 有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。
2. Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三，当处于Pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。
3. 如果某些事件不断地反复发生，一般来说，使用stream模式是比部署Promise更好的选择。

### 15.2 基本用法

1. ES6规定，**Promise对象是一个构造函数**，用来生成Promise实例。

下面代码创造了一个Promise实例。

var promise = new Promise(function(resolve, reject) {

// ... some code

if (/\* 异步操作成功 \*/){

resolve(value);

} else {

reject(error);

}

});

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由JavaScript引擎提供，不用自己部署。

resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从Pending变为Resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从Pending变为Rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

Promise实例生成以后，可以用then方法分别指定Resolved状态和Reject状态的**回调函数**。

注释:Promise新建后就会立即执行，使用**then方法**指定Promise对象**状态改变时的回调函数**。

promise.then(function(value) {

// success

}, function(error) {

// failure

});

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为Resolved时调用，第二个回调函数是Promise对象的状态变为Reject时调用。其中，第二个函数是可选的，不一定要提供。这两个函数都接受Promise对象传出的值作为参数。

下面是一个Promise对象的简单例子。

function timeout(ms) {

return new Promise((resolve, reject) => {

setTimeout(resolve, ms, **'done'**);

});

}

timeout(100).then((value) => {

console.log(value);

});

上面代码中，timeout方法返回一个Promise实例，表示一段时间以后才会发生的结果。过了指定的时间（ms参数）以后，Promise实例的状态变为Resolved，就会触发then方法绑定的回调函数。

Promise新建后就会立即执行。

let promise = new Promise(function(resolve, reject) {

console.log('Promise');

resolve();

});

promise.then(function() {

console.log('Resolved.');

});

console.log('Hi!');

// Promise

// Hi!

// Resolved

上面代码中，Promise新建后立即执行，所以首先输出的是“Promise”。然后，**then方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行**，所以“Resolved”最后输出。

注意:Promise实例创建之后立即异步执行，Promise实例的then方法指定的回调函数，在当前脚本所有同步任务执行完成之后才执行。在上个例子中，**即使Promise实例中并没有异步操作，then方法的回调函数仍然最后执行**。

1. 下面是异步加载图片的例子。

function loadImageAsync(url) {

return new Promise(function(resolve, reject) {

var image = new Image();

image.onload = function() {

resolve(image);

};

image.onerror = function() {

reject(new Error('Could not load image at ' + url));

};

image.src = url;

});

}

上面代码中，使用Promise包装了一个图片加载的异步操作。如果加载成功，就调用resolve方法，否则就调用reject方法。

1. **下面是一个用Promise对象实现的Ajax操作的例子。**

var getJSON = function(url) {

var promise = new Promise(function(resolve, reject){

var client = new XMLHttpRequest();

client.open("GET", url);

client.onreadystatechange = handler;

client.responseType = "json";

client.setRequestHeader("Accept", "application/json");

client.send();

function handler() {

if (this.readyState !== 4) {

return;

}

if (this.status === 200) {

resolve(this.response);

} else {

reject(new Error(this.statusText));

}

};

});

return promise;

};

getJSON("/posts.json").then(function(json) {

console.log('Contents: ' + json);

}, function(error) {

console.error('出错了', error);

});

上面代码中，getJSON是对XMLHttpRequest对象的封装，用于发出一个针对JSON数据的HTTP请求，并且返回一个Promise对象。需要注意的是，在getJSON内部，resolve函数和reject函数调用时，都带有参数。

如果调用resolve函数和reject函数时带有参数，那么它们的参数会被传递给回调函数。reject函数的参数通常是Error对象的实例，表示抛出的错误；resolve函数的参数除了正常的值以外，还可能是另一个Promise实例，表示异步操作的结果有可能是一个值，也有可能是另一个异步操作，比如像下面这样。

var p1 = new Promise(function (resolve, reject) {

// ...});

var p2 = new Promise(function (resolve, reject) {

// ... resolve(p1);})

上面代码中，p1和p2都是Promise的实例，但是p2的resolve方法将p1作为参数，即一个异步操作的结果是返回另一个异步操作。

注意，这时p1的状态就会传递给p2，也就是说，p1的状态决定了p2的状态。如果p1的状态是Pending，那么p2的回调函数就会等待p1的状态改变；如果p1的状态已经是Resolved或者Rejected，那么p2的回调函数将会立刻执行。

注意:Promise实例对象的resolve方法执行时**传递的实参**被Promise实例对象的then方法的第二个参数函数(resolve方法的回调函数)接收。若该实参为另一个Promise实例对象，则resolve方法的回调函数是否执行以及执行时机取决于另一个Promise实例对象的状态。

var p1 = new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('fail')), 3000)})

var p2 = new Promise(function (resolve, reject) {

setTimeout(() => resolve(p1), 1000)})

p2

.then(result => console.log(result))

.catch(error => console.log(error))

// Error: fail

上面代码中，p1是一个Promise，3秒之后变为rejected。p2的状态在1秒之后改变，resolve方法返回的是p1。此时，由于p2返回的是另一个Promise，所以后面的then语句都变成针对后者（p1）。又过了2秒，p1变为rejected，导致触发catch方法指定的回调函数。

### 15.3 Promise.prototype.then()

1. Promise实例具有then方法，也就是说，then方法是定义在原型对象Promise.prototype上的。它的作用是为Promise实例添加状态改变时的回调函数。前面说过，then方法的第一个参数是Resolved状态的回调函数，第二个参数（可选）是Rejected状态的回调函数。

then方法返回的是一个新的Promise实例（注意，不是原来那个Promise实例）。因此可以采用链式写法，即then方法后面再调用另一个then方法。

getJSON("/posts.json").then(function(json) {

return json.post;}).then(function(post) {

// ...

});

上面的代码使用then方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。

1. 采用链式的then，可以指定一组按照次序调用的回调函数。这时，**前一个回调函数，有可能返回的还是一个Promise对象（即有异步操作）**，这时后一个回调函数，就会等待该Promise对象的状态发生变化，才会被调用。

getJSON("/post/1.json").then(function(post) {

return getJSON(post.commentURL);

}).then(function funcA(comments) {

console.log("Resolved: ", comments);

}, function funcB(err){

console.log("Rejected: ", err);

});

上面代码中，第一个then方法指定的回调函数，返回的是另一个Promise对象。这时，第二个then方法指定的回调函数，就会等待这个新的Promise对象状态发生变化。如果变为Resolved，就调用funcA，如果状态变为Rejected，就调用funcB。

如果采用箭头函数，上面的代码可以写得更简洁。

getJSON("/post/1.json").then(

post => getJSON(post.commentURL)

).then(

comments => console.log("Resolved: ", comments),

err => console.log("Rejected: ", err)

);

总结:Promise实例具有then方法，该方法返回一个新的Promise实例。该Promise实例的状态取决于then方法中的函数，当then方法中的函数返回一般的值时(return 值)时，表示该Promise实例的状态成为完成(Resolved)，当then方法中的函数返回另一个Promise实例时，另一个Promise实例的状态转变为Resolved（已完成）时，表示该Promise实例的状态成为完成(Resolved)，另一个Promise实例的状态转变为Rejected（已失败）时，表示该Promise实例的状态成为以失败Rejected（已失败）。

**总结: Promise实例有以下几种存在形式。**

1. new Promise(function (resolve, reject) {})
2. new Promise(function (resolve, reject) {

resolve(promise);

})

1. promise.then()
2. promise.then(function(){return promise})
3. Promise.resolve(promise)

解释: 每一个标红代码块即表示一种promise的存在形式。

其中，(1)为通过Promise构造函数生成，这是其他所有Promise的存在的源头；(3)为promise实例对象的then()方法返回一个新的promise对象；**(2)**为一个新的promise对象作为resolve方法的实参，这种使用方式会改变后面then语句的promise指向(P39)；**(4)**为promise实例作为resolve方法回调函数的返回值，这种使用方式会将promise实例的状态当做then方法返回的promise对象的状态(P39)。(5)为promise作为Promise.resolve()方法的参数，那么Promise.resolve将不做任何修改、原封不动地返回这个实例(p47)。

### 15.4 Promise.prototype.catch()

1. Promise.prototype.catch方法是.then(null, rejection)的别名，用于指定发生错误时的回调函数。

getJSON("/posts.json").then(function(posts) {

// ...

}).catch(function(error) {

// 处理 getJSON 和 前一个回调函数运行时发生的错误

console.log('发生错误！', error);

});

上面代码中，getJSON方法返回一个Promise对象，如果该对象状态变为Resolved，则会调用then方法指定的回调函数；如果异步操作抛出错误，状态就会变为Rejected，就会调用catch方法指定的回调函数，处理这个错误。另外，then方法指定的回调函数，如果运行中抛出错误，也会被catch方法捕获。

p.then((val) => console.log("fulfilled:", val))

.catch((err) => console.log("rejected:", err));

// 等同于

p.then((val) => console.log("fulfilled:", val))

.then(**null**, (err) => console.log("rejected:", err));

注释:catch方法既会捕获promise实例中代码执行过程抛出的错误，又会捕获回调函数中代码执行过程抛出的错误(Promise对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。)。

1. 下面是一个例子。

var promise = new Promise(function(resolve, reject) {

throw new Error('test');

});

promise.catch(function(error) {

console.log(error);

});

// Error: test

上面代码中，promise抛出一个错误，就被catch方法指定的回调函数捕获。注意，**上面的写法与下面两种写法是等价的**。

// 写法一

var promise = new Promise(function(resolve, reject) {

try {

throw new Error('test');

} catch(e) {

reject(e);

}

});

promise.catch(function(error) {

console.log(error);});

// 写法二

var promise = new Promise(function(resolve, reject) {

reject(new Error('test'));

});

promise.catch(function(error) {

console.log(error);

});

比较上面两种写法，可以发现**reject方法的作用，等同于抛出错误**。

注释:throw new Error('test');等价于reject(new Error('test'));

1. 如果Promise状态已经变成Resolved，再抛出错误是无效的。

var promise = new Promise(function(resolve, reject) {

resolve('ok');

throw new Error('test');

});

promise

.then(function(value) { console.log(value) })

.catch(function(error) { console.log(error) });

// ok

上面代码中，Promise在resolve语句后面，再抛出错误，不会被捕获，等于没有抛出。

1. Promise对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个catch语句捕获。

getJSON("/post/1.json").then(function(post) {

return getJSON(post.commentURL);

}).then(function(comments) {

// some code

}).catch(function(error) {

// 处理前面三个Promise产生的错误

});

上面代码中，一共有三个Promise对象：一个由getJSON产生，两个由then产生。它们之中任何一个抛出的错误，都会被最后一个catch捕获。

1. 一般来说，不要在then方法里面定义Reject状态的回调函数（即then的第二个参数），总是使用catch方法。

// bad

promise

.then(function(data) {

// success

}, function(err) {

// error

});

// good

promise

.then(function(data) { //cb

// success

})

.catch(function(err) {

// error

});

上面代码中，第二种写法要好于第一种写法，理由是第二种写法可以捕获前面then方法执行中的错误，也更接近同步的写法（try/catch）。因此，建议总是使用catch方法，而不使用then方法的第二个参数。

1. 跟传统的try/catch代码块不同的是，如果没有使用catch方法指定错误处理的回调函数，Promise对象抛出的错误不会传递到外层代码，即不会有任何反应。

var someAsyncThing = function() {

return new Promise(function(resolve, reject) {

// 下面一行会报错，因为x没有声明

resolve(x + 2);

});

};

someAsyncThing().then(function() {

console.log('everything is great');

});

上面代码中，someAsyncThing函数产生的Promise对象会报错，但是由于没有指定catch方法，这个错误不会被捕获，也不会传递到外层代码，导致运行后没有任何输出。注意，Chrome浏览器不遵守这条规定，它会抛出错误“ReferenceError: x is not defined”。

var promise = new Promise(function(resolve, reject) {

resolve("ok");

setTimeout(function() { throw new Error('test') }, 0)

});

promise.then(function(value) { console.log(value) });

// ok

// Uncaught Error: test

上面代码中，Promise指定在下一轮“事件循环”再抛出错误，结果由于没有指定使用try...catch语句，就冒泡到最外层，成了未捕获的错误。因为此时，Promise的函数体已经运行结束了，所以这个错误是在Promise函数体外抛出的。

1. Node.js有一个unhandledRejection事件，专门监听未捕获的reject错误。

process.on('unhandledRejection', function (err, p) {

console.error(err.stack)

});

上面代码中，unhandledRejection事件的监听函数有两个参数，第一个是错误对象，第二个是报错的Promise实例，它可以用来了解发生错误的环境信息。

1. 需要注意的是，catch方法返回的还是一个Promise对象，因此后面还可以接着调用then方法。

var someAsyncThing = function() {

return new Promise(function(resolve, reject) {

// 下面一行会报错，因为x没有声明

resolve(x + 2);

});

};

someAsyncThing().catch(function(error) {

console.log('oh no', error);

}).then(function() {

console.log('carry on');

});

// oh no [ReferenceError: x is not defined]

// carry on

上面代码运行完catch方法指定的回调函数，会接着运行后面那个then方法指定的回调函数。如果没有报错，则会跳过catch方法。

Promise.resolve()

.catch(function(error) {

console.log('oh no', error);

}).then(function() {

console.log('carry on');

});

// carry on

上面的代码因为没有报错，跳过了catch方法，直接执行后面的then方法。此时，要是then方法里面报错，就与前面的catch无关了。

1. catch方法之中，还能再抛出错误。

var someAsyncThing = function() {

return new Promise(function(resolve, reject) {

// 下面一行会报错，因为x没有声明

resolve(x + 2);

});

};

someAsyncThing().then(function() {

return someOtherAsyncThing();

}).catch(function(error) {

console.log('oh no', error);

// 下面一行会报错，因为y没有声明

y + 2;

}).then(function() {

console.log('carry on');

});

// oh no [ReferenceError: x is not defined]

上面代码中，catch方法抛出一个错误，因为后面没有别的catch方法了，导致这个错误不会被捕获，也不会传递到外层。如果改写一下，结果就不一样了。

someAsyncThing().then(function() {

return someOtherAsyncThing();

}).catch(function(error) {

console.log('oh no', error);

// 下面一行会报错，因为y没有声明

y + 2;

}).catch(function(error) {

console.log('carry on', error);

});

// oh no [ReferenceError: x is not defined]

// carry on [ReferenceError: y is not defined]

### 15.5 Promise.all()

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

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

上面代码中，Promise.all方法接受一个数组作为参数，p1、p2、p3都是Promise对象的实例，如果不是，就会先调用下面讲到的Promise.resolve方法，将参数转为Promise实例，再进一步处理。（Promise.all方法的参数可以不是数组，但必须具有Iterator接口，且返回的每个成员都是Promise实例。）

1. p的状态由p1、p2、p3决定，分成两种情况。

(1)只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

(2)只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

下面是一个具体的例子。

// 生成一个Promise对象的数组

var promises = [2, 3, 5, 7, 11, 13].map(function (id) {

return getJSON("/post/" + id + ".json");

});

Promise.all(promises).then(function (posts) {

// ...

}).catch(function(reason){

// ...

});

上面代码中，promises是包含6个Promise实例的数组，只有这6个实例的状态都变成fulfilled，或者其中有一个变为rejected，才会调用Promise.all方法后面的回调函数。

下面是另一个例子。

const databasePromise = connectDatabase();

const booksPromise = databaseProimse

.then(findAllBooks);

const userPromise = databasePromise

.then(getCurrentUser);

Promise.all([

booksPromise,

userPromise])

.then(([books, user]) => pickTopRecommentations(books, user));

上面代码中，booksPromise和userPromise是两个异步操作，只有等到它们的结果都返回了，才会触发pickTopRecommentations这个回调函数。

### 15.6 Promise.race()

1. Promise.race方法同样是将多个Promise实例，包装成一个新的Promise实例。

var p = Promise.race([p1,p2,p3]);

上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的Promise实例的返回值，就传递给p的回调函数。

1. Promise.race方法的参数与Promise.all方法一样，如果不是Promise实例，就会先调用下面讲到的Promise.resolve方法，将参数转为Promise实例，再进一步处理。

下面是一个例子，如果指定时间内没有获得结果，就将Promise的状态变为reject，否则变为resolve。

var p = Promise.race([

fetch('/resource-that-may-take-a-while'),

new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('request timeout')), 5000)

})

])

p.then(response => console.log(response))

p.catch(error => console.log(error))

上面代码中，如果5秒之内fetch方法无法返回结果，变量p的状态就会变为rejected，从而触发catch方法指定的回调函数。

### 15.7 Promise.resolve()

1. 有时需要将现有对象转为Promise对象，Promise.resolve方法就起到这个作用。

var jsPromise = Promise.resolve($.ajax('/whatever.json'));

上面代码将jQuery生成的deferred对象，转为一个新的Promise对象。

Promise.resolve等价于下面的写法。

Promise.resolve('foo')

// 等价于

new Promise(resolve => resolve('foo'))

（1）参数是一个Promise实例

如果参数是Promise实例，那么Promise.resolve将不做任何修改、原封不动地返回这个实例。

1. 参数是一个thenable对象

thenable对象指的是具有then方法的对象，比如下面这个对象。

let thenable = {

then: function(resolve, reject) {

resolve(42);

}};

Promise.resolve方法会将这个对象转为Promise对象，然后就立即执行thenable对象的then方法。

let thenable = {

then: function(resolve, reject) {

resolve(42);

}

};

let p1 = Promise.resolve(thenable);

p1.then(function(value) {

console.log(value); // 42

});

上面代码中，thenable对象的then方法执行后，对象p1的状态就变为resolved，从而立即执行最后那个then方法指定的回调函数，输出42。

1. 参数不是具有then方法的对象，或根本就不是对象

如果参数是一个原始值，或者是一个不具有then方法的对象，则Promise.resolve方法返回一个新的Promise对象，状态为Resolved。

var p = Promise.resolve('Hello');

p.then(function (s){

console.log(s)

});

// Hello

上面代码生成一个新的Promise对象的实例p。由于字符串Hello不属于异步操作（判断方法是它不是具有then方法的对象），返回Promise实例的状态从一生成就是Resolved，所以回调函数会立即执行。Promise.resolve方法的参数，会同时传给回调函数。

（4）**不带有任何参数**

Promise.resolve方法允许调用时不带参数，直接返回一个Resolved状态的Promise对象。

所以，如果希望得到一个Promise对象，比较方便的方法就是直接调用Promise.resolve方法。

var p = Promise.resolve();

p.then(function () {

// ...

});

上面代码的变量p就是一个Promise对象。

需要注意的是，立即resolve的Promise对象，**是在本轮“事件循环”（event loop）的结束时，而不是在下一轮“事件循环”的开始时**。

setTimeout(function () {

console.log('three');

}, 0);

Promise.resolve().then(function () {

console.log('two');

});

console.log('one');

// one

// two

// three

上面代码中，setTimeout(fn, 0)在下一轮“事件循环”开始时执行，Promise.resolve()在本轮“事件循环”结束时执行，console.log(‘one’)则是立即执行，因此最先输出。

### 15.8 Promise.reject()

1. Promise.reject(reason)方法也会返回一个新的Promise实例，该实例的状态为rejected。它的参数用法与Promise.resolve方法完全一致。

var p = Promise.reject('出错了');

// 等同于

var p = new Promise((resolve, reject) => reject('出错了'))

p.then(null, function (s){

console.log(s)

});

// 出错了

上面代码生成一个Promise对象的实例p，状态为rejected，回调函数会立即执行。

### 15.9 两个有用的附加方法

1. ES6的Promise API提供的方法不是很多，有些有用的方法可以自己部署。下面介绍如何部署两个不在ES6之中、但很有用的方法。
2. done()

Promise对象的回调链，**不管以then方法或catch方法结尾，要是最后一个方法抛出错误，都有可能无法捕捉到**（因为Promise内部的错误不会冒泡到全局）。因此，我们可以提供一个done方法，总是处于回调链的尾端，保证抛出任何可能出现的错误。

asyncFunc()

.then(f1)

.catch(r1)

.then(f2)

.done();

它的实现代码相当简单。

Promise.prototype.done = function (onFulfilled, onRejected) {

this.then(onFulfilled, onRejected)

.catch(function (reason) {

// 抛出一个全局错误

setTimeout(() => { throw reason }, 0);

});

};

从上面代码可见，done方法的使用，可以像then方法那样用，提供Fulfilled和Rejected状态的回调函数，**也可以不提供任何参数**。但不管怎样，done都会捕捉到任何可能出现的错误，并向全局抛出。

1. finally()

finally方法用于指定不管Promise对象最后状态如何，都会执行的操作。它与done方法的最大区别，它接受一个普通的回调函数作为参数，该函数不管怎样都必须执行。

下面是一个例子，服务器使用Promise处理请求，然后使用finally方法关掉服务器。

server.listen(0)

.then(function () {

// run test

})

.finally(server.stop);

它的实现也很简单。

Promise.prototype.finally = function (callback) {

let P = this.constructor;

return this.then(

value => P.resolve(callback()).then(() => value),

reason => P.resolve(callback()).then(() => { throw reason })

);

};

注释:①this.constructor==Promise, P.resolve(callback())即是调用Promise.resolve的第三种情况(P47)。即：Promise.resolve(参数)如果参数是一个原始值，或者是一个不具有then方法的对象，则Promise.resolve方法返回一个新的Promise对象，状态为Resolved。Promise.resolve方法的参数，会同时传给回调函数。②finally方法的封装有两个效果:一是执行callback函数，而是不改变前一个.then()方法的返回值。

### 15.10 应用

1. 加载图片

我们可以将图片的加载写成一个Promise，一旦加载完成，Promise的状态就发生变化。

const preloadImage = function (path) {

return new Promise(function (resolve, reject) {

var image = new Image();

image.onload = resolve;

image.onerror = reject;

image.src = path;

});

};

注释:image.src写在image.onload之后image.onerror，避免图片已经加载，而事件还未绑定。

1. Generator函数与Promise的结合

使用Generator函数管理流程，遇到异步操作的时候，通常返回一个Promise对象。

function getFoo () {

return new Promise(function (resolve, reject){

resolve('foo');

});

}

var g = function\* () {

try {

var foo = yield getFoo();

console.log(foo);

} catch (e) {

console.log(e);

}

};

function run (generator) {

var it = generator();

function go(result) {

if (result.done) return result.value;

return result.value.then(function (value) {

return go(it.next(value));

}, function (error) {

return go(it.throw(error));

});

}

go(it.next());

}

run(g);

上面代码的Generator函数g之中，有一个异步操作getFoo，它返回的就是一个Promise对象。函数run用来处理这个Promise对象，并调用下一个next方法。

## 16-异步操作和Async函数

## 17-Class

## 18-Decorator

## 19-Module

## 20-编程风格

## 21-读懂规格

## 22-二进制数组

## 23-SIMD

## 24-参考链接