# ECMASript简介

·ECMA

- ECMA（European Computer Manufacturers Association）中文名称为欧洲计算机制造商协会，这个组织的目标是评估、开发和认可电信和计算机标准。1994 年后该组织改名为 Ecma 国际

·ECMAScript

- ECMAScript 是由 Ecma 国际通过 ECMA-262 标准化的脚本程序设计语言

·ECMA-262

- Ecma 国际制定了许多标准，而 ECMA-262 只是其中的一个，所有标准列表查看http://www.ecma-international.org/publications/standards/Standard.htm

·自es6开始每年都会发行一个新的版本

·谁在维护 ECMA-262

- TC39（Technical Committee 39）是推进 ECMAScript 发展的委员会。其会员都是公司（其中主要是浏览器厂商，有苹果、谷歌、微软、因特尔等）。TC39 定期召开会议，会议由会员公司的代表与特邀专家出席

·为什么要学习 ES6

- ES6 的版本变动内容最多，具有里程碑意义

- ES6 加入许多新的语法特性，编程实现更简单、高效

- ES6 是前端发展趋势，就业必备技能

·ES6 兼容性

- http://kangax.github.io/compat-table/es6/ 可查看兼容性

# let 关键字

·let 关键字用来声明变量，使用 let 声明的变量有几个特点

·不允许重复声明，否则会报错

//1. 变量不能重复声明  
let ***star*** = '罗志祥';  
let ***star*** = '小猪';

·块级作用域

- 声明的变量只在块内有效

//2. 块儿级作用域 全局, 函数, eval  
// if else while for   
{  
 let girl = '周扬青';  
}  
***console***.log(girl);

·不存在变量提升

- 不允许在变量声明前使用变量，否则会报错

//3. 不存在变量提升  
***console***.log(***song***);  
let ***song*** = '恋爱达人';

·不影响作用域链

//4. 不影响作用域链  
{  
 let school = '尚硅谷';  
 function fn(){  
 ***console***.log(school);  
 }  
 fn();  
}

·应用场景

- 以后声明变量使用 let 就对了

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>let 关键字</title>  
</head>  
<body>  
  
<script type="text/javascript">  
 //声明变量  
 let ***a***;  
 let ***b***,***c***,***d***;  
 let ***e*** = 100;  
 let ***f*** = 521,***g*** = "iloveyou",***h***=[];  
 //1. 变量不能重复声明  
 let ***star*** = '罗志祥';  
 let ***star*** = '小猪';  
  
 //2. 块儿级作用域 全局, 函数, eval  
 // if else while for  
 // {  
 // let girl = '周扬青';  
 // }  
 // console.log(girl);  
  
 //3. 不存在变量提升  
 // console.log(song);  
 // let song = '恋爱达人';  
  
 //4. 不影响作用域链  
 {  
 let school = '尚硅谷';  
 function fn(){  
 ***console***.log(school);  
 }  
 fn();  
 }  
</script>  
</body>  
</html>

## let经典案例实践

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>点击 DIV 换色</title>  
 <link crossorigin="anonymous" href="https://cdn.bootcss.com/twitter-bootstrap/3.3.7/css/bootstrap.min.css"  
 rel="stylesheet">  
 <style>  
 .item {  
 width: 100px;  
 height: 50px;  
 border: solid 1px rgb(42, 156, 156);  
 float: left;  
 margin-right: 10px;  
 }  
 </style>  
</head>  
  
<body>  
<div class="container">  
 <h2 class="page-header">点击切换颜色</h2>  
 <div class="item"></div>  
 <div class="item"></div>  
 <div class="item"></div>  
</div>  
<script>  
 //获取div元素对象  
 let ***items*** = ***document***.getElementsByClassName("item");  
 //遍历并绑定事件  
 for(let i=0;i<***items***.length;i++){  
 ***items***[i].onclick = function () {  
 //修改当前元素的背景颜色  
 //如果用ver声明i变量，只能用this  
 // this.style.background= "pink";  
 //如果用let声明i变量，可以之间用items[i]，因为let声明的变量在各个代码块之间是互不影响的  
 ***items***[i].style.background= "pink";  
 }  
 }  
</script>  
</body>  
</html>

# const 关键字

·const 关键字用来声明常量，const 声明有以下特点

·声明必须赋初始值，否则会报错

//1. 一定要赋初始值  
const A;

·标识符一般为大写

- 用小写也没问题

//2. 一般常量使用大写(潜规则)  
const ***a*** = 100;

·不允许重复声明

·值不允许修改，否则会报错

//3. 常量的值不能修改  
const ***SCHOOL*** = '尚硅谷';  
***SCHOOL*** = 'ATGUIGU';

·块儿级作用域

//4. 块儿级作用域  
{  
 const PLAYER = 'UZI';  
}  
***console***.log(PLAYER);

·注意

- 对象属性修改和数组元素变化不会触发 const 错误

//5. 对于数组和对象的元素修改, 不算做对常量的修改, 不会报错  
const ***TEAM*** = ['UZI','MXLG','Ming','Letme'];  
***TEAM***.push('Meiko');//不报错  
***TEAM*** = 100;//会报错

·应用场景

- 声明对象类型使用 const，非对象类型声明选择 let

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>const 定义常量</title>  
</head>  
<body>  
 <script>  
 //1. 一定要赋初始值  
 // const A;  
 //2. 一般常量使用大写(潜规则)  
 // const a = 100;  
 //3. 常量的值不能修改  
 // const SCHOOL = '尚硅谷';  
 // SCHOOL = 'ATGUIGU';  
 //4. 块儿级作用域  
 // {  
 // const PLAYER = 'UZI';  
 // }  
 // console.log(PLAYER);  
 //5. 对于数组和对象的元素修改, 不算做对常量的修改, 不会报错  
 const ***TEAM*** = ['UZI','MXLG','Ming','Letme'];  
 ***TEAM***.push('Meiko');//不报错  
 // TEAM = 100;//会报错  
 </script>  
</body>  
</html>

# 变量的解构赋值

·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

为解构赋值

·数组的解构赋值

// 1. 数组的解构赋值  
const ***F4*** = ['小沈阳','刘能','赵四','宋小宝'];  
let [***xiao***, ***liu***, ***zhao***, ***song***] = ***F4***;  
***console***.log(***xiao***);//小沈阳  
***console***.log(***liu***);//刘能  
***console***.log(***zhao***);//赵四  
***console***.log(***song***);//宋小宝

·对象的解构赋值

//2. 对象的解构  
const ***zhao*** = {  
 name: '赵本山',  
 age: '不详',  
 xiaopin: function(){  
 ***console***.log("我可以演小品");  
 }  
};  
  
let {***name***, ***age***, xiaopin} = ***zhao***;  
***console***.log(***name***);//赵本山  
***console***.log(***age***);//不详  
***console***.log(xiaopin);  
//调用xiaopin函数  
xiaopin();  
  
// let {xiaopin} = zhao;  
// //直接调用xiaopin函数  
// xiaopin();

·注意：如果频繁使用对象方法、数组元素，就可以使用解构赋值形式

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>变量的解构赋值</title>  
</head>  
<body>  
 <script>  
 //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，  
 //这被称为解构赋值。  
 // 1. 数组的解构赋值  
 // const F4 = ['小沈阳','刘能','赵四','宋小宝'];  
 // let [xiao, liu, zhao, song] = F4;  
 // console.log(xiao);//小沈阳  
 // console.log(liu);//刘能  
 // console.log(zhao);//赵四  
 // console.log(song);//宋小宝  
  
 //2. 对象的解构  
 const ***zhao*** = {  
 name: '赵本山',  
 age: '不详',  
 xiaopin: function(){  
 ***console***.log("我可以演小品");  
 }  
 };  
  
 let {***name***, ***age***, xiaopin} = ***zhao***;  
 ***console***.log(***name***);//赵本山  
 ***console***.log(***age***);//不详  
 ***console***.log(xiaopin);  
 //调用xiaopin函数  
 xiaopin();  
  
 // let {xiaopin} = zhao;  
 // //直接调用xiaopin函数  
 // xiaopin();  
  
  
 </script>  
</body>  
  
</html>

# 模板字符串

·模板字符串（template string）是增强版的字符串，用反引号（`）标识

//1. 声明  
let ***str*** = `我也是一个字符串哦!`;  
***console***.log(***str***, typeof ***str***);

·字符串中可以出现换行符

//2. 内容中可以直接出现换行符  
let ***str*** = `<ul>  
 <li>沈腾</li>  
 <li>玛丽</li>  
 <li>魏翔</li>  
 <li>艾伦</li>  
 </ul>`;

·可以使用 ${xxx} 形式输出变量

//3. 变量拼接  
let ***lovest*** = '魏翔';  
let ***out*** = `${***lovest***}是我心目中最搞笑的演员!!`;  
***console***.log(***out***);//魏翔是我心目中最搞笑的演员!!

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>模板字符串</title>  
</head>  
<body>  
 <script>  
 // ES6 引入新的声明字符串的方式 『``』 '' ""   
 //1. 声明  
 let ***str*** = `我也是一个字符串哦!`;  
 ***console***.log(***str***, typeof ***str***);  
  
 //2. 内容中可以直接出现换行符  
 let ***str*** = `<ul>  
 <li>沈腾</li>  
 <li>玛丽</li>  
 <li>魏翔</li>  
 <li>艾伦</li>  
 </ul>`;  
 //3. 变量拼接  
 let ***lovest*** = '魏翔';  
 let ***out*** = `${***lovest***}是我心目中最搞笑的演员!!`;  
 ***console***.log(***out***);//魏翔是我心目中最搞笑的演员!!  
  
 </script>  
</body>  
</html>

# 简化对象写法

·ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。这

样的书写更加简洁

//ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。  
//这样的书写更加简洁  
let ***name*** = '尚硅谷';  
let change = function(){  
 ***console***.log('我们可以改变你!!');  
}  
const ***school*** = {  
 ***name***,  
 change,  
 //可以把:function省略  
 improve(){  
 ***console***.log("我们可以提高你的技能");  
 }  
}  
***console***.log(***school***);

·注意

对象简写形式简化了代码，所以以后用简写就对了

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>简化对象写法</title>  
</head>  
<body>  
 <script>  
 //ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。  
 //这样的书写更加简洁  
 let ***name*** = '尚硅谷';  
 let change = function(){  
 ***console***.log('我们可以改变你!!');  
 }  
 const ***school*** = {  
 ***name***,  
 change,  
 //可以把:function省略  
 improve(){  
 ***console***.log("我们可以提高你的技能");  
 }  
 }  
 ***console***.log(***school***);  
  
 </script>  
</body>  
</html>

# 箭头函数

·ES6 允许使用「箭头」（=>）定义函数。

·(参数1,参数2)=>{}相当于function(参数1,参数2){}

// ES6 允许使用(参数1，参数2)=>{}定义函数。  
//声明一个函数  
// let fn = function(){  
// return a + b;  
// }  
let fn = (a,b) => {  
 return a + b;  
}  
// 调用函数  
let ***result*** = fn(1, 2);  
***console***.log(***result***);//3

·如果形参只有一个，则小括号可以省略

//1) 省略小括号, 当形参有且只有一个的时候  
let add = n => {  
 return n + n;  
}  
***console***.log(add(9));//18

·函数体如果只有一条语句，则花括号可以省略，函数的返回值为该条语句的

执行结果

//2) 省略花括号, 当代码体只有一条语句的时候, 此时 return 必须省略  
// 而且语句的执行结果就是函数的返回值  
let pow = n => n \* n;  
***console***.log(pow(8));

·箭头函数 this 指向声明时所在作用域下 this 的值

//设置 window 对象的 name 属性  
***window***.name = '尚硅谷';  
const ***school*** = {  
 name: "ATGUIGU"  
}  
  
// 直接调用  
getName();//尚硅谷  
getName2();//尚硅谷  
  
//call 方法调用  
getName.call(***school***);//ATGUIGU  
getName2.call(***school***);//尚硅谷

·箭头函数不能作为构造函数实例化

//2. 不能作为构造实例化对象  
let Person = (name, age) => {  
 this.name = name;  
 this.age = age;  
}  
let ***me*** = new Person('xiao',30);  
***console***.log(***me***);//报错

·不能使用 arguments

//3. 不能使用 arguments 变量  
let fn = () => {  
 ***console***.log(arguments);  
}  
fn(1,2,3);//报错

·注意：箭头函数不会更改 this 指向，用来指定回调函数会非常合适

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>箭头函数</title>  
</head>  
  
<body>  
 <script>  
 // ES6 允许使用(参数1，参数2)=>{}定义函数。  
 //声明一个函数  
 // let fn = function(){  
 // return a + b;  
 // }  
 // let fn = (a,b) => {  
 // return a + b;  
 // }  
 // // 调用函数  
 // let result = fn(1, 2);  
 // console.log(result);//3  
  
  
 // //1. this 是静态的. this 始终指向函数声明时所在作用域下的 this 的值  
 // function getName(){  
 // console.log(this.name);  
 // }  
 // let getName2 = () => {  
 // console.log(this.name);  
 // }  
 //  
 // //设置 window 对象的 name 属性  
 // window.name = '尚硅谷';  
 // const school = {  
 // name: "ATGUIGU"  
 // }  
 //  
 // // 直接调用  
 // getName();//尚硅谷  
 // getName2();//尚硅谷  
 //  
 // //call 方法调用  
 // getName.call(school);//ATGUIGU  
 // getName2.call(school);//尚硅谷  
  
 //2. 不能作为构造实例化对象  
 // let Person = (name, age) => {  
 // this.name = name;  
 // this.age = age;  
 // }  
 // let me = new Person('xiao',30);  
 // console.log(me);//报错  
  
 //3. 不能使用 arguments 变量  
 // let fn = () => {  
 // console.log(arguments);  
 // }  
 // fn(1,2,3);//报错  
  
 //4. 箭头函数的简写  
 //1) 省略小括号, 当形参有且只有一个的时候  
 // let add = n => {  
 // return n + n;  
 // }  
 // console.log(add(9));//18  
 //2) 省略花括号, 当代码体只有一条语句的时候, 此时 return 必须省略  
 // 而且语句的执行结果就是函数的返回值  
 let pow = n => n \* n;  
 ***console***.log(pow(8));  
  
 </script>  
</body>  
  
</html>

## 箭头函数的实践与应用场景

·箭头函数适合与 this 无关的回调. 定时器, 数组的方法回调

·箭头函数不适合与 this 有关的回调. 事件回调, 对象的方法

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>箭头函数实践</title>  
 <style>  
 div {  
 width: 200px;  
 height: 200px;  
 background: #58a;  
 }  
 </style>  
</head>  
<body>  
 <div id="ad"></div>  
 <script>  
 //需求-1 点击 div 2s 后颜色变成『粉色』  
 //获取元素  
 let ***ad*** = ***document***.getElementById("ad");  
 ***ad***.addEventListener("click",function () {  
 //保存this的值，这里的this是ad对象  
 // let \_this = this;  
 //定时器  
 // setTimeout(function () {  
 // //修改背景颜色，这里的this是window，会报错  
 // // this.style.background = "pink";  
 // //\_this保存的是之前this的值，不会报错  
 // \_this.style.background = "pink";  
 // },2000);  
 // 定时器  
 setTimeout(()=>{  
 //修改背景颜色，这里的this是声明函数时ad对象  
 this.style.background = "pink";  
 },2000);  
 });  
  
 //需求-2 从数组中返回偶数的元素  
 const ***arr*** = [1,6,9,10,100,25];  
 // //filter过滤数组，里面的函数遍历数组，如果返回true则将值返回给result  
 // const result = arr.filter(function (item) {  
 // if (item % 2 === 0){  
 // return true;  
 // }else {  
 // return false;  
 // }  
 // });  
 // console.log(result);//[6,10,100]  
  
 //filter过滤数组，里面的函数遍历数组，如果返回true则将值返回给result  
 // const result = arr.filter(item=>{  
 // if (item % 2 === 0) {  
 // return true;  
 // } else {  
 // return false;  
 // }  
 // });  
 // console.log(result);//[6,10,100]  
  
 //filter过滤数组，里面的函数遍历数组，如果返回true则将值返回给result  
 const ***result*** = ***arr***.filter(item=> item % 2 === 0);  
 ***console***.log(***result***);//[6,10,100]  
  
 // 箭头函数适合与 this 无关的回调. 定时器, 数组的方法回调  
 // 箭头函数不适合与 this 有关的回调. 事件回调, 对象的方法  
  
 </script>  
</body>  
</html>

# 函数参数的默认值设置

·ES6允许给函数参数赋初始值

·形参初始值 具有默认值的参数, 一般位置要靠后(潜规则)

//1. 形参初始值 具有默认值的参数, 一般位置要靠后(潜规则)  
function add(a,b,c=10) {  
 return a + b + c;  
}  
let ***result*** = add(1,2);  
***console***.log(***result***);

·与解构赋值结合

//2. 与解构赋值结合  
function connect({host="127.0.0.1", username,password, port}){  
 ***console***.log(host);//atguigu.com  
 ***console***.log(username);//root  
 ***console***.log(password);//root  
 ***console***.log(port);//3306  
}  
connect({  
 host: 'atguigu.com',  
 username: 'root',  
 password: 'root',  
 port: 3306  
})

# rest 参数

·ES6 引入 rest 参数，用于获取函数的实参，用来代替 arguments

// ES5 获取实参的方式  
function date(){  
 ***console***.log(arguments);  
}  
date('白芷','阿娇','思慧');  
  
// rest 参数  
function date(...args){  
 ***console***.log(args);//["阿娇", "柏芝", "思慧"]  
}  
date('阿娇','柏芝','思慧');

·rest 参数必须要放到参数最后

// rest 参数必须要放到参数最后  
function fn(a,b,...args){  
 ***console***.log(a);//1  
 ***console***.log(b);//2  
 ***console***.log(args);//[3, 4, 5, 6]  
}  
fn(1,2,3,4,5,6);

·注意：rest 参数非常适合不定个数参数函数的场景

# spread 扩展运算符

·扩展运算符（spread）也是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列，对数组进行解包。

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>扩展运算符</title>  
</head>  
<body>  
 <script>  
 // 『...』 扩展运算符能将『数组』转换为逗号分隔的『参数序列』  
 //声明一个数组 ...  
 const ***tfboys*** = ['易烊千玺','王源','王俊凯'];  
 // => '易烊千玺','王源','王俊凯'  
  
 // 声明一个函数  
 function chunwan(){  
 ***console***.log(arguments);  
 }  
  
 chunwan(***tfboys***);// chunwan(['易烊千玺','王源','王俊凯'])  
 chunwan(...***tfboys***);// chunwan('易烊千玺','王源','王俊凯')  
 </script>  
</body>  
</html>

# 应用

·数组的合并

//1. 数组的合并  
const ***kuaizi*** = ["王太利","肖央"];  
const ***fenghuang*** = ["曾毅","玲花"];  
//合并数组  
// const zuixuanxiaopingguo = kuaizi.concat(fenghuang);  
const ***zuixuanxiaopingguo*** = [...***kuaizi***,...***fenghuang***];  
***console***.log(***zuixuanxiaopingguo***);// ["王太利", "肖央", "曾毅", "玲花"]

·数组的克隆

//2. 数组的克隆  
const ***sanzhihua*** = ["E","G","M"];  
const ***sanyecao*** = [...***sanzhihua***];  
***console***.log(***sanyecao***);//["E", "G", "M"]

·将伪数组转为真正的数组

//3. 将伪数组转为真正的数组  
const ***divs*** = ***document***.querySelectorAll("div");  
const ***divArr*** = [...***divs***];  
***console***.log(***divArr***);

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>扩展运算符应用</title>  
</head>  
<body>  
 <div></div>  
 <div></div>  
 <div></div>  
 <script>  
 //1. 数组的合并  
 const ***kuaizi*** = ["王太利","肖央"];  
 const ***fenghuang*** = ["曾毅","玲花"];  
 //合并数组  
 // const zuixuanxiaopingguo = kuaizi.concat(fenghuang);  
 const ***zuixuanxiaopingguo*** = [...***kuaizi***,...***fenghuang***];  
 ***console***.log(***zuixuanxiaopingguo***);// ["王太利", "肖央", "曾毅", "玲花"]  
  
 //2. 数组的克隆  
 const ***sanzhihua*** = ["E","G","M"];  
 const ***sanyecao*** = [...***sanzhihua***];  
 ***console***.log(***sanyecao***);//["E", "G", "M"]  
  
 //3. 将伪数组转为真正的数组  
 const ***divs*** = ***document***.querySelectorAll("div");  
 const ***divArr*** = [...***divs***];  
 ***console***.log(***divArr***);  
 </script>  
</body>  
</html>

# Symbol

·ES6 引入了一种新的原始数据类型 Symbol，表示独一无二的值。它是

JavaScript 语言的第七种数据类型，是一种类似于字符串的数据类型

·Symbol 的值是唯一的，用来解决命名冲突的问题

·Symbol 值不能与其他数据进行运算

·Symbol 定义 的 对象属 性 不能 使 用 for…in 循 环遍 历 ，但 是可 以 使 用Reflect.ownKeys 来获取对象的所有键名

## Symbol 基本使用

·创建Symbol

- 相同的备注创建的值是不同的

//创建Symbol  
let ***s*** = ***Symbol***();  
***console***.log(***s***, typeof ***s***);//Symbol() "symbol"  
//尚硅谷相当于对数据的备注  
let ***s2*** = ***Symbol***('尚硅谷');  
let ***s3*** = ***Symbol***('尚硅谷');  
***console***.log(***s2***);//Symbol(尚硅谷)  
***console***.log(***s3***);//Symbol(尚硅谷)  
***console***.log(***s2*** === ***s3***);//false

·Symbol.for 创建

- 相同的备注创建的值是相同的

//Symbol.for 创建  
let ***s4*** = ***Symbol***.for('尚硅谷');  
let ***s5*** = ***Symbol***.for('尚硅谷');  
***console***.log(***s4*** === ***s5***);//true

·不能与其他数据进行运算

//不能与其他数据进行运算  
let ***result*** = ***s*** + 100;//报错  
let ***result*** = ***s*** > 100;//报错  
let ***result*** = ***s*** > "100";//报错  
let ***result*** = ***s*** + ***s***;//报错

·USONB you are so niubility

- u undefined

- s string symbol

- o object

- n null number

- b boolean

·注:遇到唯一性的场景时要想到 Symbol

## 应用（还是不知道干嘛用的）

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Symbol 创建对象属性</title>  
</head>  
<body>  
 <script>  
 //向对象中添加方法 up down  
 // let game = {  
 // name:'俄罗斯方块',  
 // up: function(){},  
 // down: function(){}  
 // };  
 //  
 // // 声明一个对象  
 // let methods = {  
 // up: Symbol(),  
 // down: Symbol()  
 // };  
 //  
 // game[methods.up] = function(){  
 // console.log("我可以改变形状");  
 // }  
 //  
 // game[methods.down] = function(){  
 // console.log("我可以快速下降!!");  
 // }  
 //  
 // console.log(game);  
   
   
 let ***youxi*** = {  
 name:"狼人杀",  
 [***Symbol***('say')]: function(){  
 ***console***.log("我可以发言")  
 },  
 [***Symbol***('zibao')]: function(){  
 ***console***.log('我可以自爆');  
 }  
 }  
  
 ***console***.log(***youxi***)  
 </script>  
</body>  
</html>

## Symbol 内置值

·除了定义自己使用的 Symbol 值以外，ES6 还提供了 11 个内置的 Symbol 值，指向语言内部使用的方法。可以称这些方法为魔术方法，因为它们会在特定的场

景下自动执行。



·Symbol.hasInstance

- 当其他对象使用 instanceof 运算符时调用的方法

class Person{  
 static [***Symbol***.hasInstance](param){  
 ***console***.log(param);  
 ***console***.log("我被用来检测类型了");  
 return false;  
 }  
}  
  
let ***o*** = {};  
//将o传递给[Symbol.hasInstance](param)方法，进行类型检测  
***console***.log(***o*** instanceof Person);

·Symbol.isConcatSpreadable

- 表示该对象用于 Array.prototype.concat()时，是否可以展开

const ***arr*** = [1,2,3];  
const ***arr2*** = [4,5,6];  
***arr2***[***Symbol***.isConcatSpreadable] = false;//不可展开  
***console***.log(***arr***.concat(***arr2***));//[1, 2, 3, Array(3)]  
***arr2***[***Symbol***.isConcatSpreadable] = true;//可展开  
***console***.log(***arr***.concat(***arr2***));//[1, 2, 3, 4, 5, 6]

# 迭代器

·遍历器（Iterator）就是一种机制。它是一种接口，为各种不同的数据结构提

供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作

·ES6 创造了一种新的遍历命令 for...of 循环，Iterator 接口主要供 for...of 消费

·原生具备 iterator 接口的数据(可用 for of 遍历)

- Array

- Arguments

- Set

- Map

- String

- TypedArray

- NodeList

·工作原理

- 创建一个指针对象，指向当前数据结构的起始位置

- 第一次调用对象的 next 方法，指针自动指向数据结构的第一个成员

- 接下来不断调用 next 方法，指针一直往后移动，直到指向最后一个成员

- 每调用 next 方法返回一个包含 value 和 done 属性的对象

·注: 需要自定义遍历数据的时候，要想到迭代器。

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>迭代器</title>  
</head>  
<body>  
 <script>  
 //声明一个数组  
 const ***xiyou*** = ['唐僧','孙悟空','猪八戒','沙僧'];  
 //使用 for...of 遍历数组  
 for(let v of ***xiyou***){  
 ***console***.log(v);  
 }  
 //获取iterator对象  
 let ***iterator*** = ***xiyou***[***Symbol***.iterator]();  
 //调用对象的next方法  
 ***console***.log(***iterator***.next());//{value: "唐僧", done: false}  
 ***console***.log(***iterator***.next());//{value: "孙悟空", done: false}  
 ***console***.log(***iterator***.next());//{value: "猪八戒", done: false}  
 ***console***.log(***iterator***.next());//{value: "沙僧", done: false}  
 ***console***.log(***iterator***.next());//{value: undefined, done: true}  
 </script>  
</body>  
</html>

## 迭代器的应用

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>自定义遍历数据</title>  
</head>  
  
<body>  
 <script>  
 //声明一个对象  
 const ***banji*** = {  
 name:"终极一班",  
 stus:[  
 "小明",  
 "小宁",  
 "小天",  
 "neki"  
 ],  
 //自定义迭代器  
 [***Symbol***.iterator](){  
 //索引变量  
 let index = 0;  
 //保存当前对象  
 let \_this = this;  
 return {  
 next:function () {  
 if (index < \_this.stus.length){//如果当前下标小于要遍历的长度  
 //获取当前下标的值  
 const result = {  
 value:\_this.stus[index],  
 done:false  
 };  
 //下标自增  
 index++;  
 //返回结果  
 return result;  
 }else {//如果当前下标等于要遍历的长度  
 return {  
 value: undefined,  
 done: true  
 }  
 }  
 }  
 }  
 }  
 }  
  
 //遍历banji对象  
 for (let v of ***banji***){  
 ***console***.log(v);  
 }  
 </script>  
</body>  
  
</html>

# 生成器函数

·生成器是一个特殊的函数

·生成器函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同

·\* 的位置没有限制

·生成器函数返回的结果是迭代器对象，调用迭代器对象的 next 方法可以得到

yield 语句后的值

·yield 相当于函数的暂停标记，也可以认为是函数的分隔符，每调用一次 next

方法，执行一段代码

·next 方法可以传递实参，作为 yield 语句的返回值

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>生成器</title>  
</head>  
<body>  
 <script>   
 //生成器其实就是一个特殊的函数  
 //异步编程 纯回调函数 node fs ajax mongodb  
 //yield是函数代码的分隔符，可以把函数代码切分成几块  
 //以下把函数代码分成4块  
 function \* gen(){  
 ***console***.log(111);  
 yield '一只没有耳朵';  
 ***console***.log(222);  
 yield "一只没有尾巴";  
 ***console***.log(333);  
 yield '真奇怪';  
 ***console***.log(444);  
 }  
 //获取迭代器对象  
 let ***iterator*** = gen();  
 //iterator.next()执行的是每一份代码块  
 ***iterator***.next();//执行console.log(111);  
 ***iterator***.next();//执行console.log(222);  
 //先执行iterator.next()，再打印迭代器返回的对象  
 ***console***.log(***iterator***.next());//先执行console.log(333)，再打印{value: "真奇怪", done: false}  
 ***console***.log(***iterator***.next());//先执行console.log(444);，再打印{value: undefined, done: true}  
  
 // 遍历  
 for(let v of gen()){  
 ***console***.log(v);//执行代码块并打印迭代器对象的value值  
 }  
 //依次打印  
 //111  
 //一只没有耳朵  
 //222  
 //一只没有尾巴  
 //333  
 //真奇怪  
 //444  
 </script>  
</body>  
</html>

## 生成器函数的参数传递

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>生成器函数参数</title>  
</head>  
<body>  
 <script>  
 function \* gen(arg){  
 ***console***.log(arg);  
 let one = yield 111;  
 ***console***.log(one);  
 let two = yield 222;  
 ***console***.log(two);  
 let three = yield 333;  
 ***console***.log(three);  
 }  
  
 //执行获取迭代器对象  
 let ***iterator*** = gen('AAA');  
 ***console***.log(***iterator***.next());//AAA {value: 111, done: false}  
 //next方法可以传入实参  
 ***console***.log(***iterator***.next('BBB'));//BBB {value: 222, done: false}  
 ***console***.log(***iterator***.next('CCC'));//CCC {value: 333, done: false}  
 ***console***.log(***iterator***.next('DDD'));//DDD {value: undefined, done: true}  
 </script>  
</body>  
</html>

## 生成器函数实例

·异步编程

- 文件操作

- 网络操作(ajax, request)

- 数据库操作

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>生成器函数实例</title>  
</head>  
  
<body>  
 <script>  
 // 异步编程 文件操作 网络操作(ajax, request) 数据库操作  
 // 1s 后控制台输出 111 2s后输出 222 3s后输出 333   
 // 回调地狱  
 // setTimeout(() => {  
 // console.log(111);  
 // setTimeout(() => {  
 // console.log(222);  
 // setTimeout(() => {  
 // console.log(333);  
 // }, 3000);  
 // }, 2000);  
 // }, 1000);  
  
 //声明3个函数  
 //第一个函数1s 后控制台输出 111  
 function one(){  
 setTimeout(()=>{  
 ***console***.log(111);  
 //调用two()  
 ***iterator***.next();  
 },1000)  
 }  
  
 //2s后输出 222  
 function two(){  
 setTimeout(()=>{  
 ***console***.log(222);  
 //three()  
 ***iterator***.next();  
 },2000)  
 }  
  
 //3s后输出 333  
 function three(){  
 setTimeout(()=>{  
 ***console***.log(333);  
 //调用下一个函数，下一个迭代器对象为{value: undefined, done: true}  
 // console.log(iterator.next());  
 ***iterator***.next();  
 },3000)  
 }  
  
 //将三个函数依次放入生成器函数中  
 function \* gen(){  
 yield one();  
 yield two();  
 yield three();  
 }  
  
 //调用生成器函数  
 let ***iterator*** = gen();  
 ***iterator***.next();//调用one()  
 </script>  
</body>  
  
</html>

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>生成器函数</title>  
</head>  
<body>  
 <script>  
 //模拟获取 用户数据 订单数据 商品数据  
 //有先后顺序  
 //模拟获取用户数据  
 function getUsers(){  
 setTimeout(()=>{  
 let data = '用户数据';  
 //调用getOrders(users)方法, 并且将数据传入  
 ***iterator***.next(data);  
 }, 1000);  
 }  
  
 //模拟获取订单数据  
 function getOrders(users){  
 setTimeout(()=>{  
 //获取到用户数据  
 ***console***.log(users);  
 let data = '订单数据';  
 //调用getGoods(orders)方法, 并且将数据传入  
 ***iterator***.next(data);  
 }, 1000)  
 }  
  
 //模拟获取商品数据  
 function getGoods(orders){  
 setTimeout(()=>{  
 //获取到订单数据  
 ***console***.log(orders);  
 let data = '商品数据';  
 //调用next方法,并且将数据传入，此时没有方法，只用于传递数据  
 ***iterator***.next(data);  
 }, 1000)  
 }  
  
 function \* gen(){  
 let users = yield getUsers();  
 //获取到用户数据  
 // console.log(users);  
 let orders = yield getOrders(users);  
 //获取到订单数据  
 // console.log(orders);  
 let goods = yield getGoods(orders);  
 //获取到商品数据  
 ***console***.log(goods);  
 }  
  
 //调用生成器函数  
 let ***iterator*** = gen();  
 //调用getUsers()  
 ***iterator***.next();  
 </script>  
</body>  
</html>

# Promise

·Promise 是 ES6 引入的异步编程的新解决方案。语法上 Promise 是一个构造函数，用来封装异步操作并可以获取其成功或失败的结果

·Promise 构造函数: Promise (excutor) {}

·Promise.prototype.then 方法

·Promise.prototype.catch 方法

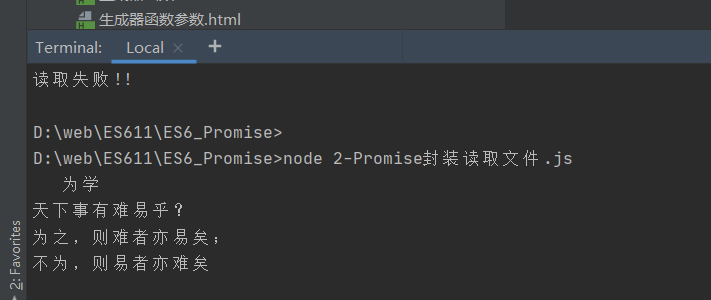
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Promise基本语法</title>  
</head>  
<body>  
 <script>  
 //实例化 Promise 对象  
 //resolve()和reject()可以改变Promise对象的状态  
 const ***p*** = new ***Promise***(function(resolve, reject){  
 setTimeout(function(){  
 //  
 // let data = '获取到数据库中的用户数据';  
 // resolve  
 //调用resolve(data)后会将Promise对象的状态变为成功，将参数值传给then的第一个函数，并执行  
 // resolve(data);  
  
 let err = '数据读取失败';  
 //调用reject(err)后会将Promise对象的状态变为失败，将参数值传给then的第二个函数，并执行  
 reject(err);  
 }, 1000);  
 });  
  
 //调用 promise 对象的 then 方法  
 ***p***.then(function(value){//成功调用的方法  
 ***console***.log(value);  
 }, function(reason){//失败调用的方法  
 //打印错误原因  
 ***console***.error(reason);  
 })  
 </script>  
</body>  
</html>

## Promise封装读取文件

·编写js文件

// //引入fs模块  
const fs = require("fs");  
// //读取文件  
// //err如果出错是错误对象,如果失败是none  
// //data表示读取结果  
// fs.readFile("./resources/为学.md",(err,data)=>{  
// //如果失败,则抛出错误  
// if (err) throw err;  
// //如果没有出错,则输出内容  
// console.log(data.toString());  
// });  
  
//使用Promise封装  
const p = new ***Promise***(function (resolve,reject) {  
 fs.readFile("./resources/为学.md",(err,data)=>{  
 //如果失败,则改变promise的状态  
 if(err) reject(err);  
 //如果没有出错,则输出内容  
 resolve(data);  
 });  
});  
//处理成功和失败  
p.then(function (value) {  
 //打印结果  
 ***console***.log(value.toString());  
},function (reason) {  
 ***console***.log("读取失败!!");  
});

·使用node命令运行文件



## Promise封装AJAX请求

·接口地址: <https://api.apiopen.top/getJoke>

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>发送 AJAX 请求</title>  
</head>  
  
<body>  
 <script>  
 // 接口地址: https://api.apiopen.top/getJoke  
 //promise封装AJAX结果  
 const ***p*** = new ***Promise***((resolve, reject) => {  
 //创建对象  
 const xhr = new ***XMLHttpRequest***();  
 //初始化  
 xhr.open("GET", "https://api.apiopen.top/getJoke");  
 //发送  
 xhr.send();  
 //绑定事件,处理响应结果  
 xhr.onreadystatechange = function () {  
 //判断  
 if (xhr.readyState === 4) {  
 //判断响应状态码  
 //200-300表示成功  
 if (xhr.status >= 200 && xhr.status <= 300) {  
 //如果成功  
 resolve(xhr.response);  
 } else {//如果失败  
 reject(xhr.status);  
 }  
 }  
 }  
 });  
  
 //指定回调  
 ***p***.then(function (value) {  
 ***console***.log(value);  
 },function (reason) {  
 ***console***.log(reason);  
 });  
 </script>  
</body>  
  
</html>

## Promise.prototype.then 方法

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Promise.prototype.then</title>  
</head>  
<body>  
 <script>  
 //创建 promise 对象  
 const ***p*** = new ***Promise***((resolve, reject)=>{  
 setTimeout(()=>{  
 resolve('用户数据');  
 // reject('出错啦');  
 }, 1000)  
 });  
  
 //调用then方法,成功则调用第一个参数方法  
 //失败则调用第二个参数方法  
 //then方法的返回结果是promise对象,对象的 状态由回调函数的执行结果决定  
 // const result = p.then(value => {  
 // console.log(value);  
 // //如果回调函数中返回的结果是 非 promise 类型的属性, 状态为成功, 返回值为对象的成功的值  
 // //非 promise 类型的属性  
 // // return "哈哈哈哈";  
 // //promise 对象  
 // // return new Promise((resolve, reject) => {  
 // // // resolve("ok");  
 // // reject("error");   
 // // });  
 // //抛出错误  
 // // throw new Error("出错啦");  
 // throw "出错啦";  
 // },reason => {  
 // console.log(reason);  
 // });  
   
 //链式调用  
 ***p***.then(value => {  
   
 },reason => {  
   
 }).then(value => {  
   
 },reason => {  
   
 });  
  
 ***console***.log(***result***);  
  
 //调用 then 方法 then方法的返回结果是 Promise 对象, 对象状态由回调函数的执行结果决定  
 //1. 如果回调函数中返回的结果是 非 promise 类型的属性, 状态为成功, 返回值为对象的成功的值  
  
 // const result = p.then(value => {  
 // console.log(value);  
 // //1. 非 promise 类型的属性  
 // // return 'iloveyou';  
 // //2. 是 promise 对象  
 // // return new Promise((resolve, reject)=>{  
 // // // resolve('ok');  
 // // reject('error');  
 // // });  
 // //3. 抛出错误  
 // // throw new Error('出错啦!');  
 // throw '出错啦!';  
 // }, reason=>{  
 // console.warn(reason);  
 // });  
  
 //链式调用  
 // p.then(value=>{  
 //  
 // }).then(value=>{  
 //  
 // });  
 </script>  
</body>  
</html>

# Promise实践练习-多个文件内容读取

//先读取为学,再读取插秧诗,然后读取观书有感  
//引入fs模块  
const fs = require("fs");  
//原始读取实现  
// fs.readFile("./resources/为学.md",(err,data1)=>{  
// fs.readFile("./resources/插秧诗.md",(err,data2)=>{  
// fs.readFile("./resources/观书有感.md",(err,data3)=>{  
// let result = data1 + '\r\n' + data2 + '\r\n' + data3;  
// console.log(result);  
// });  
// });  
// });  
  
//使用promise实现  
const p = new ***Promise***((resolve, reject) => {  
 fs.readFile("./resources/为学.md",(err,data)=>{  
 resolve(data);  
 });  
});  
  
  
p.then(value => {  
 //返回一个promise对象  
 return new ***Promise***((resolve, reject) => {  
 fs.readFile("./resources/插秧诗.md",(err,data)=>{  
 resolve([value,data]);  
 });  
 });  
}).then(value => {  
 //返回一个promise对象  
 return new ***Promise***((resolve, reject) => {  
 fs.readFile("./resources/观书有感.md",(err,data)=>{  
 //压入  
 value.push(data);  
 resolve(value);  
 });  
 });  
}).then(value => {  
 ***console***.log(value.join('\r\n'));  
})

## Promise.prototype.catch 方法

·Promise对象失败的回调

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>catch方法</title>  
</head>  
<body>  
 <script>  
 const ***p*** = new ***Promise***((resolve, reject) => {  
 setTimeout(()=>{  
 //设置p对象的状态为失败,传入失败的值  
 reject("出错啦");  
 },1000);  
 });  
  
 //成功和失败的回调  
 // p.then(function (value) {  
 //  
 // },function (reason) {  
 // console.error(reason);  
 // })  
  
 //失败的回调  
 ***p***.catch(function (reason) {  
 ***console***.warn(reason);  
 });  
 </script>  
</body>  
</html>

# Set

·ES6 提供了新的数据结构 Set（集合）。它类似于数组，但成员的值都是唯

一的，如果不唯一会自动去重。集合实现了 iterator 接口，所以可以使用『扩展运算符』和『for…of…』进行遍历，集合的属性和方法

·size

- 返回集合的元素个数

//元素个数  
***console***.log(***s2***.size);

·add

- 增加一个新元素，返回当前集合

//添加新的元素  
***s2***.add('喜事儿');

·delete

- 删除元素，返回 boolean 值

//删除元素  
***s2***.delete('坏事儿');

·has

- 检测集合中是否包含某个元素，返回 boolean 值

//检测是否有"糟心事"  
***console***.log(***s2***.has('糟心事'));

·clear

- 清空集合，返回 undefined

//清空  
***s2***.clear();

·for…of…遍历

//遍历集合  
for(let v of ***s2***){  
 ***console***.log(v);  
}

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>集合</title>  
</head>  
<body>  
 <script>  
 //声明一个 set  
 let ***s*** = new ***Set***();  
 // console.log(s,typeof s);//Set(0) "object"  
 //会自动去重  
 let ***s2*** = new ***Set***(['大事儿','小事儿','好事儿','坏事儿','小事儿']);  
 // console.log(s2);//Set(4) {"大事儿", "小事儿", "好事儿", "坏事儿"}  
 //元素个数  
 ***console***.log(***s2***.size);  
 //添加新的元素  
 ***s2***.add('喜事儿');  
 //删除元素  
 ***s2***.delete('坏事儿');  
 //检测是否有"糟心事"  
 ***console***.log(***s2***.has('糟心事'));  
 //清空  
 // s2.clear();  
 // console.log(s2);  
  
 //遍历集合  
 for(let v of ***s2***){  
 ***console***.log(v);  
 }  
   
 </script>  
</body>  
</html>

## 集合实践

·数组去重

let ***arr*** = [1,2,3,4,5,4,3,2,1];  
//数组去重,并扩展  
let ***result*** = [...new ***Set***(***arr***)];  
***console***.log(***result***);// [1, 2, 3, 4, 5]

·取交集

//取交集  
let ***arr*** = [1,2,3,4,5,4,3,2,1];  
let ***arr2*** = [4,5,6,5,6];  
//arr先去重,然后扩展,再过滤  
let ***result*** = [...new ***Set***(***arr***)].filter(items =>{  
 //将arr2去除  
 let s2 = new ***Set***(***arr2***);//4,5,6  
 //判断这个数组的值在不在arr2里边,如果在就返回true  
 // if (s2.has(items)){  
 // return true;  
 // }else {  
 // return false;  
 // }  
 return s2.has(items);  
});  
***console***.log(***result***);//[4, 5]

·并集

//并集  
let ***arr*** = [1,2,3,4,5,4,3,2,1];  
let ***arr2*** = [4,5,6,5,6];  
//合并数组并去重,然后转化为数组  
let ***union*** = [...new ***Set***([...***arr***,...***arr2***])];  
***console***.log(***union***);//[1, 2, 3, 4, 5, 6]

·差集

//差集就是交集的逆运算,arr-arr2  
let ***arr*** = [1,2,3,4,5,4,3,2,1];  
let ***arr2*** = [4,5,6,5,6];  
let ***result*** = [...new ***Set***(***arr***)].filter(items =>{  
 return !(new ***Set***(***arr2***).has(items));  
});  
***console***.log(***result***);//[1, 2, 3]

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Set 实践</title>  
</head>  
<body>  
 <script>  
 //数组去重,并扩展  
 // let arr = [1,2,3,4,5,4,3,2,1];  
 // let result = [...new Set(arr)];  
 // console.log(result);// [1, 2, 3, 4, 5]  
 //取交集  
 // let arr = [1,2,3,4,5,4,3,2,1];  
 // let arr2 = [4,5,6,5,6];  
 // //arr先去重,然后扩展,再过滤  
 // let result = [...new Set(arr)].filter(items =>{  
 // //将arr2去除  
 // let s2 = new Set(arr2);//4,5,6  
 // //判断这个数组的值在不在arr2里边,如果在就返回true  
 // // if (s2.has(items)){  
 // // return true;  
 // // }else {  
 // // return false;  
 // // }  
 // return s2.has(items);  
 // });  
 // console.log(result);//[4, 5]  
  
 //并集  
 // let arr = [1,2,3,4,5,4,3,2,1];  
 // let arr2 = [4,5,6,5,6];  
 // //合并数组并去重,然后转化为数组  
 // let union = [...new Set([...arr,...arr2])];  
 // console.log(union);//[1, 2, 3, 4, 5, 6]  
  
 //差集就是交集的逆运算,arr-arr2  
 let ***arr*** = [1,2,3,4,5,4,3,2,1];  
 let ***arr2*** = [4,5,6,5,6];  
 let ***result*** = [...new ***Set***(***arr***)].filter(items =>{  
 return !(new ***Set***(***arr2***).has(items));  
 });  
 ***console***.log(***result***);//[1, 2, 3]  
 </script>  
</body>  
  
</html>

# Map

·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）都可以当作键。Map 也实现了iterator 接口，所以可以使用『扩展运算符』和『for…of…』进行遍历。Map 的属性和方法

·size

- 返回 Map 的元素个数

//size  
***console***.log(***m***.size);

·set

- 增加一个新元素，返回当前 Map

//删除  
***m***.delete('name');

·get

- 返回键名对象的键值

//获取  
***console***.log(***m***.get('change'));  
***console***.log(***m***.get(***key***));

·has

- 检测 Map 中是否包含某个元素，返回 boolean 值

***console***.log("检测:"+***m***.has("change"));

·clear

- 清空集合，返回 undefined

//清空  
***m***.clear();

·for…of…遍历

//遍历  
for(let v of ***m***){  
 ***console***.log(v);  
}

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Map</title>  
</head>  
<body>  
 <script>  
 //声明 Map  
 let ***m*** = new ***Map***();  
  
 //添加元素  
 ***m***.set('name','尚硅谷');  
 ***m***.set('change', function(){  
 ***console***.log("我们可以改变你!!");  
 });  
 let ***key*** = {  
 school : 'ATGUIGU'  
 };  
 ***m***.set(***key***, ['北京','上海','深圳']);  
  
 //size  
 ***console***.log(***m***.size);  
  
 //删除  
 ***m***.delete('name');  
  
 //获取  
 ***console***.log(***m***.get('change'));  
 ***console***.log(***m***.get(***key***));  
  
 //检测  
 ***console***.log("检测:"+***m***.has("change"));  
  
 //清空  
 ***m***.clear();  
  
 //遍历  
 for(let v of ***m***){  
 ***console***.log(v);  
 }  
  
 // console.log(m);  
  
 </script>  
</body>  
</html>

# class 类

·ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过 class 关键字，可以定义类。基本上，ES6 的 class 可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的 class 写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已

·原始构造函数

//原始构造函数的方法  
//手机  
function Phone(brand, price){  
 this.brand = brand;  
 this.price = price;  
}  
  
//添加方法  
Phone.prototype.call = function(){  
 ***console***.log("我可以打电话!!");  
}  
  
//实例化对象  
let ***Huawei*** = new Phone('华为', 5999);  
***Huawei***.call();//我可以打电话!!  
***console***.log(***Huawei***);//Huawei对象

·class

- 声明类

//class  
class Shouji{  
 //构造方法 名字不能修改,必须叫constructor  
 constructor(brand, price){  
 this.brand = brand;  
 this.price = price;  
 }  
  
 //方法必须使用该语法, 不能使用 ES5 的对象完整形式  
 call(){  
 ***console***.log("我也可以打电话!!");  
 }  
}  
  
let ***onePlus*** = new Shouji("1+", 1999);  
***onePlus***.call();//我也可以打电话!!  
***console***.log(***onePlus***);//onePlus对象

·constructor

- 定义构造函数初始化

- 构造函数只能有一个

//构造方法 名字不能修改,必须叫constructor  
constructor(brand, price){  
 this.brand = brand;  
 this.price = price;  
}

·extends

- 继承父类

·super

- 调用父级构造方法

·static

- 定义静态方法和属性

·父类方法可以重写

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>类声明</title>  
</head>  
<body>  
 <script>  
 //原始构造函数的方法  
 //手机  
 function Phone(brand, price){  
 this.brand = brand;  
 this.price = price;  
 }  
  
 //添加方法  
 Phone.prototype.call = function(){  
 ***console***.log("我可以打电话!!");  
 }  
  
 //实例化对象  
 let ***Huawei*** = new Phone('华为', 5999);  
 ***Huawei***.call();//我可以打电话!!  
 ***console***.log(***Huawei***);//Huawei对象  
  
 //class  
 class Shouji{  
 //构造方法 名字不能修改,必须叫constructor  
 constructor(brand, price){  
 this.brand = brand;  
 this.price = price;  
 }  
  
 //方法必须使用该语法, 不能使用 ES5 的对象完整形式  
 call(){  
 ***console***.log("我也可以打电话!!");  
 }  
 }  
  
 let ***onePlus*** = new Shouji("1+", 1999);  
 ***onePlus***.call();//我也可以打电话!!  
 ***console***.log(***onePlus***);//onePlus对象  
 </script>  
</body>  
</html>

## class静态成员

·属性和方法是属于函数的,而不是属于实例对象的,这样的属性和方法我们称为静态属性和方法

·原始

function Phone(){  
  
}  
  
// 往Phone函数添加属性和方法,这样的属性和方法是属于函数的,而不是属于实例对象的,这样的属性和方法我们称为静态属性和方法  
Phone.name = '手机';  
Phone.change = function(){  
 ***console***.log("我可以改变世界");  
}  
//要在原型对象上添加属性和方法才可以  
Phone.prototype.size = '5.5inch';  
  
let ***nokia*** = new Phone();  
  
***console***.log(***nokia***.name);//undefined  
// nokia.change();//报错  
***console***.log(***nokia***.size);//5.5inch

·class

class Phone{  
 //静态属性  
 static *name* = '手机';  
 static *change*(){  
 ***console***.log("我可以改变世界");  
 }  
}  
  
let ***nokia*** = new Phone();  
***console***.log(***nokia***.name);//undefined  
***console***.log(Phone.*name*);//手机

## ES5构造函数继承

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>对象继承</title>  
</head>  
<body>  
 <script>  
 //原始(ES5)构造函数实现继承  
 //手机  
 function Phone(brand, price){  
 this.brand = brand;  
 this.price = price;  
 }  
  
 Phone.prototype.call = function(){  
 ***console***.log("我可以打电话");  
 }  
  
 //智能手机  
 function SmartPhone(brand, price, color, size){  
 //调用父类的构造函数  
 Phone.call(this, brand, price);  
 this.color = color;  
 this.size = size;  
 }  
  
 //设置子级构造函数的原型  
 SmartPhone.prototype = new Phone;  
 SmartPhone.prototype.constructor = SmartPhone;  
  
 //声明子类的方法  
 SmartPhone.prototype.photo = function(){  
 ***console***.log("我可以拍照")  
 }  
  
 SmartPhone.prototype.playGame = function(){  
 ***console***.log("我可以玩游戏");  
 }  
  
 const ***chuizi*** = new SmartPhone('锤子',2499,'黑色','5.5inch');  
  
 ***console***.log(***chuizi***);  
  
 </script>  
</body>  
</html>

## class类继承

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>类继承-2</title>  
</head>  
<body>  
 <script>  
 //声明父类  
 class Phone{  
 //构造方法  
 constructor(brand, price){  
 this.brand = brand;  
 this.price = price;  
 }  
 //父类的成员属性  
 call(){  
 ***console***.log("我可以打电话!!");  
 }  
 }  
  
 //SmartPhone继承父类的属性  
 class SmartPhone extends Phone {  
 //构造方法  
 constructor(brand, price, color, size){  
 //调用父类的构造方法初始化  
 super(brand, price);// Phone.call(this, brand, price)  
 this.color = color;  
 this.size = size;  
 }  
  
 //为子类添加特有的方法  
 photo(){  
 ***console***.log("拍照");  
 }  
  
 playGame(){  
 ***console***.log("玩游戏");  
 }  
  
 //重写父类的方法  
 call(){  
 //普通的成员里面不能调用super()函数  
 // super();//报错  
 ***console***.log('我可以进行视频通话');  
 }  
 }  
  
 const ***xiaomi*** = new SmartPhone('小米',799,'黑色','4.7inch');  
 // console.log(xiaomi);  
 ***xiaomi***.call();//我可以进行视频通话  
 ***xiaomi***.photo();//拍照  
 ***xiaomi***.playGame();//玩游戏  
 </script>  
</body>  
</html>

## class中的getter和setter设置

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>get 和 set</title>  
</head>  
<body>  
 <script>  
 // get 和 set   
 class Phone{  
 //构造函数初始化价格  
 constructor(price) {  
 this.price = price;  
 }  
 //get方法获取价格  
 get getPrice(){  
 return this.price;  
 }  
 //set方法修改价格  
 set setPrice(newVal){  
 this.price = newVal;  
 }  
 }  
 //实例化对象  
 let ***s*** = new Phone("1999");  
 //调用get方法  
 ***console***.log(***s***.getPrice);//1999  
 //调用set方法修改价格  
 ***s***.setPrice = 'free';  
 //调用get方法  
 ***console***.log(***s***.getPrice);//free  
 </script>  
</body>  
</html>

# 数值扩展

·Number.EPSILON

- JavaScript 表示的最小精度

- EPSILON 属性的值接近于 2.2204460492503130808472633361816E-16

//0. Number.EPSILON 是 JavaScript 表示的最小精度  
//EPSILON 属性的值接近于 2.2204460492503130808472633361816E-16  
//用于浮点数精确运算，如果两值之差小于EPSILON，则说明两数相等  
***console***.log(0.1 + 0.2);//0.30000000000000004  
***console***.log(0.1 + 0.2 === 0.3);//false  
//判断数值是否相等的数  
function equal(a, b){  
 //Math.abs(a-b)取ab差的绝对值  
 // if(Math.abs(a-b) < Number.EPSILON){  
 // return true;  
 // }else{  
 // return false;  
 // }  
 return ***Math***.abs(a-b) < ***Number***.EPSILON;  
}  
***console***.log(equal(0.1 + 0.2, 0.3));//true

·二进制和八进制

- ES6 提供了二进制和八进制数值的新的写法，分别用前缀 0b 和 0o 表示。

// 1. 二进制和八进制  
//二进制数  
let ***b*** = 0b1010;  
***console***.log(***b***);//10  
//八进制数  
let ***o*** = 0o777;  
***console***.log(***o***);//511  
//十进制数  
let ***d*** = 100;  
***console***.log(***d***);//100  
//16进制数  
let ***x*** = 0xff;  
***console***.log(***x***);//255

·Number.isFinite

- 检测一个数值是否为有限数

//2. Number.isFinite 检测一个数值是否为有限数  
***console***.log(***Number***.isFinite(100));//true  
***console***.log(***Number***.isFinite(100/0));//false  
***console***.log(100/3);//33.333333333333336  
***console***.log(***Number***.isFinite(100/3));//true  
***console***.log(***Number***.isFinite(***Infinity***));//false

·Number.isNaN

- 检测一个数值是否为 NaN

//3. Number.isNaN 检测一个数值是否为 NaN   
***console***.log(***Number***.isNaN(123)); //false  
***console***.log("sd" \* 12);//NaN  
***console***.log(***Number***.isNaN("sd" \* 12));//true

·Number.parseInt

- 字符串转整数

// 4. Number.parseInt 字符串转整数  
***console***.log(***Number***.parseInt('5211314.123love'));//5211314

·Number.parseFloat

- 字符串转浮点数

//Number.parseFloat 字符串转浮点数  
***console***.log(***Number***.parseFloat('3.1415926神奇'));//3.1415926

·Number.isInteger

- 判断一个数是否为整数

//5. Number.isInteger 判断一个数是否为整数  
***console***.log(***Number***.isInteger(5));//true  
***console***.log(***Number***.isInteger(2.5));//false

·Math.trunc

- 将数字的小数部分抹掉

//6. Math.trunc 将数字的小数部分抹掉   
***console***.log(***Math***.trunc(3.642));//3

·Math.sign

- 判断一个数到底为正数 负数 还是零

- 正数返回1

- 0返回0

- 负数返回-1

//7. Math.sign 判断一个数到底为正数 负数 还是零  
***console***.log(***Math***.sign(100));  
***console***.log(***Math***.sign(0));  
***console***.log(***Math***.sign(-20000));

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>数值扩展</title>  
</head>  
<body>  
 <script>  
 //0. Number.EPSILON 是 JavaScript 表示的最小精度  
 //EPSILON 属性的值接近于 2.2204460492503130808472633361816E-16  
 //用于浮点数精确运算，如果两值之差小于EPSILON，则说明两数相等  
 // console.log(0.1 + 0.2);//0.30000000000000004  
 // console.log(0.1 + 0.2 === 0.3);//false  
 // //判断数值是否相等的数  
 // function equal(a, b){  
 // //Math.abs(a-b)取ab差的绝对值  
 // // if(Math.abs(a-b) < Number.EPSILON){  
 // // return true;  
 // // }else{  
 // // return false;  
 // // }  
 // return Math.abs(a-b) < Number.EPSILON;  
 // }  
 // console.log(equal(0.1 + 0.2, 0.3));//true  
  
 // // 1. 二进制和八进制  
 // //二进制数  
 // let b = 0b1010;  
 // console.log(b);//10  
 // //八进制数  
 // let o = 0o777;  
 // console.log(o);//511  
 // //十进制数  
 // let d = 100;  
 // console.log(d);//100  
 // //16进制数  
 // let x = 0xff;  
 // console.log(x);//255  
  
 //2. Number.isFinite 检测一个数值是否为有限数  
 // console.log(Number.isFinite(100));//true  
 // console.log(Number.isFinite(100/0));//false  
 // console.log(100/3);//33.333333333333336  
 // console.log(Number.isFinite(100/3));//true  
 // console.log(Number.isFinite(Infinity));//false  
   
 //3. Number.isNaN 检测一个数值是否为 NaN   
 // console.log(Number.isNaN(123)); //false  
 // console.log("sd" \* 12);//NaN  
 // console.log(Number.isNaN("sd" \* 12));//true  
  
 // 4. Number.parseInt 字符串转整数  
 // console.log(Number.parseInt('5211314.123love'));//5211314  
 // //Number.parseFloat 字符串转浮点数  
 // console.log(Number.parseFloat('3.1415926神奇'));//3.1415926  
  
 // //5. Number.isInteger 判断一个数是否为整数  
 // console.log(Number.isInteger(5));//true  
 // console.log(Number.isInteger(2.5));//false  
  
 //6. Math.trunc 将数字的小数部分抹掉   
 // console.log(Math.trunc(3.642));//3  
  
 //7. Math.sign 判断一个数到底为正数 负数 还是零  
 ***console***.log(***Math***.sign(100));  
 ***console***.log(***Math***.sign(0));  
 ***console***.log(***Math***.sign(-20000));  
  
 </script>  
</body>  
</html>

# 对象扩展

·Object.is

- 判断两个值是否完全相等

- 类似于===

//1. Object.is 判断两个值是否完全相等,类似于===  
***console***.log(***Object***.is(0.1+0.2, 0.3));//false  
***console***.log(***Object***.is(120, 120));//true  
***console***.log(***Object***.is("123qwe", "123qwe"));//true  
***console***.log(***Object***.is(***NaN***, ***NaN***));//true  
//NaN与任何值===都为false  
***console***.log(***NaN*** === ***NaN***);//false

·Object.assign

- 对象的合并

//2. Object.assign 对象的合并  
const ***config1*** = {  
 host: 'localhost',  
 port: 3306,  
 name: 'root',  
 pass: 'root',  
 test: 'test'  
};  
const ***config2*** = {  
 host: 'http://atguigu.com',  
 port: 33060,  
 name: 'atguigu.com',  
 pass: 'iloveyou',  
 test2: 'test2'  
}  
//如果有相同的属性，后面那个会把前面那个覆盖  
***console***.log(***Object***.assign(***config1***, ***config2***));

·Object.setPrototypeOf

- 设置原型对象

·Object.getPrototypeof

- 获取原型对象

·不建议使用

//Object.setPrototypeOf 设置原型对象  
//Object.getPrototypeof 获取原型对象  
const ***school*** = {  
 name: '尚硅谷'  
}  
const ***cities*** = {  
 xiaoqu: ['北京','上海','深圳']  
}  
//为school对象设置原型  
***Object***.setPrototypeOf(***school***, ***cities***);  
***console***.log(***school***);  
//获取school的原型对象  
***console***.log(***Object***.getPrototypeOf(***school***));

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>对象方法扩展</title>  
</head>  
<body>  
 <script>  
 //1. Object.is 判断两个值是否完全相等,类似于===  
 // console.log(Object.is(0.1+0.2, 0.3));//false  
 // console.log(Object.is(120, 120));//true  
 // console.log(Object.is("123qwe", "123qwe"));//true  
 // console.log(Object.is(NaN, NaN));//true  
 // //NaN与任何值===都为false  
 // console.log(NaN === NaN);//false  
  
 //2. Object.assign 对象的合并  
 // const config1 = {  
 // host: 'localhost',  
 // port: 3306,  
 // name: 'root',  
 // pass: 'root',  
 // test: 'test'  
 // };  
 // const config2 = {  
 // host: 'http://atguigu.com',  
 // port: 33060,  
 // name: 'atguigu.com',  
 // pass: 'iloveyou',  
 // test2: 'test2'  
 // }  
 // //如果有相同的属性，后面那个会把前面那个覆盖  
 // console.log(Object.assign(config1, config2));  
  
 //Object.setPrototypeOf 设置原型对象  
 //Object.getPrototypeof 获取原型对象  
 const ***school*** = {  
 name: '尚硅谷'  
 }  
 const ***cities*** = {  
 xiaoqu: ['北京','上海','深圳']  
 }  
 //为school对象设置原型  
 ***Object***.setPrototypeOf(***school***, ***cities***);  
 ***console***.log(***school***);  
 //获取school的原型对象  
 ***console***.log(***Object***.getPrototypeOf(***school***));  
 </script>  
</body>  
</html>

# 模块化

·模块化是指将一个大的程序文件，拆分成许多小的文件，然后将小文件组合起来

·模块化的好处

- 防止命名冲突

- 代码复用

- 高维护性

·模块化规范产品

- ES6 之前的模块化规范

-- CommonJS => NodeJS、Browserify

-- AMD => requireJS

-- CMD => seaJS

## ES6 模块化语法

·模块功能主要由两个命令构成：export 和 import

- export 命令用于规定模块的对外接口

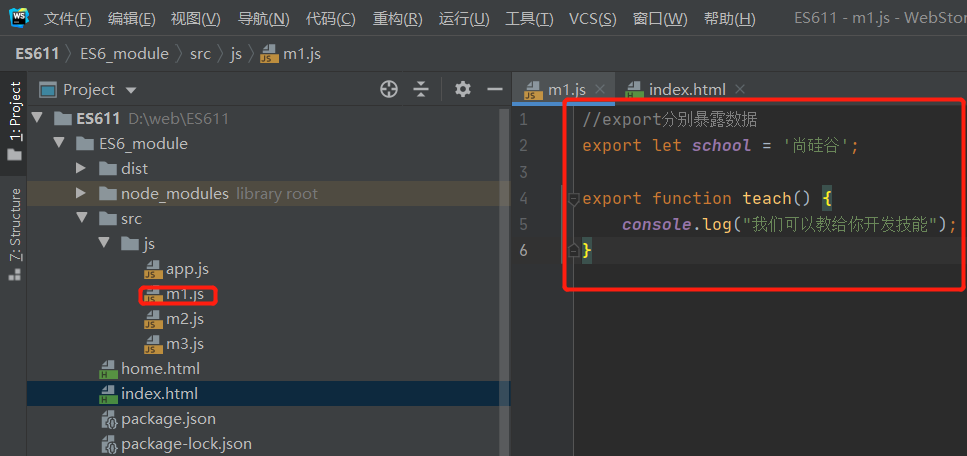
- import 命令用于输入其他模块提供的功能

·as是设置别名

·以下为通用形式引入模块

·分别暴露

- 创建一个模块

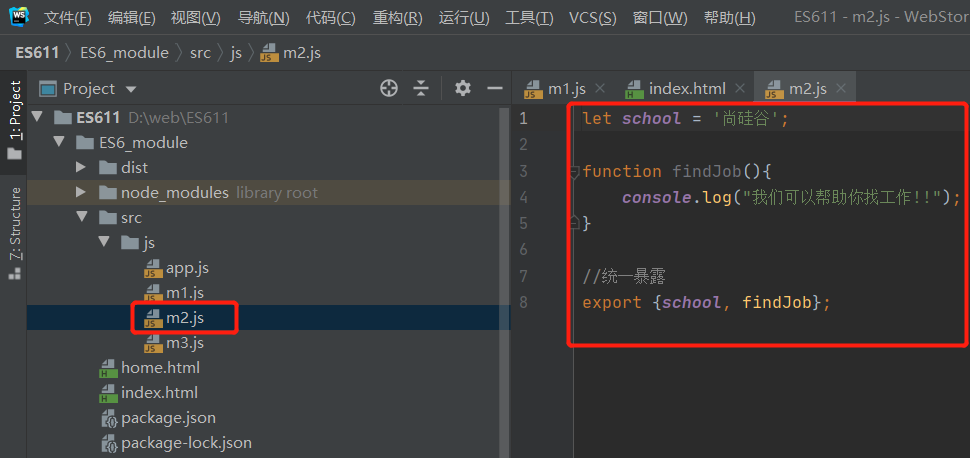


- 在页面中使用模块的内容

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>ES6 模块化</title>  
</head>  
<body>  
 <!--先声明type为module-->  
 <script type="module">  
 //1. 通用的导入方式  
 //引入 m1.js 模块内容  
 import \* as m1 from "./src/js/m1.js";  
 //使用模块内容  
 ***console***.log(m1.***school***);//尚硅谷  
 m1.teach();//我们可以教给你开发技能  
 </script>  
</body>  
</html>

·统一暴露

- 创建一个模块

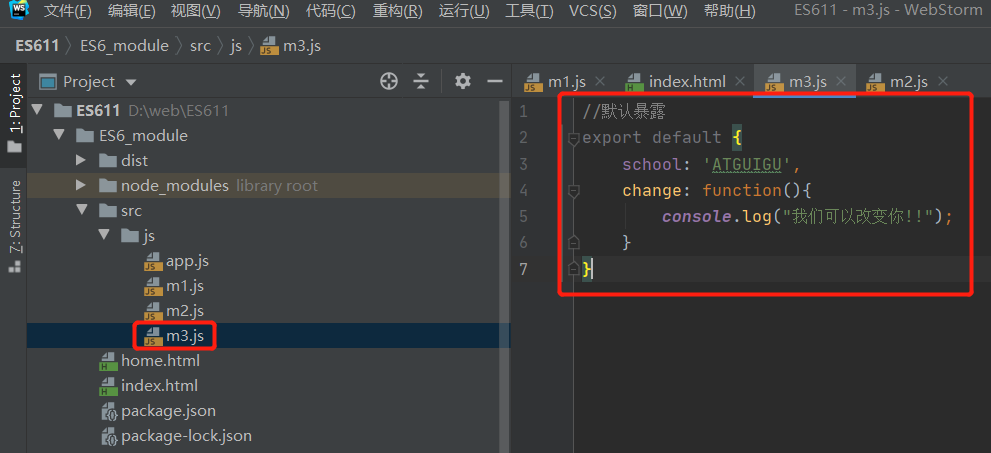


- 在页面中使用模块的内容

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>ES6 模块化</title>  
</head>  
<body>  
 <!--先声明type为module-->  
 <script type="module">  
 //1. 通用的导入方式  
 //引入 m2.js 模块内容  
 import \* as m2 from "./src/js/m2.js";  
 //使用模块内容  
 ***console***.log(m2.***school***);//尚硅谷  
 m2.findJob();//我们可以帮助你找工作!!  
 </script>  
  
</body>  
</html>

·默认暴露

- 创建一个模块



- 在页面中使用模块的内容

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>ES6 模块化</title>  
</head>  
<body>  
 <!--先声明type为module-->  
 <script type="module">  
 //1. 通用的导入方式  
 //引入 m3.js 模块内容  
 import \* as m3 from "./src/js/m3.js";  
 //使用模块内容  
 ***console***.log(m3.default.school);//ATGUIGU  
 m3.default.change();//我们可以改变你!!  
 </script>  
  
</body>  
</html>

·解构赋值形式引入模块

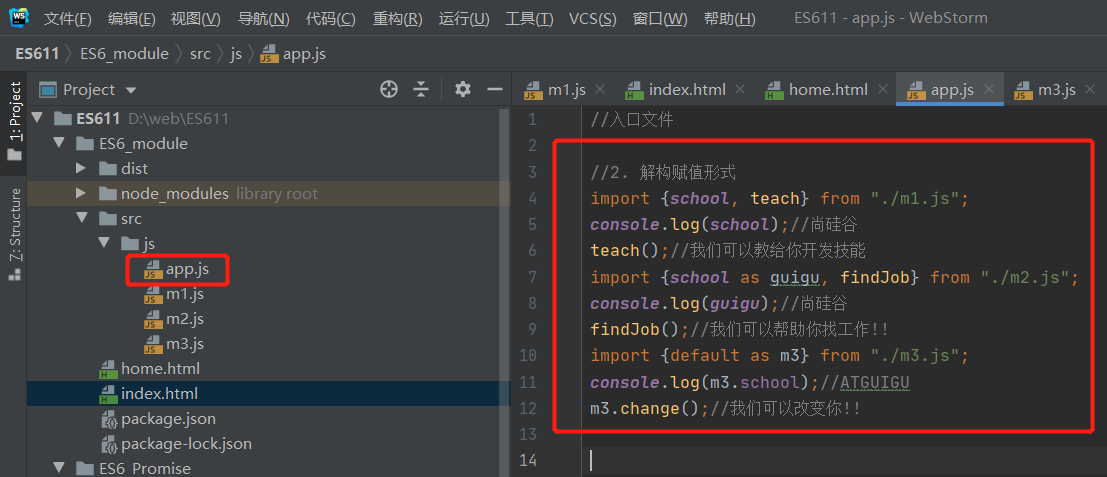
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>ES6 模块化</title>  
</head>  
<body>  
 <!--先声明type为module-->  
 <script type="module">  
 //2. 解构赋值形式  
 import {***school***, teach} from "./src/js/m1.js";  
 ***console***.log(***school***);//尚硅谷  
 teach();//我们可以教给你开发技能  
 import {***school*** as guigu, findJob} from "./src/js/m2.js";  
 ***console***.log(***guigu***);//尚硅谷  
 findJob();//我们可以帮助你找工作!!  
 import {default as m3} from "./src/js/m3.js";  
 ***console***.log(m3.school);//ATGUIGU  
 m3.change();//我们可以改变你!!  
   
 </script>  
  
</body>  
</html>

·简便形式引入模块

- 针对默认暴露

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>ES6 模块化</title>  
</head>  
<body>  
 <!--先声明type为module-->  
 <script type="module">  
  
 //3. 简便形式 针对默认暴露  
 import m3 from "./src/js/m3.js";  
 ***console***.log(m3.school);//ATGUIGU  
 m3.change();//我们可以改变你!!  
 </script>  
  
</body>  
</html>

·外部引入js文件



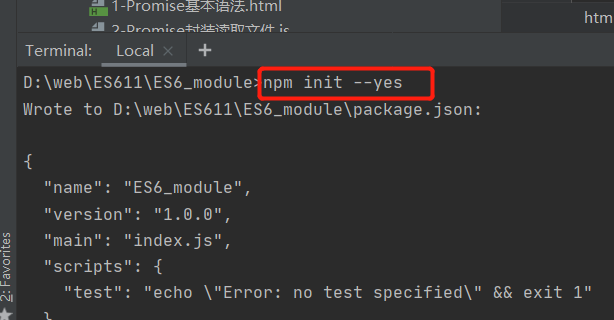
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>ES6 模块化</title>  
</head>  
<body>  
 <!--需要声明type为module-->  
 <script src="./src/js/app.js" type="module"></script>  
</body>  
</html>

## babel对ES6模块化代码转换

·以下操作注意路径

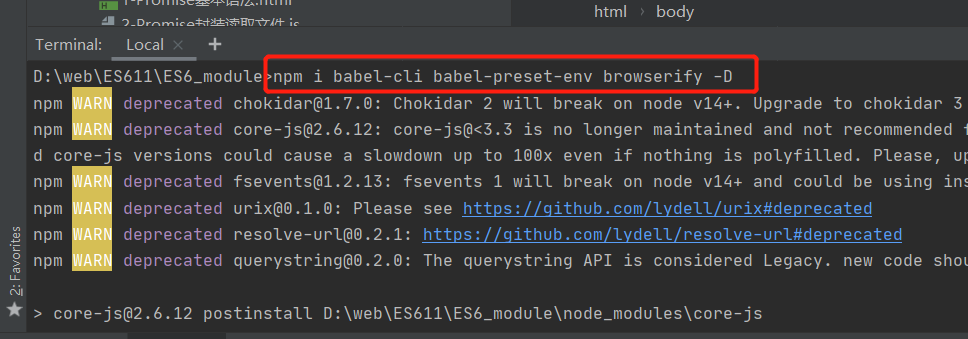
·初始化npm

- npm init --yes



·安装babel相关工具

- npm i babel-cli babel-preset-env browserify -D

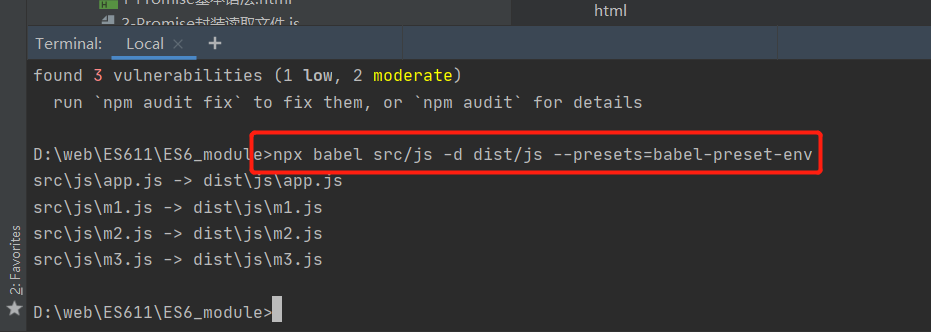


·执行babel命令编译js文件

- npx babel src/js -d dist/js --presets=babel-preset-env

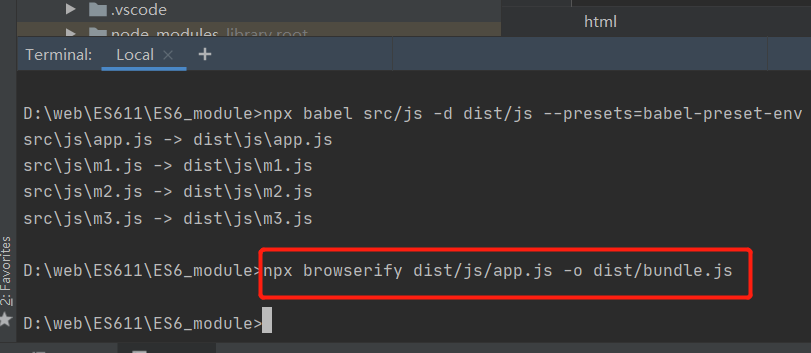
- src/js是要编译js文件夹的路径

- 编译结果保存到dist/js文件夹



·打包编译后的文件

- npx browserify dist/js/app.js -o dist/bundle.js



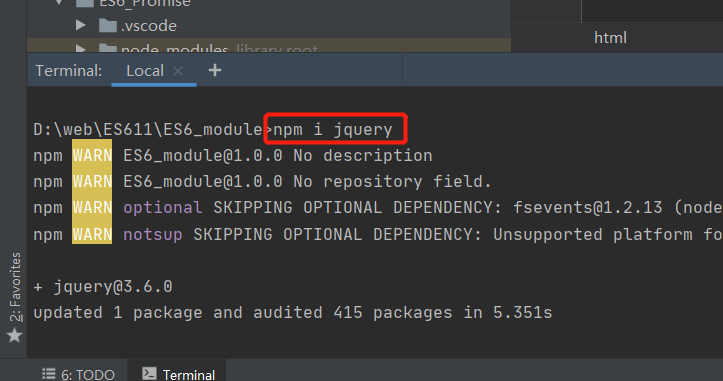
·引入文件

<!--引入文件-->  
 <script src="dist/bundle.js"></script>

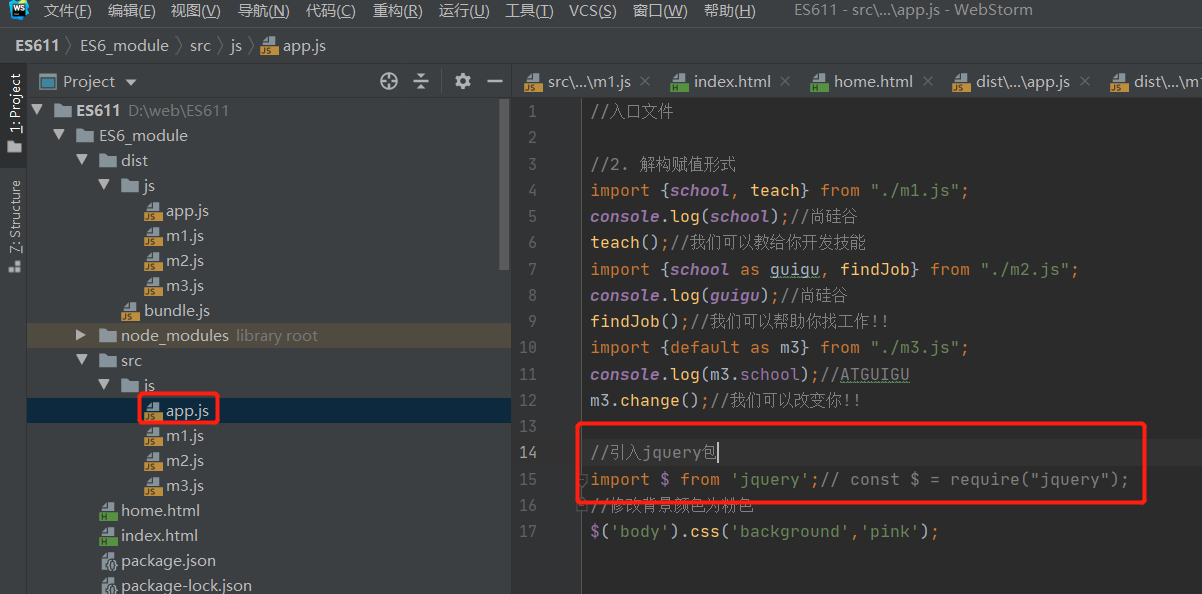
## 引入NPM包

·安装jquery包

- npm i jquery



·引入jquery包



import $ from 'jquery';

·重新编译并打包js文件

# ES7新特性

·Array.prototype.includes

- 用来检测数组中是否包含某个元素，返回布尔类型值

// includes indexOf  
const ***mingzhu*** = ['西游记','红楼梦','三国演义','水浒传'];  
//判断数组是否含有某个元素  
***console***.log(***mingzhu***.includes('西游记'));//true  
***console***.log(***mingzhu***.includes('金瓶梅'));//false

·指数操作符

- 在 ES7 中引入指数运算符「\*\*」，用来实现幂运算，功能与 Math.pow 结果相同

//幂运算  
***console***.log(2 \*\* 10);//1024  
***console***.log(***Math***.pow(2, 10));//1024

# ES8新特性

## async 和 await

·async 和 await 两种语法结合可以让异步代码像同步代码一样

### async 函数

·async 函数的返回值为 promise 对象

·promise 对象的结果由 async 函数执行的返回值决定

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>async函数</title>  
</head>  
<body>  
 <script>  
 //async 函数  
 async function fn(){  
 // 返回一个字符串  
 // return '尚硅谷';  
 // 返回的结果不是一个 Promise 类型的对象, 返回的结果就是成功 Promise 对象  
 // return;  
 //抛出错误, 返回的结果是一个失败的 Promise  
 // throw new Error('出错啦!');  
 //返回的结果如果是一个 Promise 对象  
 return new ***Promise***((resolve, reject)=>{  
 resolve('成功的数据');  
 // reject("失败的错误");  
 });  
 }  
  
 const ***result*** = fn();//接收promise的返回结果  
  
 //调用 then 方法  
 ***result***.then(value => {  
 ***console***.log(value);  
 }, reason => {  
 ***console***.warn(reason);  
 })  
 </script>  
</body>  
</html>

### await 表达式

·await 必须写在 async 函数中

·await 右侧的表达式一般为 promise 对象

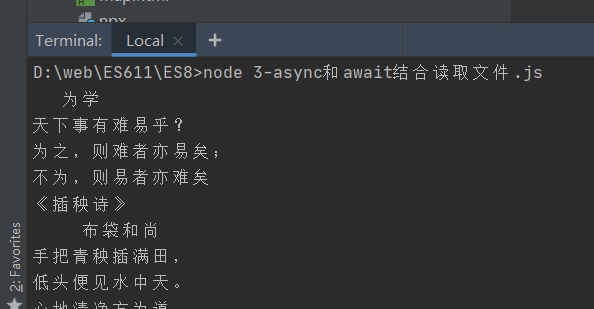
·await 返回的是 promise 成功的值

·await 的 promise 失败了, 就会抛出异常, 需要通过 try...catch 捕获处理

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>await</title>  
</head>  
  
<body>  
 <script>  
 //创建 promise 对象  
 const ***p*** = new ***Promise***((resolve, reject) => {  
 // resolve("用户数据");  
 reject("失败啦!");  
 })  
  
 // await 要放在 async 函数中.  
 async function main() {  
 try {  
 let result = await ***p***;  
 //  
 ***console***.log(result);  
 } catch (e) {  
 ***console***.error(e);  
 }  
 }  
 //调用函数  
 main();  
 </script>  
</body>  
  
</html>

## async和await结合读取文件

//依次读取 为学 插秧诗 观书有感  
  
//1. 引入 fs 模块  
const fs = require("fs");  
  
//封装三个任务，返回结果都是promise对象  
//读取为学  
function readWeiXue() {  
 return new ***Promise***((resolve, reject) => {  
 fs.readFile("./resources/为学.md",(err,data)=>{  
 if (err){//如果读取失败  
 reject(err);//将失败结果返回  
 }  
 //如果成功  
 resolve(data);//将成功数据返回  
 })  
 });  
}  
  
//读取插秧诗  
function readChaYangShi() {  
 return new ***Promise***((resolve, reject) => {  
 fs.readFile("./resources/插秧诗.md",(err,data)=>{  
 if (err){//如果读取失败  
 reject(err);//将失败结果返回  
 }  
 //如果成功  
 resolve(data);//将成功数据返回  
 })  
 });  
}  
  
//读取观书有感  
function readGSYG() {  
 return new ***Promise***((resolve, reject) => {  
 fs.readFile("./resources/观书有感.md",(err,data)=>{  
 if (err){//如果读取失败  
 reject(err);//将失败结果返回  
 }  
 //如果成功  
 resolve(data);//将成功数据返回  
 })  
 });  
}  
  
//声明一个 async函数  
async function main() {  
 try {  
 //调用函数，获取为学内容  
 let weixue = await readWeiXue();  
 //调用函数，获取插秧诗内容  
 let chayang = await readChaYangShi();  
 //调用函数，获取观书有感内容  
 let gsyg = await readGSYG();  
 ***console***.log(weixue.toString());  
 ***console***.log(chayang.toString());  
 ***console***.log(gsyg.toString());  
 }catch (e) {  
 ***console***.log(e);  
 }  
}  
//调用async函数  
main()



## anync与await结合发送AJAX请求

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>发送 AJAX 请求</title>  
</head>  
  
<body>  
 <script>  
 // 发送 AJAX 请求, 返回的结果是 Promise 对象  
 //使用axios更方便  
 function sendAJAX(url) {  
 return new ***Promise***((resolve, reject) => {  
 //创建对象  
 const xhr = new ***XMLHttpRequest***();  
 //初始化  
 xhr.open("GET",url);  
 //发送  
 xhr.send();  
 //事件绑定  
 xhr.onreadystatechange = function () {  
 if (xhr.readyState === 4){  
 if (xhr.status>=200 && xhr.status<300){  
 //成功啦  
 resolve(xhr.response);  
 }else {  
 //如果失败  
 reject(xhr.status);  
 }  
 }  
 }  
 });  
 }  
  
 //promise then 测试  
 // const result = sendAJAX("https://api.apiopen.top/getJoke");  
 // result.then(value => {  
 // console.log(value);  
 // },reason => {  
 // console.error(reason);  
 // });  
  
 //async 与 await测试  
 async function main() {  
 try {  
 //发送AJAX请求  
 let result = await sendAJAX("https://api.apiopen.top/getJoke");  
 //再次发送AJAX请求  
 let result1 = await sendAJAX(" http://api.k780.com/?app=weather.today&weaId=1&appkey=10003&sign=b59bc3ef6191eb9f747dd4e83c99f2a4&format=json");  
 ***console***.log(result);  
 ***console***.log(result1);  
 }catch (e) {  
 ***console***.log(e);  
 }  
 }  
 //调用main()  
 main();  
 </script>  
</body>  
  
</html>

## ES8对象方法扩展

·Object.keys()

- 获取对象所有的键的数组

//获取对象所有的键的数组  
***console***.log(***Object***.keys(***school***));//["name", "cities", "xueke"]

·Object.values()

- 返回一个给定对象的所有可枚举属性值的数组

//获取对象所有的值  
***console***.log(***Object***.values(***school***));//["尚硅谷", Array(3), Array(4)]

·Object.entries()

- 返回一个给定对象自身可遍历属性 [key,value] 的数组

//entries获取对象的键和值，类似键值对，方便创建 Map  
***console***.log(***Object***.entries(***school***));//[Array(2), Array(2), Array(2)]  
//创建 Map，将school对象的键与值存入其中  
const ***m*** = new ***Map***(***Object***.entries(***school***));  
***console***.log(***m***.get('cities'));//["北京", "上海", "深圳"]

·Object.getOwnPropertyDescriptors

- 该方法返回指定对象所有自身属性的描述对象

//对象属性的描述对象，包含属性名、属性值和属性特性  
***console***.log(***Object***.getOwnPropertyDescriptors(***school***));  
  
//创建对象时就可设置属性特性  
const ***obj*** = ***Object***.create(null, {  
 name: {  
 //设置值  
 value: '尚硅谷',  
 //属性特性  
 //是否可写  
 writable: true,  
 //是否可删除  
 configurable: true,  
 //是否可枚举  
 enumerable: true  
 }  
});

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>ES8 对象方法扩展</title>  
</head>  
<body>  
 <script>  
 //声明对象  
 const ***school*** = {  
 name:"尚硅谷",  
 cities:['北京','上海','深圳'],  
 xueke: ['前端','Java','大数据','运维']  
 };  
  
 //获取对象所有的键的数组  
 ***console***.log(***Object***.keys(***school***));//["name", "cities", "xueke"]  
 //获取对象所有的值  
 ***console***.log(***Object***.values(***school***));//["尚硅谷", Array(3), Array(4)]  
 //entries获取对象的键和值，类似键值对，方便创建 Map  
 ***console***.log(***Object***.entries(***school***));//[Array(2), Array(2), Array(2)]  
 //创建 Map，将school对象的键与值存入其中  
 const ***m*** = new ***Map***(***Object***.entries(***school***));  
 ***console***.log(***m***.get('cities'));//["北京", "上海", "深圳"]  
  
 //对象属性的描述对象，包含属性名、属性值和属性特性  
 ***console***.log(***Object***.getOwnPropertyDescriptors(***school***));  
  
 //创建对象时就可设置属性特性  
 const ***obj*** = ***Object***.create(null, {  
 name: {  
 //设置值  
 value: '尚硅谷',  
 //属性特性  
 //是否可写  
 writable: true,  
 //是否可删除  
 configurable: true,  
 //是否可枚举  
 enumerable: true  
 }  
 });  
 </script>  
</body>  
</html>

# ES9的新特性

## Rest/Spread 属性

·Rest 参数与 spread 扩展运算符在 ES6 中已经引入，不过 ES6 中只针对于数组，在 ES9 中为对象提供了像数组一样的 rest 参数和扩展运算符

·原始方法

// 原始方法  
function connect({host, port, username,password}){  
 ***console***.log(host);  
 ***console***.log(port);  
 ***console***.log(username);  
 ***console***.log(password);  
}  
  
//调用函数  
connect({  
 host: '127.0.0.1',  
 port: 3306,  
 username: 'root',  
 password: 'root',  
});  
//以上输出  
//127.0.0.1  
//3306  
//root  
//root

·使用rest参数

//rest 参数,传入的其余的值存到user对象里边  
function connect({host, port, ...user}){  
 ***console***.log(host);  
 ***console***.log(port);  
 ***console***.log(user);  
}  
  
connect({  
 host: '127.0.0.1',  
 port: 3306,  
 username: 'root',  
 password: 'root',  
 type: 'master'  
});  
//以上输出  
//127.0.0.1  
//{username: "root", password: "root", type: "master"}

·使用rest合并对象

//对象合并  
const ***skillOne*** = {  
 q: '天音波'  
}  
  
const ***skillTwo*** = {  
 w: '金钟罩'  
}  
  
const ***skillThree*** = {  
 e: '天雷破'  
}  
const ***skillFour*** = {  
 r: '猛龙摆尾',  
 t: '猛男过江'  
}  
  
//合并对象  
// ...skillFoue => r: '猛龙摆尾', t: '猛男过江'  
const ***mangseng*** = {...***skillOne***, ...***skillTwo***, ...***skillThree***, ...***skillFour***};  
  
***console***.log(***mangseng***);//{q: "天音波", w: "金钟罩", e: "天雷破", r: "猛龙摆尾", t: "猛男过江"}

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>对象展开</title>  
</head>  
  
<body>  
 <!--   
 Rest 参数与 spread 扩展运算符在 ES6 中已经引入，不过 ES6 中只针对于数组，  
 在 ES9 中为对象提供了像数组一样的 rest 参数和扩展运算符  
 -->  
 <script>  
 // 原始方法  
 // function connect({host, port, username,password}){  
 // console.log(host);  
 // console.log(port);  
 // console.log(username);  
 // console.log(password);  
 // }  
 //  
 // //调用函数  
 // connect({  
 // host: '127.0.0.1',  
 // port: 3306,  
 // username: 'root',  
 // password: 'root',  
 // });  
 //以上输出  
 //127.0.0.1  
 //3306  
 //root  
 //root  
  
 //rest 参数,传入的其余的值存到user对象里边  
 // function connect({host, port, ...user}){  
 // console.log(host);  
 // console.log(port);  
 // console.log(user);  
 // }  
 //  
 // connect({  
 // host: '127.0.0.1',  
 // port: 3306,  
 // username: 'root',  
 // password: 'root',  
 // type: 'master'  
 // });  
 //以上输出  
 //127.0.0.1  
 //{username: "root", password: "root", type: "master"}  
  
 //对象合并  
 const ***skillOne*** = {  
 q: '天音波'  
 }  
  
 const ***skillTwo*** = {  
 w: '金钟罩'  
 }  
  
 const ***skillThree*** = {  
 e: '天雷破'  
 }  
 const ***skillFour*** = {  
 r: '猛龙摆尾',  
 t: '猛男过江'  
 }  
  
 //合并对象  
 // ...skillFoue => r: '猛龙摆尾', t: '猛男过江'  
 const ***mangseng*** = {...***skillOne***, ...***skillTwo***, ...***skillThree***, ...***skillFour***};  
  
 ***console***.log(***mangseng***);//{q: "天音波", w: "金钟罩", e: "天雷破", r: "猛龙摆尾", t: "猛男过江"}  
  
  
  
 </script>  
  
</body>  
  
</html>

## 正则表达式命名捕获组

·ES9 允许命名捕获组使用符号『?<name>』,这样获取捕获结果可读性更强

·普通捕获

//声明一个字符串  
let ***str*** = '<a href="http://www.atguigu.com">尚硅谷</a>';  
  
//提取 url 与 『标签文本』  
const ***reg*** = /<a href="(.\*)">(.\*)<\/a>/;  
  
//执行  
const ***result*** = ***reg***.exec(***str***);  
  
***console***.log(***result***[0]);//<a href="http://www.atguigu.com">尚硅谷</a> 整个字符串的结果  
***console***.log(***result***[1]);//http://www.atguigu.com 第一个括号里的结果  
***console***.log(***result***[2]);//尚硅谷 第二个括号里的结果

·命名捕获

let ***str*** = '<a href="http://www.atguigu.com">尚硅谷</a>';  
//分组命名捕获  
const ***reg*** = /<a href="(?<url>.\*)">(?<text>.\*)<\/a>/;  
  
const ***result*** = ***reg***.exec(***str***);  
***console***.log(***result***.groups.url);//http://www.atguigu.com  
***console***.log(***result***.groups.text);//尚硅谷

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>正则扩展-命名捕获分组</title>  
</head>  
  
<body>  
 <script>  
 //声明一个字符串  
 // let str = '<a href="http://www.atguigu.com">尚硅谷</a>';  
 //  
 // //提取 url 与 『标签文本』，“.”默认表示任意字符，\*是出现0次及以上  
 // const reg = /<a href="(.\*)">(.\*)<\/a>/;  
 //  
 // //执行  
 // const result = reg.exec(str);  
 //  
 // console.log(result[0]);//<a href="http://www.atguigu.com">尚硅谷</a> 整个字符串的结果  
 // console.log(result[1]);//http://www.atguigu.com 第一个括号里的结果  
 // console.log(result[2]);//尚硅谷 第二个括号里的结果  
  
   
 let ***str*** = '<a href="http://www.atguigu.com">尚硅谷</a>';  
 //分组命名捕获  
 const ***reg*** = /<a href="(?<url>.\*)">(?<text>.\*)<\/a>/;  
  
 const ***result*** = ***reg***.exec(***str***);  
 ***console***.log(***result***.groups.url);//http://www.atguigu.com  
 ***console***.log(***result***.groups.text);//尚硅谷  
 </script>  
</body>  
  
</html>

## 正则表达式反向断言

·ES9 支持反向断言，通过对匹配结果前面的内容进行判断，对匹配进行筛选

·正向断言

//声明字符串  
let ***str*** = 'JS5211314你知道么555啦啦啦';  
//正向断言,\d是纯数字,+是出现多次,(?=啦)是如果后面是啦,就将前面的提取出来  
const ***reg*** = /\d+(?=啦)/;  
const ***result*** = ***reg***.exec(***str***);  
***console***.log(***result***[0]);//555

·反向断言

//反向断言  
let ***str*** = 'JS5211314你知道么6666啦啦啦';  
const ***reg*** = /(?<=么)\d+/;  
const ***result*** = ***reg***.exec(***str***);  
***console***.log(***result***[0]);//6666

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>正则扩展-反向断言</title>  
</head>  
<body>  
 <script>  
 //声明字符串  
 // //正向断言,\d是纯数字,+是出现1次及以上,(?=啦)是如果后面是啦,就将前面的提取出来  
 // const reg = /\d+(?=啦)/;  
 // const result = reg.exec(str);  
 // console.log(result[0]);//555  
  
 //反向断言  
 let ***str*** = 'JS5211314你知道么6666啦啦啦';  
 const ***reg*** = /(?<=么)\d+/;  
 const ***result*** = ***reg***.exec(***str***);  
 ***console***.log(***result***[0]);//6666  
 </script>  
</body>  
</html>

## 正则表达式 dotAll 模式

·正则表达式中点.匹配除回车外的任何单字符，标记『s』改变这种行为，允许行终止符出现

·普通模式

//普通模式声明正则,\s是空格,?是出现0次或1次  
let ***reg*** = /<li>\s+<a>(.\*?)<\/a>\s+<p>(.\*?)<\/p>/g;  
const ***result*** = ***reg***.exec(***str***);  
***console***.log(***result***);

·dogAll模式

- 可以一次匹配多个

//dogAll模式  
// reg = /<li>\s+<a>(.\*?)<\/a>\s+<p>(.\*?)<\/p>/gs;  
//也可以用这个  
***reg*** = /<li>.\*?<a>(.\*?)<\/a>.\*?<p>(.\*?)<\/p>/gs;  
// 执行匹配  
// const result = reg.exec(str);  
let ***result***;  
let ***data*** = [];  
while(***result*** = reg.exec(***str***)){  
 ***data***.push({title: ***result***[1], time: ***result***[2]});  
}  
// 输出结果  
***console***.log(***data***);

<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>正则扩展-dotAll模式</title>  
</head>  
  
<body>  
 <script>  
 //dot . 元字符 除换行符以外的任意单个字符  
 let ***str*** = `  
 <ul>  
 <li>  
 <a>肖生克的救赎</a>  
 <p>上映日期: 1994-09-10</p>  
 </li>  
 <li>  
 <a>阿甘正传</a>  
 <p>上映日期: 1994-07-06</p>  
 </li>  
 </ul>`;  
 //普通模式声明正则,\s是空格,?是出现0次或1次  
 // let reg = /<li>\s+<a>(.\*?)<\/a>\s+<p>(.\*?)<\/p>/g;  
 // const result = reg.exec(str);  
 // console.log(result);  
  
 //dogAll模式  
 // reg = /<li>\s+<a>(.\*?)<\/a>\s+<p>(.\*?)<\/p>/gs;  
 //也可以用这个  
 ***reg*** = /<li>.\*?<a>(.\*?)<\/a>.\*?<p>(.\*?)<\/p>/gs;  
 // 执行匹配  
 // const result = reg.exec(str);  
 let ***result***;  
 let ***data*** = [];  
 while(***result*** = reg.exec(***str***)){  
 ***data***.push({title: ***result***[1], time: ***result***[2]});  
 }  
 // 输出结果  
 ***console***.log(***data***);  
 </script>  
</body>  
  
</html>

# ES10新特性

## 对象扩展方法

·Object.fromEntries

- 将二维数组转化为对象

- 二维数组

//二维数组  
const ***result*** = ***Object***.fromEntries([  
 ['name','尚硅谷'],  
 ['xueke', 'Java,大数据,前端,云计算']  
]);  
***console***.log(***result***.name);//尚硅谷  
***console***.log(***result***.xueke);//Java,大数据,前端,云计算

- map

//Map  
const ***m*** = new ***Map***();  
***m***.set('name','ATGUIGU');  
const ***result*** = ***Object***.fromEntries(***m***);  
***console***.log(***m***.get("name"));//ATGUIGU

·Object.entries ES8

- 将对象转化为二维数组

//Object.entries ES8  
const ***arr*** = ***Object***.entries({  
 name: "尚硅谷"  
})  
***console***.log(***arr***);

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Object.fromEntries</title>  
</head>  
<body>  
 <script>  
 //二维数组  
 // const result = Object.fromEntries([  
 // ['name','尚硅谷'],  
 // ['xueke', 'Java,大数据,前端,云计算']  
 // ]);  
 // console.log(result.name);//尚硅谷  
 // console.log(result.xueke);//Java,大数据,前端,云计算  
  
 //Map  
 // const m = new Map();  
 // m.set('name','ATGUIGU');  
 // const result = Object.fromEntries(m);  
 // console.log(m.get("name"));//ATGUIGU  
 //Object.entries ES8  
 const ***arr*** = ***Object***.entries({  
 name: "尚硅谷"  
 })  
 ***console***.log(***arr***);  
 </script>  
</body>  
</html>

## 字符串方法扩展

·trimStart

- 清除字符串左侧空白

·trimEnd

- 清除字符串右侧空白

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>trimStart 与 trimEnd</title>  
</head>  
<body>  
 <script>   
 // trim  
 let ***str*** = ' iloveyou ';  
  
 ***console***.log(***str***);// iloveyou  
 //清除字符串左侧空白  
 ***console***.log(***str***.trimStart());//iloveyou  
 //清除字符串右侧空白  
 ***console***.log(***str***.trimEnd());// iloveyou  
 </script>  
</body>  
</html>

## 数组方法扩展

·flat

- 将多维数组转化为低维数组

- 参数为深度 是一个数字 默认值为1

//flat 平  
//将多维数组转化为低位数组  
//二维数组  
const ***arr*** = [1,2,3,4,[5,6]];  
//三维数组  
const ***arr1*** = [1,2,3,4,[5,6,[7,8,9]]];  
//参数为深度 是一个数字  
***console***.log(***arr***.flat());//[1, 2, 3, 4, 5, 6]  
//三维转二维  
***console***.log(***arr1***.flat());//[1, 2, 3, 4, 5, 6, Array(3)]  
//三维转一维  
***console***.log(***arr1***.flat(2));

·flatMap

- 将多维数组转化为低维数组，然后进行运算

//flatMap  
const ***arr*** = [1,2,3,4];  
//arr中每个值\*10然后返回  
const ***result*** = ***arr***.flatMap(item => [item \* 10]);  
***console***.log(***result***);//[10, 20, 30, 40]

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>flat 与 flatMap</title>  
</head>  
<body>  
 <script>  
 //flat 平  
 //将多维数组转化为低位数组  
 //二维数组  
 // const arr = [1,2,3,4,[5,6]];  
 // //三维数组  
 // const arr1 = [1,2,3,4,[5,6,[7,8,9]]];  
 // //参数为深度 是一个数字  
 // console.log(arr.flat());//[1, 2, 3, 4, 5, 6]  
 // //三维转二维  
 // console.log(arr1.flat());//[1, 2, 3, 4, 5, 6, Array(3)]  
 // //三维转一维  
 // console.log(arr1.flat(2));  
  
 //flatMap  
 const ***arr*** = [1,2,3,4];  
 //arr中每个值\*10然后返回  
 const ***result*** = ***arr***.flatMap(item => [item \* 10]);  
 ***console***.log(***result***);//[10, 20, 30, 40]  
 </script>  
</body>  
</html>

## description

·获取symbol的备注字符串

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Symbol.prototype.description</title>  
</head>  
<body>  
 <script>  
 //创建 Symbol  
 let ***s*** = ***Symbol***('尚硅谷');  
 ***console***.log(***s***.description);//尚硅谷  
 </script>  
</body>  
  
</html>

# ES11新特性

## 私有属性

- 类的外部不可见

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>私有属性</title>  
</head>  
<body>  
 <script>  
 class Person{  
 //公有属性  
 name;  
 //私有属性  
 #age;  
 #weight;  
 //构造方法  
 constructor(name, age, weight){  
 this.name = name;  
 this.#age = age;  
 this.#weight = weight;  
 }  
  
 //获取年龄  
 get getAge(){  
 return this.#age;  
 }  
 get getWeight(){  
 return this.#weight;  
 }  
  
 }  
  
 //实例化  
 const ***girl*** = new Person('晓红', 18, '45kg');  
  
 ***console***.log(***girl***.name);//晓红  
 // console.log(girl.#age);//报错  
 // console.log(girl.#weight);//报错  
 ***console***.log(***girl***.getAge);//18  
 ***console***.log(***girl***.getWeight);//45kg  
 </script>  
</body>  
</html>

## Promise.allSettled

·适用于批量异步调用的场景

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Promise.allSettled</title>  
</head>  
<body>  
 <script>  
 //声明两个promise对象  
 const ***p1*** = new ***Promise***((resolve, reject)=>{  
 //模拟网络延时  
 setTimeout(()=>{  
 resolve('商品数据 - 1');  
 // reject('出错啦!');  
 },1000)  
 });  
  
 const ***p2*** = new ***Promise***((resolve, reject)=>{  
 //模拟网络延时  
 setTimeout(()=>{  
 // resolve('商品数据 - 2');  
 reject('出错啦!');  
 },1000)  
 });  
  
 //调用 allsettled 方法  
 // 返回结果是promise对象  
 //返回状态永远是成功  
 //promise对象的值是一个数组,里面包含对应promise的返回状态和返回值  
 const ***result*** = ***Promise***.allSettled([***p1***, ***p2***]);  
 ***console***.log(***result***);  
  
 //与allSettled类似  
 //如果都成功他才返回成功  
 const ***res*** = ***Promise***.all([***p1***, ***p2***]);  
 ***console***.log(***res***);  
  
 </script>  
</body>  
</html>

## String.prototype.matchAll

·正则批量匹配的方法

·对数据的批量提取非常有用

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>String.prototype.matchAll</title>  
</head>  
<body>  
 <script>  
 let ***str*** = `<ul>  
 <li>  
 <a>肖生克的救赎</a>  
 <p>上映日期: 1994-09-10</p>  
 </li>  
 <li>  
 <a>阿甘正传</a>  
 <p>上映日期: 1994-07-06</p>  
 </li>  
 </ul>`;  
  
 //声明正则  
 const ***reg*** = /<li>.\*?<a>(.\*?)<\/a>.\*?<p>(.\*?)<\/p>/sg;  
  
 //调用方法  
 const ***result*** = ***str***.matchAll(***reg***);  
 //返回结果是一个可迭代对象  
 ***console***.log(***result***);  
  
 //使用for of迭代  
 // for(let v of result){  
 // console.log(v);  
 // }  
  
 //使用扩展运算符  
 const ***arr*** = [...***result***];  
  
 ***console***.log(***arr***);  
  
 </script>  
</body>  
</html>

## 可选链操作符

·?.

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>可选链操作符</title>  
</head>  
<body>  
 <script>  
 // ?.  
 function main(config){  
 //获取数据库链接的host属性  
 //方法一  
 //如果用户没传参数就会报错  
 const dbHost = config && config.db && config.db.host;  
 ***console***.log(dbHost);//192.168.1.100  
  
 //方法2 可选链操作符  
 //如果有config则读取.db  
 //如果有.db则读取.host  
 //如果用户没传参数也不会报错  
 const dbHost1 = config?.db?.host;  
 ***console***.log(dbHost1);//192.168.1.100  
 }  
  
 main({  
 db: {  
 host:'192.168.1.100',  
 username: 'root'  
 },  
 cache: {  
 host: '192.168.1.200',  
 username:'admin'  
 }  
 })  
 </script>  
</body>  
</html>

## 动态 import

·按需加载，用到的时候才加载对应模块

·hello.js

export function hello(){  
 alert('Hello');  
}

·app.js

//静态引入  
// import \* as m1 from "./hello.js";  
//获取元素  
const btn = ***document***.getElementById('btn');  
  
btn.onclick = function(){  
 //引入hello.js模块,并使用模块里面的属性  
 import('./hello.js').then(module => {  
 module.hello();  
 });  
}

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>动态 import </title>  
</head>  
<body>  
 <button id="btn">点击</button>  
 <script src="./js/app.js" type="module"></script>  
</body>  
</html>

## BigInt数据类型

·新的数据类型

·大整型

·用于大的数值运算

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>BigInt</title>  
</head>  
<body>  
 <script>  
 //大整形  
 let ***n*** = 521n;  
 ***console***.log(***n***, typeof(***n***));//521n "bigint"  
  
 //函数  
 ***n*** = 123;  
 //将普通值转化成大整形的值  
 ***console***.log(***BigInt***(***n***));//123n  
 //不能转换浮点型数据  
 // console.log(BigInt(1.2));//报错  
  
 //大数值运算  
 let ***max*** = ***Number***.MAX\_SAFE\_INTEGER;//最大安全整数  
 ***console***.log(***max***);//9007199254740991  
 ***console***.log(***max*** + 1);//9007199254740992  
 ***console***.log(***max*** + 2);//9007199254740992  
 //转换成大整形  
 ***max*** = ***BigInt***(***max***);  
 ***console***.log(***max***);//9007199254740991n  
 //不能与普通number运算  
 // console.log(max + 2);//报错  
 ***console***.log(***max*** + ***BigInt***(2));//9007199254740993n  
 ***console***.log(***max*** + ***BigInt***(5));//9007199254740996n  
 </script>  
</body>  
</html>

## 绝对全局变量globalThis

·始终指向全局对象

·在浏览器中是window对象

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>globalThis</title>  
</head>  
<body>  
 <script>  
 ***console***.log(globalThis);//Window对象  
 </script>  
</body>  
</html>

·在nodejs中是Object [global] 对象