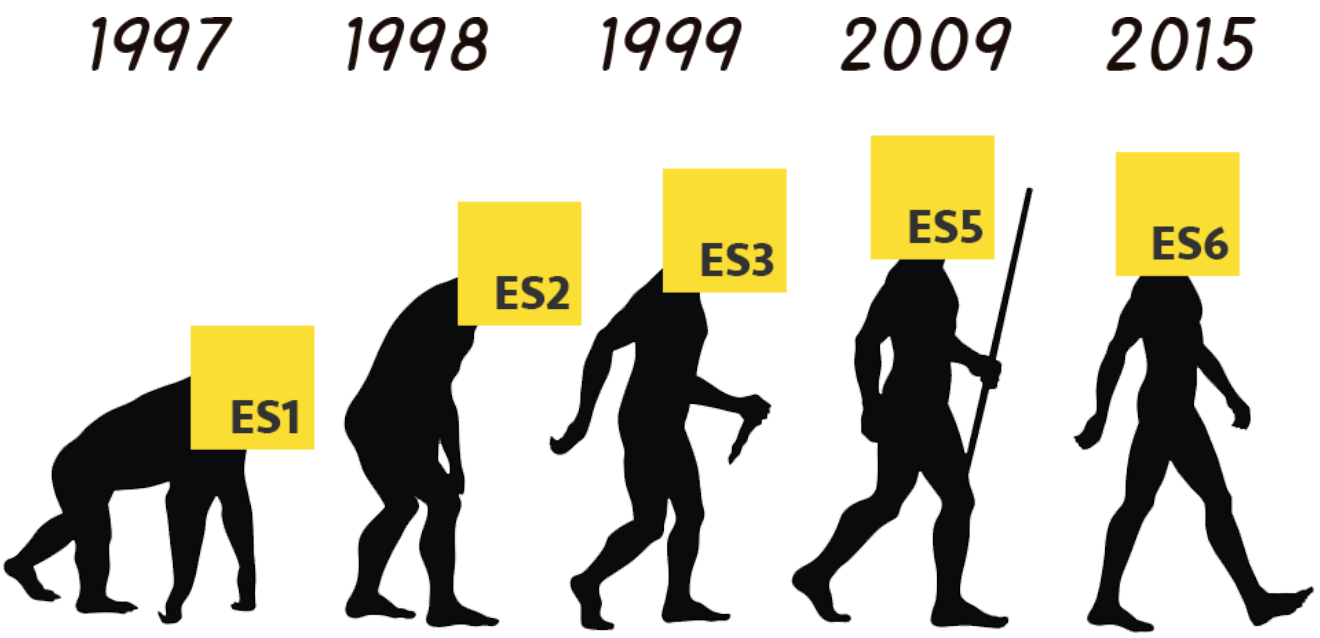
ES6快速入门

1、ECMAScript 6.0 简介

ECMAScript 6.0 (以下简称ES6, ECMAScript是一种由Ecma国际(前身为欧洲计算机制造商协会,英文名称是European Computer Manufacturers Association)通过ECMA-262标准化的脚本程序设计语言)**一种脚本语言的标准**,已经在2015年6月正式发布了,并且从ECMAScript6开始,开始采用年号来做版本。即ECMAScript 2015,就是ECMAScript6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。**每年一个新版本。**



ECMAScript 2015　　es6　　2015年6月17日发布
ECMAScript2016　　es7　　2016年6月19日发布
ECMAScript2017　　es8　　2017在7月12日
ECMAScript 2018　　es9　　2018年 6 月底
ECMAScript 2019　ES10　2019/02/21
ECMAScript 2020　ES11

关键点：

- ECMAScript是每年更新的
- 初始的ECMAScript版本是从1按照数字递增的：**ES1**,**ES2**,**ES3**,**ES4**,**ES5**
- 新的版本(从2015年开始)根据发布的年份命名：**ES2015**,**ES2016**,**ES2017**
- **ECMAScript**是个标准。**Javascript**是这个标准最流行普遍的实现。其他实现包括：SpiderMonkey,V8和ActionScript

2、 ES6新特性-let&const

我们以后会经常使用let来声明一个变量，还有一个const常量（声明之后不允许改变，一旦声明必须初始化，否则报错）

```
<script>
 const a=3
 a=4
//Uncaught TypeError: Assignment to constant variable. at let&const.html:36

</script>
```

```
<script>
//var声明的变量往往会越域
 //let声明的变量有严格的局部作用域
 {
 var a=1
 let b=2
 }
 console.log(a)
 console.log(b)
//Uncaught ReferenceError: b is not defined at let&const.html:19
</script>
```

```
<script>
 //var可以声明多次,let只可以声明一次
 var a=1
 var a=3
 let b=2
 let b=4
 console.log(a)
 console.log(b)
//Uncaught SyntaxError: Identifier 'b' has already been declared
</script>
```

```
<script>
 //var会变量提升
 //let不会变量提升
 console.log(a)
 var a=1
```

```
 console.log(b)
 let b=2
//let&const.html:33 Uncaught ReferenceError: b is not defined at let&const.html:33
</script>
```

3、 ES6新特性-解构&字符串

```
<script>
//数组解构
 let arr=[1,2,3];

 let d=arr[0];
 let b=arr[1];
 let c=arr[2];
 let [d,b,c]=arr;
 console.log(d,b,c);
</script>
```

```
<script>
//对象解构
 let person={
 name: "jack",
 age: 21,
 language: ['java','js','css'],

 }
 let {name,age,language}=person
  console.log(name,age,language)
</script>
```

```
<script>
 //字符串扩展
 let str="hello.vue";
 console.log(str.startsWith("hello"))//true
 console.log(str.endsWith(".vue"))//true
 console.log(str.includes("e"));//true
 console.log(str.includes("hello"))//true
</script>
```

```
<script>
 //字符串模板
 let ss=`<div>
 <a>11</a>
 </div>`
 console.log(ss)//<div><a>11</a></div>



</script>
```

```
<script>
 let person={
 name: "jack",
 age: 21,
 language: ['java','js','css'],

 }

 let {name,age,language}=person
  console.log(name,age,language)

 //字符串插入变量和表达式.变量名写在${},${}中可以放入js表达式
 function fun(){
 return "这是一个函数"
 }

 let info=`我是${name},今年${age+10},我想说${fun()}`
 console.log(info)
</script>
```

4、 ES6新特性-函数优化

```
<script>

 //函数默认值
 //在ES6以前，我们无法给一个函数参数设置默认值，只能采用变通写法：
 function add(a, b) {
 // 判断b是否为空，为空就给默认值1
 b = b || 1;
 return a + b;
 }
  // 传一个参数
  console.log(add(10));


 //现在可以这么写：直接给参数写上默认值，没传就会自动使用默认值
 function add2(a, b = 1) {
 return a + b;
 }
 console.log(add2(20));

 //可变长度参数
 function fun(...values){
 console.log(values.length)
 }
 fun(5)
 fun(5,5,6)
 //简单的箭头函数
 function fun(a,b){
 return a+b;
```

```
}
var sum=(a,b) => a+b
console.log(sum(11,11))



//箭头函数
const person={
name: "jack",
age: 21,
language: ['java','js','css'],

}
function hello (person) {
console.log(person.name)
}

let hellos=(obj) => console.log(obj.name)
hellos(person)//jack
</script>
```

5、 ES6新特性-对象优化

```
<script>
// 对象的内置函数
const person = {
name: "jack",
age: 21,
language: ['java', 'js', 'css']
}

console.log(Object.keys(person));//["name", "age", "language"]
 console.log(Object.values(person));//["jack", 21, Array(3)]
 console.log(Object.entries(person));//[Array(2), Array(2), Array(2)]

// 对象合并
const target = { a: 1 };
const source1 = { b: 2 };
const source2 = { c: 3 };

//{a:1,b:2,c:3}
Object.assign(target, source1, source2);

console.log(target);//{a:1,b:2,c:3}
```

```
//2）、声明对象简写
const age = 23
const name = ""
const person1 = { age: age, name: name }

```

```
const person2 = { age, name }
console.log(person2);
```

```

//3）、声明对象书写方式
let person={
name: "xushu",
eat: function(food){
console.log("我吃了"+food)
},
eat1: food => console.log("我吃了"+food),
eat3(food){
 console.log("我吃了"+food)

 }
 }
 person.eat("香蕉")
 person.eat1("苹果")
 person.eat3("肥肠")
 </script>
 //对象优化.html:12 我吃了香蕉
//对象优化.html:14 我吃了苹果
///对象优化.html:16 我吃了肥肠
```

```
 //4）、对象拓展运算符
 <script>
//拷贝对象
 let p1={
 name: "zlj",
 age: 19
 }
 let someone={...p1}
 console.log(someone)
  //{name: "zlj", age: 19}
</script>
```

```
<script>
//对象合并
 let name={name: "zlj"}
 let age={age: 19}

 let someone={...name,...age}
 console.log(someone)
 //{name: "zlj", age: 19}
</script>
```

6、 map，reduce

- **map**

map()方法：map，映射，即原数组映射成一个新的数组；

map方法接受一个新参数，这个参数就是将原数组变成新数组的映射关系。

```
//map()：接收一个函数，将原数组中的所有元素用这个函数处理后放入新数组返回。
 let arr = [1, 20, -5, 3];

arr = arr.map((item)=>{
return item*2
});
arr = arr.map(item=> item*2);
```

- **reduce**

语法:

```
arr.reduce(callback,[initialValue])
```

reduce为数组中的每一个元素依次执行回调函数，不包括数组中被删除或从未被赋值的元素，接受四个参数：初始值(或者上一次回调函数的返回值)，当前元素值，当前索引，调用reduce 的数组。

callback (执行数组中每个值的函数，包含四个参数)

1、previousValue (上一次调用回调返回的值，或者是提供的初始值(initialalue))

2、currentValue (数组中当前被处理的元素)

3、index (当前元素在数组中的索引)

4、array (调用reduce 的数组)

initialValue (作为第 一次调用callback 的第一个参数。)

```
<script>
//map():接受一个函数,将原来这个数组中的所有元素处理以后返回

let arr=[1,2,3]
arr1=arr.map(item => item*2)
console.log(arr1)


//reduce()：维数组中的每一个元素依次执行回调函数（不包括已删除的）
 arr=arr.reduce((a,b) => {
 console.log("上一次处理后："+a)
 console.log("当前正在处理："+b)
 return a+b
 },1)
 console.log(arr)
 </script>

```

```
(3) [2, 4, 6]
map,reduce.html:18 上一次处理后：1
map,reduce.html:19 当前正在处理：1
map,reduce.html:18 上一次处理后：11
map,reduce.html:19 当前正在处理：2
map,reduce.html:18 上一次处理后：112
map,reduce.html:19 当前正在处理：3
map,reduce.html:22 1123
```

7、 promise异步编排

- 在JavaScript的世界中，所有代码都是单线程执行的。由于这个“缺陷”，导致JavaScript的所有网络操作，浏览器事件，都必须是异步执行。异步执行可以用回调函数实现。一旦有一连串的ajax请求a,b,d...，后面的请求依赖前面的请求结果，就需要层层嵌套。这种缩进和层层嵌套的方式，非常容易造成上下文代码混乱，我们不得不非常小心翼翼处理内层函数与外层函数的数据，一旦内层函数使用了上层函数的变量，这种混乱程度就会加剧...总之，这种层叠上下文的层层嵌套方式，着实增加了神经的紧张程度。

案例：用户登录，并展示该用户的各科成绩。在页面发送两次请求:
1.查询用户，查询成功说明可以登录
2.查询用户成功，查询科目
3.根据科目的查询结果，获取去成绩
分析：此时后台应该提供三个接口，一个提供用户查询接口，一个提供科目的接口，一个提供各科成绩的接口，为了渲染方便，最好响应json数据。在这里就不编写后台接口了，而是提供三个json文件，直接提供json数据，模拟后台接口：
我们以前使用的是ajax请求来完成，如下图

```
//1、查出当前用户信息
//2、按照当前用户的id查出他的课程
//3、按照当前课程id查出分数
$.ajax({
    url: "mock/user.json",
    success(data) {
        console.log("查询用户：", data);
        $.ajax({
            url: `mock/user_corse_${data.id}.json`,
            success(data) {
                console.log("查询到课程：", data);
                $.ajax({
                    url: `mock/corse_score_${data.id}.json`,
                    success(data) {
                        console.log("查询到分数：", data);
                    },
                    error(error) {
                        console.log("出现异常了：" + error);
                    }
                });
            },
            error(error) {
                console.log("出现异常了：" + error);
            }
        });
    },
    error(error) {
        console.log("出现异常了：" + error);
    }
}
```

https://blog.csdn.net/qq_44891295

我们作如上的操作发现是非常复杂的嵌套程序，无限嵌套的方式总是让人感觉到非常凌乱，不工整，我们希望有一种操作来给他们从新编排一下，让我们感觉到操作的顺序性，逻辑性，es6就给我提供了promise功能，如下演示

```
<head>
 <meta charset="utf-8">
 //要导入script
 <script src="https://cdn.bootcss.com/jquery/3.4.1/jquery.js"></script>
</head>
<script >
 //1.promise可以封装异步操作
 //resolve操作成功以后解析数据
 //reject操作失败来拒绝
  let p=new Promise((resolve,reject) => {
```

```
//1.异步操作
$.ajax({
url: "/mock/user.json",
//操作成功以后解析数据
sucess:function(data){
console.log("查询用户成功："+data)
resolve(data)
},
//操作失败来拒绝
failure:function(err){
console.log("查询用户失败："+err)

reject(err)
}
});
});
p.then((obj) => {
return new promise((resolve,reject) => {
console.log("接受到前面传来的信息："+obj);
$.ajax({
url: "mock/user_corse_${obj.id}.json",
//操作成功以后解析数据
sucess:function(data){
console.log("查询用户课程成功："+data)
resolve(data)
},
//操作失败来拒绝
failure:function(err){
console.log("查询用户课程失败："+data)

reject(err)
}
})
})

}).then((data) => {
console.log("上一步的结果"+data)
$.ajax({
url: "mock/user_score_${data.id}.json",
//操作成功以后解析数据
sucess:function(data){
console.log("查询用户课程分数成功："+data)
resolve(data)
},
//操作失败来拒绝
failure:function(err){
console.log("查询用户课程分数失败："+data)

reject(err)
```

```
}
})
})
</script>
```

我们对上面的方法进行改造，把ajax的请求封装成一个方法，并返回

```
<script>
//对上面的方法进行封装
function get(url,data){
return new Promise((resolve,reject) => {
$.ajax({
url: url,
data: data,
//操作成功以后解析数据
sucess:function(data){
 console.log("查询用户课程分数成功："+data)
 resolve(data)
 },
 //操作失败来拒绝
 failure:function(err){
 console.log("查询用户课程分数失败："+data)

 reject(err)
 }
 })
 })
 }

 get("mock/user.json")
 .then( (data) => {
 console.log("用户查询成功-----")
 return get("mock/user_corse_${data.id}",data)
 })
 .then((data) => {
 console.log("用户课程查询成功------")
 return get("mock/user_score_${data.id}",data)
 })
 .then((data)=>{
 console.log("用户课程成绩查询成功-------")

 })
</script>
```

8、 模块化

1)、什么是模块化

模块化就是把代码进行拆分，方便重复利用。类似java中的导包:要使用一一个包，必须先导包。而JS中没有包的概念，换来的是模块。

在ES6中每一个模块即是一个文件，在文件中定义的变量，函数，对象在外部是无法获取的

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，如果你希望外部可以读取模块当中的内容，就必须使用export来对其进行暴露（输出）
    - export`不仅可以导出对象，一切JS变量都可以导出。比如：基本类型变量、函数、数组、对象。
- import 命令永固导入模块

hello.js

```
// 到处指定组件
export const util = {
sum(a, b) {
return a + b;
}
}

// 到处默认组件
export default {
 sum(a, b) {
 return a + b;
 }
}
```

user.js

```
var name = "jack"
var age = 21
function add(a,b){
 return a + b;
}

export {name,age,add}
```

main.js

```
import abc from "./hello.js"
import {name,add} from "./user.js"

abc.sum(1,2);
console.log(name);
add(1,3);
```

注意:

> 1.模块化一定要以web服务器方式运行
> 2.如果要在html中运行要在<script 加上 type="module"