ES6

第一节 JS历史

战国时代

1995年 Netscape（网景）发明了 LiveScript 后改名为JavaScript

1996年，微软发布了 IE 3.0 并搭载了一个 JavaScript 的克隆版，叫做 JScript

一统江湖

ECMAScript：欧洲计算机制造商协会（ECMA）通过ECMA-262标准化的脚本程序设计语言

1997年06月，ECMAScript 1.0版

1998年06月，ECMAScript 2.0版

1999年12月，ECMAScript 3.0版

2007年10月，ECMAScript 4.0版（版本改变太激进，死了）

2009年12月，ECMAScript 5.0版

2011年06月，ECMAscript 5.1版发布，成为ISO/IEC（国际标准化组织及国际电工委员会）16262:2011）标准

2015年06月17日，ECMAScript 6版，可以叫ECMAScript 6（ES6），也可以叫ECMAScript 2015（ES2015）

未来发展

javascript未来将由ECMA以每年一个版本的方式进行迭代更新，版本号也会按照年份来排序，ES2015、ES2016.....

ES6的十大特征

No	特性	说明
1	Default Parameters	默认参数
2	Template Literals	模板文本
3	Multi-line Strings	多行字符串
4	let、 const	新增的声明变量关键字，支持块级作用域
5	Destructuring Assignment	解构赋值
6	Enhanced Object Literals	增强的对象文本
7	Arrow Functions	箭头函数
8	Promises	优化的异步操作
9	Classes	类
10	Modules	模块

第二节 新增特性详解

1. 参数默认值

```
var link = function(height=50, color='red') {
  ...
}
```

2. 模板文本

新的语法${NAME}，并把它放在**反引号**里``：

```
var name="张三";
var age=18;
var man = `我的名字是:${name}年龄:${age}`;
```

3. 多行字符串

```
var news_title=["新闻标题1","新闻标题2"];
var html=`
<ul>
    <li><a href="##">${news_title[0]}</a></li>
    <li><a href="##">${news_title[1]}</a></li>
</ul>`;
```

4. let与const

新的声明方式，程序的复杂性和大型应用程序的出现，用var声明的变量会造成全局污染；于是，就产生了新的声明方式。

let

```
for(let i=0; i<10; i++){
    console.log(i);//0-9
}
console.log(i); //结果：undefined
```

const

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;
// TypeError: Assignment to constant variable.  类型错误：给常量赋值
```

点	var	let	const
作用域级别	函数级	块级	块级
初始化	否	否	是
变量提升	是	否	否
重复声明	是	否	否

5. 解构赋值

如果我们想从复杂的数据结构（数组 对象）中获取某一个数据，可能需要大量的遍历操作才能够完成。解构赋值，可以将这一过程进行简化。

字符串的解构赋值

```
1）使用方法
let [a, b, c, d] = "hxsd";
console.log(a, b, c, d);  // 结果：h  x  s  d

2）默认值
let [a, b, c, d='k', e='hello'] = "hxsd";
console.log(a, b, c, d, e)//结果：//结果：h    x   s   d   hello  （k被替换）
```

数组的解构赋值

```
//旧
var arr = ["zhangsan", 18];
```

```
var name = arr[0];
var age= arr[1];
console.log(name,age);

//ES6方法：结构一致，一一对应
var [name,age] = ["zhangsan", 18];
console.log(name,age);

let [x,[y,z]] = [7, [8, 9]];
console.log(l,m,n);

let [l, ,n] = [4, 5, 6];
console.log(l,n);
```

对象的解构赋值

```
let {a, b} = {a:1, b:2};
console.log(a,b);
```

```
用户列表
let res = {
  status: 200,
  id: 12,
  data: [{name: "Lily"}, {name: "Tom"}, {name: "John"}]
};
如何获使用用户列表中的每条信息？
1) 传统方法
let status = res.status;
let id = res.id;
let data = res.data;
2) ES6方法
let {status, id, data} = res;
```

修改变量名

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

默认值

```
var obj = {a: 1, b: 2, c: 3};
let {a, b=2, c:C=31, d=4, e="default"} = obj;
console.log(a,b,c,d,e);
```

函数参数的解构赋值(数组 VS 对象)

```
1) 数组
let arr = [10, 20];
//let [a, b] = arr;
function division([a, b]) {
  console.log("a/b:", a / b);
}
division(arr);
2) 对象
let obj = {a: 10, b: 20};
function division({a, b}) {
  console.log("a/b:", a / b)
}
division(obj);
```

6. 增强的对象文本

7.箭头函数

基本语法

```
//一个参数，返回一个变量
var res=function(arg) {
  return arg;
}
//多个参数，返回表达式
var res = function (a, b) {
  return a + b;
};

//没有参数 返回字符串
var res = function () {
  return "hxsd";
};

//-------------------------------------------------------------------

//参数和返回值
var res=fn=arg=>arg;//fn:函数名（任意）  1.arg：形参，2.arg：返回值

//多个参数 返回表达式
var res = (a,b) => a + b;

//没有参数,返回字符串
var res=()=>"hxsd";
```

箭头函数中的this

箭头函数的this，与父函数一致

```
//html
<button id="btn">btn</button>


//js
var btn=document.getElementById("btn");
btn.onclick=function(){
    var i=0;
    setInterval(()=>{
        console.log("箭头 函数this指向："+this);
        i++;
        this.innerHTML=i; //箭头函数的this，与父函数一致
    },1000)
}
```

8. promise

Promise 是一个构造函数，需要用 new 调用，并有以下几个 api：

```
function Promise(resolver) {}

Promise.prototype.then = function() {}
Promise.prototype.catch = function() {}

Promise.resolve = function() {}
Promise.reject = function() {}
Promise.all = function() {}
Promise.race = function() {}
```

then()方法

简单来讲，then 方法就是把原来的回调写法分离出来，在异步操作执行完后，用链式调用的方式执行回调函数。

```
//做饭
function cook() {
    console.log('开始做饭。');
    var p = new Promise(function(resolve, reject) { //做一些异步操作
    setTimeout(function() {
        console.log('饭做好了！');
        resolve('鸡蛋炒饭');//如果有结果，可以将结果返回
        },2000);
    });
    return p;
}

//吃饭
function eat(data){
    console.log('开始吃饭:'+data);
    var p = new Promise(function(resolve, reject){        //做一些异步操作
    setTimeout(function(){
        console.log('吃饭完毕!');
        resolve('一个碗和一双筷子需要刷');
```

```
    }, 2000);
    });
    return p;
}

function wash(data){
    console.log('开始洗碗：'+data);
    var p = new Promise(function(resolve, reject){        //做一些异步操作
    setTimeout(function(){
        console.log('洗碗完毕!');
        resolve('干净的碗筷');
    }, 2000);
    });
    return p;
}


//使用 then 链式调用这三个方法：
cook()
.then(function(){
    return eat();
})
.then(function(){
    return wash();
})
.then(function(){
    console.log();
});

//简写方式
//cook().then(eat).then(wash);
```

9. 类

```
class Cat{
    //构造函数
    constructor(name,age){
        this.name=name; //实例的属性
        this.age=age;//实例的属性
    }
    //实例的方法
    show(){
        console.log(this.name);
    }
    //类的静态方法
    static getColor(){
        console.log(this.color);
    }
}
//ES6规定，类里只可以有静态方法，不能有静态属性,只能如下定义
```

```
Cat.color="白色";

//使用Object.assign 为 类 添加更多方法
Object.assign(Cat.prototype,{
    play(){
        alert("毛线球");
    },
    sleep(){
        alert("打呼噜");
    }
})


//子类----------------------------------------------------
class Ch_cat extends Cat{
    //子类构造函数
    constructor(name,age){
        //使用super调用父类的构造函数
        super(name,age);
    }
    //子类的static方法
    static myColor() {
        //使用super调用父类的静态方法
        super.getColor();//this指向子类
    }
}
Ch_cat.color="黑色";

//------------------------------------------------------------
var cat1=new Cat("小白",3);
cat1.show();
Cat.getColor();

//子类调用static方法
Ch_cat.myColor(); //黑色
```

- 类与对象的关系
  - 类是对象的抽象，对象是类的具体实例
  - 类是抽象的，而对象是具体的
  - 类不占用内存，而对象占用存储空间
- Class类
  - 定义
    - 它作为对象的模板,通过`class`关键字,可以定义类，类名首字母大写
    - constructor构造方法，默认方法，一个类必须有constructor方法
    - ES5的构造函数Person,对应ES6的Person类的构造方法
    - this代表对象实例
    - 类相当于实例的原型，所有在类中定义的方法，会被实例继承
    - 注意：

1) 定义“类”的方法的时候,前面不需要加上function这个关键字

2) 方法之间不需要逗号分隔,加了会报错

- 特点
  - class类让对象原型的写法更加清晰、体现了面向对象编程的思想
  - **class不存在变量提升**,需要先定义再使用
- 类的数据类型：`function` ，且类本身指向构造函数
- 实例属性 VS 原型链属性
  - `hasOwnProperty('')`函数用于指示一个对象自身(不包括原型链)是否具有指定名称的属性

    `person.hasOwnProperty('toString')`
  - `__proto__hasOwnProperty('')` obj上所有的属性

    `person.__proto__.hasOwnProperty('toString')`

```
class Person{
  constructor(x,y,z){
    this.name = x;
    this.age = y;
    this.language = z;
    this.speak = function(){
      console.log(this.language)
    }
  }
  toString(){
    return (this.name + '的年龄是' + this.age + '岁')
  }
}
let man = new Person('张三', 18, 'chinese');
console.log(person.toString())
person.speak();

console.log(typeof Person);
console.log(Person === Person.prototype.constructor)；
```

- 内部定义 VS 单独定义

```
Person.prototype = {
  getName(){
    return '张三';
  },
  getAge(){
    return '12';
  }
}
```

- 添加方法：`Person.prototype = {}`初始化时单独定义或覆盖已定义对象，属于Person类内部定义的方法，不可枚举

- Object.assign方法：给对象Person动态的增加方法，可枚举

```
  Object.assign(Person.prototype,{
    getHeight(){
      console.log('180cm');
    },
    getWeight(){
      console.log('80kg');
    }
  })
```

- 不可枚举性

  `Object.keys(Person.prototype)`

  `Object.getOwnPropertyNames(obj)` obj上所有的实例属性

- Class的静态方法
  - 类相当于实例的原型，所有在类中定义的方法，都会被实例继承
  - 静态方法:Class本身的方法，不是实例方法也不是原型方法
    - 定义：`static`关键字
    - 使用：直接通过类来调用

```
static getName(){
 return '获取Name的静态方法:人类';
}
Person.getName()
```

- Class的静态属性
  - 静态属性指的是Class本身的属性，不是定义在实例对象（this）上的属性
  - 说明：ES6明确规定，Class内部只有静态方法，没有静态属性
  - 应用场景：如果一个数据需要被所有对象共享使用时

```
class Person{
  constructor(x,y,z){
    this.name = x;
    this.age = y;
    this.language = z;
    this.speak = function(){
      console.log(this.language)
    }
  }
  toString(){
    return (this.name + '的年龄是' + this.age + '岁')
  }
}
Person.type = 'function';    //静态属性
console.log(Person.type)
```

- 为什么使用静态
  - 在类第一次加载的时候，static就已经在内存中了，直到程序结束后，该内存才会释放
  - 静态变量在程序运行期间，该内存空间对所有该类的对象实例而言是共享的，有些时候可以认为是全局变量。因此在某些时候为了节省系统内存开销、共享资源，可以将类中的一些变量声明为静态变量!
- 私有方法。。。(不介绍)
- 继承: React
  - 什么是继承？
  - 如何实现子类继承父类？

```
class Parent{
  constructor(name='父类'){
    this.name = name;
  }
}
class Baby extends Person{}
console.log(Baby)
```

- 子类如何向父类传递参数：`super()`
  - super()没有参数，会继承父类所有的属性
  - super一定要放在构造方法的第一行！！！

```
class Parent{
  constructor(name='Parent'){
    this.name = name;
  }
}
class Baby extends Parent{
  constructor(name = 'Baby'){
    super(name);
    this.type = 'Baby';
  }
}
console.log(Baby)
```

- getter

```
class Parent{
  constructor(name='Parent'){
    this.name = name;
  }

  get longName(){       //此处longName为属性
    return 'good' + this.name
  }
}
let p = new Parent();
console.log('getter:',p.longName);
```

- setter

```
class Parent{
  constructor(name='Parent'){
    this.name = name;
  }

  get longName(){       //此处longName为属性
    return 'good' + this.name
  }
  set longName(value){
    this.name = value
  }
}
let p = new Parent();
p.longName = 'bad';
console.log('setter:',p.longName);
```

10. 模块

在项目开发中，js文件要按照加载顺序引入，引入一个js文件，还要知道这个js文件是否“依赖”其他js文件，这就造成了js引入的不便，在团队协作开发中，js文件都是由不同的开发人员编写的，js文件的引入极易造成混乱。

js模块化需要解决的问题：

1. 如何安全的包装一个模块的代码？（不污染模块外的任何代码）
2. 如何唯一标识一个模块？
3. 如何优雅的把模块的API暴漏出去？（不能增加全局变量）
4. 如何方便的使用所依赖的模块？

ES6移除了关于模块如何加载/执行的内容，只保留了定义、引入模块的语法。**目前还没有浏览器能支持**，只能说它是面向未来了。

export输出

```
//a.js 方式一 分别输出每一个，可以是变量 函数 对象 类
export var a = 1;
export var obj = {name: 'abc', age: 20};
export function run(){console.log("aaaaa")};
```

```
//b.js 方式二 export default 对象方式输出全部
var a = 1;
var obj = {name: 'abc', age: 20};
function run(){console.log("bbbbbb")};

export default {a, obj, run};
//default 表示不给导出的对象起名字，名字交给引入方（引入时也无需查看名称），并且可以选择性导出
```

import输入

输入文件一定要使用 "/" "./" "../" 路径

```
//在花括号中指明需使用的API，并且可以用as指定别名
import {a} from "./a.js";
import {run as go} from './a.js';
go();

//可以输入用export default定义的对象，
import mod from "./b.js";
mod.run();
```

第三节 新增

字符串新增方法

includes()

判断是否存在某个字符

```
let str = 'hello';
console.log(str.includes('l'));
```

startsWith()

是否以某字符开头

```
let str = 'hello';
console.log(str.startsWith('h'));
```

endsWith()

是否以某字符结尾

```
console.log(str.endsWith('o'));// 是否以o结尾
```

repeat()

返回一个新字符串，将原字符串重复n次

```
let str1='a';
let str2=str1.repeat(3);
console.log(str2)//aaa
```

数组新增方法

fill()

作用：使用给定值，填充一个数组，fill方法用于空数组的初始化非常方便。数组中已有的元素，会被全部抹去

语法：fill (要替换的内容 , 起始位置 , 终止位置)

```
let arr = ['ss', '中国', 'jsd'];
arr.fill('web',1,3);
console.log(arr);
```

from()

作用：将对象转换成数组，对象必须有length属性，没有length，转出来的就是空数组。

```
//json 转 数组
let json = {
        "0": "aaa",
        "1": "bbb",
        "2": "ccc",
        "length": 3
    };
let arr = Array.from(json);
console.log(json);
console.log(arr);
```

扩展运算符

扩展运算符（ spread ）是 **三个点**（ ... ）

功能：**将一个数组转为用逗号分隔的参数序列**

用途：主要用于函数调用（函数参数不确定时，每个函数最多只能声明一个不定参数，而且一定要放在所有参数的末尾）

```
let [r,...t] = [7, 8, 9];
console.log(r,t);
```

```
//传统方法：
function compare(a,b,c){
    if(a>b && a>c){
        console.log('最大值:', a);
    }else if(b>c){
        console.log('最大值:', b);
    }else{
        console.log('最大值:', c);
    }
}
compare(1,2,3);

//ES6方法：
let max = 0;
function compare(...arg){
    for(let i = 0; i < arg.length; i++){
        if(arg[i] > max){
            max = arg[i];

        }
```

```
    }
    console.log('最大值:', max);
}
compare(1,2,3);
```

```
1）原始数据容易被篡改
let arr1 = ["as", "se", "vd"];
let arr2 = arr1;
arr2.push("qw");
console.log("arr1:",arr1);

2）保持原始数据不变
let arr1 = ["as", "se", "vd"];
let arr2 = [...arr1];
arr2.push("qw");
console.log("arr1:",arr1);
```

新增数据结构

Set数据结构

> 解释：Set是一个数据集合，与数组类似，
>
> 作用：数据去重
>
> 语法

```
let arr = new Set([1, 1, 2, 2, 2, 's', 's']);
console.log("arr:",arr);
```

- 遍历

  重点：Array可以使用下标，Map和Set不能使用下标，所以不能使用for循环进行遍历

```
1）forEach()方法,无返回值
2）map()方法，有返回值
3）for...in，
4）for...of（ES6方法）

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

1）for...of循环
遍历value值：
for(let value of set){
  console.log('value:',value);
}

for(let value of set.values()){
  console.log('value:',value);
}


遍历key值：
```

```
for(let key of set.keys()){
  console.log('keys:',key);
}

遍历key-value值：
for(let [key,value] of set.entries()){
  console.log('entries:',key,value);
}

//注意Set的键名和键值是同一个值，所以key()和values()行为是一致的
```

Map数据结构

> 解释：Map也是一个数据集合，与数组类似
>
> 作用：是对object对象的一个补充，Map的key可以是任意类型的(数组、对象、数字都可以作为它的key)，
>
> 而传统对象的key必须是字符串;

```
1) object对象
let a ={
  data: "string",
  1: "num"
};
console.log(a.data, a.1);
2) Map数据结构
let map = new Map([['data',"string"],[1,'num']]);
console.log(map);
```

- Array、Set、Map对比

```
let arr = [1, 2, 3];
let set = new Set([1, 2, 3]);
let map = new Map([['a', 1],['b', 2],['c', 3]]);
```

类别	Array	Set	Map
长度	arr.length	set.size	map.size
增	arr.push(4)	set.add(4)	map.set('t', 1)
删	arr.splice(0,1)	set.delete(2)	map.delete('t')
改	arr.splice(0,0,9)	遍历	map.set('t',2)
查	遍历	set.has(1)	map.has('t')
清空	arr = []	set.clear()	map.clear()