

补充：JSON

什么是JSON

JSON语法规则

JSON 与 JS 对象的关系

JSON 和 JS 对象互转

eval()

JavaScript提高部分

1.面向对象编程介绍

1.1两大编程思想

1.2面向过程编程POP

1.3面向对象编程OOP

1.4面向过程和面向对象的对比

2.ES6中的类和对象

2.1对象

2.2类class

2.3创建类

2.4类constructor构造函数

2.5类添加方法

2.6类的get和set

3.ES6的类的继承

3.1继承

3.2 super关键字

3.3使用类的注意点

案例：面向对象tab栏

功能需求：

1.该对象具有切换功能

注意：

完整代码

2.面向对象版tab栏切换添加功能

案例分析

关键点

完整代码

3.面向对象版 tab 栏切换删除功能

案例分析

关键点：

细节及功能完善：

完整代码

4.面向对象版tab栏切换编辑功能

案例分析

关键点：

细节：

完整代码

案例小结

4.构造函数和原型

4.1概述

4.2构造函数

4.3构造函数的问题

4.4构造函数原型prototype

4.5对象原型__proto__

4.6 constructor构造函数

4.7构造函数、实例、原型对象三者之间的关系

4.8原型链

4.9 JavaScript的成员查找机制(规则)

4.10扩展内置对象

5.ES5的继承

- 5.1 call()
- 5.2 借用构造函数继承父类型属性
- 5.3 借用原型对象继承父类型方法

6.ES5中的新增方法

- 6.1 ES5新增方法概述
- 6.2 数组方法

1. `forEach()`

2. `filter()`

3. `some()`

filter与some的区别

查询商品案例

- 1. 把数据渲染到页面中
- 2. 根据价格显示数据 filter
- 3. 根据商品名称显示数据

forEach与some的区别

6.3 字符串方法

6.4 对象方法

- 1. `Object.keys()` 用于获取对象自身所有的属性
- 2. `Object.defineProperty()` 定义对象中新属性或修改原有的属性

7.ES6函数进阶

7.1 函数的定义和调用

7.1.1 函数的定义方式

7.1.2 函数的调用方式

7.2 this

7.2.1 函数内 this 的指向

7.2.2 改变函数内部 this 指向

1. `call` 方法

2. `apply` 方法

3. `bind` 方法

call apply bind 总结

7.3 严格模式

7.3.1 什么是严格模式

7.3.2 开启严格模式

1. 为脚本开启严格模式

2. 为函数开启严格模式

7.3.3 严格模式中的变化

1. 变量规定

2. 严格模式下 this 指向问题

3. 函数变化

7.4 高阶函数

7.5 闭包

7.5.1 变量作用域

7.5.2 什么是闭包

7.5.3 闭包的作用

7.5.4 闭包的应用

1. 循环点击注册事件

2. 循环中的 `setTimeout()`

3. 计算打车价格

7.5.5 闭包思考题

7.6 递归

7.6.1 什么是递归?

7.6.2 利用递归求数学题

7.6.3 利用递归遍历数组对象

7.6.4 浅拷贝和深拷贝

8.正则表达式

8.1 正则表达式概述

8.1.1 什么是正则表达式

- 8.1.2正则表达式的特点
- 8.2正则表达式在JavaScript中的使用
 - 8.2.1创建正则表达式
 - 8.2.2测试正则表达式test
- 8.3正则表达式中的特殊字符
 - 8.3.1正则表达式的组成
 - 8.3.2边界符
 - 8.3.3字符类
 - 8.3.4量词符
 - 8.3.5括号总结
 - 8.3.6预定义类
- 案例：品优购注册页面表单验证
- 8.4正则表达式中的替换
 - 8.4.1 replace()替换
 - 8.4.2正则表达式参数
 - 8.4.3search()检索

JavaScript ES6新特性

9.let和const命令

- 9.1let变量声明
- 案例：点击dive盒子变色
- 9.2ES6块级作用域
- 9.3const常量声明
- 9.4顶层对象的属性
- 9.5globalThis对象

10.变量的解构赋值

- 10.1数组的解构赋值
 - 默认值
- 10.2对象的解构赋值
- 10.3用途

11.字符串的扩展

- 11.1模板字符串
- 11.2方法：trimStart(), trimEnd()

12.对象的扩展

- 12.1对象的简化写法
- 12.2对象的扩展运算符
- 12.3对象方法的扩展
 - 1.Object.is()
 - 2.Object.assign()
- 方法的规则：
 - 注意点：
 - 常见用途
- 3.proto属性，Object.setPrototypeOf(), Object.getPrototypeOf()
- 4.Object.keys(), Object.values(), Object.entries()
- 遗漏：Object.create()
- 5.Object.getOwnPropertyDescriptors()
- 6.Object.fromEntries

12.4链判断运算符

13.函数的扩展

- 13.1箭头函数
- 13.2参数默认值
- 13.3rest参数

14.数组的扩展

- 14.1扩展运算符
- 14.2扩展运算符的应用
- 14.3map和reduce方法
 - map()
 - reduce()
- 14.4Array.prototype.includes

14.5flat(), flatMap()

15.Symbol

15.1概述

15.2Symbol.prototype.description

15.3作为属性名的 Symbol

15.4Symbol属性名遍历

15.5Symbol.for(), Symbol.keyFor()

15.6内置的 Symbol 值

 Symbol.hasInstance

 Symbol.isConcatSpreadable

16.Iterator 和 for...of 循环

16.1Iterator (遍历器) 的概念

16.2默认 Iterator 接口

16.3自定义Iterator 接口

16.4调用 Iterator 接口の場合

17.Generator 函数

17.1基本概念

17.2yield 表达式

17.3与 Iterator 接口的关系

17.4next 方法的参数

17.5for...of 循环

17.6Generator 函数的案例

 案例1

 案例2

18.Set和Map数据结构

18.1Set

18.2Set集合实践

18.3Map

babel.js编译

方法1：引入js文件

方法2：编译JS文件

19.Promise

19.1Promise的含义

19.2基本用法

19.3案例

 Promise封装读取文件

 Promise封装AJAX请求

19.4Promise.prototype.then()

 返回一个新Promise对象

 链式调用

19.5Promise案例-多个文件内容读取

19.6Promise.prototype.catch()

19.7Promise.all()

19.8Promise.race()

19.9Promise.allSettled()

20.数值的扩展

20.1二进制和八进制表示法

20.2Number.isFinite(), Number.isNaN()

20.3Number.parseInt(), Number.parseFloat()

20.4Number.isInteger()

20.5Number.EPSILON

20.6安全整数和 Number.isSafeInteger()

20.7Math 对象的扩展

 Math.trunc()

 Math.sign()

20.8指数运算

20.9BigInt

 20.9.1BigInt数据类型

20.9.2 BigInt 对象

21.模块化Module

21.1概述

21.2ES6模块化语法

21.2.1export命令

21.2.2import 命令

21.2.3模块引入方法

21.3 babel对IES6模块化代码转换

21.4ES6模块化引入NPM包

21.5动态import()

22.async 函数

22.1基本用法

22.1.1async函数

22.1.2await表达式

22.1.3案例-读取文件

22.1.4案例-发送AJAX请求

23.正则的扩展

23.1RegExp 构造函数

23.2正则表达式的特殊字符

23.2.1字符型

23.2.2量词

23.3正则表达式方法

23.3.1RegExp对象方法

23.3.2支持正则表达式的String对象的方法

23.3.3方法小结

23.4具名组匹配

组匹配

具名组匹配

案例：提取url和标签文本

23.5 u修饰符

23.5.1点字符

23.5.2Unicode 字符

23.5.3量词

23.5.4预定义模式

23.6后行断言

23.7 s 修饰符：dotAll 模式

23.8String.prototype.matchAll()

24.Class的基本语法

24.1static关键字

24.2私有属性

24.3实例属性的新写法

补充：JSON

什么是JSON

JSON (JavaScript object Notation JS对象表示法) , JSON和JS对象的格式一样，只不过**JSON字符串中的属性名必须加双引号**，其他的和JS语法一致。

JS中的对象只有JS自己认识，其他的语言都不认识，而JSON 可以将 JavaScript 对象中表示的一组数据转换为字符串，然后就可以在网络或者程序之间轻松地传递这个字符串，并在需要的时候将它还原为各编程语言所支持的数据格式。

JSON语法规则

JSON是一个序列化的对象{}或数组[]，可以被任意的语言所识别，并且可以转换为任意语言中的对象，JSON在开发中主要用来数据的交互。

JSON中允许的值:可以是对象、数组、数字、字符串或者三个字面值(false、null、true)中的一个
标准情况下：属性名和属性值为字符串类型必须双引号。

JSON 与 JS 对象的关系

JSON 是 JS 对象的字符串表示法，它使用文本表示一个 JS 对象的信息，本质是一个字符串。

```
var obj = {a: 'Hello', b: 'world'}; //这是一个对象，注意属性名使用双引号
var json = '{"a": "Hello", "b": "world"}'; //这是一个 JSON 字符串，本质是一个字符串
```

JSON 和 JS 对象互转

将JSON字符串转换为JS对象，使用 `JSON.parse()` 方法：写法必须标准，否则会报错

```
var obj = JSON.parse('{"a": "Hello", "b": "World"}'); //结果是 {a: 'Hello', b: 'World'}
```

JS对象转换为JSON字符串，使用 `JSON.stringify()` 方法。写法可以不标准

```
var json = JSON.stringify({a: 'Hello', b: 'world'});
//结果是 '{"a": "Hello", "b": "world"}'
```

标准情况下：属性名和属性值为字符串下必须双引号，可以从结果来看，自动将js中不标准的转化为标准的，将单引号变为双引号。

如果需要兼容IE7及以下的JSON操作，则可以通过引入一个外部的js文件来处理

eval()

`eval()` 函数可计算某个字符串，并执行其中的 JavaScript 代码，将执行结果返回

注意：

- 1.若`eval()`执行的字符串中含有{}，会默认把{}当成是代码块（需要加分号等语法规规范），如果不希望将其当成代码块解析，则需要在字符串前后各加一个()
- 2.`eval()`这个函数的功能很强大，可以直接执行一个字符串中的js代码，但是在开发中尽量不要使用，首先它的执行性能比较差，且具有安全隐患

JavaScript提高部分

1.面向对象编程介绍

1.1两大编程思想

- 面向过程
- 面向对象

1.2面向过程编程POP

面向过程编程POP(Process-oriented programming)

面向过程就是分析出解决问题所需要的步骤,然后用函数把这些步骤一步一步实现,使用的时候再一个一个的依次调用就可以了。

举个栗子:将大象装进冰箱,面向过程做法。



面向过程,就是按照我们分析好了的步骤,按照步骤解决问题。

1.3 面向对象编程OOP

面向对象编程OOP (Object Oriented Programming)

面向对象是把事务分解成为一个个对象,然后由对象之间分工与合作。

举个栗子:将大象装进冰箱,面向对象做法。

先找出对象,并写出这些对象的功能:

1. 大象对象

- 进去

2. 冰箱对象

- 打开
- 关闭

3. 使用大象和冰箱的功能

面向对象是以对象功能来划分问题,而不是步骤。

在面向对象程序开发思想中,每一个对象都是功能中心,具有明确分工。

面向对象编程具有灵活、代码可复用、容易维护和开发的优点,更适合多人合作的大型软件项目。

面向对象的特性:

• 封装性

• 继承性

• 多态性

1.4 面向过程和面向对象的对比

面向过程

• 优点: 性能比面向对象高,适合跟硬件联系很紧密的东西,例如单片机就采用的面向过程编程。

• 缺点: 没有面向对象易维护、易复用、易扩展

面向对象

• 优点: 易维护、易复用、易扩展,由于面向对象有封装、继承、多态性的特性,可以设计出低耦合的系统,使系统更加灵活、更加易于维护

• 缺点: 性能比面向过程低

用面向过程的方法写出来的程序是一份蛋炒饭（蛋、饭、酱汁混在一起），而用面向对象写出来的程序是一份盖浇饭（蛋、饭、酱汁分离）。各有各的好，程序简单使用面向过程，程序复杂使用面向对象编程。

2. ES6中的类和对象

面向对象

面向对象更贴近我们的实际生活,可以使用面向对象描述现实世界事物。但是事物分为具体的事物和抽象的事物，抽象的(泛指的)，具体的(特指的)

面向对象的思维特点:

1. 抽取(抽象)对象共用的属性和行为组织(封装)成一个类(模板)
2. 对类进行实例化,获取类的对象

面向对象编程我们考虑的是有哪些对象,按照面向对象的思维特点,不断的创建对象、使用对象、指挥对象做事情.

2.1 对象

现实生活中:万物皆对象,对象是一个具体的事物,看得见摸得着的实物。例如一本书、一辆汽车、一个人可以是“对象”,一个数据库、一张网页、一个与远程服务器的连接也可以是“对象”。

在JavaScript中,对象是一组无序的相关属性和方法的集合,所有的事物都是对象,例如字符串、数值、数组、函数等。

对象是由属性和方法组成的:

- 属性:事物的特征,在对象中用属性来表示(常用名词)
- 方法:事物的行为,在对象中用方法来表示(常用动词)

2.2 类class

在ES6中新增加了类的概念,可以使用class关键字声明一个类,之后以这个类来实例化对象。

类抽象了对象的公共部分,它泛指某一大类(class)

对象特指某一个,通过类实例化一个具体的对象

2.3 创建类

语法:

```
class Name {  
    //class body  
}
```

创建实例:

```
var xx = new name();
```

注意:类必须使用new实例化对象

2.4 类constructor构造函数

constructor()方法是类的构造函数(默认方法),用于传递参数,返回实例对象,通过new命令生成对象实例时,生成对象示例时自动调用该方法。如果没有显示定义,类内部会自动给我们创建一个**constructor()**

示例:

```
class Star {  
    constructor(uname, age) {  
        this.name = uname  
        this.age = age  
    }  
}  
var ldh = new Star('刘德华', 18)  
console.log(ldh.name);  
var zxy = new Star('张学友', 20)  
console.log(zxy);
```

刘德华
▶ Star {name: "张学友", age: 20}

注意:

- 1.通过class关键字创建类,类名习惯性定义首字母大写
- 2.类里面有个constructor函数,可以接受传递过来的参数,同时返回实例对象,所以不需要return
- 3.constructor函数只要new生成实例时,就会自动调用这个函数,如果我们不写这个函数,类也会自动生成这个函数
- 4.**生成实例new不能省略**
- 5.最后注意语法规规范,创建类类名后面不要加小括号,生成实例类名后面加小括号,构造函数不需要加function

2.5类添加方法

语法:

```
class Person {  
    constructor (name, age) { // constructor 构造器或者构造函数  
        this.name = name ;  
        this.age = age ;  
    }  
    say() {  
        console.log (this.name + '你好') ;  
    }  
}
```

示例:

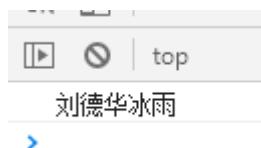
```
class Star {  
    //类的共有属性放到constructor内  
    constructor(uname, age) {  
        this.name = uname  
        this.age = age  
    }  
}
```

```

sing(song) {
    // console.log('你好');
    console.log(this.name + song);
}
}

//2.利用类创建对象new
var ldh = new Star('刘德华', 18)
var zxy = new Star('张学友', 20)
// 类里面的所有函数不需要写function
// 多个函数方法之间不需要添加逗号分隔
ldh.sing('冰雨')

```



注意：

- 1.类里面的所有函数不需要写function
- 2.多个函数方法之间不需要添加逗号分隔

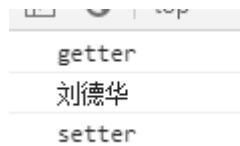
2.6类的get和set

注意get和set的声明和调用格式

```

class Star {
    constructor(name, age) {
        this.name = name,
        this.age = age
    }
    get fans() {
        console.log('getter');
        return this.name
    }
    // set里必须写有参数
    set audience(val) {
        console.log('setter');
    }
}
let ldh = new Star('刘德华', 18)
// 注意get和set的调用，后面不能加()
console.log(ldh.fans);
// set调用需要赋值一个参数
ldh.audience = '观众'

```



3.ES6的类的继承

3.1继承

现实中的继承:子承父业,比如我们都继承了父亲的姓。

程序中的继承:子类可以继承父类的一些属性和方法。**extends关键字**

语法:

```
class Father{ // 父类
}
class Son extends Father { // 子类继承父类
}
```

3.2 super关键字

super关键字用于访问和调用对象父类上的函数。可以调用父类的构造函数,也可以调用父类的普通函数。

super()是父类构造方法,只在创建对象时调用,可以通过super对象调用父类的方法。

可以实现子类的constructor函数里的参数传入到父类的constructor函数,就可以使用父类方法中的一些数据。

示例1: 调用父类的构造函数

```
class Father {
    constructor(x, y) {
        this.x = x
        this.y = y
    }
    /* money() {
        console.log(100);
    } */
    sum() {
        console.log(this.x + this.y);
    }
}
class Son extends Father {
    constructor(x, y) {
        /* this.x = x
        this.y = y */
        super(x, y)//调用了父类中的构造函数
    }
}
var son = new Son(1, 2);
// son.money();//直接调用父类的方法
son.sum()//报错,原因:父类里的方法的this.x this.y this指的是父亲对象,而实例化子类中的参数传给的是子类,里面的this指向的是儿子对象,因此父类中不能得到这两个参数
//一句话:必须是父类传过来的参数x y,数才能相加。所以需要将子类中的x y传入父类--super关键字
```

3

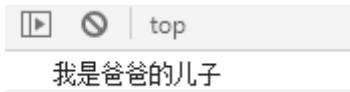
示例2: 调用父类的普通函数

```
class Father {
    say() {
        return '我是爸爸';
    }
}
```

```

        }
    }
    class Son extends Father {
        say() {
            // console.log('我是儿子');
            // super.say() 就是调用父亲中的普通函数say()
            console.log(super.say() + '的儿子');
        }
    }
    var son = new Son()
    son.say()

```



继承中的属性或者方法查找原则:就近原则

1. 继承中，如果实例化子类输出一个方法，先看子类有没有这个方法，如果有就先执行子类的
2. 继承中，如果子类里面没有，就去查找父类有没有这个方法，如果有，就执行父类的这个方法

示例3：子类继承父类方法同时扩展自己的方法

```

//父类有加法
class Father {
    constructor(x, y) {
        this.x = x
        this.y = y
    }
    sum() {
        console.log(this.x + this.y);
    }
}
//子类继承父类加法方法，同时扩展减法
class Son extends Father {
    constructor(x, y) {
        //利用super调用父类的构造函数
        //super必须在子类this之前调用
        super(x, y)
        this.x = x
        this.y = y
    }
    subtract() {
        console.log(this.x - this.y);
    }
}
var son = new Son(5, 3)//参数传入实例化对象内，不是方法内
son.subtract() //2
son.sum() //8

```

注意：子类在构造函数中使用super，必须放到this前面（必须先调用父类的构造方法，在使用子类构造方法）

3.3 使用类的注意点

1.在ES6中类没有变量提升，所以必须先定义类，才能通过类实例化对象

2.类里面的共有的属性和方法一定要加this使用.

3.类里面的this指向问题.

4.**constructor** 里面的this指向实例对象，方法里面的this指向这个方法的调用者，谁调用指向谁

示例:

```
var that;
class Star {
    constructor(uname, age) {
        //constructor 里面的this指向的是 创建的示例对象 如lhd
        that = this //将this赋值给that,让that也指向示例对象
        //uname age是传入的参数
        this.name = uname
        this.age = age
        // sing() 没有这个方法
        // this.sing //只有对象有方法，调用对象里的方法
        //实现点击按钮，就调用sing()方法，因此按钮需要绑定对象
        this.btn = document.querySelector('button')
        this.btn.onclick = this.sing //不需要加()调用，若加了，示例化后就立马调用，而非点击后再调用
    }
    sing() {
        //方法里没有构造函数里的参数
        // console.log(uname); 会报错，没有uname这个参数，所以必须加this，输出对象的属性
        // console.log(this.name); //注意调用该方法的对象是btn，这个this指向btn，而btn.name并没有定义，所以出现undefined
        console.log(that.name); //仍想输出实例对象的name，通过全局变量that实现，在构造函数内将that指向this即实例对象
    }
    dance() {
        console.log(this); //lhd调用该方法，指向lhd实例对象
    }
}
//EC6中没有变量提升，所以必须先定义类，后实例化
var lhd = new Star('刘德华', 18)
// lhd.sing()
lhd.dance()
```

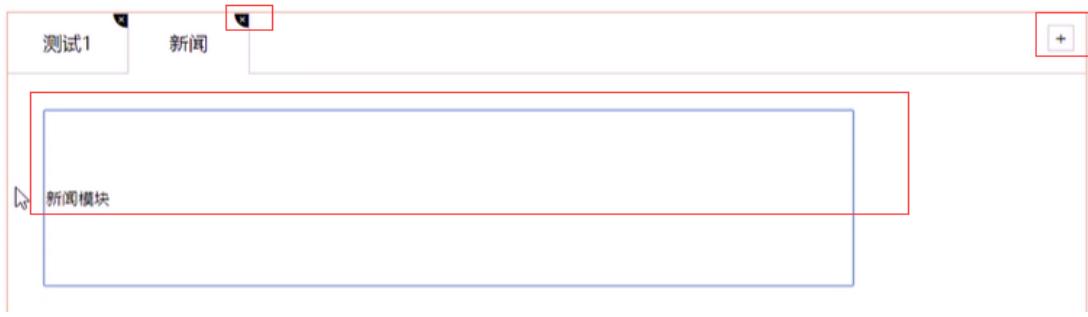
```
▼ Star {name: "刘德华", age: 18, btn: button} ⓘ
  age: 18
  ▶ btn: button
  name: "刘德华"
  ▶ __proto__: Object
```

刘德华

说明: dance()方法内的this指向实例化对象lhd，点击按钮后输出“刘德华”

注意: 定义that全局变量，将this赋值给that,让that也指向示例对象，保证that始终指向实例对象，可以随时使用that调用示例对象里的方法

案例：面向对象tab栏



功能需求:

- 1.点击tab栏可以切换效果.
- 2.点击+号,可以添加tab项和内容项.
- 3.点击x号,可以删除当前的tab项和内容项,
- 4.双击tab项文字或者内容项文字, 可以修改里面的文字内容.

```

<div class="box">
    <h4>
        js面向对象 动态添加标签页
    </h4>
    <div class="tab-box" id="tab">
        <!-- tab 标签 -->
        <div class="tab-list">
            <ul>
                <li class="liactive">测试1</li>
                <li>测试2</li>
            </ul>
            <div class="tab-add">
                <span>+</span>
            </div>
        </div>
        <!-- tab 内容 -->
        <div class="tab-con">
            <section class="conactive">测试1模块</section>
            <section>测试2模块</section>
        </div>
    </div>
    <script src="js/tab.js"></script>

```

抽象对象: Tab对象

- 1.该对象具有切换功能
- 2.该对象具有添加功能
- 3.该对象具有删除功能
- 4.该对象具有修改功能

1.该对象具有切换功能

注意：

1.this的所指对象，方法内：谁调用指向谁

2.使用一个全局变量that，指向示例对象，因为this有时候不指向示例对象，而我们需要使用实例对象下面的方法和属性

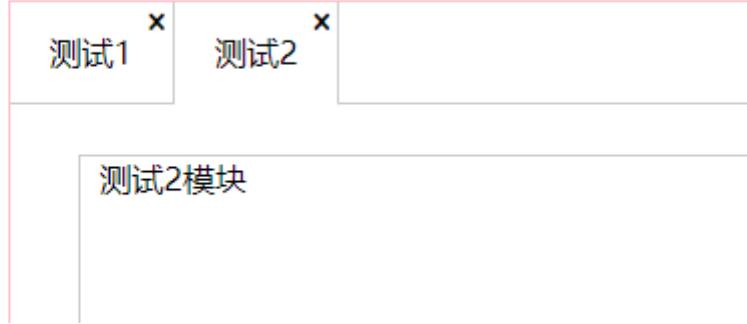
3.使用classList添加和移除类实现tab-list和tab-cont切换，注意移除类是在that下移除，this指向当前选中的li，无法找到其他元素；也注意是先移除类后添加类，否则刚添加完的类就被移除了。

选择有选中状态（有.liactive）的类进行移除.liactive类，对当前li添加类。

完整代码

```
var that;
class Tab {
    constructor(id) {
        that = this
        // 获取元素
        this.box = document.querySelector(id)
        // 获取选项卡和内容模块
        this.lis = this.box.querySelectorAll('li')
        this.sections = this.box.querySelectorAll('section')
        this.init()
    }
    init() {
        // 初始化操作，让相关的元素绑定事件
        for (var i = 0; i < this.lis.length; i++) {
            // 切换功能，给选项卡li绑定事件
            this.lis[i].index = i
            // 切换功能应放入toggleTab方法内，在此处调用toggleTab方法
            this.lis[i].onclick = this.toggleTab
        }
    }
    // 1.切换功能
    toggleTab() {
        // 注意此时的this指向上面的调用者lis[i]，当前选中的li
        // 使用classList添加和移除类，注意移除类是在that下移除，this中只有当前选中的li，无法找到其他元素
        that.box.querySelector('.liactive').classList.remove('liactive')
        this.classList.add('liactive')
        that.box.querySelector('.conactive').classList.remove('conactive')
        that.sections[this.index].classList.add('conactive')
    }
}

new Tab('#tab')
```



2.面向对象版tab栏切换添加功能

案例分析

- 1.点击+可以实现添加新的选项卡和内容
- 2.第一步:创建新的选项卡li和新的内容section
- 3.第二步:把创建的两个元素追加到对应的父元素中.

4.以前的做法:动态 createElement ,但是元素里面内容较多,需要innerHTML赋值, 再 appendChild追加到父元素里面.

5.现在高级做法:利用insertAdjacentHTML()可以直接把字符串格式元素添加到父元素中, 无需创建元素, 直接添加

6.appendChild不支持追加字符串的子元素, insertAdjacentHTML支持追加字符串的元素

关键点

1.insertAdjacentHTML()

语法:

```
element.insertAdjacentHTML(position, text);
```

position表示插入内容相对于元素的位置, 并且必须是以下字符串之一:

- 'beforebegin': 元素自身的前面。
- 'afterbegin': 插入元素内部的第一个子节点之前。
- 'beforeend': 插入元素内部的最后一个子节点之后。
- 'afterend': 元素自身的后面

位置名称的可视化

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  foo
  <!-- beforeend -->
</p>
<!-- afterend -->
```

2.问题：新添加的选项卡和内容不具有切换功能。

原因：因为我们是动态添加元素。如下代码块，一开始我们就获取选项卡和内容模块，而此时里面不包括新添加的选项卡和内容，后面初始化给li绑定事件也就没能给新增的li绑定事件

```
constructor(id) {
  that = this
  // 获取元素
  this.box = document.querySelector(id)
  // 获取选项卡和内容模块
  this.lis = this.box.querySelectorAll('li')
  this.sections = this.box.querySelectorAll('section')
  this.init()
}
```

解决方法：每一次初始化(init())都要重新获得所有li和section，再给当前所有的li绑定事件。而点击+号新增完内容后再初始化，获得动态添加后所有的元素，让动态添加的li也绑定事件。

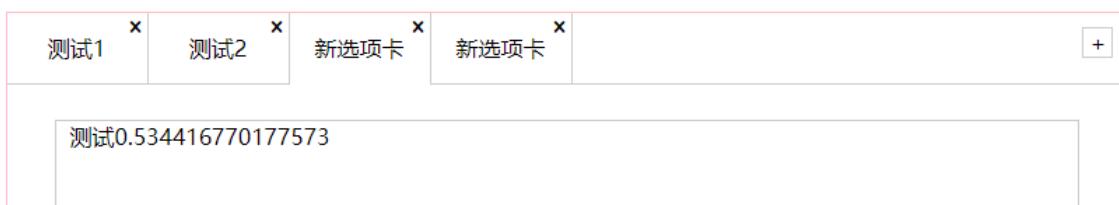
```
init() {
  // 切换功能 获取选项卡和内容模块
  this.lis = this.box.querySelectorAll('li')
  this.sections = this.box.querySelectorAll('section')

  // 初始化操作，让相关的元素绑定事件
  // 切换事件绑定li
  for (var i = 0; i < this.lis.length; i++) {
    // 切换功能，给选项卡li绑定事件
    this.lis[i].index = i
    // 切换功能应放入toggleTab方法内，在此处调用toggleTab方法
    this.lis[i].onclick = this.toggleTab
  }
  // 添加事件绑定+
  this.add.onclick = this.addTab
}
```

4.细节：添加选项后，让其默认选中新添加的选项卡和内容，新添加的选项卡和内容默认处于被选中状态，那么需要清除其他选项卡和内容的选中状态

完整代码

```
// 2.添加功能
addTab() {
    // 新添加的选项卡和内容默认处于被选中状态，那么需要清除其他选项卡和内容的选中状态
    that.removeClass()
    var random = Math.random()
    // 1)创建新的选项卡li和新的内容section
    var li = '<li class="liactive"><span>新选项卡</span></li>'
    var section = '<section class="conactive">测试<span class="icon-cross">' +
        random + '</section>'
    // 2)把创建的两个元素追加到对应的父元素中
    that.ul.insertAdjacentHTML('beforeend', li)
    that.section.insertAdjacentHTML('beforeend', section)
    // 初始化，让新增的li绑定事件
    that.init()
}
```



3.面向对象版 tab 样切换删除功能

案例分析

- 1.点击x可以删除当前的li选项卡和当前的section
- 2.X是没有索引号的,但是它的父亲li有索引号,这个索引号正是我们想要的索引号
- 3.所以核心思路是:点击x号可以删除这个索引号对应的li和section

关键点:

1.删除x在li里面,因此动态添加tab-list时也会动态添加x,所以需要和li/section一样在初始化内重新获取元素,同时可以在绑定li事件中给x绑定删除事件

```
init() {
    // 因为我们是动态添加元素, 所以需要重新获得对应的元素
    // 切换功能 获取选项卡和内容模块
    this.lis = this.box.querySelectorAll('li')
    this.sections = this.box.querySelectorAll('section')
    // 1.删除功能, 获得x号
    this.dels = this.ul.querySelectorAll('.icon-cross')

    // 初始化操作, 让相关的元素绑定事件
    // 切换事件绑定li
    for (var i = 0; i < this.lis.length; i++) {
        // 切换功能, 给选项卡li绑定事件, 并添加索引号
        this.lis[i].index = i
        // 切换功能应放入toggleTab方法内, 在此处调用toggleTab方法
        this.lis[i].onclick = this.toggleTab
        // 1.删除功能, 给x绑定事件, 调用删除方法
        this.dels[i].onclick = this.removeTab
    }
}
```

```
}
```

2.注意尽量不要将x字体图片通过伪元素放置在li的后面，目前不会使用js选中li的伪元素，还是建议在li里面插入一个span盒子放入伪元素，通过span的类名选中伪元素，并为伪元素绑定事件。

```
<li class="liactive">测试1 <span class="icon-cross"></span></li>
```

3.使用remove()方法直接删除指定的元素

细节及功能完善：

1.在删除方法内添加e.stopPropagation()//阻止冒泡，防止冒泡触发li的切换事件

2.删除选项卡后选项卡数量改变，需要重新获取当前的li的个数，即init()初始化

3.删除选项卡后，让其默认选中删除li的前一个选项卡，做法：手动调用触发点击事件，而不是为前一个li和section添加类，这样更方便。利用与逻辑中断实现，删除第一个选项卡时不执行点击事件

```
index--  
// 注意：删除第一个选项卡时，没有前一个选项卡，则不触发点击事件，利用与逻辑中断实现  
that.lis[index] && that.lis[index].click()
```

4.删除非选中状态的li时，让原来选中状态的li保持不变。做法：再删除完毕当前选中的li之后，默认选择删除前一个选项卡之前，添加一个判断条件，判断当前是否有.liactive类（即判断是否有被选中的li），如果有选中状态的li则使用return结束代码，不执行后面的默认点击事件。

```
if (document.querySelector('.liactive')) return false;
```

注意直接写return无法结束后面代码，最好返回false或空字符等。

5.小bug：删除全部选项卡后，再进行添加选项卡时，添加功能里的清除类报错，解决方法：需要在添加功能移除类之前添加一个判断条件，如果有选项卡即li个数不为0情况下才清除选项卡和内容的类。

```
if (that.lis.length != 0) {  
    that.removeClass()  
}
```

完整代码

```
// 3.删除功能  
removeTab(e) {  
    e.stopPropagation() //阻止冒泡，防止触发li的切换事件  
    var index = this.parentNode.index  
    // console.log(index);  
    // 删除当前索引号对应的li和section，remove()方法可以直接删除指定的元素  
    that.lis[index].remove()  
    that.sections[index].remove()  
    // 删除后，选项卡个数改变，重新获得当前li数量，及初始化  
    that.init()  
    // 删除后，让其默认选中删除li的前一个选项卡  
    // 如果我们删除的不是选中状态的li，那么让原来选中状态的li保持不变，即如果有选中状态的  
    // li则使用return结束代码，不执行后面的点击事件  
    if (document.querySelector('.liactive')) return false;  
    // 通过手动调用触发点击事件，而不是为前一个li和section添加类，这样更方便  
    index--
```

```
// 注意：删除第一个选项卡时，没有前一个选项卡，则不触发点击事件，利用与逻辑中断实现  
that.lis[index] && that.lis[index].click()  
}
```



4. 面向对象版tab栏切换编辑功能

案例分析

1. 双击选项卡li或者 section里面的文字，可以实现修改功能

2. 双击事件是: ondblclick

3. 如果双击文字，会默认选定文字，此时需要双击禁止选中文字

4. 禁止选中文字代码：会使用即可，不用记

```
window.getSelection ? window.getSelection().removeAllRanges():  
document.selection.empty();
```

5. 核心思路：双击文字的时候，在里面生成一个文本框，当失去焦点或者按下回车然后把文本框输入的值给原先元素即可

关键点：

1. 动态添加选项卡和内容时，也会动态添加选项卡里面的内容，因此也需要动态获取li和section里面的文字元素，将获取元素放在初始化里面，并给它们绑定事件

2. 双击文字的时候，在里面生成一个文本框，并将选中元素的值放入文本框内

```
// 1) 双击文字的时候，在里面生成一个文本框  
// 将选中的选项卡内容放入input表单内  
var str = this.innerHTML;  
// this指向当前选中的span，在span里面插入input表单  
this.innerHTML = '<input type="text">'  
var input = this.children[0]  
input.value = str;
```

3. 当失去焦点时将文本框输入的值赋值给原先的元素，回车时也可以实现，同理。函数里手动调用表单失去焦点事件，实现赋值，更方便

```

// 失去焦点即离开文本框时，将文本框里面的值给span
input.onblur = function () {
    // this指代input, span是input的父级
    this.parentNode.innerHTML = this.value
}

// 按下回车键时也可以将文本框里面的值给span
input.onkeyup = function (e) {
    if (e.keyCode == 13) {
        // 手动调用表单失去焦点事件，更方便，也可以复制上面的代码
        this.blur()
    }
}

```

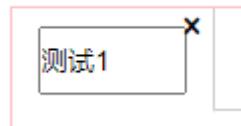
4.选项卡和下面模块内容部分修改操作一样。

细节：

1.由于需要获取li里面的文字，所以最好使用span等标签放入文字，这样方便获取元素

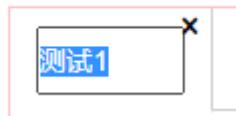
2.双击文字出现的文本框，**使用select()让里面的文字默认处于全部选中状态**，这样方便用户更改选项卡。如下图，

未设置选中前：



设置选中后：

```
input.select() // 文本框里面的文字处于选中状态
```



完整代码

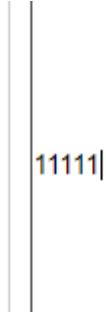
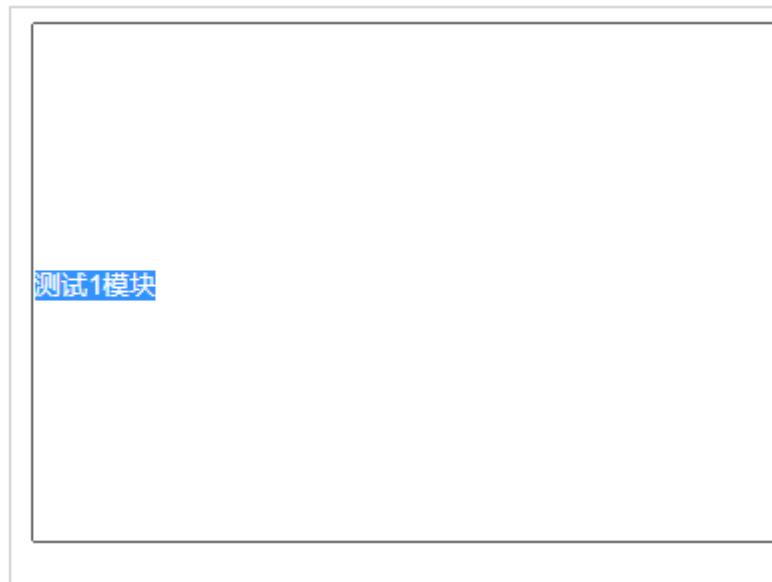
```

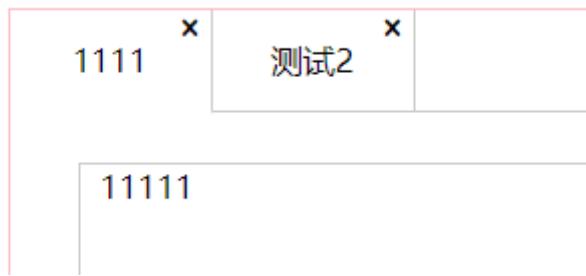
for (var i = 0; i < this.lis.length; i++) {
    // 切换功能，给选项卡li绑定事件，并添加索引号
    this.lis[i].index = i
    // 切换功能应放入toggleTab方法内，在此处调用toggleTab方法
    this.lis[i].onclick = this.toggleTab
    // 删除功能，给x绑定事件，调用删除方法
    this.dels[i].onclick = this.removeTab
    // 编辑功能，给选项卡里的文字与下面的内容模块绑定事件，调用编辑方法
    this.spans[i].ondblclick = this.updateTab
    this.sections[i].ondblclick = this.updateTab
}

// 4.修改功能
updateTab() {
    // 禁止选中文字
    window.getSelection ? window.getSelection().removeAllRanges() :
    document.selection.empty();
    // 1) 双击文字的时候，在里面生成一个文本框
}

```

```
// 将选中的选项卡内容放入input表单内
var str = this.innerHTML;
// this指向当前选中的span, 在span里面插入input表单
this.innerHTML = '<input type="text">'
var input = this.children[0]
input.value = str;
input.select() //文本框里面的文字处于选中状态
// 失去焦点即离开文本框时, 将文本框里面的值给span
input.onblur = function () {
    // this指代input, span是input的父级
    this.parentNode.innerHTML = this.value
}
// 按下回车键时也可以将文本框里面的值给span
input.onkeyup = function (e) {
    if (e.keyCode == 13) {
        // 手动调用表单失去焦点事件, 更方便, 也可以复制上面的代码
        this.blur()
    }
}
```





案例小结

1. 使用全局变量that确保指向示例对象
2. 在constructor函数内进行获取元素，一般获取的元素是那些对象功能不会影响元素个数即元素个数不太会变化。
3. 需要一个初始化函数 init() 初始化操作，让相关的元素绑定事件即调用对象的相关功能函数，有需要的话可以利用该函数重新获得当前的元素（主要是那些会因为增删改查改变元素个数的那些元素，我们需要获得最新的元素个数）

全部js代码

```
var that;
class Tab {
    constructor(id) {
        that = this
        // 获取元素
        this.box = document.querySelector(id)
        // 添加功能 获取+号, li的父元素ul, section的父元素
        this.add = this.box.querySelector('.tab-add')
        this.ul = this.box.querySelector('.tab-list ul:first-child')
        this.section = this.box.querySelector('.tab-con')
        this.init()
    }
    init() {
        // 因为我们是动态添加元素，所以需要重新获得对应的元素
        // 切换功能 获取选项卡和内容模块元素
        this.lis = this.box.querySelectorAll('li')
        this.sections = this.box.querySelectorAll('section')
        // 删除功能，获得x号元素
        this.dels = this.box.querySelectorAll('.icon-cross')
        // 编辑功能，获得文字元素
        this.spans = this.box.querySelectorAll('.tab-list li span:first-child')

        // 初始化操作，让相关的元素绑定事件
        // 切换事件绑定li
        for (var i = 0; i < this.lis.length; i++) {
            // 切换功能，给选项卡li绑定事件，并添加索引号
            this.lis[i].index = i
            // 切换功能应放入toggleTab方法内，在此处调用toggleTab方法
            this.lis[i].onclick = this.toggleTab
            // 删除功能，给x绑定事件，调用删除方法
            this.dels[i].onclick = this.removeTab
            // 编辑功能，给选项卡里的文字与下面的内容模块绑定事件，调用编辑方法
            this.spans[i].ondblclick = this.updateTab
            this.sections[i].ondblclick = this.updateTab
        }
    }
}
```

```

// 添加事件绑定+号
this.add.onclick = this.addTab
}

// 清除选项卡和内容的选中状态
removeClass() {
    that.box.querySelector('.liactive').classList.remove('liactive')
    that.box.querySelector('.conactive').classList.remove('conactive')
}

// 1.切换功能
toggleTab() {
    // 注意此时的this指向上面的调用者lis[i],当前选中的li
    // 使用classList添加和移除类，注意移除类是在that下移除，this中只有当前选中的li，无法找到其他元素
    // tab-list切换需要的添加和移除类
    that.removeClass()
    this.classList.add('liactive')
    // tab-con切换需要的添加和移除类
    that.sections[this.index].classList.add('conactive')
}

// 2.添加功能
addTab() {
    // 新添加的选项卡和内容默认处于被选中状态，那么需要清除其他选项卡和内容的选中状态
    that.removeClass()
    var random = Math.random()
    // 1)创建新的选项卡li和新的内容section
    var li = '<li class="liactive"><span>新选项卡</span><span class="icon-cross"></span></li>'
    var section = '<section class="conactive">测试' + random + '</section>'
    // 2)把创建的两个元素追加到对应的父元素中
    that.ul.insertAdjacentHTML('beforeend', li)
    that.fsection.insertAdjacentHTML('beforeend', section)
    // 初始化，让新增的li绑定事件
    that.init()
}

// 3.删除功能
removeTab(e) {
    e.stopPropagation() //阻止冒泡，防止触发li的切换事件
    var index = this.parentNode.index
    // console.log(index);
    // 删除当前索引号对应的li和section,remove()方法可以直接删除指定的元素
    that.lis[index].remove()
    that.sections[index].remove()
    // 删除后，选项卡个数改变，重新获得当前li数量，及初始化
    that.init()
    // 删除后，让其默认选中删除li的前一个选项卡
    // 如果我们删除的不是选中状态的li，那么让原来选中状态的li保持不变，即如果有选中状态的li则使用return结束代码，不执行后面的点击事件
    if (document.querySelector('.liactive')) return false;
    // 通过手动调用触发点击事件，而不是为前一个li和section添加类，这样更方便
    index--
    // 注意：删除第一个选项卡时，没有前一个选项卡，则不触发点击事件，利用与逻辑中断实现
    that.lis[index] && that.lis[index].click()
}

// 4.修改功能
updateTab() {
    // 禁止选中文字
    window.getSelection ? window.getSelection().removeAllRanges() :
    document.selection.empty();
}

```

```

// 1) 双击文字的时候，在里面生成一个文本框
// 将选中的选项卡内容放入input表单内
var str = this.innerHTML;
// this指向当前选中的span，在span里面插入input表单
this.innerHTML = '<input type="text">';
var input = this.children[0];
input.value = str;
input.select() //文本框里面的文字处于选中状态
// 失去焦点即离开文本框时，将文本框里面的值给span
input.onblur = function () {
    // this指代input, span是input的父级
    this.parentNode.innerHTML = this.value
}
// 按下回车键时也可以将文本框里面的值给span
input.onkeyup = function (e) {
    if (e.keyCode == 13) {
        // 手动调用表单失去焦点事件，更方便，也可以复制上面的代码
        this.blur()
    }
}
}

new Tab('#tab')

```

4.构造函数和原型

4.1概述

在典型的OOP的语言中(如Java), 都存在类的概念,类就是对象的模板,对象就是类的实例,但在ES6之前, JS中并没用引入类的概念。

ES6,全称ECMAScript6.0 , 2015.06发版。但是目前浏览器的JavaScript是ES5版本,大多数高版本的浏览器也支持ES6 ,不过只实现了ES6的部分特性和功能。

在ES6之前,对象不是基于类创建的,而是用一种称为构建函数的特殊函数来定义对象和它们的特征。

创建对象可以通过以下三种方式:

1.对象字面量

2.new Object()

3.自定义构造函数

4.2构造函数

构造函数是一种特殊的函数 主要用来初始化对象 ,即为对象成员变量赋初始值 ,它总与new一起使用。 我们可以把对象中一些公共的属性和方法抽取出来,然后封装到这个函数里面。

在JS中,使用构造函数时要注意以下两点:

1.构造函数用于创建某一类对象,其首字母要大写

2.构造函数要和**new 一起使用**才有意义

new在执行时会做四件事情:

①在内存中创建一个新的空对象。

②让this指向这个新的对象。

③执行构造函数里面的代码,给这个新对象添加属性和方法。

④返回这个新对象(所以构造函数里面不需要return)

JavaScript的构造函数中可以添加一些成员,可以在构造函数本身上添加,也可以在构造函数内部的this上添

加。通过这两种方式添加的成员,就分别称为**静态成员**和**实例成员**。

•静态成员:在构造函数本身上添加的成员称为**静态成员**,只能由**构造函数本身来访问**

•实例成员:在构造函数内部创建的对象成员称为**实例成员**,只能由**实例化的对象来访问**

示例:

```
// 构造函数中的属性和方法我们称为成员, 成员可以添加
function Star(uname, age) {
    this.uname = uname ;
    this.age = age;
    this.sing = function() {
        console.log('我会唱歌');
    }
}
var ldh = new Star('刘德华', 18);
// 1. 实例化成员是构造函数内部通过this添加的成员, 如uname age sing, 只能通过实例对象来访问
console.log(ldh.uname);
console.log(ldh.age); // 不可以通过构造函数来访问示例成员
// 2. 静态成员是构造函数本身上添加的成员, 如sex
Star.sex = '男'
// 静态成员只能通过构造函数来访问
console.log(Star.sex);
console.log(ldh.sex); // 不能通过对象来访问
```

刘德华
undefined
男
undefined



4.3构造函数的问题

构造函数方法很好用,但是存在**浪费内存的问题**。因为方法是复杂数据类型,每个对象的方法都需要重新开辟一个空间,如下图



```

function Star(uname, age) {
    this.uname = uname
    this.age = age
    this.sing = function () {
        console.log("唱歌");
    }
}
var ldh = new Star('刘德华', 18)
var zxy = new Star('张学友', 28)
console.log(ldh.sing == zxy.sing);

```

`false` 结果为false,因为两者比较的是地址是否相同,结果说明两者都方法存放在内存中地址不相同

我们希望所有的对象使用同一个函数,这样就比较节省内存,那么我们要怎样做呢?

4.4构造函数原型prototype

构造函数通过原型分配的函数是所有对象所共享的。

JavaScript规定,每一个构造函数都有一个prototype属性,指向另一个对象。注意这个prototype就是一个对象,这个对象的所有属性和方法,都会被构造函数所拥有。

我们可以把那些不变的方法,直接定义在prototype对象上,这样所有对象的实例就可以共享这些方法。

注意:

1.在构造函数中,里面this指向的是对象实例ldh

2.原型对象函数里面的this指向的也是实例对象ldh(具体指谁,调用了才知道,谁调用指向谁,一般调用原型对象的是实例对象)

示例:

```

function Star(uname, age) {
    this.uname = uname
    this.age = age
}
Star.prototype.sing = function () {
    console.log('唱歌');
}
var ldh = new Star('刘德华', 18)
var zxy = new Star('张学友', 28)
console.log(ldh.sing == zxy.sing);
console.dir(Star)

```

`true`

```

▼ f Star(uname, age) ⓘ
  arguments: null
  caller: null
  length: 2
  name: "Star"
  ► prototype: {sing: f, constructor: f}
  ▼ __proto__: f ()
    ► apply: f apply()
    arguments: (...)


```

结果为true,说明两个对象所调用的方法地址相

同;同时可以观察到prototype是{},是一个对象,里面存放了我们刚添加的sing函数

问答?

1.原型是什么?

一个对象,我们也称为prototype为原型对象。

2.原型的作用是什么?

共享方法。

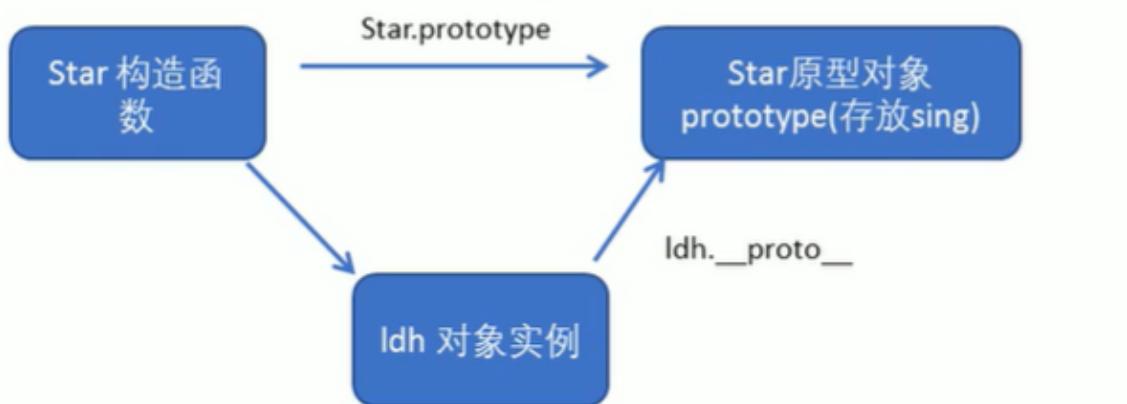
一般情况下,我们的公共属性定义到构造函数里面, 公共的方法我们放到原型对象身上

4.5 对象原型 `proto`

对象都会有一个属性 `__proto__`, 指向构造函数的 prototype 原型对象, 因此我们对象可以使用构造函数 prototype 原型对象的属性和方法, 就是因为对象有 `__proto__` 原型的存在。

- `__proto__` 对象原型和原型对象 prototype 是等价的
- `__proto__` 对象原型的意义就在于为对象的查找机制提供一个方向, 或者说一条路线, 但是它是一个非标准属性, 因此实际开发中, 不可以使用这个属性, 它只是内部指向原型对象 prototype

为了区分原型 prototype, 我们称这个为对象原型



示例:

```
console.log(ldh); // 每个对象上都有一个__proto__属性, 指向我们构造函数里的原型对象
console.log(ldh.__proto__ === Star.prototype); // 因为__proto__指向原型对象
prototype, 所以结果为true
```



注意方法的查找规则: 首先看实例对象 ldh 上是否有 sing 方法, 若有则执行对象上的 sing; 若没有, 则使用 `__proto__` 属性去构造函数原型对象 prototype 里查找 sing 方法。

4.6 `constructor` 构造函数

对象原型(`__proto__`) 和构造函数(`prototype`) 原型对象里面都有一个属性——`constructor` 属性, `constructor` 我们称为构造函数, 因为它指向构造函数本身。

`constructor` 主要用于记录该对象引用于哪个构造函数, 它可以让原型对象重新指向原来的构造函数。

示例:

```
console.log(ldh.__proto__);
console.log(Star.prototype);
console.log(ldh.__proto__.constructor);
console.log(Star.prototype.constructor);
```

The screenshot shows the browser's developer tools console output. It displays two objects: `ldh` and `Star`. The `ldh` object has a constructor function `Star`. The `Star` object has its own constructor function, which is also `Star`, indicating that the constructor property points back to the constructor function itself. This illustrates that the constructor property always points to the constructor function, even if it's part of the prototype chain.

```
▼ {constructor: f} ⓘ
  ► constructor: f Star(uname, age)
  ► __proto__: Object
  ▼ {constructor: f} ⓘ
    ► constructor: f Star(uname, age)
    ► __proto__: Object
    f Star(uname, age) {
      this.uname = uname
      this.age = age
    }
    f Star(uname, age) {
      this.uname = uname
      this.age = age
    }
```

都指向构造函数本身

```
// Star.prototype.sing = function () {
//   console.log('唱歌');
// }
// 当方法较多时，我们使用对象形式将方法放入到原型对象里
Star.prototype = {
  sing: function () {
    console.log('唱歌');
  },
  movie: function () {
    console.log('电影');
  }
}
console.log(ldh.__proto__.constructor);
console.log(Star.prototype.constructor);
```

The screenshot shows the browser's developer tools console output. It displays two objects: `ldh` and `Star`. The `ldh` object now has a `sing` method added via assignment. The `Star` object still has its original `sing` and `movie` methods. However, the `constructor` property for both objects now points to the `Object` constructor, as shown by the native code entries in the console. This demonstrates that using assignment to add methods to the prototype object does not preserve the original `constructor` property.

```
f Object() { [native code] }
f Object() { [native code] }
```

式将方法放入到原型对象，使用的是“`=`”，相当于是赋值（之前单个方法是通过`sing`来添加方法），将里面原有的`constructor`已经覆盖掉了，因此无法指向构造函数了，所以**我们需要手动的利用`constructor`这个属性指回构造函数**，如下代码块：

```
// 当方法较多时，我们使用对象形式将方法放入到原型对象里
Star.prototype = {
  // 很多情况下，我们需要手动的利用constructor这个属性指回构造函数
  constructor: Star,
  sing: function () {
    console.log('唱歌');
  },
  movie: function () {
    console.log('电影');
  }
}
```

```

}
console.log(ldh.__proto__);
console.log(Star.prototype);
console.log(ldh.__proto__.constructor);
console.log(Star.prototype.constructor);

```

```

{
  constructor: f,
  sing: f,
  movie: f
}

constructor: f Star(uname, age)
movie: f ()
sing: f ()
__proto__: Object

{
  constructor: f,
  sing: f,
  movie: f
}

constructor: f Star(uname, age)
movie: f ()
sing: f ()
__proto__: Object

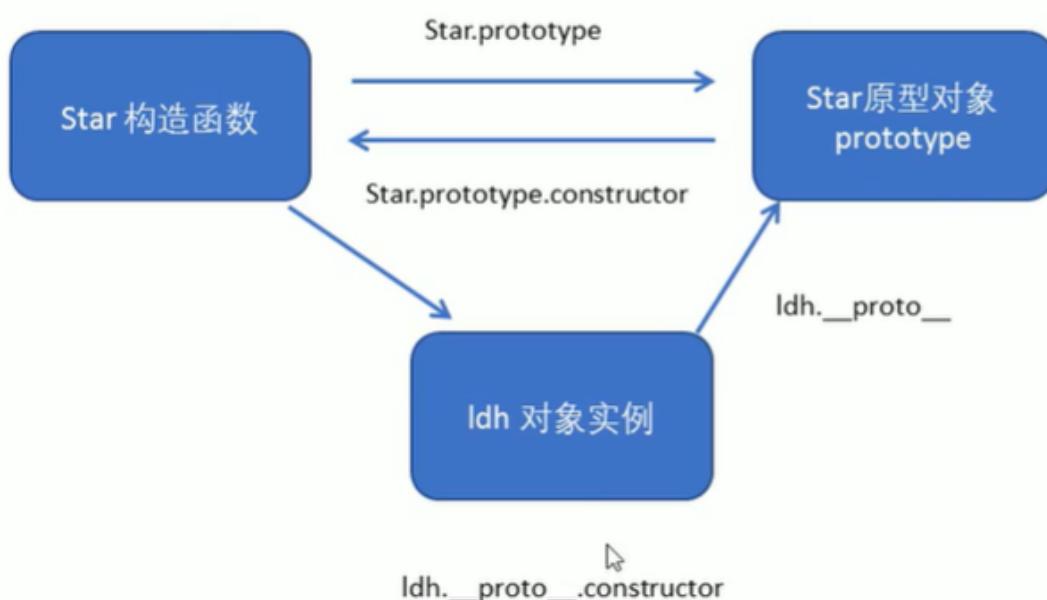
f Star(uname, age) {
  this.uname = uname
  this.age = age
}

f Star(uname, age) {
  this.uname = uname
  this.age = age
}

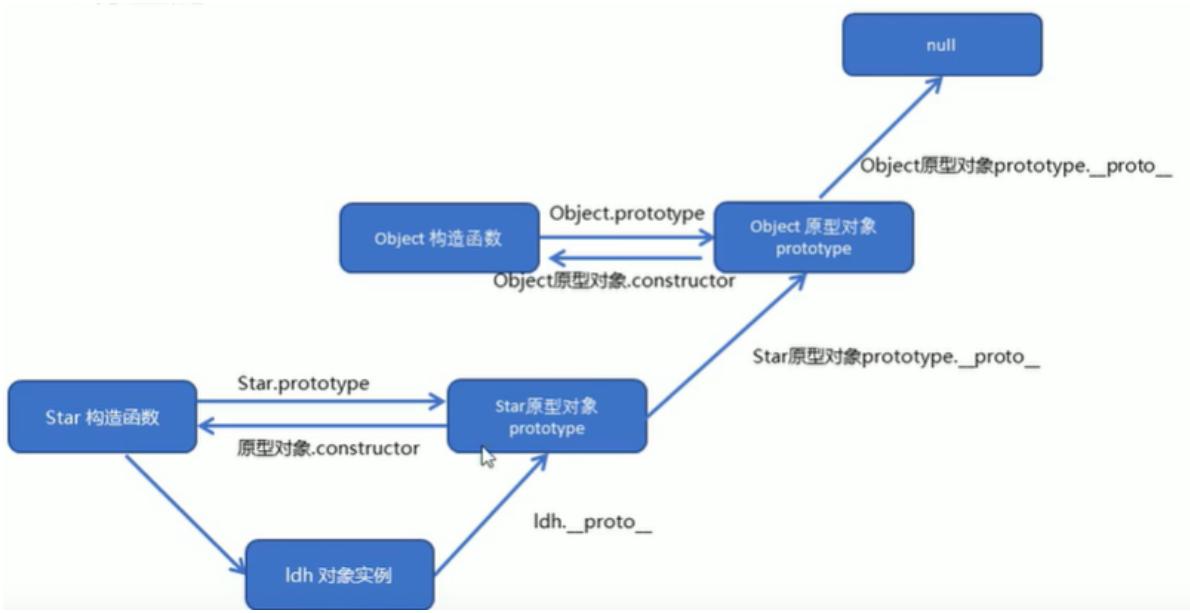
```

此时的原型对象和对象原型里包含constructor（指向构造函数）和我们添加的方法sing和movie

4.7 构造函数、实例、原型对象三者之间的关系



4.8 原型链



```
// 1.只要是对象就有__proto__ 原型，指向原型对象
console.log(Star.prototype);
console.log(Star.prototype.__proto__ == Object.prototype); //true
// 2.我们Star原型对象里面的__proto__ 原型指向的是object.prototype
console.log(Star.prototype.__proto__);
// 3.我们Object.prototype原型对象里面的__proto__原型指向为null
console.log(Object.prototype.__proto__);
```

```
▼ {sing: f, constructor: f} ⓘ
  ► sing: f ()
  ► constructor: f Star(uname, age)
    ► __proto__: Object
object.prototype
```

Star原型对象里面的__proto__ 原型指向的是

4.9 JavaScript的成员查找机制(规则)

- ①当访问一个对象的属性(包括方法)时,首先查找这个**对象自身**有没有该属性。
- ②如果没有就查找它的**原型**(也就是__proto__指向的**prototype原型对象**)。
- ③如果还没有就查找**原型对象的原型**(**Object的原型对象**)。
- ④依此类推一直找到**Object为止 (null)**。
- ⑤__proto__ 对象原型的意义就在于为对象成员查找机制提供一个方向,或者说一条路线。

因此实例对象可以使用Object上的**toString()**方法 (实例对象及其原型对象所没有的方法), 我没有该方法, 但我可以使用爸爸的爸爸的方法

4.10 扩展内置对象

可以通过**原型对象**,对原来的内置对象进行扩展自定义的方法。比如给数组增加自定义求偶数和的功能。

注意:数组和字符串内置对象不能给原型对象覆盖操作**Array.prototype= {}** (=赋值操作), 只能是**Array prototype.xxx = function(){} 的方式。(.的追加方法)**

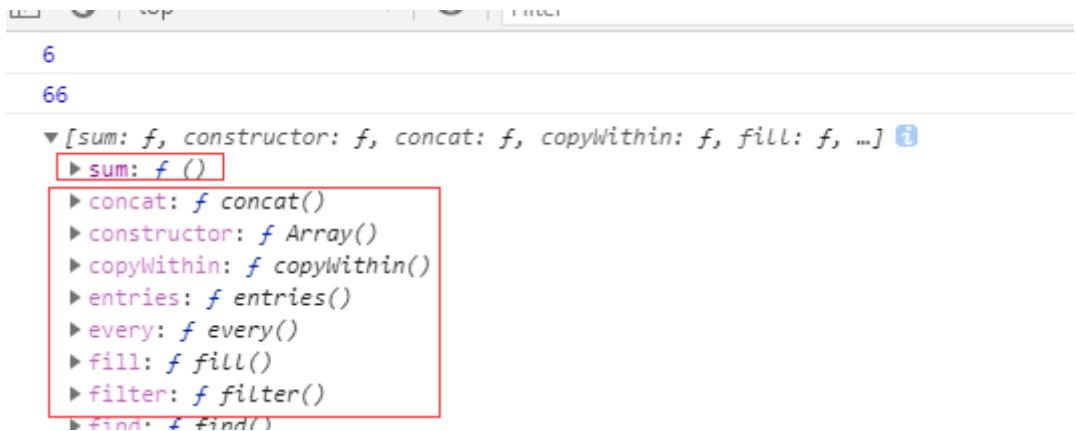
示例:

```
// 原型对象的应用 扩展内置对象方法
Array.prototype.sum = function () {
  var sum = 0
```

```

// this指向调用者
for (var i = 0; i < this.length; i++) {
    sum += this[i]
}
return sum
}
// 数组arr是一个对象，可以通过对象进行声明，因此具有原型对象
var arr1 = new Array(11, 22, 33)
var arr = [1, 2, 3]
console.log(arr.sum());
console.log(arr1.sum());
console.log(Array.prototype);

```



从结果可以发现，我们成功为数组对象添加了sum()求和方法，字体比数组对象自带的方法高亮些

5.ES5的继承

ES6之前并没有给我们提供extends继承。我们可以通过**构造函数+原型对象**模拟实现继承,被称为**组合继承**。

5.1 call()

使用该方法可以调用函数，也可以修改函数运行时的this指向

```
fun.call (thisArg, arg1, arg2, ...)
```

- thisArg :当前调用函数this的指向对象

- arg1 , arg2 :传递的其他参数,可以看作传递给函数的实参

示例:

```

function fn(a, b) {
    console.log('对于可控的事情，我们要保持谨慎');
    console.log(this);
    // console.log(a + b);
}
var o = {
    name: 'auue'
}
// 1.call()可以调用函数
fn.call()

```

对于可控的事情，我们要保持谨慎

► Window {window: Window, self: Window}

```
// 2.call()可以改变调用函数的this指向，修改后，由原来指向window，现指向o  
fn.call(o)
```

对于可控的事情，我们要保持谨慎

► {name: "auue"}

▶

```
// 3.call()可以传递参数  
fn.call(o, 1, 3)
```

对于可控的事情，我们要保持谨慎

► {name: "auue"}

4

▶

5.2 借用构造函数继承父类型属性

核心原理：通过call()把父类型的this指向子类型的this，这样就可以实现子类型继承父类型的属性。

```
Father.call(this, uname, age)
```

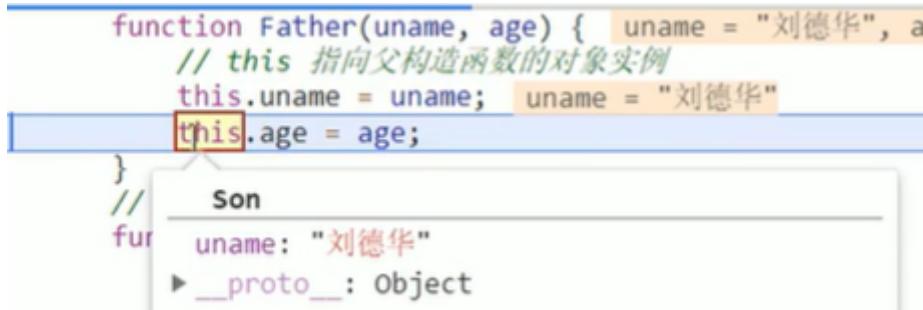
// 1.父构造函数

```
function Father(uname, age) {  
    // 这个this指向父构造函数的对象示例  
    this.uname = uname  
    this.age = age  
}
```

// 2.子构造函数

```
function Son(uname, age, score) {  
    // 这个this指向父构造函数的对象示例  
    // 通过调用父构造函数，并使用call()将父中的this指向子中的this，从而实现继承父的  
    // 属性  
    Father.call(this, uname, age)  
    this.score = score  
}  
var son = new Son('刘德华', 18, 100)  
console.log(son);
```

```
▼ Son {uname: "刘德华", age: 18, score: 100}  
  age: 18  
  score: 100  
  uname: "刘德华"  
  ► __proto__: Object
```



5.3 借用原型对象继承父类型方法

错误方法

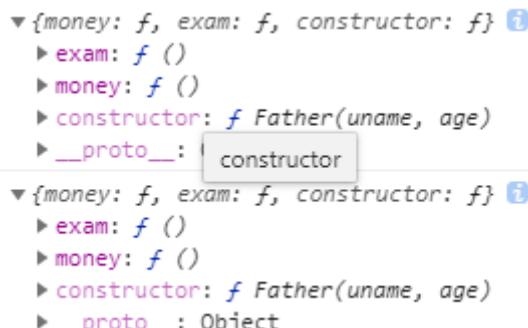
将父的原型对象赋值给子的原型对象

```

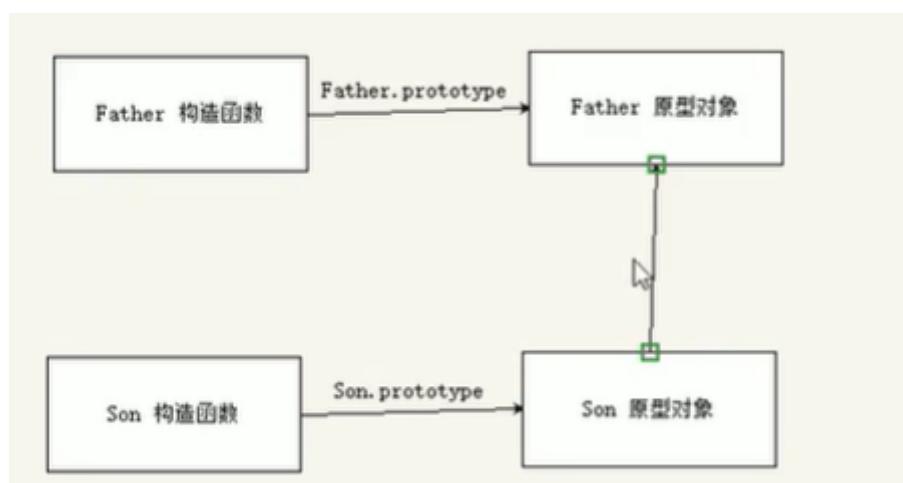
// 3.父构造方法
Father.prototype.money = function () {
    console.log('父金钱');
}
Son.prototype = Father.prototype
// 4.子构造方法
Son.prototype.exam = function () {
    console.log('子考试');
}

var son = new Son('刘德华', 18, 100)
console.log(Son.prototype);
console.log(Father.prototype);

```



虽然子继承了父的构造方法，但父也拥有了子的构造方法。原因是 `Son.prototype = Father.prototype`，让Son原型对象指向了Father原型对象，那么Son原型对象发生改变后，Father原型对象也会发生相应的改变



正确方法

```
// 通过让子原型对象指向父实例对象实现，而父实例对象的__proto__属性又指向父原型对象
Son.prototype=new Father()
// 注意：利用对象的形式修改了原型对象，别忘了利用constructor手动指回原来的构造函数
Son.prototype.constructor=Son

console.log(son);
console.log(Son.prototype.constructor);
console.log(Father.prototype);
```

```
▼ Son {uname: "刘德华", age: 18, score: 100} ⓘ
  age: 18
  score: 100
  uname: "刘德华"
  ▶ __proto__: Father
    age: undefined
    ▶ constructor: f Son(uname, age, score)
    ▶ exam: f ()
    uname: undefined
    ▶ __proto__:
      ▶ money: f ()
      ▶ constructor: f Father(uname, age)
      ▶ __proto__: Object

f Son(uname, age, score) {
  // 这个this指向父构造函数的对象示例
  // 通过调用父构造函数，并使用call()将父中的this指向子中的this，从而实现继承父的属性
  Father.call(this, uname, age)
  this.score = score
}

▼ {money: f, constructor: f} ⓘ
  ▶ money: f ()
  ▶ constructor: f Father(uname, age)
  ▶ __proto__: Object
```

使用一个中介对象——父实例对象实现，**让子原型对象指向父实例对象实现,而父示例对象的__proto__属性又指向父原型对象**。这样子原型对象可以使用父原型对象的方法（根据js成员查找机制和原型链），也不会影响父原型对象里的方法(Father实例对象与Father原型对象是两个不同的对象)。

注意：利用对象（赋值实例对象，实例对象也是对象，如此constructor会指向实例对象Father的构造函数）的形式修改了原型对象，别忘了利用constructor手动指回原来的构造函数



6.ES5中的新增方法

6.1 ES5新增方法概述

ES5中给我们新增了一些方法,可以很方便的操作数组或者字符串,这些方法主要包括:

- 数组方法
- 字符串方法
- 对象方法

6.2 数组方法

迭代(遍历)方法: `forEach()`、`map()`、`filter()`、`some()`、`every()`;

`map()`与`forEach()`类似, `every()`与`some()`类似

1. `forEach()`

```
array.forEach (function (currentValue, index, arr) {})
```

•`currentValue`:数组当前项的值, 可以省略

•`index`:数组当前项的索引, 可以省略

•`arr`:数组对象本身, 可以省略

参数用到哪个留哪个, 其他可以省略, 下面方法参数也一样

```

var arr = [1, 2, 3]
var sum = 0
arr.forEach(function (value, index, array) {
    console.log('每个数组元素' + value);
    console.log('每个数组元素的索引号' + index);
    console.log('数组本身' + array);
    sum += value
})
console.log(sum);

```

每个数组元素1
 每个数组元素的索引号0
 数组本身1,2,3
 每个数组元素2
 每个数组元素的索引号1
 数组本身1,2,3
 每个数组元素3
 每个数组元素的索引号2
 数组本身1,2,3
 6

2. filter()

```
array.filter(function (currentValue, index, arr){})
```

- `filter()` 方法创建一个新的数组,新数组中的元素是通过检查指定数组中符合条件的所有元素,主要用于**筛选数组**

- 注意它直接返回一个新数组

- `currentValue`: 数组当前项的值

- `index` :数组当前项的索引

- `arr`:数组对象本身

示例:

```

var arr = [11, 4, 9, 20]
var newArr = arr.filter(function (value, index) {
    // 筛选数组元素大于等于10的
    return value >= 10
})
console.log(newArr);

```

▼ Array(2) ⓘ
 0: 11
 1: 20
 length: 2
 ► __proto__: Array(0)

```
// 筛选数组元素为偶数的
return value % 2 == 0
```

3. some()

```
array.some (function (currentValue, index, arr) )
```

- some() 方法用于检测数组中的元素是否满足指定条件,通俗点**查找数组中是否有满足条件的元素**
- 注意它**返回值是布尔值**, 如果查找到这个元素, 就返回true,如果查找不到就返回false.
- 如果找到第一个满足条件的元素,则终止循环, 不再继续查找**, 因为找到后return就结束后面代码了
- currentValue: 数组当前项的值
- index :数组当前项的索引
- arr :数组对象本身

示例:

```
var arr = [11, 4, 9, 20]
var flag = arr.some(function (value) {
    // 查找数组中元素是否有大于等于20的
    return value >= 20
})
console.log(flag);
var arr1 = ['red', 'pink', 'blue']
var flag1 = arr1.some(function (value) {
    // 查找数组元素里是否有pink
    return value == 'pink'
})
console.log(flag1);
```

```
true
true
```

找到'pink'后, 就不再判断'pink' 后面的'blue'字符串是否相等。

filter与some的区别

- 1.filter也是查找满足条件的元素, **返回的是一个数组**, 而且是把**所有满足条件的元素返回回来**
- 2.some 也是查找满足条件的元素是否存在, **返回的是个布尔值**, 如果查找到第一个满足条件的元素就**终止循环**

查询商品案例

实现:

按照价格查询: - 按照商品名称搜索:

id	产品名称	价格
1	小米	3999
2	oppo	2999
3	华为	3499
4	荣耀	2499

1.把数据渲染到页面中 forEach

2.根据价格显示数据 filter

3.根据商品名称显示数据 some

数据存储的方式：数组+对象的形式

```
// 利用新增数组方法操作数据
var data = [
  {
    id: 1,
    name: '小米',
    price: '3999'
  }, {
    id: 2,
    name: 'oppo',
    price: '2999'
  }, {
    id: 3,
    name: '华为',
    price: '3499'
  }, {
    id: 4,
    name: '荣耀',
    price: '2499'
  }
]
```

1.把数据渲染到页面中

```
// 获取元素
var tbody = document.querySelector('tbody')
// 1.把数据渲染到页面中
// 遍历对象数组动态添加数据
data.forEach(function (value) {
  var tr = document.createElement('tr')
  tr.innerHTML = '<td>' + value.id + '</td><td>' + value.name + '</td>
<td>' + value.price + '</td>'
  tbody.append(tr)
})
```

按照价格查询: - 按照商品名称搜索:

id	产品名称	价格
1	小米	3999
2	oppo	2999
3	华为	3499
4	荣耀	2499

2.根据价格显示数据 filter

筛选数组后也需要渲染页面，因此可以把这部分封装成函数。

注意：

1.渲染前需要清空tbody内的数据

2.页面打开就需要初始渲染页面

```
// 初始渲染页面
setData(data)
function setData(myData) {
    // 渲染前需要清空tbody内的数据
    tbody.innerHTML = ''
    // 遍历对象数组动态添加数据
    myData.forEach(function (value) {
        var tr = document.createElement('tr')
        tr.innerHTML = '<td>' + value.id + '</td><td>' + value.name +
        '</td><td>' + value.price + '</td>'
        tbody.append(tr)
    })
}
// 2.根据价格查询商品
// 当我们点击了按钮，就可以根据我们的商品价格去筛选数组里面的对象
var start = document.querySelector('.start')
var end = document.querySelector('.end')
var search_price = document.querySelector('.search-price')
search_price.addEventListener('click', function () {
    // 使用filter筛选数组
    var newData = data.filter(function (value) {
        return value.price >= start.value && value.price <= end.value
    })
    // console.log(newData);
    // 渲染数据到页面
    setData(newData)
})
```

按照价格查询： - 按照商品名称搜索：

id	产品名称	价格
2	oppo	2999
3	华为	3499
4	荣耀	2499

3.根据商品名称显示数据

注意：

1.如果查询数组中唯一的元素，使用some方法更合适，找到后就不再循环，效率更高

2.因为some方法返回的是布尔值，所以使用数组输出找到的对象

```
// 3.根据商品名称查找商品
// 如果查询数组中唯一的元素，使用some方法更合适，找到后就不再循环，效率更高
var product = document.querySelector('.product')
```

```

var search_product = document.querySelector('.search-product')
search_product.addEventListener('click', function () {
    var arr = []
    data.some(function (value) {
        if (value.name == product.value) {
            arr.push(value) //使用数组存放找到的对象
        }
    })
    // 把拿到的数据渲染到页面
    setData(arr)
})

```

按照价格查询: - 按照商品名称搜索:

id	产品名称	价格
1	小米	3999

forEach与some的区别

```

var arr = ['green', 'pink', 'red']
arr.forEach(function (value) {
    if (value == 'pink') {
        console.log('find');
        return true // forEach里面的return不会终止迭代
    }
    console.log(11);
})
console.log('_____');
arr.some(function (value) {
    if (value == 'pink') {
        console.log('find');
        return true //some里面遇到return会终止迭代，效率更高
    }
    console.log(11);
})

```

6.3字符串方法

trim()方法会从一个字符串的两端删除空白字符（字符串中间的空白字符不会删去）。

trim()方法并不影响原字符串本身,它返回的是一个新的字符串。

`str.trim()`

示例：

```
var str = ' a ue '
var str1 = str.trim() //trim返回的是一个新字符串
console.log(str);
console.log(str1);
// 应用，更严谨的判断用户输入的内容，排除没用的空格输入
var input = document.querySelector('input')
var btn = document.querySelector('button')
btn.addEventListener('click', function () {
    var str = input.value.trim()
    if (str == '') {
        alert('请输入内容')
    } else {
        console.log(input.value);
        console.log(str.length);
        alert(str)
    }
})
```



可以应用于更严谨的判断用户输入的内容，只输入空格无效，并可以去除用户在字符串两侧输入的空格。

6.4 对象方法

1. Object.keys() 用于获取对象自身所有的属性

```
Object.keys(obj)
```

• 效果类似 for...in

• 返回一个由属性名组成的数组，可以结合 forEach 进行遍历输出

示例：

```
var data = {
    id: 1,
    name: '小米',
    price: '3999',
    num: 1111
}
var arr = Object.keys(data)
console.log(arr);
arr.forEach(function (value) {
    console.log(value);
})
```

```
▶ (4) ["id", "name", "price", "num"]
  id
  name
  price
  num
  >
```

2.Object.defineProperty()定义对象中新属性或修改原有的属性

```
Object.defineProperty(obj, prop, descriptor)
```

- obj: 必需。目标对象
- prop: 必需。需定义或修改的属性的名字，如果对象里没有这个名字，即为添加，有为修改
- descriptor :必需。目标属性所拥有的特性

Object.defineProperty() 第三个参数descriptor 说明:以对象形式{}书写

```
•value: 设置属性的值，默认为undefined
•writable: 值是否可以重写即是否可以被修改 true | false 默认为false 用于唯一标识如ID
•enumerable: 目标属性是否可以被枚举即遍历是否可以拿到 true | false 默认为false
•configurable: 目标属性是否可以被删除或是否可以再次修改特性 true | false 默认为false
```

示例:

```
var data = {
  id: 1,
  name: '小米',
  price: '3999',
}
Object.defineProperty(data, 'num', {
  value: 100
})
Object.defineProperty(data, 'price', {
  value: 2999
})
console.log(data);
Object.defineProperty(data, 'id', {
  // false 为不可重写, true为可重写
  writable: false
})
Object.defineProperty(data, 'id', {
  id: 2999
})
console.log(data);
```

```
▶ {id: 1, name: "小米", price: 2999, num: 100}
  ▶ {id: 1, name: "小米", price: 2999, num: 100}
  >
```

```

Object.defineProperty(data, 'address', {
    value: '山东蓝翔XXXX',
    // enumerable默认为false, 为不允许遍历得到
    enumerable: false,
    // configurable默认为false, 为不允许删除这个属性, 且无法修改特性
    configurable: false
})
console.log(Object.keys(data));
data.address = '11'
delete data.address
console.log(data);

```

```

▶ (3) ["id", "name", "price"]
▶ {id: 1, name: "小米", price: 2999, num: 100, address: "山东蓝翔XXXX"}

```

结果说明：因为之前添加的'num'没有设置enumerable，所以默认为false，不能被遍历，'address'同理。设置了configurable为false后，无法对它进行修改和删除。

7.ES6函数进阶

7.1函数的定义和调用

7.1.1函数的定义方式

1. 函数声明方式function关键字(命名函数)

2. 函数表达式(匿名函数)

3. new Function()

```

var fn = new Function('参数1', '参数2....', '函数体')
var f = new Function('a','b','console.log(a + b)')
f(1,3)//4

```

- Function里面参数都必须是字符串格式

- 第三种方式执行效率低，也不方便书写，因此较少使用

- 所有函数都是Function的实例对象

- 函数也属于对象

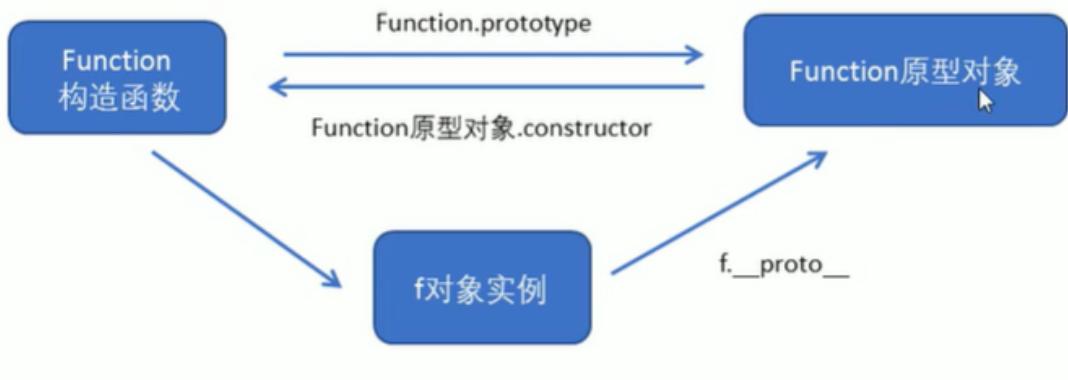
```

▶ prototype: {construct
▶   proto_: f ()      函数属于对象，因此也有__proto__属性，指向Function原型对象
  [[FunctionLocation]]:
  caller: (...)

▶ constructor: f Function()      Function原型对象的构造函数指向Function构造
  length: 0
  name: ""
  [[FunctionLocation]]: function Function() {
}

函数

```



7.1.2 函数的调用方式

1. 普通函数

```

function fn() {
    console.log('保持谨慎');
}
fn()
fn.call()

```

2. 对象的方法

```

var o = {
    sing: function () {
        console.log('保持谨慎');
    }
}
o.sing()

```

3. 构造函数

```

function Star() { }
new Star

```

4. 绑定事件函数

```

btn.onclick = function () { } //点击时调用
btn.click() // 或者自己点击调用

```

5. 定时器函数

```

setInterval(function () { }, 1000); //每隔1s自己调用一次

```

6. 立即执行函数

```

(function () {
    console.log('保持谨慎');
})(); //自动调用

```

7.2 this

7.2.1 函数内this的指向

这些this的指向,是当我们调用函数的时候确定的。调用方式的不同决定了this的指向不同
一般指向我们的调用者

调用方式	this指向
普通函数调用	window
构造函数调用	实例对象, 原型对象里面的方法也指向实例对象
对象方法调用	该方法所属对象
事件绑定方法	绑定事件对象
定时器函数	window
立即执行函数	window

```
// 1.普通函数 this指向window
function fn(){
    console.log('普通函数的this' + this);
}
window.fn();
```

```
// 2.对象的方法 this指向的是对象o
var o={
    sayHi: function() {
        console.log('对象方法的this:' + this);
    }
}
o.sayHi();
```

1.2的结果
普通函数的this[object Window]
对象方法的this:[object Object]

```
// 3.构造函数 this指向lhd这个实例对象,原型对象里面的this也是指向lhd这个实例对象
function Star() {};
Star.prototype.sing = function() {
}
var lhd = new Star();
```

```
// 4.绑定事件函数 this指向的是函数的调用者btn这个按钮对象
var btn = document.querySelector('button');
btn.onclick = function() {
    console.log('绑定时间函数的this:' + this);
};
```

```
// 5.定时器函数this 指向的也是window
window.setTimeout(function() {
    console.log('定时器的this:' + this);
}, 1000);
```

```
// 6.立即执行函数 this还是指向window
(function() {
    console.log('立即执行函数的this' + this);
})();
```

5.6的结果
立即执行函数的this[object Window]
定时器的this:[object Window]

7.2.2 改变函数内部this指向

JavaScript为我们专门提供了一些函数方法来帮我们更优雅的处理函数内部 this 的指向问题,常用的有 `bind()`、`call()`、`apply()` 三种方法。

1.call方法

`call()`方法调用一个对象。简单理解为调用函数的方式,但是它可以改变函数的this指向。

```
fun.call (thisArg, arg1, arg2, ...)
// call第一个可以调用函数,第二个可以改变函数内的this指向
// call的主要作用可以实现继承
```

2.apply方法

`apply()`方法调用一个函数。简单理解为调用函数的方式,但是它可以改变函数的this指向。

```
fun.apply (thisArg, [argsArray])
```

- `thisArg` : 在 `fun` 函数运行时指定的 `this` 值
- `argsArray` : 传递的值,必须包含在数组(包括伪数组)里面, 参数类型取决于数组内的值的类型, 不一定为数组
- 返回值就是函数的返回值, 因为它就是调用函数

应用: 如利用 `apply` 借助于数学内置对象求最大值, 数组本身没有自带求最值方法

示例:

```
var o = {
    name: 'auue'
}
function fn(arr) {
    console.log(this);
    console.log(arr); // 注意输出的不是数组, 而是字符串'pink', 因为传递的是数组里的
                     // 字符串
}
fn.apply(o, ['pink'])
// 作用:1. 调用函数 2. 可以改变函数内部的this指向
// 注意: 参数必须是数组(伪数组)
// 应用: 如利用apply借助于数学内置对象求最大值, 数组本身没有自带求最值方法
var arr = [1, 3, 21, 42]
var max = Math.max.apply(Math, arr) // 此处的this指向最好写成调用者Math
console.log(max);
```

```
top
▶ {name: "auue"}
pink
42
>
```

3.bind方法

bind()方法不会调用函数，但是能改变函数内部this指向，常用

```
fun.bind(thisArg, arg1, arg2, ...)
```

- thisArg : 在 fun 函数运行时指定的 this 值
- arg1 , arg2 : 传递的其他参数
- 返回由指定的 this 值和初始化参数改造的原函数拷贝

bind 意为绑定、捆绑

示例1：

```
var o = {
    name: 'auue'
}
function fn(a, b) {
    console.log(this);
    console.log(a + b);
}
var f = fn.bind(o, 1, 2)
f()
// 1. 不会调用原来的函数，可以改变原来函数内部的this指向
// 2. 返回的是原函数改变this之后产生的新函数
```

```
▼ Object ⓘ
  name: "auue"
  ▶ __proto__: Object
  3
```

示例2：应用：有一个按钮，点击之后禁用，3s后再开启这个按钮。

下行代码块只要使用对象改变，下面this相应会作出改变，更方便，更易维护。

注意：bind要放在定时器函数外面，让定时器隔3s后自己调用

```
// 3. 如果有的函数不需要立即调用，但又想改变函数内部的this指向，此时用bind合适不过了
// bind应用：有一个按钮，点击之后禁用，3s后再开启这个按钮
var btn = document.querySelector('button')
btn.onclick = function () {
    this.disabled = true // this指向btn
    setTimeout(function () {
        // this.disabled=false // 定时器里的this指向window，因此该行代码不能实现
        // 我们的需求
        this.disabled = false // 使用bind将this指向调用者btn
    }.bind(this), 3000)// this指向btn这个对象，注意bind放在定时器函数外面，还是让
    // 定时器隔3s后自己调用
}
```

点击

call apply bind总结

相同点:

都可以改变函数内部的this指向

区别点:

1. call 和apply会调用函数，并且改变函数内部this指向。
2. call 和apply传递的参数形式不一样，call传递参数arv1, arv2形式，apply 必须数组形式[arg]
3. bind 不会调用函数，可以改变函数内部this指向，

主要应用场景:

1. call 经常做继承
2. apply 经常跟数组有关系，比如借助于数学对象实现数组最大值最小值
3. bind 不调用函数但是还想改变this指向，比如改变定时器内部的this指向。

7.3严格模式

7.3.1什么是严格模式

JavaScript除了提供正常模式外，还提供了**严格模式(strict mode)**。ES5 的严格模式是采用具有限制性 JavaScript 变体的一种方式，即在严格的条件下运行JS代码。

严格模式在IE10以上版本的浏览器中才会被支持，旧版本浏览器中会被忽略。

严格模式对正常的JavaScript语义做了一些更改：

1. 消除了Javascript语法的一些不合理、不严谨之处（如变量必须先声明再使用），减少了一些怪异行为。
2. 消除代码运行的一些不安全之处，保证代码运行的安全。
3. 提高编译器效率，增加运行速度。
4. 禁用了在ECMAScript的未来版本中可能会定义的一些语法，为未来新版本的Javascript做好铺垫。比如一些保留字如：class, enum, export, extends, import, super 不能做变量名

7.3.2开启严格模式

严格模式可以应用到整个脚本或个别函数中。因此在使用时，我们可以将严格模式分为**为脚本开启严格模式**和

为函数开启严格模式两种情况。

1.为脚本开启严格模式

为整个脚本文件开启严格模式，需要在所有语句之前放一个特定语句 "usestrict;"
(或 'usestrict';)。

```
<script>
  "use strict";
  console.log("这是严格模式。");
</ script>
```

有的script基本是严格模式,有的script脚本是正常模式,这样不利于文件合并,所以可以将整个脚本文件放在一个立即执行的匿名函数之中。这样独立创建一个作用域而不影响其他 script脚本文件。

```
<script>
  (function () {
    "use strict";
    var num = 10;
    function fn() {}
  })();
</script>
```

因为"use strict"加了引号,所以老版本的浏览器会把它当作一行普通字符串而忽略。

2.为函数开启严格模式

要给某个函数开启严格模式,需要把 "use strict" ; (或 'use strict'); 声明放在函数体所有语句之前。

```
<! -- 为某个函数开启严格模式 -->
<script>
  //此时只是给fn函数开启严格模式
  function fn() {
    'use strict';
    //下面的代码按照严格模式执行
  }
  function fun() {
    //里面的还是按照普通模式执行
  }
</ script>
```

7.3.3严格模式中的变化

严格模式对Javascript的语法和行为,都做了一些改变。

1.变量规定

①在正常模式中,如果一个变量没有声明就赋值,默认是全局变量。严格模式禁止这种用法,变量都必须先用

var命令声明,再使用。

②严禁删除已经声明变量。例如, delete x;语法是错误的。

2.严格模式下this指向问题

①以前在全局作用域函数中的this指向window对象。

②严格模式下全局作用域中函数中的this是undefined

```
// 3.严格模式下全局作用域中函数中的this是undefined
function fn() {
  console.log(this); // undefined
}
fn();
```

③以前构造函数时不加new也可以调用,当普通函数, this 指向全局对象

④严格模式下，如果构造函数不加new调用，this会报错。

⑤new 实例化的构造函数指向创建的对象实例。

⑥定时器this还是指向window。

⑦事件、对象还是指向调用者。

5.6.7不变，2.4变化

3. 函数变化

①函数不能有重名的参数。

```
function fn(a,a) {  
    console.log(a + a);  
};  
fn(1, 2); //非严格模式下，首先a=1；然后a=2；覆盖之前的，得到结果2+2=4  
//严格模式下不能有重名的参数，fn(a,a)
```

②函数必须声明在顶层新版本的JavaScript会引入“块级作用域”(ES6中已引入)。为了与新版本接轨，不允许在非函数的代码块内声明函数 (如在循环体里不可以声明函数)，只能函数里嵌套函数。

```
"use strict";  
if (true){  
    function f() {} // !!!语法错误  
    f();  
}  
for(var i=0;i<5 ;i++){  
    function f2() {} // !!! 语法错误  
    f2();  
}  
function baz() {} //合法  
    function eit() {}  
} //同样合法
```

更多严格模式要求参考: https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Strict_mode

7.4 高阶函数

高阶函数是对其他函数进行操作的函数,它接收函数作为参数或将函数作为返回值输出。

此时fn就是一个高阶函数，函数也是一种数据类型，同样可以作为参数，传递给另外一个参数使用。最典型的就是作为回调函数

接收函数作为参数示例：回调函数最后被调用

```
//回调函数  
<script>  
function fn (callback) {  
    callback&&callback();  
}  
fn(function(){alert('hi')})  
</script>
```

```
//盒子移动完变色
$("div").animate({
    left: 500
}, function() {
    $("div").css("backgroundColor", "purple");
})
```

函数作为返回值输出示例：

```
<script>
function fn() {
    return function() {}
}
fn();
</ script>
```

7.5闭包

7.5.1变量作用域

变量根据作用域的不同分为两种：全局变量和局部变量。

1. 函数内部可以使用全局变量。
2. 函数外部不可以使用局部变量。
3. 当函数执行完毕，本作用域内的局部变量会销毁。

7.5.2什么是闭包

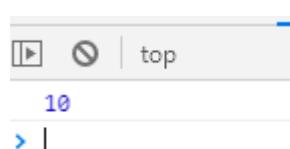
闭包（closure）指有权访问另一个函数作用域中变量的函数。---- JavaScript 高级程序设计

抽取主干后发现：闭包是个函数

简单理解就是一个作用域可以访问另外一个函数内部的局部变量。

示例：

```
// 闭包：fn这个函数作用域访问了另外一个函数fn里面的局部变量num，那么这个局部变量所在函数fn即为
闭包，也可以看作是一种现象
function fn() {
    // fn作用域
    var num = 10
    function fun() {
        // fun作用域
        console.log(num);
    }
    fun()
}
fn()
```



通过Chrome打断点发现：scope 为作用域，Global 意思是全局作用域

```

10 <script>
11 <body>
12   <script>
13     // 闭包: fn这个函数作用域访问了另外一个函数作用域中变量的函数
14     // fn: fn这个函数作用域访问了另外一个函数fn里面的局部变量num,那么这个局部变量所在函数fn即为闭包,也可以立即为一种现象
15     function fn() {
16       // fn作用域
17       var num = 10
18       function fun() {
19         // fun作用域
20         console.log(num);
21       }
22     }
23   </script>
24   fn()
25 </script>
26 </body>

```

Local 为局部作用域, 运行到fun函数内, 出现了 closure 闭包, 里面是闭包函数fn被访问的变量num

```

14 // 闭包: fn这个函数作用域访问了另外一个函数fn里面的局部变量num,那么这个局部变量所在函数fn即为闭包,也可以立即为一种现象
15 function fn() {
16   // fn作用域
17   var num = 10
18   function fun() {
19     // fun作用域
20     console.log(num);
21   }
22 }
23 fn()
24 </script>
25 ...

```

7.5.3 闭包的作用

闭包的主要作用:延伸了变量的作用范围

示例: fn返回fun函数, fn变为高阶函数

```

// fn 外面的作用域也可以访问fn内部的局部变量
function fn() {
  // fn作用域
  var num = 10
  function fun() {
    // fun作用域
    console.log(num);
  }
  return fun
}
var f = fn()
/* 相当于 f=function fun() {
  // fun作用域
  console.log(num);
} */
f() // f调用时, 就会调用fun, 就会访问里面的num(fn的局部变量)

```

打断点可以发现存在闭包

```

12 <script>
13   // fn 外面的作用域也可以访问fn内部的局部变量
14   function fn() {
15     // fn作用域
16     var num = 10
17     function fun() {
18       // fun作用域
19       console.log(num);
20     }
21     return fun
22   }
23   var f = fn()
24   /* 相当于 f=function fun() {
25     // fun作用域
26     console.log(num);
27   }*/
28   f() // f调用时, 就会调用fun, 就会访问里面的num(fn的局部变量)
29 </script>

```

可以将返回fun函数部分代码改进, 直接返回一个匿名函数

```

/* function fun() {
    // fun作用域
    console.log(num);
} */
return function () {
    console.log(num);
}

```

```

24:         /*
25:          return function () {
26:              console.log(num);
27:          }
28:      /* 相当于 f=function fun() {
29:          // fun作用域
30:          console.log(num);
31:      }*/
32:      f() // f调用时，就会调用fun，就会访问里面的num(fun的局部变量)
33:  </script>
34: </body>

```

7.5.4 闭包的应用

1. 循环点击注册事件

点击li输出当前li的索引号

1.以前

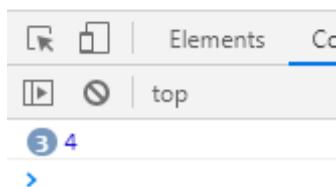
错误方法：

```

// 1.以前
var lis = document.querySelector('.nav').querySelectorAll('li')
for (var i = 0; i < lis.length; i++) {
    lis[i].onclick = function () {
        console.log(i); // function是异步任务，点击了才会执行，而循环是同步任务，会立马执行循环，循环全部执行完毕后li为4，因此，无论点击哪个li，输出都为4
    }
}

```

- 苹果
- 香蕉
- 草莓
- 梨



原因：function是异步任务，点击了才会执行；而循环是同步任务，会立马执行循环，循环全部执行完毕后li为4，因此，无论点击哪个li，输出都为4。

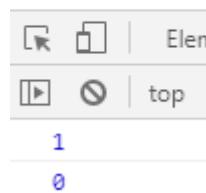
正确的做法：利用动态添加属性的方法

```

var lis = document.querySelector('.nav').querySelectorAll('li')
for (var i = 0; i < lis.length; i++) {
    lis[i].index = i
    lis[i].onclick = function () {
        // console.log(i); // function是异步任务，点击了才会执行，而循环是同步任务，会立马执行循环，循环全部执行完毕后i为4，因此，无论点击哪个li，输出都为4
        console.log(this.index);
    }
}

```

- 苹果
- 香蕉
- 草莓
- 梨



2.利用闭包的方式

立即执行函数也称为小闭包，因为立即执行函数里面的任何一个函数都可以使用i这个变量

```

// 2.利用闭包的方式
for (var i = 0; i < lis.length; i++) {
    // 利用for循环创建4个立即执行函数
    // 立即执行函数也称为小闭包，因为立即执行函数里面的任何一个函数都可以使用i这个变量
    (function (a) { //这个i是传入的形参
        lis[a].onclick = function () {
            console.log(a); //匿名函数里的i用到了立即执行函数里的i，产生了闭包
        }
    })(i) //这个i是调用传给函数的参数，是实参
}

```

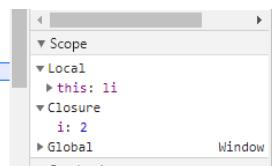


```

// 2.利用闭包的方式
for (var i = 0; i < lis.length; i++) {
    // 利用for循环创建4个立即执行函数
    (function (i) { //这个i是传入的形参
        lis[i].onclick = function () {
            console.log(i); //匿名函数里的i用到了立即执行函数里
        }
    })(i) //这个i是调用传给函数的参数，是实参
}

```

说明：立即执行函数的第二个()是调用，里面放入实参i；将它传入到第一个()，作为形参。点击的匿名函数内使用了这个形参，便产生了闭包。



```
// 利用for循环创建4个立即执行函数
(function (i) { //这个i是传入的形参
    lis[i].onclick = function () {
        console.log(i); //匿名函数里的i用到了立即执行函数里的i，产生了闭包
    }
})(i) //这个i是调用体给函数的参数，是实参
}

```

注意：该案例的两种方法中，显然第一种以前的方法更简单，且闭包方法中的变量i需要点击之后才会被销毁，不点击就不会销毁，占内存，说明闭包并不都是最好的方法，要根据情况选择哪个方法。

2. 循环中的setTimeout()

3s后打印所有li元素内容，同样利用小闭包立即执行函数实现。

```
// 闭包应用-3s后打印所有li元素内容
var lis = document.querySelector('.nav').querySelectorAll('li')
for (var i = 0; i < lis.length; i++) {
    (function (i) {
        setTimeout(function () {
            console.log(lis[i].innerHTML);
        }, 3000)
    })(i)
    // 错误做法
    /* setTimeout(function () {
        console.log(lis[i].innerHTML);
    }, 3000) */
    // 结果报错，原因：定时器里的匿名函数也是异步任务，因此最后的i为4，不存在lis[4]
}
```



3. 计算打车价格

打车起步价13(3公里内)，之后每多一公里增加5块钱。用户输入公里数就可以计算打车价格
如果有拥堵情况，总价格多收取10块钱拥堵费

注意：

1. 函数确实要调用，可以直接写成立即执行函数
2. 函数返回两个值，使用对象返回。有返回值就需要有接收值。
3. 想要使用某个函数，使用对象调用方法即可
4. 判断拥堵返回结果值时，使用三元表达式更方便。

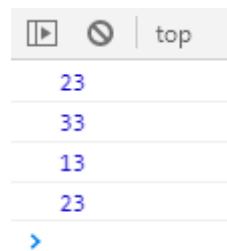
```
var car = (function () {
    var start = 13 //起步价，局部变量
    var total = 0 //总价，局部变量
    return {
        // 正常的总价
        price: function (n) {
```

```

        if (n <= 3) {
            total = start
        } else {
            total = start + (n - 3) * 5
        }
        return total
    },
    // 拥堵之后的费用
    yd: function (flag) {
        // 使用三元表达式更方便
        return flag ? total + 10 : total
    }
}
})()
console.log(car.price(5)); //23
console.log(car.yd(true)); //33

console.log(car.price(2)); //13
console.log(car.yd(true)); //23

```



price和yd里的匿名函数分别使用到了立即执行函数里的start/total变量、total变量，产生了闭包。

A screenshot of a browser's developer tools debugger. On the left, there is a code editor with some JavaScript code. On the right, there is a variable inspector window. The 'Closure' section shows variables: 'n: 5', 'start: 13', and 'total: 0'. A red box highlights the 'Closure' section. Below it, the 'Global' section shows 'this: Object'. At the bottom, there is a 'Breakpoints' section with a checked checkbox for a breakpoint in '08-闭包应用-计算打车费.html... console.log(car.price(5)...)'.

7.5.5 闭包思考题

思考题1

判断下面程序的输出结果：

```

//思考题1
var name = "The Window";
var object = {
    name: "My object",
    getNameFunc: function () {
        return function () {
            return this.name;
        };
    };
}
console.log(object.getNameFunc()())

```

分析：

```

// 相当于
// 1.f=getNameFunc()
/* f = function () {
    return this.name;
}; */
// 2.f()
/* f() = function () {
    return this.name;
}(); */
// 即立即执行函数, this指向window
// window内有属性name, 因此结果为'The Window'

```

全局作用域下属性是放在window内的，没有局部变量，也没有访问局部变量，没有闭包产生

思考题2

```

var name = "The Window";
var object = {
    name: "My Object",
    getNameFunc: function () {
        var that = this;
        return function () {
            return that.name;
        };
    }
};
console.log(object.getNameFunc()())

```

分析：

```

// 相当于
// 1.f=getNameFunc()
/* f =
    var that = this;
    function () {
        return that.name;
}; */
var object = {
    name: "My Object",
    getNameFunc: function () {
        var that = this; //此时的this指向谁，谁调用指向谁，调用者是object
        return function () {
            return that.name;
        };
    }
};
// 2.f()
return that.name; //那么that也指向object，结果为 "My Object"

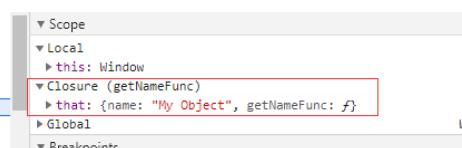
```

此处使用了getNameFunc里that这个局部变量，产生了闭包

```

37 |     var name = "The Window";
38 |     var object = {
39 |         name: "My Object",
40 |         getNameFunc: function () {
41 |             var that = this; //此时的this指向谁，谁调用指向谁，调用者是object
42 |             return function () {
43 |                 return that.name; //return that.name;
44 |             };
45 |         }
46 |     };

```



7.6递归

7.6.1什么是递归？

如果一个函数在内部可以调用其本身，那么这个函数就是递归函数。

简单理解：函数内部自己调用自己，这个函数就是递归函数

递归函数的作用和循环效果一样

由于递归很容易发生“栈溢出”错误（stack overflow），所以必须加退出条件return。

7.6.2利用递归求数学题

1.求 $1*2*3\dots*n$ 阶乘

```
// 求1*2*3...*n 阶乘。
function fn(n) {
    // 退出条件
    if (n == 1) {
        return 1
    }
    return n * fn(n - 1)
}
console.log(fn(3));//6
```

```
// 分析思路，假如用户输入的是3
// fn(3) return 3*fn(2)
// return 3*(2*fn(1))
// return 3*(2*1)
// return 3*(2)
// return 6
```

递归：当前这一项 \times 前一项，从后往前。如果是循环实现，是从前往后。

2.求斐波那契数列

```
// 用户输入一个数字n就可以求出这个数字对应的兔子序列值
// 我们只需要知道用户输入的n的前面两项(n-1 n-2)就可以计算出n对应的序列值
function fb(n) {
    // 退出条件
    if (n == 1 || n == 2) {
        return 1
    }
    return fb(n - 1) + fb(n - 2)
}
console.log(fb(5));//5
```

7.6.3利用递归遍历数组对象

以数组形式存储对象数据，并嵌套对象

```
// 以数组形式存储对象数据
var data = [
    {
        id: 1,
        name: '家电',
```

```

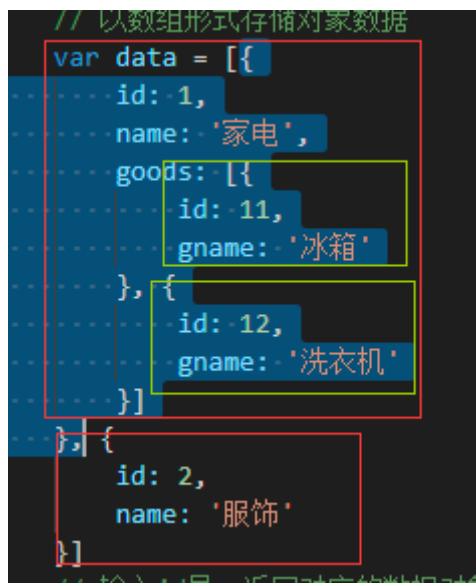
goods: [
    {
        id: 11,
        gname: '冰箱'
    },
    {
        id: 12,
        gname: '洗衣机'
    }
],
{
    id: 2,
    name: '服饰'
}
]
// 输入id号，返回对应的数据对象
// 1.forEach遍历每个对象
function getID(json, id) {
    json.forEach(function (item) {
        // console.log(item); //2个数组元素
        // 先遍历外层数据
        if (item.id == id) {
            console.log(item);
            // 2.利用递归遍历内层数据，goods数组存在且数组长度大于0
        } else if (item.goods && item.goods.length > 0) {
            getID(item.goods, id)
        }
    })
}
getID(data, 1)
getID(data, 12)

```

```

▶ {id: 1, name: "家电", goods: Array(2)}
▶ {id: 12, gname: "洗衣机"}

```



注意**data数据的存储格式**，data数组内有2个对象，第一个对象内嵌套着两个对象。

先遍历外层数据，再利用递归遍历内层数据，goods数组存在且数组长度大于0

上面代码改进，要求获取返回对应的对象再打印，而不是直接打印出来

```

function getID(json, id) {
    var o = {} //存放返回的对象
    json.forEach(function (item) {
        // console.log(item); //2个数组元素

```

```

    // 先遍历外层数据
    if (item.id == id) {
        // console.log(item);
        o = item
        // return o //在forEach内使用return无效
        // 2.利用递归遍历内层数据，goods数组存在且数组长度大于0
    } else if (item.goods && item.goods.length > 0) {
        o = getID(item.goods, id) //需要一个值来接收遍历返回的o
    }
}
return o
}

console.log(getID(data, 1));
console.log(getID(data, 11));

```

注意：

1. 使用o一个空对象来存放返回的值
2. 在forEach内使用return无效
3. 在递归遍历内层数据时，只有return o，但没有接收，所以需要设置一个值来接收遍历返回的o。



4. 对于数据，可以在“冰箱”内再进行嵌套，名字仍需为'goods'，与前一个“冰箱”嵌套的'goods'一样。

```

// 以数组形式存储对象数据
var data = [
    {
        id: 1,
        name: '家电',
        goods: [
            {
                id: 11,
                gname: '冰箱',
                goods: [
                    {
                        id: 111,
                        gname: '美的'
                    },
                    {
                        id: 112,
                        gname: '格力'
                    }
                ]
            }
        ]
    }
]

```

7.6.4 浅拷贝和深拷贝

1. 浅拷贝，只是拷贝一层，更深层次对象级别的只拷贝引用
2. 深拷贝，拷贝多层，每一级别的数据都会拷贝。
3. `Object.assign(target, sources)` es6 新增方法可以浅拷贝

浅拷贝

示例：

```
var obj = {
```

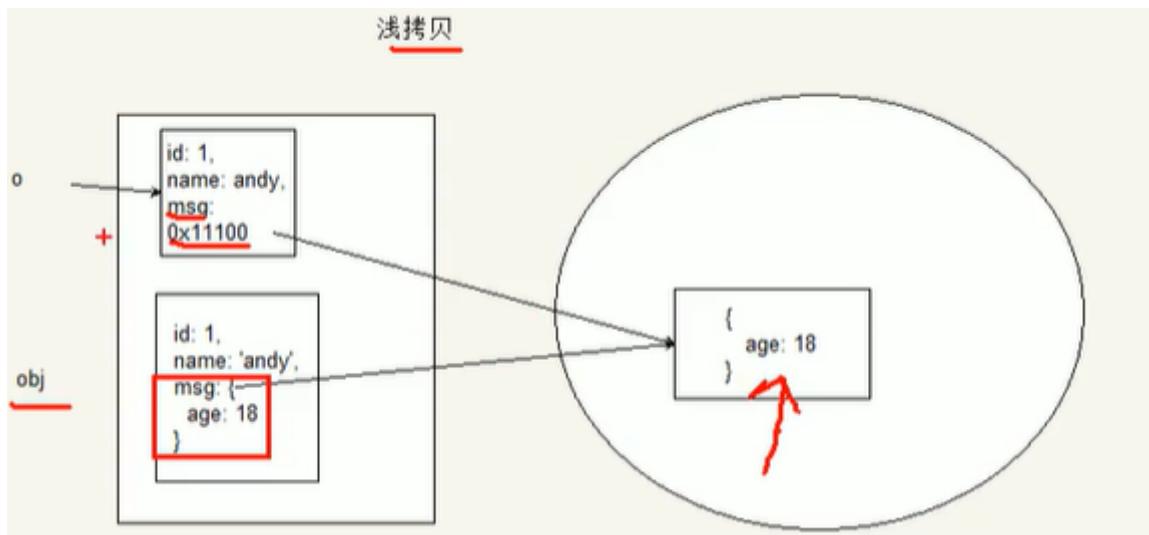
```

        id: 1,
        name: '火腿',
        other: {
            age: 4
        }
    }
o = {}
// 1.浅拷贝
// 1) 遍历赋值实现浅拷贝
for (var k in obj) {
    // k是属性名,obj[k]是属性值
    o[k] = obj[k]
}
console.log(o);
o.other.age = 15
console.log(obj);
console.log('-----');
// 2)Object.assign(拷贝的,被拷贝的)
a = {}
Object.assign(a, obj)
console.log(a);

```



浅拷贝：拷贝的是地址，对其属性值进行修改，原拷贝对象里面的属性值也会被改变。



深拷贝

```

var obj = {
    id: 1,
    name: '火腿',
    other: {
        age: 4
    },
    color: ['pink', 'red']
}

var o = []
// 使用遍历实现深拷贝，深层次的数据遍历拷贝
function deepCopy(newobj, oldobj) {
    for (var k in oldobj) {
        // 判断属性值的数据类型
        var item = oldobj[k]
        // 简单数据类型直接赋值，复杂数据类型（如数组、对象）需要进入里面使用递归方式
    }
}

deepCopy(o, obj)
console.log(o);
o.other.age = 10
console.log(obj);

```

```

▼ {id: 1, name: "火腿", other: {...}, color: Array(2)} ⓘ
  ► color: (2) ["pink", "red"]
  ► id: 1
  ► name: "火腿"
  ► other: {age: 10}  
  ► __proto__: Object

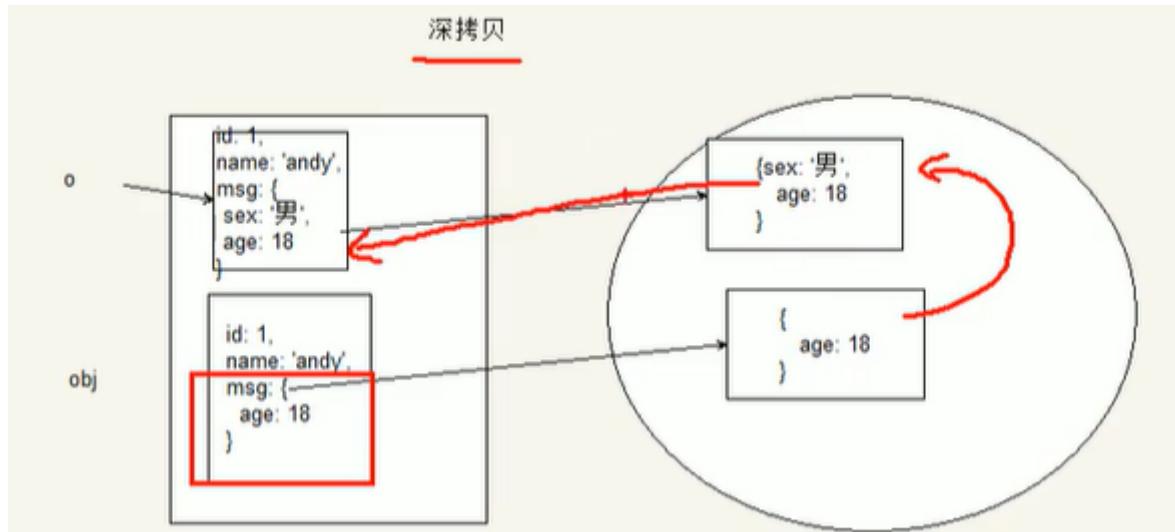
▼ {id: 1, name: "火腿", other: {...}, color: Array(2)} ⓘ
  ► color: (2) ["pink", "red"]
  ► id: 1
  ► name: "火腿"
  ► other: {age: 4}  
  ► __proto__: Object

```

注意：

1.先判断是否是数组，然后判断是否是对象，顺序不能颠倒。原因：数组也属于对象，如果放后面，数组赋值就会按对象来进行赋值，那就不是数组了。

2. 深拷贝将里面的内容重新复制了一份，另开辟了一个空间存放，不会影响被拷贝的数据。



8. 正则表达式

8.1 正则表达式概述

正则表达式(Regular Expression)是用于匹配字符串中字符组合的模式。在JavaScript中,正则表达式也是对象。

正则表通常被用来检索、替换那些符合某个模式(规则)的文本，例如验证表单:用户名表单只能输入英文字母、数字或者下划线，昵称输入框中可以输入中文(匹配)。此外,正则表达式还常用于过滤掉页面内容中的一些敏感词(替换),或从字符串中获取我们想要的特定部分(提取)等。

其他语言也会使用正则表达式,本阶段我们主要是利用JavaScript正则表达式完成表单验证。

8.1.2 正则表达式的特点

1. 灵活性、逻辑性和功能性非常的强。
2. 可以迅速地用极简单的方式达到字符串的复杂控制。
3. 对于刚接触的人来说,比较晦涩难懂。比如: `^\w+([-.\w+]*)*\@\w+([-.\w+]*)\.\w+([-.\w+]*)$`
4. 实际开发,一般都是直接复制写好的正则表达式,但是要求会使用正则表达式并且根据实际情况修改正则表达式比如用户名: `/^[a-zA-Z_-]{3,16}$/`

8.2 正则表达式在JavaScript中的使用

8.2.1 创建正则表达式

在JavaScript中,可以通过两种方式创建一个正则表达式。

1. 通过调用 RegExp 对象的构造函数创建

```
var 变量名 = new RegExp (/表达式/i);
```

2. 通过字面量创建,更常用

```
var 变量名 = /表达式/i;
```

8.2.2 测试正则表达式test

test()正则对象方法,用于检测字符串是否符合该规则,该对象会返回true或false,其参数是测试字符串。

```
regexObj.test (str)
```

- 1.regexObj 是写的正则表达式
- 2.str我们要测试的文本
- 3.就是检测str文本是否符合我们写的正则表达式规范

示例:

```
// 1.通过调用RegExp对象的构造函数创建正则表达式
var regexp = new RegExp(/123/)
console.log(regexp);
// 2.通过字面量创建
var rg = /123/
// 3.test方法用来检测字符串是否符合正则表达式要求的规范
console.log(rg.test(123)); //包含了123才符合要求, 12等都不符合要求
console.log(rg.test('ab'));
```



8.3 正则表达式中的特殊字符

8.3.1 正则表达式的组成

一个正则表达式可以由简单的字符构成,比如/abc/,也可以是简单和特殊字符的组合,比如/ab*c/。其中特殊字符也被称为元字符,在正则表达式中是具有特殊意义的专用符号,如^、\$、+等。

特殊字符非常多,可以参考:

- MDN : https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions
- jQuery手册:正则表达式部分
- 正则测试工具: <http://tool.oschina.net/regex>

8.3.2 边界符

正则表达式中的边界符(位置符)用来提示字符所处的位置,主要有两个字符。

边界符	说明
^	表示匹配行首的文本(以谁开始)
\$	表示匹配行尾的文本(以谁结束)

如果^和\$在一起,表示必须是精确匹配。只能是两个符号内的字符,其他都不符合规范

示例：

```
// 边界符 ^ $  
var rg = /abc/ // 正则表达式里面不需要加引号，不管是数字型还是字符串型  
// /abc/ 只要包含了abc这个字符返回都是true  
console.log(rg.test('abc')); // 测试时字符串需要带引号  
console.log(rg.test('abcd'));  
console.log(rg.test('aacd'));  
console.log('-----');  
var reg = /^abc/ // 以abc字符为开头的返回true  
console.log(reg.test('abc')); // true  
console.log(reg.test('abcd')); // true  
console.log(reg.test('aacd')); // false  
console.log('-----');  
var regk = /^abc$/ // 精确匹配，要求必须是abc字符串才符合规范  
console.log(regk.test('abc')); // true  
console.log(regk.test('abcd')); // false  
console.log(regk.test('aacd')); // false  
console.log(regk.test('abcabc')); // false
```

```
true  
true  
false  
-----  
true  
true  
false  
-----  
true  
false  
false  
false
```

注意：

1.正则表达式里面不需要加引号，不管是数字型还是字符串型

2.var regk = /^abc\$/，精确匹配，要求必须是abc字符串才符合规范，'abcabc'也不符合规范。

8.3.3字符类

字符类表示有一系列字符可供选择，只要匹配其中一个就可以了。所有可供选择的字符都放在方括号内。

-方括号内部范围符-

示例：

```
// var rg=/abc/ // 包含了abc就可以  
// 字符类： [] 表示有一系列字符可供选择，只要匹配其中一个就可以了  
var rg = /[abc]/ // 只要包含有a/b/c都返回true  
console.log(rg.test('auue')); // 包含了a, true  
console.log(rg.test('baby')); // 包含了a和b, true  
console.log(rg.test('color')); // 包含了c, true  
console.log(rg.test('red')); // 没有包含a/b/c, false  
console.log('-----');
```

```
var reg = /^[abc]$/ //三选一，只能是a/b/c字母时才返回true
console.log(reg.test('aa')); // false
console.log(reg.test('a')); // true
console.log(reg.test('b')); // true
console.log(reg.test('c')); // true
console.log(reg.test('abc')); // false
console.log('=====');
var req = /^[a-z]$/ //26选一字母，只能小写返回true， -表示的是a到z的范围
console.log(req.test('f')); // true
console.log(req.test('F')); // false
console.log(req.test(1)); // false
```

字符组合

```
/^[a-z0-9]$/.test('a') // true
```

示例：

```
// 字符组合
var reg1 = /^[a-zA-Z0-9_-]$/ //26选一大小写字母， -/_， 数字0-9任一个返回为true
console.log(reg1.test('s')); //true
console.log(reg1.test('c')); //true
console.log(reg1.test(4)); //true
console.log(reg1.test('-')); //true
console.log(reg1.test('_')); //true
console.log(reg1.test('!')); //false
```

[^]方括号内部取反符^

```
/[^abc]$/. test('a')// false
```

示例：

```
// ^放在中括号内部意为取反，与边界符^不要混淆
var reg2 = /^[^a-zA-Z0-9_-]$/
console.log(reg2.test('s')); //false
console.log(reg2.test('c')); //false
console.log(reg2.test(4)); //false
console.log(reg2.test('-')); //false
console.log(reg2.test('_')); //false
console.log(reg2.test('!')); //true
```

以上只能验证一个字符。

8.3.4量词符

量词符用来设定某个模式出现的次数。

量词	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

注意量词符都写在a的后面,{n,m}之间没有空格

```
var rg = /^a*$/
```

示例：

```
//量词符:用来设定某个模式出现的次数
//简单理解:就是让下面的a这个字符重复多少次
// var rg=/^a$/

// * 相当于>=0, 出现0次及以上
var rg = /^a*$/;
console.log(rg.test('')); //出现0次, true
console.log(rg.test('a')); //true
console.log(rg.test('aa')); //true

// + 相当于>=1, 出现1次及以上
var rg = /^a+$/;
console.log(rg.test('')); //出现0次, false
console.log(rg.test('a')); //true
console.log(rg.test('aa')); //true

// ? 相当于1||0,1次或0次
var rg = /^a?$/;
console.log(rg.test('')); //出现0次, true
console.log(rg.test('a')); //true
console.log(rg.test('aa')); //false

// {n} 重复n次, 只有重复n次才符合规范, 多一次少一次都为false
var rg = /^a{3}$/;
console.log(rg.test('')); //出现0次, false
console.log(rg.test('a')); //false
console.log(rg.test('aa')); //false
console.log(rg.test('aaa')); //true

// {n,} 重复大于等于n次, 重复n次及以上都符合规范
var rg = /^a{2,}$/;
console.log(rg.test('')); //出现0次, false
console.log(rg.test('a')); //false
console.log(rg.test('aa')); //true
console.log(rg.test('aaa')); //true

// {n,m} 重复大于等于n次, 且小于等于m次
```

```

var rg = /^[a{2,6}】/
console.log(rg.test('')); //出现0次, false
console.log(rg.test('a')); //false
console.log(rg.test('aa')); //true
console.log(rg.test('aaa')); //true
console.log(rg.test('aaaaaaaa')); //重复7次, false

```

案例：用户名验证

功能需求：

- 1.如果用户名输入合法,则后面提示信息为:用户名合法, 并且颜色为绿色
- 2.如果用户名输入不合法则后面提示信息为:用户名不符合规范,并且颜色为红色

分析:

- 1.用户名只能为英文字母,数字, 下划线或者短横线组成, 并且用户名长度为6~16位.
- 2.首先准备好这种正则表达式模式 /^[a-zA-Z0-9-_]{6,16}\$/
- 3.当表单失去焦点就开始验证
- 4.如果符合正则规范, 则让后面的span标签添加right类.
- 5.如果不符合正则规范, 则让后面的span标签添加wrong类

```

var reg = /^[a-zA-Z0-9-_]{6,16}$/ //用户只能输入英文字母、数字、下划线、短横线，通过量词
增加验证字符串长度
/* console.log(reg.test('auue_red')); //true
console.log(reg.test('auue-01')); //true
console.log(reg.test('auue001')); //true
console.log(reg.test('auue!00')); //出现不符合规范的!, false */
var ipt = document.querySelector('input')
var span = document.querySelector('span')
ipt.onblur = function () {
    if (reg.test(ipt.value)) {
        span.innerHTML = '用户名输入格式正确'
        span.classList = 'right'
    } else {
        span.innerHTML = '用户名输入格式错误'
        span.classList = 'wrong'
    }
}

```

用户名输入格式错误

用户名输入格式正确

用户名输入格式错误

8.3.5括号总结

- 1.花括号,量词符,里面表示重复次数
- 2.中括号,字符集合,匹配方括号中的任意字符
- 3.小括号 表示优先级

可以在线测试: <https://c.runoob.com/>

```

//中括号 字符集合.匹配方括号中的任意字符。
// var reg = /^[abc]$/ ;
//a也可以b也可以c可以 a||b||c
//大括号 量词符，里面表示重复次数
var reg = /abc{3}$/; //它只是让c重复三次,只能是abccc, 其他都不符合规范
console.log(reg.test('abc')); //false
console.log(reg.test('abcabcabc'));//false
console.log(reg.test('abccc'));//true
//小括号表示优先级
var reg = /^(abc){3}$/; //它是让abc都重复三次,只能是abcabcabc
console.log(reg.test('abc'));
console.log(reg.test('abcabcabc'));
console.log(reg.test('abccc'));

```

在线网站中常见的正则表达式：

三、特殊需求表达式



- Email地址: ^\w+([-.\w+]*)@\w+([-.\w+]*)\.\w+([-.\w+]*)\$
- 域名: [a-zA-Z0-9][a-zA-Z0-9]{0,62}(\.[a-zA-Z0-9]{0,62})+\.?
- InternetURL: [a-zA-z]+\//[^s]* 或 ^http://([w-]+.)+([w-]/%&=]*)?\$
- 手机号码: ^[13[0-9]|14[57]|15[0|1|2|3|4|5|6|7|8|9]|18[0|1|2|3|5|6|7|8|9]]\d{8}\$
- 电话号码("XXX-XXXXXX"、"XXXX-XXXXXX"、"XXX-XXXXXX"、"XXX-XXXXXX"、"XXXXXX"和"XXXXXXX"): ^((\d{3,4}-)|\d{3,4}-)?\d{7,8}\$
- 国内电话号码(0511-4405222、021-87888822): \d{3}-\d{8}|\d{4}-\d{7}
- 电话号码正则表达式 (支持手机号码, 3-4位区号, 7-8位直播号码, 1-4位分机号) : ((\d{11})|^\d{7,8}|\d{4}\d{3}-\d{7,8}|\d{4}\d{3}-\d{7,8}-\d{4}\d{3})|\d{2}(\d{1})|\d{7,8}-\d{4}\d{3})|\d{4}(\d{1})\$
- 身份证号(15位、18位数字), 最后一位是校验位, 可能为数字或字符X: (^{\d{15}}|(^{\d{18}})(?=\d{17}\d{1}X|x))\$
- 帐号是否合法(字母开头, 允许5-16字节, 允许字母数字下划线): ^[a-zA-Z][a-zA-Z0-9_]{4,15}\$
- 密码(以字母开头, 长度在6~18之间, 只能包含字母、数字和下划线): ^[a-zA-Z]\w{5,17}\$
- 强密码(必须包含大小写字母和数字的组合, 不能使用特殊字符, 长度在8-10之间): ^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])[a-zA-Z0-9]{8,10}\$
- 强密码(必须包含大小写字母和数字的组合, 可以使用特殊字符, 长度在8-10之间): ^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}\$
- 日期格式: ^\d{4}-\d{1,2}-\d{1,2}
- 一年的12个月(01~09和1~12): ^[0-9]{1}[0-2]\$
- 一个月的31天(01~09和1~31): ^((0[1-9])|(1[0-2]))|30|31\$

8.3.6预定义类

预定义类指的是某些常见模式的简写方式。

预定类	说明
\d	匹配0-9之间的任一数字, 相当于[0-9]
\D	匹配所有0-9以外的字符, 相当于[^0-9]
\w	匹配任意的字母、数字和下划线, 相当于[A-Za-z0-9_]
\W	除所有字母、数字和下划线以外的字符, 相当于[^A-Za-z0-9_]
\s	匹配空格(包括换行符、制表符、空格符等), 相当于[\t\r\n\f]
\S	匹配非空格的字符, 相当于[^t\r\f]

示例:

座机号码验证:全国座机号码两种格式:010-12345678或者0530-1234567

```

// 正则里面的或者符号 |
var reg = /\d{3}-\d{8}|\d{4}-\d{7}$/
```

注意：正则里面的或者符号 |，左右两侧没有空格

/	<code>^\d{3}-\d{8} \d{4}-\d{7}\$</code>
051-12345678	
共找到 1 处匹配： 051-12345678	

案例：品优购注册页面表单验证

手机号:

QQ:

昵称:

短信验证码:

登录密码:

安全强度 弱 中 强

确认密码:

同意协议并注册 [《知晓用户协议》](#)

分析：

- 手机号码：/`^1[3|4|5|7|8][0-9]{9}$`/
- QQ：/[1-9][0,9]{4,}/ (腾讯QQ号从10000开始)
- 昵称是中文：/`[\u4e00-\u9fa5]{2,8}`\$

1.手机号验证

注意：

1.通过表单的兄弟节点找到后面的提示语文本框span。由于默认没有提示语，输入后鼠标离开后再判断给出提示语，需要将内容删除，但不能全部删除span盒子，若span盒子全部删除，则无法找到兄弟节点span，因此必须在表单后面保留一个span盒子，后面验证也是如此

```
<li>
    <label for="">手机号: </label>
    <input type="text" name="phonenum" id="tel">
    <span class="">
        <!-- <i class="error_icon"></i>
        手机号码格式不正确，请重新输入--> </span>
</li>
```

2.手机号码的正则表达式 `/^1[3|4|5|7|8]\d{9}$/`

```
window.onload = function () {
    // 手机号码的正则表达式，前两位只有13/14/15/17/18开头的，共11位
    var regtel = /^1[3|4|5|7|8]\d{9}$/,
        tel = document.querySelector('#tel')
    tel.addEventListener('blur', function () {
        if (regtel.test(this.value)) {
            // console.log('right');
            // console.log(this.nextElementSibling);
            // 正确的，修改图标、文字内容和样式类
            this.nextElementSibling.className = 'success'
            this.nextElementSibling.innerHTML = '<i class="success_icon"></i>手机号码格式输入正确'
        } else {
            // console.log('error');
            // 错误的，修改图标、文字内容和样式类
            this.nextElementSibling.className = 'error'
            this.nextElementSibling.innerHTML = '<i class="error_icon"></i>手机号码格式不正确，请重新输入'
        }
    })
}
```

默认

手机号:

输入正确

手机号:  手机号码格式输入正确

输入错误

手机号:  手机号码格式不正确，请重新输入

2.QQ/昵称/短信验证码/登录密码验证

注意：验证都是同理，可以将手机验证的那段代码进行函数封装。

```

// QQ号码从10000开始
var regqq = /^[1-9]\d{4,}$/,
// 昵称是中文, 2-8个汉字
var regnc = /^[\u4e00-\u9fa5]{2,8}$/,
// 短信验证码是6位数字
var regmsg = /^\d{6}$/,
// 密码
var regpwd = /^[a-zA-Z0-9-_]{6,16}$/,
var tel = document.querySelector('#tel'),
var qq = document.querySelector('#qq'),
var nc = document.querySelector('#nc'),
var msg = document.querySelector('#msg'),
var pwd = document.querySelector('#pwd'),
regexp(tel, regtel) //手机号码
regexp(qq, regqq) //qq号码
regexp(nc, regnc) //昵称
regexp(msg, regmsg) //短信验证
regexp(pwd, regpwd) //密码
// 表单验证函数
function regexp(ele, reg) {
    ele.addEventListener('blur', function () {
        if (reg.test(this.value)) {
            // console.log('right');
            // console.log(this.nextElementSibling);
            // 正确的, 修改图标、文字内容和样式类
            this.nextElementSibling.className = 'success'
            this.nextElementSibling.innerHTML = '<i class="success_icon">' +
                '</i>恭喜您, 输入正确'
        } else {
            // console.log('error');
            // 错误的, 修改图标、文字内容和样式类
            this.nextElementSibling.className = 'error'
            this.nextElementSibling.innerHTML = '<i class="error_icon"></i>输入不正确, 请重新输入'
        }
    })
}

```

QQ:	<input type="text" value="10000000"/>	 恭喜您, 输入正确
昵称:	<input type="text" value="打发打发"/>	 恭喜您, 输入正确
短信验证码:	<input type="text" value="123456"/>	 恭喜您, 输入正确
QQ:	<input type="text" value="00000000"/>	 输入不正确, 请重新输入
昵称:	<input type="text" value="打"/>	 输入不正确, 请重新输入
短信验证码:	<input type="text" value="1234563"/>	 输入不正确, 请重新输入

登录密码: 12345

 输入不正确, 请重新输入

3.二次确认密码验证

注意：“登录密码”和“确认密码”表单失去焦点时，都要判断“确认密码”是否与“登录密码”表单内容一样(即后期修改登录密码时也要再次判断两者密码是否一样)，并且修改的都是“确认密码”的提示语和样式。

```
surepwd.addEventListener('blur', function () {
    if (this.value == pwd.value) {
        this.nextElementSibling.className = 'success'
        this.nextElementSibling.innerHTML = '<i class="success_icon"></i>恭喜您, 输入正确'
    } else {
        this.nextElementSibling.className = 'error'
        this.nextElementSibling.innerHTML = '<i class="error_icon"></i>密码输入不一致'
    }
})
pwd.addEventListener('blur', function () {
    if (this.value == surepwd.value) {
        surepwd.nextElementSibling.className = 'success'
        surepwd.nextElementSibling.innerHTML = '<i class="success_icon"></i>恭喜您, 输入正确'
    } else {
        surepwd.nextElementSibling.className = 'error'
        surepwd.nextElementSibling.innerHTML = '<i class="error_icon"></i>密码输入不一致'
    }
})
```

登录密码: 12345678

 恭喜您, 输入正确

安全强度  弱  中  强

确认密码: 1234567

 密码输入不一致

8.4正则表达式中的替换

8.4.1 replace()替换

replace()方法可以实现替换字符串操作,用来替换的参数可以是一个字符串或是一个正则表达式。

```
stringObject.replace(regexp/substr,replacement)
```

1. 第一个参数:被替换的字符串或者正则表达式
2. 第二个参数: 替换为的字符串
3. 返回值是一个替换完毕的新字符串

8.4.2 正则表达式参数

/表达式/[switch]

switch(也称为修饰符)按照什么样的模式来匹配,有三种值:

- g:全局匹配
- i:忽略大小写
- gi:全局匹配+忽略大小写

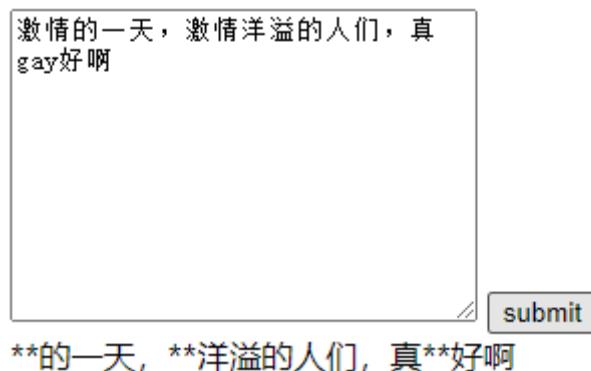
示例:

```
var str = 'auue和andy'  
// var newStr = str.replace('auue', 'pupe')  
var newStr = str.replace(/auue/, 'pupe')  
console.log(newStr);
```



应用：将敏感词替换为**

```
var text = document.querySelector('textarea')  
var btn = document.querySelector('button')  
var div = document.querySelector('div')  
btn.onclick = function () {  
    div.innerHTML = text.value.replace(/激情|gay/gi, '**')  
}
```



8.4.3 search()检索

search()方法可以实现检索字符串操作,用来检索的参数可以是一个字符串或是一个正则表达式。

stringObject.search(regexp/subtr)

1. 第一个参数:被替换的字符串或者正则表达式
2. 返回值是子串的起始位置(索引号)

```
var str = "Visit Runoob!";
var n = str.search("Runoob");
n //6
```

JavaScript ES6新特性

使用教程: <https://es6.ruanyifeng.com/>

9.let和const命令

9.1 let变量声明

用法类似于 var，但是所声明的变量，只在 let 命令所在的代码块内有效。

特性：

1. 变量不能重复声明,会报错
2. 块级作用域，仅在代码块内有效，出代码块无效。if else while for等循环内也是块级作用域
3. 不存在变量提升，必须先声明后使用
4. 不影响作用域链
5. 暂时性死区 (temporal dead zone, 简称 TDZ)，若块级作用域内存在let命令，那么在它声明变量之前，该变量不受外部影响，因此若在声明前使用该变量会报错。设计目的：养成先声明变量后使用的好习惯。

```
// 声明变量
let a;
let b;
// 1. 变量不能重复声明,会报错
/* let c = 0;
var c = 'abu' */

// 2. 块级作用域，仅在代码块内有效，出代码块无效。
// if else while for等循环内也是块级作用域
// 作用域分类还有全局作用域、函数作用域、eval作用域（es5里）
{
  let star = '刘德华'
  var st = '张学友'
}
// console.log(star); //报错, star is not defined
console.log(st); //张学友

// 3. 不存在变量提升
console.log(c); //undefined
console.log(d); //报错, Cannot access 'd' before initialization
var c = 6;
let d = 9;

// 4. 不影响作用域链
{
  let e = 'auue'
  function fn() {
    console.log(e);
```

```

        }
        fn() //auue
    }

// 5.暂时性死区
// 块级作用域内存在let命令，那么在它声明变量之前，该变量不受外部影响。因此若在声明前使用该变量会报错。
var f = '死区';
if (true) {
    f = '暂时' //Cannot access 'f' before initialization
    let f;
}

```

案例：点击div盒子变色

```

let divs = document.querySelectorAll('div')
// var 声明变量
/* for (var i = 0; i < divs.length; i++) {
    divs[i].onclick = function () {
        this.style.backgroundColor = 'pink'
        // divs[i].style.backgroundColor='pink' //不能这么写，循环完成后i的
值变为3
    }
} */
// console.log(window.i); //3
// let 声明变量
for (let i = 0; i < divs.length; i++) {
    divs[i].onclick = function () {
        // this.style.backgroundColor = 'pink'
        divs[i].style.backgroundColor = 'pink' //此时可行，因为当前let只在本
次循环内有效
    }
}

```

说明：

1. 使用 `var` 声明变量 `i`，在全局范围内都有效（函数级，多人只有一辆车），所以全局只有一个变量 `i`。每一次循环，变量 `i` 的值都会发生改变。循环内部的函数内部的 `divs[i].style.backgroundColor='pink'`，里面的 `i` 指向的就是全局的 `i`，所有数组 `div` 的成员里面的 `i`，指向的都是同一个 `i`，导致运行时输出的是最后一轮的 `i` 的值，也就是 3。循环结束后，用来计数的循环变量并没有消失，泄露成了全局变量。

2. 使用 `let` 声明变量 `i`，当前的 `i` 只在本轮循环有效（块级作用域，每一个人一辆车），所以每一次循环的 `i` 其实都是一个新的变量。JavaScript 引擎内部会记住上一轮循环的值，初始化本轮的变量 `i` 时，是在上一轮循环的基础上进行计算的。

3. `for` 循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域

```

for (let i = 0; i < 3; i++) {
    let i = 'abc';
    console.log(i);
}

```



连续输出三个abc，说明循环变量i和函数内部变量i不在一个作用域内

9.2ES6块级作用域

1. `let` 实际上为 JavaScript 新增了**块级作用域**。

```
function f1() {  
    let n = 5;  
    if (true) {  
        let n = 10;  
    }  
    console.log(n); // 5  
}
```

运行后输出 5。这表示**外层代码块不受内层代码块的影响**。如果两次都使用 `var` 定义变量 `n`，最后输出的值为 10（就近原则）

2. 块级作用域的出现，实际上使得获得广泛应用的匿名立即执行函数表达式（匿名 IIFE）不再必要了。

```
// IIFE 写法  
(function () {  
    var tmp = ...;  
    ...  
}());  
  
// 块级作用域写法  
{  
    let tmp = ...;  
    ...  
}
```

3. 避免在块级作用域内声明函数。如果确实需要，需要写成函数表达式，而不是函数声明语句。

4. ES6 的**块级作用域必须有大括号（for while 等也可以算作块级作用域）**，如果没有大括号，JavaScript 引擎就认为不存在块级作用域

9.3const常量声明

语法格式

```
// 声明常量  
const PI = 3.142592
```

特点：

1. 常量一般使用大写（语法规范）

2. 常量的值不能修改

```
const PI = 3 // Identifier 'PI' has already been declared
```

3. `const` 一旦声明就要赋初始值，只声明不赋值会报错

```
const A //Missing initializer in const declaration
```

4. 块级作用域

5.常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用

6.不可重复声明

```
// 4.块级作用域
/*
    const MAX = 7
}
console.log(MAX); // MAX is not defined */
//5.常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用
/*
    console.log(MAX);// ReferenceError
    const MAX = 7
} */
// 6.不可重复声明
/* var msg = 'hello'
```

7.常量的值不能修改，其本质是变量指向的内存地址不能修改，因此可以修改复合类型的数据（数组和对象）变量，它们变量本身就指向内存地址。

```
const MAX = []
MAX.push(7)
console.log(MAX);
const MIN = {}
MIN.prop = 12
console.log(MIN);
```

9.4顶层对象的属性

顶层对象，在浏览器环境指的是 `window` 对象，在 Node 指的是 `global` 对象。ES5 之中，顶层对象的属性与全局变量是等价的(顶层对象的属性赋值与全局变量的赋值，是同一件事)。

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

示例：

```
var a = 1;
// 如果在 Node 的 REPL 环境，可以写成 global.a
// 或者采用通用方法，写成 this.a
window.a // 1

let b = 1;
window.b // undefined
```

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

9.5globalThis对象

顶层对象在各种实现里面是不统一的。

- 浏览器里面，顶层对象是 `window`，但 Node 和 Web Worker 没有 `window`。
- 浏览器和 Web Worker 里面，`self` 也指向顶层对象，但是 Node 没有 `self`。

- Node 里面，顶层对象是 `global`，但其他环境都不支持
- ES2020在语言标准的层面，引入 `globalThis` 作为顶层对象。也就是说，任何环境下，`globalThis` 都是存在的，都可以从它拿到顶层对象，指向全局环境下的 `this`

10. 变量的解构赋值

10.1 数组的解构赋值

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**（Destructuring）。

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

注意：

1. 等号左边的变量多于值：解构不成功，变量的值为`undefined`，

2. 等号左边的变量少于值：不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组，缺失右侧部分值

3. 赋值与解构必须同时完成（对象的解构赋值同），不能分开。

```
// 以前
/* let a = 1;
let b = 2;
let c = 3; */

// 现在，可以从数组中提取值，按照对应位置，对变量赋值。本质上这种写法属于“模式匹配”，
// 只要等号两边的模式相同，左边的变量就会被赋予对应的值
let [a, b, c] = [1, 2, 3]
console.log(a, b, c);
// 1) 使用嵌套数组进行解构
let [d, [f, g], h] = [3, [7, 2], 9]
console.log(d, f, g, h);
let [, , third] = ['a', 'b', 'c']
console.log(third); // c
let [m, , n] = ['one', 'two', 'three']
console.log(m, n); // one three
let [head, ...tail] = [1, 2, 3, 4]
console.log(head, tail); // 1 [2, 3, 4]
let [x, y, ...z] = ['a']
console.log(x, y, z); // 'a' undefined []
// 2) 如果解构不成功，变量的值为undefined
let [t, p] = ['b']
console.log(t, p); // 'b' undefined
// 3) 不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组
let [u, w] = [1, 2, 3]
console.log(u, w); // 1 2
let [r, [s], i] = [1, [2, 3], 4]
console.log(r, s, i); // 1 [2, 3] 4
```

默认值

解构赋值允许指定默认值，只有当一个数组成员严格等于（`==`）`undefined`，默认值才会生效

示例：

```
let [x = 1] = [undefined];
```

x // 1 默认值生效

```
let [x = 1] = [null];
```

x // null 默认值不生效

10.2 对象的解构赋值

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而**对象的属性没有次序，变量必须与属性同名，才能取到正确的值。**

注意：

1. 若对象没有对应的同名属性，则解构失败，变量的值等于 `undefined`

2. 对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。**真正被赋值的是后者，而不是前者**。如下面代码块，`foo` 是匹配的模式，`baz` 才是变量。真正被赋值的是变量 `baz`，而不是模式 `foo`。

```
let { foo: baz } = { foo: 'aaa', bar: 'bbb' };
baz // "aaa"
```

3. 对象的解构赋值也可以嵌套赋值，如果解构模式是嵌套的对象，而且子对象所在的父属性不存在，那么将会报错。

```
// 嵌套赋值
let obj = {};
let arr = [];
({ foo: obj.prop, bar: arr[0] } = { foo: 123, bar: true });
obj // {prop:123}
arr // [true]
```

```
// 报错
let {foo: {bar}} = {baz: 'baz'};
// 子对象bar所在的父属性foo不存在
```

4. 可以取到继承的属性

```
const obj1 = {};
const obj2 = { foo: 'bar' };
Object.setPrototypeOf(obj1, obj2);
```

```
const { foo } = obj1;
```

foo // "bar"

`// 对象obj1的原型对象是obj2。obj1通过继承obj2的属性获得foo属性，解构赋值可以取到这个属性`

5. 默认值：默认值生效的条件是，对象的属性值严格等于 `undefined`。

示例：

```
let ldh = {
    name: '刘德华',
    age: 18,
    sing: function () {
        console.log('唱歌');
    }
}
let { name, age, sing } = ldh
console.log(name);
console.log(age);
console.log(sing);
let { sex } = ldh
console.log(sex); //undefined
```

```
刘德华
18
f () {
    console.log('唱歌');
}
undefined
```

10.3用途

1.交换变量的值

```
let x = 1;
let y = 2;

[x, y] = [y, x];
```

简单又简洁明了

2.从函数返回多个值

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

```
// 返回一个数组
function example() {
    return [1, 2, 3];
}
let [a, b, c] = example();

// 返回一个对象
function example() {
    return {
        foo: 1,
        bar: 2
    };
}
let { foo, bar } = example();
```

3.函数参数的定义

解构赋值可以方便地将一组参数与变量名对应起来。

```
// 参数是一组有次序的值
function f([x, y, z]) { ... }
f([1, 2, 3]);

// 参数是一组无次序的值
function f({x, y, z}) { ... }
f({z: 3, y: 2, x: 1});
```

4. 提取 JSON 数据

解构赋值对提取 JSON 对象中的数据，尤其有用。

```
let jsonData = {
  id: 42,
  status: "OK",
  data: [867, 5309]
};

let { id, status, data: number } = jsonData;
console.log(id, status, number);
// 42, "OK", [867, 5309]
```

上面代码可以快速提取 JSON 数据的值。

(5) 函数参数的默认值

```
jQuery.ajax = function (url, { //函数的两个形参：url和一个对象参数，默认第二个参数为空对象
  async = true,
  beforeSend = function () {},
  cache = true,
  complete = function () {},
  crossDomain = false,
  global = true,
  // ... more config
} = {}) {
  // ... do stuff 函数内容代码块
};
```

指定参数的默认值，就避免了在函数体内部再写 `var foo = config.foo || 'default foo';` 这样的语句。

11. 字符串的扩展

11.1 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。

作用：

1. 声明字符串

2. 用来定义多行字符串，**内容中可以直接出现换行符**

3. **变量拼接，在字符串中嵌入变量。语法：** ``${变量名}``，该变量可以是函数、对象、字符串等，如果花括号中的值不是字符串，将按照一般的规则转为字符串，如果花括号内部是一个字符串，将会原样输出。

注意点：4.如果在模板字符串中需要使用反引号，则前面要用反斜杠\转义。

示例：

```
// 新声明字符串方式 ``  
// 1. 声明  
let str = `字符串`;  
console.log(str, typeof str);  
// 2. 用来定义多行字符串，内容中可以直接出现换行符  
let star = `  


- 刘德华
- 张学友

`  
console.log(star);  
console.log(star.trim()); // 使用trim()消除ul标签前的换行  
// 3. 变量拼接，在字符串中嵌入变量  
let love = '双皮奶'  
let myLove = `我喜欢${love}` // 我喜欢双皮奶  
console.log(myLove);  
// 4. 如果在模板字符串中需要使用反引号，则前面要用反斜杠\转义。  
let st = `\`yoho`  
console.log(st); // `yoho
```

字符串 string

```
<ul>- 刘德华
- 张学友
</ul>  
<ul>- 刘德华
- 张学友
</ul>  
我喜欢双皮奶  
`yoho`
```

11.2方法：trimStart(), trimEnd()

trimStart() 和 trimEnd() 方法的行为与 trim() 一致。

trimStart() 消除字符串头部的空格，trimEnd() 消除尾部的空格。

它们返回的都是新字符串，不会修改原始字符串。

对于tab 键、换行符等不可见的空白符号也有效。

示例：

```
const s = ' abc ';  
  
s.trim() // "abc"  
s.trimStart() // "abc "  
s.trimEnd() // " abc"
```

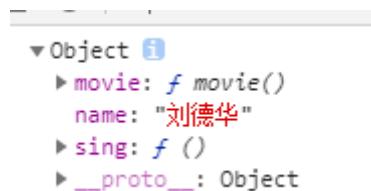
trimLeft() 是 trimStart() 的别名，trimRight() 是 trimEnd() 的别名。

12.对象的扩展

12.1 对象的简化写法

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，简写属性和方法。

```
let name = '刘德华'
let sing = function () {
    console.log('唱歌');
}
// 属性简写
let star = {
    name, // 等同于 name:name，属性名就是变量名，属性值就是变量值
    sing, // sing:sing,
    /* movie: function () {
        console.log('电影');
    } */
    // 方法简写，可以省略function关键字
    movie() {
        console.log('电影');
    }
}
console.log(star);
```

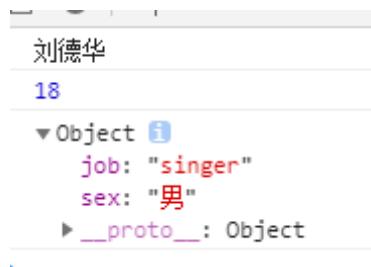


12.2 对象的扩展运算符

在ES9中为对象提供了像数组一样的rest参数和扩展运算符。

解构赋值

```
function get({ name, age, ...args }) {
    console.log(name); // 刘德华
    console.log(age); // 18
    console.log(args); // {sex: ..., job: ...}
}
get({
    name: '刘德华',
    age: 18,
    sex: '男',
    job: 'singer'
})
```



用途：

- 1) 可以使用解构赋值拷贝对象，注意是浅拷贝 `let { ...o3 } = o2;`，
- 2) 也可以反过来用此合并对象 `let o={...o1,...o2,...o3}` |
- 3) 扩展某个函数的参数，引入其他操作

```
function baseFunction({ a, b }) {  
    // ...  
}  
function wrapperFunction({ x, y, ...restConfig }) {  
    // 使用 x 和 y 参数进行操作  
    // 其余参数传给原始函数  
    return baseFunction(restConfig);  
}
```

函数 `wrapperFunction` 在 `baseFunction` 的基础上进行了扩展，能够接受多余的参数，并且保留原始函数的行为。

注意点：

1. 扩展运算符的解构赋值，**不能复制继承自原型对象的属性**

例1：

```
let o1 = { a: 1 };  
let o2 = { b: 2 };  
o2.__proto__ = o1;  
let { ...o3 } = o2;  
o3 // { b: 2 }  
o3.a // undefined
```

对象 `o3` 复制了 `o2`，但是只复制了 `o2` 自身的属性，没有复制它的原型对象 `o1` 的属性。

例2：

```
const o = Object.create({ x: 1, y: 2 });  
o.z = 3;  
  
let { x, ... newObj } = o;  
let { y, z } = newObj;  
x // 1  
y // undefined  
z // 3
```

原因： `newObj` 不能继承 `o` 原型对象里的 `y` 属性

2. ES6 规定，变量声明语句之中，如果使用解构赋值，扩展运算符后面必须是一个变量名，而不能是一个解构赋值表达式

```
let { x, ...{ y, z } } = o;  
// SyntaxError 报错
```

12.3 对象方法的扩展

1.Object.is()

ES5 比较两个值是否相等，只有两个运算符：相等运算符（`==`）和严格相等运算符（`===`）。它们都有缺点，前者会自动转换数据类型，后者的 `NaN` 不等于自身，以及 `+0` 等于 `-0`。

`object.is()` 用来**比较两个值是否严格相等**，与严格比较运算符（`==`）的行为基本一致。

特点：**不同之处只有两个：一是 `+0` 不等于 `-0`，二是 `NaN` 等于自身。**

```
// 1.object.is() 判断两个值是否完全相等
console.log(Object.is(120, 120)); //true
console.log(Object.is(NaN, NaN)); //true
console.log(NaN === NaN); //false
console.log(Object.is(+0, -0)); //false
console.log(+0 === -0); //true
```

2.Object.assign()

`object.assign()` 方法用于**对象的合并**，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

语法：

```
Object.assign(target, source, source...)
```

第一个参数是目标对象，后面的参数都是源对象

方法的规则：

1.如果目标对象与源对象**有同名属性**，或多个源对象有同名属性，则**后面的属性会覆盖前面的属性**。

```
// 2.object.assign() 对象的合并
let star1 = {
  name: '刘德华',
  age: 18,
  skill() {
    console.log('唱歌');
  }
}
let star2 = {
  name: '张学友',
  age: 20,
  sex: '男'
}
console.log(Object.assign(star1, star2));
```



说明：绿色框是同名属性，后面的属性覆盖前面的；红色框是目标对象独有的；绿色框是源对象独有的。

2.如果只有一个参数，`Object.assign()` 会直接返回该参数

```
const obj = {a: 1};  
Object.assign(obj) === obj // true
```

3.如果该参数不是对象，则会先转成对象，然后返回

```
typeof Object.assign(2) // "object"
```

由于`undefined`和`null`无法转成对象，所以如果它们作为参数，就会报错

```
Object.assign(undefined) // 报错  
Object.assign(null) // 报错
```

4.非对象参数出现在源对象的位置（即非首参数）。处理规则：首先，这些参数都会转成对象，如果无法转成对象，就会跳过。

只有字符串的包装对象，会产生可枚举属性，以数组形式拷贝进入。其他类型的值不会产生效果

```
// 如果undefined和null不在首参数，就不会报错  
let obj = {a: 1};  
Object.assign(obj, undefined) === obj // true  
Object.assign(obj, null) === obj // true  
  
// 其他类型的值（即数值、字符串和布尔值）不在首参数，也不会报错  
// 但是，字符串会以数组形式，拷贝入目标对象，其他值都不会产生效果  
const v1 = 'abc'; //字符串，以数组形式拷贝进入  
const v2 = true; //布尔型  
const v3 = 10; //数值型  
  
const obj = Object.assign({}, v1, v2, v3);  
console.log(obj); // { "0": "a", "1": "b", "2": "c" }
```

5.`Object.assign()`拷贝的属性是有限制的，只拷贝源对象的自身属性（不拷贝继承属性），不拷贝不可枚举的属性（`enumerable: false`）。

6.属性名为`Symbol`值的属性，也会被`Object.assign()`拷贝。

```
Object.assign({ a: 'b' }, { [Symbol('c')]: 'd' })  
// { a: 'b', Symbol(c): 'd' }
```

注意点：

1. `Object.assign()`方法实行的是**浅拷贝**，而不是深拷贝
- 2.遇到同名属性，`Object.assign()`的处理方法是**替换**，而不是添加。
3. `Object.assign()`可以用来处理数组，但是会**把数组视为对象**，把数组视为属性名为0、1、2的对象。

```
Object.assign([1, 2, 3], [4, 5])  
// [4, 5, 3]
```

常见用途

(1) 为对象添加属性

```
class Point {  
    constructor(x, y) {  
        Object.assign(this, {x, y});  
    }  
}
```

通过 `Object.assign()` 方法，将 `x` 属性和 `y` 属性添加到 `Point` 类的对象实例

3.proto属性, `Object.setPrototypeOf()`, `Object.getPrototypeOf()`

`__proto__` 属性

`__proto__` 属性（前后各两个下划线），用来读取或设置当前对象的原型对象，该属性没有写入 ES6 的正文，而是写入了附录。

因此，无论从语义的角度，还是从兼容性的角度，都**不要使用这个属性**，而是使用下面的 `Object.setPrototypeOf()`（写操作）、`Object.getPrototypeOf()`（读操作）、`Object.create()`（生成操作）代替。

`Object.setPrototypeOf()`

`Object.setPrototypeOf` 方法的作用与 `__proto__` 相同，用来**设置一个对象的原型对象**（prototype），**返回参数对象本身**。它是 ES6 正式推荐的设置原型对象的方法。

语法：

```
// 格式  
Object.setPrototypeOf(object, prototype)
```

示例：

```
let series = {  
    name: '西游记'  
}  
let characters = {  
    group: ['唐僧', '孙悟空', '猪八戒', '沙僧']  
}  
Object.setPrototypeOf(series, characters)
```

将 `characters` 对象设置为 `series` 对象的原型对象，可以从 `series` 对象调用 `characters` 对象的属性

```
console.log(series.group);
```

```
▶ (4) ["唐僧", "孙悟空", "猪八戒", "沙僧"]
```

`Object.getPrototypeOf()`

用于**读取一个对象的原型对象**

接上个示例

```
console.log(Object.getPrototypeOf(series))
```

```
▼ {group: Array(4)} ⓘ  
▶ group: (4) ["唐僧", "孙悟空", "猪八戒", "沙僧"]  
▶ __proto__: Object
```

```
console.log(series);
```

```
▼ {name: "西游记"} ⓘ  
name: "西游记"  
▼ __proto__:  
▶ group: (4) ["唐僧", "孙悟空", "猪八戒", "沙僧"]  
▶ __proto__: Object
```

4.Object.keys(), Object.values(), Object.entries()

三个方法都供 `for...of` 循环使用

Object.keys()

`object.keys` 方法，返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键名。

```
// object.keys() 返回一个数组，由可枚举属性的键名组成  
let star = {  
    name: '张学友',  
    age: 20,  
    sex: '男',  
    works: ['饿狼传说', '吻别', '情书']  
}  
console.log(Object.keys(star));
```

```
▶ (4) ["name", "age", "sex", "works"]
```

Object.values()

`object.values` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值。

```
// object.values() 返回一个数组，由可枚举属性的键值组成  
console.log(Object.values(star));
```

```
▼ (4) ["张学友", 20, "男", Array(3)] ⓘ  
0: "张学友"  
1: 20  
2: "男"  
▶ 3: (3) ["饿狼传说", "吻别", "情书"]  
length: 4  
▶ proto : Array(0)
```

注意点：

1. `Object.values` 会过滤属性名为 `Symbol` 值的属性。

```
Object.values({ [Symbol()]: 123, foo: 'abc' });  
// ['abc']
```

2. `Object.values` 方法的参数是一个字符串，会返回各个字符组成的一个数组。会先将字符串转成一个类似数组的对象。

```
Object.values('foo')
// ['f', 'o', 'o']
```

3. 如果参数不是对象，`Object.values` 会先将其转为对象。由于数值和布尔值的包装对象，都不会为实例添加非继承的属性。所以，`Object.values` 会返回空数组。

```
Object.values(42) // []
Object.values(true) // []
```

Object.entries()

`Object.entries` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对数组。将对象转为二维数组

```
// Object.entries() 返回一个数组，由属性名和属性值的键值对构成
console.log(Object.entries(star));
```

```
▼ (4) [Array(2), Array(2), Array(2), Array(2)] ⓘ
  ► 0: (2) ["name", "张学友"]
  ► 1: (2) ["age", 20]
  ► 2: (2) ["sex", "男"]
  ► 3: (2) ["works", Array(3)]
    length: 4
  ► __proto__: Array(0)
```

除了返回值不一样，该方法的行为与 `Object.values` 基本一致。

注意点：

1. `Object.entries` 会过滤属性名为 `Symbol` 值的属性

2. `Object.entries` 的基本用途

1) 遍历对象的属性

```
// 基本用途1) 遍历对象的属性
for (let [k, i] of Object.entries(star)) {
  // k 是属性名, i 是属性值, 转换成JSON形式
  console.log(`"${JSON.stringify(k)}": ${JSON.stringify(i)}`);
```

```
"name": "张学友"
"age": 20
"sex": "男"
"works": ["饿狼传说", "吻别", "情书"]
```

2) 将对象转为真正的Map结构

```
// 2) 将对象转为真正的Map结构
let m = new Map(Object.entries(star))
console.log(m);
console.log(m.get('works'));
```

```
▶ Map(4) {"name" => "张学友", "age" => 20, "sex" => "男", "works" => Array(3)}  
▶ (3) ["饿狼传说", "吻别", "情书"]
```

遗漏: Object.create()

`object.create()`方法创建一个新对象，使用现有的对象来提供新创建的对象的`__proto__`

语法:

```
Object.create(proto[,propertiesObject])
```

参数说明:

`proto` 新创建对象的原型对象。

`propertiesObject` 可选。需要传入一个对象，该对象的属性类型参照`Object.defineProperties()`的第二个参数。

```
// Object.create() 创建一个对象  
let o = Object.create(Object.prototype, {  
    name: {  
        // 设置值  
        value: '刘德华',  
        // 属性特性  
        writable: true,  
        configurable: true,  
        enumerable: true  
    }  
})  
console.log(o);
```

```
▼ Object ⓘ  
  name: "刘德华"  
  ▶ __proto__: Object
```

5.Object.getOwnPropertyDescriptors()

`Object.getOwnPropertyDescriptors()`方法，返回指定对象所有自身属性（非继承属性）的描述对象。

返回一个对象，所有原对象的属性名都是该对象的属性名，对应的属性值就是该属性的描述对象。

```
// 5.Object.getOwnPropertyDescriptors() 返回指定对象所有自身属性的描述对象  
let obj = {  
    name: 'auue',  
    age: 18  
}  
console.log(Object.getOwnPropertyDescriptors(obj));
```

```
▼ {name: {...}, age: {...}} ⓘ  
  ▶ age: {value: 18, writable: true, enumerable: true, configurable: true}  
  ▶ name: {value: "auue", writable: true, enumerable: true, configurable: true}  
  ▶ __proto__: Object
```

用途1：对象合并，可以拷贝get和set属性（`assign()`方法不能拷贝，因为只会拷贝值，不拷贝方法）

```

let source = {
    name: '被拷贝',
    value: 'clone',
    set foo(value) {
        console.log(value);
    }
}
let target = {}
function shadowMerge(target, source) {
    return Object.defineProperties(target,
Object.getOwnPropertyDescriptors(source));
}
let res = shadowMerge(target, source)
console.log(res);

```

```

▼ {name: "被拷贝", value: "clone"} ⓘ
  name: "被拷贝"
  value: "clone"
  ▶ set foo: f foo(value)
  ▶ __proto__: Object
  >

```

用途2：配合 `Object.create()` 方法，将对象属性克隆到一个新对象。这属于浅拷贝。

```

// 2) 配合Object.create()方法，将对象属性克隆到一个新对象，浅拷贝。
let clone = Object.create(Object.getPrototypeOf(source),
Object.getOwnPropertyDescriptors(source))
console.log(clone);

```

```

▼ {name: "被拷贝", value: "clone"} ⓘ
  name: "被拷贝"
  value: "clone"
  ▶ set foo: f foo(value)
  ▶ __proto__: Object

```

6.Object.fromEntries

`Object.fromEntries()` 方法是 `Object.entries()` 的逆操作，用于将一个键值对数组（二维数组）转为对象。

用途：是将键值对的数据结构还原为对象，

1.特别适合将 Map 结构转为对象。

2.配合 `URLSearchParams` 对象，将查询字符串转为对象

`URLSearchParams`对象的方法用于处理 URL 的查询字符串，只能去除字符串起始位置的 ?

```

// Object.fromEntries()将键值对转化为对象
let res = Object.fromEntries([
    ['name', '孙悟空'],
    ['age', 18]
])
console.log(res);
// 1) 适合将Map结构转换为对象
let m = new Map()
m.set('name', '沙僧')

```

```

let o = Object.fromEntries(m)
console.log(o);
// URLSearchParams对象的方法用于处理 URL 的查询字符串，只能去除字符串起始位置的？
let paramsString = "q=URLUtilsSearchParams&topic=api"
let searchParams = new URLSearchParams(paramsString);
// console.log(searchParams);
// 使用for...of遍历查询字符串
for (let p of searchParams) {
    console.log(p);
}
// 2) 配合URLSearchParams对象，将查询字符串转为对象
console.log(Object.fromEntries(new URLSearchParams('?
foo=bar&baz=qux')));
// { foo: "bar", baz: "qux" }

```

```

▶ {name: "孙悟空", age: 18}
▶ {name: "沙僧"}
▶ (2) ["q", "URLUtilsSearchParams"]
▶ (2) ["topic", "api"]
▶ {foo: "bar", baz: "qux"}

```

12.4 链判断运算符

链判断运算符 `?.` 运算符，直接在链式调用的时候判断，左侧的对象是否为 `null` 或 `undefined`。如果是的，就不再往下运算，而是返回 `undefined`（这点类似逻辑中断，即短路机制）

示例：

要读取db下的host属性安全的写法如下：

注意：host属性在对象的第三层，所以需要判断三次，每一层是否有值

```

// 要读取db下的host属性安全的写法如下
function main(config){
    // 判断 参数存在？ 参数是否有db对象？ 参数内的db对象是否有host属性？
    // host属性在对象的第三层，所以需要判断三次，每一层是否有值
    // const dbHost = config && config.db && config.db.host;
    // 使用“链判断运算符”?..，简化写法。
    const dbHost = config?.db?.host;
    console.log(dbHost);
}

main({
    db: {
        host: '192.168.1.100'
        username: 'root'
    }
})

```

链判断运算符有三种用法：

- `obj?.prop` // 对象属性
- `obj?.[expr]` // 同上
- `func?(...args)` // 函数或对象方法的调用

示例：

```
a?.b()  
// 等同于  
a == null ? undefined : a.b()  
// 里面的a.b不是函数，不可调用，报错
```

13. 函数的扩展

13.1 箭头函数

ES6 允许使用“箭头” (`=>`) 定义函数。

箭头函数的声明与调用：

省略了`function`关键字，添加了`=>`

```
// 声明箭头函数，省略了function关键字  
let fn = (a, b) => {  
    return a + b  
}  
// 等同于  
/* let fn = function (a, b) {  
    return a + b  
} */  
// 调用箭头函数，与之前一样  
let result = fn(1, 2)  
console.log(result); //3
```

注意点：

1. `this`是静态的，始终指向函数声明时所在作用域下的`this`的值，无法修改，不可变。其本质是箭头函数根本没有自己的`this`，导致内部的`this`就是外层代码块的`this`

```
function getName1() {  
    console.log(this.name);  
}  
let getName2 = () => {  
    console.log(this.name);  
}  
window.name = 'window'  
let obj = {  
    name: 'auue'  
}  
// 直接调用  
getName1() //window  
getName2() //window  
// call()改变this指向调用  
getName1.call(obj) //auue  
getName2.call(obj) //window, 始终指向函数声明时所在作用域下的this的值
```

window
window
auue
window

2.由于箭头函数的this是静态的，因此箭头函数不适用于定义对象的方法与元素绑定事件，适用于数组、定时器的方法回调，见案例。

3.由于箭头函数没有自己的this，不能作为构造实例化对象，不可以使用new命令，否则会抛出一个错误

```
let Star = (name, age) => {
    this.name = name,
    this.age = age
}
let ldh = new Star('刘德华', 18)
console.log(ldh); // TypeError
```

4.不能使用arguments变量，该对象在函数体内不存在。

```
let f = () => {
    console.log(arguments);
}
f(1, 2, 3) //ReferenceError: arguments is not defined
```

5.箭头函数的简写：

1)当形参只有一个时，可以省略小括号()

2)当代码体只有一条语句且有return，可以省略花括号{}，此时return必须省略。且语句的执行结果就是函数的返回值

3)花括号被解释为代码块，因此如果箭头函数直接返回一个对象，必须在对象外面加上括号，否则会报错

```
// 4.箭头函数的简写
/* let add = (a) => {
    return (a + a);
} */
// 1) 当形参只有一个时，可以省略小括号()
// 简写为
let add = a => {
    return (a + a);
}
console.log(add(3)); //6
// 2) 当代码体只有一条语句时，可以省略花括号{}，此时return必须省略。且语句的执行结果就是函数的返回值
// 再简写
let pow = a => a * a
console.log(pow(3)); //9
// 3) 大括号被解释为代码块，因此如果箭头函数直接返回一个对象，必须在对象外面加上括号，否则会报错
let merge = a => ({ add: a + a, pow: a * a })
console.log(merge(4)); // {add: 8, pow: 16}
```

案例：

1.点击div 2s后背景颜色变粉色

```

let div = document.querySelector('div')
div.addEventListener('click', function () {
    /* setTimeout(function(){
        // this.style.backgroundColor='pink'// 这里的this指向window, 而不是
div
    },2000) */
    setTimeout(() => {
        this.style.backgroundColor = 'pink' //使用箭头函数, 此时的this绑定定义时所在的作用域(绑定事件内), 指向div
    }, 2000)
})

```

2.从数组中返回偶数的元素

```

// 2.从数组中返回偶数的元素
let arr = [1, 3, 2, 6]
// 普通函数
let result = arr.filter(function (item) {
    return item % 2 === 0
})
console.log(result); // [2, 6]
// 箭头函数,更简洁点
let res = arr.filter(item => item % 2 === 0)
console.log(res); // [2, 6]

```

3.请问下面的代码之中有几个this及输出结果

```

function foo() {
    return () => {
        return () => {
            return () => {
                console.log('id:', this.id);
            };
        };
    };
}

var f = foo.call({ id: 1 });

var t1 = f.call({ id: 2 })();
var t2 = f().call({ id: 3 })();
var t3 = f().call({ id: 4 });

```

t1/t2/t3结果输出为1，代码中只有foo函数一个this，里面的箭头函数都指向这个this，因此结果为1

13.2参数默认值

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。一般放在尾部，这样比较容易看出省略了哪些参数，如果非尾部的参数设置默认值，实际上这个参数是没法省略的。

```
// 1. 形参初始值，具有默认值的参数。一般放在尾部（语法规范）
function add(a, b, c = 10) {
    return a + b + c
}
let result = add(1, 2)
console.log(result); // 3, 如果c没有默认值，那么c为undefined，输出结果就为NaN
```

注意点：

参数默认值不是传值的，而是每次都重新计算默认值表达式的值

```
// 2. 参数默认值不是传值的，而是每次都重新计算默认值表达式的值
let x = 99
function f(p = x + 1) {
    console.log(p);
}
f() // 100
x = 100
f() // 101
// 每次调用f函数，都会重新计算p=x+1，而不是默认p等于100
```

与解构赋值结合

```
// 3. 与解构赋值结合
function connect({ host, username, password, port = 3306 }) {
    console.log(host);
    console.log(username);
    console.log(password);
    console.log(port);
}
connect({
    host: 'xxx.com',
    username: 'root',
    password: 'root',
})
```

注意：

1. 如果函数参数是对象，**只为对象的属性设置了默认值，这样不能省略该参数**，如上个代码块，会报错。

```
connect()
```

2. 如果函数参数是对象，**设置了对象默认值，那么可以省略该参数**，如下代码块。出现了双重默认值

```
function connect({ host, username, password, port = 3306 } = {}) {}
connect()
```

```
undefined
undefined
undefined
3306
```

练习：下面两种写法有什么差别？

```
// 写法一
function m1({x = 0, y = 0} = {}) {
    return [x, y];
}

// 写法二
function m2({x, y} = {x: 0, y: 0}) {
    return [x, y];
}
```

方法1：设置了**函数参数的默认值**为空对象{}，又设置了**对象解构赋值的默认值**，双重默认值。

方法2：设置了函数参数的默认值为 { x: 0, y: 0 }，默认为x/y值为0的对象，但没有设置对象解构赋值的默认值。

判断下面的值：

```
// 1. 函数没有参数的情况
m1()
m2()

// 2. x 和 y 都有值的情况
m1({x: 3, y: 8})
m2({x: 3, y: 8})

// x 有值, y 无值的情况
m1({x: 3})
m2({x: 3})

// x 和 y 都无值的情况
m1({})
m2({})
```

1.都为 [0, 0]

2.都为 [3, 8]

3. [3, 0] / [3, undefined]

4. [0, 0] / [undefined, undefined]

5. [0, 0] / [undefined, undefined]

对于m2函数，只要有值传入，默认值就没用了，输出的就是传入的值

13.3rest参数

ES6 引入 **rest 参数**（形式为 ... 变量名），用于获取函数的多余参数，这样就不需要使用 arguments 对象了。rest 参数搭配的变量是一个**数组**，该变量将多余的参数放入数组中。

注意：

1. 注意rest参数必须放在参数**最后**,否则会报错
2. arguments 对象不是数组，而是一个类似数组的对象。**rest 参数就是一个真正的数组**，因此可以使用数组的方法，如push()

```
// ES6引入rest参数，用于获取参数的实参，用来代替arguments
// ES5获取实参的方式
function data() {
    console.log(arguments);
}
data('刘德华', '张学友', '郭富城')
// rest参数获取实参方法
function data1(...args) {
    console.log(args); //返回形式是数组
}
data1('刘德华', '张学友', '郭富城')
// 注意rest参数必须放在参数最后
```



14.数组的扩展

14.1 扩展运算符

扩展运算符 (spread) 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。简单理解：去掉了中括号[]

```
// [...] 扩展运算符能将【数组】转换为逗号分隔的【参数序列】
let arr = [1, 2, 3] //=>1,2,3
function out() {
    console.log(arguments);
}
out(arr)
out(...arr)//相当于out(1,2,3)
```

```

▼ Arguments [Array(3), callee: f, Symbol(Symbol.iterator): f] ⓘ
  ► 0: (3) [1, 2, 3]
  ► callee: f out()
    length: 1
  ► Symbol(Symbol.iterator): f values()
  ► __proto__: Object

▼ Arguments(3) [1, 2, 3, callee: f, Symbol(Symbol.iterator): f] ⓘ
  0: 1
  1: 2
  2: 3
  ► callee: f out()
    length: 3
  ► Symbol(Symbol.iterator): f values()
  ► __proto__: Object

```

>

说明: `out(...arr)` 传入了三个参数, 扩展运算符将数组转换为了逗号分隔的参数序列

注意, 只有函数调用时, 扩展运算符才可以放在小括号中, 否则会报错。

```

(...[1, 2])
// Uncaught SyntaxError: Unexpected number, 扩展运算符所在的括号不是函数调用。

console.log(...[1, 2])
// Uncaught SyntaxError: Unexpected number, 扩展运算符所在的括号不是函数调用。

console.log(...[1, 2])
// 1 2

```

14.2 扩展运算符的应用

1. 数组的合并

`concat()`方法用于连接两个或多个字符串, 不改变原有字符串, 返回一个新字符串

```

// 1. 数组的合并
let arr1 = ['刘德华', '张学友']
let arr2 = ['小撒', '小尼']
//ES5
let merge = arr1.concat(arr2) //concat()方法用于连接两个或多个字符串, 不改变原有字符串, 返回一个新字符串
let mer = [...arr1, ...arr2] //相当于['刘德华', '张学友', '小撒', '小尼']
console.log(merge);
console.log(mer);
//上面两种方法属于浅拷贝, 使用时注意下

```

```

► (4) ["刘德华", "张学友", "小撒", "小尼"]
► (4) ["刘德华", "张学友", "小撒", "小尼"]

```

将`arr1`合并到`arr2`尾部, 别忘记`push()`返回值是新数组的长度, `push()`方法的参数不能是数组

```

// 直接将arr1合并到arr2尾部
arr2.push(...arr1); //push()方法的参数不能是数组, 使用扩展运算符转为参数序列
console.log(arr2);

```

```

► (4) ["小撒", "小尼", "刘德华", "张学友"]

```

2.数组的克隆

```
// 2.数组的克隆，注意是浅拷贝，复制的是内存地址，修改cloneArr1会直接导致arr1的变化
let cloneArr1 = [...arr1]
console.log(cloneArr1);
```

```
▶ (2) ["刘德华", "张学友"]
```

注意是浅拷贝，复制的是内存地址，修改cloneArr1会直接导致arr1的变化

3.把伪数组转为真正的数组

```
// 3.把伪数组转为真正的数组
let divs = document.querySelectorAll('div')
let divArr = [...divs]
console.log(divs); //实际上是对象类型
console.log(divArr);
```

```
▶ (2) [“刘德华”, “张学友”]
▼ NodeList(3) [div, div, div] ⓘ
  ▶ 0: div
  ▶ 1: div
  ▶ 2: div
  length: 3
  ▼ __proto__: NodeList
    ▶ entries: f entries()
    ▶ forEach: f forEach()
    ▶ item: f item()
    ▶ keys: f keys()
    length: (...)
    ▶ values: f values()
    ▶ constructor: f NodeList()
    ▶ Symbol(Symbol.iterator): f values()
    Symbol(Symbol.toStringTag): "NodeList"
    ▶ get length: f length()
    ▶ __proto__: Object
  ▶ (3) [div, div, div] ⓘ
    ▶ 0: div
    ▶ 1: div
    ▶ 2: div
    length: 3
    ▶ __proto__: Array(0)
```

4.替代函数的apply()

```
// 4.替代函数的apply()
// ES5
console.log(Math.max.apply(Math, [14, 3, 22])); //22
// ES6
console.log(Math.max(...[14, 3, 22])); //22
```

14.3map和reduce方法

map()

map() 方法返回一个新数组，数组中的元素为原始数组元素调用函数处理后的值，按照原始数组元素顺序依次处理元素。

注意： map() 不会改变原始数组。

语法：

```
array.map(function(currentValue, index, arr), thisvalue)
```

参数说明：

1. `function(currentValue, index, arr)` **函数**，数组中的每个元素都会执行这个函数，**必须**。

`currentValue` **必须。当前元素的值**

`index` 可选。当前元素的索引值

`arr` 可选。当前元素属于的数组对象

2. `thisValue` 可选。对象作为该执行回调时使用，传递给函数，用作 "this" 的值。如果省略了 thisValue，或者传入 null、undefined，那么回调函数的 this 为全局对象。

用途：进行映射,数据处理，可以将数字转换为字符等。

案例：

将60分以下归为不及格，60-75为及格，75-90为良好，90-100为优秀。

```
// 将60分以下归为不及格，60-75为及格，75-90为良好，90-100为优秀。
let arr = [50, 73, 98, 38, 80, 62]
let arr1 = arr.map(function (item) {
    if (item > 90) {
        return '优秀'
    } else if (item > 75) {
        return '良好'
    } else if (item > 60) {
        return '及格'
    } else {
        return '不及格'
    }
})
console.log(arr);
console.log(arr1);
// 如果只化为两档，可以使用三元表达式和箭头函数让代码简化
let arr2 = arr.map(item => item > 60 ? '及格' : '不及格')
console.log(arr2);
```

```
▶ (6) [50, 73, 98, 38, 80, 62]
▼ (6) ["不及格", "及格", "优秀", "不及格", "良好", "及格"]
  0: "不及格"
  1: "及格"
  2: "优秀"
  3: "不及格"
  4: "良好"
  5: "及格"
  length: 6
  ▶ __proto__: Array(0)
▶ (6) ["不及格", "及格", "及格", "不及格", "及格", "及格"]
  0: "不及格"
  1: "及格"
  2: "及格"
  3: "不及格"
  4: "及格"
  5: "及格"
  length: 6
  ▶ __proto__: Array(0)
```

reduce()

reduce() 方法接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，**最终计算为一个值**。因此适合求平均值、求和等。

语法：

```
array.reduce(function(total, currentValue, currentIndex, arr), initialValue)
```

参数说明：

1. `function(total, currentValue, index, arr)` 必须。用于执行每个数组元素的函数。

`total` 必须。**初始值, 或者计算结束后的返回值**。

`currentValue` 必须。当前元素的值

`currentIndex` 可选。当前元素的索引

`arr` 可选。当前元素所属的数组对象

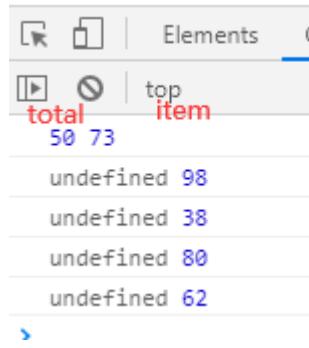
2. `initialValue` 可选。传递给函数的初始值

注意：

1. `total` 是**初始值, 或者计算结束后的返回值**，因此第二次回调如果没有返回值，`total`为`undefined`，所以必须要有返回值。

2. `currentValue` 是当前元素的值，第一次回调的值是`total`之后的那个值。

```
let arr = [50, 73, 98, 38, 80, 62]
let arr1 = arr.reduce(function (total, item, index) {
  console.log(total, item);
})
```



示例：求平均值

```
let arr = [50, 73, 98, 38, 80, 62]
let res = arr.reduce(function (total, item, index) {
    // console.log(total, item);
    if (index == arr.length - 1) {
        // 遍历到最后一个元素就求和求平均
        return (total + item) / index
    } else {
        return total + item //先求和后平均
    }
})
console.log(res);
```



14.4 Array.prototype.includes

`includes` 方法返回一个布尔值，用来检测数组中是否包含某个元素，与字符串的 `includes` 方法类似（第二个参数默认从0）。该方法是ES7新特性。

格式：

```
arr.includes(searchElement, fromIndex)
```

参数说明：

`searchElement` 必须。需要查找的元素值。

`fromIndex` 可选，表示搜索的起始位置，默认为0。如果第二个参数为负数，则表示倒数的位置，如果这时它大于数组长度（比如第二个参数为-4，但数组长度为3），则会重置为从0开始。

示例：

```
let mingzu = ['西游记', '红楼梦', '三国演义', '水浒传']
// Array.prototype.includes 检测数组中是否包含某个元素
console.log(mingzu.includes('西游记')); //true
console.log(mingzu.includes('金瓶梅')); //false
[1, 2, 3].includes(3, 3); // false
[1, 2, 3].includes(3, -1); // true
```

注意点：

1.与 `indexOf` 方法的区别：`indexOf` 方法有两个缺点，一是不够语义化，它的含义是找到参数值的第一个出现位置，所以要去比较是否不等于 `-1`，表达起来不够直观。二是，它内部使用严格相等运算符（`==`）进行判断，这会导致对 `Nan` 的误判。`includes` 方法没有这些问题

2.**Map 和 Set 数据结构有一个 `has` 方法，需要注意与 `includes` 区分**，`Map` 结构的 `has` 方法，是用来查找键名的；`Set` 结构的 `has` 方法，是用来查找值的

14.5 `flat()`, `flatMap()`

`flat([deep])` 方法将多维数组转化为低维数组，参数是深度，为整数。

默认参数是1，表示‘拉平’一层，维度降低一层。如3维降为2维

示例：

```
[1, 2, [3, [4, 5]]].flat()  
// [1, 2, 3, [4, 5]]  
[1, 2, [3, [4, 5]]].flat(2)  
// [1, 2, 3, 4, 5]
```

`flatMap()` 方法对原数组的每个成员执行一个函数（相当于执行 `Array.prototype.map()`），然后对返回值组成的数组执行 `flat()` 方法。该方法**返回一个新数组，不改变原数组**。（即 `map+flat` 两个方法缝合）

`flatMap()` 只能展开一层数组

示例：

```
// 相当于 [[2, 4], [3, 6], [4, 8]].flat()  
[2, 3, 4].flatMap((x) => [x, x * 2])  
// [2, 4, 3, 6, 4, 8]
```

15. Symbol

15.1 概述

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。

JS的七种数据类型记忆：U-undefined、S-String/Symbol、O-Object、N-null/number、B-boolean

创建语法：

```
// 1. 通过Symbol函数生成  
let s = Symbol()  
console.log(s, typeof s);
```

Symbol() "symbol"

Symbol特点：

1. `Symbol` 的值是唯一，用来解决命名冲突的问题。

2. `Symbol` 函数前不能使用 `new` 命令，否则会报错。生成的 `Symbol` 是一个原始类型的值，是一种类似于字符串的数据类型，不是对象。

3. `Symbol` 函数可以接受一个**字符串作为参数，表示对 `Symbol` 实例的描述**，主要作用是在控制台显示，或者转为字符串时容易区分。

`Symbol` 函数的参数只是表示对当前 `Symbol` 值的描述，因此**相同参数的 `symbol` 函数的返回值并不相等**。

如果 `Symbol` 的参数是一个对象，就会调用该对象的 `toString` 方法，将其转为字符串，然后才生成一个 `Symbol` 值。

```
let s1 = Symbol('刘德华')
let s2 = Symbol('刘德华')
console.log(s1, s2);
console.log(s1 === s2);
```

```
Symbol(刘德华) Symbol(刘德华)
false
```

4. `Symbol` 值不能与其他类型的值进行任何运算，会报错

```
// Symbol 值不能与其他类型的值进行运算，会报错
let result = s + 100 //Uncaught SyntaxError: Identifier 'result' has
already been declared
let result = s > 100
let result = s + s
```

5. `Symbol` 值可以转为布尔型

```
//转为布尔型
let sym = Symbol();
Boolean(sym) // true
!sym // false

if (sym) {
  ...
}
```

15.2 `Symbol.prototype.description`

[ES2019](#) 提供了一个实例属性 `description`，可以直接返回 `Symbol` 的描述。

```
let s2 = Symbol('刘德华')
console.log(s2.description); //刘德华
```

15.3 作为属性名的 `Symbol`

`Symbol` 值是唯一的，所以 **Symbol 值很适合作为标识符用于对象的属性名**，就能保证不会出现同名的属性。这对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖。

symbol 作为属性名的三种写法：

```
let sym = Symbol('1')
// symbol 作为属性名
```

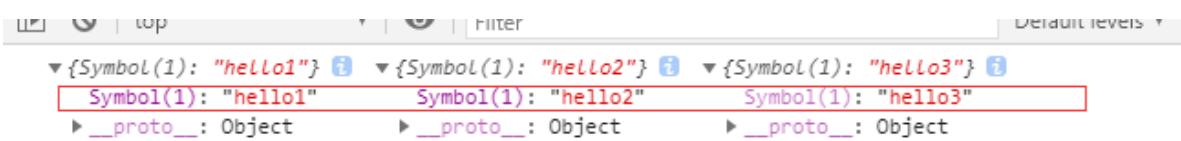
```

// 第一种写法
let a = {}
a[sym] = 'hello1'

// 第二种写法
let b = {
  [sym]: 'hello2'
}

// 第三种写法
let c = {}
Object.defineProperty(c, sym, {
  value: 'hello3'
})
console.log(a, b, c);

```



注意：

1. Symbol 值作为对象属性名时，**不能用点运算符** `a.sym=....` 因为点运算符后面总是字符串，会将a.后面的属性名sym读取为字符串，而不是一个Symbol值。
2. 在对象的内部，使用 Symbol 值定义属性时，Symbol 值必须放在**方括号**之中。`[sym]:` 如果 `sym` 不放在方括号中，该属性的键名就是字符串 `sym`，而不是 `sym` 所代表的那个 Symbol 值。

Symbol作为属性值

Symbol 类型还可以用于定义一组常量，保证这组常量的值都是不相等的

```

let log = {}
log.levels = {
  DEBUG: Symbol('debug'),
  INFO: Symbol('info'),
  WARN: Symbol('warn')
}
console.log(log.levels.DEBUG);

```

Symbol(debug)

15.4 Symbol属性名遍历

Symbol 作为属性名时，不能被 `for...in` 循环遍历，但可以使用 `Reflect.ownKeys()` 方法和 `Object.getOwnPropertySymbols()` 方法获取。

1. `Object.getOwnPropertySymbols(对象名)` 方法

```

// 1.Object.getOwnPropertySymbols()方法，获取指定对象的所有 symbol 属性名
// 注意：
// 1. 只能获取Symbol属性名，普通的属性名不能获取
// 2. 返回的形式是一个数组
let obj = {
  [Symbol('a')]: 'a',
  [Symbol('b')]: 'b',
  c: 'c'
}

```

```
}

let syms = Object.getOwnPropertySymbols(obj)// [Symbol(a), Symbol(b)]
console.log(syms);
/* for (let i in obj) {
    console.log(i); //只能输出普通的属性名
} */
```

▶ (2) [Symbol(a), Symbol(b)]

注意：

- 1.只能获取Symbol属性名，普通的属性名不能获取
- 2.返回的形式是一个数组

2. Reflect.ownKeys(对象名) 方法

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

▶ (3) ["c", Symbol(a), Symbol(b)]

注意:该方法可以返回**所有类型的属性名**，包括常规键名和 Symbol 键名，返回的形式是一个**数组**

应用：由于以 Symbol 值作为键名，不会被常规方法遍历得到。我们可以利用这个特性，**为对象定义一些非私有的、但又希望只用于内部的方法。**

示例：

```
let size = Symbol('size');
class Collection {
    constructor() {
        this[size] = 0;
    }

    add(item) {
        this[this[size]] = item;
        this[size]++;
    }

    static sizeof(instance) {
        return instance[size];
    }
}

let x = new Collection();
console.log(Collection.sizeof(x));

x.add('foo');
console.log(Collection.sizeof(x));
/* add('foo') {
    this[0] = 'foo'; //给对象添加了一个属性及属性名
    this[size]++; //this[Symbol(size)]=1,修改了属性值
} */
console.log(x);
console.log(Object.keys(x));
console.log(Object.getOwnPropertySymbols(x));
console.log(Reflect.ownKeys(x));
```

```
0
1
▼Collection {0: "foo", Symbol(size): 1} ⓘ
  0: "foo"
  Symbol(size): 1
  ► __proto__: Object
  ► ["0"]
  ► [Symbol(size)]
  ► (2) ["0", Symbol(size)]
```

15.5Symbol.for(), Symbol.keyFor()

`Symbol.for()`方法可以接受一个字符串作为参数，然后搜索有没有以该参数作为名称的Symbol值。如果有，就返回这个Symbol值（所指地址相同），否则就新建一个以该字符串为名称的Symbol值，并将其注册到全局。

Symbol()与Symbol.for()方法的区别

相同点：都能创建symbol

区别：

`Symbol.for()`会被登记在全局环境中供搜索，不会每次调用就返回一个新的Symbol类型的值，而是会先检查给定的key是否已经存在，如果不存在才会新建一个值。

`Symbol()`每次调用都会返回一个新的Symbol类型的值，没有登记机制，每次调用都会返回一个不同的值。

```
// 1. Symbol.for()方法，可以使用同一个Symbol值，
let s1 = Symbol('1dh')
let s2 = Symbol('1dh')
console.log(s1 === s2); //false
let s3 = Symbol.for('1dh')
let s4 = Symbol.for('1dh')
console.log(s3 === s4); //true
```

`Symbol.keyFor()`方法返回一个已登记的Symbol类型值的key。

注意，`Symbol.for()`为Symbol值登记的名字，是全局环境的，不管有没有在全局环境运行。

```
// 2. Symbol.keyFor()方法返回一个已登记的 Symbol 类型值的key
console.log(Symbol.keyFor(s1)); //undefined
console.log(Symbol.keyFor(s3)); //1dh
```

15.6内置的 Symbol 值

ES6 提供了 11 个内置的 Symbol 值，指向语言内部使用的方法。

Symbol.hasInstance

对象的`Symbol.hasInstance`属性，指向一个内部方法。当对象使用`instanceof`运算符时会自动调用这个方法。

```
// 对象的Symbol.hasInstance属性，指向一个内部方法。当其他对象使用instanceof运算符，判断是否为该对象的实例时，会调用这个方法。
class Person { //Person类
    [Symbol.hasInstance](foo) { //Symbol.hasInstance方法
        console.log(foo) //方法代码块
        return foo instanceof Array;
    }
}
let arr = [1, 2]
let person = new Person()
console.log(arr instanceof person); //Person的实例对象在使用 instanceof运算符时，自动调用Symbol.hasInstance方法
```

▶ (2) [1, 2]

true

>

Symbol.isConcatSpreadable

对象的Symbol.isConcatSpreadable属性是一个布尔值，表示是否展开。

用于数组 concat() 时，数组的默认行为是展开，Symbol.isConcatSpreadable的默认值是undefined，true也是展开；设置为false，才是不展开。

```
let arr1 = ['c', 'd'];
['a', 'b'].concat(arr1, 'e') // ['a', 'b', 'c', 'd', 'e']
arr1[Symbol.isConcatSpreadable] // 默认为undefined 展开

let arr2 = ['c', 'd'];
arr2[Symbol.isConcatSpreadable] = false; //不展开
['a', 'b'].concat(arr2, 'e') // ['a', 'b', ['c', 'd'], 'e']
```

用于对象 concat() 时，默认行为是不展开，正好与数组相反。属性设为true，才可以展开。

```
let obj = {length: 2, 0: 'c', 1: 'd'};
['a', 'b'].concat(obj, 'e') // ['a', 'b', obj, 'e']

obj[Symbol.isConcatSpreadable] = true;
['a', 'b'].concat(obj, 'e') // ['a', 'b', 'c', 'd', 'e']
```

16.Iterator 和 for...of 循环

16.1 Iterator (遍历器) 的概念

遍历器 (Iterator) 是一种接口(可以理解为属性)，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。

Iterator 的作用：

一是为各种数据结构，提供一个统一的、简便的访问接口；

二是使得数据结构的成员能够按某种次序排列；

三是 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。**

Iterator 的遍历过程：

- (1) 创建一个指针对象，指向当前数据结构的起始位置。即遍历器对象本质上，就是一个**指针对象**。
- (2) 第一次调用指针对象的 `next` 方法，可以将指针指向数据结构的第一个成员。
- (3) 第二次调用指针对象的 `next` 方法，指针就指向数据结构的第二个成员。
- (4) 不断调用指针对象的 `next` 方法，直到它指向数据结构的结束位置。
- (5) **每一次调用 `next` 方法，返回一个包含 `value` 和 `done` 两个属性的对象。** 其中，`value` 属性是当前成员的值，`done` 属性是一个布尔值，表示遍历是否结束，`false` 表示没有结束。

需要自定义遍历数据时，就要想到遍历器

```
let f4 = ['唐僧', '孙悟空', '猪八戒', '沙僧']
// for..of遍历
for (let i of f4) {
    console.log(i);
}
for (let i in f4) {
    console.log(i);
}
```

唐僧
孙悟空
猪八戒
沙僧
0
1
2
3

注意`for..of`与`for..in`的区别：`for...of`循环读取键值，`for...in`循环读取键名。

16.2 默认 Iterator 接口

原生具备 `Iterator` 接口的数据结构如下

- `Array`
- `Map`
- `Set`
- `String`
- `TypedArray`
- 函数的 `arguments` 对象
- `NodeList` 对象

注意对象没有默认部署 `Iterator` 接口，是因为对象的哪个属性先遍历，哪个属性后遍历是不确定的，需要开发者手动指定。

默认的 `Iterator` 接口部署在数据结构的 `Symbol.iterator` 属性，或者说，一个数据结构只要具有 `Symbol.iterator` 属性，就可以认为是“可遍历的”

`Symbol.iterator` 属性本身是一个**函数**，就是当前数据结构默认的遍历器生成函数。

```
console.log(f4);
```

```
► values: f values()
▼ Symbol(Symbol.iterator): f values()
  arguments: (...)

  caller: (...)

  length: 0

  name: "values"
► __proto__: f ()
► [[Scopes]]: Scopes[0]
```

执行 `Symbol.iterator` 属性，会返回一个遍历器对象。该对象的根本特征就是具有 `next` 方法。

```
let iterator = f4[Symbol.iterator]() //遍历器对象
console.log(iterator);
```

```
▼ Array Iterator {} ⓘ
  ▼ __proto__ : Array Iterator
    ► next: f next()
      Symbol(Symbol.toStringTag): "Array Iterator"
    ► __proto__: Object
```

每次调用 `next` 方法，都会返回一个代表当前成员的信息对象，具有 `value` 和 `done` 两个属性。其中，`value` 属性是当前成员的值，`done` 属性是一个布尔值，表示遍历是否结束，`false` 表示没有结束，`true` 表示结束。

```
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

```
► {value: "唐僧", done: false}
► {value: "孙悟空", done: false}
► {value: "猪八戒", done: false}
► {value: "沙僧", done: false}
► {value: undefined, done: true}
```

16.3 自定义 Iterator 接口

一个对象如果要具备可被 `for...of` 循环调用的 `Iterator` 接口，就必须在 `Symbol.iterator` 的属性上部署遍历器生成方法。

要素：

1. `[Symbol.iterator]` 属性，本身是个函数
2. 执行属性会返回一个带有 `next` 方法的遍历器对象
3. 调用 `next` 方法会返回一个具有 `value` 和 `done` 属性的对象并将指针指向数据结构的下一个成员

```
let f4 = {
  start: 'daTang',
  end: 'west',
  member: ['唐僧', '孙悟空', '猪八戒', '沙僧'],
```

```

[Symbol.iterator]: function () { //可遍历必须要有Symbol.iterator属性,
该属性本身是个函数
    // 定义一个索引，通过索引输出当前成员的值
    let index = 0
    let that = this
    return {
        //执行Symbol.iterator属性，会返回一个遍历器对象
        next: function () { //对象内有next方法
            if (index < that.member.length) {
                return {
                    //每次调用next方法，会返回一个带value和done属性的
                    对象，且指针指向数组的下一个成员
                    value: that.member[index++], //value属性是当前成员
                    的值
                    done: false //false表示遍历未结束
                }
            } else {
                return {
                    value: undefined,
                    done: true
                }
            }
        }
    }
}
for (let i of f4) {
    console.log(i);
}

```



对于遍历器对象来说，done: false和value: undefined属性可以省略

```

// 代码进一步简写，使用三元表达式
// 对于遍历器对象来说，done: false和value: undefined属性可以省略
return {
    next: function () {
        return index < that.member.length ? { value:
that.member[index++]} : { done: true }
    }
}

```

对于伪数组（存在数值键名和length属性）添加Iterator接口，最简单的方法就是直接引入数组的Iterator接口，普通对象使用该方法无效。

```
NodeList.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];
```

16.4 调用 Iterator 接口的场合

1. 扩展运算符

扩展运算符 (...) 也会调用默认的 Iterator 接口。

```
// 例一
var str = 'hello';
[...str] // ['h', 'e', 'l', 'l', 'o']

// 例二
let arr = ['b', 'c'];
['a', ...arr, 'd']
// ['a', 'b', 'c', 'd']
```

实际上，这提供了一种简便机制，可以将任何部署了 Iterator 接口的数据结构，转为数组。也就是说，只要某个数据结构部署了 Iterator 接口，就可以对它使用扩展运算符，将其转为数组。

17. Generator 函数

17.1 基本概念

Generator 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。

语法上，可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态，执行 Generator 函数会返回一个遍历器对象。返回的遍历器对象，可以依次遍历 Generator 函数内部的每一个状态。

形式上，Generator 函数是一个普通函数，但有两个特征：

1. `function` 关键字与函数名之间有一个星号；
2. 函数体内部使用 `yield` 表达式，定义不同的内部状态（`yield` 意为“产出”）。

注意：星号位置没有规定，可以靠 `function` 关键字，可以放中间，也可以靠函数名

调用 Generator 函数，会返回一个遍历器对象，代表 Generator 函数的内部指针。

示例：

```
// Generator 函数就是一个普通的函数
// 函数声明：1. function 与函数名之间有个*号
function* gen() {
    // 2. 内部使用 yield 表达式，定义不同的内部状态
    // 该函数有三个状态：hello, world 和 return 语句
    console.log(1);
    console.log(2);
    yield 'hello'
    yield 'world'
    console.log(3);
    return 'ending'
}
// Generator 函数的调用方法与普通函数一样，但是调用后并不执行，返回的一个指向内部状态
// 的指针对象即遍历器对象 Iterator
let g = gen()
console.log(g);
```

```

▼ gen {<suspended>} ⓘ
  ▼ __proto__: Generator
    ▼ __proto__: Generator
      ► constructor: GeneratorFunction {prototype: Generator}
        ► next: f next()
        ► return: f return()
        ► throw: f throw()
        ► Symbol(Symbol.toStringTag): "Generator"
        ► __proto__: Object
      [[GeneratorLocation]]: 1-Generator函数语法.html:20
      [[GeneratorState]]: "suspended"
      ► [[GeneratorFunction]]: f* gen()
      ► [[GeneratorReceiver]]: Window
      ► [[Scopes]]: Scopes[3]

```

每次调用遍历器对象的 `next` 方法，让指针移向下一个状态，直到遇到下一个 `yield` 表达式（或 `return` 语句）

并返回一个有着 `value` 和 `done` 两个属性的对象。`value` 属性表示当前的内部状态的值，即 `yield` 表达式后面那个表达式的值；`done` 属性是一个布尔值，表示是否遍历结束。

```

//next运行逻辑：调用遍历器对象的next方法，使得指针移向下一个状态，直到遇到下一个yield表达式（或
//return语句）
console.log(g.next());

//执行yield表达式及之前的语句
/*console.log(1);
console.log(2);
yield 'hello'*/

```

```

1
2
► {value: "hello", done: false}

```

```

console.log(g.next());
console.log(g.next());
console.log(g.next());

```

```

► {value: "world", done: false}
3
► {value: "ending", done: true}
► {value: undefined, done: true}

```

17.2 yield 表达式

Generator 函数返回的遍历器对象，只有调用 `next` 方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。`yield` 表达式就是暂停标志。

遍历器对象的 `next` 方法的运行逻辑：

- (1) 遇到 `yield` 表达式，就暂停执行后面的操作，并将 `yield` 后面表达式的值，作为返回的对象的 `value` 属性值。
- (2) 下一次调用 `next` 方法时，再继续往下执行，直到遇到下一个 `yield` 表达式。

(3) 如果没有再遇到新的 `yield` 表达式，就一直运行到函数结束，直到 `return` 语句为止，并将 `return` 语句后面的表达式的值，作为返回的对象的 `value` 属性值。

(4) 如果该函数没有 `return` 语句，则返回的对象的 `value` 属性值为 `undefined`

遇到 `yield` 就停止执行语句，并返回对象 {`value`: `yield` 表达式内的值}；再调用 `next`，在上个 `yield` 语句后继续执行，同理；若没有新的 `yield` 就找 `return` 语句，`return` 也没有那么返回 {`value`: `undefined`}

注意：

1. `yield` 表达式后面的表达式，只有当调用 `next` 方法、内部指针指向该语句时才会执行，因此等于为 JavaScript 提供了手动的“惰性求值”

```
function* gen() {
  yield 123 + 456;
}

//next移动到yield语句才会计算表达式的值
```

2. `yield` 表达式与 `return` 语句的区别：`return` 语句不具备位置记忆的功能。一个函数里面，只能执行一次（或者说一个）`return` 语句，但是可以执行多次（或者说多个）`yield` 表达式。

正因如此，Generator 函数可以返回一系列的值，因为可以有任意多个 `yield`。

3. `yield` 表达式只能在 Generator 函数内使用，否则会报错（如普通函数内，如数组的 `forEach` 函数）。

4. `yield` 表达式如果用在另一个表达式之中，必须加小括号

```
function* demo() {
  console.log('Hello' + yield); // SyntaxError
  console.log('Hello' + yield 123); // SyntaxError

  console.log('Hello' + (yield)); // OK
  console.log('Hello' + (yield 123)); // OK
}
```

5. `yield` 表达式用作函数参数或放在赋值表达式的右边，可以不加括号

```
function* demo() {
  foo(yield 'a', yield 'b'); // OK 作为函数参数，可以不加小括号
  let input = yield; // OK 作为赋值表达式的右边，可以不加小括号
}
```

17.3 与 Iterator 接口的关系

任意一个对象的 `Symbol.iterator` 方法，等于该对象的遍历器生成函数，调用该函数会返回该对象的一个遍历器对象。

Generator 函数就是遍历器生成函数，因此可以把 Generator 赋值给对象的 `Symbol.iterator` 属性，从而使得该对象具有 Iterator 接口。

```

var myIterable = {};
myIterable[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  yield 3;
};

[...myIterable] // [1, 2, 3]
//myIterable对象具有了 Iterator 接口，可以被...运算符遍历

```

Generator 函数执行后，返回一个**遍历器对象**。该对象本身也具有 `Symbol.iterator` 属性，执行后返回自身。

```

let g = gen()
// 与接口的关系
console.log(g); //g遍历器对象也拥有Iterator接口，执行后返回自己
console.log(g[Symbol.iterator]() == g); //true

```

```

▼ gen {<suspended>} ⓘ
  ▼ __proto__: Generator
    ▼ __proto__: Generator
      ► constructor: GeneratorFunction {prototype: Generator, Symbol(Symbol.toStringTag): "Generator"}
      ► next: f next()
      ► return: f return()
      ► throw: f throw()
      ► Symbol(Symbol.iterator): f [Symbol.iterator]()
      ► __proto__: Object
[[[GeneratorLocation]]]: 1-Generator函数语法.html:15
[[ConstructorLocation]]: "1-Generator"

```

17.4next 方法的参数

`yield` 表达式本身没有返回值，或者说总是返回 `undefined`。`next` 方法可以带一个参数，该参数就会被当作上一个 `yield` 表达式的返回值。

示例：

```

// next方法可以带一个参数，该参数就会被当作上一个yield表达式的返回值。
function* f() {
  // 定义一个无限运行的Generator函数
  for (let i = 0; true; i++) {
    let reset = yield i
    if (reset) { i = -1 }
  }
}
var g = f()
console.log(g.next());
// 相当于
/* i=0
true
yield 0
没有进行reset的赋值操作 reset:undefined */
console.log(g.next());
// 相当于
/* i++ //i变为1

```

```

yiled 1
没有进行reset的赋值操作 reset:undefined */
console.log(g.next(true));
//相当于
/*
将参数作为上一个yield表达式的返回值
let reset = true //此时i还是1
if (reset) { i = -1 } //i=-1
i++ //i=0
yield 0
*/

```

```

▶ {value: 0, done: false}
▶ {value: 1, done: false}
▶ {value: 0, done: false}

<html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>next 方法的参数</title>
9 </head>
10 <body>
11   <script>
12     // next方法可以带一个参数，该参数就会被重置
13     function* f() {
14       // 定义一个无限运行的Generator函数
15       for (let i = 0; true; i++) {
16         let reset = yield i
17         if (reset) { i = -1 }
18       }
19     }

```

说明：

next方法没有参数，每次运行到yield表达式就暂停执行后面语句，变量reset的值总是undefined。

当next方法带一个参数true时，变量reset就被重置为这个参数（即true），因此i会等于-1，下一轮循环就会从-1开始递增。

意义：通过next参数，可以在Generator函数运行的不同阶段，从外部向内部注入不同的值，从而调整函数行为。

示例：

```

function* foo(x) {
  var y = 2 * (yield (x + 1));
  var z = yield (y / 3);
  return (x + y + z);
}

var a = foo(5);
a.next() // Object{value:6, done:false}
a.next() // Object{value:NaN, done:false}
a.next() // Object{value:NaN, done:true}

```

第二次运行next方法不带参数，导致y的值等于 $2 * \text{undefined}$ （即NaN），除以3以后还是NaN

第三次运行 `next` 方法不带参数，所以 `z` 等于 `undefined`，返回对象的 `value` 属性等于 `5 + NaN + undefined`，即 `Nan`。

如果 `next` 方法带参数

```
var b = foo(5);
b.next() // { value:6, done:false }
b.next(12) // { value:8, done:false }
b.next(13) // { value:42, done:true }
```

第二次调用 `next` 方法，将上一次 `yield` 表达式(`yield (x + 1)`)的值设为 `12`，因此 `y` 等于 `24`，返回 `y / 3` 的值 `8`

第三次调用 `next` 方法，将上一次 `yield` 表达式(`yield (y / 3)`)的值设为 `13`，因此 `z` 等于 `13`，这时 `x` 等于 `5`，`y` 等于 `24`，所以 `return` 语句的值等于 `42`。

从语义上讲，第一个 `next` 方法用来启动遍历器对象，不用带有参数。

17.5 for...of 循环

`for...of` 循环可以自动遍历 Generator 函数运行时生成的 `Iterator` 对象，且此时不再需要调用 `next` 方法。

```
function* f() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
    return 6;
}
//注意：一旦next方法的返回对象的done属性为true就停止循环，因此无法输出最后一个
//return返回的值
for (let v of f()) {
    console.log(v);
}
```

```
1
2
3
4
5
> |
```

注意：一旦 `next` 方法的返回对象的 `done` 属性为 `true` 就会停止循环，因此无法输出最后一个 `return` 返回的值

17.6 Generator 函数的案例

案例1

要求：1s后控制台输出111，2s后输出222，3s后输出333

1. 可以使用定时器嵌套实现

但存在几个很严重的问题：代码不断缩进，可能超出编辑区，不方便维护，可读性差，称此为 **回调地狱**，尽量不要这样使用

```
// 1. 可以使用定时器嵌套实现
// 但存在几个很严重的问题：代码不断缩进，可能超出编辑区，不方便维护，可读性差
// 因此，称此为 回调地狱
setTimeout(() => {
    console.log(111);
    setTimeout(() => {
        console.log(222);
        setTimeout(() => {
            console.log(333);
        }, 3000);
    }, 2000);
}, 1000);
```

2. 使用生成器函数结合定时器实现

注意其中的原理

```
// 2. 使用生成器完成
function one() {
    setTimeout(() => {
        console.log(111);
        // 注意next方法还是放在定时器函数内部
        g.next()
    }, 1000);

}

function two() {
    setTimeout(() => {
        console.log(222);
        // 注意next方法还是放在定时器函数内部
        g.next()
    }, 2000);
}

function three() {
    setTimeout(() => {
        console.log(333);
        // 注意next方法还是放在定时器函数内部
        g.next()
    }, 3000);
}

// 使用生成器函数的yield实现调用函数
function* gen() {
    yield one()
    yield two()
    yield three()
}

let g = gen()
g.next() //先调用第一个函数，在第一个函数内部再调用next()方法让其调用下一个函数，有点类似多米诺骨牌的原理
```

111

222

333

、

案例2

要求：先获取用户数据，再获取订单数据，最后获取商品数据。并且可以对相应数据进行操作——使用 `next()` 方法添加参数从而传入数据，然后可以对 user、order 等数据进行操作。

```
// 要求：先获取用户数据，再获取订单数据，最后获取商品数据。
function getUser() {
    setTimeout(() => {
        let data = '用户数据'
        // 调用next方法，将数据传入
        g.next(data)
    }, 1000);
}

function getOrder() {
    setTimeout(() => {
        let data = '订单数据'
        g.next(data)
    }, 1000);
}

function getGoods() {
    setTimeout(() => {
        let data = '商品数据'
        g.next(data)
    }, 1000);
}

function* gen() {
    let user = yield getUser()
    console.log(user);
    let order = yield getOrder()
    console.log(order);
    let goods = yield getGoods()
    console.log(goods);
}

let g = gen()
g.next()
```



从上面两个案例可以发现：生成器函数结合定时器可以让几个函数/方法按顺序依次执行，也是其异步编程的应用

18. Set 和 Map 数据结构

18.1 Set

ES6 提供了新的数据结构 **Set**。它类似于数组（数学中的集合），但是成员的值都是唯一的，**没有重复的值**。集合实现了 iterator 接口，可以使用「扩展运算符」和 `for...of` 进行遍历。

语法：

```
let s = new Set()  
// 可以接受参数（有Iterator接口的数据类型）  
let s1 = new Set([1, 9, 3, 2, 3])
```

集合的**属性和方法**：

- 1) `size` 返回集合的元素个数
- 2) `add()` 增加一个新元素，**返回当前集合**
- 3) `delete()` 删除元素，返回boolean值
- 4) `has()` 检测集合中是否包含某个元素，返回boolean值
- 5) `clear()` 清除所有成员，**没有返回值**

特点：

1. **自动去重**，可以利用此去除数组中的重复成员
2. 可以使用「扩展运算符」和 `for...of` 进行遍历
3. 本质数据类型是对象类型

```
console.log(typeof s1); //object
```

4. 可以接受参数（有Iterator接口的数据类型）

示例：

```
let s1 = new Set([1, 9, 3, 2, 3])  
// 特点：自动去重，可以利用此去除数组中的重复成员  
console.log(s1); //1,9,3,2  
// 可以使用for..of遍历  
for (let i of s1) {  
    console.log(i);  
}  
// 属性和方法  
// 1) size属性，返回集合的元素个数  
console.log(s1.size); //4  
// 2) add()方法添加一个新元素，并返回当前集合  
console.log(s1.add('add'));  
// 3) delete()方法删除属性，返回布尔值，删除成功返回true  
console.log(s1.delete(1));  
// 4) has()方法检测集合中是否包含某个元素，返回布尔值，true表示存在  
console.log(s1.has('add'));  
// 5) clear()方法清空集合  
s1.clear()  
console.log(s1); //空集合
```

```

▶ Set(4) {1, 9, 3, 2}
1
9
3
2
4
▶ Set(5) {1, 9, 3, 2, "add"}
true
true
▶ Set(0) {}

```

18.2 Set集合实践

案例：

数组去重、交集、并集、差集

注意：

1. 扩展运算符和 Set 结构相结合，可以去除数组的重复成员

2. 差集是所有属于A且不属于B的元素构成的集合，数组元素筛选条件正好与交集的相反

3. 交集和并集的实现：Set结构和 filter 方法相结合

```

let arr = [1, 4, 2, 7, 2, 4]
let arr1 = [1, 9, 3]
// 1. 数组去重，注意去重后是对象类型，使用扩展运算符和[]变为数组类型
result = [...new Set(arr)]
console.log(result);
// 2. 交集，取相同的数组元素
// 使用数组的过滤方法filter()实现，过滤的条件是数组元素相同
let union = [...new Set(arr)].filter(item => {
    // 需要将第二个数组先去重
    let s = new Set(arr1)
    // 使用Set的has()方法判断是否有相同的元素
    if (s.has(item)) {
        return true
    } else {
        return false
    }
})
// 代码简化
let u = [...new Set(arr)].filter(item => new Set(arr1).has(item))
console.log(union);
console.log(u);
// 3. 并集，两个数组先分别去重，合到一起后再次去重，返回数组类型
let intersect = [...new Set([...new Set(arr), ...new Set(arr1)])]
console.log(intersect);
// 4. 差集，返回arr-arr1，与交集相反
let diff = [...new Set(arr)].filter(item => !(new Set(arr1).has(item)))
console.log(diff);

```

```
▶ (4) [1, 4, 2, 7]
▶ [1]
▶ [1]
▶ (6) [1, 4, 2, 7, 9, 3]
▶ (3) [4, 2, 7]
```

18.3 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。集合实现了 iterator 接口，可以使用「扩展运算符」和 `for...of` 进行遍历

与对象最大的区别在于“键”， 对象中的键只能是字符串类型，而 Map 里的键可以是任何类型。

语法

```
let m = new Map();
```

Map的属性和方法

- 1) `size` 返回 Map 的元素个数
- 2) `set(键名,键值)` 增加一个新元素，返回当前 Map
- 3) `delete(键名)` 删除某个键名及其对应的键值，返回 boolean 值
- 4) `has(键名)` 检测 Map 中是否包含某个元素，返回 boolean 值
- 5) `clear()` 清除所有成员，返回 `undefined`
- 6) `get(键名)` 返回键名对象的键值

注意： Map 可以接受一个数组作为参数。该数组的成员是一个个表示键值对的数组。

```
const map = new Map([
  ['name', '张三'],
  ['title', 'Author']
]);

map.size // 2
map.has('name') // true
map.get('name') // "张三"
map.has('title') // true
map.get('title') // "Author"
```

示例：

```
// 声明
let m = new Map()
// 属性和方法
// 1) set(键名,键值)添加
m.set('name', '孙悟空') //键名和键值都为字符串
m.set('do', function () { //键名为字符串，键值为函数
  console.log('七十二变');
})
let key = {
```

```

    teacher: '唐僧'
}
m.set(key, ['猪八戒', '沙僧']) //键名为对象，键值为数组
console.log(m);
// 2) size属性，返回元素个数
console.log(m.size);
// 3) delete(键名) 删除某个键名及其对应的键值
m.delete('name')
console.log(m);
// 4) get(键名)返回键名对象的键值
console.log(m.get(key));
// 5) has(键名)检测是否有该键名，返回布尔值
console.log(m.has('do'));
// 6) clear()清空，返回undefined
// console.log(m.clear()); //undefined

// 使用for...of遍历
for (let i of m) {
  console.log(i);
}

```

```

▶ Map(3) {"name" => "孙悟空", "do" => f, ...} => Array(2)
3
▼ Map(2) {"do" => f, ...} => Array(2) ⓘ
  ▼ [[Entries]]
    ▼ 0: {"do" => function () { //键名为字符串，键值为函数 console.log('七十二变'); }}
      ▶ key: "do"
      ▶ value: f ()
    ▼ 1: {Object => Array(2)}
      ▶ key: {teacher: "唐僧"}
      ▶ value: (2) ["猪八戒", "沙僧"]
    size: (...)

  ▶ __proto__: Map
▶ (2) ["猪八戒", "沙僧"]
true
▶ (2) ["do", f]
▶ (2) [...], Array(2)
>

```

babel.js编译

[Babel](#) 是一个广泛使用的 ES6 转码器，可以将 ES6 代码转为 ES5 代码，从而在老版本的浏览器执行。这意味着，你可以用 ES6 的方式编写程序，又不用担心现有环境是否支持。

<https://babeljs.io/>

方法1：引入js文件

```

<script src="browser.min.js"></script>
<script type="text/babel"></script>

```

注意第二条语句，平常我们写的默认是 `<script type="text/javascript"></script>`，可以省略里面的type，但此时我们引入了babel.js，使用的不是js语言，而是babel语言，因此需要改成 `text/babel`，否则低版本浏览器执行仍会报错。

该方法在客户端完成编译，但有两个问题：

- 1.降低代码性能，在客户端编译会有延迟
- 2.本身就不支持低版本的ie

因此不建议使用该方法

方法2：编译JS文件

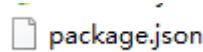
1. 安装Node.js、初始化项目

```
npm init -y
```

```
C:\Users\Daiei\Desktop\前端学习案例\js-ES6-day03>npm init -y
Wrote to C:\Users\Daiei\Desktop\前端学习案例\js-ES6-day03\package.json:

{
  "name": "js-ES6-day03",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

文件目录下多一个json文件



2. 安装 babel-cli

```
npm i @babel/core @babel/cli @babel/preset-env -D
npm i @babel/polyfill -s //安装后可以兼容ie6及以下低版本，太低了不想管了
```

说明：

```
i 是install简写
@babel      babel前面有@，最新版babel写法
@babel/core    babel的核心库
@babel/cli     命令行工具@babel/cli，用于命令行转码
preset        预先设置好的配置
preset-env     environment环境，环境预设，内部自带浏览器的兼容表，会根据你的配置来具体编译
babel
@babel/preset-env   最新转码规则
-D 表示开发时依赖
```

3. 添加执行脚本

在package.json里

点击进入，在scripts里面添加即添加自定义的脚本命令

```
"build": "babel src -d dest" // bulid，构建；src为自定义放js的文件名；-d dest，往哪个目录里输出，自定义目录名字，一般用dist(第一次写错成dest了，以后记得改)
```

```
"scripts": {
  "test": "echo \\\"Error: no test specified\\\" && exit 1",
  "build": "babel src -d dest"
},
```

4.配置文件.babelrc(按理应该放在第一步的)

Babel 的配置文件是 `.babelrc`，存放在项目的根目录下。使用 Babel 的第一步，就是配置这个文件。

该文件用来设置转码规则和插件，基本格式如下：

```
{  
  "presets": [  
    "@babel/env",  
    "@babel/preset-react" //这个是react转码规则，目前没学，也没安装react的转码规则，略  
    去  
  ],  
  "plugins": []  
}
```

5.执行编译

`npm run build`,这个build就是之前添加的脚本命令，必须和之前添加的名字一模一样，否则无法执行。

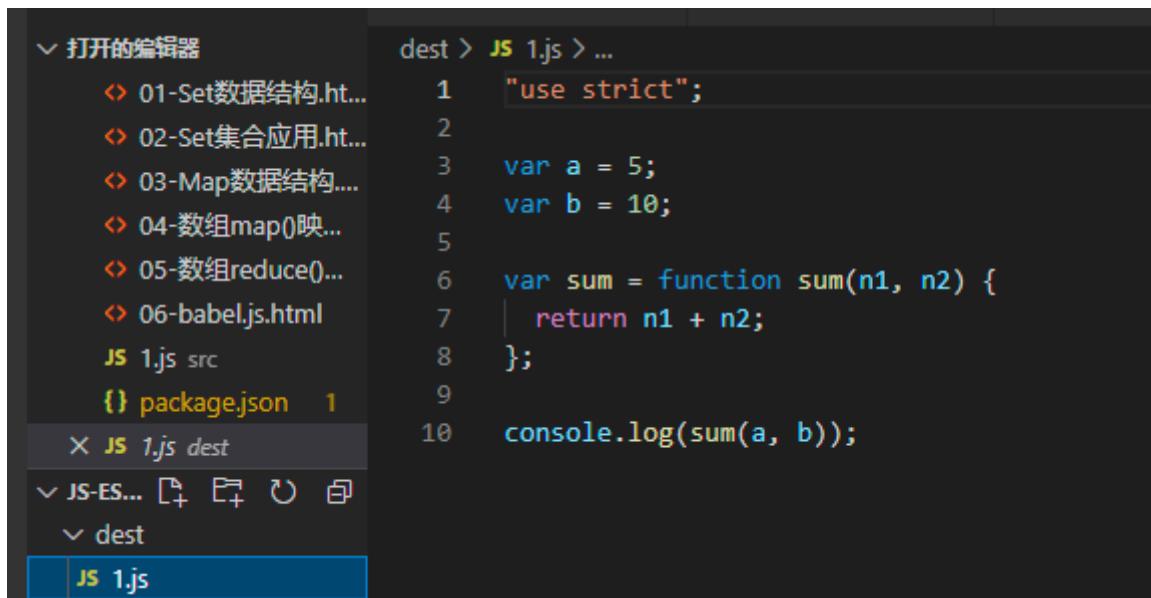
先在src中新建一个js文件，随便写点ES6的语法，如下：

```
let a = 5  
let b = 10  
let sum = (n1, n2) => n1 + n2  
console.log(sum(a, b));
```

再在命令行窗口输入 `npm run build` 进行babel编译

```
C:\Users\Daai\Desktop\前端学习案例\js-ES6-day03>npm run build  
> js-ES6-day03@1.0.0 build  
> babel src -d dest  
  
Successfully compiled 1 file with Babel (928ms).
```

编译完成后可进入dest文件查看编译完后的js文件



The screenshot shows a code editor interface with the following details:

- File Structure:** The left sidebar shows a tree view of files:
 - open editor
 - 01-Set数据结构.html
 - 02-Set集合应用.html
 - 03-Map数据结构....
 - 04-数组map()映...
 - 05-数组reduce0...
 - 06-babel.js.html
 - JS 1.js** (selected in the sidebar)
 - {} package.json
 - X JS 1.js dest
- Destinations:** A dropdown menu at the bottom left shows options: JS-ES... (selected), E+, E-, O, and dest.
- Content:** The main editor area displays the following code:

```
1  "use strict";  
2  
3  var a = 5;  
4  var b = 10;  
5  
6  var sum = function sum(n1, n2) {  
7    return n1 + n2;  
8  };  
9  
10 console.log(sum(a, b));
```

可以发现：添加了 `use strict` 严格模式，将let转为var，箭头函数改为普通函数。

如此，引用该js后，在低版本的ie浏览器（ie7及以上）中也能正常运行，实现兼容

19.Promise

19.1Promise的含义

`Promise` 是异步编程的一种解决方案（解决回调地狱）。但总的作用还是封装。

`Promise` 对象有以下两个特点：

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

(2) **一旦状态改变，就不会再变，任何时候都可以得到这个结果。** `Promise` 对象只有两种状态改变：从`pending` 变为`fulfilled` 和从`pending` 变为`rejected`。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 `resolved`（已定型）。如果改变已经发生了，你再对 `Promise` 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

注意后面的 `resolved` 只指 `fulfilled` 状态，不包含 `rejected` 状态。

`Promise` 对象可以将异步操作以同步操作的流程表达出来（异步、同步融合），并提供统一的接口，使得控制异步操作更加容易。

19.2基本用法

ES6 规定，`Promise` 对象是一个构造函数，用来封装异步操作并可以获取其成功或失败的结果。

语法格式：

```
const promise = new Promise(function(resolve, reject) {  
    // ... some code  
  
    if /* 异步操作成功 */{  
        resolve(value);  
    } else {  
        reject(error);  
    }  
});
```

`Promise` 构造函数接受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`。

`resolve` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“成功”（即从 `pending` 变为 `resolved`），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去。

`reject` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“失败”（即从 `pending` 变为 `rejected`），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

`Promise` 实例生成以后，可以用 `then` 方法分别指定 `resolved` 状态和 `rejected` 状态的回调函数（可以只写一种状态的回调函数）。`then()`方法内有两个回调函数作为参数。

语法格式：

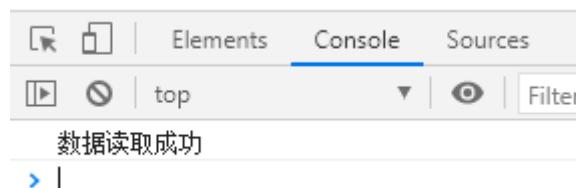
```
promise.then(function(value) {  
    // success  
}, function(error) {  
    // failure  
});
```

示例：

注意失败的reject函数输出报出的错误语句，`console.err(reason);`

```
// 先实例化一个Promise对象，Promise构造函数接收一个函数作为参数，该函数有两个参数  
let p = new Promise(function (resolve, reject) {  
    setTimeout(() => {  
        let value = '数据读取成功'  
        // resolve 异步操作成功，调用resolve函数，将操作结果作为参数传递  
        resolve(value)  
    }, 1000);  
}  
// promise对象的then()，参数是成功和失败的两个回调函数，通常成功函数的参数是value，  
失败函数的参数是reason/error  
p.then(function (value) {  
    // 异步操作成功，调用成功的回调函数  
    console.log(value);  
}, function (reason) {  
    // 异步操作失败，调用失败的回调函数  
    console.err(reason);  
})
```

异步操作成功



异步操作失败

```
let err = '数据读取失败'  
// resolve 异步操作失败，调用reject函数，将操作错误作为参数传递  
reject(err)
```



19.3案例

Promise封装读取文件

`toString()` 方法可把一个 Number 对象转换为一个字符串，并返回结果

```
// 1.引入fs模块  
let fs = require('fs')  
// 2.调用方法异步读取文件
```

```

/* fs.readFile('resources/座右铭.txt', (err, data) => {
    // 读取失败，则抛出错误
    if (err) throw err
    // 读取成功，则输出内容
    console.log(data.toString());
}) */
// 3. 使用Promise封装
let p = new Promise(function (reslove, reject) {
    fs.readFile('resources/座右铭.txt', (err, data) => {
        // 异步操作失败即读取失败
        if (err) reject(err)
        // 读取成功
        reslove(data)
    })
})

p.then(function (value) {
    console.log(value.toString());
}, function (reason) {
    console.log('读取失败');
})

```

未封装前，第2步

```

PS C:\Users\Daai\Desktop\前端学习案例\js-ES6-day02> nodemon 7-Promise读取文件.js
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node 7-Promise读取文件.js`
对于可控的事情，我们要保持谨慎；对于不可控的事情，我们要保持乐观
[nodemon] 1 error - waiting for changes before restarting...

```

封装后

```

PS C:\Users\Daai\Desktop\前端学习案例\js-ES6-day02> node 7-Promise读取文件.js
对于可控的事情，我们要保持谨慎；对于不可控的事情，我们要保持乐观

```

修改文件名使之读取失败

```

PS C:\Users\Daai\Desktop\前端学习案例\js-ES6-day02> node 7-Promise读取文件.js
读取失败

```

Promise封装AJAX请求

优点：通过then()指定回调，结构更清晰；且不会产生回调地狱问题。

```

// 接口地址: https://api.apiopen.top/getJoke
let p = new Promise(function (resolve, reject) {
    let xhr = new XMLHttpRequest()
    xhr.open('GET', 'https://api.apiopen.top/getJoke')
    xhr.send()
    xhr.onreadystatechange = function () {
        if (xhr.readyState == 4) {
            if (xhr.status >= 200 & xhr.status < 300) {
                // 响应成功
                // console.log(xhr.response);
                resolve(xhr.response)
            } else {

```

```

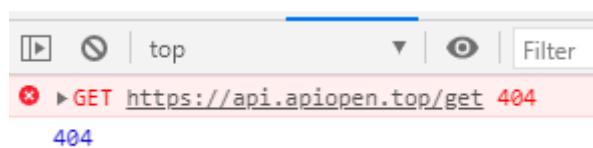
        // 响应失败
        // console.log(xhr.status);
        reject(xhr.status)
    }
}
})
// 通过then()指定回调，结构更清晰
p.then(function (value) {
    console.log(value);
}, function (reason) {
    console.log(reason);
})

```

响应成功



响应失败



19.4 Promise.prototype.then()

`then` 方法的第一个参数是 `resolved` 状态的回调函数，第二个参数是 `rejected` 状态的回调函数，它们都是可选的。作用是为 `Promise` 实例添加状态改变时的回调函数。

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

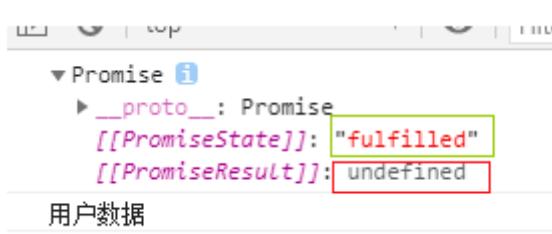
采用箭头函数代码更简洁。

返回一个新Promise对象

1. 调用`then`方法，`then`方法返回的结果是一个新的Promise对象，对象的状态由回调函数的执行结果决定，有以下四种情况

1) 没有返回`return`语句，则相当于`result`只声明未赋值，`undefined`

```
// 1) 没有返回return语句，则相当于result只声明未赋值，undefined
console.log(result);
```



2)返回非Promise类型的值

```
// 2)返回非Promise类型的值  
return 'aaa'
```

```
▼ Promise {<pending>} ⓘ  
▶ __proto__: Promise  
  [[PromiseState]]: "fulfilled"  
  [[PromiseResult]]: "aaa"  
用户数据  
>
```

3)返回promise对象

```
// 3)返回promise对象  
return new Promise((resolve, reject) => {  
  resolve('ok')  
})
```

```
▼ Promise {<pending>} ⓘ  
▶ __proto__: Promise  
  [[PromiseState]]: "fulfilled"  
  [[PromiseResult]]: "ok"  
用户数据  
>
```

```
return new Promise((resolve, reject) => {  
  // resolve('ok')  
  reject('err')  
})
```

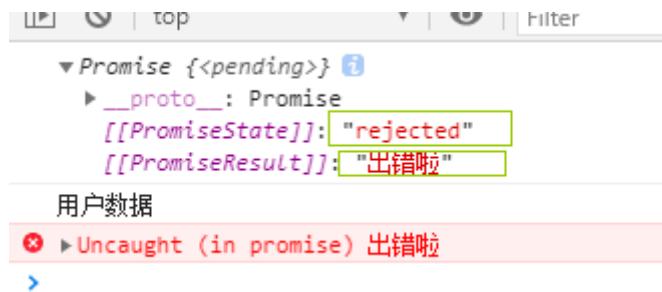
```
▼ Promise {<pending>} ⓘ  
▶ __proto__: Promise  
  [[PromiseState]]: "rejected"  
  [[PromiseResult]]: "err"  
用户数据  
✖ ▶ Uncaught (in promise) err  
>
```

4)抛出错误

```
// 4)抛出错误  
throw new Error('出错啦')
```

```
▼ Promise {<pending>} ⓘ  
▶ __proto__: Promise  
  [[PromiseState]]: "rejected"  
  [[PromiseResult]]: Error: 出错啦 at file:///C:/Users/Dai  
用户数据  
✖ ▶ Uncaught (in promise) Error: 出错啦  
  at 9-Promise.prototype.then().html:30  
>
```

```
throw '出错啦'
```



```
// 1) 2)两种情况下状态都是成功，返回值为对象成功的值
// 3)情况状态和返回值取决于promise对象内调用的函数
// 4)情况状态失败
let result = p.then(value => {
  console.log(value);
  // 2)返回非promise类型的值
  // return 'aaa'
  // 3)返回promise对象
  /* return new Promise((resolve, reject) => {
    // resolve('ok')
    reject('error')
  }) */
  // 4)抛出错误
  // throw new Error('出错啦')
  throw '出错啦'
}, reason => {
  console.error('出错了');
})
// 1)没有返回return语句，则相当于result只声明未赋值，undefined
console.log(result);
```

链式调用

```
// 2.链式调用
p.then(value => {

}).then(value => {

})
```

原理：第一个回调函数完成以后，会将**返回结果作为参数，传入第二个回调函数**。返回结果若是一个Promise对象，则等该Promise对象的状态发生变化时，才会被调用，然后根据状态调用相应的函数。

19.5Promise案例-多个文件内容读取

注意：

1.核心：**then方法返回Promise对象并链式调用**

2.使用**模板字符串**进行文档内容拼接

3.为了输出美观，**添加\r\n文档间的换行**

```
let fs = require('fs')
// 读取resources下的三个文件
// 回调地狱的做法，注意修改参数防止弄混
/* fs.readFile('resources/座右铭.txt', (err, data1) => {
```

```

        fs.readFile('resources/梦想.txt', (err, data2) => {
            fs.readFile('resources/目标.txt', (err, data3) => {
                console.log(data1 + '\n' + data2 + '\n' + data3);
            })
        })
    }
}

// Promise链式调用实现
let p = new Promise((resolve, reject) => {
    fs.readFile('resources/座右铭.txt', (err, data) => {
        resolve(data)
    })
})

p.then(value => {
    return new Promise((resolve, reject) => {
        fs.readFile('resources/梦想.txt', (err, data) => {
            // 使用模板字符串拼接
            resolve(` ${value}\r\n${data}`)
        })
    })
}).then(value => {
    return new Promise((resolve, reject) => {
        fs.readFile('resources/目标.txt', (err, data) => {
            resolve(` ${value}\r\n${data}`)
        })
    })
}).then(value => {
    console.log(value);
})
}

```

PS C:\Users\Daai\Desktop\前端学习案例\js-ES6-day02> node 10-Promise案例-多个文件内容读取.js
对于可控的事情，我们要保持谨慎；对于不可控的事情，我们要保持乐观
当一个纸灯，发光发亮
合理安排每一天

19.6Promise.prototype.catch()

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

`then()` 方法指定的回调函数，如果运行中抛出错误，也会被 `catch()` 方法捕获。

要点：

1. `reject()` 方法的作用，等同于抛出错误

```

const promise = new Promise(function(resolve, reject) {
    throw new Error('test');
});
promise.catch(function(error) {
    console.log(error);
});
// Error: test

```

2. 如果 `Promise` 状态已经变成 `resolved`，再抛出错误是无效的。因为 `Promise` 的状态一旦改变，就永远保持该状态，不会再变了。

```

const promise = new Promise(function(resolve, reject) {
  resolve('ok');
  throw new Error('test');
});
promise.then(
  function(value) {
    console.log(value)
  }
).catch(
  function(error) {
    console.log(error)
  }
);
// ok

```

3. 错误总是会被下一个 `catch` 语句捕获，‘冒泡’的性质，所以建议 **Promise 对象后面要跟 `catch()` 方法，这样可以处理/抓捕遗漏 `Promise` 内部发生的错误。**

```

getJSON('/post/1.json').then(function(post) {
  return getJSON(post.commentURL);
}).then(function(comments) {
  // some code
}).catch(function(error) {
  // 处理前面三个Promise产生的错误
});

```

4. 建议**总是使用 `catch()` 方法，而不使用 `then()` 方法的第二个参数。**

```

// bad
promise
  .then(function(data) {
    // success
  }, function(err) {
    // error
  });

// good
promise
  .then(function(data) { //cb
    // success
  })
  .catch(function(err) {
    // error
  });

```

5. **Promise 内部的错误不会影响到 Promise 外部的代码**，通俗的说法就是“Promise 会吃掉错误”，需要使用 `catch()` 抓捕错误。

```

const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

```

```
someAsyncThing().then(function() {
  console.log('everything is great');
});

setTimeout(() => { console.log(123) }, 2000);
// uncaught (in promise) ReferenceError: x is not defined
// 123
```

不会因报错而退出进程、终止脚本执行

6. `catch()` 方法返回的还是一个 Promise 对象，因此后面还可以接着调用 `then()` 方法，链式调用。

7. 先写 `catch()` 方法后写 `then()` 方法，如果没有报错，则会跳过 `catch()` 方法，要是 `then()` 方法里面有报错，就与前面的 `catch()` 无关了

```
Promise.resolve()
  .catch(function(error) {
    console.log('oh no', error);
  })
  .then(function() {
    console.log('carry on');
  });
// carry on
```

8. `catch()` 方法之中，还能再抛出错误，可以使用第二个 `catch()` 方法用来捕获前一个 `catch()` 方法抛出的错误

```
const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
  // 下面一行会报错，因为y没有声明
  y + 2;
}).catch(function(error) {
  console.log('carry on', error);
});
// oh no [ReferenceError: x is not defined]
// carry on [ReferenceError: y is not defined]
```

19.7 Promise.all()

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

`Promise.all()` 方法的参数可以不是数组，但必须具有 Iterator 接口，且返回的每个成员都是 Promise 实例。

语法：

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

p的状态由里面的参数决定，若p1, p2, p3状态都是 fulfilled， p才为 fulfilled；只要有一个状态 rejected，则p状态就为 rejected。（类似与逻辑）

19.8Promise.race()

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

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

Promise.race() 方法的参数与 Promise.all() 方法一样。

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

19.9Promise.allSettled()

Promise.allSettled() 参数是Promise对象，该方法返回的新的 Promise 实例

只有等到所有这些参数实例都返回结果，不管是 fulfilled 还是 rejected，包装实例才会结束。（注意与all方法区分）

```
const resolved = Promise.resolve(42);
const rejected = Promise.reject(-1);

const allSettledPromise = Promise.allSettled([resolved, rejected]);

allSettledPromise.then(function (results) {
  console.log(results);
});

// [
//   { status: 'fulfilled', value: 42 },
//   { status: 'rejected', reason: -1 }
// ]
```

Promise.allSettled() 方法可以确保所有操作都结束，可以利用该方法过滤出成功的请求，而 Promise.all() 无法确定所有请求都结束

```
const promises = [ fetch('index.html'), fetch('https://does-not-exist/') ];
const results = await Promise.allSettled(promises);

// 过滤出成功的请求
const successfulPromises = results.filter(p => p.status === 'fulfilled');

// 过滤出失败的请求，并输出原因
const errors = results
  .filter(p => p.status === 'rejected')
  .map(p => p.reason);
```

20.数值的扩展

20.1二进制和八进制表示法

ES6 提供了二进制和八进制数值的新的写法。

二进制前缀: `0b` (或 `0B`)

八进制前缀: `0o` (或 `0O`) 表示。

```
// 1.二进制和八进制
let b = 0b0101 //5
let o = 0o42 //34
let d = 100 //100
let x = 0xa3 //163 十六进制
console.log(b, o, d, x);
```

如果要将 `0b` 和 `0o` 前缀的字符串数值转为十进制, 要使用 `Number` 方法。

```
Number('0b111') // 7
Number('0o10') // 8
```

20.2 Number.isFinite(), Number.isNaN()

小结:

1.参数类型不是数值, `Number.isFinite`一律返回`false`

2.参数类型不是`NaN`, `Number.isNaN`一律返回`false`

```
// 2.Number.isFinite() 检查一个数值是否为有限的
console.log(Number.isFinite(5)); //true
console.log(Number.isFinite(0.3)); //true
console.log(Number.isFinite(Infinity)); //false
console.log(Number.isFinite('Infinity')); //false
// 参数类型不是数值, Number.isFinite一律返回false
// Number.isNaN() 检查一个值是否为NaN
console.log(Number.isNaN(NaN)); //true
console.log(Number.isNaN(3)); //false
// 参数类型不是NaN, Number.isNaN一律返回false
```

与传统的全局方法 `isFinite()` 和 `isNaN()` 的区别在于, 传统方法先调用 `Number()` 将非数值的值转为数值, 再进行判断, 而这两个新方法只对数值有效, `Number.isFinite()` 对于非数值一律返回 `false`, `Number.isNaN()` 只有对于 `NaN` 才返回 `true`, 非 `NaN` 一律返回 `false`。

20.3 Number.parseInt(), Number.parseFloat()

ES6 将全局方法 `parseInt()` 和 `parseFloat()`, 移植到 `Number` 对象上面, 行为完全保持不变。

可以除去后面的字符串

```
// 3.Number.parseInt(), Number.parseFloat() 字符串转为整数
console.log(Number.parseInt('233haha')); //233
console.log(Number.parseFloat('3.1415aaa')); //3.1415
```

20.4 Number.isInteger()

`Number.isInteger()` 用来判断一个数值是否为整数。

```
// 4.Number.isInteger() 判断一个数值是否为整数
console.log(Number.isInteger(3)); //true
console.log(Number.isInteger(3.0)); //true
console.log(Number.isInteger(3.3)); //false
```

注意：

1. 整数和浮点数采用的是同样的储存方法，所以 3 和 3.0 被视为同一个值，都为 true。
2. 由于 JavaScript 采用 IEEE 754 标准，数值存储为 64 位双精度格式，数值精度最多可以达到 53 个二进制位（1 个隐藏位与 52 个有效位）。如果数值的精度超过这个限度，第 54 位及后面的位就会被丢弃，这种情况下，Number.isInteger 可能会误判。

```
Number.isInteger(3.0000000000000002) // true
```

原因：这个小数的精度达到了小数点后 16 个十进制位，转成二进制位 ($16 \times 4 = 54$) 超过了 53 个二进制位，第 54 位及后面的位就会被丢弃，最后的那个 2 被丢弃了。

3. 如果一个数值的绝对值小于 Number.MIN_VALUE（没有等于， 5×10^{-324} ），即小于 JavaScript 能够分辨的最小值，会被自动转为 0。

```
Number.isInteger(5E-324) // false
Number.isInteger(5E-325) // true
```

小结：如果参数不是数值，Number.isInteger 返回 false。

如果对数据精度的要求较高，不建议使用 Number.isInteger() 判断一个数值是否为整数。

20.5 Number.EPSILON

Number.EPSILON 是 JavaScript 表示的最小精度，表示 1 与大于 1 的最小浮点数之间的差。

引入目的：因为浮点数计算是不精确的，在于为浮点数计算，设置一个误差范围

Number.EPSILON 用来设置“能够接受的误差范围”，即如果两个浮点数的差小于这个值，就认为这两个浮点数相等。

```
// 5.Number.EPSILON 是 JavaScript 表示的最小精度
console.log(0.1 + 0.2); // 0.30000000000000004
// 如果两个浮点数的差小于这个值，则认为这两个浮点数相等。
function equal(a, b) {
    if (Math.abs(a - b) < Number.EPSILON) {
        return true
    } else {
        return false
    }
}
console.log(equal(0.1 + 0.2, 0.3)); // true
```

20.6 安全整数和 Number.isSafeInteger()

JavaScript 能够准确表示的整数范围在 -2^{53} 到 2^{53} 之间（不含两个端点），超过这个范围，无法精确表示这个值。

ES6 引入了 `Number.MAX_SAFE_INTEGER` 和 `Number.MIN_SAFE_INTEGER` 这两个常量，用来表示这个范围的上下限。

```
console.log(Math.pow(2, 53)); //9007199254740992
console.log(Math.pow(2, 53) + 1); //9007199254740992
console.log(Number.MAX_SAFE_INTEGER === Math.pow(2, 53) - 1); //true
console.log(Number.MAX_SAFE_INTEGER === -Number.MIN_SAFE_INTEGER); //true
```

`Number.isSafeInteger()` 则是用来判断一个整数是否落在这个范围之内

注意：验证运算结果是否落在安全整数的范围内，不要只验证运算结果，而要同时验证参与运算的每个值。

```
console.log(Number.isSafeInteger(99999999 - 100)); //true
console.log(Number.isSafeInteger(3333)); //true
console.log(Number.isSafeInteger(Math.pow(2, 53))); //false
```

20.7 Math 对象的扩展

Math.trunc()

`Math.trunc` 方法用于去除一个数的小数部分，返回整数部分。

```
console.log(Math.trunc(3.3)); //3
console.log(Math.trunc(3.7)); //3
```

对于非数值，`Math.trunc` 内部使用 `Number` 方法将其先转为数值

```
console.log(Math.trunc('23.1')); //23
```

对于空值和无法截取整数的值，返回 `Nan`。

```
Math.trunc(NaN); // NaN
Math.trunc('foo'); // NaN
Math.trunc(); // NaN
Math.trunc(undefined) // NaN
```

Math.sign()

`Math.sign` 方法用来判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。

会返回五个值：

- 参数为正数，返回 1；
- 参数为负数，返回 -1；
- 参数为 0，返回 0；
- 参数为 -0，返回 -0；
- 其他值，返回 `NaN`。

```
console.log(Math.sign(2)); //1
console.log(Math.sign(-2));//-1
console.log(Math.sign(0));//0
console.log(Math.sign(-0));//0
console.log(Math.sign(NaN));//NaN
console.log(Math.sign('2'));//1
```

20.8指数运算

指数运算符 (`**`) 用来实现幂运算，与 `Math.pow` 结果功能一样。, **特点是右结合，从最右边开始计算的**

```
// 指数运算 **
console.log(2 ** 10); //等同于Math.pow(2,10) //1024
console.log(2 ** 3 ** 2); //相当于 2 ** (3 ** 2) //512
```

指数运算符可以与等号结合，形成一个新的赋值运算符 (`**=`)

```
let a = 1.5;
a **= 2;
// 等同于 a = a * a;而不是2的a次。
```

20.9 BigInt

20.9.1 BigInt数据类型

新的数据类型 **BigInt (大整数)** , **BigInt 只用来表示整数** (浮点型不可以) , 没有位数的限制，任何位数的整数都可以精确表示，可以表示大于2的1024次方的数值。

注意点:

1.为了与 `Number` 类型区别，**BigInt 类型的数据必须添加后缀 `n`**，数据类型为 `BigInt`

2.`BigInt` 的运算，只能是 `BigInt` 类型之间的运算

```
1234 // 普通整数
1234n // BigInt
typeof 1234n // 'bigint'

// BigInt 的运算，只能是BigInt类型之间的运算
1n + 2n // 3n
```

3.`BigInt` 可以使用负号 (`-`) , 但是不能使用正号 (`+`)

```
-42n // 正确
+42n // 报错
```

4.`BigInt` 与普通整数是两种值，它们之间并不相等

```
42n === 42 // false
```

20.9.2 BigInt 对象

`BigInt()` 构造函数生成 BigInt 类型的数值，必须要有参数，且参数必须可以正常转为数值，不能是小数

```
BigInt(123) // 123n  
BigInt('123') // 123n  
BigInt(false) // 0n  
BigInt(true) // 1n
```

21.模块化Module

21.1概述

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。 (感觉类似JAVA里的 package, Python里的import)

- 历史上, JavaScript 一直没有模块 (module) 体系。

ES6 之前, 社区制定了一些模块加载方案:

- 1) CommonJS => NodeJS. Browserify 用于服务器
- 2) AMD=> requireJS 用于浏览器
- 3) CMD=> seaJS

- 模块化的好处有以下几点:

- 1)防止命名冲突
- 2)代码复用
- 3)高维护性

21.2ES6模块化语法

ES6 的模块自动采用严格模式。

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

21.2.1export命令

使用 `export` 命令定义了模块的对外接口

写法

写法1

```
//分别暴露数据 m1.js  
export let firstname = 'Michael'  
export let lastName = 'Jackson'  
export let year = 1989
```

写法2 注意{}
写法3

```
// 统一暴露 m2.js
let firstname = 'Michael'
let lastName = 'Jackson'
let year = 1999

export { firstname, lastName, year }
```

写法3：`export default` 命令，为模块指定默认输出。

```
// 默认暴露 m3.js
export default {
  name: '刘德华',
  sex: '男'
}
```

一个模块只能有一个默认输出，因此`export default`命令只能使用一次。

本质上，`export default`就是输出一个叫做`default`的变量或方法，然后系统允许你为它取任意名字，所以它后面不能跟变量声明语句。

```
// 错误
export default var a = 1;
// 正确
var a = 1;
export default a;
```

注意点：

1. `export`可以输出各种类型的数据，引入文件名中的'.js'可以不写，当前路径'./'必须要写。

2. `export`可以使用`as关键字重命名`

```
function v1() { ... }
function v2() { ... }

export {
  v1 as streamV1,
  v2 as streamV2,
  v2 as streamLatestVersion
};
```

重命名后，`v2`可以用不同的名字输出两次。

3. `export`命令规定的是对外的接口，必须与模块内部的变量建立一一对应关系。要加{}
错误的写法

```
// 报错
export 1;

// 报错
var m = 1;
export m;

// 报错
function f() {}
export f;
```

正确的写法

```
// 写法一
export var m = 1;

// 写法二
var m = 1;
export {m};

// 写法三
var n = 1;
export {n as m};

//
// 正确
export function f() {};

// 正确
function f() {}
export {f};
```

4. `export` 语句输出的接口，与其对应的值是**动态绑定**关系，即通过该接口，可以取到模块内部实时的值。

5. `export` 命令可以出现在模块的任何位置，只要**处于模块顶层**就可以。如果处于块级作用域内，就会报错。因为处于条件代码块之中，就没法做静态优化了，违背了 ES6 模块的设计初衷。

```
function foo() {
  export default 'bar' // SyntaxError
}
foo()
// export语句放在函数之中，结果报错
```

21.2.2 import 命令

使用 `import` 命令加载模块。

注意点：

1. `import` 命令输入的**变量都是只读的**，因为它的本质是输入接口。也就是说，**不允许在加载模块的脚本里面，改写接口**。如果接口是一个对象，可以改变其属性，但不建议那么做，很难查错。
2. `import` 命令具有**提升效果**，会提升到整个模块的头部，首先执行
3. 多次重复执行同一句 `import` 语句，结果只会执行一次，而不会执行多次

4.在script标签内引入，需要设置类型为module <script type="module"></script>。

5.测试时使用live server插件打开浏览器，默认方式打开会报错。

写法：

写法1

```
// 1.解构赋值形式
import { firstName, lastName, year } from "./js/m1.js"
console.log(firstName, lastName, year);
// 有重名的变量需要使用as关键字进行重命名，否则会报错
import { firstName as first, lastName as last, year as y } from
"./js/m2.js"
console.log(first, last, y);
// 默认暴露数据的必须进行重命名
import { default as m3 } from "./js/m3.js"
console.log(m3);
```

注意点：

1.有重名的变量需要使用as关键字进行重命名，否则会报错

2.默认暴露数据的必须进行重命名，否则会报错



写法2

```
// 2.整体加载 用星号(*)指定一个对象，所有输出值都加载在这个对象上面，全部加载
// 引入m1.js模块内容
import * as m1 from "./js/m1.js"
console.log(m1);
// 引入m2.js模块内容
import * as m2 from "./js/m2.js"
console.log(m2);
// 引入m3.js模块内容
import * as m3 from "./js/m3.js"
console.log(m3);
console.log(m3.default.name);
```

```
▶ Module {Symbol(Symbol.toStringTag): "Module"}
▶ Module {Symbol(Symbol.toStringTag): "Module"}
▶ Module {Symbol(Symbol.toStringTag): "Module"}
刘德华
```

写法3，针对默认暴露，注意没有{}

```
// 3.简便形式 针对默认暴露 m3是自定义的名字
import m3 from "./js/m3.js"
console.log(m3);
```

```
▶ {name: "刘德华", sex: "男"}
```

显然，一个模块只能有一个默认输出，因此 `export default` 命令只能使用一次。所以，`import` 命令后面才不用加大括号，因为只可能唯一对应 `export default` 命令。

21.2.3 模块引入方法

1. 在 `<script type="module"></script>` 内引入需要的模块

缺点：若引入模块数量多，则代码会很长。

```
<script type="module">
    // 1.解构赋值形式
    /* import { firstName, lastName, year } from "./js/m1.js"
    console.log(firstName, lastName, year);
    // 有重名的变量需要使用as关键字进行重命名，否则会报错
    import { firstName as first, lastName as last, year as y } from
    "./js/m2.js"
    console.log(first, last, y);
    // 默认暴露数据的必须进行重命名
    import { default as m3 } from "./js/m3.js"
    console.log(m3); */

    // 2.整体加载 用星号(*) 指定一个对象，所有输出值都加载在这个对象上面，全部加载
    /* // 引入m1.js模块内容
    import * as m1 from "./js/m1.js"
    console.log(m1);
    // 引入m2.js模块内容
    import * as m2 from "./js/m2.js"
    console.log(m2);
    // 引入m3.js模块内容
    import * as m3 from "./js/m3.js"
    console.log(m3);
    console.log(m3.default.name); */
    // 3.简便形式 针对默认暴露 m3是自定义的名字
    import m3 from "./js/m3.js"
    console.log(m3);
</script>
```

2. 单独建立一个 `app.js` 入口文件，里面引入模块。这样仅需引入 `app.js` 文件即可。模块的引入方式和之前一样。

注意模块引用路径，和类型设置为模块

```
// 入口文件 app.js

// 模块引入
import * as m1 from "./m1.js"
// 引入m2.js模块内容
import * as m2 from "./m2.js"
// 引入m3.js模块内容
import * as m3 from "./m3.js"

console.log(m1);
console.log(m2);
console.log(m3);
```

```
<script src="./js/app.js" type="module"></script>
```

21.3 babel对IES6模块化代码转换

1.安装工具babel-cli babel-preset-env browserify(webpack)

browserify是打包工具，后期会使用webpack打包，目前先用这个。

```
PS C:\Users\Daai\Desktop\前端学习案例\js-ES6-day04> npm i browserify -D
```

2. npx babel js -d dist/js 和之前build命令功能一样，先用babel将js代码转换

3. 打包 npx browserify dist/app.js -o dist/bundle.js 将转换后的js代码进行打包

bundle意为一捆，取改名表示打包后的文件

```
JS-ES...
└── dest
    ├── app.js
    ├── m1.js
    ├── m2.js
    └── m3.js
    └── dist
        └── bundle.js
```

转换后的文件及打包后的文件。

注意：修改原始文件内app.js的代码并不会改变打包后的文件内bundle.js的代码，需要再次babel转换，再次打包。

如此，引入打包后的文件即可使用

```
<script src="./dist/bundle.js"></script>
```

```
▼ Object ⓘ
  firstName: "Michael"
  lastName: "Jackson"
  year: 1989
  __esModule: true
  ► __proto__: Object

▼ Object ⓘ
  firstName: "Michael"
  lastName: "Jackson"
  year: 1999
  __esModule: true
  ► __proto__: Object

▼ Object ⓘ
  ► default: {name: "刘德华", sex: "男"}
  __esModule: true
  ► __proto__: Object
```

21.4ES6模块化引入NPM包

要求：背景颜色变粉色。

首先要下载npm

1.先在终端输入命令 `npm i jquery` 安装jQuery

```
PS C:\Users\Daai\Desktop\前端学习案例\js-ES6-day04> npm i jquery
added 1 package in 4s
```

2.导入jquery并使用

```
// 修改背景颜色
// 导入jquery $是变量名 'jquery'是模块的名字
import $ from 'jquery';
$('body').css('background', 'pink')
```

3.再次编译，再次打包

```
PS C:\Users\Daai\Desktop\前端学习案例\js-ES6-day04> npm run build
> js-ES6-day04@1.0.0 build
> babel js -d dest

Successfully compiled 4 files with Babel (1269ms).
PS C:\Users\Daai\Desktop\前端学习案例\js-ES6-day04> npx browserify dest/app.js -o dist/bundle.js
PS C:\Users\Daai\Desktop\前端学习案例\js-ES6-day04> 
```



21.5动态import()

`import()` 函数，支持动态加载模块（需要时才加载，懒加载）。

`import()` 返回一个 **Promise** 对象，可以使用 `then` 方法进行调用里面暴露的数据

```
import(specifier) //参数specifier, 指定所要加载的模块的位置
```

示例：

```
import('./hello.js').then(module => { //module参数即暴露的数据
    module.hello();
});

// hello.js
export function hello(){
    alert('Hello');
}
```

22.async 函数

`async` 和 `await` 两种语法结合可以让异步代码像同步代码一样

`async` 函数是 Generator 函数的语法糖（一种便捷写法，对原有功能没有影响）

22.1 基本用法

22.1.1 async 函数

1. `async` 函数返回一个 **Promise** 对象，对象的状态和 `then()` 方法规则一模一样。

返回的状态：

- 1) 返回的结果不是一个 `Promise` 类型的对象，返回的状态都是成功的 `fulfilled`
- 2) 抛出错误，返回状态是一个失败的 `Promise`
- 3) 返回一个 `Promise` 对象，则 `Promise` 对象的状态由 `async` 函数执行的返回值决定

```
// async函数 声明
async function fn() {
    // 1. 返回的结果不是一个Promise类型的对象，返回的状态都是成功的fulfilled
    // return 'hello' //返回一个字符串
    // return undefined
    // 2. 抛出错误，返回状态是一个失败的Promise
    // throw new Error('出错啦')
    // 3. 返回一个Promise对象，则Promise对象的状态由async函数执行的返回值决定
    return new Promise((resolve, reject) => {
        // resolve('成功的数据')
        reject('失败的数据')
    })
}
// 调用 和普通函数一样
let res = fn()
console.log(res);
```

返回一个字符串

```

    ▼ Promise ⓘ
      ► __proto__: Promise
      [PromiseState]: "fulfilled"
      [PromiseResult]: "hello"
  
```

未设置返回值

```

    ▼ Promise {<fulfilled>: undefined} ⓘ
      ► __proto__: Promise
      [PromiseState]: "fulfilled"
      [PromiseResult]: undefined
  
```

抛出一个错误

```

    ▼ Promise {<rejected>: Error: 出错啦}
      at fn (file:///C:/Users/Daii/Desktop/%E5%89%8D%1
      ► __proto__: Promise
      [PromiseState]: "rejected"
      [PromiseResult]: Error: 出错啦 at fn (file:///C:...
  
```

捕获并处理

```

    ▼ Promise {<rejection>: Uncaught (in promise) Error: 出错啦}
      at fn (10-async函数基本用法.html:19)
      at 10-async函数基本用法.html:22
  
```

返回Promise对象

```

    ▼ Promise {<pending>} ⓘ
      ► __proto__: Promise
      [PromiseState]: "fulfilled"
      [PromiseResult]: "成功的数据"
  
```

失败的Promise对象

```

    ▼ Promise {<pending>} ⓘ
      ► __proto__: Promise
      [PromiseState]: "rejected"
      [PromiseResult]: "失败的数据"
  
```

捕获并处理

```

    ⚠ Uncaught (in promise) 失败的数据
  
```

2. 可以使用 `then` 方法添加回调函数。

```
res.then(value => {
  console.log(value);
})
```

成功的数据

22.1.2 await 表达式

1. `await` 必须写在 `async` 函数中

2. `await` 右侧的表达式一般为 `promise` 对象

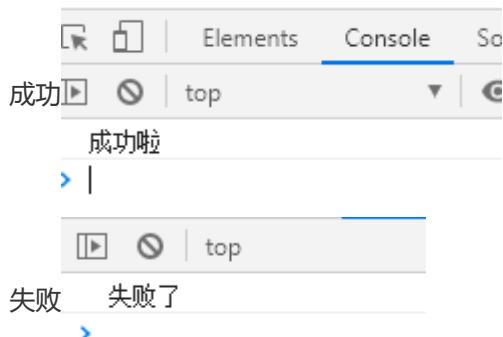
3. `await` 返回的是 `promise` 成功的值

4. `await` 的 `promise` 失败了，就会抛出异常，需要通过 `try...catch` 捕获。所以 `await` 经常与 `try...catch` 配合使用。

```

let p = new Promise((resolve, reject) => {
    // resolve('成功啦')
    reject('失败了')
})
async function main() {
    try {
        let result = await p
        console.log(result);
    } catch (e) {
        console.log(e);
    }
}
main()

```



22.1.3案例-读取文件

3.当函数执行的时候，一旦遇到 `await` 就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。

要点：使用函数封装读取文件，返回一个Promise对象，用async函数里的await表达式接收

```

// 引入fs模块
let fs = require('fs')

// 读取文件函数封装
function dream() {
    return new Promise((reslove, reject) => {
        fs.readFile("./resources/梦想.txt", (err, data) => {
            // 如果失败
            if (err) reject(err)
            // 如果成功
            reslove(data)
        })
    })
}

function goal() {
    return new Promise((reslove, reject) => {
        fs.readFile("./resources/目标.txt", (err, data) => {
            // 如果失败
            if (err) reject(err)
            // 如果成功
            reslove(data)
        })
    })
}

```

```

}

function ming() {
    return new Promise((reslove, reject) => {
        fs.readFile("./resources/座右铭.txt", (err, data) => {
            // 如果失败
            if (err) reject(err)
            // 如果成功
            reslove(data)
        })
    })
}

// 定义一个async函数
async function main() {
    try {
        let res1 = await dream()
        let res2 = await goal()
        let res3 = await ming()
        console.log(res1.toString());
        console.log(res2.toString());
        console.log(res3.toString());
    } catch (e) {
        console.log(e);
    }
}
main()

```

```

PS C:\Users\Daiei\Desktop\前端学习案例\js-ES6-day03> node 12-async和await读取文件.js
当一个纸灯，发光发亮
合理安排每一天
对于可控的事情，我们要保持谨慎；对于不可控的事情，我们要保持乐观
PS C:\Users\Daiei\Desktop\前端学习案例\js-ES6-day03>

```

22.1.4案例-发送AJAX请求

要点：

封装发送AJAX请求函数，返回结果是Promise对象，将AJAX四个基本操作都放入Promise对象内

```

// 封装发送AJAX请求函数，返回结果是Promise对象
function send(url) {
    return new Promise((resolve, reject) => {
        let xhr = new XMLHttpRequest()
        xhr.open('GET', url)
        xhr.send()
        xhr.onreadystatechange = function () {
            if (xhr.readyState == 4) {
                if (xhr.status >= 200 & xhr.status < 300) {
                    resolve(xhr.response)
                } else {
                    reject(xhr.response)
                }
            }
        }
    })
}

```

```
// promise then方法测试
/*  send('https://api.apiopen.top/getJoke').then(value => {
    console.log(value);
}) */
// async与await测试
async function main() {
    // 发送AJAX请求
    let res = await send('https://api.apiopen.top/getJoke')
    console.log(res);
}
main()
```

The screenshot shows a browser's developer tools Network tab with a single request listed. The URL is 'https://api.apiopen.top/getJoke'. The response status is 200, the message is '成功!', and the content type is 'application/json'. The response body is a JSON object containing various fields like code, message, result, etc., which describe a joke.

```
{"code":200,"message":"成功!","result":[{"sid":"29775505","text":"男朋友送礼物还带小票是什么心理?","type":"image","thumbnail":null,"video":null,"images":"http://wimg.spriteapp.cn/ugc/2019/09/16/5d1d899197fc3_mini.jpg","top_cc_dro", "top_comments_voiceuri": "", "top_comments_uid": "23130256", "top_comments_name": "强强很开心", "top_comments_header": "http://wimg.spriteapp.cn/profile/large/2019/07/04/5d1d89d602024_mini.jpg", "pi~", "type": "gif", "thumbnail": "http://wimg.spriteapp.cn/ugc/2018/12/06/5c093cde7303f_a_1.jpg", "video": null, "forward": "1", "comment": "5", "uid": "19889419", "name": "天天趣图 [天天趣图]", "header": "http://wimg.spriteapp.cn/profile/large/2018/07/31/5b60500856689_mini.jpg", "top_comments_op_comments_header": null, "passtime": "2018-12-08 02:48:02"}, {"sid": "29689612", "text": "这下商家亏成\u0d83d\\n"}]}
```

23.正则的扩展

23.1 RegExp 构造函数

ES6如果 `RegExp` 构造函数第一个参数是一个正则对象，那么可以使用第二个参数指定修饰符。而且，返回的正则表达式会忽略原有的正则表达式的修饰符，只使用新指定的修饰符。

```
new RegExp(/abc/ig, 'i')  
// 使用第二个参数指定的修饰符--i
```

23.2 正则表达式的特殊字符

23.2.1 字符型

符号	说明	举例
.	除了换行符以外的任意单个字符	
\d	相当于 [0-9]	
\D	相当于 [^0-9]	
\w	匹配一个单字字符，相当于 [A-Za-z0-9_]，注意不包括横线-	
\W	匹配一个非单字字符，相当于 [^A-Za-z0-9_]	
\s	匹配一个空白字符，包括空格、制表符、换页符和换行符	/\s\w*/ 匹配"foo bar."中的'bar'
\S	匹配一个非空白字符	/\s\w*/ 匹配"foo bar."中的'foo'
\t	匹配一个水平制表符	
\r	匹配一个回车符	
\n	匹配一个换行符	

23.2.2量词

默认情况下，像 `*` (`{0,}`) 和 `+` (`{1,}`) 这样的量词是“贪婪的”，这意味着它们试图匹配尽可能多的字符串。`?` (`{0,1}`) 量词后面的字符使量词“非贪婪”：意思是它一旦找到匹配就会停止，匹配字符串尽可能的短。

给定一个字符串 `some <foo> <bar> new </bar> </foo> thing`：

`.` 和 `*` 结合使用：**贪婪匹配，在使整个表达式能得到匹配的前提下匹配尽可能多的字符。** 下面两个示例的比较：

```
/<.*>/ will match "<foo> <bar> new </bar> </foo>"
```

`.` 和 `*` 和 `?` 结合使用：**.*? 表示匹配任意数量的重复，但是在能使整个匹配成功的前提下使用最少的重复。**

```
/<.*?>/ will match "<foo>"
```

23.3正则表达式方法

23.3.1 RegExp 对象方法

方法	描述
exec	检索字符串中的正则表达式的匹配，返回一个数组，存放匹配的结果。未匹配到则返回 null。
test	检索字符串中指定的值。返回 true 或 false

示例：

```
// exec方法 返回一个数组（未匹配到则返回 null）
let reg = /<.*?>/g
let arr = reg.exec('<foo> <bar> new')
console.log(arr);

▼ [ "<foo>", index: 0, input: "<foo> <bar> new", groups: undefined ] ⓘ
  0: "<foo>" ⓘ
  groups: undefined ⓘ
  index: 0
  input: "<foo> <bar> new"
  length: 1
  ► __proto__: Array(0)
```

```
console.log(reg.test('<foo> <bar> new')); //true
```

23.3.2 支持正则表达式的String对象的方法

ES6 将这 4 个方法，在语言内部全部调用 RegExp 的实例方法，从而做到所有与正则相关的方法，**全都定义在 RegExp 对象上。**（不知道怎么通过 RegExp 对象调用这四个方法？）

- `String.prototype.match` 调用 `RegExp.prototype[Symbol.match]`
- `String.prototype.replace` 调用 `RegExp.prototype[Symbol.replace]`
- `String.prototype.search` 调用 `RegExp.prototype[Symbol.search]`
- `String.prototype.split` 调用 `RegExp.prototype[Symbol.split]`

方法	描述
search	检索与正则表达式相匹配的值，返回匹配到的位置索引，在失败时返回-1
match	找到一个或多个正则表达式的匹配，返回一个数组，在未匹配到时会返回 null
replace	替换与正则表达式匹配的子串，返回一个新字符串
split	把字符串分割为字符串数组

示例：

```
// 支持正则的String对象方法
// 1. search方法 检索与正则表达式相匹配的值，只能匹配最近的一个
let str = 'new<foo> <bar> '
console.log(str.search(/<.*?>/g)); //3
// 2. match方法 找到一个或多个正则表达式的匹配，返回一个数组
console.log(str.match(/<.*?>/g)); //["<foo>", "<bar>"]
// 3. replace方法 替换与正则表达式匹配的子串，返回一个新字符串
console.log(str.replace(/o/g, '哦哦')); //new<f哦哦> <bar>
// 4. split方法 把字符串分割为字符串数组
let s = 'How are you doing today?'
console.log(s.split("")); //H,o,w, ,a,r,e, ,y,o,u, ,d,o,i,n,g,
,t,o,d,a,y,?
console.log(s.split(" ")); //以空格分隔， ["How", "are", "you", "doing",
,"today?"]
console.log(s.split(/o/)); //以字母o分隔， ["H", "w are y", "u d", "ing t",
,"day?"]
```

23.3.3方法小结

想要知道一个字符串中的一个**匹配**是否被找到，使用 `test` 或 `search` 方法

想得到更多的信息（但是比较慢）则使用 `exec` 或 `match` 方法

23.4具名组匹配

组匹配

正则表达式使用**圆括号**进行组匹配

示例：使用圆括号进行组匹配从标准模式中**提取年月日**

```
const RE_DATE = /(\d{4})-(\d{2})-(\d{2})/;

const matchObj = RE_DATE.exec('1999-12-31');
const year = matchObj[1]; // 1999
const month = matchObj[2]; // 12
const day = matchObj[3]; // 31
console.log(matchObj);
```

```
▼ (4) ["1999-12-31", "1999", "12", "31"]
  0: "1999-12-31"
  1: "1999"
  2: "12"
  3: "31"
  groups: undefined
  index: 0
  input: "1999-12-31"
  length: 4
  ▶ __proto__: Array(0)
```

缺点：每一组的匹配含义可读性差，而且只能用数字序号（比如 `matchObj[1]`）引用，要是组的顺序变了，引用的时候就必须修改序号。

具名组匹配

ES8中提出**具名组匹配**：圆括号+“问号 + 尖括号 + 组名”（`?<year>`），然后就可以在 `exec` 方法返回结果的 `groups` 属性上引用该组名 `.groups.year`。

特点：具名组匹配等于为每一组匹配加上了 ID，把数据放入**变量名**中，操作更方便，即使组顺序乱了也不影响结果。

如果具名组没有匹配，那么对应的 `groups` 对象属性会是 `undefined`。

示例： w765

```
const RE_DATE = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/;

const matchObj = RE_DATE.exec('1999-12-31');
const year = matchObj.groups.year; // "1999"
const month = matchObj.groups.month; // "12"
const day = matchObj.groups.day; // "31"
console.log(matchObj);
```

```
▼ (4) [ "1999-12-31", "1999", "12", "31", index: 0, input: "1999-12-31", groups: {...} ] ⓘ
  0: "1999-12-31"
  1: "1999"
  2: "12"
  3: "31"
  ▼ groups:
    day: "31"
    month: "12"
    year: "1999"
  index: 0
  input: "1999-12-31"
  length: 4
  ► __proto__: Array(0)
```

案例：提取url和标签文本

别忘记链接a <\\> 添加转义字符 \\

方法1：组匹配 正则 reg = /(.*)/

```
// 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
// 方法1
let reg = /<a href="(.*)">(.*)</a>/
let res = reg.exec(str)
console.log(res);
console.log(res[1]);
console.log(res[2]);
```

```
▼ (3) [ "<a href='http://www.atguigu.com'>尚硅谷</a>", "http://www.atguigu.com", "尚硅谷"
  0: "<a href='http://www.atguigu.com'>尚硅谷</a>"
  1: "http://www.atguigu.com"
  2: "尚硅谷"
  groups: undefined
  index: 0
  input: "<a href='http://www.atguigu.com'>尚硅谷</a>"
  length: 3
  ► __proto__: Array(0)
http://www.atguigu.com
尚硅谷
```

方法2：具名组匹配 正则 /<a href="(?<url>.*)">(?:<text>.*)/

```
// 方法2 比方法1更好，可以使用变量名进行操作更方便
let reg = /<a href="(?<url>.*)">(?:<text>.*)</a>/
let res = reg.exec(str)
console.log(res);
console.log(res.groups.url);
console.log(res.groups.text);
```

```
▼ (3) [ "<a href='http://www.atguigu.com'>尚硅谷</a>", "http://www.atguigu.com", "<a href='http://www.atguigu.com'>尚硅谷</a>" ]
  0: "<a href='http://www.atguigu.com'>尚硅谷</a>"
  1: "http://www.atguigu.com"
  2: "尚硅谷"
  ▶ groups:
    text: "尚硅谷"
    url: "http://www.atguigu.com"
  index: 0
  input: "<a href='http://www.atguigu.com'>尚硅谷</a>"
  length: 3
  ▶ __proto__: Array(0)
  http://www.atguigu.com
  尚硅谷
```

23.5 u修饰符

u修饰符，含义为“**Unicode 模式**”，用来正确处理大于0xFFFF的 Unicode 字符。也就是说，会**正确处理四个字节的 UTF-16 编码**。

```
// u修饰符 表示unicode编码
console.log(/^\uD83D/u.test('\uD83D\uDC2A')); //false
console.log(/^\uD83D/.test('\uD83D\uDC2A')); //true
```

上面代码中，`\uD83D\uDC2A`是一个**四个字节的 UTF-16 编码，代表一个字符**。但是，ES5 不支持四个字节的 UTF-16 编码，会将其识别为两个字符，导致第二行代码结果为 `true`。加了 `u` 修饰符以后，ES6 就会识别其为一个字符，所以第一行代码结果为 `false`。

23.5.1点字符

点（.）字符：除了换行符以外的任意单个字符。

对于码点大于0xFFFF的 Unicode 字符，点字符不能识别，必须加上 `u` 修饰符。（那就一般使用都加`u`修饰符）

```
// 1)点字符 表示空格外的任意单个字符
let a = '吉'
let b = '吉'
console.log(/^\$/u.test(a)); //false
console.log(/^\$/u.test(b)); //true
console.log(/^\$/u.test(a)); //true
```

23.5.2Unicode 字符

使用**大括号表示 Unicode 字符**，这种表示法在正则表达式中**必须加上 u 修饰符**，才能识别当中的大括号，否则会被解读为量词。

```
// 2)使用大括号表示 Unicode 字符，必须加上u修饰符
console.log(/\u{61}/u.test('a')); //false
console.log(/\u{61}/u.test('a')); //true
console.log(/\u{20BB7}/u.test('吉')) //true
```

23.5.3量词

使用u修饰符后，所有量词都会正确识别码点大于0xFFFF的Unicode字符。

```
/a{2}/.test('aa') // true  
/a{2}/u.test('aa') // true  
/𠮷{2}/.test('𠮷𠮷') // false  
/𠮷{2}/u.test('𠮷𠮷') // true
```

23.5.4预定义模式

只有加了u修饰符，才能正确匹配码点大于0xFFFF的Unicode字符。

```
/^\s/.test('𠮷') // false  
/^\\s$/u.test('𠮷') // true
```

\s 是预定义模式，匹配所有非空白字符

应用：正确返回字符串长度的函数

23.6后行断言

先行断言（正向断言）：根据后面单个字符匹配后面字符，只能是后面规定的那个字符

示例：前面的字符(?=后面的字符)

```
x只有在y前面才匹配  
/x(?=y)/  
只匹配百分号之前的数字  
/\d+(?=%)/  
/\d+(?=%)/.exec('100% of US presidents have been male') // ["100"]
```

先行否定断言：根据后面单个字符匹配后面字符，除后面规定的那个字符都可以

示例：前面的字符(?!后面的字符)

```
x只有不在y前面才匹配  
/x(?!y)/  
只匹配不在百分号之前的数字  
/\d+(?!%)/  
/\d+(?!%)/.exec('that's all 44 of them')
```

括号之中的部分((?=%))，不计入返回结果

后行断言：“后行断言”正好与“先行断言”相反，根据前面单个字符匹配后面字符，只能是前面规定的那个字符

示例：(?<=前面的字符)后面的字符

```
x只有在y后面才匹配  
/(?=<=y)x/  
只匹配美元符号之后的数字  
/(?=<=\$)\d+/  
/(?=<=\$)\d+/.exec('Benjamin Franklin is on the $100 bill') // ["100"]
```

使用后行断言进行字符串替换

```
const RE_DOLLAR_PREFIX = /(?(?=<=$)foo)/g;
'$foo %foo foo'.replace(RE_DOLLAR_PREFIX, 'bar');
// '$bar %foo foo'
```

只有在美元符号后面的 `foo` 才会被替换

实现原理：“后行断言”的实现，需要先匹配 `/(?(?=<=y)x/` 的 `x`，然后再回到左边，匹配 `y` 的部分，**执行顺序“先右后左”**

因此，“后行断言”的**反斜杠引用**，也与通常的**顺序相反**，必须放在对应的那个括号之前。

```
/(?(?=<(o)d\1)r/.exec('hodor') // null
/(?(?=<\1d(o))r/.exec('hodor') // ["r", "o"]
```

反斜杠引用 (`\1`)

23.7 s 修饰符：dotAll 模式

点 (.) 是一个特殊字符，代表任意的单个字符，但是有两个例外：一个是四个字节的 UTF-16 字符，这个可以用 `u` 修饰符解决；另一个是行终止符：换行符 `\n`、回车符 `\r`

`s` 修饰符，使得 `.` 可以匹配任意单个字符，被称为 `dotAll` 模式，即点 (dot) 代表一切字符

示例：从 `li` 标签中提取标题和时间内容

```
/ s修饰符，让.可以匹配任意字符
// 案例：从下面字符串中提取标题和时间内容
let str = `
<ul>
  <li>
    <a>肖申克的救赎</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>`;
let reg = /<li>.*?<a>(.*)</a>.*?<p>(.*)</p>/gs;
// let res = reg.exec(str)
// console.log(res);
// 通过循环将匹配结果放入数组汇总
let result
let data = []
while (result = reg.exec(str)) {
  data.push({ title: result[1], time: result[2] })
}
console.log(data);
```

23.8 String.prototype.matchAll()

`String.prototype.matchAll()` 方法，可以一次性取出所有匹配（不需要循环取出所有匹配结果），返回的是一个遍历器（Iterator），因此可以用 `for...of` 循环取出，相比较与数组，在数据大时更节省资源。

用 `for...of` 循环取出匹配结果

示例：

```
let str = `<ul>
  <li>
    <a>肖申克的救赎</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>`;
let reg = /<li>.*?<a>(.*)</a>.*?<p>(.*)</p>/gs;
// 一次性去除标题和时间
// 调用方法
let res = str.matchAll(reg)
for (let i of res) {
  console.log(i);
}
```

```
▼ (3) ["<li><a>肖申克的救赎</a></li>", "肖申克的救赎", "1994-09-10"]
  ▼ 0: "<li><a>肖申克的救赎</a></li>" ["肖申克的救赎", "1994-09-10"]
    ▍ 1: "肖申克的救赎"
    ▍ 2: "上映日期: 1994-09-10"
    groups: undefined
    index: 26
    input: "<ul><li><a>肖申克的救赎</a></li>" ["肖申克的救赎", "1994-09-10"]
    length: 3
    ▶ __proto__: Array(0)

  ▼ (3) ["<li><a>阿甘正传</a></li>", "阿甘正传", "1994-07-06"]
    ▼ 0: "<li><a>阿甘正传</a></li>" ["阿甘正传", "1994-07-06"]
      ▍ 1: "阿甘正传"
      ▍ 2: "上映日期: 1994-07-06"
      groups: undefined
      index: 130
      input: "<ul><li><a>肖申克的救赎</a></li>" ["肖申克的救赎", "1994-09-10"]
      length: 3
      ▶ __proto__: Array(0)
```

遍历器转为数组 `...`` 运算符

```
// 将遍历器结果转换为数组
let arr = [...res]
console.log(arr);
```

```

▼ (2) [Array(3), Array(3)]
  ▼ 0: Array(3)
    0: "<li><a href='http://www.1993.org/movies/1994-09-10/130'>肖申克的救赎</a>"      <p>上映日期: 1994-09-10</p>
    1: "肖申克的救赎"
    2: "上映日期: 1994-09-10"
    groups: undefined
    index: 26
    input: "<ul><li><a href='http://www.1993.org/movies/1994-09-10/130'>肖申克的救赎</a>"      <a href='http://www.1993.org/movies/1994-09-10/130'>肖申克的救赎</a>"      <p>上映日期: 1994-09-10</p>
    length: 3
    ▶ __proto__: Array(0)

  ▼ 1: Array(3)
    0: "<li><a href='http://www.1993.org/movies/1994-07-06/130'>阿甘正传</a>"      <p>上映日期: 1994-07-06</p>
    1: "阿甘正传"
    2: "上映日期: 1994-07-06"
    groups: undefined
    index: 130
    input: "<ul><li><a href='http://www.1993.org/movies/1994-07-06/130'>肖申克的救赎</a>"      <a href='http://www.1993.org/movies/1994-07-06/130'>肖申克的救赎</a>"      <p>上映日期: 1994-07-06</p>
    length: 3
    ▶ __proto__: Array(0)
  length: 2
  ▶ __proto__: Array(0)

```

24. Class的基本语法

24.1 static关键字

类 (class) 通过 **static** 关键字定义静态方法或属性，表示方法不会被实例继承

特点：

1. 静态方法调用直接在类上进行，不能在类的实例上调用（**只能通过类调用，不能通过类的实例调用**）。如果在实例上调用静态方法，会抛出一个错误，表示不存在该方法。
2. 静态方法通常用于创建实用程序函数。
3. 静态方法下的 `this` 指的是类，而不是实例。
4. 静态方法可以与非静态方法重名。

示例：

```

class Foo {
  static staticProperty = 'someValue';
  static bar() {
    this.baz(); //this指向Foo, 等同于调用Foo.baz()
  }
  static baz() {
    console.log('hello');
  }
  baz() { // 静态方法可以与非静态方法重名
    console.log('world');
  }
}

Foo.staticProperty // 使用类调用静态属性
Foo.baz() // 使用类调用静态方法
Foo.bar() // hello

```

5. 父类的静态方法，**可以被子类继承**。

6. 静态方法也可以从 `super` 对象上调用。

```
class Foo {
    static classMethod() {
        return 'hello';
    }
}

class Bar extends Foo {
    static classMethod() {
        return super.classMethod() + ', too';
    }
}

Bar.classMethod() // "hello, too"
```

24.2 私有属性

在属性名之前，使用 # 表示，前缀 # 表示私有属性

#count 就是私有属性，只能在类的内部使用（this.#count）。如果在类的外部使用，就会报错。

```
class IncreasingCounter {
    #count = 0;
    get value() {
        console.log('Getting the current value!');
        return this.#count;
    }
    increment() {
        this.#count++;
    }
}
```

私有属性也可以设置 getter 和 setter 方法，也可以用#表示私有方法

```
class Foo {
    #a;
    #b;
    constructor(a, b) {
        this.#a = a;
        this.#b = b;
    }
    #sum() { // #sum() 就是一个私有方法
        return this.#a + this.#b;
    }
    get #x() { return #a; } // #x 是一个私有属性
    printSum() {
        console.log(this.#sum());
    }
}
```

可以加上 static 关键字，表示这是一个静态的私有属性或私有方法。

```
static PI = 22 / 7;
static #totallyRandomNumber = 4;
```

24.3 实例属性的新写法

实例属性除了定义在 `constructor()` 方法里面的 `this` 上面，也可以**定义在类的最顶层**。这样的写法比较整齐清晰，一眼就知道有哪些实例属性。

示例：

```
class IncreasingCounter {
    // 现在可以直接写在类的最顶层
    _count = 0

    /*以前的做法，将属性写在constructor里面
    constructor() {
        this._count = 0;
    }*/
    get value() {
        console.log('Getting the current value!');
        return this._count;
    }
    increment() {
        this._count++;
    }
}
```

实例属性 `_count` 与取值函数 `value()` 和 `increment()` 方法，**处于同一个层级**。这时，**不需要在实例属性前面加上 `this`**。