**Promise封装数据请求**

*const* getJson =*function*(*url*){//创建数据请求函数

                return new Promise((*resolve*,*reject*)*=>*{//在数据请求函数中返回Promise对象

*const* xhr = new XMLHttpRequest();// 在对象中用原生JS封装发送请求

                    xhr.open('GET',*url*)//指定请求方式和接口

                    xhr.onreadystatechange = handler;//封装状态改变函数

                    xhr.responseType = 'json';//指定返回数据的类型

                    xhr.setRequestHeader('Accept','application/json');//设置请求头

                    xhr.send();//发送数据到后台

*function* handler(){//封装状态改变函数

                        if(this.readyState === 4){

                            if(this.status === 200){

*resolve*(this.response)

                            }else{

*reject*(new Error(this.statusText));

                            }

                        }

                    }

                })

            }

            // 调用封装好的函数

*var* url = 'http://jindianwl.com/weixin/index/test\_api.html';

            getJson(url).then((*data*)*=>*{

                console.log(*data*);

            }).catch((*error*)*=>*{

                console.log(*error*);

            })

            // getJson(url).then((data)=>{

            //  console.log(data);

            // },(error)=>{

            //  console.log(error);

            // })

**Promise的其他方法**

            // 01   resolve()方法   将任何对象转化成一个Promise对象

*let* op = Promise.resolve('Foo');

            console.log(op);

            op.then((*data*)*=>*{

                console.log(*data*);

            })

            // 02   all()方法    等待资源全部加载完成后菜开始初始化

*let* Promise1 = new Promise((*resolve*,*reject*)*=>*{});

*let* Promise2 = new Promise((*resolve*,*reject*)*=>*{});

*let* Promise3 = new Promise((*resolve*,*reject*)*=>*{});

*let* p4 = Promise.all([Promise1,Promise2,Promise3]);

            p4.then(()*=>*{

            }).catch((*err*)*=>*{

            })

            // 03   race()方法   设置请求超时的时间，并指定超时时的操作

*function* requestImg(*imgSrc*){

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

*const* img = new Image();

                    img.onload = *function*(){

*resolve*(img)

                    }

                    img.src= *imgSrc*;

                })

            }

*function* timeout(){

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

                    setTimeout(()*=>*{

*reject*('超时');

                    },3000)

                })

            }

            Promise.race([requestImg(image/asd.png),timeout()]).then((*data*)*=>*{

                console.log(*data*);

            }).catch((*err*)*=>*{

                console.log(*err*);

            })

**ES6 数组**

// 数组创建

Array.of()

// 将参数中所有值作为元素形成数组

console.log(Array.of(1, 2, 3, 4)); // [1, 2, 3, 4]

// 参数值可为不同类型

console.log(Array.of(1, '2', true)); // [1, '2', true]

// 参数为空时返回空数组

console.log(Array.of()); // []

Array.from()

// 将类数组对象或可迭代对象转化为数组。

// 参数为数组,返回与原数组一样的数组

console.log(Array.from([1, 2])); // [1, 2]

// 参数含空位

console.log(Array.from([1, , 3])); // [1, undefined, 3]

// 扩展的方法

// 查找

find()

// 查找数组中符合条件的元素,若有多个符合条件的元素，则返回第一个元素。

*let* arr = Array.of(1, 2, 3, 4);

console.log(arr.find(*item* *=>* *item* > 2)); // 3

// 数组空位处理为 undefined

console.log([, 1].find(*n* *=>* true)); // undefined

findIndex()

// 查找数组中符合条件的元素索引，若有多个符合条件的元素，则返回第一个元素索引。

*let* arr = Array.of(1, 2, 1, 3);

// 参数1：回调函数

// 参数2(可选)：指定回调函数中的 this 值

console.log(arr.findIndex(*item* *=>* *item* = 1)); // 0

// 数组空位处理为 undefined

console.log([, 1].findIndex(*n* *=>* true)); //0

// 填充

fill()

// 将一定范围索引的数组元素内容填充为单个指定的值。

*let* arr = Array.of(1, 2, 3, 4);

// 参数1：用来填充的值

// 参数2：被填充的起始索引

// 参数3(可选)：被填充的结束索引，默认为数组末尾

console.log(arr.fill(0,1,2)); // [1, 0, 3, 4]

copyWithin()

// 将一定范围索引的数组元素修改为此数组另一指定范围索引的元素。

// 参数1：被修改的起始索引

// 参数2：被用来覆盖的数据的起始索引

// 参数3(可选)：被用来覆盖的数据的结束索引，默认为数组末尾

console.log([1, 2, 3, 4].copyWithin(0,2,4)); // [3, 4, 3, 4]

// 参数1为负数表示倒数

console.log([1, 2, 3, 4].copyWithin(-2, 0)); // [1, 2, 1, 2]

console.log([1, 2, ,4].copyWithin(0, 2, 4)); // [, 4, , 4]

**ES6 数组遍历**

// 遍历

entries()

// 遍历键值对。

for(*let* [key, value] of ['a', 'b'].entries()){

    console.log(key, value);

}

// 0 "a"

// 1 "b"

// 不使用 for... of 循环

*let* entries = ['a', 'b'].entries();

console.log(entries.next().value); // [0, "a"]

console.log(entries.next().value); // [1, "b"]

// 数组含空位

console.log([...[,'a'].entries()]); // [[0, undefined], [1, "a"]]

keys()

// 遍历键名。

for(*let* key of ['a', 'b'].keys()){

    console.log(key);

}

// 0

// 1

// 数组含空位

console.log([...[,'a'].keys()]); // [0, 1]

values()

// 遍历键值。

for(*let* value of ['a', 'b'].values()){

    console.log(value);

}

// "a"

// "b"

// 数组含空位

console.log([...[,'a'].values()]); // [undefined, "a"]

// 包含

includes()

// 数组是否包含指定值。

// 注意：与 Set 和 Map 的 has 方法区分；Set 的 has 方法用于查找值；Map 的 has 方法用于查找键名。

// 参数1：包含的指定值

[1, 2, 3].includes(1);    // true

// 参数2：可选，搜索的起始索引，默认为0

[1, 2, 3].includes(1, 2); // false

// NaN 的包含判断

[1, NaN, 3].includes(NaN); // true

// 嵌套数组转一维数组

flat()

console.log([1 ,[2, 3]].flat()); // [1, 2, 3]

// 指定转换的嵌套层数

console.log([1, [2, [3, [4, 5]]]].flat(2)); // [1, 2, 3, [4, 5]]

// 不管嵌套多少层

console.log([1, [2, [3, [4, 5]]]].flat(Infinity)); // [1, 2, 3, 4, 5]

// 自动跳过空位

console.log([1, [2, , 3]].flat());<p> // [1, 2, 3]

flatMap()

// 先对数组中每个元素进行了的处理，再对数组执行 flat() 方法。

// 参数1：遍历函数，该遍历函数可接受3个参数：当前元素、当前元素索引、原数组

// 参数2：指定遍历函数中 this 的指向

console.log([1, 2, 3].flatMap(*n* *=>* [n \* 2])); // [2, 4, 6]

**ES6 class类**

            // ES5造类的方式

*function* Person(*name*,*age*) {

                this.name = *name*;

                this.age = *age*;

            }

            Person.prototype.sayNanme = *function* () {

                return this.name;

            }

*let* p1 = new Person('小鱼儿',28);

            console.log(p1);

            // ES6造类的方式

*class* Person2{

                //实例化的时候回立即被调用

*constructor*(*name*,*age*){

                    this.name = *name*;

                    this.age = *age*;

                }

                sayName(){

                    return this.name;

                }

                sayAge(){

                    return this.Age;

                }

            }

*let* p2 = new Person('小鱼儿2',30);

            console.log(p2);

**ES6 class类的继承**

  // 父类

        // 使用关键字 extends

*class* Animal{

*constructor*(*name*,*age*){

                    this.name = *name*;

                    this.age = *age*;

                }

                sayName(){

                    return this.name;

                }

                sayAge(){

                    return this.Age;

                }

            }

*class* Dog extends Animal{

*constructor*(*name*,*age*,*color*){

                    super(*name*,*age*);//相当于Animal.call(this,name,age)   call改变this指向

                    this.color = *color*;

                }

                // 子类自己内部的方法

                sayColor(){

                      return `${this.name}是${this.age}岁了，他的颜色是${this.color}`;

                }

            }

            // 创建实例

*let* dog = new Dog('小黄',28,'红褐色');

            // 继承父类属性

            console.log(dog);

            //调用自己内部方法

            console.log(dog.sayColor());

**ES6 模块**

**创建一个外部JS文件**

// ES6的模块的功能主要有两个命令组成：export 和 import

// export 为抛出   用于规定模块的对外接口

// import 为导入   用于输入其他模块的提供的接口

// 一个模块就是一个独立的变量

// 实例

// 抛出数据

export *const* name = '张三';

// 抛出函数

export *function* sayNmae(){

}

// 结构是通过{sayNmae} 拿到

// 另一种抛出方式

// 一个JS文件中只能用一次

export default *Object*;

// 结构是通过Object 拿到

**在页面是接受抛出的数据并结构**

        // ES6模块    module

        // 使用模块需要安装Live Server插件

        // 这种方式需要服务器的支持

        import {name} from './modules/index.js';  //{}为结构赋值，结构module中的name ，可以接受多个参数

        console.log(name);

**ES5、ES6数组操作方法**

ES6数组操作方法

SevenLonely

0.228

2019.07.30 11:10:36

字数 1,265阅读 270

一、concat()

concat() 方法用于连接两个或多个数组。该方法不会改变现有的数组，仅会返回被连接数组的一个副本。

*var* arr1 = [1,2,3];

*var* arr2 = [4,5];

*var* arr3 = arr1.concat(arr2);

console.log(arr1); //[1, 2, 3]

console.log(arr3); //[1, 2, 3, 4, 5]

二、join()

join() 方法用于把数组中的所有元素放入一个字符串。元素是通过指定的分隔符进行分隔的，默认使用','号分割，不改变原数组。

*var* arr = [2,3,4];

console.log(arr.join()); //2,3,4

console.log(arr);  //[2, 3, 4]

三、push()

push() 方法可向数组的末尾添加一个或多个元素，并返回新的长度。末尾添加，返回的是长度，会改变原数组。

*var* a = [2,3,4]; *var* b = a.push(5);

console.log(a); //[2,3,4,5]

console.log(b);  //4

push方法可以一次添加多个元素push(data1,data2....)

四、pop()

pop() 方法用于删除并返回数组的最后一个元素。返回最后一个元素，会改变原数组。

*var* arr = [2,3,4];

console.log(arr.pop()); //4

console.log(arr);  //[2,3]

五、shift()

shift() 方法用于把数组的第一个元素从其中删除，并返回第一个元素的值。返回第一个元素，改变原数组。

*var* arr = [2,3,4];

console.log(arr.shift()); //2

console.log(arr);  //[3,4]

六、unshift()

unshift() 方法可向数组的开头添加一个或更多元素，并返回新的长度。返回新长度，改变原数组。

*var* arr = [2,3,4,5];

console.log(arr.unshift(3,6)); //6

console.log(arr); //[3, 6, 2, 3, 4, 5]

tip:该方法可以不传参数,不传参数就是不增加元素。

七、slice()

返回一个新的数组，包含从 start 到 end （不包括该元素）的 arrayObject 中的元素。返回选定的元素，该方法不会修改原数组。

*var* arr = [2,3,4,5];

console.log(arr.slice(1,3));  //[3,4]

console.log(arr);  //[2,3,4,5]

八、splice()

splice() 方法可删除从 index 处开始的零个或多个元素，并且用参数列表中声明的一个或多个值来替换那些被删除的元素。如果从 arrayObject 中删除了元素，则返回的是含有被删除的元素的数组。splice() 方法会直接对数组进行修改。

*var* a = [5,6,7,8];

console.log(a.splice(1,0,9)); //[]

console.log(a);  // [5, 9, 6, 7, 8]

*var* b = [5,6,7,8];

console.log(b.splice(1,2,3));  //[6, 7]

console.log(b); //[5, 3, 8]

九、substring() 和 substr()

相同点：如果只是写一个参数，两者的作用都一样：都是是截取字符串从当前下标以后直到字符串最后的字符串片段。

substr(startIndex);

substring(startIndex);

*var* str = '123456789';

console.log(str.substr(2));    // "3456789"

console.log(str.substring(2)) ;// "3456789"

不同点：第二个参数

substr（startIndex,lenth）： 第二个参数是截取字符串的长度（从起始点截取某个长度的字符串）；

substring（startIndex, endIndex）： 第二个参数是截取字符串最终的下标 （截取2个位置之间的字符串,‘含头不含尾’）。

console.log("123456789".substr(2,5));    // "34567"

console.log("123456789".substring(2,5)) ;// "345"

十、sort 排序

按照 Unicode code 位置排序，默认升序

*var* fruit = ['cherries', 'apples', 'bananas'];

fruit.sort(); // ['apples', 'bananas', 'cherries']

*var* scores = [1, 10, 21, 2];

scores.sort(); // [1, 10, 2, 21]

十一、reverse()

reverse() 方法用于颠倒数组中元素的顺序。返回的是颠倒后的数组，会改变原数组。

*var* arr = [2,3,4];

console.log(arr.reverse()); //[4, 3, 2]

console.log(arr);  //[4, 3, 2]

十二、indexOf 和 lastIndexOf

都接受两个参数：查找的值、查找起始位置

不存在，返回 -1 ；存在，返回位置。indexOf 是从前往后查找， lastIndexOf 是从后往前查找。

indexOf

*var* a = [2, 9, 9];

a.indexOf(2); // 0

a.indexOf(7); // -1

if (a.indexOf(7) === -1) {

  // element doesn't exist in array

}

lastIndexOf

*var* numbers = [2, 5, 9, 2];

numbers.lastIndexOf(2);     // 3

numbers.lastIndexOf(7);     // -1

numbers.lastIndexOf(2, 3);  // 3

numbers.lastIndexOf(2, 2);  // 0

numbers.lastIndexOf(2, -2); // 0

numbers.lastIndexOf(2, -1); // 3

十三、every

对数组的每一项都运行给定的函数，每一项都返回 ture,则返回 true

*function* isBigEnough(*element*, *index*, *array*) { return *element* < 10;

}

[2, 5, 8, 3, 4].every(isBigEnough);   // true

十四、some

对数组的每一项都运行给定的函数，任意一项都返回 ture,则返回 true

*function* compare(*element*, *index*, *array*) { return *element* > 10;

}

[2, 5, 8, 1, 4].some(compare);  // false

[12, 5, 8, 1, 4].some(compare); // true

十五、filter

对数组的每一项都运行给定的函数，返回 结果为 ture 的项组成的数组

*var* words = ["spray", "limit", "elite", "exuberant", "destruction", "present", "happy"];

*var* longWords = words.filter(*function*(*word*){ return *word*.length > 6;}); // Filtered array longWords is ["exuberant", "destruction", "present"]

十六、map

对数组的每一项都运行给定的函数，返回每次函数调用的结果组成一个新数组

*var* numbers = [1, 5, 10, 15];

*var* doubles = numbers.map(*function*(*x*) { return *x* \* 2;}); // doubles is now [2, 10, 20, 30] // numbers is still [1, 5, 10, 15]

十七、forEach 数组遍历

*const* items = ['item1', 'item2', 'item3'];

*const* copy = [];

items.forEach(*function*(*item*){

  copy.push(*item*)

});

ES6新增新操作数组的方法

1、find()：

传入一个回调函数，找到数组中符合当前搜索规则的第一个元素，返回它，并且终止搜索。

*const* arr = [1, "2", 3, 3, "2"]

console.log(arr.find(*n* *=>* typeof *n* === "number")) // 1

2、findIndex()：

传入一个回调函数，找到数组中符合当前搜索规则的第一个元素，返回它的下标，终止搜索。

console.log(arr.findIndex(*n* *=>* typeof *n* === "number")) // 0

3、fill()：

用新元素替换掉数组内的元素，可以指定替换下标范围。

arr.fill(value, start, end)

4、copyWithin()：

选择数组的某个下标，从该位置开始复制数组元素，默认从0开始复制。也可以指定要复制的元素范围。

arr.copyWithin(target, start, end)

*const* arr = [1, 2, 3, 4, 5]

console.log(arr.copyWithin(3)) // [1,2,3,1,2] 从下标为3的元素开始，复制数组，所以4, 5被替换成1, 2

*const* arr1 = [1, 2, 3, 4, 5]

console.log(arr1.copyWithin(3, 1)) // [1,2,3,2,3] 从下标为3的元素开始，复制数组，指定复制的第一个元素下标为1，所以4, 5被替换成2, 3

*const* arr2 = [1, 2, 3, 4, 5]

console.log(arr2.copyWithin(3, 1, 2)) // [1,2,3,2,5] 从下标为3的元素开始，复制数组，指定复制的第一个元素下标为1，结束位置为2，所以4被替换成2

5、from

将类似数组的对象（array-like object）和可遍历（iterable）的对象转为真正的数组

*const* bar = ["a", "b", "c"];

Array.from(bar); // ["a", "b", "c"]

 Array.from('foo'); // ["f", "o", "o"]

6、of

用于将一组值，转换为数组。这个方法的主要目的，是弥补数组构造函数 Array() 的不足。因为参数个数的不同，会导致 Array() 的行为有差异。

Array() // []

Array(3) // [, , ,]

Array(3, 11, 8) // [3, 11, 8]

Array.of(7);       // [7]

Array.of(1, 2, 3); // [1, 2, 3]

 Array(7);          // [ , , , , , , ]

Array(1, 2, 3);    // [1, 2, 3]

7、entries() 返回迭代器：返回键值对

//数组

*const* arr = ['a', 'b', 'c']; for(*let* v of arr.entries()) {

  console.log(v)

} // [0, 'a'] [1, 'b'] [2, 'c']

//Set

*const* arr = new Set(['a', 'b', 'c']); for(*let* v of arr.entries()) {

  console.log(v)

} // ['a', 'a'] ['b', 'b'] ['c', 'c']

//Map

*const* arr = new Map();

arr.set('a', 'a');

arr.set('b', 'b'); for(*let* v of arr.entries()) {

  console.log(v)

} // ['a', 'a'] ['b', 'b']

8、values() 返回迭代器：返回键值对的value

//数组

*const* arr = ['a', 'b', 'c']; for(*let* v of arr.values()) {

  console.log(v)

} //'a' 'b' 'c'

//Set

*const* arr = new Set(['a', 'b', 'c']); for(*let* v of arr.values()) {

  console.log(v)

} // 'a' 'b' 'c'

//Map

*const* arr = new Map();

arr.set('a', 'a');

arr.set('b', 'b'); for(*let* v of arr.values()) {

  console.log(v)

} // 'a' 'b'

9、keys() 返回迭代器：返回键值对的key

//数组

*const* arr = ['a', 'b', 'c']; for(*let* v of arr.keys()) {

  console.log(v)

} // 0 1 2

//Set

*const* arr = new Set(['a', 'b', 'c']); for(*let* v of arr.keys()) {

  console.log(v)

} // 'a' 'b' 'c'

//Map

*const* arr = new Map();

arr.set('a', 'a');

arr.set('b', 'b'); for(*let* v of arr.keys()) {

  console.log(v)

} // 'a' 'b'

10、includes

判断数组中是否存在该元素，参数：查找的值、起始位置，可以替换 ES5 时代的 indexOf 判断方式。indexOf 判断元素是否为 NaN，会判断错误。

*var* a = [1, 2, 3];

a.includes(2); // true

a.includes(4); // false

11.获取所有对象的key值

*const* obj = {'ad':1,"dsdafsd":32};

*const* keys = Object.getOwnPropertyNames(obj);

console.log(keys)

12. 替换对象所有的key

*const* data = {id：‘11’,name:‘张三’};

*const* keyMap = {id: ‘序列’, name: ‘姓名’};

*const* objs= Object.keys(data).reduce((*newData*, *key*) *=>* {

*let* newKey = keyMap[*key*] || *key*

*newData*[newKey] = data[*key*]

    return *newData*

}, {})

console.log(objs)

data对象就变成了 {序列：“11”,姓名：“张三” }

[**ES6中数组遍历的方法**](https://www.cnblogs.com/qjb2404/p/12202836.html)

//for of

*let* arr = [1,2,3,4,5]

for(*let* item of arr){

   console.log(item)

}

es6新增加的for of方法不仅可以遍历数组和对象，还可以遍历自定义的数据结构

ES6提供三个新的方法——entries()，keys()和values()——用于遍历数组。它们都返回一个遍历器对象. for...of 会自动的调用遍历器的next() .

5. 数组实例的 entries() 、keys() 和values()

ES6提供三个新的方法——entries()，keys()和values()——用于遍历数组。它们都返回一个遍历器对象. for...of 会自动的调用遍历器的next() .

for (*let* index of ['a', 'b'].keys()) {

 console.log(index);

}

// 0

// 1

for (*let* elem of ['a', 'b'].values()) {

 console.log(elem);

}

// 'a'

// 'b'

for (*let* [index, elem] of ['a', 'b'].entries()) {

 console.log(index, elem);

}

// 0 "a"

// 1 "b"

如果不使用for...of循环，可以手动调用遍历器对象的next方法，进行遍历。

*let* letter = ['a', 'b', 'c'];

*let* entries = letter.entries();

console.log(entries.next().value); // [0, 'a']

console.log(entries.next().value); // [1, 'b']

console.log(entries.next().value); // [2, 'c']