集合引用类型

Object

显式创建Object两种方式：

- new Object()
- 对象字面量表示法

```
let person = new Object();
person.name = "Nicholas";
person.age = 29;

let person = {
 name: "Nicholas",
 age: 29
};
```

最后一个属性后加逗号在非常老的浏览器会报错，现代浏览器都支持

对象字面量定义对象时，并不会调用Object构造函数！

属性访问可以用点语法，也可以用中括号，中括号里面需要属性名的字符串！

Array

ES的数组可以每个槽位都可以存储任意类型的数据

创建数组

- 使用构造函数new Array()，使用Array构造函数可以省略new

```
let colors = new Array(3); // 创建一个包含 3 个元素的数组
let names = new Array("Greg"); // 创建一个只包含一个元素，即字符串"Greg"的数组
let colors = new Array("red", "blue", "green");

let colors = Array(3); // 创建一个包含 3 个元素的数组
let names = Array("Greg"); // 创建一个只包含一个元素，即字符串"Greg"的数组
```

- 数组字面量

```
let colors = ["red", "blue", "green"]; // 创建一个包含 3 个元素的数组
let names = []; // 创建一个空数组
let values = [1,2,]; // 创建一个包含 2 个元素的数组
```

- ES6新增 Array构造函数的静态方法 from() of()
  from() 类数组解构转换为数组 浅拷贝
  第二个参数可选映射函数参数
  - 字符串会被拆分为单字符数组
  - 将集合和映射转换为一个新数组
  - 对现有数组执行浅复制
  - arguments对象转换成数组
  - 转换带有必要属性（length）的自定义对象

```
// 字符串会被拆分为单字符数组
console.log(Array.from("Matt")); // ["M", "a", "t", "t"]
// 可以使用 from()将集合和映射转换为一个新数组
const m = new Map().set(1, 2)
.set(3, 4);
const s = new Set().add(1)
.add(2)
.add(3)
.add(4);
console.log(Array.from(m)); // [[1, 2], [3, 4]]
console.log(Array.from(s)); // [1, 2, 3, 4]
// Array.from()对现有数组执行浅复制
const a1 = [1, 2, 3, 4];
const a2 = Array.from(a1);
console.log(a1); // [1, 2, 3, 4]
alert(a1 === a2); // false
// 可以使用任何可迭代对象
const iter = {
*[Symbol.iterator]() {
yield 1;
yield 2;
yield 3;
yield 4;
}
};
console.log(Array.from(iter)); // [1, 2, 3, 4]
// arguments 对象可以被轻松地转换为数组
function getArgsArray() {
return Array.from(arguments);
}
console.log(getArgsArray(1, 2, 3, 4)); // [1, 2, 3, 4]
// from()也能转换带有必要属性的自定义对象
const arrayLikeObject = {
0: 1,
1: 2,
2: 3,
3: 4,
length: 4
};
console.log(Array.from(arrayLikeObject)); // [1, 2, 3, 4]


const a1 = [1, 2, 3, 4];
const a2 = Array.from(a1, x => x**2);
const a3 = Array.from(a1, function(x) {return x**this.exponent}, {exponent: 2});
console.log(a2); // [1, 4, 9, 16]
console.log(a3); // [1, 4, 9, 16]
```

of())可以把一组参数转换为数组。这个方法用于替代在 ES6之前常用的 Array.prototype.slice.call()

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

空位数组

使用数组字面量初始化数字时，可以用一串逗号表示空位 hole

ES6把空位当成undefined

```
const options = [,,,,,]; // 创建包含 5 个元素的数组
console.log(options.length); // 5
console.log(options); // [,,,,,]

// ES6
const options = [1,,,,5];
for (const option of options) {
 console.log(option === undefined);
}
// false
// true
// true
// true
// false

alert(Array.of(...[,,,])); // [undefined, undefined, undefined]
for (const [index, value] of options.entries()) {
 alert(value);
}
// 1
// undefined
// undefined
// undefined
// 5
```

ES6之前的方法会忽略空位

```
const options = [1,,,,5];
// map()会跳过空位置
console.log(options.map(() => 6)); // [6, undefined, undefined, undefined, 6]
// join()视空位置为空字符串
console.log(options.join('-')); // "1----5"
```

能不用空位就不用空位，一定需要，则显式使用undefined

数组索引

可以通过修改.length属性 添加/删除数组末尾元素

```
let colors = ["red", "blue", "green"]; // 创建一个包含 3 个字符串的数组
colors.length = 2;
alert(colors[2]); // undefined
```

数组最多可以包含 4 294 967 295 个元素

检测数组

判断数组两种方法

- instanceof
  只适用于一个全局执行上下文。
  如果有多个框架，传参，数组的构造函数可能不同
- Array.isArray()

迭代器方法

ES6中，Array原型上暴露了3个用于检索数组内容的方法
Key() 返回数组索引的迭代器
values() 返回数组元素的迭代器
entries() 返回索引/值的迭代器

因为式迭代器，一般都与Array.from()连用

```
const a = ["foo", "bar", "baz", "qux"];
// 因为这些方法都返回迭代器，所以可以将它们的内容
// 通过 Array.from()直接转换为数组实例
const aKeys = Array.from(a.keys());
const aValues = Array.from(a.values());
const aEntries = Array.from(a.entries());
console.log(aKeys); // [0, 1, 2, 3]
console.log(aValues); // ["foo", "bar", "baz", "qux"]
console.log(aEntries); // [[0, "foo"], [1, "bar"], [2, "baz"], [3, "qux"]]
```

ES6解构 非常容易地拆分键值对

```
const a = ["foo", "bar", "baz", "qux"];
for (const [idx, element] of a.entries()) {
 alert(idx);
 alert(element);
}
// 0
// foo
// 1
// bar
// 2
// baz
// 3
// qux
```

复制和填充方法

ES6 新增两个方法：
批量复制 copyWithin()
填充数组 fill()

不会改变数组大小

Array.fill(fillValue,start,end) [start,end)

```
const zeroes = [0, 0, 0, 0, 0];
// 用 5 填充整个数组
zeroes.fill(5);
console.log(zeroes); // [5, 5, 5, 5, 5]
zeroes.fill(0); // 重置
// 用 6 填充索引大于等于 3 的元素
zeroes.fill(6, 3);
console.log(zeroes); // [0, 0, 0, 6, 6]
zeroes.fill(0); // 重置
// 用 7 填充索引大于等于 1 且小于 3 的元素
zeroes.fill(7, 1, 3);
console.log(zeroes); // [0, 7, 7, 0, 0];
zeroes.fill(0); // 重置
// 用 8 填充索引大于等于 1 且小于 4 的元素
// (-4 + zeroes.length = 1)
// (-1 + zeroes.length = 4)
zeroes.fill(8, -4, -1);
console.log(zeroes); // [0, 8, 8, 8, 0];
```

fill()静默忽略超出数组边界、零长度及方向相反的索引范围：

```
const zeroes = [0, 0, 0, 0, 0];
// 索引过低，忽略
zeroes.fill(1, -10, -6);
console.log(zeroes); // [0, 0, 0, 0, 0]
// 索引过高，忽略
zeroes.fill(1, 10, 15);
console.log(zeroes); // [0, 0, 0, 0, 0]
// 索引反向，忽略
zeroes.fill(2, 4, 2);
console.log(zeroes); // [0, 0, 0, 0, 0]
// 索引部分可用，填充可用部分
zeroes.fill(4, 3, 10)
console.log(zeroes); // [0, 0, 0, 4, 4]
```

copyWithin()
浅拷贝

```
let ints,
 reset = () => ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();
// 从 ints 中复制索引 0 开始的内容，插入到索引 5 开始的位置
// 在源索引或目标索引到达数组边界时停止
ints.copyWithin(5);
```

```
console.log(ints); // [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
reset();
// 从 ints 中复制索引 5 开始的内容，插入到索引 0 开始的位置
ints.copyWithin(0, 5);
console.log(ints); // [5, 6, 7, 8, 9, 5, 6, 7, 8, 9]
reset();
// 从 ints 中复制索引 0 开始到索引 3 结束的内容
// 插入到索引 4 开始的位置
ints.copyWithin(4, 0, 3);
alert(ints); // [0, 1, 2, 3, 0, 1, 2, 7, 8, 9]
reset();
// JavaScript 引擎在插值前会完整复制范围内的值
// 因此复制期间不存在重写的风险
ints.copyWithin(2, 0, 6);
alert(ints); // [0, 1, 0, 1, 2, 3, 4, 5, 8, 9]
reset();
// 支持负索引值，与 fill()相对于数组末尾计算正向索引的过程是一样的
ints.copyWithin(-4, -7, -3);
alert(ints); // [0, 1, 2, 3, 4, 5, 3, 4, 5, 6]
```

copyWithin()静默忽略超出数组边界、零长度及方向相反的索引范围：

```
let ints,
 reset = () => ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();
// 索引过低，忽略
ints.copyWithin(1, -15, -12);
alert(ints); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset()
// 索引过高，忽略
ints.copyWithin(1, 12, 15);
alert(ints); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();
// 索引反向，忽略
ints.copyWithin(2, 4, 2);
alert(ints); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();
// 索引部分可用，复制、填充可用部分
ints.copyWithin(4, 7, 10)
alert(ints); // [0, 1, 2, 3, 7, 8, 9, 7, 8, 9];
```

转换方法

valueOf() 返回数组本身
toString() 返回由数组中每个值的等效字符串用，拼接
toLocaleString() 对每个值调用toLocaleString()方法

```
let person1 = {
 toLocaleString() {
 return "Nikolaos";
 },
 toString() {
 return "Nicholas";
 }
};
```

```
let person2 = {
 toLocaleString() {
 return "Grigorios";
 },
 toString() {
 return "Greg";
 }
};
let people = [person1, person2];
alert(people); // Nicholas,Greg
alert(people.toString()); // Nicholas,Greg
alert(people.toLocaleString()); // Nikolaos,Grigorios
```

栈方法

看起来像另一种数据结构

- push() 任意数量参数，添加到数组末尾
- pop() 移除数组最后一项，并返回删除项

队列方法

- push()
- shift() 删除队列第一项，并返回删除项
- unshift() 与shift相反，队首插入 (非队列动作)

排序方法

reverse() 反向排列
sort() 升序排序
两者都返回 数组的引用
sort()可以接收比较函数
比较函数 返回-1 1 0
sort()根据-1 1 0 进行排序
第一个参数应该排在第二个参数前面，就返回负值
两个参数相等，就返回 0
第一个参数应该排在第二个参数后面，就返回正值

```
function compare(value1, value2) {
 if (value1 < value2) {
 return -1;
 } else if (value1 > value2) {
 return 1;
 } else {
 return 0;
 }
}

let values = [0, 1, 5, 10, 15];
values.sort((a, b) => a < b ? 1 : a > b ? -1 : 0);
alert(values); // 15,10,5,1,0
```

```
// 如果是数值 还可以更简单
function compare(value1, value2){
 return value2 - value1;
}

// (value1,value2)=> value2-value1
```

操作方法

- concat()
  参数可以是多个参数

  i. 创建数组副本
  ii. 如果参数是数组，每一项加入新数组尾部
  iii. 如果参数不是数组，直接加入新数组尾部
  iv. 返回副本

```
let colors = ["red", "green", "blue"];
let colors2 = colors.concat("yellow", ["black", "brown"]);
console.log(colors); // ["red", "green","blue"]
console.log(colors2); // ["red", "green", "blue", "yellow", "black", "brown"]
```

  参数是数值是否打平可以重写，通过数组中加入Symbol.isConcatSpreadable
  true强制打平，false不打平

```
let colors = ["red", "green", "blue"];
let newColors = ["black", "brown"];
let moreNewColors = {
 [Symbol.isConcatSpreadable]: true,
 length: 2,
 0: "pink",
 1: "cyan"
};
newColors[Symbol.isConcatSpreadable] = false;
// 强制不打平数组
let colors2 = colors.concat("yellow", newColors);
// 强制打平类数组对象
let colors3 = colors.concat(moreNewColors);
console.log(colors); // ["red", "green", "blue"]
console.log(colors2); // ["red", "green", "blue", "yellow", ["black", "brown"]]
console.log(colors3); // ["red", "green", "blue", "pink", "cyan"]
```

- slice()
  创建一个包含原有数组中1个或多个的新数组
  浅拷贝
  接收一个或2个参数
  slice(startIndex:number,endInedx?:number)
  参数是负数代表倒数第几个。

```
let colors = ["red", "green", "blue", "yellow", "purple"];
let colors2 = colors.slice(1);
let colors3 = colors.slice(1, 4);
```

```
alert(colors2); // green,blue,yellow,purple
alert(colors3); // green,blue,yellow
```

- splice()
  splice(index,delNum,...insertValue)
  - 删除
    2个参数，(index，delNum)
  - 插入
    3个及以上参数，(index,0,...insertValue)
  - 替换
    3个及以上参数，(index,delNum,...insertValue)

搜索和位置方法

- 严格相等
  这三种方法都是全等比较
  - indexOf()
  - lastIndexOf()
  - includes() ES7新增

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
alert(numbers.indexOf(4)); // 3
alert(numbers.lastIndexOf(4)); // 5
alert(numbers.includes(4)); // true
alert(numbers.indexOf(4, 4)); // 5
alert(numbers.lastIndexOf(4, 4)); // 3
alert(numbers.includes(4, 7)); // false
let person = { name: "Nicholas" };
let people = [{ name: "Nicholas" }];
let morePeople = [person];
alert(people.indexOf(person)); // -1
alert(morePeople.indexOf(person)); // 0
alert(people.includes(person)); // false
alert(morePeople.includes(person)); // true
```

- 断言函数
  断言函数接收3个参数：元素，索引，数组本身
  find(),findIndex()

```
const people = [
 {
 name: "Matt",
 age: 27
 },
 {
 name: "Nicholas",
 age: 29
 }
];
alert(people.find((element, index, array) => element.age < 28));
// {name: "Matt", age: 27}
```

```
alert(people.findIndex((element, index, array) => element.age < 28));
// 0
```

找到匹配项后，这两个方法都不再继续搜索

```
const evens = [2, 4, 6];
// 找到匹配后，永远不会检查数组的最后一个元素
evens.find((element, index, array) => {
 console.log(element);
 console.log(index);
 console.log(array);
 return element === 4;
});
// 2
// 0
// [2, 4, 6]
// 4
// 1
// [2, 4, 6]
```

迭代方法

- every()：对数组每一项都运行传入的函数，如果对每一项函数都返回 true，则这个方法返回 true。
- filter()：对数组每一项都运行传入的函数，函数返回 true 的项会组成数组之后返回。
- forEach()：对数组每一项都运行传入的函数，没有返回值。
- map()：对数组每一项都运行传入的函数，返回由每次函数调用的结果构成的数组。
- some()：对数组每一项都运行传入的函数，如果有一项函数返回 true，则这个方法返回 true。

这些方法都不改变调用它们的数组

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let everyResult = numbers.every((item, index, array) => item > 2);
alert(everyResult); // false
let someResult = numbers.some((item, index, array) => item > 2);
alert(someResult); // true

let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let filterResult = numbers.filter((item, index, array) => item > 2);
alert(filterResult); // 3,4,5,4,3

let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let mapResult = numbers.map((item, index, array) => item * 2);
alert(mapResult); // 2,4,6,8,10,8,6,4,2

let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
numbers.forEach((item, index, array) => {
 // 执行某些操作
});
```

归并方法

reduce() reduceRight()
都会迭代数组所有项，构建返回一个最终值
reduce 从头遍历到尾
reduceRight 从尾遍历到头

两个方法接收2个参数

- 归并函数
  - prev 上一个归并值
  - cur 当前项
  - index 当前项索引
  - array 数组本身
- 初始值

如果没给初始值，迭代从数组第二项开始，prev为数组的第一项

```
let values = [1, 2, 3, 4, 5];
let sum = values.reduce((prev, cur, index, array) => prev + cur);
alert(sum); // 15
```