目录

Chapter 1

语言基础

任何语言的核心所描述的都是这门语言在最基本的层面上如何工作，涉及语法、操作符、数据类型以及内置功能，在此基础之上才可以构建复杂的解决方案。

1.1 语法

ECMAScript 的语法很大程度上借鉴了 C 语言和其他类 C 语言，如 Java 和 Perl。

1.1.1 区分大小写

ECMAScript 中一切都区分大小写。无论是变量、函数名还是操作符，都区分大小写。

1.1.2 标识符

所谓标识符，就是变量、函数、属性或函数参数的名称。标识符可以由一或多个下列字符组成：

- 第一个字符必须是一个字母、下划线（_）或美元符号（$）；
- 剩下的其他字符可以是字母、下划线、美元符号或数字。

标识符中的字母可以是扩展 ASCII（Extended ASCII）中的字母，也可以是 Unicode 的字母字符。

按照惯例，ECMAScript 标识符使用驼峰大小写形式，即第一个单词的首字母小写，后面每个单词的首字母大写。

关键字、保留字、true、false 和 null 不能作为标识符。具体内容参考关键字和保留字。

1.1.3 注释

ECMAScript 采用 C 语言风格的注释，包括单行注释和块注释。单行注释以两个斜杠字符开头。块注释以一个斜杠和一个星号（/*）开头，以它们的反向组合（*/）结尾。

1.1.4 严格模式

严格模式是一种不同的 JavaScript 解析和执行模型，ECMAScript 3 的一些不规范写法在这种模式下会被处理，对于不安全的活动将抛出错误。要对整个脚本启用严格模式，在脚本开头加上这一行：

```
"use strict";
```

1.1.5 语句

1.2 关键字和保留字

ECMA-262 描述了一组保留的关键字，这些关键字有特殊用途。按照规定，保留的关键字不能用作标识符或属性名。ECMA-262 第 6 版规定的所有关键字如下：break do in typeof case else instanceof var catch export new void class extends return while const finally super with continue for switch yield debugger function this default if throw delete import try

规范中也描述了一组未来的保留字，同样不能用作标识符或属性名。虽然保留字在语言中没有特定用途，但它们是保留给将来做关键字用的。

以下是 ECMA-262 第 6 版为将来保留的所有词汇。

- 始终保留：enum
- 严格模式下保留：implements package public interface protected static let private
- 模块代码中保留：await

1.3 变量

ECMAScript 变量是松散类型的，意思是变量可以用于保存任何类型的数据。每个变量只不过是一个用于保存任意值的命名占位符。有 3 个关键字可以声明变量：var、const 和 let。其中，var 在 ECMAScript 的所有版本中都可以使用，而 const 和 let 只能在 ECMAScript 6 及更晚的版本中使用。

1.3.1 var关键字

要定义变量，可以使用 var 操作符（注意 var 是一个关键字），后跟变量名（即标识符）。

```
var msg;
```

不初始化的情况下，变量会保存一个特殊值 undefined，下一节讨论数据类型时会谈到。）ECMAScript 实现变量初始化，因此可以同时定义变量并设置它的值：

```
var msg = 'hi';
```

message 被定义为一个保存字符串值 hi 的变量。像这样初始化变量不会将它标识为字符串类型，只是一个简单的赋值而已。随后，不仅可以改变保存的值，也可以改变值的类型：

```
var message = "hi";
message = 100; // legal, but not recommended
```

var声明作用域

关键的问题在于，使用 var 操作符定义的变量会成为包含它的函数的局部变量。

```
function test(){
        var msg = 'hi'; // local variable
    }
test();
console.log(msg); // error
```

不过，在函数内定义变量时省略 var 操作符，可以创建一个全局变量：

```
function test(){
        msg = 'hi'; // global variable
    }
test();
console.log(msg); // 'hi'
```

虽然可以通过省略 var 操作符定义全局变量，但不推荐这么做。在局部作用域中定义的全局变量很难维护，也会造成困惑。这是因为不能一下子断定省略 var 是不是有意而为之。在严格模式下，如果像这样给未声明的变量赋值，则会导致抛出 ReferenceError。

如果需要定义多个变量，可以在一条语句中用逗号分隔每个变量（及可选的初始化）：

```
var msg='hi',
found=false,
age=29;
```

因为 ECMAScript 是松散类型的，所以使用不同数据类型初始化的变量可以用一条语句来声明。插入换行和空格缩进并不是必需的，但这样有利于阅读理解。

在严格模式下，不能定义名为`eval`和`arguments`的变量，否则会导致语法错误。

var声明提升

`var`关键字声明的变量会自动提升到函数作用域顶部：

```
function foo() {
        console.log(age);
        var age = 26;
    }
foo(); // undefined
```

之所以不会报错，是因为 ECMAScript 运行时把它看成等价于如下代码：

```
function foo() {
        var age;
        console.log(age);
        age = 26;
    }
foo(); // undefined
```

这就是所谓的“提升”（hoist），也就是把所有变量声明都拉到函数作用域的顶部。此外，反复多次使用 var 声明同一个变量也没有问题。

1.3.2 let声明

let 跟 var 的作用差不多，但有着非常重要的区别。最明显的区别是，let 声明的范围是块作用域，而 var 声明的范围是函数作用域。

```
if (true) {
        var name = 'Matt';
        console.log(name);
    }
console.log(name);

if (true) {
        let age = 26;
        console.log(age);
    }
console.log(age); // ReferenceError: age is not defined
```

age 变量之所以不能在 if 块外部被引用，是因为它的作用域仅限于该块内部。块作用域是函数作用域的子集，因此适用于 var 的作用域限制同样也适用于 let。

let 也不允许同一个块作用域中出现冗余声明。

JavaScript 引擎会记录用于变量声明的标识符及其所在的块作用域，因此嵌套使用相同的标识符不会报错，而这是因为同一个块中没有重复声明。

对声明冗余报错不会因混用 let 和 var 而受影响。这两个关键字声明的并不是不同类型的变量，它们只是指出变量在相关作用域如何存在。

```
var name;
let name; // SyntaxError: Identifier 'name' has already been declared

let age;
var age; // SyntaxError: Identifier 'name' has already been declared
```

暂时性死区

全局声明

条件声明

for循环中的let声明

1.3.3 const声明

1.3.4 数据类型

typeof操作符

Undefined类型

Null类型

Boolean类型

Number类型

1.3.5 String类型

1.3.6 Symbol类型

1.3.7 Object类型

1.4 操作符

1.5 语句

1.5.1 if语句

1.5.2 do-while语句

1.5.3 while语句

1.5.4 for语句

1.5.5 for-in语句

1.5.6 for-of语句

1.5.7 标签语句

1.5.8 break和continue语句

break 和 continue 语句为执行循环代码提供了更严格的控制手段。其中，break 语句用于立即退出循环，强制执行循环后的下一条语句。而 continue 语句也用于立即退出循环，但会再次从循环顶部开始执

行。

1.5.9 with语句

1.5.10 switch语句

Chapter 2

集合引用类型

2.1 Object

2.2 Array

ECMAScript 数组也是一组有序的数据，数组中每个槽位可以存储任意类型的数据。这意味着可以创建一个数组，它的第一个元素是字符串，第二个元素是数值，第三个是对象。ECMAScript 数组也是动态大小的，会随着数据添加而自动增长。

2.2.1 创建数组

一种是使用 Array 构造函数。如果知道数组中元素的数量，那么可以给构造函数传入一个数值，然后 length 属性就会被自动创建并设置为这个值。可以给 Array 构造函数传入要保存的元素。

```
let colors1 = new Array();
let colors2 = new Array(20);
let colors3 = new Array("red", "green", "blue");
```

创建数组时可以给构造函数传一个值。这时候就有点问题了，因为如果这个值是数值，则会创建一个长度为指定数值的数组；而如果这个值是其他类型的，则会创建一个只包含该特定值的数组。

```
let colors = new Array(3);
// create an array with three items
let names = new Array("Stephen CUI");
// create an array with one item, the string "Stephen CUI"
```

在使用 Array 构造函数时，也可以省略 new 操作符。

另一种创建数组的方式是使用数组字面量（array literal）表示法。数组字面量是在中括号中包含以逗号分隔的元素列表。

```
let colors = ["red", "green", "blue"];
let names = [];
let values = [1, 2];
```

与对象一样，在使用数组字面量表示法创建数组不会调用 Array 构造函数。

Array 构造函数还有两个 ES6 新增的用于创建数组的静态方法：from()和 of()。from()用于将类数组结构转换为数组实例，而 of()用于将一组参数转换为数组实例。

Array.from()的第一个参数是一个类数组对象，即任何可迭代的结构，或者有一个 length 属性和可索引元素的结构。

```
// Strings will be broken up into an array of single characters
console.log(Array.from("Stephen"));
const m = new Map().set(1, 2).set(3, 4);
const s = new Set().add(1).add(2).add(4).add(4);
console.log(m);
console.log(s);

const a1 = [1, 2, 3, 4];
// Array.from() performs a shallow copy of an existing array
const a2 = Array.from(a1);

console.log(a1);
alert(a1 === a2); // false

// Any iterable object can be used
const iter = {
*[Symbol.iterator]() {
        yield 1;
        yield 2;
        yield 3;
        yield 4;
    },
};
console.log(Array.from(iter));
```

Array.from()还接收第二个可选的映射函数参数。这个函数可以直接增强新数组的值，而无须像调用 Array.from().map()那样先创建一个中间数组。还可以接收第三个可选参数，用于指定映射函数中 this 的值。但这个重写的 this 值在箭头函数中不适用。

```
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]
```

Array.of()可以把一组参数转换为数组。这个方法用于替代在 ES6 之前常用的一种异常笨拙的将 arguments 对象转换为数组的写法： `Array.prototype.slice.call(arguments)`。

```
console.log(Array.of(1, 2, 3, 4)); //[1, 2, 3, 5]
console.log(Array.of(undefined)); //[undefined]
```

2.2.2 数组空位

使用数组字面量初始化数组时，可以使用一串逗号来创建空位（hole）。ECMAScript 会将逗号之间相应索引位置的值当成空位，ES6 规范重新定义了该如何处理这些空位。

```
const options = [, , , , ,];
console.log(options.length);
console.log(options);
```

ES6 新增方法普遍将这些空位当成存在的元素，只不过值为 undefined：

```
const options = [1, , , , 5];
for (const option of options) {
        console.log(option == undefined); //false true true true false
    }
const a = Array.from([, , ,]);
for (const val of a) {
        console.log(val === undefined); // true * 3
    }
console.log(Array.of(...[, , ,])); //undefined undefined undefined
for (const [index, value] of options.entries()) {
```

```
            alert(value); //1 undefined undefined undefined 5
        }
```

由于行为不一致和存在性能隐患，因此实践中要避免使用数组空位。如果确实需要空位，则可以显式地用 undefined 值代替。

2.2.3 数组索引

要取得或设置数组的值，需要使用中括号并提供相应值的数字索引。中括号中提供的索引表示要访问的值。如果索引小于数组包含的元素数，则返回存储在相应位置的元素。设置数组的值方法也是一样的，就是替换指定位置的值。如果把一个值设置给超过数组最大索引的索引，则数组长度会自动扩展到该索引值。

```
let colors = ["red", "green", "blue"];
console.log(colors[0]);
colors[2] = "black";
colors[3] = "brown";
colors[7] = "red";
```

数组中元素的数量保存在 length 属性中，这个属性始终返回 0 或大于 0 的值。

```
let colors = ["red", "green", "blue"];
let names = [];
console.log(colors.length); //3
console.log(names.length); //0
```

数组 length 属性的独特之处在于，它不是只读的。通过修改 length 属性，可以从数组末尾删除或添加元素。

```
const colors = ["red", "green", "blue"];
colors.length = 2;
console.log(colors[2]); //undefined
```

使用 length 属性可以方便地向数组末尾添加元素:

```
const colors = ["red", "green", "blue"];
colors[colors.length] = "black";
colors[colors.length] = "brown";
console.log(colors); //['red', 'green', 'blue', 'black', 'brown']
```

数组最多可以包含 4 294 967 295 个元素，这对于大多数编程任务应该足够了。如果尝试添加更多项，则会导致抛出错误。以这个最大值作为初始值创建数组，可能导致脚本运行时间过长的错误。

2.2.4 检测数组

一个经典的 ECMAScript 问题是判断一个对象是不是数组。在只有一个网页（因而只有一个全局作用域）的情况下，使用 instanceof 操作符就足矣：

```
    if (value instanceof Array) {
            // do something on the array
        }
```

使用 instanceof 的问题是假定只有一个全局执行上下文。如果网页里有多个框架，则可能涉及两个不同的全局执行上下文，因此就会有两个不同版本的 Array 构造函数。如果要把数组从一个框架传给另一个框架，则这个数组的构造函数将有别于在第二个框架内本地创建的数组。

ECMAScript 提供了 Array.isArray()方法。这个方法的目的就是确定一个值是否为数组，而不用管它是在哪个全局执行上下文中创建的。

```
    if (Array.isArray(value)) {
            // do something on the array
        }
```

2.2.5 迭代器方法

在 ES6 中，Array 的原型上提供了 3 个用于检索数组内容的方法：keys()、values()和entries()。keys()返回数组索引的迭代器，values()返回数组元素的迭代器，而 entries()返回索引/值对的迭代器：

```
    const a = ["foo", "bar", "baz", "qux"];
    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']]
```

使用 ES6 的解构可以非常容易地在循环中拆分键/值对：

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

```
    // 2 'baz'
    // 3 'qux'
```

2.2.6 复制和填充方法

ES6 新增了两个方法：批量复制方法 copyWithin()，以及填充数组方法 fill()，都需要指定既有数组实例上的一个范围，包含开始索引，不包含结束索引（左闭右开）。

使用 fill()方法可以向一个已有的数组中插入全部或部分相同的值。开始索引用于指定开始填充的位置，它是可选的。如果不提供结束索引，则一直填充到数组末尾。负值索引从数组末尾开始计算。也可以将负索引想象成数组长度加上它得到的一个正索引：

```
const zeroes = [0, 0, 0, 0, 0];
zeroes.fill(0);
console.log(zeroes); // [5, 5, 5, 5, 5]
zeroes.fill(0); // reset

zeroes.fill(6, 3);
console.log(zeroes); // [5, 5, 5, 6, 6]
zeroes.fill(0); // reset

zeroes.fill(7, 1, 3);
console.log(zeroes); // [5, 7, 7, 5, 5]
zeroes.fill(0); // reset

zeroes.fill(8, -4, -1);
console.log(zeroes); // [5, 8, 8, 8, 5]
```

fill()静默忽略超出数组边界、零长度及方向相反的索引范围。

copyWithin()会按照指定范围浅复制数组中的部分内容，然后将它们插入到指定索引开始的位置。开始索引和结束索引则与 fill()使用同样的计算方法：

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();

// Copy the contents of ints beginning at index 0 to the values beginning at index 5.
// Stops when it reaches the end of the array either in the source
// indices or the destination indices.
```

```
    ints.copyWithin(5);
    console.log(ints); // [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
    reset();

    // Copy the contents of ints beginning at index 5 to the values beginning at index 0.
    ints.copyWithin(0, 5);
    console.log(ints); // [5, 6, 7, 8, 9, 5, 6, 7, 8, 9]
    reset();

    // Copy the contents of ints beginning at index 0 and ending at index 3 to values
    // beginning at index 4.
    ints.copyWithin(4, 0, 3);
    console.log(ints); // [0, 1, 2, 3, 0, 1, 2, 7, 8, 9]
    reset();

    // The JS engine will perform a full copy of the range of values before inserting,
    // so there is no danger of overwrite during the copy.
    ints.copyWithin(2, 0, 6);
    console.log(ints); // [0, 1, 0, 1, 2, 3, 4, 5, 8, 9]
    reset();

    // Support for negative indexing behaves identically to fill() in that negative
    // indices are calculated relative to the end of the array
    ints.copyWithin(-4, -7, -3);
    console.log(ints); // [0, 1, 2, 3, 4, 5, 3, 4, 5, 6]
    reset();

```

2.2.7 转换方法

所有对象都有 toLocaleString()、toString()和 valueOf()方法。其中，valueOf()返回的还是数组本身。而 toString()返回由数组中每个值的等效字符串拼接而成的一个逗号分隔的字符串。也就是说，对数组的每个值都会调用其 toString()方法，以得到最终的字符串。

```
    let colors = ["red", "green", "blue"];
    alert(colors.toString()); // red,blue,green
    alert(colors.valueOf()); // red,blue,green
    alert(colors); // red,blue,green
```

最后一行代码直接用 alert()显示数组，因为 alert()期待字符串，所以会在后台调用数组的 toString()方法，从而得到跟前面一样的结果。

toLocaleString()方法也可能返回跟 toString()和 valueOf()相同的结果，但也不一定。在调用数组的 toLocaleString()方法时，会得到一个逗号分隔的数组值的字符串。它与另外两个方法唯一的区别是，为了得到最终的字符串，会调用数组每个值的 toLocaleString()方法，而不是toString()方法。看下面的例子：

```
    let person1 = {
    toLocaleString() {
            return "Stephen";
        },
    toString() {
            return "Stephen";
        },
    };
    let person2 = {
    toLocaleString() {
            return "Stephen";
        },
    toString() {
            return "CUI";
        },
    };
    let people = [person1, person2];
    alert(people); // Stephen,CUI
    alert(people.toString()); // Stephen,CUI
    alert(people.toLocaleString()); // Stephen,Stephen
```

继承的方法 toLocaleString()以及 toString()都返回数组值的逗号分隔的字符串。如果想使用不同的分隔符，则可以使用 join()方法。join()方法接收一个参数，即字符串分隔符，返回包含所有项的字符串。

```
    let colors = ["red", "green", "blue"];
    alert(colors.join("-")); // red-green-blue
    alert(colors.join("||")); // red||green||blue
```

如果不给 join()传入任何参数，或者传入 undefined，则仍然使用逗号作为分隔符。

如果数组中某一项是 null 或 undefined，则在 join()、toLocaleString()、toString()和 valueOf()返回的结果中会以空字符串表示。

```
const a = ["red", "green", , "blue"];
console.log(a.toString()); // red,green,,blue
```

2.2.8 栈方法

数组对象可以像栈一样，也就是一种限制插入和删除项的数据结构。栈是一种后进先出（LIFO，Last-In-First-Out）的结构，也就是最近添加的项先被删除。数据项的插入（称为推入，push）和删除（称为弹出，pop）只在栈的一个地方发生，即栈顶。ECMAScript 数组提供了 push()和 pop()方法，以实现类似栈的行为。

push()方法接收任意数量的参数，并将它们添加到数组末尾，返回数组的最新长度。pop()方法则用于删除数组的最后一项，同时减少数组的 length 值，返回被删除的项。

```
const reset = () => (colors = ["red", "green", "blue"]);
reset();
let count = colors.push("red", "green");
console.log(count); // 5

count = colors.push("blue");
console.log(count); // 6

let item = colors.pop();
console.log(item); // blue
console.log(colors.length); // 5
```

2.2.9 队列方法

队列以先进先出（FIFO，First-In-First-Out）形式限制访问。队列在列表末尾添加数据，但从列表开头获取数据。因为有了在数据末尾添加数据的 push()方法，所以要模拟队列就差一个从数组开头取得数据的方法了。这个数组方法叫 shift()，它会删除数组的第一项并返回它，然后数组长度减 1。使用 shift()和 push()，可以把数组当成队列来使用：

```
const reset = () => (colors = ["red", "green", "blue"]);
reset();

let item = colors.shift();
console.log(item); // red
console.log(colors.length); // 2
```

ECMAScript 也为数组提供了 unshift()方法。顾名思义，unshift()就是执行跟 shift()相反的操作：在数组开头添加任意多个值，然后返回新的数组长度。通过使用 unshift()和 pop()，可以在相反方向上模拟队列，即在数组开头添加新数据，在数组末尾取得数据。

```
const reset = () => (colors = ["red", "green", "blue"]);
reset();

let count = colors.unshift("red", "green");
console.log(count); // 5

let item = colors.pop();
console.log(item);
console.log(colors.length);
```

2.2.10 排序方法

数组有两个方法可以用来对元素重新排序：reverse()和 sort()。顾名思义，reverse()方法就是将数组元素反向排列。

```
let values = [1, 2, 6, 2, 5, 3, 6];
values.reverse();
console.log(values); // [6, 3, 5, 2, 6, 2, 1]
```

默认情况下，sort()会按照升序重新排列数组元素，即最小的值在前面，最大的值在后面。但是，sort()会在每一项上调用 String()转型函数，然后比较字符串来决定顺序。即使数组的元素都是数值，也会先把数组转换为字符串再比较、排序。

```
let values = [0, 1, 14, 50, 40, 5];
values.sort();
console.log(values); // [0, 1, 14, 40, 5, 50]
```

为此，sort()方法可以接收一个比较函数，用于判断哪个值应该排在前面。比较函数接收两个参数，如果第一个参数应该排在第二个参数前面，就返回负值；如果两个参数相等，就返回 0；如果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, 14, 50, 40, 5];
    values.sort(compare);
    console.log(values); // [0, 1, 5, 14, 40, 50]
```

这个比较函数还可以简写为一个箭头函数：

```
    let values = [0, 1, 14, 50, 40, 5];
    values.sort((a, b) => (a < b ? -1 : a > b ? 1 : 0));
    console.log(values); // [0, 1, 5, 14, 40, 50]
```

如果数组的元素是数值，或者是其 valueOf()方法返回数值的对象（如 Date 对象），这个比较函数还可以写得更简单，因为这时可以直接用第二个值减去第一个值：

```
    let values = [0, 1, 14, 50, 40, 5];
    values.sort((a, b) => a - b);
    console.log(values); // [0, 1, 5, 14, 40, 50]
```

2.2.11 操作方法

concat()方法可以在现有数组全部元素基础上创建一个新数组。它首先会创建一个当前数组的副本，然后再把它的参数添加到副本末尾，最后返回这个新构建的数组。如果传入一个或多个数组，则 concat()会把这些数组的每一项都添加到结果数组。如果参数不是数组，则直接把它们添加到结果数组末尾。

```
    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']
```

打平数组参数的行为可以重写，方法是在参数数组上指定一个特殊的符号：Symbol.isConcatSpreadable。这个符号能够阻止 concat()打平参数数组。相反，把这个值设置为 true 可以强制打平类数组对象：

```
    let colors = ["red", "green", "blue"];
    let newColors = ["black", "brown"];
    let moreNewColors = {
    [Symbol.isConcatSpreadable]: true,
    length: 2,
    0: "pink",
```

```
    1: "cyan",
    };

    newColors[Symbol.isConcatSpreadable] = false;

    // Force the array to not be flattened
    let colors2 = colors.concat("yellow", newColors);
    // Force the array-like object to be flattened
    let colors3 = colors.concat(moreNewColors);

    console.log(colors); // ['red', 'green', 'blue']
    console.log(colors2); // ['red', 'green', 'blue', 'yellow', Array(2)]
    console.log(colors3); //  ['red', 'green', 'blue', 'pink', 'cyan']
```

方法 slice()用于创建一个包含原有数组中一个或多个元素的新数组。slice()方法可以接收一个或两个参数：返回元素的开始索引和结束索引。如果只有一个参数，则 slice()会返回该索引到数组末尾的所有元素。如果有两个参数，则 slice()返回从开始索引到结束索引对应的所有元素，其中不包含结束索引对应的元素。记住，这个操作不影响原始数组。

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

console.log(colors2); // ['green', 'blue', 'yellow', 'purple']
console.log(colors3); // ['green', 'blue', 'yellow']
```

最强大的数组方法就属 splice()了，使用它的方式可以有很多种。splice()的主要目的是在数组中间插入元素，但有 3 种不同的方式使用这个方法。

- 删除：需要给 splice()传 2 个参数：要删除的第一个元素的位置和要删除的元素数量。可以从数组中删除任意多个元素，比如 splice(0, 2)会删除前两个元素。
- 插入：需要给 splice()传 3 个参数：开始位置、0（要删除的元素数量）和要插入的元素，可以在数组中指定的位置插入元素。第三个参数之后还可以传第四个、第五个参数，乃至任意多个要插入的元素。比如，splice(2, 0, ”red”, ”green”)会从数组位置 2 开始插入字符串”red”和”green”。
- 替换：splice()在删除元素的同时可以在指定位置插入新元素，同样要传入 3 个参数：开始位置、要删除元素的数量和要插入的任意多个元素。要插入的元素数量不一定跟删除的元素数量一致。比如，splice(2, 1, ”red”, ”green”)会在位置 2 删除一个元素，然后从该位置开始向数组中插入”red”和”green”。

splice()方法始终返回这样一个数组，它包含从数组中被删除的元素（如果没有删除元素，则返回空数组）。

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(0, 1);
console.log(colors); // ['green', 'blue']
console.log(removed); // ['red']

removed = colors.splice(1, 0, "yellow", "orange");
console.log(colors); // ['green', 'yellow', 'orange', 'blue']
console.log(removed); // []

removed = colors.splice(1, 1, "red", "purple");
console.log(colors); // ['green', 'red', 'purple', 'orange', 'blue']
console.log(removed); // ['yellow']
```

2.2.12 搜索方法和位置方法

ECMAScript 提供两类搜索数组的方法：按严格相等搜索和按断言函数搜索。

严格相等

ECMAScript 提供了 3 个严格相等的搜索方法：indexOf()、lastIndexOf()和 includes()。其中，前两个方法在所有版本中都可用，而第三个方法是 ECMAScript 7 新增的。这些方法都接收两个参数：要查找的元素和一个可选的起始搜索位置。indexOf()和 includes()方法从数组前头（第一项）开始向后搜索，而 lastIndexOf()从数组末尾（最后一项）开始向前搜索。

indexOf()和 lastIndexOf()都返回要查找的元素在数组中的位置，如果没找到则返回 1。includes()返回布尔值，表示是否至少找到一个与指定元素匹配的项。在比较第一个参数跟数组每一项时，会使用全等（===）比较，也就是说两项必须严格相等。

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
console.log(numbers.indexOf(4)); // 3
console.log(numbers.lastIndexOf(4)); // 5
console.log(numbers.includes(4)); // true

console.log(numbers.indexOf(4, 3)); // 3
console.log(numbers.lastIndexOf(4, 3)); // 3
console.log(numbers.includes(4, 7)); // false

let person = [{ name: "Stephen" }];
let people = [{ name: "Stephen" }];
let morePeople = [person];
```

```
13
14      // person and people are two separate arrays,
15      // even though they contain the same object,
16      // so they are not strictly equal to each other.
17      console.log(people.indexOf(person)); // -1
18      console.log(morePeople.indexOf(person)); // 0
19      console.log(people.includes(person)); // false
20      console.log(morePeople.includes(person)); // true
```

```
1       const c = ["guanli"];
2       const a = [c, "guanli", "b"];
3       const b = a;
4
5       // includes() is used to check if an array contains a specific element,
6       // but b is an array itself and not an element of a.
7       // Therefore, a.includes(b) will always return false.
8       console.log(a.includes(c)); // true
9       console.log(a.includes(b)); // false
10      console.log(a == b); // true
```

断言函数

ECMAScript 也允许按照定义的断言函数搜索数组，每个索引都会调用这个函数。断言函数的返回值决定了相应索引的元素是否被认为匹配。

断言函数接收 3 个参数：元素、索引和数组本身。其中元素是数组中当前搜索的元素（可以理解为项），索引是当前元素的索引，而数组就是正在搜索的数组。断言函数返回真值，表示是否匹配。

find()和 findIndex()方法使用了断言函数。这两个方法都从数组的最小索引开始。find()返回第一个匹配的元素，findIndex()返回第一个匹配元素的索引。这两个方法也都接收第二个可选的参数，用于指定断言函数内部 this 的值。

```
1       const people = [
2       {
3               name: "Matt",
4               age: 23,
5           },
6       {
7               name: "Stephen",
8               age: 27,
9           },
```

```
    ];

    // console.log(people.find((element, index, array) => element.age < 28));
    console.log(people.find((element) => element.age > 23)); // {name: 'Stephen', age: 27}

    // console.log(people.findIndex((element, index, array) => element.age < 28));
    console.log(people.findIndex((element) => element.age > 23)); // 1
```

找到匹配项后，这两个方法都不再继续搜索。

```
const evens = [2, 4, 6];
// Element of array will never be inspected after match is found
evens.find((element, index, array) => {
console.log(element);
console.log(index);
console.log(array);
return element === 4;
});
```

2.2.13 迭代方法

2.2.14 归并方法

2.3 定型数组

2.4 Map

2.5 WeakMap

2.6 Set

2.7 WeakSet

2.8 迭代与拓展操作

2.9 小结