Javascript Introduction

1. Keywords

Keyword	Description
break	Terminates a switch or a loop
continue	Jumps out of a loop and starts at the top
debugger	Stops the execution of JavaScript, and calls (if available) the debugging function
do ... while	Executes a block of statements, and repeats the block, while a condition is true
for	Marks a block of statements to be executed, as long as a condition is true
function	Declares a function
if ... else	Marks a block of statements to be executed, depending on a condition
return	Exits a function
switch	Marks a block of statements to be executed, depending on different cases
try ... catch	Implements error handling to a block of statements
var	Declares a variable

2. Statements

JavaScript 程序的执行单位为行（line），也就是一行一行地执行。一般情况下，每一行就是一个语句。

```
var a, b, c;     // Declare 3 variables
a = 5;          // Assign the value 5 to a
b = 6;          // Assign the value 6 to b
c = a + b;       // Assign the sum of a and b to c
```

2.a Semicolons

语句以分号结尾，一个分号就表示一个语句结束。多个语句可以写在一行内。

```
var a, b, c; a = 5; b = 6; c = a + b;
```

2.b White Space

语句中的额外空格会被忽略。

```
var person = "Alice";
var person="Alice";
```

3. Variable

变量是对“值”的具名引用。变量就是为“值”起名，然后引用这个名字，就等同于引用这个值。变量的名字就是变量名。

```
var x = 5;
var y = 6;
var z = x + y;
```

JavaScript 的变量名区分大小写，`Name` 和 `name` 是两个不同的变量。

```
var name = 'Alice';
var Name = 'Bob';
```

3.a Types

- 数值（number）：整数和小数（比如1和3.14）
- 字符串（string）：文本（比如Hello World）。
- 布尔值（boolean）：表示真伪的两个特殊值，即true（真）和false（假）
- undefined：表示“未定义”或不存在，即由于目前没有定义，所以此处暂时没有任何值
- null：表示空值，即此处的值为空。
- 对象（object）：各种值组成的集合
- function?
- array?

3.b Hoisting (变量提升)

从概念的字面意义上说，“变量提升”意味着变量和函数的声明会在物理层面移动到代码的最前面，但这么说并不准确。实际上变量和函数声明在代码里的位置是不会动的，而是在编译阶段被放入内存中。

```
catName("Tigger"); // "我的猫名叫 Tigger"

function catName(name) {
    console.log("我的猫名叫 " + name);
}
```

只有声明被提升

```
console.log(num); // Returns undefined
var num;
num = 6;
```

3.c Identifier

- 第一个字符，可以是任意 Unicode 字母（包括英文字母和其他语言的字母），以及美元符号（$）和下划线（_）。
- 第二个字符及后面的字符，除了 Unicode 字母、美元符号和下划线，还可以用数字0-9。

```
arg0
_tmp
$elem
π

1a  // 第一个字符不能是数字
23  // 同上
***  // 标识符不能包含星号
a+b  // 标识符不能包含加号
-d  // 标识符不能包含减号或连词线

var 临时变量 = 1; // 中文是合法的标识符，可以用作变量名。
```

4. 条件语句

4.a if...else

```
if (x = 2) { // 不报错
if (2 = x) { // 报错
```

```
// else代码块总是与离自己最近的那个if语句配对。

var m = 1;
var n = 2;

if (m !== 1)
if (n === 2) console.log('hello');
else console.log('world');
```

```
if (m !== 1) {
  if (n === 2) {
    console.log('hello');
  } else {
    console.log('world');
  }
}
```

4.b Comparison Operators

`===` 和 `==` (严格)相等

```
3 === 3   // true
3 === '3' // false
3 == '3' // true
```

Object

```
{} == {} // false
{} === {} // false
```

4.c Switch

多个if...else连在一起使用的时候，可以转为使用更方便的switch结构。

```
switch (fruit) {
  case "banana":
    // ...
    break;
  case "apple":
    // ...
    break;
  default:
    // ...
}
```

4.d Conditional Operator

`(条件) ? 表达式_左 : 表达式_右`

```
var even = (n % 2 === 0) ? true : false;

var even;
if (n % 2 === 0) {
  even = true;
} else {
  even = false;
}
```

4.e 短路计算

```
const a = true;

if (a || b) {
  console.log('HERE');
}

const a = false;

if (a && b) {
  console.log('HERE');
}

程序员出去摆摊

赚 1000

if (出门捡到一万块钱 || 赚到了1000块钱 || 到了凌晨3点 || 老婆喊我回家) {
  回家();
}

if (没有看到老同学 && &&) {
  摆摊();
}


// ...
```

由于逻辑表达式的运算顺序是从左到右，也可以用以下规则进行"短路"计算：

- (some falsy expression) && (expr) 短路计算的结果为假。
- (some truthy expression) || (expr) 短路计算的结果为真。

```
显示Dropdown // true

显示Dropdown && (<div>My Dropdown</div>)
```

```
a4 = true  && "Dog"     // t && t 返回 "Dog"
a5 = "Cat" && "Dog"     // t && t 返回 "Dog"
a6 = false && "Cat"     // f && t 返回 false
a7 = "Cat" && false     // t && f 返回 false
a8 = ''    && false     // f && f 返回 ""
a9 = false && ''        // f && f 返回 false

o5 = "Cat" || "Dog"     // t || t 返回 "Cat"
o6 = false || "Cat"     // f || t 返回 "Cat"
o7 = "Cat" || false     // t || f 返回 "Cat"
o8 = ''    || false     // f || f 返回 false
o9 = false || ''        // f || f 返回 ""
```

5. 循环语句

5.a while

5.b do...while

5.c for

5.d break / continue

`break` 语句用于跳出代码块或循环。

`continue` 语句用于立即终止本轮循环。

```
for (var i = 0; i < 10; i ++) {
  if (i === 5) {
    break;
  }
  console.log(i);
}
// 0 1 2 3 4

for (var i = 0; i < 10; i ++) {
  if (i === 5) {
    continue;
  }
  console.log(i);
}
// 0 1 2 3 4 6 7 8 9
```

6. Function

一般来说，一个函数是可以通过代码调用的一个“子程序”（或在递归的情况下由函数调用）。

像程序本身一样，一个函数由称为函数体的一系列语句组成。值可以传递给一个函数，函数将返回一个值。

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}

function sum(a, b) {
  return a + b;
}

var c = sum(1, 2) // 3

function greeting(name) {
  console.log('Hello: ' + name);
}

var d = greeting('Alice') // c undefined, Hello: Alice
```

6.a 参数的省略

函数参数不是必需的，Javascript 允许省略参数。

```
function f(a, b) {
  return a;
}

f(1, 2, 3) // 1
f(1) // 1
f() // undefined
```

6.b 匿名函数

```
function myFunction() {
  console.log(a.b);
}

(function() {
  console.log(a.b);
})()

// 命名函数表达式的好处是当我们遇到错误时，堆栈跟踪会显示函数名，容易寻找错误。
var myFunction = function namedFunction(){
  console.log(a.b);
}
```

7. 课上作业

1. Flight Stops

given by an array of flights, returns stops statement to user.

ex.

- flights: `[{ origin: 'MEL', destination: 'CAN' }]` -> 'Direct'
- flights: `[{ origin: 'MEL', destination: 'CAN' }, { origin: 'CAN', destination: 'PVG' }]` -> '1 stop'
- flights: `[{ origin: 'MEL', destination: 'HKG' }, { HKG - CAN }, {}, { origin: 'CAN', destination: 'PVG' }]` -> '3 stops'
- flights: `[{ origin: 'MEL', destination: 'HKG' }, { HKG - CAN }, { CAN - SNG }, { origin: 'CAN', destination: 'PVG' }]` -> 'n stops'

```
const flights = [{ origin: 'MEL', destination: 'CAN' }];
getStops(flights) // 'Direct'
```

arr.length 1 -> Direct 2 -> 1 stop 3 -> 2 stops 4 -> 3 stops 5 -> 4 stops n -> ... stops

Qantas 环游世界航班 Around The World

- flights: 76个航班

getStops(flights) -> // 75 stops x // Dream Line: ATW

88 -> // Rich Run

```
// Oskar A
const getStops = (numberOfFlights) => numberOfFlights.length===1?
  console.log(`Direct`)
  :numberOfFlights.length===76?
  console.log(`Dream Line: ATW`):console.log(`${numberOfFlights.length - 1} stops`);

// Jessie B
const getstop=(array)=>{
  if (array.length===1){
    return 'direct';
  }  else if(array.length>1 && array.length<76){
    return `${array.length-1} stops`;
  }  else if (array.length===76){
    return 'dreamline';
  }  else if (array.length > 76 && array.length < 88) {
    return 'Rich run'
  }

  return "";
};
```

```
function getStops(flights) {
  const specialMap = {
    1: 'Direct',
    2: '1 stop',
    34: 'World Trip',
  };

  const legs = flights.length;

  const specialMessage = specialMap[legs];

  if (specialMessage) {
    return specialMessage;
  }

  return `${legs - 1} stops`;

  return {
    1: 'Direct',
    2: '1 stop',
    34: 'World Trip',
  }[flights.length] || `${flights.length - 1} stops`;
}

getStops() => 'Direct';
getStops() => '1 stop';
getStops() => 'n stops';
```

2. Tax Calculation

Income thresholds	Rate	Tax payable on this income
$0 – $18,200	0%	Nil
$18,201 – $37,000	19%	19c for each $1 over $18,200
$37,001 – $90,000	32.5%	$3,572 plus 32.5% of amounts over $37,000
$90,001 – $180,000	37%	$20,797 plus 37% of amounts over $90,000
$180,001 and over	45%	$54,096 plus 45% of amounts over $180,000

calTax(150000);

所谓的 if else 就是经验不足用的笨方法

笨方法 笨方法 笨方法 笨方法 笨方法 笨方法 笨方法

可读性 可维护性 可复用性、

这里有问题啊龙哥 这里的 '人生经验' 就是具体化成 method 所谓的 if else 就是经验不足用的笨方法

```
// 90000 - 180000
// -> 90000 - 120000
// -> 120000 - 180000

// 初中毕业 X
// 大学毕业 V

function calTax(income) {
  let tax = 0;
  if (income > 0 && income <= 18200) {
    return tax = 'Nil';
  } else if (income >= 18201 && income <= 37000) {
    return tax = (income - 18200) * 0.19;
  } else if (income >= 37001 && income <= 90000) {
    return tax = 3572 + (income - 37000) * 0.325;
  } else if (income >= 90001 && income <= 180000) {
    return tax = 20797 + (income - 90000) * 0.37;
  } else {
    return tax = 54096 + (income - 180000) * 0.45;
  }
};
```

```
// 90000 - 180000
// -> 90000 - 120000
// -> 120000 - 180000

// 2020FY
// 2021FY -> Tax Change
const getMyIncomeTax2 = (income) => {
  // SOLID
  // SOD
  // Single Responsibility  单一职责
  // Open / Close           开关原则
  // Dependencies injection 依赖注入

  // 不要动这段代码，我也不知道为什么可以工作
  const taxTable = [
    { min: 0, max: 18200, accumulate: 0, rate: 0 }, { min: 18200, max: 37000, accumulate: 0, rate: 0.19 },
    { min: 37000, max: 90000, accumulate: 3572, rate: 0.325 },
    { min: 9000, max: 120000, accumulate: 20797, rate: 0.37 }
    { min: 120000, max: 180000, accumulate: 22311, rate: 0.39 }
    {
      min: 180000,
      max: Number.POSITIVE_INFINITY,
      accumulate: 54097,
      rate: 0.45,
    },
  ];
  for (let i = 0; i < taxTable.length; i++) {
    if (income <= taxTable[i].max)
      return (
        taxTable[i].accumulate + (income - taxTable[i].min) * taxTable[i].rate
      );
  }
};
```

```
const TAX_TABLE_2020 = [{
  min: 0,
  max: 18200,
  floor: 0,
  base: 0,
  rate: 0,
}, {
  min: 18201,
  max: 37000,
  floor: 18200,
  base: 0,
  rate: 0.19
}, {
  min: 37001,
  max: 90000,
  floor: 37000,
  base: 3572,
  rate: 0.325
}, {
  min: 90001,
  max: 180000,
  floor: 90000,
  base: 20797,
  rate: 0.37
}, {
  min: 180001,
  max: Number.POSITIVE_INFINITY,
  floor: 180000,
  base: 54096,
  rate: 0.45
}];

const calculateTax = (taxTable, income) => {
  // const taxTable = [];

  let row;

  for(let i = 0; i < taxTable.length; i ++) {
    if (income > taxTable[i].min && income <= taxTable[i].max) {
      row = taxTable[i];

      break;
    }
  }

  const { floor, base, rate } = row;

  return (income - floor) * rate + base;
}

// 2020

calculateTax(TAX_TABLE_2020, 120000);
calculateTax(TAX_TABLE_2021, 120000);

// const calculateTax = (taxTable, income) => {
//   const { floor, base, rate } = taxTable.find((row) => income > row.min && income <= row.max);

//   return (income - floor) * rate + base;
// }
```

8. Readable, Maintainable, Reusable

Readable, Maintainable, Reusable Readable, Maintainable, Reusable Readable, Maintainable, Reusable Readable, Maintainable, Reusable Readable, Maintainable,

9. Object Oriented Programing

没有对象的 JavaScript

Everything is `Object` ;

Person { name: string, greeting: Function, }

```
function createNewPerson(name) {
  var obj = {};

  obj.name = name;
  obj.greeting = function () {
    console.log('Hi! I\'m ' + this.name + '.');
  }

  return obj;
}

// Low 版本的 JS OOP
var alice = createNewPerson('Alice');
alice.greeting();
// Hi! I'm Alice.
```

如果我们知道如何创建一个对象，就没有必要创建一个新的空对象并且返回它。

```
Class Person {
  constructor(name) {
    this.name = name
  }

  greeting() {
    system.out.print('Hi! I\'m ' + this.name + '.');
  }
}
```

```
function Person(name) {
  this.name = name;

  this.greeting = function() {
    alert('Hi! I\'m ' + this.name + '.');
  };
}

// var alice = Person('Alice');
var alice = new Person('Alice');
alice.greeting();
```

9.a prototype

```
var alice = new Person('Alice');
var bob = new Person('Bob');

console.log(alice.greeting === bob.greeting) // false
```

```
// 构造函数
function Person(name) {
  this.name = name;
}

// method
Person.prototype.greeting = function() {
  alert('Hi! I\'m ' + this.name + '.');
};

const alice = new Person('alice');
alice.greeting();
```

10. 一等公民

Pure function is the one and only first-class citizen 函数是一等公民 函数是普通老百姓

- 可以在程序执行时动态创建函数
- 可以将函数赋值给变量
- 可以将函数作为参数传给例外一个函数
- 可以作为返回值返回

```
function greeting(name) {
  function sayHi(s) {
    alert(s + ' ' + name);
  }

  return sayHi;
};

// var myGreeting = greeting;

var sayHiToAlice = greeting('Alice');
console.log(sayHiToAlice);

sayHiToAlice('Hello');
// alert Hello Alice

var mySayHi = greeting('Bob');
setTimeout(mySayHi, 1000);
// 1s -> alert Hello Bob
```

11. Scope

执行上下文 - The current context of execution.

在函数外的声明，将会存在于函数外的全局作用域

```
var b = 'bar';
function foo() {
  console.log(b);
}

foo(); // bar
```

在函数内的声明，将会存在与函数的私有作用域

```
function foo() {
  var a = 'bar';
  console.log(a);
}
foo(); // bar
```

平级私有作用域互不相通

```
function bar() {
  var b = 'bar';
}
function foo() {
  console.log(b);
}
foo(); // Uncaught ReferenceError: b is not defined
```

作用域自上而下传递

```
function foo() {
  var b = 'bar';
}
console.log(b); // Uncaught ReferenceError: b is not defined
```

在执行一段函数的时候，当读取变量时，先“就近”在函数私有作用域寻找该变量的声明和赋值。如果无法找到，则到更上层作用域去寻找。这里的“更上层作用域”可能也是一个函数私有作用域或者全局作用域。

```
function bar() {
  var a = 'bar';

  function foo() {
    console.log(a);
  }

  return foo;
}

bar()(); // bar
```

- 函数作用域
- 函数是一等公民

```
function x() {
  var xx = 10;
  return function(xz) {
    var xy = 20;
    console.log(xx + xy + xz);
  }
}

var a = x(); // =>

var a = function(xz) {
  var xy = 20;
  console.log(10 + xy + xz);
}

a(30); // =>
60;

x()(30);
```

12. 闭包

函数和对其周围状态（lexical environment，词法环境（*作用域））的引用捆绑在一起构成闭包（closure）

```
function createCounter() {
  var privateCounter = 0;

  function increment() {
    privateCounter += 1;
  }

  function decrement() {
    privateCounter -= 1;
  }

  function print() {
    console.log('Counter: ' + privateCounter);
  }

  var counter = {
    increment: increment,
    decrement: decrement,
    print: print,
  };

  return counter;
}

var Counter = createCounter();
Counter.increment();
Counter.increment();
Counter.decrement();
Counter.print() // Counter: 1
```

13. this

```
function Counter() {
  this.privateCounter = 0;
}

Counter.prototype.increment = function() {
  this.privateCounter += 1;
}

Counter.prototype.decrement = function() {
  this.privateCounter -= 1;
}

Counter.prototype.print = function() {
  console.log('Counter: ' + this.privateCounter);
}

var counter = new Counter();
counter.increment();
counter.increment();
counter.decrement();
counter.print(); // Counter: 1

setTimeout(counter.print, 1000); // Counter: undefined
```

this 的指向是在调用时确定的。

用大白话来讲，“谁调用，指向谁。”

this 的指向，是在调用函数时根据上下文所动态确定的

具体环节和规则，可以先“死记硬背”以下几条规律。

- 在函数体中，简单调用该函数时，严格模式下 this 绑定到 undefined，否则绑定到全局对象 window (browser) / global (node)；
- 一般构造函数 new 调用，绑定到新创建的对象上；

- 一般由上下文对象调用，绑定在该对象上；
- 一般由 call/apply/bind 方法显式调用，绑定到指定参数的对象上；

```
function Person(name) {
  this.name = name;
}

Person.prototype.getName = function() {
  return this.name;
}

var alice = new Person('Alice');
var bob = new Person('Bob');

alice.brother = bob;

console.log(alice.brother.getName());

var myBrother = alice.brother;
console.log(myBrother.getName());

var getMyBrotherName = alice.brother.getName;
console.log(getMyBrotherName());
```

```
var alice = {
  name: 'Alice',
  brother: {
    name: 'Bob',
    getName: function() {
      return this.name;
    },
  },
};

alice.getName = alice.brother.getName;
alice.getName()
```

13.a 面试题

```
var text = 'o0';

var o1 = {
  text: 'o1',
  fn: function() {
    return this.text;
  },
};

var o2 = {
  text: 'o2',
  fn: function() {
    return o1.fn();
  }
};

var o3 = {
  text: 'o3',
  fn: function() {
    o2.fn = o1.fn;
    o2.fn = function() {
      return this.text;
    }
    return o2.fn(); // o2
  },
};

console.log(o1.fn());
console.log(o2.fn());
console.log(o3.fn());
```

14. Vanilla Javascript

Vanilla JS is a fast, lightweight, cross-platform framework for building incredible, powerful JavaScript applications.

http://vanilla-js.com

Vanilla JS 基于 Native JS 对浏览器APP 进行开发 Node JS 基于 Native JS 对服务器APP 今习性开发

```
var a = 1;
var b = 2;

console.log(a + b); // 3
// console 是 Native JS 提供的 API

document.getElementById('my-heading'); // Vanilla JS
// document 是 Vanilla JS 提供的 API

window.scrollTo(100);
// window 是 Vanilla JS 提供的 API



document.getElementsByClassName('my-list-item');

// DOM -> 对 DOM element 的操作

document.getElementById('btn').classList.add('active'); // Vanilla JS 作为 lib 提供了使用 JavaScript 操作 DOM 的 api

$('#my-heading').class.add('active'); // jQuery 作为 lib 提供了 api，帮我们简化了一些操作
```

15. ES6

ECMAScript

Version (版本)

每个新版本都是新升级

- 加入新 API
- 修正一些 Bug
- 提升一下性能

Java 15 编译型语言 编译 compile (version) -> 执行文件 -> 任何一个环境

Python 3.9 C# 9.0 php 7 .net 4.8.0

JS 是什么版本？

JavaScript 解释型语言（ 翻译型语言 ） code -> 用户浏览器上执行

不同浏览器，对代码的解释方式解释手段，都是不同的

一份中文文件 -> ? 翻译 -> 澳洲 NAATI -> 翻译人员 -> 使用自己的标准来培训 一份中文文件 -> 接受过 NAATI 培训的翻译人员来翻译 -> 澳洲

一份JS代码 -> ? 翻译 -> 用户使用 一份JS代码 -> 解释 -> 用户使用 一份JS代码 -> 浏览器解释 -> 用户使用

ECMA组织 -> 定义标准 ECMAScript n -> ES n - ES4 - ES4.6 - ES4.9 - ES New - ES5 (ECMA 重组，制定了一系列规范，成立公司) - ES6 * - ES2018 - ES7 - ES8 - ES2019 - ES9

一份JS代码 (符合 ES6 标准的代码) -> 浏览器根据 ES6 定义的标准解释 -> 用户使用

不同的浏览器解释出来的结果是否相同？

解释型，弱类型语言

-> 编译型Native JS -> 强类型

作用域，原形链，this，pure function，闭包

JavaScript 从入门到放弃 入门 -> !放弃 -> 精通

- Chrome
- FF
- Safari
- IE (Edge)

浏览器兼容性

```简历 Skills:

JavaScript(ES6, Vanilla), CSS3, HTML5, React
```

ES6之前生不如死

15.a Arrow Function

ES6 允许使用“箭头”(=>)定义函数。

```js
function sum(num1, num2) {
  return num1 + num2;
}

// 函数是一等公民
// Pure function is the one and only first-class citizen
// 函数是普通老百姓
// 函数和变量没有任何区别
// 函数也可以作为变量的值储存，也可以作为参数传递，也可以作为返回值返回
var sum = function(num1, num2) {
  return num1 + num2;
};

var sum = (num1, num2) => {
  return num1 + num2;
};

var sum = (num1, num2) => num1 + num2;

var sum = (num1, num2) => {
  console.log('My sum');
  return num1 + num2;
}

sum // 存储好的函数
sum(1, 2) // 调用函数

// 为这个变量赋值一个函数
// Function is first class citizen
// 函数是一等公民
// 函数是普通老百姓(误)

// this?
写函数的时候统一使用箭头函数
```

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用return语句返回。

由于大括号被解释为代码块，所以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。

函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。

```
var Person = function(name) {
  this.name = name;

  this.getName = () => this.name;
  // this.getName = function() {
  //   return this.name;
  // }
}

// Person.propType.getName = () => this.name;

var alice = new Person('Alice');
console.log(alice.getName());
var bob = new Person('Bob');
bob.getName = alice.getName;
console.log(bob.getName());
```

不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。

15.b Class

JavaScript构造对象的写法跟传统的面向对象语言（比如 C++ 和 Java）差异很大，很容易让新学习这门语言的程序员感到困惑。

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过class关键字，可以定义类。

语法糖!!!

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

  sayHi() {
    console.log('My name is ' + this.name);
  }

  joinMeeting(meeting) {
    meeting.talks.push(this.sayHi);
  }
}

class Meeting {
  constructor() {
    this.talkers = [];
  }

  start() {
    function talk(t) {
      t();
    }

    this.talks.forEach(talk);
  }
}

var alice = new Person('Alice');
var bob = new Person('Bob');

var standup = new Meeting();

alice.joinMeeting(standup);
bob.joinMeeting(standup);

standup.start();
```

15.c let/const

ES6 新增了let/const命令，用来声明变量。它的用法类似于var，但是所声明的变量，只在let/const命令所在的代码块内有效。

JS 为了实现效率最大化 提升 (Scan -> 尝试纠错) -> 执行

let/const
need to declare then use
const = final
let can change its value later

1. 变量提升

```
b = 1;
a = b + 2;

let a;
let b;
```

15.d 块级作用域

变量 i 只用来控制循环，但是循环结束后，它并没有消失，泄露成了全局变量。

```
var fns = [];

var i;

for (i = 0; i < 10; i ++) {
  //
  fns[i] = function() {
    console.log(i);
  }
  //
}

var i = 10;

// 0
fns[0] = function() {
  console.log(i);
}

// 1
fns[1] = function() {
  console.log(i);
}

// 2
fns[2] = function() {
  console.log(i);
}

// 3
fns[2] = function() {
  console.log(i);
}

...

// 10
fns[10] = function() {
  console.log(i);
}

function() {
  console.log(i);
}

fns[0](); // 10
```

我们可以使用立即执行函数创建一个函数作用域

```
var fns = [];

for (var i = 0; i < 10; i ++) {
  (function(i) {
    fns[i] = function() {
      console.log(i);
    }
  })(i)
}

fns[0]();
```

块语句（或其他语言的复合语句）用于组合零个或多个语句。该块由一对大括号界定。

```
var i = 0;
function a() {
  console.log(i);
}
{
  var i = 1;
  function b() {
    console.log(i);
  }
}
{
  var i = 2;
  function c() {
    console.log(i);
  }
}
```

let/const 创建块级作用域

```
var fns = [];

var i;
for (i = 0; i < 10; i ++) {
  fns[i] = function() {
    console.log(i);
  }
}

for (let i = 0; i < 10; i ++) {
  fns[i] = function() {
    console.log(i);
  }
}

// 0
let i0 = 0;
fns[0] = function() {
  console.log(i0);
}

// 1
let i1 = 0;
fns[1] = function() {
  console.log(i1);
}

fns[0]();
```

let vs const?

var 命令会发生“变量提升”现象，即变量可以在声明之前使用，值为undefined。

```
console.log(foo); // undefined
var foo = 2;

console.log(bar); // ReferenceError
let bar = 2;
```

```
let foo;    // 变量
const bar;  // 常量

let result;

result = a + 1;
result = result + 1;

const string = 'Greeting';
```

15.e Template String 模板字符串

这些字符串操作在 JavaScript 语言是非常繁琐的

- 嵌入变量
- 换行

```
// '' 占用
// "" 占用

const s = 'Hello, this is 'ABC'';
const s = "Hello, this is 'ABC'";
// Hello, this is 'ABC'
const s = 'Hello, this is "ABC"';
// Hello, this is "ABC"

// RMR
const greeting = 'Hello, my name is ' + name + ' , ' + age + ' years old.';
// Hello, my name isAlice,28years old.
// Hello, my name is Alice, 28 years old.

// `反引号` 1左面的那个键
const greeting = `Hello ${name}`;
// Hello Alice

const multiLinesGreeting = 'Hello,' +
  'My name is ' + name + '.' +
  'Nice to meet you.';

const multiLinesGreeting = `
Hello,
My Name is ${name}.
Nice to meet you.
`;
```

``` used to string
${ } variable

### 15.f Tagged Template 标签模版

模版字符串可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串

```
alert`hello`
// 等同于
alert(['hello'])
```
```

```
let a = 5;
let b = 10;

tag`Hello ${ a + b } world ${ a * b }`;
tag(['Hello ', ' world ', ''], 15, 50);
```

15.g 函数参数的默认值

ES6 之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

```
const hello = (name) => {
  if (!name) {
    name = 'World';
  }

  console.log(`Hello ${name}`);
}

hello() // Hello World
hello('Alice') // Hello Alice

// RMR
// 能不写 if...else 就不写 if...else
// 能不写 if...else 就不写 if...else
// 能不写 if...else 就不写 if...else

const hello = (name = 'World') => {
  console.log(`Hello ${name}`);
}
```

default value of name is 'World'

```
const m1 = ({x = 0, y = 0} = {}) => console.log(x, y);

const m2 = ({x, y} = { x: 0, y: 0 }) => console.log(x, y);
```

15.h Destructuring 解构赋值

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）。

程序员应该是非常懒的 能写 => 就不写 function 能不写 return 就不写 return jQuery `$`

```
const student = {
  name: 'Alice',
  age: 26,
  courses: [{
    name: 'Introduction to JavaScript',
  }, {
    name: 'How to give up JavaScript',
  }],
};

// 在 ctrl c / cmd c 上面放一个报警装置
// 一旦 copy paste 就一定有问题
const name = student.name;
const age = student.age;
const courses = student.courses;

// 我是一个非常懒的程序员

// 我一开始以为解构就自己随便命名几个变量名去接object里面的属性变量  -> 有自我思考能力的AI

// 结构, object -> 解析结构 {}
const {
  name: studentName,
  age,
```

```
  courses: [{ name: introName }, { name: giveUpName }]
} = student;
console.log(studentName);
console.log(age);
console.log(introName); // 'Introduction to JavaScript'
console.log(giveUpName); // How to give up JavaScript

// {
//   name: 'How to give up JavaScript',
// }

const { name } = giveUp;
console.log(name); // How to give up JavaScript


const items = [1, 2, 3];

const x = arr[0];
const y = arr[1];
const z = arr[2];

// 结构, array -> 解构 [];
const [x, y, z, w] = items;
w -> items[3];
console.log(w) // undefined;

// 解构赋值, Destruction

const student = {
  name: 'Alice',
  age: 26,
  address: '1 Melbourne St, Sydney',
};

// 语法糖
const { name, ...studentB } = student;
console.log(name) // Alice
console.log(studentB) //


回国
// 绿码, 登机牌, 核酸检测, 行李牌
-> const { 绿码, ...其余的A } = 人;
check(绿码);

-> const { 登机牌, ...其余的B } = 人;
scan(登机牌)

// {
//   age,
//   address,
// }







const { ...rest, name } = student;
console.log(name) // Alice
console.log(rest) // { ... } 除去 name
// {
//   age: 26,
//   courses: [{
```

```
//     name: 'Introduction to JavaScript',
//   }, {
//     name: 'How to give up JavaScript',
//   }],
// };

const { enrolment } = student;
// const enrolment = student.enrolment;
// undefined

const name = student.name;
const age = student.age;
const courses = student.courses;

const name1 = student.name;

const { name: name1, age: foo, courses } = student;
console.log(name1) // Alice
console.log(foo) // 26

// 但是变量的数量必须和array的数量相等吧
const arr = ['a', 'b', 'c'];
const [v1, ...arr1] = arr;
const [v1, v2, v3, v4] = arr;
// const v4 = arr[3];
console.log(v3);
// c
console.log(v4);
// undefined



console.log(courses)

[{
  name: 'Introduction to JavaScript',
}, {
  name: 'How to give up JavaScript',
}]

const intro = courses[0];
const giveUp = courses[1];
const [intro, giveUp] = courses;
const [{ name }, giveUp] = courses;
console.log(name);
// ? Introduction to JavaScript

const student = {
  name: 'Alice',
  age: 26,
  courses: [{
    name: 'Introduction to JavaScript',
  }, {
    name: 'How to give up JavaScript',
  }],
};

const { name: studentName, age, courses: [ intro, { name: courseName } ] } = student;
// const { name: studentName, age, courses: [ intro, course2 ] } = student;
// const { name: courseName } = course2;
console.log(courseName);
// How to give up JavaScript


console.log(intro);
console.log(courseName);
```

```
// const enrolment = student.enrolment || 'Unknown';

const { name, age, enrolment = 'Unknown' } = student;
const { courses } = student;
const [{ name: courseOneName }] = course;
```

15.i 浅拷贝

在解构赋值中，可以使用 ... 进行浅拷贝。

```
const student = {
  name: 'Alice',
  age: 26,
  address: 'Melbourne',
  courses: [{
    name: 'Introduction to JavaScript',
  }, {
    name: 'How to give up JavaScript',
  }]
};

const { name, ...rest } = student;
console.log(name);
console.log(reset);

const [v1, ...arr] = [1, 2, 3];
console.log(v1);
console.log(arr);
```

16. Reference or Value

- 数值（number）：整数和小数（比如1和3.14）
- 字符串（string）：文本（比如Hello World）。
- 布尔值（boolean）：表示真伪的两个特殊值，即true（真）和false（假）
- undefined：表示“未定义”或不存在，即由于目前没有定义，所以此处暂时没有任何值
- null：表示空值，即此处的值为空。
- 对象（object）：各种值组成的集合

JavaScript 在传递参数的时候，除了对象，都是拷贝值传递。

```
const foo = (a) => {
  a = false;

  return a;
};

const a = 'Alice';
let b = a;
b = 'Bob';
const c = foo(a);

console.log(a);
console.log(b);
console.log(c);
```

对象也是拷贝值传递

```
const foo = (a) => {
  a = { name: 'Tifa' };
  return a;
};

const a = { name: 'Alice' };
let b = a;
b = { name: 'Bob' };
const c = foo(a);

console.log(a);
console.log(b);
console.log(c);
```

但是，对象的按值传递，是指变量地址的值。

```
const foo = (a) => {
  a.name = 'Tifa';
  return a;
};

const a = { name: 'Alice' };
const b = a;
b.name = 'Bob';
const c = foo(a);

console.log(a);
console.log(b);
console.log(c);
```

17. Immutable

那么问题来了

```
const arr1 = ['Apple', 'Banana'];

const getFruit = (arr) => {
  const fruit = arr.pop();
  return fruit;
};

const fruit = getFruit(arr1);
console.log(fruit);
console.log(arr1);
```

```
const getFirstNTasks = (n, tasks) => tasks.splice(0, n)

const student = {
  name: 'Alice',
  score: 60,
};

const s1 = getStudentWithFormatScore(student);
const s2 = getStudentWith...(student);
const sn = n...(student);

console.log(student);
```

17.a 通过拷贝实现

在 ES6 之前

```
const arr1 = ['Apple', 'Banana'];

const getFruit = (arr) => {
  const fruit = Array.from(arr).pop();
  return fruit;
};

const fruit = getFruit(arr1);
console.log(fruit);
console.log(arr1);
```

```
const getStudentWithFormatScore = (student) => {
  const newStudent = Object.assign({}, student);

  newStudent.score = ...
};
```

通过解构赋值进行浅拷贝

```
const [...arr2] = arr1;
const {...obj2} = obj1;
```

18. 异步 & Call API

18.a Callback

等待用户点击按钮就是一个异步编程

button.onclick = () => {}

我们将方法作为参数传递出去，方法当下不会直接执行，而是会在特定的情况下被执行。

```
function handleClick = () => {
  console.log('clicked');
}

button.onclick = handleClick;
button.classList.add('btn');
button.innerText = 'Submit';

console.log('button did render');
```

现实生活的例子

- 同步

买咖啡 -> 上班

同步的世界效率很低，因为我们在买咖啡到上班这个过程是同步的，我们没有办法去做任何其他事情。

- 异步

买咖啡 -> 玩手机 买好咖啡 -> 上班

异步的世界我们可以做的事情就非常多了。

Callback 其实已经解决了绝大多数异步问题。

```
起床();
洗漱();
出门(() => {
  刷抖音();
  买咖啡(() => {
    停止刷抖音();
    刷朋友圈();
    买早点(() => {
      吃早点();
      前往公司(() => {
        停止听歌();
```

```
        停止刷朋友圈();
        喝咖啡();
        开早会(() => {
          听歌();
          写代码(() => {
            停止听歌();
            约饭局(() => {
              刷抖音();
              买午饭(() => {
                吃午饭(() => {
                  学习();
                  开会(() => {
                    停止学习();
                    买咖啡(() => {
                      写代码(() => {
                        // ...
                      });
                    })
                  });
                });
              });
            });
          });
        });
      });
    });
  });
});

// JS 函数是一等公民
console.log('1');

loadFile(fileToLoad, (file) => {
  // 当读取结束后执行这个 function
  console.log('2');

  loadFile(anotherFile, (fileB) => {
    // 当读取结束后执行这个 function
    console.log('3');
  });

  console.log('4');
});

console.log('5');

// 15 等待1s 2 ASYNC 4 等待1s 3
// CPU Clock
// Register Async

// JS 本质是单线程的
// Event Loop
// Single Thread

const loadFile(fileName, onLoaded) {
  // ... 调用 IO -> file
  // 需要 1s 执行
  onLoaded(file);
}


// 有一个问题 假设callback的函数本身需要1秒的执行时间 那callback函数后面的代码会和函数同时执行吗？
```

回掉地狱

18.b Promise

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。 所谓 Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。 有了 Promise 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise 对象提供统一的接口，使得控制异步操作更加容易。

```
 console.log('1');

loadFile(fileToLoad, (file) => {
  // 当读取结束后执行这个 function
  console.log('2');

  loadFile(anotherFile, (fileB) => {
    // 当读取结束后执行这个 function
    console.log('3');
  });

  console.log('4');
});

console.log('5');

console.log('1');
loadFile(fileToLoad)
  .then(() => {
    // 当读取 fileToLoad 结束后执行这个 function
    console.log('2');

    // Chain promise
    return loadFile(anotherFileToLoad);
  })
  .then(() => {
    // 当读取 anotherFileToLoad 结束后执行这个 function
    console.log('3');
  });

console.log('4');

myFastLoadFile(target)
  .then(() => {
    console.log('5');
  })
  .then(() => {
    return myFastLoadFile(anotherTarget)
  })
  .then(() => {
    console.log('6');
  })

// loadFile 需要1s
// myFastLoadFile 需要 0.6s
// 145263
```

```
起床();
洗漱();
听歌();
出门()
  .then(() => {
    看新闻():
    return 买咖啡()
      .then(() => {
        停止看新闻();
        刷朋友圈();
        return 买早点();
      });
      .then(() => {
        停止刷朋友圈();
        停止听歌();
        刷抖音();
        吃早点();
        return 前往公司();
      });
  })
  .catch(() => {
    return 投诉卖咖啡()
      .then(() => 换一家买咖啡())
      .catch(() => 我太难了());
  })
```

```
const promise = new Promise((resolve, reject) => {
  // ... some code

  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});

// 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。
promise.then(
  (value) => {
    // success
  },
  (error) => {
    // failure
  }
);

// 也可以使用 catch 来捕获错误
promise
  .then((value) => {
    // success
  })
  .catch((error) => {
    // failure
  });
```

改写 Timeout

```
// 一秒后执行 fn
setTimeout(() => {
  setTimeOut(() => {

  }, 2000)
}, 1000);

timeout(1000)
  .then(() => timeout(2000))
  .then(() => {

  });


function timeout(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('DONE');
    }, ms);
  });
}

timeout(100).then((value) => {
  console.log(value); // DONE
});
```

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）。因此可以采用链式写法，即then 方法后面再调用另一个then方法。

```
function timeout(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('DONE');
    }, ms);
  });
}

console.log('1');

timeout(100)
  .then((value) => {
    console.log(value);

    return 'Hello world';
  })
  .then((value) => {
    console.log(value);

    return 'My name is Long';
  })
  .then((value) => {
    console.log(value);

    // 这里没有任何 return
  })
  .then((value) =>  {
    console.log(value); // undefined

    return 'Foo';
  })
  .then((value) => {
    console.log(value) // Foo

    return 1 + 1;
  })
  .then((value) => {
    console.log(value) // 2
  })

console.log('2');

// 1 2 DONE Hello world My name is Long
```

常用API

- Promise.resolve(value)

```
Promise
  .resolve('Hello world')
  .then((v) => console.log(v)); // Hello world
```

- Promise.reject(value)

```
Promise
  .reject(new Error())
  .catch((v) => console.log(v));
```

- Promise.all([])

```
Promise.all([
  喝咖啡，3m
  吃面包，5m
  打一把王者，20m
])
  .then(([喝咖啡结果，吃面包结果，打一把王者结果]) => {
    // 20m
  })
  .catch([喝咖啡失败，吃面包失败，打一把王者失败] => {
  });
```

- Promise.race([])

```
Promise.race([
  喝咖啡，3m
  吃面包，5m
  打一把王者，20m
])
  .then((最快的结果) => {
    3m
  });
```

- Promise.any([])

Promise 其实已经解决了绝大多数异步问题？（大雾）

我们在上火车前买了一杯咖啡准备在到公司开早会开早会的时候喝。

```
买咖啡((咖啡) => {
  刷朋友圈();

  前往公司(() => {
    喝咖啡(咖啡);

    开早会();
  });
});

买咖啡()
  .then((咖啡) => {
    刷朋友圈();

    return 前往公司();
  })
  .then(() => {
    喝咖啡(咖啡);
    开早会();
  });

console.log('1');
买咖啡()
  .then((咖啡) => {
    console.log('2');
    刷朋友圈();

    前往公司().then(() => {
      console.log('3');
      喝咖啡(咖啡);
      return 开早会().then(() => {
        console.log('4');
        return 买早餐();
      });
    });
  })
  .then((结果) => {
    console.log('5');
  });
console.log('6');

// 162345
// 162534
```

```
// 两个then不在一个作用域的问题，也可以用promise。all解决吗？
买咖啡()
  .then((咖啡) => {
    刷朋友圈();

    return Promise.all([
      Promise.resolve(咖啡),
      前往公司,
    ]);
  })
  .then(([咖啡, 前往公司结果]) => {
    喝咖啡(咖啡);
    return 开早会();
  })
  .then(() => {
    ...
  });
```

```
买咖啡()
  .then((咖啡) => {
    刷朋友圈();
    return 前往公司()
      .then(() => {
        喝咖啡(咖啡);
        return 开早会();
      });
  })
  .then(() => {
    ...
  });
```

这种写法在代码中非常常见，比如:

获取 学生信息 -> 获取 课堂信息 -> 根据 学生信息 和 课堂信息 获取 学生成绩

```
getStudentInfo()
  .then((student) => {
    const { id: studentId } = student;

    return getStudentCourses(studentId);
  })
  .then((courses) => {
    const CS = courses[0];

    const { id: courseId } = CS;

    return getStudentCourseResult(studentId, courseId)
  });

getStudentInfo()
  .then((student) => {
    const { id: studentId } = student;

    return getStudentCourses(studentId);
  })
  .then((studentCourses) => {
    const CS = studentCourses[0];

    const { result } = CS;
  });
```

18.c Async 函数

async 函数返回一个 Promise 对象，可以使用 then 方法添加回调函数。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。

```
买咖啡()
  .then((咖啡) => {
    console.log('2');
    刷朋友圈();

    前往公司().then(() => {
      console.log('3');
      喝咖啡(咖啡);
      return 开早会().then(() => {
        console.log('4');
        return 买早餐();
      });
    });
  })
  .then((结果) => {
    console.log('5');
  });
console.log('6');
```

Async Await 函数

Async function 只是 promise 的语法糖

```
const 我的一天 = async () => {
  console.log('1');
  const 咖啡 = await 买咖啡();
  刷朋友圈();

  console.log('2');

  await 前往公司();
  // 这里是在什么时候发生 前往公司后
  喝咖啡(咖啡);

  console.log('3');
  await 开早会();

  买早餐();
  console.log('4');

  return 'Hello world';
};

class CurrentWeather extends React.Component() {
  async getCurrentWeather() {
    const response = await fetch('openWeatherAPI'); // 网络操作 异步操作
    const data = await response.json(); // I/O 异步操作

    this.setState({
      data,
    });
  }

  componentDidMount() {
    this.getCurrentWeather();
  }

  render() {
    const { data } = this.state;
    // data
  }
}

console.log('5');
我的一天().then((value) => {
  console.log('7');
  console.log(value);
});
console.log('6');

// 5612347 Hello world

async function 我的一天() {

}

// 1 2 3 4
```

```
const student = await getStudentInfo();

const courses = await getStudentCourse(student.id);
const cs = courses[0];

const result = await getCourseResult(student.id, cs.id);
```

```
刷朋友圈();
const 咖啡 = await 买咖啡();

await 前往公司();

喝咖啡(咖啡);
await 开早会();

写代码
```

```
promise
  .then(function(value) {
    // success
  })
  .catch(function(error) {
    // failure
  });
```

```
try {
  const value = await promise();
  // success
} catch (error) {
  // failure
}
```

await 不能单独使用，需要配合 async 函数。

```
const asyncFn = async () => {
  // ...
  await promise();
  // ...
}

async function asyncFn() {
  // ...
  await promise();
  // ...
}

asyncFn().then(() => {});

await asyncFn();
```

19. Tests

Web应用程序越来越复杂，这意味着有更多的可能出错。测试是帮助我们提高代码质量、降低错误的最好方法和工具之一。

- 测试可以确保得到预期结果。
- 加快开发速度。
- 方便维护。
- 提供用法的文档。

通过测试提供软件的质量，在开始的时候，可能会降低开发速度。但是从长期看，尤其是那种代码需要长期维护、不断开发的情况，测试会大大加快开发速度，减轻维护难度。

19.a 测试金字塔

https://insights.thoughtworks.cn/practical-test-pyramid/

UI Tests Services Tests Unit Tests

19.b 单元测试

“单元测试”这个词，本身就暗示，软件应该以模块化结构存在。每个模块的运作，是独立于其他模块的。一个软件越容易写单元测试，往往暗示着它的模块化结构越好，各模块之间的耦合就越弱；越难写单元测试，或者每次单元测试，不得不模拟大量的外部条件，很可能暗示软件的模块化结构越差，模块之间存在较强的耦合。

19.c TDD

TDD是“测试驱动的开发”（Test-Driven Development）的简称，指的是先写好测试，然后再根据测试完成开发。使用这种开发方式，会有很高的测试覆盖率。

1. 先写一个测试。
2. 写出最小数量的代码，使其能够通过测试。
3. 优化代码。
4. 重复前面三步。
5. ...

19.d BDD

BDD是“行为驱动的开发”（Behavior-Driven Development）的简称。

BDD认为，不应该针对代码的实现细节写测试，而是要针对行为写测试。BDD测试的是行为，即软件应该怎样运行。