一面 1：ES 基础知识点与高频考题解析

JavaScript 是 ECMAScript 规范的一种实现，本小节重点梳理下 ECMAScript 中的常考知识点，然后就一些容易出现的题目进行解析。

知识点梳理

- 变量类型
  - JS 的数据类型分类和判断
  - 值类型和引用类型
- 原型与原型链（继承）
  - 原型和原型链定义
  - 继承写法
- 作用域和闭包
  - 执行上下文
  - this
  - 闭包是什么
- 异步
  - 同步 vs 异步
  - 异步和单线程
  - 前端异步的场景
- ES6/7 新标准的考查
  - 箭头函数
  - Module
  - Class
  - Set 和 Map
  - Promise

变量类型

JavaScript 是一种弱类型脚本语言，所谓弱类型指的是定义变量时，不需要什么类型，在程序运行过程中会自动判断类型。

ECMAScript 中定义了 6 种原始类型：

- Boolean
- String
- Number
- Null
- Undefined
- Symbol（ES6 新定义）

注意：原始类型不包含 Object。

> 题目：类型判断用到哪些方法?

typeof

`typeof xxx`得到的值有以下几种类型：`undefined` `boolean` `number` `string` `object` `function`、`symbol`，比较简单，不再一一演示了。这里需要注意的有三点：

- `typeof null`结果是`object`，实际这是`typeof`的一个bug，null是原始值，非引用类型
- `typeof [1, 2]`结果是`object`，结果中没有`array`这一项，引用类型除了`function`其他的全部都是`object`
- `typeof Symbol()`用`typeof`获取`symbol`类型的值得到的是`symbol`，这是 ES6 新增的知识点

instanceof

用于实例和构造函数的对应。例如判断一个变量是否是数组，使用`typeof`无法判断，但可以使用`[1, 2] instanceof Array`来判断。因为，`[1, 2]`是数组，它的构造函数就是`Array`。同理：

```
function Foo(name) {
    this.name = name
}
var foo = new Foo('bar')
console.log(foo instanceof Foo) // true
```

> 题目：值类型和引用类型的区别

值类型 vs 引用类型

除了原始类型，ES 还有引用类型，上文提到的`typeof`识别出来的类型中，只有`object`和`function`是引用类型，其他都是值类型。

根据 JavaScript 中的变量类型传递方式，又分为**值类型**和**引用类型**，值类型变量包括 Boolean、String、Number、Undefined、Null，引用类型包括了 Object 类的所有，如 Date、Array、Function 等。在参数传递方式上，值类型是按值传递，引用类型是按共享传递。

下面通过一个小题目，来看下两者的主要区别，以及实际开发中需要注意的地方。

```
// 值类型
var a = 10
var b = a
b = 20
console.log(a)  // 10
console.log(b)  // 20
```

上述代码中，`a` `b`都是值类型，两者分别修改赋值，相互之间没有任何影响。再看引用类型的例子：

```
// 引用类型
var a = {x: 10, y: 20}
var b = a
b.x = 100
b.y = 200
console.log(a)  // {x: 100, y: 200}
console.log(b)  // {x: 100, y: 200}
```

上述代码中，`a` `b`都是引用类型。在执行了`b = a`之后，修改`b`的属性值，`a`的也跟着变化。因为`a`和`b`都是引用类型，指向了同一个内存地址，即两者引用的是同一个值，因此`b`修改属性时，`a`的值随之改动。

再借助题目进一步讲解一下。

> 说出下面代码的执行结果，并分析其原因。

```
function foo(a){
    a = a * 10;
}
function bar(b){
    b.value = 'new';
}
var a = 1;
var b = {value: 'old'};
foo(a);
bar(b);
console.log(a); // 1
console.log(b); // value: new
```

通过代码执行，会发现：

- `a`的值没有发生改变
- 而`b`的值发生了改变

这就是因为`Number`类型的`a`是按值传递的，而`Object`类型的`b`是按共享传递的。

JS 中这种设计的原因是：按值传递的类型，复制一份存入栈内存，这类类型一般不占用太多内存，而且按值传递保证了其访问速度。按共享传递的类型，是复制其引用，而不是整个复制其值（C 语言中的指针），保证过大的对象等不会因为不停复制内容而造成内存的浪费。

引用类型经常会在代码中按照下面的写法使用，或者说**容易不知不觉中造成错误**！

```
var obj = {
    a: 1,
    b: [1,2,3]
}
var a = obj.a
var b = obj.b
a = 2
b.push(4)
console.log(obj, a, b)
```

虽然 `obj` 本身是个引用类型的变量（对象），但是内部的 `a` 和 `b` 一个是值类型一个是引用类型，`a` 的赋值不会改变 `obj.a`，但是 `b` 的操作却会反映到 `obj` 对象上。

原型和原型链

JavaScript 是基于原型的语言，原型理解起来非常简单，但却特别重要，下面还是通过题目来理解下 JavaScript 的原型概念。

> 题目：如何理解 JavaScript 的原型

对于这个问题，可以从下面这几个要点来理解和回答，**下面几条必须记住并且理解**

- **所有的引用类型（数组、对象、函数），都具有对象特性，即可自由扩展属性（`null` 除外）**
- **所有的引用类型（数组、对象、函数），都有一个 `__proto__` 属性，属性值是一个普通的对象**
- **所有的函数，都有一个 `prototype` 属性，属性值也是一个普通的对象**
- **所有的引用类型（数组、对象、函数），`__proto__` 属性值指向它的构造函数的 `prototype` 属性值**

通过代码解释一下，大家可自行运行以下代码，看结果。

```
// 要点一：自由扩展属性
var obj = {}; obj.a = 100;
var arr = []; arr.a = 100;
function fn () {}
fn.a = 100;

// 要点二：__proto__
console.log(obj.__proto__);
console.log(arr.__proto__);
console.log(fn.__proto__);

// 要点三：函数有 prototype
console.log(fn.prototype)

// 要点四：引用类型的 __proto__ 属性值指向它的构造函数的 prototype 属性值
console.log(obj.__proto__ === Object.prototype)
```

原型

先写一个简单的代码示例。

```
// 构造函数
function Foo(name, age) {
    this.name = name
}
Foo.prototype.alertName = function () {
    alert(this.name)
}
// 创建示例
var f = new Foo('zhangsan')
f.printName = function () {
    console.log(this.name)
}
// 测试
f.printName()
f.alertName()
```

执行`printName`时很好理解，但是执行`alertName`时发生了什么？这里再记住一个重点 **当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么会去它的`__proto__`（即它的构造函数的`prototype`）中寻找**，因此`f.alertName`就会找到`Foo.prototype.alertName`。

那么如何判断这个属性是不是对象本身的属性呢？使用`hasOwnProperty`，常用的地方是遍历一个对象的时候。

```
var item
for (item in f) {
    // 高级浏览器已经在 for in 中屏蔽了来自原型的属性，但是这里建议大家还是加上这个判断，保证程序的健壮性
    if (f.hasOwnProperty(item)) {
        console.log(item)
    }
}
```

> 题目：如何理解 JS 的原型链

原型链

还是接着上面的示例，如果执行`f.toString()`时，又发生了什么？

```
// 省略 N 行

// 测试
f.printName()
f.alertName()
f.toString()
```

因为 `f` 本身没有 `toString()`，并且 `f.__proto__`（即 `Foo.prototype`）中也没有 `toString`。这个问题还是得拿出刚才那句话——**当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么会去它的 `__proto__`（即它的构造函数的 `prototype`）中寻找。**

如果在 `f.__proto__` 中没有找到 `toString`，那么就继续去 `f.__proto__.__proto__` 中寻找，因为 `f.__proto__` 就是一个普通的对象而已嘛！

- `f.__proto__` 即 `Foo.prototype`，没有找到 `toString`，继续往上找
- `f.__proto__.__proto__` 即 `Foo.prototype.__proto__`。`Foo.prototype` 就是一个普通的对象，因此 `Foo.prototype.__proto__` 就是 `Object.prototype`，在这里可以找到 `toString`
- 因此 `f.toString` 最终对应到了 `Object.prototype.toString`

这样一直往上找，你会发现是一个链式的结构，所以叫做“原型链”。如果一直找到最上层都没有找到，那么就宣告失败，返回 `undefined`。最上层是什么 —— `Object.prototype.__proto__ === null`

原型链中的 `this`

所有从原型或更高级原型中得到、执行的方法，其中的 `this` 在执行时，就指向了当前这个触发事件执行的对象。因此 `printName` 和 `alertName` 中的 `this` 都是 `f`。

作用域和闭包

作用域和闭包是前端面试中，最可能考查的知识点。例如下面的题目：

> 题目：现在有个 HTML 片段，要求编写代码，点击编号为几的链接就 `alert` 弹出其编号

```
<ul>
    <li>编号1，点击我请弹出1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
```

一般不知道这个题目用闭包的话，会写出下面的代码：

```
var list = document.getElementsByTagName('li');
for (var i = 0; i < list.length; i++) {
    list[i].addEventListener('click', function(){
        alert(i + 1)
    }, true)
}
```

实际上执行才会发现始终弹出的是 `6`，这时候就应该通过闭包来解决：

```
var list = document.getElementsByTagName('li');
for (var i = 0; i < list.length; i++) {
    list[i].addEventListener('click', function(i){
        return function(){
            alert(i + 1)
        }
    }(i), true)
}
```

要理解闭包，就需要我们从「执行上下文」开始讲起。

执行上下文

先讲一个关于 **变量提升** 的知识点，面试中可能会遇见下面的问题，很多候选人都回答错误：

> 题目：说出下面执行的结果（这里笔者直接注释输出了）

```
console.log(a)  // undefined
var a = 100

fn('zhangsan')  // 'zhangsan' 20
function fn(name) {
    age = 20
    console.log(name, age)
    var age
}

console.log(b); // 这里报错
// Uncaught ReferenceError: b is not defined
b = 100;
```

在一段 JS 脚本（即一个 `<script>` 标签中）执行之前，要先解析代码（所以说 JS 是解释执行的脚本语言），解析的时候会先创建一个 **全局执行上下文** 环境，先把代码中即将执行的（内部函数的不算，因为你不知道函数何时执行）变量、函数声明都拿出来。变量先暂时赋值为 `undefined`，函数则先声明好可使用。这一步做完了，然后再开始正式执行程序。再次强调，这是在代码执行之前才开始的工作。

我们来看下上面的面试小题目，为什么 `a` 是 `undefined`，而 `b` 却报错了，实际 JS 在代码执行之前，要「全文解析」，发现 `var a`，知道有个 `a` 的变量，存入了执行上下文，而 `b` 没有找到 `var` 关键字，这时候没有在执行上下文提前「占位」，所以代码执行的时候，提前报到的 `a` 是有记录的，只不过值暂时还没有赋值，即为 `undefined`，而 `b` 在执行上下文没有找到，自然会报错（没有找到 `b` 的引用）。

另外，一个函数在执行之前，也会创建一个 **函数执行上下文** 环境，跟 **全局上下文** 差不多，不过 **函数执行上下文** 中会多出 `this` `arguments` 和函数的参数。参数和 `arguments` 好理解，这里的 `this` 咱们需要专门讲解。

总结一下：

- 范围：一段`<script>`、js 文件或者一个函数
- 全局上下文：变量定义，函数声明
- 函数上下文：变量定义，函数声明，`this`，`arguments`

this

先搞明白一个很重要的概念 —— **`this`的值是在执行的时候才能确认，定义的时候不能确认！** 为什么呢 —— 因为`this`是执行上下文环境的一部分，而执行上下文需要在代码执行之前确定，而不是定义的时候。看如下例子

```
var a = {
    name: 'A',
    fn: function () {
        console.log(this.name)
    }
}
a.fn()  // this === a
a.fn.call({name: 'B'})  // this === {name: 'B'}
var fn1 = a.fn
fn1()  // this === window
```

`this`执行会有不同，主要集中在这几个场景中

- 作为构造函数执行，构造函数中
- 作为对象属性执行，上述代码中`a.fn()`
- 作为普通函数执行，上述代码中`fn1()`
- 用于`call` `apply` `bind`，上述代码中`a.fn.call({name: 'B'})`

下面再来讲解下什么是作用域和作用域链，作用域链和作用域也是常考的题目。

> 题目：如何理解 JS 的作用域和作用域链

作用域

ES6 之前 JS 没有块级作用域。例如

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

从上面的例子可以体会到作用域的概念，作用域就是一个独立的地盘，让变量不会外泄、暴露出去。上面的`name`就被暴露出去了，因此，**JS 没有块级作用域，只有全局作用域和函数作用域。**

```
var a = 100
function fn() {
    var a = 200
    console.log('fn', a)
}
console.log('global', a)
fn()
```

全局作用域就是最外层的作用域，如果我们写了很多行 JS 代码，变量定义都没有用函数包括，那么它们就全部都在全局作用域中。这样的坏处就是很容易撞车、冲突。

```
// 张三写的代码中
var data = {a: 100}

// 李四写的代码中
var data = {x: true}
```

这就是为何 jQuery、Zepto 等库的源码，所有的代码都会放在`(function(){....})()`中。因为放在里面的所有变量，都不会被外泄和暴露，不会污染到外面，不会对其他的库或者 JS 脚本造成影响。这是函数作用域的一个体现。

附：ES6 中开始加入了块级作用域，使用`let`定义变量即可，如下：

```
if (true) {
    let name = 'zhangsan'
}
console.log(name)  // 报错，因为let定义的name是在if这个块级作用域
```

作用域链

首先认识一下什么叫做 **自由变量** 。如下代码中，`console.log(a)`要得到`a`变量，但是在当前的作用域中没有定义`a`（可对比一下`b`）。当前作用域没有定义的变量，这成为 **自由变量** 。自由变量如何得到 —— 向父级作用域寻找。

```
var a = 100
function fn() {
    var b = 200
    console.log(a)
    console.log(b)
}
fn()
```

如果父级也没呢？再一层一层向上寻找，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是 **作用域链** 。

```
var a = 100
function F1() {
    var b = 200
    function F2() {
        var c = 300
        console.log(a) // 自由变量，顺作用域链向父作用域找
        console.log(b) // 自由变量，顺作用域链向父作用域找
        console.log(c) // 本作用域的变量
    }
    F2()
}
F1()
```

闭包

讲完这些内容，我们再来看一个例子，通过例子来理解闭包。

```
function F1() {
    var a = 100
    return function () {
        console.log(a)
    }
}
var f1 = F1()
var a = 200
f1()
```

自由变量将从作用域链中去寻找，但是 **依据的是函数定义时的作用域链，而不是函数执行时**，以上这个例子就是闭包。闭包主要有两个应用场景：

- **函数作为返回值**，上面的例子就是
- **函数作为参数传递**，看以下例子

```
function F1() {
    var a = 100
    return function () {
        console.log(a)
    }
}
function F2(f1) {
    var a = 200
    console.log(f1())
}
var f1 = F1()
F2(f1)
```

至此，对应着「作用域和闭包」这部分一开始的点击弹出`alert`的代码再看闭包，就很好理解了。

异步

异步和同步也是面试中常考的内容，下面笔者来讲解下同步和异步的区别。

同步 vs 异步

先看下面的 demo，根据程序阅读起来表达的意思，应该是先打印`100`，1秒钟之后打印`200`，最后打印`300`。但是实际运行根本不是那么回事。

```
console.log(100)
setTimeout(function () {
    console.log(200)
}, 1000)
console.log(300)
```

再对比以下程序。先打印`100`，再弹出`200`（等待用户确认），最后打印`300`。这个运行效果就符合预期要求。

```
console.log(100)
alert(200)  // 1秒钟之后点击确认
console.log(300)
```

这俩到底有何区别？—— 第一个示例中间的步骤根本没有阻塞接下来程序的运行，而第二个示例却阻塞了后面程序的运行。前面这种表现就叫做 **异步**（后面这个叫做 **同步** ），即**不会阻塞后面程序的运行**。

异步和单线程

JS 需要异步的根本原因是 **JS 是单线程运行的**，即在同一时间只能做一件事，不能“一心二用”。

一个 Ajax 请求由于网络比较慢，请求需要 5 秒钟。如果是同步，这 5 秒钟页面就卡死在这里啥也干不了了。异步的话，就好很多了，5 秒等待就等待了，其他事情不耽误做，至于那 5 秒钟等待是网速太慢，不是因为 JS 的原因。

讲到单线程，我们再来看个真题：

> 题目：讲解下面代码的执行过程和结果

```
var a = true;
setTimeout(function(){
    a = false;
}, 100)
while(a){
    console.log('while执行了')
}
```

这是一个很有迷惑性的题目，不少候选人认为`100ms`之后，由于`a`变成了`false`，所以`while`就中止了，实际不是这样，因为JS是单线程的，所以进入`while`循环之后，没有「时间」（线程）去跑定时器了，所以这个代码跑起来是个死循环！

前端异步的场景

- 定时`setTimeout` `setInterval`
- 网络请求，如`Ajax` `<img>`加载

Ajax 代码示例

```
console.log('start')
$.get('./data1.json', function (data1) {
    console.log(data1)
})
console.log('end')
```

img 代码示例（常用于打点统计）

```
console.log('start')
var img = document.createElement('img')
// 或者 img = new Image()
img.onload = function () {
    console.log('loaded')
    img.onload = null
}
img.src = '/xxx.png'
console.log('end')
```

ES6/7 新标准的考查

> 题目：ES6 箭头函数中的`this`和普通函数中的有什么不同

箭头函数

箭头函数是 ES6 中新的函数定义形式，`function name(arg1, arg2) {...}`可以使用`(arg1, arg2) => {...}`来定义。示例如下：

```
// JS 普通函数
var arr = [1, 2, 3]
arr.map(function (item) {
    console.log(index)
    return item + 1
})

// ES6 箭头函数
const arr = [1, 2, 3]
arr.map((item, index) => {
    console.log(index)
    return item + 1
})
```

箭头函数存在的意义，第一写起来更加简洁，第二可以解决 ES6 之前函数执行中`this`是全局变量的问题，看如下代码

```
function fn() {
    console.log('real', this)  // {a: 100} ，该作用域下的 this 的真实的值
    var arr = [1, 2, 3]
    // 普通 JS
    arr.map(function (item) {
        console.log('js', this)  // window 。普通函数，这里打印出来的是全局变量，令人费解
        return item + 1
    })
    // 箭头函数
    arr.map(item => {
        console.log('es6', this)  // {a: 100} 。箭头函数，这里打印的就是父作用域的 this
        return item + 1
    })
}
fn.call({a: 100})
```

> 题目：ES6 模块化如何使用?

Module

ES6 中模块化语法更加简洁，直接看示例。

如果只是输出一个唯一的对象，使用`export default`即可，代码如下

```
// 创建 util1.js 文件，内容如
export default {
    a: 100
}

// 创建 index.js 文件，内容如
import obj from './util1.js'
console.log(obj)
```

如果想要输出许多个对象，就不能用 `default` 了，且 `import` 时候要加 `{...}`，代码如下

```
// 创建 util2.js 文件，内容如
export function fn1() {
    alert('fn1')
}
export function fn2() {
    alert('fn2')
}

// 创建 index.js 文件，内容如
import { fn1, fn2 } from './util2.js'
fn1()
fn2()
```

> 题目：ES6 class 和普通构造函数的区别

class

class 其实一直是 JS 的关键字（保留字），但是一直没有正式使用，直到 ES6 。 ES6 的 class 就是取代之前构造函数初始化对象的形式，从语法上更加符合面向对象的写法。例如：

JS 构造函数的写法

```
function MathHandle(x, y) {
  this.x = x;
  this.y = y;
}

MathHandle.prototype.add = function () {
  return this.x + this.y;
};

var m = new MathHandle(1, 2);
console.log(m.add())
```

用 ES6 class 的写法

```
class MathHandle {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  add() {
    return this.x + this.y;
  }
}
const m = new MathHandle(1, 2);
console.log(m.add())
```

注意以下几点，全都是关于 class 语法的：

- class 是一种新的语法形式，是`class Name {...}`这种形式，和函数的写法完全不一样
- 两者对比，构造函数函数体的内容要放在 class 中的`constructor`函数中，`constructor`即构造器，初始化实例时默认执行
- class 中函数的写法是`add() {...}`这种形式，并没有`function`关键字

使用 class 来实现继承就更加简单了，至少比构造函数实现继承简单很多。看下面例子

JS 构造函数实现继承

```
// 动物
function Animal() {
    this.eat = function () {
        console.log('animal eat')
    }
}
// 狗
function Dog() {
    this.bark = function () {
        console.log('dog bark')
    }
}
Dog.prototype = new Animal()
// 哈士奇
var hashiqi = new Dog()
```

ES6 class 实现继承

```
class Animal {
    constructor(name) {
        this.name = name
    }
    eat() {
        console.log(`${this.name} eat`)
    }
```

```
}

class Dog extends Animal {
    constructor(name) {
        super(name)
        this.name = name
    }
    say() {
        console.log(`${this.name} say`)
    }
}
const dog = new Dog('哈士奇')
dog.say()
dog.eat()
```

注意以下两点：

- 使用 `extends` 即可实现继承，更加符合经典面向对象语言的写法，如 Java
- 子类的 `constructor` 一定要执行 `super()`，以调用父类的 `constructor`

> 题目：ES6 中新增的数据类型有哪些?

Set 和 Map

Set 和 Map 都是 ES6 中新增的数据结构，是对当前 JS 数组和对象这两种重要数据结构的扩展。由于是新增的数据结构，目前尚未被大规模使用，但是作为前端程序员，提前了解是必须做到的。先总结一下两者最关键的地方：

- Set 类似于数组，但数组可以允许元素重复，Set 不允许元素重复
- Map 类似于对象，但普通对象的 key 必须是字符串或者数字，而 Map 的 key 可以是任何数据类型

Set

Set 实例不允许元素有重复，可以通过以下示例证明。可以通过一个数组初始化一个 Set 实例，或者通过 `add` 添加元素，元素不能重复，重复的会被忽略。

```
// 例1
const set = new Set([1, 2, 3, 4, 4]);
console.log(set) // Set(4) {1, 2, 3, 4}

// 例2
const set = new Set();
[2, 3, 5, 4, 5, 8, 8].forEach(item => set.add(item));
for (let item of set) {
  console.log(item);
}
// 2 3 5 4 8
```

Set 实例的属性和方法有

- `size`：获取元素数量。
- `add(value)`：添加元素，返回 Set 实例本身。
- `delete(value)`：删除元素，返回一个布尔值，表示删除是否成功。
- `has(value)`：返回一个布尔值，表示该值是否是 Set 实例的元素。
- `clear()`：清除所有元素，没有返回值。

```
const s = new Set();
s.add(1).add(2).add(2); // 添加元素

s.size // 2

s.has(1) // true
s.has(2) // true
s.has(3) // false

s.delete(2);
s.has(2) // false

s.clear();
console.log(s);  // Set(0) {}
```

Set 实例的遍历，可使用如下方法

- `keys()`：返回键名的遍历器。
- `values()`：返回键值的遍历器。不过由于 Set 结构没有键名，只有键值（或者说键名和键值是同一个值），所以 `keys()` 和 `values()` 返回结果一致。
- `entries()`：返回键值对的遍历器。
- `forEach()`：使用回调函数遍历每个成员。

```
let set = new Set(['aaa', 'bbb', 'ccc']);

for (let item of set.keys()) {
  console.log(item);
}
// aaa
// bbb
// ccc

for (let item of set.values()) {
  console.log(item);
}
// aaa
// bbb
// ccc

for (let item of set.entries()) {
  console.log(item);
}
```

```
// ["aaa", "aaa"]
// ["bbb", "bbb"]
// ["ccc", "ccc"]

set.forEach((value, key) => console.log(key + ' : ' + value))
// aaa : aaa
// bbb : bbb
// ccc : ccc
```

Map

Map 的用法和普通对象基本一致，先看一下它能用非字符串或者数字作为 key 的特性。

```
const map = new Map();
const obj = {p: 'Hello World'};

map.set(obj, 'OK')
map.get(obj) // "OK"

map.has(obj) // true
map.delete(obj) // true
map.has(obj) // false
```

需要使用`new Map()`初始化一个实例，下面代码中`set` `get` `has` `delete`顾名即可思义（下文也会演示）。其中，`map.set(obj, 'OK')`就是用对象作为的 key （不光可以是对象，任何数据类型都可以），并且后面通过`map.get(obj)`正确获取了。

Map 实例的属性和方法如下：

- `size`：获取成员的数量
- `set`：设置成员 key 和 value
- `get`：获取成员属性值
- `has`：判断成员是否存在
- `delete`：删除成员
- `clear`：清空所有

```
const map = new Map();
map.set('aaa', 100);
map.set('bbb', 200);

map.size // 2

map.get('aaa') // 100

map.has('aaa') // true

map.delete('aaa')
map.has('aaa') // false
```

```
map.clear()
```

Map 实例的遍历方法有：

- `keys()`：返回键名的遍历器。
- `values()`：返回键值的遍历器。
- `entries()`：返回所有成员的遍历器。
- `forEach()`：遍历 Map 的所有成员。

```
const map = new Map();
map.set('aaa', 100);
map.set('bbb', 200);

for (let key of map.keys()) {
  console.log(key);
}
// "aaa"
// "bbb"

for (let value of map.values()) {
  console.log(value);
}
// 100
// 200

for (let item of map.entries()) {
  console.log(item[0], item[1]);
}
// aaa 100
// bbb 200

// 或者
for (let [key, value] of map.entries()) {
  console.log(key, value);
}
// aaa 100
// bbb 200
```

Promise

`Promise`是 CommonJS 提出来的这一种规范，有多个版本，在 ES6 当中已经纳入规范，原生支持 Promise 对象，非 ES6 环境可以用类似 Bluebird、Q 这类库来支持。

`Promise` 可以将回调变成链式调用写法，流程更加清晰，代码更加优雅。

简单归纳下 Promise：**三个状态、两个过程、一个方法**，快速记忆方法：**3-2-1**

三个状态：`pending`、`fulfilled`、`rejected`

两个过程：

- pending→fulfilled（resolve）
- pending→rejected（reject）

一个方法：`then`

当然还有其他概念，如`catch`、`Promise.all/race`，这里就不展开了。

关于 ES6/7 的考查内容还有很多，本小节就不逐一介绍了，如果想继续深入学习，可以在线看《ES6 入门》。

小结

本小节主要总结了 ES 基础语法中面试经常考查的知识点，包括之前就考查较多的原型、异步、作用域，以及 ES6 的一些新内容，这些知识点希望大家都要掌握。