ES6基础知识

美团平台/终端基础技术中心/标准组件和API组　　方伟

2024年4月

美团

目录

ES6简介

- ES6是ECMAScript6的简称，是JavaScript的下一代标准
- ECMAScript和JavaScript的关系

  前者是后者的规格和标准，后者是前者的一种实现

- ECMAScript 6的发展
  - 2011年，ECMAScript5.1发布后就开始制定6.0标准了，每年6月份正式发布一次，作为当年的正式版本
  - 2015年6月发布ECMAScript 6的第一个版本，后面每年都会小幅度修正
  - ES6是一个泛指，包括ES2015、ES2016、ES2017

ES6变量的定义

- ES6新增let命令，用来声明变量，类似于var

```
let a = 6;
var b = 5;
console.log(a);
console.log(b);
```

- let命令声明的变量只在let命令所在的代码块有效

```
{
        let a = 6;
        var b = 5;
}
console.log(a);  //ReferenceError: a is not defined
console.log(b); // 5
```

ES6变量的定义

- let命令不存在变量提升（使用在声明前）

```
console.log(a); //undefined
var a = 5;
console.log(b); //报错
let b = 6;
```

- let命令不允许在相同作用域内声明相同的变量

```
{
        var a = 6;
        let a = 5; //报错
}
{
        let a = 5;
        {let a = 6; console.log(a) // 6}
        console.log(a) // 5
}
```

ES6变量的定义

- const命令声明一个只读常量

```
const a = 5
console.log(a) // 5
a = 6 //报错
```

- const命令作用域、变量提升和重复声明和let命令一样

```
{
        const a = 6;
        console.log(a); //6
}

console.log(a) //ReferenceError: a is not defined
```

ES6变量的定义

- const命令本质

  const实际上保证的并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指向实际数据的指针，const只能保证这个指针是固定的（即总是指向另一个固定的地址），至于它指向的数据结构是不是可变的，就完全不能控制了。

```
const student = {};
student.name = “li ming”;
student.name = “lisa”;
console.log(student.name); //lisa
studuent = {}; //TypeError: “student” is read-only
```

ES6变量解构赋值

- 从数组和对象中提取值，对变量进行赋值，被称为解构

```
es5之前赋值：
var a = 1;
var b = 2;
var c = 3;

es6赋值：
let [a,b,c] = [1,2,3];
```

- 本质是模式匹配，只要左右两边模式一样，左边的变量可以赋值对应的值

```
let [a, [b],c] = [1, [2,3],4];
a//1
b//2
c//3
let [,,c] = [1,2,3];
c //3
```

ES6变量解构赋值

- 如果左右两边模式不匹配，赋值会报错

```
let [a] = 1; // 报错
let [a] = false; // 报错
let [a] = null; // 报错
let [a] = {};  //报错
let [a] = [1,2,3,4];  //1
```

- 解构可以赋默认值，默认值可以引用解构赋值的其他变量，但该变量必须已经声明。

```
let [a, b = 2] = [1]; //a =1, b=2
let [a, b=a] = [1]; // a=1,b=1
let [a=b, b=2] = [1]; //ReferenceError: b is not defined
```

ES6变量解构赋值

- 解构也可以用于对象，变量必须与对象的属性同名，才能取到正确的值

```
let {a} = {a:1,b:2}; // a = 1
let {c} = {a:1,b:2}; // c = undefined
```

- 字符串解构赋值，字符串解构赋值可以转换成字符串数组解构赋值。

```
let [a,b,c,d,e] = “hello”;
a // ’h’
b // ‘e’
c // ‘l’
d // ‘l’
e // ‘o’
```

思考题：解构赋值的用途有哪些?

字符串的扩展

- 在传统JavaScript中，输出字符串模板

```
$(“#resule”).append(
    “The student‘name is <b>” + student.name + “</b>” +
    “, his age is 23”
);
```

- ES6模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量

```
//单行模板字符串
`In Javascript ‘\n’ is a line-feed`
//多行模板字符串
`In Javascript this is
 not legal`
//字符串中嵌入变量
let name = “blob”, time = “today”;
`Hello ${name}, how are you ${time}`
```

字符串的新增方法

- includes()、startsWith()、endsWith()

  includes()：返回布尔值，表示是否找到了参数字符串。
  startsWith()：返回布尔值，表示参数字符串是否在原字符串的头部。
  endsWith()：返回布尔值，表示参数字符串是否在原字符串的尾部。

```
let s = “hello”;
s.includes(‘o’)  //true
s.startsWith(‘el’) //false
s.endsWith(‘llo’) //true
```

- repeat()

  repeat方法返回一个新字符串，表示将原字符串重复n次。

```
“x”.repeat(5)  //”xxxxx”
“hello”.repeat(2) //”hellohello”
“hello”.repeat(0) //””
```

字符串的新增方法

- padStart()、padEnd()

  字符串补全长度的功能。如果某个字符串不够指定长度，会在头部或尾部补全。padStart()用于头部补全，padEnd()用于尾部补全。

```
“x”.padStart(5, “ab”)  //”ababx”
“x”.padEnd(4, “ab”)  //”xaba”
“x”.padStart(5) //”    x”
```

- trimStart()、trimEnd()

  trimStart()消除字符串头部的空格，trimEnd()消除尾部的空格。它们返回的都是新字符串，不会修改原始字符串。

```
const s = “   hello   ”;
s.trim(); // “hello”
s.trimStart(); //”hello   ”
s.trimEnd(); //”   hello”
```

字符串的新增方法

- replaceAll()

  字符串的实例方法replace()只能替换第一个匹配，replaceAll()方法可以一次性替换所有匹配，返回一个新字符串，不会改变原字符串。

```
”aabbcc”.replace(“b”, “_”)  // ”aa_bcc”
“aabbcc”.replaceAll(“b”, ”_”)  // ”aa__cc”
```

- at()

  at()方法接受一个整数作为参数，返回参数指定位置的字符，支持负索引（即倒数的位置）。

```
const s = “hello”;
s.at(1); // “e”
s.at(-1); // ”o”
s.at(6); // undefined
```

函数的扩展

- 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，ES6允许为函数的参数设置默认值，即直接写在参数定义的后面，使用参数默认值时，函数不能有同名参数。

```
function log(x, y = “world”) {
                console.log(x, y);
}

log(“hello”); // “hello world”
log(“hello”, “china”); //”hello china”
log(“hello”, “”);   // “hello”
```

- name属性

函数的name属性，返回该函数的函数名

```
var f = function log() {}
f.name; // “log”
```

函数的扩展

- 箭头函数

  ES6允许使用“箭头”（=>）定义函数，如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。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。

```
var sum = (sum1, sum2) => sum1 + sum2;
var sum = (sum1, sum2) => {return sum1 + sum2;}
```

对象的扩展

- 属性的简洁表示法

ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。属性的赋值器（setter）和取值器（getter），事实上也是采用简洁写法。

```
let birth = 23
var person = {
        name: “zhangsan”,
        birth,
        hello() {console.log(“hello” +name);}
}
```

```
const cart = {
        _wheels: 4,
        get wheels () { return this._wheels; },
        set wheels (value) {this._wheels = value; }
}
```

对象的扩展

- 属性的遍历

  for...in：循环遍历对象自身的和继承的可枚举属性
  Object.keys(obj)：返回一个数组，包括对象自身的（不含继承的）所有可枚举属性的键名。
  Object.getOwnPropertyNames(obj)：返回一个数组，包含对象自身的所有属性包括不可枚举属性）的键名。

- super关键字

  this关键字总是指向函数所在的当前对象，super关键字指向当前对象的原型对象。

```
const proto = { foo: 'hello' };
const obj = {
              foo: 'world’,
             find() { return super.foo; }
};
Object.setPrototypeOf(obj, proto);
obj.find(); // "hello"
```

对象的扩展

- 扩展运算符

对象的扩展运算符（...）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。对象的扩展运算符，只会返回参数对象自身的、可枚举的属性，不会返回对象的方法。

```
class c {
                p = 12;
                m() {}
} ;
let c = new C();
let clone = { ...c };
clone.p; // ok
clone.m(); // 报错
```

运算符的扩展

- 指数运算符

ES6新增了一个指数运算符（**）。

```
2 ** 2 // 4
2 ** 3 // 8
```

- 链式运算符

?.运算符判断左侧的对象是否为null或undefined。如果是的，返回undefined，如果不是，执行.后面的。

```
a?.b  // 等同于 a == null ? undefined : a.b
a?.[x]  // 等同于 a == null ? undefined : a[x]
a?.b()  // 等同于 a == null ? undefined : a.b()
```

Set和Map数据结构

- Set的用法

ES6提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。

```
const s = new Set();
[2, 3, 5, 4, 5, 2, 2].forEach(
        x => s.add(x)
);

for (let i of s) {
        console.log(i);
}
// 2 3 5 4
```

Set和Map数据结构

- Set的常用属性和方法

size：返回Set实例的成员总数。
add(value)：添加某个值，返回 Set 结构本身。
delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。
has(value)：返回一个布尔值，表示该值是否为Set的成员。
clear()：清除所有成员，没有返回值。

```
s.add(1).add(2).add(2); // 注意2被加入了两次
s.size // 2
s.has(1) // true
s.has(2) // true
s.has(3) // false
s.delete(2);
s.has(2) // false
```

Set和Map数据结构

美团

- Set的遍历

keys()：返回键名的遍历器
values()：返回键值的遍历器
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(['red', 'green', 'blue']);
for (let item of set.keys()) { console.log(item); } // red // green // blue
for (let item of set.values()) { console.log(item); } // red // green // blue
for (let item of set.entries()) { console.log(item); } // ["red", "red"] // ["green", "green"] // ["blue", "blue"]
let set = new Set([1, 4, 9]);
set.forEach((value, key) => console.log(key + ' : ' + value)) // 1 : 1  4 : 4  9 : 9
```

- Set的扩展运算符（...）

```
let set = new Set(['red', 'green', 'blue']);
var array = [...set];
// ['red', 'green', 'blue']
```

Set和Map数据结构

- Map的用法

ES6 提供了Map数据结构，它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键

```
const m = new Map();
const o = {p: 'Hello World’};
m.set(o, 'content’)
m.get(o) // "content"
m.has(o) // true
m.delete(o) // true
m.has(o) // false
```

Set和Map数据结构

- Map的常用属性和方法

size：返回Map实例的成员总数。
set(key, value)：设置键名key对应的键值为value，然后返回整个 Map 结构。如果key已经有值，则键值会被更新，否则就新生成该键。
get(key)读取key对应的键值，如果找不到key，返回undefined。
has(key)：返回一个布尔值，表示某个键是否在当前 Map 对象之中。
delete(key)：删除某个键，返回true。如果删除失败，返回false。
clear()：清除所有成员，没有返回值。

```
const map = new Map();
map.set('foo', true);
map.set('bar', false);
map.size // 2
map.get(‘foo’) //true
map.has(’barl’) // false
map.delete(‘bar’) //true
map.clear()
map.size // 0
```

Set和Map数据结构

美团

- Map的遍历

keys()：返回键名的遍历器。
values()：返回键值的遍历器
entries()：返回键值对的遍历器
forEach()：使用回调函数遍历每个成员

```
const map = new Map([ ['F', 'no'], ['T', 'yes'], ]);
for (let key of map.keys()) { console.log(key); }  // "F" // "T"
for (let value of map.values()) { console.log(value); }   // "no" // "yes"
for (let item of map.entries()) { console.log(item[0], item[1]); }  // "F" "no" // "T" "yes"
```

- Map的扩展运算符（...）

```
const map = new Map([ [1, 'one'], [2, 'two'], [3, 'three'], ]);
 [...map.keys()]   // [1, 2, 3]
 [...map.values()]  // ['one', 'two', 'three’]
 [...map.entries()]  // [[1,'one'], [2, 'two'], [3, 'three’]]
 [...map]  // [[1,'one'], [2, 'two'], [3, 'three']]
```

异步处理

- Promise的含义

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。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。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。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。

- Promise的特点

1.对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：pending（进行中）、resolved（已成功）和rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。
2.一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从pending变为resolved和从pending变为rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）。如果改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。

异步处理

- Promise的基本用法

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。resolve函数的作用是将Promise对象的状态从未完成变为成功（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject函数的作用是将Promise对象的状态从未完成变为失败（即从 pending 变为 rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

```
function loadImageAsync(url) {
        return new Promise(function(resolve, reject) {
                const image = new Image();
                 image.onload = function() { resolve(image); };
                image.onerror = function() { reject(new Error('Could not load image at ' + url)); };
                image.src = url;
        });
}
```

异步处理

- Promise的then方法

Promise实例具有then方法，它的作用是为Promise实例添加状态改变时的回调函数。then方法的第一个参数是resolved状态的回调函数，第二个参数是rejected状态的回调函数，它们都是可选的。then方法返回的是一个新的Promise实例。

```
getJSON("/post/1.json").then(function(post) {
                return getJSON(post.commentURL);
}).then(function (comments) {
                console.log("resolved: ", comments);
}, function (err){
                 console.log("rejected: ", err);
 });
```

异步处理

- Promise的catch方法

  catch()方法是.then(null, rejection)或.then(undefined, rejection)的别名，用于指定发生错误时的回调函数。catch方法返回的是一个新的Promise实例。

```
const promise = new Promise(function(resolve, reject) {
        throw new Error('test');
});
promise.catch(function(error) {
        console.log(error);
});

// Error: test
```

异步处理

- Promise的finally方法

  finally()方法用于指定不管 Promise 对象最后状态如何，都会执行的操作， finally方法的回调函数不接受任何参数。

```
const promise = new Promise(function(resolve, reject) {
                throw new Error(“ test”  );
});
promise.catch(function(error) {
                console.log(error);
}).finally(function() {
               console.log( “End”  )
});

// Error: test
// End
```

异步处理

美团

- Promise的all方法

Promise.all()方法接受一个Promise数组作为参数，返回一个新的Promise实例，只有所有参数实例状态都变成resolved，包装实例的状态才会变成resolved，此时参数实例的返回值组成一个数组，传递给包装实例的回调函数。只要参数实例之中有一个被rejected，包装实例的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给包装实例的回调函数。

```
const p1 = new Promise((resolve, reject) => { resolve('hello'); }) .then(result => result);
const p2 = new Promise((resolve, reject) => { throw new Error(‘error'); }) .then(result => result);
Promise.all([p1, p2]) .then(
            result => console.log(result)
) .catch(
            e => console.log(e)
);
// Error: error
```

异步处理

- Promise的any方法

Promise.any()方法接受一个Promise数组作为参数，返回一个新的Promise实例，只要参数实例有一个变成resolved状态，包装实例就会变成resolved状态；如果所有参数实例都变成rejected状态，包装实例就会变成rejected状态。Promise.any()抛出的错误是一个 AggregateError 实例，这个 AggregateError 实例对象的errors属性是一个数组，包含了所有成员的错误。

```
Promise.any([
            fetch('https://v8.dev/').then(() => 'home’),
            fetch('https://v8.dev/blog').then(() => 'blog’),
            fetch('https://v8.dev/docs').then(() => 'docs’)
]).then((first) => {
            // 只要有一个 fetch() 请求成功
            console.log(first);
}).catch((error) => {
            // 所有三个 fetch() 全部请求失败
             console.log(error);
});
```

异步处理

- Promise的resove方法

Promise.resolve()方法直接返回一个状态为resolved的 Promise 对象。

```
const p = Promise.resolve('Hello’);
p.then(function (s) { console.log(s) });
// Hello
```

- Promise的reject方法

Promise.reject(reason)方法直接返回一个状态为rejected的Promise 实例

```
Promise.reject('出错了’)
.catch(e => {
                console.log(e === '出错了’)
}) ;
// true
```

异步处理

- async命令

  async命令返回一个 Promise 对象，可以使用then方法添加回调函数。async命令内部return语句返回的值，会成为then方法回调函数的参数。async命令内部抛出错误，会导致返回的 Promise 对象变为rejected状态。抛出的错误对象会被catch方法回调函数接收到。

```
async function getStockPriceByName(name) {
        const symbol = await getStockSymbol(name);
        const stockPrice = await getStockPrice(symbol);
        return stockPrice;
}
getStockPriceByName('goog’)
.then(function (result) {
        console.log(result);
});
```

异步处理

- await命令

  await命令需要在async函数中使用，await命令后面是一个 Promise 对象，返回该对象的结果。如果不是 Promise 对象，就直接返回对应的值。await命令后面的 Promise 对象如果变为rejected状态，则reject的参数会被catch方法的回调函数接收到。任何一个await语句后面的 Promise 对象变为rejected状态，那么整个async函数都会中断执行。

```
async function f() {
        await new Promise(function (resolve, reject) {
                throw new Error('出错了’);
        });
}

f() .then(v => console.log(v))
.catch(e => console.log(e))
// Error：出错了
```

Class基础知识

美团

- 类的概念

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。

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

        toString() {
                return '(' + this.x + ', ' + this.y + ')’;
        }
}
```

Class基础知识

美团

- 类的构造方法

一个类必须有constructor()方法，如果没有显式定义，一个空的constructor()方法会被默认添加。

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

- 类的实例化

使用new命令生成类的实例。如果忘记加上new，将会报错。

```
var p1 = new Point(2,3) //正确
var p2 = Point(2,3) //错误
```

Class基础知识

美团

- 类的属性

  类的属性可以定义在constructor()方法里面的this上面，也可以定义在类内部的最顶层。

写法1：

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

写法2：

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

Class基础知识

- 类的取值函数和存值函数

在类的内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。

```
class MyClass {
        constructor() {
                // ...
        }
        get prop() {
                return 'getter’;
        }
        set prop(value) {
                console.log('setter: '+value);
        }
}

let inst = new MyClass();
inst.prop = 123; // setter: 123
inst.prop // 'getter'
```

Class基础知识

- 静态属性和方法

静态属性指的是类本身的属性，不是实例对象的属性。静态属性是在属性前加static关键字。静态方法是类方法，不会被实例继承，直接通过类来调用，静态方法也是在方法前加static关键字。

```
class Foo {
        static classVar = 10;
        static classMethod() { return 'hello’; }
 }

Foo.classVar; // 10
Foo.classMethod(); //  'hello’
var foo = new Foo();
foo.classVar; // TypeError: foo.classVar is not defined
foo.classMethod() // TypeError: foo.classMethod is not a function
```

Class基础知识

- 私有属性和方法

私有方法和私有属性，是只能在类的内部访问的方法和属性，外部不能访问。早期通过在属性和方法前面加下划线(_)表示私有属性和私有方法，ES2022正式约定在属性和方法前面加#表示私有属性和私有方法。

```
class Foo {
        #a, #b;
        constructor(a, b) {
                this.#a = a;
                this.#b = b;
        }
        #sum() {
                return this.#a + this.#b;
        }
         printSum() {
                 console.log(this.#sum());
        }
}
```

Class基础知识

- this关键字

this关键字可以作为方法，表示调用当前类的构造函数，也可以作为对象，在普通方法中指向当前类的原型对象，在静态方法中，指向当前类。

```
class Point {
        static z = 10;
        constructor(x, y) {
                this.x = x;
                this.y = y;
        }
        print() {
                console.log(this.x, this.y);
        }
        static printZ() {
                console.log(this.z);
        }
}
```

Class基础知识

美团

- 类的继承

Class 可以通过extends关键字实现继承，让子类继承父类的属性和方法。子类必须在constructor()方法中调用super()，否则就会报错。

```
class Point {
        constructor(x, y) {
                this.x = x;
                this.y = y;
         }
 }
class ColorPoint extends Point {
        constructor(x, y, color) {
                super(x, y);
                this.color = color;
        }
 }
```

Class基础知识

- 属性和方法的继承

父类所有的属性和方法，都会被子类继承，除了私有的属性和方法。

```
class Foo {
        #p = 1;
        #m() { console.log('hello'); }
 }

class Bar extends Foo {
         constructor() {
                 super();
                 console.log(this.#p); // 报错
                 this.#m(); // 报错
         }
}
```

Class基础知识

- super关键字

super关键字可以作为方法，表示调用父类的构造函数，也可以作为对象，在普通方法中指向父类的原型对象，在静态方法中，指向父类。

```
class A {
        p() {
                return 2;
         }
 }
class B extends A {
        constructor() {
                super();
                console.log(super.p()); // 2
        }
 }

 let b = new B();
```

Module的加载

- 概述

在 ES6 之前，社区制定了一些模块加载方案，最主要的有 CommonJS 和 AMD 两种。CommonJS 和 AMD 模块都只能在运行时确定这些东西。比如，CommonJS 模块就是对象，输入时必须查找对象属性。

```
// CommonJS模块
let { stat, exists, readfile } = require('fs’);
// 等同于
let _fs = require('fs'); let stat = _fs.stat; let exists = _fs.exists; let readfile = _fs.readfile;
```

ES6 模块的设计思想是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。ES6 通过export命令显式指定输出的代码，再通过import命令输入。

```
// ES6模块
 import { stat, exists, readFile } from 'fs';
```

Module的加载

- export命令

一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。通常情况下，export输出的变量就是本来的名字，但是可以使用as关键字重命名。

```
// profile.js
var firstName = 'Michael’;
var lastName = 'Jackson’;
 var year = 1958;
export { firstName, lastName, year };
```

```
function v1() { ... }
 function v2() { ... }
export { v1 as streamV1, v2 as streamV2, v2 as streamLatestVersion };
```

Module的加载

- import命令

import命令接受一对大括号，里面指定要从其他模块导入的变量名。大括号里面的变量名，必须与被导入模块对外接口的名称相同。如果想为输入的变量重新取一个名字，import命令要使用as关键字，将输入的变量重命名。

```
// main.js
import { firstName, lastName, year } from './profile.js';
function setName(element) {
            element.textContent = firstName + ' ' + lastName;
 }
```

如果多次重复执行同一句import语句，那么只会执行一次，而不会执行多次。也就是说一个模块只会被加载一次。

```
import { foo } from 'my_module';
import { bar } from 'my_module';
// 等同于
import { foo, bar } from 'my_module';
```

Module的加载

- 模块的整体加载

除了指定加载某个输出值，还可以使用整体加载，即用星号（*）指定一个对象，所有输出值都加载在这个对象上面。

```
// circle.js
export function area(radius) {
        return Math.PI * radius * radius;
}
export function circumference(radius) {
        return 2 * Math.PI * radius;
 }
```

```
//main.js
import * as circle from './circle’;
console.log('圆面积：' + circle.area(4));
console.log('圆周长：' + circle.circumference(14));
```

Module的加载

- export default命令

使用import命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。但是，用户肯定希望快速上手，未必愿意阅读文档，去了解模块有哪些属性和方法。为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。一个模块只能有一个默认输出，因此export default命令只能使用一次。import命令后面不用加大括号，因为只可能唯一对应export default命令。

```
// 默认导出
export default function crc32() {  ... }  //输出
import crc32 from 'crc32'; // 输入

//非默认导出
export function crc32() {... };  //输出
import {crc32} from 'crc32'; // 输入
```

Module的加载

- export和import的复合用法

如果在一个模块之中，先输入后输出同一个模块，import语句可以与export语句写在一起。

```
export { foo, bar } from 'my_module’;
// 等同于
import { foo, bar } from 'my_module’;
export { foo, bar };
```

```
export { es6 as default } from './someModule’;
// 等同于
import { es6 } from './someModule’;
export default es6;
```

```
export * as ns from "mod";
// 等同于
import * as ns from "mod";
export {ns};
```

ES6的其他特性(自学)

- Symbol
- Reflect
- Proxy
- 迭代器
- Generator函数
- 装饰器

学习参考资料：

https://es6.ruanyifeng.com/#docs/generator-async

https://es6.ruanyifeng.com/

课后练习题

定义一个图片类，分别使用Promise和async函数完成多张网络图片的加载实现，并把图片类通过导出。注：图片的地址可以自定义。