4.1. 属性的简写

ES6中，当对象键名与对应值名相等的时候，可以进行简写

JavaScript | 复制代码

```
const baz = {foo:foo}

// 等同于
const baz = {foo}
```

方法也能够进行简写

```javascript
const o = {
  method() {
    return "Hello!";
  }
};

// 等同于

const o = {
  method: function() {
    return "Hello!";
  }
}
```

在函数内作为返回值，也会变得方便很多

```javascript
function getPoint() {
  const x = 1;
  const y = 10;
  return {x, y};
}

getPoint()
// {x:1, y:10}
```

注意：简写的对象方法不能用作构造函数，否则会报错

```javascript
const obj = {
  f() {
    this.foo = 'bar';
  }
};

new obj.f() // 报错
```

4.2. 属性名表达式

ES6 允许字面量定义对象时，将表达式放在括号内

```javascript
let lastWord = 'last word';

const a = {
  'first word': 'hello',
  [lastWord]: 'world'
};

a['first word'] // "hello"
a[lastWord] // "world"
a['last word'] // "world"
```

表达式还可以用于定义方法名

```javascript
let obj = {
  ['h' + 'ello']() {
    return 'hi';
  }
};

obj.hello() // hi
```

注意，属性名表达式与简洁表示法，不能同时使用，会报错

```javascript
// 报错
const foo = 'bar';
const bar = 'abc';
const baz = { [foo] };

// 正确
const foo = 'bar';
const baz = { [foo]: 'abc'};
```

注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串 `[object Object]`

```javascript
const keyA = {a: 1};
const keyB = {b: 2};

const myObject = {
  [keyA]: 'valueA',
  [keyB]: 'valueB'
};

myObject // Object {[object Object]: "valueB"}
```

4.3. super关键字

`this` 关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字 `super` ，指向当前对象的原型对象

```javascript
const proto = {
  foo: 'hello'
};

const obj = {
  foo: 'world',
  find() {
    return super.foo;
  }
};

Object.setPrototypeOf(obj, proto); // 为obj设置原型对象
obj.find() // "hello"
```

4.4. 扩展运算符的应用

在解构赋值中，未被读取的可遍历的属性，分配到指定的对象上面

```javascript
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x // 1
y // 2
z // { a: 3, b: 4 }
```

注意：解构赋值必须是最后一个参数，否则会报错

解构赋值是浅拷贝

```javascript
let obj = { a: { b: 1 } };
let { ...x } = obj;
obj.a.b = 2; // 修改obj里面a属性中键值
x.a.b // 2，影响到了结构出来x的值
```

对象的扩展运算符等同于使用 `Object.assign()` 方法

4.5. 属性的遍历

ES6 一共有 5 种方法可以遍历对象的属性。

- for...in：循环遍历对象自身的和继承的可枚举属性（不含 Symbol 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（不含 Symbol 属性）的键名
- Object.getOwnPropertyNames(obj)：回一个数组，包含对象自身的所有属性（不含 Symbol 属性，但是包括不可枚举属性）的键名
- Object.getOwnPropertySymbols(obj)：返回一个数组，包含对象自身的所有 Symbol 属性的键名
- Reflect.ownKeys(obj)：返回一个数组，包含对象自身的（不含继承的）所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举

上述遍历，都遵守同样的属性遍历的次序规则：

- 首先遍历所有数值键，按照数值升序排列
- 其次遍历所有字符串键，按照加入时间升序排列
- 最后遍历所有 Symbol 键，按照加入时间升序排

```JavaScript
Reflect.ownKeys({ [Symbol()]:0, b:0, 10:0, 2:0, a:0 })
// ['2', '10', 'b', 'a', Symbol()]
```

4.6. 对象新增的方法

关于对象新增的方法，分别有以下：

- Object.is()
- Object.assign()
- Object.getOwnPropertyDescriptors()
- Object.setPrototypeOf()，Object.getPrototypeOf()
- Object.keys()，Object.values()，Object.entries()
- Object.fromEntries()

4.6.1. Object.is()

严格判断两个值是否相等，与严格比较运算符（===）的行为基本一致，不同之处只有两个：一是 `+0` 不等于 `-0`，二是 `NaN` 等于自身

```JavaScript
+0 === -0 //true
NaN === NaN // false

Object.is(+0, -0) // false
Object.is(NaN, NaN) // true
```

4.6.2. Object.assign()

`Object.assign()` 方法用于对象的合并，将源对象 `source` 的所有可枚举属性，复制到目标对象 `target`

`Object.assign()` 方法的第一个参数是目标对象，后面的参数都是源对象

```JavaScript
const target = { a: 1, b: 1 };

const source1 = { b: 2, c: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

注意：`Object.assign()` 方法是浅拷贝，遇到同名属性会进行替换

4.6.3. Object.getOwnPropertyDescriptors()

返回指定对象所有自身属性（非继承属性）的描述对象

```JavaScript
const obj = {
  foo: 123,
  get bar() { return 'abc' }
};

Object.getOwnPropertyDescriptors(obj)
// { foo:
//    { value: 123,
//      writable: true,
//      enumerable: true,
//      configurable: true },
//   bar:
//    { get: [Function: get bar],
//      set: undefined,
//      enumerable: true,
//      configurable: true } }
```

4.6.4. Object.setPrototypeOf()

`Object.setPrototypeOf` 方法用来设置一个对象的原型对象

JavaScript | 复制代码

```
Object.setPrototypeOf(object, prototype)

// 用法
const o = Object.setPrototypeOf({}, null);
```

4.6.5. Object.getPrototypeOf()

用于读取一个对象的原型对象

JavaScript | 复制代码

```
Object.getPrototypeOf(obj);
```

4.6.6. Object.keys()

返回自身的（不含继承的）所有可遍历（enumerable）属性的键名的数组

JavaScript | 复制代码

```
var obj = { foo: 'bar', baz: 42 };
Object.keys(obj)
// ["foo", "baz"]
```

4.6.7. Object.values()

返回自身的（不含继承的）所有可遍历（enumerable）属性的键对应值的数组

JavaScript | 复制代码

```
const obj = { foo: 'bar', baz: 42 };
Object.values(obj)
// ["bar", 42]
```

4.6.8. Object.entries()

返回一个对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对的数组

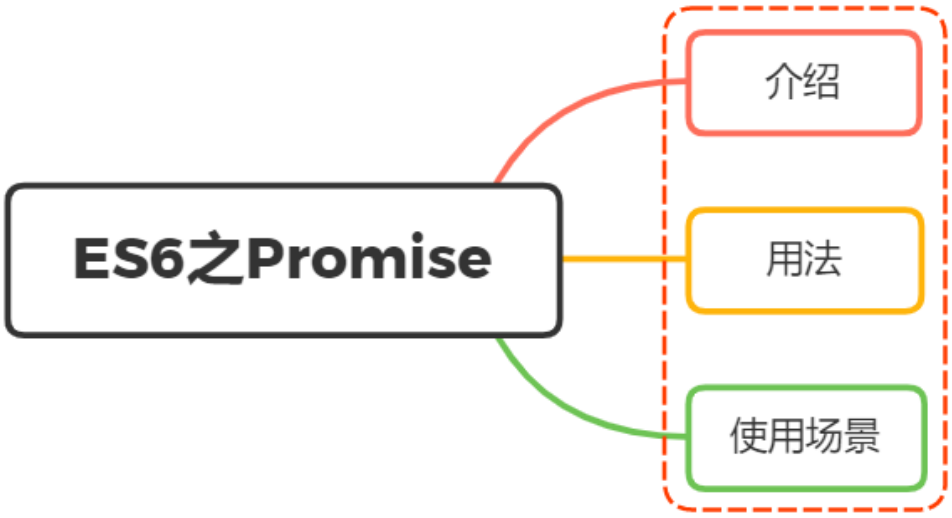
```JavaScript
const obj = { foo: 'bar', baz: 42 };
Object.entries(obj)
// [ ["foo", "bar"], ["baz", 42] ]
```

4.6.9. Object.fromEntries()

用于将一个键值对数组转为对象

```JavaScript
Object.fromEntries([
  ['foo', 'bar'],
  ['baz', 42]
])
// { foo: "bar", baz: 42 }
```

5. 你是怎么理解ES6中 Promise的？使用场景？



5.1. 介绍

`Promise`，译为承诺，是异步编程的一种解决方案，比传统的解决方案（回调函数）更加合理和更加强大

在以往我们如果处理多层异步操作，我们往往会像下面那样编写我们的代码

```javascript
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('得到最终结果: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

阅读上面代码，是不是很难受，上述形成了经典的回调地狱

现在通过 `Promise` 的改写上面的代码

```javascript
doSomething().then(function(result) {
    return doSomethingElse(result);
})
.then(function(newResult) {
    return doThirdThing(newResult);
})
.then(function(finalResult) {
    console.log('得到最终结果: ' + finalResult);
})
.catch(failureCallback);
```

瞬间感受到 `promise` 解决异步操作的优点：

- 链式操作减低了编码难度
- 代码可读性明显增强

下面我们正式来认识 `promise` ：

5.1.1. 状态

`promise` 对象仅有三种状态

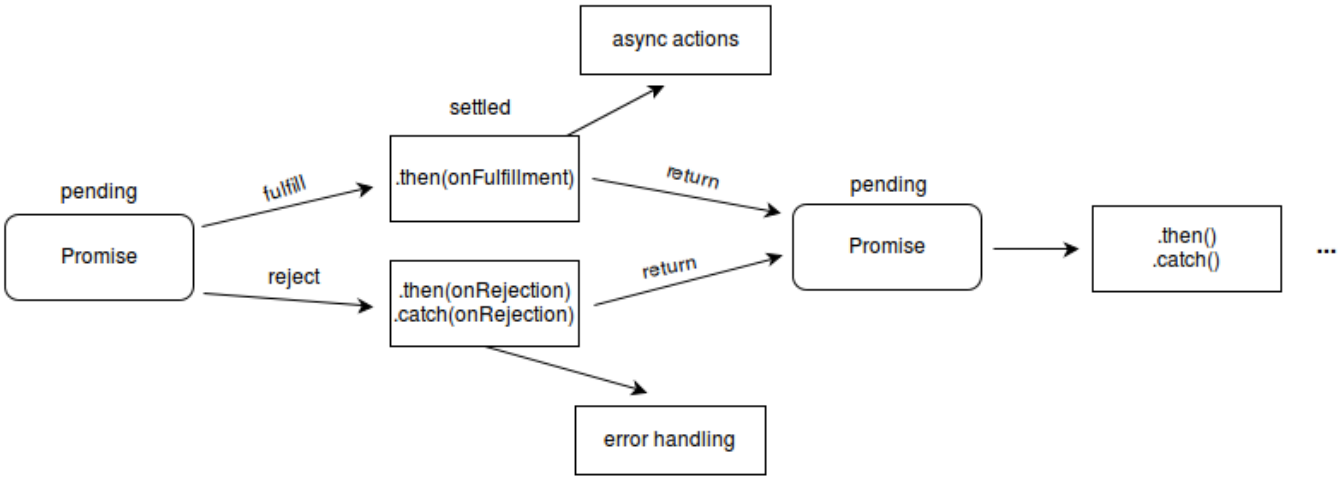
- `pending` （进行中）
- `fulfilled` （已成功）
- `rejected` （已失败）

5.1.2. 特点

- 对象的状态不受外界影响，只有异步操作的结果，可以决定当前是哪一种状态
- 一旦状态改变（从 `pending` 变为 `fulfilled` 和从 `pending` 变为 `rejected` ），就不会再变，任何时候都可以得到这个结果

5.1.3. 流程

认真阅读下图，我们能够轻松了解 `promise` 整个流程



5.2. 用法

`Promise` 对象是一个构造函数，用来生成 `Promise` 实例

```
const promise = new Promise(function(resolve, reject) {});
```

`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”
- `reject` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“失败”

5.2.1. 实例方法

`Promise` 构建出来的实例存在以下方法：

- then()
- catch()
- finally()

5.2.1.1. then()

`then` 是实例状态发生改变时的回调函数，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

`then` 方法返回的是一个新的 `Promise` 实例，也就是 `promise` 能链式书写的原因

JavaScript 复制代码

```
getJSON("/posts.json").then(function(json) {
  return json.post;
}).then(function(post) {
  // ...
});
```

5.2.1.2. catch

`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

JavaScript 复制代码

```
getJSON('/posts.json').then(function(posts) {
  // ...
}).catch(function(error) {
  // 处理 getJSON 和 前一个回调函数运行时发生的错误
  console.log('发生错误! ', error);
});
```

`Promise` 对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止

JavaScript 复制代码

```
getJSON('/post/1.json').then(function(post) {
  return getJSON(post.commentURL);
}).then(function(comments) {
  // some code
}).catch(function(error) {
  // 处理前面三个Promise产生的错误
});
```

一般来说，使用 `catch` 方法代替 `then()` 第二个参数

`Promise` 对象抛出的错误不会传递到外层代码，即不会有任何反应

```javascript
const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};
```

浏览器运行到这一行，会打印出错误提示 `ReferenceError: x is not defined` ，但是不会退出进程

`catch()` 方法之中，还能再抛出错误，通过后面 `catch` 方法捕获到

5.2.1.3. finally()

`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

```javascript
promise
.then(result => {···})
.catch(error => {···})
.finally(() => {···});
```

5.2.2. 构造函数方法

`Promise` 构造函数存在以下方法：

- all()
- race()
- allSettled()
- resolve()
- reject()
- try()

5.2.3. all()

`Promise.all()` 方法用于将多个 `Promise` 实例，包装成一个新的 `Promise` 实例

```
const p = Promise.all([p1, p2, p3]);
```

接受一个数组（迭代对象）作为参数，数组成员都应为 `Promise` 实例

实例 `p` 的状态由 `p1` 、 `p2` 、 `p3` 决定，分为两种：

- 只有 `p1` 、 `p2` 、 `p3` 的状态都变成 `fulfilled` ， `p` 的状态才会变成 `fulfilled` ，此时 `p1` 、 `p2` 、 `p3` 的返回值组成一个数组，传递给 `p` 的回调函数
- 只要 `p1` 、 `p2` 、 `p3` 之中有一个被 `rejected` ， `p` 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 `p` 的回调函数

注意，如果作为参数的 `Promise` 实例，自己定义了 `catch` 方法，那么它一旦被 `rejected` ，并不会触发 `Promise.all()` 的 `catch` 方法

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
})
.then(result => result)
.catch(e => e);

const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
})
.then(result => result)
.catch(e => e);

Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// ["hello", Error: 报错了]
```

如果 `p2` 没有自己的 `catch` 方法，就会调用 `Promise.all()` 的 `catch` 方法

```javascript
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
})
.then(result => result);

const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
})
.then(result => result);

Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// Error: 报错了
```

5.2.4. race()

`Promise.race()` 方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例

```javascript
const p = Promise.race([p1, p2, p3]);
```

只要 `p1` 、 `p2` 、 `p3` 之中有一个实例率先改变状态， `p` 的状态就跟着改变

率先改变的 Promise 实例的返回值则传递给 `p` 的回调函数

```javascript
const p = Promise.race([
  fetch('/resource-that-may-take-a-while'),
  new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error('request timeout')), 5000)
  })
]);

p
.then(console.log)
.catch(console.error);
```

5.2.5. allSettled()

`Promise.allSettled()` 方法接受一组 Promise 实例作为参数，包装成一个新的 Promise 实例

只有等到所有这些参数实例都返回结果，不管是 `fulfilled` 还是 `rejected` ，包装实例才会结束

JavaScript | 复制代码

```
const promises = [
  fetch('/api-1'),
  fetch('/api-2'),
  fetch('/api-3'),
];

await Promise.allSettled(promises);
removeLoadingIndicator();
```

5.2.5.1. resolve()

将现有对象转为 `Promise` 对象

JavaScript | 复制代码

```
Promise.resolve('foo')
// 等价于
new Promise(resolve => resolve('foo'))
```

参数可以分成四种情况，分别如下：

- 参数是一个 Promise 实例， `promise.resolve` 将不做任何修改、原封不动地返回这个实例
- 参数是一个 `thenable` 对象， `promise.resolve` 会将这个对象转为 `Promise` 对象，然后就立即执行 `thenable` 对象的 `then()` 方法
- 参数不是具有 `then()` 方法的对象，或根本就不是对象， `Promise.resolve()` 会返回一个新的 Promise 对象，状态为 `resolved`
- 没有参数时，直接返回一个 `resolved` 状态的 Promise 对象

5.2.5.2. reject()

`Promise.reject(reason)` 方法也会返回一个新的 Promise 实例，该实例的状态为 `rejected`

```javascript
const p = Promise.reject('出错了');
// 等同于
const p = new Promise((resolve, reject) => reject('出错了'))

p.then(null, function (s) {
  console.log(s)
});
// 出错了
```

`Promise.reject()` 方法的参数，会原封不动地变成后续方法的参数

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

5.3. 使用场景

将图片的加载写成一个 `Promise` ，一旦加载完成， `Promise` 的状态就发生变化

```javascript
const preloadImage = function (path) {
  return new Promise(function (resolve, reject) {
    const image = new Image();
    image.onload  = resolve;
    image.onerror = reject;
    image.src = path;
  });
};
```

通过链式操作，将多个渲染数据分别给个 `then` ，让其各司其职。或当下个异步请求依赖上个请求结果的时候，我们也能够通过链式操作友好解决问题

JavaScript 复制代码

```
// 各司其职
getInfo().then(res=>{
    let { bannerList } = res
    //渲染轮播图
    console.log(bannerList)
    return res
}).then(res=>{

    let { storeList } = res
    //渲染店铺列表
    console.log(storeList)
    return res
}).then(res=>{
    let { categoryList } = res
    console.log(categoryList)
    //渲染分类列表
    return res
})
```

通过 `all()` 实现多个请求合并在一起，汇总所有请求结果，只需设置一个 `loading` 即可

JavaScript 复制代码

```
function initLoad(){
    // loading.show() //加载loading
    Promise.all([getBannerList(),getStoreList(),getCategoryList()]).then(res=>{
        console.log(res)
        loading.hide() //关闭loading
    }).catch(err=>{
        console.log(err)
        loading.hide()//关闭loading
    })
}
//数据初始化
initLoad()
```

通过 `race` 可以设置图片请求超时

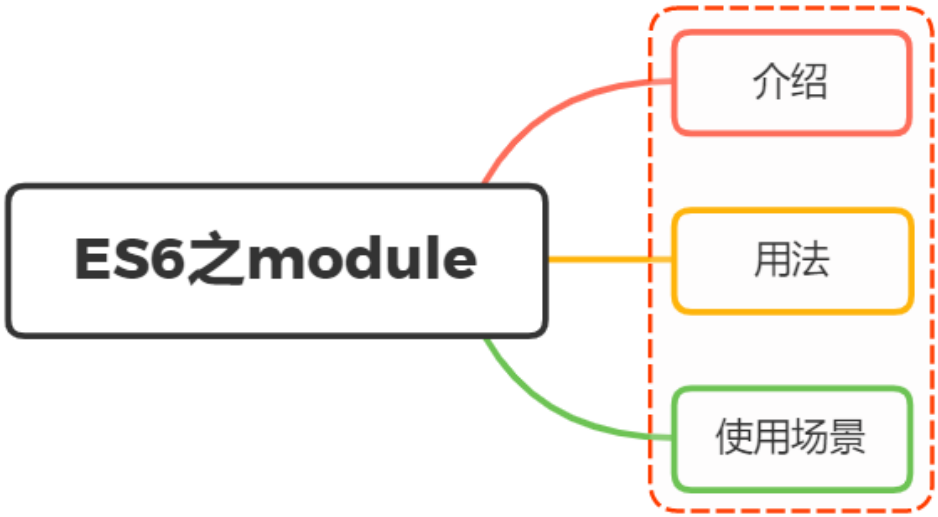
JavaScript | 复制代码

```javascript
//请求某个图片资源
function requestImg(){
    var p = new Promise(function(resolve, reject){
        var img = new Image();
        img.onload = function(){
           resolve(img);
        }
        //img.src = "https://b-gold-cdn.xitu.io/v3/static/img/logo.a7995ad.svg"; 正确的
        img.src = "https://b-gold-cdn.xitu.io/v3/static/img/logo.a7995ad.svg1";
    });
    return p;
}

//延时函数，用于给请求计时
function timeout(){
    var p = new Promise(function(resolve, reject){
        setTimeout(function(){
            reject('图片请求超时');
        }, 5000);
    });
    return p;
}

Promise
.race([requestImg(), timeout()])
.then(function(results){
    console.log(results);
})
.catch(function(reason){
    console.log(reason);
});
```

6. 你是怎么理解ES6中Module的？使用场景？



6.1. 介绍

模块，（Module），是能够单独命名并独立地完成一定功能的程序语句的**集合（即程序代码和数据结构的集合体）**。

两个基本的特征：外部特征和内部特征

- 外部特征是指模块跟外部环境联系的接口（即其他模块或程序调用该模块的方式，包括有输入输出参数、引用的全局变量）和模块的功能
- 内部特征是指模块的内部环境具有的特点（即该模块的局部数据和程序代码）

6.1.1. 为什么需要模块化

- 代码抽象
- 代码封装
- 代码复用
- 依赖管理

如果没有模块化，我们代码会怎样?

- 变量和方法不容易维护，容易污染全局作用域
- 加载资源的方式通过script标签从上到下。
- 依赖的环境主观逻辑偏重，代码较多就会比较复杂。
- 大型项目资源难以维护，特别是多人合作的情况下，资源的引入会让人奔溃

因此，需要一种将`JavaScript`程序模块化的机制，如

- CommonJs (典型代表：node.js早期)
- AMD (典型代表：require.js)
- CMD (典型代表：sea.js)

6.1.2. AMD

`Asynchronous ModuleDefinition` （AMD），异步模块定义，采用异步方式加载模块。所有依赖模块的语句，都定义在一个回调函数中，等到模块加载完成之后，这个回调函数才会运行

代表库为 `require.js`

JavaScript | 复制代码

```
/** main.js 入口文件/主模块 **/
// 首先用config()指定各模块路径和引用名
require.config({
  baseUrl: "js/lib",
  paths: {
    "jquery": "jquery.min",  //实际路径为js/lib/jquery.min.js
    "underscore": "underscore.min",
  }
});
// 执行基本操作
require(["jquery","underscore"],function($,_){
  // some code here
});
```

6.1.3. CommonJs

`CommonJS` 是一套 `Javascript` 模块规范，用于服务端

JavaScript | 复制代码

```
// a.js
module.exports={ foo , bar}

// b.js
const { foo,bar } = require('./a.js')
```

其有如下特点：

- 所有代码都运行在模块作用域，不会污染全局作用域
- 模块是同步加载的，即只有加载完成，才能执行后面的操作

- 模块在首次执行后就会缓存，再次加载只返回缓存结果，如果想要再次执行，可清除缓存
- `require` 返回的值是被输出的值的拷贝，模块内部的变化也不会影响这个值

既然存在了 `AMD` 以及 `CommonJs` 机制， `ES6` 的 `Module` 又有什么不一样?

ES6 在语言标准的层面上，实现了 `Module` ，即模块功能，完全可以取代 `CommonJS` 和 `AMD` 规范，成为浏览器和服务器通用的模块解决方案

`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

JavaScript | 复制代码

```
// 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` 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

JavaScript | 复制代码

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

上述代码，只加载3个方法，其他方法不加载，即 `ES6` 可以在编译时就完成模块加载

由于编译加载，使得静态分析成为可能。包括现在流行的 `typeScript` 也是依靠静态分析实现功能

6.2. 二、使用

`ES6` 模块内部自动采用了严格模式，这里就不展开严格模式的限制，毕竟这是 `ES5` 之前就已经规定好

模块功能主要由两个命令构成：

- `export` ： 用于规定模块的对外接口
- `import` ： 用于输入其他模块提供的功能

6.2.1. export

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

JavaScript | 复制代码

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

或
// 建议使用下面写法，这样能瞬间确定输出了哪些变量
var firstName = 'Michael';
var lastName = 'Jackson';
var year = 1958;

export { firstName, lastName, year };
```

输出函数或类

JavaScript | 复制代码

```
export function multiply(x, y) {
  return x * y;
};
```

通过 `as` 可以进行输出变量的重命名

JavaScript | 复制代码

```
function v1() { ... }
function v2() { ... }

export {
  v1 as streamV1,
  v2 as streamV2,
  v2 as streamLatestVersion
};
```

6.2.2. import

使用 `export` 命令定义了模块的对外接口以后，其他 JS 文件就可以通过 `import` 命令加载这个模块

```JavaScript
// main.js
import { firstName, lastName, year } from './profile.js';

function setName(element) {
  element.textContent = firstName + ' ' + lastName;
}
```

同样如果想要输入变量起别名，通过 `as` 关键字

```JavaScript
import { lastName as surname } from './profile.js';
```

当加载整个模块的时候，需要用到星号 `*`

```JavaScript
// 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(circle)   // {area:area,circumference:circumference}
```

输入的变量都是只读的，不允许修改，但是如果是对象，允许修改属性

```JavaScript
import {a} from './xxx.js'

a.foo = 'hello'; // 合法操作
a = {}; // Syntax Error : 'a' is read-only;
```

不过建议即使能修改，但我们不建议。因为修改之后，我们很难差错

`import` 后面我们常接着 `from` 关键字， `from` 指定模块文件的位置，可以是相对路径，也可以是绝对路径

```javascript
import { a } from './a';
```

如果只有一个模块名，需要有配置文件，告诉引擎模块的位置

```javascript
import { myMethod } from 'util';
```

在编译阶段，`import` 会提升到整个模块的头部，首先执行

```javascript
foo();

import { foo } from 'my_module';
```

多次重复执行同样的导入，只会执行一次

```javascript
import 'lodash';
import 'lodash';
```

上面的情况，大家都能看到用户在导入模块的时候，需要知道加载的变量名和函数，否则无法加载

如果不需要知道变量名或函数就完成加载，就要用到 `export default` 命令，为模块指定默认输出

```javascript
// export-default.js
export default function () {
    console.log('foo');
}
```

加载该模块的时候，`import` 命令可以为该函数指定任意名字

```javascript
// import-default.js
import customName from './export-default';
customName(); // 'foo'
```

6.2.3. 动态加载

允许您仅在需要时动态加载模块，而不必预先加载所有模块，这存在明显的性能优势

这个新功能允许您将 `import()` 作为函数调用，将其作为参数传递给模块的路径。 它返回一个 `promise` ，它用一个模块对象来实现，让你可以访问该对象的导出

JavaScript 复制代码

```
import('/modules/myModule.mjs')
  .then((module) => {
    // Do something with the module.
  });
```

6.2.4. 复合写法

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

JavaScript 复制代码

```
export { foo, bar } from 'my_module';

// 可以简单理解为
import { foo, bar } from 'my_module';
export { foo, bar };
```

同理能够搭配 `as` 、 `*` 搭配使用

6.3. 使用场景

如今， `ES6` 模块化已经深入我们日常项目开发中，像 `vue` 、 `react` 项目搭建项目，组件化开发处处可见，其也是依赖模块化实现

`vue` 组件

```
<template>
  <div class="App">
      组件化开发 ---- 模块化
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>
```

`react` 组件

```
function App() {
  return (
    <div className="App">
    组件化开发 ---- 模块化
    </div>
  );
}

export default App;
```

包括完成一些复杂应用的时候，我们也可以拆分成各个模块

7. 你是怎么理解ES6中 Generator的？使用场景？