目录

[Promise 2](#_Toc529733829)

[基础 2](#_Toc529733830)

[构造 2](#_Toc529733831)

[all 2](#_Toc529733832)

[resolve和reject 3](#_Toc529733833)

[.then和.catch 3](#_Toc529733834)

[class 4](#_Toc529733835)

[class和函数对比 4](#_Toc529733836)

[箭头函数 4](#_Toc529733837)

[@装饰器 4](#_Toc529733838)

[装饰类 4](#_Toc529733839)

[装饰类属性 5](#_Toc529733840)

[class 5](#_Toc529733841)

[static 5](#_Toc529733842)

[... 5](#_Toc529733843)

[模块化 6](#_Toc529733844)

[整理 6](#_Toc529733845)

[default 6](#_Toc529733846)

[规则 7](#_Toc529733847)

[原理 7](#_Toc529733848)

[export 7](#_Toc529733849)

[规则 7](#_Toc529733850)

[import 8](#_Toc529733851)

[\* as 8](#_Toc529733852)

[总结 9](#_Toc529733853)

[export default 9](#_Toc529733854)

[import()动态加载 9](#_Toc529733855)

[node中 10](#_Toc529733856)

[规则 10](#_Toc529733857)

[加载 10](#_Toc529733858)

[与commonjs模块混用 11](#_Toc529733859)

[其他 12](#_Toc529733860)

[复合写法 12](#_Toc529733861)

[对比commonjs 12](#_Toc529733862)

[html中使用 12](#_Toc529733863)

# Promise

## 基础

new的时候就直接执行其中的语句了。

setTimeOut要比promise慢很多。

promise里的resolve可以理解成一个标志位，下一步传回来的函数在这里调用。

promise对象：不可获取属性： PromiseStatus, PromiseValue。

使用promise时，promise外面的主线程还在往下面执行，promise变成resolve状态时再进行回调。（类比setTimeout）

### 构造

new Promise()

Promise.resolve() 把现有对象转换成Promise对象。

传入promise的函数可以是一个异步函数。

## all

返回的promise的PromiseValue是数组

## resolve和reject

正常情况下这两个函数要手动调用，比如通过if else.

.then中传入的是回调函数，会自动执行。

resolve: 普通数据或空值：转换成promise，数据作为resolve的参数。含有then的对象，转换后执行then方法。promise对象，不作修改。

Promise.reject()方法的参数，会原封不动地作为reject的理由，变成后续方法的参数。这一点与Promise.resolve方法不一致。

## .then和.catch

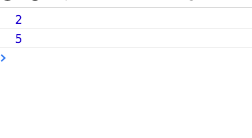
是同一类，可以理解catch是then的一半，所以在一条链上。

Promise.resolve(2).then((x)=>{console.log(x)})

.then(()=>{throw new Error('myerror');})

.catch((e)=>5)

.then((x)=>{console.log(x)})

.catch(e=>{console.log(e)})

catch以后，后面的then,catch都没有执行了，即catch不能链式调用。

then是传入的是一个函数，如果不是函数，这一层无效，继续传递前面返回的数据，执行下一层。

# generator

用于在同步函数里包裹异步代码。（比如）。

yield异步代码执行时，同步代码继续向下执行。

每次next返回yield执行后返回的值到迭代器的value。

Generator 函数也不能跟new命令一起用，会报错。

## 迭代器

遍历器是 Generator 函数的实例，也继承了 Generator 函数的prototype对象上的方法。但它是遍历器对象，而不是this对象。

# async

在async函数里面，遇到await异步函数时，同步代码跳出这个函数，继续往下执行。异步执行完以后，在继续执行函数里await后面的内容。

## 实现

一个自动执行的generator。（用的递归的方式）

# class

## class和函数对比

class没有变量提升。

函数不会覆盖变量。即存在内部函数和变量重名的情况时。

成员函数不能被枚举。

类的属性名，可以采用表达式。

## static

不会被实例继承

如果静态方法包含this关键字，这个this指的是类，而不是实例。

## constructor

b.constructor === B.prototype.constructor // true

## super

当做函数使用。指向父类的构造函数，但是super内部的this指向子类，相当于A.prototype.constructor.call(this)。

所以在react子组件的constructor中传入props能绑定到子组件上。

在普通方法中，指向父类的原型对象（即Father.prototype， 所以定义到实例上的方法无法调用，比如在constructor上才定义绑定到this上的）

在静态方法中，指向父类。

通过super调用父类的方法时，super会绑定子类的this。

# 箭头函数

this总是指向词法作用域，也就是定义时的外层

用call()或者apply()调用箭头函数时，无法对this进行绑定，即传入的第一个参数被忽略：

# @装饰器

修饰器只能用于类和类的方法，不能用于函数，因为存在函数提升。

## 装饰类

target

## 装饰类属性

target,,name(属性名),descriptor

特殊执行函数，在执行原始函数之前执行

// descriptor对象原来的值如下

// {

// value: specifiedFunction,这个函数，可以调用，通常加上apply(不确定参数时。）通过重新定义value去改造原函数。

// enumerable: false,

// configurable: true,

// writable: true

// };

两个target不同，前一个代表类本身，后一个代表类原型（prototype）（可以理解成未来的类的实例）

如果同一个方法有多个修饰器，会像剥洋葱一样，先从外到内进入，然后由内向外执行。

# 解构析构

不匹配不会报错，报错一般是因为格式问题：比如等号右边不能被遍历。

等号右边的值不是对象或数组，就先将其转为对象。由于undefined和null无法转为对象，所以对它们进行解构赋值，都会报错。

嵌套的不匹配会报错（因为父级为undefined）

嵌套的时候作为匹配模式（键）是不赋值的。

let a; {a} = {a:1} =>出错。因为行首的{}会被认为是代码块

let a;({a} ={a:1}) 用[]也可以

可以重复匹配：let {0:b, 0:c} = [0];

只有赋值语句的非模式部分，可以使用圆括号。即声明语句不行

## 解构析构

### 默认值

let [foo = true] = [];

var {x: y = 3} = {x: 5};

默认值：严格等于undefined，默认值才会生效。

默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

### 数组

### 对象

let { foo: baz } = { foo: 'aaa', bar: 'bbb' }; 键相同可以简写（es6对象的特性）。

### 字符串

字符串也可以解构赋值。这是因为此时，字符串被转换成了一个类似数组的对象。

还可以对length结构赋值

### 函数参数

## ...扩展

可以把对象变成几个并列的值或对象。

不能随便用，只能在特殊场景下用。console.log里可以用的场景不代表所有场景都能用。

展开后有相同的key时会覆盖，展开到数组里不会（序号自动添加）。

### 不转换

对象，数组里可以，字符串不可以。

### 转换

数组可以展开到数组，字符串里。

对象展开到数组，对象里。

数组里展开对象 => 出错。（因为数组肯定是对象，对象不一定是数组）

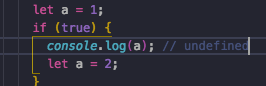
对象展开到字符串里 => 出错。

字符串展开到数组，对象里（对象里和数组一样，序号作为key）

函数参数里：可以展开数组，不能展开对象。

## ...rest

# 作用域



# 模块化

ES6 模块的设计思想，是尽量的静态化（**因为导出的只是一些静态的定义，所以解析时就能确定依赖关系，可以通过这个方式判断导出有没有写对**），使得**编译时(require是执行到require时，运行其中的代码)**就能确定模块的依赖关系，以及输入和输出的变量。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。

这种加载称为“编译时加载”或者静态加载，即 ES6 可以在编译时就完成模块加载，效率要比 CommonJS 模块的加载方式高。当然，这也导致了没法引用 ES6 模块本身，因为它不是对象。

## 整理

可以export一段代码，这段代码申明了一个变量或者函数。也可以申明变量，然后放入export的对象里。

这两种方式可以同时使用(不能对同一个变量使用两种方式)

export const a = 1;

等效于

const a = 1;

export {

*a*

}

### default

一个特殊的变量。

export const a = 2;

export {

*a* as default

}

等效于

const a = 2;

export default a;

### 规则

通过export命令显式指定**需要输出**的代码，再通过import命令输入。

import命令会被 JavaScript 引擎静态分析，先于模块内的其他模块执行。

### 原理

导出输入的都只是一些静态的定义，也就是解析时确定的那些变量名。

输入的模块变量，只是一个“符号连接”，所以这个变量是只读的，对它进行重新赋值会报错。所以不同的脚本加载这个接口，得到的都是同样的实例。

会执行引入的js文件（多次引用执行一次），import的文件会提前执行。**优先级在函数提升后面（函数提升在代码执行之前）。**

## export

### 规则

\*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。

export {

v1 as streamV1,

};

\*export命令规定的是对外的接口，必须与模块内部的变量建立一一对应关系。

export var m = 1 对。

var m = 1; export m; 错

function f() {} export f; 错

\*export语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块**内部实时的值**。

\*export命令需要位于顶层，如果处于块级作用域内，就会报错，下一节的import命令也是如此。这是因为处于条件代码块之中，就没法做静态优化了，违背了 ES6 模块的设计初衷。

## import

也可以用as

import后面的from指定模块文件的位置，.js后缀可以省略。如果只是模块名(第三方插件之类)，不带有路径，那么必须有配置文件，告诉 JavaScript 引擎该模块的位置。

由于import是静态执行，所以不能使用表达式和变量(包括if这些需要运行时才能确定的)，这些只有在运行时才能得到结果的语法结构。

import语句会执行所加载的模块

import 'lodash';

上面代码仅仅执行lodash模块，但是不输入任何值。

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。

### \* as

\* as 后的对象不允许在运行时改变。

如果包含default,x.default可以，直接x.a不行。

import \* as x from '../config/request';

*console*.log(x.default);// 3

*console*.log(x.b) // 报错

## 总结

### export default

接口和代码块不需要一一对应。

import时不需要大括号

本质上，export default就是输出一个叫做default的变量或方法，然后系统允许你为它取任意名字。

因为export default命令其实只是输出一个叫做default的变量，所以它后面不能跟变量声明语句。

export default var a = 1;错

### import()动态加载

提案中。

import命令能够接受什么参数，import()函数就能接受什么参数，两者区别主要是后者为动态加载。import()返回一个 Promise 对象。

import()类似于 Node 的require方法，区别主要是前者是异步加载，后者是同步加载。

## node中

### 规则

ES6 模块之中，顶层的this指向undefined；CommonJS 模块的顶层this指向当前模块。其次，以下这些顶层变量在 ES6 模块之中都是不存在的。

arguments，require，module，exports，\_\_filename，\_\_dirname。

Node 的import命令是异步加载，这一点与浏览器(浏览器的script标签，不是js中的输入)的处理方法相同。

Node 要求 ES6 模块采用.mjs后缀文件名。require命令不能加载.mjs文件，会报错，只有import命令才可以加载.mjs文件。反过来，.mjs文件里面也不能使用require命令，必须使用import。import命令只支持加载本地模块（file:协议），不支持加载远程模块。**但是import可以加载js 文件**

### 加载

如果模块名不含路径，那么import命令会去node\_modules目录寻找这个模块。

如果脚本文件省略了后缀名，比如import './foo'，Node 会依次尝试四个后缀名：./foo.mjs、./foo.js、./foo.json、./foo.node。如果这些脚本文件都不存在，Node 就会去加载./foo/package.json的main字段指定的脚本。如果./foo/package.json不存在或者没有main字段，那么就会依次加载./foo/index.mjs、./foo/index.js、./foo/index.json、./foo/index.node。如果以上四个文件还是都不存在，就会抛出错误。

### 与commonjs模块混用

#### es6加载commonjs模块。

Node 会自动将module.exports属性，当作模块的默认输出，即等同于export default {}。

// 例子1 写法三

import \* as baz from './a';

// baz = {

// get default() {return module.exports;},

// get foo() {return this.default.foo}.bind(baz),

// get bar() {return this.default.bar}.bind(baz)

// }

这种写法要baz.default才能得到module.exports

// 例子2 正确的写法一

import \* as express from 'express';

const app = express.default();

// 正确的写法二

import express from 'express';

const app = express();

#### commonjs加载es6模块

CommonJS 模块加载 ES6 模块，不能使用require命令，而要使用import()函数。ES6 模块的所有输出接口，会成为输入对象的属性。

## 其他

### 复合写法

export { foo, bar } from 'my\_module';

// 等同于

import { foo, bar } from 'my\_module';

export { foo, bar };

export \* from命令会忽略模块的default方法。

### 对比commonjs

CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。所以es6得到实时的值。commonjs得到加载时的值。

CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

### html中使用

<script type="module" src="./foo.js"></script>

<script type="module">

import utils from "./utils.js";

// other code

</script>

# proxy

