当 webpack 处理应用程序时，它会递归地构建一个依赖关系图 (dependency graph) ，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 bundle 。

webpack 主要有四个核心概念:

* 入口 (entry)
* 输出 (output)
* loader
* 插件 (plugins)

变量声明：

let 声明的变量只在 let 命令所在的代码块内有效。

注意：**不能重复声明**

let 只能声明一次， var 可以声明多次。

**不存在变量提升**

let 不存在变量提升，var 会变量提升。

const 声明一个只读的常量，一旦声明，常量的值就不能改变。

var 是在全局范围内有效。

#### const 命令

const 声明一个只读变量，声明之后不允许改变。意味着，一旦声明必须初始化，否则会报错。

ES6 明确规定，代码块内如果存在 let 或者 const，代码块会对这些命令声明的变量从块的开始就形成一个封闭作用域。代码块内，在声明变量 PI 之前使用它会报错

**注意要点**

const 如何做到变量在声明初始化之后不允许改变的？其实 const 其实保证的不是变量的值不变，而是保证变量指向的内存地址所保存的数据不允许改动。此时，你可能已经想到，简单类型和复合类型保存值的方式是不同的。是的，对于简单类型（数值 number、字符串 string 、布尔值 boolean）,值就保存在变量指向的那个内存地址，因此 const 声明的简单类型变量等同于常量。而复杂类型（对象 object，数组 array，函数 function），变量指向的内存地址其实是保存了一个指向实际数据的指针，所以 const 只能保证指针是固定的，至于指针指向的数据结构变不变就无法控制了，所以使用 const 声明复杂类型对象时要慎重。

#### 结构模型

解构的源，解构赋值表达式的右边部分。

* 解构的目标，解构赋值表达式的左边部分。

#### ES6的数据模型：

ES6 数据类型除了 Number 、 String 、 Boolean 、 Objec t、 null 和 undefined ，还新增了 Symbol 。

Symbol ，表示独一无二的值，最大的用法是用来定义对象的唯一属性名

注意：

Symbol 作为对象属性名时不能用.运算符，要用方括号。因为.运算符后面是字符串，所以取到的是字符串 sy 属性，而不是 Symbol 值 sy 属性。

### 注意点

Symbol 值作为属性名时，该属性是公有属性不是私有属性，可以在类的外部访问。但是不会出现在 for...in 、 for...of 的循环中，也不会被 Object.keys() 、 Object.getOwnPropertyNames() 返回。如果要读取到一个对象的 Symbol 属性，可以通过 Object.getOwnPropertySymbols() 和 Reflect.ownKeys() 取到。

Symbol 的值是唯一的，所以不会出现相同值得常量，即可以保证 switch 按照代码预想的方式执行。

##### Symbol.for()

Symbol.for() 类似单例模式，首先会在全局搜索被登记的 Symbol 中是否有该字符串参数作为名称的 Symbol 值，如果有即返回该 Symbol 值，若没有则新建并返回一个以该字符串参数为名称的 Symbol 值，并登记在全局环境中供搜索。

##### Symbol.keyFor()

Symbol.keyFor() 返回一个已登记的 Symbol 类型值的 key ，用来检测该字符串参数作为名称的 Symbol 值是否已被登记

#### Maps 和 Objects 的区别

* 一个 Object 的键只能是字符串或者 Symbols，但一个 Map 的键可以是任意值。
* Map 中的键值是有序的（FIFO 原则），而添加到对象中的键则不是。
* Map 的键值对个数可以从 size 属性获取，而 Object 的键值对个数只能手动计算。
* Object 都有自己的原型，原型链上的键名有可能和你自己在对象上的设置的键名产生冲突。

forEach()

var myMap = new Map(); myMap.set(0, "zero"); myMap.set(1, "one"); // 将会显示两个 logs。 一个是 "0 = zero" 另一个是 "1 = one" myMap.forEach(function(value, key) { console.log(key + " = " + value); }, myMap)

#### Set 对象

Set 对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用。

##### Set 中的特殊值

Set 对象存储的值总是唯一的，所以需要判断两个值是否恒等。有几个特殊值需要特殊对待：

* +0 与 -0 在存储判断唯一性的时候是恒等的，所以不重复；
* undefined 与 undefined 是恒等的，所以不重复；
* NaN 与 NaN 是不恒等的，但是在 Set 中只能存一个，不重复。

#### Proxy

一个 Proxy 对象由两个部分组成： target 、 handler 。在通过 Proxy 构造函数生成实例对象时，需要提供这两个参数。 target 即目标对象， handler 是一个对象，声明了代理 target 的指定行为。

#### 字符串

* **includes()**：返回布尔值，判断是否找到参数字符串。
* **startsWith()**：返回布尔值，判断参数字符串是否在原字符串的头部。
* **endsWith()**：返回布尔值，判断参数字符串是否在原字符串的尾部。

以上三个方法都可以接受两个参数，需要搜索的字符串，和可选的搜索起始位置索引。

###### 字符串重复

repeat()：返回新的字符串，表示将字符串重复指定次数返回。

console.log("Hello,".repeat(2)); // "Hello,Hello,"

字符串补全

* **padStart**：返回新的字符串，表示用参数字符串从头部（左侧）补全原字符串。
* **padEnd**：返回新的字符串，表示用参数字符串从尾部（右侧）补全原字符串。
* 以上两个方法接受两个参数，第一个参数是指定生成的字符串的最小长度，第二个参数是用来补全的字符串。如果没有指定第二个参数，默认用空格填充。
* console.log("h".padStart(5,"o")); // "ooooh"
* console.log("h".padEnd(5,"o")); // "hoooo"
* console.log("h".padStart(5)); // " h"

模板字符串

模板字符串相当于加强版的字符串，用反引号 **`**,除了作为普通字符串，还可以用来定义多行字符串，还可以在字符串中加入变量和表达式。

字符串插入变量和表达式。

变量名写在 ${} 中，${} 中可以放入 JavaScript 表达式。

### 数字处理

Math.trunc

Math.fround

**普通计算**

Math.cbrt

Math.imul

Math.hypot

Math.clz32

### 判断

Math.sign

### 判断

Math.sign

### 双曲函数方法

* Math.sinh(x): 用于计算双曲正弦。
* Math.cosh(x): 用于计算双曲余弦。
* Math.tanh(x): 用于计算双曲正切。
* Math.asinh(x): 用于计算反双曲正弦。
* Math.acosh(x): 用于计算反双曲余弦。
* Math.atanh(x): 用于计算反双曲正切。

属性名表达式

* ES6允许用表达式作为属性名，但是一定要将表达式放在方括号内。
* const obj = { ["he"+"llo"](){ return "Hi"; } } obj.hello(); //"Hi"
* 注意点：属性的简洁表示法和属性名表达式不能同时使用，否则会报错。

注意点:

自定义的属性和拓展运算符对象里面属性的相同的时候：**自定义的属性在拓展运算符后面，则拓展运算符对象内部同名的属性将被覆盖掉。**

## 数组创建

### Array.of()

将参数中所有值作为元素形成数组。

console.log(Array.of(1, 2, 3, 4)); // [1, 2, 3, 4] // 参数值可为不同类型 console.log(Array.of(1, '2', true)); // [1, '2', true] // 参数为空时返回空数组 console.log(Array.of()); // []

### Array.from()

将类数组对象或可迭代对象转化为数组。

// 参数为数组,返回与原数组一样的数组 console.log(Array.from([1, 2])); // [1, 2] // 参数含空位 console.log(Array.from([1, , 3])); // [1, undefined, 3]

**参数**

Array.from(arrayLike[, mapFn[, thisArg]])

返回值为转换后的数组。

**arrayLike**

想要转换的类数组对象或可迭代对象。

console.log(Array.from([1, 2, 3])); // [1, 2, 3]

**mapFn**

可选，map 函数，用于对每个元素进行处理，放入数组的是处理后的元素。

console.log(Array.from([1, 2, 3], (n) => n \* 2)); // [2, 4, 6]

**thisArg**

可选，用于指定 map 函数执行时的 this 对象。

let map = { do: function(n) { return n \* 2; } } let arrayLike = [1, 2, 3]; console.log(Array.from(arrayLike, function (n){ return this.do(n); }, map)); // [2, 4, 6]

### 类数组对象

一个类数组对象必须含有 length 属性，且元素属性名必须是数值或者可转换为数值的字符。

let arr = Array.from({ 0: '1', 1: '2', 2: 3, length: 3 }); console.log(); // ['1', '2', 3] // 没有 length 属性,则返回空数组 let array = Array.from({ 0: '1', 1: '2', 2: 3, }); console.log(array); // [] // 元素属性名不为数值且无法转换为数值，返回长度为 length 元素值为 undefined 的数组 let array1 = Array.from({ a: 1, b: 2, length: 2 }); console.log(array1); // [undefined, undefined]

### 转换可迭代对象

**转换 map**

let map = new Map(); map.set('key0', 'value0'); map.set('key1', 'value1'); console.log(Array.from(map)); // [['key0', 'value0'],['key1', // 'value1']]

**转换 set**

let arr = [1, 2, 3]; let set = new Set(arr); console.log(Array.from(set)); // [1, 2, 3]

**转换字符串**

let str = 'abc'; console.log(Array.from(str)); // ["a", "b", "c"]

## 扩展的方法

### 查找

**find()**

查找数组中符合条件的元素,若有多个符合条件的元素，则返回第一个元素。

let arr = Array.of(1, 2, 3, 4); console.log(arr.find(item => item > 2)); // 3 // 数组空位处理为 undefined console.log([, 1].find(n => true)); // undefined

**findIndex()**

查找数组中符合条件的元素索引，若有多个符合条件的元素，则返回第一个元素索引。

let arr = Array.of(1, 2, 1, 3); // 参数1：回调函数 // 参数2(可选)：指定回调函数中的 this 值 console.log(arr.findIndex(item => item = 1)); // 0 // 数组空位处理为 undefined console.log([, 1].findIndex(n => true)); //0

### 填充

**fill()**

将一定范围索引的数组元素内容填充为单个指定的值。

let arr = Array.of(1, 2, 3, 4); // 参数1：用来填充的值 // 参数2：被填充的起始索引 // 参数3(可选)：被填充的结束索引，默认为数组末尾 console.log(arr.fill(0,1,2)); // [1, 0, 3, 4]

**copyWithin()**

将一定范围索引的数组元素修改为此数组另一指定范围索引的元素。

// 参数1：被修改的起始索引 // 参数2：被用来覆盖的数据的起始索引 // 参数3(可选)：被用来覆盖的数据的结束索引，默认为数组末尾 console.log([1, 2, 3, 4].copyWithin(0,2,4)); // [3, 4, 3, 4] // 参数1为负数表示倒数 console.log([1, 2, 3, 4].copyWithin(-2, 0)); // [1, 2, 1, 2] console.log([1, 2, ,4].copyWithin(0, 2, 4)); // [, 4, , 4]

### 遍历

**entries()**

遍历键值对。

for(let [key, value] of ['a', 'b'].entries()){ console.log(key, value); } // 0 "a" // 1 "b" // 不使用 for... of 循环 let entries = ['a', 'b'].entries(); console.log(entries.next().value); // [0, "a"] console.log(entries.next().value); // [1, "b"] // 数组含空位 console.log([...[,'a'].entries()]); // [[0, undefined], [1, "a"]]

**keys()**

遍历键名。

for(let key of ['a', 'b'].keys()){ console.log(key); } // 0 // 1 // 数组含空位 console.log([...[,'a'].keys()]); // [0, 1]

**values()**

遍历键值。

for(let value of ['a', 'b'].values()){ console.log(value); } // "a" // "b" // 数组含空位 console.log([...[,'a'].values()]); // [undefined, "a"]

### 包含

**includes()**

数组是否包含指定值。

注意：与 Set 和 Map 的 has 方法区分；Set 的 has 方法用于查找值；Map 的 has 方法用于查找键名。

// 参数1：包含的指定值 [1, 2, 3].includes(1); // true // 参数2：可选，搜索的起始索引，默认为0 [1, 2, 3].includes(1, 2); // false // NaN 的包含判断 [1, NaN, 3].includes(NaN); // true

### 嵌套数组转一维数组

**flat()**

console.log([1 ,[2, 3]].flat()); // [1, 2, 3] // 指定转换的嵌套层数 console.log([1, [2, [3, [4, 5]]]].flat(2)); // [1, 2, 3, [4, 5]] // 不管嵌套多少层 console.log([1, [2, [3, [4, 5]]]].flat(Infinity)); // [1, 2, 3, 4, 5] // 自动跳过空位 console.log([1, [2, , 3]].flat());<p> // [1, 2, 3]

**flatMap()**

先对数组中每个元素进行了的处理，再对数组执行 flat() 方法。

// 参数1：遍历函数，该遍历函数可接受3个参数：当前元素、当前元素索引、原数组 // 参数2：指定遍历函数中 this 的指向 console.log([1, 2, 3].flatMap(n => [n \* 2])); // [2, 4, 6]

## 数组缓冲区

数组缓冲区是内存中的一段地址。

定型数组的基础。

实际字节数在创建时确定，之后只可修改其中的数据，不可修改大小。

### 创建数组缓冲区

通过构造函数创建:

let buffer = new ArrayBuffer(10); console.log(buffer.byteLength); // 10 分割已有数组缓冲区 let buffer = new ArrayBuffer(10); let buffer1 = buffer.slice(1, 3); console.log(buffer1.byteLength); // 2

### 视图

视图是用来操作内存的接口。

视图可以操作数组缓冲区或缓冲区字节的子集,并按照其中一种数值数据类型来读取和写入数据。

DataView 类型是一种通用的数组缓冲区视图,其支持所有8种数值型数据类型。

创建:

// 默认 DataView 可操作数组缓冲区全部内容 let buffer = new ArrayBuffer(10); dataView = new DataView(buffer); dataView.setInt8(0,1); console.log(dataView.getInt8(0)); // 1 // 通过设定偏移量(参数2)与长度(参数3)指定 DataView 可操作的字节范围 let buffer1 = new ArrayBuffer(10); dataView1 = new DataView(buffer1, 0, 3); dataView1.setInt8(5,1); // RangeError

## 定型数组

数组缓冲区的特定类型的视图。

可以强制使用特定的数据类型，而不是使用通用的 DataView 对象来操作数组缓冲区。

### 创建

通过数组缓冲区生成

let buffer = new ArrayBuffer(10), view = new Int8Array(buffer); console.log(view.byteLength); // 10

通过构造函数

let view = new Int32Array(10); console.log(view.byteLength); // 40 console.log(view.length); // 10 // 不传参则默认长度为0 // 在这种情况下数组缓冲区分配不到空间，创建的定型数组不能用来保存数据 let view1 = new Int32Array(); console.log(view1.byteLength); // 0 console.log(view1.length); // 0 // 可接受参数包括定型数组、可迭代对象、数组、类数组对象 let arr = Array.from({ 0: '1', 1: '2', 2: 3, length: 3 }); let view2 = new Int16Array([1, 2]), view3 = new Int32Array(view2), view4 = new Int16Array(new Set([1, 2, 3])), view5 = new Int16Array([1, 2, 3]), view6 = new Int16Array(arr); console.log(view2 .buffer === view3.buffer); // false console.log(view4.byteLength); // 6 console.log(view5.byteLength); // 6 console.log(view6.byteLength); // 6

### 注意要点

length 属性不可写，如果尝试修改这个值，在非严格模式下会直接忽略该操作，在严格模式下会抛出错误。

let view = new Int16Array([1, 2]); view.length = 3; console.log(view.length); // 2

定型数组可使用 entries()、keys()、values()进行迭代。

let view = new Int16Array([1, 2]); for(let [k, v] of view.entries()){ console.log(k, v); } // 0 1 // 1 2

find() 等方法也可用于定型数组，但是定型数组中的方法会额外检查数值类型是否安全,也会通过 Symbol.species 确认方法的返回值是定型数组而非普通数组。concat() 方法由于两个定型数组合并结果不确定，故不能用于定型数组；另外，由于定型数组的尺寸不可更改,可以改变数组的尺寸的方法，例如 splice() ，不适用于定型数组。

let view = new Int16Array([1, 2]); view.find((n) > 1); // 2

所有定型数组都含有静态 of() 方法和 from() 方法,运行效果分别与 Array.of() 方法和 Array.from() 方法相似,区别是定型数组的方法返回定型数组,而普通数组的方法返回普通数组。

let view = Int16Array.of(1, 2); console.log(view instanceof Int16Array); // true

定型数组不是普通数组，不继承自 Array 。

let view = new Int16Array([1, 2]); console.log(Array.isArray(view)); // false

定型数组中增加了 set() 与 subarray() 方法。 set() 方法用于将其他数组复制到已有定型数组, subarray() 用于提取已有定型数组的一部分形成新的定型数组。

// set 方法 // 参数1：一个定型数组或普通数组 // 参数2：可选，偏移量，开始插入数据的位置，默认为0 let view= new Int16Array(4); view.set([1, 2]); view.set([3, 4], 2); console.log(view); // [1, 2, 3, 4] // subarray 方法 // 参数1：可选，开始位置 // 参数2：可选，结束位置(不包含结束位置) let view= new Int16Array([1, 2, 3, 4]), subview1 = view.subarray(), subview2 = view.subarray(1), subview3 = view.subarray(1, 3); console.log(subview1); // [1, 2, 3, 4] console.log(subview2); // [2, 3, 4] console.log(subview3); // [2, 3]

## 扩展运算符

### 复制数组

let arr = [1, 2], arr1 = [...arr]; console.log(arr1); // [1, 2] // 数组含空位 let arr2 = [1, , 3], arr3 = [...arr2]; console.log(arr3); [1, undefined, 3]

合并数组

console.log([...[1, 2],...[3, 4]]); // [1, 2, 3, 4]

## 函数参数的扩展

### 默认参数

基本用法

function fn(name,age=17){ console.log(name+","+age); } fn("Amy",18); // Amy,18 fn("Amy",""); // Amy, fn("Amy"); // Amy,17

注意点：使用函数默认参数时，不允许有同名参数。

// 不报错 function fn(name,name){ console.log(name); } // 报错 //SyntaxError: Duplicate parameter name not allowed in this context function fn(name,name,age=17){ console.log(name+","+age); }

只有在未传递参数，或者参数为 undefined 时，才会使用默认参数，null 值被认为是有效的值传递

函数参数默认值存在暂时性死区，在函数参数默认值表达式中，还未初始化赋值的参数值无法作为其他参数的默认值

### 不定参数

不定参数用来表示不确定参数个数，形如，...变量名，由...加上一个具名参数标识符组成。具名参数只能放在参数组的最后，并且有且只有一个不定参数。

### 箭头函数

箭头函数提供了一种更加简洁的函数书写方式。基本语法是：

参数 => 函数体

基本用法：

var f = v => v;

//等价于 var f = function(a)

{

return a;

}

f(1); //1

当箭头函数没有参数或者有多个参数，要用 **()** 括起来。

var f = (a,b) => a+b; f(6,2); //8

当箭头函数函数体有多行语句，用 **{}** 包裹起来，表示代码块，当只有一行语句，并且需要返回结果时，可以省略 **{}** , 结果会自动返回。

var f = (a,b) => { let result = a+b; return result; }

f(6,2); // 8

当箭头函数要返回对象的时候，为了区分于代码块，要用 **()** 将对象包裹起来

// 报错 var f = (id,name) => {id: id, name: name}; f(6,2); // SyntaxError: Unexpected token : // 不报错 var f = (id,name) => ({id: id, name: name}); f(6,2); // {id: 6, name: 2}

注意点：没有 this、super、arguments 和 new.target 绑定。

var func = () => { // 箭头函数里面没有 this 对象， // 此时的 this 是外层的 this 对象，即 Window console.log(this) } func(55) // Window var func = () => { console.log(arguments) } func(55); // ReferenceError: arguments is not defined

箭头函数体中的 this 对象，是定义函数时的对象，而不是使用函数时的对象。

function fn(){ setTimeout(()=>{ // 定义时，this 绑定的是 fn 中的 this 对象 console.log(this.a); },0) } var a = 20; // fn 的 this 对象为 {a: 19} fn.call({a: 18}); // 18

不可以作为构造函数，也就是不能使用 new 命令，否则会报错

### Iterator

Iterator 是 ES6 引入的一种新的遍历机制，迭代器有两个核心概念：

* 迭代器是一个统一的接口，它的作用是使各种数据结构可被便捷的访问，它是通过一个键为Symbol.iterator 的方法来实现。
* 迭代器是用于遍历数据结构元素的指针（如数据库中的游标）。

### 迭代过程

迭代的过程如下：

* 通过 Symbol.iterator 创建一个迭代器，指向当前数据结构的起始位置
* 随后通过 next 方法进行向下迭代指向下一个位置， next 方法会返回当前位置的对象，对象包含了 value 和 done 两个属性， value 是当前属性的值， done 用于判断是否遍历结束
* 当 done 为 true 时则遍历结束

**可迭代的数据结构**

以下是可迭代的值:

* Array
* String
* Map
* Set
* Dom元素（正在进行中）

### 普通对象不可迭代

普通对象是由 object 创建的，不可迭代：

// TypeError for (let item of {}) { console.log(item); }

**Map**

let myMap = new Map();

myMap.set(0, "zero");

myMap.set(1, "one");

myMap.set(2, "two");

// 遍历 key 和 value

for (let [key, value] of myMap)

{ console.log(key + " = " + value); }

for (let [key, value] of myMap.entries())

{ console.log(key + " = " + value); }

// 只遍历 key

for (let key of myMap.keys()) { console.log(key); }

// 只遍历 value

for (let value of myMap.values()) { console.log(value); }

### 类定义

类表达式可以为匿名或命名。

// 匿名类 let Example = class { constructor(a) { this.a = a; } } // 命名类 let Example = class Example { constructor(a) { this.a = a; } }

### 类声明

class Example { constructor(a) { this.a = a; } }

注意要点：不可重复声明。

class Example{} class Example{} // Uncaught SyntaxError: Identifier 'Example' has already been // declared let Example1 = class{} class Example{} // Uncaught SyntaxError: Identifier 'Example' has already been // declared

### 注意要点

类定义不会被提升，这意味着，必须在访问前对类进行定义，否则就会报错。

类中方法不需要 function 关键字。

方法间不能加分号。

new Example(); class Example {}

静态属性

静态属性：class 本身的属性，即直接定义在类内部的属性（ Class.propname ），不需要实例化。 ES6 中规定，Class 内部只有静态方法，没有静态属性。

class Example { // 新提案 static a = 2; }

// 目前可行写法

Example.b = 2;

公共属性

class Example{}

Example.prototype.a = 2;

实例属性

实例属性：定义在实例对象（ this ）上的属性。

class Example { a = 2; constructor () { console.log(this.a); } }

name 属性

返回跟在 class 后的类名(存在时)。

let Example=class Exam { constructor(a) { this.a = a; } }

console.log(Example.name); // Exam

let Example=class { constructor(a) { this.a = a; } }

console.log(Example.name); // Example

**方法**

constructor 方法

constructor 方法是类的默认方法，创建类的实例化对象时被调用。

class Example{ constructor(){ console.log('我是constructor'); } }

new Example(); // 我是constructor

## decorator

decorator 是一个函数，用来修改类的行为，在代码编译时产生作用。

### 类修饰

一个参数

第一个参数 target，指向类本身。

function testable(target) { target.isTestable = true; } @testable class Example {} Example.isTestable; // true

多个参数——嵌套实现

function testable(isTestable) { return function(target) { target.isTestable=isTestable; } } @testable(true) class Example {} Example.isTestable; // true

实例属性

上面两个例子添加的是静态属性，若要添加实例属性，在类的 prototype 上操作即可。

### 方法修饰

3个参数：target（类的原型对象）、name（修饰的属性名）、descriptor（该属性的描述对象）。

class Example { @writable sum(a, b) { return a + b; } } function writable(target, name, descriptor) { descriptor.writable = false; return descriptor; // 必须返回 }

修饰器执行顺序

由外向内进入，由内向外执行。

class Example { @logMethod(1) @logMthod(2) sum(a, b){ return a + b; } } function logMethod(id) { console.log('evaluated logMethod'+id); return (target, name, desctiptor) => console.log('excuted logMethod '+id); } // evaluated logMethod 1 // evaluated logMethod 2 // excuted logMethod 2 // excuted logMethod 1

### 方法修饰

3个参数：target（类的原型对象）、name（修饰的属性名）、descriptor（该属性的描述对象）。

class Example { @writable sum(a, b) { return a + b; } } function writable(target, name, descriptor) { descriptor.writable = false; return descriptor; // 必须返回 }

### extends

通过 extends 实现类的继承。

class Child extends Father { ... }

调用父类构造函数,只能出现在子类的构造函数。

调用父类方法, super 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 Child2 extends Father { constructor(){ super(); // 调用父类普通方法 console.log(super.test()); // 0 } static test3(){ // 调用父类静态方法 return super.test1+2; } } Child2.test3(); // 3

## ES6模块化

### export 与 import

### 基本用法

模块导入导出各种类型的变量，如字符串，数值，函数，类。

* 导出的函数声明与类声明必须要有名称（export default 命令另外考虑）。
* 不仅能导出声明还能导出引用（例如函数）。
* export 命令可以出现在模块的任何位置，但必需处于模块顶层。
* import 命令会提升到整个模块的头部，首先执行。

### as 的用法

export 命令导出的接口名称，须和模块内部的变量有一一对应关系。

导入的变量名，须和导出的接口名称相同，即顺序可以不一致。

/\*-----export [test.js]-----\*/

let myName = "Tom";

export { myName as exportName }

/\*-----import [xxx.js]-----\*/

import { exportName } from "./test.js";

console.log(exportName);// Tom

使用 as 重新定义导出的接口名称，隐藏模块内部的变量

/\*-----export [test1.js]-----\*/

let myName = "Tom"; export { myName }

/\*-----export [test2.js]-----\*/

let myName = "Jerry"; export { myName }

/\*-----import [xxx.js]-----\*/

import { myName as name1 } from "./test1.js";

import { myName as name2 } from "./test2.js";

console.log(name1);// Tom

console.log(name2);// Jerry

不同模块导出接口名称命名重复， 使用 as 重新定义变量名。

### import 命令的特点

**只读属性**：不允许在加载模块的脚本里面，改写接口的引用指向，即可以改写 import 变量类型为对象的属性值，不能改写 import 变量类型为基本类型的值。

**单例模式**：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。import 同一模块，声明不同接口引用，会声明对应变量，但只执行一次 import 。

静态执行特性：import 是静态执行，所以不能使用表达式和变量。

import { "f" + "oo" } from "methods"; // error

let module = "methods";

import { foo } from module; // error

if (true) { import { foo } from "method1"; }

else { import { foo } from "method2"; } // error

**export default 命令**

* 在一个文件或模块中，export、import 可以有多个，export default 仅有一个。
* export default 中的 default 是对应的导出接口变量。
* 通过 export 方式导出，在导入时要加{ }，export default 则不需要。
* export default 向外暴露的成员，可以使用任意变量来接收。

var a = "My name is Tom!"; export default a; // 仅有一个 export default var c = "error"; // error，default 已经是对应的导出变量，不能跟着变量声明语句 import b from "./xxx.js"; // 不需要加{}， 使用任意变量接收

## 复合使用

***注****：import() 是提案，这边暂时不延伸讲解。*

export 与 import 可以在同一模块使用，使用特点：

* 可以将导出接口改名，包括 default。
* 复合使用 export 与 import ，也可以导出全部，当前模块导出的接口会覆盖继承导出的。