ES_13_模块_模块封装代码

- JS 诞生一切时代码加载方式是浏览器直接嵌入脚本语言的方式之一，其语法是使用包 package 记录的模块来定义代码的作用域,然而在 ES6 之前一个应用每个 JS 文件所定义的所有内容都会全局作用域共享，当 web 应用变得更复杂，需要使用越来越多的 JS 代码时,这种方式导致了诸多问题,例如命名冲突、安全问题、依赖管理困难、性能问题、代码可维护性差等问题。JS 应用变得更复杂，这便是模块化的几点。

NodeJs 检查对ES6 支持程度

- 第一步：检查本地是否安装 Nodejs
  - node -v
- 第二步：检查是否安装 taobao cnpm(包管理工具)
  - cnpm -v
- 第三步：安装es-checker
  - cnpm install -g es-checker
- 第四步：检查Node.js 支持 ES 6的程度
  - es-checker

何为模块?

模块(Modules)是使用不同方式加载的 JS 文件(与 JS 源代码解释加载方式相同)，这 种不同模式将每个单元为它元素的 script (导入大不同的语义)

1. 模块代码自动运行在严格模式下,并且没有任何方法脱出严格模式
2. 在模块的顶级作用域创建的变量,不会被自动添加到共享的全局作用域 它们只在模块顶级范围内存在,
   块顶级作用域的内部存在。
3. 模块顶级作用域的 this 值为 undefined
4. 模块不从任意代码中导出内容(与其他模块以导出语句来自于早期浏览器的历史遗留物 ?)。
5. 对于需要引模块内部代码执行的代码,需要从这等导出它们 ? 从外部导出其他模块导入绑定。

基本的导出

- 你可以使用 export 关键字将它选中代码部分公开成模块内,也就是可以在其他文件中使用,这是因为其 export 语句定义了一个 JS 模块的公开接口并且共享变量。

```
// 导出变量
export var color = "red";
export let name = "Nicholas";
export let age = 29;
// 导出函数
export function sum(num1, num2) {
  return num1 + num2;
}
// 导出类
export class Rectangle {
  constructor(length, width) {
    this.length = length;
    this.width = width;
  }
}
// 内部函数不导出到外
function subtract(num1, num2) {
  return num1 - num2;
}
// 定义一个函数...
function multiply(num1, num2) {
  return num1 * num2;
}
// 导出列表中的内容
export { multiply };
```

注意

首先,除了 export 关键字之外,每个声明都以之等形式结束一样。每个使用公开函数的变量等语法是因为导出公开函数声明与此声明语法是等名相。你不能使用这种语法来导出变量或函数表达式,因为导出除了 default 外键字

其次,就像一个 extends() 函数,它所实现变量之间继承法。这是因为你不仅能导出声明,还可以导出引用(例如可以导出一个)。

最后要注意,此时声明等同 extends() 函数,此函数用来为变量不可变实现为长期变量 ,解锁变量的变量,函数用来都会在模块内部引用。

基本的导入

一旦你有了某个导出的模块,就能在其他模块内使用 import 关键字来访问这模块的功能。 import 语句有两个部分—要重要导入的标识符,以要导入的标识符的模块模块。,此处是要从,这时这是多时间。

```
import { identifier1, identifier2 } from "./example.js";
```

想在node中测试 export 和 import

1. 将文件名修改成.mjs
2. 通过下面的命令来启动文件
   - Node 9版块了一个向处于 Experimental 状态的模块，让我们可以指并 babel 等一类工具的支持，直接在 Node 环境下使用 import/export。
   - node —experimental-modules /文件名.mjs

导入单个绑定

Node 支持的模块代码加载方式是使用 package 记录的模块,即使用 require 代表一个包,用 module.js 即代表一个 类型文件。

```
// 导入单个
import { sum } from "./example.js";
console.log(sum(1, 2));
sum = 1;
```

对于基本的导入这个的第二个例子,为绑定公开了一个文件夹为 example.js 的模块,现 在使用该例子这是导入并使用该模块的绑定。模拟,可以公开导入一个标识符。

导入多个绑定

如果您从 example 模块导入多个绑定,应可以像下面这样使用这种语句:

```
import { sum, multiply, magnitude } from "./
example.js";
console.log(sum(1, 2));console.log(multiply(1, 2));
console.log(magnitude(1, 2));
```

还有一种特殊情况,即从你的模型个模块当作单一对象的行导入,这模块的所有导出都是你作为对象的属性导入。例如:

```
// 导入整个
import * as example from "./example.js";
console.log(example.sum);
console.log(example.multiply);
console.log(example.magnitude);
```

在实际的第一个例子中,这与先前的例子一样,这时 import 语句将 example 模块的所有内容导入并 export 语句定义了一个名为 example 的对象。这时,这个对象的属性 sum 指向 sum 函数,multiply 指向 multiply 函数,magnitude 指向 magnitude 函数。

完全导入一个模块

```
// 导入整个
import * as example from "./example.js";
console.log(example.sum);
console.log(example.multiply);
console.log(example.magnitude);
```

在实际的第一个例子中,这与先前的例子一样,这时 import 语句将 example 模块的所有内容导入并 export 语句定义了一个名为 example 的对象。这时,这个对象的属性 sum 指向 sum 函数,multiply 指向 multiply 函数,magnitude 指向 magnitude 函数。

导入绑定的一个微妙怪异点

```
import * as example from "./example.js";
console.log(example.sum);
console.log(example.multiply);
console.log(example.magnitude);
sum = 1;
multiply = 1;
magnitude = 1;
```

export 与 import 都有一个重要的限制,那就是它们必须使用在顶级语句或模块式的文件内部。例如,以下代码将会出错语法:

```
// 这行不会工作
if (flag) {
  export flag = true;
}
```

此时的 export 语句位于一个 if 语句内部,这是不被允许的。同样,这句不能使用条件 import 语句。

```
// 这行不会工作
if (flag) {
  import flag from "./example.js";
}
```

表达的,你不能在一个语句内部使用 import 也只能表现在顶级作用域,这意味着 以下代码也会出错语法:

```
function exchange() {
  export flag from "./example.js";
}
```

模块语法的限制(Module Syntax Limitation)

ES6 的 import 语句为变量,函数与类创建了只读绑定,而不像普通变量那样简单引用了其 值(例如,可变导入的情况及在其绑定的值,但在原导出的模块内部,那就这一点,用 值,但是这在变量更像是以下模块:

```
import sum from "./example.js";
console.log(typeof sum);
sum = 1;
```

当你导入了这两个绑定后, setNum() 函数还可以改变 value 的值

```
import {name, setNum } from "./example.js";
console.log(name);
setNum(10);
console.log(name);
name = "Nicholas";
// 错误
// "Hong"
// "Nicholas"
// 错误
```

调用 setNum("Hong") 会创建导出 setNum() 的模块内部,并且变量的 name 设置为 "Hong" ,这意这个变化会反应在模块的导入的 name 绑定上,这意味为模 块内部 name 变量的值 name 会反应到所有使用这一表示符的一个事情。

补充

```
let a = 1;
let b = 2;
console.log(a);
console.log(b);
a = 3;
console.log(a);
console.log(b);
// 1 2
// 3 2
```

在此代码中,变量 a 开始时对变量 a 进行了一个引用,但只是被 a 的值赋予了一个值。如果对变量 a 的值进行 一个改变,这个改变会反应在 console.log() 。

然后范围内的最终导入与导出,外部模块导入的 value 变量与在 example.js 模块内部的 value 变量相比,即使是对于在最初的示例来说,自始终共享这点的者的变化,然 而是 的值在范围导出的模块中发生了变化,这种绑定关系也不会被破坏,模块导出与导入时 绑定机制,与写出一个文件模块内的代码是等同的。

重命名导出与导入

```
function sum(num1, num2) {
  return num1 + num2;
}
export { sum as add };
```

此时的 sum() 函数被作为 add() 导出,即使变量名称是用 export 语句重新导出 的时 exported name ,这 这意味着当你一个模块要导入此函数时,它必须调用 add 这个名称 ,不是

```
import { add } from "./example.js";
```

假如要给导入函数的变量取一个名称,同样也可以用 as 关键字:

```
import { add as sum } from "./example.js";
console.log(sum(1, 2));
console.log(sum(1, 2));
```

模块语法确定为从模块中导出或导入默认值进行了优化,而这一模式在其他语言的中非常 普遍,例如在 CommonJS 的实现这个名称的 JS 系统一种模块规范。

默认导出的一个模块,这个模块的 export 语句会使用 关键字指定这个变量。这意味着,你可以指定一个模块中只 能设置一个默认导出,在 default 关键字跟下多个导出会 报语法错误。

```
// 导出函数
export default function(num1, num2) {
  return num1 + num2;
}
```

以下是使用 default 关键字的一个简单例子:

```
export default function(num1, num2) {
  return num1 + num2;
}
```

此模块将一个函数作为默认导出进行了导出。default 关键字后跟了这是一个匿名导出。此函 数并不需要有名称,因为它现代表这个模块自身。

你也能在 export default 后面跟着一个标识符,以指定默认的导出,此项

```
function sum(num1, num2) {
  return num1 + num2;
}
export default sum;
```

导出默认值

- 以下是使用 default 关键字的一个简单例子:

```
export default function(num1, num2) {
  return num1 + num2;
}
```

- 此模块将一个函数作为默认导出进行了导出。default 关键字后跟了这是一个匿名导出。此函 数并不需要有名称,因为它现代表这个模块自身。
- 你也能在 export default 后面跟着一个标识符,以指定默认的导出,此项

```
function sum(num1, num2) {
  return num1 + num2;
}
export default sum;
```

- 这种特定的 export 会进入指定模块变量 default 的实际变量,这意 是因其导出。当然,你也可以选 择将一个值用不同名称导出:

```
export { sum as add } from "./example.js";
```

- 此处从 "./example.js" 导入的 sum 随后以 add 的名称被导出了。你也可用来自另一个模块的内容重定向导出,可以使用星号(*)模式:

```
export * from "./example.js";
```

- 有些模块也许没有进行任何导出,相反只是修改全局作用域的变量。可能这种模块的范围就是 ,函数或类是 ,但并不是由这模块以全局作用域,但这并不意味着这模块无法进行任何作用。 模块 import 语句 export 之后 ,这种方式 内置对象的共享变量之后模块内部就可以执行,并且对于 这些对象的改变也对其他模块中

- 例如,某些提供为所有数组添加一个 puzzle() 方法,你可以修下面这样变这一个模块:

```
// 给数组添加一个方法
Array.prototype.puzzle = function(num) {
  // 检查参数是数字
  if (typeof num !== "number") {
    throw new TypeError("Argument must be an
array");
  }
  // 返回数组的第一个元素减去参数
  return this[0] - num;
};
```

- 这是一个有效的模块,因为此此没有任何导出与导入。此代码可以作为模块或脚本来使用。 由于它没有导出任何东西,你可以使用模块的导入语法来执行此模块的代码,而无须导入任 何绑定:

```
import "./example.js";
console.log([1, 2, 3].puzzle(0));
// 输出: [1, 2, 3]
typeof Array.prototype.puzzle;
// "function"
```

无绑定的导入

在 Web 浏览器中使用模块

- 即使在 ES6 之前, web 浏览器相关等不方式在 web 应用中加载 JS ,这是可能的最本加载 ,因为是

- 为了在支持模块 web 浏览器必须更新这些机制。相关的可能这这 HTML 脚为本

在 script 标签中使用模块

- script- 元素默认以脚本方式(将非模块)来加载 JS 文件,只要 type 属性缺失,或是 type 属性含有与 JS 对应的有效类型(例如 'text/javascript' 或 application 只要提供给 JS 的模块 ,而是 指模块 .mjs 中指定的文件。为了支持模块,添加了 type 属性的新值 "module" 。

```
<!-- 这个 script 元素将会加载一个模块文件 -->
<script type="module" src="module.js"></script>
<!-- 内联模块 -->
<script type="module">
import sum from "./example.js";
console.log(sum(1, 2));
</script>
```

- 当 type 的值为 "module" 时,浏览器会期望这个 script 元素指向一个模块文件,或是内联模块。 script- 元素默认以脚本方式(将非模块)来加载 JS 文件,只要 type 属性缺失,或是 type 属性含有与 JS 对应的有效类型(例如 'text/javascript' 或 application 只要提供给 JS 的模块 ,而是 指模块 .mjs 中指定的文件。为了支持模块,添加了 type 属性的新值 "module" 。
- 当 type 的值为 "module" 时,浏览器期望这个 script 元素指向一个模块文件,或是内联模块。

```
<!-- 这个 script 元素将会加载一个模块文件 -->
<script type="module" src="module.js"></script>
<!-- 内联模块 -->
<script type="module">
import sum from "./example.js";
console.log(sum(1, 2));
</script>
```

- 每个模块可能要使用 import 导入了一个或多个其他模块,这就以事情变复杂了。这也就意味 着为了能加载这个模块,浏览器必须先加载 import 语句,以便知道 第一个文件 Web 浏览器以和就这是从 import 中导入的所有模块加载 ,