13_模块,模块封装代码

- JS "共享一切"的代码加载方式是该语言混乱且最易出错的方面之一。其他语言使用包(package)之类的概念来定义代码的作用域,然而在 ES6 之前,一个应用的每个 JS 文件所 定义的所有内容都由全局作用域共享。当 web 应用变得更加复杂、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、安全问题等。 ES6 的设计目标之一就是 要解决作用域问题,并让 JS 应用变得更有条理。这便是模块的切入点。
- NodeJs 检查对ES6 支持程度
  - 第一步：检查本地是否安装Node.js
    - 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 原先的脚本加载方式相对)。这 种不同模式很有必要,因为它与脚本(script)有大大不同的语义:
    - 1. 模块代码自动运行在严格模式下,并且没有任何办法跳出严格模式;
    - 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,它们只会在模
    - 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. 模块不允许在代码中使用HTML风格的注释(这是JS来自于早期浏览器的历史遗留特 性);
    - 5. 对于需要让模块外部代码访问的内容,模块必须导出它们; 6. 允许模块从其他模块导入绑定。
  - 这些差异乍一看似乎很小,但它们代表了 JS 代码加载与执行方面的显著改变,我将在整章中 对其进行论述。模块的真实力量是按需导出与导入代码的能力,而不用将所有内容放在同一 个文件内。对于导出与导入的清楚理解,是辨别模块与脚本差异的基础。
- 基本的导出
  - 你可以使用 `export` 关键字将已发布代码部分公开给其他模块。最简单方法就是将 `export` 放置在任意变量、函数或类声明之前,从模块中将它们公开出去,就像这样:
    ```
    // 导出数据
    export var color = "red";
    export let name = "Nicholas";
    export const magicNumber = 7;
    // 导出函数
    export function sum(num1, num2) {
        return num1 + num1;
    }
    // 导出类
    export class Rectangle {
        constructor(length, width) {
            this.length = length;
            this.width = width;
    } }
    // 此函数为模块私有
    function subtract(num1, num2) {
        return num1 - num2;
    }
    // 定义一个函数......
    function multiply(num1, num2) {
        return num1 * num2;
    }
    // ......稍后将其导出
    export { multiply };
    ```
    - 注意
      - 首先,除了 `export` 关键字之外,每个声明都与正常形式完全一 样。每个被导出的函数或类都有名称,这是因为导出的函数声明与类声明必须要有名称。你 不能使用这种语法来导出匿名函数或匿名类,除非使用了 `default` 关键字
      - 其次,细看一下 `multiply()` 函数,它并没有在定义时被导出。这是因为你不仅能导出声 明,还可以导出引用(即代码最后一行)。
      - 最后请注意,此例并未导出 `subtract()` 函数。此函数在模块外部不可访问,因为任意没有 被显式导出的变量、函数或类都会在模块内保持私有。
- 基本的导入
  - 一旦你有了包含导出的模块,就能在其他模块内使用 `import` 关键字来访问已被导出的功 能。 `import` 语句有两个部分,一是需要导入的标识符,二是需导入的标识符的来源模块。 此处是导入语句的基本形式:
    ```
     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提供了一个尚处于 `Experimental` 阶段的模块，让我们可以抛弃 babel 等一类工具的束缚，直接在 Node 环境下使用 `import/export`。
      - node --experimental-modules ./文件名.mjs
  - Node.js 则遵循了它的传统惯例,基于文件系统前缀来分 辨本地文件与包(package),例如, `example` 代表一个包,而 `./example.js` 则代表一个 本地文件。
- 导入单个绑定
  - 对于"基本的导入"小节的第一个例子,先假设它位于一个文件名为 `example.js` 的模块内。你 能用多种方式来导入并使用来自该模块的绑定。例如,你可以仅导入一个标识符:
    ```
    // 单个导入
    import { sum } from "./example.js";
    console.log(sum(1, 2)); // 3
    sum = 1; // 出错
    ```
- 导入多个绑定
  - 如果你想从 example 模块导入多个绑定,你可以像下面这样显式的列出它们:
  ```
  // 多个导入
  import { sum, multiply, magicNumber } from "./
  example.js";
  console.log(sum(1, magicNumber));   // 8
  console.log(multiply(1, 2));        // 2
  ```
- 完全导入一个模块
  - 还有一种特殊情况,即允许你将整个模块当作单一对象进行导入,该模块的所有导出都会作为对象的属性存在。例如:
  ```
  // 完全导入
  import * as example from "./example.js";
  console.log(example.sum(1,
          example.magicNumber));          // 8
  console.log(example.multiply(1, 2));    // 2
  ```
  - 无论你对同一个模块使用了多少次 `import` 语句,该模块都只会被执行一次。 在导出模块的代码执行之后,已被实例化的模块就会保留在内存中,并随时都能被其他 `import` 所引用。研究以下例子:
  ```
  import { sum } from "./example.js";
  import { multiply } from "./example.js";
  import { magicNumber } from "./example.js";
  ```
- 模块语法的限制(Module Syntax Limitations)
  - `export` 与 `import` 都有一个重要的限制,那就是它们必须被用在其他语句或表达式的 外部。例如,以下代码有语法错误:
  ```
  if (flag) {
  export flag; // 语法错误
  }
  ```
  - 此处的 `export` 语句位于一个 `if` 语句内部,这是不被许可的。导出语句不能是有条件 的,也不能以任何方式动态使用。原因之一是模块语法需要让 JS 能静态判断需要导出什 么,正因为此,你只能在模块的顶级作用域使用 `export` 。
  - 类似的,你不能在一个语句内部使用 `import` ,也只能将其用在顶级作用域。这意味着 以下代码也有语法错误:
  ```
  function tryImport() {
  import flag from "./example.js"; // 语法错误
  }
  ```
- 导入绑定的一个微妙怪异点
  - 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原 始绑定。尽管导入绑定的模块无法修改绑定的值,但负责导出的模块却能做到这一点。例 如,假设你想要使用以下模块:
  ```
  export var name = "Nicholas";
  export function setName(newName) {
      name = newName;
  }
  ```
  - 当你导入了这两个绑定后, `setName()` 函数还可以改变 `name` 的值:
  ```
  import { name, setName } from "./example.js";
  console.log(name);
  setName("Greg");
  console.log(name);
  name = "Nicholas";
  // "Nicholas"
  // "Greg"
  // error
  ```
  - 调用 `setName("Greg")` 会回到导出 `setName()` 的模块内部,并在那里执行,从而将 `name` 设 置为 `"Greg"` 。注意这个变化会自动反映到所导入的 `name` 绑定上,这是因为绑定的 `name` 是导出的 `name` 标识符的本地名称,二者并非同一个事物。
  - 补充
    ```
    let a = 1;
    let b = a;
    console.log(a);
    console.log(b);
    a = 2;
    console.log(a);
    console.log(b);
    // 1 // 1
    // 2 // 1
    ```
    - 在此代码中,变量 `b` 开始时对变量 `a` 进行了一个"引用",但只是将 `a` 的值拷贝了一 份。如果对变量 `a` 的值进行修改,变量 `b` 的值是不会随着变化的。 而在范例中的模块导入与导出,外部模块导入的 `name` 变量与在 `example.js` 模块内部 的 `name` 变量对比,前者是对于后者的只读引用,会始终反映出后者的变化。就算后者 的值在负责导出的模块中发生了变化,这种绑定关系也不会被破坏。模块导出与导入的 绑定机制,与写在一个文件或模块内的代码是不同的。
- 重命名导出与导入
  - 有时,你可能并不想使用从模块中导出的变量、函数或类的原始名称。幸好,你可以更改导出的名称,无论在导出过程中,还是导入过程中,都可以。 前一种情况下,假设你想用不同的名称来导出一个函数,你可以使用 `as` 关键字来指定新的 名称,以便在模块外部用此名称指代目标函数:
  ```
  function sum(num1, num2) {
      return num1 + num2;
  }
  export { sum as add };
  ```
  - 此处的 `sum()` 函数被作为 `add()` 导出,前者是本地名称(local name),后者则是导出 名称(exported name)。这意味着当另一个模块要导入此函数时,它必须改用 `add` 这个 名称:
  ```
   import { add } from "./example.js";
  ```
  - 假若模块导入函数时想使用另一个名称,同样也可以用 `as` 关键字:
  ```
  import { add as sum } from "./example.js";
  console.log(typeof add);            // "undefined"
  console.log(sum(1, 2));             // 3
  ```
- 模块的默认值
  - 模块语法确实为从模块中导出或导入默认值进行了优化,而这一模式在其他模块系统中非常 普遍,例如在 CommonJS (在浏览器之外运行 JS 的另一种模块规范)中。模块的默认值(default value)是使用 `default` 关键字所指定的单个变量、函数或类,而你在每个模块中 只能设置一个默认导出,将 `default` 关键字用于多个导出会是语法错误。
  - 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值进行了导出, `default` 关键字标明了这是一个默认导出。此函 数并不需要有名称,因为它就代表这个模块自身。
    - 你也能在 `export default` 后面放置一个标识符,以指定默认的导出,正如:
    ```
    function sum(num1, num2) {
        return num1 + num2;
    }
    export default sum;
    ```
    - 将标识符作为默认导出来指定的第三种方式,是使用重命名语法,如下:
    ```
    function sum(num1, num2) {
        return num1 + num2;
    }
    export { sum as default };
    ```
    - `default` 标识符有特别含义,既作为重命名导出,又标明了模块需要使用的默认值。由于 `default` 在 JS 中是一个关键字,它就不能被用作变量、函数或类的名称(但它可以被用作 属性名称)。因此使用 `default` 来重命名一个导出是个特例,与非默认导出的语法保持了一 致性。若你想用单个语句一次性进行多个导出,并要求包含默认导出,这种语法就非常有 用。
  - 导入默认值
    - 你可以使用如下语法来从一个模块中导入默认值:
    ```
    // 导入默认值
    import sum from "./example.js";
    console.log(sum(1, 2));     // 3
    ```
    - 这个导入语句从 `example.js` 模块导入了其默认值。注意此处并未使用花括号,与之前在非 默认的导入中看到的不同。本地名称 `sum` 被用于代表目标模块所默认导出的函数。这种语法 是最简洁的,而 ES6 的标准制定者也期待它成为在网络上进行导入的主要形式,这样你就能导入已存在的对象。
    - 对于既导出了默认值、又导出了一个或更多非默认的绑定的模块,你可以使用单个语句来导入它的所有导出绑定。例如,假设你有这么一个模块:
      - 你可以像下面这样使用 `import` 语句,来同时导入 `color` 以及作为默认值的函数:
      ```
      import sum, { color } from "./example.js";
      console.log(sum(1, 2));     // 3
      console.log(color);         // "red"
      ```
      - 逗号将默认的本地名称与非默认的名称分隔开,后者仍旧被花括号所包裹。要记住在 `import` 语句中默认名称必须位于非默认名称之前。
      - 如同导出默认值,你也能使用重命名语法进行默认值的导入:
      ```
      // 等价于上个例子
      import { default as sum, color } from "example";
      console.log(sum(1, 2));     // 3
      console.log(color);         // "red"
      ```
      - 在此代码中,默认的导出(`default`)被重命名为 `sum` ,并且附加的 `color` 导出也被一 并导入了。此例与前面的例子是等效的。
  - 绑定的再导出
    - 也许有时你会想将当前模块已导入的内容重新再导出(例如,假设要用几个小模块来创建一个库)。你能使用本章已描述过的模式来将已导入的值再导出,就像这样:
    ```
    import { sum } from "./example.js";
    export { sum }
    ```
    - 此方法能奏效,但还可以使用单个语句来完成相同任务:
    ```
     export { sum } from "./example.js";
    ```
    - 这种形式的 `export` 会进入指定模块查看 `sum` 的定义,随后将其导出。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。 若你想将来自另一个模块的所有值完全导出,可以使用星号(*)模式:
    ```
     export * from "./example.js";
    ```
  - 无绑定的导入
    - 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对象。尽管这种模块的顶级变 量、函数或类最终并不会自动被加入全局作用域,但这并不意味着该模块无法访问全局作用 域。诸如 `Array` 与 `Object` 之类的内置对象的共享定义在模块内部是可访问的,并且对于这些对象的修改会反映到其他模块中
    - 例如,若你想为所有数组添加一个 `pushAll()` 方法,你可以像下面这样定义一个模块:
    ```
    // 没有导出与导入的模块
    Array.prototype.pushAll = function(items) {
    // items 必须是一个数组
        if (!Array.isArray(items)) {
            throw new TypeError("Argument must be an
    array.");
    }
    // 使用内置的 push() 与扩展运算符
        return this.push(...items);
    };
    ```
    - 这是一个有效的模块,尽管此处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,你可以使用简化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";
    let colors = ["red", "green", "blue"];
    let items = [];
    items.pushAll(colors);
    ```
- 在 Web 浏览器中使用模块
  - 即使在 ES6 之前, web 浏览器都有多种方式在 web 应用中加载 JS 。这些可能的脚本加载 选择是:
    - 1. 使用 `<script>` 元素以及 `src` 属性来指定代码加载的位置,以便加载 JS 代码文件;
    - 2. 使用 `<script>` 元素但不使用 `src` 属性,来嵌入内联的JS 代码;
    - 3. 加载JS代码文件并作为Worker(例如WebWorker或ServiceWorker)来执行。
  - 为了完全支持模块, web 浏览器必须更新这些机制。相关细节被定义在 HTML 规范中
  - 在 script 标签中使用模块
    - `<script>` 元素默认以脚本方式(而非模块)来加载 JS 文件,只要 `type` 属性缺失,或者 `type` 属性含有与 JS 对应的内容类型(例如 `"text/javascript"`)。 `<script>` 元素能够执 行内联脚本,也能加载在 `src` 中指定的文件。为了支持模块,添加了 `"module"` 值作为 `type` 的选项。将 `type` 设置为 `"module"` ,就告诉浏览器要将内联代码或是指定文件中的 代码当作模块,而不是当作脚本。此处有个简单范例:
      ```
      <!-- load a module JavaScript file -->
      <script type="module" src="module.js"></script>
      <!-- include a module inline -->
      <script type="module">
      import { sum } from "./example.js";
      let result = sum(1, 2);
      </script>
      ```
      - 此例中第一个 `<script>` 元素使用 `src` 加载了外部模块文件,与加载脚本唯一的区别是将 `type` 指定为 `"module"` 。第二个 `<script>` 元素则包含了一个直接嵌入到网页内的模块, `result` 变量并未被暴露到全局,因为它只在使用 `<script>` 元素定义的这个模块内部存在,因此也没有被添加为 `window` 对象的属性。 正如你所见,在网页中包含模块十分简单,并且类似于包含脚本。然而,在如何加载模块方面有一些区别。
      - 你可能已经注意到 `"module"` 并不是与 `"text/javascript"` 相似的内容类型。模块 JS 文件的内容类型与脚本 JS 文件相同,因此不可能依据文件的内容类型将它们完全区别开 来。此外,当 `type` 属性无法辨认时,浏览器就会忽略 `<script>` 元素,因此不支持模 块的浏览器也就会自动忽略 `<script type="module">` 声明,从而提供良好的向下兼容 性。
  - Web 浏览器中的模块加载次序
    - 模块相对脚本的独特之处在于:它们能使用 `import` 来指定必须要加载的其他文件,以保证 正确执行。为了支持此功能, `<script type="module">` 总是表现得像是已经应用了 `defer` 属性。
    - `defer` 属性是加载脚本文件时的可选项,但在加载模块文件时总是自动应用的。当HTML解 析到拥有 `src` 属性的 `<script type="module">` 标签时,就会立即开始下载模块文件,但并 不会执行它,直到整个网页文档全部解析完为止。模块也会按照它们在 HTML 文件中出现的 顺序依次执行,这意味着第一个 `<script type="module">` 总是保证在第二个之前执行,即使 其中有些模块不是用 `src` 指定而是包含了内联脚本。例如:
      ```
      <!-- this will execute first -->
      <script type="module" src="module1.js"></script>
      <!-- this will execute second -->
      <script type="module">
      import { sum } from "./example.js";
      let result = sum(1, 2);
      </script>
      <!-- this will execute third -->
      <script type="module" src="module2.js"></script>
      ```
      - 这三个 `<script>` 元素依照它们被指定的顺序执行,因此 `module1.js` 保证在内联模块之前 执行,而内联模块又保证在 `module2.js` 之前执行。
      - 每个模块可能都用 `import` 导入了一个或多个其他模块,这就让事情变复杂了。这也就是模 块为何首先需要被解析,因为这样才能识别所有的 `import` 语句。每个 `import` 语句又会触 发一次 fetch (无论是从网络还是从缓存中获取),并且在所有用 `import` 导入的资源被加载与执行完毕之前,没有任何模块会被执行。
      - 所有模块,无论是用 `<script type="module">` 显式包含的,还是用 `import` 式包含的,都 会依照次序加载与执行。在前面的范例中,完整的加载次序是:
        - 1. 下载并解析 `module1.js` ;
        - 2. 递归下载并解析在 `module1.js` 中使用 `import` 导入的资源;
        - 3. 解析内联模块;
        - 4. 递归下载并解析在内联模块中使用 `import` 导入的资源;
        - 5. 下载并解析 `module2.js` ;
        - 6. 递归下载并解析在 `module2.js` 中使用 `import` 导入的资源。
        - 一旦加载完毕,直到页面文档被完整解析之前,都不会有任何代码被执行。在文档解析完毕后,会发生下列行为:
          - 1. 递归执行 `module1.js` 导入的资源;
          - 2. 执行 `module1.js` ;
          - 3. 递归执行内联模块导入的资源;
          - 4. 执行内联模块;
          - 5. 递归执行 `module2.js` 导入的资源;
          - 6. 执行 `module2.js` 。
          - 注意内联模块除了不必先下载代码之外,与其他两个模块的行为一致,加载 `import` 的资源 与执行模块的次序都是完全一样的。
          - `<script type="module">` 上的 `defer` 属性总是会被忽略,因为它已经应用了该属性。