模块设计

#使用分析

项目变大时需要把不同的业务分割成多个文件，这就是模块的思想。模块是比对象与函数更大的单元，使用模块组织程序便于维护与扩展。

生产环境中一般使用打包工具如 `webpack` 构建，他提供更多的功能。但学习完本章节后会再学习打包工具会变得简单。

- 模块就是一个独立的文件，里面是函数或者类库
- 虽然 JS 没有命名空间的概念，使用模块可以解决全局变量冲突
- 模块需要隐藏内部实现，只对外开发接口
- 模块可以避免滥用全局变量，造成代码不可控
- 模块可以被不同的应用使用，提高编码效率

#实现原理

在过去 JS 不支持模块时我们使用`AMD/CMD（浏览器端使用）`、`CommonJS（Node.js使用）`、`UMD(两者都支持)`等形式定义模块。

AMD 代表性的是 `require.js`，CMD 代表是淘宝的 `seaJS` 框架。

下面通过定义一个类似 `require.js` 的 `AMD` 模块管理引擎，来体验模块的工作原理。

> 向军大叔写的`hdjs` 使用的是 AMD 规范构建

```
let module = (function() {
  //模块列表集合
  const moduleLists = {};
  function define(name, modules, action) {
    modules.map((m, i) => {
      modules[i] = moduleLists[m];
    });
    //执行并保存模块
    moduleLists[name] = action.apply(null, modules);
  }

  return { define };
})();

//声明模块不依赖其它模块
module.define("hd", [], function() {
```

```
  return {
    show() {
      console.log("hd module show");
    }
  };
});

//声明模块时依赖其它模块
module.define("xj", ["hd"], function(hd) {
  hd.show();
});

```

#基础知识

#标签使用

在浏览器中使用以下语法靠之脚本做为模块使用，这样就可以在里面使用模块的代码了。
在 html 文件中导入模块，需要定义属性 `type="module"`

```
<script type="module"></script>

```

#模块路径

在浏览器中引用模块必须添加路径如`./`，但在打包工具如`webpack`中则不需要，因为他们有自己的存放方式。
测试的 `hd.js` 的模块内容如下

```
export let hd = {
  name: "后盾人"
};

```

下面没有指定路径将发生错误

```
<script type="module">
  import { hd } from "hd.js";
```

```
</script>

```

正确使用需要添加上路径

```
<script type="module">
  import { hd } from "./hd.js";
</script>

```

#延迟解析

模块总是会在所有 html 解析后才执行，下面的模块代码可以看到后加载的 `button` 按钮元素。

- 建议为用户提供加载动画提示，当模块运行时再去掉动画

```
<body>
  <script type="module">
    console.log(document.querySelector("button")); //Button
  </script>
  <script>
    console.log(document.querySelector("button")); //undefined
  </script>
  <button>后盾人</button>
</body>

```

#严格模式

模块默认运行在严格模式，以下代码没有使用声明语句将报错

```
<script type="module">
        hd = "houdunren"; // Error
</script>

```

下面的 `this` 也会是 `undefined`

```
<script>
```

```
  console.log(this); //Window
</script>
<script type="module">
  console.log(this); //undefiend
</script>

```

#作用域

模块都有独立的顶级作用域，下面的模块不能互相访问

```
<script type="module">
  let hd = "houdunren.com";
</script>

<script type="module">
  alert(hd); // Error
</script>

```

单独文件作用域也是独立的，下面的模块 `1.2.js` 不能访问模块 `1.1.js` 中的数据

```
<script type="module" src="1.1.js"></script>
<script type="module" src="1.2.js"></script>

文件内容如下
# 1.1.js
let hd = "houdunren";

# 1.2.js
console.log(hd)

```

#预解析

模块在导入时只执行一次解析，之后的导入不会再执行模块代码，而使用第一次解析结果，并共享数据。

- 可以在首次导入时完成一些初始化工作
- 如果模块内有后台请求，也只执行一次即可

引入多入`hd.js` 脚本时只执行一次

```
<script type="module" src="hd.js"></script>
<script type="module" src="hd.js"></script>

#hd.js内容如下
console.log("houdunren.com");

```

下面在导入多次 `hd.js` 时只解析一次

```
<script type="module">
  import "./hd.js";
  import "./hd.js";
</script>

# hd.js内容如下
console.log("houdunren.com");

```

#导入导出

ES6 使用基于文件的模块，即一个文件一个模块。

- 使用

export 将开发的接口导出

- 使用

import 导入模块接口

- 使用

*可以导入全部模块接口

- 导出是以引用方式导出，无论是标量还是对象，即模块内部变量发生变化将影响已经导入的变量

> 有关于模块打包知识请在 后盾人搜索 `webpack`

#导出模块

下面定义模块 `modules/houdunren.js` ，使用 `export` 导出模块接口，没有导出的变量都是模块私有的。

下面是对定义的 `hd.js` 模块，分别导出内容

```
export const site = "后盾人";
export const func = function() {
  return "is a module function";
};
export class User {
  show() {
    console.log("user.show");
  }
}

```

下面定义了`hd.js` 模块，并使用指量导出

```
const site = "后盾人";
const func = function() {
  return "is a module function";
};
class User {
  show() {
    console.log("user.show");
  }
}
export { site, func, User };

```

#具名导入

下面导入上面定义的 `hd.js` 模块，分别导入模块导出的内容

```
<script type="module">
  import { User, site, func } from "./hd.js";
  console.log(site);
  console.log(User);
</script>

```

像下面这样在 `{}` 中导入是错误的，模块默认是在顶层静态导入，这是为了分析使用的模块方便打包

```
if (true) {
  import { site, func } from "./hd.js"; // Error
}

```

#批量导入

如果要导入的内容比较多，可以使用 `*` 来批量导入。

```
<script type="module">
  import * as api from "./hd.js";
  console.log(api.site);
  console.log(api.User);
</script>

```

#导入建议

因为以下几点，我们更建议使用明确导入方式

- 使用

webpack 构建工具时，没有导入的功能会删除节省文件大小

- 可以更清晰知道都使用了其他模块的哪些功能

#别名使用

#导入别名

可以为导入的模块重新命名，下面是为了测试定义的 `hd.js` 模块内容。

- 有些导出的模块命名过长，起别名可以理简洁
- 本模块与导入模块重名时，可以通过起别名防止错误

```
const site = "后盾人";
const func = function() {
  return "is a module function";
};
class User {
  show() {
    console.log("user.show");
```

```
  }
}
export { site, func, User };

```

模块导入使用 `as` 对接口重命名，本模块中已经存在 `func` 变量，需要对导入的模块重命名防止重名错误。

```
<script type="module">
  import { User as user, func as action, site as name } from "./hd.js";
  let func = "houdunren";
  console.log(name);
  console.log(user);
  console.log(action);
</script>

```

#导出别名

模块可以对导出给外部的功能起别名，下面是`hd.js` 模块对导出给外部的模块功能起了别名

```
const site = "后盾人";
const func = function() {
  console.log("is a module function");
};
class User {
  show() {
    console.log("user.show");
  }
}
export { site, func as action, User as user };

```

这时就要使用新的别名导入了

```
<script type="module">
  import { user, action } from "./hd.js";
  action();
</script>
```

```
5
```

#默认导出

很多时候模块只是一个类，也就是说只需要导入一个内容，这地可以使用默认导入。

使用`default` 定义默认导出的接口，导入时不需要使用 `{}`

- 可以为默认导出自定义别名
- 只能有一个默认导出
- 默认导出可以没有命名

#单一导出

下面是`hd.js` 模块内容，默认只导出一个类。并且没有对类命名，这是可以的

```
export default class {
  static show() {
    console.log("User.method");
  }
}

```

从程序来讲如果将一个导出命名为 `default` 也算默认导出

```
class User {
  static show() {
    console.log("User.method");
  }
}
export { User as default };

```

导入时就不需要使用 `{}` 来导入了

```
<script type="module">
  import User from "./hd.js";
  User.show();
</script>

```

默认导出的功能可以使用任意变量接收

```
<script type="module">
  import hd from "./hd.js";
  hd.show();
</script>

```

#混合导出

模块可以存在默认导出与命名导出。

使用`export default`导出默认接口，使用`export {}`导入普通接口

```
const site = "后盾人";
const func = function() {
  console.log("is a module function");
};
export default class {
  static show() {
    console.log("user.show");
  }
}
export { site, func };

```

也可以使用以下方式导出模块

```
const site = "后盾人";
const func = function() {
  console.log("is a module function");
};
class User {
  static show() {
    console.log("user.show");
  }
}
export { site, func, User as default };

```

导入默认接口时不需要使用`{}`，普通接口还用`{}`导入

```
<script type="module">
        //可以将 hd 替换为任何变量
  import hd from "./hd.js";
  import { site } from "./hd.js";
  console.log(site);
  hd.show();
</script>

```

可以使用一条语句导入默认接口与常规接口

```
import show, { name } from "/modules/houdunren.js";

```

也可以使用别名导入默认导出

```
import { site, default as hd } from "./hd.js";
console.log(site);
hd.show();

```

如果是批量导入时，使用 `default` 获得默认导出

```
<script type="module">
  import * as api from "./hd.js";
  console.log(api.site);
  api.default.show();
</script>

```

#使用建议

对于默认导出和命名导出有以下建议

- 不建议使用默认导出，会让开发者导入时随意命名

```
import hd from "./hd.js";
```

```
import xj from "./hd.js";

```

- 如果使用默认导入最好以模块的文件名有关联，会使用代码更易阅读

```
import hd from "./hd.js";

```

#导出合并

#解决问题

可以将导入的模块重新导出使用，比如项目模块比较多如下所示，这时可以将所有模块合并到一个入口文件中。
这样只需要使用一个模块入口文件，而不用关注多个模块文件

```
|--hd.js
|--houdunren.js
...

```

#实际使用

下面是 `hd.js` 模块内容

```
const site = "后盾人";
const func = function() {
  console.log("is a module function");
};
export { site, func };

```

下面是 `houdunren.js` 模块内容

```
export default class {
  static get() {
    console.log("houdunren.js.get");
```

```
  }
}

```

下面是 `index.js` 模块内容，使用 `*` 会将默认模块以 `default` 导出

```
export * as hd from "./hd.js";
// 默认模块需要单独导出
export { default as houdunren } from "./houdunren.js";
// 以下方式导出默认模块是错误的
// export houdunren from "./houdunren.js";

```

使用方法如下

```
<script type="module">
  import * as api from "./index.js";
  console.log(api);
  api.houdunren.get();
  console.log(api.hd.site);
</script>

```

#动态加载

使用 `import` 必须在顶层静态导入模块，而使用`import()` 函数可以动态导入模块，它返回一个 `promise` 对象。

#静态导入

使用 `import` 顶层静态导入，像下面这样在 `{}` 中导入是错误的，这是为了分析使用的模块方便打包，所以系统禁止这种行为

```
if (true) {
  import { site, func } from "./hd.js"; // Error
}

```

#动态使用

测试用的 `hd.js` 模块内容如下

```
const site = "后盾人";
const func = function() {
  console.log("is a module function");
};
export { site, func };

```

使用 `import()` 函数可以动态导入，实现按需加载

```
<script>
  if (true) {
    let hd = import("./hd.js").then(module => {
      console.log(module.site);
    });
  }
</script>

```

下面是在点击事件发生后按需要加载模块

```
<button>后盾人</button>
<script>
  document.querySelector("button").addEventListener("click", () => {
    let hd = import("./hd.js").then(module => {
      console.log(module.site);
    });
  });
</script>

```

因为是返回的对象可以使用解构语法

```
<button>后盾人</button>
<script>
  document.querySelector("button").addEventListener("click", () => {
```

```
    let hd = import("./hd.js").then(({ site, func }) => {
      console.log(site);
    });
  });
</script>

```

#指令总结

表达式	说明
export function show(){}	导出函数
export const name='后盾人'	导出变量
export class User{}	导出类
export default show	默认导出
const name = '后盾人' export {name}	导出已经存在变量
export {name as hd_name}	别名导出
import defaultVar from 'houdunren.js'	导入默认导出
import {name,show} from 'a.j'	导入命名导出
Import {name as hdName,show} from 'houdunren.js'	别名导入
Import * as api from 'houdunren.js'	导入全部接口

#编译打包

编译指将 ECMAScript 2015+ 版本的代码转换为向后兼容的 JavaScript 语法，以便能够运行在当前和旧版本的浏览器或其他环境中。

首先登录 https://nodejs.org/en/ 官网下载安装`Node.js`，我们将使用其他的 npm 命令，npm 用来安装第三方类库。

在命令行输入 `node -v` 显示版本信息表示安装成功。

> 有关详细 webpack 视频教程，请登录后盾人网站检索

#安装配置

使用以下命令生成配置文件 `package.json`

```
npm init -y
```

修改`package.json`添加打包命令

```
...
"main": "index.js",
"scripts": {
 "dev": "webpack --mode development --watch"
},
...
```

安装 webpack 工具包，如果安装慢可以使用淘宝 cnpm (opens new window)命令

```
npm i webpack webpack-cli --save-dev
```

#目录结构

```
index.html
--dist #压缩打包后的文件
--src
----index.js  #入口
----style.js //模块
```

index.html 内容如下

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
```

```
    <title>Document</title>
  </head>
  <body>
    <script src="dist/main.js"></script>
  </body>
</html>

```

index.js 内容如下

```
import style from "./style";
new style().init();

```

style.js

```
export default class User {
  () {}
  init() {
    document.body.style.backgroundColor = "green";
  }
}

```

#执行打包

运行以下命令将生成打包文件到 `dist`目录，因为在命令中添加了 `--watch`参数，所以源文件编辑后自动生成打包文件。

```
npm run dev
```