# 模块打包

## CommonJS

2009年提出的包含模块、文件、IO、控制台在内的一系列标准

CommonJS 最初只为服务端设计，直到有了Browserify。

### 模块

CommonJS规定每个文件时一个模块
每个模块有自己的作用域，所有变量与函数只有自己能访问，对外不可见

### 导出

导出是模块向外暴露自生的唯一方式

CommonJS相当于默认在首部添加了如下代码

```
var module = {
    exports:{}
};
var exports = module.exports;
```

CommonJS，通过module.exports可以导出模块内容
module.exports简写exports

exports指向 module.exports的地址
所以exports直接赋值，就指向了新对象，导致exports失效

真正有用的只是module.exports
如果module.exports指向新对象，会导致exports失效

### 导入

CommonJS使用require导入
`const b = require('./b.js')`

如果require的模块是第一次加载，执行该模块，导出内容
如果require的模块曾被加载过，不会再次执行，直接导出上一次执行后的结果。

module.loaded 记录是否被加载过

## ES6 Module

### 导出

- 命名导出
  export {标识符}
  export 标识符 =
- 默认导出
  export default

### 导入

- 导入默认
  import ... from
- 导入命名
  import { ... as ... } from
- 整体导入
  import * as ... from
- 混合导入
  import React,{ Component } from 'react'

### 复合写法

模块导入后立即导出，一般用在模块入口文件
export {...} from
默认导出无法使用复合写法，只能导入导出拆开写
import ... from
export default ...

## ES6 Module 与 CommonJS 区别

### 动态与静态

模块依赖关系建立在代码运行阶段还是编译阶段
CommonJS require模块路径可以动态指定，所以不运行代码无法明确依赖关系 动态
ES6 Module 导入导出都是声明式的 静态

ES6 Module优点

- 死代码检测和排除
  静态分析依赖关系，未调用的模块永远不会执行（死代码），静态分析后打包可以去掉这些代码，减小打包资源体积
- 模块变量类型检查
  静态模块结构有助于确保模块之间传递的值或接口类型式正确的
- 编译器优化
  CommonJS 本质上导出的是 module.exports 对象，ES6 module支持导出变量，可以减少引用层级，程序效率更高

## 值拷贝与动态映射

CommonJS 导入的是导出的值拷贝，并不会影响导出本身
ES6 Module 导入是原有值的映射，read-only，无法直接修改，但是可以通过 其它导出方法去修改值。

```
// calculator.js
let count = 0;
const add = function (a,b) {
    count +=1;
    return a + b
}
export {count, add}

// index.js
import {count, add} from './calculator.js'

console.log(count); // 0
add(2,3)
console.log(count); // 1
// count += 1  报错 SyntaxError:'count' is read-only
```

## 循环依赖

A依赖B，B依赖A

设计时，应避免循环依赖

当依赖太多的时候很容易产生隐式循环依赖：A-B B-C C-D D-A

```
// foo.js
const bar = require('./bar.js');
console.log('value of bar:',bar);
module.exports = 'This is foo.js'

// bar.js
const foo = require('./foo.js');
console.log('value of foo:',foo);
module.exports = 'This is bar.js'

// index.js
require('./foo.js')
```

执行index.js
输出：

```
value of foo: {}
value of bar: This is bar.js
```

首先导入时候，会标记已经执行（先标记，再执行）
index
->foo,标记执行
->bar,依赖foo,发现foo已经执行过了，直接取module.exports,此时foo还在第一行，module.exports为{}

->输出 value of foo: {}
->bar执行完毕，回到foo，输出：value of bar: This is bar.js

```
// The require function
function __webpack_require__(moduleId) {
  if(installedModules[moduleId]) {
    return installedModules[moduleId].exports;
  }
  // Create a new module (and put it into the cache)
  var module = installedModules[moduleId] = {
    i: moduleId,
    l: false,
    exports: {}
  };
  ...
}
```

上述例子用ES6 Module 也是输出的是 ==undefined
但是ES6 Moudle用的是引用，相对来讲还是可以支持循环引用的

```
//index.js
import foo from './foo.js';
foo('index.js');

// foo.js
import bar from './bar.js';
function foo(invoker) {
    console.log(invoker + ' invokes foo.js');
    bar('foo.js');
}
export default foo;

// bar.js
import foo from './foo.js';
let invoked = false;
function bar(invoker) {
    if(!invoked) {
        invoked = true;
        console.log(invoker + ' invokes bar.js');
        foo('bar.js');
    }
}
export default bar;
```

# 加载其它类型模块

AMD、CMD、UMD等其他标准

## 非模块化文件

非模块文件指的是并不遵循任何一种模块标准的文件
老项目经常会有.script标签中引入的jQuery及其它插件

使用Webpack打包非模块化文件，直接import引入即可
`import './jquery.min.js'`

需要考虑非模块文件是否全局?
Webpack会给每个文件包装一层函数作用域避免污染全局，如果是隐式全局变量声明，则无法讲模块挂在全局，会出错

```
// 通过顶层作用域声明变量的方式暴露接口
var calculator = {
    // ...
}
```

# AMD

Asynchronous Module Definition 异步模块定义

```
define('getSum',['calculator'],function(math) {
    return function(a,b) {
        console.log('sum:'+ calculator.add(a,b))
    }
})
```

AMD 使用define函数定义模块
参数:

- 当前模块ID（模块名）
- 当前模块依赖
- 模块导出值，可以是函数或者对象

导入 使用require函数，但是是异步
参数:

- 加载的模块
- 加载完成后执行的回调

```
require(['getSum'],function(getSum){
    getSum(2,3)
})
```

优点：异步加载，非阻塞，不会阻塞浏览器
缺点：语法冗长，异步加载不如同步清晰，容易造成回调地狱

# UMD

Universal Module Definition，通用模块标准，使一个模块能够运行在各种环境下

```
// calculator.js
(function (global, main) {
    // 根据当前环境采取不同的导出方式
    if (typeof define === 'function' && define.amd) {
        // AMD
        define(...);
    } else if (typeof exports === 'object') {
        // CommonJS
        module.exports = ...;
    } else {
        // 非模块化环境
        global.add = ...;
    }
}(this, function () {
    // 定义模块主体
    return {...}
}));
```

类似浏览器兼容，通过判断来决定导出
UMD默认判断顺序 判断顺序 AMD->CommonJS
但是AMD导出无法被CommonJS与ES6 Module正确引入，可以改变UMD模块中的判断顺序

## CMD （不重要）

Common Module Definition，也是通用模块定义

- 异步加载模块
- 推崇就近依赖，只有用到某个模块时候再去require，与其他依赖前置标准不同

使用define来定义define(id?, deps?, factory)

- factory是一个函数，有三个参数，function(require, exports, module)
  - require 是一个方法，接受 模块标识 作为唯一参数，用来获取其他模块提供的接口：require(id)
  - exports 是一个对象，用来向外提供模块接口
  - module 是一个对象，上面存储了与当前模块相关联的一些属性和方法

```
 // 定义模块  module.js
define(function(require, exports, module) {
  var $ = require('jquery.min.js')
  $('div').addClass('active');
});

// 加载模块
```

```
seajs.use(['module.js'], function(my){
});
```

## 加载npm模块

npm，让开发者在其它平台上找到他人开发和发布的库，并安装到项目中
Java Maven；Ruby gem；js npm/yarn
npm/yarn仓库是共通的

npm会把库安装在 node_modules目录下，并将模块的依赖信息记录在package.json中
只需要import ... from '包名' 即可使用

Webpack打包解析到这条语句，直接去node_modules中寻找包名，不需要我们写出路径
在node_modules的每个包，都有自己的package.json,其中main字段是入口文件

也可以单独加载模块内部某个js文件，打包的时候就只会打包那个文件
`import all from 'lodash/fp/all.js'`
这样可以减小打包资源的体积

# 模块打包原理

代码经过Webpack打包后会变成如下形式（大体结构，实际有差异）

```
// 立即执行匿名函数
(function(modules) {
    //模块缓存
    var installedModules = {};
    // 实现require
    function __webpack_require__(moduleId) {
        ...
    }
    // 执行入口模块的加载
    return __webpack_require__(__webpack_require__.s = 0);
})({
    // modules: 以key-value的形式储存所有被打包的模块
    0: function(module, exports, __webpack_require__) {
        // 打包入口
        module.exports = __webpack_require__("3qiv");
    },
    "3qiv": function(module, exports, __webpack_require__) {
        // index.js内容
    },
    jkzz: function(module, exports) {
        // calculator.js 内容
    }
});
```

- 变成立即执行的匿名函数，包裹整个bundle，构成自身作用域
- installedModules对象实现模块缓存（闭包）
- 执行返回的函数 `__webpack__require__`
- key-value存储被打包的模块，也相当于每个模块有自己的命名空间即key了
  key是hash字符串，即模块的id

# bundle如何在浏览器中执行

1. 最外层匿名函数初始化浏览器执行环境，为模块的加载和执行做一些准备
   定义installedModules对象，`__webpack__require__`函数等
2. 加载入口模块
   每个bundle都有且只有一个入口模块，浏览器会从它开始执行
3. 执行模块代码
   如果执行到了 module.exports则记录模块的导出值
   遇到require函数，暂时交出执行权，进入`__webpack__require__`函数内加载其它模块的逻辑
4. `__webpack__require__`函数会判断是否存在installedModules中，如果存在，直接取值，不存在回到第三步
5. 所有依赖执行完毕，执行权回到入口模块
6. 入口模块执行到结尾，bundle运行结束

# 小结

CommonJS 值拷贝
ES6 映射，静态特性带来的优化
AMD
CMD
UMD
模块打包原理
bundle如何执行