# ES6模块化

ES6规定每个js文件都是一个独立模块

* 导入与导出

导入 用 import

import j1 from './01\_默认导出.js'

**可以多个导入 名称要合法**

向外部暴露接口用 export

export default {…}

一个js 只能一次导出

**按需导入**

import {

    a

} from './04\_按需导出.js'

console.log(a)

**按需导出**

export var a = 111

export var b = 222

按需导入可以配合默认导入一起使用

* Promise回调地域

**什么是promise:**

1.主要用于**异步计算**  
2、可以将**异步操作队列化**，按照期望的顺序执行，返回符合预期的结果  
3、可以在对象之间传递和操作promise，帮助我们处理队列

**什么是回调地狱**  多层的回调函数相互嵌套 形成回调地域

，一个异步请求套着一个异步请求，一个异步请求依赖于另一个的执行结果，使用回调的方式相互嵌套。

**缺点也是特点** 难维护 可读性差

**解决措施：** async/await

**Promise：构造函数**

**Promise.prototype** 上包含一个.then()方法

当new promise（）构造函数得到实例对象，都能通过原型链访问到.then（）

* .then()方法

用于预先指定成功和失败的回调函数

特性：

成功回调函数必选，失败可选

异步读取（顺序不定）

结果：

thenFs.readFile('./text/1.text', 'utf8').then((*r1*) => {

    console.log(*r1*)

})

thenFs.readFile('./text/2.text', 'utf8').then((*r1*) => {

    console.log(*r1*)

})

111 222

222 111

* 基于promise异步顺序读取

*// 基于promise顺序读取*

thenFs

    .readFile('./text/11.text', 'utf8')

    .then((*r1*) => {

        console.log(*r1*)

        return thenFs.readFile('./text/2.text', 'utf8')

    })

    .then((*r2*) => {

        console.log(*r2*)

*// ...可以继续嵌套*

    })

    .catch((*err*) => {//捕获所有异常

        console.log(*err*.message)

    })

结果：

111 111 …

222 222 …

* Promise.all() Promise.race

import thenFs from 'then-fs'

const promiseArr = [

    thenFs.readFile('./text/1.text', 'utf8'),

    thenFs.readFile('./text/2.text', 'utf8'),

    thenFs.readFile('./text/3.text', 'utf8')

]

Promise.all(promiseArr).then(*result* => { *//按序顺序读取所有的*

    console.log(*result*)

})

Promise.race(promiseArr).then(*result* => { *//读取最先到达的一个！*

    console.log(*result*);

})

* Promise封装读文件方法

1.方法名：getFile 形参fpath 返回值为promise对象

2．再获取.then的两个实参 在promise中添加两个参数resolve和reject回调函数

3.

getFile('./text/11.text').then((*r1*) => {

        console.log(*r1*);

    })

    .catch((*err*) => console.log(*err*.message))

* async/await

用于简化promise的异步操作

1. 基本使用

*给promise实例则可以加 await 关键词来简化promise异步操作*

1. 注意事项

1．await async 同时出现不能少

2.async中第一个await前的代码会同步执行，之后的异步执行

* 同步任务/异步任务 EventLoop 事件循环

同步任务：非耗时任务

异步任务：耗时任务

**同步任务，异步任务的执行过程：**

1. 同步任务有js主线程**次序**执行！
2. 异步任务**委托给宿主环境**（如node.js）执行
3. 宿主执行完后，对应的**回调函数加入任务队列等待**js主线程执行
4. 当js执行栈空，js主线程**读取任务队列**并会**次序**执行
5. 重复前四

* 宏任务与微任务

**宏任务：(macrotask)**

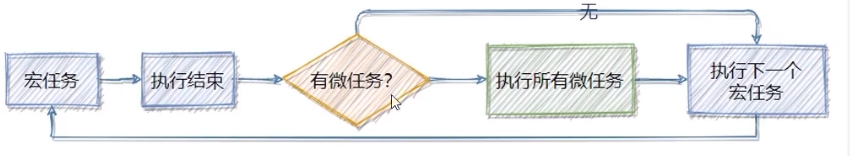
Ajax

setTimeout setInterval

文件操作

其他宏任务

任务执行顺序：



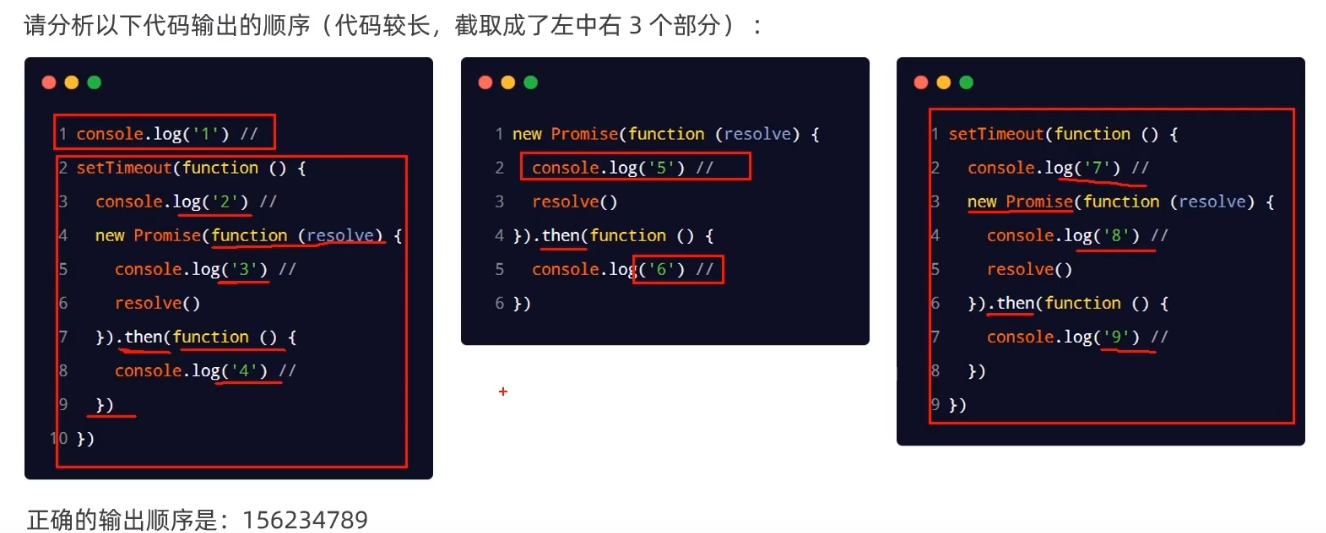
**微任务：(microtask)**

Promise.then() .catch 和 .finally

Process.nextTick

其他微任务

所有任务执行顺序：先找同步任务—>宏任务—>微任务下一个宏任务



# Webpack

**前端工程化：模块化，组件化，规范化，自动化**

优点：提高效率，降低技术选型，让前端开发自成体系

解决方案: grunt gulp(过时) webpack

* 1. webpack安装及使用:

**安装**

**npm i webpack@5.5.1 webpack-cli@4.2.0 -D**

**-D是安装在非依赖节点（开发节点）**

webpack处理**简单js**

**配置**

***在package.js中配置***

 "scripts": {

    "dev": "webpack"

  },

之后就可以用 npm run dev 打包了

***在 webpack.config.js中配置***

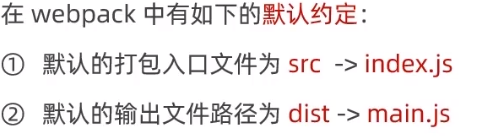
**可以通过使用**

module.exports = {

    mode: 'development'

}

来进设置打包模式 压缩模式和非压缩模式



### Webpack-dev-serve

**安装**

npm i webpack-dev-server -D 安装在开发节点

**配置**

***在package.js中配置***

  "scripts": {

    "dev": "webpack serve"

  },

实时打包放在内存中，提高了性能

放置于项目文件中，但是是虚拟文件，不可见

**更改inde.html中 src改为根目录的bundle.js**

    <script src="/bundle.js"></script>

**自动打开和网站指定**

***在webpack.config.js中配置***

  devServer: {

        open: true,

        host: '127.0.0.1',

        port: 80

    },

### html-webpack-plugin使用

**安装**

npm i html-webpack-plugin@4.5.0 -D

**配置**

***在webpack.config.js中配***

*// 导入 html-wbepack-plugin*

const HtmlPlugin = require('html-webpack-plugin')

*// 创建实例对象*

const htmlPlugin = new htmlPlugin({

    template: './src/index.html', *//目标文件位置*

    filename: './index.html', *//复制文件地址* **也放在内存中 不可见**

})

*// 挂载html-plugin对象 在module.exports 下*

*Plugins: [htmlPlugin]*

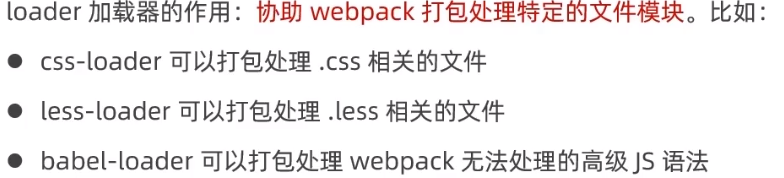
顺序不能反先创建对象再 向module.exports…挂载

* Loader

**作用**

webpack只能打包处理.js的文件

非.js用和**复杂js**loader



**一切皆模块儿**

#### 在index.js中导入css

**安装：**

**npm i style-loader@2.0.0 css-loader@5.0.1 -D**

**配置**

***在webpack.config.js中配置***

 module: {

        rules: [{

            test: /\.css$/,

            use: ['style-loader', 'css-loader']

        }]

    },

use 的顺序不能反 会报错

#### 在index.js中导入less

**安装：**

**npm i** [**less-loader@7.1.0**](mailto:less-loader@7.1.0) **less@3.12.2 -D**

**配置**

***在webpack.config.js中配置***

   module: {

        rules: [{

            test: /\.less$/,

            use: ['style-loader', 'css-loader', 'less-loader']

        }]

    },

#### 在index.js中导入url相关文件

**安装：**

**npm i url-loader@4.1.1 file-loader@6.2.0 -D**

**配置**

***在webpack.config.js中配置***

   module: {

        rules: [ {

                test: /\.jpg|png|gif|jpeg$/,

                loader: 'url-loader',

                options: { *//参数项*

                    limit: 1145980 *//limit 指定图片大小单位是字节，当<=limit才会转base64格式图片*

                }

            }]

    }

#### 在index.js中导入js高级

**安装：**

**npm i babel-loader@8.2.1 @babel/core@7.12.3 @babel/plugin-proposal-class-properties@7.12.1 -D**

**配置**

***在webpack.config.js中配置***

#### 整理dist目录

*// 指定打包出口*

    output: {

*// 出口路径*

        path: path.join(\_\_dirname, './dist'),

*// 输出文件名称*

        filename: 'js/bundle.js' //更改输出文件目录

    },

#### 添加image目录到dist

 options: { *//参数项*

                    limit: 10240, *//limit 指定图片大小单位是字节，当<=limit才会转base64格式图片*

                    outputPath: './image',

                },

#### 清除dist的旧文件

const cleanPlugin = new cleanWebpackPlugin() *//清楚dist的旧文件*

//挂载

    plugins: [htmlPlugin, cleanPlugin]

* Source Map

概念：一个**信息文件**，存储的**位置信息**，记录着代码压缩前后的对应关系

作用：便于调试

***开发环境***下的 Source Map 是默认启用的

默认的 Source Map 报错会有行数偏差

配置

***在webpack.config.js中配置***

*//仅开发模式使用解决报错时代码行数偏差问题*

    devtool: 'eval-source-map',

***生产环境***下的 Source Map 省略了 devtool选项 可以防止源代码暴露

配置

***在webpack.config.js中配置***

*//仅开发模式使用解决报错时代码行数偏差问题*

    devtool: 'nosources-source-map',

不向外暴露源码又能知道报错位置