ES6模块化与异步编程

1.ES6模块化的规范

是浏览器端与服务器端通用的模块化开发规范

```
ES6 模块化规范中定义：
1、每个 js 文件都是一个独立的模块
2、导入其它模块成员使用 import 关键字
3、向外共享模块成员使用 export 关键字
```

2.ES6在node.js中的使用

新建项目后，如何生成package.json文件：npm init -y

在 package.json文件中添加："type": "module";

作用：配置 node.js 环境，声明可以使用 ES6 模块化

3.ES6模块化的基本语法

主要包括以下3种用法：

```
① 默认导出与默认导入
② 按需导出与按需导入
③ 直接导入并执行模块中的代码
```

3.1默认导出

语法：**export default 默认导出的成员**

注意：每个模块中只能使用一次 export default

3.2默认导入

语法：**import 接受名称 from '模块标识符'**

3.3按需导出

语法：**export 按需导出的成员**

3.4按需导入

语法：**import { s1 } from '模块标识符'**

注意：若需要按需导入多个成员，以英文 ","隔开

3.5按需导出和按需导入的注意事项

```
① 每个模块中可以使用多次按需导出
② 按需导入的成员名称必须和按需导出的名称保持一致
③ 按需导入时，可以使用 as 关键字进行重命名
④ 按需导入可以和默认导入一起使用
```

3.6直接导入并执行模块中的代码

如果只想单纯地执行某个模块中的代码，并不需要得到模块中向外共享的成员。此时，可以直接导入并执行模 块代码

4.Promise

4.1回调地狱

多层回调函数的相互嵌套，就形成了回调地狱

```
回调地狱的缺点：
 1、代码耦合性太强，牵一发而动全身，难以维护
 2、大量冗余的代码相互嵌套，代码的可读性变差
```

为了解决回调地狱的问题，ES6新增了Promise

4.2Promise的概念

1、Promise是一个构造函数，new出来的Promise实例对象，代表一个异步操作

```
const p = new Promise()
```

2、Promise.prototype上包含一个.then()方法

```
每一次 new Promise() 构造函数得到的实例对象，都可以通过原型链的方式访问到 .then() 方法，例如 p.then()
```

3、.then() 方法用来预先指定成功和失败的回调函数

```
p.then(成功的回调函数，失败的回调函数)
p.then(result => { }, error => { })
调用 .then() 方法时，成功的回调函数是必选的、失败的回调函数是可选的
```

4.3基于then-fs读取文件

安装 then-fs：npm i then-fs

```
调用 then-fs 提供的 readFile() 方法，可以异步地读取文件的内容，它的返回值是 Promise 的实例对象。因此可以调用 .then() 方法为每个 Promise 异步操作指定成功和失败之后的回调函数。
readFile()方法可以实现异步读取，但不能保证文件读取的顺序
```

.then()方法的特性：上一个 .then() 方法中返回了一个新的 Promise 实例对象，则可以通过下一个 .then() 继续进行处理。通 过 .then() 方法的链式调用，就解决了回调地狱的问题。

在 Promise 的链式操作中如果发生了错误，可以通过 .catch 方法进行捕获和处理

可以把.catch方法提前放到前面就不会导致错误发生后面的代码无法执行

4.4Promise.all()方法

```
Promise.all() 方法会发起并行的 Promise 异步操作，等所有的异步操作全部结束后才会执行下一步的 .then
操作（等待机制）。
```

4.5Promise.race()方法

```
Promise.race() 方法会发起并行的 Promise 异步操作，只要任何一个异步操作完成，就立即执行下一步的.then 操作（赛跑机制）
```

4.6基于Promise 封装读文件的方法

```
方法的封装要求：
① 方法的名称要定义为 getFile
② 方法接收一个形参 fpath，表示要读取的文件的路径
③ 方法的返回值为 Promise 实例对象
```

4.7async和await

async/await 是 ES8（ECMAScript 2017）引入的新语法，用来**简化 Promise 异步操作。**在 async/await 出 现之前，开发者只能通过链式 .then() 的方式处理 Promise 异步操作 。

```
注意事项：
    1.如果在 function 中使用了 await，则 function 必须被 async 修饰
    2.在 async 方法中，第一个 await 之前的代码会同步执行，await 之后的代码会异步执行
```

5.EventLoop

JavaScript 是一门单线程执行的编程语言

5.1同步任务和异步任务

为了防止某个耗时任务导致程序假死的问题，JavaScript 把待执行的任务分为了两类 :

```
① 同步任务（synchronous）
    又叫做非耗时任务，指的是在主线程上排队执行的那些任务
    只有前一个同步任务执行完毕，才能执行后一个任务
② 异步任务（asynchronous）
    又叫做耗时任务，异步任务由 JavaScript 委托给宿主环境进行执行
    当异步任务执行完成后，会通知 JavaScript 主线程执行异步任务的回调函数
```

① 同步任务由 JavaScript 主线程次序执行

② 异步任务委托给宿主环境执行

③ 已完成的异步任务对应的回调函数，会被加入到任务队列中等待执行

④ JavaScript 主线程的执行栈被清空后，会读取任务队列中的回调函数，次序执行

⑤ JavaScript 主线程不断重复上面的第 4 步

5.2EventLoop的概念

```
JavaScript 主线程从“任务队列”中读取异步任务的回调函数，放到执行栈中依次执行。这个过程是循环不断的，所以整个的这种运行机制又称为 EventLoop（事件循环）。
```

经典面试题1

```
import thenFs from 'then-fs'

console.log('A')
thenFs.readFile('./files/1.txt','utf8').then(dataStr=>{
    console.log('B')
})
setTimeout(()=>{
    console.log('C')
},0)
console.log('D')
```

正确的输出结果：ADCB

```
A和D属于同步任务，会根据代码的先后顺序依次执行
C和B属于异步代码，它们的回调函数会被加入到任务队列中，等待主线程空闲时再执行
```

6.宏任务和微任务

JavaScript 把异步任务（耗时任务）又做了进一步的划分，异步任务又分为两类，分别是：

① 宏任务（macrotask）

异步 Ajax 请求

setTimeout、setInterval

文件操作

其它宏任务

② 微任务（microtask）

Promise.then、.catch 和 .finally

process.nextTick

其它微任务

1.宏任务和微任务的执行顺序

```
宏任务---> 执行结束---> 有微任务？---> 执行所有微任务---> 执行下一个宏任务
                              |                                  |
                              无----------------------------------
```

每一个宏任务执行完之后，都会检查是否存在待执行的微任务， 如果有，则执行完所有微任务之后，再继续执行下一个宏任务 。

经典面试题2

```
setTimeout(function(){
    console.log('1')
})

new Promise(function(resolve){
    console.log('2')
    resolve()
}).then(function(){
    console.log('3')
})

console.log('4')
```

输出结果为：2431

```
分析：1.先执行所有的同步任务：按顺序：2，4
     2.再执行微任务：.then()
     3.最后执行下一个宏任务：setTimeout()
```

经典面试题3

```
console.log('1')
setTimeout(function(){
    console.log('2')
    new Promise(function(resolve){
        console.log('3')
        resolve()
    }).then(function(){
        console.log('4')
    })
})
new Promise(function(resolve){
    console.log('5')
    resolve()
}).then(function(){
    console.log('6')
})
setTimeout(function(){
    console.log('7')
    new Promise(function(resolve){
        console.log('8')
        resolve()
    }).then(function(){
        console.log('9')
    })
})
```

正确的输出顺序是：156234789