目录

[第一节 ES6模块化 1](#_Toc3507)

[1.node.js中如何实现模块化 1](#_Toc26989)

[2.前端模块化规范的分类 1](#_Toc4424)

[3.什么是ES6模块化规范 1](#_Toc14926)

[4.在node.js中体验ES6模块化 2](#_Toc10347)

[5.ES6模块化的基本语法 2](#_Toc440)

[5.1默认导出 2](#_Toc11600)

[5.2默认导入 2](#_Toc23743)

[5.3默认导出导入的注意事项 3](#_Toc28099)

[5.4 按需导出 3](#_Toc31473)

[5.5 按需导入 4](#_Toc30759)

[5.6 按需导出与按需导入的注意事项 4](#_Toc8692)

[5.7 直接导入并执行模块中的代码 5](#_Toc20783)

[第二节 Promise 6](#_Toc19145)

[1.回调地狱 6](#_Toc24714)

[2.如何解决回调地狱的问题 6](#_Toc3661)

[3.Promise的基本概念 6](#_Toc5090)

[4.基于回调函数按顺序读取文件内容 7](#_Toc698)

[5.基于then-fs读取文件内容 7](#_Toc29402)

[5.1 then-fs 的基本使用 8](#_Toc31282)

[5.2 then( )方法的特性 8](#_Toc29679)

[5.3 基于Promise按顺序读取文件的内容 8](#_Toc21146)

[5.4 通过.catch捕获错误 9](#_Toc8549)

[5.5 Promise.all( )方法 10](#_Toc18765)

[5.6 Promise.race( )方法 10](#_Toc22190)

[6.基于Promise封装读文件的方法 11](#_Toc6513)

[6.1 getFile方法的基本定义 11](#_Toc25063)

[6.2 创建具体的异步操作 11](#_Toc8694)

[6.3 获取.then的两个实参 12](#_Toc29281)

[6.4 调用resolve和reject回调函数 12](#_Toc30856)

[第三节 async/await 13](#_Toc17489)

[1.什么是async/await 13](#_Toc14575)

[2.async/await的基本使用 13](#_Toc15127)

[3.async/await的使用注意事项 14](#_Toc7362)

[第四节 EventLoop 14](#_Toc26312)

[1.JavaScript是单线程的语言 14](#_Toc1362)

[2.同步任务和异步任务 15](#_Toc5841)

[3.同步任务和异步任务的执行过程 15](#_Toc12847)

[4.EventLoop的基本概念 16](#_Toc27377)

[5.结合EventLoop分析输出的顺序 16](#_Toc28871)

[6.宏任务和微任务 17](#_Toc7396)

[6.1 什么是宏任务和微任务 17](#_Toc13791)

[6.2 宏任务和微任务的执行顺序 18](#_Toc20588)

[6.3 举例分析宏任务和微任务的执行顺序 18](#_Toc22522)

[6.4 分析以下代码输出的顺序 18](#_Toc25758)

[6.5 经典面试题 19](#_Toc6644)

[第五节 总结 20](#_Toc22686)

**第一节 ES6模块化**

**1.node.js中如何实现模块化**

node.js遵循了CommonJS的模块化规范。其中:

·导入其它模块使用require( )方法

·模块对外共享成员使用module.exports对象

模块化的好处:

大家都遵守同样的模块化规范写代码，降低了沟通的成本，极大方便了各个模块之间的相互调用，利人利己。

**2.前端模块化规范的分类**

在ES6模块化规范诞生之前，JavaScript社区已经尝试并提出了AMD、CMD、CommonJS等模块化规范。

但是，这些由社区提出的模块化标准，还是存在一定的差异性与局限性、并不是浏览器与服务器通用的模块化标准，例如:

·AMD和CMD适用于浏览器端的Javascript模块化

·CommonJS适用于服务器端的Javascript模块化

太多的模块化规范给开发者增加了学习的难度与开发的成本。因此，大一统的ES6模块化规范诞生了!

**3.什么是ES6模块化规范**

ES6模块化规范是浏览器端与服务器端通用的模块化开发规范。它的出现极大的降低了前端开发者的模块化学习成本，开发者不需再额外学习AMD、CMD 或 CommonJS等模块化规范。

ES6模块化规范中定义:

·每个js文件都是一个独立的模块

·导入其它模块成员使用import 关键字

·向外共享模块成员使用export 关键字

**4.在node.js中体验ES6模块化**

node.js 中默认仅支持CommonJS模块化规范，若想基于node.js体验与学习ES6的模块化语法，可以按照如下两个步骤进行配置:

①确保安装了v14.15.1或更高版本的node.js

②在package.json 的根节点中添加"type" : "module"节点

**5.ES6模块化的基本语法**

ES6 的模块化主要包含如下3种用法:

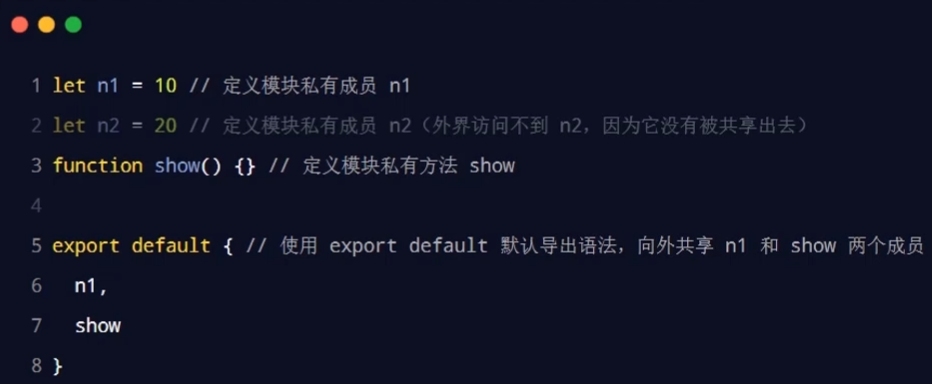
①默认导出与默认导入

②按需导出与按需导入

③直接导入并执行模块中的代码

**5.1默认导出**

默认导出的语法: export default默认导出的成员



**5.2默认导入**

默认导入的语法: import接收名称from '模块标识符'



**5.3默认导出导入的注意事项**

每个模块中，只允许使用唯一的一次export default，否则会报错!



默认导入时的接收名称可以任意名称，只要是合法的成员名称即可:



**5.4 按需导出**

按需导出的语法: export 按需导出的成员



**5.5 按需导入**

按需导入的语法: import { s1（模块成员） } from '模块标识符'



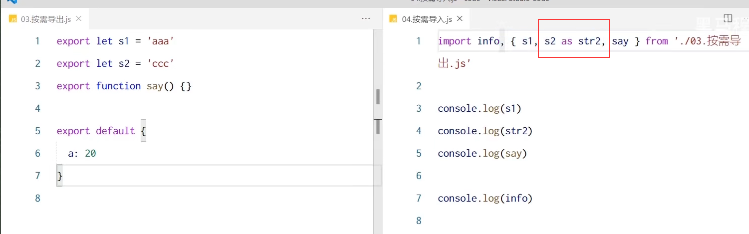
**5.6 按需导出与按需导入的注意事项**

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



**5.7 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，示例代码如下:



**第二节 Promise**

**1.回调地狱**

多层回调函数的相互嵌套，就形成了回调地狱。示例代码如下︰



回调地狱的缺点:

·代码耦合性太强，牵一发而动全身，难以维护

·大量冗余的代码相互嵌套，代码的可读性变差

**2.如何解决回调地狱的问题**

为了解决回调地狱的问题，ES6 (ECMAScript 2015）中新增了Promise的概念。

**3.Promise的基本概念**

①Promise是一个构造函数

·我们可以创建Promise的实例const p = new Promise( )

·new出来的Promise 实例对象，代表一个异步操作

②Promise.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 =>{ })

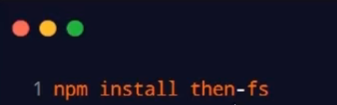
调用.then( )方法时，成功的回调函数是必选的、失败的回调函数是可选的

**4.基于回调函数按顺序读取文件内容**



**5.基于then-fs读取文件内容**

由于node.js 官方提供的fs模块仅支持以回调函数的方式读取文件，不支持Promise的调用方式。因此，需要先运行如下的命令，安装 then-fs这个第三方包，从而支持我们基于Promise的方式读取文件的内容:



**5.1 then-fs 的基本使用**

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。示例代码如下:



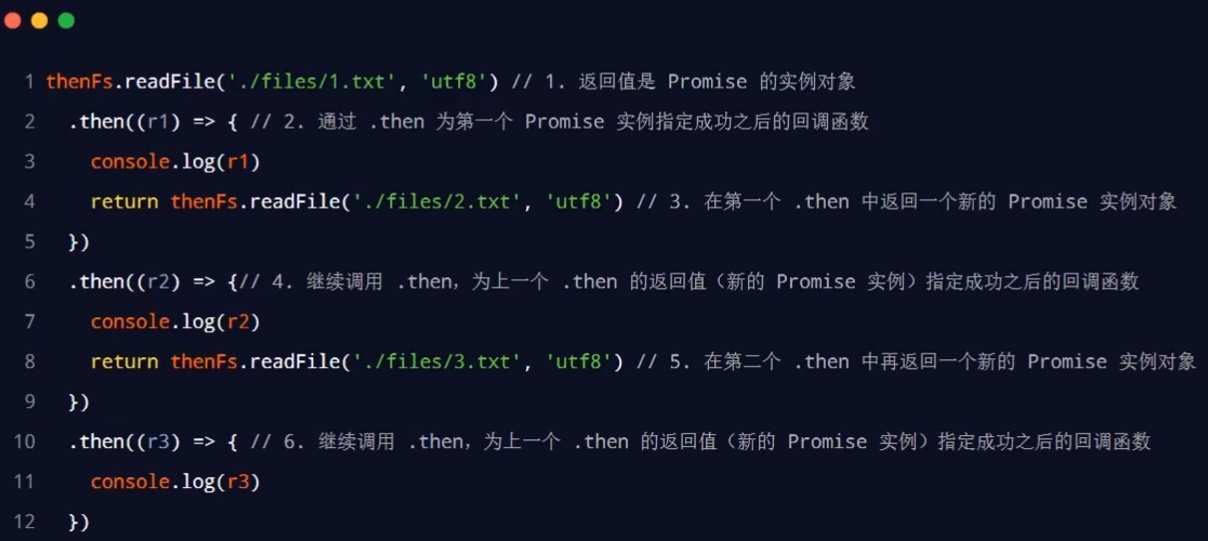
注意:上述的代码无法保证文件的读取顺序，需要做进一步的改进!

**5.2 then( )方法的特性**

如果上一个.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。

**5.3 基于Promise按顺序读取文件的内容**

Promise支持链式调用，从而来解决回调地狱的问题。示例代码如下:



**5.4 通过.catch捕获错误**

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



如果不希望前面的错误导致后续的.then无法正常执行，则可以将.catch的调用提前，示例代码如下:



**5.5 Promise.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)。示例代码如下:



注意:数组中Promise实例的顺序，就是最终结果的顺序!

**5.6 Promise.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)。示例代码如下:



**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

**6.1 getFile方法的基本定义**



注意:第5行代码中的new Promise( )只是创建了一个形式上的异步操作。

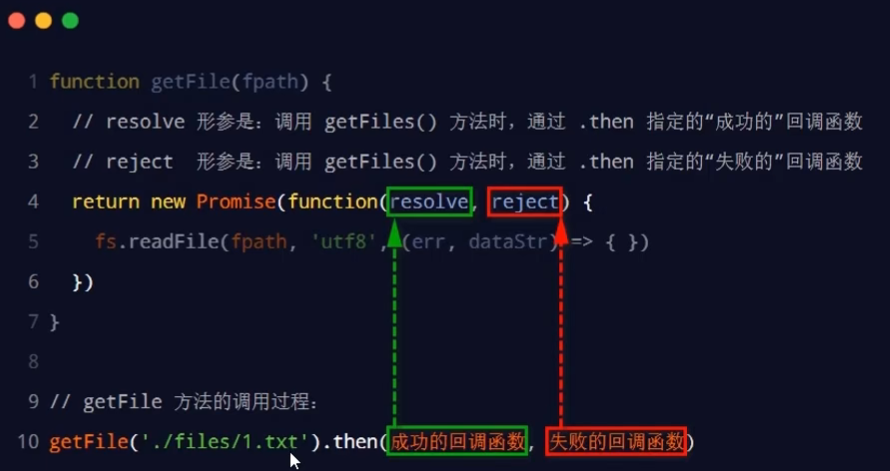
**6.2 创建具体的异步操作**

如果想要创建具体的异步操作，则需要在new Promise( )构造函数期间，传递一个function函数，将具体的异步操作定义到function函数内部。示例代码如下:



**6.3 获取.then的两个实参**

通过.then( )指定的成功和失败的回调函数，可以在function的形参中进行接收，示例代码如下:



**6.4 调用resolve和reject回调函数**

Promise 异步操作的结果，可以调用resolve或reject回调函数进行处理。示例代码如下:



**第三节 async/await**

**1.什么是async/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。示例代码如下:

.then链式调用的优点:解决了回调地狱的问题

.then链式调用的缺点:代码冗余、阅读性差、不易理解

**2.async/await的基本使用**

使用async/await简化Promise异步操作的示例代码如下:



**3.async/await的使用注意事项**

①如果在function中使用了await，则 function 必须被async修饰

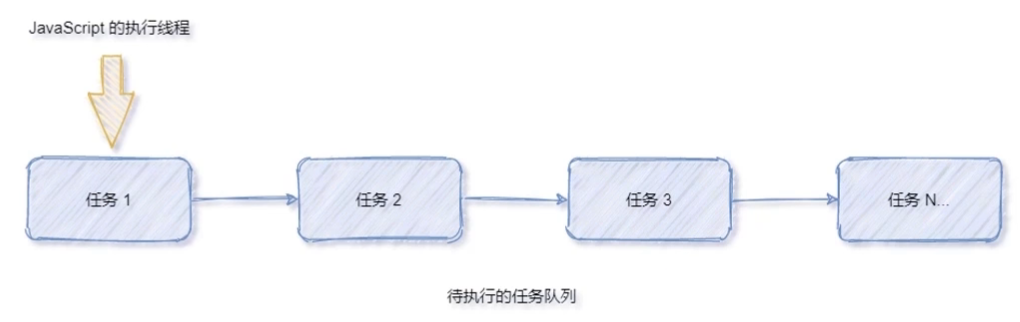
②在async方法中，第一个await之前的代码会同步执行，await 之后的代码会异步执行



**第四节 EventLoop**

**1.JavaScript是单线程的语言**

JavaScript是一门单线程执行的编程语言。也就是说，同一时间只能做一件事情。



单线程执行任务队列的问题:

如果前一个任务非常耗时，则后续的任务就不得不一直等待，从而导致程序假死的问题。

**2.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

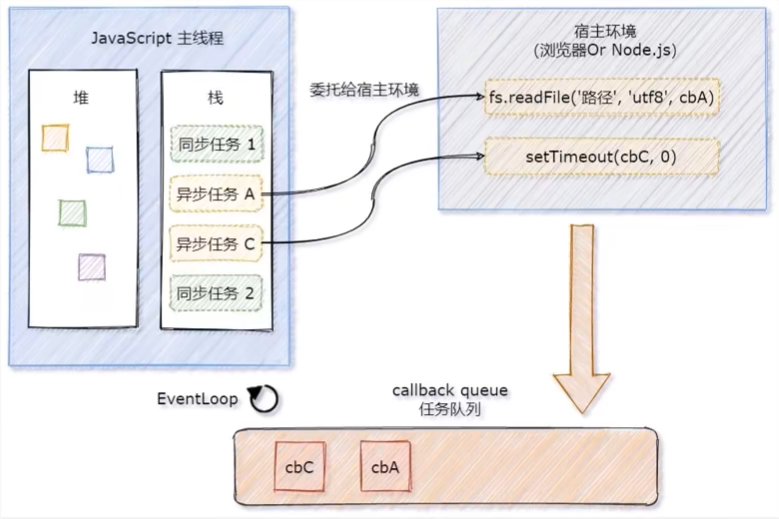
·只有前一个任务执行完毕，才能执行后一个任务

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

**3.同步任务和异步任务的执行过程**



①同步任务由JavaScript主线程依次序执行

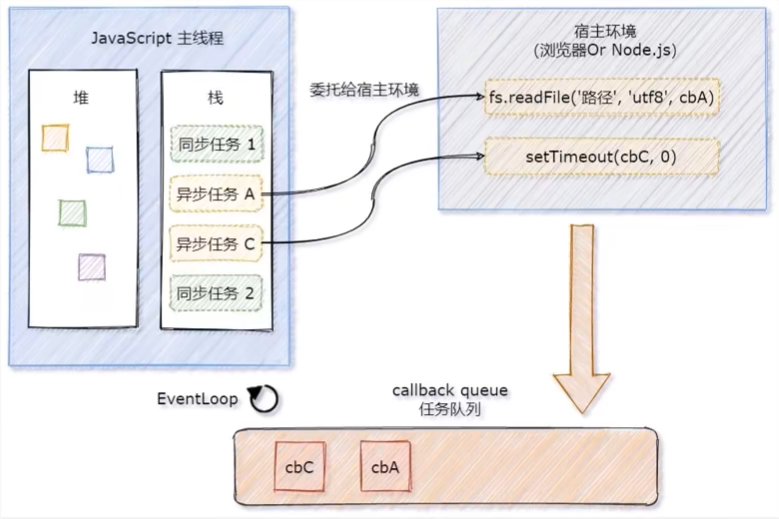
②异步任务委托给宿主环境执行

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

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

⑤JavaScript主线程不断重复上面的第4步

**4.EventLoop的基本概念**



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)。

**5.结合EventLoop分析输出的顺序**



正确的输出结果:ADCB。其中:

·A和D属于同步任务。会根据代码的先后顺序依次被执行。

·C和B属于异步任务。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**

**6.1 什么是宏任务和微任务**

JavaScript把异步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

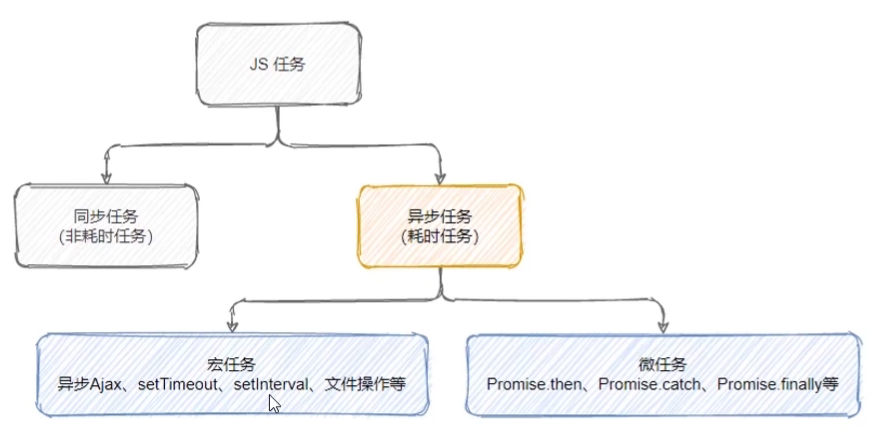
·其它宏任务

②微任务（ microtask )

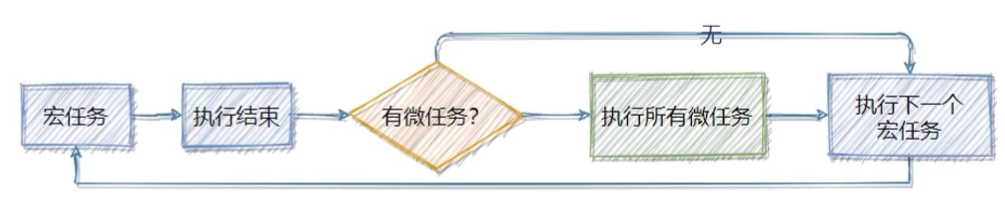
·Promise.then、.catch和.finally

·process.nextTick

·其它微任务



**6.2 宏任务和微任务的执行顺序**



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。

**6.3 举例分析宏任务和微任务的执行顺序**

①小云和小腾去银行办业务。首先，需要取号之后进行排队

·宏任务队列

②假设当前银行网点只有一个柜员，小云在办理存款业务时，小腾只能等待

·单线程，宏任务按次序执行

③小云办完存款业务后，柜员询问他是否还想办理其它业务?

·当前宏任务执行完，检查是否有微任务

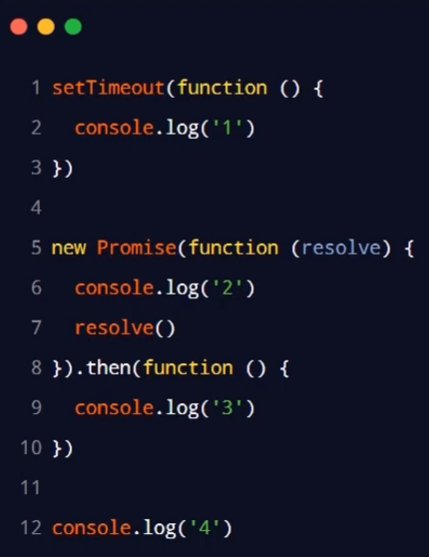
④小云告诉柜员:想要买理财产品、再办个信用卡、最后再兑换点马年纪念币?

·执行微任务，后续宏任务被推迟

⑤小云离开柜台后，柜员开始为小腾办理业务

·所有微任务执行完毕，开始执行下一个宏任务

**6.4 分析以下代码输出的顺序**



正确的输出顺序是：2431

分析:

①先执行所有的同步任务

·执行第6行、第12行代码

②再执行微任务

·执行第9行代码

③再执行下一个宏任务

·执行第2行代码

**6.5 经典面试题**

请分析以下代码输出的顺序（代码较长，截取成了左中右3个部分)︰



正确的输出顺序是：156234789

**第五节 总结**

①能够知道如何使用ES6的模块化语法

·默认导出与默认导入、按需导出与按需导入

②能够知道如何使用Promise解决回调地狱问题

·promise.then( )、promise.catch( )

③能够使用async/ await简化Promise的调用

· 方法中用到了await，则方法需要被async修饰

④能够说出什么是EventLoop

·EventLoop示意图

⑤能够说出宏任务和微任务的执行顺序

·在执行下一个宏任务之前，先检查是否有待执行的微任务