1-ECMAScript6 基本概念

`ECMAScript 6.0`【以下简称`ES6`】是`JavaScript`语言的下一代标准，已经在 2015 年 6 月正式发布了。

它的目标，是使得`JavaScript`语言可以用来编写复杂的大型应用程序，成为企业级开发语言

> **ECMAScript6 和 JavaScript关系**

- 1996 年 11 月，`JavaScript`的创造者`Netscape`公司，决定将`JavaScript`提交给标准化组织`ECMA`，希望这种语言能够成为国际标准。
- `ECMA`发布`262`号标准文件【`ECMA-262`】的第一版，规定了浏览器脚本语言的标准，并将这种语言称为`ECMAScript`，这个版本就是`1.0`版。
- 因此，`ECMAScript`【宪法】和`JavaScript`【律师】的关系是，前者是后者的规格，后者是前者的一种实现。

> **ES6 与 ECMAScript 2015 的关系**

- 2011年，`ECMAScript` 5.1 版发布后，就开始制定 6.0 版了。因此`ES6`这个词的原意，就是指`JavaScript`语言的下一个版本。
- `ES6`既是一个历史名词，也是一个泛指，含义是`5.1`版以后的`JavaScript`的下一代标准，涵盖了`ES2015`、`ES2016`、`ES2017`等。

2- 搭建环境

2.1 Node 环境

2.1.1 什么是Node.js

简单的说`Node.js`就是运行在服务端的`JavaScript`。

`JavaScript`程序，必须要依赖浏览器才能运行！没有浏览器怎么办？`nodejs`帮你解决

`Node.js`是脱离浏览器环境运行的`JavaScript`程序，基于`Google`的`V8`引擎，`V8`引擎执行`Javascript`的速度非常快，性能非常好。

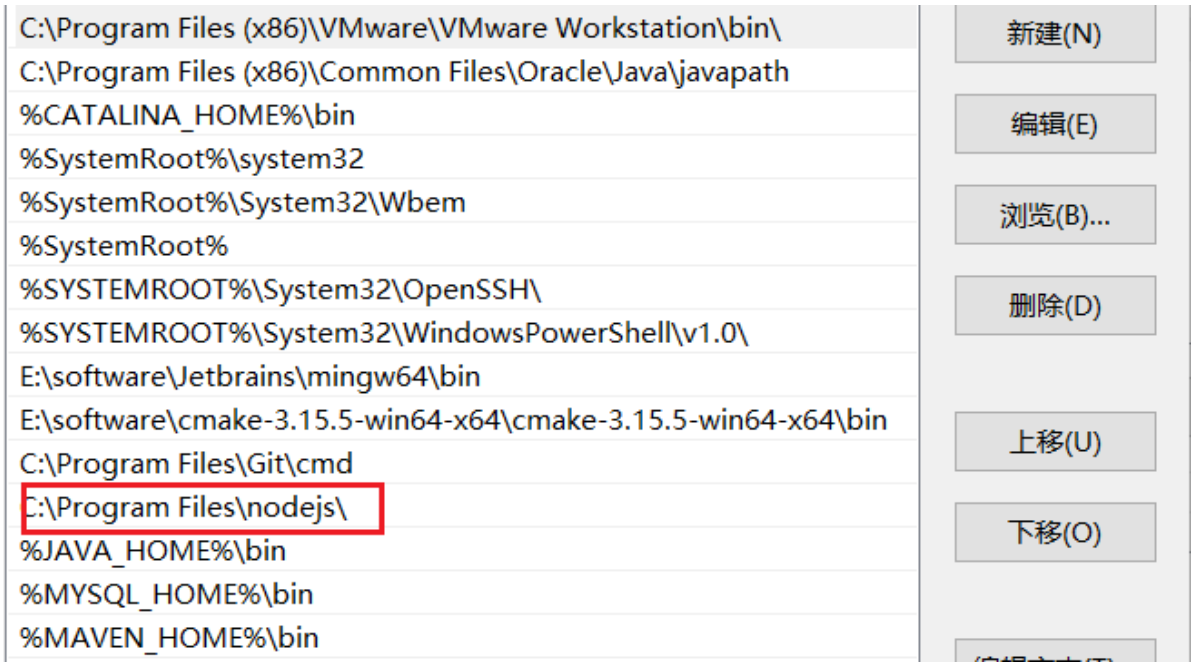
2.1.2 Node.js具体作用

`Node.js`是运行在服务端的`JavaScript`，如果熟悉`Javascript`，部署一些高性能的服务，那么`Node.js`是一个非常好的选择。

2.1.3 安装与下载

> **下载**

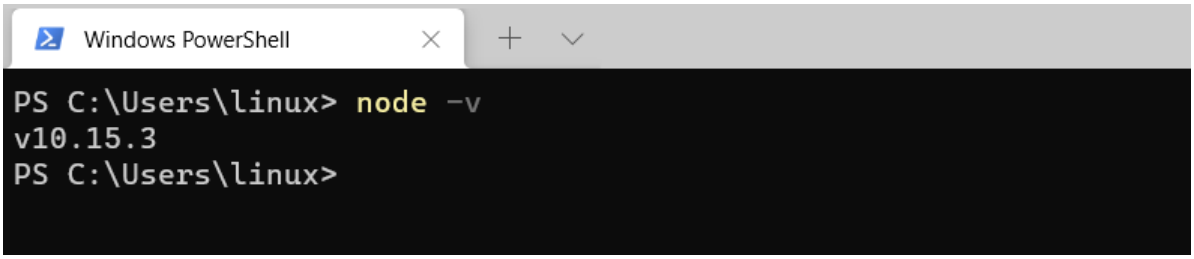
- 官网:https://nodejs.org/en/
- 中文网:http://nodejs.cn/
- LTS：长期支持版本，Current：最新版
- 安装：Windows下双击点击安装——>Next——>finish。
- 配置环境变量



> **查看版本**

在Terminal窗口中执行命令查看版本号

```
node -v
```



> **创建文件夹ES6**

用`vscode`打开目录，其目录下创建`hello.js`

```
console.log("hello,nodejs");
```

打开命令行终端：`Ctrl + Shift + y`

输入命令

```
node hello.js
```

> **执行结果**

2.2 NPM环境

2.2.1 什么是NPM

NPM全称Node Package Manager，是Node.js包管理工具，是全球最大的模块生态系统，里面所有的模块都是开源免费的，也是Node.js的包管理工具，相当于前端的Maven仓库。如果一个项目需要引用很多第三方的js文件，比如地图，报表等，文件杂而乱，自己去网上下载，到处是广告和病毒。把这些js文件统一放在一个仓库里，大家谁需要，谁就去仓库中拿过来，方便多了。npm就是这个仓库系统，如果你需要某个js文件，那就去远程仓库中下载，放在本地磁盘中，进而引用到项目中。

2.2.2 NPM安装位置

node的环境在安装的过程中，npm工具就已经安装好了。Node.js默认安装的npm包和工具的位置：Node.js目录\node_modules。

在这个目录下你可以看见npm目录，npm本身就是被NPM包管理器管理的一个工具，说明Node.js已经集成了npm工具。

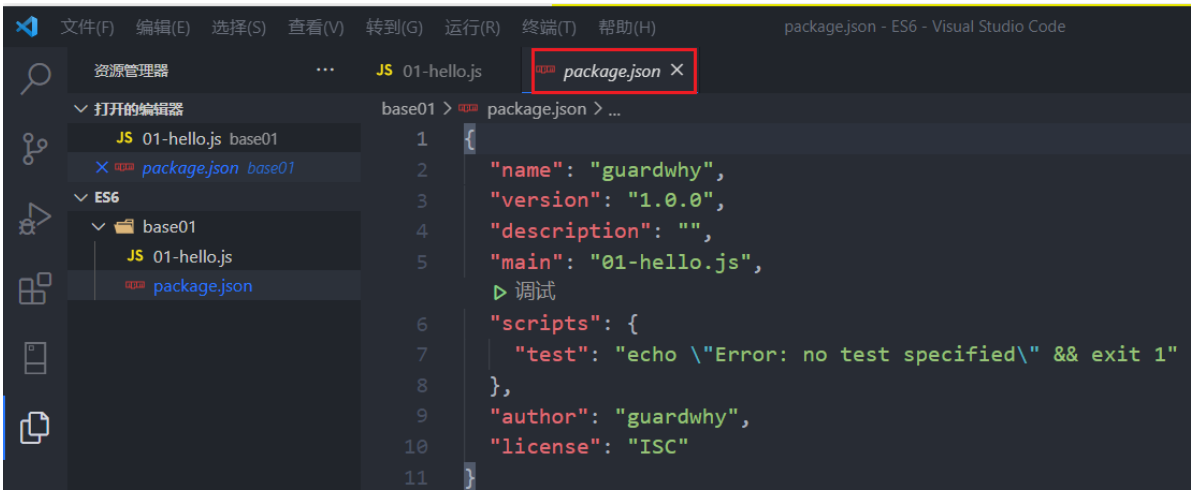
```
#在命令提示符输入 npm -v 可查看当前npm版本
npm -v
```

2.2.3 项目初始化

全新创建一个目录，作为项目目录，使用Terminal命令进入此目录，输入命令

```
npm init

# 接下来是一堆项目信息等待着你输入，如果使用默认值或你不知道怎么填写，则直接回车即可。

# package name:  你的项目名字叫啥
# version: 版本号
# description: 对项目的描述
# entry point: 项目的入口文件（一般你要用那个js文件作为node服务，就填写那个文件）
# test command: 项目启动的时候要用什么命令来执行脚本文件（默认为node app.js）
# git repository: 如果你要将项目上传到git中的话，那么就需要填写git的仓库地址（这里就不写地址了）
# keywirds:  项目关键字（我也不知道有啥用，所以我就不写了）
# author: 作者的名字（也就是你叫啥名字）
# license: 发行项目需要的证书（这里也就自己玩玩，就不写了）
```

最后会生成package.json文件，这个是包的配置文件，相当于maven的pom.xml



上述初始化一个项目也太麻烦了，要那么多输入和回车。想简单点，一切都按照默认值初始化即可。

```
npm init -y
```

2.2.4 修改npm镜像

NPM官方的管理的包都是从 http://npmjs.com下载的，但是这个网站在国内速度很慢。这里推荐使用淘宝 NPM 镜像 http://npm.taobao.org/ ，淘宝 `NPM` 镜像是一个完整`npmjs.com` 镜像，同步频率目前为`10`分钟一次，以保证尽量与官方服务同步。

设置镜像和存储地址：

```
# 由于npm代码仓库的服务器在国外，由于Great Firewall的缘故,这时可以使用淘宝的npm代码仓库，通过npm安装cnpm
npm install -g cnpm --registry=https://registry.npm.taobao.org
# 安装成功后，可以通过以下命令查看cnpm版本：
cnpm -v
#设置npm下载包时保存在本地的地址(建议英文目录),通过cnpm来操作下载速度会得到很大提升，但包的版本不一定是最新的。
cnpm config set prefix "E:\\Develop\\repo_npm"
#查看cnpm配置信息
cnpm config list
```

执行结果

```
linux@GuardWhy MINGW64 /e/workspace/Web/ES6/base01
$ npm install -g cnpm --registry=https://registry.npm.taobao.org
npm WARN deprecated har-validator@5.1.5: this
E:\Develop\repo_npm\cnpm -> E:\Develop\repo_npm\node_modules\cnpm\bin\cnpm
+ cnpm@6.2.0
added 700 packages from 976 contributors in 19.796s

linux@GuardWhy MINGW64 /e/workspace/Web/ES6/base01
$

linux@GuardWhy MINGW64 /e/workspace/Web/ES6/base01
$ cnpm -v
cnpm@6.1.1 (C:\Users\linux\AppData\Roaming\npm\node_modules\cnpm\lib\parse_argv.js)
npm@6.14.4 (C:\Users\linux\AppData\Roaming\npm\node_modules\cnpm\node_modules\npm\lib\npm.js)
node@10.15.3 (C:\Program Files\nodejs\node.exe)
npminstall@3.27.0 (C:\Users\linux\AppData\Roaming\npm\node_modules\cnpm\node_modules\npminstall\lib\index.js)
prefix=E:\Develop\repo_npm
win32 x64 10.0.19042
registry=https://r.npm.taobao.org

linux@GuardWhy MINGW64 /e/workspace/Web/ES6/base01
$ cnpm config set prefix "E:\\Develop\\repo_npm"

linux@GuardWhy MINGW64 /e/workspace/Web/ES6/base01
$ cnpm config list
; cli configs
disturl = "https://npm.taobao.org/mirrors/node"
metrics-registry = "https://r.npm.taobao.org/"
registry = "https://r.npm.taobao.org/"
scope = ""
user-agent = "npm/6.14.4 node/v10.15.3 win32 x64"
userconfig = "C:\\Users\\linux\\.cnpmrc"

; userconfig C:\Users\linux\.cnpmrc
prefix = "E:\\Develop\\repo_npm"

; node bin location = C:\Program Files\nodejs\node.exe
; cwd = E:\workspace\Web\ES6\base01
; HOME = C:\Users\linux
; "npm config ls -l" to show all defaults.
```

2.2.5 cnpm install命令

```
cnpm install jquery@(指定版本号)
```

- 使用 npm install 安装依赖包的最新版
- 模块安装的位置：项目目录\node_modules
- 安装会自动在项目目录下添加 package-lock.json文件，这个文件帮助锁定安装包的版本
- 同时package.json 文件中，依赖包会被添加到dependencies节点下，类似maven中的 <dependencies>

> **执行结果**

3- ES6基本语法

ES标准中不包含 DOM 和 BOM的定义，只涵盖基本数据类型、关键字、语句、运算符、内建对象、内建函数等通用语法。

3.1 let声明变量

> 作用域不同

```
{
    var a = 1; // var声明的变量是全局变量
    let b = 2; // let声明的变量是局部变量
}

console.log(a);
// console.log(b); //b is not defined：b没有定义
```

> 声明次数不同

```
// var可以声明多次
// let只能声明一次
var m = 1;
var m = 2;
let n = 3;
let n = 4; //SyntaxError: Identifier 'n' has already been declared（语法错误：n已经声明过了）

console.log(m);
console.log(n);
```

> 声明与使用顺序不同

```
// var 声明的变量会全局存储
// let 声明的变量只能在执行后才存储

console.log( x ); //没有报错，输出：undefined
var a = "James";

console.log(y);  //y is not defined（y没有定义）
let b = "guardwhy";
```

3.2 const声明常量

const 声明常量,为只读变量

1. 一旦声明之后，其值是不允许改变的。
2. 一但声明必须初始化，否则会报错 SyntaxError: Missing initializer in const declaration（语法错误，声明常量丢失了初始化）

```
const PI = 3.14;
PI = 3.141;  //Assignment to constant variable.(声明的是常量)
-- 输出结果
console.log( PI );
```

3.3 解构赋值

- 解构赋值是对赋值运算符的扩展,它是一种针对**数组**或者**对象**进行模式匹配，然后对其中的变量进行赋值。
- 解构，顾名思义，就是将集合型数据进行分解，拆分，把里面的值逐一遍历获取
- 在代码书写上简洁且易读，语义更加清晰明了；也方便了复杂对象中数据字段获取。

> 数组解构

```
var arr = [1,2,3];

// 传统的js
let a1 = arr[0];
let b1 = arr[1];
let c1 = arr[2];
console.log(a1,b1,c1);

//es6的解构
var [a2,b2,c2] = arr;
console.log(a2,b2,c2);
```

> 对象解构

```
var user = {
    name : "guardwhy",
    age:"23",
    address:"广州"
};

// 传统的js
let userName = user.name;
let userAge = user.age;
let userAddress = user.address;
console.log("姓名:"+userName+",年龄:"+userAge+",地址:"+userAddress);

//es6的解构
let {name,age,address} = user;  // 注意：解构的变量名必须是对象中的属性
console.log("姓名:"+userName+",年龄:"+userAge+",地址:"+userAddress);
```

执行结果

```
linux@Guardwhy MINGW64 /e/workspace/Web/ES6/base01
$ node 03-数据解构.js
姓名:guardwhy,年龄:23,地址:广州
姓名:guardwhy,年龄:23,地址:广州

linux@Guardwhy MINGW64 /e/workspace/Web/ES6/base01
$
```

3.4 模板字符串

- 模板字符串相当于加强版的字符串
- 用反引号 `,除了作为普通字符串，还可以用来定义多行字符串，还可以在字符串中加入变量和表达式。

3.4.1 定义多行字符串

换行(` `), \n

```
let str = `hello,
ES6,
从入门到放弃`;
console.log(str);
```

3.4.2 字符串插入变量和表达式

```
let username = `guardwhy`;
let userage = 27;

// 传统的拼接字符串
var info1 = "我叫:"+ username +",我今年"+userage+"岁！";
console.log(info1);

// es6的拼接字符串
var info2 = `我叫:${username},我明年${userage+1}岁！`;
console.log(info2);
```

执行结果

```
linux@GuardWhy MINGW64 /e/workspace/Web/ES6/base01
$ node 04-模板字符串.js
我叫:guardwhy,我今年27岁！
我叫:guardwhy,我明年28岁！

linux@GuardWhy MINGW64 /e/workspace/Web/ES6/base01
$
```

3.4.3 字符串中调用函数

```
function test1(){
    return "努力Coding";
}

let str = `快乐的人生，从${test1()}开始`;
console.log( str );
```

3.5 声明对象简写

定义对象的时候，可以用变量名作为属性名。

```
let name = `guardwhy`;
let age = 28;
```

```
//传统方式
let user1 = {
    name : name,
    age : age
};

console.log(user1);

//es6新语法中的简写
let user2 = {name,age};
console.log(user2);
```

3.6 定义方法简写

```
// 传统
let func1 = {
    say : function(){
        console.log("hello ES6! ! ! ");
    }
};

func1.say();

//es6
let func2 = {
    say(){
        console.log("hello ES6!!!");
    }
};

func2.say();
```

4- 对象拓展运算符

拓展运算符 {...} 将参数对象中所有可以遍历的属性拿出来，然后拷贝给新对象。

4.1 拷贝对象(深拷贝)

```
let user1 = {
    username:"guardwhy",
    userage:28
};

let user2 = {...user1}; // 深拷贝（克隆）

console.log(user1);
console.log(user2);
```

4.2 合并对象

吞噬合并（两个对象合并成一个对象）

```
let user1 = {
    name:"guardwhy",
    age:27
};

let user2 = {head:"james"};

let user = {...user1,...user2};

console.log( user );
```

5-函数(Function)

5.1 函数的默认参数

形参处已声明，但不传入实参会怎样?

```
function test(name , age = 18){
    console.log(`我叫${name}，我今年${age}岁`);
}

test("kobe",33); //我叫kobe，我今年33岁
test("james"); //我叫james，我今年18岁
test("Rondo",null); //我叫Rondo，我今年null岁
test("guardwhy",""); //我叫guardwhy，我今年岁
test("Paul",undefined); //我叫Paul，我今年18岁
```

5.2 函数的不定参数

定义方法时，不确定有几个参数?

```
function test( ...arg ){
    console.log(`传入了${arg.length}个参数`);
    for(var i = 0 ;i<arg.length;i++){
        console.log(arg[i]);
    }
}

test(1);
test(1,2);
test(1,2,3,4,5,6);
test();
test("科","比",41);
```

5.3 箭头函数

箭头函数提供了一种更加简洁的函数书写方式。基本语法是：参数 => 函数体。

```
let 函数名称 = (形参列表) =>{
    需要封装的代码;
}
```

> **箭头函数的基本使用**

```
```

```
// 箭头函数: 也是一种定义函数的方式
  // 1.定义函数的方式: function
  const a1 = function () {
  }

// 2.对象字面量中定义函数
const obj = {
  b1() {
  }
}

// 3.ES6中的箭头函数
// const c1 = (参数列表) => {
//
// }
const c1 = () => {
}
```

箭头函数的参数和返回值

```
// 1.1.放入两个参数
const sum = (num1, num2) => {
    return num1 + num2
  }

  // 1.2.放入一个参数，小括号可以省略
  const power = num => {
    return num * num
  }

  // 2.1.函数代码块中有多行代码时
  const test = () => {
    // 1.打印Hello World
    console.log('Hello World');

    // 2.打印Hello Vuejs
    console.log('Hello Vuejs');
  }

  // 2.2.函数代码块中只有一行代码
  // const mul = (num1, num2) => {
  //   return num1 + num2
  // }
  const mul = (num1, num2) => num1 * num2
  console.log(mul(20, 30)); // 600

  // const demo = () => {
  //   console.log('Hello Demo');
  // }
  const demo = () => console.log('Hello ES6! ! ')
  console.log(demo()); // 结果是Hello ES6! !  undefined
```

箭头函数的this使用

箭头函数中的this是如何查找，向外层作用域中, 一层层查找this, 直到有this的定义。

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<script>
  const obj = {
    func() {
      setTimeout(function () {
        setTimeout(function () {
          console.log(this); // window
        })

        setTimeout(() => {
          console.log(this); // window
        })
      })

      setTimeout(() => {
        setTimeout(function () {
          console.log(this); // window
        })

        setTimeout(() => {
          console.log(this); // obj
        })
      })
    }
  }
  // 调用函数
  obj.func()
</script>
</body>
</html>
```

6- 模块化

- 如果在func1.js文件中定义了3个方法，现在func2.js文件中想使用func1中的3个方法，怎么办?
- java语言的做法是import引入之后，就能使用了。es6的模块化，就是这个过程。
- 将一个js文件声明成一个模块导出之后，另一个js文件才能引入这个模块。
- 每一个模块只加载一次（是单例的），若再去加载同目录下同文件，直接从内存中读取。

6.1 传统的模块化

创建test1.js文件

```
function addUser(name){
    return `保存${name}成功！`;
}

function removeUser(id){
    return `删除${id}号用户！`;
}

// 声明模块导出的简写
module.exports={
    addUser,
    removeUser
}
```

创建test2.js文件

```
let user = require("./test01"); // 引入test01模块

console.log(user);

let result1 = user.addUser("guardwhy");
let result2 = user.removeUser(11);

// 输出结果
console.log(result1);
console.log(result2);
```

执行结果

6.2 ES6的模块化

test01.js

```
let name = "guardwhy";
let age = 28;
let func1 = function(){
    return `我是${name}!我今年${age}岁了！`;
}

// 声明模块并导出
export{
    name,
    age,
    func1
}
```

test02.js

```
import {name,age, func1} from "./test01"

console.log(name);
console.log(age);
console.log(func1);
```

执行结果

运行test.js，报错：SyntaxError: Unexpected token {　（语法错误，在标记{的位置）

原因是node.js并不支持es6的import语法，我们需要将es6转换降级为es5！

```
SyntaxError: Unexpected token {
    at new Script (vm.js:80:7)
    at createScript (vm.js:274:10)
    at Object.runInThisContext (vm.js:326:10)
    at Module._compile (internal/modules/cjs/loader.js:664:28)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:712:10)
    at Module.load (internal/modules/cjs/loader.js:600:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:539:12)
    at Function.Module._load (internal/modules/cjs/loader.js:531:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:754:12)
    at startup (internal/bootstrap/node.js:283:19)
```

6.3 babel环境

babel是一个广泛使用的**转码器**，可以将ES6代码转为ES5代码，从而在现有的环境中执行。

6.3.1 安装babel客户端环境

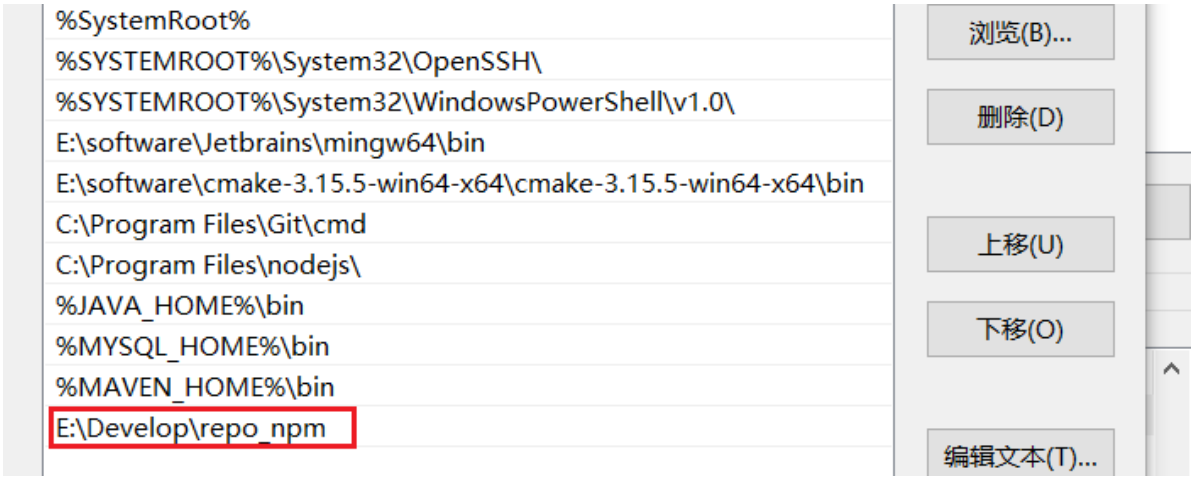
创建新目录 moduls03，在终端中打开，运行命令：

```
cnpm install --global babel-cli
```

查看版本

```
babel --version
```

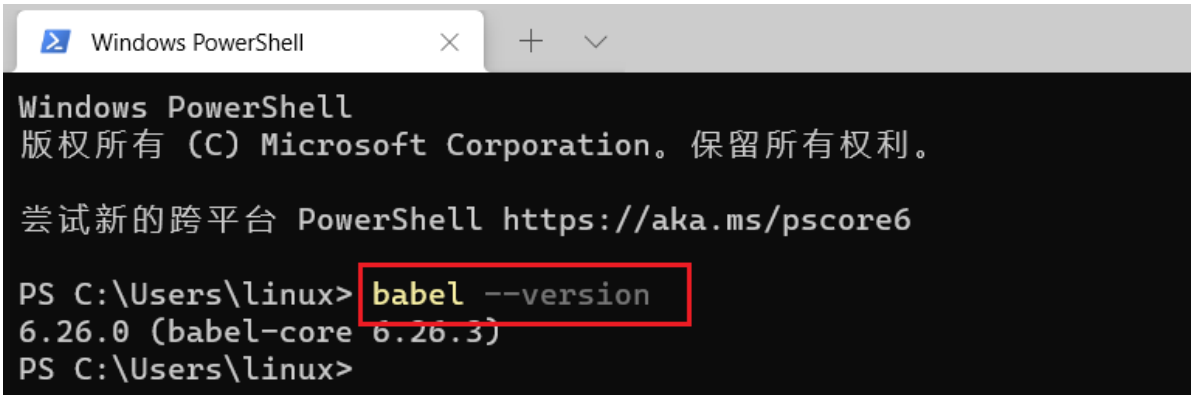
配置环境变量



开始菜单-> Windows PowerShell (切记要以管理员身份运行)

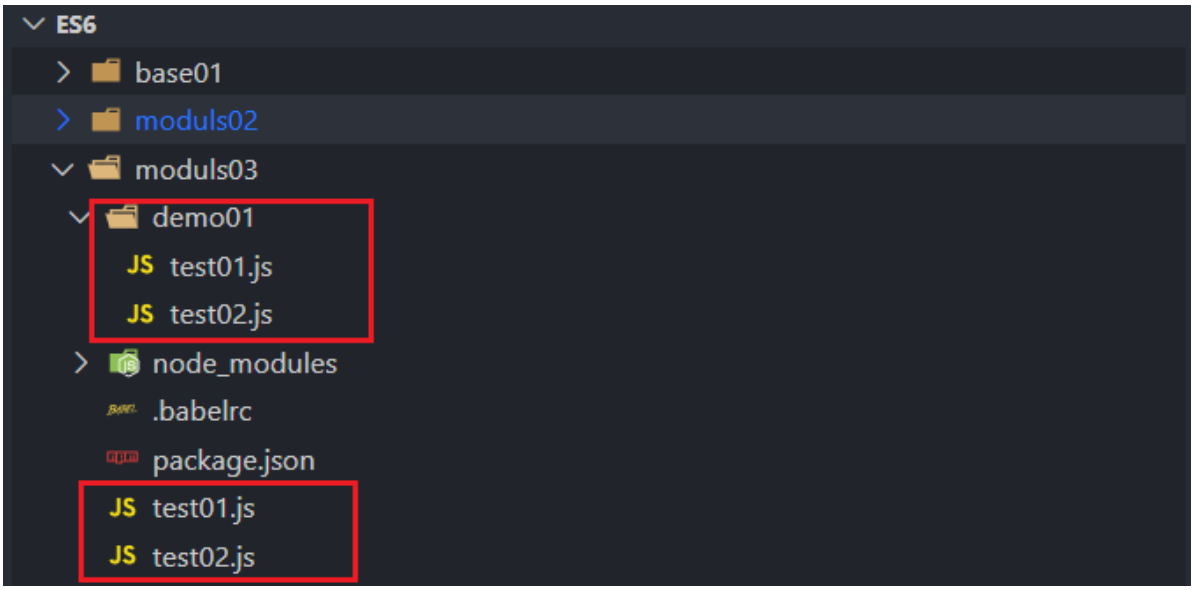
```
set-ExecutionPolicy RemoteSigned
```

查看版本



6.3.2 安装转码器

项目目录



1、初始化项目

```
npm init -y
```

2、创建babel配置文件 .babelrc ， 并输入代码配置

```
{
  "presets": ["es2015"],
  "plugins": []
}
```

3、安装转码器

```
cnpm install --save-dev babel-preset-es2015
```

4、转码创建demo01目录，用来存放转码后的文件

```
babel test01.js -o ./demo01/test01.js
或者
babel test01.js --out-file ./demo01/test01.js
```

5、运行转码后的文件

```
node ./demo01/test02.js
```

> **执行结果**

```
linux@GuardWhy MINGW64 /e/workspace/Web/ES6/moduls03
$ node ./demo01/test02.js
guardwhy
28
[Function: func1]

linux@GuardWhy MINGW64 /e/workspace/Web/ES6/moduls03
```

6.4 ES6模块化另一种写法

6.4.1 as的用法

user.js：如果不想暴露模块当中的变量名字，可以通过as来进行操作

```
let name = "guardwhy";
let age = 28;
let fn = function(){
    return `我是${name}!我今年${age}岁了！`;
}

// 声明模块并导出
export{
    name as a,
    age as b,
    fn as c
}
```

test03.js

```
import {a,b,c} from "./user.js";

console.log(a);
console.log(b);
console.log( c() );
```

也可以接收整个模块

test04.js

```
import * as info from "./user.js";   // 通过*来批量接收，as来指定接收的名字

console.log(info.a);
console.log(into.b);
console.log(into.c());
```

执行结果

```
linux@GuardWhy MINGW64 /e/workspace/Web/ES6/moduls03
$ node ./demo01/test03.js
guardwhy
28
我是guardwhy!我今年28岁了！
```

6.4.2 默认导出

可以将所有需要导出的变量放入一个对象中，然后通过default export进行导出。

people01.js

```
export default{
    name:"guardwhy",
    study(){
        return "好好学习，天天向上!!!";
    }
}
```

people02.js

```
import p from "./people01"
// 输出结果
console.log(p.name, p.study());
```

执行结果

```
linux@GuardWhy MINGW64 /e/workspace/Web/ES6/moduls03
$ babel people01.js -o ./demo01/people01.js

linux@GuardWhy MINGW64 /e/workspace/Web/ES6/moduls03
$ babel people02.js -o ./demo01/people02.js

linux@GuardWhy MINGW64 /e/workspace/Web/ES6/moduls03
$ node ./demo01/people02.js
guardwhy 好好学习，天天向上!!!
```

6.4.3 重命名export和import

如果导入的多个文件中，变量名字相同，即会产生命名冲突的问题，

为了解决该问题，ES6为提供了重命名的方法，当在导入名称时可以这样做

username1.js

```
export let name = "guardwhy";
```

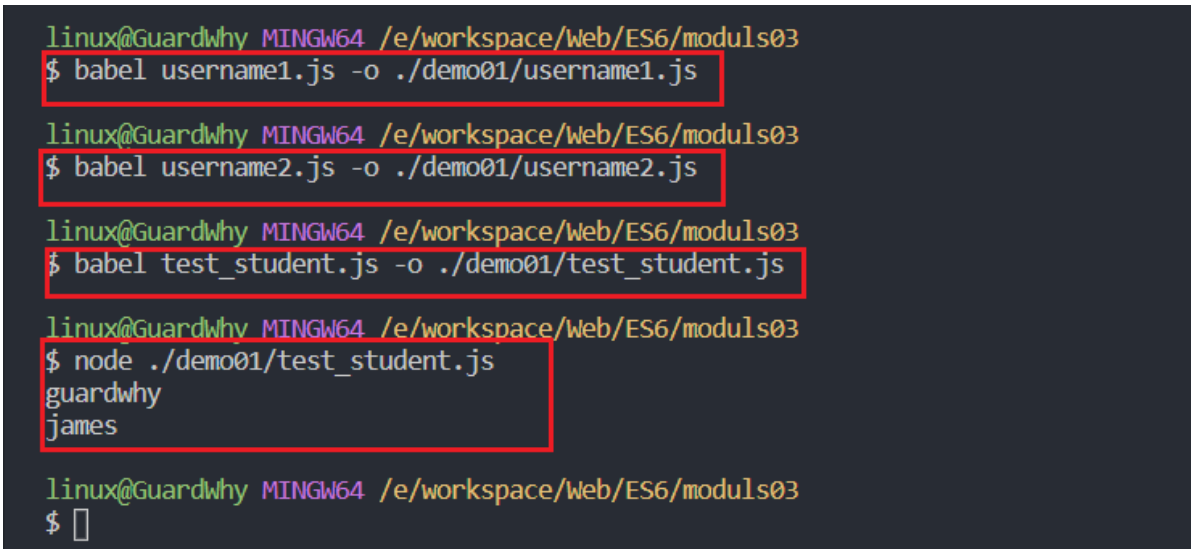
username2.js

```
export let name = "james";
```

test_student.js

```
import {name as name1} from './username1.js';
import {name as name2} from './username2.js';

console.log( name1 ); // 我是来自username1.js
console.log( name2 ); // 我是来自username2.js
```

> **执行结果**



7- Promise

7.1 什么是promise?

promise是ES6中新增的异步编程解决方案, 在代码中的表现是一个对象。

> **promise作用**

企业开发中为了保存异步代码的执行顺序, 那么就会出现回调函数层层嵌套。

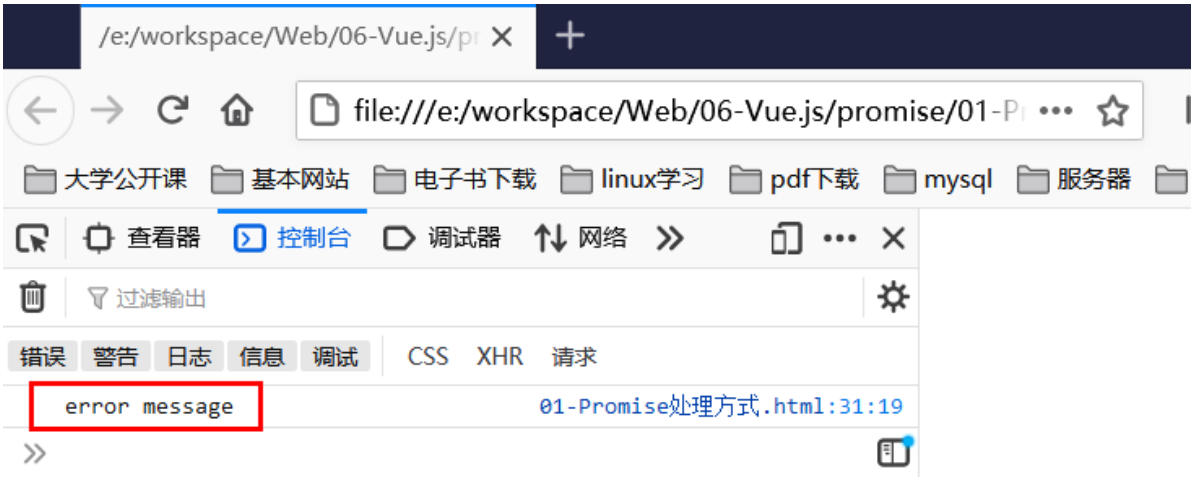
如果回调函数嵌套的层数太多, 就会导致代码的阅读性, 可维护性大大降低。

promise对象可以将异步操作以同步流程来表示, 避免了回调函数层层嵌套(回调地狱)。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>

<script>
```

```
  // 1.使用setTimeout
  // setTimeout(() => {
  //   console.log('Hello World');
  // }, 1000)

  // 什么情况下会用到Promise?
  // 一般情况下是有异步操作时,使用Promise对这个异步操作进行封装
  // new -> 构造函数(1.保存了一些状态信息  2.执行传入的函数)
  // 在执行传入的回调函数时, 会传入两个参数, resolve, reject.本身又是函数
  new Promise((resolve, reject) => {
    setTimeout(() => {
      // 成功的时候调用resolve
      // resolve('Hello World')

      // 失败的时候调用reject
      reject('error message')
    }, 1000)
  }).then((data) => {
    // 1.100行的处理代码
    console.log(data);
    console.log(data);
    console.log(data);
    console.log(data);
    console.log(data);
  }).catch((err) => {
    console.log(err);
  })
</script>
</body>
</html>
```

执行结果



7.2 如何创建Promise对象

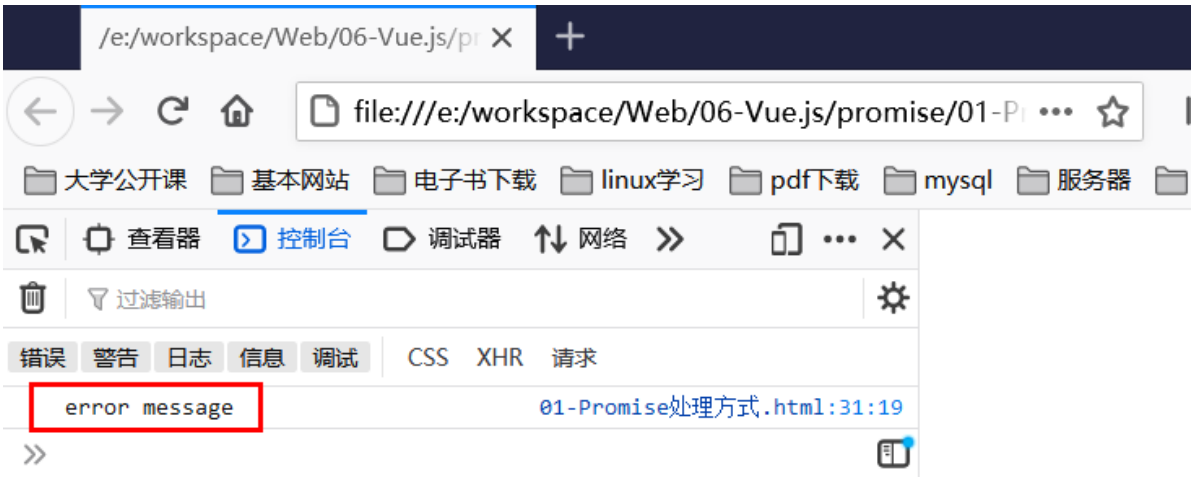
可以通过`new Promise(function(resolve, reject){ });`promise对象不是异步的, 只要创建promise对象就会立即执行存放的代码。

promise对象是通过状态的改变来实现的, 只要状态发生改变就会自动触发对应的函数。

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title></title>
</head>
<body>
    <script>
    /*什么情况下会用到Promise?
      一般情况下是有异步操作时,使用Promise对这个异步操作进行封装
      new -> 构造函数(1.保存了一些状态信息  2.执行传入的函数)
      在执行传入的回调函数时, 会传入两个参数, resolve, reject.本身又是函数
    */
      new Promise((resolve, reject) =>{
          setTimeout(() =>{
              // 1.成功的时候调用resolve
              // resolve("hello Vue.js!!!!")
              // 2.失败的时候调用reject
              reject('error message')
          }, 1000)
      }).then(data =>{
          console.log(data);
      },err =>{
          console.log(err);
      })
    </script>
</body>
</html>
```

执行结果



7.3 Promise对象三种状态

- pending: 默认状态，只要没有告诉promise任务是成功还是失败就是pending状态。
- fulfilled(resolved): 只要调用resolve函数, 状态就会变为fulfilled, 表示操作成功。
- rejected: 只要调用rejected函数, 状态就会变为rejected, 表示操作失败。

> **三种状态注意点**

状态一旦改变既不可逆, 既从pending变为fulfilled, 那么永远都是fulfilled。既从pending变为rejected, 那么永远都是rejected。

```
<!DOCTYPE html>
```

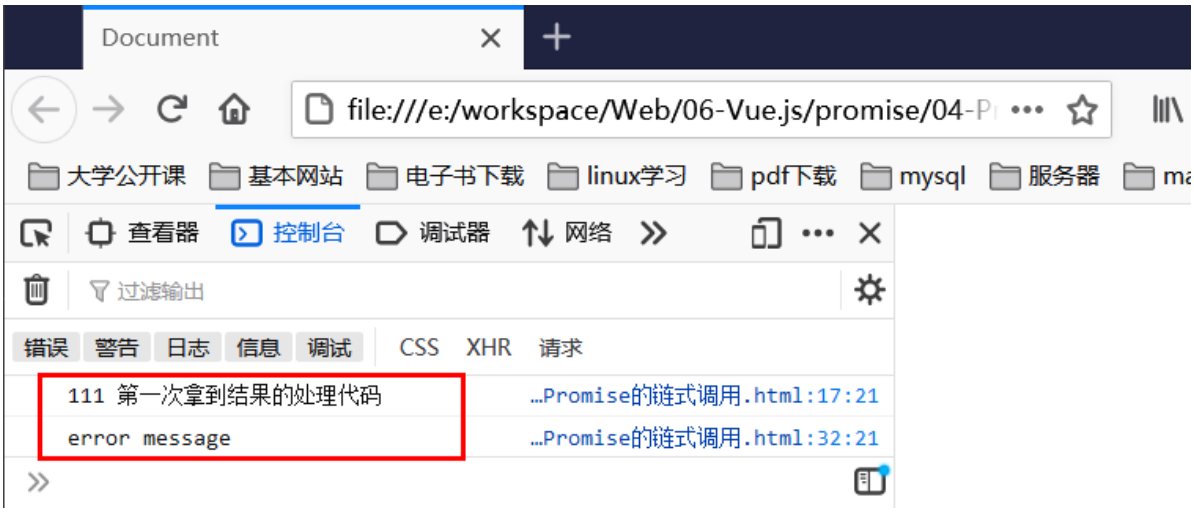
```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
   <script>
    /*什么情况下会用到Promise?
      一般情况下是有异步操作时,使用Promise对这个异步操作进行封装
      new -> 构造函数(1.保存了一些状态信息  2.执行传入的函数)
      在执行传入的回调函数时, 会传入两个参数, resolve, reject.本身又是函数
    */
    new Promise((resolve, reject) =>{
        // 第一次网络请求的代码
        setTimeout(() =>{
            resolve()
        },1000)
    }).then(() =>{
        // 第一次拿到结果的处理代码
        console.log("hello Vue.js!!!!");
        console.log("hello Vue.js!!!!");
        console.log("hello Vue.js!!!!");

        return new Promise((resolve, reject) =>{
              // 第二次网络请求
            setTimeout(() =>{
                resolve()
            }, 1000)
        })
    }).then(() =>{
        // 第二次拿到结果的处理代码
        console.log("hello python! ! ! ");
        console.log("hello python! ! ! ");
        console.log("hello python! ! ! ");
        console.log("hello python! ! ! ");

        return new Promise((resolve, reject) =>{
            // 第三次网络请求
            setTimeout(() =>{
                resolve()
            })
        })
    }).then(() =>{
        // 第二次拿到结果的处理代码
        console.log("hello javaScript! ! ! ");
        console.log("hello javaScript! ! ! ");
        console.log("hello javaScript! ! ! ");
        console.log("hello javaScript! ! ! ");
    })
   </script>
</body>
</html>
```

7.4 链式调用简写

可以将数据直接包装成`Promise.resolve`，那么在then中可以直接返回数据。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        new Promise((resolve, reject) =>{
            setTimeout(() =>{
                resolve('111')
            },1000)
        }).then(res =>{
            // 1.自己处理该行代码
            console.log(res, '第一次拿到结果的处理代码')
            // 2.对结果进行第一次处理
            // return Promise.resolve(res + '111');

            // return Promise.reject('error message')

            // 3.简写
            throw 'error message'

        }).then(res =>{
            console.log(res, '第二次拿到结果处理代码')
            return Promise.resolve(res + '222')
        }).then(res =>{
            console.log(res, "第三次拿到结果处理代码")
        }).catch(err =>{
            console.log(err);
        })
    </script>
</body>
</html>
```

执行结果

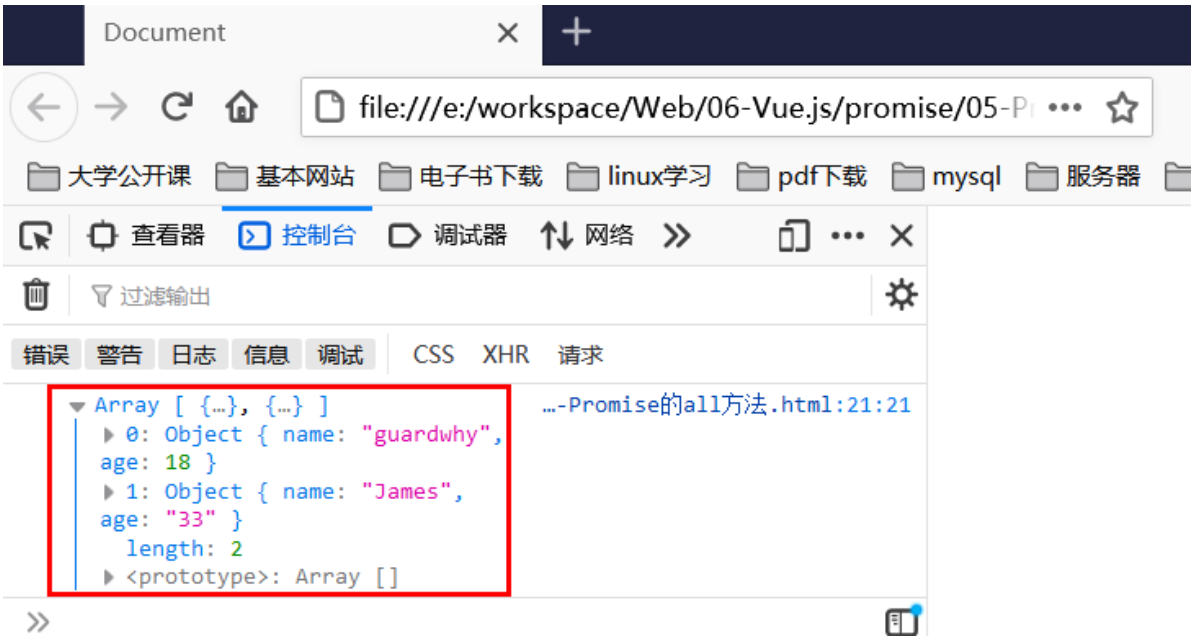


7.5 Promise的all静态方法

- all方法接收一个数组,如果数组中有多个Promise对象,只有都成功才会执行then方法。
- 并且会按照添加的顺序, 将所有成功的结果重新打包到一个数组中返回给我们。
- 如果数组中不是Promise对象, 那么会直接执行then方法。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        Promise.all([
            new Promise((resolve, reject) =>{
                setTimeout(() =>{
                    resolve({name:'guardwhy', age:18})
                },2000)
            }),
            new Promise((resolve, reject) =>{
                resolve({name: 'James', age:'33'})
            }, 1000)
        ]).then(results =>{
            console.log(results);
        })
    </script>
</body>
</html>
```

执行结果


Document
file:///e:/workspace/Web/06-Vue.js/promise/05-P
大学公开课
基本网站
电子书下载
linux学习
pdf下载
mysql
服务器
查看器
控制台
调试器
网络
过滤输出
错误
警告
日志
信息
调试
CSS
XHR
请求
Array [{…}, {…}]
0: Object { name: "guardwhy", age: 18 }
1: Object { name: "James", age: "33" }
length: 2
<prototype>: Array []
…-Promise的all方法.html:21:21