

第一章：认识node

1.1 什么是Node.js

Node.js是一个基于V8 JavaScript引擎的JavaScript运行时环境

1.2 浏览器内核是什么

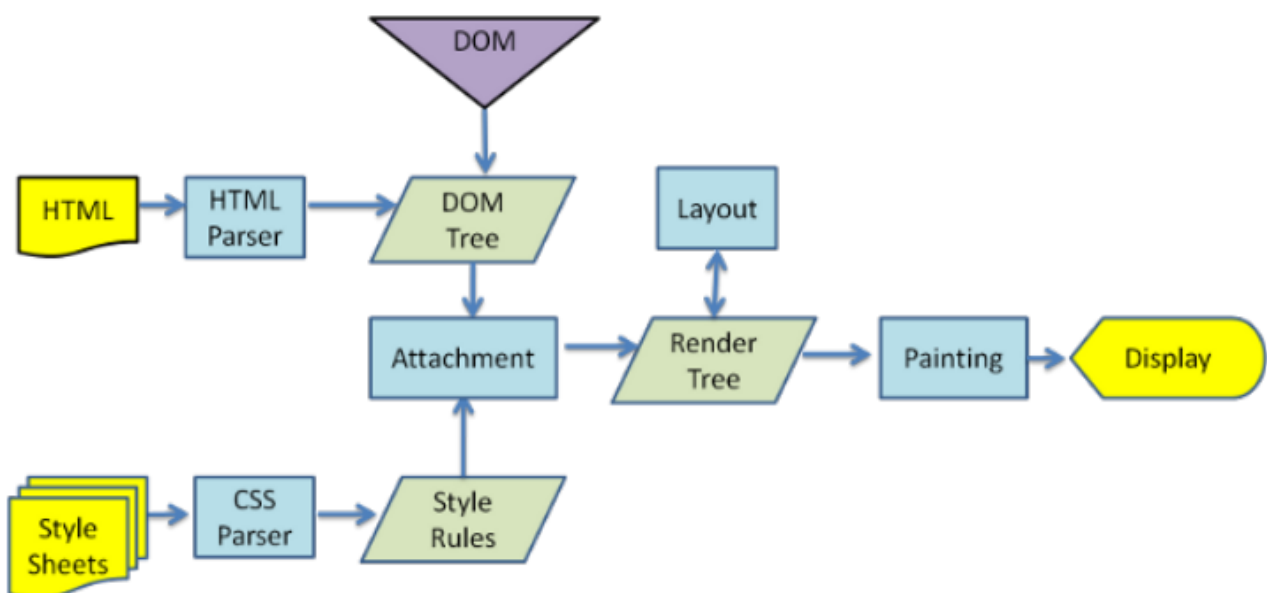
1.不同的浏览器有不同的内核组成：

- Gecko：早期被Netscape和Mozilla Firefox浏览器使用；
- Trident：微软开发，被IE4~IE11浏览器使用，但是Edge浏览器已经转向Blink；
- Webkit：苹果基于KHTML开发、开源的，用于Safari，Google Chrome之前也在使用
- Blink：是Webkit的一个分支，Google开发，目前应用于Google Chrome、Edge、Opera等；

2.浏览器内核指的是浏览器的排版引擎：

排版引擎（layout engine），也称为浏览器引擎（browser engine）、页面渲染引擎（rendering engine）或样版引擎。

1.3 渲染引擎工作的过程



1.4 JavaScript引擎

1.为什么需要JavaScript引擎呢？

事实上我们编写的JavaScript无论你交给浏览器或者Node执行，最后都是需要被CPU执行的；但是CPU只认识自己的指令集，实际上是机器语言，才能被CPU所执行；所以我们需要JavaScript引擎帮助我们将JavaScript代码翻译成CPU指令来执行；

2.比较常见的JavaScript引擎有哪些呢？

- SpiderMonkey：第一款JavaScript引擎，由Brendan Eich开发（也就是JavaScript作者）

- Chakra：微软开发，用于IT浏览器；
- JavaScriptCore：WebKit中的JavaScript引擎，Apple公司开发；
- V8：Google开发的强大JavaScript引擎，也帮助Chrome从众多浏览器中脱颖而出；

1.5 WebKit内核

1.WebKit事实上由两部分组成的：

- WebCore：负责HTML解析、布局、渲染等相关的工作
- JavaScriptCore：解析、执行JavaScript代码

1.6 V8引擎

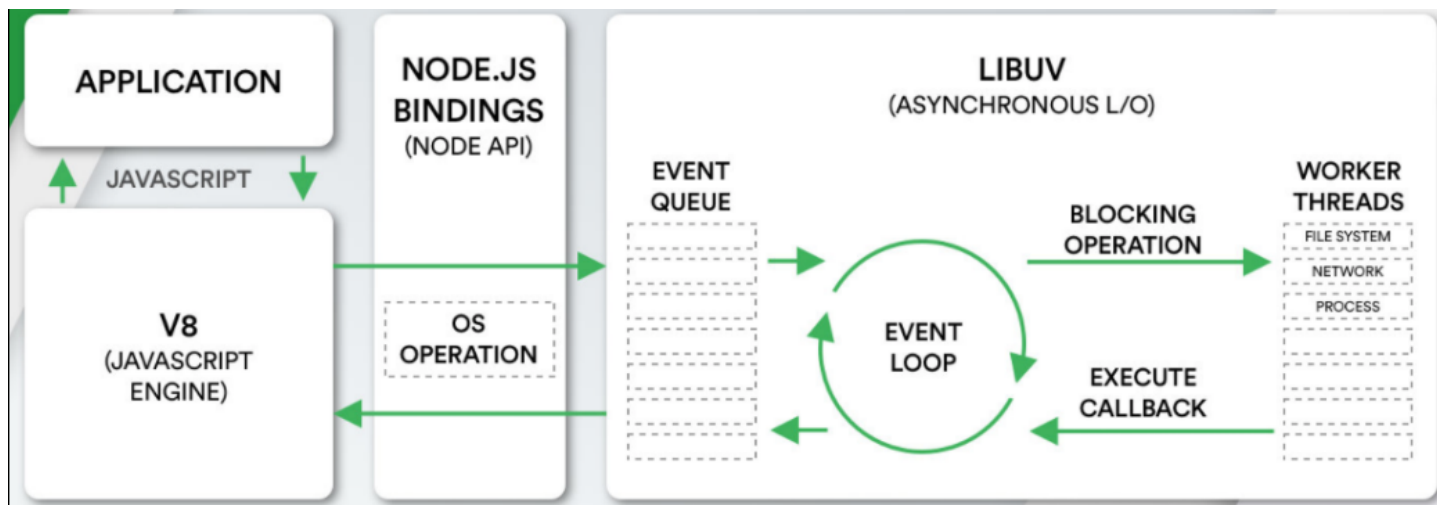
- V8是用C++编写的Google开源高性能JavaScript和WebAssembly引擎，它用于Chrome和Node.js等。
- 它实现ECMAScript和WebAssembly，并在Windows 7或更高版本，macOS 10.12+和使用x64，IA-32，ARM或MIPS处理器的Linux系统上运行
- V8可以独立运行，也可以嵌入到任何C++应用程序中。

V8引擎的原理：

- Parse模块会将JavaScript代码转换成AST（抽象语法树），这是因为解释器并不直接认识JavaScript代码
 - 如果函数没有被调用，那么是会被转换成AST的；
- Ignition是一个解释器，会将AST转换成ByteCode（字节码）
 - 同时会收集TurboFan优化所需要的信息（比如函数参数的类型信息，有了类型才能进行真实的运算）；
 - 如果函数只调用一次，Ignition会执行解释执行ByteCode
- TurboFan是一个编译器，可以将字节码编译为CPU可以直接执行的机器码；
 - 如果一个函数被多次调用，那么就会被标记为热点函数，那么就会经过TurboFan转换成优化的机器码，提高代码的执行性能
 - 但是，机器码实际上也会被还原为ByteCode，这是因为如果后续执行函数的过程中，类型发生了变化（比如sum函数原来执行的是number类型，后来执行变成了string类型），之前优化的机器码并不能正确的处理运算，就会逆向的转换成字节码；

1.7 Node.js架构

- 我们编写的JavaScript代码会经过V8引擎，再通过Node.js的Bindings，将任务放到Libuv的事件循环中
- libuv（Unicorn Velociraptor—独角伶盗龙）是使用C语言编写的库
- libuv提供了事件循环、文件系统读写、网络IO、线程池等等内容；



1.8 Node的版本工具

- nvm
- n

1.9 Node程序传递参数

```
1 node index.js env=development
```

获取参数其实是在process的内置对象中的，argv属性

```
1 console.log(process.argv)
2 process.argv.forEach(item=>{
3   console.log(item);
4 })
```

1.10 Node的输出

- console.log
- console.clear
- console.trace

1.11 常见的全局对象

- process对象：process提供了Node进程中相关的信息
 - 比如Node的运行环境、参数信息等
- console对象
- 定时器函数：在Node中使用定时器有好几种方式
 - setTimeout
 - setInterval

- `setImmediate`: I/O事件后的回调的“立即”执行;
- `process.nextTick`: 添加到下一次tick队列中

1.12 特殊的全局对象

`__dirname`、`__filename`、`exports`、`module`、`require()`

- `__dirname`: 获取当前文件所在的路径
- `__filename`: 获取当前文件所在的路径和文件名称

1.13 global和window的区别

- 在浏览器中, 全局变量都是在window上的, 比如有`document`、`setInterval`、`setTimeout`、`alert`、`console`等等
- 在Node中, 我们也有一个`global`属性, 并且看起来它里面有很多其他对象
- 但是在浏览器中执行的JavaScript代码, 如果我们在顶级范围内通过`var`定义的一个属性, 默认会被添加到`window`对象上
- 但是在node中, 我们通过`var`定义一个变量, 它只是在当前模块中有一个变量, 不会放到全局中

第二章: JavaScript模块化

2.1 什么是模块化

1.模块化开发

- 事实上模块化开发最终的目的是将程序划分成一个个小的结构
- 这个结构中编写属于自己的逻辑代码, 有自己的作用域, 不会影响到其他的结构
- 这个结构可以将自己希望暴露的变量、函数、对象等导出给其他结构使用
- 也可以通过某种方式, 导入另外结构中的变量、函数、对象等

2.上面说提到的结构, 就是模块; 按照这种结构划分开发程序的过程, 就是模块化开发的过程;

2.2 CommonJS和Node

1.我们需要知道CommonJS是一个规范, 最初提出来是在浏览器以外的地方使用, 并且当时被命名为`ServerJS`, 后来为了体现它的广泛性, 修改为`CommonJS`, 平时我们也会简称为CJS。

- Node是CommonJS在服务器端一个具有代表性的实现
- `Browserify`是CommonJS在浏览器中的一种实现;
- `webpack`打包工具具备对CommonJS的支持和转换;

2.Node中对CommonJS进行了支持和实现, 让我们在开发node的过程中可以方便的进行模块化开发:

- 在Node中每一个js文件都是一个单独的模块
- 这个模块中包括CommonJS规范的核心变量: `exports`、`module.exports`、`require`
- 我们可以使用这些变量来方便的进行模块化开发

3.Node导入导出

- exports和module.exports可以负责对模块中的内容进行导出
- require函数可以帮助我们导入其他模块（自定义模块、系统模块、第三方库模块）中的内容

2.3 module.exports又是什么

CommonJS中是没有module.exports的概念的，但是为了实现模块的导出，Node中使用的是Module的类，每一个模块都是Module的一个实例，也就是 module；所以在Node中真正用于导出的其实根本不是 exports，而是module.exports因为module才是导出的真正实现者。

为什么exports也可以导出呢？

这是因为module对象的exports属性是exports对象的一个引用

2.4 require细节

require是一个函数，可以帮助我们引入一个文件（模块）中导入的对象。

导入格式如下：require(X)

- 情况一：X是一个核心模块，比如path、http
 - 直接返回核心模块，并且停止查找
- X是以 ./ 或 ../ 或 /（根目录）开头的
 - 第一步：将X当做一个文件在对应的目录下查找
 - 如果有后缀名，按照后缀名的格式查找对应的文件
 - 如果没有后缀名，会按照如下顺序
 - 直接查找文件X
 - 查找X.js文件
 - 查找X.json文件
 - 查找X.node文件
 - 第二步：没有找到对应的文件，将X作为一个目录
 - 查找目录下面的index文件
 - 查找X/index.js文件
 - 查找X/index.json文件
 - 查找X/index.node文件
 - 如果没有找到，那么报错：not found
- 直接是一个X（没有路径），并且X不是一个核心模块
 - 在node_modules下面查找，会一层的找，包括电脑下的users文件夹到你编写代码的这条路径中的node_modules下面去找。
 - 如果上面的路径中都没有找到，那么报错：not found

2.5 模块的加载过程

- 模块在被第一次引入时，模块中的js代码会被运行一次

- 模块被多次引入时，会缓存，最终只加载（运行）一次
 - 为什么只会加载运行一次呢？
 - 这是因为每个模块对象module都有一个属性：loaded,为false表示还没有加载，为true表示已经加载；
- 如果有循环引入，那么加载顺序是什么
 - 图结构在遍历的过程中，有深度优先搜索和广度优先搜索
 - Node采用的是深度优先算法

```
1 Module._load = function(request, parent, isMain) {
2   let relResolveCacheIdentifier;
3   if (parent) {
4     debug('Module._load REQUEST %s parent: %s', request, parent.id);
5     // Fast path for (lazy loaded) modules in the same directory. The indirect
6     // caching is required to allow cache invalidation without changing the old
7     // cache key names.
8     relResolveCacheIdentifier = `${parent.path}\\x00${request}`;
9     const filename = relativeResolveCache[relResolveCacheIdentifier];
10    reportModuleToWatchMode(filename);
11    if (filename !== undefined) {
12      const cachedModule = Module._cache[filename];
13      if (cachedModule !== undefined) {
14        updateChildren(parent, cachedModule, true);
15        if (!cachedModule.loaded)
16          return getExportsForCircularRequire(cachedModule);
17        return cachedModule.exports;
18      }
19      delete relativeResolveCache[relResolveCacheIdentifier];
20    }
21  }
22
23  if (StringPrototypeStartsWith(request, 'node:')) {
24    // Slice 'node:' prefix
25    const id = StringPrototypeSlice(request, 5);
26
27    const module = loadBuiltinModule(id, request);
28    if (!module?.canBeRequiredByUsers) {
29      throw new ERR_UNKNOWN_BUILTIN_MODULE(request);
30    }
31
32    return module.exports;
```

```
33 }
34
35 const filename = Module._resolveFilename(request, parent, isMain);
36 const cachedModule = Module._cache[filename];
37 if (cachedModule !== undefined) {
38     updateChildren(parent, cachedModule, true);
39     if (!cachedModule.loaded) {
40         const parseCachedModule = cjsParseCache.get(cachedModule);
41         if (!parseCachedModule || parseCachedModule.loaded)
42             return getExportsForCircularRequire(cachedModule);
43         parseCachedModule.loaded = true;
44     } else {
45         return cachedModule.exports;
46     }
47 }
48
49 const mod = loadBuiltinModule(filename, request);
50 if (mod?.canBeRequiredByUsers &&
51     BuiltinModule.canBeRequiredWithoutScheme(filename)) {
52     return mod.exports;
53 }
54
55 // Don't call updateChildren(), Module constructor already does.
56 const module = cachedModule || new Module(filename, parent);
57
58 if (isMain) {
59     process.mainModule = module;
60     module.id = '.';
61 }
62
63 reportModuleToWatchMode(filename);
64
65 Module._cache[filename] = module;
66 if (parent !== undefined) {
67     relativeResolveCache[relResolveCacheIdentifier] = filename;
68 }
69
70 let threw = true;
71 try {
```

```

72     module.load(filename);
73     threw = false;
74 } finally {
75     if (threw) {
76         delete Module._cache[filename];
77         if (parent !== undefined) {
78             delete relativeResolveCache[relResolveCacheIdentifier];
79             const children = parent?.children;
80             if (Array.isArray(children)) {
81                 const index = Array.prototype.indexOf(children, module);
82                 if (index !== -1) {
83                     Array.prototype.splice(children, index, 1);
84                 }
85             }
86         }
87     } else if (module.exports &&
88                 !isProxy(module.exports) &&
89                 Object.getPrototypeOf(module.exports) ===
90                 CircularRequire.prototypeWarningProxy) {
91         Object.setPrototypeOf(module.exports, Object.prototype);
92     }
93 }
94
95 return module.exports;
96 };

```

2.6 CommonJS规范缺点

- CommonJS加载模块是同步的
 - 同步的意味着只有等到对应的模块加载完毕，当前模块中的内容才能被运行
- 在早期为了可以在浏览器中使用模块化，通常会采用AMD或CMD：
 - 但是目前一方面现代的浏览器已经支持ES Modules，另一方面借助于webpack等工具可以实现对CommonJS或者 ES Module代码的转换；

2.7 AMD规范

1. AMD主要是应用于浏览器的一种模块化规范：

- 它采用的是异步加载模块
- AMD实现的比较常用的库是require.js和curl.js

2.require.js

- 第一步：下载require.js: <https://github.com/requirejs/requirejs>
- 第二步：定义HTML的script标签引入require.js和定义入口文件
 - data-main属性的作用是在加载完src的文件后会加载执行该文件。
 - `<script src="./lib/require.js" data-main="./index.js"></script>`

```
1 <script src="./lib/require.js" data-main="./index.js"></script>
```

index.js 将模块产生映射关系

```
1 ;(function () {
2   require.config({
3     baseUrl: '',
4     paths: {
5       'bar': './modules/bar',
6       'info': './modules/info'
7     }
8   })
9
10  require(['bar'], function (bar) {})
11 })()
12
```

info.js

```
1 define(function () {
2   const name = 'yihua'
3   const age = 18
4   const sayHello = function (name) {
5     console.log('你好' + name)
6   }
7
8   return {
9     name,
10    age,
11    sayHello
12  })
13}
```

bar.js

```
1 define(['info'], function (info) {
2   console.log(info.name)
3   console.log(info.age)
4 })
```

```
4   info.sayHello('node')
5 })
```

2.8 CMD规范

1.CMD规范也是应用于浏览器的一种模块化规范

- 采用了异步加载模块，但是它将CommonJS的优点吸收了过来
- SeaJS

2.SeaJS

- 第一步：下载SeaJS: <https://github.com/seajs/seajs>
- 第二步：定义HTML的script标签引入require.js和定义入口文件
 - data-main属性的作用是在加载完src的文件后会加载执行该文件

```
1 <script src="./lib/sea.js"></script>
2 <script>
3   seajs.use('./index.js');
4 </script>
```

info.js

```
1 define(function(require, exports, module) {
2   const name = "yihua";
3   const age = 20;
4   const sayHello = function(name) {
5     console.log("你好" + name);
6   }
7
8   module.exports = {
9     name: name,
10    age: age,
11    sayHello: sayHello
12  }
13 });
```

index.js

```
1 define(function(require, exports, module) {
2   const info = require('./modules/info');
3   console.log(info.name);
4   console.log(info.age);
5   foo.sayHello("liu");
```

2.9 ES Module

1. ES Module 模块采用 export 和 import 关键字来实现模块化

- export 负责将模块内的内容导出
- import 负责从其他模块导入内容

2. ES Module 和 CommonJS 的模块化有一些不同之处

- 一方面它使用了 import 和 export 关键字
- 另一方面它采用编译期的静态分析，并且也加入了动态引用的方式

3. export 关键字

- 方式一：在语句声明的前面直接加上 export 关键字
- 方式二：将所有需要导出的标识符，放到 export 后面的 {} 中
 - 这里的 {} 里面不是 ES6 的对象字面量的增强写法，{} 也不是表示一个对象的；
- 方式三：导出时给标识符起一个别名

```
1  const name = "yihua";
2  const age = 18;
3  const sayHello = function(name) {
4      console.log("你好" + name);
5  }
6
7  // 导出方式三种
8  // 1. 方式一：
9  // export const name = "yihua";
10 // export const age = 18;
11 // export const sayHello = function(name) {
12 //     console.log("你好" + name);
13 // }
14
15 // 2. 方式二：{} 中统一导出
16 // {} 大括号，但是不是一个对象
17 // {} 放置要导出的变量的引用列表
18 export {
19     name,
20     age,
21     sayHello
22 }
23
```

```
24 // 3.方式三: {} 导出时, 可以给变量起名别
25 // export {
26 //   name as fName,
27 //   age as fAge,
28 //   sayHello as fSayHello
29 // }
```

4.import关键字

- 方式一: import {标识符列表} from '模块';
 - 这里的{}也不是一个对象, 里面只是存放导入的标识符列表内容;
- 方式二: 导入时给标识符起别名
- 方式三: 通过 * 将模块功能放到一个模块功能对象 (a module object) 上

```
1 // 方式一: import {} from '路径';
2 // import { name, age, sayHello } from './modules/foo.js';
3
4 // 方式二: 导出变量之后可以起别名
5 // import { name as wName, age as wAge, sayHello as wSayHello } from './modules/foo.js';
6 // import { fName as wName, fAge as wAge, fSayHello as wSayHello } from
  './modules/foo.js';
7
8 // 方式三: * as foo
9 // import * as foo from './modules/foo.js';
10
11 // console.log(foo.name);
12 // console.log(foo.age);
13 // foo.sayHello("王小波");
```

5.Export和import结合使用

index.js

```
1 import * as my from './modules/my.js'
2 console.log(my.name);
3 console.log(my.age);
4 console.log(my.english);
5 console.log(my.math);
```

my.js

```
1 export {age , name} from './info.js'
2 export {math , english} from './source.js'
```

info.js

```
1 const name = 'yihua'
2 const age = 18
3 export {
4   name,
5   age
6 }
```

source.js

```
1 const math = 98
2 const english = 98
3 export {
4   math,
5   english
6 }
```

6.default用法

默认导出export时不需要指定名字。在一个模块中，只能有一个默认导出（default export）

format.js

```
1 export default function() {
2   console.log("对某一个东西进行格式化");
3 }
```

index.js

```
1 import format from './format.js'
```

7.import函数

通过import加载一个模块，是不可以在其放到逻辑代码中的。为什么会出现这个情况呢？

- 这是因为ES Module在被JS引擎解析时，就必须知道它的依赖关系
- 由于这个时候js代码没有任何的运行，所以无法在进行类似于if判断中根据代码的执行情况
- 甚至下面的这种写法也是错误的：因为我们必须到运行时能确定path的值

错误的写法

```
1 if(flag){
2   import foo from './foo'
3 }
```

正确写法

```

1  let flag = true;
2  if (flag) {
3    // require的本质是一个函数
4    // require('')
5
6    // 执行函数
7    // 如果是webpack的环境下：模块化打包工具：es CommonJS require()
8    // 纯ES Module环境下面：import()
9    // 脚手架 -> webpack: import()
10   import('./foo.js').then(res => {
11     console.log("在then中的打印");
12     console.log(res.name);
13     console.log(res.age);
14   }).catch(err => {
15
16   })
17 }

```

8.ES Module加载过程

- ES Module加载js文件的过程是编译（解析）时加载的，并且是异步的：
 - 编译时（解析）时加载，意味着import不能和运行时相关的内容放在一起使用
 - 比如from后面的路径需要动态获取
 - 比如不能将import放到if等语句的代码块中
 - 所以我们有时候也称ES Module是静态解析的，而不是动态或者运行时解析的
- JS引擎在遇到import时会去获取这个js文件，但是这个获取的过程是异步的，并不会阻塞主线程继续执行
 - 也就是说设置了 type=module 的代码，相当于在script标签上也加上了 async 属性；
 - 如果我们后面有普通的script标签以及对应的代码，那么ES Module对应的js文件和代码不会阻塞它们的执行
- ES Module通过export导出的是变量本身的引用：
 - export在导出一个变量时，js引擎会解析这个语法，并且创建模块环境记录
 - 模块环境记录会和变量进行 绑定（binding），并且这个绑定是实时的
 - 而在导入的地方，我们是可以实时的获取到绑定的最新值的

2.10 Node对ES Module的支持

- 方式一：在package.json中配置 type: module
- 方式二：文件以 .mjs 结尾，表示使用的是ES Module

2.11 CommonJS和ES Module交互

- 结论一：通常情况下，CommonJS不能加载ES Module

- 因为CommonJS是同步加载的，但是ES Module必须经过静态分析等，无法在这个时候执行JavaScript代码
- 但是这个并非绝对的，某些平台在实现的时候可以对代码进行针对性的解析，也可能会支持
- Node当中是不支持的
- 结论二：多数情况下，ES Module可以加载CommonJS
 - ES Module在加载CommonJS时，会将其module.exports导出的内容作为default导出方式使用
 - 这个依然需要看具体的实现，比如webpack中是支持的、Node最新的Current版本也是支持的

第三章：常用内置模块

3.1 内置模块path

- path模块用于对路径和文件进行处理，提供了很多好用的方法。
- 并且我们知道在Mac OS、Linux和window上的路径时不一样的
 - window上会使用 \ 或者 \\ 来作为文件路径的分隔符，当然目前也支持 /
 - 在Mac OS、Linux的Unix操作系统上使用 / 来作为文件路径的分隔符
- 可移植操作系统接口
 - Linux和Mac OS都实现了POSIX接口
 - Window部分电脑实现了POSIX接口

1. path常见的API

- 从路径中获取信息
 - dirname：获取文件的父文件夹
 - basename：获取文件名；
 - extname：获取文件扩展名
- 路径的拼接
 - 如果我们希望将多个路径进行拼接，但是不同的操作系统可能使用的是不同的分隔符
 - 这个时候我们可以使用path.join函数
- 将文件和某个文件夹拼接
 - 如果我们希望将某个文件和文件夹拼接，可以使用 path.resolve
 - resolve函数会判断我们拼接的路径前面是否有 /或../或./;
 - 如果有表示是一个绝对路径，会返回对应的拼接路径
 - 如果没有，那么会和当前执行文件所在的文件夹进行路径的拼接

```
1  const path = require('path')
2
3  const filePath = 'stu_node/learn_node/03_path/test.txt'
4
5  // 1. 获取路径信息
6  console.log(path.dirname(filePath)) //stu_node/learn_node/03_path
7  console.log(path.basename(filePath)) //test.txt
```

```

8 console.log(path.extname(filePath)) // .txt
9
10 // 2.join路径拼接
11 const basePath = 'stu_node/learn_node/03_path'
12 const fileName = 'test.txt'
13 console.log(path.join(basePath, fileName)) // stu_node\learn_node\03_path\test.txt
14
15 // 3.resolve拼接,会判断路径中是否有./ ../
16 console.log(path.resolve(basePath, fileName))
// C:\Users\yihua\Desktop\code\stu_node\learn_node\03_path\stu_node\learn_node\03_path\test.txt

```

3.2 内置模块fs

借助于Node帮我们封装的文件系统，我们可以在任何的操作系统（window、Mac OS、Linux）上面直接去 操作文件

1.API大多数都提供三种操作方式：

- 同步操作文件：代码会被阻塞，不会继续执行；
- 异步回调函数操作文件：代码不会被阻塞，需要传入回调函数，当获取到结果时，回调函数被执行
- 异步Promise操作文件：代码不会被阻塞，通过 fs.promises 调用方法操作，会返回一个Promise， 可以通过 then、catch 进行处理；

```

1 const fs = require('fs')
2
3 // 读取文件信息
4 const filePath = './abc.txt'
5 // 1.同步操作
6 const info = fs.statSync(filePath)
7 // 后续的代码会被阻塞
8 console.log(info)
9
10 // 2.异步操作,不阻塞
11 fs.stat(filePath, (err, info) => {
12   if (err) {
13     console.log(err);
14   } else {
15     console.log(info);
16   }
17 })

```



```
18
19 // 3.promise,不阻塞
20 fs.promises.stat(filePath).then(info=>{
21   console.log(info);
22 }).catch(err=>{
23   console.log(err);
24 })
```

2.文件描述符

2.1 是什么？

- 在 POSIX 系统上，对于每个进程，内核都维护着一张当前打开着的文件和资源的表格
- 每个打开的文件都分配了一个称为文件描述符的简单的数字标识符
- 在系统层，所有文件系统操作都使用这些文件描述符来标识和跟踪每个特定的文件。
- Windows 系统使用了一个虽然不同但概念上类似的机制来跟踪资源

2.2 为了简化用户的工作，Node.js 抽象出操作系统之间的特定差异，并为所有打开的文件分配一个数字型的文件描述符

2.3 fs.open() 方法用于分配新的文件描述符

- 一旦被分配，则文件描述符可用于从文件读取数据、向文件写入数据、或请求关于文件的信息

```
1  const fs = require('fs')
2
3  fs.open('./abc.txt',(err,fd)=>{
4    if(err){
5      console.log(err);
6    }else{
7      console.log(fd);
8      // 通过文件描述符去操作文件
9      fs.fstat(fd,(err,info)=>{
10        console.log(info);
11      })
12    }
13  })
```

3.文件的读写

flag选项

- w 打开文件写入，默认值
- w+打开文件进行读写，如果不存在则创建文件；
- r+ 打开文件进行读写，如果不存在那么抛出异常

- r打开文件读取，读取时的默认值
- a打开要写入的文件，将流放在文件末尾。如果不存在则创建文件
- a+打开文件以进行读写，将流放在文件末尾。如果不存在则创建文件

```
1  const fs = require('fs');
2
3  // 1.文件写入
4  const content = "hello node";
5
6  fs.writeFile('./abc.txt', content, {flag: "a"}, err => {
7    console.log(err);
8  });
9
10 // 2.文件读取
11 fs.readFile("./abc.txt", {encoding: 'utf-8'}, (err, data) => {
12   console.log(data);
13 });
```

encoding选项: <https://www.jianshu.com/p/899e749be47c>

4.文件夹操作

```
1  const fs = require('fs')
2  const path = require('path')
3
4  // 1.创建文件夹
5  const dirname = './yihua'
6  // if (!fs.existsSync(dirname)) {
7  //   fs.mkdir(dirname, err => {
8  //     console.log(err)
9  //   })
10 // }
11
12 // 2.读取文件夹中的所有文件
13 fs.readdir(dirname, (err, files) => {
14   console.log(files)
15 })
16
17 function getFiles (dirname) {
18   fs.readdir(dirname, { withFileTypes: true }, (err, files) => {
```

```

19   for (let file of files) {
20     // fs.stat(file) 可以, 但是有点麻烦
21     if (file.isDirectory()) {
22       const filepath = path.resolve(dirname, file.name)
23       getFiles(filepath)
24     } else {
25       console.log(file.name)
26     }
27   }
28 })
29 }
30
31 getFiles(dirname)
32
33 // 3.重命名
34 fs.rename('./yihua', './test', err => {
35   console.log(err)
36 })

```

5.文件夹的复制

```

1  const fs = require('fs')
2  const path = require('path')
3
4  const sourceDir = 'source'
5  const targetDir = './target'
6
7  const walkSync = (curDirPath, cb) => {
8    fs.readdirSync(curDirPath, {withFileTypes: true}).forEach(dir => {
9      const filePath = path.join(curDirPath, dir.name)
10     if (dir.isFile()) {
11       cb(filePath)
12     } else if (dir.isDirectory()) {
13       const dirname = path.join(curDirPath.replace(sourceDir, targetDir), dir.name)
14       // 创建文件夹
15       if (!fs.existsSync(dirname)) {
16         fs.mkdir(dirname, err=>{
17           console.log(err);
18         })

```

```

19     }
20     walkSync(filePath, cb)
21   }
22 })
23 }
24
25 const copyFile = filePath => {
26   console.log(filePath, filePath.replace(sourceDir, targetDir))
27   fs.copyFile(filePath, filePath.replace(sourceDir, targetDir), (err) => {
28     console.log(err)
29   })
30 }
31
32 walkSync(sourceDir, copyFile)

```

3.3 events模块

发出事件和监听事件都是通过EventEmitter类来完成的，它们都属于events对象。

- emitter.on(eventName, listener): 监听事件，也可以使用 addListener;
- emitter.off(eventName, listener): 移除事件监听，也可以使用removeListener;
- emitter.emit(eventName[, ...args]): 发出事件，可以携带一些参数;

```

1  const EventEmitter = require('events')
2
3  // 1.创建发射器
4  const emitter = new EventEmitter()
5
6  // 2.监听事件
7  emitter.on('click', (...args) => {
8    console.log('监听1', args)
9  })
10
11 const listener2 = (...args) => {
12   console.log('监听2到click事件', args)
13 }
14 emitter.on('click', listener2)
15
16 // 3.触发事件
17 setTimeout(() => {
18   emitter.emit('click', 'yihua', 18, '男')

```

```
19 emitter.off('click', listener2)
20 emitter.emit('click', 'yihua', 18, '男')
21 }, 2000)
```

1.常见的属性

- emitter.eventNames(): 返回当前 EventEmitter对象注册的事件字符串数组;
- emitter.getMaxListeners(): 返回当前 EventEmitter对象的最大监听器数量, 可以通过setMaxListeners() 来修改, 默认是10;
- emitter.listenerCount(事件名称): 返回当前 EventEmitter对象某一个事件名称, 监听器的个数
- emitter.listeners(事件名称): 返回当前 EventEmitter对象某个事件监听器上所有的监听器数组;

```
1 console.log(emitter.eventNames())
2 console.log(emitter.getMaxListeners())
3 console.log(emitter.listenerCount('click'))
4 console.log(emitter.listeners('click'))
```

2.方法的补充

- emitter.once(eventName, listener): 事件监听一次
- emitter.prependListener(): 将监听事件添加到最前面
- emitter.prependOnceListener(): 将监听事件添加到最前面, 但是只监听一次
- emitter.removeAllListeners([eventName]): 移除所有的监听器

```
1 // 只执行一次
2 emitter.once('click', (...args) => {
3   console.log('监听1到click事件', args)
4 })
5
6 // 将本次监听放到最前面
7 emitter.prependListener('click', (...args) => {
8   console.log('监听2到click事件', args)
9 })
10
11 emitter.prependOnceListener('click', (...args) => {
12   console.log('监听2到click事件', args)
13 })
14
15 emitter.removeAllListeners('click')
```

第四章：包管理工具详解

4.1 包管理工具npm

1.npm管理的包可以在哪里查看、搜索呢

<https://www.npmjs.com/>

2.npm管理的包存放在哪里呢

- 我们发布自己的包其实是发布到registry上面的;
- 当我们安装一个包时其实是从registry上面下载的包

4.2 项目配置文件

这个配置文件在Node环境下面（无论是前端还是后端）就是package.json

常见的属性

- 必须填写的属性：name、version
 - name是项目的名称
 - version是当前项目的版本号
 - description是描述信息，很多时候是作为项目的基本描述
 - author是作者相关信息（发布时用到）
 - license是开源协议（发布时用到）
- private属性
 - private属性记录当前的项目是否是私有的
 - 当值为true时，npm是不能发布它的，这是防止私有项目或模块发布出去的方式;
- main属性
 - 设置程序的入口
 - 发布一个模块的时候会用到的
- scripts属性
 - scripts属性用于配置一些脚本命令，以键值对的形式存在
 - 配置后我们可以通过 npm run 命令的key来执行这个命令
 - npm start和npm run start的区别是什么
 - 对于常用的 start、test、stop、restart可以省略掉run直接通过 npm start等方式运行
- dependencies属性
 - dependencies属性是指定无论开发环境还是生成环境都需要依赖的包
 - 通常是我们项目实际开发用到的一些库模块
- devDependencies属性
 - 一些包在生成环境是不需要的，比如webpack、babel等;
 - 这个时候我们会通过 npm install webpack --save-dev，将它安装到devDependencies属性中
- engines属性
 - engines属性用于指定Node和NPM的版本号
 - 在安装的过程中，会先检查对应的引擎版本，如果不符合就会报错

- 事实上也可以指定所在的操作系统 "os" : ["darwin", "linux"], 只是很少用到;
- browserslist属性
 - 用于配置打包后的JavaScript浏览器的兼容情况, 参考
 - 否则我们需要手动的添加polyfills来让支持某些语法
 - 也就是说它是为webpack等打包工具服务的一个属性

4.3 版本管理的问题

npm的包通常需要遵从semver版本规范:

semver版本规范是X.Y.Z:

- X主版本号 (major) : 当你做了不兼容的 API 修改 (可能不兼容之前的版本)
- Y次版本号 (minor) : 当你做了向下兼容的功能性新增 (新功能增加, 但是兼容之前的版本)
- Z修订号 (patch) : 当你做了向下兼容的问题修正 (没有新功能, 修复了之前版本的bug)

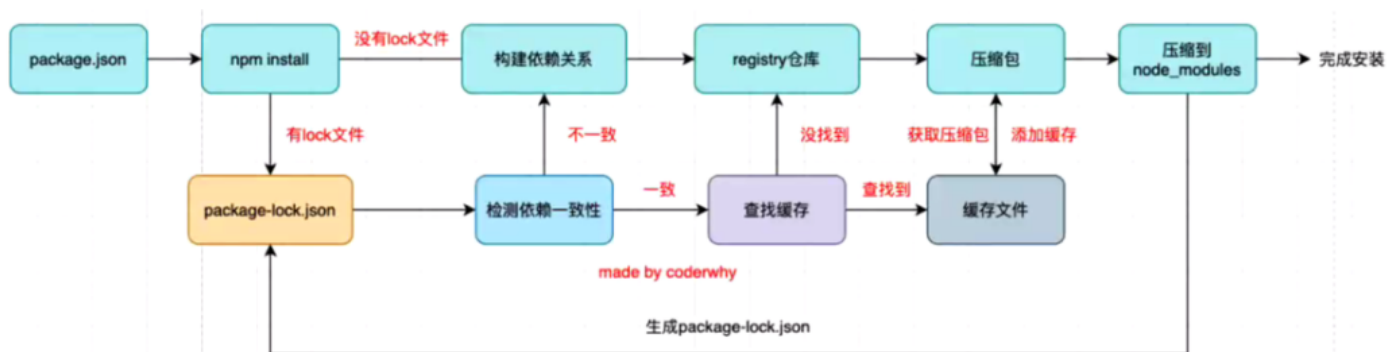
解释一下 ^和~的区别:

- ^x.y.z: 表示x是保持不变的, y和z永远安装最新的版本
- ~x.y.z: 表示x和y保持不变的, z永远安装最新的版本

4.4 npm install 原理

npm install会检测是有package-lock.json文件:

- 没有lock文件
 - 分析依赖关系, 这是因为我们可能包会依赖其他的包, 并且多个包之间会产生相同依赖的情况
 - 从registry仓库中下载压缩包 (如果我们设置了镜像, 那么会从镜像服务器下载压缩包)
 - 获取到压缩包后会对压缩包进行缓存 (从npm5开始有的) ;
 - 将压缩包解压到项目的node_modules文件夹中
- 有lock文件
 - 检测lock中包的版本是否和package.json中一致
 - 不一致, 那么会重新构建依赖关系, 直接会走顶层的流程;
 - 一致的情况下, 会去优先查找缓存
 - 没有找到, 会从registry仓库下载, 直接走顶层流程
 - 查找到, 会获取缓存中的压缩文件, 并且将压缩文件解压到node_modules文件夹中



4.5 package-lock.json

- name: 项目的名称
- version: 项目的版本
- lockfileVersion: lock文件的版本
- requires: 使用requires来跟着模块的依赖关系;
- dependencies: 项目的依赖
 - 当前项目依赖axios, 但是axios依赖follow-redirects;
 - axios中的属性如下
 - version表示实际安装的axios的版本
 - resolved用来记录下载的地址, registry仓库 中的位置;
 - requires记录当前模块的依赖;
 - integrity用来从缓存中获取索引, 再通过索引 去获取压缩包文件;

4.6 npm其他命令

- 卸载某个依赖包: npm uninstall package
- 强制重新build: npm rebuild
- 清除缓存: npm cache clean

4.7 yarn

- yarn是由Facebook、Google、Exponent 和 Tilde 联合推出了一个新的 JS 包管理工具
- yarn 是为了弥补 npm 的一些缺陷而出现的
- 早期的npm存在很多的缺陷, 比如安装依赖速度很慢、版本依赖混乱等等一系列的问题
- 虽然从npm5版本开始, 进行了很多的升级和改进, 但是依然很多人喜欢使用yarn

Npm	Yarn
npm install	yarn install
npm install [package]	yarn add [package]
npm install --save [package]	yarn add [package]
npm install --save-dev [package]	yarn add [package] [--dev/-D]
npm rebuild	yarn install --force
npm uninstall [package]	yarn remove [package]
npm uninstall --save [package]	yarn remove [package]
npm uninstall --save-dev [package]	yarn remove [package]
npm uninstall --save-optional [package]	yarn remove [package]
npm cache clean	yarn cache clean
rm -rf node_modules && npm install	yarn upgrade

4.8 cnpm

- 查看npm镜像: `npm config get registry`
- 我们可以直接设置npm的镜像: `npm config set registry https://registry.npm.taobao.org`
- 但是对于大多数人来说 (比如我), 并不希望将npm镜像修改了:
 - 不太希望随意修改npm原本从官方下来包的渠道
 - 担心某天淘宝的镜像挂了或者不维护了, 又要改来改去;
- 这个时候, 我们可以使用cnpm, 并且将cnpm设置为淘宝的镜像
 - `npm install -g cnpm --registry=https://registry.npm.taobao.org`

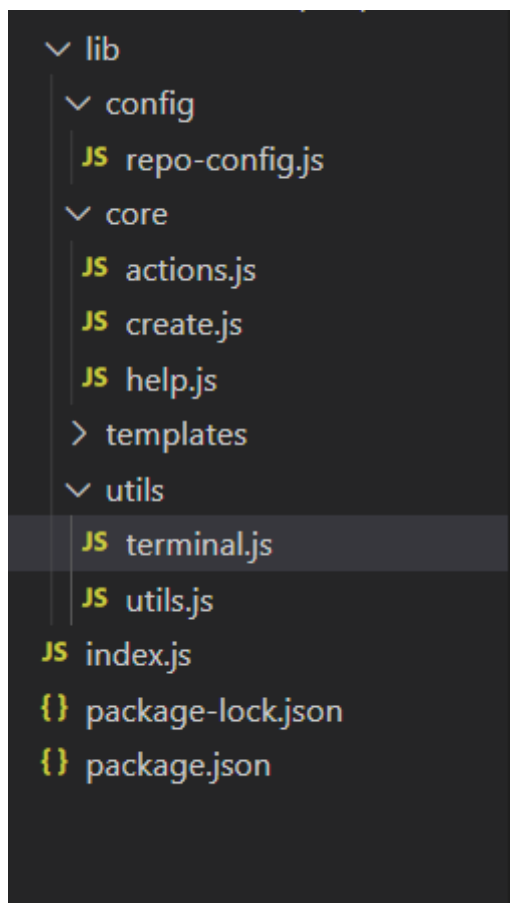
4.9 npx工具

npx是npm5.2之后自带的一个命令。

npx的原理非常简单, 它会到当前目录的`node_modules/.bin`目录下查找对应的命令

```
1 npx webpack --version
```

第五章：开发脚手架工具



5.1 创建项目

1. 项目根目录下编写`index.js`, 在`index.js`中编写一些测试的代码

2. 使用npm init -y 初始化package.json文件

3. 创建yihua命令

a. 在index.js中添加#!/usr/bin/env node

```
1  #!/usr/bin/env node
2
3  console.log('yihua');
```

b. 修改package.json

```
1  "bin":{
2    "yihua":"index.js"
3  },
```

c. 使用npm link

5.2 Commander的使用

npm i commander

index.js

```
1  #!/usr/bin/env node
2
3  const program = require('commander')
4
5  // 查看版本号
6  program.version(require('./package.json').version)
7
8  // 增加自己的option
9  program.option('-y --yihua', 'a yihua cli')
10 program.option('-d --dest <dest>', 'a destination folder ,例如: -d /src/components')
11 program.option('-f --framework <framework>', 'your framework')
12
13 // 修改help
14 program.on('--help',function(){
15   console.log("");
16   console.log("Other:");
17   console.log(" other options~");
18 })
19
20 program.parse(process.argv)
```

将help抽离

新建lib文件夹-> core -> help.js

```
1  const program = require('commander')
2
3  const helpOptions = () => {
4    // 增加自己的option
5    program.option('-y --yihua', 'a yihua cli')
6    program.option('-d --dest <dest>', 'a destination folder ,例如: -d /src/components')
7    program.option('-f --framework <framework>', 'your framework')
8
9    // 修改help
10   program.on('--help', function () {
11     console.log('')
12     console.log('Other:')
13     console.log(' other options~')
14   })
15 }
16
17 module.exports = helpOptions;
```

修改index.js,引入helpOptions 并调用

```
1  #!/usr/bin/env node
2  const program = require('commander')
3
4  const helpOptions = require('./lib/core/help')
5
6  // 查看版本号
7  program.version(require('./package.json').version)
8
9  helpOptions()
10 program.parse(process.argv)
```

5.3 yihua create demo的实现

需要用到的库：

- download-git-repo 用于clone
- open 用于打开浏览器
- inquiry 该项目中没有使用但是可以制作类似于vue-cli的选择项，由我们去选择需要加载的依赖，比如可选vue的

版本。

在create.js中编写注册create的命令的函数

create.js

```
1  const program = require('commander')
2
3  const {createProjectAction} = require('./actions')
4  const createCommands = () => {
5    program
6      .command('create <project> [others...]' )
7      .description('clone repository into a folder')
8      .action(createProjectAction)
9  }
10
11  module.exports = createCommands
```

在index.js中引入调用 createCommands 进行命令的注册

index.js

```
1  const createCommands = require('./lib/core/create')
2  createCommands()
```

createProjectAction的实现

actions.js

```
1  // 转promise
2  const { promisify } = require('util')
3  // 可以从远程仓库下载
4  const download = promisify(require('download-git-repo'))
5  // 打开浏览器
6  const open = require('open');
7
8  // 拿到下载模板的地址
9  const { VueRepo } = require('../config/repo-config')
10
11 // 使用commandSpawn执行终端命令
12 const { commandSpawn } = require('../utils/terminal')
13
14 const createProjectAction = async (project, others) => {
15   console.log('yihua helps you create your project~');
16   // 1.clone项目
```

```

17   await download(VueRepo,project,{ clone: true})
18
19   // 2.执行npm install,需要判断是否是windows系统, windows系统执行的是npm.cmd, 而mac和linux执行的是npm
20   const command = process.platform === 'win32' ? 'npm.cmd' : 'npm'
21   await commandSpawn(command , ['install'], { cwd:`.${project}`})
22   // 4.打开浏览器 , 因为使用npm run serve会阻塞进程, 就不会打开浏览器, 我们也可以在模板通过配置webpack中运行的npm run serve 直接打开。
23   open('http://localhost:8080')
24   // 3.执行npm run serve
25   await commandSpawn(command, ['run' , 'serve'], {cwd:`.${project}`})
26 }
27
28 module.exports = {
29   createProjectAction
30 }

```

config.js

```

1  const VueRepo = 'direct:https://github.com/coderwhy/hy-vue-temp.git';
2
3  module.exports = {
4    VueRepo
5  }

```

在终端里面执行命令：类似于npm install

terminal.js

```

1  /**
2   * 执行终端命令相关的代码
3   */
4
5  // 用于开启子进程
6  const { spawn } = require('child_process')
7
8  const commandSpawn = (...args) => {
9    return new Promise((resolve, reject) => {
10      // 执行命令, 并拿到执行命令对应的子进程
11      const childProcess = spawn(...args)
12      // 将执行的终端命令的输出放到主进程去输出
13      childProcess.stdout.pipe(process.stdout)
14      childProcess.stderr.pipe(process.stderr)

```

```

15 // 监听改命令执行完
16 childProcess.on('close', () => {
17     resolve()
18 })
19 })
20 }
21
22 module.exports = {
23     commandSpawn
24 }

```

5.4 yihua addpage pageName 及 addcpn、addstore

在create.js编写注册这些命令的方法

create.js

```

1  program
2      .command('addcpn <name>')
3      .description('add vue component, eg: yihua addcpn HelloWorld [-d src/components]')
4      .action((name) => {
5          addComponentsAction(name, program.dest || 'src/components')
6      })
7
8  program
9      .command('addpage <page>')
10     .description('add vue page and router config, eg: yihua addpage Home [-d src/pages]')
11     .action((page) => {
12         addPageAndRouteAction(page, program.dest || 'src/pages')
13     })
14
15  program
16     .command('addstore <store>')
17     .description('add vue page and router config, eg: yihua addpage Home [-d src/pages]')
18     .action((store) => {
19         addStoreAction(store, program.dest || 'src/store/modules')
20     })

```

在action.js中具体实现

- 编写对应文件的ejs模板

- 编译模板
- 将编译后的结果写入项目中

action.js

```
1 // 转promise
2 const { promisify } = require('util')
3 const path = require('path')
4 // 可以从远程仓库下载
5 const download = promisify(require('download-git-repo'))
6 // 打开浏览器
7 const open = require('open');
8
9 // 拿到下载模板的地址
10 const { VueRepo } = require('../config/repo-config')
11
12 // 使用commandSpawn执行终端命令
13 const { commandSpawn } = require('../utils/terminal')
14
15 // 编译ejs模板
16 const { compile , writeFile , createDirSync } = require('../utils/utils')
17
18 const createProjectAction = async (project, others) => {
19   console.log('yihua helps you create your project~');
20   // 1.clone项目
21   await download(VueRepo,project,{ clone: true})
22
23   // 2.执行npm install,需要判断是否是windows系统, windows系统执行的是npm.cmd, 而mac和linux执行的是npm
24   const command = process.platform === 'win32' ? 'npm.cmd' : 'npm'
25   await commandSpawn(command , ['install'], { cwd:`./${project}`})
26   // 4.打开浏览器 , 因为使用npm run serve会阻塞进程, 就不会打开浏览器, 我们也可以在模板通过配置webpack中运行的npm run serve 直接打开。
27   open('http://localhost:8080')
28   // 3.执行npm run serve
29   await commandSpawn(command, ['run' , 'serve'], {cwd:`./${project}`})
30 }
31
32 // 添加组件的action
33 const addComponentsAction = async (name , dest) => {
34   // 1.编译ejs模板 result
```

```
35   const result = await compile("vue-component.ejs", {name, lowerName:
    name.toLowerCase()});
36   // console.log(result);
37   // 2.写入文件
38   const targetPath = path.resolve(dest, `${name}.vue`)
39   writeFile(targetPath , result)
40 }
41
42 // 添加页面和路由
43 const addPageAndRouteAction = async (name, dest) => {
44   // 1.编译ejs模板
45   const data = {name, lowerName: name.toLowerCase()};
46   const pageResult = await compile('vue-component.ejs', data);
47   const routeResult = await compile('vue-router.ejs', data);
48
49   // 3.写入文件
50   const targetDest = path.resolve(dest, name.toLowerCase());
51   if (createDirSync(targetDest)) {
52     const targetPagePath = path.resolve(targetDest, `${name}.vue`);
53     const targetRoutePath = path.resolve(targetDest, 'router.js')
54     writeFile(targetPagePath, pageResult);
55     writeFile(targetRoutePath, routeResult);
56   }
57 }
58
59 const addStoreAction = async (name, dest) => {
60   // 1.遍历的过程
61   const storeResult = await compile('vue-store.ejs', {});
62   const typesResult = await compile('vue-types.ejs', {});
63
64   // 2.创建文件
65   const targetDest = path.resolve(dest, name.toLowerCase());
66   if (createDirSync(targetDest)) {
67     const targetPagePath = path.resolve(targetDest, `${name}.js`);
68     const targetRoutePath = path.resolve(targetDest, 'types.js')
69     writeFile(targetPagePath, storeResult);
70     writeFile(targetRoutePath, typesResult);
71   }
72 }
73
```



```
74 module.exports = {
75   createProjectAction,
76   addComponentsAction,
77   addPageAndRouteAction,
78   addStoreAction
79 }
```

utils.js中的编译ejs工具及文件写入和文件夹创建

utils.js

```
1 // 用来编译ejs文件得到模板
2 const ejs = require('ejs')
3 const fs = require('fs')
4 const path = require('path')
5
6 const compile = (templateName, data) => {
7   const templatePosition = `../templates/${templateName}`
8   const templatePath = path.resolve(__dirname, templatePosition)
9
10  return new Promise((resolve, reject) => {
11    ejs.renderFile(templatePath, { data }, {}, (err, result) => {
12      if (err) {
13        console.log(err)
14        reject(err)
15        return
16      }
17
18      resolve(result)
19    })
20  })
21 }
22
23 const createDirSync = (pathName) => {
24   if (fs.existsSync(pathName)) {
25     return true
26   } else {
27     if (createDirSync(path.dirname(pathName))) {
28       fs.mkdirSync(pathName)
29       return true
30     }
31   }
32 }
```

```

31   }
32 }
33
34 const writeFile = (path, content) => {
35   // 判断path是否存在，如果不存在，创建对应的文件夹
36   return fs.promises.writeFile(path, content)
37 }
38
39 module.exports = {
40   compile,
41   writeFile,
42   createDirSync
43 }
44

```

模板:

vue-component.ejs

```

1 <template>
2   <div class="<%= data.lowerName %>">
3     {{msg}}
4   </div>
5 </template>
6
7 <script setup>
8 import { ref } from 'vue'
9 const msg = ref("hello <%= data.lowerName %>")
10 </script>
11
12 <style scoped></style>

```

vue-router.ejs

```

1 // 普通加载路由
2 // import <%= data.name %> from './<%= data.name %>.vue'
3 // 懒加载路由
4 const <%= data.name %> = () => import('./<%= data.name %>.vue')
5 export default {
6   path: './<%= data.lowerName %>',
7   name: '<%= data.name %>',
8   component: <%= data.name %>,

```

```
9   children: [  
10  ]  
11 }  
12
```

vue-store.ejs

```
1 import * as types from './types.js'  
2 export default {  
3   namespaced: true,  
4   state: {  
5   },  
6   mutations: {  
7   },  
8   actions: {  
9   },  
10  getters: {  
11  }  
12 }  
13
```

vue-types.ejs

```
1 export {  
2  
3 }
```

5.5 发布到npm

1. 注册npm账号:<https://www.npmjs.com/>
2. 在命令行登录:npm login
3. 修改package.json

```
"keywords": ["why", "vue", "coderwhy"],  
"author": "coderwhy",  
"license": "MIT",  
"homepage": "https://github.com/coderwhy/coderwhy",  
"repository": {  
  "type": "git",  
  "url": "https://github.com/coderwhy/coderwhy"  
},
```

4. 发布到npm registry上: npm publish

5. 更新仓库：修改版本号、重新发布
6. 删除发布的包：npm unpublish
7. 让发布的包过期：npm deprecate

第六章：Buffer的使用

6.1 Buffer和二进制

对于前端开发来说，通常很少会和二进制打交道，但是对于服务器端为了做很多的功能，我们必须直接去操作其二进制的数据；所以Node为了可以方便开发者完成更多功能，提供给了我们一个类Buffer，并且它是全局的。

1.Buffer中存储的是二进制数据，那么到底是如何存储呢？

- 我们可以将Buffer看成是一个存储二进制的数组
- 这个数组中的每一项，可以保存8位二进制：00000000

2.为什么是8位呢

- 在计算机中，很少的情况我们会直接操作一位二进制，因为一位二进制存储的数据是非常有限的；
- 所以通常会将8位合在一起作为一个单元，这个单元称之为一个字节（byte）；
- 也就是说 1byte = 8bit, 1kb=1024byte, 1M=1024kb;
- 比如RGB的值分别都是255，所以本质上在计算机中都是用一个字节存储的；

6.2 Buffer和字符串

- Buffer相当于是一个字节的数组，数组中的每一项对于一个字节的大小：
- 如果我们希望将一个字符串放入到Buffer中，是怎么样的过程呢

```
1  const message = 'Hello world'
2
3  // 1.创建方式一
4  // const buffer = new Buffer(message)
5  // console.log(buffer)
6
7  // 2.创建方式二
8  const buffer = Buffer.from(message)
9  console.log(buffer);
```

将字符串转成对应的ascii码的16进制，Buffer中存储的是其对应的16进制。

中文如何解析

- 默认编码：utf-8
- 可以改成utf16

```
1  const message = '你好啊'
```

```
2
3 // const buffer = Buffer.from(message, 'utf8')
4 const buffer = Buffer.from(message, 'utf16le')
5 console.log(buffer)
6 console.log(buffer.toString('utf16le'))
```

编码解码需要保持同一种格式

解码可以使用toString()

6.3 Buffer的其它创建方式

- Class: Buffer
 - Static method: Buffer.alloc(size[, fill[, encoding]])
 - Static method: Buffer.allocUnsafe(size)
 - Static method: Buffer.allocUnsafeSlow(size)
 - Static method: Buffer.byteLength(string[, encoding])
 - Static method: Buffer.compare(buf1, buf2)
 - Static method: Buffer.concat(list[, totalLength])
 - Static method: Buffer.from(array)
 - Static method: Buffer.from(arrayBuffer[, byteOffset[, length]])
 - Static method: Buffer.from(buffer)
 - Static method: Buffer.from(object[, offsetOrEncoding[, length]])
 - Static method: Buffer.from(string[, encoding])
 - Static method: Buffer.isBuffer(obj)
 - Static method: Buffer.isEncoding(encoding)

Buffer.alloc

```
1 const buffer = Buffer.alloc(8)
2 console.log(buffer)
3 buffer[0] = 0x88
4 console.log(buffer)
```

6.4 Buffer和文件读取

sharp:图片的处理，包括裁剪。 <https://blog.lcddjm.com/sharp-documents-cn/>

npm i sharp

```
1  const fs = require('fs')
2  const sharp = require('sharp')
3
4  // 读文本文件
5  // fs.readFile('./foo.txt', (err, data) => {
6  //   console.log(data)
7  //   console.log(data.toString())
8  // })
9
10 // 读图片
11 // fs.readFile('./bar.png', (err, data) => {
12 //   console.log(data)
13 //   // 复制文件
14 //   fs.writeFile('./copybar.png', data, (err, data) => {
15 //     console.log(err)
16 //   })
17 // })
18
19 // sharp的使用
20 sharp('./bar.png')
21   .resize(300, 300)
22   .toBuffer()
23   .then(data => {
24     fs.writeFile('./bax.png', data, err => console.log(err))
25   })
```

6.5 Buffer的创建过程

事实上我们创建Buffer时，并不会频繁的向操作系统申请内存，它会默认先申请一个8 * 1024个字节大小的内存，也就是8kb。

```
Buffer.poolSize = 8 * 1024;
let poolSize, poolOffset, allocPool;

const encodingsMap = ObjectCreate(null);
for (let i = 0; i < encodings.length; ++i)
  encodingsMap[encodings[i]] = i;

function createPool() {
  poolSize = Buffer.poolSize;
  allocPool = createUnsafeBuffer(poolSize).buffer;
  markAsUntransferable(allocPool);
  poolOffset = 0;
}
createPool();
```

第七章：事件循环和异步IO

7.1 什么是事件循环？

- 事件循环是什么
 - 事实上我把事件循环理解成我们编写的JavaScript和浏览器或者Node之间的一个桥梁。
- 浏览器的事件循环是一个我们编写的JavaScript代码和浏览器API调用(setTimeout/AJAX/监听事件等)的一个桥梁, 桥梁之间他们通过回调函数进行沟通。
- Node的事件循环是一个我们编写的JavaScript代码和系统调用 (file system、network等) 之间的一个桥梁, 桥梁之间他们通过回调函数进行沟通的。

7.2 进程和线程

线程和进程是操作系统中的两个概念：

- 进程 (process)：计算机已经运行的程序；
- 线程 (thread)：操作系统能够运行运算调度的最小单位；

多进程多线程开发。操作系统是如何做到同时让多个进程（边听歌、边写代码、边查阅资料）同时工作呢？

- 这是因为CPU的运算速度非常快，它可以快速的在多个进程之间迅速的切换；
- 当我们的进程中的线程获取获取到时间片时，就可以快速执行我们编写的代码；
- 对于用于来说是感受不到这种快速的切换的；

7.3 浏览器和JavaScript

我们经常会说JavaScript是单线程的，但是JavaScript的线程应该有自己的容器进程：浏览器或者Node。

浏览器是一个进程吗，它里面只有一个线程吗？

- 目前多数的浏览器其实都是多进程的，当我们打开一个tab页面时就会开启一个新的进程，这是为了防止一个页面卡死而造成所有页面无法响应，整个浏览器需要强制退出；
- 每个进程中又有很多的线程，其中包括执行JavaScript代码的线程

但是JavaScript的代码执行是在一个单独的线程中执行的：

- 这就意味着JavaScript的代码，在同一个时刻只能做一件事
- 如果这件事是非常耗时的，就意味着当前的线程就会被阻塞；

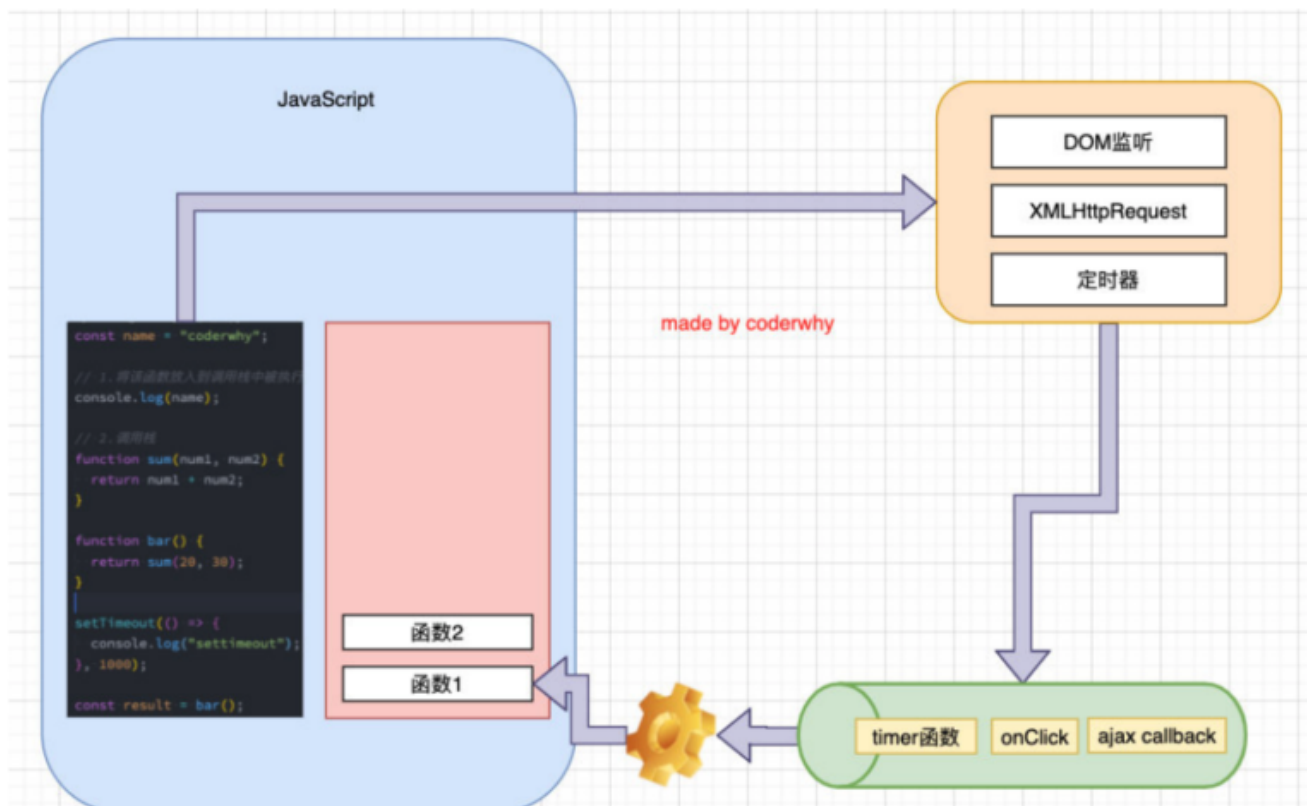
7.4 浏览器的事件循环

如果在执行JavaScript代码的过程中，有异步操作呢？

- 中间我们插入了一个setTimeout的函数调用
- 这个函数被放到入调用栈中，执行会立即结束，并不会阻塞后续代码的执行；

那么，传入的一个函数（比如我们称之为timer函数），会在什么时候被执行呢？

- 事实上，setTimeout是调用了web api，在合适的时机，会将timer函数加入到一个事件队列中；
- 事件队列中的函数，会被放入到调用栈中，在调用栈中被执行；



7.5 宏任务和微任务

但是事件循环中并非只维护着一个队列，事实上是有两个队列

- 宏任务队列 (macrotask queue) : ajax、setTimeout、setInterval、DOM监听、UI Rendering等
- 微任务队列 (microtask queue) : Promise的then回调、Mutation Observer API、queueMicrotask()等

那么事件循环对于两个队列的优先级是怎么样子的呢？

- main script中的代码优先执行（编写的顶层script代码）；
- 在执行任何一个宏任务之前（不是队列，是一个宏任务），都会先查看微任务队列中是否有任务需要执行
 - 也就是宏任务执行之前，必须保证微任务队列是空的；
 - 如果不为空，那么久优先执行微任务队列中的任务（回调）；

浏览器面试题一：

```
1  setTimeout(function () {
2    console.log("set1");
3    new Promise(function (resolve) {
4      resolve();
5    }).then(function () {
6      new Promise(function (resolve) {
7        resolve();
8      }).then(function () {
9        console.log("then4");
10     });
11     console.log("then2");
12   });
13 });
14
15 new Promise(function (resolve) {
16   console.log("pr1");
17   resolve();
18 }).then(function () {
19   console.log("then1");
20 });
21
22 setTimeout(function () {
23   console.log("set2");
24 });
25
26 console.log(2);
27
28 queueMicrotask(() => {
29   console.log("queueMicrotask1")
```

```
30 });
31
32 new Promise(function (resolve) {
33     resolve();
34 }).then(function () {
35     console.log("then3");
36 });
37
38 // pr1
39 // 2
40 // then1
41 // queueMicrotask1
42 // then3
43 // set1
44 // then2
45 // then4
46 // set2
```

浏览器面试题二：

```
1  async function async1 () {
2      console.log('async1 start')
3      await async2();
4      console.log('async1 end')
5  }
6
7  async function async2 () {
8      console.log('async2')
9  }
10
11 console.log('script start')
12
13 setTimeout(function () {
14     console.log('setTimeout')
15 }, 0)
16
17 async1();
18
19 new Promise (function (resolve) {
20     console.log('promise1')
```

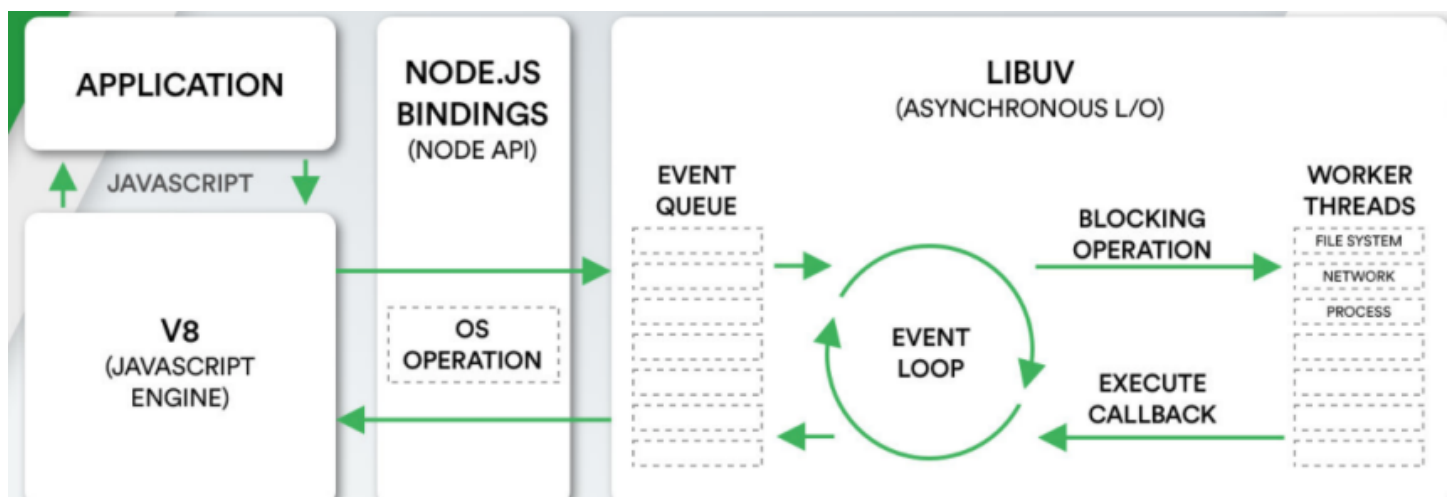
```

21   resolve();
22 }).then (function () {
23   console.log('promise2')
24 })
25
26 console.log('script end')
27
28
29 // script start
30 // async1 start
31 // async2
32 // promise1
33 // script end
34 // aysnc1 end
35 // promise2
36 // setToueout

```

7.5 Node的架构分析

- 浏览器中的EventLoop是根据HTML5定义规范来实现的，不同的浏览器可能会有不同的实现，而Node中是由libuv实现的。
- libuv中主要维护了一个EventLoop和worker threads（线程池）；EventLoop负责调用系统的一些其他操作：文件的IO、Network、child-processes等
- libuv是一个多平台的专注于异步IO的库，它最初是为Node开发的，但是现在也被使用到Luvit、Julia、pyuv等其他地方；



7.6 阻塞IO和非阻塞IO

操作系统为我们提供了阻塞式调用和非阻塞式调用：

- 阻塞式调用：调用结果返回之前，当前线程处于阻塞态（阻塞态CPU是不会分配时间片的），调用线程只有在

得到调用结果之后才会继续执行。

- 非阻塞式调用：调用执行之后，当前线程不会停止执行，只需要过一段时间来检查一下有没有结果返回即可。

所以我们开发中的很多耗时操作，都可以基于这样的 非阻塞式调用：

- 比如网络请求本身使用了Socket通信，而Socket本身提供了select模型，可以进行非阻塞方式的工作
- 比如文件读写的IO操作，我们可以使用操作系统提供的基于事件的回调机制；

非阻塞IO的问题

但是非阻塞IO也会存在一定的问题：我们并没有获取到需要读取（我们以读取为例）的结果。

- 那么就意味着为了可以知道是否读取到了完整的数据，我们需要频繁的去确定读取到的数据是否是完整的；
- 这个过程我们称之为轮询操作

那么这个轮训的工作由谁来完成呢？

- 如果我们的主线程频繁的去进行轮训的工作，那么必然会大大降低性能；
- 并且开发中我们可能不只是一个文件的读写，可能是多个文件；
- 而且可能是多个功能：网络的IO、数据库的IO、子进程调用；

libuv提供了一个线程池

- 线程池会负责所有的操作，并且会通过轮训等方式等待结果
- 当获取到结果时，就可以将对应的回调放到事件循环（某一个事件队列）中；
- 事件循环就可以负责接管后续的回调工作，告知JavaScript应用程序执行对应的回调函数；



理解：比如我们写一个文件读写的函数，在该函数中传入的回调函数会被传入事件队列。线程池中的线程会去执行文件的读写，当线程完成了文件的读写，得到了结果之后。就会把该回调函数放入事件循环，回调函数在JavaScript任务队列中执行。

阻塞和非阻塞，同步和异步的区别：

阻塞和非阻塞是对于被调用者来说的

- 在我们这里就是系统调用，操作系统为我们提供了阻塞调用和非阻塞调用；

同步和异步是对于调用者来说的

- 在我们这里就是自己的程序
- 如果我们在发起调用之后，不会进行其他任何的操作，只是等待结果，这个过程就称之为同步调用；
- 如果我们再发起调用之后，并不会等待结果，继续完成其他的工作，等到有回调时再去执行，这个过程就是 异步调用；

Libuv采用的就是非阻塞异步IO的调用方式；

7.7 Node事件循环的阶段

无论是我们的文件IO、数据库、网络IO、定时器、子进程，在完成对应的操作后，都会将对应的结果和回调 函数放到事件循环（任务队列）中；事件循环会不断的从任务队列中取出对应的事件（回调函数）来执行。

一次完整的事件循环Tick分成很多个阶段：

- 定时器（Timers）：本阶段执行已经被 `setTimeout()` 和 `setInterval()` 的调度回调函数
- 待定回调（Pending Callback）：对某些系统操作（如TCP错误类型）执行回调，比如TCP连接时接收到 `ECONNREFUSED`。
- idle, prepare：仅系统内部使用。
- 轮询（Poll）：检索新的 I/O 事件；执行与 I/O 相关的回调
- 检测：`setImmediate()` 回调函数在这里执行
- 关闭的回调函数：一些关闭的回调函数，如：`socket.on('close', ...)`。



7.8 Node的微任务和宏任务

- 宏任务（macrotask）：`setTimeout`、`setInterval`、IO事件、`setImmediate`、close事件；
- 微任务（microtask）：`Promise`的then回调、`process.nextTick`、`queueMicrotask`；

Node中的事件循环不只是 微任务队列和 宏任务队列。任务队列及其执行顺序如下

- 微任务队列：
 - next tick queue: `process.nextTick`;
 - other queue: `Promise`的then回调、`queueMicrotask`;

- 宏任务队列
 - timer queue: setTimeout、setInterval;
 - poll queue: IO事件;
 - check queue: setImmediate;
 - close queue: close事件;

面试题一:

```
1  async function async1() {
2    console.log('async1 start')
3    await async2()
4    console.log('async1 end')
5  }
6
7  async function async2() {
8    console.log('async2')
9  }
10
11 console.log('script start')
12
13 setTimeout(function () {
14   console.log('setTimeout0')
15 }, 0)
16
17 setTimeout(function () {
18   console.log('setTimeout2')
19 }, 300)
20
21 setImmediate(() => console.log('setImmediate'));
22
23 process.nextTick(() => console.log('nextTick1'));
24
25 async1();
26
27 process.nextTick(() => console.log('nextTick2'));
28
29 new Promise(function (resolve) {
30   console.log('promise1')
31   resolve();
32   console.log('promise2')
```

```

33 }).then(function () {
34     console.log('promise3')
35 })
36
37 console.log('script end')
38
39
40 // script start
41 // async1 start
42 // async2
43 // promise1
44 // promise2
45 // script end
46 // nextTick1
47 // nextTick2
48 // async1 end
49 // promise3
50 // setTimeout0
51 // setImmediate
52 // setTimeout2

```

面试题二：

```

1  setTimeout(() => {
2      console.log("setTimeout");
3  }, 0);
4
5  setImmediate(() => {
6      console.log("setImmediate");
7  });
8
9  // 问题: setTimeout setImmediate

```

为什么面试题二有时候 setImmediate会早于 setTimeout?

- 在Node源码的deps/uv/src/timer.c中141行，有一个 uv__next_timeout 的函数；
- 这个函数决定了，poll阶段要不要阻塞在这里；
- 阻塞在这里的目的是当有异步IO被处理时，尽可能快的让代码被执行

情况一：如果事件循环开启的时间(ms)是小于 setTimeout函数的执行时间的；

- 也就意味着先开启了event-loop，但是这个时候执行到timer阶段，并没有 定时器的回调被放到入 timer queue 中；

- 所以没有被执行，后续开启定时器和检测到有setImmediate时，就会跳过 poll阶段，向后继续执行；
- 这个时候是先检测 setImmediate，第二次的tick中执行了timer中的 setTimeout；

情况二：如果事件循环开启的时间(ms)是大于 setTimeout函数的执行时间的；

- 这就意味着在第一次 tick中，已经准备好了timer queue；
- 所以会直接按照顺序执行即可；

```
int uv__next_timeout(const uv_loop_t* loop) {
    const struct heap_node* heap_node;
    const uv_timer_t* handle;
    uint64_t diff;

    // 计算距离当前时间节点最小的计时器
    heap_node = heap_min(timer_heap(loop));
    // 如果为空，那么返回-1，表示为阻塞状态
    if (heap_node == NULL)
        return -1; /* block indefinitely */

    // 如果计时器的时间小于当前loop的开始时间，那么返回0
    // 继续执行后续阶段，并且开启下一次tick
    handle = container_of(heap_node, uv_timer_t, heap_node);
    if (handle->timeout <= loop->time)
        return 0;

    // 如果不大于loop的开始时间，那么会返回时间差
    diff = handle->timeout - loop->time;
    if (diff > INT_MAX)
        diff = INT_MAX;

    return (int) diff;
}
```

第八章：Stream

8.1 认识Stream

- 是连续字节的一种表现形式和抽象概念；
- 流应该是可读的，也是可写的；

在之前学习文件的读写时，我们可以直接通过 readFile或者 writeFile方式读写文件，为什么还需要流呢？

- 直接读写文件的方式，虽然简单，但是无法控制一些细节的操作；
- 比如从什么位置开始读、读到什么位置、一次性读取多少个字节；

- 读到某个位置后，暂停读取，某个时刻恢复读取等等；
- 或者这个文件非常大，比如一个视频文件，一次性全部读取并不合适；

文件读写的Stream

事实上Node中很多对象是基于流实现的：

- http模块的Request和Response对象
- process.stdout对象；

官方：另外所有的流都是EventEmitter的实例

Node.js中有四种基本流类型：

- Writable：可以向其写入数据的流
- Readable：可以从中读取数据的流
- Duplex：同时为Readable和的流Writable
- Transform：Duplex可以在写入和读取数据时修改或转换数据的流

8.2 Readable

- start：文件读取开始的位置；
- end：文件读取结束的位置；
- highWaterMark：一次性读取字节的长度，默认是64kb；

```
1  const fs = require('fs')
2
3  // 创建reader，从第三个字节开始到第六个字节结束，每次读两次字节
4  const reader = fs.createReadStream('./foo.txt', {
5    start: 3,
6    end: 6,
7    highWaterMark: 2
8  })
9
10 // 文件读完之后的回调
11 reader.on('data', (data) => {
12   console.log(data.toString())
13   // 暂停reader
14   reader.pause()
15   setTimeout(() => {
16     // 恢复reader
17     reader.resume()
18   }, 1000)
19 })
20
21 reader.on('open', () => {
```

```
22 console.log('文件被打开')
23 })
24
25 reader.on('close', () => {
26   console.log('文件被关闭')
27 })
```

8.3 Writable

- flags: 默认是w, 如果我们希望是追加写入, 可以使用 a或者 a+;
- start: 写入的位置;

```
1  const fs = require('fs')
2
3  const writer = fs.createWriteStream('./bar.txt', {
4    flags: 'a',
5    start: 4
6  })
7
8  writer.write('你好啊', (err) => {
9    if (err) {
10      console.log(err);
11      return ;
12    }
13    console.log('写入成功');
14  })
15 // 关闭, 很少用
16 // writer.close()
17 // end做了两件事, 先writer, 然后关闭流。
18 writer.end()
19
20 writer.on('close', () => {
21   console.log('文件被关闭');
22 })
```

8.3 pipe方法

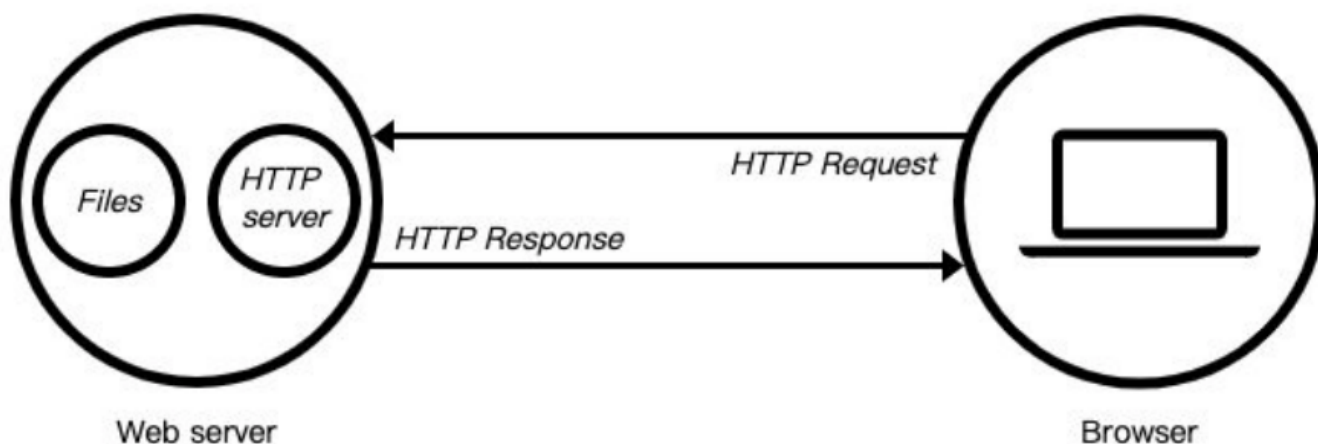
```
1  const fs = require('fs')
2
```

```
3 const reader = fs.createReadStream('./foo.txt')
4 const writer = fs.createWriteStream('./foz.txt')
5
6 reader.pipe(writer)
7 writer.close()
```

第九章：HTTP模块

9.1 Web服务器

当应用程序（客户端）需要某一个资源时，可以向一台服务器，通过Http请求获取到这个资源；提供资源的这个服务器，就是一个Web服务器；



目前有很多开源的Web服务器：Nginx、Apache（静态）、Apache Tomcat（静态、动态）、Node.js

9.2 Web服务器初体验

```
1 const http = require('http')
2
3 // 创建一个web服务器
4 const server = http.createServer((req, res) => {
5   res.end('hello http')
6 })
7
8 server.listen(3000, '0.0.0.0', () => {
9   console.log('server runing at localhost:3000')
10 })
```

9.3 创建服务器

```
1  const http = require('http')
2
3  const server1 = http.createServer((req, res) => {
4    res.end('server1')
5  })
6
7  server1.listen(8000, () => {
8    console.log('server1 启动')
9  })
10
11 const server2 = new http.Server((req, res) => {
12   res.end('server2')
13 })
14
15 server2.listen(8001, () => {
16   console.log('server2 启动')
17 })
```

9.4 监听主机和端口号

Server通过listen方法来开启服务器，并且在某一个主机和端口上监听网络请求：

- 也就是当我们通过 ip:port的方式发送到我们监听的Web服务器上时；
- 我们就可以对其进行相关的处理；

listen函数有三个参数：

- 端口port: 可以不传, 系统会默认分配端, 后续项目中我们会写入到环境变量中
- 主机host: 通常可以传入localhost、ip地址127.0.0.1、或者ip地址0.0.0.0，默认是0.0.0.0
 - localhost: 本质上是一个域名，通常情况下会被解析成127.0.0.1；
 - 127.0.0.1: 回环地址（Loop Back Address），表达的意思其实是我们主机自己发出去的包，直接被自己接收；
 - 正常的数据库包经常 应用层 - 传输层 - 网络层 - 数据链路层 - 物理层；
 - 而回环地址，是在网络层直接就被获取到了，是不会经常数据链路层和物理层的；
 - 比如我们监听 127.0.0.1时，在同一个网段下的主机中，通过ip地址是不能访问的；
 - 0.0.0.0
 - 监听IPV4上所有的地址，再根据端口找到不同的应用程序
 - 比如我们监听 0.0.0.0时，在同一个网段下的主机中，通过ip地址是可以访问的；

- 回调函数：服务器启动成功时的回调函数；

9.5 request对象

在向服务器发送请求时，我们会携带很多信息，比如：

- 本次请求的URL，服务器需要根据不同的URL进行不同的处理；
- 本次请求的请求方式，比如GET、POST请求传入的参数和处理的方式是不同的；
- 本次请求的headers中也会携带一些信息，比如客户端信息、接受数据的格式、支持的编码格式等；

这些信息，Node会帮助我们封装到一个request的对象中，我们可以直接来处理这个request对象

```
1  const http = require('http')
2  const url = require('url')
3  const qs = require('qs')
4  const server = http.createServer((req, res) => {
5    console.log(req)
6    console.log(req.url)
7    const { pathname, query } = url.parse(req.url)
8    console.log(pathname, query)
9    const { username, password } = qs.parse(query)
10   console.log(req.method)
11   console.log(req.headers)
12   res.end('Hello Server')
13 })
14
15 server.listen(8080, () => {
16   console.log('localhost:8080')
17 })
```

9.6 创建用户接口

```
1  const http = require('http')
2  const url = require('url')
3  const qs = require('qs')
4  const server = http.createServer((req, res) => {
5    const { pathname } = url.parse(req.url)
6    if(pathname === '/login'){
7      if(req.method === 'POST'){
8        // 拿到body中的数据
9        req.setEncoding('utf-8')
10        req.on('data', data =>{
```

```
11     console.log(JSON.parse(data));
12   })
13 }
14 }
15 res.end('hello HTTP')
16 })
17
18 server.listen(8080, () => {
19   console.log('localhost:8080')
20 })
21
```

9.7 headers属性

content-type是这次请求携带的数据的类型：

- application/json表示是一个json类型；
- text/plain表示是文本类型；
- application/xml表示是xml类型；
- multipart/form-data表示是上传文件；

content-length：文件的大小和长度

keep-alive：

- http是基于TCP协议的，但是通常在进行一次请求和响应结束后会立刻中断
- 在http1.0中，如果想要继续保持连接：
 - 浏览器需要在请求头中添加 connection: keep-alive；
 - 服务器需要在响应头中添加 connection:keep-alive；
 - 当客户端再次放请求时，就会使用同一个连接，直接一方中断连接；
- 在http1.1中，所有连接默认是 connection: keep-alive的；
 - 不同的Web服务器会有不同的保持 keep-alive的时间；
 - Node中默认是5s中；

accept-encoding：告知服务器，客户端支持的文件压缩格式，比如js文件可以使用gzip编码，对应.gz文件；

accept：告知服务器，客户端可接受文件的格式类型；

user-agent：客户端相关的信息；

9.8 返回响应结果

如果我们希望给客户端响应的结果数据，可以通过两种方式：

- Write方法：这种方式是直接写出数据，但是并没有关闭流；
- end方法：这种方式是写出最后的数据，并且写出后会关闭流；

如果我们没有调用 `end`和`close`，客户端将会一直等待结果

9.9 返回状态码

状态代码	状态描述	说明
200	OK	客户端请求成功。
400	Bad Request	由于客户端请求有语法错误，不能被服务器所理解。
401	Unauthorized	请求未经授权。这个状态代码必须和WWW-Authenticate报头域一起使用。
403	Forbidden	服务器收到请求，但是拒绝提供服务。服务器通常会在响应正文中给出不提供服务的原因。
404	Not Found	请求的资源不存在，例如，输入了错误的URL。
500	Internal Server Error	服务器发生不可预期的错误，导致无法完成客户端的请求。
503	Service Unavailable	服务器当前不能够处理客户端的请求，在一段时间之后，服务器可能会恢复正常。

```
1 res.statusCode = 200
2 res.writeHead(200)
```

9.10 响应头文件

返回头部信息，主要有两种方式：

- `res.setHeader`：一次写入一个头部信息
- `res.writeHead`：同时写入header和status；

```
1 res.setHeader('Content-type', 'application/json;charset=utf8')
2 res.writeHead(200, {
3   'Content-type': 'application/json;charset=utf8'
4 })
```

9.11 文件上传

```
1 const http = require('http');
2 const fs = require('fs');
3 const qs = require('querystring');
4
5 const server = http.createServer((req, res) => {
6   if (req.url === '/upload') {
7     if (req.method === 'POST') {
```

```
8     req.setEncoding('binary');
9
10    let body = '';
11    const totalBoundary = req.headers['content-type'].split(';')[1];
12    const boundary = totalBoundary.split('=')[1];
13
14    req.on('data', (data) => {
15        body += data;
16    });
17
18    req.on('end', () => {
19        console.log(body);
20        // 处理body
21        // 1.获取image/png的位置
22        const payload = qs.parse(body, "\r\n", ": ");
23        const type = payload["Content-Type"];
24
25        // 2.开始在image/png的位置进行截取
26        const typeIndex = body.indexOf(type);
27        const typeLength = type.length;
28        let imageData = body.substring(typeIndex + typeLength);
29
30        // 3.将中间的两个空格去掉
31        imageData = imageData.replace(/\s\s*/, '');
32
33        // 4.将最后的boundary去掉
34        imageData = imageData.substring(0, imageData.indexOf(`--${boundary}--`));
35
36        fs.writeFile('./foo.png', imageData, 'binary', (err) => {
37            res.end("文件上传成功~");
38        })
39    })
40 }
41 }
42 });
43
44 server.listen(8000, () => {
45     console.log("文件上传服务器开启成功~");
46 })
```


第十章：Express

10.安装

- 方式一：通过express提供的脚手架，直接创建一个应用的骨架；
 - 安装express-generator：npm install -g express-generator
 - 创建项目：express express-demo
 - npm install
 - 启动项目：node bin/www
- 方式二：从零搭建自己的express应用结构；

10.2 认识中间件

Express是一个路由和中间件的Web框架，它本身的功能非常少。Express应用程序本质上是一系列中间件函数的调用；

中间件是什么呢

- 中间件的本质是传递给express的一个回调函数
- 这个回调函数接受三个参数
 - 请求对象（request对象）
 - 响应对象（response对象）；
 - next函数（在express中定义的用于执行下一个中间件的函数）

中间件中可以执行哪些任务呢

- 执行任何代码
- 更改请求（request）和响应（response）对象
- 结束请求-响应周期（返回数据）
- 调用栈中的下一个中间件

如果当前中间件功能没有结束请求-响应周期，则必须调用next()将控制权传递给下一个中间件功能，否则，请求 将被挂起。

应用中间件 – 自己编写

- express主要提供了两种方式：app/router.use和app/router.methods

普通中间件

```
1  const express = require('express')
2
3  const app = express()
4
5  // 编写普通的中间件
```

```
6 app.use((req, res, next) => {
7   console.log('注册了第一个普通的中间件')
8   // res.end('end current request...')
9   next()
10 })
11
12 app.use((req, res, next) => {
13   console.log('注册了第二个普通的中间件')
14   res.end('end current request...')
15 })
16
17 app.listen(12000, () => {
18   console.log('express初体验')
19 })
20
```

路径中间件

```
1 const express = require('express')
2
3 const app = express()
4
5 // 编写普通的中间件
6 app.use((req, res, next) => {
7   console.log('注册了第一个普通的中间件')
8   // res.end('end current request...')
9   next()
10 })
11
12 // 路径匹配中间件
13 app.use('/home', (req, res, next) => {
14   console.log('home')
15   next()
16 })
17
18 // 路径匹配中间件
19 app.use('/home', (req, res, next) => {
20   res.end('home')
21 })
22
```

```
23
24 app.listen(12000, () => {
25   console.log('express初体验')
26 })
27
```

路径和方法匹配的中间件

```
1  const express = require('express')
2  const app = express()
3
4  // 路径和方法匹配的中间件
5  app.get('/home', (req, res, next) => {
6    console.log('匹配home的get')
7    res.end('home get')
8  })
9
10 app.listen(12000, () => {
11   console.log('express初体验')
12 })
```

连续注册多个中间件

```
1  const express = require('express')
2  const app = express()
3
4  app.use((req, res, next) => {
5    console.log('common middleware 01')
6    next()
7  })
8
9  app.get('/home', (req, res, next) => {
10   console.log('GET HOME 01')
11   next()
12 }, (req, res, next) => {
13   console.log('GET HOME 02')
14   next()
15 }, (req, res, next) => {
16   console.log('GET HOME 03')
17   res.end('Home page')
18 })
```

```
19
20 app.listen(12000, () => {
21   console.log('express初体验')
22 })
```

应用中间件 – body解析

```
1  const express = require('express')
2  const app = express()
3
4  app.use((req, res, next) => {
5    if (req.headers['content-type'] === 'application/json') {
6      req.on('data', data => {
7        const info = JSON.parse(data.toString())
8        req.body = info
9      })
10
11      req.on('end', () => {
12        next()
13      })
14    } else {
15      next()
16    }
17  })
18
19 app.post('/login', (req, res, next) => {
20   console.log(req.body)
21   res.end('login')
22 })
23
24 app.post('/products', (req, res, next) => {
25   console.log(req.body)
26   res.end('products')
27 })
28
29 app.listen(12000, () => {
30   console.log('express初体验')
31 })
```

应用中间件 – express提供

```

1 // 使用express提供给我们的body解析
2 // body-parser: express3.x 内置express框架
3 // body-parser: express4.x 被分离出去
4 // body-parser类似功能: express4.16.x 内置成函数
5 app.use(express.json());
6 // extended
7 // true: 那么对urlencoded进行解析时, 它使用的是第三方库: qs
8 // false: 那么对urlencoded进行解析时, 它使用的是Node内置模块: querystring
9 app.use(express.urlencoded({extended: true}));

```

应用中间件 – 第三方中间件

```

1 const express = require('express')
2 const multer = require('multer')
3
4 const app = express()
5
6 app.use(express.json())
7 app.use(express.urlencoded({extended:true}))
8
9 const upload = multer()
10
11 app.post('/login',upload.any(),(req,res,next)=>{
12   console.log(req.body);
13   res.end('用户登录成功')
14 })
15
16 app.listen(8000,()=>{
17   console.log('localhost:8000');
18 })

```

上传文件配合multer中间件解析formData格式。并修改上传的文件名。注意不要使用upload.any()作为全局中间件

```

1 const express = require('express')
2 const multer = require('multer')
3
4 const app = express()
5
6 app.use(express.json())

```

```
7 app.use(express.urlencoded({extended: true}))
8
9 const storage = multer.diskStorage({
10   destination: (req, res, cb) => {
11     cb(null, './uploads/')
12   },
13   filename: (req, file, cb) => {
14     cb(null, file.originalname)
15   }
16 })
17
18 const upload = multer({
19   storage})
20
21 app.post('/upload', upload.single('file'), (req, res, next) => {
22   console.log(req.file)
23   res.end('文件上传')
24 })
25
26 // app.post('/upload', upload.array('file'), (req, res, next) => {
27 //   console.log(req.files)
28 //   res.end('文件上传')
29 // })
30
31 app.listen(8000, () => {
32   console.log('localhost:8000')
33 })
34
```

morgan (日志保存)

```
1 const express = require('express')
2 const morgan = require('morgan')
3 const fs = require('fs')
4
5 const app = express()
6
7 const writerStream = fs.createWriteStream('./logs/access.log', {
8   flags: 'a+'
9 })
```

```

10
11 app.use(morgan('combined', {stream: writerStream}))
12
13 app.get('/home', (req, res, next) => {
14   res.end('home')
15 })
16
17 app.listen(8000, () => {
18   console.log('express初体验')
19 })
20

```

10.3 客户端发送请求的方式

客户端传递到服务器参数的方法常见的是5种

- 通过get请求中的URL的params
- 通过get请求中的URL的query;
- 通过post请求中的body的json格式;
- 通过post请求中的body的x-www-form-urlencoded格式;
- 通过post请求中的form-data格式

```

1  const express = require('express')
2
3  const app = express()
4
5  app.get('/products/:id/:name', (req, res, next) => {
6    console.log(req.params);
7    res.end('商铺详情')
8  })
9
10 app.get('/login', (req, res, next) => {
11   console.log(req.query);
12   res.end('登录成功')
13 })
14
15 app.listen(12000, () => {
16   console.log('express初体验')
17 })
18

```

10.4 响应数据

- end方法：类似于http中的response.end方法，用法是一致的
- json方法：json方法中可以传入很多的类型：object、array、string、boolean、number、null等，它们会被转换成json格式返回；
- status方法：用于设置状态码：
- 更多响应的方式：<https://www.expressjs.com.cn/4x/api.html#res>

```
1  const express = require('express')
2
3  const app = express()
4
5  app.get('/login', (req, res, next) => {
6    res.json({
7      name: 'yihua'
8    })
9  })
10
11 app.listen(8000, () => {
12   console.log('express初体验')
13 })
14
```

10.5 Express的路由

我们可以使用 express.Router来创建一个路由处理程序

```
1  const express = require('express')
2  const userRouter = require('./router/users')
3  const app = express()
4
5  app.use('/users', userRouter)
6
7  app.listen(8000, () => {
8    console.log('路由服务器启动')
9  })
10
```

```
1  /**
```



```
2  * 举个例子:
3  *   请求所有的用户信息: get /users
4  *   请求所有的某个用户信息: get /users/:id
5  *   请求所有的某个用户信息: post /users body {username: passwod:}
6  *   请求所有的某个用户信息: delete /users/:id
7  *   请求所有的某个用户信息: patch /users/:id {nickname: }
8  */
9
10 const express = require('express');
11
12 const router = express.Router();
13
14 router.get('/', (req, res, next) => {
15   res.json(["why", "kobe", "lilei"]);
16 });
17
18 router.get('/:id', (req, res, next) => {
19   res.json(`${req.params.id}用户的信息`);
20 });
21
22 router.post('/', (req, res, next) => {
23   res.json("create user success~");
24 });
25
26 module.exports = router;
27
```

10.6 静态资源服务器

```
1  const express = require('express');
2
3  const app = express();
4
5  app.use(express.static('./build'));
6
7  app.listen(8000, () => {
8    console.log("路由服务器启动成功~");
9  });
10
```

10.7 错误处理（错误级别中间件）

```
1  const express = require('express');
2
3  const app = express();
4
5  const USERNAME_DOES_NOT_EXISTS = "USERNAME_DOES_NOT_EXISTS";
6  const USERNAME_ALREADY_EXISTS = "USERNAME_ALREADY_EXISTS";
7
8  app.post('/login', (req, res, next) => {
9    // 加入在数据中查询用户名时，发现不存在
10   const isLogin = false;
11   if (isLogin) {
12     res.json("user login success~");
13   } else {
14     // res.type(400);
15     // res.json("username does not exists~")
16     next(new Error(USERNAME_DOES_NOT_EXISTS));
17   }
18 })
19
20 app.post('/register', (req, res, next) => {
21   // 加入在数据中查询用户名时，发现不存在
22   const isExists = true;
23   if (!isExists) {
24     res.json("user register success~");
25   } else {
26     // res.type(400);
27     // res.json("username already exists~")
28     next(new Error(USERNAME_ALREADY_EXISTS));
29   }
30 });
31
32 app.use((err, req, res, next) => {
33   let status = 400;
34   let message = "";
35   console.log(err.message);
36
```

```

37  switch(err.message) {
38      case USERNAME_DOES_NOT_EXISTS:
39          message = "username does not exists~";
40          break;
41      case USERNAME_ALREADY_EXISTS:
42          message = "USERNAME_ALREADY_EXISTS~"
43          break;
44      default:
45          message = "NOT FOUND~"
46  }
47
48  res.status(status);
49  res.json({
50      errCode: status,
51      errMsg: message
52  })
53 })
54
55 app.listen(8000, () => {
56     console.log("路由服务器启动成功~");
57 });
58

```

10.8 express源码分析

首先通过express创建app,拿到的app挂载了很多方法，其中包括了listen。使用listen的时候会调用http.createServer(this)。然后调用http.listen并将参数传入。

当我们使用app.use的时候会将中间件进行注册，注册到express.router上。后续用户发送请求就会去express.router上匹配对应的layer。layer是app.use的时候注册的实例。然后会调用next()函数。next函数会将stack中的layer进行匹配，然后调用。并且记录当前调用的，下一次执行next就会在当前的执行的后续layer进行匹配执行。

1.调用express()到底创建的是什么？

express函数的本质其实是createApplication

```

1  function createApplication() {
2      var app = function(req, res, next) {
3          app.handle(req, res, next);
4      };
5
6      mixin(app, EventEmitter.prototype, false);

```

```

7   mixin(app, proto, false);
8
9   // expose the prototype that will get set on requests
10  app.request = Object.create(req, {
11    app: { configurable: true, enumerable: true, writable: true, value: app }
12  })
13
14  // expose the prototype that will get set on responses
15  app.response = Object.create(res, {
16    app: { configurable: true, enumerable: true, writable: true, value: app }
17  })
18
19  app.init();
20  return app;
21 }

```

app本质上就是一个函数，这个函数上挂载了很多属性，比如listen是通过mixin挂载在app上的。执行express实质上拿到的就是app

2.app.listen()如何结合原生启动服务器

```

1  app.listen = function listen() {
2    var server = http.createServer(this);
3    return server.listen.apply(server, arguments);
4  };

```

此时的this指向就是app，使用http创建server。再通过apply将server的listen执行改为创建的server

3.app.use(中间件)内部到底发生了什么

我们会发现无论是app.use还是app.methods都会注册一个主路由。app本质上会将所有的函数，交给这个主路由去处理的

第十一章：koa

11.1 koa初体验

```

1  const Koa = require('koa');
2
3  const app = new Koa();
4

```

```
5 app.use((ctx, next) => {
6   ctx.response.body = "Hello World";
7 });
8
9 app.listen(8000, () => {
10  console.log("koa初体验服务器启动成功~");
11 });
12
```

注册中间件

```
1 const Koa = require('koa');
2
3 const app = new Koa();
4
5 // use注册中间件
6 app.use((ctx, next) => {
7   if (ctx.request.url === '/login') {
8     if (ctx.request.method === 'GET') {
9       console.log("来到了这里~");
10      ctx.response.body = "Login Success~";
11    }
12   } else {
13     ctx.response.body = "other request~";
14   }
15 });
16
17 // 没有提供下面的注册方式
18 // methods方式: app.get().post
19 // path方式: app.use('/home', (ctx, next) => {})
20 // 连续注册: app.use((ctx, next) => {
21 //   }, (ctx, next) => {
22 // })
23
24 app.listen(8000, () => {
25  console.log("koa初体验服务器启动成功~");
26 });
27
```

11.2 路由

userRouter.allowedMethods()可以处理没有被注册的路由

```
1  const Koa = require('koa');
2
3  const userRouter = require('./router/user');
4
5  const app = new Koa();
6
7  app.use((ctx, next) => {
8    // ctx.response.body = "Hello World";
9    next();
10 });
11
12 app.use(userRouter.routes());
13 app.use(userRouter.allowedMethods());
14
15 app.listen(8000, () => {
16   console.log("koa路由服务器启动成功~");
17 });
18
```

./router/user.js

```
1  const Router = require('koa-router');
2
3  const router = new Router({prefix: "/users"});
4
5  router.get('/', (ctx, next) => {
6    ctx.response.body = "User Lists~";
7  });
8
9  router.put('/', (ctx, next) => {
10   ctx.response.body = "put request~";
11 });
12
13
14 module.exports = router;
```

11.3 参数解析

1.params - query

```
1  const Koa = require('koa')
2
3  const app = new Koa()
4  const Router = require('koa-router')
5
6  const userRouter = new Router({prefix: '/users'})
7
8  userRouter.get('/:id', (ctx, next) => {
9    console.log(ctx.request.params)
10   console.log(ctx.request.query)
11 })
12
13 app.use(userRouter.routes())
14
15 app.listen(8000, () => {
16   console.log('参数处理服务器启动成功~')
17 })
18
```

2.参数解析: json 、 formData

npm install koa-bodyparse

npm install koa-multer

```
1  const Koa = require('koa')
2  const bodyParser = require('koa-bodyparser')
3  const multer = require('koa-multer')
4  const Router = require('koa-router')
5
6  const app = new Koa()
7
8  const upload = multer()
9
10 // 解析json
11 app.use(bodyParser())
```

```
12 // 解析formData
13 app.use(upload.any())
14
15 app.use((ctx, next) => {
16   console.log(ctx.request.body)
17   console.log(ctx.req.body)
18   ctx.response.body = 'Hello World'
19 })
20
21 app.listen(8000, () => {
22   console.log('koa初体验服务器启动成功~')
23 })
24
```

11.4 Multer上传文件

```
1 const Koa = require('koa')
2 const Router = require('koa-router')
3 const multer = require('koa-multer')
4
5 const app = new Koa()
6 const uploadRouter = new Router({prefix: '/upload'})
7
8 const storage = multer.diskStorage({
9   destination: (req, res, cb) => {
10     cb(null, './uploads/')
11   },
12   filename: (req, file, cb) => {
13     cb(null, file.originalname)
14   }
15 })
16
17 const upload = multer({
18   // dest: './uploads/'
19   storage
20 })
21
22 uploadRouter.post('/avatar', upload.single('avatar'), (ctx, next) => {
23   console.log(ctx.req.file)
```



```
24   ctx.response.body = '上传头像成功~'
25 })
26
27 app.use(uploadRouter.routes())
28
29 app.listen(8000, () => {
30   console.log('koa初体验服务器启动成功~')
31 })
```

11.5 数据的响应

body将响应主体设置为以下之一

- string：字符串数据
- Buffer：Buffer数据
- Stream：流数据
- Object|| Array：对象或者数组
- null：不输出任何内容
- 如果response.status尚未设置，Koa会自动将状态设置为200或204。

请求状态：status

ctx.body是ctx.response.body 的代理

```
1  const Koa = require('koa');
2
3  const app = new Koa();
4
5  app.use((ctx, next) => {
6    // ctx.request.query
7    // ctx.query
8
9    // 设置内容
10   // ctx.response.body
11   // ctx.response.body = "Hello world~"
12   // ctx.response.body = {
13     //   name: "coderwhy",
14     //   age: 18,
15     //   avatar_url: "https://abc.png"
16   // };
17   // 设置状态码
18   // ctx.response.status = 400;
```

```
19 // ctx.response.body = ["abc", "cba", "nba"];
20
21 // ctx.response.body = "Hello World~";
22 ctx.status = 404;
23 ctx.body = "Hello Koa~";
24 });
25
26 app.listen(8000, () => {
27   console.log("koa初体验服务器启动成功~");
28 });
```

11.6 静态服务器

npm install koa-static

```
1 const Koa = require('koa');
2 const staticAssets = require('koa-static');
3
4 const app = new Koa();
5
6 app.use(staticAssets('./build'));
7
8 app.listen(8000, () => {
9   console.log("koa初体验服务器启动成功~");
10 });
```

11.7 错误处理

```
1 const Koa = require('koa');
2
3 const app = new Koa();
4
5 app.use((ctx, next) => {
6   const isLogin = false;
7   if (!isLogin) {
8     ctx.app.emit('error', new Error("您还没有登录~"), ctx);
9   }
10 });
11
```

```
12 app.on('error', (err, ctx) => {
13   ctx.status = 401;
14   ctx.body = err.message;
15 })
16
17 app.listen(8000, () => {
18   console.log("koa初体验服务器启动成功~");
19 });
```

11.8 和express对比

- express是完整和强大的，其中帮助我们内置了非常多好用的功能
- koa是简洁和自由的，它只包含最核心的功能，并不会对我们使用其他中间件进行任何的限制
 - 甚至是在app中连最基本的get、post都没有给我们提供；
 - 我们需要通过自己或者路由来判断请求方式或者其他功能；
- 因为express和koa框架他们的核心其实都是中间件：
 - 但是他们的中间件事实上，它们的中间件的执行机制是不同的，特别是针对某个中间件中包含异步操作时；

1.express实现-同步数据

```
1  const express = require('express');
2
3  const app = express();
4
5  const middleware1 = (req, res, next) => {
6    req.message = "aaa";
7    next();
8    res.end(req.message);
9  }
10
11 const middleware2 = (req, res, next) => {
12   req.message += "bbb";
13   next();
14 }
15
16 const middleware3 = (req, res, next) => {
17   req.message += "ccc";
18 }
19
```

```
20 app.use(middleware1, middleware2, middleware3);
21
22 app.listen(8000, () => {
23   console.log("服务器启动成功~");
24 })
```

2.express实现-异步数据

```
1  const express = require('express');
2  const axios = require('axios');
3
4  const app = express();
5
6  const middleware1 = (req, res, next) => {
7    req.message = "aaa";
8    next();
9  }
10
11 const middleware2 = (req, res, next) => {
12   req.message += "bbb";
13   next();
14 }
15
16 const middleware3 = async (req, res, next) => {
17   const result = await axios.get('http://123.207.32.32:9001/lyric?id=167876');
18   req.message += result.data.lrc.lyric;
19   res.end(req.message);
20 }
21
22 app.use(middleware1, middleware2, middleware3);
23
24 app.listen(8000, () => {
25   console.log("服务器启动成功~");
26 })
```

3.koa实现-同步数据

```
1  const Koa = require('koa');
2
```

```
3  const app = new Koa();
4
5  const middleware1 = (ctx, next) => {
6    ctx.message = "aaa";
7    next();
8    ctx.body = ctx.message;
9  }
10
11 const middleware2 = (ctx, next) => {
12   ctx.message += "bbb";
13   next();
14 }
15
16 const middleware3 = (ctx, next) => {
17   ctx.message += "ccc";
18 }
19
20 app.use(middleware1);
21 app.use(middleware2);
22 app.use(middleware3);
23
24 app.listen(8000, () => {
25   console.log("服务器启动成功~");
26 })
```

4.koa实现-异步数据

```
1  const Koa = require('koa');
2  const axios = require('axios');
3
4  const app = new Koa();
5
6  const middleware1 = async (ctx, next) => {
7    ctx.message = "aaa";
8    await next();
9    next();
10   ctx.body = ctx.message;
11 }
12
```

```
13 const middleware2 = async (ctx, next) => {
14   ctx.message += "bbb";
15   await next();
16 }
17
18 const middleware3 = async (ctx, next) => {
19   const result = await axios.get('http://123.207.32.32:9001/lyric?id=167876');
20   ctx.message += result.data.lrc.lyric;
21 }
22
23 app.use(middleware1);
24 app.use(middleware2);
25 app.use(middleware3);
26
27 app.listen(8000, () => {
28   console.log("服务器启动成功~");
29 })
```

11.9 Koa洋葱模型

- 中间件处理代码的过程
- Response返回body执行

先执行最外层中间件->根据next一层层执行，到了最后一层就会往外执行，一直等到所有中间件执行完才会执行到第一层next后面的代码。并且通过ctx.body进行返回。

第十二章：MySQL



12.1 初识数据库

终端连接数据库

mysql -uroot -p密码

或者 mysql -uroot -p 回车，另起一行输入密码。

显示数据库

show databases

创建数据库-表

- 在终端直接创建一个属于自己的新的数据库：create database coderhub;
- 使用我们创建的数据库coderhub：use coderhub;
- 在数据库中，创建一张表

```
1 create table user(
2   name varchar(20),
3   age int,
4   height double
5 );
6 # 插入数据
7 insert into user (name, age, height) values ('why', 18, 1.88);
8 insert into user (name, age, height) values ('kobe', 40, 1.98);
```

12.2 SQL语句的分类

常见的SQL语句我们可以分成四类

- DDL (Data Definition Language) : 数据定义语言
 - 可以通过DDL语句对数据库或者表进行: 创建、删除、修改等操作
- DML (Data Manipulation Language) : 数据操作语言
 - 可以通过DML语句对表进行: 添加、删除、修改等操作
- DQL (Data Query Language) : 数据查询语言
 - 可以通过DQL从数据库中查询记录; (重点)
- DCL (Data Control Language) : 数据控制语言;
 - 对数据库、表格的权限进行相关访问控制操作;

常用DDL语句

1.DDL-操作数据库

```
1 # 查询所有数据库
2 SHOW DATABASES;
3
4 # 选择某一个数据库
5 -- use 数据库名称
6 use coderhub
7
8 # 查看当前正在使用的数据库
9 SELECT DATABASE();
10
11 # 创建一个新的数据库
12 -- CREATE DATABASES 数据库名称;
13 -- CREATE DATABASES IF NOT EXISTS 数据库名称;
14 CREATE DATABASE IF NOT EXISTS 数据库名称
15                                DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci;
16
17 # 删除数据库
18 DROP DATABASE 数据库名称;
19 DROP DATABASE IF EXIST 数据库名称;
20
21 # 修改数据库的字符集和排序规则
22 ALTER DATABASE 数据库名称 CHARACTER SET = utf8 COLLATE = utf8_unicode_ci;
```

2.DDL-操作数据表

```
1 # 查看所有的表
```



```
2  SHOW TABLES;
3
4  # 新建表
5  CREATE TABLE IF NOT EXISTS `students` (
6      `name` VARCHAR(10) NOT NULL,
7      `age` int,
8      `score` int,
9      `height` DECIMAL(10,2),
10     `birthday` YEAR,
11     `phoneNum` VARCHAR(20) UNIQUE
12 );
13
14 # 删除表
15 DROP TABLE IF EXISTS `moment`;
16
17 # 查看表的结构
18 DESC students;
19 # 查看创建表的SQL语句
20 SHOW CREATE TABLE `students`;
21 -- CREATE TABLE `students` (
22 --     `name` varchar(10) DEFAULT NULL,
23 --     `age` int DEFAULT NULL,
24 --     `score` int DEFAULT NULL
25 -- ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
26
27 # 完整的创建表的语法
28 CREATE TABLE IF NOT EXISTS `users` (
29     id INT PRIMARY KEY AUTO_INCREMENT,
30     name VARCHAR(20) NOT NULL,
31     age INT DEFAULT 0,
32     phoneNum VARCHAR(20) UNIQUE DEFAULT '',
33     createTime TIMESTAMP
34 );
35
36 # 修改表
37 # 1. 修改表的名字
38 ALTER TABLE `users` RENAME TO `user`;
39 # 2. 添加一个新的列
40 ALTER TABLE `user` ADD `updateTime` TIMESTAMP;
```

```

41 # 3.修改字段的名称
42 ALTER TABLE `user` CHANGE `phoneNum` `telPhone` VARCHAR(20);
43 # 4.修改字段的类型
44 ALTER TABLE `user` MODIFY `name` VARCHAR(30);
45 # 5.删除某一个字段
46 ALTER TABLE `user` DROP `age`;
47
48 # 补充
49 # 根据一个表结构去创建另外一张表
50 CREATE TABLE `user2` LIKE `user`;
51 # 根据另外一个表中的所有内容，创建一个新的表
52 CREATE TABLE `user3` (SELECT * FROM `user`);

```

3.DML-对数据库增删改

```

1 # DML
2 # 插入数据
3 INSERT INTO `user` VALUES (110, 'why', '020-110110', '2020-10-20', '2020-11-11');
4 INSERT INTO `user` (name, telPhone, createTime, updateTime)
5                                     VALUES ('kobe', '000-1111111', '2020-10-
6                                     10', '2030-10-20');
7
8 INSERT INTO `user` (name, telPhone)
9                                     VALUES ('lilei', '000-1111112');
10
11 # 需求: createTime和updateTime可以自动设置值
12 ALTER TABLE `user` MODIFY `createTime` TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
13 ALTER TABLE `user` MODIFY `updateTime` TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
14 CURRENT_TIMESTAMP;
15
16 INSERT INTO `user` (name, telPhone) VALUES ('hmm', '000-1111212');
17 INSERT INTO `user` (name, telPhone) VALUES ('lucy', '000-1121212');
18
19 # 删除数据
20 # 删除所有的数据
21 DELETE FROM `user`;
22 # 删除符合条件的数据
23 DELETE FROM `user` WHERE id = 110;

```

```
24 # 更新数据
25 # 更新所有的数据
26 UPDATE `user` SET name = 'lily', telPhone = '010-110110';
27 # 更新符合条件的数据
28 UPDATE `user` SET name = 'lily', telPhone = '010-110110' WHERE id = 115;
```

4.DQL-聚合函数GroupBy

```
1 # 1.聚合函数的使用
2 # 求所有手机的价格的总和
3 SELECT SUM(price) totalPrice FROM `products`;
4 # 求一下华为手机的价格的总和
5 SELECT SUM(price) FROM `products` WHERE brand = '华为';
6 # 求华为手机的平均价格
7 SELECT AVG(price) FROM `products` WHERE brand = '华为';
8 # 最高手机的价格和最低手机的价格
9 SELECT MAX(price) FROM `products`;
10 SELECT MIN(price) FROM `products`;
11
12 # 求华为手机的个数
13 SELECT COUNT(*) FROM `products` WHERE brand = '华为';
14 SELECT COUNT(*) FROM `products` WHERE brand = '苹果';
15 SELECT COUNT(url) FROM `products` WHERE brand = '苹果';
16
17 SELECT COUNT(price) FROM `products`;
18 SELECT COUNT(DISTINCT price) FROM `products`;
19
20 # 2.GROUP BY的使用
21 SELECT brand, AVG(price), COUNT(*), AVG(score) FROM `products` GROUP BY brand;
22
23
24 # 3.HAVING的使用
25 SELECT brand, AVG(price) avgPrice, COUNT(*), AVG(score) FROM `products` GROUP BY brand
    HAVING avgPrice > 2000;
26
27
28 # 4.需求：求评分score > 7.5的手机的，平均价格是多少？
29 # 升级：平均分大于7.5的手机，按照品牌进行分类，求出平均价格。
30 SELECT brand, AVG(price) FROM `products` WHERE score > 7.5 GROUP BY brand;
```

5.DQL-数据的查询语句

```
1  # 创建products的表
2  CREATE TABLE IF NOT EXISTS `products` (
3      id INT PRIMARY KEY AUTO_INCREMENT,
4      brand VARCHAR(20),
5      title VARCHAR(100) NOT NULL,
6      price DOUBLE NOT NULL,
7      score DECIMAL(2,1),
8      voteCnt INT,
9      url VARCHAR(100),
10     pid INT
11 );
12
13 # 1.基本查询
14 # 查询表中所有的字段以及所有的数据
15 SELECT * FROM `products`;
16 # 查询指定的字段
17 SELECT title, price FROM `products`;
18 # 对字段结果起一个别名
19 SELECT title as phoneTitle, price as currentPrice FROM `products`;
20
21
22 # 2.where条件
23 # 2.1. 条件判断语句
24 # 案例：查询价格小于1000的手机
25 SELECT title, price FROM `products` WHERE price < 1000;
26 # 案例二：价格等于999的手机
27 SELECT * FROM `products` WHERE price = 999;
28 # 案例三：价格不等于999的手机
29 SELECT * FROM `products` WHERE price != 999;
30 SELECT * FROM `products` WHERE price <> 999;
31 # 案例四：查询品牌是华为的手机
32 SELECT * FROM `products` WHERE brand = '华为';
33
34 # 2.2. 逻辑运算语句
35 # 案例一：查询1000到2000之间的手机
36 SELECT * FROM `products` WHERE price > 1000 AND price < 2000;
37 SELECT * FROM `products` WHERE price > 1000 && price < 2000;
```

```

38 # BETWEEN AND 包含等于
39 SELECT * FROM `products` WHERE price BETWEEN 1099 AND 2000;
40
41 # 案例二：价格在5000以上或者是品牌是华为的手机
42 SELECT * FROM `products` WHERE price > 5000 || brand = '华为';
43
44 # 将某些值设置为NULL
45 -- UPDATE `products` SET url = NULL WHERE id >= 85 and id <= 88;
46 # 查询某一个值为NULL
47 SELECT * FROM `products` WHERE url IS NULL;
48 -- SELECT * FROM `products` WHERE url = NULL;
49 -- SELECT * FROM `products` WHERE url IS NOT NULL;
50
51 # 2.3.模糊查询
52 SELECT * FROM `products` WHERE title LIKE '%M%';
53 SELECT * FROM `products` WHERE title LIKE '%P%';
54 SELECT * FROM `products` WHERE title LIKE '_P%';
55
56
57 # 3.对结果进行排序
58 SELECT * FROM `products` WHERE brand = '华为' || brand = '小米' || brand = '苹果';
59 SELECT * FROM `products` WHERE brand IN ('华为', '小米', '苹果')
60 ORDER BY price ASC, score DESC;
61
62
63 # 4.分页查询
64 # LIMIT limit OFFSET offset;
65 # Limit offset, limit;
66 SELECT * FROM `products` LIMIT 20 OFFSET 0;
67 SELECT * FROM `products` LIMIT 20 OFFSET 20;
68 SELECT * FROM `products` LIMIT 40, 20;

```

6.对象和数组类型

```

1 # 将联合查询到的数据转成对象（一对多）
2 SELECT
3     products.id id, products.title title, products.price price,
4     JSON_OBJECT('id', brand.id, 'name', brand.name, 'website', brand.website) brand
5 FROM `products`

```

```

6  LEFT JOIN `brand` ON products.brand_id = brand.id;
7
8  # 将查询到的多条数据，组织成对象，放入到一个数组中(多对多)
9  SELECT
10     stu.id, stu.name, stu.age,
11     JSON_ARRAYAGG(JSON_OBJECT('id', cs.id, 'name', cs.name, 'price', cs.price))
12 FROM `students` stu
13 JOIN `students_select_courses` ssc ON stu.id = ssc.student_id
14 JOIN `courses` cs ON ssc.course_id = cs.id
15 GROUP BY stu.id;
16
17 SELECT * FROM products WHERE price > 6000;

```

7.多表的设计-查询

```

1  # 1.获取到的是笛卡尔乘积
2  SELECT * FROM `products`, `brand`;
3  # 获取到的是笛卡尔乘积进行筛选
4  SELECT * FROM `products`, `brand` WHERE products.brand_id = brand.id;
5
6  # 2.左连接
7  # 2.1. 查询所有的手机（包括没有品牌信息的手机）以及对应的品牌 null
8  SELECT * FROM `products` LEFT OUTER JOIN `brand` ON products.brand_id = brand.id;
9
10 # 2.2. 查询没有对应品牌数据的手机
11 SELECT * FROM `products` LEFT JOIN `brand` ON products.brand_id = brand.id WHERE
    brand.id IS NULL;
12 -- SELECT * FROM `products` LEFT JOIN `brand` ON products.brand_id = brand.id WHERE
    brand_id IS NULL;
13
14
15 # 3.右连接
16 # 3.1. 查询所有的品牌（没有对应的手机数据，品牌也显示）以及对应的手机数据；
17 SELECT * FROM `products` RIGHT OUTER JOIN `brand` ON products.brand_id = brand.id;
18
19 # 3.2. 查询没有对应手机的品牌信息
20 SELECT * FROM `products` RIGHT JOIN `brand` ON products.brand_id = brand.id WHERE
    products.brand_id IS NULL;
21
22

```

```

23 # 4.内连接
24 SELECT * FROM `products` JOIN `brand` ON products.brand_id = brand.id;
25 SELECT * FROM `products` JOIN `brand` ON products.brand_id = brand.id WHERE price =
    8699;
26
27 # 5.全连接
28 # mysql是不支持FULL OUTER JOIN
29 SELECT * FROM `products` FULL OUTER JOIN `brand` ON products.brand_id = brand.id;
30
31
32 (SELECT * FROM `products` LEFT OUTER JOIN `brand` ON products.brand_id = brand.id)
33 UNION
34 (SELECT * FROM `products` RIGHT OUTER JOIN `brand` ON products.brand_id = brand.id)
35
36 (SELECT * FROM `products` LEFT OUTER JOIN `brand` ON products.brand_id = brand.id WHERE
    brand.id IS NULL)
37 UNION
38 (SELECT * FROM `products` RIGHT OUTER JOIN `brand` ON products.brand_id = brand.id WHERE
    products.brand_id IS NULL)

```

8.多表的设计-外键

```

1 # 1.创建brand的表和插入数据
2 CREATE TABLE IF NOT EXISTS `brand` (
3     id INT PRIMARY KEY AUTO_INCREMENT,
4     name VARCHAR(20) NOT NULL,
5     website VARCHAR(100),
6     phoneRank INT
7 );
8
9 INSERT INTO `brand` (name, website, phoneRank) VALUES ('华为', 'www.huawei.com', 2);
10 INSERT INTO `brand` (name, website, phoneRank) VALUES ('苹果', 'www.apple.com', 10);
11 INSERT INTO `brand` (name, website, phoneRank) VALUES ('小米', 'www.mi.com', 5);
12 INSERT INTO `brand` (name, website, phoneRank) VALUES ('oppo', 'www.oppo.com', 12);
13
14 INSERT INTO `brand` (name, website, phoneRank) VALUES ('京东', 'www.jd.com', 8);
15 INSERT INTO `brand` (name, website, phoneRank) VALUES ('Google', 'www.google.com', 9);
16
17
18 # 2.给brand_id设置引用brand中的id的外键约束

```

```
19 # 添加一个brand_id字段
20 ALTER TABLE `products` ADD `brand_id` INT;
21 -- ALTER TABLE `products` DROP `brand_id`;
22
23 # 修改brand_id为外键
24 ALTER TABLE `products` ADD FOREIGN KEY(`brand_id`) REFERENCES brand(id);
25
26 # 设置brand_id的值
27 UPDATE `products` SET `brand_id` = 1 WHERE `brand` = '华为';
28 UPDATE `products` SET `brand_id` = 2 WHERE `brand` = '苹果';
29 UPDATE `products` SET `brand_id` = 3 WHERE `brand` = '小米';
30 UPDATE `products` SET `brand_id` = 4 WHERE `brand` = 'oppo';
31
32 # 3.修改和删除外键引用的id （错误）
33 UPDATE `brand` SET `id` = 100 WHERE `id` = 1;
34
35 # 4.修改brand_id关联外键时的action
36 # 4.1.获取到目前的外键的名称
37 SHOW CREATE TABLE `products`;
38
39
40 -- CREATE TABLE `products` (
41 --   `id` int NOT NULL AUTO_INCREMENT,
42 --   `brand` varchar(20) DEFAULT NULL,
43 --   `title` varchar(100) NOT NULL,
44 --   `price` double NOT NULL,
45 --   `score` decimal(2,1) DEFAULT NULL,
46 --   `voteCnt` int DEFAULT NULL,
47 --   `url` varchar(100) DEFAULT NULL,
48 --   `pid` int DEFAULT NULL,
49 --   `brand_id` int DEFAULT NULL,
50 --   PRIMARY KEY (`id`),
51 --   KEY `brand_id` (`brand_id`),
52 --   CONSTRAINT `products_ibfk_1` FOREIGN KEY (`brand_id`) REFERENCES `brand` (`id`)
53 -- ) ENGINE=InnoDB AUTO_INCREMENT=109 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
54
55 # 4.2.根据名称将外键删除掉
56 ALTER TABLE `products` DROP FOREIGN KEY products_ibfk_1;
57
```



```
58 # 4.2.重新添加外键约束
59 ALTER TABLE `products` ADD FOREIGN KEY (brand_id) REFERENCES brand(id)
                                     ON UPDATE CASCADE
                                     ON DELETE RESTRICT;
60
61 UPDATE `brand` SET `id` = 100 WHERE `id` = 1;
```

9.多对多关系-设计

```
1 # 1.基本数据的模拟
2 CREATE TABLE IF NOT EXISTS students(
3     id INT PRIMARY KEY AUTO_INCREMENT,
4     name VARCHAR(20) NOT NULL,
5     age INT
6 );
7
8 CREATE TABLE IF NOT EXISTS courses(
9     id INT PRIMARY KEY AUTO_INCREMENT,
10    name VARCHAR(20) NOT NULL,
11    price DOUBLE
12 );
13
14 INSERT INTO `students` (name, age) VALUES('why', 18);
15 INSERT INTO `students` (name, age) VALUES('tom', 22);
16 INSERT INTO `students` (name, age) VALUES('lilei', 25);
17 INSERT INTO `students` (name, age) VALUES('lucy', 16);
18 INSERT INTO `students` (name, age) VALUES('lily', 20);
19
20 INSERT INTO `courses` (name, price) VALUES ('英语', 100);
21 INSERT INTO `courses` (name, price) VALUES ('语文', 666);
22 INSERT INTO `courses` (name, price) VALUES ('数学', 888);
23 INSERT INTO `courses` (name, price) VALUES ('历史', 80);
24 INSERT INTO `courses` (name, price) VALUES ('物理', 888);
25 INSERT INTO `courses` (name, price) VALUES ('地理', 333);
26
27
28 # 2.建立关系表
29 CREATE TABLE IF NOT EXISTS `students_select_courses` (
```

```

30         id INT PRIMARY KEY AUTO_INCREMENT,
31         student_id INT NOT NULL,
32         course_id INT NOT NULL,
33         FOREIGN KEY (student_id) REFERENCES students(id) ON UPDATE CASCADE,
34         FOREIGN KEY (course_id) REFERENCES courses(id) ON UPDATE CASCADE
35     );
36
37 # 3. 学生选课
38 # why 选择了英文、数学、历史
39 INSERT INTO `students_select_courses` (student_id, course_id) VALUES (1, 1);
40 INSERT INTO `students_select_courses` (student_id, course_id) VALUES (1, 3);
41 INSERT INTO `students_select_courses` (student_id, course_id) VALUES (1, 4);
42
43
44 INSERT INTO `students_select_courses` (student_id, course_id) VALUES (3, 2);
45 INSERT INTO `students_select_courses` (student_id, course_id) VALUES (3, 4);
46
47
48 INSERT INTO `students_select_courses` (student_id, course_id) VALUES (5, 2);
49 INSERT INTO `students_select_courses` (student_id, course_id) VALUES (5, 3);
50 INSERT INTO `students_select_courses` (student_id, course_id) VALUES (5, 4);
51
52
53 # 4. 查询的需求
54 # 4.1. 查询所有有选课的学生，选择了哪些课程
55 SELECT stu.id id, stu.name stuName, stu.age stuAge, cs.id csId, cs.name csName,
56        cs.price csPrice
57 FROM `students` stu
58 JOIN `students_select_courses` ssc ON stu.id = ssc.student_id
59 JOIN `courses` cs ON ssc.course_id = cs.id;
60
61 # 4.2. 查询所有的学生的选课情况
62 SELECT stu.id id, stu.name stuName, stu.age stuAge, cs.id csId, cs.name csName,
63        cs.price csPrice
64 FROM `students` stu
65 LEFT JOIN `students_select_courses` ssc ON stu.id = ssc.student_id
66 LEFT JOIN `courses` cs ON ssc.course_id = cs.id;
67
68 # 4.3. 哪些学生是没有选课

```

```

68 SELECT stu.id id, stu.name stuName, stu.age stuAge, cs.id csId, cs.name csName,
    cs.price csPrice
69 FROM `students` stu
70 LEFT JOIN `students_select_courses` ssc ON stu.id = ssc.student_id
71 LEFT JOIN `courses` cs ON ssc.course_id = cs.id
72 WHERE cs.id IS NULL;
73
74 # 4.4. 查询哪些课程是没有被选择的
75 SELECT stu.id id, stu.name stuName, stu.age stuAge, cs.id csId, cs.name csName,
    cs.price csPrice
76 FROM `students` stu
77 RIGHT JOIN `students_select_courses` ssc ON stu.id = ssc.student_id
78 RIGHT JOIN `courses` cs ON ssc.course_id = cs.id
79 WHERE stu.id IS NULL;;
80
81 # 4.5. 某一个学生选了哪些课程 (why)
82 SELECT stu.id id, stu.name stuName, stu.age stuAge, cs.id csId, cs.name csName,
    cs.price csPrice
83 FROM `students` stu
84 LEFT JOIN `students_select_courses` ssc ON stu.id = ssc.student_id
85 LEFT JOIN `courses` cs ON ssc.course_id = cs.id
86 WHERE stu.id = 2;

```

12.3 SQL数据类型

MySQL支持的数据类型有：数字类型，日期和时间类型，字符串（字符和字节）类型，空间类型和 JSON数据类型。

1.数字类型

整数数字类型：INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT;

Type	Storage (Bytes)	Minimum Value Signed	Minimum Value Unsigned	Maximum Value Signed	Maximum Value Unsigned
TINYINT	1	-128	0	127	255
SMALLINT	2	-32768	0	32767	65535
MEDIUMINT	3	-8388608	0	8388607	16777215
INT	4	-2147483648	0	2147483647	4294967295
BIGINT	8	-2 ⁶³	0	2 ⁶³ -1	2 ⁶⁴ -1

浮点数字类型：FLOAT, DOUBLE (FLOAT是4个字节, DOUBLE是8个字节)

精确数字类型：DECIMAL, NUMERIC (DECIMAL是NUMERIC的实现形式) ;

2.日期类型

- YEAR以YYYY格式显示值
 - 范围 1901到2155, 和 0000。
- DATE类型用于具有日期部分但没有时间部分的值:
 - DATE以格式YYYY-MM-DD显示值 ;
 - 支持的范围是 '1000-01-01' 到 '9999-12-31';
- DATETIME类型用于包含日期和时间部分的值:
 - DATETIME以格式'YYYY-MM-DD hh:mm:ss'显示值;
 - 支持的范围是1000-01-01 00:00:00到9999-12-31 23:59:59;
- TIMESTAMP数据类型被用于同时包含日期和时间部分的值:
 - TIMESTAMP以格式'YYYY-MM-DD hh:mm:ss'显示值;
 - 但是它的范围是UTC的时间范围: '1970-01-01 00:00:01'到'2038-01-19 03:14:07';
- 另外: DATETIME或TIMESTAMP 值可以包括在高达微秒 (6位) 精度的后小数秒一部分
 - 比如DATETIME表示的范围可以是'1000-01-01 00:00:00.000000'到'9999-12-31 23:59:59.999999';

3.字符串类型

- CHAR类型在创建表时为固定长度, 长度可以是0到255之间的任何值;
 - 在被查询时, 会删除后面的空格;
- VARCHAR类型的值是可变长度的字符串, 长度可以指定为0到65535之间的值;
 - 在被查询时, 不会删除后面的空格;
- BINARY和VARBINARY 类型用于存储二进制字符串, 存储的是字节字符串;
 - <https://dev.mysql.com/doc/refman/8.0/en/binary-varbinary.html>
- BLOB用于存储大的二进制类型;
- TEXT用于存储大的字符串类型

12.4 表约束

主键: PRIMARY KEY

一张表中, 我们为了区分每一条记录的唯一性, 必须有一个字段是永远不会重复, 并且不会为空的, 这个字段我们通常会 将它设置为主键:

- 主键是表中唯一的索引;
- 并且必须是NOT NULL的, 如果没有设置 NOT NULL, 那么MySQL也会隐式的设置为NOT NULL;
- 主键也可以是多列索引, PRIMARY KEY(key_part, ...), 我们一般称之为联合主键;
- 建议: 开发中主键字段应该是和业务无关的, 尽量不要使用业务字段来作为主键;

唯一: UNIQUE

某些字段在开发中我们希望是唯一的, 不会重复的, 比如手机号码、身份证号码等, 这个字段我们可以使用UNIQUE来约束:

- 使用UNIQUE约束的字段在表中必须是不同的;
- 对于所有引擎, UNIQUE 索引允许NULL包含的列具有多个值NULL。

不能为空：NOT NULL

某些字段我们要求用户必须插入值，不可以为空，这个时候我们可以使用 NOT NULL 来约束

默认值：DEFAULT

某些字段我们希望在没有设置值时给予一个默认值，这个时候我们可以使用 DEFAULT来完成

自动递增：AUTO_INCREMENT

某些字段我们希望不设置值时可以进行递增，比如用户的id，这个时候可以使用AUTO_INCREMENT来完成；

外键约束

12.5 更新外键

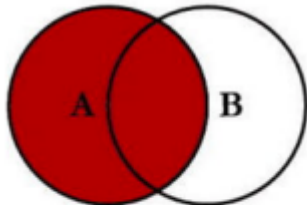
我们可以给更新或者删除时设置几个值：

- RESTRICT（默认属性）：当更新或删除某个记录时，会检查该记录是否有关联的外键记录，有的话会报错的，不允许更新或删除；
- NO ACTION：和RESTRICT是一致的，是在SQL标准中定义的；
- CASCADE：当更新或删除某个记录时，会检查该记录是否有关联的外键记录，有的话：
 - 更新：那么会更新对应的记录；
 - 删除：那么关联的记录会被一起删除掉；
- SET NULL：当更新或删除某个记录时，会检查该记录是否有关联的外键记录，有的话，将对应的值设置为NULL；

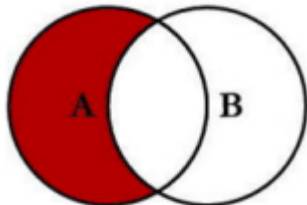
12.6 多表查询、多表之间的连接

```
1 # 多表查询
2 select * from products , brand where products.brand_id = brand.id;
```

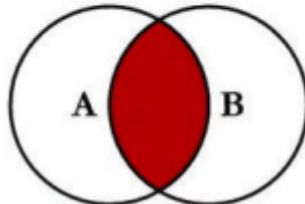
SQL JOINS



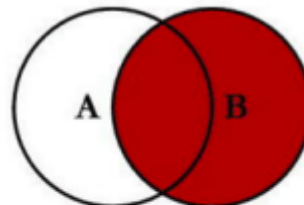
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



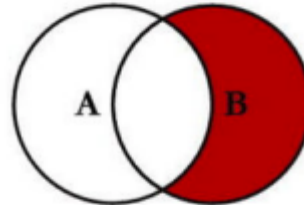
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



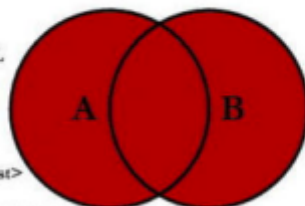
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



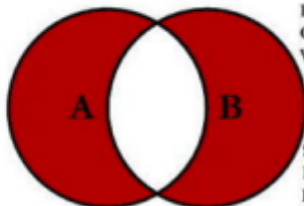
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

© C.L. Moffitt, 2008

```
1  •# 多表查询
2  select * from products , brand where products.brand_id = brand.id;
3
4  # 左连接
5  # 2.1 查询所有的手机（包括没有品牌信息的手机）以及对应的品牌 null
6  select * from products left outer join brand on products.brand_id = brand.id;
7
8  # 2.2 查询没有对应匹配的手机
9  select * from products left join brand on products.brand_id where brand.id is NULL;
10
11 # 右连接
12 # 3.1 查询所有的品牌以及对应的品牌对应的手机数据
13 select * from products right outer join brand on products.brand_id = brand.id;
14
15 # 3.2 查询没有对应手机的匹配
16 select * from products right outer join brand on products.brand_id = brand.id WHERE
    products.id is null;
17
18 # 4. 内连接
19 select * from products join brand on products.brand_id=brand.id;
20
```

```

21 # 5.全连接
22 # mysql是不支持FULL OUTER JOIN
23 select * from products full outer join brand on products.brand_id = brand.id;
24
25 (SELECT * FROM `products` LEFT OUTER JOIN `brand` ON products.brand_id = brand.id)
26 UNION
27 (SELECT * FROM `products` RIGHT OUTER JOIN `brand` ON products.brand_id = brand.id)
28
29 (SELECT * FROM `products` LEFT OUTER JOIN `brand` ON products.brand_id = brand.id WHERE
    brand.id IS NULL)
30 UNION
31 (SELECT * FROM `products` RIGHT OUTER JOIN `brand` ON products.brand_id = brand.id WHERE
    products.brand_id IS NULL)

```

第十三章：Node使用MySQL

13.1 认识mysql2

如何可以在Node的代码中执行SQL语句来，这里我们可以借助于两个库：

- mysql：最早的Node连接MySQL的数据库驱动；
- mysql2：在mysql的基础之上，进行了很多的优化、改进；

mysql2兼容mysql的API，并且提供了一些附加功能

- 更快/更好的性能；
- Prepared Statement（预编译语句）：
 - 提高性能：将创建的语句模块发送给MySQL，然后MySQL编译（解析、优化、转换）语句模块，并且存储它但是不执行，之后我们在真正执行时会给?提供实际的参数才会执行；就算多次执行，也只会编译一次，所以性能是更高的；
 - 防止SQL注入：之后传入的值不会像模块引擎那样就编译，那么一些SQL注入的内容不会被执行；or 1 = 1 不会被执行；
- 支持Promise，所以我们可以使用async和await语法

13.2 使用mysql2

npm install mysql2

- 第一步：创建连接（通过createConnection），并且获取连接对象；
- 第二步：执行SQL语句即可（通过query）；

```

1 const mysql = require('mysql2')
2
3 // 1.创建数据库连接

```

```

4  const connection = mysql.createConnection({
5    host: 'localhost',
6    port: 3306,
7    database: 'coderhub',
8    user: 'root',
9    password: 'root'
10 })
11
12 // 2.执行sql语句
13 const statement = `
14 select * from products where price > 8000
15 `
16 connection.query(statement, (err, rst, fields) => {
17   console.log(rst)
18 })

```

13.3 Prepared Statement

优点:

- 提高性能: 将创建的语句模块发送给MySQL, 然后MySQL编译(解析、优化、转换)语句模块, 并且存储它但是不执行, 之后我们在真正执行时会给?提供实际的参数才会执行; 就算多次执行, 也只会编译一次, 所以性能是更高的;
- 防止SQL注入: 之后传入的值不会像模块引擎那样就编译, 那么一些SQL注入的内容不会被执行; or 1 = 1不会被执行;
- 强调: 如果再次执行该语句, 它将会从LRU (Least Recently Used) Cache中获取获取, 省略了编译statement的时间来提高性能。

```

1  const mysql = require('mysql2')
2
3  // 1.创建数据库连接
4  const connection = mysql.createConnection({
5    host: 'localhost',
6    port: 3306,
7    database: 'coderhub',
8    user: 'root',
9    password: 'root'
10 })
11
12 // 2.执行sql语句
13 const statement = `

```



```

14 select * from products where price > ? and score > ?;
15 `
16 connection.execute(statement, [6000, 7], (err, rst, fields) => {
17   console.log(rst)
18 })

```

13.4 Connection Pools

前面我们是创建了一个连接（connection），但是如果我们有多个请求的话，该连接很有可能正在被占用，那么 我们是否需要每次一个请求都去创建一个新的连接呢？

- 事实上，mysql2给我们提供了连接池（connection pools）；
- 连接池可以在需要的时候自动创建连接，并且创建的连接不会被销毁，会放到连接池中，后续可以继续使用；
- 我们可以在创建连接池的时候设置LIMIT，也就是最大创建个数；

```

1  const mysql = require('mysql2')
2
3  // 1.创建数据库连接池
4  const connections = mysql.createPool({
5    host: 'localhost',
6    port: 3306,
7    database: 'coderhub',
8    user: 'root',
9    password: 'root',
10   connectionLimit: 10
11 })
12
13 // 2.使用连接池
14 const statement = `
15 select * from products where price > ? and score > ?
16 `
17 connections.execute(statement, [6000, 7], (err, rst) => {
18   console.log(rst)
19 })

```

13.5 Promise方式

```

1  const mysql = require('mysql2')
2
3  // 1.创建数据库连接池

```

```

4  const connections = mysql.createPool({
5    host: 'localhost',
6    port: 3306,
7    database: 'coderhub',
8    user: 'root',
9    password: 'root',
10   connectionLimit: 10
11 })
12
13 // 2.使用连接池
14 const statement = `
15 select * from products where price > ? and score > ?
16 `
17 connections.promise().execute(statement, [6000, 7]).then(([results]) => {
18   console.log(results)
19 }).catch(err => {
20   console.log(err)
21 })

```

13.6 认识ORM

对象关系映射，是一种程序设计的方案：

- 从效果上来讲，它提供了一个可在编程语言中，使用虚拟对象数据库的效果；
- 比如在Java开发中经常使用的ORM包括：Hibernate、MyBatis；

Node当中的ORM我们通常使用的是 sequelize；

- Sequelize是用于Postgres，MySQL，MariaDB，SQLite和Microsoft SQL Server的基于Node.js 的 ORM；
- 它支持非常多的功能

如果我们希望将Sequelize和MySQL一起使用，那么我们需要先安装两个东西：

- mysql2：sequelize在操作mysql时使用的是mysql2；
- sequelize：使用它来让对象映射到表中；

```
1 npm install sequelize mysql2
```

13.7 Sequelize的使用

1.Sequelize的连接数据库：

- 第一步：创建一个Sequelize的对象，并且指定数据库、用户名、密码、数据库类型、主机地址等；
- 第二步：测试连接是否成功；

```

1  const { Sequelize } = require('sequelize')
2
3  const sequelize = new Sequelize('coderhub', 'root', 'root', {
4    port: 3306,
5    host: 'localhost',
6    dialect: 'mysql'
7  })
8
9  sequelize.authenticate().then(()=>{
10    console.log('连接数据库成功');
11  }).catch(err=>{
12    console.log(err);
13  })

```

2.Sequelize单表操作

```

1  const { Sequelize, Model, DataTypes , Op} = require('sequelize')
2
3  const sequelize = new Sequelize('coderhub', 'root', 'root', {
4    port: 3306,
5    host: 'localhost',
6    dialect: 'mysql'
7  })
8
9  class Product extends Model {}
10 Product.init({
11   id: {
12     type: DataTypes.INTEGER,
13     primaryKey: true,
14     autoIncrement: true
15   },
16   title: {
17     type: DataTypes.STRING,
18     allowNull: false
19   },
20   price: DataTypes.DOUBLE,
21   score: DataTypes.DOUBLE
22 }, {

```

```
23   tableName: 'products',
24   createdAt: false,
25   updatedAt: false,
26   sequelize
27 })
28
29 async function queryproducts(){
30   // 1.查询products表中所有内容
31   const rst = await Product.findAll()
32   console.log(rst);
33
34   // 2.查询价格大于等于9900的手机
35   const rst1 = await Product.findAll({
36     where:{
37       price:{
38         [Op.gte]:9900
39       }
40     }
41   })
42   console.log(rst1);
43
44   // 3.创建用户
45   const rst2 = await Product.create({
46     title:'三星Nova',
47     price:9999,
48     score:9.9
49   })
50
51   console.log(rst2);
52
53   // 4.更新数据
54   const rst3 = await Product.update({
55     price:3688
56   },{
57     where:{
58       id:1
59     }
60   })
61
```

```
62 console.log(rst3);
63 }
64
65 queryproducts()
```

3.Sequelize一对多

```
1  const { Sequelize, DataTypes, Model, Op } = require('sequelize');
2
3  const sequelize = new Sequelize("coderhub", 'root', 'root', {
4    host: 'localhost',
5    dialect: 'mysql'
6  });
7
8  class Brand extends Model {};
9  Brand.init({
10    id: {
11      type: DataTypes.INTEGER,
12      primaryKey: true,
13      autoIncrement: true
14    },
15    name: {
16      type: DataTypes.STRING,
17      allowNull: false
18    },
19    website: DataTypes.STRING,
20    phoneRank: DataTypes.INTEGER
21  }, {
22    tableName: 'brand',
23    createdAt: false,
24    updatedAt: false,
25    sequelize
26  });
27
28  class Product extends Model {}
29  Product.init({
30    id: {
31      type: DataTypes.INTEGER,
32      primaryKey: true,
```

```
33     autoIncrement: true
34   },
35   title: {
36     type: DataTypes.STRING,
37     allowNull: false
38   },
39   price: DataTypes.DOUBLE,
40   score: DataTypes.DOUBLE,
41   brandId: {
42     field: 'brand_id',
43     type: DataTypes.INTEGER,
44     references: {
45       model: Brand,
46       key: 'id'
47     }
48   }
49 }, {
50   tableName: 'products',
51   createdAt: false,
52   updatedAt: false,
53   sequelize
54 });
55
56 // 将两张表联系在一起
57 Product.belongsTo(Brand, {
58   foreignKey: 'brandId'
59 });
60
61 async function queryProducts() {
62   const result = await Product.findAll({
63     include: {
64       model: Brand
65     }
66   });
67   console.log(result);
68 }
69
70 queryProducts();
```

4.Sequelize多对多

```
1  const { Sequelize, DataTypes, Model, Op } = require('sequelize');
2
3  const sequelize = new Sequelize("coderhub", 'root', 'root', {
4    host: 'localhost',
5    dialect: 'mysql'
6  });
7
8  // Student
9  class Student extends Model {}
10 Student.init({
11   id: {
12     type: DataTypes.INTEGER,
13     primaryKey: true,
14     autoIncrement: true
15   },
16   name: {
17     type: DataTypes.STRING,
18     allowNull: false
19   },
20   age: DataTypes.INTEGER
21 }, {
22   tableName: 'students',
23   createdAt: false,
24   updatedAt: false,
25   sequelize
26 });
27
28 // Course
29 class Course extends Model {}
30 Course.init({
31   id: {
32     type: DataTypes.INTEGER,
33     primaryKey: true,
34     autoIncrement: true
35   },
36   name: {
37     type: DataTypes.STRING,
```

```
38     allowNotNull: false
39   },
40   price: DataTypes.DOUBLE
41 }, {
42   tableName: 'courses',
43   createdAt: false,
44   updatedAt: false,
45   sequelize
46 });
47
48 // StudentCourse
49 class StudentCourse extends Model {}
50 StudentCourse.init({
51   id: {
52     type: DataTypes.INTEGER,
53     primaryKey: true,
54     autoIncrement: true
55   },
56   studentId: {
57     type: DataTypes.INTEGER,
58     references: {
59       model: Student,
60       key: 'id'
61     },
62     field: 'student_id'
63   },
64   courseId: {
65     type: DataTypes.INTEGER,
66     references: {
67       model: Course,
68       key: 'id'
69     },
70     field: 'course_id'
71   }
72 }, {
73   tableName: 'students_select_courses',
74   createdAt: false,
75   updatedAt: false,
76   sequelize
77 });
```



```
78
79 // 多对多关系的联系
80 Student.belongsToMany(Course, {
81   through: StudentCourse,
82   foreignKey: 'studentId',
83   otherKey: 'courseId'
84 });
85
86 Course.belongsToMany(Student, {
87   through: StudentCourse,
88   foreignKey: 'courseId',
89   otherKey: 'studentId'
90 });
91
92 async function queryProducts() {
93   const result = await Student.findAll({
94     include: {
95       model: Course
96     }
97   });
98   console.log(result);
99 }
100
101 queryProducts();
```

第十四章：coderhub实战

14.1 coderhub功能接口说明

- 用户管理系统
- 内容管理系统
- 内容评论管理
- 内容标签管理
- 文件管理系统

项目的搭建

功能一：目录结构的划分

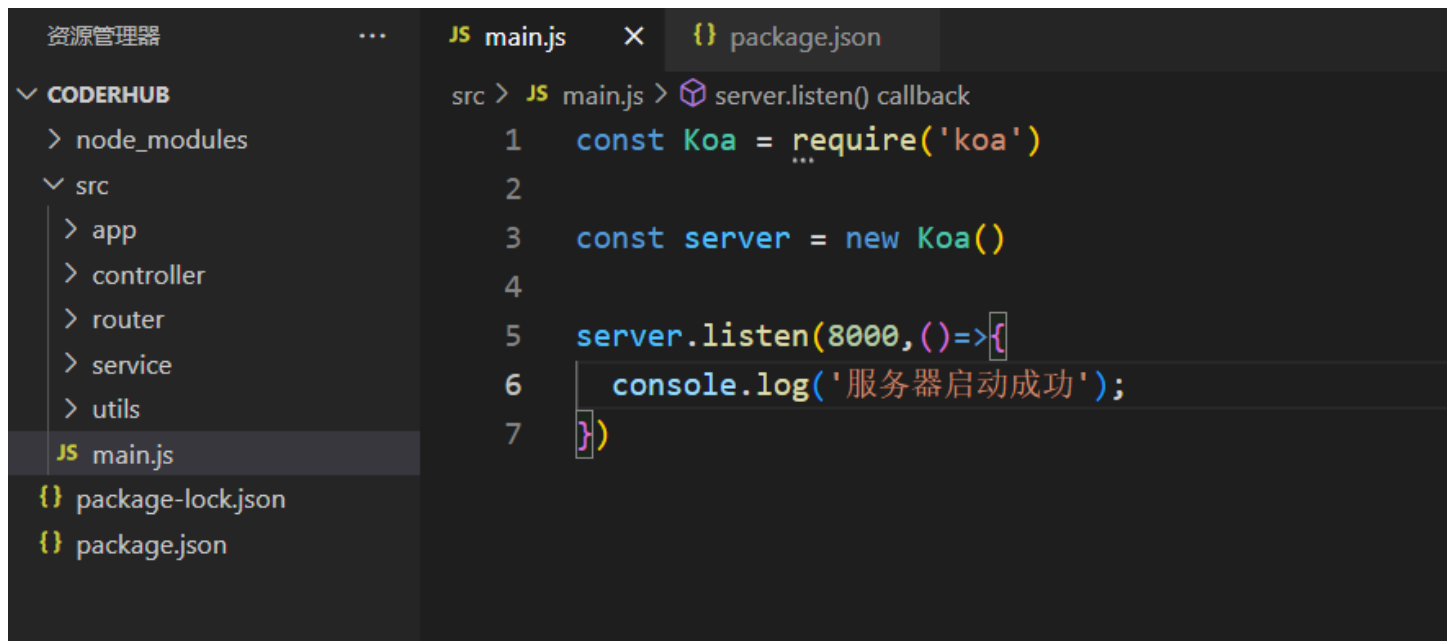
- 按照功能模块划分
- 按照业务模块划分

功能二：应用配置信息写到环境变量

- 编写.env文件
- 通过dotenv加载配置的变量

功能三：创建和启动服务器

- 基于koa创建app;
- 启动服务器;



```

src > JS main.js > server.listen() callback
1  const Koa = require('koa')
2
3  const server = new Koa()
4
5  server.listen(8000, ()=>{
6    console.log('服务器启动成功');
7  })
  
```

使用dotenv进行环境变量。npm i dotenv
.env

```

1  APP_PORT=8000
  
```

app/index.js

```

1  const Koa = require('koa')
2  const app = new Koa()
3
4  module.exports = app
  
```

app/config.js

```

1  const dotenv = require('dotenv')
2
3  dotenv.config()
4
5  module.exports = {APP_PORT} = process.env
  
```

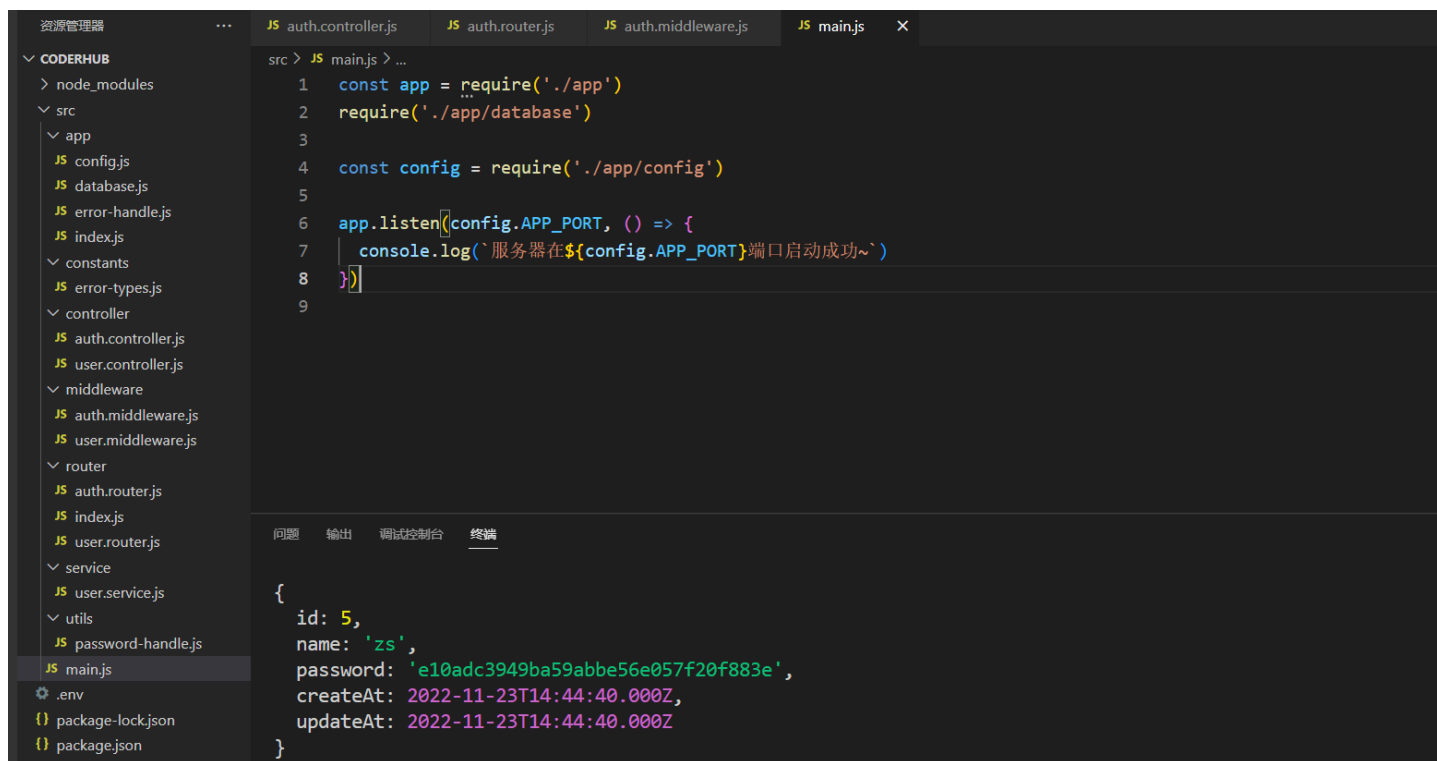
main.js

```

1  const app = require('./app')
2  const config = require('./app/config')
  
```

```
3
4 app.listen(config.APP_PORT, () => {
5   console.log('服务器启动成功')
6 })
```

14.2 用户模块



- 所有和app相关的放在app文件下，比如创建app。app对应的配置信息获取，全局中间件，连接数据库...
- 在app下的index通过调用router目录下的index.js中的方法，进行所有路由的注册。
- constants存放项目所有的常量
- 路由使用controller的控制器处理业务逻辑，对应的middleware用于辅助。比如密码校验，登录校验的中间件放在middleware。
- 所有操作数据库的放在对应的service下。

1.为什么需要登录凭证

web开发中，我们使用最多的协议是http，但是http是一个无状态的协议

- 我们登录了一个网站
- 登录的时候我们需要输入用户名和密码
- 登录成功之后，我们要以用户的身份去访问其他的数据和资源，还是通过http请求去访问

2.认识cookie

- Cookie（复数形态Cookies），又称为“小甜饼”。类型为“小型文本文件，某些网站为了辨别用户身份而存储在用户本地终端（Client Side）上的数据。

- 浏览器会在特定的情况下携带上cookie来发送请求，我们可以通过cookie来获取一些信息；
- Cookie总是保存在客户端中，按在客户端中的存储位置，Cookie可以分为内存Cookie和硬盘Cookie。
 - 内存Cookie由浏览器维护，保存在内存中，浏览器关闭时Cookie就会消失，其存在时间是短暂的
 - 硬盘Cookie保存在硬盘中，有一个过期时间，用户手动清理或者过期时间到时，才会被清理；
- 如果判断一个cookie是内存cookie还是硬盘cookie呢？
 - 没有设置过期时间，默认情况下cookie是内存cookie，在关闭浏览器时会自动删除；
 - 有设置过期时间，并且过期时间不为0或者负数的cookie，是硬盘cookie，需要手动或者到期时，才会删除

客户端设置cookie

```
1 document.cookie = 'name=yihua;max-age=10'
```

服务器设置cookie

```
1 myRouter.get('test',(ctx,next)=>{
2   ctx.cookies.set('name','yihua',{
3     maxAge:1000*60*60
4   })
5   ctx.body = 'cookie设置成功'
6 })
7
8 //获取cookie
9 myRouter.get('test',(ctx,next)=>{
10   const name = ctx.cookies.get('name')
11   ctx.body = name
12 })
```

Session是基于cookie实现机制

```
1 const Session = require('koa-session')
2 // 创建Session的配置
3 const session = Session({
4   key: 'sessionid',
5   maxAge: 10 * 1000,
6   signed: true, // 是否使用加密签名
7 }, app)
8 app.keys = ['aaaa'] //加密使用的key
9 app.use(session)
10 // 登录接口
11 testRouter.get('/test', (ctx, next) => {
12   // 查找数据库匹配成功身份之后存取session
13   ctx.session.user = {id, name}
```

```
14   ctx.body = 'test'
15 })
16
17 testRouter.get('/demo', (ctx, next) => {
18   // 其余接口通过ctx.session.user验证用户信息
19   ctx.body = 'demo'
20 })
```

2.认识token

cookie和session的方式有很多的缺点：

- Cookie会被附加在每个HTTP请求中，所以无形中增加了流量（事实上某些请求是不需要的）；
- Cookie是明文传递的，所以存在安全性的问题；
- Cookie的大小限制是4KB，对于复杂的需求来说是不够的；
- 对于浏览器外的其他客户端（比如iOS、Android），必须手动的设置cookie和session；
- 对于分布式系统和服务器集群中如何可以保证其他系统也可以正确的解析session？

所以，在目前的前后端分离的开发过程中，使用token来进行身份验证的是最多的情况：

- token可以翻译为令牌；
- 也就是在验证了用户账号和密码正确的情况，给用户颁发一个令牌
- 这个令牌作为后续用户访问一些接口或者资源的凭证；
- 我们可以根据这个凭证来判断用户是否有权限来访问；

所以token的使用应该分成两个重要的步骤：

- 生成token：登录的时候，颁发token；
- 验证token：访问某些资源或者接口时，验证token；

3.JWT实现Token机制

JWT生成的Token由三部分组成：

- header
 - alg：采用的加密算法，默认是 HMAC SHA256（HS256），采用同一个密钥进行 加密和解密；
 - typ：JWT，固定值，通常都写成JWT即可；
 - 会通过base64Url算法进行编码；
- payload
 - 携带的数据，比如我们可以将用户的id和name放到payload中；
 - 默认也会携带iat（issued at），令牌的签发时间；
 - 我们也可以设置过期时间：exp（expiration time）；
 - 会通过base64Url算法进行编码
- signature
 - 设置一个secretKey，通过将前两个的结果合并后进行HMACSHA256的算法；

- HMACSHA256(base64Url(header)+base64Url(payload), secretKey);
- 但是如果secretKey暴露是一件非常危险的事情，因为之后就可以模拟颁发token，也可以解密token；

4.token的使用

npm i jsonwebtoken

private.key

```

1  -----BEGIN RSA PRIVATE KEY-----
2  MIICWwIBAAKBgQDD8cZ0CgrectPM2QizGOJRej5M2UkyqZW7Qh8XmZU2vAU/KCzk
3  6CwVAfLGBWoiuQW8fS6j9CiscNg3pAT4EFor4x3X3tcCEjKzJsdnKYkYeWeaFPEc
4  tonMs9MXbCKB01JLk+c1V8jgBX7nAAMZp3njxgnvYP0Yd/xLve/w/GtONQIDAQAB
5  AoGAUDbRBD38NxaQ6EJNEmx0ceB2UqV9FrVf65nk+pdQA2kzSKicwFTffvYeObyL
6  t41A8OnaRxoz8Gv9x8Fom1irt3A3xxWZnhMBNyFjR2eTB1B9hgU2eDt3qKr65L3y
7  Chc5KCH7EzpFWG618PCsLJ2FvMgy1bG/xh1DmUXN3UYL8KECQD/k6EkteF44iR6
8  +uiMmekQI8QU1Q+EyHQKxQqKZcr1GyVSiStFwONeU5loZXeXzgsWfeY0UhoADKkI
9  k/ac7ar5AkEAxETc0ltCgUG6VrM6xXu9GBpfF3Q27mjTH+6miWjlTyIKETUC7AoU
10 VeM3nvPQGCQg/rXb69E048gbnDSbesJwHQJAIGVw6gYcIY9Y89X6ptzGHZPlQjCq
11 hC565AQexxWN0LmwWjBQR1TK73+JaFA5f0e6SiAwSL61H1SQG8g2h+VLYQJAdnDL
12 qde6uX/tsDZ2qAg1I59+dQvnlW52pJNl60OfIOhKaMhAUpP3UjCHwUpNfhPhQZA
13 Ef3WC3WN2+UPUrSVrQJAemOD6Xi0iTuVvto2mYE9I10EMSxms4gl/e7h34VHfoyn
14 3u51hFjmYxINSPoSFIKQuzIp+U2lt6AFLOAJ+B2Y+g==
15 -----END RSA PRIVATE KEY-----
16

```

public.key

```

1  -----BEGIN PUBLIC KEY-----
2  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDD8cZ0CgrectPM2QizGOJRej5M
3  2UkyqZW7Qh8XmZU2vAU/KCzk6CwVAfLGBWoiuQW8fS6j9CiscNg3pAT4EFor4x3X
4  3tcCEjKzJsdnKYkYeWeaFPEctonMs9MXbCKB01JLk+c1V8jgBX7nAAMZp3njxgnv
5  YP0Yd/xLve/w/GtONQIDAQAB
6  -----END PUBLIC KEY-----
7

```

app.js

```

1  const Koa = require('koa');
2  const Router = require('koa-router');
3  const jwt = require('jsonwebtoken');
4  const fs = require('fs');
5  const { pathToFileURL } = require('url');

```

```
6
7  const app = new Koa();
8  const testRouter = new Router();
9
10 // 在项目中的任何一个地方的相对路径，都是相对于process.cwd()
11 console.log(process.cwd());
12
13 const PRIVATE_KEY = fs.readFileSync('./keys/private.key');
14 const PUBLIC_KEY = fs.readFileSync('./keys/public.key');
15
16 // 登录接口
17 testRouter.post('/test', (ctx, next) => {
18   const user = {id: 110, name: 'why'};
19   const token = jwt.sign(user, PRIVATE_KEY, {
20     expiresIn: 10,
21     algorithm: "RS256"
22   });
23
24   ctx.body = token;
25 });
26
27 // 验证接口
28 testRouter.get('/demo', (ctx, next) => {
29   const authorization = ctx.headers.authorization;
30   const token = authorization.replace("Bearer ", "");
31
32   try {
33     const result = jwt.verify(token, PUBLIC_KEY, {
34       algorithms: ["RS256"]
35     });
36     ctx.body = result;
37   } catch (error) {
38     console.log(error.message);
39     ctx.body = "token是无效的~";
40   }
41 });
42
43 app.use(testRouter.routes());
44 app.use(testRouter.allowedMethods());
```

```
45
46 app.listen(8080, () => {
47   console.log("服务器启动成功~");
48 })
```

5.非对称加密

HS256加密算法—单密钥暴露就是非常危险的事情,这个时候我们可以使用非对称加密，RS256。

- 私钥 (private key)：用于发布令牌;
- 公钥 (public key)：用于验证令牌;

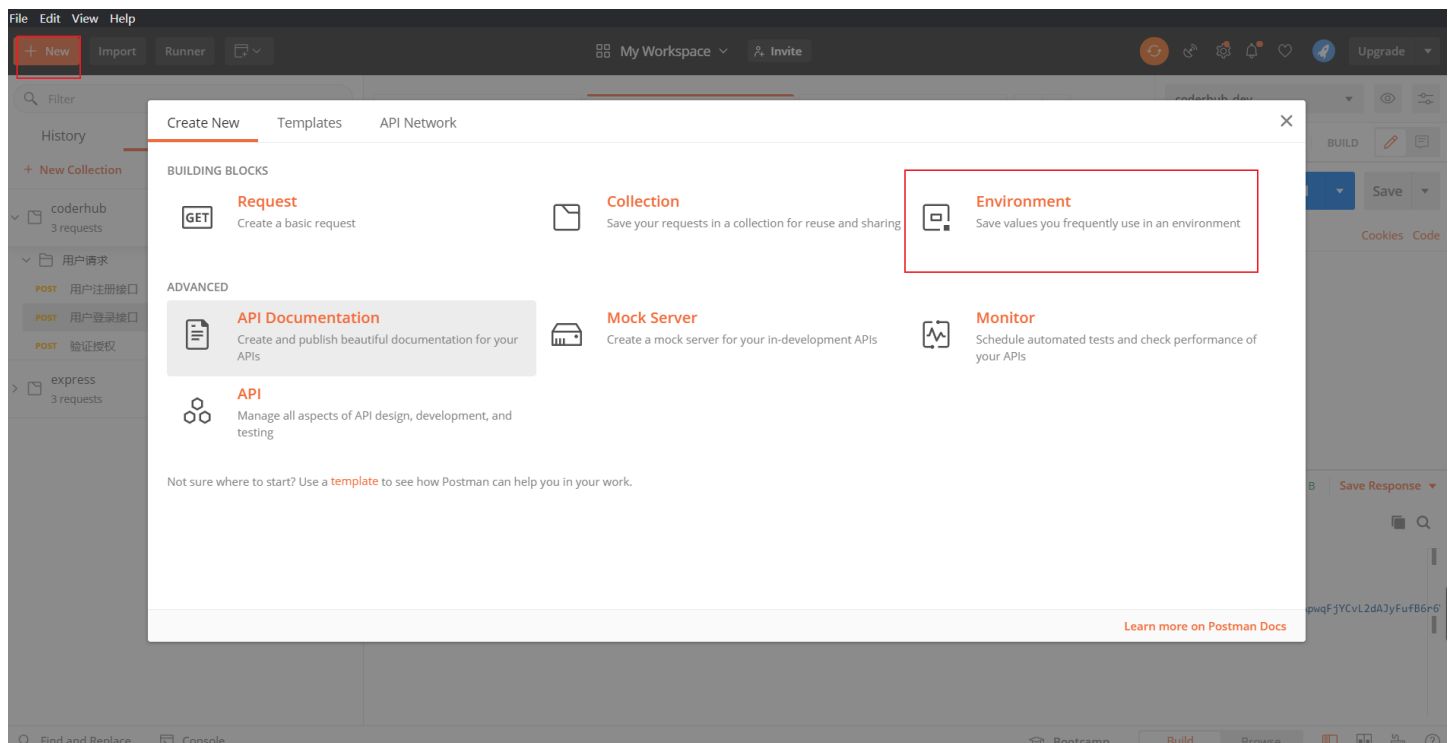
我们可以使用openssl来生成一对私钥和公钥：

- Mac直接使用terminal终端即可;
- Windows默认的cmd终端是不能直接使用的，建议直接使用git bash终端;

在需要生成公钥私钥的文件目录下git bash，输入以下命令。

```
1 openssl
2 genrsa -out private.key 1024
3 rsa -in private.key -pubout -out public.key
```

6.postman使用变量



可以设置baseUrl，然后在用的时候采用{{baseUrl}},

