# Node.js基础

# 目录

# 01 什么是Node.js

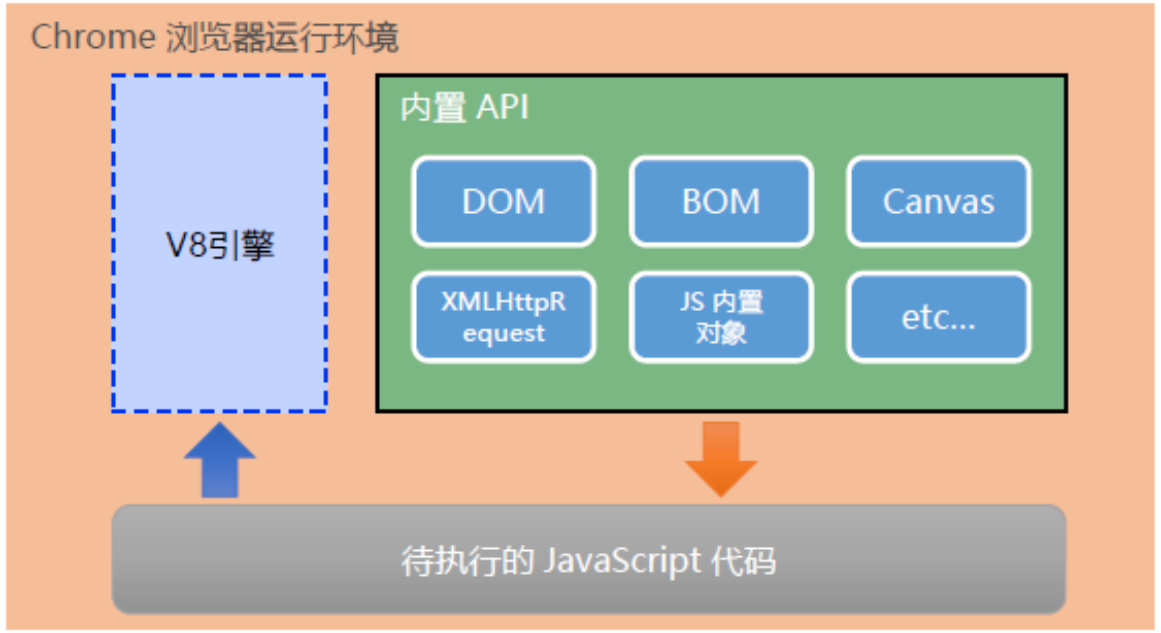
Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine。

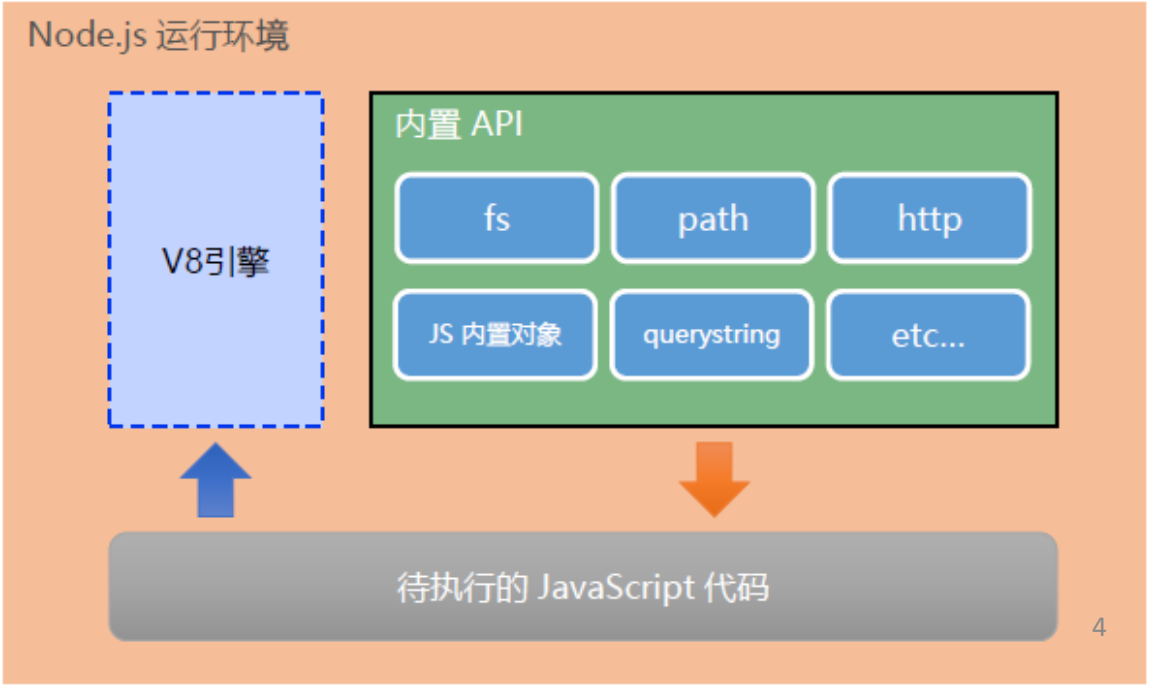
Node.js 是一个基于Chrome V8引擎（Google Chrome内核）的JavaScript运行环境

Node.js 的官网地址：https://nodejs.org

# Node.js 中的 JavaScript 运行环境

1. 浏览器 是 JavaScript 的 前端运行环境 。
2. Node.js 是 JavaScript 的 后端运行环境 。
3. Node.js 中 无法调用 DOM 和 BOM 等浏览器内置 API 。



# 与 JavaScript 的区别

1. 基于异步 I/O 相关接口
2. 基于 node_modules 和 require 的模块依赖
3. 提供 C++ addon API 与系统交互

# Node 的发展历程

2009 年，由 Joyent 的员工 Ryan Dahl 开发而成

Ryan在2012年离开社区

2015年由于 Node 贡献者对 es6 新特性集成问题的分歧，分裂出iojs

iojs 发布1.0、2.0和3.0版本

2015年Node基金会的成立，与iojs 合并，顺利发布了4.0版本

Node.js基金会发展很好，稳定地发布多个大版本

Ryan 在2018年发布了 deno（ https://linux265.com/course/deno-intro.html ）

# Node 特性

1.事件驱动

通过事件驱动能够让Node.js具备承载更多IO的能力；

2.单线程/异步/非阻塞

由JavaScript本身具备单线程/异步/非阻塞 的特性让Node.js能够更高效的去执行请求或者命令；

3.npm（ https://www.npmjs.com/ ）

Node.js 的包管理工具，用来安装各种 Node.js 的扩展，具有非常好的生态。

# Node.js 可以干什么？

1. Web 服务端：Web Server、爬虫

可以基于Express框架，快速构建Web应用

2. CLI 命令行脚本：webpack

读写和操作数据库、创建实用的命令行工具辅助前端开发

3. GUI 客户端软件：VSCode、网易云音乐

基于 Electron框架，可以快速构建跨平台的桌面应用

4. IoT，图像处理, 实时通讯，加密货币...

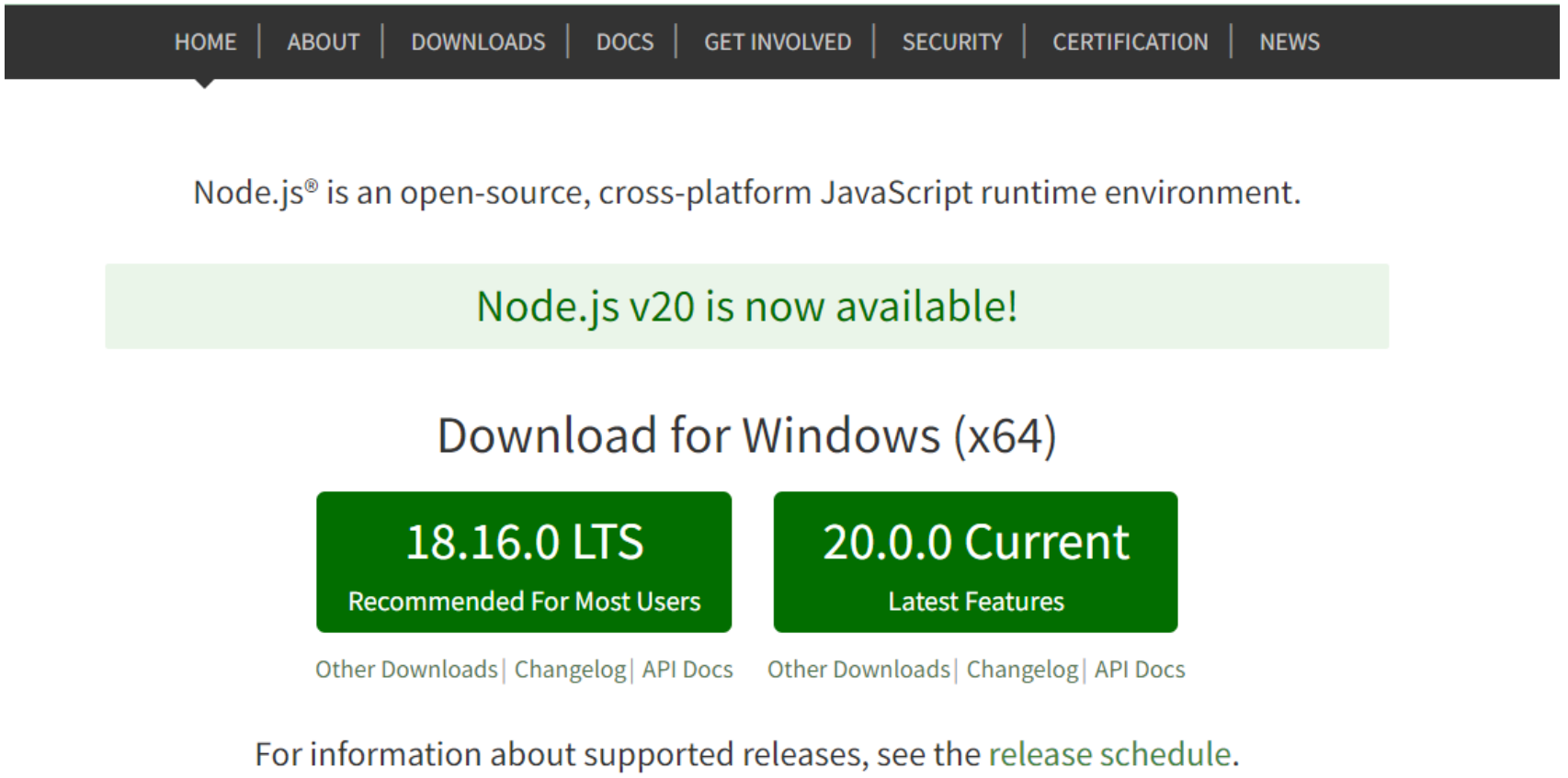
# Node.js 可以干什么？

自动打开浏览器

```
const puppeteer = require('puppeteer')
const puppeteerTest = async () => {
  const user_agent = '--user-agent=Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.183 Safari/537.36'
  const options = {
    //当为true时，客户端不会打开，使用无头模式；为false时，可打开浏览器界面
    headless: false,
    args: ['--no-sandbox', user_agent]
  }
  const browser = await puppeteer.launch(options)
  const page = await browser.newPage()
  await page.setViewport({ width: 1280, height: 900 })
  await page.goto('https://www.baidu.com')
}
puppeteerTest()
```

# 02 Node.js语法

## 安装Node.js



# 使用 Node.js 编写的网络服务器示例



helloworld.js

```
const http = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

# 数据类型

原始数据类型

Number、String、Boolean、Null、Undefined、Symbol

引用数据类型

Object、Array、Function、Map、Set、…

# 对象

```
let obj = {
  a: 1,
  b: {
    c: 'hello',
    d: true
  }
};

console.log(obj.b.c);    //hello

let arr = [1,2,3];
console.log(arr.length); // 3
console.log(arr.map(v=> v * 2));    // [2, 4, 6]
typeof arr; // object
```

# 变量定义



```
1 //var 定义的变量可以重复定义
2 var x = 0;
3 var x = 1;
4
5 //let 定义的变量只允许定义一次
6 let x = 0;
7 let x = 1;
8 // Uncaught SyntaxError:
  // Identifier 'x'
  // has already been declared
```

```
9  // const 定义的变量不可重新赋值
10 const x = 0;
11 x = 1;// Uncaught TypeError: Assignment to constant variable
12
13 // 不允许换行，使用 + 拼接字符串
14 var name = 'Li Lei';
15 var x = 'Hello World ' + name;
16
17 // 使用 `` 模板字符串定义
18 var x = `Hello World ${name}`;
19
20 // 对象
21 var x = {
22   a: 0,
23    b:'hello'
24 }
```

# 条件语句



```
1 const a = '1';
2 if(a) {
3 console.log('hello world');
4 }
5
6 if(a === '1') {
7 console.log('hello world again!');
8 }
9
10 if(a == 1) {
11 console.log('hello world again and again!');
12 }
13
14 console.log(a && 2); // 2
15 console.log(a || 2); // ‘1’
16
```

```
17 switch(a) {
18   case '1':
19     console.log(a);
20     break;
21
22   default:
23     console.log(‘nothing.’);
24     break;
25 }
```

# 循环

```
1 for(let i = 0; i < 10; i++){
2   console.log(i);
3 }
4 let i = 0;
5 while(i < 10) {
6  console.log(i);
7  i++;
8 }
9 let i = 0;
10 do {
11  console.log(i);
12  i++;
13 }while(i<10);
```

```
14 //以下是数组的一些循环方法，本质也是for循环
15 [1,2,3,4].forEach;
16 [1,2,3,4].map;
```

# 函数

```
1 function hello() {
2  console.log('hello');
3 }
4 hello();
5
6 function hello(name) {
7  console.log(`hello ${name}`);
8 }
9 hello('Li Lei');
10
11 function hello(name = 'Han Meimei') {
12  console.log(`hello ${name}`);
13 }
14 hello();
```

# 异步

何为异步？

代码在执行的时候告诉JavaScript，先不要执行这个任务，把任务放到异步队列中，先执行后边的代码，等异步队列有结果返回，再去执行异步队列中的任务。

```
// callback 异步回调
setTimeout(function() {
  console.log('World');
}, 1000);
console.log('Hello');

setTimeout(function() {
  console.log(1);
  setTimeout(function() {
    console.log(2);
  }, 1000);
}, 1000);
```

Hello
World

1
2

# 异步-Promise

Promise 是异步编程的一种解决方案，比传统的回调函数以及事件更合理、强大。简单说就是一个容器，保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上，Promise 是一个对象，从它可以获取异步操作的消息。

通过Promise对象，可以将异步任务以同步操作的流程表达出来，避免层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

# 异步-Promise



```
// Promise
const promise1 = new Promise(function (resolve, reject) {
  setTimeout(function () {
    resolve('hello', 500);
  });
});

promise1.then((value) => {
  // expected output: 'hello'
  console.log(value);
})

console.log(promise1);
```

Promise { <pending> }
hello

# 异步

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

resolve 函数的作用是Promise 对象的状态从“pending”变为“fulfilled”，在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject 函数的作用是，将Promise 对象的状态从“pending”变为“rejected”，在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise 对象的状态变为resolved时调用，第二个回调函数是Promise 对象的状态变为rejected时调用。其中，第二个函数是可选的，不一定要提供。

# 异步—Promise对象then链式调用



```
15  // setTimeout
16  function timeout(time) {
17      return new Promise(function (resolve) {
18          setTimeout(resolve, time);
19      });
20  }
21  timeout(1000).then(function () {
22      console.log(1);
23  }).then(function () {
24      timeout(1000);
25  }).then(function () {
26      console.log(2);
27  });
```

1
2

# 异步—async和await

```
30  //·setTimeout
31  async·function·timeout(time)·{
32  ··return·new·Promise(function·(resolve)·{
33  ····setTimeout(resolve,·time);
34  ··});
35  }
36  (async·()·=>·{
37  ··await·timeout(1000);
38  ··console.log(1);
39  ··await·timeout(1000);
40  ··console.log(2);
41  })();
```

- async 用于声明一个 function 是异步的，异步函数也就意味着该函数的执行不会阻塞后面代码的执行。await用于等待一个异步方法执行完成。
- async返回的是一个Promise对象，await就是等待这个promise的返回结果后，再继续执行。
- await 关键字只能放到async 函数里面。

1
2

# 异步



```
setTimeout(function () {
    console.log(1);
    setTimeout(function () {
        console.log(2);
        setTimeout(function () {
            console.log(3);
        }, 1000);
    }, 1000);
}, 1000);
```

```
15  // setTimeout
16  function timeout(time) {
17      return new Promise(function (resolve) {
18          setTimeout(resolve, time);
19      });
20  }
21  timeout(1000).then(function () {
22      console.log(1);
23  }).then(function () {
24      timeout(1000);
25  }).then(function () {
26      console.log(2);
27  });
```

```
30  // setTimeout
31  async function timeout(time) {
32      return new Promise(function (resolve) {
33          setTimeout(resolve, time);
34      });
35  }
36  (async () => {
37      await timeout(1000);
38      console.log(1);
39      await timeout(1000);
40      console.log(2);
41  })();
```

# 03 fs模块

Node.js中的内置模块：fs、path、http、url模块等

**fs模块**是 Node.js 官方提供的、用来操作文件的模块。使用时需要先导入。

```
const fs = require('fs');
```

语法: readFile( 路径， [格式]， 回调函数)

- 路径：要读取的文件路径
- 格式：表示以什么编码读取文件
- 回调函数：读取成功后执行的回调函数

```
const fs = require('fs');

fs.readFile('./a.txt','utf8', function(err,data) {
  if(err) return console.log(err)
  console.log('Loading finished')
  console.log(data)
})
```

# 03 fs模块

向指定的文件中写内容

语法: writeFile( 路径， 内容，[格式]， 回调函数)

- 路径：表示文件的存放路径。
- 内容：表示要写入的内容。
- 格式：以什么格式写入文件，默认值utf8
- 回调函数：文件写入完成后执行的回调函数

# 03 fs模块

Node.js内置模块

```
const fs = require('fs');

fs.writeFile('./b.txt', 'hello world', (err) => {
    if (err)
        console.log('文件写入失败' + err.message);
    console.log('写入内容完成');
})
```

# 03 path模块



操作和路径相关的内容，内置模块，直接导入就可以使用

方法：

extname()：专门获取一个路径中的后缀名

语法： path.extname('abc.html') //输出.html

isAbsolute()：判定是不是绝对路径

语法： path.isAbsolute('/test') //true

- 相对路径和绝对路径

  从根目录出发叫绝对路径

  和当前目录有关系的路径, 叫做相对路径

# 03 path模块



__dirname

当前模块的目录名。 与 path.dirname(__filename ) 含义相同。

假设从 /Users/mjr 运行 node example.js：

```
console.log(__dirname);

// 打印: /Users/mjr

console.log(path.dirname(__filename));

// 打印: /Users/mjr
```

# 03 path模块



join()：多个参数直接拼成相对路径；

语法: path.join(‘address1’, ‘address2’, ‘address3’)

例如：

```
const res = path.join('/a','/b','/c‘,'./d','../e')
console.log(res) //输出 /a/b/c/e
```

resolve()：多个参数直接拼成绝对路径；

语法: path.resolve(‘address1’, ‘address2’, ‘address3’)

# 03 path模块

basename():获取路径中的最后一部分，即获取路径中的文件名。

语法： path.basename(path,[ext]);

path：表示必选参数，表示一个路径的字符串

ext ： 可选参数，表示文件扩展名

```
const path = require('path');

const fullname = path.basename('./a.txt');
console.log(fullname)
const nameWithoutExt = path.basename('./a.txt', '.txt')
console.log(nameWithoutExt)
```

a.txt
a

# 03 path模块

parse()：解析一个路径, 成为一个对象

语法: path.parse()

返回值: 一个对象, 里面包含地址的所有信息

```
const path = require('path');

const res = path.parse( 'D:\\B\\Notes\\JS\\a\\c\\a.js')
console.log(res)
```

```
{
  root: 'D:\\',
  dir: 'D:\\B\\Notes\\JS\\a\\c',
  base: 'a.js',
  ext: '.js',
  name: 'a'
}
```

# url模块

Node.js内置模块

专门操作 url 地址的模块

直接引入使用

parse()：解析 url 地址

语法: url.parse(url地址[, 是否解析query]) 默认false

返回值: 一个对象, 包含整个 url 地址的所有信息

```
Url {
  protocol: 'http:',
  slashes: true,
  auth: null,
  host: 'nodejs.cn',
  port: null,
  hostname: 'nodejs.cn',
  hash: '#xyz',
  search: '?name=lly&num=100',
  query: 'name=lly&num=100',
  pathname: '/eslint/',
  path: '/eslint/?name=lly&num=100',
  href: 'http://nodejs.cn/eslint/?name=lly&num=100#xyz'
}
```

```
const url = require('url')
const str="http://nodejs.cn/eslint/?name=lly&num=100#xyz"
const res = url.parse(str)
console.log(res)
```

# http模块

Node.js内置模块

客户端：在网络节点中，负责消费资源的节点

服务器：负责对外提供网络资源的节点

http模块是Node.js提供用来创建web服务器的模块。通过 http 模块提供的 http.createServer () 方法，就能方便的把一台普通的电脑，变成一台 Web 服务器，从而对外提供 Web 资源服务。

# http模块

Node.js内置模块

createServer()：专门用来创建 http 服务的方法

语法: http.createServer(函数)

返回值: 一个 http 服务

listen()：监听某一个端口使用的方法

语法: 服务.listen(端口号, 回调函数)

```
const http = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

# http模块



当监听端口号完毕的时候，在 cmd 里面运行起来代码，cmd 窗口就因为这段代码变成了一个服务器。此时客户端请求 localhost:8080 的时候，每一个请求都会触发一次 createServer 的参数函数，这个函数接收两个参数：

request: 本次请求的所有请求信息

1. 服务器解析完请求报文后组装的内容
2. 需要从请求报文里获取什么信息
3. 直接在request 里面找

response: 将来组装成响应报文的东西

1. 想要在响应报文里添加什么，就往这个response 里面的指定位置添加

# http模块

创建web服务器的步骤可以概括为：

1. 导入http 模块
2. 创建web 服务器实例
3. 为服务器实例绑定**request**事件，监听客户端的请求
4. 启动服务器

需注意中文乱码问题：

当调用res.end () 方法，向客户端发送中文内容的时候，会出现乱码问题，此时，需要手动设置内容的编码格式。

通过设置响应头Content-Type ， res.setHeader(‘Content-Type’, ’text/html; charset=utf-8’);

# http模块

根据不同的url响应不同的html内容

1. 获取 请求的 url 地址
2. 设置 默认的响应内容 为 “404 Not found ”
3. 判断用户请求的是否为 / 或 /index.html 首页
4. 判断用户请求的是否为 /about.html 关于页面
5. 设置 Content Type 响应头 ， 防止中文乱码
6. 使用 res.end () 把内容响应给客户端

# http模块

Node.js内置模块

```
const http = require('http');
const server = http.createServer();
server.on('request', function (req,res) {
  const url = req.url;
  let content = '<h1>404 Not Found</h1>';
  if (url === '/' || url === '/index.html') {
    content='<h1>首页</h1>'
  } else if (url === '/help.html') {
    content='<h1>帮助</h1>'
  }
  res.setHeader('Content-Type', 'text/html; charset=utf-8');
  res.end(content);
})
server.listen(8080, () => {
  console.log('server running at port 8080!');
})
```

# 模块化

模块化是指解决一个复杂问题时，自顶向下逐层把系统划分成若干模块的过程。对于整个系统来说，模块是可组合、分解和更换的单元。

对于编程来说，模块化就是遵守一定规则（模块化规范），把一个大文件拆成独立并互相依赖的多个小模块。

优势：

1. 提高了代码的复用性
2. 提高了代码的可维护性
3. 可以实现按需加载