nodejs讲义

从前端到后端；扩展前端技能栈；

安装并使用nodejs

安装nodejs

下载

官网： https://Nodejs.org/en/ 中文网：http://Nodejs.cn/

- 版本说明
  - **LTS**： 长期稳定版(Long Term Support))。 项目开发建议使用长期稳定版
  - **Current**： 最新版。最新版包含了一些新功能，如果想学习最新的功能，则可以使用该版本。最新版可能会有一些未知的bug。

安装

1) 双击安装文件开始安装（不同系统选择对应的安装文件）

2) 傻瓜式安装，一路 'next' 即可

注意：

- 建议安装目录所使用 `英文路径`

测试是否安装成功

打开任意一个**小黑窗**，输入`node -v`能够看到Nodejs版本号即为安装成功。

- cmd窗口(window+R, --->运行-->录入cmd,回车)
- powershell（window10操作系统）

在node环境下运行js代码

我们前面的学习中，js代码都是在浏览器中运行的，现在开始学习nodejs后，我们有了第二个环境中可以运行js代码。

有两种方式可以运行js代码：

- 在nodejs 提供的repl中环境
- 单独执行外部的js文件

方法1：在 REPL中运行（了解）

REPL(Read Eval Print Loop:交互式解释器) 表示一个电脑的环境，类似 Window 系统的终端或 Unix/Linux shell，我们可以在终端中输入命令，并接收系统的响应。

Node 自带了交互式解释器，可以执行以下任务：

- **读取** - 读取用户输入，解析输入了Javascript 数据结构并存储在内存中。
- **执行** - 执行输入的数据结构
- **打印** - 输出结果
- **循环** - 循环操作以上步骤直到用户两次按下 **ctrl-c** 按钮退出。

具体操作：

1. 在任意控制台中输入node 并回车确定，即可进行入node自带的REPL环境。
2. 此时，你可以正常写入js代码，并执行。
3. 如果要退出，**连续按下两次ctrl+c**

方法2：执行一个JS文件（常用）

1. 请事先准备好一个js文件。
    - 例设这的路径是：e:/index.js
    - 具体内容是

```
var a = 1;
console.info(a + 2);
```

2. 打开小黑窗，进入到这个文件的目录
    - 技巧，在资源管理器中按下shift，同时点击鼠标右键，可以选择在此处打开powershell窗口。
    - cd 命令可以用来切换当前目录。
3. 接下来 通过 `node js文件的路径` 的格式来执行这个js文件。

```
node index.js
```

注意:

- 执行js文件时，如果当前命令行目录和js文件**不在**同一个盘符下，要先**切换盘符**
    - 切换方式，输入`盘符:`并回车
- 如果当前命令行目录和js文件**在**同一个盘符中，则可以使用**相对路径**找到js文件并执行

nodejs的helloworld程序

下面，我们来通过一个最基本的http服务器程序来见识nodejs的作用。

第一步：新建一个文件，名为 `d:/http.js`(文件名及路径名可以自行设置，建议均不使用中文字符)

第二步：在文件中录入如下代码。

```
// 引入http模块
const http = require('http');

// 创建服务
const server = http.createServer(function(req, res) {
  console.log(`来自${req.connection.remoteAddress}的客户端在${new
Date().toLocaleTimeString()}访问了本服务器`);
  res.end('<h1>hello world! very good!!</h1> <p>' + req.connection.remoteAddress + '</p>');
});
// 启动服务
server.listen(8081, function() {
  console.log('服务器启动成功，请在http://localhost:8081中访问....');
});
```

第三步：在小黑窗中进入到d盘根目录，键入命令 `node http.js`

第四步：打开一个浏览器容器，输入'http://localhost:8081'，观察效果

第五步：把localhost改成你自己电脑的ip地址，再把这个路径发你的同学来访问。

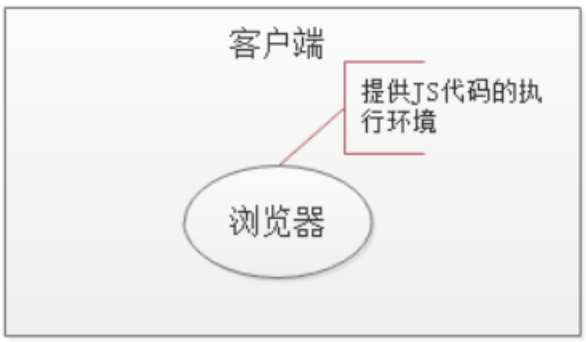
- 如果不能访问，有可能你需要手动关闭你自己计算机的防火墙。

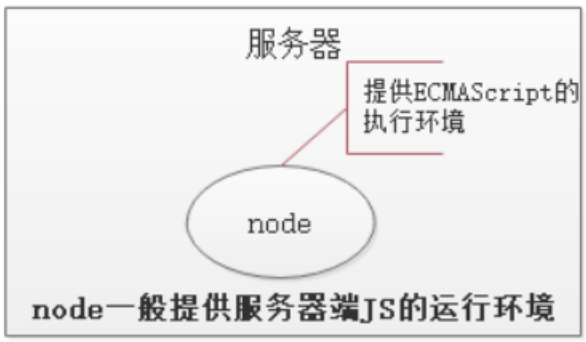
node.js基本介绍

node.js是什么

> Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.
>
> ---
>
> Node.js® 是一个基于 Chrome V8 引擎 的 JavaScript 运行时

- Node全名是Node.js，但它不是一个js文件，而是一个软件
- Node.js是一个**基于Chrome V8引擎的ECMAScript的运行环境**，在这个环境中可以执行js代码
- Node.js提供了大量的工具（API），能够让我们完成文件读写、Web服务器创建等功能。





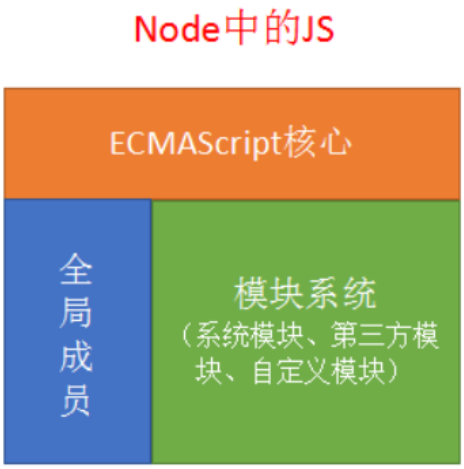
nodejs和浏览器和javascript的关系

nodejs和浏览器的关系

相同之处：

- 都可以运行js(严格来讲是ECMAScript)代码

不同之处：



- 安装了浏览器这个软件，它不但可以执行**ECMAScript**，浏览器这个软件内置了window对象，所以浏览器有处理**DOM和BOM**的能力。
- 安装了NodeJs这个软件，它不但可以执行**ECMAScript**，NodeJS这个软件也内置了一些功能，包括**全局成员和模块系统**，同时还可以载**入第三方模块**来完成更强大的功能。

nodejs和javascript的区别?

- nodejs是一个容器（不是一个新语言），ECAMScript程序可以在这个容器中运行。
  - 不能在nodejs使用window对象，也不能在nodejs使用dom操作。因为nodejs中并不包含这个对象。
- javascript是由三个部分组成：ECMAScript,BOM,DOM

学习Nodejs的意义

在我们熟悉的浏览器上执行JS不是很好吗？为什么要学习Nodejs呢？主要原因：

- 大前端必备技能
- 使得JS能够和操作系统 “互动” （读取文件，写入文件等，管理进程）
- 为JavaScript提供了服务端编程的能力
  - 文件IO
  - 网络IO
  - 数据库
- **了解接口开发，进一步理解Web开发，了解后端同学的工作内容**

学习的主要内容

模块

nodejs中的模块化

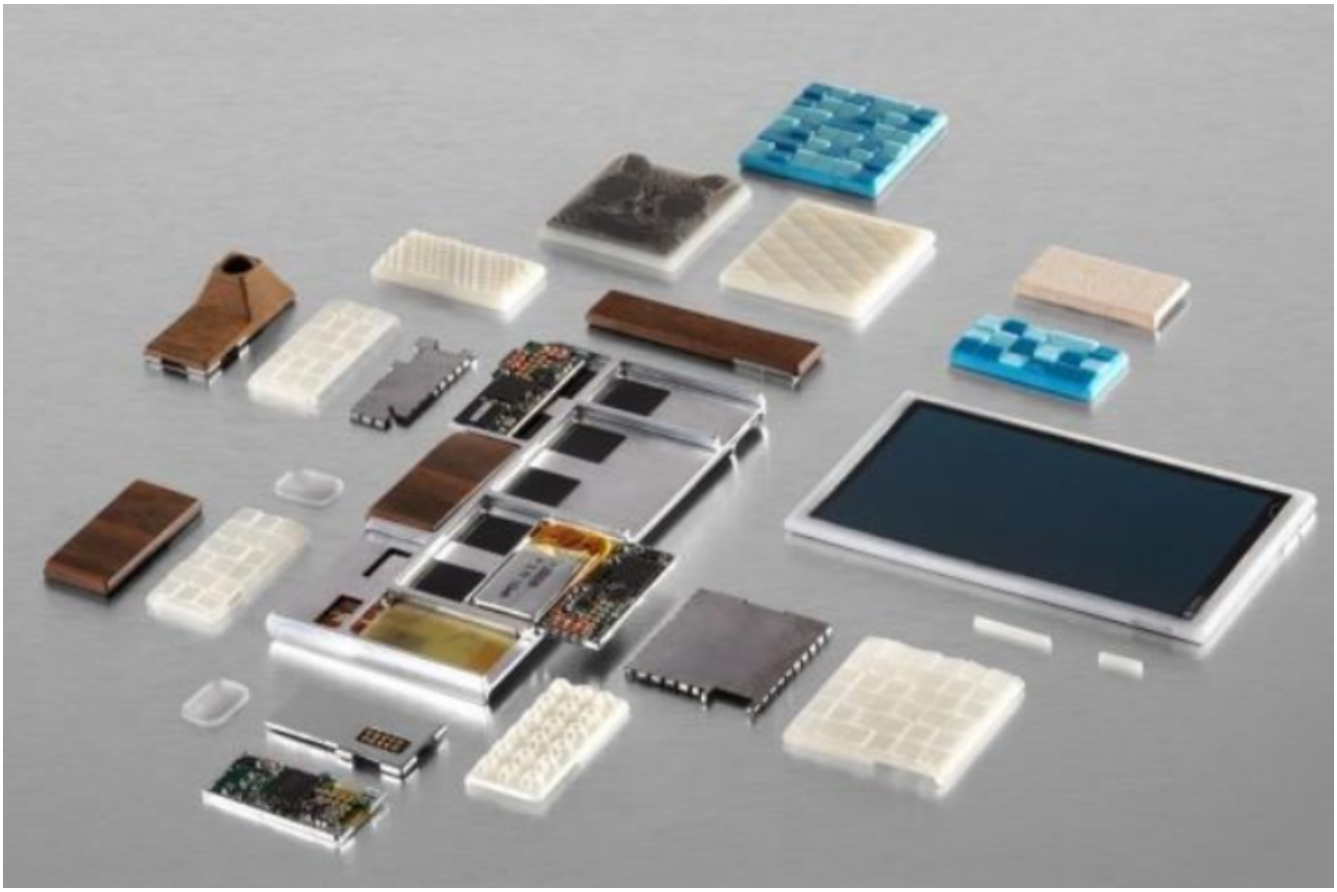
在项目的开发过程中，随着功能的不断增强，代码量，文件数量也急剧增加，我们需要把一个大函数拆成小函数，把一个大文件拆成小文件，把一个大功能拆成若干个小功能。这里很自然地就涉及到模块化的想法：一个复杂的系统分成几个子系统，体现在几个小的文件在一起组成一个大的文件，集成强大的功能。

遗憾的是es5不支持模块化：就是在一个js文件内不能引入其他js文件。不能通过一个大文件去集成若干个小文件。（不是说一个html文件中不能包含多个js文件）。

```
<script src="https://ecpm.tanx.com/ex?i=mm_12852562_1778064_13672849&cb=jsonp_callba…
1e8e2f48d7c750f88b&u=https%3A%2F%2Fwww.taobao.com%2F&psl=1&nk=&sk=&refpid=" async></script>
<script charset="utf-8" src="https://tce.taobao.com/api/mget.htm?callback=jsonpXctrl318&tce_sid=318…
&tce_vid=2&tid=&tab=&topic=&count=&env=online&cna=OqRtFR6JDAoCAT0wjJvHbxEr" async></script>
<script src="https://ecpm.tanx.com/ex?i=mm_12852562_1778064_13674396&cb=jsonp_callba…
1e8e2f48d7c750f88b&u=https%3A%2F%2Fwww.taobao.com%2F&psl=1&nk=&sk=&refpid=" async></script>
<script src="https://ecpm.tanx.com/ex?i=mm_12852562_1778064_48458242&cb=jsonp_callba…
1e8e2f48d7c750f88b&u=https%3A%2F%2Fwww.taobao.com%2F&psl=1&nk=&sk=&refpid=" async></script>
<script src="https://ecpm.tanx.com/ex?i=mm_12852562_1778064_48432768&cb=jsonp_callba…
1e8e2f48d7c750f88b&u=https%3A%2F%2Fwww.taobao.com%2F&psl=1&nk=&sk=&refpid=" async></script>
<script src="https://ecpm.tanx.com/ex?i=mm_12852562_1778064_95126332&cb=jsonp_callba…
1e8e2f48d7c750f88b&u=https%3A%2F%2Fwww.taobao.com%2F&psl=1&nk=&sk=&refpid=" async></script>
<script src="https://ecpm.tanx.com/ex?i=mm_12852562_1778064_13670999&cb=jsonp_callba…
1e8e2f48d7c750f88b&u=https%3A%2F%2Fwww.taobao.com%2F&psl=1&nk=&sk=&refpid=" async></script>
<script charset="gbk" src="//g.alicdn.com/mm/tanx-cdn/t/tanxssp.js?v=2" async></script>
<script src="//g.alicdn.com/??secdev/entry/index.js,alilog/oneplus/entry.js" async></script>
<script src="https://suggest.taobao.com/sug?
_ksTS=1565775770416_254&callback=jsonp255&area=shade_recommand&code=utf-8" async></script>
<script charset="utf-8" src="https://g.alicdn.com/kg/??component/6.2.3/extension/content-box/xtpl/
view.xtpl-min.js" async></script>
```

这样就会带来多个问题：

1. 文件的加载先后顺序
2. 不同的文件内部定义的变量共享



模块化

一个js文件中可以引入其他的js文件，能使用引入的js文件的中的变量、数据，这种特性就称为模块化。使用模块化开发可以很好的解决变量、函数名冲突问题，也能灵活的解决文件依赖问题。

- 以前

  es5不支持模块化，让前端人员很为难。为了让支持模块化，我们一般会借用第三方库来实现：

  - sea.js. https://www.zhangxinxu.com/sp/seajs/
  - require.js. https://requirejs.org/
- 现在
  - es6原生语法也支持模块化
  - Nodejs内部也支持模块化（与es6的模块化有些不同之处），具体的语法在后面来介绍

nodejs中的模块

每个模块都是一个独立的文件。每个模块都可以完成特定的功能，我们需要时就去引入它们，并调用。不需要时也不需要要管它。（理解于浏览器的js中的Math对象）

nodejs模块的分类

- **核心模块**
  - 就是nodejs自带的模块，在安装完nodejs之后，就可以随意使用啦。相当于学习js时使用的内置对象。
  - 全部模块的源代码 https://github.com/nodejs/node/tree/master/lib
- **自定义模块**
  - 程序员自己写的模块。就相当于我们在学习js时的自定义函数。
- **第三方模块**
  - 其他程序员写好的模块。nodejs生态提供了一个专门的工具来管理第三方模块，后面我们会专门讲到。
  - 相当于别人写好的函数或者库。例如我们前面学习的jQuery库，artTemplate等。

核心模块

> 官网文档 https://nodejs.org/dist/latest-v10.x/docs/api/
>
> 中文文档 http://nodejs.cn/api/
>
> **学会查 API，远远比会几个 API 更重要**

- 核心模块就是 Node 内置的模块，需要通过唯一的标识名称来进行获取。
- 每一个核心模块基本上都是暴露了一个对象，里面包含一些方法供我们使用
- 一般在加载核心模块的时候，变量的起名最好就和核心模块的标识名同名即可
  - 例如：`const fs = require('fs')`

示例：用fs模块读取文件

```
const fs = require('fs');
let htmlStr = fs.readFileSync( 'index.html')).toString();
console.log(htmlStr)
```

注意：**require()中直接写模块的名字**

- 不要加.js
- 不要加其它路径

fs模块

fs模块是文件操作模块。fs是 FileSystem的简写。它用来对文件，文件夹进行操作。

手册：http://nodejs.cn/api/fs.html

```
// 引入模块。
// 可以使用var、let，但是建议使用const，因为我们不希望它被改变
const fs = require('fs');
```

fs模块中操作文件(或者文件夹)的方法，大多都提供了两种选择：

- 同步版本的
- 异步版本的

文件内容读取 - readFile

异步格式

```
fs.readFile('文件路径'[,选项], (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

说明：

- 第一个参数：文件路径。 相对路径和绝对路径均可。
- 第二个参数： 配置项，可选参数。主要用来配置字符集。一般可设置为'utf8'

如果不设置该参数，文件内容会以二进制形式返回。

- 参数3: 读取完成后触发的回调函数。有两个参数 --- err 和 data
  - 读取成功
    - err: null
    - data: 文件内容，如果不设置参数2,则返回二进制数据。可以使用 toString() 方法将二进制数据转为正常字符串
  - 读取失败
    - err: 错误对象
    - data: undefined

示例：

```
const fs = require("fs")
fs.readFile('文件路径',"utf8", (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

同步格式

与异步格式不同在于：

- api的名字后面有Sync
- 不是通过回调函数来获取值，而是像一个普通的函数调用一样，直接获取返回值

```
const fs = require("fs")
let rs = fs.readFileSync('文件路径',"utf8");
console.log(rs)
```

文件写入

覆盖写入 writeFile

功能：向指定文件中写入字符串（覆盖写入）， 如果没有该文件则尝试创建该文件。它把把文件中的内容全部删除，再填入新的内容。

格式：`fs.writeFile(var1, var2, var3, var4);`

参数1: 要写入的文件路径 --- 相对路径和绝对路径均可，推荐使用绝对路径

参数2: 要写入文件的字符串

参数3: 配置项，设置写入的字符集，默认utf-8

参数4: 写入完成后触发的回调函数，有一个参数 --- err （错误对象）

```
const fs = require('fs')
fs.writeFile('./a.txt', 'hello world niahi \n asfsdf', err => {
  if (err) {
    console.info(err)
    throw err
  }
})
```

文件追加 appendFile

功能： 向指定文件中写入字符串（追加写入）， 如果没有该文件则尝试创建该文件

格式：`fs.appendFile(var1, var2, var3, var4);`

参数1: 要写入的文件路径 --- 相对路径和绝对路径均可，推荐使用绝对路径

参数2: 要写入文件的字符串

参数3: 配置项，设置写入的字符集，默认utf-8

参数4: 写入完成后触发的回调函数，有一个参数 --- err （错误对象）

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

fs.appendFile('./a.txt', '\n 为天地立命', err => {
  if (err) {
    console.info(err)
    throw err
  }
})
```

路径问题

在读取文件时，写相对路径是容易出问题的。下面我们来看会出什么问题。

假设有如下两个文件，它们所处的目录及文件名如下所示：

```
day01/js/fs.js
day01/js/text.txt
```

fs.js代码的作用是读出text.txt中的内容，并显示出来。

```
const fs = require('fs');
fs.readfilesync("./text.txt",'utf8');
//注意这里对text.txt的访问使用的是相对"fs.js" 本身的路径
```

现在，我们想要运行fs.js这个文件有两种方式：

- 如果终端中的路径定位在`js目录`下，则通过`node fs.js`
- 如果终端中的路径定位在`day01目录`下，则通过：`node js/fs.js`

  此时就不能正确找到文件了。

原因是：我们在fs中读取文件时，使用的是相对路径，而相对路径的参考点是运行这个js文件的小黑窗的路径。而这个路径我们是可以通过cd命名来调整的。

解决方法，就是在操作文件时，使用**绝对路径**来定位文件。

__dirname __filename 获取绝对路径

绝对路径： 从磁盘根目录开始到指定文件的路径。

相对路径：是以某个文件的位置为起点，相对于这个位置来找另一个文件。

nodejs中提供了两个全局变量来获取获取绝对路径：

- __dirname：获取当前被执行的文件的文件夹所处的绝对路径
- __filename：获取当前被执行的文件的绝对路径

path模块

http://nodejs.cn/api/path.html#path_path

它是一个核心模块，用来处理路径问题：拼接，分析，取后缀名。

- path.basename（路径）获取
- **path.join()** （常用）
- path.parse(path) 转成一个对象

```
path.basename('/foo/bar/baz/asdf/quux.html');// 返回: 'quux.html'
path.basename('/foo/bar/baz/asdf/quux.html', '.html');// 返回: 'quux'
path.dirname('/foo/bar/baz/asdf/quux');// 返回: '/foo/bar/baz/asdf'
path.extname('index.html');// 返回: '.html'
```

附：fs模块中的常用方法

API	作用	备注
fs.access(path, callback)	判断路径是否存在	
fs.appendFile(file, data, callback)	向文件中追加内容	
fs.copyFile(src, callback)	复制文件	
fs.mkdir(path, callback)	创建目录	
fs.readDir(path, callback)	读取目录列表	
fs.rename(oldPath, newPath, callback)	重命名文件/目录	
fs.rmdir(path, callback)	删除目录	只能删除空目录
fs.stat(path, callback)	获取文件/目录信息	
fs.unlink(path, callback)	删除文件	
fs.watch(filename[, options][, listener])	监视文件/目录	
fs.watchFile(filename[, options], listener)	监视文件	
fs.existsSync(absolutePath)	判断路径是否存在	

附：path模块其它方法列表

方法	作用
path.basename(path[, ext])	获取返回 path 的最后一部分(文件名)
path.dirname(path)	返回目录名
path.extname(path)	返回路径中文件的扩展名(包含.)
path.format(pathObject)	将一个对象格式化为一个路径字符串
path.join([...paths])	拼接路径
path.parse(path)	把路径字符串解析成对象的格式
path.resolve([...paths])	基于当前工作目录拼接路径

http模块-基本使用

http是nodejs的核心模块，让我们能够通过简单的代码创建一个Web服务器，处理http请求。

快速搭建Web服务器

1. 新建文件，写入如下代码。

```
// http.js
// 引入核心模块http
const http = require('http');

// 创建服务
const server = http.createServer(function(req, res) {
  console.log(req.connection.remoteAddress);
  res.end('hello world');
});
// 启动服务
server.listen(8081, function() {
  console.log('success');
});
```

2. 运行代码，`node http.js`
3. 在浏览器地址栏中输入：localhost:8081 观察效果。

说明

1. 把localhost改成本机ip地址，让同一局域网的同学访问。
2. 如果你修改了代码，必须先停止服务，然后再启动。这样才能生效。

   ctrl+c 停止服务。
3. 更改res.end()的内容，`重启`后，再次观察。
   - 获取ip，返回给浏览器

理解请求与响应

在上面的代码中，我们通过http.createServer方法创建一个http服务。

```
// 创建服务
const server = http.createServer((req, res) => {
  console.log(req.connection.remoteAddress);
  res.end('hello world');
});
```

其中的参数是一个函数，这个函数是一个匿名函数，这个函数充当回调函数的角色，当有http请求时，它会自动被调用。

这个回调函数有它有两个参数。这两个参数非常重要，也非常复杂.

- 第一个参数表示`来自客户端浏览器的请求`，第二个参数用来`设置对本次请求的响应`。它们的形参名并不重要，但是，一般第一个参数名使用req或者request表示，第二个参数使用res或者resposne表示。
- 当某个客户端来请求这个服务器时，这个函数会自动调用，同时会自动给这两个参数赋值。第一个参数中包括本次请求的信息。
  - req：请求
    - **req.url**。本次请求的地址
    - **req.method**。 获取请求行中的请求方法
    - **req.headers**。 获取请求头
- 第二个参数用来设置本服务器对这次请求的处理。
  - 这个参数一般命名是res，它是一个对象，其中有很多方法和属性。
  - **res.end()**
    - 把把本次的处理结果返回给客户端浏览器
    - 如果不写这一句，则客户端浏览器永远收不到响应。
  - **res.setHeader()** 设置响应头，比如设置响应体的编码

    `res.setHeader('content-type', 'text/html;charset=utf-8');`
  - **res.statusCode** 设置状态码

根据不同 url 地址处理不同请求

前面已经可以对浏览器的请求做出响应了，但是响应的内容总是一样的。能不能根据url的不同，做出合适的响应呢？当然可以，那么首先就需要知道浏览器请求的url是什么。

涉及到和请求相关的信息，都是通过请求响应处理函数的第一个参数完成的。代码示例

```
// http.js
// 引入核心模块http
const http = require('http');
```

```
// 创建服务
const server = http.createServer(function(req, res) {
  if(req.url === "/a.html"){
      // 读出文件内容
      // 通过res.end()返回
  }
  else if(req.url === "/b.html"){

  }
    else{
        res.end("");
    }
});
// 启动服务
server.listen(8081, function() {
  console.log('success');
});
```

res.setHeader('content-type', 'text/html;charset=utf-8');

使用 nodemon来自动重启http服务

我们每次修改了代码，都需要重启http服务器:

1. 进入控制台
2. 按下ctrl+c，停止已有http服务器。
3. 手动运行：node index.js 来重启服务器。

这会很麻烦。

有没有一个工具会自动检测到我们的修改并自动重新运行我们的代码呢?

有，它叫nodemon。https://www.npmjs.com/package/nodemon

安装 nodemon

通过npm包管理工具来进行安装。任意打开一个小黑窗，输入如下命令

```
npm install -g nodemon
```

此操作`需要联网`，根据网络速度所耗时间不同。

- npm是一个工具。用来管理node代码中要使用的第三方模块。它是随着node的安装而自动安装的：如果你安装node，则npm也已经安装过了，你可以直接使用。

使用nodemon

等待安装成功之后，使用方法也非常简单：在命令中，使用nodemon来代替node。

例如，

```
node server.js  //
// 改成 nodemon server.js
nodemon server.js
```

它的好处在于会自动监听server.js这个文件的变化，如果变化了，就会重新自动再去运行。

说明：

- 它是一个第三方的包（其它开发者写的工具），我们这里是通过全局安装的方式进行。

http模块-处理静态资源

静态资源指的是html文件中链接的外部资源，如css、js、image文件等等。

处理二次请求

从服务器获取html文件之后，如果这个html文件中还引用了其它的外部资源（图片，样式文件等），则浏览器会重新再发请求。

假设在index.html中还引入了 style.css 1.png 或者 .js文件，则：

浏览器请求localhost:index.html之后，得到的从服务器反馈的内容，解析的过程中还发现有外部的资源，所以浏览器会再次发出第二次请求，再去请求相应的资源。

一个最朴素的想法是根据不同的请求来返回不同的文件。

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

//创建服务器
const app = http.createServer((req, res) => {

  if (req.url === '/index.html') {
    let htmlString = fs.readFileSync(path.join(__dirname, 'index.html'));
    res.end(htmlString);
  }
  else if (req.url === '/style.css') {
    let cssString = fs.readFileSync(path.join(__dirname, 'style.css'));
    res.setHeader('content-type', 'text/css');
    res.end(cssString);
  } else if (req.url === '/1.png') {
    let pngString = fs.readFileSync(path.join(__dirname, '/1.png'));
    res.end(pngString);
  } else {
    res.setHeader('content-type', 'text/html;charset=utf-8');
    res.statusCode = 404;
    res.end('<h2>可惜了，找不到你要的资源' + req.url + '</h2>');
  }
});
//启动服务器，监听8082端口
```

```
app.listen(8082, () => {
  console.log('8082端口启动');
});
```

为不同的文件类型设置不同的 Content-Type

通过使用res对象中的setHeader方法，我们可以设置content-type这个响应头。这个响应头的作用是告诉浏览器，本次响应的内容是什么格式的内容。以方便浏览器进行处理。

常见的几中文件类型及content-type如下。

- .html：`res.setHeader('content-type', 'text/html;charset=utf-8')`
- .css：`res.setHeader('content-type', 'text/css;charset=utf-8')`
- .js：`res.setHeader('content-type', 'application/javascript')`
- .png：`res.setHeader('content-type', 'image/png')`

其它类型，参考这里：https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

批量处理请求

由于我们不法事先得知一个.html文件中会引用多少个静态资源，所以，我们不能像处理某个页面一样去处理它们。我们的解决办法有两大类是：

1. 把这类静态资源连同所有的.html文件全放在固定的文件夹中。在用户请求时，当判断当前的req.url是在这个文件夹下就是直接读内容，并返回。
- 2. 分析后缀名，如果是允许的，就直接返回

http模块-实现接口功能

在前面学习ajax时，我们说接口是后端同学写好的，我们前端同学只需要调用即可。现在，我们学习了nodejs，我们就可以客串一把后端同学的角色，来试着写写接口了。

get类型的接口-无参数

现在假设我们自己就是一名后端程序员，现在要实现一个get类型的接口。具体要求如下：

地址：/gettime

功能：以json字符串格式返回服务器的时间戳。

示例：

```
输入:localhost:8080/gettime;
返回:{_t:1563265441778}
```

要使用postman软件进行测试。

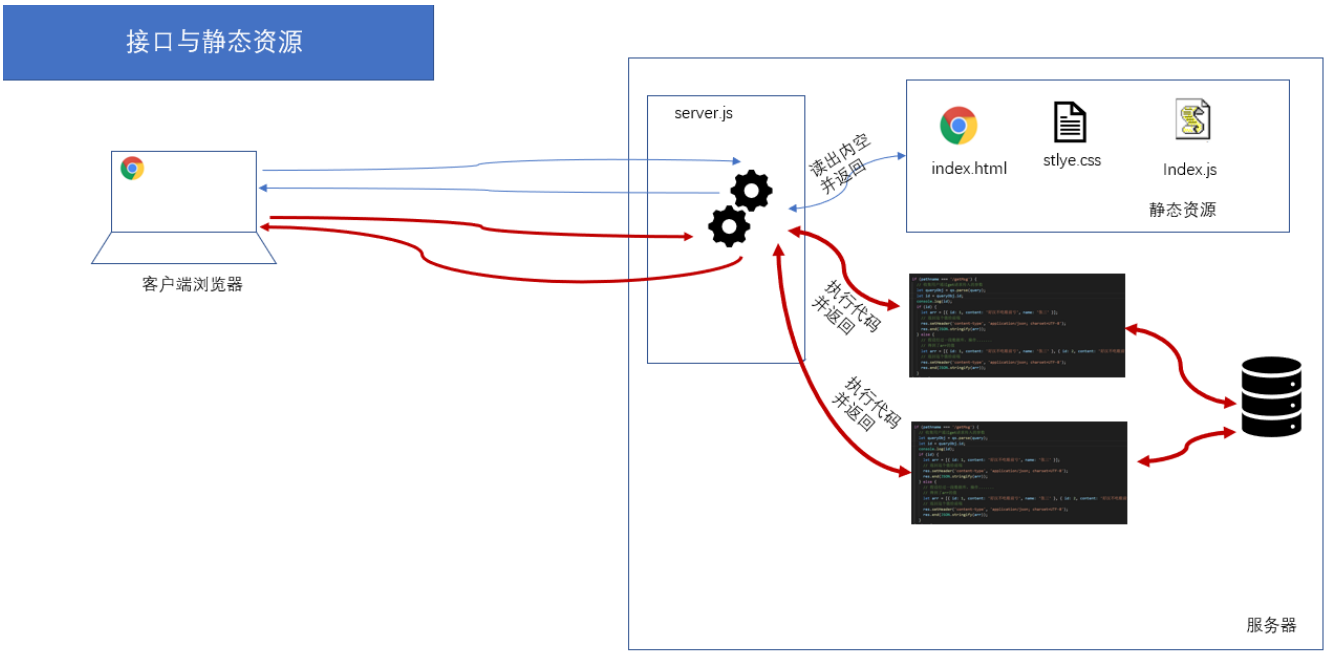
参考代码：

```
const http = require('http');
const app = http.createServer((req, res) => {
  if (pathname === '/gettime') {
    let obj = {_t : Date.now()}
    res.end(JSON.stringify(obj));//  把对象转成字符串之后再返回
  } else {
    res.end('error');
  }
});
app.listen(8083, () => {
  console.log(8083);
});
```

说明：

- req.method 可以判断请求的类型
- res.end()的参数只能是字符串，而不能是对象

接口与静态资源的区别



服务器上有很多的资源，每个资源都有自己的url。客户端浏览器想要访问某个资源就要向服务器发起对应的请求。

资源的分类：

- 静态资源。
  - 它们一般表现为一个一个的**文件**。例如index.html, style.css, index.js。
  - 处理请求静态资源时，服务器一般就直接读出资源的内容，再返回给客户端浏览器
- 动态资源：接口
  - 它们不是以某个具体的文件存在的，而是通过服务器上的一段代码处理得到的**数据**，访问接口时，服务器会执行这段代码，然后把代码的执行结果返回给客户端浏览器。

nodejs中的url模块

> 作用:url模块用来对url（例如：http://itcast.cn:80/schools/students?id=18&name=zs#photo）进行解析，进而得到各种信息。

- 文档地址：http://nodejs.cn/api/url.html

步骤：

- 引入

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

- 使用它的方法
  - obj = url.parse(地址栏中输入的url)

```
let urlobj = url.parse(req.url); // urlobj对象中，就有我们需要的信息
urlobj.pathname :获取用户输入的url的路径名 ('/schools/students')
urlobj.search: '?id=18&name=zs',
urlobj.query: 获取用户输入的url中的查询字符串( 'id=18&name=zs' )
urlobj.path: '/schools/students?id=18&name=zs',
urlobj.href: '/schools/students?id=18&name=zs'
```

上面urlobj.query只是获得了传递的全部参数，我们一般还需从地址栏中分析传递的数据。即从`http://itcast.cn:80/schools/students?id=18&name=zs#phot`中分析出id和name的值来。这个操作是如何实现的呢?

nodejs中的querystring模块

> 用来对url中的查询字符串这部分进行处理。nodejs中提供了querystring这个核心模块来帮助我们处理这个需求。

- 文档地址：https://nodejs.org/api/querystring.html#querystring_querystring_parse_str_sep_eq_options

基本示例

```
const qs= require('querystring');
let obj = qs.parse('id=18&name=zs');
console.log(obj)
```

get类型的接口-带参数

现在假设我们自己就是一名后端程序员，现在要实现一个get类型的接口。具体要求如下：

> 地址：localhost:8080/get
>
> 功能：获取用户传入的参数，并以json字符串格式返回，在返回的信息中要加上时间戳.
>
> 示例：

> 1.不加参数
> 输入:localhost:8080/get;
> 返回:{_t:1563265441778}
> 2.带参数
> 输入:localhost:8080/get?name=filex&age=30;
> 返回:{name:filex,age:30,_t:1563265441778}

> 要求：能通过postman软件的测试。

分析：get请求的参数附加在url中，我们可以使用url模块来取出用户url中的参数部分，再使用querystring模块取出具体的参数值。

这里我们直接使用两个核心模块`url`和`querystring`来实现上述的需求。

```
const http = require('http');
const queryString = require('querystring');
const url = require('url');

const server = http.createServer(function(req, res) {
  var { pathname, query } = url.parse(req.url);
  var obj = queryString.parse(query);

  console.log(p, url.parse(req.url));
  if (pathname === '/get' && req.method === 'GET') {
    res.setHeader('content-type', 'application/json');
    obj.d: Date.now() };
    res.end(JSON.stringify(str));
  } else {
    res.setHeader('content-type', 'text/html;charset=utf-8');
    res.end('大家好');
  }
});
server.listen(8088, function() {
  console.log('success', 8088);
});
```

post接口

假设我们自己就是一名后端程序员，现在要实现一个post类型的接口。具体要求如下：

> 地址：/post
>
> 功能：获取用户传入的参数，并以json字符串格式返回，在返回的信息中要加上时间戳.
>
> 示例：

> 接口地址:localhost:8080/post
> 参数：name=filex&age=30;
> 返回:{name:filex,age:30,_t:1563265441778}
>
> 要求：通过postman软件的测试。

post类型与get类型的接口区别较大，主要在两个方面：

1. 类型不同

   对于类型不同还比较好判断，我们可以通过 req.method 来获取

2. 传参不同

   - get请求参数在请求行中（附加在url后面）
   - post请求参数在`请求体`中

对于获取post参数就相对复杂一些，主要是用到request对象的两个事件data,end。

基本流程是：

1. 在req对象上添加两个事件，用来收集参数
   1. req.on("data",function(chunk){ })

      每次收到一部分数据就会触发一次这个事件，回调函数也会相应的执行一次。其中的chunk是一个形参（你也可以换个参数名），它是一个buffer。

   2. req.on("end",function(){})
2. 解析参数
   1. queryString

```
const http = require('http');
const url = require('url');
const queryString = require('querystring');
const server = http.createServer(function(req, res) {
  var { pathname } = url.parse(req.url);
  if (pathname === '/post' && req.method === 'POST') {
    let data = '';
    req.on('data', chunk => {
      data += chunk;
    });
    req.on('end', () => {
      res.setHeader('content-type', 'application/json');
      var str = { ...queryString.parse(data), d: Date.now() };
      res.end(JSON.stringify(str));
    });
  } else {
    res.setHeader('content-type', 'text/html;charset=utf-8');
    res.end('大家好');
  }
});

server.listen(8088, function() {
```

```
  console.log('success', 8088);
});
```

在发post请求时，传递的数据会在请求体中，它也是字符串格式，并且是一点一点上传到web服务器的（是积小成多，而不是一蹴而就）每上传一部分就会触发data事件，而最后全部上传完成之后，会触发end事件。

下面是一个示例代码，用来模拟使用post请求发送大量的数据，以观察req.on('data', chunk => {})多次触发的现象。

```
var xhr =new XMLHttpRequest();
xhr.open('post','http://localhost:8080/post');
xhr.setRequestHeader('content-type','application/x-www-form-urlencoded');
xhr.send("name="+"imissyou".repeat(100000));
```

自定义模块

我们自己写的模块就是自定义模块。在nodejs中，我们对代码的封装是以模块（一个一个的文件）为单位进行的。一般的做法是实现好某一个功能之后，封装成一个模块，然后在其它文件中使用这个模块。

使用一个模块，就是一个js文件中去使用另一个js文件中定义的变量，常量，函数....

基本步骤

1.定义模块

新建一个js文件，用模块名给它命名。例如，你的模块叫myModule，则这个js文件最好叫myModule.js

2.导出模块

在myModule.js内部，我们定义一些函数，变量，当然，它们会根据我们的业务要求做一些不同的工作。最后根据情况导出这些函数，变量。

```
//myModule.js
const myPI = 3;
function add(a, b) {
  return a + b;
}
// 通过module.exports来导出
module.exports = {
  myPI,
  add
};
```

注意：

- module.exports 是固定写法，一般放在文件的最末尾，也只用一次。
- module.exports表示当前模块要暴露给其它模块的功能。你当然不必须要所有在模块中定义的函数都暴露出来。

3.引入模块

在你需要使用模块的文件中，使用require语句引入你定义好的模块，注使用相对路径。假设我们当前的文件是index.js，而我们希望在index.js文件中使用myModule.js中的add方法。

我们的做法是：

```
// index.js
const myMath = require('./myMath');
```

上面的require()就是用来引入模块的。这里要注意使用自定义模块时，使用相对路径，而使用核心模块时，不需要写路径。

4.使用模块

当一个模块被成功引入之后，就可以按使用核心模块的过程一样去使用它们了。

```
// index.js
const myMath = require('./myMath');

// 在使用之前请先打印出来看看。
console.log(myMath);

let rs = myMath.add(23,45);
console.log(rs)
```

导出模块的两种方式

在自定义模块过程中，有两种导出模块的内容的方式：

- exports
- module.exports

参考：https://nodejs.org/api/modules.html#modules_exports_shortcut

它们的关系是： exports是module.exports的别名，即：

```
exports === module.exports
```

所以下面两种写法的效果是一样的：

> ```
> //1 mymodule.js
> exports.f = function(){ }
> exports.pi = 3.1415926
> ```
>
> ```
> //2 mymodule.js
> module.exports.f = function(){ }
> module.exports.pi = 3.1415926
> ```

区别在于：

- 在引入某模块时，以该模块中module.exports指向的内容为准。
- 在定义模块时：
  - 在初始时，exports和module.exports是指向同一块内存区域，其内容都是一个空对象。
  - 如果直接给exports对象赋值（例如：exports={a:1,b:2}），此时，exports就不会再指向module.exports，而转而指向这个新对象，此时，exports与module.exports不是同一个对象。而在引入模块时，是以模块的中的module.exports为准，因此，此时写在exports上的对象是无法导出的。
- 在导出模块过程中，建议只用一种方式（建议直接使用module.exports）。

npm使用

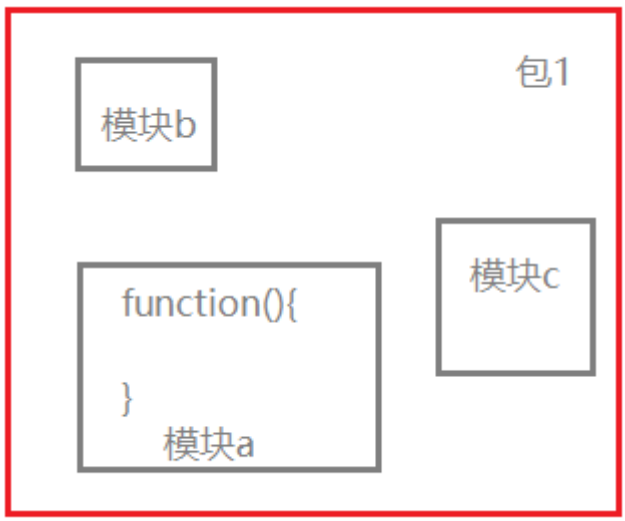
nodejs通过自带的`npm`(node package manager)工具来管理第三方模块，所以，在学习使用第三方模块时，我们先要学习npm的使用。

- `npm` 全称 `Node Package Manager`(node 包管理器)，它的诞生是为了解决 Node 中第三方包共享的问题。
- `npm` 不需要单独安装。在安装Node的时候，会连带一起安装 `npm`。
- 官网

当我们谈到npm时，我们在说两个东西：

- 命令行工具。这个工具在安装node时，已经自动安装过了。
- npm网站。这是一个第三方模块的"超市"，我们可以自由地下载，上传模块。

包（package）与模块关系



nodejs中一个模块就是一个单独的js文件

- **包是多个模块的集合。**一个模块能够解决的问题比较单一，一个包中有多个模块。
- npm 管理的单位就是包

类似于网站和网页的区别：一个网站一般会包含多个网页。

通过npm命令行下载第三方模块（包）

分成三步：

- 初始化项目。如果之前已经初始化，则可以省略
- 安装包。 **npm install 包名**
- 引入模块，使用。

第一步：初始化项目

进入到项目所在的根目录下，启动小黑窗（按下shift键，点击右键，在弹出的菜单中选择 在此处打开命令行）

输入如下命令：

```
npm init --yes
// 或者是 npm init -y
```

init命令用来在根目录下生成一个package.json文件，这个文件中记录了我们当前项目的基本信息。它是一切工作的开始。

第二步：安装包

生成了package.json文件之后，我们就可以来安装第三方包了。在npm官网中，有上百万个包，供我们使用。

npm init 命令

在某个目录下开启小黑窗，输入如下命令：

```
npm init
```

它会启动一个交互式的程序，让你填入一些关于本项目的信息。最后会生成一个package.json文件。

如果你希望直接采用默认信息，可以使用:

```
npm init --yes
// 或者是 npm init -y
```

说明：

- 这个命令只需要运行一次，它的目的仅仅是生成一个package.json文件。而这个package.json文件在后期是可以手动修改的。
- 如果项目根目录下已经有了package.json文件，就不需要再去运行这个命令了
- 一般我们从网上clone下来的项目都会包含这个文件。

package.json文件

它整体是一个json字符串，对当前项目的整体描述。其中最外层可以看作是一个js的对象（每一个属性名都加了""，这就是一个典型的json标记）。这个文件中有非常多的内容，我们目前学习如下几个：

- name

  表示这个项目的名字。如是它是一个第三方包的话，它就决定了我们在require()时应该要写什么内容

- version

  版本号

node_modules文件夹

这个文件夹中保存着我们从npm中下载来的第三方包。在使用npm install 命令时，会修改这个文件夹中的内容。具体如下来说，当键入`npm install XXX`之后（这里假设这个XXX包是存在的，也没有出现任何的网络错误）：

1. 如果有package.json

   (1) 修改package.json文件。根据开发依赖和生产依赖的不同，决定把这句记录在加在devDependencies或者是dependencies列表中。

   (2) 修改node_modules文件夹

   1. 如果有node_modules文件夹，则直接在下面新建名为XXX的文件夹，并从npm中下来这个包
   2. 如果没有node_modules，则先创建这个文件夹，再去下载相应的包
2. 如果没有package.json。会给一个警告信息

说明：

```
- 建议先创建package.json之后，再去install
- 在分享代码时，我们一般不需要把node_modules也给别人（就像你不需要把jquery.js给别人，因为他们可以自己去下载）。对方拿到我们的代码之后，先运行`npm install`(后面不接任何的包名)，自己去安装这些个依赖包。
```

全局安装包和本地安装包

我们通过`npm install`命令来安装包，简单说就是把包从npm的官网上下载到我们自己的电脑中。具体这个包下载到哪里了，还是有一点讲究的。

分成两类：

- 全局安装: 包被安装到了系统目录（一般在系统盘的node_modules中），本机都可以使用使用。
  - 命令：`npm install -g 包名`

- 局部安装（或者叫本地安装），包并安装在当前项目的根目录下，与package.json同级目录的node_modules。就只在这个项目中可以使用。
    - 命令：`npm install 包名`

全局安装nrm包

因为下载包时，默认是从npm官网（国外的网站）下载，速度可能会比较慢。在npm有一个工具可以来手动设置从哪里去下载包。这个工具就是nrm。这个工具是帮助我们切换安装包的来源的，不应该只限于某个具体的项目，所以我们采用全局安装的方式来安装它。

nrm包的地址：https://www.npmjs.com/package/nrm

nrm的使用方法。

```
// 第一步：  全局安装
npm install nrm -g

// 第二步：列出所有的源信息
//  (*) 标注的就是当前使用的源
nrm ls

// 第三步：根据需要切换源
// 例如：指定使用taobao源
nrm use taotao
```

全局包与本地包的区别

- 全局安装的包一般可提供直接执行的命令。我们通过对一些工具类的包采用这种方式安装，如：

  gulp, nodemon, live-server,nrm等。
- 本地安装的包是与具体的项目有关的， 我们需要在开发过程中使用这些具体的功能。

一个经验法则：

- 要用到该包的命令执行任务的就需要全局安装
- 要通过require引入使用的就需要本地安装

require的加载机制（了解）

在我们使用一个模块时，我们会使用require命令来加载这个模块。以加载一个自定义模块为例，require(文件名)的效果是：

1. 执行这个文件中的代码
2. 把这个文件中的module.exports对象中的内容返回出来。

以如下代码为例：

```
// moudule1.js
var a = 1;
var b = 2;
console.log(a+b);
var c = a+b;
module.exports = {
    data: c
}
```

在index.js中使用模块

```
// index.js
const obj = require('./moudule1.js');
console.log(obj);

这里的obj对象就是moudule1.js中的module.exports对象
```

- `require` 优先加载**缓存**中的模块
- 如果是**相对路径**，则根据路径加载**自定义模块**，并缓存
  - `require('./main')` 省略扩展名的情况
  - 先加载 `main.js`，如果没有再加载 `main.json`，如果没有再加载 `main.node`(c/c++编写的模块)
- 如果不是相对路径，则加载核心模块，并缓存
- 如果不是自定义模块，也不是核心模块，则加载**第三方模块**
  - node 会去本级 node_modules 目录中找`
  - 如果在 node_modules 目录中找到 `moment` 目录，则加载该模块并缓存
  - 如果过程都找不到，node 则取上一级目录下找 `node_modules` 目录，规则同上
  - 如果一直找到代码文件的文件系统的根路径还找不到，则报错

npm 常用命令

- 查看

  ```
  npm --version
  npm -v  // 查看npm 版本
  npm root -g // 查看全局包的安装目录
  ```

- 升级 npm （npm自己安装自己）

  ```
  npm install npm --global // 或者
  npm install npm -g
  ```

- 初始化 `package.json`

```
npm init -y // 或者
npm init --yes
```

- 安装第三方包

```
// 安装package.json中列出的所有的包
npm install

// 全局安装
npm install 包名 -g

// 本地安装，没有指定版本，默认安装最新的版本
npm install 包名
// 一次安装多个包
npm install 包名1 包名2 包名3
// 安装指定版本的包
npm install 包名@版本号

// 简写，把install简写成 i
npm i 包名
```

- 删除已安装的包

```
npm uninstall 本地安装的包名
npm uninstall -g 全局安装的包名
```

Express框架

Express 介绍

- Express 是一个基于 Node.js 平台，快速、开放、极简的 **web 开发框架**
- Express 是一个第三方模块，有丰富的 API 支持，强大而灵活的**中间件**特性
- Express 不对 Node.js 已有的特性进行二次抽象，只是在它之上扩展了 Web 应用所需的基本功能
- 链接
  - Express 官网
  - Express 中文文档（非官方）
  - Express GitHub仓库

运行第一个express程序

由于它是第三方框架，我们需要先安装它。

安装

> 参考文档：http://expressjs.com/en/starter/installing.html

```
# 在你的项目根目录下，打开小黑窗

# 1. 初始化 package.json 文件
npm init -y

# 2. 本地安装 express 到项目中
# npm install express
npm i express
```

注意：

- 项目目录名字不要取中文，也不要取express
- 如果安装不成功
  - 换个网络环境
  - 运行下`npm cache clean -f`清除缓存，再重装试试

使用

> 参考文档：http://expressjs.com/en/starter/hello-world.html

在项目根目录下新建一个js文件，例如app.js，其中输入代码如下：

```
// 0. 加载 Express
const express = require('express')

// 1. 调用 express() 得到一个 app
//    类似于 http.createServer()
const app = express()

// 2. 设置请求对应的处理函数
//    当客户端以 GET 方法请求 / 的时候就会调用第二个参数：请求处理函数
app.get('/', (req, res) => {
  res.send('hello world')
})

// 3. 监听端口号，启动 Web 服务
app.listen(3000, () => console.log('app listening on port 3000!'))
```

说明：

- app.get('/')相当于添加个事件监听：当用户以get方式求"/"时，它后面的回调函数会执行，其回调函数中的req,res与前面所学http模块保持一致。
- res.send()是exprss提供的方法，用于结束本次请求。类似的还有res.json(),res.sendFile() 。

托管静态资源

参考文档：http://expressjs.com/en/starter/static-files.html

让用户直接访问静态资源是一个web服务器最基本的功能。

```
http://localhost:3000/1.png
http://localhost:3000/css/style.css
http://localhost:3000/js/index.js
```

例如，如上url分别是请求一张图片，一份样式文件，一份js代码。我们实现的web服务器需要能够直接返回这些文件的内容给客户端浏览器。

在前面学习http模块时，我们已经实现了这些功能了，但是要写很多代码，现在使用express框架，只需一句代码就可以搞定了，这句代码是 `express.static('public')`

忽略前缀

```
// 0. 加载 Express
const express = require('express')

// 1. 调用 express() 得到一个 app
//    类似于 http.createServer()
const app = express();

// 2. 设置请求对应的处理函数
app.use(express.static('public'))


// 3. 监听端口号，启动 Web 服务
app.listen(3000, () => console.log('app listening on port 3000!'))
```

此时，所有放在public下的内容可以直接访问，注意，此时在url中并不需要出现public这级目录

- 在public下新建index.html。可以直接访问到。

限制前缀

```
// 限制访问前缀
app.use('/public', express.static('public'))
```

这意味着想要访问public下的内容，必须要在请求url中加上/public

路由

> 参考文档：http://expressjs.com/en/starter/basic-routing.html

路由是指确定应用程序如何处理客户端的请求。路由（**Routing**）是由一个 **URL**（或者叫路径标识）和一个特定的 **HTTP 方法**（GET、POST 等）组成的，涉及到应用如何处理响应客户端请求。每一个路由都可以有一个或者多个处理器函数，当匹配到路由时，这些个函数将被执行。

格式

```
app.METHOD(PATH, HANDLER)
```

其中：

- `app` 是 express 实例
- `METHOD` 是一个 HTTP 请求方法。 全小写格式。如：post,get,delete等
- `PATH` 是服务端路径（定位标识）

浏览器url	服务端路径
http://localhost:8080	/
http://localhost:8080/public/a/index.html	/public/a/index.html
http://localhost:8080/index.html?a=1&b=2	/index.html

- `HANDLER` 是当路由匹配到时需要执行的处理函数

示例

- 路径
  - http://127.0.0.1:3000/xxxx
  - app.get('路径')
  - 路径：域名后面的path
- 处理 get 请求

```
// 当你以 GET 方法请求 / 的时候，执行对应的处理函数
app.get('/', function (req, res) {
  res.send('Hello World!')
})

// 当你以 GET 方法请求 / 的时候，执行对应的处理函数
app.get('/file.html', function (req, res) {
  res.send('file.html');
  res.sendfile('文件路径')
  // 这里的文件路径必须是绝对路径
})
```

- 处理 post 请求

```
// 当你以 POST 方法请求 / 的时候，指定对应的处理函数
app.post('/', function (req, res) {
  res.send('Got a POST request')
})
```

写get接口

get无参数

```
const express = require('express');
const app = express();
app.get('/get', function(req, res) {
  // 直接返回对象
  res.json({ name: 'abc' });
});
app.listen('8088', () => {
  console.log('8088');
});
```

get有参数

express框架会自动收集get参数，并保存在req对象的query属性中。我们直接来获取即可。

```
const express = require('express');
const app = express();
app.get('/get', function(req, res) {
  // 直接返回对象
  console.log(req.query);

  res.send({ name: 'abc' });
});
app.listen('8088', () => {
  console.log('8088');
});
```

写post接口

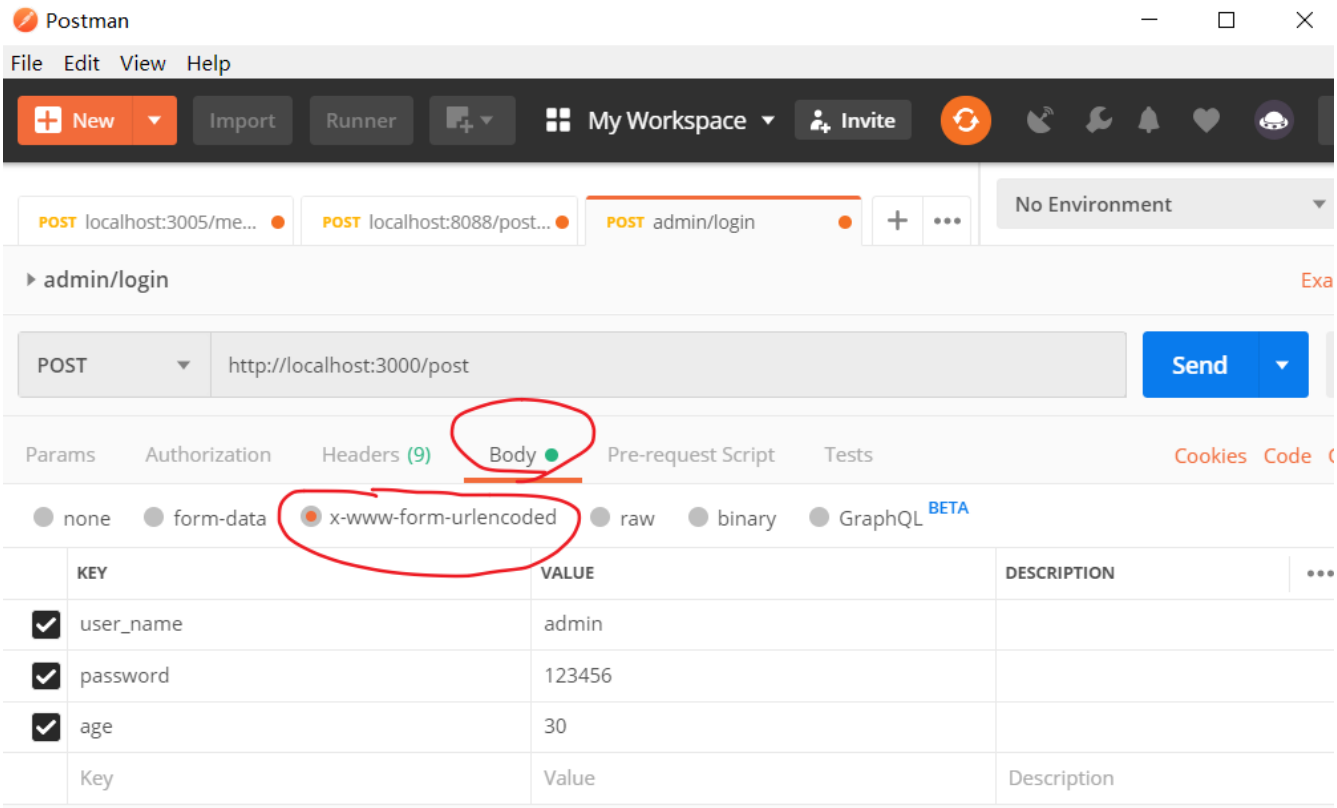
无参数

```
const app = express();
app.post('/post',function(req,res){
    res.send({name:"abc"})
})
```

普通键值对参数

获取post普通键值对数据，要通过第三方模块`body-parser`来解析。

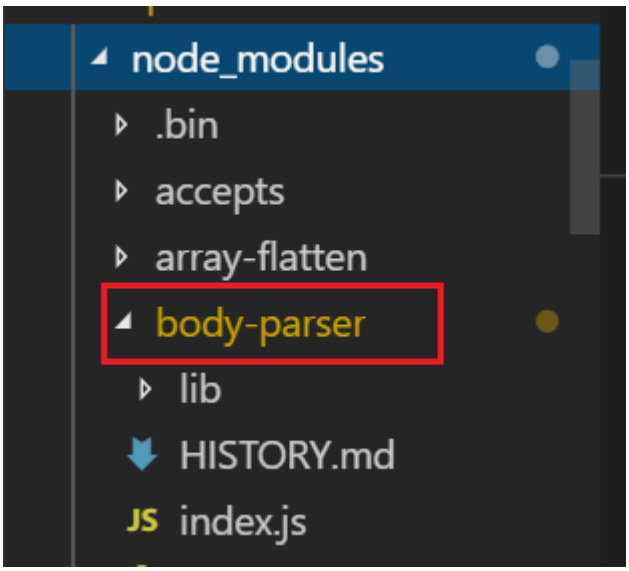
具体来说当content-type为x-www-form-urlencoded时，表示上传的普通简单的键值对 。如果通过postman测试的话，对应的设置如下：



一般要先下载body-parser这个包。

```
npm install body-parser
```

在 express4中，已经预先下载安装过了（在npm install exprss 时，就已经安装了body-parse，你可以在node_modules中查看到），这样就可以直接使用了



步骤

```
// 1. 引入包
const bodyParser = require('body-parser');

// 2. 使用包
app.use(bodyParser.urlencoded({extended:false}));

app.post("/add",function(req,res){

    //3. 可以通过req.body来获取post传递的键值对

})
```

注意：

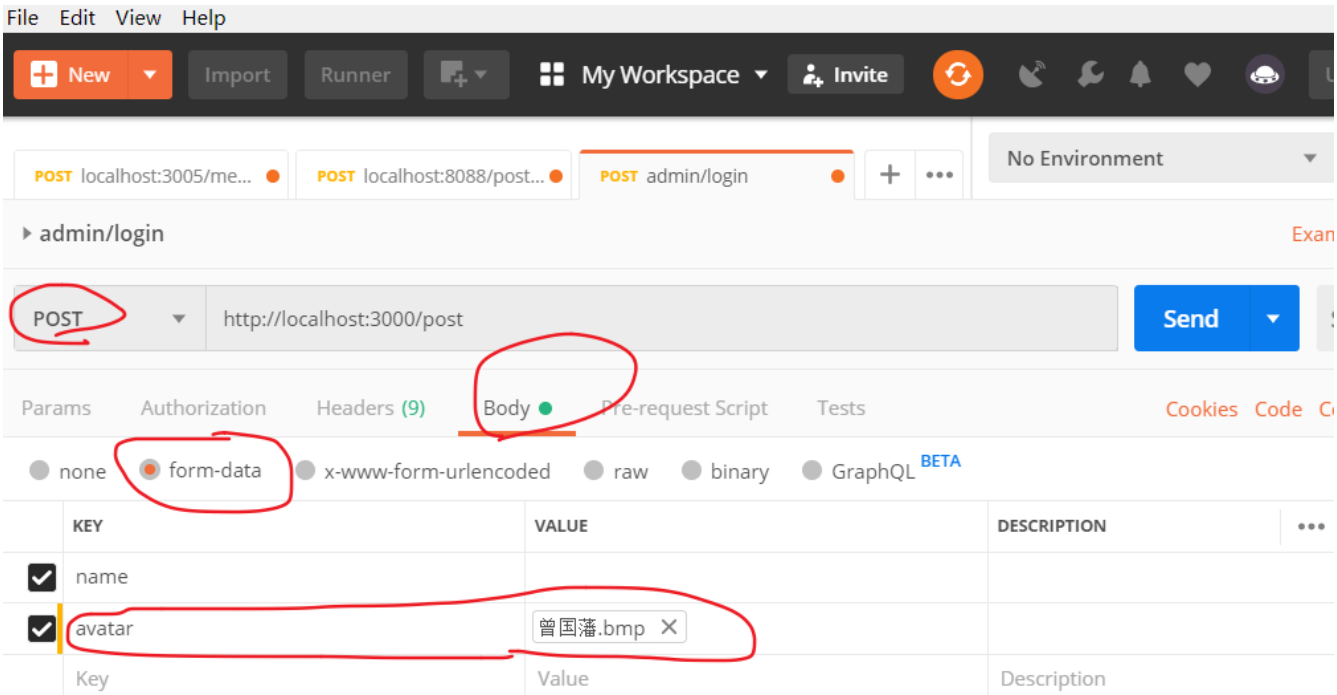
- app.use(....)之后，在res.body中就会多出一个属性res.body。

文件上传

如果post涉及文件上传操作，则会要使用`multer`这个包来获取上传的信息。

```
enctype="multipart/form-data"
```

对应postman的操作如下：



步骤

1.安装

```
npm install multer
```

2.使用

```
// 1. 引入包
const multer = require('multer');
// 2. 配置
const upload = multer({dest:'uploads/'}) // 上传的文件会保存在这个目录下
uploads表示一个目录名，你也可以设置成其它的

// 3. 使用
// 这个路由使用第二个参数 .upload.single表示单文件上传， 'cover' 表示要上传的文件在本次上次数据中的键名。
类似于<input type="file" name='cover'/>

app.post("postfile",upload.single('cover'), function(req,res){

})
```

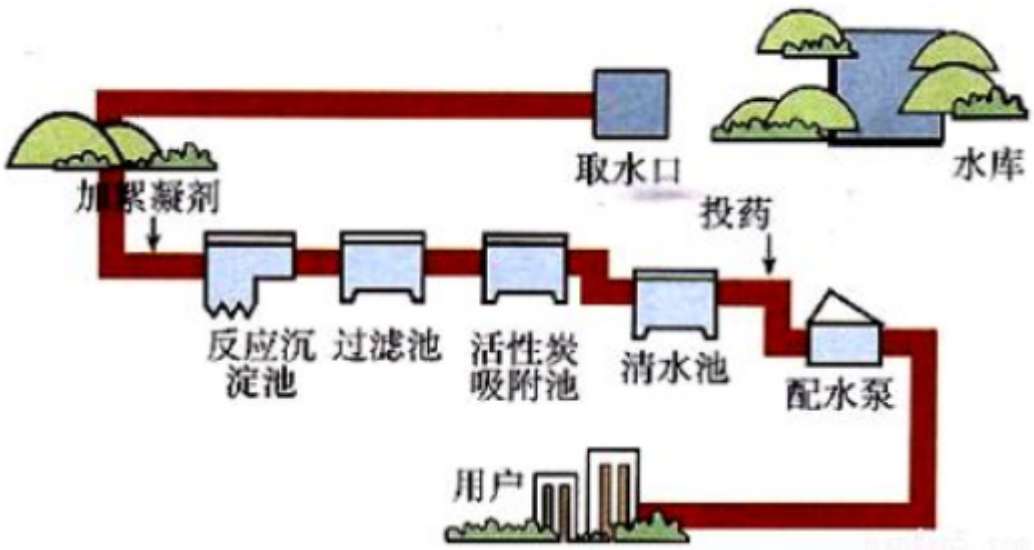
说明：

- 如果当前目录下没有uploads，它会自动创建uploads这个文件夹

中间件技术

在实际的工作中，我们需要对某些请求（或者某一类请求）进行特殊的处理，例如：要记录每一次请求的详细信息。需求：在调用某个接口时，打印出调用者的ip地址及调用时间。此时需要使用到中间件技术。同时对express而言，中间件是它的一个非常重要的概念，掌握中间件的思想对于理解学习express，提升编程水平都有很大的帮助。

生活中的中间件



在上图中，自来水厂从获取水源到净化处理交给用户，中间经历了一系列的处理环节。

- 一个流程结束之后，按顺序进入下一个流程；
- 一个流程如果出了问题，下一个流程也会受影响。
- 我们可以很方便地插入某一个特殊的处理环节，而不需要对原有环节造成影响。这样做的目的既提高了生产效率也保证了可维护性。

我们可以称其中的每一个处理环节就是一个中间件。

express中间件

中间件**是**一个特殊的url地址处理**函数**

- 中间件是 express 的最大特色，也是最重要的一个设计。`Express是一个自身功能极简，完全是路由和中间件构成一个web开发框架：从本质上来说，一个Express应用就是在调用各种中间件。`
- 一个 express 应用，就是由许许多多的中间件来完成的

格式及基本示例

格式

```
// 具名函数格式：
const handler1 = (req, res, next) => {
  console.log(Date.now());
  next();
}
app.use(handler1);

// 匿名函数格式：
app.use((req, res, next) => {
  console.log(Date.now());
  next();
});
```

- 中间件本质就是一个函数，它被当作 `app.use(中间件函数)` 的参数来使用
- 中间件函数中有三个基本参数， req、res、next
  - req就是请求相关的对象，它和下一个中间件函数中的req对象是一个对象
  - res就是响应相关的对象，它和下一个中间件函数中的res对象是一个对象
  - next：它是一个函数，调用它将会跳出当前的中间件函数，执行中间件；如果不调用next，则整个请求都会在当前中间件卡住，而得不到返回。

示例：使用中间件打印日志

```
var express = require('express')
var app = express()

var myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}
// 注册中间件
app.use(myLogger)

app.get('/', function (req, res) {
  res.send('Hello World!')
})

app.listen(3000)
```

示例：使用多个中间件

```
app.use((req, res, next) => {
  console.log("第1个中间件");
  next();
});

app.use((req, res, next) => {
  console.log("第2个中间件");
  res.setHeader('content-type', 'text/html;charset=utf8');
  res.a = 1;
  next();
});

app.use((req, res, next) => {
  console.log("第3个中间件");
  res.b = 2;
  console.log(res.a,res.b)
  res.end('中间件');
});
```

- 注意先后顺序。
- 注意通过req来附加额外的信息。

示例：设定特定路径的中间件

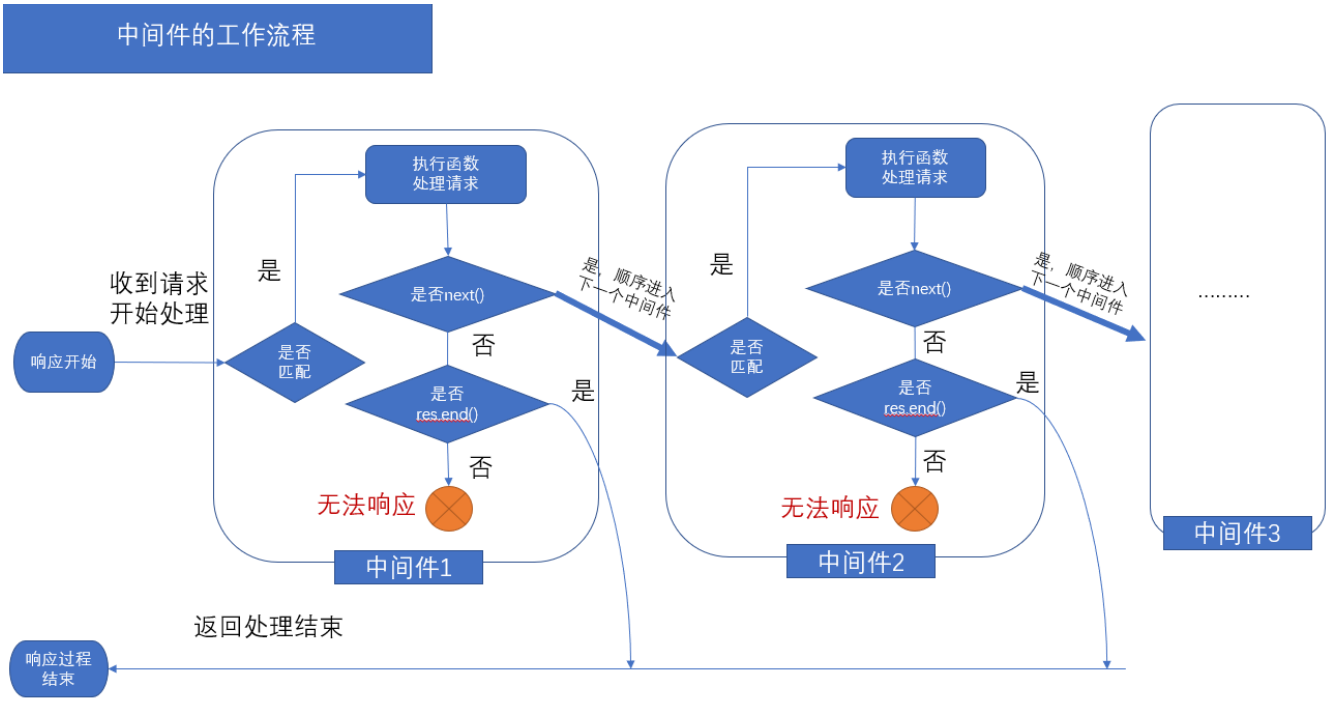
```
var express = require('express')
var app = express()

app.use(function (req, res, next) {
  console.log('应用级中间件，能匹配所有请求')
  next()
})
app.use('/api1',function (req, res, next) {
  console.log('只匹配/api1请求')
  next()
})

app.get('/api1', function (req, res) {
  res.send('Hello World!')
})

app.listen(3000)
```

中间件的执行流程



中间件的应用

模拟body-parser

```
app.use((req, res, next) => {
  if (req.method === 'POST') {
    let bodyStr = '';
    req.on('data', chunck => {
      bodyStr += chunck;
    });
    req.on('end', () => {
      req.body = qs.parse(bodyStr);
      next();
    });
  }
});
```

路由中间件

使用场景

路由过多时，代码不好管理。以大事件的代码为例，我们定义了管理员角色的接口和普通游客的接口，这些接口如果全写在一个入口文件中(如下只是显示了4个接口，如果是40个接口，就会很难读了)，是很不好维护的。

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

```
const app = express();
// 两种用户的操作，对应不同的接口
app.get('/getfrontdetail', (req, res) => {
  res.send('获取游客详情');
});

app.get('/getfrontinfo', (req, res) => {
  res.send('获取游客信息');
});

// 两种用户的操作，对应不同的接口
app.get('/getadmincate', (req, res) => {
  res.send('管理员获文章类别信息');
});

app.get('/getadmininfo', (req, res) => {
  res.send('获取管理员信息');
});

app.listen(3000, () => {
  console.log(3000);
});
```

我们的目标就是把它们拆开到不同的文件中，以便于管理。

基本步骤

1. 整理接口名。

   对众多的接口名进行整理和分类，以一级目录，二级目录这样的方式进行。例如：

   /admin/getcate

   /admin/getinfo

   /front/getinfo

   /front/getdetail

   2.通过nodejs的模块化，分模块定义路由中间件，并导出

   3.在入口文件中，导入并使用路由中间件

定义两个路由文件：

- front.js
- server.js

```
// ./router/front.js
const express = require('express');
const router = express.Router();
router.get('/getinfo', function(req, res) {
  res.send('getinfo');
});
router.get('/getdetail', function(req, res) {
  res.send('getdetail');
});
module.exports = router;
```

```
// ./router/server.js
const express = require('express');
const router = express.Router();

router.get('/getinfo', function(req, res) {
  res.send('管理员getinfo');
});
router.get('/getdetail', function(req, res) {
  res.send('管理员getdetail');
});
module.exports = router;
```

在主入口文件中使用它们

```
const express = require('express');
const app = express();

const frontRouter = require('./router/front');
const serverRouter = require('./router/admin');

app.use('/front', frontRouter);
app.use('/server', serverRouter);
```

会话技术

有很多的网站都有登录的功能：

```
|--login.html (登录页)
|--index.html(主页)
|--setting.html(设置页)
```

实际开发，我们必须解决页面之间的数据共享问题：例如用户从login.html页面登陆之后，再去访问index.html或者setting.html页面时，应该还是能够获取用户的登陆信息。

由于 http是无状态的，就是无记忆的，对于HTTP协议而言，无状态同样指每次request请求之前是相互独立的，当前请求并不会记录它的上一次请求信息。每次请求都是独立的，没有关联的，所以服务器和客户端都不知道是否是登录过的。

> 无状态的理发店：理发店剪头发，店长不记忆是否是老客户，每人每次25元。这就是无记忆的。
>
> 有状态的理发店：理发店剪头发，店长给你办理会员，下次你再来就记得你。这就是有记忆的。
>
> 店长能够知道你是谁，有两种记录会员的信息：
>
> - 第一次理发之后，给你一张会员片，下次再来的话，带上会员卡。（ cookie ： 把信息保存在浏览器端）
> - 第一次理发之后，把你的电话号码留下来，然后在**本子**上做记录。下次理发时，报出电话号码，他去查本子上的信息。（session：把信息保存在服务器上）
>
> 理发店：服务器
>
> 顾客：浏览器

什么是会话控制

会话控制就是用来弥补http无记忆的缺陷的一种技术。它能够将数据持久化（保存数据）的保存在客户端(浏览器)或者服务器端，从而让浏览器和服务器进行多次数据交换时，产生连续性。

让每一次的请求和响应都知道对方是谁。

会话控制的分类

- cookie： 将数据保存到**客户端**（浏览器）
- session： 将数据保存到**服务器端**

cookie（饼干，甜点）

查看cookie

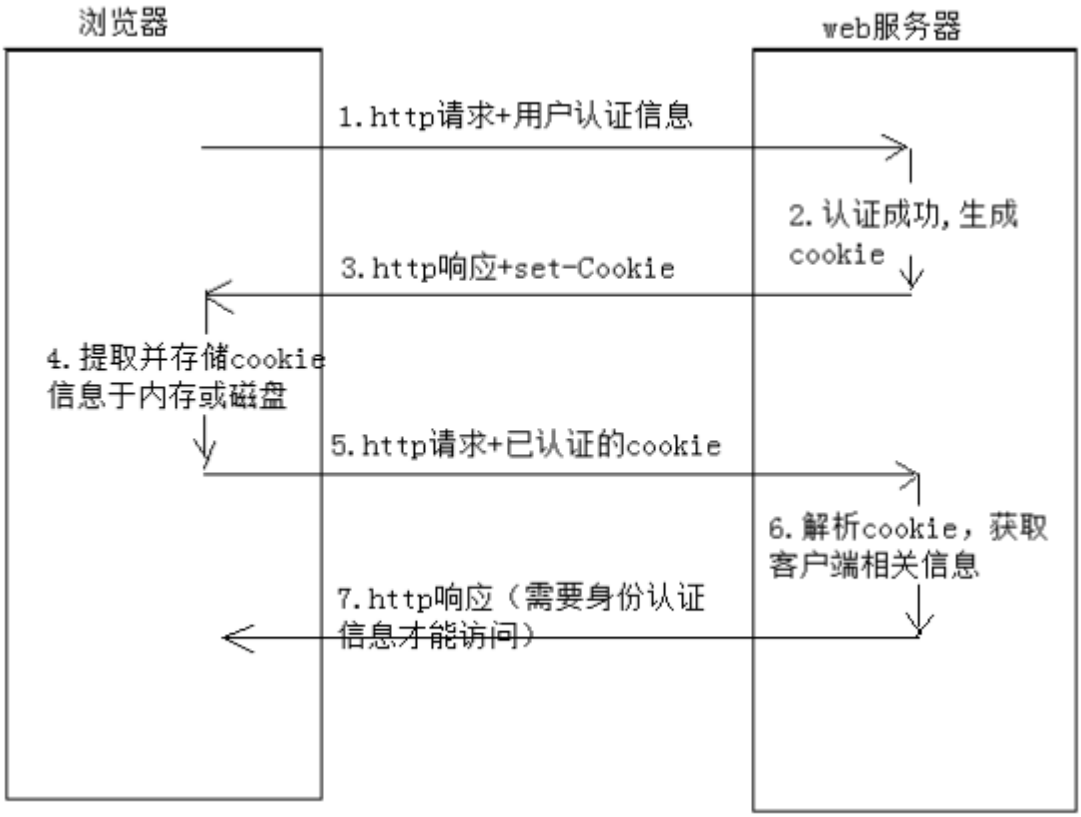
在浏览器中查看

- 在application--> cookie中查看。

在发送请求时的请求头中查看

理解cookie

- cookie是将数据持久化（保存）存储到客户端（浏览器）的一种技术
- cookie是**键值对格式的字符串**
- 可以通过浏览器查看某个网站的cookie
- 如果浏览器保存了cookie，则向服务器发请求时，就会**自动带上这个cookie。**把cookie放在请求头中，发送给服务器。



从服务器发送cookie给客户端

原生的方法

设置单个cookie

```
//在nodejs中，通过res.setHeader来设置响应头。
res.setHeader('set-cookie', 'name=curry');
```

设置多个cookie

```
//在nodejs中，通过res.setHeader来设置响应头。
res.setHeader('set-cookie', ['name=curry', 'age=30']);
```

如果cookie值是中文的话，还要对这个值进行额外的编码。

```
let name = new Buffer('好汉').toString('base64'); //5aW95rGJ
var info = new Buffer('5aW95rGJ', 'base64').toString();
```

express中提供的方法

express框架给我们提供了一个res.cookie方法，用来设置cookie

`res.cookie(cookie名, cookie值,{其它属性});`

cookie详细设置

参考：https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Set-Cookie

从服务器端，发送cookie给客户端，是通过设置Set-Cookie这个特殊的响应头来实现的。包括了对应的cookie的名称，值，以及各个属性。

```
Set-Cookie: <cookie-name>=<cookie-value>
Set-Cookie: <cookie-name>=<cookie-value>; Expires=<date>
Set-Cookie: <cookie-name>=<cookie-value>; Max-Age=<non-zero-digit>
Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>
Set-Cookie: <cookie-name>=<cookie-value>; Path=<path-value>
Set-Cookie: <cookie-name>=<cookie-value>; Secure
Set-Cookie: <cookie-name>=<cookie-value>; HttpOnly
Set-Cookie: <cookie-name>=<cookie-value>; SameSite=Strict
Set-Cookie: <cookie-name>=<cookie-value>; SameSite=Lax

// Multiple directives are also possible, for example:
Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>; Secure; HttpOnly
```

一个cookie所具有的主要的属性包括：

- Domain：域，表示当前cookie所属于哪个域或子域下面。对于服务器返回的Set-Cookie中，如果没有指定Domain的值，那么其Domain的值是默认为当前所提交的http的请求所对应的主域名的。比如访问 http://www.example.com，返回一个cookie，没有指名domain值，那么其为值为默认的www.example.com。
- Path：表示cookie的所属路径。
- Expire time/Max-age：表示了cookie的有效期。expire的值，是一个时间，过了这个时间，该cookie就失效了。或者是用max-age指定当前cookie是在多长时间之后而失效。如果服务器返回的一个cookie，没有指定其expire time，那么表明此cookie有效期只是当前的session，即是session cookie，当前session会话结束后，就过期了。对应的，当关闭（浏览器中）该页面的时候，此cookie就应该被浏览器所删除了。
- secure：表示该cookie只能用https传输。一般用于包含认证信息的cookie，要求传输此cookie的时候，必须用https传输。
- httponly：表示此cookie必须用于http或https传输。这意味着，浏览器脚本是不允许访问操作此cookie的(document.cookie不能访问)。

设置有效期

expires字段来用设置这个cookie在哪个时间内是有效的。值得注意的是，时间格式是UTC时间格式（不是中国时间）。

具体的语法是：

`"cooke-name=cookie-value;expires=UTC时间"`

如下示例

```
res.setHeader('set-cookie', ['id=1;expires=' + new Date(Date.now() + 1000 *
5).toUTCString()]);
```

表示id=1这个cookies在5秒之后失效。

如果使用express带的setCookie，则可以

`res.cookie('name', 'ok', { expires: new Date(Date.now() + 1000*5) })`

在服务器端获取cookie

1. 手动解析

向服务器发送的请求中会自动携带cookie，具体来说它会在req.headers.cookie中保存。要注意取到的cookie中只包括键值对，而cookies的属性（如过期时间）是看不到的。

```
req.headers.cookie; //isLogin=true; name=xsfss
```

这个字符中中包含了全部的cookie，为了把它们的值解析出来成一个对象，我们可以通过node的核心对象querystring来进行解析。

```
// 1. 把; 替换成&，以让querystring能够解析
let cookiestr = req.headers.cookie.replace('; ', '&');
console.log(req.headers.cookie);
console.log(cookiestr);
// 2 解析成对象
let cookieObj = qs.parse(cookiestr);
let { isLogin, name } = cookieObj;
```

2 . 使用cookie-parser进行解析

如果想快速解析，则可以使用cookie-parser这个包。

先安装`npm install cookie-parser`

再使用：

```
var cookieParser = require('cookie-parser');
app.use(cookieParser());

// 某个具体的路由回调函数中，cookies会以对象的格式保存在req对象中
console.log(req.cookies);
```

删除cookie

express框架提供了一个删除方法。从服务器端删除:

```
app.get('/quit', (req, res) => {
  res.clearCookie('name');
  res.clearCookie('isLogin');
  res.redirect('/login.html');
});
```

session

原理

session 从字面上讲，就是会话。这个就类似于你和一个人交谈，你怎么知道当前和你交谈的是张三而不是李四呢？对方肯定有某种特征（长相等）表明他就是张三。

session 也是类似的道理，服务器要知道当前发请求给自己的是谁。为了做这种区分，服务器就要给每个客户端分配不同的“身份标识”，然后客户端每次向服务器发请求的时候，都带上这个“身份标识”，服务器就知道这个请求来自于谁了。至于客户端怎么保存这个“身份标识”，可以有很多种方式，对于浏览器客户端，默认采用 cookie 的方式来保存这个身份标记。

服务器使用session把用户的信息临时保存在了服务器上，用户离开网站后session会被销毁。这种用户信息存储方式相对cookie来说更安全，可是session有一个缺陷：如果web服务器做了负载均衡，那么下一个操作请求到了另一台服务器的时候session会丢失。或者服务器重启了session数据也会丢失。

安装包

在express框架下， 我们可以通过安装 `express-session` 包来实现session的功能。

设置

```
1. 引入session包
const session = require('express-session');

const app = express();

//2. 配置项
let conf = {
  secret: '123456', //加密字符串。 使用该字符串来加密session数据，自定义
  resave: false, //强制保存session即使它并没有变化
  saveUninitialized: false //强制将未初始化的session存储。当新建了一个session且未
  //设定属性或值时，它就处于未初始化状态。
};

//3. 注册为express-session中间件
app.use(session(conf));
```

设置session

```
app.post('/loginAPI', (req, res) => {
    // 此处省略用户信息校验
    // 直接通过req.session来设置
    req.session.isLogin = true;
    req.session.name = req.body.username;
    res.end()
};
```

获取session

```
app.get('/index.html', (req, res) => {
    console.log(req.session)
};
```

删除session

```
app.get('/quit', (req, res) => {
    req.session.destroy();
};
```

参考代码

```
const express = require('express');
const session = require('express-session');
const bodyparser = require('body-parser');
const qs = require('querystring');
const app = express();

//2. 配置项
let conf = {
  secret: '4ey32erfyf3fgpg', //加密字符串。 使用该字符串来加密session数据，自定义
  resave: false, //强制保存session即使它并没有变化
  saveUninitialized: false //强制将未初始化的session存储。当新建了一个session且未
  //设定属性或值时，它就处于未初始化状态。
};
//3. 注册为express-session中间件
app.use(session(conf));
app.use(bodyparser.urlencoded({ extended: false }));
app.use(express.static('public'));

app.post('/loginAPI', (req, res) => {
  res.send(req.body.username);
  req.session.isLogin = true;
  req.session.name = req.body.username;

  let str = `
  <h1>${req.body.username}登录成功</h1>
  <a href='./vip.html'>vip</a>`;
  res.send(str);
});
app.get('/vip.html', (req, res) => {
  // session的值可以是任何的数据类型，比如布尔，数组，对象等

  let { isLogin, name } = req.session;

  if (isLogin) {
    let htmlstr = `
    <h1>weclome ${name}</h1>
    <a href='./quit'>退出</a>`;
    res.send(htmlstr);
```

```
  } else {
    res.send('<h1>请先登录</h1> <a href="./login.html">登录</a>');
  }
});

app.get('/quit', (req, res) => {
  req.session.destroy();
  res.redirect('/login.html');
});

app.listen(3000, () => {
  console.log(3000);
});
```

cookie、session原理

cookie原理：

- 从服务器端向客户端留下信息；每次访问时都带上

session原理：

- 服务器端会为每个用户（浏览器）各自保存一个session（文件）。当服务器保存session之后，会以cookie的形式告诉浏览器，你的session号是哪一个。它把session号返回给了浏览器，而把真实的数据保存在服务器。
- 下次再来访问服务器的时候，浏览器就会带着它自己的session号去访问，服务器根据session号就可以找到对应的session了。

cookie和session的优缺点

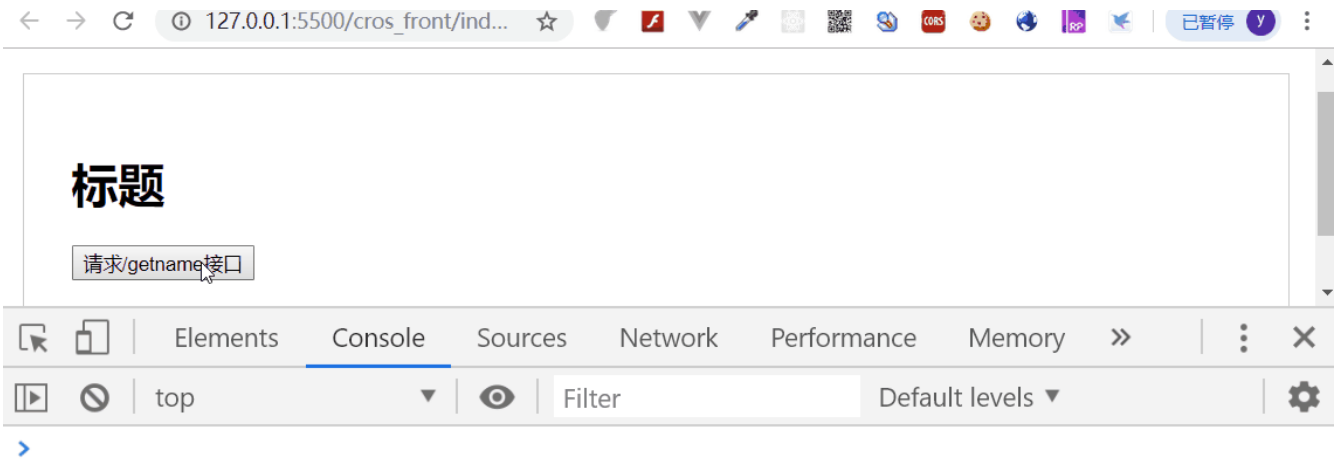
cookie：优点是节省服务器空间，缺点不安全。不要保存敏感信息。

session：优点是安全，缺点需要服务器空间， 是一种最常见的解决方案。

跨域

跨域问题是我们前端开发中经常会遇到的问题，也是面试中的高频题目。通过这一节的学习，我们就能解决这类问题啦。

问题演示



实现跨域请求的方案--CORS

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。它允许浏览器向跨源服务器，发出`XMLHttpRequest`请求，从而克服了AJAX只能同源使用的限制。CORS需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE浏览器不能低于IE10(ie8通过XDomainRequest能支持CORS)。

参考

通过在**被请求的路由中**设置header头，可以实现跨域。不过这种方式只从最新的浏览器（IE10）才支持。

```
app.get('/time', (req, res) => {
  // // 允许任意源访问，不安全
  // res.setHeader('Access-Control-Allow-Origin', '*')
  // 允许指定源访问
  res.setHeader('Access-Control-Allow-Origin', 'http://www.xxx.com')
  res.send(Date.now().toString())
})
```

- 这种方案无需客户端作出任何变化（客户端不用改代码），就当跨域问题不存在一样。
- 服务端响应的时候添加一个 `Access-Control-Allow-Origin` 的响应头
- 如果ajax请求中还附加了cookie，则还需要设置一句：`res.setHeader('Access-Control-Allow-Credentials', 'true');`

自行下载使用 npm cors https://www.npmjs.com/package/cors

jsonp vs cors 对比

jsonp：

- 不是ajax
- 只能使用`get方式`传参

- 兼容性好

cors:

- 就是ajax
- 支持各种方式的请求(post,get....)
- 浏览器的支持不好