

Node.js基础

■ 目录

模块和NPM

Express框架

■ Node.js模块

内置模块：例如 http、fs、net、process、path 等;

文件模块：原生模块之外的模块，和文件(夹)一一对应;

第三方模块：由第三方开发出来的模块，并非官方提供的内置模块，也不是用户创建的自定义模块，使用前需要先下载。

■ 使用文件模块

```
// app.js
var circle = require('./circle.js');
console.log('半径为4的圆面积是: ' + circle.area(4));
```

定义模块

```
// circle.js
const pi = Math.PI;
module.exports.area = function (r) {
    return pi * r * r;
};
module.exports.circumference = function (r) {
    return 2 * pi * r;
};
```

```
{ area: [Function (anonymous)], circumference: [Function (anonymous)] }
半径为4的圆面积是: 50.26548245743669
```

■ 模块加载

```
// 加载绝对路径文件  
require('/foo/bar/a.js');
```

```
// 加载相对路径文件  
require('../a.js');
```

```
// 加载无后缀的文件  
require('../a');
```

```
// 加载第三方模块  
require('moment');
```

■ 模块作用域

在自定义模块中定义的变量、方法等成员，只能在当前模块内被访问，这种模块级别的访问限制，叫做模块作用域。

- 模块间如何通信呢？

通过`require`加载模块，并通过`exports`接口对象暴露成员。`require`方法默认会执行被加载模块中的代码，并得到被加载模块的`exports`接口对象（该对象默认为一个空对象）。

■ 模块作用域

向外共享模块作用域中的成员

1. 在每个.js 自定义模块中都有一个 `module` 对象，它里面存储了和当前模块有关的信息。共享模块内的成员时，可以通过`module.exports`对象。用 `require()` 方法导入自定义模块时，得到的就是 `module.exports` 所指向的对象。
2. Node 提供了 `exports` 对象。默认情况下 `exports` 和 `module.exports` 指向同一个对象。

■ 模块化规范

Node.js遵循了 CommonJS 模块化规范， CommonJS 规定了模块的特性和各模块之间如何相互依赖。

1. 一个文件就是一个模块；
2. 每个文件中的变量函数都是私有的，对其他文件不可见的；
3. 每个文件中的变量函数必须通过`exports`暴露(导出)之后其它文件才可以使用；
4. 想要使用其它文件暴露的变量函数必须通过`require()`导入模块才可以使用；

■ npm包管理器

包其实指的是Node.js中的第三方模块。Node.js 中的包都是免费且开源的，可以免费下载使用，网址：<https://www.npmjs.com/>。

要想下载包，需要利用Node Package Manager（简称 npm 包管理工具），npm随着Node.js的安装一起安装到计算机中。

在项目中安装包的命令：

`npm install 包的完整名称`

可简写为：`npm i 包的完整名称`

例如：`npm i express`

■ npm包管理器

初次装包完成后，在项目文件夹下多一个叫做`node_modules`的文件夹和`package-lock.json`的配置文件。

其中：

`node_modules`文件夹用来存放所有已安装到项目中的包。`require()` 导入第三方包时，就是从这个目录中查找并加载包。

`package-lock.json` 配置文件 用来 记录 `node_modules` 目录下的每一个包的下载信息，例如包的名字、版本号、下载地址等。

注意：项目开发中一定要将 `node_modules` 文件夹添加到 `.gitignore`忽略文件中。

■ npm包管理器

安装包的时候还可以指定版本，使用@符号：npm i [express@4.18.2](#)

包的语义化版本规范：

其中每一位数字所代表的的含义如下：

第1 位数字： 大版本

第2 位数字： 功能版本

第3 位数字： Bug 修复版本

版本号的提升规则： 只要第一位数字的版本号增长，则后面的数字自动清零。

■ npm包管理器

项目根目录下必须提供一个名为`package.json` 的包管理配置文件。用来记录与项目有关的一些配置信息。例如：

1. 项目的名称、版本号、描述等；
 2. 项目中都用到了哪些包；
 3. 哪些包只在开发期间会用到；
 4. 哪些包在开发和部署时都需要用到；
- 可以通过`npm init -y`快速创建`package.json`文件

■ package.json文件

package.json 是对项目或者模块包的描述，里面包含许多元信息。比如项目名称，项目版本，项目执行入口文件，项目贡献者等等。

在项目根目录下执行 `npm init`，然后根据提示一步步输入相应的内容完成后即可自动创建该文件。

//描述，方便别人了解你的模块作用，搜索的时候也有用。

//main 项目的主入口文件,在模块化项目中都会有一个主模块,main 里面填写的就是主模块的入口文件

//定义命令别名,当命令很长时可以使用别名替换

//使用方法: `npm run 别名`

//以键值对 (key: value) 的形式来取“别名”

//项目遵循的协议,默认是ISC也就是开放源代码的协议

```
{
  "name": "expr",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

■ package.json文件

一个项目会以 `package.json` 作为索引文件记录

- 项目的依赖
- 项目的 `scripts` 命令

进入项目后使用 `npm install` 命令会自动读取索引文件安装项目的依赖模块

■ npm包管理器

package.json文件中，有一个 `dependencies` 节点，专门用来记录程序员使用 `npm install` 命令安装的第三方模块。

```
{  
  "dependencies": {  
    "express": "^4.18.2"  
  }  
}
```

如果大家在 `git clone` 一个剔除了 `node_modules` 的项目之后，需要先把所有的包下载到项目中，才能将项目运行起来。使用 `npm i` 一次性安装所有的依赖包。

如果想要卸载一个包，可以使用 `npm uninstall` 完整包名称。

■ npm包管理器

有些包只在项目开发阶段会用到，在项目上线之后不会用到，建议把这些包记录到 **devDependencies** 节点中。与之对应的，如果某些包在开发和项目上线之后都需要用到，建议把这些包记录到 **dependencies** 节点中。

使用命令：

`npm i 包名 -D`

或者

`npm i 包名 --save-dev`

■ npm包管理器

切换npm镜像源，提高包下载速度

查看当前镜像源：

```
npm config get registry
```

切换为淘宝镜像源：

```
npm config set registry https://registry.npm.taobao.org/
```

检查是否成功：

```
npm config get registry
```

■ npm包管理器

包的类型，两类：

1、项目包（安装在项目根目录下的node_modules文件夹中）

- 开发依赖包，即devDependencies节点中的包；
- 核心依赖包，即dependencies节点中的包；

2、全局包

npm install -g 包名

会把包安装为全局包，卸载一样，使用**npm uninstall -g 包名**

全局包会被安装到C:\Users\用户目录\AppData\Roaming\npm\node_modules 目录下。

只有工具性质的包，才有全局安装的必要性，提供了好用的终端命令。

■ 模块加载机制

优先从缓存中加载：

模块在第一次加载后会被缓存。因此多次调用 `require()` 不会导致模块的代码被执行多次。下次加载时直接读取缓存结果，避免文件 I/O 和解析时间。

不论是内置模块、用户自定义模块、还是第三方模块，它们都会优先从缓存中加载，从而提高模块的加载效率。

内置模块加载机制：

内置模块的加载优先级最高。`require('fs')` 始终返回内置的 `fs` 模块，即使在 `node_modules` 目录下有名字相同的包也叫做 `fs`。

■ 模块加载机制

自定义模块加载机制：

使用`require()` 加载自定义模块时，必须指定以`./`或`../`开头的 路径标识符 。如果没有指定，则 `node` 会把它当作内置模块或 第三方模块进行加载。

`require()`方式导入自定义模块时，如果省略了文件扩展名，按以下顺序分别尝试加载：

1. `.js`
2. `.json`
3. `.node`
4. `.mjs`
5. ...
6. 加载失败，终端报错

■ 模块加载机制

第三方模块加载机制：

如果传递给`require()` 的模块标识符不是一个内置模块，也没有以 `‘./’` 或 `‘../’` 开头，则 `Node.js` 会从当前模块的父目录开始，尝试从 `node_modules` 文件夹中加载第三方模块。如果没有找到对应的第三方模块， 则移动到再上一层父目录中，进行加载， 直到文件系统的根目录。

例如，假设在 `C:\Users\wen\nodeProjects\app.js` 文件里调用了 `require('utils')`， 则 `Node.js` 会按以下顺序查找：

1. `C:\Users\wen\nodeProjects\node_modules\utils`
2. `C:\Users\wen\node_modules\utils`
3. `C:\Users\node_modules\utils`
4. `C:\node_modules\utils`

■ 模块加载机制

将目录作为模块的加载机制

有三种加载方式：

1. 在被加载的目录下查找一个叫做 `package.json` 的文件，并寻找 `main` 属性，作为 `require()` 加载的入口
2. 如果目录里没有 `package.json` 文件，或者 `main` 入口不存在或无法解析，则 `Node.js` 将会试图加载目录下的 `index.js` 文件。
3. 如果以上两步都失败了，则 `Node.js` 会在终端打印错误消息，报告模块的缺失：`Error: Cannot find module 'xxx'`

■ 包依赖

```
"dependencies": {  
  "accepts": "^1.2.2",  
  "content-disposition": "~0.5.0",  
  "cookies": "~0.7.0",  
  "debug": "*",  
  "delegates": "^1.0.0",  
  "escape-html": "~1.0.1",  
  "fresh": "^0.5.2",  
  "only": "0.0.2",  
  "parseurl": "^1.3.0",  
  "statuses": "^1.2.0",  
  "type-is": "^1.5.5",  
  "vary": "^1.0.0"  
}
```

1.0.0 Must match version exactly

>1.0.0 Must be greater than version

>=1.0.0 <1.0.0 <=1.0.0

~1.0.0 "Approximately equivalent to version"

^1.0.0 "Compatible with version" 1.2.x 1.2.0, 1.2.1, etc., but not 1.3.0

* Matches any version

version1 - version2 Same as >=version1

<=version2.

...

■ 构建简单Web应用

```
const http = require('http')
const server = http.createServer((req, res) => {
  res.end('Hello World')
})
server.listen(3000)
```



Hello World

■ Express

中文官网: <http://www.expressjs.com.cn/>

安装方式: `npm i express@4.18.2`

Express 是基于 Node.js 平台快速、开放、极简的 Web 开发框架。

通俗的理解:

Express 的作用和 Node.js 内置的 http 模块类似, 是专门用来创建 Web 服务器的。

Express的本质: 就是一个 npm 上的第三方包, 提供了快速创建 Web 服务器的便捷方法。

■ Express

Express作用

对于前端程序员来说，最常见的两种服务器，分别是：

1. **Web 网站服务器**：专门对外提供 **Web 网页资源**的服务器。
2. **API 接口服务器**：专门对外提供 **API 接口**的服务器。

使用**Express**，可以方便、快速的创建 **Web 网站**的服务器或 **API 接口**的服务器。

■ Express

运行原理：底层http模块。

```
1  var http = require('http')
2
3  var app = http.createServer(function (request, response) {
4    response.writeHead(200, { 'Content-Type': 'text/plain' })
5    response.end('Hello world!')
6  })
7
8  app.listen(3000, 'localhost')
```

http模块

```
10 var express = require('express')
11 var app = express()
12
13 app.get('/', function (req, res) {
14   res.send('Hello world! ---express')
15 })
16
17 app.listen(3000)
```

express框架-核心是对http模块的再包装

■ Express托管静态资源

`express.static()`

通过它，我们可以非常方便地创建一个静态资源服务器。

例如，通过如下代码就可以将 `public` 目录下的图片、CSS 文件、JavaScript 文件对外开放访问了。

```
app.use(express.static('public'))
```

这句代码执行之后，就可以访问 `public` 目录下所有文件。

<http://localhost:3000/images/img1.jpg>

<http://localhost:3000/css/example.css>

注意：Express 在指定的静态目录中查找文件，并对外提供资源的访问路径。因此，存放静态文件的目录名不会出现在 URL 中。

■ Express托管静态资源

如果要托管多个静态资源目录，多次调用`express.static()`函数就可以。访问静态资源文件时，`express.static()` 函数会根据目录的添加顺序查找所需的文件。

```
app.use(express.static('public'))  
app.use(express.static('files'))
```

■ Express托管静态资源

如果希望在托管的静态资源访问路径之前，挂载路径前缀：

```
app.use('/public', express.static('public'))
```

可以通过带有/public前缀地址来访问public目录中的文件了

<http://localhost:3000/public/images/img1.jpg>

<http://localhost:3000/public/css/example.css>

■ Express路由

在Express 中，路由指的是客户端的请求与服务器处理函数之间的映射关系。

Express中的路由分 3 部分组成，分别是请求的类型、请求的 URL地址、处理函数。

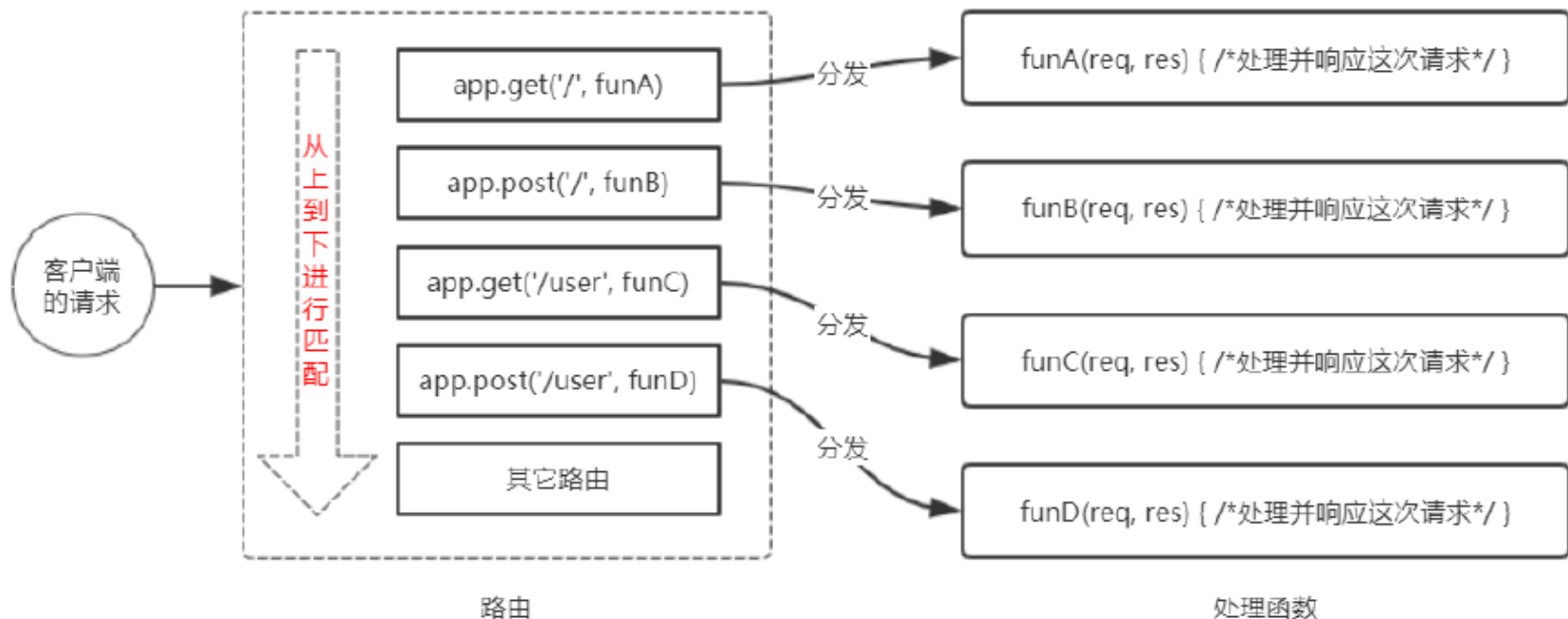
`app.METHOD(PATH, HANDLER)`

Express路由

路由的匹配过程:

每当一个请求到达服务器之后, 需要先经过路由的匹配, 只有匹配成功之后, 才会调用对应的处理函数。

在匹配时, 会按照路由的顺序进行匹配, 如果请求类型和请求的 URL 同时匹配成功, 则 Express 会将这次请求, 转交给对应的function 函数进行处理。



- 1、按照先后顺序进行匹配
- 2、请求类型相同并且URL相同

■ Express路由

最简单的使用方式:

```
1  const express = require('express')
2  const app = express()
3
4  //挂载路由
5  app.get('/hello', (req, res) => {
6    res.send('HelloWorld...')
7  })
8  app.get('/data', (req, res) => {
9    res.send({ name: 'Nodejs', flag: true })
10 })
11 app.listen(3000, () => {
12   console.log('server is running at http://localhost:3000')
13 })
```

■ Express路由

为了方便进行模块化管理，一般不建议把路由直接挂载到app上。常规做法是把路由抽离为单独的模块。

1、先创建路由模块user/user.js文件

```
1  const express = require('express')
2  // 创建路由对象
3  const router = express.Router()
4  // 挂载获取用户列表的路由
5  router.get('/user/list', function (req, res) {
6    res.send([
7      { name: 'Tom', age: 20, school: 'nku' },
8      { name: 'Tom2', age: 22, school: 'tju' }
9    ])
10 })
11 // 挂载添加用户的路由
12 router.post('/user/add', function (req, res) {
13   res.send({ name: 'Jerry', age: 18, school: 'nku' })
14 })
15 // 向外导出路由对象
16 module.exports = router
```

■ Express路由

为了方便进行模块化管理，一般不建议把路由直接挂载到app上。常规做法是把路由抽离为单独的模块。

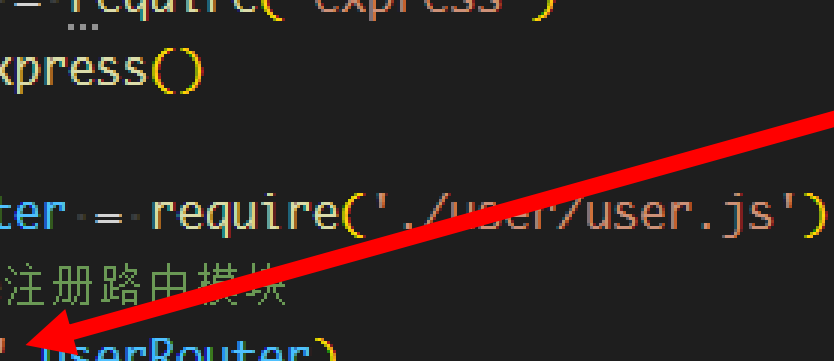
2、再注册路由模块index.js文件

```
nodejs-app > expr >  index.js > ...  
1  const express = require('express')  
2  const app = express()  
3  // 导入路由模块  
4  const userRouter = require('./user/user.js')  
5  // 使用app.use注册路由模块  
6  app.use(userRouter)  
7  
8  app.listen(3000, () => {  
9    | console.log('server is running.')10  })
```

■ Express路由

还可以给路由模块添加前缀，类似于托管静态资源，给静态资源统一挂载访问前缀一样，路由模块添加前缀的方式也非常简单。

```
1  const express = require('express')
2  const app = express()
3  // 导入路由模块
4  const userRouter = require('./user/user.js')
5  // 使用app.use注册路由模块
6  app.use('/api', userRouter)
7
8  app.listen(3000, () => {
9    console.log('server is running.')
10 })
11
```



■ Express中间件

Express 是一个自身功能极简，完全是由路由和中间件构成一个的 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。

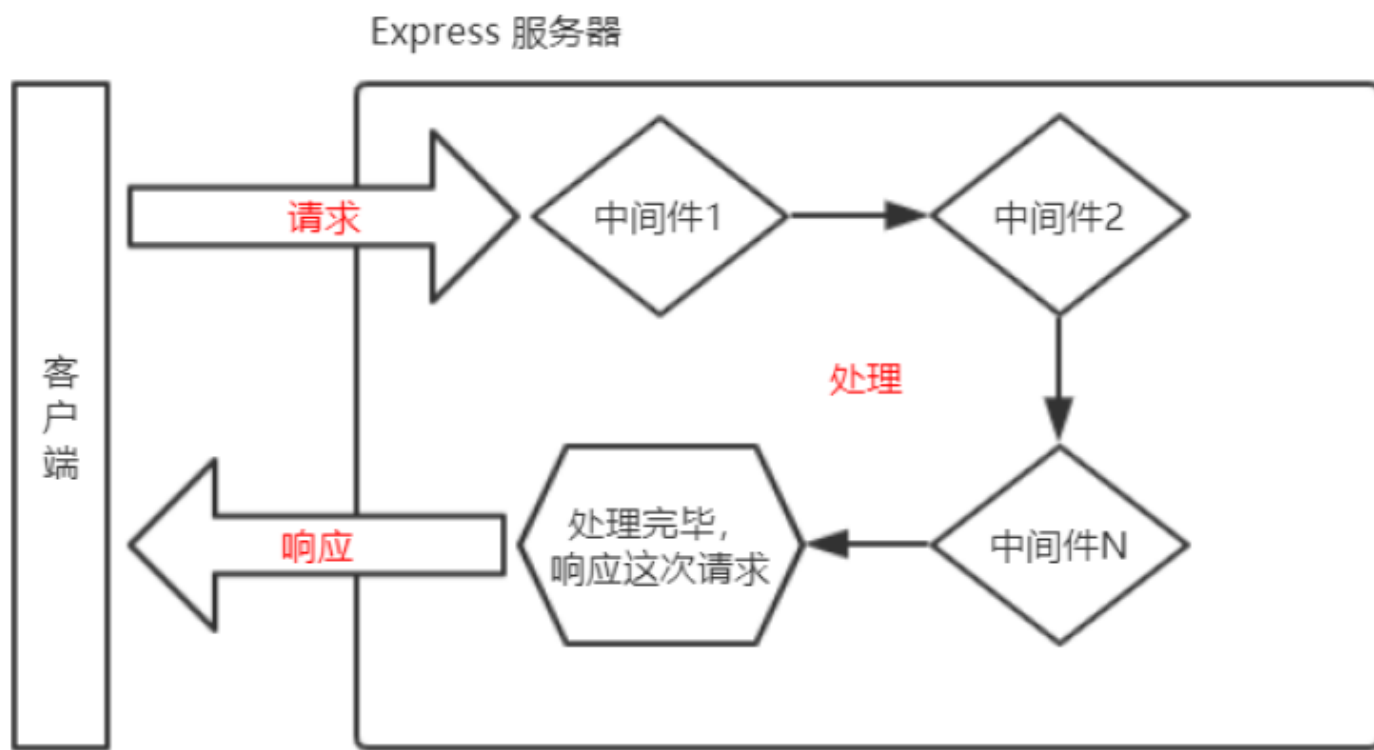
中间件（Middleware） 是一个函数，它可以访问请求对象（request object (req)），响应对象（response object (res)），和 web 应用中处于请求-响应循环流程中的中间件，一般被命名为 next 的变量。

```
function umw(req, res, next) {  
  next();  
}
```

■ Express中间件

中间件概念：

中间件（Middleware），指的是业务流程的中间处理环节。



■ Express中间件

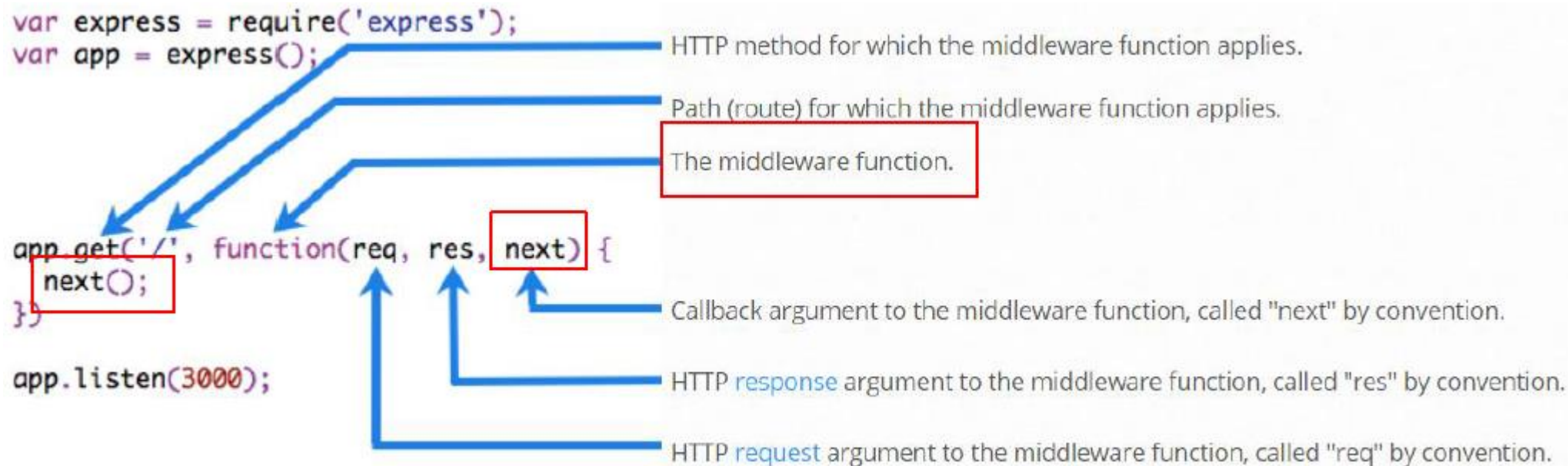
中间件功能：

1. 执行任何代码。
2. 修改请求和响应对象。
3. 终结请求-响应循环。
4. 调用堆栈中的下一个中间件。

如果当前中间件没有终结请求-响应循环，则必须调用 `next()` 方法将控制权交给下一个中间件，否则请求就会挂起。

■ Express中间件

格式:

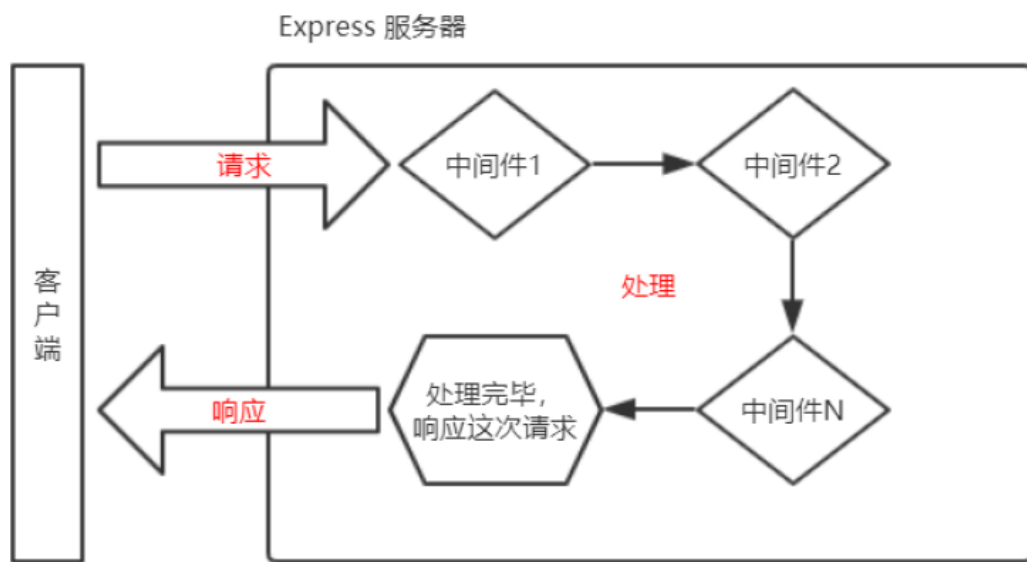


注意：中间件函数的形参列表中，必须包含 `next` 参数。而路由处理函数中只包含 `req` 和 `res`。

■ Express中间件

next()函数作用：

是实现多个中间件连续调用的关键，它表示把流转关系转交给下一个中间件或路由。



■ Express中间件

Express中间件有全局生效和局部生效两种：

1、客户端发起的任何请求，到达服务器之后都会触发的中间件，叫做全局生效的中间件。通过调用`app.use` (中间件函数)，即可定义一个全局生效的中间件。

```
//中间件函数
const mw = function (req, res, next) {
  console.log('Simple Middleware function')
  next()
}

//全局生效的中间件
app.use(mw)

app.get('/', function (req, res) {
  res.send('Hello world! ---express')
})
```

■ Express中间件

Express中间件有全局生效和局部生效两种：

2、不使用app.use ()定义的中间件，叫做局部生效的中间件。

```
const mw1 = function (req, res, next) {  
  console.log('Middleware-1')  
  next()  
}  
  
const mw2 = function (req, res, next) {  
  console.log('Middleware-2')  
  next()  
}  
  
app.get('/', mw1, mw2, (req, res) => {  
  res.send('Hello world! ---express')  
})  
  
// mw1、mw2不会影响/help这个路由  
app.get('/help', (req, res) => {  
  res.send('Help Page.')})
```

■ Express中间件

Express中间件使用的注意事项:

1. 一定要在路由之前注册中间件;
2. 客户端发送过来的请求, 可以连续调用多个中间件进行处理;
3. 执行完中间件的业务代码之后, 不要忘记调用 `next()` 函数;
4. 为了防止代码逻辑混乱, 调用 `next()` 函数后不要再写额外的代码;
5. 连续调用多个中间件时, 多个中间件之间, 共享 `req` 和 `res` 对象;

■ Express中间件

Express官方把中间件用法分为了常见的5类:

1. 应用级别的中间件
2. 路由级别的中间件
3. 错误级别的中间件—必须注册在所有路由之后
4. Express内置的中间件—`express.static()`、`express.json()`、`express.urlencoded()`
5. 第三方的中间件—按需下载注册使用, 可以提高开发效率