

黑马程序员™
www.itheima.com

传智播客旗下
高端IT教育品牌

Express



目录 Contents

- ◆ 初识 Express
- ◆ Express 路由
- ◆ Express 中间件
- ◆ 使用 Express 写接口



1. 初识 Express

1.1 Express 简介

1. 什么是 Express

官方给出的概念：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 的中文官网：<http://www.expressjs.com.cn/>



1. 初识 Express

1.1 Express 简介

2. 进一步理解 Express

思考：不使用 Express 能否创建 Web 服务器？

答案：能，使用 Node.js 提供的原生 http 模块即可。

思考：既生瑜何生亮（有了 http 内置模块，为什么还有用 Express）？

答案：http 内置模块用起来很复杂，开发效率低；Express 是基于内置的 http 模块进一步封装出来的，能够极大的提高开发效率。

思考：http 内置模块与 Express 是什么关系？

答案：类似于浏览器中 Web API 和 jQuery 的关系。后者是基于前者进一步封装出来的。



1. 初识 Express

1.1 Express 简介

3. Express 能做什么

对于前端程序员来说，最常见的两种服务器，分别是：

- **Web 网站服务器**：专门对外提供 Web 网页资源的服务器。
- **API 接口服务器**：专门对外提供 API 接口的服务器。

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

■ 1. 初识 Express

1.2 Express 的基本使用

1. 安装

在项目所处的目录中，运行如下的终端命令，即可将 express 安装到项目中使用：

```
1 npm i express@4.17.1
```



1. 初识 Express

1.2 Express 的基本使用

2. 创建基本的 Web 服务器

```
1 // 1. 导入 express
2 const express = require('express')
3 // 2. 创建 web 服务器
4 const app = express()
5
6 // 3. 调用 app.listen(端口号, 启动成功后的回调函数), 启动服务器
7 app.listen(80, () => {
8   console.log('express server running at http://127.0.0.1')
9 })
```



1. 初识 Express

1.2 Express 的基本使用

3. 监听 GET 请求

通过 `app.get()` 方法，可以监听客户端的 GET 请求，具体的语法格式如下：

```
1 // 参数1: 客户端请求的 URL 地址
2 // 参数2: 请求对应的处理函数
3 //      req: 请求对象 (包含了与请求相关的属性与方法)
4 //      res: 响应对象 (包含了与响应相关的属性与方法)
5 app.get('请求URL', function(req, res) { /*处理函数*/ })
```




1. 初识 Express

1.2 Express 的基本使用

4. 监听 POST 请求

通过 `app.post()` 方法，可以监听客户端的 POST 请求，具体的语法格式如下：

```
1 // 参数1: 客户端请求的 URL 地址
2 // 参数2: 请求对应的处理函数
3 //      req: 请求对象 (包含了与请求相关的属性与方法)
4 //      res: 响应对象 (包含了与响应相关的属性与方法)
5 app.post('请求URL', function(req, res) { /*处理函数*/ })
```



1. 初识 Express

1.2 Express 的基本使用

5. 把内容响应给客户端

通过 `res.send()` 方法，可以把处理好的内容，发送给客户端：

```
1 app.get('/user', (req, res) => {  
2   // 向客户端发送 JSON 对象  
3   res.send({ name: 'zs', age: 20, gender: '男' })  
4 })  
5  
6 app.post('/user', (req, res) => {  
7   // 向客户端发送文本内容  
8   res.send('请求成功')  
9 })
```



1. 初识 Express

1.2 Express 的基本使用

6. 获取 URL 中携带的查询参数

通过 `req.query` 对象，可以访问到客户端通过查询字符串的形式，发送到服务器的参数：

```
1 app.get('/', (req, res) => {  
2   // req.query 默认是一个空对象  
3   // 客户端使用 ?name=zs&age=20 这种查询字符串形式，发送到服务器的参数，  
4   // 可以通过 req.query 对象访问到，例如：  
5   // req.query.name    req.query.age  
6   console.log(req.query)  
7 })
```



1. 初识 Express

1.2 Express 的基本使用

7. 获取 URL 中的动态参数

通过 `req.params` 对象，可以访问到 URL 中，通过 `:` 匹配到的动态参数：

```
1 // URL 地址中，可以通过 :参数名 的形式，匹配动态参数值
2 app.get('/user/:id', (req, res) => {
3   // req.params 默认是一个空对象
4   // 里面存放着通过 : 动态匹配到的参数值
5   console.log(req.params)
6 })
```



1. 初识 Express

1.3 托管静态资源

1. express.static()

express 提供了一个非常好用的函数，叫做 `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：

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

现在，你就可以访问 public 目录中的所有文件了：

`http://localhost:3000/images/bg.jpg`

`http://localhost:3000/css/style.css`

`http://localhost:3000/js/login.js`

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。



1. 初识 Express

1.3 托管静态资源

2. 托管多个静态资源目录

如果要托管多个静态资源目录，请多次调用 `express.static()` 函数：

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

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。



1. 初识 Express

1.3 托管静态资源

3. 挂载路径前缀

如果希望在托管的静态资源访问路径之前，挂载路径前缀，则可以使用如下的方式：

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

现在，你可以通过带有 `/public` 前缀地址来访问 `public` 目录中的文件了：

`http://localhost:3000/public/images/kitten.jpg`

`http://localhost:3000/public/css/style.css`

`http://localhost:3000/public/js/app.js`



1. 初识 Express

1.4 nodemon

1. 为什么要使用 nodemon

在编写调试 Node.js 项目的时候，如果修改了项目的代码，则需要频繁的手动 close 掉，然后再重新启动，非常繁琐。

现在，我们可以使用 nodemon (<https://www.npmjs.com/package/nodemon>) 这个工具，它能够监听项目文件的变动，当代码被修改后，nodemon 会自动帮我们重启项目，极大方便了开发和调试。

1. 初识 Express

1.4 nodemon

2. 安装 nodemon

在终端中，运行如下命令，即可将 nodemon 安装为全局可用的工具：

```
1 npm install -g nodemon
```



1. 初识 Express

1.4 nodemon

3. 使用 nodemon

当基于 Node.js 编写了一个网站应用的时候，传统的方式，是运行 `node app.js` 命令，来启动项目。这样做的坏处是：代码被修改之后，需要手动重启项目。

现在，我们可以将 `node` 命令替换为 `nodemon` 命令，使用 `nodemon app.js` 来启动项目。这样做的好处是：代码被修改之后，会被 `nodemon` 监听到，从而实现自动重启项目的效果。

```
1 node app.js
2 # 将上面的终端命令，替换为下面的终端命令，即可实现自动重启项目的效果
3 nodemon app.js
```

目录 Contents

- ◆ 初识 Express
- ◆ Express 路由
- ◆ Express 中间件
- ◆ 使用 Express 写接口

2. Express 路由

2.1 路由的概念

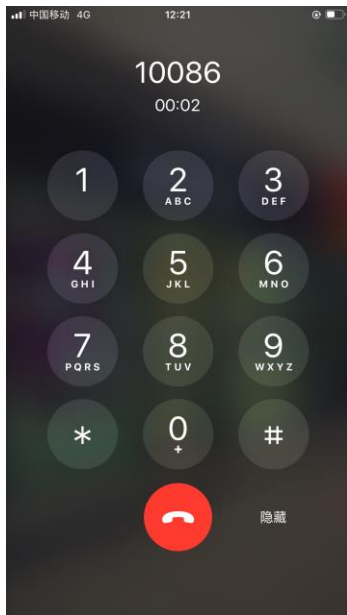
1. 什么是路由

广义上来讲，路由就是映射关系。

2. Express 路由

2.1 路由的概念

2. 现实生活中的路由



按键 1 -> 业务查询

按键 2 -> 手机充值

按键 3 -> 业务办理

按键 4 -> 密码服务与停复机

按键 5 -> 家庭宽带

按键 6 -> 话费流量

按键 8 -> 集团业务

按键 0 -> 人工服务

在这里，路由是按键与服务之间的映射关系

2. Express 路由

2.1 路由的概念

3. Express 中的路由

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，格式如下：

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

2. Express 路由

2.1 路由的概念

4. Express 中的路由的例子

```
1 // 匹配 GET 请求, 且请求 URL 为 /
2 app.get('/', function (req, res) {
3   res.send('Hello World!')
4 })
5
6 // 匹配 POST 请求, 且请求 URL 为 /
7 app.post('/', function (req, res) {
8   res.send('Got a POST request')
9 })
```

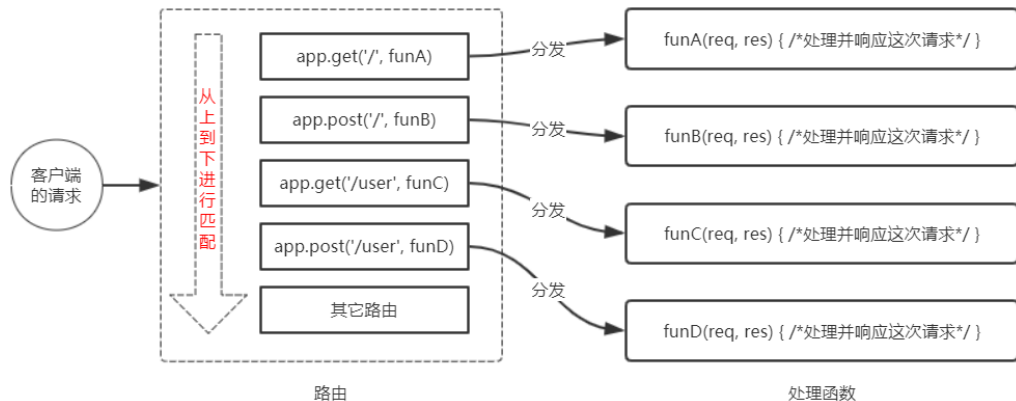
2. Express 路由

2.1 路由的概念

5. 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。



路由匹配的注意点：

- ① 按照定义的**先后顺序**进行匹配
- ② **请求类型**和**请求的URL**同时匹配成功，才会调用对应的处理函数

2. Express 路由

2.2 路由的使用

1. 最简单的用法

在 Express 中使用路由最简单的方式，就是把路由挂载到 app 上，示例代码如下：

```
1 const express = require('express')
2 // 创建 Web 服务器，命名为 app
3 const app = express()
4
5 // 挂载路由
6 app.get('/', (req, res) => { res.send('Hello World.') })
7 app.post('/', (req, res) => { res.send('Post Request.') })
8
9 // 启动 Web 服务器
10 app.listen(80, () => { console.log('server running at http://127.0.0.1') })
```

2. Express 路由

2.2 路由的使用

2. 模块化路由

为了方便对路由进行模块化的管理，Express **不建议**将路由直接挂载到 app 上，而是**推荐**将路由抽离为单独的模块。

将路由抽离为单独模块的步骤如下：

- ① 创建路由模块对应的 .js 文件
- ② 调用 `express.Router()` 函数创建路由对象
- ③ 向路由对象上挂载具体的路由
- ④ 使用 `module.exports` 向外共享路由对象
- ⑤ 使用 `app.use()` 函数注册路由模块

2. Express 路由

2.2 路由的使用

3. 创建路由模块

```
1 var express = require('express')           // 1. 导入 express
2 var router = express.Router()               // 2. 创建路由对象
3
4 router.get('/user/list', function (req, res) { // 3. 挂载获取用户列表的路由
5   res.send('Get user list.')
6 })
7 router.post('/user/add', function (req, res) { // 4. 挂载添加用户的路由
8   res.send('Add new user.')
9 })
10
11 module.exports = router                     // 5. 向外导出路由对象
```

2. Express 路由

2.2 路由的使用

4. 注册路由模块

```
1 // 1. 导入路由模块
2 const userRouter = require('./router/user.js')
3
4 // 2. 使用 app.use() 注册路由模块
5 app.use(userRouter)
```

2. Express 路由

2.2 路由的使用

5. 为路由模块添加前缀

类似于托管静态资源时，为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 // 1. 导入路由模块
2 const userRouter = require('./router/user.js')
3
4 // 2. 使用 app.use() 注册路由模块，并添加统一的访问前缀 /api
5 app.use('/api', userRouter)
```

目录 Contents

- ◆ 初识 Express
- ◆ Express 路由
- ◆ Express 中间件
- ◆ 使用 Express 写接口

3. Express 中间件

3.1 中间件的概念

1. 什么是中间件

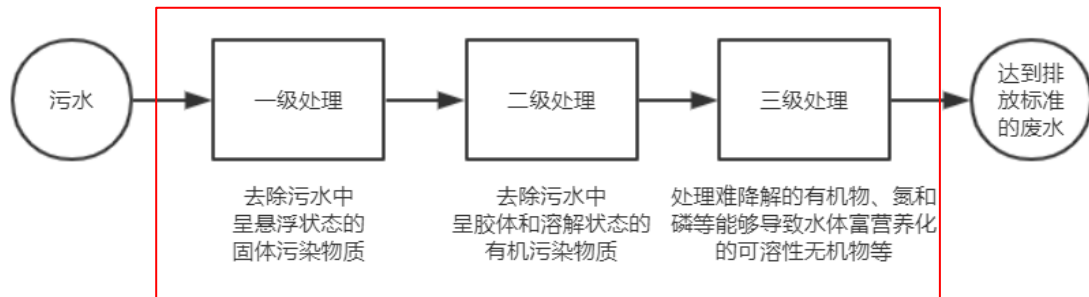
中间件（Middleware），特指[业务流程](#)的[中间处理环节](#)。

3. Express 中间件

3.1 中间件的概念

2. 现实生活中的例子

在处理污水的时候，一般都要经过三个处理环节，从而保证处理过后的废水，达到排放标准。



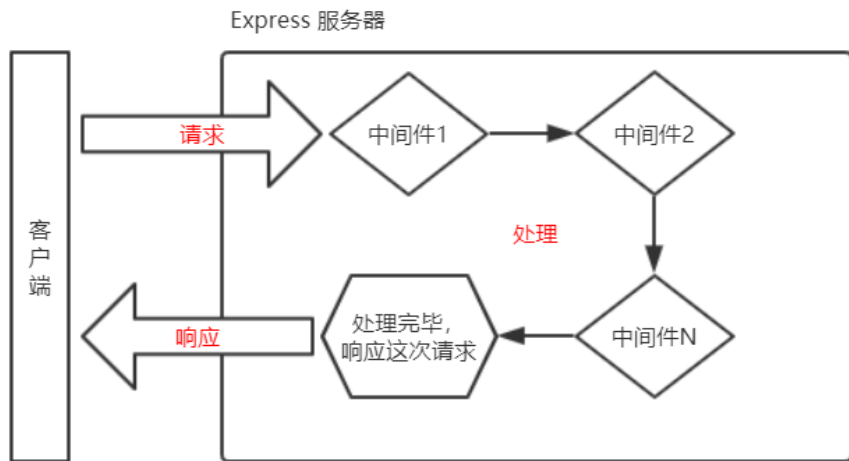
处理污水的这三个中间处理环节，就可以叫做中间件。

3. Express 中间件

3.1 中间件的概念

3. Express 中间件的调用流程

当一个请求到达 Express 的服务器之后，可以连续调用多个中间件，从而对这次请求进行预处理。



3. Express 中间件

3.1 中间件的概念

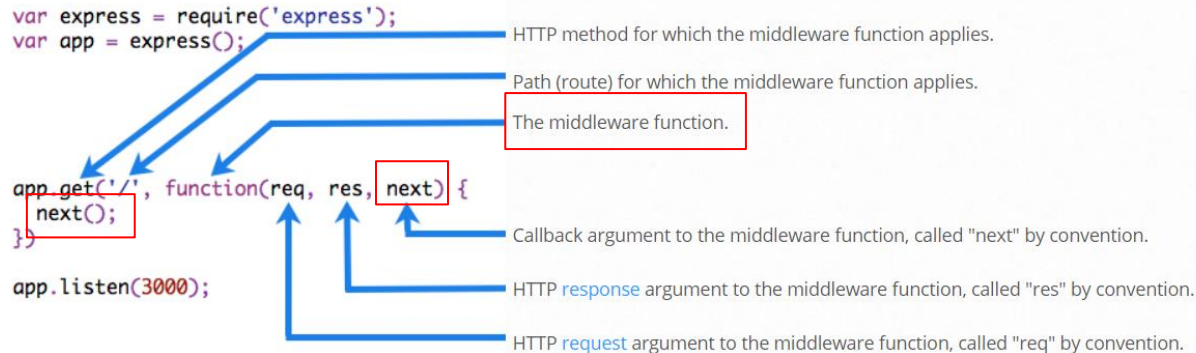
4. Express 中间件的格式

Express 的中间件，本质上就是一个 **function 处理函数**，Express 中间件的格式如下：

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

app.get('/', function(req, res, next) {
  next();
});

app.listen(3000);
```



HTTP method for which the middleware function applies.

Path (route) for which the middleware function applies.

The middleware function.

Callback argument to the middleware function, called "next" by convention.

HTTP response argument to the middleware function, called "res" by convention.

HTTP request argument to the middleware function, called "req" by convention.

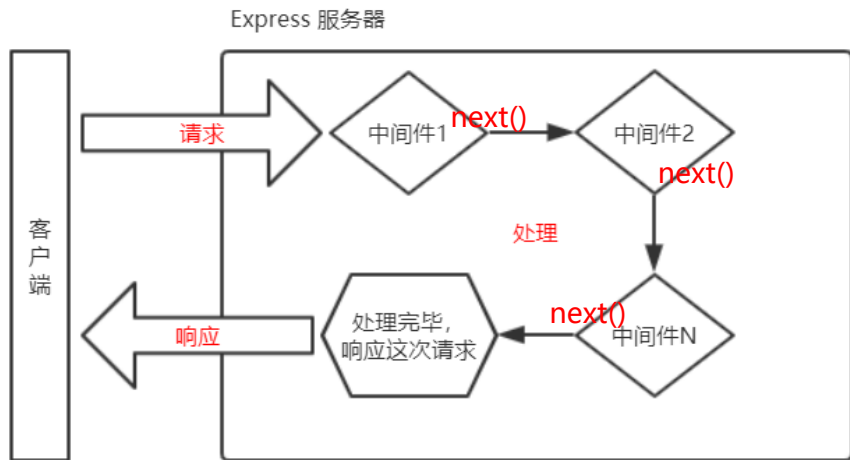
注意：中间件函数的形参列表中，**必须包含 next 参数**。而路由处理函数中只包含 req 和 res。

3. Express 中间件

3.1 中间件的概念

5. next 函数的作用

next 函数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。



3. Express 中间件

3.2 Express 中间件的初体验

1. 定义中间件函数

可以通过如下的方式，定义一个最简单的中间件函数：

```
1 // 常量 mw 所指向的，就是一个中间件函数
2 const mw = function (req, res, next) {
3   console.log('这是一个最简单的中间件函数')
4   // 注意：在当前中间件的业务处理完毕后，必须调用 next() 函数
5   // 表示把流转关系转交给下一个中间件或路由
6   next()
7 }
```



3. Express 中间件

3.2 Express 中间件的初体验

2. 全局生效的中间件

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，示例代码如下：

```
1 // 常量 mw 所指向的，就是一个中间件函数
2 const mw = function (req, res, next) {
3   console.log('这是一个最简单的中间件函数')
4   next()
5 }
6
7 // 全局生效的中间件
8 app.use(mw)
```

3. Express 中间件

3.2 Express 中间件的初体验

3. 定义全局中间件的简化形式

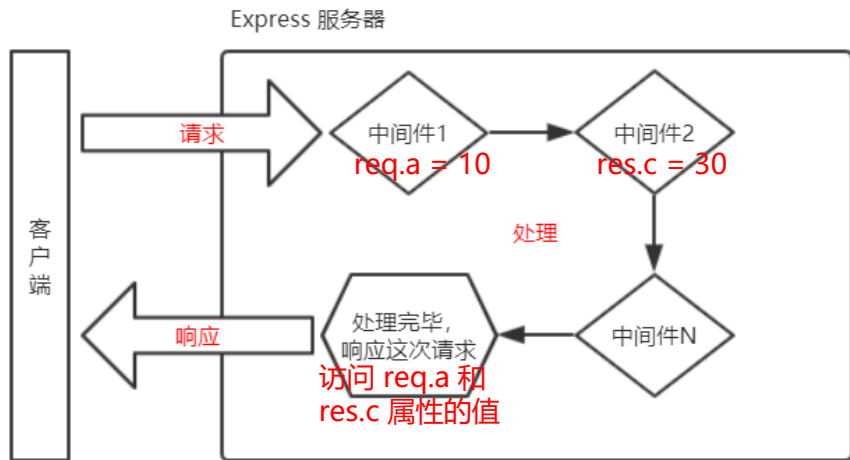
```
1 // 全局生效的中间件
2 app.use(function (req, res, next) {
3   console.log('这是一个最简单的中间件函数')
4   next()
5 })
```

3. Express 中间件

3.2 Express 中间件的初体验

4. 中间件的作用

多个中间件之间，**共享同一份 req 和 res**。基于这样的特性，我们可以在**上游**的中间件中，**统一**为 req 或 res 对象添加**自定义的属性或方法**，供**下游**的中间件或路由进行使用。





3. Express 中间件

3.2 Express 中间件的初体验

5. 定义多个全局中间件

可以使用 `app.use()` 连续定义多个全局中间件。客户端请求到达服务器之后，会按照中间件定义的先后顺序依次进行调用，示例代码如下：

```
1 app.use(function(req, res, next) { // 第1个全局中间件
2   console.log('调用了第1个全局中间件')
3   next()
4 })
5 app.use(function(req, res, next) { // 第2个全局中间件
6   console.log('调用了第2个全局中间件')
7   next()
8 })
9 app.get('/user', (req, res) => { // 请求这个路由，会依次触发上述两个全局中间件
10  res.send('Home page.')
11 })
```


3. Express 中间件

3.2 Express 中间件的初体验

6. 局部生效的中间件

不使用 `app.use()` 定义的中间件，叫做局部生效的中间件，示例代码如下：

```
1 // 定义中间件函数 mw1
2 const mw1 = function(req, res, next) {
3   console.log('这是中间件函数')
4   next()
5 }
6 // mw1 这个中间件只在 "当前路由中生效"，这种用法属于 "局部生效的中间件"
7 app.get('/', mw1, function(req, res) {
8   res.send('Home page.')
9 })
10 // mw1 这个中间件不会影响下面这个路由 ↓↓↓
11 app.get('/user', function(req, res) { res.send('User page.') })
```



3. Express 中间件

3.2 Express 中间件的初体验

7. 定义多个局部中间件

可以在路由中，通过如下两种等价的方式，使用多个局部中间件：

```
1 // 以下两种写法是"完全等价"的，可根据自己的喜好，选择任意一种方式进行使用
2 app.get('/', mw1, mw2, (req, res) => { res.send('Home page.') })
3 app.get('/', [mw1, mw2], (req, res) => { res.send('Home page.') })
```



3. Express 中间件

3.2 Express 中间件的初体验

8. 了解中间件的5个使用注意事项

- 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

3. Express 中间件

3.3 中间件的分类

为了方便大家理解和记忆中间件的使用，Express 官方把常见的中间件用法，分成了 5 大类，分别是：

- 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
- ④ Express 内置的中间件
- ⑤ 第三方的中间件

3. Express 中间件

3.3 中间件的分类

1. 应用级别的中间件

通过 `app.use()` 或 `app.get()` 或 `app.post()`，绑定到 `app` 实例上的中间件，叫做应用级别的中间件，代码示例如下：

```
1 // 应用级别的中间件（全局中间件）
2 app.use((req, res, next) => {
3   next()
4 })
5
6 // 应用级别的中间件（局部中间件）
7 app.get('/', mw1, (req, res) => {
8   res.send('Home page.')
9 })
```

3. Express 中间件

3.3 中间件的分类

2. 路由级别的中间件

绑定到 `express.Router()` 实例上的中间件，叫做路由级别的中间件。它的用法和应用级别中间件没有任何区别。只不过，应用级别中间件是绑定到 `app` 实例上，路由级别中间件绑定到 `router` 实例上，代码示例如下：

```
1 var app = express()
2 var router = express.Router()
3
4 // 路由级别的中间件
5 router.use(function (req, res, next) {
6   console.log('Time:', Date.now())
7   next()
8 })
9
10 app.use('/', router)
```

3. Express 中间件

3.3 中间件的分类

3. 错误级别的中间件

错误级别中间件的**作用**：专门用来捕获整个项目中发生的异常错误，从而防止项目异常崩溃的问题。

格式：错误级别中间件的 function 处理函数中，**必须有 4 个形参**，形参顺序从前到后，分别是 (err, req, res, next)。

```
1 app.get('/', function (req, res) {      // 1. 路由
2   throw new Error('服务器内部发生了错误! ') // 1.1 抛出一个自定义的错误
3   res.send('Home Page.')
4 })
5 app.use(function (err, req, res, next) { // 2. 错误级别的中间件
6   console.log('发生了错误: ' + err.message) // 2.1 在服务器打印错误消息
7   res.send('Error! ' + err.message)        // 2.2 向客户端响应错误相关的内容
8 })
```

注意：错误级别的中间件，
必须注册在所有路由之后！

3. Express 中间件

3.3 中间件的分类

3. 错误级别的中间件

错误级别中间件的 function 处理函数中，**必须有 4 个形参**，形参顺序从前到后，分别是 (err, req, res, next)。

```
1 app.get('/', function (req, res) {      // 1. 路由
2   throw new Error('服务器内部发生了错误! ') // 1.1 抛出一个自定义的错误
3   res.send('Home Page.')
4 })
5 app.use(function (err, req, res, next) { // 2. 错误级别的中间件
6   console.log('发生了错误: ' + err.message) // 2.1 在服务器打印错误消息
7   res.send('Error! ' + err.message)        // 2.2 向客户端响应错误相关的内容
8 })
```

注意：错误级别的中间件，
必须注册在所有路由之后！

3. Express 中间件

3.3 中间件的分类

4. Express内置的中间件

自 Express 4.16.0 版本开始，Express 内置了 3 个常用的中间件，极大的提高了 Express 项目的开发效率和体验：

- ① `express.static` 快速托管静态资源的内置中间件，例如：HTML 文件、图片、CSS 样式等（无兼容性）
- ② `express.json` 解析 JSON 格式的请求体数据（有兼容性，仅在 4.16.0+ 版本中可用）
- ③ `express.urlencoded` 解析 URL-encoded 格式的请求体数据（有兼容性，仅在 4.16.0+ 版本中可用）

```
1 // 配置解析 application/json 格式数据的内置中间件
2 app.use(express.json())
3 // 配置解析 application/x-www-form-urlencoded 格式数据的内置中间件
4 app.use(express.urlencoded({ extended: false })))
```

3. Express 中间件

3.3 中间件的分类

5. 第三方的中间件

非 Express 官方内置的，而是由第三方开发出来的中间件，叫做第三方中间件。在项目中，大家可以[按需下载并配置](#)第三方中间件，从而提高项目的开发效率。

例如：在 express@4.16.0 之前的版本中，经常使用 body-parser 这个第三方中间件，来解析请求体数据。使用步骤如下：

- ① 运行 `npm install body-parser` 安装中间件
- ② 使用 `require` 导入中间件
- ③ 调用 `app.use()` 注册并使用中间件

注意：Express 内置的 `express.urlencoded` 中间件，就是基于 `body-parser` 这个第三方中间件进一步封装出来的。

3. Express 中间件

3.4 自定义中间件

1. 需求描述与实现步骤

自己**手动模拟**一个类似于 `express.urlencoded` 这样的中间件，来**解析 POST 提交到服务器的表单数据**。

实现步骤：

- ① 定义中间件
- ② 监听 req 的 data 事件
- ③ 监听 req 的 end 事件
- ④ 使用 `querystring` 模块解析请求体数据
- ⑤ 将解析出来的数据对象挂载为 `req.body`
- ⑥ 将自定义中间件封装为模块

3. Express 中间件

3.4 自定义中间件

2. 定义中间件

使用 `app.use()` 来定义全局生效的中间件，代码如下：

```
1 app.use(function(req, res, next) {  
2     // 中间件的业务逻辑  
3 })
```

3. Express 中间件

3.4 自定义中间件

3. 监听 req 的 data 事件

在中间件中，需要监听 req 对象的 data 事件，来获取客户端发送到服务器的数据。

如果数据量比较大，无法一次性发送完毕，则客户端会**把数据切割后，分批发送到服务器**。所以 data 事件可能会触发多次，每一次触发 data 事件时，**获取到数据只是完整数据的一部分**，需要手动对接收到的数据进行拼接。

```
1  // 定义变量，用来存储客户端发送过来的请求体数据
2  let str = ''
3  // 监听 req 对象的 data 事件（客户端发送过来的新的请求体数据）
4  req.on('data', (chunk) => {
5      // 拼接请求体数据，隐式转换为字符串
6      str += chunk
7  })
```



3. Express 中间件

3.4 自定义中间件

4. 监听 req 的 end 事件

当请求体数据接收完毕之后，会自动触发 req 的 end 事件。

因此，我们可以在 req 的 end 事件中，拿到并处理完整的请求体数据。示例代码如下：

```
1 // 监听 req 对象的 end 事件 (请求体发送完毕后自动触发)
2 req.on('end', () => {
3   // 打印完整的请求体数据
4   console.log(str)
5   // TODO: 把字符串格式的请求体数据，解析成对象格式
6 })
```



3. Express 中间件

3.4 自定义中间件

5. 使用 querystring 模块解析请求体数据

Node.js 内置了一个 `querystring` 模块，专门用来处理查询字符串。通过这个模块提供的 `parse()` 函数，可以轻松把查询字符串，解析成对象的格式。示例代码如下：

```
1 // 导入处理 querystring 的 Node.js 内置模块
2 const qs = require('querystring')
3
4 // 调用 qs.parse() 方法, 把查询字符串解析为对象
5 const body = qs.parse(str)
```



3. Express 中间件

3.4 自定义中间件

6. 将解析出来的数据对象挂载为 **req.body**

上游的中间件和下游的中间件及路由之间，**共享同一份 req 和 res**。因此，我们可以将解析出来的数据，挂载为 req 的自定义属性，命名为 **req.body**，供下游使用。示例代码如下：

```
1 req.on('end', () => {
2   const body = qs.parse(str) // 调用 qs.parse() 方法，把查询字符串解析为对象
3   req.body = body           // 将解析出来的请求体对象，挂载为 req.body 属性
4   next()                   // 最后，一定要调用 next() 函数，执行后续的业务逻辑
5 })
6 })
```


3. Express 中间件

3.4 自定义中间件

7. 将自定义中间件封装为模块

为了优化代码的结构，我们可以把自定义的中间件函数，封装为独立的模块，示例代码如下：

```
1 // custom-body-parser.js 模块中的代码
2 const qs = require('querystring')
3 function bodyParser(req, res, next) { /* 省略其它代码 */ }
4 module.exports = bodyParser // 向外导出解析请求体数据的中间件函数
5
6 // -----分割线-----
7
8 // 1. 导入自定义的中间件模块
9 const myBodyParser = require('custom-body-parser')
10 // 2. 注册自定义的中间件模块
11 app.use(myBodyParser)
```

目录 Contents

- ◆ 初识 Express
- ◆ Express 路由
- ◆ Express 中间件
- ◆ 使用 Express 写接口

4. 使用 Express 写接口

4.1 创建基本的服务器

```
1 // 导入 express 模块
2 const express = require('express')
3 // 创建 express 的服务器实例
4 const app = express()
5
6 // write your code here...
7
8 // 调用 app.listen 方法, 指定端口号并启动web服务器
9 app.listen(80, function () {
10   console.log('Express server running at http://127.0.0.1')
11 })
```

4. 使用 Express 写接口

4.2 创建 API 路由模块

```
1 // apiRouter.js 【路由模块】
2 const express = require('express')
3 const apiRouter = express.Router()
4
5 // bind your router here...
6
7 module.exports = apiRouter
8
9 // -----
10
11 // app.js 【导入并注册路由模块】
12 const apiRouter = require('./apiRouter.js')
13 app.use('/api', apiRouter)
```

4. 使用 Express 写接口

4.3 编写 GET 接口

```
1 apiRouter.get('/get', (req, res) => {  
2   // 1. 获取到客户端通过查询字符串, 发送到服务器的数据  
3   const query = req.query  
4   // 2. 调用 res.send() 方法, 把数据响应给客户端  
5   res.send({  
6     status: 0,           // 状态, 0 表示成功, 1 表示失败  
7     msg: 'GET请求成功! ', // 状态描述  
8     data: query          // 需要响应给客户端的具体数据  
9   })  
10 })
```

4. 使用 Express 写接口

4.4 编写 POST 接口

```
1 apiRouter.post('/post', (req, res) => {  
2   // 1. 获取客户端通过请求体, 发送到服务器的 URL-encoded 数据  
3   const body = req.body  
4   // 2. 调用 res.send() 方法, 把数据响应给客户端  
5   res.send({  
6     status: 0,           // 状态, 0 表示成功, 1 表示失败  
7     msg: 'POST请求成功! ', // 状态描述消息  
8     data: body           // 需要响应给客户端的具体数据  
9   })  
10 })
```

注意: 如果要获取 URL-encoded 格式的请求体数据, 必须配置中间件 `app.use(express.urlencoded({ extended: false }))`

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

1. 接口的跨域问题

刚才编写的 GET 和 POST接口，存在一个很严重的问题：**不支持跨域请求**。

解决接口跨域问题的方案主要有两种：

- ① **CORS** (主流的解决方案，**推荐使用**)
- ② **JSONP** (有缺陷的解决方案：只支持 GET 请求)

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

2. 使用 cors 中间件解决跨域问题

cors 是 Express 的一个第三方中间件。通过安装和配置 cors 中间件，可以很方便地解决跨域问题。

使用步骤分为如下 3 步：

- ① 运行 `npm install cors` 安装中间件
- ② 使用 `const cors = require('cors')` 导入中间件
- ③ 在路由之前调用 `app.use(cors())` 配置中间件

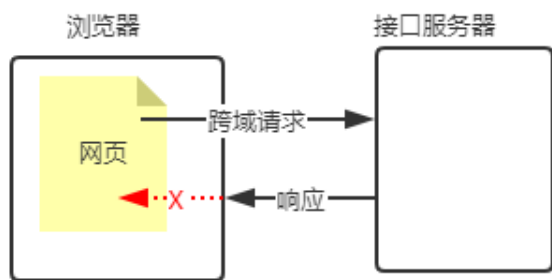
4. 使用 Express 写接口

4.5 CORS 跨域资源共享

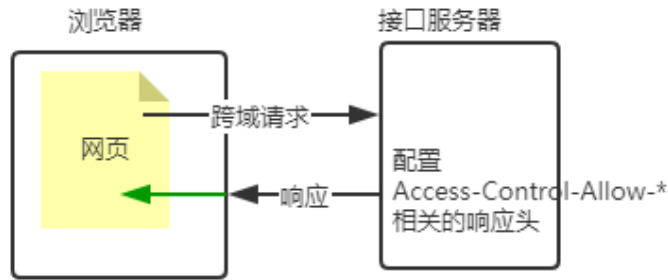
3. 什么是 CORS

CORS (Cross-Origin Resource Sharing, 跨域资源共享) 由一系列 HTTP 响应头组成, 这些 HTTP 响应头决定浏览器是否阻止前端 JS 代码跨域获取资源。

浏览器的同源安全策略默认会阻止网页“跨域”获取资源。但如果接口服务器配置了 CORS 相关的 HTTP 响应头, 就可以解除浏览器端的跨域访问限制。



响应的结果被浏览器拦截, 网页无法获取到跨域响应的数据



在服务器端, 配置cors相关的响应头, 从而解除浏览器端的跨域访问限制

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

4. CORS 的注意事项

- ① CORS 主要在服务器端进行配置。客户端浏览器无须做任何额外的配置，即可请求开启了 CORS 的接口。
- ② CORS 在浏览器中有兼容性。只有支持 XMLHttpRequest Level2 的浏览器，才能正常访问开启了 CORS 的服务端接口（例如：IE10+、Chrome4+、FireFox3.5+）。

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

5. CORS 响应头部 - Access-Control-Allow-Origin

响应头部中可以携带一个 **Access-Control-Allow-Origin** 字段，其语法如下：

```
1 Access-Control-Allow-Origin: <origin> | *
```

其中，origin 参数的值指定了允许访问该资源的外域 URL。

例如，下面的字段值将**只允许**来自 `http://itcast.cn` 的请求：

```
1 res.setHeader('Access-Control-Allow-Origin', 'http://itcast.cn')
```

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

5. CORS 响应头部 - Access-Control-Allow-Origin

如果指定了 Access-Control-Allow-Origin 字段的值为通配符`*`，表示允许来自任何域的请求，示例代码如下：

```
1 res.setHeader('Access-Control-Allow-Origin', '*')
```

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

6. CORS 响应头部 - Access-Control-Allow-Headers

默认情况下，CORS 仅支持客户端向服务器发送如下的 9 个请求头：

Accept、Accept-Language、Content-Language、DPR、Downlink、Save-Data、Viewport-Width、Width、Content-Type（值仅限于 text/plain、multipart/form-data、application/x-www-form-urlencoded 三者之一）

如果客户端向服务器发送了额外的请求头信息，则需要在服务器端，通过 Access-Control-Allow-Headers 对额外的请求头进行声明，否则这次请求会失败！

```
1 // 允许客户端额外向服务器发送 Content-Type 请求头和 X-Custom-Header 请求头
2 // 注意：多个请求头之间使用英文的逗号进行分割
3 res.setHeader('Access-Control-Allow-Headers', 'Content-Type, X-Custom-Header')
```

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

7. CORS 响应头部 - Access-Control-Allow-Methods

默认情况下，CORS 仅支持客户端发起 GET、POST、HEAD 请求。

如果客户端希望通过 PUT、DELETE 等方式请求服务器的资源，则需要在服务器端，通过 Access-Control-Allow-Methods 来指明实际请求所允许使用的 HTTP 方法。

示例代码如下：

```
1 // 只允许 POST、GET、DELETE、HEAD 请求方法
2 res.setHeader('Access-Control-Allow-Methods', 'POST, GET, DELETE, HEAD')
3 // 允许所有的 HTTP 请求方法
4 res.setHeader('Access-Control-Allow-Methods', '*')
```

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

8. CORS请求的分类

客户端在请求 CORS 接口时，根据请求方式和请求头的不同，可以将 CORS 的请求分为两大类，分别是：

- ① 简单请求
- ② 预检请求

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

9. 简单请求

同时满足以下两大条件的请求，就属于简单请求：

- ① **请求方式**：GET、POST、HEAD 三者之一
- ② **HTTP 头部信息**不超过以下几种字段：**无自定义头部字段**、Accept、Accept-Language、Content-Language、DPR、Downlink、Save-Data、Viewport-Width、Width、Content-Type（只有三个值application/x-www-form-urlencoded、multipart/form-data、text/plain）

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

10. 预检请求

只要符合以下任何一个条件的请求，都需要进行预检请求：

- ① 请求方式为 GET、POST、HEAD 之外的请求 Method 类型
- ② 请求头中包含自定义头部字段
- ③ 向服务器发送了 application/json 格式的数据

在浏览器与服务器正式通信之前，浏览器会先发送 OPTION 请求进行预检，以获知服务器是否允许该实际请求，所以这一次的 OPTION 请求称为“预检请求”。服务器成功响应预检请求后，才会发送真正的请求，并且携带真实数据。

4. 使用 Express 写接口

4.5 CORS 跨域资源共享

11. 简单请求和预检请求的区别

简单请求的特点：客户端与服务器之间**只会发生一次请求**。

预检请求的特点：客户端与服务器之间会发生两次请求，**OPTION 预检请求成功之后，才会发起真正的请求**。

4. 使用 Express 写接口

4.6 JSONP 接口

1. 回顾 JSONP 的概念与特点

概念：浏览器端通过 `<script>` 标签的 `src` 属性，请求服务器上的数据，同时，服务器返回一个函数的调用。这种请求数据的方式叫做 JSONP。

特点：

- ① JSONP 不属于真正的 Ajax 请求，因为它没有使用 XMLHttpRequest 这个对象。
- ② JSONP 仅支持 GET 请求，不支持 POST、PUT、DELETE 等请求。

4. 使用 Express 写接口

4.6 JSONP 接口

2. 创建 JSONP 接口的注意事项

如果项目中已经配置了 CORS 跨域资源共享，为了防止冲突，必须在配置 CORS 中间件之前声明 JSONP 的接口。否则 JSONP 接口会被处理成开启了 CORS 的接口。示例代码如下：

```
1 // 优先创建 JSONP 接口【这个接口不会被处理成 CORS 接口】
2 app.get('/api/jsonp', (req, res) => { })
3
4 // 再配置 CORS 中间件【后续的所有接口，都会被处理成 CORS 接口】
5 app.use(cors())
6
7 // 这是一个开启了 CORS 的接口
8 app.get('/api/get', (req, res) => { })
```

4. 使用 Express 写接口

4.6 JSONP 接口

3. 实现 JSONP 接口的步骤

- ① 获取客户端发送过来的回调函数的名字
- ② 得到要通过 JSONP 形式发送给客户端的数据
- ③ 根据前两步得到的数据，拼接出一个函数调用的字符串
- ④ 把上一步拼接得到的字符串，响应给客户端的 `<script>` 标签进行解析执行

4. 使用 Express 写接口

4.6 JSONP 接口

4. 实现 JSONP 接口的具体代码

```
1 app.get('/api/jsonp', (req, res) => {
2   // 1. 获取客户端发送过来的回调函数的名字
3   const funcName = req.query.callback
4   // 2. 得到要通过 JSONP 形式发送给客户端的数据
5   const data = { name: 'zs', age: 22 }
6   // 3. 根据前两步得到的数据, 拼接出一个函数调用的字符串
7   const scriptStr = `${funcName}(${JSON.stringify(data)})`
8   // 4. 把上一步拼接得到的字符串, 响应给客户端的 <script> 标签进行解析执行
9   res.send(scriptStr)
10 })
```

4. 使用 Express 写接口

4.6 JSONP 接口

5. 在网页中使用 jQuery 发起 JSONP 请求

调用 \$.ajax() 函数，提供 JSONP 的配置选项，从而发起 JSONP 请求，示例代码如下：

```
1 $('#btnJSONP').on('click', function () {  
2     $.ajax({  
3         method: 'GET',  
4         url: 'http://127.0.0.1/api/jsonp',  
5         dataType: 'jsonp', // 表示要发起 JSONP 的请求  
6         success: function (res) {  
7             console.log(res)  
8         }  
9     })  
10 })
```



传智播客旗下高端IT教育品牌