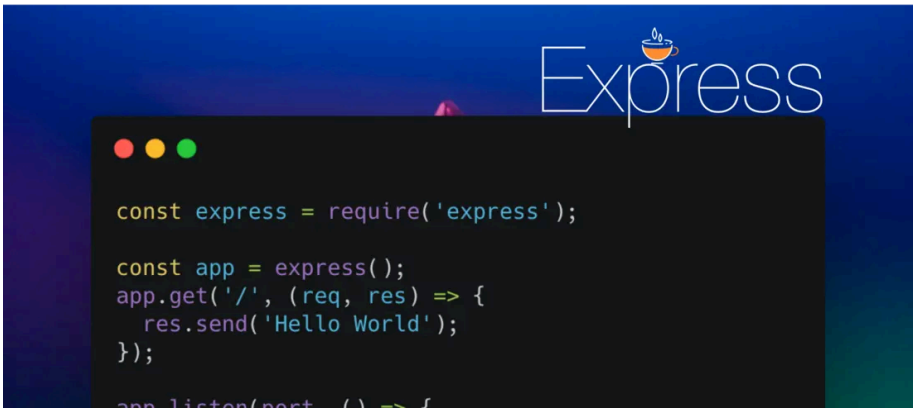
一杯茶的时间，上手 Express 框架开发

一只图雀 2020-03-16 阅读 13 分钟



Node.js 已经成为 Web 后台开发圈一股不容忽视的力量，凭借其良好的异步性能、丰富的 npm 库以及 JavaScript 语言方面的优势，已经成为了很多大公司开发其后台架构的重要技术之一，而 Express 框架则是其中知名度最高、也是最受欢迎的后端开发框架。在这篇教程中，你将了解 Express 在 Node 内置 http 模块的基础上做了怎样的封装，并掌握路由和中间件这两个关键概念，学习和使用模板引擎、静态文件服务、错误处理和 JSON API，最终开发出一个简单的个人简历网站。

> 此教程属于 Node.js 后端工程师学习路线的一部分，欢迎来 Star 一波，鼓励我们继续创作出更好的教程，持续更新中~。

旧时代：用内置 http 模块实现一个服务器

自从 Ryan Dahl 在 2009 年的 JSConf 正式推出 Node.js 平台后，这门技术的使用率就如同坐了火箭一般迅速上升，成为了最受喜爱的后端开发平台之一，而 Express 则是其中最为耀眼的 Web 框架。在正式开始这篇教程之前，我们将列举一下这篇教程所需要的预备知识、所用技术和学习目标。

预备知识

本教程假定你已经知道了：

- JavaScript 语言基础知识（包括一些常用的 ES6+ 语法）
- Node.js 基础知识，特别是异步编程（这篇教程主要用到的是回调函数）和 Node 模块机制，还有 npm 的基本使用，可以参考这篇教程进行学习
- HTTP 协议基础知识，浏览器和服务器之间是如何互动的

所用技术

- **Node.js**：8.x 及以上
- **npm**：6.x 及以上
- **Express.js**：4.x

学习目标

读完这篇教程后，你将学会

- Express 框架的两大核心概念：路由和中间件
- 用 Nodemon 加速开发迭代
- 使用模板引擎渲染页面，并接入 Express 框架中
- 使用 Express 的静态文件服务
- 编写自定义的错误处理函数
- 实现一个简单的 JSON API 端口
- 通过子路由拆分逻辑，实现模块化

> **注意**
> 虽然数据库是后端开发中非常重要的环节，但 Express 并不内置处理数据库的模块，需要额外的第三方库提供支持。这篇教程将重点放在了 Express 相关的概念讲解上，因此不会涉及数据库的开发。在学完这篇教程后，你可以浏览 Express 相关的进阶教程。

用内置 http 模块创建服务器

在讲解 Express 之前，我们先了解一下怎么用 Node.js 内置的 http 模块来实现一个服务器，从而能够更好地了解 Express 对底层的 Node 代码做了哪些抽象和封装。如果你还没有安装 Node.js，可以去官方网站下载并安装。

我们将实现一个个人简历网站。创建一个文件夹 express_resume，并进入其中：

```
mkdir express_resume && cd express_resume
```

创建 server.js 文件，代码如下：

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

const hostname = 'localhost';
const port = 3000;

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

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

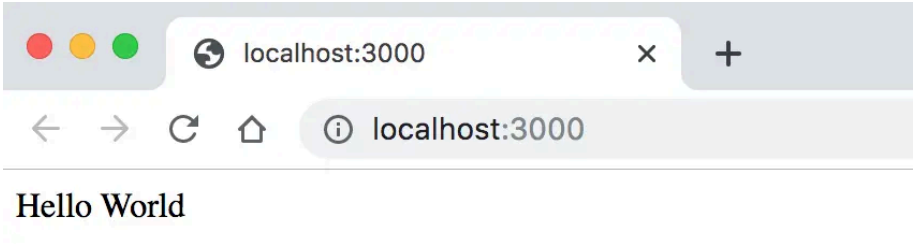
如果你熟悉 Node.js，上面的代码含义很清晰：

1. 导入 http 模块
2. 指定服务器的主机名 `hostname` 和端口号 `port`
3. 用 `http.createServer` 创建 HTTP 服务器，参数为一个回调函数，接受一个请求对象 `req` 和响应对象 `res`，并在回调函数中写入响应内容（状态码 200，类型为 HTML 文档，内容为 `Hello World`）
4. 在指定的端口开启服务器

最后运行 server.js：

```
node server.js
```

用浏览器打开 localhost:3000，可以看到 Hello World 的提示：



可以发现，直接用内置的 http 模块去开发服务器有以下明显的弊端：

- **需要写很多底层代码**——例如手动指定 HTTP 状态码和头部字段，最终返回内容。如果我们需要开发更复杂的功能，涉及到多种状态码和头部信息（例如用户鉴权），这样的手动管理模式非常不方便
- **没有专门的路由机制**——路由是服务器最重要的功能之一，通过路由才能根据客户端的不同请求 URL 及 HTTP 方法来返回相应内容。但是上面这段代码只能在 `http.createServer` 的回调函数中通过判断请求 `req` 的内容才能实现路由功能，搭建大型应用时力不从心

由此就引出了 Express 对内置 http 的两大封装和改进：

- 更强大的请求（Request）和响应（Response）对象，添加了很多实用方法
- 灵活方便的路由的定义与解析，能够很方便地进行代码拆分

接下来，我们将开始用 Express 来开发 Web 服务器！

新时代：用 Express 搭建服务器

在第一步中，我们把服务器放在了一个 JS 文件中，也就是一个 Node 模块。从现在开始，我们将把这个项目变成一个 npm 项目。输入以下命令创建 npm 项目：

```
npm init
```

接着你可以一路回车下去（当然也可以仔细填），就会发现 package.json 文件已经创建好了。然后添加 Express 项目依赖：

```
npm install express
```

在开始用 Express 改写上面的服务器之前，我们先介绍一下上面提到的**两大封装与改进**。

更强大的 Request 和 Response 对象

首先是 Request 请求对象，通常我们习惯用 `req` 变量来表示。下面列举一些 `req` 上比较重要的成员（如果不知道是什么也没关系哦）：

- `req.body`：客户端请求体的数据，可能是表单或 JSON 数据
- `req.params`：请求 URI 中的路径参数
- `req.query`：请求 URI 中的查询参数
- `req.cookies`：客户端的 cookies

然后是 Response 响应对象，通常用 `res` 变量来表示，可以执行一系列响应操作，例如：

```
// 发送一串 HTML 代码
res.send('HTML String');

// 发送一个文件
res.sendFile('file.zip');

// 渲染一个模板引擎并发送
res.render('index');
```

Response 对象上的操作非常丰富，并且还可以链式调用：

```
// 设置状态码为 404，并返回 Page Not Found 字符串
res.status(404).send('Page Not Found');
```

> **提示**
> 在这里我们并没有简单地列举 Request 和 Response 的全部 API，因为图雀社区的理念是——从实战中学习和深化理解，拒绝枯燥的 API 记忆！

路由机制

客户端（包括 Web 前端、移动端等等）向服务器发起请求时包括两个元素：**路径**（URI）以及 **HTTP 请求方法**（包括 GET、POST 等等）。路径和请求方法合起来一般被称为 API 端点（Endpoint）。而服务器根据客户端访问的端点选择相应处理逻辑的机制就叫做路由。

在 Express 中，定义路由只需按下面这样的形式：

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

其中：

- `app` 就是一个 `express` 服务器对象
- `METHOD` 可以是任何**小写**的 HTTP 请求方法，包括 `get`、`post`、`put`、`delete` 等等
- `PATH` 是客户端访问的 URI，例如 `/` 或 `/about`
- `HANDLER` 是路由被触发时的回调函数，在函数中可以执行相应的业务逻辑

nodemon 加速开发

Nodemon 是一款颇受欢迎的开发服务器，能够检测工作区代码的变化，并自动重启。通过以下命令安装 nodemon：

```
npm install nodemon --save-dev
```

这里我们将 nodemon 安装为开发依赖 `devDependencies`，因为仅仅只有在开发时才需要用到。同时我们在 package.json 中加入 `start` 命令，代码如下：

```
{
  "name": "express_resume",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.2"
  }
}
```

正式实现

到了动手的时候了，我们用 Express 改写上面的服务器，代码如下：

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

const hostname = 'localhost';
const port = 3000;

const app = express();
app.get('/', (req, res) => {
  res.send('Hello World');
});

app.listen(port, () => {
```

```
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

在上面的代码中，我们首先用 `express()` 函数创建一个 Express 服务器对象，然后用上面提到的路由定义方法 `app.get` 定义了主页 `/` 的路由，最后同样调用 `listen` 方法开启服务器。

从这一步开始，我们运行 `npm start` 命令即可开启服务器，并且同样可以看到 Hello World 的内容，但是代码却简单明了了不少。

> **提示**
> 在运行 `npm start` 之后，可以让服务器一直打开着，编辑代码并保存后，Nodemon 就会自动重启服务器，运行最新的代码。

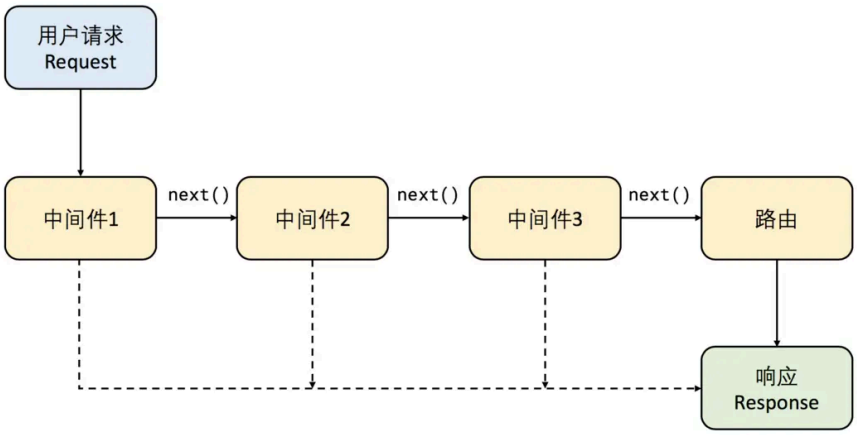
编写第一个中间件

接下来我们开始讲解 Express 第二个重要的概念：**中间件**（Middleware）。

理解中间件

中间件并不是 Express 独有的概念。相反，它是一种广为使用的软件工程概念（甚至已经延伸到了其他行业），是指**将具体的业务逻辑和底层逻辑解耦的组件**（可查看这个讨论）。换句话说，中间件就是能够适用多个应用场景、可复用性良好的代码。

Express 的简化版中间件流程如下图所示：



首先客户端向服务器发起请求，然后服务器依次执行每个中间件，最后到达路由，选择相应的逻辑来执行。

> **提示**
> 这个是一个简化版的流程描述，目的是便于你对中间件有个初步的认识，在后面的章节中我们将进一步完善这一流程。

有两点需要特别注意：

- 中间件是**按顺序执行**的，因此在配置中间件时顺序非常重要，不能弄错
- 中间件在执行内部逻辑的时候可以选择将请求传递给下一个中间件，也可以直接返回用户响应

Express 中间件的定义

在 Express 中，中间件就是一个函数：

```
function someMiddleware(req, res, next) {
  // 自定义逻辑
  next();
}
```

三个参数中，`req` 和 `res` 就是前面提到的 Request 请求对象和 Response 响应对象；而 `next` 函数则用来触发下一个中间件的执行。

> **注意**
> 如果忘记在中间件中调用 `next` 函数，并且又不直接返回响应时，服务器会直接卡在这个中间件不会继续执行下去哦！

在 Express 使用中间件有两种方式：**全局中间件**和**路由中间件**。

全局中间件

通过 `app.use` 函数就可以注册中间件，并且此中间件会在用户发起**任何请求**都可能会执行，例如：

```
app.use(someMiddleware);
```

路由中间件

通过在路由定义时注册中间件，此中间件只会在用户访问该路由对应的 URI 时执行，例如：

```
app.get('/middleware', someMiddleware, (req, res) => {
  res.send('Hello World');
```

```
});
```

那么用户只有在访问 `/middleware` 时，定义的 `someMiddleware` 中间件才会被触发，访问其他路径时不会触发。

编写中间件

接下来我们就开始实现第一个 Express 中间件。功能很简单，就是在终端打印客户端的访问时间、 HTTP 请求方法和 URI，名为 `loggingMiddleware`。代码如下：

```
// ...

const app = express();

function loggingMiddleware(req, res, next) {
  const time = new Date();
  console.log(`[${time.toLocaleString()}] ${req.method} ${req.url
  next();
}

app.use(loggingMiddleware);

app.get('/', (req, res) => {
  res.send('Hello World');
});

// ...
```

> **注意**
> 在中间件中写 `console.log` 语句是比较糟糕的做法，因为 `console.log`（包括其他同步的代码）都会阻塞 Node.js 的异步事件循环，降低服务器的吞吐率。在实际生产中，推荐使用第三方优秀的日志中间件，例如 morgan、winston 等等。

运行服务器，然后用浏览器尝试访问各个路径。这里我访问了首页（localhost:3000）和 `/hello`（localhost:3000/hello，浏览器应该看到的是 404），可以看到控制台相应的输出：

```
[11/28/2019, 3:54:05 PM] GET /
[11/28/2019, 3:54:11 PM] GET /hello
```

这里为了让你初步理解中间件的概念，我们只实现了一个功能很简单的中间件。实际上，中间件不仅可以读取 `req` 对象上的各个属性，还可以添加

新的属性或修改已有的属性（后面的中间件和路由函数都可以获取），能够很方便地实现一些复杂的业务逻辑（例如用户鉴权）。

用模板引擎渲染页面

最后，我们的网站要开始展示一些实际内容了。Express 对当今主流的模板引擎（例如 Pug、Handlebars、EJS 等等）提供了很好的支持，可以做到两行代码接入。

> **提示**
> 如果你不了解模板引擎，不用担心，这篇教程几乎不需要用到它的高级功能，你只需理解成一个“升级版的 HTML 文档”即可。

这篇教程将使用 Handlebars 作为模板引擎。首先添加 npm 包：

```
npm install hbs
```

创建 views 文件夹，用于放置所有的模板。然后在其中创建首页模板 index.hbs，代码如下：

```
<h1>个人简历</h1>
<p>我是一只小小的图雀，渴望学习技术，磨炼实战本领。</p>
<a href="/contact">联系方式</a>
```

创建联系页面模板 contact.hbs，代码如下：

```
<h1>联系方式</h1>
<p>QQ：1234567</p>
<p>微信：一只图雀</p>
<p>邮箱：mrc@tuture.co</p>
```

最后便是在 server.js 中配置和使用模板。配置模板的代码非常简单：

```
// 指定模板存放目录
app.set('views', '/path/to/templates');

// 指定模板引擎为 Handlebars
app.set('view engine', 'hbs');
```

在使用模板时，只需在路由函数中调用 `res.render` 方法即可：

```
// 渲染名称为 hello.hbs 的模板
res.render('hello');
```

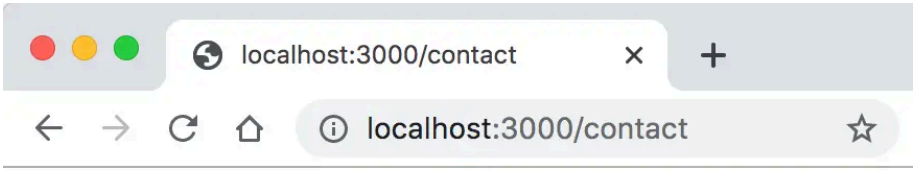
修改后的 server.js 代码如下：

```
// ...

const app = express();

app.set('views', 'views');
app.set('view engine', 'hbs');

// 定义和使用 loggingMiddleware 中间件 ...

app.get('/', (req, res) => {
  res.render('index');
});

app.get('/contact', (req, res) => {
  res.render('contact');
})

// ...
```

注意在上面的代码中，我们添加了 `GET /contact` 的路由定义。

最后，我们再次运行服务器，访问我们的主页，可以看到：

点击"联系方式"，跳转到相应页面：



联系方式

QQ：1234567

微信：一只图雀

邮箱：mrc@tuture.co

添加静态文件服务

通常网站需要提供静态文件服务，例如图片、CSS 文件、JS 文件等等，而 Express 已经自带了静态文件服务中间件 `express.static`，使用起来非常方便。

例如，我们添加静态文件中间件如下，并指定静态资源根目录为 `public`：

```
// ...

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

app.get('/', (req, res) => {
  res.render('index');
});

// ...
```

假设项目的 public 目录里面有这些静态文件：

```
public
├── css
│   └── style.css
└── img
    └── tuture-logo.png
```

就可以分别通过以下路径访问：

```
http://localhost:3000/css/style.css
http://localhost:3000/img/tuture-logo.png
```

样式文件 public/css/style.css 的代码如下（直接复制粘贴即可）：

```
body {
  text-align: center;
}

h1 {
  color: blue;
}

img {
  border: 1px dashed grey;
}

a {
  color: blueviolet;
}
```

图片文件可通过这个 GitHub 上的链接下载，然后下载到 public/img 目录中。当然，你也可以使用自己的图片，记得在模板中替换相应的链接就可以了。

在首页模板 views/index.hbs 中加入 CSS 样式表和图片：

```
<link rel="stylesheet" href="/css/style.css" />

<h1>个人简历</h1>
<img src="/img/tuture-logo.png" alt="Logo" />
<p>我是一只小小的图雀，渴望学习技术，磨炼实战本领。</p>
<a href="/contact">联系方式</a>
```

在联系模板 views/contact.hbs 中加入样式表：

```
<link rel="stylesheet" href="/css/style.css" />

<h1>联系方式</h1>
<p>QQ：1234567</p>
<p>微信：一只图雀</p>
<p>邮箱：mrc@tuture.co</p>
```

再次运行服务器，并访问我们的网站。首页如下：

联系我们页面如下：

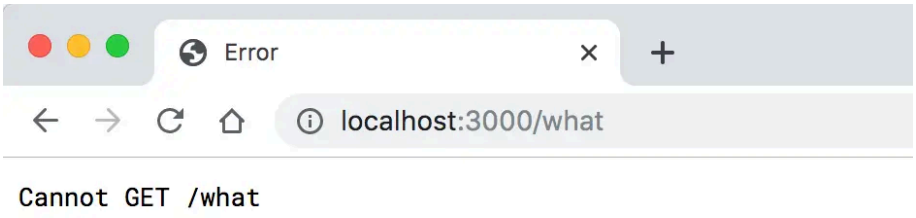
可以看到样式表和图片都成功加载出来了！

处理 404 和服务器错误

人有悲欢离合，月有阴晴圆缺，服务器也有出错的时候。HTTP 错误一般分为两大类：

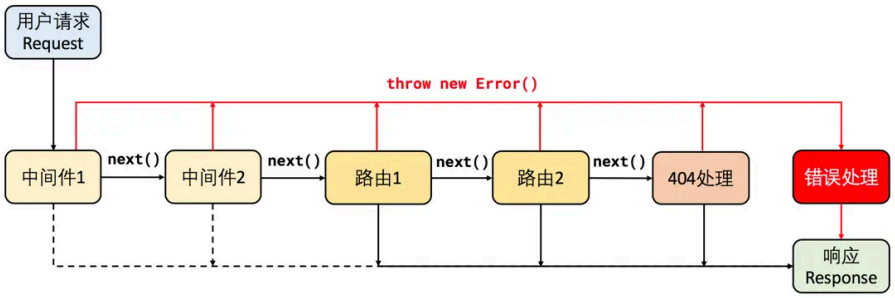
- 客户端方面的错误（状态码 4xx），例如访问了不存在的页面（404）、权限不够（403）等等
- 服务器方面的错误（状态码 5xx），例如服务器内部出现错误（500）或网关错误（503）等等

如果你打开服务器，访问一个不存在的路径，例如 `localhost:3000/what`，就会出现这样的页面：



很显然，这样的用户体验是很糟糕的。

在这一节中，我们将讲解如何在 Express 框架中处理 404（页面不存在）及 500（服务器内部错误）。在此之前，我们要完善一下 Express 中间件的运作流程，如下图所示：



这张示意图和之前的图有两点重大区别：

- 每个路由定义本质上是一个**中间件**（更准确地说是一个**中间件容器**，可包含多个中间件），当 URI 匹配成功时直接返回响应，匹配失败时继续执行下一个路由
- 每个中间件（包括路由）不仅可以调用 `next` 函数向下传递、直接返回响应，还可以**抛出异常**

从这张图就可以很清晰地看出怎么实现 404 和服务器错误的处理了：

- 对于 404，只需在所有路由之后再加一个中间件，用来接收所有路由均匹配失败的请求
- 对于错误处理，前面所有中间件抛出异常时都会进入错误处理函数，可以使用 Express 自带的，也可以自定义。

处理 404

在 Express 中，可以通过中间件的方式处理访问不存在的路径：

```
app.use('*', (req, res) => {
  // ...
});
```

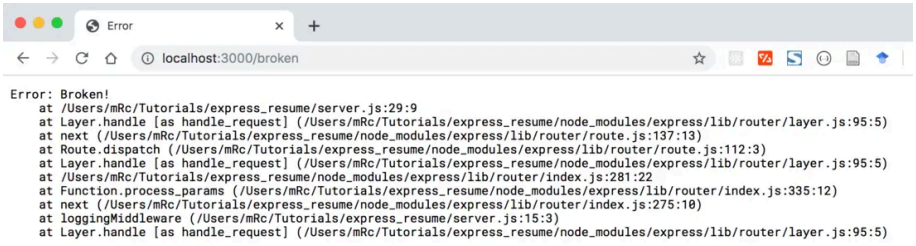
`*` 表示匹配任何路径。将此中间件放在所有路由后面，即可捕获所有访问路径均匹配失败的请求。

处理内部错误

Express 已经自带了错误处理机制，我们先来体验一下。在 server.js 中添加下面这条"坏掉"的路由（模拟现实中出错的情形）：

```
app.get('/broken', (req, res) => {
  throw new Error('Broken!');
});
```

然后开启服务器，访问 `localhost:3000/broken`：



> **危险！**
> 服务器直接返回了出错的调用栈！很明显，向用户返回这样的调用栈不仅体验糟糕，而且大大增加了被攻击的风险。

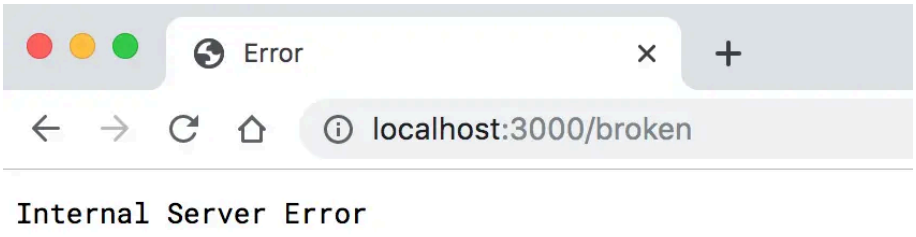
实际上，Express 的默认错误处理机制可以通过设置 `NODE_ENV` 来进行切换。我们将其设置为生产环境 `production`，再开启服务器。如果你在 Linux、macOS 或 Windows 下的 Git Bash 环境中，可以运行以下命令：

```
NODE_ENV=production node server.js
```

如果你在 Windows 下的命令行，运行以下命令：

```
set NODE_ENV=production
node server.js
```

这时候访问 `localhost:3000/broken` 就会直接返回 Internal Server Error（服务器内部错误），不会显示任何错误信息：



体验还是很不好，更理想的情况是能够返回一个友好的自定义页面。这可以通过 Express 的自定义错误处理函数来解决，错误处理函数的形式如下：

```
function (err, req, res, next) {
  // 处理错误逻辑
}
```

和普通的中间件函数相比，多了第一个参数，也就是 `err` 异常对象。

实现自定义处理逻辑

通过上面的讲解，实现自定义的 404 和错误处理逻辑也就非常简单了。在 server.js 所有路由的后面添加如下代码：

```
// 中间件和其他路由 ...

app.use('*', (req, res) => {
  res.status(404).render('404', { url: req.originalUrl });
});

app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).render('500');
});

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

> **提示**
> 在编写处理 404 的逻辑时，我们用到了模板引擎中的变量插值功能。具体而言，在 `res.render` 方法中将需要传给模板的数据作为第二个参数（例如这里的 `{ url: req.originalUrl }` 传入了用户访问的路径），在模板中就可以通过 `{{ url }}` 获取数据了。

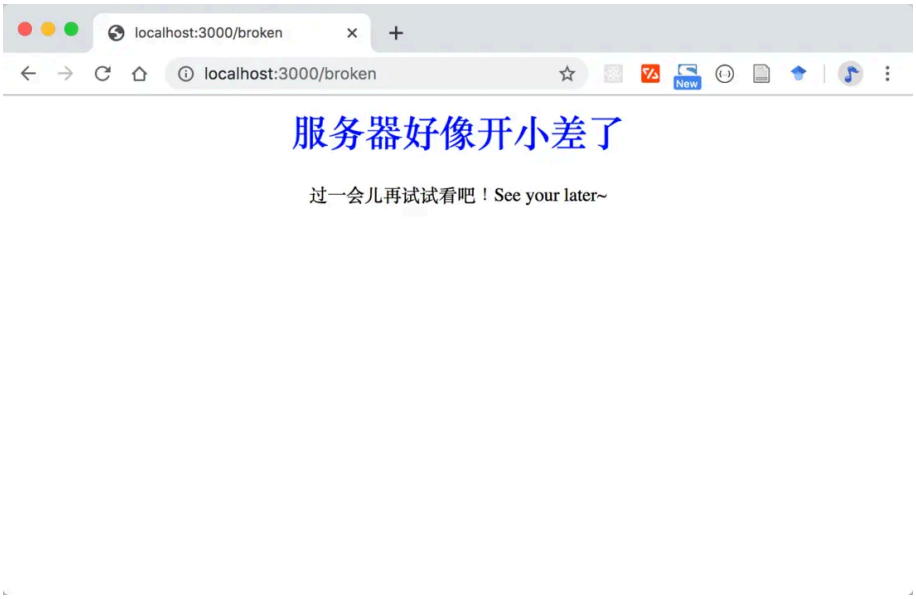
404 和 500 的模板代码分别如下：

```
<link rel="stylesheet" href="/css/style.css" />

<h1>找不到你要的页面了！ </h1>
<p>你所访问的路径 {{ url }} 不存在</p>
```

```
<link rel="stylesheet" href="/css/style.css" />

<h1>服务器好像开小差了</h1>
<p>过一会儿再试试看吧！See your later~</p>
```

再次运行服务器，访问一个不存在的路径：

访问 `localhost:3000/broken`：



体验很不错!

三行代码实现 JSON API

在这篇教程的最后，我们将实现一个非常简单的 JSON API。如果你有过其他后端 API 开发（特别是 Java）的经验，那么你一定会觉得用 Express 实现一个 JSON API 端口简单得不可思议。在之前提到的 Response 对象中，Express 为我们封装了一个 `json` 方法，直接就可以将一个 JavaScript 对象作为 JSON 数据返回，例如：

```
res.json({ name: '百万年薪', price: 996 });
```

会返回 JSON 数据 `{ "name": "百万年薪", "price": 996 }`，状态码默认为 200。我们还可以指定状态码，例如：

```
res.status(502).json({ error: '公司关门了' });
```

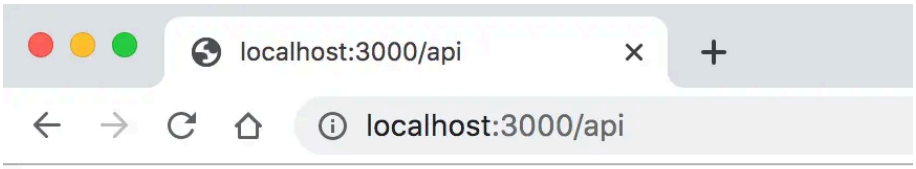
会返回 JSON 数据 `{ "error": "公司关门了"}`，状态码为 502。

到了动手环节，让我们在 server.js 中添加一个简单的 JSON API 端口 `/api`，返回关于图雀社区的一些数据：

```
// ...

app.get('/api', (req, res) => {
  res.json({ name: '图雀社区', website: 'https://tuture.co' });
```

```
});

// ...
```

我们可以用浏览器访问 localhost:3000/api 端口，看到返回了想要的数据：




```
{"name":"图雀社区","website":"https://tuture.co"}
```

或者你可以用 Postman 或 Curl 访问，也能看到想要的数据哦。

使用子路由拆分逻辑

当我们的网站规模越来越大时，把所有代码都放在 server.js 中可不是一个好主意。“拆分逻辑”（或者说“模块化”）是最常见的做法，而在 Express 中，我们可以通过子路由 `Router` 来实现。

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

`express.Router` 可以理解为一个迷你版的 `app` 对象，但是它功能完备，同样支持注册中间件和路由：

```
// 注册一个中间件
router.use(someMiddleware);

// 添加路由
router.get('/hello', helloHandler);
router.post('/world', worldHandler);
```

最后，由于 Express 中“万物皆中间件”的思想，一个 `Router` 也作为中间件加入到 `app` 中：

```
app.use('/say', router);
```

这样 `router` 下的全部路由都会加到 `/say` 之下，即相当于：

```
app.get('/say/hello', helloHandler);
app.post('/say/world', worldHandler);
```

正式实现

到了动手环节，首先创建 routes 目录，用于存放所有的子路由。创建 routes/index.js 文件，代码如下：

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

router.get('/', (req, res) => {
  res.render('index');
});

router.get('/contact', (req, res) => {
  res.render('contact');
});

module.exports = router;
```

创建 routes/api.js，代码如下：

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

router.get('/', (req, res) => {
  res.json({ name: '图雀社区', website: 'https://tuture.co' });
});

router.post('/new', (req, res) => {
  res.status(201).json({ msg: '新的篇章，即将开始' });
});

module.exports = router;
```

最后我们把 server.js 中老的路由定义全部删掉，替换成刚刚实现的两个 `Router`，代码如下：

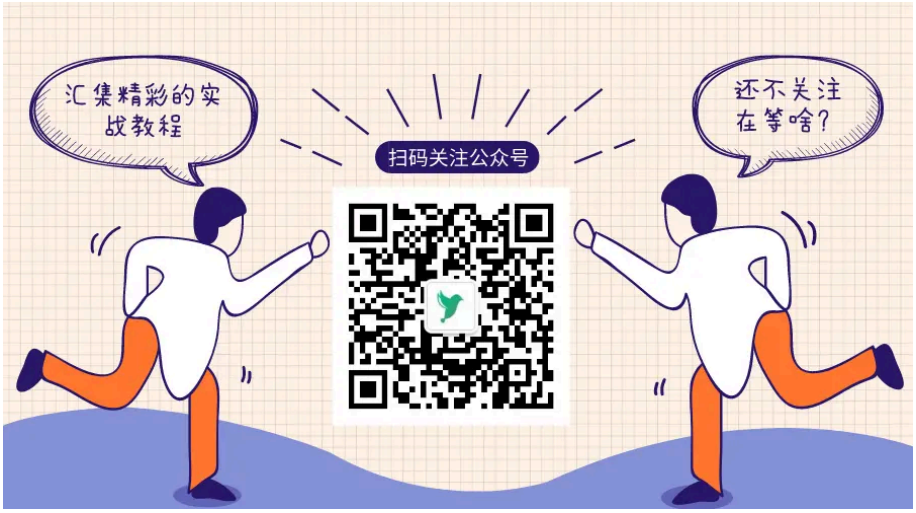
```
const express = require('express');
const path = require('path');

const indexRouter = require('./routes/index');
const apiRouter = require('./routes/api');

const hostname = 'localhost';
const port = 3000;

const app = express();

// ...
app.use(express.static('public'));

app.use('/', indexRouter);
app.use('/api', apiRouter);

app.use('*', (req, res) => {
  res.status(404).render('404', { url: req.originalUrl });
});

// ...
```

是不是瞬间清爽了很多呢！如果你服务器还开着，可以测试一下之前的路由是否还能成功运行哦。这里我贴一下用 Curl 测试 `/api` 路由的结果：

```
$ curl localhost:3000/api
{"name":"图雀社区","website":"https://tuture.co"}
$ curl -X POST localhost:3000/api/new
{"msg":"新的篇章，即将开始"}
```

至此，这篇教程也就结束了。所完成的网站的确很简单，但是希望你能从中学到 Express 的两大精髓：路由和中间件。掌握了这两大概念之后，后续进阶教程的学习也会轻松很多哦！

> 想要学习更多精彩的实战技术教程？来图雀社区逛逛吧。



node.js express 后端

赞 21 收藏 16 分享

阅读 2.9k • 发布于 2020-03-16

一只图雀
863 声望 • 1.2k 粉丝
我们图雀社区是一个供大家分享用 Tuture 写作工具撰写教程的一个平台。在这里，读者们可以尽情享受高质量的实战教...
关注作者

« 上一篇
Redux 包教包会（二）：引入...
下一篇 »
从零到部署：用 Vue 和 Express 实...

引用和评论

被 1 篇内容引用

从零到部署：用 Vue 和 Express 实现迷你全栈电商应用（二） 2

推荐阅读

Taro 小程序开发大型实战（九）：使用 Authing 打造具有微信登录的企业级用户系统
一只图雀 • 赞 1 • 阅读 4.4k

2024最新最全Java和Go面经，面试了30多场，终于上岸了！
王中阳Go • 赞 12 • 阅读 1.5k • 评论 3

爽了！免费的SSL，还能自动续期！
小傅哥 • 赞 5 • 阅读 343

金三银四，你的面试利器：一站式资源导航
九旬 • 赞 5 • 阅读 2.8k

Puppeteer实践：复杂的问题简单化
南城FE • 赞 4 • 阅读 579

Object.assign vs Object Spread
specialCoder • 赞 3 • 阅读 2.6k

【工具】用nvm管理nodejs版本切换，真香！
JavaDog程序狗 • 赞 4 • 阅读 7.8k • 评论 1

5 条评论

得票 | 最新

撰写评论 ...

提交评论

评论支持部分 Markdown 语法：`**粗体** _斜体_ [链接](http://example.com) `代码` - 列表 > 引用`。你还可以使用 @ 来通知其他用户。

指尖泛出的繁华：这篇教程不错，学习了?
1 • 回复 • 2020-03-16

maYunLaoXi：有基础的人是一杯茶，没有基础的人是一个下午
• 回复 • 2020-03-16

陈小亮Chan：现在还有人用express的?
• 回复 • 2020-03-16

oldfu：@陈小亮Chan 为什么不用呢?

• 回复 • 2020-03-17

heath_learning：虽然我学习了node、express、会前端、也会mysql的增删改查，但我还是不知道如何用node搭建一个网站，没有搭建网站的经验，求指教

• 回复 • 2020-03-17