Express 文档翻译稿件

author 李正尧 2022 03 15

express()

创建 express 应用程序。该函数由 module.express() 提供。

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

方法

express.json([options])

> 此中间件在Express v4.16.0 可用。

这是 express 中的中间件，可以对发送来的请求实用 JSON 解析，并且它是基于 body-parser的。 返回只解析JSON并只查看 Content-Type 请求头部与类型选项所匹配的请求的中间件。该解析器接受请求 body 部分的任何 Unicode 编码，并支持 gzip 的自动解压缩和 deflate 编码。 在中间件（即req.body）之后的请求对象上填充一个包含解析数据的新 body 对象，如果没有要解析的 body、内容类型不匹配或发生错误，则填充一个空对象（{}）。

> 按要求。body 的形态基于用户控制的输入，此对象中的所有属性和值都不受信任，应该在信任之前进行验证。例如，req.GoodHealth.ToString() 可能有若干种失败，就像 foo 可能不存在或可能不是字符串，而 toString 可能不是函数，而是字符串或其他用户输入。

下表介绍了可选选项对象的属性。

属性	描述	类型	默认值
inflate	启用或禁用压缩（解压）body 部分；禁用时不可解压	Boolean	true
limit	控制请求的最大 body 大小，如果是数值将会被解析为 byte 单位，如果是字符串将会交给字符库进行解析	Mixed	"100kb"
reviver	恢复器选项直接传递给 JSON.parse 作为第二个参数。	Function	null
strict	仅启用或禁用接受数组和对象；禁用时将接受任何 JSON.parse 的接受值。	Boolean	true
type	用于确定中间件将解析的媒体类型。此选项可以是字符串、字符串数组或函数。如果不是函数，type选项将直接传递给 type-is 库，它可以是扩展名（如 json）、mime类型（如 application/json）或带有通配符的 mime 类型（如 / 或 */json）。如果是函数，则类型选项被称为 fn（req），如果返回真值，则请求被解析。	Mixed	"application/json"
verify	如果提供此选项，被称为 verify(req, res, buf, encoding)，其中 buf 是原始请求 body 的 buffer，encoding 是请求的编码。可以通过抛出错误中止解析。	Function	undefined

express.Router([options])

创建一个 router 对象。

```
var router = express.Router([options])
```

可选参数 options 指明了路由行为。

属性	描述	默认值	可用版本
caseSensitive	区别大小写	Disabled by default, treating “/Foo” and “/foo” as the same.	
mergeParams	保留请求。来自父路由器的参数值。如果父级和子级的参数名称冲突，则子级的值优先。	false	4.5.0+
strict	启用严格路由	Disabled by default, “/foo” and “/foo/” are treated the same by the router.	

你可以像应用程序一样，向路由器添加中间件和HTTP方法路由（如get、put、post等）。

express.static(root, [options])

这是Express中的内置中间件功能。它为静态文件提供服务，并基于 serve static。

> 注意：为了获得最佳结果，请使用反向代理缓存来提高服务器处理静态资源的性能。

root 参数指定为静态资产提供服务的根目录。该函数通过接受到的 req.url 和提供的根目录来确定要发送的文件。当找不到文件时，它不会发送 404 响应，而是调用 next（）以转到下一个中间件，以允许压栈和回退。

下表介绍可选对象的属性。

属性	描述	类型	默认值
dotfiles	明确如何处理点文件（以点“.”开头的文件或目录）	String	“ignore”
etag	启用或禁用etag生成 注：express.static 总是发送弱 ETags	Boolean	true
extensions	设置文件扩展名回退：如果找不到文件，请搜索具有指定扩展名的文件，并为找到的第一个文件提供服务。示例：['html'，'htm']。	Mixed	false
fallthrough	让客户端错误作为未处理的请求通过，否则转发客户端错误。	Boolean	true
immutable	启用或禁止报文头部 Cache-Control 中的不可变指令。若启动，应同时指定 maxAge 选项以启用缓存。不可变指令将防止受支持的客户端在maxAge选项的有效期内发出条件请求，以检查文件是否已更改。	Boolean	false
index	发送指定的目录索引文件。设置为 false 可禁用目录索引。	Mixed	“index.html”
lastModified	将 Last-Modified 的头设置为操作系统上文件的上次修改日期	Boolean	true
maxAge	以毫秒为单位设置 Cache-Control 标头的 max-age 属性，或以毫秒为单位设置字符串。	Number	0
redirect	当路径名为目录时，重定向到尾部“/”	Boolean	true
setHeaders	用于设置要与文件一起使用的 HTTP 头的函数	Function	

点文件

此选项的可能值为：

- "allow" 对点文件没有特殊处理
- "deny" 拒绝对点文件的请求并返回 403 响应，然后调用 next()
- "ignore" 如果dotfile不存在，返回 404 响应，然后调用next()

注意：使用默认值时，它不会忽略以点开头的目录中的文件。

fallthrough

如果此选项为 true，则客户端错误（例如错误的请求或对不存在的文件的请求）将导致此中间件只需调用next()即可调用堆栈中的下一个中间件。如果为 false，这些错误（甚至404）将调用next(err)。 将此选项设置为 true，这样就可以将多个物理目录映射到同一个 web 地址，或者用路由来填充不存在的文件。 如果您已将该中间件安装在严格设计为单个文件系统目录的路径上，则使用 false，这允许短路 404 以减少开销。该中间件还将响应所有方法。

setHeaders

对于此选项，请指定一个函数来设置自定义响应头。标题的更改必须同步进行。 该函数的签名为：

```
fn(res, path, stat)
```

参数：

- res 报文对象
- path 被发送的文件路径
- stat 文件被发送的状态

express.static **使用例**

下面是一个带有复杂选项对象的静态中间件功能的，express.static 使用例：

```
var options = {
  dotfiles: 'ignore',
  etag: false,
  extensions: ['htm', 'html'],
  index: false,
  maxAge: '1d',
  redirect: false,
  setHeaders: function (res, path, stat) {
    res.set('x-timestamp', Date.now())
  }
}

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

app

app.route(path)

返回单个路由的实例，然后可以使用可选的中间件来处理 HTTP 谓词。使用app.route()以避免重复的路由名称（从而避免拼写错误）。

```
var app = express()

app.route('/events')
  .all(function (req, res, next) {
    // runs for all HTTP verbs first
    // think of it as route specific middleware!
  })
  .get(function (req, res, next) {
    res.json({})
  })
  .post(function (req, res, next) {
    // maybe add a new event...
  })
```

app.get(name)

返回一个在 app setting table 中 name 字符串的值。例如：

```
app.get('title')
// => undefined

app.set('title', 'My Site')
app.get('title')
// => "My Site"
```

app.get(path, callback [, callback ...])

使用指定的回调函数将HTTP GET请求路由到指定路径。

参数表

参数	描述	默认值
path	调用中间件功能的路径；可以是以下任何一种：1·表示路径的字符串 2·路径模式 3·匹配路径的正则表达式模式 4·上述任意组合的数组	'/' (root path)
callback	回调函数；可以是：1·中间件函数 2·中间件函数构成的序列（用逗号分隔） 3·中间件函数构成的数组 4·以上所有要素的结合 你可以提供多个回调函数，它们的行为与中间件类似，只是这些回调函数可以调用next（“route”）来绕过其余的路由回调。你可以使用此机制对路由施加先决条件，然后在没有理由继续当前路由的情况下，将控制权传递给后续路由。由于路由器和应用程序实现了中间件接口，您可以像使用任何其他中间件功能一样使用它们。	None

用例：

```
app.get('/', function (req, res) {
  res.send('GET request to homepage')
})
```

app.listen(path, [callback])

启动并监听一个来自给出 path 的 UNIX socket。此方法与 node 的 http.Server.listen() 相同。

```
var express = require('express')
var app = express()
app.listen('/tmp/sock')
```

app.listen([port[, host[, backlog]]][, callback])

绑定并侦听指定主机和端口上的连接。此方法与 node 的 http.Server.listen() 相同。 如果端口被省略或为0，操作系统将分配一个任意未使用的端口，这对于自动化任务（测试等）等情况很有用。

```
var express = require('express')
var app = express()
app.listen(3000)
```

被 express() 返回的实际上是一个用来作为回调函数传递给 node 的 HTTP 服务器的 JavaScript 函数，以处理请求。这使得为应用程序的HTTP和HTTPS版本提供相同的代码基础变得很容易，因为应用程序不会继承这些代码（它只是一个回调）：

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

http.createServer(app).listen(80)
https.createServer(options, app).listen(443)
```

app.listen() 方法返回一个 http.Server 对象，并且（对 HTTP 来说）是更方便的方法，用来：

```
app.listen = function () {
  var server = http.createServer(this)
  return server.listen.apply(server, arguments)
}
```

> 注：所有形式的 node 中 http.Server.listen() 方法实际上都已被支持

respons

res 对象表示 express 应用在收到 HTTP 请求时发送的 HTTP 响应。 在本文档中，按照惯例，对象总是被称为res（HTTP 请求是 req），但它的实际名称由您使用的回调函数的参数决定。 例如：

```
app.get('/user/:id', function (req, res) {
  res.send('user ' + req.params.id)
})
```

但你也可以：

```
app.get('/user/:id', function (request, response) {
  response.send('user ' + request.params.id)
})
```

res 对象是 node 自身响应对象的增强版本，支持所有内置字段和方法。

res.end([data] [, encoding])

结束响应过程。实际上该方法来自 node 内核中的 http.ServerResponse 的 response.end() 方法，用于在没有任何数据的情况下快速结束响应。如果需要响应数据，则用 res.send() 和 res.json() 替代。

```
res.end()
res.status(404).end()
```

res.get(field)

返回指定域名的 HTTP 响应（域名不区分大小写）。

```
res.get('Content-Type')
// => "text/plain"
```

res.json([body])

发送一个具有正确格式的 JSON 响应。参数必须是实用 JSON.stringify() 后的可被转换为 JSON 字符串的。

```
res.json(null)
res.json({ user: 'tobi' })
res.status(500).json({ error: 'message' })
```

res.redirect([status,] path)

重定向到从指定路径派生的、具有指定状态的URL，该URL是一个正整数，对应于HTTP状态代码。如果未指定，状态默为 "302 found"。

```
res.redirect('/foo/bar')
res.redirect('http://example.com')
res.redirect(301, 'http://example.com')
res.redirect('../login')
```

重定向可以是完全限定的URL，用于重定向到其他站点：

```
res.redirect('http://google.com')
```

重定向可以相对于主机名的根目录。例如，如果应用运行于 http://example.com/admin/post/new, 那么将会重定向到 URL http://example.com/admin:

```
res.redirect('/admin')
```

重定向可以相对于当前URL。例如，从 http://example.com/blog/admin/ （注意后面的斜杠），下面的内容将重定向到 URL http://example.com/blog/admin/post/new。

```
res.redirect('post/new')
```

从 post/new 重定向到 http://example.com/blog/admin（无尾随斜杠），将转到http://example.com/blog/post/new。如果您发现上述行为令人困惑，请将路径段视为目录（带有尾随斜杠）和文件。路径相对重定向也是可能的。如果你在 http://example.com/admin/post/new，以下内容将重定向到 http://example.com/admin/post:

```
res.redirect('..')
```

res.send([body])

发送HTTP响应。body 参数可以是 Buffer 对象、字符串、对象、布尔值或数组。例如：

```
res.send(Buffer.from('whoop'))
res.send({ some: 'json' })
res.send('<p>some html</p>')
res.status(404).send('Sorry, we cannot find that!')
res.status(500).send({ error: 'something blew up' })
```

此方法对简单的非流式响应执行许多有用的任务：例如，它自动分配内容长度HTTP响应头字段（除非之前定义），并提供自动头和HTTP缓存新鲜度支持。当参数是 Buffer 对象时，该方法将内容类型响应头字段设置为“应用程序/八位字节流”，除非之前定义如下：

```
res.set('Content-Type', 'text/html')
res.send(Buffer.from('<p>some html</p>'))
```

当参数为字符串时，该方法将内容类型设置为“text/html”：

```
res.send('<p>some html</p>')
```

当参数是数组或对象时，Express会使用 JSON 表示法进行响应：

```
res.send({ user: 'tobi' })
res.send([1, 2, 3])
```

res.set(field [, value])

将响应的 HTTP 头字段设置为值。要同时设置多个字段，需传递一个对象作为参数。

```
res.set('Content-Type', 'text/plain')

res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
  ETag: '12345'
})
```

别名为 res.header(field [, value])。

res.status(code)

为 HTTP 响应设置状态码。

```
res.status(403).end()
res.status(400).send('Bad Request')
res.status(404).sendFile('/absolute/path/to/404.png')
```

Router

router 对象是中间件和路由的独立实例。您可以将其视为一个“迷你应用”，只能够执行中间件和路由功能。每个Express 应用都有一个内置的应用程序 router 。 router 行为类似中间件本身，所以你可以将它作为 app.use() 的参数使用，或者别的 router 的 use() 模式。 最高等级的 express() 对象有一个 Router() 方法来创建一个新 router 对象。 一旦创建了路由器对象，就可以像应用程序一样向其添加中间件和HTTP方法路由（如 get、put、post 等）。例如：

```
// invoked for any requests passed to this router
router.use(function (req, res, next) {
  // .. some logic here .. like any other middleware
  next()
})

// will handle any request that ends in /events
// depends on where the router is "use()'d"
router.get('/events', function (req, res, next) {
  // ..
})
```

然后，你可以使用 router 来创建特定的根 URL，通过这种方式将你的路由分割成文件甚至迷你应用。

```
// only requests to /calendar/* will be sent to our "router"
app.use('/calendar', router)
```

router.use([path], [function, ...] function)

使用指定的一个或多个中间件函数，以及默认为“/”的可选装载路径。 该方法类似于 app.use()。下面描述了一个例子。中间件就像管道：请求从定义的第一个中间件功能开始，并在中间件堆栈中“向下”处理它们匹配的每个路径。

```
var express = require('express')
var app = express()
var router = express.Router()


// simple logger for this router's requests
// all requests to this router will first hit this middleware
router.use(function (req, res, next) {
  console.log('%s %s %s', req.method, req.url, req.path)
  next()
})

// this will only be invoked if the path starts with /bar from the mount point
router.use('/bar', function (req, res, next) {
  // ... maybe some additional /bar logging ...
  next()
})

// always invoked
router.use(function (req, res, next) {
  res.send('Hello World')
})

app.use('/foo', router)
```

```
app.listen(3000)
```

“mount”路径被剥离，中间件功能不可见。此功能的主要作用是，无论“prefix”路径名如何，挂载的中间件功能都可以在不更改代码的情况下运行。 使用 router.use() 定义中间件的顺序非常重要。它们是按顺序调用的，因此顺序定义了中间件的优先级。例如，通常情况下，记录器是您使用的第一个中间件，因此每个请求都会被记录下来。

```
var logger = require('morgan')
var path = require('path')

router.use(logger())
router.use(express.static(path.join(__dirname, 'public')))
router.use(function (req, res) {
  res.send('Hello')
})
```

假设忽略记录请求静态文件，但继续记录 logger() 后的中间件和路由。你将在加入 logger 中间件前轻松将 express.static() 移动到最顶部：

```
router.use(express.static(path.join(__dirname, 'public')))
router.use(logger())
router.use(function (req, res) {
  res.send('Hello')
})
```

另一个例子是为多个目录中的文件提供服务，给“/public”优先于其他目录的优先级：

```
router.use(express.static(path.join(__dirname, 'public')))
router.use(express.static(path.join(__dirname, 'files')))
router.use(express.static(path.join(__dirname, 'uploads')))
```

router.use() 方法还支持命名参数，因此其他路由器的装载点可以受益于使用命名参数预加载。

注意：尽管这些中间件功能是通过特定的 router 添加的，但它们运行的时间是由它们所连接的路径（而不是 router）定义的。因此，通过一个 router 添加的中间件如果其路由匹配，可能会为其他 router 运行。例如，此代码显示安装在同一路径上的两个不同 router：

```
var authRouter = express.Router()
var openRouter = express.Router()

authRouter.use(require('./authenticate').basic(usersdb))

authRouter.get('/:user_id/edit', function (req, res, next) {
  // ... Edit user UI ...
})
openRouter.get('/', function (req, res, next) {
  // ... List users ...
})
openRouter.get('/:user_id', function (req, res, next) {
  // ... View user ...
})

app.use('/users', authRouter)
app.use('/users', openRouter)
```

尽管身份验证中间件是通过 authRouter 添加的，但它也将在 openRouter 定义的路由上运行，因为这两个 router 都加载在 /users 上。要避免这种行为，请为每个 router 使用不同的路径。