Express指南

极客学院出版

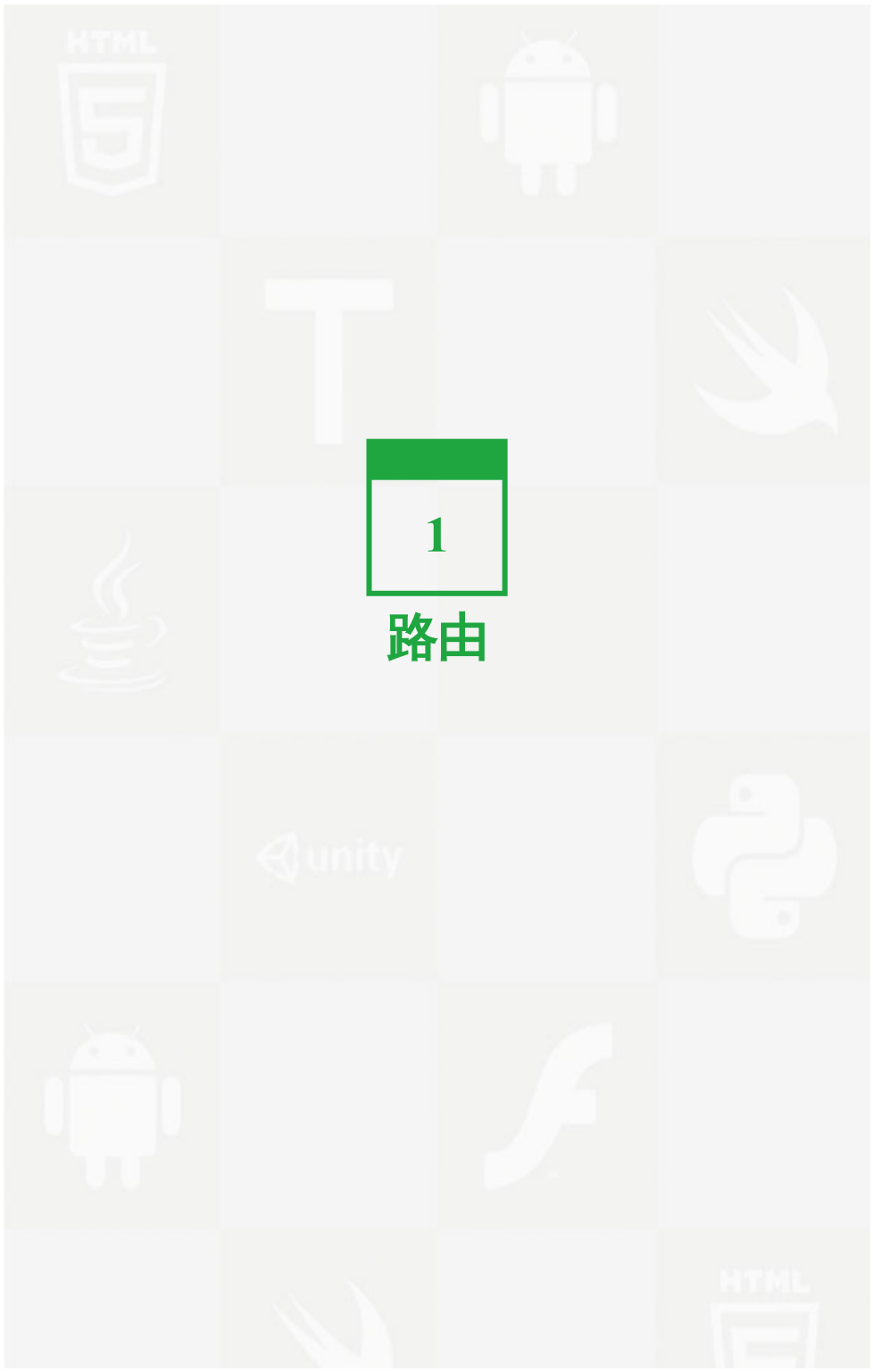
前言

Express 是一个简洁、灵活的 node.js Web 应用开发框架, 它提供一系列强大的特性，帮助你创建各种 Web 和移动设备应用。

适用人群

该教程适用于初学者，本文是官网翻译版本，可帮助掌握 Express 的基本功能及使用方法。

更新日期	更新内容
2015-06-15	Express 指南



1

路由

路由是指如何定义应用的端点（URIs）以及如何响应客户端的请求。

路由是由一个 URI、HTTP 请求（GET、POST等）和若干个句柄组成，它的结构如下： `app.METHOD(path, [callback...], callback)` ， `app` 是 `express` 对象的一个实例， `METHOD` 是一个 HTTP 请求方法， `path` 是服务器上的路径， `callback` 是当路由匹配时要执行的函数。

下面是一个基本的路由示例：

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

// 当主页收到 GET 请求时，返回 "hello world"。
app.get('/', function(req, res) {
  res.send('hello world');
});
```

路由方法

路由方法源于 HTTP 请求方法，和 `express` 实例相关联。

下面这个例子展示了为应用跟路径定义的 GET 和 POST 请求：

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

// POST 路由
app.post('/', function (req, res) {
  res.send('POST request to the homepage');
});
```

Express 定义了如下和 HTTP 请求对应的路由方法：`get`、`post`、`put`、`head`、`delete`、`options`、`trace`、`copy`、`lock`、`mkcol`、`move`、`purge`、`propfind`、`proppatch`、`unlock`、`report`、`mkactivity`、`checkout`、`merge`、`m-search`、`notify`、`subscribe`、`unsubscribe`、`patch`、`search` 和 `connect`。

有些路由方法名不是合规的 JavaScript 变量名，此时使用括号记法，比如 `app['m-search']('/', function ...`

`app.all()` 是一个特殊的路由方法，没有任何 HTTP 方法与其对应，它的作用是对于一个路径上的所有请求加载中间件。

在下面的例子中，来自 “/secret” 的请求，不管使用 GET、POST、PUT、DELETE 或其他任何 http 模块支持的 HTTP 请求，句柄都会得到执行。

```
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...');
  next(); // 交给下一个句柄处理
});
```

路由路径

路由路径和请求方法一起定义了请求的端点，它可以是字符串、字符串模式或者正则表达式。

Express 使用 path-to-regexp 匹配路由路径，请参考文档查阅所有定义路由路径的方法。 Express Route Tester 是测试基本 Express 路径的好工具，但不支持模式匹配。

查询字符串不是路由路径的一部分。

使用字符串的路由路径示例：

```
// 匹配根路径的请求
app.get('/', function (req, res) {
  res.send('root');
});

// 匹配 /about 路径的请求
app.get('/about', function (req, res) {
  res.send('about');
});

// 匹配 /random.text 路径的请求
app.get('/random.text', function (req, res) {
  res.send('random.text');
});
```

使用字符串模式的路由路径示例：

```
// 匹配 acd 和 abcd
app.get('/ab?cd', function(req, res) {
  res.send('ab?cd');
});

// 匹配 abcd、abbcd、abbbcd等
app.get('/ab+cd', function(req, res) {
  res.send('ab+cd');
});

// 匹配 abcd、abxcd、abRABDOMcd、ab123cd等
app.get('/ab*cd', function(req, res) {
  res.send('ab*cd');
});

// 匹配 /abe 和 /abcde
app.get('/ab(cd)?e', function(req, res) {
 res.send('ab(cd)?e');
});
```

字符 ?、+、* 和 () 是正则表达式的子集，- 和 . 在基于字符串的路径中按照字面值解释。

使用正则表达式的路由路径示例：

```
// 匹配任何路径中含有 a 的路径：
app.get(/a/, function(req, res) {
  res.send('/a/');
});

// 匹配 butterfly、dragonfly，不匹配
butterflyman、dragonfly man等
app.get(/.*fly$/, function(req, res) {
  res.send('/.*fly$/');
});
```

路由句柄

可以为请求处理提供多个回调函数，其行为类似 中间件。唯一的区别是这些回调函数有可能调用 `next('route')` 方法而略过其他路由回调函数。可以利用该机制为路由定义前提条件，如果在现有路径上继续执行没有意义，则可将控制权交给剩下的路径。

路由句柄有多种形式，可以是一个函数、一个函数数组，或者是两者混合，如下所示.

使用一个回调函数处理路由：

```
app.get('/example/a', function (req, res) {
  res.send('Hello from A!');
});
```

使用多个回调函数处理路由（记得指定 `next` 对象）：

```
app.get('/example/b', function (req, res, next)
{
  console.log('response will be sent by the
next function ...');
  next();
}, function (req, res) {
```

```
  res.send('Hello from B!');
});
```

使用回调函数数组处理路由：

```
var cb0 = function (req, res, next) {
  console.log('CB0');
  next();
}

var cb1 = function (req, res, next) {
  console.log('CB1');
  next();
}

var cb2 = function (req, res) {
  res.send('Hello from C!');
}

app.get('/example/c', [cb0, cb1, cb2]);
```

混合使用函数和函数数组处理路由：

```
var cb0 = function (req, res, next) {
  console.log('CB0');
  next();
}

var cb1 = function (req, res, next) {
  console.log('CB1');
  next();
}

app.get('/example/d', [cb0, cb1], function
(req, res, next) {
  console.log('response will be sent by the
next function ...');
  next();
}, function (req, res) {
  res.send('Hello from D!');
});
```

响应方法

下表中响应对象（`res`）的方法向客户端返回响应，终结请求响应的循环。如果在路由句柄中一个方法也不调用，来自客户端的请求会一直挂起。

方法	描述
res.download()	提示下载文件。
res.end()	终结响应。
res.json()	发送一个 JSON 响应。
res.jsonp()	发送一个支持 JSONP 的 JSON 响应。
res.redirect()	重定向请求。
res.render()	渲染视图模板。
res.send()	发送各种类型的响应。
res.sendFile	以八位字节流的形式发送文件。
res.sendStatus()	设置响应状态代码，并将其以字符串形式作为响应体的一部分发送。

app.route()

可使用 `app.route()` 创建路由路径的链式路由句柄。由于路径在一个地方指定，这样做有助于创建模块化的

路由，而且减少了代码冗余和拼写错误。请参考 Router() 文档 了解更多有关路由的信息。

下面这个示例程序使用 `app.route()` 定义了链式路由句柄。

```
app.route('/book')
  .get(function(req, res) {
    res.send('Get a random book');
  })
  .post(function(req, res) {
    res.send('Add a book');
  })
  .put(function(req, res) {
    res.send('Update the book');
  });
```

express.Router

可使用 `express.Router` 类创建模块化、可挂载的路由句柄。`Router` 实例是一个完整的中间件和路由系统，因此常称其为一个 “mini-app”。

下面的实例程序创建了一个路由模块，并加载了一个中间件，定义了一些路由，并且将它们挂载至应用的路径上。

在 app 目录下创建名为 `birds.js` 的文件，内容如下：

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

// 该路由使用的中间件
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});
// 定义主页路由
```

```
router.get('/', function(req, res) {
  res.send('Birds home page');
});
// 定义 about 路由
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

然后在应用中加载路由模块：

```
var birds = require('./birds');
...
app.use('/birds', birds);
```

应用即可处理发自 `/birds` 和 `/birds/about` 的请求，并且调用为该路由指定的 `timeLog` 中间件。

2

使用中间件

从本质上来说，一个 Express 应用就是在调用各种中间件。

中间件是一个可访问请求对象（`req`）和响应对象（`res`）的函数，在 Express 应用的请求-响应循环里，下一个内联的中间件通常用变量 `next` 表示。中间件的功能包括：

- 执行任何代码。
- 修改请求和响应对象。
- 终结请求-响应循环。
- 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。

使用可选则挂载路径，可在应用级别或路由级别装载中间件。可装载一系列中间件函数，在挂载点创建一个中间件系统栈。

Express 应用可使用如下几种中间件：

- 应用级中间件
- 路由级中间件
- 错误处理中间件
- 内置中间件
- 第三方中间件

应用级中间件

应用级中间件绑定到 `express` 实例，使用 `app.use()` 和 `app.VERB()`。

```
var app = express();

// 没有挂载路径的中间件，应用的每个请求都会执行该中间件
app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});

// 挂载至 /user/:id 的中间件，任何指向 /user/:id 的请求都会执行它
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});

// 路由和句柄函数(中间件系统)，处理指向 /user/:id 的 GET 请求
app.get('/user/:id', function (req, res, next) {
  res.send('USER');
});
```

下面这个例子展示了在一个挂载点装载一组中间件。

```
// 一个中间件栈，对任何指向 /user/:id 的 HTTP 请求打印出相关信息
app.use('/user/:id', function(req, res, next) {
  console.log('Request URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
```

作为中间件系统的路由句柄，使得为路径定义多个路由成为可能。在下面的例子中，为指向 `/user/:id` 的 GET 请求定义了两个路由。第二个路由虽然不会带来任何问题，但却永远不会被调用，因为第一个路由已经终止了请求-响应循环。

```
// 一个中间件栈，处理指向 /user/:id 的 GET 请求
app.get('/user/:id', function (req, res, next)
{
  console.log('ID:', req.params.id);
  next();
}, function (req, res, next) {
  res.send('User Info');
});

// 处理 /user/:id， 打印出用户 id
app.get('/user/:id', function (req, res, next)
{
  res.end(req.params.id);
});
```

如果需要在中间件栈中跳过剩余中间件，调用 `next('route')` 方法将控制权交给下一个路由。需要注意的是 `next('route')` 只对使用 `app.VERB()` 或 `router.VERB()` 加载的中间件有效。

```
// 一个中间件栈，处理指向 /user/:id 的 GET 请求
app.get('/user/:id', function (req, res, next)
{
  // 如果 user id 为 0, 跳到下一个路由
  if (req.params.id == 0) next('route');
  // 负责将控制权交给栈中下一个中间件
  else next(); //
}, function (req, res, next) {
  // 渲染常规页面
  res.render('regular');
});
```

```
// 处理 /user/:id， 渲染一个特殊页面
app.get('/user/:id', function (req, res, next)
{
  res.render('special');
});
```

路由级中间件

路由级中间件和应用级中间件一样，只是它绑定的对象为 `express.Router()`。

```
var router = express.Router();
```

路由级使用 `router.use()` 或 `router.VERB()` 加载。

上述在应用级创建的中间件系统，可通过如下代码改写为路由级：

```
var app = express();
var router = express.Router();

// 没有挂载路径的中间件，通过该路由的每个请求都会执行该中间件
router.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});

// 一个中间件栈，显示任何指向 /user/:id 的 HTTP 请求的信息
router.use('/user/:id', function(req, res,
next) {
  console.log('Request URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
```

```
// 一个中间件栈，处理指向 /user/:id 的 GET 请求
router.get('/user/:id', function (req, res,
next) {
  // 如果 user id 为 0，跳到下一个路由
  if (req.params.id == 0) next('route');
  // 负责将控制权交给栈中下一个中间件
  else next(); //
}, function (req, res, next) {
  // 渲染常规页面
  res.render('regular');
});

// 处理 /user/:id， 渲染一个特殊页面
router.get('/user/:id', function (req, res,
next) {
  console.log(req.params.id);
  res.render('special');
});

// 将路由挂载至应用
app.use('/', router);
```

错误处理中间件

错误处理中间件有 4 个参数，定义错误处理中间件时必须使用这 4 个参数。即使不需要 `next` 对象，也必须在签名中声明它，否则中间件会被识别为一个常规中间件，不能处理错误。

错误处理中间件和其他中间件定义类似，只是要使用 4 个参数，而不是 3 个，其签名如下： `(err, req, res, next)`

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
```

```
  res.status(500).send('Something broke!');
});
```

请参考错误处理一章了解更多关于错误处理中间件的内容。

内置中间件

在 4.x 版本中，Express 已经不再依赖 Connect。除了 `express.static`，Express 以前包括的中间件现在已经在一个单独的库里，请参考中间件列表。

express.static(root, [options])

`express.static` 是 Express 唯一内置的中间件，它基于 serve-static，负责在 Express 应用中提供静态资源。

参数 `root` 指提供静态资源的根目录。

可选的 `options` 参数拥有如下属性。

属性	描述	类型	缺省值
`dotfiles`	是否提供 dotfiles，可选值为 “allow”、“deny” 和 “ignore”	String	“ignore”
`etag`	是否启用 etag 生成	Boolean	`true`
`extensions`	设置文件扩展回退	Boolean	`false`

属性	描述	类型	缺省值
`index`	发送目录索引文件，设置为 `false` 禁用目录索引。	Mixed	“index.html”
`lastModified`	设置 `Last-Modified` 头为文件在操作系统上的最后修改日期。可能值为 `true` 或 `false`。	Boolean	`true`
`maxAge`	以毫秒或者其字符串格式设置 Cache-Control 头的 max-age 属性。	Number	0
`redirect`	当路径为目录时，重定向至“/”。	Boolean	`true`
`setHeaders`	设置 HTTP 头以提供文件的函数。	Function	

下面的例子使用了 `express.static` 中间件，其中的 `options` 对象经过了精心的设计。

```
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));
```

每个应用可有多个静态目录。

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

请参考 serve-static 文档，了解更多关于 `serve-static` 和其选项的信息。

第三方中间件

Express 是一款提供路由和中间件的 Web 框架，但其本身的功能却异常精简。Express 应用的功能通过第三方中间件来添加。

安装所需功能的 node 模块，并在应用中加载，可以在应用级加载，也可以在路由级加载。

下面的例子安装并加载了一个解析 cookie 的中间件：`cookie-parser`

```
$ npm install cookie-parser
```

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

// 加载 cookie 解析中间件
app.use(cookieParser());
```

请参考 Third-party middleware 获取 Express 中经常用到的第三方中间件列表。

3

在 Express 中使用模板引擎

需要在应用中进行如下设置才能让 Express 渲染模板文件：

- `views`，放模板文件的目录，比如 `app.set('views', './views')`。
- `view engine`，模板引擎，比如 `app.set('view engine', 'jade')`。

然后安装相应的模板引擎 npm 软件包。

```
$ npm install jade --save
```

和 Express 兼容的模板引擎，比如 Jade，通过 `res.render()` 调用其导出方法 `__express(filePath, options, callback)` 渲染模板。

有一些模板引擎不遵循这种约定，Consolidate.js 能将 Node 中所有流行的模板引擎映射为这种约定，这样就可以和 Express 无缝衔接。

一旦 `view engine` 设置成功，就不需要显式指定引擎，或者在应用中加载模板引擎模块，Express 已经在内部加载，如下所示。

```
app.set('view engine', 'jade');
```

在 `views` 目录下生成名为 `index.jade` 的 jade 模板文件，内容如下：

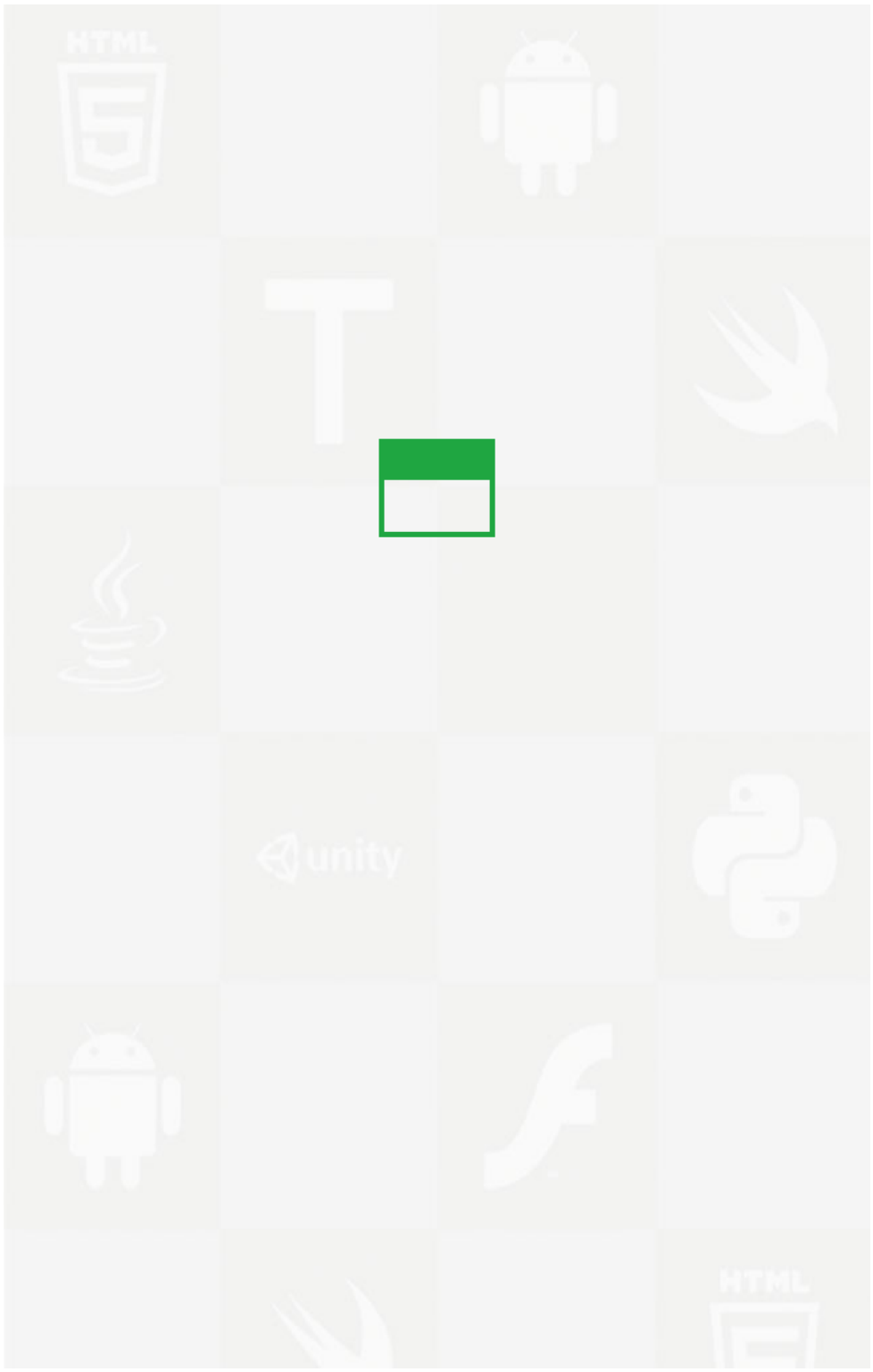
```
html
  head
    title!= title
  body
    h1!= message
```

然后创建一个路由渲染 `index.jade` 文件。如果没有设置 `view engine`，您需要指明视图文件的后缀，否则就会遗漏它。

```
app.get('/', function (req, res) {
  res.render('index', { title: 'Hey', message:
'Hello there!'});
});
```

此时向主页发送请求，“index.jade” 会被渲染为 HTML。

请阅读 为 Express 开发模板引擎 了解模板引擎在 Express 中是如何工作的。



错误处理

定义错误处理中间件和定义其他中间件一样，除了需要 4 个参数，而不是 3 个，其签名如下 `(err, req, res, next)`。

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

在其他 `app.use()` 和路由调用后，最后定义错误处理中间件，比如：

```
var bodyParser = require('body-parser');
var methodOverride = require('method-
override');

app.use(bodyParser());
app.use(methodOverride());
app.use(function(err, req, res, next) {
  // 业务逻辑
});
```

中间件返回的响应是随意的，可以响应一个 HTML 错误页面、一句简单的话、一个 JSON 字符串，或者其他任何您想要的东西。

为了便于组织（更高级的框架），您可能会像定义常规中间件一样，定义多个错误处理中间件。比如您想为使用 XHR 的请求定义一个，还想为没有使用的定义一个，那么：

```
var bodyParser = require('body-parser');
var methodOverride = require('method-
override');

app.use(bodyParser());
app.use(methodOverride());
```

```
app.use(logErrors);
app.use(clientErrorHandler);
app.use(errorHandler);
```

`logErrors` 将请求和错误信息写入标准错误输出、日志或类似服务：

```
function logErrors(err, req, res, next) {
  console.error(err.stack);
  next(err);
}
```

`clientErrorHandler` 的定义如下（注意这里将错误直接传给了 `next`）：

```
function clientErrorHandler(err, req, res,
next) {
  if (req.xhr) {
    res.status(500).send({ error: 'Something
blew up!' });
  } else {
    next(err);
  }
}
```

`errorHandler` 能捕获所有错误，其定义如下：

```
function errorHandler(err, req, res, next) {
  res.status(500);
  res.render('error', { error: err });
}
```

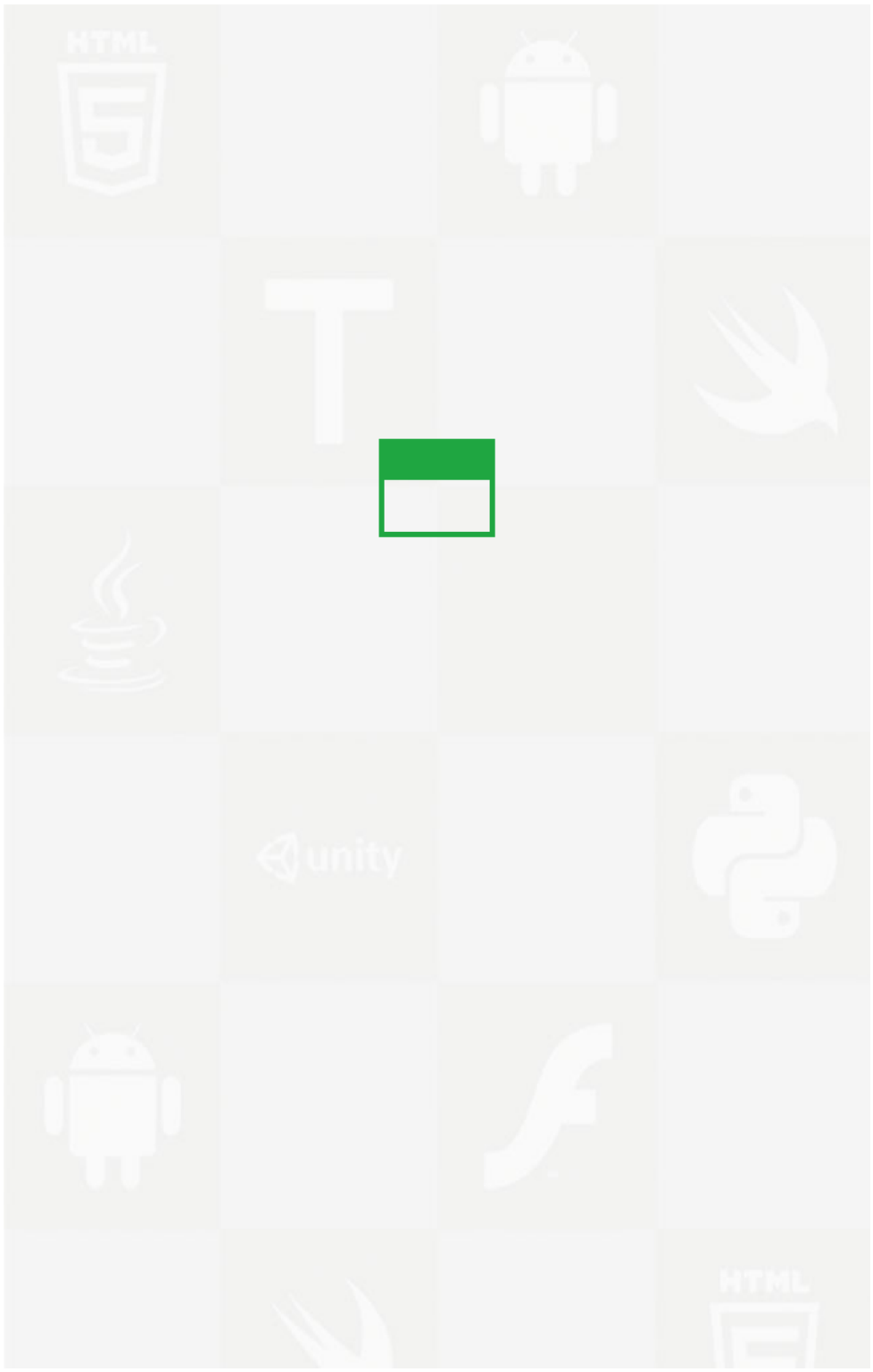
如果向 `next()` 传入参数（除了 'route' 字符串），Express 会认为当前请求有错误的输出，因此跳过后续其他非错误处理和路由/中间件函数。如果需做特殊处理，需要创建新的错误处理路由，如下节所示。

如果路由句柄有多个回调函数，可使用 'route' 参数跳到下一个路由句柄。比如：

```
app.get('/a_route_behind_paywall',
  function checkIfPaidSubscriber(req, res,
next) {
    if(!req.user.hasPaid) {
      // 继续处理该请求
      next('route');
    }
  }, function getPaidContent(req, res, next) {
    PaidContent.find(function(err, doc) {
      if(err) return next(err);
      res.json(doc);
    });
  });
```

在这个例子中，句柄 `getPaidContent` 会被跳过，但 `app` 中为 `/a_route_behind_paywall` 定义的其他句柄则会继续执行。

> `next()` 和 `next(err)` 类似于 `Promise.resolve()` 和 `Promise.reject()`。它们让您可以向 Express 发信号，告诉它当前句柄执行结束并且处于什么状态。`next(err)` 会跳过后续句柄，除了那些用来处理错误的句柄。



调试 Express

Express 内部使用 debug 模块记录路由匹配、使用到的中间件、应用模式以及请求-响应循环。

`debug` 有点像改装过的 `console.log`，不同的是，您不需要在生产代码中注释掉 `debug`。它会默认关闭，而且使用一个名为 `DEBUG` 的环境变量还可以打开。

在启动应用时，设置 `DEBUG` 环境变量为 `express:*`，可以查看 Express 中用到的所有内部日志。

```
$ DEBUG=express:* node index.js
```

在 Windows 系统里，使用相应的命令。

```
> set DEBUG=express:* & node index.js
```

在由 express generator 生成的默认应用中执行，会打印出如下信息：

```
$ DEBUG=express:* node ./bin/www
  express:router:route new / +0ms
  express:router:layer new / +1ms
  express:router:route get / +1ms
  express:router:layer new / +0ms
  express:router:route new / +1ms
  express:router:layer new / +0ms
  express:router:route get / +0ms
  express:router:layer new / +0ms
  express:application compile etag weak +1ms
  express:application compile query parser
extended +0ms
  express:application compile trust proxy false
+0ms
  express:application booting in development
mode +1ms
  express:router use / query +0ms
  express:router:layer new / +0ms
  express:router use / expressInit +0ms
```

```
  express:router:layer new / +0ms
  express:router use / favicon +1ms
  express:router:layer new / +0ms
  express:router use / logger +0ms
  express:router:layer new / +0ms
  express:router use / jsonParser +0ms
  express:router:layer new / +1ms
  express:router use / urlencodedParser +0ms
  express:router:layer new / +0ms
  express:router use / cookieParser +0ms
  express:router:layer new / +0ms
  express:router use / stylus +90ms
  express:router:layer new / +0ms
  express:router use / serveStatic +0ms
  express:router:layer new / +0ms
  express:router use / router +0ms
  express:router:layer new / +1ms
  express:router use /users router +0ms
  express:router:layer new /users +0ms
  express:router use / <anonymous> +0ms
  express:router:layer new / +0ms
  express:router use / <anonymous> +0ms
  express:router:layer new / +0ms
  express:router use / <anonymous> +0ms
  express:router:layer new / +0ms
```

当应用收到请求时，能看到 Express 代码中打印出的日志。

```
express:router dispatching GET / +4h
  express:router query  : / +2ms
  express:router expressInit  : / +0ms
  express:router favicon  : / +0ms
  express:router logger  : / +1ms
  express:router jsonParser  : / +0ms
  express:router urlencodedParser  : / +1ms
  express:router cookieParser  : / +0ms
  express:router stylus  : / +0ms
  express:router serveStatic  : / +2ms
  express:router router  : / +2ms
```

```
  express:router dispatching GET / +1ms
  express:view lookup "index.jade" +338ms
  express:view stat
"/projects/example/views/index.jade" +0ms
  express:view render
"/projects/example/views/index.jade" +1ms
```

设置 DEBUG 的值为 `express:router`，只查看路由部分的日志；设置 DEBUG 的值为 `express:application`，只查看应用部分的日志，依此类推。

express-generated 应用

使用 `express` 命令行生成的应用也使用了 `debug` 模块，它的命名空间限制在应用中。

如果您通过下述命令生成应用：

```
$ express sample-app
```

则可通过下述命令打开调试信息：

```
$ DEBUG=sample-app node ./bin/www
```

可通过逗号隔开的名字列表来指定多个调试命名空间，如下所示：

```
$ DEBUG=http,mail,express:* node index.js
```

请查阅 调试指南 获取更多有关 `debug` 的文档。

6

代理之后的 Express

当在代理服务器之后运行 Express 时，请将应用变量 `trust proxy` 设置（使用 `app.set()`）为下述列表中的一项。

如果没有设置应用变量 `trust proxy`，应用将不会运行，除非 `trust proxy` 设置正确，否则应用会误将代理服务器的 IP 地址注册为客户端 IP 地址。

类型	**Value**
Boolean	如果为 `true`，客户端 IP 地址为 `X-Forwarded-*` 头最左边的项。 如果为 `false`, 应用直接面向互联网，客户端 IP 地址从 `req.connection.remoteAddress` 得来，这是默认的设置。

类型	Value

IP 地址	IP 地址、子网或 IP 地址数组和可信的子网。下面是预配置的子网列表。
	
	• loopback - `127.0.0.1/8`, `::1/128`
	• linklocal - `169.254.0.0/16`, `fe80::/10`
	• uniquelocal - `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`, `fc00::/7`
	
	使用如下方式设置 IP 地址：
	
	`app.set('trust proxy', 'loopback') // 指定唯一子网`
	`app.set('trust proxy', 'loopback, 123.123.123.123') // 指定子网和 IP 地址`
	`app.set('trust proxy', 'loopback, linklocal, uniquelocal') // 指定多个子网`
	`app.set('trust proxy', ['loopback', 'linklocal', 'uniquelocal']) // 使用数组指定多个子网`
	
	当指定地址时，IP 地址或子网从地址确定过程中被除去，离应用服务器最近的非受信 IP 地址被当作客户端 IP 地址。

数字	将代理服务器前第 `n` 跳当作客户端。

类型	Value

函数	定制实现，只有在您知道自己在干什么时才能这样做。
	`app.set('trust proxy', function (ip) {`
	`  if (ip === '127.0.0.1' || ip === '123.123.123.123') return true; // 受信的 IP 地址`
	`  else return false;`
	`})`

设置 `trust proxy` 为非假值会带来两个重要变化：

- 反向代理可能设置 `X-Forwarded-Proto` 来告诉应用使用 https 或简单的 http 协议。请参考 req.protocol。
- req.ip 和 req.ips 的值将会由 `X-Forwarded-For` 中列出的 IP 地址构成。

`trust proxy` 设置由 proxy-addr 软件包实现，请参考其文档了解更多信息。

7

迁移到 Express 4

概览

Express 4 是对 Express 3 的一个颠覆性改变，也就是说如果您更新了 Express， Express 3 应用会无法工作。

该章包含如下内容：

- Express 4 的变化。
- 一个从 Express 3 迁移到 Express 4 的示例。
- 迁移到 Express 4 应用生成器。

Express 4 的变化

Express 4 的主要变化如下：

- 对 Express 内核和中间件系统的改进。不再依赖 Connect 和内置的中间件，您需要自己添加中间件。
- 对路由系统的改进。
- 其他变化。

其他变化请参考：

- 4.x 中的新功能
- 从 3.x 迁移到 4.x

对 Express 内核和中间件系统的改进

Express 4 不再依赖 Connect，而且从内核中移除了除 `express.static` 外的所有内置中间件。也就是说现在

的 Express 是一个独立的路由和中间件 Web 框架，Express 的版本升级不再受中间件更新的影响。

移除了内置的中间件后，您必须显式地添加所有运行应用需要的中间件。请遵循如下步骤：

1. 安装模块：`npm install --save <module-name>`
2. 在应用中引入模块：`require('module-name')`
3. 按照文档的描述使用模块：`app.use( ... )`

下表列出了 Express 3 和 Express 4 中对应的中间件。

Express 3	Express 4
`express.bodyParser`	body-parser + multer
`express.compress`	compression
`express.cookieSession`	cookie-session
`express.cookieParser`	cookie-parser
`express.logger`	morgan
`express.session`	express-session
`express.favicon`	serve-favicon
`express.responseTime`	response-time
`express.errorHandler`	errorhandler
`express.methodOverride`	method-override
`express.timeout`	connect-timeout
`express.vhost`	vhost
`express.csrf`	csurf

`express.directory`	serve-index
`express.static`	serve-static

这里是 Express 4 的所有中间件列表。

多数情况下，您可以直接使用 Express 4 中对应的中间件替换 Express 3 中的中间件，请参考 GitHub 中的模块文档了解更多信息。

app.use accepts parameters

在 Express 4 中，可以从路由句柄中读取参数，以该参数的值作为路径加载中间件，比如像下面这样：

```
app.use('/book/:id', function(req, res, next) {
  console.log('ID:', req.params.id);
  next();
});
```

路由系统

应用现在隐式地加载路由中间件，因此不需要担心中间件加载顺序。

定义路由的方式依然未变，但是新的路由系统有两个新功能能帮助您组织路由：

- 新方法 `app.route()` 可以为路由路径创建链式路由句柄。
- 新类 `express.Router` 可以创建可挂载的模块化路由句柄。

app.route() 方法

新增加的 `app.route()` 方法可为路由路径创建链式路由句柄。由于路径在一个地方指定，会让路由更加模块

化，也能减少代码冗余和拼写错误。请参考 Router() 文档获取更多关于路由的信息。

下面是一个使用 `app.route()` 方法定义链式路由句柄的例子。

```
app.route('/book')
  .get(function(req, res) {
    res.send('Get a random book');
  })
  .post(function(req, res) {
    res.send('Add a book');
  })
  .put(function(req, res) {
    res.send('Update the book');
  });
```

express.Router 类

另外一个帮助组织路由的是新加的 `express.Router` 类，可使用它创建可挂载的模块化路由句柄。`Router` 类是一个完整的中间件和路由系统，鉴于此，人们常称之为“迷你应用”。

下面的例子创建了一个模块化的路由，并加载了一个中间件，然后定义了一些路由，并且在主应用中将其挂载到指定路径。

在应用目录下创建文件 `birds.js`，其内容如下：

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

// 特针对于该路由的中间件
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});
// 定义主页路由
router.get('/', function(req, res) {
  res.send('Birds home page');
```

```
});
// 定义 about 页面路由
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

在应用中加载该路由：

```
var birds = require('./birds');
...
app.use('/birds', birds);
```

应用现在就可以处理发送到 `/birds` 和 `/birds/about` 的请求，并且会调用特针对于该路由的 `timeLog` 中间件。

其他变化

下表列出了 Express 4 中其他一些尽管不大，但是非常重要的变化。

对象	描述
Node	Express 4 需要 Node 0.10.x 或以上版本，已经放弃了对 Node 0.8.x 的支持。
`http.createServer()`	<已经不再需要 code>http 模块，除非您需要直接使用它（socket.io/SPDY/HTTPS），使用 `app.listen()` 启动应用。

`app.configure()`	已经删除 `app.configure()` ，使用 `process.env.NODE_ENV` 或者 `app.get('env')` 检测环境并配置应用。
`json spaces`	Express 4 默认禁用 `json spaces` 属性。
`req.accepted()`	使用 `req.accepts()` 、`req.acceptsEncodings()` 、`req.acceptsCharsets()` 和 `req.acceptsLanguages()` 。
`res.location()`	不再解析相对 URLs。
`req.params`	从数组变为对象。
`res.locals`	从函数变为对象。
`res.headerSent`	变为 `res.headersSent` 。
`app.route`	变为 `app.mountpath` 。
`res.on('header')`	已删除。
`res.charset`	已删除。
`res.setHeader('Set-`	功能仅限于设置基本的

`Cookie', val)`	cookie 值，使用 `res.cookie()` 访问更多功能。

迁移示例

下面是一个从 Express 3 迁移到 Express 4 的例子，请留意 `app.js` 和 `package.json`。

Express 3 应用

`app.js`

请看如下 Express 3 应用，其 `app.js` 文件内容如下：

```
var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// 环境
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname,
'views'));
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.methodOverride());
app.use(express.session({ secret: 'your secret
here' }));
app.use(express.bodyParser());
app.use(app.router);
app.use(express.static(path.join(__dirname,
'public')));
```

```
// 只为开发使用
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'),
function(){
  console.log('Express server listening on port
' + app.get('port'));
});
```

package.json

相应的 `package.json` 文件内容如下：

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.12.0",
    "jade": "*"
  }
}
```

迁移过程

首先安装 Express 4 应用需要的中间件，使用如下命令将 Express 和 Jade 更新至最新版本：

```
$ npm install serve-favicon morgan method-
override express-session body-parser multer
errorhandler express@latest jade@latest --save
```

按如下方式修改 `app.js` 文件：

1. `express.favicon`、`express.logger`、`express.methodOverride`、`express.session`、`express.bodyParser`、`express.errorHandler` 这些内置中间件在 `express` 对象中已经没有了，您必须手动安装相应的中间件，并在应用中加载它们。
2. 不需要加载 `app.router` ，它不再是一个合法的 Express 4 对象，删掉 `app.use(app.router);` 。
3. 确保加载中间件的顺序正确，加载完应用路由后再加载 `errorHandler` 。

Express 4 应用

package.json

运行上述 npm 命令后，会将 `package.json` 文件更新为：

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "body-parser": "^1.5.2",
    "errorhandler": "^1.1.1",
    "express": "^4.8.0",
    "express-session": "^1.7.2",
    "jade": "^1.5.0",
    "method-override": "^2.1.2",
    "morgan": "^1.2.2",
    "multer": "^0.1.3",
    "serve-favicon": "^2.0.1"
  }
}
```

app.js

删掉非法的代码，加载需要的中间件，再做一些必要的修改，新的 `app.js` 内容如下：

```
var http = require('http');
var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var path = require('path');

var favicon = require('serve-favicon');
var logger = require('morgan');
var methodOverride = require('method-
override');
var session = require('express-session');
var bodyParser = require('body-parser');
var multer = require('multer');
var errorHandler = require('errorhandler');

var app = express();

// 环境
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname,
'views'));
app.set('view engine', 'jade');
app.use(favicon(__dirname +
'/public/favicon.ico'));
app.use(logger('dev'));
app.use(methodOverride());
app.use(session({ resave: true,
                  saveUninitialized: true,
                  secret: 'uwotm8' }));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true
}));
app.use(multer());
app.use(express.static(path.join(__dirname,
'public')));
```

```
app.get('/', routes.index);
app.get('/users', user.list);

// 加载路由完成后才能加载错误处理中间件
if ('development' == app.get('env')) {
  app.use(errorHandler());
}

var server = http.createServer(app);
server.listen(app.get('port'), function(){
  console.log('Express server listening on port
' + app.get('port'));
});
```

除非需要直接使用 `http` 模块（socket.io/SPDY/HTTPS），否则不必加载它，可使用如下方式启动应用：

```
app.listen(app.get('port'), function(){
  console.log('Express server listening on port
' + app.get('port'));
});
```

运行应用

迁移完成后，应用就变成了 Express 4 应用。为了确保迁移成功，使用如下命令启动应用：

```
$ node .
```

输入 http://localhost:3000，即可看到经由 Express 4 渲染的主页。

迁移到 Express 4 应用生成器

生成 Express 应用的命令行还是 `express`，为了升级到最新版本，您必须首先卸载 Express 3 的应用生成器，然后安装新的 `express-generator`。

安装

如果您已经安装了 Express 3 应用生成器，请使用如下命令卸载：

```
$ npm uninstall -g express
```

根据您的文件目录权限，您可能需要以 `sudo` 权限执行该命令。

然后安装新的生成器：

```
$ npm install -g express-generator
```

根据您的文件目录权限，您可能需要以 `sudo` 权限执行该命令。

现在系统的 `express` 命令就升级为 Express 4 应用生成器了。

应用生成器的变化

大部分命令参数和使用方法都维持不变，除过如下选项：

- 删掉了 `--sessions` 选项。
- 删掉了 `--jshtml` 选项。
- 增加了 `--hogan` 选项以支持 Hogan.js。

示例

运行下述命令创建一个 Express 4 应用：

```
$ express app4
```

如果查看 `app4/app.js` 的内容，会发现应用需要的所有中间件（不包括 `express.static`）都作为独立模块载入，而且再不显式地加载 `router` 中间件。

您可能还会发现，和旧的生成器生成的应用相比，`app.js` 现在成了一个 Node 模块。

安装完依赖后，使用如下命令启动应用：

```
$ npm start
```

如果看一看 `package.json` 文件中的 npm 启动脚本，会发现启动应用的真正命令是 `node ./bin/www`，在 Express 3 中则为 `node app.js`。

Express 4 应用生成器生成的 `app.js` 是一个 Node 模块，不能作为应用（除非修改代码）单独启动，需要通过一个 Node 文件加载并启动，这里这个文件就是 `node ./bin/www`。

创建或启动 Express 应用时，`bin` 目录或者文件名没有后缀的 `www` 文件都不是必需的，它们只是生成器推荐的做法，请根据需要修改。

如果不想保留 `www`，想让应用变成 Express 3 的形式，则需要删除 `module.exports = app;`，并在 `app.js` 末尾粘贴如下代码。

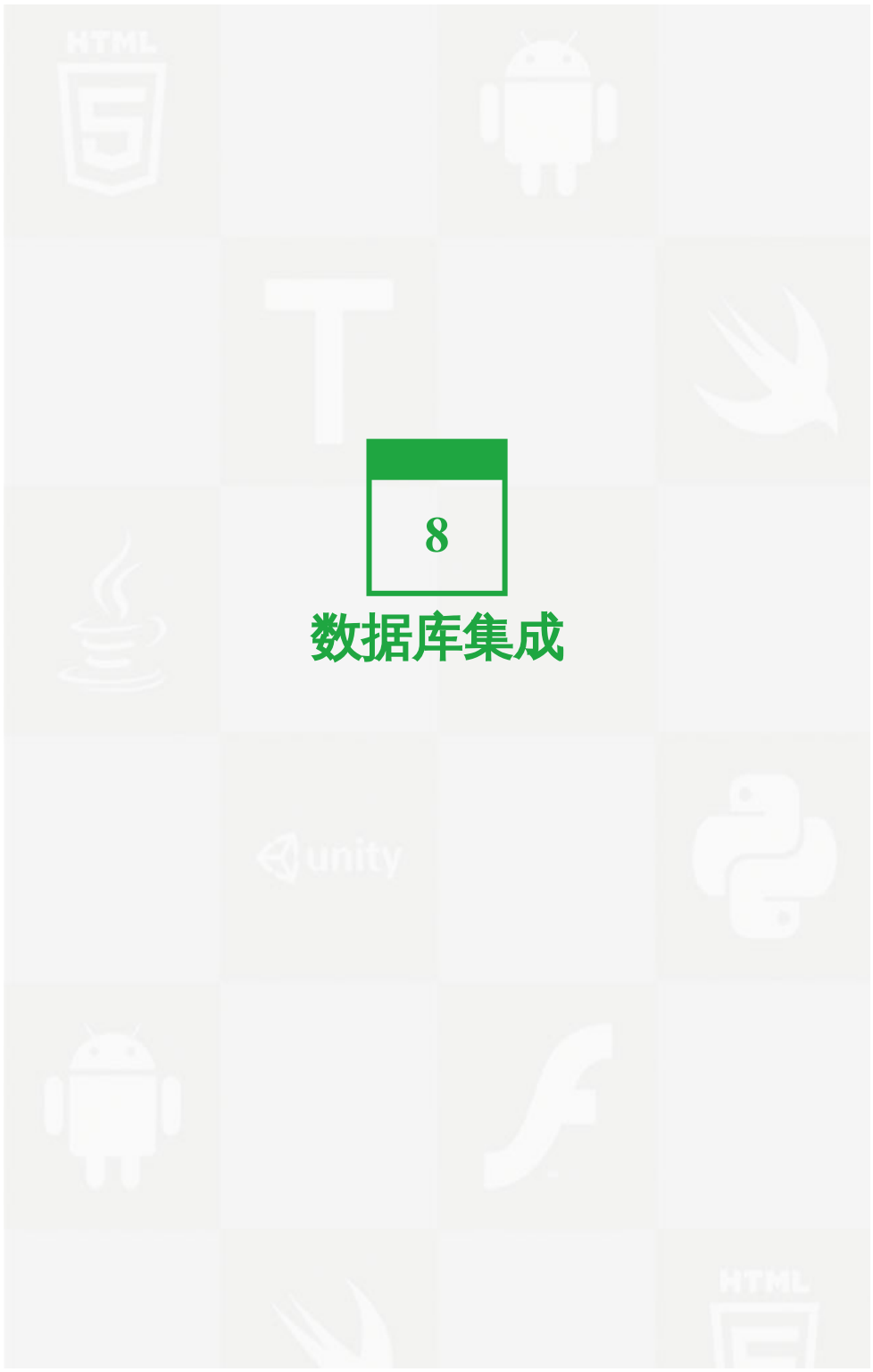
```
app.set('port', process.env.PORT || 3000);

var server = app.listen(app.get('port'),
function() {
  debug('Express server listening on port ' +
server.address().port);
});
```

记得在 `app.js` 上方加入如下代码加载 `debug` 模块。

```
var debug = require('debug')('app4');
```

然后将 `package.json` 文件中的 `"start": "node ./bin/www"` 修改为 `"start": "node app.js"`。

现在就将 `./bin/www` 的功能又改回到 `app.js` 中了。我们并不推荐这样做，这个练习只是为了帮助大家理解 `./bin/www` 是如何工作的，以及为什么 `app.js` 不能再自己启动。



8

数据库集成

为 Express 应用添加连接数据库的能力，只需要加载相应数据库的 Node.js 驱动即可。这里将会简要介绍如何为 Express 应用添加和使用一些常用的数据库 Node 模块。

- Cassandra
- CouchDB
- LevelDB
- MySQL
- MongoDB
- Neo4j
- PostgreSQL
- Redis
- SQLite
- ElasticSearch

这些数据库驱动只是其中一部分，可在 npm 官网 查找更多驱动。

Cassandra

模块: cassandra-driver
安装

```
$ npm install cassandra-driver
```

示例

```
var cassandra = require('cassandra-driver');
var client = new cassandra.Client({
contactPoints: ['localhost']});

client.execute('select key from system.local',
function(err, result) {
```

```
  if (err) throw err;
  console.log(result.rows[0]);
});
```

CouchDB

模块: nano
安装

```
$ npm install nano
```

示例

```
var nano = require('nano')
('http://localhost:5984');
nano.db.create('books');
var books = nano.db.use('books');

//在数据库 books 中插入一条图书记录
books.insert({name: 'The Art of war'}, null,
function(err, body) {
  if (!err){
    console.log(body);
  }
});

//返回图书列表
books.list(function(err, body){
  console.log(body.rows);
}
```

LevelDB

模块: levelup
安装

```
$ npm install level levelup leveldown
```

示例

```
var levelup = require('levelup');
var db = levelup('./mydb');

db.put('name', 'LevelUP', function (err) {

  if (err) return console.log('Ooops!', err);
  db.get('name', function (err, value) {
    if (err) return console.log('Ooops!', err);
    console.log('name=' + value)
  });

});
```

MySQL

模块: mysql
安装

```
$ npm install mysql
```

示例

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'dbuser',
  password : 's3kreee7'
});

connection.connect();

connection.query('SELECT 1 + 1 AS solution',
function(err, rows, fields) {
  if (err) throw err;
  console.log('The solution is: ',
```

```
rows[0].solution);
});

connection.end();
```

MongoDB

模块: mongoskin
安装

```
$ npm install mongoskin
```

示例

```
var db =
require('mongoskin').db('localhost:27017/animal
s');

db.collection('mamals').find().toArray(function
(err, result) {
  if (err) throw err;
  console.log(result);
});
```

如果想获取 MongoDB 的对象模型驱动，请参考 Mongoose。

Neo4j

模块: apoc
安装

```
$ npm install apoc
```

示例

```
var apoc = require('apoc');

apoc.query('match (n) return n').exec().then(
  function (response) {
    console.log(response);
  },
  function (fail) {
    console.log(fail);
  }
);
```

PostgreSQL

模块: pg
安装

```
$ npm install pg
```

示例

```
var pg = require('pg');
var conString =
"postgres://username:password@localhost/databas
e";

pg.connect(conString, function(err, client,
done) {

  if (err) {
    return console.error('error fetching client
from pool', err);
  }
  client.query('SELECT $1::int AS number',
['1'], function(err, result) {
    done();
    if (err) {
      return console.error('error running
query', err);
```

```
    }
    console.log(result.rows[0].number);
  });

});
```

Redis

模块: redis
安装

```
$ npm install redis
```

示例

```
var client = require('redis').createClient();

client.on('error', function (err) {
  console.log('Error ' + err);
});

client.set('string key', 'string val',
redis.print);
client.hset('hash key', 'hashtest 1', 'some
value', redis.print);
client.hset(['hash key', 'hashtest 2', 'some
other value'], redis.print);

client.hkeys('hash key', function (err,
replies) {

  console.log(replies.length + ' replies:');
  replies.forEach(function (reply, i) {
    console.log('    ' + i + ': ' + reply);
  });

  client.quit();
```

```
});
```

SQLite

模块: sqlite3
安装

```
$ npm install sqlite3
```

示例

```
var sqlite3 = require('sqlite3').verbose();
var db = new sqlite3.Database(':memory:');

db.serialize(function() {

  db.run('CREATE TABLE lorem (info TEXT)');
  var stmt = db.prepare('INSERT INTO lorem
VALUES (?)');

  for (var i = 0; i < 10; i++) {
    stmt.run('Ipsum ' + i);
  }

  stmt.finalize();

  db.each('SELECT rowid AS id, info FROM
lorem', function(err, row) {
    console.log(row.id + ': ' + row.info);
  });
});

db.close();
```

ElasticSearch

模块: elasticsearch
安装

```
$ npm install elasticsearch
```

示例

```
var elasticsearch = require('elasticsearch');
var client = elasticsearch.Client({
  host: 'localhost:9200'
});

client.search({
  index: 'books',
  type: 'book',
  body: {
    query: {
      multi_match: {
        query: 'express js',
        fields: ['title', 'description']
      }
    }
  }
}).then(function(response) {
  var hits = reponse.hits.hits;
}, function(error) {
  console.trace(error.message);
});
```