第8章 Node.js框架与Express

Node.js开发实战教程

内容导航

Contents

8.1 Node.js框架概述

- MVC框架
- REST API框架
- 全栈框架与实时框架
- Node.js框架的选择

8.2 Express框架基础
8.3 Express路由
8.4 Express中间件
8.5 视图与模板引擎
8.6 Express与数据库集成
8.7 Express错误处理
8.8 实战演练——图书信息管理的REST API接口

Section 00

【学习目标】

（1）了解Node.js框架；

（2）会根据项目需要选择合适的框架。

框架的英文为Framework，有骨骼、支架的含义。框架被定义为整个或部分系统的可重用设计，或者是可被应用开发人员定制的应用骨架。开发人员使用框架可以快速高效地开发网络应用程序，而不需要借助任何其他第三方Web服务器、应用服务器和工具。

Section 01

【MVC框架】

类Sinatra框架

- Express：目前最流行的Node.js Web开发框架。
- Koa：Node.js MVC框架的后起之秀，是由Express团队开发的。
- Hapi：致力于完全分离Node.js的HTTP服务器、路由以及业务逻辑。
- Flatiron：Node.js和浏览器的框架组件。

类Rails框架

- Sails：可靠的、可伸缩的优秀框架，提供建立任何规模的Web应用所需要的所有功能。
- Geddy：简单、结构化的Node.js MVC开发框架。

Section 02

【REST API框架】

- REST API框架旨在为跨平台应用提供统一的数据模型。
- 视图渲染则由前端或客户端自行解决。
  - Restify：基于Node.js的REST应用框架，支持服务器端和客户端。
  - LoopBack：建立在Express基础上的企业级框架，只需编写少量代码就能创建动态端到端的REST API，可以使Node.js应用程序方便地与各种设备通过API进行互联。

MVC框架等Web框架大都支持REST API应用的开发。

Section 03

【全栈框架】

- 全栈框架提供必需的应用开发基础库、完整的模版引擎、网络Socket以及持久化的库来对实时可扩展的网络和移动应用进行构建。
- 全栈框架是快速构建大型Web应用程序的利器。
  - Meteor：业界领先的全栈框架，是JavaScript框架的集大成者，拥有专业的开发团队支持。
  - Mean.IO：完全的JavaScript开发框架，旨在简化和加速开发基于MEAN栈的网络应用程序。

MEAN的含义与LAMP非常类似，代表的是MongoDB、Express、Angular和Node.js捆绑在一起的组合。

Section 04

【实时框架】

- 实时框架是指那些支持WebSocket双向通信功能，能够在服务器和客户端实现实时通信的框架。
- 实时框架适合开发单页Web应用、多用户游戏、聊天客户端、网络应用、交易平台以及所有需要将数据从服务端实时推送到客户端的应用。
- Socket.IO是使用最广泛的Node.js实时通信解决方案，许多框架都增加对它的支持来提供实时特性。
- 常用的实时框架
  - Meteor：业界领先的全栈框架，具有实时通信功能。
  - Mean.IO：完全的JavaScript开发框架，具有实时通信功能。
  - SocketStream：实时Web应用程序的框架，专注于客户端和服务端数据的快速同步，支持前后端数据的实时更新。
  - Derby：作为运行在Node.js、MongoDB和Redis基础上的全栈框架，拥有一个称为Racer的数据同步引擎。

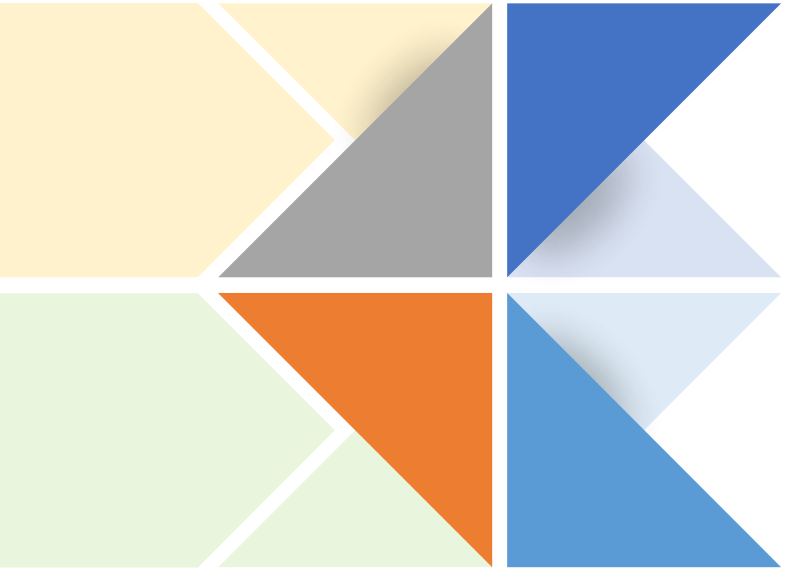
Section 05

【Node.js框架的选择】

- 初学者要使用Node.js开发Web后端服务，可以考虑从Express开始；如果对JavaScript的Promise和async等异步编程比较熟悉，也可以考虑从Koa框架开始。
- 要开发内容管理站点，可以选择Meteor全栈框架。
- 企业应用开发首选Egg。Egg是阿里基于Node.js和Koa的企业级应用开发框架，特别适合国内用户。另外，Sails和Loopback也都适合开发企业应用。
- 开发实时Web应用程序推荐使用Meteor。
- 构建基于微服务架构的REST API首选Hapi，也可以选择restify。对于大型的复杂应用程序，可考虑选用LoopBack。

内容导航

Contents



Section 00

【学习目标】

（1）初步了解Express框架；
（2）理解Express工作机制；
（3）使用Express生成器创建项目脚手架。

Express是一个简洁而灵活的Web应用程序开发框架，为Web和移动应用程序提供一系列强大的功能，可以用来快速地搭建一个功能完整的Web网站。

Express实质上是在Node.js内置的http模块的基础上构建了一层抽象，目的是拓展Node.js的功能以提高开发效率。所有Express功能都可以直接使用Node.js实现。使用Express的同时仍然可以使用Node.js原生方法。

Section 01

【“Hello World”示例程序】

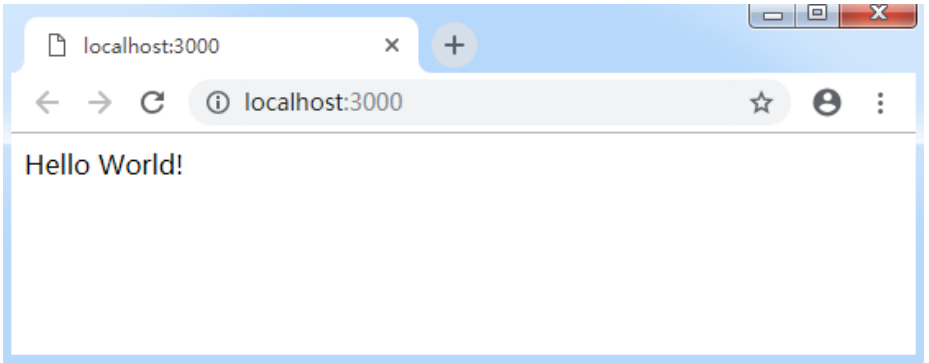
- 创建一个项目目录，并在其中创建一个名为helloworld的Node.js项目。
- 安装Express包。
- 示例程序

```
const express = require('express'); //导入express模块
const app = express();//创建一个名为app的Express应用
//设置访问根路径的路由，并将响应的信息设置为“Hello World!”
app.get('/', (req, res) => res.send('Hello World!'))
//设置服务程序监听的端口，并输出服务启动成功的信息
app.listen(3000, () => console.log('示例app监听端口3000!'))
```

- 运行该应用程序并进行测试

```
C:\nodeapp\ch08\helloworld>node hello.js
示例app监听端口3000!
```

在浏览器中访问进行实测



Section 02

【使用Express生成器创建项目脚手架】

▸▸▸ Express生成器

- 全局安装express-generator

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

- express-generator包含的express命令行工具

```
express -h
  Usage: express [options] [dir]
  Options:
        --version        输出版本号
    -e, --ejs            添加对EJS模板引擎的支持
        --pug            添加对Pug模板引擎的支持
        --hbs            添加对Handlebars模板引擎的支持
    -H, --hogan          添加对hogan.js模板引擎的支持
    -v, --view <engine>  添加对视图引擎的支持(dust|ejs|hbs|hjs|jade|pug|twig|vash) (默认为jade)
        --no-view        使用静态HTML而不用视图引擎
    -c, --css <engine>   添加样式表引擎的支持(less|stylus|compass|sass) (默认普通的CSS)
        --git            添加 .gitignore
    -f, --force          强制在非空目录下创建
    -h, --help           输出使用方法
```

Section 02

【使用Express生成器创建项目脚手架】

▸▸▸ 创建Express应用

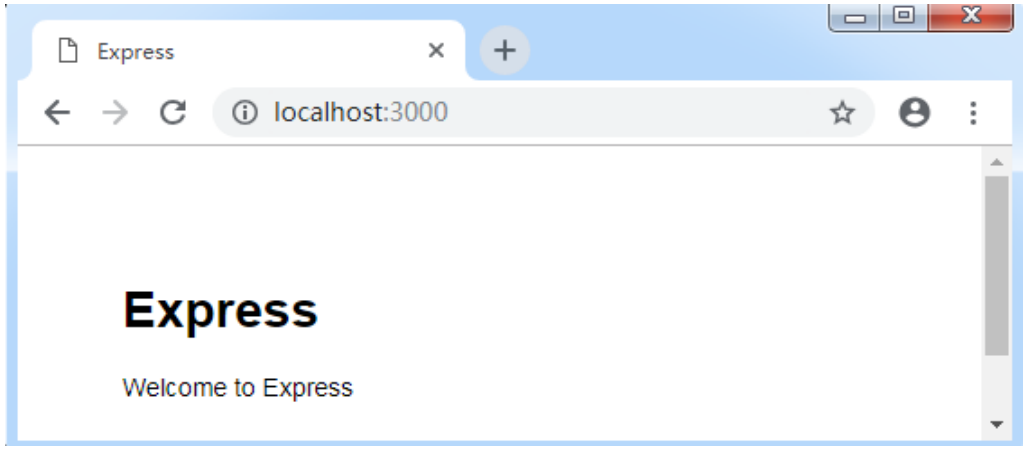
➢ 创建名称为myapp的Express应用

```
C:\nodeapp\ch08>express --view=pug myapp
   create : myapp\
(此处省略)
#以下为提示信息
   change directory:
     > cd myapp
   install dependencies:
     > npm install     #安装所有依赖包
   run the app:
     > SET DEBUG=myapp:* & npm start     #启动此应用程序
```

➢ 运行该应用程序并进行测试

```
C:\nodeapp\ch08\myapp>SET DEBUG=myapp:* & npm start
> myapp@0.0.0 start C:\nodeapp\ch08\myapp
> node ./bin/www
  myapp:server Listening on port 3000 +0ms
```

在浏览器中访问Express项目脚手架程序



Section 02

【使用Express生成器创建项目脚手架】

▶▶▶ Express项目脚手架的目录结构

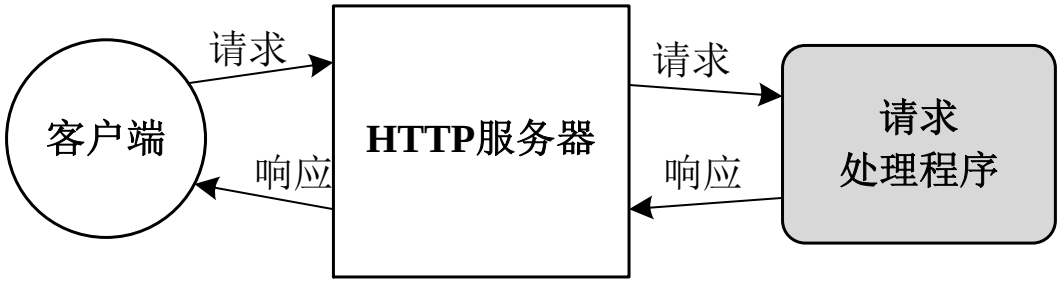
```
C:\NODEAPP\CH08\MYAPP
│  app.js                核心文件（项目入口文件）
│  package.json          项目配置文件
├─bin                    存放启动项目的脚本文件
│      www
├─public                 静态资源文件 [前端]
│  ├─images
│  ├─javascripts
│  └─stylesheets
│          style.css
├─routes                 路由文件 [后端]
│      index.js
│      users.js
└─views                  视图模板文件 [前端]
        error.pug
        index.pug
        layout.pug
```

Section 03

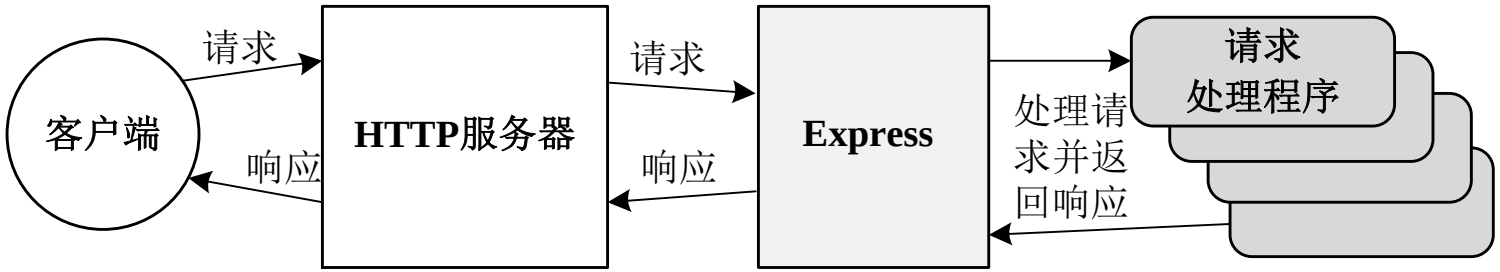
【Express工作机制】

Web应用程序架构

- http模块支持的Web应用程序架构



- Express的应用程序架构



Section 03

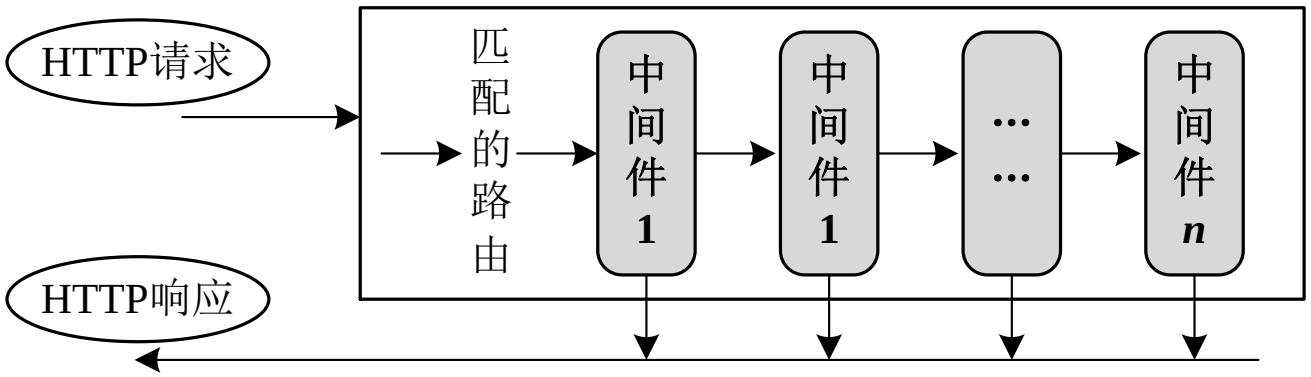
【Express工作机制】

▸▸▸ 路由

- 路由决定应用程序如何响应客户端对特定端点的请求，端点可以是一个URI（或路径）和特定的HTTP请求方法（GET、POST等）。
- 路由根据请求的URI和HTTP方法来决定请求的处理方式。

▸▸▸ 中间件

- 中间件是将请求处理函数进行模块化分解后的若干子处理函数，一系列子处理函数可以形成一个中间件堆栈。



Section 03

【Express工作机制】

▸▸▸ 请求和响应

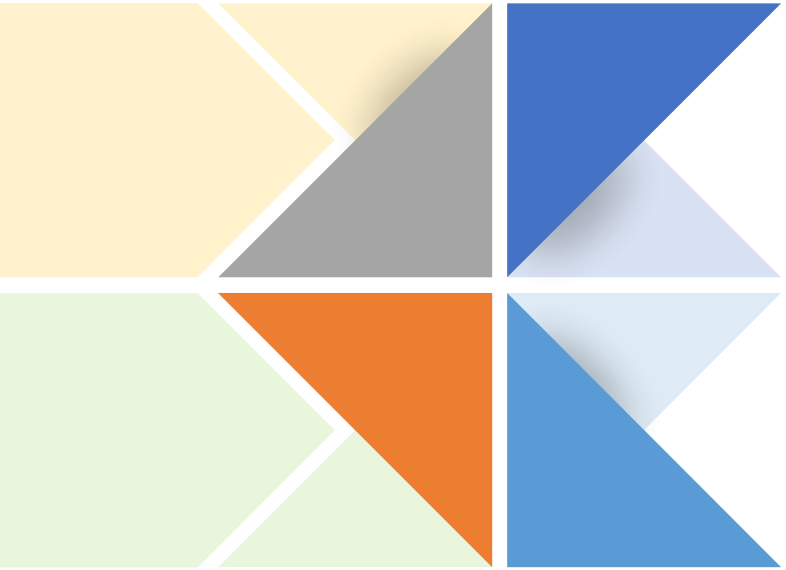
- 通过req.ip属性获取发送客户端IP地址。
- 使用res.endFile()方法可以直接发送文件。
- 使用res.redirect()方法可以直接进行重定向。
- 通过中间件函数对请求和响应对象进行修改。

▸▸▸ 视图渲染

- 通过向模板传递参数来动态渲染HTML页面，Express支持多种模版引擎。

内容导航

Contents



Section 00

【学习目标】

（1）理解路由结构；
（2）掌握Express路由的用法；
（3）理解和使用Express路由器。

路由用于定义应用的端点（URI）以及响应客户端的请求。路由通过对HTTP方法和URI的组合进行映射来实现对不同请求的分别处理。

Section 01

【路由结构】

- 路由由URI、HTTP方法（GET、POST等）和若干回调函数组成：

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

- 简单的路由示例

```
var express = require('express')
var app = express()
// 当GET请求被提交到主页时，回应"欢迎访问！"消息
app.get('/', function (req, res) {
  res.send('欢迎访问！')
})
```

Section 02

【路由方法】

- 为访问应用程序根路径的GET和POST方法定义的路由：

```
// GET方法路由
app.get('/', function (req, res) {
  res.send('到主页的GET请求')
})
// POST方法路由
app.post('/', function (req, res) {
  res.send('到主页的POST请求')
})
```

- 使用app.all()方法处理所有HTTP请求

```
app.all('/secret', function (req, res, next) {
  console.log('查看秘密数据 ...')
  next() // 将控制传递给下一个处理函数
})
```

Section 03

【路由路径】

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

```
app.get('/', function (req, res) {  //将匹配到根路由/的请求
  res.send('root')
})
app.get('/about', function (req, res) {//将匹配到/about的请求
  res.send('about')
})
app.get('/random.text', function (req, res) {//将匹配到/random.text的请求
  res.send('random.text')
})
```

Section 03

【路由路径】

- 基于字符串模式的路由路径

```
app.get('/ab?cd', function (req, res) {//将匹配/acd和/abcd
  res.send('ab?cd')
})
app.get('/ab*cd', function (req, res) {//匹配/abcd、/abxcd、/abRANDOMcd、/ab123cd等
  res.send('ab*cd')
})
```

- 基于正则表达式的路由路径

```
app.get(/a/, function (req, res) {//匹配含有“a”的任何路径
  res.send('/a/')
})
//以下路径匹配butterfly和dragonfly，但不匹配butterflyman，dragonflyman等
app.get(/.*fly$/, function (req, res) {
  res.send('/.*fly$/')
})
```

Section 04

【路由参数】

- 在路由中使用参数表示变量名（使用冒号标识参数）。
- 举例说明含参数的路由路径与请求URL和req.params（请求参数）之间的对应关系：

  路由路径（含参数）： /users/:userId/books/:bookId
  请求URL： http://localhost:3000/users/34/books/8989
  req.params： { "userId": "34", "bookId": "8989" }
- 要使用路由参数定义路由，只需在路由路径中指定路由参数，例如：

```
app.get('/users/:userId/books/:bookId', function (req, res) {
  res.send(req.params)
})
```

Section 05

【路由处理程序】

- 路由处理程序可以采用函数、函数数组或两者组合的形式。
- 单个回调函数可以处理路由：

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

- 多个回调函数可以处理路由（已指定next参数）：

```
app.get('/example/b', function (req, res, next) {
  console.log('响应将由next()函数发送 ...')
  next();
}, function (req, res) {
  res.send('来自B的问候!');
});
```

Section 06

【响应方法】

方法	说明
res.download()	提示要下载的文件
res.end()	终结响应进程
res.json()	发送JSON响应
res.jsonp()	发送支持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字符串表示形式作为响应体发送

Section 07

【app.route()方法】

- app.route()方法可用来为路由路径创建链式的路由处理程序。

```
app.route('/book')
  .get(function (req, res) {
    res.send('获得图书信息');
  })
  .post(function (req, res) {
    res.send('添加一本图书');
  })
  .put(function (req, res) {
    res.send('修改图书信息');
  });
```

Section 08

【Express路由器】

Express路由器概述

- Express应用程序拆分为多个子应用程序，即路由器（Router）。
- 路由器是一个完整的中间件和路由系统，是只能执行中间件和路由功能的小型应用程序。
- 实际上每个Express应用程序都有一个内置的路由器实例。
- 路由器的行为与中间件类似，可以通过use()方法来调用其他的路由器实例。
- 顶级Express对象通过提供Router()方法来创建路由器实例。

Section 08

【Express路由器】

▸▸▸ Express路由器示例

定义路由器模块

➢ 鸟类：birds.js文件

```
var express = require('express');
var router = express.Router();
// 该路由器专用的中间件
router.use(function timeLog (req, res, next) {
  console.log('时间: ', Date.now());
  next();
});
// 定义主页路由
router.get('/', function (req, res) {
  res.send('鸟类主页');
});
// 定义/about路由
router.get('/about', function (req, res) {
  res.send('关于鸟类');
});
module.exports = router;
```

➢ 鱼类：fishes.js文件

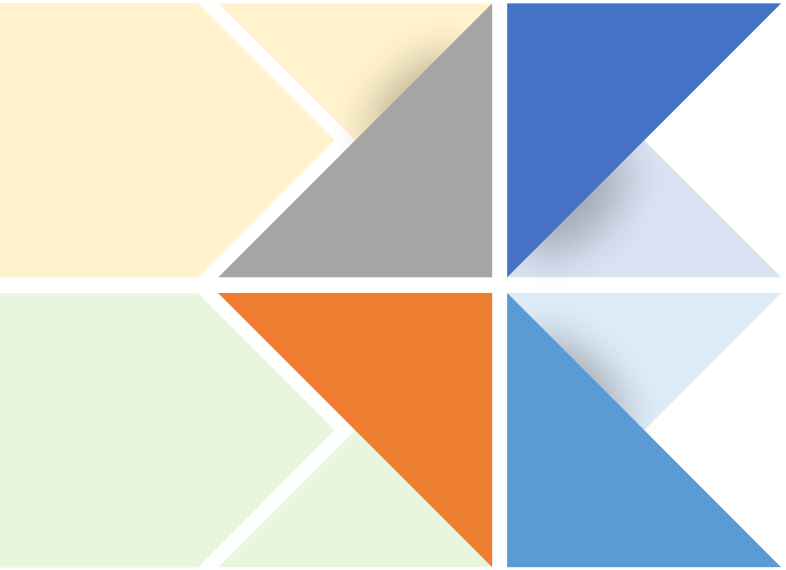
```
 ......
```

主程序中加载路由器模块

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

内容导航

Contents



Section 00

【学习目标】

（1）进一步理解中间件；
（2）掌握Express中间件的用法；
（3）编写自己的Express中间件。

Express应用程序实质上就是一系列的中间件调用。中间件就是处理HTTP请求的函数，用来完成各种特定的任务。

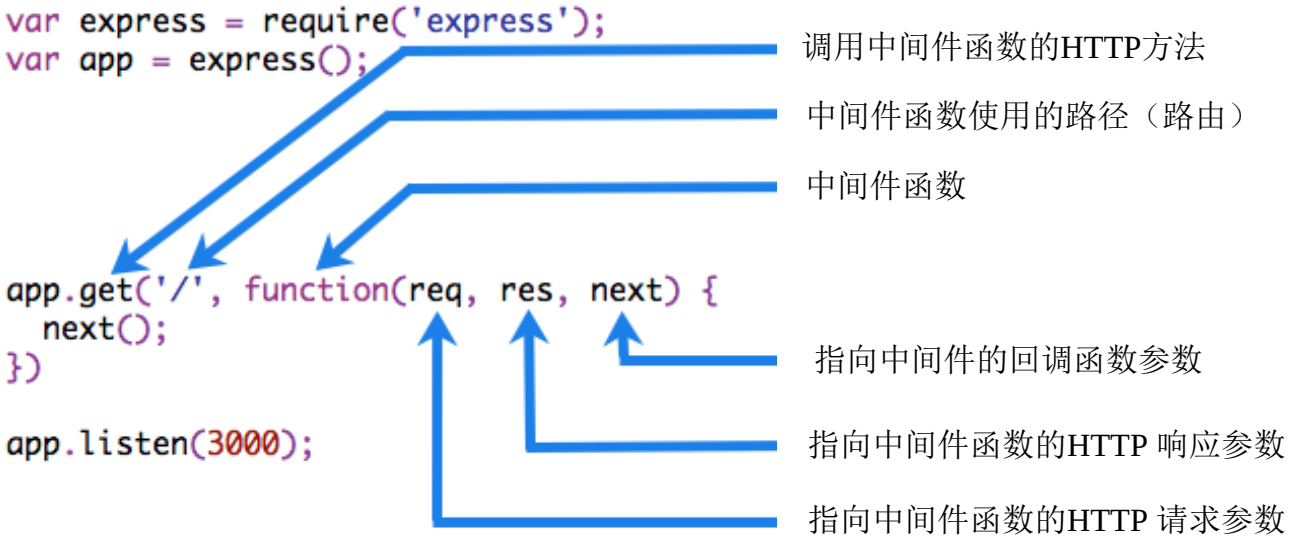
Express最显著的特点是，当一个中间件处理完成请求之后，再将请求传递给下一个中间件。

Section 01

【进一步理解中间件】

- 中间件是在应用程序的“请求—响应”周期中能够访问请求对象（req）、响应对象（res）和next()函数的函数。
- 可以将中间件视为在收到请求后和发送响应之前这个阶段执行的一些函数。
- 中间件函数执行以下任务。
  - 执行任何代码。
  - 对请求和响应对象进行修改。
  - 终结“请求—响应”周期。
  - 调用堆栈中下一个中间件。
- 如果当前中间件函数没有终结“请求—响应”周期，则它必须调用next()函数将控制传递给下一个中间件函数。

中间件函数调用的要素



Section 02

【使用中间件】

▸▸▸ 应用程序级中间件

- 使用app.use()和app.METHOD()函数将应用程序级中间件绑定到应用对象的实例上。
- 示例：没有挂载路径的中间件函数。

```
var app = express();
app.use(function (req, res, next) {
  console.log('时间:', Date.now());
  next();
});
```

- 示例：挂在到/user/:id路径的中间件函数

```
app.use('/user/:id', function (req, res, next) {
  console.log('请求类型:', req.method);
  next();
});
```

- 示例：对/user/:id路径的GET请求

```
app.get('/user/:id', function (req, res, next) {
  res.send('用户信息');
})
```

Section 02

【使用中间件】

▸▸▸ 应用程序级中间件

➢ 示例：一个中间件子堆栈

```
app.use('/user/:id', function (req, res, next) {
  console.log('请求URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('请求类型:', req.method);
  next();
});
```

Section 02

【使用中间件】

▸▸▸ 路由器级中间件

- 路由器级中间件被绑定到路由器实例上。
- 可使用router.use()和router.METHOD()函数挂载路由器级中间件。
- 示例：通过使用路由器级中间件改写上述用于应用程序级中间件的中间件系统

```
var app = express();
var router = express.Router();
//没有装挂载路径的中间件函数，对路由器的任何请求都会执行
router.use(function (req, res, next) {
  console.log('时间:', Date.now());
  next();
});
//中间件子堆栈显示对/user/:id路径的任何类型的HTTP请求的信息
router.use('/user/:id', function (req, res, next) {
  console.log('请求URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('请求类型:', req.method);
  next();
});
//将路由器在挂载到应用程序上
app.use('/', router);
```

Section 02

【使用中间件】

▸▸▸ 错误处理中间件

- 错误处理中间件总是需要4个参数。
- 示例

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

▸▸▸ 内置中间件

- Express的内置中间件
  - express.static：提供静态资源（如HTML文件、图像等）服务。
  - express.json：解析带有JSON有效负载的传入请求。
  - express.urlencoded：解析带有使用URL编码的有效负载的传入请求。
- express.static中间件的用法

```
express.static(root, [options])
```

Section 02

【使用中间件】

▸▸▸ 第三方中间件

- 第三方中间件需要安装相应的Node.js包，然后在应用程序级或路由器级将其挂载到应用程序。
- 示例：Cookie解析中间件函数cookie-parser

（1）安装相应包：

```
npm install cookie-parser
```

（2）在程序中挂载它：

```
var express = require('express');
var app = express();
var cookieParser = require('cookie-parser');
app.use(cookieParser());// 挂载cookie-parsing中间件
```

Section 03

【编写自己的中间件】

▸▸▸ 示例一：日志中间件函数myLogger

```
const express = require('express');
const app = express();
// 中间件函数被指派给名为myLogger的变量,其功能是在对应用程序的请求通过时输出“已记录”信息
var myLogger = (req, res, next)=> {
    console.log('已记录');
    next();
}
app.use(myLogger);//挂载该中间件函数
app.get('/', (req, res) => res.send('Hello World!'));
app.listen(3000);
```

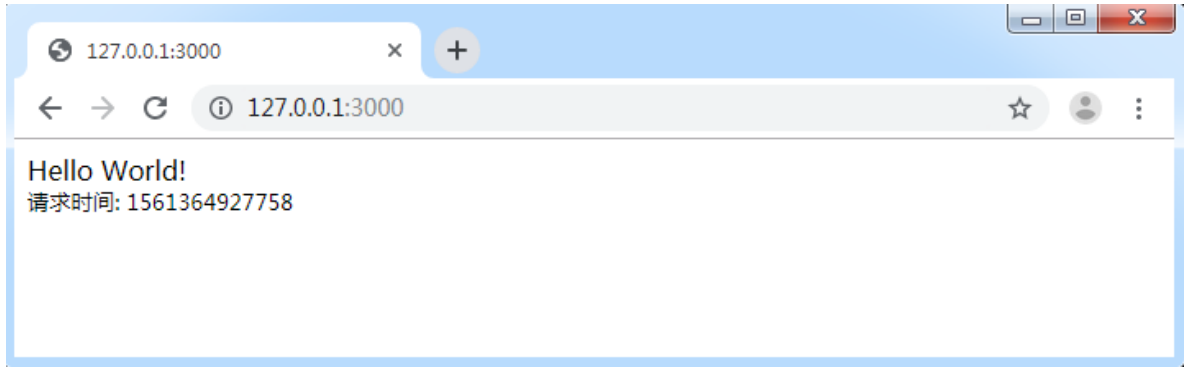
Section 03

【编写自己的中间件】

▸▸▸ 示例二：请求时间中间件函数requestTime

```
const express = require('express');
const app = express();
// 中间件函数用于获取请求时间
var requestTime = function (req, res, next) {
    req.requestTime = Date.now();
    next();
}
app.use(requestTime);//挂载该中间件函数
app.get('/', function (req, res) {
    var responseText = 'Hello World!<br>';
    responseText += '<small>请求时间: ' +
req.requestTime + '</small>';
    res.send(responseText);
});
app.listen(3000);
```

在浏览器中显示请求的时间戳



Section 03

【编写自己的中间件】

▸▸▸ 可配置的中间件

- 导出一个接受options对象或其他参数的函数，该函数根据输入参数返回中间件的实现。
- 示例：my-middleware.js文件

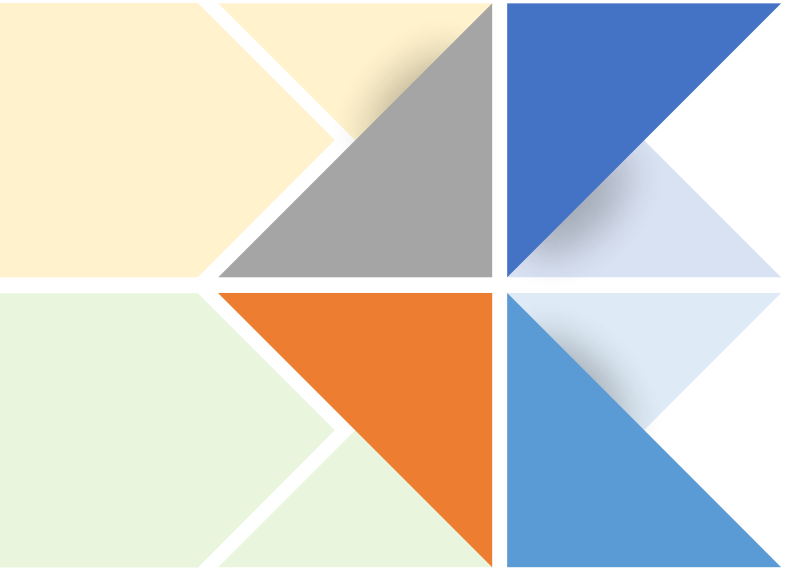
```
module.exports = function(options) {
  return function(req, res, next) {
    // 根据options对象实现中间件功能
    next()
  }
}
```

- 使用该中间件

```
var mw = require('./my-middleware.js');
app.use(mw({ option1: '1', option2: '2' }));
```

内容导航

Contents



Section 00

【学习目标】

（1）了解Express的模板引擎；
（2）熟悉EJS语法，编写EJS模板；
（3）掌握EJS模板引擎的使用方法。

Express支持通过模板引擎在应用程序中使用静态模板文件。在程序运行时，模板引擎使用实际的值替换模板文件中的变量，并将模板转换为要发送给客户端的HTML文件。这种方法使得设计HTML页面变得更加容易，轻松实现数据与视图分离，有利于大型项目中的前后端开发的分工协作。

Section 01

【模板引擎概述】

Express推荐的模板引擎

- Pug
- Mustache
- EJS

渲染模板文件

- 在Express项目脚手架中的app.js文件中配置应用程序设置属性。
  - views：模板文件所在的目录，默认为应用程序根目录中的views目录。
  - view engine：要使用的模板引擎类型。
- 安装相应模板引擎的npm包。例如：

```
npm install pug --save
```

Section 01

【模板引擎概述】

▶▶▶ 模板文件应用的完整步骤示范

(1) 设置要使用的Pug模板引擎。

```
app.set('view engine', 'pug')
```

(2) 在views目录中创建一个名为index.pug的Pug模板文件：

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

(3) 创建一个路由来渲染index.pug文件。

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

(4) 当向主页发出请求时，该index.pug文件将呈现为HTML样式。

Section 02

【EJS语法】

▸▸▸ EJS的基本语法

语法格式	用途
.ejs	模板文件的扩展名
<% 代码 %>	代码块
<%= 表达式 %>	使用传统的方式在模板中输出表达式的值，特殊字符将被转义
<%- 表达式 %>	在模板中直接输出表达式的值，不进行转义
<% 表达式 -%>	删除新的空白行
<%_ 表达式 _%>	清除开始标签前面的和结束标签后面的空格符
<%% 表达式 %>	将HTML代码解析为字符串，但是会在输出的字符串前后加上<% %>
<%# 注释内容 %>	注释内容

Section 02

【EJS语法】

▸▸▸ 内置的条件判断和循环语法

➢ 条件判断的示例：

```
<% if(isLogin){ %>
    <p><a href="#">Bob</a> | <a href="#">退出</a></p>
<% }else{ %>
    <p><a href="#">登录</a> | <a href="#">注册</a></p>
<% } %>
```

➢ 循环语句的示例：

```
<ul>
    <% for(var i in users){ %>
    <li><%= users[i].username %>--<%= users[i].age %></li>
  <% } %>
</ul>
```

Section 02

【EJS语法】

▸▸▸ 与Express过滤器结合使用

➢ 对EJS变量进行过滤的方法：

```
<%=: 表达式 | 过滤器 %>
```

▸▸▸ 模板嵌套

```
<html lang="en">
  <head>
    <% include ./head %>
  </head>
  <body>
    <main>
      <% include ./header %>
      <div>main content</div>
      <% include ./footer %>
    </main>
  </body>
</html>
```

Section 03

【EJS模板引擎的使用】

(1) 设置模板引擎类型。

```
app.set("view engine","ejs")
```

(2) 配置EJS模板文件存放路径。

```
app.set("view",path)
```

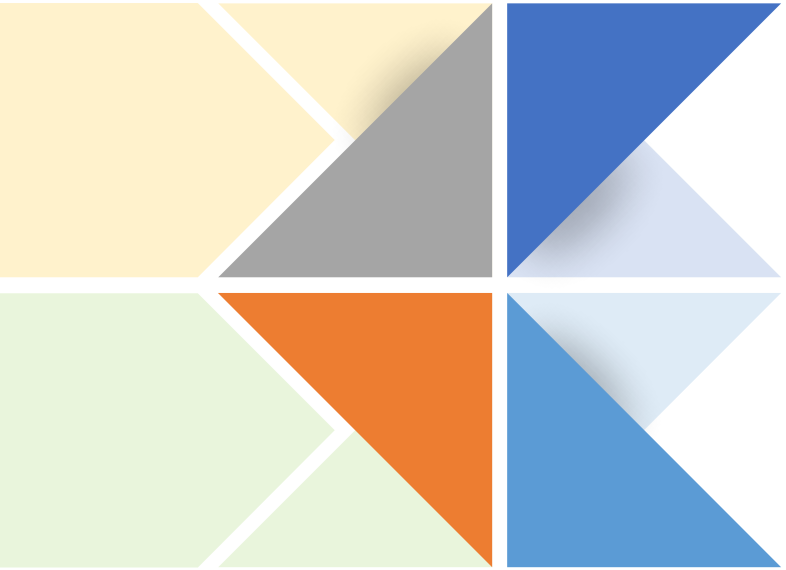
(3) 准备EJS模板文件。

(4) 将EJS模板渲染成HTML页面后返回给浏览器。

```
res.render(path,data)
```

内容导航

Contents



Section 00

【学习目标】

（1）了解Express与数据库的集成方法；
（2）掌握SQL Server的连接方法。

Section 01

【Node.js支持的数据库】

- Express应用程序中使用数据库实现持久化存储，需为数据库系统加载适当的Node.js驱动程序。
- Node.js支持的数据库
  - Cassandra
  - Couchbase
  - CouchDB
  - LevelDB
  - MySQL
  - MongoDB
  - Neo4j
  - Oracle
  - PostgreSQL
  - Redis
  - SQL Server
  - SQLite
  - ElasticSearch

Section 02

【SQL Server的连接方法】

- SQL Server所用的Node.js驱动程序由tedious包提供。

```
npm install tedious
```

- 访问SQL Server的示例代码

```
var Connection = require('tedious').Connection;
var Request = require('tedious').Request;
var config = {
  userName: 'your_username', // 用户账号
  password: 'your_password', // 密码
  server: 'localhost'          // 服务器
}
var connection = new Connection(config);
//建立连接并执行语句
connection.on('connect', function(err) {
  if (err) {
    console.log(err);
  } else {
    executeStatement();
  }
});
```

Section 02

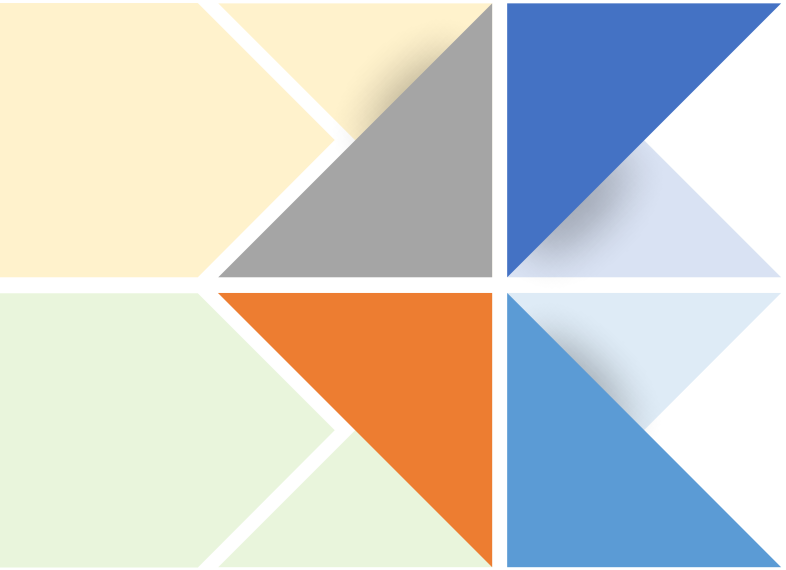
【SQL Server的连接方法】

- 访问SQL Server的示例代码

```
//执行语句的函数
function executeStatement() {
  request = new Request("select 123, 'hello world'", function(err, rowCount) {
    if (err) {
      console.log(err);
    } else {
      console.log(rowCount + ' rows');
    }
    connection.close();
  });
  request.on('row', function(columns) {
    columns.forEach(function(column) {
      if (column.value === null) {
        console.log('NULL');
      } else {
        console.log(column.value);
      }
    });
  });
  connection.execSql(request);
}
```

内容导航

Contents



Section 00

【学习目标】

（1）了解Express的错误处理；

（2）基本掌握错误处理程序的编写。

错误处理是指Express如何捕获和处理同步和异步发生的错误。Express内置默认的错误处理程序，开发人员刚开始使用Express时不必编写自己的错误处理程序。

Section 01

【捕获错误】

- 同步代码抛出的错误Express会捕获和处理它，例如：

```
app.get("/", function (req, res) {
  throw new Error("出问题了"); // Express自身会捕获此错误
});
```

- 异步代码返回的错误必须传递给next()函数，Express会捕获和处理，例如：

```
app.get("/", function (req, res, next) {
  fs.readFile("/file-does-not-exist", function (err, data) {
    if (err) {
      next(err); // 将错误传递给Express
    }
    else {
      res.send(data);
    }
  });
});
```

Section 02

【默认错误处理程序】

- 默认的错误处理中间件函数被添加到中间件函数堆栈的末尾。
- 若将一个错误传递给next()函数，并且没有在自定义的错误处理程序中处理它，则该错误将由内置的错误处理程序来处理，并使用堆栈跟踪写入客户端。
- 在自定义错误处理程序中，当响应头已经被发送到客户端时，必须将错误委派到Express中的默认错误处理程序。

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

Section 03

【编写错误处理程序】

- 错误处理函数有4个参数（err、req、res、next）

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

- 在其他app.use()函数和路由调用之后（即在最后）定义错误处理中间件，例如：

```
var bodyParser = require('body-parser');
var methodOverride = require('method-override');
app.use(bodyParser.urlencoded({
  extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());
app.use(function (err, req, res, next) {
  // 错误处理
});
```

内容导航

Contents

- 了解REST API
- 实现思路与技术准备
- 创建项目并编写程序
- 测试

Section 00

【学习目标】

(1) 了解REST API;
(2) 掌握REST API项目实施流程;
(3) 掌握Express应用程序的编写。

REST是目前最流行的API设计规范，主要用于Web数据接口的设计。REST API是目前异构系统之间互联与集成的主要手段。使用Express的路由和中间件很容易构建REST API程序。这里示范构建一个简单的图书信息管理项目。

Section 01

【了解REST API】

什么是REST API

- REST即Representational State Transfer的缩写，通常译为表现层状态转化。
- 面向资源是REST最明显的特征，对于同一个资源的一组不同的操作，REST要求必须通过统一的接口对资源执行各种操作。
- REST是所有Web应用都应该遵守的架构设计指导原则。

REST请求

- POST——Create（创建）
- GET——Read（读取）
- PUT——Update（更改）
- DELETE ——Delete（删除）

Section 01

【了解REST API】

HTTP状态码

- 1xx：成功接收到请求。
- 2xx：操作成功。
- 3xx：重定向。
- 4xx：客户端错误。
- 5xx：服务器错误。

响应的数据部分

- REST API返回的数据通常是一个JSON对象。
- Node.js、Express和JavaScript对JSON提供很好的支持。
- JSON是使用最广泛的结构化数据标准。

Section 02

【演练目标】

HTTP方法	URI	操作（Action）
GET	/books	列出所有图书的信息
GET	/books/ID	通过ID获取指定图书的信息
POST	/books	添加图书的信息
PUT	/books/ID	通过ID修改指定图书的信息
DELETE	/books/ID	通过ID删除指定图书的信息

Section 03

【实现思路与技术准备】

▸▸▸ 基本思路

- 将MySQL数据库的表操作封装为一个数据操作接口供应用程序调用。
- 通过Express路由、中间件和内置函数解析网络请求，并将JSON数据和HTTP状态码封装到响应对象，然后返回给客户端。

▸▸▸ 解析提交的JSON数据

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

Section 03

【实现思路与技术准备】

▸▸▸ 将异步操作结果作为函数返回值

- 在数据操作接口程序中定义一个含有回调函数参数的接口函数getall(function(cb))：

```
exports.getall =  function(callback){
  query('SELECT * FROM 'bookinfo'',function (err, results, fields) {
    callback(results);// 通过回调函数callback返回接口函数操作结果
  });
}
```

- 在另一个程序中调用该函数，通过回调函数参数获得数据操作结果。

```
app.get('/api/books', (req, res) => {
    dbquery.getall(function(cb){
        res.json(cb).end();//通过回调函数参数获得接口函数操作结果
    });
});
```

▸▸▸ 使用REST API测试工具

- 推荐使用专门工具Wisdom RESTClient

Section 04 【创建项目】

- 创建一个名为restapi的目录，再在其中创建一个package.json文件。

```
C:\nodeapp\ch08>mkdir restapi && cd restapi
C:\nodeapp\ch08\restapi>npm init
```

- 执行以下命令安装mysql2和express包：

```
C:\nodeapp\ch08\restapi>cnpm install mysql2 -–save
C:\nodeapp\ch08\restapi>cnpm install express --save
```

Section 05

【编写数据库接口程序】

▶▶▶ 数据库连接程序

```
//定义连接池
const mysql = require('mysql2');
const pool = mysql.createPool({
  host: 'localhost',
  user: 'root',
  password: 'abc123',
  database: 'testmydb',
  dateStrings: true,
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});
```

```
//定义通用的查询接口并将其导出
var query=function(sql,values,callback){
    pool.getConnection(function(err,conn){
        if(err){
            callback(err,null,null);
        }else{

conn.query(sql,values,function(qerr,results,fields){
                conn.release();//释放连接
                callback(qerr,results,fields); //事件驱动回调
            });
        }
    });
};
module.exports=query;    //导出模块
```

Section 05

【编写数据库接口程序】

▶▶▶ 数据操作接口

```
const query = require("./dbconn.js");
// 返回所有记录
exports.getall =  function(callback){
  query('SELECT * FROM 'bookinfo'',function
(err, results, fields) {
    if(err){
        console.log('数据查询失败');
        throw err;
    }
    callback(results);
  });
}
```

```
// 返回指定记录
exports.get = function(bookid,callback){
  var arr = [];
  arr.push(bookid);
  query('SELECT * FROM 'bookinfo' WHERE 'id'
= ?', arr,function(err, results, fields){
        if(err){
            console.log('数据查询失败');
            throw err;
        }
        callback(results);
    });
}
```

Section 06

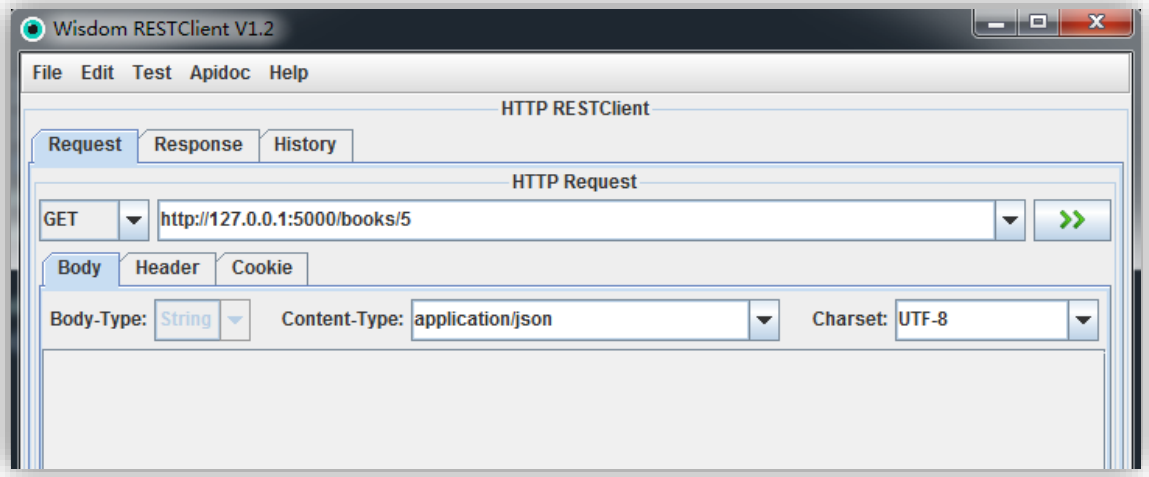
【编写主入口文件】

```
const express = require('express');
const bookquery = require('./bookquery.js');
const app = express();
app.use(express.json());
app.use(express.urlencoded());
//获取所有图书信息
app.get('/', function(req, res){
    res.redirect('/books'); // 重定向
});
app.get('/books', (req, res) => {
    bookquery.getall(function(rec){
        res.json(rec).end();
    });
});
//获取指定id的图书信息，这里用到路由参数id
app.get('/books/:id', function(req, res){
    var bookid = parseInt(req.params.id);
    bookquery.get(bookid,function(rec){
        res.json(rec).end();
    });
});
//使用POST方法添加图书信息
app.post('/books', function(req, res){
    var reqbody = req.body;
    var arr = [];
    //遍历JSON数据中所有的值
    for(var key in reqbody){
        arr.push(reqbody[key]);
    }
    bookquery.add(arr,function(rec){
        res.json(rec).end();
    });
});
(此处省略)
//主程序监听端口
const port = process.env.PORT || 5000;
app.listen(port, function() {console.log('监听端口：${port}')});
```

Section 07

【运行程序进行测试】

查询操作请求



查询操作结果

Section 07

【运行程序进行测试】

添加操作请求

添加操作结果

Section 08

【控制API版本】

- 在Express中可以使用路由器来实现API版本管理，将每个版本作为一个模块。
- API版本示例

```
const express = require('express');
const bookquery = require('./bookquery.js');
//const app = express();
var router = express.Router();//创建路由器实例
//以下代码中的app都改为router
router.use(express.json());
router.use(express.urlencoded({ extended: true }));
//获取所有图书信息
router.get('/', function(req, res){
    res.redirect('/books'); // 重定向
});
  (此处省略)
//路由器实例不能独立运行
/* const port = process.env.PORT || 5000;
app.listen(port, function() {console.log('监听端口：${port}')}); */
module.exports = router;  //导出路由器实例
```

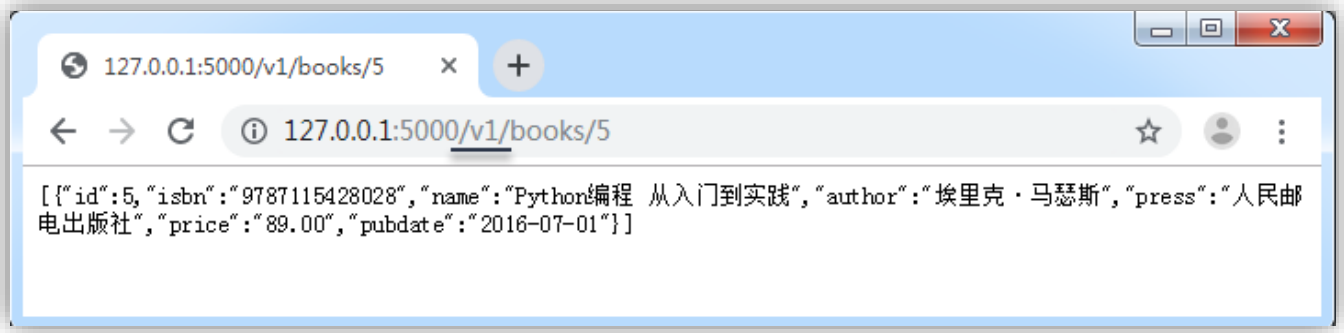
Section 08

【控制API版本】

➢ 新的入口文件示例

```
const express = require('express');
const apiv1 = require("./appv1.js");//导入版本1文件
const apiv2 = require("./appv2.js");//导入版本2文件
const app = express();
app.use("/", apiv2);// 将不带版本号的请求交给第2个路由器实例处理
app.use("/v1", apiv1);//对版本1的请求交给第1个路由器实例处理
app.use("/v2", apiv2);//对版本2的请求交给第2个路由器实例处理
//主程序监听端口
const port = process.env.PORT || 5000;
app.listen(port, function() {console.log('监听端口：${port}')});
```

➢ 带版本号调用API



本章小结

Summary

使用Node.js开发Web应用程序时，一般要使用框架。Node.js提供不同类型的框架，如MVC框架、全栈框架、REST API，这些框架支持众多的特性和功能，易于使用，能够缩短大型应用软件系统的开发周期，提高开发质量。本章重点讲解的是经典的Express框架，掌握该框架的基本使用方法之后，就可以触类旁通，学习其他框架也不是难事。

Thank you