Express

1. Express 入门

1.1 Express 应用

WEB应用

Express 是一个基于 Node.js 平台的极简、灵活的 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。

API

丰富的 HTTP 快捷方法和任意排列组合的 Connect 中间件，让你创建健壮、友好的 API 变得既快速又简单。

1.2 相关网站

- 官网： http://expressjs.com/
- Github: https://github.com/expressjs/express

1.3 安装

```
npm install express --save
```

1.4 创建http服务和简单路由

```
//导入 express 模块
const express = require('express');

//创建express实例
const app = express();

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

app.listen(3000, function () {
    console.log('http server is running on port 300');
});
```

1.4 托管静态文件(中间件的使用)

将静态资源文件所在的目录作为参数传递给 `express.static` 中间件就可以提供静态资源文件的访问了。例如，假设在 `public` 目录放置了图片、CSS 和 JavaScript 文件，你就可以：

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

如果你的静态资源存放在多个目录下面，你可以多次调用 `express.static` 中间件：

访问静态资源文件时， `express.static` 中间件会根据目录添加的顺序查找所需的文件。

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

如果你希望所有通过 `express.static` 访问的文件都存放在一个“虚拟（virtual）”目录（即目录根本不存在）下面，可以通过为静态资源目录指定一个挂载路径的方式来实现，如下所示：

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

2 路由

2.1 路由的使用

```
// 对网站首页的访问返回 "Hello World!" 字样
app.get('/', function (req, res) {
  res.send('Hello World!');
});

// 网站首页接受 POST 请求
app.post('/', function (req, res) {
  res.send('Got a POST request');
});

// /user 节点接受 PUT 请求
app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user');
});

// /user 节点接受 DELETE 请求
app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user');
});
```

2.2 路由方法

Express 定义了如下和 HTTP 请求对应的路由方法

- app.get()
- app.post()
- app.put()
- app.head()
- app.delete()
- app.options()
- app.trace()
- app.connect()
- ……

`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。

2.3 路由路径

注意：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

```
// 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

```
// 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

```
// 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$/, function(req, res) {
  res.send('/.*fly$/');
});
```

路径中带有查询字符串

```
// /search?name=lili&age=100
app.get('/articles', function(req, res){
    res.send(req.query.name+ ' . '+req.query.age)
})
```

路径中带有参数

```
// /articles/12313
app.get('/articles/:id', function(req, res){
    res.send(req.params.id)
})
```

2.4 路由句柄(回调函数)

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!');
});
```

2.4 路由模块化

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。

```
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();
});
// 定义网站主页的路由
router.get('/', function(req, res) {
  res.send('Birds home page');
});
// 定义 about 页面的路由
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);
```

3 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（Middleware） 是一个函数，它可以接收参数 请求对象（req)），响应对象（ res），和 web 应用中处于请求-响应循环流程中的中间件，一般被命名为 `next` 的变量。

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。

3.1 应用级中间件

应用级中间件绑定到 app 对象 使用 `app.use()` 和 `app.METHOD()`， 其中， `METHOD` 是需要处理的 HTTP 请求的方法，例如 GET, PUT, POST 等等，全部小写。

```
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();
});

// 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();
});

// 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');
});
```

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。 **注意**： `next('route')` 只对使用 `app.METHOD()` 或 `router.METHOD()` 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');
  // 否则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');
});
```

3.2 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()`。

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

```
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);
```

3.3 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，其写法如下：`(err, req, res, next)`。

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

3.4 内置中间件

- express.static 静态资源服务，如 html文件、图片 等
- express.json parses incoming requests with JSON payloads. **NOTE: Available with Express 4.16.0+**
- express.urlencoded parses incoming requests with URL-encoded payloads. **NOTE: Available with Express 4.16.0+**

3.5 第三方中间件

常用的第三方中间件 列表 http://expressjs.com/en/resources/middleware.html

- body-parser 解析HTTP 请求体
- compression 压缩HTTP响应
- connect-rid 生成唯一请求ID
- cookie-parser 解析cookie header 并 填充req.cookies
- cookie-session 创建基于cookie的session
- cors 启用跨来源资源共享（CORS）通过各种选项。
- csurf CSRF保护
- errorhandler 错误处理与调试
- method-override 覆盖HTTP请求方法
- morgan HTTP请求
- multer 处理 multi-part form data
- response-time 记录HTTP响应时间
- serve-favicon 提供网站图标(favicon)

- serve-index 显示目录列表
- serve-static 静态文件服务
- session 建立基于服务器的会话（仅开发）
- timeout 设置HTTP请求处理的超时周期。
- vhost 创建虚拟主机

5 请求和响应

5.1 请求对象

常用属性

- req.baseUrl 上级路由的基础URL
- req.originUrl 完整的URL
- req.protocol
- req.hostname
- req.path
- req.body
- req.ip
- req.params
- req.query
- req.xhr

常用方法

- req.get(field)

5.2 响应对象

常用属性

- res.headersSent

常用方法

- res.append(field [, value]) 添加响应头

- res.set(field [, value]) 设置响应头
- res.status(code) 设置响应状态吗

  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。
- res.type(type) 设置Content-type

  `res.download(path [, filename][, fn])` 提示下载文件。

  res.end(data) 终结响应处理流程。

  res.json() 发送一个 JSON 格式的响应。

  res.jsonp() 发送一个支持 JSONP 的 JSON 格式的响应。

  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。

6 模板引擎

6.1 模板引擎设置

- `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')`

6.2 渲染

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

6.3 Express模板引擎原理

通过 `app.engine(ext, callback)` 方法即可创建一个你自己的模板引擎。其中，`ext` 指的是文件扩展名、`callback` 是模板引擎的主函数，接受文件路径、参数对象和回调函数作为其参数。

```
var fs = require('fs'); // 此模板引擎依赖 fs 模块
app.engine('ntl', function (filePath, options, callback) { // 定义模板引擎
  fs.readFile(filePath, function (err, content) {
    if (err) return callback(new Error(err));
    // 这是一个功能极其简单的模板引擎
    var rendered = content.toString().replace('#title#', '<title>'+
options.title +'</title>')
    .replace('#message#', '<h1>'+ options.message +'</h1>');
    return callback(null, rendered);
  })
});
app.set('views', './views'); // 指定视图所在的位置
app.set('view engine', 'ntl'); // 注册模板引擎


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

7 错误处理

7.1 错误处理中间件的定义和使用

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，其格式如下 (err, req, res, next)

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

一般情况下，在其他 app.use() 和路由调用后，最后定义错误处理中间件

```
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。

7.2 多个错误处理中间件

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

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

7.3 默认错误处理

Express 内置了一个错误处理句柄，它可以捕获应用中可能出现的任意错误。这个缺省的错误处理中间件将被添加到中间件堆栈的底部。

如果你向 next() 传递了一个 error ，而你并没有在错误处理句柄中处理这个 error，Express 内置的缺省错误处理句柄就是最后兜底的。最后错误将被连同堆栈追踪信息一同反馈到客户端。堆栈追踪信息并不会在生产环境中反馈到客户端。

7.4 404页面定制

```
app.use((req, res, next) => {
    console.log('执行否');
    next(new Error('404'));
});

app.use((err, req, res, next) => {
    if (err.message == 404) {
        res.render('404');
    } else {
        next(err);
    }
})
```

8 项目生成器和调试

8.1 项目生成器

安装

```
npm install express-genertor -g
```

使用生成器生成项目骨架

```
express 项目名称
```

目录解构

> . ├── app.js #入口文件 ├── bin #命令目录 │ └── www ├── package.json ├── public #静态资源目录 │ ├── images │ ├── javascripts │ └── stylesheets │ └── style.css ├── routes #路由目录 │ ├── index.js │ └── users.js └── views # 模板目录 ├── error.jade ├── index.jade └── layout.jade

8.2 调试

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。

express 生成器的项目

```
set DEBUG=myapp & npm start # windows平台
DEBUG=myapp & npm start     # linux max 平台
```