

# Express框架

王红元  
coderwhy



实力IT教育

- 前面我们已经学习了使用http内置模块来搭建Web服务器，为什么还要使用框架？
  - 原生http在进行很多处理时，会较为复杂；
  - 有URL判断、Method判断、参数处理、逻辑代码处理等，都需要我们自己来处理 and 封装；
  - 并且所有的内容都放在一起，会非常的混乱；
- 目前在Node中比较流行的Web服务器框架是express、koa；
  - 我们先来学习express，后面再学习koa，并且对他们进行对比；
- express早于koa出现，并且在Node社区中迅速流行起来：
  - 我们可以基于express快速、方便的开发自己的Web服务器；
  - 并且可以通过一些实用工具和中间件来扩展自己功能；
- **Express整个框架的核心就是中间件，理解了中间件其他一切都非常简单！**



# Express安装

■ express的使用过程有两种方式：

- 方式一：通过express提供的脚手架，直接创建一个应用的骨架；
- 方式二：从零搭建自己的express应用结构；

## ■ 方式一：安装express-generator

安装脚手架

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

创建项目

```
express express-demo
```

安装依赖

```
npm install
```

启动项目

```
node bin/www
```

## ■ 方式二：从零搭建自己的express应用结构；

```
npm init -y
```

# Express的基本使用

## ■ 我们来创建第一个express项目：

- 我们会发现，之后的开发过程中，可以方便的将请求进行分离：
- 无论是不同的URL，还是get、post等请求方式；
- 这样的方式非常方便我们已经进行维护、扩展；
- 当然，这只是初体验，接下来我们来探索更多的用法；

## ■ 请求的路径中如果有一些参数，可以这样表达：

- /users/:userId；
- 在request对象中通过 req.params.userId；

## ■ 返回数据，我们可以方便的使用json：

- res.json(数据)方式；
- 可以支持其他方式，可以自行查看文档；
- <https://www.expressjs.com.cn/guide/routing.html>

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

// 创建服务器
const app = express();

// /home的get请求处理
app.get("/home", (req, res) => {
  res.end("Hello Home");
});

// /login的post请求处理
app.post("/login", (req, res) => {
  res.end("Hello Login");
});

// 开启监听
app.listen(8000, () => {
  console.log("服务器启动成功~");
})
```

■ Express是一个路由和中间件的Web框架，它本身的功能非常少：

□ Express应用程序本质上是一系列中间件函数的调用；

■ 中间件是什么呢？

□ 中间件的本质是传递给express的一个回调函数；

□ 这个回调函数接受三个参数：

✓ 请求对象（request对象）；

✓ 响应对象（response对象）；

✓ next函数（在express中定义的用于执行下一个中间件的函数）；

## ■ 中间件中可以执行哪些任务呢？

- 执行任何代码；
- 更改请求（request）和响应（response）对象；
- 结束请求-响应周期（返回数据）；
- 调用栈中的下一个中间件；

## ■ 如果当前中间件功能没有结束请求-响应周期，则必须调用next()将控制权传递给下一个中间件功能，否则，请求将被挂起。

```
var express = require('express');  
var app = express();  
  
app.get('/', function(req, res, next) {  
  next();  
})  
  
app.listen(3000);
```

中间件功能适用的HTTP方法。

中间件功能适用的路径（路由）。

中间件功能。

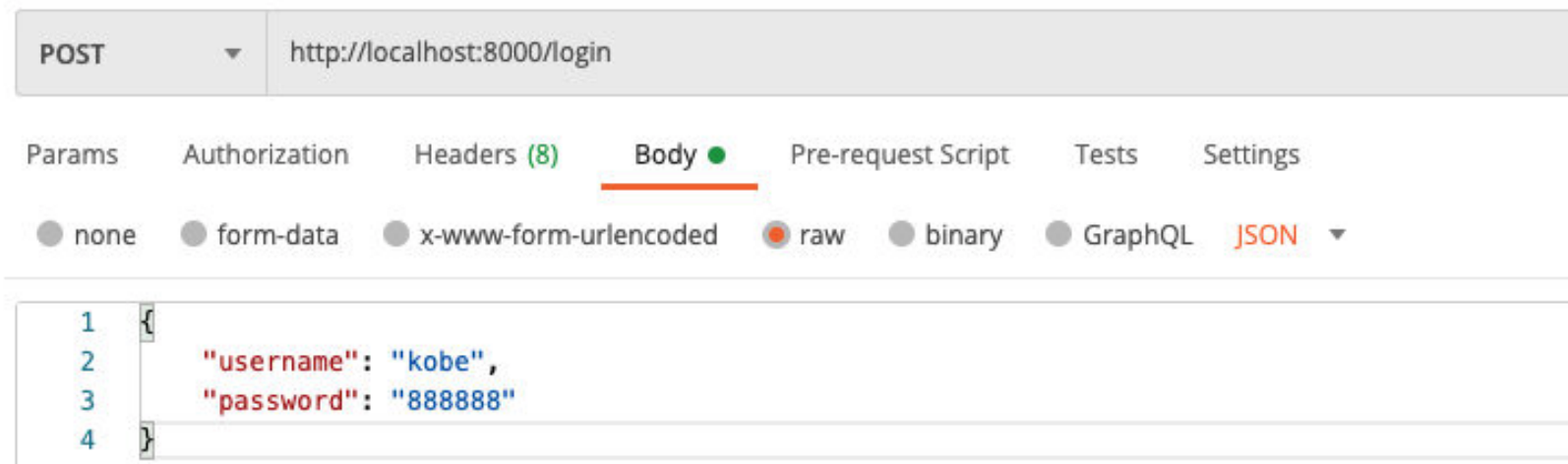
中间件函数的回调参数，按照惯例称为“next”。

中间件函数的HTTP响应参数，按照惯例称为“res”。

中间件功能的HTTP请求参数，按照惯例称为“req”。

- 那么，如何将一个中间件应用到我们的应用程序中呢？
  - express主要提供了两种方式：app/router.use和app/router.methods；
  - 可以是 app，也可以是router，router我们后续再学习：
  - methods指的是常用的请求方式，比如： app.get或app.post等；
- 我们先来学习use的用法，因为methods的方式本质是use的特殊情况；
  - **案例一：最普通的中间件**
  - **案例二：path匹配中间件**
  - **案例三：path和method匹配中间件**
  - **案例四：注册多个中间件**

- 并非所有的中间件都需要我们从零去编写：
  - express有内置一些帮助我们完成对request解析的中间件；
  - registry仓库中也有很多可以辅助我们开发的中间件；
- 在客户端发送post请求时，会将数据放到body中：
  - 客户端可以通过json的方式传递；
  - 也可以通过form表单的方式传递；







# 编写解析request body中间件

```
app.use((req, res, next) => {  
  if (req.headers['content-type'] === 'application/json') {  
    req.on('data', (data) => {  
      const userInfo = JSON.parse(data.toString());  
      req.body = userInfo;  
    })  
    req.on('end', () => {  
      next();  
    })  
  } else {  
    next();  
  }  
})  
  
app.post('/login', (req, res, next) => {  
  console.log(req.body);  
  res.end("登录成功~");  
});
```

- 但是，事实上我们可以使用express内置的中间件或者使用body-parser来完成：

```
app.use(express.json());

app.post('/login', (req, res, next) => {
  console.log(req.body);
  res.end("登录成功~");
});
```

- 如果我们解析的是 application/x-www-form-urlencoded：

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

app.post('/login', (req, res, next) => {
  console.log(req.body);
  res.end("登录成功~");
});
```

# 应用中间件 – 第三方中间件

- 如果我们希望将请求日志记录下来，那么可以使用express官网开发的第三方库：morgan

□ 注意：需要单独安装

```
const loggerWriter = fs.createWriteStream('./log/access.log', {
  flags: 'a+'
})
app.use(morgan('combined', {stream: loggerWriter}));
```

- 上传文件，我们可以使用express提供的multer来完成：

```
const upload = multer({
  dest: "uploads/"
})

app.post('/upload', upload.single('file'), (req, res, next) => {
  console.log(req.file.buffer);
  res.end("文件上传成功~");
})
```

# 上传文件中间件 - 添加后缀名

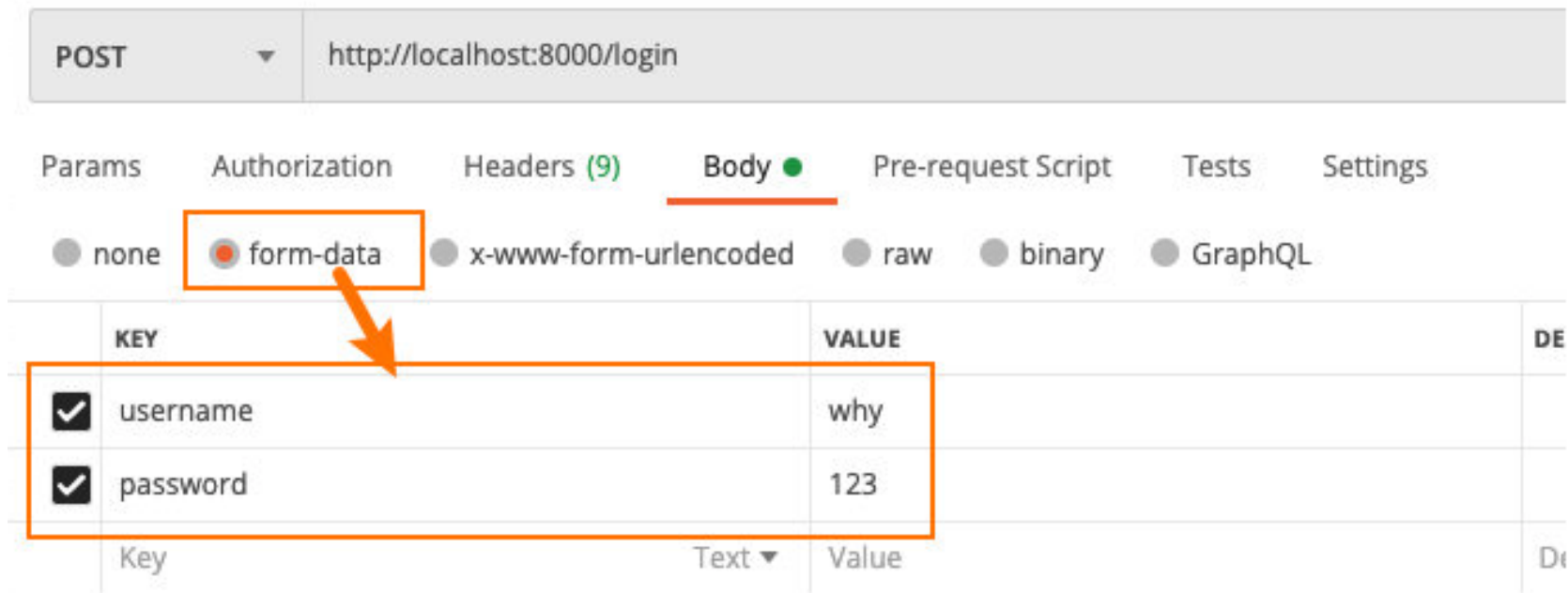
- 上传文件，我们可以使用express提供的multer来完成：

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, "uploads/")
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + path.extname(file.originalname));
  }
})

const upload = multer({
  storage
})

app.post('/upload', upload.single('file'), (req, res, next) => {
  console.log(req.file.buffer);
  res.end("文件上传成功~");
})
```

- 如果我们希望借助于multer帮助我们解析一些form-data中的普通数据，那么我们可以使用any：



KEY	VALUE	DE
<input checked="" type="checkbox"/> username	why	
<input checked="" type="checkbox"/> password	123	

Key	Text ▼	Value	De
-----	--------	-------	----

```
app.use(upload.any());

app.use('/login', (req, res, next) => {
  console.log(req.body);
});
```

# 客户端发送请求的方式

■ 客户端传递到服务器参数的方法常见的是5种：

- 方式一：通过get请求中的URL的params；
- 方式二：通过get请求中的URL的query；
- 方式三：通过post请求中的body的json格式（中间件中已经使用过）；
- 方式四：通过post请求中的body的x-www-form-urlencoded格式（中间件使用过）；
- 方式五：通过post请求中的form-data格式（中间件中使用过）；

■ 目前我们主要有两种方式没有讲，下面我进行一个演练。

# 传递参数params和query

■ 请求地址：<http://localhost:8000/login/abc/why>

■ 获取参数：

```
app.use('/login/:id/:name', (req, res, next) => {  
  console.log(req.params);  
  res.json("请求成功~");  
})
```

■ 请求地址：<http://localhost:8000/login?username=why&password=123>

■ 获取参数：

```
app.use('/login', (req, res, next) => {  
  console.log(req.query);  
  res.json("请求成功~");  
})
```

## ■ end方法

- 类似于http中的response.end方法，用法是一致的

## ■ json方法

- json方法中可以传入很多的类型：object、array、string、boolean、number、null等，它们会被转换成json格式返回；

## ■ status方法

- 用于设置状态码：

- 更多响应的方式：<https://www.expressjs.com.cn/4x/api.html#res>



■ 如果我们将所有的代码逻辑都写在app中，那么app会变得越来越复杂：

- 一方面完整的Web服务器包含非常多的处理逻辑；
- 另一方面有些处理逻辑其实是一个整体，我们应该将它们放在一起：比如对users相关的处理
  - ✓ 获取用户列表；
  - ✓ 获取某一个用户信息；
  - ✓ 创建一个新的用户；
  - ✓ 删除一个用户；
  - ✓ 更新一个用户；

■ 我们可以使用 `express.Router`来创建一个路由处理程序：

- 一个Router实例拥有完整的中间件和路由系统；
- 因此，它也被称为 迷你应用程序（mini-app）；

```
// 用户相关的处理
const userRouter = express.Router();

userRouter.get('/', (req, res, next) => {
  res.end("用户列表");
});

userRouter.post('/', (req, res, next) => {
  res.end("创建用户");
});

userRouter.delete('/', (req, res, next) => {
  res.end("删除用户");
});

app.use('/users', userRouter);
```

■ 部署静态资源我们可以选择很多方式：

□ Node也可以作为静态资源服务器，并且express给我们提供了方便部署静态资源的方法；

```
const express = require('express');  
  
const app = express();  
  
app.use(express.static('./build'));  
  
app.listen(8000, () => {  
  console.log("静态服务器启动成功~");  
})
```

```
app.use((err, req, res, next) => {  
  const message = err.message;  
  
  switch (message) {  
    case "USER DOES NOT EXISTS":  
      res.status(400).json({message})  
    }  
  
  res.status(500)  
})
```

# 创建app的过程

- express函数的本质其实是createApplication：

```
function createApplication() {  
  var app = function(req, res, next) {  
    app.handle(req, res, next);  
  };  
  
  mixin(app, EventEmitter.prototype, false);  
  mixin(app, proto, false);  
  
  // expose the prototype that will get set on requests  
  app.request = Object.create(req, {  
    app: { configurable: true, enumerable: true, writable: true, value: app }  
  });  
  
  // expose the prototype that will get set on responses  
  app.response = Object.create(res, {  
    app: { configurable: true, enumerable: true, writable: true, value: app }  
  });  
  
  app.init();  
  return app;  
}
```

```
JS express.js JS application.js X  
lib > JS application.js > listen  
608 *  
609 *... http.createServer(app).listen(80);  
610 *... https.createServer({...}, app).listen(443);  
611 *  
612 * @return {http.Server}  
613 * @public  
614 */  
615  
616 app.listen = function listen() {  
617   var server = http.createServer(this);  
618   return server.listen.apply(server, arguments);  
619 };  
620  
621 /**  
622 * Log error using console.error.  
623 *  
624 * @param {Error} err  
625 * @private  
626 */  
627  
628 function logerror(err) {  
629   /* istanbul ignore next */  
630   if (this.get('env') !== 'test') console.error(err.stack || err.toString());  
631 }
```

this就是app对象

■ 比如我们通过use来注册一个中间件，源码中发生了什么？

- 我们会发现无论是app.use还是app.methods都会注册一个主路由；
- 我们会发现app本质上会将所有的函数，交给这个主路由去处理的；

```
application.js > use
87 app.use = function use(fn) {
88   var offset = 0;
89   var path = '/';
90
91   // default path to '/'
92   // disambiguate app.use([fn])
93   if (typeof fn !== 'function') { ...
94   }
95
96   var fns = flatten(slice.call(arguments, offset));
97
98   if (fns.length === 0) { ...
99   }
100
101   // setup router
102   this.lazyrouter();
103   var router = this._router;
104
105   fns.forEach(function (fn) {
106     // non-express app
107     if (!fn || !fn.handle || !fn.set) {
108       return router.use(path, fn);
109     }
110
111     debug('.use app under %s', path);
112     fn.mountpath = path;
113     fn.parent = this;
114
115     // restore .app property on req and res
116     router.use(path, function mounted_app(req, res, next) {
```

```
JS express.js JS application.js JS index.js
lib > router > JS index.js > use
432 // default path to '/'
433 // disambiguate router.use([fn])
434 if (typeof fn !== 'function') { ...
446 }
447
448 var callbacks = flatten(slice.call(arguments, offset));
449
450 if (callbacks.length === 0) { ...
452 }
453
454 for (var i = 0; i < callbacks.length; i++) {
455   var fn = callbacks[i];
456
457   if (typeof fn !== 'function') {
458     throw new TypeError('Router.use() requires a middleware funct
459   }
460
461   // add the middleware
462   debug('use %o %s', path, fn.name || '<anonymous>')
463
464   var layer = new Layer(path, {
465     sensitive: this.caseSensitive,
466     strict: false,
467     end: false
468   }, fn);
469
470   layer.route = undefined;
471
472   this.stack.push(layer);
473 }
```

■ 如果有一个请求过来，那么从哪里开始呢？

□ app函数被调用开始的；

```
function createApplication() {  
  var app = function(req, res, next) {  
    app.handle(req, res, next);  
  };  
  
  mixin(app, EventEmitter.prototype, false);  
  mixin(app, proto, false);  
  
  // expose the prototype that will get set on requests  
  app.request = Object.create(req, {  
    app: { configurable: true, enumerable: true, writable: true, value: app }  
  });  
  
  // expose the prototype that will get set on responses  
  app.response = Object.create(res, {  
    app: { configurable: true, enumerable: true, writable: true, value: app }  
  });  
  
  app.init();  
  return app;  
}
```

```
app.handle = function handle(req, res, callback) {  
  var router = this._router;  
  
  // final handler  
  var done = callback || finalhandler(req, res, {  
    env: this.get('env'),  
    onerror: logerror.bind(this)  
  });  
  
  // no routes  
  if (!router) {  
    debug('no routes defined on app');  
    done();  
    return;  
  }  
  
  router.handle(req, res, done);  
};
```

# router.handle中做的什么事情呢？

```
5 proto.handle = function handle(req, res, out) {
6   var self = this;
7
8   debug('dispatching %s %s', req.method, req.url);
9
10  var idx = 0;
11  var protohost = getProtohost(req.url) || '';
12  var removed = '';
13  var slashAdded = false;
14  var paramcalled = {};
15
16  // store options for OPTIONS request
17  // only used if OPTIONS request
18  var options = [];
19
20  // middleware and routes
21  var stack = self.stack;
22
23  // manage inter-router variables
24  var parentParams = req.params;
25  var parentUrl = req.baseUrl || '';
26  var done = restore(out, req, 'baseUrl', 'next', 'params');
27
28  // setup next layer
29  req.next = next;
30
31  // for options requests, respond with a default if nothing else responds
32  if (req.method === 'OPTIONS') {
33    done = wrap(done, function(old, err) {
34      if (err || !options.length) return old(err);
35      res.writeHead(200, {
36        'allow': options.join(', ')
37      });
38      old();
39    });
40  }
41
42  // dispatch
43  function dispatch(err) {
44    var layer = stack[idx++];
45    if (!layer) return done(err);
46    if (layer.method && !~layer.method.indexOf(req.method)) return dispatch(err);
47    debug('%s %s', layer.method, layer.name);
48    var rmw = layer.regexp.match(req.url);
49
50    if (rmw) {
51      var path = req.url.substr(0, rmw.index);
52      if (path === '/' && !slashAdded) {
53        req.url = path + '/';
54        slashAdded = true;
55      }
56      req.baseUrl = parentUrl + path;
57      req.params = merge(parentParams, rmw.params);
58      layer.handle(req, res, out, dispatch);
59    } else {
60      layerError = layerError || layer;
61      continue;
62    }
63  }
64
65  dispatch(err);
66}
```

取出stack

```
router > JS index.js > handle > next
while (match !== true && idx < stack.length) {
  layer = stack[idx++];
  match = matchLayer(layer, path);
  route = layer.route;

  if (typeof match !== 'boolean') {
    // hold on to layerError
    layerError = layerError || match;
  }

  if (match !== true) {
    continue;
  }

  if (!route) {
    // process non-route handlers normally
    continue;
  }

  if (layerError) {
    // routes do not match with a pending error
    match = false;
    continue;
  }

  var method = req.method;
  var has_method = route._handles_method(method);

  // build up automatic options response
  if (!has_method && method === 'OPTIONS') {
    options.push(method);
    continue;
  }
}
```

查看是否是匹配的