# MVC全栈框架—Express

# 意义

- 1. 了解内核设计思想，提升总体的掌控力度
- 2. 对于定制本土化业务模块时能更有把握
- 3. 实现针对Express内核的监控保障

# Express

- Express 是一个简洁而灵活的 node.js Web应用框架, 提供一系列强大特性帮助你创建各种Web应用。Express 不对 node.js 已有的特性进行二次抽象，我们只是在它之上扩展了Web应用所需的功能。丰富的HTTP工具以及来自Connect框架的中间件随取随用，创建强健、友好的API变得快速又简单

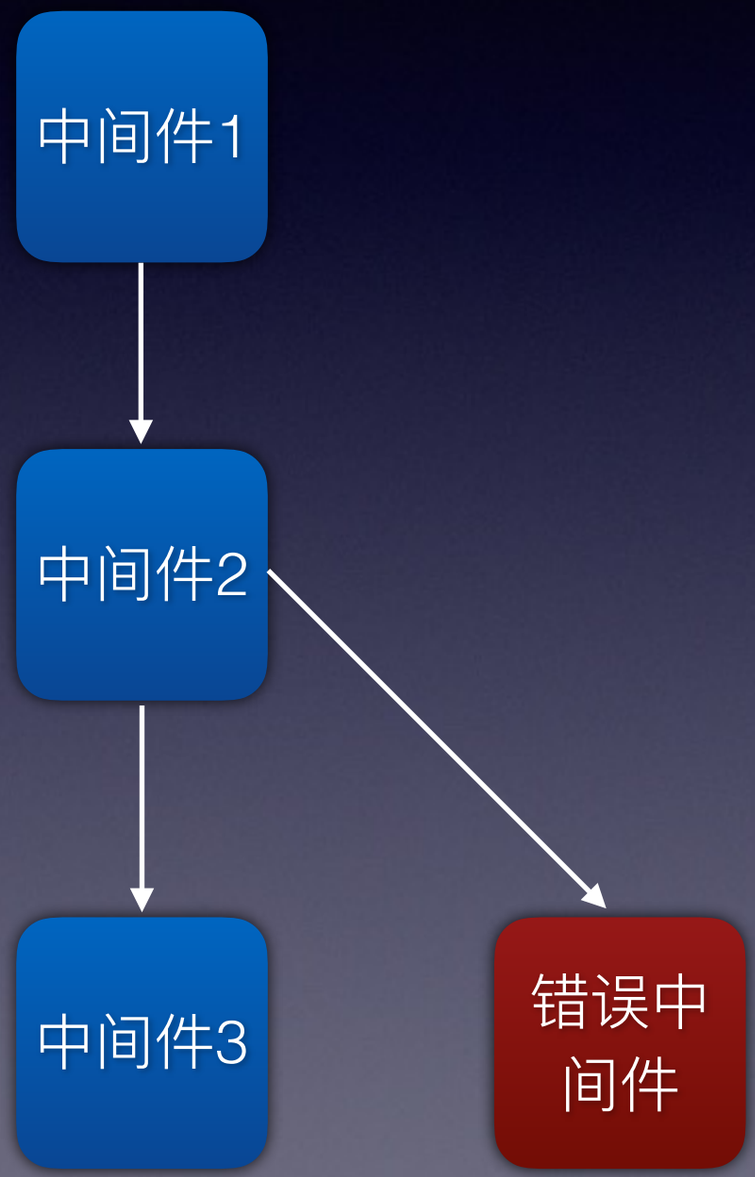
# Express

- 成熟案例众多(稳定性好)
- 社区活跃(查询问题)
- 源代码逻辑清晰(深入理解)
- 官方API文档全面(中文)

# 官方API文档

http://www.expressjs.com.cn/

# 原理设计



# 官方Hello World

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

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

var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;
    console.log('Example app listening at ' +
        'http://%s:%s', host, port);
});
```

引入Express：同样使用require引入express。express其实是一个生成app实例的函数，因此第二行执行这个函数生成了一个全局的app实例。

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

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

var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;
    console.log('Example app listening at ' +
        'http://%s:%s', host, port);
});
```

编写路由：利用第一步生成的app实例，调用app.get方法创建路由函数。第一个参数为客户端访问服务器路径，第二个参数为处理本次请求的路由处理函数。

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

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

var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;

    console.log('Example app listening at ' +
        'http://%s:%s', host, port);
});
```

处理本地请求的函数中三个重要的参数：req：基于HTTP模块的request的二次封装；res：基于HTTP模块的response的二次封装；next：进入下一个中间件的驱动方法

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

app.get('/index', function (req, res, next) {
    res.send(`Hello World!---${req.path}---${req.url}`);
});

var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;
    console.log('Example app listening at ' +
        'http://%s:%s', host, port);
});
```

侦听端口号：调用app.lisnten侦听本地的3000端口。这里linsten方法的第二个参数是一个回调函数，在Node底层侦听端口成功后才会调用。

# POST请求怎么办

- 将app.get替换为app.post即可，其余写法完全一致。
- app.all：所有客户端访问的http方法均命中。比如对于鉴权操作，不管是post请求还是get请求均需要鉴权，此时写成：app.all(‘/user/*’, authMethod)即可。

# 参数获取

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

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

app.get('/user/:userName', function (req, res, next) {
    let userName = req.params.userName;
    let location = req.query.location;

    res.send(`用户: ${userName} 在 ${location} 登录成功!`)
});

var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;
    console.log('Example app listening at ' +
        'http://%s:%s', host, port);
});
```

- rest风格的URL：参数在URL中，写法如左边所示，URL中对应的参数可以使用req.params.xxx的方式直接获取到。

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

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

app.get('/user/:userName', function (req, res, next) {
    let userName = req.params.userName;
    let location = req.query.location;
    res.send(`用户: ${userName} 在 ${location} 登录成功!`)
});

var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;
    console.log('Example app listening at ' +
        'http://%s:%s', host, port);
});
```

- querystring方式的URL：即携带在?后面的参数例如：/user/hyj?location=上海，就可以使用req.query.xxx的方式获取到。

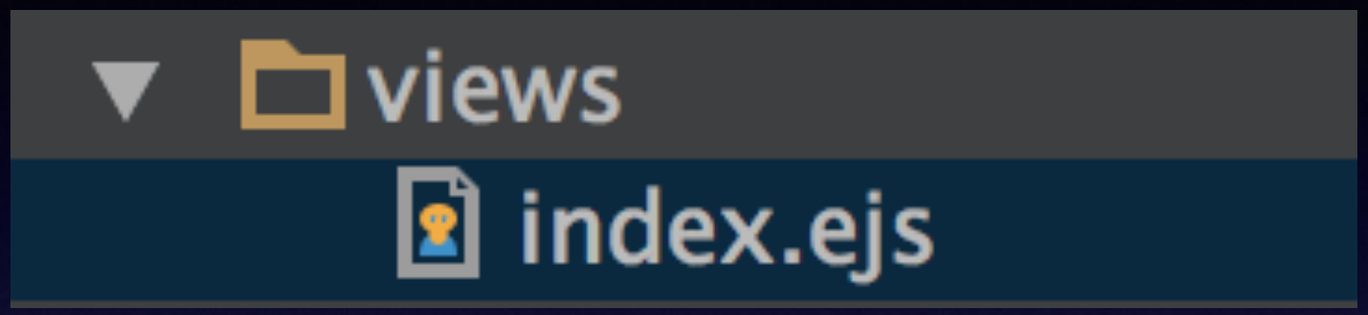
POST请求参数则使用

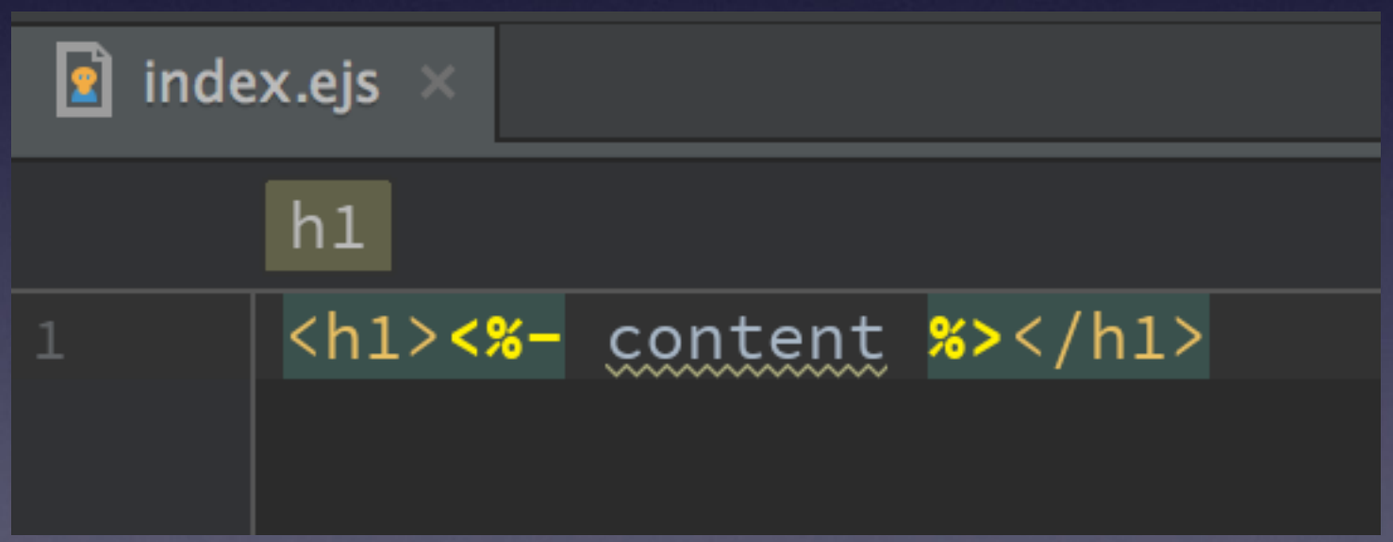
req.body.xxx的方式获取

# 渲染页面

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

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

app.get('/', function (req, res) {
    res.render('index', {content: 'I am ejs rendered!'});
});

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

app.get('/user/:userName', function (req, res, next) {
    let userName = req.params.userName;
    let location = req.query.location;

    res.send(`用户: ${userName} 在 ${location} 登录成功!`)
});

var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;
    console.log('Example app listening at ' +
        'http://%s:%s', host, port);
});
```

- 设置模板view文件路径：app.set('views', path)
- 设置模板解析引擎：app.set('view engine', 模板引擎名称)

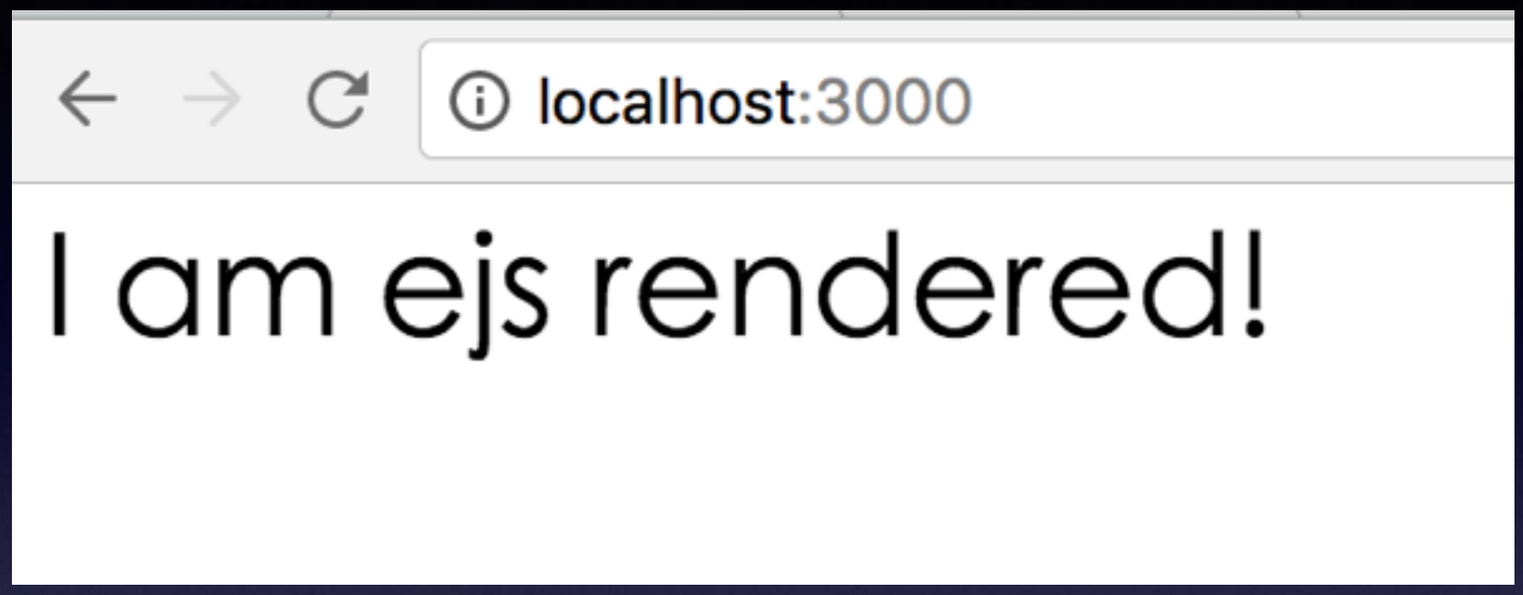


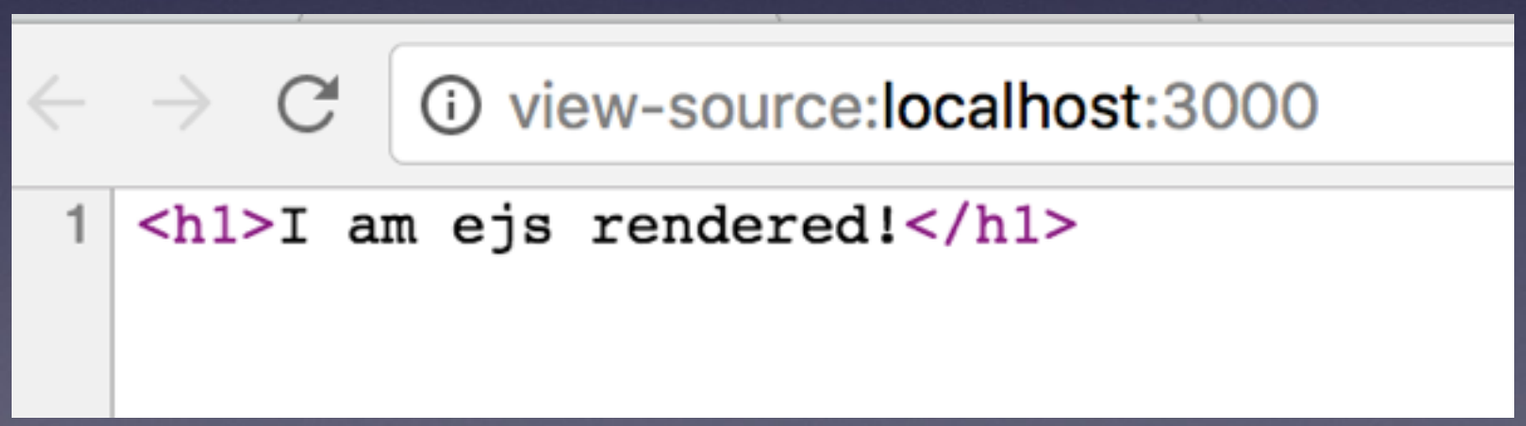




- 创建views文件夹和对应的view文件：这里首先在当前目录下创建了views文件夹，然后在views文件夹下创建了index.ejs这个模板文件，内容如左图所示。

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

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

app.get('/', function (req, res) {
    res.render('index', {content: 'I am ejs rendered!'});
});

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

app.get('/user/:userName', function (req, res, next) {
    let userName = req.params.userName;
    let location = req.query.location;

    res.send(`用户: ${userName} 在 ${location} 登录成功!`)
});

var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;
    console.log('Example app listening at ' +
        'http://%s:%s', host, port);
});
```

- 渲染：调用res.render方法来渲染页面。
- 参数：第一个参数为views目录下的模板文件名称(可以省略后缀，这里省略了.ejs)；第二个参数为渲染到模板的对象。




- 测试访问：访问http://localhost:3000，即可看到刚才的ejs模板在服务器完成了数据渲染，将模板文件中的ejs语法部分替换为数据内容，并且将得到的HTML字符串输出到了浏览器。

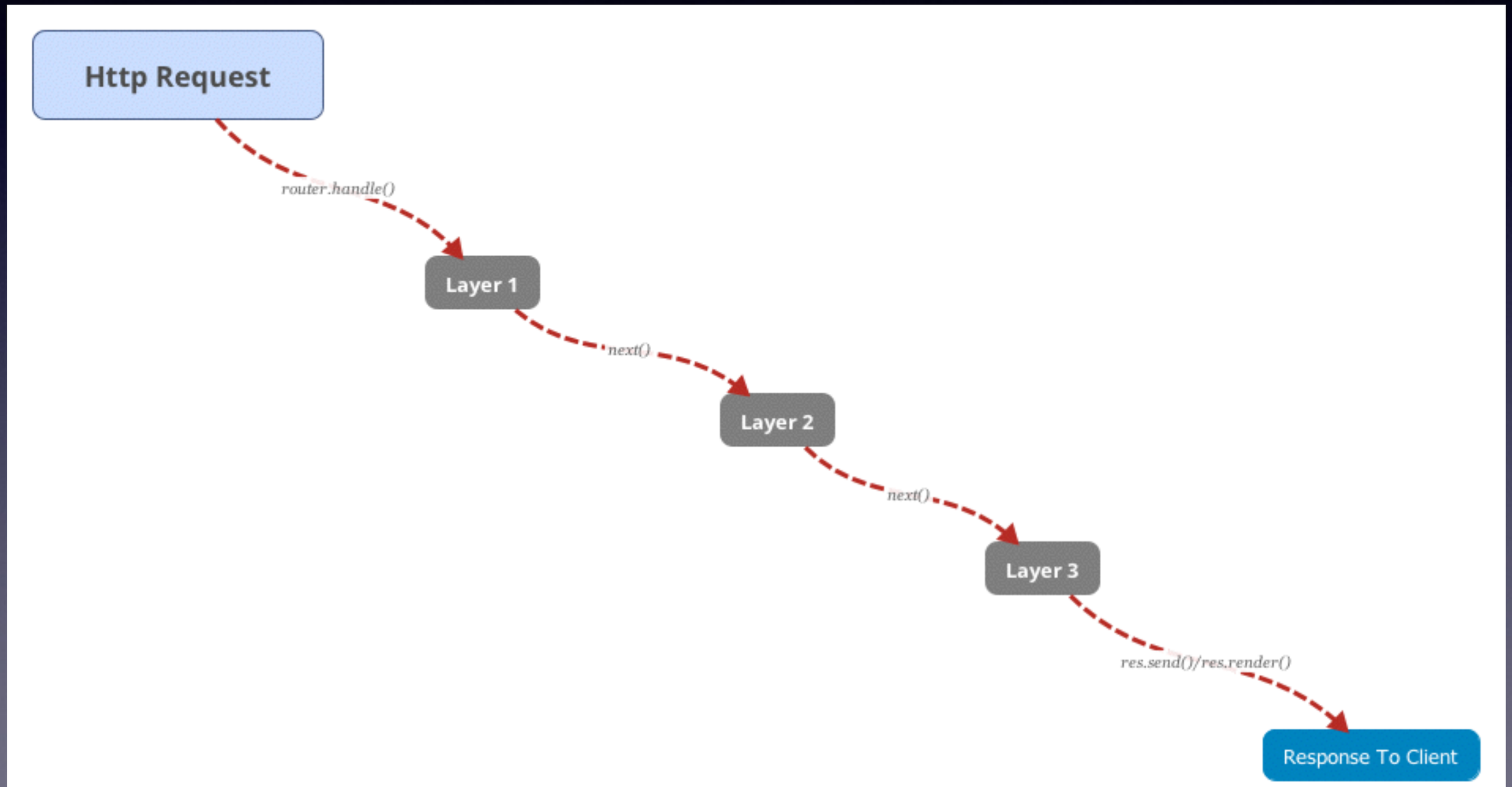
# 快速创建

# Express应用生成器

- 安装生成器：npm install express-generator -g
- 生成目录结构：express -e myapp
- 安装依赖：cd myapp && npm install
- 启动项目：DEBUG=“myapp:server” npm start
- 访问页面：http://127.0.0.1:3000/

# 核心架构

# 核心设计：中间件顺序处理



# 中间件Layer类设计

```
class Layer {
    constructor(path, options, fn) {
        let opts = options || {};
        this.handle = fn;
        this.name = fn.name || '<anonymous>';
        this.params = undefined;
        this.path = undefined;
        this.regexp = pathRegexp(path,
            this.keys = [], opts);
        if (path === '/' && opts.end === false) {
            this.regexp.fast_slash = true;
        }
    }
}
```

- this.handle: 本Layer的处理函数，即客户端请求命中本中间件后，调用的处理函数。

```
class Layer {
    constructor(path, options, fn) {
        let opts = options || {};
        this.handle = fn;
        this.name = fn.name || '<anonymous>';
        this.params = undefined;
        this.path = undefined;
        this.regexp = pathRegexp(path,
            this.keys = [], opts);
        if (path === '/' && opts.end === false) {
            this.regexp.fast_slash = true;
        }
    }
}
```

- this.regexp：本Layer的匹配http请求的path正则表达式，用来判断当前的Layer是否命中本次客户端请求。

```
class Layer {
    constructor(path, options, fn) {
        let opts = options || {};
        this.handle = fn;
        this.name = fn.name || '<anonymous>';
        this.params = undefined;
        this.path = undefined;
        this.regexp = pathRegexp(path,
            this.keys = [], opts);
        if (path === '/' && opts.end === false) {
            this.regexp.fast_slash = true;
        }
    }
}
```

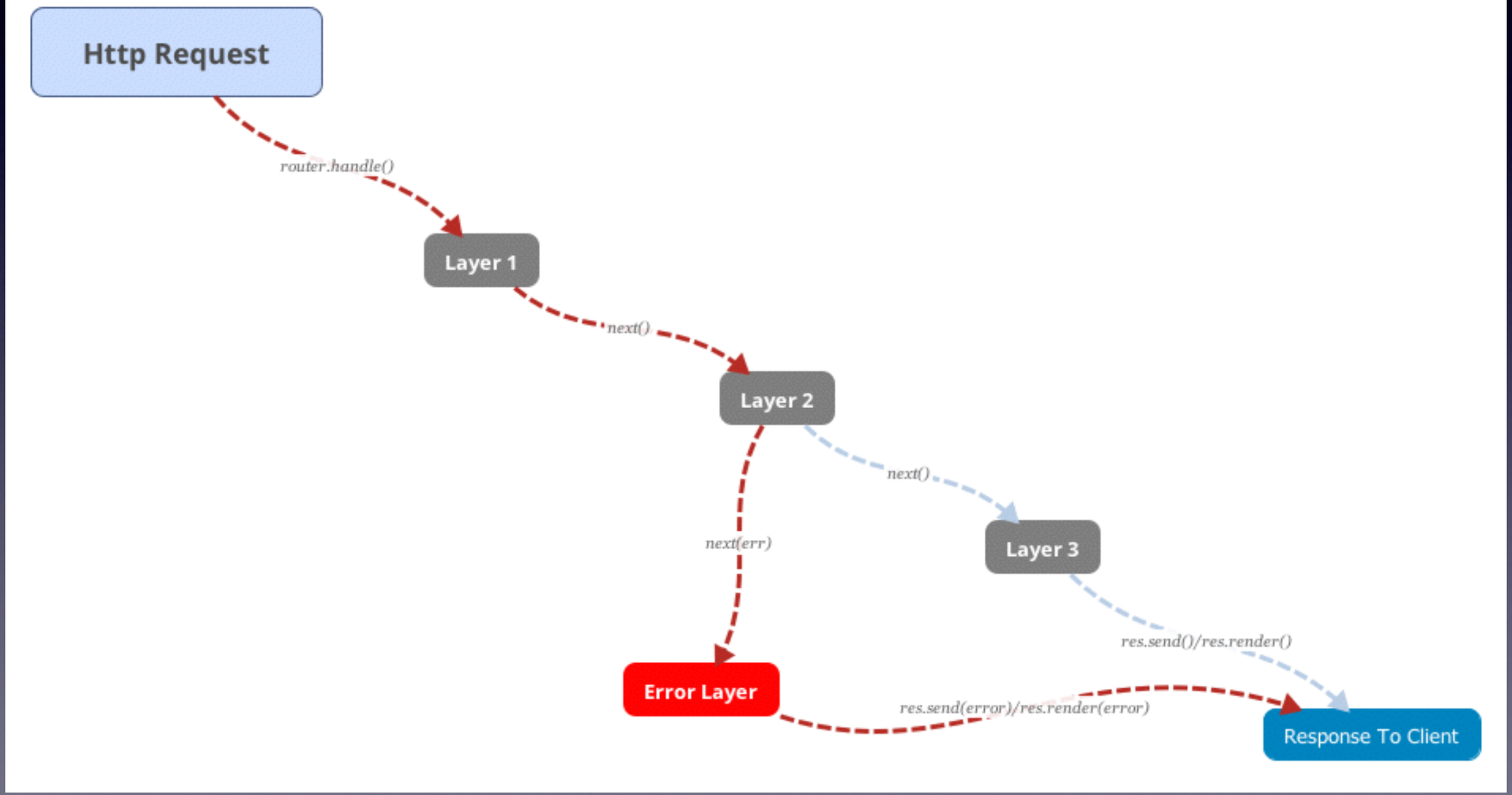
- this.regexp.fast_slash：非路由类的，且不携带路径参数的中间件快速命中：形如app.use(fn)

- regexp.fast_slash为true时，无需进行regexp正则表达式判断，直接命中。

# 每一个Layer匹配逻辑

- 获取到本次Http请求的Path
- 判断regexp.fast_salsh为true，直接匹配成功
- 否则，执行regexp.exec(path)判断
- 匹配成功后调用handle保存的中间件处理函数。

# 特殊的Layer：错误处理中间件



# 判断是否为错误处理中间件

Express判断是否是路由处理中间件做的也很有意思：

function.length === 4

依据中间件的处理函数的入参长度来判断，也就是说，中间件入参为4个的就判断为错误处理中间件，传入error、req、res和next四个参数进行函数调用。

# 这一连串的Layer保存在哪?

```
var proto = module.exports = function
    (options) {
    var opts = options || {};

    function router(req, res, next) {
        router.handle(req, res, next);
    }
    //原型链指向
    router.__proto__ = proto;
    //…省略其余属性
    router.stack = [];
    return router;
};
```

- Router类：这是比较奇怪的构造一个类的方法，构造函数返回一个函数，因此new出来的就是这个函数，并且代码里手动__proto__原型链赋值。

- router.__proto__ = proto
- proto.param = function(){}
- proto.handle = function(){}
- proto.use = function(){}
- proto.route = function(){}
- 这样相当于手动构造了原型链，new Router()后得到的router函数，也可以通过原型链访问param、handle、use和route等方法。

```
var proto = module.exports = function
    (options) {
    var opts = options || {};

    function router(req, res, next) {
        router.handle(req, res, next);
    }
    //原型链指向
    router.__proto__ = proto;
    //...省略其余属性
    router.stack = [];
    return router;
};
```

- stack数组: 保存是所有的Layer（中间件）的地方。

```
var proto = module.exports = function
    (options) {
    var opts = options || {};

    function router(req, res, next) {
        router.handle(req, res, next);
    }
    //原型链指向
    router.__proto__ = proto;
    //…省略其余属性
    router.stack = [];
    return router;
};
```

- router.handle: Express框架的驱动中间件顺序执行的方法，遍历的中间件数组就是前面的router.stack

# 把Layer和Router串起来

入口JS文件
启动

调用express()方法
生成全局的app实例

```
var app = function(req, res, next) {
    app.handle(req, res, next);
 }
//app对象的原型链扩充，包括继承EventEmitter类，扩充方法等
```

app._router: 如果不存在则赋值为 Router类的实例，所有的中间件都保存在这个Router类实例的stack属性数组中。

app._router的初始化是在app.lazyrouter方法中执行的。实际上，只要是向app._router.stack数组中push中间件Layer的操作，之前都会调用一次app.lazyrouter，是否赋值的判断也是在此函数中执行的

app.get/post/all方法：通过调用app._router.route方法，创建一个Layer中间件类实例，保存到app._router.stack数组中。

app.use方法：通过app._router.use方法，创建一个Layer中间件类实例，并将此实例保存到app._router.stack数组中。

有请求到达Express服务器时，app.handle方法被触发，app.handle又会调用app._router.handle方法，此时对存储在app._router.stack数组中的Layer实例中间件开始按照顺序遍历处理。

http.createServer(app).listen(8888)

# Express工作机制总结

- Express构造了一个全局的app函数（对象）
- app._router为Router类的一个实例
- 用户编写的每一个中间件函数经过Layer类封装存储到app._router.stack对应的数组中
- Express将app._router.handle函数经过封装后注入到http.createrServer(fn)中，作为处理http请求的方法入口
- 当每一个请求到达服务器时，触发app._router.handle函数，进而存储在app._router.stack数组中的Layer按照顺序匹配执行

# Thank You

# END