Express 4.x API

updated by github@bajian 标签（空格分隔）： express 翻译 api文档 中文

---

express()

`express()`用来创建一个Express的程序。`express()`方法是***express***模块导出的顶层方法。

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

Methods

express.static(root, [options])

`express.static`是Express中唯一的内建中间件。它以server-static模块为基础开发，负责托管 Express 应用内的静态资源。 参数`root`为静态资源的所在的根目录。 参数`options`是可选的，支持以下的属性：

属性	描述	类型	默认值
dotfiles	是否响应点文件。供选择的值有"allow"，"deny"和"ignore"	String	"ignore"
etag	使能或者关闭etag	Boolean	true
extensions	设置文件延期回退	Boolean	true
index	发送目录索引文件。设置false将不发送。	Mixed	"index.html"
lastModified	设置文件在系统中的最后修改时间到`Last-Modified`头部。可能的取值有`false`和`true`。	Boolean	true
maxAge	在Cache-Control头部中设置`max-age`属性，精度为毫秒(ms)或则一段ms format的字符串	Number	0
redirect	当请求的pathname是一个目录的时候，重定向到尾随"/"	Boolean	true
setHeaders	当响应静态文件请求时设置headers的方法	Funtion	

如果你想获得更多关于使用中间件的细节，你可以查阅Serving static files in Express。

Application()

`app`对象一般用来表示Express程序。通过调用Express模块导出的顶层的`express()`方法来创建它:

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

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

app.listen(3000);
```

`app`对象具有以下的方法：

- 路由HTTP请求；具体可以看app.METHOD和app.param这两个例子。
- 配置中间件；具体请看app.route。
- 渲染HTML视图；具体请看app.render。
- 注册模板引擎；具体请看app.engine。

它还有一些属性设置，这些属性可以改变程序的行为。获得更多的信息，可以查阅Application settings。

Properties

app.locals

`app.locals`对象是一个javascript对象，它的属性就是程序本地的变量。

```
app.locals.title
// => 'My App'

app.locals.email
// => 'me@myapp.com'
```

一旦设定，`app.locals`的各属性值将贯穿程序的整个生命周期，与其相反的是`res.locals`，它只在这次请求的生命周期中有效。

在程序中，你可以在渲染模板时使用这些本地变量。它们是非常有用的，可以为模板提供一些有用的方法，以及`app`级别的数据。通过`req.app.locals`(具体查看req.app)，Locals可以在中间件中使用。

```
app.locals.title = 'My App';
app.locals.strftime = require('strftime');
app.locals.email = 'me@myapp.com';
```

app.mountpath

`app.mountpath`属性是子程序挂载的路径模式。

> 一个子程序是一个`express`的实例，其可以被用来作为路由句柄来处理请求。
>
> ```
> var express = require('express');
> var app = express(); // the main app
> var admin = express(); // the sub app
> admin.get('/', function(req, res) {
>     console.log(admin.mountpath); // /admin
>     res.send('Admin Homepage');
> });
> ```

app.use('/admin', admin); // mount the sub app

它和req对象的[baseUrl](http://expressjs.com/4x/api.html#req.baseUrl)属性比较相似，除了`req.baseUrl`是匹配的URL路径，而不是匹配的模式。如果一个子程序被挂载在多

```
var admin = express();
admin.get('/', function(req, res) {
    console.log(admin.mountpath); // ['adm*n', '/manager']
    res.send('Admin Homepage');
});

var secret = express();
secret.get('/', function(req, res) {
    console.log(secret.mountpath); // /secr*t
    res.send('Admin secret');
```

```
});

admin.use('secr*t', secret); // load the 'secret' router on '/secr*t', on the 'admin' sub app
app.use(['/adm*n', '/manager'], admin); // load the 'admin' router on '/adm*n' and  '/manager' , on the parent app
```

Events

app.on('mount', callback(parent))

当子程序被挂载到父程序时，`mount`事件被发射。父程序对象作为参数，传递给回调方法。

```
var admin = express();
admin.on('mount', function(parent) {
    console.log('Admin Mounted');
    console.log(parent); // refers to the parent app
});

admin.get('/', function(req, res) {
    res.send('Admin Homepage');
});

app.use('/admin', admin);
```

Methods

app.all(path, callback[, callback ...]

`app.all`方法和标准的`app.METHOD()`方法相似，除了它匹配所有的HTTP动词。 对于给一个特殊前缀映射一个全局的逻辑处理，或者无条件匹配，它是很有效的。例如，如果你把下面内容放在所有其他的路由定义的前面，它要求所有从这个点开始的路由需要认证和自动加载一个用户。记住这些回调并不是一定是终点:`loadUser`可以在完成了一个任务后，调用`next()`方法来继续匹配随后的路由。

```
app.all('*', requireAuthentication, loadUser);
```

或者这种相等的形式:

```
    app.all('*', requireAuthentication);
    app.all('*', loadUser);
```

另一个例子是全局的白名单方法。这个例子和前面的很像，然而它只是限制以 `/api` 开头的路径。

```
app.all('/api/*', requireAuthentication);
```

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

路由 `HTTP DELETE` 请求到有特殊回调方法的特殊的路径。获取更多的信息，可以查阅routing guide。 你可以提供多个回调函数，它们的行为和中间件一样，除了这些回调可以通过调用 `next('router')` 来绕过剩余的路由回调。你可以使用这个机制来为一个路由设置一些前提条件，如果不能满足当前路由的处理条件，那么你可以传递控制到随后的路由。

```
app.delete('/', function(req, res) {
    res.send('DELETE request to homepage');
});
```

app.disable(name)

设置类型为布尔的设置名为 `name` 的值为 `false` ，此处的 `name` 是app settings table中各属性的一个。调用 `app.set('foo', false)` 和调用 `app.disable('foo')` 是等价的。 比如:

```
app.disable('trust proxy');
app.get('trust proxy');
// => false
```

app.disabled(name)

返回 `true` 如果布尔类型的设置值 `name` 被禁用为 `false` ，此处的 `name` 是app settings table中各属性的一个。

```
    app.disabled('trust proxy');
    // => true
    app.enable('trust proxy');
    app.disabled('trust proxy');
    // => false
```

app.enable(name)

设置布尔类型的设置值 `name` 为 `true` ，此处的 `name` 是app settings table中各属性的一个。调用 `app.set('foo', true)` 和调用 `app.enable('foo')` 是等价的。

```
app.enable('trust proxy');
app.get('trust proxy');
// => true
```

app.enabled(name)

返回`true`如果布尔类型的设置值`name`被启动为`true`，此处的`name`是app settings table中各属性的一个。

```
app.enabled('trust proxy');
// => false
app.enable('trust proxy');
app.enabled('trust proxy');
// => true
```

app.engine(ext, callback)

注册给定引擎的回调，用来渲染处理ext文件。 默认情况下，Express需要使用`require()`来加载基于文件扩展的引擎。例如，如果你尝试渲染一个`foo.jade`文件，Express在内部调用下面的内容，同时缓存`require()`结果供随后的调用，来加速性能。

```
app.engine('jade', require('jade').__express);
```

使用下面的方法对于那些没有提供开箱即用的`.__express`方法的模板，或者你希望使用不同的模板引擎扩展。 比如，使用EJS模板引擎来渲染`.html`文件：

```
app.engine('html', require('ejs').renderFile);
```

在这个例子中，EJS提供了一个`.renderFile`方法，这个方法满足了Express规定的签名规则：`(path, options, callback)`，然而记住在内部它只是`ejs.__express`的一个别名，所以你可以在不做任何事的情况下直接使用`.ejs`扩展。 一些模板引擎没有遵循这种规范，consolidate.js库映射模板引擎以下面的使用方式，所以他们可以无缝的和Express工作。

```
var engines = require('consolidate');
app.engine('haml', engines.haml);
app.engine('html', engines.hogan);
```

app.get(name)

获得设置名为`name`的app设置的值，此处的`name`是app settings table中各属性的一个。 如下：

```
app.get('title');
// => undefined

app.set('title', 'My Site');
app.get('title');
// => 'My Site'
```

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

路由`HTTP GET`请求到有特殊回调的特殊路径。获取更多的信息，可以查阅routing guide。 你可以提供多个回调函数，它们的行为和中间件一样，除了这些回调可以通过调用`next('router')`来绕过剩余的路由回调。你可以使用这个机制来为一个路由设置一些前提条件，如果请求没能满足当前路由的处理条件，那么传递控制到随后的路由。

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

app.listen(port, [hostname], [backlog], [callback])

绑定程序监听端口到指定的主机和端口号。这个方法和`Node`中的http.Server.listen()是一样的。

```
    var express = require('express');
    var app = express();
    app.listen(3000);
```

通过调用`express()`返回得到的`app`实际上是一个JavaScript的`Function`，被设计用来作为一个回调传递给`Node HTTP servers`来处理请求。这样，其就可以很简便的基于同一份代码提供http和https版本，所以app没有从这些继承(它只是一个简单的回调)。

```
    var express = require('express');
    var https = require('https');
    var http = require('http');

    http.createServer(app).listen(80);
    https.createServer(options, app).listen(443);
```

`app.listen()` 方法是下面所示的一个便利的方法(只针对HTTP协议):

```
app.listen = function() {
    var server = http.createServer(this);
    return server.listen.apply(server, arguments);
};
```

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

路由一个HTTP请求， `METHOD` 是这个请求的HTTP方法，比如 `GET` ， `PUT` ， `POST` 等等，注意是小写的。所以，实际的方法是 `app.get()` ， `app.post()` ， `app.put()` 等等。下面有关于方法的完整的表。 获取更多信息，请看routing guide。 Express支持下面的路由方法，对应与同名的HTTP方法：

• checkout • connect • copy • delete • get • head • lock • merge • mkactivity	• mkcol • move • m-search • notify • options • patch • post • propfind • proppatch	• purege • put • report • search • subscribe • trace • unlock • unsubscribe

> 如果使用上述方法时，导致了无效的javascript的变量名，可以使用中括号符号，比如, `app['m-search']('/', function ...`

你可以提供多个回调函数，它们的行为和中间件一样，除了这些回调可以通过调用 `next('router')` 来绕过剩余的路由回调。你可以使用这个机制来为一个路由设置一些前提条件，如果请求没有满足当前路由的处理条件，那么传递控制到随后的路由。

> 本API文档把使用比较多的HTTP方法 `app.get()` ， `app.post` ， `app.put()` ， `app.delete()` 作为一个个单独的项进行说明。然而，其他上述列出的方法以完全相同的方式工作。

`app.all()` 是一个特殊的路由方法，它不属于HTTP协议中的规定的方法。它为一个路径加载中间件，其对所有的请求方法都有效。

```
app.all('/secret', function (req, res) {
    console.log('Accessing the secret section...');
    next(); // pass control to the next handler
});
```

app.param([name], callback)

给路由参数添加回调触发器，这里的`name`是参数名或者参数数组，`function`是回调方法。回调方法的参数按序是`请求对象`，`响应对象`，`下个中间件`，`参数值`和`参数名`。 如果`name`是数组，会按照各个参数在数组中被声明的顺序将回调触发器注册下来。还有，对于除了最后一个参数的其他参数，在他们的回调中调用`next()`来调用下个声明参数的回调。对于最后一个参数，在回调中调用`next()`将调用位于当前处理路由中的下一个中间件，如果`name`只是一个`string`那就和它是一样的(就是说只有一个参数，那么就是最后一个参数，和数组中最后一个参数是一样的)。 例如，当`:user`出现在路由路径中，你可以映射用户加载的逻辑处理来自动提供`req.user`给这个路由，或者对输入的参数进行验证。

```
    app.param('user', function(req, res, next, id) {
        User.find(id, function(error, user) {
            if (err) {
                next(err);
            }
            else if (user){
                req.user = user;
            } else {
                next(new Error('failed to load user'));
            }
        });
    });
```

对于`Param`的回调定义的路由来说，他们是局部的。它们不会被挂载的app或者路由继承。所以，定义在`app`上的`Param`回调只有是在`app`上的路由具有这个路由参数时才起作用。 在定义`param`的路由上，`param`回调都是第一个被调用的，它们在一个请求-响应循环中都会被调用一次并且只有一次，即使多个路由都匹配，如下面的例子：

```
app.param('id', function(req, res, next, id) {
    console.log('CALLED ONLY ONCE');
    next();
});

app.get('/user/:id', function(req, res, next) {
    console.log('although this matches');
    next();
});

app.get('/user/:id', function(req, res) {
    console.log('and this mathces too');
    res.end();
```

```
});
```

当`GET /user/42`，得到下面的结果:

```
    CALLED ONLY ONCE
    although this matches
    and this matches too
```

```
app.param(['id', 'page'], function(req, res, next, value) {
    console.log('CALLED ONLY ONCE with', value);
    next();
});

app.get('/user/:id/:page', function(req. res, next) {
    console.log('although this matches');
    next();
});

app.get('/user/:id/:page', function (req, res, next) {
    console.log('and this matches too');
    res.end();
});
```

当执行`GET /user/42/3`，结果如下：

```
CALLED ONLY ONCE with 42
CALLED ONLY ONCE with 3
although this matches
and this mathes too
```

> 下面章节描述的`app.param(callback)`在v4.11.0之后被弃用。

通过只传递一个回调参数给`app.param(name, callback)`方法，`app.param(naem, callback)`方法的行为将被完全改变。这个回调参数是关于`app.param(name, callback)`该具有怎样的行为的一个自定义方法，这个方法必须接受两个参数并且返回一个中间件。 这个回调的第一个参数就是需要捕获的url的参数名，第二个参数可以是任一的JavaScript对象，其可能在实现返回一个中间件时被使用。 这个回调方法返回的中间件决定了当URL中包含这个参数时所采取的行为。 在下面的例子中，`app.param(name, callback)`参数签名被修改成了`app.param(name, accessId)`。替换接受一个参数名和回调，`app.param()`现在接受一个参数名和一个数字。

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

app.param(function(param, option){
    return function(req, res, next, val) {
        if (val == option) {
            next();
        }
        else {
            res.sendStatus(403);
        }
    }
});

app.param('id', 1337);

app.get('/user/:id', function(req, res) {
    res.send('Ok');
});

app.listen(3000, function() {
    console.log('Ready');
});
```

在这个例子中，`app.param(name, callback)`参数签名保持和原来一样，但是替换成了一个中间件，定义了一个自定义的数据类型检测方法来检测`user id`的类型正确性。

```
app.param(function(param, validator) {
    return function(req, res, next, val) {
        if (validator(val)) {
            next();
        }
        else {
            res.sendStatus(403);
        }
    }
});

app.param('id', function(candidate) {
    return !isNaN(parseFloat(candidate)) && isFinite(candidate);
});
```

> 在使用正则表达式来，不要使用`.`。例如，你不能使用`/user-.+/`来捕获`user-gami`，用使用`[\\s\\S]`或者`[\\w\\>W]`来代替(正如`/user-[\\s\\S]+/`)。

```
//captures '1-a_6' but not '543-azser-sder'
router.get('/[0-9]+-[[\\w]]*', function);

//captures '1-a_6' and '543-az(ser"-sder' but not '5-a s'
router.get('/[0-9]+-[[\\S]]*', function);

//captures all (equivalent to '.*')
router.get('[[\\s\\S]]*', function);
```

app.path()

通过这个方法可以得到`app`典型的路径，其是一个`string`。

```
var app = express()
  , blog = express()
  , blogAdmin = express();

app.use('/blog', blog);
app.use('/admin', blogAdmin);

console.log(app.path()); // ''
console.log(blog.path()); // '/blog'
console.log(blogAdmin.path()); // '/blog/admin'
```

如果`app`挂载很复杂下，那么这个方法的行为也会很复杂：一种更好用的方式是使用`req.baseUrl`来获得这个app的典型路径。

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

路由`HTTP POST`请求到有特殊回调的特殊路径。获取更多的信息，可以查阅routing guide。 你可以提供多个回调函数，它们的行为和中间件一样，除了这些回调可以通过调用`next('router')`来绕过剩余的路由回调。你可以使用这个机制来为一个路由设置一些前提条件，如果请求没能满足当前路由的处理条件，那么传递控制到随后的路由。

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

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

路由`HTTP PUT`请求到有特殊回调的特殊路径。获取更多的信息，可以查阅routing guide。 你可以提供多个回调函数，它们的行为和中间件一样，除了这些回调可以通过调用`next('router')`来绕过剩余的路由回调。你可以使用这个机制来为一个路由设置一些前提条件，如果请求没能满足当前路由的处理条件，那么传递控制到随后的路由。

```
    app.put('/', function(req, res) {
        res.send('PUT request to homepage');
    });
```

app.render(view, [locals], callback)

通过`callback`回调返回一个`view`渲染之后得到的HTML文本。它可以接受一个可选的参数，可选参数包含了这个`view`需要用到的本地数据。这个方法类似于`res.render()`，除了它不能把渲染得到的**HTML**文本发送给客户端。

> 将`app.render()`当作是可以生成渲染视图字符串的工具方法。在`res.render()`内部，就是使用的`app.render()`来渲染视图。
>
> 如果使能了视图缓存，那么本地变量缓存就会保留。如果你想在开发的过程中缓存视图，设置它为`true`。在生产环境中，视图缓存默认是打开的。

```
    app.render('email', function(err, html) {
    // ...
    });

    app.render('email', {name:'Tobi'}, function(err, html) {
    // ...
    });
```

app.route(path)

返回一个单例模式的路由的实例，之后你可以在其上施加各种HTTP动作的中间件。使用`app.route()`来避免重复路由名字(例如错字错误)--说的意思应该是使用`app.router()`这个单例方法来避免同一个路径多个路由实例。

```
var app = express();

app.route('/events')
.all(function(req, res, next) {
  // runs for all HTTP verbs first
  // think of it as route specific middleware!
})
```

```
.get(function(req, res, next) {
  res.json(...);
})
.post(function(req, res, next) {
  // maybe add a new event...
})
```

app.set(name, value)

给`name`设置项赋`value`值，`name`是app settings table中属性的一项。 对于一个类型是布尔型的属性调用`app.set('foo', ture)`等价于调用`app.enable('foo')`。同样的，调用`app.set('foo', false)`等价于调用`app.disable('foo')`。 可以使用`app.get()`来取得设置的值：

```
    app.set('title', 'My Site');
    app.get('title'); // 'My Site'
```

Application Settings 如果`name`是程序设置之一，它将影响到程序的行为。下边列出了程序中的设置。

属性	类型	值	默认
case sensitive routing	Boolean	启用区分大小写。	不启用。对`/Foo`和`/foo`处理是一样。
env	String	环境模型。	process.env.NODE_ENV(NODE_ENV环境变量)或者"development"
etag	Varied	设置`ETag`响应头。可取的值，可以查阅etag options table。更多关于HTTP ETag header。	weak
jsonp callback name	String	指定默认JSONP回调的名称。	?callback=
json replacer	String	JSON替代品回调	null
json spaces	Number	当设置了这个值后，发送缩进空格美化过的JSON字符串。	Disabled
query		设置值为`false`来禁用`query parser`，或者设置`simple`,`extended`，	

parser	Varied	也可以自己实现 `query string` 解析函数。 `simple` 基于 `Node` 原生的 `query` 解析，querystring。	"extend"
strict routing	Boolean	启用严格的路由。	不启用。 对 `/foo` 和 `/foo/` 的路由处理是一样。
subdomain offset	Number	用来删除访问子域的主机点分部分的个数	2
trust proxy	Varied	指示 `app` 在一个反向代理的后面，使用 `x-Forwarded-*` 来确定连接和客户端的IP地址。 注意: `X-Forwarded-*` 头部很容易被欺骗，所有检测客户端的IP地址是靠不住的。 `trust proxy` 默认不启用。当启用时， Express尝试通过前端代理或者一系列代理来获取已连接的客户端IP地址。 `req.ips` 属性包含了已连接客户端IP地址的一个数组。为了启动它，需要设置在下面trust proxy options table中定义的值。 `trust proxy` 的设置实现使用了 `proxy-addr` 包。 如果想获得更多的信息，可以查阅它的文档	Disable
views	String or Array	`view` 所在的目录或者目录数组。如果是一个数组，将按在数组中的顺序来查找 `view` 。	process.cwd() + '/views'
view cache	Boolean	启用视图模板编译缓存。	在生成环境默认开启。
view engine	String	省略时，默认的引擎被扩展使用。	
x-powered-by	Boolean	启用 `X-Powered-By:Express` HTTP头部	true

Options for `trust proxy` settings 查阅Express behind proxies来获取更多信息。

Type	Value
Boolean	如果为 `true` ，客户端的IP地址作为 `X-Forwarded-*` 头部的最左边的条目。如果为 `false` ，可以理解为 `app` 直接与英特网直连， 客户端的IP地址衍生自 `req.connection.remoteAddress` 。 `false` 是默认设置。
	一个IP地址，子网，或者一组IP地址，和委托子网。下面列出的是一个预先配置的子网名列表。 • loopback - `127.0.0.1/8` , `::1/128` • linklocal - `169.254.0.0/16` , `fe80::/10`

IP addresses	• uniquelocal - `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`, `fc00::/7` 使用下面方法中的任何一种来设置IP地址: ``` app.set('trust proxy', 'loopback') // specify a single subnet app.set('trust proxy', 'loopback, 123.123.123.123') // specify a subnet and an address app.set('trust proxy', 'loopback, linklocal, uniquelocal') // specify multiple subnets as CSV app.set('trust proxy', ['loopback', 'linklocal', 'uniquelocal']) // specify multiple subnets as an array ``` 当指定IP地址之后, 这个IP地址或子网会被设置了这个IP地址或子网的`app`排除在外, 最靠近程序服务的没有委托的地址将被看做客户端IP地址。
Number	信任从反向代理到app中间小于等于n跳的连接为客户端。
Function	客户自定义委托代理信任机制。如果你使用这个，请确保你自己知道你在干什么。 ``` app.set('trust proxy', function (ip) { if (ip === '127.0.0.1' || ip === '123.123.123.123') return true; // trusted IPs else return false; }) ```

Options for `etag` settings `ETag`功能的实现使用了etag包。如果你需要获得更多的信息，你可以查阅它的文档。

Type	Value
Boolean	设置为`true`，启用weak ETag。这个是默认设置。设置`false`，禁用所有的ETag。
String	如果是`strong`，使能strong ETag。如果是`weak`，启用`weak` ETag。
Function	客户自定义`ETag`方法的实现. 如果你使用这个，请确保你自己知道你在干什么。 ``` app.set('etag', function (body, encoding) { return generateHash(body, encoding); // consider the function is defined }) ```

app.use([path,], function [, function...])

挂载中间件方法到路径上。如果路径未指定，那么默认为"/"。

> 一个路由将匹配任何路径如果这个路径以这个路由设置路径后紧跟着"/"。比如：`app.use('/appale', ...)`将匹配"/apple"，"/apple/images"，"/apple/images/news"等。
>
> 中间件中的`req.originalUrl`是`req.baseUrl`和`req.path`的组合，如下面的例子所示。

```
app.use('/admin', function(req, res, next) {
// GET 'http://www.example.com/admin/new'
console.log(req.originalUrl); // '/admin/new'
console.log(req.baseUrl); // '/admin'
console.log(req.path);// '/new'
});
```

在一个路径上挂载一个中间件之后，每当请求的路径的前缀部分匹配了这个路由路径，那么这个中间件就会被执行。 由于默认的路径为`/`，中间件挂载没有指定路径，那么对于每个请求，这个中间件都会被执行。

```
    // this middleware will be executed for every request to the app.
    app.use(function(req, res, next) {
        console.log('Time: %d', Date.now());
        next();
    });
```

中间件方法是顺序处理的，所以中间件包含的顺序是很重要的。

```
    // this middleware will not allow the request to  go beyond it
    app.use(function(req, res, next) {
        res.send('Hello World');
    });

    // this middleware will never reach this route
    app.use('/', function(req, res) {
        res.send('Welcome');
    });
```

路径可以是代表路径的一串字符，一个路径模式，一个匹配路径的正则表达式，或者他们的一组集合。

> 下面是路径的简单的例子。

Type	Example
Path	`// will match paths starting with /abcd` `app.use('/abcd', function (req, res, next) {` `  next();` `})`
Path Pattern	`// will match paths starting with /abcd and /abd` `app.use('/abc?d', function (req, res, next) {` `  next();` `})` `// will match paths starting with /abcd, /abbcd, /abbbbbcd and so on` `app.use('/ab+cd', function (req, res, next) {` `  next();` `})` `// will match paths starting with /abcd, /abxcd, /abFOOcd, /abbArcd and so on` `app.use('/ab\*cd', function (req, res, next) {` `  next();` `})` `// will match paths starting with /ad and /abcd` `app.use('/a(bc)?d', function (req, res, next) {` `  next();` `})`
Regular Expression	`// will match paths starting with /abc and /xyz` `app.use(/\/abc|\/xyz/, function (req, res, next) {` `  next();` `})`
Array	`// will match paths starting with /abcd, /xyza, /lmn, and /pqr` `app.use(['/abcd', '/xyza', /\/lmn|\/pqr/], function (req, res, next) {` `  next();` `})`

方法可以是一个中间件方法，一系列中间件方法，一组中间件方法或者他们的集合。由于`router`和`app`实现了中间件接口，你可以像使用其他任一中间件方法那样使用它们。

Usage	Example
单个中间件	你可以局部定义和挂载一个中间件。 `app.use(function (req, res, next) {` `  next();` `})` 一个`router`是有效的中间件。 `var router = express.Router();` `router.get('/', function (req, res, next) {` `  next();` `})` `app.use(router);` 一个`Express`程序是一个有效的中间件。 `var subApp = express();` `subApp.get('/', function (req, res, next) {` `  next();` `})` `app.use(subApp);`
一系列中间件	对于一个相同的挂载路径，你可以挂载超过一个的中间件。 `var r1 = express.Router();` `r1.get('/', function (req, res, next) {` `  next();` `})` `var r2 = express.Router();` `r2.get('/', function (req, res, next) {` `  next();` `})`

	app.use(r1, r2);
一组中间件	在逻辑上使用一个数组来组织一组中间件。如果你传递一组中间件作为第一个或者唯一的参数，接着你需要指定挂载的路径。 var r1 = express.Router(); r1.get('/', function (req, res, next) { next(); }) var r2 = express.Router(); r2.get('/', function (req, res, next) { next(); }) app.use('/', [r1, r2]);
组合	你可以组合下面的所有方法来挂载中间件。 function mw1(req, res, next) { next(); } function mw2(req, res, next) { next(); } var r1 = express.Router(); r1.get('/', function (req, res, next) { next(); }); var r2 = express.Router(); r2.get('/', function (req, res, next) { next(); }); var subApp = express(); subApp.get('/', function (req, res, next) { next(); }); app.use(mw1, [mw2, r1, r2], subApp);

下面是一些例子，在`Express`程序中使用`express.static`中间件。 为程序托管位于程序目录下的`public`目录下的静态资源：

```
// GET /style.css etc
app.use(express.static(__dirname + '/public'));
```

在`/static`路径下挂载中间件来提供静态资源托管服务，只当请求是以`/static`为前缀的时候。

```
// GET /static/style.css etc.
app.use('/static', express.static(express.__dirname + '/public'));
```

通过在设置静态资源中间件之后加载日志中间件来关闭静态资源请求的日志。

```
    app.use(express.static(__dirname + '/public'));
    app.use(logger());
```

托管静态资源从不同的路径，但`./public`路径比其他更容易被匹配：

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

Request

`req`对象代表了一个HTTP请求，其具有一些属性来保存请求中的一些数据，比如`query string`， `parameters`， `body`， `HTTP headers`等等。在本文档中，按照惯例，这个对象总是简称为`req`(http响应简称为`res`)，但是它们实际的名字由这个回调方法在那里使用时的参数决定。 如下例子:

```
    app.get('/user/:id', function(req, res) {
        res.send('user' + req.params.id);
    });
```

其实你也可以这样写：

```
    app.get('/user/:id', function(request, response) {
        response.send('user' + request.params.id);
    });
```

Properties

在`Express 4`中， `req.files`默认在`req`对象中不再是可用的。为了通过`req.files`对象来获得上传的文件，你可以使用一个`multipart-handling`(多种处理的工具集)中间件，比如`busboy`， `multer`， `formidable`， `multipraty`， `connect-multiparty`或者`pez`。

req.app

这个属性持有`express`程序实例的一个引用，其可以作为中间件使用。 如果你按照这个模式，你创建一个模块导出一个中间件，这个中间件只在你的主文件中`require()`它，那么这个中间件可以通过`req.app`来获取express的实例。 例如:

```
    // index.js
    app.get("/viewdirectory", require('./mymiddleware.js'));
```

```
// mymiddleware.js
module.exports = function(req, res) {
    res.send('The views directory is ' + req.app.get('views'));
};
```

req.baseUrl

一个路由实例挂载的Url路径。

```
    var greet = express.Router();
    greet.get('/jp', function(req, res) {
        console.log(req.baseUrl); // greet
        res.send('Konichiwa!');
    });
    app.use('/greet', greet);
```

即使你使用的路径模式或者一系列路径模式来加载路由，`baseUrl`属性返回匹配的字符串，而不是路由模式。下面的例子，`greet`路由被加载在两个路径模式上。

```
app.use(['/gre+t', 'hel{2}o'], greet); // load the on router on '/gre+t' and '/hel{2}o'
```

当一个请求路径是`/greet/jp`，`baseUrl`是`/greet`，当一个请求路径是`/hello/jp`，`req.baseUrl`是`/hello`。`req.baseUrl`和`app`对象的mountpath属性相似，除了`app.mountpath`返回的是路径匹配模式。

req.body

在请求的body中保存的是提交的一对对键值数据。默认情况下，它是`undefined`，当你使用比如`body-parser`和`multer`这类解析`body`数据的中间件时，它是填充的。下面的例子，给你展示了怎么使用`body-parser`中间件来填充`req.body`。

```
var app = require('express');
var bodyParser = require('body-parser');
var multer = require('multer'); // v1.0.5
var upload = multer(); // for parsing multipart/form-data
app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({extended:true})); // for parsing application/x-www-form-urlencoded

app.post('/profile', upload.array(), function(req, res, next) {
    console.log(req.body);
    res.json(req.body);
});
```

req.cookies

当使用`cookie-parser`中间件的时候，这个属性是一个对象，其包含了请求发送过来的`cookies`。如果请求没有带`cookies`，那么其值为`{}`。

```
// Cookie: name=tj
req.cookies.name
// => "tj"
```

获取更多信息，问题，或者关注，可以查阅cookie-parser。

req.fresh

指示这个请求是否是新鲜的。其和`req.stale`是相反的。 当`cache-control`请求头没有`no-cache`指示和下面中的任一一个条件为`true`，那么其就为`true`：

- `if-modified-since`请求头被指定，和`last-modified`请求头等于或者早于`modified`响应头。
- `if-none-match`请求头是`*`。
- `if-none-match`请求头在被解析进它的指令之后，和`etag`响应头的值不相等

> ps:If-None-Match作用: If-None-Match和ETag一起工作，工作原理是在HTTP Response中添加ETag信息。 当用户再次请求该资源时，将在HTTP Request 中加入If-None-Match信息(ETag的值)。如果服务器验证资源的ETag没有改变（该资源没有更新），将返回一个304状态告诉客户端使用本地缓存文件。否则将返回200状态和新的资源和Etag. 使用这样的机制将提高网站的性能

```
req.fresh
// => true
```

req.hostname

包含了源自`Host` HTTP头部的`hostname`。 当`trust proxy`设置项被设置为启用值，`X-Forwarded-Host`头部被使用来代替`Host`。这个头部可以被客户端或者代理设置。

```
// Host: "example.com"
req.hostname
// => "example.com"
```

req.ips

当`trust proxy`设置项被设置为启用值，这个属性包含了一组在`X-Forwarded-For`请求头中指定的IP地址。不然，其就包含一个空的数组。这个头部可以被客户端或者代理设置。 例如，如果`X-Forwarded-For`是`client`， `proxy1`， `proxy2`， `req.ips`就是`["clinet", "proxy1", "proxy2"]`，这里`proxy2`就是最远的下游。

req.originalUrl

> `req.url`不是一个原生的`Express`属性，它继承自Node's http module。

这个属性很像`req.url`；然而，其保留了原版的请求链接，允许你自由地重定向`req.url`到内部路由。比如，`app.use()`的`mounting`特点可以重定向`req.url`跳转到挂载点。

```
// GET /search?q=something
req.originalUrl
// => "/search?q=something"
```

req.params

一个对象，其包含了一系列的属性，这些属性和在路由中命名的参数名是一一对应的。例如，如果你有`/user/:name`路由，`name`属性可作为`req.params.name`。这个对象默认值为`{}`。

```
// GET /user/tj
req.params.name
// => "tj"
```

当你使用正则表达式来定义路由规则，捕获组的组合一般使用`req.params[n]`，这里的`n`是第几个捕获租。这个规则被施加在无名通配符匹配，比如`/file/*`的路由：

```
// GET /file/javascripts/jquery.js
```

```
req.params[0]
// => "javascripts/jquery.js"
```

req.path

包含请求URL的部分路径。

```
// example.com/users?sort=desc
req.path
// => "/users"
```

> 当在一个中间件中被调用，挂载点不包含在`req.path`中。你可以查阅app.use()获得跟多的信息。

req.protocol

请求的协议，一般为`http`，当启用TLS加密，则为`https`。 当`trust proxy`设置一个启用的参数，如果存在`X-Forwarded-Proto`头部的话，其将被信赖和使用。这个头部可以被客户端或者代理设置。

```
req.ptotocol
// => "http"
```

req.query

一个对象，为每一个路由中的`query string`参数都分配一个属性。如果没有`query string`，它就是一个空对象，`{}`。

```
// GET /search?q=tobi+ferret
req.query.q
// => "tobi ferret"

// GET /shoes?order=desc&shoe[color]=blue&shoe[type]=converse
req.query.order
// => "desc"

req.query.shoe.color
// => "blue"

req.query.shoe.type
// => "converse"
```

req.route

当前匹配的路由，其为一串字符。比如：

```
app.get('/user/:id?', function userIdHandler(req, res) {
    console.log(req.route);
    res.send('GET')
})
```

前面片段的输出为:

```
{ path:"/user/:id?"
    stack:
    [
        { handle:[Function:userIdHandler],
          name:"userIdHandler",
          params:undefined,
          path:undefined,
          keys:[],
          regexp:/^\/?$/i,
          method:'get'
        }
    ]
    methods:{get:true}
}
```

req.secure

一个布尔值，如果建立的是TLS的连接，那么就为`true`。等价与：

```
    'https' == req.protocol;
```

req.signedCookies

当使用`cookie-parser`中间件的时候，这个属性包含的是请求发过来的签名`cookies`，这个属性取得的是不含签名，可以直接使用的值。签名的`cookies`保存在不同的对象中来体现开发者的意图；不然，一个恶意攻击可以被施加在`req.cookie`值上(它是很容易被欺骗的)。记住，签名一个`cookie`不是把它藏起来或者加密；而是简单的防止篡改(因为签名使用的加密是私人的)。如果没有发送签名的`cookie`，那么这个属性默认为`{}`。

```
// Cookie:  user=tobi.CP7AWaXDfAKIRfH49dQzKJx7sKzzSoPq7/AcBBRVwlI3
req.signedCookies.user
// => "tobi"
```

为了获取更多的信息，问题或者关注，可以参阅cookie-parser。

req.stale

指示这个请求是否是`stale`(陈旧的)，它与`req.fresh`是相反的。更多信息，可以查看req.fresh。

```
req.stale
// => true
```

req.subdomains

请求中域名的子域名数组。

```
// Host: "tobi.ferrets.example.com"
req.subdomains
// => ["ferrets", "tobi"]
```

req.xhr

一个布尔值，如果`X-Requested-With`的值为`XMLHttpRequest`，那么其为`true`，其指示这个请求是被一个客服端库发送，比如`jQuery`。

```
req.xhr
// => true
```

Methods

req.accepts(types)

检查这个指定的内容类型是否被接受，基于请求的`Accept` HTTP头部。这个方法返回最佳匹配，如果没有一个匹配，那么其返回`undefined`(在这个case下，服务器端应该返回406和"Not Acceptable")。`type`值可以是一个单的`MIME type`字符串(比如`application/json`)，一个扩展名比如`json`，一个逗号分隔的列表，或者一个数组。对于一个列表或者数组，这个方法返回最佳项(如果有的话)。

```
// Accept: text/html
req.accepts('html');
// => "html"

// Accept: text/*, application/json
req.accepts('html');
// => "html"

req.accepts('text/html');
// => "text/html"

req.accepts(['json', 'text']);
// => "json"

req.accepts('application/json');
// => "application/json"

// Accept: text/*, application/json
req.accepts('image/png');
req.accepts('png');
// => undefined

// Accept: text/*;q=.5, application/json
req.accepts(['html', 'json']);
// => "json"
```

获取更多信息，或者如果你有问题或关注，可以参阅accepts。

req.acceptsCharsets(charset[, ...])

返回指定的字符集集合中第一个的配置的字符集，基于请求的`Accept-Charset`HTTP头。如果指定的字符集没有匹配的，那么就返回false。 获取更多信息，或者如果你有问题或关注，可以参阅accepts。

req.acceptsEncodings(encoding[, ...])

返回指定的编码集合中第一个的配置的编码，基于请求的`Accept-Encoding`HTTP头。如果指定的编码集没有匹配的，那么就返回false。 获取更多信息，或者如果你有问题或关注，可以参阅accepts。

req.acceptsLanguages(lang [, ...])

返回指定的语言集合中第一个的配置的语言，基于请求的`Accept-Language`HTTP头。如果指定的语言集没有匹配的，那么就返回false。 获取更多信息，或者如果你有问

题或关注，可以参阅accepts。

req.get(field)

返回指定的请求HTTP头部的域内容(不区分大小写)。`Referrer`和`Referer`的域内容可互换。

```
req.get('Content-type');
// => "text/plain"

req.get('content-type');
// => "text/plain"

req.get('Something')
// => undefined
```

其是`req.header(field)`的别名。

req.is(type)

如果进来的请求的`Content-type`头部域匹配参数`type`给定的`MIME type`，那么其返回`true`。否则返回`false`。

```
// With Content-Type: text/html; charset=utf-8
req.is('html');
req.is('text/html');
req.is('text/*');
// => true

// When Content-Type is application/json
req.is('json');
req.is('application/json');
req.is('application/*');
// => true

req.is('html');
// => false
```

获取更多信息，或者如果你有问题或关注，可以参阅type-is。

req.param(naem, [, defaultValue])

> 过时的。可以在适合的情况下，使用`req.params`， `req.body`或者`req.query`。

返回当前参数`name`的值。

```
// ?name=tobi
req.param('name')
// => "tobi"

// POST name=tobi
req.param('name')
// => "tobi"
// /user/tobi for /user/:name
req.param('name')
// => "tobi"
```

按下面给出的顺序查找：

- req.params
- req.body
- req.query

可选的，你可以指定一个`defaultValue`来设置一个默认值，如果这个参数在任何一个请求的对象中都不能找到。

> 直接通过`req.params`， `req.body`， `req.query`取得应该更加的清晰-除非你确定每一个对象的输入。 `Body-parser`中间件必须加载，如果你使用`req.param()`。详细请看req.body。

Response

`res`对象代表了当一个HTTP请求到来时， `Express`程序返回的HTTP响应。在本文档中，按照惯例，这个对象总是简称为`res`(http请求简称为`req`)，但是它们实际的名字由这个回调方法在那里使用时的参数决定。 例如：

```
app.get('/user/:id', function(req, res) {
    res.send('user' + req.params.id);
});
```

这样写也是一样的：

```
app.get('/user/:id', function(request, response) {
    response.send('user' + request.params.id);
});
```

Properties

res.app

这个属性持有`express`程序实例的一个引用，其可以在中间件中使用。 `res.app`和请求对象中的`req.app`属性是相同的。

res.headersSent

布尔类型的属性，指示这个响应是否已经发送HTTP头部。

```
app.get('/', function(req, res) {
    console.log(res.headersSent); // false
    res.send('OK'); // send之后就发送了头部
    console.log(res.headersSent); // true
});
```

res.locals

一个对象，其包含了本次请求的响应中的变量和因此它的变量只提供给本次请求响应的周期内视图渲染里使用(如果有视图的话)。 其他方面，其和`app.locals`是一样的。 这个参数在导出请求级别的信息是很有效的，这些信息比如请求路径，已认证的用户，用户设置等等。

```
app.use(function(req, res, next) {
    res.locals.user = req.user;
    res.locals.authenticated = !req.user.anonymous;
    next();
});
```

Methods

res.append(field [, value])

> res.append()方法在`Expresxs`4.11.0以上版本才支持。

在指定的`field`的HTTP头部追加特殊的值`value`。如果这个头部没有被设置，那么将用`value`新建这个头部。`value`可以是一个字符串或者数组。 注意:
在`res.append()`之后调用`app.set()`函数将重置前面设置的值。

```
res.append('Lind', ['<http://localhost>', '<http://localhost:3000>']);
res.append('Set-Cookie', 'foo=bar;Path=/;HttpOnly');
res.append('Warning', '199 Miscellaneous warning');
```

res.attachment([filename])

设置HTTP响应的`Content-Disposition`头内容为"attachment"。如果提供了`filename`，那么将通过`res.type()`获得扩展名来设置`Content-Type`，并且设置`Content-Disposition`内容为"filename="parameter。

```
res.attachment();
// Content-Disposition: attachment

res.attachment('path/to/logo.png');
// Content-Disposition: attachment; filename="logo.png"
// Content-Type: image/png
```

res.cookie(name, value [,options])

设置`name`和`value`的`cookie`，`value`参数可以是一串字符或者是转化为json字符串的对象。 options是一个对象，其可以有下列的属性。

属性	类型	描述
domain	String	设置cookie的域名。默认是你本app的域名。
expires	Date	cookie的过期时间，GMT格式。如果没有指定或者设置为0，则产生新的cookie。
httpOnly	Boolean	这个cookie只能被web服务器获取的标示。
maxAge	String	是设置过去时间的方便选项，其为过期时间到当前时间的毫秒值。
path	String	cookie的路径。默认值是`/`。
secure	Boolean	标示这个cookie只用被`HTTPS`协议使用。
signed	Boolean	指示这个cookie应该是签名的。

> res.cookie()所作的都是基于提供的`options`参数来设置`Set-Cookie`头部。没有指定任何的`options`，那么默认值在`RFC6265`中指定。

使用实例：

```
res.cookie('name', 'tobi', {'domain':'.example.com', 'path':'/admin', 'secure':true});
res.cookie('remenberme', '1', {'expires':new Date(Date.now() + 90000), 'httpOnly':true});
```

`maxAge`是一个方便设置过期时间的方便的选项，其以当前时间开始的毫秒数来计算。下面的示例和上面的第二条功效一样。

```
res.cookie('rememberme', '1', {'maxAge':90000}, "httpOnly":true);
```

你可以设置传递一个对象作为`value`的参数。然后其将被序列化为Json字符串，被`bodyParser()`中间件解析。

```
res.cookie('cart', {'items':[1, 2, 3]});
res.cookie('cart', {'items':[1, 2, 3]}, {'maxAge':90000});
```

当我们使用`cookie-parser`中间件的时候，这个方法也支持签名的cookie。简单地，在设置`options`时包含`signed`选项为`true`。然后`res.cookie()`将使用传递给`cookieParser(secret)`的密钥来签名这个值。

```
res.cookie('name', 'tobi', {'signed':true});
```

res.clearCookie(name [,options])

根据指定的`name`清除对应的cookie。更多关于`options`对象可以查阅`res.cookie()`。

```
res.cookie('name', 'tobi', {'path':'/admin'});
res.clearCookie('name', {'path':'admin'});
```

res.download(path, [,filename], [,fn])

传输`path`指定文件作为一个附件。通常，浏览器提示用户下载。默认情况下，`Content-Disposition`头部"filename="的参数为`path`(通常会出现在浏览器的对话框中)。通过指定`filename`参数来覆盖默认值。 当一个错误发生时或者传输完成，这个方法将调用`fn`指定的回调方法。这个方法使用`res.sendFile()`来传输文件。

```
res.download('/report-12345.pdf');

res.download('/report-12345.pdf', 'report.pdf');
```

```
res.download('report-12345.pdf', 'report.pdf', function(err) {
// Handle error, but keep in mind the response may be partially-sent
// so check res.headersSent
if (err) {
} else {
// decrement a download credit, etc.
}
});
```

res.end([data] [, encoding])

结束本响应的过程。这个方法实际上来自`Node`核心模块，具体的是response.end() method of http.ServerResponse。 用来快速结束请求，没有任何的数据。如果你需要发送数据，可以使用res.send()和res.json()这类的方法。

```
res.end();
res.status(404).end();
```

res.format(object)

进行内容协商，根据请求的对象中`Accept` HTTP头部指定的接受内容。它使用req.accepts()来选择一个句柄来为请求服务，这些句柄按质量值进行排序。如果这个头部没有指定，那么第一个方法默认被调用。当不匹配时，服务器将返回`406` "Not Acceptable"，或者调用`default`回调。 `Content-Type`请求头被设置，当一个回调方法被选择。然而你可以改变他，在这个方法中使用这些方法，比如`res.set()`或者`res.type()`。 下面的例子，将回复`{"message":"hey"}`，当请求的对象中`Accept`头部设置成"application/json"或者"*/json*"(不过如果是`/*`，然后这个回复就是"hey")。

```
res.format({
    'text/plain':function() {
        res.send('hey');
    },
    'text/html':function() {
        res.send('<p>hey</p>');
    },
    'application/json':function() {
        res.send({message:'hey'});
    },
    'default':function() {
        res.status(406).send('Not Acceptable');
    }
})
```

除了规范化的MIME类型之外，你也可以使用拓展名来映射这些类型来避免冗长的实现：

```
res.format({
    text:function() {
        res.send('hey');
    },
    html:function() {
        res.send('<p>hey</p>');
    },
    json:function() {
        res.send({message:'hey'});
    }
})
```

res.get(field)

返回`field`指定的HTTP响应的头部。匹配是区分大小写。

```
res.get('Content-Type');
// => "text/plain"
```

res.json([body])

发送一个json的响应。这个方法和将一个对象或者一个数组作为参数传递给`res.send()`方法的效果相同。不过，你可以使用这个方法来转换其他的值到json，例如`null`，`undefined`。(虽然这些都是技术上无效的JSON)。

```
res.json(null);
res.json({user:'tobi'});
res.status(500).json({error:'message'});
```

res.jsonp([body])

发送一个json的响应，并且支持JSONP。这个方法和`res.json()`效果相同，除了其在选项中支持JSONP回调。

```
res.jsonp(null)
// => null

res.jsonp({user:'tobi'})
```

```
// => {"user" : "tobi"}

res.status(500).jsonp({error:'message'})
 // => {"error" : "message"}
```

默认情况下，jsonp的回调方法简单写作`callback`。可以通过jsonp callback name设置来重写它。 下面是一些例子使用JSONP响应，使用相同的代码:

```
// ?callback=foo
res.jsonp({user:'tobo'})
// => foo({"user":"tobi"})

app.set('jsonp callback name', 'cb')

// ?cb=foo
res.status(500).jsonp({error:'message'})
// => foo({"error":"message"})
```

res.links(links)

连接这些`links`， `links`是以传入参数的属性形式提供，连接之后的内容用来填充响应的Link HTTP头部。

```
res.links({
    next:'http://api.example.com/users?page=2',
    last:'http://api.example.com/user?page=5'
});
```

效果:

```
Link:<http://api.example.com/users?page=2>;rel="next",
<http://api.example.com/users?page=5>;rel="last"
```

res.location(path)

设置响应的`Location` HTTP头部为指定的`path`参数。

```
    res.location('/foo/bar');
    res.location('http://example.com');
```

```
    res.location('back');
```

当`path`参数为`back`时，其具有特殊的意义，其指定URL为请求对象的`Referer`头部指定的URL。如果请求中没有指定，那么其即为"/"。

> Express传递指定的URL字符串作为回复给浏览器响应中的`Location`头部的值，不检测和操作，除了`back`这个参数。浏览器会将用户重定向到`location`设置的url或者`Referer`的url（`back`参数的情况）

res.redirect([status,] path)

重定向来源于指定`path`的URL，以及指定的HTTP status code`status`。如果你没有指定`status`，status code默认为"302 Found"。

```
res.redirect('/foo/bar');
res.redirect('http://example.com');
res.redirect(301, 'http://example.com');
res.redirect('../login');
```

重定向也可以是完整的URL，来重定向到不同的站点。

```
res.redirect('http://google.com');
```

重定向也可以相对于主机的根路径。比如，如果程序的路径为`http://example.com/admin/post/new`，那么下面将重定向到`http://example.com/admim`：

```
res.redirect('/admin');
```

重定向也可以相对于当前的URL。比如，来之于`http://example.com/blog/admin/`(注意结尾的`/`)，下面将重定向到`http://example.com/blog/admin/post/new`。

```
res.redirect('post/new');
```

如果来至于`http://example.com/blog/admin`（没有尾部`/`），重定向`post/new`，将重定向到`http://example.com/blog/post/new`。如果你觉得上面很混乱，可以把路径段认为目录(有'/')或者文件，这样是可以的。相对路径的重定向也是可以的。如果你当前的路径为`http://example.com/admin/post/new`，下面的操作将重定向到`http://example.com/admin/post`：

```
res.redirect('..');
```

`back`将重定向请求到referer，当没有`referer`的时候，默认为`/`。

```
res.redirect('back');
```

res.render(view [, locals] [, callback])

渲染一个视图，然后将渲染得到的HTML文档发送给客户端。可选的参数为:

- `locals`，定义了视图本地参数属性的一个对象。
- `callback`，一个回调方法。如果提供了这个参数，`render`方法将返回错误和渲染之后的模板，并且不自动发送响应。当有错误发生时，可以在这个回调内部，调用`next(err)`方法。

> 本地变量缓存使能视图缓存。在开发环境中缓存视图，需要手动设置为true；视图缓存在生产环境中默认开启。

```
// send the rendered view to the client
res.render('index');

// if a callback is specified, the render HTML string has to be sent explicitly

res.render('index', function(err, html) {
    res.send(html);
});

// pass a local variable to  the view
res.render('user', {name:'Tobi'}, function(err, html) {
    // ...
});
```

res.send([body])

发送HTTP响应。`body`参数可以是一个`Buffer`对象，一个字符串，一个对象，或者一个数组。比如：

```
res.send(new Buffer('whoop'));
res.send({some:'json'});
res.send('<p>some html</p>');
res.status(404).send('Sorry, we cannot find that!');
res.status(500).send({ error: 'something blew up' });
```

对于一般的非流请求，这个方法可以执行许多有用的的任务：比如，它自动给`Content-Length`HTTP响应头赋值(除非先前定义)，也支持自动的HEAD和HTTP缓存更新。

当参数是一个`Buffer`对象，这个方法设置`Content-Type`响应头为`application/octet-stream`，除非事先提供，如下所示:

```
res.set('Content-Type', 'text/html');
res.send(new Buffer('<p>some html</p>'));
```

当参数是一个字符串，这个方法设置`Content-Type`响应头为`text/html`：

```
res.send('<p>some html</p>');
```

当参数是一个对象或者数组，Express使用JSON格式来表示：

```
res.send({user:'tobi'});
res.send([1, 2, 3]);
```

res.sendFile(path [, options] [, fn])

> `res.sendFile()`从`Express v4.8.0`开始支持。

传输`path`指定的文件。根据文件的扩展名设置`Content-Type`HTTP头部。除非在`options`中有关于`root`的设置，`path`一定是关于文件的绝对路径。 下面的表提供了`options`参数的细节:

属性	描述	默认值	可用版本
maxAge	设置`Cache-Control`的`max-age`属性，格式为毫秒数，或者是ms format的一串字符串	0	
root	相对文件名的根目录		
lastModified	设置`Last-Modified`头部为此文件在系统中的最后一次修改时间。设置`false`来禁用它	Enable	4.9.0+
headers	一个对象，包含了文件相关的HTTP头部。		
dotfiles	是否支持点开头文件名的选项。可选的值"allow","deny","ignore"	"ignore"	

当传输完成或者发生了什么错误，这个方法调用`fn`回调方法。如果这个回调参数指定了和一个错误发生，回调方法必须明确地通过结束请求-响应循环或者传递控制到下个路由来处理响应过程。 下面是使用了所有参数的使用`res.sendFile()`的例子：

```
app.get('/file/:name', function(req, res, next) {
```

```
    var options = {
        root:__dirname + '/public',
        dotfile:'deny',
        headers:{
            'x-timestamp':Date.now(),
            'x-sent':true
        }
    };

    var fileName = req.params.name;
    res.sendFile(fileName, options, function(err) {
        if (err) {
            console.log(err);
            res.status(err.status).end();
        }
        else {
            console.log('sent', fileName);
        }
    });



});
```

`res.sendFile` 提供了文件服务的细粒度支持，如下例子说明：

```
app.get('/user/:uid/photos/:file', function(req, res) {
    var uid = req.params.uid
        , file = req.params.file;


    req.user.mayViewFilesFrom(uid, function(yes) {
        if (yes) {
            res.sendFile('/upload/' + uid + '/' + file);
        }
        else {
            res.status(403).send('Sorry! you cant see that.');
        }
    });

})
```

获取更多信息，或者你有问题或者关注，可以查阅send。

res.sendStatus(statusCode)

设置响应对象的`HTTP status code`为`statusCode`并且发送`statusCode`的相应的字符串形式作为响应的Body。

```
    res.sendStatus(200); // equivalent to res.status(200).send('OK');
    res.sendStatus(403); // equivalent to res.status(403).send('Forbidden');
    res.sendStatus(404); // equivalent to res.status(404).send('Not Found');
    res.sendStatus(500); // equivalent to res.status(500).send('Internal Server Error')
```

如果一个不支持的状态被指定，这个HTTP status依然被设置为`statusCode`并且用这个code的字符串作为Body。

```
res.sendStatus(2000); // equivalent to res.status(2000).send('2000');
```

More about HTTP Status Codes

res.set(field [, value])

设置响应对象的HTTP头部`field`为`value`。为了一次设置多个值，那么可以传递一个对象为参数。

```
res.set('Content-Type', 'text/plain');

res.set({
    'Content-Type':'text/plain',
    'Content-Length':'123',
    'ETag':'123456'
})
```

其和`res.header(field [,value])`效果一致。

res.status(code)

使用这个方法来设置响应对象的HTTP status。其是Node中response.statusCode的一个连贯性的别名。

```
res.status(403).end();
res.status(400).send('Bad Request');
res.status(404).sendFile('/absolute/path/to/404.png');
```

res.type(type)

程序将设置`Content-Type`HTTP头部的MIME type，如果这个设置的`type`能够被mime.lookup解析成正确的`Content-Type`。如果`type`中包含了`/`字符，那么程序会直接设置`Content-Type`为`type`。

```
res.type('.html');              // => 'text/html'
res.type('html');               // => 'text/html'
res.type('json');               // => 'application/json'
res.type('application/json');   // => 'application/json'
res.type('png');                // => image/png:
```

res.vary(field)

在没有Vary应答头部时增加Vary应答头部。

> ps：vary的意义在于告诉代理服务器/缓存/CDN，如何判断请求是否一样，vary中的组合就是服务器/缓存/CDN判断的依据，比如Vary中有User-Agent，那么即使相同的请求，如果用户使用IE打开了一个页面，再用Firefox打开这个页面的时候，CDN/代理会认为是不同的页面，如果Vary中没有User-Agent，那么CDN/代理会认为是相同的页面，直接给用户返回缓存的页面，而不会再去web服务器请求相应的页面。通俗的说就相当于`field`作为了一个缓存的key来判断是否命中缓存

```
res.vary('User-Agent').render('docs');
```

Router

一个`router`对象是一个单独的实例关于中间件和路由。你可以认为其是一个"mini-application"（迷你程序），其具有操作中间件和路由方法的能力。每个`Express`程序有一个内建的app路由。 路由自身表现为一个中间件，所以你可以使用它作为`app.use()`方法的一个参数或者作为另一个路由的`use()`的参数。 顶层的`express`对象有一个`Router()`方法，你可以使用`Router()`来创建一个新的`router`对象。

Router([options])

如下，可以创建一个路由：

```
var router = express.Router([options]);
```

`options`参数可以指定路由的行为，其有下列选择：

属性	描述	默认值	可用性
caseSensitive	是否区分大小写	默认不启用。对待 `/Foo` 和 `/foo` 一样。	
mergeParams	保存父路由的 `res.params` 。如果父路由参数和子路由参数冲突，子路由参数优先。	false	4.5.0+
strict	使能严格路由。	默认不启用， `/foo` 和 `/foo/` 被路由一样对待处理	

你可以将 `router` 当作一个程序，可以在其上添加中间件和HTTP路由方法(例如 `get` ， `put` ， `post` 等等)。

```
// invoked for any requests passed to this router
router.use(function(req, res, next) {
  // .. some logic here .. like any other middleware
  next();
});

// will handle any request that ends in /events
// depends on where the router is "use()'d"
router.get('/events', function(req, res, next) {
  // ..
});
```

你可以在一个特别的根URL上挂载一个路由，这样你就以将你的各个路由放到不同的文件中或者甚至是mini的程序。

```
// only requests to /calendar/* will be sent to our "router"
app.use('/calendar', router);
```

Methods

router.all(path, [callback, ...] callback)

这个方法和 `router.METHOD()` 方法一样，除了这个方法会匹配所有的HTTP动作。 这个方法对想映射全局的逻辑处理到特殊的路径前缀或者任意匹配是十分有用的。比如，如果你放置下面所示的这个路由在其他路由的前面，那么其将要求从这个点开始的所有的路由进行验证操作和自动加载用户信息。记住，这些全局的逻辑操作，不需要结束请求响应周期： `loaduser` 可以执行一个任务，然后调用 `next()` 来将执行流程移交到随后的路由。

```
router.all('*', requireAuthentication, loadUser);
```

相等的形式:

```
router.all('*', requireAuthentication)
router.all('*', loadUser);
```

这是一个白名单全局功能的例子。这个例子很像前面的，不过其仅仅作用于以`/api`开头的路径:

```
router.all('/api/*', requireAuthentication);
```

router.METHOD(path, [callback, ...] callback)

`router.METHOD()`方法提供了路由方法在`Express`中，这里的`METHOD`是HTTP方法中的一个，比如`GET`， `PUT`， `POST`等等，但`router`中的METHOD是小写的。所以，实际的方法是`router.get()`， `router.put()`， `router.post()`等等。 你可以提供多个回调函数，它们的行为和中间件一样，除了这些回调可以通过调用`next('router')`来绕过剩余的路由回调。你可以使用这个机制来为一个路由设置一些前提条件，如果请求没有满足当前路由的处理条件，那么传递控制到随后的路由。 下面的片段可能说明了最简单的路由定义。Experss转换path字符串为正则表达式，用于内部匹配传入的请求。在匹配的时候，是不考虑`Query strings`，例如，"GET /"将匹配下面的路由，"GET /?name=tobi"也是一样的。

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

如果你对匹配的path有特殊的限制，你可以使用正则表达式，例如，下面的可以匹配"GET /commits/71dbb9c"和"GET /commits/71bb92..4c084f9"。

```
router.get(/^\/commits\/(\w+)(?:\.\.(\w+))?$/, function(req, res) {
    var from = req.params[0];
    var to = req.params[1];
    res.send('commit range ' + from + '..' + to);
});
```

router.param(name, callback)

给路由参数添加回调触发器，这里的`name`是参数名，`function`是回调方法。回调方法的参数按序是请求对象，响应对象，下个中间件，参数值和参数名。虽然`name`在技术上是可选的，但是自Express V4.11.0之后版本不推荐使用(见下面)。

> 不像`app.param()`， `router.param()`不接受一个数组作为路由参数。

例如，当`:user`出现在路由路径中，你可以映射用户加载的逻辑处理来自动提供`req.user`给这个路由，或者对输入的参数进行验证。

```
    router.param('user', function(req, res, next, id) {
        User.find(id, function(error, user) {
            if (err) {
                next(err);
            }
            else if (user){
                req.user = user;
            } else {
                next(new Error('failed to load user'));
            }
        });
    });
```

对于`Param`的回调定义的路由来说，他们是局部的。它们不会被挂载的app或者路由继承。所以，定义在`router`上的`param`回调只有是在`router`上的路由具有这个路由参数时才起作用。 在定义`param`的路由上，`param`回调都是第一个被调用的，它们在一个请求-响应循环中都会被调用一次并且只有一次，即使多个路由都匹配，如下面的例子：

```
router.param('id', function(req, res, next, id) {
    console.log('CALLED ONLY ONCE');
    next();
});

router.get('/user/:id', function(req, res, next) {
    console.log('although this matches');
    next();
});

router.get('/user/:id', function(req, res) {
    console.log('and this mathces too');
    res.end();
});
```

当`GET /user/42`，得到下面的结果:

```
    CALLED ONLY ONCE
    although this matches
    and this matches too
```

`

> 下面章节描述的`router.param(callback)`在v4.11.0之后被弃用。

通过只传递一个回调参数给`router.param(name, callback)`方法，`router.param(naem, callback)`方法的行为将被完全改变。这个回调参数是关于`router.param(name, callback)`该具有怎样的行为的一个自定义方法，这个方法必须接受两个参数并且返回一个中间件。 这个回调的第一个参数就是需要捕获的url的参数名，第二个参数可以是任一的JavaScript对象，其可能在实现返回一个中间件时被使用。 这个回调方法返回的中间件决定了当URL中包含这个参数时所采取的行为。
在下面的例子中，`router.param(name, callback)`参数签名被修改成了`router.param(name, accessId)`。替换接受一个参数名和回调，`router.param()`现在接受一个参数名和一个数字。

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

router.param(function(param, option){
    return function(req, res, next, val) {
        if (val == option) {
            next();
        }
        else {
            res.sendStatus(403);
        }
    }
});

router.param('id', 1337);

router.get('/user/:id', function(req, res) {
    res.send('Ok');
});

app.use(router);

app.listen(3000, function() {
    console.log('Ready');
});
```

在这个例子中，`router.param(name. callback)`参数签名保持和原来一样，但是替换成了一个中间件，定义了一个自定义的数据类型检测方法来检测`user id`的类型正确性。

```
    router.param(function(param, validator) {
        return function(req, res, next, val) {
            if (validator(val)) {
                next();
            }
            else {
                res.sendStatus(403);
            }
        }
    });

    router.param('id', function(candidate) {
        return !isNaN(parseFloat(candidate)) && isFinite(candidate);
    });
```

router.route(path)

返回一个单例模式的路由的实例，之后你可以在其上施加各种HTTP动作的中间件。使用`router.route()`来避免重复路由名字(例如错字错误)--说的意思应该是使用`router.route()`这个单例方法来避免同一个路径多个路由实例。

构建在上面的`router.param()`例子之上，下面的代码展示了怎么使用`router.route()`来指定各种HTTP方法的处理句柄。

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

router.param('user_id', function(req, res, next, id) {
    // sample user, would actually fetch from DB, etc...
    req.user = {
        id:id,
        name:"TJ"
    };
    next();
});

router.route('/users/:user_id')
    .all(function(req, res, next) {
        // runs for all HTTP verbs first
        // think of it as route specific middleware!
        next();
    })
    .get(function(req, res, next) {
        res.json(req.user);
```

```
    })
    .put(function(req, res, next) {
        // just an example of maybe updating the user
        req.user.name = req.params.name;
        // save user ... etc
        res.json(req.user);
    })
    .post(function(req, res, next) {
        next(new Error('not implemented'));
    })
    .delete(function(req, res, next) {
        next(new Error('not implemented'));
    })
```

这种方法重复使用单个`/usrs/:user_id`路径来添加了各种的HTTP方法。

router.use([path], [function, ...] function)

给可选的`path`参数指定的路径挂载给定的中间件方法，未指定`path`参数，默认值为`/`。 这个方法类似于`app.use()`。一个简单的例子和用例在下面描述。查阅app.use()获得更多的信息。 中间件就像一个水暖管道，请求在你定义的第一个中间件处开始，顺着中间件堆栈一路往下，如果路径匹配则处理这个请求。

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


// simple logger for this router`s requests
// all requests to this router will first hit this middleware

router.use(function(req, res, next) {
    console.log('%s %s %s', req.method, req.url, req.path);
    next();
})

// this will only be invoked if the path starts with /bar form the mount ponit
router.use('/bar', function(req, res, next) {
    // ... maybe some additional /bar logging ...
    next();
})

// always be invoked
router.use(function(req, res, next) {
```

```
    res.send('hello world');
})

app.use('/foo', router);

app.listen(3000);
```

对于中间件`function`，挂载的路径是被剥离的和不可见的。关于这个特性主要的影响是对于不同的路径，挂载相同的中间件可能对代码不做改动，尽管其前缀已经改变。 你使用`router.use()`定义中间件的顺序很重要。中间们是按序被调用的，所以顺序决定了中间件的优先级。例如，通常日志是你将使用的第一个中间件，以便每一个请求都被记录。

```
var logger = require('morgan');

router.use(logger());
router.use(express.static(__dirname + '/public'));
router.use(function(req, res) {
    res.send('Hello');
});
```

现在为了支持你不希望记录静态文件请求，但为了继续记录那些定义在`logger()`之后的路由和中间件。你可以简单的将`static()`移动到前面来解决：

```
router.use(express.static(__dirname + '/public'));
router.use(logger());
router.use(function(req, res){
  res.send('Hello');
});
```

另外一个确凿的例子是从不同的路径托管静态文件，你可以将`./public`放到前面来获得更高的优先级:

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

`router.use()`方法也支持命名参数，以便你的挂载点对于其他的路由而言，可以使用命名参数来进行预加载，这样做是很有益的。