Lecture 10
Express

什么是Express?

- Express是最流行的node web框架，它是许多其他流行的节点web框架的底层库。它提供了机制:
  - 在不同的URL路径(路由)中使用不同HTTP动词的请求编写处理程序。
  - 与“视图”呈现引擎集成，以便通过将数据插入模板来生成响应。
  - 设置常见的web应用程序设置，比如用于连接的端口，以及用于呈现响应的模板的位置。
  - 在请求处理管道的任何位置添加额外的请求处理“中间件”。
- 虽然Express本身是非常简单的，但是开发人员已经创建了兼容的中间件包来解决几乎所有的web开发问题。
  - cookie、会话、用户登录、URL参数、POST数据、安全标头等等。

特点

- 精简
- 灵活
- web程序框架
- 单页web程序
- 多页和混合的web程序

Is Express opinionated?

- Web框架通常将自己称为“固执己见”或“不固执己见”。

- 固执的框架认为应该有一套“标准答案”来解决各类具体任务。通常支持特定领域的快速开发（解决特定类型的问题）。因为标准答案通常易于理解且文档丰富。然而在解决主领域之外的问题时，就会显得不那么灵活，可用的组件和方法也更少。

- Express是不固执己见的,是高度包容的。
  - 几乎可以将任何您喜欢的任何兼容的中间件插入到请求处理链中。
  - 可以在一个文件或多个文件中构造该应用程序，并使用任何目录结构。

  - 有太多的选择！

Hello world

```
npm init
npm install express —save

app.js
var express = require('express');
var app = express();

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

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

应用生成器工具

- 通过应用生成器工具 express-generator 可以快速创建一个应用的骨架。
  - npm install express-generator -g
  - express helloworld
  - cd helloworld
  - npm install
  - # Run the helloworld on Windows
  - SET DEBUG=helloworld:* & npm start
  - # Run helloworld on Linux/Mac OS X
  - DEBUG=helloworld:* npm start

```
> helloworld@0.0.0 start /Users/liuhaitao/code/
 expressexample/helloworld
> node ./bin/www

  helloworld:server Listening on port 3000 +0ms
GET / 200 296.211 ms - 170
GET /stylesheets/style.css 200 4.926 ms - 111
GET /favicon.ico 404 18.522 ms - 1292
```

文件结构

app.js
bin
www
node_modules
package-lock.json
package.json
public
images
javascripts
stylesheets
style.css
routes
index.js
users.js
views
error.pug
index.pug
layout.pug

基本路由

- 路由用于确定应用程序如何响应对特定端点的客户机请求，包含一个 URI（或路径）和一个特定的 HTTP 请求方法（GET、POST 等）。
- 每个路由可以具有一个或多个处理程序函数，这些函数在路由匹配时执行。
- 路由定义采用以下结构：
  - app.METHOD(PATH, HANDLER)
  - 其中：
    - app 是 express 的实例。
    - METHOD 是 HTTP 请求方法。
    - PATH 是服务器上的路径。
    - HANDLER 是在路由匹配时执行的函数。

路由示例

- 以主页上的 Hello World! 进行响应：

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

- 在根路由 (/) 上（应用程序的主页）对 POST 请求进行响应：

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

- 对 /user 路由的 PUT 请求进行响应：

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

-

- app.js or routes/index.js

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

// respond with "hello world" when a GET request is made to the homepage
app.get('/', function(req, res) {
  res.send('hello world');
});
```

路由方法

- 路由方法派生自 HTTP 方法之一，附加到 express 类的实例。

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

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

特殊路由方法：app.all()

- 有一种特殊路由方法：app.all()，它并非派生自 HTTP 方法。该方法用于在所有请求方法的路径中装入中间件函数。
- 在以下示例中，无论使用 GET、POST、PUT、DELETE 还是在 http 模块中支持的其他任何 HTTP 请求方法，都将为针对“/secret”的请求执行处理程序。

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

路由路径

- 路由路径与请求方法相结合，用于定义可以在其中提出请求的端点。路由路径可以是字符串、字符串模式或正则表达式。
- Express 使用 path-to-regexp 来匹配路由路径；

```
此路由路径将请求与根路由 / 匹配。
app.get('/', function (req, res) {
  res.send('root');
});

此路由路径将请求与 /about 匹配。
app.get('/about', function (req, res) {
  res.send('about');
});

此路由路径将请求与 /random.text 匹配。
app.get('/random.text', function (req, res) {
  res.send('random.text');
});
基于字符串模式的路由路径的示例。此路由路径将匹配 acd 和 abcd。
app.get('/ab?cd', function(req, res) {
  res.send('ab?cd');
});
```

路由处理程序

- 可以提供多个回调函数，以类似于中间件的行为方式来处理请求。唯一例外是这些回调函数可能调用 next('route') 来绕过剩余的路由回调。可以使用此机制对路由施加先决条件，在没有理由继续执行当前路由的情况下，可将控制权传递给后续路由。
- 路由处理程序的形式可以是一个函数、一组函数或者两者的结合，如以下示例中所示。
- 单个回调函数可以处理一个路由。例如：
-

单个回调函数可以处理一个路由。例如：

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

多个回调函数可以处理一个路由（确保您指定 next 对象）。例如：

```
app.get('/example/b', function (req, res, next) {
  console.log('the response will be sent by the next function ...');
  next();
}, function (req, res) {
  res.send('Hello from B!');
});
```

一组回调函数可以处理一个路由。例如：

```
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]);
```

独立函数与一组函数的组合可以处理一个路由。例如：

```
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('the response will be sent by the next function ...');
  next();
}, function (req, res) {
  res.send('Hello from D!');
});
```

响应方法

- 下表中响应对象 (res) 的方法可以向客户机发送响应，并终止请求/响应循环。如果没有从路由处理程序调用其中任何方法，客户机请求将保持挂起状态。

方法	描述
res.download()	提示将要下载文件。
res.end()	结束响应进程。
res.json()	发送 JSON 响应。
res.jsonp()	在 JSONP 的支持下发送 JSON 响应。
res.redirect()	重定向请求。
res.render()	呈现视图模板。
res.send()	发送各种类型的响应。
res.sendFile	以八位元流形式发送文件。
res.sendStatus()	设置响应状态码并以响应主体形式发送其字符串表示。

app.route()

- 您可以使用 app.route() 为路由路径创建链式路由处理程序。因为在单一位置指定路径，所以可以减少冗余和输入错误。有关路由的更多信息，请参阅 Router() 文档。
- 以下是使用 app.route() 定义的链式路由处理程序的示例。
-

```
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

- 使用 express.Router 类来创建可安装的模块化路由处理程序。
- Router 实例是完整的中间件和路由系统；因此，常常将其称为“微型应用程序”。
- 以下示例将路由器创建为模块，在其中装入中间件，定义一些路由，然后安装在主应用程序的路径中。
- 在应用程序目录中创建名为 birds.js 的路由器文件，其中包含以下内容：
-

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

// middleware that is specific to this router
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});
// define the home page route
router.get('/', function(req, res) {
  res.send('Birds home page');
});
// define the about route
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

- 接着，在应用程序中装入路由器模块：
-

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

编写中间件

- Connect 创造了“中间件”(middleware)这个术语来描述插入式的 Node 模块
- 从概念上讲，中间件是一种功能的封装方式，具体来说就是封装在程序中处理 HTTP 请求 的功能。
- 中间件是在管道中执行的。
  - 在 Express 程序中，通过调用 app.use 向管道中插入中间件。

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

app.get('/', function(req, res, next) {
  next();
})

app.listen(3000);
```

中间件函数适用的 HTTP 方法。

中间件函数适用的路径（路由）。

中间件函数。

中间件函数的回调自变量，按约定称为“next”。

中间件函数的 HTTP 响应自变量，按约定称为“res”。

中间件函数的 HTTP 请求自变量，按约定称为“req”。

Express工作重点

- 路由处理器(app.get、app.post 等，经常被统称为 app.VERB)可以被看作只处理特定 HTTP 谓词(GET、POST 等)的中间件。同样，也可以将中间件看作可以处理全部 HTTP 谓词的路由处理器(基本上等同于 app.all，可以处理任何 HTTP 谓词;对于 PURGE 之 类特别的谓词会有细微的差别，但对于普通的谓词而言，效果是一样的)。
- 路由处理器的第一个参数必须是路径。如果你想让某个路由匹配所有路径，只需用 /*。 中间件也可以将路径作为第一个参数，但它是可选的(如果忽略这个参数，它会匹配所有路径，就像指定了 /* 一样)。
- 路由处理器和中间件的参数中都有回调函数，这个函数有 2 个、3 个或 4 个参数(从技 术上讲也可以有 0 或 1 个参数，但这些形式没有意义)。
  - 如果有 2 个或 3 个参数，头两 个参数是请求和响应对象，第三个参数是 next 函数。
  - 如果有 4 个参数，它就变成了错 误处理中间件，第一个参数变成了错误对象，然后依次是请求、响应和 next 对象。
- 如果不调用 next()，管道就会被终止，也不会再有处理器或中间件做后续处理。如果不调用 next()，则应该发送一个响应到客户端(res.send、res.json、res.render 等); 如果你不这样做，客户端会被挂起并最终导致超时。
- 如果调用了 next()，一般不宜再发送响应到客户端。如果你发送了，管道中后续的中间件或路由处理器还会执行，但它们发送的任何响应都会被忽略。

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

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

app.listen(3000);
```

中间件函数的简单示例

- 此函数仅在应用程序的请求通过它时显示“LOGGED”。中间件函数会分配给名为 myLogger 的变量。
- 请注意以上对 next() 的调用。
  - 调用此函数时，将调用应用程序中的下一个中间件函数。
  - next() 函数不是 Node.js 或 Express API 的一部分，而是传递给中间件函数的第三自变量。
  - next() 函数可以命名为任何名称，但是按约定，始终命名为“next”。
  - 为了避免混淆，请始终使用此约定。
-

```
var myLogger = function (req, res, next) {
  console.log('LOGGED');
  next();
};
```

- 要装入中间件函数，请调用 app.use() 并指定中间件函数。 例如，以下代码在根路径 (/) 的路由之前装入 myLogger 中间件函数。
-

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

var myLogger = function (req, res, next) {
  console.log('LOGGED');
  next();
};

app.use(myLogger);

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

app.listen(3000);
```

- 中间件装入顺序很重要
  - 首先装入的中间件函数也首先被执行。
- 如果在根路径的路由之后装入 myLogger，那么请求永远都不会到达该函数，应用程序也不会显示“LOGGED”，因为根路径的路由处理程序终止了请求/响应循环。
- 中间件函数 myLogger 只是显示消息，然后通过调用 next() 函数将请求传递到堆栈中的下一个中间件函数。

```
var requestTime = function (req, res, next) {
  req.requestTime = Date.now();
  next();
};
```

- 现在，该应用程序使用 requestTime 中间件函数。此外，根路径路由的回调函数使用由中间件函数添加到 req（请求对象）的属性。

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

var requestTime = function (req, res, next) {
  req.requestTime = Date.now();
  next();
};

app.use(requestTime);

app.get('/', function (req, res) {
  var responseText = 'Hello World!';
  responseText += 'Requested at: ' + req.requestTime + '';
  res.send(responseText);
});

app.listen(3000);
```

使用中间件

- Express 是一个路由和中间件 Web 框架，其自身只具有最低程度的功能：
  - Express 应用程序基本上是一系列中间件函数调用。
- 中间件函数能够访问请求对象 (req)、响应对象 (res) 以及应用程序的请求/响应循环中的下一个中间件函数。下一个中间件函数通常由名为 next 的变量来表示。
- 中间件函数可以执行以下任务：
  - 执行任何代码。
  - 对请求和响应对象进行更改。
  - 结束请求/响应循环。
  - 调用堆栈中的下一个中间件函数。
- 如果当前中间件函数没有结束请求/响应循环，那么它必须调用 next()，以将控制权传递给下一个中间件函数。否则，请求将保持挂起状态。

- Express 应用程序可以使用以下类型的中间件：
  - 应用层中间件
  - 路由器层中间件
  - 错误处理中间件
  - 内置中间件
  - 第三方中间件
- 可以使用可选安装路径来装入应用层和路由器层中间件。 还可以将一系列中间件函数一起装入，这样会在安装点创建中间件系统的子堆栈。

应用层中间件

- 使用 app.use() 和 app.METHOD() 函数将应用层中间件绑定到应用程序对象的实例，其中 METHOD 是中间件函数处理的请求的小写 HTTP 方法（例如 GET、PUT 或 POST）。

- 此示例显示没有安装路径的中间件函数。应用程序每次收到请求时执行该函数。

```
var app = express();

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

此示例显示安装在 /user/:id 路径中的中间件函数。在 /user/:id 路径中为任何类型的 HTTP 请求执行此函数。

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

此示例显示一个路由及其处理程序函数（中间件系统）。此函数处理针对 /user/:id 路径的 GET 请求。

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

在安装点使用安装路径装入一系列中间件函数的示例。 演示一个中间件子堆栈，用于显示针对 /user/:id 路径的任何类型 HTTP 请求的信息。

```
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();
});
```

- 路由处理程序可以为一个路径定义多个路由。以下示例为针对 /user/:id 路径的 GET 请求定义两个路由。第二个路由不会导致任何问题，但是永远都不会被调用，因为第一个路由结束了请求/响应循环。
- 此示例显示一个中间件子堆栈，用于处理针对 /user/:id 路径的 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');
});

// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', function (req, res, next) {
  res.end(req.params.id);
});
```

- 要跳过路由器中间件堆栈中剩余的中间件函数，请调用 next('route') 将控制权传递给下一个路由。
- next('route') 仅在使用 app.METHOD() 或 router.METHOD() 函数装入的中间件函数中有效。
- 此示例显示一个中间件子堆栈，用于处理针对 /user/:id 路径的 GET 请求。

```
app.get('/user/:id', function (req, res, next) {
  // if the user ID is 0, skip to the next route
  if (req.params.id == 0) next('route');
  // otherwise pass the control to the next middleware function in this stack
  else next(); //
}, function (req, res, next) {
  // render a regular page
  res.render('regular');
});

// handler for the /user/:id path, which renders a special page
app.get('/user/:id', function (req, res, next) {
  res.render('special');
});
```

路由器层中间件

- 路由器层中间件的工作方式与应用层中间件基本相同，差异之处在于它绑定到 express.Router() 的实例。
-

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

- 使用 router.use() 和 router.METHOD() 函数装入路由器层中间件。

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

// a middleware function with no mount path. This code is executed for every request to the router
router.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});

// a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path
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();
});

// a middleware sub-stack that handles GET requests to the /user/:id path
router.get('/user/:id', function (req, res, next) {
  // if the user ID is 0, skip to the next router
  if (req.params.id == 0) next('route');
  // otherwise pass control to the next middleware function in this stack
  else next(); //
}, function (req, res, next) {
  // render a regular page
  res.render('regular');
});

// handler for the /user/:id path, which renders a special page
router.get('/user/:id', function (req, res, next) {
  console.log(req.params.id);
  res.render('special');
});

// mount the router on the app
  app.use('/', router);
```

错误处理中间件

- 错误处理中间件始终采用四个自变量。必须提供四个自变量，以将函数标识为错误处理中间件函数。即使无需使用 next 对象，也必须指定该对象以保持特征符的有效性。否则，next 对象将被解释为常规中间件，从而无法处理错误。
- 错误处理中间件函数的定义方式与其他中间件函数基本相同，差别在于错误处理函数有四个自变量而不是三个，专门具有特征符 (err, req, res, next)：

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

内置中间件

- 自 V4.x 起，Express 不再依赖于 Connect。除 express.static 外，先前 Express 随附的所有中间件函数现在以单独模块的形式提供。请查看中间件函数的列表。
- express.static(root, [options])
  - Express 中唯一内置的中间件函数是 express.static。此函数基于 serve-static，负责提供 Express 应用程序的静态资源。
  - root 自变量指定从其中提供静态资源的根目录。
  - 可选的 options 对象可以具有以下属性：

属性	描述	类型	缺省值
dotfiles	是否对外输出文件名以点（.）开头的文件。有效值包括“allow”、“deny”和“ignore”	字符串	“ignore”
etag	启用或禁用 etag 生成	布尔	true
extensions	用于设置后备文件扩展名。	数组	[]
index	发送目录索引文件。设置为 false 可禁用建立目录索引。	混合	“index.html”
lastModified	将 Last-Modified 的头设置为操作系统上该文件的上次修改日期。有效值包括 true 或 false。	布尔	true
maxAge	设置 Cache-Control 头的 max-age 属性（以毫秒或者 ms 格式中的字符串为单位）	数字	0
redirect	当路径名是目录时重定向到结尾的“/”。	布尔	true
setHeaders	用于设置随文件一起提供的 HTTP 头的函数。	函数	

- 以下示例将使用了 express.static 中间件，并且提供了一个详细的’options’对象（作为示例）：

```
var options = {
  dotfiles: 'ignore',
  etag: false,
  extensions: ['htm', 'html'],
  index: false,
  maxAge: '1d',
  redirect: false,
  setHeaders: function (res, path, stat) {
    res.set('x-timestamp', Date.now());
  }
}

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

- 对于每个应用程序，可以有多个静态目录：
-

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

第三方中间件

- 使用第三方中间件向 Express 应用程序添加功能。
- 安装具有所需功能的 Node.js 模块，然后在应用层或路由器层的应用程序中将其加装入。
- 以下示例演示如何安装和装入 cookie 解析中间件函数 cookie-parser。
  - $ npm install cookie-parser
-

```
var express = require('express');
var app = express();
var cookieParser = require('cookie-parser');

// load the cookie-parsing middleware
app.use(cookieParser());
```

将模板引擎用于 Express

- 在 Express 可以呈现模板文件之前，必须设置以下应用程序设置：
  - views：模板文件所在目录。例如：app.set('views', './views')
  - view engine：要使用的模板引擎。例如：app.set('view engine', 'pug')
- 然后安装对应的模板引擎 npm 包：
  - $ npm install pug --save

在设置视图引擎之后，不必指定该引擎或者在应用程序中装入模板引擎模块；Express 在内部装入此模块，如下所示（针对以上示例）。

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

在 views 目录中创建名为 index.pug 的 Pug 模板文件，其中包含以下内容：

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

随后创建路由以呈现 index.pug 文件。如果未设置 view engine 属性，必须指定 view 文件的扩展名。否则，可以将其忽略。

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

向主页发出请求时，index.pug 文件将呈现为 HTML。

为 Express 开发模板引擎

- 可以使用 app.engine(ext, callback) 方法创建自己的模板引擎。
  - ext 表示文件扩展名，
  - callback 表示模板引擎函数，它接受以下项作为参数：文件位置、选项对象和回调函数。
- 以下代码示例实现非常简单的模板引擎以呈现 .ntl 文件。
-

```
var fs = require('fs'); // this engine requires the fs module
app.engine('ntl', function (filePath, options, callback) { // define the template engine
  fs.readFile(filePath, function (err, content) {
    if (err) return callback(new Error(err));
    // this is an extremely simple template engine
    var rendered = content.toString().replace('#title#', ''+ options.title +'')
    .replace('#message#', ''+ options.message +'');
    return callback(null, rendered);
  });
});
app.set('views', './views'); // specify the views directory
app.set('view engine', 'ntl'); // register the template engine
```

应用程序现在能够呈现 .ntl 文件。在 views 目录中创建名为 index.ntl 且包含以下内容的文件：

```
#title#
#message#
```

然后，在应用程序中创建以下路径：

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

您向主页发出请求时，index.ntl 将呈现为 HTML。

调试 Express

- Express 在内部使用调试模块来记录关于路由匹配、使用的中间件函数、应用程序模式以及请求/响应循环流程的信息。
- debug 就像是扩充版的 console.log，但是与 console.log 不同，不必注释掉生产代码中的 debug 日志。缺省情况下，日志记录功能已关闭，可以使用 DEBUG 环境变量有条件地开启日志记录。
- 要查看 Express 中使用的所有内部日志，在启动应用程序时，请将 DEBUG 环境变量设置为 express:*。
  - $ DEBUG=express:* node index.js
- 在 Windows 上，使用对应的命令。
  - > set DEBUG=express:* & node index.js

```
$ DEBUG=express:* node ./bin/www
 express:router:route new / +0ms
 express:router:layer new / +1ms
 express:router:route get / +1ms
 express:router:layer new / +0ms
 express:router:route new / +1ms
 express:router:layer new / +0ms
 express:router:route get / +0ms
 express:router:layer new / +0ms
 express:application compile etag weak +1ms
 express:application compile query parser extended
+0ms
 express:application compile trust proxy false +0ms
 express:application booting in development mode
+1ms
```

向应用程序发出请求时，可以看到 Express 代码中指定的日志：

```
express:router dispatching GET / +4h
express:router query  : / +2ms
express:router expressInit  : / +0ms
express:router favicon  : / +0ms
express:router logger  : / +1ms
express:router jsonParser  : / +0ms
express:router urlencodedParser  : / +1ms
express:router cookieParser  : / +0ms
express:router stylus  : / +0ms
express:router serveStatic  : / +2ms
express:router router  : / +2ms
express:router dispatching GET / +1ms
express:view lookup "index.pug" +338ms
express:view stat "/projects/example/views/
index.pug" +0ms
```

Cookie

- cookie 的想法很简单:服务器发送一点信息，浏览器在一段可配置的时期内保存它。
- 发送哪些信息确实是由服务器来决定:通常只是一个唯一 ID 号，标识特定浏览器，从而维持 一个有状态的假象。
-

关于 cookie

- cookie对用户来说不是加密的
  - 服务器向客户端发送的所有 cookie 都能被客户端查看。绝没有加密那样的安全性。
- 用户可以删除或禁用cookie
  - 用户对 cookie 有绝对的控制权，并且浏览器支持批量或单个删除 cookie。
  - 用户也可以禁用 cookie， 但这更容易造成问题，因为只有最简单的 Web 应用程序才不需要依赖 cookie。
- 一般的cookie可以被篡改
  - 要确 保 cookie 不被篡改，请使用签名 cookie。
- cookie可以用于攻击
  - 跨站脚本攻击(XSS)攻击中有一种技术就涉及用恶意的 JavaScript 修改 cookie 中的内容。所以不要轻易相信返回到你的服务器的 cookie 内容。用签名 cookie 会有帮助(不管是用户修改的还是恶意 JavaScript 修改的， 这些篡改都会在签名 cookie 中留下明显的痕迹)，并且还可以设定选项指明 cookie 只能 由服务器修改。这些 cookie 的用途会受限，但它们肯定更安全。
- 如果滥用cookie，用户会注意到
  - 如果在用户的电脑上设了很多 cookie，或者存了很多数据，这可能会惹恼用户，所以应该避免出现这种情况。尽量把对 cookie 的使用限制在最小范围内。
- 如果可以选择，会话要优于cookie
  - 大多数情况下，可以用会话维持状态，一般来说这样做是明智的。并且会话更容易， 你不用担心会滥用用户的存储，而且也更安全。当然，会话要依赖 cookie，但如果你使 用会话，Express 会帮你做很多工作。

Express中的Cookie

```
npm install --save cookie-parser
app.use(require('cookie-parser')(credentials.cookieSecret));

res.cookie('monster', 'nom nom');
res.cookie('signed_monster', 'nom nom', { signed: true });

var monster = req.cookies.monster;
 var signedMonster = req.signedCookies.monster;

res.clearCookie('monster');
```

会话

- cookie 恐慌
  - url 或者html5本地存储
- 两种实现会话的方法:
  - 把所有东西都存在 cookie 里
    - 被称为“基于 cookie 的会话”。
  - 只在 cookie 里 存一个唯一标识，其他东西都存在服务器上。

```
npm install --save express-session

app.use(require('cookie-parser')(credentials.cookieSecret));
app.use(require(‘express-session')());
```

使用会话

```
req.session.userName = 'Anonymous';
 var colorScheme = req.session.colorScheme || 'dark';
```

会话的用途

- 跨页保存用户的偏好
- 提供用户验证信息
  - 登录后就会创建一个会话。之后就不用在每次重新加载页面时再登录一次。
  - 即便没有用户账号，会话也有用。
    - 网站一般都要记住你喜欢如何排列东西，或者喜欢哪种日期格式，这些都不需要登录。
- 尽管建议优先选择会话而不是 cookie，但理解 cookie 的工作机制也很重要(特别是因为有cookie才能用会话)。它对于你在应用中诊断问题、理解安全性及隐私问题都有帮助。

参考

- http://expressjs.com/zh-cn/
- https://developer.mozilla.org/zh-CN/docs/Learn/Server-side/Express_Nodejs/Introduction

Thanks!!!