课堂练习的操作指导

目录

Backbone

实验目的

1. 理解前端框架的路由组件（`router`）的作用

操作步骤

1. 浏览器打开`demos/backbone-demo/index.html`
2. 点击页面上的链接，注意浏览器 URL 的变化
3. 仔细查看`js/main.js`的源码，看懂 Router 组件的使用方式

Angular

实验目的

1. 理解 Angular 的双向绑定机制

操作步骤

1. 浏览器打开`demos/angular-demo/index.html`
2. 在输入框填入内容，注意页面变化
3. 查看`index.html`的源码，理解 Angular 对 HTML 标签的增强

Vue

实验目的

1. 理解 Vue 的模板与数据的双向绑定

操作步骤

1. 浏览器打开`demos/vue-demo/index1.html`
2. 在输入框填入内容，注意页面变化
3. 查看`app1.js`，理解 Vue 组件的基本写法

注意事项

1. `index2.html` 是一个稍微复杂的例子，模板如何绑定数据对象的一个字段。
2. `index3.html` 是事件绑定模板的例子。

JSX

实验目的

1. 掌握 JSX 基本语法

操作步骤

1. 浏览器打开 `demos/jsx-demo/index.html` ，仔细查看源码。

注意事项

1. `ReactDOM.render` 方法接受两个参数：一个虚拟 DOM 节点和一个真实 DOM 节点，作用是将虚拟 DOM 挂载到真实 DOM。

练习

1. 修改源码，将显示文字变为 ”Hello React!“。

React 组件语法

实验目的

1. 掌握 React 组件的基本写法

操作步骤

1. 浏览器打开`demos/react-component-demo/index1.html`，仔细查看源码。

注意事项

1. `class MyTitle extends React.Component`是ES6语法，表示自定义一个`MyTitle`类，该类继承了基类`React.Component`的所有属性和方法。
2. 每个组件都必须有`render`方法，定义输出的样式。
3. `<MyTitle/>`表示生成一个组件类的实例，每个实例一定要有闭合标签，写成`<MyTilte></MyTitle>`也可。

React组件的参数

实验目的

1. 学会向React组件传参数

操作步骤

1. 浏览器打开`demos/react-component-demo/index2.html`，仔细查看源码。

注意事项

1. 组件内部通过`this.props`对象获取参数。

练习

1. 将组件的颜色，从红色（`red`）换成黄色（`yellow`）。

React组件的状态

实验目的

1. 学会通过状态变动，引发组件的重新渲染。

操作步骤

1. 浏览器打开`demos/react-component-demo/index3.html`，仔细查看源码。

注意事项

```
  class MyTitle extends React.Component {
    constructor(...args) {
      super(...args);
      this.state = {
        name: '访问者'
      };
    }
    // ...
```

`constructor`是组件的构造函数，会在创建实例时自动调用。`...args`表示组件参数，`super(...args)`是 ES6 规定的写法。`this.state`对象用来存放内部状态，这里是定义初始状态。

```
<div>
  <input
    type="text"
    onChange={this.handleChange.bind(this)}
  />
  <p>你好, {this.state.name}</p>
</div>;
```

`this.state.name`表示读取`this.state`的`name`属性。每当输入框有变动，就会自动调用`onChange`指定的监听函数，这里是`this.handleChange`，`.bind(this)`表示该方法内部的`this`，绑定当前组件。

```
handleChange(e) {
  let name = e.target.value;
  this.setState({
    name: name
```

```
    });
  }
```

`this.setState`方法用来重置`this.state`，每次调用这个方法，就会引发组件的重新渲染。

React 组件实战

实验目的

1. 学会自己写简单的 React 组件。

操作步骤

1. 浏览器打开`demos/react-component-demo/index4.html`。
2. 点击`Hello World`，看看会发生什么。

练习

1. 修改源码，使得点击`Hello World`后，会显示当前的日期，比如`Hello 2016年1月1日`。
2. 请在上一步练习的基础上，进一步修改。现在`Hello World`点击一次，会改变内容，再点击就不会有反应了。请将其改成，再点击一次变回原样。

提示

练习一、下面的代码可以得到当前日期。

```
var d = new Date();
d.getFullYear() // 当前年份
d.getMonth() + 1 // 当前月份
d.getDate() // 当前是每个月的几号
```

练习二、可以在`this.state`里面设置一个开关变量`isClicked`。

```
this.state = {
  text: 'World',
  isClicked: false
};
```

然后，在`this.handleClick`方法里面，做一个`toggle`效果。

```
let isClicked = !this.state.isClicked;
this.setState({
  isClicked: isClicked,
  text: isClicked ? 'Clicked' : 'World'
});
```

React 组件的生命周期

实验目的

1. 掌握钩子方法的基本用法
2. 掌握组件如何通过 Ajax 请求获取数据，并对数据进行处理

操作步骤

1. 打开`demos/react-lifecycle-demo/index.html`，仔细查看源码。

注意事项

```
componentDidMount() {
  const url = '...';
  $.getJSON(url)
    .done()
    .fail();
}
```

- `componentDidMount`方法在组件加载后执行，只执行一次。本例在这个方法里向服务器请求数据，操作结束前，组件都显示`Loading`。

- `$.getJSON`方法用于向服务器请求 JSON 数据。本例的数据从 Github API 获取，可以打开源码里面的链接，看看原始的数据结构。

练习

1. 本例的 JSON 数据是 Github 上面最受欢迎的 JavaScript 项目。请在网页上显示一个列表，列出这些项目。

提示

（1） `this.state.loading`记录数据加载是否结束。只要数据请求没有结束，`this.state.loading`就一直是`true`，网页上显示`loading`。

（2） `this.state.error`保存数据请求失败时的错误信息。如果请求失败，`this.state.error`就是返回的错误对象，网页上显示报错信息。

（3） `this.state.data`保存从服务器获取的数据。如果请求成功，可以先用`console.log`方法，将它在控制台里打印出来，看看数据结构。

```
render() {
  // 加一行打印命令，看看数据结构
  console.log(this.state.data);
  return {
  // ...
```

（4） `this.state.data`里面的`this.state.data.items`应该是一个数组，保存着每个项目的具体信息。可以使用`forEach`方法进行遍历处理。

```
var projects = this.state.data.items;
var results = [];
projects.forEach(p => {
  var item = <li>{p.name}</li>;
    results.push(item);
});
```

（5）然后，将上一步的`results`插入网页即可。

```
<div>
  <ul>{results}</ul>
</div>
```

ReCharts

实验目的

1. 了解如何使用第三方组件库。

操作步骤

1. 浏览器打开`demos/recharts-demo/index.html`，查看效果。

MobX

实验目的

1. 理解 MobX 框架

操作步骤

（1） 命令行进入`demos/mobx-demo/`目录，执行如下的命令。

```
$ npm install
$ npm start
```

（2） 打开浏览器，访问 http://localhost:8080 ，查看结果，并仔细研究代码。

注意事项

```
@observer
class App extends React.Component {
  render() {
    // ...
```

```
  }
}
```

`@observer`是一种新的语法，表示对整个类执行指定的函数。

数据保存在`Store`里面。`Store`的属性分成两种：被观察的属性（`@observable`），和自动计算得到的属性`@computed`。

```
class Store {
  @observable name = 'Bartek';
  @computed get decorated() {
    return `${this.name} is awesome!`;
  }
}
```

`Store`的变化由用户引发。组件观察到`Store`的变化，自动重新渲染。

```
<p>
  {this.props.store.decorated}
</p>
<input
  defaultValue={this.props.store.name}
  onChange={
    (event) =>
      this.props.store.name = event.currentTarget.value
  }
/>
```

Redux

实验目的

1. 理解 Redux 架构

操作步骤

（1） 命令行下进入`demos/redux-demo`目录，执行如下的命令。

```
$ npm install
$ npm start
```

（2）打开浏览器，访问 http://localhost:8080，查看结果，并仔细研究代码。

注意事项

（1） Redux 要求 UI 的渲染组件都是纯组件，即不包含任何状态（`this.state`）的组件。

```
<div className="index">
  <p>{this.props.text}</p>
  <input
    defaultValue={this.props.name}
    onChange={this.props.onChange}
  />
</div>
```

（2） 进行数据处理、并包含状态的组件，称为”容器组件“。Redux 使用`connect`方法，自动生成 UI 组件对应的”容器组件“。

```
// MyComponent 是纯的 UI 组件
const App = connect(
  mapStateToProps,
  mapDispatchToProps
)(MyComponent);
```

（3） `mapStateToProps`函数返回一个对象，表示一种映射关系，将 UI 组件的参数映射到`state`。

```
function mapStateToProps(state) {
  return {
    text: state.text,
    name: state.name
  };
}
```

（4） `mapDispatchToProps`函数也是返回一个对象，表示一种映射关系，但定义的是哪些用户的操作应该当作`Action`，传给`Store`。

```
function mapDispatchToProps(dispatch) {
  return {
    onChange: (e) => dispatch({
      type: 'change',
      payload: e.target.value
    })
  }
}
```

（5） `reducer`函数用来接收`action`，算出新的`state`。

```
function reducer(state = {
  text: '你好，访问者',
  name: '访问者'
}, action) {
  switch (action.type) {
    case 'change':
      return {
        name: action.payload,
        text: '你好，' + action.payload
      };
  }
}
```

`Store`由 Redux 提供的`createStore`方法生成，该方法接受`reducer`作为参数。

```
const store = createStore(reducer);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.body.appendChild(document.createElement('div'))
);
```

为了把`Store`传入组件，必须使用 Redux 提供的`Provider`组件在应用的最外面，包裹一层。

Simple App

实验目的

1. 学会使用 Node 编写简单的前端应用。

操作步骤

（1）新建一个目录

```
$ mkdir simple-app-demo
$ cd simple-app-demo
```

（2）在该目录下，新建一个`package.json`文件。

```
$ npm init -y
```

`package.json`是项目的配置文件。

（3）安装`jquery`和`webpack`这两个模块。

```
$ npm install -S jquery
$ npm install -S webpack
```

打开`package.json`文件，会发现`jquery`和`webpack`都加入了`dependencies`字段，并且带有版本号。

（4）在项目根目录下，新建一个网页文件`index.html`。

```
<html>
  <body>
    <h1>Hello World</h1>
    <script src="bundle.js"></script>
  </body>
</html>
```

（5）在项目根目录下，新建一个脚本文件`app.js`。

```
const $ = require('jquery');
$('h1').css({ color: 'red'});
```

上面代码中，`require`方法是 Node 特有的模块加载命令。

（6）打开`package.json`，在`scripts`字段里面，添加一行。

```
"scripts": {
  "build": "webpack app.js bundle.js",
  "test": "...."
},
```

（7） 在项目根目录下，执行下面的命令，将脚本打包。

```
$ npm run build
```

执行完成，可以发现项目根目录下，新生成了一个文件`bundle.js`。

（8）浏览器打开`index.html`，可以发现`Hello World`变成了红色。

练习

1. 修改样式，将标题变为蓝色，然后重新编译生成打包文件。

REST API

实验目的

1. 熟悉 REST API 的基本用法

操作步骤

（1） 命令行进入`rest-api-demo`目录，执行下面的命令。

```
$ npm install -S json-server
```

（2） 在项目根目录下，新建一个 JSON 文件`db.json`。

```
{
  "posts": [
    { "id": 1, "title": "json-server", "author": "typicode" }
  ],
```

```
  "comments": [
    { "id": 1, "body": "some comment", "postId": 1 }
  ],
  "profile": { "name": "typicode" }
}
```

（3）打开`package.json`，在`scripts`字段添加一行。

```
"scripts": {
  "server": "json-server db.json",
  "test": "..."
},
```

（4）命令行下执行下面的命令，启动服务。

```
$ npm run server
```

（5）打开Chrome浏览器的Postman应用。依次向`http://127.0.0.1:3000/posts`、`http://127.0.0.1:3000/posts/1`发出`GET`请求，查看结果。

（6）向`http://127.0.0.1:3000/comments`发出`POST`请求。注意，数据体`Body`要选择`x-www-form-urlencoded`编码，然后依次添加下面两个字段。

```
body: "hello world"
postId: 1
```

发出该请求后，再向`http://127.0.0.1:3000/comments`发出`GET`请求，查看结果。

（7）向`http://127.0.0.1:3000/comments/2`发出`PUT`请求，数据体`Body`要选择`x-www-form-urlencoded`编码，然后添加下面的字段。

```
body: "hello react"
```

发出该请求后，再向`http://127.0.0.1:3000/comments`发出`GET`请求，查看结果。

（8）向`http://127.0.0.1:3000/comments/2`发出`delete`请求。

发出该请求后，再向`http://127.0.0.1:3000/comments`发出`GET`请求，查看结果。

Express

实验目的

1. 学会 Express 搭建 Web 应用的基本用法。

操作步骤

（1）进入`demos/express-demo`目录，命令行执行下面的命令，安装依赖。

```
$ cd demos/express-demo
$ npm install
```

（2）打开`app1.js`，尝试看懂这个脚本。

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

上面代码调用`express`，生成一个 Web 应用的实例。

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

router.get('/', function(req, res) {
  res.send('<h1>Hello World</h1>');
});

app.use('/home', router);
```

上面代码新建了一个路由对象，该对象指定访问根路由（`/`）时，返回`Hello World`。然后，将该路由加载在`/home`路径，也就是说，访问`/home`会返回`Hello World`。

`router.get`方法的第二个参数是一个回调函数，当符合指定路由的请求进来，会被这个函数处理。该函数的两个参数，`req`和`res`都是Express 内置的对象，分别表示用户的请求和 Web 服务器的回应。`res.send`方法就表示服务器回应所送出的内容。

```
var port = process.env.PORT || 8080;

app.listen(port);
console.log('Magic happens on port ' + port);
```

上面代码指定了外部访问的端口，如果环境变量没有指定，则端口默认为`8080`。最后两行是启动应用，并输出一行提示文字。

（3）在命令行下，启动这个应用。

```
$ node app1.js
```

浏览器访问`localhost:8080/home`，看看是否输出`Hello World`。

然后，命令行下按 Ctrl + C，退出这个进程。

（4）打开`app2.js`，查看新增的那个路由。

```
router.get('/:name', function(req, res) {
  res.send('<h1>Hello ' + req.params.name + '</h1>');
});
```

上面代码新增了一个路由，这个路由的路径是一个命名参数`:name`，可以从`req.params.name`拿到这个传入的参数。

在命令行下，启动这个应用。

```
$ node app2.js
```

浏览器访问`localhost:8080/home/张三`，看看是否输出`Hello 张三`。

然后，命令行下按 Ctrl + C，退出这个进程。

（5）打开`app3.js`，先查看页面头部新增的两行代码。

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

// 新增代码...
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));

// ...
```

上面代码中，`body-parser`模块的作用，是对`POST`、`PUT`、`DELETE`等 HTTP 方法的数据体进行解析。`app.use`用来将这个模块加载到当前应用。有了这两句，就可以处理`POST`、`PUT`、`DELETE`等请求了。

下面查看新增的那个路由。

```
router.post('/', function (req, res) {
  var name = req.body.name;
```

```
  res.json({message: 'Hello ' + name});
});
```

上面代码表示，如果收到了`/`路径（实际上是`/home`路径）的`POST`请求，先从数据体拿到`name`字段，然后返回一段 JSON 信息。

在命令行下，启动这个应用。

```
$ node app3.js
```

然后，在 Chrome 浏览器的 Postman 插件里面，向`http://127.0.0.1:8080/home`发出一个`POST`请求。数据体的编码方法设为`x-www-form-urlencoded`，里面设置一个`name`字段，值可以随便取，假定设为`Alice`。也就是说，发出这样一个请求。

```
POST /home HTTP/1.1
Host: 127.0.0.1:8080
Content-Type: application/x-www-form-urlencoded

name=Alice
```

如果一切正常，服务器会返回一段 JSON 信息。

```
{
  "message": "Hello Alice"
}
```

（6）打开`app4.js`，查看在所有路由之前新增的那个函数。

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

// 新增的代码
router.use(function(req, res, next) {
  console.log('Thers is a requesting.');
  next();
});

router.get('/', function(req, res) {
  // ...
```

`router.use`的作用是加载一个函数。这个函数被称为中间件，作用是在请求被路由匹配之前，先进行一些处理。上面这个中间件起到 logging 的作用，每收到一个请求，就在命令行输出一条记录。请特别注意，这个函数内部的`next()`，它代表下一个中间件，表示将处理过的请求传递给下一个中间件。这个例子只有一个中间件，就进入路由匹配处理

（实际上，`bodyparser`、`router`本质都是中间件，整个 Express 的设计哲学就是不断对 HTTP 请求加工，然后返回一个 HTTP 回应）。

练习

1. URL 的查询字符串，比如`localhost:8080?name=Alice`里面的`name`，可以用`req.query.name`拿到。请修改一个路由，使之可以收到查询字符串，然后输出`'Hello ' + req.query.name`。

ESLint

实验目的

1. 学会使用 ESLint 进行代码检查。

操作步骤

（1）进入`demos/eslint-demo`目录，安装 ESLint。

```
$ cd demos/eslint-demo
$ npm install eslint --save-dev
```

（2）通常，我们会使用别人已经写好的代码检查规则，这里使用的是 Airbnb 公司的规则。所以，还要安装 ESLint 这个规则模块。

```
$ npm install eslint-plugin-import eslint-config-airbnb-base --save-
dev
```

上面代码中，`eslint-plugin-import`是运行这个规则集必须的，所以也要一起安装。

（3）ESLint 的配置文件是`.eslintrc.json`，放置在项目的根目录下面。新建这个文件，在里面指定使用 Airbnb 的规则。

```
{
  "extends": "airbnb-base"
}
```

（4）打开项目的`package.json`，在`scripts`字段里面添加三个脚本。

```
{
  // ...
  "scripts" : {
    "test": "echo \"Error: no test specified\" && exit 1",
    "lint": "eslint **/*.js",
    "lint-html": "eslint **/*.js -f html -o ./reports/lint-
results.html",
    "lint-fix": "eslint --fix **/*.js"
  },
  // ...
}
```

除了原有的`test`脚本，上面代码新定义了三个脚本，它们的作用如下。

- `lint`：检查所有`js`文件的代码
- `lint-html`：将检查结果写入一个网页文件`./reports/lint-results.html`
- `lint-fix`：自动修正某些不规范的代码

（5）运行静态检查命令。

```
$ npm run lint

  1:5  error    Unexpected var, use let or const instead  no-var
  2:5  warning  Unexpected console statement              no-console

✖ 2 problems (1 error, 1 warning)
```

正常情况下，该命令会从`index.js`脚本里面，检查出来两个错误：一个是不应该使用`var`命令，另一个是不应该在生产环境使用`console.log`方法。

（6）修正错误。

```
$ npm run lint-fix
```

运行上面的命令以后，再查看`index.js`，可以看到`var x = 1;`被自动改成了`const x = 1;`。这样就消除了一个错误，但是还留下一个错误。

（7）修改规则。

由于我们想要允许使用`console.log`方法，因此可以修改`.eslintrc.json`，改变`no-console`规则。请将`.eslintrc.json`改成下面的样子。

```
{
  "extends": "airbnb-base",

  "rules": {
    "no-console": "off"
  }
}
```

再运行`npm run lint`，就不会报错了。

```
$ npm run lint
```

Mocha

实验目的

1. 学会使用 Mocha 进行单元测试。

操作步骤

（1）进入`demos/mocha-demo`目录，安装 Mocha 和 Chai。

```
$ cd demos/mocha-demo
$ npm install -D mocha
$ npm install -D chai
```

（2）打开`add.js`文件，查看源码，我们要测试的就是这个脚本。

```
function add(x, y) {
  return x + y;
}

module.exports = add;
```

（3）编写一个测试脚本`add.test.js`。

```
var add = require('./add.js');
var expect = require('chai').expect;

describe('加法函数的测试', function() {
  it('1 加 1 应该等于 2', function() {
    expect(add(1, 1)).to.be.equal(2);
  });
});
```

测试脚本与所要测试的源码脚本同名，但是后缀名为`.test.js`（表示测试）或者`.spec.js`（表示规格）。比如，`add.js`的测试脚本名字就是`add.test.js`。

测试脚本里面应该包括一个或多个`describe`块，每个`describe`块应该包括一个或多个`it`块。

`describe`块称为"测试套件"（test suite），表示一组相关的测试。它是一个函数，第一个参数是测试套件的名称（"加法函数的测试"），第二个参数是一个实际执行的函数。

`it`块称为"测试用例"（test case），表示一个单独的测试，是测试的最小单位。它也是一个函数，第一个参数是测试用例的名称（"1 加 1 应该等于 2"），第二个参数是一个实际执行的函数。

上面的测试脚本里面，有一句断言。

```
expect(add(1, 1)).to.be.equal(2);
```

所谓"断言"，就是判断源码的实际执行结果与预期结果是否一致，如果不一致就抛出一个错误。上面这句断言的意思是，调用`add(1, 1)`，结果应该等于`2`。

所有的测试用例（`it`块）都应该含有一句或多句的断言。它是编写测试用例的关键。断言功能由断言库来实现，Mocha本身不带断言库，所以必须先引入断言库。

```
var expect = require('chai').expect;
```

断言库有很多种，Mocha并不限制使用哪一种。上面代码引入的断言库是`chai`，并且指定使用它的`expect`断言风格。

（4）打开`package.json`文件，改写`scripts`字段的`test`脚本。

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},

// 改成
```

```
  "scripts": {
    "test": "mocha *.test.js"
  },
```

（5）命令行下，执行下面的命令，运行测试用例。

```
$ npm test
```

正常情况下，命令行会有提示，表示测试用例已经通过了。

练习

1. 请在`add.test.js`里面添加一个测试用例，测试`3`加上`-3`，`add`函数应该返回`0`。

Nightmare

实验目的

1. 学会使用 Nightmare 完成功能测试。

操作步骤

（1）进入`./demos/nightmare-demo`目录，安装依赖。

```
$ cd demos/nightmare-demo

# Linux & Mac
$ env ELECTRON_MIRROR=https://npm.taobao.org/mirrors/electron/ npm
install

# Windows
$ set ELECTRON_MIRROR=https://npm.taobao.org/mirrors/electron/
$ npm install
```

注意，Nightmare 会先安装 Electron，而 Electron 的安装需要下载境外的包，有时会连不上，导致安装失败。所以，这里先设置了环境变量，指定使用国内的 Electron 源，然后

才执行安装命令。

（2）查看一下浏览器自动化脚本`taobao.test.js`。

```
var Nightmare = require('nightmare');
var nightmare = Nightmare({ show: true });
```

上面代码表示新建一个 Nightmare 实例，并且运行功能中，自动打开浏览器窗口。

```
nightmare
  .goto('https://www.taobao.com/')
  .type('#q', '电视机')
  .click('form[action*="/search"] [type=submit]')
  .wait('#mainsrp-itemlist')
  .evaluate(function () {
    return document.querySelector('#mainsrp-itemlist .item .ctx-box
a.J_ClickStat')
      .textContent.trim();
  })
  .end()
```

上面代码表示，打开淘宝首页，在搜索框键入`电视机`，点击”搜索“按钮，等待`#mainsrp-itemlist`元素出现，在页面内注入（`evaluate`）代码，将执行结果返回。

```
  .then(function (result) {
    console.log(result);
  })
  .catch(function (error) {
    console.error('Search failed:', error);
  });
```

Nightmare 会返回一个 Promise 对象，`then`方法指定操作成功的回调函数，`catch`方法指定操作失败的回调函数。

（3）命令行下运行这个示例脚本。

```
$ node taobao.test.js
```

正常情况下，运行结束后，命令行会显示淘宝”电视机“搜索结果的第一项。

（4）浏览器打开`index.html`文件，这是 React 练习时做过的一个例子，点击`Hello World`，标题会变成`Hello Clicked`。我们就要编写测试脚本，测试这个功能。

（5）打开测试脚本`test.js`。

```
var Nightmare = require('nightmare');
var expect = require('chai').expect;
var fork = require('child_process').fork;

describe('test index.html', function() {
  var child;

  before(function (done) {
    child = fork('./server.js');
    child.on('message', function (msg) {
      if (msg === 'listening') {
        done();
      }
    });
  });

  after(function () {
    child.kill();
  });
```

上面代码中，`before`和`after`是 Mocha 提供的两个钩子方法，分别在所有测试开始前和结束后运行。这里，我们在`before`方法里面，新建一个子进程，用来启动 HTTP 服务器；测试结束后，再杀掉这个子进程。

注意，`before`方法的参数是一个函数，它接受`done`作为参数。`done`是 Mocha 提供的一个函数，用来表示异步操作完成。如果不调用`done`，Mocha 就会认为异步操作没有结束，一直停在这一步，不往下执行，从而导致超时错误。

子进程脚本`server.js`的代码非常简单，只有四行。

```
var httpServer = require('http-server');
var server = httpServer.createServer();
server.listen(8080);
process.send('listening');
```

上面代码中，我们在`8080`端口启动 HTTP 服务器，然后向父进程发消息，表示启动完成。

（6）真正的自动化测试脚本如下。

```
  it('点击后标题改变', function(done) {
    var nightmare = Nightmare({ show: true });
    nightmare
      .goto('http://127.0.0.1:8080/index.html')
      .click('h1')
      .wait(1000)
      .evaluate(function () {
```

```
          return document.querySelector('h1').textContent;
        })
        .end()
        .then(function(text) {
          expect(text).to.equal('Hello Clicked');
          done();
        })
    });
```

上面代码中，首先打开网页，点击`h1`元素，然后等待 1 秒钟，注入脚本获取`h1`元素的文本内容。接着，在`then`方法里面，做一个断言，判断获取的文本是否正确。

（7）运行这个测试脚本。

```
$ npm test
```

如果一切正常，命令行下会显示测试通过。

练习

1. 你可以改变`h1`的背景色或前景色，然后编写测试用例，验证浏览器最后渲染的颜色是否正确。（提示：可以使用`Window.getComputedStyle()`方法，获取元素的最终颜色。）

Travis CI

实验目的

1. 了解持续集成的做法，学会使用 Travis CI。

操作步骤

（1）注册 Github 的账户。如果你已经注册过，跳过这一步。

（2）访问这个代码库`github.com/ruanyf/travis-ci-demo`，点击右上角的`Fork`按钮，将它克隆到你自己的空间里面。

（3）将你`fork`的代码库，克隆到本地。

```
$ git clone git@github.com:[your_username]/travis-ci-demo.git
```

（4）使用你的 Github 账户，登录 Travis CI 的首页。然后，访问 Profile 页面，选定 `travis-ci-demo` 代码库运行自动构建。

（5）回到命令行，进入你本地的 `travis-ci-demo` 目录，切换到 `demo01` 分支。

```
$ cd travis-ci-demo
$ git checkout demo01
```

项目根目录下面有一个 `.travis.yml` 文件，这是 Travis CI 的配置文件。如果没有这个文件，就不会触发 Travis CI 的自动构建。打开看一下。

```
language: node_js
node_js:
  - "node"
```

上面代码指定，使用 Node 完成构建，版本是最新的稳定版。

指定 Node 的版本号也是可以的。

```
language: node_js
node_js:
  - "4.1"
```

上面代码指定使用 Node 4.1 版。

（6）Travis CI 默认依次执行以下九个脚本。

- `before_install`
- `install`
- `before_script`
- `script`
- `after_success` 或者 `after_failure`
- `after_script`
- `before_deploy`（可选）
- `deploy`（可选）
- `after_deploy`（可选）

用户需要用到哪个脚本，就需要提供该脚本的内容。

对于 Node 项目，以下两个脚本有默认值，可以不用自己设定。

```
"install": "npm install",
"script": "npm test"
```

（7）打开当前分支的`package.json`，可以发现它的`test`脚本是一个`lint`命令。

```
"scripts": {
  "test": "jshint hello.js"
},
```

（8）在项目根目录下，新建一个新文件`NewUser.txt`，内容是你的用户名。提交这个文件，就会触发 Travis CI 的自动构建。

```
$ git add -A
$ git commit -m 'Testing Travis CI'
$ git push
```

（9）等到 Travis CI 完成自动构建，到页面上检查构建结果。

（10）切换到`demo02`分支，打开`package.json`，可以看到`test`脚本，现在需要完成两步操作了。

```
  "scripts": {
    "lint": "jshint hello.js hello.test.js",
    "test": "npm run lint && mocha hello.test.js"
  },
```

（11）重复上面第 8 步和第 9 步。

练习

1. 修改`hello.js`，让其输出`Hello Node`。并修改测试用例`hello.test.js`，使之能够通过 Travis CI 的自动构建。