文 轻松入门React和Webpack

react.js webpack **天镶** 2015年05月15日发布

说说React

一个组件，有自己的结构，有自己的逻辑，有自己的样式，会依赖一些资源，会依赖某些其他组件。比如日常写一个组件，比较常规的方式：

- 通过前端模板引擎定义结构
- JS文件中写自己的逻辑
- CSS中写组件的样式
- 通过RequireJS、SeaJS这样的库来解决模块之间的相互依赖

那么在React中是什么样子呢？

结构和逻辑

在React的世界里，结构和逻辑交由JSX文件组织，React将模板内嵌到逻辑内部，实现了一个JS代码和HTML混合的JSX。

结构

在JSX文件中，可以直接通过`React.createClass`来定义组件：

```
var CustomComponent = React.creatClass({
    render: function(){
        return (<div className="custom-component"></div>);
    }
});
```

通过这种方式可以很方便的定义一个组件，组件的结构定义在render函数中，但这并不是简单的模板引擎，我们可以通过js方便、直观的操控组件结构，比如我想给组件增加几个节点：

```
var CustomComponent = React.creatClass({
    render: function(){
        var $nodes = ['h','e','l','l','o'].map(function(str){
            return (<span>{str}</span>);
        });
        return (<div className="custom-component">{$nodes}</div>);
    }
});
```

通过这种方式，React使得组件拥有灵活的结构。那么React又是如何处理逻辑的呢？

逻辑

写过前端组件的人都知道，组件通常首先需要相应自身DOM事件，做一些处理。必要时候还需要暴露一些外部接口，那么React组件要怎么做到这两点呢？

事件响应

比如我有个按钮组件，点击之后需要做一些处理逻辑，那么React组件大致上长这样：

```
var ButtonComponent = React.createClass({
    render: function(){
        return (<button>屠龙宝刀，点击就送</button>);
    }
});
```

点击按钮应当触发相应地逻辑，一种比较直观的方式就是给button绑定一个`onclick`事件，里面就是需要执行的逻辑了：

```
function getDragonKillingSword() {
    //送宝刀
}
var ButtonComponent = React.createClass({
    render: function(){
        return (<button onclick="getDragonKillingSword()">屠龙宝刀，点击就送</button>);
    }
});
```

但事实上`getDragonKillingSword()`的逻辑属于组件内部行为，显然应当包装在组件内部，于是在React中就可以这么写：

```
var ButtonComponent = React.createClass({
    getDragonKillingSword: function(){
        //送宝刀
    },
    render: function(){
        return (<button onClick={this.getDragonKillingSword}>屠龙宝刀，点击就送</button>);
    }
});
```

这样就实现内部事件的响应了，那如果需要暴露接口怎么办呢？

暴露接口

事实上现在`getDragonKillingSword`已经是一个接口了，如果有一个父组件，想要调用这个接口怎么办呢？

父组件大概长这样：

```
var ImDaddyComponent = React.createClass({
    render: function(){
        return (
            <div>
                //其他组件
                <ButtonComponent />
                //其他组件
            </div>
        );
    }
});
```

那么如果想手动调用组件的方法，首先在ButtonComponent上设置一个`ref=""`属性来标记一下，比如这里把子组件设置成`<ButtonComponent ref="getSwordButton"/>`，那么在父组件的逻辑里，就可以在父组件自己的方法中通过这种方式来调用接口方法：

```
this.refs.getSwordButton.getDragonKillingSword();
```

看起来屌屌哒~那么问题又来了，父组件希望自己能够按钮点击时调用的方法，那该怎么办呢？

配置参数

父组件可以直接将需要执行的函数传递给子组件：

```
<ButtonComponent clickCallback={this.getSwordButtonClickCallback}/>
```

然后在子组件中调用父组件方法：

```
var ButtonComponent = React.createClass({
    render: function(){
        return (<button onClick={this.props.clickCallback}>屠龙宝刀，点击就送</button>);
    }
});
```

子组件通过`this.props`能够获取在父组件创建子组件时传入的任何参数，因此`this.props`也常被当做配置参数来使用

屠龙宝刀每个人只能领取一把，按钮点击一下就应该灰掉，应当在子组件中增加一个是否点击过的状态，这又应当处理呢？

组件状态

在React中，每个组件都有自己的状态，可以在自身的方法中通过`this.state`取到，而初始状态则通过`getInitialState()`方法来定义，比如这个屠龙宝刀按钮组件，它的初始状态应该是没有点击过，所以`getInitialState`方法里面应当定义初始状态`clicked: false`。而在点击执行的方法中，应当修改这个状态值为`click: true`：

```
var ButtonComponent = React.createClass({
    getInitialState: function(){
        //确定初始状态
        return {
            clicked: false
        };
    },
    getDragonKillingSword: function(){
        //送宝刀

        //修改点击状态
        this.setState({
            clicked: true
        });
    },
    render: function(){
        return (<button onClick={this.getDragonKillingSword}>屠龙宝刀，点击就送</button>);
    }
});
```

这样点击状态的维护就完成了，那么render函数中也应当根据状态来维护节点的样式，比如这里将按钮设置为`disabled`，那么render函数就要添加相应的判断逻辑：

```
render: function(){
    var clicked = this.state.clicked;
    if(clicked)
        return (<button disabled="disabled" onClick={this.getDragonKillingSword}>屠龙宝刀
    else
        return (<button onClick={this.getDragonKillingSword}>屠龙宝刀，点击就送</button>);
}
```

小节

这里简单介绍了通过JSX来管理组件的结构和逻辑，事实上React给组件还定义了很多方法，以及组件自身的生命周期，这些都使得组件的逻辑处理更加强大

资源加载

CSS文件定义了组件的样式，现在的模块加载器通常都能够加载CSS文件，如果不能一般也提供了相应的插件。事实上CSS、图片可以看做是一种资源，因为加载过来后一般不需要做什么处理。

React对这一方面并没有做特别的处理，虽然它提供了Inline Style的方式把CSS写在JSX里面，但估计没有多少人会去尝试，毕竟现在CSS样式已经不再只是简单的CSS文件了，通常都会去用Less、Sass等预处理，然后再用像postcss、myth、autoprefixer、cssmin等等后处理。资源加载一般也就简单粗暴地使用模块加载器完成了

组件依赖

组件依赖的处理一般分为两个部分：组件加载和组件使用

组件加载

React没有提供相关的组件加载方法，依旧需要通过`<script>`标签引入，或者使用模块加载器加载组件的JSX和资源文件。

组件使用

如果细心，就会发现其实之前已经有使用的例子了，要想在一个组件中使用另外一个组件，比如在`ParentComponent`中使用`ChildComponent`，就只需要在`ParentComponent`的`render()`方法中写上`<ChildComponent />`就行了，必要的时候还可以传些参数。

疑问

到这里就会发现一个问题，React除了只处理了结构和逻辑，资源也不管，依赖也不管。是的，React将

近两万行代码，连个模块加载器都没有提供，更与Angularjs，jQuery等不同的是，他还不带啥脚手架...没有Ajax库，没有Promise库，要啥啥没有...

虚拟DOM

那它为啥这么大？因为它实现了一个虚拟DOM（Virtual DOM）。虚拟DOM是干什么的？这就要从浏览器本身讲起

如我们所知，在浏览器渲染网页的过程中，加载到HTML文档后，会将文档解析并构建DOM树，然后将其与解析CSS生成的CSSOM树一起结合产生爱的结晶——RenderObject树，然后将RenderObject树渲染成页面（当然中间可能会有一些优化，比如RenderLayer树）。这些过程都存在与渲染引擎之中，渲染引擎在浏览器中是于JavaScript引擎（JavaScriptCore也好V8也好）分离开的，但为了方便JS操作DOM结构，渲染引擎会暴露一些接口供JavaScript调用。由于这两块相互分离，通信是需要付出代价的，因此JavaScript调用DOM提供的接口性能不咋地。各种性能优化的最佳实践也都在尽可能的减少DOM操作次数。

而虚拟DOM干了什么？它直接用JavaScript实现了DOM树（大致上）。组件的HTML结构并不会直接生成DOM，而是映射生成虚拟的JavaScript DOM结构，React又通过在这个虚拟DOM上实现了一个 diff 算法找出最小变更，再把这些变更写入实际的DOM中。这个虚拟DOM以JS结构的形式存在，计算性能会比较好，而且由于减少了实际DOM操作次数，性能会有较大提升

道理我都懂，可是为什么我们没有模块加载器？

所以就需要Webpack了

说说Webpack

什么是Webpack？

事实上它是一个打包工具，而不是像RequireJS或SeaJS这样的模块加载器，通过使用Webpack，能够像Node.js一样处理依赖关系，然后解析出模块之间的依赖，将代码打包

安装Webpack

首先得有Node.js

然后通过`npm install -g webpack`安装webpack，当然也可以通过gulp来处理webpack任务，如果使用gulp的话就`npm install --save-dev gulp-webpack`

配置Webpack

Webpack的构建过程需要一个配置文件，一个典型的配置文件大概就是这样

```
var webpack = require('webpack');
var commonsPlugin = new webpack.optimize.CommonsChunkPlugin('common.js');

module.exports = {
    entry: {
        entry1: './entry/entry1.js',
        entry2: './entry/entry2.js'
    },
    output: {
        path: __dirname,
        filename: '[name].entry.js'
    },
    resolve: {
        extensions: ['', '.js', '.jsx']
    },
    module: {
        loaders: [{
            test: /\.js$/,
            loader: 'babel-loader'
        }, {
            test: /\.jsx$/,
            loader: 'babel-loader!jsx-loader?harmony'
        }]
    },
    plugins: [commonsPlugin]
};
```

这里对Webpack的打包行为做了配置，主要分为几个部分：

- entry：指定打包的入口文件，每有一个键值对，就是一个入口文件
- output：配置打包结果，path定义了输出的文件夹，filename则定义了打包结果文件的名称，filename里面的`[name]`会由entry中的键（这里是entry1和entry2）替换
- resolve：定义了解析模块路径时的配置，常用的就是extensions，可以用来指定模块的后缀，这样在引入模块时就不需要写后缀了，会自动补全
- module：定义了对模块的处理逻辑，这里可以用loaders定义了一系列的加载器，以及一些正则。当需要加载的文件匹配test的正则时，就会调用后面的loader对文件进行处理，这正是webpack强大的原因。比如这里定义了凡是`.js`结尾的文件都是用`babel-loader`做处理，而`.jsx`结尾的文件会先经过`jsx-loader`处理，然后经过`babel-loader`处理。当然这些loader也需要通过`npm install`安装
- plugins: 这里定义了需要使用的插件，比如commonsPlugin在打包多个入口文件时会提取出公用的部分，生成common.js

当然Webpack还有很多其他的配置，具体可以参照它的配置文档

执行打包

如果通过`npm install -g webpack`方式安装webpack的话，可以通过命令行直接执行打包命令，比如这

样：

```
$webpack --config webpack.config.js
```

这样就会读取当前目录下的webpack.config.js作为配置文件执行打包操作

如果是通过gulp插件gulp-webpack，则可以在gulpfile中写上gulp任务：

```
var gulp = require('gulp');
var webpack = require('gulp-webpack');
var webpackConfig = require('./webpack.config');
gulp.task("webpack", function() {
    return gulp
        .src('./')
        .pipe(webpack(webpackConfig))
        .pipe(gulp.dest('./build'));
});
```

组件编写

使用Babel提升逼格

Webpack使得我们可以使用Node.js的CommonJS规范来编写模块，比如一个简单的Hello world模块，就可以这么处理：

```
var React = require('react');

var HelloWorldComponent = React.createClass({
    displayName: 'HelloWorldComponent',
    render: function() {
        return (

<div>Hello world</div>

);
    }
});

module.exports = HelloWorldComponent;
```

等等，这和之前的写法没啥差别啊，依旧没有逼格...程序员敲码要有geek范，要逼格than逼格，这太low了。现在都ES6了，React的代码也要写ES6，`babel-loader`就是干这个的。Babel能够将ES6代码转换成ES5。首先需要通过命令`npm install --save-dev babel-loader`来进行安装，安装完成后就可以使用了，一种使用方式是之前介绍的在`webpack.config.js`的loaders中配置，另一种是直接在代码中使用，比如：

```
var HelloWorldComponent = require('!babel!jsx!./HelloWorldComponent');
```

那我们应当如何使用Babel提升代码的逼格呢？改造一下之前的HelloWorld代码吧：

```
import React from 'react';

export default class HelloWorldComponent extends React.Component {
    constructor() {
        super();
        this.state = {};
    }
    render() {
        return (

<div>Hello World</div>

);
    }
}
```

这样在其他组件中需要引入HelloWorldComponent组件，就只要就可以了：

```
import HelloWorldComponent from './HelloWorldComponent'
```

怎么样是不是更有逼格了？通过import引入模块，还可以直接定义类和类的继承关系，这里也不再需要`getInitialState`了，直接在构造函数`constructor`中用`this.state = xxx`就好了

Babel带来的当然还不止这些，在其帮助下还能尝试很多优秀的ES6特性，比如箭头函数，箭头函数的特点就是内部的this和外部保持一致，从此可以和`that`、`_this`说再见了

```
['H', 'e', 'l', 'l', 'o'].map((c) => {
    return (<span>{c}</span>);
});
```

其他还有很多，具体可以参照Babel的学习文档

样式编写

我是一个强烈地Less依赖患者，脱离了Less直接写CSS就会出现四肢乏力、不想干活、心情烦躁等现象，而且还不喜欢在写Less时候加前缀，平常都是gulp+less+autoprefixer直接处理的，那么在Webpack组织的React组件中要怎么写呢？

没错，依旧是使用loader

可以在`webpack.config.js`的loaders中增加Less的配置：

```
{
  test: /\.less$/,
  loader: 'style-loader!css-loader!autoprefixer-loader!less-loader'
}
```

通过这样的配置，就可以直接在模块代码中引入Less样式了：

```
import React from 'react';

require('./HelloWorldComponent.less');

export default class HelloWorldComponent extends React.Component {
    constructor() {
        super();
        this.state = {};
    }
    render() {
        return (

<div>Hello World</div>

);
    }
}
```

其他

Webpack的loader为React组件化提供了很多帮助，像图片也提供了相关的loader：

```
{ test: /\.png$/, loader: "url-loader?mimetype=image/png" }
```

更多地loader可以移步webpack的wiki

在Webpack下实时调试React组件

Webpack和React结合的另一个强大的地方就是，在修改了组件源码之后，不刷新页面就能把修改同步到页面上。这里需要用到两个库`webpack-dev-server`和`react-hot-loader`。

首先需要安装这两个库，`npm install --save-dev webpack-dev-server react-hot-loader`

安装完成后，就要开始配置了，首先需要修改entry配置：

```
entry: {
  helloworld: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './helloworld'
  ]
},
```

通过这种方式指定资源热启动对应的服务器，然后需要配置`react-hot-loader`到loaders的配置当中，比如我的所有组件代码全部放在scripts文件夹下：

```
{
  test: /\.js?$/,
  loaders: ['react-hot', 'babel'],
  include: [path.join(__dirname, 'scripts')]
}
```

最后配置一下plugins，加上热替换的插件和防止报错的插件：

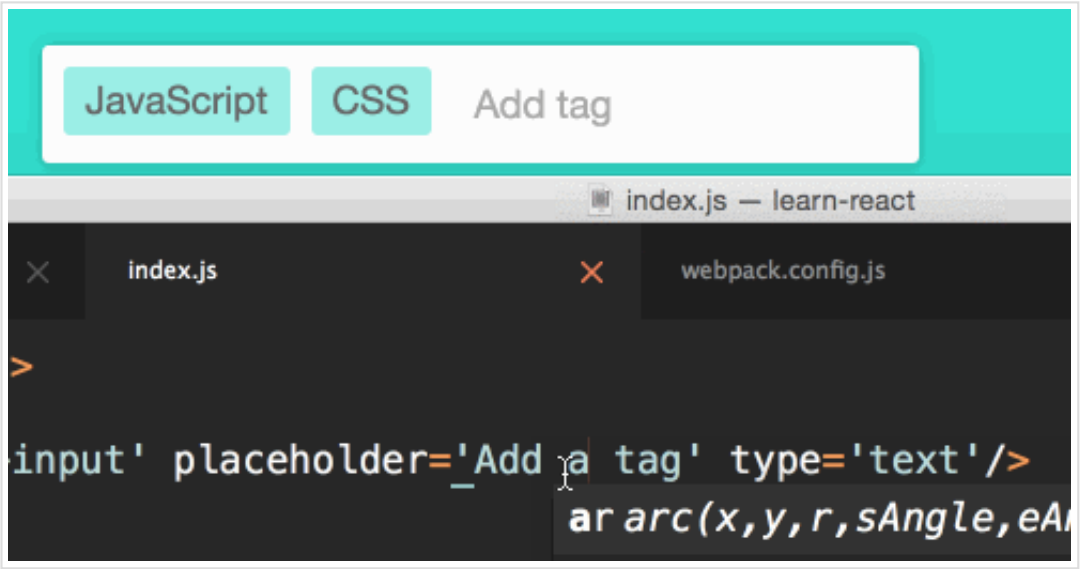
```
plugins: [
  new webpack.HotModuleReplacementPlugin(),
  new webpack.NoErrorsPlugin()
]
```

这样配置就完成了，但是现在要调试需要启动一个服务器，而且之前配置里映射到`http://localhost:3000`，所以就在本地3000端口起个服务器吧，在项目根目录下面建个server.js：

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('./webpack.config');

new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
  if (err) console.log(err);
  console.log('Listening at localhost:3000');
});
```

这样就可以在本地3000端口开启调试服务器了，比如我的页面是根目录下地`index.html`，就可以直接通过`http://localhost:3000/index.html`访问页面，修改React组件后页面也会被同步修改，这里貌似使用了websocket来同步数据。图是一个简单的效果：



结束

React的组件化开发很有想法，而Webpack使得React组件编写和管理更加方便，这里只涉及到了React和Webpack得很小一部分，还有更多的最佳实践有待在学习的路上不断发掘

2015年05月15日发布

24 推荐 收藏