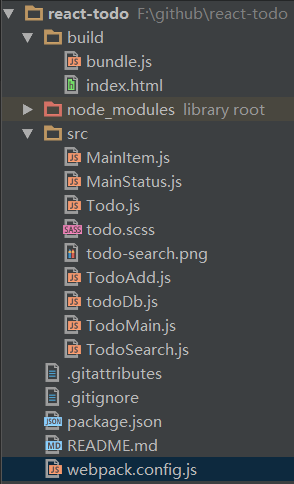
# react-todo实战(一)之webpack

发表于 2016-11-25   |   分类于 [react](http://gjincai.github.io/categories/react/)  |     |   阅读次数 54

看过一遍webpack/es6/react的基础之后，感觉还算不上真正入门react的实战开发，于是乎想边学边搞一个小项目、把这三webpack/es6/react结合起来用一遍，从实际小项目中反过来学习基础要点。  
(react的基础学习在 [前面的学习](http://gjincai.github.io/categories/react/)。)

全部亲手一行行敲完后，终于有了这个稍微成型的入门小实例：  
完整代码github：[react-todo](https://github.com/gjincai/react-todo" \t "_blank)  
演示链接demo：[react-todo](http://ohe5avf3y.bkt.clouddn.com/pro/react/react-todo/index.html" \t "_blank)  
实践中收货良多，过后便稍微总结一下系列的笔记。

## react-todo 目录

[](http://ohe5avf3y.bkt.clouddn.com/blog/201612/jam-blog-react-todo-20161203.png)

## webpack.config.js最终版

webpack的配置初接触还是坑满满的，先上一个最终版的：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | var path = require('path'); module.exports = {  entry: path.resolve(\_\_dirname,'./src/Todo.js'),  output: {  path: path.resolve(\_\_dirname,'build'),  filename: 'bundle.js'  },  devtool: 'eval-source-map',  module: {  loaders: [  {test: /\.jsx?$/, exclude: /node\_modules/, loader: 'babel',query: {presets: ['es2015','react']}},  {test: /\.(png|jpg|gif)$/, loader: 'url-loader?limit=8192'},  {test: /\.scss$/, loader: 'style!css!sass'}  ]  } } |

接下来再拆分说明一下

## 模块引入

三种方式：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | //es6 import todoDb from './todoDb.js'; //CommonJS var todoDb = require('./todoDb.js'); //AMD define(['./todoDb.js'], function (todoDb) {  // }); |

注意：**相对路径**是相对当前目录，**绝对路径**是相对入口文件(这里是Todo.js)。

## 配置简介

**exports的两种写法**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | //1 module.exports ={  //entry:...... }  //2 var config = {  //entry:...... } module.exports = config; |

**配置参数**  
entry：打包的入口文件，每有一个键值对，就有一个入口文件；  
output：打包结果，path定义输出文件夹，filename定义打包结果文件的名称；  
devtool: 调试错误定位，一般学习dev下设置为’eval-source-map’；  
module：定义对模块的处理，对正则匹配的模块采用相应引入的loader进行处理。

**loaders**  
webpack本身只能处理JS模块，要处理其它类型文件（scss/css/图片等）就需要引入相应的loader进行转换。  
loader的一般参数：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | //1 test: /\.jsx?$/, //正则匹配要处理的文件 loader: 'babel', //要引用的loader exclude: /node\_modules/, //要忽略的文件 include: [path.resolve(\_\_dirname,'src')], //要处理的文件 query: { //query:要传给loader的属性  presets: ['es2015','react'] //使用安装的es2015、react模块 }  //2 test: /\.(png|jpg|gif)$/, loader: 'url-loader?limit=8192' //图片小于8192则转换为64位base的data  //3 test: /\.scss$/, loader: 'style!css!sass' //同时引用多个loader,用！隔开 |

## 自动刷新

**方式一iframe**  
编辑package.json文件的script，设置webpack-dev-server指向的目录：

|  |  |
| --- | --- |
| 1 2 3 4 | "scripts": {  "dev": "webpack-dev-server --content-base ./build",  "build": "webpack --progress --colors" } |

然后直接访问localhost:8080/webpack-dev-server/index.html即可。

**方式二inline**  
新增一个 --inline:

|  |  |
| --- | --- |
| 1 2 3 4 | "scripts": {  "dev": "webpack-dev-server --inline --content-base ./build",  "build": "webpack --progress --colors" } |

然后访问localhost:8080/index.html即可。

## 热加载

新增一个 --hot:

|  |  |
| --- | --- |
| 1 | "dev": "webpack-dev-server --inline --hot --content-base ./build", |

坑：我当初用的是webstorm/phpstorm IDE，这个东西强大得太智能了，当修改了文件后按ctrl+s保存后，hot热加载并没有生效啊！  
当时折腾了好久才猛然发现，是IDE的自动保存缓存之类搞的鬼；  
解决：设置webstorm/phpstorm：在 system settings中 取消 use “safe write”….，并设置修改过的文件用 \* 标记。

（话说有个React-transform热替换更好，往后再试试）

## 遇到的坑

**node\_modules惹的祸**  
webstorm/phpstorm完全被卡住，最后直接挂掉；github客户端也同步不了……  
原因都是因为node\_modules！  
解决：  
webstorm/phpstorm：先建好node\_modules文件夹；然后设置将其忽略、不检索该目录的依赖；再npm install安装loaders。  
github客户端：在 .gitignore文件下加入 node\_modules/，将其忽略不必提交到git。

**loaders安装**  
在公司电脑安装好全部loaders,然后拿着package.json回到宿舍再 npm install 、各种报错；什么鬼版本号、缺少模块，全部卸载重新安装、补充安装、增加设置……

# react-todo实战(二)之react/es6

发表于 2016-11-27   |   分类于 [es6](http://gjincai.github.io/categories/es6/)  |   [暂无评论](http://gjincai.github.io/2016/11/27/react-todo%E5%AE%9E%E6%88%98-%E4%BA%8C-%E4%B9%8Breact-es6/#comments)  |   阅读次数 33

## module的导入导出

**module的导入**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | import React from 'react'; import ReactDOM from 'react-dom'; //import { render } from 'react-dom'; //导入模块的某一个功能函数 import todoDb from './todoDb.js'; //导入本地数据库localStorage模块  import TodoSearch from './TodoSearch.js'; //导入搜索任务组件 import TodoMain from './TodoMain.js'; //导入main任务组件 import TodoAdd from './TodoAdd.js'; //导入add新增任务组件  import './todo.scss'; //导入scss import todoSearch from './todo-search.png'; //导入图片 |

**module的导出**

|  |  |
| --- | --- |
| 1 2 3 4 5 | //默认导出 export default class TodoSearch extends React.Component{} //命名导出 class TodoSearch extends React.Component{} export TodoSearch; |

## class

直接来看MainItem.js组件里的一个 class的结构:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | class MainItem extends React.Component{  deleteClick(){  this.props.deleteTask(this.props.index);  }  changeClick(){  let isDone = !this.props.isDone;  this.props.changeDone(this.props.index,isDone);  }  render(){  let isDoneClass = this.props.isDone ? 'li-status' : 'li-status li-status-no';  //已/未完成的样式  let isDoneTxt = this.props.isDone ? '已完成' : '未完成'; //已/未完成的文字  return(  <li>  <span className="li-text">{this.props.index+1}.{this.props.task}</span>  <button className="li-delete" onClick={this.deleteClick.bind(this)}>删除</button>  <button className={isDoneClass} onClick={this.changeClick.bind(this)}>{isDoneTxt}</button>  </li>  )  } } |

类MainItem里面，定义方法不用写function，方法之间不用写逗号 ,；  
在render里面的dom元素可以绑定这些事件：onClick={this.deleteClick.bind(this)}；  
虚拟事件对象一般有如下这些，它们是 react 封装好的，跟浏览器本地事件很像：  
onKeyDown onKeyPress onKeyUp onFocus onBlur onClick onDoubleClick 等；  
具体可 [查阅文档](http://www.css88.com/react/docs/events.html)

## props refs

**props**：父组件向子组件传递数据：

|  |  |
| --- | --- |
| 1 2 3 4 5 | //Todo.js中向子组件传递 searchTasks 方法 <TodoSearch searchTasks={this.searchTasks.bind(this)}/>  //TodoSearch.js 通过 this.props 调用 this.props.searchTasks(val); |

**ref**: 获取当前组件下的真实dom：

|  |  |
| --- | --- |
| 1 2 3 4 5 | //TodoSearch.js搜索组件中，需要获取 input 的值；先给 input 定义 ref <input type="text" ref='inputSearch' onKeyUp={this.searchUp.bind(this)} placeholder="搜索"/>  //通过 this.refs 访问获取 let val = ReactDOM.findDOMNode(this.refs.inputSearch).value; |

## spread操作符

Todo.js 通过 spread 向子组件 TodoMain.js 传递 count

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | //Todo.js 的 render(){}里面： var count = {  countAll: this.state.tasks.length || 0,  countDone: (this.state.tasks && this.state.tasks.filter((task) => task.isDone)).length || 0 } <TodoMain {...count}/> |

那么，我们在 TodoMain.js 的 this.props 得到的就是一个对象：{countAll:数字, countDone:数字}。

spread 操作符 在子组件向孙组件传递父组件的数据时，特别好用。比如子组件 TodoMain.js 向 孙组件 MainItem.js 传递父组件 Todo.js 的数据：<MainItem {...this.props}/> 。

## 几个函数

**filter**  
过滤，遍历数组，对每一个元素触发一个回调函数，经处理判断、保留或删除当前元素，最后返回一个新数组；  
如，在 Todo.js 的 searchTasks方法中，就是通过 filter 筛选出搜索结果的：

|  |  |
| --- | --- |
| 1 | let searchResult = this.todoDb.get().filter((tasks) => tasks.task.includes(val)); |

this.todoDb.get() 得到的是一个对象为元素的数组，具体形式如下：  
[{'task':'任务1','isDone':true},{'task':'任务2','isDone':false}]  
箭头函数无需 return 。

**map**  
遍历数组，对每一个元素触发一个回调函数，经处理后生成新的当前元素，最后返回一个新数组；  
如：在 TodoMain.js 向 MainItem.js 传递数据时就采用 map 遍历：

|  |  |
| --- | --- |
| 1 2 3 | this.props.tasks.map((tasks,index) => {  return <MainItem key={index} {...tasks} index={index} {...this.props}/> }) |

array1.map(callbackfn[, thisArg]) ：  
array1 必选 一个数组对象。  
callbackfn 必选 最多可以接受三个参数的函数。  
thisArg 可选

回调函数 callbackfn 语法：  
function callbackfn(value, index, array1):  
Value 数组元素的值。(tasks：数组里的一个对象，如：{‘task’:’任务1’,’isDone’:true})  
index 数组元素的数字索引。（index：当前元素的索引）  
array1 包含该元素的数组对象。

**some()、every()**  
遍历数组，对每一个元素触发一个回调函数，通过判断，返回一个布尔值。  
some()：只要有一个满足判断，就返回true，  
every()：只要有一项不满足判断，就返回false。

**todoDb**  
本地数据采用 localStorage 来存取，具体详见 todoDb.js 模块：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | export default class todoDb{  constructor(name){  this.name = name;  if(JSON.stringify(this.get(this.name)) == '{}' ){  // this.set([{'task':'默认的任务','isDone':false}]);  this.set([]);  }  }  set(val){  window.localStorage.setItem(this.name,JSON.stringify(val));  }  get(){  return JSON.parse(window.localStorage.getItem(this.name)) || {};  } } |

localStorage 不能存数组，所以关键是  
存的时候（set）先把数组对象转成字符串： JSON.stringify()；  
取的时候（get）先把字符串转成数组对象： JSON.parse()。

构造函数 constructor 里先判断如果数据为空，则默认先设置一个空的数组；不然 被 map 遍历时出错（map遍历的是数组）。