### React：

**安装使用：**

npm install react react-dom --save

1,cass：用className,label的for用forHtml

2,style写法：style={{textAlign:"center",color:'red'}}

3，获取dom,ref和findDOMNode区别：

this.ref.xx //获取虚拟的dom，拿到子组件的props/state/方法

ReactDOM.findDOMNode(this.refs.xx)) //获取真实的dom，原生的标签相关属性

注：之前的写法是 React.findDOMNode，现在findDOMNode放在ReactDom中，所以要import ReactDOM from 'react-dom'

注：新版react中不推荐ref string方式，代替使用ref callback方式，如，用ref={child => this.\_child = child}代替ref=”child”

获取用 this.\_child 代替this.refs.child;

ReactDOM.findDOMNode(this.\_child）代替 ReactDOM.findDOMNode(this.refs.child))

4，生命周期：

**componentWillMount()：**

首次渲染之前调用, render 方法调用之前修改 state 的最后一次机会

**componentDidMount():**

组件第一次渲染完成，可以对jquery插件进行初始化，如：$(ReactDOM.findDOMNode(this.refs.ick)).iCheck()

**componentWillUpdate(object nextProps, object nextState)：**

在组件接收到了新的 props 或者 state 即将进行重新渲染前会被调用，注意不要在此方面里再去更新 props 或者 state

**componentDidUpdate(object prevProps, object prevState)：**

重新被渲染之后会被调用。可以在这里访问并修改 DOM

**componentWillUnmount()：**

组件销毁前调用,可以对插件进行卸载

**componentWillReceiveProps(object nextProps)：**

当props发生变化时执行，可以在这个方法里更新 state,以触发 render 方法重新渲染组件

**shouldComponentUpdate(object nextProps, object nextState)：**

组件判断是否重新渲染时调用，如果你确定组件的 props 或者 state 的改变不需要重新渲染，可以通过在这个方法里通过返回 false 来阻止组件的重新渲染，返回 `false 则不会执行 render 以及后面的 componentWillUpdate，componentDidUpdate 方法。

5,绑定方法要绑定this：如：onClick={this.\_login.bind(this)}或

es6写法onClick={()=>this.\_login()}

**propTypes:**

propTypes 使用來规范元件Props的数据类型，若没有按照规范提示警告

propTypes: {

// required

requiredFunc: React.PropTypes.func.isRequired,

// primitives, optional by default

bool: React.PropTypes.bool,

func: React.PropTypes.func

}

**context：**

只要parent component定义了context，那么可以不必通过props层层传递，就能直接在child components拿到。

相关api

**withContext**：会执行一个指定的上下文信息的回调函数,在当前组件和子孙组件都有context的访问权限，必须显式的指定contextTypes 的属性。

如：React.withContext({'name': 'Jonas'}, function () {

React.render(<A />, document.body);

});

var A = React.createClass({

contextTypes: {

name: React.PropTypes.string.isRequired,

},

render: function() {

return <div>My name is: {this.context.name}</div>;

}

});

**childContextTypes**：声明传递给子组件的属性的数据类型。

**getChildContext**：设置传递给子组件的属性，可以覆盖，也可以新增。子孙组件都有context的访问权限。 需要先通过 childContextTypes 来指定，不然会产生错误。

如： Parent.childContextTypes={

name: React.PropTypes.string.isRequired

}

getChildContext(){

return {

name: "Jonas"

};

}

**contextTypes**：当需要在当前组件使用从上级组件传入的context的属性时，需要为用到的属性声明数据类型

在组件中，想访问context里面的属性的组件都必须显式的指定一个 contextTypes 的属性,通过this.context访问context中的属性或方法。

如：Son.contextTypes={

fruit: React.PropTypes.string.isRequired

}

注：context 是就近引用的，如果你通过withContext 指定了context元素，然后又通过 getChildContext 指定了该元素，该元素的值将会被覆盖。

### redux：

**1，安装使用：**

npm install --save redux

npm install --save react-redux

**2，redux流程：**

1，view直接触发dispatch；

2，dispatch将action发送到reducer中后，reducer会根据这个action的type返回需要新的数据，根节点上会更新props，更新全局view。

react-redux：是对redux流程的一种封装，使其可以适配与react的代码结构。

react-redux首先提供了一个Provider，可以将从createStore返回的store放入context中，使子集可以获取到store并进行操作；

Provider：利用react的context特性，将store放置在根节点的context中

<Provider store={store}>

{() => <App />}

</Provider>

其次react-redux提供了connect方法，将原始根节点包装在Connect下，在Connect中的state存储不可变对象，并将state对象中的props和store中的dispatch函数传递给原始根节点

connect接受3个参数

connect(mapStateToProps, mapDispatchToProps, mergeToProps)(App);

第一个函数接收store中state和props，使页面可以根据当前的store中state和props返回新的\*\*stateProps\*\*；使用时，直接用this.props.\*\*

如：const mapStateToProps=(state) =>{

return {

"clmsChannel":state.clmsChannel,

"clmsConfig": state.clmsConfig,

}

}

第二个函数接收store中的dispatch和props，使页面可以复写dispatch方法，返回新的\*\*dispatchProps\*\*；使用时，直接用this.props.\*\*

如const mapDispatchToProps = (dispatch, ownProps) => {

return {

increase: (...args) => dispatch(actions.increase(...args)),

decrease: (...args) => dispatch(actions.decrease(...args))

}

}：

第三个函数接收前2个函数生成的\*\*stateProps\*\*和\*\*dispatchProps\*\*，在加上\*\*原始的props\*\*，合并成新的props，并传给原始根节点的props。可以不传这个参数，connect 就会使用 Object.assign 替代该方法。

**3，redux中间件：**

redux中间件：applyMiddleware

//导入

import { applyMiddleware, createStore } from 'redux';

//使用

const store = createStore(

reducer,

applyMiddleware(thunk, promise, logger) //3个参数就是中间件，有次序要求，如logger必须放在最后

);

thunk:异步多个action,渲染一次页面

如：export function incrementAsync(action) {

return dispatch => {

dispatch(addData({"ccc":"111"}));

setTimeout(() => {

dispatch(action);

}, 1000);

dispatch(addData({"bbb":"222"}));

};

}

### router：

**安装使用：**

$ npm install -S react-router

import { Router } from 'react-router';

render(<Router/>, document.getElementById('app'));

**传参方式：**

1，/clmsConfig(/:id)，获取参数方式：this.props.params.id

2，/clmsConfig?id=””或to={{ pathname: '/users/ryan', query: { id: “”} }}，获取参数方式：this.props.location.query.id

**路由跳转**（Link，to）：用于取代<a>元素，生成一个链接

1， this.context.router.push('/') ，注：这个写法会把跳转载入浏览器历史，若不想留下历史记录则可以 this.context.router.replace('/')

**路由权限控制：**

onEnter可以在路由跳转前拦截，有3个参数分别是，nextState, replace, next

nextState：路由信息，location，routes，params，components

replace：重定向， replace('/error')

next：成功跳转，next()

**IndexRoute:**

根路由组建，默认显示页

注：如果链接到根路由/，不要使用Link组件，而要使用IndexLink组件。

因为/会匹配任何子路由，activeStyle和activeClassName会失效

另一种解决方法是使用Link组件的onlyActiveOnIndex属性，也能达到同样效果。

如：<Link to="/" activeClassName="active" onlyActiveOnIndex={true}>

Home

</Link>

**路径匹配：**

1. :paramName：匹配URL的一个部分，直到遇到下一个/、?、#为止。这个路径参数可以通过this.props.params.paramName取出。

如：path="/hello/:name" // 匹配 /hello/michael

//匹配 /hello/ryan

1. ()：表示URL的这个部分是可选的。

如： path="/hello(/:name)" // 匹配 /hello

// 匹配 /hello/michael

1. \*：匹配任意字符，直到模式里面的下一个字符为止。匹配方式是非贪婪模式。

如：path="/files/\*" // 匹配 /files/

// 匹配 /files/a

// 匹配 /files/a/b

1. \*\*：匹配任意字符，直到下一个/、?、#为止。匹配方式是贪婪模式。

如：path="/\*\*/\*.jpg" // 匹配 /files/hello.jpg

// 匹配 /files/path/to/file.jpg

注：同时匹配两个规则，第二个规则不会生效，当用户访问/about/me时，含有path="/:userName/:id" 和path="/about/me"时，第二个路径不会生效，因此带参数的路径一般要写在路由规则的底部

**histroy 属性**

1，**browserHistory**:显示正常的路径example.com/some/path，背后调用的是浏览器的History API，但是，这种情况需要对服务器改造。否则用户直接向服务器请求某个子路由，会显示网页找不到的404错误。如果开发服务器使用的是webpack-dev-server, --history-api-fallback

浏览器地址形式：example.com/about?id=asas

链接形式：to="/about?id=asas"

获取参数形式：{this.props.location.query.id}

2，**hashHistory**:路由将通过URL的hash部分（#）切换，URL的形式类似example.com/#/some/path，参数路由配置： <Route path="/about(/:id)" component={About}/>

浏览器地址形式：example.com/#/about/asas

链接形式：to="/about/asas"

获取参数形式： {this.props.params.id}

WebPack

**1,安装：**

npm install -g webpack

项目安装webpack：npm install webpack --save-dev

安装webpack-dev-server服务器：npm install webpack-dev-server --save-dev

**2，配置：**

webpack配置文件主要分为三大块

1，**entry** 入口文件 让webpack用哪个文件作为项目的入口；

可以直接用文件夹名称 默认会找index.js，

如：entry:path.resolve(ROOT\_PATH, 'src');

也可以确定是哪个文件名字

如：

entry: {

app: path.resolve(APP\_PATH, 'routes.js')

},

2，**output** 出口 让webpack把处理完成的文件放在哪里；

如： output: {//输出的文件名 合并以后的js会命名为bundle.js

path: path.resolve(ROOT\_PATH, 'dist');,

filename: 'bundle.js' //filename: '[name].js'

},

3，**module** 模块 要用什么不同的模块来处理各种类型的文件

使用loader的方式来处理各种各样的资源，比如说样式文件，我们需要两种loader，css-loader 和 style－loader，css-loader会遍历css文件，找到所有的url(...)并且处理。style-loader会把所有的样式插入到你页面的一个style tag中。

安装我们的loader：

npm install css-loader style-loader --save-dev

配置loader，在webpack.config.js中：

module: {

loaders: [

{

test: /\.css$/,

loaders: ['style', 'css'],

include: APP\_PATH

}

]

}

入口文件加载css:import "./client.css"

处理其他文件：

//处理恶事es6语法的支持

{

test: /\.jsx?$/,loader: 'babel',include: APP\_PATH,

//有了.babelrc文件，就不用写query

query: {

presets: ['es2015','react','stage-0']

}

},

//处理jquery

{ test: require.resolve('jquery'), loader: 'expose?$!expose?jQuery' },

//图片压缩

{test: /\.(jpg|png|gif)$/, loader: "file"},

//除里字体图片，如iconfont

{ test: /\.(woff|svg|eot|ttf)\??.\*$/,loader:'url-loader?limit=10000'}

//limit，当你图片大小小于这个限制的时候，会自动启用base64编码图片

**插件plugins：**

//添加插件 会自动生成一个html文件

plugins: [

new HtmlwebpackPlugin({

title: 'app',

template: path.resolve(path.resolve(APP\_PATH, 'templates'), 'index.html'),

//引用模板文件名

filename: 'index.html',

inject: 'body'

})

]

**配置webpack-dev-server：**

devServer: {

historyApiFallback: true,

hot: true,

inline: true,

progress: true

}

**添加React Transform支持**

npm install --save-dev babel-plugin-react-transform

//如果想实现Hot Module Replacement (说白了就是页面不刷新，直接替换修改的Component)，再安装一个transform

npm install --save-dev react-transform-hmr

//如果我们还要再来一个在页面上直接显示catch到的错误的transform，（不想打开console看到底有啥错误，直接显示在页面上多好），简单！再安装一个transform:

npm install --save-dev react-transform-catch-errors redbox-react

//要让新建的两个transform生效,只需再安装一个present。

npm install babel-preset-react-hmre --save-dev

安装完毕，将支持HMR和Catch Error的present添加到.babelrc

如：{

"presets": ["react", "es2015"],

//在开发的时候才启用HMR和Catch Error，打包时要去掉下面的代码

"env": {

"development": {

"presets": ["react-hmre"]

}

}

}

**package.json 中的 包/库 部分说明**

babel-core babel6 的基础模块

babel-eslint ESLint 是前端JS代码检测利器。而 babel-eslint 则允许你检测所有的 Babel 代码。

babel-loader 这个包允许你使用 Babel 和 webpack 转译（Transpiling） JavaScript 文件。

babel-plugin-react-transform 这个插件通过任意转换的方式去封装 React 组件。换句话说，你可以随心所欲的摆弄你的组件了。

babel-plugin-transform-runtime 在 Babel 转换过程中，详细的展示引用的相关辅助工具和内置命令，并自动的聚合填充你的代码而不会污染全局。

babel-preset-es2015 此预设包含了所有的 es2015 插件。

babel-preset-react 此预设包含了所有的 React 插件。

babel-preset-stage-0 此预设包含了 stage 0 中的所有插件。

eslint JavaScript 语法检测利器：分析出你代码潜在的错误和非标准用法。

eslint-plugin-react ESLint 中关于 React 语法检测的插件。

react-transform-hmr 一个 React 转换装置，该装置通过引用 Hot Module Replacement API 使热重载 React 的类成为可能。

react-transform-catch-errors 呈现你 React 组件的错误信息。

webpack-dev-server 为 wepack app 提供一个服务器，如果你的代码有任何变化，浏览器将自动刷新显示，极大的方便前期开发。

babel-runtime Babel 自带的运行环境。

另外，我增加了style-loader，css-loader，less，less-loader这四个包用于加载.less文件模块。（注意，less-loader与less必须同时存在才能正常工作。）

**部署上线**

刚才说的各种情况都是在开发时候的情况，那么假如项目已经开发完了，需要部署上线了。我们应该新创建一个单独的config文件，因为部署上线使用webpack的时候我们不需要一些dev-tools,dev-server和jshint校验等。

复制我们现有的config文件，命名为 webpack.production.config.js，将里面关于 devServer等和开发有关的东西删掉。

在package.json中添加一个命令。

"scripts": {

"start": "webpack-dev-server --hot --inline",

"build": "webpack --progress --profile --colors --config webpack.production.config.js"

},

当要上线的时候,运行 npm run build