React同构漫谈

同构指的是相同代码可以同时在客户端与服务端同时渲染的技术，本文将从实际项目出发，谈谈开发过程中遇到的问题以及解决方案。

在开始阅读本文之前，你需要有一定的react同构基本概念。如果尚未接触过同构，建议先参考一些相关的同构项目：

* <https://github.com/RickWong/react-isomorphic-starterkit>
* <https://github.com/kriasoft/react-starter-kit>

React同构主要分成以下几个步骤：

* 服务端将请求交由React Router解析
* React Router生成页面布局
* 服务端将生成结果文本化返回给客户端
* 客户端由React Router生成页面布局
* React将其与服务端布局进行对比
* 对比成功，复用当前页面；反之则重新渲染

Server

Client

React Router

Request Access

View

React Router

Web Content

View

React reuse?

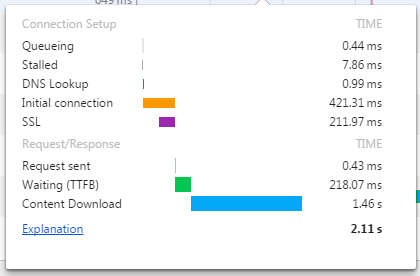
Redraw

Done

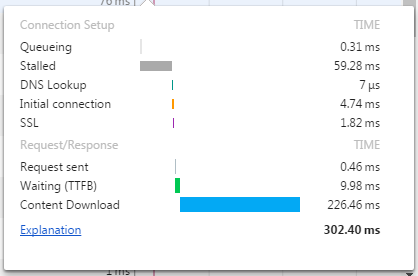
No

Yes

利用local api call响应更快的特点，可以减少总体的页面可用性等待时间。



Remote API Call



Local API Call

或许你会疑惑，为什么React在客户端还需要再次进行渲染并验证。这是需要分成两个问题来分别讨论。

1. **为何需要再次渲染?**

React中，组件通过Props和State来决定组件表现。例如，我们现在有一个Checkbox组件：

**const** CheckBox = ({ title, checked }) => (  
 <label>  
 <input type="checkbox" checked={checked} />  
 {title}  
 </label>  
);

通过服务端渲染转换成dom element后将会丢失virtual dom结构信息：

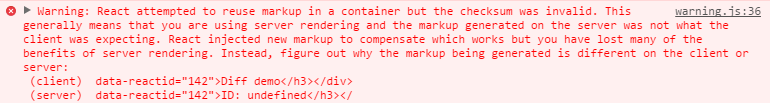
<label><input type="checkbox" checked="checked" />Hello World</label>

因此，为了React能够在客户端正常运行。客户端也需要进行一次渲染，构建出virtual dom结构。

1. **为何需要验证?**

既然在客户端和服务端都进行了渲染，那么就有可能存在前后端渲染出来的结构不同步的情况（之后会给出例子）。当出现这种情况时，为了保证单页应用能够正常工作，React总是会以客户端渲染为准。

当验证后，发现当前的页面元素相匹配。React便会跳过virtual dom -> real dom的过程，直接复用已有元素，从而加快了页面构建速度。反之，只能抛弃服务端的渲染内容。从新创建页面：



好了，在大部分的演讲中。同构似乎就这么简单，了解了基本流程，然后改造上线，同构完成了。其实不然，这仅仅是一个开始。你需要在开发过程中不断复现出以上的渲染流程，才能保证在服务端和客户端的控制台中不打印出讨厌的warning信息。那么，你会遇到什么问题呢？

1. **保持数据同步**

在实际同构中，你需要保证服务端与客户端共享相同的数据集合才能生成出相同的virtual dom。在我的开发中，通过使用redux进行数据管理。在渲染完成后，将store的内容通过js传递给客户端：

**const** store = createStore(reducers, { ... });  
  
**const** App = (  
 <StaticRouter location={req.url} context={context}>  
 <Provider store={store}>  
 <Main />  
 </Provider>  
 </StaticRouter>  
);

**const** componentHTML = renderToString(App);

res.end(  
`<!DOCTYPE html>  
<html>  
<head>  
 ...  
</head>  
<body>

<div id="root">${componentHTML}</div>  
 <script>  
 window.\_\_INITIAL\_STATE\_\_ = ${store.getState()};   
 </script>  
</body>  
</html>`);

如果你开发过大型单页应用，你可能已经发现了问题。在实际的项目中，我们不会一下子便初始化store的所有内容。例如购物车页面不会需要管理你的好友信息，优惠券页面不需要你的支付信息等等。我们会将store进行部分初始化，将大部分页面通用的内容进行填充。但是对于剩余内容，在页面打开后才进行数据请求：

**class** UserInfo **extends** React.Component {  
 componentWillMount() {  
 **const** { user, userInfo } = **this**.props;  
  
 **if** (!userInfo) {  
 dispatch(loadUser(user));  
 }  
 }  
  
 ...

}

当页面存在异步请求的时候，你会发现同构变成了一团乱麻。用户访问的页面在渲染给客户端的时候，需要state还为填充，以至于客户端需要再次发起api请求。同时，服务端的这次请求白白浪费了。

更甚者是，当数据存在依赖关系。页面在渲染时需要多次有序api请求时，你自然而然会想到一种解决方案：**路由表**

* 1. **路由表**

思路非常简单，在数据初始化之前。我们让用户访问的url进行一次路由表匹配。从而填充需要的state信息：

**const** ROUTER\_TABLE = {  
 '/user': [loadUser],  
 '/user/info': [loadUser, loadUserInfo],  
 '/shoppingCart': [loadUser, loadShoppingCart],

...   
};

然而问题在于，随着页面的增多，以及相关的页面逻辑更改。你总是需要同时维护两份数据依赖逻辑（服务端和客户端），同构并没有解放你的双手。

接着，你会开始尝试寻找可以前后端通用的解决方案：**Promise队列**

* 1. **Promise队列**

对于服务端渲染，我们需要解决的是在返回用户web content之前，等待所有的异步api完成。因而我们需要监视当前的渲染的api状态。同时，由于存在数据依赖。我们需要循环监听api请求，直到队列中没有额外的请求：

View Build

Promise List

User Info

Promise.all(promiseList)

Has Promise?

Done

Yes

No

这里，我们就不得不提到React的context属性。Context允许你在组件之间传递共享数据和方法，而不需要经过props传递。因而当你在Top component中注册了promiseListener后，所有子组件都可以将异步promise置于其中。

**class** Main **extends** React.Component {  
 getChildContext() {  
 **const** { promiseList } = **this**.props;  
 **return** {  
 addIsomorphicPromise: promise => {

**if** (promiseList) promiseList.push(promise);

},   
 };  
 }

...

}

...

根组件Main接受一个promiseList属性，并提供全局的addIsomorphicPromise方法。当子组件/页面发起请求时。我们将promise放入list之中。由于仅有服务端渲染会用到promise队列，当props中没有promiseList（客户端）则不添加。

接着，我们简单改造一下dispatch的过程：

**class** UserInfo **extends** React.Component {  
 componentWillMount() {  
 **const** { user, userInfo } = **this**.props;

**const** { addIsomorphicPromise } = **this**.context;  
  
 **if** (!userInfo) {  
 addIsomorphicPromise(dispatch(loadUser(user)));  
 }  
 }  
  
 ...

}

...

注：这里使用了[react-thunk](https://github.com/gaearon/redux-thunk)对action进行封装，返回值为fetch promise。

最后，在server端编写递归方法：

**function** loopRender($app, promiseList) {  
 promiseList.splice(0);  
  
 **return new** Promise((resolve, reject) => {  
 **const** componentHTML = renderToString($app);  
  
 **if** (promiseList.length === 0) {  
 resolve(renderToString(componentHTML));   
 } **else** {  
 Promise.all(promiseList).then(() => {  
 resolve(loopRender($app, promiseList));  
 }).catch((err) => {  
 reject(err);  
 });  
 }  
 });  
}

此外，在实际开发过程中，还需要做递归次数限制以防止逻辑错误导致遗漏添加promise导致store未更新产生的死循环。同时，如果你的页面存在通过store动态构建的子组件/页面嵌套dispatch，那么在promiseList为空时还需要额外的一次render check以防止页面渲染未是最终态。

经过以上改造，你的页面代码已经实现了数据加载的复用。但是，并非所有情况下。你都需要让服务端完全渲染完毕页面再返回给用户。你需要适当地对数据请求进行拆分以到达速度响应与可用性的平衡：

User

Shopping Cart

Important non-emergency like Item Detail

Isomorphic

Client

（部分依赖数据后置）

Data1.1

Date1.2

Date1.3

Data2.1

Data2.2

Isomorphic

Client

Page

Data1

Data2

Data1 & 2

（组合数据拆分）

将页面的基本组成进行服务端渲染后，部分内容提供载入动画以达到用户体验的平衡。我们通过使用React组件的2个生命周期方法组合可以实现这个效果：

**componentWillMount**

上文已经提到过。使用该方法实现同构的数据请求。

**componentDidMount**

该方法仅会在客户端触发，因而在该方法中进行数据请求不会在服务端触发。从而达到数据拆分的效果。

**class** Sample **extends** React.Component {  
 componentWillMount() {  
 **const** { data1 } = **this**.props;  
  
 **if** (!data1) {  
 dispatch(loadData1());  
 }  
 }

componentDidMount() {  
 **const** { data2 } = **this**.props;  
  
 **if** (!data2) {  
 dispatch(loadData2());  
 }  
 }

...

}

当搞定这些，你的同构代码离work更近了一步。记得在上文，我们提到过。如果服务端和客户端的virtual dom tree不同步时，总会以客户端为准。然而当准备完这些内容，我们仍然会在console中看到warning信息。为什么呢？我们需要再从redux说起。

**按需加载的得与失**

我们将应用内的数据拆分在多个redux state中，当用户访问不同页面的时候，通过componentWillMount和componentDidMount异步加载。当我们的component state数据有部分来自于redux state的延伸数据。我们需要额外做一次处理。

1. **页面初次载入**

代码非常好理解，数据fetch完毕后setState更新组件：

componentWillMount() {  
 **const** { dispatch, data1 } = **this**.props;

**const** { addIsomorphicPromise } = **this**.context;  
  
 **if** (!data1) {  
 addIsomorphicPromise.push(dispatch(loadData1()).then(() => {  
 **this**.setState({

...

});  
 }));  
 }  
}

1. **页面再次载入**

当第二次打开页面时，由于不会再请求数据，从而我们需要额外调用setState。不过好在，简单做一下封装变可以省去在constructor和componentWillMount重复出现setState：

componentWillMount() {  
 **const** { dispatch, data1 } = **this**.props;

**const** { addIsomorphicPromise } = **this**.context;  
  
 **if** (!data1) {  
 addIsomorphicPromise.push(dispatch(loadData1()).then(() => {  
 **this**.doSomeUpdate();  
 }));  
 } **else** {

**this**.doSomeUpdate();

}

}

doSomeUpdate = () => {

**const** { data1 } = **this**.props;

**this**.setState({

...

});

};

1. **componentWillReciveProps?**

好吧，这不是一个推荐的做法，但是我也同样把它列在这里。如果你对React组件的生命周期方法很熟悉的话。你会很容易想起有一个componentWillReceiveProps方法。该方法在props更新时会调用。所以，如果我们想偷懒，可以直接监听Props update然后再setState来更新组件的state。

但是大部分情况下，我们的组件不会只接收一个prop：

<MyComponent prop1={prop1} prop2={prop2} prop3={prop3} />

当存在多个props时，我们需要对props进行检查。省略没有必要的更新：

componentWillReceiveProps(nextProps) {

**if** (nextProps.prop1 !== this.props.prop1) {

**this**.setState({

...

});  
 }

}

（什么？你无所谓性能？当我什么都没有……）

**来自服务端的warning**

好了，当你完成这份代码。你会发现在控制台会打印出警告信息。

Warning: setState(...): Can only update a mounting component. This usually means you called setState() outside componentWillMount() on the server. This is a no-op. Please check the code for the Configuration component.

为何会发生这种情况呢？原因在于，当你在服务端渲染renderToString时，virtual dom tree已经完成渲染。这时当异步请求完成，调用setState已经无法生效。

对此，我们需要对环境进行检查：

**export const** isClient = !!(  
 **typeof** window !== 'undefined' &&  
 window.document &&  
 window.document.createElement  
);

如果是异步更新，那么只有处于client端才进行更新：

componentWillMount() {  
 **const** { dispatch, data1 } = **this**.props;

**const** { addIsomorphicPromise } = **this**.context;  
  
 **if** (!data1) {  
 addIsomorphicPromise.push(dispatch(loadData1()).then(() => {  
 **if** (isClient) **this**.doSomeUpdate();  
 }));  
 } **else** {

**this**.doSomeUpdate();

}

}

好了，当完成这些内容后。开始享受你的同构之旅吧！