如何用 React 构建前端架构

早期的前端是由后端开发的，最开始的时候仅仅做展示，点一下链接跳转到另外一个页面去，渲染表单，再用Ajax的方式请求网络和后端交互，数据返回来还需要把数据渲染到DOM上。写这样的代码的确是很简单。在Web交互开始变得复杂时，一个页面往往有非常多的元素构成，像社交网络的Feed需要经常刷新，展示内容也五花八门，为了追求用户体验需要做很多的优化。

当时说到架构时，可能会想前端需要架构吗，如果需要，应该使用什么架构呢？也有可能一下就想起了MVC/MVP/MVVM架构。在无架构的状态下，我们可以写一个HTML文件，HTML、Script和Style都一股脑的写在一起，这种原始的方式适合个人开发Demo用，或者只是做个玩玩的东西。

当我们写的Script和Style越来越多时，就考虑是否将这些片段代码整理放在同一个文件里，所以，我们就把Script写到了JS文件，把Style写到CSS文件。一个项目肯定有许多的逻辑功能，就需要考虑分层，把独立的功能单独抽取出来可以复用的组件。
Anguar和Ember作为MVC架构的框架，采用MVVM双向绑定技术能够快速开发一个复杂的Web应用。可是，我们不用。

React将自己定位为MVC的V部分，仅仅是一个View库，这就给我们很大的自由空间，并且引入了基于组件的架构和基于状态的架构概念。

MVC将Model、Controller和View分离，它们的通信是单向的，View只和Controller通信，Controller只跟Model交互，Model也只更新View，然而前端的重点在View上，导致Controller非常薄，而View却很重，有些前端框架会把Controller当作Router层处理。

MVP在MVC的基础上改进了一下，把Controller替换成了Presenter，并将Model放到最后，整个模型的交互变成了View只能和Presenter之间互相传递消息，Presenter也只能和Model相互通信，View不能直接和Model交互，这样又导致Presenter非常的重，所有的逻辑基本上都写在这里。

让我们想象一下国家电网，或者更接近我们的经常接触的领域——网络。网络有非常严格的定义，必须是有序的流，因为并不是所有连接到互联网的计算机都与其他计算机直接连接，它们通过路由节点间接连接。只有这样，网络才变得可以理解，因而易于管理。状态管理也是如此，状态的流动必须是有序的。

组件架构

你可以将组件视为组成用户界面的一个个小功能。我们要描述Gitchat的用户界面，可以看到Tabbar是一个组件，发现页的达人课是一个组件，Chat也是一个组件。这些组件中都包装在一个容器内，它们彼此独立又互相交互。组件有自己的结构，自己的方法和自己的API，组件也是可重用的。

有些组件还有AJAX的请求，直接从客户端调用服务端，允许动态更新DOM，而无需页面刷新。组件每个都有自己的接口，可以调用服务端并更新其接口。因为组件是独立的，所以一个组件可以刷新而不影响其他组件。React使用称为虚拟DOM的东西，它使用“diffing”算法来检测组件的更改，并且仅渲染这些更改，而不是重新渲染整个组件。在设计组件的时候最好遵循组件的结构中仅存在与单个组件有关的所有方法和接口。

虽然这种组件的架构鼓励可重用性和单一责任，但它往往会导致臃肿。MV*的目的是确保应用程序的每个层次都有各自的职责，而基于组件的架构目的是将所有这些职责封装在一个空间内。当使用许多组件时，可读性可能会降低。

React提供了两种组件，Stateful和Stateless，简单来说，这两种组件的区别就是状态管理，Stateful组件内部封装了State管理，Stateless则是纯函数式的，由Props传递状态。

```
class App extends Component {
  state = {
      welcome: 'hello world'
  }

  componentDidMount() {
    ...
```

我们先从一个简单的例子开始看，组件内部维护了一个状态，当状态发生变化是，会通知render更新。这个组件不仅带有State，还有和组件相关的Hook。我们可以使用这种组件构建一个高内聚低耦合的组件，将复杂的交互细节封装在组件内部。当然我们还可以使用PureComponent的组件优化，只有需要更新的时候才执行更新的操作。

```
const App = ({ welcome }) => (
  <div>{welcome}</div>
)
```

无状态的组件，状态由上层传递，组件纯展示，相比带状态的组件来说，无状态的组件性能更好，没有不必要的Hook。

```
import { observer } from 'mobx-react'

const App = observer(({ welcome }) => (
  <div>{welcome}</div>
))
```

observer函数实际上是在组件上包装了一层，当可观察的State改变时，它会更新状态以Props的形式传递给组件。这样的组件设计能够帮助更好可复用组件。

当我们拿到设计稿的时候，一开始需要做的事情就是划分一个个小组件，并且保证组件的职责单一，而且越简单越短小越好。并尽量保持组件是无状态的。如果需要有状态，也仅仅是内部关联的状态，就是与业务无关的状态。

状态架构

如果你之前用过jQuery或Angular或任何其他的框架，通常使用命令式的编程方式调用函数，函数执行数据更新。在使用React就需要调整一下观念。

```
import React, { Component } from 'react'
import { render } from 'react-dom'

class Counter extends Component {
  state = {
    counter: 0,
  }

  increment = (e) => {
    e.preventDefault()
    this.setState({ counter: this.state.counter++ })
  }

  decrement = () => {
     e.preventDefault()
    this.setState({ counter: this.state.counter-- })
  }

  render() {
    return (
      <div>
        <div id='counter'>{this.state.counter}</div>
        <button onClick={this.increment}>+</button>
        <button onClick={this.decrement}>-</button>
      </div>
    )
  }
}

render(<Counter />, document.querySelector('#app'))
```


组件自己管理的状态数据相关联的一个缺点，就是将状态管理与组件生命周期相耦合。如果某些数据存在于组件的本地状态中，那么它将与该组件一起消失，没有进一步保存数据，那么只要组件卸载，State的内容就会丢失。

组件的层次结构的在很大程度上取决于通过DOM布局，因为状态主要通过React中的

```
const counter = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}


const store = createStore(counter)

const Counter = ({
  value,
  onIncrement,
  onDecrement
}) => (
  <div>
    <div id='counter'>{value}</div>
    <button onClick={onIncrement}>+</button>
    <button onClick={onDecrement}>-</button>
  </div>
);

const App = () => (
  <Counter
    value={store.getState()}
    onIncrement={() => store.dispatch({ type: 'INCREMENT' })}
    onDecrement={() => store.dispatch({ type: 'DECREMENT' })}
  />
)

const renderer = render(<App />, document.querySelector('#app'))
store.subscribe(renderer)
```

```
  increment() { this.counter++ }

  decrement() { this.counter-- }
}

const store = new CounterStore();

const Counter = observer(() => (
  <div>
    <div id='counter'>{store.counter}</div>
    <button onClick={store.increment}>+</button>
    <button onClick={store.decrement}>-</button>
  </div>
))

render(<App />, document.querySelector('#app'))
```

Mobx觉得这种方式太麻烦了，为了更新一个状态居然要绕一大圈，它强调状态应该自动获得，只要定义一个可观察的State，让View观察State的变化，State的变化之后发出更新通知。在Redux里要实现一个特性，你需要更改至少4个地方。包括reducers、actions、组件容器和组件代码。Mobx只要求你更改最少2个地方，Store和View。很明显看到使用Mobx写出来的代码非常精简，OOP风格和良好的开发实践，你可以快速构建应用。


```
import React from 'react'
import { render } from 'react-dom'
import { types } from 'mobx-state-tree'
import { observer } from 'mobx-react'

const CounterModel = types
  .model({
      counter: types.optional(types.number, 0)
  })
  .actions(self => ({
      increment() {
        self.counter++
      }
```

```
render(<App store={store}/>, document.querySelector('#app'))
```

这种管理状态看起来很像MVVM的双向绑定，MST「Mobx State Tree」受Elm和SAM架构的影响，背后的思想也非常简单：

1. 稳定的参考态和直接可变的对象。也就是有一个变量指向一个对象，并对其进行后续的读取或写入，不用担心你正在使用旧的数据。
2. 状态为不可变的、结构性的共享树。

每次的操作，MST都会将不可变的数据状态生成一个快照，类似虚拟DOM的实现方案，因为React的render也只是比较差异再渲染的，所以开销并不会太大。

与MobX不同的是，MST是一种有架构体系的库，它对状态组织施行严格的管理。修改状态和方法现在由MST树处理。使用MobX向父组件注入树。一旦注入，树就可以用于父组件及其子组件。父组件不需要通过子组件将任何方法传递给子组件B。React组件根本不需要处理任何状态。子组件B可以直接调用树中的动作来修改树的属性。

异步方案

我们都知道Javascript的代码运行在主线程上，像DOM事件、定时器运行在工作线程上。一般情况下，我们写一段异步操作的代码，一开始可能就想到使用回调函数。

```
asyncOperation1(data1,function (result1) {
    asyncOperation2(data2,function(result2){
        asyncOperation3(data3,function (result3) {
            asyncOperation4(data4,function (result4) {
                // do something
            })
        })
    })
})
```

ES6语法中引入了generator，使用yeild和*函数封装了一下Promise，在调用的时候，需要执行next()函数，就像python的yield一样。

```
function* generateOperation(data1) {
    var result1 = yield asyncOperation1(data1);
    var result2 = yield asyncOperation2(result1);
    var result3 = yield asyncOperation3(result2);
    var result4 = yield asyncOperation4(result3);
    // more
}
```

ES7由出现了async/await关键字，其实就是在ES6的基础上把*换成async，把yield换成了await，从语义上这种方案更容易理解。

```
async function generateOperation(data1) {
    var result1 = await asyncOperation1(data1);
    var result2 = await asyncOperation2(result1);
    var result3 = await asyncOperation3(result2);
    var result4 = await asyncOperation4(result3);
    // more
}
```


在调用generateOperation时，ES6和ES7的异步方案返回的都是Promise。

传统的异步方案：

1. 嵌套太多
2. 函数中间太过于依赖，一旦某个函数发生错误，就不能继续执行下去了
3. 变量污染

使用Promise方案：

布很久之后才发布了它的CLI工具：create-react-app。屏蔽了和React无关的配置，如Babel、Webpack。我们可以使用这个工具快速创建一个项目。

用npm安装一个全局的create-react-app，`npm install -g create-react-app`，然后运行 `create-react-app hello-world`，就初始化好了一份React项目了，只要运行 `npm run start` 就能启动开发服务器了。

然而，复杂的项目必然需要自定义Webpack配置，create-react-app提供了eject命令，这个命令是不可逆的，也就是说，当你运行了eject之后，就不能再用之前的命令了。这是必经的过程，所以我们继续来看Webpack的配置。

Webpack核心配置非常简单，只要掌握三个主要概念即可：

1. entry 入口
2. output 出口
3. loader 加载器

entry支持字符串的单入口和数组/对象的多入口：

```
{
  entry: './src'
}
```



```
entry: { // pagesDir是前面准备好的入口文件集合目录的路径
  pageOne: './src/pageOne.js',
  pageTwo: './src/pageTwo.js',
}
```

output是打包输出相关的配置，它可以指定打包出来的包名/切割的包名、路径。

```
{
  output: {
```

```
            test: /\.(js|jsx)$/,
            loader: 'babel-loader',
          }
        }
    }
```

以及resolve，plugins配置提供更丰富的配置，create-react-app就是利用resolve的module配置支持多个node_modules，如create-react-app目录下的node_modules和当前项目下的node_modules。

现在再来看看Webpack的多入口方案，Webpack的output内置变量[name]打包出来的包名对应entry对象的key，用HtmlWebpackPlugin插件自动添加JS和CSS。如果使用CommonsChunk可以将Vendor单独剥离出来，这样多入口就可以复用同一个Vendor。

通常，我们一个项目有许多独立的子项目，使用Webpack的多入口方案就会造成打包的速度非常慢，不必要更新的入口也一起打包了。这时应该拆分为多个package，每个package是单入口也是独立的，这种方案称为monorepos「之前社区很流行将每个组件都建一个代码仓库，使用不太优雅的方案npm link在本地开发，这种方案的缺点非常明显，代码太分散，版本管理也是一个灾难」。

我们可以使用Monorepos的方案，将子项目都放在packages目录下，并且使用Lerna「实现Monorepos方案的一个工具」管理packages，可以批量初始化和执行命令。配合Lerna加Yarn的Workspace方案，所有子项目的node_modules都统一放在项目根目录。


```
{
  "lerna": "2.1.2",
  "commands": {
    "publish": {
      "ignore": ["ignored-file", "*.md"]
    }
  },
  "packages": ["packages/*"],
  "npmClient": "yarn",
  "version": "2.6.1",
```

2. 以组件为目录

   组件内需要的文件放在同一个目录下，如Alert和Notification可以建两个目录，目录内部有代码、样式和测试用例。

3. 以功能为目录

   如components、 containers、 stores按其功能放在一个目录内，将组件都放在components目录内，containers则是组装component。

团队协作

团队开发必然也会遇到一个问题，每个人写的代码风格都不一样，不同的编辑器也不尽相同。

有人喜欢双引号，也有人使用单引号，代码结尾要不要分号，最后一个对象要不要逗号，花括号放哪里，80列还是100列的问题。

还有更贱的情况，有人把代码格式化绑定在编辑器上，一打开文件就格式化了一下代码，如果他在提交一下代码，简直是异常灾难，花了半天写代码，又花了半天解决代码冲突问题。

像Go语言自带了代码格式化工具，使每个人写出来的代码风格是一致的，消除了程序员的战争。

前端也有类似的工具，Prettier配合ESLint最近在前端大受欢迎，再使用husky和lint-staged工具，在提交代码的时候就将提交的代码格式化。

Prettier是什么呢？就是强制格式化代码风格的工具，在这之前也有类似的工具，像Standardjs，这个工具仅格式化JS代码，无法处理JSX。而Prettier能够格式化JS和LESS以及JSON。

```
{
  "scripts": {
    "precommit": "lint-staged"
  }
}
```

这样，在提交代码时，就自动格式化代码，使每个开发者的风格强制的保存一致。

测试驱动

很多人都不喜欢写测试用例代码，觉得浪费时间，主要是维护测试代码非常的繁琐。但是当你尝试开始写测试代码的时候，特别是基础组件类的，就会发现测试代码是多么好用。不仅仅提高组件的代码质量，但是当发生依赖库更新，版本变化时，就能够马上发现这些潜在的问题。如果没有测试代码，也谈不上自动化测试。

前端有非常多的工具可以选择，Mocha、Jasmine、Karma、Jest等等太多的工具，要从这些工具里面选择也是个困难的问题，有个简单的办法看社区的推荐，React现在主流推荐使用Jest作为测试框架，Enzyme作为React组件测试工具。

我们做单元测试也主要关注四个方面：组件渲染、状态变化、事件响应、网络请求。

而测试的方法论，可以根据自己的喜好实践，如TDD和BDD，Jest对这两者都支持。

首先我们测试一个Stateless的组件

```
import React from 'react'
import { string } from 'prop-types'

const Link = ({ title, url }) => <a href={url}>{title}</a>

Link.propTypes = {
  title: string.isRequired
```

```
describe('Link', () => {
  it('should render correctly', () => {
    const output = shallow(
      <Link title="testTitle" url="testUrl" />
    )

    expect(shallowToJson(output)).toMatchSnapshot()
  })
})
```

在运行测试之后，Jest会创建一个快照。

```
exports[`Link should render correctly 1`] = `
<a
  href="testUrl"
>
  testTitle
</a>
`;
```