浅尝初试 React 技术栈

React

React 是什么？

Facebook 官方对 React 定义是：用来构建用户界面的库（Library）。注意到这里的用词是库（Library）而不是框架（Framework）。React 不像早期版本的 Angular 这样功能非常完备的 mvvm 框架，它主要只专注于解决 MVC 当中 V 层，也就是视图层（View）方面的问题。

不过我们也不必太过纠结库（Library）或框架（Framework）的定义。复杂的，给出你一整套解决方案的就叫框架（Framework）；简单的，专注解决一个问题并做到极致（Do one thing and do it well）的就叫库（Library）。

不过我们还是习惯性地称 React 是一个 JavaScript 框架，因为除了 React 核心库本身，在 React 的生态圈当中，还有很多其他可以搭配协同的工具库，比如在这次分享当中我们要介绍的用来解决状态管理问题的 Redux；用来提供前端路由功能的 react-router。我们把这些工具库统称为 React 技术栈，组合使用 React 技术栈也就完全撑得起一个框架提供的功能了。

React 有哪些特性？

声明式

说白了声明式就是你告诉程序你要一个什么样的东西的编写代码的方式。这也是在开发构建用户界面时最友好的方式。在 React 当中，你可以很轻松地告诉 React 你想要一个什么样的界面。我们使用一种叫做 JSX 的类似于 HTML/XML 的 JavaScript 语法扩展来和 React 交流：

```
<应用>
    <输入框></输入框>
    <按钮></按钮>
</应用>
```

这就好像我们可以直接对 React 说：

> 这里我要一个按钮！
> 这里我要一个表单！

是不是非常的直观明了呢？

组件化

在 React 当中，我们是以组件（Component）的概念来划分用户界面的。通常我们开发的页面都可以拆成一个个通用的组件，例如导航、表单、列表项、页脚等等。

使用 React 可以在很大程度上提高你代码的可复用性，编写页面就如搭积木一般简单：

```
<应用>
    <导航/>
    <注册表单/>
    <页脚/>
</应用>
```

这也同样意味着，我们除了可以在开发页面时复用自己编写的组件以外，还能把别人编写好的通用组件直接拿过来用。自定义一下样式，传个数据进去，组合起来，一个页面分分钟就搞定。

一次学习，随处编写

React 最强大的地方在于，其内部实现的虚拟DOM屏蔽了所有的底层实现，通过不同的渲染器（renderer），你编写的同一套代码可以用来构建包含 **浏览器/桌面操作系统/Android/iOS** 等几乎所有平台的用户界面。也就是说，掌握了 React 之后，你的能力将不止局限于写网页，而是可以在几乎所有的平台上开发用户界面。

这也就是为什么我们使用 React 的时候需要调用两个库 react 和 react-dom，react 库文件用来实现 React 的核心功能，react-dom 则用来把它渲染到浏览器当中。

目前已有的其他平台的解决方案还包括：

- React Native
- React VR
- React XP

例如在使用 React Native 的时候，我们同样是使用 react 核心库来实现基础功能，然后通过 react-native 库将我们编写的界面渲染到移动端上。

也就是说，有了 React 之后，我们可以用一种统一的描述方式来开发用户界面，至于在什么平台上实现，只要有相应的渲染器（renderer），我们就能够把我们开发的界面在对应的平台上面渲染出来。例如在美剧《西部世界》当中，React 甚至可以用来编写人工智能 Host 的故事线：

用 React 编写组件

React 组件

我们用 React 编写的代码绝大多数都是组件的代码。编写 React 组件需要遵循 React 内部的一系列规范，因此用 React 编写出来的应用自带前端工程化属性。不管新手还是老司机，只要是用 React 写组件，我们都能保证他写出来的代码是差不多的。这也就非常有利于一个项目组当中多个开发者之间进行协作。非常适合高级做架构，中级封组件，初级写业务的模式。

React 组件其实就相当于 JavaScript 当中的一种函数，接受应用数据作为为参数，内部进行一系列处理（包含事件处理函数、生命周期函数等，此处不展开讲），返回一个 React 元素。

React 元素

这里要注意到，React 组件和 React 元素是两个不同的概念。React 元素是 React 组件的一部分，也就是 React 组件返回的要拿来渲染的内容。

JSX

在 React 当中，我们通过一种叫做 JSX 的 JavaScript 语法扩展来描述 React 元素。

```
const title = <h1>Counter</h1>;
```

这里特别要注意的是，JSX 既不是原生的 HTML，也不是 jQuery 当中的字符串 `$('<h1>Counter</h1>')`，更不是 pug(jade) 当中的模板 `h1 Counter`。这是 React 内部自己的一套实现，可以允许你像写 HTML 一样，在 JavaScript 代码当中直接写页面，React 会在随后的渲染过程当中自动把 JSX 转译成页面当中真实的 DOM 元素。

函数定义组件 & 类定义组件

在 React 当中，有两种定义组件的方式。（注：在 react@15.6 当中已经废弃了 `createClass` 方法，如果你从来没用过 React 请自动忽略）

函数定义组件

比较简单的一些，只接受外部传入的数据的组件，我们一般通过函数定义的方式来编写：

```
var Button = function(props) {
    return <button onClick={props.onClick}>+</button>;
}

// 当然也可以用 ES6 的 箭头函数 arrow function

const Number = ({ number }) => <p>{number}</p>;
```


props & state

上述示例当中的 props 就是组件数据的一种。在 React 当中，最常用的组件数据有两种：props 和 state.

其中 props 是从外部传入的，内部无法修改，用来渲染展示的数据。

而 state 则是组件内部维护，可以跟随应用状态改变而改变的数据（例如用户输入的表单项）。

类定义组件

比较复杂的，需要处理事件，调用声明周期函数，与服务器交互数据的组件，我们通过类定义组件的方式来声明：

```
// 从 React 库当中获取组件的基础支持
const { Component } = React;
// 使用 ES6 当中的 class 关键字来声明组件
class Container extends Component {
    /* 类中的构造方法，调用super方法来确保我们能够获取到this，组件自身的
```

```
state 数据也在构造方法当中初始化。*/
    constructor() {
        super();
        this.state = {
            number: 0
        }
    }
    /* 事件处理方法，在 React 当中我们通过调用 `setState` 方法来修改
state 数据，这样才能出发组件在界面当中自动重新渲染更新 */
    handleClick() {
        this.setState({number: this.state.number+1});
    }
    // 渲染方法，返回 React 元素
    render() {
        return (
            <div>
                <Title />
                <Number number={this.state.number} />
                <Button onClick={() => this.handleClick()} />
            </div>
            );
    }
}
```


展示组件 & 容器组件

在本文的开头我们已经介绍过了，React 是一个视图层的框架，也就是说它只有 V，而真正在编写前端代码的时候，除了页面展示的内容以外，我们还需要进行处理用户输入、验证表单、和服务器进行数据交互之类的操作。

那么在实际的编码过程当中，我们要如何解耦这些应用的业务逻辑和用户界面的结构样式呢？

这时我们就需要引入一组展示组件和容器组件的概念。

展示组件

- 主要负责用户界面的结构和样式。
- 从 props 接收父组件传递来的数据。
- 大多数情况可以通过函数定义组件声明。

容器组件

- 主要负责组件如何交互，业务逻辑等。
- 拥有自身的state，从服务器获取数据，或与 redux 等其他数据处理模块协作。

- 需要通过类定义组件声明，可以包含生命周期函数、事件处理函数等。

例如：

```
// 展示组件
const Button = props => <button onClick={props.onClick}>+
</button>;
// 容器组件
class Counter extends Component {
  constructor() {
    super();
    this.state = {
      number: 0
    }
  }

  handleClick() {
    this.setState({number: this.state.number + 1});
  }

  render() {
    return (
      <div>
        <Title />
        <Number number={this.state.number} />
        <Button onClick={() => this.handleClick()} />
      </div>
    )
  }
}
```


Redux

Redux 又是什么？

在上述 React 部分的介绍当中，我们已经提到了，React 用来处理数据的方式主要有 props 和 state 两种（另外还有一种不常用的 context）。

其中的 props 必须是从父组件传递到子组件，如果嵌套层级很多，props 必须逐级从保存数据的组件层层传递到使用 props 的组件当中。而 state 在使用的时候，必须通过调用 `this.setState()` 方法，在改变 state 值的同时，触发 React 组件运行的生命周期，来触发界面的更新。 `this.setState()` 方法可以传递数据、方法、回调函数。在同一次操作中，连续调用多次 `this.setState()` 方法也会造成许多难以预料的结果，仅仅通过看代码你很难判断出最后值会变成什么。

而我们使用 React 开发界面的主要场景是在Web应用当中，不同于传统的以内容为主的网页。Web应用涉及到非常多的状态数据的改变，包括用户的交互、服务器通信、界面的动画、样式的改变等等内容。

我们在开头也提到了，React 是一个专注于视图层的库，在数据的改变，状态管理方面，它并没有做得很好。因此，当我们的应用复杂到一定程度时，就需要引入一些其他的工具库来帮助我们解决状态管理的问题。

什么是状态管理

```
Component(state) = View
```

我们通过一个简单的公式来说明这个问题。在 React 的理念当中，组件其实就是一个方法，我们向组件方法传入数据得出要渲染的视图内容。

这里之所以把传入的数据称为 **状态(state)**，是因为在一个应用当中，许多数据都是处于变化当中的，根据用户的不同操作响应发生改变（比如说在我们计数器的示例当中，点击按钮，计数器的数字就会随之改变增加）。

也就是说，我们看到的视图，网页的内容，如果把应用状态数据不同的改变不同的时刻看作是一段动画的话，页面在某一刻显示的内容其实就是动画的某一帧。

所以，我们在开发构建界面时，除了界面的样式和逻辑以外，如何处理状态数据就成了另外一个我们需要主要关注的问题。这一问题的解决方案，自然也就叫做状态管理了。

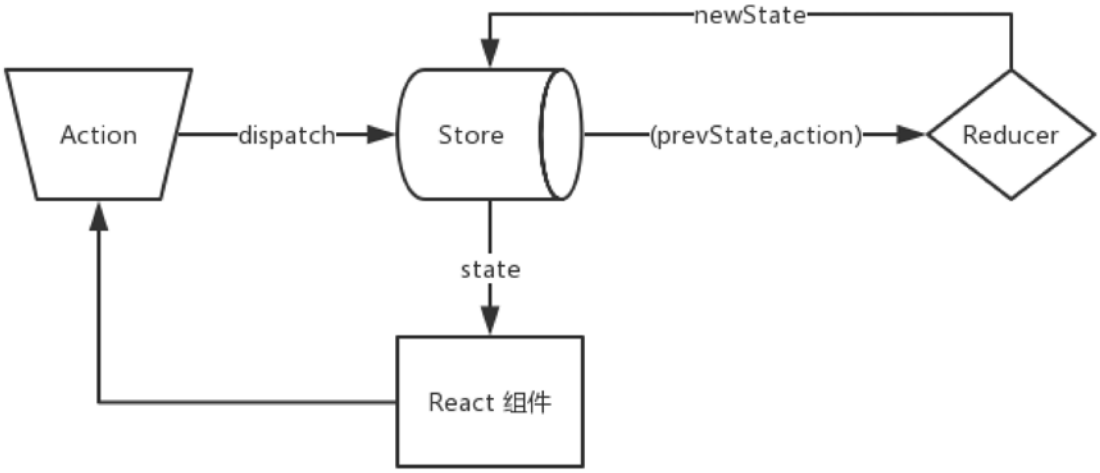
用 Redux 管理应用的状态数据

React 应用的开发理念告诉我们，在一个应用当中，如果有两个组件需要使用同一数据，那么我们需要把这一组数据提升到它们共同的父组件当中保存；在实际开发当中，应该尽量控制有状态组件（含有 state 的组件）的数量。

在一个Web应用当中，会涉及到显示数据的增删改查、服务器数据获取、界面切换显示内容等各种各样类型的状态数据改变。

那么我们为什么不把所有的状态数据改变，用一种统一的方式描述；既然要控制有状态组件的数量，那么我们为什么不干脆直接把一个应用的所有状态数据存储在一个统一的地方集中管理？

这也就是 Redux 的理念。



Action

Action 就是我们上述的，用统一的形式，描述所有改变应用状态数据的操作的方法。说白了，它其实就是一个带有 `type` 属性的 Javascript 对象：

```
{
  type: 'INCREMENT',
  value: 1
}
```


例如在我们的计数器当中，点击按钮数字增加1的操作可以用上述格式内容的对象来描述表示。Redux 对 Action 的要求并不是非常严格，你只需要保证它包含 `type` 属性，其余的内容完全由你自己决定。当然如果你希望你制定的 Action 更加符合规范，可以遵循 Flux Standard Action 标准。

Reducer

Reducer 则是 Redux 的设计理念当中最核心的方法，它接受当前的状态数据以及触发的 Action 作为参数，根据内部 `switch` 结构的逻辑判断，返回一个新的状态数据：

```
(previousState, action) => newState
```

例如在我们的计数器当中，可以抽象编写出这样一个 Reducer 方法：

```
function counter(state = 0, action) {
    switch (action.type) {
      case 'INCREMENT':
        return state + action.value
      default:
```

```
            return state
        }
    }
```

我们可以看到 counter 函数接受 state 和 action 两个参数，返回值则是经过一个 switch 结构判断的新的 state 数据。这样结构的函数也就是我们在使用 Redux 时编写的 Reducer 方法。

这是 Redux 理念当中最核心的一个部分，它决定了一个应用当中的状态数据在不同的 Action 被触发时具体会如何改变。

有关 Reducer 的更详细解释，可以参阅我之前发表的 Redux 中的 reducer 到底是什么，以及它为什么叫 reducer 一文。

Store

Store 则是 Redux 当中我们用来存储状态数据的地方，它提供了3个主要的方法：

- 用来获取当前状态数据的 `getState()`
- 用来触发应用 action 动作的 `dispatch(action)`
- 用来订阅响应事件（state改变之后进行的操作）的 `subscribe(listener)`

而在使用 Redux 时，我们可以通过它提供的 `createStore` 方法，直接从 reducer 函数生成对应的 store：

```
const { createStore } = Redux;

const store = createStore(counter);
```

在 React 应用当中使用 Redux

我们可以直接在 React 项目当中使用 Redux:

```
// 把之前 React 的渲染函数命名为 render
const render = () => {
  /* 传入 store.getState() 获取 Redux 当中存储的状态数据
   * 传入 store.dispatch() 方法来执行对应 action 修改状态数据
   */
  ReactDOM.render(<Counter
              number={store.getState()}
              onIncrement={() => store.dispatch({
                          type: 'INCREMENT',
                          value: 1
```

```
                        })}
                />,
                document.getElementById('root'));
}
// 调用一次 render 方法进行初次渲染
render()
// 使用 store.subscribe 方法订阅 render 这样每次 store.dispatch 方法
触发时就会自动调用 render
store.subscribe(render);
```

react-redux

当然，每次 Redux 当中的状态数据改变时都强制执行 ReactDOM 的 render 方法并不是最优选择。事实上，社区已经开发出了一个名为 react-redux 的库专门来辅助我们对 React 和 Redux 进行协同使用。

```
/* Provider 充当为整个 React 应用传入 Redux 当中 store 的容器组件
 * connect 用来为需要使用 store 的组件提供相应的状态数据或 dispatch 方法
 */
const { Provider, connect } = ReactRedux;
/* 我们通过 mapStateToProps 来将 Redux 当中的状态数据映射到 React 相应
的 props 当中 */
const mapStateToProps = state => ({
  number: state
});

class Counter extends Component {
  constructor(props) {
    super(props);
  }

  handleClick() {
    // 在这里调用传入组件的 dispatch 方法
    this.props.dispatch({
      type: 'INCREMENT',
      value: 1
    });
  }

  render() {
    return (
      <div>
        <Title />
        <Number number={this.props.number} />
        <Button onClick={() => this.handleClick()} />
      </div>
    )
  }
}
```

```
/* 我们需要通过 connect 方法来包装一下 React 的 Counter 组件，使其获取到
Redux 的 store 当中的方法和数据 */
Counter = connect(mapStateToProps)(Counter);
```

react-router

react-router 是什么？

react-router 是 React 生态圈当中前端路由功能的实现。它最大的特点是可以不用添加额外的路由配置文件，像使用所有其他 React 组件的方式一样，只需要引入几个组件就可以轻松为你的 React 应用添加前端路由的功能。

什么是前端路由？

我们都知道，在传统的网站当用，一个 URL 就对应着某个特定的页面。当我们在浏览器地址栏当中输入这个 URL 的时候，浏览器就会从网站的服务器请求该页面，获取相应的内容。

而在Web应用的开发当中，我们可以通过操纵浏览器暴露给我们的 history 接口以及异步服务器数据请求等方式，在前端就实现路由的切换，而不需要每次都让服务器后端解析 URL 路由请求再返回内容。

使用前端路由可以很大程度上提升Web应用，尤其是单页面应用的使用体验。

在 React 应用当中使用 react-router

react-router 的使用非常简单，它目前已经发行到了 v4 版本，而之前的3个版本在网络上也能找到非常多的应用。在这里我们仅拿最新的版本作为示例。在 react-router@4 版本当中，专门为Web端提供了高度封装好的 react-router-dom 库，这下我们几乎不需要任何的配置就可以直接使用前端路由功能了：

```
/* 这里引入的3个方法全部都是封装好的 React 组件，使用方法和其他 React 组件
几乎没有任何差别 */
const { HashRouter, Route, Redirect } = ReactRouterDOM;
```

HashRouter

`HashRouter` 为我们的应用提供了 hash 形式（也就是带#的路由）路由的功能支持。

```
{/* 在通常情况下，我们不需要为 HashRouter 进行任何设置，直接引入使用即可。
*/}
<HashRouter>
  <App/>
</HashRouter>
```

主要到在 react-router 提供的所有类型的 `Router` 组件当中，第一级的子组件有且只能有一个。因此我们在使用的时候，通常在我们应用组件的最外层包裹上一个 `<div>` 标签：

```
<HashRouter>
  <div>
    ...
  </div>
</HashRouter>
```

这里为了方便在线演示，所以我们使用了 `HashRouter` 组件，在实际的开发当中，更经常使用的是 `BrowserRouter` 组件，它可以为我们提供不带 `#` 的前端路由支持，更加友好。

Route

前端路由的主要功能就是通过判断不同的浏览器地址显示不同的内容，那么具体某个路由地址要怎么展示某个组件呢？

这就是 `Route` 组件为我们提供的功能：

```
<HashRouter>
    <div>
      <Route path='/:title?' component={App} />
    </div>
</HashRouter>
```

其中的 path 属性用来设置匹配的目标路由地址，路由地址可以是固定的字符串，例如 home/about/user 之类的，也可以像我们示例中一样，以冒号开头将路由的地址作为参数，之后我们可以在组件当中获取到对应的路由参数（以 ? 结尾则表示这一参数是可选的）：

```
const Title = props => <h1>{props.title}</h1>;

const App = ({ match }) => (
  <Provider store={store}>
    <Counter title={match.params.title} />
```

```
    </Provider>
  );
```

这里的 `match.params.title` 也就是我们路由参数当中对应的值了。

Link

有了前端路由的内容，我们还需要相应的前端路由的导航。前端路由导航的主要功能是实现浏览器地址栏 URL 的切换，并触发Web应用展示对应的内容，而不是像原生的 HTML 超链接试图向服务器发起对应 URL 的请求。

react-router 同样为我们提供了现成的 `Link` 导航组件：

```
<HashRouter>
    <div>
      <ul>
        <li><Link to='/react'>react</Link></li>
        <li><Link to='/redux'>redux</Link></li>
        <li><Link to='/react-router'>react-router</Link></li>
      </ul>
      <Route path='/:title?' component={App} />
    </div>
</HashRouter>
```


备注

使用到所有库的链接

- react
- redux
- react-redux
- react-router

使用库的方式

直接在浏览器中使用

为了方便我们在线演示，更快地直接上手，在本文的示例当中，我们均采用了直接在浏览器当中使用这些库的方法。在 Codepen 示例当中，我已经事先引入了所有库的 CDN 文件，这些库都会向页面暴露一个全局的对象，然后我们可以通过解构赋值的方式，获取到对象当中我们要使用的方法，例如：

```
const { Component } = React;
```

如果你是在本地进行练习，也可以通过 `<script>` 标签引入相应库的 CDN 文件，之后通过相同的方式进行调用。

> P.S. 如果你使用最新版的 Chrome 进行调试，这些 ES6 的新特性都可以直接在浏览器当中运行，无需编译。

通过 npm 来使用

在正式的开发项目当中，我们会使用 npm 来管理安装各个库，之后通过 `import` 的方式来调用：

首先安装：

```
npm install react react-dom --save
```

然后调用：

```
import React from 'react';
```


分享源码完整示例

本文所有代码完整示例可以在 GitChat React Examples 在线查看调试。包含：

- React 版计数器
- React+Redux 版计数器
- React+Redux+react-redux 版计数器
- React+Redux+react-redux+react-router 版计数器

学习资源推荐

文档

- React 中文文档
- Redux 中文文档

知乎专栏

- 从零学习React技术栈
- LeanReact
- pure render

编辑器插件

- VS Code React 技术栈代码补全插件