React 入门

React 的核心思想是：封装组件。

各个组件维护自己的状态和 UI，当状态变更，自动重新渲染整个组件。

基于这种方式的一个直观感受就是我们不再需要不厌其烦地来回查找某个 DOM 元素，然后操作 DOM 去更改 UI。

React 大体包含下面这些概念：

- 组件
- JSX
- Virtual DOM
- Data Flow

这里通过一个简单的组件来快速了解这些概念，以及建立起对 React 的一个总体认识。

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

export default class HelloMessage extends Component {
    static defaultProps = {
        name:'zhang san'
    }
    render() {
        return <div>Hello {this.props.name}</div>;
    }
}
```

> *组件*

React 应用都是构建在组件之上。

上面的 HelloMessage 就是一个 React 构建的组件

props 是组件包含的两个核心概念之一，另一个是 state（这个组件没用到）。可以把 props 看作是组件的配置属性，在组件内部是不变的，只是在调用这个组件的时候传入不同的属性（比如这里的 name）来定制显示这个组件。

> *JSX*

从上面的代码可以看到将 HTML 直接嵌入了 JS 代码里面(***实际上是一种语法糖***)，这个就是 React 提出的一种叫 JSX 的语法，这应该是最开始接触 React 最不能接受的设定之一，因为前端被“表现和逻辑层分离”这种思想“洗脑”太久了。但实际上组件的 HTML 是组成一个组件不可分割的一部分，能够将 HTML 封装起来才是组件的完全体，React 发明了 JSX 让 JS 支持嵌入 HTML 不得不说是一种非常聪明的做法，让前端实现真正意义上的组件化成为了可能。

好消息是你可以不一定使用这种语法，后面会进一步介绍 JSX，到时候你可能就会喜欢上了。现在要知道的是，要使用包含 JSX 的组件，是需要“编译”输出 JS 代码才能使用的，之后就会讲到开发环境。

> *Virtual DOM*

当组件状态 state 有更改的时候，React 会自动调用组件的 render 方法重新渲染整个组件的 UI。

当然如果真的这样大面积的操作 DOM，性能会是一个很大的问题，所以 React 实现了一个Virtual DOM，组件 DOM 结构就是映射到这个 Virtual DOM 上，React 在这个 Virtual DOM 上实现了一个 diff 算法，当要重新渲染组件的时候，会通过 diff 寻找到要变更的 DOM 节点，再把这个修改更新到浏览器实际的 DOM 节点上，所以实际上不是真的渲染整个 DOM 树。这个 Virtual DOM 是一个纯粹的 JS 数据结构，所以性能会比原生 DOM 快很多。

> *Data Flow*

“单向数据绑定”是 React 推崇的一种应用架构的方式。当应用足够复杂时才能体会到它的好处，虽然在一般应用场景下你可能不会意识到它的存在，也不会影响你开始使用 React，你只要先知道有这么个概念。

配置环境

上面简单介绍一下react及其特性，下面以一个经典todolist的例子来介绍react基本语法。

安装环境：

1. 安装 node
2. npm install -g create-react-app
3. create-react-app my-app
4. cd my-app/
5. npm start

接着浏览器会默认打开 `http://localhost:3000/`，这样环境就配置成功了

运行效果

设计组件层次结构

- TodoList 用于显示 todos 列表。
    - todos: Array 以 { text, completed } 形式显示的 todo 项数组。
    - onTodoClick(index: number) 当 todo 项被点击时调用的回调函数。
- Todo 一个 todo 项。
    - text: string 显示的文本内容。
    - completed: boolean todo 项是否显示删除线。
    - onClick() 当 todo 项被点击时调用的回调函数。
- AddTodo 增加一个 todo 项。
    - onAddClick: 增加todo方法。
- Footer 一个允许用户改变可见 todo 过滤器的组件。
    - onFilterChange: 改变filter的方法
    - filter： filter类型常量

- App 根组件，渲染余下的所有内容。

编写代码

> *编写 Todo组件*

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'; //最新的react已经把PropTypes剔除，需要从这个包里导入

// es6语法导出一个Todo组件
export default class Todo extends Component {
    render() {
        return (
            <li
                onClick={this.props.onClick}
                className="todo"
                style={{
                    textDecoration: this.props.completed ? 'line-through' : 'none',
                    cursor: this.props.completed ? 'default' : 'pointer'
                }}>
                {this.props.text}
            </li>
        )
    }
}
//Typechecking With PropTypes
Todo.propTypes = {
    onClick: PropTypes.func.isRequired,
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
}
```

注意点：

- css class属性在jsx代码里需要传入的属性名称为classname，因为class是js的关键字
- `onClick={this.props.onClick}` jsx中传入属性需要用大括号
- `style={{....}}` jsx中直接写样式代码，需要用两个大括号，第一个大括号代表需要传入的属性，第二个大括号代表传入的是一个对象
- 引入PropTypes做typecheck，需要写`import PropTypes from 'prop-types'`， react16已经从把这部分代码抽成单独的库

> *编写 TodoList组件*

```
import React, { Component } from 'react'
import PropTypes from 'prop-types';
import Todo from './Todo'

export default class TodoList extends Component {
    render() {
        return (
            <ul>
                {this.props.todos.map((todo, index) =>
                    <Todo {...todo}
                        key={index}
                        onClick={() => this.props.onTodoClick(index)} />
                )}
            </ul>
        )
    }
}

TodoList.propTypes = {
    onTodoClick: PropTypes.func.isRequired,
    todos: PropTypes.arrayOf(PropTypes.shape({
        text: PropTypes.string.isRequired,
        completed: PropTypes.bool.isRequired
    }).isRequired).isRequired
}
```

注意点：

- jsx里map遍历的时候，一定要传key值，并且不能重复，在业务代码里，不推荐直接传index，这里仅做演示

> *编写 Footer组件*

```
import React, { Component } from 'react'
import PropTypes from 'prop-types';

export default class Footer extends Component {
    renderFilter(filter, name) {
        if (filter === this.props.filter) {
            return name
        }

        return (
            <a href='#' onClick={e => {
                e.preventDefault()
                this.props.onFilterChange(filter)
            }}>
                {name}
            </a>
        )
    }

    render() {
        return (
            <p>
                Show:
        {' '}
                {this.renderFilter('SHOW_ALL', 'All')}
                {', '}
                {this.renderFilter('SHOW_COMPLETED', 'Completed')}
                {', '}
                {this.renderFilter('SHOW_ACTIVE', 'Active')}
                .
      </p>
        )
    }
}

Footer.propTypes = {
    onFilterChange: PropTypes.func.isRequired,
    filter: PropTypes.oneOf([
        'SHOW_ALL',
        'SHOW_COMPLETED',
        'SHOW_ACTIVE'
    ]).isRequired
}
```

注意点：

- a标签里拦截了默认的跳转，转而执行自定义的js代码

> *编写AddTodo组件*

```
import React, { Component } from 'react'
import PropTypes from 'prop-types';

export default class AddTodo extends Component {
    render() {
        return (
            <div>
                <input type='text' ref='input' />
                <button onClick={(e) => this.handleClick(e)}>
                    Add
                </button>
            </div>
        )
    }

    handleClick(e) {
        const node = this.refs.input
        const text = node.value.trim()
        this.props.onAddClick(text)
        node.value = ''
    }
}

AddTodo.propTypes = {
    onAddClick: PropTypes.func.isRequired
}
```

注意点：

- React 提供了一个特殊的属性`refs`，可以是字符串，也可以是function，当ref传入一个方法的时候，方法的参数是一个类的实例或者是一个DOM节点。上面AddTodo组件通过refs获取输入框节点，直接对节点进行赋值操作

> *编写 TodoApp组件*

```
import React, { Component } from 'react'
import PropTypes from 'prop-types';
import AddTodo from './component/AddTodo'
import TodoList from './component/TodoList'
import Footer from './component/Footer'

//filter 常量
const VisibilityFilters = {
    SHOW_ALL: 'SHOW_ALL',
    SHOW_COMPLETED: 'SHOW_COMPLETED',
    SHOW_ACTIVE: 'SHOW_ACTIVE'
};
export default class TodoApp extends Component {
    constructor(props) {
        super(props);
        this.state = {
            visibilityFilter: VisibilityFilters.SHOW_ALL,
            visibleTodos: []
```

```
        }

    }
    addTodo(text) {
        var nextVisibleTodos = [...this.state.visibleTodos];
        nextVisibleTodos.push({
            text: text,
            completed: false
        })
        this.setState({
            visibleTodos: nextVisibleTodos
        })
    }
    completeTodo(index) {
        var nextVisibleTodos = [
            ...this.state.visibleTodos.slice(0, index),
            Object.assign({}, this.state.visibleTodos[index], {
                completed: true
            }),
            ...this.state.visibleTodos.slice(index + 1)
        ]
        this.setState({
            visibleTodos: nextVisibleTodos
        })

    }
    setVisibilityFilter(nextFilter) {
        this.setState({
            visibilityFilter: nextFilter
        })
    }
    render() {
        const { visibleTodos, visibilityFilter } = this.state
        var todos = []
        switch (visibilityFilter) {
            case VisibilityFilters.SHOW_ALL:
                todos = visibleTodos;
                break;
            case VisibilityFilters.SHOW_COMPLETED:
                todos = visibleTodos.filter(todo => todo.completed);
                break;
            case VisibilityFilters.SHOW_ACTIVE:
                todos = visibleTodos.filter(todo => !todo.completed);
                break;
        }
        return (
            <div>
                <AddTodo
                    onAddClick={text =>
                        this.addTodo(text)
                    } />
                <TodoList
                    todos={todos}
                    onTodoClick={index =>
                        this.completeTodo(index)
                    } />
                <Footer
```

```
                    filter={visibilityFilter}
                    onFilterChange={nextFilter =>
                        this.setVisibilityFilter(nextFilter)
                    } />
            </div>
        )
    }
}
```

注意点：

- TodoApp负责把组件组织成一个页面，后面会调用渲染方法渲染到对应节点中去
- `Object.assign({}, ...),` 做一个浅拷贝生成一个新的对象

总结

整个demo比较简单，如果跟着代码把TodoApp写一遍，基本可以入门，但是基本生命周期管理还未涉及，可以参照官方文档学习https://reactjs.org/docs/state-and-lifecycle.html

demo源码地址