《React 精髓》虽然没有最新的Reduce，但是还是值得一看。

一、给项目安装必要的工具

React牛在何处？--React将命令式的API包装成为声明式的API，React牛在改变了我们的编写代码的方式；

声明式编程的代码要少于命令式编程--声明式是告诉计算机要去做什么而不管怎么做，而命令式是要描述如何去做。jQuery和JavaScript DOM API就是典型的命令式编程。

Snapterest项目：

核心功能

- 实时的搜集和展示推文
- 在集合中添加和删除推文
- 查看搜集的推文
- 将推文导出一段html代码来分享

Snapterest项目功能划分

- 1 实时的从snapkite引擎服务器接收推文
- 2 每条推文显示1.5秒
- 3 通过用户点击事件，将推文添加到集合中
- 4 显示集合中的推文
- 5 为集合创建HTML代码并导出
- 6 通过用户点击删除集合中的推文

安装node.js npm git， 从twitter streaming API中获得数据

应用的账号与密码：

```
Consumer Key (API Key)  PopNtZ4b0b9WCxMI4GjtTwUbF

Consumer Secret (API Secret)    LIrKVl8XXtBjSQc95UIe2FfEPbr0Puc1YqE1vn9kko5CI9RXLL

Access Token

Access Token    828801733111328768-iey9hh8Z9fvBGLmUSApZMyymDTKoTr3

Access Token Secret CqgpQmiohppheMmwV9q72tOHtR9rSW7MLDzf8cWnoazHd

Access Level    Read and write
Owner   gpxiao118
Owner ID    828801733111328768
```

使用snapkite引擎库`https://github.com/snapkite/snapkite-engine.git`

使用snapkite过滤器filter

```
https://github.com/snapkite/snapkite-filter-is-possibly-sensitive.git
https://github.com/snapkite/snapkite-filter-has-mobile-photo.git
https://github.com/snapkite/snapkite-filter-is-retweet.git
https://github.com/snapkite/snapkite-filter-has-text.git
```

项目到上线前要构建，构建工具（gulp,grunt）将源文件与项目所依赖的包转换为单个文件，交给浏览器执行；在构建中重要的一环是打包，我们需要打包(webpack,browserify)哪些文件,复用哪些组件与Node.js模块；

一、创建第一个**React**应用；

理解虚拟DOM

为什么要操作dom，现在的网页应用都是动态的，我们要关注两种事件，

- 用户的交互事件：点击，滚动，键盘按下，页面缩放等，
- 服务端事件：应用程序从服务端接收数据或者错误等；

这些事件的触发都会反映到浏览器绘制的用户界面UI上来，这些事假的触发，我么通常会更新应用程序所依赖的数据，相应的数据模型发生了改变，我们通常想通过更新UI来告诉用户的这些变化；

现在很流行的方式是数据双向绑定，即UI的状态与数据模型状态同步：其中一种是键值观察（KVO）模式（Ember,Knockout,Backbone）,还有就是脏检查，Angular。

而React采用的不是这种方案而是虚拟dom（真实dom的一个快速，仅存于内存中的映射）。

- 只要数据模型的状态发生改变，React就会将UI重新绘制为虚拟dom的形式
- React会计算出数据改变前后两个虚拟dom的差别，两个虚拟dom的区别就是真实dom需要实际更新的部分；

有`reactFragment，reactNode`的数组，有`reactElement,reactText`.当然还提供了`React.CreateFactory(type)`来生成lis这样的繁琐元素。

React初始页面加载时慢？使用React后，页面不用再创建静态的web页面，当浏览器加载时，只是加载了一些容器元素或者是父元素，然而其他的dom元素都是用JavaScript来创建的，为此往往需要从服务器端请求额外的数据。接收到额外的数据之后才会更新dom，但是请求数据是要时间的。为此可以采用服务端进行dom更新操作

dom依赖JavaScript，而node.js为JavaScript提供了服务器运行环境，通过`ReactDOMServer.renderToString(ReactElement)`渲染出初始的页面，`ReactDOMServer.renderToStaticMarkup(ReactElement)`生成静态页面；

三、创建第一个React组件；

react组件与状态。假设用一个简单的状态机来表示用户界面，那么用户行为将会触发状态机状态的改变，每个状态用不同的react元素表示，那么这个状态机就是React组件。

React组件的state属性应该用来存储事件处理函数随时可能发生改变的数据，已达到重新渲染组件的目的；应该保持在state对象中组件状态的最小可能表示形式，无论在state中放了什么都要自己手动更新，而render（）函数中放的东西，都会通render（）自动更新。

在事件处理中，默认React会在冒泡阶段触发事件处理函数，但可以在事件处理函数名后添加Capture来告诉React，在捕获阶段触发他们。如onClickCapture。此外React在SyntheticEvent对象上包装了浏览器的原生事件，这意味着可以照常使用stopPropagation()和preventDefault()方法，如果要访问浏览器的原生事件，可以通过nativeEvent属性来做，如需要处理touch事件，可以简单的调用React.initializeTouchEvets(true).同时React不会将事件处理函数渲染到真实dom中，而是React在顶层使用一个监听函数来监听事件，并把他们代理到相对应的事件实例函数。

四、让React组件变得可响应；

项目设计中，可以从顶层React元素开始，然后实现它的子组件，自顶向下来构建组件的层级

组件的写法：

- 引入依赖模块
- 定义React组件
- 作为模块导出React组件

子组件更新父组件的机制

- 父组件传递一个回调函数作为子组件的属性。
- 每当子组件想要更新父组件的state时，它就会调用这个回调函数并传递必要的数据到父组件的新状态中
- 父组件更新它的state，触发render()函数重新渲染所有有必要更新的子组件；

React单向数据流绑定，父组件通过getInitialState设定默认数据，数据作为属性传递给子组件，子组件再通过回调函数，通过访问state更新父组件的数据。这种数据流单向流动的模式，有助于增加组价的数量，而不增加页面的复杂度；但是这种数据流的层层传递会浪费很多冗余的数据，但是容易调试。要优化这种方案有很多，Flux就是其中的一种。

五、结合其他库来使用React；

在上面的功能划分中，第一条是React做不到的，这和UI没什么关系，所以可以结合其他库来实现功能，使用snapkite引擎；

安装snapkite数据流模块

```
npm install --save snapkite-stream-client
```

React组件的生命周期

创建期

- getDefaultProps():
- getInitialState():
- componentWillMount():
- render():
- componentDidMount():此阶段好和外部JavaScript库，发送AJax请求，设置定时器setInterval(),setTimeout();

销毁期

- componentWillUnmount():清理一些数据和不需要的业务逻辑

六、更新React组件；

存在期

- componentWillReceiveProps():
- shouldComponentUpdate():
- componentWillUpdate():
- render():
- componentDidUpdate():

将Css样式作为JavaScript对象内联到React组件中，好处是

- 移植：移植性比较好，在同一个JavaScript文件中可以将样式都分享出去。
- 封装：将样式内联可以限制它的影响范围
- 灵活：可以用JavaScript来计算Css

缺点也很明显，CSP内容安全策略会阻止内联样式生效。

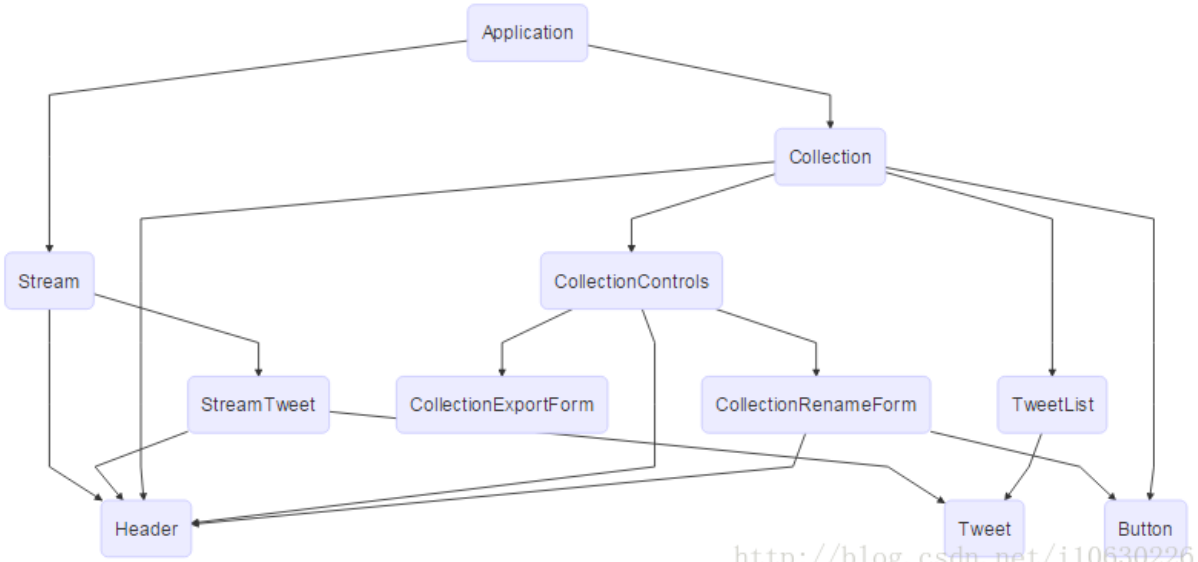
通过ReactDOMServer生成的组件HTML字符串，可以看出react的灵活性，既可以使用相同的React组件来渲染DOM元素，也可以生成HTML字符串传递给第三方API。

React render（）渲染虚拟DOM只能是一个根节点，如果要渲染多个，则要用一个父容器包裹一下，如div.

React通过组件的propType对象来验证组件的属性。

七、构建复杂的React组件；

父子组价 间的数据传递。管理状态是关键。



八、使用Jest来测试React应用组件；

许多自信的React开发人员都会觉得自己的React组件没有什么重大缺陷，这也导致了一个误区：不写单元测试。

怎么针对JavaScript函数写单元测试呢？可以借助很多单元测试框架--Jest(Facebook出品)，它是有测试框架Jasmine演变而来的。

通过模拟对象来进行测试，叫Mocking(模拟)，Jest会扫描**tests**目录下的所有测试用例，并执行。

在Jest中describe定义一个测试套件，接收两个参数

- 套件名称，描述即将被测试的功能；
- 套件实现，实现该套件的函数；

一个单独的测试称为测试规范（specs）,通过全局函数it()定义，和describe()函数类似，接收两个参数

- 测试规范名称：描述测试规范的功能
- 测试规范的实现：实现测试规范的函数

通过实际的与期望的输出值做比较，得到结果，是通过规范还是失败规范调用期望expect()以及匹配器toEqual（toBe）比较。toEqual是深度比较，适合数组，而toBe是简单数值，boolean数据类型的比较。toBeDefined()：非undefined值；toBe(false):返回false；not.toBe(true)

测试规范名称是很有必要的，因为这个在测试不通过时的日志信息很重要。

安装jest

```
npm install --save-dev jest-cli
npm test
```

修改package.json脚本

```
  "scripts": {
    "test": "jest"
  },
```

jest.autoMockOff()设置不要去做自动模拟数据，自动模拟会真实调用require依赖的结果作为模拟的数据，但是这个的预期达不到效果，所以可以调用jest.dontMock()方法，但是这样的话每次依赖写都麻烦，所以可以直接jest.autoMockOff。

测试React组件：在组件的目录下创建**test**测试目录，并安装相应的测试库

```
npm install --save-dev react-addons-test-utils
```

jest.genMockFunction()返回一个模拟函数，TestUtils.findRenderedDOMComponentWithTag（）方法从已经渲染的DOM中查找对应的元素。

TestUtils.Simulate.click（）模拟一个点击事件发送给DOM节点。最后expected(handleClick).toBeCalled(),是否被调用。所以的Jest模拟函数都有一个.mock属性，这个属性保存了模拟函数被调用时的数据，handleClick.mock.calls是自己每次被调用的数据，是一个数组。handleClick.mock.calls.length来判断调用的次数。

为了让Jest支持JSX的语法，还要配置babel-jest模块。

```
  "jest": {
    "scriptPreprocessor": "<rootDir>/node_modules/babel-jest",
    "testFileExtensions": [
      "es6",
      "js"
    ],
```

安装babel-jest模块

```
npm install --save-dev babel-jest
```

九、使用Flux来完善React框架；

通过构建复杂的React组件，完整的数据流向是什么？数据入口在哪里？数据从Stream组件开始进入应用，通过层层传递到达Tweet组件，然后又层层传递到达父组件Application。由父组件负责管理和保存。父组件同时将数据传递给其他子组件。但是在其中很多数据在中间是使用不到的，但还要维持他的状态，这就造成了维护上的极大难度。比如onAdddTweetToCollection（）函数只在Tweet组件中使用，但是被传递了多次

Application > Stream > StreamTweet > Tweet

向应用组件较少时，这种方案还勉强可以维护，但是当应用组件很多时，这就晕了，同时性能上也不允许。为了阻止这样的噩梦发生，我们可以做两件事

- 改变数据进入应用的方式；
- 改变组件获取和修改数据的方式；

在Flux中，数据也是单向流动**Action > Dispatcher > Store**，把应用的关注点分为4个部分：

- Action(动作):每当应用状态改变，都会创建一个Action;在Flux中动作不能脱离Dispathcer单独使用，所有动作都通过分发器分发到存储中。
- Dispatcher(分发器): 管理注册过的回调函数，注册的动作都会通过分发器分发到存储中；
- Store(存储):数据最终流向存储，存储负责管理应用中的数据，提供了读取数据的方法，但没有修改数据的方法，修改数据的方法，通过分发器分发动作来实现。
- View(视图)

安装flux `npm install --save flux`，`npm install --save object-assign`

1.创建分发器

```
var Dispatcher = require('flux').Dispatcher;
module.exports = new Dispatcher();
```

2.创建动作生成器

创建action对象，指定属性类型，以及状态数据，并通过分发器分发action

3.创建存储

首先引入依赖模块，定义私有方法和数据，然后定义公共的Store对象；创建action的回调处理函数，并想分发器注册该store,将dispatchToken附加为store对象，并导出；

一般讲更新的数据setData方法定义为私有的方法，这样数据就不能被外部访问到，只能通过store来获取数据。创建action并通过分发去向注册过的存储器分发出去，当存储器接收到这个action 后，在决定是否更新数据；在Flux中，每一个存储都会接收所有的动作，但并不是每一个存储都需要全部的action，只会选择性的处理action；

- 存储向分发器注册自己
- 动作生成器创建action并通过分发器分发action；
- 存储处理和自己相关的action，并根据这些action更新数据
- 存储向所有的订阅者通知数据发生变化；

十、使用Flux提升应用的可维护性；

借助Flux来实现解耦

React组件获取数据的两种常见方式

- 调用jQuery.ajax(),或者像snapkiteStreamClient等第三方外部库
- 通过props属性接受父组件传递的数据；

为了方便应用中的其他部分从存储中获取数据，Store会导出一些getter方法。为了保持Flux架构的程序数据流的单向流动，一般在Store中也会封装一些内部的setter方法，供自己使用，不让外部直接从存储中更改数据。

通过Flux我们比较前面的组件，可以看出大致的思路：在Flux中可以随意扩展React组件，动作生成器Action,Store等，因为数据流是单向流动，所以增加新的组件相当于横向扩展，不会增加传递数据的深度；

- 通过getter（）方法获取Store中的最新数据；
- 监听Store中的数据变更事件；
- 当Store中的数据发生变化时，通过getter()方法获取最新的数据，并更新到组件的state中；
- 在组件销毁时，停止对Store的监听；