React和Redux的搭配

## Redux和Flux：

### Flux的含义：

Flux是一种架构（思想），Redux是Flux的具体实现框架。在React中，View的变化主要来自于State的变化，而State的变化主要来自于服务器数据变化和用户操作。当组件变多时各个组件之间的状态影响变得复杂（涉及到多个层次的组件，一个组件的变化可能引起其他多个组件的变化，如果依靠一层一层的组件传递props回调函数，将会使得工作量变得巨大）。Flux应运而生，Flux是一种用于集中统一管理state变化的架构，Flux维护着一个或者多个Store，类似于MVC中的Model，当有事件触发组件state变化时，Flux对Store进行更新，并通常由应用的根组件获取最新的state进行层层渲染。

### Flux架构示意

 在上述架构示意图中，Dispatcher是整个控制机制的核心，它用于收集各个组件的Action操作，由其集中对各个store进行state的更新（store是state的容器和变更器）。

### Redux是Flux的实现

Redux是Flux思想的具体实现，但又有些不同。Redux只含有一个Store，而Flux中有多个Store（可以认为Redux中将多个Store合并成一个Store）；Redux中没有Dispatcher，取而代之的是Reducer。Reducer是纯函数，表述为（previousStste，action）=>newState，REdux中有多个Reducer，每个Reducer负责维护一部分组件的state，有combineReducers函数将各个Reducer合并为一个根Reducer，对应唯一的Store。

### Flux和Redux的视图更新示意图（区别）





## React和Redux的搭配使用

### 环境配置：

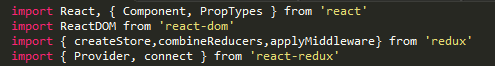
在上节的说明中，可知Redux主要由Store、reducer构成。Redux的使用需要使用npm安装相应库（需要使用npm安装相应库、ES6语法）：



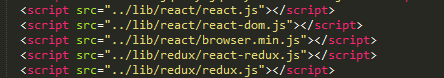
或者直接引入redux的相关库文件：



上述两种方法的全部宝引入如下：



和



### Redux的具体实现：

#### 构造store和reducer

下面将以ES6的写法实现redux：

先构造reducer和store：



如上代码所示：

efaultState是设置初始状态，需要为每一个reducer设置初始的state值；

chooseType’和‘chooseProject’是为每个reducer配对的不同action类型；

combineReducer是Redux中将不同reducer合并为一个Reducer的方法，这个函数中参数以对象的形式传入，projectState:projectReducer表示projectReducer对应处理的是projectState;

applyMiddleware是store的操作日志函数，将会打印相关操作日志。

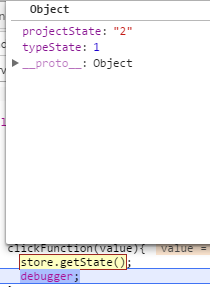
#### 对reducer的触发以及对于store状态的获取



首先为了不在每次向store传递action的时候不重复编写代码，需要构造addTodo函数统一action样式；

Store.diapatch(addTodo(‘chooseProject’))用于向store传递类型为chooseProject的action，该操作将会引发与store对应的rootReducer进行类型判断,依次遍历rootReducer的子reducer，直到匹配到对于的action类型，更新对应的projectState。

Store.getState()函数能够得到目前store中的所有state值，用户可以根据创建rootReducer时所构造的state键值进行筛选获取。如下图：



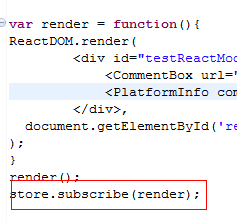
This.projectCLick和this.typeClick将类型选择点击和项目选择点击事件作为props的回调函数传递给子组件DropDownApp；

This.clickFunction作为BusinessComponent组件的内置函数，在点击时获取store的状态值。

### Subscribe函数：store状态更新自动触发view的重新渲染

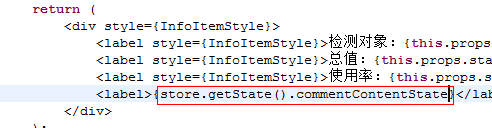
在redux的store模式中，各个组件的数据都托管在store仓库中，当某个action触发了store的状态更新后，由store.subscribe(render)来绑定根组件渲染函数，完成所有依赖于store状态的组件的重新渲染。

#### 入口文件设置方式如下：



#### **依赖于store状态的组件的设置如下**：

直接根据store内的值对组件进行渲染



#### 或者

根据store内的值作为props传递给子组件，由子组件完成渲染



当有action触发store的reducer导致了store的状态更新之后，将由redux自动完成所有相关节点的重新渲染，完成组件间通信。

### Connect(React-redux)函数的使用

React-Redux是Redux的设计者为react量身定制的一个框架，在React-Redux中组件由UI组件和容器组件构成。具有如下的特性：

#### UI组件：

只负责UI的呈现，不带有任务业务逻辑；

没有状态（即不使用this.state变量）；

所有数据都有参数提供（this.props）；

不使用任何Redux提供的API；

#### 容器组件：

负责管理数据和业务逻辑，不负责UI的呈现

带有内部状态

使用REdux的API

因此，在使用React-Redux的体现中，组件按照该规范分离成UI组件和容器组件。

#### 使用方式：

##### Provider：

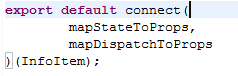
需要在调用处引入Provider，提供store供组件获取自己对应的状态值：





##### Conncet封装：





###### UI组件

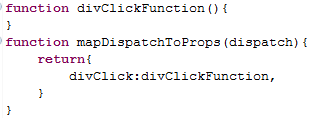


###### 容器组件

MapToProps：



mapToDispatch：



#### 分析

mapToProps方法，把组件和store的状态绑定了，这样一来组件失去了通用性，本来应该由父组件传入的参数改为从store中直接获取。每个组件变成了唯一的个体，对应着store中唯一的state；虽然可以通过父组件传入自己的参数加以区分，但是无法解决组件失去通用性的问题；

该方法不适用于共有组件（复用性高），而适用于模块化的大组件（默认每个模块都应该是唯一的不存在被复用的情况）

conncet解决的问题在于：

将组件的渲染和事件都由mapToProps和maptoDispatch进行管理，转化后props；

能够直接访问到store的状态，不必由父组件一层一层传入。

将UI组件（单纯负责渲染）和容器组件（负责数据和逻辑交互）分离