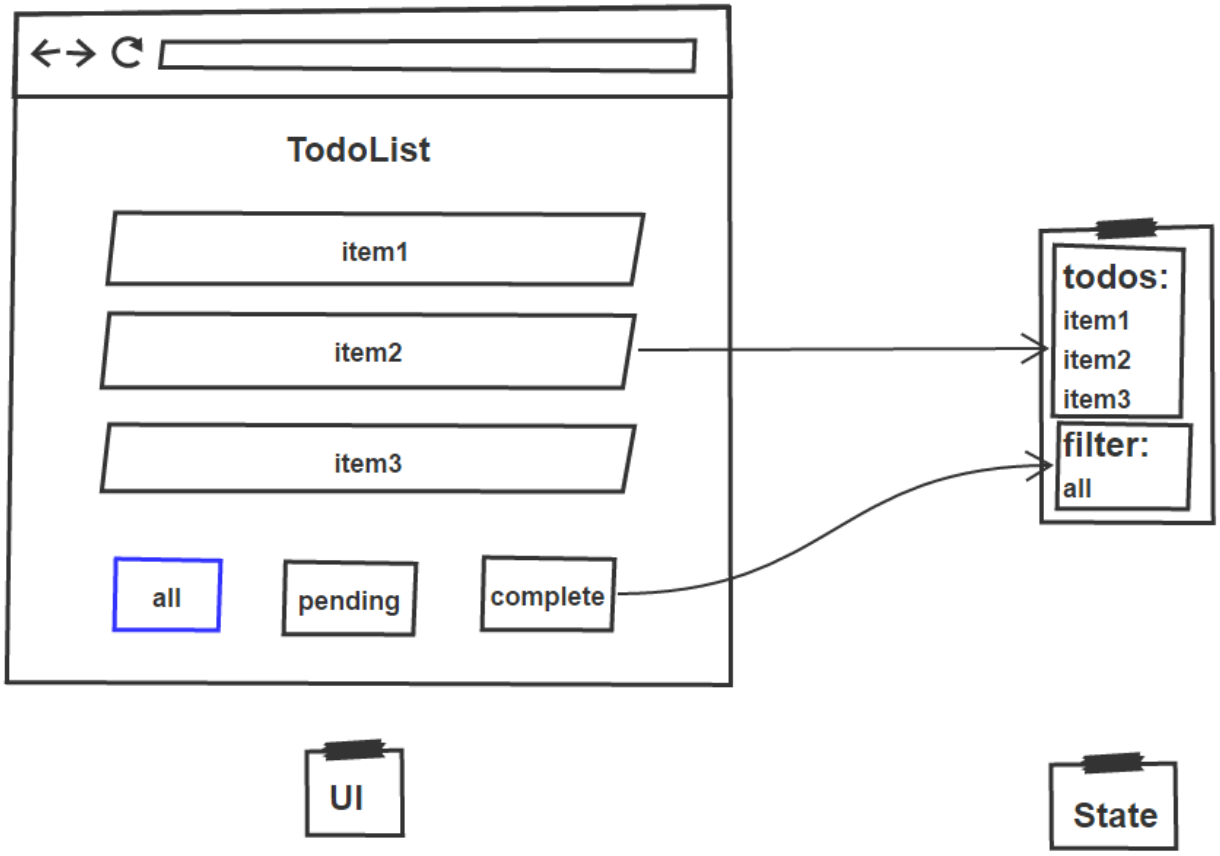
redux,一种页面状态管理的优雅方案

> 前端是工程能力比技术能力更重要的领域，而最近一两年，前端在构建流程、组件化、同构渲染等方面有了深入的发展，其入行门槛也在逐步提高。

前端社区的活跃程度让人惊叹，各种工具层出不穷，比如grunt、gulp、webpack、fis等，即便你没有全部用过，也该了解过它们中的大部分，这些工具极大的解放了前端的生产力，解决了前端的构建流程问题。而React的出现将前端引入新的境界，它优雅的解决了前端UI层组件化的问题，使得组件化也成了前端项目的标配。为了提高页面渲染速度，页面首屏由后端直出，也已经有了很多解决方案。这里我们探讨一下容易被大家忽略的领域：页面状态的管理。

1. 页面状态

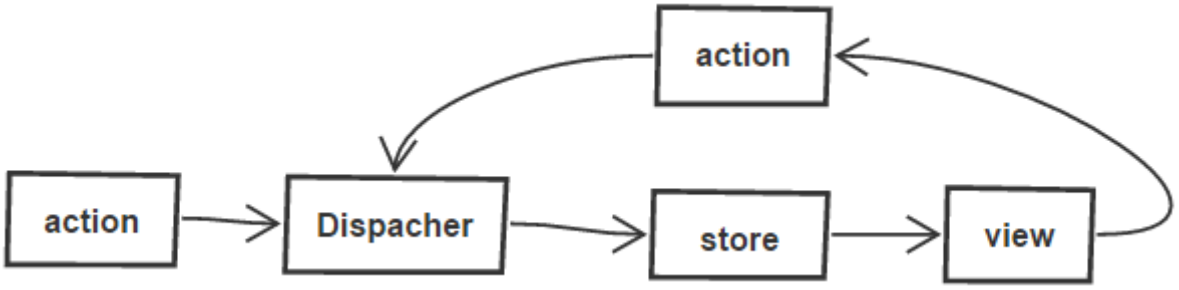
页面上所有UI层的显示都可以用对应的状态描述，比如，比如当前的列表项、当前被选中的标签等。如下图所示：



可以简单的将前端项目抽象为对UI的管理和对状态的管理，UI和状态之间相互作用，处理它们之间的相互关系很复杂，行业内有不同的解决方案，比如以angularJS为代表的双向绑定、以及flux提出的单向数据流。本文我们将抽象的理解facebook提出单项数据流方案flux，以及它的具体实现redux。

2. 单向数据流flux

flux是facebook提出的一种应用程序框架，其基本架构如下入所示，其核心理念是单向数据流，它完善了React对应用状态的管理。



上图描述了页面的启动和运行原理：

```
1.通过dispatcher派发action，并利用store中的action处理逻辑更新状态和view

2.而view也可以触发新的action，从而进入新的步骤1
```

其中的 `action` 是用于描述动作的简单对象，通常通过用户对view的操作产生，包括动作类型和动作所携带的所需参数，比如描述删除列表项的action:

```
{
    type: types.DELETE_ITEM,
    id: id
};
```

而 `dispatcher` 用于对 `action` 进行分发，分发的目标就是注册在 `store` 里的事件处理函数:

```
dispatcher.register(function (action) {
  switch(action.type) {
    case 'DELETE_ITEM':
      sotre.deleteItem(action.id); //更新状态
      store.emitItemDeleted(); //通知视图更新
      break;
    default:
      // no op
  }
})
```

`store` 包含了应用的所有状态和逻辑，它有点像传统的MVC模型中的model层，但又与之有明显的区别， `store` 包括的是一个应用特定功能的全部状态和逻辑，它代表了应用的整个逻辑层；而不是像Model一样包含的是数据库中的一些记录和与之对应的逻辑。

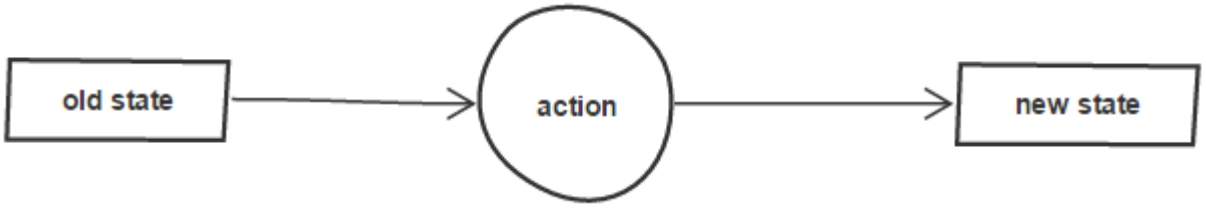
3. 一种对flux的实现，redux

随着前端应用的复杂性指数级的提升，前端页面需要管理的状态也越来越多，flux给出了管理状态的基本数据流，而redux对flux就是对它最好的实现之一，而且其对flux的理念进行了更进一步的扩展。

redux倡导三大原则：

```
1.一个对象存储整个应用的状态
2.状态对象是只读的，只能通过action触发改变
3.通过普通函数处理action的逻辑
```

其基本流程如下：

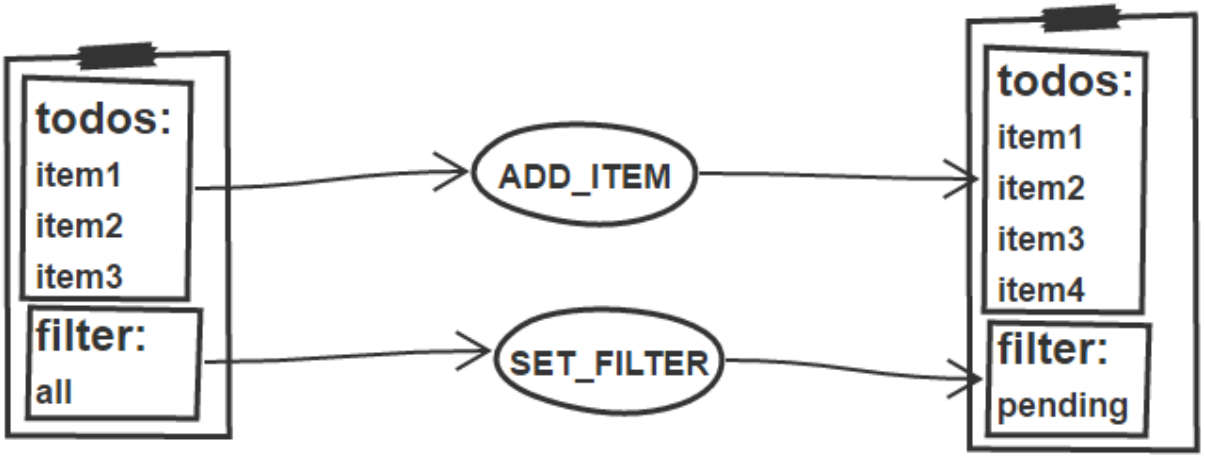


其相对于flux有如下不同之处：

```
1. redux没有dispacher，其通过普通函数处理action逻辑，并改变应用状态
2. redux的状态对象是immutable的，每一个action都会局部地创建新的状态对象
```

需要强调的是， redux不一定要和react搭配，它是一种应用状态管理方案，不涉及UI层 ，你可以任意选择自己的UI层；正因为redux脱离UI层，提供了整个应用状态的管理，使得我们的开发流程有了颠覆性的改变。

我们可以在UI层ready之前，完成应用的逻辑设计和实现。比如我们将应用的逻辑设计如下：



ADD_ITEM的action触发todos列表状态的改变，SET_FILTER的action触发filter状态的改变。因为每一个action都是简单对象，我们可以轻易的模拟。这也就使得我们可以在UI层ready之前，对前端全部逻辑写单独的测试用例。

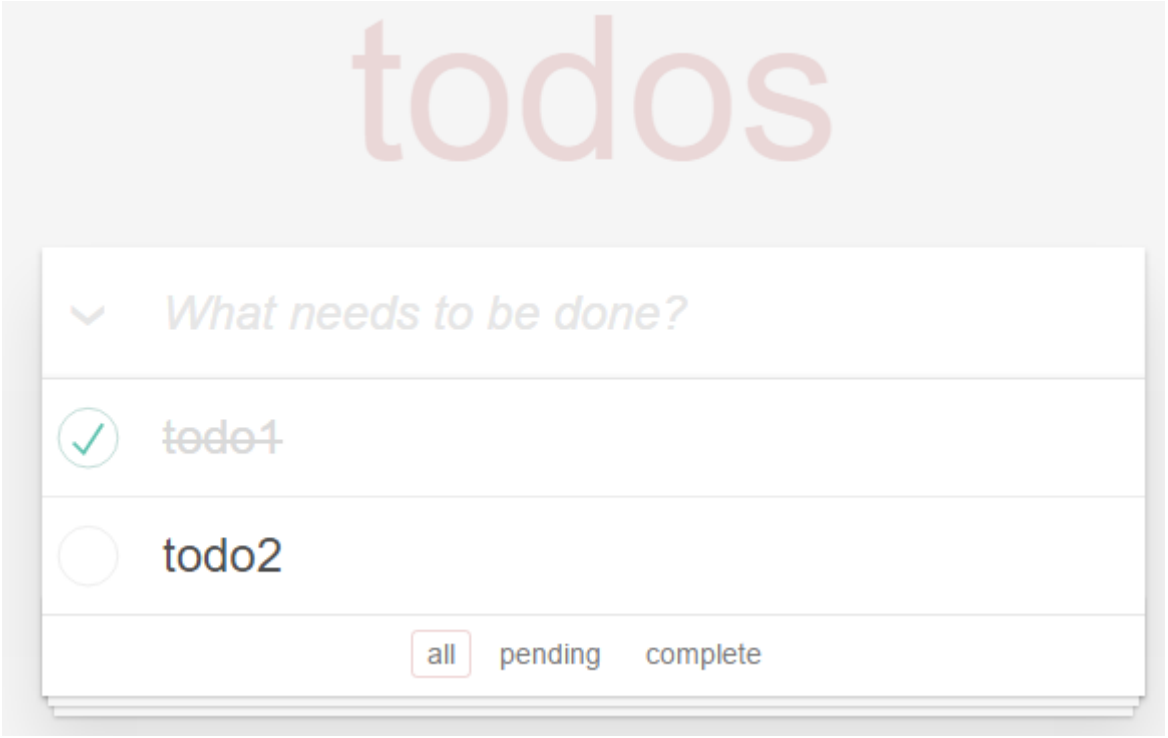
然后，在UI层ready之后，将UI层或网络层的事件映射为redux的action。比如将表单提交事件映射为ADD_ITEM，将标签页切换按钮点击事件映射为SET_FILTER。UI层和逻辑层相互独立，并仅仅通过事件与action的映射来建立联系，这种方案使得复杂的前端项目有了更清晰的架构。

4.redux与react的配合

redux只负责应用的逻辑层，而通过使用 `react-redux` 模块，其可以天衣无缝的和react配合。

4.1 经典案例

我们简单了解一下，如何使用react+redux实现经典的todolist案例。案例的源代码在此(https://github.com/foio/react-redux-isomorphic-todolist)。最终的界面如下：



(1) 逻辑层设计

前端应用本质上是：通过事件触发应用状态的改变。而redux包办了应用状态(state)、事件(action)、和事件处理函数(reducer)，使得我们可以抛开UI层，先设计应用的逻辑层。redux使用单一对象存储整个应用的状态，todolist应用的状态(state)树如下：

```
{
  filter: 'show_all'
  todos: [
    {
      id: 1,
      text: 'todo1',
      marked: true
    }
  ]
}
```

其中filter为列表过滤策略，用于标志底部三个按钮选中的状态，而todos作为代办事项列表。上文提到过state是immutable的，只能通过action触发对state的改变，一个编辑todo条目的action如下：

```
var editTodo = function (id, text) {
    return {
        type: types.EDIT_TODO,
        id: id,
        text: text
    };
}
```

相应的，redux提倡用使用简单函数(又称作reducer)处理action，如下为EDIT_TODO的处理逻辑：

```
module.todo = function(state, action) {
    state = state || [];
    switch (action.type) {

        case types.EDIT_TODO:
            return state.map(function(todo) {
                return todo.id === action.id ?
                    assign({}, todo, { text: action.text }) :
                    todo
            });

        default:
            return state;
    }
}
```

每一个action处理函数都是将action作用在old state上，从而产生new state。state、action、reducer太零散，通过createStore可以将它统一在store中，而store则代表了整个应用的逻辑。

```
var store = createStore(reducer);
```

(2) UI层设计

本案例采用react作为UI层的组件化方案，在这里不再详述。需要注意的是为了配合redux，在写react组件时，我们需要对组件类别进行划分，将其划分为展示型组件和容器型组件 (https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0)。redux只需要和容器型组件通信，而不用管理展示型组件。用一个表格可以很好的说明它们和区别：

	展示型组件	容器型组件
组件设计目标	纯粹的UI(标签、样式)	管理页面逻辑，和展示型组件
对redux是否可见	不可见	可见
数据来源	从props中读取数据	通过redux的state获取数据
改变数据的方式	调用通过props获取的回调函数	派发redux的actions
组件生成方式	手动写组件	通过react redux动态生成

(3) 组合逻辑层与UI层

上文提到，redux只关注react的容器型组件，而且容器型组件可以由react-redux动态生成，以防止state注入容器型组件时的硬编码。比如我们使用如下代码将应用状态注入一个容器型组件TodoApp中：

```
var App = React.createClass({
    render: function() {
        return (
            <Provider store={store}>
                {function() { return <TodoApp />; }}
            </Provider>
        );
    }
});

module.exports = App;
```

上述代码通过Provider将store注入容器型组件TodoApp中。在TodoApp组件中，我们可以通过 `this.props` 来获取store中存储的应用状态了：

```
var TodoApp = React.createClass({
    render: function () {
        var todos = this.props.todos;
        var filter = this.props.filter;
        return (
            <div>
                ......
            </div>
        );
    },
});
function mapStateToProps(state) {
    return state;
}
module.exports = connect(mapStateToProps)(TodoApp);
```

其中connect函数用于动态的创建容器型组件。借助于Provider和容器型组件，我们就将应用的逻辑层和UI层组合在一起了。

4.2 高级特性

了解了react和redux结合的基本思路以后，让我们一起看一看redux的高级特性。

(1) 状态树分治

redux提倡用一个对象存储整个应用的状态，而复杂应用的状态对象是很大的，这样会不会有性能问题？各个容器型组件都对整个应用状态对象进行操作，会不会引起混乱？对此redux有充分的考虑。首选在逻辑层设计时，我们就应该充分的考虑到状态树的分治，比如在设计action的处理函数(reducer)时，针对状态树的不同部分，将其对应的actions处理函数存储在不同的文件中，redux通过combineReducers对此提供了支持。比如

```
var todos = require('../reducers/todos');
var filter = require('../reducers/filter');
combineReducers({filter: filter, todos: todos});
```

其次，在UI层我们也可以很方便的只将部分状态树注入某个容器型组件,redux在使用connect生成容器型组件时，接收一个函数(mapStateToProps)作为参数，该函数可以只返回整个状态树的部分状态，因此，connect生成的容器型组件也就只能感知到部分状态树。这种方式，避免了应用状态树过大的混乱，通过分治降低了复杂度。如下代码，创建了一个只关注整个状态树中state.todos的容器型组件:

```
var TodoApp = React.createClass({
    render: function () {
        var todos = this.props.todos;
        return (
            <div>
                ......
            </div>
        );
    },
});

function mapStateToProps(state) {
    return state.todos;
}
module.exports = connect(mapStateToProps)(TodoApp);
```

(2) 异步action

一般来说，异步action并不能算是高级特性，因为它太常见了。比如发送一个网络请求，这是再寻常不过的需求了。只是用redux触发异步action并不是那么直接。我们需要首先了解redux的中间件概念，它可以用于在action被触发和action到达处理函数reducer之前，对action进行处理。

可以在创建store时，通过applyMiddleware函数提供redux的中间件：

```
createStore( todosApp,applyMiddleware(someMiddleWare))
```

一个典型的redux中间件是redux-logger (https://github.com/evgenyrodionov/redux-logger)，它在控制台中记录每一次action作用前后的应用状态变化，非常适合在开发阶段进行调试。

```
action @ 20:14:57.028 SET_VISIBILITY_FILTER
    prev state ▶Object {filter: "show_all", todos: Array[5]}
    action Object {type: "SET_VISIBILITY_FILTER", filter: "show_unmarked"}
    next state ▶Object {filter: "show_unmarked", todos: Array[5]}
```

redux官方提供了 `thunkMiddleware` 的中间件，用于处理异步action，它使得redux可以派发一个函数而不是一个普通action对象，在该函数中我们可以进行异步网络请求：

```
var fetchTodos = function () {
    return function (dispatch) {
        return fetch('/todos');
    }
}
```

我们可以使用dispach函数像派发普通action一样，派发异步函数，异步函数的返回值还可以是Promise，其返回值会透传过dispch函数。

```
dispach(fetchTodos)
  .then(function(json){
      //handle response
  })
  .catch(function(error){
      //handle error
  });
```

通过网络加载数据，并在数据到达时更新应用状态是一种比较常见的应用场景，对于这种场景，一种最优雅的方案：

```
1. 派发异步函数，用于进行网络请求
2. 在网络请求完成时，派发同步action用于更新应用状态
```

可以用如下代码表示：

```
var fetchTodos = function () {
    return function (dispatch) {
        return fetch('/todos')
            .then(function (json) {
                //派发同步aciton，用于更新应用状态，初始化todo列表
                dispatch(initTodos(json.data || []));
            }).catch(function () {
                //派发同步action，用于更新应用状态，设置加载失败标志
                dispatch(failLoadedTodos());
            });
    }
}
```

(3) 同构渲染

前后端同构，应用首屏由后端直出是近年来比较流行的性能优化方案，redux对此也有完善的支持。基本流程是：

```
1. 服务端初始化state
2. 将服务端state传递到应用的页面端
3. 页面端用服务端传递的状态初始化应用state
```

在遵从这个基本流程的情况下，服务端和页面端的使用方法开发方法基本一致，如下是服务端代码：

```
 store.dispatch(todoActions.loadInitTodos()).then(function () {
        var contentHtml = React.renderToString(
            <Provider store={store}>
                {function () {
                    return <TodoApp />;
                }}
            </Provider>
        );
        var initialState = JSON.stringify(store.getState());
        res.render('index.ejs', {contentHtml: contentHtml, initialState: initial
    }).catch(function(error){
        res.json({errMsg: 'internal error'})
    });
```

上述服务端代码通过派发初始化异步函数更新应用状态，该异步函数返回一个Promise，Promise对象会透传过dispach函数。在Promise处理完成后，我们得到应用的最新状态。最后我们将由React输出的HTML字符串contentHtml和初始化应用状态initialState，传递到模板文件index.ejs中，模板文件如下：

```
<html>
  <head>
    <title>Redux TodoMVC</title>
  </head>
  <body>
    <div class="todoapp" id="root"><%-contentHtml%></div>
  </body>
  <script>
    window.__INITIAL_STATE__ =  <%-initialState%>;
  </script>
</html>
```

通过浏览器的window对象，我们将服务端的初始状态传递到了页面端。

```
var todosApp = combineReducers({filter: filter, todos: todos});
var store = createStore(  todosApp,
                          window.__INITIAL_STATE__,
                          applyMiddleware(thunkMiddleware, reduxLogger())
                        );
```

作者信息：

关于作者: 曾先后参与百度地图用户中心后端系统的设计与研发、百度国际化手机助手前端架构的设计，目前在腾讯QQ手机浏览器团队从事前端开发工作，关注web全栈架构领域。博客地址：foio.github.io
文章署名：张鹏
联系地址：深圳市南山区深南大道9988号大族科技中心
本人银行账号：6225 8801 6286 2792
开户行：招商银行(北京分行上地支行)
身份证号码: 411526198711011334
Email: syszhpe@gmail.com (mailto:syszhpe@gmail.com)
电话: 18510332879

本文案例源代码：
https://github.com/foio/react-redux-isomorphic-todolist (https://github.com/foio/react-redux-isomorphic-todolist)

参考文献：
http://redux.js.org/ (http://redux.js.org/)
https://facebook.github.io/flux/docs/overview.html
(https://facebook.github.io/flux/docs/overview.html)
https://sketchboard.me/ (https://sketchboard.me/)
https://github.com/gaearon/redux-devtools (https://github.com/gaearon/redux-devtools)
https://github.com/foio/react-redux-isomorphic-todolist (https://github.com/foio/react-redux-isomorphic-todolist)