现状

1. 项目规模越来越大，以P2P H5项目（backbone）为例（66个视图、21个模型、12个集合、80+个模板文件、3W+行代码）。不算是很大嘛！！
2. 我很懒，非常非常的懒，想让自己能够更懒。愚笨之人总想让自己更懒….
3. 为了不重复造轮子，我想尝试把公共部分抽离出来，编程高度解耦的公共组件。（代码贴来贴去干的毛线…又不好维护）
4. 目前所有视图中可复用的组件屈指可数（视图嵌套的方式）。仅有的几个算是对backbone项目中组件化的一种尝试。
5. backbone中组件化较为艰难，一个个视图嵌套看似完美解耦，其实带来的只是维护的深渊。
6. 视图与嵌套的子实体之间的除了基于模型监听的方式，其他数据的交换异常艰难。
7. 无法直观了解视图嵌套情况。
8. 对于前段来说MVVM或者MVC都太臃肿。
9. 每次一界面变化都是基于模型的监听，我很懒，因此模型的change监听都是调render方法重新渲染视图。恩，当然好一点应该是change某些字段，做最小化dom变更。但是我很懒。
10. 努力做到单向数据流。但是仍不可避免的直接操作界面。其实我想扔掉jquery(zepto)的。

目标

1. 清晰可维护。
2. 高度复用的组件化。
3. 每次界面变化都是整体刷新。
4. 优秀的渲染效率。
5. 单向数据流。

编写可预测，符合习惯的代码

> “
>
> *软件工程向来不提倡用高深莫测的技巧去编程，相反，如何写出可理解可维护的代码才是质量和效率的关键。*

一些命令式编程代码的不可读，导致“破窗”，当代码规模越来越大后，“破窗”原来越多，最终导致代码的不可维护。我们往往需要添加大量的注释，来使我们不至于忘记某一个判断甚至于某一个变量的作用。
React的项目经理Tom Occhino进一步阐述React诞生的初衷，在演讲中提到，React最大的价值究竟是什么？是高性能虚拟DOM、服务器端Render、封装过的事件机制、还是完善的错误提示信息？尽管每一点都足以重要。但他指出，其实React最有价值的是声明式的，直观的编程方式。

使用JSX直观的定义用户界面

> “
>
> *界面层和业务逻辑分离，这句话，相信只要做过一段时间码农的，都至少会挺过。我们花了很多年实践总结出业务逻辑与界面分离的“最佳实践“。*

想想backbone是如何定义用户界面的?

我们用了模板方式将Model，生成HTML并添加到页面DOM中，以达到显示与逻辑分离。但是在前端这样真的有意义么?

视图的逻辑代码依然依赖于模板中定义的内容和DOM结构。尽管你可以将他们在文件层面上分离，但实际上他们任然是强耦合，实际上两者是一体的。

二这种分离以后造成的接口就是2者之间的协作不能不引入很多机制，比如多模型的监听，对DOM的监听，中间可能掺杂了很多直接的js dom操作来直接对界面进行操作。（当然这对于MVVM其实也是一种反模式）。

而且这样的方式并不利于，纵向的组件化拆分解耦。

于是，可以认为在没有其他的解决方案来为它们解耦之前，这样的分离其实没有

实际意义。至少对于Web前端开发来说，没有实际意义。

我们看看JSX干了些啥?

```
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        Hello, world! I am a CommentBox.
      </div>
    );
  }
});
ReactDOM.render(
  <CommentBox />,
  document.getElementById('content')
);
```

JSX是React的核心组成部分，它使用XML标记的方式去直接声明界面，界面组件之间可以互相嵌套。严格意义上来说它不能算是一种模板的实现方式。
我们可以直接用js来实现逻辑上的DOM输出，比如下面的列表渲染:

```
render: function () {
  var lis = this.todoList.todos.map(function (todo) {
    return  (
      <li>
        <input type="checkbox" checked={todo.done}>
        <span className="done-{todo.done}">{todo.text}</span>
      </li>);
  });
  return (
    <ul class="unstyled">
      {lis}
    </ul>
  );
}
```

组件化

react十分适合封装公共组件，及其嵌套调用。组件的DOM结构、逻辑、甚至样式皆可封装在一起。

```
var Avatar = React.createClass({
  render: function() {
    return (
      <div>
        <PagePic pagename={this.props.pagename} />
        <PageLink pagename={this.props.pagename} />
      </div>
    );
  }
});

var PagePic = React.createClass({
  render: function() {
    return (
      <img src={'https://graph.facebook.com/' + this.props.pagename
 + '/picture'} />
    );
  }
});

var PageLink = React.createClass({
  render: function() {
    return (
      <a href={'https://www.facebook.com/' + this.props.pagename}>
        {this.props.pagename}
      </a>
    );
  }
});

ReactDOM.render(
  <Avatar pagename="Engineering" />,
  document.getElementById('example')
);
```

React将用户界面看做简单的状态机器。当组件处于某种状态时，那么就输出这个状态对应的界面。这样是的方式，很容易确保界面的一致性。

```
var LikeButton = React.createClass({
  getInitialState: function() {
    return {liked: false};
  },
  handleClick: function(event) {
    this.setState({liked: !this.state.liked});
  },
  render: function() {
    var text = this.state.liked ? 'like' : 'haven\'t liked';
    return (
      <p onClick={this.handleClick}>
```

```
        You {text} this. Click to toggle.
      </p>
    );
  }
});
```

整体刷新

之前backbone遇到的问题

让我们想想之前用backbone的时候，当模型数据改变的时候我们是怎么做的?

首先我们得有个模型，比如说用户模型，他有多个attributes

id
loginName 登录名
nick 昵称 string
honor 勋章 array
amount 账户余额 number

需要做一个账户详情页，有如下几个User story:

当用户修改昵称后界面自动显示出新的昵称
当用户获得勋章后，点亮对应的勋章图标
当用户的账户余额大于4000，点亮vip图标
然后对应这些story，我们在视图里添加监听

```
initialize: function(){
    this.listenTo(this.model, 'change: nick', this.renderNick);
    this.listenTo(this.model, 'change: honor', this.renderHonor);
    this.listenTo(this.model, 'change: amount', this.renderVip);

    return this.render();
}
```

好烦躁啊好烦躁，3个render局部的方法还是另外写。即麻烦又不好维护。
我们姑且称之为层叠式更新，这只是单视图，后面会有嵌套的子视图。
界面越来越复杂，story越来越多，所以啊，懒人就该有懒人的做法。

嗯，干脆改下。

```
initialize: function(){
    this.listenTo(this.model, 'change', this.render);

    return this.render();
}
```

好了，这样干净多了，一劳永逸哈哈，但这有出现了另一个问题。如，用户获得了一个勋章，恩，其实应该只需要为对应的勋章元素添加一个class即可，但我们为此确重新render的视图…
不管从渲染性能、实际效果还是用户体验来说都不是一种理想的处理方式。

React如何处理这样的问题呢?

React是整体刷新来解决层叠式更新复杂度的问题。即只要state发生变化，都相当一次刷新，React框架自身来解决局部更新的问题。
React为此引入的虚拟DOM的解决方案。
简而言之就是，UI界面是一棵DOM树，对应的我们创建一个全局唯一的数据模型，每次数据模型有任何变化，都将整个数据模型应用到UI DOM树上，由React来负责去更新需要更新的界面部分。事实证明，这种方式不但简化了开发逻辑并且极大的提高了性能。

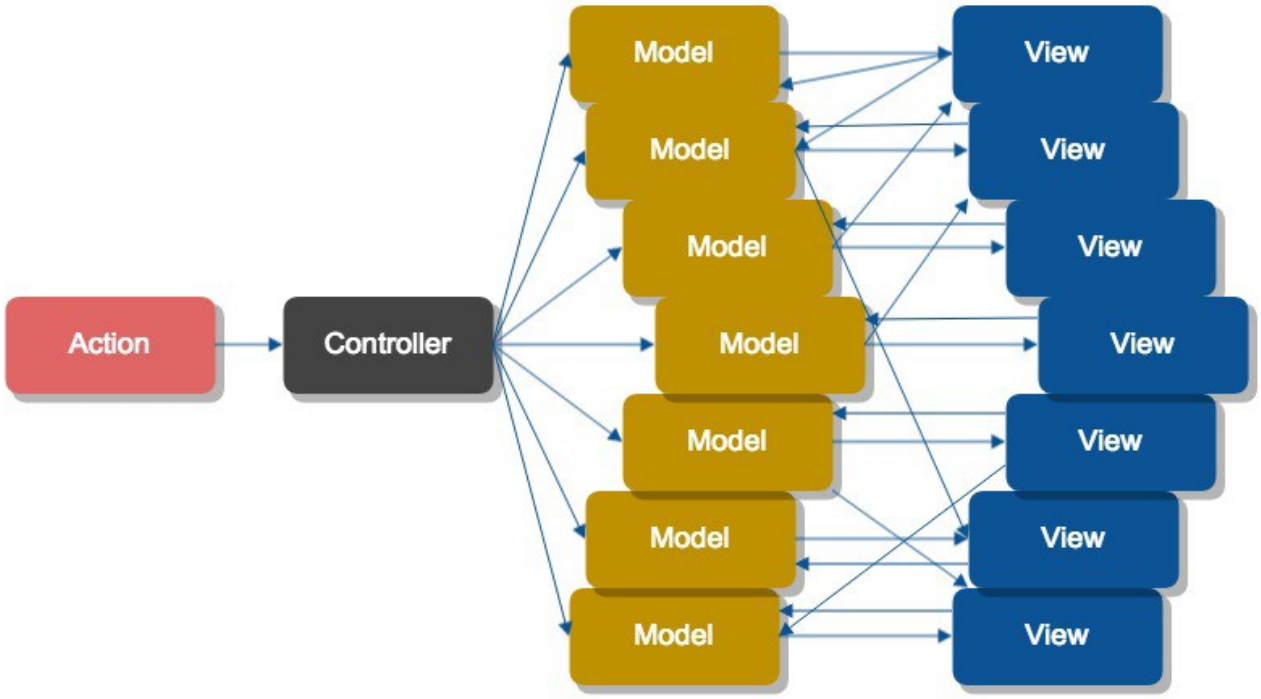
它的性能依赖于React diff算法。详细的介绍参考http://www.w3ctech.com/topic/1598

单向数据流

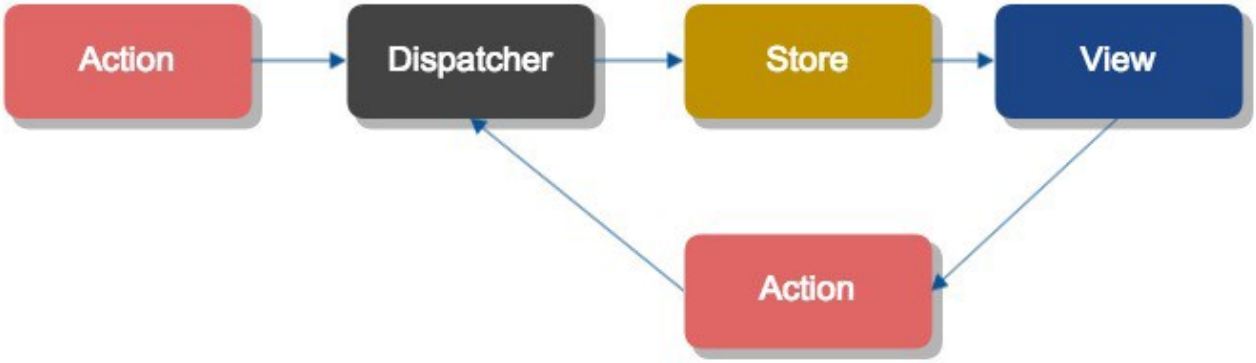
Flux其实并非是一个框架，它是一种理念，一种思想，可以称之为”单向数据流”。

Facebook推出的Flux只是这种思想的一种实现。

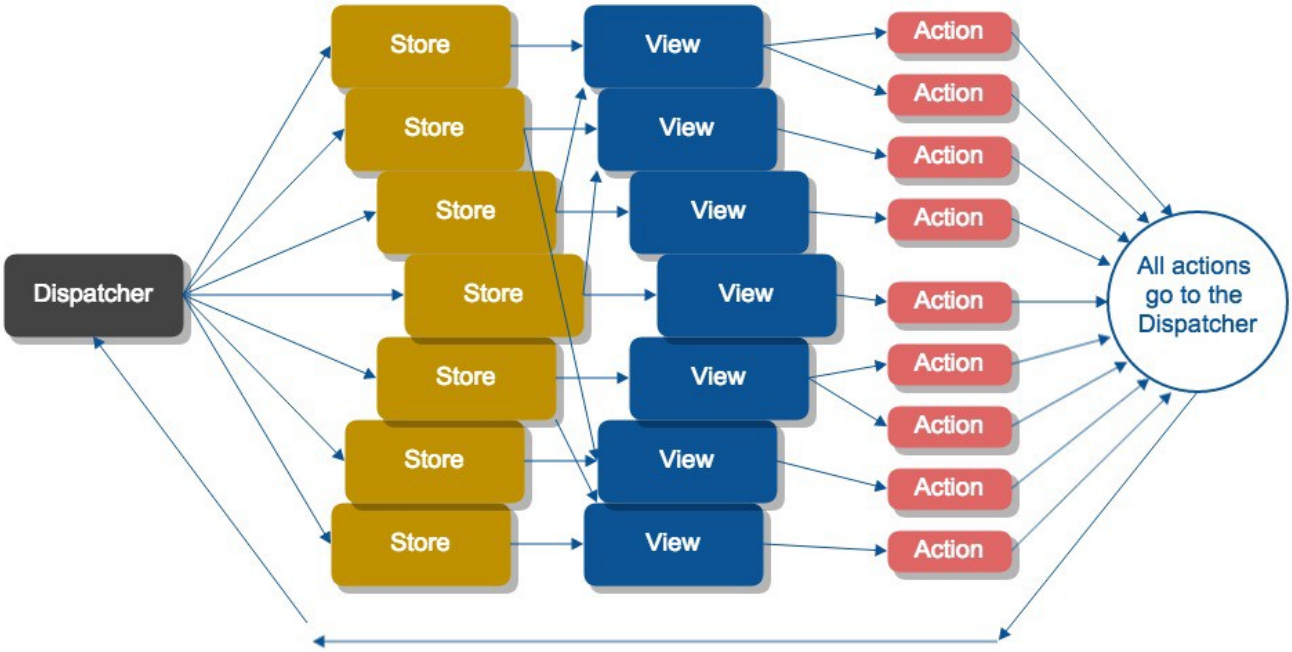
先看看MV*:



Flux:



…看上去是这样的简单？
其实应该是这样的：



看上去完全和MV*是一个量级的么…那有毛意义?

好吧！让我们分析下这样做的优势:

1. Flux中所有的箭头都指向同一个方向。
2. Flux种的派发器确保了系统中一次只会有一个action流。派发器也能让开发者指明回调函数执行的顺序，其中会使用waitFor方法来告诉回调函数依次执行。

这些特点无不为了提高代码的可预测性这个最终目标。是开发者在操作数据源交互时变得简单。