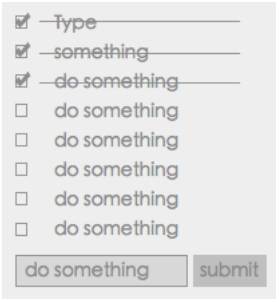
这篇文章将从 AngularJS ReactJS Polymer 这几个流行的框架入手，分析前端框架在这几年发展中的关键技术点，作为2015前端技术选型的参考。摘要：

1. 初体验

拿TODO来作为引子好了.



Angular 的实现

```
angular.module('todomvc')
  .controller('TodoCtrl', function TodoCtrl($scope, $routeParams, $filter, store) {
    'use strict';

    var todos = $scope.todos = store.todos;

    $scope.newTodo = '';
    $scope.editedTodo = null;

    $scope.addTodo = function () {
      var newTodo = {
        title: $scope.newTodo.trim(),
        completed: false
      };


      store.insert(newTodo)
        .then(function success() {
          $scope.newTodo = '';
        })
    };
    /*...省略...*/
  });
```

```
<section >
    <header >
        <h1>todos</h1>
        <form  ng-submit="addTodo()">
            <input ng-model="newTodo">
        </form>
    </header>
    <section  ng-show="todos.length" >
        <ul >
            <li ng-repeat="todo in todos track by $index">
                <div >
                    <input  type="checkbox" ng-model="todo.completed" ng-change="toggleCompleted(todo)">
                    <label ng-dblclick="editTodo(todo)">{{todo.title}}</label>
                    <button  ng-click="removeTodo(todo)"></button>
                </div>
                <form ng-submit="saveEdits(todo, 'submit')">
                    <input  ng-trim="false" ng-model="todo.title" ng-blur="saveEdits(todo, 'blur')">
                </form>
            </li>
        </ul>
    </section>
</section>
```

React的实现(非flux架构)

```
var app = app || {};

(function () {
    'use strict';

    var TodoFooter = app.TodoFooter;
    var TodoItem = app.TodoItem;

    var ENTER_KEY = 13;

    var TodoApp = React.createClass({
        getInitialState: function () {
            return {
                nowShowing: app.ALL_TODOS
            };
        },
        save: function (todoToSave, text) {
            this.props.model.save(todoToSave, text);
            this.setState({editing: null});
        },
        render: function () {
            var footer;
            var main;
            var todos = this.props.model.todos;

            var todoItems = todos.map(function (todo) {
                return (
                    <TodoItem
                        key={todo.id}
                        todo={todo}
                    />
                );
            }, this);


            if (todos.length) {
                main = (
                    <section id="main">
                        <input
                            id="toggle-all"
                            type="checkbox"
                            onChange={this.toggleAll}
                            checked={activeTodoCount === 0} />
                        <ul id="todo-list">
                            {todoItems}
                        </ul>
                    </section>
                );
            }

            return (
                <div>
                    <header id="header">
                        <h1>todos</h1>
                        <input
                            ref="newField"
                            id="new-todo"
                            placeholder="What needs to be done?"
                            onKeyDown={this.handleNewTodoKeyDown}
                            autoFocus={true} />
                    </header>
                    {main}
                    {footer}
                </div>
            );
        }
    });

    var model = new app.TodoModel('react-todos');

    function render() {
        React.render(
            <TodoApp model={model}/>,
            document.getElementById('todoapp')
        );
    }

    model.subscribe(render);
    render();
})();
```

Polymer的实现

```
<polymer-element name="td-todos" attributes="route modelId">
    <template>
        <flatiron-director route="{{route}}"></flatiron-director>
        <section >
            <header >
                <input is="td-input"  on-td-input-commit="{{addTodoAction}}">
            </header>
            <section>
                <ul  on-td-item-changed="{{itemChangedAction}}">
                    <template repeat="{{model.filtered}}">
                        <li is="td-item" item="{{}}"></li>
                    </template>
                </ul>
            </section>
        </section>
    </template>

    <script>
        (function() {
            var ENTER_KEY = 13;
            var ESC_KEY = 27;
            Polymer('td-todos', {
                addTodoAction: function() {
                    this.model.newItem(this.$['new-todo'].value);
                    // when polyfilling Object.observe, make sure we update immediately
                    Platform.flush();
                    this.$['new-todo'].value = '';
                },
                /*...省略...*/
            });
        })();
    </script>

</polymer-element>
```

```
<polymer-element name="td-model" attributes="filter items storageId">
    <script>
        Polymer('td-model', {

            newItem: function(title) {
                title = String(title).trim();
                if (title) {
                    var item = {
                        title: title,
                        completed: false
                    };
                    this.items.push(item);
                    this.itemsChanged();
                }
            },
            /*...省略...*/

        });
    </script>
</polymer-element>
```

三者共同对比

在Angular中有controller和component的概念是分离的，而react和polymer中只有component的概念。

实际上三者在最简单的使用场景下差异并不大，Angular和polymer模板和代码分离的方式更贴近于传统的前端做法，而React写法更像后端渲染。关于学习和使用成本的谁高谁低得问题没有什么好争论的，在MVVM已经流行了这么久的情况下，三者入门门槛都差不多，但要用好都需要深入其中的运行机制才行。

2. 技术特点

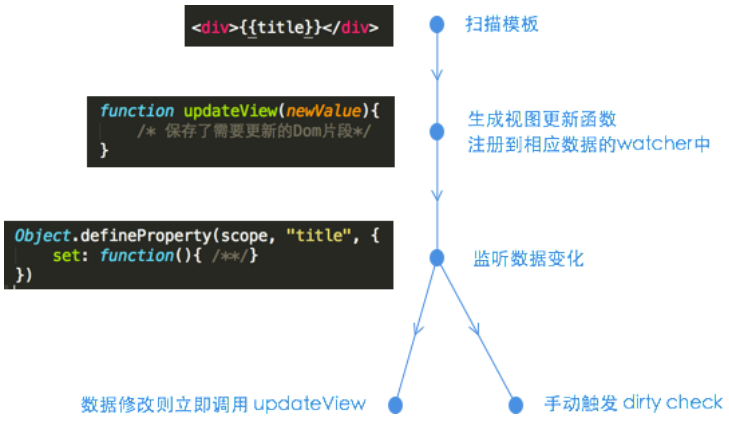
实际上所谓的MVVM框架的关键技术就一个：数据与视图的绑定。在Angular/polymer/knockout/vue/avalon 中，这项技术的实现又可以拆分成两个关键点：模板分析和数据监测。

模板分析的主要目的是对 {{title}} 这样的标记进行收集。收集完成之后生成一个视图更新函数，在函数内部保存着这个标记所在的Dom片段和相关的数据名称，函数被调用时会去重新取数据名称对应的数据(或者由外部将相应的数据作为参数传入)，然后更新dom片段。这样就实现了视图的更新。一般框架会在启动时就将模板分析完，生成相应的视图更新函数。当数据更新的时候，就调用这些更新函数来更新视图，那么问题来了，如何检测数据的改动?

knockout/angular/avalon代表了三种方案:

> 使用自定义的数据对象及其指定的get和set函数。例如你只能使用 user.set("name","john")来给user对象的name属性赋值，因为这样它才能在set函数中知道修改了什么属性，并且只调用相应的视图更新函数。这种方式不太爽的地方在于改变了原有的JS对象使用的方式。
> 使用 Object.defineProperty 的get和set函数来检测对象属性的改动，本质上和上种没有什么区别。但是它有一个缺陷，就是无法检测新增的或删除的属性。有的框架是通过Object.observe来补充这种方案的，不过Object.observe 目前也只有chrome支持。这种方法改良了上面的开发体验，你可以像使用原生JS对象一样来操作你的数据。但是在实现上较为复杂。
> dirty check。这是angular正在使用的机制，它并不能像前两种一样一旦数据发生变化立即触发更新回调。而是必须在调用了angular提供的一些方法，或者触发了页面上使用了ng-click等的元素上的事件后才会触发。这些触发时机是angular内部就已经实现了的，所以你几乎感觉不到。这种方法被称为"dirty"的原因是，它保存了所有属性上一次的值，检测是通过遍历对象的所有属性，对比它和上一次值是否一样来实现的。如果是深层对象的话，它会层层遍历。这种检测方式结合了上面两种的优势,但是对性能造成了负担。

至此，两个关键技术点都已讲清楚，用一张图来回顾一下



而在React中则相对简单，React用的是类似于重绘的机制，当触发了 setState 之后，就完全重新渲染(并非立即触发，中间有类似于缓存的性能提升机制)。这看起来比起前面的方案简单粗暴，但是却因为virtual dom的实现化腐朽为神奇了。virtual dom指的是React内部用来模拟真实dom的一种数据对象。当重新渲染时，实际上是先生成这样virtual dom，然后将其和上一次的virtual dom进行对比，找出差异，最后由react在真实的dom上更新有差异的部分就够了。因为virtual dom始终在内存中，真实的dom操作非常少，而前面的几种框架在更新视图时常常会有大量的dom操作，因此react在性能上大大领先前一种类型的框架。同时也因为virtual dom仍然是标准的 js对象，所以使得"服务端渲染"也成为可能。
值得注意的是，虽然React本身并不会像前面的框架一样深入的去检测数据的哪一部分发生了变化，但是可以通过官方提供的addon 和immutable.js来进一步提高这一块的性能。

3. 组件化

在组件化的方向上 react 和其他几种框架几乎已经分道扬镳了。从 angular2.0的设计和新出的 aurelia 等框架中可以看到大家都在尝试往 webcomponent 靠近。polymer号称下个版本代码将大幅减少，那无非是因为浏览器将实现标准了。靠近 webcomponent 的好处在于任何一个框架都将不再封闭，以 custom element作为接口层，能实现生态圈的融合。
虽然 react 也有封装成 custom element的方案，但是 react 并没有很好的调用其他框架生成的 custom element 的方案。"像使用原生dom元素一样使用custom element"的组件使用方式意味着尊重原生的dom使用方式，包括dom的事件等等。这和react"不操作真实dom"的基础已经方向相悖了。

react和其他框架的分歧其实目前看来并无优劣之分，因为webcomponent目前除了chrome以外其他浏览器支持仍然不全面。另外考虑到特殊国情的话，大公司的产品仍然要面对IE8。不幸的是目前polymer的polyfill最低也只到IE9。而React能无痛支持IE8。再考虑到移动端的浏览器情况的话，也是使用react的技术阻力远小于webcomponent。

总体来看，webcomponent肯定会是趋势，并且将促进各个框架变得更加开放，更易互相融合。而react也仍将继续依靠自己在实现上的优势继续走下去。也许未来在这中间又将诞生新东西。

暂时抛开react和webcomponent。我们继续深入两个目前讨论得很少但是却很重要的问题(下面讨论的组件问题都以封装成custom element为基础):

> 如何能把组件变得更易重用? 具体一点:
> > 我在用某个组件时需要重新调整一下组件里面元素的顺序怎么办?
> > 我想要去掉组件里面某一个元素怎么办?
> 如何把组件变得更易扩展? 具体一点:
> > 业务方不断要求给组件加功能怎么办?

针对第一个问题，我所在的团队目前提出一个叫做"模板复写"的规则，这个规则又分为"完全重写"和"部分重写"两种规则：

```
<!--原始模板-->
<div>
    <div>{{title}}</div>
    <div>{{content}}</div>
</div>
```

```
<!--完全覆写-->
<story>
    <header class="custom_class">{{title}}</header>
    <section id="custom_id">{{content}}</section>
    <a onClick="like(this)">like!</a>
</story
```

部分重写

```
<!--原始模板-->
<div>
    <div role="title">{{title}}</div>
    <div role="content">{{content}}</div>
</div>


<!--部分覆写-->
<story tpl-partial>
    <header role="title" class="custom_class">{{title}}</header>
</story>
<!-- 将生成
<div>
    <header role="title" class="custom_class">{{title}}</header>
    <div role="content">{{content}}</div>
</div>
-->

<!--部分包含-->
<story tpl-include>
    <header role="title"></header>
</story>
<!-- 将生成
<div>
    <div role="title">{{title}}</div>
</div>
-->


<!--部分省略-->
<story tpl-exclude="title">
</story>
<!-- 将生成
<div>
    <div role="content">{{content}}</div>
</div>
-->
```

这种方案已在angular中实现。并且在组件重用率高的系统中已经验证非常实用。但它也有缺陷，缺陷在于你必须知道当前组件的实现方式和原有模板才能复写。

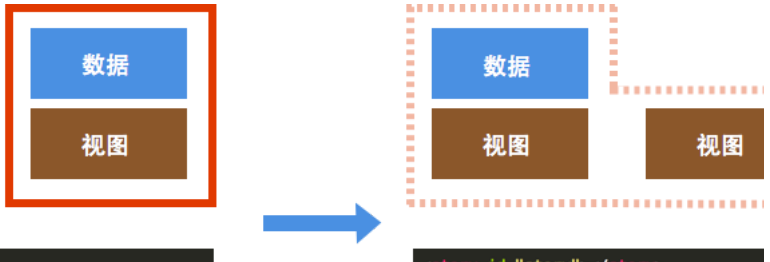
第二个问题，可以用一种称为"共享作用域"的方式来解决。例如上面的例子中story没有显示like数量，现在要显示出来。常规方案有两种：

改组件，在组件中增加这个功能。

给组件增加api用于获取统计数据，同时在统计数据发生变化时抛出事件通知外部。

第一种方案可能碰到的问题是当再次发生变化，例如统计数据不要显示在组件里面了。就得继续改成第二种方案。 第二种方案可能碰到的问题是可能不断有新的需求提出来，最后不得不把每一个内部状态都暴露出来，每一个操作过程都抛出事件。

"作用域共享"共享的方案是：通过在一个特殊标记 "import-to" 将某一段外部html引入到某个组件中去一起参与"模板解析"和"数据绑定",当完成时再放回原来的位置。这样这个外部html就能获取到组件内部任何状态和数据了。这种方案看起来有点像hack，但其实只是换了一种方式来理解组件:组件分成两个部分，一是数据，二是视图。视图理论上应该只受到它的逻辑是否足够内聚的约束，而不应该受到它的子元素是否放在一起的约束。但是目前我们刚好使用了dom作为视图的基础，所以视图受到html结构的约束，这个约束是不合理的。我们来用图对比一下使用"作用域共享"前后的场景：



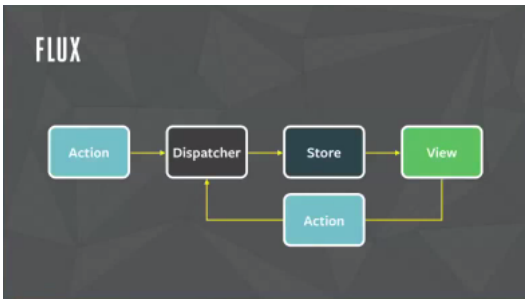

```
<story id="story"></story>

<div id="statics"></div>
<script>
    function render(){
        document.querySelector("#statics").innerHTML =
        document.querySelector("#story").getLikes()
    }

    document.querySelector("#story").addEventListener("change",render)

    render()
</script>
```

```
<story id="story"></story>
<div import-to="story">{{likes}}</div
```

当然，这种方案的缺陷仍然是你必须知道组件的具体实现。但这并不是一个不可克服的缺陷，我们看下aurelia的设计，它将template等等关键部分都设计成了可插拔的形式，这种结构意味着未来有可能实现一种通用的模板语法来实现上述两个功能。这样就不再和底层耦合。

4. 应用架构

应用架构的范围太广，我们这里只讨论那些已经很好地组件化了的应用，或者是没组件化但是有明确层级划分的应用。我们以React 对应的 FLUX 为切入点。



我们再来结合facebook的官方FLUX代码示例来看看每个部分：

Dispatcher

```
var Dispatcher = require('flux').Dispatcher;

module.exports = new Dispatcher();
```

Action

```
var AppDispatcher = require('../dispatcher/AppDispatcher');

var TodoActions = {
  create: function(text) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_CREATE,
      text: text
    });
  },
  /*...省略...*/
}
```

Store

```
var AppDispatcher = require('../dispatcher/AppDispatcher');
var EventEmitter = require('events').EventEmitter;
var TodoConstants = require('../constants/TodoConstants');
var assign = require('object-assign');
var CHANGE_EVENT = 'change';
var _todos = {};


var TodoStore = assign({}, EventEmitter.prototype, {

    create: function(text) {
      var id = (+new Date() + Math.floor(Math.random() * 999999)).toString(36);
      _todos[id] = {
        id: id,
        complete: false,
        text: text
      }
    },
    /*...省略...*/
});


AppDispatcher.register(function(action) {
  var text;

  switch(action.actionType) {
    case TodoConstants.TODO_CREATE:
      text = action.text.trim();
      if (text !== '') {
        TodoStore.create(text);
      }
      TodoStore.emitChange();
      break;
    /*...省略...*/

    default:
  }
});

module.exports = TodoStore;
```

facebook在介绍FLUX的时候的主要观点是"MVC扩展性不够，FLUX可扩展性高"。暂且不去讨论FLUX与MVC的区别，我们先来它是如何扩展的，从上面的代码中可以看到，ACTION不只是一个界面上的点击事件所产生的，ajax请求、甚至一个初始化过程都可以产生动作，"动作"只是一个抽象。动作将传递给dispatcher,有dispatcher在去触发store注册的回调。你可能会想，从这个dispatcher实际上什么也没干，这和我直接定义一个方法，触发事件就直接调用这个方法有什么区别?区别在于，当应用增加功能、进行扩展时，应用可能有多个部分要协同对同一个action进行响应，并且不同的协同部分可能在执行顺序上有严格的先后之分。

举个例子，如果我要对上面的TODO增加一个"统计区块"，如果是传统的MVC写法，你可能要新增一个statisticModel，然后在controller中的createTODO、deleteTODO中增加代码来操作这个新的statisticModel。而FLUX不用修改已有的任何代码，只需要写

新的store，并注册一些回调到createAction、deleteAction中就够了。所以可以看做是将MVC中的 "C主动操作M" 反转成 "M来决定何时运行"(当然这种情况也就没有C了)，但更好的是理解成是一种"事件系统"的变种。这就是它和MVC的区别。严格来说 FLUX 并不能算是facebook"发明"出来的，这样的模型在很多事件驱动的后端框架中很常见，如zero、yii，只不过拿到前端来作为应用架构时比较新颖。

FLUX是目前高度推荐的应用架构方式，它并没有强制使用的库或者框架，所以并不局限于react，在angular、polymer中同样能自由实现。特别是目前angular、polymer中的应用开发并没有一种应用架构的最佳实践。angular中的模块化既没有异步加载也没有作用域隔离的作用，实际使用时很鸡肋。但是angular中的依赖注入、filter、service的设计非常全面，如果再能加上FLUX的架构的话，威力不容小觑。对polymer来说情况更简单，应为polymer目前只考虑到element这一层，所以上层的应用架构可以自由实现。

值得补充的是，FLUX中的store，dispatcher可以更好地加强一下。store可以使用一些自动支持REST的库来简化开发，dispatcher可以使用支持自定义顺序等高级的事件代理实现。

5. 总结

2015将是前端框架相互借鉴相互融合的一年，随着webcomponent的落地，大家都在像标准靠近。提前储备这方面的技术肯定没有问题。再深入到框架的技术细节中，我们看到在"渲染机制"、"数据绑定"、"组件化"、"模块化"这些关键技术点中各个框架中都有非常精彩的实现，值得深入学习。React异军突起，也推荐持续关注，特别是在"应用架构"上，FLUX确实在整个业界起到了启发的作用，相信会越来越流行，并且有越来越多实现方式。