深入浅出 React

（一）：React的设计哲学 - 简单之美

> 原文：http://www.infoq.com/cn/articles/react-art-of-simplity
> 作者：王沛
>
> **编者按**：自2013年Facebook发布以来，React吸引了越来越多的开发者，基于它的衍生技术，如React Native、React Canvas等也层出不穷。InfoQ精心策划“深入浅出React”系列文章，为读者剖析React开发的技术细节。

React最初来自Facebook内部的广告系统项目，项目实施过程中前端开发遇到了巨大挑战，代码变得越来越臃肿且混乱不堪，难以维护。于是痛定思痛，他们决定抛开很多所谓的“最佳实践”，重新思考前端界面的构建方式，于是就有了React。

React带来了很多开创性的思路来构建前端界面，虽然选择React的最重要原因之一是性能，但是相关技术背后的设计思想更值得我们去思考。之前我也曾写过一篇React的入门文章，并提供了示例代码，大家可以结合参考。

目录

上个月React发布了最新的0.13版，并提供了对ES6的支持。在新版本中，一个小小的改变是React取消了函数的自动绑定，也就是说，以前可以这样去绑定一个事件：

```
<button onClick={this.handleSubmit}>Submit</button>
```

而在以ES6语法定义的组件中，必须写为：

```
<button onClick={this.handleSubmit.bind(this)}>Submit</button>
```

了解前端开发和JavaScript的同学都知道，做事件绑定时我们需要通过bind（或类似函数）来实现一个闭包以让事件处理函数自带上下文信息，这是由JavaScript语言特性决定的。而在0.13版本之前，React会自动在初始化时对组件的每一个方法做一次这样的绑定，类似于 `this.func = this.func.bind(this)` ，这样在

JSX的事件绑定中就可以直接写为 `onClick={this.handleSubmit}` 。

表面上看自动绑定给开发带来了便利，而Facebook却认为这破坏了JavaScript的语言习惯，其背后的神奇（Magic）逻辑或许会给初学者带来困惑，甚至开发者如果从React再转到其它库也可能会无所适从。基于同样的理由，React还取消了对mixin的支持，基于ES6的React组件不再能够以mixin的形式进行代码复用或者扩展。尽管这带来了很大不便，但Facebook认为mixin增加了代码的不可预测性，无法直观的去理解。关于mixin的思考，还可以参考这篇文章。

以简单直观、符合习惯的（idiomatic）方式去编程，让代码更容易被理解，从而易于维护和不断演进。这正是React的设计哲学。

编写可预测，符合习惯的代码

所谓可预测（predictable），即容易理解的代码。在年初的React开发者大会上，React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。

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。试想，一个月之后你回头看你写的代码，是否一眼就明白某个变量，某个if判断的含义；一个新加入的同事想去增加一个小小的新功能或是修复某个Bug，他是否对自己的代码有足够的信心不引入任何副作用？随着功能的增加，代码很容易变得越来越复杂，这些问题也将越来越严重，最终导致一份难以维护的代码。而React号称，新同事甚至在加入的第一天就能开始开发新功能。

那么React是如何做的呢？

使用JSX直观的定义用户界面

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。但是JSX给人的第一印象却是相当“丑陋”。当下面这样的例子被第一次展示的时候，甚至很多人称之为“巨大的退步（Huge Step Backwards）”：

```
var React = require('React');
var message =
  <div class="hello" onClick={someFunc}>
    <span>Hello World</span>
  </div>;
React.renderComponent(message, document.body);
```

将HTML直接嵌入到JavaScript代码中看上去确实是一件足够疯狂的事情。人们花了多年时间总结出的界面和业务逻辑相互分离的“最佳实践”就这么被彻底打破。那么React为何要如此另类？

模板出现的初衷是让非开发人员也能对界面做一定的修改。但这个初衷在当前Web程序里已完全不适用，每个模板背后的代码逻辑严重依赖模板中的内容和DOM结构，两者是紧密耦合的。即使做到文件位置的分

离，实际上两者还是一体的，并且为了两者之间的协作而不得不引入很多机制和概念。以Angularjs的首页示例代码为例：

```
<ul class="unstyled">
  <li ng-repeat="todo in todoList.todos">
    <input type="checkbox" ng-model="todo.done">
    <span class="done-{{todo.done}}">{{todo.text}}</span>
  </li>
</ul>
```

尽管我们很容易看懂这一小段模板的含义，但你却无法开始写这样的代码，因为你需要学习这一整套语法。比如说，你得知道有ng-repeat这样的标记的准确含义，其中的”todo in todoList.todos”看上去是repeat语法的一部分，或许还有其它语法存在；可以看到有{{todo.text}}这样的数据绑定，那么如果要对这段文本格式化（加一个formatter）该怎么做；另外，ng-model背后又需要什么样的数据结构？

现在来看React怎么写这段逻辑：

```
//...
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>
  );
}
//...
```

可以看到，JSX中除了另类的HTML标记之外，并没有引入其它任何新的概念（事实上HTML标记也可以完全用JavaScript去写）。Angular中的repeat在这里被一个简单的数组方法map所替代。在这里你可以利用熟悉的JavaScript语法去定义界面，在你的思维过程中其实已经不需要存在模板的概念，需要考虑的仅仅是如何用代码构建整个界面。这种自然而直观的方式直接降低了React的学习门槛并且让代码更容易理解。

简化的组件模型：所谓组件，其实就是状态机器

组件并不是一个新的概念，它意味着某个独立功能或界面的封装，达到复用、或是业务逻辑分离的目的。而React却这样理解界面组件：

> **所谓组件，就是状态机器**

> React将用户界面看做简单的状态机器。当组件处于某个状态时，那么就输出这个状态对应的界面。通过这种方式，就很容易去保证界面的一致性。
>
> 在React中，你简单的去更新某个组件的状态，然后输出基于新状态的整个界面。React负责以最高效的方式去比较两个界面并更新DOM树。

这种组件模型简化了我们思考的方式：对组件的管理就是对状态的管理。不同于其它框架模型，React组件很少需要暴露组件方法和外部交互。例如，某个组件有只读和编辑两个状态。一般的思路可能是提供 `beginEditing()` 和 `endEditing()` 这样的方法来实现切换；而在React中，需要做的是 `setState({editing: true/false})` 。在组件的输出逻辑中负责正确展现当前状态。这种方式，你不需要考虑beginEditing和endEditing中应该怎样更新UI，而只需要考虑在某个状态下，UI是怎样的。显然后者更加自然和直观。

组件是React中构建用户界面的基本单位。它们和外界的交互除了状态（state）之外，还有就是属性（props）。事实上，状态更多的是一个组件内部去自己维护，而属性则由外部在初始化这个组件时传递进来（一般是组件需要管理的数据）。React认为属性应该是只读的，一旦赋值过去后就不应该变化。关于状态和属性的使用在后续文章中还会深入探讨。

每一次界面变化都是整体刷新

数据模型驱动UI界面的两层编程模型从概念角度看上去是直观的，而在实际开发中却困难重重。一个数据模型的变化可能导致分散在界面多个角落的UI同时发生变化。界面越复杂，这种数据和界面的一致性越难维护。在Facebook内部他们称之为“Cascading Updates”，即层叠式更新，意味着UI界面之间会有一种互相依赖的关系。开发者为了维护这种依赖更新，有时不得不触发大范围的界面刷新，而其中很多并不真的需要。React的初衷之一就是，既然整体刷新一定能解决层叠更新的问题，那我们为什么不索性就每次都这么做呢？让框架自身去解决哪些局部UI需要更新的问题。这听上去非常有挑战，但React却做到了，实现途径就是通过虚拟DOM（Virtual DOM）。

关于虚拟DOM的原理我在去年底的文章有过比较详细的介绍，这里不再重复。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。

以这种思路出发，我们在考虑不断变化的UI界面时，仅仅需要整体考虑UI的构成。编程模型的简化带来的是代码的精简和易于理解，也即React不断提到的可预测（Predictable）的代码，代码的功能一目了然易于理解。Tom Occhino在2015 React开发者大会上也分享了React在Facebook内部的应用案例，随着新功能被不断的添加到系统中，开发进度非但没有变慢，甚至越来越快。

单向数据流动：Flux

既然已经有了组件机制去定义界面，那么还需要一定的机制来定义组件之间，以及组件和数据模型之间如何通信。为此，Facebook提出了Flux框架用于管理数据流。Flux是一个相当宽松的概念框架，同样符合

React简单直观的原则。不同于其它大多数MVC框架的双向数据绑定，Flux提倡的是单向数据流动，即永远只有从模型到视图的数据流动。

Flux引入了Dispatcher和Action的概念：Dispatcher是一个全局的分发器负责接收Action，而Store可以在Dispatcher上监听到Action并做出相应的操作。简单的理解可以认为类似于全局的消息发布订阅模型。Action可以来自于用户的某个界面操作，比如点击提交按钮；也可以来自服务器端的某个数据更新。当数据模型发生变化时，就触发刷新整个界面。

Flux的定义非常宽松，除了Facebook自己的实现之外，社区中还出现了很多Flux的不同实现，各有特点，比较流行的包括Flexible, Reflux, Flummox等等。

让数据模型也变简单：Immutability

Immutability含义是只读数据，React提倡使用只读数据来建立数据模型。这又是一个听上去相当疯狂的机制：所有数据都是只读的，如果需要修改它，那么你只能产生一份包含新的修改的数据。假设有如下数据：

```
var employee = {
  name: ‘John’,
  age: 28
};
```

如果要修改年龄，那么你需要产生一份新的数据：

```
var updated = {
  name: employee.name,
  age: 29
};
```

这样，原来的employee对象并没有发生任何变化，相反，产生了一个新的updated对象，体现了年龄发生了变化。这时候需要把新的updated对象应用到界面组件上来进行界面的更新。

只读数据并不是Facebook的全新发明，而是起源于Clojure, Scala, Haskell等函数式编程语言。只读的数据可以让代码更加的安全和易于维护，你不再需要担心数据在某个角落被某段神奇的代码所修改；也就不必再为了找到修改的地方而苦苦调试。而结合React，只读数据能够让React的组件仅仅通过比较对象引用是否相等来决定自身是否要重新Render。这在复杂的界面上可以极大的提高性能。

针对只读数据，Facebook开发了一整套框架immutable.js，将只读数据的概念引入JavaScript，并且在github开源。如果不希望一开始就引入这样一个较大的框架，React还提供了一个工具类插件，帮助管理和操作只读数据：React.addons.update。

React思想的衍生：React Native, React Canvas等等

在前几天的Facebook F8开发者大会上，React Native终于众望所归的发布，它将React的思想延伸到了原生移动开发。它的口号是“Learn Once, Write Anywhere”，有React开发经验的开发人员将可以无缝的进行React Native开发。无论是组件化的思想，调试工具，动态代码加载等React具有的强大特性都可以应用在React Native。相信这会对以后的移动开发布局产生重要影响。

React对UI层进行了完美的抽象，写Web界面时甚至能够做到完全的去DOM化：开发者可以无需进行任何DOM操作。因此，这也让对UI层进行整体替换成为了可能。React Native正是将浏览器基于DOM的UI层换成了iOS或者Android的原生控件。而Flipboard则将UI层换成了Canvas。

React Canvas是Flipboard出品的一套前端框架，所有的界面元素都通过Canvas来绘制，infoQ之前也有文章对其进行了介绍。Flipboard追求极致的性能和用户体验，因此对浏览器的缓慢DOM操作深恶痛绝，不惜大刀阔斧彻底舍弃了DOM，而完全用Canvas实现了整套UI控件。有兴趣的同学不妨一试。

小结

React并不是突然从哪里蹦出来，而是为了解决前端开发中的痛点而生。以简单为原则设计也决定了React具有极其平缓的学习曲线，开发者可以快速上手并应用到实际项目中。本文总结分析了其相关技术背后的设计思想，希望通过这个角度能让大家对React有一个总体的认识，从而在React的实际项目开发中，遵循简单直观的原则，进行高效率高质量的产品开发。

参考资料

1. React官方网站：http://facebook.github.io/react/
2. React博客：http://facebook.github.io/react/blog/

3. React入门：http://ryanclark.me/getting-started-with-react/
4. 颠覆式前端UI框架：React：http://www.infoq.com/cn/articles/subversion-front-end-ui-development-framework-react
5. Immutable.js: http://facebook.github.io/immutable-js/
6. React Native: http://facebook.github.io/react-native/
7. Flux: https://facebook.github.io/flux/
8. Flux框架对比：https://github.com/voronianski/flux-comparison
9. React开发者大会网站：http://conf.reactjs.com/index.html
10. React在Slack上的聊天社区：http://reactiflux.com/

感谢徐川对本文的审校。

（二）：React开发神器Webpack

> 原文：http://www.infoq.com/cn/articles/react-and-webpack
> 作者：王沛

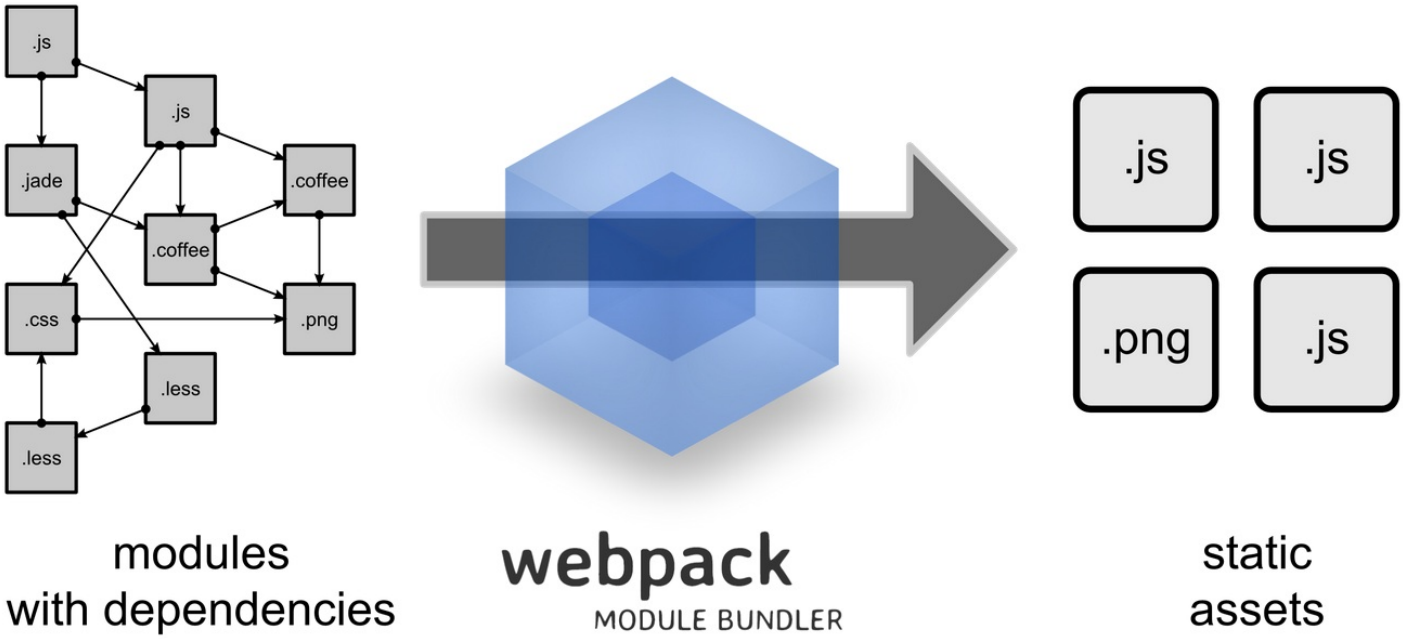
上一篇我们对React有了一个总体的认识，在介绍其中的技术细节之前，我们首先来了解一下用于React开发和模块管理的主流工具Webpack。称之为React开发神器有点标题党了，不过Webpack确实是笔者见过的功能最为强大的前端模块管理和打包工具。虽然Webpack是一个通用的工具，并不只适合于React，但是很多React的文章或者项目都使用了Webpack，尤其是react-hot-loader这样的神器存在，让Webpack成为最主流的React开发工具。

CommonJS和AMD是用于JavaScript模块管理的两大规范，前者定义的是模块的同步加载，主要用于NodeJS；而后者则是异步加载，通过requirejs等工具适用于前端。随着npm成为主流的JavaScript组件发布平台，越来越多的前端项目也依赖于npm上的项目，或者自身就会发布到npm平台。因此，让前端项目更方便的使用npm上的资源成为一大需求。于是诞生了类似browserify这样的工具，代码中可以使用require函数直接以同步语法形式引入npm模块，打包后再由浏览器执行。

Webpack其实有点类似browserify，出自Facebook的Instagram团队，但功能比browserify更为强大。其主要特性如下：

1. 同时支持CommonJS和AMD模块（对于新项目，推荐直接使用CommonJS）；
2. 串联式模块加载器以及插件机制，让其具有更好的灵活性和扩展性，例如提供对CoffeeScript、ES6的支持；
3. 可以基于配置或者智能分析打包成多个文件，实现公共模块或者按需加载；
4. 支持对CSS，图片等资源进行打包，从而无需借助Grunt或Gulp；
5. 开发时在内存中完成打包，性能更快，完全可以支持开发过程的实时打包需求；
6. 对sourcemap有很好的支持，易于调试。

Webpack将项目中用到的一切静态资源都视之为模块，模块之间可以互相依赖。Webpack对它们进行统一的管理以及打包发布，其官方主页用下面这张图来说明Webpack的作用：



可以看到Webpack的目标就是对项目中的静态资源进行统一管理，为产品的最终发布提供最优的打包部署方案。本文就将围绕React对其相关用法做一个总体介绍，从而能让你将其应用在自己的实际项目之中。

目录

安装Webpack，并加载一个简单的React组件

Webpack一般作为全局的npm模块安装：

```
npm install -g webpack
```

之后便有了全局的webpack命令，直接执行此命令会默认使用当前目录的webpack.config.js作为配置文件。如果要指定另外的配置文件，可以执行：

```
webpack —config webpack.custom.config.js
```

尽管Webpack可以通过命令行来指定参数，但我们通常会将所有相关参数定义在配置文件中。一般我们会定义两个配置文件，一个用于开发时，另外一个用于产品发布。生产环境下的打包文件不需要包含

sourcemap等用于开发时的代码。配置文件通常放在项目根目录之下，其本身也是一个标准的CommonJS模块。

一个最简单的Webpack配置文件webpack.config.js如下所示：

```
module.exports = {
  entry:[
    './app/main.js'
  ],
  output: {
    path: __dirname + '/assets/',
    publicPath: "/assets/",
    filename: 'bundle.js'
  }
};
```

其中entry参数定义了打包后的入口文件，数组中的所有文件会按顺序打包。每个文件进行依赖的递归查找，直到所有相关模块都被打包。output参数定义了输出文件的位置，其中常用的参数包括：

- **path**: 打包文件存放的绝对路径
- **publicPath**: 网站运行时的访问路径
- **filename**: 打包后的文件名

现在来看如何打包一个React组件。假设有如下项目文件夹结构：

```
- react-sample
  + assets/
   - js/
     Hello.js
     entry.js
   index.html
   webpack.config.js
```

其中Hello.js定义了一个简单的React组件，使用ES6语法：

```
var React = require('react');
class Hello extends React.Component {
  render() {
    return (
      <h1>Hello {this.props.name}!</h1>
    );
  }
}
```

entry.js是入口文件，将一个Hello组件输出到界面：

```
var React = require('react');
var Hello = require('./Hello');
React.render(<Hello name="Nate" />, document.body);
```

index.html的内容如下：

```
<html>
<head></head>
<body>
<script src="/assets/bundle.js"></script>
</body>
</html>
```

在这里Hello.js和entry.js都是JSX组件语法，需要对它们进行预处理，这就要引入webpack的JSX加载器。因此在配置文件中加入如下配置：

```
module: {
  loaders: [
    { test: /\.jsx?$/, loaders: ['jsx?harmony']}
  ]
}
```

加载器的概念稍后还会详细介绍，这里只需要知道它能将JSX编译成JavaScript并加载为Webpack模块。这样在当前目录执行webpack命令之后，在assets目录将生成bundle.js，打包了entry.js的内容。当浏览器打开当前服务器上的index.html，将显示“Hello Nate!”。这是一个非常简单的例子，演示了如何使用Webpack来进行最简单的React组件打包。

加载AMD或CommonJS模块

在实际项目中，代码以模块进行组织，AMD是在CommonJS的基础上考虑了浏览器的异步加载特性而产生的，可以让模块异步加载并保证执行顺序。而CommonJS的 `require` 函数则是同步加载。在Webpack中笔者更加推荐CommonJS方式去加载模块，这种方式语法更加简洁直观。即使在开发时，我们也是加载Webpack打包后的文件，通过sourcemap去进行调试。

除了项目本身的模块，我们也需要依赖第三方的模块，现在比较常用的第三方模块基本都通过npm进行发布，使用它们已经无需单独下载管理，需要时执行 `npm install` 即可。例如，我们需要依赖jQuery，只需执行：

```
npm install jquery —save-dev
```

更多情况下我们是在项目的package.json中进行依赖管理，然后通过直接执行npm install来安装所有依赖。这样在项目的代码仓库中并不需要存储实际的第三方依赖库的代码。

安装之后，在需要使用jquery的模块中需要在头部进行引入：

```
var $ = require('jquery');
$('body').html('Hello Webpack!');
```

可以看到，这种以CommonJS的同步形式去引入其它模块的方式代码更加简洁。浏览器并不会实际的去同步加载这个模块，require的处理是由Webpack进行解析和打包的，浏览器只需要执行打包后的代码。Webpack自身已经可以完全处理JavaScript模块的加载，但是对于React中的JSX语法，这就需要使用Webpack的扩展加载器来处理了。

Webpack开发服务器

除了提供模块打包功能，Webpack还提供了一个基于Node.js Express框架的开发服务器，它是一个静态资源Web服务器，对于简单静态页面或者仅依赖于独立服务的前端页面，都可以直接使用这个开发服务器进行开发。在开发过程中，开发服务器会监听每一个文件的变化，进行实时打包，并且可以推送通知前端页面代码发生了变化，从而可以实现页面的自动刷新。

Webpack开发服务器需要单独安装，同样是通过npm进行：

```
npm install -g webpack-dev-server
```

之后便可以运行webpack-dev-server命令来启动开发服务器，然后通过localhost:8080/webpack-dev-server/访问到页面了。默认情况下服务器以当前目录作为服务器目录。在React开发中，我们通常会结合react-hot-loader来使用开发服务器，因此这里不做太多介绍，只需要知道有这样一个开发服务器可以用于开发时的内容实时打包和推送。详细配置和用法可以参考官方文档。

Webpack模块加载器（Loaders）

Webpack将所有静态资源都认为是模块，比如
JavaScript，CSS，LESS，TypeScript，JSX，CoffeeScript，图片等等，从而可以对其进行统一管理。为此Webpack引入了加载器的概念，除了纯JavaScript之外，每一种资源都可以通过对应的加载器处理成模块。和大多数包管理器不一样的是，Webpack的加载器之间可以进行串联，一个加载器的输出可以成为另一个加载器的输入。比如LESS文件先通过less-load处理成css，然后再通过css-loader加载成css模块，最后由style-loader加载器对其做最后的处理，从而运行时可以通过style标签将其应用到最终的浏览器环境。

对于React的JSX也是如此，它通过jsx-loader来载入。jsx-loader专门用于载入React的JSX文件，Webpack的加载器支持参数，jsx-loader就可以添加?harmony参数使其支持ES6语法。为了让Webpack识别什么样的资源应该用什么加载器去载入，需要在配置文件进行配置：通过正则表达式对文件名进行匹配。例如：

```
module: {
    preLoaders: [{
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'jsxhint'
    }],
    loaders: [{
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'react-hot!jsx-loader?harmony'
    }, {
      test: /\.less/,
      loader: 'style-loader!css-loader!less-loader'
    }, {
      test: /\.(css)$/,
      loader: 'style-loader!css-loader'
    }, {
      test: /\.(png|jpg)$/,
      loader: 'url-loader?limit=8192'
    }]
  }
```

可以看到，该使用什么加载器完全取决于这里的配置，即使对于JSX文件，我们也可以用js作为后缀，从而所有的JavaScript都可以通过jsx-loader载入，因为jsx本身就是完全兼容JavaScript的，所以即使没有JSX语法，普通JavaScript模块也可以使用jsx-loader来载入。

加载器之间的级联是通过感叹号来连接，例如对于LESS资源，写法为style-loader!css-loader!less-loader。对于小型的图片资源，也可以将其进行统一打包，由url-loader实现，代码中`url-loader?limit=8192`含义就是对于所有小于8192字节的图片资源也进行打包。这在一定程度上可以替代Css Sprites方案，用于减少对于小图片资源的HTTP请求数量。

除了已有加载器，你也可以自己实现自己的加载器，从而可以让Webpack统一管理项目特定的静态资源。现在也已经有很多第三方的加载器实现常见静态资源的打包管理，可以参考Webpack主页上的加载器列表。

React开发神器：react-hot-loader

Webpack本身具有运行时模块替换功能，称之为Hot Module Replacement (HMR)。当某个模块代码发生变化时，Webpack实时打包将其推送到页面并进行替换，从而无需刷新页面就实现代码替换。这个过程相对比较复杂，需要进行多方面考虑和配置。而现在针对React出现了一个第三方react-hot-loader加载器，使用这个加载器就可以轻松实现React组件的热替换，非常方便。其实正是因为React的每一次更新都是全局刷新的虚拟DOM机制，让React组件的热替换可以成为通用的加载器，从而极大提高开发效率。

要使用react-hot-loader，首先通过npm进行安装：

```
npm install —save-dev react-hot-loader
```

之后，Webpack开发服务器需要开启HMR参数hot，为了方便，我们创建一个名为server.js的文件用以启动Webpack开发服务器：

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('../webpack.config');
new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  noInfo: false,
  historyApiFallback: true
}).listen(3000, '127.0.0.1', function (err, result) {
  if (err) {
    console.log(err);
  }
  console.log('Listening at localhost:3000');
});
```

为了热加载React组件，我们需要在前端页面中加入相应的代码，用以接收Webpack推送过来的代码模块，进而可以通知所有相关React组件进行重新Render。加入这个代码很简单：

```
entry: [
  'webpack-dev-server/client?http://127.0.0.1:3000', // WebpackDevServer host and port
  'webpack/hot/only-dev-server',
  './scripts/entry' // Your app's entry point
]
```

需要注意的是，这里的client?http://127.0.0.1:3000需要和在server.js中启动Webpack开发服务器的地址匹配。这样，打包生成的文件就知道该从哪里去获取动态的代码更新。下一步，我们需要让Webpack用react-hot-loader去加载React组件，如上一节所介绍，这通过加载器配置完成：

```
loaders: [{
    test: /\.js$/,
    exclude: /node_modules/,
    loader: 'react-hot!jsx-loader?harmony'
  },
  ...
]
```

做完这些配置之后，使用Node.js运行server.js：

```
node server.js
```

即可启动开发服务器并实现React组件的热加载。为了方便，我们也可以在package.json中加入一节配

置：

```
"scripts": {
  "start": "node ./js/server.js"
}
```

从而通过npm start命令即可启动开发服务器。示例代码也上传在Github上，大家可以参考。

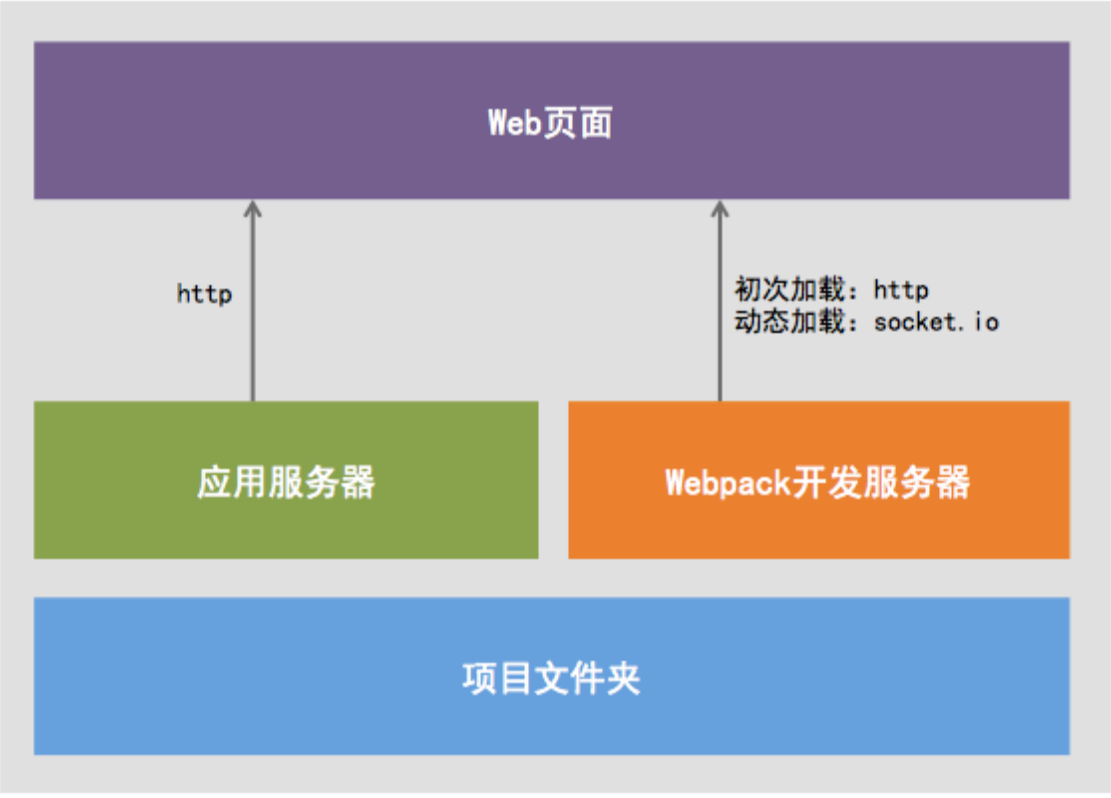
这样，React的热加载开发环境即配置完成，任何修改只要以保存，就会在页面上立刻体现出来。无论是对样式修改，还是对界面渲染的修改，甚至事件绑定处理函数的修改，都可以立刻生效，不得不说是提高开发效率的神器。

将Webpack开发服务器集成到已有服务器

尽管Webpack开发服务器可以直接用于开发，但实际项目中我们可能必须使用自己的Web服务器。这就需要我们能将Webpack的服务集成到已有服务器，来使用Webpack提供的模块打包和加载功能。要实现这一点其实非常容易，只需要在载入打包文件时指定完整的URL地址，例如：

```
<script src="http://127.0.0.1:3000/assets/bundle.js"></script>
```

这就告诉当前页面应该去另外一个服务器获得脚本资源文件，在之前我们已经在配置文件中指定了开发服务器的地址，因此打包后的文件也知道应该通过哪个地址去建立Socket IO来动态加载模块。整个资源架构如下图所示：



打包成多个资源文件

将项目中的模块打包成多个资源文件有两个目的：

1. 将多个页面的公用模块独立打包，从而可以利用浏览器缓存机制来提高页面加载效率；
2. 减少页面初次加载时间，只有当某功能被用到时，才去动态的加载。

Webpack提供了非常强大的功能让你能够灵活的对打包方案进行配置。首先来看如何创建多个入口文件：

```
{
  entry: { a: "./a", b: "./b" },
  output: { filename: "[name].js" },
  plugins: [ new webpack.CommonsChunkPlugin("init.js") ]
}
```

可以看到，配置文件中定义了两个打包资源“a”和“b”，在输出文件中使用方括号来获得输出文件名。而在插件设置中使用了CommonsChunkPlugin，Webpack中将打包后的文件都称之为“Chunk”。这个插件可以将多个打包后的资源中的公共部分打包成单独的文件，这里指定公共文件输出为“init.js”。这样我们就获得了三个打包后的文件，在html页面中可以这样引用：

```
<script src="init.js"></script>
<script src="a.js"></script>
<script src="b.js"></script>
```

除了在配置文件中对打包文件进行配置，还可以在代码中进行定义：require.ensure，例如：

```
require.ensure(["module-a", "module-b"], function(require) {
  var a = require("module-a");
  // ...
});
```

Webpack在编译时会扫描到这样的代码，并对依赖模块进行自动打包，运行过程中执行到这段代码时会自动找到打包后的文件进行按需加载。

小结

本文结合React介绍了Webpack的基本功能和用法，希望能让大家对这个新兴而强大的模块管理工具有一个总体的认识，并能将其应用在实际的项目开发中。笔者也将其应用在之前提供的React示例组件项目中，大家可以参考。除了这里介绍的功能，Webpack还有许多强大的特性，例如插件机制、支持动态表达式的require、打包文件的智能重组、性能优化、代码混淆等等。限于篇幅不再一一介绍，其官方文档也非常完善，需要时可以参考。

（三）：理解JSX和组件

> 原文：http://www.infoq.com/cn/articles/react-jsx-and-component

通过前两篇文章的介绍，相信大家对JSX和组件已经有了一定的了解。JSX这种混合使用JavaScript和XML的语言第一眼看上去很“丑”，也很神奇，但是其语法和背后的逻辑却极其简单。相信读完本文你就可以对JSX和组件有一个全面的了解，并能够用JSX来直观的构造用户界面。

目录

什么是JSX

React的核心机制之一就是虚拟DOM：可以在内存中创建的虚拟DOM元素。React利用虚拟DOM来减少对实际DOM的操作从而提升性能。类似于真实的原生DOM，虚拟DOM也可以通过JavaScript来创建，例如：

```
var child1 = React.createElement('li', null, 'First Text Content');
var child2 = React.createElement('li', null, 'Second Text Content');
var root = React.createElement('ul', { className: 'my-list' }, child1, child2);
```

使用这样的机制，我们完全可以用JavaScript构建完整的界面DOM树，正如我们可以用JavaScript创建真实DOM。但这样的代码可读性并不好，于是React发明了JSX，利用我们熟悉的HTML语法来创建虚拟DOM：

```
var root =(
  <ul className="my-list">
    <li>First Text Content</li>
    <li>Second Text Content</li>
  </ul>
);
```

这两段代码是完全等价的，后者将XML语法直接加入到JavaScript代码中，让你能够高效的通过代码而不是模板来定义界面。之后JSX通过翻译器转换到纯JavaScript再由浏览器执行。在实际开发中，JSX在产品打包阶段都已经编译成纯JavaScript，JSX的语法不会带来任何性能影响。另外，由于JSX只是一种语法，因此JavaScript的关键字class, for等也不能出现在XML中，而要如例子中所示，使用className, htmlFor代替，这和原生DOM在JavaScript中的创建也是一致的。

因此，JSX本身并不是什么高深的技术，可以说只是一个比较高级但很直观的语法糖。它非常有用，却不是一个必需品，没有JSX的React也可以正常工作：只要你乐意用JavaScript代码去创建这些虚拟DOM元素。

为什么使用JSX

前端界面的最基本功能在于展现数据，为此大多数框架都使用了模板引擎，例如在AngularJS中：

```
<div ng-if="person != null">
   Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
</div>
<div ng-if="person == null">
   Please log in.
</div>
```

在EmberJS中：

```
{{#if person}}
  Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
{{else}}
  Please log in.
{{/if}}
```

在Knockoutjs中：

```
<div data-bind="if: person != null">
   Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
</div>
<div data-bind="if: person == null">
   Please log in.
</div>
```

模板可以直观的定义UI来展现Model中的数据，你不必手动的去拼出一个很长的HTML字符串，几乎每种框架都有自己的模板引擎。传统MVC框架强调界面展示逻辑和业务逻辑的分离，因此为了应对复杂的展示逻辑需求，这些模板引擎几乎都不可避免的需要发展成一门独立的语言，如上面代码所示，每个框架都有自己的模板语言语法。而这无疑增加了框架的门槛和复杂度。

如果说掌握一种模板语言并不是很大的问题，那么其实由模板带来的架构复杂性则是让框架也变得复杂的重要原因之一，例如：

- 模板需要对应数据模型，即上下文，如何去绑定和实现？
- 模板可以嵌套，不同部分界面可能来自不同数据模型，如何处理？
- 模板语言终究是一个轻量级语言，为了满足项目需求，你很可能需要扩展模板引擎的功能。

为了解决这些复杂度，框架本身需要精心的设计，以及创造新的概念（例如Angular的Directive）。这些都会让框架变得复杂和难以掌握，不仅增加了开发成本，各种难以调试的Bug还会降低开发质量。

正因为如此，React直接放弃了模板而发明了JSX。看上去很像模板语言，但其本质是通过代码来构建界面，这使得我们不再需要掌握一门新的语言就可以直观的去定义用户界面：掌握了JavaScript就已经掌握了JSX。这里不妨再引用之前文章举过的例子，在展示一个列表时，模板语言通常提供名为Repeat的语法，例如在Angular中：

```
<ul class="unstyled">
  <li ng-repeat="todo in todoList.todos">
    <input type="checkbox" ng-model="todo.done">
    <span class="done-{{todo.done}}">{{todo.text}}</span>
  </li>
</ul>
```

而使用JSX，则代码如下：

```
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>
  );
});

var ul = (
  <ul class="unstyled">
    {lis}
  </ul>
);
```

可以看到，JSX完美利用了JavaScript自带的语法和特性，我们只要记住HTML只是代码创建DOM的一种语法形式，就很容易理解JSX。而这种使用代码构建界面的方式，完全消除了业务逻辑和界面元素之间的隔阂，让代码更加直观和易于维护。

JSX的语法

JSX本身就和XML语法类似，可以定义属性以及子元素。唯一特殊的是可以用大括号来加入JavaScript表达式，例如：

```
var person = <Person name={window.isLoggedIn ? window.name : ''} />;
```

一般每个组件都定义了一组属性（props，properties的简写）接收输入参数，这些属性通过XML标记的属性来指定。大括号中的语法就是纯JavaScript表达式，返回值会赋予组件的对应属性，因此可以使用任何JavaScript变量或者函数调用。上述代码经过JSX编译后会得到：

```
var person = React.createElement(
  Person,
  {name: window.isLoggedIn ? window.name : ''}
);
```

对于子元素也是类似，大括号中使用JavaScript表达式来返回需要展现的元素，例如文章开头提到的例子使用JSX可以写成：

```
var node = (
  <div className="container">
    {
      person ? <span>Welcome back, <b>{person.firstName} {person.lastName}</b>!</span>
      : <span>Please log in</span>
    }
  </div>
);
```

既然大括号中是JavaScript，而JSX又允许在JavaScript中使用XML，因此在大括号中仍然可以使用XML来声明组件，不断递归使用。

如果需要展现一组子节点，只需表达式返回一个JavaScript数组，数组的每个元素都是一个React组件，例如上一节的例子，其中lis就是有多个“li”元素的数组。：

```
var ul = (
  <ul class="unstyled">
    {lis}
  </ul>
);
```

在JSX中使用事件

如果你在90年代写过HTML，那么也许会有点怀念那时的事件绑定是多么的直观和简单：

```
<button onclick="checkAndSubmit(this.form)">Submit</button>
```

那时的JavaScript应用范围非常有限，最有用的也许就是做表单有效性验证。因为逻辑都很简单，直接写到HTML中并没有问题，而且这种方式非常直观易读。但是现在因为Web程序变的越来越复杂，我们就需

要使用JavaScript来绑定事件，例如在jQuery中：

```
$('#my-button').on('click', this.checkAndSubmit.bind(this));
```

在看到这段事件绑定和验证逻辑之前，你无法直观的看到有事件绑定在某个元素上，这种隐藏的界面元素和业务逻辑的耦合是很多Bug和内存泄露产生的根源。幸运的是，现在JSX可以让事件绑定返璞归真：

```
<button onClick={this.checkAndSubmit.bind(this)}>Submit</button>
```

和原生HTML定义事件的唯一区别就是JSX采用驼峰写法来描述事件名称，大括号中仍然是标准的JavaScript表达式，返回一个事件处理函数。在JSX中你不需要关心什么时机去移除事件绑定，因为React会在对应的真实DOM节点移除时就自动解除了事件绑定。

React并不会真正的绑定事件到每一个具体的元素上，而是采用事件代理的模式：在根节点document上为每种事件添加唯一的Listener，然后通过事件的target找到真实的触发元素。这样从触发元素到顶层节点之间的所有节点如果有绑定这个事件，React都会触发对应的事件处理函数。这就是所谓的React模拟事件系统。

尽管整个事件系统由React管理，但是其API和使用方法与原生事件一致。这种机制确保了跨浏览器的一致性：在所有浏览器（IE8及以上）都可以使用符合W3C标准的API，包括stopPropagation()，preventDefault()等等。对于事件的冒泡（bubble）和捕获（capture）模式也都完全支持。

在JSX中使用样式

尽管在大部分场景下我们应该将样式写在独立的CSS文件中，但是有时对于某个特定组件而言，其样式相当简单而且独立，那么也可以将其直接定义在JSX中。在JSX中使用样式和真实的样式也很类似，通过style属性来定义，但和真实DOM不同的是，属性值不能是字符串而必须为对象，例如：

```
<div style={{color: '#ff0000', fontSize: '14px'}}>Hello World.</div>
```

乍一看，这段JSX中的大括号是双的，有点奇怪，但实际上里面的大括号只是标准的JavaScript对象表达式，外面的大括号是JSX的语法。所以，样式你也可以先赋值给一个变量，然后传进去，代码会更易读：

```
var style = {
  color: '#ff0000',
  fontSize: '14px'
};

var node = <div style={style}>HelloWorld.</div>;
```

在JSX中可以使用所有的的样式，基本上属性名的转换规范就是将其写成驼峰写法，例如“background-

color”变为“backgroundColor”，“font-size”变为“fontSize”，这和标准的JavaScript操作DOM样式的API是一致的。

使用自定义组件

在JSX中，我们不仅可以使用React自带div, input...这些虚拟DOM元素，还可以自定义组件。组件定义之后，也都可以利用XML语法去声明，而能够使用的XML Tag就是在当前JavaScript上下文的变量名，这一点非常好用，你不必再去考虑某个Tag是如何对应到相应的组件实现。例如React官方教程中的例子：

```
class HelloWorld extends React.Component{
  render() {
    return (
      <p>
        Hello, <input type="text" placeholder="Your name here" />!
        It is {this.props.date.toTimeString()}
      </p>
    );
  }
};

setInterval(function() {
  React.render(
    <HelloWorld date={new Date()} />,
    document.getElementById('example')
  );
}, 500);
```

其中声明了一个名为HelloWorld的组件，那么就可以在XML中使用，这个Tag就是JavaScript变量名，我们可以用任意变量名：

```
var MyHelloWorld = HelloWorld;
React.render(<MyHelloWorld />, ...);
```

甚至，我们还可以引入命名空间：

```
var sampleNameSpace = {
  MyHelloWorld: HelloWorld
};
React.render(<sampleNameSpace.MyHelloWorld />, ...);
```

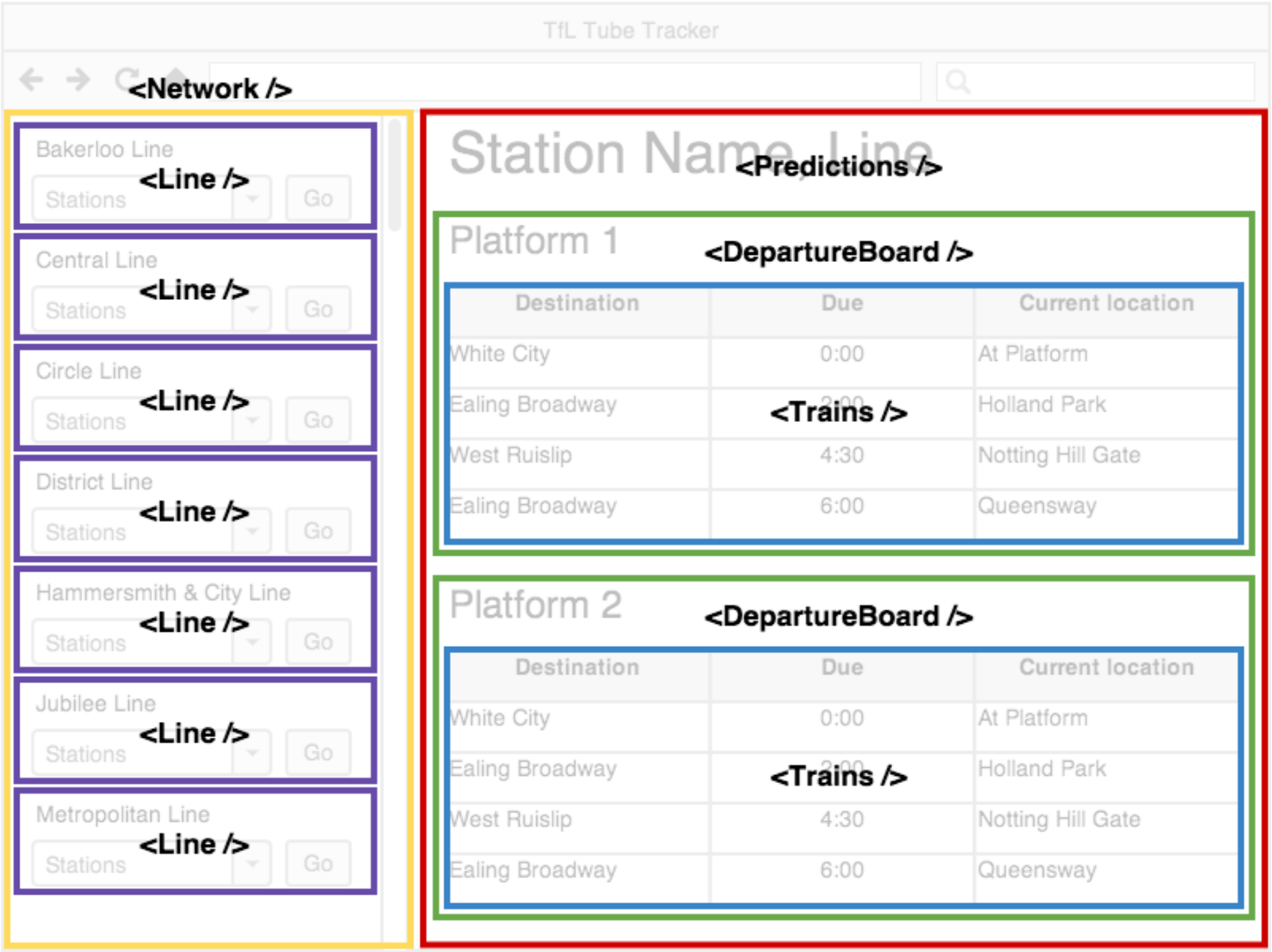
这些语法看上去有点怪，但是如果我们记住JSX语法只是JavaScript语法的一个语法映射，那么这些就非常容易理解了。

组件的概念和生命周期

React使用组件来封装界面模块，整个界面就是一个大组件，开发过程就是不断优化和拆分界面组件、构造

整个组件树的过程。可以认为组件类似于其他框架中Widget（或Control）的概念，但又有所不同。React中的界面一切皆为组件，而Widget一般只是嵌入到界面中为完成某个功能的独立模块。

如下图，整个页面是一个大的组件，然后再将其拆分成很多小的组件。组件机制加上JSX的语法，让你在构造界面时就像有一套符合项目需求的HTML标记，界面定义变得非常直观。



组件自身定义了一组props作为对外接口，展示一个组件时只需要指定props作为XML节点的属性。组件很少需要对外公开方法，唯一的交互途径就是props。这使得使用组件就像使用函数一样简单，给定一个输入，组件给定一个界面输出。当给予的参数一定时，那么输出也是一定的。而传统控件通常提供很多方法让你在外部改变控件的状态和行为，当控件的状态在不同场景不同逻辑中可以被随意控制时，开发和调试也会变得复杂。

而React组件通过唯一的props接口避免了逻辑复杂性，让开发测试都更加容易。这种特性完全得益于虚拟DOM机制，让你可以每次props改变都能以整体刷新页面的思路去考虑界面展现逻辑。

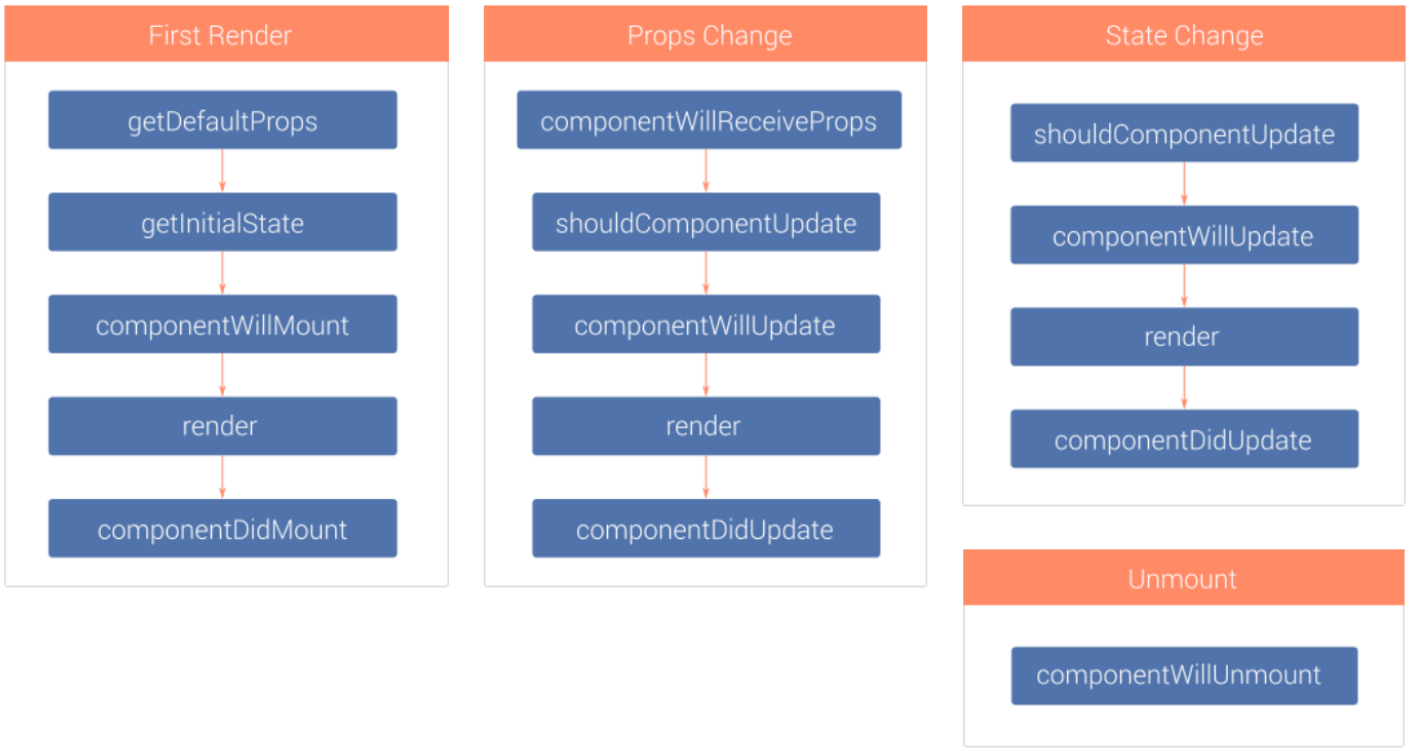
如果整个项目完全采用React，那么界面上就只有一个组件根节点；如果局部使用React，那么每个局部使用的部分都有一个根节点。在Render时，根节点由React.render函数去触发：

```
React.render(
  <App />,
  document.getElementById('react-root')
);
```

而所有的子节点则都是通过父节点的render方法去构造的。每个组件都会有一个render方法，这个方法返回组件的实例，最终整个界面得到一个虚拟DOM树，再由React以最高效的方式展现在界面上。

除了props之外，组件还有一个很重要的概念：state。组件规范中定义了setState方法，每次调用时都会更新组件的状态，触发render方法。需要注意，render方法是被异步调用的，这可以保证同步的多个setState方法只会触发一次render，有利于提高性能。和props不同，state是组件的内部状态，除了初始化时可能由props来决定，之后就完全由组件自身去维护。在组件的整个生命周期中，React强烈不推荐去修改自身的props，因为这会破坏UI和Model的一致性，props只能够由使用者来决定。

对于自定义组件，唯一必须实现的方法就是render()，除此之外，还有一些方法会在组件生命周期中被调用，如下图所示：



图中的方法几乎已经包括了React的所有API，自定义组件时根据需要在组件生命周期的不同阶段实现不同的逻辑。除了必须的render方法之外，其它常用的方法包括：

componentDidMount: 在组件第一次render之后调用，这时组件对应的DOM节点已被加入到浏览器。在这个方法里可以去实现一些初始化逻辑。

componentWillUnmount: 在DOM节点移除之后被调用，这里可以做一些相关的清理工作。

shouldComponentUpdate: 这是一个和性能非常相关的方法，在每一次render方法之前被调用。它提供了一个机会让你决定是否要对组件进行实际的render。例如：

```
shouldComponentUpdate(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

当此函数返回false时，组件就不会调用render方法从而避免了虚拟DOM的创建和内存中的Diff比较，从而有助于提高性能。当返回true时，则会进行正常的render的逻辑。

组件是React的核心，虽然功能很强大，但是其API和概念却十分简单，以至于你只要实现一个render方法就可以创建一个组件。这大大降低了React学习门槛。

使用Babel进行JSX编译

就在本文撰写过程中，React官方博客发布了一篇文章，声明其自身用于JSX语法解析的编译器JSTransform已经过期，不再维护，React JS和React Native已经全部采用第三方 Babel的JSX编译器实现。原因是两者在功能上已经完全重复，而Babel作为专门的JavaScript语法编译工具，提供了更为强大的功能。在这里笔者也不得不感叹Facebook的胸怀，以非常开放的态度去拥抱开源社区，从而达到共赢的目的。

JSX是一种新的语法，浏览器并不能直接运行，因此需要这种翻译器。在上一篇文章中我们推荐使用Webpack进行React的开发，要将JSX的编译器从JSTransform切换到Babel非常简单，首先通过npm安装Babel：

```
npm install —save-dev babel-loader
```

只需稍微改变一下webpack.config.js的配置，将原来的jsx-loader变为babel-loader：

```
module: {
  loaders: [
    { test: /\.jsx?$/, loaders: ['babel-loader']}
  ]
}
```

小结

本文主要介绍了React中最重要的组件机制，以及声明组件的语法JSX。看似有点神秘的JSX背后的原理非常简单：只是一种用于创建组件的XML语法。让代码直观易懂是软件项目质量的重要保证之一，这意味着代码更加容易理解和维护，出现Bug时更容易调试和修复。因此React这种采用JSX语法，以声明式的方法来直观的定义用户界面的方式，正是其最大的价值。

整个组件机制运行的基础是虚拟DOM，正因为React能够以极高的性能去比较两个虚拟DOM树的Diff，才实现了每次局部更新都通过刷新整个页面这种思考模式，降低了开发复杂度。在下一篇文章中就将会和大家一起研究虚拟DOM的Diff算法，了解其背后的运行原理。

（四）：虚拟DOM Diff算法解析

React中最神奇的部分莫过于虚拟DOM，以及其高效的Diff算法。这让我们可以无需担心性能问题而”毫无顾忌”的随时“刷新”整个页面，由虚拟DOM来确保只对界面上真正变化的部分进行实际的DOM操作。React在这一部分已经做到足够透明，在实际开发中我们基本无需关心虚拟DOM是如何运作的。然而，作为有态度的程序员，我们总是对技术背后的原理充满着好奇。理解其运行机制不仅有助于更好的理解React组件的生命周期，而且对于进一步优化React程序也会有很大帮助。

什么是DOM Diff算法

Web界面由DOM树来构成，当其中某一部分发生变化时，其实就是对应的某个DOM节点发生了变化。在React中，构建UI界面的思路是由当前状态决定界面。前后两个状态就对应两套界面，然后由React来比较两个界面的区别，这就需要对DOM树进行Diff算法分析。

即给定任意两棵树，找到最少的转换步骤。但是标准的的Diff算法复杂度需要O(n^3)，这显然无法满足性能要求。要达到每次界面都可以整体刷新界面的目的，势必需要对算法进行优化。这看上去非常有难度，然而Facebook工程师却做到了，他们结合Web界面的特点做出了两个简单的假设，使得Diff算法复杂度直接降低到O(n)

1. 两个相同组件产生类似的DOM结构，不同的组件产生不同的DOM结构；
2. 对于同一层次的一组子节点，它们可以通过唯一的id进行区分。

算法上的优化是React整个界面Render的基础，事实也证明这两个假设是合理而精确的，保证了整体界面构建的性能。

不同节点类型的比较

为了在树之间进行比较，我们首先要能够比较两个节点，在React中即比较两个虚拟DOM节点，当两个节点不同时，应该如何处理。这分为两种情况：（1）节点类型不同 ，（2）节点类型相同，但是属性不同。本节先看第一种情况。

当在树中的同一位置前后输出了不同类型的节点，React直接删除前面的节点，然后创建并插入新的节点。假设我们在树的同一位置前后两次输出不同类型的节点。

```
renderA: <div />
renderB: <span />
=> [removeNode <div />], [insertNode <span />]
```

当一个节点从div变成span时，简单的直接删除div节点，并插入一个新的span节点。这符合我们对真实DOM操作的理解。

需要注意的是，删除节点意味着彻底销毁该节点，而不是再后续的比较中再去看是否有另外一个节点等同于该删除的节点。如果该删除的节点之下有子节点，那么这些子节点也会被完全删除，它们也不会用于后面的比较。这也是算法复杂能够降低到O（n）的原因。

上面提到的是对虚拟DOM节点的操作，而同样的逻辑也被用在React组件的比较，例如：

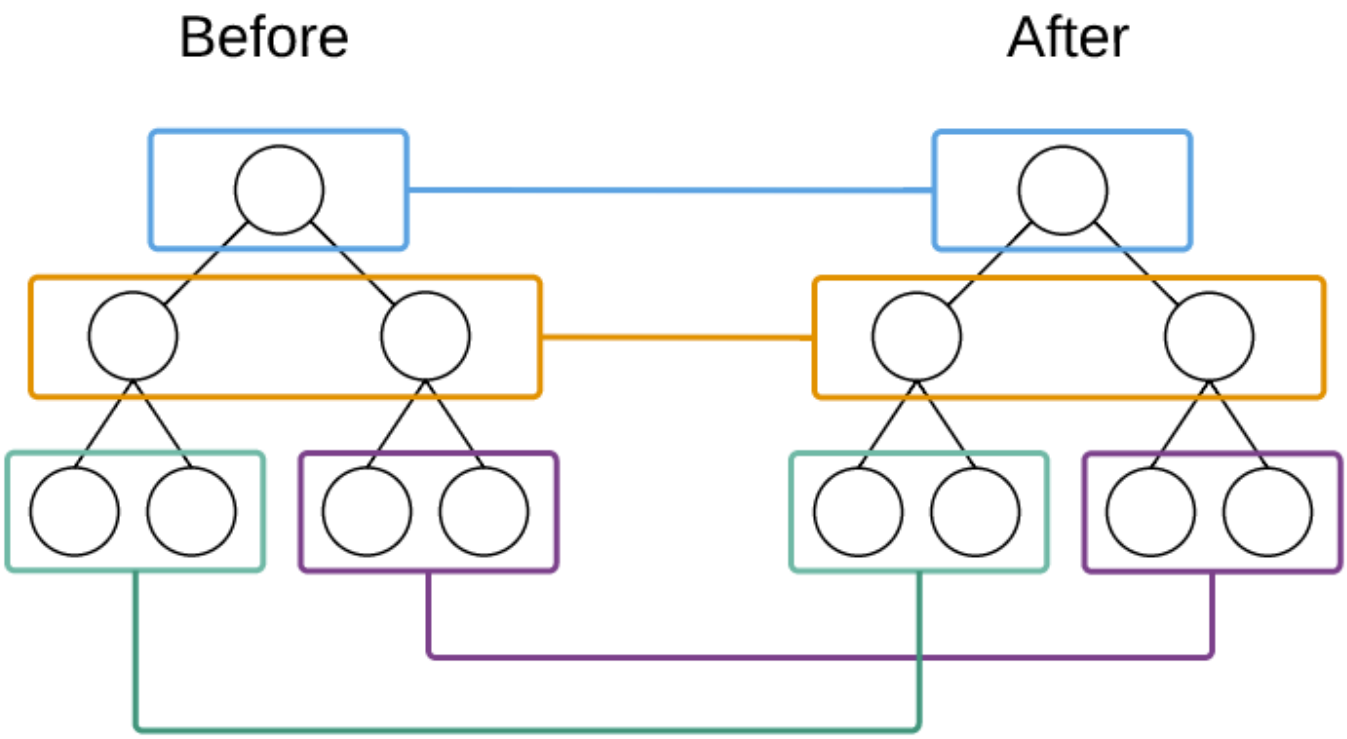
```
renderA: <Header />
renderB: <Content />
=> [removeNode <Header />], [insertNode <Content />]
```

当React在同一个位置遇到不同的组件时，也是简单的销毁第一个组件，而把新创建的组件加上去。这正是应用了第一个假设，不同的组件一般会产生不一样的DOM结构，与其浪费时间去比较它们基本上不会等价的DOM结构，还不如完全创建一个新的组件加上去。

由这一React对不同类型的节点的处理逻辑我们很容易得到推论，那就是React的DOM Diff算法实际上只会对树进行逐层比较，如下所述。

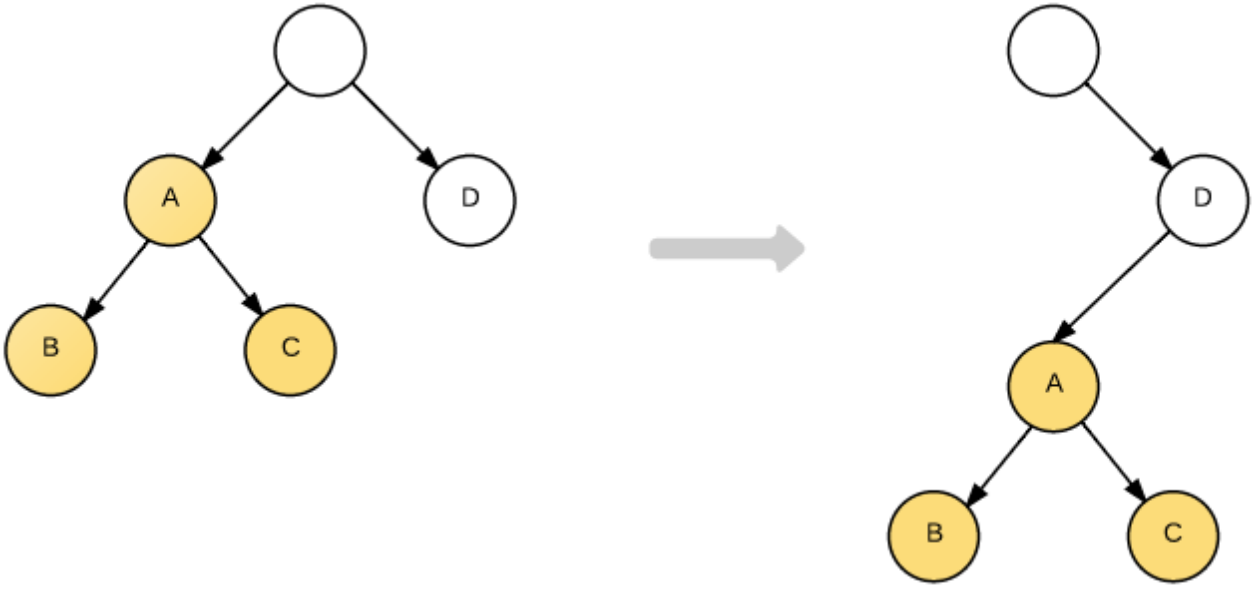
逐层进行节点比较

提到树，相信大多数同学立刻想到的是二叉树，遍历，最短路径等复杂的数据结构算法。而在React中，树的算法其实非常简单，那就是两棵树只会对同一层次的节点进行比较。如下图所示：



React只会对相同颜色方框内的DOM节点进行比较，即同一个父节点下的所有子节点。当发现节点已经不存在，则该节点及其子节点会被完全删除掉，不会用于进一步的比较。这样只需要对树进行一次遍历，便能完成整个DOM树的比较。

例如，考虑有下面的DOM结构转换：



A节点被整个移动到D节点下，直观的考虑DOM Diff操作应该是

```
A.parent.remove(A);
D.append(A);
```

但因为React只会简单的考虑同层节点的位置变换，对于不同层的节点，只有简单的创建和删除。当根节点

发现子节点中A不见了，就会直接销毁A；而当D发现自己多了一个子节点A，则会创建一个新的A作为子节点。因此对于这种结构的转变的实际操作是：

```
A.destroy();
A = new A();
A.append(new B());
A.append(new C());
D.append(A);
```

可以看到，以A为根节点的树被整个重新创建。

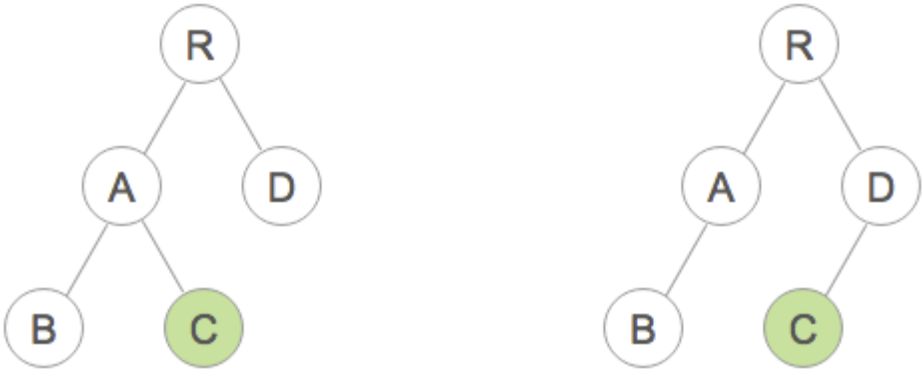
虽然看上去这样的算法有些“简陋”，但是其基于的是第一个假设：两个不同组件一般产生不一样的DOM结构。根据React官方博客，这一假设至今为止没有导致严重的性能问题。这当然也给我们一个提示，在实现自己的组件时，保持稳定的DOM结构会有助于性能的提升。例如，我们有时可以通过CSS隐藏或显示某些节点，而不是真的移除或添加DOM节点。

由DOM Diff算法理解组件的生命周期

在上一篇文章中介绍了React组件的生命周期，其中的每个阶段其实都是和DOM Diff算法息息相关的。例如以下几个方法：

- constructor: 构造函数，组件被创建时执行；
- componentDidMount: 当组件添加到DOM树之后执行；
- componentWillUnmount: 当组件从DOM树中移除之后执行，在React中可以认为组件被销毁；
- componentDidUpdate: 当组件更新时执行。

为了演示组件生命周期和DOM Diff算法的关系，笔者创建了一个示例：https://supnate.github.io/react-dom-diff/index.html ，大家可以直接访问试用。这时当DOM树进行如下转变时，即从“shape1”转变到“shape2”时。我们来观察这几个方法的执行情况：



浏览器开发工具控制台输出如下结果：

```
C will unmount.
C is created.
B is updated.
A is updated.
C did mount.
D is updated.
R is updated.
```

可以看到，C节点是完全重建后再添加到D节点之下，而不是将其“移动”过去。如果大家有兴趣，也可以fork示例代码：https://github.com/supnate/react-dom-diff 。从而可以自己添加其它树结构，试验它们之间是如何转换的。

相同类型节点的比较

第二种节点的比较是相同类型的节点，算法就相对简单而容易理解。React会对属性进行重设从而实现节点的转换。例如：

```
renderA: <div id="before" />
renderB: <div id="after" />
=> [replaceAttribute id "after"]
```

虚拟DOM的style属性稍有不同，其值并不是一个简单字符串而必须为一个对象，因此转换过程如下：

```
renderA: <div style={{color: 'red'}} />
renderB: <div style={{fontWeight: 'bold'}} />
=> [removeStyle color], [addStyle font-weight 'bold']
```

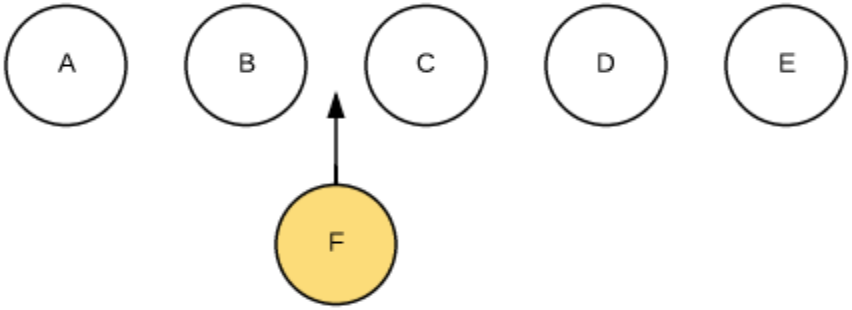
列表节点的比较

上面介绍了对于不在同一层的节点的比较，即使它们完全一样，也会销毁并重新创建。那么当它们在同一层时，又是如何处理的呢？这就涉及到列表节点的Diff算法。相信很多使用React的同学大多遇到过这样的警告：

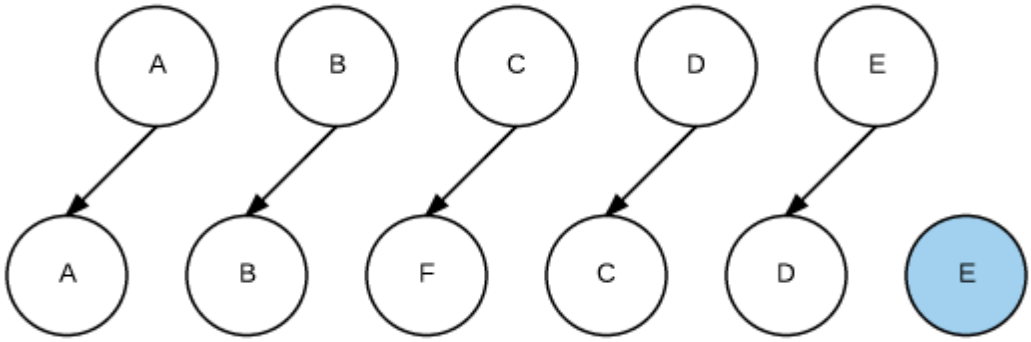
```
⚠ Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of Wrapper. See http://fb.me/react-warning-keys for more information.   runner-3.34.2.min.js:1
```

这是React在遇到列表时却又找不到key时提示的警告。虽然无视这条警告大部分界面也会正确工作，但这通常意味着潜在的性能问题。因为React觉得自己可能无法高效的去更新这个列表。

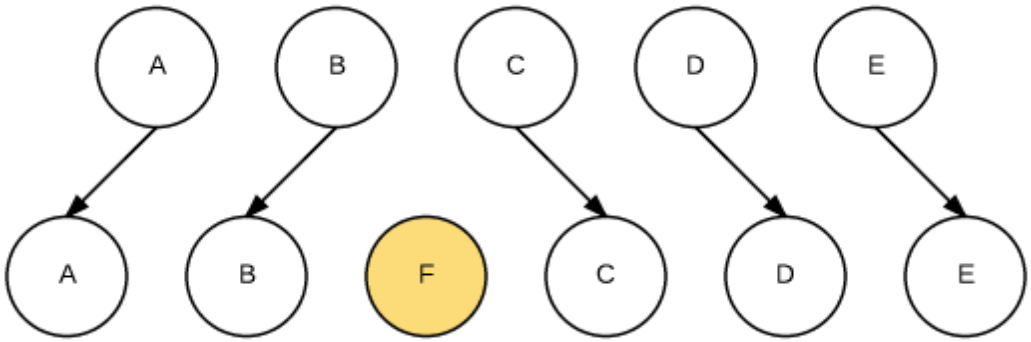
列表节点的操作通常包括添加、删除和排序。例如下图，我们需要往B和C直接插入节点F，在jQuery中我们可能会直接使用$(B).after(F)来实现。而在React中，我们只会告诉React新的界面应该是A-B-F-C-D-E，由Diff算法完成更新界面。



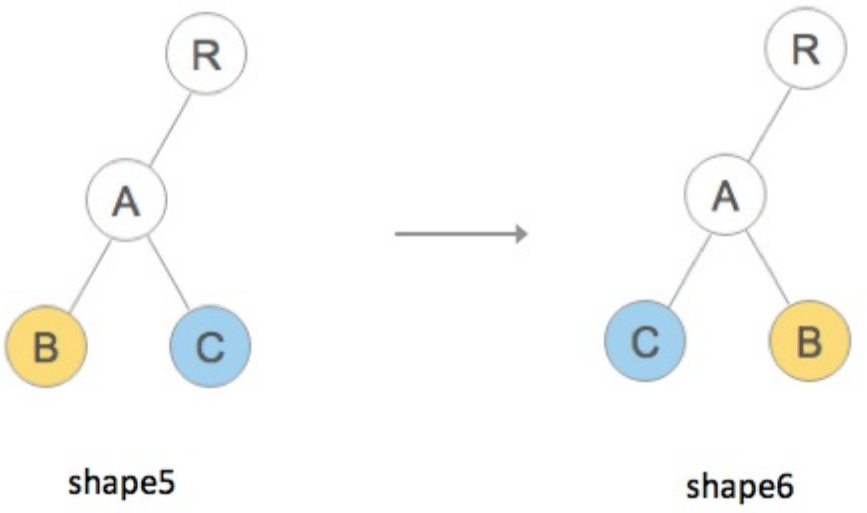
这时如果每个节点都没有唯一的标识，React无法识别每一个节点，那么更新过程会很低效，即，将C更新成F，D更新成C，E更新成D，最后再插入一个E节点。效果如下图所示：



可以看到，React会逐个对节点进行更新，转换到目标节点。而最后插入新的节点E，涉及到的DOM操作非常多。而如如果给每个节点唯一的标识（key），那么React能够找到正确的位置去插入新的节点，入下图所示：



对于列表节点顺序的调整其实也类似于插入或删除，下面结合示例代码我们看下其转换的过程。仍然使用前面提到的示例：https://supnate.github.io/react-dom-diff/index.html ，我们将树的形态从shape5转换到shape6：



即将同一层的节点位置进行调整。如果未提供key，那么React认为B和C之后的对应位置组件类型不同，因此完全删除后重建，控制台输出如下：

```
B will unmount.
C will unmount.
C is created.
B is created.
C did mount.
B did mount.
A is updated.
R is updated.
```

而如果提供了key，如下面的代码：

```
shape5: function() {
  return (
    <Root>
      <A>
        <B key="B" />
        <C key="C" />
      </A>
    </Root>
  );
},

shape6: function() {
  return (
    <Root>
      <A>
        <C key="C" />
        <B key="B" />
      </A>
    </Root>
  );
},
```

那么控制台输出如下：

```
C is updated.
B is updated.
A is updated.
R is updated.
```

可以看到，对于列表节点提供唯一的key属性可以帮助React定位到正确的节点进行比较，从而大幅减少DOM操作次数，提高了性能。

小结

本文分析了React的DOM Diff算法究竟是如何工作的，其复杂度控制在了O（n），这让我们考虑UI时可以完全基于状态来每次render整个界面而无需担心性能问题，简化了UI开发的复杂度。而算法优化的基础是文章开头提到的两个假设，以及React的UI基于组件这样的一个机制。理解虚拟DOM Diff算法不仅能够帮助我们理解组件的生命周期，而且也对我们实现自定义组件时如何进一步优化性能具有指导意义。