

React.js 教程

编者 xiedrsz



中科软信息系统有限公司

sinosoft

目录

[简介 2](#_Toc475368376)

[概述 2](#_Toc475368377)

[快速入门 10](#_Toc475368378)

[组件 11](#_Toc475368379)

[JSX 语法 17](#_Toc475368380)

[安装 21](#_Toc475368381)

[Hello World 22](#_Toc475368382)

# 简介

## 概述

注：未整理

React 是 Facebook 推出的一个用来构建用户界面的 JavaScript 库。具备以下特性：

不是一个 MVC 框架

不使用模板

响应式更新非常简单

HTML5 仅仅是个开始

仅仅是 UI

许多人使用 React 作为 MVC 架构的 V 层。 尽管 React 并没有假设过你的其余技术栈， 但它仍可以作为一个小特征轻易地在已有项目中使用

虚拟 DOM

React 为了更高超的性能而使用虚拟 DOM 作为其不同的实现。 它同时也可以由服务端 Node.js 渲染 － 而不需要过重的浏览器 DOM 支持

数据流

React 实现了单向响应的数据流，从而减少了重复代码，这也是它为什么比传统数据绑定更简单。

一个简单的组件

React 组件通过一个render()方法，接受输入的参数并返回展示的对象。

以下这个例子使用了 JSX，它类似于XML的语法

输入的参数通过render()传入组件后，将存储在this.props

JSX 是可选的，并不强制要求使用。

Live JSX Editor

var HelloMessage = React.createClass({

render: function() {

return <div>Hello {this.props.name}</div>;

}

});

React.render(<HelloMessage name="John" />, mountNode)

Hello John

Compiled JS

var HelloMessage = React.createClass({displayName: "HelloMessage",

render: function() {

return React.createElement("div", null, "Hello ", this.props.name);

}

});

React.render(React.createElement(HelloMessage, {name: "John"}), mountNode);

一个有状态的组件

除了接受输入数据（通过 this.props ），组件还可以保持内部状态数据（通过 this.state ）。当一个组件的状态数据的变化，展现的标记将被重新调用 render() 更新。

Live JSX Editor

var Timer = React.createClass({

getInitialState: function() {

return {secondsElapsed: 0};

},

tick: function() {

this.setState({secondsElapsed: this.state.secondsElapsed + 1});

},

componentDidMount: function() {

this.interval = setInterval(this.tick, 1000);

},

componentWillUnmount: function() {

clearInterval(this.interval);

},

render: function() {

return (

<div>Seconds Elapsed: {this.state.secondsElapsed}</div>

);

}

});

React.render(<Timer />, mountNode);

Compiled JS

var Timer = React.createClass({displayName: "Timer",

getInitialState: function() {

return {secondsElapsed: 0};

},

tick: function() {

this.setState({secondsElapsed: this.state.secondsElapsed + 1});

},

componentDidMount: function() {

this.interval = setInterval(this.tick, 1000);

},

componentWillUnmount: function() {

clearInterval(this.interval);

},

render: function() {

return (

React.createElement("div", null, "Seconds Elapsed: ", this.state.secondsElapsed)

);

}

});

React.render(React.createElement(Timer, null), mountNode);

一个应用程序

通过使用props和state, 我们可以组合构建一个小型的 Todo 程序。

下面例子使用state去监测当前列表的项以及用户已经输入的文本。 尽管事件绑定似乎是以内联的方式，但他们将被收集起来并以事件代理的方式实现。

Live JSX EditorCompiled JS

var TodoList = React.createClass({

render: function() {

var createItem = function(itemText) {

return <li>{itemText}</li>;

};

return <ul>{this.props.items.map(createItem)}</ul>;

}

});

var TodoApp = React.createClass({

getInitialState: function() {

return {items: [], text: ''};

},

onChange: function(e) {

this.setState({text: e.target.value});

},

handleSubmit: function(e) {

e.preventDefault();

var nextItems = this.state.items.concat([this.state.text]);

var nextText = '';

this.setState({items: nextItems, text: nextText});

},

render: function() {

return (

<div>

<h3>TODO</h3>

<TodoList items={this.state.items} />

<form onSubmit={this.handleSubmit}>

<input onChange={this.onChange} value={this.state.text} />

<button>{'Add #' + (this.state.items.length + 1)}</button>

</form>

</div>

);

}

});

React.render(<TodoApp />, mountNode);

Compiled JS

var TodoList = React.createClass({displayName: "TodoList",

render: function() {

var createItem = function(itemText) {

return React.createElement("li", null, itemText);

};

return React.createElement("ul", null, this.props.items.map(createItem));

}

});

var TodoApp = React.createClass({displayName: "TodoApp",

getInitialState: function() {

return {items: [], text: ''};

},

onChange: function(e) {

this.setState({text: e.target.value});

},

handleSubmit: function(e) {

e.preventDefault();

var nextItems = this.state.items.concat([this.state.text]);

var nextText = '';

this.setState({items: nextItems, text: nextText});

},

render: function() {

return (

React.createElement("div", null,

React.createElement("h3", null, "TODO"),

React.createElement(TodoList, {items: this.state.items}),

React.createElement("form", {onSubmit: this.handleSubmit},

React.createElement("input", {onChange: this.onChange, value: this.state.text}),

React.createElement("button", null, 'Add #' + (this.state.items.length + 1))

)

)

);

}

});

React.render(React.createElement(TodoApp, null), mountNode);

一个使用外部插件的组件

React 是灵活的，并且提供方法允许你跟其他库和框架对接。

下面例子展现了一个案例，使用外部库 Markdown 实时转化 textarea 的值。

Live JSX Editor

var converter = new Showdown.converter();

var MarkdownEditor = React.createClass({

getInitialState: function() {

return {value: 'Type some \*markdown\* here!'};

},

handleChange: function() {

this.setState({value: this.refs.textarea.getDOMNode().value});

},

render: function() {

return (

<div className="MarkdownEditor">

<h3>Input</h3>

<textarea

onChange={this.handleChange}

ref="textarea"

defaultValue={this.state.value} />

<h3>Output</h3>

<div

className="content"

dangerouslySetInnerHTML={{

\_\_html: converter.makeHtml(this.state.value)

}}

/>

</div>

);

}

});

React.render(<MarkdownEditor />, mountNode);

Compiled JS

var converter = new Showdown.converter();

var MarkdownEditor = React.createClass({displayName: "MarkdownEditor",

getInitialState: function() {

return {value: 'Type some \*markdown\* here!'};

},

handleChange: function() {

this.setState({value: this.refs.textarea.getDOMNode().value});

},

render: function() {

return (

React.createElement("div", {className: "MarkdownEditor"},

React.createElement("h3", null, "Input"),

React.createElement("textarea", {

onChange: this.handleChange,

ref: "textarea",

defaultValue: this.state.value}),

React.createElement("h3", null, "Output"),

React.createElement("div", {

className: "content",

dangerouslySetInnerHTML: {

\_\_html: converter.makeHtml(this.state.value)

}}

)

)

);

}

});

React.render(React.createElement(MarkdownEditor, null), mountNode);

## 快速入门

注：未整理

开始先下载入门教程包：<http://facebook.github.io/react/downloads/react-0.13.1.zip>

在入门教程包的根目录，创建一个含有下面代码的 helloworld.html。

<!DOCTYPE html>

<html>

<head>

<script src="build/react.js"></script>

<script src="build/JSXTransformer.js"></script>

</head>

<body>

<div id="example"></div>

<script type="text/jsx">

React.render(

<h1>Hello, world!</h1>,

document.getElementById('example')

);

</script>

</body>

</html>

在 JavaScript 代码里写着 XML 格式的代码称为 JSX。为了把 JSX 转成标准的 JavaScript，我们用 <script type="text/jsx"> 标签包裹着含有 JSX 的代码，然后引入 JSXTransformer.js 库来实现在浏览器里的代码转换。

分离文件

你的 React JSX 代码文件可以写在另外的文件里。新建下面的 src/helloworld.js。

React.render(

<h1>Hello, world!</h1>,

document.getElementById('example')

);

然后在 helloworld.html 引用该文件：

<script type="text/jsx" src="src/helloworld.js"></script>

离线转换

先安装命令行工具（依赖 npm）：

npm install -g react-tools

为什么使用 React?

React 是一个 Facebook 和 Instagram 用来创建用户界面的 JavaScript 库。很人多认为 React 是 MVC 中的 V（视图）。

我们创造 React 是为了解决一个问题：构建随着时间数据不断变化的大规模应用程序。

显示数据

用户界面能做的最基础的事就是显示一些数据。React 让显示数据变得简单，当数据变化的时候，用户界面会自动同步更新。

React 是不会去操作 DOM 的。它用一种更快的内置仿造的 DOM 来操作差异，为你计算出出效率最高的 DOM 改变。

## 组件

注：未整理

React 中全是模块化、可组装的组件。

我们通过 JavaScript 对象传递一些方法到 React.createClass() 来创建一个新的React组件。其中最重要的方法是 render，该方法返回一颗 React 组件树，这棵树最终将会渲染成 HTML。

这个 <div> 标签不是真实的DOM节点；他们是 React div 组件的实例。你可以认为这些就是React知道如何处理的标记或者一些数据。React 是安全的。我们不生成 HTML 字符串，因此默认阻止了 XSS 攻击。

你没有必要返回基本的 HTML。你可以返回一个你（或者其他人）创建的组件树。这就使得 React 变得组件化：一个关键的前端维护原则。

React.render() 实例化根组件，启动框架，注入标记到原始的 DOM 元素中，作为第二个参数提供。

制作组件

让我们为 CommentList 和 CommentForm 构建骨架，这也会是一些简单的 <div> ：

var CommentList = React.createClass({

render: function() {

return (

<div className="commentList">

Hello, world! I am a CommentList.

</div>

);

}

});

var CommentForm = React.createClass({

render: function() {

return (

<div className="commentForm">

Hello, world! I am a CommentForm.

</div>

);

}

});

下一步，更新 CommentBox 组件，使用这些新的组件：

var CommentBox = React.createClass({

render: function() {

return (

<div className="commentBox">

<h1>Comments</h1>

<CommentList />

<CommentForm />

</div>

);

}

});

HTML 组件就是普通的 React 组件，就像你定义的一样，只有一点不一样。JSX 编译器会自动重写 HTML 标签为 React.createElement(tagName) 表达式，其它什么都不做。这是为了避免全局命名空间污染。

组件属性

var CommentList = React.createClass({

render: function() {

return (

<div className="commentList">

<Comment author="Pete Hunt">This is one comment</Comment>

<Comment author="Jordan Walke">This is \*another\* comment</Comment>

</div>

);

}

});

请注意，我们已经从父节点 CommentList 组件传递给子节点 Comment 组件一些数据。例如，我们传递了 Pete Hunt （通过一个属性）和 This is one comment （通过类似于XML的子节点）给第一个 Comment。从父节点传递到子节点的数据称为 props，是属性（properties）的缩写。

使用props

让我们创建评论组件。通过 props，就能够从中读取到从 CommentList 传递过来的数据，然后渲染一些标记：

var Comment = React.createClass({

render: function() {

return (

<div className="comment">

<h2 className="commentAuthor">

{this.props.author}

</h2>

{this.props.children}

</div>

);

}

});

在 JSX 中通过将 JavaScript 表达式放在大括号中（作为属性或者子节点），你可以生成文本或者 React 组件到节点树中。我们访问传递给组件的命名属性作为 this.props 的键，任何内嵌的元素作为 this.props.children。

响应状态变化（Reactive state）

到目前为止，每一个组件都根据自己的 props 渲染了自己一次。props 是不可变的：它们从父节点传递过来，被父节点“拥有”。为了实现交互，我们给组件引进了可变的 state。this.state 是组件私有的，可以通过调用 this.setState() 来改变它。当状态更新之后，组件重新渲染自己。

render() methods are written declaratively as functions of this.props and this.state. 框架确保UI始终和输入保持一致。

当服务器获取数据的时候，我们将会用已有的数据改变评论。让我们给 CommentBox 组件添加一个评论数组作为它的状态：

var CommentBox = React.createClass({

getInitialState: function() {

return {data: []};

},

render: function() {

return (

<div className="commentBox">

<h1>Comments</h1>

<CommentList data={this.state.data} />

<CommentForm />

</div>

);

}

});

getInitialState()在组件的生命周期中仅执行一次，设置组件的初始化状态。

更新状态

var CommentBox = React.createClass({

getInitialState: function() {

return {data: []};

},

componentDidMount: function() {

$.ajax({

url: this.props.url,

dataType: 'json',

success: function(data) {

this.setState({data: data});

}.bind(this),

error: function(xhr, status, err) {

console.error(this.props.url, status, err.toString());

}.bind(this)

});

},

render: function() {

return (

<div className="commentBox">

<h1>Comments</h1>

<CommentList data={this.state.data} />

<CommentForm />

</div>

);

}

});

在这里，componentDidMount是一个在组件被渲染的时候React自动调用的方法。动态更新的关键点是调用this.setState()。我们把旧的评论数组替换成从服务器拿到的新的数组，然后UI自动更新。正是有了这种响应式，一个小的改变都会触发实时的更新。

使表单可交互

var CommentForm = React.createClass({

handleSubmit: function(e) {

e.preventDefault();

var author = this.refs.author.getDOMNode().value.trim();

var text = this.refs.text.getDOMNode().value.trim();

if (!text || !author) {

return;

}

// TODO: send request to the server

this.refs.author.getDOMNode().value = '';

this.refs.text.getDOMNode().value = '';

return;

},

render: function() {

return (

<form className="commentForm" onSubmit={this.handleSubmit}>

<input type="text" placeholder="Your name" ref="author" />

<input type="text" placeholder="Say something..." ref="text" />

<input type="submit" value="Post" />

</form>

);

}

});

事件

React使用驼峰命名规范的方式给组件绑定事件处理器。我们给表单绑定一个onSubmit处理器，用于当表单提交了合法的输入后清空表单字段。

在事件回调中调用preventDefault()来避免浏览器默认地提交表单。

Refs

我们利用Ref属性给子组件命名，this.refs引用组件。我们可以在组件上调用getDOMNode()获取浏览器本地的DOM元素。

组件就像是函数

React 组件非常简单。你可以认为它们就是简单的函数，接受 props 和 state (后面会讨论) 作为参数，然后渲染出 HTML。正是应为它们是这么的简单，这使得它们非常容易理解。

注意:

只有一个限制: React 组件只能渲染单个根节点。如果你想要返回多个节点，它们必须被包含在同一个节点里。

## JSX 语法

注：未整理

JSX 让你可以用 HTML 语法去写 JavaScript 函数调用 为了在 React 生成一个链接，通过纯 JavaScript 你可以这么写：

React.createElement('a', {href: 'http://facebook.github.io/react/'}, 'Hello React!')。

通过 JSX 这就变成了

<a href="http://facebook.github.io/react/">Hello React!</a>。

你完全可以选择是否使用 JSX，并不是 React 必须的。你可以通过 React.createElement 来创建一个树。第一个参数是标签，第二个参数是一个属性对象，每三个是子节点。

var child = React.createElement('li', null, 'Text Content');

var root = React.createElement('ul', { className: 'my-list' }, child);

React.render(root, document.getElementById('example'));

方便起见，你可以创建基于自定义组件的速记工厂方法。

var Factory = React.createFactory(ComponentClass);

...

var root = Factory({ custom: 'prop' });

React.render(root, document.getElementById('example'));

React 已经为 HTML 标签提供内置工厂方法。

var root = React.DOM.ul({ className: 'my-list' },

React.DOM.li(null, 'Text Content')

);

JSX 是一个看起来很像 XML 的 JavaScript 语法扩展。

HTML 标签对比 React 组件

React 可以渲染 HTML 标签 (strings) 或 React 组件 (classes)。

要渲染 HTML 标签，只需在 JSX 里使用小写字母开头的标签名。

var myDivElement = <div className="foo" />;

React.render(myDivElement, document.getElementById('example'));

要渲染 React 组件，只需创建一个大写字母开头的本地变量。

var MyComponent = React.createClass({/\*...\*/});

var myElement = <MyComponent someProperty={true} />;

React.render(myElement, document.getElementById('example'));

React 的 JSX 里约定分别使用首字母大、小写来区分本地组件的类和 HTML 标签。

注意:

由于 JSX 就是 JavaScript，一些标识符像 class 和 for 不建议作为 XML 属性名。作为替代，React DOM 使用 className 和 htmlFor 来做对应的属性。

注释

JSX 里添加注释很容易；它们只是 JS 表达式而已。你只需要在一个标签的子节点内(非最外层)小心地用 {} 包围要注释的部分。

var content = (

<Nav>

{/\* 一般注释, 用 {} 包围 \*/}

<Person

/\* 多

行

注释 \*/

name={window.isLoggedIn ? window.name : ''} // 行尾注释

/>

</Nav>

);

JSX 展开属性

如果你事先知道组件需要的全部 Props（属性），JSX 很容易地这样写：

var component = <Component foo={x} bar={y} />;

修改 Props 是不好的，明白吗

如果你不知道要设置哪些 Props，那么现在最好不要设置它：

var component = <Component />;

component.props.foo = x; // 不好

component.props.bar = y; // 同样不好

这样是反模式，因为 React 不能帮你检查属性类型（propTypes）。这样即使你的 属性类型有错误也不能得到清晰的错误提示。

Props 应该被当作禁止修改的。修改 props 对象可能会导致预料之外的结果，所以最好不要去修改 props 对象。

展开属性（Spread Attributes）

现在你可以使用 JSX 的新特性 - 展开属性：

var props = {};

props.foo = x;

props.bar = y;

var component = <Component {...props} />;

传入对象的属性会被复制到组件内。

它能被多次使用，也可以和其它属性一起用。注意顺序很重要，后面的会覆盖掉前面的。

var props = { foo: 'default' };

var component = <Component {...props} foo={'override'} />;

console.log(component.props.foo); // 'override'

JSX 陷阱

注：未整理

JSX 与 HTML 非常相似，但是有些关键区别要注意。

HTML 实体

HTML 实体可以插入到 JSX 的文本中。

<div>First &middot; Second</div>

如果想在 JSX 表达式中显示 HTML 实体，可以会遇到二次转义的问题，因为 React 默认会转义所有字符串，为了防止各种 XSS 攻击。

// 错误: 会显示 “First &middot; Second”

<div>{'First &middot; Second'}</div>

有多种绕过的方法。最简单的是直接用 Unicode 字符。这时要确保文件是 UTF-8 编码且网页也指定为 UTF-8 编码。

<div>{'First · Second'}</div>

安全的做法是先找到 实体的 Unicode 编号 ，然后在 JavaScript 字符串里使用。

<div>{'First \u00b7 Second'}</div>

<div>{'First ' + String.fromCharCode(183) + ' Second'}</div>

可以在数组里混合使用字符串和 JSX 元素。

<div>{['First ', <span>&middot;</span>, ' Second']}</div>

万不得已，可以直接使用原始 HTML。

<div dangerouslySetInnerHTML={{'{{'}}\_\_html: 'First &middot; Second'}} />

## 事件

注：未整理

简单例子

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>

);

}

});

React.render(

<LikeButton />,

document.getElementById('example')

);

事件处理与合成事件（Synthetic Events）

React 里只需把事件处理器（event handler）以骆峰命名（camelCased）形式当作组件的 props 传入即可，就像使用普通 HTML 那样。React 内部创建一套合成事件系统来使所有事件在 IE8 和以上浏览器表现一致。也就是说，React 知道如何冒泡和捕获事件，而且你的事件处理器接收到的 events 参数与 W3C 规范 一致，无论你使用哪种浏览器。

幕后原理：自动绑定（Autobinding）和事件代理（Event Delegation）

在幕后，React 做了一些操作来让代码高效运行且易于理解。

Autobinding: 在 JavaScript 里创建回调的时候，为了保证 this 的正确性，一般都需要显式地绑定方法到它的实例上。有了 React，所有方法被自动绑定到了它的组件实例上。React 还缓存这些绑定方法，所以 CPU 和内存都是非常高效。而且还能减少打字！

事件代理 ： React 实际并没有把事件处理器绑定到节点本身。当 React 启动的时候，它在最外层使用唯一一个事件监听器处理所有事件。当组件被加载和卸载时，只是在内部映射里添加或删除事件处理器。当事件触发，React 根据映射来决定如何分发。当映射里处理器时，会当作空操作处理。参考 David Walsh 很棒的文章 了解这样做高效的原因。

组件其实是状态机（State Machines）

React 把用户界面当作简单状态机。把用户界面想像成拥有不同状态然后渲染这些状态，可以轻松让用户界面和数据保持一致。

React 里，只需更新组件的 state，然后根据新的 state 重新渲染用户界面（不要操作 DOM）。React 来决定如何最高效地更新 DOM。

State 工作原理

常用的通知 React 数据变化的方法是调用 setState(data, callback)。这个方法会合并（merge） data 到 this.state，并重新渲染组件。渲染完成后，调用可选的 callback 回调。大部分情况下不需要提供 callback，因为 React 会负责把界面更新到最新状态。

哪些组件应该有 State？

大部分组件的工作应该是从 props 里取数据并渲染出来。但是，有时需要对用户输入、服务器请求或者时间变化等作出响应，这时才需要使用 State。

尝试把尽可能多的组件无状态化。 这样做能隔离 state，把它放到最合理的地方，也能减少冗余并，同时易于解释程序运作过程。

常用的模式是创建多个只负责渲染数据的无状态（stateless）组件，在它们的上层创建一个有状态（stateful）组件并把它的状态通过 props 传给子级。这个有状态的组件封装了所有用户的交互逻辑，而这些无状态组件则负责声明式地渲染数据。

哪些 应该 作为 State？

State 应该包括那些可能被组件的事件处理器改变并触发用户界面更新的数据。 真实的应用中这种数据一般都很小且能被 JSON 序列化。当创建一个状态化的组件时，想象一下表示它的状态最少需要哪些数据，并只把这些数据存入 this.state。在 render() 里再根据 state 来计算你需要的其它数据。你会发现以这种方式思考和开发程序最终往往是正确的，因为如果在 state 里添加冗余数据或计算所得数据，需要你经常手动保持数据同步，不能让 React 来帮你处理。

this.state 应该仅包括能表示用户界面状态所需的最少数据。因些，它不应该包括：

计算所得数据： 不要担心根据 state 来预先计算数据 —— 把所有的计算都放到 render() 里更容易保证用户界面和数据的一致性。例如，在 state 里有一个数组（listItems），我们要把数组长度渲染成字符串， 直接在 render() 里使用 this.state.listItems.length + ' list items' 比把它放到 state 里好的多。

React 组件： 在 render() 里使用当前 props 和 state 来创建它。

基于 props 的重复数据： 尽可能使用 props 来作为惟一数据来源。把 props 保存到 state 的一个有效的场景是需要知道它以前值的时候，因为未来的 props 可能会变化。

## 复合组件

注：未整理

目前为止，我们已经学了如何用单个组件来展示数据和处理用户输入。下一步让我们来体验 React 最激动人心的特性之一：可组合性（composability）。

动机：关注分离

通过复用那些接口定义良好的组件来开发新的模块化组件，我们得到了与使用函数和类相似的好处。具体来说就是能够通过开发简单的组件把程序的不同关注面分离。如果为程序开发一套自定义的组件库，那么就能以最适合业务场景的方式来展示你的用户界面。

组合实例

一起来使用 Facebook Graph API 开发显示个人图片和用户名的简单 Avatar 组件吧。

var Avatar = React.createClass({

render: function() {

return (

<div>

<ProfilePic username={this.props.username} />

<ProfileLink username={this.props.username} />

</div>

);

}

});

var ProfilePic = React.createClass({

render: function() {

return (

<img src={'http://graph.facebook.com/' + this.props.username + '/picture'} />

);

}

});

var ProfileLink = React.createClass({

render: function() {

return (

<a href={'http://www.facebook.com/' + this.props.username}>

{this.props.username}

</a>

);

}

});

React.render(

<Avatar username="pwh" />,

document.getElementById('example')

);

从属关系

上面例子中，Avatar 拥有 ProfilePic 和 ProfileLink 的实例。拥有者 就是给其它组件设置 props 的那个组件。更正式地说， 如果组件 Y 在 render() 方法是创建了组件 X，那么 Y 就拥有 X。上面讲过，组件不能修改自身的 props - 它们总是与它们拥有者设置的保持一致。这是保持用户界面一致性的关键性原则。

把从属关系与父子关系加以区别至关重要。从属关系是 React 特有的，而父子关系简单来讲就是 DOM 里的标签的关系。在上一个例子中，Avatar 拥有 div、ProfilePic 和 ProfileLink 实例，div 是 ProfilePic 和 ProfileLink 实例的父级（但不是拥有者）。

子级

实例化 React 组件时，你可以在开始标签和结束标签之间引用在 React 组件或者 Javascript 表达式：

<Parent><Child /></Parent>

Parent 能通过专门的 this.props.children props 读取子级。this.props.children 是一个不透明的数据结构： 通过 React.Children 工具类来操作。

子级校正（Reconciliation）

校正就是每次 render 方法调用后 React 更新 DOM 的过程。 一般情况下，子级会根据它们被渲染的顺序来做校正。例如，下面代码描述了两次渲染的过程：

// 第一次渲染

<Card>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</Card>

// 第二次渲染

<Card>

<p>Paragraph 2</p>

</Card>

直观来看，只是删除了<p>Paragraph 1</p>。事实上，React 先更新第一个子级的内容，然后删除最后一个组件。React 是根据子级的顺序来校正的。

子组件状态管理

对于大多数组件，这没什么大碍。但是，对于使用 this.state 来在多次渲染过程中里维持数据的状态化组件，这样做潜在很多问题。

多数情况下，可以通过隐藏组件而不是删除它们来绕过这些问题。

// 第一次渲染

<Card>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</Card>

// 第二次渲染

<Card>

<p style={{'{{'}}display: 'none'}}>Paragraph 1</p>

<p>Paragraph 2</p>

</Card>

动态子级

如果子组件位置会改变（如在搜索结果中）或者有新组件添加到列表开头（如在流中）情况会变得更加复杂。如果子级要在多个渲染阶段保持自己的特征和状态，在这种情况下，你可以通过给子级设置惟一标识的 key 来区分。

render: function() {

var results = this.props.results;

return (

<ol>

{results.map(function(result) {

return <li key={result.id}>{result.text}</li>;

})}

</ol>

);

}

当 React 校正带有 key 的子级时，它会确保它们被重新排序（而不是破坏）或者删除（而不是重用）。 务必 把 key 添加到子级数组里组件本身上，而不是每个子级内部最外层 HTML 上：

// 错误！

var ListItemWrapper = React.createClass({

render: function() {

return <li key={this.props.data.id}>{this.props.data.text}</li>;

}

});

var MyComponent = React.createClass({

render: function() {

return (

<ul>

{this.props.results.map(function(result) {

return <ListItemWrapper data={result}/>;

})}

</ul>

);

}

});

// 正确 :)

var ListItemWrapper = React.createClass({

render: function() {

return <li>{this.props.data.text}</li>;

}

});

var MyComponent = React.createClass({

render: function() {

return (

<ul>

{this.props.results.map(function(result) {

return <ListItemWrapper key={result.id} data={result}/>;

})}

</ul>

);

}

});

也可以传递 object 来做有 key 的子级。object 的 key 会被当作每个组件的 key。但是一定要牢记 JavaScript 并不总是保证属性的顺序会被保留。实际情况下浏览器一般会保留属性的顺序，除了 使用 32 位无符号数字做为 key 的属性。数字型属性会按大小排序并且排在其它属性前面。一旦发生这种情况，React 渲染组件的顺序就是混乱。可能在 key 前面加一个字符串前缀来避免：

render: function() {

var items = {};

this.props.results.forEach(function(result) {

// 如果 result.id 看起来是一个数字（比如短哈希），那么

// 对象字面量的顺序就得不到保证。这种情况下，需要添加前缀

// 来确保 key 是字符串。

items['result-' + result.id] = <li>{result.text}</li>;

});

return (

<ol>

{items}

</ol>

);

}

数据流

React 里，数据通过上面介绍过的 props 从拥有者流向归属者。这就是高效的单向数据绑定(one-way data binding)：拥有者通过它的 props 或 state 计算出一些值，并把这些值绑定到它们拥有的组件的 props 上。因为这个过程会递归地调用，所以数据变化会自动在所有被使用的地方自动反映出来。

性能提醒

你或许会担心如果一个拥有者有大量子级时，对于数据变化做出响应非常耗费性能。值得庆幸的是执行 JavaScript 非常的快，而且 render() 方法一般比较简单，所以在大部分应用里这样做速度极快。此外，性能的瓶颈大多是因为 DOM 更新，而非 JS 执行，而且 React 会通过批量更新和变化检测来优化性能。

但是，有时候需要做细粒度的性能控制。这种情况下，可以重写 shouldComponentUpdate() 方法返回 false 来让 React 跳过对子树的处理。参考 React reference docs 了解更多。

注意：

如果在数据变化时让 shouldComponentUpdate() 返回 false，React 就不能保证用户界面同步。当使用它的时候一定确保你清楚到底做了什么，并且只在遇到明显性能问题的时候才使用它。不要低估 JavaScript 的速度，DOM 操作通常才是慢的原因。

## 可复用组件

注：未整理

设计接口的时候，把通用的设计元素（按钮，表单框，布局组件等）拆成接口良好定义的可复用的组件。这样，下次开发相同界面程序时就可以写更少的代码，也意义着更高的开发效率，更少的 Bug 和更少的程序体积。

Prop 验证

随着应用不断变大，保证组件被正确使用变得非常有用。为此我们引入 propTypes。React.PropTypes 提供很多验证器 (validator) 来验证传入数据的有效性。当向 props 传入无效数据时，JavaScript 控制台会抛出警告。注意为了性能考虑，只在开发环境验证 propTypes。下面用例子来说明不同验证器的区别：

React.createClass({

propTypes: {

// 可以声明 prop 为指定的 JS 基本类型。默认

// 情况下，这些 prop 都是可传可不传的。

optionalArray: React.PropTypes.array,

optionalBool: React.PropTypes.bool,

optionalFunc: React.PropTypes.func,

optionalNumber: React.PropTypes.number,

optionalObject: React.PropTypes.object,

optionalString: React.PropTypes.string,

// 所有可以被渲染的对象：数字，

// 字符串，DOM 元素或包含这些类型的数组。

optionalNode: React.PropTypes.node,

// React 元素

optionalElement: React.PropTypes.element,

// 用 JS 的 instanceof 操作符声明 prop 为类的实例。

optionalMessage: React.PropTypes.instanceOf(Message),

// 用 enum 来限制 prop 只接受指定的值。

optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),

// 指定的多个对象类型中的一个

optionalUnion: React.PropTypes.oneOfType([

React.PropTypes.string,

React.PropTypes.number,

React.PropTypes.instanceOf(Message)

]),

// 指定类型组成的数组

optionalArrayOf: React.PropTypes.arrayOf(React.PropTypes.number),

// 指定类型的属性构成的对象

optionalObjectOf: React.PropTypes.objectOf(React.PropTypes.number),

// 特定形状参数的对象

optionalObjectWithShape: React.PropTypes.shape({

color: React.PropTypes.string,

fontSize: React.PropTypes.number

}),

// 以后任意类型加上 `isRequired` 来使 prop 不可空。

requiredFunc: React.PropTypes.func.isRequired,

// 不可空的任意类型

requiredAny: React.PropTypes.any.isRequired,

// 自定义验证器。如果验证失败需要返回一个 Error 对象。不要直接

// 使用 `console.warn` 或抛异常，因为这样 `oneOfType` 会失效。

customProp: function(props, propName, componentName) {

if (!/matchme/.test(props[propName])) {

return new Error('Validation failed!');

}

}

},

/\* ... \*/

});

默认 Prop 值

React 支持以声明式的方式来定义 props 的默认值。

var ComponentWithDefaultProps = React.createClass({

getDefaultProps: function() {

return {

value: 'default value'

};

}

/\* ... \*/

});

当父级没有传入 props 时，getDefaultProps() 可以保证 this.props.value 有默认值，注意 getDefaultProps 的结果会被 缓存。得益于此，你可以直接使用 props，而不必写手动编写一些重复或无意义的代码。

传递 Props：小技巧

有一些常用的 React 组件只是对 HTML 做简单扩展。通常，你想少写点代码来把传入组件的 props 复制到对应的 HTML 元素上。这时 JSX 的 spread 语法会帮到你：

var CheckLink = React.createClass({

render: function() {

// 这样会把 CheckList 所有的 props 复制到 <a>

return <a {...this.props}>{'√ '}{this.props.children}</a>;

}

});

React.render(

<CheckLink href="/checked.html">

Click here!

</CheckLink>,

document.getElementById('example')

);

单个子级

React.PropTypes.element 可以限定只能有一个子级传入。

var MyComponent = React.createClass({

propTypes: {

children: React.PropTypes.element.isRequired

},

render: function() {

return (

<div>

{this.props.children} // 有且仅有一个元素，否则会抛异常。

</div>

);

}

});

Mixins

组件是 React 里复用代码最佳方式，但是有时一些复杂的组件间也需要共用一些功能。有时会被称为 跨切面关注点。React 使用 mixins 来解决这类问题。

一个通用的场景是：一个组件需要定期更新。用 setInterval() 做很容易，但当不需要它的时候取消定时器来节省内存是非常重要的。React 提供 生命周期方法 来告知组件创建或销毁的时间。下面来做一个简单的 mixin，使用 setInterval() 并保证在组件销毁时清理定时器。

var SetIntervalMixin = {

componentWillMount: function() {

this.intervals = [];

},

setInterval: function() {

this.intervals.push(setInterval.apply(null, arguments));

},

componentWillUnmount: function() {

this.intervals.map(clearInterval);

}

};

var TickTock = React.createClass({

mixins: [SetIntervalMixin], // 引用 mixin

getInitialState: function() {

return {seconds: 0};

},

componentDidMount: function() {

this.setInterval(this.tick, 1000); // 调用 mixin 的方法

},

tick: function() {

this.setState({seconds: this.state.seconds + 1});

},

render: function() {

return (

<p>

React has been running for {this.state.seconds} seconds.

</p>

);

}

});

React.render(

<TickTock />,

document.getElementById('example')

);

关于 mixin 值得一提的优点是，如果一个组件使用了多个 mixin，并用有多个 mixin 定义了同样的生命周期方法（如：多个 mixin 都需要在组件销毁时做资源清理操作），所有这些生命周期方法都保证会被执行到。方法执行顺序是：首先按 mixin 引入顺序执行 mixin 里方法，最后执行组件内定义的方法。

## 传递 Props

注：未整理

React 里有一个非常常用的模式就是对组件做一层抽象。组件对外公开一个简单的属性（Props）来实现功能，但内部细节可能有非常复杂的实现。

可以使用 JSX 展开属性 来合并现有的 props 和其它值：

return <Component {...this.props} more="values" />;

如果不使用 JSX，可以使用一些对象辅助方法如 ES6 的 Object.assign 或 Underscore \_.extend。

return Component(Object.assign({}, this.props, { more: 'values' }));

下面的教程介绍一些最佳实践。使用了 JSX 和 ES7 的还在试验阶段的特性。

手动传递

大部分情况下你应该显式地向下传递 props。这样可以确保只公开你认为是安全的内部 API 的子集。

var FancyCheckbox = React.createClass({

render: function() {

var fancyClass = this.props.checked ? 'FancyChecked' : 'FancyUnchecked';

return (

<div className={fancyClass} onClick={this.props.onClick}>

{this.props.children}

</div>

);

}

});

React.render(

<FancyCheckbox checked={true} onClick={console.log.bind(console)}>

Hello world!

</FancyCheckbox>,

document.getElementById('example')

);

在 JSX 里使用 ... 传递

有时把所有属性都传下去是不安全或啰嗦的。这时可以使用解构赋值中的剩余属性特性来把未知属性批量提取出来。

列出所有要当前使用的属性，后面跟着 ...other。

var { checked, ...other } = this.props;

这样能确保把所有 props 传下去，除了 那些已经被使用了的。

注意:

上面例子中，checked 属性也是一个有效的 DOM 属性。如果你没有使用解构赋值，那么可能无意中把它传下去。

在传递这些未知的 other 属性时，要经常使用解构赋值模式。

var FancyCheckbox = React.createClass({

render: function() {

var fancyClass = this.props.checked ? 'FancyChecked' : 'FancyUnchecked';

// 反模式：`checked` 会被传到里面的组件里

return (

<div {...this.props} className={fancyClass} />

);

}

});

使用和传递同一个 Prop

如果组件需要使用一个属性又要往下传递，可以直接使用 checked={checked} 再传一次。这样做比传整个 this.props 对象要好，因为更利于重构和语法检查。

var FancyCheckbox = React.createClass({

render: function() {

var { checked, title, ...other } = this.props;

var fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';

var fancyTitle = checked ? 'X ' + title : 'O ' + title;

return (

<label>

<input {...other}

checked={checked}

className={fancyClass}

type="checkbox"

/>

{fancyTitle}

</label>

);

}

});

注意:

顺序很重要，把 {...other} 放到 JSX props 前面会使它不被覆盖。上面例子中我们可以保证 input 的 type 是 "checkbox"。

## 安装

1. 独立版本

下载对应的vue.js，然后使用<script>标签引入即可。

1. npm

# 最新稳定版本

$ npm install vue

# 最新稳定 CSP 兼容版本

$ npm install vue@csp

1. 命令行工具

# 全局安装 vue-cli

$ npm install -g vue-cli

# 创建一个基于 "webpack" 模板的新项目

$ vue init webpack my-project

# 安装依赖，走你

$ cd my-project

$ npm install

$ npm run dev

1. bower

# 最新稳定版本

$ bower install vue

## Hello World

1. node.js 工程

依据 安装 中的 三 的流程一次执行，执行完毕后即可生成一个基于node.js的vue工程。