# 第二章 JSX

在React中，组件是用于分离关注点的，而不是被当作模板或处理显示逻辑的。在使用React时。你必须接受这样的事实，那就是HTML标签以及生成这些标签的代码之间存在着内在的紧密联系。该设计允许你在构建标签结构时充分利用JavaScript的强大能力，而不必在笨拙的模板语言上浪费时间。

React包含了一种可选的类HTML标记语言。不过在继续之前，我们要先说清楚一件事-------对于那些厌恶在javaScript中写HTML标签以及那些还不明确JSX用处的人，请考虑在React中使用JSX的下列好处：

A. 允许使用使用熟悉的语法来定义HTML元素树。

B. 允许更加语义化且易懂的标签。

C. 程序结构更容易被直观化。

D. 抽象了React Element的创建过程。

E. 可以随时掌控HTML标签以及生成这些标签的代码。

F. 是原生的JavaScript。

在本章里我们会探索JSX的诸多优点，如何使用JSX，以及将它与HTML区分开来的一些注意事项。记住，JSX并不是必需的。如果你决定不使用它，则可以直接跳到本章结尾了解在React中不使用JSX的一些小提示。

## 什么是JSX

JSX即JavaScript XML------一种在React组件内部构建标签的类XML语法。React在不使用JSX的情况一样可以工作，然而使用JSX可以提高组件的可读性，因此推荐你使用JSX。

举个例子，在不使用JSX的React程序中创建一个标题的函数调用大概是这个样子：

//v0.11

React.DOM.h1({className:’quest’},’Questions’);

//v0.12

React.createElement(‘h1’,{className:’question’},’Questions’);

如果使用了JSX，上述调用就变成了下面这种更熟悉且且简练的标签：

<h1 className=”question”>Questions</h1>

与以往在JavaScript中嵌入HTML标签的几种方案相比，JSX有以下几点明显特征：

1、JSX是一种语法变换------每一个JSX节点对应着一个JavaScript函数。

2、JSX即不提供也不需要运行时库。

3、JSX并没有改变或添加javascript的语义------它只是简单的函数调用而已。

与html的相似之处赋予了JSX在React中强大的表现力。下面我们将要讨论使用JSX的好处以及它在程序中发挥的作用，同时还会讨论JSX与HTML的关键区别。

## 使用JSX的好处

当讨论JSX时，很多人会问到的问题就是-------为什么要用它，为什么在已经有那么多模板语言的情况下还要使用JSX，为什么不直接使用原生的javaScript。毕竟，JSX最后只是被简单地转换成对应的javaScript函数而已。

使用JSX有很多好处，而且这些好处会随着代码库的日益增大、组件的愈加复杂而变得越来越明显。我们来看看这些好处究竟是什么。

### 更加熟悉

许多团队包括了非开发人员，例如熟悉HTML的UI及UX设计师和负责完整测试产品的质量保证人员。使用JSX之后，这些团队成员都可以更轻松地阅读和贡献代码。任何熟悉基于XML语言的人都能轻松地掌握JSX。

此外，由于React组件囊括了所有可能的DOM表现形式，因此JSX能巧妙地用简单明了的方式来展现这种结构。

### 更加语义化

除了更加熟悉外，JSX还能够将JavaScript代码转化为更加语义化、更加有意义的标签。这种设计为我们提供了使用类HTML语法来声明组件和数据流向的能力，我们知道它们后续转换为原生的JavaScript。

JSX允许你在应用程序中使用所有预定义的HTML5标签及自定义组件。稍后会讲述更多关于自定义组件的内容，而这里只是简单地说明JSX是如何做到让JavaScript更具有可读性的。

举个例子，让我们设想一个Divider元素，它会渲染出一个位于左边的标题和一个撑满右边的水平分割线。这个Divider的HTML结构大概是下面的样子：

<div className=’divider’>

<h2>Questions</h2><hr />

</div>

把上面的HTML包裹进一个Divider React Component后，你就可以像使用其他任何HTML元素一样使用它了。相比原生的HTML，Diveider提供了更丰富的语义。

<Divider>Questions</Divder>

### 更加直观

即使像上述例子一样的小组件，JSX也能让它更加简单、明了、直观。在上百个组件及更深层标签树的大项目中，这种好处会被成倍地放大。

下面是前面提到的Divider组件。我们注意到在函数作用域内，使用JSX语法的版本与使用原生JavaScript相比，其标签的意图变得更加直观，可读性也更高。

以下是原生JavaScript版本：

//v0.11

Render:function(){

Return React.DOM.div({className:”divider”}),

“Label Text”,

React.DOM.hr()

}

//v0.12

Render:function(){

Return React.createElement(‘div’,{className:”divider”}

“Label Text”,

React.createElement(‘hr’)

);

}

以下是使用JSX的版本：

Render:function(){

Return <div className=”divider”>

Label Text<hr/>

</div>;

}

很多人认为JSX版本更加易懂，也更容易调试。

### 抽象化

注意，在前面的几个例子中，我们提供了React0.11和React0.12两个不同版本的JavaScript代码，而JSX在两个版本中都能正常运行。这是因为JSX的编译器抽象了将标签转化为JavaScript的过程。对于使用JSX的人来说，从React0.11升级到React0.12是无痛的------不需要修改任何代码。

虽然不是灵丹妙药，但JSX提供的抽象能力确实能够减少代码在项目开发过程中的改动。

### 关注点分离

最后，也是React的核心，指在HTML标签以及生成这些标签的代码内在地紧密联系在一起。在React中，你不需要把整个应用程序甚至单个组件的关注点分离成视图和模板文件。相反，React鼓励你为每一个关注点创造一个独立的组件，并把所有的逻辑和标签封装在其中。

JSX以干净且简洁的方式保证了组件中的标签与所有业务逻辑的相互分离。它不仅提供了一个清晰直观的方式来描述组件树，同时还让你的应用程序更加符合逻辑。

### 复合组件

目前为止，我们已经看到了使用JSX的若干好处，同时也看到了它是如何用简洁的标记格式来表示一个组件的。下面我们看看JSX如何帮我们组装多个组件。

这个小节包含了以下内容：

A.在JavaScript文件中包含JSX的准备工作。

B.详细说明组件的组装过程。

C.讨论组件的所有权以及父/子组件关系。

### 定义一个自定义组件

继续来看我们之前提到的分页组件，下面再次列出我们期望输出的HTML。

<div className=”divider”>

<h2>Questions</h2><hr />

</div>

要将这个HTML片段表示为React Component，你只需要把它像下面这样包装起来，然后在render方法中返回这些标签。

Var Divider=React.createClass({

Render:function(){

Return (

<div className=”divider”>

<h2>Questions</h2><hr />

</div>

);

}

});

当然目前这还只是一个一次性的组件。要让这个组件变得实用，还需要一种将h2标签中的文本表示出来的动态方法。

### 使用动态值

JSX将两个花括号之间的内容{...}渲染为动态值。花括号指明了一个JavaScript上下文环境------你在花括号中放入的任何东西都会被进行求值，得到的结果被渲染为标签中的若干节点。

对于简单值，比如文本或者数字，你可以直接引用对应的变量。可以像下面这样渲染一个动态的H2标签：

Var text=’Questions’;

<h2>{text}</h2>

//<h2>Question</h2>

对于更复杂的逻辑，你可能更倾向于将其转化为一个函数进行求解。可以通过花括号中调用这个函数来渲染期望的结果：

Function dateToString(d){

Return [

d.getFullYear(),

d.getMonth()+1,

d.getDate()

].join(‘-’);

};

<h2>{dateToString(new Date())}</h2>

//<h2>2014-10-18</h2>

Reat通过将数组中的每个元素渲染为一个节点的方式对数组进行自动求值。

Var text=[‘hello’,’world’];

<h2>{text}</h2>

比起简单值，我们通常希望渲染一些更复杂的数据。比如说，你可能希望把数组中的所有数据渲染为若干个<li>元素。这就要说到子节点了。

### 子节点

在HTML中，使用<h2>Questions</h2>来渲染一个header元素，这里的“Questions”就是h2元素的子文本节点。而在JSX中，我们的目标就是用下面的方式来表示它：

<Divider>Questions</Divider>

React将开始标签与结束标签之间的所有子节点保存在一个名为this.props.children的特殊组件属性中。在这个例子中，this.Props.children==[“Questions”]。

掌握这一点后，你就可以将硬编码的“Questions”转为变量this.props.children了。现在React会把你放在<Divider>标签之间的任何东西渲染出来。

Var Divider=React.createClass({

Render:function(){

Return(

<div className=”divider”>

<h2>{this.props.children}</h2><hr/>

);

}

});

至此，你就可以像使用任何HTML元素一样使用<Divider>组件了。

<Divider>Questions</Divider>

当我们把上面的JSX代码转换为JavaScript时，会得到下面的结果：

Var Divider=React.createClass({displayName:’Divider’,

Render:function(){

Return(

React.createElement(“div”,

{className:”divider”},

React.createElement(“h2”,null,this.props.children),

React.createElement(“hr”,null)

)

)

}

})

而最终渲染输出的结果正如你所期待的那样：

<div className=”divider”>

<h2>Questions</h2><hr/>

</div>

### JSX与HTML有何不同

JSX很像HTML，但却不是HTML语法的完美复制品。实际上，JSX规范中这样声明：

这个规范（JSX）并不尝试去遵循任何XML或HTML规范。JSX是作为一种ECMAScript特性来设计的，至于大家觉得JSF像XML这一事实，那仅仅是因为大家比较熟悉XML。

下面我们探索一下JSX与HTML语法上的几点关键区别。

#### 属性

在html中我们用内联的方式给每个节点设置属性，像这样：

<div id=”some-id” class=”some-class-name”>…</div>

JSX以同样的方式实现了属性的设置，同时还提供了将属性设置为动态JavaScript的便利。要设置动态的属性，你需要把原本用引号括起来的文本替换成花括号包裹的JavaScript变量。

Var surveyQuestionId=this.props.id;

Var classes=’some-class-name’;

…

<div id={ surveyQuestionId } className={classes}>…<div>

对于更复杂的情景，你还可以把属性设置为一个函数调用返回的结果。

<div id={this.getSurveyId()}>…</div>

现在，React每渲染一个组件时，我们指定的变量和函数就会被求值，而最终生成的DOM结构会反应出这个新的状态。

#### 条件判断

在React中，一个组件的HTML标签与生成这些标签的代码内在地紧密联系在一起。这意味着你可以轻松地利用JavaScript强大的魔力，比如循环和条件判断。

要想在组件中添加条件判断似乎是件很困难的事，因为if/else逻辑很难用HTML标签来表达。直接往JSX中加入if语句会渲染出无效的JavaScript：

<div className=”if(isComplete){‘is-complete’}”>…</div>

而解决的办法就是使用以下某种办法：

A. 使用三目运算符

B. 设置一个变量并在属性中引用它

C. 将逻辑转化到逻辑中去

D. 使用&&运算符

##### 使用三目运算符

…

Render: function(){

Return <div className={

This.state.isComplete?’is-complete’:’’

}>…</div>

}

…

虽然对于文本来说三目运算符可以正常运行，但是如果你想要在其他情况下很好的应用React Component，三目运算符就可能显得笨重又麻烦了。对于这些情况最好是使用下面的方法。

##### 使用变量

…

getIsComplete:function(){

return this.state.isComplete?’is-complete’:’’;

},

Render:function(){

Var is Complete=this.getIscomplete();

Return <div className={isComplete}>…</div>;

}

…

### 使用函数

…

getIsComplete:function(){

return this.state.isComplete?’is-complete’:’’;

},

Render:function(){

Return <div className={this.getIsComplete}>…</div>

}

…

#### 使用逻辑与（&&）运算符

由于对于null或false值React不会输出任何内容，因此你可以使用一个后面跟随了期望字符串的布尔值来实现条件判断。如果这个布尔值为true，那么后续的字符串就会被使用。

Render:function(){

Return <div className={this.state.isComplete&&’is-complete’}></div>

…

</div>

}

### 非DOM属性

下面的特殊属性只在JSX中存在：

A. key

B. ref

C. dangerouslySetInnerHTML

下面我们将讨论更多的细节。

###### 键（key）

Key是一个可选的唯一标识符。在程序运行的过程中，一个组件可能会在组件树中调整位置，比如当用户进行搜索操作时，或者当一个列表中的物品被增加、删除时。当这些情况发生时，组件可能并不需要被销毁并重新创建。

通过更智能地决定应该重用一个组件，还是销毁并重新创建一个组件，进而提升渲染性能。当两个已经存在于DOM中的组件交换位置时，React能够匹配对应的键并进行相应的移动，且不需要重新渲染DOM。