# 第8章 DOM操作

多数情况下，React的虚拟DOM足以用来创建你想要的用户体验，而根本不需要直接操作底层真实的DOM。通过将组件组合到一起，你可以把复杂的交互聚合为呈现给用户的连贯整体。

然而，在某些情况下，为了实现某些需求就不得不去操作底层的DOM。最常见的场景包括：需要与一个没有使用React的第三方类库进行整合，或者执行一个React没有原生支持的操作。

为了使这些操作变得容易，React提供了一个可用于处理其自身控制的DOM节点的方法。这些方法仅在生命周期的特定阶段才能被访问到。不过，使用它们足以应付上述场景。

## 访问受控的DOM节点

想要访问受React控制的DOM节点，首先必须能够访问到负责控制这些DOM的组件。这可以通过为子组件添加一个ref属性来实现。

var DoodleArea=React.createClass({

render:function(){

return <canvas ref=”mainCanvas” />;

}

});

这样，你就可以通过this.refs.mainCanvas访问到<canvas>组件。如你所想，你必须保证赋给每个子组件的ref值在所有子组件中是唯一的；如果你为另一个子组件的ref也赋值为mainCanvas，那么操作就会失效。

一旦访问到了我们刚刚讨论的子组件，就可以通过它的getDOMNode()方法访问到底层的DOM节点。然而请不要试图在render方法中这样做。因为，在render方法完成并且React执行更新之前，底层的DOM节点可能不是最新的（甚至尚未创建）。

同样，直到组件被挂载你才能去调用getDOMNode()方法——此时，componentDidMount事件处理器将会被触发。

var DoodleArea=React.createClass({

render:function(){

// render方法调用时，组件还未挂载，所以这将引起异常！

this.getDOMNode();

return <canvas ref=”mainCanvas” />

},

componentDidMount:function(){

var canvasNode=this.refs.mainCanvas.getDOMDode()

//这里是有效的！我们现在可以访问到HTML5 Canvas节点，并且可以在它上面随//意调用painting方法。

}

});

注意，componentDidMount内部并不是getDOMNode方法的唯一执行环境。事件处理器也可以在组件挂载后触发，所以你可以在事件处理器中调用getDOMNode，就像在componentDidMount方法中一样简单。

var RichText=React.createClass({

render:function(){

return <div ref=”editableDiv” contentEditable=”true” onKeyDown={this.handleKeyDown} ></div>;

},

handleKeyDown:function(){

var editor=this.refs.editableDiv.getDOMNode();

html=editor.innerHTML;

//现在我们可以存储用户已经输入的HTML内容！

}

});

上面的例子创建了一个带有contentEditable属性（值为true）的div，允许用户在其内部输入富文本。

尽管React本身并没有提供访问原生HTML内容的方法，但是keyDown处理器可以访问到div对应的底层DOM节点。换而言之，它可以访问原生的HTML。在此处，你可以保存用户已经输入内容的一份拷贝，计算并展示出文字的个数，等等。

请记住，尽管refs和getDOMNode很强大，但请在没有其他方式能够实现你需要的功能时再去选择它们。使用它们会成为React在性能优化上的障碍，并且会增加应用的复杂性。所以，只有当常规的技术无法完成所需要的功能时，才应该考虑它们。

## 整合非React类库

有很多好用的JavaScript类库并没有使用React构建。一些类库不需要访问DOM（比如日期和时间操作库），但如果需要使用它们，保持它们的状态和React的状态之间的同步是成功整合的关键。

假如你需要使用一个autocomplete类库，包括了下面的示例代码：

autocomplete({

target:document.getElementById(“cities”),

data:[

“San Francisco”,

“St. Louis”,

“Amsterdam”,

“Los Angeles”

],

events:{

select:function(city){

alert(“You have selected the city of ”+city);

}

}

});

这个autoComplete函数需要一个目标DOM节点、一个用作数据展现的字符串清单，以及一些事件监听器。为了兼得React和该类库的优势，我们从创建一个使用了这两个库的React组件开始。

Var AutocompleteCities=React.createClass({

Render:function(){

Return <div id=”cities” ref=”autocompleteTarget” />;

},

getDefaultProps:function(){

return {

data:[

“San Francisco”,

“St. Louis”,

“Amsterdam”,

“Los Angeles”

]

};

},

handleSelect:function(city){

alert(“You have select the city of “+city);

}

});

为了将该类库封装到React中，需要添加一个componentDidMount处理器。它可以通过autocompleteTarget所指向子组件的底层DOM节点来连接这两个接口。

Var AutocompleteCities=React.createClass({

Render:function(){

Return <div id=”cities” ref=”autocompleteTarget” />;

},

getDefaultProps:function(){

return {

data:[

“San Francisco”,

“St. Louis”,

“Amsterdam”,

“Los Angeles”

]

};

},

handleSelect:function(city){

alert(“You have select the city of “+city);

},

componentDidMount:function(){

autocomplete({

target:this.refs.autocompleteTarget.getDOMNode(),

data:this.props.data,

events:{

select:this.handleSelect

}

})

}

});

注意，componentDidMount方法只会为每个DOM节点调用一次。因此我们不用担心，在同一个节点上两次调用autocomplete方法（这个示例中）是否会有副作用。

也就是说，需要记住该组件可能被移除，然后在其他的DOM节点上重新渲染，如果在componentDidMount方法内导致了DOM节点无法被移除，有可能导致内存泄漏或者其他的问题。如果你担心这一点，请确保指定一个componentWillUnmount监听器，用于在组件的DOM节点移除时清理它自身。