react 简介

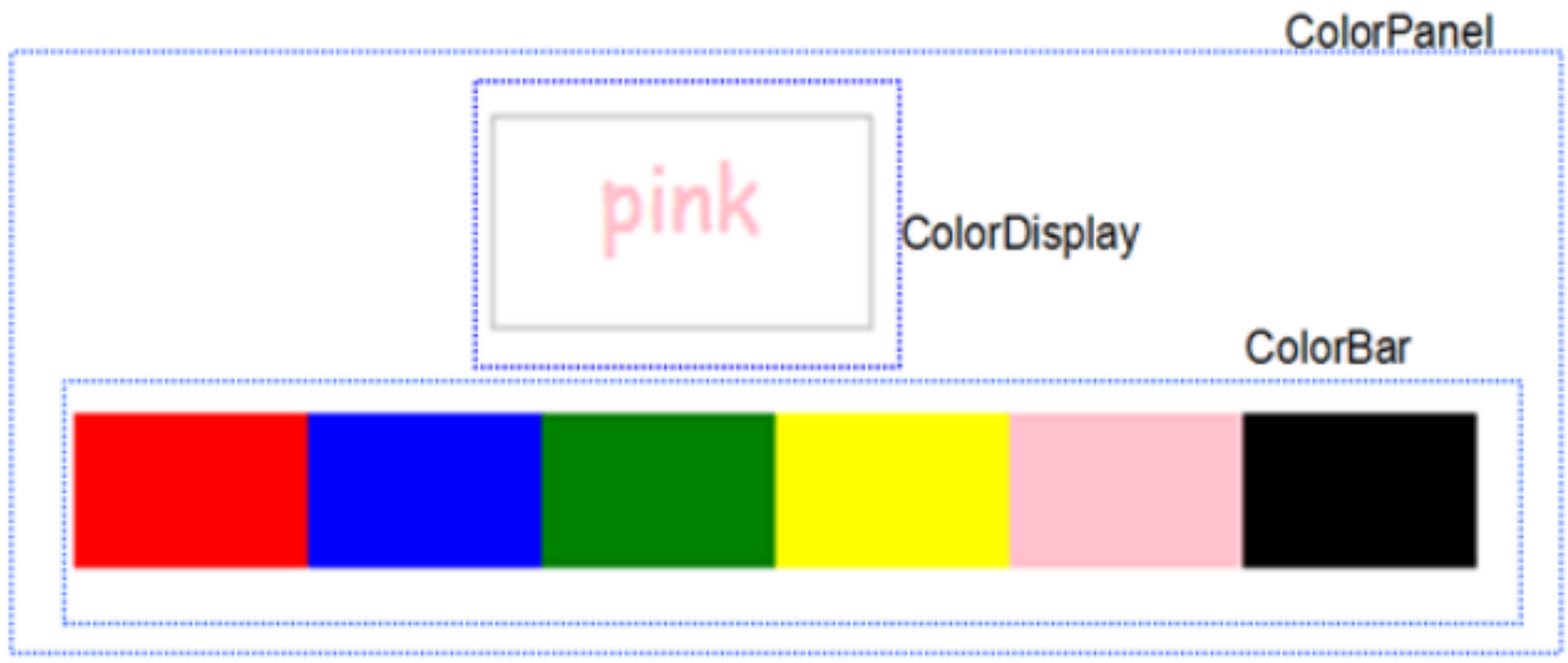
引言

React 是近期非常热门的一个前端开发框架， 其本身作为 MVC 中的 View 层可以用来构建 UI，也可以以插件的形式应用到 Web 应用非 UI 部分的构建中， 轻松实现与其他 JS 框架的整合， 比如 AngularJS 。同时，React 通过对虚拟 DOM 中的微操作来实对现实际 DOM 的局部更新，提高性能。其组件的模块化开发提高了代码的可维护性。单向数据流的特点，让每个模块根据数据量自动更新，让开发者可以只专注于数据部分，改善程序的可预测性。

React 概况介绍

React 中最基础最重要的就是 Component 了，它的构造和功能相当于 AngularJS 里面的 Directive ，或是其他 JS 框架里面的 Widgets 或 Modules 。Component 可以认为是由 HTML 、CSS、JS 和一些内部数据组合而成的模块。当然 Component 也可以由很多 Component 组建而成。不同的 Component 既可以用纯 JavaScript 定义，也可以用特有的 JavaScript 语法 JSX 创建而成。关于 JSX，我们会在后面加以详细介绍。为了能让您更好的理解 Component 概念，让我们来看一下本文的实例。

图 1. 实例展示



本文的实例是一个颜色盘，当用户把鼠标移动到底部的颜色条上时，上方方框内会提示对应的颜色。如果用 React 来实现这个实例， 这个颜色盘会被分割成为多个 Component 。最外面一层父级 Component 称为 ColorPanel ，同时它包含了两个子 Component ：上方的 ColorDisplay 和下方的 ColorBar 。我们在 ColorPanel 的属性里面赋值了该颜色盘所要显示的所有颜色， ColorBar 获取这个颜色属性渲染出所有的颜色， 当鼠标移动到 ColorBar 的某个颜色上时， ColorDisplay 显示对应的颜色和文字。在这个过程中，我们并不需要去手动触发所有组件的渲染，只需要在程序开始时渲染 ColorPanel 。当 ColorPanel 的当前选中颜色这一状态发生变化时，子 Component （比如 ColorDisplay ）就会根据这一状态自动重新渲染。这正是 React 的奇妙所在——用户更多的去关心数据的变化，而无需在意 DOM 节点的渲染。

创建第一个 Component

通过上面上面一段的介绍， 相信您已经对于 React 有了一定的概念， 下面就让我们开始着手自己开发一个 React 的小

实例吧！首先第一步就是创建一个简单的 Component 。

清单 1. 创建 ColorPanel Component

```
var ColorPanel = React.createClass({
render: function() {
return (
<div>
 Color Panel
</div>
)
}
});
React.render(<ColorPanel/>, document.getElementById('demo'));
```

清单 1 为创建 ColorPanel 这个 Component 的代码。通过 createClass 方法来创建了一个 Component 。其中，createClass 需要传入一个 object 对象，这个对象可以定义不同的属性， 比如 getInitialState 、getDefaultProps 、render 等等。 此处值得一提的是， 与其它属性不同， render 方法是必须存在的， 因为 render 方法的返回值代表的是 Component 的 template 。 清单 1 的例子即是创建一个包含着 Color Panel 字样的 Component 。

当然，createClass 的功能是用来创建 Component 对象，如果要渲染这个 Component 到显示节点上， 就需要调用 React.render 方法。在这个方法中包含两个参数，第一个参数需要指定需要渲染的 Component 对象，第二个参数表示该 Component 所挂载到父节点。

看到这里，您可能会觉得疑惑：为什么 JavaScript 代码中嵌入了 HTML 标签？其实这段嵌套在在 render 方法里面的并非真正意义上的 HTML， React 称之为“ JSX ”JSX 可以允许把 HTML 类似的语法转换为轻量级的 JavaScript 对象。

清单 2. 把 JSX 转化成 JavaScript 的方式来创建 Component

```
var ColorPanel = React.createClass({
displayName: "ColorPanel",
render: function() {
return React.createElement("div", null, "Color Panel");
}
});
React.render(<ColorPanel/>, document.getElementById('demo'));
```

在清单 2 的代码为 JSX 转化为 JavaScript 的表达方式。然而，在实际使用 React 开发中， 选着使用 JSX 的语法方式可以使代码结构更加清晰简洁。使用 JSX，开发人员还可以跳出 HTML 节点的限制自定义 Component 。

开发人员只需要在 HTML 页面中简单的添加清单 3 中的 JS 文件—— React 提供的解析转化脚本，就可以轻松实现 JSX 到 JavaScript 的转化。

清单 3. 在 HTML 中引用 JSX 转化脚本

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

介绍完 JSX，此处需要引入另一个重要概念——虚拟 DOM 。React 引入这种概念是因为虚拟 DOM 把当前真实的 DOM 节点转化成 JavaScript 对象。这使得 React 可以追踪当前虚拟 DOM(某些数据变化后生成)和之前的虚拟 DOM(某

些数据变化之前)的差异。 React 通过两者对比提取出来这些差异点，然后再在真实的 DOM 节点上执行必要的更新。在传统的开发中，通常 UI 上诸多状态变化会让维护应用的状态变得很困难和复杂。 React 通过检测状态变化来每次重新渲染虚拟 DOM 节点，然后按需自动更新真实节点，这种方式可以让开发人员可以简单地专注在应用的状态上。

总结一下上述操作的过程是： 首先某一信号触发应用中某些数据发生改变， React 就会重新渲染整个虚拟 DOM 。然后 React 会比较现在的虚拟 DOM 与之前的虚拟 DOM 的差异，获知哪些是需要更新的，最后在真实的 DOM 上应用这些必要的更新。更具体的虚拟 DOM 技术，会在下面章节中详细介绍。

给 Component 添加 state

上文中我们提到 UI 界面元素的诸多状态使得维护 UI 页面变的十分困难， React 却让着一切变的轻松起来。 因为在 React 中，每一个 Component 都会维护自己的 state ，当子 Component 需要这些 state 作为 props 时，则将其顺序传递下去。换句话说， 如果有多个层级的 Component ，它们公共的父级 Component 负责管理 state ，然后通过 props 把 state 传递到子 Component 中。

清单 4. 为 Component 添加 state

```
var ColorPanel = React.createClass({
getInitialState: function() {
return {
selectedColor: 'red'
}
},
render: function() {
return (
<div>
{this.state.selectedColor}
</div>
)
}
});
React.render(<ColorPanel/>, document.getElementById('demo'));
```

清单 4 是一个 Component 使用内部 state 的一个例子。您会注意在这里创建组件时添加了一个 getInitialState 方法，它是用来设置 Component 的 state 的，它返回的是一个对象包含了 Component 的 data 或者 state 值。在这个例子中，也是通过这个方法告诉 Component 需要保存一个叫 selectedColor 的对象，在 Component 的 render 方法中就可以通过 {this.state.selectedColor} 来使用了。

当然， Component 也可以来更改这个内部状态值，这要通过 setState 方法。之前说的 “ 某一信号触发应用中某些数据发生改变 ” 指的就是 setState 方法。不管 setState 方法何时调用，虚拟 DOM 都会被重新渲染，之后运行差异算法并按需更新真实的 DOM 。

从父 Component 中获取 State

在前面的介绍也提到了不可或缺的 props ，它指的就是从父 Component 传递到子 Component 的数据。通过

props ， React 框架可以保持良好的数据的直线传递——在最顶层的父级 Component 中处理所需要使用的特殊数据，当子的 Component 也需要使用时就把它们通过 props 来传递下去。 事实上 props 对于 Component 就像 Attribute 对于 HTML 一样，当我们提供了 property 的 name 和 value 时就传递到 Component 里面了。在 Component 里面通过 this.props 来获取 Properties 。清单 5 中展示了从父级节点获取 state 的实例。

清单 5. 从父节点中获取 state

```
var ColorPanel = React.createClass({
render: function() {
return (
<div>
<ColorBar colors={this.props.colors} />
</div>
);
},
});
var colors = [
{id: 1, value: 'red', title: 'red'},
{id: 2, value: 'blue', title: 'blue'},
{id: 3, value: 'green', title: 'green'},
{id: 4, value: 'yellow', title: 'yellow'},
{id: 5, value: 'pink', title: 'pink'},
{id: 6, value: 'black', title: 'black'}
];
React.render(<ColorPanel colors={colors}/>, document.getElementById('demo'));
```

我们可以通过改变 render 函数的内容来改变组件的行为。 但是如果想要根据外部的信息来改变组件的行为， 就需要使用 Properties 。当然，我们也可以通过 getDefaultProps() 来返回 property 的初始值，该方法的返回值必须是一个对象。任何通过 getDefaultProps() 返回的对象都会被所用实例共享。清单 6 中展示了通过 getDefaultProps 方法获取 props 初始值的实例。

清单 6. 通过 getDefaultProps 获取 props 初始值

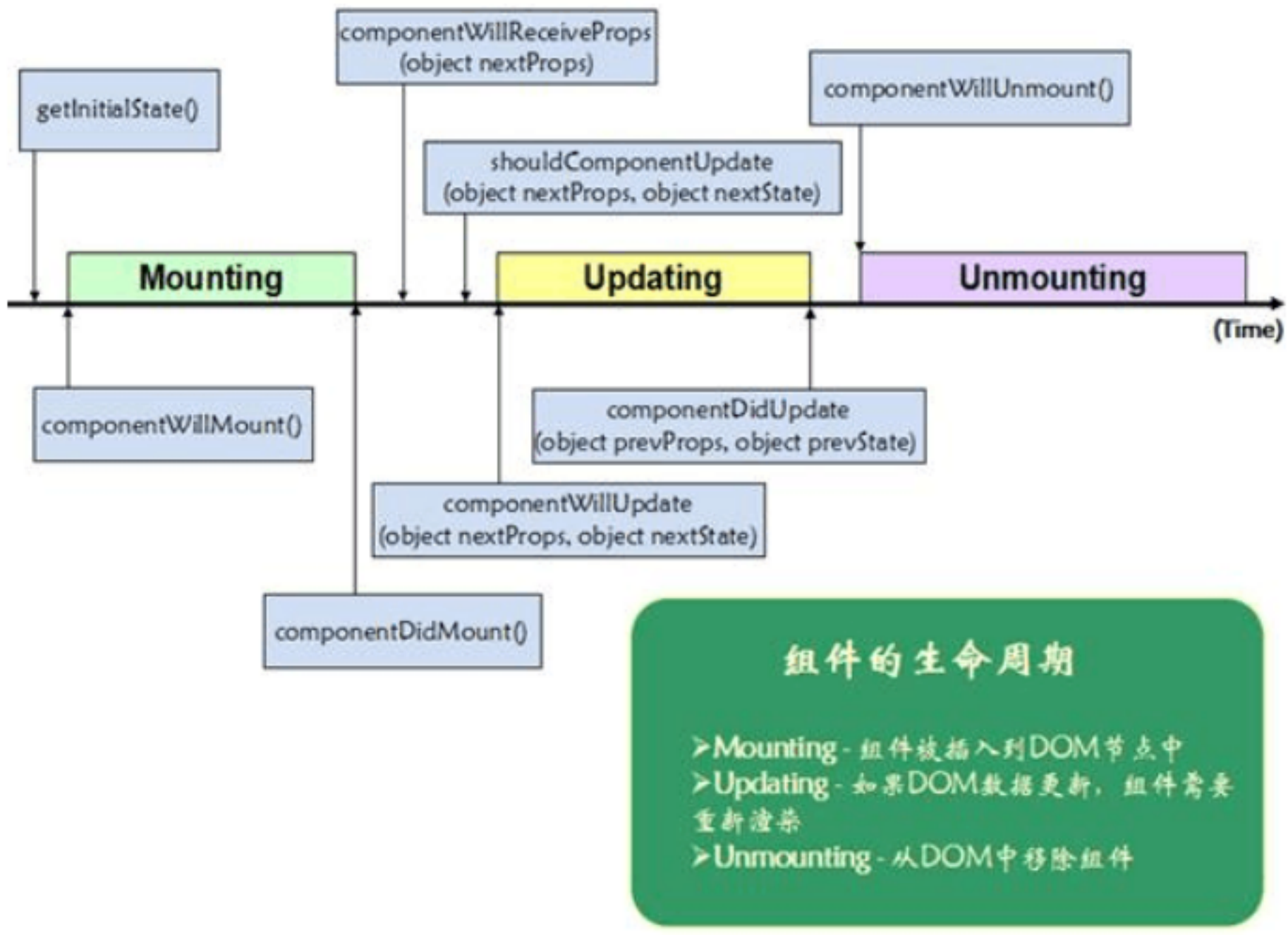
```
var ColorPanel = React.createClass({
getDefaultProps: function() {
return {
colors: [
{id: 1, value: 'red', title: 'red'},
{id: 2, value: 'blue', title: 'blue'},
{id: 3, value: 'green', title: 'green'},
{id: 4, value: 'yellow', title: 'yellow'},
{id: 5, value: 'pink', title: 'pink'},
{id: 6, value: 'black', title: 'black'}
]
```

```
}
},
render: function() {
return (
<div>
<ColorBar colors={this.props.colors} />
</div>
);
},
});
React.render(<ColorPanel/>, document.getElementById('demo'));
```

Component 的生命周期

每个 Component 都有自己的生命周期，在此期间 React 提供了很多方法用于对不同阶段的组件加以操作。下图 4 介绍了一个 Component 在其生命中期中可以执行的方法。

图 2. Component 的生命周期



组件的生命周期主要可以分为三个阶段， Mounting 、Updating 、Unmounting 。React 在每一阶段开始前提供了 will 方法用于执行恰好在这一阶段开始前需要实行的操作， 为每一段结束之后提供了 did 方法用于执行恰好这一阶段结束时需要

实现的操作。下面我们详细说明一下每一个阶段的具体实现。

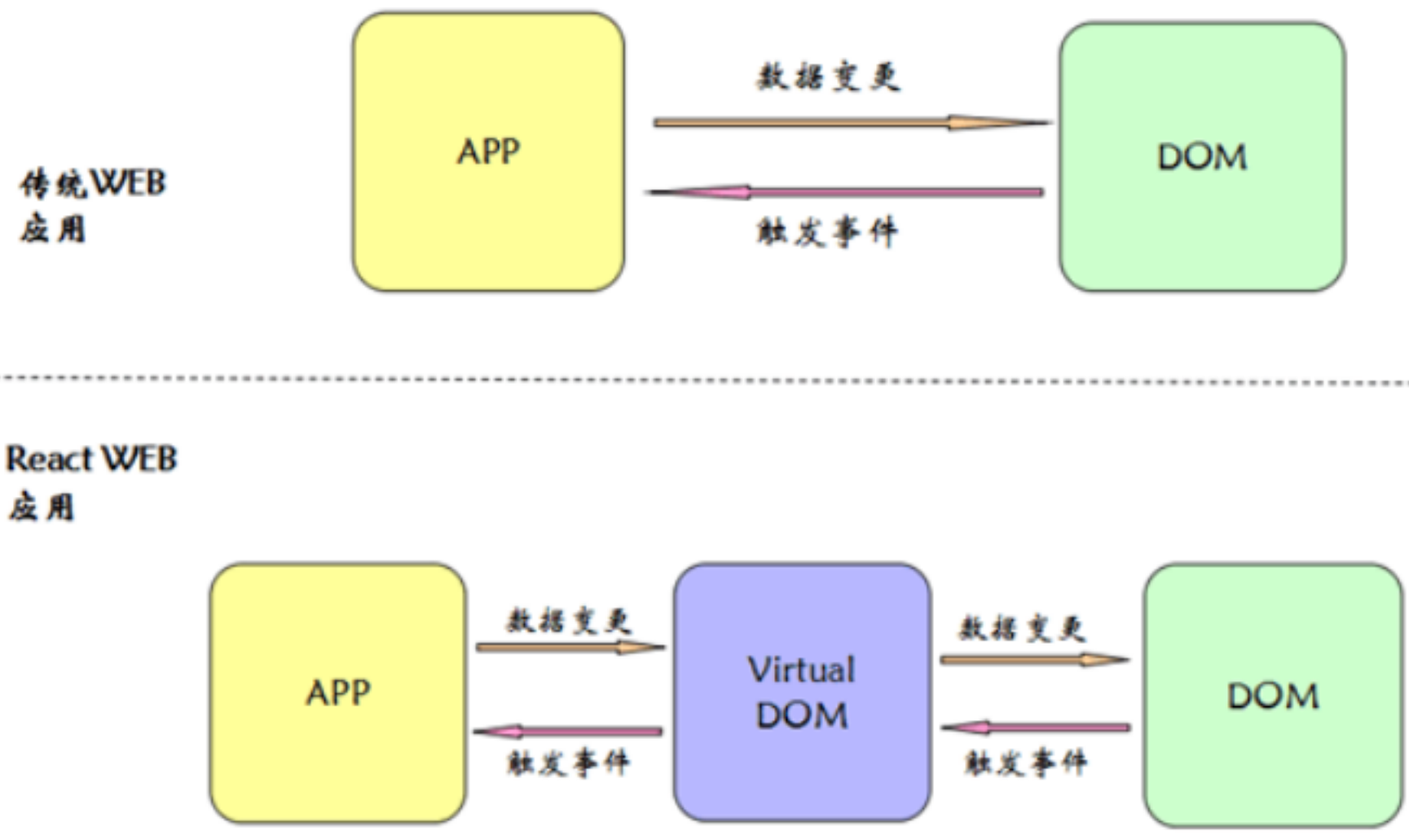
首先在 Mounting 阶段，Component 通过 React.createClass 被创建出来，然后调用 getInitialState 方法初始化 this.state 。在 Component 执行 render 方法之前，通过调用 componentWillMount （方法修改 state 状态），然后执行 render 。Reder 的过程即是组件生成 HTML 结构的过程。在 render 之后，Component 会调用 componentDidMount 方法。在这个方法执行之后，开发人员才能通过 this.getDOMNode() 获取到组件的 DOM 节点。当 Component 在 mount 结束之后，它当中有任何数据修改导致的更新都会在 Updating 阶段执行。Component 的 componentWillReceiveProps 方法会监听组件中 props 。监听到 props 发生修改，它会比对新的数据与之前的数据之间是否存在差别进而修改 state 的值。当比对的结果为数据变化需要对 Component 对应的 DOM 节点做出修改的时候，shouldCoponentUpdate 方法它会返回 true 用于触发 componentWillUpdate 和 componentDidUpdate 方法。在默认的情况下 shouldComponentUpdate 返回为 true 。有些特殊的情况是当 component 中的 props 发生修改，但是其本身数据并没有改变，或者是开发人员手工设置 shouldComponentUpdate 为 false 时，React 就不会更新这个 component 对应的 DOM 节点了。与 componentWillMount 和 componentDidMount 相类似，componentWillUpdate 和 componentDidUpdate 也分别在组件更新的 render 过程前后执行。

当开发人员需要将 component 从 DOM 中移除时，就会触发 UnMounting 阶段。在这个阶段中，React 只提供了一个 componentWillUnmount 方法在卸载和销毁这个 component 之前触发，用于删除 component 中的 DOM 元素等。

虚拟 DOM

在传统的 Web 应用中，我们往往会把数据的变化实时地更新到用户界面中，于是每次数据的微小变动都会引起 DOM 树的重新渲染。如果当前 DOM 结构较为复杂，频繁的操作很可能会引发性能问题。React 为了解决这个问题，引入了虚拟 DOM 技术。图 2 为传统 Web 应用于 React Web 应用的对比图。

图 3. 传统 Web 应用于使用 React 的 Web 应用对比

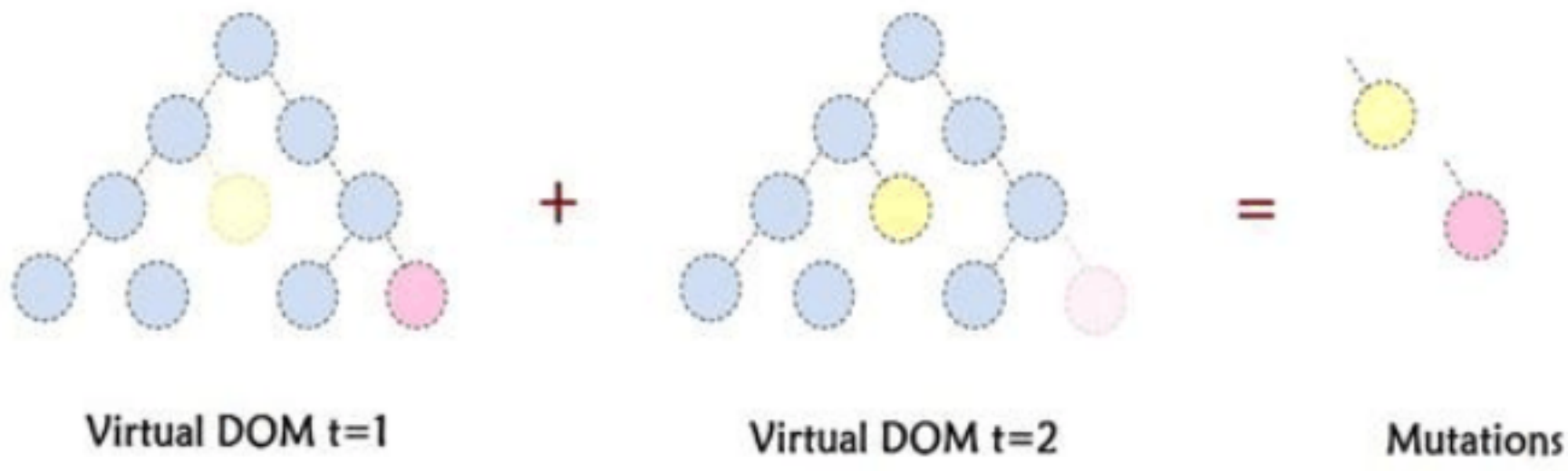


虚拟 DOM 是一个 JavaScript 的树形结构，包含了 React 元素和模块。组件的 DOM 结构就是映射到对应的虚拟 DOM

上，React 通过渲染虚拟 DOM 到浏览器，使得用户界面得以显示。与此同时，React 在虚拟的 DOM 上实现了一个 diff 算法，当要更新组件的时候，会通过 diff 寻找到要变更的 DOM 节点，再把这个修改更新到浏览器实际的 DOM 节点上，所以在 React 中，当页面发生变化时实际上不是真的渲染整个 DOM 树。

React 虚拟 DOM 中的诸多如 div 一类的标签与实际 DOM 中的 div 是相互独立的两个概念，它是一个纯粹的 JS 数据结构，它只是提供了一个与 DOM 类似的 Tag 和 API。React 会通过自身的逻辑和算法，转化为真正的 DOM 节点。也正是因为这样的结构，虚拟 DOM 的性能要比原生 DOM 快很多。图 3 模拟了虚拟 DOM 数据更新的场景：

图 4. 虚拟 DOM 中数据的更新过程

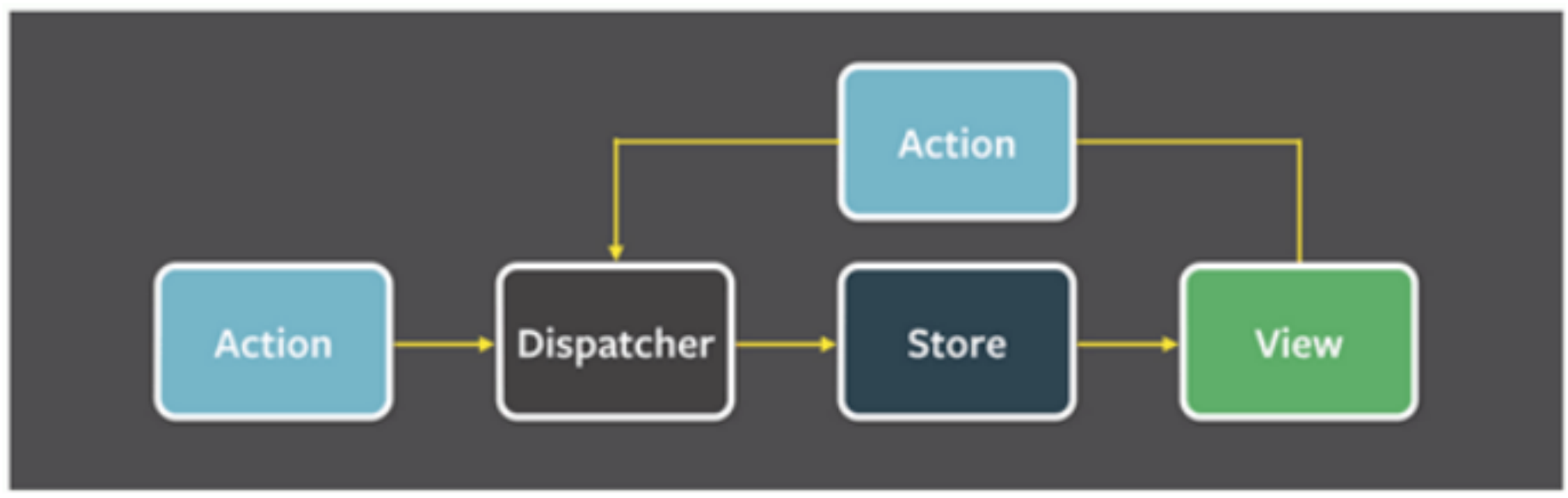


在当前页面中数据发生变化时，React 会重新构建其 DOM 树，也就是我们所说的虚拟 DOM 。然后 React 会将这个新构建好的虚拟 DOM 树与更新之前的虚拟 DOM 树加以比对得出结构差异： 增加一个黄色节点， 删除一个粉红色节点。通过这样的对比，使得浏览器实际的更新过程中，可以只修改变更部分——黄色节点和粉红色节点，而其它节点保持不变。

Flux

Flux 其实就是一种单向数据流的模式。与常见的 View 和 Store 相互交互的 MVC 模式不同的是， Flux 引入了 Dispatcher 。用户在 View 上的操作，不会直接引起 Store 的变化，而是通过 Action 触发在 Dispatcher 上注册的回调函数，从而触发 Store 数据的更新， 最终组件会重新渲染。 这样一来，即可以保证数据流是单向流通的。 Flux 相对于 MVC ，让事情变得更加可以预见。

图 5.Flux 的数据流向图



当用户和 View （Component ）交互的时候， View 会触发一个 Action 到中央 Dispatcher 。然后将 Action 再分配

到保存着应用的数据和业务逻辑的 Store 里面，Store 内数据的更新会引起所有 View 的更新。这个对 React 这种申明式的语法非常适用，这样就允许 Store 在更新的时候，不用去关注不同的 View 和 State 是如何交互的。

Dispatcher 类似于一个中央枢纽，管理着所有的数据流。它本身没有什么业务逻辑，只是负责把不同的 Action 分发到不同的 Store 中。其逻辑是 Store 会在 Dispatcher 中按照不同的 actiontype 注册不同的回调函数。当 Action 到达 Dispatcher 的时候，Dispatcher 会根据 Action 的 type 找到之前注册 Store，并触发其回调函数，从而更新 Store 的数据，达到 View 的更新。

Store 存储着应用的 state 和逻辑。它们的角色类似于传统 MVC 里面的 model，但是它们可以管理许多对象的状态，它们不只是代表了一个 ORM 模型或者说像 Backbone 里面的 collection。它们更多的是管理着应用的一个特定领域状态。就像之前提到的，Store 会注册自己和对应的回调函数到 Dispatcher 上。回调函数会接收 Action 作为参数。不同的 actiontype 会对应不同的业务逻辑，从而更新不同的 state。当 Store 被更新后会广播告知 View 去更新 Component。Views 和 controller-views：React 提供了可组建的、会自动重新渲染的 View。在嵌套结构的最上层，有一种特殊类型的 View 监听这着 Store 的广播，我们称之为 controller-view，它会获取 Store 里面的数据，然后会调用 setState 方法，从而触发该 Component 的 render 方法和子 Component 的 render 方法。

我们通常会把 Store 的所有的 state 放在一个对象里面沿着链式地 View 传递下去，这样允许不同的子孙 Component 都能获取的它们所需。并且把这种 controller-view 放在嵌套结构的最顶层，是为了保持子孙 Component 逻辑更简单，而且也可以减少需要管理的 props 的数量。

Dispatcher 暴露了一个方法，允许我们来触发 Action 到 Store 的转发，而且可以把 Action 作为参数传递过去。Action 除了来自用户和 View 的交互，也可以来自 server 端。

实例分析

通过对于上一个部分的了解，相信测试此时的您心中已经有了 React 的基本轮廓。下面让我来看一下完整的 ColorPanel 实例的具体实现吧！首先，在 index.html 里面引入三个 JSX 文件。这三个文件就是我们使用 React 实现 ColorPanel 的源代码。

清单 7. HTML 文件中引用的 JSX 文件

```
<script type="text/jsx" src="js/colorBar.jsx"></script>
<script type="text/jsx" src="js/colorDisplay.jsx"></script>
<script type="text/jsx" src="js/colorPanel.jsx"></script>
```

清单 8. colorPanel.jsx 代码

```
var ColorPanel = React.createClass({
getDefaultProps: function() {
return {
 colors: [
{id: 1, value: 'red', title: 'red'},
{id: 2, value: 'blue', title: 'blue'},
{id: 3, value: 'green', title: 'green'},
{id: 4, value: 'yellow', title: 'yellow'},
{id: 5, value: 'pink', title: 'pink'},
{id: 6, value: 'black', title: 'black'}
```

```
 ],
 defaultColorId: 1
 }
},
getInitialState: function() {
return {
 selectedColor: this.getSelectedColor(this.props.defaultColorId)
 }
},
getSelectedColor: function(colorId) {
if(!colorId)
return null;
var length = this.props.colors.length;
for(var i = 0; i< length; i++) {
if(this.props.colors[i].id === colorId)
   break;
 }
 return this.props.colors[i];
},
shouldComponentUpdate: function(nextProps, nextState) {
   return this.state.selectedColor.id !== nextState.selectedColor.id;
},
render: function() {
console.log('Render Color Panel');
return (
 <div>
 <ColorDisplay selectedColor={this.state.selectedColor}/>
<ColorBar colors={this.props.colors} onColorHover={this.onColorHover} />
 </div>
);
},
onColorHover: function(colorId) {
this.setState({selectedColor: this.getSelectedColor(colorId)});
}
});
React.render(<ColorPanel/>, document.getElementById('demo'));
```

清单 9. colorBar.jsx 代码

```
var ColorBar = React.createClass({
shouldComponentUpdate: function(nextProps, nextState) {
   return false;
},
```

```
render: function() {
 console.log('Render Color Bar Component');
 return (
 <ul>
 {this.props.colors.map(function(color){
   return (
 <li key={color.id}
onMouseOver={this.props.onColorHover.bind(null, color.id)}
className={color.value}></li>
   )
 }, this)}
 </ul>
);
}
});
```

清单 10. colorDisplay.jsx 代码

```
var ColorDisplay = React.createClass({
shouldComponentUpdate: function(nextProps, nextState) {
   return this.props.selectedColor.id !== nextProps.selectedColor.id;
},
render: function() {
 console.log('Render Color Display Component');
return (
<div className="color-display">
<div className={this.props.selectedColor.value}>
{this.props.selectedColor.title}
</div>
</div>
);
}
});
```

这个实例的实现原理是： 当用户把鼠标停留在某一个 colorbar 上时，就会触发 mouseover 上绑定的 onColorHover 事件，同时传递了当前颜色的 ID 。然后，onColorHover 事件根据传递进来的颜色 ID 重置 selectedColor 。selectedColor 作为 ColorDisplay 这个 Component 的 state 值，当它发生变化时引发的虚拟 DOM 节点的重新 render 进而引起了真实 DOM 节点的重新渲染。于是页面上所显示的文字内容和颜色类型转换成为我们所选择的对应值。

在实际的代码逻辑中， Color Bar 的颜色为六种固定颜色，于是我们把这些颜色的值（ value ）的定义存放在父级 Component 的 props 中，通过 getDefaultProps 来设置默认的属性以及初始时所选着的颜色（此处选择默认 ID 为 1 的颜色）。接下来在 ColorPanel 中设置一个 selectedColor 作为内部状态值 state 表示当前所选中的颜色。它的定义和初始化是在 getInitialState 中实现的，而修改则是在 getSelectedColor 方法中完成的（根据选中的 color ID 来获取 color 对象）。在这里，所有的 Color 都是 Component 的 props ，可以通过 this.props.colors 获取所有颜色值。

接下来是 ColorBar 。在 ColorBar 里面使用 this.props.colors.map 来遍历生成了多个 li。它的原理跟 Array.prototype.map 类似， this.props.colors 其实是一个 array ，包含了所有的设置的颜色。 Map 方法，就是把 array 里面的 item 取出来传递到回调函数中。回调函数会把传进来的 color 值填充到颜色条的模板中。在这个模板的 li 上也绑定了 mouseover 事件，提前把对应的颜色绑定在了 onColorHover 这个属性方法中， 当回调函数返回了 li 形成一个新的 array 填充到 Color Bar 的 ul 中。

最后让我们来看一下 ColorDisplay ，它是一个负责显示当前选中颜色的提示框。父级 Component 中的 selectedColor 通过一个 props 传递到子级 Component 中并通过 this.props.selectedColor 获取。 然后子级 Component 通过所获取到的 selectedColor 值来确定颜色类型（ class ）决定所需要显示的文字。

那么颜色提示是何时更新的呢？答案是当父级 Component 的 state selectedColor 发生更新，也即 setState 方法被调用时。当用户把鼠标放在某个颜色块时，就触发 onColorHover 这个由父节点传进来的 props 方法，在这个方法内我们会调用 setState 方法，根据传进来的 color id 值重置这个 selectedColor ，从而更新 color Display 的颜色提示。 React 默认只要 setState 方法的调用，就会更新这个 Component 。虽然是虚拟的 Component 的更新，但是有时候我们可以手动的去决定是否有必要更新这个 Component ，这时候就可以采用 Component 生命周期里面的 shouldComponentUpdate 方法。比如说在文章实例中，当用户先把鼠标从某一个颜色条上移除，然后又移回到之前的颜色条上时。此时用户所选择的还是同一种颜色，即 selectedColor 没有变化时，我们可以使用 shouldComponentUpdate 来避免 setState 引起的二次更新。判断当前的 selectedColor.id 和之前的 selectedColor.id 值的区别，此时两者没有差异则返回 false ， Color Panel 不会被重新渲染了。同理， Color Bar 其实是一个固定的 Component ，完全不需要每次根据 selectedColor 来重新渲染，所以我们可以在这个 component 的 shouldComponentUpdate 直接返回 false 。

结语

React 的问世把前段开发人员从复杂的数据交互和 UI 渲染不可预测的问题中解放出来， 让开发人员可以将精力专注于数据的变化上。于此同时， React 运用虚拟 DOM 技术，极大的提升了页面的性能。而且，它通过对模块的划分，让应用的逻辑和数据流更加清晰明了。 相信本文通过对于实例的演示， 您可以发现 React 代码良好的可读性和易用性， 让 React 的学习和使用变动轻松有趣。