1. React组件的数据

React组件的数据分为两种，prop和state,无论prop或者state的改变，都可能引发组件的重新渲染，prop是组件的对外接口，state是组件的内部状态，对外使用prop，内部使用state。属性使用的演示例子，demo1. 在counter组件中，使用了caption和initValue两个prop。通过名为caption的prop，ControlPanel传递给Counter组件实例说明文字。通过名为initValue的prop传递给Counter组件一个初始的计数值。

<1.11>给prop赋值

Eg:<SampleButton id=”sample” borderWidth={2} onClick={onButtonClick} style={{color:”red”}}/>

<1.12>读取Prop值

Counter组件内部接收传入的prop,首先是构造函数，代码如下：

Class Counter extends Component{

Constructor(props){

Super(props);

This.onClickIncrementButton = this.onClickIncrementButton.bind(this);

This.onClickDecrementButton = this.onClickDecrementButton.bind(this);

This.state = {

Count:props.initValue || 0

}

}

除了在构造函数中使用上面的方法获取prop值之外，在其他的函数中可以使用this.props获取属性值。Eg:在Counter组件的render函数中，我们就是通过这种方法获取传入的caption,代码如下：

Render(){

Const {caption} = this.props;

Return(

<div>

<button style={buttonStyle} onClick={this.onClickIncrementButton}>+</button>

<button style={buttonStyle} onClick={this.onClickDecrementButton}>-</button>

<span>{caption} count:{this.state.count}</span>

</div>

);

}

<1.21>React的state

驱动组件渲染的过程除了prop,还有state，state代表组件的内部状态。由于React组件不能修改传入的prop,所以需要记录自身数据变化时，就要使用state.

在Counter组件中，最初显示初始计数，可以通过initValue这个prop来定制，在Counter已经显示出来之后，用户点击“+”和“-”按钮改变这个计数，这个变化的数据就要Counter组件自己通过state来存储了。

<1.22>读取和更新state

通过给button的onClick属性挂载点击事件处理函数，我们可以改编组件的state，以点击”+”按钮的响应函数为例，代码如下：

onClickIncrementButton(){

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

}

<1.3> prop和state的对比

<1>prop用于定义外部接口，state用于记录内部状态

<2>prop的赋值在外部世界使用组件时，state的赋值在组件内部

<3>组件不应该改变prop的值，而state存在的目的就是让组件来改变的。组建的state，就是相当于组件的记忆，其存在的意义就是被修改，每一次通过this.setState函数修改state就改变了组件的状态，然后通过渲染过程把这种变化体现出来。但是，组件绝不允许去修改传入的props的值，否则会带来混乱。

1. 组件向外传递数据的实现demo2

<2.1>在上面的例子中我们的ControlPanel组件中，包含三个Control子组件实例，每个Counter都有一个可以动态改变的数值，我们希望ControlPanel能够及时显示出这三个组件当前的计数值之和，这个功能，我们需要解决的问题是让ControlPanel“知道”三个子组件当前的计数值，而且每次变更都要立刻知道，解决这个问题的方法依然是利用prop.组建的prop可以是任何JavaScript对象，而在JavaScript中，函数是一等公民，函数本身就是一种对象，既可以像其他对象一样作为prop的值从父组件传递给子组件，又可以被子组件作为函数调用。实例参见demo2.

功能实现的关键代码如下：

在Counter组件中，对于点击”+”和”-”按钮的事件处理代码如下：

onClickIncrementButton(){

this.updateCount(true);

}

onClickDecrementButton(){

this.updateCount(false);

}

updateCount(isIncrement){

Const previousValue = this.state.count;

Const newValue = isIcrement?previousValue+1:previousValue-1;

this.setState({count: newValue})

this.props.onUpdate(newValue,previousValue)

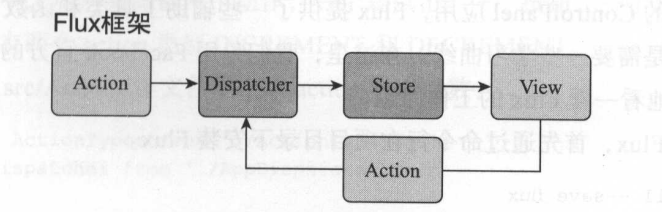
}

新增加的prop叫做onUpdate，类型是一个函数，当Counter的状态改变的时候，就会调用这个给定的函数，从而达到通知父组件的作用。

1. React组件的state和prop的局限

通过上面的demo2我们不难发现实现的并不精妙，每个Counter组件有自己的状态记录当前计数，而父组件ControlPanel也有一个状态存储所有Counter计数总和，也就是说数据发生了重复，数据重复就会带来如何保证数据一致性的问题。例如上面例子中，ControlPanel通过onUpdate回调函数传递的新值和旧值来计算新的计数总和，如果由于某种原因导致某个按钮的点击更新没有通知到ControlPanel，结果就会让ControlPanel中的sum状态和所有子组件Counter的count状态之和不一致，这时候就导致了数据的不一致性。而解决这一问题的一个比较好的方案就是使数据源唯一。让各个组件保持和全局状态的一致性，这个全局状态就是唯一可靠的数据源。而这个全局状态就是我们要使用的Flux和Redux中的Store.除了state,利用prop在组件之间传递信息也会遇到问题，比如一个应用中包含三级或者三级以上的组件结构，顶层的祖父级组件想要传递一个数据给最低层的子组件，采用prop的方式，就只能通过父组件中转。及时父组件并不需要这个prop也必须支持这个prop,扮演好搬运工的角色。

1. Flux应用demo3



图：Flux的单向数据流

<1>Dispatcher,处理动作分发，维持Store之间的依赖关系；（派发action）

<2>Store,负责存储数据和处理数据相关逻辑；

<3>Action,驱动Dispatcher的JavaScript对象；

<4>View，视图部分，负责显示用户界面。

Flux的好处：最重要的好处就是“单向数据流”的管理方式，在Flux的理念中，如果要改变界面，必须改变Store中的状态，如果要改变Store中的状态，必须派发一个action对象，这就是规矩，这样一个应用的逻辑就很清晰。

Flux的不足：1.Store之间依赖关系，例如上面例子中的waitFor()函数，因为SummaryStore对action类型的处理依赖于CounterStore已经处理过了，所以必须通过WaitFor函数告诉Dispatcher，先让CounterStore处理这些action对象，等到CounterStore搞定之后SummaryStore才能继续。2.难以进行服务器端的渲染。所以后面会引出Redux的store。

1. Redux的基本原则,Redux在Flux的基本原则“单向数据流”的基础上，增加了以下3个原则。

<1>唯一数据源

<2>保持状态只读

<3>数据改变只能通过纯函数完成

1. 组件拆分（容器组件和傻瓜组件）

一个React组件基本功能包括以下两点：

<1>和Redux Store打交道，读取Store的状态，用于初始化组件的状态，同时还要监听Store的状态改变；当Store状态发生变化时，需要更新组件状态，从而驱动组件重新渲染；当需要更新Store状态时，就要派发action对象；

<2>根据当前props和state,渲染出用户界面。

React的设计原则是让一个组件只专注做一件事情，如果发现一个组件做的事情太多了，就可以把这个组件拆分成多个组件，让每个组件值专注做一件事。

容器组件：承担第一个任务的组件，负责和Redux Store打交道，处于外层

展示组件（傻瓜组件）：承担第二个任务的组件，专心负责渲染界面的组件，处于内层，是一个纯函数，根据props产生结果。

Demo5可以展示容器组件和傻瓜组件如何协同工作，是对Demo4的改进，只有试图部分代码改变。

从上面的例子我们发现一个问题，无论是Counter组件还是Summary组件文件，他们都是直接导入Redux Store。这样不利于组件的复用，一个应用中最好只有一个地方需要直接导入Store,这个位置应该是调用最顶层React组件的位置。为了使用的话就需要prop去传递，但是这样又需要所有组件都帮助搬运这个props。React提供了一个叫Context的功能，完美的解决了这个问题。所谓Context,就是“上下文环境”让一个树状组件上的所有组件都能访问一个共同的对象。这就引出了Provider组件，它将是一个通用的context提供者，可以应用在任何一个应用中。

1. 从上面可以看出，我们使用了两个方法改进React应用，第一个是把一个组件拆分为容器组件和傻瓜组件，第二个是使用React的Context来提供一个所有组件都可以直接访问的Context，不难发现这两种方法都有套路，可以把套路部分抽取出来复用，而已经存在完成这种复用的库react-redux。使用该库，会实现代码的简化。详见demo6.具体可以了解connect和Provider的实现。
2. 表格展示react-redux的进化历程，以及进化原因。

|  |  |  |
| --- | --- | --- |
|  | 优点 | 局限 |
| React | 虚拟DOM/JSX/函数编程思想 | React的state和prop存在局限，改进使用store |
| Flux | 单向数据流/action/dispatch/store/view | 1. Store之间的依赖关系 2. 难以进行服务器端渲染 3. Store之间混杂了逻辑和状态 |
| Redux | 单向数据流/唯一数据源/状态只读/数据改变只能通过纯函数完成 | 组件功能过于复杂，可以优化为容器组件和傻瓜组件，需要提供一个所有组件可以直接访问的Context |
| React-redux | 提供了Provider/connect实现了上面Redux的局限，使开发效率更高。 |  |