React 组件 React Props

# React State(状态)

React 把组件看成是一个状态机（State Machines）。通过与用户的交互，实现不同状态，然后渲染 UI，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）。

以下实例创建一个名称扩展为 React.Component 的 ES6 类，在 render() 方法中使用 this.state 来修改当前的时间。

添加一个类构造函数来初始化状态 this.state，类组件应始终使用 props 调用基础构造函数。

**React 实例**

```
class Clock extends React.Component {
constructor(props) {
super(props);
this.state = {date: new Date()};
}
render() {
return (
<div>
<h1>Hello, world!</h1>
<h2>现在是 {this.state.date.toLocaleTimeString()}.</h2>
</div>
);
}
}
ReactDOM.render(
<Clock />,
document.getElementById('example')
);
```

尝试一下 »

接下来，我们将使Clock设置自己的计时器并每秒更新一次。

## 将生命周期方法添加到类中

在具有许多组件的应用程序中，在销毁时释放组件所占用的资源非常重要。

每当 Clock 组件第一次加载到 DOM 中的时候，我们都想生成定时器，这在 React 中被称为**挂载**。

同样，每当 Clock 生成的这个 DOM 被移除的时候，我们也会想要清除定时器，这在 React 中被称为**卸载**。

我们可以在组件类上声明特殊的方法，当组件挂载或卸载时，来运行一些代码：

**React 实例**

```
class Clock extends React.Component {
constructor(props) {
super(props);
this.state = {date: new Date()};
}
componentDidMount() {
```

```
this.timerID = setInterval(
() => this.tick(),
1000
);
}
componentWillUnmount() {
clearInterval(this.timerID);
}
tick() {
this.setState({
date: new Date()
});
}
render() {
return (
<div>
<h1>Hello, world!</h1>
<h2>现在是 {this.state.date.toLocaleTimeString()}.</h2>
</div>
);
}
}
ReactDOM.render(
<Clock />,
document.getElementById('example')
);
```

尝试一下 »

**实例解析：**

**`componentDidMount()`** 与 **`componentWillUnmount()`** 方法被称作生命周期钩子。

在组件输出到 DOM 后会执行 **`componentDidMount()`** 钩子，我们就可以在这个钩子上设置一个定时器。

this.timerID 为定时器的 ID，我们可以在 **`componentWillUnmount()`** 钩子中卸载定时器。

**代码执行顺序：**

1. 当 `<Clock />` 被传递给 `ReactDOM.render()` 时，React 调用 `Clock` 组件的构造函数。 由于 `Clock` 需要显示当前时间，所以使用包含当前时间的对象来初始化 `this.state` 。 我们稍后会更新此状态。

2. React 然后调用 `Clock` 组件的 `render()` 方法。这是 React 了解屏幕上应该显示什么内容，然后 React 更新 DOM 以匹配 `Clock` 的渲染输出。

3. 当 `Clock` 的输出插入到 DOM 中时，React 调用 `componentDidMount()` 生命周期钩子。 在其中，`Clock` 组件要求浏览器设置一个定时器，每秒钟调用一次 `tick()`。

4. 浏览器每秒钟调用 `tick()` 方法。 在其中，`Clock` 组件通过使用包含当前时间的对象调用 `setState()` 来调度UI更新。 通过调用 `setState()` ，React 知道状态已经改变，并再次调用 `render()` 方法来确定屏幕上应当显示什么。 这一次，`render()` 方法中的 `this.state.date` 将不同，所以渲染输出将包含更新的时间，并相应地更新 DOM。

5. 一旦 `Clock` 组件被从 DOM 中移除，React 会调用 `componentWillUnmount()` 这个钩子函数，定时器也就会被清除。

## 数据自顶向下流动

父组件或子组件都不能知道某个组件是有状态还是无状态，并且它们不应该关心某组件是被定义为一个函数还是一个类。

这就是为什么状态通常被称为局部或封装。 除了拥有并设置它的组件外，其它组件不可访问。

以下实例中 FormattedDate 组件将在其属性中接收到 date 值，并且不知道它是来自 Clock 状态、还是来自 Clock 的属性、亦或手工输入：

**React 实例**

```
function FormattedDate(props) {
return <h2>现在是 {props.date.toLocaleTimeString()}.</h2>;
}
class Clock extends React.Component {
constructor(props) {
super(props);
this.state = {date: new Date()};
}
componentDidMount() {
this.timerID = setInterval(
() => this.tick(),
1000
);
}
componentWillUnmount() {
clearInterval(this.timerID);
}
tick() {
this.setState({
date: new Date()
});
}
render() {
return (
<div>
<h1>Hello, world!</h1>
<FormattedDate date={this.state.date} />
</div>
);
}
}
ReactDOM.render(
<Clock />,
document.getElementById('example')
);
```

尝试一下 »

这通常被称为自顶向下或单向数据流。 任何状态始终由某些特定组件所有，并且从该状态导出的任何数据或 UI 只能影响树中下方的组件。

如果你想象一个组件树作为属性的瀑布，每个组件的状态就像一个额外的水源，它连接在一个任意点，但也流下来。

为了表明所有组件都是真正隔离的，我们可以创建一个 App 组件，它渲染三个Clock：

**React 实例**

```
function FormattedDate(props) {
return <h2>现在是 {props.date.toLocaleTimeString()}.</h2>;
}
class Clock extends React.Component {
constructor(props) {
super(props);
this.state = {date: new Date()};
}
componentDidMount() {
this.timerID = setInterval(
() => this.tick(),
1000
);
}
componentWillUnmount() {
clearInterval(this.timerID);
}
tick() {
this.setState({
date: new Date()
});
}
render() {
return (
<div>
<h1>Hello, world!</h1>
<FormattedDate date={this.state.date} />
</div>
);
}
}
function App() {
return (
<div>
<Clock />
<Clock />
<Clock />
</div>
);
}
ReactDOM.render(<App />, document.getElementById('example'));
```

尝试一下 »

以上实例中每个 Clock 组件都建立了自己的定时器并且独立更新。

在 React 应用程序中，组件是有状态还是无状态被认为是可能随时间而变化的组件的实现细节。

我们可以在有状态组件中使用无状态组件，也可以在无状态组件中使用有状态组件。

React 组件 React Props

## 1 篇笔记

写笔记

关于挂载时的 setInterval 中调用 tick() 的方式 ()=>this.tick()：

**1、()=>this.tick()**

`()=>this.tick()` 是 ES6 中声明函数的一种方式，叫做箭头函数表达式，引入箭头函数有两个方面的作用：更简短的函数并且不绑定 this。

```
var f = ([参数]) => 表达式（单一）
// 等价于以下写法
var f = function([参数]){
   return 表达式;
}
```

箭头函数的基本语法如下：

```
(参数1, 参数2, …, 参数N) => { 函数声明 }
(参数1, 参数2, …, 参数N) => 表达式（单一）
//相当于：(参数1, 参数2, …, 参数N) =>{ return 表达式; }

// 当只有一个参数时，圆括号是可选的：
(单一参数) => {函数声明}
单一参数 => {函数声明}

// 没有参数的函数应该写成一对圆括号。
() => {函数声明}
```

根据以上概念，尝试将 setInterval 中调用 tick() 的方式改为通常声明方式：

```
this.timerID = setInterval(function(){
    return this.tick();
  },1000
);
```

但是会报错，tick() 不是一个方法。

**2、this.tick()**

this.tick() 中的 this 指代的是 function，而不是我们想要的指代所在的组件类 Clock，所以我们要想办法让 this 能被正常指代。这里我们采用围魏救赵的办法:

```
let that = this;
this.timerID = setInterval(function () {
  return that.tick();
},1000);
```

在闭包函数的外部先用 that 引用组件 Clock 中挂载组件方法 componentDidMount() 中 this 的值，然后在 setInterval 中闭包函数中使用that，that 无法找到声明，就会根据作用域链去上级（上次层）中继承 that，也就是我们引用的组件类 Clock 中的 this。

到此为止，将 () => this.tick()等价代换为了我们熟悉的形式。

ch4o5 6个月前 (09-07)