第3章 组件的生命周期

在组件的整个生命周期中，随着该组件的 props 或者 state 发生改变，它的 DOM 表现也将有相应的变化。正如在介绍 JSX 的第 2 章中所提到的，一个组件就是一个状态机：对于特定的输入，它总会返回一致的输出。

React 为每个组件提供了生命周期钩子函数去响应不同的时刻——创建时、存在期及销毁时。我们在这里将按照这些时刻出现的顺序依次介绍——从实例化开始，到活动期，直到最后被销毁。

生命周期方法

React 的组件拥有简洁的生命周期 API，它仅仅提供你所需要的方法，而不会去追求全面。接下来我们按照它们在组件中的调用顺序来看一下每个方法。

实例化

一个实例初次被创建时所调用的生命周期方法与其他各个后续实例被创建时所调用的方法略有不同。当你首次使用一个组件类时，你会看到下面这些方法依次被调用：

- `getDefaultProps`
- `getInitialState`
- `componentWillMount`
- `render`

- componentDidMount

对于该组件类的所有后续应用，你将会看到下面的方法依次被调用。注意，getDefaultProps 方法已经不在列表中。

- getInitialState
- componentWillMount
- render
- componentDidMount

存在期

随着应用状态的改变，以及组件逐渐受到影响，你将会看到下面的方法依次被调用：

- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate
- render
- componentDidUpdate

销毁 & 清理期

最后，当该组件被使用完成后，componentWillUnmount 方法将会被调用，目的是给这个实例提供清理自身的机会。

现在，我们将会依次详细介绍到这三个阶段：实例化、存在期及销毁 & 清理期。

实例化

当每个新的组件被创建、首次渲染时，有一系列的方法可以用来为其做准备工作。这些方法中的每一个都有明确的职责，如下所示。

getDefaultProps

对于组件类来说，这个方法只会被调用一次。对于那些没有被父辈组件指定 props 属性的新建实例来说，这个方法返回的对象可用于为实例设置默认的 props 值。

> 值得注意的是，任何复杂的值，比如对象和数组，都会在所有的实例中共享——而不是拷贝或者克隆。

getInitialState

对于组件的每个实例来说，这个方法的调用次数有且只有一次。在这里你将有机会初始化每个实例的 state。与 `getDefaultProps` 方法不同的是，每次实例创建时该方法都会被调用一次。在这个方法里，你已经可以访问到 `this.props`。

componentWillMount

该方法会在完成首次渲染之前被调用。这也是在 render 方法调用前可以修改组件 state 的最后一次机会。

render

在这里你会创建一个虚拟 DOM，用来表示组件的输出。对于一个组件来说，render 是唯一一个必需的方法，并且有特定的规则。render 方法需要满足下面几点：

- 只能通过 `this.props` 和 `this.state` 访问数据。
- 可以返回 `null`、`false` 或者任何 React 组件。
- 只能出现一个顶级组件（不能返回一组元素）。
- 必须纯净，意味着不能改变组件的状态或者修改 DOM 的输出。

render 方法返回的结果不是真正的 DOM，而是一个虚拟的表现，React 随后会把它和真实的 DOM[1]做对比，来判断是否有必要做出修改。

componentDidMount

在 render 方法成功调用并且真实的 DOM 已经被渲染之后，你可以在 `componentDidMount` 内部通过 `this.getDOMNode()` 方法访问到它。

[1]原文是 “real DOM”，应该理解为内存中的 DOM 表现，而非浏览器中的。——译者注

这就是你可以用来访问原始 DOM 的生命周期钩子函数。比如，当你需要测量渲染出 DOM 元素的高度，或者使用计时器来操作它，亦或运行一个自定义的 jQuery 插件时，可以将这些操作挂载到这个方法上。

举例来说，假设需要在一个通过 React 渲染出的表单元素上使用 jQuery UI 的 Autocomplete 插件，则可以这样使用它：

```
// 需要自动补全的字符串列表
var datasource = [...];

var MyComponent = React.createClass({
  render: function () {
    return <input ... />;
  },
  componentDidMount: function () {
    $(this.getDOMNode()).autocomplete({
      sources: datasource
    });
  }
});
```

> 注意，当 React 运行在服务端时，componentDidMount 方法不会被调用。

存在期

此时，组件已经渲染好并且用户可以与它进行交互。通常是通过一次鼠标点击、手指点按或者键盘事件来触发一个事件处理器。随着用户改变了组件或者整个应用的 state，便会有新的 state 流入组件树，并且我们将会获得操控它的机会。

componentWillReceiveProps

在任意时刻，组件的 props 都可以通过父辈组件来更改。出现这种情况时，componentWillReceiveProps 方法会被调用，你也将获得更改 props 对象及更新 state 的机会。

比如，在我们的问卷制作工具中有一个 AnswerRadioInput 组件，允许用户切换一个单选框。父辈组件能够改变这个布尔值，并且我们可以对其做出响应，比如基于父辈组件输入的 props 更新组件自身的内部 state。

```
componentWillReceiveProps: function(nextProps) {
  if (nextProps.checked !== undefined) {
    this.setState({
      checked: nextProps.checked
    });
  }
}
```

> 我们有一个贯穿全书的示例项目，一个问卷制作工具，你可以在 *https://github.com/backstopmedia/bleeding-edge-sample-app* 阅读全部源码。

shouldComponentUpdate

React 非常快。不过你还可以让它更快——通过调用 `shouldComponentUpdate` 方法在组件渲染时进行精确优化。

如果你确定某个组件或者它的任何子组件不需要渲染新的 props 或者 state，则该方法会返回 `false` 。

> 在首次渲染期间或者调用了 `forceUpdate` 方法后，这个方法不会被调用。

返回 false 则是在告诉 React 要跳过调用 `render` 方法，以及位于 render 前后的钩子函数：`componentWillUpdate` 和 `componentDidUpdate` 。

该方法是非必需的，并且大多数情况下没必要在开发中使用它。草率地使用它可能导致不可思议的 bug，所以最好等到能够准确地测量出应用的瓶颈后，再去选择要在何处进行恰当的优化。

如果你谨慎地使用了不可变的数据结构作为 state，同时只在 render 方法中读取 props 和 state 中的数据，那你就可以放心地重写 `shouldComponentUpdate` 方法来比较新旧 props 及 state 了。

另外一个关于性能调优的选项是 React 插件提供的 `PureRenderMixin` 方法。如果你的组件是纯净的，即对于相同的 props 和 state，它总会渲染出一样的 DOM，那么这个 mixin 会自动调用 `shouldComponentUpdate` 方法来比较 props 和 state，如果比较结果一致则返回 false。

componentWillUpdate

和 componentWillMount 方法类似，组件会在接收到新的 props 或者 state 进行渲染之前，调用该方法。

注意，你不可以在该方法中更新 state 或者 props。而应该借助 componentWillReceiveProps 方法在运行时更新 state。

componentDidUpdate

和 componentDidMount 方法类似，该方法给了我们更新已经渲染好的 DOM 的机会。

销毁 & 清理期

每当 React 使用完一个组件，这个组件就必须从 DOM 中卸载随后被销毁。此时，仅有的一个钩子函数会做出响应，完成所有的清理和销毁工作，这很必要。

componentWillUnmount

最后，随着一个组件从它的层级结构中移除，这个组件的生命也走到了尽头。该方法会在组件被移除之前被调用，让你有机会做一些清理工作。你在 componentDidMount 方法中添加的所有任务都需要在该方法中撤销，比如创建的定时器或者添加的事件监听器。

反模式：把计算后的值赋给 state

值得注意的是，在 getInitialState 方法中，尝试通过 this.props 来创建 state 的做法是一种反模式。React 专注于维护数据的单一来源。它的设计使得传递数据的来源更加显而易见，这也是 React 的一个优势。

上文提到从 props 中计算值然后将它赋值为 state 的做法是一种反模式。比如，在组件中，把日期转化为字符串形式，或者在渲染之前将字符串转换为大写。这些都不是 state，只能够在渲染时进行计算。

当组件的 state 值和它所基于的 prop 不同步，因而无法了解到 render 函数的内部结构时，可以认定为一种反模式。

```
// 反模式：经过计算后值不应该赋给 state
getDefaultProps: function() {
  return {
    date: new Date()
  };
},
getInitialState: function() {
  return {
    day: this.props.date.getDay()
  }
},
render: function() {
  return <div>Day: {this.state.day}</div>;
}
```

正确的模式应该是在渲染时计算这些值。这保证了计算后的值永远不会与派生出它的 props 值不同步。

```
// 在渲染时计算值是正确的
getDefaultProps: function() {
  return {
    date: new Date()
  };
},
render: function() {
  var day = this.props.date.getDay();
  return <div>Day: {day}</div>;
}
```

然而，如果你的目的并不是同步，而只是简单的初始化 state，那么在 getInitialState 方法中使用 props 是没问题的。只是一定要明确你的意图，比如为 prop 添加 initial 前缀。

```
getDefaultProps: function () {
  return {
    initialValue: 'some-default-value'
  };
},
getInitialState: function () {
```

```
  return {
    value: this.props.initialValue
  };
},
render: function () {
  return <div>{this.props.value}</div>
}
```

总结

React 生命周期方法提供了精心设计的钩子函数，会伴随组件的整个生命周期。和状态机类似，每个组件都被设计成了能够在整个生命周期中输出稳定、语义化的标签。

组件不会独立存在。随着父组件将 props 推送给它们的子组件，以及那些子组件渲染它们自身的子组件，你必须谨慎地考虑数据是如何流经整个应用的。每一个子组件真正需要掌控多少数据，哪个组件来控制应用的状态？这些涉及了下一章的话题：数据流。