React 状态管理（1）：组件状态

在前面的章节中，我们反复声明过 React 其实就是这样一个公式：

```
UI = f(data)
```

f 的参数 data，除了 props，就是 state。props 是组件外传递进来的数据，state 代表的就是 React 组件的内部状态。

为什么要了解 React 组件自身状态管理

可能读者也知道在 React 开发社区中 Redux 和 Mobx 这样的状态管理工具，不过，我们首先不要管这些第三方工具，先从了解 React 组件自身的管理开始。

为什么呢？

第一个原因，因为 React 组件自身的状态管理是基础，其他第三方工具都是在这个基础上构筑的，连基础都不了解，无法真正理解第三方工具。

另一个重要原因，对于很多应用场景，React 组件自身的状态管理就足够解决问题，犯不上动用 Redux 和 MobX 这样的大杀器，简单问题简单处理，可以让代码更容易维护。

组件自身状态 state

什么数据放在 state 中

对于 React 组件而言，数据分为两种：

1. props
2. state

二者的区别显而易见，简单说就是，props 是外部传给组件的数据，而 state 是组件自己维护的数据，对外部是不可见的。

所以，判断某个数据以 props 方式存在，还是以 state 方式存在，并不难，只需要判断这个状态是否是组件内部状态。

一个经常被问到的问题，就是为什么不把组件的数据直接存放在组件类的成员变量中？比如像下面这样：

```
class Foo extends React.Component {
  foo = 'foo'

  render() {
    return (
      <React.Fragment>{this.foo}</React.Fragment>
    );
  }
}
```

像上面，数据存在 `this.foo` 中，而不是存在 `this.state.foo` 中，当这个组件渲染的时候，当然 `this.foo` 的值也就被渲染出来了，问题是，更新 `this.foo` 并不会引发组件的重新渲染，这很可能不是我们想要的。

所以，判断一个数据应该放在哪里，用下面的原则：

1. 如果数据由外部传入，放在 props 中；
2. 如果是组件内部状态，是否这个状态更改应该立刻引发一次组件重新渲染？如果是，放在 state 中；不是，放在成员变量中。

修改 state 的正确方式

组件自身的状态可以通过 `this.state` 读到，`this.state` 本身就是一个对象，但是修改状态不应该通过直接修改 `this.state` 对象来完成。因为，我们修改 state，当然不只是想修改这个对象的值，而是想引发 React 组件的重新渲染。

```
this.state.foo = 'bar'; //错误的方式

this.setState({foo:'bar'}); //正确的方式
```

如上面代码所示，如果只是修改 `this.state`，那改了也就只是改了这个对象，其他的什么都不会发生；如果使用 `setState` 函数，那不光修改 `state`，还能引发组件的重新渲染，在重新渲染中就会使用修改后的 `state`，这也就是达到根据 `state` 改变公式左侧 UI 的目的。

```
UI = f(state)
```

state 改变引发重新渲染的时机

现在我们知道应该用 `setState` 函数来修改组件 state，而且可以引发组件重新渲染，有意思的是，并不是一次 `setState` 调用肯定会引发一次重新渲染。

这是 React 的一种性能优化策略，如果 React 对每一次 `setState` 都立刻做一次组件重新渲染，那代价有点大，比如下面的代码：

```
this.setState({count: 1});
this.setState({caption: 'foo'});
this.setState({count: 2});
```

连续的同步调用 `setState`，第三次还覆盖了第一次调用的效果，但是效果只相当于调用了下面这样一次：

```
this.setState({count: 2, caption: 'foo'});
```

虽然明智的开发者不会故意连续写三个 setState 调用，但是代码一旦写得复杂，可能有多个 setState 分布在一次执行的不同代码片段中，还是会同步连续调用 setState，这时候，如果真的每个 setState 都引发一次重新渲染，实在太浪费了。

React 非常巧妙地用任务队列解决了这个问题，可以理解为每次 setState 函数调用都会往 React 的任务队列里放一个任务，多次 setState 调用自然会往队列里放多个任务。React 会选择时机去批量处理队列里执行任务，当批量处理开始时，React 会合并多个 setState 的操作，比如上面的三个 setState 就被合并为只更新 state 一次，也只引发一次重新渲染。

因为这个任务队列的存在，React 并不会同步更新 state，所以，在 React 中，setState 也不保证同步更新 state 中的数据。

state 不会被同步修改

简单说来，调用 setState 之后的下一行代码，读取 this.state 并不是修改之后的结果。

```
  console.log(this.state.count);// 修改之前this.state.count为0
  this.setState({count: 1})
  console.log(this.state.count);// 在这里this.state.count依然为0
```

这乍看是很让人费解的结果，但是如果你理解了上面 React 任务队列的设计，一切也不难理解。

setState 只是给任务队列里增加了一个修改 this.state 的任务，这个任务并没有立即执行，所以 this.state 并不会立刻改变。

好吧，其实问题也没有那么简单，上面我所举的例子中，都假设 setState 是由 React 的生命周期函数或者事件处理函数中同步调用，这种情况下 setState 不会立即同步更新 state 和重新渲染，但是，如果调用 setState 由其他条件引发，就不是这样了。

看下面的代码，结果可能会出乎你的所料：

```
  setTimeout(() => {
    this.setState({count: 2}); //这会立刻引发重新渲染
    console.log(this.state.count); //这里读取的count就是2
  }, 0);
```

为什么 setTimeout 能够强迫 setState 同步更新 state 呢？

可以这么理解，当 React 调用某个组件的生命周期函数或者事件处理函数时，React 会想：“嗯，这一次函数可能调用多次 setState，我会先打开一个标记，只要这个标记是打开的，所有的 setState 调用都是往任务队列里放任务，当这一次函数调用结束的时候，我再去批量处理任务队列，然后把这个标记关闭。”

因为 setTimeout 是一个 JavaScript 函数，和 React 无关，对于 setTimeout 的第一个函数参数，这个函数参数的执行时机，已经不是 React 能够控制的了，换句话说，React 不知道什么时候这个函数参数会被执行，所以那个“标记”也没有打开。

当那个“标记”没有打开时，setState 就不会给任务列表里增加任务，而是强行立刻更新 state 和引发重新渲染。这种情况下，React 认为：“这个 setState 发生在自己控制能力之外，也许开发者就是想要强行同步更新呢，宁滥勿缺，那就同步更新了吧。”

知道这个“技巧”之后，可能会有开发者说：好啊，那么以后我就用 setTimeout 来调用 setState 吧，能够立刻更新 state，多好！

我劝你不要这么做。

就像上面所说，React 选择不同步更新 state，是一种性能优化，如果你用上 setTimeout，就没机会让 React 优化了。

而且，每当你觉得需要同步更新 state 的时候，往往说明你的代码设计存在问题，绝大部分情况下，你所需要的，并不是“state 立刻更新”，而是，“确定 state 更新之后我要做什么”，这就引出了 setState 另一个功能。

setState 的第二个参数

setState 的第二个参数可以是一个回调函数，当 state 真的被修改时，这个回调函数会被调用。

```
  console.log(this.state.count); // 0
  this.setState({count: 1}, () => {
    console.log(this.state.count); // 这里就是1了
  })
  console.log(this.state.count); // 依然为0
```

当 setState 的第二个参数被调用时，React 已经处理完了任务列表，所以 this.state 就是更新后的数据。

如果需要在 state 更新之后做点什么，请利用第二个参数。

函数式 setState

不管怎么说，setState 不能同步更新的确会带来一些麻烦，尤其是多个 setState 调用之间有依赖关系的时候，很容易写错代码。

一个很典型的例子，当我们不断增加一个 state 的值时：

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

上面的代码表面上看会让 this.state.count 增加 3，实际上只增加了 1，因为 setState 没有同步更新 this.state 啊，所以给任务队列加的三个任务都是给 this.state.count 同一个值而已。

面对这种情况，我们很自然地想到，如果任务列表中的任务不只是给 state 一个固定数据，如果任务列表里的“任务”是一个函数，能够根据当前 state 计算新的状态，那该多好！

实际上，setState 已经支持这种功能，到现在为止我们给 setState 的第一个参数都是对象，其实也可以传入一个函数。

当 setState 的第一个参数为函数时，任务列表上增加的就是一个可执行的任务函数了，React 每处理完一个任务，都会更新 this.state，然后把新的 state 传递给这个任务函数。

setState 第一个参数的形式如下：

```
function increment(state, props) {
  return {count: state.count + 1};
}
```

可以看到，这是一个纯函数，不光接受当前的 state，还接受组件的 props，在这个函数中可以根据 state 和 props 任意计算，返回的结果会用于修改 this.state。

如此一来，我们就可以这样连续调用 setState：

```
  this.setState(increment);
  this.setState(increment);
  this.setState(increment);
```

用这种函数式方式连续调用 setState，就真的能够让 this.state.count 增加 3，而不只是增加 1。

小结

通过这一小节，读者应该能够明白：

1. 如何确定数据以 props 还是以 state 形式存在；
2. 更新 state 的正确方法；
3. setState 通常并不会立刻更新 state；
4. 函数参数形式的 setState 才是推荐的用法。

留言

评论将在后台进行审核，审核通过后对所有人可见

鲜知
setTimeout可把setState异步变为同步，这个有点意思
▲ 0 评论 9天前

sanseo 前端工程师
函数式的setState很厉害，完美解决了多次setState互相依赖的嵌套问题
▲ 0 评论 18天前

阿五 web前端开发工程师
函数式的setState真的挺好用，一般都会这么写
this.setState((preState, props) => ({})
▲ 1 评论 25天前

冯冯墨墨 默默耕耘的搬砖者
如果要在setState方法后，直接取用更动后的state值，正确的使用方式，在官方文件中的说明，需要利用setState的第二传参，传入一个回调(callback)函式.

因为setState这个方法，它在React中的执行行为可以认为"异步的"
▲ 2 评论 1月前

肖炎 前端开发 @ 今日头条
React 每处理完一个任务，都会更新 this.state，然后把新的 state 传递给这个任务函数(ps:是不是应该是传递给下一个任务函数)
▲ 0 评论 1月前

大伙子~😣
setState的第一个参数也可以是一个函数，这个涨姿势了哈哈
▲ 0 评论 1月前