React 状态管理（3）：Mobx 使用模式

既然说了 Redux，没有理由不聊一聊 Mobx，这两个状态管理工具在业界都被广泛应用，走得却是完全不同的两条路。

理解 Mobx

虽然 Mobx 和 Redux 有很大不同，但是至少还有一个共同点——这两个工具都和 React 没有任何直接关系，只不过凑巧 React 社区大量使用它们罢了。从技术上说，Mobx 和 Redux 都是中立的状态管理工具，他们能够应用于 React，也可以用于其他需要状态管理的场景。

还是用 create-react-app 产生的应用来演示 Mobx 吧，要使用 Mobx，首先我们要在项目中安装对应的 npm 包。

```
npm install mobx
```

我们用 Mobx 来实现一个很简单的计数工具，首先，需要有一个对象来记录计数值，代码如下：

```
import {observable} from 'mobx';

const counter = observable({
  count: 0
});
```

在上面的代码中， `counter` 是一个对象，其实就是用 `observable` 函数包住一个普通 JavaScript 对象，但是 `observable` 的介入，让 `counter` 对象拥有了神力。

我们用最简单的代码来展示这种“神力”，代码如下：

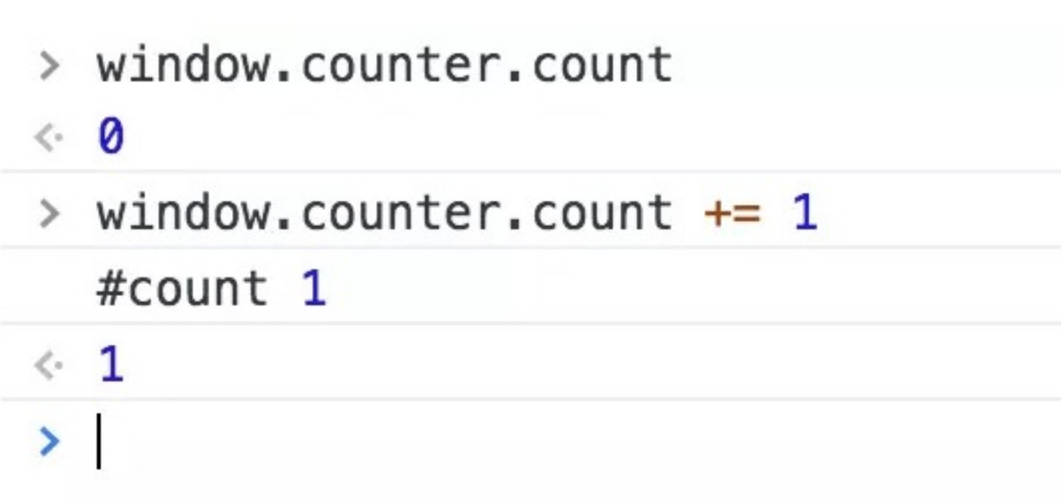
```
import {autorun} from 'mobx';

window.counter = counter;

autorun(() => {
  console.log('#count', counter.count);
});
```

把 `counter` 赋值给 `window.counter`，是为了让我们在 Chrome 的开发者界面可以访问。用 `autorun` 包住了一个函数，这个函数输出 `counter.count` 的值，这段代码的作用，我们很快就能看到。

在 Chrome 的开发者界面，我们可以直接访问 `window.counter.count`，神奇之处是，如果我们直接修改 `window.counter.count` 的值，可以直接触发 `autorun` 的函数参数！



```
> window.counter.count
< 0
> window.counter.count += 1
  #count 1
< 1
> |
```

这个现象说明， `mobx` 的 `observable` 拥有某种“神力”，任何对这个对象的修改，都会立刻引发某些函数被调用。和 observable 这个名字一样，被包装的对象变成了“被观察者”，而被调用的函数就是“观察者”，在上面的例子中，autorun 的函数参数就是“观察者”。

Mobx 这样的功能，等于实现了设计模式中的“观察者模式”（Observer Pattern），通过建立 observer 和 observable 之间的关联，达到数据联动。不过，传统的“观察者模式”要求我们写代码建立两者的关联，也就是写类似下面的代码：

```
observable.register(observer);
```

Mobx 最了不起之处，在于不需要开发者写上面的关联代码，Mobx自己通过解析代码就能够自动发现 observer 和 observable 之间的关系。

我们很自然想到，如果让我们的数据拥有这样的“神力”，那我们就不用在修改完数据之后，再费心去调用某些函数使用这些数据了，数据管理会变得十分轻松。

decorator

因为 Mobx 的作用就是把简单的对象赋予神力，总要有一种方法能够在不改变对象代码的前提，去改变对象的行为，这就用得上“装饰者模式”（Decorator Pattern）。

单独说“装饰者模式”，这只是面向对象编程思想下的一种模式，不过对 JavaScript 语言而言，就不只是一种模式，而是一种语言特性，它在语法上对这种模式提供了强大的支持，所谓强大，就是指使用起来代码极其简洁。

根据 JavaScript 语法，我们可以这样创造一个 decorator，叫做 `log`：

```
function log(target, name, descriptor) {
  console.log('#target', target);
  console.log('#name', name);
  console.log('#descriptor', descriptor);
  return descriptor;
}
```

当然，很明显这个 decorator 什么实质的事情都没做，只是用 `console.log` 输出了三个参数秀了一下存在感，最后返回的 `descriptor`，就是被这个『装饰者』所『装饰』的对象。

下面是使用这个 decorator 的代码示例：

```
@log
class Bar {
  @log
  bar() {
    console.log('bar');
  }
}
```

可以看到， `@` 符号就是使用 decorator 的标志，将 `@log` 作用于一个类 `Bar`，那么最后得到的 `Bar` 其实是调用 `log` 函数返回的结果；将 `@log` 作用于一个类成员 `@bar`，最后得到的 `bar` 同样是调用 `log` 函数之后得到的结果。可见，如果我们巧妙地编写 `log` 函数，控制返回的结果，就可以操纵被『装饰』的类或者成员。

编写 decorator 是一个复杂的过程，也超出了这本小册的范围，有兴趣的读者可以自行研究。在这里，读者只需要知道，虽然使用 Mobx 并不是必须使用 decorator，但是使用 decorator 会让 Mobx 的应用代码简洁易读很多。

在 create-react-app 中增加 decorator 支持

很不幸，create-react-app 产生的应用并不支持 decorator，官方解释 是：decorator 并没有成为稳定的标准，为了防止今天写的代码没多久就不好使，create-react-app 不会支持这样的不稳定的功能。

不过，这并不表示完全没有办法，事情可以解决，只是有些麻烦，我们要做的只是在应用中添加支持 decorator 的 babel plugin。

首先，我们动用 create-react-app 的 `eject` 功能，这表示我们和 create-react-app 缺省照顾一切的 `react-scripts` 一刀两断，从此之后，webpack 和 babel 配置就完全由我们自己控制。要注意，eject 是不可逆的，做了就收不回来了，所以，请谨慎使用 eject，不过，为了支持 decorator，我们也别无选择。

```
npm run eject
```

在 eject 之后，我们手动安装支持 decorator 的 babel 插件:

```
npm install --save-dev babel-plugin-transform-decorators-legacy
```

然后，我们找到 `package.json` 里面 `babel` 这一部分，添加 `plugins` 部分，让这一部分变成这个样子：

```
  "babel": {
    "plugins": [
      "transform-decorators-legacy"
    ],
    "presets": [
      "react-app"
    ]
  },
```

现在，我们离在 create-react-app 产生的应用中使用 decorator 只差一步了，记得我们说过 Mobx 和 React 并无直接关系吗？为了建立二者的关系，还需要安装 `mobx-react`：

```
npm install mobx-react
```

现在，我们可以使用 decorator 来操作 Mobx了。

用 decorator 来使用 Mobx

还是以 Counter 为例，看如何用 decorator 使用 Mobx，我们先看代码：

```
import {observable} from 'mobx';
import {observer} from 'mobx-react';

@observer
class Counter extends React.Component {
  @observable count = 0;

  onIncrement = () => {
    this.count ++;
  }

  onDecrement = () => {
    this.count --;
  }

  componentWillUpdate() {
    console.log('#enter componentWillUpdate');
  }

  render() {
    return(
      <CounterView
        caption="With decorator"
        count={this.count}
        onIncrement={this.onIncrement}
        onDecrement={this.onDecrement}
        />
    );
  }
}
```

在上面的代码中， `Counter` 这个 React 组件自身是一个 `observer`，而 `observable` 是 `Counter` 的一个成员变量 `count`。

注意 `observer` 这 个 `decorator` 来自于 `mobx-react`，它是 Mobx 世界和 React 的桥梁，被它“装饰”的组件，只要用到某个被 Mobx 的 `observable` “装饰”过的数据，自然会对这样的数据产生反应。所以，只要 `Counter` 的 `count` 成员变量一变化，就会引发 `Counter` 组件的重新渲染。

可以注意到， `Counter` 的代码中并没有自己的 state，其实，被 `observer` 修饰过之后， `Counter` 被强行"注入”了 state，只不过我们看不见而已。

独立的 Store

虽然把 observer 和 observable 集中在一个 React 组件中可行，但是，这也让 observable 的状态被封存在了 React 组件内，那我们直接用 React 自身的 state 管理也能解决问题，所以，这样使用 Mobx 意义不大。

更多适用于 Mobx 的场合，就和适用于 Redux 的场合一样，是一个状态需要多个组件共享，所以 observable 一般是在 React 组件之外。

我们重写一遍 Counter 组件，代码如下：

```
const store = observable({
  count: 0
});
store.increment = function() {
  this.count ++;
};
store.decrement = function() {
  this.count --;
}

@observer // this decorator is must
class Counter extends React.Component {
  onIncrement = () => {
    store.increment();
  }

  onDecrement = () => {
    store.decrement();
  }

  render() {
    return(
      <CounterView
        caption="With external state"
        count={store.count}
        onIncrement={this.onIncrement}
        onDecrement={this.onDecrement}
        />
    );
  }
}
```

可以看到，我们把 `count` 提到组件之外，甚至就把它叫做 store，这延续的是 Redux 的命名方法。

小结

在这一小节中，我们介绍了 Mobx，读者应该能学到：

1. Mobx 的基本功能就是“观察者模式”
2. decorator 是“装饰者模式”在 JavaScript 语言中的实现
3. Mobx 通常利用 decorator 来使用
4. 如何利用 mobx-react 来管理 React组件的状态

留言

评论将在后台进行审核，审核通过后对所有人可见

Artyhacker 前端 @ 北京某小国企
不想eject或不想使用装饰器的话可以直接用mobx里的decorate函数包一下就好，得多写几行代码。
▲ 0 评论 19天前

Hack-Jay
npm install mobx-mobx
打错了~
▲ 1 收起评论 1月前

程墨 Hulu
改过来了，谢谢指正。
1月前

评论审核通过后显示 评论

blacker
忽然想到一个问题，在vue和react中，数据的更新都是异步的，而不是同步的，也就是说这两个框架本身实现了数据变更时，组件的重新渲染是放到了队列中了，定时去更新组件，在Mobx中，发现数据变更了是不是也是一样有缓冲的机制呢
▲ 0 收起评论 1月前

程墨 Hulu
就我所知，Mobx本身是不会做延时更新的，因为Mobx就是一个单纯的数据管理，不应该考虑太多。React和Vue的延时更新都是为了提高性能，不单纯是数据管理，而是对UI的渲染优化。
1月前

评论审核通过后显示 评论

blacker
跟vue的响应式原理一样
▲ 1 评论 1月前

blacker
很有价值，带我认识了一个新的数据管理方式
▲ 0 评论 1月前

红谷滩陈冠希 前端 @ 无限996公司
并不需要 eject, 安装这个 @babel/plugin-proposal-decorators，配合 react-app-rewired 就能使用 Mobx
▲ 4 收起评论 1月前

kylin1993 FE
想怎么玩怎么玩
1月前

zhangyanling77 前端开发 @ 成都
方式多了去了，eject，rewired或者重写scrpts都可以
1月前

评论审核通过后显示 评论