React 状态管理（2）：Redux 使用模式

这一节我们来介绍 React 社区里名望很高的一个状态管理工具 Redux。不过，首先要明白，Redux 和 React 在技术上并没有什么直接关系。

虽然 Redux 的两个创造者 Dan Abramov 和 Andrew Clark 目前都是 React 的核心开发人员，而且 Redux 得到最广泛应用的场景是和 React 配合，但是，我们还是要明确，Redux 和 React 没有任何直接关系，Redux 可以用来管理 React 的状态，也可以用来管理其他应用的状态，只是由于历史原因 Redux 在 React 社区被应用最多罢了。

在出版《深入浅出React和Redux》一书后，我收到读者反馈，有不少读者反映看到 Redux 部分就看不大懂了，很大一部分原因，就是这些读者在学习 React 时已经习惯了用户界面的概念，突然出现一个和用户界面无关的 Redux，就觉得不好理解了。所以，我们还是从理解 Redux 开始。

理解 Redux

要理解 Redux，首先要明白我们为什么需要 Redux，或者说，Redux 适用于什么样的场景。

应用的状态往往十分复杂，如果应用状态就是一个普通 JavaScript 对象，而任何能够访问到这个对象的代码都可以修改这个状态，就很容易乱了套。当 bug 发生的时候，我们发现是状态错了，但是也很难理清到底谁把状态改错了，到底是如何走到出 bug 这一步。

Redux 的主要贡献，就是限制了对状态的修改方式，让所有改变都可以被追踪。

虽然 Redux 和 React 没有直接关系，但是我们依然以 React 应用为例，来说明 Redux 扮演什么角色。

在真实应用中，React 组件树会很庞大很复杂，两个没有父子关系的 React 组件之间要共享信息，怎么办呢?

最直观的方法，就是创建一个独立于这两个组件的对象，在这个对象中存放共享的数据，没错，这个对象，相当于一个 Store。

如果只是一个简单对象，那么任何人都可以修改 Store，这不大合适。所以我们做出一些限制，让 Store 只接受某些『事件』，如果要修改 Store 上的数据，就往 Store 上发送这些『事件』，Store 对这些『事件』的响应，就是修改状态。

这里所说的『事件』，就是 action，而对应修改状态的函数，就是 reducer。

Redux 中的 Store 其实实现的就是上述过程和概念，只不过实现很巧妙，让人没有办法绕过上面过程来修改状态，这种限制，是 Redux 成功的要素之一。

Redux 的内容并不少，具体使用方法超出了本小册的范围，因为本小册着重讲解高阶的技巧，读者如果对 Redux 还不熟悉，可以去看我写的书《深入浅出React和Redux》，了解 Redux 怎么用之后，更有利于学习后续内容。

适合 Redux 的场景

当一个 React 应用采用 Redux 之后，开发者往往就会陷入这样的纠结：对于某个状态，到底是放在 Redux 的 Store 中呢，还是放在 React 组件自身的状态中呢?

如果所有状态全都放在 Redux 的 Store 上，那就要对应增加 reducer 和 action 的代码，虽然拥有了可以跟踪的好处，但是对一些很细小的状态也要增加 reducer 和 action，会感觉很啰嗦（真的，Redux 本身就是一个啰嗦的技术，利用“啰嗦”来实现可维护性），开发者又会觉得得不偿失。

如果状态放在 React 组件中，感觉又白白放弃了 Redux 的优势，回到了 React 原生管理状态的老路上去，令人很不甘心。

面对这种左右为难的纠结状况，我这里有一套步骤，可以帮助开发者决定如何防止应用状态。

第一步，看这个状态是否会被多个 React 组件共享。

所谓共享，就是多个组件需要读取或者修改这个状态，如果是，那不用多想，应该放在 Store 上，因为 Store 上状态方便被多个组件共用，避免组件之间传递数据；如果不是，继续看第二步。

第二步，看这个组件被 unmount 之后重新被 mount，之前的状态是否需要保留。

举个简单例子，一个对话框组件。用户在对话框打开的时候输入了一些内容，不做提交直接关闭这个对话框，这时候对话框就被 unmount 了，然后重新打开这个对话框（也就是重新 mount），需求是否要求刚才输入的内容依然显示？如果是，那么应该把状态放在 Store 上，因为 React 组件在 unmount 之后其中的状态也随之消失了，要想在重新 mount 时重获之前的状态，只能把状态放在组件之外，Store 当然是一个好的选择；如果需求不要求重新 mount 时保持 unmount 之前的状态，继续看第三步。

第三步，到这一步，基本上可以确定，这个状态可以放在 React 组件中了。

不过，如果你觉得这个状态很复杂，需要跟踪修改过程，那看你个人喜好，可以选择放在 Store 上；如果你想简单处理，可以心安理得地让这个状态由 React 组件自己管理。

我想说明的是，React 组件的状态管理已经很强大了（在第 11 小节中有介绍），对于简单状态，尽量用 React 自己来搞定，只有那些适用场合不限于一个组件的，才有足够理由让 Redux 来管理。

代码组织方式

在应用中引入 Redux 之后，就会引入 action 和 reducer。从方便管理的角度出发，和 React 组件一样，action 和 reducer 都有自己独立的源代码文件，很自然，我们需要决定如何组织这些代码。

最傻的一种方法，就是把所有源代码文件放在一个目录下，代码文件少的时候还凑合着能看，一旦多起来，一个目录下各种类型文件会看花眼，所以这种方式不可取。

更好的方法，是把源代码文件分类放在不同的目录中，根据分类方式，可以分为两种：

1. 基于角色的分类（role based）
2. 基于功能的分类（feature based）

如果你曾经开发过 MVC 类应用，对基于角色的分类不会陌生。MVC 应用中在一个目录下放所有的 controller，在另一个目录下放所有的 view，在第三个目录下放所有的 model，每个目录下的文件都是同样的“角色”，这就是基于角色的分类。对应到使用 React 和 Redux 的应用，做法就是把所有 reducer 放在一个目录（通常就叫做 reducers)，把所有 action 放在另一个目录（通常叫 actions），最后，把所有的纯 React 组件放在另一个目录。

另一种基于功能的分类方式，是把一个模块相关的所有源代码放在一个目录。例如，对于博客系统，有 Post（博客文章）和 Comment（注释）两个基本模块，建立两个目录 Post 和 Comment，每个目录下都有各自的 `action.js` 和 `reducer.js` 文件，如下所示，每个目录都代表一个模块功能，这就是基于功能的分类方式。

```
Post -- action.js
     |_ reucer.js
     |_ view.js
Comment -- action.js
        |_ reucer.js
        |_ view.js
```

一般说来，基于功能的分类方式更优。因为每个目录是一个功能的封装，方便共享，不过，我们也看到很多应用依然采用基于角色的方式组织代码，连 Facebook 开源的一些应用也采用这种方法，这很大程度上是因为这些应用开发一个模块的时候，没想过有朝一日要分享这些模块，换句话说这些模块开发出来就只被指望在这个应用中使用，这样一来，基于功能的组织方式也就没有必要了。

具体用哪种方式来组织代码，主要就看你是否预期这些模块会被共享，如果会，那采用基于功能的方式就是首选。

react-redux 中的模式

因为 Redux 是一个中立的状态管理工具，和 React 没有直接联系，所以，如果在 React 应用中使用 Redux，我们除了要引入 Redux，还需要导入 react-redux 这个安装包，安装方法如下：

```
npm install redux react-redux
```

在第 8 小节，我们介绍了『提供者模式』，react-redux 就是『提供者模式』的实践。在组件树的一个比较靠近根节点的位置，我们通过 `Provider` 来引入一个 store，代码如下：

```
import {createStore} from 'redux';
import {Provider} from 'react-redux';

const store = createStore(...);

// JSX
  <Provider store={store}>
    { // Provider之下的所有组件都可以connect到给定的store }
  </Provider>
```

这个 `Provider` 当然也是利用了 React 的 Context 功能。在这个 Provider 之下的所有组件，如果使用 connect，那么『链接』的就是 Provider 的 state。

以最简单的 Counter 为例来介绍一下 connect 的用法，首先，我们需要一个『傻瓜组件』，可以由纯函数实现，如下：

```
const CounterView = ({count, onIncrement}) => {
  return (
    <div>
      <div>{count}</div>
      <button onClick={onIncrement}>+</button>
    </div>
  );
};
```

上面的 `CounterView` 没有自己的 state，完全依赖于外部存储计数值，那么计数值存在哪呢？存在 store 上。我们要做的就是把 CounterView 和 store 连接起来，代码如下：

```
import {connect} from 'react-redux';

const mapStateToProps = (state) => {
  return {
    count: state.count
  };
}

const mapDispatchToProps = (dispatch) => ({
  onIncrement: () => dispatch({type: 'INCREMENT'})
});

const Counter = connect(mapStateToProps, mapDispatchToProps)(CounterView);
```

这里的 `connect` 函数接受两个参数，一个 `mapStateToProps` 是把 Store 上的 state 映射为 props；另一个 `mapDispatchToProps` 则是把回调函数类型的 props 映射为派发 action 的动作，connect 函数调用会产生一个『高阶组件』。

在第 6 小节我们介绍过『高阶组件』模式，一个高阶组件就是一个函数，它接受 React 组件为参数，返回一个新的 React 组件为结果。在上面的例子中，connect 产生的高阶组件产生了一个新的 React 组件 Counter，这个 Counter 其实就是一个『聪明组件』，它负责管理状态，而 CounterView 是一个『傻瓜组件』，只负责渲染。

从上面可以看出，在 react-redux 中，应用了三个 React 模式：

1. 提供者模式
2. 高阶组件
3. 聪明组件和傻瓜组件的分离

Redux 和 React 结合的最佳实践

应用 Redux 的时候，有这些业界已经证明的最佳实践：

1. Store 上的数据应该范式化。

所谓范式化，就是尽量减少冗余信息，像设计 MySQL 这样的关系型数据库一样设计数据结构。

2. 使用 selector。

对于 React 组件，需要的是『反范式化』的数据，当从 Store 上读取数据得到的是范式化的数据时，需要通过计算来得到反范式化的数据。你可能会因此担心出现问题，这种担心不是没有道理，毕竟，如果每次渲染都要重复计算，这种浪费积少成多可能真会产生性能影响，所以，我们需要使用 seletor。业界应用最广的 selector 就是 reslector 。

reselector 的好处，是把反范式化分为两个步骤，第一个步骤是简单映射，第二个步骤是真正的重量级运算，如果第一个步骤发现产生的结果和上一次调用一样，那么第二个步骤也不用计算了，可以直接复用缓存的上次计算结果。

绝大部分实际场景中，总是只有少部分数据会频繁发生变化，所以 reselector 可以避免大量重复计算。

3. 只 connect 关键点的 React 组件

当 Store 上状态发生改变的时候，所有 connect 上这个 Store 的 React 组件会被通知：『状态改变了！』

然后，这些组件会进行计算。connect 的实现方式包含 `shouldComponentUpdate` 的实现，可以阻挡住大部分不必要的重新渲染，但是，毕竟处理通知也需要消耗 CPU，所以，尽量让关键的 React 组件 connect 到 store 就行。

一个实际的例子就是，一个列表种可能包含几百个项，让每一个项都去 connect 到 Store 上不是一个明智的设计，最好是只让列表去 connect，然后把数据通过 props 传递给各个项。

一个还是多个 Store

虽然理论上一个应用可以有任意多个 Store，但是按照官方的推荐，一个应用只应该有一个 Store。实际上，一切用了多个 Store 的应用，都可以改为用单个 Store 解决。

不过，我在一次技术咨询中，见过使用多个 Store 的应用，这个应用十分庞大，不只是组件多，参与的团队也多，而且地域和管理结构上都很分散，每个团队都在一个网页上贡献组件，为了避免互相踩到对方的脚，他们干脆就各自创建和管理各自的 Store，各自开发的组件也只把状态放在自己的 Store 上。

上面这种多个 Store 的方式当然行得通，不同团队之间需要共享数据。如果一个 React 组件需要访问多个 Store，情况就会比较复杂。

使用 react-redux 的话，虽然 Provider 可以嵌套，但是，最里层的 Provider 提供的 store 才生效。

在下面的代码示例中，Foo 能够 connect 到的 store 是s tore1，而 Bar 能够 connect 到的是 store2，因为内层的 Provider 会屏蔽掉外层的 Provider。

```
  <Provider store={store1}>
    <React.Fragment>
      <Foo />
      <Provider store={store2} >
        <React.Fragment>
          <Bar />
        </React.Fragment>
      </Provider>
   </React.Fragment>
  </Provider>
```

如果真的需要让 Bar 来访问到 store1，那么就不能通过 Provider 来传递，只能通过 props 等方式传递，如此一来，引入了新的复杂度。

所以，建议还是尽量使用一个 Store，如果真的需要多个 Store，除非认定只有很少组件会访问多个 Store。

如何实现异步操作

使用 Redux 对于同步状态更新非常顺手，但是，遇到需要异步更新状态的场景，例如调用 AJAX 从服务器获得数据，这时候单用 Redux 就不够了，需要其他方式来辅助。

至今为止，还无法推荐一个杀手级的方法，各种方法都在吹嘘自己多厉害，但是任何一种方法都是易用性和复杂性的平衡。

最简单的 redux-thunk，代码量少，只有几行，用起来也很直观，但是开发者要写很多代码；而比较复杂的 redux-observable 相当强大，可以只用少量代码就实现复杂功能，但是前提是你要学会 RxJS，RxJS 本身学习曲线很陡，内容需要 一本书 的篇幅来介绍，这就是代价。

读者在自己的项目中，无论选择什么方式，一定要考虑这个方式的复杂度和学习成本。

在这里我不想过多介绍任何一种 Redux 扩展，因为任何一种都比不上 React 将要支持的 Suspense，Suspense 才是 React 中做异步操作的未来，在第 19 小节会详细介绍 Suspense。

小结

这一小节我们介绍了 Redux，读者应该掌握：

1. Redux 中的基本概念 action、reducer 和 store；
2. 使用 react-redux 会应用哪些设计模式；
3. 如何设计 Redux 的 Store。

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

一直用的redux-thunk,直接复制样板代码...现在貌似redux-saga比较流行，不过看到这里决定去看看RxJS，看不懂的话就继续支持作者另一本书啦

▲ 0　评论　19天前

ITSheng

揪两个细节，第一，最后一个代码块最后少了一层</Provider>，第二，「小结」那里的markdown，#号后面少了一个空格，导致没有识别到目录结构

▲ 0　评论　1月前

Jexxie

"这里的 connect 函数接受两个参数，一个 mapStateToProps 是把 Store 上的 state 映射为 props；另一个 mapDispatchToProps 则是把回调函数类型的 props 映射为派发 action 的动作" mapDispatchToProps的作用是不是说反了?

▲ 0　收起评论　1月前

zhangyanling77 前端开发 @ 成都

并没有说反哈

1月前

评论审核通过后显示　评论

blacker

redux和react-redux源码分析来一波吧

▲ 0　收起评论　1月前

程墨 Hulu

如果你想要吃的是香肠，而且香肠的确好吃而且不贵，就没必要自己学着去做香肠。

1月前

Hack-Jay

他这话的意思是这源码很简单，让你自己网上查就行了

1月前

肖炎 前端开发 @ 今日头条

回复 Hack-Jay：他的意思是 会开车就行 不用学造车

1月前

kylin1993 FE

回复 肖炎：他的意思是自己做的香肠不好吃。

1月前

Da'Mn° 前端开发工程师

回复 肖炎：同意你的观点

1月前

1　2　下一页

评论审核通过后显示　评论