React.js 小书

<-- 返回首页

# 动手实现 Redux（四）：共享结构的对象提高性能


- 作者：胡子大哈
- 原文链接： http://huziketang.com/books/react/lesson33



（本文未审核）

接下来两节某些地方可能会稍微有一点点抽象，但是我会尽可能用简单的方式进行讲解。如果你觉得理解起来有点困难，可以把这几节多读多理解几遍，其实我们一路走来都是符合“逻辑”的，都是发现问题、思考问题、优化代码的过程。所以最好能够用心留意、思考我们每一个提出来的问题。

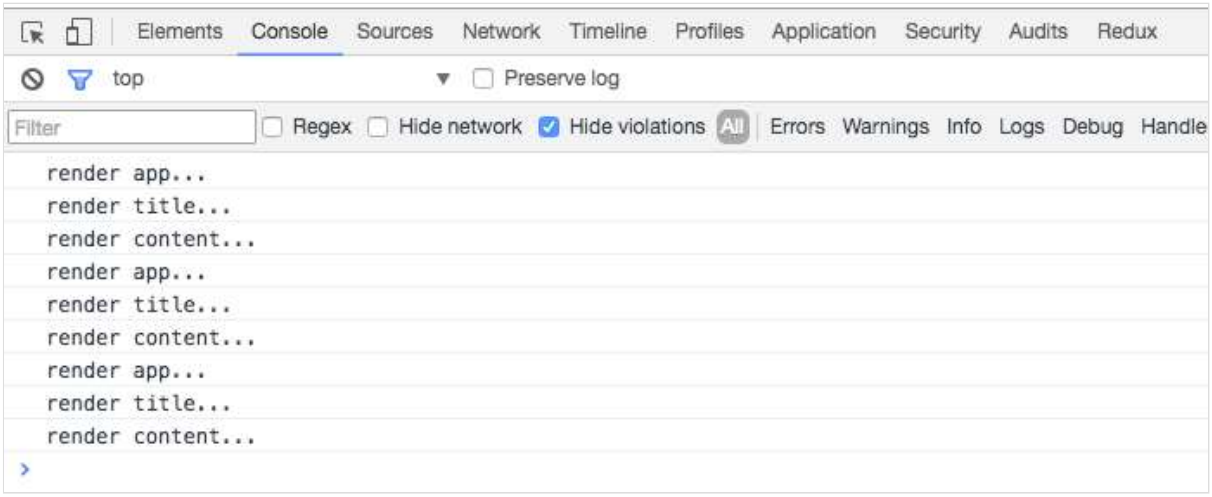
细心的朋友可以发现，其实我们之前的例子当中是有比较严重的**性能问题**的。我们在每个渲染函数的开头打一些 Log 看看：

```
function renderApp (appState) {
  console.log('render app...')
  renderTitle(appState.title)
  renderContent(appState.content)
}

function renderTitle (title) {
  console.log('render title...')
  const titleDOM = document.getElementById('title')
  titleDOM.innerHTML = title.text
  titleDOM.style.color = title.color
}

function renderContent (content) {
  console.log('render content...')
  const contentDOM = document.getElementById('content')
  contentDOM.innerHTML = content.text
  contentDOM.style.color = content.color
}
```

依旧执行一次初始化渲染，和两次更新，这里代码保持不变：

```
const store = createStore(appState, stateChanger)
store.subscribe(() => renderApp(store.getState())) // 监听数据变化

renderApp(store.getState()) // 首次渲染页面
store.dispatch({ type: 'UPDATE_TITLE_TEXT', text: '《React.js 小书》 ' }) // 修改
store.dispatch({ type: 'UPDATE_TITLE_COLOR', color: 'blue' }) // 修改标题颜色
```

可以在控制台看到：




前三个毫无疑问是第一次渲染打印出来的。中间三个是第一次 `store.dispatch` 导致的，最后三个是第二次 `store.dispatch` 导致的。可以看到问题就是，每当更新数据就重新渲染整个 App，但其实我们两次更新都没有动到 `appState` 里面的 `content` 字段的对象，而动的是 `title` 字段。其实并不需要重新 `renderContent`，它是一个多余的更新操作，现在我们需要优化它。

这里提出的解决方案是，在每个渲染函数执行渲染操作之前先做个判断，判断传入的新数据和旧的数据是不是相同，相同的话就不渲染了。

```
function renderApp (newAppState, oldAppState = {}) { // 防止 oldAppState 没有传入
  if (newAppState === oldAppState) return // 数据没有变化就不渲染了
  console.log('render app...')
  renderTitle(newAppState.title, oldAppState.title)
  renderContent(newAppState.content, oldAppState.content)
}

function renderTitle (newTitle, oldTitle = {}) {
  if (newTitle === oldTitle) return // 数据没有变化就不渲染了
  console.log('render title...')
  const titleDOM = document.getElementById('title')
  titleDOM.innerHTML = newTitle.text
  titleDOM.style.color = newTitle.color
}

function renderContent (newContent, oldContent = {}) {
  if (newContent === oldContent) return // 数据没有变化就不渲染了
  console.log('render content...')
  const contentDOM = document.getElementById('content')
  contentDOM.innerHTML = newContent.text
  contentDOM.style.color = newContent.color
}
```

然后我们用一个 `oldState` 变量保存旧的应用状态，在需要重新渲染的时候把新旧数据传进入去：

```
const store = createStore(appState, stateChanger)
let oldState = store.getState() // 缓存旧的 state
store.subscribe(() => {
  const newState = store.getState() // 数据可能变化，获取新的 state
  renderApp(newState, oldState) // 把新旧的 state 传进去渲染
  oldState = newState // 渲染完以后，新的 newState 变成了旧的 oldState，等待下一
})
...
```

希望到这里没有把大家忽悠到，**上面的代码根本不会达到我们的效果**。看看我们的 `stateChanger`：

```
function stateChanger (state, action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      state.title.text = action.text
      break
    case 'UPDATE_TITLE_COLOR':
      state.title.color = action.color
      break
    default:
      break
  }
}
```

即使你修改了 `state.title.text`，但是 `state` 还是原来那个 `state`，`state.title` 还是原来的 `state.title`，这些引用指向的还是原来的对象，只是对象内的内容发生了改变。所以即使你在每个渲染函数开头加了那个判断又什么用？这就像是下面的代码那样自欺欺人：

```
let appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  },
  content: {
    text: 'React.js 小书内容',
    color: 'blue'
  }
}
const oldState = appState
appState.title.text = '《React.js 小书》'
oldState !== appState // false，其实两个引用指向的是同一个对象，我们却希望它们不
```

但是，我们接下来就要让这种事情变成可能。

## 共享结构的对象

希望大家都知道这种 ES6 的语法：

```
const obj = { a: 1, b: 2}
const obj2 = { ...obj } // => { a: 1, b: 2 }
```

`const obj2 = { ...obj }` 其实就是新建一个对象 `obj2`，然后把 `obj` 所有的属性都复制到 `obj2` 里面，相当于对象的浅复制。上面的 `obj` 里面的内容和 `obj2` 是完全一样的，但是却是两个不同的对象。除了浅复制对象，还可以覆盖、拓展对象属性：

```
const obj = { a: 1, b: 2}
const obj2 = { ...obj, b: 3, c: 4} // => { a: 1, b: 3, c: 4 }，覆盖了 b，新增了
```

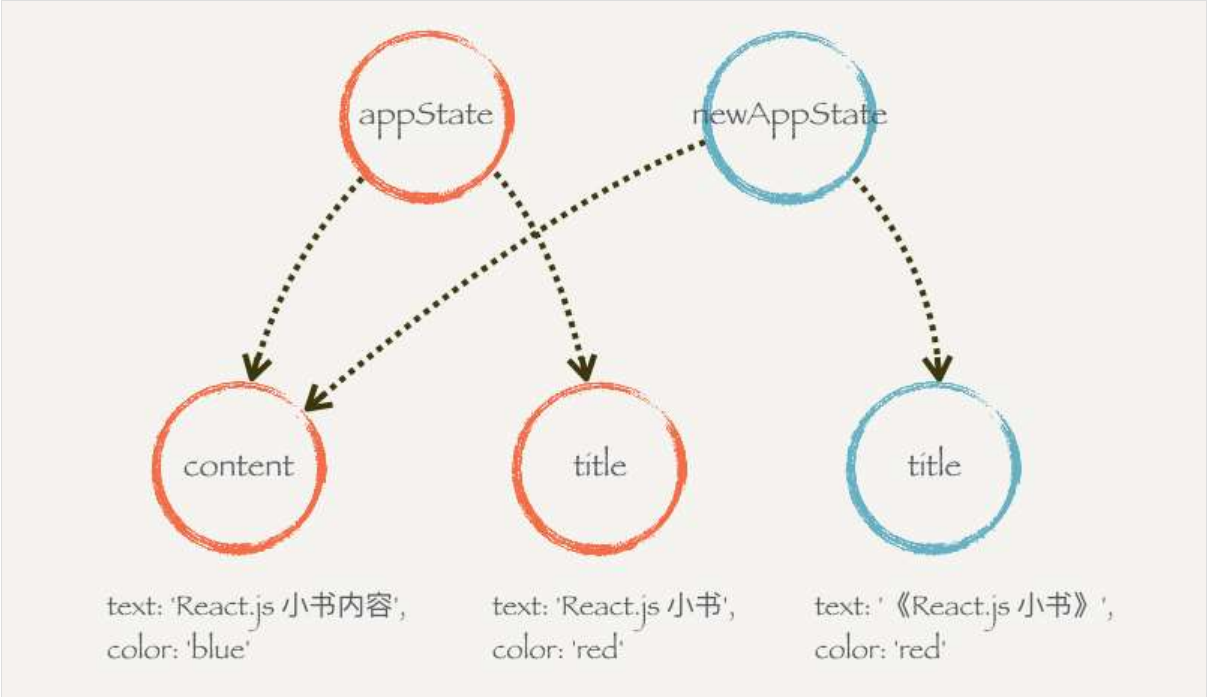
我们可以把这种特性应用在 `state` 的更新上，我们禁止直接修改原来的对象，一旦你要修改某些东西，你就得把修改路径上的所有对象复制一遍，例如，我们不写下面的修改代码：

```
appState.title.text = '《React.js 小书》'
```

取而代之的是，我们新建一个 `appState`，新建 `appState.title`，新建 `appState.title.text`：

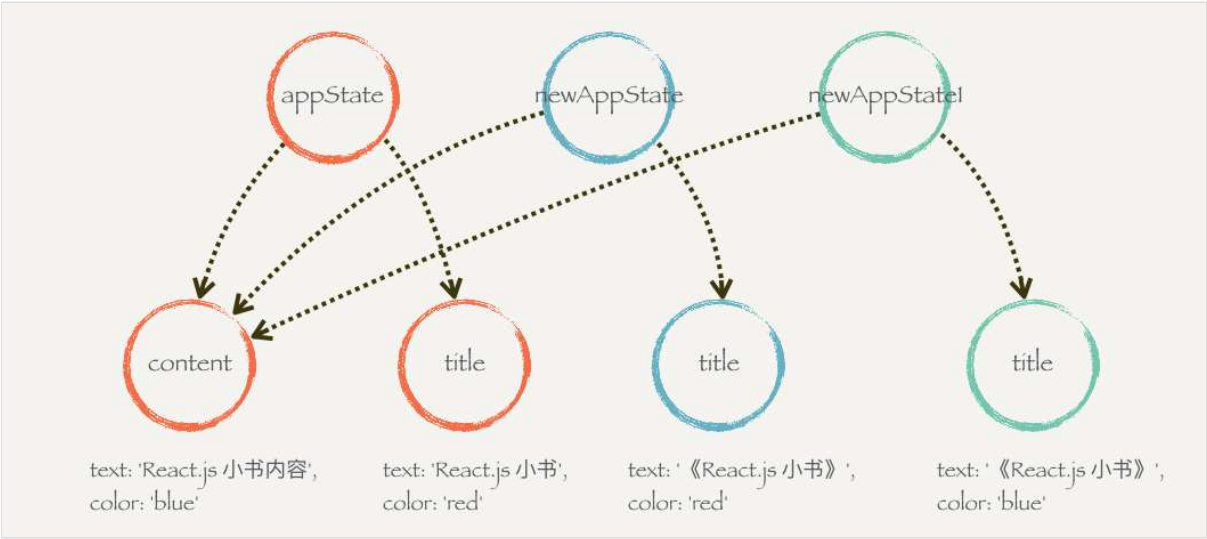
```
let newAppState = { // 新建一个 newAppState
  ...appState, // 复制 appState 里面的内容
  title: { // 用一个新的对象覆盖原来的 title 属性
    ...appState.title, // 复制原来 title 对象里面的内容
    text: '《React.js 小书》' // 覆盖 text 属性
  }
}
```

如果我们用一个树状的结构来表示对象结构的话：




`appState` 和 `newAppState` 其实是两个不同的对象，因为对象浅复制的缘故，其实它们里面的属性 `content` 指向的是同一个对象；但是因为 `title` 被一个新的对象覆盖了，所以它们的 `title` 属性指向的对象是不同的。同样地，修改 `appState.title.color`：

```
let newAppState1 = { // 新建一个 newAppState1
  ...newAppState, // 复制 newAppState1 里面的内容
  title: { // 用一个新的对象覆盖原来的 title 属性
    ...newAppState.title, // 复制原来 title 对象里面的内容
    color: "blue" // 覆盖 color 属性
  }
}
```




我们每次修改某些数据的时候，都不会碰原来的数据，而是把需要修改数据路径上的对象都 copy 一个出来。这样有什么好处？看看我们的目的达到了：

```
appState !== newAppState // true，两个对象引用不同，数据变化了，重新渲染
appState.title !== newAppState.title // true，两个对象引用不同，数据变化了，重新
appState.content !== appState.content // false，两个对象引用相同，数据没有变化，
```

修改数据的时候就把修改路径都复制一遍，但是保持其他内容不变，最后的所有对象具有某些不变共享的结构（例如上面三个对象都共享 `content` 对象）。大多数情况下我们可以保持 50% 以上的内容具有共享结构，这种操作具有非常优良的特性，我们可以用它来优化上面的渲染性能。

## 优化性能

我们修改 `stateChanger`，让它修改数据的时候，并不会直接修改原来的数据 `state`，而是产生上述的共享结构的对象：

```
function stateChanger (state, action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      return { // 构建新的对象并且返回
        ...state,
        title: {
          ...state.title,
          text: action.text
        }
      }
    case 'UPDATE_TITLE_COLOR':
      return { // 构建新的对象并且返回
        ...state,
        title: {
          ...state.title,
          color: action.color
        }
      }
    default:
      return state // 没有修改，返回原来的对象
  }
}
```

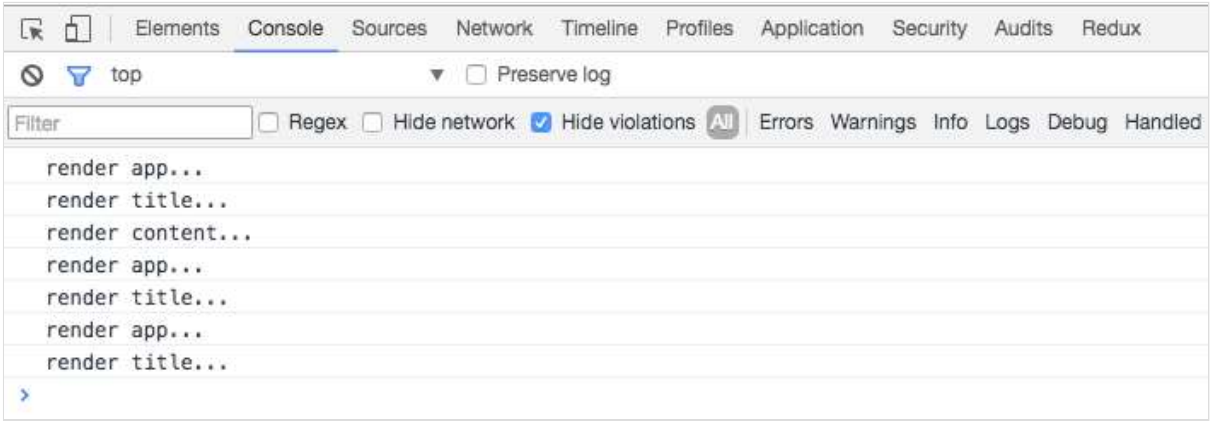
代码稍微比原来长了一点，但是是值得的。每次需要修改的时候都会产生新的对象，并且返回。而如果没有修改（在 `default` 语句中）则返回原来的 `state` 对象。

因为 `stateChanger` 不会修改原来对象了，而是返回对象，所以我们需要修改一下 `createStore`。让它用每次 `stateChanger(state, action)` 的调用结果覆盖原来的 `state`：

```
function createStore (state, stateChanger) {
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
```

```
  const getState = () => state
  const dispatch = (action) => {
    state = stateChanger(state, action) // 覆盖原对象
    listeners.forEach((listener) => listener())
  }
  return { getState, dispatch, subscribe }
}
```

保持上面的渲染函数开头的对象判断不变，再看看控制台：




前三个是首次渲染。后面的 `store.dispatch` 导致的重新渲染都没有关于 `content` 的 Log 了。因为产生共享结构的对象，新旧对象的 `content` 引用指向的对象是一样的，所以触发了 `renderContent` 函数开头的：

```
...
  if (newContent === oldContent) return
...
```

我们成功地把不必要的页面渲染优化掉了，问题解决。另外，并不需要担心每次修改都新建共享结构对象会有性能、内存问题，因为构建对象的成本非常低，而且我们最多保存两个对象引用（`oldState` 和 `newState`），其余旧的对象都会被垃圾回收掉。

本节完整代码：

```
function createStore (state, stateChanger) {
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
  const getState = () => state
  const dispatch = (action) => {
    state = stateChanger(state, action) // 覆盖原对象
    listeners.forEach((listener) => listener())
  }
  return { getState, dispatch, subscribe }
}

function renderApp (newAppState, oldAppState = {}) { // 防止 oldAppState 没有传入
  if (newAppState === oldAppState) return // 数据没有变化就不渲染了
  console.log('render app...')
  renderTitle(newAppState.title, oldAppState.title)
```

```
  renderContent(newAppState.content, oldAppState.content)
}

function renderTitle (newTitle, oldTitle = {}) {
  if (newTitle === oldTitle) return // 数据没有变化就不渲染了
  console.log('render title...')
  const titleDOM = document.getElementById('title')
  titleDOM.innerHTML = newTitle.text
  titleDOM.style.color = newTitle.color
}

function renderContent (newContent, oldContent = {}) {
  if (newContent === oldContent) return // 数据没有变化就不渲染了
  console.log('render content...')
  const contentDOM = document.getElementById('content')
  contentDOM.innerHTML = newContent.text
  contentDOM.style.color = newContent.color
}

let appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  },
  content: {
    text: 'React.js 小书内容',
    color: 'blue'
  }
}

function stateChanger (state, action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      return { // 构建新的对象并且返回
        ...state,
        title: {
          ...state.title,
          text: action.text
        }
      }
    case 'UPDATE_TITLE_COLOR':
      return { // 构建新的对象并且返回
        ...state,
        title: {
          ...state.title,
          color: action.color
        }
      }
    default:
      return state // 没有修改，返回原来的对象
  }
}

const store = createStore(appState, stateChanger)
let oldState = store.getState() // 缓存旧的 state
```

```
store.subscribe(() => {
  const newState = store.getState() // 数据可能变化，获取新的 state
  renderApp(newState, oldState) // 把新旧的 state 传进去渲染
  oldState = newState // 渲染完以后，新的 newState 变成了旧的 oldState，等待下一
})

renderApp(store.getState()) // 首次渲染页面
store.dispatch({ type: 'UPDATE_TITLE_TEXT', text: '《React.js 小书》' }) // 修改
store.dispatch({ type: 'UPDATE_TITLE_COLOR', color: 'blue' }) // 修改标题颜色
```

> 因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 React.js 小书的论坛 发帖，我会回答大家的疑问。

下一节：动手实现 Redux（五）：不要问为什么的 reducer

上一节：动手实现 Redux（三）：纯函数（Pure Function）简介

## 2 条评论，2 人参与。

0

我有话说...

使用社交帐号登录

发布前先点击左边的按钮登录

### 最新评论

**de** • 6月27日 10:10

...不支持对象吧

顶 • 回复 • 分享»

门双人走 de • 7月5日 12:46

支持的 浏览器 现在应该还不支持 但是在 create-react-app 搭建的环境里是可以正常使用的

顶 • 回复 • 分享»

友言?