React.js 小书

<-- 返回首页

动手实现 Redux（五）：不要问为什么的 reducer

- 作者：胡子大哈

- 原文链接： http://huziketang.com/books/react/lesson34

（本文未审核）

经过了这么多节的优化，我们有了一个很通用的 `createStore`：

```
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 }
}
```

它的使用方式是：

```
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 {
        ...state,
        title: {
          ...state.title,
          text: action.text
        }
      }
    case 'UPDATE_TITLE_COLOR':
```

```
      return {
        ...state,
        title: {
          ...state.title,
          color: action.color
        }
      }
    default:
      return state
  }
}

const store = createStore(appState, stateChanger)
...
```

我们再优化一下，其实 `appState` 和 `stateChanger` 可以合并到一起去：

```
function stateChanger (state, action) {
  if (!state) {
    return {
      title: {
        text: 'React.js 小书',
        color: 'red',
      },
      content: {
        text: 'React.js 小书内容',
        color: 'blue'
      }
    }
  }
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      return {
        ...state,
        title: {
          ...state.title,
          text: action.text
        }
      }
    case 'UPDATE_TITLE_COLOR':
      return {
        ...state,
        title: {
          ...state.title,
          color: action.color
        }
      }
    default:
      return state
  }
}
```

`stateChanger` 现在既充当了获取初始化数据的功能，也充当了生成更新数据的功能。如果有传入 `state` 就生成更新数据，否则就是初始化数据。这样我们可以优化 `createStore` 成一个参数，因为 `state` 和 `stateChanger` 合并到一起了：

```
function createStore (stateChanger) {
  let state = null
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
  const getState = () => state
  const dispatch = (action) => {
    state = stateChanger(state, action)
    listeners.forEach((listener) => listener())
  }
  dispatch({}) // 初始化 state
  return { getState, dispatch, subscribe }
}
```

`createStore` 内部的 `state` 不再通过参数传入，而是一个局部变量 `let state = null`。`createStore` 的最后会手动调用一次 `dispatch({})`，`dispatch` 内部会调用 `stateChanger`，这时候的 `state` 是 `null`，所以这次的 `dispatch` 其实就是初始化数据了。`createStore` 内部第一次的 `dispatch` 导致 `state` 初始化完成，后续外部的 `dispatch` 就是修改数据的行为了。

我们给 `stateChanger` 这个玩意起一个通用的名字：reducer，不要问为什么，它就是个名字而已，修改 `createStore` 的参数名字：

```
function createStore (reducer) {
  let state = null
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
  const getState = () => state
  const dispatch = (action) => {
    state = reducer(state, action)
    listeners.forEach((listener) => listener())
  }
  dispatch({}) // 初始化 state
  return { getState, dispatch, subscribe }
}
```

这是一个最终形态的 `createStore`，它接受的参数叫 `reducer`，`reducer` 是一个函数，细心的朋友会发现，它其实是一个纯函数（Pure Function）。

reducer

`createStore` 接受一个叫 reducer 的函数作为参数，**这个函数规定是一个纯函数**，它接受两个参数，一个是 `state`，一个是 `action`。

如果没有传入 `state` 或者 `state` 是 `null`，那么它就会返回一个初始化的数据。如果有传入 `state` 的话，就会根据 `action` 来“修改“数据，但其实它没有、也规定不能修改 `state`，而是要通过上节所说的把修改路径的对象都复制一遍，然后产生一个新的对象返回。如果它不能识别你的 `action`，它就不会产生新的数据，而是（在 `default` 内部）把 `state` 原封不动地返回。

reducer 是不允许有副作用的。你不能在里面操作 DOM，也不能发 Ajax 请求，更不能直接修改 `state`，它要做的仅仅是 —— **初始化和计算新的 `state`**。

现在我们可以用这个 `createStore` 来构建不同的 `store` 了，只要给它传入符合上述的定义的 `reducer` 即可：

```
function themeReducer (state, action) {
  if (!state) return {
    themeName: 'Red Theme',
    themeColor: 'red'
  }
  switch (action.type) {
    case 'UPATE_THEME_NAME':
      return { ...state, themeName: action.themeName }
    case 'UPATE_THEME_COLOR':
      return { ...state, themeColor: action.themeColor }
    default:
      return state
  }
}

const store = createStore(themeReducer)
...
```

> 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（六）：Redux 总结

上一节：动手实现 Redux（四）：共享结构的对象提高性能