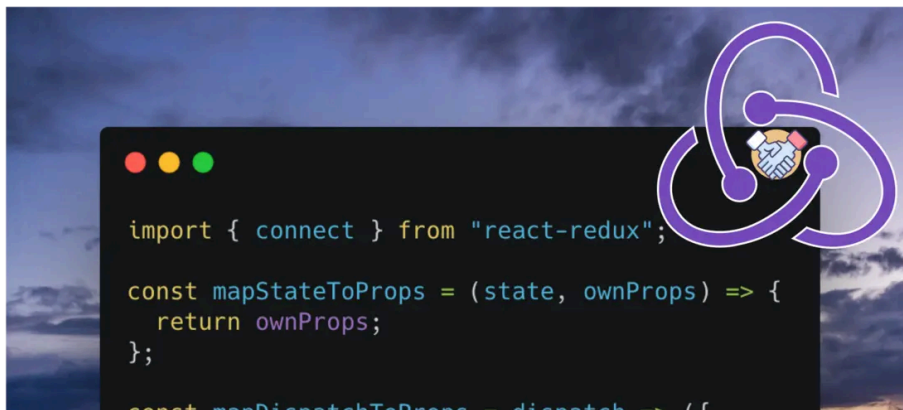


# Redux 包教包会（一）：解救 React 状态危机



一只图雀

2020-03-12 阅读 16 分钟



前端应用的状态管理日益复杂。随着大前端时代的到来，前端愈来愈注重处理逻辑，而不只是专注 UI 层面的改进，而以 React 为代表的前端框架的出现，大大简化了我们编写 UI 界面的复杂度。虽然 React 提供了 State 机制实现状态管理，也有诸如“状态提升”等开发约定，但是这些方案只适用于小型应用，当你的前端应用有多达 10 个以上页面时，如何让应用状态可控、让协作开发高效成为了亟待解决的问题，而 Redux 的出现正是为了解决这些问题而生的！Redux 提出的“数据的唯一真相来源”、单向数据流、“纯函数 Reducers”大大简化了前端逻辑，使得我们能够以高效、便于协作的方式编写任意复杂的前端应用。本篇教程致力于用简短的文字讲透 Redux，在实践中掌握 Redux 的概念和精髓。

欢迎阅读 Redux 包教包会系列：

- Redux 包教包会（一）：解救 React 状态危机（也就是这篇）
- Redux 包教包会（二）：趁热打铁，完全重构
- Redux 包教包会（三）：各司其职，重拾初心

此教程属于[React 前端工程师学习路线](#)的一部分，欢迎来 Star 一波，鼓励我们继续创作出更好的教程，持续更新中~。

## 在我们阅读教程之前

Redux 官方文档对 Redux 的定义是：**一个可预测的 JavaScript 应用状态管理容器。**

这就意味着，Redux 是无法单独运作的，它需要与一个具体的 View 层的前端框架相结合才能发挥出它的威力，这里的 View 层包括但不限于 React、Vue 或者 Angular 等。这里我们将使用 React 作为绑定视图层，因为 Redux 最初诞生于 React 社区，为解决 React 的状态管理问题而设计。

图雀社区

注册登录

主页 关于 RSS

近来 React Hooks 确实很火，展现出惊人的潜力，甚至有人声称可以抛弃 Redux 了。其实笔者觉得这种说法不完全正确，Redux 的成功其实不仅仅是因为这个框架本身，还因为围绕其构建起来的生态系统，比如 Enhancers、Middlewares、还有诸如 redux-form, redux-immutable 等，甚至还有基于 Redux 的上层框架，如 Dva 还有 Rematch，这些都为 Redux 巩固了在 React 社区的王者地位。而有趣的是，我们注意到 Redux 的 React 绑定库 react-redux 现在正在用 React Hooks 重构，以求让代码更加精炼和高效，所以笔者觉得 React Hooks 首先还处于萌芽阶段，一小部分尝鲜者在视图使用它来构建更好的 React 项目或者框架，React Hooks 可以让之前的这些项目和框架变得更好，以更好的辅助 Redux 生态的继续繁荣，所以我们有理由相信，React Hooks 的出现，会让 React 社区变得更加高效和专业，也会帮助 Redux 工具链变得更加轻量，最终作为 React 的一个优秀的特性将 React 和其生态带向更好的远方。

## 前提条件

本篇教程是关于 Redux 的快速入门教程，并致力于讲解与 React 绑定时的使用，而了解和掌握 Redux 对于一个 React 开发者来说属于较为进阶的内容，所以我们假设在阅读本篇教程之前，你需要拥有以下的知识储备：

- 对 ES6 的函数、类、`const`、对象解构、函数默认参数等概念有良好的了解，当然如果你了解过函数式编程，如纯函数、不变性等就更好了
- 对 React 有良好的了解，当然如果有独立开发过至少有 5 个页面的 React 应用的经验就更好了，可以参考这篇[入门教程](#)进行学习
- 了解 Node 和 npm，有过相关的安装依赖的经验即可，可以参考这篇[教程](#)进行学习

## 你将学到什么

在本篇教程中，我们将首先给出了一个[使用 React 实现的待办事项小应用](#)（比[上篇教程](#)中完成的版本多了筛选的功能），它将是我们的学习 Redux 的起点，当你熟悉了这份初始代码，并了解了它的功能之后，你就可以关闭它，然后开始我们教程的学习啦！

我们将基于这个纯 React 写成的模板，分析 React 在处理状态时存在的问题，以及用 Redux 重构带来的优势。接着我们将通过实战的方式学习如何将一个纯 React 应用一步步地重构成一个 Redux 应用，最终实现一个升级版的待办事项小应用。

## 代码和最终效果

本教程所实现的源代码都托管在 Github 上：

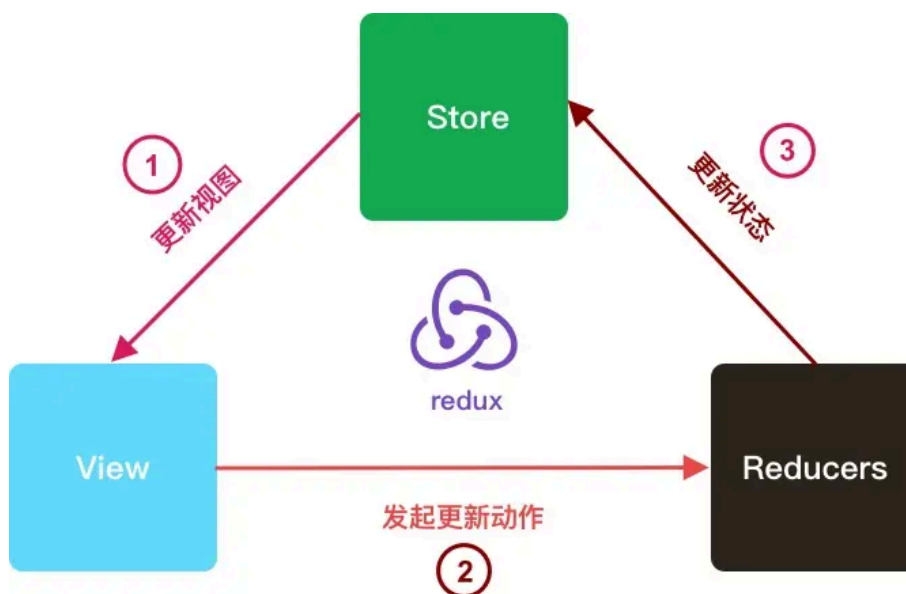
- 纯 React 源码：[源码地址](#)。
- 使用 Redux 重构后的源码：[源码地址](#)。

你可以通过 CodeSandbox 查看代码最终的效果：

- 纯 React 效果：[最终效果地址](#)。
- 使用 Redux 重构后的效果：[最后效果地址](#)。

## 开始 Redux 之旅

不管外界把 Redux 吹得如何天花乱坠，实际上它可以用一张图来概括，这张图也有利于帮助你思考前端的本质是什么：



我们先来详解一下这张图，并且在教程之后的内容中，你会多次看到这张图以不同的形式出现。我们希望学完本篇教程之后，每当你想起 Redux

时，脑海里就是上面这张图。

## View

首先我们来看 View，在前端开发中，我们称这个为**视图层**，就是展示给最终用户的效果，在本篇教程的学习中，我们的 View 就是 React。

## Store

随着前端应用要完成的工作越来越丰富，我们对前端也提出了要保持“状态”的要求。在 React 中，这个“状态”将保存在 `this.state`。在 Redux 中，这个状态将保存在 Store。

这个 Store 从抽象意义上来说可以看做一个前端的“数据库”，它保存着前端的状态（state），并且分发这些状态给 View，使得 View 根据这些状态渲染不同的内容。

注意到，Redux 是一个可预测的 JavaScript 应用状态管理容器，这个**状态容器**就是这里的 Store。

## Reducers

我们日常生活中看到的网页，它不是一成不变的，而是会响应用户的“动作”，无论是页面跳转还是登陆注册，这些动作会改变当前应用的状态。

在 Redux 框架中，Reducers 的作用就是响应不同的动作。更精确地说，Reducers 是**负责更新 Store 中状态的 JavaScript 函数**。

当我们对这三个核心概念有了粗略的认知之后，就可以开始 Redux 的学习了。

## 准备初始代码

将初始 React 代码模板 Clone 到本地，进入仓库，并切换到 initial-code 分支（初始代码模板）：

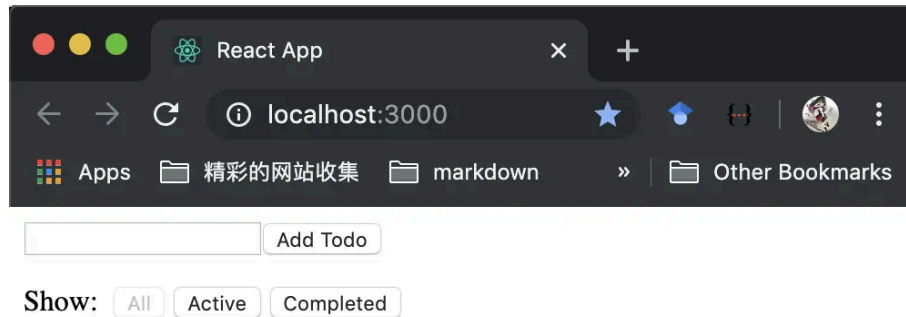
```
git clone https://github.com/pftom/redux-quickstart-tutorial.git
cd redux-quickstart-tutorial
git checkout initial-code
```



安装项目依赖，并打开开发服务器：

```
npm install  
npm start
```

接着 React 开发服务器会打开浏览器，如果你看到下面的效果，并且可以进行操作，那么代表代码准备完成：

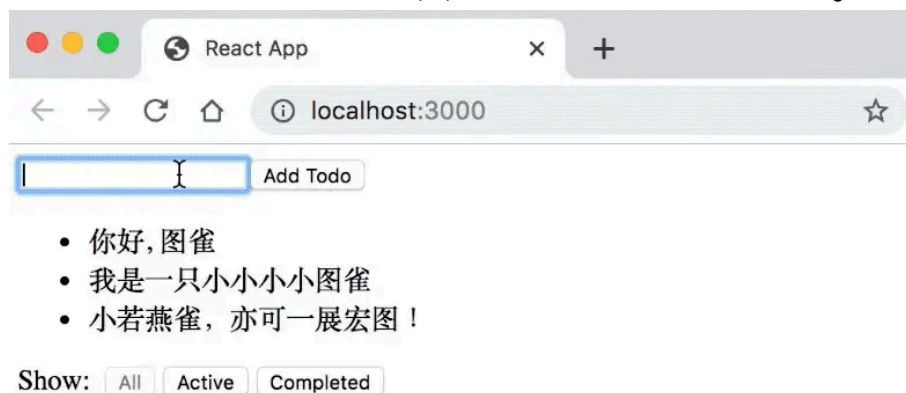


### 提示

由于我们使用 [Create React App](#) 脚手架，它使用 Webpack Development Server (WDS) 作为开发服务器，因此在后面编辑代码的时候只需保存文件，我们的 React 应用就会自动刷新，非常方便。

## 探索初始代码

我们完成的这个待办事项小应用比[上篇教程](#)中实现的要高级一点，如下面这个动图所示：



我们希望展示一个 todo 列表，当一个 todo 被点击时，它将被加上删除线表示此 todo 已经完成，我们还加上了一个输入框，使得用户可以增加新的 todo。在底部，我们展示了三个按钮，可以切换展示 todo 的类型。

整份 React 代码组件设计如下（首先是组件，然后是组件所拥有的属性）：

- `TodoList` 用来展示 todo 列表：
  - `todos: Array` 是一个 todo 数组，它其中的每个元素的样子类似 `{ id, text, completed }`。
  - `toggleTodo(id: number)` 是当一个 todo 被点击时会调用的回调函数。
- `Todo` 是单一 todo 组件：
  - `text: string` 是这个 todo 将显示的内容。
  - `completed: boolean` 用来表示是否完成，如果完成，那么样式上就会给这个元素划上删除线。
  - `onClick()` 是当这个 todo 被点击时将调用的回调函数。
- `Link` 是一个展示过滤的按钮：
  - `active: boolean` 代表此时被选中，那么此按钮将不能被点击
  - `onClick()` 表示这个 link 被点击时将调用的回调函数。
  - `children: ReactComponent` 展示子组件
- `Footer` 用于展示三个过滤按钮：
  - `filter: string` 代表此时的被选中的过滤器字符串，它是 `[SHOW_ALL, SHOW_COMPLETED, SHOW_ACTIVE]` 其中之一。
  - `setVisibilityFilter()` 代表 Link 被点击时将设置对应被点击的 `filter` 的回调函数。
- `App` 是 React 根组件，最终组合其他组件并使用 ReactDOM 对其进行编译渲染，我们在它的 `state` 上定义了上面的几个组件会用到的属

性，同时定义其他组件会用到的方法，还有 `nextTodoId`，`VisibilityFilters`，`getVisibleTodos` 等一些辅助函数。

## 准备 Redux 环境

我们知道 Redux 可以与多种视图层开发框架如 React，Vue 和 Angular 等搭配使用，而 Redux 只是一个状态管理容器，所以为了在 React 中使用 Redux，我们还需要安装一下对应的依赖。

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

做得好！现在一切已经准备就绪，相信你已经迫不及待的想要编写一点 Redux 相关的代码了，别担心，在下一节中，我们将引出 Redux Store 的详细概念，并且通过代码讲解它将替换 React 的哪个部分。

## 理解 Store: 数据的唯一真相来源

我们前面提到了 Store 在 Redux 中的作用是用来保存状态的，相当于我们在前端建立了一个简单的“数据库”。在目前的富状态前端应用中，如果每一次状态的修改（例如点击一个按钮）都需要与后端通信，那么整个网站的平均响应时间将变得难以接受，用户体验将糟糕透顶。

根据不完全统计：“一个网站能留住一名用户的时间只有 8S，如果你在 8S 内不能吸引住用户，或者网站出现了问题，那么你将彻底地丢失这名用户！”

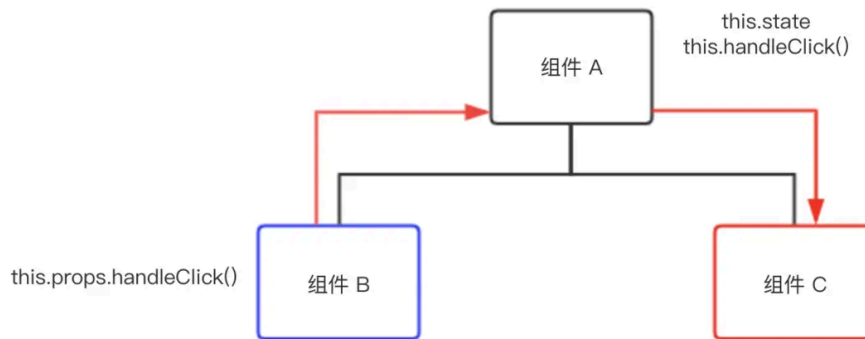
所以为了适应用户的访问需求，聪明的前端拓荒者们开始将后端的“数据库”理念引入到前端中，这样大多数的前端状态可以直接在前端搞定，完全不需要后端的介入。

## React 状态“危机”

在 React 中，我们将状态存在每个组件的 `this.state` 中，每个组件的 `state` 为组件所私有，如果要在一个组件中操作另外一个组件，实现起来是相当繁琐的。

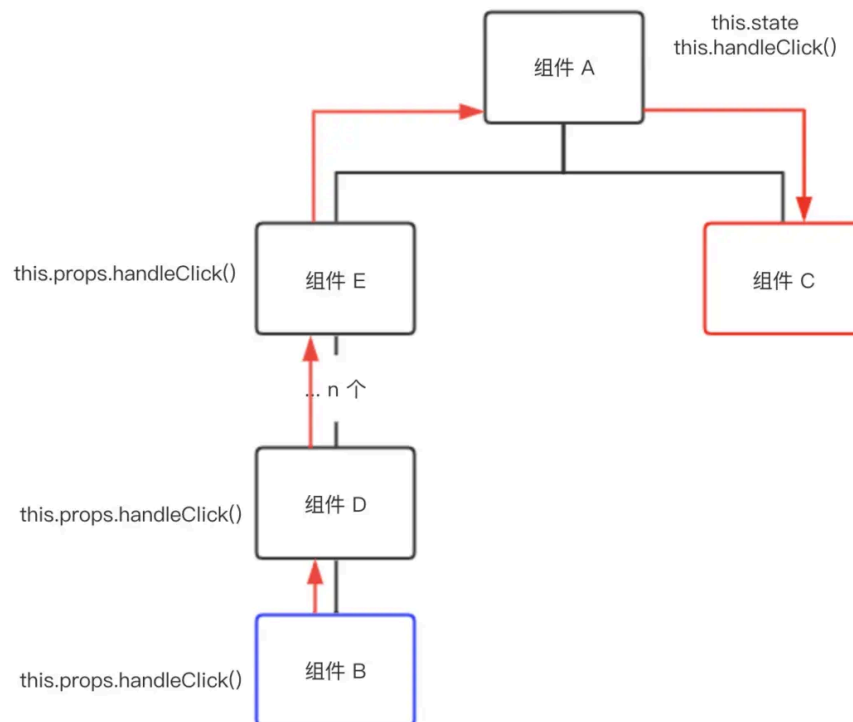
我们将用下面这张图来演示一下为什么繁琐：





组件 A 是组件 B 和 C 的父组件。如果组件 B 想要操作组件 C，那么它首先需要调用父组件 A 传给它的 `handleClick` 方法，然后通过这个方法修改父组件 A 的 `state`，进而通过 React 的自动重新渲染机制，触发组件 C 的变化。

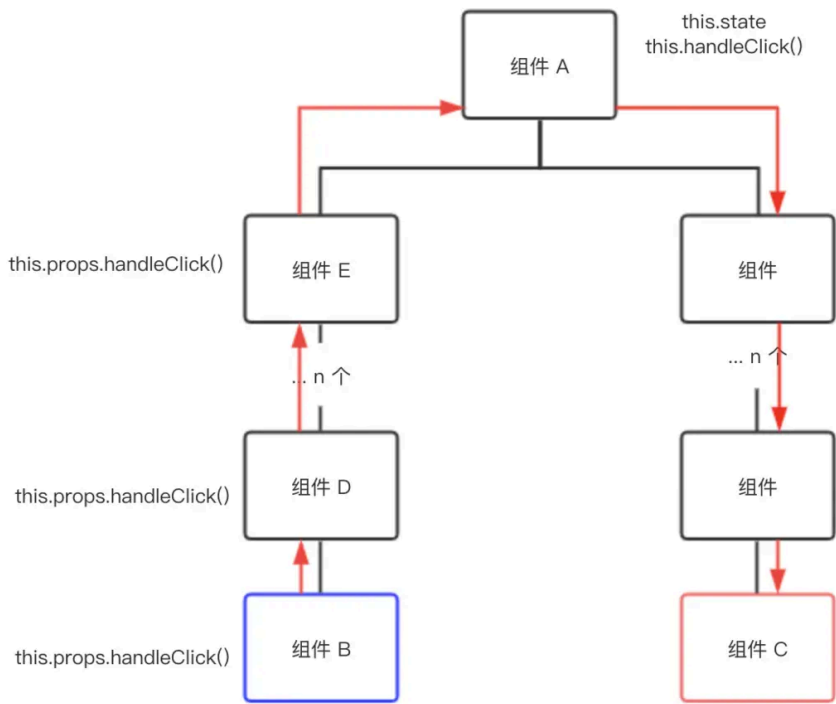
现在组件 B 和组件 C 是处于平级的，你可能还感觉不到这种跨组件改变有什么问题，让我们再来看一张图：



我们看到上面这张图，组件 B 和组件 C 相差了很多级，图中的 `n` 可能为 10，也可能更多。这个时候如果再想在组件 B 中修改组件 C，那就要把这个 `handleClick` 方法一层一层地往下传。每次要修改的时候，都要进行调用，这已经相当繁琐了。

如果组件 C 离组件 A 还有很深的层级，情况就更复杂了：



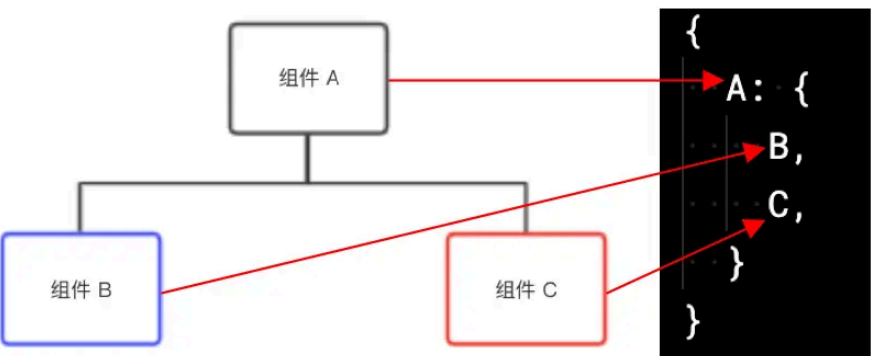


这时候，不仅要把 `handleClick` 方法通过很深的层级传给组件 B，当组件 B 调用 `handleClick` 方法时，修改组件 A 的 `state`，再反过来传递给组件 C 时，组件 A 到组件 C 之间的所有组件都会触发重新渲染，这带来了巨额的渲染开销，当我们的应用越来越复杂，这种开销显然是承受不起的。

## 解救者：Store

React 诞生的初衷就是为了更好、更高效地编写用户界面，它不应该也不需要来承担状态管理的职责。

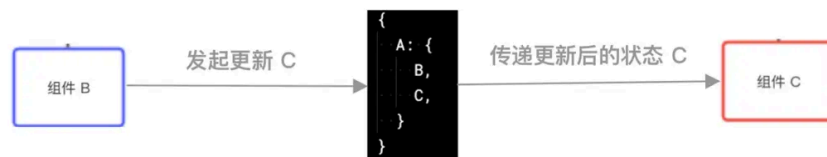
于是备受折磨的前端拓荒者们构想出了伟大的 Store。我们完全不需要让每个组件单独保持状态，直接抽离所有组件的状态，类比 React 组件树，构造一个**中心化的状态树**，这棵**状态树**与 React **组件树**一一对应，相当于对 React 组件树进行了状态化建模：



可以看到，我们将组件的 `state` 去掉，取而代之的是一棵状态树，它是一个普通的 JavaScript 对象。通过对象的嵌套来类比组件的嵌套组合，这棵

由 JavaScript 对象建模的状态树就是 Redux 中的 Store。

当我们将组件的状态抽离出去之后，我们在使用组件 B 操作组件 C 就变得相当简单且高效。



我们在组件 B 中发起一个更新状态 C 的动作，此动作对应的更新函数更新 Store 状态树，之后将更新后的状态 C 传递给组件 C，触发组件 C 的重新渲染。

可以看到，当我们引入这种机制之后，组件 B 与组件 C 之间的交互就能够单独进行，不会影响 React 组件树中的其他组件，也不需要传递很深层级的 `handleClick` 函数了，再也不需要把更新后的 `state` 一层一层地传给组件 C，性能有了质的飞跃。

有了 Redux Store 之后，所有 React 应用中的状态修改都是对这棵 JavaScript 对象树的修改，所有状态的获取都是从此棵 JavaScript 对象树获取，这棵 JavaScript 对象代表的状态树成了整个应用的“数据的唯一真相来源”。

## 打湿你的双手

了解了 Redux Store 之于 React 的作用之后，我们马上在 React 中应用 Redux，看看神奇的 Store 是如何介入并产生如此大的变化的。

我们修改初始代码模板中的 `src/index.js`，修改后的代码如下：

```
import React from "react";
import ReactDOM from "react-dom";
import App, { VisibilityFilters } from "../components/App";

import { createStore } from "redux";
import { Provider } from "react-redux";

const initialState = {
  todos: [
    {
      id: 1,
      text: "你好，图雀",
      completed: false
    },
    {
      id: 2,
```

```

    text: "我是一只小小小小图雀",
    completed: false
  },
  {
    id: 3,
    text: "小若燕雀，亦可一展宏图！",
    completed: false
  }
],
filter: VisibilityFilters.SHOW_ALL

```

可以看到，上面的代码做了下面几项工作：

- 我们首先进行了导包操作，从 `redux` 中导出了 `createStore`，从 `react-redux` 导出了 `Provider`，从 `src/components/App.js` 中导出了 `VisibilityFilters`。
- 接着我们定义了一个 `initialState` 对象，这将作为我们之后创建 Store 的初始状态数据，也是我们之前提到的那棵 JavaScript 对象树的初始值。
- 然后我们定义了一个 `rootReducer` 函数，它是一个箭头函数，接收 `state` 和 `action` 然后返回 `state`，这个函数目前还没有完成任何工作，但是它是创建 Store 所必须的参数之一，我们将在之后的 Reducers 中详细讲解它。
- 再接着，我们调用之前导出的 Redux API：`createStore` 函数，传入定义的 `rootReducer` 和 `initialState`，生成了我们本节的主角：`store`！
- 最后我们在 `App` 组件的最外层使用 `Provider` 包裹，并接收我们上一步创建的 `store` 作为参数，这确保之后我们可以在子组件中访问到 `store` 中的状态。`Provider` 是 `react-redux` 提供的 API，是 Redux 在 React 使用的绑定库，它搭建起 Redux 和 React 交流的桥梁。

现在我们已经创建了 Store，并使用了 React 与 Redux 的绑定库 `react-redux` 提供的 `Provider` 组件将 Store 与 React 组件组合在了一起。我们马上来看一下整合 Store 与 React 之后的效果。

打开 `src/components/App.js`，修改代码如下：

```

import React from "react";
import AddTodo from "../AddTodo";
import TodoList from "../TodoList";
import Footer from "../Footer";

import { connect } from "react-redux";

// 省略了 VisibilityFilters 和 getVisibleTodos 函数...

class App extends React.Component {

```

```

constructor(props) {
  super(props);

  this.toggleTodo = this.toggleTodo.bind(this);
  this.onSubmit = this.onSubmit.bind(this);
  this.setVisibilityFilter = this.setVisibilityFilter.bind(this);
}

// 省略中间其他方法...

render() {
  const { todos, filter } = this.props;

  return (
    <div>

```

可以看到，上面的代码做了这几项工作：

- 首先我们从 `react-redux` 绑定库里面导出了 `connect` 函数。
- 然后在文件底部，我们定义了一个 `mapStateToProps` 箭头函数，它接收 `state` 和 `props`，这个 `state` 就是我们那棵 Store 里面保存的 JavaScript 对象状态树，目前就是我们在上一个文件中定义的 `initialState` 内容；这个 `props` 就是我们熟悉的原 React 组件的 `props`，它对于 `mapStateToProps` 是一个可选参数。 `mapStateToProps` 返回的 `{ todos, filter }` 就是最终的组件 `props`，（当然这里我们并没有使用原组件 `props` 内容）并通过 `connect` 函数传递给 `App` 组件。
- `connect` 函数接收 `mapStateToProps` 函数，获取 `mapStateToProps` 返回的最终组合后的状态，然后将其注入到 `App` 组件中，返回一个新的组件，然后交给 `export default` 导出。
- 经过上面的工作，我们在 `App` 组件中就可以取到通过 `mapStateToProps` 返回的 `{ todos, filter }` 内容了，我们通过对象解构，从 `this.props` 拿到 `todos` 和 `filter` 属性。
- 最后我们删除不再需要的 `constructor` 中的 `this.state` 内容。

### 注意

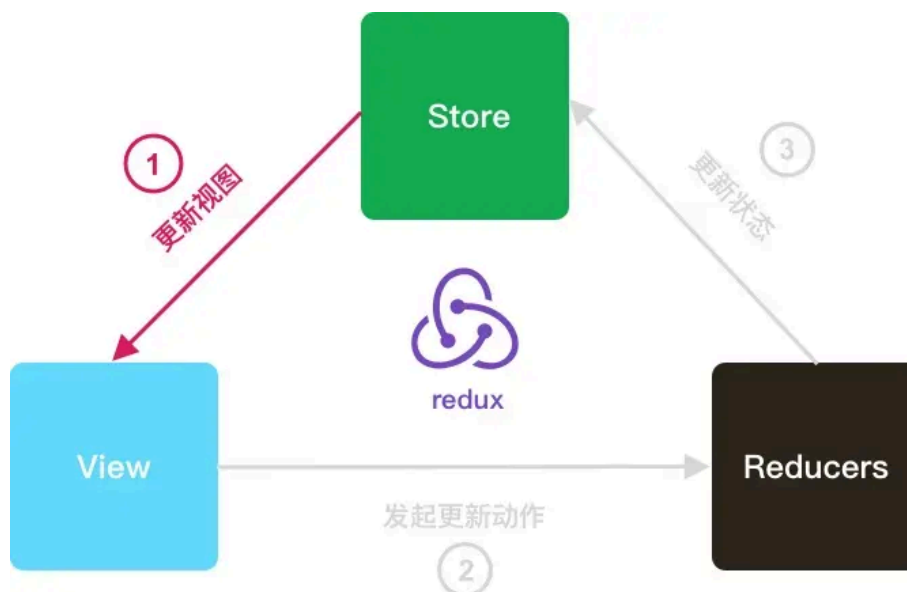
`connect` 其实是一个高阶函数，高阶函数就是指可以接收参数调用并返回另外一个函数的函数。这里 `connect` 通过接收 `mapStateToProps` 然后调用返回一个新函数，接着这个新函数再接收 `App` 组件作为参数，通过 `mapStateToProps` 注入 `todos` 和 `filter` 属性，最后返回注入后的 `App` 组件。

### 提示

这里之所以我们能在 `App` 组件中通过 `mapStateToProps` 拿到 Store 中保存的 JavaScript 对象状态树，是因为我们在之前通过 `Provider` 包裹了

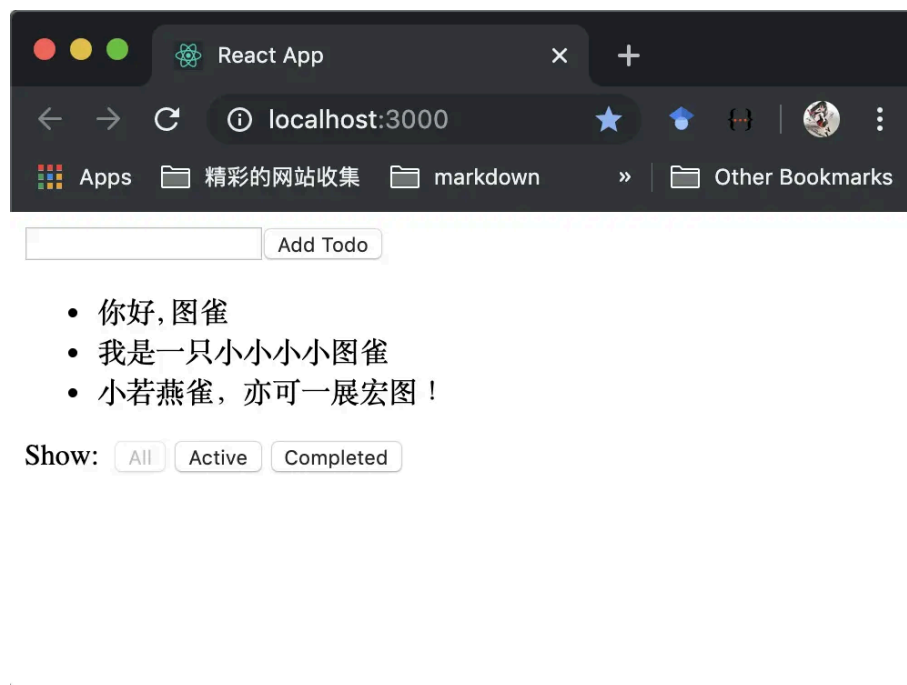
App 组件，并将 store 作为属性传递给了 Provider。

## 再现 Redux 环形图



现在再来看一看我们在第一步骤中提到的环形图，我们现在处于这个流程的第一步，即将 Store 里面的状态传递到 View 中，具体我们是通过 React 的 Redux 绑定库 `react-redux` 中的 `connect` 实现的。

保存改变的内容，如果你的 React 开发服务器打开着，那么你应该可以在浏览器中看到如下内容：



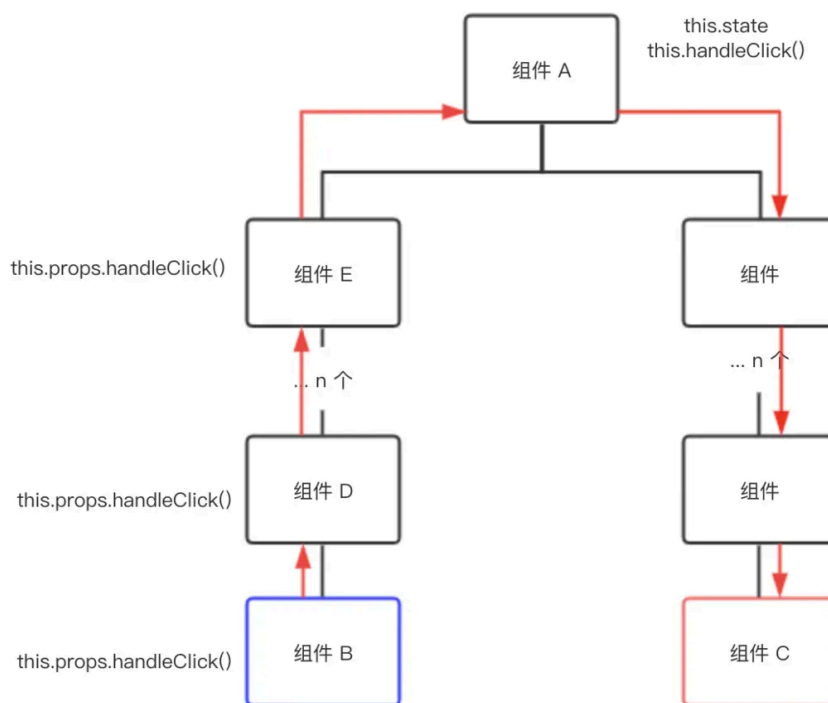
恭喜你！你已经成功编写了 Redux 的 Store，完成将 Redux 整合进 React 工作的 1/3。通过在 React 中接入 Store，你成功的将 Redux 和 React 之

间的数据打通，并删除了 `this.state`，使用 Store 的状态来取代 `this.state`。

但是！当你此时点击 `Add Todo` 按钮，你的浏览器应该会显示出红色的错误，因为我们已经删除了 `this.state` 的内容，所以在 `onSubmit` 方法中读取 `this.state.todos` 就会报错。别担心，我们将在下一节中：`Action` 中讲解如何解决这些错误。

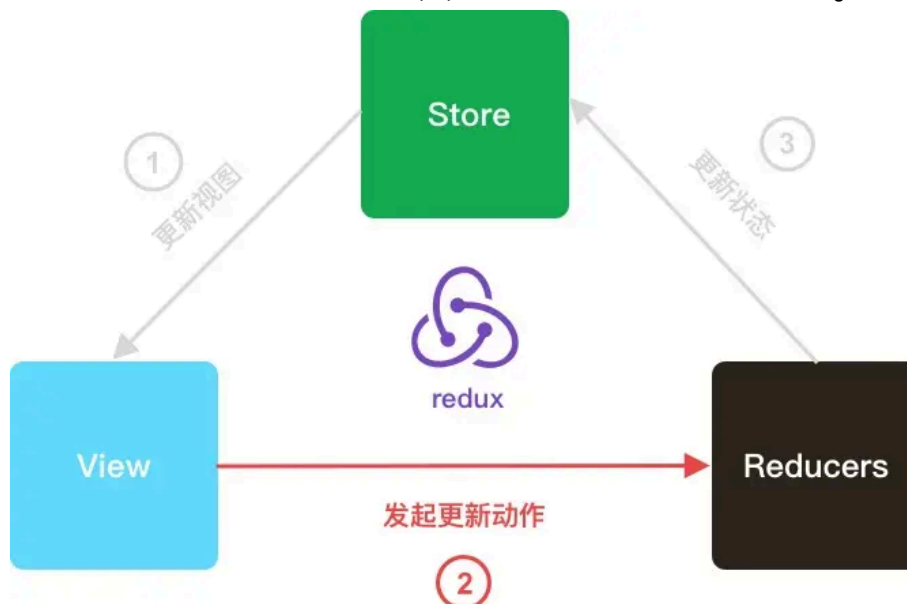
## 理解 Action: 改变 State 的唯一手段

欢迎来到 Redux Action 环节，让我们再一次引用上一节提到的图：



在上一节中，我们就在组件 B 中完成某种动作来修改组件 C 中的内容，详细剖析了完全基于 React 实现的弊端，并通过引出 Redux Store 的概念，讲解了我们只需要建一个全局 JavaScript 对象状态树，然后所有的状态的改变都是通过修改这一状态树，进而将修改后的新状态传给相应的组件并触发重新渲染来完成我们的目的。并且我们讲解了如何将 Store 里面的状态传给 React 组件使用。

这一节我们就来讲一讲，如何修改 Redux Store 中保存的状态。让我们再抛出熟悉的 Redux 状态环形图：



修改 Store 中保存的状态就是上面这张图的第二个部分，即我们已经创建好了 Store，并在里面存储了一棵 JavaScript 对象状态树，我们通过“发起更新动作”来修改 Store 中保存的状态。

## Action 是什么？

在 Redux 的概念术语中，更新 Store 的状态有且仅有一种方式：那就是调用 `dispatch` 函数，传递一个 `action` 给这个函数。

一个 Action 就是一个简单的 JavaScript 对象：

```
{ type: 'ADD_TODO', text: '我是一只小小小图雀' }
```

我们可以看到一个 `action` 包含动作的类型，以及更新状态需要的数据，其中 `type` 是必须的，其它内容都是可选的，这里我们除了 `type`，还额外添加了一个 `text`，代表我们发起 `type` 为 `ADD_TODO` 的动作是，额外传递了一个 `text` 内容。

所以如果我们需要更新 Store 状态，那么就需要类似下面的函数调用：

```
dispatch({ type: 'ADD_TODO', text: '我是一只小小小图雀' })
```

## 使用 Action Creators

因为我们在创建 Action 的时候，有时候有些内容是固定了，比如我们的待办事项添加教程的 Action，有三个字段，分别是 `type`，`text`，`id`，我们



可能会要在多个地方可以 `dispatch` 这个 Action，那么我们每次都需要写下面长长的一串：

```
{ type: 'ADD_TODO', text: '我是一只小小小图雀' , id: 0}  
{ type: 'ADD_TODO', text: '小若燕雀，亦可一展宏图' , id: 1}  
...  
{ type: 'ADD_TODO', text: '欢迎你加入图雀社区！' , id: 10}
```

对 JavaScript 函数比较熟悉的同学可能就知道该如何解决这种问题。是的，我们只需要定义一个函数，然后传入需要变化的参数就可以了。

```
let nextTodoId = 0;  
  
const addTodo = text => ({  
  type: "ADD_TODO",  
  id: nextTodoId++,  
  text  
});
```

这种接收一些需要修改的参数，返回一个 Action 的函数在 Redux 中被称为 Action Creators（动作创建器）。

当我们使用 Action Creators 来创建 Action 之后，我们再想要修改 Store 的状态就变成了下面这样：

```
dispatch(addTodo('我是一只小小小图雀'))
```

可以看到，我们的逻辑大大简化了，每次发起一个新的 "ADD\_TODO" action，都只需要传入对应的 text。

## 与 React 整合

了解了 Action 的基础概念之后，我们马上来尝试一下如何在 React 中发起更新动作。

首先，我们在 `src` 文件夹下面创建 `actions` 文件夹，然后在 `actions` 文件夹下创建 `index.js` 文件，并在里面添加下面的 Action Creators：

```
let nextTodoId = 0;  
  
export const addTodo = text => ({  
  type: "ADD_TODO",  
  id: nextTodoId++,
```

```
text
});
```

因为在使用 Redux 的 React 应用中, 我们将需要创建大量的 Action 或者 Action Creators, 所以 Redux 社区的最佳实践推荐我们创建一个独立的 `actions` 文件夹, 并在这个文件夹里面编写特定的 Action 逻辑。

可以看到, 我们加入了一个 `addTodo` Action Creator, 它接收 `text` 参数, 并每次自增一个 `id`, 然后返回带有 `id` 和 `text`, 并且类型为 `"ADD_TODO"` 的 Action。

接着我们修改 `src/components/AddTodo.js` 文件, 将之前的 `onSubmit` 替换成以 `dispatch(action)` 的形式来修改 Store 的状态:

```
import React from "react";
import { connect } from "react-redux";
import { addTodo } from "../actions";

const AddTodo = ({ dispatch }) => {
  let input;

  return (
    <div>
      <form
        onSubmit={e => {
          e.preventDefault();
          if (!input.value.trim()) {
            return;
          }
          dispatch(addTodo(input.value));
          input.value = "";
        }}
      >
        <input ref={node => (input = node)} />
        <button type="submit">Add Todo</button>
      </form>
    </div>
  );
};
```

可以看到, 上面的代码做了这几项改变:

- 首先我们从 `react-redux` 中导出了 `connect` 函数, 它负责将 Store 中的状态注入组件的同时, 还给组件传递了一个额外的方法: `dispatch`, 这样我们就可以在组件的 `props` 中获取这个方法。注意到我们在 `AddTodo` 函数式组件中使用了对象解构来获取 `dispatch` 方法。

- 导出了我们刚刚创建的 `addTodo` Action Creators。
- 之后我们使用 `addTodo` 接收 `input.value` 输入值，创建一个类型为 `"ADD_TODO"` 的 Action，并使用 `dispatch` 函数将这个 Action 发送给 Redux，请求更新 Store 的内容，更新 Store 的状态需要 Reducers 来进行操作，我们将在 Reducer 中详细讲解它。

因为我们已经将直接修改 `this.state` 的 `onSubmit` 换成了 `dispatch` 一个 Action，所以我们删除 `src/components/App.js` 相应的代码，因为我们现在已经不需要它们了：

```
import React from "react";
import AddTodo from "./AddTodo";
import TodoList from "./TodoList";
import Footer from "./Footer";

import { connect } from "react-redux";

// 省略 VisibilityFilters 和 getVisibleTodos ...

class App extends React.Component {
  constructor(props) {
    super(props);

    this.toggleTodo = this.toggleTodo.bind(this);
    this.setVisibilityFilter = this.setVisibilityFilter.bind(this)
  }

  toggleTodo(id) {
    const { todos } = this.state;

    this.setState({
      todos: todos.map(todo =>
        todo.id === id ? { ...todo, completed: !todo.completed
      )
    });
  }
}
```

可以看到我们删除了 `nextTodoId`，因为我们已经在 `src/actions/index.js` 中重新定义了它；接着我们删除了 `onSubmit` 方法；最后我们删除了传递给 `AddTodo` 组件的 `onSubmit` 方法。

保存修改的内容，我们在待办事项小应用的输入框里面输入点内容，然后点击 `Add Todo` 按钮，我们发现，之前的错误没有再次出现。

## 留有遗憾的小结

在这一节中，我们完成了 Redux 状态环形图的第二个部分，即发起更新动作，我们首先讲解了什么是 Action 和 Action Creators，然后通过

`dispatch(action)` 的方式来发起一个更新 Store 中状态的动作。

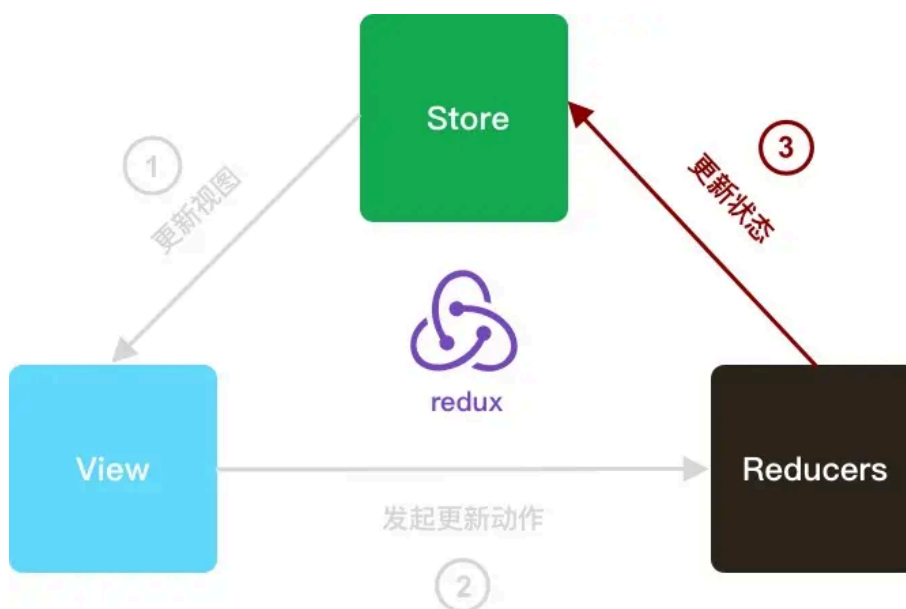
当我们使用了 `dispatch(action)` 之后，传递给子组件，用来修改父组件 State 的方法就不需要了，所以我们在代码中删除了它们。在我们的 `AddTodo` 中，这个方法就是 `onSubmit`。

但是有一点遗憾就是，我们虽然删除了 `onSubmit` 方法，但是我们这一节中讲到和实现的 `dispatch(action)` 还只能完成之前 `onSubmit` 方法的一半功能，即发起修改动作，但是我们目前还无法修改 Store 中的状态。为了修改 Store 中的 State，我们需要定义 Reducers，用于响应我们 `dispatch` 的 Action，并根据 Action 的要求修改 Store 中对应的数据。

## 理解 Reducers: 响应 Action 的指令

在这一节中，我们马上来了结上一节中留下的遗憾，即我们好像放了一声空炮，`dispatch` 了一个 Action，但是没有收获任何效果。

首先祭出我们万能的 Redux 状态循环图：



我们已经完成了前两步了，离 Redux 整合进 React 只剩下最后一个步骤，即响应从组件中 `dispatch` 出来 Action，并更新 Store 中的状态，这在 Redux 的概念中被称之为 Reducers。

## 纯化的 Reducers

`reducer` 是一个普通的 JavaScript 函数，它接收两个参数：`state` 和 `action`，前者为 Store 中存储的那棵 JavaScript 对象状态树，后者即为我们组件中 `dispatch` 的那个 Action。

```
reducer(state, action) {  
  // 对 state 进行操作  
  return newState;  
}
```

reducer 根据 action 的指示，对 state 进行对应的操作，然后返回操作后的 state，Redux Store 会自动保存这份新的 state。

### 注意

Redux 官方社区对 reducer 的约定是一个纯函数，即我们不能直接修改 state，而是可以使用 {...} 等对象解构手段返回一个被修改后的新 state。

比如我们对 state = { a: 1, b: 2 } 进行修改，将 a 替换成 3，我们应该这么做：newState = { ...state, a: 3 }，而不应该 state.a = 3。这种不直接修改原对象，而是返回一个新对象的修改，我们称之为“纯化”的修改。

## 准备响应 Action 的修改

当了解了 Reducer 的概念之后，我们马上在应用中响应我们之前 dispatch 的 Action，来弥补我们在上一节中留下的遗憾。

打开 src/index.js，对 rootReducer 作出如下修改：

```
// ...  
  
const rootReducer = (state, action) => {  
  switch (action.type) {  
    case "ADD_TODO": {  
      const { todos } = state;  
  
      return {  
        ...state,  
        todos: [  
          ...todos,  
          {  
            id: action.id,  
            text: action.text,  
            completed: false  
          }  
        ]  
      };  
    }  
    default:  
      return state;  
  }  
}
```

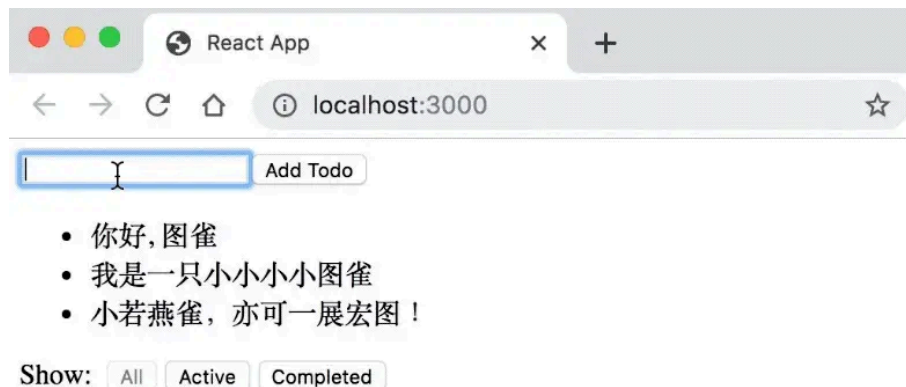
```
};  
  
// ...
```

上面的代码做了这么几项工作：

- 可以看到，我们将之前的 `rootReducer` 进行改进，从单纯地返回原来的 `state`，变成了一个 `switch` 语句，在 `switch` 语句中对 `action` 的 `type` 进行判断，然后做出对应的处理。
- 当 `action.type` 的类型为 `"ADD_TODO"` 时，我们从 `state` 中取出了 `todos`，然后使用 `{...}` 语法给 `todos` 添加一个新的元素对象，并设置 `completed` 属性为 `false` 代表此 `todo` 未完成，最后再通过一层 `{...}` 语法将新的 `todos` 合并进老的 `state` 中，返回这个新的 `state`。
- 当 `action.type` 没有匹配 `switch` 的任何条件时，我们返回默认的 `state`，表示 `state` 没有任何更新。

当我们对 `rootReducer` 函数做了上述的改动之后，Redux 通过 `Reducer` 函数就可以响应从组件中 `dispatch` 出来的 `action` 了，目前我们还只可以响应 `action.type` 为 `"ADD_TODO"` 的 `action`，它表示新增一个 `todo`。

保存修改的代码，打开浏览器，在输入框里面输入点内容，然后点击 `Add Todo` 按钮，现在网页应该可以正确响应你的操作了，我们又可以愉快地添加新的待办事项了。

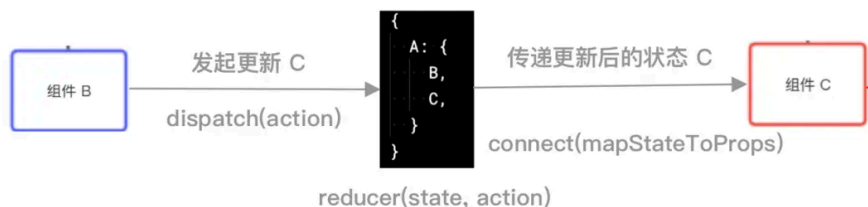


## 小结

在这一小节中，我们实现了第一个可以响应组件 `dispatch` 出来的 `Action` 的 `Reducer`，它判断 `action.type` 的类型，并根据这些类型对 `state` 进行

“纯化”的修改，当 `action.type` 没有匹配 Reducer 中任何类型时，我们返回原来的 `state`。

当了解了 Redux 三大概念：Store，Action，Reducers 之后，我们再来看一张图：



这张图我们之前看过类似的，只不过这一次我们在这张图上加了点东西，分别标出了 `dispatch`、`reducers` 和 `connect` 所完成的工作。

- `dispatch(action)` 用来在 React 组件中发出修改 Store 中保存状态的指令。在我们需要新加一个待办事项时，它取代了之前定义在组件中的 `onSubmit` 方法。
- `reducer(state, action)` 用来根据这一指令修改 Store 中保存状态对应的部分。在我们需要新加一个待办事项时，它取代了之前定义在组件中的 `this.setState` 操作。
- `connect(mapStateToProps)` 用来将更新好的数据传给组件，然后触发 React 重新渲染，显示最新的状态。它架设起 Redux 和 React 之间的数据通信桥梁。

现在，Redux 的核心概念你已经全部学完了，并且我们的应用已经完全整合了 Redux。但是，我们还有一点工作没有完成，那就是将整个应用完全使用 Redux 重构。在下一篇文章中，我们将使用我们在上面三节学到的知识，一步一步将我们的待办事项应用的其他部分重构成 Redux，欢迎继续阅读。

想要学习更多精彩的实战技术教程？来图雀社区逛逛吧。





[前端](#) [react.js](#) [redux](#) 赞 12 收藏 9 分享

阅读 3.7k • 更新于 2020-03-22



## 一只图雀

863 声望 • 1.2k 粉丝

我们图雀社区是一个供大家分享用 Tuture 写作工具撰写教程的一个平台。在这里，读者们可以尽情享受高质量的实战教...

[关注作者](#)

« 上一篇

[一杯茶的时间，上手 Node.js](#)

下一篇 »

[Redux 包教包会 \(二\) : 引入...](#)


## 引用和评论

### 被 5 篇内容引用




Taro 小程序开发大型实战 (五) : 使用 Hooks 版的 Redux 实现应用状态管理 (下篇)



Taro 小程序开发大型实战 (六) : 尝鲜微信小程序云 (上篇)  1



Taro 小程序开发大型实战 (四) : 使用 Hooks 版的 Redux 实现应用状态管理 (上篇)  1



Redux 包教包会 (三) : 使用容器组件和展示组件进一步分离组件状态



Redux 包教包会 (二) : 引入 combineReducers 拆分和组合状态逻辑  1

### 推荐阅读



Taro 小程序开发大型实战 (九) : 使用 Authing 打造具有微信登录的企业级用户系统

一只图雀 • 赞 1 • 阅读 4.4k

**深入理解React Diff算法**

nero · 赞 36 · 阅读 15.8k · 评论 4

**前端开发方法集**

寒青 · 赞 23 · 阅读 2.6k

**React中的高优先级任务插队机制**

nero · 赞 32 · 阅读 15.1k · 评论 14

**关于小程序如何做到强制更新**

南玖 · 赞 13 · 阅读 828 · 评论 1

**【你不知道的canvas】之更换绿屏视频背景**

雾岛听风 · 赞 15 · 阅读 6.3k · 评论 8

**【动画进阶】巧用 CSS/SVG 实现复杂线条光效动画**

chokcoco · 赞 14 · 阅读 924

**3 条评论**

得票

最新



撰写评论 ...



提交评论

评论支持部分 Markdown 语法: **\*\*粗体\*\*** *\_斜体\_* [链接]  
(<http://example.com>) ``代码`` - 列表 > 引用。你还可以使用 @  
来通知其他用户。

**指尖泛出的繁华**: 大佬又出文章了? 更新这么频繁, 抱拳了

👍 · 回复 · 2020-03-13

**一只图雀** (作者): @指尖泛出的繁华 这要归功于我们的写作工具, 让文章创作变得极其高效。有了高效工具的加持, 我们就可以把精力放在内容质量上了, 希望这篇文章能够给您带来帮助!!!

👍 · 回复 · 2020-03-13

**joyco**: 每次看都有不同体会, 受教啦!! 谢谢大佬

👍 · 回复 · 2020-09-26

©2024 图雀社区

除特别声明外，作品采用《署名-非商业性使用-禁止演绎 4.0 国际》进行许可

 使用 SegmentFault 发布

SegmentFault - 凝聚集体智慧，推动技术进步

服务协议 · 隐私政策 · 浙ICP备15005796号-2 · 浙公网安备33010602002000号