

# Redux的使用详解(一)

王红元 coderwhy

# 目录

## content



**1** Redux的核心思想

**2** Redux的基本使用

**3** React结合Redux

**4** Redux的异步操作

**5** redux-devtool

**6** reducer的模块拆分

# 理解JavaScript纯函数

- 函数式编程中有一个非常重要的概念叫**纯函数**，JavaScript符合**函数式编程的范式**，所以也有**纯函数的概念**；
  - 在react开发中纯函数是被多次提及的；
  - 比如react中组件就被要求像是一个**纯函数**（为什么是像，因为还有class组件），**redux**中有一个**reducer**的概念，也是要求必须是一个纯函数；
  - 所以**掌握纯函数对于理解很多框架的设计是非常有帮助的**；
- **纯函数的维基百科定义**：
  - 在程序设计中，若一个函数**符合以下条件**，那么这个函数被称为纯函数：
    - 此函数**在相同的输入值时**，需**产生相同的输出**。
    - 函数的**输出和输入值以外的其他隐藏信息或状态无关**，也和由I/O设备产生的**外部输出无关**。
    - 该函数**不能有语义上可观察的函数副作用**，诸如“**触发事件**”，使输出设备输出，或更改输出值以外物件的内容等。
- 当然上面的定义会过于的晦涩，所以我简单总结一下：
  - **确定的输入，一定会产生确定的输出**；
  - **函数在执行过程中，不能产生副作用**；

# 副作用概念的理解

■ 那么这里又有一个概念，叫做副作用，什么又是副作用呢？

- 副作用 (side effect) 其实本身是医学的一个概念，比如我们经常说吃什么药本来是为了治病，可能会产生一些其他的副作用；
- 在计算机科学中，也引用了副作用的概念，表示在执行一个函数时，除了返回函数值之外，还对调用函数产生了附加的影响，比如修改了全局变量，修改参数或者改变外部的存储；

■ 纯函数在执行的过程中就是不能产生这样的副作用：

- 副作用往往是产生bug的“温床”。

# 纯函数的案例

## ■ 我们来看一个对数组操作的两个函数：

□ **slice**：slice截取数组时不会对原数组进行任何操作,而是生成一个新的数组；

□ **splice**：splice截取数组, 会返回一个新的数组, 也会对原数组进行修改；

## ■ slice就是一个纯函数，不会修改数组本身，而splice函数不是一个纯函数；

```
var names = ["abc", "cba", "nba", "dna"]

// slice截取数组时不会对原数组进行任何操作, 而是生成一个新的数组
var newNames = names.slice(0, 2)
console.log(newNames)

// splice截取数组, 会返回一个新的数组, 也会对原数组进行修改
var newNames2 = names.splice(0, 2)
console.log(newNames2)
console.log(names)
```

# 判断下面函数是否是纯函数?

```
function sum(num1, num2) {  
  return num1 + num2;  
}
```

```
let foo = 5;  
  
function add(num) {  
  return foo + num;  
}  
  
console.log(add(5));  
foo = 10;  
console.log(add(5));
```

```
function printInfo(info) {  
  console.log(info.name, info.age);  
  info.name = "哈哈";  
}
```

# 纯函数的作用和优势

## ■ 为什么纯函数在函数式编程中非常重要呢？

- 因为你可以**安心的编写**和**安心的使用**；
- 你在**写的时候**保证了函数的纯度，只是**单纯实现自己的业务逻辑**即可，**不需要关心传入的内容**是如何获得的或者**依赖其他的外部变量**是否已经发生了修改；
- 你在**用的时候**，你确定**你的输入内容不会被任意篡改**，并且**自己确定的输入**，一定会有**确定的输出**；

## ■ React中就要求我们无论是函数还是class声明一个组件，这个组件都必须像纯函数一样，保护它们的props不被修改：

React 非常灵活，但它也有一个严格的规则：

所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。

## ■ 在接下来学习redux中，reducer也被要求是一个纯函数。

# 为什么需要redux

## ■ JavaScript开发的应用程序，已经变得越来越复杂了：

- JavaScript需要管理的状态越来越多，越来越复杂；
- 这些状态包括服务器返回的数据、缓存数据、用户操作产生的数据等等，也包括一些UI的状态，比如某些元素是否被选中，是否显示加载动效，当前分页；

## ■ 管理不断变化的state是非常困难的：

- 状态之间相互会存在依赖，一个状态的变化会引起另一个状态的变化，View页面也有可能引起状态的变化；
- 当应用程序复杂时，state在什么时候，因为什么原因而发生了变化，发生了怎么样的变化，会变得非常难以控制和追踪；

## ■ React是在视图层帮助我们解决了DOM的渲染过程，但是State依然是留给我们自己来管理：

- 无论是组件定义自己的state，还是组件之间的通信通过props进行传递；也包括通过Context进行数据之间的共享；
- React主要负责帮助我们管理视图，state如何维护最终还是我们自己来决定；

```
UI = render(state)
```

## ■ Redux就是一个帮助我们管理State的容器：Redux是JavaScript的状态容器，提供了可预测的状态管理；

## ■ Redux除了和React一起使用之外，它也可以和其他界面库一起来使用（比如Vue），并且它非常小（包括依赖在内，只有2kb）



# Redux的核心理念 - Store

- Redux的核心理念非常简单。
- 比如我们有一个朋友列表需要管理：
  - 如果我们没有定义统一的规范来操作这段数据，那么整个数据的变化就是无法跟踪的；
  - 比如页面的某处通过products.push的方式增加了一条数据；
  - 比如另一个页面通过products[0].age = 25修改了一条数据；
- 整个应用程序错综复杂，当出现bug时，很难跟踪到底哪里发生的变化；

```
const initialState = {  
  friends: [  
    { name: "why", age: 18 },  
    { name: "kobe", age: 40 },  
    { name: "lilei", age: 30 },  
  ],  
};
```

# Redux的核心理念 - action

## ■ Redux要求我们通过action来更新数据:

- 所有数据的变化, 必须通过派发 (dispatch) action来更新;
- action是一个普通的JavaScript对象, 用来描述这次更新的type和content;

## ■ 比如下面就是几个更新friends的action:

- 强制使用action的好处是可以清晰的知道数据到底发生了什么样的变化, 所有的数据变化都是可跟追、可预测的;
- 当然, 目前我们的action是固定的对象;
- 真实应用中, 我们会通过函数来定义, 返回一个action;

```
const action1 = { type: "ADD_FRIEND", info: { name: "lucy", age: 20 } }  
const action2 = { type: "INC_AGE", index: 0 }  
const action3 = { type: "CHANGE_NAME", payload: { index: 0, newName: "coderwhy" } }
```

# Redux的核心理念 - reducer

■ 但是如何将state和action联系在一起呢？答案就是reducer

□ reducer是一个纯函数；

□ reducer做的事情就是将传入的state和action结合起来生成一个新的state；

```
function reducer(state = initialState, action) {  
  switch (action.type) {  
    case "ADD_FRIEND":  
      return { ...state, friends: [...state.friends, action.info] }  
    case "INC_AGE":  
      return {  
        ...state, friends: state.friends.map((item, index) => {  
          if (index === action.index) {  
            return { ...item, age: item.age + 1 }  
          }  
          return item;  
        })  
      }  
    case "CHANGE_NAME":  
      return {  
        ...state, friends: state.friends.map((item, index) => {  
          if (index === action.index) {  
            return { ...item, name: action.newName }  
          }  
          return item;  
        })  
      }  
    default:  
      return state;  
  }  
}
```

# Redux的三大原则

## ■ 单一数据源

- 整个应用程序的state被存储在一颗object tree中，并且这个object tree只存储在一个 store 中；
- Redux并没有强制让我们不能创建多个Store，但是那样做并不利于数据的维护；
- 单一的数据源可以让整个应用程序的state变得方便维护、追踪、修改；

## ■ State是只读的

- 唯一修改State的方法一定是触发action，不要试图在其他地方通过任何的方式来修改State；
- 这样就确保了View或网络请求都不能直接修改state，它们只能通过action来描述自己想要如何修改state；
- 这样可以保证所有的修改都被集中化处理，并且按照严格的顺序来执行，所以不需要担心race condition（竞态）的问题；

## ■ 使用纯函数来执行修改

- 通过reducer将旧state和 actions联系在一起，并且返回一个新的State；
- 随着应用程序的复杂度增加，我们可以将reducer拆分成多个小的reducers，分别操作不同state tree的一部分；
- 但是所有的reducer都应该是纯函数，不能产生任何的副作用；

# Redux测试项目搭建

## ■ 安装redux:

```
npm install redux --save  
# 或  
yarn add redux
```

## ■ 1.创建一个新的项目文件夹: learn-redux

```
# 执行初始化操作  
yarn init  
  
# 安装redux  
yarn add redux
```

## ■ 2.创建src目录, 并且创建index.js文件

## ■ 3.修改package.json可以执行index.js

```
"scripts": {  
  "start": "node src/index.js"  
}
```

# Redux的使用过程

- 1.创建一个对象，作为我们要保存的状态：
- 2.创建Store来存储这个state
  - 创建store时必须创建reducer；
  - 我们可以通过 `store.getState` 来获取当前的state；
- 3.通过action来修改state
  - 通过dispatch来派发action；
  - 通常action中都会有type属性，也可以携带其他的数据；
- 4.修改reducer中的处理代码
  - 这里一定要记住，reducer是一个纯函数，不需要直接修改state；
  - 后面我会讲到直接修改state带来的问题；
- 5.可以在派发action之前，监听store的变化：
- 注意：这里不贴出代码，直接查看上课代码。



# Redux结构划分

## ■ 如果我们将所有的逻辑代码写到一起，那么当redux变得复杂时代码就难以维护。

- 接下来，我会对代码进行拆分，将store、reducer、action、constants拆分成一个个文件。
- 创建store/index.js文件：
- 创建store/reducer.js文件：
- 创建store/actionCreators.js文件：
- 创建store/constants.js文件：

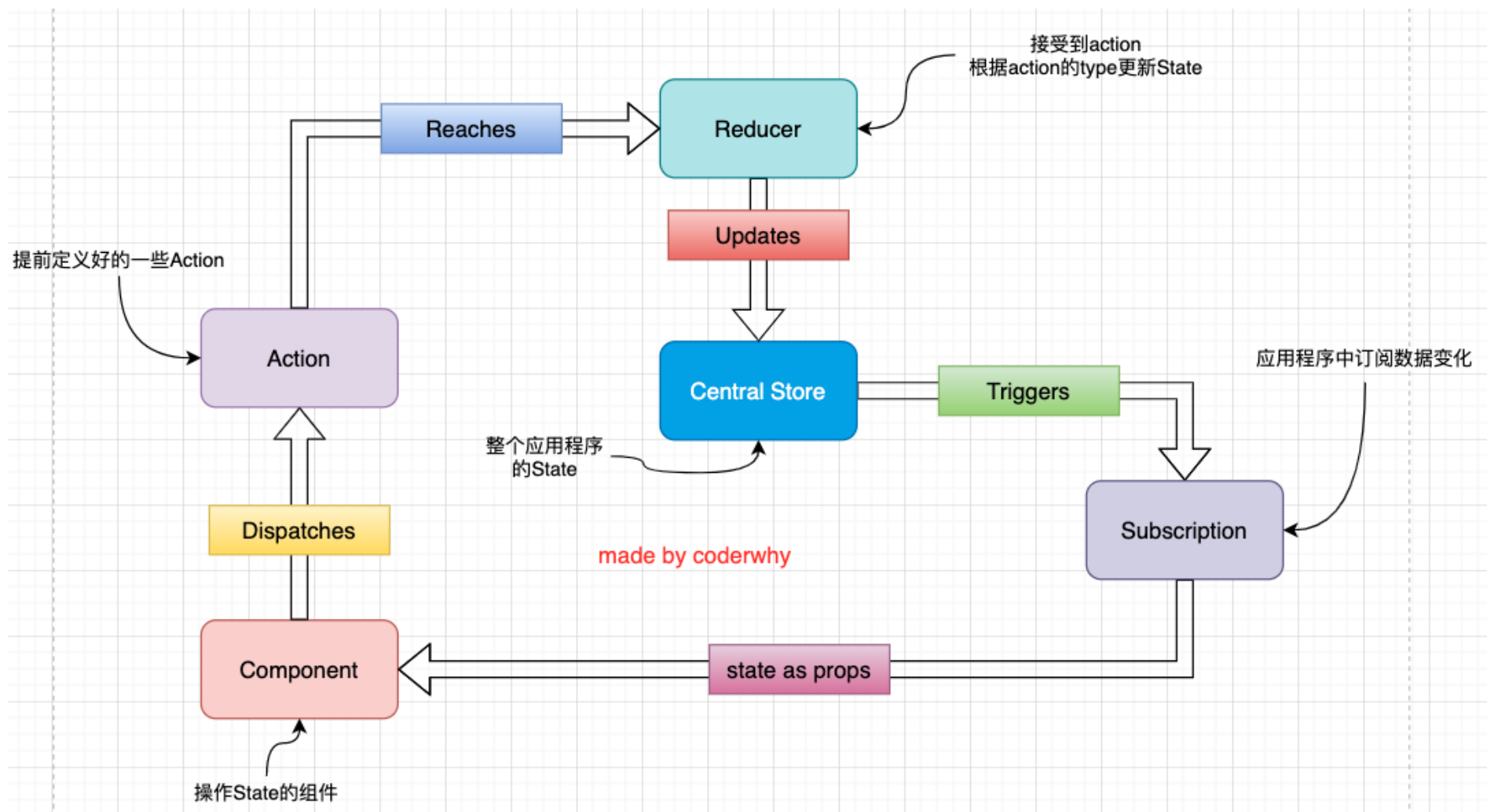
## ■ 注意：node中对ES6模块化的支持

- 目前我使用的node版本是v12.16.1，从node v13.2.0开始，node才对ES6模块化提供了支持：
- node v13.2.0之前，需要进行如下操作：
  - ✓ 在package.json中添加属性： "type": "module";
  - ✓ 在执行命令中添加如下选项： node --experimental-modules src/index.js;
- node v13.2.0之后，只需要进行如下操作：
  - ✓ 在package.json中添加属性： "type": "module";

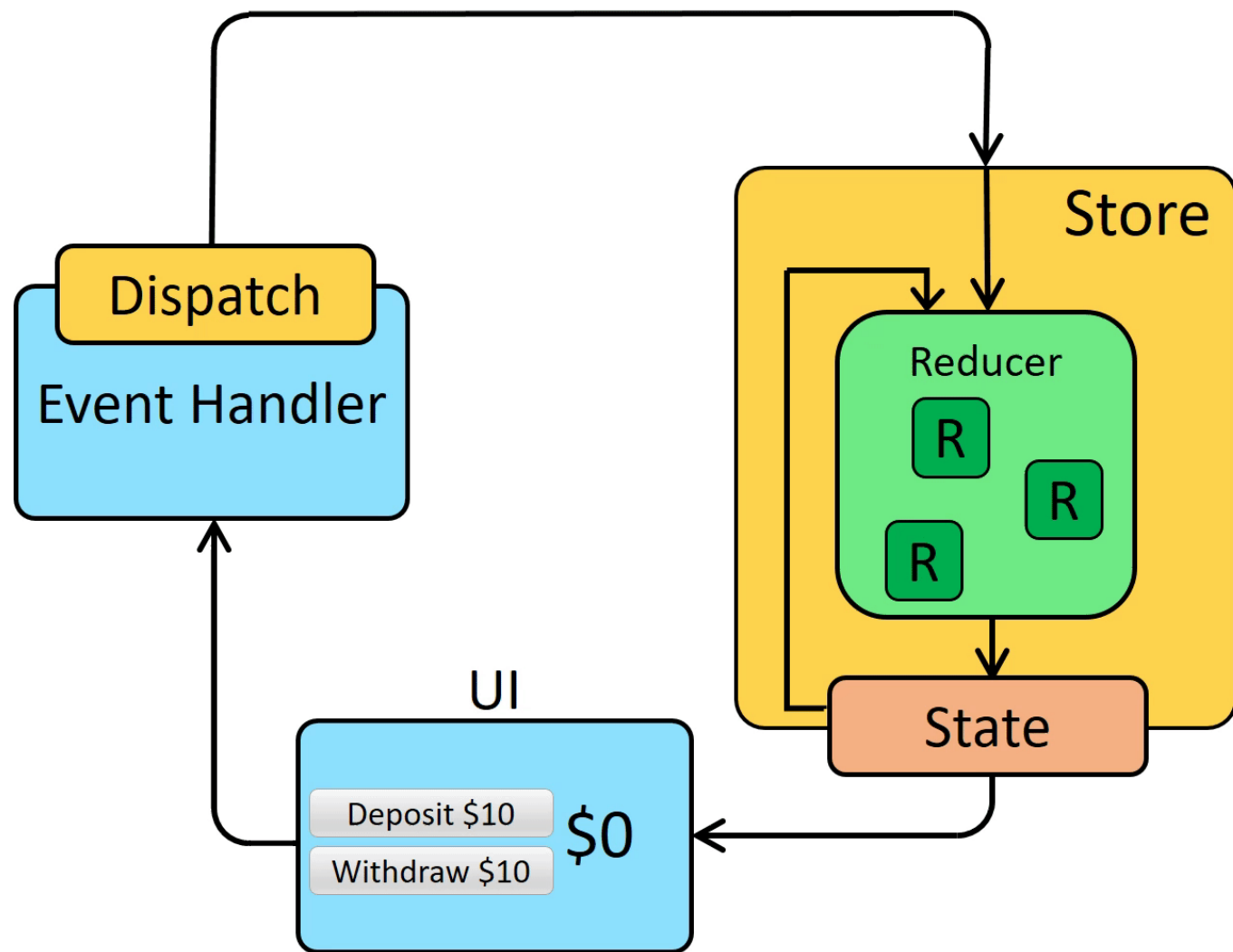
## ■ 注意：导入文件时，需要跟上.js后缀名；

# Redux使用流程

- 我们已经知道了redux的基本使用过程，那么我们就更加清晰来认识一下redux在实际开发中的流程：







# redux融入react代码

■ 目前redux在react中使用是最多的，所以我们需要将之前编写的redux代码，融入到react当中去。

■ 这里我创建了两个组件：

□ Home组件：其中会展示当前的counter值，并且有一个+1和+5的按钮；

□ Profile组件：其中会展示当前的counter值，并且有一个-1和-5的按钮；

## Home

当前计数: 0

+1 +5

## Profile

当前计数: 0

-1 -5

■ 核心代码主要是两个：

□ 在 componentDidMount 中定义数据的变化，当数据发生变化时重新设置 counter；

□ 在发生点击事件时，调用store的dispatch来派发对应的action；

# react-redux使用

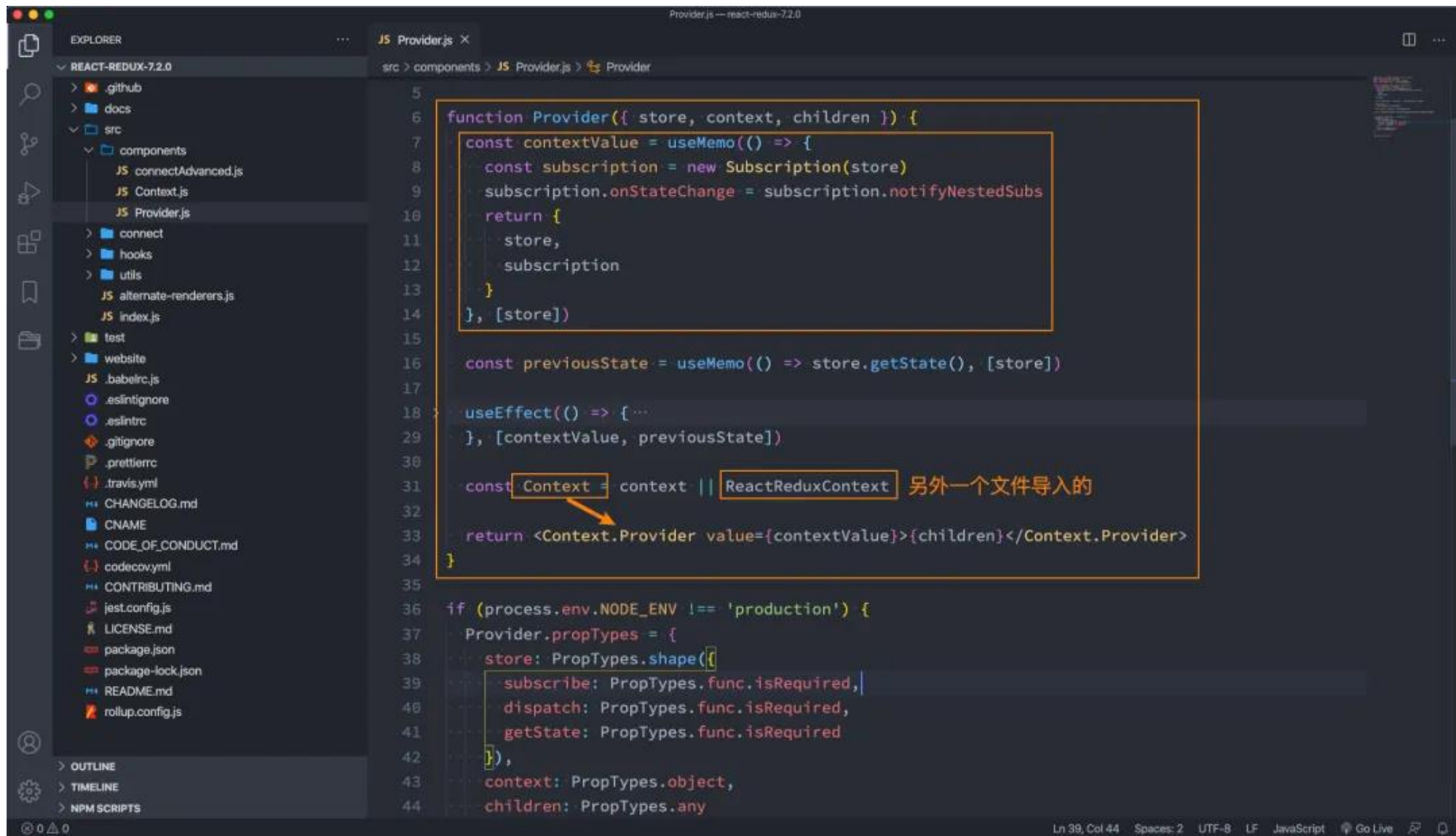
- 开始之前需要强调一下，redux和react没有直接的关系，你完全可以在React, Angular, Ember, jQuery, or vanilla JavaScript中使用Redux。
- 尽管这样说，redux依然是和React库结合的更好，因为他们是通过state函数来描述界面的状态，Redux可以发射状态的更新，让他们作出相应。
- 虽然我们之前已经实现了connect、Provider这些帮助我们完成连接redux、react的辅助工具，但是实际上redux官方帮助我们提供了 react-redux 的库，可以直接在项目中使用，并且实现的逻辑会更加的严谨和高效。
- 安装react-redux：

yarn add react-redux

```
root.render(  
  <React.StrictMode>  
    <Provider store={store}>  
      <App />  
    </Provider>  
  </React.StrictMode>  
);
```

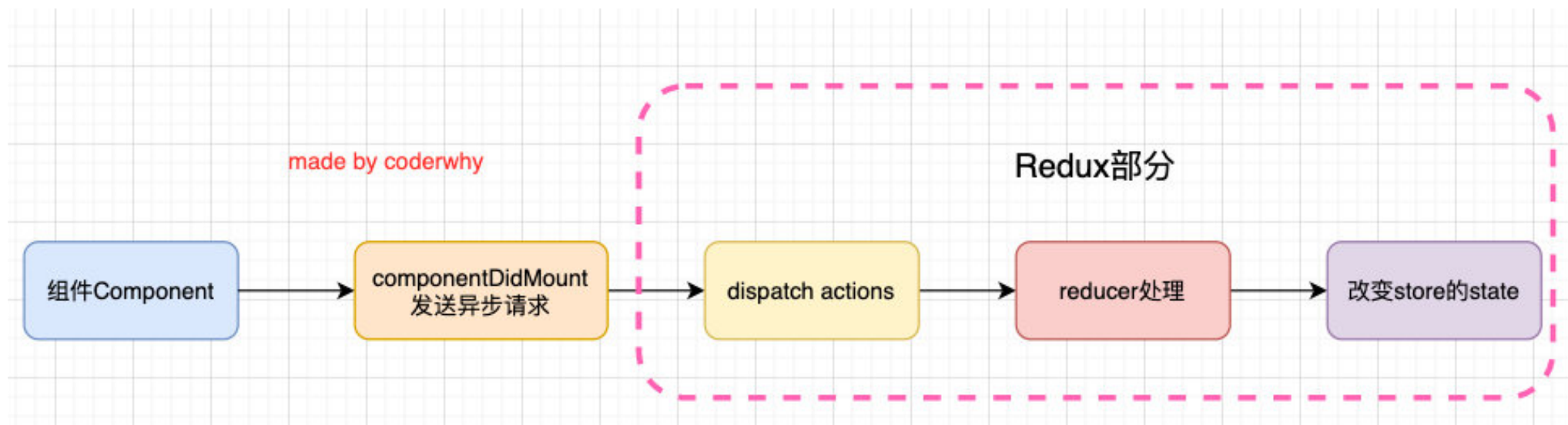
```
const mapDispatchToProps = dispatch => ({  
  addNumber: function(num) {  
    dispatch(addNumberAction(num))  
  },  
  subNumber: function(num) {  
    dispatch(subNumberAction(num))  
  },  
  getHomeMultidataAction: function() {  
    dispatch(getHomeMultidataAction())  
  }  
})  
  
export default connect(mapStateToProps, mapDispatchToProps)(Home)
```

# react-redux源码导读



# 组件中异步操作

- 在之前简单的案例中，redux中保存的counter是一个本地定义的数据
  - 我们可以直接通过同步的操作来dispatch action，state就会被立即更新。
  - 但是真实开发中，**redux中保存的很多数据可能来自服务器**，我们需要进行**异步的请求**，再将**数据保存到redux**中。
- 在之前学习网络请求的时候我们讲过，网络请求可以在class组件的componentDidMount中发送，所以我们可以有这样的结构：

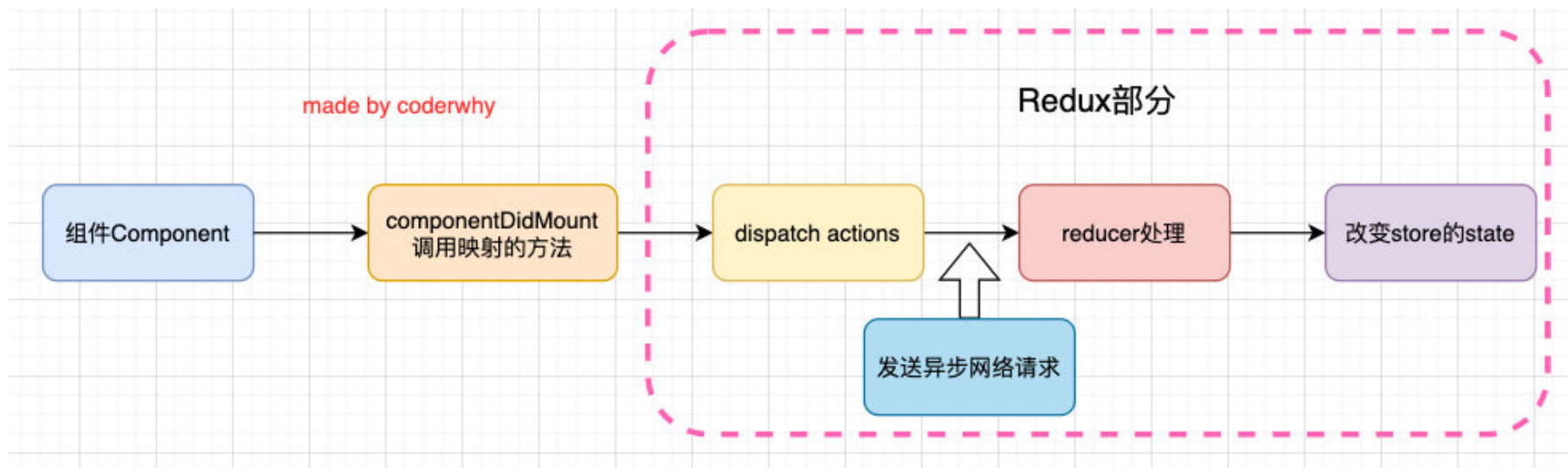


- 我现在完成如下案例操作：
  - 在Home组件中请求banners和recommends的数据；
  - 在Profile组件中展示banners和recommends的数据；

# redux中异步操作

## ■ 上面的代码有一个缺陷：

- 我们必须将网络请求的异步代码放到组件的生命周期中来完成；
- 事实上，网络请求到的数据也属于我们状态管理的一部分，更好的一种方式应该是将其也交给redux来管理；



## ■ 但是在redux中如何进行异步的操作呢？

- 答案就是使用中间件 (Middleware) ；
- 学习过Express或Koa框架的童鞋对中间件的概念一定不陌生；
- 在这类框架中，Middleware可以帮助我们请求和响应之间嵌入一些操作的代码，比如cookie解析、日志记录、文件压缩等操作；

# 理解中间件

## ■ redux也引入了中间件（Middleware）的概念：

- 这个中间件的目的是在dispatch的action和最终达到的reducer之间，扩展一些自己的代码；
- 比如日志记录、调用异步接口、添加代码调试功能等等；

## ■ 我们现在要做的事情就是发送异步的网络请求，所以我们可以添加对应的中间件：

- 这里官网推荐的、包括演示的网络请求的中间件是使用 `redux-thunk`；

## ■ `redux-thunk`是如何做到让我们可以发送异步的请求呢？

- 我们知道，默认情况下的`dispatch(action)`，`action`需要是一个JavaScript的对象；
- `redux-thunk`可以让`dispatch(action函数)`，`action`可以是一个函数；
- 该函数会被调用，并且会传给这个函数一个`dispatch`函数和`getState`函数；
  - ✓ `dispatch函数`用于我们之后再次派发`action`；
  - ✓ `getState函数`考虑到我们之后的一些操作需要依赖原来的状态，用于让我们可以获取之前的一些状态；

# 如何使用redux-thunk

## ■ 1.安装redux-thunk

```
yarn add redux-thunk
```

## ■ 2.在创建store时传入应用了middleware的enhance函数

- 通过applyMiddleware来结合多个Middleware, 返回一个enhancer;
- 将enhancer作为第二个参数传入到createStore中;

```
const enhancer = applyMiddleware(thunkMiddleware);  
const store = createStore(reducer, enhancer);
```

## ■ 3.定义返回一个函数的action:

- 注意: 这里不是返回一个对象了, 而是一个函数;
- 该函数在dispatch之后会被执行;

```
const getHomeMultidataAction = () => {  
  return (dispatch) => {  
    axios.get("http://123.207.32.32:8000/home/multidata").then(res => {  
      const data = res.data.data;  
      dispatch(changeBannersAction(data.banner.list));  
      dispatch(changeRecommendsAction(data.recommend.list));  
    })  
  }  
}
```



# redux-devtools

## ■ 我们之前讲过，redux可以方便的让我们对状态进行跟踪和调试，那么如何做到呢？

- redux官网为我们提供了redux-devtools的工具；
- 利用这个工具，我们可以知道每次状态是如何被修改的，修改前后的状态变化等等；

## ■ 安装该工具需要两步：

- 第一步：在对应的浏览器中安装相关的插件（比如Chrome浏览器扩展商店中搜索Redux DevTools即可）；
- 第二步：在redux中继承devtools的中间件；

```
import { createStore, applyMiddleware, compose } from 'redux';
import thunkMiddleware from 'redux-thunk';
import reducer from './reducer.js';

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

// 通过applyMiddleware来结合多个Middleware，返回一个enhancer
const enhancer = composeEnhancers(applyMiddleware(thunkMiddleware));
// 将enhancer作为第二个参数传入到createStore中
const store = createStore(reducer, enhancer);

export default store;
```



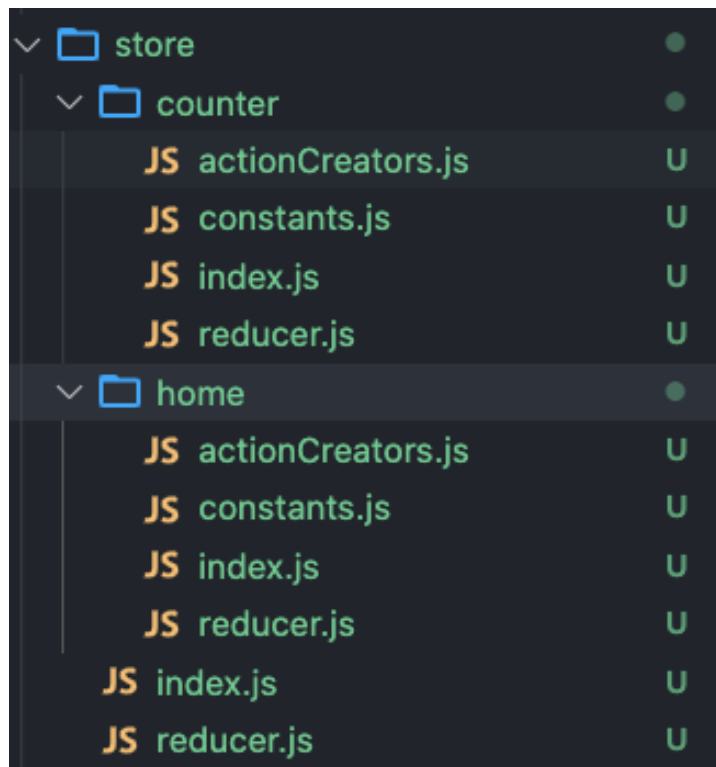
# Reducer代码拆分

- 我们先来了解一下，为什么这个函数叫reducer？
- 我们来看一下目前我们的reducer：
  - 当前这个reducer既有处理counter的代码，又有处理home页面的数据；
  - 后续counter相关的状态或home相关的状态会进一步变得更加复杂；
  - 我们也会继续添加其他的相关状态，比如购物车、分类、歌单等等；
  - 如果将所有的状态都放到一个reducer中进行管理，随着项目的日趋庞大，必然会造成代码臃肿、难以维护。
- 因此，我们可以对reducer进行拆分：
  - 我们先抽取一个对counter处理的reducer；
  - 再抽取一个对home处理的reducer；
  - 将它们合并起来；

# Reducer文件拆分

## ■ 目前我们已经将不同的状态处理拆分到不同的reducer中，我们来思考：

- 虽然已经放到不同的函数了，但是这些函数的处理依然是在同一个文件中，代码非常的混乱；
- 另外关于reducer中用到的constant、action等我们也依然是在同一个文件中；



# combineReducers函数

- 目前我们合并的方式是通过每次调用reducer函数自己来返回一个新的对象。
- 事实上，redux给我们提供了一个**combineReducers函数**可以方便的让我们对多个reducer进行合并：

```
const reducer = combineReducers({  
  counterInfo: counterReducer,  
  homeInfo: homeReducer  
})  
  
export default reducer;
```

## ■ 那么combineReducers是如何实现的呢？

- 事实上，它也是将我们传入的reducers合并到一个对象中，最终返回一个combination的函数（相当于我们之前的reducer函数了）；
- 在执行combination函数的过程中，它会通过判断前后返回的数据是否相同来决定返回之前的state还是新的state；
- 新的state会触发订阅者发生对应的刷新，而旧的state可以有效的组织订阅者发生刷新；

- 可以查看源码来学习。