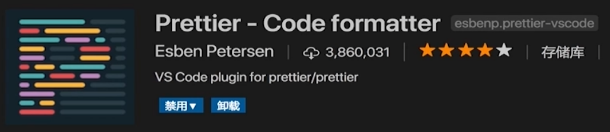
Visual Studio Code相關

## Visual Studio Code中React的快捷键

VisualStudioCode中，輸入rcc後按下Tab鍵，可以快速形成基本的react的js組件格式。

## Visual Studio Code上React相關插件

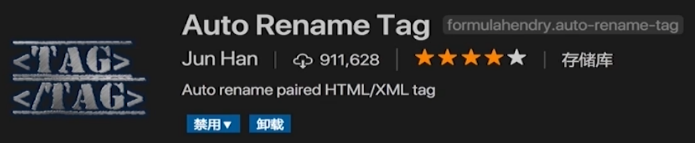
Prettier



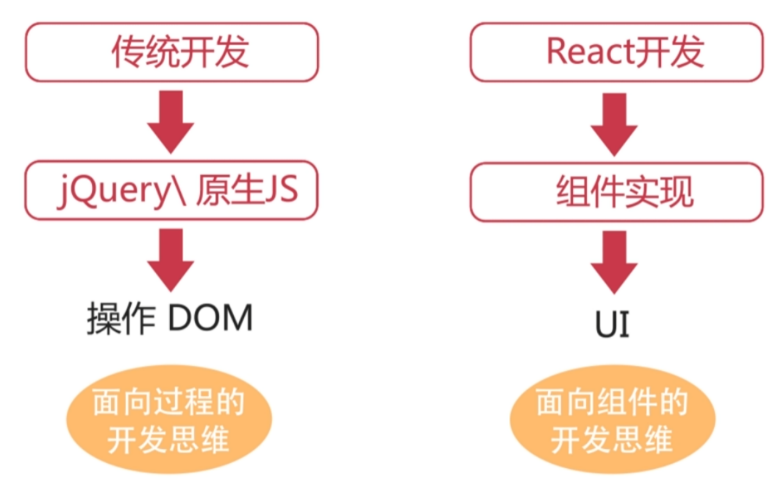
Reactjs code snippets



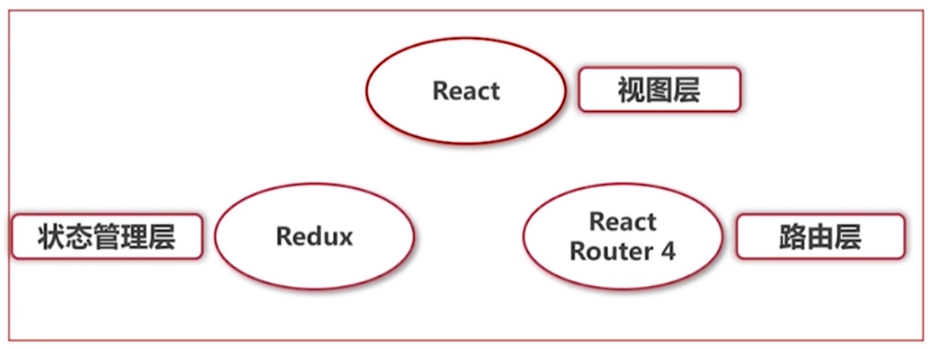
Auto Rename Tag



傳統JavaScript開發和React開發的區別



React全家桶



（<https://blog.csdn.net/qq_39261142/article/details/107310638>

這個鏈接裡有詳細的react和redux講解，寫得很好）

React 理念

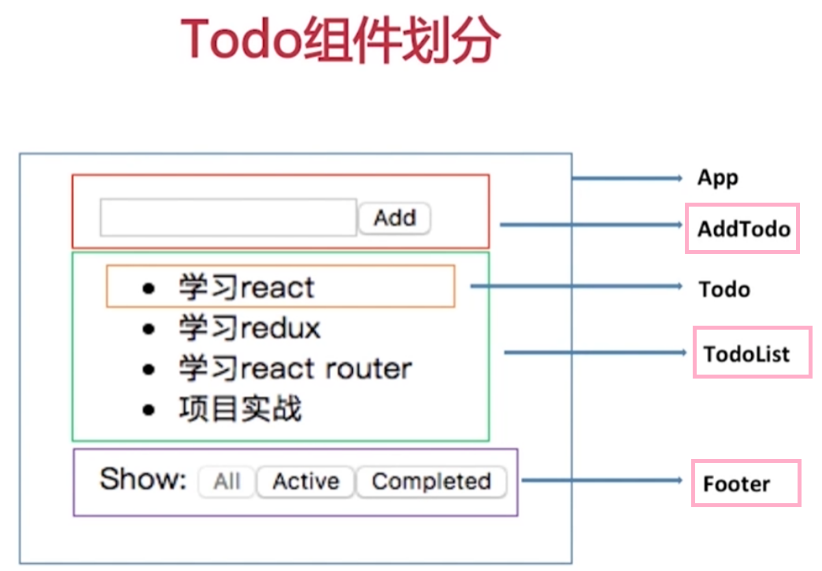
## 組件化分

組件劃分好處

* 解耦：降低單一模塊/組件的複雜度
* 複用：保證組件一致性，提升開發效率

注意：避免組件顆粒度過大或過小

例子



TODO List組件可以劃分為AddToDo，ToDoList，Footer三個大組件。

TodoList組件中又有Todo小組件。

在三大組件外面套上App這個全局組件。

## 編寫靜態組件

開發過程解耦：靜態頁面和動態頁面交互

組件開發順序：自上而下 or 自下而上

App → TodoList → Todo → AddTodo → Footer

## 設計state

代表UI的完整且最小狀態集合

這些都不是state

* 通過父組件props傳入
* 不會隨著時間、交互操作變化
* 通過其它state或props計算得到

TODOList的State

* 新增輸入框中的文本
* 待辦事項列表
* 當前篩選條件

分析State保存的位置

* 確定依賴state的每一個組件
* 如果某個state被多個組件依賴，尋找共同的父組件（狀態上移）

## 增加交互行為

* 藉助props，添加反向數據流
* 新增todo，修改todo狀態，過濾顯示

和UI渲染不相干的，不用定義到state中，比如下面的nextTodoId

  constructor(*props*) {

    super(props);

    this.state = {

      todos: [],

      filter: 'all'

    }

    this.nextTodoId = 0

  }

Redux

## Why Redux?

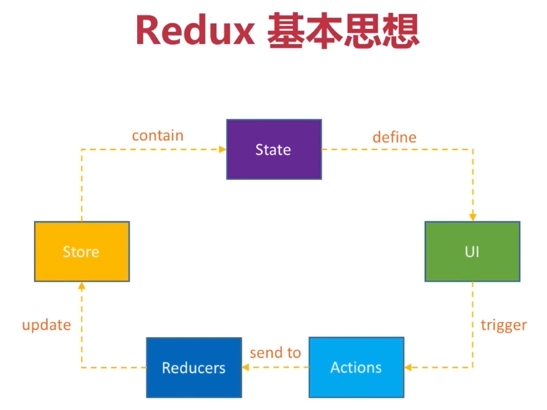
* 复杂的状态：API数据，本地数据，UI状态等

## Redux核心概念

Redux在任何地方都可用，不只是运用于React

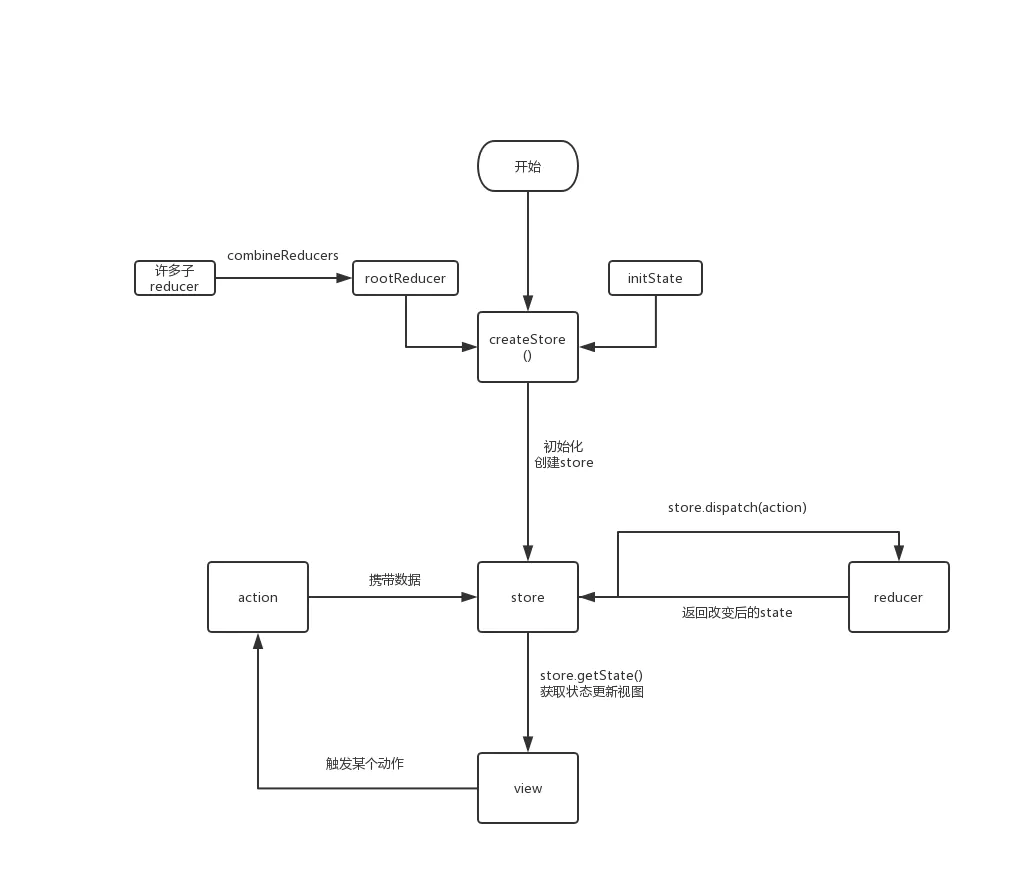
Redux用于state的管理，管理的是在整个网站各个地方都通用的state，但是不去管理那些只作用于组件自身的state

Redux的核心概念其實很簡單：store裡存放state：项目中所有的state都以一个对象树的形式存储在一个单一的store中。需要修改state時，發起一個action用來描述發生了什麼，用reducers描述action如何改變state tree。



創建store的時候需要傳入reducer，真正能改變store中資料的是store.dispatch API。

下圖來源：<https://blog.csdn.net/qq_39261142/article/details/107310638> （講了redux和react-redux）



## 設計Redux的State

Redux的State是應用（網站/APP）的狀態，其概念基本等同於React組件中的State。

|  |  |
| --- | --- |
| **Redux的State** | **React組件中的State** |
| 集中管理，全局唯一 | 不同組件，可以有自己的State |
| 不可變。不能直接修改原來的State。要創建一個新的State，再修改。 | |
| 把不同React組件中的State集中放到Redux的State中。 |  |

## Redux的Action

Action 是把資料從應用（譯者注：這裡之所以不叫 view 是因為這些資料有可能是伺服器回應、用戶輸入或其它非 view 的資料 ）傳到 store 的有效載荷。它是 store 資料的唯一來源。一般來說你會通過 store.dispatch() 將 action 傳到 store。

Action 本質上是 JavaScript 的JSON普通對象。我們約定，action 內必須使用一個字串類型的 type 字段來表示將要執行的動作。多數情況下，type 會被定義成字符串常量。當應用規模越來越大時，建議使用單獨的模塊或文件來存放 action。

除了 type 字段外，action 對象的結構完全由你自己決定。參照 Flux 標準 Action 獲取關於如何構造 action 的建議。

這時，我們還需要再添加一個 action index 來表示用戶完成任務的動作序號。因為資料是存放在數組中的，所以我們通過下標 index 來引用特定的任務。而實際項目中一般會在新建數據的時候生成唯一的 ID 作為數據的引用標識。

我們應該儘量減少在 action 中傳遞的數據。比如上面的例子，傳遞 index 就比把整個任務對象傳過去要好。

下面是一些 action 的示例：

{ type: 'ADD\_TODO', text: 'Go to swimming pool' }

{ type: 'TOGGLE\_TODO', index: 1 }

{ type: 'SET\_VISIBILITY\_FILTER', filter: 'SHOW\_ALL' }

再如：

var addTodoAction = {

  type: 'ADD\_TODO',

  id: 1,

  text: '学习react'

}

但在實際開發中，會寫成函數的形式（叫“action creator”）來返回一個Action的JSON對象：

const addTodo = () => ({

  type: 'ADD\_TODO',

  id: 1,

  text: '学习react'

})

習慣上，type的命名和action creator函數的名字一致。

實際開發的時候，不會把action的所有參數都寫死。把addTodo 導出export出去，方便其它模塊引用。

let nextTodoId = 0;

export const addTodo = (*text*) => ({

  type: 'ADD\_TODO',

  id: nextTodoId++,

  text

})

### es6-箭頭函數返回一個對象

方法一，用return

      const getJsonObj1 = (*a*) => {

        return { data: a };

      };

      console.log(getJsonObj1(111));

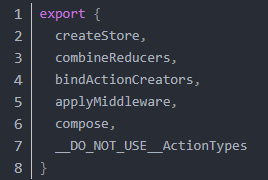
方法二，在對象外側加小括號

      const getJsonObj2 = (*b*) => ({ data: b });

      console.log(getJsonObj2(222));

### Redux的方法

redux只對外暴露了以下方法，除了這些都是react-redux的內容



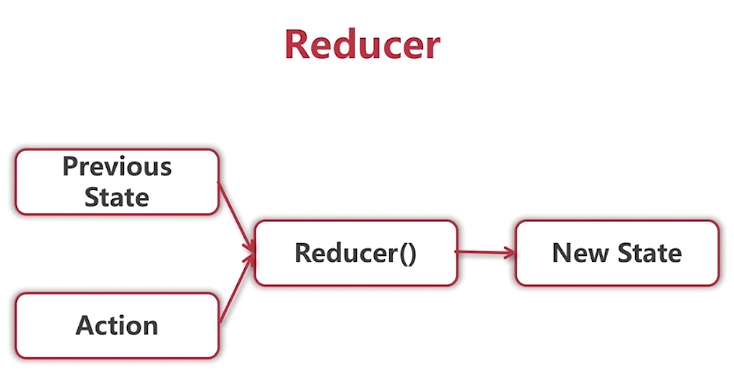
## Reducer

### 理解Reducer

把Reducer理解成Action的解析器

Action描述了如何去修改State

Reducer把這種描述落實到代碼層面



Reducer 函數只是一個純函數，它接收應用程序的當前狀態以及發生的 action，然後返回修改後的新狀態（或者有人稱之為歸並後的狀態）。

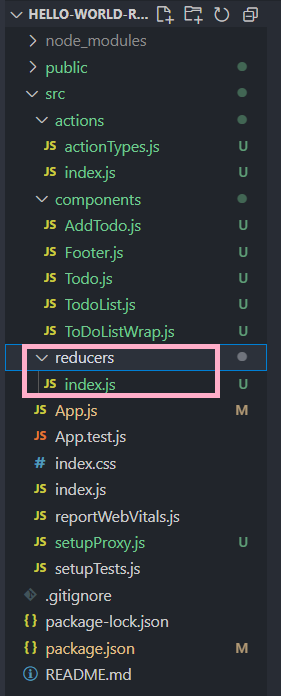
### Reducer代碼

reducer 只是一個接收 state 和 action、並返回新的 state 的函數。

對於大的應用來說，不大可能僅僅只寫一個這樣的函數，所以我們編寫很多小函數來分別管理 state 的一部分：

這裏有個常見模式：在 reducer 裏用 switch 來響應對應的 action 。用 switch 的時候， \*\*永遠\*\* 不要忘記放個 「default」 來返回 「state」

示例代碼



import { ADD\_TODO, TOGGLE\_TODO, SET\_TODO\_TEXT, SET\_FILTER } from "../actions/actionTypes"

// 应用启动时给state赋初始值

const initialState = {

  filter: 'all',

  text: '',

  todos: []

}

// state = initialState保证应用在初始化的时候可有值

const todoApp = (*state* = initialState, *action*) => {

  switch (action.type) {

    case ADD\_TODO:

      return {

        ...state,

        todos: [

          ...state.todos,

          {

            id: action.id,

            text: action.text,

            completed: false

          }

        ]

      }

    case TOGGLE\_TODO:

      return {

        ...state,

        todos: state.todos.map(*todo* => {

          return todo.id === action.id ? {

            ...todo, completed: !todo.completed

          } : todo

        })

      }

    case SET\_TODO\_TEXT:

      return {

        ...state,

        text: action.text

      }

    case SET\_FILTER:

      return {

        ...state,

        filter: action.filter

      }

    default:

      return state

  }

}

export default todoApp

### Reducer拆分

拆分邏輯：根據state拆分

比如todolist項目

const initialState = {

  filter: 'all',

  text: '',

  todos: []

}

State是filter，text，dotos三個部分

所以Reducer也可以拆成這三個部分。每個分state對應一個分reducer

然後利用redux中的combineReducers來合併各個reducer

src/reducers/index.js代碼

import { combineReducers } from 'redux';

import todos from './todos';

import filter from './filter';

import text from './text';

export default combineReducers({

  todos,

  text,

  filter

})

src/reducers/todos.js代碼

import { ADD\_TODO, TOGGLE\_TODO } from "../actions/actionTypes"

const todos = (*state* = [], *action*) => {

  switch (action.type) {

    case ADD\_TODO:

      return [

        ...state,

        {

          id: action.id,

          text: action.text,

          completed: false

        }

      ]

    case TOGGLE\_TODO:

      return state.map(*todo* => {

        return todo.id === action.id ? {

          ...todo, completed: !todo.completed

        } : todo

      })

    default:

      return state

  }

}

export default todos

src/reducers/filter.js代碼

import { SET\_FILTER } from "../actions/actionTypes"

const filter = (*state* = 'all', *action*) => {

  switch (action.type) {

    case SET\_FILTER:

      return action.filter

    default:

      return state

  }

}

export default filter

src/reducers/text.js代碼

import { SET\_TODO\_TEXT } from "../actions/actionTypes"

const text = (*state* = '', *action*) => {

  switch (action.type) {

    case SET\_TODO\_TEXT:

      return action.text

    default:

      return state

  }

}

export default text

### 保證reducer純凈

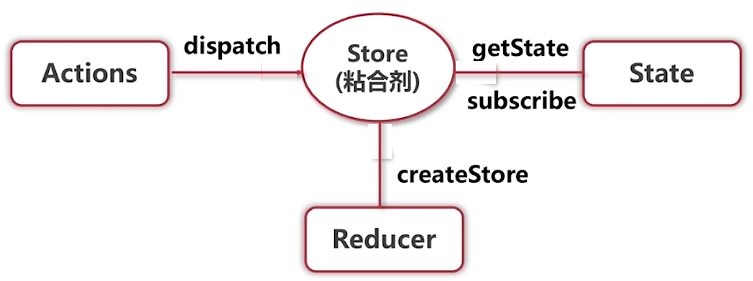
不可在reducer裏執行如下操作：

* 修改傳入參數
* 執行有副作用的參數，如API請求和路由跳轉
* 調用非純函數，如Data.now()或Math.random()

只要傳入參數相同，返回計算得到的下一個state就一定相同，沒有特殊情況，沒有副作用，沒有API請求，沒有變量修改，單純執行計算

reducer的最終目的只是接受一個條件（action）並根據條件修改state，其他不是以修改state為目的的操作應該都在外部操作

## Redux的Store



* Redux提供createStore這個API，接收Reducer作為參數，來創建Store

注意，下面的createStore在新版中不再使用

import { createStore } from 'redux'

而用

import { configureStore } from '@reduxjs/toolkit';

import rootReducer from './reducers'

const store = configureStore({

  reducer: rootReducer

})

configureStore()接收一個對象。對象屬性為reducer，值為根reducer

* Store對象提供了dispatch這個API，來發送action
* Store有getState這個API可以獲得全局State；可以通過subscribe這個API獲得全局State的變化



* store.getState( )

返回當前應用的state樹，與store的最後一個reducer返回值相同，註意，這個state是所有reducer傳入的state，即獲取到了所有全局的state的值

* store.dispatch(action)

觸發state變化的唯一途徑，會使用當前getState()和傳入的action以同步方式調用store的reducer函數，返回值會被作為下一個state，成為getState()的新返回值，同時變化監聽函數（change listener）被觸發

如果action是一個函數，應該去執行，即store.dispatch(action())，因為執行了創建方法這樣才能返回一個action對象

action在聲明reducer時並沒有將具體的action傳入。真正將具體的action傳入reducer是在調用dispatch(action)的時候

* store.subscribe(listener)

每當執行store.dispatch(action)時便會自動執行此方法，需要操作當前state時可用store.getState()獲取當前state

### store示例代碼

2022年6月筆記（react版本 "^18.2.0"）

src/store.js

import { configureStore } from '@reduxjs/toolkit';

import rootReducer from './reducers'

import { addTodo, toggleTodo, setFilter, setTodoText } from './actions'

// 創建state

const store = configureStore({

  reducer: rootReducer

})

console.log(store)

// 订阅state的变化。這個監聽必須寫在dispatch的前面，否則dispatch後無法及時得知state的變化

const unsubscribe = store.subscribe(() => {

  console.log(store.getState())

})

store.dispatch(addTodo('Learn about actions'))

store.dispatch(toggleTodo(0))

store.dispatch(setFilter('active'))

store.dispatch(setTodoText('Learn'))

//取消订阅

unsubscribe()

export default store

打印結果：State總共打印了5次



還有個例子（以前的createStore的寫法）



## 觸發dispatch的方法

### （1）store.dispatch()直接发送

一般來說你會通過 store.dispatch() 將 action 傳到 store。

唯一改變 state 的方法就是觸發 action，action 是一個用於描述已發生事件的普通對象。

store.dispatch({

type: 'COMPLETE\_TODO',

index: 1

})

store.dispatch({

type: 'SET\_VISIBILITY\_FILTER',

filter: 'SHOW\_COMPLETED'

})

### （2）通过react-redux

把dispatch作为参数传递给mapDispatchToProps函数

// 第一个参数用于接受store.dispatch()方法（可以为其它变量名）,第二个参数用于接受组件自身的props

const mapDispatchToProps = (dispatch, ownProps) => {

return {

add: (id) => { dispatch(increment(id)) },

dec: (id) => { dispatch(decrement(id)) }

}

}

上面的代碼中，increment(id)和decrement(id)都是函數類型的action

### （3）通过bindActionCreators

是通过dispatch将action包裹起来，这样可以通过bindActionCreators创建的方法，直接调用dispatch(action)(隐式调用），相当于直接组合成

dispatch({type:type.ADD\_ITEM, text})

这种形式。  
用法：

let newAction = bindActionCreators(oldActionCreator,dispatch)

<child {...newAction}/>

以上代码将dispatch和action组合成的对象直接传给了子组件，然后在子组件中就可以通过调用 newAction.action1就相当于实现了dispatch（action1）  
由此便实现了在没有store和dispatch的组件中调用dispatch（action1）

## 一旦刷新頁面，Redux中存儲的數據會被清空。

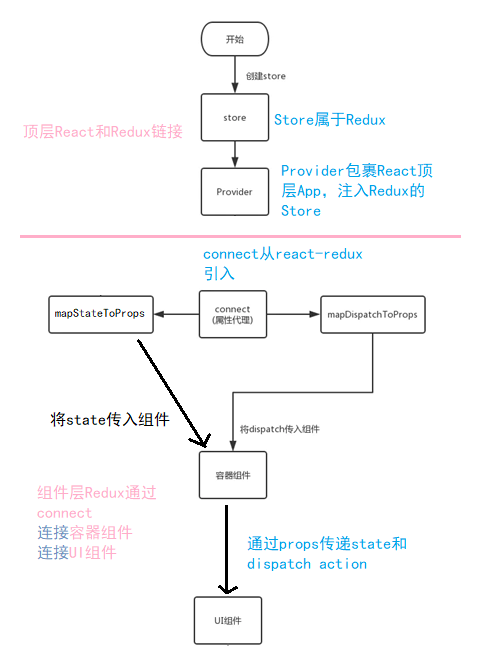
如果用localStorage.setItem儲存在本地，刷新時就不會清空數據。

React-Redux

如果單獨用Redux，需要視圖層自己去監聽State的變化。

React提供的React-Redux封裝了所有的監聽過程。借助react-redux可以很方便的在react中使用redux

實際項目中，需要權衡是直接使用Redux還是用React-Redux



## React-Redux作用

1）向根組件注入Store，通過Provider 組件實現

2）連接React組件和Redux狀態層 ，通過 connect組件實現

3）獲取React組件所需的State和Actions，通過map api實現

## 展示型組件和容器型組件

React-Redux 將所有組件分成兩大類：UI 組件（presentational component）和容器組件（container component）



如果一個UI組件需要和Store交互，就創建一個相應的容器型組件。

比如TODOList項目中，components中的3個UI組件需要和Store交互，就在containers文件夾中創建了3個相應的容器型組件。



## 容器型組件

Redux提供的connect方法可以輔助創建容器型組件

Connect本質是個高階組件。

高階組件是一個函數（而不是組件），它接受一個組件作為參數，返回一個新的組件。這個新的組件會使用你傳給它的組件作為子組件。

通過connect( )( )自動生成的容器組件（高階組件），經過connect操作後會將dispatch方法傳入該組件

使用了connect之後是自動訂閱state的變化並進行重新渲染的，不需要再去通過store.subscribe(listener)去監聽state的變化

函數mapStateToProps()是把Redux中的State映射到React組件中的Props上。

函數mapDispatchToProps()是把Redux中的Action方法映射到React組件的Props上。

mapDispatchToProps()返回的是對象。對象的每個屬性都是函數，映射到Redux的dispatch上。

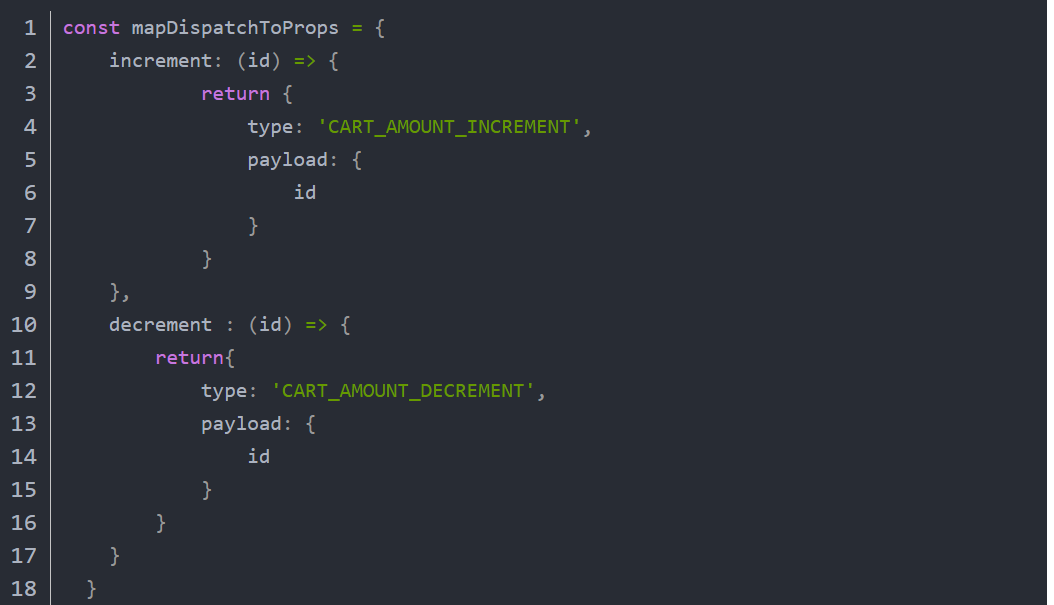
mapDispatchToProps()它可以是一個函數，也可以是一個對象。

mapDispatchToProps()為一個函數時：則本來應使用this.props.dispatch()方法，現在因為有了映射，可以直接this.props.add()和this.props.dec()執行對應的reducer功能。





mapDispatchToProps()為一個對象時，鍵值內容應為一action對象或action創建函數，會主動識別傳入的action並自動調用dispatch(action)方法並將返回值映射到props，此時調用dispatch方法時則使用this.props.increment()和this.props.decrement()，而非this.props.dispatch(this.props.increment())



示例代碼

### （1）構建容器型組件

TodoListContainer.js

import { connect } from 'react-redux';

import TodoList from '../components/TodoList'

import { toggleTodo } from '../actions'

const getVisibleTodos = (*todos*, *filter*) => {

  switch (filter) {

    case 'all':

      return todos

    case 'completed':

      return todos.filter(*t* => t.completed)

    case 'active':

      return todos.filter(*t* => !t.completed)

    default:

      return new Error('Unknown filter:' + filter)

  }

}

const mapStateToProps = (*state*) => ({

  todos: getVisibleTodos(state.todos, state.filter)

})

const mapDispatchToProps = (*dispatch*) => ({

  toggleTodo: *id* => dispatch(toggleTodo(id))

})

export default connect(mapStateToProps, mapDispatchToProps)(TodoList)

其它兩個容器型組件進行類似改造

### （2）在ToDoListWrap.js中引入容器型組件

（ToDoListWrap.js以前是直接引入UI組件的）

import React, { Component } from 'react';

import AddTodoContainer from '../containers/AddTodoContainer';

import TodoListContainer from '../containers/TodoListContainer';

import FooterContainer from '../containers/FooterContainer';

class ToDoListWrap extends Component {

  render() {

    return (

      <div>

        <AddTodoContainer />

        <TodoListContainer />

        <FooterContainer />

      </div>

    );

  }

}

export default ToDoListWrap;

### （3）UI型組件中通過props獲取state和dispatch action

AddTodo.js代碼

import React, { Component } from 'react';

class AddTodo extends Component {

  render() {

    return (

      <div>

        <input value={this.props.text} onChange={this.handleChange} />

        <button onClick={this.handleClick}>Add</button>

      </div>

    );

  }

  handleChange = (*e*) => {

    this.props.setTodoText(e.target.value)

  }

  handleClick = () => {

    this.props.addTodo(this.props.text)

    this.props.setTodoText('')

  }

}

export default AddTodo;

Footer.js中的代码

import React, { Component } from 'react';

class Footer extends Component {

  render() {

    // 把FooterContainer中的setFilter重命名为setListFilter

    const { filter, setFilter: setListFilter } = this.props;

    return (

      <div>

        <span>Show:</span>

        <button disabled={filter === "all"} onClick={() => setListFilter("all")}>All</button>

        <button disabled={filter === "active"} onClick={() => setListFilter("active")}>Active</button>

        <button disabled={filter === "completed"} onClick={() => setListFilter("completed")}>Completed</button>

      </div>

    );

  }

}

export default Footer;

### （4）在React的根目錄中引入store和provider

下面引入的store代碼見上文：Redux的store – store示例代碼 - src/store.js

import store from './store'

import { Provider } from 'react-redux'

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <React.StrictMode>

    <Provider store={store}>

      <App />

    </Provider>

  </React.StrictMode>

);

由於<App/>為組件渲染根元素，因此任意子組件內都可直接使用Provider傳遞的數據，無需層層相傳，Provider必須要擁有store屬性，值為創建的store。

## 低層級組件connect React和Redux

連接React和Redux時，盡量在低層級的組件連接，以避免不必要的渲染。

原理：

比如UI組件TodoList通過容器組件TodoListContainer和Redux連接。

const mapStateToProps = (*state*) => ({

  todos: getVisibleTodos(state.todos, state.filter)

})

只有todos這個state發生變化，TodoList才會重新渲染

而state.filter變化，不會導致TodoList重新渲染

Redux-Thunk

redux-thunk是作為redux的 middleware 存在的，用法和普通 middleware 的用法是一樣的

## middleware概述

（1）middleware的一個例子：redux-thunk

默認情況下，dispatch只能處理對象類型的action。

redux-thunk讓dispatch方法可以處理函數類型的action。

（2）中間件本質：增強store dispatch的能力。

（3）可以使用多個中間件

图示

描述已自动生成

（4）中間件是個函數

①

上述函数等同于：

*store* => *next* => *action* => {

…

}

把store打印一下

console.log(store)

可以得到：

文本, 信件

描述已自动生成

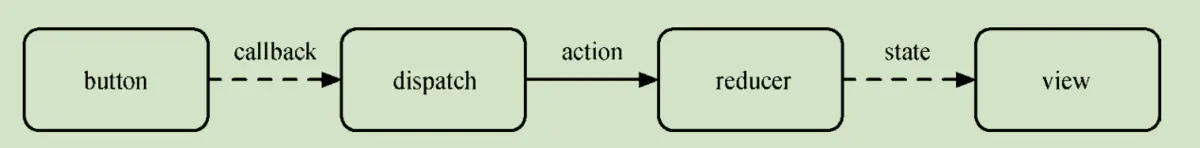
中間件函數首先接受對象作為參數，然後返回一個新函數，新函數接收next作為參數，接下來又返回一個新新函數，新新函數接收action作為參數

②dispatch和next都能發送action。它們的區別：

假設在mid2發送1個action

1. 如果用dispatch發送，會從mid1開始，大家一起處理這個action；
2. 如果用next發送，這個action會直接到達下一個中間件，前面的中間件不會處理這個action。

使用中間件處理前：



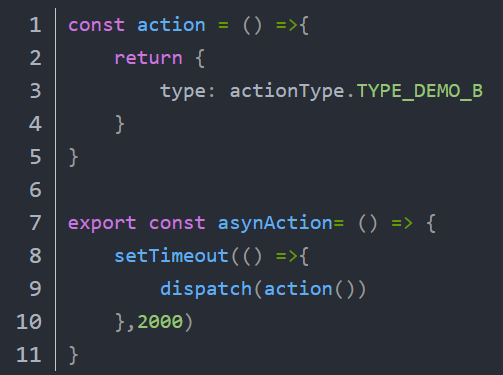
使用中間件處理後：

图示

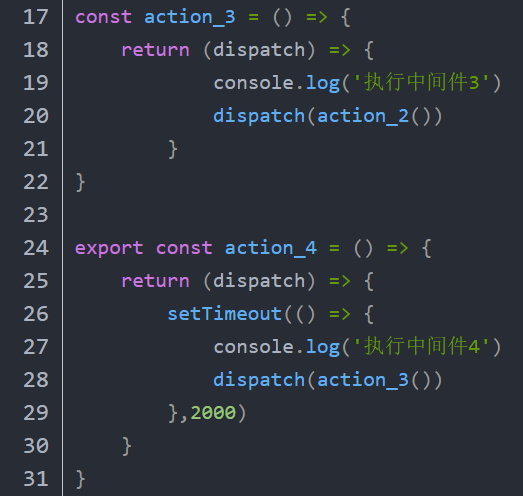
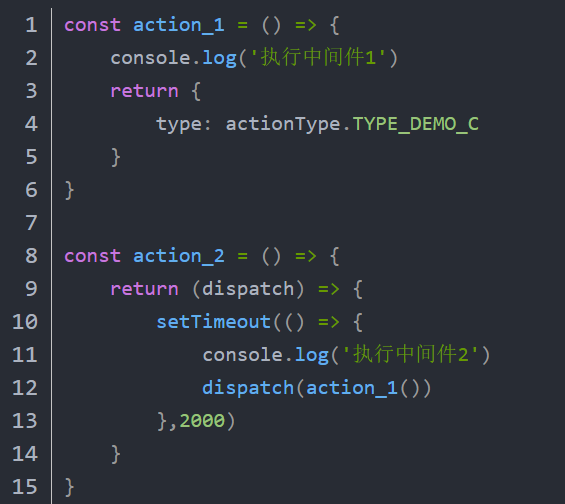
描述已自动生成

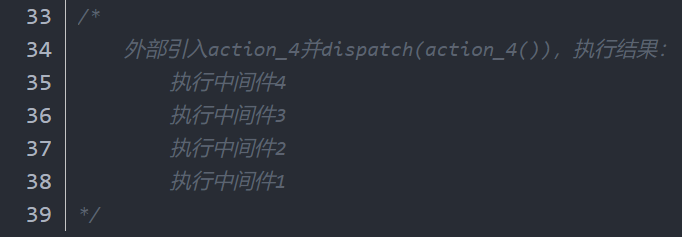
將action傳遞給中間件，每一個中間件可以處理action並返回新的action，最後一個中間件處理完action後才會將最終的action傳遞給reducer處理，因此保證了reducer處理的是異步執行完步後的action，不會因為立即執行dispatch而沒有及時拿到到action報錯

一般而言，會把同步action和異步操作分開，同步action用於定義action.type等數據，而異步只引用同步

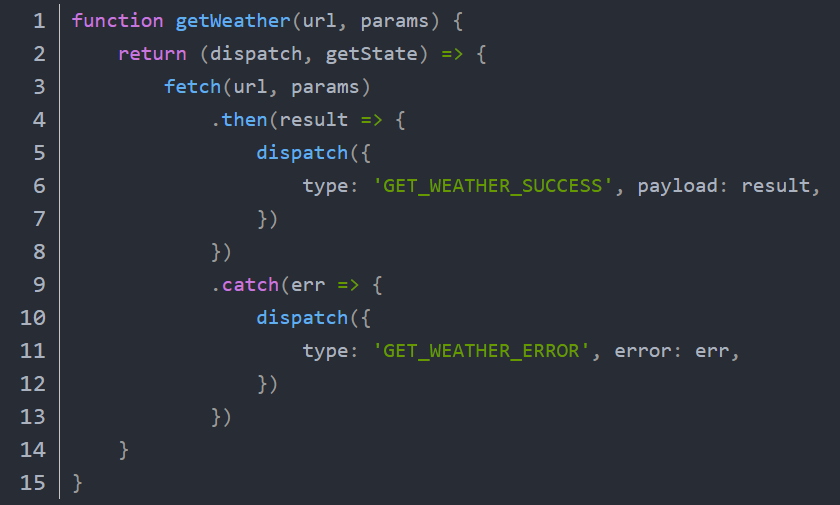


多個中間件處理action的情況：只需導出最外層的中間件並在外部dispatch該中間件即可



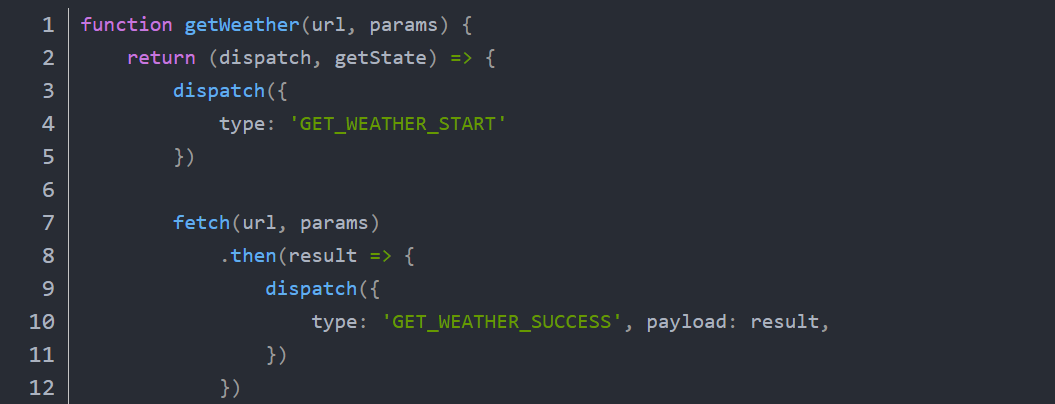


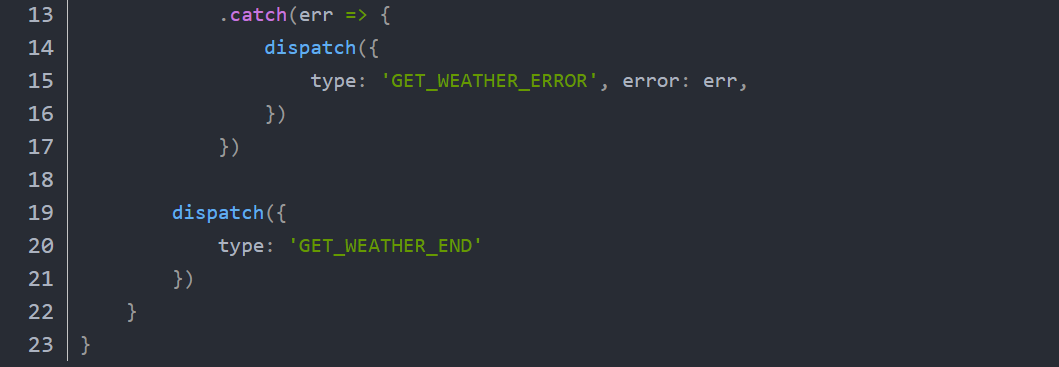
使用redux-thunk在真實開發中實現數據請求的寫法



可以在請求前後另外dispatch一些操作

比如在請求開始前將一個表示加載中的state設置成true，在請求完成時設置成flase，就實現需要在請求加載中時做的一些操作的效果





## middleware的寫法

dispatch一个action之后，到达reducer之前，进行一些额外的操作，就需要用到middleware。你可以利用 Redux middleware 来进行日志记录、创建崩溃报告、调用异步接口或者路由等等。

换言之，中间件都是对store.dispatch()的增强

### 中間件的用法——傳入thunk

import { applyMiddleware, createStore } from 'redux';

import thunk from 'redux-thunk';

const store = createStore(

reducers,

applyMiddleware(thunk)

);

直接将thunk中间件引入，放在applyMiddleware方法之中，传入createStore方法，就完成了store.dispatch()的功能增强。即可以在reducer中进行一些异步的操作。

2022年6月25日更新（此時版本"react": "^18.2.0"）

新版的react不是這麼傳入中間件的

例如，在todolist項目的src/store.js文件裡面

import { configureStore } from '@reduxjs/toolkit';

import rootReducer from './reducers'

import thunkMiddleware from 'redux-thunk';

const store = configureStore({

  reducer: rootReducer,

  middleware: [thunkMiddleware]

})

middleware接受的是個數組中間件

### 中間件的用法——多個參數

其实applyMiddleware就是Redux的一个原生方法，将所有中间件组成一个数组，依次执行。

中间件多了可以当做参数依次传进去

const store = createStore(

reducers,

applyMiddleware(thunk, logger)

);

## Redux-Thunk可以用來處理異步action

Redux store 僅支持同步數據流。使用 thunk 等中間件可以幫助在 Redux 應用中實現異步性。可以將 thunk 看做 store 的 dispatch() 方法的封裝器；我們可以使用 thunk action creator 派遣函數或 Promise，而不是返回 action 對象。

處理異步action 3個時間點：①開始發起數據請求，②數據成功返回，③返回數據出錯。

### TodoList相關示例代碼（）

### public/mock/todos.json

把模擬數據放在這裡，可以模擬向服務器取數據

[

  {

    "id": 100,

    "text": "Learn finance",

    "completed": false

  },

  {

    "id": 101,

    "text": "Learn history",

    "completed": false

  },

  {

    "id": 102,

    "text": "Swim",

    "completed": false

  },

  {

    "id": 103,

    "text": "Walk my puppy",

    "completed": false

  }

]

### src/actions/index.js

fetchTodos這個action是個函數，函數裡面dispatch action去獲取服務器端的todoList數據

import { ADD\_TODO, TOGGLE\_TODO, SET\_TODO\_TEXT, SET\_FILTER, FETCH\_TODO\_REQUEST, FETCH\_TODO\_SUCCESS, FETCH\_TODO\_FAILURE } from "./actionTypes"

const fetchTodoRequest = () => ({

  type: FETCH\_TODO\_REQUEST

})

const fetchTodoSuccess = (*data*) => ({

  type: FETCH\_TODO\_SUCCESS,

  data

})

const fetchTodoFailute = (*error*) => ({

  type: FETCH\_TODO\_FAILURE,

  error

})

// 异步action返回函数而非js对象。此函数可接收store的dispatch方法。

export const fetchTodos = () => {

  return (*dispatch*) => {

    dispatch(fetchTodoRequest());

    fetch("./mock/todos.json", {

      headers: {

        'Content-Type': 'application/json',

        'Accept': 'application/json'

      }

    }).then(

*response* => {

        response.json().then(*data* => {

          dispatch(fetchTodoSuccess(data));

        })

      },

*error* => {

        dispatch(fetchTodoFailute(error));

        console.log("An error occurred:" + error)

      }

    )

  }

}

…

如果報錯

Uncaught (in promise) SyntaxError: Unexpected token < in JSON at position 0，

可以在Fetch取數據時，加上上面黃色高亮的headers參數。

沒報錯就不用加headers。

### src/containers/ TodoListContainer.js

在容器型組件裡把在action中寫好的異步action和UI組件綁定

import { fetchTodos, toggleTodo } from '../actions'

const mapDispatchToProps = (*dispatch*) => ({

  toggleTodo: *id* => dispatch(toggleTodo(id)),

  fetchTodos: () => dispatch(fetchTodos())

})

### src/components/ TodoList.js

UI組件加載完畢時，調用fetchTodos，從服務器獲取已有的todoList數據

class TodoList extends Component {

  componentDidMount() {

    this.props.fetchTodos();

  }

  render() {

…

### src/store.js

把中間件redux-thunk引入store，讓它可以處理異步的action

這是新版react的創建store和插入中間件的的寫法（2020年6月25日，此時版本"react": "^18.2.0"）

import { configureStore } from '@reduxjs/toolkit';

import rootReducer from './reducers'

import thunkMiddleware from 'redux-thunk';

const store = configureStore({

  reducer: rootReducer,

  middleware: [thunkMiddleware]

})

這個store會最終匯入入口文件index.js中（store.js裡面的代碼可以直接寫在入口文件index.js中）

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

import store from './store'

import { Provider } from 'react-redux'

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <React.StrictMode>

    <Provider store={store}>

      <App />

    </Provider>

  </React.StrictMode>

);

// If you want to start measuring performance in your app, pass a function

// to log results (for example: reportWebVitals(console.log))

// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals

reportWebVitals();

## action調用下一個action

大眾點評和treasure store的home.js文件裡調用action來獲取discounts和likes就是這樣的原理。

applyMiddleware

import { applyMiddleware } from 'redux'

作用：加載 middleware，將 action 進行多層組合，並且將dispatch和getState方法傳入到 action 中，使 action 能通過 dispatch 向下調用新的 action或查看當前的state狀態【要配合中間件才能起作用】

const action = () =>{

return {

type: xxx

}

}

const beforeAction = () => {

return (dispatch,getState) => {

console.log(getState()) // 获取state

dispatch(action()) // 手动dispatch调用下一个action

}

}

Redux調試工具：Redux DevTools

## （1）在Chrome瀏覽器中安裝Redux DevTools插件

<https://github.com/zalmoxisus/redux-devtools-extension>



用這個，原因：①便於使用，②不會影響項目結構

這個插件在Chrome和Firefox中都可以使用。

安裝完成後，F12打開開發者工具：



在以前的項目中，需要添加下面黃色的高亮代碼。現在不用了。（2022年6月25日筆記react版本 "^18.2.0"）

官網的初級版本寫法：

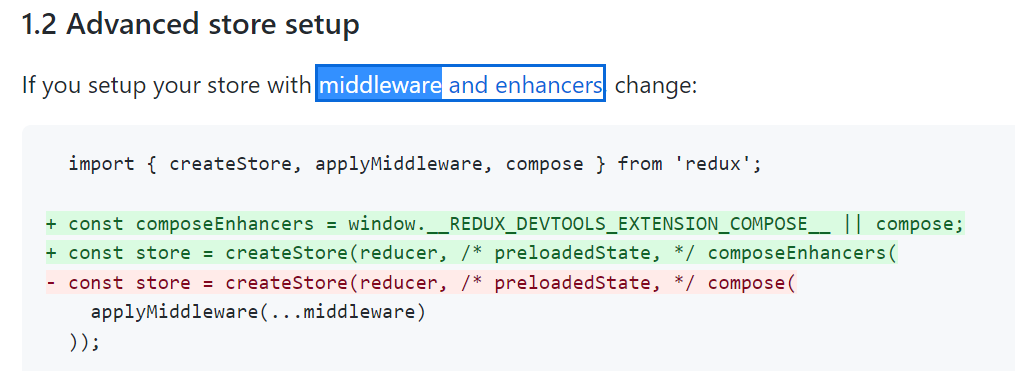
const store = createStore(

reducer, /\* preloadedState, \*/

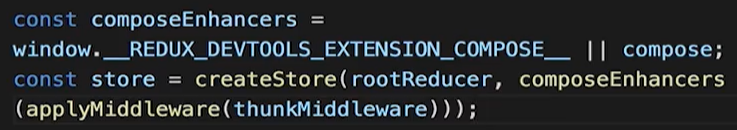
window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()

);

官網的高級版本寫法：



導師的代碼



## （2）在項目的index.js中

<https://github.com/reduxjs/redux-devtools>

① import { compose } from 'redux';

② 添加

const composeEnhancers = window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_ || compose;

const store = createStore(reducer, composeEnhancers(

React項目結構組織方式

## （1）按照類型（官網展示的形式）



項目大了以後，這種結構不方便。

## （2）按照功能模塊



缺點：feature1和feature2的狀態存在耦合情況。項目越複雜，這種可能性越大。

## （3）Ducks（鴨子）（推薦使用這種。在實戰案例React大眾點評中用的這種。）

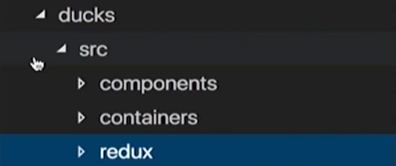
① 出處：<https://github.com/erikras/ducks-modular-redux>

② reducers、action types、actions組合到一個文件中，作為獨立模塊。

③ 劃分模塊依據：應用狀態State，而不是界面功能。同一個應用狀態的actions和reducers都會定義在同一個文件中。

④ 把視圖層和狀態層徹底分開。

### ① components



src/components裡面放置應用級別的通用組件。比如loading，錯誤彈框。

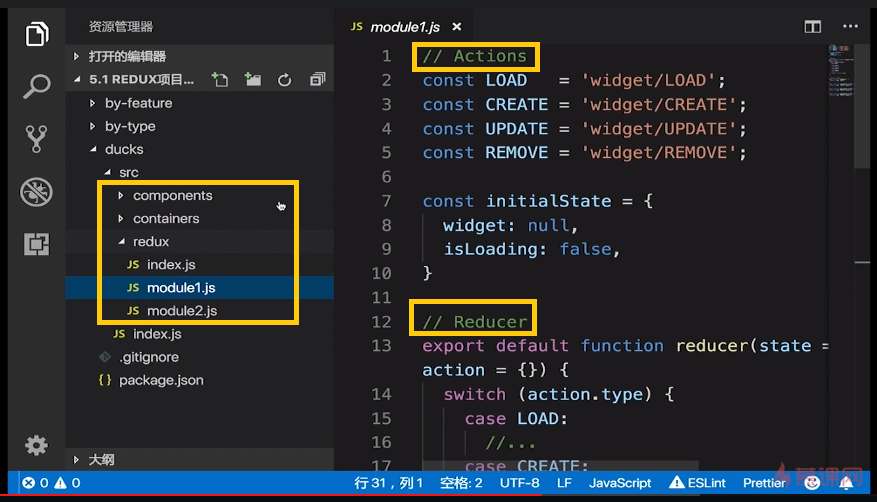
### ② containers

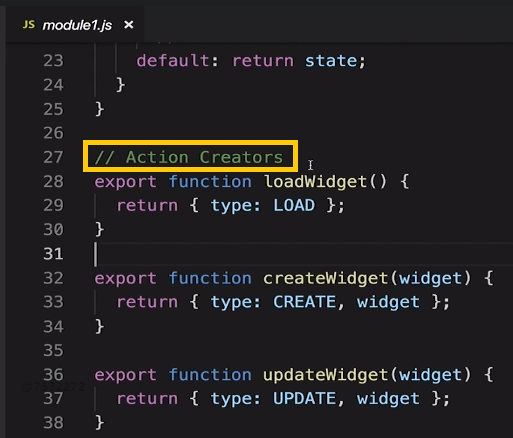
下圖的containers文件夾中有文件夾如feature1，feature2。按照功能模塊來劃分文件夾，每一個功能對應一個文件夾。feature1和feature2下有components文件夾和index.js。feature下的components存放該功能專用的組件。index.js是feature文件夾對外暴露的接口，是個容器型組件。

文本

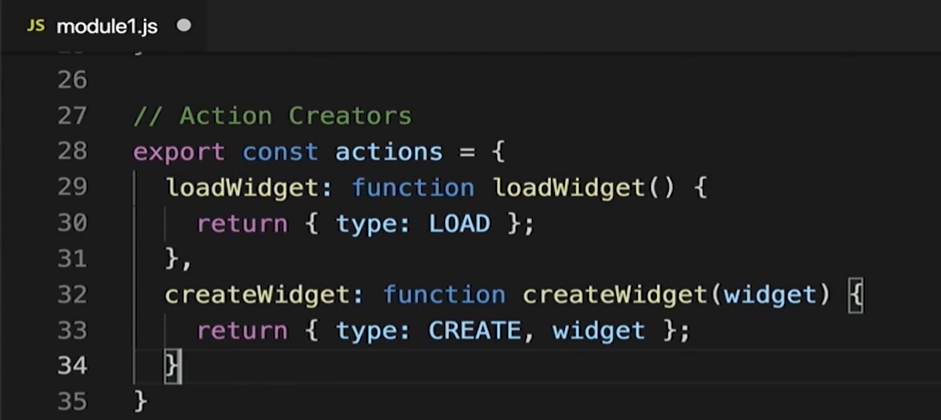
描述已自动生成

### ③ redux



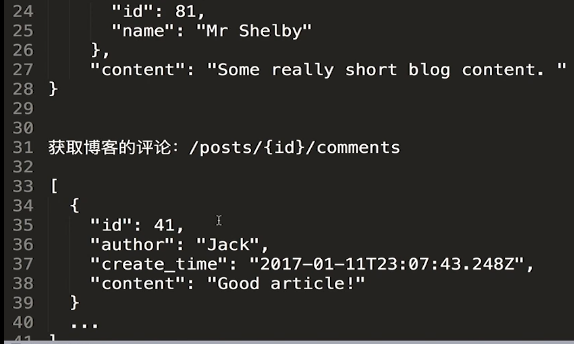
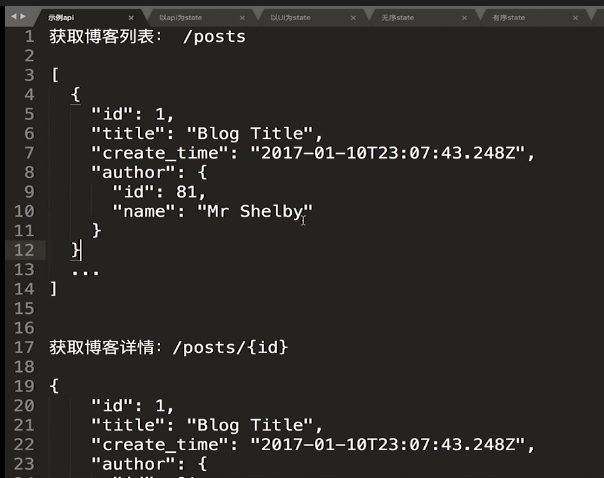


實際開發中，action creators會寫成這樣的對象：



Redux State設計原則

3個數據API



## （1）兩種錯誤：

### ①以API為設計State的依據

以API為設計State的依據，會導致State中出現很多重複數據，取數據時不方便。

錯誤的State設計示範（以API為子State劃分依據）：



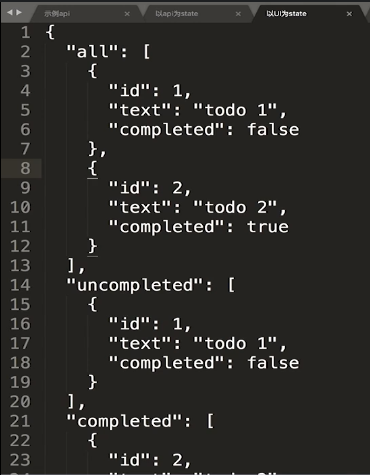
比如上面posts和currentPost中，title、creat\_time、author等字段的內容都是一樣的

還有，posts和currentComments是數組。要查找某個post或者comment的時候，要遍歷數組，會非常不方便。

### ②以頁面UI為設計State的依據

以頁面UI為設計State的依據，會導致State中出現很多重複數據，有不同State部分中數據不一致的風險。

錯誤的State設計示範（以UI為子State劃分依據）：



上面是todoList按照UI劃分的state，有很多重復數據。

而且新增或者修改數據時，需要修改不止一個地方。比如新增一個待辦事項，all和uncompleted下面同時要新增數據，這樣造成存儲浪費，又存在數據不一致的風險。

## （2）Redux State正確設計原則：像設計數據庫一樣設計State

### ① 後台API返回的數據（叫領域數據）

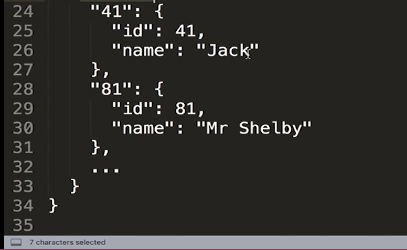
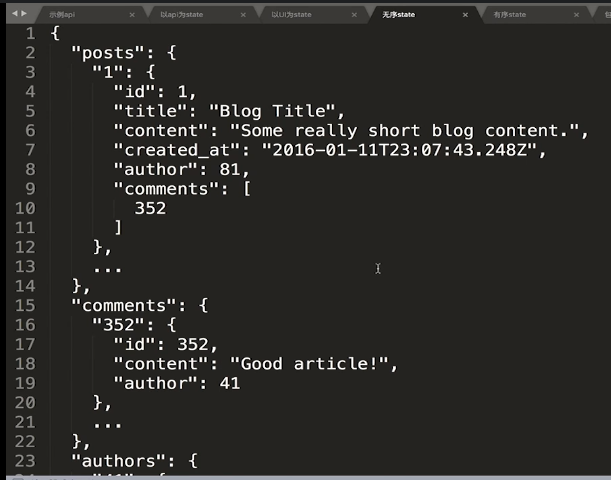
1. 數據按照領域（Domain）分類，存儲在不同的表（子State）中，不同的表（子State）中存儲的列數據不能重複。
2. 表中每一列的數據都依賴於這張表的主鍵。

表中State以鍵值對的結構存儲數據，以記錄的key/ID作為記錄的索引，記錄中的其他字段都依賴於索引。

1. 表中除了主鍵以外的其他列，互相之間不能有直接依賴關係。

State中不能保存可以通過已有數據計算而來的數據，即State中的字段不互相依賴。

正確的State設計示範：

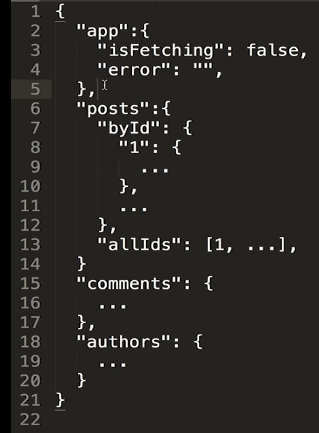


上面的state和後端數據表類比：posts，comments，authors是表名稱。每一條key（比如數字1）和value（數字1對應的對象）就相當於數據表中的一行數據

添加一行代碼，可以保證博文posts有序顯示。postIds中的每個元素，是posts的id值。類比數據庫：一張表的主鍵用作另一張表的外鍵。這裡就是postIds的外鍵引用了posts的主鍵。

### ② 包含UI狀態的State設置：

下面的isFetching是指是否正在請求後台的數據，error是指是否有錯誤



設計reducers時，就能根據子State（app，posts，comments，authors）來設計。

## （3）注意

①State應該盡量扁平化，避免嵌套層級過深，否則不便於訪問State中某一層級的數據，也不便於State數據的擴展。

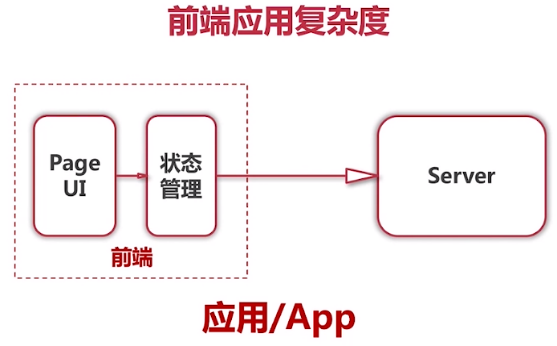
②UI State具有鬆散性特點，所以可以把UI State合併，以避免State一級節點過多。

selector函數

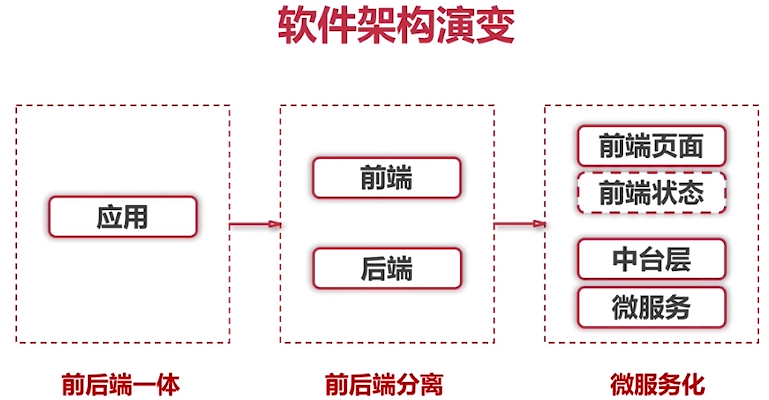
Selector函數幫助React的container組件從Redux State中讀取部分數據，或計算讀取到的數據後再返回給React的container組件。讓視圖層和狀態層解耦。

以TodoListApp為例：在AddTodoContainer中，map text和state時，把text和getText(state)map起來。這樣哪怕以後有多個組件使用text，如果要修改text，只要修改selectors中的text這1個地方即可。

深入理解前端狀態管理思想



前端的狀態管理層把服務器傳來的數據做相應處理後，再給視圖層使用。



微服務提供的數據接口非常具體細小，針對某個領域：比如用戶服務，商品服務。

微服務這種細小的數據接口無法直接給前端使用，於是出現了中台層，來聚合這些微服務，然後提供給前端。

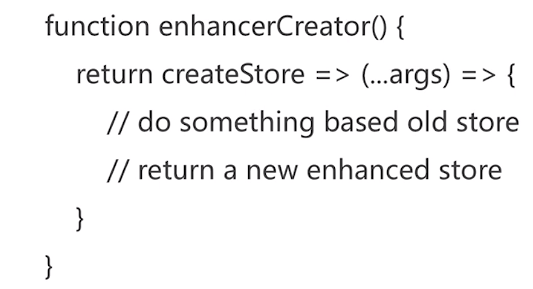
但前端頁面實在變化快，所以當中台層的接口無法直接滿足視圖層需求時，就出現了前端狀態層來處理這些數據。

Store Enhancer

（1） Store Enhancer作用：增強redux store功能

（2）createStore(reducer, [preloadedState], [enhancer]) （其中的enhancer參數即Store Enhancer）

（3）Store Enhancer一般結構



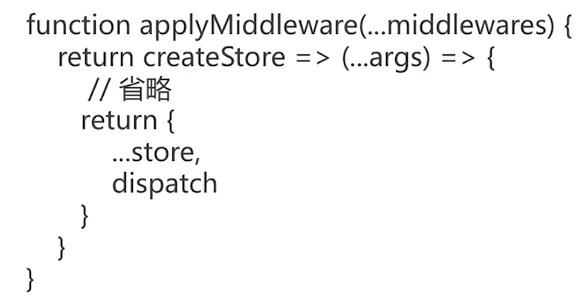
enhancerCreator()返回一個函數，這個函數接收createStore作為參數

(…args)是createStore會使用到的參數（reducer，初始的state等）

（4）Store Enhancer是Middleware的關係

Middleware是Store Enhancer的特例

下面是applyMiddleware的源碼，applyMiddleware返回的就是store enhancer



在日常使用中，盡可能用Middleware，以避免無意中破壞store的緣由邏輯。

常和Redux放在一起使用的庫：Immutable.js

## Immutable.js介绍

（1）Immutable.js是用來操作不可變對象的JS庫

Redux不可修改原有的state。若有修改，必須創建新state。當state層級較深時，用es6…state的語法來寫，就很麻煩。所以immutable.js應運而生。

（2）immutable.js在GitHub上的地址：https://github.com/facebook/immutable-js

yarn add immutable

（3）immutable的API可創建不同層級state的不可變對象

①Immutable.fromJS()可將js對象創建為不可變對象

②merge可以把新對象merge到原不可變對象上，merge後的對象也是不可變對象。

getIn逐層遍歷對象每一層級的屬性，即獲得嵌套對象的值。它接受的是一個數組為參數。數組的每個值代表由外到內每一層級屬性的名稱。

（4）在TodoList練習項目中，修改reducers、selectors、containers這些文件夾內的文件

①reducers的index中，redux提供的combineReducers只能識別普通的js對象，現在子reducer返回的都是immutable類型的對象，combine時，得用從redux-immutable中

Import的 { combineReducers }

②containers中map的時候，這裡取出來的是個immutable對象，得轉換為普通js對象才能使用

## 更高效地使用Immutable

（1）不高效的地方

toJS在state沒有變化的情況下，每次也會重新返回對象，被connect捕捉到後，會判定state發生了變化，進而重新渲染組件。

const mapStateToProps = (*state*) => ({

    todos: getVisibleTodos(state).toJS()

    //這裡取出來的是個immutable對象，得轉換為普通js對象才能使用

})

（2）解決方式：

在高階組件中完成Immutable對象到普通對象的轉換

創建HOCs/toJS.js文件

高階組件就是一个函数，它接收react組件為參數，返回一個新的react組件

Object.entries()用法示例

**示例1 -- 传入对象**

const obj = { foo: 'bar', baz: 'abc' };

console.log(Object.entries(obj)); // [['foo', 'bar'], ['baz', 'abc']]

**示例2.1 -- 数组**

const arr = [1, 2, 3];

console.log(Object.entries(arr)); // [['0', 1], ['1', '2'], ['2', '3']]

**示例2.2 -- 数组（数组中包含对象）**

const arr1 = [{ a: 1 }, 2, 3];

console.log(Object.entries(arr1)); // [['0', { a: 1 }], ['1', '2'], ['2', '3']]

**示例2.3 -- 数组（数组中的值全部为对象）**

const arr2 = [{ a: 1 }, { b: 2 }, { c: 3 }];

console.log(Object.entries(arr2)); // [['0', { a: 1 }], ['1', { b: 2 }], ['2', { c: 3 }]]

**示例3 -- 字符串**

const str = '123';

console.log(Object.entries(str)); // [['0', '1'], ['1', '2'], ['2', '3']]

**示例4 -- 数字、浮点数**

const num = 123;

console.log(Object.entries(num)); // []

const float1 = 12.3;

console.log(Object.entries(float1)); // []

**示例5 -- 将 Object 转化为 Map**

new Map() 构造函数接受一个可迭代的 entries 。 借助 Object.entries 方法你可以很容易的将 Object 转换为 Map:

const obj2 = { foo: 'bar', baz: 'abc' };

console.log(Object.entries(obj2)); // [['foo', 'bar'], ['baz', 'abc']]

const map = new Map(Object.entries(obj2));

console.log(map); // Map {'foo' => 'bar', 'baz' => 'abc'}

Array.reduce()方法解析

**语法：**

array.reduce(function(accumulator, currentValue, currentIndex, array), initialValue)；

accumulator：上一次调用回调返回的值，或者是提供的初始值（initialValue）。如果没有设置初始值，则将数组中的第一个元素作为初始值。

currentValue：数组中正在处理的元素。

currentIndex：数据中正在处理的元素索引，如果提供了 initialValue ，从0开始；否则从1开始。

array： 调用 reduce 的数组。

initialValue：可选项，其值用于第一次调用 callback 的第一个参数。如果没有设置初始值，则将数组中的第一个元素作为初始值。**空数组调用reduce时没有设置初始值将会报错。**

**应用：**

比如求和或者需要连续操作数组中的每个元素获得最终结果，for循环可以解决的可以尝试使用reduce方法来做。

1. 求和

1. **var** res = [
2. 0, 1, 2, 3, 4
3. ].reduce((
4. accumulator, currentValue, currentIndex, array
5. ) => {
6. **return** accumulator + currentValue;
7. });

 callback 被调用四次，每次调用的参数和返回值如下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| callback | accumulator | currentValue | currentIndex | array | return value |
| first call | 0 | 1 | 1 | [0,1,2,3,4] | 1 |
| second call | 1 | 2 | 2 | [0,1,2,3,4] | 3 |
| third call | 3 | 3 | 3 | [0,1,2,3,4] | 6 |
| fourth call | 6 | 4 | 4 | [0,1,2,3,4] | 10 |

2. 求最大值

1. **var** res = [1,2,3,4,5].reduce((pre, curr) => {
2. **return** curr > pre ? curr : pre;
3. //return Math.max(pre,curr);
4. });

3. 将二维数组转化为一维数组

1. **var** flattened = [[0, 1], [2, 3], [4, 5]].reduce((a, b) => {
2. **return** a.concat(b);
3. });

Reselect

（1）Reselect作用：減少state重複計算。

比如下圖是TodoList項目的selectors/index.js。當state發生變化，每一個函數都會被觸發，這樣導致性能浪費。

getText和getFilter直接從state中獲取數據，無需用reselect改造。值得用reselect改造的是getVisibleTodos這樣需要通過計算得到結果的複雜函數。



（2）地址：<https://github.com/reduxjs/reselect>

Install: yarn add reselect

React Router

## （1）React Router：全新的React組件式的動態路由庫

## （2）MPA和SPA

①MPA（Multiple Page App）：多頁面應用

特點：服務器根據不同URL返回不同的HTML頁面。依賴服務器端路由的功能。

優點：比SPA更有利於SEO

缺點：頁面切換沒有SPA流暢

②SPA（Single Page App）：單頁面應用

特點：即便URL不同，服務器返回的HTML頁面一樣。依賴瀏覽器端路由的功能，比如React Router

## （3）React Router相關庫

① react-router 實現路由核心功能

② react-router-dom 把路由功能和外部API綁定

③ react-router-native 是react-router和react-native之間的綁定

## （4）<Router>組件

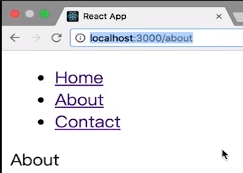
在所有組件最外層包裹<Router>組件，以保證路由功能正常工作。

常用的<Router>具體實現：<BrowserRouter>和<HashRouter>

## （5）<BrowserRouter>

①依據HTML5 history API （pushState，replaceState等）

②需要Web服務器額外配置

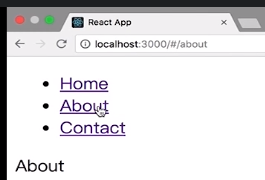


以上圖為例。如果不額外配置服務器，用戶直接輸入localhost:3000/about，服務器回去根目錄下找about文件夾。而事實上，這樣的文件夾是不存在的。結果會出現404。

配置了Web服務器後，可以保證不論用戶訪問的是那個分地址，都能正確返回根目錄下的HTML，並按需渲染給用戶。

## （6）<HashRouter>

使用url的hash部分作為路由信息。例子如下：



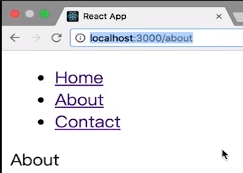
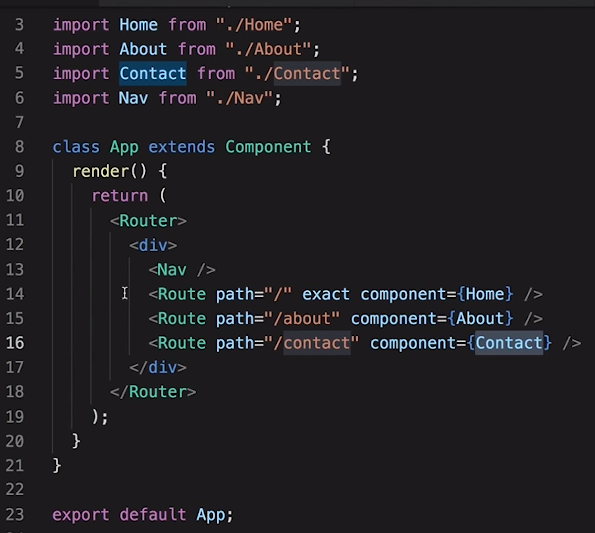
主要為了兼容老版本的瀏覽器

如上所述：BrowserRouter依據HTML5 history API來設計，而老版本瀏覽器不支持HTML5，所以<HashRouter>應運而生。

但目前老版本的瀏覽器很少了，且BrowserRouter的網址更容易理解，所以在大部分項目中，用BrowserRouter。

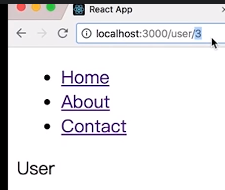
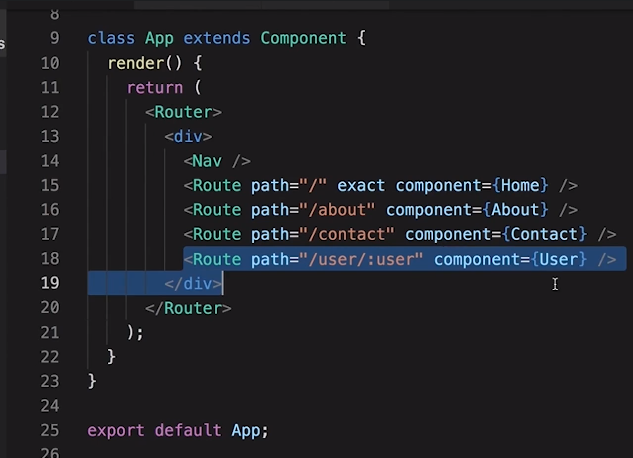
## （7）Route

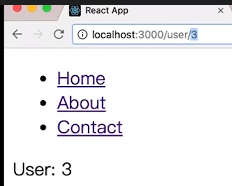
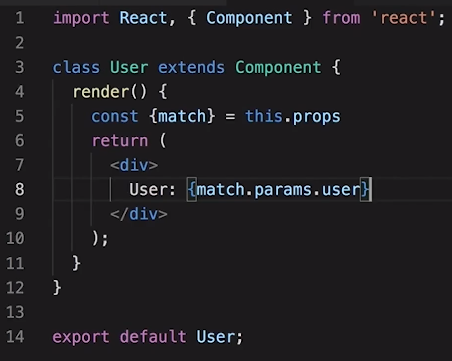
①path



②match

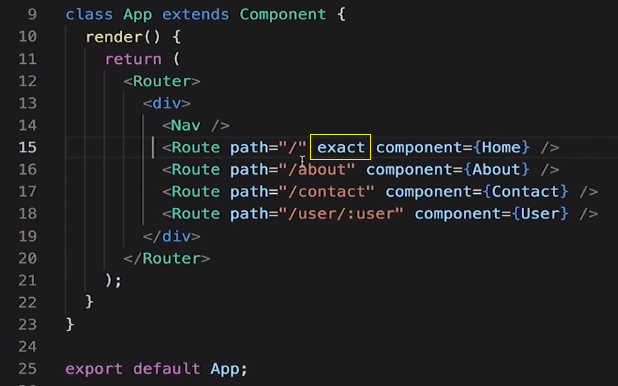
如下圖，不論冒號後面是什麼內容，都會顯示user組件





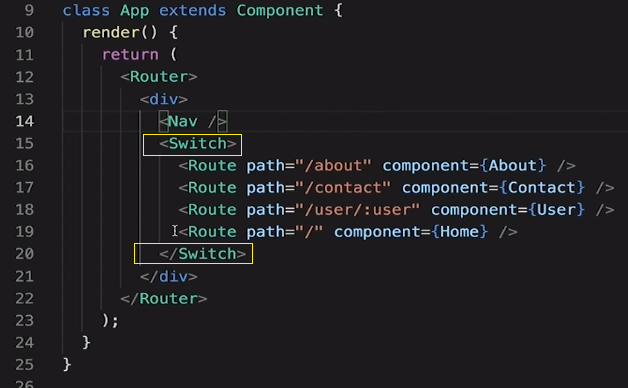
## （8）路由匹配

exact：若在route中定義了exact屬性，那麼只有當URL和route定義的路徑完全相同時，才會匹配。



<Switch>

被switch包裹後，路由在匹配上第一個後，就不再做後續匹配

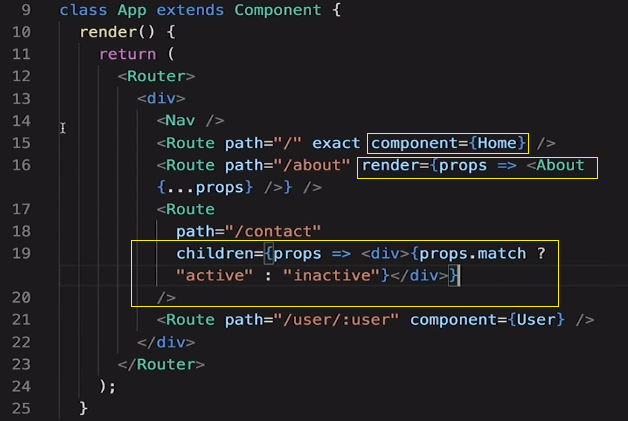


## （9）<Route>渲染組件的方式

① <Route component>

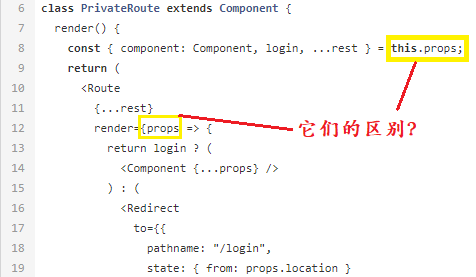
② <Route render>

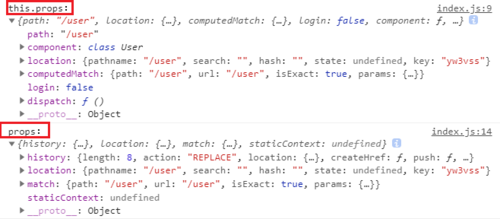
③ <Route children>



运用例子1：

在PrivateRoute中，this.props和props不是一个对象。它们之间没有关系，第一个是组件的属性：props，第二个是一个函数的参数，只不过参数名叫props。





运用例子2：

StackOverFlow上的一篇文章：

<https://stackoverflow.com/questions/51226685/reactjs-component-vs-render-in-route>

Reactjs - `component` vs `render` in Route

I have two doubts regarding usage of Route from react-router-dom(**v4.3.1**):

1. When do we use component vs render in Route:
2. <Route exact path='/u/:username/' component={ProfileComponent} />

<Route exact path='/u/:username/' render={() => <ProfileComponent />} />

1. How to access the variable username in the **URL** in both ways?

asked Jul 7 '18 at 20:18

11

When you pass a component to the component prop, the component will get the path parameters in the props.match.params object, i.e props.match.params.username in your example:

class ProfileComponent extends React.Component {

render() {

return <div>{this.props.match.params.username}</div>;

}

}

When using the render prop, the path parameters can be accessed through the props given to the render function:

<Route

exact

path='/u/:username/'

render={(props) =>

<ProfileComponent username={props.match.params.username}/>

}

/>

You generally use the render prop when you need some data from the component that contains your routes, since the component prop gives no real way of passing in additional props to the component.

answered Jul 7 '18 at 20:28

[](https://stackoverflow.com/users/3617886/tholle)

[Tholle](https://stackoverflow.com/users/3617886/tholle)

**66.1k**99 gold badges109109 silver badges110

例子3

render、children、component傻傻分不清楚(react-router v5.0)

<https://blog.csdn.net/weixin_44809405/article/details/91393342>

## （10）React Router 4 全新思維

① 一切路由皆組件

② 動態路由的離散式聲明方式，可以在各個組件中定義路由。

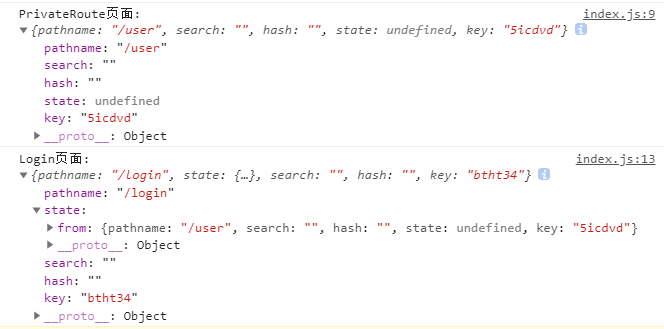
## （11）this.props.location

在頁面打印this.props.location，可以得到一個對象。

下面是大眾點評項目中，在PrivateRoute頁面和Login頁面打印this.props.location得到的對象。

state可以用來傳遞數據，這裡展示的是上一個頁面傳遞過來的數據。

Login頁面中，state的from屬性，是{from: props.location}，存的是跳轉到Login頁面之前的路由位置信息。



React 大眾點評項目

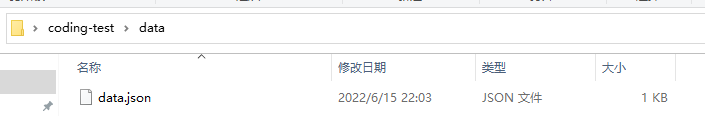
## Mock數據

### （1）代理到mock服務器

把mock的數據放到這台服務器，讓前端向這個代理服務器去發送請求。

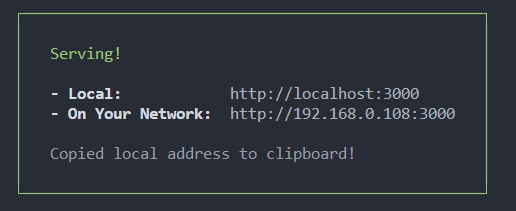
可以使用serve這個node.js開發的服務器作為mock服務器。

1. 安裝npm install -g serve
2. 進入需要託管數據的目錄

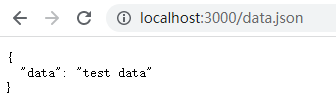
比如

1. 在這個目錄下的命令行運行serve

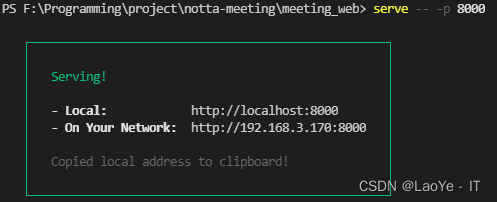
託管成功會出現這個



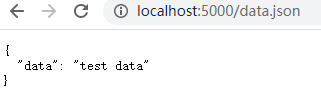
1. 在瀏覽器打開，可以看到被託管的數據



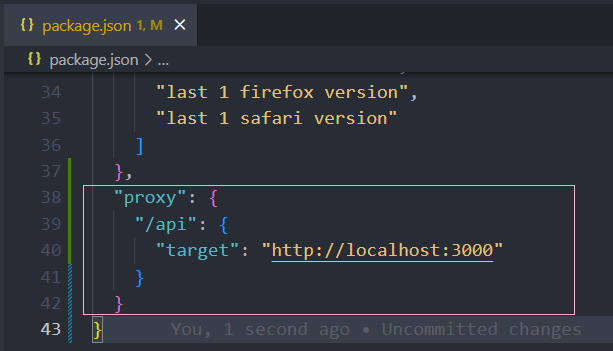
1. 如果要設置serve的端口



比如我輸入serve -- -p 5000可以看到



1. 在package.json中配置代理，把請求導向到代理服務器



注意：只要修改了package.json，就必須重啟應用。

上述package.json中的proxy代碼只在create-react-app的版本是1.x時管用

1. 2022年6月15日的版本可以這樣寫：

A 刪除上述package.json中的proxy代碼

B安裝 http-proxy-middleware，在項目根路徑下執行： npm install http-proxy-middleware

C 在src文件夾根目錄下，創建 setupProxy.js 文件，文件內容為：

const { createProxyMiddleware } = require('http-proxy-middleware');

module.exports = function (*app*) {

  app.use(createProxyMiddleware('/api', {

    target: 'http://localhost:5000/',

    changeOrigin: true,

    pathRewrite: {

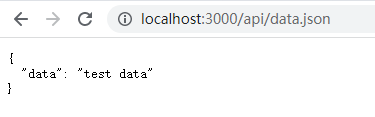
      "^/api": "/" // 把/api置空。http://localhost:3000/api/data.json （前端React所在端口） 相當於http://localhost:5000/data.json （服務器serve所在端口）

    }

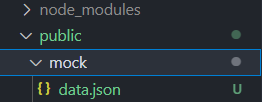
  }));

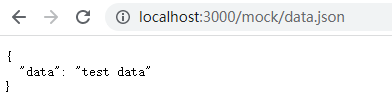
};

可以看到，在前端react頁面帶有/api/data.json時，成功獲取後端的數據



### （2）在public下放置mock數據





這樣也可以模擬前端通過http請求訪問後端數據

因為public下面的是靜態資源

React的Public文件夾下的文件不會參與編譯。要模擬被請求的服務器端的數據，可以放在這個文件夾下。

## 前端架構

（1）軟件架構過程：

抽象→解耦→組合

架構→框架設計模式→代碼

（2）前端架構的特殊性

前端不是一個獨立的子系統，又橫跨整個系統

前端架構實現2個目標：

* 前端具有分散性，所以得做到前端工程化。可控和高效相互交織。
  + 可控：腳手架，開發規範等。
  + 高效：框架、組件庫，Mock平台，構建部署工具等。
* 實現頁面的抽象、解耦、組合
  + 頁面UI抽象：組件
  + 通用邏輯抽象：領域實體、網絡請求、異常處理等

## React大眾點評項目功能路徑

（1）展示：首頁 → 詳情頁

（2）搜索：搜索頁 → 結果頁

（3）購買：登錄 → 下單 → 我的訂單 → 註銷

## 前端架構——工程化準備1

（1）前端架構——工程化準備1：技術選型和項目腳手架

（2）原則：

* 業務滿足程度
* 技術棧的成熟度（使用人數、周邊生態、倉庫維護等）
* 團隊對技術棧的熟悉度

（3）技術選型：

* UI層：React
* 路由：React Router
* 狀態管理：Redux

（4）腳手架：Create React App

npx create-react-app dianping-react

腳手架作用：

* 零配置創建React應用（不需要Webpack等配置）
* 打包構建JS、CSS、圖片
* 開發效率：自動刷新、代理轉發、單元測試等

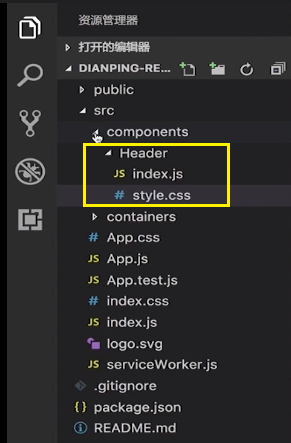
新建項目：npx create-react-app 項目名稱 （注意npm版本必須>= 5.2）

## 前端架構——工程化準備2

（1）目錄結構

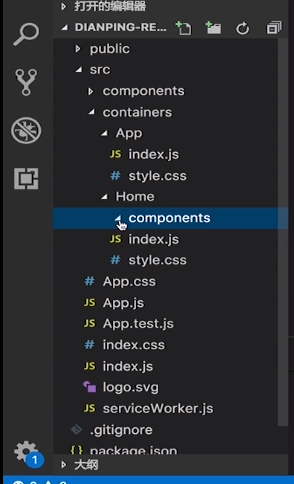
① 基於組件開發的目錄結構：展示型組件components，容器型組件containers

② 根目錄下的components放置全局型組件，比如loading組件，error提示組件，header，footer。若一個組件只在一個容器型組件中使用，或者只在某個頁面中使用，則放在對應頁面的容器型組件下。



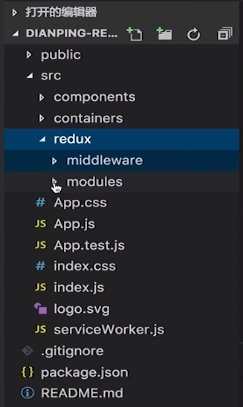
③containers下存放的容器型組件，可能對應著一個頁面，也可能對應著複雜頁面中的一部分模塊。

下圖例子中，containers下，有Home文件夾，Home/components放與home相關的展示型組件。



④ Redux數據管理模式：duck模式。把整個模塊涉及到的action，action type，reducer，selector都放在一個文件中。

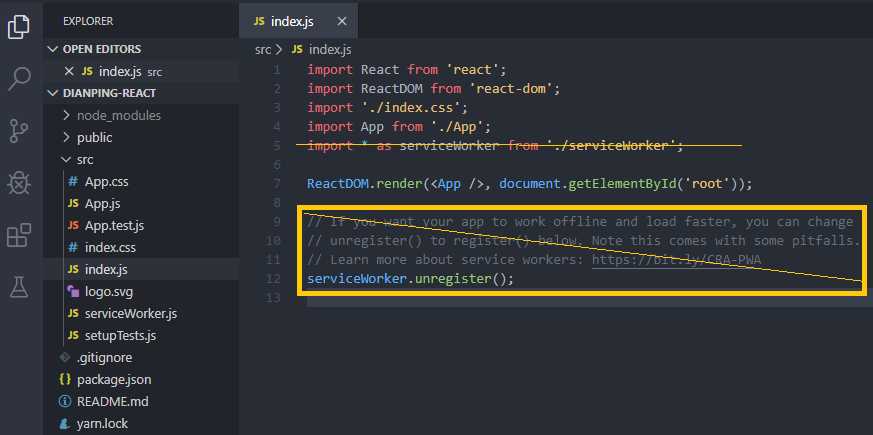
在modules文件夹下放置redux的模块·

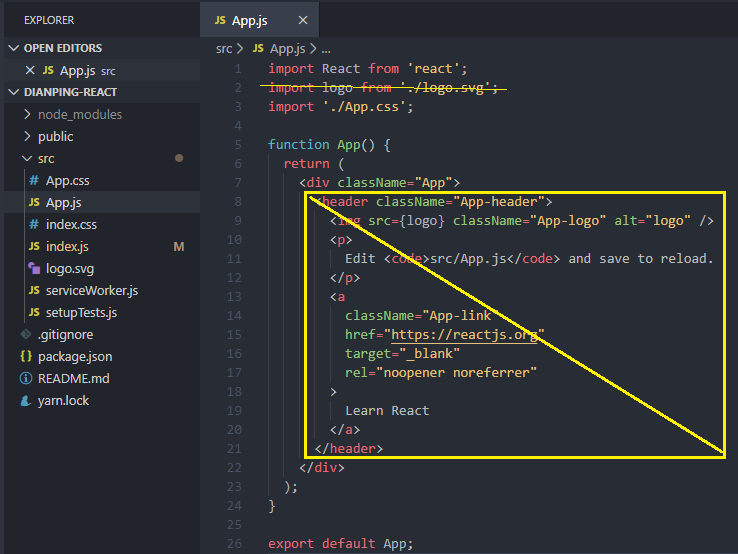
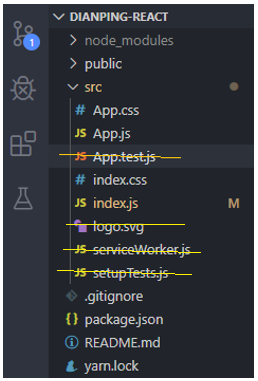


⑤src/utils文件夹放置常用的工具类

⑥images放置图片

⑦去掉不必要的代碼及文件





⑧把app.js和app.css移到src/containers/app文件夾中，然後把兩個文件改名為index.js和style.css。

把src/containers/app/index.js中的引入css文件改為import ‘./style.css’;

修改src/index.js的引入：把import App from './App';改為import App from './containers/App/index';

（2）構建體系

一般React開發的項目，用Webpack來構建。但是Create React App已經對WebPage做了封裝，所以直接用Create React App給的命令來啟動、封裝整個項目即可。

（3）Mock數據

建立public/mock文件夾。public文件夾下的資源是靜態資源，不會參與到webpage的打包構建。這裡的文件可以直接通過URL訪問到。

## 前端架構之抽象1：狀態模塊定義

（1）領域實體 src/redux/modules/entites

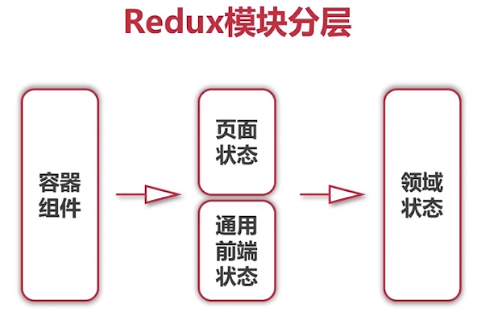
商品、店鋪、訂單、評論

（2）各頁面UI狀態

多選框是否處於選中狀態，輸入框的輸入信息什麼樣，loading彈出框是否處於彈出狀態

（3）前端基礎狀態

登錄狀態、全局異常信息。



src/redux/modules/index.js聚合所有的UI狀態和領域狀態，這是創建redux store時組合了所有reducer時的最終reducer

處於性能考慮，只在開發環境下集成調試工具

## 前端架構之抽象2：網絡請求層封裝（redux-thunk）

redux不支持異步action。如果想進行網絡請求，必然涉及到異步action，可藉助redux-thunk實現。

## 前端架構之抽象2：網絡請求層封裝（redux中間件）

## 前端架構之抽象3：通用錯誤處理

（1）準備：

錯誤信息組件 src/components/ErrorToast/index.js

錯誤狀態组件 src/redux/modules/app.js

（2）把两个组件连接起来：src/containers/App/index.js

## 首頁開發

### （1）category 分類菜單

① yarn add react-slick

github: <https://github.com/akiran/react-slick>

官網 <http://react-slick.neostack.com/> →

a. Get Started → Quick start，把2個css的link放在public/index.html中

b. Docs → Examples → AutoPlay → 建立

src/containers/Home/components/category/index.js

在展示型組件中，給css命名時，用BEM（block element）規則

### （2）headline 點評頭條

也用的react-slick

### （3）discount 超值特惠

建立src/containers/Home/components/Discount/index.js

去掉a標籤Hoover時的下劃線src/index.css

在src/index.js中引入 import "./index.css"

刪掉src/containers/App/style.css文件，

在src/containers/App/index.js中，去掉import './style.css';

### （4）LikeList 猜你喜歡

①

建立src/containers/Home/components/LikeList/index.js

LikeList裡的單個item抽出來，寫個獨立組件：

建立src/containers/Home/components/LikeItem/index.js

注意：產品標題/描述的css，寫上

overflow: hidden;

text\_overflow: ellipsis;

white-space; nowrap;

在標題/描述過長時，系統能自動省略

②加載更多的功能

建立Loading组件：src/components/Loading/index.js，在LikeList中引入Loading組件

### （5）其它組件開發

Header, Footer

橫幅Banner

活動Activity

### （6）連接Redux組件和首頁展示型組件

① 定義Selectors：src/redux/modules/home.js

② mapStateToProps，mapDispatchToProps

a. 在src/containers/Home/index.js引入

import { bindActionCreators } from “redux”

import { connect } from “react-redux”

import { actions as homeActions, getLikes, getDiscounts, getPageCountOfLikes } from “../../redux/modules/home”

b. 定義mapStateToProps，mapDispatchToProps

c. 組件使用

在class Home extends Component 中取出likes，discounts，pageCout，傳遞給相關組件。

### （7）Redux作為數據緩存層的作用

由詳情頁返回HomePage，一般的前端應用都會重新獲取數據。

而超值特惠和猜你喜歡這兩個組件的數據是不會變的，為了節約性能，可以：

①在src/redux/modules/home.js中的loadDiscounts函數中做個判斷。

const {ids} = getState().home.discounts

      if(ids.length > 0) {

        return null;

      }

②在src/containers/Home/components/LikeLiest/index.js中的componentDidMount()中做個判斷。

if(this.props.pageCount === 0) {

      this.props.fetchData();

    }

### （8）集成React Router

① Router基本結構定義

a. 在src/containers/App/index.js中

import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

b. 在App中的render中用<Router><Router/>包裹下層組件

c. Router組件下層使用Switch組件。這樣當router成功匹配到第一個組件後，就會停止匹配其它組件。

② 添加首頁到其他頁面的Link

a. src/containers/Home/components/HomeHeader/index.js

b. src/containers/Home/components/Discount/index.js

c. src/containers/Home/components/LikeItem/index.js

## 10.團購詳情頁開發

### （1）團購詳情頁ProductDetail

a. 創建src/containers/ProductDetail/index.js

b. 在src/containers/App/index.js中設置路徑

import ProductDetail from '../ProductDetail';

<Route path="/detail/:id" component={ProductDetail} />

### （2）團購基本信息ProductOverview

a. 創建src/containers/ProductDetail/components/ProductOverview/index.js

b. 創建css文件

注意：.productOverview\_\_content的寫法

-webkit-line-clamp 多行文字溢出

CSS代碼：

.box {  
    width: 100px;   
    display: -webkit-box;  
    -webkit-line-clamp: 2;  
    -webkit-box-orient: vertical;  
    overflow: hidden;  
}

HTML代碼：

<div class="box">

 多行文字溢出 多行文字溢出 多行文字溢出 多行文字溢出 多行文字溢出 多行文字溢出 多行文字溢出 多行文字溢出

</div>

效果：

 多行文字溢出 多行文字溢出 多行文字溢出 多行文字溢出

 多行文字溢出 多行文字溢出 多行文字溢出 多行文字溢出...

二、理解

注意：-webkit-line-clamp是webkit的私有屬性，是一個 不規範的屬性（unsupported WebKit property），它沒有出現在 CSS 規範草案中。

-webkit-line-clamp用來限制在一個塊元素顯示的文本的行數。 為了實現該效果，它需要組合其他的WebKit屬性。常見結合屬性：

display: -webkit-box; 必須結合的屬性 ，將物件作為彈性伸縮盒子模型顯示 。

-webkit-box-orient 必須結合的屬性 ，設置或檢索伸縮盒物件的子元素的排列方式 。

延伸：單行文本溢出顯示省略號...代碼

overflow:hidden;

text-overflow:ellipsis;

white-space:nowrap

c. 在src/container/ProductDetail/index.js中引入

d. 修改src/index.css 的body

### （3）商戶基本信息ShopInfo

a. 創建src/containers/ProductDetail/components/ShopInfo/index.js

b. 創建css文件

c. 在src/container/ProductDetail/index.js中引入

### （4）團購詳情Detail

a. 創建src/containers/ProductDetail/components/Detail/index.js

其中有一塊內容可以用table結構佈局。

b. 創建css文件

c. 在src/container/ProductDetail/index.js中引入

### （5）購買須知Remark

a. 創建src/containers/ProductDetail/components/Remark/index.js

用dl dt dd佈局

b. 創建css文件

c. 在src/container/ProductDetail/index.js中引入

### （6）其它組件 BuyButton和Header

① BuyButton

a. 創建src/containers/ProductDetail/components/BuyButton/index.js

b. 創建css文件

c. 在src/container/ProductDetail/index.js中引入

② Header

a. 創建src/components/Header/index.js

動態渲染Header組件

b. 創建css文件

c. 在src/container/ProductDetail/index.js中引入

### （7）redux狀態管理：設計詳情頁state

① 添加mock數據：product\_detail和shops

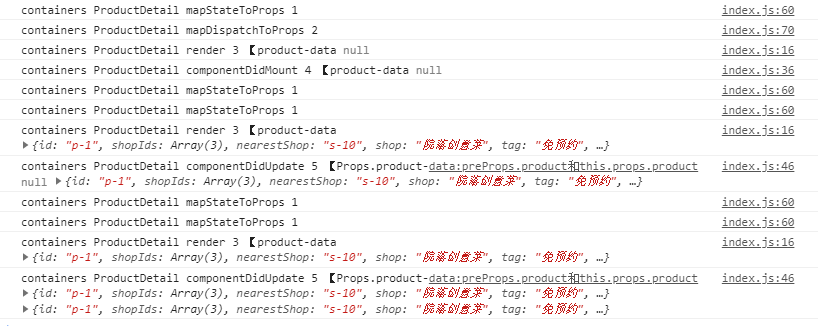
② 修改src/redux/modules/entities/shops.js

③ 創建src/utils/createReducer.js

### （8）redux狀態管理：設計詳情頁actions

### （9）詳情頁數據流

① product的數據流



② product和relatedShop的數據流





## 13.个人中心頁開發

### （1）map資料中，onClick指向的函數的寫法

① onClick={this.getFetchData.bind(this,item.id)}

② onClick={(event)=>this.getFetchData(event)}

③ onClick={this.handleSubmit}

以上方法中，①②適用於需要傳遞參數的情況，②③適用於無需傳遞參數的情況。

不能在onClick後的函數中直接寫括弧()，否則沒點擊就會直接調用函數

比如這個，就是錯誤的寫法：

<a onClick={this.handleClick(index)}>查看詳情</a>

## 15. 性能優化

### （1）集成Reselect

使用reselect库中提供的createSelector来创建具有缓存功能的selector函数。createSelector接收2个参数。第1个参数是数组，数组中的每一个元素都是一个selector函数。第2个参数是个函数，这个函数中接收到的参数是前面的selector函数返回的结果。

這是purchase.js中的例子。user.js中也有例子。

import { createSelector } from 'reselect';

export const getTotalPrice = createSelector(

    [getProduct, getQuantity],

    (*product*, *quantity*) => {

        if (!product) {

            return 0;

        }

        return (product.currentPrice \* quantity).toFixed(1);

    }

)

### （2）組件按需加載

需要動態加載一個模塊時，在import內寫入組件的路徑，這樣就可以對路徑對應的組件進行動態加載。

import(“../pathtocomponent”)

上面的import不是函數，而是運算符。這個運算符運算後的返回結果是個promise對象。當這個組件加載完成，promise的狀態會變成完成狀態。這時候可以正常使用這個組件的功能。

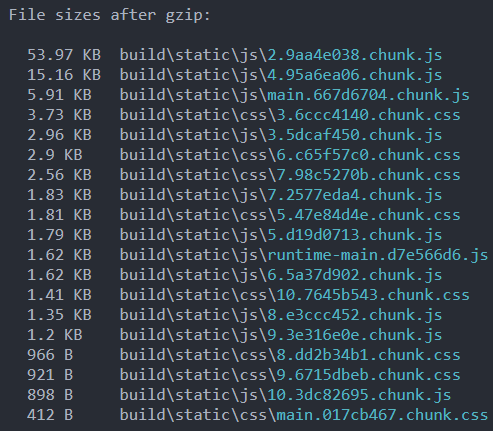
可以把上面的代碼放在一個函數中，這樣可以根據需要來通過調用函數實現動態加載。

在項目中，utils文件夾下，定義一個高階組件AsyncComponent.js來封裝上述邏輯。

有几个路由，就把js代码分割成几个编译的文件。这样每访问一个路由的URL，就只加载对应的js代码，不会加载其它页面的代码，提升速度，节约性能。

AsyncComponent这个高阶组件可以用在任意的地方（不一定按照路由来分组js文件代码），对组件进行异步加载。但一般习惯根据路由的URL做代码分组。

用yarn build打包后，可以发现，除了一个mai.js外，还有其它的一些代码chunk.js。这些chunk.js对应的就是通过AsyncComponent动态加载生成的代码。



如果react-scripts的版本在2以上，yarn build后，不仅根据路由单独打包出js文件，也会把这个页面的CSS文件单独打包出来。1以下的react-scripts版本，只分开打包js文件。

最好使用react-scripts 2以上的版本。如果使用react-scripts 1的版本，每个页面的css代码被打包在js文件中，页面加载的时候会后加载css代码（因为需要时间去临时创建css文件，导致js代码先加载css代码后加载），导致用户会先看到没有css效果的页面，影响用户体验。而react-scripts 2以上的版本单独打包css文件，且css文件先于js文件引入，从而避免上述问题。

css文件和js文件不一定一一对应，因为有的js代码是依赖包或者提取出来的各个页面共享的js代码。

### （3）項目構建和部署

①

Nginx是目前最流行的一種Web服務器，常用來託管靜態資源和做反向代理。

安裝好Nginx後，在nginx.conf中配置：

# 這是treasure-store網站

    server {

        listen       8066;

        server\_name  localhost;

        location / {

            root   D:/CSProgrammes/nginx-1.18.0/html/treasure-store;

            index  index.html;

            try\_files $uri /index.html;  #當發現在當前的路徑下找不到對應的文件時（比如在產品詳情頁面刷新，會出現404的情況），把路徑重新定向到默認的index.html文件下

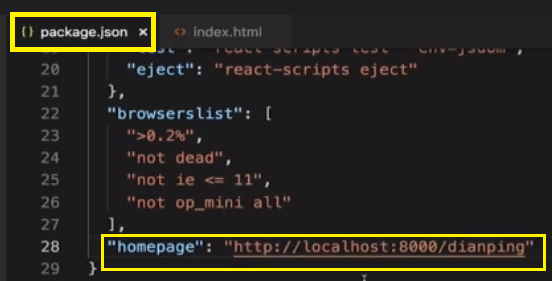
        }

    }

②把項目配置到某個子文件下（共a-e 5步）

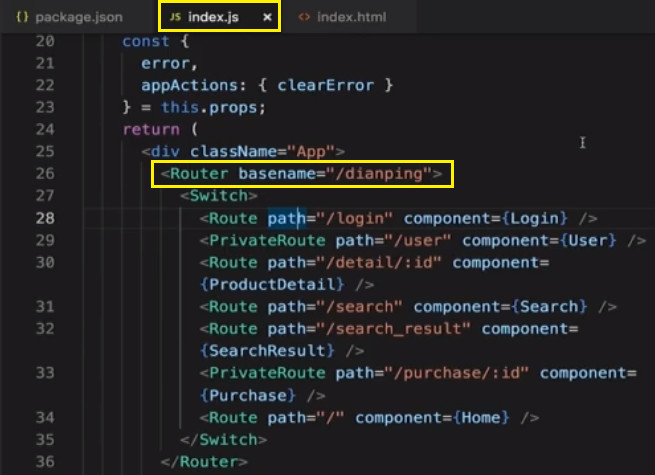
比如講師把項目放在dianping這個文件夾下，通過localhost:8000/dianping來訪問

a 在react項目的package.json中配置homepage



b 修改項目src/containers/App/index.js

給Router加個basename=”/dianping”這樣所有的Route路徑都會放在這個dianping下面



c 運行yarn build

d 把項目build中的所有文件放入Nginx項目文件夾中

e 修改nginx.conf



### （4）build後運行項目出錯：react failed to load source

解決方案：在package.json中添加"homepage": "./",

然後重新npm run build打包