【学习笔记 - React18】努力上岸的小码农

> 一边卷前端、一边卷上岸 —— 远离中年危机
> 好记性不如烂笔头 —— 张薄

创建项目

使用 `create-react-app` 脚手架工具创建一个 TS 版本的 React 项目：

复制代码

```
npx create-react-app 项目名 --template typescript
```

概念术语

钩子（hooks）

- 这是一类特殊的函数，为你的函数式组件注入特殊的功能，往往以 `use` 开头命名。
- 代替**无状态组件**和**高阶组件（HOC）** 这两个组件复用的方案。
  - 无状态组件：没有 `state` 的函数式组件，仅依赖于 `props` 注入，但问题是没有生命周期和副作用，没有办法进行访问数据和异步更新。
  - 高阶组件：直接在原先的组件外再套一层组件，但问题是加深了组件的嵌套性，容易产生类似回调地狱一样的 DOM 结构。

副作用

- 纯函数（pure function）：给一个函数同样的参数，它永远返回同样的值。例如在 React 组件输入相同的参数（props），渲染 UI 应该永远一样。
- 副作用（side effect）：一个函数处理了与返回值无关的事情。例如函数中修改了全局变量，函数中进行了 Ajax 调用，修改了 DOM 元素，甚至打印了 `console.log` 等等。

全局数据传递

原生 Context

父组件（ index.tsx ）中，三个步骤，见下面代码注释：

tsx 复制代码

```tsx
import React from 'react';
import ReactDom from 'react-dom';
import App from './App';

// Step1. 创建上下文对象时必须定义一个初始值
const defaultContextValue = {
  username: "测试用户"
}

// Step2. 创建上下文对象，并且这个上下文关系对象将会在 index.tsc 以外被使用，所以需要 export
export const appContext = React.createContext(defaultContextValue)

ReactDOM.render(
  <React.StrictMode>
    {/* Step3. 要传递数据给 App 组件及其子组件，就要用 Provider 包裹起来，并将要传递的数据注入 value */}
    <appContext.Provider value={defaultContextValue}>
      <App/>
    </appContext.Provider>
  </React.StrictMode>,
  document.getElementById("root")
)
```

子组件的子组件（ Robot.tsx ），两个步骤，见下面代码注释：

tsx 复制代码

```tsx
import React from 'react';
// Step1. 引入上下文关系对象
import { appContext } from '../index';

interface RobotProps {
  id: number;
  name: string;
  email: string;
}

const Robot: React.FC<RobotProps> = ({ id, name, email }) => {
  return (
    // Step2. 用 Consumer（消费者，对应生产者 Provider）来包裹所有的 jsx 代码
    <appContext.Consumer>
      {/* Consumer 组件内部用花括号+箭头函数来 return 原先的 jsx，参数 value 是上下文关系对象的取值 */}
      {(value) => {
        return (
```

```
          <div>
            <p>{name}</p>
            <p>{email}</p>
            {/* 使用 value. 来访问上下文关系对象中的值 */}
            <p>{value.username}</p>
          </div>
        );
      }}
    </appContext.Consumer>
  );
};

export defalut Robot;
```

useContext()

随着 hooks 的引入，使用 `useContext()` 可以简化上面的操作。

父组件不变，子组件的子组件（ `Robot.tsx` ）中不再需要 Consumer 组件来包裹，见下面代码注释（看上去大步骤多了一步，实际上使用的时候清爽很多了）：

tsx 复制代码

```tsx
// Step1. 引入 useContext 这个钩子函数
import React, {useContext} from 'react';
// Step2. 引入上下文关系对象
import { appContext } from '../index';

interface RobotProps {
  id: number;
  name: string;
  email: string;
}

const Robot: React.FC<RobotProps> = ({ id, name, email }) => {
  // Step3. 直接在函数式组件内部使用 useContext 访问上下文关系对象
  // 传入的参数指定了使用哪个上下文关系对象
  const value = useContext(appContext)
  return (
    {/* return 中可以直接使用 value 了，不再需要 Consumer 来包裹 */}
    <div>
      <p>{name}</p>
      <p>{email}</p>
      {/* 直接使用 value. 来访问上下文关系对象中的值 */}
      <p>{value.username}</p>
    </div>
  );
};
```

```tsx
export defalut Robot;
```

封装全局状态管理

为了让项目结构更清晰，一般会将上下文关系对象拎出来做成一个单独的组件，而不是混写到其它组件中，例如 `AppState.tsx`：

tsx 复制代码

```tsx
import React, {useState, PropsWithChildren} from 'react';

// 定义 ContextValue 的类型
interface AppStateValue {
  username: string;
  shoppingCart: { items: {id: number, name: string}[] }
}

// 定义 ContextValue 的默认值
const defaultContextValue: AppStateValue = {
  username: "测试用户",
  shoppingCart: { items: [] }
};

// 创建上下文对象并导出给别的组件用
export const appContext = React.createContext(defaultContextValue)

// 创建 Context 组件，它就是用来包裹其它组件的 Provider
// children 表示所有子组件都被包裹，且为它们提供全局数据支持
// React.FC 的泛型是固定写法，react18 + ts 时，使用 children 就要这么写
// 继续添加的泛型 <{}> 是当前组件自定义的类型属性，当前组件没有定义就用空花括号
export const AppStateProvider: React.FC<PropsWithChildren<{}>> = (props) => {
  const [state, setState] = useState(defaultContextValue);

  return (
    <appContext.Provider value={state}>
      {props.children}
    </appContext.Provider>
  );
};
```

使用时，在父组件中（`index.tsx`）使用上面定义的 Provider 组件：

tsx 复制代码

```tsx
import React from 'react';
import ReactDom from 'react-dom';
import App from './App';
```

```tsx
// 引入自定义的 Provider 组件
import { AppStateProvider } from './AppState';

ReactDOM.render(
  <React.StrictMode>
    {/* 用 Provider 包裹起来 */}
    <AppStateProvider>
      <App/>
    </AppStateProvider>
  </React.StrictMode>,
  document.getElementById("root")
);
```

使用时，在子组件中（ Robot.tsx ）引入自定义的 appContext ：

tsx 复制代码

```tsx
import React, {useContext} from 'react';
// 引入上下文关系对象
import { appContext } from '../AppState';

interface RobotProps {
  id: number;
  name: string;
  email: string;
}

const Robot: React.FC<RobotProps> = ({ id, name, email }) => {
  const value = useContext(appContext)
  return (
    <div>
      <p>{name}</p>
      <p>{email}</p>
      <p>{value.username}</p>
    </div>
  );
};

export defalut Robot;
```

全局状态更新

全局状态（state）的更新需要使用到 setState 这个钩子函数，为了能够共享这个钩子，就需要创建一个新的 context 来连接这个 setState 函数。

修改 AppState.tsx 代码，有两个步骤：

tsx 复制代码

```tsx
import React, {useState, PropsWithChildren} from 'react';

interface AppStateValue {
  username: string;
  shoppingCart: { items: {id: number, name: string}[] }
}

const defaultContextValue: AppStateValue = {
  username: "测试用户",
  shoppingCart: { items: [] }
};

export const appContext = React.createContext(defaultContextValue)

// Step1. 创建一个新的 context，因为初始化的是函数，所以可以传入 undefined 作为初始化值
// 因为是 ts，需要传入一个联合类型（setState 函数的类型，鼠标悬浮获取后复制过来，并加上 undefined 类型）
export const appSetStateContext = React.createContext<
  React.Dispatch<React.SetStateAction<AppStateValue>> | undefined>(undefined);

export const AppStateProvider: React.FC<PropsWithChildren<{}>> = (props) => {
  const [state, setState] = useState(defaultContextValue);

  return (
    <appContext.Provider value={state}>
      {/* Step2. 继续添加 Provider，传入 setState 函数 */}
      <appSetStateContext.Provider value={setState}>
        {props.children}
      </appSetStateContext.Provider>
    </appContext.Provider>
  );
};
```

在子组件（ Robot.tsx ）中修改 context 中的值，有三个步骤：

tsx 复制代码

```tsx
import React, {useContext} from 'react';
// Step1. 引入 appSetStateContext
import { appContext, appSetStateContext } from '../AppState';

interface RobotProps {
  id: number;
  name: string;
  email: string;
}

const Robot: React.FC<RobotProps> = ({ id, name, email }) => {
  const value = useContext(appContext)
  // Step2. 添加对 appSetStateContext 访问
```

```
  const setState = useContext(appSetStateContext)
  const addToCart = () => {
    // 初始化的时候，setState 函数使用的是 undefined，需要判断一下
    if (setState) {
      setState(state => {
        return {
          ...state,
          shoppingCart: {
            items: [...state.shoppingCart.items, {id, name}]
          }
        }
      })
    }
  }
  return (
    <div>
      <p>{name}</p>
      <p>{email}</p>
      <p>{value.username}</p>
      {/* Step3. 使用 appSetStateContext */}
      <button onClick={addToCart}>加入购物车</button>
    </div>
  );
};

export defalut Robot;
```