React Hooks – useReducer & useContext

1. useReducer

当状态更新逻辑较复杂时可以考虑使用 useReducer。useReducer 可以同时更新多个状态，而且能把对状态的修改**从组件中独立出来。**

相比于 useState，useReducer 可以更好的描述“如何更新状态”。例如：组件负责发出行为，useReducer 负责更新状态。

好处是：让代码逻辑更清晰，代码行为更易预测。

1.1 useReducer 的语法格式

useReducer 的基础语法如下：

```
const [state, dispatch] = useReducer(reducer, initState, initAction?)
```

其中：

1. **reducer** 是一个函数，类似于 `(prevState, action) => newState`。形参 `prevState` 表示旧状态，形参 `action` 表示本次的行为，返回值 `newState` 表示处理完毕后的新状态。
2. **initState** 表示初始状态，也就是默认值。
3. **initAction** 是进行状态初始化时候的处理函数，它是可选的，如果提供了 initAction 函数，则会把 initState 传递给 initAction 函数进行处理，initAction 的返回值会被当做初始状态。
4. 返回值 state 是状态值。dispatch 是更新 state 的方法，让他接收 action 作为参数，useReducer 只需要调用 `dispatch(action)` 方法传入的 action 即可更新 state。

1.2 useReducer 的基础用法

1. 定义组件的基础结构

1. 定义名为 `Father` 的父组件如下：

```
import React from 'react'

// 父组件
export const Father: React.FC = () => {
 return (
   <div>
     <button>修改 name 的值</button>
     <div className="father">
       <Son1 />
       <Son2 />
     </div>
   </div>
 )
}
```

2. 定义名为 `Son1` 和 `Son2` 的两个子组件如下：

```
// 子组件1
const Son1: React.FC = () => {
 return
}

// 子组件2
const Son2: React.FC = () => {
 return
}
```

3. 在 `index.css` 中添加对应的样式：

```
.father {
 display: flex;
 justify-content: space-between;
 width: 100vw;
}

.son1 {
 background-color: orange;
 min-height: 300px;
 flex: 1;
 padding: 10px;
}

.son2 {
 background-color: lightblue;
 min-height: 300px;
 flex: 1;
 padding: 10px;
}
```

2. 定义 useReducer 的基础结构

1. 按需导入 `useReducer` 函数：

```
import React, { useReducer } from 'react'
```

2. 定义**初始数据**：

```
const defaultState = { name: 'liulongbin', age: 16 }
```

3. 定义 `reducer` 函数，它的作用是：**根据旧状态，进行一系列处理，最终返回新状态**：

```
const reducer = (prevState) => {
 console.log('触发了 reducer 函数')
 return prevState
}
```

4. 在 `Father` 组件中，调用 `useReducer(reducerFn, 初始状态)` 函数，并得到 reducer 返回的状态：

```
// 父组件
export const Father: React.FC = () => {
 // useReducer(fn, 初始数据, 对初始数据进行处理的fn)
 const [state] = useReducer(reducer, defaultState)
 console.log(state)

 return (
   <div>
     <button>修改 name 的值</button>
     <div className="father">
       <Son1 />
       <Son2 />
     </div>
   </div>
 )
}
```

5. 为 reducer 中的 initState 指定数据类型：

```
// 定义状态的数据类型
type UserType = typeof defaultState

const defaultState = { name: 'liulongbin', age: 16 }

// 给 initState 指定类型为 UserType
const reducer = (prevState: UserType) => {
 console.log('触发了 reducer 函数')
 return prevState
}
```

接下来，在 `Father` 组件中使用 state 时，就可以出现类型的智能提示啦：

```
// 父组件
export const Father: React.FC = () => {
 const [state] = useReducer(reducer, defaultState)
 console.log(state.name, state.age)

 return (
   <div>
     <button>修改 name 的值</button>
     <div className="father">
       <Son1 />
       <Son2 />
     </div>
```

```
    </div>
  )
}
```

3. 使用 initAction 处理初始数据

1. 定义名为 `initAction` 的处理函数，如果初始数据中的 age 为小数、负数、或 0 时，对 age 进行非法值的处理：

```
const initAction = (initState: UserType) => {
 // 把 return 的对象，作为 useReducer 的初始值
 return { ...initState, age: Math.round(Math.abs(initState.age)) || 18 }
}
```

2. 在 `Father` 组件中，使用步骤1声明的 `initAction` 函数如下：

```
// 父组件
export const Father: React.FC = () => {
 // useReducer(fn, 初始数据，对初始数据进行处理的fn)
 const [state] = useReducer(reducer, defaultState, initAction)

 // 省略其它代码...
}
```

> 可以在定义 defaultState 时，为 age 提供非法值，可以看到非法值在 initAction 中被处理掉了。

4. 在 Father 组件中点击按钮修改 name 的值

4.1 错误示范

```
// 父组件
export const Father: React.FC = () => {
  // useReducer(fn, 初始数据，对初始数据进行处理的fn)
  const [state] = useReducer(reducer, defaultState, initAction)
  console.log(state)

  const onChangeName = () => {
    // 注意：这种用法是错误的，因为不能直接修改 state 的值
    // 因为存储在 useReducer 中的数据都是“不可变”的!
    // 要想修改 useReducer 中的数据，必须触发 reducer 函数的重新计算，
    // 根据 reducer 形参中的旧状态对象（initState），经过一系列处理，返回一个“全新的”状态对象
    state.name = 'escook'
  }

  return (
    <div>
      <button onClick={onChangeName}>修改 name 的值</button>
      <div className="father">
        <Son1 />
        <Son2 />
      </div>
    </div>
```

```
  )
}
```

4.2 正确的操作

1. 为了能够触发 reducer 函数的重新执行，我们需要在调用 `useReducer()` 后接收返回的 `dispatch` 函数。示例代码如下：

```
// Father 父组件
const [state, dispatch] = useReducer(reducer, defaultState, initAction)
```

2. 在 button 按钮的点击事件处理函数中，调用 `dispatch()` 函数，从而触发 reducer 函数的重新计算：

```
// Father 父组件
const onChangeName = () => {
 dispatch()
}
```

3. 点击 Father 组件中如下的 button 按钮：

```
修改 name 的值
```

会触发 reducer 函数的重新执行，并打印 reducer 中的 `console.log()`，代码如下：

```
const reducer = (prevState: UserType) => {
 console.log('触发了 reducer 函数')
 return prevState
}
```

4.4 调用 dispatch 传递参数给 reducer

1. 在 Father 父组件按钮的点击事件处理函数 `onChangeName` 中，调用 **dispatch()** 函数并把参数传递给 **reducer** 的第2个形参，代码如下：

```
const onChangeName = () => {
 // 注意：参数的格式为 { type, payload? }
 // 其中：
 // type 的值是一个唯一的标识符，用来指定本次操作的类型，一般为大写的字符串
 // payload 是本次操作需要用到的数据，为可选参数。在这里，payload 指的是把用户名改为字符串 '刘龙彬'
 dispatch({type: 'UPDATE_NAME', payload: '刘龙彬'})
}
```

2. 修改 reducer 函数的形参，添加名为 `action` 的第2个形参，用来接收 `dispatch` 传递过来的数据：

```
const reducer = (prevState: UserType, action) => {
 // 打印 action 的值，终端显示的值为：
 // {type: 'UPDATE_NAME', payload: '刘龙彬'}
 console.log('触发了 reducer 函数', action)
 return prevState
}
```

3. 在 reducer 中，根据接收到的 `action.type` 标识符，**决定进行怎样的更新操作**，最终 return 一个计算好的新状态。示例代码如下：

```
const reducer = (prevState: UserType, action) => {
 console.log('触发了 reducer 函数', action)
 // return prevState

 switch (action.type) {
   // 如果标识符是字符串 'UPDATE_NAME'，则把用户名更新成 action.payload 的值
   // 最后，一定要返回一个新状态，因为 useReducer 中每一次的状态都是“不可变的”
   case 'UPDATE_NAME':
     return { ...prevState, name: action.payload }
   // 兜底操作：
   // 如果没有匹配到任何操作，则默认返回上一次的旧状态
   default:
     return prevState
 }
}
```

4. 在上述的 `switch...case...` 代码期间，没有任何 TS 的类型提示，这在大型项目中是致命的。因此，我们需要为 reducer 函数的第2个形参 **action** 指定操作的类型：

```
// 1. 定义 action 的类型
type ActionType = { type: 'UPDATE_NAME'; payload: string }

// 2. 为 action 指定类型为 ActionType
const reducer = (prevState: UserType, action: ActionType) => {
 console.log('触发了 reducer 函数', action)

 // 3. 删掉之前的代码，再重复编写这段逻辑的时候，会出现 TS 的类型提示，非常 Nice
 switch (action.type) {
   case 'UPDATE_NAME':
     return { ...prevState, name: action.payload }
   default:
     return prevState
 }
}
```

同时，在 Father 组件的 `onChangeName` 处理函数内，调用 `dispatch()` 时也有了类型提示：

```
const onChangeName = () => {
 dispatch({ type: 'UPDATE_NAME', payload: '刘龙彬' })
}
```

> 注意：在今后的开发中，正确的顺序是先定义 ActionType 的类型，再修改 reducer 中的 switch... case... 逻辑，最后在组件中调用 dispatch() 函数哦！这样能够充分利用 TS 的类型提示。

5. 把用户信息渲染到子组件中

1. 在 Father 父组件中，通过展开运算符把 state 数据对象绑定为 `Son1` 和 `Son2` 的 props 属性：

```
// 父组件
export const Father: React.FC = () => {
 const [state, dispatch] = useReducer(reducer, defaultState, initAction)

 const onChangeName = () => {
   dispatch({ type: 'UPDATE_NAME', payload: '刘龙彬' })
 }

 return (
   <div>
     <button onClick={onChangeName}>修改 name 的值</button>
     <div className="father">
       <!-- 通过 props 的数据绑定，把数据传递给子组件 -->
       <Son1 {...state} />
       <Son2 {...state} />
     </div>
   </div>
 )
}
```

2. 在子组件中，指定 props 的类型为 `React.FC<UserType>`，并使用 props 接收和渲染数据：

```
// 子组件1
const Son1: React.FC<UserType> = (props) => {
 return (
   <div className="son1">
     <p>用户信息：</p>
     <p>{JSON.stringify(props)}</p>
   </div>
 )
}

// 子组件2
const Son2: React.FC<UserType> = (props) => {
 return (
   <div className="son2">
     <p>用户信息：</p>
     <p>{JSON.stringify(props)}</p>
   </div>
 )
}
```

> 修改完成后，点击父组件中的 button 按钮修改用户名，我们发现两个子组件中的数据同步发生了变化。

6. 在子组件中实现点击按钮 age 自增操作

1. 扩充 `ActionType` 的类型如下：

```
// 定义 action 的类型
type ActionType = { type: 'UPDATE_NAME'; payload: string } | { type: 'INCREMENT';
payload: number }
```

2. 在 `reducer` 中添加 `INCREMENT` 的 `case` 匹配：

```
const reducer = (prevState: UserType, action: ActionType) => {
 console.log('触发了 reducer 函数', action)

 switch (action.type) {
   case 'UPDATE_NAME':
     return { ...prevState, name: action.payload }
   // 添加 INCREMENT 的 case 匹配
   case 'INCREMENT':
     return { ...prevState, age: prevState.age + action.payload }
   default:
     return prevState
 }
}
```

3. 在子组件 `Son1` 中添加 `+1` 的 button 按钮，并绑定点击事件处理函数：

```
// 子组件1
const Son1: React.FC<UserType> = (props) => {
 const add = () => {}

 return (
   <div className="son1">
     <p>用户信息：</p>
     <p>{JSON.stringify(props)}</p>
     <button onClick={add}>+1</button>
   </div>
 )
}
```

4. 现在的问题是：子组件 Son1 中无法调用到父组件的 `dispatch` 函数。为了解决这个问题，我们需要在 Father 父组件中，通过 props 把父组件中的 `dispatch` 传递给子组件：

```
// 父组件
export const Father: React.FC = () => {
 // useReducer(fn, 初始数据, 对初始数据进行处理的fn)
 const [state, dispatch] = useReducer(reducer, defaultState, initAction)

 const onChangeName = () => {
   dispatch({ type: 'UPDATE_NAME', payload: '刘龙彬' })
 }
```

```
  return (
    <div>
      <button onClick={onChangeName}>修改 name 的值</button>
      <div className="father">
        <Son1 {...state} dispatch={dispatch} />
        <Son2 {...state} />
      </div>
    </div>
  )
}
```

5. 在 `Son1` 子组件中，扩充 `React.FC<UserType>` 的类型，并从 `props` 中把 **dispatch** 和**用户信息对象**分离出来：

```
// 子组件1
const Son1: React.FC<UserType & { dispatch: React.Dispatch<ActionType> }> = (props)
=> {
  const { dispatch, ...user } = props

  const add = () => dispatch({ type: 'INCREMENT', payload: 1 })

  return (
    <div className="son1">
      <p>用户信息：</p>
      <p>{JSON.stringify(user)}</p>
      <button onClick={add}>+1</button>
    </div>
  )
}
```

7. 在子组件中实现点击按钮 age 自减操作

1. 扩充 `ActionType` 的类型如下：

```
// 定义 action 的类型
type ActionType = { type: 'UPDATE_NAME'; payload: string } | { type: 'INCREMENT';
payload: number } | { type: 'DECREMENT'; payload: number }
```

2. 在 `reducer` 中添加 `DECREMENT` 的 `case` 匹配：

```
const reducer = (prevState: UserType, action: ActionType) => {
  console.log('触发了 reducer 函数', action)

  switch (action.type) {
    case 'UPDATE_NAME':
      return { ...prevState, name: action.payload }
    case 'INCREMENT':
      return { ...prevState, age: prevState.age + action.payload }
    // 添加 DECREMENT 的 case 匹配
    case 'DECREMENT':
      return { ...prevState, age: prevState.age - action.payload }
```

```
    default:
      return prevState
  }
}
```

3. 在子组件 `Son2` 中添加 `-5` 的 button 按钮，并绑定点击事件处理函数：

```
// 子组件2
const Son2: React.FC<UserType> = (props) => {
 const sub = () => { }

 return (
   <div className="son2">
     <p>用户信息：</p>
     <p>{JSON.stringify(props)}</p>
     <button onClick={sub}>-5</button>
   </div>
 )
}
```

4. 现在的问题是：子组件 Son2 中无法调用到父组件的 `dispatch` 函数。为了解决这个问题，我们需要在 Father 父组件中，通过 props 把父组件中的 `dispatch` 传递给子组件：

```
// 父组件
export const Father: React.FC = () => {
 // useReducer(fn, 初始数据, 对初始数据进行处理的fn)
 const [state, dispatch] = useReducer(reducer, defaultState, initAction)

 const onChangeName = () => {
   dispatch({ type: 'UPDATE_NAME', payload: '刘龙彬' })
 }

 return (
   <div>
     <button onClick={onChangeName}>修改 name 的值</button>
     <div className="father">
       <Son1 {...state} dispatch={dispatch} />
       <Son2 {...state} dispatch={dispatch} />
     </div>
   </div>
 )
}
```

5. 在 `Son2` 子组件中，扩充 `React.FC<UserType>` 的类型，并从 `props` 中把 **dispatch** 和**用户信息对象**分离出来：

```
// 子组件2
const Son2: React.FC<UserType & { dispatch: React.Dispatch<ActionType> }> = (props)
=> {
 const { dispatch, ...user } = props
 const sub = () => dispatch({ type: 'DECREMENT', payload: 5 })

 return (
   <div className="son2">
     <p>用户信息：</p>
     <p>{JSON.stringify(user)}</p>
     <button onClick={sub}>-5</button>
   </div>
 )
}
```

8. 在 GrandSon 组件中实现重置按钮

1. 扩充 `ActionType` 的类型如下：

```
// 定义 action 的类型
type ActionType = { type: 'UPDATE_NAME'; payload: string } | { type: 'INCREMENT';
payload: number } | { type: 'DECREMENT'; payload: number } | { type: 'RESET' }
```

2. 在 `reducer` 中添加 `RESET` 的 `case` 匹配：

```
const reducer = (prevState: UserType, action: ActionType) => {
 console.log('触发了 reducer 函数', action)

 switch (action.type) {
   case 'UPDATE_NAME':
     return { ...prevState, name: action.payload }
   case 'INCREMENT':
     return { ...prevState, age: prevState.age + action.payload }
   case 'DECREMENT':
     return { ...prevState, age: prevState.age - action.payload }
   // 添加 RESET 的 case 匹配
   case 'RESET':
     return defaultState
   default:
     return prevState
 }
}
```

3. 在 `GrandSon` 组件中，添加重置按钮，并绑定点击事件处理函数：

```
const GrandSon: React.FC<{ dispatch: React.Dispatch<ActionType> }> = (props) => {
 const reset = () => props.dispatch({ type: 'RESET' })

 return (
   <>
     <h3>这是 GrandSon 组件</h3>
     <button onClick={reset}>重置</button>
   </>
 )
}
```

9. 使用 Immer 编写更简洁的 reducer 更新逻辑

1. 安装 immer 相关的依赖包：

```
npm install immer use-immer -S
```

2. 从 `use-immer` 中导入 `useImmerReducer` 函数，并替换掉 React 官方的 `useReducer` 函数的调用：

```
// 1. 导入 useImmerReducer
import { useImmerReducer } from 'use-immer'

// 父组件
export const Father: React.FC = () => {
 // 2. 把 useReducer() 的调用替换成 useImmerReducer()
 const [state, dispatch] = useImmerReducer(reducer, defaultState, initAction)
}
```

3. 修改 reducer 函数中的业务逻辑，`case` 代码块中不再需要 return 不可变的新对象了，只需要在 prevState 上进行修改即可。**Immer 内部会复制并返回新对象**，因此降低了用户的心智负担。改造后的 reducer 代码如下：

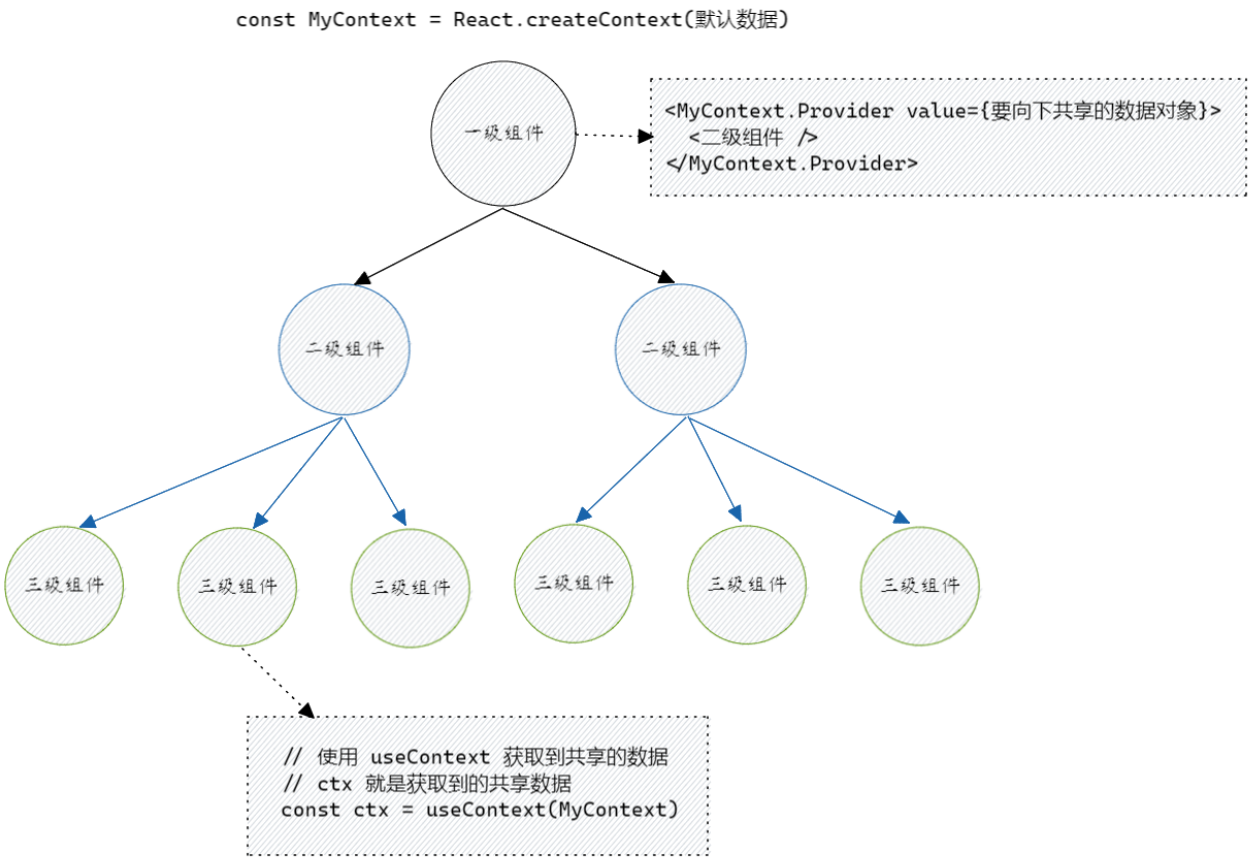
```
const reducer = (prevState: UserType, action: ActionType) => {
 console.log('触发了 reducer 函数', action)

 switch (action.type) {
   case 'UPDATE_NAME':
     // return { ...prevState, name: action.payload }
     prevState.name = action.payload
     break
   case 'INCREMENT':
     // return { ...prevState, age: prevState.age + action.payload }
     prevState.age += action.payload
     break
   case 'DECREMENT':
     // return { ...prevState, age: prevState.age - action.payload }
     prevState.age -= action.payload
     break
   case 'RESET':
     return defaultState
   default:
```

```
      return prevState
  }
}
```

2. useContext

在 react 函数式组件中，如果组件的嵌套层级很深，当父组件想把数据共享给最深层的子组件时，传统的办法是**使用 props，一层一层把数据向下传递。**

使用 props 层层传递数据的维护性太差了，我们可以使用 `React.createContext()` + `useContext()` 轻松实现多层组件的数据传递。



2.1 useContext 的语法格式

主要的使用步骤如下：

1. 在**全局**创建 Context 对象
2. 在**父组件**中使用 Context.Provider 提供数据
3. 在**子组件**中使用 useContext 使用数据

```
import React, { useContext } from 'react'

// 全局
const MyContext = React.createContext(初始数据)

// 父组件
const Father = () => {
```

```
  return <MyContext.Provider value={{name: 'escook', age: 22}}>
    <!-- 省略其它代码 -->
  </MyContext.Provider>
}

// 子组件
const Son = () => {
  const myCtx = useContext(MyContext)
  return <div>
    <p>姓名：{myCtx.name}</p>
    <p>年龄：{MyCtx.age}</p>
  </div>
}
```

2.2 useContext 的基础用法

1. 定义组件结构

定义 `LevelA`，`LevelB`，`LevelC` 的组件结构如下：

```
import React, { useState } from 'react'

export const LevelA: React.FC = () => {
  // 定义状态
  const [count, setCount] = useState(0)

  return (
    <div style={{ padding: 30, backgroundColor: 'lightblue', width: '50vw' }}>
      <p>count值是：{count}</p>
      <button onClick={() => setCount((prev) => prev + 1)}>+1</button>
      {/* 使用子组件 */}
      <LevelB />
    </div>
  )
}

export const LevelB: React.FC = () => {
  return (
    <div style={{ padding: 30, backgroundColor: 'lightgreen' }}>
      {/* 使用子组件 */}
      <LevelC />
    </div>
  )
}

export const LevelC: React.FC = () => {
  return (
    <div style={{ padding: 30, backgroundColor: 'lightsalmon' }}>
      <button>+1</button>
      <button>重置</button>
    </div>
  )
}
```

2. createContext 配合 useContext 使用

在父组件中，调用 `React.createContext` 向下共享数据；在子组件中调用 `useContext()` 获取数据。示例代码如下：

```
import React, { useState, useContext } from 'react'

// 声明 TS 类型
type ContextType = { count: number; setCount:
React.Dispatch<React.SetStateAction<number>> }

// 1. 创建 Context 对象
const AppContext = React.createContext<ContextType>({} as ContextType)

export const LevelA: React.FC = () => {
  const [count, setCount] = useState(0)

  return (
    <div style={{ padding: 30, backgroundColor: 'lightblue', width: '50vw' }}>
      <p>count值是: {count}</p>
      <button onClick={() => setCount((prev) => prev + 1)}>+1</button>
      {/* 2. 使用 Context.Provider 向下传递数据 */}
      <AppContext.Provider value={{ count, setCount }}>
        <LevelB />
      </AppContext.Provider>
    </div>
  )
}

export const LevelB: React.FC = () => {
  return (
    <div style={{ padding: 30, backgroundColor: 'lightgreen' }}>
      <LevelC />
    </div>
  )
}

export const LevelC: React.FC = () => {
  // 3. 使用 useContext 接收数据
  const ctx = useContext(AppContext)

  return (
    <div style={{ padding: 30, backgroundColor: 'lightsalmon' }}>
      {/* 4. 使用 ctx 中的数据和方法 */}
      <p>count值是: {ctx.count}</p>
      <button onClick={() => ctx.setCount((prev) => prev + 1)}>+1</button>
      <button onClick={() => ctx.setCount(0)}>重置</button>
    </div>
  )
}
```

3. ☆☆☆以非侵入的方式使用 Context

在刚才的案例中，我们发现父组件 `LevelA` 为了向下传递共享的数据，在代码中侵入了 `<AppContext.Provider>` 这样的代码结构。

为了保证父组件中代码的单一性，也为了提高 Provider 的通用性，我们可以考虑把 `Context.Provider` 封装到独立的 `Wrapper` 函数式组件中，例如：

```
// 声明 TS 类型
type ContextType = { count: number; setCount:
React.Dispatch<React.SetStateAction<number>> }
// 创建 Context 对象
const AppContext = React.createContext<ContextType>({} as ContextType)

// 定义独立的 Wrapper 组件，被 Wrapper 嵌套的子组件会被 Provider 注入数据
export const AppContextWrapper: React.FC<React.PropsWithChildren> = (props) => {
  // 1. 定义要共享的数据
  const [count, setCount] = useState(0)
  // 2. 使用 AppContext.Provider 向下共享数据
  return <AppContext.Provider value={{ count, setCount }}>{props.children}
</AppContext.Provider>
}
```

定义好 `Wrapper` 组件后，我们可以在 `App.tsx` 中导入并使用 `Wrapper` 和 `LevelA` 组件，代码如下：

```
import React from 'react'
import { AppContextWrapper, LevelA } from '@/components/use_context/01.base.tsx'

const App: React.FC = () => {
  return (
    <AppContextWrapper>
      <!-- AppContextWrapper 中嵌套使用了 LevelA 组件，形成了父子关系 -->
      <!-- LevelA 组件会被当做 children 渲染到 Wrapper 预留的插槽中 -->
      <LevelA />
    </AppContextWrapper>
  )
}

export default App
```

这样，组件树的嵌套关系为：`App` => `Wrapper` => `LevelA` => `LevelB` => `LevelC`。因此在 `LevelA`、`LevelB` 和 `LevelC` 组件中，都可以使用 context 中的数据。例如，LevelA 组件中的代码如下：

```
export const LevelA: React.FC = () => {
  // 使用 useContext 接收数据
  const ctx = useContext(AppContext)

  return (
    <div style={{ padding: 30, backgroundColor: 'lightblue', width: '50vw' }}>
      {/* 使用 ctx 中的数据和方法 */}
      <p>count值是: {ctx.count}</p>
      <button onClick={() => ctx.setCount((prev) => prev + 1)}>+1</button>
      <LevelB />
    </div>
  )
}
```

LevelC 组件中的代码如下：

```
export const LevelC: React.FC = () => {
  // 使用 useContext 接收数据
  const ctx = useContext(AppContext)

  return (
    <div style={{ padding: 30, backgroundColor: 'lightsalmon' }}>
      {/* 使用 ctx 中的数据和方法 */}
      <p>count值是: {ctx.count}</p>
      <button onClick={() => ctx.setCount((prev) => prev + 1)}>+1</button>
      <button onClick={() => ctx.setCount(0)}>重置</button>
    </div>
  )
}
```

> **核心思路**：每个 Context 都创建一个对应的 Wrapper 组件，在 Wrapper 组件中使用 Provider 向 children 注入数据。

4. 使用 useContext 重构 useReducer 案例

1. 定义 Context 要向下共享的数据的 TS 类型，代码如下：

   ```
   // 1. 定义 Context 的 TS 类型
   // 在这一步，我们必须先明确要向子组件注入的数据都有哪些
   type UserInfoContextType = { user: UserType; dispatch: React.Dispatch }
   ```

2. 使用 `React.createContext` 创建 Context 对象：

   ```
   // 2. 创建 Context 对象
   const UserInfoContext = React.createContext({} as UserInfoContextType)
   ```

3. 创建 ContextWrapper 组件如下，把 `Father` 组件中的 `useImmerReducer` 调用过程，抽离到 ContextWrapper 中：

```
// 3. 创建 ContextWrapper 组件
export const UserInfoContextWrapper: React.FC = ({ children }) => {
 const [state, dispatch] = useImmerReducer(reducer, defaultState, initAction)
 return {children}
}
```

4. 改造 Father 组件，调用 `useContext` 获取并使用 Context 中的数据。同时，Father 组件也不必再使用 props 把 `state` 和 `dispatch` 函数传递给 Son 子组件：

```
export const Father: React.FC = () => {
 // 4. 调用 useContext 导入需要的数据
 const { user: state, dispatch } = useContext(UserInfoContext)

 const changeUserName = () => dispatch({ type: 'UPDATE_NAME', payload: '刘龙彬' })

 return (
   <div>
     <button onClick={changeUserName}>修改用户名</button>
     <p>{JSON.stringify(state)}</p>
     <div className="father">
       {/* 5. 这里没有必要再往子组件传递 props 了 */}
       {/* <Son1 {...state} dispatch={dispatch} />
       <Son2 {...state} dispatch={dispatch} /> */}
       <Son1 />
       <Son2 />
     </div>
   </div>
 )
}
```

5. 改造 App 根组件，分别导入 `UserInfoContextWrapper` 和 `Father` 组件，并形成父子关系的嵌套，这样 Father 组件及其子组件才可以访问到 Context 中的数据：

```
import React from 'react'
import { UserInfoContextWrapper, Father } from
'@/components/use_reducer/01.base.tsx'

const App: React.FC = () => {
 return (
   <UserInfoContextWrapper>
     <Father />
   </UserInfoContextWrapper>
 )
}

export default App
```

6. 最后，改造 `Son1`，`Son2` 和 `GrandSon` 组件，删除 props 及其类型定义，改用 `useContext()` 来获取 `UserInfoContextWrapper` 向下注入的数据。示例代码如下：

```jsx
const Son1: React.FC = () => {
 // 6. 把 props 替换为 useContext() 的调用
 const { dispatch, user } = useContext(UserInfoContext)

 const add = () => dispatch({ type: 'INCREMENT', payload: 1 })

 return (
   <div className="son1">
     <p>{JSON.stringify(user)}</p>
     <button onClick={add}>年龄+1</button>
   </div>
 )
}

const Son2: React.FC = () => {
 // 7. 把 props 替换为 useContext() 的调用
 const { dispatch, user } = useContext(UserInfoContext)

 const sub = () => dispatch({ type: 'DECREMENT', payload: 5 })

 return (
   <div className="son2">
     <p>{JSON.stringify(user)}</p>
     <button onClick={sub}>年龄-5</button>
     <hr />
     <GrandSon />
   </div>
 )
}

const GrandSon: React.FC = () => {
 // 8. 把 props 替换为 useContext() 的调用
 const { dispatch } = useContext(UserInfoContext)
 const reset = () => dispatch({ type: 'RESET' })

 return (
   <>
     <h3>这是 GrandSon 组件</h3>
     <button onClick={reset}>重置</button>
   </>
 )
}
```