

React Hooks

Preface

这是一篇有关React和React Hooks使用的一份指南。只需要有基本的JavaScript基础和一点点基本的HTML知识便可达到基本的阅读门槛。

这篇教程的目标读者是不熟悉React Hooks的前端开发者，大致涵盖了：

1. 没有接触过任何前端框架的新手；
2. 接触过其他前端框架(比如Vue)但不熟悉React的开发者；
3. 熟悉React Class Component但不了解React Hooks的开发者。

而在我个人看来，新手对于React Hooks的思想接收速度往往是最快的；接触过Vue框架的开发者可能需要时间来适应React与之不同的开发风格；而已经熟悉了React类组件写法的开发者，想要学习Hooks在很多细节上就会感到摸不着头脑。

React Hooks与类组件完全不同，尽管用生命周期类比Hooks的运行机制可能会对曾经的React开发者有所帮助，可一旦抱定这样的信念不放手，Hooks便也失去了存在的意义。

所以我建议，在学习React Hooks时，忘掉你之前学习到的一切。想要理解Hooks的思想模型，根本不需要了解生命周期抑或是其他的一些令人头痛的概念，只需要你真正理解什么是一个function. 当你对此有正确的理解之后，你就会发现Hooks的出现和使用是多么地自然。

除此之外，大部分Hooks都会附上对应的代码实现；由于React的底层是基于Fiber结构，想要详细地解释源码是十分困难的。本文也并非是研究React Hooks如何实现的，因此，这些代码仅供读者参考，笔者不会做除简单注释以外的其他解释...

最后，所有的示例代码都可以在CodeSandBox中直接运行并查看效果。如果你想要尝试这些示例，只需要在CodeSandBox当中打开React&TS模板，在App.tsx文件中将原始代码改为章节中的示例即可。

Part.0 : $UI = f(state)$

在React当中,有一个公式贯穿始终: $UI = f(state)$. 这可以看作是React的核心所在。

这里的f函数与我们数学中定义的函数是类似的(也就是pure的)，同一个输入得到相同的输出。

这意味着，在使用React进行前端开发时，所构建的页面 (UI) 实际上是某个函数在当前状态下的返回结果。

基于这样一个理念，React采用了“一切皆函数”的JSX语法。

JSX是一个语法扩展，可以让你在JavaScript中直接使用HTML语法。JSX最终会被编译为`React.createElement()`函数调用，返回称为React元素的JavaScript对象。

来看一个简单的例子：

```
1 let MyComponent = {
2   view: function() {
3     return m("main", [
4       m("h1", "Hello world"),
5     ])
6   }
7 }
8
```

```

9 // 使用 JSX 可以写成:
10 let MyComponent = {
11     view: function() {
12         return (
13             <main>
14                 <h1>Hello world</h1>
15             </main>
16         )
17     }
18 }

```

要想使用JSX就需要在依赖中加入Babel编译器并开启JSX选项。

例如，下面这段代码

```

1 <MyButton color="blue" shadowSize={2}>
2   Click Me
3 </MyButton>

```

在Babel中会被编译为:

```

1 React.createElement(
2   MyButton,
3   {color: 'blue', shadowSize: 2},
4   'Click Me'
5 )

```

更多基本用法可以参考[JSX简介](#)。

在React中配合使用JSX，可以很好地描述UI交互的本质形式。

例如，构建一个非常简单的返回 Hello World! 一级标题的页面：

```

1 import * as React from "react";
2
3 const App: React.VFC = () => {
4     return (
5         <>
6             <h1>Hello World!</h1>
7         </>
8     );
9 };
10
11 export default App;

```

(建议在[CodeSandbox](#)中运行这些简单的demo帮助理解)

可以看到App组件就是一个JavaScript函数，`return`部分返回了我们需要的

元素。

(在React当中，默认返回的

元素可以简写为`<>...</>`)

我们也可以给App组件加入更多的内容，例如给一个数值变量`state`并将它作为正文内容出现在`h1`元素下方：

```

1 import * as React from "react";
2
3 const App: React.VFC = () => {
4     const state = 0;
5
6     return (
7         <>
8             <h1>Hello World!</h1>
9             <p>{state}</p>
10        </>
11    );
12 };
13
14 export default App;

```

尽管如此，常见的复杂用户界面使用JavaScript函数来表示仍然存在着一些困难。我们在这里展示的也都是一些比较简单的静态页面。

不过React提供了一系列的API，使得JSX在表达用户界面时，能够与常用的模板语法同样方便。这也是React Hooks的由来。

Part.01 : useState

之前，在App组件中已经引入了一个数值变量`state`，但显然这个变量在页面中是无法改变的。如果用户在一个界面当中，没有任何一个元素是可交互的，那么这个页面未免有些无趣。

我们不妨加入一个`onChange`函数，当用户每点击一次`state`，该数值便自增1。

那么，下面的写法可能是很自然的：

```
1 import * as React from "react";
2
3 const App: React.VFC = () => {
4   let state = 0;
5   const onChange = () => {
6     state++;
7   }
8   return (
9     <>
10       <h1>Hello World!</h1>
11       <p onClick={onChange}>{state}</p>
12     </>
13   );
14 };
15
16 export default App;
```

然而，在CodeSandBox上运行demo时会发现，当点击state时，数值并没有发生相应的变化。

这就要与前面提到的公式相联系了： $UI = f(state)$ 。

如果我们对state采取直接赋值，那么在编译器看来，state变量的地址并没有发生变化，可以视为是同一个state。

对于一个函数，当输入不变时，其输出也不会发生变化，于是该UI界面不会改变。

那么，想要真正触发UI的改变，需要创建一个全新的变量，并且取代原先的state出现在用户界面当中。

这也就是useState提供的功能，我们可以用useState重写上面的程序：

```
1 import * as React from "react";
2
3 const App: React.VFC = () => {
4   const [state, setState] = React.useState(0);
5   const onChange = () => {
6     setState(state => state + 1);
7   };
8
9   return (
10     <>
11       <h1>Hello World!</h1>
12       <p onClick={onChange}>{state}</p>
13     </>
14   );
15 };
16
17 export default App;
```

仔细观察上面的代码，它只有两处发生了改变：

1. 采用useState(0)的形式给state赋初值

2. 利用第二个函数参数 `setState` 来触发状态更新

`useState` 的基本实现可以简单地用下面的代码来展示：

```
1 // 赋初值时
2 function useState(initialState) {
3   let _state = initialState;
4   const setState = (newState) => {
5     _state = newState;
6     ReactDOM.render(<App />, rootElement);
7   };
8   return [_state, setState];
9 }
10
11 // 状态更新时
12 let _state;
13 function useState(initialState) {
14   _state = _state === undefined ? initialState :
15   _state;
16   const setState = (newState) => {
17     _state = newState;
18     ReactDOM.render(<App />, rootElement);
19   };
20   return [_state, setState];
21 }
```

当然，这只是对 `useState` 不完善的模仿，但核心观点却是相同的——通过 `setState` 创建一个新的 `state` 变量，并根据 `state` 变量的不同进行重渲染。

这实际上相当于每一次点击 `state` 时都创建了一个同名的新变量 `state` 并进行重新渲染：

```
1 import * as React from "react";
2
3 // 当state=0时
4 const App: React.VFC = () => {
```

```

5     const state = 0;
6
7     return (
8         <>
9             <h1>Hello World!</h1>
10            <p>{state}</p>
11        </>
12    );
13 };
14
15 //调用setState创建一个新的state=1
16 const newApp: React.VFC = () => {
17     const new_state = 1;
18
19     return (
20         <>
21             <h1>Hello World!</h1>
22             <p>{new_state}</p>
23         </>
24     );
25 };
26
27 export default newApp;

```

至此，我们介绍完了React Hooks当中最常用的`useState`。

Part.02 : useMemo

现在我们已经给App组件添加了一个动态元素`state`，并使用`useState`实现了函数式的状态更新。

那么，我们或许可以再添加一个与`state`有联结的因变量。为了方便，不妨将`state`替换为`x`，将这个因变量命名为`y`。

```

1 import * as React from "react";
2
3 const App: React.VFC = () => {

```



```

4   const [x, setX] = React.useState(0);
5   const y = 2 * x + 1;
6
7   const onChange = () => {
8     setX(x => x + 1);
9   };
10
11  return (
12    <>
13      <h1>Hello World!</h1>
14      <p onClick={onChange}>{x}</p>
15      <p>{y}</p>
16    </>
17  );
18 };
19
20 export default App;

```

在CodeSandBox上试运行时，会发现上面这段代码运行得非常完美。

但回头看我们的公式: $UI = f(state)$. 记住一点，当 `state` 与之前相同时，UI也应该是相同的，那么React就不会进行重渲染，只有当状态更新被触发，`state` 与之前所有的状态都不相同时，React才会进行重渲染。

这一点或许很容易让人想到电影中的帧，React当中的每一个状态可以类比于电影中的每一帧:每一次触发状态更新，是插入新的一帧，而不是改变原本帧的画面。那么同样，如果当前的画面与之前的某一帧相同，我们只需要插入相同的一帧而不是重新进行拍摄。

理解了这个观念之后，不难发现，每当 `setX` 创建新的UI时，无论这个UI与之前是否相同，`y` 变量都会被重新计算:

```

1  import * as React from "react";
2

```

```
3 //useState(0)赋初值
4 const App: React.VFC = () => {
5   const x = 0;
6   const y = 2 * x + 1; //第一次计算2 * 0 + 1
7
8   return (
9     <>
10      <h1>Hello World!</h1>
11      <p>{x}</p>
12      <p>{y}</p>
13    </>
14  );
15 };
16
17 //setX触发状态更新
18 const newApp: React.VFC = () => {
19   const x = 1;
20   const y = 2 * x + 1; //状态不同, 重新计算2 * 1 + 1
21
22   return (
23     <>
24      <h1>Hello World!</h1>
25      <p>{x}</p>
26      <p>{y}</p>
27    </>
28  );
29 };
30
31 //重置页面(x = 0)
32 const oldApp: React.VFC = () => {
33   const x = 0;
34   const y = 2 * x + 1;
35   //由于没有缓存该计算式和结果, 状态相同, 仍需要重新计算2
    * 0 + 1
36
37   return (
38     <>
```

```

39     <h1>Hello World!</h1>
40     <p>{x}</p>
41     <p>{y}</p>
42   </>
43 );
44 };
45
46 export default oldApp;

```

很显然，由于 `y` 变量的计算结果在 UI 刷新时，并不会像 `useState` 绑定的 `x` 变量一样被保留下来。

因此无论 `x` 变量是否发生改变，由于 **React** 的帧数刷新机制，`y` 的计算结果没有被保留，必须被重新创建和计算。

于是，当我们增加一个因变量时，还需要一个 **Hook** 帮助实现因变量的函数式更新。也就是说，`y` 的计算结果也应该能够被 **React** 记忆。这也就是 `useMemo` 的作用。

采用 `useMemo` 改写上面的程序：

```

1  import * as React from "react";
2
3  const App: React.VFC = () => {
4    const [x, setX] = React.useState(0);
5    //const y = 2 * x + 1;
6    const y = React.useMemo(() => 2 * x + 1, [x]);
7
8    const onChange = () => {
9      setX(x => x + 1);
10   };
11
12   return (
13     <>
14       <h1>Hello World!</h1>
15       <p onClick={onChange}>{x}</p>
16       <p>{y}</p>

```

```
17     </>
18   );
19 };
20
21 export default App;
```

仔细观察上面的代码，`useMemo` 实际上在实现两个功能：

1. 为`y`的计算式添加依赖项`[x]`，`React`会将`y`的计算结果与`x`绑定；
2. 监听依赖项，缓存当前依赖项下的计算结果。

熟悉`Vue`的读者可能会发现`useMemo`和`Vue`中的`computed`属性有类似之处。

我们可以用下面的代码大致实现`useMemo`的功能：

```
1  const useMemo = (memoFn, dependencies) => {
2    if (memorizedState[index]) {
3      // 不是第一次执行
4      let [lastMemo, lastDependencies] =
5        memorizedState[index];
6
7      let hasChanged = !dependencies.every((item,
8        index) => item === lastDependencies[index]);
9      if (hasChanged) {
10         memorizedState[index++] = [memoFn(),
11         dependencies];
12         return memoFn();
13       } else {
14         index++;
15         return lastMemo;
16       }
17     } else {
18       // 第一次执行
19       memorizedState[index++] = [memoFn(),
20         dependencies];
```

```
17 |     return memoFn();  
18 | }  
19 | }
```

至此，对 `useMemo` 的介绍告一段落...

Part.03 : useCallback

之前已经使用 `useState` 和 `useMemo` 实现了自变量 `x` 和因变量 `y` 的函数式更新，不过，这个程序也仍然存在着一些不符合 $UI = f(state)$ 的情况。

回顾 `useMemo` 的使用场景可能会发现，在 **React** 当中创建的函数计算式和计算结果由于帧数刷新机制，并不会保留下来，而是会重新创建和计算。

也就是说，在 **App** 组件当中，触发状态更新的 `onChange` 函数在新的状态下会被重新创建。

从性能优化的角度来看，重新创建函数对渲染性能的影响更加微妙，在很多情况下，它甚至不会对渲染性能造成负担。

也就是说，在大多数情况下，你可以不考虑这种开销会带来多少性能问题。实际上，这个时候如果采用 **Hooks** 将函数引用缓存，其额外开销远比收益要更大！

但也存在下面的情况：

有一个父组件，其中包含子组件，子组件接收一个函数作为 `props`；通常而言，如果父组件更新了，子组件也会执行更新。

在大多数场景下，这种更新是没有必要的。如果 **React** 能够将该函数引用缓存下来，子组件就能避免不必要的更新。

使用 `useCallback` 可以实现这一需求，我们在这里选择改写 `onChange` 函数，仅作演示，并不代表这就是正确的使用场景：

```
1 | import * as React from "react";
```

```

2
3 const App: React.VFC = () => {
4   const [x, setX] = React.useState(0);
5   const y = React.useMemo(() => 2 * x + 1, [x]);
6   const onChange = React.useCallback(
7     () => setX(x => x + 1), [x]
8   );
9
10  return (
11    <>
12      <h1>Hello World!</h1>
13      <p onClick={onChange}>{x}</p>
14      <p>{y}</p>
15    </>
16  );
17 };
18
19 export default App;

```

`useCallback` 与 `useMemo` 有些类似，只不过它缓存的是函数的引用而不是计算结果。

需要注意的是，`useCallback` 的使用范围比 `useMemo` 要小的多，React 社区也存在许多消灭 `useCallback` 的声音。因此，该 Hook 并不常用且争议较大。

我们可以用下面的 TypeScript 代码来实现 `useCallback` 的功能：

```

1 //初次挂载时
2 function mountCallback<T>(callback: T, deps:
  Array<mixed> | void | null): T {
3   // 添加到Fiber节点上的Hooks链表
4   const hook = mountWorkInProgressHook();
5   const nextDeps = deps === undefined ? null :
  deps;
6   // memoizedState存的值是callback

```

```

7   hook.memoizedState = [callback, nextDeps];
8   return callback;
9 }
10
11 //状态更新时
12 function updateCallback<T>(callback: T, deps:
    Array<mixed> | void | null): T {
13     // 找到该useMemo对应的hook数据对象
14     const hook = updateWorkInProgressHook();
15     const nextDeps = deps === undefined ? null :
    deps;
16     const prevState = hook.memoizedState;
17     if (prevState !== null) {
18         if (nextDeps !== null) {
19             const prevDeps: Array<mixed> | null =
    prevState[1];
20             if (areHookInputsEqual(nextDeps,
    prevDeps)) {
21                 return prevState[0];
22             }
23         }
24     }
25     hook.memoizedState = [callback, nextDeps];
26     return callback;
27 }

```

对于 `useCallback`，我们点到为止...

Part.3 $\frac{1}{2}$: Take A Break

到目前为止，我们已经学习三个与 $UI = f(state)$ 密切相关的 Hooks.

- `useState` 用于独立变量的创建和更新
- `useMemo` 用于与依赖项相关的计算结果的缓存
- `useCallback` 用于与依赖项相关的引用函数的缓存

然而，在React当中，不是所有组件都能够保持永久纯粹的。对于那些非纯组件用到的Hooks，让我们留到下期来分析讲解！

Part.04 : useRef

目前，纯函数式风格的React Hooks基本完结。但对于实际工程来说，严格意义上的纯函数未免过于理想化，也难以满足不同开发任务的需求。

因此，我们接下来会介绍两个破坏函数纯度的Hooks: `useRef` 和 `useEffect`。

不过，请记住，这些非纯的Hooks并非是异端，也不是银弹；它们的内在设计仍然遵循着React Hooks的基本原则。

完全拒绝副作用和可变性只会走向另一种极端，React Hooks不应该是学院派的试验品，而是能够在不同的解决方案之间进行取舍，以达到平衡，提供最佳的用户体验和开发者体验的成熟方案。

函数式的帧数刷新机制给React带来了数据不可变和渲染负担的减少，也为组件化提供了更好的使用体验。不过这样的机制也为一部分需要获取**即时更新值**的操作带来了麻烦。

例如，在我们当前的页面中，加入一个用户点击次数`count`。由于React新的UI当中大多数元素都是新的，因此难以对点击次数进行很好的监听和追踪。

而在常规的赋值触发变量更新操作中，我们只需要：

```
1 import * as React from "react";
2
3 const App: React.VFC = () => {
4   let state = 0;
5   let count = 0;
6   const onChange = () => {
7     state => state + 1;
8     count++;
```



```

9      }
10     return (
11       <>
12         <h1>Hello World!</h1>
13         <p onClick={onChange}>{state}</p>
14       </>
15     );
16   };
17
18   export default App;

```

尽管这段代码在React当中是无效的，但在 `onChange` 函数中，`count` 变量实现了响应式的功能。

当我们假设React采用的是这种响应式的更新机制时，也就是说，这段代码能够正常发挥作用，`count` 毫无疑问将会记录用户点击界面的次数。

当我们企图在React Hooks当中模仿该操作时，就会遇到麻烦：

```

1   import * as React from "react";
2
3   const App: React.VFC = () => {
4     const [x, setX] = React.useState(0);
5     let count = 0;
6     const isSingle = count > 1;
7     const onChange = () => {
8       setX(x => x + 1);
9       count++;
10    };
11
12    return (
13      <>
14        <h1>Hello World!</h1>
15        <p onClick={onChange}>x = {x}</p>

```

```

16      {isSingle ? <p>You have click {count}
    times</p> :
17          <p>You have click {count}
    time</p>}
18      </>
19  );
20  };
21
22  export default App;

```

此时，`count` 无法获取到最新的值。这是因为在帧数刷新当中，每一次状态更新都会创建一个新的 `count` 变量，其值为0，原有的 `count` 变量触发的改变不会保留。

那么，如果有办法能够让 **React** 提供响应式更新数据，那么想要获取到最新的值，就会非常简单。

这也就是 `useRef` 的作用。使用 `useRef` 绑定的数据将和响应式更新一样，在帧数刷新中被持续地跟踪和监听。

这时，我们就可以这样实现 `count`：

```

1  import * as React from "react";
2
3  const App: React.VFC = () => {
4    const [x, setX] = React.useState(0);
5    const count = React.useRef(0);
6    const isSingle = count.current > 1;
7    const onChange = () => {
8      setX(x => x + 1);
9      count.current++;
10   };
11
12   return (
13     <>
14       <h1>Hello World!</h1>
15       <p onClick={onChange}>x = {x}</p>

```

```

16         {isSingle ?
17             <p>You have click {count.current}
times</p> :
18             <p>You have click {count.current}
time</p>}
19         </>
20     );
21 };
22
23 export default App;

```

(上面的程序实际上是有缺陷的，在 `onChange` 函数中直接操作可变的 `count` 会使得该函数带有副作用，破坏了 `onChange` 函数的纯度。在同一输入下可能会因为 `count` 值的响应式更新机制而得到不同的输出)

`useRef` 提供了一个 `current` 属性，通过访问这一属性能够直接获取到最新的值。

`useRef` 神奇的地方在于:可以在不 `re-render` 的状态下更新值。

在首次声明时，`useRef` 和其他 `Hook` 一样创建一个 `Hook` 对象，然后创建 `current` 的初始值，缓存到 `Hook` 的 `memoizedState` 属性，并返回该值。

在 `current` 值更新时，`useRef` 直接从 `Hook` 实例中返回之前缓存的值。

也正因为如此，一般情况下，`useState` 不能用 `useRef` 来替代；并且一旦组件中出现过多的 `useRef`，将严重影响组件中数据的不可变性。

`useRef` 的实现如下：

```

1  //初次加载时
2  function mountRef<T>(initialValue: T):
    { |current: T | } {

```

```

3   const hook = mountWorkInProgressHook();
4   const ref = {current: initialValue};
5   if (__DEV__) {
6     Object.seal(ref);
7   }
8   hook.memoizedState = ref;
9   return ref;
10 }
11
12 //current值更新时
13 function updateRef<T>(initialValue: T):
14   {|current: T|} {
15   const hook = updateWorkInProgressHook();
16   return hook.memoizedState;
17 }

```

对 `useRef` 的介绍就姑且告一段落...

Part.05 : useEffect

我们来到了本文最后一个Hook，也是最有挑战性的Hook——控制副作用的 `useEffect`。

在正式进入主题之前，不妨来思考一个有趣的小话题：

为什么大多数程序都需要有副作用？

当一个函数依赖或修改它的参数之外的东西来做某事时，就会带有副作用。例如，读取或写入数据库、文件或控制台中的变量的函数可以被描述为具有副作用。

副作用在很多时候都不是一个很令人愉悦的事物。副作用往往意味着难以控制和不可预料。

如果给定相同的输入，函数总是返回相同的输出，没有任何副作用，则该函数是纯函数。

对比带有副作用的函数，纯函数在**单元测试**上有着天然的优势，因为它的结果完全取决于它的输入；纯函数下想要做**并行运行**也容易得多，因为它不依赖于对共享状态的访问。最重要的是，它是**可以预测的**：同一输入总是返回相同的输出，不管我们运行了多少次或者周围系统的状态如何。

但是，如果一个应用程序不能与外部世界交互，那么它就没有真正的用处，不是吗？

那么，我们如何处理诸如打印到控制台、从数据库读写、生成随机数等副作用呢？

答案是，我们将业务逻辑写成纯函数，并将副作用移动到流程的边缘——也就是说，不在流程中间从数据库中读取数据；在流程的最后，我们进行**IO**设计，于是我们有一个核心业务逻辑，最终也有了**IO**。

因此，函数式程序员的目标不应该是消除**IO**，而是将其移到边缘，以保持业务逻辑的纯粹性、可组合性和可测试性。

同样地，在一个**React**应用当中，我们也同样需要副作用来实现与控制台的交互。例如，我们希望在控制台打印每一次的 `x` 的值，下面的写法或许是很自然的：

```
1 import * as React from "react";
2
3 const App: React.VFC = () => {
4   const [x, setX] = React.useState(0);
5   const onChange = () => {
6     setX(x => x + 1);
7   };
8   console.log(x);
9
10  return (
11    <div>
12      <h1>Hello World!</h1>
13      <p onClick={onChange}>x={x}</p>
```

```
14     </div>
15   );
16 };
17
18 export default App;
```

如果你在CodeSandBox当中运行这段代码，貌似这段代码很好地实现了我们想要的效果。

套用之前的帧数刷新机制，由于每一次调用 `setX` 都会触发状态更新，创建新的UI界面，进而会创建新的 `console.log` 并执行。

于是每一次点击 `x` 使其自增1，都会创建新的 `console.log`，并将当前的 `x` 写入到控制台。

但这只是一个巧合，需要注意的是，每一次的状态更新，都会新建 `console.log` 并执行。这与我们的初衷并不相符，我们希望能够 `x` 的值更新时，将新的 `x` 写入控制台，而不是每一次存在状态更新时，执行 `console.log`。也就是说，这段代码之所以能够达成效果，只不过是因为该场景下的状态更新与 `x` 的值刷新是同步的。

如果将上面的代码改写成如下形式：

```
1 import * as React from "react";
2
3 const App: React.VFC = () => {
4   const [x, setX] = React.useState(0);
5   const [y, setY] = React.useState(1);
6
7   const changeX = () => {
8     setX(x => x + 1);
9   };
10
11   const changeY = () => {
12     setY(y => y + 1);
13   };
14
```

```

15     console.log(x);
16
17     return (
18         <div>
19             <h1>Hello World!</h1>
20             <p onClick={changeX}>x={x}</p>
21             <p onClick={changeY}>y={y}</p>
22         </div>
23     );
24 };
25
26 export default App;

```

此时，如果点击 `y` 元素触发状态更新；尽管 `x` 的值并没有发生变化，控制台仍会写入 `x` 的值(此处为0)，这就与我们想要的出现了偏差。

除此之外，这种写法还带来了其他问题: 我们在一个 **React** 组件当中直接写入一个副作用是十分危险的。这样会大大破坏此组件的纯度，对于单元测试和业务逻辑的可组合性造成的影响是不可逆转的。

合理地利用副作用，有两点需求:

1. 副作用需要有对应的**依赖项**，能够被严格控制在需求的范围；
2. 副作用应该在**UI渲染执行的流程最后**来处理，避免业务逻辑的纯粹性受到影响。

我们可以通过合理使用 `useEffect` 来实现上述需求:

```

1  import * as React from "react";
2
3  const App: React.VFC = () => {
4      const [x, setX] = React.useState(0);
5      const [y, setY] = React.useState(1);
6

```

```

7   const changeX = () => {
8     setX(x => x + 1);
9   };
10  const changeY = () => {
11    setY(y => y + 1);
12  };
13
14  React.useEffect(() => {
15    console.log(x);
16  }, [x]);
17
18  return (
19    <div>
20      <h1>Hello World!</h1>
21      <p onClick={changeX}>x={x}</p>
22      <p onClick={changeY}>y={y}</p>
23    </div>
24  );
25 };
26
27 export default App;

```

`useEffect` 提供了一个依赖参数，由开发者手动声明依赖项。在上面这段代码中依赖项就是 `[x]`。加入依赖项之后，`useEffect` 将会持续监听依赖项，当且仅当依赖项变化时，副作用才会触发，进而实现控制副作用的效果。

注意，这里的“监听”实际上并不准确，在 **React Hooks** 当中，有且仅有 `useRef` 始终贯穿于一个组件的不同状态。`useEffect` 在函数式组件每一个状态中都是新的，**React** 会记住你提供的 `effect` 函数，并在 **UI** 渲染完成后执行副作用。

所以虽然我们说的是一个 `useEffect`，但其实每次 **UI** 重渲染都是一个不同的函数，并且每个副作用函数“看到”的 `state` 都来自于它从属的那次特定渲染。

除此之外，有些副作用可能比较“顽固”，会影响到组件其他状态的副作用执行，进而导致副作用的失控，因此，清除副作用是十分有必要的。

而 `useEffect` 恰好隐式地提供了副作用清理，要实现这一点，`useEffect` 需返回一个清除函数。

例如，我们看官方文档中的例子：

```
1  useEffect(  
2    () => {  
3      const subscription =  
4      props.source.subscribe();  
5      //组件卸载时，清除订阅  
6      return () => {  
7        subscription.unsubscribe();  
8      };  
9    },  
10   [props.source],  
11 );
```

在卸载组件时，`useEffect` 中的订阅信息也会被清理。

需要注意的是，`useEffect` 只会在UI渲染后运行副作用，这使得你的应用更流畅，因为这样的话，大多数副作用就不会阻塞屏幕的更新。副作用的清除同样被延迟了。上一次的副作用会在重新渲染后被清除，并在这之后执行此次的副作用。

因此，整个流程如下：

1. React渲染当前状态下的UI;
2. React清除上一次的副作用;
3. React运行当前的副作用。

这种清除并不会影响到下一次的UI渲染，因为组件内的每一个函数(包括事件处理函数，副作用，定时器或者API调用等等)只会捕获定义它们的那次渲染中的 `state`。

对于之前 `useRef` 一节中并不完美的样例，我们可以用 `useEffect` 改写：

```
1 import * as React from "react";
2
3 const App: React.VFC = () => {
4   const [x, setX] = React.useState(0);
5   const count = React.useRef(0);
6   const isSingle = count.current > 1;
7   const onChange = () => {
8     setX(x => x + 1);
9   };
10
11   React.useEffect(() => {
12     count.current++;
13   }, [x]);
14
15   return (
16     <>
17       <h1>Hello World!</h1>
18       <p onClick={onChange}>x = {x}</p>
19       {isSingle ?
20         <p>You have click {count.current}
21         times</p> :
22         <p>You have click {count.current}
23         time</p>}
24     </>
25   );
26 }
27
28 export default App;
```

由于 `useEffect` 是所有 `Hooks` 中实现最复杂(同样也是功能最复杂)的一个，如果附上源码会占据极大的篇幅且意义不大，因此这里不会有源码的展示。

在掌握这5个基本的Hooks之后，我们可以写一个简单的程序，应用这几个Hooks:

```
1 import { useCallback, useEffect } from "react";
2 import { useMemo, useRef, useState } from
  "react";
3
4 export default function App() {
5   const [x, setX] = useState(0);
6
7   //const y = 2 * x + 1;
8   const y = useMemo(() => 2 * x + 1, [x]);
9
10  const changeX = useCallback(() => setX(x + 1),
    [x]);
11
12  const renderCountRef = useRef(1);
13  const isOdd = renderCountRef.current % 2 ===
    1;
14
15  useEffect(() => {
16    renderCountRef.current++;
17  });
18
19  useEffect(() => {
20    console.log(x);
21  }, [x]);
22
23  return (
24    <div className="App">
25      <ul onClick={changeX}>
26        {isOdd ? <li> x = {x} </li> : null}
27        <li> y = {y} </li>
28      </ul>
29    </div>
30  );
31 }
```

好了，`useEffect` 讲的已经够多了。让我们稍事休息，进入最后一节的学习...

Part.06 : SWR & Jotai

先暂停一下，来回顾我们之前学到的:

- `useState` 创建自变量并提供值的函数式更新
- `useMemo` 缓存函数计算结果，实现因变量值的函数式更新
- `useCallback` 缓存函数引用，实现函数式的函数引用
- `useRef` 突破状态更新机制，响应式获取和更新值
- `useEffect` 控制并清除副作用函数

在最后一个部分，我们将不会继续学习 `useContext` 或是 `useReducer` 这样的 **React** 官方提供的其他 **Hooks**，而是转向探究两个有趣的 **React** 外部库。

这是因为，对于理解 **React Hooks** 的核心理念来说，学习前面的5个 **Hooks** 便已经足够。对于我本人来说，与其继续学习其他大同小异的 **Hooks**，在 **React** 社区中寻找不同需求的 **Hooks** 化解决方案毫无疑问会更加有趣。

这里，选取了两个话题——数据请求和状态管理。

数据请求是前端业务极为重要的部分，随着前后端分离的推进，前端越来越多地承担了构建业务逻辑的任务；总的来说，前端并非是简单地实现页面样式，还需要能够向后端搭建的数据库发送请求，从而返回用户想要的结果。

在 **Hooks** 诞生之前，数据请求(简称为"取数")一般通过 **Fetch API** 或 **Axios** 来完成。

从使用上看，这两种方法都是通过调用 **Web** 原生的或外部库提供的 **API** 来发送和响应数据请求，和 **UI** 本身的交互和渲染并没有很强的关联。将"取数"这一环节放在组件之外，从数据流中请求数据，是最流行的解决方案。

然而，从业务逻辑的角度上看，取数也应该是UI交互的一部分。之所以无法直接在UI中处理数据请求，是因为一个UI组件的动态性还不够强。

得益于React Hooks的出现，函数式组件的灵活性有了很大的提升。相对地，一个组件的能力边界也得到了拓展。

那么，借助一些自定义的Hooks，我们是否能够直接在组件处理数据请求呢？

这也正是Vercel开发SWR的初衷——提供数据请求的Hooks解决方案。

在具体分析SWR之前，先看一个简单的样例：

```
1 import useSWR from "swr";
2 import React from "react";
3
4 function Profile() {
5   const { data, error } = useSWR("/api/user",
    fetcher);
6
7   if (error) return <div>failed to load</div>;
8   if (!data) return <div>loading ... </div>;
9   return <div>hello {data.name}!</div>;
10 }
```

这个例子是不完备的，也就是说它不能在CodeSandBox上面运行。

不过，幸运的是，这个例子十分简单，不需要运行也可以猜到Profile组件实现的大概功能。

TODO

我们来看SWR的基本用法：

```
1 import React from "react";
```

```

2 import useSWR from "swr";
3
4 const fetcher = (url) => fetch(url).then((res)
  => res.json());
5
6 export default function App() {
7   const { data, error } = useSWR(
8     "https://api.github.com/repos/vercel/swr",
9     fetcher
10  );
11
12  if (error) return "An error has occurred.";
13  if (!data) return "Loading ... ";
14  return (
15    <div>
16      <h1>{data.name}</h1>
17      <p>{data.description}</p>
18      <strong>👤 {data.subscribers_count}
19    </strong>{" "}
20    <strong>🌟 {data.stargazers_count}
21    </strong>{" "}
22    <strong>🔧 {data.forks_count}</strong>
23    </div>
24  );
25 }

```

TODO