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:

```
let MyComponent = {
    view: function() {
        return m("main", [
            m("h1", "Hello world"),
        ])
    }
}

```

```
// 使用 JSX 可以写成:
let MyComponent = {
    view: function() {
        return (
            <main>
                <h1>Hello world</h1>
            </main>
        )
    }
}
```

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

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

在Babel中会被编译为:

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

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:

```tsx
import * as React from "react";

const App: React.VFC = () => {
    return (
        <>
          <h1>Hello World!</h1>
        </>
    );
};

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`div`元素。

(在React当中，默认返回的`div`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：

```tsx
import * as React from "react";

const App: React.VFC = () => {
    const state = 0;

    return (
        <>
          <h1>Hello World!</h1>
          <p>{state}</p>
        </>
    );
};

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`，但很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：

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

const App: React.VFC = () => {
  let state = 0;
  const onChange = () => {
    state++;
  }
  return (
    <>
      <h1>Hello World!</h1>
      <p onClick={onChange}>{state}</p>
    </>
  );
};

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:$\mathrm{UI} = \mathrm{f}(\mathrm{state})$.

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:

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

const App: React.VFC = () => {
  const [state, setState] = React.useState(0);
  const onChange = () => {
    setState(state => state + 1);
  };

  return (
    <>
      <h1>Hello World!</h1>
      <p onClick={onChange}>{state}</p>
    </>
  );
};

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：

```
//赋初值时
function useState(initialState) {
  let _state = initialState;
  const setState = (newState) => {
    _state = newState;
    ReactDOM.render(<App />, rootElement);
  };
  return [_state, setState];
}

//状态更新时
let _state;
function useState(initialState) {
  _state = _state === undefined ? initialState : _state;
  const setState = (newState) => {
    _state = newState;
    ReactDOM.render(<App />, rootElement);
  };
  return [_state, setState];
}
```

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：

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

//当state=0时
const App: React.VFC = () => {
```

```
    const state = 0;

    return (
        <>
          <h1>Hello World!</h1>
          <p>{state}</p>
        </>
    );
};

//调用setState创建一个新的state=1
const newApp: React.VFC = () => {
    const new_state = 1;

    return (
        <>
          <h1>Hello World!</h1>
          <p>{new_state}</p>
        </>
    );
};

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`.

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

const App: React.VFC = () => {
```

```
  const [x, setX] = React.useState(0);
  const y = 2 * x + 1;

  const onChange = () => {
    setX(x => x + 1);
  };

  return (
    <>
      <h1>Hello World!</h1>
      <p onClick={onChange}>{x}</p>
      <p>{y}</p>
    </>
  );
};

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:

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

```

```
//useState(0)赋初值
const App: React.VFC = () => {
  const x = 0;
  const y = 2 * x + 1;//第一次计算2 * 0 + 1

  return (
    <>
      <h1>Hello World!</h1>
      <p>{x}</p>
      <p>{y}</p>
    </>
  );
};

//setX触发状态更新
const newApp: React.VFC = () => {
  const x = 1;
  const y = 2 * x + 1;//状态不同，重新计算2 * 1 + 1

  return (
    <>
      <h1>Hello World!</h1>
      <p>{x}</p>
      <p>{y}</p>
    </>
  );
};

//重置页面(x = 0)
const oldApp: React.VFC = () => {
  const x = 0;
  const y = 2 * x + 1;
  //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

  return (
    <>
```

```
      <h1>Hello World!</h1>
      <p>{x}</p>
      <p>{y}</p>
    </>
  );
};

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:

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

const App: React.VFC = () => {
  const [x, setX] = React.useState(0);
  //const y = 2 * x + 1;
  const y = React.useMemo(() => 2 * x + 1, [x]);

  const onChange = () => {
    setX(x => x + 1);
  };

  return (
    <>
      <h1>Hello World!</h1>
      <p onClick={onChange}>{x}</p>
      <p>{y}</p>
```

```
    </>
  );
};

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:

```
const useMemo = (memoFn, dependencies) => {
  if (memorizedState[index]) {
    // 不是第一次执行
    let [lastMemo, lastDependencies] = memorizedState[index];

    let hasChanged = !dependencies.every((item, index) => item === lastDependencies[index]);
    if (hasChanged) {
      memorizedState[index++] = [memoFn(), dependencies];
      return memoFn();
    } else {
      index++;
      return lastMemo;
    }
  } else {
    // 第一次执行
    memorizedState[index++] = [memoFn(), dependencies];
```

```
    return memoFn();
  }
}
```

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$\mathrm{UI}=\mathrm{f}(\mathrm{state})$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：

```
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
```

```
  hook.memoizedState = [callback, nextDeps];
  return callback;
}

//状态更新时
function updateCallback<T>(callback: T, deps: Array<mixed> | void | null): T {
  // 找到该useMemo对应的hook数据对象
  const hook = updateWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  const prevState = hook.memoizedState;
  if (prevState !== null) {
    if (nextDeps !== null) {
      const prevDeps: Array<mixed> | null = prevState[1];
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        return prevState[0];
      }
    }
  }
  hook.memoizedState = [callback, nextDeps];
  return callback;
}
```

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$\mathrm{UI} = \mathrm{f}(\mathrm{state})$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++;
```

```
  }
  return (
    <>
      <h1>Hello World!</h1>
      <p onClick={onChange}>{state}</p>
    </>
  );
};

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:

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

const App: React.VFC = () => {
  const [x, setX] = React.useState(0);
  let count = 0;
  const isSingle = count > 1;
  const onChange = () => {
    setX(x => x + 1);
    count++;
  };

  return (
    <>
      <h1>Hello World!</h1>
      <p onClick={onChange}>x = {x}</p>
```

```
      {isSingle ? <p>You have click {count} times</p> :
                  <p>You have click {count} time</p>}
    </>
  );
};

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`：

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

const App: React.VFC = () => {
  const [x, setX] = React.useState(0);
  const count = React.useRef(0);
  const isSingle = count.current > 1;
  const onChange = () => {
    setX(x => x + 1);
    count.current++;
  };

  return (
    <>
      <h1>Hello World!</h1>
      <p onClick={onChange}>x = {x}</p>
```

```
      {isSingle ?
        <p>You have click {count.current} times</p> :
        <p>You have click {count.current} time</p>}
    </>
  );
};

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:

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

```
  const hook = mountWorkInProgressHook();
  const ref = {current: initialValue};
  if (__DEV__) {
    Object.seal(ref);
  }
  hook.memoizedState = ref;
  return ref;
}

//current值更新时
function updateRef<T>(initialValue: T):
{|current: T|} {
  const hook = updateWorkInProgressHook();
  return hook.memoizedState;
}
```

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:

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

const App: React.VFC = () => {
  const [x, setX] = React.useState(0);
  const onChange = () => {
    setX(x => x + 1);
  };
  console.log(x);

  return (
    <div>
      <h1>Hello World!</h1>
      <p onClick={onChange}>x={x}</p>
```

```
    </div>
  );
};

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，我们希望能够在`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:

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

const App: React.VFC = () => {
  const [x, setX] = React.useState(0);
  const [y, setY] = React.useState(1);

  const changeX = () => {
    setX(x => x + 1);
  };

  const changeY = () => {
    setY(y => y + 1);
  };

```

```
  console.log(x);

  return (
    <div>
      <h1>Hello World!</h1>
      <p onClick={changeX}>x={x}</p>
      <p onClick={changeY}>y={y}</p>
    </div>
  );
};

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:

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

const App: React.VFC = () => {
  const [x, setX] = React.useState(0);
  const [y, setY] = React.useState(1);

```

```
  const changeX = () => {
    setX(x => x + 1);
  };
  const changeY = () => {
    setY(y => y + 1);
  };

  React.useEffect(() => {
    console.log(x);
  }, [x]);

  return (
    <div>
      <h1>Hello World!</h1>
      <p onClick={changeX}>x={x}</p>
      <p onClick={changeY}>y={y}</p>
    </div>
  );
};

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:

```
useEffect(
  () => {
    const subscription =
props.source.subscribe();
    //组件卸载时，清除订阅
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

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:

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

const App: React.VFC = () => {
  const [x, setX] = React.useState(0);
  const count = React.useRef(0);
  const isSingle = count.current > 1;
  const onChange = () => {
    setX(x => x + 1);
  };

  React.useEffect(() => {
    count.current++;
  }, [x]);

  return (
    <>
      <h1>Hello World!</h1>
      <p onClick={onChange}>x = {x}</p>
      {isSingle ?
        <p>You have click {count.current} times</p> :
        <p>You have click {count.current} time</p>}
    </>
  );
};

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:

```
import { useCallback, useEffect } from "react";
import { useMemo, useRef, useState } from
"react";

export default function App() {
  const [x, setX] = useState(0);

  //const y = 2 * x + 1;
  const y = useMemo(() => 2 * x + 1, [x]);

  const changeX = useCallback(() => setX(x + 1),
[x]);

  const renderCountRef = useRef(1);
  const isOdd = renderCountRef.current % 2 ===
1;

  useEffect(() => {
    renderCountRef.current++;
  });

  useEffect(() => {
    console.log(x);
  }, [x]);

  return (
    <div className="App">
      <ul onClick={changeX}>
        {isOdd ? <li> x = {x} </li> : null}
        <li> y = {y} </li>
      </ul>
    </div>
  );
}
```

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：Algebraic Effects

在本文的最后一部分，我希望谈一些更“形而上”的内容，或者说是一些有关系统设计的哲学(
The Software Foundations of React Hooks).

在本节，我可能会不加说明地描述类组件，并使用一门与JavaScript大不相同的语言进行演示。

React的核心原则之一是，应用程序的用户界面是应用程序状态的纯函数。在这里，“状态”可以指本地组件状态和全局状态的任何组合，例如 Redux store. 当状态发生变化并通过组件树传播时，输出表示状态发生变化后的新UI. 当然，这是对更新实际发生的具体细节的抽象，因为React需要处理必要的实际协调和DOM更新。但这个核心原则意味着，至少在理论上，我们的UI总是与数据同步的。

当然，这并不总是正确的。如果我们不能有效地处理生命周期方法中的状态更改，类组件会暴露某些场景，使得我们忽略状态更改。简而言之，类组件使用不同的生命周期方法来处理副作用，将副作用映射到DOM操作，而不是状态更改。这意味着，虽然我们UI的可视元素可能会响应状态变化，但我们的副作用可能不会。

因此组件必须执行这些内部更新，以便在`props`更改时同步它们的内部状态。所以根据定义，它们是不纯的。但是在React看来，UI应该要是一个纯粹的状态函数。

这就是Hooks起作用的地方。

Hooks代表了一种不同的思考“效应”的方式。与考虑组件的整个生命周期不同，Hooks允许我们将注意力集中在当前状态上。然后我们可以声明我们希望“效应”运行的状态，确保这些状态变化反映在我们的UI中。

当然，“效应”可以是很多东西，使用`useState`处理状态、使用`useEffect`发出网络请求或者手动更新`DOM`，又或者使用`useCallback`计算代价高昂的回调函数。

尽管这种效应在React中不一定是副作用，但正如我们之前提到的，在JavaScript中赋值仍然是触发状态更新的常用手段，因此，它也可以视为是一种副作用，因此在下文中，“效应”与副作用同义。

那么我们如何在一个纯函数中推理出这些副作用呢?

这就需要借助PLT中的一个有趣的概念Algebraic Effects.

Algebraic Effects是在纯上下文中通过定义一个效应、一组操作和一个效应处理程序来推理计算效应的一种广义方法，它负责处理如何实现效应的语义。

Algebraic Effects的一个常见用例是处理有状态计算。请记住，效应只存在于一个带有一组操作的接口中。

遗憾的是，JavaScript本身并不支持Algebraic Effects，我们这里采用一种与OCaml语法类似的语言Eff来演示(不了解OCaml也不必担心，我将对程序进行必要的解释):

```
(* state.eff *)

(* A user with a name and age *)
type user = string * int

effect Get: user
effect Set: user → unit
```

在Eff中，我们使用`effect`关键字和类型签名定义副作用。在这里，我们定义了两个副作用`Get`和`Set`，它们与JavaScript中的`getter`和`setter`类似。

一旦我们定义了副作用的样子，我们就可以通过使用`handler`关键字来定义如何处理副作用。

```
let state = handler
  | y → fun currentState → (y, currentState)
  | effect Get k → (fun currentState → (continue k currentState) currentState)
  | effect (Set newState) k → (fun _ → (continue k ()) newState)
;;
```

这看起来有点棘手 让我们分解一下。我们有一个具有三个分支的处理程序，它们都返回一个函数。该函数将用于处理某些副作用。

第一个分支`y -> fun currentState -> (y, currentState)`表示当我们到达要处理的块的末尾时(稍后将看到)不存在副作用的情况。`y`是函数的返回值，所以它只返回内部返回值和状态的一个元组。

第二个和第三个分支匹配我们的副作用，并执行相应的副作用。

这里有一个特殊参数`k`，它是一个`continuation`，它表示在我们执行副作用之后计算的其余部分。

因为`continuation`表示正在运行的整个进程，所以它们实质上是在执行副作用时调用堆栈的快照。当我们执行一个副作用时，几乎就像我们在计算中按下了一个巨大的暂停按钮，直到我们正确地处理了这个副作用。调用`continue k`就像再次点击播放按钮一样。

好了，我认为我们已经准备好我们的副作用处理程序。现在，我们有一个地区的用户; 让我们在他们的生日那天祝福他们:

```
let celebrate = with state handle
  let (name, age) = perform Get in

  print_string "Happy Birthday, ";
  print_string name;
  print_endline "!";

  perform (Set (name, age + 1));
  perform Get
;;

celebrate(("ZhangSan", 39));;
```

当我们开始计算时，我们首先从`state`获取用户，`state`运行处理程序中的第二个分支。此时，我们已经按下了暂停按钮，因此当我们从`state`获取这个值时，函数已经停止运行。处理程序返回一个函数，该函数调用`continue k currentState`，使用`currentState`值恢复我们的计算。

每当我们`perform`一个副作用时，这个流程就会重复一次。

在这里，代数效应的力量真正显现出来。**我们怎么处理`state`并不重要**。

在这里，它只是内存中的一个对象，但是如果它是在数据库中呢，如果它存储在浏览器的`localStorage`中又会怎样?

就`celebrate`而言，这些都是一样的。如果需要，我们甚至可以用存储在键值存储中的`redisState`处理程序来替换`state`处理程序。

在JavaScript中，你的代码必须知道什么是同步的，什么是异步的。如果这在未来发生变化，并且状态被异步处理，那么我们将需要开始处理`Promises`，这将需要对涉及此函数的所有内容进行更改。

但是，通过代数效应，我们可以简单地完全停止当前的进程，直到我们的效应完成，而不是维护一个持有对不同进程的引用的正在运行的进程。

这也正是React Hooks帧数刷新机制的来源!

当然，状态并不是我们能够使用代数效应的唯一场景。假设我们有一些需要创建的网络请求或者需要执行的清除操作，但是我们只希望在函数完成之后执行。我们称之为延迟效应。

```
effect Defer: (unit → unit) → unit

let defer = handler
    | y → fun () → ()
    | effect (Defer effectFunc) k →
        (fun () →
            continue k ();
            effectFunc ()
        )
;;
```

注意，`continue k ()`不一定是处理程序的最后一部分，因为它在我们的状态处理程序中。我们可以随时随地调用`continuations`. 请记住，它们只是一个过程的表示。

为了确保这样做的效果符合预期，让我们快速描述一下这样做在实践中可能的效果:

```
let runWithCleanup = with defer handle
    print_endline "Starting our computation";
    perform (Defer fun () → print_endline
"Running cleanup");
    (* Do some work *)
    print_endline "Finishing computation"
;;

runWithCleanup();;
```

我们在Eff中列出的两个副作用存在于React中，只是名称不同:

- `state`处理程序对应`useState`;
- 而我们的`defer`处理程序的工作方式非常类似于简化的`useEffect`.

之前的例子与用户界面没有直接关系，但是暂停和恢复过程的心智模型，以及延续之后的调度效果，是理解Hooks和React的核心。

所以让我们把注意力转回到React上来。我们已经清楚了Hooks的好处，那么如何从代数效应的角度会看Hooks?

回想一下我们最初对代数效应的定义，它是一组操作和一组效应处理程序。这里的操作是我们的Hooks(即`useState`、`useEffect`等)，React在渲染期间处理这些副作用。

由于Hooks的一些规则，我们知道效应处理程序是React渲染循环的一部分。例如，如果你尝试在React组件之外调用`useEffect`，你可能会得到类似`Invalid hook`这样的错误。

Hooks只能在函数组件的主体内部调用。同样地，如果你在Eff中执行一个副作用而没有正确处理它，你会看到`Runtime error: uncaught effect Defer`.

正如我们必须在Eff中设置处理程序，在React中，它们被设置为渲染周期的一部分。

React Hooks通过分离副作用和渲染，减轻程序的部分复杂性。

这一点至关重要，React中越来越多的功能，如Suspense和Concurrent Mode的心智模型与代数效应相吻合。

好了，我们的最后一节，就到此结束了！！！

Epilogue

或许很多读者会对最后一节的内容感到困惑，为什么要选择Algebraic Effects这样一个话题而不是讲解更多更新的Hooks呢?

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。

而通常最痛苦的`bug`来自于我们使用工具的方式与它的心智模型之间的误差。对于许多React开发人员来说，当我们使用称之为`useState`的Hook时想要理解究竟发生了什么是很困难的。

我的希望是，至少理解代数效应可以为Hooks在幕后所做的工作提供一个好的模型。当然，值得重申的是，这并不意味着Hooks实际上就是这样工作的——它只是简单地尝试并理解它们的一种理论模型。

这篇文章没有深入到React的内部工作原理，但是希望它提供了一个更好的关于Hooks和副作用的直觉。

代数效应在程序语言设计理论中是一个算得上年轻的领域，幸运的是，它得到了一定的关注，并被应用在一些实际场景之中。

作为一名函数式编程爱好者和PLT学习者，这令我感到激动不已。

许多人第一眼看到JSX都有种劝退的感觉。但熟悉过后，时间越久，这种把UI当作 “值” 来思考和处理的方式就越令人感到震撼。React一直将自己定位成library(甚至可以理解为function)而非framework. JSX和render function是其中精髓。而最近的Hooks进一步加强了这一点。

好的工具可以影响创作者的思想方式，而不仅是让你更方便地写代码。

尽管社区中有些人抱怨说React变成了一个黑匣子，但重要的是要记住，新工具的诞生是有原因的，而Hooks和React的主要目标是让我们避开一部分我们不想处理的复杂性，让我们专注于构建更好的的UI和满足我们的用户。

回到根源，我并不选择去关注某一堆代码，而是它背后一群有创造力的人（和他们的天才想法）。

Dan Abramov、Evan You、Cheng Lou、Andrew Clark、Jordan Walke、Ryan Florence、Sebastian Markbåge、Christopher Chedeau...每一个人都值得我bet on !