React Hooks学习指北

解决方案A组FE： 边星培

不知道初学React的你是否跟我一样：

- 曾经为组件中的this指向而晕头转向
- 为该使用无状态组件还是有状态、function还是class而烦恼
- 记不住各个又臭又长生命周期命名，还看不明白正确的使用方法

现在，随着React Hooks的发布，你会发现React核心团队正在向react框架使用者传递一个信号：相比于Class，大家应该更倾向于使用函数式组件写法。
从这点上就可以说明Hooks对react开发影响是巨大的，值得所有使用者学习与应用。

本文不会对hooks做太详细介绍（请先移步十全大补文档），只整理一些个人对hooks的学习和认知，旨在让更多同学认识到它的强大，能在下一个项目中用hooks取代class、并写出自己的hooks。

什么是 hooks?

官方文档写的很清晰，它的出现主要是为了解决三个问题：

> It's hard to reuse stateful logic between components
> Complex components become hard to understand
> Classes confuse both people and machines

Hooks本质上就是一类特殊的函数，它们可以为你的函数式组件注入一些特殊的功能。它所解决的问题，是在**函数内使用生命周期和state**，并且还能把逻辑提取出来复用，让代码共享的是数据处理逻辑，但不共享数据本身。

官方hooks

官方hooks列表。了解了hooks基本概念，首先要熟悉官方最基本的hooks使用方法。下面主要讲几个最常见的：

useState 管理状态

---

useState可以让函数式组件也拥有状态

```
function App () {
  const [value, setValue] = useState(initValue);
  const [moreValue, setMoreValue] = useState();
  ...

  return <Button onClick={() => setValue(newValue)}></Button>
}
```

这里通过数组结构产生一个状态value和更新状态的函数setValue，后者一定是更新前者，前者一定是被后者更新。函数执行完毕，状态会被react记住而不是被销毁。

值得注意的是:

- 看到的所有示例，value都是一个非常细粒度的值，当你需要维护多个状态时，推荐多次调用useState，每次useState产生的状态都相互独立。
- useState返回的第二个改变状态函数，是**覆盖式**的，而非react原本setState非覆盖式更新，所以当你的value是对象时，改变状态时要小心
- 根据官方文档，当你需要维护复杂状态逻辑时，可以使用useReducer

useEffect 处理副作用

这是被讨论最多的hook，也最难理解、也是最重要的hooks。按照Dan Abramov的说法，要想真正理解并使用它，要先忘记你已经学到的。。。:

> “Unlearn what you have learned.” – Yoda

之前我们写有状态的组件，通常会产生很多side effect，比如ajax请求，一些事件监听的注册与销毁，修改dom等等。之前的做法是把这些副作用写在如componentDidMount，componentDidUpdate和componentWillUnmount这些生命周期钩子里，而useEffect所能做的就是这些生命周期钩子的集合体。

```
  useEffect(() => doSomeSideEffectThings(), [watchVal] );
```

- useEffect 接受一个匿名函数，这个函数就是副作用。当组件首次渲染和之后的每次渲染都会**异步执行**这个匿名函数，所以不会阻塞浏览器更新，对性能不会造成影响。对于少数需要先获取DOM尺寸计算的场景，useEffect并不适用(可使用useLayoutEffect这个hook)。
- useEffect第二个参数是来告诉react只有当这个参数值发生改变时，才执行副作用，这就相当于取代了componentDidUpdate。当参数为空数组`[]`时，相当于首次渲染的时候执行。
- useEffect返回函数在组件销毁前执行(componentWillUnmount)

一个模拟生命周期的🌰:

```
function App() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    console.log('componentDidMount', count);
  }, [count] ); // count变化则触发这个副作用

  useEffect(() => {
      timer = setInterval(() => {
      setCount(prev => prev + 1);
      }, 1000);
    // 需要清理的副作用，可在effect中返回一个function
    return () => {
        console.log('componentWillUnmount', count);
        clearInterval(timer);
    };
  },[]);
}
```

这里推荐大家阅读useEffect完整指南，写的非常细致，值得背诵全文。

useRef DOM引用

```
  const inputEl = useRef(initValue);
  return (
      <input ref={inputEl} type="text">
  )
```

useRef类似React.createRef(), 可以通过inputEl.current 访问组件真实DOM节点，从而进行DOM操作。
但是useRef使用场景可没这么简单，更实用的地方是它可以利用DOM实例去存储任何改变的值。

下面是一个利用useRef 记录历史、展示实时值的🌰:

```
function App() {
  const [count, setCount] = useState(0)
  const inputEl = useRef("");

  const showMessage = () => {
    console.log(inputEl.current);
  };

  const handleSendClick = () => {
      // 需要打印实时的input输入值 而不是3秒前的
    setTimeout(showMessage, 3000);
  };

  const handleMessageChange = e => {
    setCount(e.target.value);
  };

  useEffect(() => {
   // 由于useEffect中的函数是在render完成之后异步执行的，所以在每次render时
inputEl.current的值为上一次的count值
    inputEl.current = count;
  }, [count]);
  const prevCount = inputEl.current;
```

```
  return (
    <>
      <input ref={inputEl} onChange={handleMessageChange} />
      <button onClick={handleSendClick}>Send</button>
      <h5>Now: {count}, before: {prevCount}</h5>
    </>
  );
}
```

useCallback 记忆函数

在之前学习react中，最常见的错误有这样：

```
class App {
    render() {
        return <div>
            <SomeComponent style={{ fontSize: 14 }} doSomething={ ()
          => { console.log('do something'); }}  />
        </div>;
    }
}
// or
function App() {
  const handleClick = () => {
    console.log('Click happened');
  }
  return <SomeComponent onClick={handleClick} />;
}
```

这样写的问题在于，一旦app组件prop或者状态改变，就会触发重渲染，即使这个函数和上层改变不相关。而通过使用useCallback，我们可以手动控制是否需要重渲染：如果是无关props改变，useCallback直接会返回之前记忆的方法，当依赖的props变化，则触发重渲：

```
function App() {
  const memoizedClick = useCallback(() => {
    console.log('Click happened')
  }, [someProps]); // 空数组代表无论什么情况下该函数都不会发生改变
  return <SomeComponent onClick={memoizedClick} />;
}
```

还是一个🌰：

```
function App() {
  let [count, setCount] = useState(1);
  // num 是需要记忆的内容
  let [num, setNum] = useState(1);
  const memoized = useCallback( () => {
    return num;
  },[count]); // count 变化用来控制是否记忆

return (
    <>
       <button onClick={() => {setCount(count + 1)}}> memo </button>
       <button onClick={() => {setNum(num + 1)}}> num+ </button>
       <h3>记忆的num: {memoized()} || 真实num: {num} </h3>
    </>
    )
}
```

第二个参数是传入一个数组，数组中的一项改变，才会**刷新记忆**，上面的代码useCallback订阅了count的变化，只有count改变才会返回新的记忆函数以提供给后面进行渲染。

useMemo 记忆值

useMemo和useCallback非常相似，区别是useCallback不会执行第一个参数函数，而是将这

个**参数函数返回**；useMemo则执行这个参数函数并将**执行结果返回**。
所以useCallback适合函数的记忆，而useMemo能得到的是结果，对需要得到确定值/组件的场景下使用。

 ：

```
function Parent({ a, b }) {
  // a变=> 渲染child1 ,  b变渲染child2
  const child1 = useMemo(() => <Child1 a={a} />, [a]);
  const child2 = useMemo(() => <Child2 b={b} />, [b]);
  return (
    <>
      {child1}
      {child2}
    </>
  )
}
```

自己写Hooks？

满足DRY原则，为了让我们的代码更优雅更健壮，项目中自己封装hook是必不可少的。
这里首先请看十全大补文档，官方已经提供了如何封装hook指南, 看完还不会（比如我）可以学习一下社区内一些优秀的轮子：

- usehooks.com
- react-use
- react-hanger
- awesome-react-hooks

未来

就像昨天**@李淳老师**分享时所说，react16.8发布的hooks:

> 更FP，更新粒度更更细，代码更清晰

> 很大程度上可能是react未来主流编程模式

听到没有，这就是未来，你不陪跑也得陪跑。赶紧在你的项目中新建一个src/hooks，改造你的下一个项目吧