不会封装hook？ 看下ahooks这6个hook是怎么做的

1 useUpdate

在react函数组件中如何强制组件进行刷新？ 虽然react没有提供原生的方法，但是我们知道当state值变化的时候，react函数组件就会刷新，所以useUpdate就是利用了这一点，源码如下：

```
1 javascript
2 复制代码
3 import { useCallback, useState } from 'react';const useUpdate = () => {  const
  [, setState] = useState({});  return useCallback(() => setState({}),
  []);};export default useUpdate;
```

可以看到useUpdate的返回值函数，就是每次都用一个新的对象调用setState，触发组件的更新。

2 useMount

react函数组件虽然没有了mount的生命周期，但是我们还会有这种需求，就是在组件第一次渲染之后执行一次的需求，就可以封装useEffect实现这个需求， 只需要把依赖项设置成空数组，那么就只在渲染结束后，执行一次回调：

```
1 javascript
2 复制代码
3 import { useEffect } from 'react';const useMount = (fn: () => void) => {
  useEffect(() => {    fn?.();  }, []);};export default useMount;
```

3 useLatest

react函数组件是一个可中断，可重复执行的函数，所以在每次有state或者props变化的时候，函数都会重新执行，我们知道函数的作用域是创建函数的时候就固定下来的，如果其中的内部函数是不更新的，那么这些函数获取到的外部变量就是不会变的。例如：

```
1 javascript
2 复制代码
```

```
3 import React, { useState, useEffect } from 'react';import { useLatest } from
  'ahooks';export default () => {  const [count, setCount] = useState(0);
  useEffect(() => {    const interval = setInterval(() => {      setCount(count
  + 1);    }, 1000);    return () => clearInterval(interval);  }, []);  return (
     <>      <pcount: {count}</p    </>  );};
```

这是一个定时更新count值的例子，但是上边的代码只会让count一直是1，因为setInterval中的函数在创建的时候它的作用域就定下来了，它拿到的count永远是0, 当执行了setCount后，会触发函数的重新执行，重新执行的时候，虽然count值变成了1，但是这个count已经不是它作用域上的count变量了，函数的每次执行都会创建新的环境，而useState， useRef 等这些hooks 是提供了函数重新执行后保持状态的能力， 但是对于那些没有重新创建的函数，他们的作用域就永远的停留在了创建的时刻。如何让count正确更新， 简单直接的方法如下，在setCount的同时，也直接更新count变量，也就是直接改变这个闭包变量的值，这在JS中也是允许的。

```
1 javascript
2 复制代码
3 import React, { useState, useEffect } from 'react';import { useLatest } from
  'ahooks';export default () => {  let [count, setCount] = useState(0);
  useEffect(() => {    const interval = setInterval(() => {    count = count + 1
       setCount(count + 1);    }, 1000);    return () =>
  clearInterval(interval);  }, []);  return (    <>      <pcount: {count}</p
  </>  );};
```

setCount是为了让函数刷新，并且更新函数的count值，而直接给count赋值，是为了更新定时任务函数中维护的闭包变量。 这显然不是一个好的解决办法，更好的办法应该是让定时任务函数能够拿到函数最新的count值。 useState返回的count每次都是新的变量，也就是变量地址是不同的，应该让定时任务函数中引用一个变量地址不变的对象，这个对象中再记录最新的count值，而实现这个功能就需要用到了useRef，它就能帮助我们在每次函数刷新都返回相同变量地址的对象, 实现方式如下：

- javascript
  复制代码
  import React, { useState, useEffect, useRef } from 'react'export default () => { const [count, setCount] = useState(0) const latestCount = useRef(count) latestCount.current = count useEffect(() => {   const interval = setInterval(() => {   setCount(latestCount.current1)   }, 1000)   return () => clearInterval(interval) }, []) return (<>   <pcount: {count}</p   </>)}

可以看到定时函数获取的latestCount永远是定义时的变量，但因为useRef，每次函数执行它的变量地址都不变，并且还把count的最新值，赋值给了latestCount.current, 定时函数就可以获取到了最新的count值。 所以这个功能可以封装成了useLatest，获取最新值的功能。

```
1 csharp
```

```
2 复制代码
3 import { useRef } from 'react';function useLatest<T(value: T) {  const ref =
  useRef(value);  ref.current = value;  return ref;}export default useLatest;
```

上边的例子是为了说明useLatest的作用，但针对这个例子，只是为了给count+1，还可以通过setCount方法本身获取，虽然定时任务函数中的setCount页一直是最开始的函数，但是它的功能是可以通过传递函数的方式获取到最新的count值，代码如下：

```
1 scss
2 复制代码
3   const [count, setCount] = useState(0)  useEffect(() => {    const interval =
  setInterval(() => {      setCount(count=>count+1)    }, 1000)    return () =>
  clearInterval(interval)  }, [])
```

4 useUnMount

有了useMount就会有useUnmount，利用的就是useEffect的函数会返回一个cleanup函数，这个函数在组件卸载和useEffect的依赖项变化的时候触发。 正常情况 我们应该是useEffect的时候做了什么操作，返回的cleanup函数进行相应的清除，例如useEffect创建定时器，那么返回的cleanup函数就应该清除定时器：

```
1 scss
2 复制代码
3  const [count, setCount] = useState(0); useEffect(() => {    const interval =
  setInterval(() => {    count = count + 1      setCount(count + 1);    },
  1000);    return () => clearInterval(interval);  }, []);
```

所以useUnMount就是利用了这个cleanup函数实现useUnmount的能力，代码如下：

```
1 javascript
2 复制代码
3 import { useEffect } from 'react';import useLatest from '../useLatest';const
  useUnmount = (fn: () => void) => {  const fnRef = useLatest(fn);  useEffect(
   () => () => {      fnRef.current();    },    [],  );};export default
  useUnmount;
```

使用了useLatest存放fn的最新值，写了一个空的useEffect，依赖是空数组，只在函数卸载的时候执行。

5 useToggle和useBoolean

useToggle 封装了可以state在2个值之间的变化，useBoolean则是利用了useToggle，固定2个值只能是true和false。 看下他们的源码：

```
ini
复制代码
function useToggle<D, R>(defaultValue: D = false as unknown as D,
reverseValue?: R) {  const [state, setState] = useState<D | R>(defaultValue);
const actions = useMemo(() => {    const reverseValueOrigin = (reverseValue
=== undefined ? !defaultValue : reverseValue) as D | R;    const toggle = ()
=> setState((s) => (s === defaultValue ? reverseValueOrigin : defaultValue));
  const set = (value: D | R) => setState(value);    const setLeft = () =>
setState(defaultValue);    const setRight = () =>
setState(reverseValueOrigin);    return {      toggle,      set,      setLeft,
    setRight,    };    // useToggle ignore value change    // },
[defaultValue, reverseValue]);  }, []);  return [state, actions];}
```

可以看到，调用useToggle的时候可以设置初始值和相反值，默认初始值是false，actions用useMemo封装是为了提高性能，没有必要每次渲染都重新创建这些函数。setLeft是设置初始值，setRight是设置相反值，set是用户随意设置，toggle是切换2个值。 useBoolean则是在useToggle的基础上进行了封装，让我们用起来对更加的简洁方便。

```
ini
复制代码
export default function useBoolean(defaultValue = false): [boolean, Actions] {
 const [state, { toggle, set }] = useToggle(defaultValue);  const actions:
Actions = useMemo(() => {    const setTrue = () => set(true);    const
setFalse = () => set(false);    return {      toggle,      set: (v) =>
set(!!v),      setTrue,      setFalse,    };  }, []);  return [state,
actions];}
```

总结

本文介绍了ahooks中封装的6个简单的hook，虽然简单，但是可以通过他们的做法，学习到自定义hook的思路和作用，就是把一些能够重用的逻辑封装起来，在实际项目中我们有这个意识就可以封装出适合项目的hook。