React中的useRef

useState中遇到的问题

说道 useRef 那么我们先来看看 useState 存在的"问题"。

```js
import React, { useState, useEffect } from 'react';

export default function Demo() {
  const [like, setLike] = useState(0);

  useEffect(() => {});

  const handleClick = () => {
    setLike(like + 1);
  };

  const getLikeValue = () => {
    setTimeout(() => {
      alert(like);
    }, 2000);
  };

  return (
    <div>
      <button onClick={handleClick}>+</button>
      <button>{like} 👍</button>
      <button onClick={getLikeValue}>获得like值</button>
    </div>
  );
}
```

这是一个非常经典的例子，页面上渲染出来三个 button。当我点击 `+` 之后，页面重新渲染为1。

此时当我点击 `获得like值` 按钮，因为定时器的原因并不会立即执行 `alert`，此时我再次点击 `+` 修改 like。

当两秒过后，你会发现页面上展示的最新的like值，而 `alert` 弹出的 like 停留到了1。

先说结论：

> 当Demo函数每次运行我们都称他为每一次渲染，每一次渲染函数内部都拥有自己独立的 `props` 和 `state`，当在jsx中调用代码中的state进行渲染时，每一次渲染都会获得各自渲染作用域内的 `props` 和 `state` 。

对比vue更新原理的差异

实质上这里和vue实现响应式的原理是完全不同的，我们都知道在vue3中是通过proxy，当修改响应式值的时候会触发对应的set函数从而触发更新，并且运行对应收集的effect进行模版更新。

而在react这里state中的like仅仅是渲染函数中一个定义的数字而已。它并不是什么proxy,watcher,effect...它仅仅是表示一个数字而已。

当我们第一次调用函数，like赋予初始化值是0，当我们点击按钮调用setLike，react会再次渲染组件(运行Demo函数)。此时新函数内部的like是1，然后使用内部这个值重新调用Demo函数进行页面渲染。如此类推，就好比下方这段代码:

```
js复制代码
const like = 2 // Final value
// ...
<p>{like}</p>
// ...

// During first render
function Counter () {
  const like = 0
  // ...
  <p>{like}</p>
  // ...
}

// After a Click, our function is called again
function Counter () {
  const like = 1
  // ...
  <p>{like}</p>
  // ...
}

// After another click, our function is called again
function Counter () {
  const like = 2
  // ...
  <p>{like}</p>
  // ...
}

// ...
```

结论分析

当我们每次更新状态的时候，(修改state值)。react会重新渲染组件，每一次渲染都可以拿到独立的like状态，这个状态值是独立于每次渲染函数中的一个常量，它的作用仅仅只是渲染输出，插入jsx中的数字而已。

你可能会疑惑每次调用函数的like值是哪里来的，新的like值是由react提供，当我们调用setLike修改它的值的时候。react会带着新的值去重新运行函数进行再次渲染，保证渲染和输出一致。

这里有一个关键点，任意一次渲染周期(函数调用)中的 `state/prop` (直观来说就是like值)都不会随着时间改变，因为每次调用渲染函数中的like值都是一个常量(在各自的渲染函数作用域内)。

渲染输出会变化是因为组件函数被一次次调用，而每一次调用引起的渲染函数中包含的like值都是函数内部互相独立的。

这就是为什么setTimeout中拿到的仍然是1而不是最新的like。因为闭包的原因，当我们点击getLikeValue的时候获取的是当次渲染函数内部的like值，谨记每次渲染state和prop都是相互独立的（因为是各自函数作用域内的变量），每次独立渲染函数中的state和prop都是保持不变的常量。

每次改变 `state/props` 造成函数组件重新执行，从而每次渲染函数中的 `state/props` 都是独立的，固定的。

注意这里的固定和独立这两个关键字。

- 固定表示函数在开始执行时已经确定了本次 `state/props` 的值。(特别注意上文setTimeout因为闭包的原因访问的是固定的，已经确定的state)。
- 独立表示每次运行函数的state/props都是各自独立作用域中的。

useRef

上边我们说到关于state和props在不同渲染中的独立性，这个时候就引出了我们的主角useRef。 useRef日常主要有两种作用，我们先来说说刚才关于state碰到的问题，使用 `useRef` 来如何解决。

useRef作用一：多次渲染之间的纽带

之前通过 `state` 我们了解了，react中每一次渲染它的 `state props` 都是相互独立的，于是自然而然我们想到如何在每一次渲染之间产生关系呢。这个时候useRef就展示了他的作用。

我们先来看看关于useRef在react中返回值的类型定义:

ts复制代码

```ts
interface MutableRefObject<T> {
  current: T;
}
```

可以看到useRef返回值是一个包括属性current类型为范型 `<T>` 的一个object。

它与直接在function compoent中定义一个 `{ current:xxx }` 的区别就是。

useRef会在所有的render中保持对返回值的唯一引用。因为所有对ref的赋值和取值拿到的都是最终的状态，并不会因为不同的render中存在不同的隔离。

简单来说，你可以将useRef的返回值，想象成为一个全局变量。

我们来改写一下这个Demo再来看看:

```js复制代码
import React, { useState, useEffect, useRef } from 'react';

export default function Demo() {
  const [like, setLike] = useState(0);
  // 初始化likeRef 初始值为0
  const likeRef = useRef(0);

  useEffect(() => {});

  const handleClick = () => {
    setLike(like + 1);
    // 看做是全局变量，直接使用current属性赋值
    likeRef.current = likeRef.current + 1;
  };

  const getLikeValue = () => {
    setTimeout(() => {
      alert(likeRef.current);
    }, 2000);
  };

  return (
    <div>
      <button onClick={handleClick}>+</button>
      <button>{like} 👍</button>
      <button onClick={getLikeValue}>获得like值</button>
    </div>
  );
}
```

按照之前的步骤操作，这个时候我们可以看到alert弹出的值就是最新的值。而并非作用域隔离的值。

当然需要额外注意的是，修改 `useRef` 返回的值并不会引起react进行重新渲染执行函数，demo中的页面渲染不是因为修改Ref的值，而是因为我们在修改 `likeRef.current` 时同时修改了state中的 `setLike` 造成了页面渲染。

useRef的值改变并不会造成页面重新渲染，这一点可以做很多事情。比如可以配合 `useEffect` 查询页面是首次渲染还是更新。

总结来说， `useRef` 返回值的改变并不会造成页面更新。而且 `useRef` 类似于react中的全局变量，并且不存在不同次render中的 `state/props` 的作用域隔离机制。这就是 `useRef` 和 `useState` 这两个hook的

主要区别。

useRef作用二：获取DOM元素

vue3中获取DOM

当然这一点也是比较常用的useRef的用法，对比在vue3中获取DOM节点:

html复制代码

```
<template>
  <div ref="helloRef">Hello</div>
</template>
<script>
import { ref } from 'vue'
export default {
  setup() {
    const helloRef = ref(null)
    return {
      helloRef
    }
  }
}
</script>
```

接下来只需要通过 `helloRef.value` 便可以获得div节点。

react中获取DOM

说到上边你可以已经了解了，useRef还有一种用法就是通过它来获取页面上的DOM元素。

jsx复制代码

```
import React, { useEffect, useRef } from 'react';

export default function Demo() {
  const domRef = useRef<HTMLInputElement>(null);

  useEffect(() => {
    domRef.current?.focus();
    console.log(domRef,'domRef')
  });

  return (
    <div>
      <input ref={domRef} type="text" />
      <button>增加</button>
    </div>
  );
}
```

- 1.通过 `useRef` 创建一个变量进行保存 `(domRef)` 。
- 2.在jsx中通过 `ref={domRef}` 给对应元素节点添加属性。
- 3.在页面挂载后通过 `domRef.current` 就可以获取对应节点的真实DOM元素了。

总结：

对于上边的Demo，我们可以总结出useRef的一些特性。

我们可以将 `useRef` 返回值看作一个组件内部**全局共享变量**，它会在渲染内部共享一个相同的值。相对 `state/props` 他们是独立于不同次render中的内部作用域值。

同时额外需要注意 `useRef` 返回值的改变并不会引起组件重新render，这也是和 `state/props` 不同的地方。

当然我们在 `React.functionComponent` 中想要获取对应 `jsx` 的真实Dom元素时候也可以通过 `useRef` 进行获取到对应的Dom元素。