都React V18了，还不会正确使用React Hooks吗，万字长文解析Hooks的常见问题

杭州下雪了，突然想去西湖看雪了，雪后的杭州更有水墨画里的感觉，一下雪杭州就成了临安。希望这个冬天不要太冷吧，也希望来年春暖花开的时候，春天可以真正来到。

闲言少叙，直接进入正文

前言

今天主要想说一下react hooks，react hooks是react v16.8 之后引入的API，现在react都已经到V18了，hooks怎么还能不会用呢?

首先hooks引入的目的是给函数式组件增加数据状态管理的能力，同时增加代码的可复用能力。但是同时hooks也是一个潘多拉魔盒，因为函数式组件不再只是单纯的一个纯函数了，可以在内部处理副作用了，使用不好就会经常遇到各种各样的问题，而且错误的使用方式也会引起re-render，引起一些性能上的问题

本文主要介绍hooks的常见的几个问题与最优实践，同时介绍一下随着React最新版本的API的变化，首先，在使用之前，笔者还是想强调一下

请配置上`eslint-plugin-react-hooks` 请配置上`eslint-plugin-react-hooks` 请配置上`eslint-plugin-react-hooks`

hooks的的使用确实很爽，但是和纯函数相比，还是有挺多反直觉的写法的，比如不能在判断语句中使用hooks。这就很容易有问题，我们需要使用工具来规避这些问题，来提醒我们有些写法是错误的。当然hooks的有其自己的合理性问题，我们暂时不做讨论，这个插件提醒可以保证让我们的写法是符合当前规范的，不至于出现低级错误

异步调用的闭包问题

先看看这段代码，实现一个统计1秒内按钮点击的次数的功能

```js 复制代码
export default function Demo() {
  const [number, setNumber] = React.useState(0);
  const click = () =>
    setTimeout(() => {
      setNumber(number + 1);
    }, 1000);
  return <button onClick={click}> 点击 {number} 次</button>;
}
```

当多次点击的时候的时候，显示的点击次数是不对的。点击click方法内的闭包回调函数在组件render的时候捕获了number变量，为了解决这个问题可以使用函数方法来更新数据

```js 复制代码
const click = () =>
    setTimeout(() => {
      setNumber(number => number + 1);
    }, 1000);
```

在调用状态更新函数的时候，会将准确的数据回调给当前的更新函数

更近一步

同样的，假如我们统计开屏1秒内的点击次数，在计时结束后，将点击次数发送给server端的时候，就会遇到另一个问题

```js 复制代码
export default function Demo() {
  const [number, setNumber] = React.useState(0);
  useEffect(() => {
    const timer = setTimeout(() => {
      // do fetch
      console.log(number);
    }, 1000);
    return () => {
      clearTimeout(timer);
    };
    // 依赖这里会有个警告
  }, []);

  return <button onClick={() => setNumber((c) => c + 1)}>点击{number}次</button>;
}
```

首先我们会遇到一个 `eslint-plugin-react-hooks` 警告

```
React Hook useEffect has a missing dependency: 'number'. Either include it or remove the dependency
```

而且执行结果也是由于闭包的原因不能正常提交，熟悉useEffect到都知道，这个原因是使用useEffect的时候，依赖需要添加到依赖数组内，这样才能更新数据到useEffect内，但是在这个例子中，添加了依赖后，每次点击都会引起计时器重新执行，引起倒计时失效，就和需求有冲突了，那这种情况下怎么解决呢?

这个问题其实可以简单归累为：**不想让useEffect重新执行的依赖怎么使用的问题**，这个场景下就只能使用useRef来实现了

```js
export default function Demo() {
  const [number, setNumber] = React.useState(0);
  const numberRef = useRef(number);
  numberRef.current = number;
  useEffect(() => {
    const timer = setTimeout(() => {
      // do fetch
      console.log(numberRef.current);
    }, 1000);
    return () => {
      clearTimeout(timer);
    };
  }, []);

  return (
    <button onClick={() => setNumber((c) => c + 1)}>点击{number}次</button>
  );
}
```

通过useRef可以来保证访问到的number一直是最新的，解决了闭包的问题，同时useEffect未直接依赖number，当number变化的时候不会引起重新执行

小结

虽然和hooks的写法有关，本质上还是闭包问题，比如forEach内写的定时器也会有同样的问题；但是对于异步场景下的写法，**不想引起useEffect重新执行的变量，就可以用useRef做一层代理**，这种用法只能说是对react hooks设计的一种妥协

用useCallback肯定能提升性能吗

useCallback的作用是缓存函数，避免重复生成新函数，引起组件重新渲染，比如：

```js
const Children = (props) => <button onClick={props.doFetch}>提交</button>;

export default function Demo() {
  const doFetch = useCallback(() => {
    // fetch();
  }, []);

  return <Children doFetch={doFetch} />;
}
```

这是一个很常见的写法，但是需要明确在class组件的时候我们通常使用`shouldComponentUpdate`来拦截更新，通过比较父组件传入的props的变化，来判断是否re-render子组件，上面的🌰中，`<Children />`只是一个普通组件，只对传入的函数包裹一层useCallback是不能起到优化作用的，需要通过包裹一层`React.memo`才可以，React.memo会对对传入的props进行一次浅比较，避免非必要的更新，如下

```js
const Children = React.memo((props) => <button onClick={props.doFetch}>提交</button>);
```

另外需要补充的是，假如有一些特别复杂的对象属性需要传入的话，可以考虑通过useMemo进行一层包裹，避免一些不必要的re-render。当然，使用useMemo也是有一定的性能成本的，假如只是简单计算的话，直接计算就可以了，可能这样的成本还小一些。

思考题

下面这个场景下useCallback和React.memo组合可以起到优化的作用吗?

```js
const Children = React.memo((props) => <button onClick={props.doSubmit}>提交</button>);

export default function Demo() {
  const [number, setNumber] = useState(0);

  const doSubmit = useCallback(() => {
    console.log(`Number: ${number}`);
  }, [number]); // 把number写在依赖数组里

  return (
```

```
    <>
      <input value={number} onChange={(e) => setNumber(e.target.value)} />
      <Children doSubmit={doSubmit} />
    </>
  );
}
```

结论是不行，原因可以考虑一下useCallback的deps该怎么处理，有没有方法可以避免doSubmit函数重新生成呢

小结

只需要记得**使用useCallback的时候，对应的子组件一定要使用React.memo包裹**，否则使用useCallback就没有任何的意义。当然useCallback的deps要根据实际场景来添加，否则就不会有任何的优化效果。

另外对于useCallback，笔者个人的理解是有些破坏代码的可读性了，使用的时候需要按照场景具体评估一下，是否真的需要无脑使用，平衡好代码可维护性和性能

hooks组件怎么优雅的使用refs

其实这个问题有些伪命题的感觉，当我们使用在父组件中通过ref操作子组件的方法时，已经就是不优雅的了。官方文档这样说：

> 在常规的 React 数据流中，props 是父组件与子组件交互的唯一方式。要修改子元素，你需要用新的 props 去重新渲染子元素

只能说非必要的尽量避免直接使用ref吧，尽量避免打乱react的单向数据流。那回到我们的问题，使用hooks的时候如何使用更优雅的使用ref呢

官方文档提供了以下几个场景刚好适合使用Refs

- 处理focus、文本选择或者媒体播放
- 触发强制动画
- 基层第三方库

useRef

由于函数组件没有实例，在函数组件内不能使用string ref,callback ref, create ref，会有如下报错

> Uncaught Invariant Violation: Function components cannot have refs. Did you mean to use React.forwardRef()?

所以在函数组件内是能使用useRef,由于useRef返回一个对象，这个对象只有current一个值，并且这个值的地址在整个组件的生命周期内不会改变，所以useRef提供了组件生命周期内共享数据的存储，修改了ref.current的值不会触发组件re-render，而且组件re-render的时候，ref.current的值也会保留，类似class组件的静态变量，因此我们可以用useRef做很多事情，比如挂载特定的dom或者子组件的实例、存储全局定时器等等

那useRef怎么使用呢，针对官方问题提供的几个场景，笔者简单给出几个demo

获取特定dom

对于原生元素可以直接使用

```jsx
function Input(props) {
  const inputRef = useRef(null);

  function click() {
    inputRef.current.focus();
  }

  return (
    <div>
      <input type="text" ref={inputRef} />
      <input type="button" value="Focus the text input" onClick={click} />
    </div>
  );
}
```

通过ref.current可以直接操作input

useRef,forwardRef,useImperativeHandle组合用法

首先先说一下forwardRef

> Ref 转发是一项将 ref 自动地通过组件传递到其一子组件的技巧。对于大多数应用中的组件来说，这通常不是必需的。但其对某些组件，尤其是可重用的组件库是很有用的。

简答来说，就是函数式组件要想暴露给父组件ref的时候，需要使用forwardRef才能进行ref的传递。需要明确的是**不能在函数式组件上使用ref属性，因为函数式组件没有实例**，使用的话会直接抛出异常。

然后再说一下useImperativeHandle，函数式组件使用useImperativeHandle 可以通过 forwardRef 暴露给父组件的自定义方法。这可以类比于class组件，我们可以通过class组件获取ref的组件实例，进而调用内部方法，我们也可以通过useImperativeHandle暴露的方法，调用函数式组件的内部方法，直接看 

jsx 复制代码

```jsx
import React, { useRef, useImperativeHandle } from "react";
const Children = React.forwardRef((props, ref) => {
  const inputRef = useRef(null);
  useImperativeHandle(
    ref,
    () => {
      const handleRefs = {
        onFocus() {
          inputRef.current.focus();
        }
      };
      return handleRefs;
    },
    []
  );
  return <input placeholder="请输入" ref={inputRef} />;
});

const Demo = () => {
  const childRef = useRef(null);
  const click = () => {
    const { onFocus } = childRef.current;
    onFocus();
  };
  return (
    <>
      <Children ref={childRef} />
      <button onClick={click}>输入框focused</button>
    </>
  );
};
```

```
export default Demo;
```

useImperativeHandle 接受三个参数：

- forWardRef转发过来的ref
- 自定义处理函数，返回值时暴露给父组件的ref对象
- 依赖项deps，根据实际的依赖传入即可，用于更新ref对象

通过useImperativeHandle外部组件就可以相对优雅的获取到子组件的“实例”，调用子组件的内部方法

小结

写到这里，其实我们也能发现，useRef的使用频率还是挺高的，我们既可以把useRef当成普通的ref的相关API，用来操作子组件，也可以将其作为一个函数式组件内的静态变量来使用，可以说是hooks遇到问题的时候非常万金油的存在了

说回ref,基本对于函数式组件来说，useRef,forwardRef,useImperativeHandle组合用法能覆盖所有需求了，但是还是那句话，能尽量避免使用的就避免使用，让自己的代码有更高的可维护性

如何封装自定义hooks

自定义hooks是基于react hooks api的自定义扩展，可以将一段通用性逻辑封装起来，达到复用的效果；简单来说通过自定义 Hook，可以将组件逻辑提取到可重用的函数中，可以在组件内直接使用

封装的自定义hooks一般内部使用多个react hooks API，用于解决一些复杂且通用的逻辑。接下来看几个🧅，然后可以对比一下自己的场景，也并不是所有能用自定义hooks的地方，就必须要用自定义hooks的，还是要判断一下可复用程度再做决定

提取自定义 hooks

> 当我们想在两个函数之间共享逻辑时，我们会把它提取到第三个函数中。而组件和 Hook 都是函数，所以也同样适用这种方式。

自定义 hooks 是一个函数，其名称以 “use” 开头，函数内部可以调用其他的 Hook。自定义hooks和普通的函数并没有任何不同，但是结合公共hooks API的能力，在react的调度下，我们可以实现很多很有趣的功能，社区有实现的ahooks自定义hooks库，也可以自己参考一下

接下来简单放几个相对有代表性的  ：

useState属性合并

你的代码里是不是写过这样的代码

```js 复制代码
const [duration, setDuration] = useState(0);
const [currentTime, setCurrentTime] = useState(0);
const [volume, setCurrent] = useState(0);
const [speed, setSpeed] = useState(0);
...
```

这段代码是之前封装播放器的一段代码，里面对video的属性进行一定的提取，导致了代码中一排蔚为壮观的useState，这样的写法其实可以优化到一个字段内

```js 复制代码
const [player, setPlayer] = useState({
  duration: 0,
  currentTime: 0,
  volume: 0,
  speed: 0,
});
```

当然这个时候更新装状态需要带上旧的状态

```js 复制代码
setPlayer({...player, duration});
```

那怎么才能再省点代码，让我可以直接修改需要修改的参数，不需要改变的依旧保持现状呢，这个时候就可以封装一个自定义hooks，如下

```js 复制代码
import { useState } from 'react';

const useMergeState = (initialState) => {
  const [state, setState] = useState(initialState);
  const setMergeState = (pickState) => {
    setState(preState => {...preState, ...pickState});
```

```
  }
  return [state, setMergeState]
}
```

这样就可以直接使用了

```js 复制代码
const [player, setPlayer] = useMergeState({
  duration: 0,
  currentTime: 0,
  volume: 0,
  speed: 0,
});

setPlayer({ duration: 2});
```

这就是一个最简单的自定义hooks，当然useMergeState还有可优化的地方，比如参考useState的用法，pickState可能是个函数，需要判断一下入参类型；然后内部的实现setMergeState其实可以通过useCallback进行一下包裹，让函数不用每次都初始化等等

dom相关的自定义hooks

我们经常会监听dom的滚动来做一些自定义的逻辑，比如滚动翻页等等，一个监听滚动的自定义hooks该怎么实现呢，我们需要先确定我们需获取到哪些值，以及我们的入参数是什么

需求分析

入参：毋庸置疑肯定是dom元素的引用，可以使用useRef来实现 返回值：一般都是元素滚动的信息，可以根据自己的实际来决定

代码实现

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

const useScroll = (target) => {
  const [position, setPosition] = useState();

  useEffect(() => {
    function onscroll(e){
      setPos([target.current.scrollLeft, target.current.scrollTop])
    }
```

```
    target.current.addEventListener('scroll', onscroll, false)
    return () => {
      target.current.removeEventListener('scroll', onscroll, false)
    }
  }, [])

  return position
}

export default useScroll
```

上述代码也是有一些可以优化的地方，比如一些参数有效性的判断等，这个  只是一个演示，提供一个参考思路

防抖函数

防抖节流函数大家肯定经常在开发中遇到，面试的估计也没有被少问。一般都是我们避免频繁调用接口，或者在react中避免频繁改动state的时候需要使用的，防抖节流的区别这里也不赘述了，简单来说，其主要区别是防抖是操作后n秒内只能调用一次，假如n秒内又触发了操作，重新倒计时n秒；节流就是操作n秒内只能调用一次，即使是多次触发操作也是n秒内执行一次；节流一般用在输入框搜索，需要在最后一次提交后触发；节流一般用在滚动监听等等。

代码实现

```
js 复制代码
import { useEffect, useMemo } from 'react';
import debounce from 'lodash/debounce';

const useDebounce = (fn, wait) => {
  const wait = wait ?? 1000;

  const debounced = useMemo(
      () =>
        debounce(
          fn(),
          wait
        ),
      [],
  );
  return [debounced.run, debounced.cancel];
}
```

上述代码为了突出自定义hooks的用法，基于lodash/debounce实现了一个节流hooks，其实核心就是简化debounce的用法，当然lodash/debounce还有更多的配置，自己可以基于自己的实际场景进行封装

至于节流函数就作为一个思考题可以自己实现一下

小结

自定义hooks可以维护在项目中的一个单独hooks文件夹内，作为全局公共方法提供，当然也可以发布到自己的npm组件，提供给更多的项目使用。还有很多的有趣的封装方法就不再一一演示了，可以自己多看看第三方包的实现，加深对自定义hooks的理解

React V18新增了哪些hooks API

React V18已经发布了一段时间了，针对hooks也提供了一些新的API。新增的API有 `useId，useInsertionEffect，useTransition，useDeferredValue，useSyncExternalStore` ，让我们一一看看提供了什么新的能力。

Before，先看看React V18带来了什么

可以可选的开启 Concurrent Mode，大幅提高了React的性能，推出了时间切片和任务优先级的概念，让 React 有了更多的可能性，更多的描述可以去看看官方文档，我们关注的点还是这几个新增的hooks，让我们简单的了解一下这几个API，具体的使用方法可以自己写一写demo试一下

useSyncExternalStore

顾名思义，这个API是针对React的第三方状态管理库的。由于V18可以开启concurrent mode，使得render这个过程能够切分成以fiber为最小单位的多次任务，这些任务可能就会存在对外部状态的修改。假如在render的多个任务分片不同阶段，部分外部状态被修改了，那就会造成多次渲染结果不一致的问题，也就是**tearing 问题**，tearing 通俗地讲，就是外部状态不受 react 异步调度控制，非常容易错乱。useSyncExternalStore就是防止tearing的出现，通过订阅的方式保持最终渲染结果的一致性。

最后简单看看API

```js 复制代码
useSyncExternalStore(
    subscribe, // 接收一个subscribe函数，该函数接收一个函数用于通知react外部状态发生变更
    getSnapshot, // 返回外部状态的最新快照值
)
```

useTransition

transition 特性也是依赖 concurrent mode的，可以用来降低渲染优先级。当我们用startTransition包裹计算量大的函数的时候，就相当于通知了React当前计算的优先级很低，可以降低更新的优先级，减少重复渲染次数，可以当成一个防抖函数来处理（不完全相同）。

```js 复制代码
const Demo = () => {
  const [isPending, startTransition] = useTransition();
  return (
      <input
        type="text"
        onChange={(event) => {
          startTransition(() => {console.log(event.target.value);
        }}}
      />
  );
}
```

可以看看官方的这个demo

useDeferredValue

和useTransition类似，useDeferredValue的特性也是依赖 concurrent mode 的，可以直接翻译为使用一个推迟的值。 useDeferredValue 可以让我们延迟渲染优先级不高的部分，等高优先级的渲染任务结束后，延迟的渲染才开始，并且可中断不会阻塞用户输入。

```js 复制代码
const Demo = (value) => {
  // React 将会在合适的时间生成deferredValue
  const deferredValue = useDeferredValue(value, { timeoutMs: 1000 });
  return (
    <div>
      {
        deferredValue
      }
    </div>
```

```
  );
}
```

useInsertionEffect

类似useLayoutEffect，区别就是不能访问dom

useInsertionEffect的出现就是为了解决cssinjs问题的，用于插入样式规则，而且一般是插入全局DOM节点，比如 `<style/>` 等。这个hooks为了避免在concurrent mode下，客户端生成或编辑 `<style/>` 时引起性能问题，可以在这个hooks中操作插入一些css节点

```js 复制代码
function useCSS(rule) {
  useInsertionEffect(() => {
    document.head.appendChild(rule);
  });
  return rule;
}
const Demo = () => {
  let className = useCSS(rule);
  return <div className={className} />;
}
```

useID

一般用于服务端渲染，生成一个ID，但是不要把这个ID作为数组循环渲染的key，渲染的唯一key一般是从你的数据中生成的, 就不再详细介绍了