课件12-05 高阶组件、 react hooks、异步组件

高阶组件(HOC，High Order Component)

定义：把组件作为参数，并返回（高阶）组件的函数，称为高阶组件。

- 代码复用，状态/逻辑抽象
- 可以对 state/event/props 进行劫持、操作

假如有这样的场景，两个查询列表的页面结构相同，查询条件相同，只是表头包括操作列不一样：

显然这两个页面具有很高的复用性，不只是 UI 级别的复用，逻辑都几乎一致，这时候，高阶组件就派上用场了。我们定义查询条件的部分为组件 SearchPanel，表格组件为 Table（antd design 的 Table 自带底部分页区），那么这两个页面可能是下面的代码结构：

页面 1，可能是普通用户查看页

```
import React, { Component } from 'react';
import request from 'axios';
import { Button, Table } from 'antd';
import SearchPanel from './SearchPanel';

export default class Page1 extends Component {
    state = {
        query: {
            name: '',
            id: '',
            time: '',
            valid: ''
        },
        dataSource: []
```

```
    }
    columns = [
        {dataIndex: 'label', title: '标签'},
        {dataIndex: 'action', title: '操作',
            render: (_, record) => {
                const onOpen = () => window.open(`/xxx/${record.id}`);
                return <Button onClick={onOpen}>查看</Button>;
            }
     }]

     onChange = (params) => {
        this.setState(query => ({ ...query, ...params }));
        request('/api/list', {
            method: 'GET',
            params
        }).then(res => { // 这里暂不考虑异常
            this.setState({ dataSource: res.data });
        });
    }

    componentDidMount()  {
        this.onChange(this.state.query);
    }

    render() {
        const { query, dataSource } = this.state;
         return (
            <>
                <SearchPanel value={query} onChange={this.onChange} />
                <Table columns={this.columns} dataSource={dataSource} />
            </>
        );
    }
}

```

页面 2，可能是管理员页面

```
import React, { Component } from 'react';
import request from 'axios';
import { Button, Table } from 'antd';
import SearchPanel from './SearchPanel';
```

```
5
6  export default class Page2 extends Component {
7      state = {
8          query: {
9              name: '',
10             id: '',
11             time: '',
12             valid: ''
13         },
14         dataSource: []
15     }
16
17     onEdit = id => {}
18     onDelete = id => {}
19
20     columns = [
21            {dataIndex: 'name', title: '名称'},
22            {dataIndex: 'action', title: '操作',
23            render: (_, record) => {
24               return <>
25                 <Button onClick={() => this.onEdit(record.id)}>编辑</Button>
26                 <Button onClick={() => this.onDelete(record.id)}>删除</Button>
27               </>;
28            }
29      }]
30
31      onChange = (params) => {
32         this.setState(query => ({ ...query, ...params }));
33         request('/api/list/admin', {
34             method: 'GET',
35             params
36         }).then(res => { // 这里暂不考虑异常
37             this.setState({ dataSource: res.data });
38         });
39     }
40
41      componentDidMount()  {
42         this.onChange(this.state.query);
43      }
44
45     render() {
```

```
        const { query, dataSource } = this.state;
        return (
            <>
                <SearchPanel value={query} onChange={this.onChange} />
                <Table columns={this.columns} dataSource={dataSource} />
            </>
        );
    }
}

```

可以看到，两份代码除了表格列及其操作外，请求数据的接口也分别为 **/api/list** 和 **/api/list/admin**，这两份文件总计约 100 行代码。我们使用高阶组件整合相同的逻辑：

Page1 和 Page2 的公共 UI 部分：

```
import React from 'react';
import { Table } from 'antd';
import SearchPanel from './SearchPanel';

// 无状态组件，所以用函数实现更简洁
export default function PageCommon({ query, dataSource, onChange, columns }) {
    return (
        <>
            <SearchPanel value={query} onChange={onChange} />
            <Table columns={columns} dataSource={dataSource} />
        </>
    );
}

```

高阶组件：

```
import React, { Component } from 'react';
import request from 'axios';

export default function hoc(WrappedComponent, api) {
    return class extends Component {
        state = {
            query: {
                name: '',
                id: '',
                time: '',
```

```
            valid: ''
        },
        dataSource: []
    }

    onChange = (params) => {
        this.setState(query => ({ ...query, ...params }));
        request(api, {
            method: 'GET',
            params
        }).then(res => { // 这里暂不考虑异常
            this.setState({ dataSource: res.data });
        });
    }

    componentDidMount()  {
        this.onChange(this.state.query);
    }

    render() {
        retrun <WrappedComponent
            {...this.props}
            {...this.state}
            onChange={this.onChange}
        />;
    }
}
}
```

最终分别得到两个页面，Page1：

```
class Page1 extends Component {
    columns = [
        {dataIndex: 'label', title: '标签'},
        {dataIndex: 'action', title: '操作',
        render: (_, record) => {
            const onOpen = () => window.open(`/xxx/${record.id}`);
            return <Button onClick={onOpen}>查看</Button>;
        }
    }]
    render() {
        return <PageCommon {...this.props} columns={this.columns} />;
```

```
    }
}

export default hoc(Page1, '/api/list');

```

Page2：

```
class Page2 extends Component {
    onEdit = id => {}
    onDelete = id => {}
    columns = [
           {dataIndex: 'name', title: '名称'},
           {dataIndex: 'action', title: '操作',
           render: (_, record) => {
              return <>
                <Button onClick={() => this.onEdit(record.id)}>编辑</Button>
                <Button onClick={() => this.onDelete(record.id)}>删除</Button>
                </>;
           }
     }]
    render() {
         return <PageCommon {...this.props} columns={this.columns} />;
    }
}
export default hoc(Page2, '/api/list/admin');

```

累计 80 行左右代码，如果再来几个相似的页面，代码量的增加也只限定在特有的业务逻辑上，重点是，我们不需要再重复维护相似的多份逻辑了。从上面的例子可以看到高阶组件具有的能力：

- 向被修饰的组件注入额外的状态或方法，返回值依然是个组件（react 中，返回类组件或函数式组件均可）。
- 自定义注入内容可以实现组件功能的增强。

当高阶组件只有一个组件作为参数时，可以嵌套使用，例如一个基础组件为：

```
function Base({ list }) {
    return <ul>
    { list.map(({ id, name }) => <li key={id}>{ name }</li>) }
    </ul>
}

```

我想在 Base 组件渲染更新的时候记录日志，便于调试：

```
const insertLog = WrappedComponent => class extends Component {
```

```
    componentDidUpdate(...args) {
        console.log(...args);
    }
    render() {
         return <WrappedComponent {...this.props} />
    }
}

const BaseWithLog = insertLog(Base);

```

我想在 Base 组件渲染报错的时候不要白屏，显示一个固定的信息并上报问题：

```
const insertErrorBoundary = WrappedComponent => class extends Component {
    state = {
        error: false
    }
    componentDidCatch(error, errorInfo) {
        this.setState({ error: true }, () => {
            logErrorToMyService(error, errorInfo);  // 调用已有的接口
        });
    }
    render() {
        if (this.state.error) {
            return <p>页面不可用，请检测组件{Component.displayName}的逻辑</p>;
        }
         return <WrappedComponent {...this.props} />;
    }
}

const BaseWithErrorBoundary = insertErrorBoundary(Base);

```

我两个功能都要：

```
// 因为这两个高阶组件的参数都是唯一的，所以可以不区分顺序地组合
const BaseWithLogAndErrorBoundary = insertErrorBoundary(insertLog(Base));
// 我们希望捕获错误的范围尽量大一些，所以把 insertErrorBoundary
// 放在最外面，最后一步再修饰组件。

```

高阶组件的优点想必大家已经能自己总结了，但事物往往具有两面性，下面我们通过实例讲解高阶组件存在或易引发的问题：

```
// 这是一个能够正常运行的组件，没有任何逻辑问题，
// 父组件通过"点击聚焦"按钮可以使子组件内的输入框聚焦
import React, { Component, createRef } from 'react';

class Sub extends Component {
    input = createRef()
    focus = () => { // focus 方法执行时会让 input 元素聚焦。
        this.input.current.focus();
    }
    render() {
        return <input {...this.props} ref={this.input} />;
    }
}

class Parent extends Component {
    state = {
        value: ''
    }
    input = createRef() // 引用子组件实例，便于调用实例上的方法
    onFocus = () => {
      this.input.current.focus(); // 调用子组件实例上的方法
    }
    onChange = e => {
        this.setState({ value: e.target.value });
    }
    render() {
        return <>
            <Sub
              onChange={this.onChange}
              value={this.state.value}
              ref={this.input}
            />
            <button onClick={this.onFocus}>点击聚焦</button>
        </>;
    }
}
```

由于需求变更，我想对 Sub 组价进行增强，比如用上述 insertLog 进行日志输出：

```
...Sub 定义略
import insertLog from './insertLog';

```

```
class Parent extends Component {
    ...
    render() {
        const SubWithLog = insertLog(Sub);
        return <>
            <SubWithLog
              onChange={this.onChange}
              value={this.state.value}
              ref={this.input}
            />
            <button onClick={this.onFocus}>点击聚焦</button>
        </>;
    }
}

```

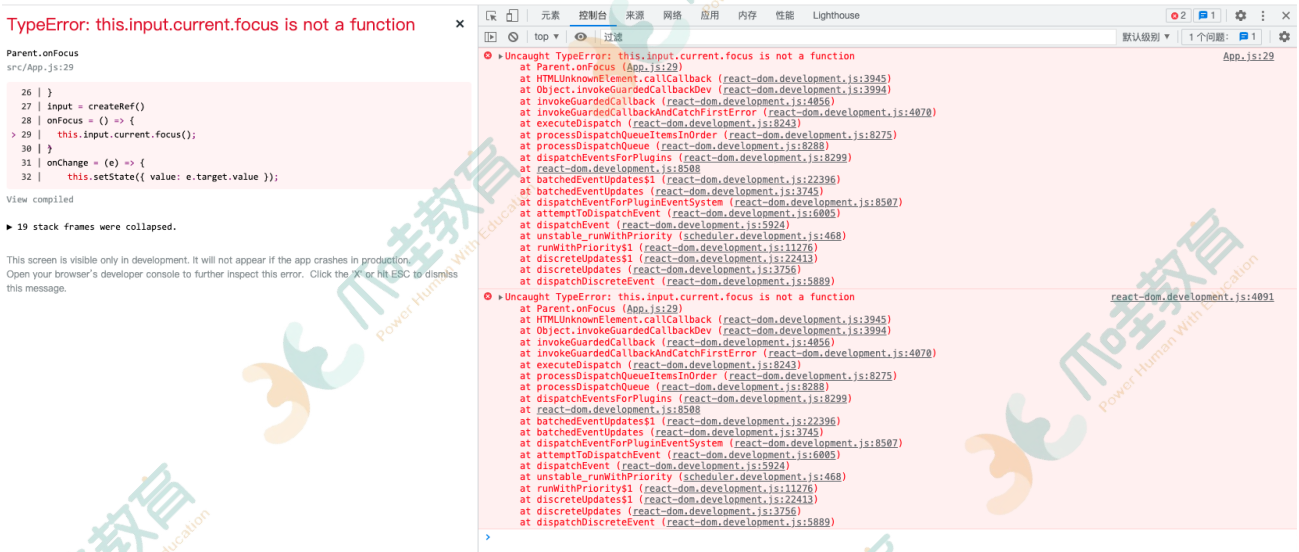
输入文本看看出现了什么问题？问题就是一旦输入一个字符，输入框就失去焦点了。高阶组件在 render 函数内频繁调用，意味着 SubWithLog 始终是新返回的组件，我们将失去 Sub 组件的状态！

fix tips：

```
const SubWithLog = insertLog(Sub);
// 放到 render 以外，可以使组件外，也可以是Parent组件的实例上
class Parent extends Component {
    render() {
        return <>
            <SubWithLog
              onChange={this.onChange}
              value={this.state.value}
              ref={this.input}
            />
            <button onClick={this.onFocus}>点击聚焦</button>
        </>
    }
}
// 或者
class Parent extends Component {
    SubWithLog = insertLog(Sub)
    render() {
        const SubWithLog = this.SubWithLog;
        return <>
            <SubWithLog
              onChange={this.onChange}
```

```
            value={this.state.value}
            ref={this.input}
        />
        <button onClick={this.onFocus}>点击聚焦</button>
    </>
  }
}
```

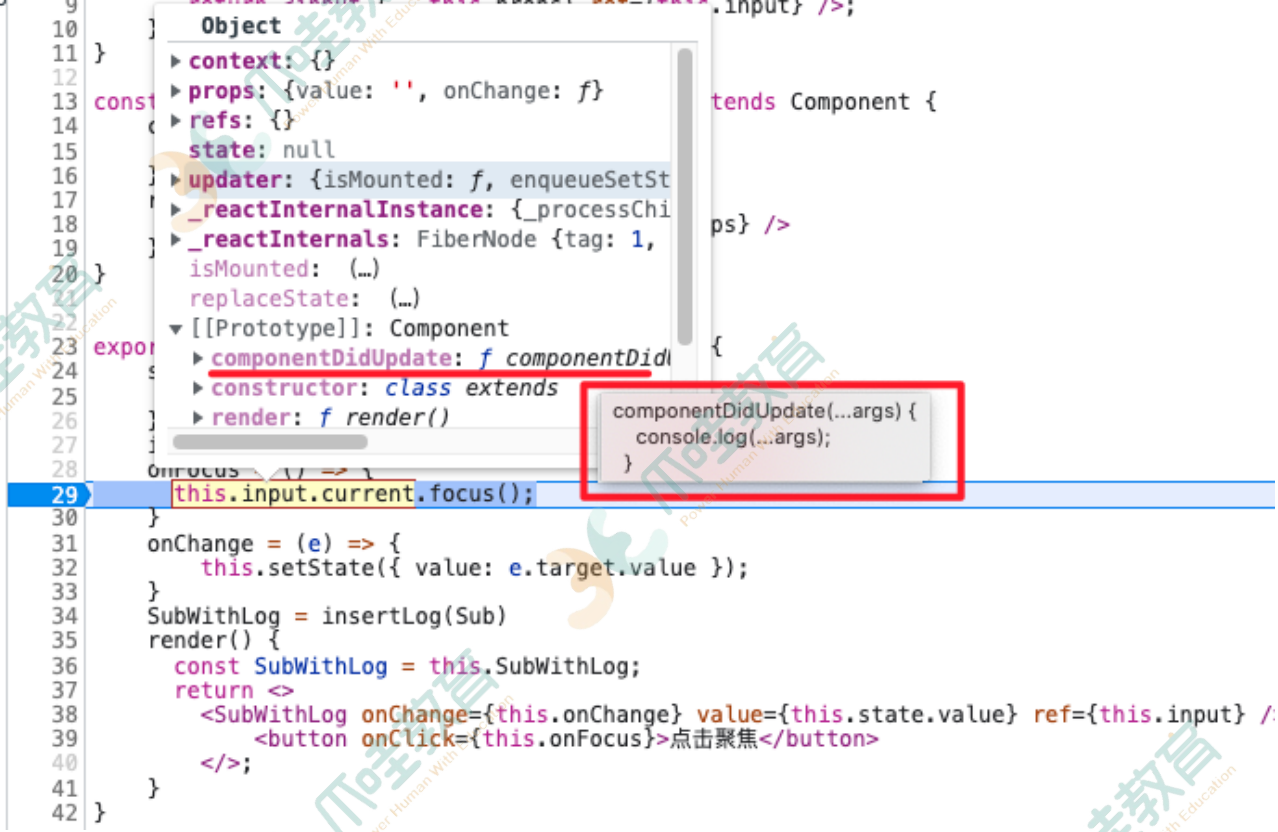
然后点击“点击聚焦”按钮，报错了：



仅仅是加了个高阶组件就报错了！看这里的注释：

```
13
14 class Parent extends Component {
15     state = {
16        value: ''
17     }
18     input = createRef() // 引用子组件实例，便于调用实例上的方法
19     onFocus = () => {
20       this.input.current.focus(); // 调用子组件实例上的方法
21     }
22     onChange = e => {
23         this.setState({ value: e.target.value });
24     }
25     render() {
26        return <>
27            <Sub onChange={this.onChange} value={this.state.value} ref={this.input} />
28            <button onClick={this.onFocus}>点击聚焦</button>
29        </>;
30     }
31 }
```

当使用 SubWithLog 时，实例已经不是 Sub 的了，那是什么呢？断点：



破案了，Parent 内，this.input.current 引用的实例不是 Sub 的，而是 insertLog 返回的那个匿名 class 的实例。根本原因是，react pops 中，key 和 ref 是两个特殊的 property，不会被转发到下层组件，一旦高阶组件没有处理好 ref 的转发，被修饰的组件就会失去与上层组件的联系。

fix tips:

```
1  import { forwardRef } from 'react';
2
3  const insertLog = WrappedComponent => {
4      class Log extends Component {
5          componentDidUpdate(...args) {
6              console.log(...args);
7          }
8          render() {
9              const { forwardedRef, ...props } = this.props
10             return <WrappedComponent {...props} ref={forwardedRef} />
11         }
12     }
13     return forwardRef((props, ref) => <Log {...props} forwardedRef={ref} />);
14 }
```

其原理是，函数式组件除了 props 参数外，还支持第二个参数 ref（如果有传递 ref 的话），我们将 ref 命名为 forwardedRef ——即当做一个 props 继续往后传递（13行），并且在第 10 行再次转换为 ref 挂载到目标组件上。最终，聚焦事件正常工作了。

HOC 缺点小结：

- 增加了组件嵌套层级，过多时对于渲染的性能有一定影响；
- ref 、displayName 等易被忽略，虽然我们不推荐使用 ref，但是类似上面的需求，尤其是组件库封装，ref 的转发是必不可少的，在一些 Dev tool 中也徒增了 UI 无关的组件嵌套；

- 对于已使用了 HOC 的业务，需求的扩展有一定的难度
- 高阶组件有相似的逻辑时，也会造成执行顺序、功能覆盖的风险.....

HOC 知名的应用案例：

- react-redux connect

```
connect(mapStateToProps, mapDispatchToProps, mergeProps)(App);

// 简化实现等价于:
export function connect(mapStateToProps, mapDispatchToProps) {
  return function (WrappedComponent) {
    class Connect extends React.Component {
      componentDidMount() {
        //从context获取store并订阅更新
        this.context.store.subscribe(this.forceUpdate.bind(this));
      }
      render() {
        return (<WrappedComponent
          // 传入该组件的 props,需要由 connect 这个高阶组件原样传回原组件
          { ...this.props }
          // 根据 mapStateToProps 把 state 挂到 this.props 上
          { ...mapStateToProps(this.context.store.getState()) }
          // 根据 mapDispatchToProps 把 dispatch(action) 挂到 this.props 上
          { ...mapDispatchToProps(this.context.store.dispatch) }
          />
        )
      }
    }
    // 接收 context 的固定写法
    Connect.contextTypes = {
      store: PropTypes.object
    }
    return Connect;
  }
}
// 因此， App 组件的 props 会被注入 action、state 等
```

- react-router-dom withRouter

```
export default withRouter(App); // App 获得了 history, location 等 props

```

```
function withRouter(Component) {
  const displayName = `withRouter(${Component.displayName || Component.name})`;
  const C = props => {
    // 如果想要设置被 withRouter 包裹的组件的 ref，这里使用 wrappedComponentRef
    const { wrappedComponentRef, ...remainingProps } = props;

    return (
      <RouterContext.Consumer>
        {context => {
          // 将 context 加入到 Component 中，注意 ref 的转发，这里注入了
          // RouterContext 中定义的各种 props，其中就包括 history，location 对象
          return (
            <Component
              {...remainingProps}
              {...context}
              ref={wrappedComponentRef}
            />
          );
        }}
      </RouterContext.Consumer>
    );
  };

  C.displayName = displayName;
  C.WrappedComponent = Component;

  // 当你给一个组件添加一个 HOC 时，原来的组件会被一个 container 的组件包裹。
  // 这意味着新的组件不会有原来组件任何静态方法。
  // 为了解决这个问题，可以在 return container 之前将 static 方
  // 法 copy 到 container 上面
  // 用 hoist-non-react-statics 来自动复制所有 non-React 的 static methods
  return hoistStatics(C, Component);
}

```

hooks（重点！！！）

高阶组件允许我们通过套娃的方式来增强组件，套娃套多了，维护起来会越来越难。hooks 的诞生也顺带解决了这个问题。因此 hooks 的强大能力依然是代码逻辑的复用，同时也简化了生命周期，使得函数式组件拥有了状态。注意， hooks 只能在函数式组件中使用，命名规范为 use 开头，且可以返回组件或任意类型的数据（也可不返回）。

接上文，使用 hooks 实现组件的增强：

- 为 Base 添加更新时的日志打印功能

```
import { useEffect } from 'react';

const useLog = (props) => {
    useEffect(() => {
        console.log(props);
    });
}

function Base(props) {
    useLog(props);
    return <ul>
    { props.list.map(({ id, name }) => <li key={id}>{ name }</li>) }
    </ul>
}

```

- 错误上报

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

const useErrorBoundary = (reactNode) => {
    const [ErrorBoundary] = useState(() =>
      class ErrorBoundary extends React.Component {
        state = {
            error: false
        }
        componentDidCatch(error, errorInfo) {
            this.setState({ error: true }, () => {
               logErrorToMyService(error, errorInfo);  // 调用已有的接口
            });
        }
        render() {
            return this.state.error ?  <p>
              页面不可用，请检测组件{Component.displayName}的逻辑
            </p> : this.props.children;
        }
    });

    return <ErrorBoundary>{ reactNode }</ErrorBoundary>;
```

```
}
function Base(props) {
    useLog(props); // 没有返回值

    // 有返回值
    const realReactNode = useErrorBoundary(<ul>
        { props.list.map(({ id, name }) => <li key={id}>{ name }</li>) }
    </ul>);
    return realReactNode;
}
```

如果涉及到 ref 转发：

```
import React, { createRef, farwardRef } from 'react';

const Sub = farwardRef((props, ref) => {
    ...
    return <input {...props} ref={ref} />;
});

class Parent extends React.Component {
    ...
    input = createRef()
    render() {
        return <Sub ref={this.input} {/* value | onChange 略 */} />;
    }
}

```

可见，这里没有增加组件嵌套层级，ref 可以顺利地分发到想要的地方。是不是比高阶组件更加好用！接下来全面进入的 hooks 学习。

注：红色必须掌握，蓝色建议掌握，黑色看懂即可。

- useState
- useEffect
- useRef
- useCallback
- useMemo
- useContext
- useImperativeHandle （与 forwardRef 一起使用）
- useReducer
- useLayoutEffect
- useDebugValue
- useTransition

> Hook 对于 Redux connect() 和 React Router 等流行的 API 来说，意味着什么?
> 你可以继续使用之前使用的 API；它们仍会继续有效。
> React Redux 从 v7.1.0 开始支持 Hook API 并暴露了 useDispatch 和 useSelector 等 hook。
> React Router 从 v5.1 开始支持 hook。useHistory，useLocation， useParams
> 其它第三库也将即将支持 hook。

useState

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

export default function UseState() {
  const [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(c => c + 1)}>加一 {count}</button>
  );
}

```

useEffect

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

export default function UseEffect() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log(`mount + update:  ${count}`); // 只要本组件有更新，就会执行
  });

  useEffect(() => {
    console.log(`mount:  ${count}`); // 只在本组件第一次加载才会执行
  }, []);

  useEffect(() => {
    console.log(`mount + update count: ${count}`); // 只要 count 发生变化，就执行
  }, [count]);

  return (
    <button onClick={() => setCount(count + 1)}>count 发生变化{ count }</button>
  );
}

```

useRef

```jsx
// 1. 挂载 dom 节点
import React, { useRef, useEffect } from 'react';

export default function UseRef() {
  const container = useRef(null);
  console.log('container', container); // 第一次是拿不到的

  useEffect(() => {
    console.log('container', container); // current 属性引用着虚拟 DOM 节点
  }, []);

  return (<button ref={container}>Ref 容器</button>);
}

// 2. 模拟类组件的 this，充当持久化数据对象
export default function UseRef() {
  const container = useRef(false);

  useEffect(() => {
    if (container.current) {
      console.log('模拟 componentDidUpdate ，即除了初始化，之后的更新进到这里');
    } else {
      container.current = true; // 初次挂载时走这里
    }
  });

  return (<button>Ref 容器</button>);
}

```

useCallback

```jsx
import React, { useState } from 'react';

const UseCallbackSub = ({ value, onChange }) => {
  console.log('子元素发生了渲染 value: ', value);
  return <input onChange={onChange} value={value} type="number" />;
};

export default function UseCallback() {
```

```
  const [count, setCount] = useState(0);
  const [value, setValue] = useState(0);
  // 每次修改 count 时，本组件发生渲染无可厚非，
  // 但是子组件 UseCallbackSub 也会进行不必要的渲染
  const onClick = () => {
    setCount(count + 1);
  };

  const onChange = e => {
    setValue(e.target.value);
  };

  return (<>
    <button onClick={onClick}>count 发生变化{ count }</button>
    <UseCallbackSub onChange={onChange} value={value} />
  </>);
}

// 通常优化这类情景，可以对子组件使用 memo 包裹

import React, { useState, useCallback, memo } from 'react';

const UseCallbackSub = memo(({ value, onChange }) => {
  console.log('子元素发生了渲染 value: ', value);
  return <input onChange={onChange} value={value} type="number" />;
});

// 父组件内:
...
  const onChange = useCallback(e => {
    setValue(e.target.value);
  }, []);
...
// 总结一句，useCallback 可以对函数进行缓存，保证 onChange 不
// 会随着组件更新而改变引用，而 memo 会默认对所有的 props 进行对比，如果不
// 发生变化则不更新组件，避免父级引起的子级渲染。当然，上述方式也可以不使用
// useCallback 达到目的(组件的更新只取决于 value 的变化)，使用自定义比对函数:

const UseCallbackSub = memo(({ value, onChange }) => {
  console.log('子元素发生了渲染 value: ', value);
  return <input onChange={onChange} value={value} type="number" />;
}, (prev, next) => prev.value === next.value);
```

useMemo

```jsx
import React, { useState, useMemo } from 'react';

export default function UseMemo() {
  const [count, setCount] = useState(0);
  const [value, setValue] = useState(0);

  // 总数 total 依赖于两个值，其中一个变化都会重新计算
  const total = useMemo(() => +count + +value, [count, value]);

  const onChange = e => {
    setValue(e.target.value);
  };

  const onClick = () => {
    setCount(count => count + 1);
  };

  return (<>
    <button onClick={onClick}>count 发生变化{ count }</button>
    <input onChange={onChange} type="number" value={value} />
    <span>总数: {total}</span>
  </>);
}
// useMemo 可以做一些影响性能的计算，避免无关因素引起的频繁运算，
// 等价于 Vue 的 computed
```

useContext

```jsx
// 提供跨组件传递信息的能力，搭配 createContext 使用
import React, { useState, createContext, useContext } from 'react';

// 1. 创建共享数据源对象
const Context = createContext();

// 父组件使用 Provider 包裹所有的后代组件
export default function Parent() {
  const [count, changeCount] = useState(0);

  const store = {
```

```
    count, changeCount,
  };

  return ( // 2. 数据源注入到根组件
    <Context.Provider value={store}>
      <button onClick={() => changeCount(count + 1)}>加一 {count}</button>
      <Sub1 />
    </Context.Provider>
  );
}

// 子组件使用 useContext 调用方法，这里并没有传递 props
function Sub1() {
  const ctx = useContext(Context);
  return <>
    <button onClick={() => ctx.changeCount(c => c + 1)}>
      Sub1 能通过 Context 访问数据源 { ctx.count }
    </button>
    <Sub2 />
  </>;
}

// 后代组件使用 useContext 取数据，这里并没有传递 props
function Sub2() {
  const ctx = useContext(Context);
  return <span>后代组件 Sub2 拿到的 Parent 数据: { ctx.count }</span>;
}
```

useImperativeHandle +forwardRef

还记得上面使用 ref 转发的案例吗？我们拿下来：

```
import React, { createRef, farwardRef } from 'react';

const Sub = farwardRef((props, ref) => {
    return <input {...props} ref={ref} />;
});

class Parent extends React.Component {
    input = createRef()
    onFocus = () => {
       this.input.current.focus();
    }
```

```
    render() {
        return <>
            <Sub ref={this.input} {/* value | onChange 略 */} />
            <button onClick={this.onFocus}>点击聚焦</button>
        </>;
    }
}

```

在这种实现中，我们通常将子组件实例(如果是类组件的话)或节点(这里是 input 标签)统一挂载到了父组件这意味着第 10 行 this.input.current 将能够访问到实例/节点的全部方法，如 input 的 onblur，其他未受控的属性也可以被随意更改，这是比较危险的行为，因为打破了实例的封闭原则。我们可以控制子组件向外暴露的方法：

```
import React, { createRef, useRef, farwardRef } from 'react';

const Sub = farwardRef((props, ref) => {
    const input = useRef();
    // 这里有意切断父级 ref 与 input 标签的联系，只让组件本身拿到 input 标签的引用

    useImperativeHandle(ref, () => ({
         onFocus() {
            input.current.focus();
         }
    }));
    return <input {...props} ref={input} />;
});

class Parent extends React.Component {
    input = createRef()
    // 这里引用的不再是子组件中的 input，而是 useImperativeHandle 第二个
    // 参数的返回值对象
    onFocus = () => {
        this.input.current.onFocus();
    }
    render() {
        return <>
            <Sub ref={this.input} {/* value | onChange 略 */} />
            <button onClick={this.onFocus}>点击聚焦</button>
        </>;
    }
}
```

```
// 这样父组件将没有权限访问到 onFocus 之外的方法和属性
```

自定义 hook 的实现

首次挂载不执行，更新时执行：

```
// 上文有提到使用 useEffect 模拟 componentDidUpdate 的例子，如果
// 这个场景比较多，我们可以封装起来：
import React, { useRef, useEffect } from 'react';

export default function useUpdated(callback) {
  const didUpdate = useRef(false);
  useEffect(() => {
    if (didUpdate.current) {
      callback?.();
    } else {
      didUpdate.current = true; // 初次挂载时走这里
    }
  });
}

// 分隔线 ----------------------------------------------------
// 使用方式
import useUpdated from './useUpdated';

export default function UseRef() {
  useUpdated(() => {
    console.log('模拟 componentDidUpdate , 即除了初始化，之后的更新进到这里');
  });
  return (<button>Ref 容器</button>);
}
```

参数变化就发起请求，自动更新数据源

```
import { useState, useEffect } from 'react';
const defaultOptions = {}; // 根据实际情况写死一些默认值

export default function useRequest(query, { url, method = 'GET' } = {}) {
  // 务必保证 query 的变化时有条件的
  const [state, setState] = useState({
    data: [], error: false, loading: false
  });
  useEffect(() => {
```

```
    const opts = { ...defaultOptions, method };
    if (method === 'GET') {
        opts.params = query;
    } else {
        opts.body = JSON.stringify(query);
    }
    setState(state => ({...state, loading: true}));
    fetch(url, opts).then(json => json()).then(res => { // 异常处理自己做一下
       setState(state => ({
           ...state,
           loading: false,
           data: res.data || [],
           error: false
       }));
    }).catch(() => {
       setState(state => ({...state, loading: false, error: true }));
    });
  }, [query]);

  return state;
}

// 分隔线 ----------------------------------------------------------
// 使用方式
import useRequest from './useRequest';
export default function List() {
  const [query, setQuery] = useState({});
  const { data, error, loading } = useRequest(query, { url: '/list' });

  const onChangeQuery = params => setQuery(query => ({ ...query, ...params }));

  return <div>
    <SearchPanel onChange={onChangeQuery} />
    <ul>
      {
        loading ? 'loading...'
          : error ? <Empty description="出错了" />
            : data.map(item => <li key={item.id}>{item.name}</li>)
      }
    </ul>
    <Pagination onChange={onChangeQuery} />
```

```
  </div>;
}
```

一些官方实现的 hooks

- useSelector, useDispatch

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';

export default function App() {
    const count = useSelector(state => state.count);
    const dispatch = useDispatch();
    const incrementCount = () => {
        dispatch({
            type: 'module1/CHANGE_COUNT',
            payload: {
              count: count + 1
            }
        });
    };
    return <button onClick={incrementCount}>{count}</button>;
}
```

- useLocation, useHistory, useParams

```
import React from 'react';
import { useLocation, useHistory, useParams } from 'react-router-dom';

export default function App() {
    const { query } = useLocation(); // 地址栏的查询参数(? 后的部分)
    const history = useHistory();
    const params = useParams(); // 拼接到地址栏的动态路由参数
    const onSkip = () => {
        history.push(`/about/${query.id}`);
    };
    return <button onClick={onSkip}>下一步</button>;
}
```

如果你使用 umi，你会发现这几个 hooks 都可以直接从 umi 中导入

异步组件

动态导入 + Suspense 占位。下面的 About 组件将独立打包为一个文件，访问的那一刻开始下载，在此之前并不会占用网络和系统资源。对于低优先级的任务，尤其是单页应用的首屏展示，异步组件显得十分必

要。

```
import { Suspense, lazy } from 'react';
const About = lazy(() => import('./About'));

export default function App() {

    return <Suspense fallback={<h1>Loading profile...</h1>}>
      <About />
    </Suspense>;
}
```

如果我们自己实现一个异步组件：

```
import React from 'react';

export default function lazy(loadComponent) {
  const Fallback = () => <h1>loading...</h1>;
  const [Component, setComponent] = useState(() => Fallback);

  useEffect(() => {
    loadComponent().then(res => {
      setComponent(res.default);
    });
  }, []);

  return <Component />;
}
// 或者使用高阶函数
export default function lazy(loadComponent) {
  return class WrapComponent extends React.Component {
    state = {
      Component: () => <h1>loading...</h1>
    }
    async componentDidMount() {
      const { default: Component } = await loadComponent();
      this.setState({ Component });
    }
    render() {
      const Component = this.state.Component;
      return <Component />;
    }
```

```
  }
}

// 分隔线 ----------------------------------------------------
// 使用方式
const AsyncAbout = lazy(() => import('./About'));
...
```