React Hook 系列(一)：彻底搞懂react-hooks 用法（万字慎点）

> 链接：https://segmentfault.com/a/1190000021261588?isPrivate=1

> 用心阅读，跟随codesandbox demo或运行源码，你将熟悉react各种组件的优缺点及用法，**彻底熟悉react hook的用法**，收益应该不小😀😀😀

大纲：

- react 不同组件用法。
- react hook 相比以前带来什么。
- react hook 的用法。

React 组件的发展

1. 功能（无状态）组件

> Functional (Stateless) Component，功能组件也叫无状态组件，一般只负责渲染。

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

2. 类（有状态）组件

> Class (Stateful) Component，类组件也是有状态组件，也可以叫容器组件。一般有交互逻辑和业务逻辑。

```
class Welcome extends React.Component {
    state = {
        name: 'tori',
    }
  componentDidMount() {
        fetch(…);
        …
    }
  render() {
    return (
        <>
            <h1>Hello, {this.state.name}</h1>
            <button onClick={() => this.setState({name: 'OO7'})}>改名</button>
        </>
      );
  }
}
```

3. 渲染组件

> Presentational Component，和功能（无状态）组件类似。

```
const Hello = (props) => {
  return (
    <div>
      <h1>Hello! {props.name}</h1>
    </div>
  )
}
```

📣 **总结:**

- 函数组件一定是无状态组件，展示型组件一般是无状态组件；
- 类组件既可以是有状态组件，又可以是无状态组件；
- 容器型组件一般是有状态组件。
- 划分的原则概括为： **分而治之、高内聚、低耦合**；

> 通过以上组件之间的组合能实现绝大部分需求。

4. 高阶组件

> Higher order components (HOC)

HOC 主要是抽离状态，将重复的受控组件的逻辑抽离到高阶组件中，以新的props传给受控组件中，高阶组件中可以操作props传入受控组件。 开源库中常见的高阶组件：Redux的connect， react-router的withRouter等等。

```
Class HocFactory extends React.Component {
    constructor(props) {
        super(props)
    }
    // 操作props
    …
    render() {
        const newProps = {…};
        return (Component) => <Component {…newProps} />;
    }
}
Const Authorized = (Component) => (permission) => {
    return Class Authorized extends React.Component {
        …
        render() {
            const isAuth = ‘’;
            return isAuth ? <Component /> : <NoMatch />;
        }
    }
}

// 项目中涉及到的高阶组件
// 主要作用是将所有action通过高阶组件代理到component的Pro上。
import { bindActionCreators } from ‘redux’;
import { connect } from ‘react-redux';
// 所有页面action集合
import * as actions from './actions';
```

```
// 缓存actions，避免render重新加载
let cachedActions;
// action通过bindActionCreators绑定dispatch，
const bindActions = (dispatch, ownProps) => {
  if (!cachedActions) {
    cachedActions = {
      dispatch,
      actions: bindActionCreators(actions, dispatch),
    };
  }
  return cachedActions;
};
const connectWithActions = (
  mapStateToProps,
  mergeProps,
  options
) => (component) => connect(
    mapStateToProps, bindActions, mergeProps, options
)(component);

export default connectWithActions;

// 类似还有log中间件样子的等等。
```

HOC的不足

- HOC产生了许多无用的组件，加深了组件层级，性能和调试受影响。
- 多个HOC 同时嵌套，劫持props， 命名可能会冲突，且内部无法判断Props是来源于那个HOC。

5. Render Props

Render Props 你可以把它理解成 JavaScript 中的回调函数

```
// 实现一个控制modal visible的高阶组件
class ToggleVisible extends React.Component {
    state = {
        visible: false
    };
    toggle = () => {
        this.setState({visible: !this.state.visible});
    }
    render() {
        return (
            <>{this.props.children({visible, toggle})}</>
        );
    }
}
//使用
const EditUser = () => (
    <ToggleVisible>
        {({visible, toggle}) => (<>
            <Modal visible={visible}/>
            <Button onClick={toggle}>打开/关闭modal</Button>
```

```
        </>)}
    </ToggleVisible>
)
```

📣 优点

- 组件复用不会产生多余的节点，也就是不会产生多余的嵌套。
- 不用担心props命名问题。

6. 组合式组件（Compound Component）

子组件所需要的props在父组件会封装好，引用子组件的时候就没必要传递所有props了。组合组件核心的两个方法是React.Children.map和React.cloneElement。

例如下面 子组件需要的click事件转移到了父组件，通过父组件内部封装到子组件上，ant-design的很多group组件用到了此方法。

> 参考文章

```
class GroupButton extends React.PureComponent {
  state = {
    activeIndex: 0
  };
  render() {
    return (
      <>
        {React.Children.map(this.props.children, (child, index) =>
          child.type
            ? React.cloneElement(child, {
                active: this.state.activeIndex === index,
                onClick: () => {
                  this.setState({ activeIndex: index });
                  this.props.onChange(child.props.value);
                }
              })
            : child
        )}
      </>
    );
  }
}
// 用法
<GroupButton
  onChange={e => {
    console.log("onChange", e);
  }}
>
  <Button value="red">red</Button>
  <Button value="yellow">yellow</Button>
  <Button value="blue">blue</Button>
  <Button value="white">white</Button>
</GroupButton>
```

废话结束，开始进入正题。。。

React hooks

Hook 出现之前，组件之间复用状态逻辑很难，解决方案（HOC、Render Props）都需要重新组织组件结构， 且代码难以理解。在React DevTools 中观察过 React 应用，你会发现由 providers，consumers，高阶组件，render props 等其他抽象层组成的组件会形成“嵌套地狱”。

组件维护越来越复杂，譬如事件监听逻辑要在不同的生命周期中绑定和解绑，复杂的页面componentDidMount包涵很多逻辑，代码阅读性变得很差。

class组件中的this难以理解，且class 不能很好的压缩，并且会使热重载出现不稳定的情况。更多引子介绍参见官方介绍。

所以hook就为解决这些问题而来：

- 避免地狱式嵌套，可读性提高。
- 函数式组件，比class更容易理解。
- class组件生命周期太多太复杂，使函数组件存在状态。
- 解决HOC和Render Props的缺点。
- UI 和 逻辑更容易分离。

> 下面逐一介绍官方提供的hook API。

1. useState

函数组件有状态了

`const [state, setState] = useState(initialState);` state为变量， `setState` 修改 state值的方法，setState也是异步执行。

DEMO1

class this.setState更新是state是合并， useState中setState是替换。

```
function Example() {
  // 声明一个叫 "count" 的 state 变量
  const [count, setCount] = useState(0);
    const [obj, setData] = useState();
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

2. useEffect

忘记生命周期，记住副作用

```
useEffect(()  =>  {// Async Action}, ?[dependencies]); // 第二参数非必填
```

DEMO2

```
function Hook2() {
  const [data, setData] = useState();
  useEffect(() => {
    console.log("useEffect");
  });
  return (
    <div>
      {(() => {
        console.log("render");
        return null;
      })()}
      <p>data: {JSON.stringify(data)}</p>
    </div>
  );
}
```

执行结果：

```
[HMR] Waiting for update signal from WDS...
render
useEffect
>
```

结论：

- useEffect 是在render之后生效执行的。

DEMO3

```
import React, { useState, useEffect, useRef } from “react”;
function Demo3() {
  const [data, setData] = useState();
  useEffect(() => {
    console.log("useEffect-[]”);
    fetch(“https://www.mxnzp.com/api/lottery/common/latest?code=ssq”)
      .then(res => res.json())
      .then(res => {
        setData(res);
      });
  }, []);

  useEffect(() => {
    console.log("useEffect ---> 无依赖");
  });

  useEffect(() => {
    console.log(“useEffect 依赖data:  data发生了变化”);
  }, [data]);
```

```
  return (
    <div>
      <p>data: {JSON.stringify(data)}</p>
    </div>
  );
}
export default Demo3;
```

执行结果：

```
[HMR] Waiting for update signal from WDS...
render
useEffect--[]
useEffect ---> 无依赖
useEffect 依赖data:  data发生了变化
render
useEffect ---> 无依赖
useEffect 依赖data:  data发生了变化
```

结论：

- effect在render后按照前后顺序执行。
- effect在没有任何依赖的情况下，render后每次都按照顺序执行。
- effect内部执行是异步的。
- 依赖`[]`可以实现类似`componentDidMount`的作用，但最好忘记生命周期， 只记副作用。

DEMO4

```
import React, { useState, useEffect, useRef } from "react";

function Demo4() {
  useEffect(() => {
    console.log(“useEffect1”);
    const timeId = setTimeout(() => {
      console.log(“useEffect1-setTimeout-2000”);
    }, 2000);
    return () => {
      clearTimeout(timeId);
    };
  }, []);
  useEffect(() => {
    console.log("useEffect2");
    const timeId = setInterval(() => {
      console.log("useEffect2-setInterval-1000");
    }, 1000);
    return () => {
```

```
      clearInterval(timeId);
    };
  }, []);
  return (
    <div>
      {() => {
        console.log("render");
        return null;
      }()}
      <p>demo4</p>
    </div>
  );
}
export default Demo4;
```

执行结果：

```
[HMR] Waiting for update signal from WDS...
render
useEffect1
useEffect2
useEffect2-setInterval-1000
useEffect1-setTimeout-2000
24 useEffect2-setInterval-1000
>
```

结论：

- effect回调函数是按照先后顺序同时执行的。
- effect的回调函数返回一个匿名函数，相当于`componentUnMount`的钩子函数，一般是remove eventLisenter， clear timeId等，主要是组件卸载后防止内存泄漏。

> 综上所述，useEffect 就是监听每当依赖变化时，执行回调函数的存在函数组件中的钩子函数。

3. useContext

跨组件共享数据的钩子函数

```
const value = useContext(MyContext);
// MyContext 为 context 对象 (React.createContext 的返回值)
// useContext 返回MyContext的返回值。
// 当前的 context 值由上层组件中距离当前组件最近的<MyContext.Provider> 的 value prop 决定。
```

DEMO5

```
import React, { useContext, useState } from "react";
const MyContext = React.createContext();
function Demo5() {
  const [value, setValue] = useState("init");
  console.log("Demo5");
```

```
  return (
    <div>
      {(() => {
        console.log("render");
        return null;
      })()}
      <button onClick={() => {
        console.log('click: 更新value')
        setValue(`${Date.now()}_newValue`)
      }}>
        改变value
      </button>
      <MyContext.Provider value={value}>
        <Child1 />
        <Child2 />
      </MyContext.Provider>
    </div>
  );
}

function Child1() {
  const value = useContext(MyContext);
  console.log("Child1-value", value);
  return <div>Child1-value: {value}</div>;
}

function Child2(props) {
  console.log('Child2')
  return <div>Child2</div>;
}
```

执行结果：

```
[HMR] Waiting for update signal from WDS...
Demo5
render
Child1-value init
Child2
click: 更新value
Demo5
render
Child1-value 1575629543704_newValue
Child2
>
```

结论：

- useContext 的组件总会在 context 值变化时重新渲染， 所以 `<MyContext.Provider>` 包裹的越多，层级越深，性能会造成影响。
- `<MyContext.Provider>` 的 value 发生变化时候， 包裹的组件无论是否订阅content value，所有组件都会从新渲染。
- demo中child2 不应该rerender, 如何避免不必要的render？ * 使用React.memo优化。

```
const Child2 = React.memo((props) => {
  return <div>Child2</div>;
})
```

执行结果：

```
[HMR] Waiting for update signal from WDS...
Demo5
render
Child1-value init
Child2
click: 更新value
Demo5
render
Child1-value 1575629628174_newValue
>
```

注意： 默认情况下React.memo只会对复杂对象做浅层对比，如果你想要控制对比过程，那么请将自定义的比较函数通过第二个参数传入来实现。 参考链接

4. useRef

传送门

```
const refContainer = useRef(initialValue);
```

- useRef 返回一个可变的 ref 对象, 和自建一个 {current: ...} 对象的唯一区别是，useRef 会在每次渲染时返回同一个 ref 对象, 在整个组件的生命周期内是唯一的。
- useRef 可以保存任何可变的值。其类似于在 class 中使用实例字段的方式。 **总结：**
- *useRef 可以存储那些不需要引起页面重新渲染的数据。*
- 如果你刻意地想要从某些异步回调中读取 /最新的/ state，你可以用 一个 ref 来保存它，修改它，并从中读取。

5. useReducer

```
const [state, dispatch] = useReducer(reducer, initialState);
```

`reducer` 就是一个只能通过 `action` 将 `state` 从一个过程转换成另一个过程的纯函数; `useReducer` 就是一种通过 `(state,action) => newState` 的过程，和 `redux` 工作方式一样。数据流： dispatch(action) => reducer更新state => 返回更新后的state

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case ‘decrement’:
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}
function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

官方推荐以下场景需要useReducer更佳：

- state 逻辑较复杂且包含多个子值， 可以集中处理。
- 下一个 state 依赖于之前的 state 。
- 想更稳定的构建自动化测试用例。
- 想深层级修改子组件的一些状态，使用 useReducer 还能给那些会触发深更新的组件做性能优化，因为 你可以向子组件传递 dispatch 而不是回调函数 。
    - 使用reducer有助于将读取与写入分开。 DEMO6

```
const fetchReducer = (state, action) => {
  switch (action.type) {
    case “FETCH_INIT":
      return {
        ...state,
        loading: true,
        error: false
      };
    case “FETCH_SUCCESS”:
      return {
        ...state,
        loading: false,
        error: false,
        data: action.payload
```

```
      };
    case "FETCH_FAIL":
      return {
        …state,
        loading: false,
        error: true
      };
    default:
      throw new Error();
  }
};

function Demo6() {
  const [state, dispatch] = useReducer(fetchReducer, {
    loading: false,
    error: false,
    msg: "",
    data: {}
  });

  const getData = useCallback(async () => {
    try {
      dispatch({ type: "FETCH_INIT" });
      const response = await fetch(
        "https://www.mxnzp.com/api/lottery/common/latest?code=ssq"
      );
      const res = await response.json();

      if (res.code) {
        dispatch({ type: "FETCH_SUCCESS", payload: res.data });
      } else {
        dispatch({ type: “FETCH_FAIL”, payload: res.msg });
      }
    } catch (error) {
      dispatch({ type: “FETCH_FAIL”, payload: error });
    }
  }, []);

  useEffect(() => {
    getData();
  }, [getData]);

  return (
    <Loading loading={state.loading}>
      <p>开奖号码:  {state.data.openCode}</p>
    </Loading>
  );
}
```

`demo6` useReducer处理了多个可以用useState实现的逻辑，包括 `loading, error, msg, data`。

useContext 和 useReducer 模拟redux管理状态

```
import React, { useReducer, useContext } from “react”;

const ModalContext = React.createContext();

const visibleReducer = (state, action) => {
  switch (action.type) {
    case “CREATE”:
      return { ...state, ...action.payload };
    case "EDIT":
      return { ...state, ...action.payload };
    default:
      return state;
  }
};
function Demo7() {
  const initModalVisible = {
    create: false,
    edit: false
  };
  const [state, dispatch] = useReducer(visibleReducer, initModalVisible);

  return (
    <ModalContext.Provider value={{ visibles: state, dispatch }}>
      <Demo7Child />
    </ModalContext.Provider>
  );
}
function Demo7Child() {
  return (
    <div>
      Demo7Child
      <Detail />
    </div>
  );
}
function Detail() {
  const { visibles, dispatch } = useContext(ModalContext);
  console.log("contextValue", visibles);
  return (
    <div>
      <p>create: {`${visibles.create}`}</p>
      <button
        onClick={() => dispatch({ type: "CREATE", payload: { create: true } })}
      >
        打开创建modal
      </button>
    </div>
  );
}
export default Demo7;
```

逻辑很清晰的抽离出来，context value中的值不需要在组件中透传，*即用即取*。DEMO7

> *注意* React 会确保 dispatch 函数的标识是稳定的，并且不会在组件重新渲染时改变。这就是为什么可以安全地从 useEffect 或 useCallback 的依赖列表中省略 dispatch。

6. useCallback

语法：

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

返回一个 memoized 回调函数。

`useCallback` 解决了什么问题？先看DEMO8

```
import React, { useRef, useEffect, useState, useCallback } from “react”;

function Child({ event, data }) {
  console.log("child-render");
  // 第五版
  useEffect(() => {
    console.log(“child-useEffect”);
    event();
  }, [event]);
  return (
    <div>
      <p>child</p>
      {/* <p>props-data: {data.data && data.data.openCode}</p> */}
      <button onClick={event}>调用父级event</button>
    </div>
  );
}

const set = new Set();

function Demo8() {
  const [count, setCount] = useState(0);
  const [data, setData] = useState({});

  // 第一版
  // const handle = async () => {
  //   const response = await fetch(
  //     "https://www.mxnzp.com/api/lottery/common/latest?code=ssq"
  //   );
  //   const res = await response.json();
  //   console.log("handle", data);
  //   setData(res);
  // };
```

```
// 第二版
// const handle = useCallback(async () => {
//   const response = await fetch(
//     “https://www.mxnzp.com/api/lottery/common/latest?code=ssq"
//   );
//   const res = await response.json();
//   console.log(“handle”, data);
//   setData(res);
// });

// 第三版
// const handle = useCallback(async () => {
//   const response = await fetch(
//     “https://www.mxnzp.com/api/lottery/common/latest?code=ssq”
//   );
//   const res = await response.json();
//   setData(res);
//   console.log(“useCallback”, data);
//   // eslint-disable-next-line react-hooks/exhaustive-deps
// }, []);

// // 第四版
// const handle = useCallback(async () => {
//   const response = await fetch(
//     “https://www.mxnzp.com/api/lottery/common/latest?code=ssq"
//   );
//   const res = await response.json();
//   setData(res);
//   console.log(“parent-useCallback", data);
//   // eslint-disable-next-line react-hooks/exhaustive-deps
// }, []);

// 第五版
const handle = useCallback(async () => {
  const response = await fetch(
    "https://www.mxnzp.com/api/lottery/common/latest?code=ssq"
  );
  const res = await response.json();
  setData(res);
  console.log("parent-useCallback", data);
  // eslint-disable-next-line react-hooks/exhaustive-deps
}, [count]);
set.add(handle);

console.log(“parent-render====>”, data);
return (
  <div>
    <button
      onClick={e => {
        setCount(count + 1);
      }}
    >
      count++
```

```
        </button>
        <p>set size: {set.size}</p>
        <p>count:{count}</p>
        <p>data: {data.data && data.data.openCode}</p>
        <p>--------------------------------</p>
        <Child event={handle} />
      </div>
    );
}
export default Demo8;
```

结论：

- 第一版：每次render，handle都是新的函数，且每次都能拿到最新的data。
- 第二版：用useCallback包裹handle，每次render， handle也是新的函数，且每次都能拿到最新的data， 和一版效果一样， 所以不建议这么用。
- 第三版：useCallback假如第二个参数deps，handle会被memoized， 所以每次data都是第一次记忆时候的data（闭包）。
- 第四版： useCallback依赖count的变化，每当useCallback 变化时，handle会被重新memoized。
- 第五版：每当count变化时，传入子组件的函数都是最新的，所以导致child的useEffect执行。 **总结：**
- useCallback将返回一个*记忆的回调版本*，仅在其中一个依赖项已更改时才更改。
- 当将回调传递给依赖于引用相等性的优化子组件以防止不必要的渲染时，此方法很有用。
- 使用回调函数作为参数传递，每次render函数都会变化，也会导致子组件rerender， useCallback可以优化rerender。 *疑问：如何优化子组件不必要的渲染?*

7. useMemo

语法： `const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);`；返回一个 memoized 值，和 `useCallback` 一样，当依赖项发生变化，才会重新计算 memoized 的值,。 useMemo和useCallback不同之处是：它允许你将 *memoized* 应用于任何值类型（不仅仅是函数）。 DEMO9

```
import React, { useState, useMemo } from “react”;

function Demo9() {
  const [count, setCount] = useState(0);
  const handle = () => {
    console.log(“handle”, count);
    return count;
  };

  const handle1 = useMemo(() => {
    console.log("handle1", count);
    return count;
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []);

  const handle2 = useMemo(() => {
    console.log(“handle2”, count);
      // 大计算量的方法
    return count;
  }, [count]);
```

```
  console.log("render-parent");

  return (
    <div>
      <p>
        demo9: {count}
        <button onClick={() => setCount(count + 1)}>++count</button>
      </p>
      <p>--------------------</p>
      <Child handle={handle1} />
    </div>
  );
}

function Child({ handle }) {
  console.log("render-child");
  return (
    <div>
      <p>child</p>
      <p>props-data: {handle}</p>
    </div>
  );
}
export default Demo9;
```

总结:

- `useMemo` 会在 `render` 前执行。
- 如果没有提供依赖项数组，useMemo 在每次渲染时都会计算新的值。
- `useMemo` 用于返回 `memoize`,防止每次render时*大计算量*带来的开销。
- 使用 `useMemo` 优化需谨慎， 因为优化本身也带来了计算，*大多数时候，你不需要考虑去优化不必要的重新渲染。*

其他Hook

1. useImperativeHandle

```
// ref: 需要传递的ref
// createHandle: 需要暴露给父级的方法。
// deps: 依赖
useImperativeHandle(ref, createHandle, [deps])
```

`useImperativeHandle` 应当与 `forwardRef` 一起使用。先看DEMO10

```
import React, {
  useRef,
  forwardRef,
  useImperativeHandle,
  useEffect,
  useState
} from "react";
```

```
const Child = forwardRef((props, ref) => {
  const inputEl = useRef();
  const [value, setVal] = useState("");
  // 第一版
  // useImperativeHandle(ref, () => {
  //   console.log("useImperativeHandle");
  //   return {
  //     value,
  //     focus: () => inputEl.current.focus()
  //   };
  // });

  // 第二版
  useImperativeHandle(
    ref,
    () => {
      console.log(“useImperativeHandle");
      return {
        value,
        focus: () => inputEl.current.focus()
      };
    },
    // eslint-disable-next-line react-hooks/exhaustive-deps
    []
  );

  return (
    <input
      ref={inputEl}
      onChange={e => setVal(e.target.value)}
      value={value}
      {...props}
    />
  );
});
function Demo10() {
  const inputEl = useRef(null);

  useEffect(() => {
    console.log(“parent-useEffect”, inputEl.current);
    inputEl.current.focus();
  }, []);

  function click() {
    console.log("click:", inputEl.current);
    inputEl.current.focus();
  }
  console.log(“Demo10”, inputEl.current);
  return (
    <div>
      <Child ref={inputEl} />
      <button onClick={click}>click focus</button>
```

```
      </div>
    );
}
export default Demo10;
```

结论：

- `useImperativeHandle`在当前组件render后执行。
- 第一版：没有deps，每当rerender时，`useImperativeHandle`都会执行， 且能拿到 state中最新的值, 父组件调用传入的方法也是最新。
- 第二版： 依赖`[]`，每当rerender时,`useImperativeHandle`不会执行，且不会更新到父组件。
- 第三版：依赖传入的state值`[value]`， 达到想要的效果。

2. useDebugValue

不常用, 只能在React Developer Tools看到，详见官方传送门 。

DEMO11

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(false);
  // 在开发者工具中的这个 Hook 旁边显示标签
  // e.g. "FriendStatus: Online"
  useDebugValue(isOnline ? "Online" : "Offline");
  return isOnline;
}
function Demo11() {
  const isOnline = useFriendStatus(567);
  return <div>朋友是否在线：{isOnline ? "在线" : "离线"}</div>;
}
```

3. useLayoutEffect

很少用，与`useEffect`相同，但它会在所有的 DOM 变更之后同步调用 effect, 详见官方传送门。