React

视频链接：

- 【一(01-05)】Udemy - The Ultimate React Course 2023 React, Redux & More 2023-6_哔哩哔哩_bilibili
- 【二(06-10)】Udemy - The Ultimate React Course 2023 React, Redux & More 2023-6_哔哩哔哩_bilibili
- 【三(11-15)】Udemy - The Ultimate React Course 2023 React, Redux & More 2023-6_哔哩哔哩_bilibili
- 【四(16-20)】Udemy - The Ultimate React Course 2023 React, Redux & More 2023-6_哔哩哔哩_bilibili
- 【五(21-25)】Udemy - The Ultimate React Course 2023 React, Redux & More 2023-6_哔哩哔哩_bilibili
- 【六(26-31)完结】Udemy - The Ultimate React Course 2023 React, Redux & More 2023-6_哔哩哔哩_bilibili

代码链接：https://github.com/jonasschmedtmann/ultimate-react-course.git

基础

- 在线环境：react.new
- Pure react

```
  <body>
    <div id="root"></div>

    <script
      src="https://unpkg.com/react@18/umd/react.development.js"
      crossorigin
    ></script>
    <script
      src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
      crossorigin
    ></script>

    <script>
      function App() {
        return React.createElement("header", null, "Hello React!");
      }

      const root = ReactDOM.createRoot(document.getElementById("root"));
      root.render(React.createElement(App));
    </script>
  </body>
</html>
```

- JS基础
  - Destructuring：当一个对象有多个属性的时候，可以直接用大括号取出其中的部分属性；列表可以用中括号。剩下的可以用 `...`

```
// const primaryGenre = genres[0];
// const secondaryGenre = genres[1];

const [primaryGenre, secondaryGenre, ...otherGenres] = genres;

console.log(primaryGenre, secondaryGenre, otherGenres);  scienc
```

  - Spread：也是用 `...` 可以取出list或者对象的内容
  - Template literals，用 `` `${}` ``，里面支持各种js

```
const summary = `${title}, a ${pages}-page long book, was written by ${author}`;
summary;  The Lord of the Rings, a 1216-page long book, was written by J. R. R. Tolkien
```

  - 短路效果：使用 `boolean_value && "Yes"`，注意 falsy value是 `0, '', null, undefined`。或的短路可以用来设置默认值。
  - `??` 如果前一个值是 `null` 或者 `undefined` 就使用后面一个值

- 我们可以使用 `?` 来看某个属性是否为 `null` 或者 `undefined`，然后我们后面加一个 `??` 就可以使用默认值。例如 `pro?.key ?? 0`
- `map` 函数和 `filter` 函数，可以级联
- `reduce` 函数例子

```
const pagesAllBooks = books.reduce((acc, book) => acc + book.pages, 0);
pagesAllBooks;  3227
```

- `sort` 例子

```
const arr = [3, 7, 1, 9, 6];
const sorted = arr.sort((a, b) => b - a);
sorted;  [ 9, 7, 6, 3, 1 ]
arr;  [ 9, 7, 6, 3, 1 ]
```

```
const arr = [3, 7, 1, 9, 6];
const sorted = arr.slice().sort((a, b) => a - b);
sorted;  [ 1, 3, 6, 7, 9 ]
arr;  [ 3, 7, 1, 9, 6 ]
```

- 添加删除更新

```
// 1) Add book object to array
const newBook = {
  id: 6,
  title: "Harry Potter and the Chamber of Secrets",
  author: "J. K. Rowling",
};
const booksAfterAdd = [...books, newBook];
booksAfterAdd;  ... librarything: [Object] } }, { id: 6, title: 'Harry Potter and the Chamber of Secrets', auth

// 2) Delete book object from array
const booksAfterDelete = booksAfterAdd.filter((book) => book.id !== 3);
booksAfterDelete;  ... librarything: [Object] } }, { id: 6, title: 'Harry Potter and the Chamber of Secrets', a
```

```
// 3) Update book object in the array
const booksAfterUpdate = booksAfterDelete.map((book) =>
  book.id === 1 ? { ...book, pages: 1 } : book
);
booksAfterUpdate;  ... librarything: [Object] } }, { id: 6, title: 'Harry Potter and the Chamber of Secrets', a
```

- className中增加三元运算符

```
<li className={`pizza ${pizzaObj.soldOut ? "sold-out" : ""}`}>
```

- 创建数组

```
<select>
  {Array.from({ length: 20 }, (_, i) => i + 1).map
  ((num) => (
    <option value={num} key={num}>
      {num}
    </option>
  ))}
</select>
<input type="text" placeholder="Item..." />
```

- React基础
  - set函数：如果新的值依赖之前的值要是使用回调函数，否则不会重复更新。同时 react 都是使用不可变量，所以列表不能使用push

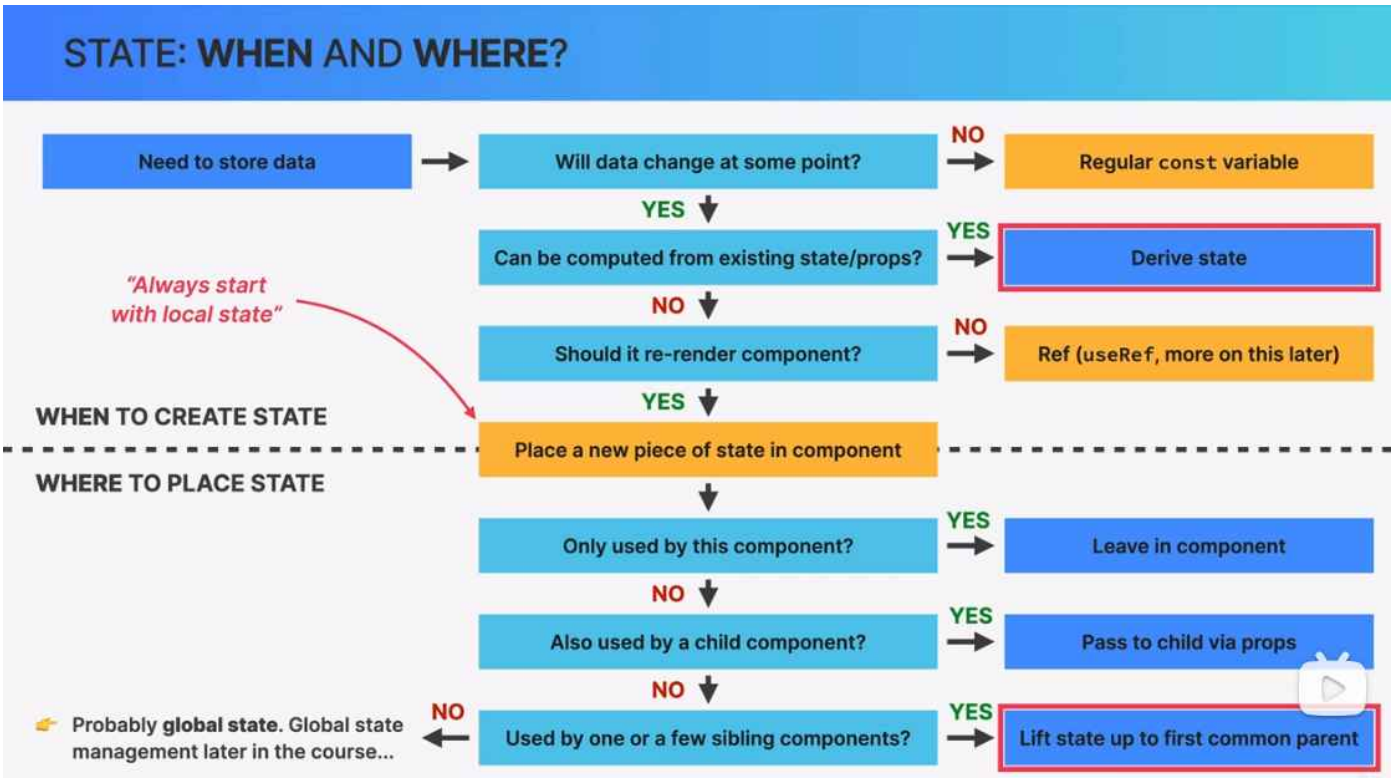
```
function handleNext() {
  if (step < 3) {
    setStep((s) => s + 1);
    setStep((s) => s + 1);
  }
```

```
function handleAddItems(item) {
  setItems((items) => [...items, item]);
}
```

  - 默认submit的时候会reload页面导致输入的内容消失，所以可以用如下的方法阻止

```
function handleSubmit(e) {
  e.preventDefault();
}
```

  - `input` 一般会设置 `value` 和 `onChange` ，和一个state绑定
  - 状态选择



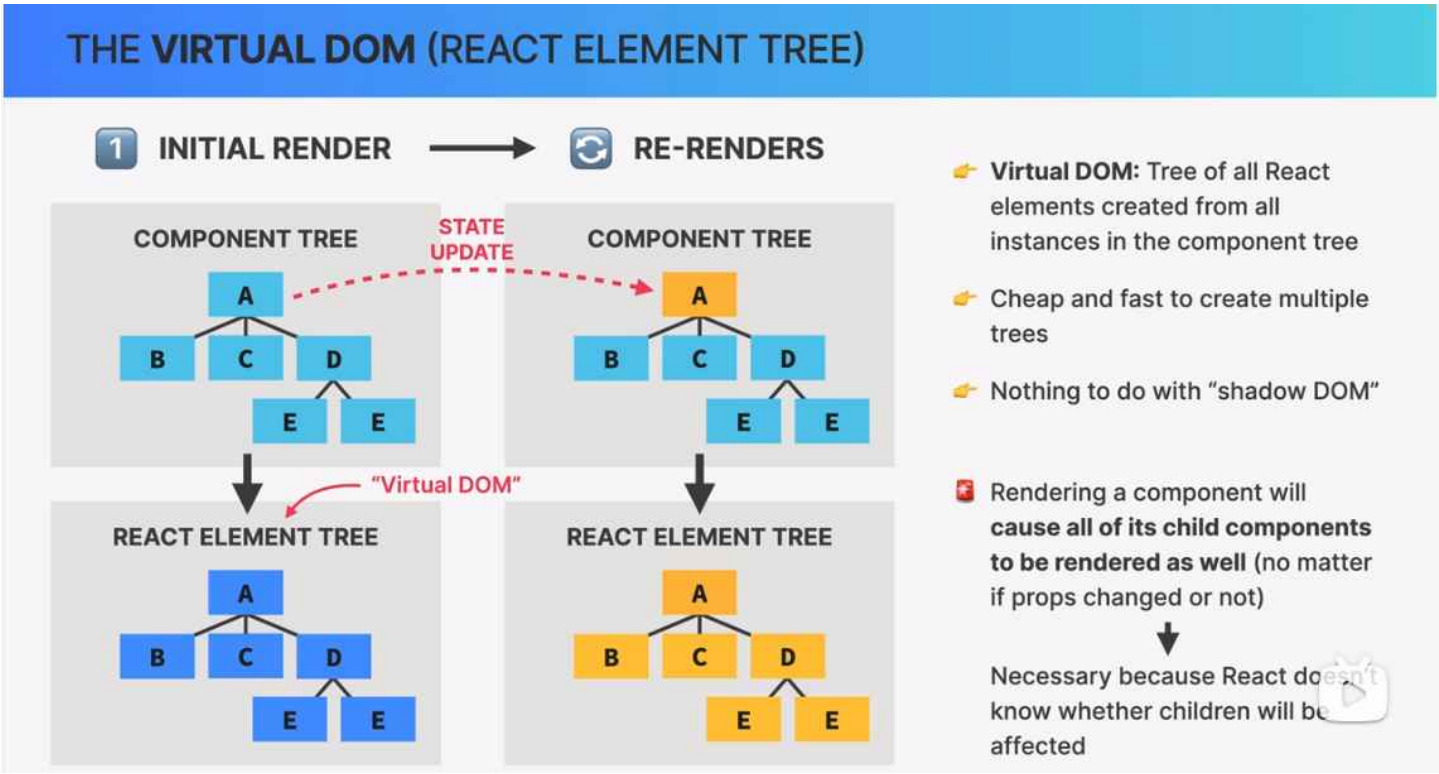
- Lift up state，如果兄弟组件需要用到一些state，可以放到父组件中然后通过prop传给子组件

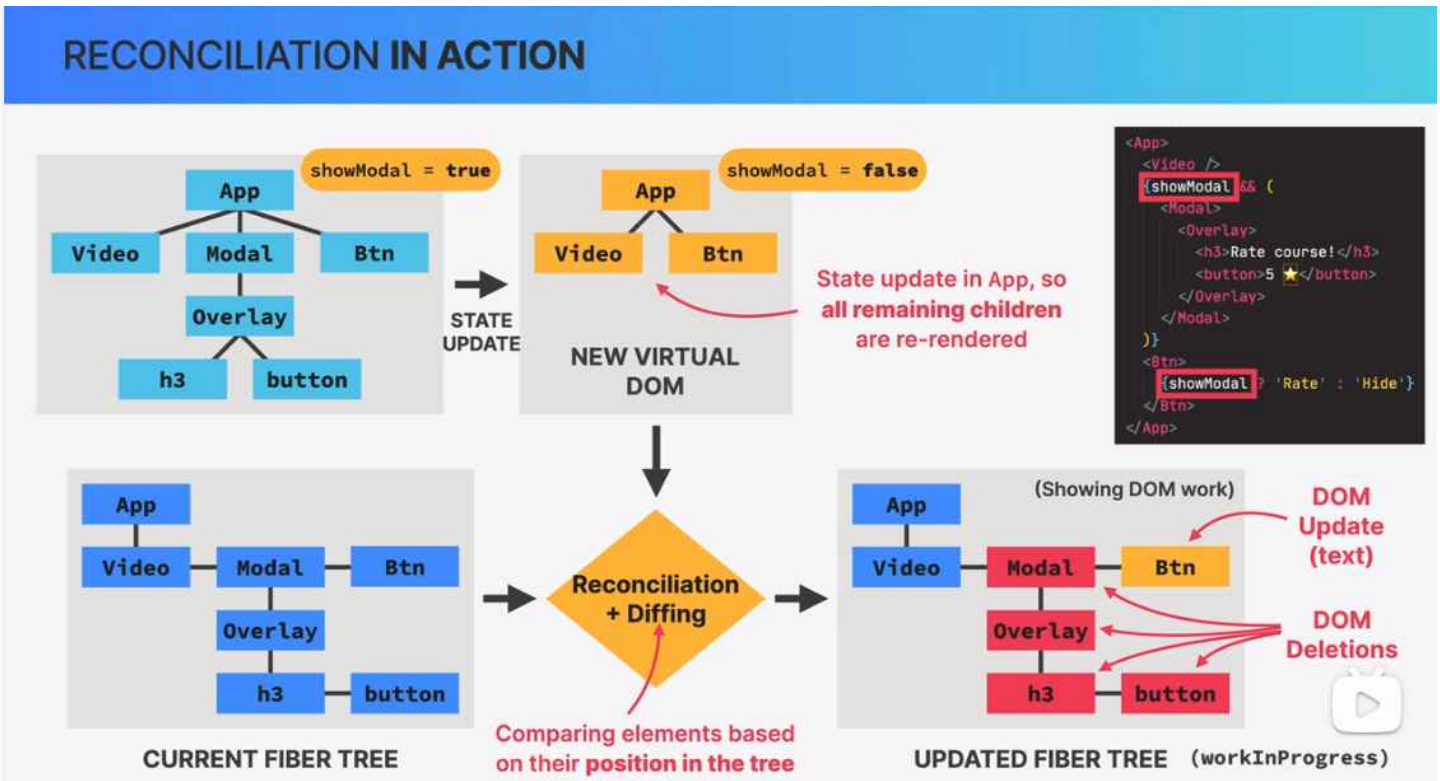
进阶（重要）

- Prop drilling问题，就是我们需要将一个prop传多级下去，解决方法是 component composition，就是将组件通过children传下去

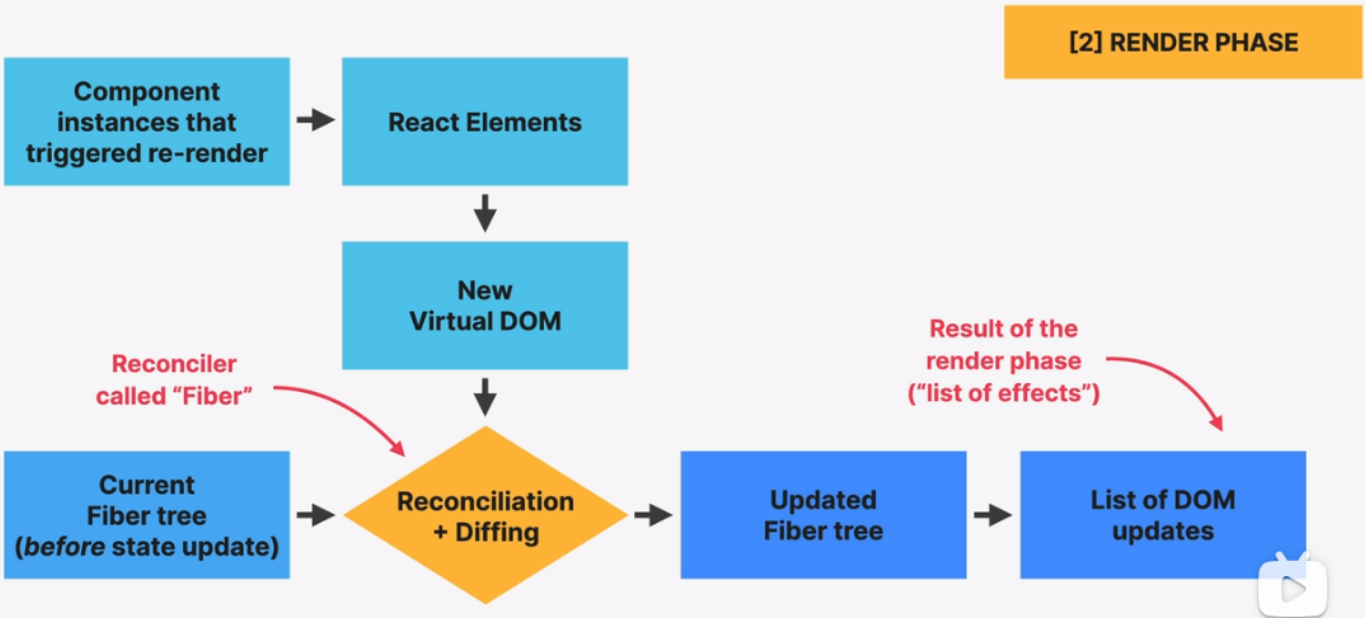
```
return (
  <>
    <NavBar>
      <Search />
      <NumResults movies={movies} />
    </NavBar>

    <Main>
      <ListBox>
        <MovieList movies={movies} />
      </ListBox>
      <WatchedBox />
    </Main>
  </>
```

- React渲染，首先将component tree变成 Virtual DOM，当有修改的时候会更新该节点以及它的所有后代（只是在Virtual DOM上，不一定是最后的DOM）。Virtual DOM之后变成Fiber Tree，这些具体省略。

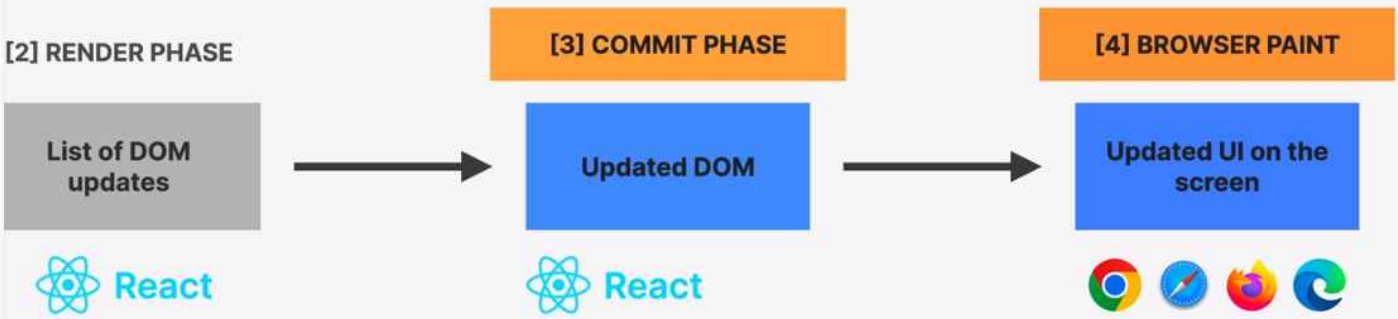




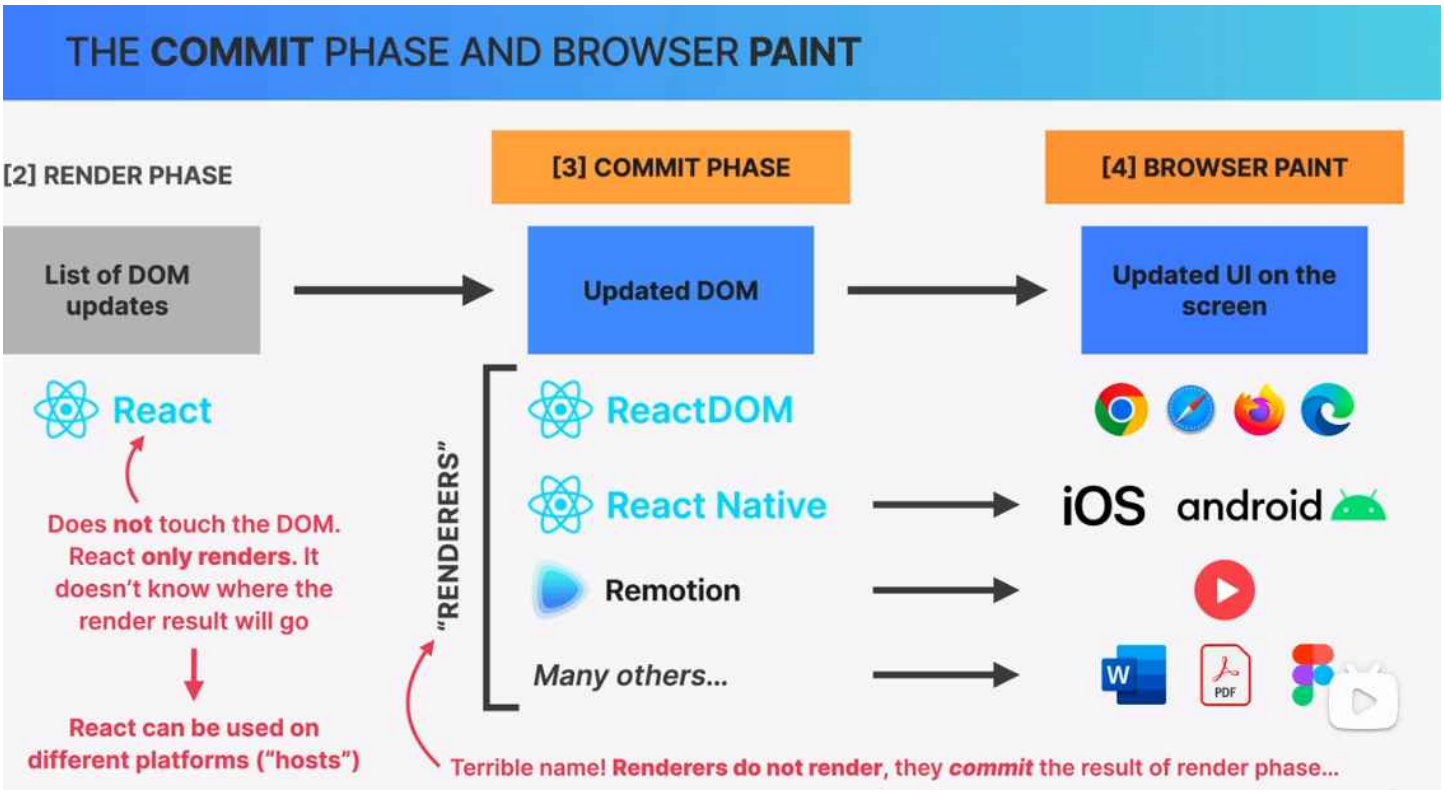
THE **RENDER** PHASE



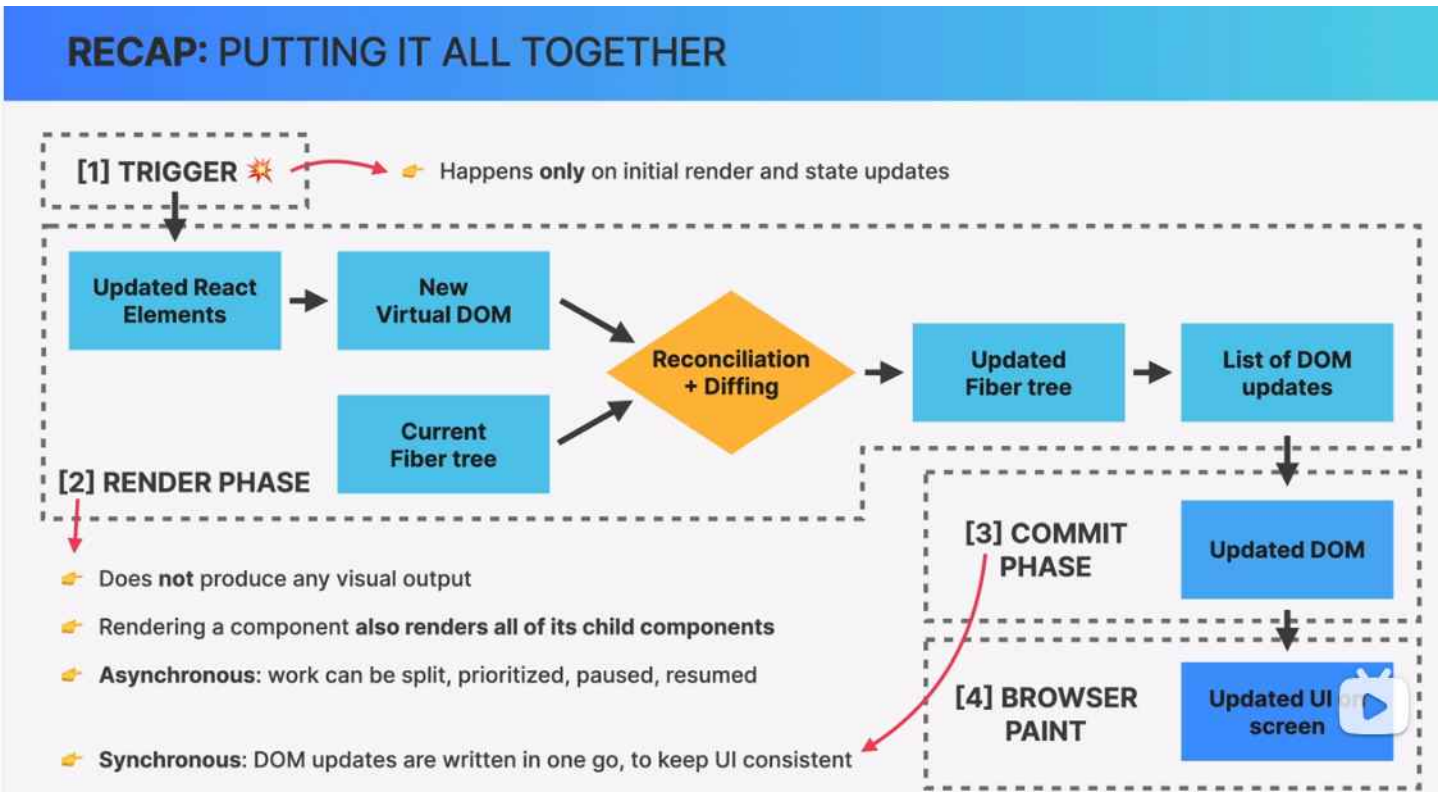
THE **COMMIT** PHASE AND BROWSER **PAINT**



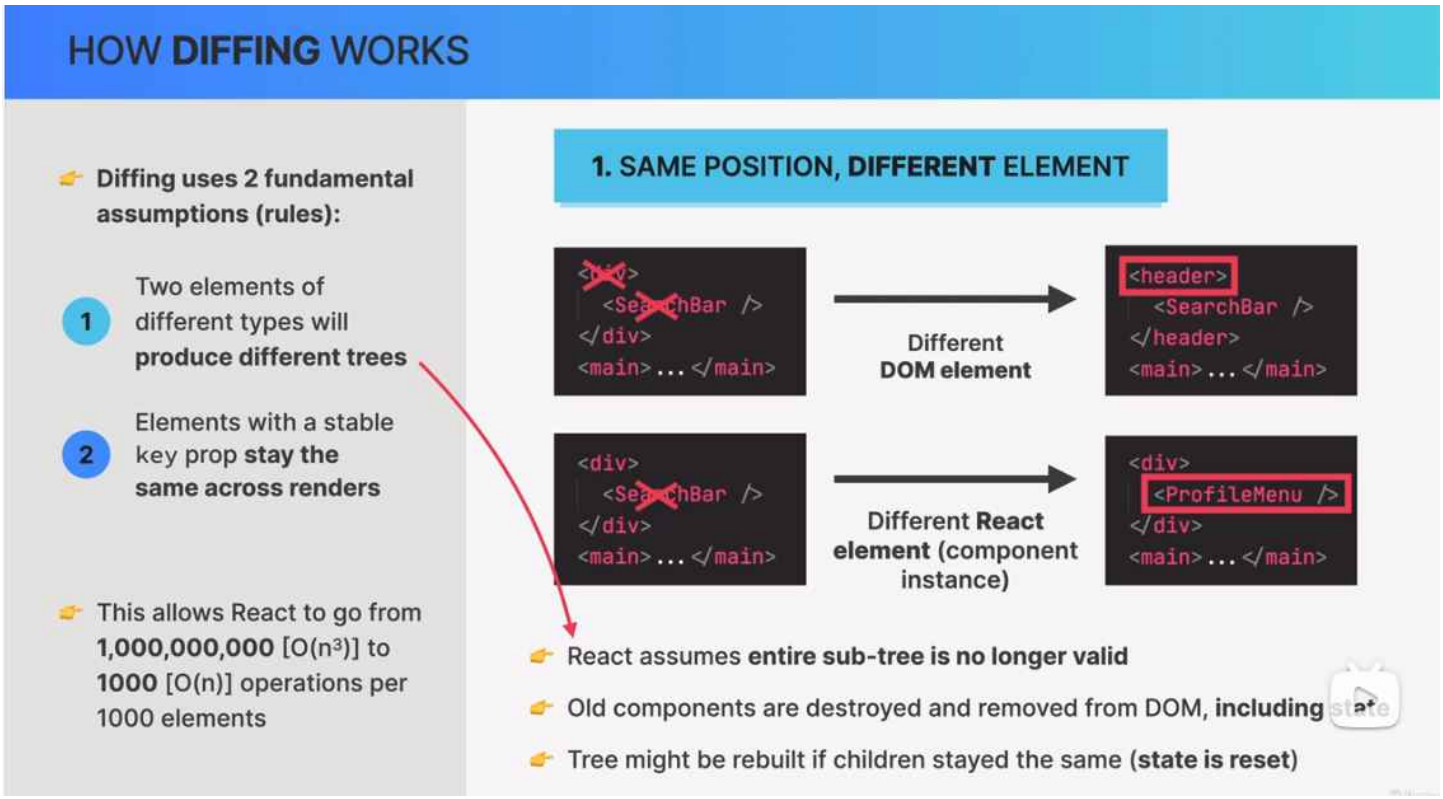
- **React writes to the DOM**: insertions, deletions, and updates (list of DOM updates are "flushed" to the DOM)
- **Committing is synchronous**: DOM is updated in one go, it can't be interrupted. This is necessary so that the DOM never shows partial results, ensuring a consistent UI (in sync with state at all times)
- After the commit phase completes, the `workInProgress` fiber tree becomes the `current` tree **for the next render cycle**

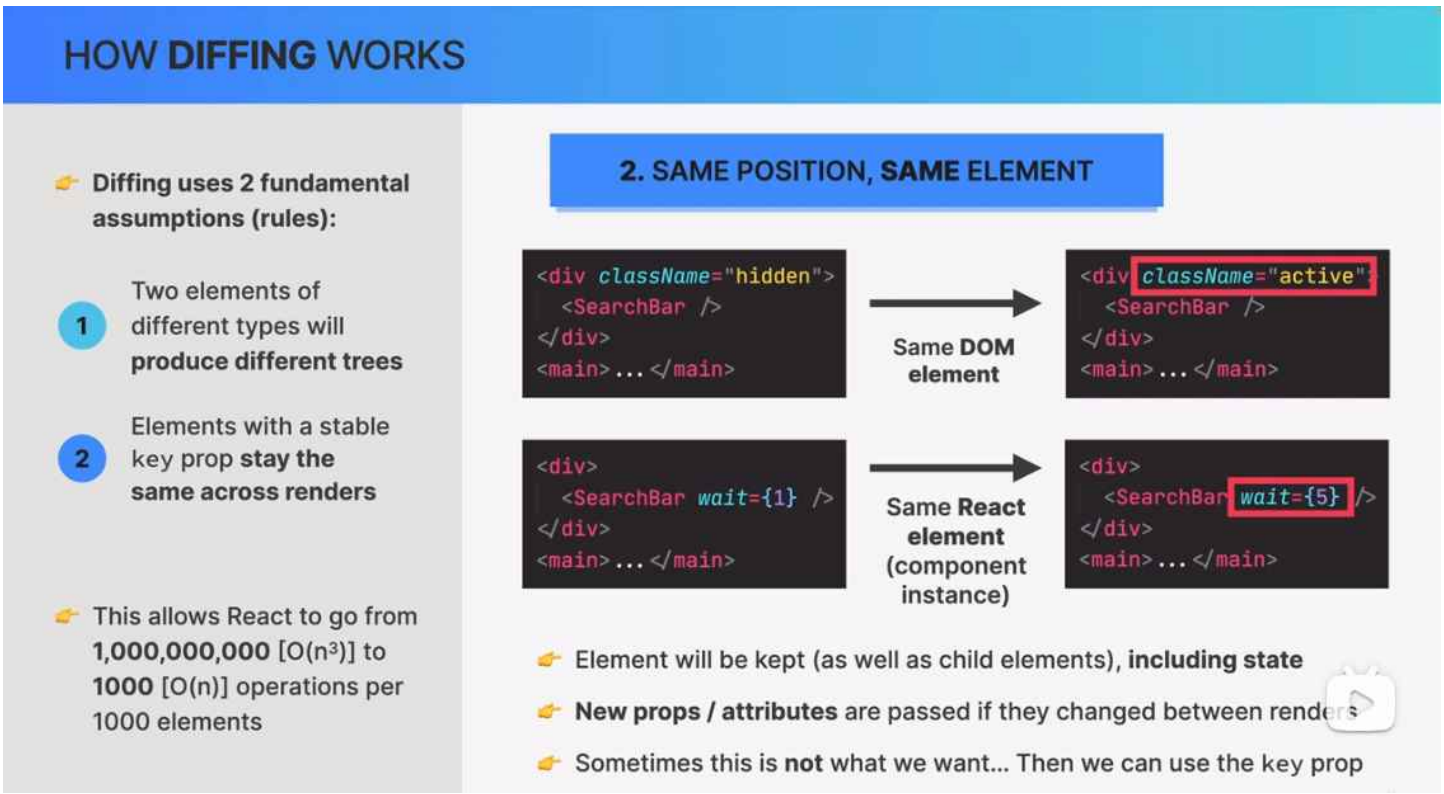

THE COMMIT PHASE AND BROWSER PAINT
[2] RENDER PHASE
[3] COMMIT PHASE
[4] BROWSER PAINT
List of DOM updates
Updated DOM
Updated UI on the screen
React
Does not touch the DOM. React only renders. It doesn't know where the render result will go
React can be used on different platforms ("hosts")
"RENDERERS"
ReactDOM
React Native
Remotion
Many others...
iOS
android
PDF
Terrible name! Renderers do not render, they commit the result of render phase...

总结：


RECAP: PUTTING IT ALL TOGETHER
[1] TRIGGER
Happens only on initial render and state updates
Updated React Elements
New Virtual DOM
Current Fiber tree
Reconciliation + Diffing
Updated Fiber tree
List of DOM updates
[2] RENDER PHASE
[3] COMMIT PHASE
Updated DOM
[4] BROWSER PAINT
Updated UI screen
Does not produce any visual output
Rendering a component also renders all of its child components
Asynchronous: work can be split, prioritized, paused, resumed
Synchronous: DOM updates are written in one go, to keep UI consistent

- Diff的流程


HOW DIFFING WORKS
Diffing uses 2 fundamental assumptions (rules):
1 Two elements of different types will produce different trees
2 Elements with a stable key prop stay the same across renders
This allows React to go from 1,000,000,000 [O(n³)] to 1000 [O(n)] operations per 1000 elements
1. SAME POSITION, DIFFERENT ELEMENT
 <SearchBar /> <main>...</main>
Different DOM element
<header> <SearchBar /> </header> <main>...</main>
 <SearchBar /> <main>...</main>
Different React element (component instance)
 <ProfileMenu /> <main>...</main>
React assumes entire sub-tree is no longer valid
Old components are destroyed and removed from DOM, including
Tree might be rebuilt if children stayed the same (state is reset)


HOW DIFFING WORKS
Diffing uses 2 fundamental assumptions (rules):
1 Two elements of different types will produce different trees
2 Elements with a stable key prop stay the same across renders
This allows React to go from 1,000,000,000 [O(n³)] to 1000 [O(n)] operations per 1000 elements
2. SAME POSITION, SAME ELEMENT
 <SearchBar /> <main>...</main>
Same DOM element
 <SearchBar /> <main>...</main>
 <SearchBar wait={1} /> <main>...</main>
Same React element (component instance)
 <SearchBar wait={5} /> <main>...</main>
Element will be kept (as well as child elements), including state
New props / attributes are passed if they changed between renders
Sometimes this is not what we want... Then we can use the key prop

- key的作用

KEY PROP

- Special prop that we use to tell the diffing algorithm that an element is **unique**
- Allows React to **distinguish** between multiple instances of the same component type
- When a key **stays the same across renders**, the element will be kept in the DOM (*even if the position in the tree changes*)

  1 ***Using keys in lists***

- When a key **changes between renders**, the element will be destroyed and a new one will be created (*even if the position in the tree is the same as before*)

  2 ***Using keys to reset state***

1. KEYS IN **LISTS** [STABLE KEY]

👎 NO KEYS

```
<ul>
  <Question question={q[1]} />
  <Question question={q[2]} />
</ul>
```

ADDING NEW LIST ITEM

```
<ul>
  <Question question={q[0]} />
  <Question question={q[1]} />
  <Question question={q[2]} />
</ul>
```

- Same elements, but **different position in tree**, so they are removed and recreated in the DOM (*bad for performance*)

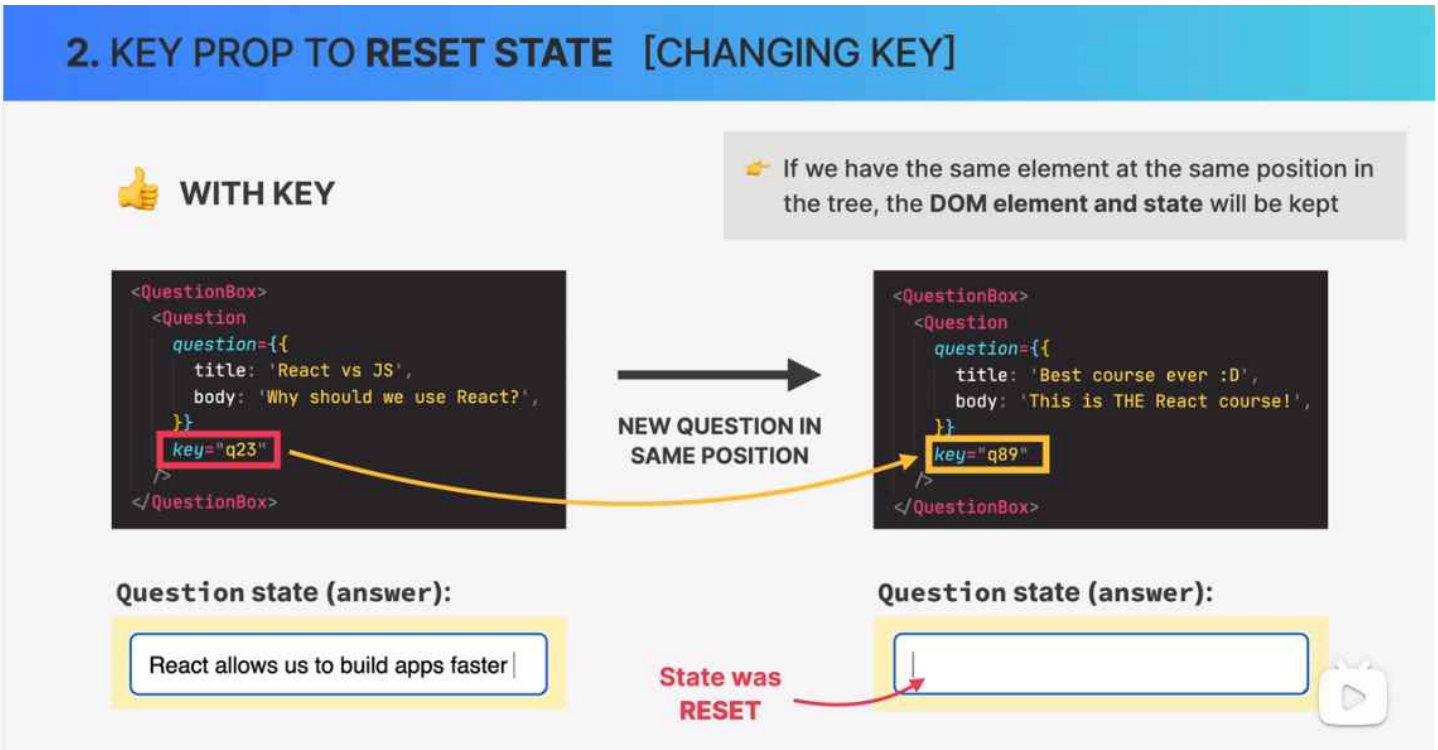
👍 WITH KEYS

```
<ul>
  <Question key='q1' question={q[1]} />
  <Question key='q2' question={q[2]} />
</ul>
```

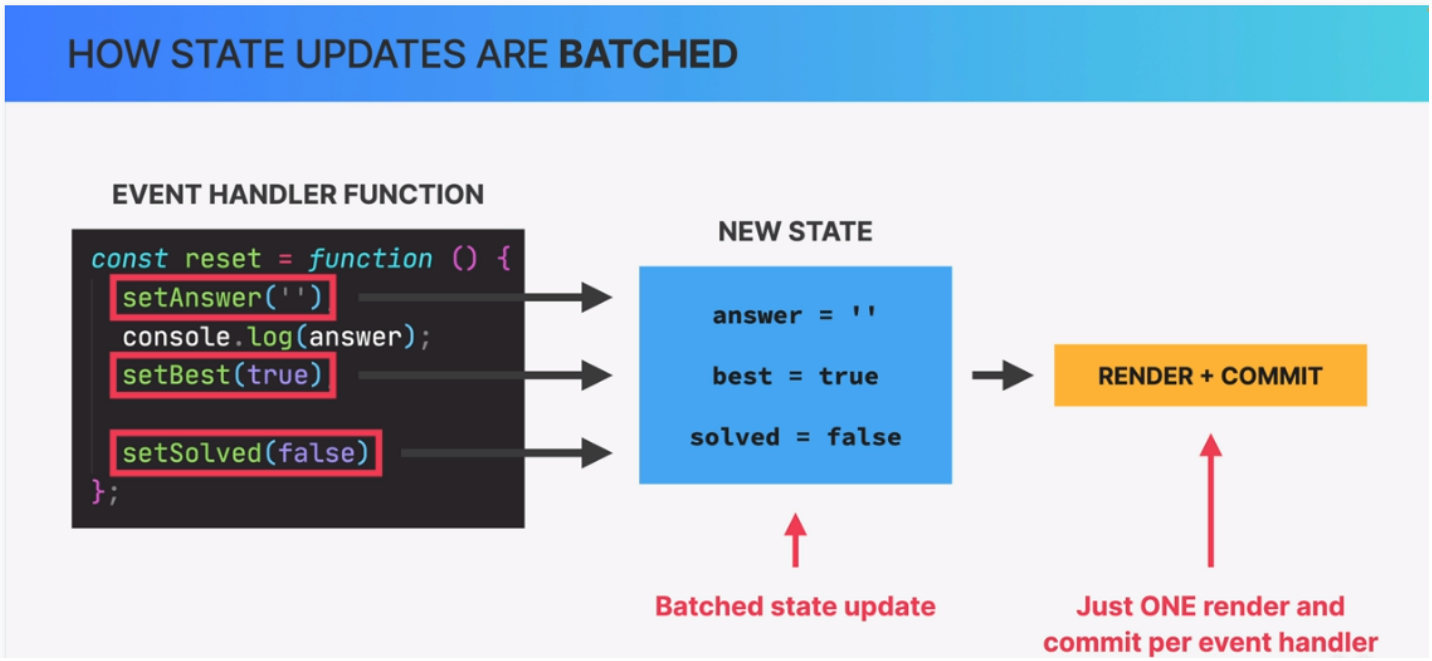
ADDING NEW LIST ITEM

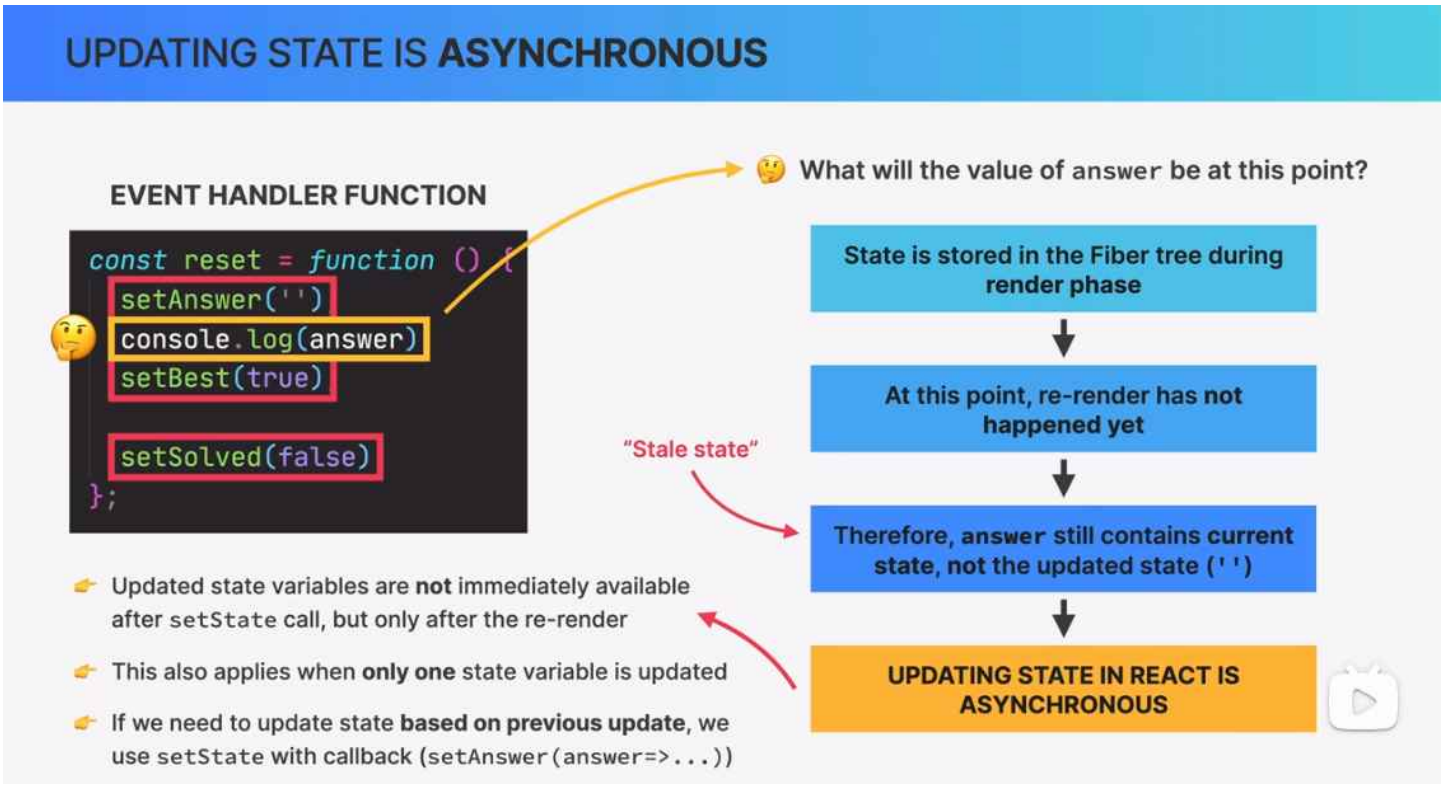
```
<ul>
  <Question key='q0' question={q[0]} />
  <Question key='q1' question={q[1]} />
  <Question key='q2' question={q[2]} />
</ul>
```

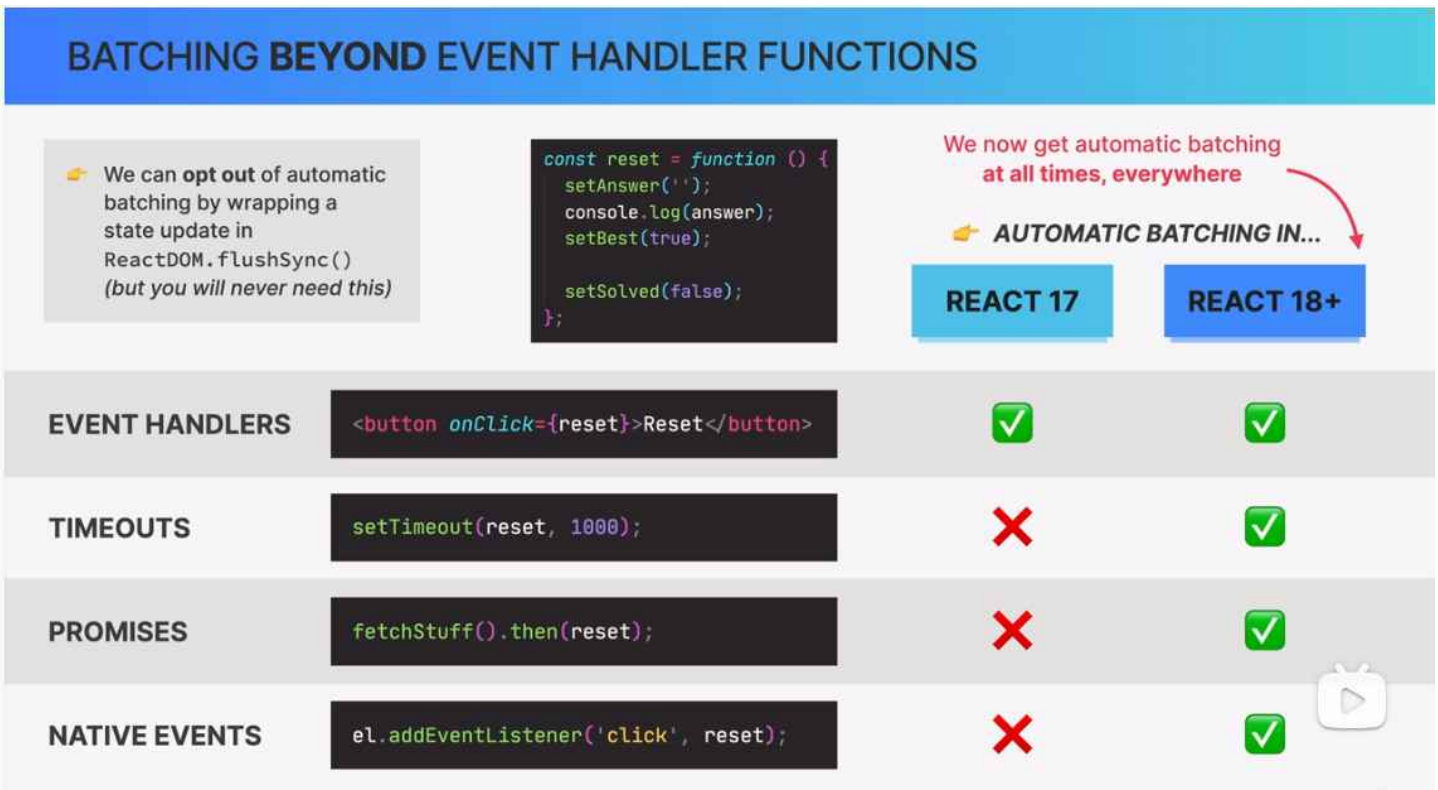
- **Different position in the tree**, but the key **stays the same**, so the elements will be kept in the DOM 👉 **Always use keys!**



- React的更新是 batched，不是分开来多次更新。不是立即更新

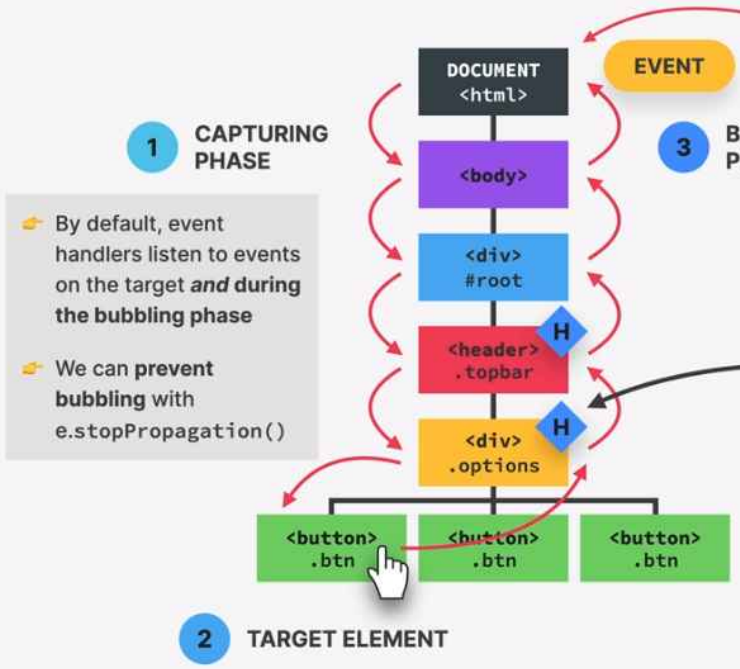



UPDATING STATE IS ASYNCHRONOUS
EVENT HANDLER FUNCTION
const reset = function () {
setAnswer('')
console.log(answer)
setBest(true)
setSolved(false)
};
What will the value of answer be at this point?
State is stored in the Fiber tree during render phase
At this point, re-render has not happened yet
"Stale state"
Therefore, answer still contains current state, not the updated state ('')
UPDATING STATE IN REACT IS ASYNCHRONOUS
Updated state variables are not immediately available after setState call, but only after the re-render
This also applies when only one state variable is updated
If we need to update state based on previous update, we use setState with callback (setAnswer(answer=>...))


BATCHING BEYOND EVENT HANDLER FUNCTIONS
We can opt out of automatic batching by wrapping a state update in ReactDOM.flushSync() (but you will never need this)
const reset = function () {
setAnswer('');
console.log(answer);
setBest(true);
setSolved(false);
};
We now get automatic batching at all times, everywhere
AUTOMATIC BATCHING IN...
REACT 17
REACT 18+
EVENT HANDLERS
<button onClick={reset}>Reset</button>
TIMEOUTS
setTimeout(reset, 1000);
PROMISES
fetchStuff().then(reset);
NATIVE EVENTS
el.addEventListener('click', reset);

- 事件处理

DOM REFRESHER: EVENT PROPAGATION AND DELEGATION



- By default, event handlers listen to events on the target ***and*** **during the bubbling phase**
- We can **prevent bubbling** with `e.stopPropagation()`

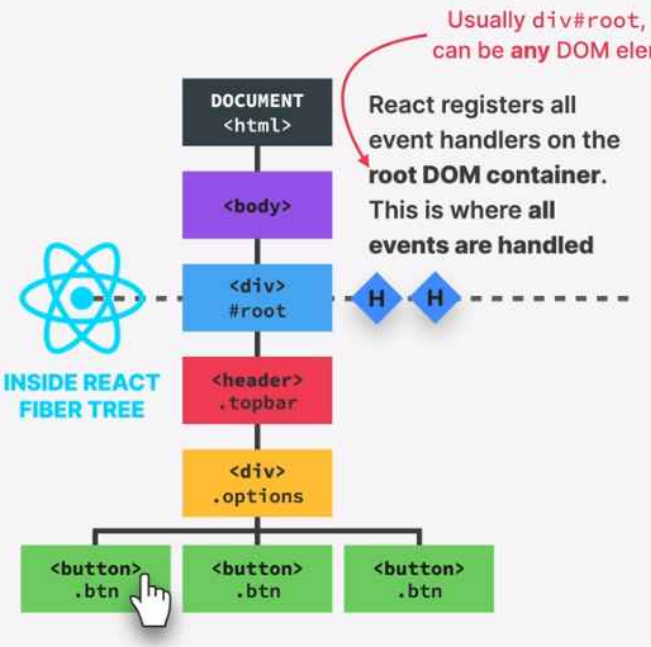
EVENT DELEGATION

- Handling events for multiple elements centrally in **one single parent element**
- Better for performance and memory, as it needs only **one handler function**

1. Add handler to **parent** (`.options`)
2. Check for **target** element (`e.target`)
3. If **target** is one of the `<button>`s, handle the event

👋 *Very common in vanilla JS apps, but not so much in React apps*

HOW **REACT** HANDLES EVENTS



React registers all event handlers on the **root DOM container**. This is where **all events are handled**

WHEN WE ATTACH AN EVENT HANDLER...

```
<button
  className="btn"
  onClick={() => setLoading(true)}
/>
```

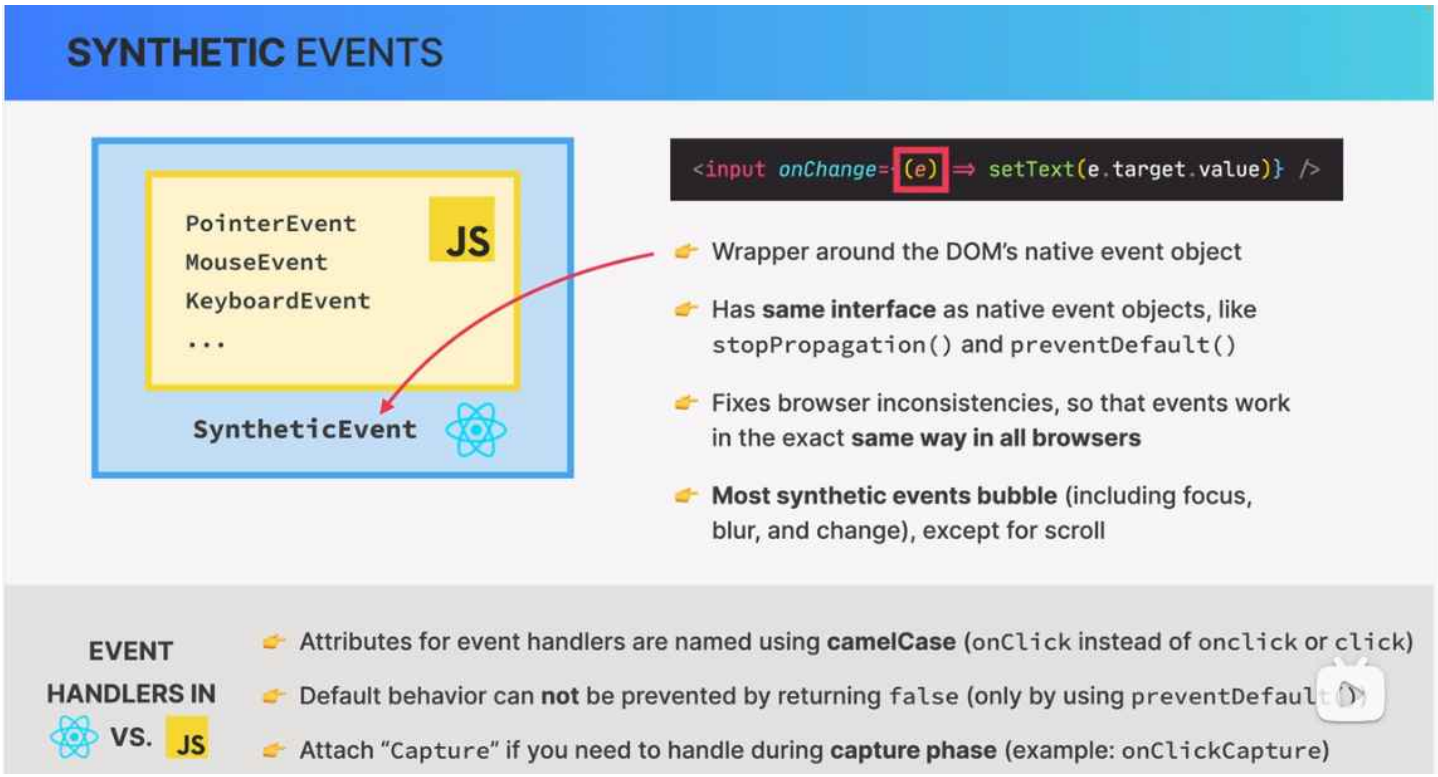
```
document
  .querySelector('.btn')
  .addEventListener(
    'click',
    () => setLoading(true)
  );
```

⛔ **... WHAT APPEARS TO BE HAPPENING**

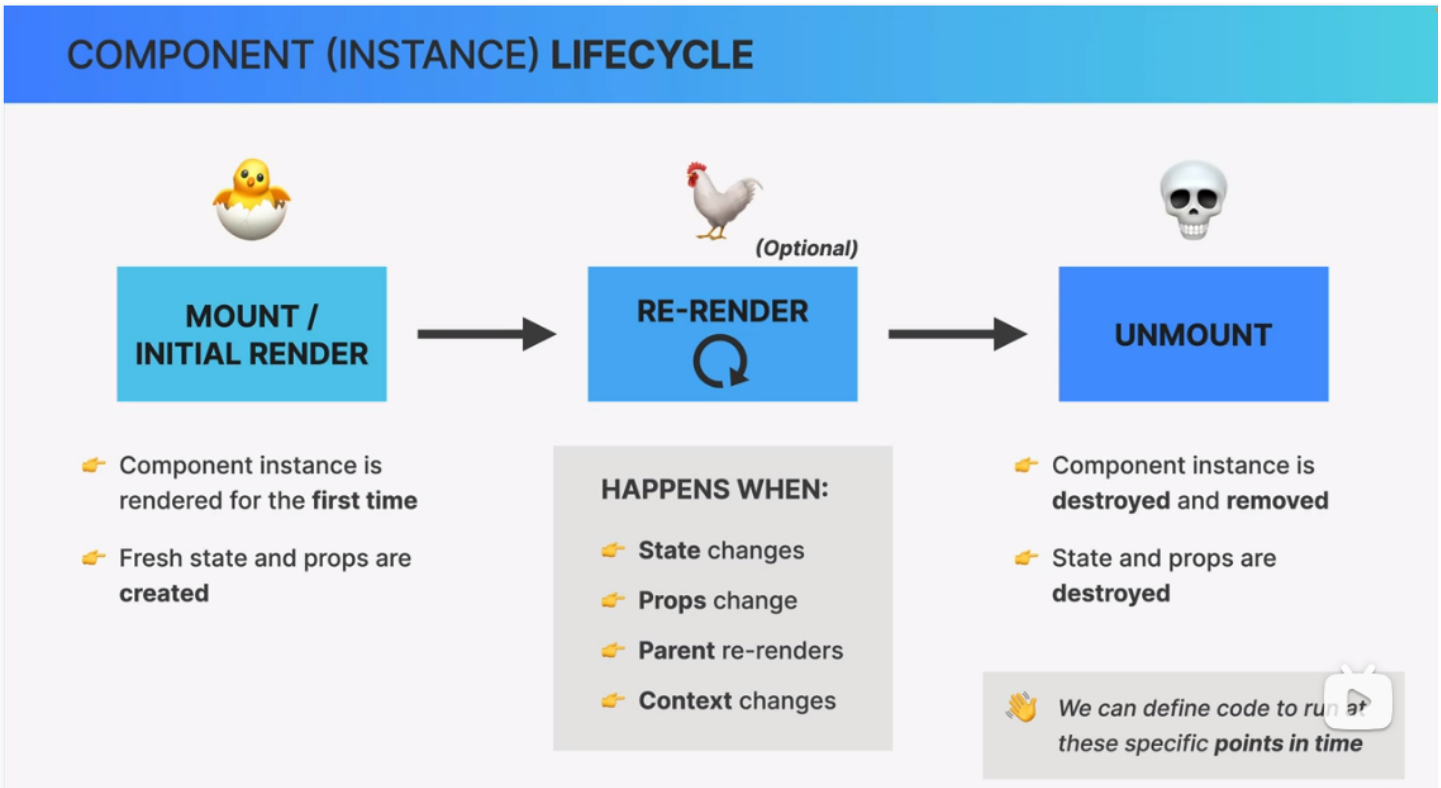
```
document
  .querySelector('#root')
  .addEventListener(
    'click',
    () => setLoading(true)
  );
```

✅ **... WHAT ACTUALLY HAPPENS INTERNALLY**

- Behind the scenes, React performs **event delegation** for all events in our applications



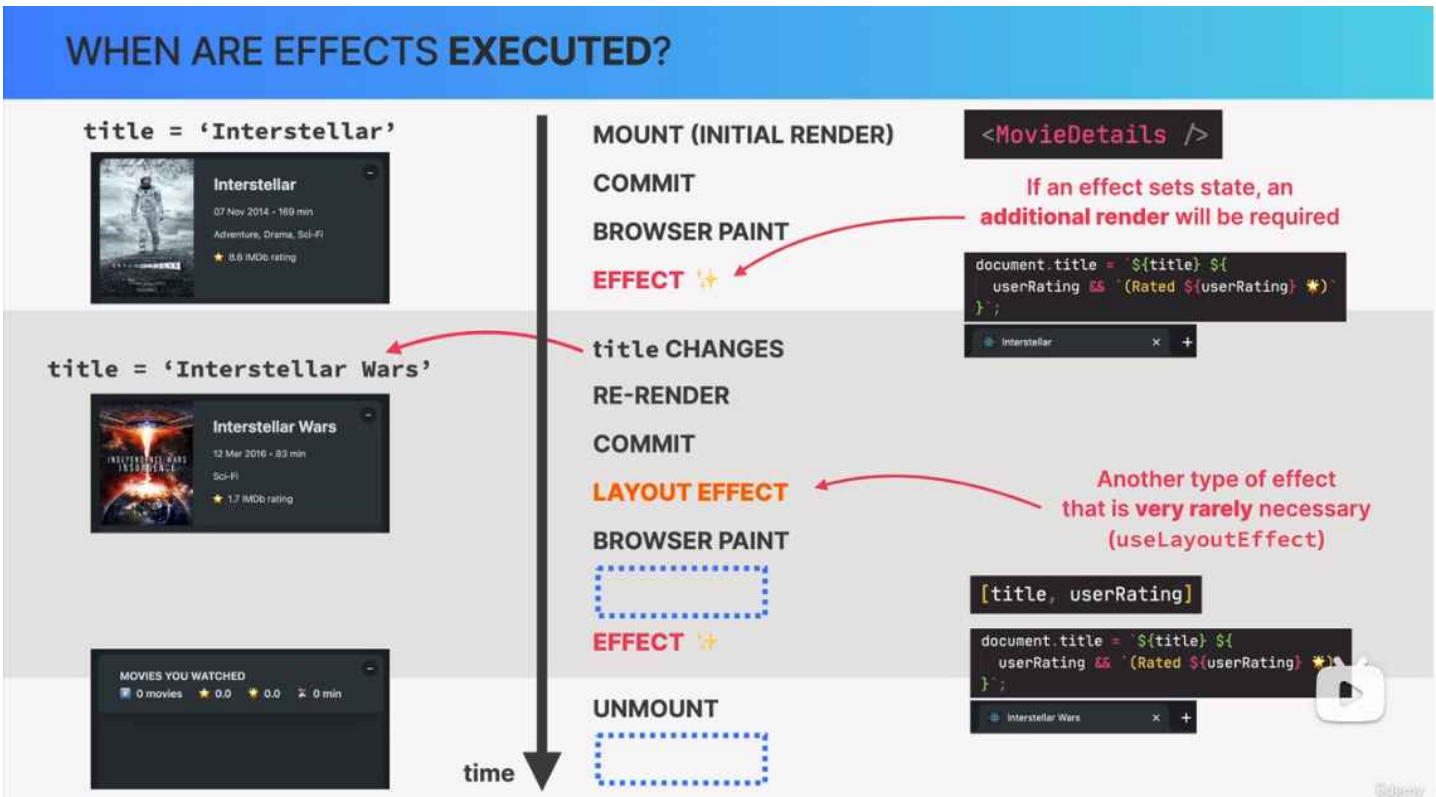
- 生命周期

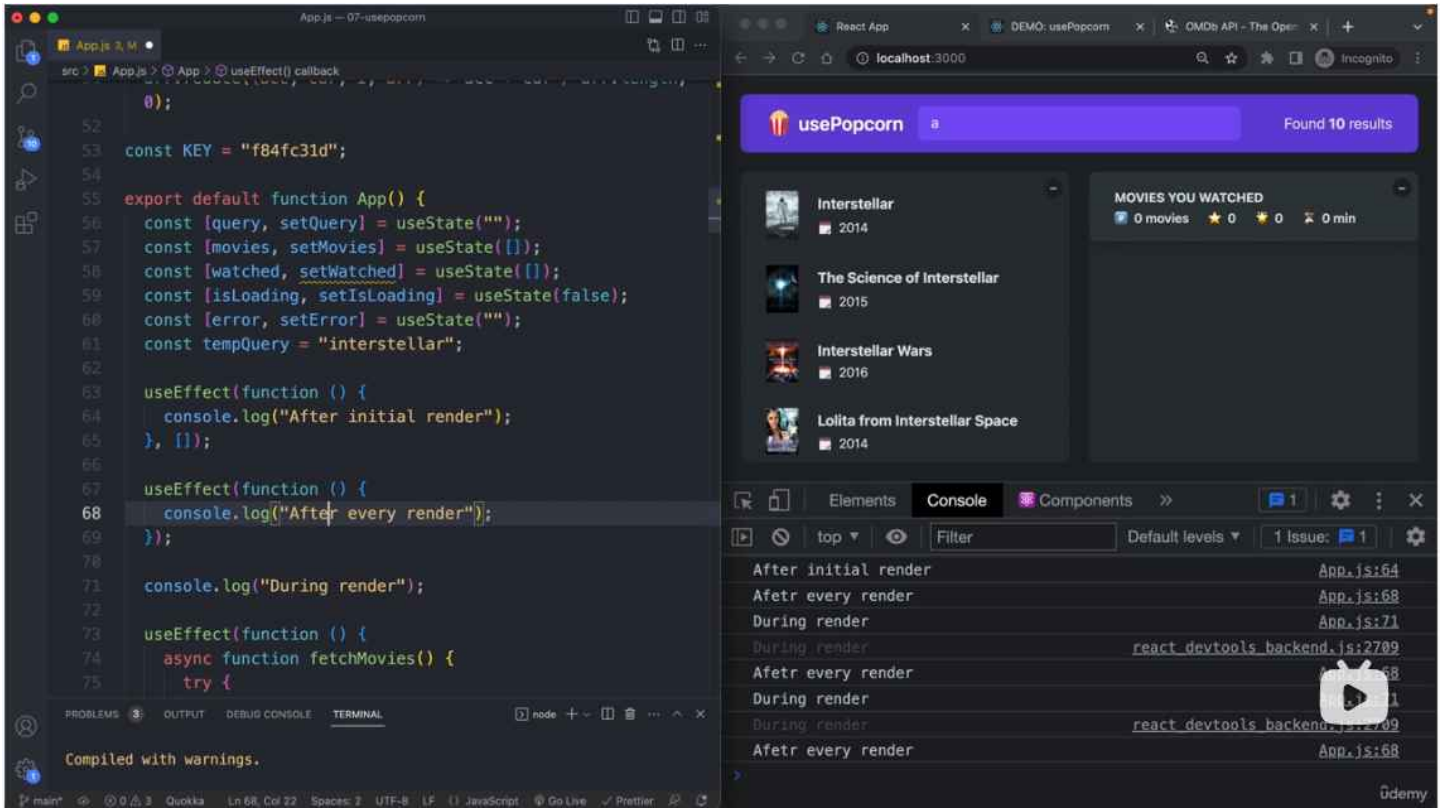


- useEffect的第二个参数设置成 `[]` 表示是initial render的时候触发。在其中使用 await需要先定义一个async函数

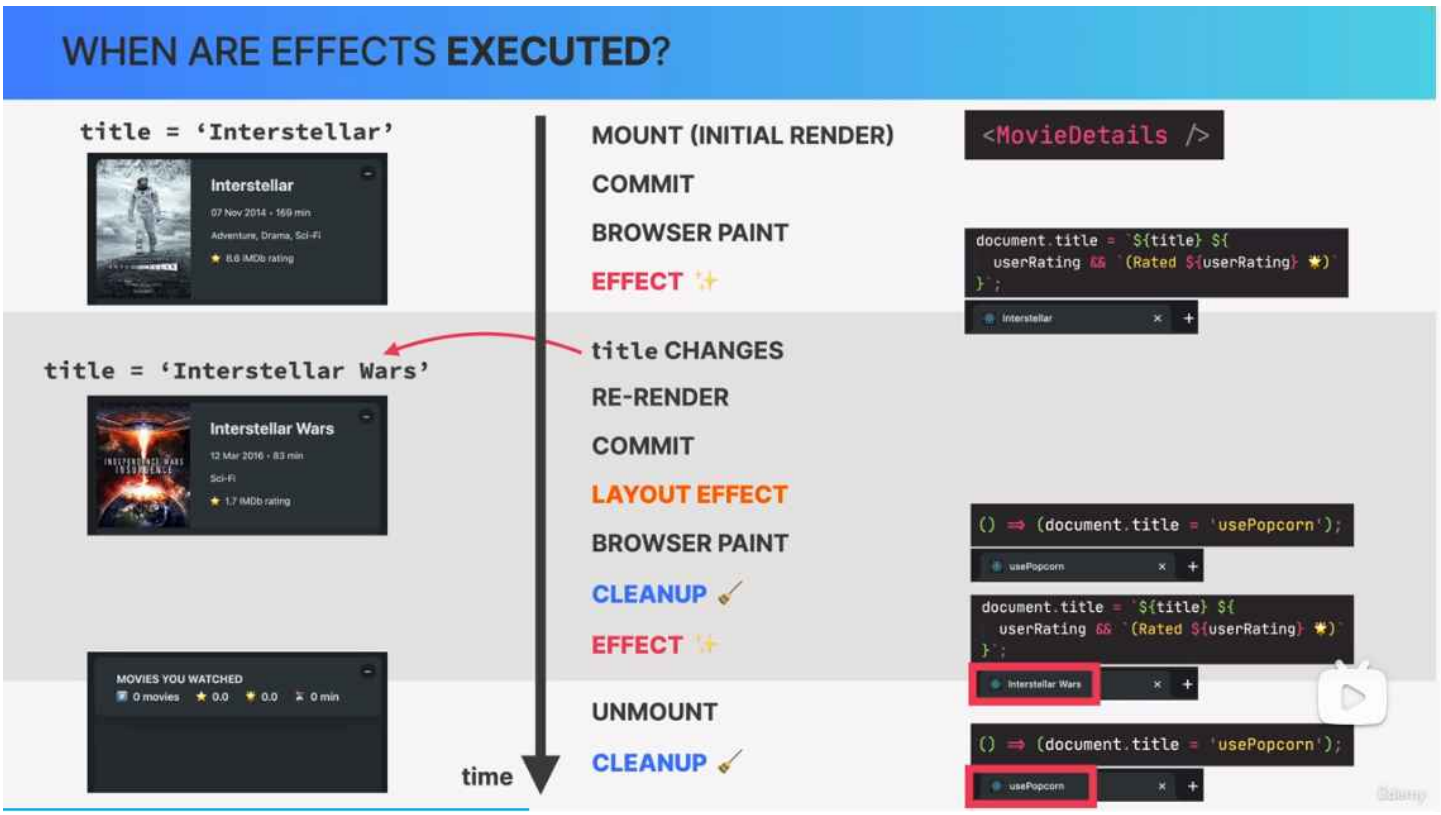
```
useEffect(function () {
  async function fetchMovies() {
    const res = await fetch(
      `http://www.omdbapi.com/?apikey=${KEY}&s=${query}`
    );
    const data = await res.json();
    setMovies(data.Search);
    console.log(data.Search);
  }
  fetchMovies();
}, []);
```

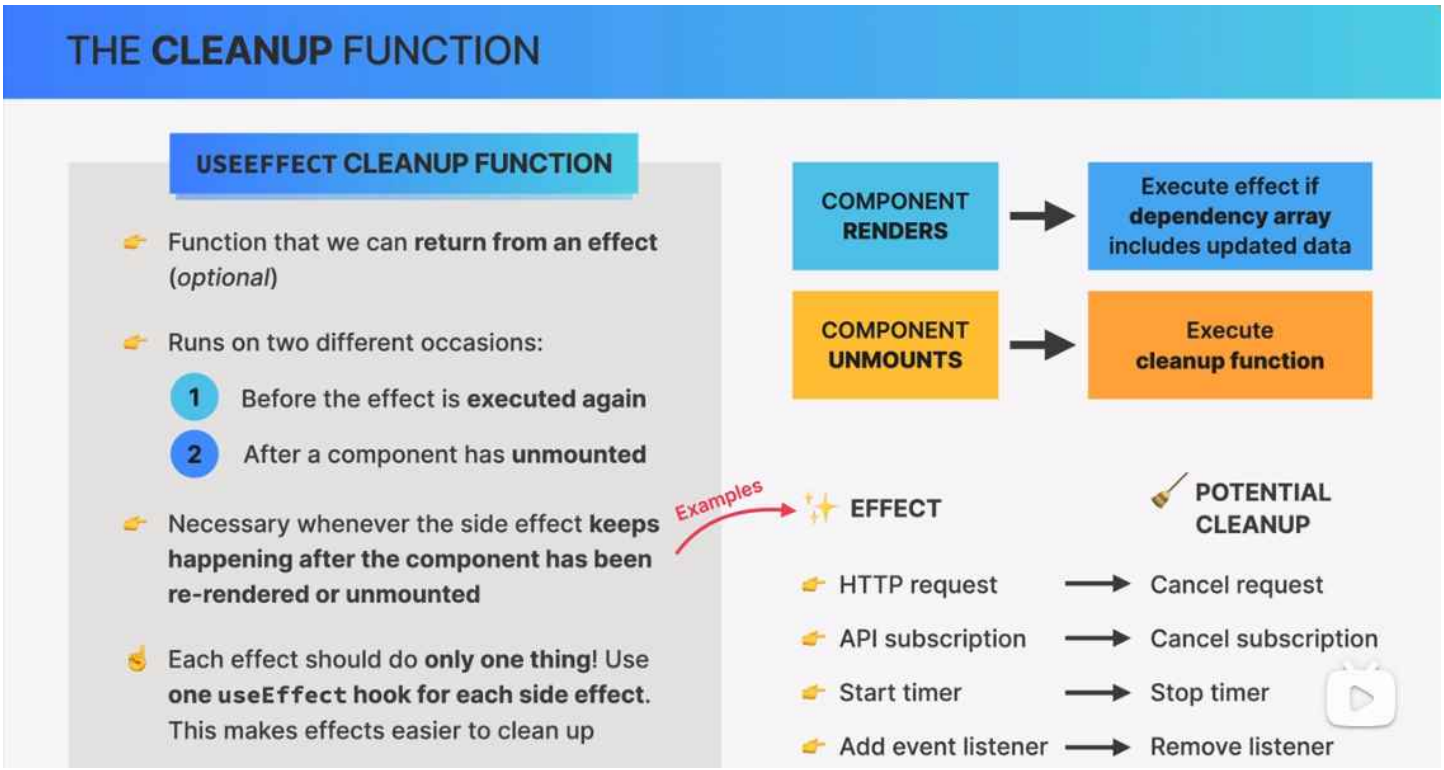
- 生命周期时间线

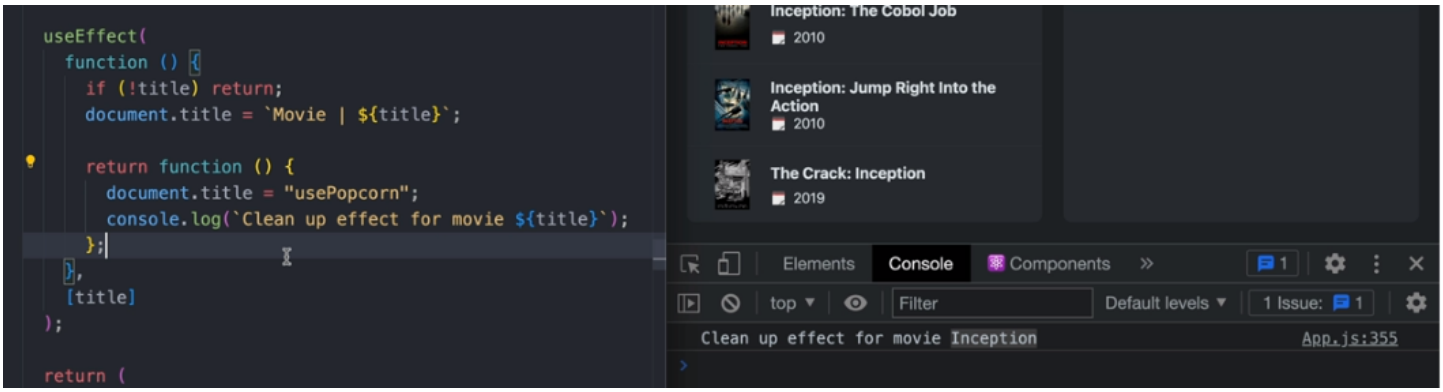




- Cleanup函数：方法在useEffect的返回值里加一个函数，cleanup函数会记录下当时调用的时候变量值







fetch避免race的方法（即每次新发一个就暂停上一个），但是注意abort是error所以如果用try catch需要将这种情况排除

```
1 const controller = new AbortController();
2 await fetch("", {signal: controller.signal});
3
4 return function() {
5     controller.abort();
6 }
```

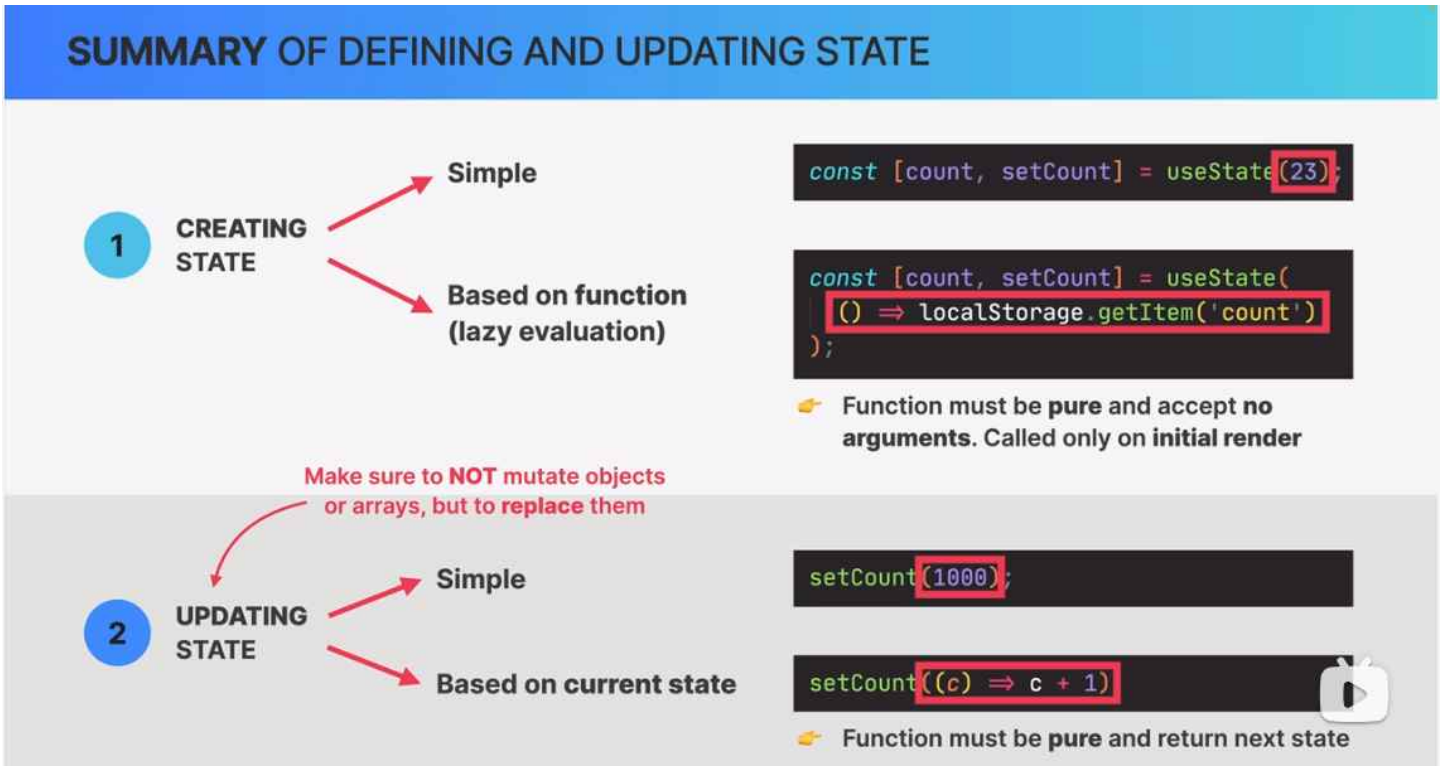
- 键盘回调

```
useEffect(
  function () {
    function callback(e) {
      if (e.code === "Escape") {
        onCloseMovie();
        console.log("CLOSING");
      }
    }

    document.addEventListener("keydown", callback);

    return function () {
      document.removeEventListener("keydown", callback);
    };
  },
  [onCloseMovie]
);
```

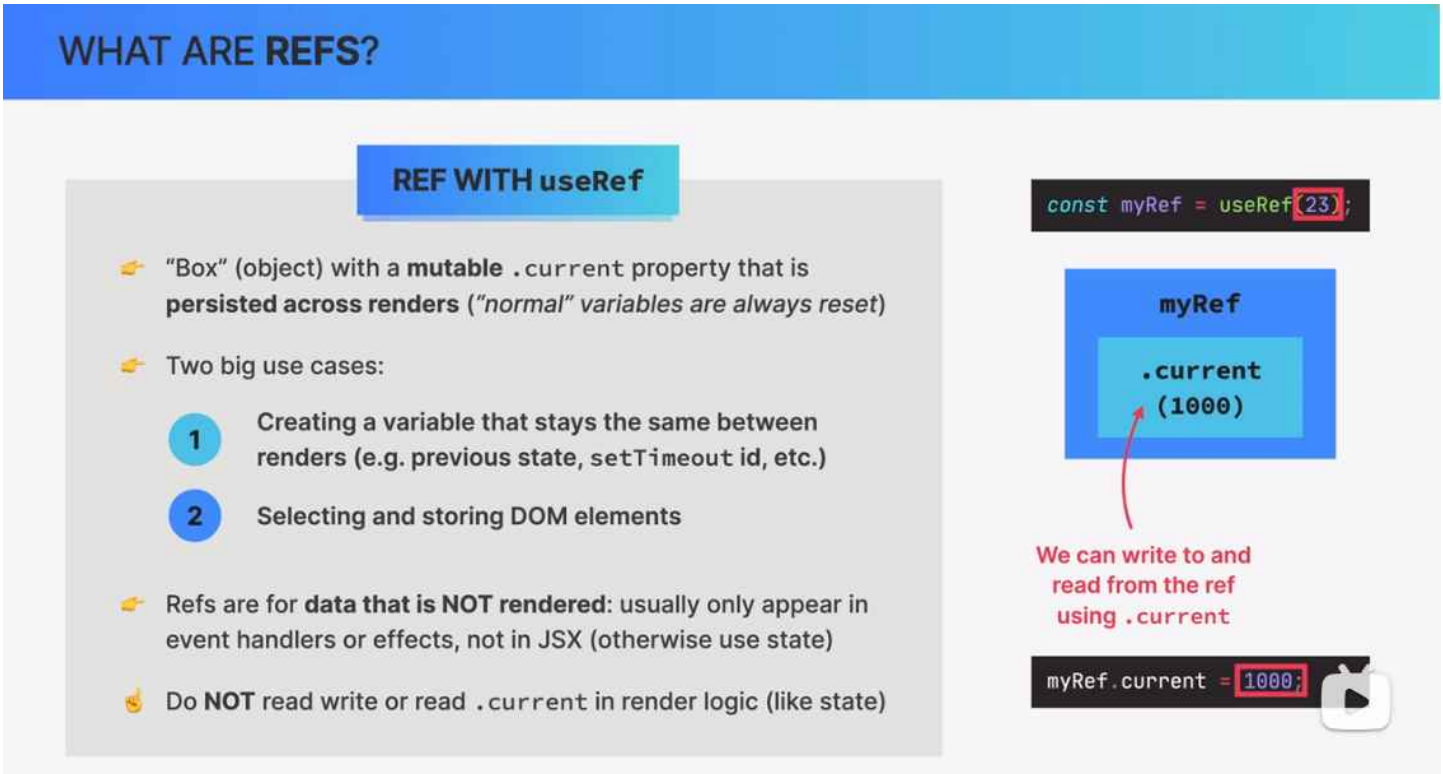
- hook的顺序不能被破坏，所以不能出现 early return（后面还有hook的定义直接return）或者 if下面定义hook
- 持久化保存：`localStorage.setItem("key", JSON.stringify())`。初始化读取可以使用如下的方式：

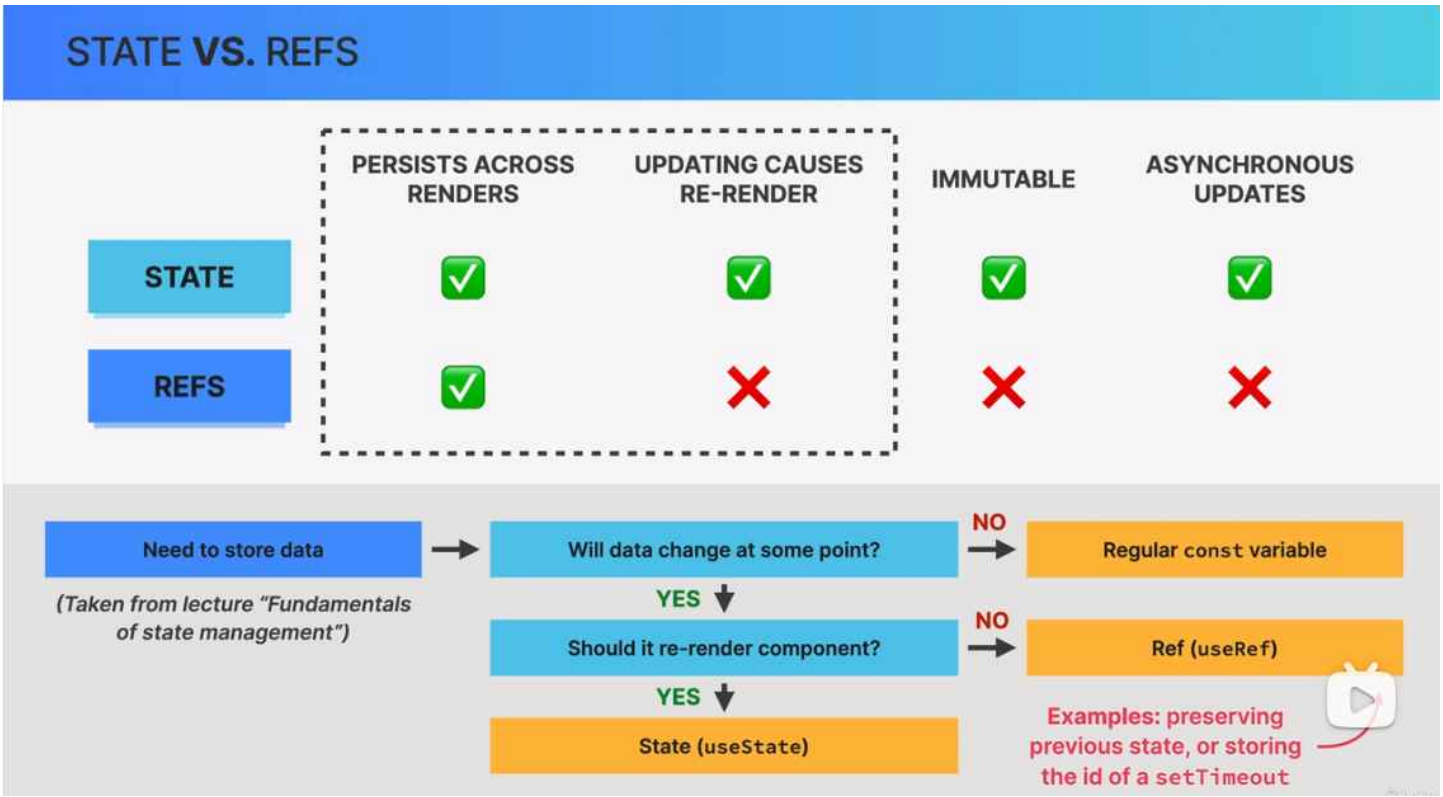
```
// const [watched, setWatched] = useState([]);
const [watched, setWatched] = useState(function () {
  const storedValue = localStorage.getItem("watched");
  return JSON.parse(storedValue);
});
```

- useState总结



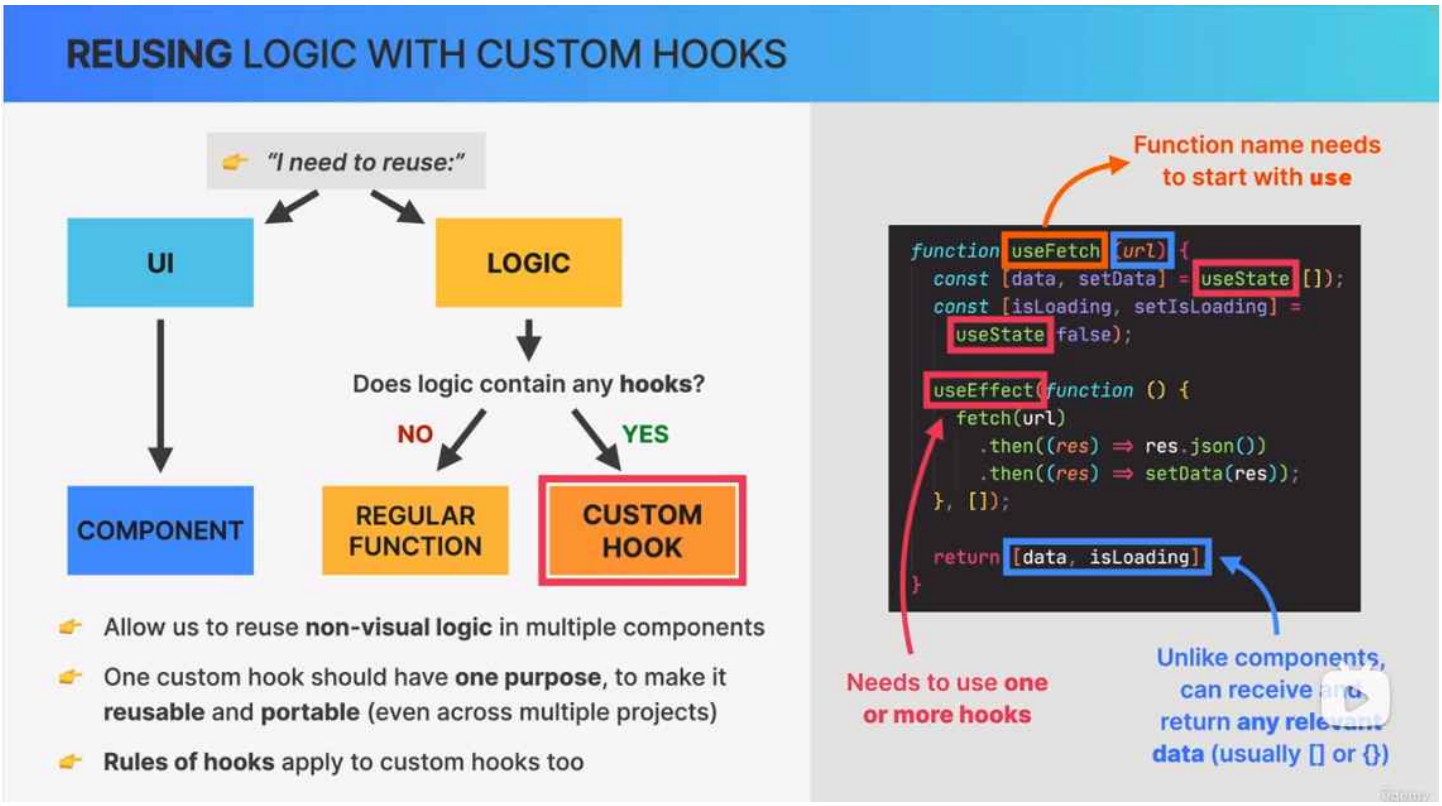
- useRef，用来选中某个元素。我们不能在 render login 中修改它，所以一般是在 useEffect 或者是 handler 中修改





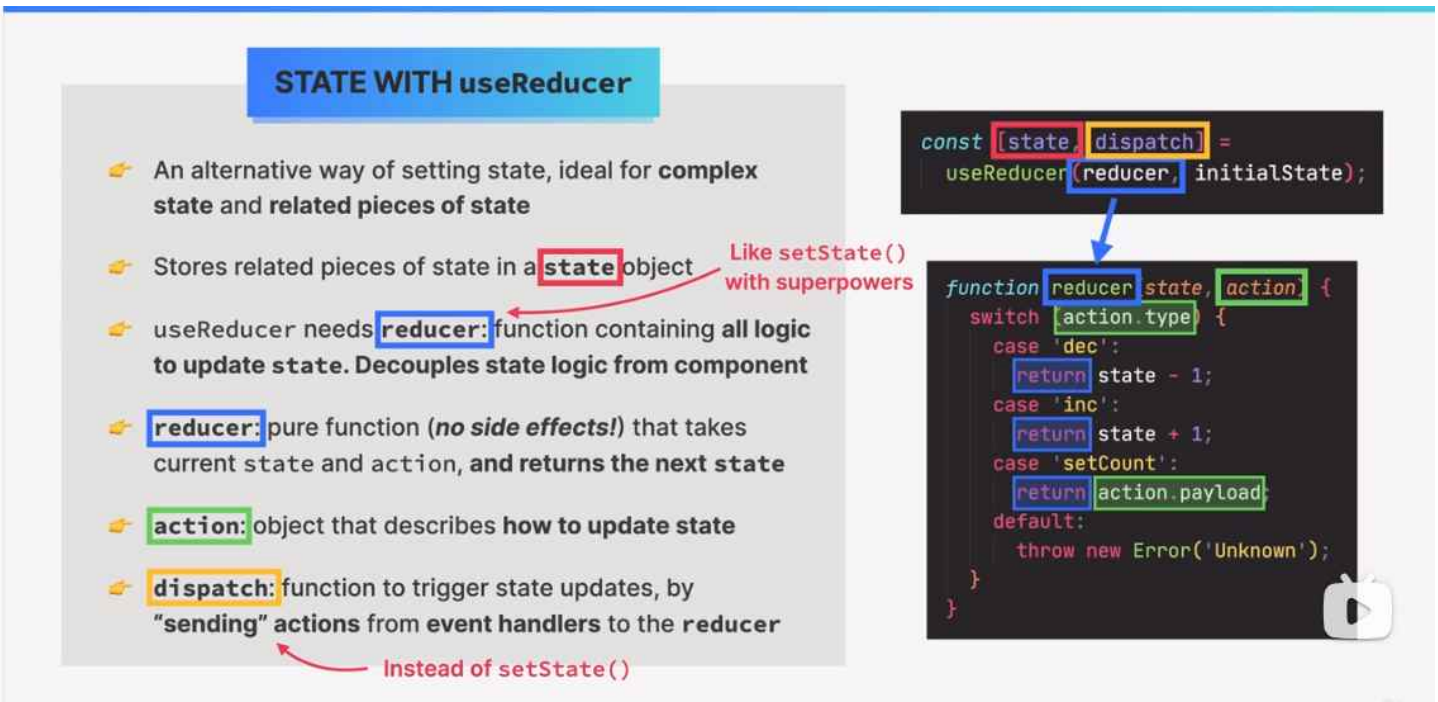

```
useEffect(
  function () {
    function callback(e) {
      if (document.activeElement === inputEl.current)
      return;

      if (e.code === "Enter") {
        inputEl.current.focus();
        setQuery("");
      }
    }

    document.addEventListener("keydown", callback);
    return () => document.addEventListener("keydown",
    callback);
  },
  [setQuery]
);
```

- 自定义 hook



高级

- useReducer，解决多state的情况



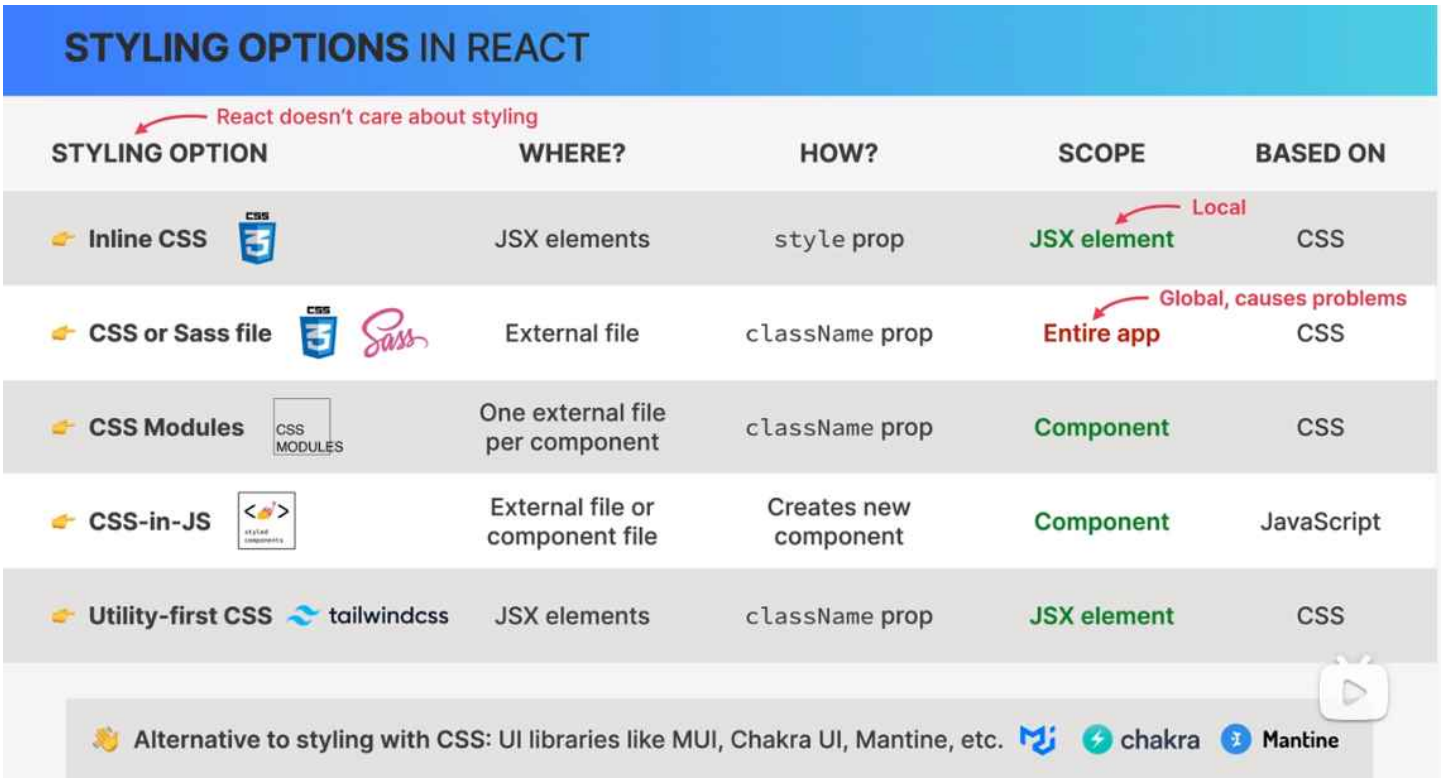
- 定时函数清除

```jsx
import { useEffect } from "react";

function Timer({ dispatch, secondsRemaining }) {
  useEffect(
    function () {
      const id = setInterval(function () {
        dispatch({ type: "tick" });
      }, 1000);

      return () => clearInterval(id);
    },
    [dispatch]
  );

  return <div classNam="timer">{secondsRemaining}</div>;
}

export default Timer;
```

- 定义Router，页面切换不要使用 `a` ，使用 `Link` 或者 `NavLink` ，后者多了一个 `active` 的 clas

```jsx
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Homepage />} />
        <Route path="product" element={<Product />} />
        <Route path="pricing" element={<Pricing />} />
        <Route path="*" element={<PageNotFound />} />
      </Routes>
    </BrowserRouter>
  );
}
```

- CSS的不同类型



- CSS Module的方法：定义 `xxx.module.css`，然后 `import styles from`，`className={styles.xxx}`。全局化

```
:global(.test) {
  background-color: red;
}
```

- Index就是默认路由，子路由的元素会填充父元素的 `<Outlet />`

```jsx
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route index element={<Homepage />} />
        <Route path="product" element={<Product />} />
        <Route path="pricing" element={<Pricing />} />
        <Route path="/login" element={<Login />} />
        <Route path="app" element={<AppLayout />}>
          <Route index element={<p>List of cities</p>} />
          <Route path="cities" element={<p>List of cities</
          p>} />
          <Route path="countries" element={<p>Countries</
          p>} />
          <Route path="form" element={<p>Form</p>} />
        </Route>
        <Route path="*" element={<PageNotFound />} />
      </Routes>
    </BrowserRouter>
  );
```

- `useParams` 得到URL中的路径参数。query参数用的是 `useSearchParams`

```jsx
<Route path="cities/:id" element={<City />} />
```

```jsx
const [searchParams, setSearchParams] = useSearchParams
();
const lat = searchParams.get("lat");
const lng = searchParams.get("lng");
```

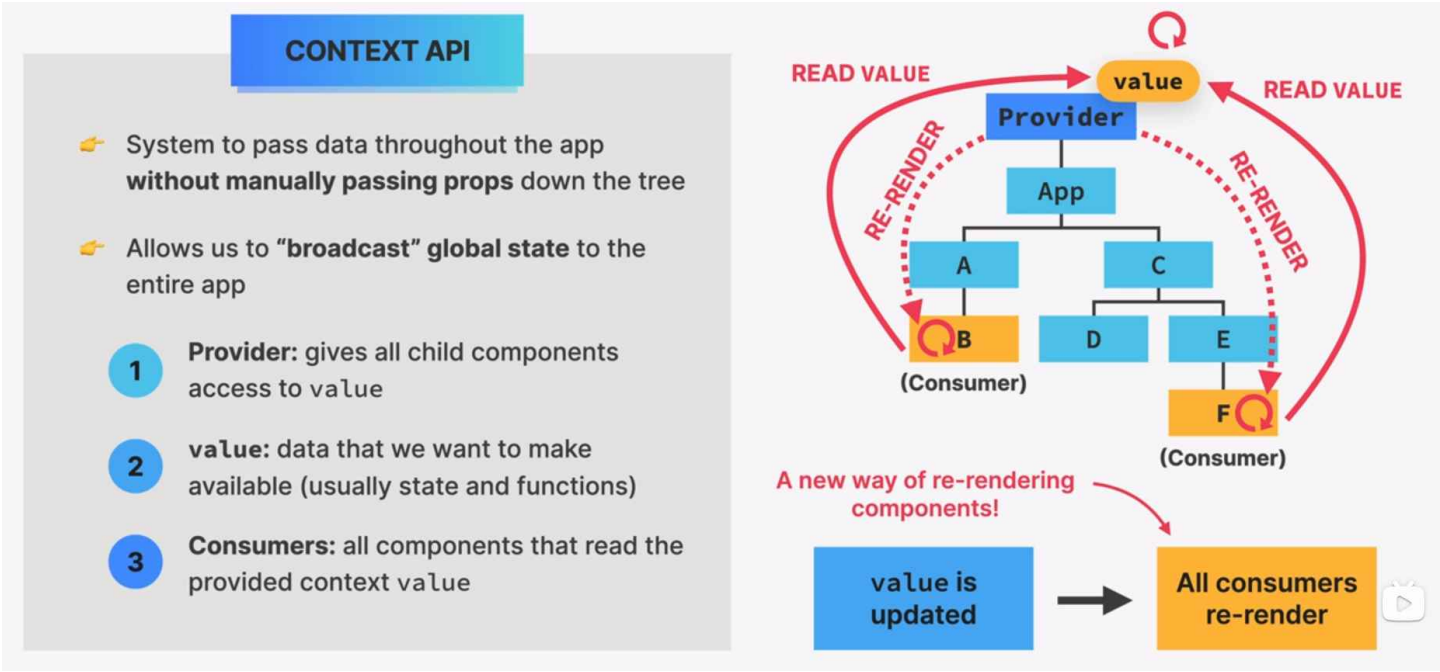
- `useNavigate` 可以直接跳转，后面可以跟地址或者数字， `-1` 表示回退一次。下面prevent的原因是默认会submit然后刷新页面

```jsx
<Button
  type="back"
  onClick={(e) => {
    e.preventDefault();
    navigate(-1);
  }}
>
  &larr; Back
</Button>
```

- 设置默认链接

```jsx
<Route index element={<Navigate replace
to="cities" />} />
```

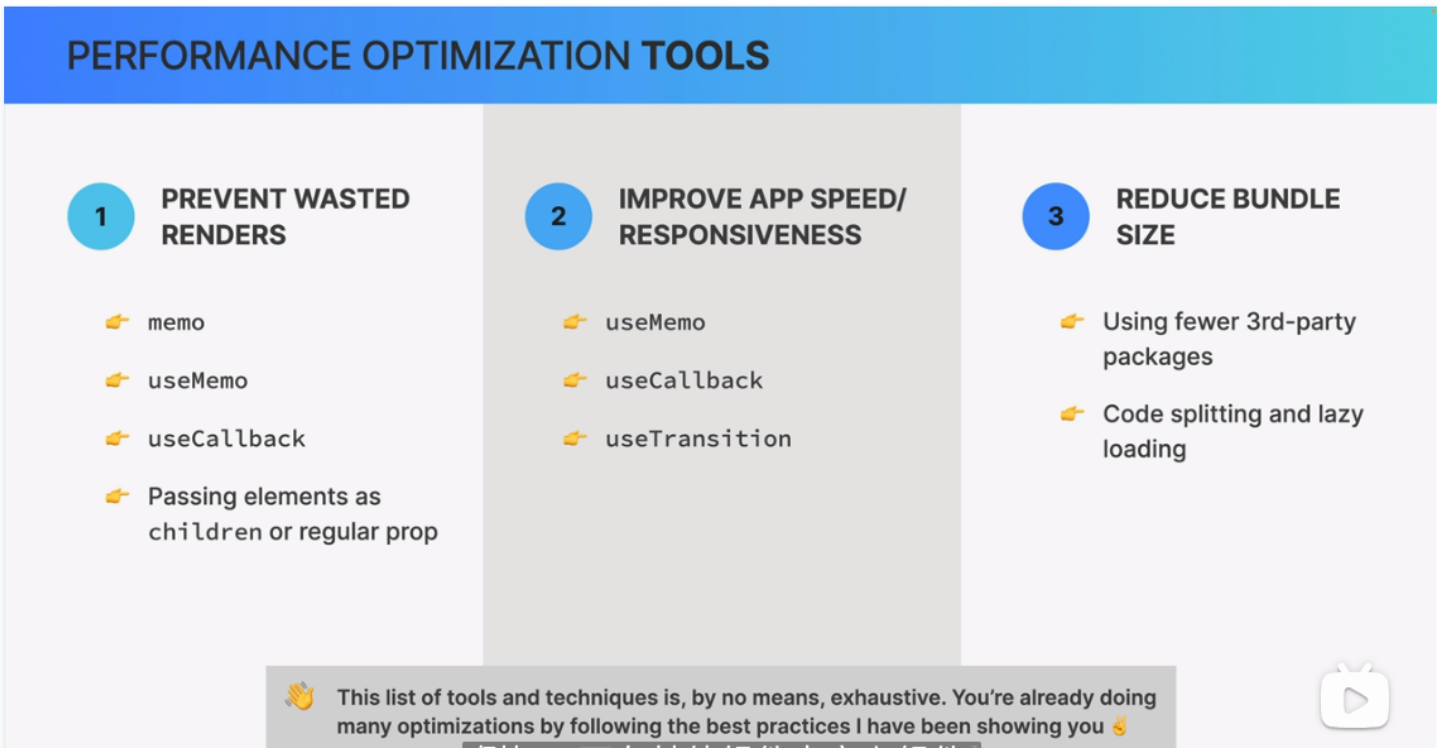
- Context



- 路由保护

```jsx
function ProtectedRoute({ children }) {
  const { isAuthenticated } = useAuth();
  const navigate = useNavigate();

  useEffect(
    function () {
      if (!isAuthenticated) navigate("/");
    },
    [isAuthenticated, navigate]
  );

  return children;
}

export default ProtectedRoute;
```

- 优化的方式



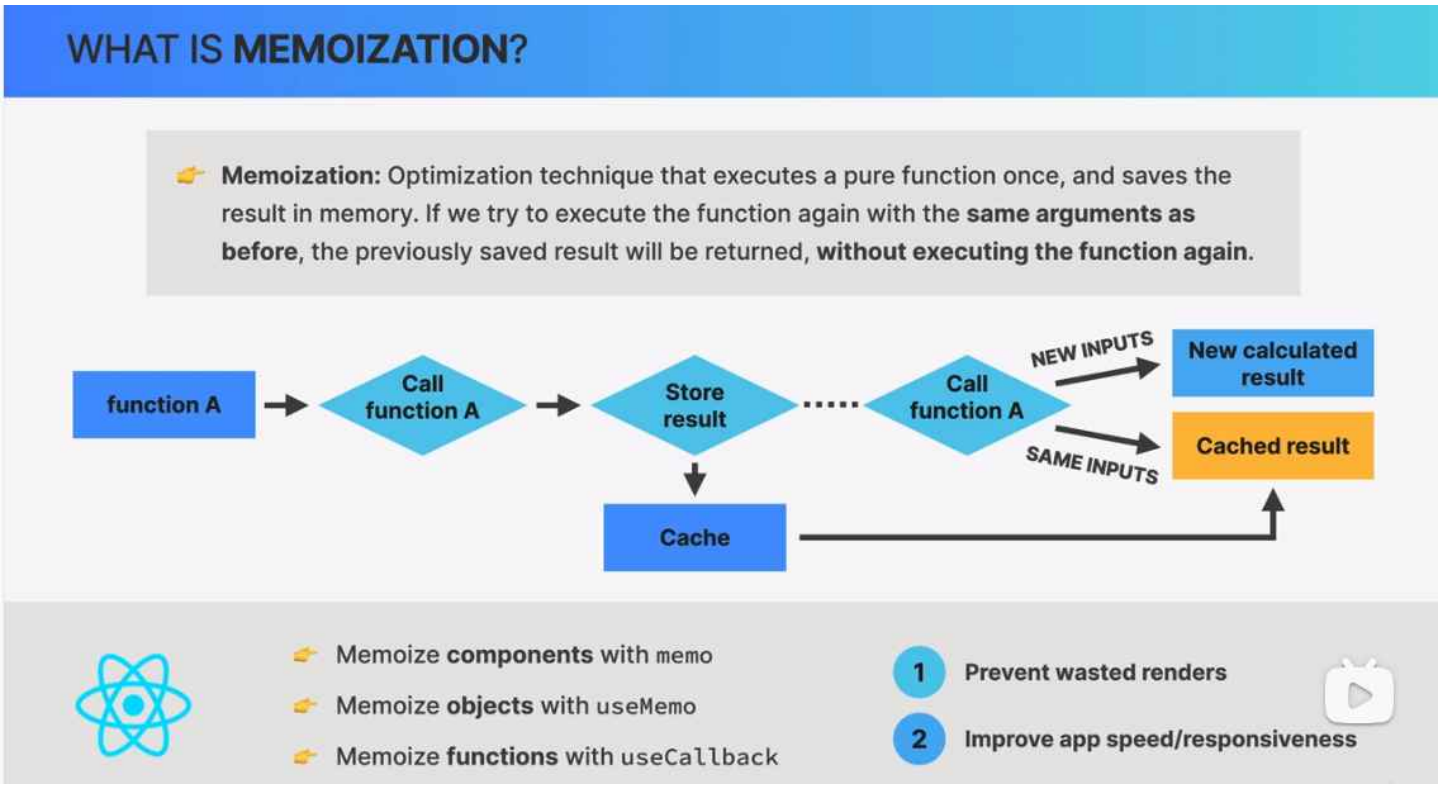
- 优化方法一：将不依赖state的组件变成children

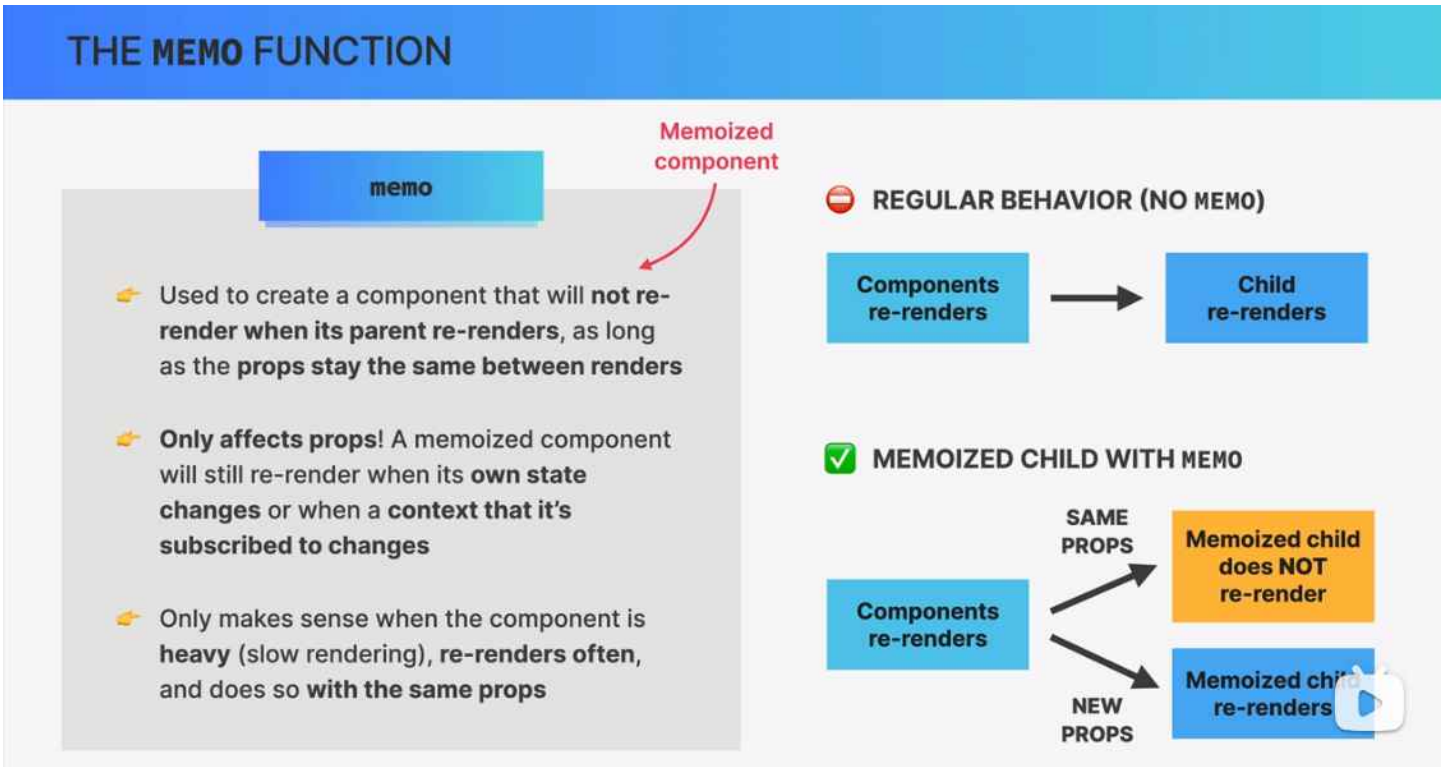

export default function Test() {
// const [count, setCount] = useState(0);
// return (
//
// <h1>Slow counter?!?</h1>
// <button onClick={() => setCount
((c) => c + 1)}>Increase: {count}</
button>
// <SlowComponent />
//
//);
return (

<h1>Slow counter?!?</h1>
<Counter>
<SlowComponent />
</Counter>

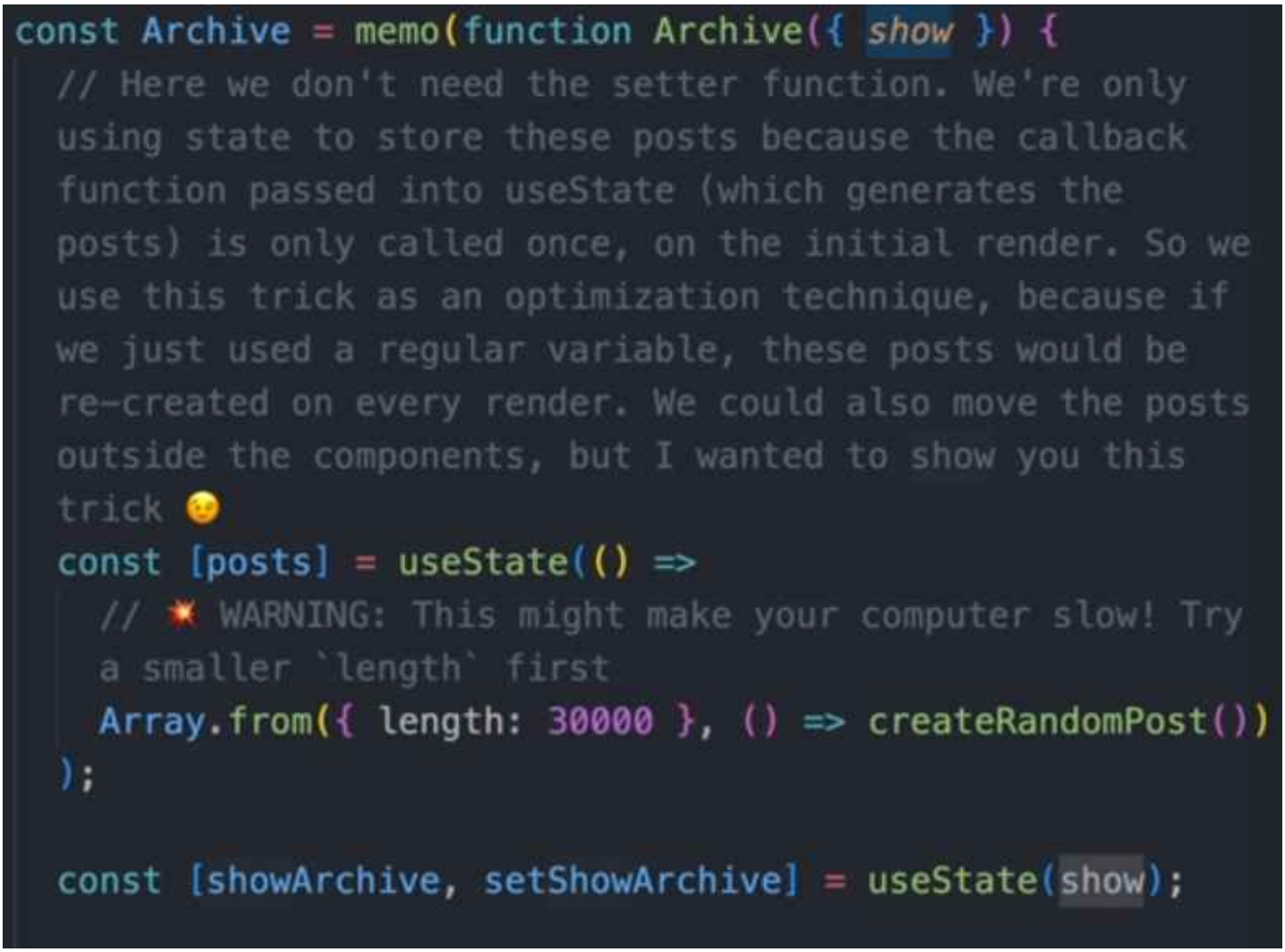
);
}

- Memo





`memo` 使用方法：给函数加一个wrapper，但是我们发现如果这个show是一个object那么就没用了，因为在js里object不相等



```
const Archive = memo(function Archive({ show }) {
  // Here we don't need the setter function. We're only
  using state to store these posts because the callback
  function passed into useState (which generates the
  posts) is only called once, on the initial render. So we
  use this trick as an optimization technique, because if
  we just used a regular variable, these posts would be
  re-created on every render. We could also move the posts
  outside the components, but I wanted to show you this
  trick 😉
  const [posts] = useState(() =>
    // 💥 WARNING: This might make your computer slow! Try
    a smaller `length` first
    Array.from({ length: 30000 }, () => createRandomPost())
  );

  const [showArchive, setShowArchive] = useState(show);
```

解决object方法：

AN **ISSUE** WITH MEMO

In React, everything is **re-created on every render** (including objects and functions)

↓

In JavaScript, two objects or functions that look the same, **are actually different** (`{} != {}`)

THEREFORE ↓

If objects or functions are passed as props, the child component will always see them as **new props on each re-render**

↓

If props are different between re-renders, *memo will not work*

SOLUTION ↓

We need to memoize objects and functions, to make them stable (preserve) between re-renders (`memoized {} == memoized {}`)

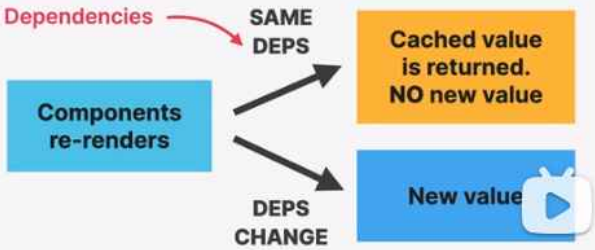
TWO NEW HOOKS: **USEMEMO** AND **USECALLBACK**

`useMemo` AND `useCallback`

- Used to memoize values (**`useMemo`**) and functions (**`useCallback`**) **between renders**
- Values passed into `useMemo` and `useCallback` will be stored in memory ("cached") and **returned in subsequent re-renders, as long as dependencies** ("*inputs*") **stay the same**
- `useMemo` and `useCallback` have a **dependency array** (like `useEffect`): whenever one **dependency changes**, the value will be **re-created**

⛔ REGULAR BEHAVIOR (NO `USEMEMO`)

✅ MEMOIZING A VALUE WITH `USEMEMO`



```
const archiveOptions = useMemo(() => {
  return {
    show: false,
    title: `Post archive in addition to ${posts.length}
    main posts`,
  };
}, [posts.length]);
```

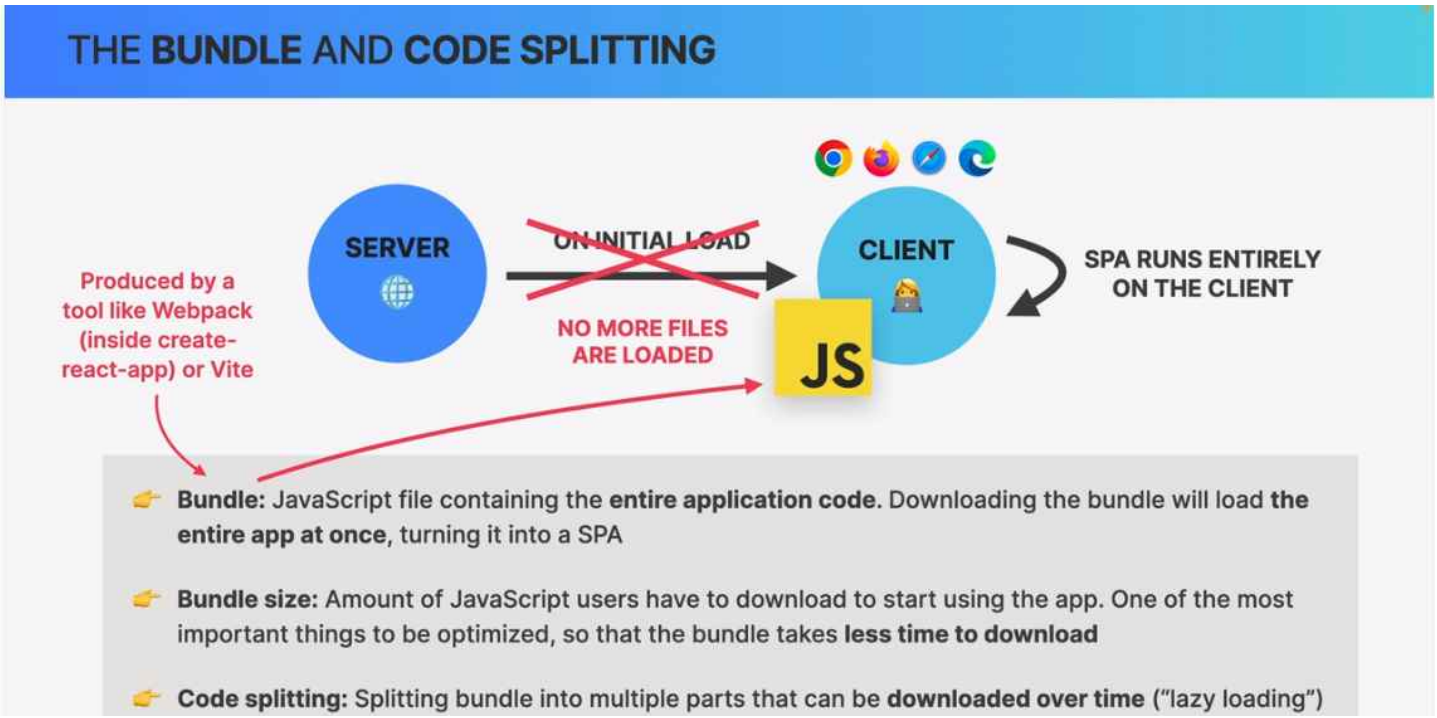
```
const handleAddPost = useCallback(function handleAddPost
(post) {
  setPosts((posts) => [post, ...posts]);
}, []);
```

另外react保证从useState得到的set函数是stable的，也就是相当于useCallback

- context优化，注意最好将context抽取出来加children。然后最好context里面的内容越少越好，否则就要将value useMemo（因为可能会出现provider的父元素刷新导致value这个object发生改变然后然后导致context rerender）

```
const value = useMemo(() => {
  return {
    posts: searchedPosts,
    onAddPost: handleAddPost,
    onClearPosts: handleClearPosts,
    searchQuery,
    setSearchQuery,
  };
}, [searchedPosts, searchQuery]);

return (
  // 2) PROVIDE VALUE TO CHILD COMPONENTS
  <PostContext.Provider value={value}>{children}</
  PostContext.Provider>
);
```

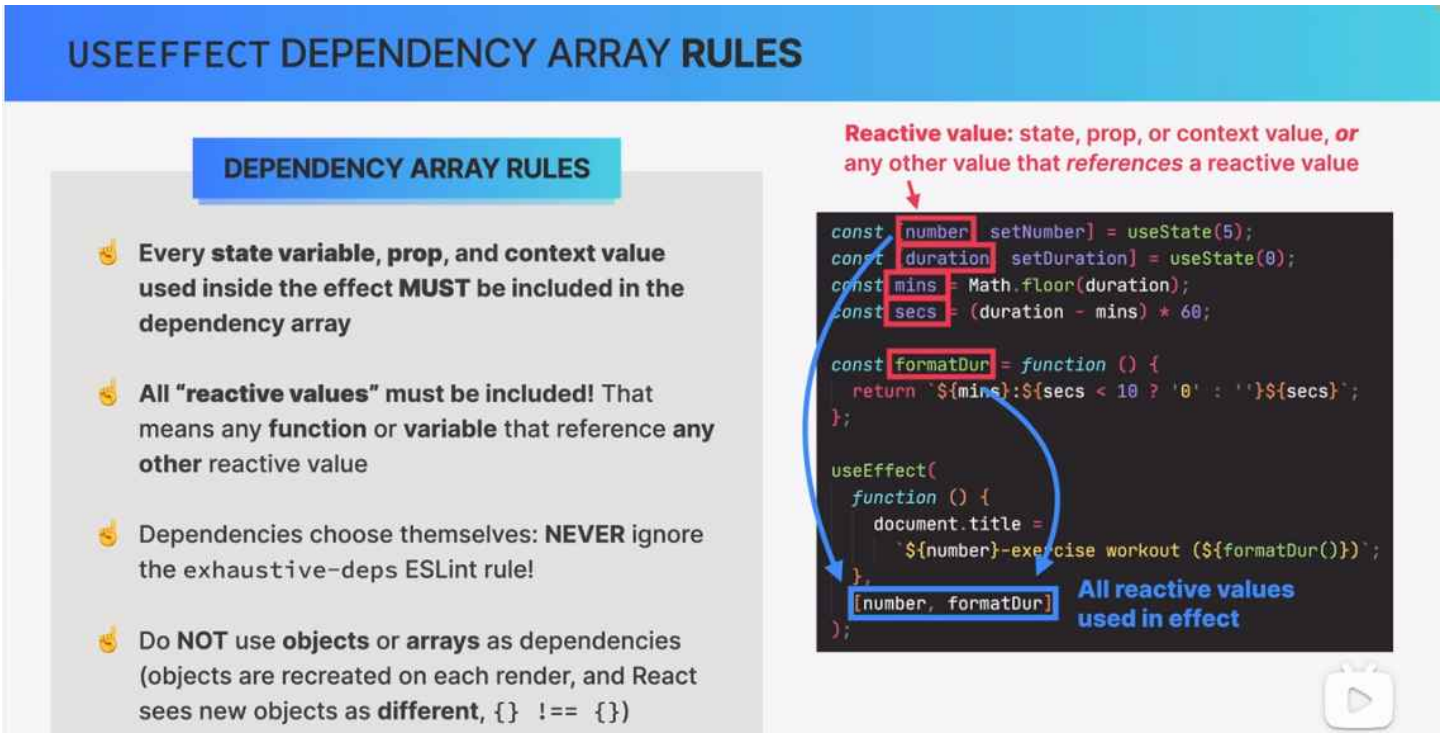
- useEffect里dependency里如果有一个函数，这个函数是另一个组件的，当这个组件被刷新的时候函数被重建然后又会调用useEffect，这时我们需要将这个函数添加到 useCallback 里。
- 优化bundle的大小：使用 lazy loading，一般是按照page进行

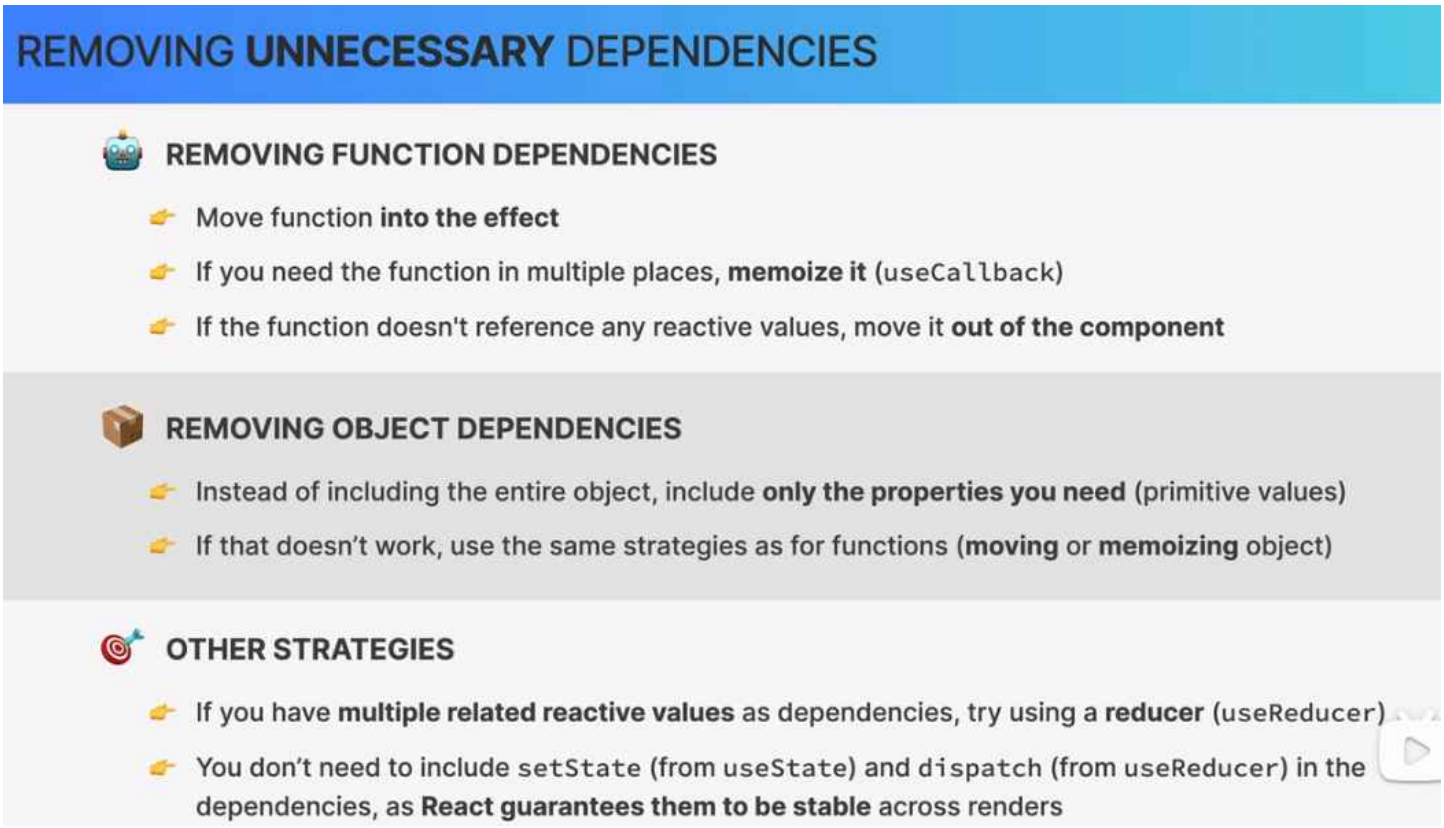


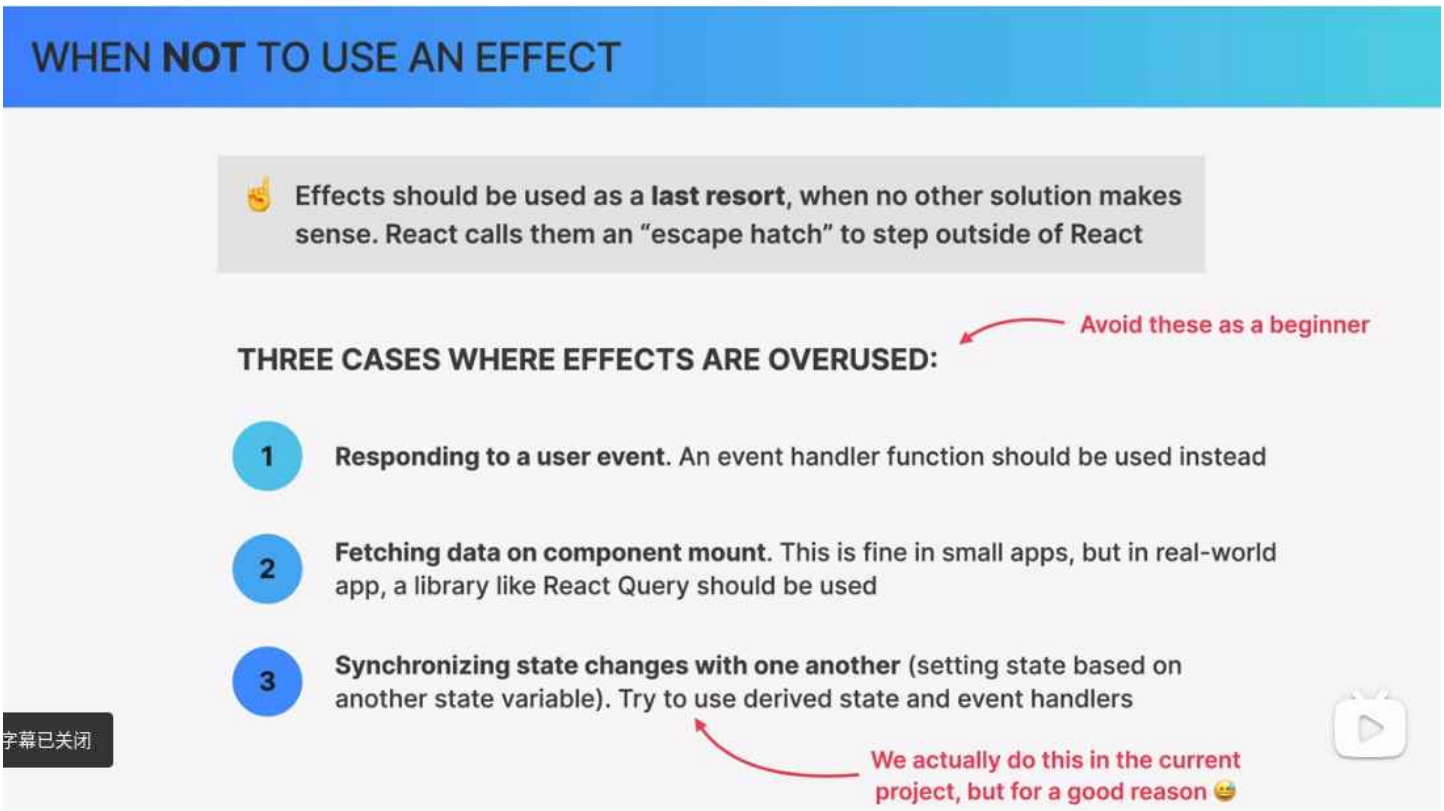

```
const Homepage = lazy(() => import("./pages/Homepage"));
const Product = lazy(() => import("./pages/Product"));
const Pricing = lazy(() => import("./pages/Pricing"));
const Login = lazy(() => import("./pages/Login"));
const AppLayout = lazy(() => import("./pages/AppLayout"));
const PageNotFound = lazy(() => import("./pages/
PageNotFound"));
```

```
<BrowserRouter>
  <Suspense fallback={<SpinnerFullPage />}>
    <Routes>
```

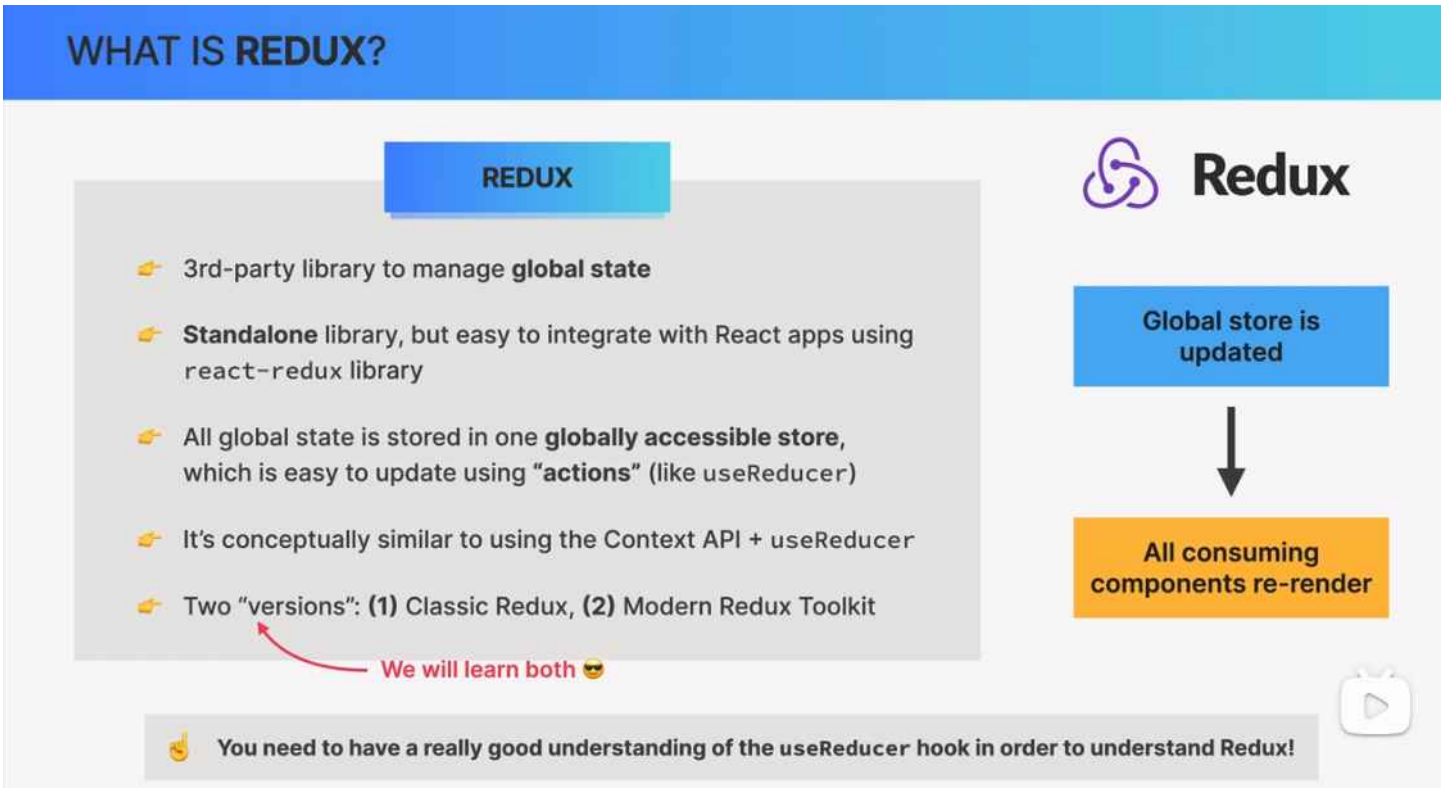
- `useEffect` 的要求







- `redux`：需要安装 `redux` 和 `react-redux`



- Classic redux
  - 传入一个reducer函数

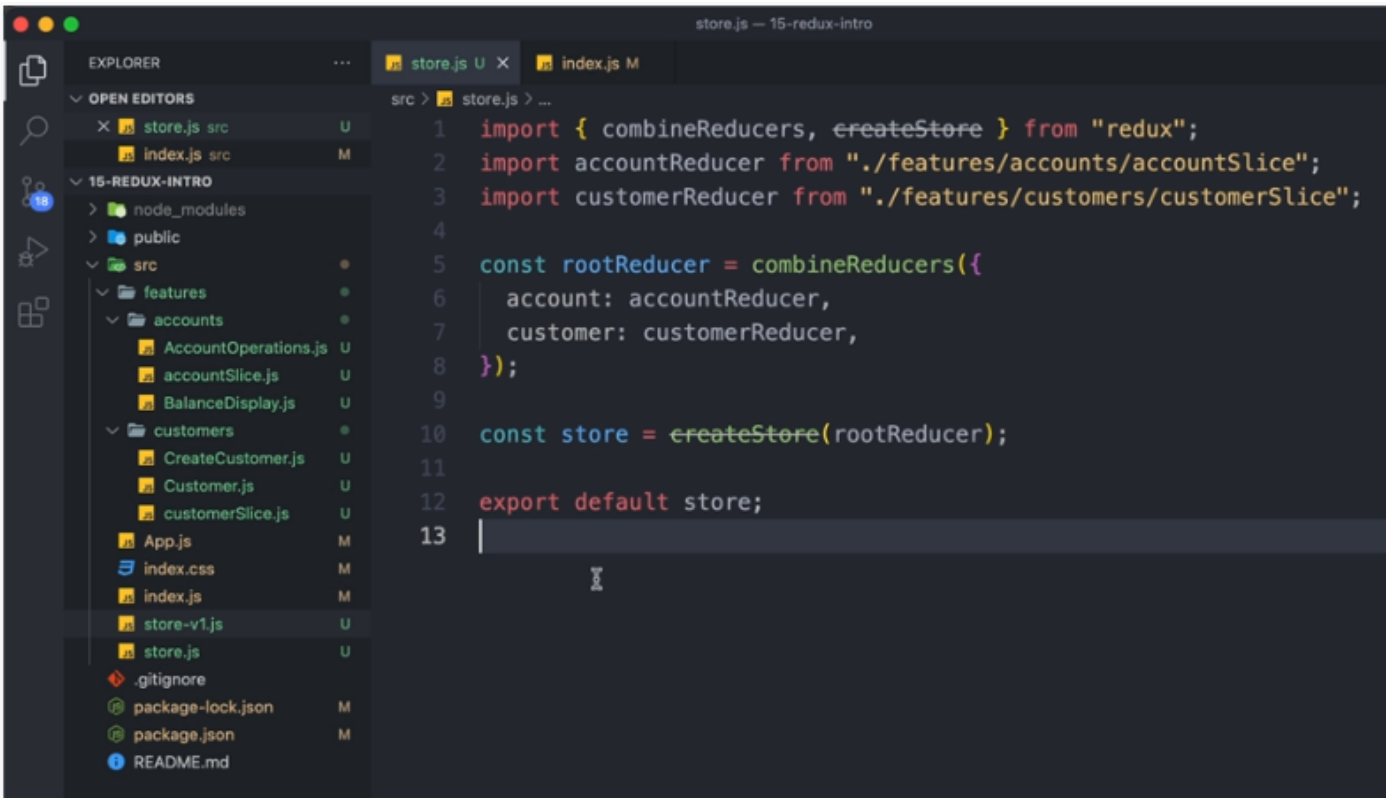
```
const store = createStore(reducer);

store.dispatch({ type: "account/deposit", payload: 500 });
store.dispatch({ type: "account/withdraw", payload: 200 });
console.log(store.getState());

store.dispatch({
  type: "account/requestLoan",
  payload: { amount: 1000, purpose: "Buy a car" },
});
console.log(store.getState());

store.dispatch({ type: "account/payLoan" });
console.log(store.getState());
```

```
function requestLoan(amount, purpose) {
  return {
    type: "account/requestLoan",
    payload: { amount, purpose },
  };
}

function payLoan() {
  return { type: "account/payLoan" };
}

store.dispatch(deposit(500));
store.dispatch(withdraw(200));
console.log(store.getState());

store.dispatch(requestLoan(1000, "Buy a cheap car"));
console.log(store.getState());
store.dispatch(payLoan());
console.log(store.getState());
```

有多个reducer的时候

```
const rootReducer = combineReducers({
  account: accountReducer,
  customer: customerReducer,
});

const store = createStore(rootReducer);
```

- 项目架构，不同部分开一个feature，其中有一个 XXXSlice，slice中放reducer



- React-redux，使用provider

```
import React from "react";
import ReactDOM from "react-dom/client";
import { Provider } from "react-redux";
import "./index.css";
import App from "./App";

import store from "./store";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

useSelector可以获得状态，useDispatch可以获得reducer

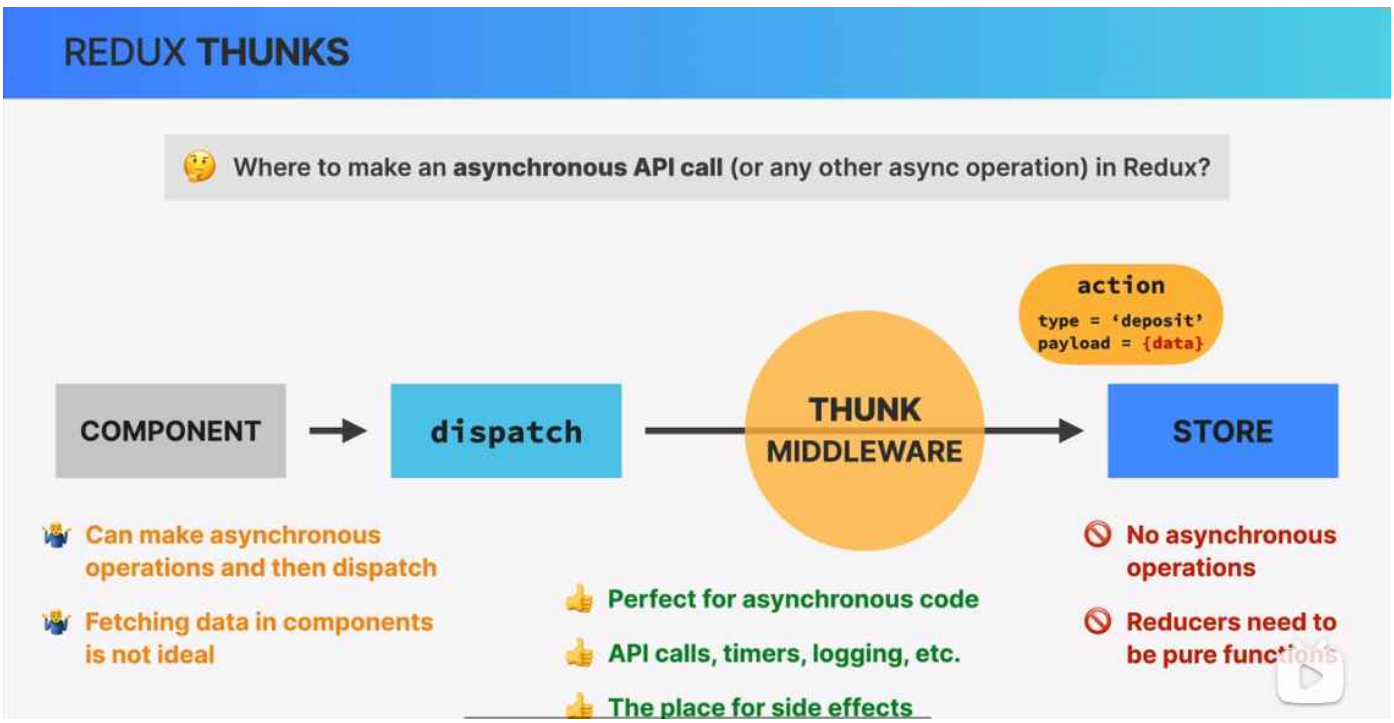
```
const dispatch = useDispatch();

function handleClick() {
  if (!fullName || !nationalId) return;
  dispatch(createCustomer(fullName, nationalId));
}
```

```
function Customer() {
  const customer = useSelector((store) => store.customer.
  fullName);

  return <h2>👋 Welcome, {customer}</h2>;
}
```

旧版中state和props链接的方法

```
import { connect } from "react-redux";

function formatCurrency(value) {
  return new Intl.NumberFormat("en", {
    style: "currency",
    currency: "USD",
  }).format(value);
}

function BalanceDisplay({ balance }) {
  return <div className="balance">{formatCurrency(balance)}</
  div>;
}

function mapStateToProps(state) {
  return {
    balance: state.account.balance,
  };
}

export default connect(mapStateToProps)(BalanceDisplay);
```

中间件redux thunk，负责处理异步逻辑，因为reducer中只能作pure function。安装 redux-thunk，使用方法在原来返回object的地方改成返回一个async函数




```
const store = createStore(rootReducer, applyMiddleware
(thunk));
```

```
export function deposit(amount, currency) {
  if (currency === "USD") return { type: "account/
  deposit", payload: amount };

  return async function (dispatch, getState) {
    const res = await fetch(
      `https://api.frankfurter.app/latest?amount=$
      {amount}&from=${currency}&to=USD`
    );
    const data = await res.json();
    const converted = data.rates.USD;

    dispatch({ type: "account/deposit", payload:
    converted });
  };
}
```

- Npm i redux-devtools-extension，

```
import { applyMiddleware, combineReducers, createStore }
from "redux";
import thunk from "redux-thunk";
import { composeWithDevTools } from
"redux-devtools-extension";

import accountReducer from "./features/accounts/
accountSlice";
import customerReducer from "./features/customers/
customerSlice";

const rootReducer = combineReducers({
  account: accountReducer,
  customer: customerReducer,
});

const store = createStore(
  rootReducer,
  composeWithDevTools(applyMiddleware(thunk))
);

export default store;
```

- Redux toolkit
  - Npm i @reduxjs/toolkit

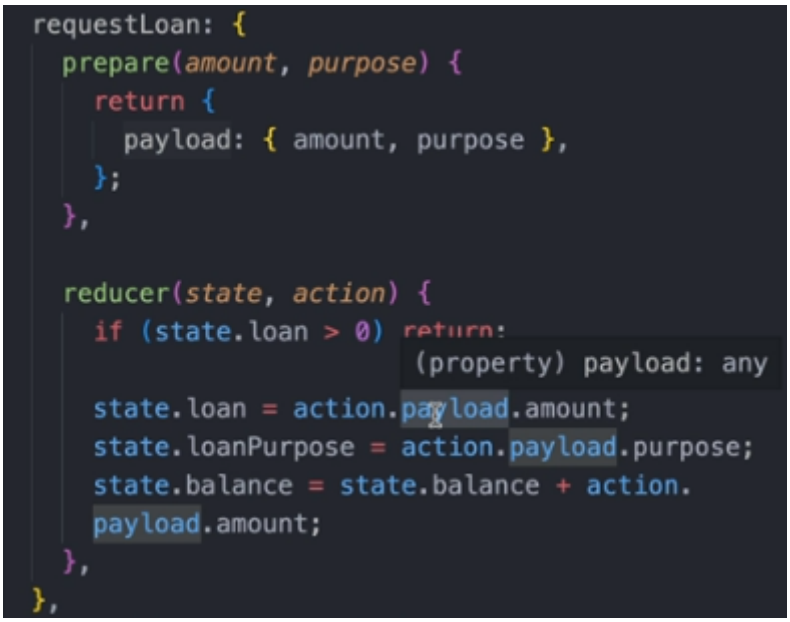
```
import { configureStore } from "@reduxjs/toolkit";

import accountReducer from "./features/accounts/
accountSlice";
import customerReducer from "./features/customers/
customerSlice";

const store = configureStore({
  reducer: {
    account: accountReducer,
    customer: customerReducer,
  },
});

export default store;
```

- 最大的好处是 immutable -> mutable 逻辑。使用 createSlice

```
const accountSlice = createSlice({
  name: "account",
  initialState,
  reducers: {
    deposit(state, action) {
      state.balance += action.payload;
    },
    withdraw(state, action) {
      state.balance -= action.payload;
    },
    requestLoan(state, action) {
      if (state.loan > 0) return;

      state.loan = action.payload.amount;
      state.loanPurpose = action.payload.purpose;
      state.balance = state.balance + action.
      payload.amount;
    },
    payLoan(state, action) {
      state.loan = 0;
      state.loanPurpose = "";
      state.balance -= state.loan;
    },
```

```
export const { deposit, withdraw, requestLoan,
payLoan } = accountSlice.actions;

export default accountSlice.reducer;
```

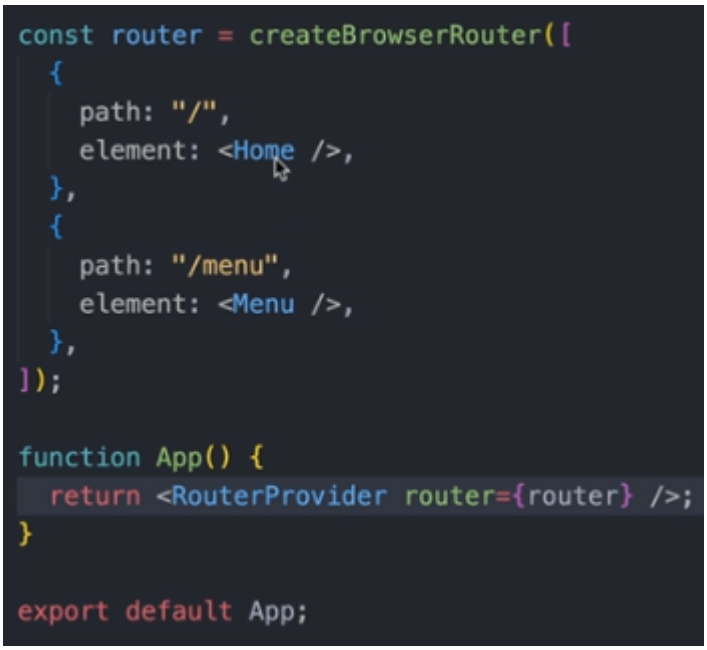
如果想要接收不止一个参数，那么：



如果要使用thunk可以直接用上面的方法，不导出deposit函数

工程

- Router的新定义方法：



可以在外层包一个组件

```
const router = createBrowserRouter([
  {
    element: <AppLayout />,
    children: [
      {
        path: "/",
        element: <Home />,
      },
      {
        path: "/menu",
        element: <Menu />,
      },
      { path: "/cart", element: <Cart /> },
      { path: "/order/new", element: <CreateOrder /> },
      { path: "/order/:orderId", element: <Order /> },
    ],
  },
]);
```

- DataLoader
  - 在一个页面中定义一个loader函数

```
export async function loader() {
  const menu = await getMenu();
  return menu;
}

export default Menu;
```

  - 在route中加上loader，这里是 `import {loader as menuloader}`

```
  {
    path: "/menu",
    element: <Menu />,
    loader: menuLoader,
  },
```

  - 最后使用这个函数

```
function Menu() {
  const menu = useLoaderData();
  console.log(menu);

  return <h1>Menu</h1>;
}
```

  - 我们需要知道是否在加载数据，我们要使用 `useNavigation` 钩子（不是 `useNavigate`）

```
const navigation = useNavigation();
const isLoading = navigation.state === "loading";
```

- 路由中可以加 ErrorElement，必须放在父路由中

```
const router = createBrowserRouter([
  {
    element: <AppLayout />,
    errorElement: <Error />,
```

```
import { useNavigate, useRouteError } from
"react-router-dom";

function Error() {
  const navigate = useNavigate();
  const error = useRouteError();
  console.log(error);

  return (
    <div>
      <h1>Something went wrong 😢</h1>
      <p>{error.data}</p>
      <button onClick={() => navigate(-1)}>&larr; Go
      back</button>
    </div>
  );
}

export default Error;
```

- 对于loader，url参数可以直接拿

```
export async function loader({ params }) {
  const order = await getOrder(params.orderId);
  return order;
}

export default Order;
```

- 给Form组件加post，首先将 `form` 变成 `From` ，然后写一个action函数，最后在route中写action

```
export async function action({ request }) {
  const formData = await request.formData();
  const data = Object.fromEntries(formData);

  const order = {
    ...data,
    cart: JSON.parse(data.cart),
    priority: data.priority === "on",
  };

  const newOrder = await createOrder(order);

  return redirect(`/order/${newOrder.id}`);
}
```

```
{
  path: "/order/new",
  element: <CreateOrder />,
  action: createOrderAction,
},
```

- TailwindCSS
  - 根据doc安装，然后安装 tailwind prettier（github上）
  - 响应式设计：下图表示从sm开始（width: 640px）padding x 为6px，tailwind是移动第一的设计。这些断点是可以修改的

```
<header className="bg-yellow-500 px-4 py-3 uppercase sm:px-6">
  <Link to="/" className="tracking-widest">
    Fast React Pizza Co.
```

  - Grid

```
<div className="grid h-screen grid-rows-[auto_1fr_auto] bg-red-500">
```

  - overflow-scroll，就是当超出内容的时候可以用滚轮查看
  - 复用，更好的方法是新建一个类

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer components {
  .input {
    @apply w-full rounded-full border border-stone-200
    px-4 py-2 text-sm transition-all duration-300
    placeholder:text-stone-400 focus:outline-none
    focus:ring focus:ring-yellow-400 md:px-6 md:py-3;
  }
}
```

  - inset-0：让上下左右都设置成0，占满整个界面
  - 在css中使用tailwind的类

```
--c: no-repeat linear-gradient(theme(colors.stone.
800) 0 0);
```

- createAsyncThunk

```
export const fetchAddress = createAsyncThunk(
  'user/fetchAddress',
  async function () {
    // 1) We get the user's geolocation position
    const positionObj = await getPosition();
    const position = {
      latitude: positionObj.coords.latitude,
      longitude: positionObj.coords.longitude,
    };

    // 2) Then we use a reverse geocoding API to get a
    description of the user's address, so we can
    display it the order form, so that the user can
    correct it if wrong
    const addressObj = await getAddress(position);
    const address = `${addressObj?.locality}, $
    {addressObj?.city} ${addressObj?.postcode}, $
    {addressObj?.countryName}`;
```

```
const initialState = {
  username: '',
  status: 'idle',
  position: {},
  address: '',
  error: '',
};
```

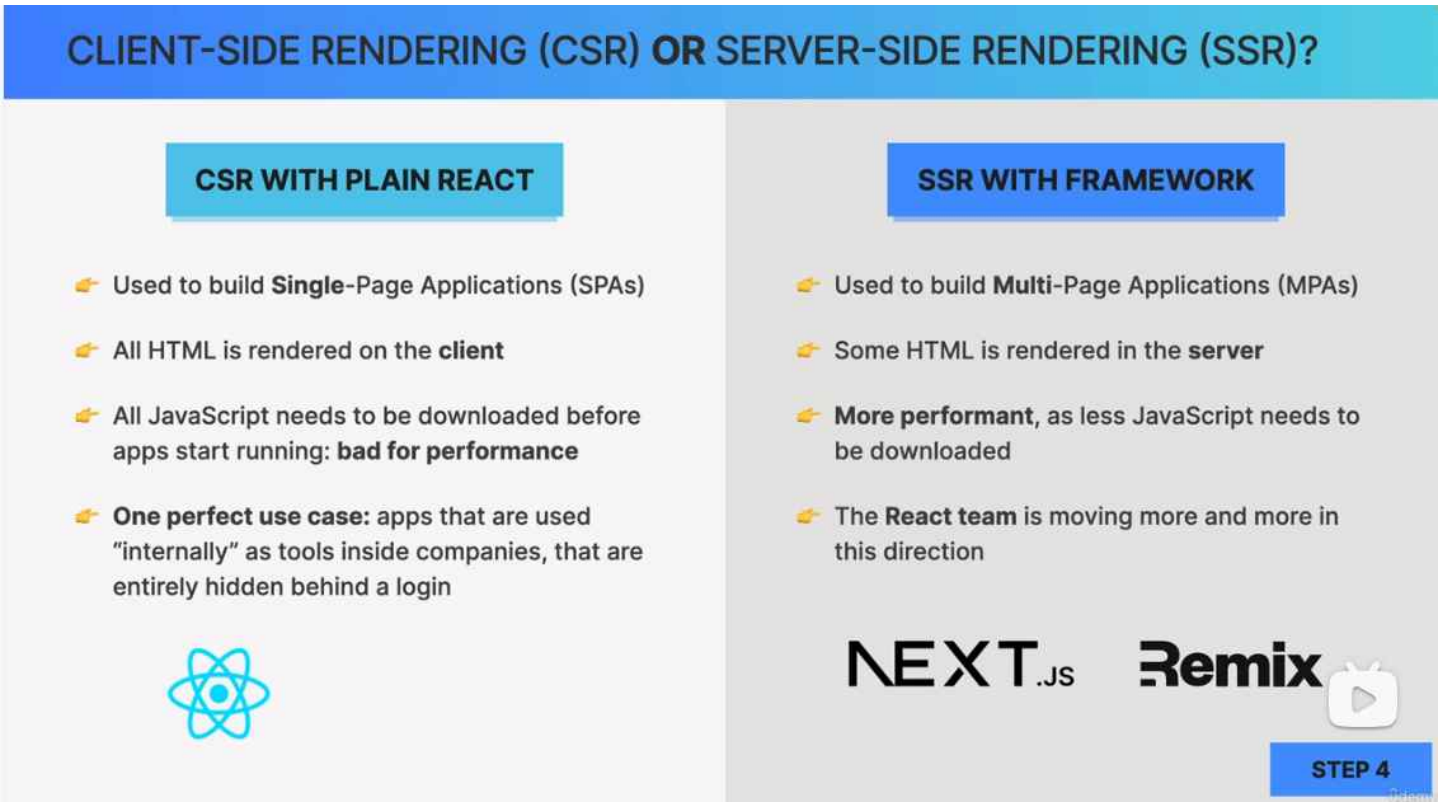
```
name: 'user',
initialState,
reducers: {
  updateName(state, action) {
    state.username = action.payload;
  },
},
extraReducers: (builder) =>
  builder
    .addCase(fetchAddress.pending, (state, action) =>
    {
      state.status = 'loading';
    })
    .addCase(fetchAddress.fulfilled, (state, action)
    => {
      state.position = action.payload.position;
      state.address = action.payload.address;
      state.state = 'idle';
    })
    .addCase(fetchAddress.rejected, (state, action)
    => {
      state.status = 'error';
      state.error = action.error.message;
    }),
```

- fetcher获取数据的另一种方法

```
const fetcher = useFetcher();

useEffect(
  function () {
    if (!fetcher.data && fetcher.state === 'idle')
    fetcher.load('/menu');
  },
  [fetcher]
);
```

```
function UpdateOrder({ order }) {
  const fetcher = useFetcher();

  return (
    <fetcher.Form method="PATCH" className="text-right">
      <Button type="primary">Make priority</Button>
    </fetcher.Form>
  );
}

export default UpdateOrder;

export async function action({ request, params }) {
  console.log('update');
  return null;
}
```

- SSR



- Styled component，需要安装scode的插件

```
import styled from "styled-components";

const H1 = styled.h1`
  font-size: 30px;
  font-weight: 600;
`;

function App() {
  return (
    <div>
      <H1>The Wild Oasis</H1>
    </div>
  );
}

export default App;
```

```jsx
function App() {
  return (
    <>
      <GlobalStyles />
      <StyledApp>
        <H1>The Wild Oasis</H1>
        <Button onClick={() => alert("Check in")}>Check
        in</Button>
        <Button onClick={() => alert("Check out")}
        >Check out</Button>

        <Input type="number" placeholder="Number of
        guests" />
        <Input type="number" placeholder="Number of
        guests" />
      </StyledApp>
    </>
  );
}
```

```jsx
const test = css`
  text-align: center;
  ${10 > 5 && "background-color: yellow"}
`;

const Heading = styled.h1`
  font-size: 20px;
  font-weight: 600;
  ${test}
`;

export default Heading;
```

通过props来设置不同的css，为了SEO建议使用as

```
const Heading = styled.h1`
  ${(props) =>
    props.type === "h1" &&
    css`
      font-size: 3rem;
      font-weight: 600;
    `}

  ${(props) =>
    props.type === "h2" &&
    css`
      font-size: 2rem;
      font-weight: 600;
    `}

    ${(props) =>
    props.type === "h3" &&
    css`
      font-size: 2rem;
      font-weight: 500;
    `}
```

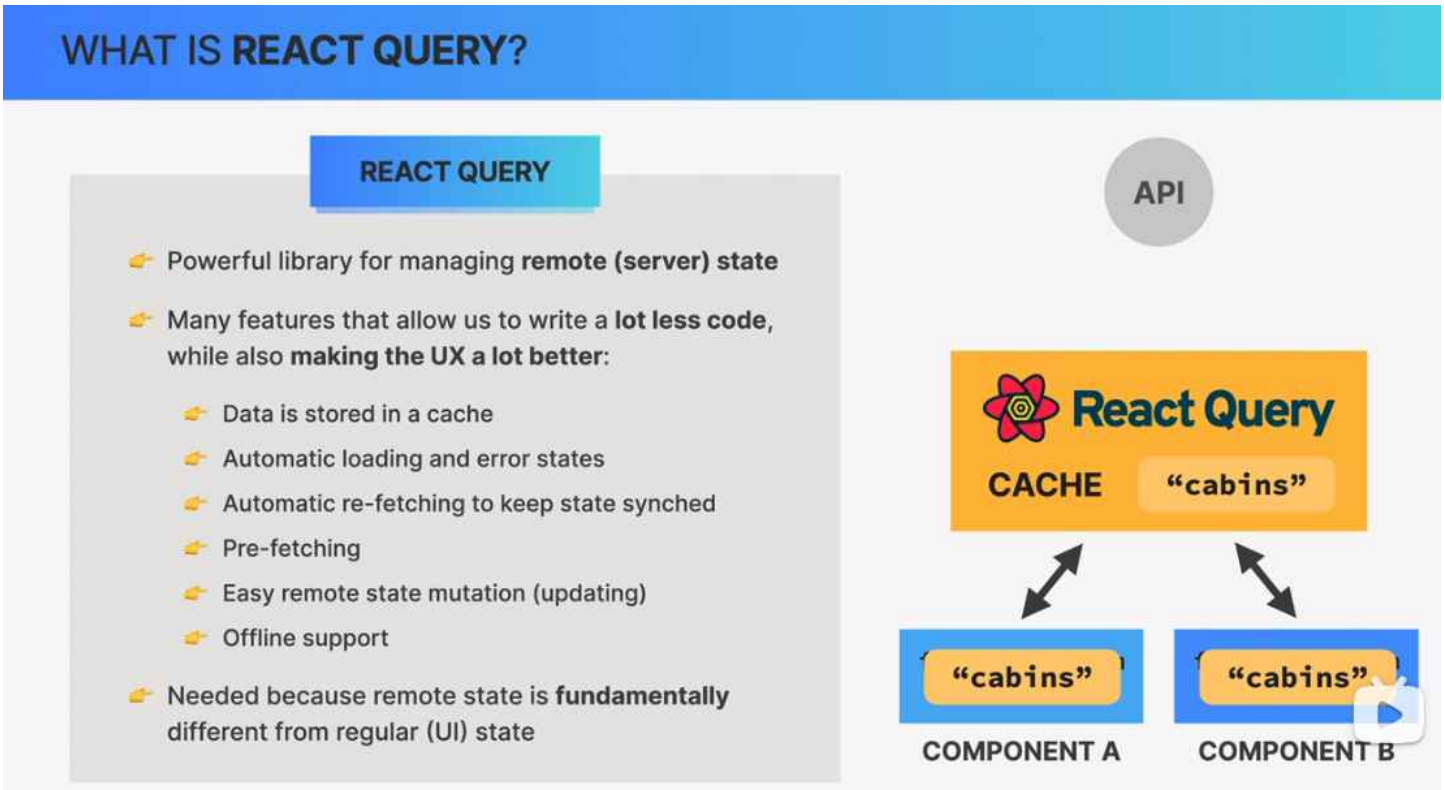
```
function App() {
  return (
    <>
      <GlobalStyles />
      <StyledApp>
        <Heading as="h1">The Wild Oasis</Heading>

        <Heading as="h2">Check in and out</Heading>
        <Button onClick={() => alert("Check in")}>Check
        in</Button>
        <Button onClick={() => alert("Check out")}
        >Check out</Button>

        <Heading as="h3">Form</Heading>
        <Input type="number" placeholder="Number of
        guests" />
        <Input type="number" placeholder="Number of
        guests" />
      </StyledApp>
    </>
  );
```

可以设置默认prop

```
Row.defaultProps = {
  type: "vertical",
};

export default Row;
```

```
const StyledNavLink = styled(NavLink)`
  &:link,
  &:visited {
    display: flex;
    align-items: center;
    gap: 1.2rem;

    color: var(--color-grey-600);
    font-size: 1.6rem;
    font-weight: 500;
    padding: 1.2rem 2.4rem;
    transition: all 0.3s;
  }

  /* This works because react-router places the active
  class on the active NavLink */
  &:hover,
  &:active,
  &.active:link,
  &.active:visited {
    color: var(--color-grey-800);
    background-color: var(--color-grey-50);
```

- supabase
- React Query




```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      // staleTime: 60 * 1000,
      staleTime: 0,
    },
  },
});

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <ReactQueryDevtools initialIsOpen={false} />
```

修改强制刷新cache：

```
const { isLoading, mutate } = useMutation({
  mutationFn: (id) => deleteCabin(id),
});

return (
  <TableRow role="row">
    <Img src={image} />
    <Cabin>{name}</Cabin>
    <div>Fits up tp {maxCapacity} guests</div>
    <Price>{formatCurrency(regularPrice)}</Price>
    <Discount>{formatCurrency(discount)}</Discount>
    <button onClick={() => mutate()}>Delete</button>
  </TableRow>
);
```

```
const queryClient = useQueryClient();

const { isLoading: isDeleting, mutate } = useMutation
({
  mutationFn: deleteCabin,
  onSuccess: () => {
    queryClient.invalidateQueries({
      queryKey: ["cabins"],
    });
  },
});
```

- react-hot-toast
- react-hook-form

```
function CreateCabinForm() {
  const { register, handleSubmit } = useForm();

  function onSubmit(data) {
    console.log(data);
  }

  return (
    <Form onSubmit={handleSubmit(onSubmit)}>
      <FormRow>
        <Label htmlFor="name">Cabin name</Label>
        <Input type="text" id="name" {...register
        ("name")} />
      </FormRow>

      <FormRow>
        <Label htmlFor="maxCapacity">Maximum capacity</
        Label>
        <Input type="number" id="maxCapacity" {...
        register("maxCapacity")} />
```

```
<Textarea
  type="number"
  id="description"
  defaultValue=""
  {...register("description", {
    required: "This field is required",
  })}
/>
```

```
{...register("discount", {
  required: "This field is required",
  validate: (value) =>
    value <= getValues().regularPrice ||
    "Discount should be less than regular
    price",
})}
```

- Render props pattern

组件获得一个render函数，然后来展示内容

```
return (
  <div className="list-container">
    <div className="heading">
      <h2>{title}</h2>
      <button onClick={toggleOpen}>
        {isOpen ? <span>&or;</span> : <span>&and;</span>}
      </button>
    </div>
    {isOpen && <ul className="list">{displayItems.map(render)}</u
    <button onClick={() => setIsCollapsed((isCollapsed) => !isCol
      {isCollapsed ? `Show all ${items.length}` : "Show less"}
    </button>
  </div>
);
```

```
<List
  title="Companies"
  items={companies}
  render={(company) => (
    <CompanyItem
      key={company.companyName}
      company={company}
      defaultVisibility={false}
    />
  )}
/>
```

- Higher-Order Component (HOC)，接受一个组件然后返回一个加强功能的组件

```
export default function withToggles(WrappedComponent) {
  return function List(props) {
    const [isOpen, setIsOpen] = useState(true);
    const [isCollapsed, setIsCollapsed] = useState(false);

    const displayItems = isCollapsed ? props.items.slice(0, 3) : pr

    function toggleOpen() {
      setIsOpen((isOpen) => !isOpen);
      setIsCollapsed(false);
    }

    return (
      <div className="list-container">
        <div className="heading">
          <h2>{props.title}</h2>
          <button onClick={toggleOpen}>
            {isOpen ? <span>&or;</span> : <span>&and;</span>}
          </button>
        </div>
        {isOpen && <WrappedComponent {...props} items={displayItems

        <button onClick={() => setIsCollapsed((isCollapsed) => !isC
```

- Compound Component Pattern。下图是两种方法的对比

```jsx
<Counter
  iconIncrease="+"
  iconDecrease="-"
  label="My NOT so flexible counter"
  hideLabel={false}
  hideIncrease={false}
  hideDecrease={false}
/>

<Counter>
  <Counter.Decrease icon="-" />
  <Counter.Count />
  <Counter.Increase icon="+" />
</Counter>
```

```jsx
import { createContext, useContext, useState } from "react";

// 1. Create a context
const CounterContext = createContext();

// 2. Create parent component
function Counter({ children }) {
  const [count, setCount] = useState(0);
  const increase = () => setCount((c) => c + 1);
  const decrease = () => setCount((c) => c - 1);

  return (
    <CounterContext.Provider value={{ count, increase, decrease
      <span>{children}</span>
    </CounterContext.Provider>
  );
}

// 3. Create child components to help implementing the common ta
function Count() {
  const { count } = useContext(CounterContext);
  return <span>{count}</span>;
}

function Label({ children }) {
  return <span>{children}</span>;
```

```
function Label({ children }) {
  return <span>{children}</span>;
}

function Increase({ icon }) {
  const { increase } = useContext(CounterContext);
  return <button onClick={increase}>{icon}</button>;
}

function Decrease({ icon }) {
  const { decrease } = useContext(CounterContext);
  return <button onClick={decrease}>{icon}</button>;
}
// 4. Add child components as proeprties to parent component
Counter.Count = Count;
Counter.Label = Label;
Counter.Increase = Increase;
Counter.Decrease = Decrease;

export default Counter;
```

- CreatePortal，插入dom中任意位置，但是在component tree中位置不变，适用于在所有组件之上的内容

```
function Modal({ children, onClose }) {
  return createPortal(
    <Overlay>
      <StyledModal>
        <Button onClick={onClose}>
          <HiXMark />
        </Button>

        <div>{children}</div>
      </StyledModal>
    </Overlay>,
    document.body
  );
}
```

- 点击窗口外关闭：注意多一个参数

```
useEffect(
  function () {
    function handleClick(e) {
      if (ref.current && !ref.current.contains(e.
      target)) {
        console.log("Click outside");
        close();
      }
    }

    document.addEventListener("click", handleClick,
    true);

    return () => document.removeEventListener
    ("click", handleClick, true);
  },
  [close]
);
```

```
export function useOutsideClick(handler,
listenCapturing = true) {
  const ref = useRef();

  useEffect(
    function () {
      function handleClick(e) {
        if (ref.current && !ref.current.contains(e.
        target)) {
          console.log("Click outside");
          handler();
        }
      }

      document.addEventListener("click", handleClick,
      true);

      return () => document.removeEventListener
      ("click", handleClick, true);
    },
    [handler]
  );
}
```

看到 P39不想看了……

随记

- VSCode插件：Quokka