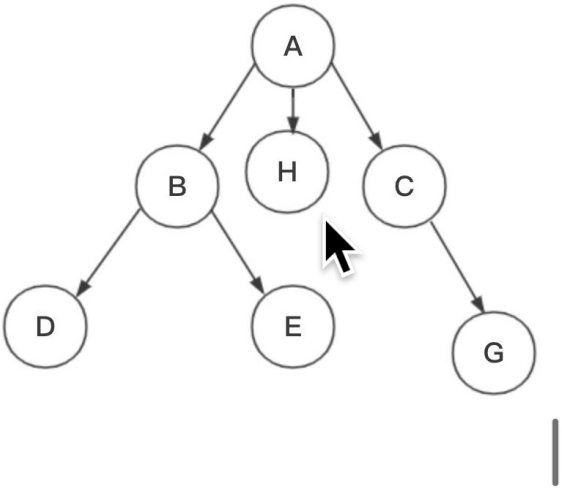
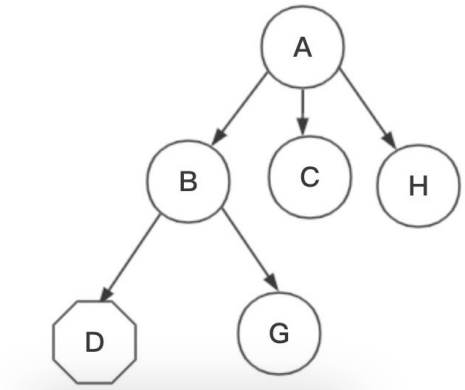
2-10 React 组件的常见性能优化

1. 复用组件

啃老本。

在协调阶段，组件复用的前提是必须同时满足三个条件：同一层级下、同一类型、同一个 key 值。**所以我们要尽量保证这三者的稳定性。**





常见错误：key=Math.random()

常见不规范写法：key=index

key 值标记了节点在当前层级下的唯一性，因此我们尽量不要取值 index，因为在同一层级下，多个循环的 index 容易重复。并且如果涉及节点的增加、删除、移动，那么 key 的稳定性将会被破坏，节点就会出现混乱现象。

2. 避免组件不必要的重新 render

组件重新 render 会导致组件进入协调，协调的核心就是我们常说的 vdom diff，所以协调本身就是比较耗时的算法，因此如果能够减少协调，复用旧的 fiber 节点，那么肯定会加快渲染完成的速度。组件如果没有进入协调阶段，我们称为进入 bailout 阶段，意思就是这层组件退出更新。

让组件进入 bailout 阶段，有以下方法：

`memo`

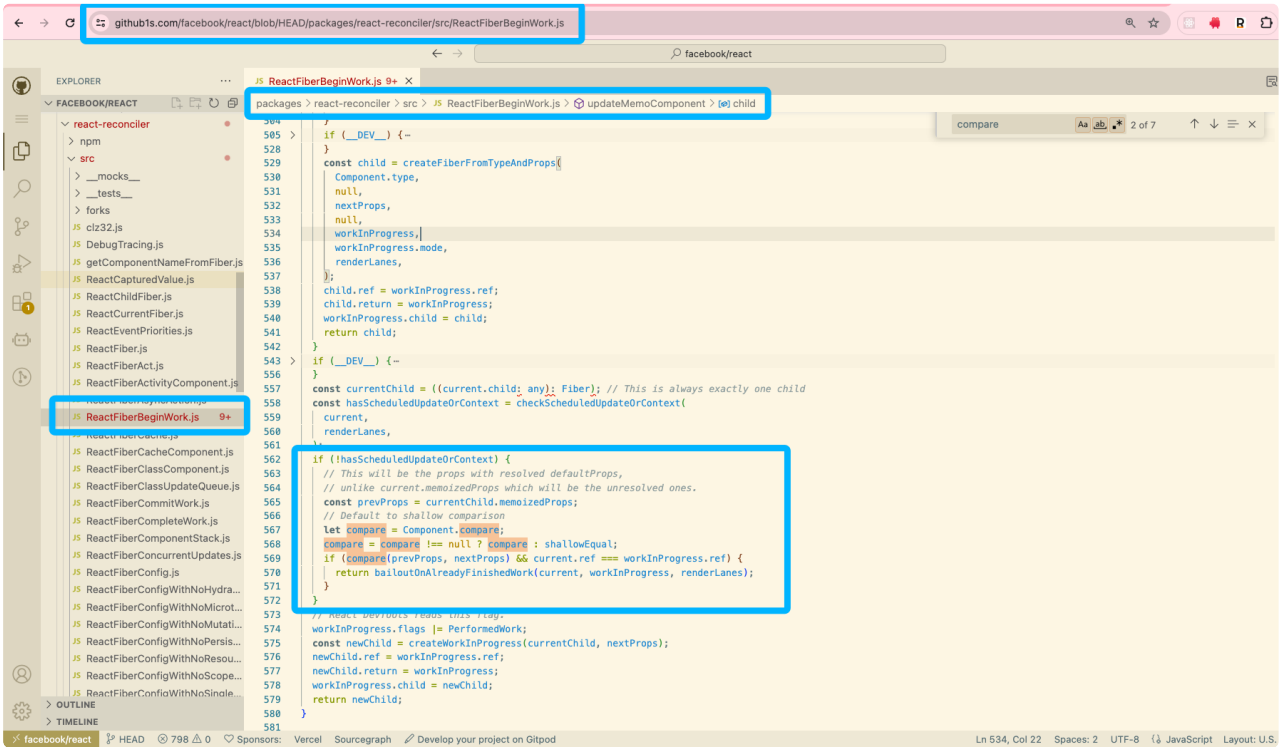
`memo` 允许你的组件在 props 没有改变的情况下跳过重新渲染。

JavaScript

```
const MemoizedComponent = memo(SomeComponent, arePropsEqual?)
```

这里的 arePropsEqual 是个函数，用户可以自定义，如果没有定义，默认使用浅比较，比较组件更新前后的 props 是否相同，如果相同，则进入 bailout 阶段。

源码地址：https://github1s.com/facebook/react/blob/HEAD/packages/react-reconciler/src/ReactFiberBeginWork.js



shouldComponentUpdate

React 可以调用它来确定是否可以跳过重新渲染。

JavaScript

```
shouldComponentUpdate(nextProps, nextState, nextContext)
```

PureComponent

`PureComponent` 同 `Component`，但是前者会浅比较 props 和 state 以及减少错过必要更新的概率。

React 源码中的浅比较函数

JavaScript

```
function shallowEqual(objA: mixed, objB: mixed): boolean {
  if (is(objA, objB)) {
    return true;
  }

  if (
    typeof objA !== 'object' ||
    objA === null ||
    typeof objB !== 'object' ||
    objB === null
  ) {
    return false;
  }

  const keysA = Object.keys(objA);
  const keysB = Object.keys(objB);

  if (keysA.length !== keysB.length) {
    return false;
  }

  // Test for A's keys different from B.
  for (let i = 0; i < keysA.length; i++) {
    const currentKey = keysA[i];
    if (
      !hasOwnProperty.call(objB, currentKey) ||
      !is(objA[currentKey], objB[currentKey])
    ) {
      return false;
    }
  }

  return true;
}
```

注意

Context

Context 本身就是一旦 Provider 传递的 value 变化，所有消费这个 value 的后代组件都要更新，因此应该尽量精简使用 Context。

Context 使用场景：*当祖先组件想要和后代组件快速通信。* *关于 Context 的使用细节，参考 Context 章节。*

3. 缓存策略/减少运算

`useMemo`

`useMemo` 是一个 React Hook，它在每次重新渲染的时候能够缓存计算的结果。

```JavaScript
const cachedValue = useMemo(calculateValue, dependencies)
```

useMemo 可以缓存参数，可以对比 useCallback 使用，useCallback(fn, deps) 相当于 useMemo(() => fn, deps)。

代码示例

```JavaScript
import * as React from "react";
import { useState, useMemo } from "react";

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

  const expensive = useMemo(() => {
    console.log("compute");
    let sum = 0;
    for (let i = 0; i < count; i++) {
      sum += i;
    }
    return sum;
```

```
    //只有count变化，这里才重新执行
  }, [count]);

  // const expensive = () => {
  //   console.log("compute");
  //   let sum = 0;
  //   for (let i = 0; i < count; i++) {
  //     sum += i;
  //   }
  //   return sum;
  //   //只有count变化，这里才重新执行
  // };

  return (
    <div>
      <h3>UseMemoPage</h3>
      <p>expensive:{expensive}</p>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>add</button>
      <input value={value} onChange={(event) => setValue(event.target.
    </div>
  );
}
```

useCallback

`useCallback` 是一个允许你在多次渲染中缓存函数的 React Hook。

JavaScript

```
const cachedFn = useCallback(fn, dependencies)
```

代码示例

JavaScript

```
import { useState, useCallback, memo, useEffect } from "react";

export default function UseCallbackPage(props) {
  const [count, setCount] = useState(0);
  const addClick = useCallback(() => {
```

```
    let sum = 0;
    for (let i = 0; i < count; i++) {
      sum += i;
    }
    return sum;
  }, [count]);

  // const addClick = () => {
  //   let sum = 0;
  //   for (let i = 0; i < count; i++) {
  //     sum += i;
  //   }
  //   return sum;
  // };
  const [value, setValue] = useState("");
  return (
    <div>
      <h3>UseCallbackPage</h3>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>add</button>
      <input value={value} onChange={(event) => setValue(event.target.
      <ChildMemo addClick={addClick} />
    </div>
  );
}

const ChildMemo = memo(function Child({ addClick }) {
  useEffect(() => {
    return () => {
      console.log("destroy"); //sy-log
    };
  }, []);
  console.log("Child"); //sy-log
  return (
    <div className="border">
      <button onClick={() => console.log(addClick())}>add</button>
    </div>
  );
});
```

代码示例

OptimizingPage

TypeScript

```
import {
  Component,
  PureComponent,
  useEffect,
  useState,
  memo,
  useMemo,
} from "react";

export default function OptimizingPage(props) {
  const [arr, setArr] = useState([0, 1, 2, 3]);

  return (
    <div className="border">
      <h3>OptimizingPage</h3>
      <button
        onClick={() => {
          setArr([...arr, arr.length]);
        }}
      >
        修改数组
      </button>

      {arr.map((item, index) => {
        return <ChildUseMemo key={"Child" + item} item={item} />;
      })}
    </div>
  );
}

function Child({ item }) {
  useEffect(() => {
    return () => {
      console.log("destroy"); //sy-log
    };
  }, []);
```

```
  console.log("Child", item); //sy-log
  return <div className="border">{item}</div>;
}

const ChildMemo = memo(Child, (prev, next) => {
  return prev.item === next.item;
});

class ChildShouldComponentUpdate extends Component {
  shouldComponentUpdate(nextProps) {
    return this.props.item !== nextProps.item;
  }
  render() {
    console.log("ChildComponent", this.props.item); //sy-log
    return (
      <div className="border">
        <p>{this.props.item}</p>
      </div>
    );
  }
}

class ChildPureComponent extends PureComponent {
  render() {
    console.log("ChildPureComponent"); //sy-log
    return (
      <div className="border">
        <p>{this.props.item}</p>
      </div>
    );
  }
}

function ChildUseMemo({ item }) {
  return useMemo(() => <Child item={item} />, []);
}
```